Contents

- ASP.NET Core documentation
- ASP.NET overview
- Tutorials
- Web Forms
  - Guidance
    - Getting Started
      - Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013
        - Getting Started with Web Forms and Visual Studio
        - Create the Project
        - Create the Data Access Layer
        - UI and Navigation
        - Display Data Items and Details
        - Shopping Cart
        - Checkout and Payment with PayPal
        - Membership and Administration
        - URL Routing
        - ASP.NET Error Handling
      - Creating a Basic Web Forms Page in Visual Studio 2013
      - Code Editing ASP.NET Web Forms in Visual Studio 2013
      - Using Page Inspector for Visual Studio 2012 in ASP.NET Web Forms
      - Visual Studio 2012 Hands On Labs
        - What's New in ASP.NET and Web Development in Visual Studio 2012
        - What's New in Web Forms in ASP.NET 4.5
        - Using Page Inspector in Visual Studio 2012
    - Working with Data
      - Model Binding and Web Forms in Visual Studio 2013
        - Retrieving and Displaying Data
        - Updating, Deleting, and Creating Data
        - Sorting, Paging, and Filtering Data

Integrating JQuery UI Datepicker
Using Query String Values to Filter Data
Adding Business Logic Layer
Security, Authentication, and Authorization
Create a secure ASP.NET Web Forms app with user registration, email confirmation and password reset (C#)
Create an ASP.NET Web Forms app with SMS Two-Factor Authentication (C#)
Performance
Using Asynchronous Methods in ASP.NET 4.5
Deployment
ASP.NET Web Deployment using Visual Studio
Introduction
Preparing for Database Deployment
Web.config File Transformations
Project Properties
Deploying to Test
Setting Folder Permissions
Deploying to Production
Deploying a Code Update
Deploying a Database Update
Command Line Deployment
Deploying Extra Files
Troubleshooting
ASP.NET 4 - Enterprise Deployment Introduction
Deploying Web Applications in Enterprise Scenarios using Visual Studio 2010
Enterprise Web Deployment: Scenario Overview
Application Lifecycle Management: From Development to Production
ASP.NET 4 - Enterprise Deployment Series 1
Web Deployment in the Enterprise
The Contact Manager Solution
Setting Up the Contact Manager Solution
Understanding the Project File
Understanding the Build Process

Building and Packaging Web Application Projects
Configuring Parameters for Web Package Deployment
Deploying Web Packages
Deploying Database Projects
Creating and Running a Deployment Command File
Manually Installing Web Packages
ASP.NET 4 - Enterprise Deployment Series 2 Configuring Servers
Configuring Server Environments for Web Deployment
Choosing the Right Approach to Web Deployment
Scenario: Configuring a Test Environment for Web Deployment
Scenario: Configuring a Staging Environment for Web Deployment
Scenario: Configuring a Production Environment for Web Deployment
Configuring a Web Server for Web Deploy Publishing (Remote Agent)
Configuring a Web Server for Web Deploy Publishing (Web Deploy Handler)
Configuring a Web Server for Web Deploy Publishing (Offline Deployment)
Configuring a Database Server for Web Deploy Publishing
Creating a Server Farm with the Web Farm Framework
Configuring Deployment Properties for a Target Environment
ASP.NET 4 - Enterprise Deployment Series 3 Configuring TFS
Configuring Team Foundation Server for Web Deployment
Creating a Team Project in TFS
Adding Content to Source Control
Configuring a TFS Build Server for Web Deployment
Creating a Build Definition That Supports Deployment
Deploying a Specific Build
Configuring Permissions for Team Build Deployment
ASP.NET 4 - Enterprise Deployment Series 4 Advanced Topics
Advanced Enterprise Web Deployment
Performing a "What If" Deployment
Customizing Database Deployments for Multiple Environments
Deploying Database Role Memberships to Test Environments
Deploying Membership Databases to Enterprise Environments

Excluding Files and Folders from Deployment
Taking Web Applications Offline with Web Deploy
Running Windows PowerShell Scripts from MSBuild Project Files
Troubleshooting the Packaging Process
Advanced
ASP.NET Web Forms Connection Resiliency and Command Interception
Older Versions - Getting Started
ASP.NET 4 - Tailspin Spyworks
Part 1: File-> New Project
Part 2: Data Access Layer
Part 3: Layout and Category Menu
Part 4: Listing Products
Part 5: Business Logic
Part 6: ASP.NET Membership
Part 7: Adding Features
Part 8: Final Pages, Exception Handling, and Conclusion
ASP.NET 4 - Getting Started with Entity Framework 4 and Web Forms
Getting Started with Entity Framework 4.0 Database First
Getting Started with Entity Framework 4.0 Database First - Part 2
Getting Started with Entity Framework 4.0 Database First - Part 3
Getting Started with Entity Framework 4.0 Database First - Part 4
Getting Started with Entity Framework 4.0 Database First - Part 5
Getting Started with Entity Framework 4.0 Database First - Part 6
Getting Started with Entity Framework 4.0 Database First - Part 7
Getting Started with Entity Framework 4.0 Database First - Part 8
ASP.NET 4 - Continuing with Entity Framework 4 and Web Forms
Entity Framework 4.0 and the ObjectDataSource Control, Part 1: Getting Started
Entity Framework 4.0 and the ObjectDataSource Control, Part 2: Adding a Business Logic Layer and Unit Tests
Entity Framework 4.0 and the ObjectDataSource Control, Part 3: Sorting and Filtering
Handling Concurrency with the Entity Framework 4.0 in ASP.NET 4
Maximizing Performance with the Entity Framework 4.0 in ASP.NET 4

What's New in the Entity Framework 4.0

ASP.NET 4 - Web Deployment with SQL Server Compact using Visual Studio

Visual Studio Web Deployment with SQL Server Compact - Introduction

Visual Studio Web Deployment with SQL Server Compact- Deploying SQL Server Compact Databases

Visual Studio Web Deployment with SQL Server Compact - Web.Config File Transformations

Visual Studio Web Deployment with SQL Server Compact - Configuring Project Properties

Visual Studio Web Deployment with SQL Server Compact - Deploying to IIS as a Test Environment

Visual Studio Web Deployment with SQL Server Compact - Setting Folder Permissions

Visual Studio Web Deployment with SQL Server Compact - Deploying to the Production Environment

Visual Studio Web Deployment with SQL Server Compact - Deploying a Code-Only Update

Visual Studio Web Deployment with SQL Server Compact - Deploying a Database Update

Visual Studio Web Deployment with SQL Server Compact - Migrating to SQL Server

Visual Studio Web Deployment with SQL Server Compact - Deploying a SQL Server Database Update

Visual Studio Web Deployment with SQL Server Compact - Troubleshooting

ASP.NET 3.5 - AJAX

Understanding Partial Page Updates with ASP.NET AJAX

Understanding ASP.NET AJAX UpdatePanel Triggers

Understanding ASP.NET AJAX Authentication and Profile Application Services

Understanding ASP.NET AJAX Localization

Understanding ASP.NET AJAX Web Services

Understanding ASP.NET AJAX Debugging Capabilities

ASP.NET 3.5 - Web Forms Master Pages

Creating a Site-Wide Layout Using Master Pages (C#)

Multiple ContentPlaceHolders and Default Content (C#)

Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page

(C#)
URLs in Master Pages (C#)
Control ID Naming in Content Pages (C#)
Interacting with the Master Page from the Content Page (C#)
Interacting with the Content Page from the Master Page (C#)
Master Pages and ASP.NET AJAX (C#)
Specifying the Master Page Programmatically (C#)
Nested Master Pages (C#)
Creating a Site-Wide Layout Using Master Pages (VB)
Multiple ContentPlaceHolders and Default Content (VB)
Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page (VB)
URLs in Master Pages (VB)
Control ID Naming in Content Pages (VB)
Interacting with the Master Page from the Content Page (VB)
Interacting with the Content Page from the Master Page (VB)
Master Pages and ASP.NET AJAX (VB)
Specifying the Master Page Programmatically (VB)
Nested Master Pages (VB)
Deploying Web Site Projects in Visual Studio 2008 and earlier
ASP.NET Hosting Options (C#)
Determining What Files Need to Be Deployed (C#)
Deploying Your Site Using an FTP Client (C#)
Deploying Your Site Using Visual Studio (C#)
Common Configuration Differences Between Development and Production (C#)
Core Differences Between IIS and the ASP.NET Development Server (C#)
Deploying a Database (C#)
Configuring the Production Web Application to Use the Production Database (C#)
Configuring a Website that Uses Application Services (C#)
Strategies for Database Development and Deployment (C#)
Displaying a Custom Error Page (C#)
Processing Unhandled Exceptions (C#)

Logging Error Details with ASP.NET Health Monitoring (C#)
Logging Error Details with ELMAH (C#)
Precompiling Your Website (C#)
Users and Roles On Production Website (C#)
ASP.NET Hosting Options (VB)
Determining What Files Need to Be Deployed (VB)
Deploying Your Site Using an FTP Client (VB)
Deploying Your Site Using Visual Studio (VB)
Common Configuration Differences Between Development and Production (VB)
Core Differences Between IIS and the ASP.NET Development Server (VB)
Deploying a Database (VB)
Configuring the Production Web Application to Use the Production Database (VB)
Configuring a Website that Uses Application Services (VB)
Strategies for Database Development and Deployment (VB)
Displaying a Custom Error Page (VB)
Processing Unhandled Exceptions (VB)
Logging Error Details with ASP.NET Health Monitoring (VB)
Logging Error Details with ELMAH (VB)
Precompiling Your Website (VB)
Users and Roles On Production Website (VB)
Older Versions - Ajax Control Toolkit
Getting Started
Get Started with the AJAX Control Toolkit (C#)
Using AJAX Control Toolkit Controls and Control Extenders (C#)
Creating a Custom AJAX Control Toolkit Control Extender (C#)
Get Started with the AJAX Control Toolkit (VB)
Using AJAX Control Toolkit Controls and Control Extenders (VB)
Creating a Custom AJAX Control Toolkit Control Extender (VB)
Accordion
Databinding to an Accordion (C#)
Dynamically Adding Accordion Pane (C#)
Databinding to an Accordion (VB)

Dynamically Adding Accordion Pane (VB)
Animation
Adding Animation to a Control (C#)
Executing Several Animations at Same Time (C#)
Executing Several Animations after Each Other (C#)
Animation Depending On a Condition (C#)
Picking One Animation Out Of a List (C#)
Animating in Response To User Interaction (C#)
Disabling Actions during Animation (C#)
Triggering an Animation in another Control (C#)
Modifying Animations From Server Side (C#)
Executing Animations Using Client-Side Code (C#)
Changing an Animation Using Client-Side Code (C#)
Animating an UpdatePanel Control (C#)
Dynamically Controlling UpdatePanel Animations (C#)
Adding Animation to a Control (VB)
Executing Several Animations at Same Time (VB)
Executing Several Animations after Each Other (VB)
Animation Depending On a Condition (VB)
Picking One Animation Out Of a List (VB)
Animating in Response To User Interaction (VB)
Disabling Actions during Animation (VB)
Triggering an Animation in another Control (VB)
Modifying Animations From Server Side (VB)
Executing Animations Using Client-Side Code (VB)
Changing an Animation Using Client-Side Code (VB)
Animating an UpdatePanel Control (VB)
Dynamically Controlling UpdatePanel Animations (VB)
CascadingDropdown
Filling a List Using CascadingDropDown (C#)
Using CascadingDropDown with a Database (C#)
Presetting List Entries with CascadingDropDown (C#)

Using Auto-Postback with CascadingDropDown (C#)
Filling a List Using CascadingDropDown (VB)
Using CascadingDropDown with a Database (VB)
Presetting List Entries with CascadingDropDown (VB)
Using Auto-Postback with CascadingDropDown (VB)
CollapsiblePanel
Collapsing and Expanding a Panel from JavaScript (C#)
Collapsing and Expanding a Panel from JavaScript (VB)
ColorPicker
ColorPicker Control Extender (C#)
ColorPicker Control Extender (VB)
ComboBox
ComboBox Control (C#)
ComboBox Control (VB)
ConfirmButton
Using a ConfirmButton In a Repeater (C#)
Using a ConfirmButton In a Repeater (VB)
DropShadow
Adjusting the Z-Index of a DropShadow (C#)
Manipulating DropShadow Properties from Client Code (C#)
Adjusting the Z-Index of a DropShadow (VB)
Manipulating DropShadow Properties from Client Code (VB)
DynamicPopulate
Dynamically Populating a Control (C#)
Dynamically Populating a Control Using JavaScript Code (C#)
Using DynamicPopulate with a User Control And JavaScript (C#)
Dynamically Populating a Control (VB)
Dynamically Populating a Control Using JavaScript Code (VB)
Using DynamicPopulate with a User Control And JavaScript (VB)
FilteredTextBox
Allowing Only Certain Characters in a Text Box (C#)
Allowing Only Certain Characters in a Text Box (VB)

HoverMenu
- Using HoverMenu with a Repeater Control (C#)
- Using HoverMenu with a Repeater Control (VB)

HTMLEditor
- HTML Editor Control (C#)
- HTML Editor Control (VB)

ModalPopup
- Launching a Modal Popup Window from Server Code (C#)
- Using ModalPopup with a Repeater Control (C#)
- Handling Postbacks from a ModalPopup (C#)
- Positioning a ModalPopup (C#)
- Launching a Modal Popup Window from Server Code (VB)
- Using ModalPopup with a Repeater Control (VB)
- Handling Postbacks from a ModalPopup (VB)
- Positioning a ModalPopup (VB)

MutuallyExclusiveCheckbox
- Creating Mutually Exclusive Checkboxes (C#)
- Creating Mutually Exclusive Checkboxes (VB)

NoBot
- Fighting Bots (C#)
- Fighting Bots (VB)

NumericUpDown
- Creating a Numeric Up/Down Control with a Web Service Backend (C#)
- Creating a Numeric Up/Down Control with a Web Service Backend (VB)

PasswordStrength
- Testing the Strength of a Password (C#)
- Testing the Strength of a Password (VB)

Popup
- Using Multiple Popup Controls (C#)
- Handling Postbacks from A Popup Control With an UpdatePanel (C#)
- Handling Postbacks from A Popup Control Without an UpdatePanel (C#)
- Using Multiple Popup Controls (VB)

- Handling Postbacks from A Popup Control With an UpdatePanel (VB)
- Handling Postbacks from A Popup Control Without an UpdatePanel (VB)
- Rating
  - Creating a Rating Control (C#)
  - Creating a Rating Control (VB)
- ReorderList
  - Using Postbacks with ReorderList (C#)
  - Drag and Drop via ReorderList (C#)
  - Using Postbacks with ReorderList (VB)
  - Drag and Drop via ReorderList (VB)
- Slider
  - Slider Control With Auto-Postback (C#)
  - Databinding the Slider Control (C#)
  - Slider Control With Auto-Postback (VB)
  - Databinding the Slider Control (VB)
- TextBoxWatermark
  - Using TextBoxWatermark in a FormView (C#)
  - Using TextBoxWatermark With Validation Controls (C#)
  - Using TextBoxWatermark in a FormView (VB)
  - Using TextBoxWatermark With Validation Controls (VB)

Older Versions - Data Access

- Introduction
  - Creating a Data Access Layer (C#)
  - Creating a Business Logic Layer (C#)
  - Master Pages and Site Navigation (C#)
  - Creating a Data Access Layer (VB)
  - Creating a Business Logic Layer (VB)
  - Master Pages and Site Navigation (VB)
- Basic Reporting
  - Displaying Data With the ObjectDataSource (C#)
  - Declarative Parameters (C#)
  - Programmatically Setting the ObjectDataSource's Parameter Values (C#)

Displaying Data With the ObjectDataSource (VB)
Declarative Parameters (VB)
Programmatically Setting the ObjectDataSource's Parameter Values (VB)
Master/Detail
Master/Detail Filtering With a DropDownList (C#)
Master/Detail Filtering With Two DropDownLists (C#)
Master/Detail Filtering Across Two Pages (C#)
Master/Detail Using a Selectable Master GridView with a Details DetailView (C#)
Master/Detail Filtering With a DropDownList (VB)
Master/Detail Filtering With Two DropDownLists (VB)
Master/Detail Filtering Across Two Pages (VB)
Master/Detail Using a Selectable Master GridView with a Details DetailView (VB)
Custom Formatting
Custom Formatting Based Upon Data (C#)
Using TemplateFields in the GridView Control (C#)
Using TemplateFields in the DetailsView Control (C#)
FormView's Templates (C#)
Displaying Summary Information in the GridView's Footer (C#)
Custom Formatting Based Upon Data (VB)
Using TemplateFields in the GridView Control (VB)
Using TemplateFields in the DetailsView Control (VB)
FormView's Templates (VB)
Displaying Summary Information in the GridView's Footer (VB)
Editing, Inserting, and Deleting Data
Overview of Inserting, Updating, and Deleting Data (C#)
Examining the Events Associated with Inserting, Updating, and Deleting (C#)
Handling BLL- and DAL-Level Exceptions in an ASP.NET Page (C#)
Adding Validation Controls to the Editing and Inserting Interfaces (C#)
Customizing the Data Modification Interface (C#)
Implementing Optimistic Concurrency (C#)
Adding Client-Side Confirmation When Deleting (C#)
Limiting Data Modification Functionality Based on the User (C#)

Overview of Inserting, Updating, and Deleting Data (VB)
Examining the Events Associated with Inserting, Updating, and Deleting (VB)
Handling BLL- and DAL-Level Exceptions in an ASP.NET Page (VB)
Adding Validation Controls to the Editing and Inserting Interfaces (VB)
Customizing the Data Modification Interface (VB)
Implementing Optimistic Concurrency (VB)
Adding Client-Side Confirmation When Deleting (VB)
Limiting Data Modification Functionality Based on the User (VB)
Paging and Sorting
Paging and Sorting Report Data (C#)
Efficiently Paging Through Large Amounts of Data (C#)
Sorting Custom Paged Data (C#)
Creating a Customized Sorting User Interface (C#)
Paging and Sorting Report Data (VB)
Efficiently Paging Through Large Amounts of Data (VB)
Sorting Custom Paged Data (VB)
Creating a Customized Sorting User Interface (VB)
Custom Button Actions
Adding and Responding to Buttons to a GridView (C#)
Adding and Responding to Buttons to a GridView (VB)
Displaying Data with the DataList and Repeater
Displaying Data with the DataList and Repeater Controls (C#)
Formatting the DataList and Repeater Based Upon Data (C#)
Showing Multiple Records per Row with the DataList Control (C#)
Nested Data Web Controls (C#)
Displaying Data with the DataList and Repeater Controls (VB)
Formatting the DataList and Repeater Based Upon Data (VB)
Showing Multiple Records per Row with the DataList Control (VB)
Nested Data Web Controls (VB)
Filtering Scenarios with the DataList and Repeater
Master/Detail Filtering With a DropDownList (C#)
Master/Detail Filtering Across Two Pages (C#)

Master/Detail Using a Bulleted List of Master Records with a Details DataList (C#)
Master/Detail Filtering With a DropDownList (VB)
Master/Detail Filtering Across Two Pages (VB)
Master/Detail Using a Bulleted List of Master Records with a Details DataList (VB)
Editing and Deleting Data Through the DataList
Overview of Editing and Deleting Data in the DataList (C#)
Performing Batch Updates (C#)
Handling BLL- and DAL-Level Exceptions (C#)
Adding Validation Controls to the DataList's Editing Interface (C#)
Customizing the DataList's Editing Interface (C#)
Overview of Editing and Deleting Data in the DataList (VB)
Performing Batch Updates (VB)
Handling BLL- and DAL-Level Exceptions (VB)
Adding Validation Controls to the DataList's Editing Interface (VB)
Customizing the DataList's Editing Interface (VB)
Paging and Sorting with the DataList and Repeater
Paging Report Data in a DataList or Repeater Control (C#)
Sorting Data in a DataList or Repeater Control (C#)
Paging Report Data in a DataList or Repeater Control (VB)
Sorting Data in a DataList or Repeater Control (VB)
Custom Button Actions with the DataList and Repeater
Custom Buttons in the DataList and Repeater (C#)
Custom Buttons in the DataList and Repeater (VB)
Accessing the Database Directly from an ASP.NET Page
Querying Data with the SqlDataSource Control (C#)
Using Parameterized Queries with the SqlDataSource (C#)
Inserting, Updating, and Deleting Data with the SqlDataSource (C#)
Implementing Optimistic Concurrency with the SqlDataSource (C#)
Querying Data with the SqlDataSource Control (VB)
Using Parameterized Queries with the SqlDataSource (VB)
Inserting, Updating, and Deleting Data with the SqlDataSource (VB)
Implementing Optimistic Concurrency with the SqlDataSource (VB)

Enhancing the GridView
- Adding a GridView Column of Radio Buttons (C#)
- Adding a GridView Column of Checkboxes (C#)
- Inserting a New Record from the GridView's Footer (C#)
- Adding a GridView Column of Radio Buttons (VB)
- Adding a GridView Column of Checkboxes (VB)
- Inserting a New Record from the GridView's Footer (VB)

Working with Binary Files
- Uploading Files (C#)
- Displaying Binary Data in the Data Web Controls (C#)
- Including a File Upload Option When Adding a New Record (C#)
- Updating and Deleting Existing Binary Data (C#)
- Uploading Files (VB)
- Displaying Binary Data in the Data Web Controls (VB)
- Including a File Upload Option When Adding a New Record (VB)
- Updating and Deleting Existing Binary Data (VB)

Caching Data
- Caching Data with the ObjectDataSource (C#)
- Caching Data in the Architecture (C#)
- Caching Data at Application Startup (C#)
- Using SQL Cache Dependencies (C#)
- Caching Data with the ObjectDataSource (VB)
- Caching Data in the Architecture (VB)
- Caching Data at Application Startup (VB)
- Using SQL Cache Dependencies (VB)

Database-Driven Site Maps
- Building a Custom Database-Driven Site Map Provider (C#)
- Building a Custom Database-Driven Site Map Provider (VB)

Working with Batched Data
- Wrapping Database Modifications within a Transaction (C#)
- Batch Updating (C#)
- Batch Deleting (C#)

Batch Inserting (C#)
Wrapping Database Modifications within a Transaction (VB)
Batch Updating (VB)
Batch Deleting (VB)
Batch Inserting (VB)
Advanced Data Access Scenarios
Creating New Stored Procedures for the Typed DataSet's TableAdapters (C#)
Using Existing Stored Procedures for the Typed DataSet's TableAdapters (C#)
Updating the TableAdapter to Use JOINs (C#)
Adding Additional DataTable Columns (C#)
Working with Computed Columns (C#)
Configuring the Data Access Layer's Connection- and Command-Level Settings (C#)
Protecting Connection Strings and Other Configuration Information (C#)
Debugging Stored Procedures (C#)
Creating Stored Procedures and User-Defined Functions with Managed Code (C#)
Creating New Stored Procedures for the Typed DataSet's TableAdapters (VB)
Using Existing Stored Procedures for the Typed DataSet's TableAdapters (VB)
Updating the TableAdapter to Use JOINs (VB)
Adding Additional DataTable Columns (VB)
Working with Computed Columns (VB)
Configuring the Data Access Layer's Connection- and Command-Level Settings (VB)
Protecting Connection Strings and Other Configuration Information (VB)
Debugging Stored Procedures (VB)
Creating Stored Procedures and User-Defined Functions with Managed Code (VB)
Older Versions - Security
ASP.NET 3.5 - Introduction to Security
Security Basics and ASP.NET Support (C#)
Overview of Forms Authentication (C#)
Forms Authentication Configuration and Advanced Topics (C#)

Security Basics and ASP.NET Support (VB)
Overview of Forms Authentication (VB)
Forms Authentication Configuration and Advanced Topics (VB)
ASP.NET 3.5 - Membership
Creating the Membership Schema in SQL Server (C#)
Creating User Accounts (C#)
Validating User Credentials Against the Membership User Store (C#)
User-Based Authorization (C#)
Storing Additional User Information (C#)
Creating the Membership Schema in SQL Server (VB)
Creating User Accounts (VB)
Validating User Credentials Against the Membership User Store (VB)
User-Based Authorization (VB)
Storing Additional User Information (VB)
ASP.NET 3.5 - Roles
Creating and Managing Roles (C#)
Assigning Roles to Users (C#)
Role-Based Authorization (C#)
Creating and Managing Roles (VB)
Assigning Roles to Users (VB)
Role-Based Authorization (VB)
ASP.NET 3.5 - Security Administration
Building an Interface to Select One User Account from Many (C#)
Recovering and Changing Passwords (C#)
Unlocking and Approving User Accounts (C#)
Building an Interface to Select One User Account from Many (VB)
Recovering and Changing Passwords (VB)
Unlocking and Approving User Accounts (VB)
Older Versions - ASP.NET 2.0
Improvements in Visual Studio 2005
ASP.NET 2.0 Page Model
Server Controls

- Master Pages
- Data Source Controls
- Data Bound Controls
- Caching
- Membership
- Configuration and Instrumentation
- Profiles, Themes, and Web Parts

Videos

How Do I?

- Learn the Tips and Tricks of Experts
- Create a Multi-Lingual Site with Localization
- Make use of Caching
- Make Greater use of Caching
- Utilize Web Parts and Personalization
- Customize my Site with Profiles and Themes
- Secure my Site using Membership and Roles
- Create a "Contact Us" Page
- Create a Full-Featured Customer Login Portal
- Use Master Pages and Site Navigation
- Create Data-Driven Web Sites
- Implement an HttpModule
- Set Up the SQL Membership Provider
- Implement URL Rewriting
- Utilize the CSS Friendly Control Adapters
- Handle Application Events using an HttpModule
- Use Cascading Style Sheets for Web Page Layout
- Create a Custom Membership Provider
- Create a Custom Profile Provider
- Implement Site Navigation in ASP.NET
- Use Validation Controls in ASP.NET
- Create a Custom User Control in ASP.NET
- Implement Health Monitoring for an ASP.NET Application

Implement Tracing in an ASP.NET Web Site

Event Handlers in ASP.NET Web Pages and Controls

Implement a Custom Health Monitoring Event

Use Control State to Persist Information for a Custom Web Server Control

Read, Write, and Delete Cookies in ASP.NET

Save and Load View State Information for a Custom Web Server Control

Create and Use Application Level Events in ASP.NET

Simple File Uploads in ASP.NET

Multiple File Uploads in ASP.NET 2

Multiple File Uploads in ASP.NET 1

File Uploads with an AJAX Style Interface

Work with Master Pages Declaratively and Programmatically

Handle Events in Master and Content Pages

Use the ObjectDataSource with a Data Access Class and Business Object

Create User Selectable Themes for a Web Site

Share Data Between Client JavaScript and Server Code Using a Hidden Field

Add JavaScript to ASP.NET Page

Add JavaScript Event Handlers to ASP.NET Server Controls

Create and Call a Simple Web Service in ASP.NET

Deploy a Web Site Copy Web Site Tool

Create and Use a Custom Configuration Section in an ASP.NET Configuration File

Configure an ASP.NET Web Site for a .NET Framework Version

Use the ASP.NET IIS Registration Tool to Configure ASP.NET

Extend and Customize an ASP.NET Server Control for a Specific Purpose

Use the Ability to Create Groups with the ListView Control for Different Data

Make HTML elements accessible from server side code

Use Roles to Segment Functionality in an ASP.NET Web Site

Work with Nested Master Pages to Create Standard Content Layouts

Detect Browser Capabilities in ASP.NET Web Pages

Map an ASP.NET Server Control to the Adaptor Used to Render It

Pass Information From One Page to Another Using a Query String

Use the aspnet_merge.exe Utility to Merge Assemblies

Implement a Cross Page PostBack in ASP.NET

Add Controls to an ASP.NET Web Page Programmatically

Use a Hidden Field to Store and Manipulate Client-Side Information

Use the Fixednames Option with the Compiler Utility for Maximum Flexibility

Implement Command Buttons for Individual Items

Use Skins with CSS for a Flexible and Maintainable ASP.NET Web Site

Determine Whether to Use a Web Site or a Web Application Project

Write Web Events to a SQL Server Database SqlWebEventProvider

Use ASP.NET to Send Email from a Web Site

Configure Email Notification for Health Monitoring on an ASP.NET Web Site

Use a Visual Studio 2008 Web Deployment Project

Send Templated Emails for Health Monitoring Events in ASP.NET

Send Email Asynchronously with ASP.NET

Embed an Image in an Email with ASP.NET

Implement Error Handling when Sending Email with ASP.NET

Create a Custom Provider for Logging Health Monitoring Events

Create a Reusable Component for Sending Email to a Distribution List

Precompile an ASP.NET Website

Use the FtpWebRequest Class to Create an FTP Connection to a FTP Server

Export Data to a Comma Delimited (CSV) File for an Application Like Excel

Use the Reponse.Filter Property to Replace HTML in an ASP.NET Page

Use OutputCache to Cache an ASP.NET Page

Cache Portions of an ASP.NET Page

Control the Caching of an ASP.NET Page Based Upon Custom Information

Cache an ASP.NET Page Based Upon Information in the HTTP Header

Synchronize Data in Secondary User Control(s) Based Upon User Selections

Use the ASP.NET Cache Object to Cache Application Information

Persist the State of a User Control During a Postback

Add Client Script Event Handlers Controls on an ASP.NET Page

Understand and Defend Against Script Injection Attacks in ASP.NET

ASP.NET Data Access Technologies, an Intro to LINQ

Create and Use Classes in JavaScript

- Create an Efficient and Standardized Approach for Implementing CSS Styles
- What is ASP.NET
- Choosing the Right Programming Model
- Build Your First ASP.NET Application with ASP.NET Web Forms
- Build Your First ASP.NET Application with ASP.NET MVC
- Best Learn ASP.NET Web Forms
- 5 Minute Intro to ASP.NET Web Forms
- Install ASP.NET and Tools

Tailspin Spyworks

- Tailspin Spyworks - Creating and Popular Products Control
- Tailspin Spyworks - Implementing and Also Purchased Control
- Tailspin Spyworks - Intro UI and EDM
- Tailspin Spyworks - Directory Organization
- Tailspin Spyworks - Category Menu
- Tailspin Spyworks - Display the Product List
- Tailspin Spyworks - Display Per Product Details
- Tailspin Spyworks - Adding Items to the Shopping Cart
- Tailspin Spyworks - Display Shopping Cart
- Tailspin Spyworks - Update the Shopping Cart
- Tailspin Spyworks - Migrate the Shopping Cart
- Tailspin Spyworks - Final Check Out
- Tailspin Spyworks - Adding User Product Reviews
- Tailspin Spyworks - Displaying User Reviews

Building 3.5 Applications

- Intro to Visual Web Developer
- IntelliSense
- Intro to Web Forms
- Page Layout
- Page Lifecycle Events
- Intro to ASP.NET Controls
- Submit and Postback
- Application Level Objects

- Session Level Objects
- Debugging
- CSS
- MasterPages
- Intro to SOAP Based Web Services with Visual Web Developer 2008
- AJAX Style Services with Microsoft Visual Web Developer 2008
- Validation
- Login Controls

Authentication

- Using Basic Forms Authentication in ASP.NET
- Change the Forms Authentication Properties
- Setup and Use Cookie-less Authentication in an ASP.NET Application
- ASP Forms Login Relocation
- Forms Login Custom Key Configuration
- Add Custom Data to the Authentication Method
- Use Custom Principal Objects
- Understanding ASP.NET Memberships
- Configuring SQL To Work with Membership Schemas
- Changing Membership Settings in the Default Membership Schema
- Creating User Accounts with the Create User Wizard
- Creating User Accounts Programmatically
- Validating Users Manually
- Validating Users with the Login Control
- Adding Users to Your Membership System
- Logging Users Into Your Membership System
- Implement the Registration Verification Pattern
- Simple Web Service Authentication
- Creating Inactive Users
- SQL Injection Defense

SQL 2005

- What is a Database
- Understanding Database Tables and Records

- More about Column Data Types and Other Properties
- Designing Relational Database Tables
- Manipulating Database Data
- More Structured Query Language
- Understanding Security and Network Connectivity
- Connecting your Web Application to SQL Server 2005 Express Edition
- Using SQL Server Management Studio
- Getting Started with Reporting Services
- Building and Customizing Reports in Business Intelligence Development Studio
- Creating and Using Stored Procedures
- Enabling Full-Text Search in your Text Data

Visual Studio 2005

- Intro to Unit Testing with Team System
- Intro to Testing Web Applications with Team System
- Intro to Load Testing Web Applications with Team System
- Intro to Manual Testing with Team System
- Intro to Managing and Running Tests with Team System
- Measuring the Business Value of AJAX
- Code Coverage of Automated Tests
- Custom Extraction Rules and Coded Web Tests
- Effects of Caching
- Load Test Agent
- Effects of ViewState
- Integrate Defect Tracking with Testing
- Create My Own Bug Work Item
- Write Code More Quickly with Unit Tests
- Practice Test-Driven Development
- Load Test a Web Application
- Tune Web Application Performance with Profiling
- Set Up Distributed Load Testing for High Volume Tests
- Enforce Coding Standards with Code Analysis
- Use Generic Tests

Publish and Analyze Test Results
Discover Application Changes Prior to Deployment
Implement Continuous Integration with Team Foundation
Automate Testing using Team Build
Deploy a Web Application during a Team Build
Run Unit Tests against a Deployed Database
Enable Code Coverage and Profiling in Production Applications
Web Deployment Projects
Web Application Projects & Web Deployment Projects
Migrating
[Intro to ASP.NET 2.0:] ASP.NET 2.0 Fundamentals
[Intro to ASP.NET 2.0:] User Interface Elements
[Migrating from] Classic ASP to ASP.NET
Intro to ASP.NET for JSP Developers: Welcome to ASP.NET 2.0
Intro to ASP.NET for JSP Developers: Building Applications
Intro to ASP.NET for ColdFusion Developers: Adding ASP.NET to Your Repertoire
Intro to ASP.NET for ColdFusion Developers: Building an ASP.NET Application
Interop between PHP and the Windows Platform
Building 2.0 Applications
[Lesson 1:] Getting Started with Visual Web Developer Express
[Lesson 2:] Creating a Web Forms User Interface
[Lesson 3:] Understanding More About Events and Postback
[Lesson 4:] Understanding Web Application State
[Lesson 5:] Debugging and Tracing Your Website
[Lesson 6:] Working with Stylesheets and Master Pages
[Lesson 7:] Databinding to User Interface Controls
[Lesson 8:] Working with the GridView and FormView
Watch ASP.NET Development in Action
General IIS Tips
Developing and Deploying In a Shared Hosting
Working with IIS7 Delegated Admin
Feature Specific Delegated Management

- Troubleshooting Production ASP.NET Apps
- Creating a Site with IIS7 Manager
- Installing FTP7
- Bit Rate Throttling
- IIS7 Playlists
- General VS 2010 Tips
  - Code Optimized Profile
  - Code Search View Hierarchy
  - IntelliSense Smart Lists
  - Multi-Monitor Support
  - New Web Project Template
  - New Multi-Targeting
  - Websites Instead of Web Projects
  - Snippets IntelliSense
- Ajax Control Toolkit
  - Get Started with the ASP.NET AJAX Control Toolkit
  - ASP.NET AJAX CascadingDropDown Control Extender
  - ASP.NET AJAX TextBoxWatermark Control Extender
  - ASP.NET AJAX Popup Control Extender
  - ASP.NET AJAX ModalPopup Extender Control
  - ASP.NET AJAX AlwaysVisible Control Extender
  - ASP.NET AJAX Accordion Control
  - ASP.NET AJAX Collapsable Panel Extender
  - ASP.NET AJAX Draggable Panel Extender
  - ASP.NET AJAX DynamicPopulate Extender
  - ASP.NET AJAX FilteredTextbox Extender
  - ASP.NET AJAX HoverMenu Extender
  - ASP.NET AJAX ToggleButton Extender
  - ASP.NET AJAX DropShadow Extender
  - ASP.NET AJAX PasswordStrength Extender
  - Get Started with the ASP.NET AJAX Animation Extender Control
  - ASP.NET AJAX ConfirmButton Extender

ASP.NET AJAX Slider Control
ASP.NET AJAX AutoComplete Control
Configure the ASP.NET AJAX Calendar Control
ASP.NET AJAX DropDown Control
ASP.NET AJAX MaskedEdit Controls
ASP.NET AJAX MutuallyExclusive Checkbox Extender
ASP.NET AJAX NoBot Control
ASP.NET AJAX ListSearch Extender
PagingBulletedList Extender Control
NumericUpDown Extender Control
ASP.NET AJAX ValidatorCallout Extender
ASP.NET AJAX ResizableControl Extender
ASP.NET AJAX Tabs Control
ASP.NET AJAX SlideShow Extender
ASP.NET AJAX UpdatePanelAnimation Extender
AJAX Toolkit Reorder Control
Utilize the AJAX Rating Control in the ASP.NET Toolkit
Control Extenders
Color Picker
Combo Box
Editor Control
Editor Control Custom
Create a New Custom Extender
ASP.NET AJAX
Get Started with ASP.NET AJAX
Implement Dynamic Partial-Page Updates with ASP.NET AJAX
Make Client-Side Network Callbacks with ASP.NET AJAX
Add ASP.NET AJAX Features to an Existing Web Application
ASP.NET AJAX Enable an Existing Web Service
ASP.NET AJAX Client Library Controls
Use an ASP.NET AJAX ScriptManagerProxy
ASP.NET AJAX RoundedCorners Extender

ASP.NET AJAX Timer Control

Implement the Predictive Fetch Pattern for AJAX

Implement the AJAX Paging Pattern

Implement the AJAX Incremental Page Display Pattern

Implement the Incremental Page Display Pattern using HTTP GET and POST

ASP.NET AJAX UpdateProgress Control

ASP.NET AJAX History Control

Implement the AJAX After Processing Pattern

Update Multiple Regions of a Page with ASP.NET AJAX

Choose Between Methods of AJAX Page Updates

Use Other JavaScript User Interface Libraries with ASP.NET AJAX

Use the ASP.NET AJAX Profile Services

Debug ASP.NET AJAX Applications Using Visual Studio 2005

Build a Custom ASP.NET AJAX Server Control

Use JavaScript to Refresh an ASP.NET AJAX UpdatePanel

Determine Whether an Asynchronous Postback has Occurred

Use the Conditional UpdateMode of the UpdatePanel

Implement the Persistent Communications Pattern with the UpdatePanel

Localize an ASP.NET AJAX Application

Implement the Persistent Communications Pattern using Web Services

Trigger an UpdatePanel Refresh from a DropDownList Control

Create an ASP.NET AJAX Extender from Scratch

Build Custom Server Controls that Work With or Without ASP.NET AJAX

Associate AJAX Client Behavior with an ASP.NET Server Control

Retrieve Values From Server Side AJAX Controls

Two Simple Techniques for Triggering Updates to Update Panels

Use ASP.NET AJAX Cascading Drop Down Control to Access a Database

Implement Infinite Data Patterns in AJAX

Basic ASP.NET Authentication in an AJAX Enabled Application

Dynamically Change CSS ASP.NET AJAX UpdatePanel

Dynamically Add Controls to a Web Page

Set Up Your Development Environment for ASP.NET 3.5

- Set Up Your Development Environment for ASP.NET 2.0
- Customize Error Handling for the ASP.NET AJAX UpdatePanel
- Use ASP.NET AJAX Client Templates

Data Access

- ADO.NET Data Services
  - Data Access Layers in ASP.NET Applications
  - Manually Bind a Dataset to a Datagrid
  - Work with Datasets and Filters from an ASP Application
- How Do I with LINQ
  - LINQ to SQL: Overview
  - LINQ to SQL: Data Model
  - LINQ to SQL: Querying the Database
  - LINQ to SQL: Updating the Database
  - LINQ to SQL: LinqDataSource
  - LINQ to SQL: Custom LinqDataSource
  - LINQ to SQL: Using Stored Procedures
  - LINQ to SQL: Updating with Stored Procedures
  - LINQ to SQL: Executing Arbitrary SQL
- LINQ Videos from the VB Team
  - Get Started with LINQ
  - Perform Group and Aggregate Queries
  - Upgrade Visual Basic Projects to Enable LINQ
  - Get Started with LINQ to XML
  - Enable XML IntelliSense and Use XML Namespaces
  - Create XML Documents from SQL Data
  - Create Excel Spreadsheets using LINQ to XML

ASP.NET Dynamic Data

- Your First Scaffold and What is Dynamic Data
- Enable Inline GridView Editing
- Change how my Fields render
- Handle Business Logic Exceptions
- Make Custom Pages

- Display Unknown datatypes
- Use a DynamicControl in ListView and DetailsView Controls
- Getting Started with Dynamic Data
- Begin Editing the Templates in ASP.NET Dynamic Data Applications
- Begin Modifying Dynamic Data Applications with URL Routing
- Enable In-Line Editing in ASP.NET Dynamic Data Applications
- Enable Table Specific Routing in Dynamic Data Applications
- Use Attribute Validation in ASP.NET Dynamic Data Applications
- Implement Custom Field Validation with Imperative Logic in VB or C#
- Remove Columns From Your DynamicData Data Grids
- Create Table Specific Custom Forms in an ASP.NET Dynamic Data Application
- ASP.NET Dynamic Data Custom Form Formatting

ASP.NET 3.5

- How Do I
  - Create a Master Page in Visual Studio 2008
  - Create Nested Master Page in Visual Studio 2008
  - Cascading Style Sheets in Visual Studio 2008
  - Working with Visual Studio 2008 .NET Framework
  - Adding Elements to a CSS File and Create New CSS on the Fly
  - Advance Cascading Style Sheet Features and Management
  - Converting a .NET 2.0 Windows Forms Application to .NET 3.5
  - Get Started with the Entity Framework
  - Use the New Entity Data Source
  - Serialize a Graph with the Entity Framework
  - Use MSBuild to Automate the ASP.NET Compiler and Merge Utilities
- ASP.NET Ajax
  - ASP.NET AJAX Support in Visual Studio 2008
  - Adding AJAX Functionality to an Existing ASP.NET Page
  - Creating and Using an AJAX-enabled Web Service in a Web Site
  - ASP.NET AJAX: A demonstration of ASP.NET AJAX
  - ADO.NET Data Services with ASP.NET AJAX Support
  - Intro to ASP.NET Ajax History

- Using Script Combining to improve AJAX performance
- Data Controls
  - ListView Control
  - DataPager Control
- Visual Studio Designer
  - Visual Studio 2008 and Nested Masterpages
- Visual Studio Editor
  - New Designer Support in Visual Studio 2008
  - JavaScript Intellisense Support in Visual Studio 2008
  - JavaScript Debugging in Visual Studio 2008
  - Multi Targeting Support in Visual Studio 2008
  - IntelliSense for Jscript and ASP.NET AJAX
  - Quick Tour of the Visual Studio 2008 Integrated Development Environment
  - Creating and Modifying a CSS File
- jQuery
  - Make Ajax Calls Using jQuery
- .NET 4
  - Ajax
    - Imperative JavaScript Syntax for Microsoft Client Side Controls
    - ScriptLoader
    - JQuery Syntax for Microsoft Ajax
    - AJAX Data Templates
  - Chart
    - Chart Control
    - Introducing the New Chart Control in Visual Studio 2010
  - Core
    - Auto Start
    - Clean Web.Config Files
    - Predictable Client IDs
    - HtmlEncoder Utility Method
  - Data
    - Persistent GridView Row Selection

- Markup Control
  - New Rendering Option for Check Box Lists and Radio Button Lists
  - Table Free Templated Controls
  - Tableless Menu Control
  - Hidden Field Divs
  - Disabled Control Styling
- Performance
  - Selective View State
  - Easy State Compression
  - Use the ViewStateMode Property for Managing ViewState
- Routing
  - Permanent Redirect
  - Imperative WebForms Routing
  - Declarative WebForms Routing
  - Outbound WebForms Routing
  - Use Routing with ASP.NET Web Forms
  - Work with URLs in ASP.NET Routing
- SEO
  - Dynamic Metadata
- ASP.NET 4.5
  - Strongly Typed Data Controls
  - Model Binding Part 1 - Selecting Data
  - Model Binding Part 2 - Filtering
  - Model Binding Part 3 - Updating
  - ASP.NET 4.5 Web Forms Model Binding
  - ASP.NET 4.5 Web Forms Strong Typed Data Controls
  - Bundling and Minification
  - Getting Started with the Next Version of ASP.NET
  - ASP.NET and Web Tools 2012.2
- Introduction to ASP.NET Web Forms
- Web Forms Video Training from Pluralsight

MVC

- Guidance

Getting Started
- Getting Started with ASP.NET MVC 5
  - Getting Started
  - Adding a Controller
  - Adding a View
  - Adding a Model
  - Creating a Connection String and Working with SQL Server LocalDB
  - Accessing Your Model's Data from a Controller
  - Examining the Edit Methods and Edit View
  - Adding Search
  - Adding a New Field
  - Adding Validation
  - Examining the Details and Delete Methods
- Getting Started with EF 6 using MVC 5
  - Create an Entity Framework Data Model
  - Implementing Basic CRUD Functionality
  - Sorting, Filtering, and Paging
  - Connection Resiliency and Command Interception
  - Code First Migrations and Deployment
  - Creating a More Complex Data Model
  - Reading Related Data
  - Updating Related Data
  - Async and Stored Procedures
  - Handling Concurrency
  - Implementing Inheritance
  - Advanced Entity Framework Scenarios
- EF Database First with ASP.NET MVC
  - Getting Started with Entity Framework 6 Database First using MVC 5
  - Creating the Web Application and Data Models
  - Generating Views
  - Changing the Database
  - Customizing a View

Enhancing Data Validation
Lifecycle of an ASP.NET MVC 5 Application
MVC Recommended Resources
MVC recommended tutorials and articles
UI, Layouts, and Themes
Using Page Inspector in ASP.NET MVC
Dynamic v. Strongly Typed Views
Security, Authentication, and Authorization
Create an ASP.NET MVC 5 App with Facebook, Twitter, LinkedIn and Google OAuth2 Sign-on (C#)
Create a secure ASP.NET MVC 5 web app with log in, email confirmation and password reset (C#)
ASP.NET MVC 5 app with SMS and email Two-Factor Authentication
XSRF/CSRF Prevention in ASP.NET MVC and Web Pages
Preventing Open Redirection Attacks (C#)
Deployment
Migrating ASP.NET MVC Applications to Windows Containers
Performance
Profile and debug your ASP.NET MVC app with Glimpse
Using Asynchronous Methods in ASP.NET MVC 4
Bundling and Minification
Internationalization and Advanced Topics
Custom MVC Template
Releases
What's New in ASP.NET MVC 5.2
What's New in ASP.NET MVC 5.1
How to Upgrade an ASP.NET MVC 4 and Web API Project to ASP.NET MVC 5 and Web API 2
Older Versions - MVC 3 and 4
Getting Started with ASP.NET MVC 4
Intro to ASP.NET MVC 4
Adding a Controller
Adding a View

- Adding a Model
- Accessing Your Model's Data from a Controller
- Examining the Edit Methods and Edit View
- Adding a New Field to the Movie Model and Table
- Adding Validation to the Model
- Examining the Details and Delete Methods

Getting Started with EF 5 using MVC 4

- Creating an Entity Framework Data Model
- Implementing Basic CRUD Functionality
- Sorting, Filtering, and Paging
- Creating a More Complex Data Model
- Reading Related Data
- Updating Related Data
- Handling Concurrency
- Implementing Inheritance
- Implementing the Repository and Unit of Work Patterns
- Advanced Entity Framework Scenarios
- Building the Chapter Downloads

MVC 4 Hands On Labs

- What's New in ASP.NET MVC 4
- ASP.NET MVC 4 Fundamentals
- ASP.NET MVC 4 Models and Data Access
- ASP.NET MVC 4 Custom Action Filters
- ASP.NET MVC 4 Entity Framework Scaffolding and Migrations
- ASP.NET MVC 4 Dependency Injection
- ASP.NET MVC 4 Helpers, Forms and Validation

ASP.NET MVC 4 Mobile Features

Using OAuth Providers with MVC 4

Getting Started with ASP.NET MVC3

- Getting Started with ASP.NET MVC 3 (C#)
  - Intro to ASP.NET MVC 3 (C#)
  - Adding a Controller (C#)

- Adding a View (C#)
- Adding a Model (C#)
- Accessing your Model's Data from a Controller (C#)
- Examining the Edit Methods and Edit View (C#)
- Adding a New Field to the Movie Model and Table (C#)
- Adding Validation to the Model (C#)
- Improving the Details and Delete Methods (C#)

Getting Started with ASP.NET MVC 3 (VB.NET)

- Intro to ASP.NET MVC 3 (VB)
- Adding a Controller (VB)
- Adding a View (VB)
- Adding a Model (VB)
- Accessing your Model's Data from a Controller (VB)
- Examining the Edit Methods and Edit View (VB)
- Adding a New Field to the Movie Model and Database Table (VB)
- Adding Validation to the Model (VB)
- Improving the Details and Delete Methods (VB)

Creating a MVC 3 Application with Razor and Unobtrusive JavaScript

MVC 3 Music Store

- Part 1: Overview and File->New Project
- Part 2: Controllers
- Part 3: Views and ViewModels
- Part 4: Models and Data Access
- Part 5: Edit Forms and Templating
- Part 6: Using Data Annotations for Model Validation
- Part 7: Membership and Authorization
- Part 8: Shopping Cart with Ajax Updates
- Part 9: Registration and Checkout
- Part 10: Final Updates to Navigation and Site Design, Conclusion

MVC 3 - The HTML5 and jQuery UI Datepicker Popup Calendar

- HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 1
- HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 2

- HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 4
- MVC 3 - Working with the DropDownList Box and jQuery
  - Using the DropDownList Helper with ASP.NET MVC
  - Examining how ASP.NET MVC scaffolds the DropDownList Helper
  - Adding a New Category to the DropDownList using jQuery UI
- Older Versions - MVC 1 and 2
  - Overview
    - ASP.NET MVC Overview
    - Understanding the ASP.NET MVC Execution Process
    - Understanding Models, Views, and Controllers (C#)
    - Understanding Models, Views, and Controllers (VB)
  - Getting Started With MVC
    - Intro to ASP.NET MVC
    - Adding a Controller
    - Adding a View
    - Creating a Database
    - Accessing your Model's Data from a Controller
    - Adding a Create Method and Create View
    - Adding Validation to the Model
    - Adding a Column to the Model
  - Models (Data)
    - Creating Model Classes with the Entity Framework (C#)
    - Creating Model Classes with LINQ to SQL (C#)
    - Displaying a Table of Database Data (C#)
    - Performing Simple Validation (C#)
    - Validating with the IDataErrorInfo Interface (C#)
    - Validating with a Service Layer (C#)
    - Validation with the Data Annotation Validators (C#)
    - Creating Model Classes with the Entity Framework (VB)
    - Creating Model Classes with LINQ to SQL (VB)
    - Displaying a Table of Database Data (VB)
    - Performing Simple Validation (VB)

Validating with the IDataErrorInfo Interface (VB)
Validating with a Service Layer (VB)
Validation with the Data Annotation Validators (VB)
ASP.NET MVC Views
ASP.NET MVC Views Overview (C#)
Creating Custom HTML Helpers (C#)
TagBuilder Class to Build HTML Helpers (C#)
Creating Page Layouts with View Master Pages (C#)
Passing Data to View Master Pages (C#)
ASP.NET MVC Views Overview (VB)
Creating Custom HTML Helpers (VB)
TagBuilder Class to Build HTML Helpers (VB)
Creating Page Layouts with View Master Pages (VB)
Passing Data to View Master Pages (VB)
Controllers and Routing
ASP.NET MVC Routing Overview (C#)
Understanding Action Filters (C#)
Improving Performance with Output Caching (C#)
Adding Dynamic Content to a Cached Page (C#)
Creating a Controller (C#)
Creating an Action (C#)
ASP.NET MVC Routing Overview (VB)
Understanding Action Filters (VB)
Improving Performance with Output Caching (VB)
Adding Dynamic Content to a Cached Page (VB)
Creating a Controller (VB)
Creating an Action (VB)
ASP.NET MVC Controller Overview (C#)
Creating Custom Routes (C#)
Creating a Route Constraint (C#)
Creating a Custom Route Constraint (C#)
ASP.NET MVC Controller Overview (VB)

Creating Custom Routes (VB)
Creating a Route Constraint (VB)
Creating a Custom Route Constraint (VB)
Deployment
Using ASP.NET MVC with Different Versions of IIS (C#)
Using ASP.NET MVC with Different Versions of IIS (VB)
Security
Authenticating Users with Forms Authentication (C#)
Authenticating Users with Windows Authentication (C#)
Preventing JavaScript Injection Attacks (C#)
Authenticating Users with Forms Authentication (VB)
Authenticating Users with Windows Authentication (VB)
Preventing JavaScript Injection Attacks (VB)
Unit Testing
Creating Unit Tests for ASP.NET MVC Applications (C#)
Creating Unit Tests for ASP.NET MVC Applications (VB)
Contact Manager
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
Iteration
NerdDinner

- Introducing the NerdDinner Tutorial
- Create a New ASP.NET MVC Project
- Create a Database
- Build a Model with Business Rule Validations
- Use Controllers and Views to Implement a Listing/Details UI
- Provide CRUD (Create, Read, Update, Delete) Data Form Entry Support
- Use ViewData and Implement ViewModel Classes
- Re-use UI Using Master Pages and Partials
- Implement Efficient Data Paging
- Secure Applications Using Authentication and Authorization
- Use AJAX to Deliver Dynamic Updates
- Use AJAX to Implement Mapping Scenarios
- Enable Automated Unit Testing
- NerdDinner Wrap Up
- Movie Database
  - Create a Movie Database Application in 15 Minutes with ASP.NET MVC (C#)
  - Create a Movie Database Application in 15 Minutes with ASP.NET MVC (VB)

Videos

- ASP.NET MVC 4
  - Using Page Inspector in ASP.NET MVC
  - ASP.NET MVC 4
- ASP.NET MVC 3
  - MVC 3 - Razor View Engine
  - MVC 3 - Razor Helpers
- ASP.NET MVC 2
  - How-Do-I
    - Creating a Tasklist Application with ASP.NET MVC
    - Creating a Movie Database Application in 15 minutes with ASP.NET MVC
    - Understanding Models, Views, and Controllers
    - ASP.NET MVC Controller Overview
    - Understanding Controllers, Controller Actions, and Action Results
    - Understanding Views, View Data, and HTML Helpers

- Intro to URL Routing
- Preventing JavaScript Injection Attacks
- Creating Unit Tests for ASP.NET MVC Applications
- Creating Custom HTML Helpers
- Creating Model Classes with LINQ to SQL
- Displaying a Table of Database Data
- What is ASP.NET MVC 80 minute technical video for developers, building NerdDinner
- Why ASP.NET MVC 3 minute overview video for decision makers
- ASP.NET MVC: How 10 minute technical video for developers
- Return JSON Formatted Data for an AJAX Call in an ASP.NET MVC Web Application
- Work with Data in ASP.NET MVC Partial Views
- Implement View Models to Manage Data for ASP.NET MVC Views
- Create a Custom HTML Helper for an MVC Application
- Work with Model Binders in an MVC Application
- Use HttpVerbs Attributes in an MVC Application
- MVC2 - Html Encoding
- MVC2 - StronglyTyped Helpers
- MVC2 - Model Validation
- MVC2 - Template Customization
- ASP.NET MVC 2 - Areas
- ASP.NET MVC 2 - Render Action
- 5 Minute Intro to ASP.NET MVC
- Best Learn ASP.NET MVC

ASP.NET MVC For the Rest of Us

- ASP.NET MVC For the Rest of Us: Part 1
- ASP.NET MVC For the Rest of Us: Part 2
- ASP.NET MVC For the Rest of Us: Part 3
- ASP.NET MVC For the Rest of Us: Part 4

Music Store

- MVC Music Store Part 1: Intro, Tools, and Project Structure
- MVC Music Store Part 2: Controllers

MVC Music Store Part 3: Views and ViewModels
ASP.NET MVC 1
ASP.NET MVC Storefront
ASP.NET MVC Storefront Part 1: Architectural Discussion and Overview
ASP.NET MVC Storefront Part 2: Repository Pattern
ASP.NET MVC Storefront Part 3: Pipes and Filters
ASP.NET MVC Storefront Part 4: Linq To Sql Spike
ASP.NET MVC Storefront Part 5: Globalization
ASP.NET MVC Storefront Part 6: Finishing Repository, and Initial UI Work
ASP.NET MVC Storefront Part 7: Routing and UI Work
ASP.NET MVC Storefront Part 8: Testing Controllers, Iteration 1 Complete
ASP.NET MVC Storefront Part 9: Shopping Cart
ASP.NET MVC Storefront Part 10: Shopping Cart Refactor and Authorization
ASP.NET MVC Storefront Part 11: Hooking Up Shopping Cart and Using Components
ASP.NET MVC Storefront Part 12: Mocking
ASP.NET MVC Storefront Part 13: Dependency Injection
ASP.NET MVC Storefront Part 14: Rich Client Interaction
ASP.NET MVC Storefront Part 15: Public Code review
ASP.NET MVC Storefront Part 16: Membership Redo With OpenID
ASP.NET MVC Storefront Part 17: Checkout With Jeff Atwood
ASP.NET MVC Storefront Part 18: Creating Experience
ASP.NET MVC Storefront Part 19: Processing Orders With Windows Workflow
ASP.NET MVC Storefront Part 19a: Windows Workflow Followup
ASP.NET MVC Storefront Part 20: Logging
ASP.NET MVC Storefront Part 21: Order Manager and Personalization
ASP.NET MVC Storefront Part 22: Restructuring, Rerouting, and PayPal
ASP.NET MVC Storefront Part 23: Getting Started With Domain-Driven Design
ASP.NET MVC Storefront Part 24: Finis
Talks
Creating NerdDinner.com with Microsoft ASP.NET Model View Controller (MVC)
America's Next Top Model View Controller Framework

- Ninja on Fire Black Belt Tips
- ASP.NET MVC 5
- ASP.NET MVC 4
- ASP.NET MVC 3
- MVC Video Training from Pluralsight
- Web API
  - Guidance
    - Getting Started
      - Getting Started with Web API 2 (C#)
      - Action Results in Web API 2
      - Using Web API with ASP.NET Web Forms
      - Creating Web API Help Pages
      - Hands On Lab: Build a Single Page Application (SPA) with ASP.NET Web API and Angular.js
    - Routing
      - Routing in Web API
      - Routing and Action Selection in Web API
      - Attribute Routing in Web API 2
      - Create a REST API with Attribute Routing in Web API 2
    - Working with Data
      - Using Web API 2 with Entity Framework 6
        - Create the Project
        - Add Models and Controllers
        - Seed the Database
        - Handling Entity Relations
        - Create Data Transfer Objects (DTOs)
        - Create the JavaScript Client
        - Create the View (UI)
        - Display Item Details
        - Add a New Item to the Database
        - Publish the App to Azure Websites
    - Mobile Clients
      - Calling Web API from a Windows Phone 8 Application (C#)

OData
- OData v4
  - Create an OData v4 Endpoint
  - Create an OData v4 Client App (C#)
  - Entity Relations in OData v4
  - Actions and Functions in OData v4
  - Containment in OData v4
  - Create a Singleton in OData v4
  - Open Types in OData v4
  - Complex Type Inheritance in OData v4
- OData v3
  - Creating an OData v3 Endpoint
  - Supporting Entity Relations in OData v3
  - OData Actions in Web API 2
  - Calling an OData Service From a .NET Client (C#)
- Supporting OData Query Options in Web API 2
- Using $select, $expand, and $value in Web API 2 OData
- Routing Conventions in Web API 2 OData
- Security Guidance for Web API 2 OData

Serialization and Model Binding
- Media Formatters in Web API 2
- JSON and XML Serialization in Web API
- BSON Support in Web API 2.1
- Content Negotiation in Web API
- Model Validation in Web API
- Parameter Binding in Web API

Error Handling
- Exception Handling in Web API
- Global Error Handling in Web API 2

Testing and Debugging
- Tracing in Web API 2
- Unit Testing Web API 2

Unit Testing Controllers in ASP.NET Web API 2
Mocking Entity Framework when Unit Testing Web API 2
Troubleshooting HTTP 405 Errors after Publishing Web API 2 Applications
Security, Authentication, and Authorization
Authentication and Authorization in Web API
Secure a Web API with Individual Accounts in Web API 2.2
External Authentication Services with Web API (C#)
Preventing Cross-Site Request Forgery (CSRF) Attacks in Web API
Enabling Cross-Origin Requests in Web API 2
Authentication Filters in Web API 2
Basic Authentication in Web API
Forms Authentication in Web API
Integrated Windows Authentication
Working with SSL
Deployment and Hosting
Use OWIN to Self-Host Web API 2
Host Web API 2 in an Azure Worker Role
Advanced Topics
Configuring Web API 2
Dependency Injection in Web API 2
Calling a Web API From a .NET Client (C#)
HTTP Message Handlers in Web API
HttpClient Message Handlers in Web API
Sending HTML Form Data in Web API: Form-urlencoded Data
Sending HTML Form Data in Web API: File Upload and Multipart MIME
HTTP Cookies in ASP.NET Web API
Releases
What's New in Web API OData 5.3
What's New in ASP.NET Web API 2.2
What's New in ASP.NET Web API 2.1
Older Versions
Enabling CRUD Operations in Web API 1

- Using ASP.NET Web API 1 with Entity Framework 5
  - Overview and Creating the Project
  - Creating the Domain Models
  - Creating an Admin Controller
  - Adding an Admin View
  - Creating a Dynamic UI with Knockout.js
  - Creating Product and Order Controllers
  - Creating the Main Page
- Self-Host Web API 1 (C#)
- Build RESTful API's with ASP.NET Web API
- Videos
  - Overview and Getting Started
    - ASP.NET Web API
    - ASP.NET Web API, Part 1: Your First Web API
    - ASP.NET Web API, Part 2: Getting Data
    - ASP.NET Web API, Part 3: Delete and Update
    - ASP.NET Web API, Part 4: Paging and Querying
    - ASP.NET Web API, Part 5: Custom Validation
    - ASP.NET Web API, Part 6: Authorization
- Web API Samples List
- Web Pages
  - Guidance
    - Getting Started
      - Program ASP.NET Web Pages in Visual Studio
      - Intro to ASP.NET Web Programming Razor Syntax
      - ASP.NET Web Pages Visual Basic
      - ASP.NET Web Pages (Razor) FAQ
      - Sending Email from Your Site
      - Adding Social Networking to Your Website
      - Introducing ASP.NET Web Pages
        - Getting Started
        - Programming Basics

- Displaying Data
- HTML Form Basics
- Entering Database Data by Using Forms
- Updating Database Data
- Deleting Database Data
- Creating a Consistent Layout
- Publishing a Site by Using WebMatrix

Routing

- Creating Readable URLs in ASP.NET Web Pages Sites

UI, Layouts, and Themes

- Working with Forms
- Validating User Input in ASP.NET Web Pages Sites
- Creating a Consistent Look
- Customizing Site-Wide Behavior
- Working with Images
- Working with Video
- Creating and Using a Helper in an ASP.NET Web Pages Site
- Installing a Helper
- Twitter Helper with ASP.NET Web Pages
- Displaying Maps in an ASP.NET Web Pages Site

Mobile

- Rendering ASP.NET Web Pages Sites for Mobile Devices

Working with Data

- Working with Data
- Displaying Data in a Chart
- Working with Files

Testing and Debugging

- Intro to Debugging
- ASP.NET Web Pages (Razor) Troubleshooting Guide

Security

- Adding Security and Membership
- Enabling Login from External Sites in an ASP.NET Web Pages Site

Using a CAPTCHA to Prevent Automated Programs (Bots) from Using Your ASP.NET Web Site
Performance
Caching to Improve the Performance of Your Website
Analyzing Traffic
Bundling and Minifying Assets in an ASP.NET Web Pages (Razor) Site
API Reference
ASP.NET API Quick Reference
Releases
What's New in ASP.NET Web Pages 3.2
What's New in ASP.NET Web Pages 3.1
Top Features in ASP.NET Web Pages 2
ASP.NET Web Pages 2 Developer Preview ReadMe
Running Different Versions of ASP.NET Web Pages Side by Side
Videos
Introduction
5 Minute Intro to ASP.NET Web Pages
Install and Use the Microsoft WebMatrix Tool
Create a Website Using WebMatrix
Change the Visual Style of a Web Page
Use the Layout Features in WebMatrix
Create a Data-Driven Dynamic Web Page
Create a Web Interface in WebMatrix
Create an Edit Data Page in WebMatrix
Create a Delete Data Page in WebMatrix
Publish a Website to the Internet Using WebMatrix
ASP.NET Web Pages
Getting Started with WebMatrix and ASP.NET Web Pages
Intro to ASP.NET Web Programming Razor Syntax
Creating a Consistent Look (Part 1)
Creating a Consistent Look (Part 2)
Working with Forms (Part 1)
Working with Forms (Part 2)

- Working with Data (Part 1)
- Working with Data (Part 2)
- Displaying Data in a Grid
- Displaying Data in a Chart (Part 1)
- Displaying Data in a Chart (Part 2)
- Working with Files
- Working with Images
- Working with Video
- Adding Email to Your Web Site
- Adding Search to Your Web Site
- Adding Social Networking to Your Website
- WebMatrix and Web Pages Video Training from Pluralsight
- WebMatrix Readme
  - WebMatrix Readme
  - Web Matrix and ASP.NET Web Pages (Razor) Beta 3 Release Readme
- WebMatrix Content Guide
- Configuration Builders
- SignalR
  - Guidance
    - Getting Started
      - Introduction to SignalR
      - Supported Platforms
      - Real-time chat with SignalR 2
      - Real-time chat with SignalR 2 and MVC 5
      - High-frequency real-time with SignalR 2
      - Server Broadcast with SignalR 2
      - Hands On Lab: Real-Time Web Applications with SignalR
    - Guide to the API
      - Hubs API Guide - Server (C#)
      - Hubs API Guide - JavaScript Client
      - Hubs API Guide - .NET Client (C#)
      - Understanding and Handling Connection Lifetime Events in SignalR

- Working with Groups in SignalR
- Mapping SignalR Users to Connections

Testing and Debugging

- SignalR Troubleshooting
- Unit Testing SignalR Applications
- Enabling SignalR Tracing

Security

- Introduction to SignalR Security
- Authentication and Authorization for SignalR Hubs
- Authentication and Authorization for SignalR Persistent Connections

Performance

- SignalR Performance
- Introduction to Scaleout in SignalR
- Azure SignalR Service
- SignalR Scaleout with Azure Service Bus
- SignalR Scaleout with Redis
- SignalR Scaleout with SQL Server
- SignalR Connection Density Testing with Crank
- Using SignalR Performance Counters in an Azure Web Role

Deployment and Hosting

- Tutorial: SignalR Self-Host
- Using SignalR with Web Apps in Azure App Service

Advanced

- Dependency Injection in SignalR

Releases

- Upgrading SignalR 1.x Projects to version 2

Older Versions

- Tutorial: Getting Started with SignalR 1.x
- Tutorial: Getting Started with SignalR 1.x and MVC 4
- High-Frequency Realtime with SignalR 1.x
- Tutorial: Server Broadcast with ASP.NET SignalR 1.x
- Hubs API Guide - Server

SignalR 1.x Hubs API Guide - JavaScript Client
Hubs API Guide - .NET Client
Understanding and Handling Connection Lifetime Events in SignalR
Working with Groups in SignalR 1.x
Mapping SignalR Users to Connections in SignalR 1.x
Introduction to SignalR Security (SignalR 1.x)
Authentication and Authorization for SignalR Hubs (SignalR 1.x)
Authentication and Authorization for SignalR Persistent Connections (SignalR 1.x)
SignalR Performance (SignalR 1.x)
Introduction to Scaleout in SignalR 1.x
SignalR Scaleout with Azure Service Bus (SignalR 1.x)
SignalR Scaleout with Redis (SignalR 1.x)
SignalR Scaleout with SQL Server (SignalR 1.x)
SignalR Troubleshooting (SignalR 1.x)
Dependency Injection in SignalR 1.x
Videos
ASP.NET SignalR Videos
SignalR and Web Sockets
Mobile
Overview
Controls-Tested Devices
Simulate Popular Mobile Devices for Testing
Single Page Application
Overview
Introduction
Single Page Application: KnockoutJS template
Know a library other than Knockout?
Templates
Backbone Template
Breeze/Angular template
Breeze/Knockout template
EmberJS template

Hot Towel template
WebHooks
Source Code and NuGet
Receiving and processing
Receiving
Processing
Receiver dependencies
Sending
Diagnostics
Logging
Debugging
Visual Studio 2017
Optimize build performance for solution
Visual Studio 2013
Creating ASP.NET Web Projects in Visual Studio 2013
Visual Studio 2013 Web Editor Features - Browser Link
Visual Studio 2013 Web Editor Features - CSS
Visual Studio 2013 Web Editor Features - JavaScript
Visual Studio 2013 Web Editor Features - Publishing
Visual Studio 2013 Web Editor Features - Page Inspector
Visual Studio 2013 Web Editor Features - HTML Editor
Visual Studio 2013 Web Editor Features - HTML5
ASP.NET Scaffolding in Visual Studio 2013
Browser Link
ASP.NET and Web Tools for Visual Studio 2013 Release Notes
ASP.NET and Web Tools 2013.2 for Visual Studio 2013 Release Notes
Hands On Lab: Visual Studio 2013 Web Tools
Hands On Lab: One ASP.NET: Integrating ASP.NET Web Forms, MVC and Web API
Visual Studio 2012
Visual Studio 2012 HTML Editing Features
Visual Studio 2012 CSS Editor
Visual Studio 2012 JavaScript Editor

- Visual Studio 2012 Page Inspector
- Visual Studio 2012 Web Publishing Improvements
- Release Notes for ASP.NET and Web Tools 2013.1 for Visual Studio 2012
- HTML Editor Smart Tasks and Event Handler Generation
- CSS Editor Hierarchical Indentation
- CSS Editor Snippets
- CSS Editor Color Picker
- Page Inspector - Introduction
- Page Inspector - Decomposing your Web Application
- Windows Azure Authentication
- ASP.NET and Web Tools 2012.2 Release Notes

Entity Framework

Identity

- Overview
  - Getting Started with ASP.NET Identity
    - Introduction to ASP.NET Identity
    - ASP.NET Identity Recommended Resources
    - Adding ASP.NET Identity to an Empty or Existing Web Forms Project
    - Developing ASP.NET Apps with Azure Active Directory
    - ASP.NET Identity: Using MySQL Storage with an EntityFramework MySQL Provider (C#)
  - Features & API
    - Best practices for deploying passwords and other sensitive data to ASP.NET and Azure App Service
    - Account Confirmation and Password Recovery with ASP.NET Identity (C#)
    - Two-factor authentication using SMS and email with ASP.NET Identity
  - Extensibility
    - Overview of Custom Storage Providers for ASP.NET Identity
    - Implementing a Custom MySQL ASP.NET Identity Storage Provider
    - Change Primary Key for Users in ASP.NET Identity
  - Migrations
    - Migrating an Existing Website from SQL Membership to ASP.NET Identity
    - Migrating Universal Provider Data for Membership and User Profiles to ASP.NET

Identity (C#)
Ajax
Microsoft Ajax Content Delivery Network
Microsoft Ajax Content Delivery Network
jQuery UI
jQuery UI 1.10.4 on the Microsoft Ajax CDN
jQuery UI 1.10.3 on the Microsoft Ajax CDN
jQuery UI 1.10.2 on the Microsoft Ajax CDN
jQuery UI 1.10.1 on the Microsoft Ajax CDN
jQuery UI 1.10.0 on the Microsoft Ajax CDN
jQuery UI 1.9.2 on the Microsoft Ajax CDN
jQuery UI 1.9.1 on the Microsoft Ajax CDN
jQuery UI 1.9.0 on the Microsoft Ajax CDN
jQuery UI 1.8.24 on the Microsoft Ajax CDN
jQuery UI 1.8.23 on the Microsoft Ajax CDN
jQuery UI 1.8.22 on the Microsoft Ajax CDN
jQuery UI 1.8.21 on the Microsoft Ajax CDN
jQuery UI 1.8.20 on the Microsoft Ajax CDN
jQuery UI 1.8.19 on the Microsoft Ajax CDN
jQuery UI 1.8.18 on the Microsoft Ajax CDN
jQuery UI 1.8.17 on the Microsoft Ajax CDN
jQuery UI 1.8.16 on the Microsoft Ajax CDN
jQuery UI 1.8.15 on the Microsoft Ajax CDN
jQuery UI 1.8.14 on the Microsoft Ajax CDN
jQuery UI 1.8.13 on the Microsoft Ajax CDN
jQuery UI 1.8.12 on the Microsoft Ajax CDN
jQuery UI 1.8.11 on the Microsoft Ajax CDN
jQuery UI 1.8.10 on the Microsoft Ajax CDN
jQuery UI 1.8.9 on the Microsoft Ajax CDN
jQuery UI 1.8.8 on the Microsoft Ajax CDN
jQuery UI 1.8.7 on the Microsoft Ajax CDN
jQuery UI 1.8.6 on the Microsoft Ajax CDN

- jQuery UI 1.8.5
- jQuery UI 1.11.2 on the Microsoft Ajax CDN
- jQuery UI 1.11.1 on the Microsoft Ajax CDN
- jQuery UI 1.11.0 on the Microsoft Ajax CDN
- jQuery UI 1.11.3 on the Microsoft Ajax CDN
- jQuery UI 1.11.4 on the Microsoft Ajax CDN
- jQuery UI 1.12.0 on the Microsoft Ajax CDN
- jQuery UI 1.12.1 on the Microsoft Ajax CDN

jQuery Validate

- jQuery Validation 1.13.0
- jQuery Validation 1.12.0
- jQuery Validation 1.11.1
- jQuery Validation 1.11.0
- jQuery Validation 1.10.0
- jquery.validate version 1.9
- jquery.validate version 1.8.1
- jquery.validate version 1.8
- jquery.validate version 1.7
- jQuery Validate 1.6
- jQuery Validate 1.5.5
- jQuery Validation 1.13.1
- jQuery Validation 1.14.0
- jQuery Validation 1.15.0
- jQuery Validation 1.15.1
- jQuery Validation 1.16.0

jQuery Mobile

- jQuery Mobile 1.4.2 on the Microsoft Ajax CDN
- jQuery Mobile 1.4.1 on the Microsoft Ajax CDN
- jQuery Mobile 1.4.0 on the Microsoft Ajax CDN
- jQuery Mobile 1.3.2 on the Microsoft Ajax CDN
- jQuery Mobile 1.3.1 on the Microsoft Ajax CDN
- jQuery Mobile 1.3.0 on the Microsoft Ajax CDN

jQuery Mobile 1.2.0 on the Microsoft Ajax CDN
jQuery Mobile 1.1.2 on the Microsoft Ajax CDN
jQuery Mobile 1.1.1 on the Microsoft Ajax CDN
jQuery Mobile 1.1.0 on the Microsoft Ajax CDN
jQuery Mobile 1.1.0 RC2 on the Microsoft Ajax CDN
jQuery Mobile 1.0.1 on the Microsoft Ajax CDN
jQuery Mobile 1.0 on the Microsoft Ajax CDN
jQuery Mobile 1.0 RC2 on the Microsoft Ajax CDN
jQuery Mobile 1.0 RC1 on the Microsoft Ajax CDN
jQuery Mobile 1.0 Beta 3 on the Microsoft Ajax CDN
jQuery Mobile 1.4.5 on the Microsoft Ajax CDN
jQuery Templates
jQuery Templates Beta 1
jQuery Cycle
jQuery Cycle 2.99
jQuery Cycle 2.94
jQuery Cycle 2.88
jQuery DataTables
jQuery DataTables 1.9.4
jQuery DataTables 1.9.3
jQuery DataTables 1.9.2
jQuery DataTables 1.9.1
jQuery DataTables 1.9.0
jQuery DataTables 1.8.2
jQuery DataTables 1.10.4
jQuery DataTables 1.10.5
ASP.NET Web Forms and Ajax 4
ASP.NET Ajax 3.5
ASP.NET Web Forms and Ajax 4.5.2
ASP.NET Web Forms and Ajax 4.5.1
Web Development Best Practices
What not to do in ASP.NET, and what to do instead

- Async and Await
- Building Real World Cloud Apps with Azure
  - Introduction
  - Automate Everything
  - Source Control
  - Continuous Integration and Delivery
  - Web Development Best Practices
  - Single Sign-On
  - Data Storage Options
  - Data Partitioning Strategies
  - Unstructured Blob Storage
  - Design to Survive Failures
  - Monitoring and Telemetry
  - Transient Fault Handling
  - Distributed Caching
  - Queue-Centric Work Pattern
  - More Patterns and Guidance
  - Appendix: The Fix It Sample Application
- Hands on Lab: Maintainable Azure Websites: Managing Change and Scale

OWIN and Katana
- An Overview of Project Katana
- Getting Started with OWIN and Katana
- OWIN SameSite
- Katana Samples
- OWIN Startup Class Detection
- Enabling Windows Authentication in Katana
- OWIN Middleware in the IIS Integrated Pipeline
- Host OWIN in an Azure Worker Role

Whitepapers
- Whitepapers
- ASP.NET SameSite
  - SameSite cookies in ASP.NET

SameSite KB articles
C# MVC
C# WebForms
VB WebForms
VB MVC
ASP.NET 4 and Visual Studio 2010 Web Development Overview
ASP.NET 4 and Visual Studio 2010 Web Development Overview
ASP.NET 4 Breaking Changes
Upgrading an ASP.NET MVC 1.0 Application to ASP.NET MVC 2
ASP.NET Denied Access to IIS Directories
What's New in ASP.NET MVC 2
ASP.NET MVC 3
ASP.NET MVC 4
Running ASP.NET 1.1 with IIS 6.0
How To: Add Mobile Pages to Your ASP.NET Web Forms / MVC Application
Fix for 'Server Application Unavailable' Error after Applying Security Update for IE
ASP.NET Side-by-Side Execution of .NET Framework 1.0 and 1.1
ASP.NET MVC 4
Request Validation - Preventing Script Attacks
ASP.NET Data Access - Recommended Resources
ASP.NET Web Deployment - Recommended Resources
ASP.NET and Web Tools 2012.2 Release Notes
What's New in ASP.NET 4.5 and Visual Studio 2012
API reference

ASP.NET overview

2/20/2020 • 4 minutes to read • Edit Online

ASP.NET is a free web framework for building great websites and web applications using HTML, CSS, and JavaScript. You can also create Web APIs and use real-time technologies like Web Sockets.

ASP.NET Core is an alternative to ASP.NET. See the guidance on how to choose between ASP.NET and ASP.NET Core.

Get started

Install Visual Studio 2019 Community edition, a free IDE for ASP.NET on Windows.

Websites and web applications

ASP.NET offers three frameworks for creating web applications: Web Forms, ASP.NET MVC, and ASP.NET Web Pages. All three frameworks are stable and mature, and you can create great web applications with any of them. No matter what framework you choose, you will get all the benefits and features of ASP.NET everywhere.

Each framework targets a different development style. The one you choose depends on a combination of your programming assets (knowledge, skills, and development experience), the type of application you're creating, and the development approach you're comfortable with.

Below is an overview of each of the frameworks and some ideas for how to choose between them. If you prefer a video introduction, see Making Websites with ASP.NET and What is Web Tools?

	IF YOU HAVE EXPERIENCE IN	DEVELOPMENT STYLE	EXPERTISE
Web Forms	Win Forms, WPF, .NET	Rapid development using a rich library of controls that encapsulate HTML markup	Mid-Level, Advanced RAD
MVC	Ruby on Rails, .NET	Full control over HTML markup, code and markup separated, and easy to write tests. The best choice for mobile and single-page applications (SPA).	Mid-Level, Advanced
Web Pages	Classic ASP, PHP	HTML markup and your code together in the same file	New, Mid-Level

Web Forms

With ASP.NET Web Forms, you can build dynamic websites using a familiar drag-and-drop, event-driven model. A design surface and hundreds of controls and components let you rapidly build sophisticated, powerful UI-driven sites with data access.

Learn more about Web Forms

MVC

ASP.NET MVC gives you a powerful, patterns-based way to build dynamic websites that enables a clean separation of concerns and that gives you full control over markup for enjoyable, agile development. ASP.NET MVC includes many features that enable fast, TDD-friendly development for creating sophisticated applications that use the latest

web standards.

Learn more about MVC

ASP.NET Web Pages

ASP.NET Web Pages and the Razor syntax provide a fast, approachable, and lightweight way to combine server code with HTML to create dynamic web content. Connect to databases, add video, link to social networking sites, and include many more features that help you create beautiful sites that conform to the latest web standards.

Learn more about Web Pages

Notes about Web Forms, MVC, and Web Pages

All three ASP.NET frameworks are based on the .NET Framework and share core functionality of .NET and of ASP.NET. For example, all three frameworks offer a login security model based around membership, and all three share the same facilities for managing requests, handling sessions, and so on that are part of the core ASP.NET functionality.

In addition, the three frameworks are not entirely independent, and choosing one does not preclude using another. Since the frameworks can coexist in the same web application, it's not uncommon to see individual components of applications written using different frameworks. For example, customer-facing portions of an app might be developed in MVC to optimize the markup, while the data access and administrative portions are developed in Web Forms to take advantage of data controls and simple data access.

Web APIs

ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework.

Learn more about Web API

Real-time technologies

ASP.NET SignalR is a new library for ASP.NET developers that makes developing real-time web functionality easier. SignalR allows bi-directional communication between server and client. Servers can push content to connected clients instantly as it becomes available. SignalR supports Web Sockets, and falls back to other compatible techniques for older browsers. SignalR includes APIs for connection management (for instance, connect and disconnect events), grouping connections, and authorization.

Learn more about SignalR

Mobile apps and sites

ASP.NET can power native mobile apps with a Web API back end, as well as mobile web sites using responsive design frameworks like Twitter Bootstrap. If you are building a native mobile app, it's easy to create a JSON-based Web API to handle data access, authentication, and push notifications for your app. If you are building a responsive mobile site, you can use any CSS framework or open grid system you prefer, or select a powerful mobile system like jQuery Mobile or Sencha and great mobile applications with PhoneGap.

Learn more about mobile app and site development

Single-page applications

ASP.NET Single Page Application (SPA) helps you build applications that include significant client-side interactions using HTML 5, CSS 3 and JavaScript. Visual Studio includes a template for building single page applications using knockout.js and ASP.NET Web API. In addition to the built-in SPA template, community-created SPA templates are

also available for download.

Learn more about single-page app development

WebHooks

WebHooks is a lightweight HTTP pattern providing a simple pub/sub model for wiring together Web APIs and SaaS services. When an event happens in a service, a notification is sent in the form of an HTTP POST request to registered subscribers. The POST request contains information about the event which makes it possible for the receiver to act accordingly.

WebHooks are exposed by a large number of services including Dropbox, GitHub, Instagram, MailChimp, PayPal, Slack, Trello, and many more. For example, a WebHook can indicate that a file has changed in Dropbox, or a code change has been committed in GitHub, or a payment has been initiated in PayPal, or a card has been created in Trello.

Learn more about WebHooks

ASP.NET tutorials

2/20/2020 • 4 minutes to read • Edit Online

For new web application development, we recommend Razor Pages. For more information, see Get started with Razor Pages.

The following step-by-step guides for developing ASP.NET applications are available.

> **NOTE**
> Looking for ASP.NET Core tutorials? Find them here.

Web Forms

- Code Editing ASP.NET Web Forms in Visual Studio 2013
- Using Page Inspector in Visual Studio 2012
- Model Binding and Web Forms in Visual Studio 2013
- ASP.NET Web Forms Connection Resiliency and Command Interception
- Create a secure ASP.NET Web Forms app with user registration, email confirmation and password reset (C#)
- Create an ASP.NET Web Forms app with SMS Two-Factor Authentication (C#)
- Using Asynchronous Methods in ASP.NET 4.5
- ASP.NET Web Deployment using Visual Studio
- ASP.NET 4 - Enterprise Deployment Introduction
- ASP.NET 4 - Enterprise Deployment Series 1
- ASP.NET 4 - Enterprise Deployment Series 2 Configuring Servers
- ASP.NET 4 - Enterprise Deployment Series 3 Configuring TFS
- ASP.NET 4 - Enterprise Deployment Series 4 Advanced Topics
- ASP.NET 4 - Tailspin Spyworks
- ASP.NET 4 - Getting Started with Entity Framework 4 and Web Forms
- ASP.NET 4 - Continuing with Entity Framework 4 and Web Forms
- ASP.NET 4 - Web Deployment with SQL Server Compact using Visual Studio

MVC

- Using Page Inspector in ASP.NET MVC
- Create an ASP.NET MVC 5 App with Facebook, Twitter, LinkedIn and Google OAuth2 Sign-on (C#)
- Create a secure ASP.NET MVC 5 web app with log in, email confirmation and password reset (C#)
- ASP.NET MVC 5 app with SMS and email Two-Factor Authentication
- XSRF/CSRF Prevention in ASP.NET MVC and Web Pages
- Preventing Open Redirection Attacks (C#)
- Migrating ASP.NET MVC Applications to Windows Containers
- Profile and debug your ASP.NET MVC app with Glimpse
- Using Asynchronous Methods in ASP.NET MVC 4
- How to Upgrade an ASP.NET MVC 4 and Web API Project to ASP.NET MVC 5 and Web API 2

Web API

- Getting Started with Web API 2 (C#)
- Using Web API with ASP.NET Web Forms
- Creating Web API Help Pages
- Hands On Lab: Build a Single Page Application (SPA) with ASP.NET Web API and Angular.js
- Create a REST API with Attribute Routing in Web API 2
- Using Web API 2 with Entity Framework 6
- Calling Web API from a Windows Phone 8 Application (C#)
- OData v4
- OData v3
- Tracing in Web API 2
- Unit Testing Web API 2
- Unit Testing Controllers in ASP.NET Web API 2
- Mocking Entity Framework when Unit Testing Web API 2
- Troubleshooting HTTP 405 Errors after Publishing Web API 2 Applications
- Secure a Web API with Individual Accounts in Web API 2.2
- External Authentication Services with Web API (C#)
- Enabling Cross-Origin Requests in Web API 2
- Use OWIN to Self-Host Web API 2
- Host Web API 2 in an Azure Worker Role
- Dependency Injection in Web API 2
- Calling a Web API From a .NET Client (C#)
- Sending HTML Form Data in Web API: Form-urlencoded Data
- Sending HTML Form Data in Web API: File Upload and Multipart MIME

Web Pages

- Program ASP.NET Web Pages in Visual Studio
- Sending Email from Your Site
- Adding Social Networking to Your Website
- Introducing ASP.NET Web Pages
- Creating Readable URLs in ASP.NET Web Pages Sites
- Working with Forms
- Validating User Input in ASP.NET Web Pages Sites
- Creating a Consistent Look
- Customizing Site-Wide Behavior
- Working with Images
- Working with Video
- Creating and Using a Helper in an ASP.NET Web Pages Site
- Installing a Helper
- Twitter Helper with ASP.NET Web Pages
- Displaying Maps in an ASP.NET Web Pages Site
- Rendering ASP.NET Web Pages Sites for Mobile Devices
- Working with Data
- Displaying Data in a Chart
- Working with Files

- Intro to Debugging
- Adding Security and Membership
- Enabling Login from External Sites in an ASP.NET Web Pages Site
- Using a CAPTCHA to Prevent Automated Programs (Bots) from Using Your ASP.NET Web Site
- Caching to Improve the Performance of Your Website
- Analyzing Traffic
- Bundling and Minifying Assets in an ASP.NET Web Pages (Razor) Site

ASP.NET SignalR

- Tutorial: Getting Started with SignalR 2
- Tutorial: Getting Started with SignalR 2 and MVC 5
- Tutorial: High-Frequency Realtime with SignalR 2
- Tutorial: Server Broadcast with SignalR 2
- Hands On Lab: Real-Time Web Applications with SignalR
- Unit Testing SignalR Applications
- Enabling SignalR Tracing
- SignalR Scaleout with Azure Service Bus
- SignalR Scaleout with Redis
- SignalR Scaleout with SQL Server
- SignalR Connection Density Testing with Crank
- Using SignalR Performance Counters in an Azure Web Role
- Tutorial: SignalR Self-Host
- Using SignalR with Web Apps in Azure App Service

ASP.NET Identity

- Adding ASP.NET Identity to an Empty or Existing Web Forms Project
- Developing ASP.NET Apps with Azure Active Directory
- ASP.NET Identity: Using MySQL Storage with an EntityFramework MySQL Provider (C#)
- Account Confirmation and Password Recovery with ASP.NET Identity (C#)
- Two-factor authentication using SMS and email with ASP.NET Identity
- Implementing a Custom MySQL ASP.NET Identity Storage Provider
- Change Primary Key for Users in ASP.NET Identity
- Migrating an Existing Website from SQL Membership to ASP.NET Identity
- Migrating Universal Provider Data for Membership and User Profiles to ASP.NET Identity (C#)

OWIN and Katana

- Getting Started with OWIN and Katana
- Host OWIN in an Azure Worker Role

ASP.NET Web Forms

2/20/2020 • 29 minutes to read • Edit Online

- Guidance
  - Getting Started
    - Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013
      - Getting Started with Web Forms and Visual Studio
      - Create the Project
      - Create the Data Access Layer
      - UI and Navigation
      - Display Data Items and Details
      - Shopping Cart
      - Checkout and Payment with PayPal
      - Membership and Administration
      - URL Routing
      - ASP.NET Error Handling
    - Creating a Basic Web Forms Page in Visual Studio 2013
    - Code Editing ASP.NET Web Forms in Visual Studio 2013
    - Using Page Inspector for Visual Studio 2012 in ASP.NET Web Forms
    - Visual Studio 2012 Hands On Labs
      - What's New in ASP.NET and Web Development in Visual Studio 2012
      - What's New in Web Forms in ASP.NET 4.5
      - Using Page Inspector in Visual Studio 2012
  - Working with Data
    - Model Binding and Web Forms in Visual Studio 2013
      - Retrieving and Displaying Data
      - Updating, Deleting, and Creating Data
      - Sorting, Paging, and Filtering Data
      - Integrating JQuery UI Datepicker
      - Using Query String Values to Filter Data
      - Adding Business Logic Layer
  - Security, Authentication, and Authorization
    - Create a secure ASP.NET Web Forms app with user registration, email confirmation and password reset (C#)
    - Create an ASP.NET Web Forms app with SMS Two-Factor Authentication (C#)
  - Performance
    - Using Asynchronous Methods in ASP.NET 4.5
  - Deployment
    - ASP.NET Web Deployment using Visual Studio

- Introduction
- Preparing for Database Deployment
- Web.config File Transformations
- Project Properties
- Deploying to Test
- Setting Folder Permissions
- Deploying to Production
- Deploying a Code Update
- Deploying a Database Update
- Command Line Deployment
- Deploying Extra Files
- Troubleshooting

- ASP.NET 4 - Enterprise Deployment Introduction
  - Deploying Web Applications in Enterprise Scenarios using Visual Studio 2010
  - Enterprise Web Deployment: Scenario Overview
  - Application Lifecycle Management: From Development to Production
- ASP.NET 4 - Enterprise Deployment Series 1
  - Web Deployment in the Enterprise
  - The Contact Manager Solution
  - Setting Up the Contact Manager Solution
  - Understanding the Project File
  - Understanding the Build Process
  - Building and Packaging Web Application Projects
  - Configuring Parameters for Web Package Deployment
  - Deploying Web Packages
  - Deploying Database Projects
  - Creating and Running a Deployment Command File
  - Manually Installing Web Packages
- ASP.NET 4 - Enterprise Deployment Series 2 Configuring Servers
  - Configuring Server Environments for Web Deployment
  - Choosing the Right Approach to Web Deployment
  - Scenario: Configuring a Test Environment for Web Deployment
  - Scenario: Configuring a Staging Environment for Web Deployment
  - Scenario: Configuring a Production Environment for Web Deployment
  - Configuring a Web Server for Web Deploy Publishing (Remote Agent)
  - Configuring a Web Server for Web Deploy Publishing (Web Deploy Handler)
  - Configuring a Web Server for Web Deploy Publishing (Offline Deployment)
  - Configuring a Database Server for Web Deploy Publishing
  - Creating a Server Farm with the Web Farm Framework
  - Configuring Deployment Properties for a Target Environment
- ASP.NET 4 - Enterprise Deployment Series 3 Configuring TFS
  - Configuring Team Foundation Server for Web Deployment
  - Creating a Team Project in TFS
  - Adding Content to Source Control
  - Configuring a TFS Build Server for Web Deployment

    - Creating a Build Definition That Supports Deployment
    - Deploying a Specific Build
    - Configuring Permissions for Team Build Deployment
  - ASP.NET 4 - Enterprise Deployment Series 4 Advanced Topics
    - Advanced Enterprise Web Deployment
    - Performing a "What If" Deployment
    - Customizing Database Deployments for Multiple Environments
    - Deploying Database Role Memberships to Test Environments
    - Deploying Membership Databases to Enterprise Environments
    - Excluding Files and Folders from Deployment
    - Taking Web Applications Offline with Web Deploy
    - Running Windows PowerShell Scripts from MSBuild Project Files
    - Troubleshooting the Packaging Process
- Advanced
  - ASP.NET Web Forms Connection Resiliency and Command Interception
- Older Versions - Getting Started
  - ASP.NET 4 - Tailspin Spyworks
    - Part 1: File-> New Project
    - Part 2: Data Access Layer
    - Part 3: Layout and Category Menu
    - Part 4: Listing Products
    - Part 5: Business Logic
    - Part 6: ASP.NET Membership
    - Part 7: Adding Features
    - Part 8: Final Pages, Exception Handling, and Conclusion
  - ASP.NET 4 - Getting Started with Entity Framework 4 and Web Forms
    - Getting Started with Entity Framework 4.0 Database First
    - Getting Started with Entity Framework 4.0 Database First - Part 2
    - Getting Started with Entity Framework 4.0 Database First - Part 3
    - Getting Started with Entity Framework 4.0 Database First - Part 4
    - Getting Started with Entity Framework 4.0 Database First - Part 5
    - Getting Started with Entity Framework 4.0 Database First - Part 6
    - Getting Started with Entity Framework 4.0 Database First - Part 7
    - Getting Started with Entity Framework 4.0 Database First - Part 8
  - ASP.NET 4 - Continuing with Entity Framework 4 and Web Forms
    - Entity Framework 4.0 and the ObjectDataSource Control, Part 1: Getting Started
    - Entity Framework 4.0 and the ObjectDataSource Control, Part 2: Adding a Business Logic Layer and Unit Tests
    - Entity Framework 4.0 and the ObjectDataSource Control, Part 3: Sorting and Filtering
    - Handling Concurrency with the Entity Framework 4.0 in ASP.NET 4
    - Maximizing Performance with the Entity Framework 4.0 in ASP.NET 4
    - What's New in the Entity Framework 4.0
  - ASP.NET 4 - Web Deployment with SQL Server Compact using Visual Studio
    - Visual Studio Web Deployment with SQL Server Compact - Introduction

- Visual Studio Web Deployment with SQL Server Compact- Deploying SQL Server Compact Databases
- Visual Studio Web Deployment with SQL Server Compact - Web.Config File Transformations
- Visual Studio Web Deployment with SQL Server Compact - Configuring Project Properties
- Visual Studio Web Deployment with SQL Server Compact - Deploying to IIS as a Test Environment
- Visual Studio Web Deployment with SQL Server Compact - Setting Folder Permissions
- Visual Studio Web Deployment with SQL Server Compact - Deploying to the Production Environment
- Visual Studio Web Deployment with SQL Server Compact - Deploying a Code-Only Update
- Visual Studio Web Deployment with SQL Server Compact - Deploying a Database Update
- Visual Studio Web Deployment with SQL Server Compact - Migrating to SQL Server
- Visual Studio Web Deployment with SQL Server Compact - Deploying a SQL Server Database Update
- Visual Studio Web Deployment with SQL Server Compact - Troubleshooting

- ASP.NET 3.5 - AJAX
  - Understanding Partial Page Updates with ASP.NET AJAX
  - Understanding ASP.NET AJAX UpdatePanel Triggers
  - Understanding ASP.NET AJAX Authentication and Profile Application Services
  - Understanding ASP.NET AJAX Localization
  - Understanding ASP.NET AJAX Web Services
  - Understanding ASP.NET AJAX Debugging Capabilities
- ASP.NET 3.5 - Web Forms Master Pages
  - Creating a Site-Wide Layout Using Master Pages (C#)
  - Multiple ContentPlaceHolders and Default Content (C#)
  - Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page (C#)
  - URLs in Master Pages (C#)
  - Control ID Naming in Content Pages (C#)
  - Interacting with the Master Page from the Content Page (C#)
  - Interacting with the Content Page from the Master Page (C#)
  - Master Pages and ASP.NET AJAX (C#)
  - Specifying the Master Page Programmatically (C#)
  - Nested Master Pages (C#)
  - Creating a Site-Wide Layout Using Master Pages (VB)
  - Multiple ContentPlaceHolders and Default Content (VB)
  - Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page (VB)
  - URLs in Master Pages (VB)
  - Control ID Naming in Content Pages (VB)
  - Interacting with the Master Page from the Content Page (VB)
  - Interacting with the Content Page from the Master Page (VB)
  - Master Pages and ASP.NET AJAX (VB)
  - Specifying the Master Page Programmatically (VB)
  - Nested Master Pages (VB)
- Deploying Web Site Projects in Visual Studio 2008 and earlier

- ASP.NET Hosting Options (C#)
- Determining What Files Need to Be Deployed (C#)
- Deploying Your Site Using an FTP Client (C#)
- Deploying Your Site Using Visual Studio (C#)
- Common Configuration Differences Between Development and Production (C#)
- Core Differences Between IIS and the ASP.NET Development Server (C#)
- Deploying a Database (C#)
- Configuring the Production Web Application to Use the Production Database (C#)
- Configuring a Website that Uses Application Services (C#)
- Strategies for Database Development and Deployment (C#)
- Displaying a Custom Error Page (C#)
- Processing Unhandled Exceptions (C#)
- Logging Error Details with ASP.NET Health Monitoring (C#)
- Logging Error Details with ELMAH (C#)
- Precompiling Your Website (C#)
- Users and Roles On Production Website (C#)
- ASP.NET Hosting Options (VB)
- Determining What Files Need to Be Deployed (VB)
- Deploying Your Site Using an FTP Client (VB)
- Deploying Your Site Using Visual Studio (VB)
- Common Configuration Differences Between Development and Production (VB)
- Core Differences Between IIS and the ASP.NET Development Server (VB)
- Deploying a Database (VB)
- Configuring the Production Web Application to Use the Production Database (VB)
- Configuring a Website that Uses Application Services (VB)
- Strategies for Database Development and Deployment (VB)
- Displaying a Custom Error Page (VB)
- Processing Unhandled Exceptions (VB)
- Logging Error Details with ASP.NET Health Monitoring (VB)
- Logging Error Details with ELMAH (VB)
- Precompiling Your Website (VB)
- Users and Roles On Production Website (VB)

- Older Versions - Ajax Control Toolkit
  - Getting Started
    - Get Started with the AJAX Control Toolkit (C#)
    - Using AJAX Control Toolkit Controls and Control Extenders (C#)
    - Creating a Custom AJAX Control Toolkit Control Extender (C#)
    - Get Started with the AJAX Control Toolkit (VB)
    - Using AJAX Control Toolkit Controls and Control Extenders (VB)
    - Creating a Custom AJAX Control Toolkit Control Extender (VB)
  - Accordion
    - Databinding to an Accordion (C#)
    - Dynamically Adding Accordion Pane (C#)
    - Databinding to an Accordion (VB)
    - Dynamically Adding Accordion Pane (VB)

- Animation
  - Adding Animation to a Control (C#)
  - Executing Several Animations at Same Time (C#)
  - Executing Several Animations after Each Other (C#)
  - Animation Depending On a Condition (C#)
  - Picking One Animation Out Of a List (C#)
  - Animating in Response To User Interaction (C#)
  - Disabling Actions during Animation (C#)
  - Triggering an Animation in another Control (C#)
  - Modifying Animations From Server Side (C#)
  - Executing Animations Using Client-Side Code (C#)
  - Changing an Animation Using Client-Side Code (C#)
  - Animating an UpdatePanel Control (C#)
  - Dynamically Controlling UpdatePanel Animations (C#)
  - Adding Animation to a Control (VB)
  - Executing Several Animations at Same Time (VB)
  - Executing Several Animations after Each Other (VB)
  - Animation Depending On a Condition (VB)
  - Picking One Animation Out Of a List (VB)
  - Animating in Response To User Interaction (VB)
  - Disabling Actions during Animation (VB)
  - Triggering an Animation in another Control (VB)
  - Modifying Animations From Server Side (VB)
  - Executing Animations Using Client-Side Code (VB)
  - Changing an Animation Using Client-Side Code (VB)
  - Animating an UpdatePanel Control (VB)
  - Dynamically Controlling UpdatePanel Animations (VB)
- CascadingDropdown
  - Filling a List Using CascadingDropDown (C#)
  - Using CascadingDropDown with a Database (C#)
  - Presetting List Entries with CascadingDropDown (C#)
  - Using Auto-Postback with CascadingDropDown (C#)
  - Filling a List Using CascadingDropDown (VB)
  - Using CascadingDropDown with a Database (VB)
  - Presetting List Entries with CascadingDropDown (VB)
  - Using Auto-Postback with CascadingDropDown (VB)
- CollapsiblePanel
  - Collapsing and Expanding a Panel from JavaScript (C#)
  - Collapsing and Expanding a Panel from JavaScript (VB)
- ColorPicker
  - ColorPicker Control Extender (C#)
  - ColorPicker Control Extender (VB)
- ComboBox
  - ComboBox Control (C#)

    - ComboBox Control (VB)
- ConfirmButton
    - Using a ConfirmButton In a Repeater (C#)
    - Using a ConfirmButton In a Repeater (VB)
- DropShadow
    - Adjusting the Z-Index of a DropShadow (C#)
    - Manipulating DropShadow Properties from Client Code (C#)
    - Adjusting the Z-Index of a DropShadow (VB)
    - Manipulating DropShadow Properties from Client Code (VB)
- DynamicPopulate
    - Dynamically Populating a Control (C#)
    - Dynamically Populating a Control Using JavaScript Code (C#)
    - Using DynamicPopulate with a User Control And JavaScript (C#)
    - Dynamically Populating a Control (VB)
    - Dynamically Populating a Control Using JavaScript Code (VB)
    - Using DynamicPopulate with a User Control And JavaScript (VB)
- FilteredTextBox
    - Allowing Only Certain Characters in a Text Box (C#)
    - Allowing Only Certain Characters in a Text Box (VB)
- HoverMenu
    - Using HoverMenu with a Repeater Control (C#)
    - Using HoverMenu with a Repeater Control (VB)
- HTMLEditor
    - HTML Editor Control (C#)
    - HTML Editor Control (VB)
- ModalPopup
    - Launching a Modal Popup Window from Server Code (C#)
    - Using ModalPopup with a Repeater Control (C#)
    - Handling Postbacks from a ModalPopup (C#)
    - Positioning a ModalPopup (C#)
    - Launching a Modal Popup Window from Server Code (VB)
    - Using ModalPopup with a Repeater Control (VB)
    - Handling Postbacks from a ModalPopup (VB)
    - Positioning a ModalPopup (VB)
- MutuallyExclusiveCheckbox
    - Creating Mutually Exclusive Checkboxes (C#)
    - Creating Mutually Exclusive Checkboxes (VB)
- NoBot
    - Fighting Bots (C#)
    - Fighting Bots (VB)
- NumericUpDown
    - Creating a Numeric Up/Down Control with a Web Service Backend (C#)

- Creating a Numeric Up/Down Control with a Web Service Backend (VB)
- PasswordStrength
  - Testing the Strength of a Password (C#)
  - Testing the Strength of a Password (VB)
- Popup
  - Using Multiple Popup Controls (C#)
  - Handling Postbacks from A Popup Control With an UpdatePanel (C#)
  - Handling Postbacks from A Popup Control Without an UpdatePanel (C#)
  - Using Multiple Popup Controls (VB)
  - Handling Postbacks from A Popup Control With an UpdatePanel (VB)
  - Handling Postbacks from A Popup Control Without an UpdatePanel (VB)
- Rating
  - Creating a Rating Control (C#)
  - Creating a Rating Control (VB)
- ReorderList
  - Using Postbacks with ReorderList (C#)
  - Drag and Drop via ReorderList (C#)
  - Using Postbacks with ReorderList (VB)
  - Drag and Drop via ReorderList (VB)
- Slider
  - Slider Control With Auto-Postback (C#)
  - Databinding the Slider Control (C#)
  - Slider Control With Auto-Postback (VB)
  - Databinding the Slider Control (VB)
- TextBoxWatermark
  - Using TextBoxWatermark in a FormView (C#)
  - Using TextBoxWatermark With Validation Controls (C#)
  - Using TextBoxWatermark in a FormView (VB)
  - Using TextBoxWatermark With Validation Controls (VB)

- Older Versions - Data Access
  - Introduction
    - Creating a Data Access Layer (C#)
    - Creating a Business Logic Layer (C#)
    - Master Pages and Site Navigation (C#)
    - Creating a Data Access Layer (VB)
    - Creating a Business Logic Layer (VB)
    - Master Pages and Site Navigation (VB)
  - Basic Reporting
    - Displaying Data With the ObjectDataSource (C#)
    - Declarative Parameters (C#)
    - Programmatically Setting the ObjectDataSource's Parameter Values (C#)
    - Displaying Data With the ObjectDataSource (VB)
    - Declarative Parameters (VB)

- Programmatically Setting the ObjectDataSource's Parameter Values (VB)
- Master/Detail
  - Master/Detail Filtering With a DropDownList (C#)
  - Master/Detail Filtering With Two DropDownLists (C#)
  - Master/Detail Filtering Across Two Pages (C#)
  - Master/Detail Using a Selectable Master GridView with a Details DetailView (C#)
  - Master/Detail Filtering With a DropDownList (VB)
  - Master/Detail Filtering With Two DropDownLists (VB)
  - Master/Detail Filtering Across Two Pages (VB)
  - Master/Detail Using a Selectable Master GridView with a Details DetailView (VB)
- Custom Formatting
  - Custom Formatting Based Upon Data (C#)
  - Using TemplateFields in the GridView Control (C#)
  - Using TemplateFields in the DetailsView Control (C#)
  - FormView's Templates (C#)
  - Displaying Summary Information in the GridView's Footer (C#)
  - Custom Formatting Based Upon Data (VB)
  - Using TemplateFields in the GridView Control (VB)
  - Using TemplateFields in the DetailsView Control (VB)
  - FormView's Templates (VB)
  - Displaying Summary Information in the GridView's Footer (VB)
- Editing, Inserting, and Deleting Data
  - Overview of Inserting, Updating, and Deleting Data (C#)
  - Examining the Events Associated with Inserting, Updating, and Deleting (C#)
  - Handling BLL- and DAL-Level Exceptions in an ASP.NET Page (C#)
  - Adding Validation Controls to the Editing and Inserting Interfaces (C#)
  - Customizing the Data Modification Interface (C#)
  - Implementing Optimistic Concurrency (C#)
  - Adding Client-Side Confirmation When Deleting (C#)
  - Limiting Data Modification Functionality Based on the User (C#)
  - Overview of Inserting, Updating, and Deleting Data (VB)
  - Examining the Events Associated with Inserting, Updating, and Deleting (VB)
  - Handling BLL- and DAL-Level Exceptions in an ASP.NET Page (VB)
  - Adding Validation Controls to the Editing and Inserting Interfaces (VB)
  - Customizing the Data Modification Interface (VB)
  - Implementing Optimistic Concurrency (VB)
  - Adding Client-Side Confirmation When Deleting (VB)
  - Limiting Data Modification Functionality Based on the User (VB)
- Paging and Sorting
  - Paging and Sorting Report Data (C#)
  - Efficiently Paging Through Large Amounts of Data (C#)
  - Sorting Custom Paged Data (C#)
  - Creating a Customized Sorting User Interface (C#)
  - Paging and Sorting Report Data (VB)
  - Efficiently Paging Through Large Amounts of Data (VB)

- Sorting Custom Paged Data (VB)
- Creating a Customized Sorting User Interface (VB)
- Custom Button Actions
  - Adding and Responding to Buttons to a GridView (C#)
  - Adding and Responding to Buttons to a GridView (VB)
- Displaying Data with the DataList and Repeater
  - Displaying Data with the DataList and Repeater Controls (C#)
  - Formatting the DataList and Repeater Based Upon Data (C#)
  - Showing Multiple Records per Row with the DataList Control (C#)
  - Nested Data Web Controls (C#)
  - Displaying Data with the DataList and Repeater Controls (VB)
  - Formatting the DataList and Repeater Based Upon Data (VB)
  - Showing Multiple Records per Row with the DataList Control (VB)
  - Nested Data Web Controls (VB)
- Filtering Scenarios with the DataList and Repeater
  - Master/Detail Filtering With a DropDownList (C#)
  - Master/Detail Filtering Across Two Pages (C#)
  - Master/Detail Using a Bulleted List of Master Records with a Details DataList (C#)
  - Master/Detail Filtering With a DropDownList (VB)
  - Master/Detail Filtering Across Two Pages (VB)
  - Master/Detail Using a Bulleted List of Master Records with a Details DataList (VB)
- Editing and Deleting Data Through the DataList
  - Overview of Editing and Deleting Data in the DataList (C#)
  - Performing Batch Updates (C#)
  - Handling BLL- and DAL-Level Exceptions (C#)
  - Adding Validation Controls to the DataList's Editing Interface (C#)
  - Customizing the DataList's Editing Interface (C#)
  - Overview of Editing and Deleting Data in the DataList (VB)
  - Performing Batch Updates (VB)
  - Handling BLL- and DAL-Level Exceptions (VB)
  - Adding Validation Controls to the DataList's Editing Interface (VB)
  - Customizing the DataList's Editing Interface (VB)
- Paging and Sorting with the DataList and Repeater
  - Paging Report Data in a DataList or Repeater Control (C#)
  - Sorting Data in a DataList or Repeater Control (C#)
  - Paging Report Data in a DataList or Repeater Control (VB)
  - Sorting Data in a DataList or Repeater Control (VB)
- Custom Button Actions with the DataList and Repeater
  - Custom Buttons in the DataList and Repeater (C#)
  - Custom Buttons in the DataList and Repeater (VB)
- Accessing the Database Directly from an ASP.NET Page
  - Querying Data with the SqlDataSource Control (C#)
  - Using Parameterized Queries with the SqlDataSource (C#)

- Inserting, Updating, and Deleting Data with the SqlDataSource (C#)
- Implementing Optimistic Concurrency with the SqlDataSource (C#)
- Querying Data with the SqlDataSource Control (VB)
- Using Parameterized Queries with the SqlDataSource (VB)
- Inserting, Updating, and Deleting Data with the SqlDataSource (VB)
- Implementing Optimistic Concurrency with the SqlDataSource (VB)

- Enhancing the GridView
  - Adding a GridView Column of Radio Buttons (C#)
  - Adding a GridView Column of Checkboxes (C#)
  - Inserting a New Record from the GridView's Footer (C#)
  - Adding a GridView Column of Radio Buttons (VB)
  - Adding a GridView Column of Checkboxes (VB)
  - Inserting a New Record from the GridView's Footer (VB)
- Working with Binary Files
  - Uploading Files (C#)
  - Displaying Binary Data in the Data Web Controls (C#)
  - Including a File Upload Option When Adding a New Record (C#)
  - Updating and Deleting Existing Binary Data (C#)
  - Uploading Files (VB)
  - Displaying Binary Data in the Data Web Controls (VB)
  - Including a File Upload Option When Adding a New Record (VB)
  - Updating and Deleting Existing Binary Data (VB)
- Caching Data
  - Caching Data with the ObjectDataSource (C#)
  - Caching Data in the Architecture (C#)
  - Caching Data at Application Startup (C#)
  - Using SQL Cache Dependencies (C#)
  - Caching Data with the ObjectDataSource (VB)
  - Caching Data in the Architecture (VB)
  - Caching Data at Application Startup (VB)
  - Using SQL Cache Dependencies (VB)
- Database-Driven Site Maps
  - Building a Custom Database-Driven Site Map Provider (C#)
  - Building a Custom Database-Driven Site Map Provider (VB)
- Working with Batched Data
  - Wrapping Database Modifications within a Transaction (C#)
  - Batch Updating (C#)
  - Batch Deleting (C#)
  - Batch Inserting (C#)
  - Wrapping Database Modifications within a Transaction (VB)
  - Batch Updating (VB)
  - Batch Deleting (VB)
  - Batch Inserting (VB)
- Advanced Data Access Scenarios

- Creating New Stored Procedures for the Typed DataSet's TableAdapters (C#)
- Using Existing Stored Procedures for the Typed DataSet's TableAdapters (C#)
- Updating the TableAdapter to Use JOINs (C#)
- Adding Additional DataTable Columns (C#)
- Working with Computed Columns (C#)
- Configuring the Data Access Layer's Connection- and Command-Level Settings (C#)
- Protecting Connection Strings and Other Configuration Information (C#)
- Debugging Stored Procedures (C#)
- Creating Stored Procedures and User-Defined Functions with Managed Code (C#)
- Creating New Stored Procedures for the Typed DataSet's TableAdapters (VB)
- Using Existing Stored Procedures for the Typed DataSet's TableAdapters (VB)
- Updating the TableAdapter to Use JOINs (VB)
- Adding Additional DataTable Columns (VB)
- Working with Computed Columns (VB)
- Configuring the Data Access Layer's Connection- and Command-Level Settings (VB)
- Protecting Connection Strings and Other Configuration Information (VB)
- Debugging Stored Procedures (VB)
- Creating Stored Procedures and User-Defined Functions with Managed Code (VB)

- Older Versions - Security
  - ASP.NET 3.5 - Introduction to Security
    - Security Basics and ASP.NET Support (C#)
    - Overview of Forms Authentication (C#)
    - Forms Authentication Configuration and Advanced Topics (C#)
    - Security Basics and ASP.NET Support (VB)
    - Overview of Forms Authentication (VB)
    - Forms Authentication Configuration and Advanced Topics (VB)
  - ASP.NET 3.5 - Membership
    - Creating the Membership Schema in SQL Server (C#)
    - Creating User Accounts (C#)
    - Validating User Credentials Against the Membership User Store (C#)
    - User-Based Authorization (C#)
    - Storing Additional User Information (C#)
    - Creating the Membership Schema in SQL Server (VB)
    - Creating User Accounts (VB)
    - Validating User Credentials Against the Membership User Store (VB)
    - User-Based Authorization (VB)
    - Storing Additional User Information (VB)
  - ASP.NET 3.5 - Roles
    - Creating and Managing Roles (C#)
    - Assigning Roles to Users (C#)
    - Role-Based Authorization (C#)
    - Creating and Managing Roles (VB)
    - Assigning Roles to Users (VB)
    - Role-Based Authorization (VB)
  - ASP.NET 3.5 - Security Administration

    - Building an Interface to Select One User Account from Many (C#)
    - Recovering and Changing Passwords (C#)
    - Unlocking and Approving User Accounts (C#)
    - Building an Interface to Select One User Account from Many (VB)
    - Recovering and Changing Passwords (VB)
    - Unlocking and Approving User Accounts (VB)
  - Older Versions - ASP.NET 2.0
    - Improvements in Visual Studio 2005
    - ASP.NET 2.0 Page Model
    - Server Controls
    - Master Pages
    - Data Source Controls
    - Data Bound Controls
    - Caching
    - Membership
    - Configuration and Instrumentation
    - Profiles, Themes, and Web Parts
- Videos
  - How Do I?
    - Learn the Tips and Tricks of Experts
    - Create a Multi-Lingual Site with Localization
    - Make use of Caching
    - Make Greater use of Caching
    - Utilize Web Parts and Personalization
    - Customize my Site with Profiles and Themes
    - Secure my Site using Membership and Roles
    - Create a "Contact Us" Page
    - Create a Full-Featured Customer Login Portal
    - Use Master Pages and Site Navigation
    - Create Data-Driven Web Sites
    - Implement an HttpModule
    - Set Up the SQL Membership Provider
    - Implement URL Rewriting
    - Utilize the CSS Friendly Control Adapters
    - Handle Application Events using an HttpModule
    - Use Cascading Style Sheets for Web Page Layout
    - Create a Custom Membership Provider
    - Create a Custom Profile Provider
    - Implement Site Navigation in ASP.NET
    - Use Validation Controls in ASP.NET
    - Create a Custom User Control in ASP.NET
    - Implement Health Monitoring for an ASP.NET Application
    - Implement Tracing in an ASP.NET Web Site
    - Event Handlers in ASP.NET Web Pages and Controls
    - Implement a Custom Health Monitoring Event

- Use Control State to Persist Information for a Custom Web Server Control
- Read, Write, and Delete Cookies in ASP.NET
- Save and Load View State Information for a Custom Web Server Control
- Create and Use Application Level Events in ASP.NET
- Simple File Uploads in ASP.NET
- Multiple File Uploads in ASP.NET 2
- Multiple File Uploads in ASP.NET 1
- File Uploads with an AJAX Style Interface
- Work with Master Pages Declaratively and Programmatically
- Handle Events in Master and Content Pages
- Use the ObjectDataSource with a Data Access Class and Business Object
- Create User Selectable Themes for a Web Site
- Share Data Between Client JavaScript and Server Code Using a Hidden Field
- Add JavaScript to ASP.NET Page
- Add JavaScript Event Handlers to ASP.NET Server Controls
- Create and Call a Simple Web Service in ASP.NET
- Deploy a Web Site Copy Web Site Tool
- Create and Use a Custom Configuration Section in an ASP.NET Configuration File
- Configure an ASP.NET Web Site for a .NET Framework Version
- Use the ASP.NET IIS Registration Tool to Configure ASP.NET
- Extend and Customize an ASP.NET Server Control for a Specific Purpose
- Use the Ability to Create Groups with the ListView Control for Different Data
- Make HTML elements accessible from server side code
- Use Roles to Segment Functionality in an ASP.NET Web Site
- Work with Nested Master Pages to Create Standard Content Layouts
- Detect Browser Capabilities in ASP.NET Web Pages
- Map an ASP.NET Server Control to the Adaptor Used to Render It
- Pass Information From One Page to Another Using a Query String
- Use the aspnet_merge.exe Utility to Merge Assemblies
- Implement a Cross Page PostBack in ASP.NET
- Add Controls to an ASP.NET Web Page Programmatically
- Use a Hidden Field to Store and Manipulate Client-Side Information
- Use the Fixednames Option with the Compiler Utility for Maximum Flexibility
- Implement Command Buttons for Individual Items
- Use Skins with CSS for a Flexible and Maintainable ASP.NET Web Site
- Determine Whether to Use a Web Site or a Web Application Project
- Write Web Events to a SQL Server Database SqlWebEventProvider
- Use ASP.NET to Send Email from a Web Site
- Configure Email Notification for Health Monitoring on an ASP.NET Web Site
- Use a Visual Studio 2008 Web Deployment Project
- Send Templated Emails for Health Monitoring Events in ASP.NET
- Send Email Asynchronously with ASP.NET
- Embed an Image in an Email with ASP.NET
- Implement Error Handling when Sending Email with ASP.NET
- Create a Custom Provider for Logging Health Monitoring Events
- Create a Reusable Component for Sending Email to a Distribution List

- Precompile an ASP.NET Website
- Use the FtpWebRequest Class to Create an FTP Connection to a FTP Server
- Export Data to a Comma Delimited (CSV) File for an Application Like Excel
- Use the Reponse.Filter Property to Replace HTML in an ASP.NET Page
- Use OutputCache to Cache an ASP.NET Page
- Cache Portions of an ASP.NET Page
- Control the Caching of an ASP.NET Page Based Upon Custom Information
- Cache an ASP.NET Page Based Upon Information in the HTTP Header
- Synchronize Data in Secondary User Control(s) Based Upon User Selections
- Use the ASP.NET Cache Object to Cache Application Information
- Persist the State of a User Control During a Postback
- Add Client Script Event Handlers Controls on an ASP.NET Page
- Understand and Defend Against Script Injection Attacks in ASP.NET
- ASP.NET Data Access Technologies, an Intro to LINQ
- Create and Use Classes in JavaScript
- Create an Efficient and Standardized Approach for Implementing CSS Styles
- What is ASP.NET
- Choosing the Right Programming Model
- Build Your First ASP.NET Application with ASP.NET Web Forms
- Build Your First ASP.NET Application with ASP.NET MVC
- Best Learn ASP.NET Web Forms
- 5 Minute Intro to ASP.NET Web Forms
- Install ASP.NET and Tools

- Tailspin Spyworks
  - Tailspin Spyworks - Creating and Popular Products Control
  - Tailspin Spyworks - Implementing and Also Purchased Control
  - Tailspin Spyworks - Intro UI and EDM
  - Tailspin Spyworks - Directory Organization
  - Tailspin Spyworks - Category Menu
  - Tailspin Spyworks - Display the Product List
  - Tailspin Spyworks - Display Per Product Details
  - Tailspin Spyworks - Adding Items to the Shopping Cart
  - Tailspin Spyworks - Display Shopping Cart
  - Tailspin Spyworks - Update the Shopping Cart
  - Tailspin Spyworks - Migrate the Shopping Cart
  - Tailspin Spyworks - Final Check Out
  - Tailspin Spyworks - Adding User Product Reviews
  - Tailspin Spyworks - Displaying User Reviews
- Building 3.5 Applications
  - Intro to Visual Web Developer
  - IntelliSense
  - Intro to Web Forms
  - Page Layout
  - Page Lifecycle Events
  - Intro to ASP.NET Controls
  - Submit and Postback

- Application Level Objects
- Session Level Objects
- Debugging
- CSS
- MasterPages
- Intro to SOAP Based Web Services with Visual Web Developer 2008
- AJAX Style Services with Microsoft Visual Web Developer 2008
- Validation
- Login Controls

- Authentication
  - Using Basic Forms Authentication in ASP.NET
  - Change the Forms Authentication Properties
  - Setup and Use Cookie-less Authentication in an ASP.NET Application
  - ASP Forms Login Relocation
  - Forms Login Custom Key Configuration
  - Add Custom Data to the Authentication Method
  - Use Custom Principal Objects
  - Understanding ASP.NET Memberships
  - Configuring SQL To Work with Membership Schemas
  - Changing Membership Settings in the Default Membership Schema
  - Creating User Accounts with the Create User Wizard
  - Creating User Accounts Programmatically
  - Validating Users Manually
  - Validating Users with the Login Control
  - Adding Users to Your Membership System
  - Logging Users Into Your Membership System
  - Implement the Registration Verification Pattern
  - Simple Web Service Authentication
  - Creating Inactive Users
  - SQL Injection Defense
- SQL 2005
  - What is a Database
  - Understanding Database Tables and Records
  - More about Column Data Types and Other Properties
  - Designing Relational Database Tables
  - Manipulating Database Data
  - More Structured Query Language
  - Understanding Security and Network Connectivity
  - Connecting your Web Application to SQL Server 2005 Express Edition
  - Using SQL Server Management Studio
  - Getting Started with Reporting Services
  - Building and Customizing Reports in Business Intelligence Development Studio
  - Creating and Using Stored Procedures
  - Enabling Full-Text Search in your Text Data
- Visual Studio 2005

- Intro to Unit Testing with Team System
- Intro to Testing Web Applications with Team System
- Intro to Load Testing Web Applications with Team System
- Intro to Manual Testing with Team System
- Intro to Managing and Running Tests with Team System
- Measuring the Business Value of AJAX
- Code Coverage of Automated Tests
- Custom Extraction Rules and Coded Web Tests
- Effects of Caching
- Load Test Agent
- Effects of ViewState
- Integrate Defect Tracking with Testing
- Create My Own Bug Work Item
- Write Code More Quickly with Unit Tests
- Practice Test-Driven Development
- Load Test a Web Application
- Tune Web Application Performance with Profiling
- Set Up Distributed Load Testing for High Volume Tests
- Enforce Coding Standards with Code Analysis
- Use Generic Tests
- Publish and Analyze Test Results
- Discover Application Changes Prior to Deployment
- Implement Continuous Integration with Team Foundation
- Automate Testing using Team Build
- Deploy a Web Application during a Team Build
- Run Unit Tests against a Deployed Database
- Enable Code Coverage and Profiling in Production Applications
- Web Deployment Projects
- Web Application Projects & Web Deployment Projects

- Migrating
  - [Intro to ASP.NET 2.0:] ASP.NET 2.0 Fundamentals
  - [Intro to ASP.NET 2.0:] User Interface Elements
  - [Migrating from] Classic ASP to ASP.NET
  - Intro to ASP.NET for JSP Developers: Welcome to ASP.NET 2.0
  - Intro to ASP.NET for JSP Developers: Building Applications
  - Intro to ASP.NET for ColdFusion Developers: Adding ASP.NET to Your Repertoire
  - Intro to ASP.NET for ColdFusion Developers: Building an ASP.NET Application
  - Interop between PHP and the Windows Platform
- Building 2.0 Applications
  - [Lesson 1:] Getting Started with Visual Web Developer Express
  - [Lesson 2:] Creating a Web Forms User Interface
  - [Lesson 3:] Understanding More About Events and Postback
  - [Lesson 4:] Understanding Web Application State
  - [Lesson 5:] Debugging and Tracing Your Website
  - [Lesson 6:] Working with Stylesheets and Master Pages
  - [Lesson 7:] Databinding to User Interface Controls

- [Lesson 8:] Working with the GridView and FormView
- Watch ASP.NET Development in Action

- General IIS Tips
  - Developing and Deploying In a Shared Hosting
  - Working with IIS7 Delegated Admin
  - Feature Specific Delegated Management
  - Troubleshooting Production ASP.NET Apps
  - Creating a Site with IIS7 Manager
  - Installing FTP7
  - Bit Rate Throttling
  - IIS7 Playlists
- General VS 2010 Tips
  - Code Optimized Profile
  - Code Search View Hierarchy
  - IntelliSense Smart Lists
  - Multi-Monitor Support
  - New Web Project Template
  - New Multi-Targeting
  - Websites Instead of Web Projects
  - Snippets IntelliSense
- Ajax Control Toolkit
  - Get Started with the ASP.NET AJAX Control Toolkit
  - ASP.NET AJAX CascadingDropDown Control Extender
  - ASP.NET AJAX TextBoxWatermark Control Extender
  - ASP.NET AJAX Popup Control Extender
  - ASP.NET AJAX ModalPopup Extender Control
  - ASP.NET AJAX AlwaysVisible Control Extender
  - ASP.NET AJAX Accordion Control
  - ASP.NET AJAX Collapsable Panel Extender
  - ASP.NET AJAX Draggable Panel Extender
  - ASP.NET AJAX DynamicPopulate Extender
  - ASP.NET AJAX FilteredTextbox Extender
  - ASP.NET AJAX HoverMenu Extender
  - ASP.NET AJAX ToggleButton Extender
  - ASP.NET AJAX DropShadow Extender
  - ASP.NET AJAX PasswordStrength Extender
  - Get Started with the ASP.NET AJAX Animation Extender Control
  - ASP.NET AJAX ConfirmButton Extender
  - ASP.NET AJAX Slider Control
  - ASP.NET AJAX AutoComplete Control
  - Configure the ASP.NET AJAX Calendar Control
  - ASP.NET AJAX DropDown Control
  - ASP.NET AJAX MaskedEdit Controls
  - ASP.NET AJAX MutuallyExclusive Checkbox Extender
  - ASP.NET AJAX NoBot Control

- ASP.NET AJAX ListSearch Extender
- PagingBulletedList Extender Control
- NumericUpDown Extender Control
- ASP.NET AJAX ValidatorCallout Extender
- ASP.NET AJAX ResizableControl Extender
- ASP.NET AJAX Tabs Control
- ASP.NET AJAX SlideShow Extender
- ASP.NET AJAX UpdatePanelAnimation Extender
- AJAX Toolkit Reorder Control
- Utilize the AJAX Rating Control in the ASP.NET Toolkit
- Control Extenders
- Color Picker
- Combo Box
- Editor Control
- Editor Control Custom
- Create a New Custom Extender

- ASP.NET AJAX
  - Get Started with ASP.NET AJAX
  - Implement Dynamic Partial-Page Updates with ASP.NET AJAX
  - Make Client-Side Network Callbacks with ASP.NET AJAX
  - Add ASP.NET AJAX Features to an Existing Web Application
  - ASP.NET AJAX Enable an Existing Web Service
  - ASP.NET AJAX Client Library Controls
  - Use an ASP.NET AJAX ScriptManagerProxy
  - ASP.NET AJAX RoundedCorners Extender
  - ASP.NET AJAX Timer Control
  - Implement the Predictive Fetch Pattern for AJAX
  - Implement the AJAX Paging Pattern
  - Implement the AJAX Incremental Page Display Pattern
  - Implement the Incremental Page Display Pattern using HTTP GET and POST
  - ASP.NET AJAX UpdateProgress Control
  - ASP.NET AJAX History Control
  - Implement the AJAX After Processing Pattern
  - Update Multiple Regions of a Page with ASP.NET AJAX
  - Choose Between Methods of AJAX Page Updates
  - Use Other JavaScript User Interface Libraries with ASP.NET AJAX
  - Use the ASP.NET AJAX Profile Services
  - Debug ASP.NET AJAX Applications Using Visual Studio 2005
  - Build a Custom ASP.NET AJAX Server Control
  - Use JavaScript to Refresh an ASP.NET AJAX UpdatePanel
  - Determine Whether an Asynchronous Postback has Occurred
  - Use the Conditional UpdateMode of the UpdatePanel
  - Implement the Persistent Communications Pattern with the UpdatePanel
  - Localize an ASP.NET AJAX Application
  - Implement the Persistent Communications Pattern using Web Services
  - Trigger an UpdatePanel Refresh from a DropDownList Control

- Create an ASP.NET AJAX Extender from Scratch
- Build Custom Server Controls that Work With or Without ASP.NET AJAX
- Associate AJAX Client Behavior with an ASP.NET Server Control
- Retrieve Values From Server Side AJAX Controls
- Two Simple Techniques for Triggering Updates to Update Panels
- Use ASP.NET AJAX Cascading Drop Down Control to Access a Database
- Implement Infinite Data Patterns in AJAX
- Basic ASP.NET Authentication in an AJAX Enabled Application
- Dynamically Change CSS ASP.NET AJAX UpdatePanel
- Dynamically Add Controls to a Web Page
- Set Up Your Development Environment for ASP.NET 3.5
- Set Up Your Development Environment for ASP.NET 2.0
- Customize Error Handling for the ASP.NET AJAX UpdatePanel
- Use ASP.NET AJAX Client Templates

- Data Access
  - ADO.NET Data Services
    - Data Access Layers in ASP.NET Applications
    - Manually Bind a Dataset to a Datagrid
    - Work with Datasets and Filters from an ASP Application
  - How Do I with LINQ
    - LINQ to SQL: Overview
    - LINQ to SQL: Data Model
    - LINQ to SQL: Querying the Database
    - LINQ to SQL: Updating the Database
    - LINQ to SQL: LinqDataSource
    - LINQ to SQL: Custom LinqDataSource
    - LINQ to SQL: Using Stored Procedures
    - LINQ to SQL: Updating with Stored Procedures
    - LINQ to SQL: Executing Arbitrary SQL
  - LINQ Videos from the VB Team
    - Get Started with LINQ
    - Perform Group and Aggregate Queries
    - Upgrade Visual Basic Projects to Enable LINQ
    - Get Started with LINQ to XML
    - Enable XML IntelliSense and Use XML Namespaces
    - Create XML Documents from SQL Data
    - Create Excel Spreadsheets using LINQ to XML
- ASP.NET Dynamic Data
  - Your First Scaffold and What is Dynamic Data
  - Enable Inline GridView Editing
  - Change how my Fields render
  - Handle Business Logic Exceptions
  - Make Custom Pages
  - Display Unknown datatypes

- Use a DynamicControl in ListView and DetailsView Controls
- Getting Started with Dynamic Data
- Begin Editing the Templates in ASP.NET Dynamic Data Applications
- Begin Modifying Dynamic Data Applications with URL Routing
- Enable In-Line Editing in ASP.NET Dynamic Data Applications
- Enable Table Specific Routing in Dynamic Data Applications
- Use Attribute Validation in ASP.NET Dynamic Data Applications
- Implement Custom Field Validation with Imperative Logic in VB or C#
- Remove Columns From Your DynamicData Data Grids
- Create Table Specific Custom Forms in an ASP.NET Dynamic Data Application
- ASP.NET Dynamic Data Custom Form Formatting

- ASP.NET 3.5
  - How Do I
    - Create a Master Page in Visual Studio 2008
    - Create Nested Master Page in Visual Studio 2008
    - Cascading Style Sheets in Visual Studio 2008
    - Working with Visual Studio 2008 .NET Framework
    - Adding Elements to a CSS File and Create New CSS on the Fly
    - Advance Cascading Style Sheet Features and Management
    - Converting a .NET 2.0 Windows Forms Application to .NET 3.5
    - Get Started with the Entity Framework
    - Use the New Entity Data Source
    - Serialize a Graph with the Entity Framework
    - Use MSBuild to Automate the ASP.NET Compiler and Merge Utilities
  - ASP.NET Ajax
    - ASP.NET AJAX Support in Visual Studio 2008
    - Adding AJAX Functionality to an Existing ASP.NET Page
    - Creating and Using an AJAX-enabled Web Service in a Web Site
    - ASP.NET AJAX: A demonstration of ASP.NET AJAX
    - ADO.NET Data Services with ASP.NET AJAX Support
    - Intro to ASP.NET Ajax History
    - Using Script Combining to improve AJAX performance
  - Data Controls
    - ListView Control
    - DataPager Control
  - Visual Studio Designer
    - Visual Studio 2008 and Nested Masterpages
  - Visual Studio Editor
    - New Designer Support in Visual Studio 2008
    - JavaScript Intellisense Support in Visual Studio 2008
    - JavaScript Debugging in Visual Studio 2008
    - Multi Targeting Support in Visual Studio 2008
    - IntelliSense for Jscript and ASP.NET AJAX
    - Quick Tour of the Visual Studio 2008 Integrated Development Environment

- Creating and Modifying a CSS File
- jQuery
  - Make Ajax Calls Using jQuery
- .NET 4
  - Ajax
    - Imperative JavaScript Syntax for Microsoft Client Side Controls
    - ScriptLoader
    - JQuery Syntax for Microsoft Ajax
    - AJAX Data Templates
  - Chart
    - Chart Control
    - Introducing the New Chart Control in Visual Studio 2010
  - Core
    - Auto Start
    - Clean Web.Config Files
    - Predictable Client IDs
    - HtmlEncoder Utility Method
  - Data
    - Persistent GridView Row Selection
  - Markup Control
    - New Rendering Option for Check Box Lists and Radio Button Lists
    - Table Free Templated Controls
    - Tableless Menu Control
    - Hidden Field Divs
    - Disabled Control Styling
  - Performance
    - Selective View State
    - Easy State Compression
    - Use the ViewStateMode Property for Managing ViewState
  - Routing
    - Permanent Redirect
    - Imperative WebForms Routing
    - Declarative WebForms Routing
    - Outbound WebForms Routing
    - Use Routing with ASP.NET Web Forms
    - Work with URLs in ASP.NET Routing
  - SEO
    - Dynamic Metadata
- ASP.NET 4.5
  - Strongly Typed Data Controls
  - Model Binding Part 1 - Selecting Data
  - Model Binding Part 2 - Filtering

    - Model Binding Part 3 - Updating
    - ASP.NET 4.5 Web Forms Model Binding
    - ASP.NET 4.5 Web Forms Strong Typed Data Controls
    - Bundling and Minification
    - Getting Started with the Next Version of ASP.NET
    - ASP.NET and Web Tools 2012.2
- Introduction to ASP.NET Web Forms
- Web Forms Video Training from Pluralsight

ASP.NET Web Forms Guidance

2/20/2020 • 16 minutes to read • Edit Online

- Getting Started
  - Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013
    - Getting Started with Web Forms and Visual Studio
    - Create the Project
    - Create the Data Access Layer
    - UI and Navigation
    - Display Data Items and Details
    - Shopping Cart
    - Checkout and Payment with PayPal
    - Membership and Administration
    - URL Routing
    - ASP.NET Error Handling
  - Creating a Basic Web Forms Page in Visual Studio 2013
  - Code Editing ASP.NET Web Forms in Visual Studio 2013
  - Using Page Inspector for Visual Studio 2012 in ASP.NET Web Forms
  - Visual Studio 2012 Hands On Labs
    - What's New in ASP.NET and Web Development in Visual Studio 2012
    - What's New in Web Forms in ASP.NET 4.5
    - Using Page Inspector in Visual Studio 2012
- Working with Data
  - Model Binding and Web Forms in Visual Studio 2013
    - Retrieving and Displaying Data
    - Updating, Deleting, and Creating Data
    - Sorting, Paging, and Filtering Data
    - Integrating JQuery UI Datepicker
    - Using Query String Values to Filter Data
    - Adding Business Logic Layer
- Security, Authentication, and Authorization
  - Create a secure ASP.NET Web Forms app with user registration, email confirmation and password reset (C#)
  - Create an ASP.NET Web Forms app with SMS Two-Factor Authentication (C#)
- Performance
  - Using Asynchronous Methods in ASP.NET 4.5
- Deployment
  - ASP.NET Web Deployment using Visual Studio
    - Introduction

- Preparing for Database Deployment
- Web.config File Transformations
- Project Properties
- Deploying to Test
- Setting Folder Permissions
- Deploying to Production
- Deploying a Code Update
- Deploying a Database Update
- Command Line Deployment
- Deploying Extra Files
- Troubleshooting

- ASP.NET 4 - Enterprise Deployment Introduction
  - Deploying Web Applications in Enterprise Scenarios using Visual Studio 2010
  - Enterprise Web Deployment: Scenario Overview
  - Application Lifecycle Management: From Development to Production
- ASP.NET 4 - Enterprise Deployment Series 1
  - Web Deployment in the Enterprise
  - The Contact Manager Solution
  - Setting Up the Contact Manager Solution
  - Understanding the Project File
  - Understanding the Build Process
  - Building and Packaging Web Application Projects
  - Configuring Parameters for Web Package Deployment
  - Deploying Web Packages
  - Deploying Database Projects
  - Creating and Running a Deployment Command File
  - Manually Installing Web Packages
- ASP.NET 4 - Enterprise Deployment Series 2 Configuring Servers
  - Configuring Server Environments for Web Deployment
  - Choosing the Right Approach to Web Deployment
  - Scenario: Configuring a Test Environment for Web Deployment
  - Scenario: Configuring a Staging Environment for Web Deployment
  - Scenario: Configuring a Production Environment for Web Deployment
  - Configuring a Web Server for Web Deploy Publishing (Remote Agent)
  - Configuring a Web Server for Web Deploy Publishing (Web Deploy Handler)
  - Configuring a Web Server for Web Deploy Publishing (Offline Deployment)
  - Configuring a Database Server for Web Deploy Publishing
  - Creating a Server Farm with the Web Farm Framework
  - Configuring Deployment Properties for a Target Environment
- ASP.NET 4 - Enterprise Deployment Series 3 Configuring TFS
  - Configuring Team Foundation Server for Web Deployment
  - Creating a Team Project in TFS
  - Adding Content to Source Control
  - Configuring a TFS Build Server for Web Deployment
  - Creating a Build Definition That Supports Deployment

    - Deploying a Specific Build
    - Configuring Permissions for Team Build Deployment
  - ASP.NET 4 - Enterprise Deployment Series 4 Advanced Topics
    - Advanced Enterprise Web Deployment
    - Performing a "What If" Deployment
    - Customizing Database Deployments for Multiple Environments
    - Deploying Database Role Memberships to Test Environments
    - Deploying Membership Databases to Enterprise Environments
    - Excluding Files and Folders from Deployment
    - Taking Web Applications Offline with Web Deploy
    - Running Windows PowerShell Scripts from MSBuild Project Files
    - Troubleshooting the Packaging Process
- Advanced
  - ASP.NET Web Forms Connection Resiliency and Command Interception
- Older Versions - Getting Started
  - ASP.NET 4 - Tailspin Spyworks
    - Part 1: File-> New Project
    - Part 2: Data Access Layer
    - Part 3: Layout and Category Menu
    - Part 4: Listing Products
    - Part 5: Business Logic
    - Part 6: ASP.NET Membership
    - Part 7: Adding Features
    - Part 8: Final Pages, Exception Handling, and Conclusion
  - ASP.NET 4 - Getting Started with Entity Framework 4 and Web Forms
    - Getting Started with Entity Framework 4.0 Database First
    - Getting Started with Entity Framework 4.0 Database First - Part 2
    - Getting Started with Entity Framework 4.0 Database First - Part 3
    - Getting Started with Entity Framework 4.0 Database First - Part 4
    - Getting Started with Entity Framework 4.0 Database First - Part 5
    - Getting Started with Entity Framework 4.0 Database First - Part 6
    - Getting Started with Entity Framework 4.0 Database First - Part 7
    - Getting Started with Entity Framework 4.0 Database First - Part 8
  - ASP.NET 4 - Continuing with Entity Framework 4 and Web Forms
    - Entity Framework 4.0 and the ObjectDataSource Control, Part 1: Getting Started
    - Entity Framework 4.0 and the ObjectDataSource Control, Part 2: Adding a Business Logic Layer and Unit Tests
    - Entity Framework 4.0 and the ObjectDataSource Control, Part 3: Sorting and Filtering
    - Handling Concurrency with the Entity Framework 4.0 in ASP.NET 4
    - Maximizing Performance with the Entity Framework 4.0 in ASP.NET 4
    - What's New in the Entity Framework 4.0
  - ASP.NET 4 - Web Deployment with SQL Server Compact using Visual Studio
    - Visual Studio Web Deployment with SQL Server Compact - Introduction
    - Visual Studio Web Deployment with SQL Server Compact- Deploying SQL Server Compact

Databases
    - Visual Studio Web Deployment with SQL Server Compact - Web.Config File Transformations
    - Visual Studio Web Deployment with SQL Server Compact - Configuring Project Properties
    - Visual Studio Web Deployment with SQL Server Compact - Deploying to IIS as a Test Environment
    - Visual Studio Web Deployment with SQL Server Compact - Setting Folder Permissions
    - Visual Studio Web Deployment with SQL Server Compact - Deploying to the Production Environment
    - Visual Studio Web Deployment with SQL Server Compact - Deploying a Code-Only Update
    - Visual Studio Web Deployment with SQL Server Compact - Deploying a Database Update
    - Visual Studio Web Deployment with SQL Server Compact - Migrating to SQL Server
    - Visual Studio Web Deployment with SQL Server Compact - Deploying a SQL Server Database Update
    - Visual Studio Web Deployment with SQL Server Compact - Troubleshooting
- ASP.NET 3.5 - AJAX
    - Understanding Partial Page Updates with ASP.NET AJAX
    - Understanding ASP.NET AJAX UpdatePanel Triggers
    - Understanding ASP.NET AJAX Authentication and Profile Application Services
    - Understanding ASP.NET AJAX Localization
    - Understanding ASP.NET AJAX Web Services
    - Understanding ASP.NET AJAX Debugging Capabilities
- ASP.NET 3.5 - Web Forms Master Pages
    - Creating a Site-Wide Layout Using Master Pages (C#)
    - Multiple ContentPlaceHolders and Default Content (C#)
    - Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page (C#)
    - URLs in Master Pages (C#)
    - Control ID Naming in Content Pages (C#)
    - Interacting with the Master Page from the Content Page (C#)
    - Interacting with the Content Page from the Master Page (C#)
    - Master Pages and ASP.NET AJAX (C#)
    - Specifying the Master Page Programmatically (C#)
    - Nested Master Pages (C#)
    - Creating a Site-Wide Layout Using Master Pages (VB)
    - Multiple ContentPlaceHolders and Default Content (VB)
    - Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page (VB)
    - URLs in Master Pages (VB)
    - Control ID Naming in Content Pages (VB)
    - Interacting with the Master Page from the Content Page (VB)
    - Interacting with the Content Page from the Master Page (VB)
    - Master Pages and ASP.NET AJAX (VB)
    - Specifying the Master Page Programmatically (VB)
    - Nested Master Pages (VB)
- Deploying Web Site Projects in Visual Studio 2008 and earlier
    - ASP.NET Hosting Options (C#)
    - Determining What Files Need to Be Deployed (C#)
    - Deploying Your Site Using an FTP Client (C#)

    - Deploying Your Site Using Visual Studio (C#)
    - Common Configuration Differences Between Development and Production (C#)
    - Core Differences Between IIS and the ASP.NET Development Server (C#)
    - Deploying a Database (C#)
    - Configuring the Production Web Application to Use the Production Database (C#)
    - Configuring a Website that Uses Application Services (C#)
    - Strategies for Database Development and Deployment (C#)
    - Displaying a Custom Error Page (C#)
    - Processing Unhandled Exceptions (C#)
    - Logging Error Details with ASP.NET Health Monitoring (C#)
    - Logging Error Details with ELMAH (C#)
    - Precompiling Your Website (C#)
    - Users and Roles On Production Website (C#)
    - ASP.NET Hosting Options (VB)
    - Determining What Files Need to Be Deployed (VB)
    - Deploying Your Site Using an FTP Client (VB)
    - Deploying Your Site Using Visual Studio (VB)
    - Common Configuration Differences Between Development and Production (VB)
    - Core Differences Between IIS and the ASP.NET Development Server (VB)
    - Deploying a Database (VB)
    - Configuring the Production Web Application to Use the Production Database (VB)
    - Configuring a Website that Uses Application Services (VB)
    - Strategies for Database Development and Deployment (VB)
    - Displaying a Custom Error Page (VB)
    - Processing Unhandled Exceptions (VB)
    - Logging Error Details with ASP.NET Health Monitoring (VB)
    - Logging Error Details with ELMAH (VB)
    - Precompiling Your Website (VB)
    - Users and Roles On Production Website (VB)
- Older Versions - Ajax Control Toolkit
  - Getting Started
    - Get Started with the AJAX Control Toolkit (C#)
    - Using AJAX Control Toolkit Controls and Control Extenders (C#)
    - Creating a Custom AJAX Control Toolkit Control Extender (C#)
    - Get Started with the AJAX Control Toolkit (VB)
    - Using AJAX Control Toolkit Controls and Control Extenders (VB)
    - Creating a Custom AJAX Control Toolkit Control Extender (VB)
  - Accordion
    - Databinding to an Accordion (C#)
    - Dynamically Adding Accordion Pane (C#)
    - Databinding to an Accordion (VB)
    - Dynamically Adding Accordion Pane (VB)
  - Animation
    - Adding Animation to a Control (C#)
    - Executing Several Animations at Same Time (C#)

    - Executing Several Animations after Each Other (C#)
    - Animation Depending On a Condition (C#)
    - Picking One Animation Out Of a List (C#)
    - Animating in Response To User Interaction (C#)
    - Disabling Actions during Animation (C#)
    - Triggering an Animation in another Control (C#)
    - Modifying Animations From Server Side (C#)
    - Executing Animations Using Client-Side Code (C#)
    - Changing an Animation Using Client-Side Code (C#)
    - Animating an UpdatePanel Control (C#)
    - Dynamically Controlling UpdatePanel Animations (C#)
    - Adding Animation to a Control (VB)
    - Executing Several Animations at Same Time (VB)
    - Executing Several Animations after Each Other (VB)
    - Animation Depending On a Condition (VB)
    - Picking One Animation Out Of a List (VB)
    - Animating in Response To User Interaction (VB)
    - Disabling Actions during Animation (VB)
    - Triggering an Animation in another Control (VB)
    - Modifying Animations From Server Side (VB)
    - Executing Animations Using Client-Side Code (VB)
    - Changing an Animation Using Client-Side Code (VB)
    - Animating an UpdatePanel Control (VB)
    - Dynamically Controlling UpdatePanel Animations (VB)
- CascadingDropdown
    - Filling a List Using CascadingDropDown (C#)
    - Using CascadingDropDown with a Database (C#)
    - Presetting List Entries with CascadingDropDown (C#)
    - Using Auto-Postback with CascadingDropDown (C#)
    - Filling a List Using CascadingDropDown (VB)
    - Using CascadingDropDown with a Database (VB)
    - Presetting List Entries with CascadingDropDown (VB)
    - Using Auto-Postback with CascadingDropDown (VB)
- CollapsiblePanel
    - Collapsing and Expanding a Panel from JavaScript (C#)
    - Collapsing and Expanding a Panel from JavaScript (VB)
- ColorPicker
    - ColorPicker Control Extender (C#)
    - ColorPicker Control Extender (VB)
- ComboBox
    - ComboBox Control (C#)
    - ComboBox Control (VB)
- ConfirmButton
    - Using a ConfirmButton In a Repeater (C#)

- Using a ConfirmButton In a Repeater (VB)
- DropShadow
  - Adjusting the Z-Index of a DropShadow (C#)
  - Manipulating DropShadow Properties from Client Code (C#)
  - Adjusting the Z-Index of a DropShadow (VB)
  - Manipulating DropShadow Properties from Client Code (VB)
- DynamicPopulate
  - Dynamically Populating a Control (C#)
  - Dynamically Populating a Control Using JavaScript Code (C#)
  - Using DynamicPopulate with a User Control And JavaScript (C#)
  - Dynamically Populating a Control (VB)
  - Dynamically Populating a Control Using JavaScript Code (VB)
  - Using DynamicPopulate with a User Control And JavaScript (VB)
- FilteredTextBox
  - Allowing Only Certain Characters in a Text Box (C#)
  - Allowing Only Certain Characters in a Text Box (VB)
- HoverMenu
  - Using HoverMenu with a Repeater Control (C#)
  - Using HoverMenu with a Repeater Control (VB)
- HTMLEditor
  - HTML Editor Control (C#)
  - HTML Editor Control (VB)
- ModalPopup
  - Launching a Modal Popup Window from Server Code (C#)
  - Using ModalPopup with a Repeater Control (C#)
  - Handling Postbacks from a ModalPopup (C#)
  - Positioning a ModalPopup (C#)
  - Launching a Modal Popup Window from Server Code (VB)
  - Using ModalPopup with a Repeater Control (VB)
  - Handling Postbacks from a ModalPopup (VB)
  - Positioning a ModalPopup (VB)
- MutuallyExclusiveCheckbox
  - Creating Mutually Exclusive Checkboxes (C#)
  - Creating Mutually Exclusive Checkboxes (VB)
- NoBot
  - Fighting Bots (C#)
  - Fighting Bots (VB)
- NumericUpDown
  - Creating a Numeric Up/Down Control with a Web Service Backend (C#)
  - Creating a Numeric Up/Down Control with a Web Service Backend (VB)
- PasswordStrength
  - Testing the Strength of a Password (C#)

    - Testing the Strength of a Password (VB)
  - Popup
    - Using Multiple Popup Controls (C#)
    - Handling Postbacks from A Popup Control With an UpdatePanel (C#)
    - Handling Postbacks from A Popup Control Without an UpdatePanel (C#)
    - Using Multiple Popup Controls (VB)
    - Handling Postbacks from A Popup Control With an UpdatePanel (VB)
    - Handling Postbacks from A Popup Control Without an UpdatePanel (VB)
  - Rating
    - Creating a Rating Control (C#)
    - Creating a Rating Control (VB)
  - ReorderList
    - Using Postbacks with ReorderList (C#)
    - Drag and Drop via ReorderList (C#)
    - Using Postbacks with ReorderList (VB)
    - Drag and Drop via ReorderList (VB)
  - Slider
    - Slider Control With Auto-Postback (C#)
    - Databinding the Slider Control (C#)
    - Slider Control With Auto-Postback (VB)
    - Databinding the Slider Control (VB)
  - TextBoxWatermark
    - Using TextBoxWatermark in a FormView (C#)
    - Using TextBoxWatermark With Validation Controls (C#)
    - Using TextBoxWatermark in a FormView (VB)
    - Using TextBoxWatermark With Validation Controls (VB)
- Older Versions - Data Access
  - Introduction
    - Creating a Data Access Layer (C#)
    - Creating a Business Logic Layer (C#)
    - Master Pages and Site Navigation (C#)
    - Creating a Data Access Layer (VB)
    - Creating a Business Logic Layer (VB)
    - Master Pages and Site Navigation (VB)
  - Basic Reporting
    - Displaying Data With the ObjectDataSource (C#)
    - Declarative Parameters (C#)
    - Programmatically Setting the ObjectDataSource's Parameter Values (C#)
    - Displaying Data With the ObjectDataSource (VB)
    - Declarative Parameters (VB)
    - Programmatically Setting the ObjectDataSource's Parameter Values (VB)
  - Master/Detail
    - Master/Detail Filtering With a DropDownList (C#)

- Master/Detail Filtering With Two DropDownLists (C#)
- Master/Detail Filtering Across Two Pages (C#)
- Master/Detail Using a Selectable Master GridView with a Details DetailView (C#)
- Master/Detail Filtering With a DropDownList (VB)
- Master/Detail Filtering With Two DropDownLists (VB)
- Master/Detail Filtering Across Two Pages (VB)
- Master/Detail Using a Selectable Master GridView with a Details DetailView (VB)

- Custom Formatting
  - Custom Formatting Based Upon Data (C#)
  - Using TemplateFields in the GridView Control (C#)
  - Using TemplateFields in the DetailsView Control (C#)
  - FormView's Templates (C#)
  - Displaying Summary Information in the GridView's Footer (C#)
  - Custom Formatting Based Upon Data (VB)
  - Using TemplateFields in the GridView Control (VB)
  - Using TemplateFields in the DetailsView Control (VB)
  - FormView's Templates (VB)
  - Displaying Summary Information in the GridView's Footer (VB)
- Editing, Inserting, and Deleting Data
  - Overview of Inserting, Updating, and Deleting Data (C#)
  - Examining the Events Associated with Inserting, Updating, and Deleting (C#)
  - Handling BLL- and DAL-Level Exceptions in an ASP.NET Page (C#)
  - Adding Validation Controls to the Editing and Inserting Interfaces (C#)
  - Customizing the Data Modification Interface (C#)
  - Implementing Optimistic Concurrency (C#)
  - Adding Client-Side Confirmation When Deleting (C#)
  - Limiting Data Modification Functionality Based on the User (C#)
  - Overview of Inserting, Updating, and Deleting Data (VB)
  - Examining the Events Associated with Inserting, Updating, and Deleting (VB)
  - Handling BLL- and DAL-Level Exceptions in an ASP.NET Page (VB)
  - Adding Validation Controls to the Editing and Inserting Interfaces (VB)
  - Customizing the Data Modification Interface (VB)
  - Implementing Optimistic Concurrency (VB)
  - Adding Client-Side Confirmation When Deleting (VB)
  - Limiting Data Modification Functionality Based on the User (VB)
- Paging and Sorting
  - Paging and Sorting Report Data (C#)
  - Efficiently Paging Through Large Amounts of Data (C#)
  - Sorting Custom Paged Data (C#)
  - Creating a Customized Sorting User Interface (C#)
  - Paging and Sorting Report Data (VB)
  - Efficiently Paging Through Large Amounts of Data (VB)
  - Sorting Custom Paged Data (VB)
  - Creating a Customized Sorting User Interface (VB)
- Custom Button Actions

    - Adding and Responding to Buttons to a GridView (C#)
    - Adding and Responding to Buttons to a GridView (VB)
- Displaying Data with the DataList and Repeater
    - Displaying Data with the DataList and Repeater Controls (C#)
    - Formatting the DataList and Repeater Based Upon Data (C#)
    - Showing Multiple Records per Row with the DataList Control (C#)
    - Nested Data Web Controls (C#)
    - Displaying Data with the DataList and Repeater Controls (VB)
    - Formatting the DataList and Repeater Based Upon Data (VB)
    - Showing Multiple Records per Row with the DataList Control (VB)
    - Nested Data Web Controls (VB)
- Filtering Scenarios with the DataList and Repeater
    - Master/Detail Filtering With a DropDownList (C#)
    - Master/Detail Filtering Across Two Pages (C#)
    - Master/Detail Using a Bulleted List of Master Records with a Details DataList (C#)
    - Master/Detail Filtering With a DropDownList (VB)
    - Master/Detail Filtering Across Two Pages (VB)
    - Master/Detail Using a Bulleted List of Master Records with a Details DataList (VB)
- Editing and Deleting Data Through the DataList
    - Overview of Editing and Deleting Data in the DataList (C#)
    - Performing Batch Updates (C#)
    - Handling BLL- and DAL-Level Exceptions (C#)
    - Adding Validation Controls to the DataList's Editing Interface (C#)
    - Customizing the DataList's Editing Interface (C#)
    - Overview of Editing and Deleting Data in the DataList (VB)
    - Performing Batch Updates (VB)
    - Handling BLL- and DAL-Level Exceptions (VB)
    - Adding Validation Controls to the DataList's Editing Interface (VB)
    - Customizing the DataList's Editing Interface (VB)
- Paging and Sorting with the DataList and Repeater
    - Paging Report Data in a DataList or Repeater Control (C#)
    - Sorting Data in a DataList or Repeater Control (C#)
    - Paging Report Data in a DataList or Repeater Control (VB)
    - Sorting Data in a DataList or Repeater Control (VB)
- Custom Button Actions with the DataList and Repeater
    - Custom Buttons in the DataList and Repeater (C#)
    - Custom Buttons in the DataList and Repeater (VB)
- Accessing the Database Directly from an ASP.NET Page
    - Querying Data with the SqlDataSource Control (C#)
    - Using Parameterized Queries with the SqlDataSource (C#)
    - Inserting, Updating, and Deleting Data with the SqlDataSource (C#)
    - Implementing Optimistic Concurrency with the SqlDataSource (C#)
    - Querying Data with the SqlDataSource Control (VB)
    - Using Parameterized Queries with the SqlDataSource (VB)

- Inserting, Updating, and Deleting Data with the SqlDataSource (VB)
- Implementing Optimistic Concurrency with the SqlDataSource (VB)
- Enhancing the GridView
  - Adding a GridView Column of Radio Buttons (C#)
  - Adding a GridView Column of Checkboxes (C#)
  - Inserting a New Record from the GridView's Footer (C#)
  - Adding a GridView Column of Radio Buttons (VB)
  - Adding a GridView Column of Checkboxes (VB)
  - Inserting a New Record from the GridView's Footer (VB)
- Working with Binary Files
  - Uploading Files (C#)
  - Displaying Binary Data in the Data Web Controls (C#)
  - Including a File Upload Option When Adding a New Record (C#)
  - Updating and Deleting Existing Binary Data (C#)
  - Uploading Files (VB)
  - Displaying Binary Data in the Data Web Controls (VB)
  - Including a File Upload Option When Adding a New Record (VB)
  - Updating and Deleting Existing Binary Data (VB)
- Caching Data
  - Caching Data with the ObjectDataSource (C#)
  - Caching Data in the Architecture (C#)
  - Caching Data at Application Startup (C#)
  - Using SQL Cache Dependencies (C#)
  - Caching Data with the ObjectDataSource (VB)
  - Caching Data in the Architecture (VB)
  - Caching Data at Application Startup (VB)
  - Using SQL Cache Dependencies (VB)
- Database-Driven Site Maps
  - Building a Custom Database-Driven Site Map Provider (C#)
  - Building a Custom Database-Driven Site Map Provider (VB)
- Working with Batched Data
  - Wrapping Database Modifications within a Transaction (C#)
  - Batch Updating (C#)
  - Batch Deleting (C#)
  - Batch Inserting (C#)
  - Wrapping Database Modifications within a Transaction (VB)
  - Batch Updating (VB)
  - Batch Deleting (VB)
  - Batch Inserting (VB)
- Advanced Data Access Scenarios
  - Creating New Stored Procedures for the Typed DataSet's TableAdapters (C#)
  - Using Existing Stored Procedures for the Typed DataSet's TableAdapters (C#)
  - Updating the TableAdapter to Use JOINs (C#)
  - Adding Additional DataTable Columns (C#)

    - Working with Computed Columns (C#)
    - Configuring the Data Access Layer's Connection- and Command-Level Settings (C#)
    - Protecting Connection Strings and Other Configuration Information (C#)
    - Debugging Stored Procedures (C#)
    - Creating Stored Procedures and User-Defined Functions with Managed Code (C#)
    - Creating New Stored Procedures for the Typed DataSet's TableAdapters (VB)
    - Using Existing Stored Procedures for the Typed DataSet's TableAdapters (VB)
    - Updating the TableAdapter to Use JOINs (VB)
    - Adding Additional DataTable Columns (VB)
    - Working with Computed Columns (VB)
    - Configuring the Data Access Layer's Connection- and Command-Level Settings (VB)
    - Protecting Connection Strings and Other Configuration Information (VB)
    - Debugging Stored Procedures (VB)
    - Creating Stored Procedures and User-Defined Functions with Managed Code (VB)
- Older Versions - Security
  - ASP.NET 3.5 - Introduction to Security
    - Security Basics and ASP.NET Support (C#)
    - Overview of Forms Authentication (C#)
    - Forms Authentication Configuration and Advanced Topics (C#)
    - Security Basics and ASP.NET Support (VB)
    - Overview of Forms Authentication (VB)
    - Forms Authentication Configuration and Advanced Topics (VB)
  - ASP.NET 3.5 - Membership
    - Creating the Membership Schema in SQL Server (C#)
    - Creating User Accounts (C#)
    - Validating User Credentials Against the Membership User Store (C#)
    - User-Based Authorization (C#)
    - Storing Additional User Information (C#)
    - Creating the Membership Schema in SQL Server (VB)
    - Creating User Accounts (VB)
    - Validating User Credentials Against the Membership User Store (VB)
    - User-Based Authorization (VB)
    - Storing Additional User Information (VB)
  - ASP.NET 3.5 - Roles
    - Creating and Managing Roles (C#)
    - Assigning Roles to Users (C#)
    - Role-Based Authorization (C#)
    - Creating and Managing Roles (VB)
    - Assigning Roles to Users (VB)
    - Role-Based Authorization (VB)
  - ASP.NET 3.5 - Security Administration
    - Building an Interface to Select One User Account from Many (C#)
    - Recovering and Changing Passwords (C#)
    - Unlocking and Approving User Accounts (C#)

    - Building an Interface to Select One User Account from Many (VB)
    - Recovering and Changing Passwords (VB)
    - Unlocking and Approving User Accounts (VB)
- Older Versions - ASP.NET 2.0
    - Improvements in Visual Studio 2005
    - ASP.NET 2.0 Page Model
    - Server Controls
    - Master Pages
    - Data Source Controls
    - Data Bound Controls
    - Caching
    - Membership
    - Configuration and Instrumentation
    - Profiles, Themes, and Web Parts

Getting Started with ASP.NET Web Forms

2/20/2020 • 2 minutes to read • Edit Online

The Web Forms model and how to create pages and use controls.

- Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013
  - Getting Started with Web Forms and Visual Studio
  - Create the Project
  - Create the Data Access Layer
  - UI and Navigation
  - Display Data Items and Details
  - Shopping Cart
  - Checkout and Payment with PayPal
  - Membership and Administration
  - URL Routing
  - ASP.NET Error Handling
- Creating a Basic Web Forms Page in Visual Studio 2013
- Code Editing ASP.NET Web Forms in Visual Studio 2013
- Using Page Inspector for Visual Studio 2012 in ASP.NET Web Forms
- Visual Studio 2012 Hands On Labs
  - What's New in ASP.NET and Web Development in Visual Studio 2012
  - What's New in Web Forms in ASP.NET 4.5
  - Using Page Inspector in Visual Studio 2012

Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013

2/20/2020 • 2 minutes to read • Edit Online

This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

- Getting Started with Web Forms and Visual Studio
- Create the Project
- Create the Data Access Layer
- UI and Navigation
- Display Data Items and Details
- Shopping Cart
- Checkout and Payment with PayPal
- Membership and Administration
- URL Routing
- ASP.NET Error Handling

Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2017

2/20/2020 • 5 minutes to read • Edit Online

Download Wingtip Toys Sample Project (C#) or Download E-book (PDF)

This tutorial series shows you how to build an ASP.NET Web Forms application with ASP.NET 4.5 and Microsoft Visual Studio 2017.

Introduction

This tutorial series guides you through creating an ASP.NET Web Forms application using Visual Studio 2017 and ASP.NET 4.5. You'll create an application named **Wingtip Toys** - a simplified storefront web site selling items online. During the series, new ASP.NET 4.5 features are highlighted.

Target audience

Developers new to ASP.NET Web Forms are the target audience for this tutorial series.

You should have some knowledge in the following areas:

- Object-oriented programming (OOP) and languages
- Web development (HTML, CSS, JavaScript)
- Relational databases
- N-tier architecture

To review these areas, consider studying the following content:

- Getting Started with Visual C#
- Web Development, HTML, CSS, JavaScript, SQL, PHP, JQuery
- Relational database
- Multitier architecture

Application features

The ASP.NET Web Form features presented in this series include:

- The Web Application Project (not Web Site Project)
- Web Forms
- Master Pages, Configuration
- Bootstrap
- Entity Framework Code First, LocalDB
- Request Validation
- Strongly-typed Data Controls
- Model Binding
- Data Annotations
- Value Providers
- SSL and OAuth
- ASP.NET Identity, Configuration, and Authorization
- Unobtrusive Validation
- Routing

- ASP.NET Error Handling

Application scenarios and tasks

Tutorial series tasks include:

- Creating, reviewing, and running a new project
- Creating a database structure
- Initializing and seeding a database
- Customizing the UI with styles, graphics, and a master page
- Adding pages and navigation
- Displaying menu details and product data
- Creating a shopping cart
- Adding SSL and OAuth support
- Adding a payment method
- Including an administrator role and a user to the application
- Restricting access to specific pages and folder
- Uploading a file to the web application
- Implementing input validation
- Registering routes for the web application
- Implementing error handling and error logging

Overview

This tutorial series is intended for someone familiar with programming concepts, but new to ASP.NET Web Forms. If you're already familiar with ASP.NET Web Forms, this series can still help you learn about new ASP.NET 4.5 features. For readers unfamiliar with programming concepts and ASP.NET Web Forms, see the additional Web Forms tutorials provided in the Getting Started section on the ASP.NET Web site.

The ASP.NET 4.5 provided in this tutorial series includes the following features:

- A simple UI for creating projects that offers support for many ASP.NET frameworks (Web Forms, MVC, and Web API).

- Bootstrap, a layout, theming, and responsive design framework.

- ASP.NET Identity, a new ASP.NET membership system that works the same in all ASP.NET frameworks and works with web hosting software other than IIS.

- Entity Framework 6

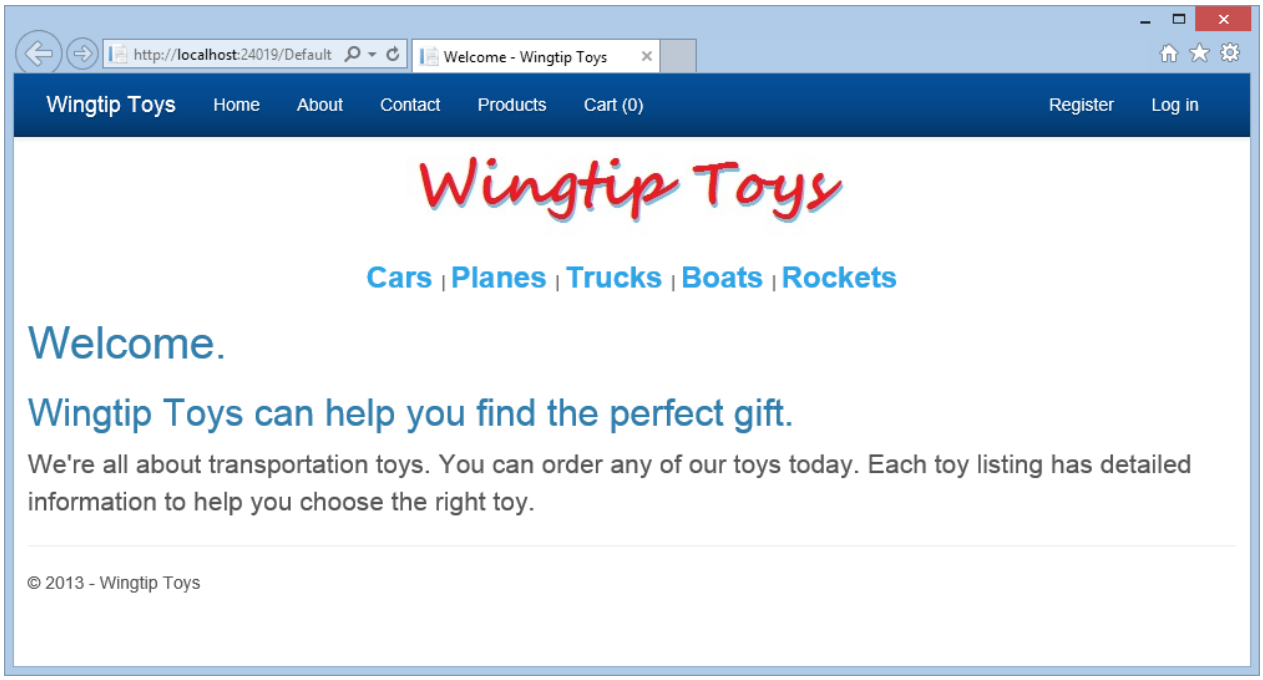
  An update to the Entity Framework enabling you to:

  - Retrieve and manipulate data as strongly-typed objects
  - Access data asynchronously
  - Handle transient connection faults
  - Log SQL statements

For a complete ASP.NET 4.5 feature list, see ASP.NET and Web Tools for Visual Studio 2013 Release Notes.

The Wingtip Toys sample application

The following screenshots are from the ASP.NET Web Forms application that you create in this tutorial series. When you run the application in Visual Studio, the following web Home page appears.

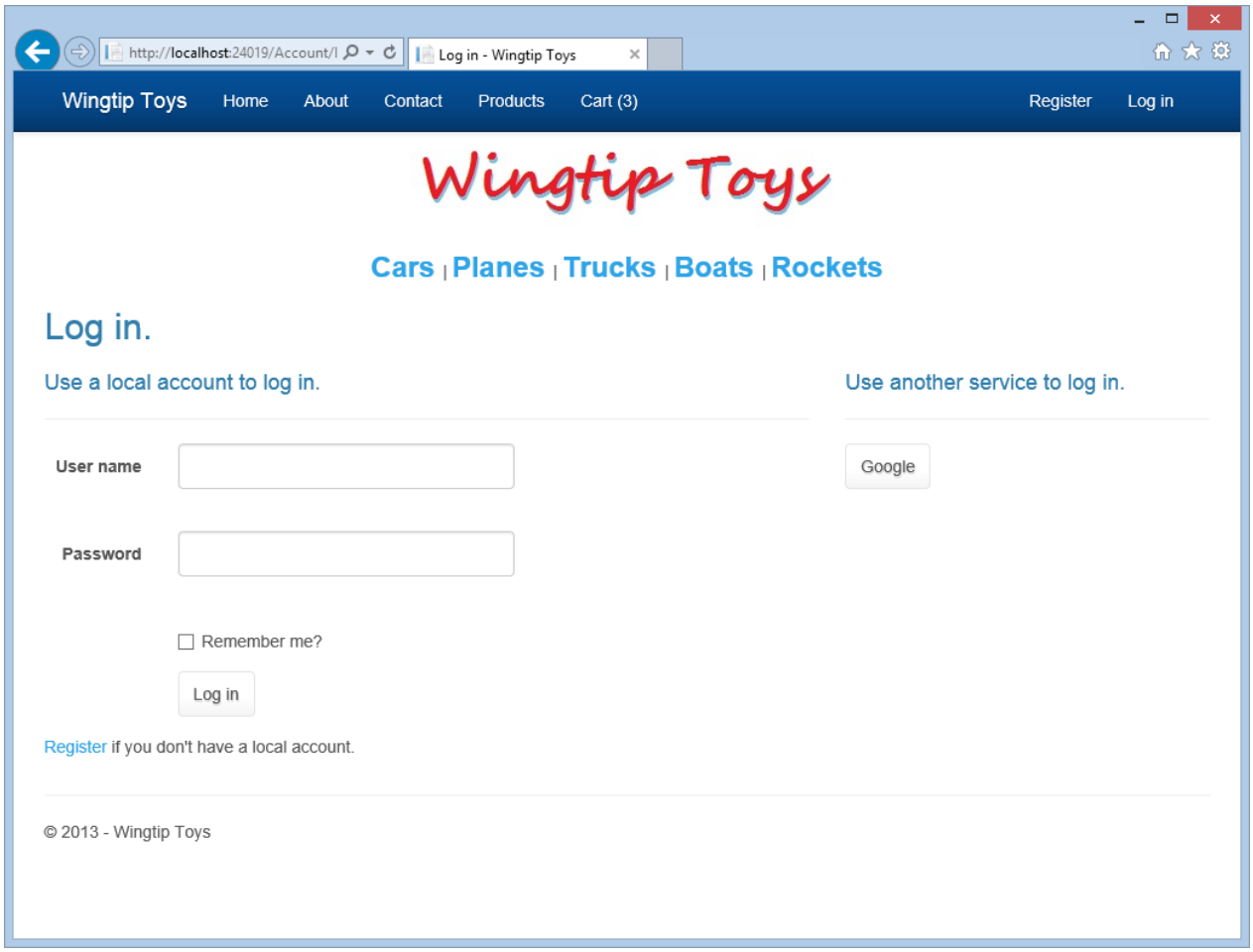


You can register as a new user, or sign in as an existing user. The top navigation has links to product categories and their products from the database.

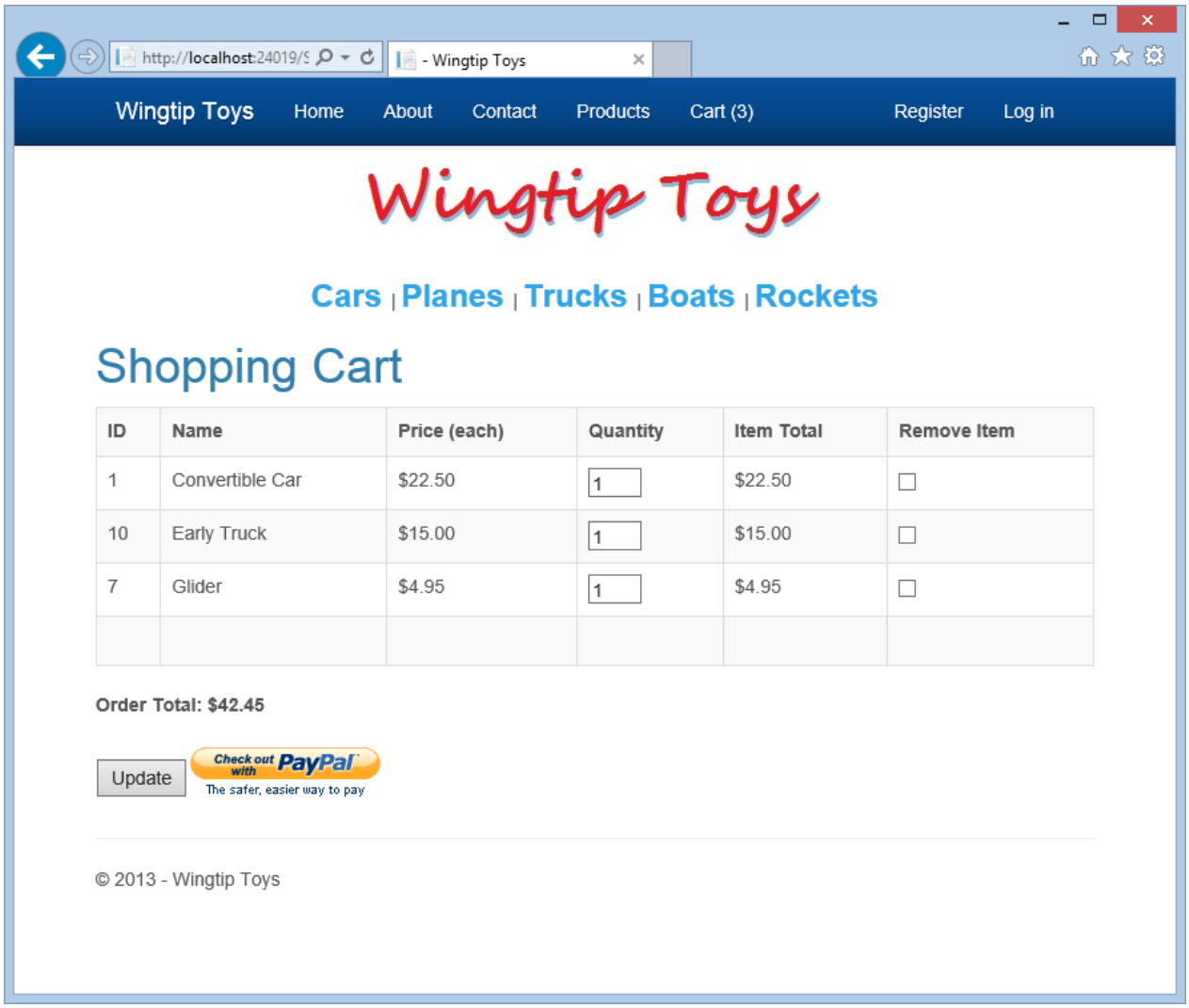
If you select **Products**, all available products are displayed.

If you select a specific product, product details are displayed.

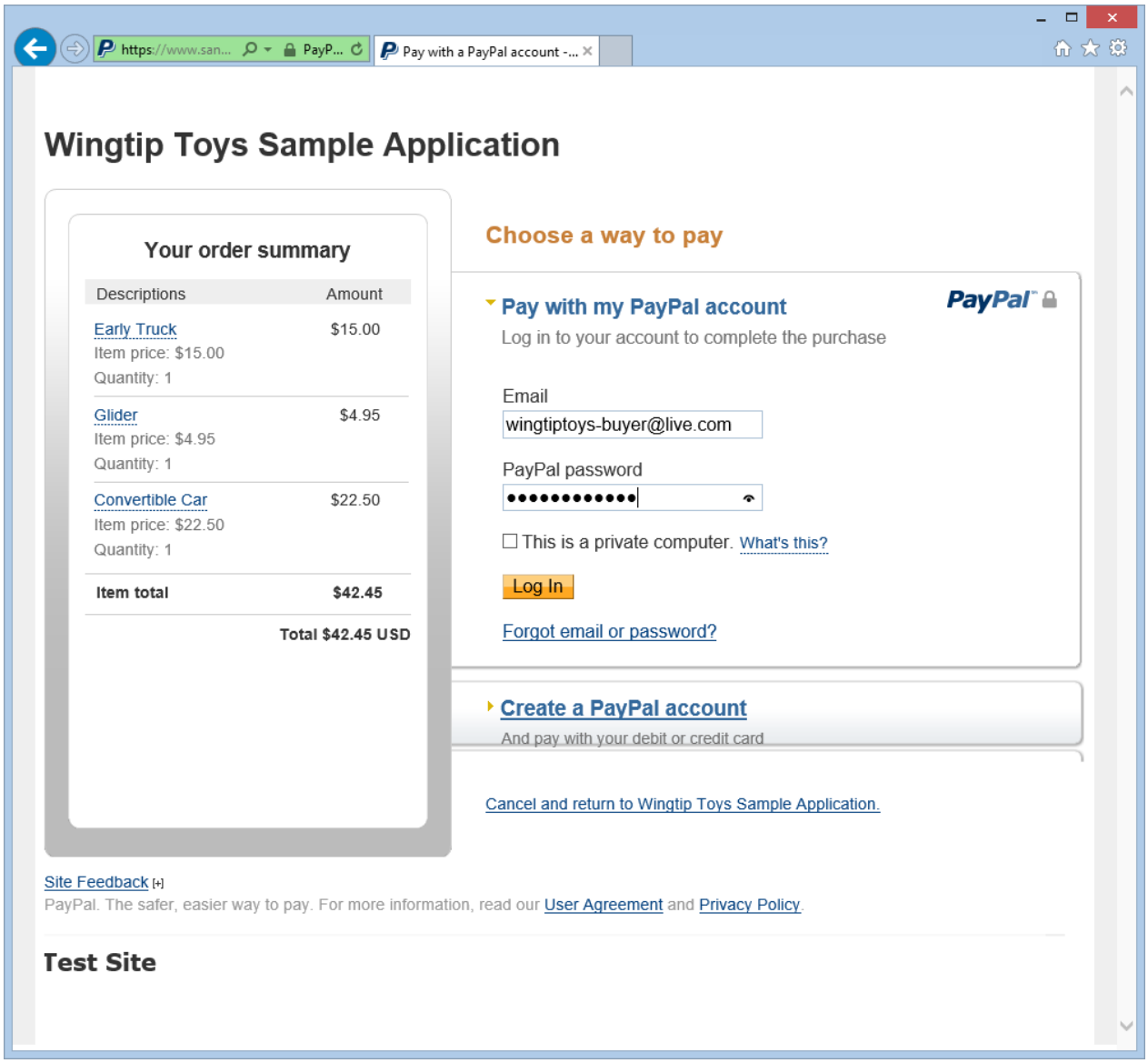
As a user, you can register and sign in with Web Forms template default functionality. This tutorial also explains how to sign in using an existing Gmail account. Additionally, you can sign in as the administrator to add and remove products from the database.



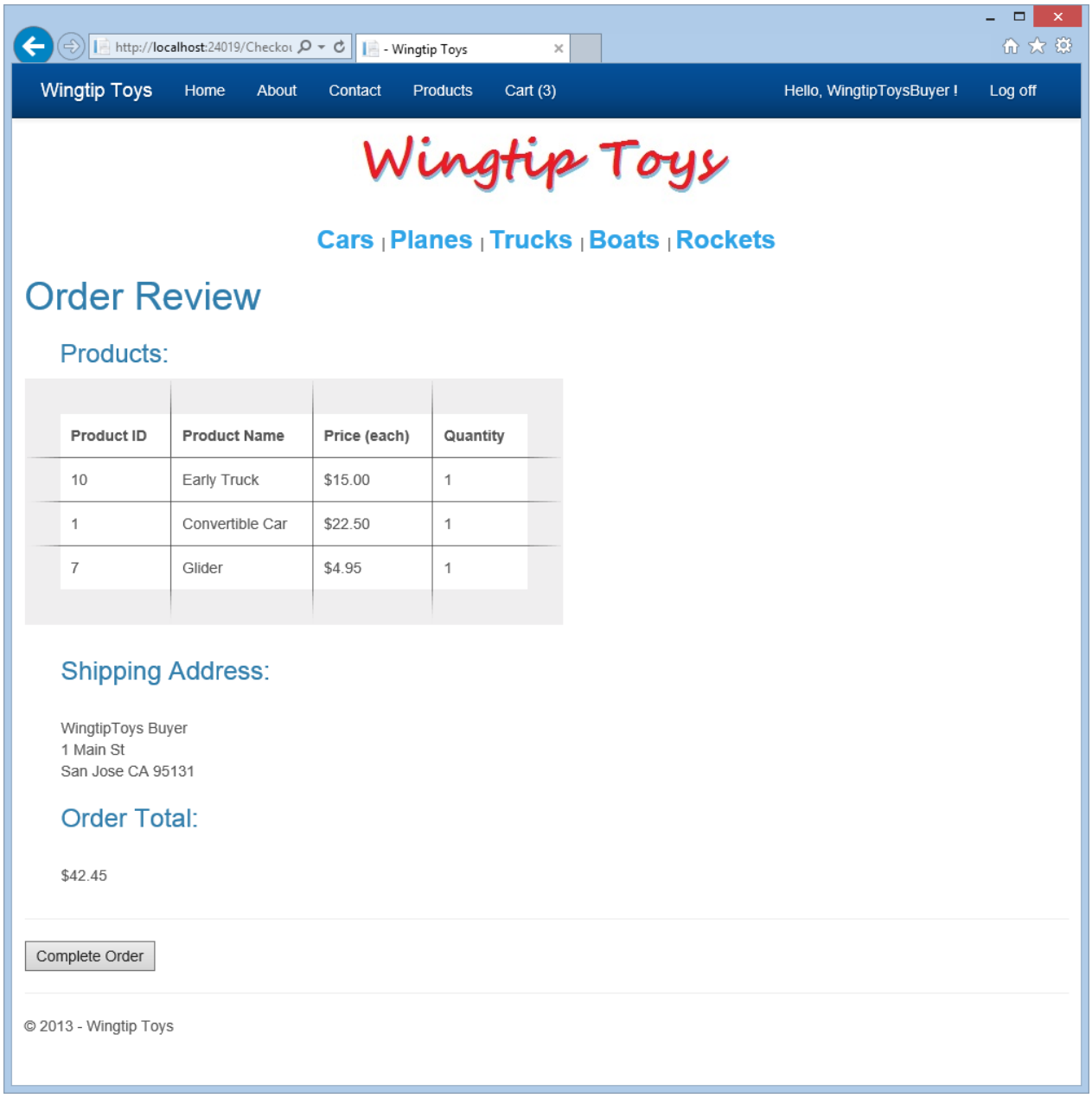
Once you've signed in as a user, you can add products to the shopping cart and checkout with PayPal. The sample application is designed to work in PayPal's developer sandbox. No actual money transaction takes place.



PayPal confirms your account, order, and payment information.



After returning from PayPal, you can review and complete your order.



Prerequisites

Before you start, make sure the following software is installed on your computer:

- Microsoft Visual Studio 2017 or Microsoft Visual Studio Community 2017.

The .NET Framework is installed automatically.

This tutorial series uses Microsoft Visual Studio Community 2017. You can use either that or Microsoft Visual Studio 2017 to complete this tutorial series.

Note the following about Visual Studio:

- Microsoft Visual Studio 2017 and Microsoft Visual Studio Community 2017 are referred to as *Visual Studio* throughout this tutorial series.
- Visual Studio 2017 is installed next to any older versions already installed. Sites created in earlier versions can be opened in Visual Studio 2017 and continue to open in previous versions.
- The first time you started Visual Studio, it is assumed you selected the *Web Development* settings. For more information, see How to: Select Web Development Environment Settings.

After installing the prerequisites, you're ready to begin creating the Web project presented in this tutorial series.

Download the sample application

You can download the completed sample application at anytime from the MSDN Samples site:

Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013 - Wingtip Toys (C#)

This download has the following items:

- The sample application in the *WingtipToys* folder.
- The resources used to create the sample application in the *WingtipToys-Assets* folder in the *WingtipToys* folder.

The download is a *.zip* file. To see the completed project that this tutorial series creates, find and select the *C#* folder in the .zip file. Save the C# folder to the folder you use to work with Visual Studio projects. By default, the Visual Studio 2017 projects folder is:

C:\Users\<*username*>\source\repos

Rename the ***C#*** folder to ***WingtipToys***.

> **NOTE**
> If you already have a folder named *WingtipToys* in your Projects folder, temporarily rename that existing folder before renaming the *C#* folder to *WingtipToys*.

To run the completed project, open the *WingtipToys* folder and double-click the *WingtipToys.sln* file. Visual Studio 2017 opens the project. Next, right-click the *Default.aspx* file in **Solution Explorer** and select **View In Browser**.

Take a ASP.NET Web Forms quiz to review content

After completing the tutorial series, take a quiz to test your knowledge and reinforce key concepts. Each question provides an explanation and links to additional guidance.

- ASP.NET Web Forms Quiz

Tutorial support and comments

For questions and comments, use the Q and A section included on the Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013 - Wingtip Toys (C#) sample page.

Comments on this tutorial series are welcome. When this tutorial series is updated, every effort is made to consider corrections or suggestions for improvements.

If an error occurs, the corresponding error messages could be confusing, with no good explanation on how to fix it. For help, you can check the ASP.NET forums. Another good source is the Q and A section in the Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013 - Wingtip Toys (C#) sample page.

NEXT

Create the Project

2/20/2020 • 11 minutes to read • Edit Online

by Erik Reitan

Download Wingtip Toys Sample Project (C#) or Download E-book (PDF)

> This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

In this tutorial you will create, review, and run the default project in Visual Studio, which will allow you to become familiar with features of ASP.NET. Also, you will review the Visual Studio environment.

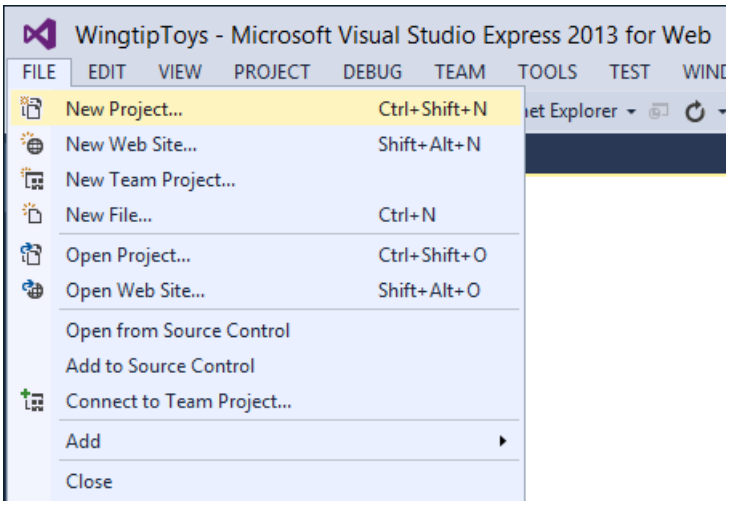
What you'll learn:

- How to create a new Web Forms project.
- The file structure of the Web Forms project.
- How to run the project in Visual Studio.
- The different features of the default Web forms application.
- Some basics about how to use the Visual Studio environment.

Creating the Project

1. Open Visual Studio.

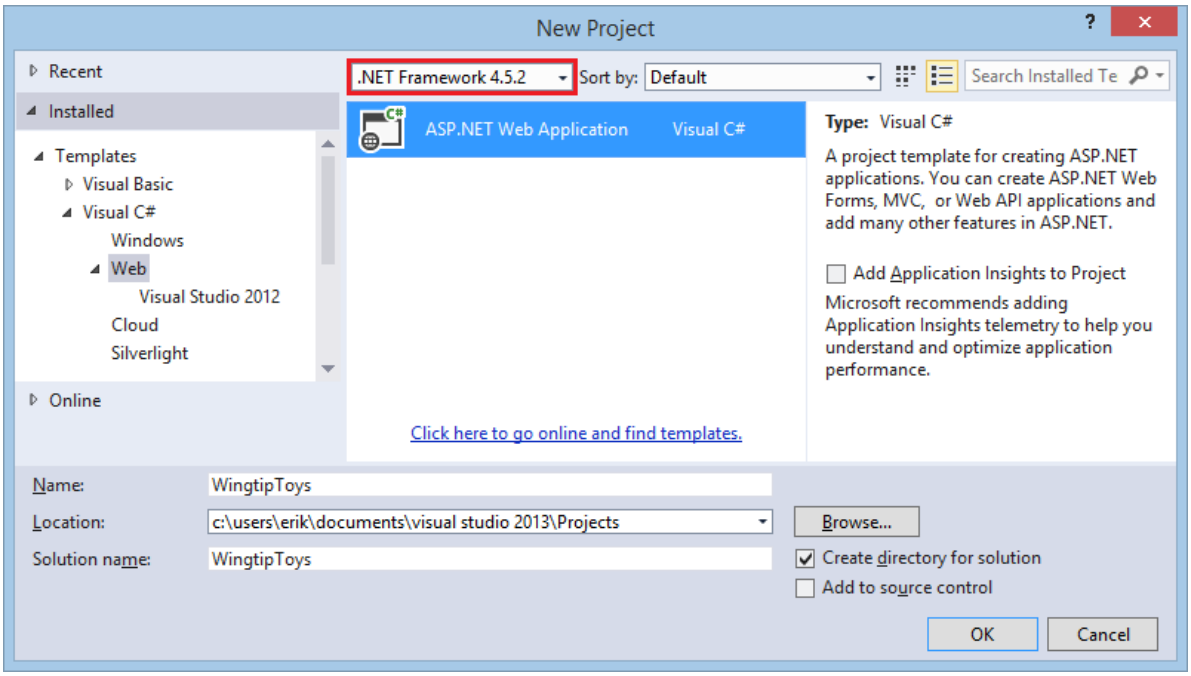
2. Select **New Project** from the **File** menu in Visual Studio.



3. Select the **Templates** -> **Visual C#** -> **Web** templates group on the left.

4. Choose the **ASP.NET Web Application** template in the center column.
   This tutorial series is using .NET Framework 4.5.2.

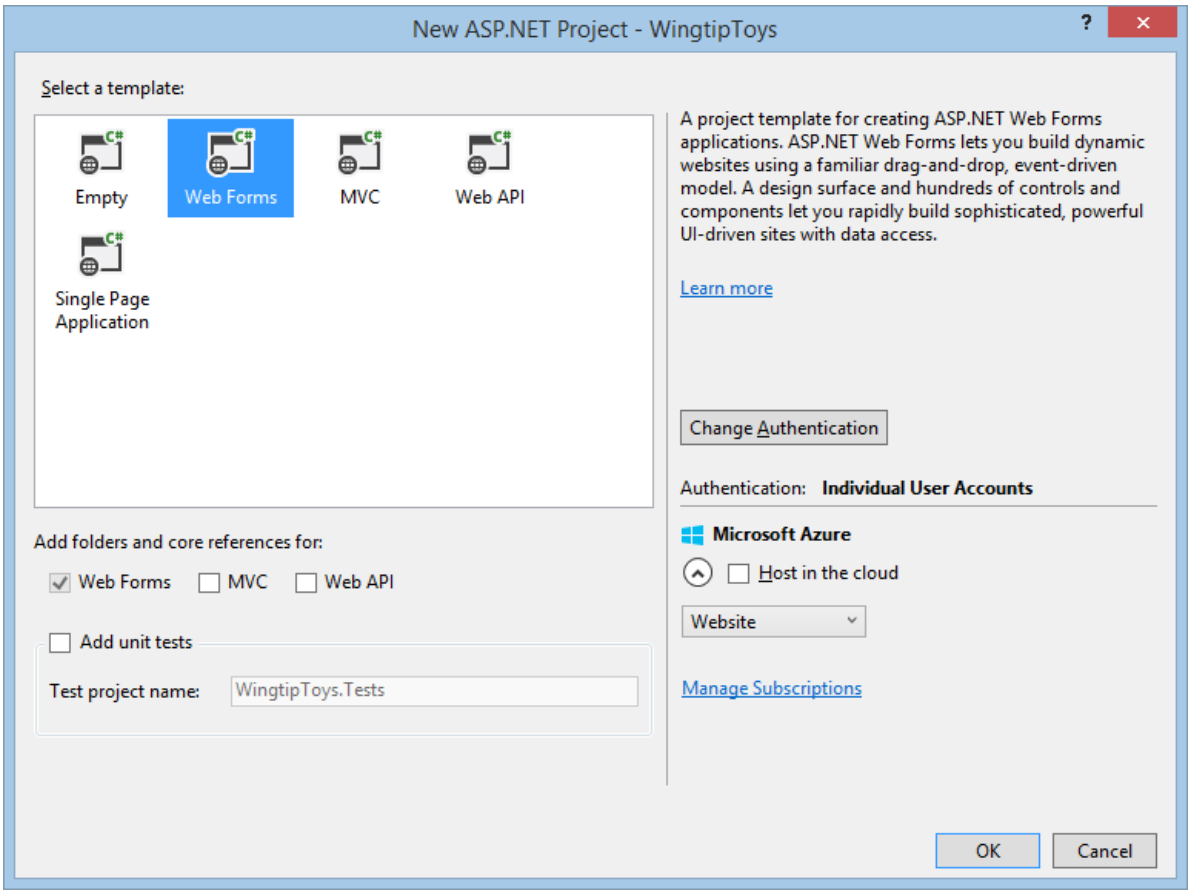
5. Name your project *WingtipToys* and choose the **OK** button.



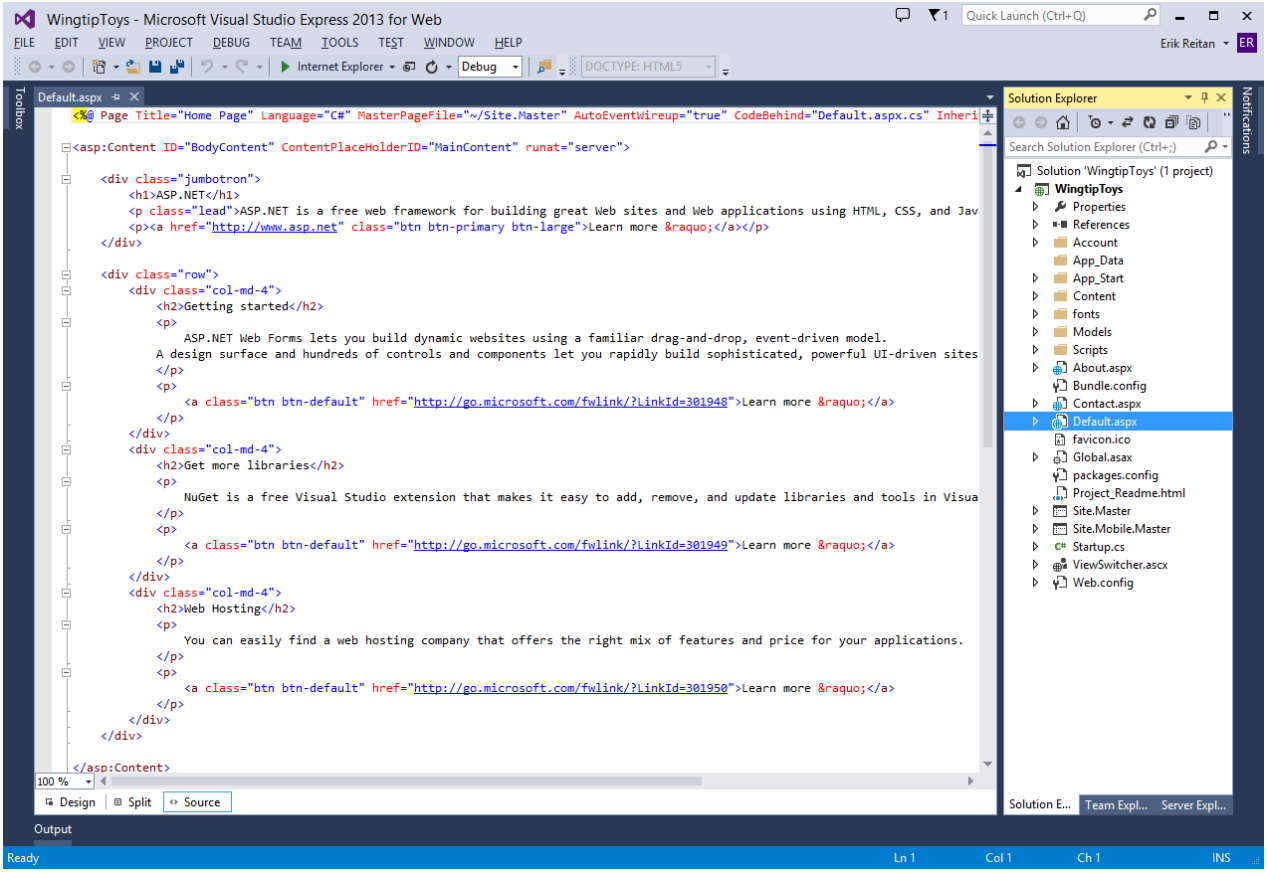
> **NOTE**
>
> The name of the project in this tutorial series is **WingtipToys**. It is recommended that you use this *exact* project name so that the code provided throughout the tutorial series functions as expected.

6. Click the **Change Authentication** button. Select **Individual User Accounts** and click the **OK** button.

7. Select the **Web Forms** template and click the **OK** button.



The project will take a little time to create. When it's ready, open the **Default.aspx** page.



You can switch between **Design** view and **Source** view by selecting an option at the bottom of the center window. **Design** view displays ASP.NET Web pages, master pages, content pages, HTML pages, and user controls using a near-WYSIWYG view. **Source** view displays the HTML markup for your Web page, which you can edit.

TIP

Understanding the ASP.NET Frameworks

ASP.NET Web Forms lets you build dynamic websites using a familiar drag-and-drop, event-driven model. A design surface and hundreds of controls and components let you rapidly build sophisticated, powerful UI-driven sites with data access. The Wingtip Toy Store is based on ASP.NET Web Forms, but many of the concepts you learn in this tutorial series are applicable to all of ASP.NET.

ASP.NET offers four primary development frameworks:

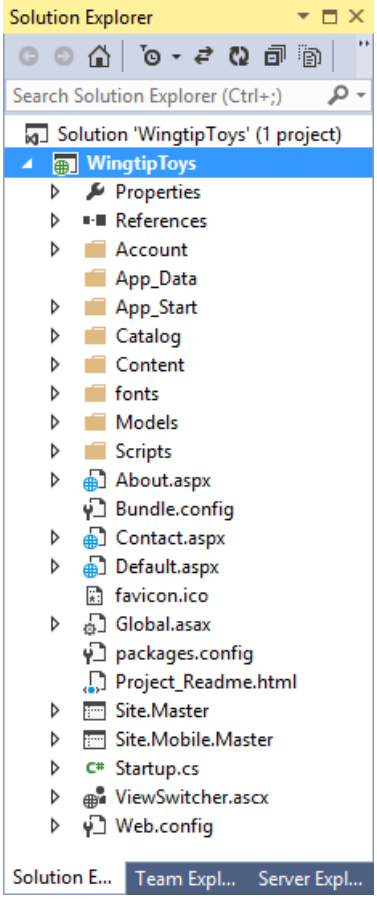
- ASP.NET Web Forms
  The Web Forms framework targets developers who prefer declarative and control-based programming, such as Microsoft Windows Forms (WinForms) and WPF/XAML/Silverlight. It offers a WYSIWYG designer-driven development model, so it's popular with developers looking for a rapid application development (RAD) environment for web development. If you are new to web programming and are familiar with the traditional Microsoft RAD client development tools (for example, for Visual Basic and Visual C#), you can quickly build a web application without having experience in HTML and JavaScript.
- ASP.NET MVC
  ASP.NET MVC targets developers who are interested in patterns and principles like test-driven development, separation of concerns, inversion of control (IoC), and dependency injection (DI). This framework encourages separating the business logic layer of a web application from its presentation layer.
- ASP.NET Web Pages
  ASP.NET Web Pages targets developers who want a simple web development story, along the lines of PHP. In the Web Pages model, you create HTML pages and then add server-based code to the page in order to dynamically control how that markup is rendered. Web Pages is specifically designed to be a lightweight framework, and it's the easiest entry point into ASP.NET for people who know HTML but might not have broad programming experience - for example, students or hobbyists. It's also a good way for web developers who know PHP or similar frameworks to start using ASP.NET.
- ASP.NET Single Page Application
  ASP.NET Single Page Application (SPA) helps you build applications that include significant client-side interactions using HTML 5, CSS 3 and JavaScript. The ASP.NET and Web Tools 2012.2 Update ships a new template for building single page applications using knockout.js and ASP.NET Web API. In addition to the new SPA template, new community-created SPA templates are also available for download.

In addition to the four main development frameworks, ASP.NET also offers additional technologies that are important to be aware of and familiar with, but are not covered in this tutorial series:

- ASP.NET Web API - A framework for building HTTP services that reach a broad range of clients, including browsers and mobile devices.
- ASP.NET SignalR - A library that makes developing real-time web functionality easy.

Reviewing the Project

In Visual Studio, the **Solution Explorer** window lets you manage files for the project. Let's take a look at the folders that have been added to your application in **Solution Explorer**. The web application template adds a basic folder structure:



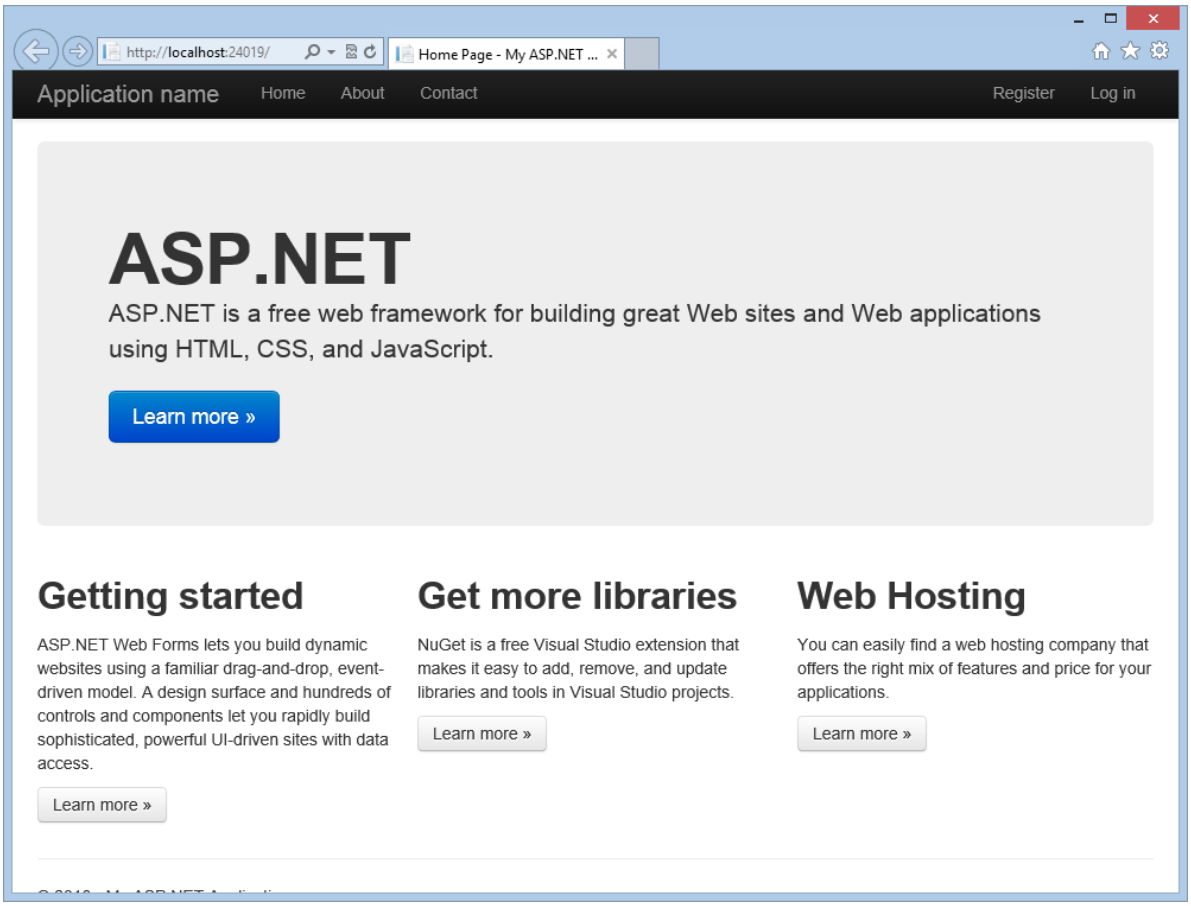
Visual Studio creates some initial folders and files for your project. The first files that you will be working with later in this tutorial are the following:

FILE	PURPOSE
Default.aspx	Typically the first page displayed when the application is run in a browser.
Site.Master	A page that allows you to create a consistent layout and use standard behavior for pages in your application.
Global.asax	An optional file that contains code for responding to application-level and session-level events raised by ASP.NET or by HTTP modules.
Web.config	The configuration data for an application.

Running the Default Web Application

The default Web application provides a rich experience based on built-in functionality and support. Without any changes to the default Web forms project, the application is ready to run on your local Web browser.

1. Press the *F5* key while in Visual Studio.
   The application will build and display in your Web browser.



2. Once you have completed review the running application, close the browser window.

There are three main pages in this default Web application: *Default.aspx* (Home), *About.aspx*, and *Contact.aspx*. Each of these pages can be reached from the top navigation bar. There are also two additional pages contained in the Account folder, the Register.aspx page and Login.aspx page. These two pages allow you to use the membership capabilities of ASP.NET to create, store, and validate user credentials.

ASP.NET Web Forms Background

ASP.NET Web Forms are pages that are based on Microsoft ASP.NET technology, in which code that runs on the server dynamically generates Web page output to the browser or client device. An ASP.NET Web Forms page automatically renders the correct browser-compliant HTML for features such as styles, layout, and so on. Web Forms are compatible with any language supported by the .NET common language runtime, such as Microsoft Visual Basic and Microsoft Visual C#. Also, Web Forms are built on the Microsoft .NET Framework, which provides benefits such as a managed environment, type safety, and inheritance.

When an ASP.NET Web Forms page runs, the page goes through a life cycle in which it performs a series of processing steps. These steps include initialization, instantiating controls, restoring and maintaining state, running event handler code, and rendering. As you become more familiar with the power of ASP.NET Web Forms, it is important for you to understand the ASP.NET page life cycle so that you can write code at the appropriate life-cycle stage for the effect you intend.

When a Web server receives a request for a page, it finds the page, processes it, sends it to the browser, and then discards all page information. If the user requests the same page again, the server repeats the entire sequence, reprocessing the page from scratch. Put another way, a server has no memory of pages that it has processed-pages are stateless. The ASP.NET page framework automatically handles the task of maintaining the state of your page and its controls, and it provides you with explicit ways to maintain the state of application-specific information.

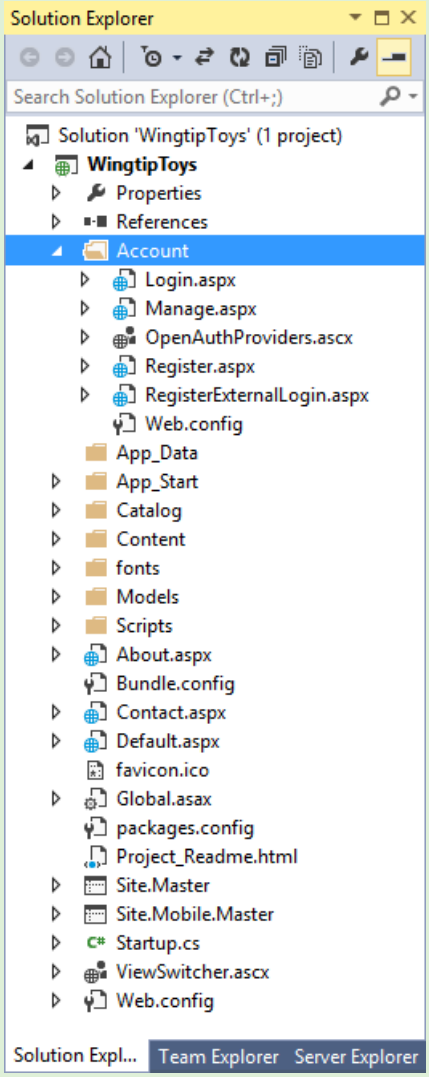
TIP

Web Application Features in the Web Forms Application Template

The ASP.NET Web Forms Application template provides a rich set of built-in functionality. It not only provides you with a *Home.aspx* page, an *About.aspx* page, a *Contact.aspx* page, but also includes membership functionality that registers users and saves their credentials so that they can log in to your website. This overview provides more information about some of the features contained in the ASP.NET Web Forms Application template and how they are used in the Wingtip Toys application.

Membership

ASP.NET Identity stores your users' credentials in a database created by the application. When your users log in, the application validates their credentials by reading the database. Your project's *Account* folder contains the files that implement the various parts of membership: registering, logging in, changing a password, and authorizing access. Additionally, ASP.NET Web Forms supports OAuth and OpenID. These authentication enhancements allow users to log into your site using existing credentials, from such accounts as Facebook, Twitter, Windows Live, and Google.



By default, the template creates a membership database using a default database name on an instance of SQL Server Express LocalDB, the development database server that comes with Visual Studio Express 2013 for Web.

SQL Server Express LocalDB

SQL Server Express LocalDB is a lightweight version of SQL Server that has many programmability features of a SQL Server database. SQL Server Express LocalDB runs in user mode and has a fast, zero-configuration installation that has a short list of installation prerequisites. In Microsoft SQL Server, any database or Transact-SQL code can be moved from SQL Server Express LocalDB to SQL Server and SQL Azure without any upgrade steps. So, SQL Server Express LocalDB can be used as a developer environment for applications targeting all editions of SQL Server. SQL Server Express LocalDB enables features such as stored procedures, user-defined functions and aggregates, .NET Framework integration, spatial types and others that are not available in SQL Server Compact.

Master Pages

An ASP.NET master page defines a consistent appearance and behavior for all of the pages in your application. The layout of the master page merges with the content from an individual content page to produce the final page that the user sees. In the Wingtip Toys application, you modify the *Site.master* master page so that all the pages in the Wingtip Toys website share the same distinctive logo and navigation bar.

HTML5

The ASP.NET Web Forms Application template supports HTML5, which is the latest version of the HTML markup language. HTML5 supports new elements and functionality that make it easier to create Web sites.

Modernizr

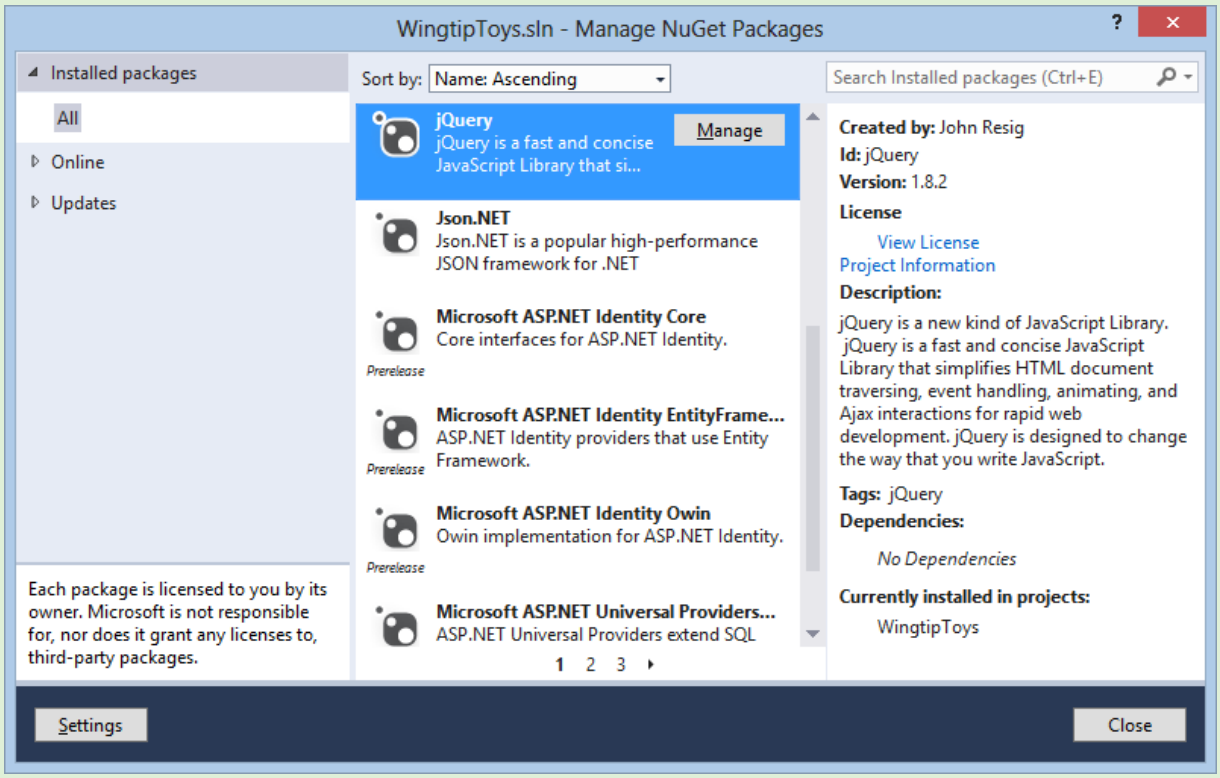
For browsers that do not support HTML5, you can use Modernizr. Modernizr is an open-source JavaScript library that can detect whether a browser supports HTML5 features, and enable them if it does not. In the ASP.NET Web Forms Application template, Modernizr is installed as a NuGet package.

Bootstrap

The Visual Studio 2013 project templates use Bootstrap, a layout and theming framework created by Twitter. Bootstrap uses CSS3 to provide responsive design, which means layouts can dynamically adapt to different browser window sizes. You can also use Bootstrap's theming feature to easily effect a change in the application's look and feel. By default, the ASP.NET Web Application template in Visual Studio 2013 includes Bootstrap as a NuGet package.

NuGet Packages

The ASP.NET Web Forms Application template includes a set of NuGet packages. These packages provide componentized functionality in the form of open source libraries and tools. There is a wide variety of packages to help you create and test your applications. Visual Studio makes it easy to add, remove, and update NuGet packages. Developers can create and add packages to NuGet as well.



When you install a package, NuGet copies files to your solution and automatically makes whatever changes are needed, such as adding references and changing the configuration associated with your Web application. If you decide to remove the library, NuGet removes files and reverses whatever changes it made in your project so that no clutter is left. NuGet is available from the **Tools** menu in Visual Studio.

jQuery

jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. The jQuery JavaScript library is included in the ASP.NET Web Forms Application template as a NuGet package.

Unobtrusive Validation

Built-in validator controls have been configured to use unobtrusive JavaScript for client-side validation logic. This significantly reduces the amount of JavaScript rendered inline in the page markup and reduces the overall page size. Unobtrusive validation is added globally to the ASP.NET Web Forms Application template based on the setting in the <appSettings> element of the *Web.config* file at the root of the application.

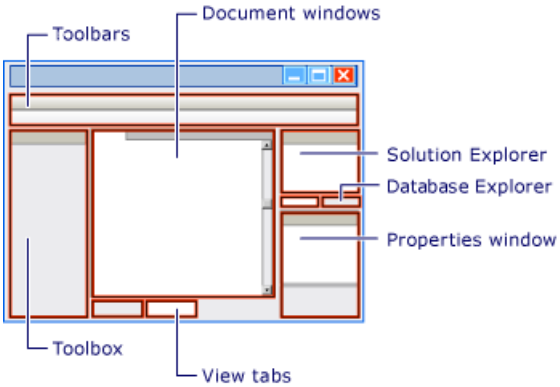
Entity Framework Code First

Besides the features in the ASP.NET Web Forms Application template, the Wingtip Toys application uses Entity Framework Code First, which is a NuGet library that enables code-centric development when you work with data. Put simply, it creates the database portion of your application for you based on the code that you write. Using the Entity Framework, you retrieve and manipulate data as strongly typed objects. This lets you focus on the business logic in your application rather than the details of how data is accessed.

For additional information about the installed libraries and packages included with the ASP.NET Web Forms template, see the list of installed NuGet packages. To do this, In Visual Studio create a new Web Forms project, select **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution**, and select **Installed packages** in the **Manage NuGet Packages** dialog box.

Touring Visual Studio

The primary windows in Visual Studio include the **Solution Explorer**, the **Server Explorer** (**Database Explorer** in Express), the **Properties Window**, the **Toolbox**, the **Toolbar**, and the **Document Window**.



For more information about Visual Studio, see Visual Guide to Visual Web Developer.

Summary

In this tutorial you have created, reviewed and run the default Web Forms application. You have reviewed the different features of the default Web forms application and learned some basics about how to use the Visual Studio environment. In the following tutorials you'll create the data access layer.

Additional Resources

Choosing the Right Programming Model
Web Application Projects versus Web Site Projects
ASP.NET Web Forms Pages Overview

PREVIOUS NEXT

Create the Data Access Layer

2/20/2020 • 13 minutes to read • Edit Online

by Erik Reitan

Download Wingtip Toys Sample Project (C#) or Download E-book (PDF)

> This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

This tutorial describes how to create, access, and review data from a database using ASP.NET Web Forms and Entity Framework Code First. This tutorial builds on the previous tutorial "Create the Project" and is part of the Wingtip Toy Store tutorial series. When you've completed this tutorial, you will have built a group of data-access classes that are in the *Models* folder of the project.

What you'll learn:

- How to create the data models.
- How to initialize and seed the database.
- How to update and configure the application to support the database.

These are the features introduced in the tutorial:

- Entity Framework Code First
- LocalDB
- Data Annotations

Creating the Data Models

Entity Framework is an object-relational mapping (ORM) framework. It lets you work with relational data as objects, eliminating most of the data-access code that you'd usually need to write. Using Entity Framework, you can issue queries using LINQ, then retrieve and manipulate data as strongly typed objects. LINQ provides patterns for querying and updating data. Using Entity Framework allows you to focus on creating the rest of your application, rather than focusing on the data access fundamentals. Later in this tutorial series, we'll show you how to use the data to populate navigation and product queries.

Entity Framework supports a development paradigm called *Code First*. Code First lets you define your data models using classes. A class is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events. You can map classes to an existing database or use them to generate a database. In this tutorial, you'll create the data models by writing data model classes. Then, you'll let Entity Framework create the database on the fly from these new classes.

You will begin by creating the entity classes that define the data models for the Web Forms application. Then you will create a context class that manages the entity classes and provides data access to the database. You will also create an initializer class that you will use to populate the database.

Entity Framework and References

By default, Entity Framework is included when you create a new **ASP.NET Web Application** using the **Web Forms** template. Entity Framework can be installed, uninstalled, and updated as a NuGet package.

This NuGet package includes the following **runtime** assemblies within your project:

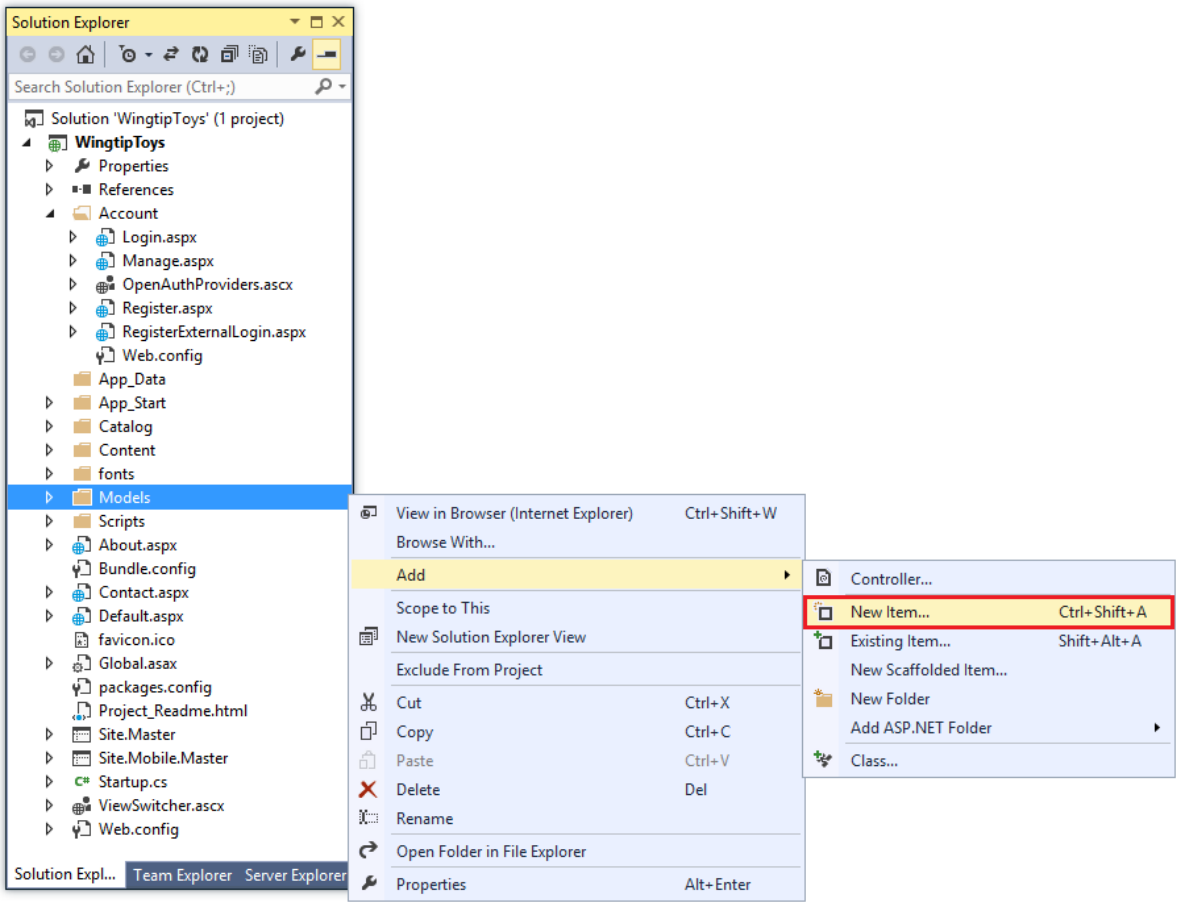
- EntityFramework.dll – All the common runtime code used by Entity Framework
- EntityFramework.SqlServer.dll – The Microsoft SQL Server provider for Entity Framework

Entity Classes

The classes you create to define the schema of the data are called entity classes. If you're new to database design, think of the entity classes as table definitions of a database. Each property in the class specifies a column in the table of the database. These classes provide a lightweight, object-relational interface between object-oriented code and the relational table structure of the database.

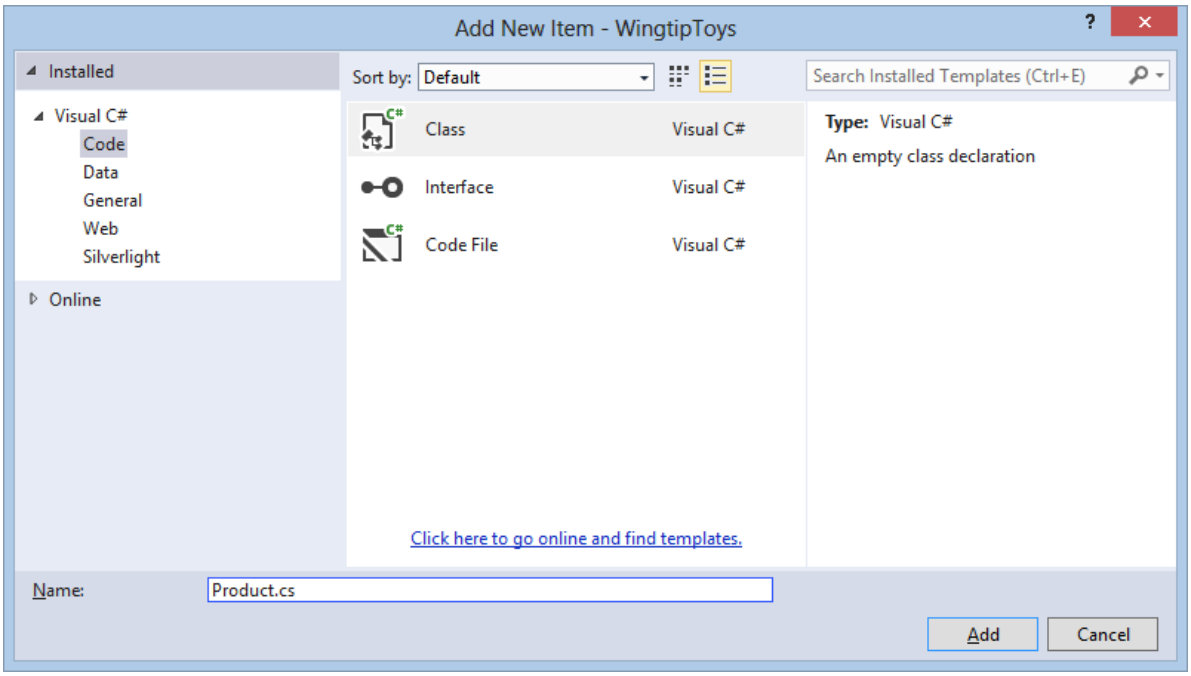
In this tutorial, you'll start out by adding simple entity classes representing the schemas for products and categories. The products class will contain definitions for each product. The name of each of the members of the product class will be `ProductID`, `ProductName`, `Description`, `ImagePath`, `UnitPrice`, `CategoryID`, and `Category`. The category class will contain definitions for each category that a product can belong to, such as Car, Boat, or Plane. The name of each of the members of the category class will be `CategoryID`, `CategoryName`, `Description`, and `Products`. Each product will belong to one of the categories. These entity classes will be added to the project's existing *Models* folder.

1. In **Solution Explorer**, right-click the *Models* folder and then select **Add** -> **New Item**.



The **Add New Item** dialog box is displayed.

2. Under **Visual C#** from the **Installed** pane on the left, select **Code**.



3. Select **Class** from the middle pane and name this new class *Product.cs*.

4. Click **Add**.
   The new class file is displayed in the editor.

5. Replace the default code with the following code:

```
using System.ComponentModel.DataAnnotations;

namespace WingtipToys.Models
{
    public class Product
    {
        [ScaffoldColumn(false)]
        public int ProductID { get; set; }

        [Required, StringLength(100), Display(Name = "Name")]
        public string ProductName { get; set; }

        [Required, StringLength(10000), Display(Name = "Product Description"),
DataType(DataType.MultilineText)]
        public string Description { get; set; }

        public string ImagePath { get; set; }

        [Display(Name = "Price")]
        public double? UnitPrice { get; set; }

        public int? CategoryID { get; set; }

        public virtual Category Category { get; set; }
    }
}
```

6. Create another class by repeating steps 1 through 4, however, name the new class *Category.cs* and replace the default code with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace WingtipToys.Models
{
    public class Category
    {
        [ScaffoldColumn(false)]
        public int CategoryID { get; set; }

        [Required, StringLength(100), Display(Name = "Name")]
        public string CategoryName { get; set; }

        [Display(Name = "Product Description")]
        public string Description { get; set; }

        public virtual ICollection<Product> Products { get; set; }
    }
}
```

As previously mentioned, the `Category` class represents the type of product that the application is designed to sell (such as "Cars", "Boats", "Rockets", and so on), and the `Product` class represents the individual products (toys) in the database. Each instance of a `Product` object will correspond to a row within a relational database table, and each property of the Product class will map to a column in the relational database table. Later in this tutorial, you'll review the product data contained in the database.

Data Annotations

You may have noticed that certain members of the classes have attributes specifying details about the member, such as `[ScaffoldColumn(false)]`. These are *data annotations*. The data annotation attributes can describe how to validate user input for that member, to specify formatting for it, and to specify how it is modeled when the database is created.

Context Class

To start using the classes for data access, you must define a context class. As mentioned previously, the context class manages the entity classes (such as the `Product` class and the `Category` class) and provides data access to the database.

This procedure adds a new C# context class to the *Models* folder.

1. Right-click the *Models* folder and then select **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.

2. Select **Class** from the middle pane, name it *ProductContext.cs* and click **Add**.

3. Replace the default code contained in the class with the following code:

```
using System.Data.Entity;
namespace WingtipToys.Models
{
    public class ProductContext : DbContext
    {
        public ProductContext() : base("WingtipToys")
        {
        }
        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }
    }
}
```

This code adds the `System.Data.Entity` namespace so that you have access to all the core functionality of Entity Framework, which includes the capability to query, insert, update, and delete data by working with strongly typed objects.

The `ProductContext` class represents Entity Framework product database context, which handles fetching, storing, and updating `Product` class instances in the database. The `ProductContext` class derives from the `DbContext` base class provided by Entity Framework.

Initializer Class

You will need to run some custom logic to initialize the database the first time the context is used. This will allow seed data to be added to the database so that you can immediately display products and categories.

This procedure adds a new C# initializer class to the *Models* folder.

1. Create another `Class` in the *Models* folder and name it *ProductDatabaseInitializer.cs*.
2. Replace the default code contained in the class with the following code:

```
using System.Collections.Generic;
using System.Data.Entity;

namespace WingtipToys.Models
{
  public class ProductDatabaseInitializer : DropCreateDatabaseIfModelChanges<ProductContext>
  {
    protected override void Seed(ProductContext context)
    {
      GetCategories().ForEach(c => context.Categories.Add(c));
      GetProducts().ForEach(p => context.Products.Add(p));
    }

    private static List<Category> GetCategories()
    {
      var categories = new List<Category> {
                new Category
                {
                    CategoryID = 1,
                    CategoryName = "Cars"
                },
                new Category
                {
                    CategoryID = 2,
                    CategoryName = "Planes"
                },
                new Category
                {
                    CategoryID = 3,
                    CategoryName = "Trucks"
                },
                new Category
                {
                    CategoryID = 4,
                    CategoryName = "Boats"
                },
                new Category
                {
                    CategoryID = 5,
                    CategoryName = "Rockets"
                },
            };

      return categories;
    }

    private static List<Product> GetProducts()
```

```
    {
      var products = new List<Product> {
                new Product
                {
                    ProductID = 1,
                    ProductName = "Convertible Car",
                    Description = "This convertible car is fast! The engine is powered by a neutrino
based battery (not included)." +
                                  "Power it up and let it go!",
                    ImagePath="carconvert.png",
                    UnitPrice = 22.50,
                    CategoryID = 1
               },
                new Product
                {
                    ProductID = 2,
                    ProductName = "Old-time Car",
                    Description = "There's nothing old about this toy car, except it's looks.
Compatible with other old toy cars.",
                    ImagePath="carearly.png",
                    UnitPrice = 15.95,
                     CategoryID = 1
               },
                new Product
                {
                    ProductID = 3,
                    ProductName = "Fast Car",
                    Description = "Yes this car is fast, but it also floats in water.",
                    ImagePath="carfast.png",
                    UnitPrice = 32.99,
                    CategoryID = 1
                },
                new Product
                {
                    ProductID = 4,
                    ProductName = "Super Fast Car",
                    Description = "Use this super fast car to entertain guests. Lights and doors
work!",
                    ImagePath="carfaster.png",
                    UnitPrice = 8.95,
                    CategoryID = 1
                },
                new Product
                {
                    ProductID = 5,
                    ProductName = "Old Style Racer",
                    Description = "This old style racer can fly (with user assistance). Gravity
controls flight duration." +
                                  "No batteries required.",
                    ImagePath="carracer.png",
                    UnitPrice = 34.95,
                    CategoryID = 1
                },
                new Product
                {
                    ProductID = 6,
                    ProductName = "Ace Plane",
                    Description = "Authentic airplane toy. Features realistic color and details.",
                    ImagePath="planeace.png",
                    UnitPrice = 95.00,
                    CategoryID = 2
                },
                new Product
                {
                    ProductID = 7,
                    ProductName = "Glider",
                    Description = "This fun glider is made from real balsa wood. Some assembly
required.",
                    ImagePath="planeglider.png",
```

```
                    UnitPrice = 4.95,
                    CategoryID = 2
                },
                new Product
                {
                    ProductID = 8,
                    ProductName = "Paper Plane",
                    Description = "This paper plane is like no other paper plane. Some folding
required.",
                    ImagePath="planepaper.png",
                    UnitPrice = 2.95,
                    CategoryID = 2
                },
                new Product
                {
                    ProductID = 9,
                    ProductName = "Propeller Plane",
                    Description = "Rubber band powered plane features two wheels.",
                    ImagePath="planeprop.png",
                    UnitPrice = 32.95,
                    CategoryID = 2
                },
                new Product
                {
                    ProductID = 10,
                    ProductName = "Early Truck",
                    Description = "This toy truck has a real gas powered engine. Requires regular tune
ups.",
                    ImagePath="truckearly.png",
                    UnitPrice = 15.00,
                    CategoryID = 3
                },
                new Product
                {
                    ProductID = 11,
                    ProductName = "Fire Truck",
                    Description = "You will have endless fun with this one quarter sized fire truck.",
                    ImagePath="truckfire.png",
                    UnitPrice = 26.00,
                    CategoryID = 3
                },
                new Product
                {
                    ProductID = 12,
                    ProductName = "Big Truck",
                    Description = "This fun toy truck can be used to tow other trucks that are not as
big.",
                    ImagePath="truckbig.png",
                    UnitPrice = 29.00,
                    CategoryID = 3
                },
                new Product
                {
                    ProductID = 13,
                    ProductName = "Big Ship",
                    Description = "Is it a boat or a ship. Let this floating vehicle decide by using
its " +
                                  "artifically intelligent computer brain!",
                    ImagePath="boatbig.png",
                    UnitPrice = 95.00,
                    CategoryID = 4
                },
                new Product
                {
                    ProductID = 14,
                    ProductName = "Paper Boat",
                    Description = "Floating fun for all! This toy boat can be assembled in seconds.
Floats for minutes!" +
                                  "Some folding required.",
```

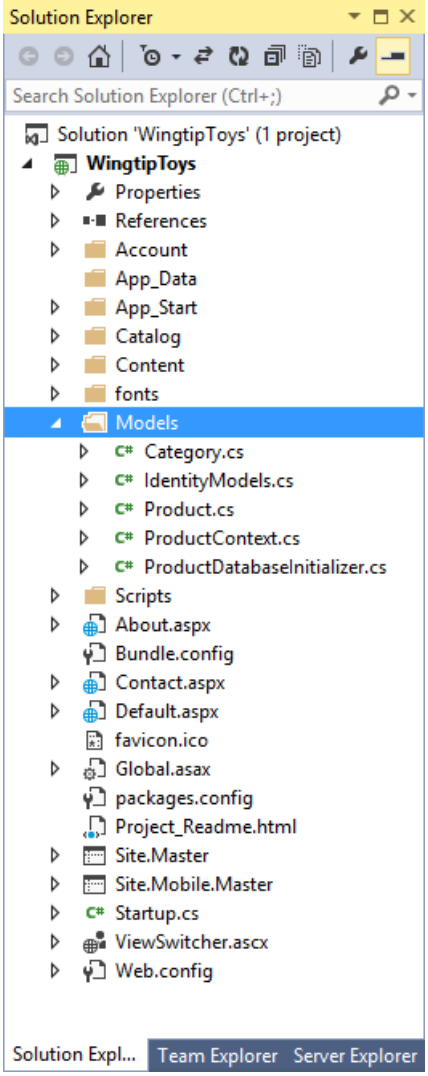
```
                    ImagePath="boatpaper.png",
                    UnitPrice = 4.95,
                    CategoryID = 4
                },
                new Product
                {
                    ProductID = 15,
                    ProductName = "Sail Boat",
                    Description = "Put this fun toy sail boat in the water and let it go!",
                    ImagePath="boatsail.png",
                    UnitPrice = 42.95,
                    CategoryID = 4
                },
                new Product
                {
                    ProductID = 16,
                    ProductName = "Rocket",
                    Description = "This fun rocket will travel up to a height of 200 feet.",
                    ImagePath="rocket.png",
                    UnitPrice = 122.95,
                    CategoryID = 5
                }
            };

        return products;
      }
    }
  }
```

As you can see from the above code, when the database is created and initialized, the `Seed` property is overridden and set. When the `Seed` property is set, the values from the categories and products are used to populate the database. If you attempt to update the seed data by modifying the above code after the database has been created, you won't see any updates when you run the Web application. The reason is the above code uses an implementation of the `DropCreateDatabaseIfModelChanges` class to recognize if the model (schema) has changed before resetting the seed data. If no changes are made to the `Category` and `Product` entity classes, the database will not be reinitialized with the seed data.

> **NOTE**
> If you wanted the database to be recreated every time you ran the application, you could use the `DropCreateDatabaseAlways` class instead of the `DropCreateDatabaseIfModelChanges` class. However for this tutorial series, use the `DropCreateDatabaseIfModelChanges` class.

At this point in this tutorial, you will have a *Models* folder with four new classes and one default class:



Configuring the Application to Use the Data Model

Now that you've created the classes that represent the data, you must configure the application to use the classes. In the *Global.asax* file, you add code that initializes the model. In the *Web.config* file you add information that tells the application what database you'll use to store the data that's represented by the new data classes. The *Global.asax* file can be used to handle application events or methods. The *Web.config* file allows you to control the configuration of your ASP.NET web application.

Updating the Global.asax file

To initialize the data models when the application starts, you will update the `Application_Start` handler in the *Global.asax.cs* file.

> **NOTE**
>
> In Solution Explorer, you can select either the *Global.asax* file or the *Global.asax.cs* file to edit the *Global.asax.cs* file.

1. Add the following code highlighted in yellow to the `Application_Start` method in the *Global.asax.cs* file.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Optimization;
using System.Web.Routing;
using System.Web.Security;
using System.Web.SessionState;
using System.Data.Entity;
using WingtipToys.Models;

namespace WingtipToys
{
    public class Global : HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
            // Code that runs on application startup
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);

            // Initialize the product database.
            Database.SetInitializer(new ProductDatabaseInitializer());
        }
    }
}
```

NOTE

Your browser must support HTML5 to view the code highlighted in yellow when viewing this tutorial series in a browser.

As shown in the above code, when the application starts, the application specifies the initializer that will run during the first time the data is accessed. The two additional namespaces are required to access the `Database` object and the `ProductDatabaseInitializer` object.

Modifying the Web.Config File

Although Entity Framework Code First will generate a database for you in a default location when the database is populated with seed data, adding your own connection information to your application gives you control of the database location. You specify this database connection using a connection string in the application's *Web.config* file at the root of the project. By adding a new connection string, you can direct the location of the database (*wingtiptoys.mdf*) to be built in the application's data directory (*App_Data*), rather than its default location. Making this change will allow you to find and inspect the database file later in this tutorial.

1. In **Solution Explorer**, find and open the *Web.config* file.
2. Add the connection string highlighted in yellow to the `<connectionStrings>` section of the *Web.config* file as follows:

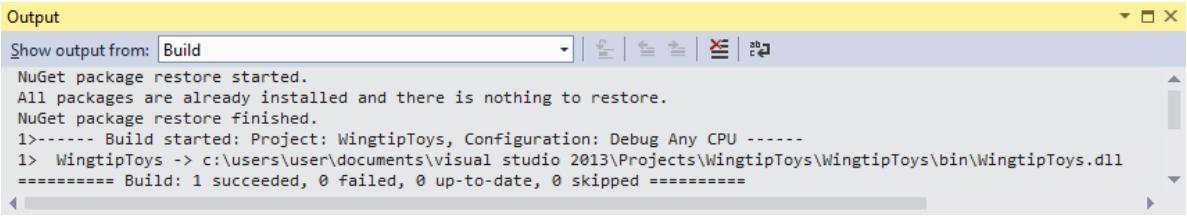
```
<connectionStrings>
<add name="DefaultConnection" connectionString="Data Source=
(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\aspnet-WingtipToys-20131119102907.mdf;Initial
Catalog=aspnet-WingtipToys-20131119102907;Integrated Security=True"
providerName="System.Data.SqlClient" />
<add name="WingtipToys"
connectionString="Data Source=
(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\wingtiptoys.mdf;Integrated Security=True"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

When the application is run for the first time, it will build the database at the location specified by the connection string. But before running the application, let's build it first.

Building the Application

To make sure that all the classes and changes to your Web application work, you should build the application.

1. From the **Debug** menu, select **Build WingtipToys**.
   The **Output** window is displayed, and if all went well, you see a *succeeded* message.



If you run into an error, re-check the above steps. The information in the **Output** window will indicate which file has a problem and where in the file a change is required. This information will enable you to determine what part of the above steps need to be reviewed and fixed in your project.

Summary

In this tutorial of the series you have created the data model, as well as, added the code that will be used to initialize and seed the database. You have also configured the application to use the data models when the application is run.

In the next tutorial, you'll update the UI, add navigation, and retrieve data from the database. This will result in the database being automatically created based on the entity classes that you created in this tutorial.

Additional Resources

Entity Framework Overview
Beginner's Guide to the ADO.NET Entity Framework
Code First Development with Entity Framework (video)
Code First Relationships Fluent API
Code First Data Annotations
Productivity Improvements for the Entity Framework

PREVIOUS NEXT

UI and Navigation

2/20/2020 • 19 minutes to read • Edit Online

by Erik Reitan

Download Wingtip Toys Sample Project (C#) or Download E-book (PDF)

> This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

In this tutorial, you will modify the UI of the default Web application to support features of the Wingtip Toys store front application. Also, you will add simple and data bound navigation. This tutorial builds on the previous tutorial "Create the Data Access Layer" and is part of the Wingtip Toys tutorial series.

What you'll learn:

- How to change the UI to support features of the Wingtip Toys store front application.
- How to configure an HTML5 element to include page navigation.
- How to create a data-driven control to navigate to specific product data.
- How to display data from a database created using Entity Framework Code First.

ASP.NET Web Forms allow you to create dynamic content for your Web application. Each ASP.NET Web page is created in a manner similar to a static HTML Web page (a page that does not include server-based processing), but ASP.NET Web page includes extra elements that ASP.NET recognizes and processes to generate HTML when the page runs.

With a static HTML page (*.htm* or *.html* file), the server fulfills a `Web` request by reading the file and sending it as-is to the browser. In contrast, when someone requests an ASP.NET Web page (*.aspx* file), the page runs as a program on the Web server. While the page is running, it can perform any task that your Web site requires, including calculating values, reading or writing database information, or calling other programs. As its output, the page dynamically produces markup (such as elements in HTML) and sends this dynamic output to the browser.

Modifying the UI

You'll continue this tutorial series by modifying the *Default.aspx* page. You will modify the UI that's already established by the default template used to create the application. The type of modifications you'll do are typical when creating any Web Forms application. You'll do this by changing the title, replacing some content, and removing unneeded default content.

1. Open or switch to the *Default.aspx* page.

2. If the page appears in **Design** view, switch to **Source** view.

3. At the top of the page in the `@Page` directive, change the `Title` attribute to "Welcome", as shown highlighted in yellow below.

```
<%@ Page Title="Welcome" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="Default.aspx.cs" Inherits="WingtipToys._Default" %>
```

4. Also on the *Default.aspx* page, replace all of the default content contained in the `<asp:Content>` tag so that

the markup appears as below.

```
<asp:Content ID="BodyContent" ContentPlaceHolderID="MainContent" runat="server">
        <h1><%: Title %>.</h1>
        <h2>Wingtip Toys can help you find the perfect gift.</h2>
        <p class="lead">We're all about transportation toys. You can order
                any of our toys today. Each toy listing has detailed
                information to help you choose the right toy.</p>
</asp:Content>
```

5. Save the *Default.aspx* page by selecting **Save Default.aspx** from the **File** menu.

   The resulting *Default.aspx* page will appear as follows:

```
<%@ Page Title="Welcome" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="Default.aspx.cs" Inherits="WingtipToys._Default" %>

<asp:Content ID="BodyContent" ContentPlaceHolderID="MainContent" runat="server">
        <h1><%: Title %>.</h1>
        <h2>Wingtip Toys can help you find the perfect gift.</h2>
        <p class="lead">We're all about transportation toys. You can order
                any of our toys today. Each toy listing has detailed
                information to help you choose the right toy.</p>
</asp:Content>
```

In the example, you have set the `Title` attribute of the `@Page` directive. When the HTML is displayed in a browser, the server code `<%: Page.Title %>` resolves to the content contained in the `Title` attribute.

The example page includes the basic elements that constitute an ASP.NET Web page. The page contains static text as you might have in an HTML page, along with elements that are specific to ASP.NET. The content contained in the *Default.aspx* page will be integrated with the master page content, which will be explained later in this tutorial.

@Page Directive

ASP.NET Web Forms usually contain directives that allow you to specify page properties and configuration information for the page. The directives are used by ASP.NET as instructions for how to process the page, but they are not rendered as part of the markup that is sent to the browser.

The most commonly used directive is the `@Page` directive, which allows you to specify many configuration options for the page, including the following:

1. The server programming language for code in the page, such as C#.
2. Whether the page is a page with server code directly in the page, which is called a single-file page, or whether it is a page with code in a separate class file, which is called a code-behind page.
3. Whether the page has an associated master page and should therefore be treated as a content page.
4. Debugging and tracing options.

If you do not include an `@Page` directive in the page, or if the directive does not include a specific setting, a setting will be inherited from the *Web.config* configuration file or from the *Machine.config* configuration file. The *Machine.config* file provides additional configuration settings to all applications running on a machine.

> **NOTE**
>
> The *Machine.config* also provides details about all possible configuration settings.

Web Server Controls

In most ASP.NET Web Forms applications, you will add controls that allow the user to interact with the page, such

as buttons, text boxes, lists, and so on. These Web server controls are similar to HTML buttons and input elements. However, they are processed on the server, allowing you to use server code to set their properties. These controls also raise events that you can handle in server code.

Server controls use a special syntax that ASP.NET recognizes when the page runs. The tag name for ASP.NET server controls starts with an `asp:` prefix. This allows ASP.NET to recognize and process these server controls. The prefix might be different if the control is not part of the .NET Framework. In addition to the `asp:` prefix, ASP.NET server controls also include the `runat="server"` attribute and an `ID` that you can use to reference the control in server code.

When the page runs, ASP.NET identifies the server controls and runs the code that is associated with those controls. Many controls render some HTML or other markup into the page when it is displayed in a browser.

Server Code

Most ASP.NET Web Forms applications include code that runs on the server when the page is processed. As mentioned above, server code can be used to do a variety of things, such as adding data to a ListView control. ASP.NET supports many languages to run on the server, including C#, Visual Basic, J#, and others.

ASP.NET supports two models for writing server code for a Web page. In the single-file model, the code for the page is in a script element where the opening tag includes the `runat="server"` attribute. Alternatively, you can create the code for the page in a separate class file, which is referred to as the code-behind model. In this case, the ASP.NET Web Forms page generally contains no server code. Instead, the `@Page` directive includes information that links the *.aspx* page with its associated code-behind file.

The `CodeBehind` attribute contained in the `@Page` directive specifies the name of the separate class file, and the `Inherits` attribute specifies the name of the class within the code-behind file that corresponds to the page.

Updating the Master Page

In ASP.NET Web Forms, master pages allow you to create a consistent layout for the pages in your application. A single master page defines the look and feel and standard behavior that you want for all of the pages (or a group of pages) in your application. You can then create individual content pages that contain the content you want to display, as explained above. When users request the content pages, ASP.NET merges them with the master page to produce output that combines the layout of the master page with the content from the content page.

The new site needs a single logo to display on every page. To add this logo, you can modify the HTML on the master page.

1. In **Solution Explorer**, find and open the **Site.Master** page.
2. If the page is in **Design** view, switch to **Source** view.
3. Update the master page by **modifying or adding** the markup highlighted in yellow:

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site.master.cs"
Inherits="WingtipToys.SiteMaster" %>

<!DOCTYPE html>

<html lang="en">
<head runat="server">
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title><%: Page.Title %> - Wingtip Toys</title>

    <asp:PlaceHolder runat="server">
        <%: Scripts.Render("~/bundles/modernizr") %>
    </asp:PlaceHolder>
    <webopt:bundlereference runat="server" path="~/Content/css" />
    <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />
```

```
</head>
<body>
    <form runat="server">
        <asp:ScriptManager runat="server">
            <Scripts>
                <%--To learn more about bundling scripts in ScriptManager see
https://go.microsoft.com/fwlink/?LinkID=301884 --%>
                <%--Framework Scripts--%>
                <asp:ScriptReference Name="MsAjaxBundle" />
                <asp:ScriptReference Name="jquery" />
                <asp:ScriptReference Name="bootstrap" />
                <asp:ScriptReference Name="respond" />
                <asp:ScriptReference Name="WebForms.js" Assembly="System.Web"
Path="~/Scripts/WebForms/WebForms.js" />
                <asp:ScriptReference Name="WebUIValidation.js" Assembly="System.Web"
Path="~/Scripts/WebForms/WebUIValidation.js" />
                <asp:ScriptReference Name="MenuStandards.js" Assembly="System.Web"
Path="~/Scripts/WebForms/MenuStandards.js" />
                <asp:ScriptReference Name="GridView.js" Assembly="System.Web"
Path="~/Scripts/WebForms/GridView.js" />
                <asp:ScriptReference Name="DetailsView.js" Assembly="System.Web"
Path="~/Scripts/WebForms/DetailsView.js" />
                <asp:ScriptReference Name="TreeView.js" Assembly="System.Web"
Path="~/Scripts/WebForms/TreeView.js" />
                <asp:ScriptReference Name="WebParts.js" Assembly="System.Web"
Path="~/Scripts/WebForms/WebParts.js" />
                <asp:ScriptReference Name="Focus.js" Assembly="System.Web"
Path="~/Scripts/WebForms/Focus.js" />
                <asp:ScriptReference Name="WebFormsBundle" />
                <%--Site Scripts--%>
            </Scripts>
        </asp:ScriptManager>

        <div class="navbar navbar-inverse navbar-fixed-top">
            <div class="container">
                <div class="navbar-header">
                    <button type="button" class="navbar-toggle" data-toggle="collapse" data-
target=".navbar-collapse">
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                    </button>
                    <a class="navbar-brand" runat="server" href="~/">Wingtip Toys</a>
                </div>
                <div class="navbar-collapse collapse">
                    <ul class="nav navbar-nav">
                        <li><a runat="server" href="~/">Home</a></li>
                        <li><a runat="server" href="~/About">About</a></li>
                        <li><a runat="server" href="~/Contact">Contact</a></li>
                    </ul>
                    <asp:LoginView runat="server" ViewStateMode="Disabled">
                        <AnonymousTemplate>
                            <ul class="nav navbar-nav navbar-right">
                                <li><a runat="server" href="~/Account/Register">Register</a></li>
                                <li><a runat="server" href="~/Account/Login">Log in</a></li>
                            </ul>
                        </AnonymousTemplate>
                        <LoggedInTemplate>
                            <ul class="nav navbar-nav navbar-right">
                                <li><a runat="server" href="~/Account/Manage" title="Manage your
account">Hello, <%: Context.User.Identity.GetUserName()  %> !</a></li>
                                <li>
                                    <asp:LoginStatus runat="server" LogoutAction="Redirect"
LogoutText="Log off" LogoutPageUrl="~/" OnLoggingOut="Unnamed_LoggingOut" />
                                </li>
                            </ul>
                        </LoggedInTemplate>
                    </asp:LoginView>
                </div>
```

```
            </div>
        </div>
        <div id="TitleContent" style="text-align: center">
            <a runat="server" href="~/">
                <asp:Image  ID="Image1" runat="server" ImageUrl="~/Images/logo.jpg" BorderStyle="None"
/>
            </a>
            <br />
        </div>
        <div class="container body-content">
            <asp:ContentPlaceHolder ID="MainContent" runat="server">
            </asp:ContentPlaceHolder>
            <hr />
            <footer>
                <p>&copy; <%: DateTime.Now.Year %> - Wingtip Toys</p>
            </footer>
        </div>
    </form>
</body>
</html>
```

This HTML will display the image named *logo.jpg* from the *Images* folder of the Web application, which you'll add later. When a page that uses the master page is displayed in a browser, the logo will be displayed. If a user clicks on the logo, the user will navigate back to the *Default.aspx* page. The HTML anchor tag `<a>` wraps the image server control and allows the image to be included as part of the link. The `href` attribute for the anchor tag specifies the root "`~/`" of the Web site as the link location. By default, the *Default.aspx* page is displayed when the user navigates to the root of the Web site. The **Image** `<asp:Image>` server control includes addition properties, such as `BorderStyle`, that render as HTML when displayed in a browser.

Master Pages

A master page is an ASP.NET file with the extension .master (for example, *Site.Master*) with a predefined layout that can include static text, HTML elements, and server controls. The master page is identified by a special `@Master` directive that replaces the `@Page` directive that is used for ordinary *.aspx* pages.

In addition to the `@Master` directive, the master page also contains all of the top-level HTML elements for a page, such as `html`, `head`, and `form`. For example, on the master page you added above, you use an HTML `table` for the layout, an `img` element for the company logo, static text, and server controls to handle common membership for your site. You can use any HTML and any ASP.NET elements as part of your master page.

In addition to static text and controls that will appear on all pages, the master page also includes one or more **ContentPlaceHolder** controls. These placeholder controls define regions where replaceable content will appear. In turn, the replaceable content is defined in content pages, such as *Default.aspx*, using the **Content** server control.

Adding Image Files

The logo image that is referenced above, along with all the product images, must be added to the Web application so that they can be seen when the project is displayed in a browser.

Download from MSDN Samples site:

Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013 - Wingtip Toys (C#)

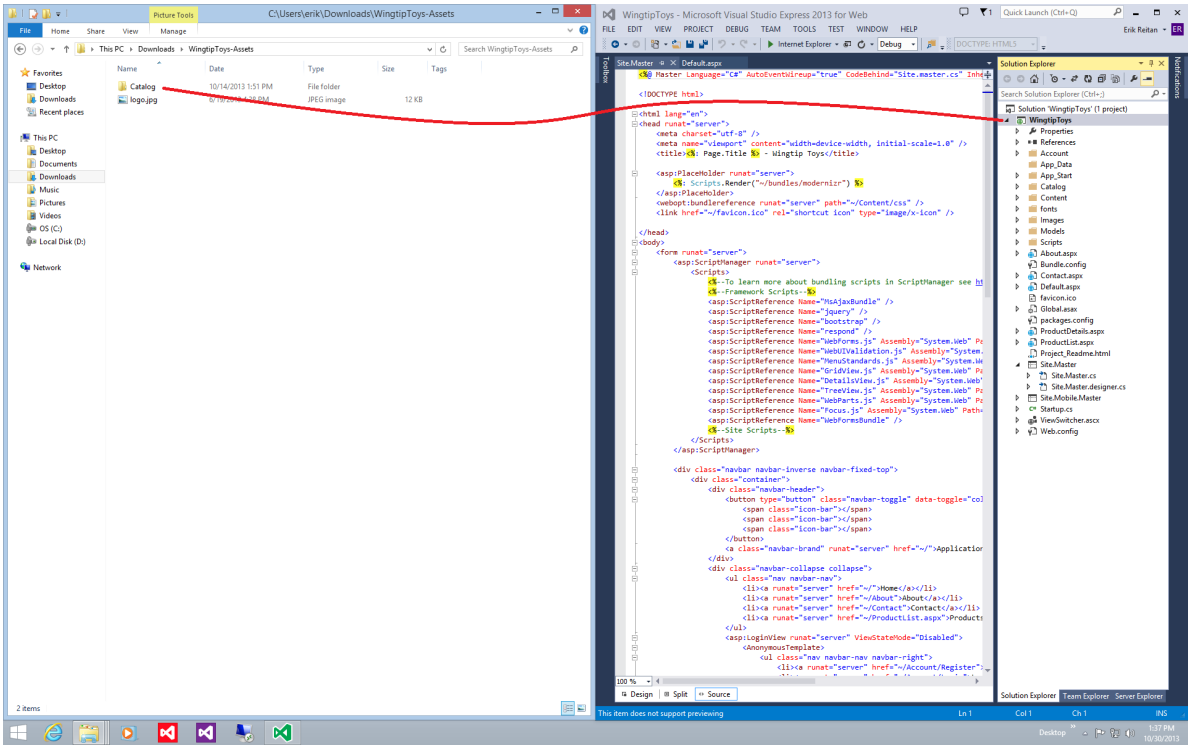
The download includes resources in the *WingtipToys-Assets* folder that are used to create the sample application.

1. If you haven't already done so, download the compressed sample files using the above link from the MSDN Samples site.

2. Once downloaded, open the .zip file and copy the contents to a local folder on your machine.

3. Find and open the *WingtipToys-Assets* folder.

4. By dragging and dropping, copy the *Catalog* folder from your local folder to the root of the Web application project in the **Solution Explorer** of Visual Studio.

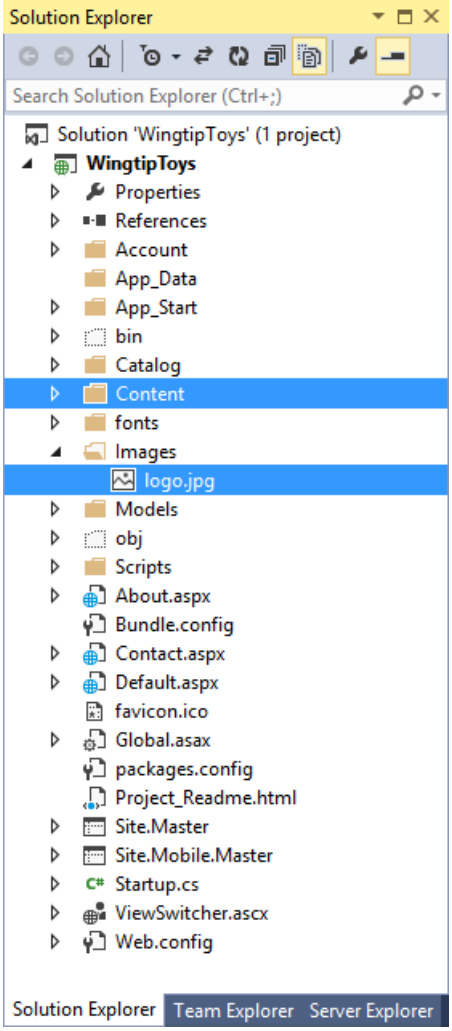


5. Next, create a new folder named *Images* by right-clicking the **WingtipToys** project in **Solution Explorer** and selecting **Add** -> **New Folder**.

6. Copy the *logo.jpg* file from the *WingtipToys-Assets* folder in **File Explorer** to the *Images* folder of the Web application project in **Solution Explorer** of Visual Studio.

7. Click the **Show All Files** option at the top of **Solution Explorer** to update the list of files if you don't see the new files.

   Solution Explorer now shows the updated project files.

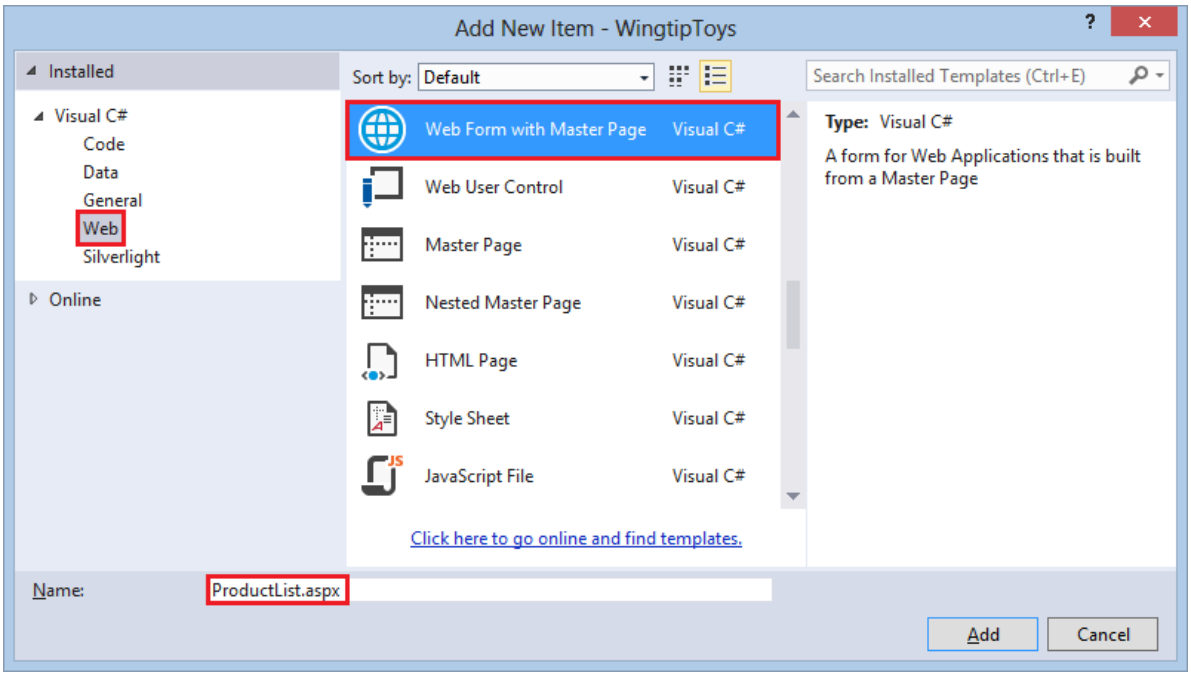


Adding Pages

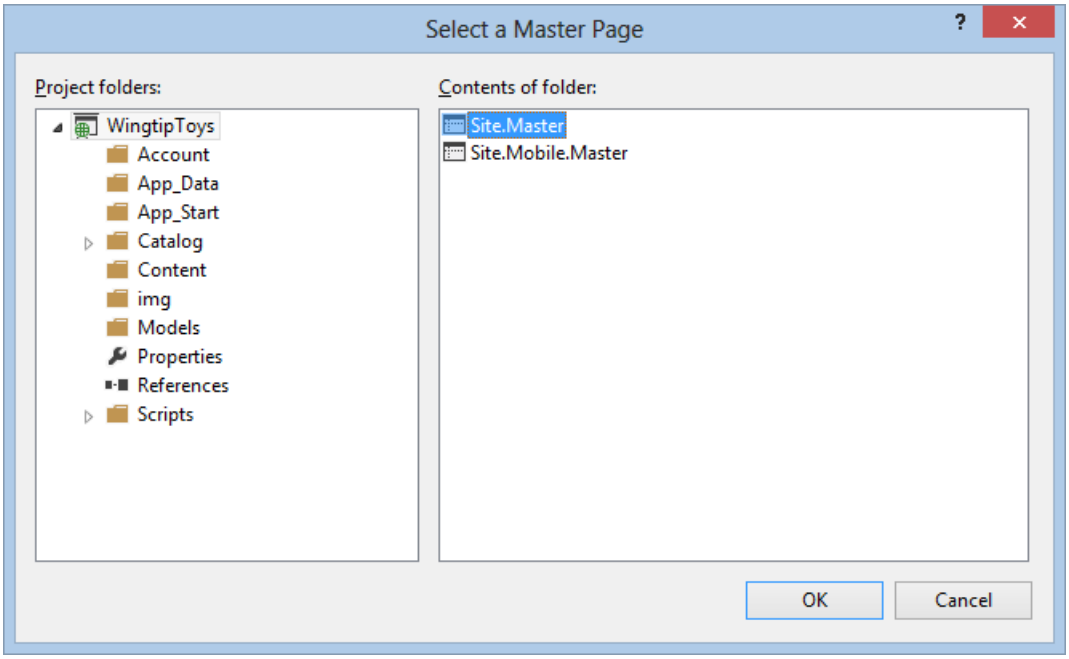
Before adding navigation to the Web application, you'll first add two new pages that you'll navigate to. Later in this tutorial series, you'll display products and product details on these new pages.

1. In **Solution Explorer**, right-click **WingtipToys**, click **Add**, and then click **New Item**.
   The **Add New Item** dialog box is displayed.

2. Select the **Visual C#** -> **Web** templates group on the left. Then, select **Web Form with Master Page** from the middle list and name it *ProductList.aspx*.



3. Select **Site.Master** to attach the master page to the newly created *.aspx* page.



4. Add an additional page named *ProductDetails.aspx* by following these same steps.

Updating Bootstrap

The Visual Studio 2013 project templates use Bootstrap, a layout and theming framework created by Twitter. Bootstrap uses CSS3 to provide responsive design, which means layouts can dynamically adapt to different browser window sizes. You can also use Bootstrap's theming feature to easily effect a change in the application's look and feel. By default, the ASP.NET Web Application template in Visual Studio 2013 includes Bootstrap as a NuGet package.

In this tutorial, you will change look and feel of the Wingtip Toys application by replacing the Bootstrap CSS files.

1. In **Solution Explorer**, open the *Content* folder.

2. Right-click the *bootstrap.css* file and rename it to *bootstrap-original.css*.

3. Rename the *bootstrap.min.css* to *bootstrap-original.min.css*.

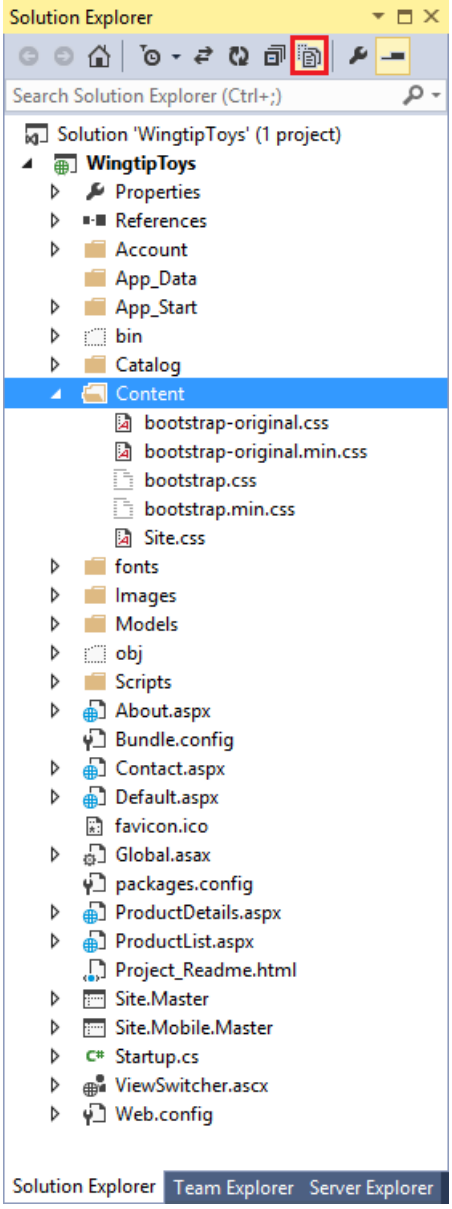
4. In **Solution Explorer**, right-click the *Content* folder and select **Open Folder in File Explorer**.
   The File Explorer will be displayed. You will save a downloaded bootstrap CSS files to this location.

5. In your browser, go to https://bootswatch.com/3/.

6. Scroll the browser window until you see the Cerulean theme.

7. Download both the *bootstrap.css* file and the *bootstrap.min.css* file to the *Content* folder. Use the path to the content folder that is displayed in the **File Explorer** window that you previously opened.

8. In **Visual Studio** at the top of **Solution Explorer**, select the **Show All Files** option to display the new files in the Content folder.



You will see the two new CSS files in the **Content** folder, but notice that the icon next to each file name is grayed out. This means that the file has not yet been added to the project.

9. Right-click the *bootstrap.css* and the *bootstrap.min.css* files and select **Include In Project**.
   When you run the Wingtip Toys application later in this tutorial, the new UI will be displayed.

> **NOTE**
>
> The ASP.NET Web Application template uses the *Bundle.config* file at the root of the project to store the path of the Bootstrap CSS files.

Modifying the Default Navigation

The default navigation for every page in the application can be modified by changing the unordered navigation list element that's in the *Site.Master* page.

1. In **Solution Explorer**, locate and open the *Site.Master* page.

2. Add the additional navigation link highlighted in yellow to the unordered list shown below:

```
<ul class="nav navbar-nav">
    <li><a runat="server" href="~/">Home</a></li>
    <li><a runat="server" href="~/About">About</a></li>
    <li><a runat="server" href="~/Contact">Contact</a></li>
    <li><a runat="server" href="~/ProductList">Products</a></li>
</ul>
```

As you can see in the above HTML, you modified each line item `<li>` containing an anchor tag `<a>` with a link `href` attribute. Each `href` points to a page in the Web application. In the browser, when a user clicks on one of these links (such as **Products**), they will navigate to the page contained in the `href` (such as **ProductList.aspx**). You will run the application at the end of this tutorial.

> **NOTE**
>
> The tilde (`~`) character is used to specify that the `href` path begins at the root of the project.

Adding a Data Control to Display Navigation Data

Next, you'll add a control to display all of the categories from the database. Each category will act as a link to the *ProductList.aspx* page. When a user clicks on a category link in the browser, they will navigate to the products page and see only the products associated with the selected category.

You'll use a **ListView** control to display all the categories contained in the database. To add a **ListView** control to the master page:

1. In the *Site.Master* page, add the following highlighted `<div>` element **after** the `<div>` element containing the `id="TitleContent"` that you added earlier:

```
<div id="TitleContent" style="text-align: center">
            <a runat="server" href="~/">
                <asp:Image  ID="Image1" runat="server" ImageUrl="~/img/logo.jpg" BorderStyle="None" />
            </a>
            <br />
        </div>
        <div id="CategoryMenu" style="text-align: center">
            <asp:ListView ID="categoryList"
                ItemType="WingtipToys.Models.Category"
                runat="server"
                SelectMethod="GetCategories" >
                <ItemTemplate>
                    <b style="font-size: large; font-style: normal">
                        <a href="/ProductList.aspx?id=<%#: Item.CategoryID %>">
                        <%#: Item.CategoryName %>
                        </a>
                    </b>
                </ItemTemplate>
                <ItemSeparatorTemplate>  |  </ItemSeparatorTemplate>
            </asp:ListView>
        </div>
```

This code will display all the categories from the database. The **ListView** control displays each category name as link text and includes a link to the *ProductList.aspx* page with a query-string value containing the `ID` of the category. By setting the `ItemType` property in the **ListView** control, the data-binding expression `Item` is available within the `ItemTemplate` node and the control becomes strongly typed. You can select details of the `Item` object using IntelliSense, such as specifying the `CategoryName`. This code is contained inside the container `<%#: %>` that marks a data-binding expression. By adding the (:) to the end of the `<%#` prefix, the result of the data-binding expression is HTML-encoded. When the result is HTML-encoded, your application is better protected against cross-site script injection (XSS) and HTML injection attacks.

> **NOTE**
>
> **Tip**
>
> When you add code by typing during development, you can be certain that a valid member of an object is found because strongly typed data controls show the available members based on IntelliSense. IntelliSense offers context-appropriate code choices as you type code, such as properties, methods, and objects.

In the next step, you will implement the `GetCategories` method to retrieve data.

Linking the Data Control to the Database

Before you can display data in the data control, you need to link the data control to the database. To make the link, you can modify the code behind of the *Site.Master.cs* file.

1. In **Solution Explorer**, right-click the *Site.Master* page and then click **View Code**. The *Site.Master.cs* file is opened in the editor.

2. Near the beginning of the *Site.Master.cs* file, add two additional namespaces so that all the included namespaces appear as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Linq;
using WingtipToys.Models;
```

3. Add the highlighted `GetCategories` method after the `Page_Load` event handler as follows:

```
protected void Page_Load(object sender, EventArgs e)
{

}

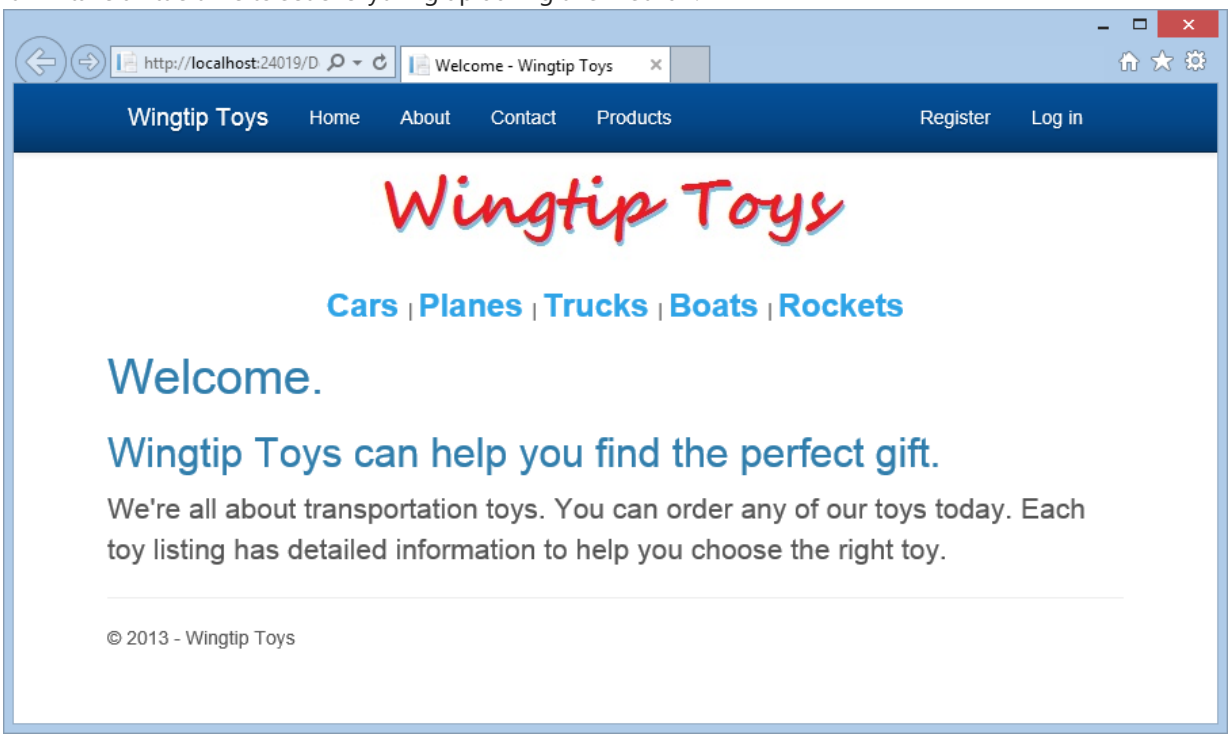
public IQueryable<Category> GetCategories()
{
  var _db = new WingtipToys.Models.ProductContext();
  IQueryable<Category> query = _db.Categories;
  return query;
}
```

The above code is executed when any page that uses the master page is loaded in the browser. The `ListView` control (named "categoryList") that you added earlier in this tutorial uses model binding to select data. In the markup of the `ListView` control you set the control's `SelectMethod` property to the `GetCategories` method, shown above. The `ListView` control calls the `GetCategories` method at the appropriate time in the page life cycle and automatically binds the returned data. You will learn more about binding data in the next tutorial.

Running the Application and Creating the Database

Earlier in this tutorial series you created an initializer class (named "ProductDatabaseInitializer") and specified this class in the *global.asax.cs* file. The Entity Framework will generate the database when the application is run the first time because the `Application_Start` method contained in the *global.asax.cs* file will call the initializer class. The initializer class will use the model classes (`Category` and `Product`) that you added earlier in this tutorial series to create the database.

1. In **Solution Explorer**, right-click the *Default.aspx* page and select **Set As Start Page**.
2. In Visual Studio press **F5**.
   It will take a little time to set everything up during this first run.



   When you run the application, the application will be compiled and the database named *wingtiptoys.mdf* will be created in the *App_Data* folder. In the browser, you will see a category navigation menu. This menu was generated by retrieving the categories from the database. In the next tutorial, you will implement the navigation.
3. Close the browser to stop the running application.

Reviewing the Database

Open the *Web.config* file and look at the connection string section. You can see that the `AttachDbFilename` value in the connection string points to the `DataDirectory` for the Web application project. The value `|DataDirectory|` is a reserved value that represents the *App_Data* folder in the project. This folder is where the database that was created from your entity classes is located.

```
<connectionStrings>
    <add name="DefaultConnection"
         connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=aspnet-WingtipToys-
20120302100502;Integrated Security=True"
         providerName="System.Data.SqlClient" />
    <add name="WingtipToys"
         connectionString="Data Source=
(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\wingtiptoys.mdf;Integrated Security=True"
         providerName="System.Data.SqlClient " />
  </connectionStrings>
```

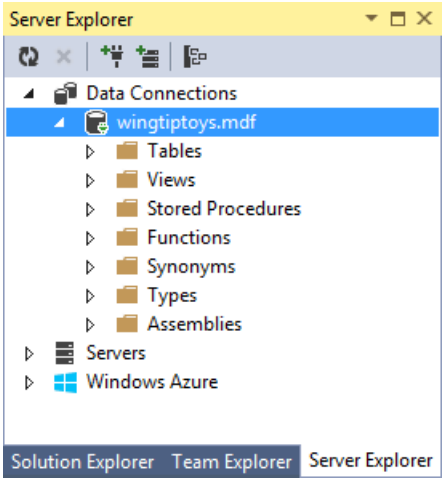
> **NOTE**
>
> If the *App_Data* folder is not visible or if the folder is empty, select the **Refresh** icon and then the **Show All Files** icon at the top of the **Solution Explorer** window. Expanding the width of the **Solution Explorer** windows may be required to show all available icons.

Now you can inspect the data contained in the *wingtiptoys.mdf* database file by using the **Server Explorer** window.

1. Expand the *App_Data* folder. If the *App_Data* folder is not visible, see the note above.

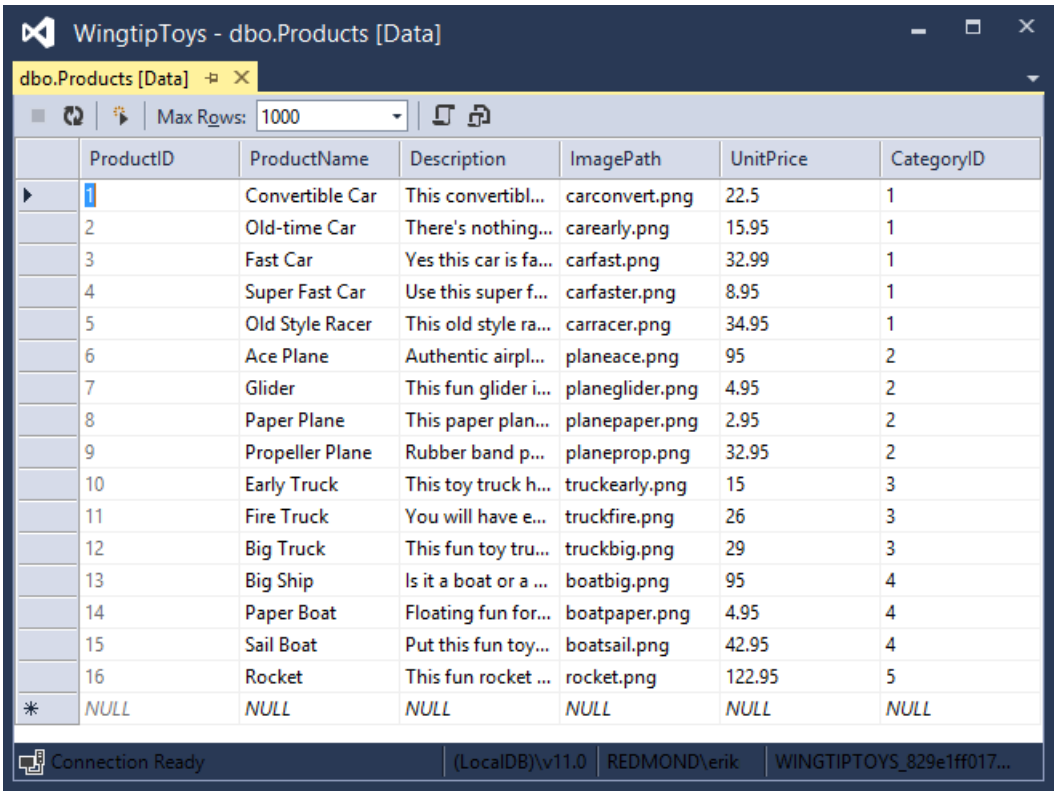
2. If the *wingtiptoys.mdf* database file is not visible, select the **Refresh** icon and then the **Show All Files** icon at the top of the **Solution Explorer** window.

3. Right-click the *wingtiptoys.mdf* database file and select **Open**.
   Server Explorer is displayed.



4. Expand the *Tables* folder.

5. Right-click the **Products** table and select **Show Table Data**.
   The **Products** table is displayed.

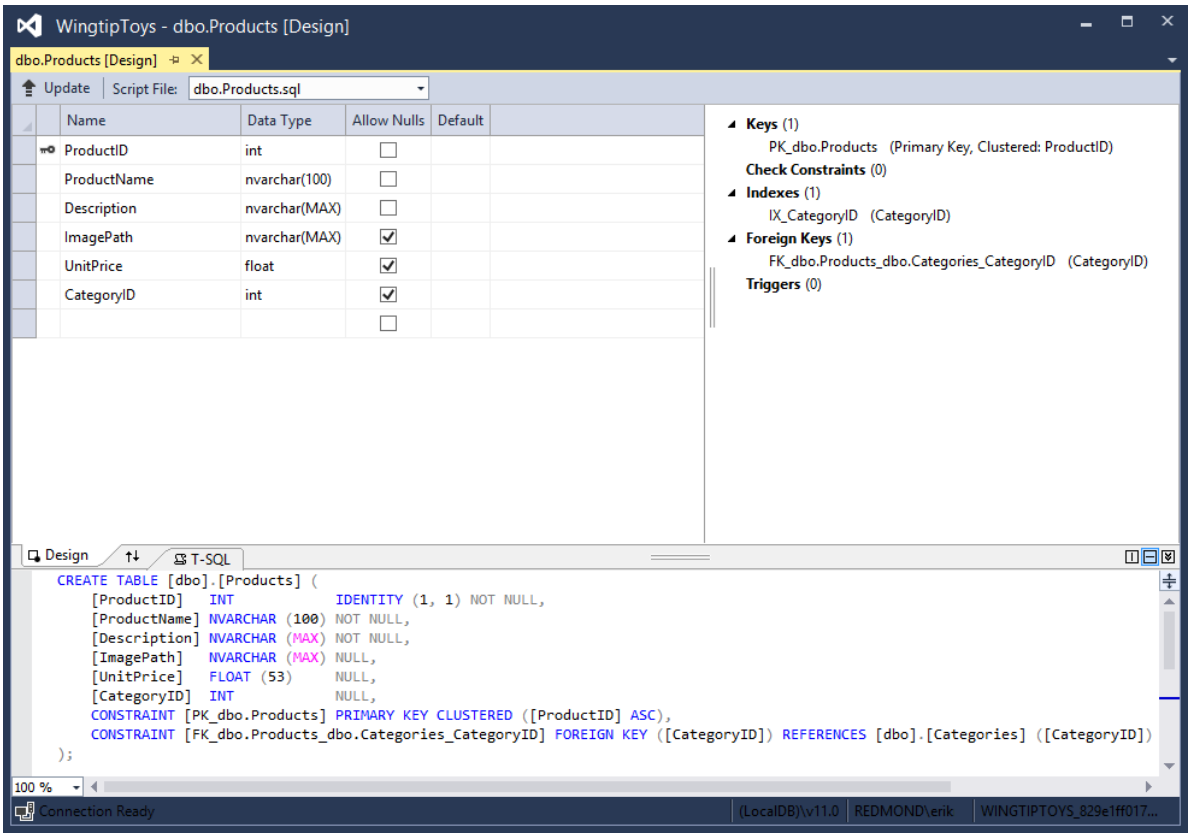


ProductID	ProductName	Description	ImagePath	UnitPrice	CategoryID
1	Convertible Car	This convertibl...	carconvert.png	22.5	1
2	Old-time Car	There's nothing...	carearly.png	15.95	1
3	Fast Car	Yes this car is fa...	carfast.png	32.99	1
4	Super Fast Car	Use this super f...	carfaster.png	8.95	1
5	Old Style Racer	This old style ra...	carracer.png	34.95	1
6	Ace Plane	Authentic airpl...	planeace.png	95	2
7	Glider	This fun glider i...	planeglider.png	4.95	2
8	Paper Plane	This paper plan...	planepaper.png	2.95	2
9	Propeller Plane	Rubber band p...	planeprop.png	32.95	2
10	Early Truck	This toy truck h...	truckearly.png	15	3
11	Fire Truck	You will have e...	truckfire.png	26	3
12	Big Truck	This fun toy tru...	truckbig.png	29	3
13	Big Ship	Is it a boat or a ...	boatbig.png	95	4
14	Paper Boat	Floating fun for...	boatpaper.png	4.95	4
15	Sail Boat	Put this fun toy...	boatsail.png	42.95	4
16	Rocket	This fun rocket ...	rocket.png	122.95	5
NULL	NULL	NULL	NULL	NULL	NULL

6. This view lets you see and modify the data in the **Products** table by hand.

7. Close the **Products** table window.

8. In the **Server Explorer**, right-click the **Products** table again and select **Open Table Definition**.
   The data design for the **Products** table is displayed.



9. In the **T-SQL** tab you will see the SQL DDL statement that was used to create the table. You can also use the UI in the **Design** tab to modify the schema.

10. In the **Server Explorer**, right-click **WingtipToys** database and select **Close Connection**.
    By detaching the database from Visual Studio, the database schema will be able to be modified later in this tutorial series.

11. Return to **Solution Explorer** by selecting the **Solution Explorer** tab at the bottom of the **Server Explorer** window.

Summary

In this tutorial of the series you have added some basic UI, graphics, pages, and navigation. Additionally, you ran the Web application, which created the database from the data classes that you added in the previous tutorial. You also viewed the contents of the *Products* table of the database by viewing the database directly. In the next tutorial, you'll display data items and details from the database.

Additional Resources

Introduction to Programming ASP.NET Web Pages
ASP.NET Web Server Controls Overview
CSS Tutorial

PREVIOUS NEXT

Display data items and details

2/20/2020 • 7 minutes to read • Edit Online

by Erik Reitan

> This tutorial series teaches you the basics of building an ASP.NET Web Forms application with ASP.NET 4.7 and Microsoft Visual Studio 2017.

In this tutorial, you'll learn how to display data items and data item details with ASP.NET Web Forms and Entity Framework Code First. This tutorial builds on the previous "UI and Navigation" tutorial as part of the Wingtip Toy Store tutorial series. After completing this tutorial, you'll see products on the *ProductsList.aspx* page and a product's details on the *ProductDetails.aspx* page.

You'll learn how to:

- Add a data control to display products from the database
- Connect a data control to the selected data
- Add a data control to display product details from the database
- Retrieve a value from the query string and use that value to limit the data that's retrieved from the database

Features introduced in this tutorial:

- Model binding
- Value providers

Add a data control

You can use a few different options to bind data to a server control. The most common include:

- Adding a data source control
- Adding code by hand
- Using model binding

Use a data source control to bind data

Adding a data source control allows you to link the data source control to the control that displays the data. With this approach, you can declaratively, rather than programmatically, connect server-side controls to data sources.

Code by hand to bind data

Coding by hand involves:

1. Reading a value
2. Checking if it's null
3. Converting it to an appropriate type
4. Checking conversion success
5. Using the value in the query

This approach lets you have full control over your data-access logic.

Use model binding to bind data

Model binding lets you bind results with far less code and gives you the ability to reuse the functionality throughout your application. It simplifies working with code-focused data-access logic while still providing a rich,

data-binding framework.

Display products

In this tutorial, you'll use model binding to bind data. To configure a data control to use model binding to select data, you set the control's `SelectMethod` property to a method name in the page's code. The data control calls the method at the appropriate time in the page life cycle and automatically binds the returned data. There's no need to explicitly call the `DataBind` method.

1. In **Solution Explorer**, open *ProductList.aspx*.

2. Replace the existing markup with this markup:

```
<%@ Page Title="Products" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
         CodeBehind="ProductList.aspx.cs" Inherits="WingtipToys.ProductList" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <section>
        <div>
            <hgroup>
                <h2><%: Page.Title %></h2>
            </hgroup>

            <asp:ListView ID="productList" runat="server"
                DataKeyNames="ProductID" GroupItemCount="4"
                ItemType="WingtipToys.Models.Product" SelectMethod="GetProducts">
                <EmptyDataTemplate>
                    <table >
                        <tr>
                            <td>No data was returned.</td>
                        </tr>
                    </table>
                </EmptyDataTemplate>
                <EmptyItemTemplate>
                    <td/>
                </EmptyItemTemplate>
                <GroupTemplate>
                    <tr id="itemPlaceholderContainer" runat="server">
                        <td id="itemPlaceholder" runat="server"></td>
                    </tr>
                </GroupTemplate>
                <ItemTemplate>
                    <td runat="server">
                        <table>
                            <tr>
                                <td>
                                    <a href="ProductDetails.aspx?productID=<%#:Item.ProductID%>">
                                        <img src="/Catalog/Images/Thumbs/<%#:Item.ImagePath%>"
                                            width="100" height="75" style="border: solid" /></a>
                                </td>
                            </tr>
                            <tr>
                                <td>
                                    <a href="ProductDetails.aspx?productID=<%#:Item.ProductID%>">
                                        <span>
                                            <%#:Item.ProductName%>
                                        </span>
                                    </a>
                                    <br />
                                    <span>
                                        <b>Price: </b><%#:String.Format("{0:c}", Item.UnitPrice)%>
                                    </span>
                                    <br />
                                </td>
                            </tr>
                            <tr>
                                <td> </td>
```

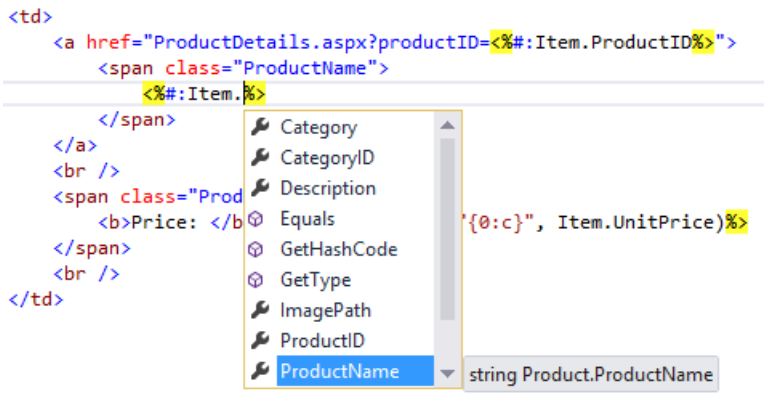
```
                            </tr>
                        </table>
                        </p>
                    </td>
                </ItemTemplate>
                <LayoutTemplate>
                    <table style="width:100%;">
                        <tbody>
                            <tr>
                                <td>
                                    <table id="groupPlaceholderContainer" runat="server"
style="width:100%">
                                        <tr id="groupPlaceholder"></tr>
                                    </table>
                                </td>
                            </tr>
                            <tr>
                                <td></td>
                            </tr>
                            <tr></tr>
                        </tbody>
                    </table>
                </LayoutTemplate>
            </asp:ListView>
        </div>
    </section>
</asp:Content>
```

This code uses a **ListView** control named `productList` to display products.

```
<asp:ListView ID="productList" runat="server"
```

With templates and styles, you define how the **ListView** control displays data. It's useful for data in any repeating structure. Though this **ListView** example simply displays database data, you can also, without code, enable users to edit, insert, and delete data, and to sort and page data.

By setting the `ItemType` property in the **ListView** control, the data-binding expression `Item` is available and the control becomes strongly typed. As mentioned in the previous tutorial, you can select Item object details with IntelliSense, such as specifying the `ProductName`:



You're also using model binding to specify a `SelectMethod` value. This value (`GetProducts`) corresponds to the method you'll add to the code behind to display products in the next step.

Add code to display products

In this step, you'll add code to populate the **ListView** control with product data from the database. The code supports showing all products and individual category products.

1. In **Solution Explorer**, right-click *ProductList.aspx* and then select **View Code**.

2. Replace the existing code in the *ProductList.aspx.cs* file with this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Models;
using System.Web.ModelBinding;

namespace WingtipToys
{
    public partial class ProductList : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }

        public IQueryable<Product> GetProducts([QueryString("id")] int? categoryId)
        {
            var _db = new WingtipToys.Models.ProductContext();
                IQueryable<Product> query = _db.Products;
                if (categoryId.HasValue && categoryId > 0)
                {
                    query = query.Where(p => p.CategoryID == categoryId);
                }
                return query;
        }
    }
}
```

This code shows the `GetProducts` method that the **ListView** control's `ItemType` property references in the *ProductList.aspx* page. To limit the results to a specific database category, the code sets the `categoryId` value from the query string value passed to the *ProductList.aspx* page when the *ProductList.aspx* page is navigated to. The `QueryStringAttribute` class in the `System.Web.ModelBinding` namespace is used to retrieve the value of the query string variable `id`. This instructs model binding to try to bind a value from the query string to the `categoryId` parameter at run time.

When a valid category is passed as a query string to the page, the results of the query are limited to those products in the database that match the `categoryId` value. For instance, if the *ProductsList.aspx* page URL is this:

```
http://localhost/ProductList.aspx?id=1
```

The page displays only the products where the `categoryId` equals `1`.

All products are displayed if no query string is included when the *ProductList.aspx* page is called.

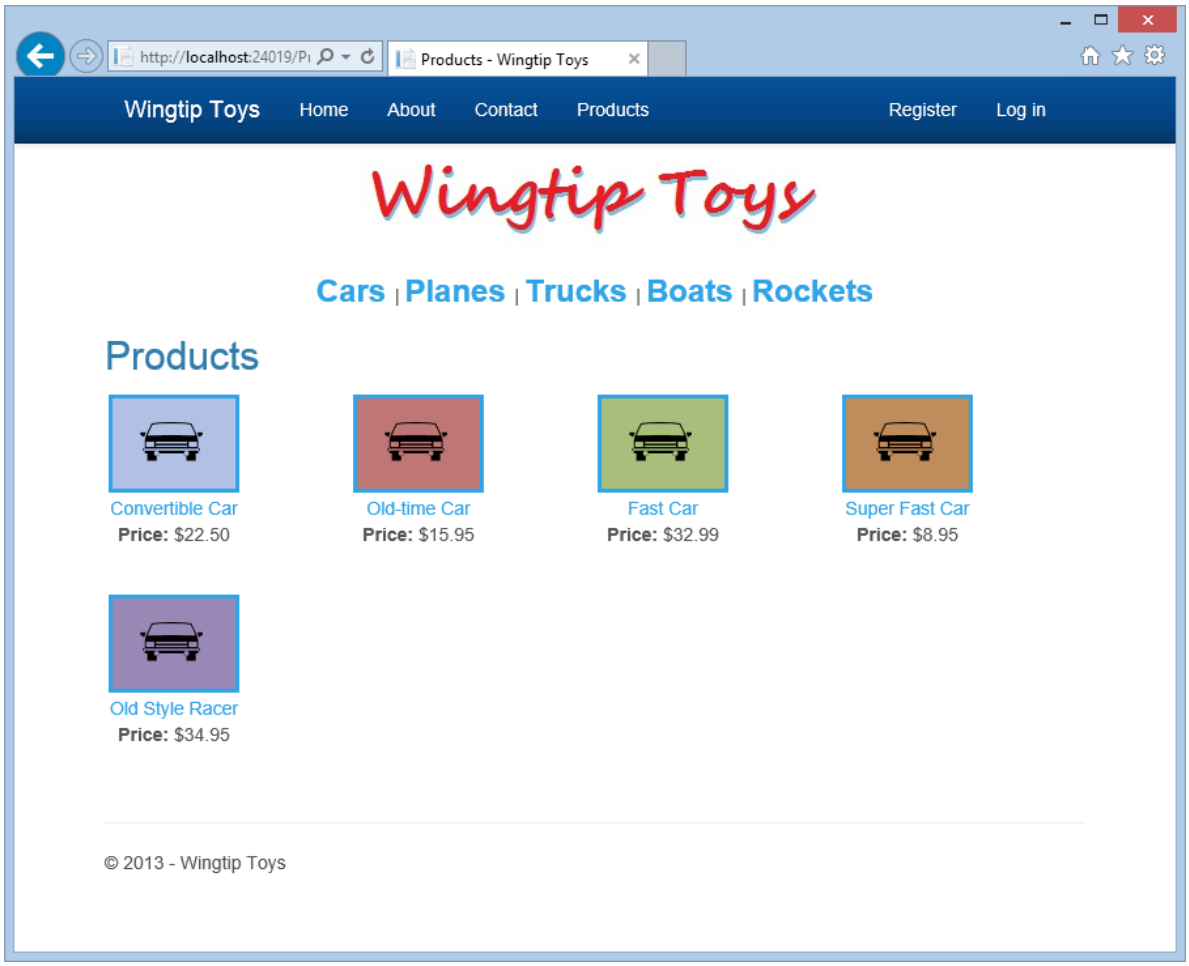
The sources of values for these methods are referred to as *value providers* (such as *QueryString*), and the parameter attributes that indicate which value provider to use are referred to as *value provider attributes* (such as `id`). ASP.NET includes value providers and corresponding attributes for all of the typical sources of user input in a Web Forms application such as the query string, cookies, form values, controls, view state, session state, and profile properties. You can also write custom value providers.

Run the application

Run the application now to view all products or a category's products.

1. Press **F5** while in Visual Studio to run the application.
   The browser opens and shows the *Default.aspx* page.

2. Select **Cars** from the product category navigation menu.
   The *ProductList.aspx* page displays showing only **Cars** category products. Later in this tutorial, you'll display product details.



3. Select **Products** from the navigation menu at the top.
   Again, the *ProductList.aspx* page is displayed, however this time it shows the entire list of products.

4. Close the browser and return to Visual Studio.

Add a data control to display product details

Next, you'll modify the markup in the *ProductDetails.aspx* page that you added in the previous tutorial to display specific product information.

1. In **Solution Explorer**, open *ProductDetails.aspx*.

2. Replace the existing markup with this markup:

```
<%@ Page Title="Product Details" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
         CodeBehind="ProductDetails.aspx.cs" Inherits="WingtipToys.ProductDetails" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <asp:FormView ID="productDetail" runat="server" ItemType="WingtipToys.Models.Product" SelectMethod
="GetProduct" RenderOuterTable="false">
        <ItemTemplate>
            <div>
                <h1><%#:Item.ProductName %></h1>
            </div>
            <br />
            <table>
                <tr>
                    <td>
                        <img src="/Catalog/Images/<%#:Item.ImagePath %>" style="border:solid;
height:300px" alt="<%#:Item.ProductName %>"/>
                    </td>
                    <td> </td>
                    <td style="vertical-align: top; text-align:left;">
                        <b>Description:</b><br /><%#:Item.Description %>
                        <br />
                        <span><b>Price:</b> <%#: String.Format("{0:c}", Item.UnitPrice) %></span>
                        <br />
                        <span><b>Product Number:</b> <%#:Item.ProductID %></span>
                        <br />
                    </td>
                </tr>
            </table>
        </ItemTemplate>
    </asp:FormView>
</asp:Content>
```

This code uses a **FormView** control to display specific product details. This markup uses methods like the methods used to display data in the *ProductList.aspx* page. The **FormView** control is used to display a single record at a time from a data source. When you use the **FormView** control, you create templates to display and edit data-bound values. These templates contain controls, binding expressions, and formatting that define the form's look and functionality.

Connecting the previous markup to the database requires additional code.

1. In **Solution Explorer**, right-click *ProductDetails.aspx* and then click **View Code**.
   The *ProductDetails.aspx.cs* file is displayed.

2. Replace the existing code with this code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Models;
using System.Web.ModelBinding;

namespace WingtipToys
{
    public partial class ProductDetails : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }

        public IQueryable<Product> GetProduct([QueryString("productID")] int? productId)
        {
            var _db = new WingtipToys.Models.ProductContext();
                IQueryable<Product> query = _db.Products;
                if (productId.HasValue && productId > 0)
                {
                    query = query.Where(p => p.ProductID == productId);
                }
                else
                {
                    query = null;
                }
                return query;
        }
    }
}
```

This code checks for a "`productID`" query-string value. If a valid query-string value is found, the matching product is displayed. If the query-string isn't found, or its value isn't valid, no product is displayed.

Run the application

Now you can run the application to see an individual product displayed based on product ID.

1. Press **F5** while in Visual Studio to run the application.
   The browser opens and shows the *Default.aspx* page.

2. Select **Boats** from the category navigation menu.
   The *ProductList.aspx* page is displayed.

3. Select **Paper Boat** from the product list. The *ProductDetails.aspx* page is displayed.

4. Close the browser.

Additional resources

Retrieving and displaying data with model binding and web forms

Next steps

In this tutorial, you added markup and code to display products and product details. You learned about strongly typed data controls, model binding, and value providers. In the next tutorial, you'll add a shopping cart to the Wingtip Toys sample application.

PREVIOUS NEXT

Shopping Cart

2/20/2020 • 29 minutes to read • Edit Online

by Erik Reitan

Download Wingtip Toys Sample Project (C#) or Download E-book (PDF)

> This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

This tutorial describes the business logic required to add a shopping cart to the Wingtip Toys sample ASP.NET Web Forms application. This tutorial builds on the previous tutorial "Display Data Items and Details" and is part of the Wingtip Toy Store tutorial series. When you've completed this tutorial, the users of your sample app will be able to add, remove, and modify the products in their shopping cart.

What you'll learn:

1. How to create a shopping cart for the web application.
2. How to enable users to add items to the shopping cart.
3. How to add a GridView control to display shopping cart details.
4. How to calculate and display the order total.
5. How to remove and update items in the shopping cart.
6. How to include a shopping cart counter.

Code features in this tutorial:

1. Entity Framework Code First
2. Data Annotations
3. Strongly typed data controls
4. Model binding

Creating a Shopping Cart

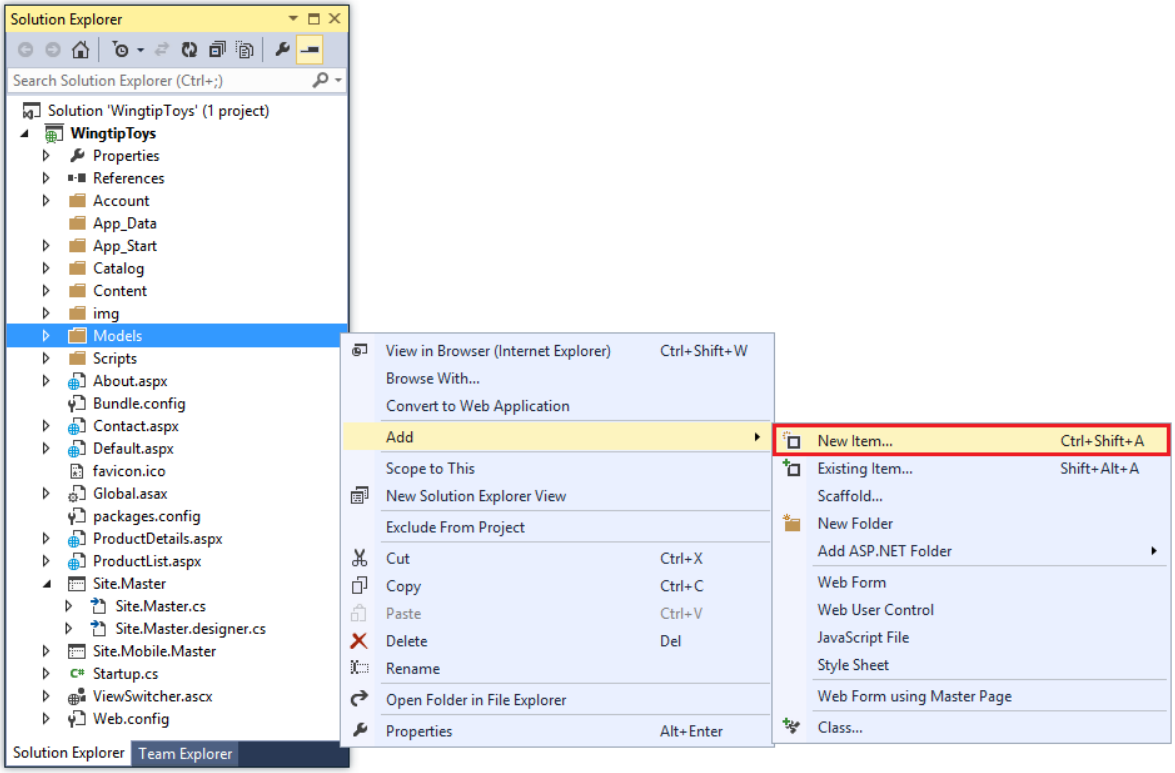
Earlier in this tutorial series, you added pages and code to view product data from a database. In this tutorial, you'll create a shopping cart to manage the products that users are interested in buying. Users will be able to browse and add items to the shopping cart even if they are not registered or logged in. To manage shopping cart access, you will assign users a unique `ID` using a globally unique identifier (GUID) when the user accesses the shopping cart for the first time. You'll store this `ID` using the ASP.NET Session state.

> **NOTE**
>
> The ASP.NET Session state is a convenient place to store user-specific information which will expire after the user leaves the site. While misuse of session state can have performance implications on larger sites, light use of session state works well for demonstration purposes. The Wingtip Toys sample project shows how to use session state without an external provider, where session state is stored in-process on the web server hosting the site. For larger sites that provide multiple instances of an application or for sites that run multiple instances of an application on different servers, consider using **Windows Azure Cache Service**. This Cache Service provides a distributed caching service that is external to the web site and solves the problem of using in-process session state. For more information see, How to Use ASP.NET Session State with Windows Azure Web Sites.

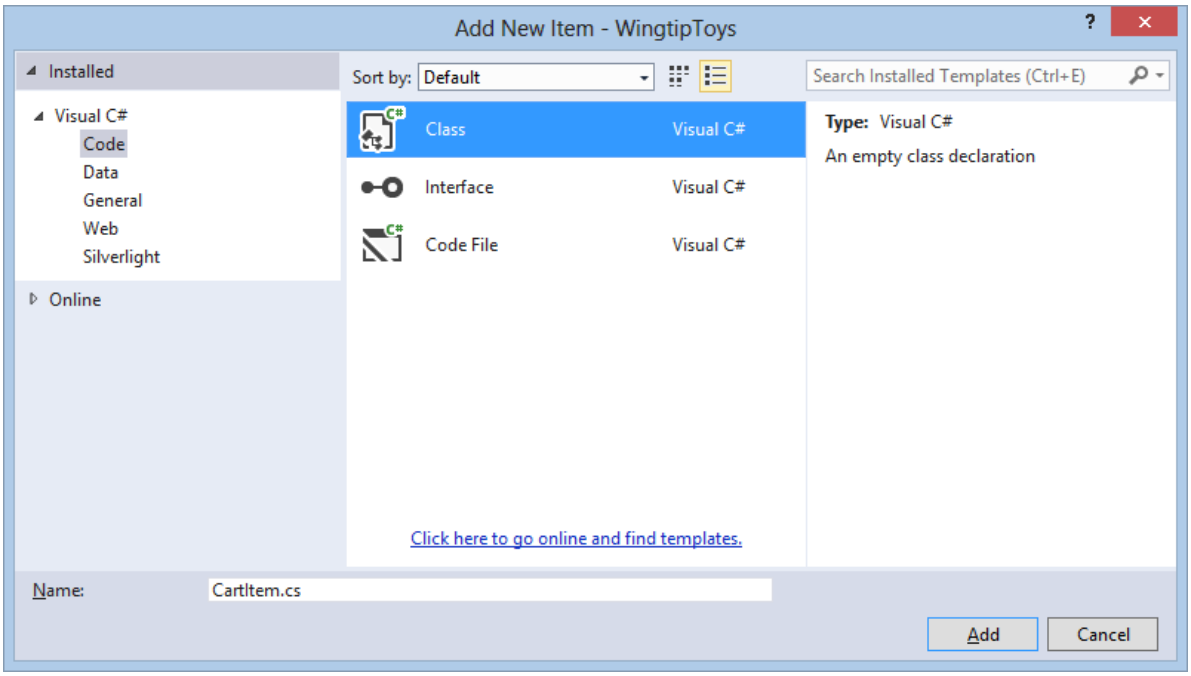
Add CartItem as a Model Class

Earlier in this tutorial series, you defined the schema for the category and product data by creating the `Category` and `Product` classes in the *Models* folder. Now, add a new class to define the schema for the shopping cart. Later in this tutorial, you will add a class to handle data access to the `CartItem` table. This class will provide the business logic to add, remove, and update items in the shopping cart.

1. Right-click the *Models* folder and select **Add** -> **New Item**.



2. The **Add New Item** dialog box is displayed. Select **Code**, and then select **Class**.



3. Name this new class *CartItem.cs*.

4. Click **Add**.
   The new class file is displayed in the editor.

5. Replace the default code with the following code:

```
using System.ComponentModel.DataAnnotations;

namespace WingtipToys.Models
{
    public class CartItem
    {
        [Key]
        public string ItemId { get; set; }

        public string CartId { get; set; }

        public int Quantity { get; set; }

        public System.DateTime DateCreated { get; set; }

        public int ProductId { get; set; }

        public virtual Product Product { get; set; }

    }
}
```

The `CartItem` class contains the schema that will define each product a user adds to the shopping cart. This class is similar to the other schema classes you created earlier in this tutorial series. By convention, Entity Framework Code First expects that the primary key for the `CartItem` table will be either `CartItemId` or `ID`. However, the code overrides the default behavior by using the data annotation `[Key]` attribute. The `Key` attribute of the ItemId property specifies that the `ItemID` property is the primary key.

The `CartId` property specifies the `ID` of the user that is associated with the item to purchase. You'll add code to create this user `ID` when the user accesses the shopping cart. This `ID` will also be stored as an ASP.NET Session variable.

Update the Product Context

In addition to adding the `CartItem` class, you will need to update the database context class that manages the entity classes and that provides data access to the database. To do this, you will add the newly created `CartItem` model class to the `ProductContext` class.

1. In **Solution Explorer**, find and open the *ProductContext.cs* file in the *Models* folder.

2. Add the highlighted code to the *ProductContext.cs* file as follows:

```
using System.Data.Entity;

namespace WingtipToys.Models
{
    public class ProductContext : DbContext
    {
        public ProductContext()
            : base("WingtipToys")
        {
        }

        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }
        public DbSet<CartItem> ShoppingCartItems { get; set; }
    }
}
```

As mentioned previously in this tutorial series, the code in the *ProductContext.cs* file adds the `System.Data.Entity` namespace so that you have access to all the core functionality of the Entity Framework. This functionality includes the capability to query, insert, update, and delete data by working with strongly typed objects. The `ProductContext` class adds access to the newly added `CartItem` model class.

Managing the Shopping Cart Business Logic

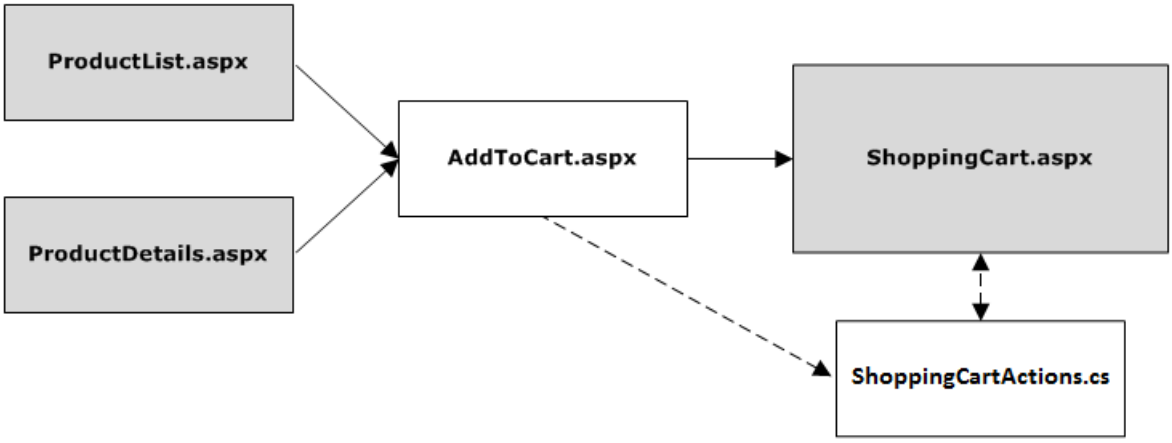
Next, you'll create the `ShoppingCart` class in a new *Logic* folder. The `ShoppingCart` class handles data access to the `CartItem` table. The class will also include the business logic to add, remove, and update items in the shopping cart.

The shopping cart logic that you will add will contain the functionality to manage the following actions:

1. Adding items to the shopping cart
2. Removing items from the shopping cart
3. Getting the shopping cart ID
4. Retrieving items from the shopping cart
5. Totaling the amount of all the shopping cart items
6. Updating the shopping cart data

A shopping cart page (*ShoppingCart.aspx*) and the shopping cart class will be used together to access shopping cart data. The shopping cart page will display all the items the user adds to the shopping cart. Besides the shopping cart page and class, you'll create a page (*AddToCart.aspx*) to add products to the shopping cart. You will also add code to the *ProductList.aspx* page and the *ProductDetails.aspx* page that will provide a link to the *AddToCart.aspx* page, so that the user can add products to the shopping cart.

The following diagram shows the basic process that occurs when the user adds a product to the shopping cart.



When the user clicks the **Add To Cart** link on either the *ProductList.aspx* page or the *ProductDetails.aspx* page, the application will navigate to the *AddToCart.aspx* page and then automatically to the *ShoppingCart.aspx* page. The *AddToCart.aspx* page will add the select product to the shopping cart by calling a method in the ShoppingCart class. The *ShoppingCart.aspx* page will display the products that have been added to the shopping cart.

Creating the Shopping Cart Class

The `ShoppingCart` class will be added to a separate folder in the application so that there will be a clear distinction between the model (Models folder), the pages (root folder) and the logic (Logic folder).

1. In **Solution Explorer**, right-click the **WingtipToys** project and select **Add->New Folder**. Name the new folder *Logic*.
2. Right-click the *Logic* folder and then select **Add -> New Item**.
3. Add a new class file named *ShoppingCartActions.cs*.
4. Replace the default code with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using WingtipToys.Models;

namespace WingtipToys.Logic
{
  public class ShoppingCartActions : IDisposable
  {
    public string ShoppingCartId { get; set; }

    private ProductContext _db = new ProductContext();

    public const string CartSessionKey = "CartId";

    public void AddToCart(int id)
    {
      // Retrieve the product from the database.
      ShoppingCartId = GetCartId();

      var cartItem = _db.ShoppingCartItems.SingleOrDefault(
          c => c.CartId == ShoppingCartId
          && c.ProductId == id);
      if (cartItem == null)
      {
        // Create a new cart item if no cart item exists.
        cartItem = new CartItem
        {
          ItemId = Guid.NewGuid().ToString(),
```

```
            ProductId = id,
            CartId = ShoppingCartId,
            Product = _db.Products.SingleOrDefault(
             p => p.ProductID == id),
            Quantity = 1,
            DateCreated = DateTime.Now
          };

          _db.ShoppingCartItems.Add(cartItem);
        }
        else
        {
          // If the item does exist in the cart,
          // then add one to the quantity.
          cartItem.Quantity++;
        }
        _db.SaveChanges();
      }

      public void Dispose()
      {
        if (_db != null)
        {
          _db.Dispose();
          _db = null;
        }
      }

      public string GetCartId()
      {
        if (HttpContext.Current.Session[CartSessionKey] == null)
        {
          if (!string.IsNullOrWhiteSpace(HttpContext.Current.User.Identity.Name))
          {
            HttpContext.Current.Session[CartSessionKey] = HttpContext.Current.User.Identity.Name;
          }
          else
          {
            // Generate a new random GUID using System.Guid class.
            Guid tempCartId = Guid.NewGuid();
            HttpContext.Current.Session[CartSessionKey] = tempCartId.ToString();
          }
        }
        return HttpContext.Current.Session[CartSessionKey].ToString();
      }

      public List<CartItem> GetCartItems()
      {
        ShoppingCartId = GetCartId();

        return _db.ShoppingCartItems.Where(
            c => c.CartId == ShoppingCartId).ToList();
      }
    }
  }
```

The `AddToCart` method enables individual products to be included in the shopping cart based on the product `ID`. The product is added to the cart, or if the cart already contains an item for that product, the quantity is incremented.

The `GetCartId` method returns the cart `ID` for the user. The cart `ID` is used to track the items that a user has in their shopping cart. If the user does not have an existing cart `ID`, a new cart `ID` is created for them. If the user is signed in as a registered user, the cart `ID` is set to their user name. However, if the user is not signed in, the cart `ID` is set to a unique value (a GUID). A GUID ensures that only one cart is created for each user, based on session.

The `GetCartItems` method returns a list of shopping cart items for the user. Later in this tutorial, you will see that model binding is used to display the cart items in the shopping cart using the `GetCartItems` method.

Creating the Add-To-Cart Functionality

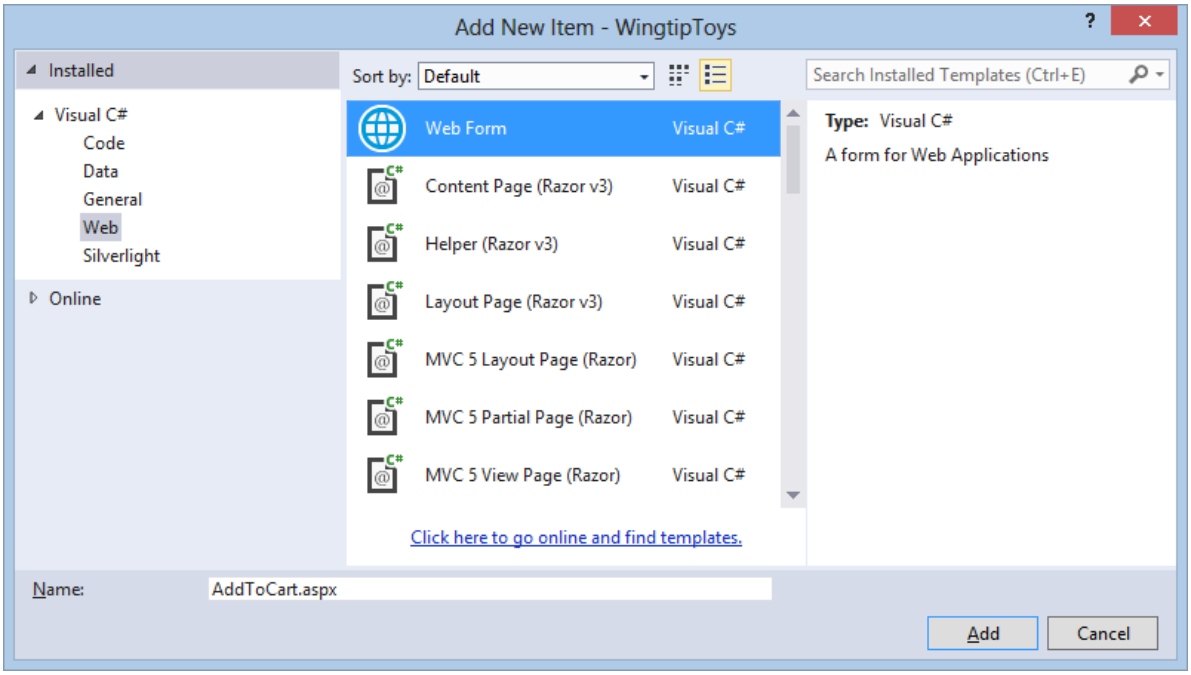
As mentioned earlier, you will create a processing page named *AddToCart.aspx* that will be used to add new products to the shopping cart of the user. This page will call the `AddToCart` method in the `ShoppingCart` class that you just created. The *AddToCart.aspx* page will expect that a product `ID` is passed to it. This product `ID` will be used when calling the `AddToCart` method in the `ShoppingCart` class.

> **NOTE**
>
> You will be modifying the code-behind (*AddToCart.aspx.cs*) for this page, not the page UI (*AddToCart.aspx*).

To create the Add-To-Cart functionality:

1. In **Solution Explorer**, right-click the **WingtipToys** project, click **Add -> New Item**.
   The **Add New Item** dialog box is displayed.

2. Add a standard new page (Web Form) to the application named *AddToCart.aspx*.



3. In **Solution Explorer**, right-click the *AddToCart.aspx* page and then click **View Code**. The *AddToCart.aspx.cs* code-behind file is opened in the editor.

4. Replace the existing code in the *AddToCart.aspx.cs* code-behind with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Diagnostics;
using WingtipToys.Logic;

namespace WingtipToys
{
  public partial class AddToCart : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      string rawId = Request.QueryString["ProductID"];
      int productId;
      if (!String.IsNullOrEmpty(rawId) && int.TryParse(rawId, out productId))
      {
        using (ShoppingCartActions usersShoppingCart = new ShoppingCartActions())
        {
          usersShoppingCart.AddToCart(Convert.ToInt16(rawId));
        }

      }
      else
      {
        Debug.Fail("ERROR : We should never get to AddToCart.aspx without a ProductId.");
        throw new Exception("ERROR : It is illegal to load AddToCart.aspx without setting a
ProductId.");
      }
      Response.Redirect("ShoppingCart.aspx");
    }
  }
}
```

When the *AddToCart.aspx* page is loaded, the product `ID` is retrieved from the query string. Next, an instance of the shopping cart class is created and used to call the `AddToCart` method that you added earlier in this tutorial. The `AddToCart` method, contained in the *ShoppingCartActions.cs* file, includes the logic to add the selected product to the shopping cart or increment the product quantity of the selected product. If the product hasn't been added to the shopping cart, the product is added to the `CartItem` table of the database. If the product has already been added to the shopping cart and the user adds an additional item of the same product, the product quantity is incremented in the `CartItem` table. Finally, the page redirects back to the *ShoppingCart.aspx* page that you'll add in the next step, where the user sees an updated list of items in the cart.

As previously mentioned, a user `ID` is used to identify the products that are associated with a specific user. This `ID` is added to a row in the `CartItem` table each time the user adds a product to the shopping cart.

Creating the Shopping Cart UI

The *ShoppingCart.aspx* page will display the products that the user has added to their shopping cart. It will also provide the ability to add, remove and update items in the shopping cart.

1. In **Solution Explorer**, right-click **WingtipToys**, click **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.

2. Add a new page (Web Form) that includes a master page by selecting **Web Form using Master Page**.
   Name the new page *ShoppingCart.aspx*.

3. Select **Site.Master** to attach the master page to the newly created *.aspx* page.

4. In the *ShoppingCart.aspx* page, replace the existing markup with the following markup:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="ShoppingCart.aspx.cs" Inherits="WingtipToys.ShoppingCart" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <div id="ShoppingCartTitle" runat="server" class="ContentHead"><h1>Shopping Cart</h1></div>
    <asp:GridView ID="CartList" runat="server" AutoGenerateColumns="False" ShowFooter="True"
GridLines="Vertical" CellPadding="4"
        ItemType="WingtipToys.Models.CartItem" SelectMethod="GetShoppingCartItems"
        CssClass="table table-striped table-bordered" >
        <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="ID" SortExpression="ProductID" />
        <asp:BoundField DataField="Product.ProductName" HeaderText="Name" />
        <asp:BoundField DataField="Product.UnitPrice" HeaderText="Price (each)" DataFormatString="
{0:c}"/>
        <asp:TemplateField   HeaderText="Quantity">
                <ItemTemplate>
                    <asp:TextBox ID="PurchaseQuantity" Width="40" runat="server" Text="<%#:
Item.Quantity %>"></asp:TextBox>
                </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Item Total">
                <ItemTemplate>
                    <%#: String.Format("{0:c}", ((Convert.ToDouble(Item.Quantity)) *
Convert.ToDouble(Item.Product.UnitPrice)))%>
                </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Remove Item">
                <ItemTemplate>
                    <asp:CheckBox id="Remove" runat="server"></asp:CheckBox>
                </ItemTemplate>
        </asp:TemplateField>
        </Columns>
    </asp:GridView>
    <div>
        <p></p>
        <strong>
            <asp:Label ID="LabelTotalText" runat="server" Text="Order Total: "></asp:Label>
            <asp:Label ID="lblTotal" runat="server" EnableViewState="false"></asp:Label>
        </strong>
    </div>
    <br />
</asp:Content>
```

The *ShoppingCart.aspx* page includes a **GridView** control named `CartList`. This control uses model binding to bind the shopping cart data from the database to the **GridView** control. When you set the `ItemType` property of the **GridView** control, the data-binding expression `Item` is available in the markup of the control and the control becomes strongly typed. As mentioned earlier in this tutorial series, you can select details of the `Item` object using IntelliSense. To configure a data control to use model binding to select data, you set the `SelectMethod` property of the control. In the markup above, you set the `SelectMethod` to use the GetShoppingCartItems method which returns a list of `CartItem` objects. The **GridView** data control calls the method at the appropriate time in the page life cycle and automatically binds the returned data. The `GetShoppingCartItems` method must still be added.

Retrieving the Shopping Cart Items

Next, you add code to the *ShoppingCart.aspx.cs* code-behind to retrieve and populate the Shopping Cart UI.

1. In **Solution Explorer**, right-click the *ShoppingCart.aspx* page and then click **View Code**. The *ShoppingCart.aspx.cs* code-behind file is opened in the editor.

2. Replace the existing code with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Models;
using WingtipToys.Logic;

namespace WingtipToys
{
  public partial class ShoppingCart : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {

    }

    public List<CartItem> GetShoppingCartItems()
    {
      ShoppingCartActions actions = new ShoppingCartActions();
      return actions.GetCartItems();
    }
  }
}
```

As mentioned above, the `GridView` data control calls the `GetShoppingCartItems` method at the appropriate time in the page life cycle and automatically binds the returned data. The `GetShoppingCartItems` method creates an instance of the `ShoppingCartActions` object. Then, the code uses that instance to return the items in the cart by calling the `GetCartItems` method.

Adding Products to the Shopping Cart

When either the *ProductList.aspx* or the *ProductDetails.aspx* page is displayed, the user will be able to add the product to the shopping cart using a link. When they click the link, the application navigates to the processing page named *AddToCart.aspx*. The *AddToCart.aspx* page will call the `AddToCart` method in the `ShoppingCart` class that you added earlier in this tutorial.

Now, you'll add an **Add to Cart** link to both the *ProductList.aspx* page and the *ProductDetails.aspx* page. This link will include the product `ID` that is retrieved from the database.

1. In **Solution Explorer**, find and open the page named *ProductList.aspx*.
2. Add the markup highlighted in yellow to the *ProductList.aspx* page so that the entire page appears as follows:

```
<%@ Page Title="Products" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
         CodeBehind="ProductList.aspx.cs" Inherits="WingtipToys.ProductList" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <section>
        <div>
            <hgroup>
                <h2><%: Page.Title %></h2>
            </hgroup>

            <asp:ListView ID="productList" runat="server"
                DataKeyNames="ProductID" GroupItemCount="4"
                ItemType="WingtipToys.Models.Product" SelectMethod="GetProducts">
                <EmptyDataTemplate>
                    <table runat="server">
                        <tr>
                            <td>No data was returned.</td>
                        </tr>
```

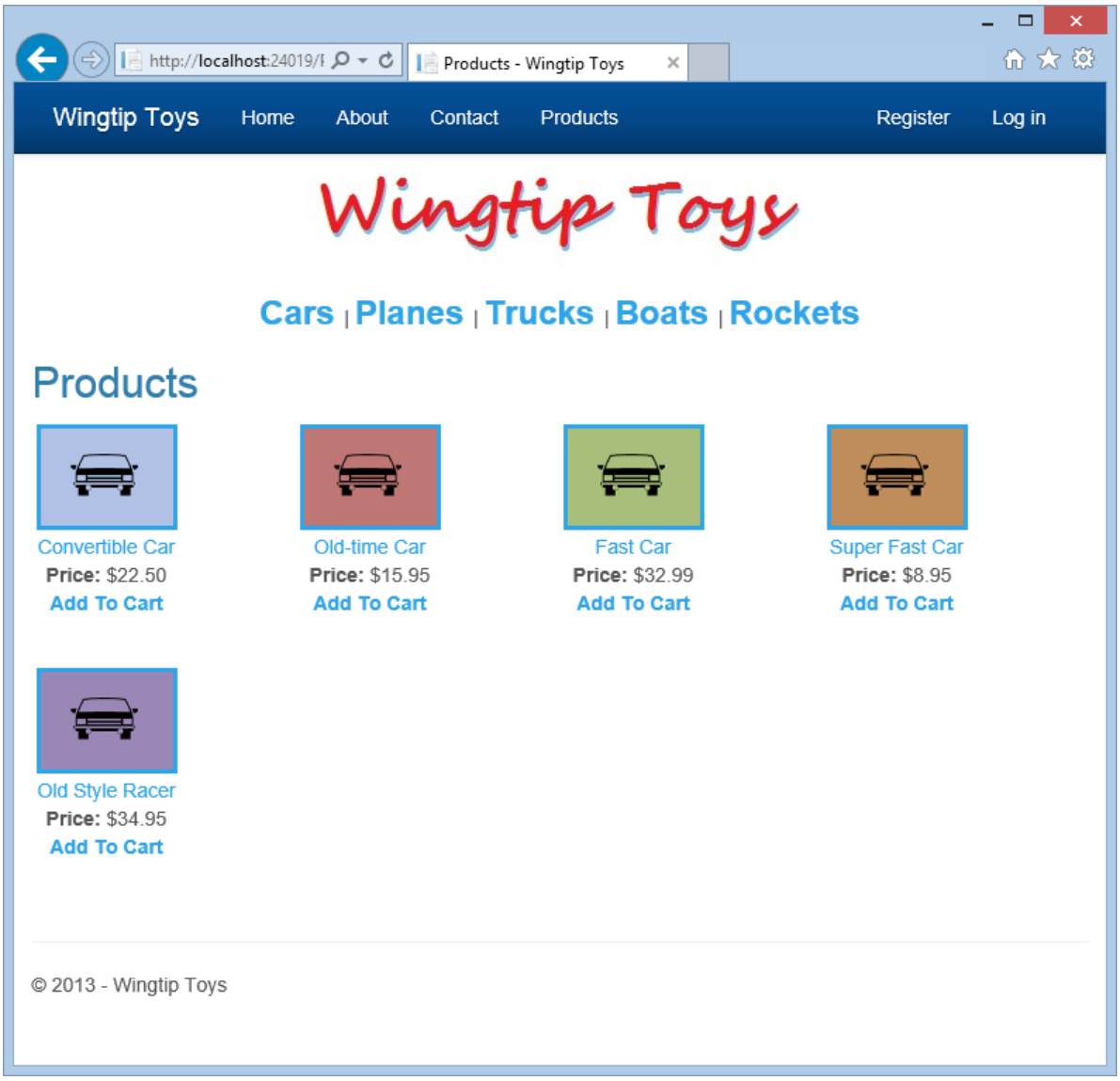
```
                    </table>
                </EmptyDataTemplate>
                <EmptyItemTemplate>
                    <td runat="server" />
                </EmptyItemTemplate>
                <GroupTemplate>
                    <tr id="itemPlaceholderContainer" runat="server">
                        <td id="itemPlaceholder" runat="server"></td>
                    </tr>
                </GroupTemplate>
                <ItemTemplate>
                    <td runat="server">
                        <table>
                            <tr>
                                <td>
                                    <a href="ProductDetails.aspx?productID=<%#:Item.ProductID%>">
                                        <img src="/Catalog/Images/Thumbs/<%#:Item.ImagePath%>"
                                            width="100" height="75" style="border: solid" /></a>
                                </td>
                            </tr>
                            <tr>
                                <td>
                                    <a href="ProductDetails.aspx?productID=<%#:Item.ProductID%>">
                                        <span>
                                            <%#:Item.ProductName%>
                                        </span>
                                    </a>
                                    <br />
                                    <span>
                                        <b>Price: </b><%#:String.Format("{0:c}", Item.UnitPrice)%>
                                    </span>
                                    <br />
                                    <a href="/AddToCart.aspx?productID=<%#:Item.ProductID %>">
                                        <span class="ProductListItem">
                                            <b>Add To Cart<b>
                                        </span>
                                    </a>
                                </td>
                            </tr>
                            <tr>
                                <td> </td>
                            </tr>
                        </table>
                        </p>
                    </td>
                </ItemTemplate>
                <LayoutTemplate>
                    <table runat="server" style="width:100%;">
                        <tbody>
                            <tr runat="server">
                                <td runat="server">
                                    <table id="groupPlaceholderContainer" runat="server"
style="width:100%">
                                        <tr id="groupPlaceholder" runat="server"></tr>
                                    </table>
                                </td>
                            </tr>
                            <tr runat="server">
                                <td runat="server">></td>
                            </tr>
                            <tr></tr>
                        </tbody>
                    </table>
                </LayoutTemplate>
            </asp:ListView>
        </div>
    </section>
</asp:Content>
```

Testing the Shopping Cart

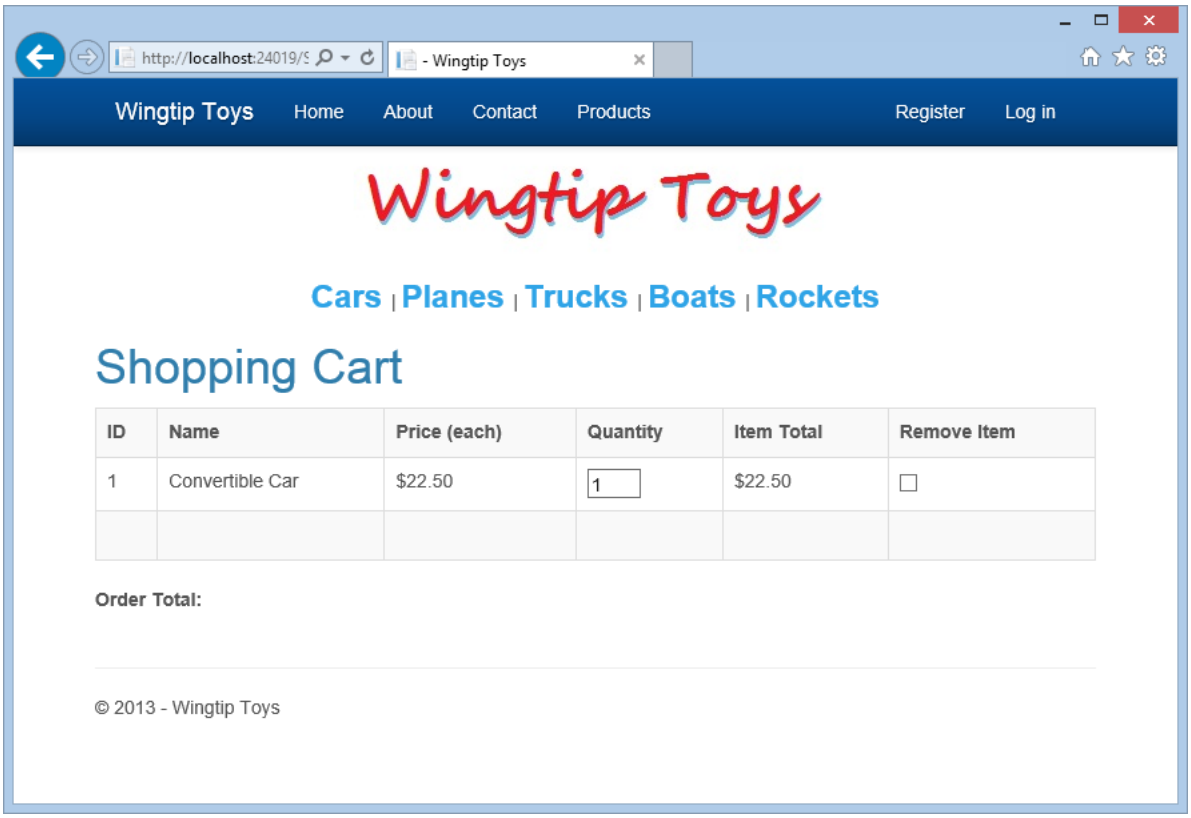
Run the application to see how you add products to the shopping cart.

1. Press **F5** to run the application.
   After the project recreates the database, the browser will open and show the *Default.aspx* page.

2. Select **Cars** from the category navigation menu.
   The *ProductList.aspx* page is displayed showing only products included in the "Cars" category.



3. Click the **Add to Cart** link next to the first product listed (the convertible car).
   The *ShoppingCart.aspx* page is displayed, showing the selection in your shopping cart.



4. View additional products by selecting **Planes** from the category navigation menu.

5. Click the **Add to Cart** link next to the first product listed.
   The *ShoppingCart.aspx* page is displayed with the additional item.

6. Close the browser.

Calculating and Displaying the Order Total

In addition to adding products to the shopping cart, you will add a `GetTotal` method to the `ShoppingCart` class and display the total order amount in the shopping cart page.

1. In **Solution Explorer**, open the *ShoppingCartActions.cs* file in the *Logic* folder.

2. Add the following `GetTotal` method highlighted in yellow to the `ShoppingCart` class, so that the class appears as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using WingtipToys.Models;

namespace WingtipToys.Logic
{
  public class ShoppingCartActions : IDisposable
  {
    public string ShoppingCartId { get; set; }

    private ProductContext _db = new ProductContext();

    public const string CartSessionKey = "CartId";

    public void AddToCart(int id)
    {
      // Retrieve the product from the database.
      ShoppingCartId = GetCartId();

      var cartItem = _db.ShoppingCartItems.SingleOrDefault(
```

```
        c => c.CartId == ShoppingCartId
        && c.ProductId == id);
    if (cartItem == null)
    {
      // Create a new cart item if no cart item exists.
      cartItem = new CartItem
      {
        ItemId = Guid.NewGuid().ToString(),
        ProductId = id,
        CartId = ShoppingCartId,
        Product = _db.Products.SingleOrDefault(
         p => p.ProductID == id),
        Quantity = 1,
        DateCreated = DateTime.Now
      };

      _db.ShoppingCartItems.Add(cartItem);
    }
    else
    {
      // If the item does exist in the cart,
      // then add one to the quantity.
      cartItem.Quantity++;
    }
    _db.SaveChanges();
  }

  public void Dispose()
  {
    if (_db != null)
    {
      _db.Dispose();
      _db = null;
    }
  }

  public string GetCartId()
  {
    if (HttpContext.Current.Session[CartSessionKey] == null)
    {
      if (!string.IsNullOrWhiteSpace(HttpContext.Current.User.Identity.Name))
      {
        HttpContext.Current.Session[CartSessionKey] = HttpContext.Current.User.Identity.Name;
      }
      else
      {
        // Generate a new random GUID using System.Guid class.
        Guid tempCartId = Guid.NewGuid();
        HttpContext.Current.Session[CartSessionKey] = tempCartId.ToString();
      }
    }
    return HttpContext.Current.Session[CartSessionKey].ToString();
  }

  public List<CartItem> GetCartItems()
  {
    ShoppingCartId = GetCartId();

    return _db.ShoppingCartItems.Where(
        c => c.CartId == ShoppingCartId).ToList();
  }

  public decimal GetTotal()
  {
    ShoppingCartId = GetCartId();
    // Multiply product price by quantity of that product to get
    // the current price for each of those products in the cart.
    // Sum all product price totals to get the cart total.
    decimal? total = decimal.Zero;
```

```
        total = (decimal?)(from cartItems in _db.ShoppingCartItems
                           where cartItems.CartId == ShoppingCartId
                           select (int?)cartItems.Quantity *
                           cartItems.Product.UnitPrice).Sum();
        return total ?? decimal.Zero;
      }
    }
  }
```

First, the `GetTotal` method gets the ID of the shopping cart for the user. Then the method gets the cart total by multiplying the product price by the product quantity for each product listed in the cart.

> **NOTE**
>
> The above code uses the nullable type "`int?`". Nullable types can represent all the values of an underlying type, and also as a null value. For more information see, Using Nullable Types.

Modify the Shopping Cart Display

Next you'll modify the code for the *ShoppingCart.aspx* page to call the `GetTotal` method and display that total on the *ShoppingCart.aspx* page when the page loads.

1. In **Solution Explorer**, right-click the *ShoppingCart.aspx* page and select **View Code**.
2. In the *ShoppingCart.aspx.cs* file, update the `Page_Load` handler by adding the following code highlighted in yellow:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Models;
using WingtipToys.Logic;

namespace WingtipToys
{
  public partial class ShoppingCart : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      using (ShoppingCartActions usersShoppingCart = new ShoppingCartActions())
      {
        decimal cartTotal = 0;
        cartTotal = usersShoppingCart.GetTotal();
        if (cartTotal > 0)
        {
          // Display Total.
          lblTotal.Text = String.Format("{0:c}", cartTotal);
        }
        else
        {
          LabelTotalText.Text = "";
          lblTotal.Text = "";
          ShoppingCartTitle.InnerText = "Shopping Cart is Empty";
        }
      }
    }

    public List<CartItem> GetShoppingCartItems()
    {
      ShoppingCartActions actions = new ShoppingCartActions();
      return actions.GetCartItems();
    }
  }
}
```

When the *ShoppingCart.aspx* page loads, it loads the shopping cart object and then retrieves the shopping cart total by calling the `GetTotal` method of the `ShoppingCart` class. If the shopping cart is empty, a message to that effect is displayed.

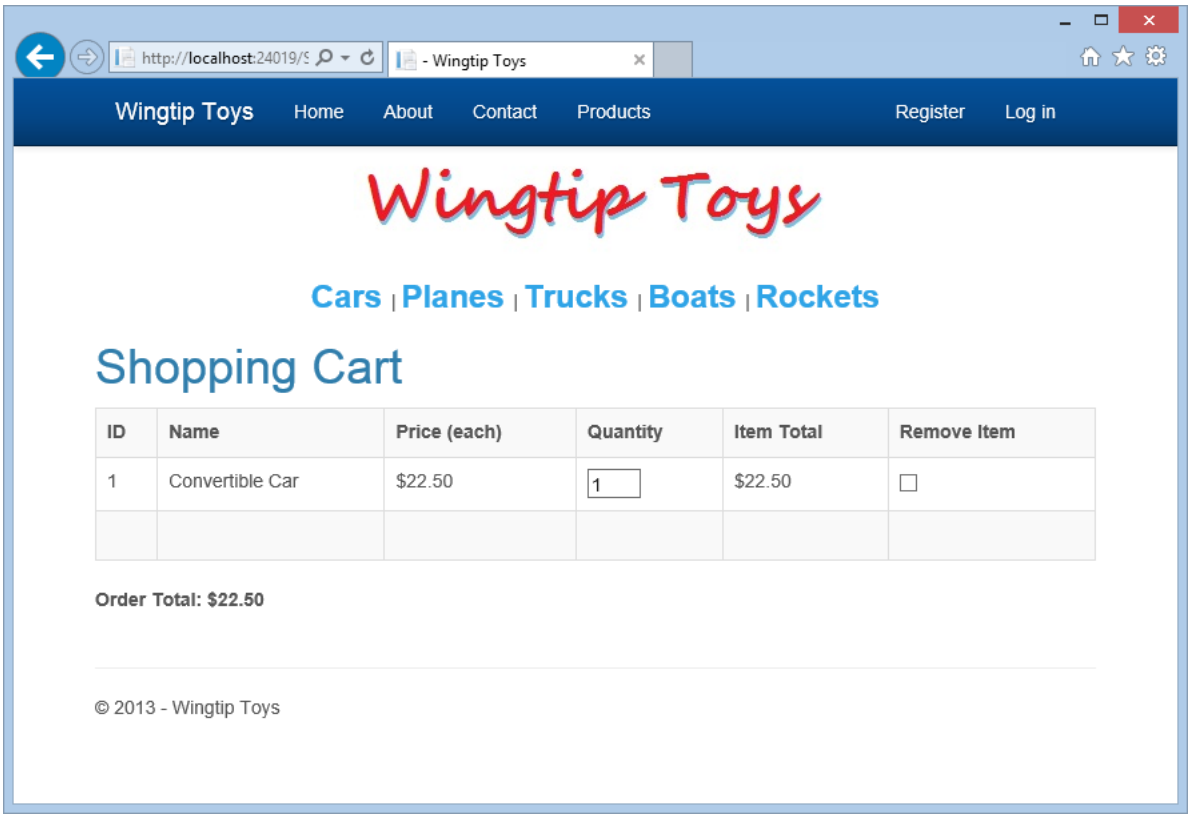
Testing the Shopping Cart Total

Run the application now to see how you can not only add a product to the shopping cart, but you can see the shopping cart total.

1. Press **F5** to run the application.
   The browser will open and show the *Default.aspx* page.

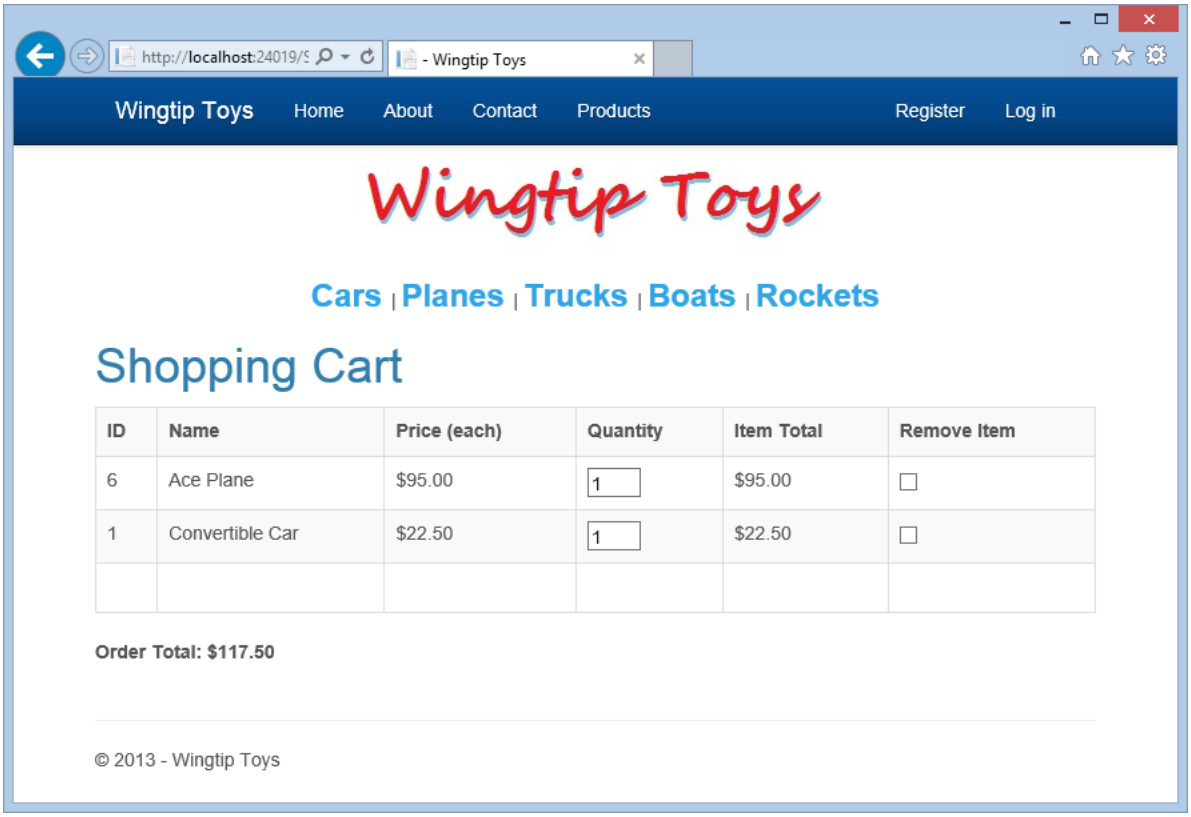
2. Select **Cars** from the category navigation menu.

3. Click the **Add To Cart** link next to the first product.
   The *ShoppingCart.aspx* page is displayed with the order total.



4. Add some other products (for example, a plane) to the cart.

5. The *ShoppingCart.aspx* page is displayed with an updated total for all the products you've added.



6. Stop the running app by closing the browser window.

Adding Update and Checkout Buttons to the Shopping Cart

To allow the users to modify the shopping cart, you'll add an **Update** button and a **Checkout** button to the shopping cart page. The **Checkout** button is not used until later in this tutorial series.

1. In **Solution Explorer**, open the *ShoppingCart.aspx* page in the root of the web application project.

2. To add the **Update** button and the **Checkout** button to the *ShoppingCart.aspx* page, add the markup highlighted in yellow to the existing markup, as shown in the following code:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="ShoppingCart.aspx.cs" Inherits="WingtipToys.ShoppingCart" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <div id="ShoppingCartTitle" runat="server" class="ContentHead"><h1>Shopping Cart</h1></div>
    <asp:GridView ID="CartList" runat="server" AutoGenerateColumns="False" ShowFooter="True"
GridLines="Vertical" CellPadding="4"
        ItemType="WingtipToys.Models.CartItem" SelectMethod="GetShoppingCartItems"
        CssClass="table table-striped table-bordered" >
        <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="ID" SortExpression="ProductID" />
        <asp:BoundField DataField="Product.ProductName" HeaderText="Name" />
        <asp:BoundField DataField="Product.UnitPrice" HeaderText="Price (each)" DataFormatString="
{0:c}"/>
        <asp:TemplateField   HeaderText="Quantity">
                <ItemTemplate>
                    <asp:TextBox ID="PurchaseQuantity" Width="40" runat="server" Text="<%#:
Item.Quantity %>"></asp:TextBox>
                </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Item Total">
                <ItemTemplate>
                    <%#: String.Format("{0:c}", ((Convert.ToDouble(Item.Quantity)) *
Convert.ToDouble(Item.Product.UnitPrice)))%>
                </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Remove Item">
                <ItemTemplate>
                    <asp:CheckBox id="Remove" runat="server"></asp:CheckBox>
                </ItemTemplate>
        </asp:TemplateField>
        </Columns>
    </asp:GridView>
    <div>
        <p></p>
        <strong>
            <asp:Label ID="LabelTotalText" runat="server" Text="Order Total: "></asp:Label>
            <asp:Label ID="lblTotal" runat="server" EnableViewState="false"></asp:Label>
        </strong>
    </div>
  <br />
    <table>
    <tr>
      <td>
        <asp:Button ID="UpdateBtn" runat="server" Text="Update" OnClick="UpdateBtn_Click" />
      </td>
      <td>
        <!--Checkout Placeholder -->
      </td>
    </tr>
    </table>
</asp:Content>
```

When the user clicks the **Update** button, the `UpdateBtn_Click` event handler will be called. This event handler will call the code that you'll add in the next step.

Next, you can update the code contained in the *ShoppingCart.aspx.cs* file to loop through the cart items and call the `RemoveItem` and `UpdateItem` methods.

1. In **Solution Explorer**, open the *ShoppingCart.aspx.cs* file in the root of the web application project.

2. Add the following code sections highlighted in yellow to the *ShoppingCart.aspx.cs* file:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Models;
using WingtipToys.Logic;
using System.Collections.Specialized;
using System.Collections;
using System.Web.ModelBinding;

namespace WingtipToys
{
  public partial class ShoppingCart : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      using (ShoppingCartActions usersShoppingCart = new ShoppingCartActions())
      {
        decimal cartTotal = 0;
        cartTotal = usersShoppingCart.GetTotal();
        if (cartTotal > 0)
        {
          // Display Total.
          lblTotal.Text = String.Format("{0:c}", cartTotal);
        }
        else
        {
          LabelTotalText.Text = "";
          lblTotal.Text = "";
          ShoppingCartTitle.InnerText = "Shopping Cart is Empty";
          UpdateBtn.Visible = false;
        }
      }
    }

    public List<CartItem> GetShoppingCartItems()
    {
      ShoppingCartActions actions = new ShoppingCartActions();
      return actions.GetCartItems();
    }

    public List<CartItem> UpdateCartItems()
    {
      using (ShoppingCartActions usersShoppingCart = new ShoppingCartActions())
      {
        String cartId = usersShoppingCart.GetCartId();

        ShoppingCartActions.ShoppingCartUpdates[] cartUpdates = new
ShoppingCartActions.ShoppingCartUpdates[CartList.Rows.Count];
        for (int i = 0; i < CartList.Rows.Count; i++)
        {
          IOrderedDictionary rowValues = new OrderedDictionary();
          rowValues = GetValues(CartList.Rows[i]);
          cartUpdates[i].ProductId = Convert.ToInt32(rowValues["ProductID"]);

          CheckBox cbRemove = new CheckBox();
          cbRemove = (CheckBox)CartList.Rows[i].FindControl("Remove");
          cartUpdates[i].RemoveItem = cbRemove.Checked;

          TextBox quantityTextBox = new TextBox();
          quantityTextBox = (TextBox)CartList.Rows[i].FindControl("PurchaseQuantity");
          cartUpdates[i].PurchaseQuantity = Convert.ToInt16(quantityTextBox.Text.ToString());
        }
        usersShoppingCart.UpdateShoppingCartDatabase(cartId, cartUpdates);
        CartList.DataBind();
        lblTotal.Text = String.Format("{0:c}", usersShoppingCart.GetTotal());
```

```
        return usersShoppingCart.GetCartItems();
      }
    }

    public static IOrderedDictionary GetValues(GridViewRow row)
    {
      IOrderedDictionary values = new OrderedDictionary();
      foreach (DataControlFieldCell cell in row.Cells)
      {
        if (cell.Visible)
        {
          // Extract values from the cell.
          cell.ContainingField.ExtractValuesFromCell(values, cell, row.RowState, true);
        }
      }
      return values;
    }

    protected void UpdateBtn_Click(object sender, EventArgs e)
    {
      UpdateCartItems();
    }
  }
}
```

When the user clicks the **Update** button on the *ShoppingCart.aspx* page, the UpdateCartItems method is called. The UpdateCartItems method gets the updated values for each item in the shopping cart. Then, the UpdateCartItems method calls the `UpdateShoppingCartDatabase` method (added and explained in the next step) to either add or remove items from the shopping cart. Once the database has been updated to reflect the updates to the shopping cart, the **GridView** control is updated on the shopping cart page by calling the `DataBind` method for the **GridView**. Also, the total order amount on the shopping cart page is updated to reflect the updated list of items.

Updating and Removing Shopping Cart Items

On the *ShoppingCart.aspx* page, you can see controls have been added for updating the quantity of an item and removing an item. Now, add the code that will make these controls work.

1. In **Solution Explorer**, open the *ShoppingCartActions.cs* file in the *Logic* folder.
2. Add the following code highlighted in yellow to the *ShoppingCartActions.cs* class file:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using WingtipToys.Models;

namespace WingtipToys.Logic
{
  public class ShoppingCartActions : IDisposable
  {
    public string ShoppingCartId { get; set; }

    private ProductContext _db = new ProductContext();

    public const string CartSessionKey = "CartId";

    public void AddToCart(int id)
    {
      // Retrieve the product from the database.
      ShoppingCartId = GetCartId();

      var cartItem = _db.ShoppingCartItems.SingleOrDefault(
          c => c.CartId == ShoppingCartId
```

```
            && c.ProductId == id);
        if (cartItem == null)
        {
          // Create a new cart item if no cart item exists.
          cartItem = new CartItem
          {
            ItemId = Guid.NewGuid().ToString(),
            ProductId = id,
            CartId = ShoppingCartId,
            Product = _db.Products.SingleOrDefault(
             p => p.ProductID == id),
            Quantity = 1,
            DateCreated = DateTime.Now
          };

          _db.ShoppingCartItems.Add(cartItem);
        }
        else
        {
          // If the item does exist in the cart,
          // then add one to the quantity.
          cartItem.Quantity++;
        }
        _db.SaveChanges();
      }

      public void Dispose()
      {
        if (_db != null)
        {
          _db.Dispose();
          _db = null;
        }
      }

      public string GetCartId()
      {
        if (HttpContext.Current.Session[CartSessionKey] == null)
        {
          if (!string.IsNullOrWhiteSpace(HttpContext.Current.User.Identity.Name))
          {
            HttpContext.Current.Session[CartSessionKey] = HttpContext.Current.User.Identity.Name;
          }
          else
          {
            // Generate a new random GUID using System.Guid class.
            Guid tempCartId = Guid.NewGuid();
            HttpContext.Current.Session[CartSessionKey] = tempCartId.ToString();
          }
        }
        return HttpContext.Current.Session[CartSessionKey].ToString();
      }

      public List<CartItem> GetCartItems()
      {
        ShoppingCartId = GetCartId();

        return _db.ShoppingCartItems.Where(
            c => c.CartId == ShoppingCartId).ToList();
      }

      public decimal GetTotal()
      {
        ShoppingCartId = GetCartId();
        // Multiply product price by quantity of that product to get
        // the current price for each of those products in the cart.
        // Sum all product price totals to get the cart total.
        decimal? total = decimal.Zero;
        total = (decimal?)(from cartItems in _db.ShoppingCartItems
```

```
                        where cartItems.CartId == ShoppingCartId
                        select (int?)cartItems.Quantity *
                        cartItems.Product.UnitPrice).Sum();
      return total ?? decimal.Zero;
    }

    public ShoppingCartActions GetCart(HttpContext context)
    {
      using (var cart = new ShoppingCartActions())
      {
        cart.ShoppingCartId = cart.GetCartId();
        return cart;
      }
    }

    public void UpdateShoppingCartDatabase(String cartId, ShoppingCartUpdates[] CartItemUpdates)
    {
      using (var db = new WingtipToys.Models.ProductContext())
      {
        try
        {
          int CartItemCount = CartItemUpdates.Count();
          List<CartItem> myCart = GetCartItems();
          foreach (var cartItem in myCart)
          {
            // Iterate through all rows within shopping cart list
            for (int i = 0; i < CartItemCount; i++)
            {
              if (cartItem.Product.ProductID == CartItemUpdates[i].ProductId)
              {
                if (CartItemUpdates[i].PurchaseQuantity < 1 || CartItemUpdates[i].RemoveItem == true)
                {
                  RemoveItem(cartId, cartItem.ProductId);
                }
                else
                {
                  UpdateItem(cartId, cartItem.ProductId, CartItemUpdates[i].PurchaseQuantity);
                }
              }
            }
          }
        }
        catch (Exception exp)
        {
          throw new Exception("ERROR: Unable to Update Cart Database - " + exp.Message.ToString(),
exp);
        }
      }
    }

    public void RemoveItem(string removeCartID, int removeProductID)
    {
      using (var _db = new WingtipToys.Models.ProductContext())
      {
        try
        {
          var myItem = (from c in _db.ShoppingCartItems where c.CartId == removeCartID &&
c.Product.ProductID == removeProductID select c).FirstOrDefault();
          if (myItem != null)
          {
            // Remove Item.
            _db.ShoppingCartItems.Remove(myItem);
            _db.SaveChanges();
          }
        }
        catch (Exception exp)
        {
          throw new Exception("ERROR: Unable to Remove Cart Item - " + exp.Message.ToString(), exp);
        }
```

```
      }
    }

    public void UpdateItem(string updateCartID, int updateProductID, int quantity)
    {
      using (var _db = new WingtipToys.Models.ProductContext())
      {
        try
        {
          var myItem = (from c in _db.ShoppingCartItems where c.CartId == updateCartID &&
c.Product.ProductID == updateProductID select c).FirstOrDefault();
          if (myItem != null)
          {
            myItem.Quantity = quantity;
            _db.SaveChanges();
          }
        }
        catch (Exception exp)
        {
          throw new Exception("ERROR: Unable to Update Cart Item - " + exp.Message.ToString(), exp);
        }
      }
    }

    public void EmptyCart()
    {
      ShoppingCartId = GetCartId();
      var cartItems = _db.ShoppingCartItems.Where(
          c => c.CartId == ShoppingCartId);
      foreach (var cartItem in cartItems)
      {
        _db.ShoppingCartItems.Remove(cartItem);
      }
      // Save changes.
      _db.SaveChanges();
    }

    public int GetCount()
    {
      ShoppingCartId = GetCartId();

      // Get the count of each item in the cart and sum them up
      int? count = (from cartItems in _db.ShoppingCartItems
                    where cartItems.CartId == ShoppingCartId
                    select (int?)cartItems.Quantity).Sum();
      // Return 0 if all entries are null
      return count ?? 0;
    }

    public struct ShoppingCartUpdates
    {
      public int ProductId;
      public int PurchaseQuantity;
      public bool RemoveItem;
    }
  }
}
```

The `UpdateShoppingCartDatabase` method, called from the `UpdateCartItems` method on the *ShoppingCart.aspx.cs* page, contains the logic to either update or remove items from the shopping cart. The `UpdateShoppingCartDatabase` method iterates through all the rows within the shopping cart list. If a shopping cart item has been marked to be removed, or the quantity is less than one, the `RemoveItem` method is called. Otherwise, the shopping cart item is checked for updates when the `UpdateItem` method is called. After the shopping cart item has been removed or updated, the database changes are saved.

The `ShoppingCartUpdates` structure is used to hold all the shopping cart items. The `UpdateShoppingCartDatabase` method uses the `ShoppingCartUpdates` structure to determine if any of the items need to be updated or removed.

In the next tutorial, you will use the `EmptyCart` method to clear the shopping cart after purchasing products. But for now, you will use the `GetCount` method that you just added to the *ShoppingCartActions.cs* file to determine how many items are in the shopping cart.

Adding a Shopping Cart Counter

To allow the user to view the total number of items in the shopping cart, you will add a counter to the *Site.Master* page. This counter will also act as a link to the shopping cart.

1. In **Solution Explorer**, open the *Site.Master* page.
2. Modify the markup by adding the shopping cart counter link as shown in yellow to the navigation section so it appears as follows:

```
<ul class="nav navbar-nav">
      <li><a runat="server" href="~/">Home</a></li>
      <li><a runat="server" href="~/About">About</a></li>
      <li><a runat="server" href="~/Contact">Contact</a></li>
      <li><a runat="server" href="~/ProductList">Products</a></li>
      <li><a runat="server" href="~/ShoppingCart" ID="cartCount"> </a></li>
  </ul>
```

3. Next, update the code-behind of the *Site.Master.cs* file by adding the code highlighted in yellow as follows:

```
using System;
using System.Collections.Generic;
using System.Security.Claims;
using System.Security.Principal;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Linq;
using WingtipToys.Models;
using WingtipToys.Logic;

namespace WingtipToys
{
    public partial class SiteMaster : MasterPage
    {
        private const string AntiXsrfTokenKey = "__AntiXsrfToken";
        private const string AntiXsrfUserNameKey = "__AntiXsrfUserName";
        private string _antiXsrfTokenValue;

        protected void Page_Init(object sender, EventArgs e)
        {
            // The code below helps to protect against XSRF attacks
            var requestCookie = Request.Cookies[AntiXsrfTokenKey];
            Guid requestCookieGuidValue;
            if (requestCookie != null && Guid.TryParse(requestCookie.Value, out
requestCookieGuidValue))
            {
                // Use the Anti-XSRF token from the cookie
                _antiXsrfTokenValue = requestCookie.Value;
                Page.ViewStateUserKey = _antiXsrfTokenValue;
            }
            else
            {
                // Generate a new Anti-XSRF token and save to the cookie
                _antiXsrfTokenValue = Guid.NewGuid().ToString("N");
                Page.ViewStateUserKey = _antiXsrfTokenValue;
```

```
                var responseCookie = new HttpCookie(AntiXsrfTokenKey)
                {
                    HttpOnly = true,
                    Value = _antiXsrfTokenValue
                };
                if (FormsAuthentication.RequireSSL && Request.IsSecureConnection)
                {
                    responseCookie.Secure = true;
                }
                Response.Cookies.Set(responseCookie);
            }

            Page.PreLoad += master_Page_PreLoad;
        }

        protected void master_Page_PreLoad(object sender, EventArgs e)
        {
            if (!IsPostBack)
            {
                // Set Anti-XSRF token
                ViewState[AntiXsrfTokenKey] = Page.ViewStateUserKey;
                ViewState[AntiXsrfUserNameKey] = Context.User.Identity.Name ?? String.Empty;
            }
            else
            {
                // Validate the Anti-XSRF token
                if ((string)ViewState[AntiXsrfTokenKey] != _antiXsrfTokenValue
                    || (string)ViewState[AntiXsrfUserNameKey] != (Context.User.Identity.Name ??
String.Empty))
                {
                    throw new InvalidOperationException("Validation of Anti-XSRF token failed.");
                }
            }
        }

        protected void Page_Load(object sender, EventArgs e)
        {

        }

        protected void Page_PreRender(object sender, EventArgs e)
        {
          using (ShoppingCartActions usersShoppingCart = new ShoppingCartActions())
          {
            string cartStr = string.Format("Cart ({0})", usersShoppingCart.GetCount());
            cartCount.InnerText = cartStr;
          }
        }

        public IQueryable<Category> GetCategories()
        {
          var _db = new WingtipToys.Models.ProductContext();
          IQueryable<Category> query = _db.Categories;
          return query;
        }

        protected void Unnamed_LoggingOut(object sender, LoginCancelEventArgs e)
        {
            Context.GetOwinContext().Authentication.SignOut();
        }
    }
}
```

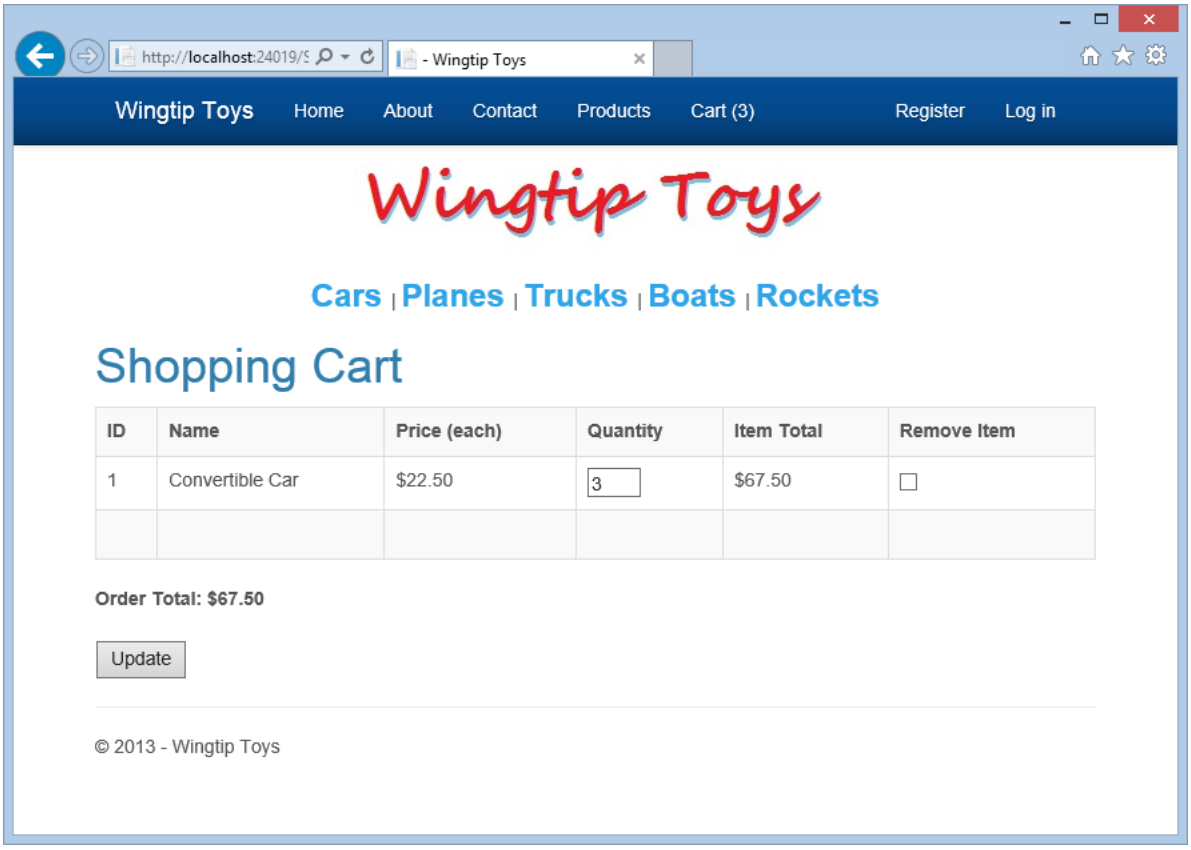
Before the page is rendered as HTML, the `Page_PreRender` event is raised. In the `Page_PreRender` handler, the total count of the shopping cart is determined by calling the `GetCount` method. The returned value is added to the `cartCount` span included in the markup of the *Site.Master* page. The `<span>` tags enables the inner elements to

be properly rendered. When any page of the site is displayed, the shopping cart total will be displayed. The user can also click the shopping cart total to display the shopping cart.

Testing the Completed Shopping Cart

You can run the application now to see how you can add, delete, and update items in the shopping cart. The shopping cart total will reflect the total cost of all items in the shopping cart.

1. Press **F5** to run the application.
   The browser opens and shows the *Default.aspx* page.
2. Select **Cars** from the category navigation menu.
3. Click the **Add To Cart** link next to the first product.
   The *ShoppingCart.aspx* page is displayed with the order total.
4. Select **Planes** from the category navigation menu.
5. Click the **Add To Cart** link next to the first product.
6. Set the quantity of the first item in the shopping cart to 3 and select the **Remove Item** check box of the second item.
7. Click the **Update** button to update the shopping cart page and display the new order total.



Summary

In this tutorial, you have created a shopping cart for the Wingtip Toys Web Forms sample application. During this tutorial you have used Entity Framework Code First, data annotations, strongly typed data controls, and model binding.

The shopping cart supports adding, deleting, and updating items that the user has selected for purchase. In addition to implementing the shopping cart functionality, you have learned how to display shopping cart items in a **GridView** control and calculate the order total.

In order to understand how the described functionality works in a real business application, you can view the example of nopCommerce - ASP.NET based open source eCommerce shopping cart. Originally, it was built on Web Forms and over the years it moved to MVC and now to ASP.NET Core.

Addition Information

ASP.NET Session State Overview

PREVIOUS NEXT

Checkout and Payment with PayPal

2/20/2020 • 42 minutes to read • Edit Online

by Erik Reitan

Download Wingtip Toys Sample Project (C#) or Download E-book (PDF)

> This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

This tutorial describes how to modify the Wingtip Toys sample application to include user authorization, registration, and payment using PayPal. Only users who are logged in will have authorization to purchase products. The ASP.NET 4.5 Web Forms project template's built-in user registration functionality already includes much of what you need. You will add PayPal Express Checkout functionality. In this tutorial you be using the PayPal developer testing environment, so no actual funds will be transferred. At the end of the tutorial, you will test the application by selecting products to add to the shopping cart, clicking the checkout button, and transferring data to the PayPal testing web site. On the PayPal testing web site, you will confirm your shipping and payment information and then return to the local Wingtip Toys sample application to confirm and complete the purchase.

There are several experienced third-party payment processors that specialize in online shopping that address scalability and security. ASP.NET developers should consider the advantages of utilizing a third party payment solution before implementing a shopping and purchasing solution.

> **NOTE**
> The Wingtip Toys sample application was designed to shown specific ASP.NET concepts and features available to ASP.NET web developers. This sample application was not optimized for all possible circumstances in regard to scalability and security.

What you'll learn:

- How to restrict access to specific pages in a folder.
- How to create a known shopping cart from an anonymous shopping cart.
- How to enable SSL for the project.
- How to add an OAuth provider to the project.
- How to use PayPal to purchase products using the PayPal testing environment.
- How to display details from PayPal in a **DetailsView** control.
- How to update the database of the Wingtip Toys application with details obtained from PayPal.

Adding Order Tracking

In this tutorial, you'll create two new classes to track data from the order a user has created. The classes will track data regarding shipping information, purchase total, and payment confirmation.

Add the Order and OrderDetail Model Classes

Earlier in this tutorial series, you defined the schema for categories, products, and shopping cart items by creating the `Category`, `Product`, and `CartItem` classes in the *Models* folder. Now you will add two new classes to define the schema for the product order and the details of the order.

1. In the **Models** folder, add a new class named *Order.cs*.
   The new class file is displayed in the editor.

2. Replace the default code with the following:

```
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;
using System.ComponentModel;

namespace WingtipToys.Models
{
  public class Order
  {
    public int OrderId { get; set; }

    public System.DateTime OrderDate { get; set; }

    public string Username { get; set; }

    [Required(ErrorMessage = "First Name is required")]
    [DisplayName("First Name")]
    [StringLength(160)]
    public string FirstName { get; set; }

    [Required(ErrorMessage = "Last Name is required")]
    [DisplayName("Last Name")]
    [StringLength(160)]
    public string LastName { get; set; }

    [Required(ErrorMessage = "Address is required")]
    [StringLength(70)]
    public string Address { get; set; }

    [Required(ErrorMessage = "City is required")]
    [StringLength(40)]
    public string City { get; set; }

    [Required(ErrorMessage = "State is required")]
    [StringLength(40)]
    public string State { get; set; }

    [Required(ErrorMessage = "Postal Code is required")]
    [DisplayName("Postal Code")]
    [StringLength(10)]
    public string PostalCode { get; set; }

    [Required(ErrorMessage = "Country is required")]
    [StringLength(40)]
    public string Country { get; set; }

    [StringLength(24)]
    public string Phone { get; set; }

    [Required(ErrorMessage = "Email Address is required")]
    [DisplayName("Email Address")]
    [RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
        ErrorMessage = "Email is is not valid.")]
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }

    [ScaffoldColumn(false)]
    public decimal Total { get; set; }

    [ScaffoldColumn(false)]
    public string PaymentTransactionId { get; set; }

    [ScaffoldColumn(false)]
```

```
    public bool HasBeenShipped { get; set; }

    public List<OrderDetail> OrderDetails { get; set; }
  }
}
```

3. Add an *OrderDetail.cs* class to the *Models* folder.

4. Replace the default code with the following code:

```
using System.ComponentModel.DataAnnotations;

namespace WingtipToys.Models
{
    public class OrderDetail
    {
        public int OrderDetailId { get; set; }

        public int OrderId { get; set; }

        public string Username { get; set; }

        public int ProductId { get; set; }

        public int Quantity { get; set; }

        public double? UnitPrice { get; set; }

    }
}
```

The `Order` and `OrderDetail` classes contain the schema to define the order information used for purchasing and shipping.

In addition, you will need to update the database context class that manages the entity classes and that provides data access to the database. To do this, you will add the newly created Order and `OrderDetail` model classes to `ProductContext` class.

1. In **Solution Explorer**, find and open the *ProductContext.cs* file.

2. Add the highlighted code to the *ProductContext.cs* file as shown below:

```
using System.Data.Entity;

namespace WingtipToys.Models
{
  public class ProductContext : DbContext
  {
    public ProductContext()
      : base("WingtipToys")
    {
    }
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<CartItem> ShoppingCartItems { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<OrderDetail> OrderDetails { get; set; }
  }
}
```

As mentioned previously in this tutorial series, the code in the *ProductContext.cs* file adds the `System.Data.Entity` namespace so that you have access to all the core functionality of the Entity Framework. This functionality

includes the capability to query, insert, update, and delete data by working with strongly typed objects. The above code in the `ProductContext` class adds Entity Framework access to the newly added `Order` and `OrderDetail` classes.

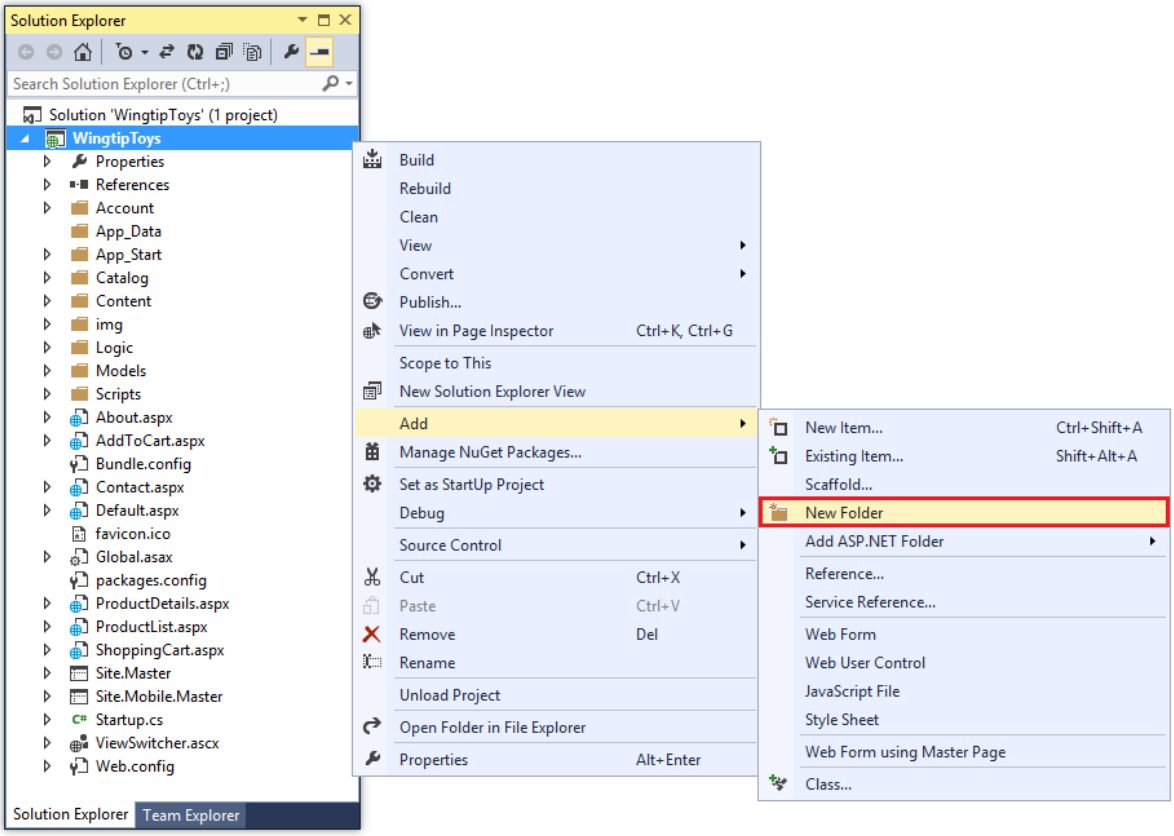
Adding Checkout Access

The Wingtip Toys sample application allows anonymous users to review and add products to a shopping cart. However, when anonymous users choose to purchase the products they added to the shopping cart, they must log on to the site. Once they have logged on, they can access the restricted pages of the Web application that handle the checkout and purchase process. These restricted pages are contained in the *Checkout* folder of the application.

Add a Checkout Folder and Pages

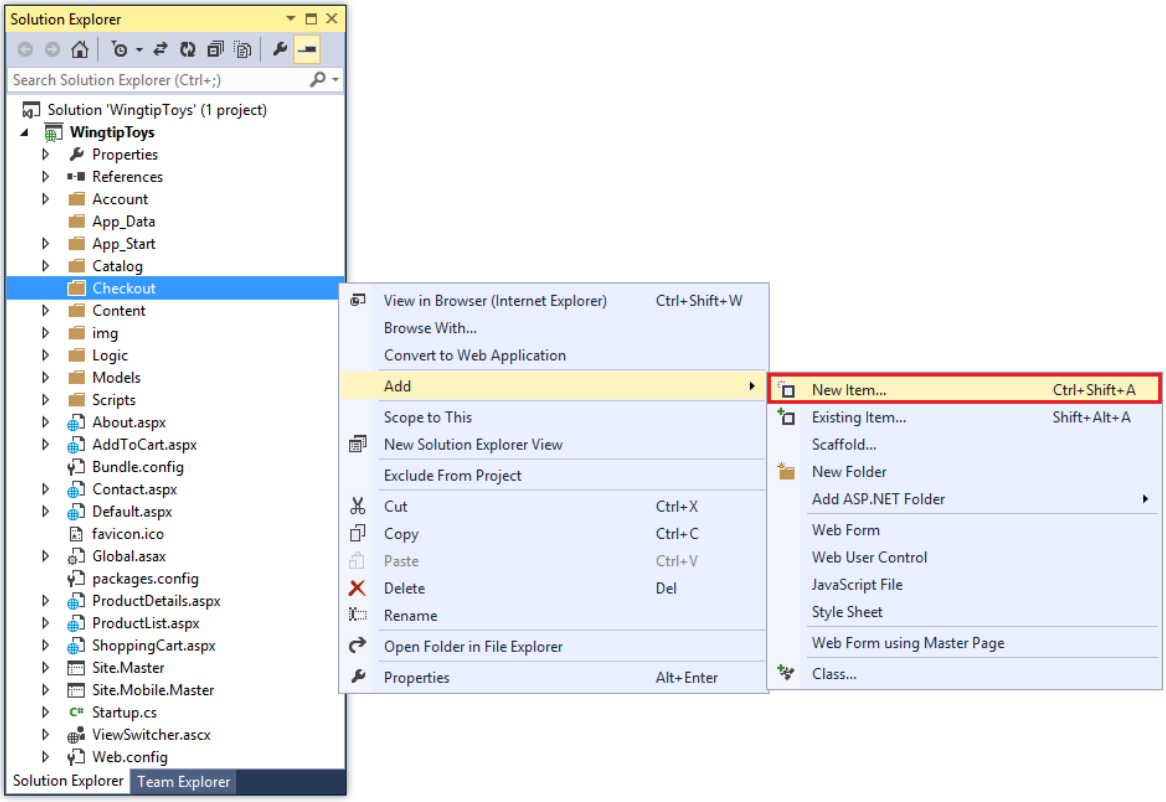
You will now create the *Checkout* folder and the pages in it that the customer will see during the checkout process. You will update these pages later in this tutorial.

1. Right-click the project name (**Wingtip Toys**) in **Solution Explorer** and select **Add a New Folder**.



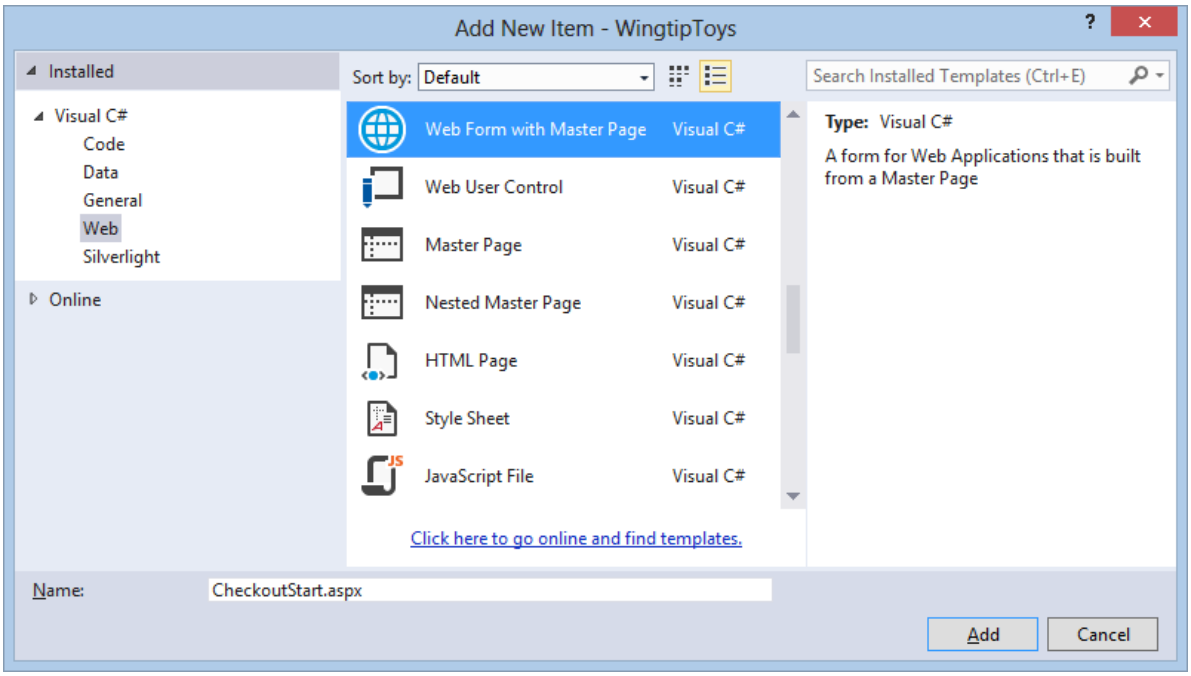
2. Name the new folder *Checkout*.

3. Right-click the *Checkout* folder and then select **Add->New Item**.



4. The **Add New Item** dialog box is displayed.

5. Select the **Visual C#** -> **Web** templates group on the left. Then, from the middle pane, select **Web Form with Master Page**and name it *CheckoutStart.aspx*.



6. As before, select the *Site.Master* file as the master page.

7. Add the following additional pages to the *Checkout* folder using the same steps above:

   - CheckoutReview.aspx
   - CheckoutComplete.aspx
   - CheckoutCancel.aspx
   - CheckoutError.aspx

Add a Web.config File

By adding a new *Web.config* file to the *Checkout* folder, you will be able to restrict access to all the pages contained in the folder.

1. Right-click the *Checkout* folder and select **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.

2. Select the **Visual C#** -> **Web** templates group on the left. Then, from the middle pane, select **Web Configuration File**, accept the default name of *Web.config*, and then select **Add**.

3. Replace the existing XML content in the *Web.config* file with the following:

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <authorization>
      <deny users="?"/>
    </authorization>
  </system.web>
</configuration>
```

4. Save the *Web.config* file.

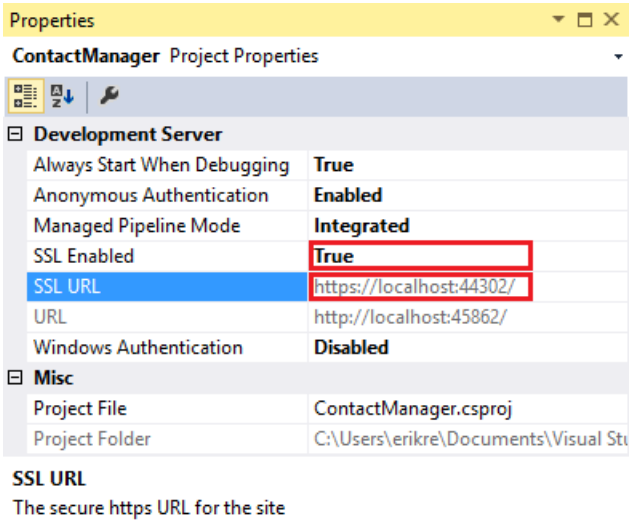
The *Web.config* file specifies that all unknown users of the Web application must be denied access to the pages contained in the *Checkout* folder. However, if the user has registered an account and is logged on, they will be a known user and will have access to the pages in the *Checkout* folder.

It's important to note that ASP.NET configuration follows a hierarchy, where each *Web.config* file applies configuration settings to the folder that it is in and to all of the child directories below it.

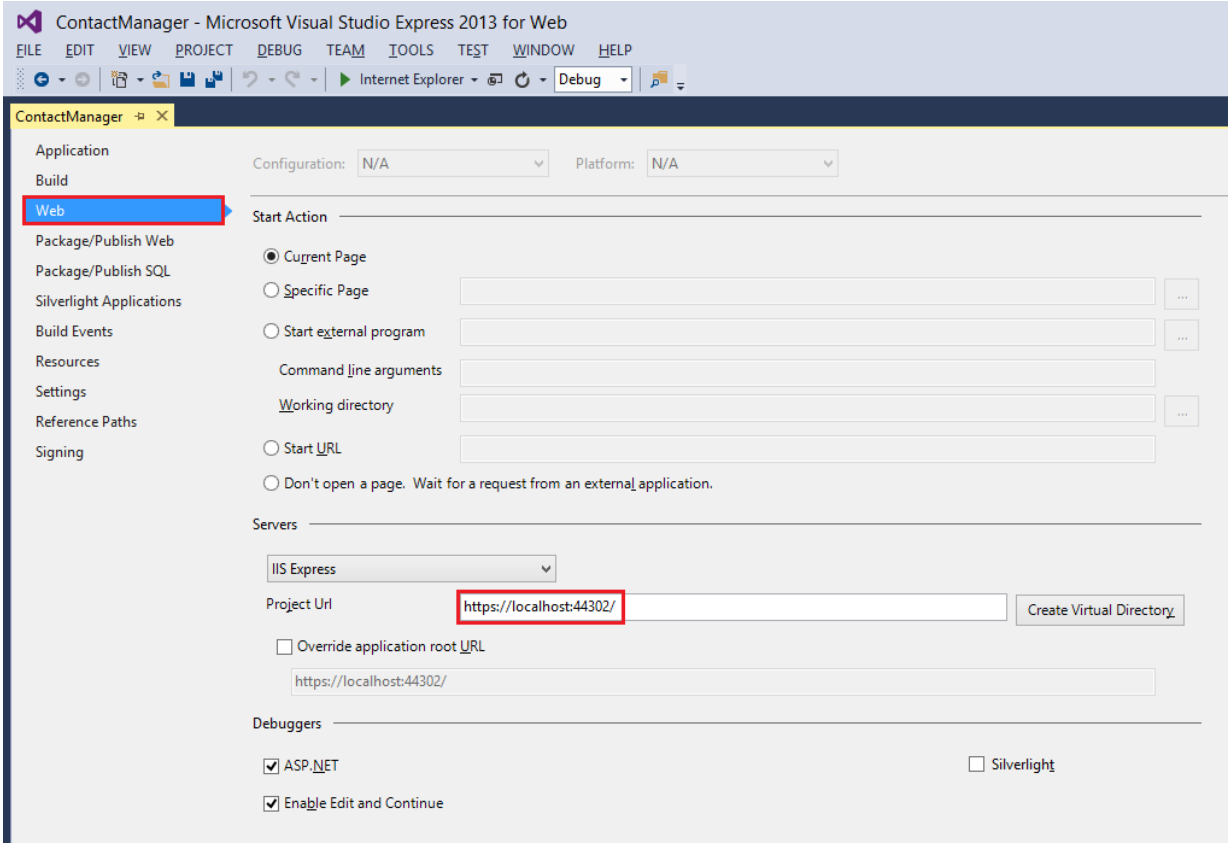
Enable SSL for the Project

Secure Sockets Layer (SSL) is a protocol defined to allow Web servers and Web clients to communicate more securely through the use of encryption. When SSL is not used, data sent between the client and server is open to packet sniffing by anyone with physical access to the network. Additionally, several common authentication schemes are not secure over plain HTTP. In particular, Basic authentication and forms authentication send unencrypted credentials. To be secure, these authentication schemes must use SSL.

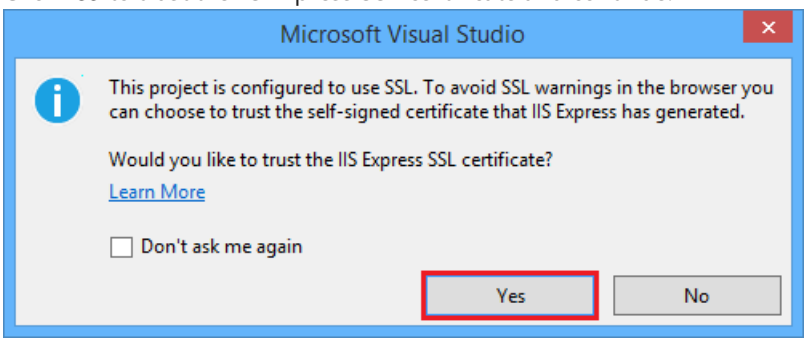
1. In **Solution Explorer**, click the **WingtipToys** project, then press **F4** to display the **Properties** window.
2. Change **SSL Enabled** to `true`.
3. Copy the **SSL URL** so you can use it later.
   The SSL URL will be `https://localhost:44300/` unless you've previously created SSL Web Sites (as shown below).



4. In **Solution Explorer**, right click the **WingtipToys** project and click **Properties**.
5. In the left tab, click **Web**.
6. Change the **Project Url** to use the **SSL URL** that you saved earlier.

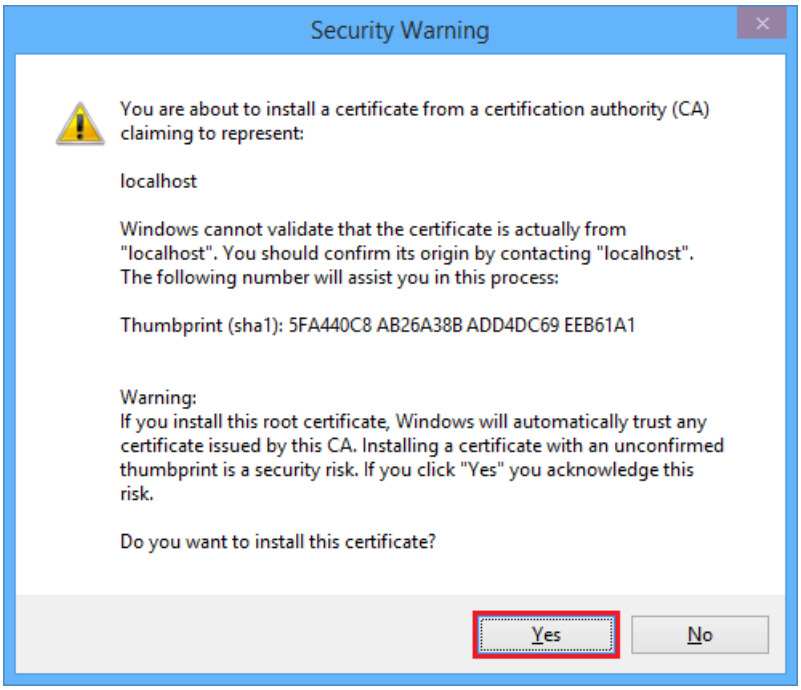


7. Save the page by pressing **CTRL+S**.
8. Press **Ctrl+F5** to run the application. Visual Studio will display an option to allow you to avoid SSL warnings.
9. Click **Yes** to trust the IIS Express SSL certificate and continue.



A Security Warning is displayed.

10. Click **Yes** to install the certificate to your localhost.



The browser window will be displayed.

You can now easily test your Web application locally using SSL.

Add an OAuth 2.0 Provider

ASP.NET Web Forms provides enhanced options for membership and authentication. These enhancements include OAuth. OAuth is an open protocol that allows secure authorization in a simple and standard method from web, mobile, and desktop applications. The ASP.NET Web Forms template uses OAuth to expose Facebook, Twitter, Google and Microsoft as authentication providers. Although this tutorial uses only Google as the authentication provider, you can easily modify the code to use any of the providers. The steps to implement other providers are very similar to the steps you will see in this tutorial.

In addition to authentication, the tutorial will also use roles to implement authorization. Only those users you add to the `canEdit` role will be able to change data (create, edit, or delete contacts).

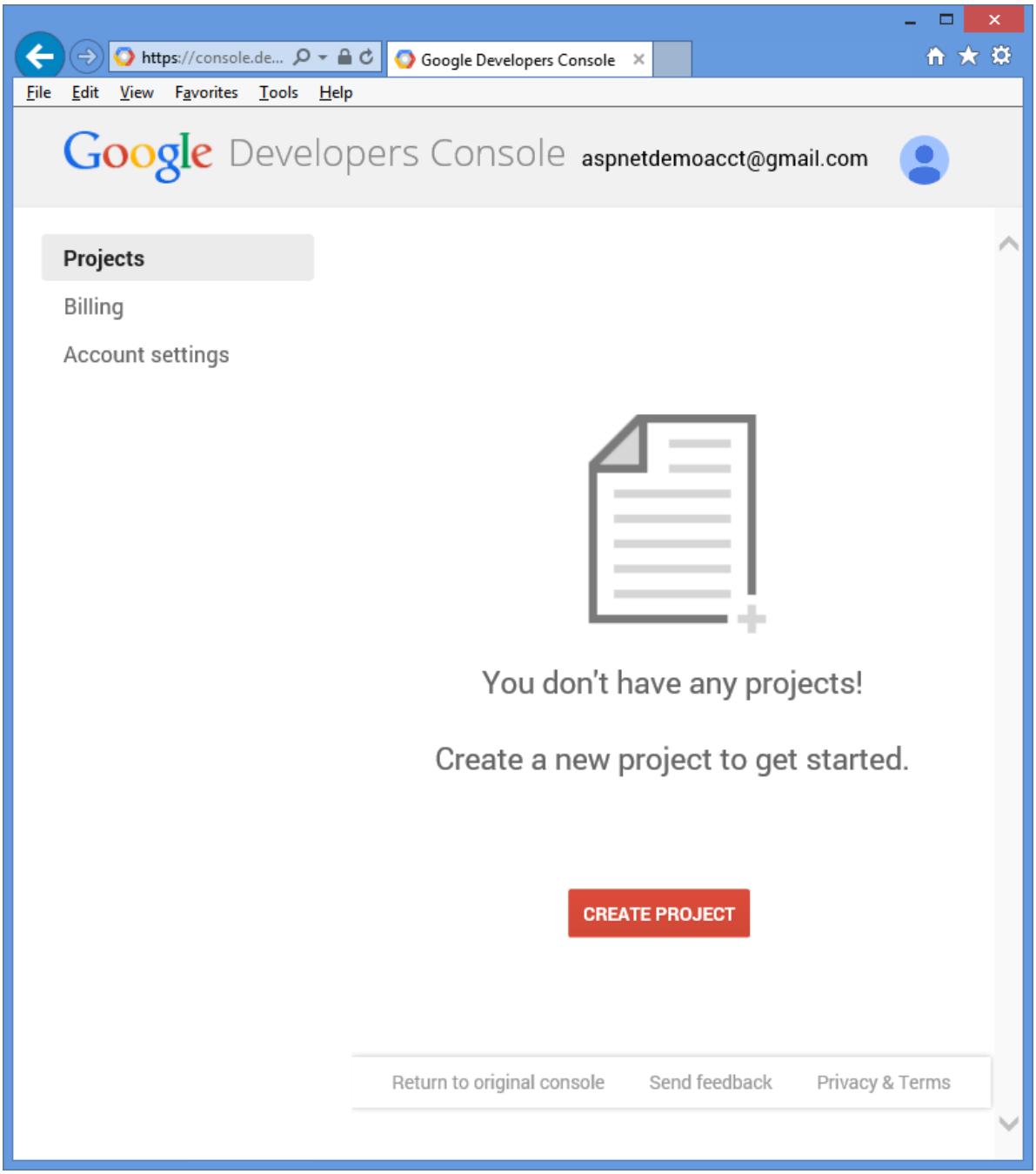
> **NOTE**
> Windows Live applications only accept a live URL for a working website, so you cannot use a local website URL for testing logins.

The following steps will allow you to add a Google authentication provider.

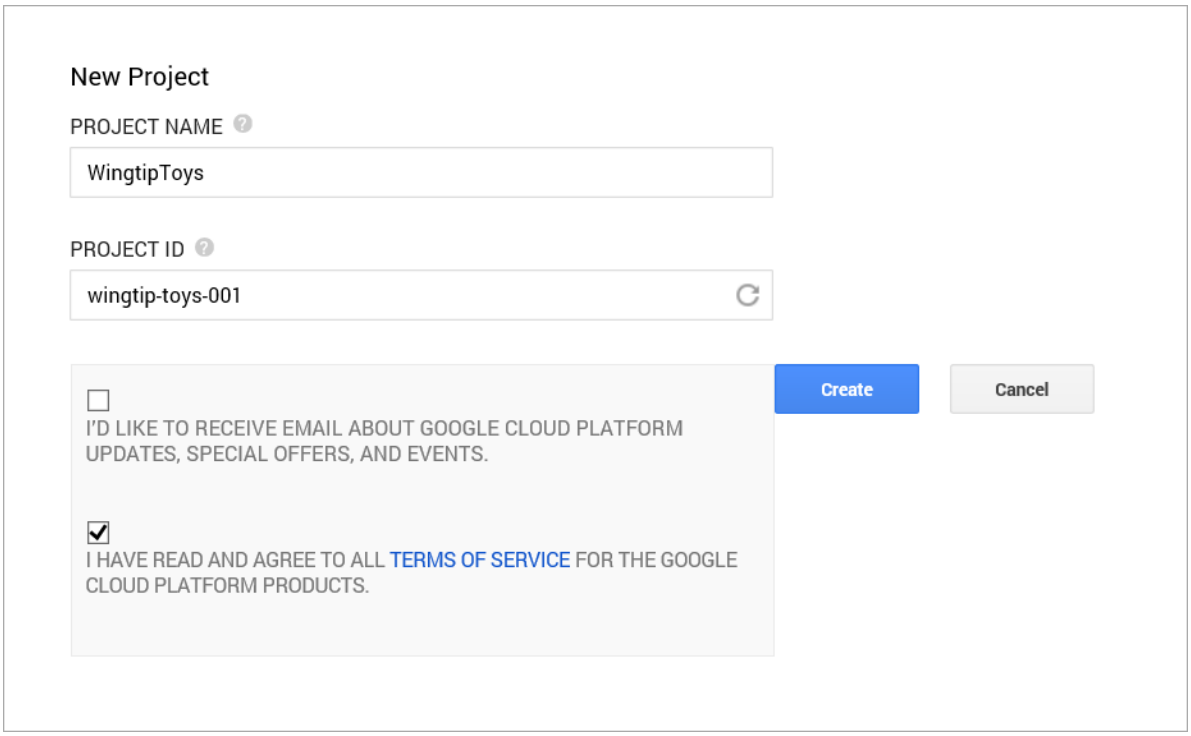
1. Open the *App_Start\Startup.Auth.cs* file.
2. Remove the comment characters from the `app.UseGoogleAuthentication()` method so that the method appears as follows:

```
app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
{
    ClientId = "",
    ClientSecret = ""
});
```

3. Navigate to the Google Developers Console. You will also need to sign-in with your Google developer email account (gmail.com). If you do not have a Google account, select the **Create an account** link. Next, you'll see the **Google Developers Console**.



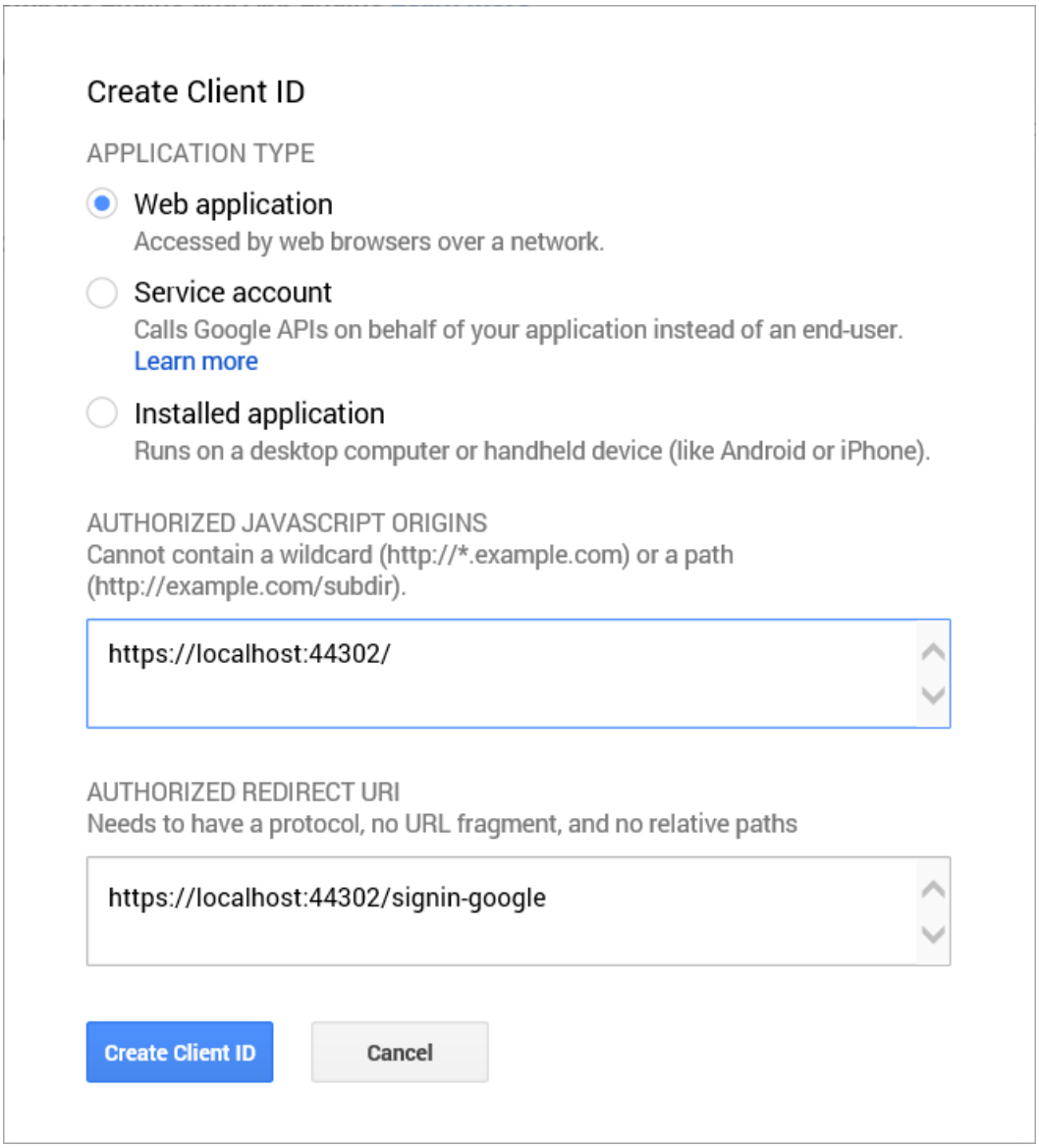
4. Click the **Create Project** button and enter a project name and ID (you can use the default values). Then, click the **agreement checkbox** and the **Create** button.



In a few seconds the new project will be created and your browser will display the new projects page.

5. In the left tab, click **APIs & auth**, and then click **Credentials**.

6. Click the **Create New Client ID** under **OAuth**.
   The **Create Client ID** dialog will be displayed.



7. In the **Create Client ID** dialog, keep the default **Web application** for the application type.

8. Set the **Authorized JavaScript Origins** to the SSL URL you used earlier in this tutorial (`https://localhost:44300/` unless you've created other SSL projects).
   This URL is the origin for your application. For this sample, you will only enter the localhost test URL. However, you can enter multiple URLs to account for localhost and production.

9. Set the **Authorized Redirect URI** to the following:

   ```
   https://localhost:44300/signin-google
   ```

   This value is the URI that ASP.NET OAuth users to communicate with the google OAuth server. Remember the SSL URL you used above (`https://localhost:44300/` unless you've created other SSL projects).

10. Click the **Create Client ID** button.

11. On the left menu of the Google Developers Console, click the **Consent screen** menu item, then set your email address and product name. When you have completed the form, click **Save**.

12. Click the **APIs** menu item, scroll down and click the **off** button next to **Google+ API**.
    Accepting this option will enable the Google+ API.

13. You must also update the **Microsoft.Owin** NuGet package to version 3.0.0.
    From the **Tools** menu, select **NuGet Package Manager** and then select **Manage NuGet Packages for**

Solution.
From the **Manage NuGet Packages** window, find and update the **Microsoft.Owin** package to version 3.0.0.

14. In Visual Studio, update the `UseGoogleAuthentication` method of the *Startup.Auth.cs* page by copying and pasting the **Client ID** and **Client Secret** into the method. The **Client ID** and **Client Secret** values shown below are samples and will not work.

```
using System;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.DataProtection;
using Microsoft.Owin.Security.Google;
using Owin;
using WingtipToys.Models;

namespace WingtipToys
{
    public partial class Startup {

        // For more information on configuring authentication, please visit
https://go.microsoft.com/fwlink/?LinkId=301883
        public void ConfigureAuth(IAppBuilder app)
        {
            // Configure the db context, user manager and signin manager to use a single instance per
request
            app.CreatePerOwinContext(ApplicationDbContext.Create);
            app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Create);
            app.CreatePerOwinContext<ApplicationSignInManager>(ApplicationSignInManager.Create);

            // Enable the application to use a cookie to store information for the signed in user
            // and to use a cookie to temporarily store information about a user logging in with a
third party login provider
            // Configure the sign in cookie
            app.UseCookieAuthentication(new CookieAuthenticationOptions
            {
                AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
                LoginPath = new PathString("/Account/Login"),
                Provider = new CookieAuthenticationProvider
                {
                    OnValidateIdentity =
SecurityStampValidator.OnValidateIdentity<ApplicationUserManager, ApplicationUser>(
                        validateInterval: TimeSpan.FromMinutes(30),
                        regenerateIdentity: (manager, user) => user.GenerateUserIdentityAsync(manager))
                }
            });
            // Use a cookie to temporarily store information about a user logging in with a third party
login provider
            app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);

            // Enables the application to temporarily store user information when they are verifying
the second factor in the two-factor authentication process.
            app.UseTwoFactorSignInCookie(DefaultAuthenticationTypes.TwoFactorCookie,
TimeSpan.FromMinutes(5));

            // Enables the application to remember the second login verification factor such as phone
or email.
            // Once you check this option, your second step of verification during the login process
will be remembered on the device where you logged in from.
            // This is similar to the RememberMe option when you log in.

app.UseTwoFactorRememberBrowserCookie(DefaultAuthenticationTypes.TwoFactorRememberBrowserCookie);

            // Uncomment the following lines to enable logging in with third party login providers
```

```
            // uncomment the following lines to enable logging in with third party login providers
            //app.UseMicrosoftAccountAuthentication(
            //    clientId: "",
            //    clientSecret: "");

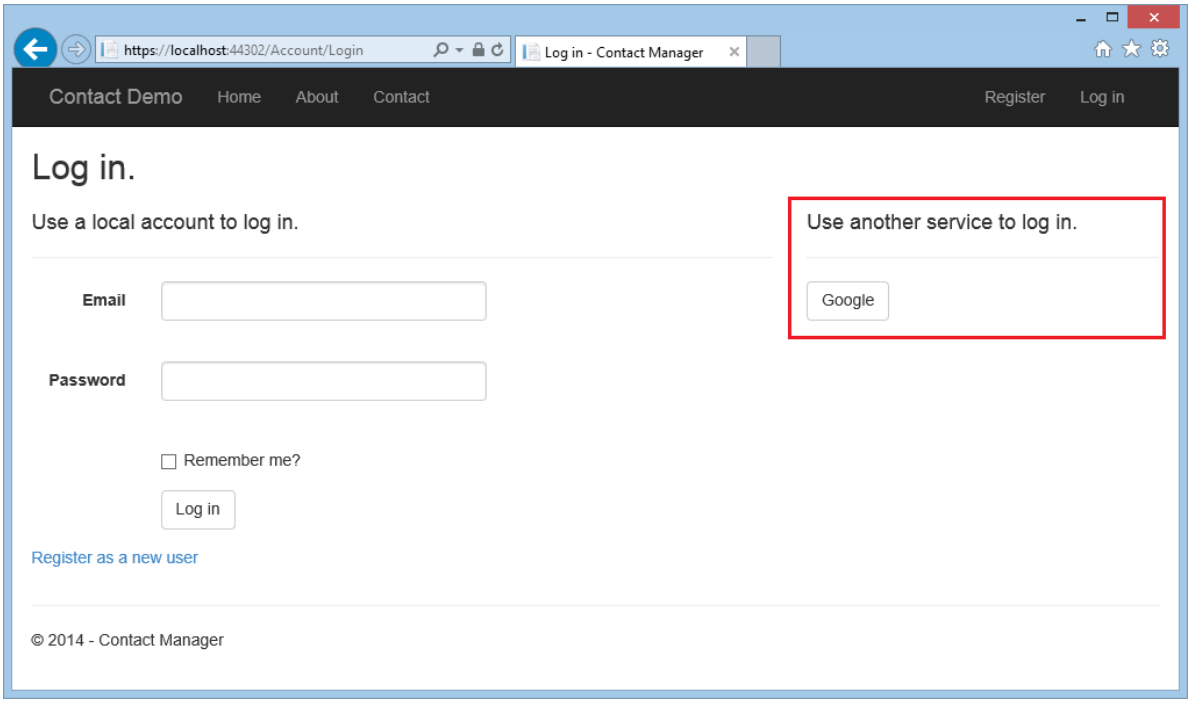
            //app.UseTwitterAuthentication(
            //   consumerKey: "",
            //   consumerSecret: "");

            //app.UseFacebookAuthentication(
            //   appId: "",
            //   appSecret: "");

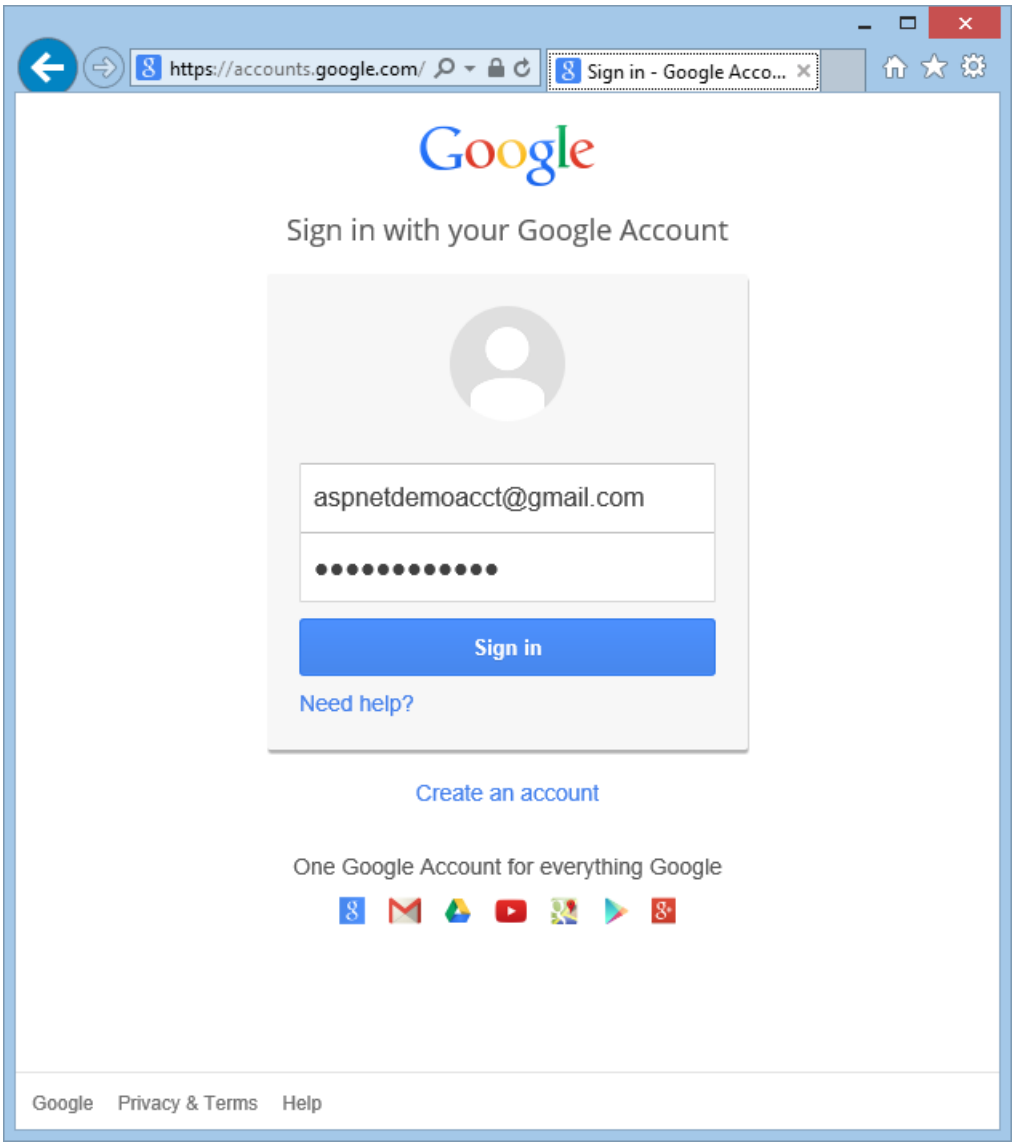
            app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
            {
              ClientId = "000000000000.apps.googleusercontent.com",
              ClientSecret = "00000000000"
            });
        }
    }
}
```

15. Press **CTRL+F5** to build and run the application. Click the **Log in** link.

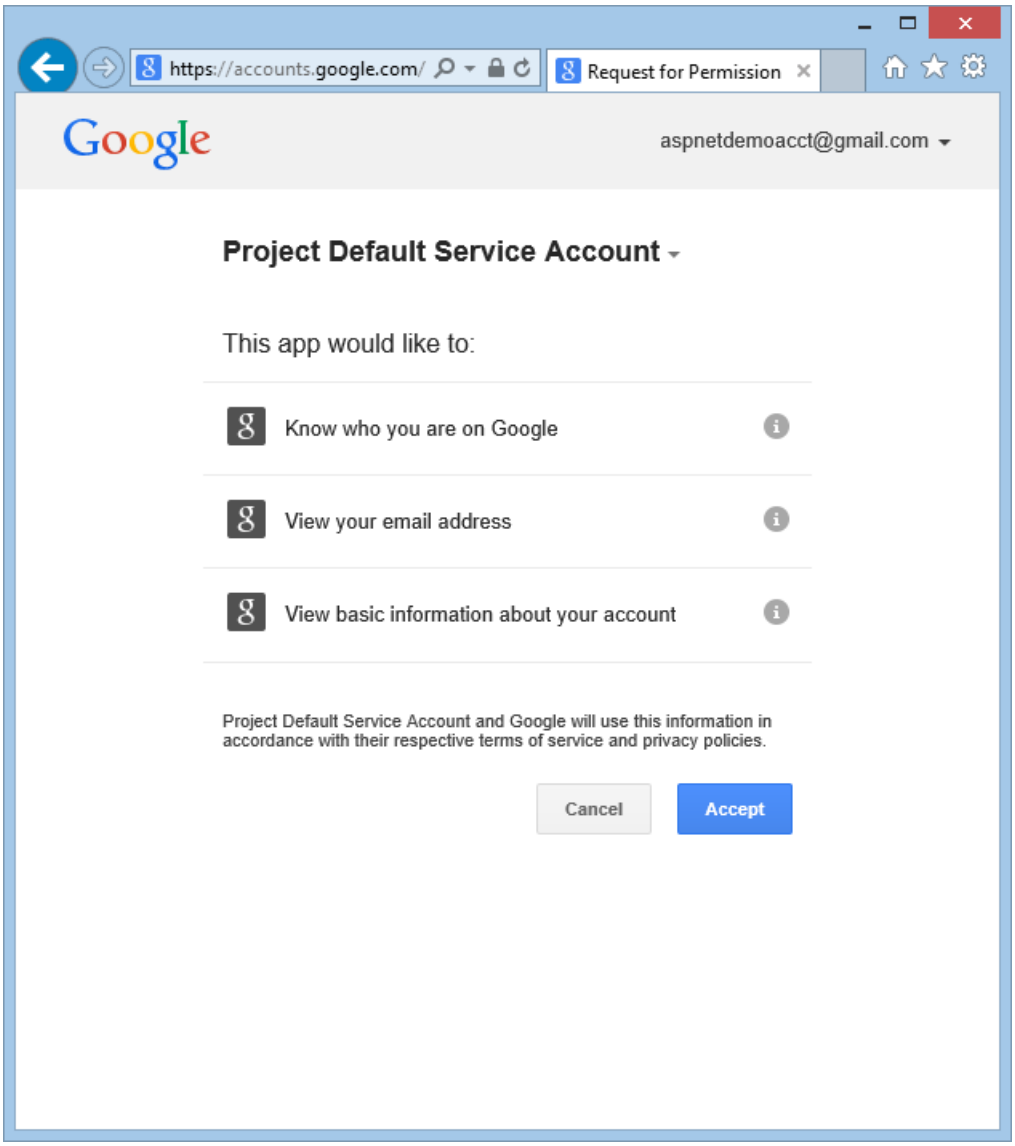
16. Under **Use another service to log in**, click **Google**.



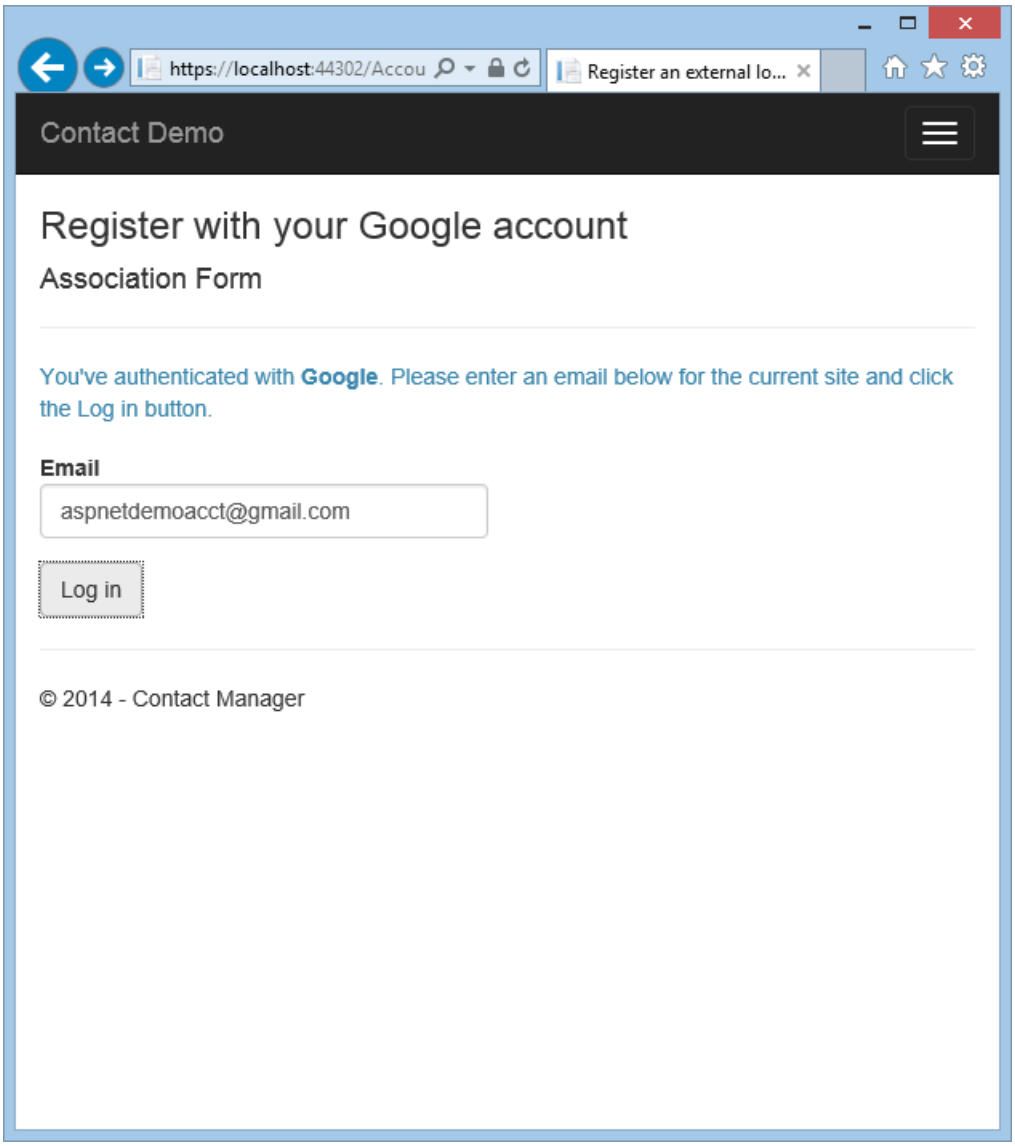
17. If you need to enter your credentials, you will be redirected to the google site where you will enter your credentials.



18. After you enter your credentials, you will be prompted to give permissions to the web application you just created.



19. Click **Accept**. You will now be redirected back to the **Register** page of the **WingtipToys** application where you can register your Google account.



20. You have the option of changing the local email registration name used for your Gmail account, but you generally want to keep the default email alias (that is, the one you used for authentication). Click **Log in** as shown above.

Modifying Login Functionality

As previously mentioned in this tutorial series, much of the user registration functionality has been included in the ASP.NET Web Forms template by default. Now you will modify the default *Login.aspx* and *Register.aspx* pages to call the `MigrateCart` method. The `MigrateCart` method associates a newly logged in user with an anonymous shopping cart. By associating the user and shopping cart, the Wingtip Toys sample application will be able to maintain the shopping cart of the user between visits.

1. In **Solution Explorer**, find and open the *Account* folder.

2. Modify the code-behind page named *Login.aspx.cs* to include the code highlighted in yellow, so that it appears as follows:

```
using System;
using System.Web;
using System.Web.UI;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using Owin;
using WingtipToys.Models;

namespace WingtipToys.Account
{
```

```
    public partial class Login : Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            RegisterHyperLink.NavigateUrl = "Register";
            // Enable this once you have account confirmation enabled for password reset functionality
            //ForgotPasswordHyperLink.NavigateUrl = "Forgot";
            OpenAuthLogin.ReturnUrl = Request.QueryString["ReturnUrl"];
            var returnUrl = HttpUtility.UrlEncode(Request.QueryString["ReturnUrl"]);
            if (!String.IsNullOrEmpty(returnUrl))
            {
                RegisterHyperLink.NavigateUrl += "?ReturnUrl=" + returnUrl;
            }
        }

        protected void LogIn(object sender, EventArgs e)
        {
            if (IsValid)
            {
                // Validate the user password
                var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
                var signinManager = Context.GetOwinContext().GetUserManager<ApplicationSignInManager>
();

                // This doen't count login failures towards account lockout
                // To enable password failures to trigger lockout, change to shouldLockout: true
                var result = signinManager.PasswordSignIn(Email.Text, Password.Text,
RememberMe.Checked, shouldLockout: false);

                switch (result)
                {
                    case SignInStatus.Success:
                        WingtipToys.Logic.ShoppingCartActions usersShoppingCart = new
WingtipToys.Logic.ShoppingCartActions();
                        String cartId = usersShoppingCart.GetCartId();
                        usersShoppingCart.MigrateCart(cartId, Email.Text);

                        IdentityHelper.RedirectToReturnUrl(Request.QueryString["ReturnUrl"], Response);
                        break;
                    case SignInStatus.LockedOut:
                        Response.Redirect("/Account/Lockout");
                        break;
                    case SignInStatus.RequiresVerification:
                        Response.Redirect(String.Format("/Account/TwoFactorAuthenticationSignIn?
ReturnUrl={0}&RememberMe={1}",
                                                          Request.QueryString["ReturnUrl"],
                                                          RememberMe.Checked),
                                            true);
                        break;
                    case SignInStatus.Failure:
                    default:
                        FailureText.Text = "Invalid login attempt";
                        ErrorMessage.Visible = true;
                        break;
                }
            }
        }
    }
}
```

3. Save the *Login.aspx.cs* file.

For now, you can ignore the warning that there is no definition for the `MigrateCart` method. You will be adding it a bit later in this tutorial.

The *Login.aspx.cs* code-behind file supports a LogIn method. By inspecting the Login.aspx page, you'll see that this page includes a "Log in" button that when click triggers the `LogIn` handler on the code-behind.

When the `Login` method on the *Login.aspx.cs* is called, a new instance of the shopping cart named `usersShoppingCart` is created. The ID of the shopping cart (a GUID) is retrieved and set to the `cartId` variable. Then, the `MigrateCart` method is called, passing both the `cartId` and the name of the logged-in user to this method. When the shopping cart is migrated, the GUID used to identify the anonymous shopping cart is replaced with the user name.

In addition to modifying the *Login.aspx.cs* code-behind file to migrate the shopping cart when the user logs in, you must also modify the *Register.aspx.cs code-behind file* to migrate the shopping cart when the user creates a new account and logs in.

1. In the *Account* folder, open the code-behind file named *Register.aspx.cs*.

2. Modify the code-behind file by including the code in yellow, so that it appears as follows:

```
using System;
using System.Linq;
using System.Web;
using System.Web.UI;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using Owin;
using WingtipToys.Models;

namespace WingtipToys.Account
{
    public partial class Register : Page
    {
        protected void CreateUser_Click(object sender, EventArgs e)
        {
            var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
            var user = new ApplicationUser() { UserName = Email.Text, Email = Email.Text };
            IdentityResult result = manager.Create(user, Password.Text);
            if (result.Succeeded)
            {
                // For more information on how to enable account confirmation and password reset please
visit https://go.microsoft.com/fwlink/?LinkID=320771
                //string code = manager.GenerateEmailConfirmationToken(user.Id);
                //string callbackUrl = IdentityHelper.GetUserConfirmationRedirectUrl(code, user.Id,
Request);
                //manager.SendEmail(user.Id, "Confirm your account", "Please confirm your account by
clicking <a href=\"" + callbackUrl + "\">here</a>.");

                IdentityHelper.SignIn(manager, user, isPersistent: false);

                using (WingtipToys.Logic.ShoppingCartActions usersShoppingCart = new
WingtipToys.Logic.ShoppingCartActions())
                {
                  String cartId = usersShoppingCart.GetCartId();
                  usersShoppingCart.MigrateCart(cartId, user.Id);
                }

                IdentityHelper.RedirectToReturnUrl(Request.QueryString["ReturnUrl"], Response);
            }
            else
            {
                ErrorMessage.Text = result.Errors.FirstOrDefault();
            }
        }
    }
}
```

3. Save the *Register.aspx.cs* file. Once again, ignore the warning about the `MigrateCart` method.

Notice that the code you used in the `CreateUser_Click` event handler is very similar to the code you used in the

`LogIn` method. When the user registers or logs in to the site, a call to the `MigrateCart` method will be made.

Migrating the Shopping Cart

Now that you have the log-in and registration process updated, you can add the code to migrate the shopping cart using the `MigrateCart` method.

1. In **Solution Explorer**, find the *Logic* folder and open the *ShoppingCartActions.cs* class file.
2. Add the code highlighted in yellow to the existing code in the *ShoppingCartActions.cs* file, so that the code in the *ShoppingCartActions.cs* file appears as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using WingtipToys.Models;

namespace WingtipToys.Logic
{
  public class ShoppingCartActions : IDisposable
  {
    public string ShoppingCartId { get; set; }

    private ProductContext _db = new ProductContext();

    public const string CartSessionKey = "CartId";

    public void AddToCart(int id)
    {
      // Retrieve the product from the database.
      ShoppingCartId = GetCartId();

      var cartItem = _db.ShoppingCartItems.SingleOrDefault(
          c => c.CartId == ShoppingCartId
          && c.ProductId == id);
      if (cartItem == null)
      {
        // Create a new cart item if no cart item exists.
        cartItem = new CartItem
        {
          ItemId = Guid.NewGuid().ToString(),
          ProductId = id,
          CartId = ShoppingCartId,
          Product = _db.Products.SingleOrDefault(
           p => p.ProductID == id),
          Quantity = 1,
          DateCreated = DateTime.Now
        };

        _db.ShoppingCartItems.Add(cartItem);
      }
      else
      {
        // If the item does exist in the cart,
        // then add one to the quantity.
        cartItem.Quantity++;
      }
      _db.SaveChanges();
    }

    public void Dispose()
    {
      if (_db != null)
      {
        _db.Dispose();
```

```
      _db = null;
    }
  }

  public string GetCartId()
  {
    if (HttpContext.Current.Session[CartSessionKey] == null)
    {
      if (!string.IsNullOrWhiteSpace(HttpContext.Current.User.Identity.Name))
      {
        HttpContext.Current.Session[CartSessionKey] = HttpContext.Current.User.Identity.Name;
      }
      else
      {
        // Generate a new random GUID using System.Guid class.
        Guid tempCartId = Guid.NewGuid();
        HttpContext.Current.Session[CartSessionKey] = tempCartId.ToString();
      }
    }
    return HttpContext.Current.Session[CartSessionKey].ToString();
  }

  public List<CartItem> GetCartItems()
  {
    ShoppingCartId = GetCartId();

    return _db.ShoppingCartItems.Where(
        c => c.CartId == ShoppingCartId).ToList();
  }

  public decimal GetTotal()
  {
    ShoppingCartId = GetCartId();
    // Multiply product price by quantity of that product to get
    // the current price for each of those products in the cart.
    // Sum all product price totals to get the cart total.
    decimal? total = decimal.Zero;
    total = (decimal?)(from cartItems in _db.ShoppingCartItems
                       where cartItems.CartId == ShoppingCartId
                       select (int?)cartItems.Quantity *
                       cartItems.Product.UnitPrice).Sum();
    return total ?? decimal.Zero;
  }

  public ShoppingCartActions GetCart(HttpContext context)
  {
    using (var cart = new ShoppingCartActions())
    {
      cart.ShoppingCartId = cart.GetCartId();
      return cart;
    }
  }

  public void UpdateShoppingCartDatabase(String cartId, ShoppingCartUpdates[] CartItemUpdates)
  {
    using (var db = new WingtipToys.Models.ProductContext())
    {
      try
      {
        int CartItemCount = CartItemUpdates.Count();
        List<CartItem> myCart = GetCartItems();
        foreach (var cartItem in myCart)
        {
          // Iterate through all rows within shopping cart list
          for (int i = 0; i < CartItemCount; i++)
          {
            if (cartItem.Product.ProductID == CartItemUpdates[i].ProductId)
            {
              if (CartItemUpdates[i].PurchaseQuantity < 1 || CartItemUpdates[i].RemoveItem == true)
```

```
                {
                  RemoveItem(cartId, cartItem.ProductId);
                }
                else
                {
                  UpdateItem(cartId, cartItem.ProductId, CartItemUpdates[i].PurchaseQuantity);
                }
              }
            }
          }
        }
        catch (Exception exp)
        {
          throw new Exception("ERROR: Unable to Update Cart Database - " + exp.Message.ToString(),
exp);
        }
      }
    }

    public void RemoveItem(string removeCartID, int removeProductID)
    {
      using (var _db = new WingtipToys.Models.ProductContext())
      {
        try
        {
          var myItem = (from c in _db.ShoppingCartItems where c.CartId == removeCartID &&
c.Product.ProductID == removeProductID select c).FirstOrDefault();
          if (myItem != null)
          {
            // Remove Item.
            _db.ShoppingCartItems.Remove(myItem);
            _db.SaveChanges();
          }
        }
        catch (Exception exp)
        {
          throw new Exception("ERROR: Unable to Remove Cart Item - " + exp.Message.ToString(), exp);
        }
      }
    }

    public void UpdateItem(string updateCartID, int updateProductID, int quantity)
    {
      using (var _db = new WingtipToys.Models.ProductContext())
      {
        try
        {
          var myItem = (from c in _db.ShoppingCartItems where c.CartId == updateCartID &&
c.Product.ProductID == updateProductID select c).FirstOrDefault();
          if (myItem != null)
          {
            myItem.Quantity = quantity;
            _db.SaveChanges();
          }
        }
        catch (Exception exp)
        {
          throw new Exception("ERROR: Unable to Update Cart Item - " + exp.Message.ToString(), exp);
        }
      }
    }

    public void EmptyCart()
    {
      ShoppingCartId = GetCartId();
      var cartItems = _db.ShoppingCartItems.Where(
          c => c.CartId == ShoppingCartId);
      foreach (var cartItem in cartItems)
      {
```

```
        {
          _db.ShoppingCartItems.Remove(cartItem);
        }
        // Save changes.
        _db.SaveChanges();
      }

      public int GetCount()
      {
        ShoppingCartId = GetCartId();

        // Get the count of each item in the cart and sum them up
        int? count = (from cartItems in _db.ShoppingCartItems
                      where cartItems.CartId == ShoppingCartId
                      select (int?)cartItems.Quantity).Sum();
        // Return 0 if all entries are null
        return count ?? 0;
      }

      public struct ShoppingCartUpdates
      {
        public int ProductId;
        public int PurchaseQuantity;
        public bool RemoveItem;
      }

      public void MigrateCart(string cartId, string userName)
      {
        var shoppingCart = _db.ShoppingCartItems.Where(c => c.CartId == cartId);
        foreach (CartItem item in shoppingCart)
        {
          item.CartId = userName;
        }
        HttpContext.Current.Session[CartSessionKey] = userName;
        _db.SaveChanges();
      }
    }
  }
```

The `MigrateCart` method uses the existing cartId to find the shopping cart of the user. Next, the code loops through all the shopping cart items and replaces the `CartId` property (as specified by the `CartItem` schema) with the logged-in user name.

Updating the Database Connection

If you are following this tutorial using the **prebuilt** Wingtip Toys sample application, you must recreate the default membership database. By modifying the default connection string, the membership database will be created the next time the application runs.

1. Open the *Web.config* file at the root of the project.
2. Update the default connection string so that it appears as follows:

```
<add name="DefaultConnection" connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=aspnet-
WingtipToys;Integrated Security=True" providerName="System.Data.SqlClient" />
```

Integrating PayPal

PayPal is a web-based billing platform that accepts payments by online merchants. This tutorial next explains how to integrate PayPal's Express Checkout functionality into your application. Express Checkout allows your customers to use PayPal to pay for the items they have added to their shopping cart.

Create PayPal Test Accounts

To use the PayPal testing environment, you must create and verify a developer test account. You will use the developer test account to create a buyer test account and a seller test account. The developer test account credentials also will allow the Wingtip Toys sample application to access the PayPal testing environment.

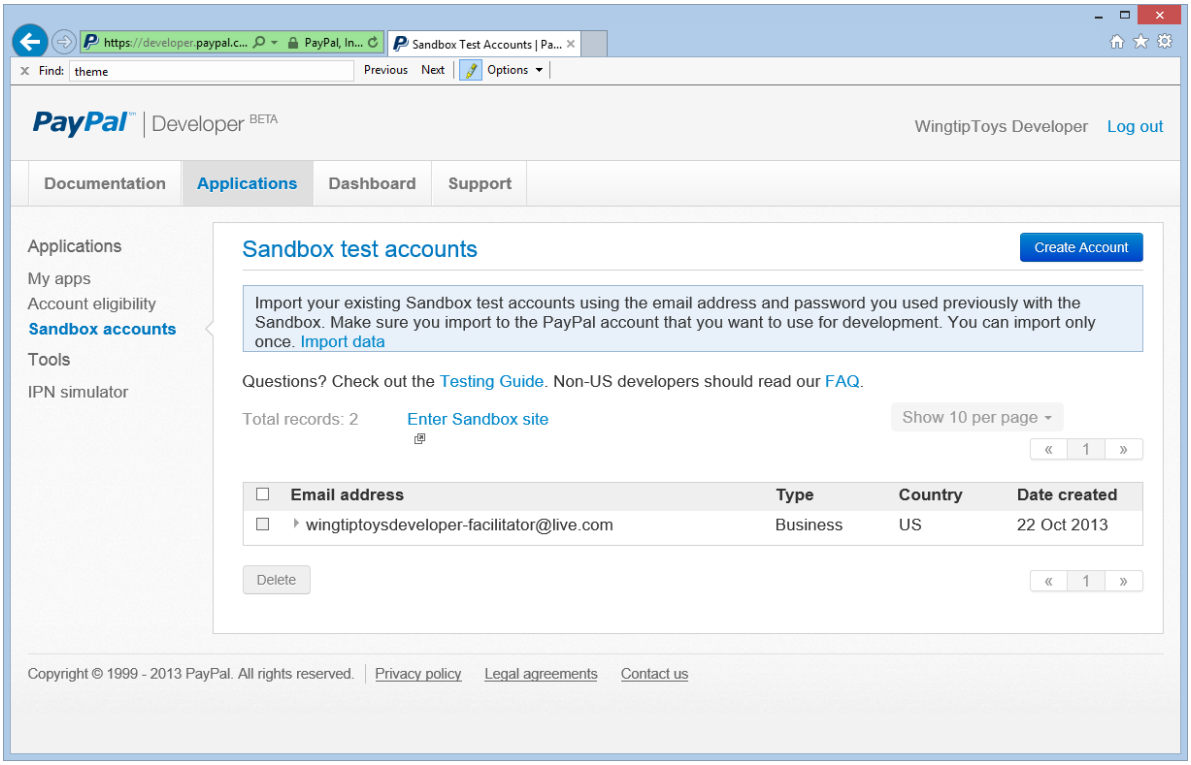
1. In a browser, navigate to the PayPal developer testing site:
   https://developer.paypal.com

2. If you don't have a PayPal developer account, create a new account by clicking **Sign Up**and following the sign up steps. If you have an existing PayPal developer account, sign in by clicking **Log In**. You will need your PayPal developer account to test the Wingtip Toys sample application later in this tutorial.

3. If you have just signed up for your PayPal developer account, you may need to verify your PayPal developer account with PayPal. You can verify your account by following the steps that PayPal sent to your email account. Once you have verified your PayPal developer account, log back into the PayPal developer testing site.

4. After you are logged in to the PayPal developer site with your PayPal developer account you need to create a PayPal buyer test account if you don't already have one. To create a buyer test account, on the PayPal site click the **Applications** tab and then click **Sandbox accounts**.
   The **Sandbox test accounts** page is shown.

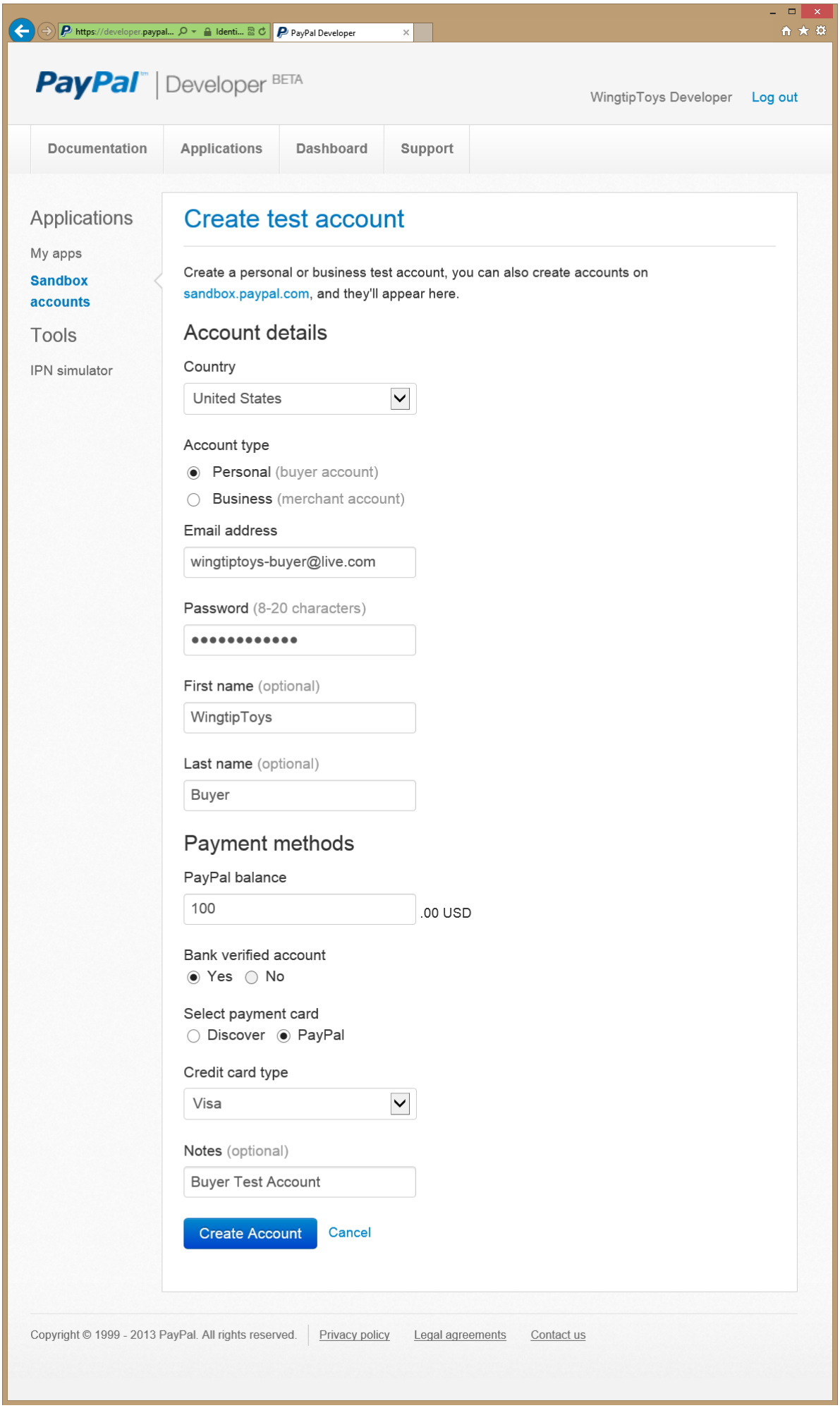
   > **NOTE**
   >
   > The PayPal Developer site already provides a merchant test account.



5. On the Sandbox test accounts page, click **Create Account**.

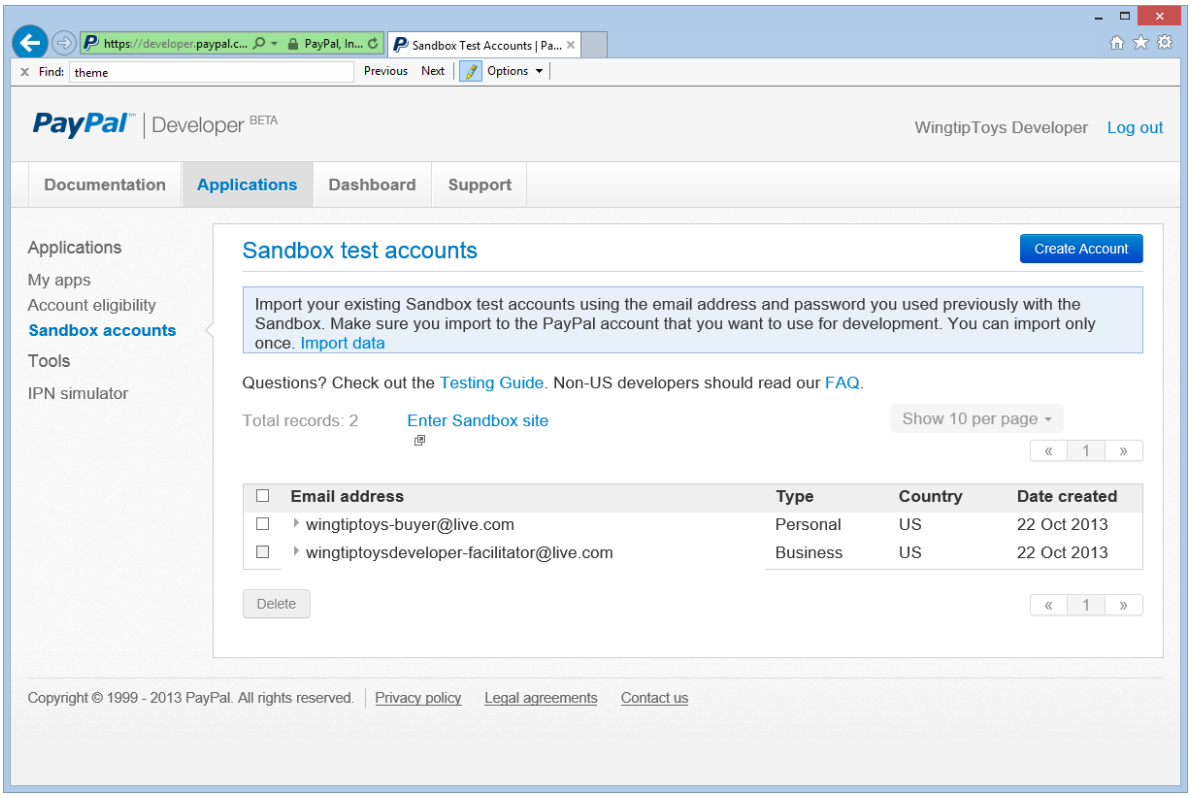
6. On the **Create test account** page choose a buyer test account email and password of your choice.

   > **NOTE**
   >
   > You will need the buyer email addresses and password to test the Wingtip Toys sample application at the end of this tutorial.



7. Create the buyer test account by clicking the **Create Account** button.

The **Sandbox Test accounts** page is displayed.



8. On the **Sandbox test accounts** page, click the **facilitator** email account.
   Profile and **Notification** options appear.

9. Select the **Profile** option, then click **API credentials** to view your API credentials for the merchant test account.

10. Copy the TEST API credentials to notepad.

You will need your displayed Classic TEST API credentials (Username, Password, and Signature) to make API calls from the Wingtip Toys sample application to the PayPal testing environment. You will add the credentials in the next step.

Add PayPal Class and API Credentials

You will place the majority of the PayPal code into a single class. This class contains the methods used to communicate with PayPal. Also, you will add your PayPal credentials to this class.

1. In the Wingtip Toys sample application within Visual Studio, right-click the **Logic** folder and then select **Add -> New Item**.
   The **Add New Item** dialog box is displayed.

2. Under **Visual C#** from the **Installed** pane on the left, select **Code**.

3. From the middle pane, select **Class**. Name this new class **PayPalFunctions.cs**.

4. Click **Add**.
   The new class file is displayed in the editor.

5. Replace the default code with the following code:

```
using System;
using System.Collections;
using System.Collections.Specialized;
using System.IO;
using System.Net;
using System.Text;
```

```
using System.Data;
using System.Configuration;
using System.Web;
using WingtipToys;
using WingtipToys.Models;
using System.Collections.Generic;
using System.Linq;

public class NVPAPICaller
{
  //Flag that determines the PayPal environment (live or sandbox)
  private const bool bSandbox = true;
  private const string CVV2 = "CVV2";

  // Live strings.
  private string pEndPointURL = "https://api-3t.paypal.com/nvp";
  private string host = "www.paypal.com";

  // Sandbox strings.
  private string pEndPointURL_SB = "https://api-3t.sandbox.paypal.com/nvp";
  private string host_SB = "www.sandbox.paypal.com";

  private const string SIGNATURE = "SIGNATURE";
  private const string PWD = "PWD";
  private const string ACCT = "ACCT";

  //Replace <Your API Username> with your API Username
  //Replace <Your API Password> with your API Password
  //Replace <Your Signature> with your Signature
  public string APIUsername = "<Your API Username>";
  private string APIPassword = "<Your API Password>";
  private string APISignature = "<Your Signature>";
  private string Subject = "";
  private string BNCode = "PP-ECWizard";

  //HttpWebRequest Timeout specified in milliseconds
  private const int Timeout = 15000;
  private static readonly string[] SECURED_NVPS = new string[] { ACCT, CVV2, SIGNATURE, PWD };

  public void SetCredentials(string Userid, string Pwd, string Signature)
  {
    APIUsername = Userid;
    APIPassword = Pwd;
    APISignature = Signature;
  }

  public bool ShortcutExpressCheckout(string amt, ref string token, ref string retMsg)
  {
    if (bSandbox)
    {
      pEndPointURL = pEndPointURL_SB;
      host = host_SB;
    }

    string returnURL = "https://localhost:44300/Checkout/CheckoutReview.aspx";
    string cancelURL = "https://localhost:44300/Checkout/CheckoutCancel.aspx";

    NVPCodec encoder = new NVPCodec();
    encoder["METHOD"] = "SetExpressCheckout";
    encoder["RETURNURL"] = returnURL;
    encoder["CANCELURL"] = cancelURL;
    encoder["BRANDNAME"] = "Wingtip Toys Sample Application";
    encoder["PAYMENTREQUEST_0_AMT"] = amt;
    encoder["PAYMENTREQUEST_0_ITEMAMT"] = amt;
    encoder["PAYMENTREQUEST_0_PAYMENTACTION"] = "Sale";
    encoder["PAYMENTREQUEST_0_CURRENCYCODE"] = "USD";

    // Get the Shopping Cart Products
    using (WingtipToys.Logic.ShoppingCartActions myCartOrders = new
```

```
WingtipToys.Logic.ShoppingCartActions())
    {
      List<CartItem> myOrderList = myCartOrders.GetCartItems();

      for (int i = 0; i < myOrderList.Count; i++)
      {
        encoder["L_PAYMENTREQUEST_0_NAME" + i] = myOrderList[i].Product.ProductName.ToString();
        encoder["L_PAYMENTREQUEST_0_AMT" + i] = myOrderList[i].Product.UnitPrice.ToString();
        encoder["L_PAYMENTREQUEST_0_QTY" + i] = myOrderList[i].Quantity.ToString();
      }
    }

    string pStrrequestforNvp = encoder.Encode();
    string pStresponsenvp = HttpCall(pStrrequestforNvp);

    NVPCodec decoder = new NVPCodec();
    decoder.Decode(pStresponsenvp);

    string strAck = decoder["ACK"].ToLower();
    if (strAck != null && (strAck == "success" || strAck == "successwithwarning"))
    {
      token = decoder["TOKEN"];
      string ECURL = "https://" + host + "/cgi-bin/webscr?cmd=_express-checkout" + "&token=" + token;
      retMsg = ECURL;
      return true;
    }
    else
    {
      retMsg = "ErrorCode=" + decoder["L_ERRORCODE0"] + "&" +
          "Desc=" + decoder["L_SHORTMESSAGE0"] + "&" +
          "Desc2=" + decoder["L_LONGMESSAGE0"];
      return false;
    }
  }

  public bool GetCheckoutDetails(string token, ref string PayerID, ref NVPCodec decoder, ref string
retMsg)
  {
    if (bSandbox)
    {
      pEndPointURL = pEndPointURL_SB;
    }

    NVPCodec encoder = new NVPCodec();
    encoder["METHOD"] = "GetExpressCheckoutDetails";
    encoder["TOKEN"] = token;

    string pStrrequestforNvp = encoder.Encode();
    string pStresponsenvp = HttpCall(pStrrequestforNvp);

    decoder = new NVPCodec();
    decoder.Decode(pStresponsenvp);

    string strAck = decoder["ACK"].ToLower();
    if (strAck != null && (strAck == "success" || strAck == "successwithwarning"))
    {
      PayerID = decoder["PAYERID"];
      return true;
    }
    else
    {
      retMsg = "ErrorCode=" + decoder["L_ERRORCODE0"] + "&" +
          "Desc=" + decoder["L_SHORTMESSAGE0"] + "&" +
          "Desc2=" + decoder["L_LONGMESSAGE0"];

      return false;
    }
  }
```

```
  public bool DoCheckoutPayment(string finalPaymentAmount, string token, string PayerID, ref NVPCodec
decoder, ref string retMsg)
  {
    if (bSandbox)
    {
      pEndPointURL = pEndPointURL_SB;
    }

    NVPCodec encoder = new NVPCodec();
    encoder["METHOD"] = "DoExpressCheckoutPayment";
    encoder["TOKEN"] = token;
    encoder["PAYERID"] = PayerID;
    encoder["PAYMENTREQUEST_0_AMT"] = finalPaymentAmount;
    encoder["PAYMENTREQUEST_0_CURRENCYCODE"] = "USD";
    encoder["PAYMENTREQUEST_0_PAYMENTACTION"] = "Sale";

    string pStrrequestforNvp = encoder.Encode();
    string pStresponsenvp = HttpCall(pStrrequestforNvp);

    decoder = new NVPCodec();
    decoder.Decode(pStresponsenvp);

    string strAck = decoder["ACK"].ToLower();
    if (strAck != null && (strAck == "success" || strAck == "successwithwarning"))
    {
      return true;
    }
    else
    {
      retMsg = "ErrorCode=" + decoder["L_ERRORCODE0"] + "&" +
          "Desc=" + decoder["L_SHORTMESSAGE0"] + "&" +
          "Desc2=" + decoder["L_LONGMESSAGE0"];

      return false;
    }
  }

  public string HttpCall(string NvpRequest)
  {
    string url = pEndPointURL;

    string strPost = NvpRequest + "&" + buildCredentialsNVPString();
    strPost = strPost + "&BUTTONSOURCE=" + HttpUtility.UrlEncode(BNCode);

    HttpWebRequest objRequest = (HttpWebRequest)WebRequest.Create(url);
    objRequest.Timeout = Timeout;
    objRequest.Method = "POST";
    objRequest.ContentLength = strPost.Length;

    try
    {
      using (StreamWriter myWriter = new StreamWriter(objRequest.GetRequestStream()))
      {
        myWriter.Write(strPost);
      }
    }
    catch (Exception)
    {
      // No logging for this tutorial.
    }

    //Retrieve the Response returned from the NVP API call to PayPal.
    HttpWebResponse objResponse = (HttpWebResponse)objRequest.GetResponse();
    string result;
    using (StreamReader sr = new StreamReader(objResponse.GetResponseStream()))
    {
      result = sr.ReadToEnd();
    }
```

```
    return result;
  }

  private string buildCredentialsNVPString()
  {
    NVPCodec codec = new NVPCodec();

    if (!IsEmpty(APIUsername))
      codec["USER"] = APIUsername;

    if (!IsEmpty(APIPassword))
      codec[PWD] = APIPassword;

    if (!IsEmpty(APISignature))
      codec[SIGNATURE] = APISignature;

    if (!IsEmpty(Subject))
      codec["SUBJECT"] = Subject;

    codec["VERSION"] = "88.0";

    return codec.Encode();
  }

  public static bool IsEmpty(string s)
  {
    return s == null || s.Trim() == string.Empty;
  }
}

public sealed class NVPCodec : NameValueCollection
{
  private const string AMPERSAND = "&";
  private const string EQUALS = "=";
  private static readonly char[] AMPERSAND_CHAR_ARRAY = AMPERSAND.ToCharArray();
  private static readonly char[] EQUALS_CHAR_ARRAY = EQUALS.ToCharArray();

  public string Encode()
  {
    StringBuilder sb = new StringBuilder();
    bool firstPair = true;
    foreach (string kv in AllKeys)
    {
      string name = HttpUtility.UrlEncode(kv);
      string value = HttpUtility.UrlEncode(this[kv]);
      if (!firstPair)
      {
        sb.Append(AMPERSAND);
      }
      sb.Append(name).Append(EQUALS).Append(value);
      firstPair = false;
    }
    return sb.ToString();
  }

  public void Decode(string nvpstring)
  {
    Clear();
    foreach (string nvp in nvpstring.Split(AMPERSAND_CHAR_ARRAY))
    {
      string[] tokens = nvp.Split(EQUALS_CHAR_ARRAY);
      if (tokens.Length >= 2)
      {
        string name = HttpUtility.UrlDecode(tokens[0]);
        string value = HttpUtility.UrlDecode(tokens[1]);
        Add(name, value);
      }
    }
  }
```

```
    }

    public void Add(string name, string value, int index)
    {
      this.Add(GetArrayName(index, name), value);
    }

    public void Remove(string arrayName, int index)
    {
      this.Remove(GetArrayName(index, arrayName));
    }

    public string this[string name, int index]
    {
      get
      {
        return this[GetArrayName(index, name)];
      }
      set
      {
        this[GetArrayName(index, name)] = value;
      }
    }

    private static string GetArrayName(int index, string name)
    {
      if (index < 0)
      {
        throw new ArgumentOutOfRangeException("index", "index cannot be negative : " + index);
      }
      return name + index;
    }
  }
```

6. Add the Merchant API credentials (Username, Password, and Signature) that you displayed earlier in this tutorial so that you can make function calls to the PayPal testing environment.

```
public string APIUsername = "<Your API Username>";
private string APIPassword = "<Your API Password>";
private string APISignature = "<Your Signature>";
```

> **NOTE**
> In this sample application you are simply adding credentials to a C# file (.cs). However, in a implemented solution, you should consider encrypting your credentials in a configuration file.

The NVPAPICaller class contains the majority of the PayPal functionality. The code in the class provides the methods needed to make a test purchase from the PayPal testing environment. The following three PayPal functions are used to make purchases:

- `SetExpressCheckout` function
- `GetExpressCheckoutDetails` function
- `DoExpressCheckoutPayment` function

The `ShortcutExpressCheckout` method collects the test purchase information and product details from the shopping cart and calls the `SetExpressCheckout` PayPal function. The `GetCheckoutDetails` method confirms purchase details and calls the `GetExpressCheckoutDetails` PayPal function before making the test purchase. The `DoCheckoutPayment` method completes the test purchase from the testing environment by calling the `DoExpressCheckoutPayment` PayPal function. The remaining code supports the PayPal methods and process, such as encoding strings, decoding strings, processing arrays, and determining credentials.

> **NOTE**
> PayPal allows you to include optional purchase details based on PayPal's API specification. By extending the code in the Wingtip Toys sample application, you can include localization details, product descriptions, tax, a customer service number, as well as many other optional fields.

Notice that the return and cancel URLs that are specified in the **ShortcutExpressCheckout** method use a port number.

```
string returnURL = "https://localhost:44300/Checkout/CheckoutReview.aspx";
       string cancelURL = "https://localhost:44300/Checkout/CheckoutCancel.aspx";
```

When Visual Web Developer runs a web project using SSL, commonly the port 44300 is used for the web server. As shown above, the port number is 44300. When you run the application, you could see a different port number. Your port number needs to be correctly set in the code so that you can successful run the Wingtip Toys sample application at the end of this tutorial. The next section of this tutorial explains how to retrieve the local host port number and update the PayPal class.

Update the LocalHost Port Number in the PayPal Class

The Wingtip Toys sample application purchases products by navigating to the PayPal testing site and returning to your local instance of the Wingtip Toys sample application. In order to have PayPal return to the correct URL, you need to specify the port number of the locally running sample application in the PayPal code mentioned above.

1. Right-click the project name (**WingtipToys**) in **Solution Explorer** and select **Properties**.
2. In the left column, select the **Web** tab.
3. Retrieve the port number from the **Project Url** box.
4. If needed, update the `returnURL` and `cancelURL` in the PayPal class (`NVPAPICaller`) in the *PayPalFunctions.cs* file to use the port number of your web application:

   ```
   string returnURL = "https://localhost:<Your Port Number>/Checkout/CheckoutReview.aspx";
   string cancelURL = "https://localhost:<Your Port Number>/Checkout/CheckoutCancel.aspx";
   ```

Now the code that you added will match the expected port for your local Web application. PayPal will be able to return to the correct URL on your local machine.

Add the PayPal Checkout Button

Now that the primary PayPal functions have been added to the sample application, you can begin adding the markup and code needed to call these functions. First, you must add the checkout button that the user will see on the shopping cart page.

1. Open the *ShoppingCart.aspx* file.
2. Scroll to the bottom of the file and find the `<!--Checkout Placeholder -->` comment.
3. Replace the comment with an `ImageButton` control so that the mark up is replaced as follows:

   ```
   <asp:ImageButton ID="CheckoutImageBtn" runat="server"
                        ImageUrl="https://www.paypal.com/en_US/i/btn/btn_xpressCheckout.gif"
                        Width="145" AlternateText="Check out with PayPal"
                        OnClick="CheckoutBtn_Click"
                        BackColor="Transparent" BorderWidth="0" />
   ```

4. In the *ShoppingCart.aspx.cs* file, after the `UpdateBtn_Click` event handler near the end of the file, add the `CheckOutBtn_Click` event handler:

```
protected void CheckoutBtn_Click(object sender, ImageClickEventArgs e)
{
    using (ShoppingCartActions usersShoppingCart = new ShoppingCartActions())
    {
        Session["payment_amt"] = usersShoppingCart.GetTotal();
    }
    Response.Redirect("Checkout/CheckoutStart.aspx");
}
```

5. Also in the *ShoppingCart.aspx.cs* file, add a reference to the `CheckoutBtn`, so that the new image button is referenced as follows:

```
protected void Page_Load(object sender, EventArgs e)
{
    using (ShoppingCartActions usersShoppingCart = new ShoppingCartActions())
    {
        decimal cartTotal = 0;
        cartTotal = usersShoppingCart.GetTotal();
        if (cartTotal > 0)
        {
            // Display Total.
            lblTotal.Text = String.Format("{0:c}", cartTotal);
        }
        else
        {
            LabelTotalText.Text = "";
            lblTotal.Text = "";
            ShoppingCartTitle.InnerText = "Shopping Cart is Empty";
            UpdateBtn.Visible = false;
            CheckoutImageBtn.Visible = false;
        }
    }
}
```

6. Save your changes to both the *ShoppingCart.aspx* file and the *ShoppingCart.aspx.cs* file.

7. From the menu, select **Debug**->**Build WingtipToys**.
   The project will be rebuilt with the newly added **ImageButton** control.

Send Purchase Details to PayPal

When the user clicks the **Checkout** button on the shopping cart page (*ShoppingCart.aspx*), they'll begin the purchase process. The following code calls the first PayPal function needed to purchase products.

1. From the *Checkout* folder, open the code-behind file named *CheckoutStart.aspx.cs*.
   Be sure to open the code-behind file.

2. Replace the existing code with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace WingtipToys.Checkout
{
    public partial class CheckoutStart : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            NVPAPICaller payPalCaller = new NVPAPICaller();
            string retMsg = "";
            string token = "";

            if (Session["payment_amt"] != null)
            {
                string amt = Session["payment_amt"].ToString();

                bool ret = payPalCaller.ShortcutExpressCheckout(amt, ref token, ref retMsg);
                if (ret)
                {
                    Session["token"] = token;
                    Response.Redirect(retMsg);
                }
                else
                {
                    Response.Redirect("CheckoutError.aspx?" + retMsg);
                }
            }
            else
            {
                Response.Redirect("CheckoutError.aspx?ErrorCode=AmtMissing");
            }
        }
    }
}
```

When the user of the application clicks the **Checkout** button on the shopping cart page, the browser will navigate to the *CheckoutStart.aspx* page. When the *CheckoutStart.aspx* page loads, the `ShortcutExpressCheckout` method is called. At this point, the user is transferred to the PayPal testing web site. On the PayPal site, the user enters their PayPal credentials, reviews the purchase details, accepts the PayPal agreement and returns to the Wingtip Toys sample application where the `ShortcutExpressCheckout` method completes. When the `ShortcutExpressCheckout` method is complete, it will redirect the user to the *CheckoutReview.aspx* page specified in the `ShortcutExpressCheckout` method. This allows the user to review the order details from within the Wingtip Toys sample application.

Review Order Details

After returning from PayPal, the *CheckoutReview.aspx* page of the Wingtip Toys sample application displays the order details. This page allows the user to review the order details before purchasing the products. The *CheckoutReview.aspx* page must be created as follows:

1. In the *Checkout* folder, open the page named *CheckoutReview.aspx*.

2. Replace the existing markup with the following:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="CheckoutReview.aspx.cs" Inherits="WingtipToys.Checkout.CheckoutReview" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <h1>Order Review</h1>
    <p></p>
    <h3 style="padding-left: 33px">Products:</h3>
    <asp:GridView ID="OrderItemList" runat="server" AutoGenerateColumns="False" GridLines="Both"
CellPadding="10" Width="500" BorderColor="#efeeef" BorderWidth="33">
        <Columns>
            <asp:BoundField DataField="ProductId" HeaderText=" Product ID" />
            <asp:BoundField DataField="Product.ProductName" HeaderText=" Product Name" />
            <asp:BoundField DataField="Product.UnitPrice" HeaderText="Price (each)" DataFormatString="
{0:c}"/>
            <asp:BoundField DataField="Quantity" HeaderText="Quantity" />
        </Columns>
    </asp:GridView>
    <asp:DetailsView ID="ShipInfo" runat="server" AutoGenerateRows="false" GridLines="None"
CellPadding="10" BorderStyle="None" CommandRowStyle-BorderStyle="None">
        <Fields>
        <asp:TemplateField>
            <ItemTemplate>
                <h3>Shipping Address:</h3>
                <br />
                <asp:Label ID="FirstName" runat="server" Text='<%#: Eval("FirstName") %>'></asp:Label>
                <asp:Label ID="LastName" runat="server" Text='<%#: Eval("LastName") %>'></asp:Label>
                <br />
                <asp:Label ID="Address" runat="server" Text='<%#: Eval("Address") %>'></asp:Label>
                <br />
                <asp:Label ID="City" runat="server" Text='<%#: Eval("City") %>'></asp:Label>
                <asp:Label ID="State" runat="server" Text='<%#: Eval("State") %>'></asp:Label>
                <asp:Label ID="PostalCode" runat="server" Text='<%#: Eval("PostalCode") %>'>
</asp:Label>
                <p></p>
                <h3>Order Total:</h3>
                <br />
                <asp:Label ID="Total" runat="server" Text='<%#: Eval("Total", "{0:C}") %>'></asp:Label>
            </ItemTemplate>
            <ItemStyle HorizontalAlign="Left" />
        </asp:TemplateField>
          </Fields>
    </asp:DetailsView>
    <p></p>
    <hr />
    <asp:Button ID="CheckoutConfirm" runat="server" Text="Complete Order"
OnClick="CheckoutConfirm_Click" />
</asp:Content>
```

3. Open the code-behind page named *CheckoutReview.aspx.cs* and replace the existing code with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Models;

namespace WingtipToys.Checkout
{
  public partial class CheckoutReview : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      if (!IsPostBack)
      {
```

```
        {
          NVPAPICaller payPalCaller = new NVPAPICaller();

          string retMsg = "";
          string token = "";
          string PayerID = "";
          NVPCodec decoder = new NVPCodec();
          token = Session["token"].ToString();

          bool ret = payPalCaller.GetCheckoutDetails(token, ref PayerID, ref decoder, ref retMsg);
          if (ret)
          {
            Session["payerId"] = PayerID;

            var myOrder = new Order();
            myOrder.OrderDate = Convert.ToDateTime(decoder["TIMESTAMP"].ToString());
            myOrder.Username = User.Identity.Name;
            myOrder.FirstName = decoder["FIRSTNAME"].ToString();
            myOrder.LastName = decoder["LASTNAME"].ToString();
            myOrder.Address = decoder["SHIPTOSTREET"].ToString();
            myOrder.City = decoder["SHIPTOCITY"].ToString();
            myOrder.State = decoder["SHIPTOSTATE"].ToString();
            myOrder.PostalCode = decoder["SHIPTOZIP"].ToString();
            myOrder.Country = decoder["SHIPTOCOUNTRYCODE"].ToString();
            myOrder.Email = decoder["EMAIL"].ToString();
            myOrder.Total = Convert.ToDecimal(decoder["AMT"].ToString());

            // Verify total payment amount as set on CheckoutStart.aspx.
            try
            {
              decimal paymentAmountOnCheckout = Convert.ToDecimal(Session["payment_amt"].ToString());
              decimal paymentAmoutFromPayPal = Convert.ToDecimal(decoder["AMT"].ToString());
              if (paymentAmountOnCheckout != paymentAmoutFromPayPal)
              {
                Response.Redirect("CheckoutError.aspx?" + "Desc=Amount%20total%20mismatch.");
              }
            }
            catch (Exception)
            {
              Response.Redirect("CheckoutError.aspx?" + "Desc=Amount%20total%20mismatch.");
            }

            // Get DB context.
            ProductContext _db = new ProductContext();

            // Add order to DB.
            _db.Orders.Add(myOrder);
            _db.SaveChanges();

            // Get the shopping cart items and process them.
            using (WingtipToys.Logic.ShoppingCartActions usersShoppingCart = new
WingtipToys.Logic.ShoppingCartActions())
            {
              List<CartItem> myOrderList = usersShoppingCart.GetCartItems();

              // Add OrderDetail information to the DB for each product purchased.
              for (int i = 0; i < myOrderList.Count; i++)
              {
                // Create a new OrderDetail object.
                var myOrderDetail = new OrderDetail();
                myOrderDetail.OrderId = myOrder.OrderId;
                myOrderDetail.Username = User.Identity.Name;
                myOrderDetail.ProductId = myOrderList[i].ProductId;
                myOrderDetail.Quantity = myOrderList[i].Quantity;
                myOrderDetail.UnitPrice = myOrderList[i].Product.UnitPrice;

                // Add OrderDetail to DB.
                _db.OrderDetails.Add(myOrderDetail);
                _db.SaveChanges();
              }
```

```
            }

            // Set OrderId.
            Session["currentOrderId"] = myOrder.OrderId;

            // Display Order information.
            List<Order> orderList = new List<Order>();
            orderList.Add(myOrder);
            ShipInfo.DataSource = orderList;
            ShipInfo.DataBind();

            // Display OrderDetails.
            OrderItemList.DataSource = myOrderList;
            OrderItemList.DataBind();
          }
        }
        else
        {
          Response.Redirect("CheckoutError.aspx?" + retMsg);
        }
      }
    }

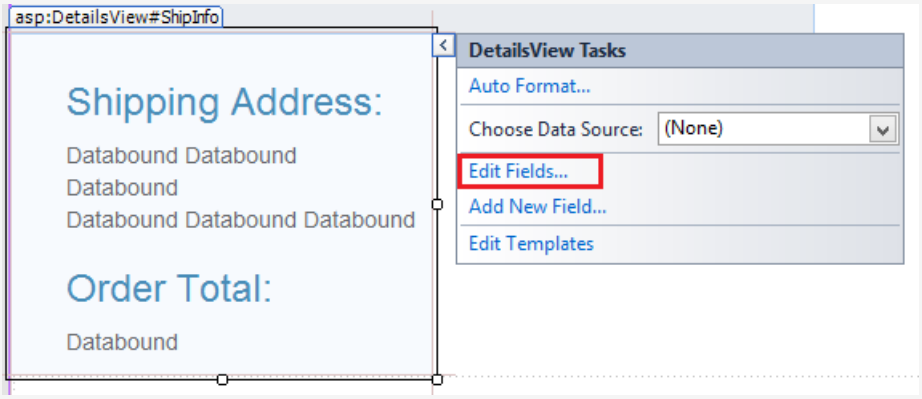
    protected void CheckoutConfirm_Click(object sender, EventArgs e)
    {
      Session["userCheckoutCompleted"] = "true";
      Response.Redirect("~/Checkout/CheckoutComplete.aspx");
    }
  }
}
```

The **DetailsView** control is used to display the order details that have been returned from PayPal. Also, the above code saves the order details to the Wingtip Toys database as an `OrderDetail` object. When the user clicks on the **Complete Order** button, they are redirected to the *CheckoutComplete.aspx* page.

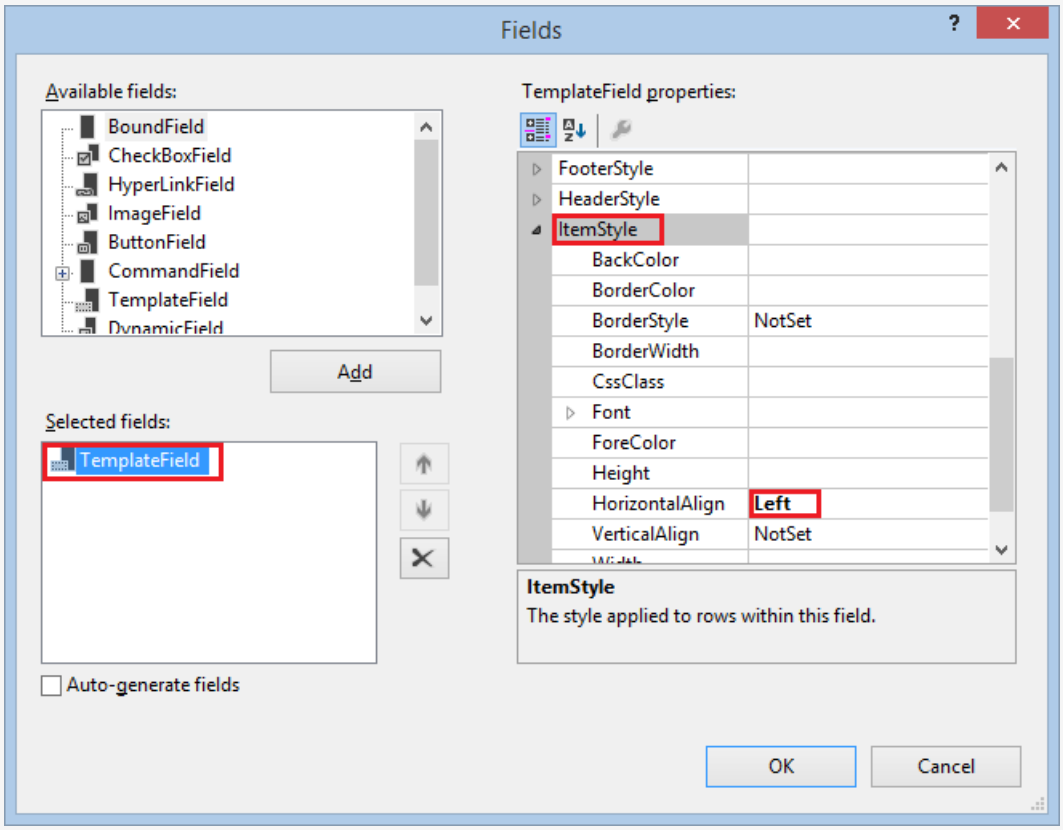
NOTE

Tip

In the markup of the *CheckoutReview.aspx* page, notice that the `<ItemStyle>` tag is used to change the style of the items within the **DetailsView** control near the bottom of the page. By viewing the page in **Design View** (by selecting **Design** at the lower left corner of Visual Studio), then selecting the **DetailsView** control, and selecting the **Smart Tag** (the arrow icon at the top right of the control), you will be able to see the **DetailsView Tasks**.



By selecting **Edit Fields**, the **Fields** dialog box will appear. In this dialog box you can easily control the visual properties, such as **ItemStyle**, of the **DetailsView** control.



Complete Purchase

CheckoutComplete.aspx page makes the purchase from PayPal. As mentioned above, the user must click on the **Complete Order** button before the application will navigate to the *CheckoutComplete.aspx* page.

1. In the *Checkout* folder, open the page named *CheckoutComplete.aspx*.

2. Replace the existing markup with the following:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="CheckoutComplete.aspx.cs" Inherits="WingtipToys.Checkout.CheckoutComplete" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <h1>Checkout Complete</h1>
    <p></p>
    <h3>Payment Transaction ID:</h3> <asp:Label ID="TransactionId" runat="server"></asp:Label>
    <p></p>
    <h3>Thank You!</h3>
    <p></p>
    <hr />
    <asp:Button ID="Continue" runat="server" Text="Continue Shopping" OnClick="Continue_Click" />
</asp:Content>
```

3. Open the code-behind page named *CheckoutComplete.aspx.cs* and replace the existing code with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Models;

namespace WingtipToys.Checkout
{
  public partial class CheckoutComplete : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      if (!IsPostBack)
      {
        // Verify user has completed the checkout process.
        if ((string)Session["userCheckoutCompleted"] != "true")
        {
          Session["userCheckoutCompleted"] = string.Empty;
          Response.Redirect("CheckoutError.aspx?" + "Desc=Unvalidated%20Checkout.");
        }

        NVPAPICaller payPalCaller = new NVPAPICaller();

        string retMsg = "";
        string token = "";
        string finalPaymentAmount = "";
        string PayerID = "";
        NVPCodec decoder = new NVPCodec();

        token = Session["token"].ToString();
        PayerID = Session["payerId"].ToString();
        finalPaymentAmount = Session["payment_amt"].ToString();

        bool ret = payPalCaller.DoCheckoutPayment(finalPaymentAmount, token, PayerID, ref decoder, ref
retMsg);
        if (ret)
        {
          // Retrieve PayPal confirmation value.
          string PaymentConfirmation = decoder["PAYMENTINFO_0_TRANSACTIONID"].ToString();
          TransactionId.Text = PaymentConfirmation;

          ProductContext _db = new ProductContext();
          // Get the current order id.
          int currentOrderId = -1;
          if (Session["currentOrderId"] != string.Empty)
          {
            currentOrderId = Convert.ToInt32(Session["currentOrderID"]);
          }
```

```
            }
            Order myCurrentOrder;
            if (currentOrderId >= 0)
            {
              // Get the order based on order id.
              myCurrentOrder = _db.Orders.Single(o => o.OrderId == currentOrderId);
              // Update the order to reflect payment has been completed.
              myCurrentOrder.PaymentTransactionId = PaymentConfirmation;
              // Save to DB.
              _db.SaveChanges();
            }

            // Clear shopping cart.
            using (WingtipToys.Logic.ShoppingCartActions usersShoppingCart =
                new WingtipToys.Logic.ShoppingCartActions())
            {
              usersShoppingCart.EmptyCart();
            }

            // Clear order id.
            Session["currentOrderId"] = string.Empty;
          }
          else
          {
            Response.Redirect("CheckoutError.aspx?" + retMsg);
          }
        }
      }

      protected void Continue_Click(object sender, EventArgs e)
      {
        Response.Redirect("~/Default.aspx");
      }
    }
  }
```

When the *CheckoutComplete.aspx* page is loaded, the `DoCheckoutPayment` method is called. As mentioned earlier, the `DoCheckoutPayment` method completes the purchase from the PayPal testing environment. Once PayPal has completed the purchase of the order, the *CheckoutComplete.aspx* page displays a payment transaction `ID` to the purchaser.

Handle Cancel Purchase

If the user decides to cancel the purchase, they will be directed to the *CheckoutCancel.aspx* page where they will see that their order has been cancelled.

1. Open the page named *CheckoutCancel.aspx* in the *Checkout* folder.
2. Replace the existing markup with the following:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="CheckoutCancel.aspx.cs" Inherits="WingtipToys.Checkout.CheckoutCancel" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <h1>Checkout Cancelled</h1>
    <p></p>
    <h3>Your purchase has been cancelled.</h3>
</asp:Content>
```

Handle Purchase Errors

Errors during the purchase process will be handled by the *CheckoutError.aspx* page. The code-behind of the *CheckoutStart.aspx* page, the *CheckoutReview.aspx* page, and the *CheckoutComplete.aspx* page will each redirect to the *CheckoutError.aspx* page if an error occurs.

1. Open the page named *CheckoutError.aspx* in the *Checkout* folder.

2. Replace the existing markup with the following:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="CheckoutError.aspx.cs" Inherits="WingtipToys.Checkout.CheckoutError" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <h1>Checkout Error</h1>
    <p></p>
<table id="ErrorTable">
    <tr>
        <td class="field"></td>
        <td><%=Request.QueryString.Get("ErrorCode")%></td>
    </tr>
    <tr>
        <td class="field"></td>
        <td><%=Request.QueryString.Get("Desc")%></td>
    </tr>
    <tr>
        <td class="field"></td>
        <td><%=Request.QueryString.Get("Desc2")%></td>
    </tr>
</table>
    <p></p>
</asp:Content>
```

The *CheckoutError.aspx* page is displayed with the error details when an error occurs during the checkout process.

Running the Application

Run the application to see how to purchase products. Note that you will be running in the PayPal testing environment. No actual money is being exchanged.

1. Make sure all your files are saved in Visual Studio.

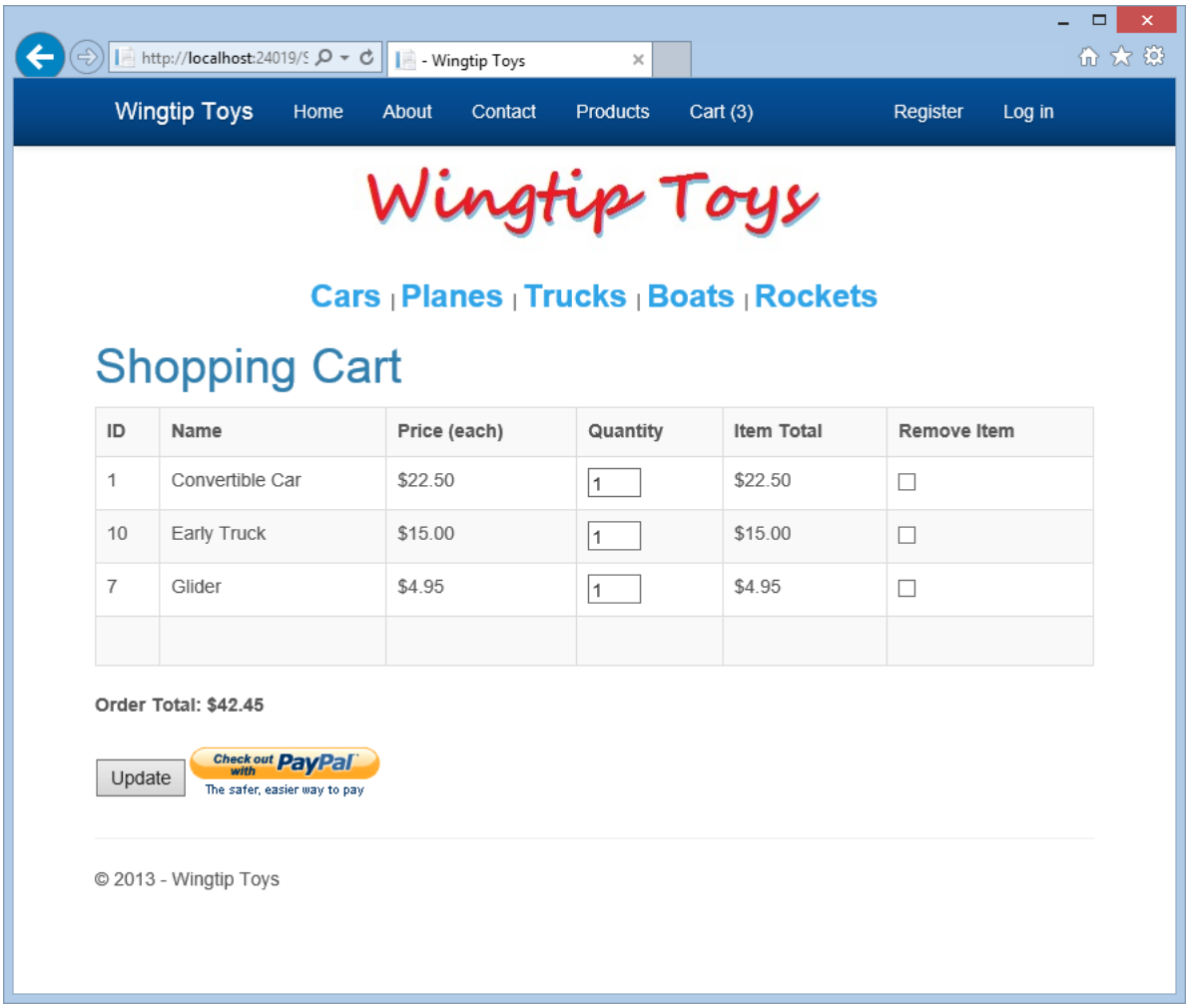
2. Open a Web browser and navigate to https://developer.paypal.com.

3. Login with your PayPal developer account that you created earlier in this tutorial.
   For PayPal's developer sandbox, you need to be logged in at https://developer.paypal.com to test express checkout. This only applies to PayPal's sandbox testing, not to PayPal's live environment.

4. In Visual Studio, press **F5** to run the Wingtip Toys sample application.
   After the database rebuilds, the browser will open and show the *Default.aspx* page.

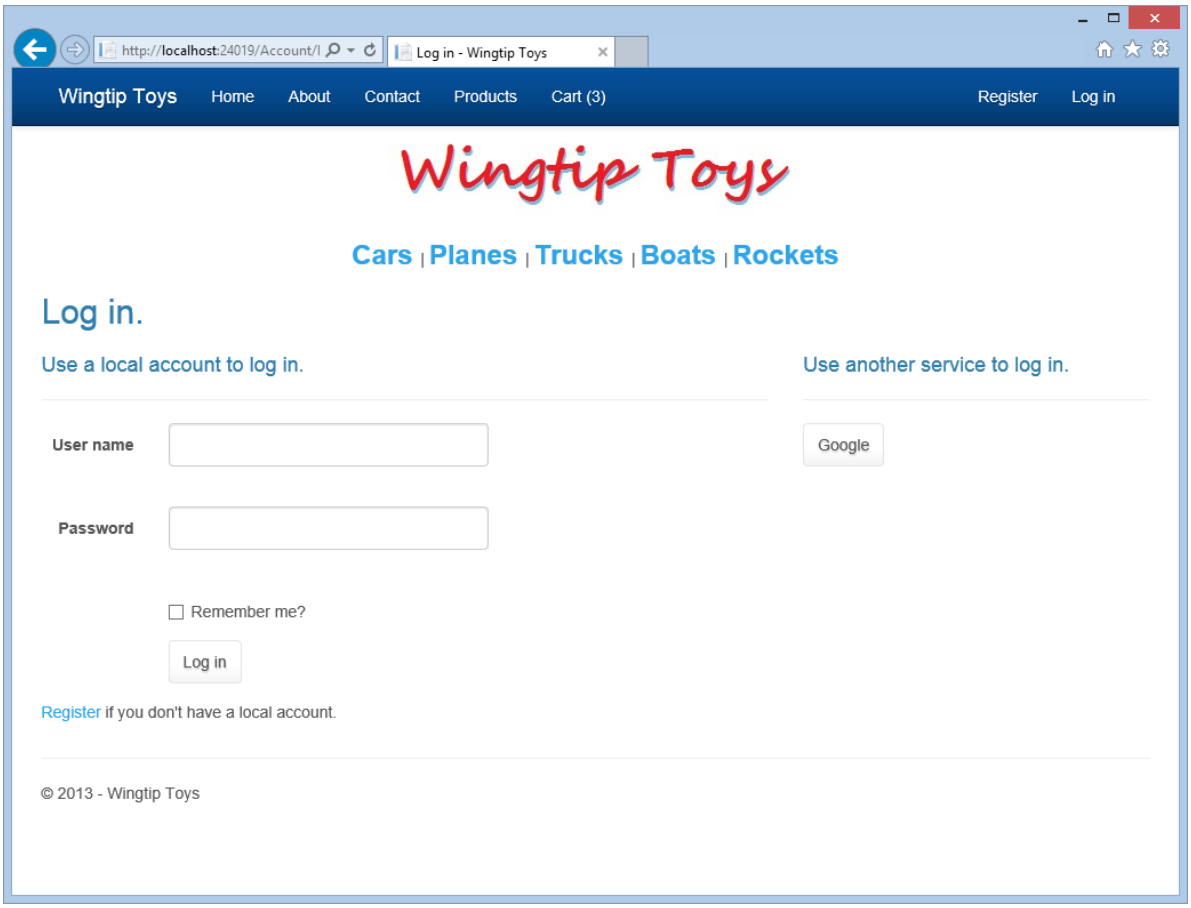
5. Add three different products to the shopping cart by selecting the product category, such as "Cars" and then clicking **Add to Cart** next to each product.
   The shopping cart will display the product you have selected.

6. Click the **PayPal** button to checkout.

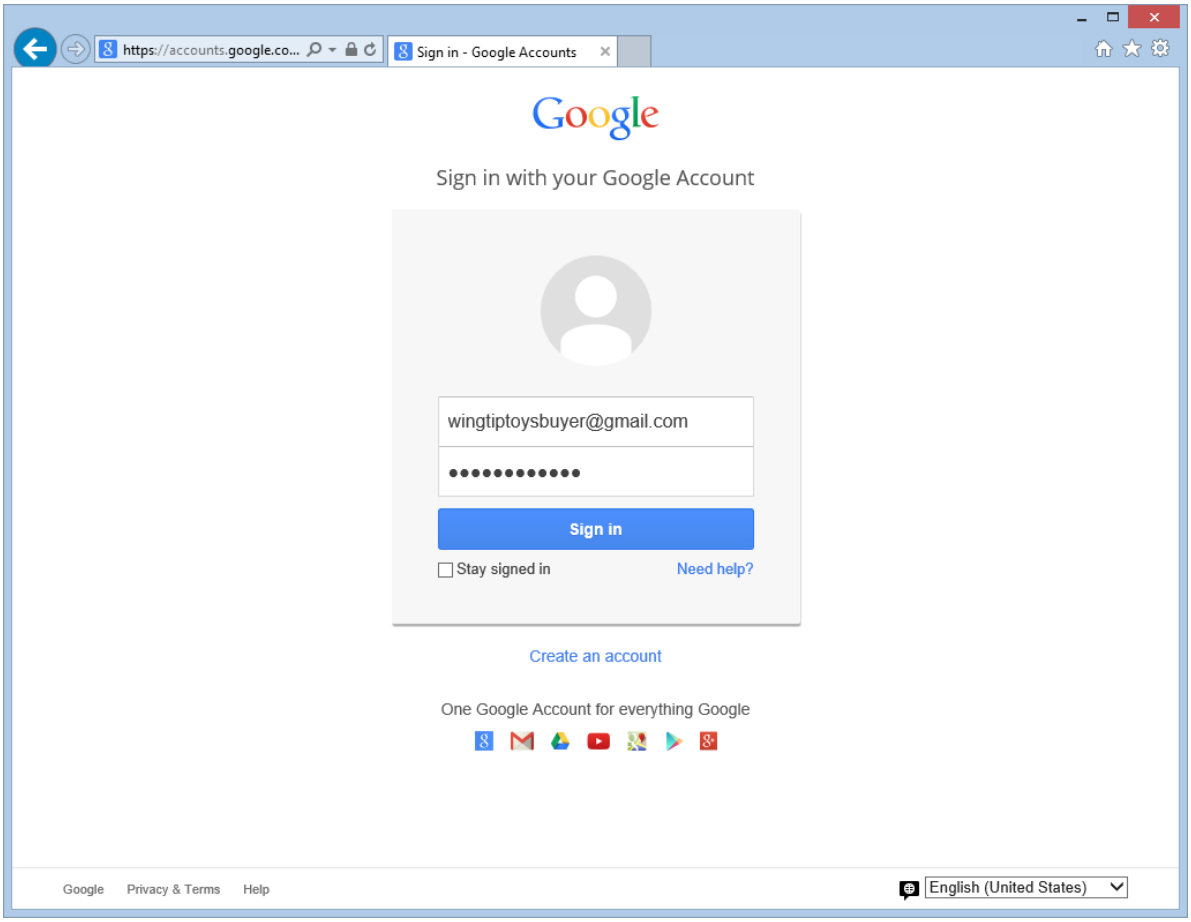


Checking out will require that you have a user account for the Wingtip Toys sample application.

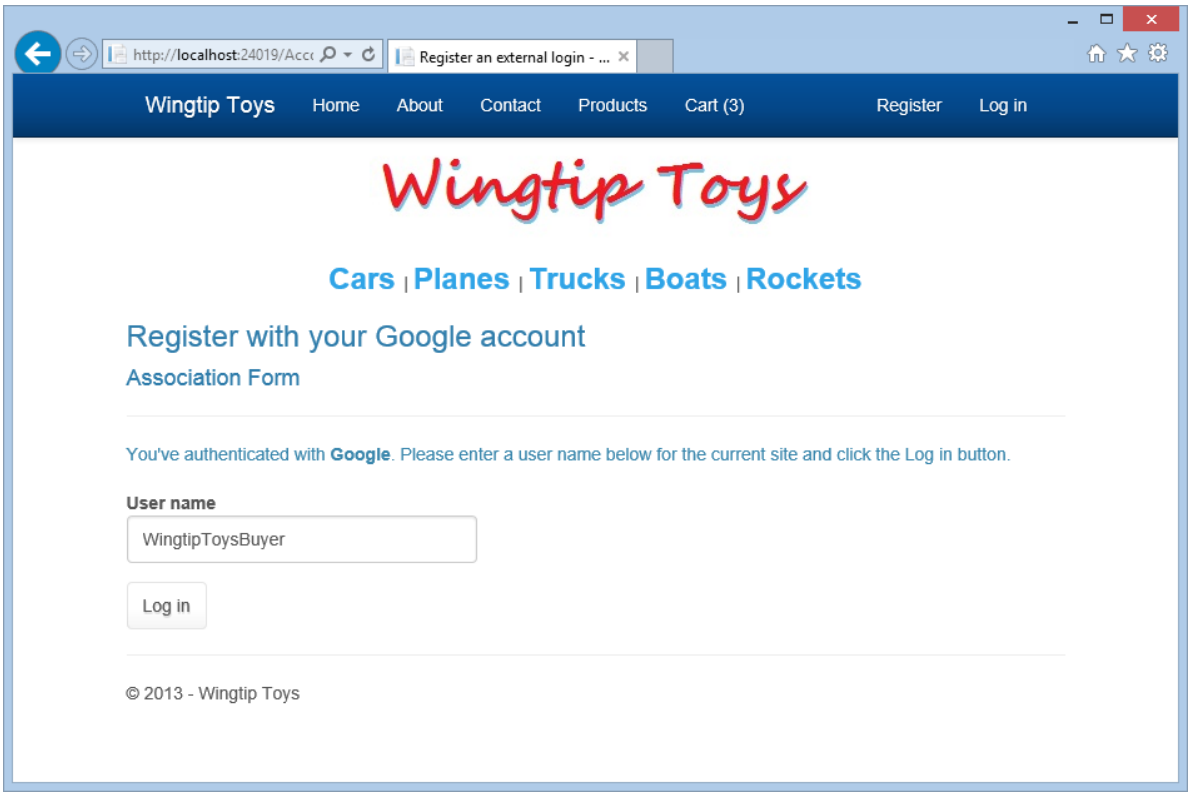
7. Click the **Google** link on the right of the page to log in with an existing gmail.com email account. If you do not have a gmail.com account, you can create one for testing purposes at www.gmail.com. You can also use a standard local account by clicking "Register".



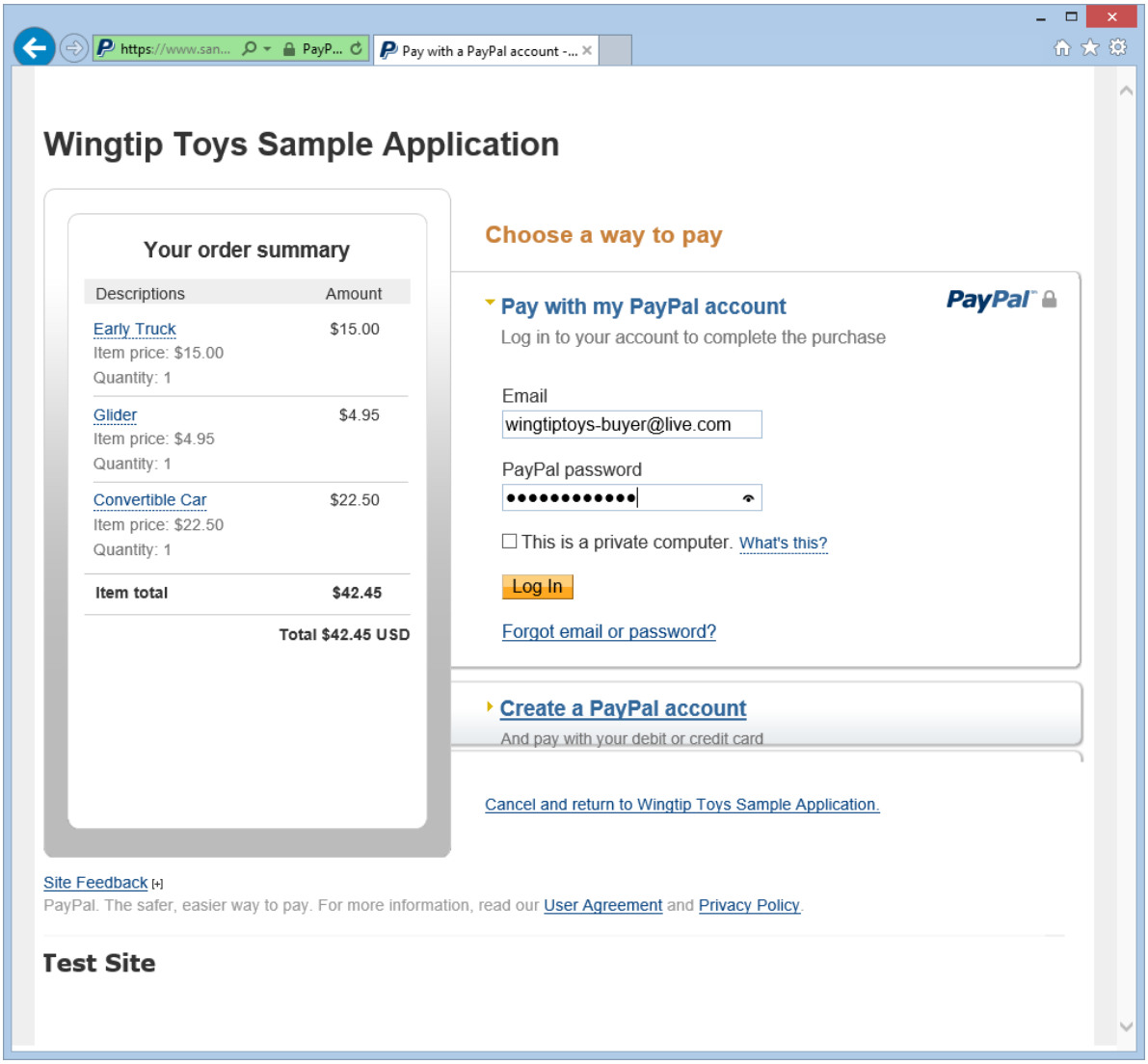
8. Sign in with your gmail account and password.



9. Click the **Log in** button to register your gmail account with your Wingtip Toys sample application user name.

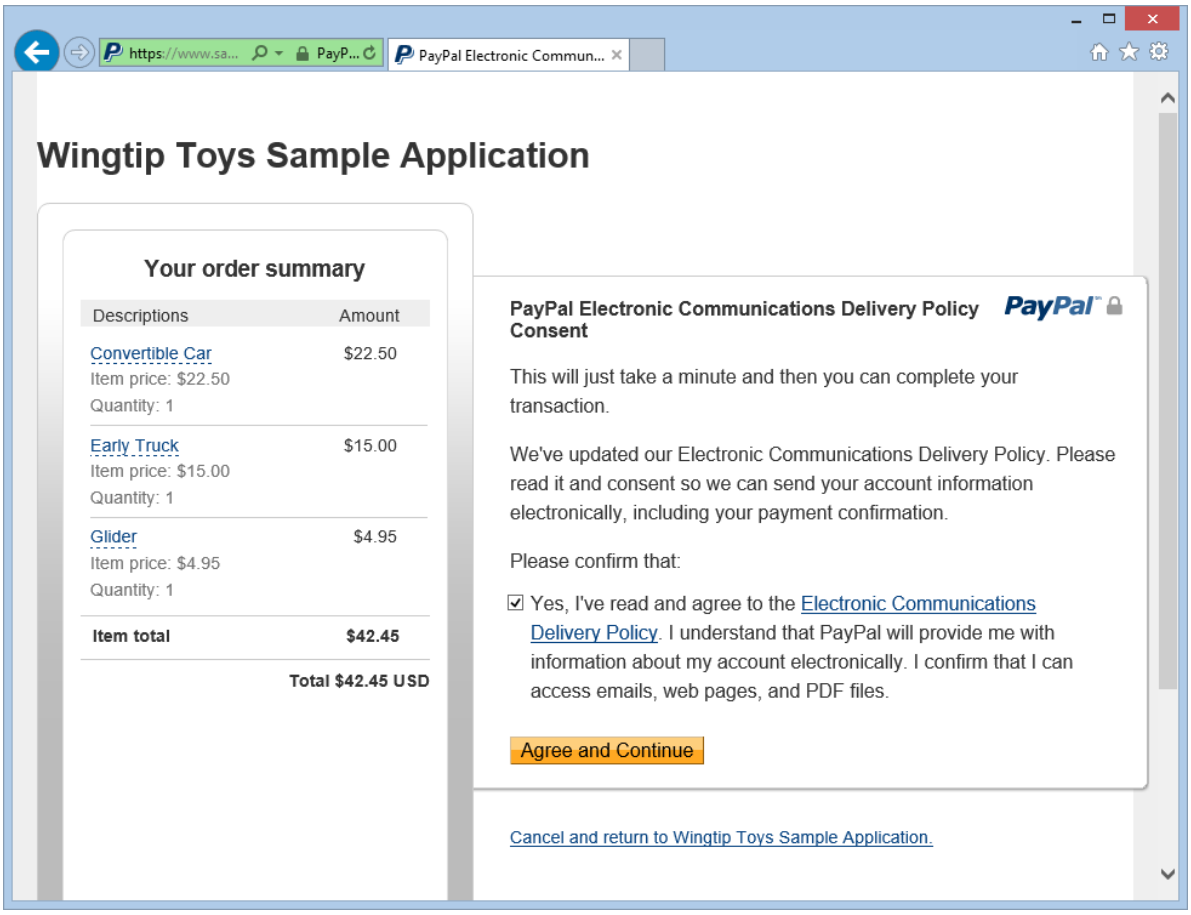


10. On the PayPal test site, add your **buyer** email address and password that you created earlier in this tutorial, then click the **Log In** button.

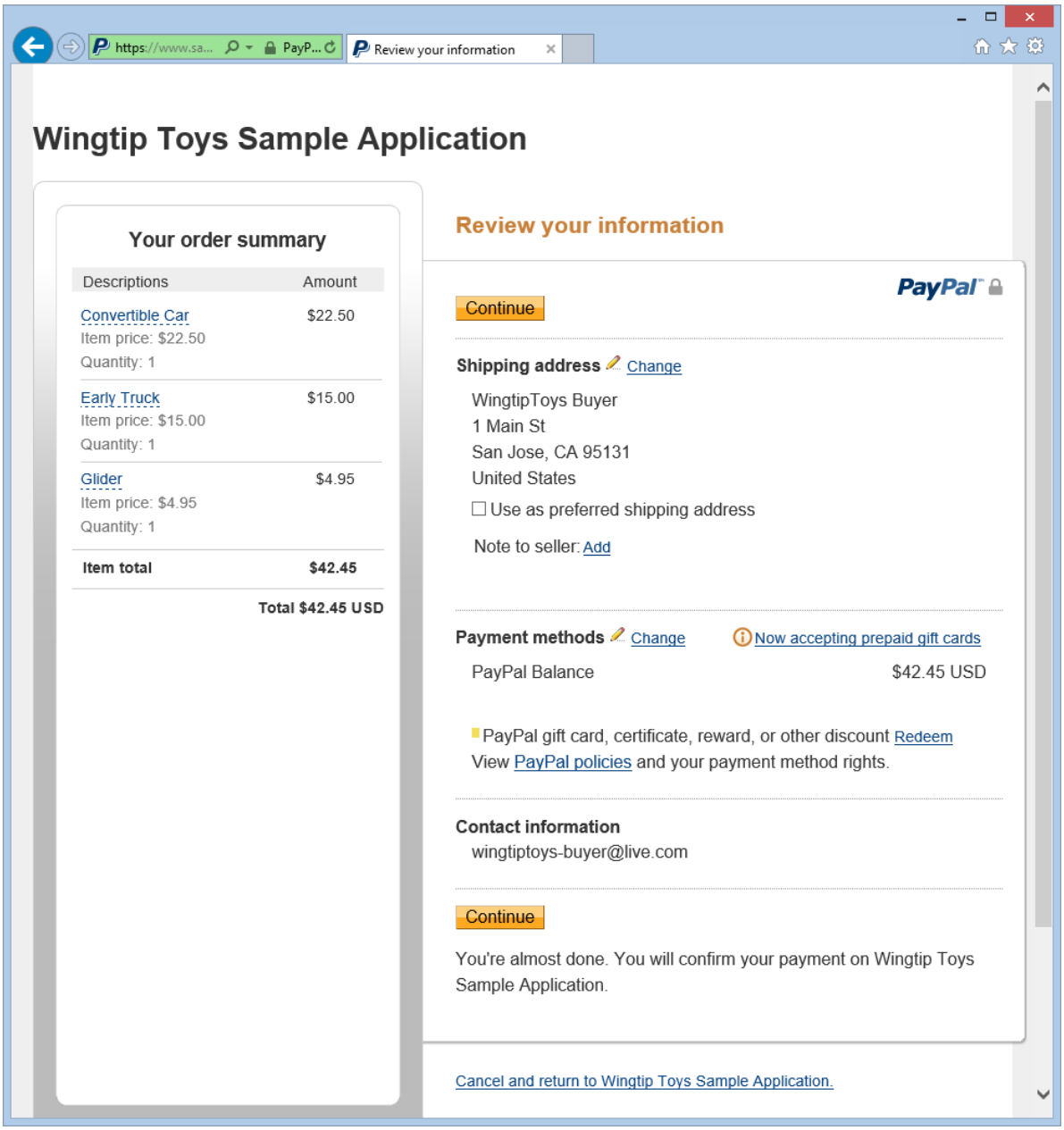


11. Agree to the PayPal policy and click the **Agree and Continue** button.

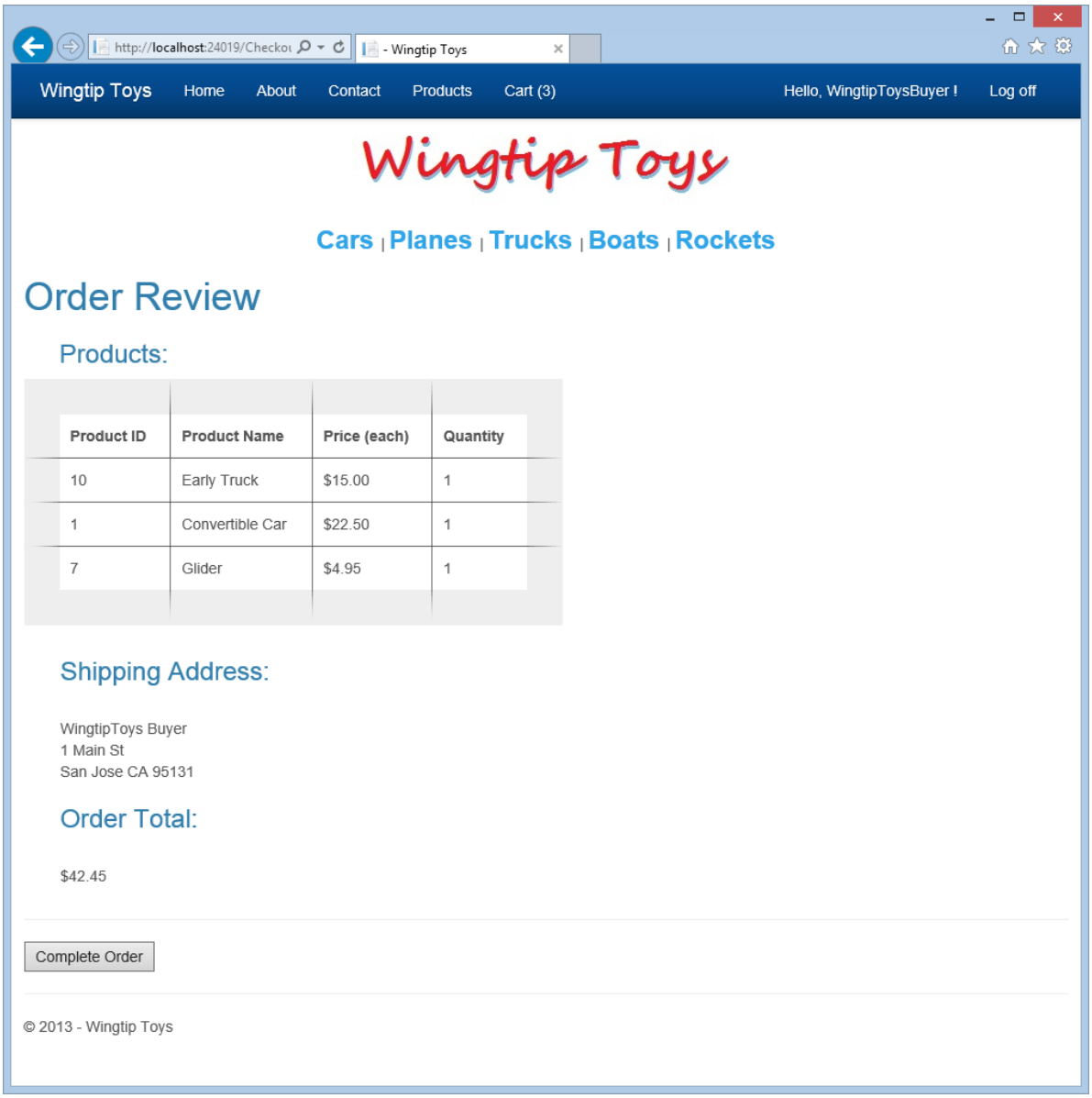
Note that this page is only displayed the first time you use this PayPal account. Again note that this is a test account, no real money is exchanged.



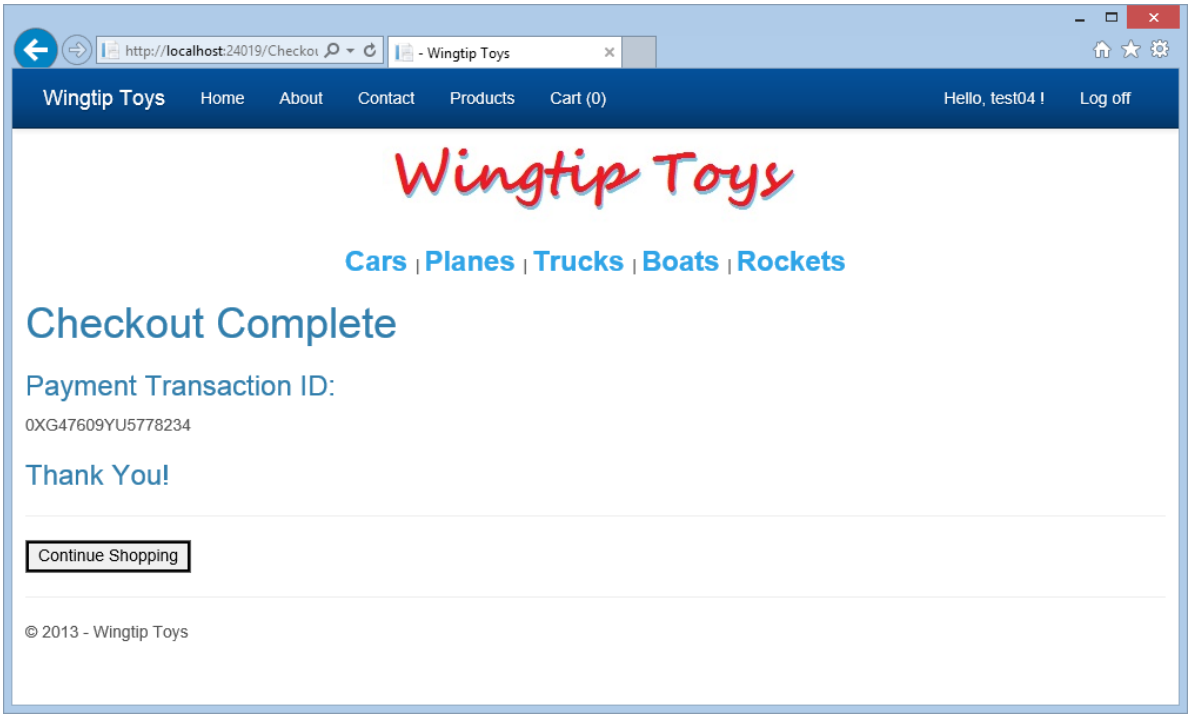
12. Review the order information on the PayPal testing environment review page and click **Continue**.



13. On the *CheckoutReview.aspx* page, verify the order amount and view the generated shipping address. Then, click the **Complete Order** button.



14. The **CheckoutComplete.aspx** page is displayed with a payment transaction ID.

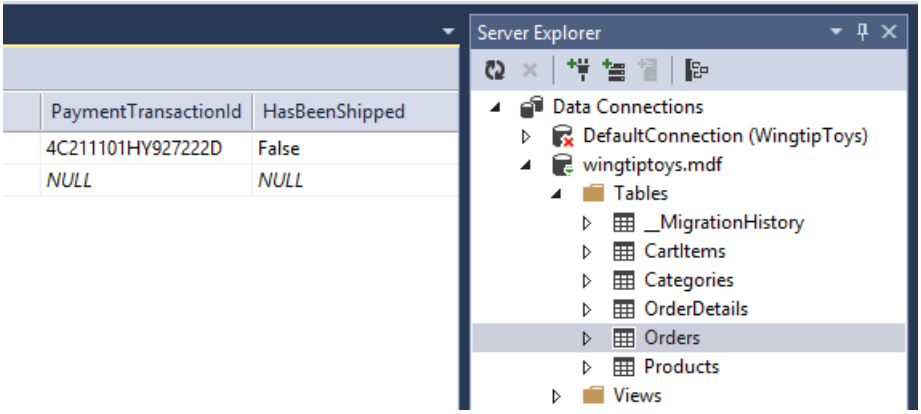


Reviewing the Database

By reviewing the updated data in the Wingtip Toys sample application database after running the application, you can see that the application successfully recorded the purchase of the products.

You can inspect the data contained in the *Wingtiptoys.mdf* database file by using the **Database Explorer** window (**Server Explorer** window in Visual Studio) as you did earlier in this tutorial series.

1. Close the browser window if it is still open.
2. In Visual Studio, select the **Show All Files** icon at the top of **Solution Explorer** to allow you to expand the **App_Data** folder.
3. Expand the **App_Data** folder.
   You may need to select the **Show All Files** icon for the folder.
4. Right-click the *Wingtiptoys.mdf* database file and select **Open**.
   Server Explorer is displayed.
5. Expand the **Tables** folder.
6. Right-click the **Orders** table and select **Show Table Data**.
   The **Orders** table is displayed.
7. Review the **PaymentTransactionID** column to confirm successful transactions.



8. Close the **Orders** table window.
9. In the Server Explorer, right-click the **OrderDetails** table and select **Show Table Data**.
10. Review the `OrderId` and `Username` values in the **OrderDetails** table. Note that these values match the `OrderId` and `Username` values included in the **Orders** table.
11. Close the **OrderDetails** table window.
12. Right-click the Wingtip Toys database file (*Wingtiptoys.mdf*) and select **Close Connection**.
13. If you do not see the **Solution Explorer** window, click **Solution Explorer** at the bottom of the **Server Explorer** window to show the **Solution Explorer** again.

Summary

In this tutorial you added order and order detail schemas to track the purchase of products. You also integrated PayPal functionality into the Wingtip Toys sample application.

Additional Resources

ASP.NET Configuration Overview

Disclaimer

This tutorial contains sample code. Such sample code is provided "as is" without warranty of any kind. Accordingly, Microsoft does not guarantee the accuracy, integrity, or quality of the sample code. You agree to use the sample code at your own risk. Under no circumstances will Microsoft be liable to you in any way for any sample code, content, including but not limited to, any errors or omissions in any sample code, content, or any loss or damage of any kind incurred as a result of the use of any sample code. You are hereby notified and do hereby agree to indemnify, save and hold Microsoft harmless from and against any and all loss, claims of loss, injury or damage of any kind including, without limitation, those occasioned by or arising from material that you post, transmit, use or rely on including, but not limited to, the views expressed therein.

Membership and Administration

2/20/2020 • 19 minutes to read • Edit Online

by Erik Reitan

Download Wingtip Toys Sample Project (C#) or Download E-book (PDF)

> This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

This tutorial shows you how to update the Wingtip Toys sample application to add a custom role and use ASP.NET Identity. It also shows you how to implement an administration page from which the user with a custom role can add and remove products from the website.

ASP.NET Identity is the membership system used to build ASP.NET web application and is available in ASP.NET 4.5. ASP.NET Identity is used in the Visual Studio 2013 Web Forms project template, as well as the templates for ASP.NET MVC, ASP.NET Web API, and ASP.NET Single Page Application. You can also specifically install the ASP.NET Identity system using NuGet when you start with an empty Web application. However, in this tutorial series you use the **Web Forms** projecttemplate, which includes the ASP.NET Identity system. ASP.NET Identity makes it easy to integrate user-specific profile data with application data. Also, ASP.NET Identity allows you to choose the persistence model for user profiles in your application. You can store the data in a SQL Server database or another data store, including *NoSQL* data stores such as Windows Azure Storage Tables.

This tutorial builds on the previous tutorial titled "Checkout and Payment with PayPal" in the Wingtip Toys tutorial series.

What you'll learn:

- How to use code to add a custom role and a user to the application.
- How to restrict access to the administration folder and page.
- How to provide navigation for the user that belongs to the custom role.
- How to use model binding to populate a DropDownList control with product categories.
- How to upload a file to the web application using the FileUpload control.
- How to use validation controls to implement input validation.
- How to add and remove products from the application.

These features are included in the tutorial:

- ASP.NET Identity
- Configuration and Authorization
- Model Binding
- Unobtrusive Validation

ASP.NET Web Forms provides membership capabilities. By using the default template, you have built-in membership functionality that you can immediately use when the application runs. This tutorial shows you how to use ASP.NET Identity to add a custom role and assign a user to that role. You will learn how to restrict access to the administration folder. You'll add a page to the administration folder that allows a user with a custom role to add and remove products, and to preview a product after it has been added.

Adding a Custom Role

Using ASP.NET Identity, you can add a custom role and assign a user to that role using code.

1. In **Solution Explorer**, right-click on the *Logic* folder and create a new class.
2. Name the new class *RoleActions.cs*.
3. Modify the code so that it appears as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace WingtipToys.Logic
{
    internal class RoleActions
    {
    }
}
```

4. In **Solution Explorer**, open the *Global.asax.cs* file.
5. Modify the *Global.asax.cs* file by adding the code highlighted in yellow so that it appears as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Optimization;
using System.Web.Routing;
using System.Web.Security;
using System.Web.SessionState;
using System.Data.Entity;
using WingtipToys.Models;
using WingtipToys.Logic;

namespace WingtipToys
{
    public class Global : HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
          // Code that runs on application startup
          RouteConfig.RegisterRoutes(RouteTable.Routes);
          BundleConfig.RegisterBundles(BundleTable.Bundles);

          // Initialize the product database.
          Database.SetInitializer(new ProductDatabaseInitializer());

          // Create the custom role and user.
          RoleActions roleActions = new RoleActions();
          roleActions.AddUserAndRole();
        }
    }
}
```

6. Notice that `AddUserAndRole` is underlined in red. Double-click the AddUserAndRole code.
   The letter "A" at the beginning of the highlighted method will be underlined.
7. Hover over the letter "A" and click the UI that allows you to generate a method stub for the `AddUserAndRole` method.

```
        // Create the administrator role and user.
        RoleActions roleActions = new RoleActions();
        roleActions.AddUserAndRole();
    }
```

Generate method stub for 'AddUserAndRole' in 'WingtipToys.Logic.RoleActions'

8. Click the option titled:
   `Generate method stub for "AddUserAndRole" in "WingtipToys.Logic.RoleActions"`

9. Open the *RoleActions.cs* file from the *Logic* folder.
   The `AddUserAndRole` method has been added to the class file.

10. Modify the *RoleActions.cs* file by removing the `NotImplementedException` and adding the code highlighted in yellow, so that it appears as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using WingtipToys.Models;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;

namespace WingtipToys.Logic
{
  internal class RoleActions
  {
    internal void AddUserAndRole()
    {
      // Access the application context and create result variables.
      Models.ApplicationDbContext context = new ApplicationDbContext();
      IdentityResult IdRoleResult;
      IdentityResult IdUserResult;

      // Create a RoleStore object by using the ApplicationDbContext object.
      // The RoleStore is only allowed to contain IdentityRole objects.
      var roleStore = new RoleStore<IdentityRole>(context);

      // Create a RoleManager object that is only allowed to contain IdentityRole objects.
      // When creating the RoleManager object, you pass in (as a parameter) a new RoleStore object.
      var roleMgr = new RoleManager<IdentityRole>(roleStore);

      // Then, you create the "canEdit" role if it doesn't already exist.
      if (!roleMgr.RoleExists("canEdit"))
      {
        IdRoleResult = roleMgr.Create(new IdentityRole { Name = "canEdit" });
      }

      // Create a UserManager object based on the UserStore object and the ApplicationDbContext
      // object. Note that you can create new objects and use them as parameters in
      // a single line of code, rather than using multiple lines of code, as you did
      // for the RoleManager object.
      var userMgr = new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(context));
      var appUser = new ApplicationUser
      {
        UserName = "canEditUser@wingtiptoys.com",
        Email = "canEditUser@wingtiptoys.com"
      };
      IdUserResult = userMgr.Create(appUser, "Pa$$word1");

      // If the new "canEdit" user was successfully created,
      // add the "canEdit" user to the "canEdit" role.
      if (!userMgr.IsInRole(userMgr.FindByEmail("canEditUser@wingtiptoys.com").Id, "canEdit"))
      {
        IdUserResult = userMgr.AddToRole(userMgr.FindByEmail("canEditUser@wingtiptoys.com").Id,
"canEdit");
      }
    }
  }
}
```

The above code first establishes a database context for the membership database. The membership database is also stored as an *.mdf* file in the *App_Data* folder. You will be able to view this database once the first user has signed in to this web application.

> **NOTE**
> If you wish to store the membership data along with the product data, you can consider using the same **DbContext** that you used to store the product data in the above code.

The *internal* keyword is an access modifier for types (such as classes) and type members (such as methods or properties). Internal types or members are accessible only within files contained in the same assembly *(.dll* file). When you build your application, an assembly file *(.dll)* is created that contains the code that is executed when you run your application.

A `RoleStore` object, which provides role management, is created based on the database context.

> **NOTE**
> Notice that when the `RoleStore` object is created it uses a Generic `IdentityRole` type. This means that the `RoleStore` is only allowed to contain `IdentityRole` objects. Also by using Generics, resources in memory are handled better.

Next, the `RoleManager` object, is created based on the `RoleStore` object that you just created. the `RoleManager` object exposes role related API which can be used to automatically save changes to the `RoleStore`. The `RoleManager` is only allowed to contain `IdentityRole` objects because the code uses the `<IdentityRole>` Generic type.

You call the `RoleExists` method to determine if the "canEdit" role is present in the membership database. If it is not, you create the role.

Creating the `UserManager` object appears to be more complicated than the `RoleManager` control, however it is nearly the same. It is just coded on one line rather than several. Here, the parameter that you are passing is instantiating as a new object contained in the parenthesis.

Next you create the "canEditUser" user by creating a new `ApplicationUser` object. Then, if you successfully create the user, you add the user to the new role.

> **NOTE**
> The error handling will be updated during the "ASP.NET Error Handling" tutorial later in this tutorial series.

The next time the application starts, the user named "canEditUser" will be added as the role named "canEdit" of the application. Later in this tutorial, you will login as the "canEditUser" user to display additional capabilities that you will added during this tutorial. For API details about ASP.NET Identity, see the Microsoft.AspNet.Identity Namespace. For additional details about initializing the ASP.NET Identity system, see the AspnetIdentitySample.

Restricting Access to the Administration Page

The Wingtip Toys sample application allows both anonymous users and logged-in users to view and purchase products. However, the logged-in user that has the custom "canEdit" role can access a restricted page in order to add and remove products.

Add an Administration Folder and Page

Next, you will create a folder named *Admin* for the "canEditUser" user belonging to the custom role of the Wingtip Toys sample application.

1. Right-click the project name (**Wingtip Toys**) in **Solution Explorer** and select **Add** -> **New Folder**.
2. Name the new folder *Admin*.
3. Right-click the *Admin* folder and then select **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.

4. Select the **Visual C#-> Web** templates group on the left. From the middle list, select **Web Form with Master Page**,name it *AdminPage.aspx*, and then select **Add**.
5. Select the *Site.Master* file as the master page, and then choose **OK**.

Add a Web.config File

By adding a *Web.config* file to the *Admin* folder, you can restrict access to the page contained in the folder.

1. Right-click the *Admin* folder and select **Add -> New Item**.
   The **Add New Item** dialog box is displayed.

2. From the list of Visual C# web templates, select **Web Configuration File**from the middle list, accept the default name of *Web.config*, and then select **Add**.

3. Replace the existing XML content in the *Web.config* file with the following:

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <authorization>
      <allow roles="canEdit"/>
      <deny users="*"/>
    </authorization>
  </system.web>
</configuration>
```

Save the *Web.config* file. The *Web.config* file specifies that only user belonging to the "canEdit" role of the application can access the page contained in the *Admin* folder.

Including Custom Role Navigation

To enable the user of the custom "canEdit" role to navigate to the administration section of the application, you must add a link to the *Site.Master* page. Only users that belong to the "canEdit" role will be able to see the **Admin** link and access the administration section.

1. In Solution Explorer, find and open the *Site.Master* page.

2. To create a link for the user of the "canEdit" role, add the markup highlighted in yellow to the following unordered list `<ul>` element so that the list appears as follows:

```
<ul class="nav navbar-nav">
    <li><a runat="server" id="adminLink" visible="false"
      href="~/Admin/AdminPage">Admin</a></li>
    <li><a runat="server" href="~/">Home</a></li>
    <li><a runat="server" href="~/About">About</a></li>
    <li><a runat="server" href="~/Contact">Contact</a></li>
    <li><a runat="server" href="~/ProductList">Products</a></li>
    <li><a runat="server" href="~/ShoppingCart"
      ID="cartCount"> </a></li>
</ul>
```

3. Open the *Site.Master.cs* file. Make the **Admin** link visible only to the "canEditUser" user by adding the code highlighted in yellow to the `Page_Load` handler. The `Page_Load` handler will appear as follows:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (HttpContext.Current.User.IsInRole("canEdit"))
    {
        adminLink.Visible = true;
    }
}
```

When the page loads, the code checks whether the logged-in user has the role of "canEdit". If the user belongs to the "canEdit" role, the span element containing the link to the *AdminPage.aspx* page (and consequently the link inside the span) is made visible.

Enabling Product Administration

So far, you have created the "canEdit" role and added an "canEditUser" user, an administration folder, and an administration page. You have set access rights for the administration folder and page, and have added a navigation link for the user of the "canEdit" role to the application. Next, you will add markup to the *AdminPage.aspx* page and code to the *AdminPage.aspx.cs* code-behind file that will enable the user with the "canEdit" role to add and remove products.

1. In **Solution Explorer**, open the *AdminPage.aspx* file from the *Admin* folder.
2. Replace the existing markup with the following:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="AdminPage.aspx.cs" Inherits="WingtipToys.Admin.AdminPage" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <h1>Administration</h1>
    <hr />
    <h3>Add Product:</h3>
    <table>
        <tr>
            <td><asp:Label ID="LabelAddCategory" runat="server">Category:</asp:Label></td>
            <td>
                <asp:DropDownList ID="DropDownAddCategory" runat="server"
                    ItemType="WingtipToys.Models.Category"
                    SelectMethod="GetCategories" DataTextField="CategoryName"
                    DataValueField="CategoryID" >
                </asp:DropDownList>
            </td>
        </tr>
        <tr>
            <td><asp:Label ID="LabelAddName" runat="server">Name:</asp:Label></td>
            <td>
                <asp:TextBox ID="AddProductName" runat="server"></asp:TextBox>
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server" Text="* Product
name required." ControlToValidate="AddProductName" SetFocusOnError="true" Display="Dynamic">
</asp:RequiredFieldValidator>
            </td>
        </tr>
        <tr>
            <td><asp:Label ID="LabelAddDescription" runat="server">Description:</asp:Label></td>
            <td>
                <asp:TextBox ID="AddProductDescription" runat="server"></asp:TextBox>
                <asp:RequiredFieldValidator ID="RequiredFieldValidator2" runat="server" Text="*
Description required." ControlToValidate="AddProductDescription" SetFocusOnError="true"
Display="Dynamic"></asp:RequiredFieldValidator>
            </td>
        </tr>
        <tr>
            <td><asp:Label ID="LabelAddPrice" runat="server">Price:</asp:Label></td>
            <td>
                <asp:TextBox ID="AddProductPrice" runat="server"></asp:TextBox>
                <asp:RequiredFieldValidator ID="RequiredFieldValidator3" runat="server" Text="* Price
```

```
required." ControlToValidate="AddProductPrice" SetFocusOnError="true" Display="Dynamic">
</asp:RequiredFieldValidator>
                <asp:RegularExpressionValidator ID="RegularExpressionValidator1" runat="server" Text="*
Must be a valid price without $." ControlToValidate="AddProductPrice" SetFocusOnError="True"
Display="Dynamic" ValidationExpression="^[0-9]*(\.)?[0-9]?[0-9]?$"></asp:RegularExpressionValidator>
            </td>
        </tr>
        <tr>
            <td><asp:Label ID="LabelAddImageFile" runat="server">Image File:</asp:Label></td>
            <td>
                <asp:FileUpload ID="ProductImage" runat="server" />
                <asp:RequiredFieldValidator ID="RequiredFieldValidator4" runat="server" Text="* Image
path required." ControlToValidate="ProductImage" SetFocusOnError="true" Display="Dynamic">
</asp:RequiredFieldValidator>
            </td>
        </tr>
    </table>
    <p></p>
    <p></p>
    <asp:Button ID="AddProductButton" runat="server" Text="Add Product"
OnClick="AddProductButton_Click"  CausesValidation="true"/>
    <asp:Label ID="LabelAddStatus" runat="server" Text=""></asp:Label>
    <p></p>
    <h3>Remove Product:</h3>
    <table>
        <tr>
            <td><asp:Label ID="LabelRemoveProduct" runat="server">Product:</asp:Label></td>
            <td><asp:DropDownList ID="DropDownRemoveProduct" runat="server"
ItemType="WingtipToys.Models.Product"
                    SelectMethod="GetProducts" AppendDataBoundItems="true"
                    DataTextField="ProductName" DataValueField="ProductID" >
                </asp:DropDownList>
            </td>
        </tr>
    </table>
    <p></p>
    <asp:Button ID="RemoveProductButton" runat="server" Text="Remove Product"
OnClick="RemoveProductButton_Click" CausesValidation="false"/>
    <asp:Label ID="LabelRemoveStatus" runat="server" Text=""></asp:Label>
</asp:Content>
```

3. Next, open the *AdminPage.aspx.cs* code-behind file by right-clicking the *AdminPage.aspx* and clicking **View Code**.

4. Replace the existing code in the *AdminPage.aspx.cs* code-behind file with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Models;
using WingtipToys.Logic;

namespace WingtipToys.Admin
{
  public partial class AdminPage : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      string productAction = Request.QueryString["ProductAction"];
      if (productAction == "add")
      {
        LabelAddStatus.Text = "Product added!";
      }
```

```
      if (productAction == "remove")
      {
        LabelRemoveStatus.Text = "Product removed!";
      }
    }

    protected void AddProductButton_Click(object sender, EventArgs e)
    {
      Boolean fileOK = false;
      String path = Server.MapPath("~/Catalog/Images/");
      if (ProductImage.HasFile)
      {
        String fileExtension = System.IO.Path.GetExtension(ProductImage.FileName).ToLower();
        String[] allowedExtensions = { ".gif", ".png", ".jpeg", ".jpg" };
        for (int i = 0; i < allowedExtensions.Length; i++)
        {
          if (fileExtension == allowedExtensions[i])
          {
            fileOK = true;
          }
        }
      }

      if (fileOK)
      {
        try
        {
          // Save to Images folder.
          ProductImage.PostedFile.SaveAs(path + ProductImage.FileName);
          // Save to Images/Thumbs folder.
          ProductImage.PostedFile.SaveAs(path + "Thumbs/" + ProductImage.FileName);
        }
        catch (Exception ex)
        {
          LabelAddStatus.Text = ex.Message;
        }

        // Add product data to DB.
        AddProducts products = new AddProducts();
        bool addSuccess = products.AddProduct(AddProductName.Text, AddProductDescription.Text,
            AddProductPrice.Text, DropDownAddCategory.SelectedValue, ProductImage.FileName);
        if (addSuccess)
        {
          // Reload the page.
          string pageUrl = Request.Url.AbsoluteUri.Substring(0, Request.Url.AbsoluteUri.Count() -
Request.Url.Query.Count());
          Response.Redirect(pageUrl + "?ProductAction=add");
        }
        else
        {
          LabelAddStatus.Text = "Unable to add new product to database.";
        }
      }
      else
      {
        LabelAddStatus.Text = "Unable to accept file type.";
      }
    }

    public IQueryable GetCategories()
    {
      var _db = new WingtipToys.Models.ProductContext();
      IQueryable query = _db.Categories;
      return query;
    }

    public IQueryable GetProducts()
    {
      var _db = new WingtipToys.Models.ProductContext();
```

```
      var _db = new WingtipToys.Models.ProductContext();
      IQueryable query = _db.Products;
      return query;
    }

    protected void RemoveProductButton_Click(object sender, EventArgs e)
    {
      using (var _db = new WingtipToys.Models.ProductContext())
      {
        int productId = Convert.ToInt16(DropDownRemoveProduct.SelectedValue);
        var myItem = (from c in _db.Products where c.ProductID == productId select c).FirstOrDefault();
        if (myItem != null)
        {
          _db.Products.Remove(myItem);
          _db.SaveChanges();

          // Reload the page.
          string pageUrl = Request.Url.AbsoluteUri.Substring(0, Request.Url.AbsoluteUri.Count() -
Request.Url.Query.Count());
          Response.Redirect(pageUrl + "?ProductAction=remove");
        }
        else
        {
          LabelRemoveStatus.Text = "Unable to locate product.";
        }
      }
    }
  }
}
```

In the code that you entered for the *AdminPage.aspx.cs* code-behind file, a class called `AddProducts` does the actual work of adding products to the database. This class doesn't exist yet, so you will create it now.

1. In **Solution Explorer**, right-click the *Logic* folder and then select **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.

2. Select the **Visual C#** -> **Code** templates group on the left. Then, select **Class** from the middle list and name it *AddProducts.cs*.
   The new class file is displayed.

3. Replace the existing code with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using WingtipToys.Models;

namespace WingtipToys.Logic
{
    public class AddProducts
    {
        public bool AddProduct(string ProductName, string ProductDesc, string ProductPrice, string
ProductCategory, string ProductImagePath)
        {
            var myProduct = new Product();
            myProduct.ProductName = ProductName;
            myProduct.Description = ProductDesc;
            myProduct.UnitPrice = Convert.ToDouble(ProductPrice);
            myProduct.ImagePath = ProductImagePath;
            myProduct.CategoryID = Convert.ToInt32(ProductCategory);

            using (ProductContext _db = new ProductContext())
            {
                // Add product to DB.
                _db.Products.Add(myProduct);
                _db.SaveChanges();
            }
            // Success.
            return true;
        }
    }
}
```

The *AdminPage.aspx* page allows the user belonging to the "canEdit" role to add and remove products. When a new product is added, the details about the product are validated and then entered into the database. The new product is immediately available to all users of the web application.

Unobtrusive Validation

The product details that the user provides on the *AdminPage.aspx* page are validated using validation controls (`RequiredFieldValidator` and `RegularExpressionValidator`). These controls automatically use unobtrusive validation. Unobtrusive validation allows the validation controls to use JavaScript for client-side validation logic, which means the page does not require a trip to the server to be validated. By default, unobtrusive validation is included in the *Web.config* file based on the following configuration setting:

```
<add key="ValidationSettings:UnobtrusiveValidationMode" value="WebForms" />
```

Regular Expressions

The product price on the *AdminPage.aspx* page is validated using a **RegularExpressionValidator** control. This control validates whether the value of the associated input control (the "AddProductPrice" TextBox) matches the pattern specified by the regular expression. A regular expression is a pattern-matching notation that enables you to quickly find and match specific character patterns. The **RegularExpressionValidator** control includes a property named `ValidationExpression` that contains the regular expression used to validate price input, as shown below:

```
<asp:RegularExpressionValidator
    ID="RegularExpressionValidator1" runat="server"
    Text="* Must be a valid price without $." ControlToValidate="AddProductPrice"
    SetFocusOnError="True" Display="Dynamic"
    ValidationExpression="^[0-9]*(\.)?[0-9]?[0-9]?$">
</asp:RegularExpressionValidator>
```

FileUpload Control

In addition to the input and validation controls, you added the **FileUpload** control to the *AdminPage.aspx* page. This control provides the capability to upload files. In this case, you are only allowing image files to be uploaded. In the code-behind file (*AdminPage.aspx.cs*), when the `AddProductButton` is clicked, the code checks the `HasFile` property of the **FileUpload** control. If the control has a file and if the file type (based on file extension) is allowed, the image is saved to the *Images* folder and the *Images/Thumbs* folder of the application.

Model Binding

Earlier in this tutorial series you used model binding to populate a **ListView** control, a **FormsView** control, a **GridView** control, and a **DetailView** control. In this tutorial, you use model binding to populate a **DropDownList** control with a list of product categories.

The markup that you added to the *AdminPage.aspx* file contains a **DropDownList** control called `DropDownAddCategory`:

```
<asp:DropDownList ID="DropDownAddCategory" runat="server"
        ItemType="WingtipToys.Models.Category"
        SelectMethod="GetCategories" DataTextField="CategoryName"
        DataValueField="CategoryID" >
    </asp:DropDownList>
```

You use model binding to populate this **DropDownList** by setting the `ItemType` attribute and the `SelectMethod` attribute. The `ItemType` attribute specifies that you use the `WingtipToys.Models.Category` type when populating the control. You defined this type at the beginning of this tutorial series by creating the `Category` class (shown below). The `Category` class is in the *Models* folder inside the *Category.cs* file.

```
public class Category
{
    [ScaffoldColumn(false)]
    public int CategoryID { get; set; }

    [Required, StringLength(100), Display(Name = "Name")]
    public string CategoryName { get; set; }

    [Display(Name = "Product Description")]
    public string Description { get; set; }

    public virtual ICollection<Product> Products { get; set; }
}
```

The `SelectMethod` attribute of the **DropDownList** control specifies that you use the `GetCategories` method (shown below) that is included in the code-behind file (*AdminPage.aspx.cs*).

```
public IQueryable GetCategories()
{
  var _db = new WingtipToys.Models.ProductContext();
  IQueryable query = _db.Categories;
  return query;
}
```

This method specifies that an `IQueryable` interface is used to evaluate a query against a `Category` type. The returned value is used to populate the **DropDownList** in the markup of the page (*AdminPage.aspx*).

The text displayed for each item in the list is specified by setting the `DataTextField` attribute. The `DataTextField` attribute uses the `CategoryName` of the `Category` class (shown above) to display each category in the **DropDownList** control. The actual value that is passed when an item is selected in the **DropDownList** control is based on the `DataValueField` attribute. The `DataValueField` attribute is set to the `CategoryID` as define in the

`Category` class (shown above).

How the Application Will Work

When the user belonging to the "canEdit" role navigates to the page for the first time, the `DropDownAddCategory` **DropDownList** control is populated as described above. The `DropDownRemoveProduct` **DropDownList** control is also populated with products using the same approach. The user belonging to the "canEdit" role selects the category type and adds product details (**Name**, **Description**, **Price**, and **Image File**). When the user belonging to the "canEdit" role clicks the **Add Product** button, the `AddProductButton_Click` event handler is triggered. The `AddProductButton_Click` event handler located in the code-behind file (*AdminPage.aspx.cs*) checks the image file to make sure it matches the allowed file types *(.gif*, *.png*, *.jpeg*, or *.jpg*). Then, the image file is saved into a folder of the Wingtip Toys sample application. Next, the new product is added to the database. To accomplish adding a new product, a new instance of the `AddProducts` class is created and named products. The `AddProducts` class has a method named `AddProduct`, and the products object calls this method to add products to the database.

```
// Add product data to DB.
AddProducts products = new AddProducts();
bool addSuccess = products.AddProduct(AddProductName.Text, AddProductDescription.Text,
    AddProductPrice.Text, DropDownAddCategory.SelectedValue, ProductImage.FileName);
```

If the code successfully adds the new product to the database, the page is reloaded with the query string value `ProductAction=add`.

```
Response.Redirect(pageUrl + "?ProductAction=add");
```

When the page reloads, the query string is included in the URL. By reloading the page, the user belonging to the "canEdit" role can immediately see the updates in the **DropDownList** controls on the *AdminPage.aspx* page. Also, by including the query string with the URL, the page can display a success message to the user belonging to the "canEdit" role.

When the *AdminPage.aspx* page reloads, the `Page_Load` event is called.

```
protected void Page_Load(object sender, EventArgs e)
{
    string productAction = Request.QueryString["ProductAction"];
    if (productAction == "add")
    {
        LabelAddStatus.Text = "Product added!";
    }

    if (productAction == "remove")
    {
        LabelRemoveStatus.Text = "Product removed!";
    }
}
```

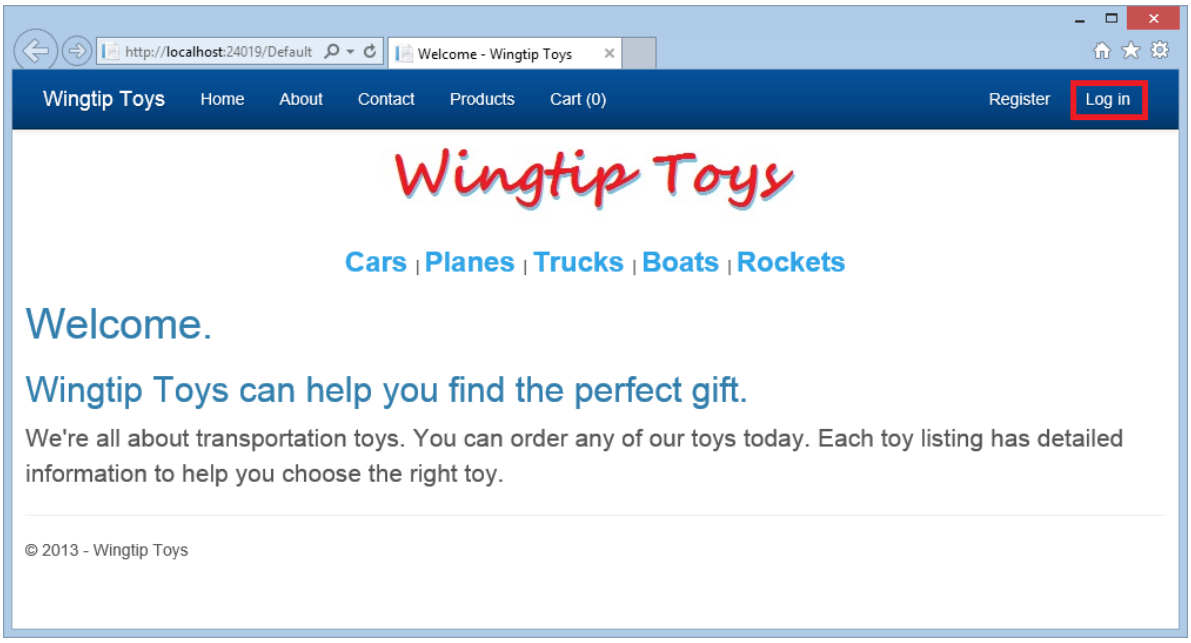
The `Page_Load` event handler checks the query string value and determines whether to show a success message.

Running the Application

You can run the application now to see how you can add, delete, and update items in the shopping cart. The shopping cart total will reflect the total cost of all items in the shopping cart.

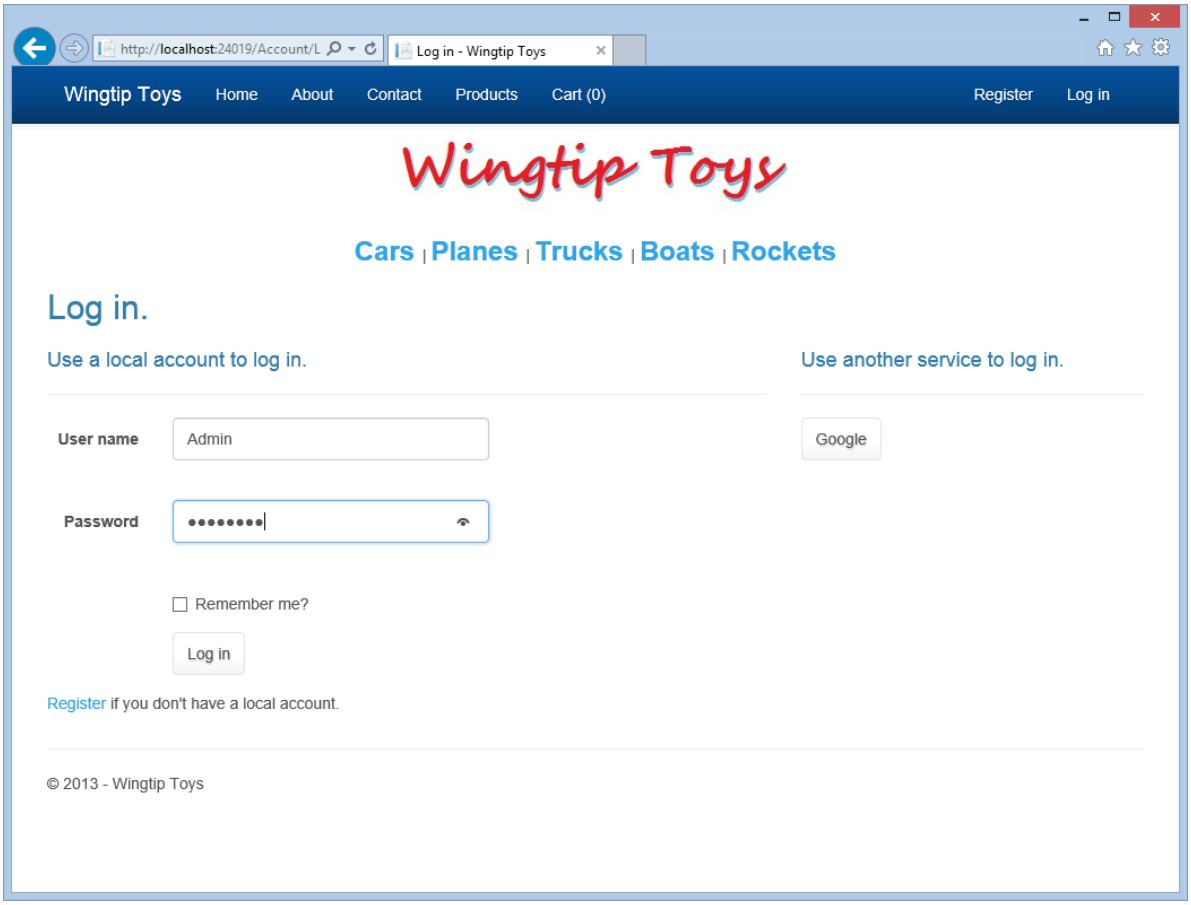
1. In Solution Explorer, press **F5** to run the Wingtip Toys sample application.
   The browser opens and shows the *Default.aspx* page.

2. Click the **Log in** link at the top of the page.



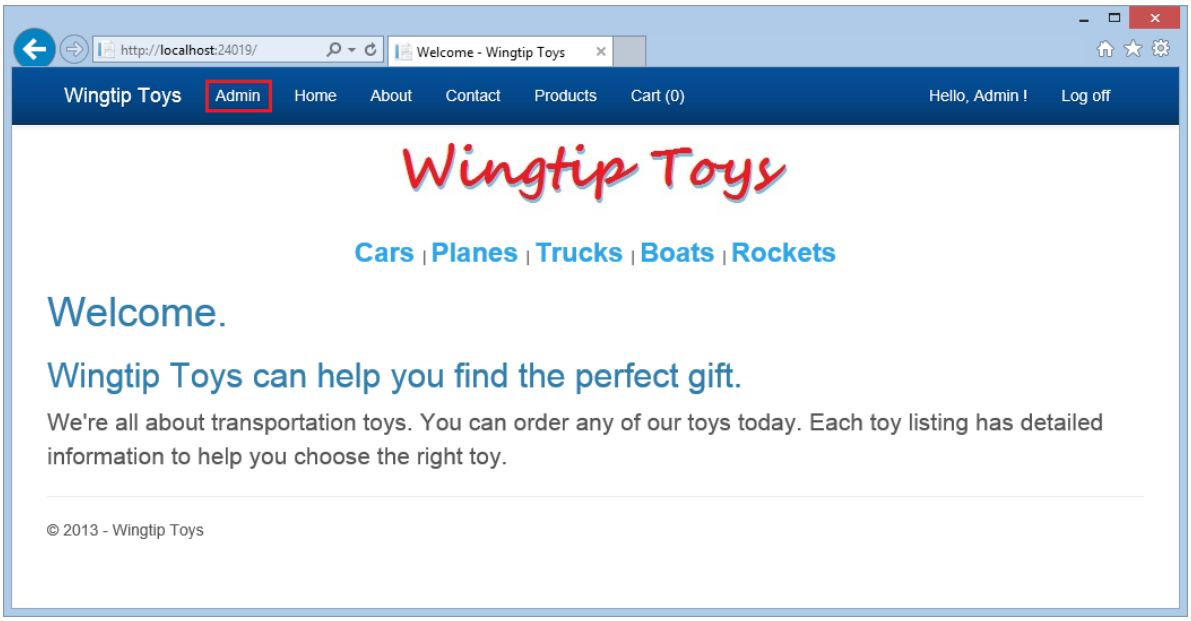
The *Login.aspx* page is displayed.

3. Use the following user name and password:
   User name: canEditUser@wingtiptoys.com
   Password: Pa$$word1

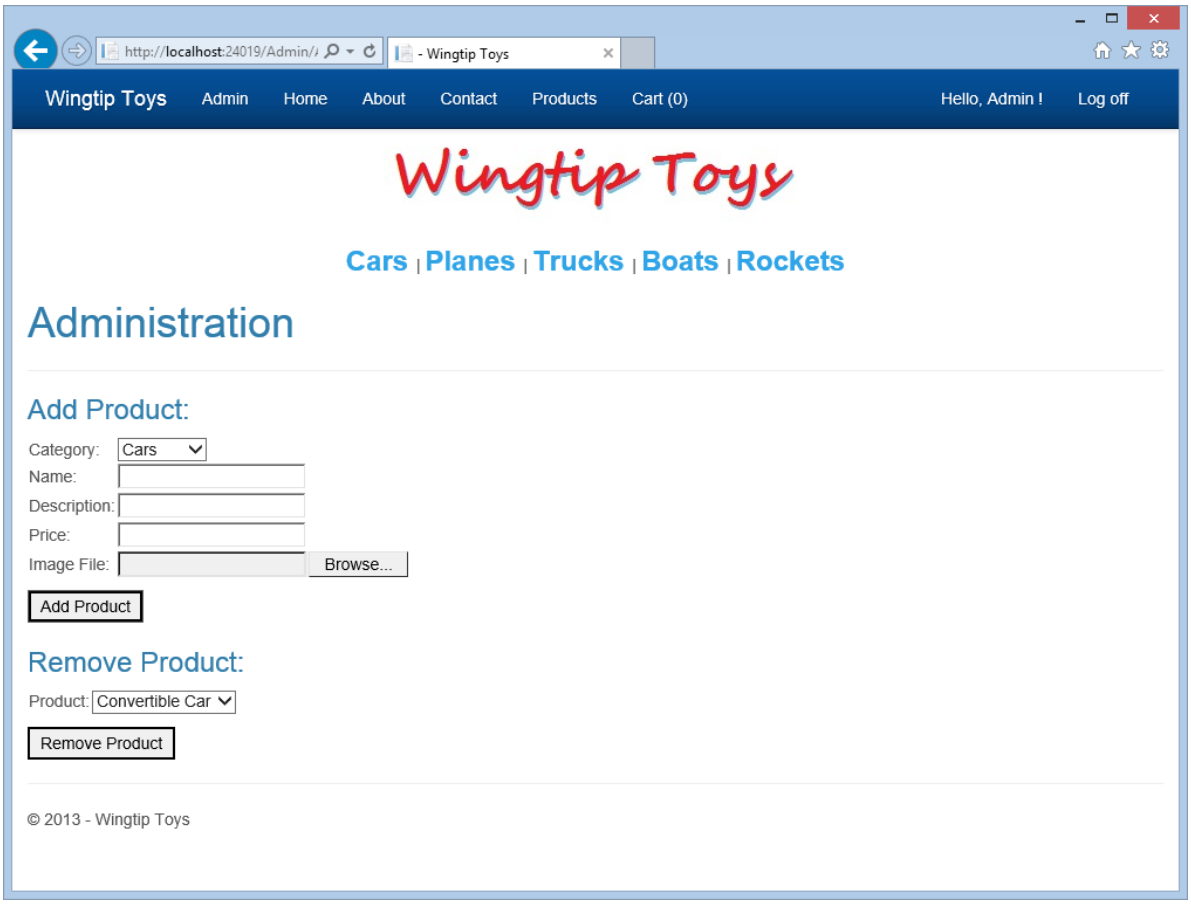


4. Click the **Log in** button near the bottom of the page.

5. At the top of the next page, select the **Admin** link to navigate to the *AdminPage.aspx* page.

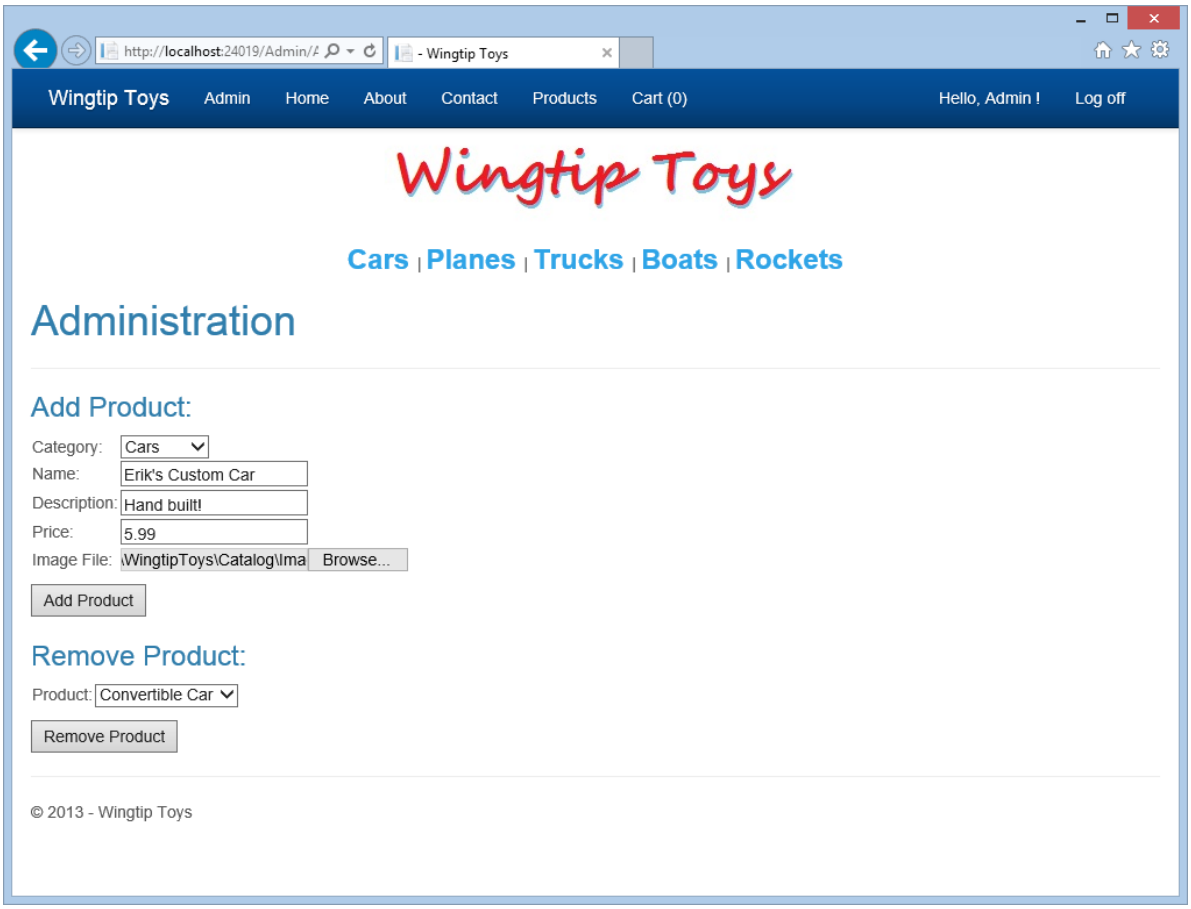


6. To test the input validation, click the **Add Product** button without adding any product details.

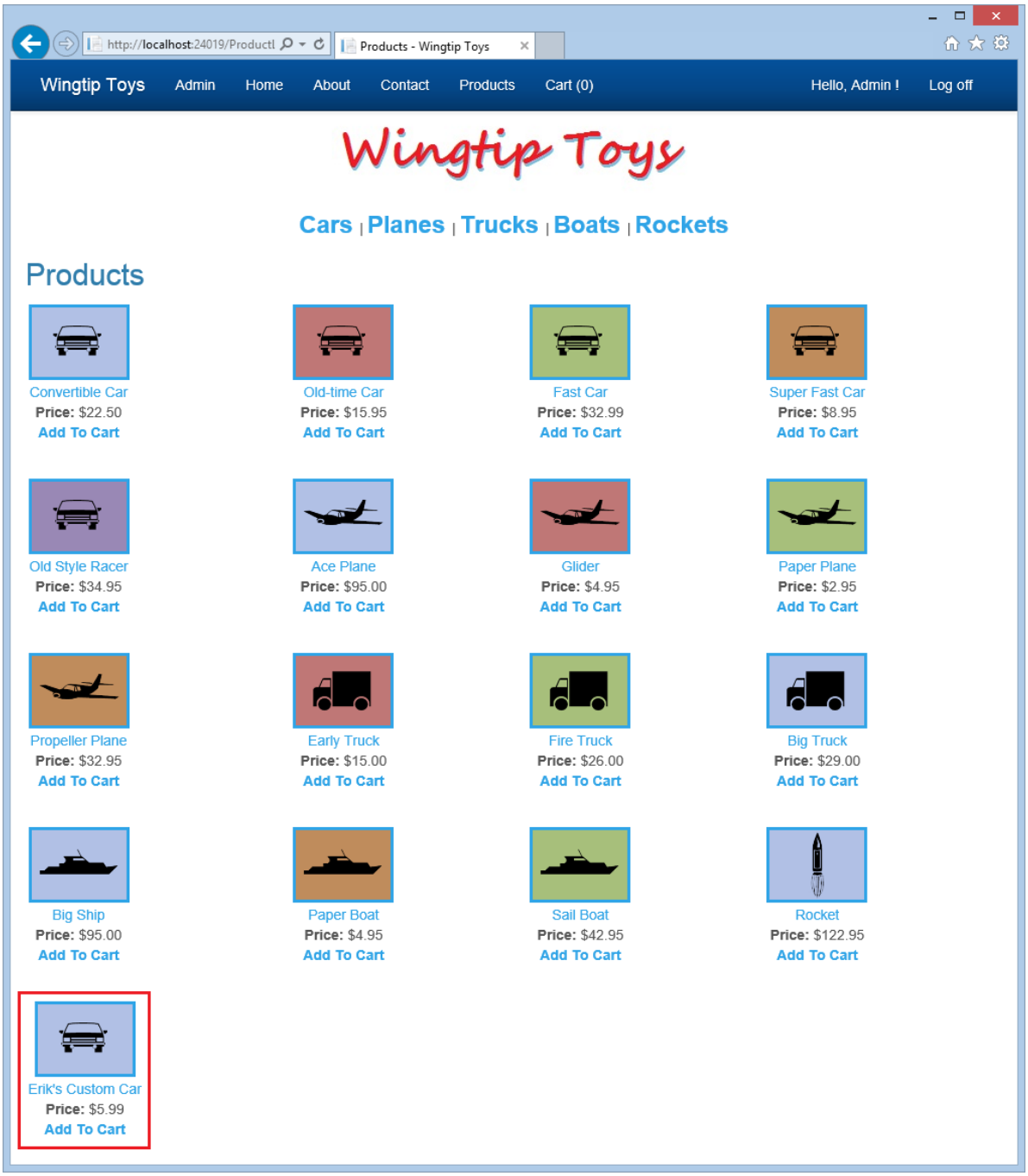


Notice that the required field messages are displayed.

7. Add the details for a new product, and then click the **Add Product** button.



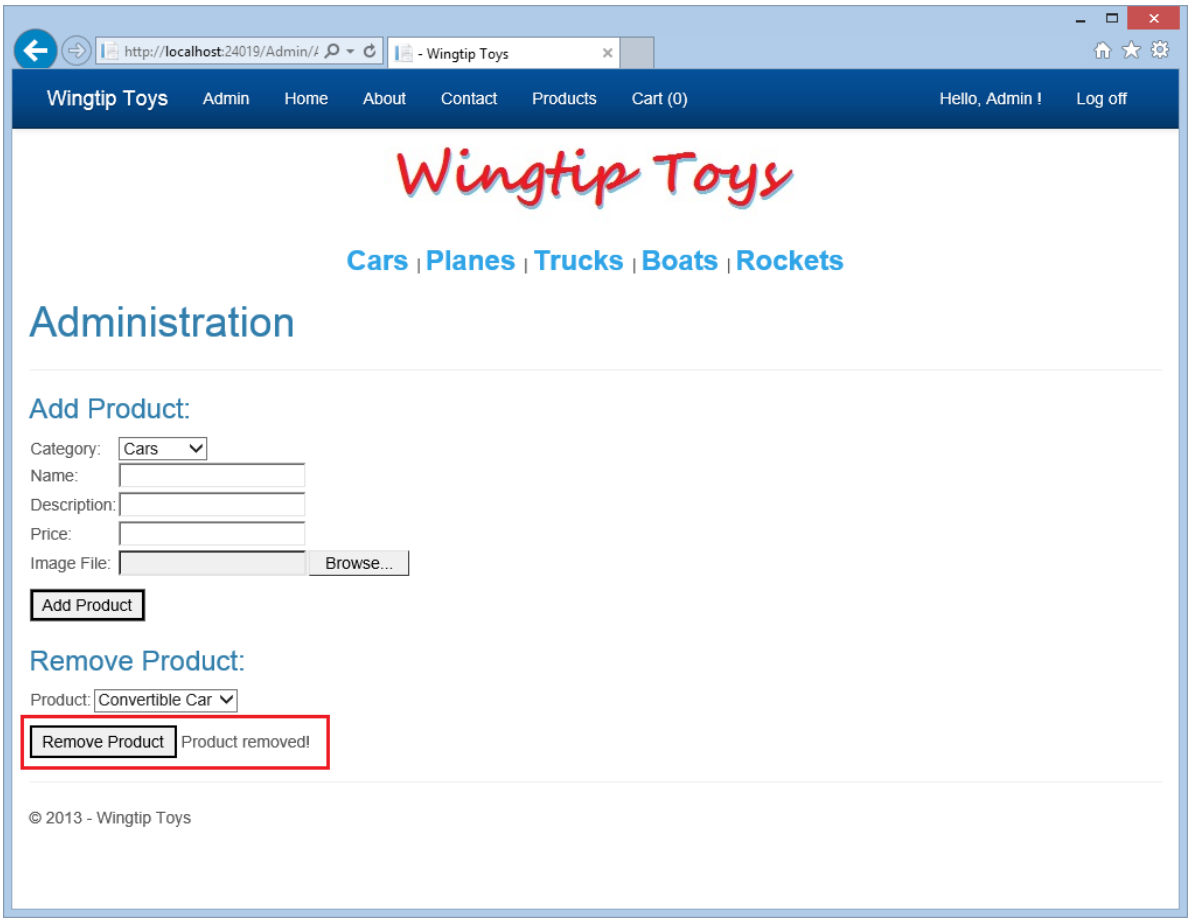
8. Select **Products** from the top navigation menu to view the new product you added.



9. Click the **Admin** link to return to the administration page.

10. In the **Remove Product** section of the page, select the new product you added in the **DropDownListBox**.

11. Click the **Remove Product** button to remove the new product from the application.



12. Select **Products** from the top navigation menu to confirm that the product has been removed.

13. Click **Log off** to exist administration mode.
    Notice that the top navigation pane no longer shows the **Admin** menu item.

Summary

In this tutorial, you added a custom role and a user belonging to the custom role, restricted access to the administration folder and page, and provided navigation for the user belonging to the custom role. You used model binding to populate a **DropDownList** control with data. You implemented the **FileUpload** control and validation controls. Also, you have learned how to add and remove products from a database. In the next tutorial, you'll learn how to implement ASP.NET routing.

Additional Resources

Web.config - authorization Element
ASP.NET Identity
Deploy a Secure ASP.NET Web Forms App with Membership, OAuth, and SQL Database to an Azure Web Site
Microsoft Azure - Free Trial

PREVIOUS NEXT

URL Routing

2/20/2020 • 8 minutes to read • Edit Online

by Erik Reitan

Download Wingtip Toys Sample Project (C#) or Download E-book (PDF)

> This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

In this tutorial, you will modify the Wingtip Toys sample application to support URL routing. Routing enables your web application to use URLs that are friendly, easier to remember, and better supported by search engines. This tutorial builds on the previous tutorial "Membership and Administration" and is part of the Wingtip Toys tutorial series.

What you'll learn:

- How to register routes for an ASP.NET Web Forms application.
- How to add routes to a web page.
- How to select data from a database to support routes.

ASP.NET Routing Overview

URL routing allows you to configure an application to accept request URLs that do not map to physical files. A request URL is simply the URL a user enters into their browser to find a page on your web site. You use routing to define URLs that are semantically meaningful to users and that can help with search-engine optimization (SEO).

By default, the Web Forms template includes ASP.NET Friendly URLs. Much of the basic routing work will be implemented by using *Friendly URLs*. However, in this tutorial you will add customized routing capabilities.

Before customizing URL routing, the Wingtip Toys sample application can link to a product using the following URL:

```
https://localhost:44300/ProductDetails.aspx?productID=2
```

By customizing URL routing, the Wingtip Toys sample application will link to the same product using an easier to read URL:

```
https://localhost:44300/Product/Convertible%20Car
```

Routes

A route is a URL pattern that is mapped to a handler. The handler can be a physical file, such as an .aspx file in a Web Forms application. A handler can also be a class that processes the request. To define a route, you create an instance of the Route class by specifying the URL pattern, the handler, and optionally a name for the route.

You add the route to the application by adding the `Route` object to the static `Routes` property of the `RouteTable` class. The Routes property is a `RouteCollection` object that stores all the routes for the application.

URL Patterns

A URL pattern can contain literal values and variable placeholders (referred to as URL parameters). The literals and placeholders are located in segments of the URL which are delimited by the slash (`/`) character.

When a request to your web application is made, the URL is parsed into segments and placeholders, and the variable values are provided to the request handler. This process is similar to the way the data in a query string is parsed and passed to the request handler. In both cases, variable information is included in the URL and passed to the handler in the form of key-value pairs. For query strings, both the keys and the values are in the URL. For routes, the keys are the placeholder names defined in the URL pattern, and only the values are in the URL.

In a URL pattern, you define placeholders by enclosing them in braces (`{` and `}`). You can define more than one placeholder in a segment, but the placeholders must be separated by a literal value. For example, `{language}-{country}/{action}` is a valid route pattern. However, `{language}{country}/{action}` is not a valid pattern, because there is no literal value or delimiter between the placeholders. Therefore, routing cannot determine where to separate the value for the language placeholder from the value for the country placeholder.

Mapping and Registering Routes

Before you can include routes to pages of the Wingtip Toys sample application, you must register the routes when the application starts. To register the routes, you will modify the `Application_Start` event handler.

1. In **Solution Explorer** of Visual Studio, find and open the *Global.asax.cs* file.
2. Add the code highlighted in yellow to the *Global.asax.cs* file as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Optimization;
using System.Web.Routing;
using System.Web.Security;
using System.Web.SessionState;
using System.Data.Entity;
using WingtipToys.Models;
using WingtipToys.Logic;

namespace WingtipToys
{
    public class Global : HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
          // Code that runs on application startup
          RouteConfig.RegisterRoutes(RouteTable.Routes);
          BundleConfig.RegisterBundles(BundleTable.Bundles);

          // Initialize the product database.
          Database.SetInitializer(new ProductDatabaseInitializer());

          // Create custom role and user.
          RoleActions roleActions = new RoleActions();
          roleActions.AddUserAndRole();

          // Add Routes.
          RegisterCustomRoutes(RouteTable.Routes);
        }

        void RegisterCustomRoutes(RouteCollection routes)
        {
          routes.MapPageRoute(
              "ProductsByCategoryRoute",
              "Category/{categoryName}",
              "~/ProductList.aspx"
          );
          routes.MapPageRoute(
              "ProductByNameRoute",
              "Product/{productName}",
              "~/ProductDetails.aspx"
          );
        }
    }
}
```

When the Wingtip Toys sample application starts, it calls the `Application_Start` event handler. At the end of this event handler, the `RegisterCustomRoutes` method is called. The `RegisterCustomRoutes` method adds each route by calling the `MapPageRoute` method of the `RouteCollection` object. Routes are defined using a route name, a route URL and a physical URL.

The first parameter ("`ProductsByCategoryRoute`") is the route name. It is used to call the route when it is needed. The second parameter ("`Category/{categoryName}`") defines the friendly replacement URL that can be dynamic based on code. You use this route when you are populating a data control with links that are generated based on data. A route is shown as follows:

```
routes.MapPageRoute(
      "ProductsByCategoryRoute",
      "Category/{categoryName}",
      "~/ProductList.aspx"
  );
```

The second parameter of the route includes a dynamic value specified by braces (`{ }`). In this case, the `categoryName` is a variable that will be used to determine the proper routing path.

> **NOTE**
>
> **Optional**
>
> You might find it easier to manage your code by moving the `RegisterCustomRoutes` method to a separate class. In the *Logic* folder, create a separate `RouteActions` class. Move the above `RegisterCustomRoutes` method from the *Global.asax.cs* file into the new `RoutesActions` class. Use the `RoleActions` class and the `createAdmin` method as an example of how to call the `RegisterCustomRoutes` method from the *Global.asax.cs* file.

You may also have noticed the `RegisterRoutes` method call using the `RouteConfig` object at the beginning of the `Application_Start` event handler. This call is made to implement default routing. It was included as default code when you created the application using Visual Studio's Web Forms template.

Retrieving and Using Route Data

As mentioned above, routes can be defined. The code that you added to the `Application_Start` event handler in the *Global.asax.cs* file loads the definable routes.

Setting Routes

Routes require you to add additional code. In this tutorial, you will use model binding to retrieve a `RouteValueDictionary` object that is used when generating the routes using data from a data control. The `RouteValueDictionary` object will contain a list of product names that belong to a specific category of products. A link is created for each product based on the data and route.

Enable Routes for Categories and Products

Next, you'll update the application to use the `ProductsByCategoryRoute` to determine the correct route to include for each product category link. You'll also update the *ProductList.aspx* page to include a routed link for each product. The links will be displayed as they were before the change, however the links will now use URL routing.

1. In **Solution Explorer**, open the *Site.Master* page if it is not already open.
2. Update the **ListView** control named "`categoryList`" with the changes highlighted in yellow, so the markup appears as follows:

```
<asp:ListView ID="categoryList"
    ItemType="WingtipToys.Models.Category"
    runat="server"
    SelectMethod="GetCategories" >
    <ItemTemplate>
        <b style="font-size: large; font-style: normal">
        <a href="<%#: GetRouteUrl("ProductsByCategoryRoute", new {categoryName = Item.CategoryName})
%>">
            <%#: Item.CategoryName %>
        </a>
        </b>
    </ItemTemplate>
    <ItemSeparatorTemplate>  |  </ItemSeparatorTemplate>
</asp:ListView>
```

3. In **Solution Explorer**, open the *ProductList.aspx* page.

4. Update the `ItemTemplate` element of the *ProductList.aspx* page with the updates highlighted in yellow, so the markup appears as follows:

```
<ItemTemplate>
  <td runat="server">
    <table>
      <tr>
        <td>
          <a href="<%#: GetRouteUrl("ProductByNameRoute", new {productName = Item.ProductName}) %>">
            <image src='/Catalog/Images/Thumbs/<%#:Item.ImagePath%>'
              width="100" height="75" border="1" />
          </a>
        </td>
      </tr>
      <tr>
        <td>
          <a href="<%#: GetRouteUrl("ProductByNameRoute", new {productName = Item.ProductName}) %>">
            <%#:Item.ProductName%>
          </a>
          <br />
          <span>
            <b>Price: </b><%#:String.Format("{0:c}", Item.UnitPrice)%>
          </span>
          <br />
          <a href="/AddToCart.aspx?productID=<%#:Item.ProductID %>">
            <span class="ProductListItem">
              <b>Add To Cart<b>
            </span>
          </a>
        </td>
      </tr>
      <tr>
        <td> </td>
      </tr>
    </table>
    </p>
  </td>
</ItemTemplate>
```

5. Open the code-behind of *ProductList.aspx.cs* and add the following namespace as highlighted in yellow:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Models;
using System.Web.ModelBinding;
using System.Web.Routing;
```

6. Replace the `GetProducts` method of the code-behind (*ProductList.aspx.cs*) with the following code:

```
public IQueryable<Product> GetProducts(
    [QueryString("id")] int? categoryId,
    [RouteData] string categoryName)
{
    var _db = new WingtipToys.Models.ProductContext();
    IQueryable<Product> query = _db.Products;

    if (categoryId.HasValue && categoryId > 0)
    {
        query = query.Where(p => p.CategoryID == categoryId);
    }

    if (!String.IsNullOrEmpty(categoryName))
    {
        query = query.Where(p =>
            String.Compare(p.Category.CategoryName,
            categoryName) == 0);
    }
    return query;
}
```

Add Code for Product Details

Now, update the code-behind (*ProductDetails.aspx.cs*) for the *ProductDetails.aspx* page to use route data. Notice that the new `GetProduct` method also accepts a query string value for the case where the user has a link bookmarked that uses the older non-friendly, non-routed URL.

1. Replace the `GetProduct` method of the code-behind (*ProductDetails.aspx.cs*) with the following code:

```
public IQueryable<Product> GetProduct(
        [QueryString("ProductID")] int? productId,
        [RouteData] string productName)
{
    var _db = new WingtipToys.Models.ProductContext();
    IQueryable<Product> query = _db.Products;
    if (productId.HasValue && productId > 0)
    {
        query = query.Where(p => p.ProductID == productId);
    }
    else if (!String.IsNullOrEmpty(productName))
    {
        query = query.Where(p =>
              String.Compare(p.ProductName, productName) == 0);
    }
    else
    {
        query = null;
    }
    return query;
}
```

Running the Application

You can run the application now to see the updated routes.

1. Press **F5** to run the Wingtip Toys sample application.
   The browser opens and shows the *Default.aspx* page.
2. Click the **Products** link at the top of the page.
   All products are displayed on the *ProductList.aspx* page. The following URL (using your port number) is displayed for the browser:
   `https://localhost:44300/ProductList`
3. Next, click the **Cars** category link near the top of the page.
   Only cars are displayed on the *ProductList.aspx* page. The following URL (using your port number) is displayed for the browser:
   `https://localhost:44300/Category/Cars`
4. Click the link containing the name of the first car listed on the page ("**Convertible Car**") to display the product details.
   The following URL (using your port number) is displayed for the browser:
   `https://localhost:44300/Product/Convertible%20Car`
5. Next, enter the following non-routed URL (using your port number) into the browser:
   `https://localhost:44300/ProductDetails.aspx?productID=2`
   The code still recognizes a URL that includes a query string, for the case where a user has a link bookmarked.

Summary

In this tutorial, you have added routes for categories and products. You have learned how routes can be integrated with data controls that use model binding. In the next tutorial, you will implement global error handling.

Additional Resources

ASP.NET Friendly URLs
Deploy a Secure ASP.NET Web Forms App with Membership, OAuth, and SQL Database to Azure App Service
Microsoft Azure - Free Trial

PREVIOUS NEXT

ASP.NET Error Handling

2/20/2020 • 21 minutes to read • Edit Online

by Erik Reitan

Download Wingtip Toys Sample Project (C#) or Download E-book (PDF)

> This tutorial series will teach you the basics of building an ASP.NET Web Forms application using ASP.NET 4.5 and Microsoft Visual Studio Express 2013 for Web. A Visual Studio 2013 project with C# source code is available to accompany this tutorial series.

In this tutorial, you will modify the Wingtip Toys sample application to include error handling and error logging. Error handling will allow the application to gracefully handle errors and display error messages accordingly. Error logging will allow you to find and fix errors that have occurred. This tutorial builds on the previous tutorial "URL Routing" and is part of the Wingtip Toys tutorial series.

What you'll learn:

- How to add global error handling to the application's configuration.
- How to add error handling at the application, page, and code levels.
- How to log errors for later review.
- How to display error messages that do not compromise security.
- How to implement Error Logging Modules and Handlers (ELMAH) error logging.

Overview

ASP.NET applications must be able to handle errors that occur during execution in a consistent manner. ASP.NET uses the common language runtime (CLR), which provides a way of notifying applications of errors in a uniform way. When an error occurs, an exception is thrown. An exception is any error, condition, or unexpected behavior that an application encounters.

In the .NET Framework, an exception is an object that inherits from the `System.Exception` class. An exception is thrown from an area of code where a problem has occurred. The exception is passed up the call stack to a place where the application provides code to handle the exception. If the application does not handle the exception, the browser is forced to display the error details.

As a best practice, handle errors in at the code level in `Try` / `Catch` / `Finally` blocks within your code. Try to place these blocks so that the user can correct problems in the context in which they occur. If the error handling blocks are too far away from where the error occurred, it becomes more difficult to provide users with the information they need to fix the problem.

Exception Class

The Exception class is the base class from which exceptions inherit. Most exception objects are instances of some derived class of the Exception class, such as the `SystemException` class, the `IndexOutOfRangeException` class, or the `ArgumentNullException` class. The Exception class has properties, such as the `StackTrace` property, the `InnerException` property, and the `Message` property, that provide specific information about the error that has occurred.

Exception Inheritance Hierarchy

The runtime has a base set of exceptions deriving from the `SystemException` class that the runtime throws when

an exception is encountered. Most of the classes that inherit from the Exception class, such as the `IndexOutOfRangeException` class and the `ArgumentNullException` class, do not implement additional members. Therefore, the most important information for an exception can be found in the hierarchy of exceptions, the exception name, and the information contained in the exception.

Exception Handling Hierarchy

In an ASP.NET Web Forms application, exceptions can be handled based on a specific handling hierarchy. An exception can be handled at the following levels:

- Application level
- Page level
- Code level

When an application handles exceptions, additional information about the exception that is inherited from the Exception class can often be retrieved and displayed to the user. In addition to application, page, and code level, you can also handle exceptions at the HTTP module level and by using an IIS custom handler.

Application Level Error Handling

You can handle default errors at the application level either by modifying your application's configuration or by adding an `Application_Error` handler in the *Global.asax* file of your application.

You can handle default errors and HTTP errors by adding a `customErrors` section to the *Web.config* file. The `customErrors` section allows you to specify a default page that users will be redirected to when an error occurs. It also allows you to specify individual pages for specific status code errors.

```
<configuration>
  <system.web>
    <customErrors mode="On" defaultRedirect="ErrorPage.aspx?handler=customErrors%20section%20-%20Web.config">
      <error statusCode="404" redirect="ErrorPage.aspx?msg=404&amp;handler=customErrors%20section%20-
%20Web.config"/>
    </customErrors>
  </system.web>
</configuration>
```

Unfortunately, when you use the configuration to redirect the user to a different page, you do not have the details of the error that occurred.

However, you can trap errors that occur anywhere in your application by adding code to the `Application_Error` handler in the *Global.asax* file.

```
void Application_Error(object sender, EventArgs e)
{
    Exception exc = Server.GetLastError();

    if (exc is HttpUnhandledException)
    {
        // Pass the error on to the error page.
        Server.Transfer("ErrorPage.aspx?handler=Application_Error%20-%20Global.asax", true);
    }
}
```

Page Level Error Event Handling

A page-level handler returns the user to the page where the error occurred, but because instances of controls are not maintained, there will no longer be anything on the page. To provide the error details to the user of the application, you must specifically write the error details to the page.

You would typically use a page-level error handler to log unhandled errors or to take the user to a page that can

display helpful information.

This code example shows a handler for the Error event in an ASP.NET Web page. This handler catches all exceptions that are not already handled within `try` / `catch` blocks in the page.

```
private void Page_Error(object sender, EventArgs e)
{
    Exception exc = Server.GetLastError();

    // Handle specific exception.
    if (exc is HttpUnhandledException)
    {
        ErrorMsgTextBox.Text = "An error occurred on this page. Please verify your " +
        "information to resolve the issue."
    }
    // Clear the error from the server.
    Server.ClearError();
}
```

After you handle an error, you must clear it by calling the `ClearError` method of the Server object (`HttpServerUtility` class), otherwise you will see an error that has previously occurred.

Code Level Error Handling

The try-catch statement consists of a try block followed by one or more catch clauses, which specify handlers for different exceptions. When an exception is thrown, the common language runtime (CLR) looks for the catch statement that handles this exception. If the currently executing method does not contain a catch block, the CLR looks at the method that called the current method, and so on, up the call stack. If no catch block is found, then the CLR displays an unhandled exception message to the user and stops execution of the program.

The following code example shows a common way of using `try` / `catch` / `finally` to handle errors.

```
try
{
    file.ReadBlock(buffer, index, buffer.Length);
}
catch (FileNotFoundException e)
{
    Server.Transfer("NoFileErrorPage.aspx", true);
}
catch (System.IO.IOException e)
{
    Server.Transfer("IOErrorPage.aspx", true);
}

finally
{
    if (file != null)
    {
        file.Close();
    }
}
```

In the above code, the try block contains the code that needs to be guarded against a possible exception. The block is executed until either an exception is thrown or the block is completed successfully. If either a `FileNotFoundException` exception or an `IOException` exception occurs, the execution is transferred to a different page. Then, the code contained in the finally block is executed, whether an error occurred or not.

Adding Error Logging Support

Before adding error handling to the Wingtip Toys sample application, you will add error logging support by

adding an `ExceptionUtility` class to the *Logic* folder. By doing this, each time the application handles an error, the error details will be added to the error log file.

1. Right-click the *Logic* folder and then select **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.

2. Select the **Visual C#** -> **Code** templates group on the left. Then, select **Class** from the middle list and name it **ExceptionUtility.cs**.

3. Choose **Add**. The new class file is displayed.

4. Replace the existing code with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.IO;

namespace WingtipToys.Logic
{
  // Create our own utility for exceptions
  public sealed class ExceptionUtility
  {
    // All methods are static, so this can be private
    private ExceptionUtility()
    { }

    // Log an Exception
    public static void LogException(Exception exc, string source)
    {
      // Include logic for logging exceptions
      // Get the absolute path to the log file
      string logFile = "~/App_Data/ErrorLog.txt";
      logFile = HttpContext.Current.Server.MapPath(logFile);

      // Open the log file for append and write the log
      StreamWriter sw = new StreamWriter(logFile, true);
      sw.WriteLine("********** {0} **********", DateTime.Now);
      if (exc.InnerException != null)
      {
        sw.Write("Inner Exception Type: ");
        sw.WriteLine(exc.InnerException.GetType().ToString());
        sw.Write("Inner Exception: ");
        sw.WriteLine(exc.InnerException.Message);
        sw.Write("Inner Source: ");
        sw.WriteLine(exc.InnerException.Source);
        if (exc.InnerException.StackTrace != null)
        {
          sw.WriteLine("Inner Stack Trace: ");
          sw.WriteLine(exc.InnerException.StackTrace);
        }
      }
      sw.Write("Exception Type: ");
      sw.WriteLine(exc.GetType().ToString());
      sw.WriteLine("Exception: " + exc.Message);
      sw.WriteLine("Source: " + source);
      sw.WriteLine("Stack Trace: ");
      if (exc.StackTrace != null)
      {
        sw.WriteLine(exc.StackTrace);
        sw.WriteLine();
      }
      sw.Close();
    }
  }
}
```

When an exception occurs, the exception can be written to an exception log file by calling the `LogException` method. This method takes two parameters, the exception object and a string containing details about the source of the exception. The exception log is written to the *ErrorLog.txt* file in the *App_Data* folder.

Adding an Error Page

In the Wingtip Toys sample application, one page will be used to display errors. The error page is designed to show a secure error message to users of the site. However, if the user is a developer making an HTTP request that is being served locally on the machine where the code lives, additional error details will be displayed on the error page.

1. Right-click the project name (**Wingtip Toys**) in **Solution Explorer** and select **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.

2. Select the **Visual C#** -> **Web** templates group on the left. From the middle list, select **Web Form with Master Page**,and name it **ErrorPage.aspx**.

3. Click **Add**.

4. Select the *Site.Master* file as the master page, and then choose **OK**.

5. Replace the existing markup with the following:

```
<%@ Page Title="" Language="C#" AutoEventWireup="true" MasterPageFile="~/Site.Master"
CodeBehind="ErrorPage.aspx.cs" Inherits="WingtipToys.ErrorPage" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Error:</h2>
    <p></p>
    <asp:Label ID="FriendlyErrorMsg" runat="server" Text="Label" Font-Size="Large" style="color: red">
</asp:Label>

    <asp:Panel ID="DetailedErrorPanel" runat="server" Visible="false">
        <p> </p>
        <h4>Detailed Error:</h4>
        <p>
            <asp:Label ID="ErrorDetailedMsg" runat="server" Font-Size="Small" /><br />
        </p>

        <h4>Error Handler:</h4>
        <p>
            <asp:Label ID="ErrorHandler" runat="server" Font-Size="Small" /><br />
        </p>

        <h4>Detailed Error Message:</h4>
        <p>
            <asp:Label ID="InnerMessage" runat="server" Font-Size="Small" /><br />
        </p>
        <p>
            <asp:Label ID="InnerTrace" runat="server"  />
        </p>
    </asp:Panel>
</asp:Content>
```

6. Replace the existing code of the code-behind (*ErrorPage.aspx.cs*) so that it appears as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using WingtipToys.Logic;

namespace WingtipToys
{
  public partial class ErrorPage : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      // Create safe error messages.
      string generalErrorMsg = "A problem has occurred on this web site. Please try again. " +
          "If this error continues, please contact support.";
      string httpErrorMsg = "An HTTP error occurred. Page Not found. Please try again.";
      string unhandledErrorMsg = "The error was unhandled by application code.";

      // Display safe error message.
```

```
      FriendlyErrorMsg.Text = generalErrorMsg;

      // Determine where error was handled.
      string errorHandler = Request.QueryString["handler"];
      if (errorHandler == null)
      {
        errorHandler = "Error Page";
      }

      // Get the last error from the server.
      Exception ex = Server.GetLastError();

      // Get the error number passed as a querystring value.
      string errorMsg = Request.QueryString["msg"];
      if (errorMsg == "404")
      {
        ex = new HttpException(404, httpErrorMsg, ex);
        FriendlyErrorMsg.Text = ex.Message;
      }

      // If the exception no longer exists, create a generic exception.
      if (ex == null)
      {
        ex = new Exception(unhandledErrorMsg);
      }

      // Show error details to only you (developer). LOCAL ACCESS ONLY.
      if (Request.IsLocal)
      {
        // Detailed Error Message.
        ErrorDetailedMsg.Text = ex.Message;

        // Show where the error was handled.
        ErrorHandler.Text = errorHandler;

        // Show local access details.
        DetailedErrorPanel.Visible = true;

        if (ex.InnerException != null)
        {
          InnerMessage.Text = ex.GetType().ToString() + "<br/>" +
              ex.InnerException.Message;
          InnerTrace.Text = ex.InnerException.StackTrace;
        }
        else
        {
          InnerMessage.Text = ex.GetType().ToString();
          if (ex.StackTrace != null)
          {
            InnerTrace.Text = ex.StackTrace.ToString().TrimStart();
          }
        }
      }

      // Log the exception.
      ExceptionUtility.LogException(ex, errorHandler);

      // Clear the error from the server.
      Server.ClearError();
    }
  }
}
```

When the error page is displayed, the `Page_Load` event handler is executed. In the `Page_Load` handler, the location of where the error was first handled is determined. Then, the last error that occurred is determined by call the `GetLastError` method of the Server object. If the exception no longer exists, a generic exception is created. Then, if

the HTTP request was made locally, all error details are shown. In this case, only the local machine running the web application will see these error details. After the error information has been displayed, the error is added to the log file and the error is cleared from the server.

Displaying Unhandled Error Messages for the Application

By adding a `customErrors` section to the *Web.config* file, you can quickly handle simple errors that occur throughout the application. You can also specify how to handle errors based on their status code value, such as 404 - File not found.

Update the Configuration

Update the configuration by adding a `customErrors` section to the *Web.config* file.

1. In **Solution Explorer**, find and open the *Web.config* file at the root of the Wingtip Toys sample application.
2. Add the `customErrors` section to the *Web.config* file within the `<system.web>` node as follows:

```
<configuration>
  <system.web>
    <customErrors mode="On" defaultRedirect="ErrorPage.aspx?handler=customErrors%20section%20-
%20Web.config">
      <error statusCode="404" redirect="ErrorPage.aspx?msg=404&amp;handler=customErrors%20section%20-
%20Web.config"/>
    </customErrors>
  </system.web>
</configuration>
```

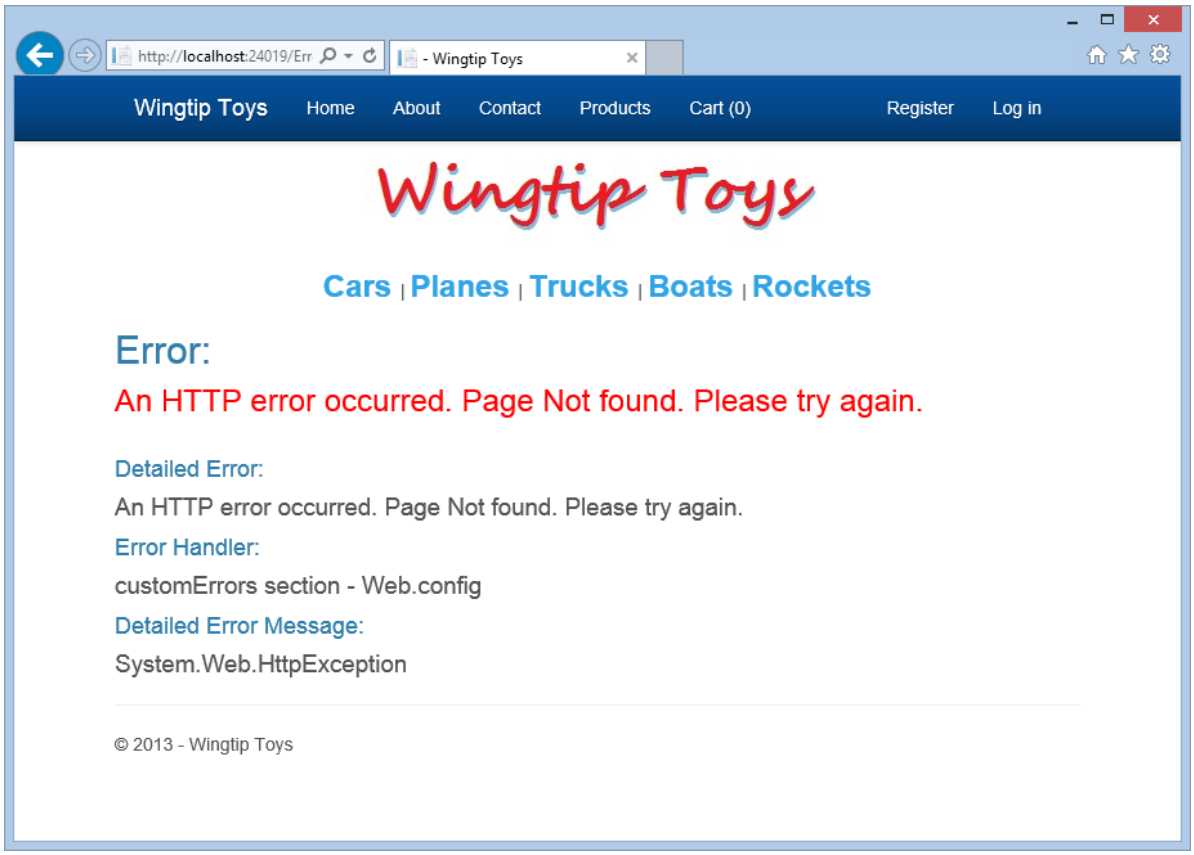
3. Save the *Web.config* file.

The `customErrors` section specifies the mode, which is set to "On". It also specifies the `defaultRedirect`, which tells the application which page to navigate to when an error occurs. In addition, you have added a specific error element that specifies how to handle a 404 error when a page is not found. Later in this tutorial, you will add additional error handling that will capture the details of an error at the application level.

Running the Application

You can run the application now to see the updated routes.

1. Press **F5** to run the Wingtip Toys sample application.
   The browser opens and shows the *Default.aspx* page.
2. Enter the following URL into the browser (be sure to use **your** port number):
   `https://localhost:44300/NoPage.aspx`
3. Review the *ErrorPage.aspx* displayed in the browser.



When you request the *NoPage.aspx* page, which does not exist, the error page will show the simple error message and the detailed error information if additional details are available. However, if the user requested a non-existent page from a remote location, the error page would only show the error message in red.

Including an Exception for Testing Purposes

To verify how your application will function when an error occurs, you can deliberately create error conditions in ASP.NET. In the Wingtip Toys sample application, you will throw a test exception when the default page loads to see what happens.

1. Open the code-behind of the *Default.aspx* page in Visual Studio.
   The *Default.aspx.cs* code-behind page will be displayed.

2. In the `Page_Load` handler, add code so that the handler appears as follows:

```
protected void Page_Load(object sender, EventArgs e)
{
    throw new InvalidOperationException("An InvalidOperationException " +
    "occurred in the Page_Load handler on the Default.aspx page.");
}
```

It is possible to create various different types of exceptions. In the above code, you are creating an `InvalidOperationException` when the *Default.aspx* page is loaded.

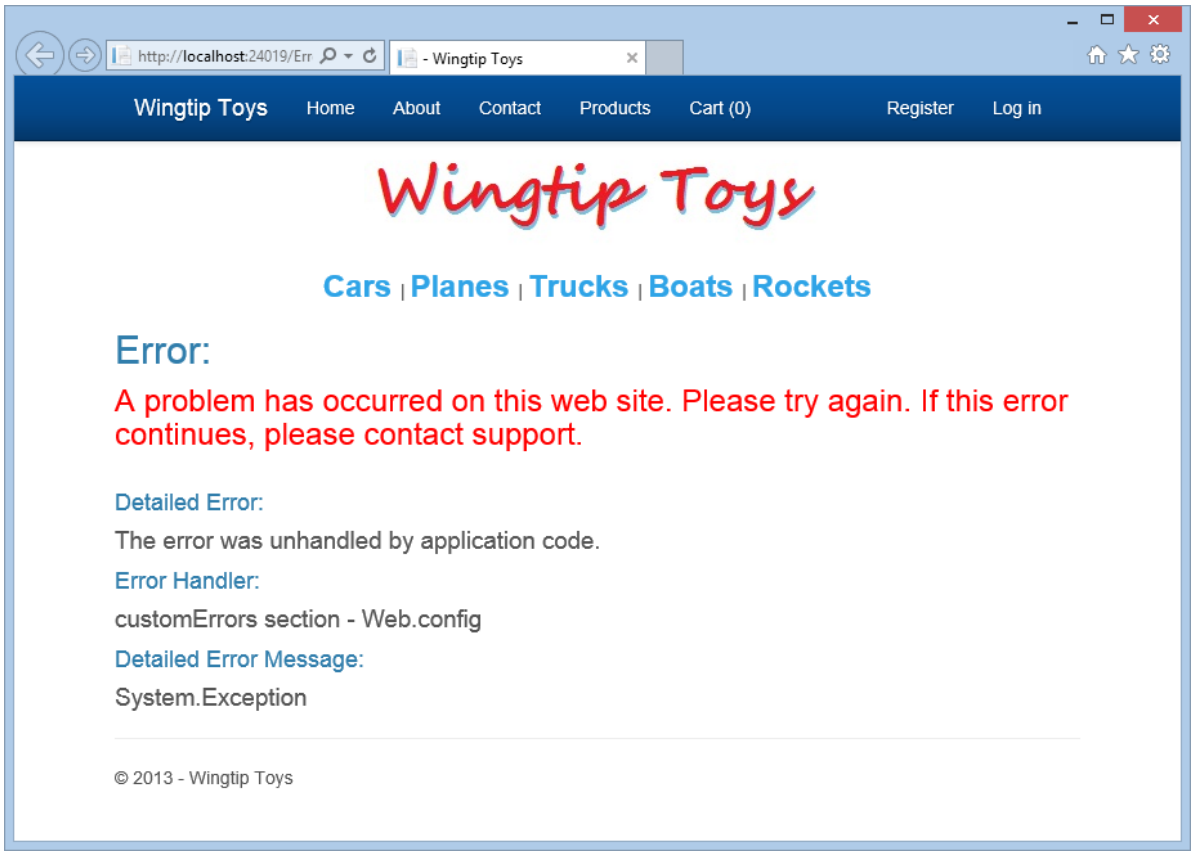
Running the Application

You can run the application to see how the application handles the exception.

1. Press **CTRL+F5** to run the Wingtip Toys sample application.
   The application throws the InvalidOperationException.

> **NOTE**
>
> You must press **CTRL+F5** to display the page without breaking into the code to view the source of the error in Visual Studio.

2. Review the *ErrorPage.aspx* displayed in the browser.



As you can see in the error details, the exception was trapped by the `customError` section in the *Web.config* file.

Adding Application-Level Error Handling

Rather than trap the exception using the `customErrors` section in the *Web.config* file, where you gain little information about the exception, you can trap the error at the application level and retrieve error details.

1. In **Solution Explorer**, find and open the *Global.asax.cs* file.

2. Add an **Application_Error** handler so that it appears as follows:

```
void Application_Error(object sender, EventArgs e)
{
  // Code that runs when an unhandled error occurs.

  // Get last error from the server
  Exception exc = Server.GetLastError();

  if (exc is HttpUnhandledException)
  {
    if (exc.InnerException != null)
    {
      exc = new Exception(exc.InnerException.Message);
      Server.Transfer("ErrorPage.aspx?handler=Application_Error%20-%20Global.asax",
          true);
    }
  }
}
```

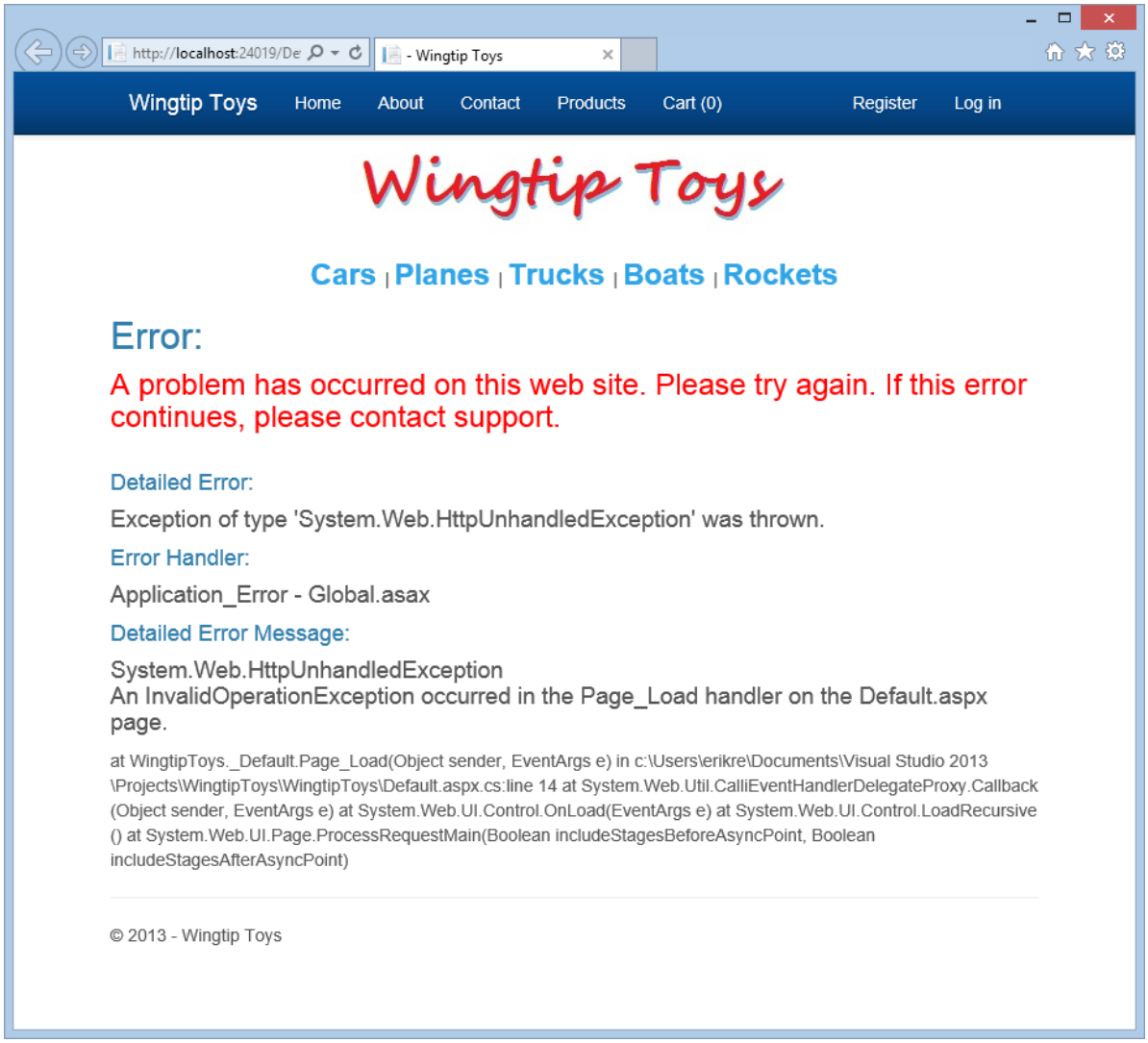
When an error occurs in the application, the `Application_Error` handler is called. In this handler, the last exception is retrieved and reviewed. If the exception was unhandled and the exception contains inner-exception details (that is, `InnerException` is not null), the application transfers execution to the error page where the exception details are displayed.

Running the Application

You can run the application to see the additional error details provided by handling the exception at the application level.

1. Press **CTRL+F5** to run the Wingtip Toys sample application.
   The application throws the `InvalidOperationException` .

2. Review the *ErrorPage.aspx* displayed in the browser.



Adding Page-Level Error Handling

You can add page-level error handling to a page either by using adding an `ErrorPage` attribute to the `@Page` directive of the page, or by adding a `Page_Error` event handler to the code-behind of a page. In this section, you will add a `Page_Error` event handler that will transfer execution to the *ErrorPage.aspx* page.

1. In **Solution Explorer**, find and open the *Default.aspx.cs* file.

2. Add a `Page_Error` handler so that the code-behind appears as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace WingtipToys
{
  public partial class _Default : Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      throw new InvalidOperationException("An InvalidOperationException " +
      "occurred in the Page_Load handler on the Default.aspx page.");
    }

    private void Page_Error(object sender, EventArgs e)
    {
      // Get last error from the server.
      Exception exc = Server.GetLastError();

      // Handle specific exception.
      if (exc is InvalidOperationException)
      {
        // Pass the error on to the error page.
        Server.Transfer("ErrorPage.aspx?handler=Page_Error%20-%20Default.aspx",
            true);
      }
    }
  }
}
```

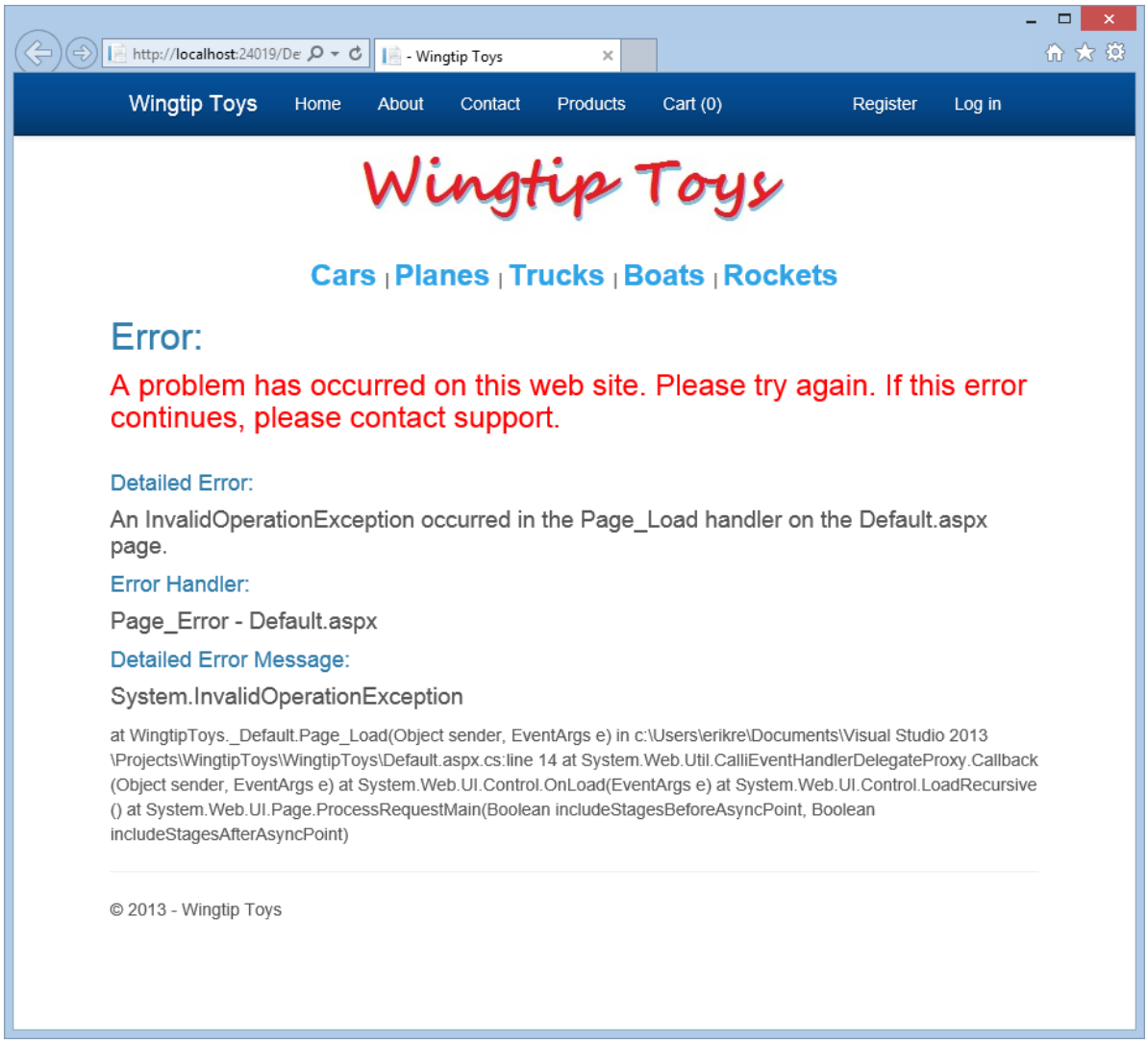
When an error occurs on the page, the `Page_Error` event handler is called. In this handler, the last exception is retrieved and reviewed. If an `InvalidOperationException` occurs, the `Page_Error` event handler transfers execution to the error page where the exception details are displayed.

Running the Application

You can run the application now to see the updated routes.

1. Press **CTRL+F5** to run the Wingtip Toys sample application.
   The application throws the `InvalidOperationException` .

2. Review the *ErrorPage.aspx* displayed in the browser.



3. Close your browser window.

Removing the Exception Used for Testing

To allow the Wingtip Toys sample application to function without throwing the exception you added earlier in this tutorial, remove the exception.

1. Open the code-behind of the *Default.aspx* page.

2. In the `Page_Load` handler, remove the code that throws the exception so that the handler appears as follows:

```
protected void Page_Load(object sender, EventArgs e)
{

}
```

Adding Code-Level Error Logging

As mentioned earlier in this tutorial, you can add try/catch statements to attempt to run a section of code and handle the first error that occurs. In this example, you will only write the error details to the error log file so that the error can be reviewed later.

1. In **Solution Explorer**, in the *Logic* folder, find and open the *PayPalFunctions.cs* file.

2. Update the `HttpCall` method so that the code appears as follows:

```
public string HttpCall(string NvpRequest)
{
  string url = pEndPointURL;

  string strPost = NvpRequest + "&" + buildCredentialsNVPString();
  strPost = strPost + "&BUTTONSOURCE=" + HttpUtility.UrlEncode(BNCode);

  HttpWebRequest objRequest = (HttpWebRequest)WebRequest.Create(url);
  objRequest.Timeout = Timeout;
  objRequest.Method = "POST";
  objRequest.ContentLength = strPost.Length;

  try
  {
    using (StreamWriter myWriter = new StreamWriter(objRequest.GetRequestStream()))
    {
      myWriter.Write(strPost);
    }
  }
  catch (Exception e)
  {
    // Log the exception.
    WingtipToys.Logic.ExceptionUtility.LogException(e, "HttpCall in PayPalFunction.cs");
  }

  //Retrieve the Response returned from the NVP API call to PayPal.
  HttpWebResponse objResponse = (HttpWebResponse)objRequest.GetResponse();
  string result;
  using (StreamReader sr = new StreamReader(objResponse.GetResponseStream()))
  {
    result = sr.ReadToEnd();
  }

  return result;
}
```

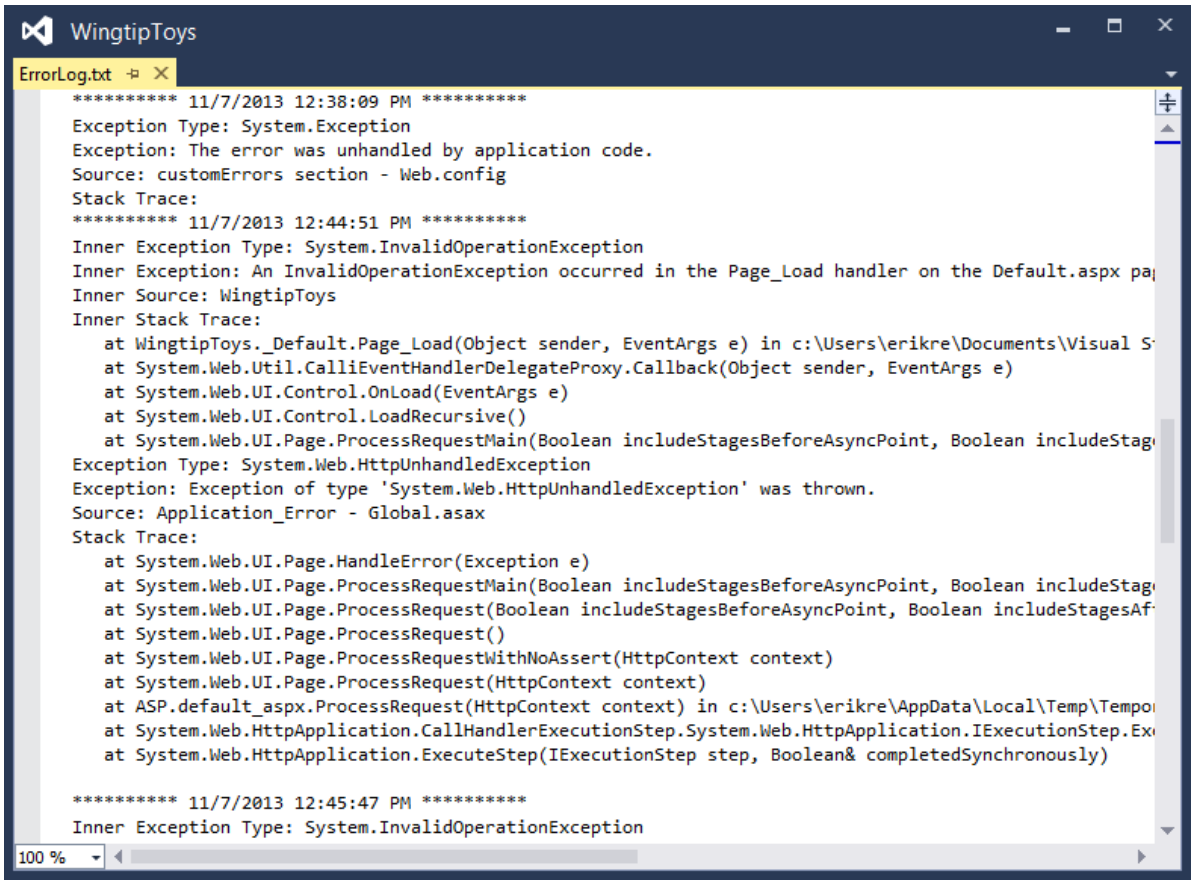
The above code calls the `LogException` method that is contained in the `ExceptionUtility` class. You added the *ExceptionUtility.cs* class file to the *Logic* folder earlier in this tutorial. The `LogException` method takes two parameters. The first parameter is the exception object. The second parameter is a string used to recognize the source of the error.

Inspecting the Error Logging Information

As mentioned previously, you can use the error log to determine which errors in your application should be fixed first. Of course, only errors that have been trapped and written to the error log will be recorded.

1. In **Solution Explorer**, find and open the *ErrorLog.txt* file in the *App_Data* folder.
   You may need to select the "**Show All Files**" option or the "**Refresh**" option from the top of **Solution Explorer** to see the *ErrorLog.txt* file.

2. Review the error log displayed in Visual Studio:



Safe Error Messages

It is **important to note** that when your application displays error messages, it should not give away information that a malicious user might find helpful in attacking your application. For example, if your application unsuccessfully tries to write in to a database, it should not display an error message that includes the user name it is using. For this reason, a generic error message in red is displayed to the user. All additional error details are only displayed to the developer on the local machine.

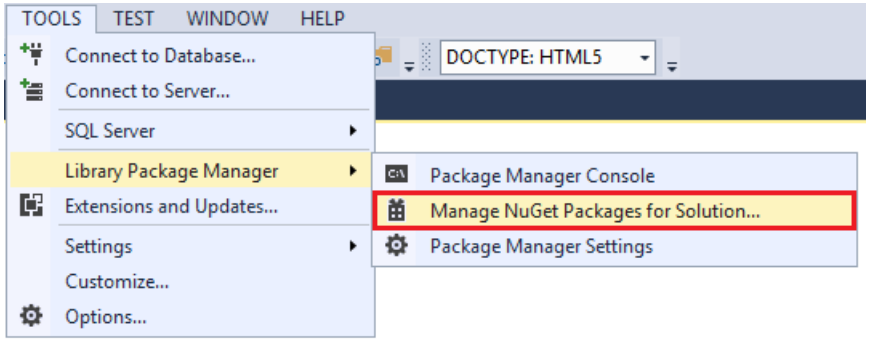
Using ELMAH

ELMAH (Error Logging Modules and Handlers) is an error logging facility that you plug into your ASP.NET application as a NuGet package. ELMAH provides the following capabilities:

- Logging of unhandled exceptions.
- A web page to view the entire log of recoded unhandled exceptions.
- A web page to view the full details of each logged exception.
- An email notification of each error at the time it occurs.
- An RSS feed of the last 15 errors from the log.

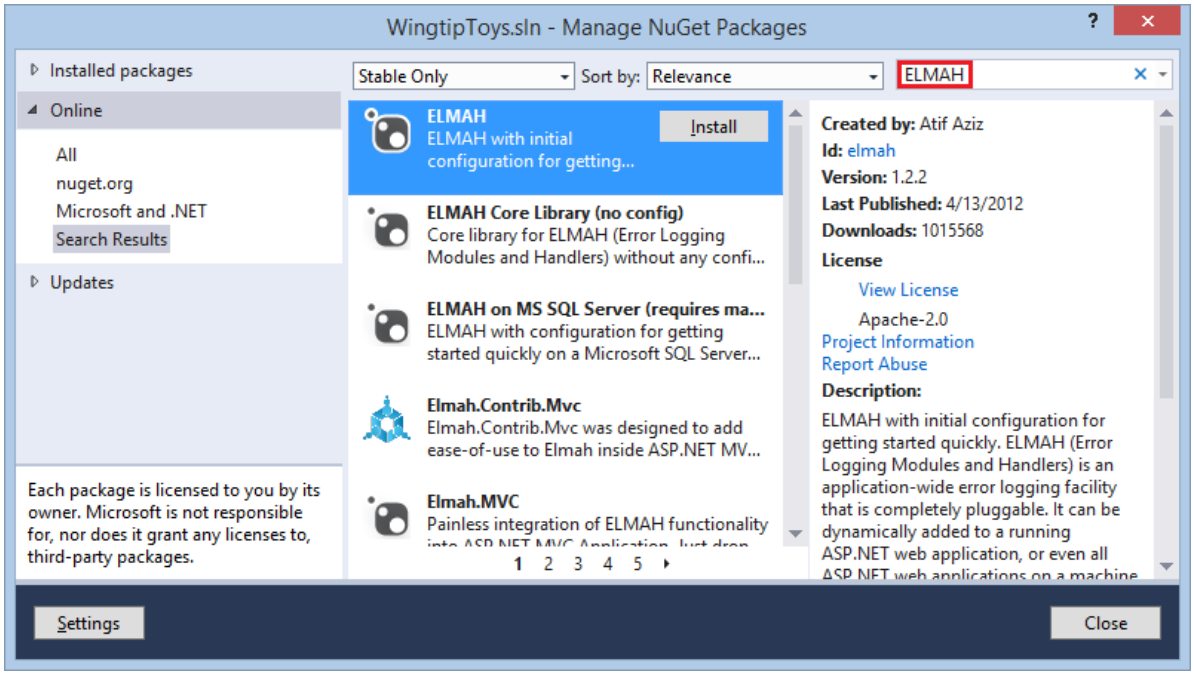
Before you can work with the ELMAH, you must install it. This is easy using the *NuGet* package installer. As mentioned earlier in this tutorial series, NuGet is a Visual Studio extension that makes it easy to install and update open source libraries and tools in Visual Studio.

1. Within Visual Studio, from the **Tools** menu, select **NuGet Package Manager** > **Manage NuGet Packages for Solution**.



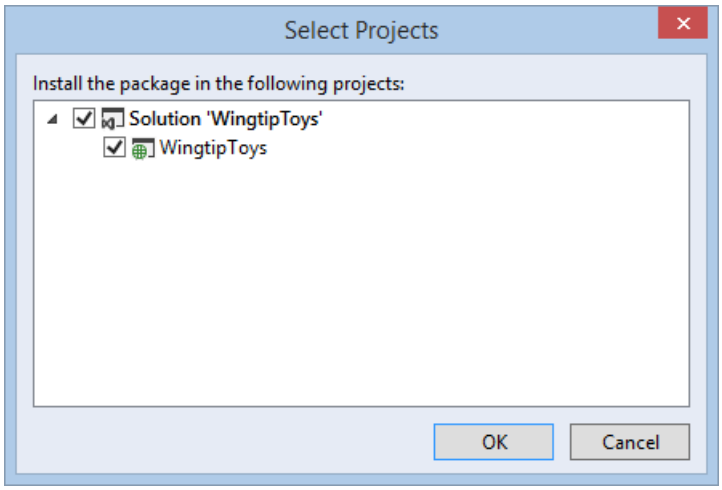
2. The **Manage NuGet Packages** dialog box is displayed within Visual Studio.

3. In the **Manage NuGet Packages** dialog box, expand **Online** on the left, and then select **nuget.org**. Then, find and install the **ELMAH** package from the list of available packages online.



4. You will need to have an internet connection to download the package.

5. In the **Select Projects** dialog box, make sure the **WingtipToys** selection is selected, and then click **OK**.



6. Click **Close** in the **Manage NuGet Packages** dialog box if needed.

7. If Visual Studio requests that you reload any open files, select "**Yes to All**".

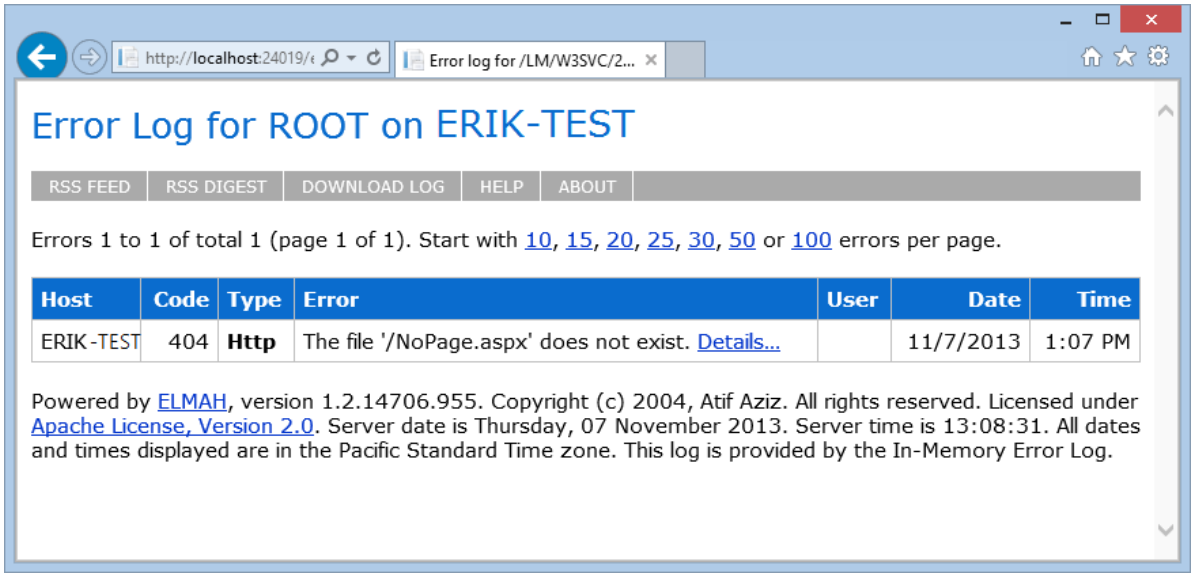
8. The ELMAH package adds entries for itself in the *Web.config* file at the root of your project. If Visual Studio asks you if you want to reload the modified *Web.config* file, click **Yes**.

ELMAH is now ready to store any unhandled errors that occur.

Viewing the ELMAH Log

Viewing the ELMAH log is easy, but first you will create an unhandled exception that will be recorded in the ELMAH log.

1. Press **CTRL+F5** to run the Wingtip Toys sample application.
2. To write an unhandled exception to the ELMAH log, navigate in your browser to the following URL (using your port number):
   `https://localhost:44300/NoPage.aspx` The error page will be displayed.
3. To display the ELMAH log, navigate in your browser to the following URL (using your port number):
   `https://localhost:44300/elmah.axd`



Summary

In this tutorial, you have learned about handling errors at the application level, the page level, and the code level. You have also learned how to log handled and unhandled errors for later review. You added the ELMAH utility to provide exception logging and notification to your application using NuGet. Additionally, you have learned about the importance of safe error messages.

Tutorial Series Conclusion

Thanks for following along. I hope this set of tutorials helped you become more familiar with ASP.NET Web Forms. If you need more information about Web Forms features available in ASP.NET 4.5 and Visual Studio 2013, see ASP.NET and Web Tools for Visual Studio 2013 Release Notes. Also, be sure to take a look at the tutorial mentioned in the **Next Steps** section and defintely try out the free Azure trial.

Thanks!
- Erik

Next Steps

Learn more about deploying your web application to Microsoft Azure, see Deploy a Secure ASP.NET Web Forms App with Membership, OAuth, and SQL Database to an Azure Web Site.

Free Trial

Microsoft Azure - Free Trial
Publishing your website to Microsoft Azure will save you time, maintenance and expense. It's a quick process to deploying your web app to Azure. When you need to maintain and monitor your web app, Azure offers a variety of tools and services. Manage data, traffic, identity, backups, messaging, media and performance in Azure. And, all of this is provided in a very cost effective approach.

Additional Resources

Logging Error Details with ASP.NET Health Monitoring
ELMAH

Acknowledgements

I would like to thank the following people who made significant contributions to the content of this tutorial series:

- Alberto Poblacion, MVP & MCT, Spain
- Alex Thissen, Netherlands (twitter: @alexthissen)
- Andre Tournier, USA
- Apurva Joshi, Microsoft
- Bojan Vrhovnik, Slovenia
- Bruno Sonnino, Brazil (twitter: @bsonnino)
- Carlos dos Santos, Brazil
- Dave Campbell, USA (twitter: @windowsdevnews)
- Jon Galloway, Microsoft (twitter: @jongalloway)
- Michael Sharps, USA (twitter: @mrsharps)
- Mike Pope
- Mitchel Sellers, USA (twitter: @MitchelSellers)
- Paul Cociuba, Microsoft
- Paulo Morgado, Portugal
- Pranav Rastogi, Microsoft
- Tim Ammann, Microsoft
- Tom Dykstra, Microsoft

Community Contributions

- Graham Mendick (@grahammendick)
  Visual Studio 2012 related code sample on MSDN: Navigation Wingtip Toys
- James Chaney (jchaney@agvance.net)
  Visual Studio 2012 related code sample on MSDN: ASP.NET 4.5 Web Forms Tutorial Series in Visual Basic
- Andrielle Azevedo - Microsoft Technical Audience Contributor (twitter: @driazevedo)
  Visual Studio 2012 translation: Iniciando com ASP.NET Web Forms 4.5 - Parte 1 - Introdução e Visão Geral

PREVIOUS

Using Visual Studio 2013 to create a Basic ASP.NET 4.5 Web Forms Page

2/20/2020 • 14 minutes to read • Edit Online

by Erik Reitan

For new web application development, we recommend Razor Pages. For more information, see Get started with Razor Pages.

This walkthrough provides you with an introduction to the Web development environment in Microsoft Visual Studio 2013 and in Microsoft Visual Studio Express 2013 for Web. This walkthrough guides you through creating a simple ASP.NET Web Forms page and illustrates the basic techniques of creating a new page, adding controls, and writing code.

Tasks illustrated in this walkthrough include:

- Creating a file system Web Forms application project.
- Familiarizing yourself with Visual Studio.
- Creating an ASP.NET page.
- Adding controls.
- Adding event handlers.
- Running and testing a page from Visual Studio.

Prerequisites

In order to complete this walkthrough, you will need:

- Microsoft Visual Studio 2013 or Microsoft Visual Studio Express 2013 for Web. The .NET Framework is installed automatically.

> **NOTE**
>
> Microsoft Visual Studio 2013 and Microsoft Visual Studio Express 2013 for Web will often be referred to as Visual Studio throughout this tutorial series.
>
> If you are using Visual Studio, this walkthrough assumes that you selected the **Web Development** collection of settings the first time that you started Visual Studio. For more information, see How to: Select Web Development Environment Settings.

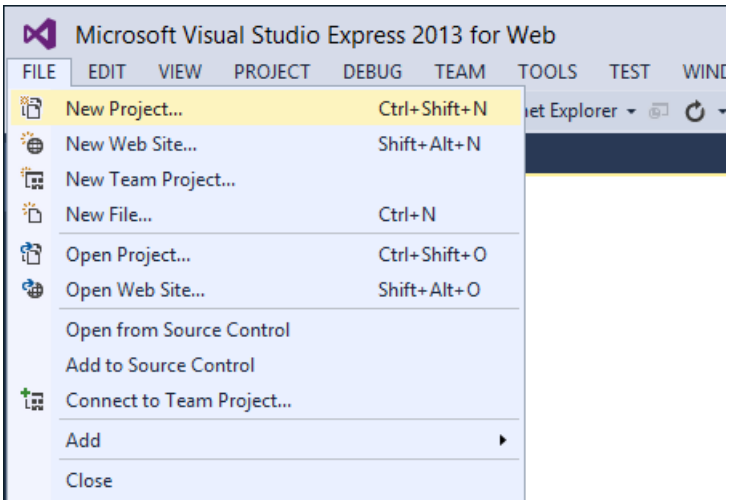
Creating a Web application project and a Page

In this part of the walkthrough, you will create a Web application project and add a new page to it. You will also add HTML text and run the page in your browser.

To create a Web application project

1. Open Microsoft Visual Studio.

2. On the **File** menu, select **New Project**.

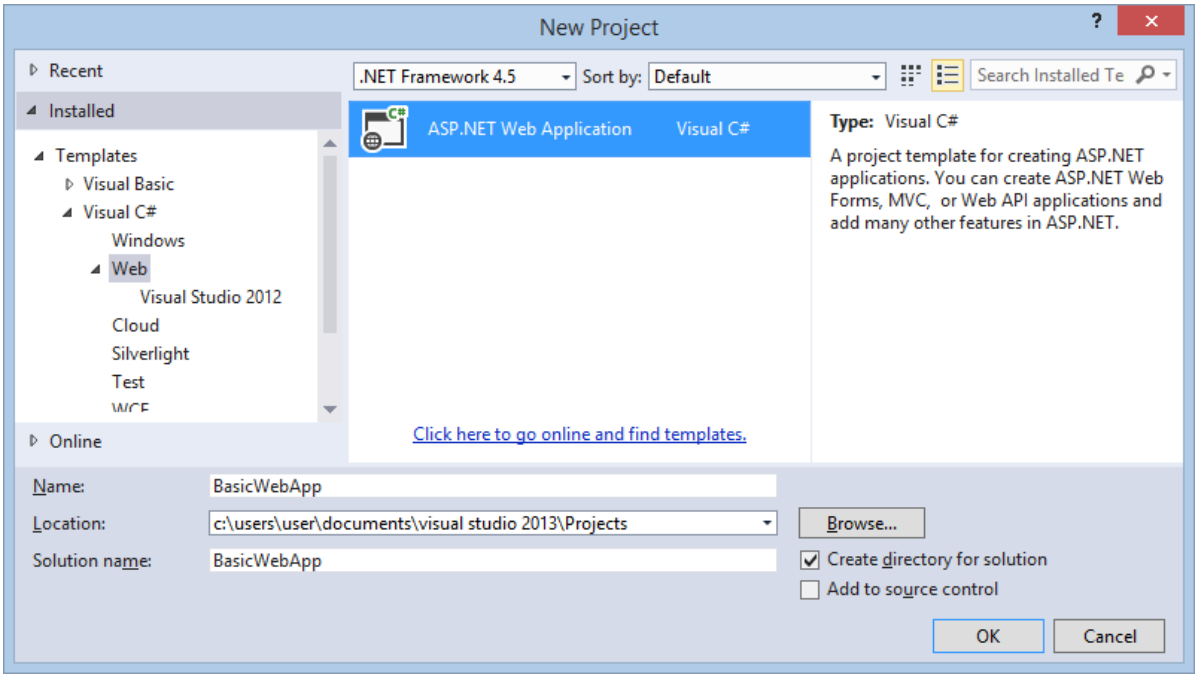


The **New Project** dialog box appears.

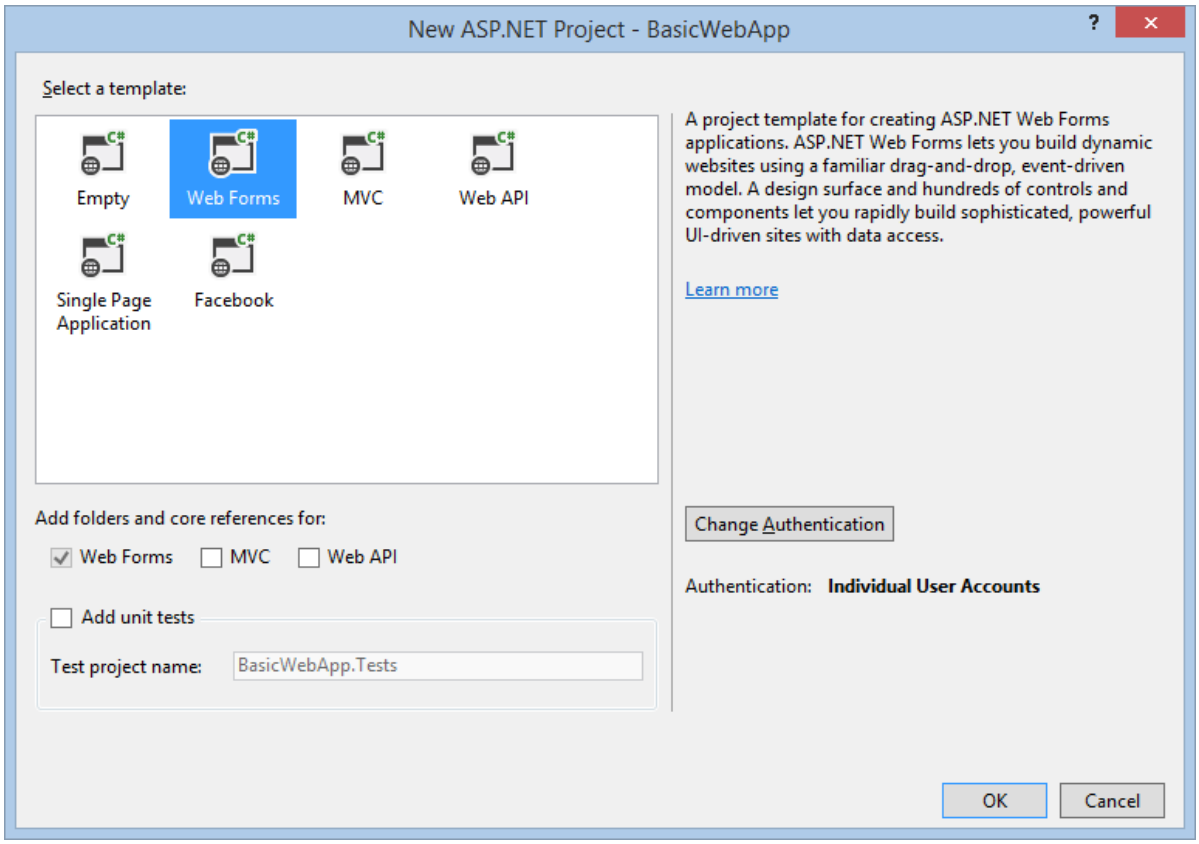
3. Select the **Templates** -> **Visual C#** -> **Web** templates group on the left.

4. Choose the **ASP.NET Web Application** template in the center column.

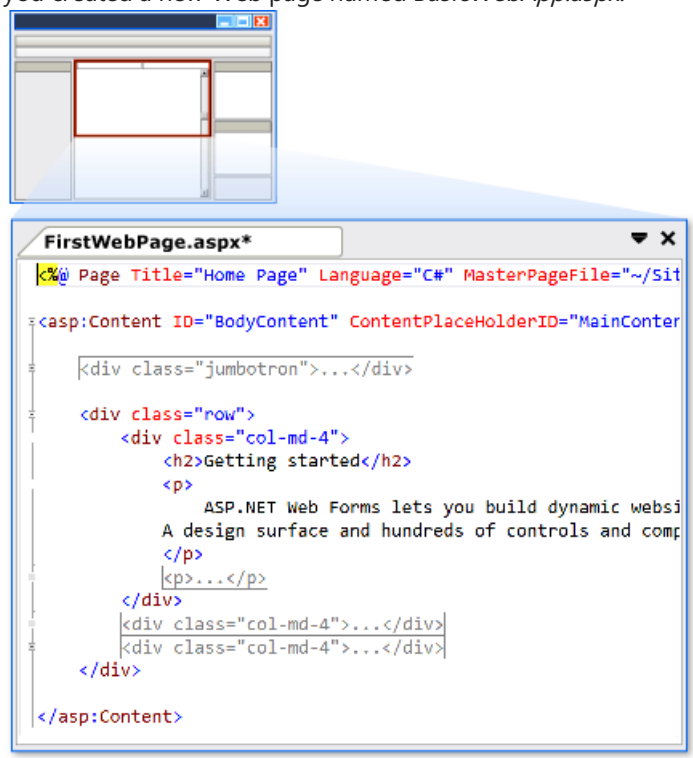
5. Name your project ***BasicWebApp*** and click the **OK** button.



6. Next, select the **Web Forms** template and click the **OK** button to create the project.



Visual Studio creates a new project that includes prebuilt functionality based on the Web Forms template. It not only provides you with a *Home.aspx* page, an *About.aspx* page, a *Contact.aspx* page, but also includes membership functionality that registers users and saves their credentials so that they can log in to your website. When a new page is created, by default Visual Studio displays the page in **Source** view, where you can see the page's HTML elements. The following illustration shows what you would see in **Source** view if you created a new Web page named *BasicWebApp.aspx*.

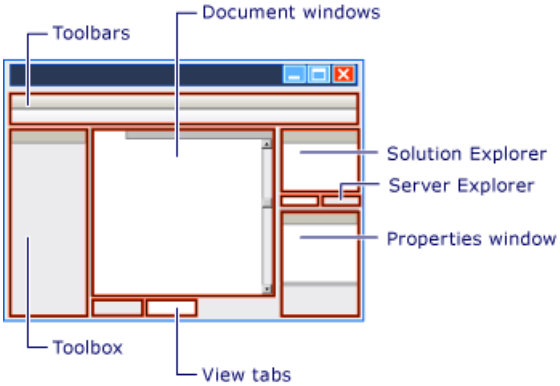


A Tour of the Visual Studio Web Development Environment

Before you proceed by modifying the page, it is useful to familiarize yourself with the Visual Studio development environment. The following illustration shows you the windows and tools that are available in Visual Studio and Visual Studio Express for Web.

> **NOTE**
> This diagram shows default windows and window locations. The **View** menu allows you to display additional windows, and to rearrange and resize windows to suit your preferences. If changes have already been made to the window arrangement, what you see will not match the illustration.

The Visual Studio environment



Familiarize yourself with the Web designer

Examine the above illustration and match the text to the following list, which describes the most commonly used windows and tools. (Not all windows and tools that you see are listed here, only those marked in the preceding illustration.)

- Toolbars. Provide commands for formatting text, finding text, and so on. Some toolbars are available only when you are working in **Design** view.
- **Solution Explorer** window. Displays the files and folders in your Web application.
- Document window. Displays the documents you are working on in tabbed windows. You can switch between documents by clicking tabs.
- **Properties** window. Allows you to change settings for the page, HTML elements, controls, and other objects.
- View tabs. Present you with different views of the same document. **Design** view is a near-WYSIWYG editing surface. **Source** view is the HTML editor for the page. **Split** view displays both the **Design** view and the **Source** view for the document. You will work with the **Design** and **Source** views later in this walkthrough. If you prefer to open Web pages in **Design** view, on the **Tools** menu, click **Options**, select the **HTML Designer** node, and change the **Start Pages In** option.
- **ToolBox**. Provides controls and HTML elements that you can drag onto your page. **Toolbox** elements are grouped by common function.
- S **erver Explorer**. Displays database connections. If Server Explorer is not visible, on the View menu, click Server Explorer.

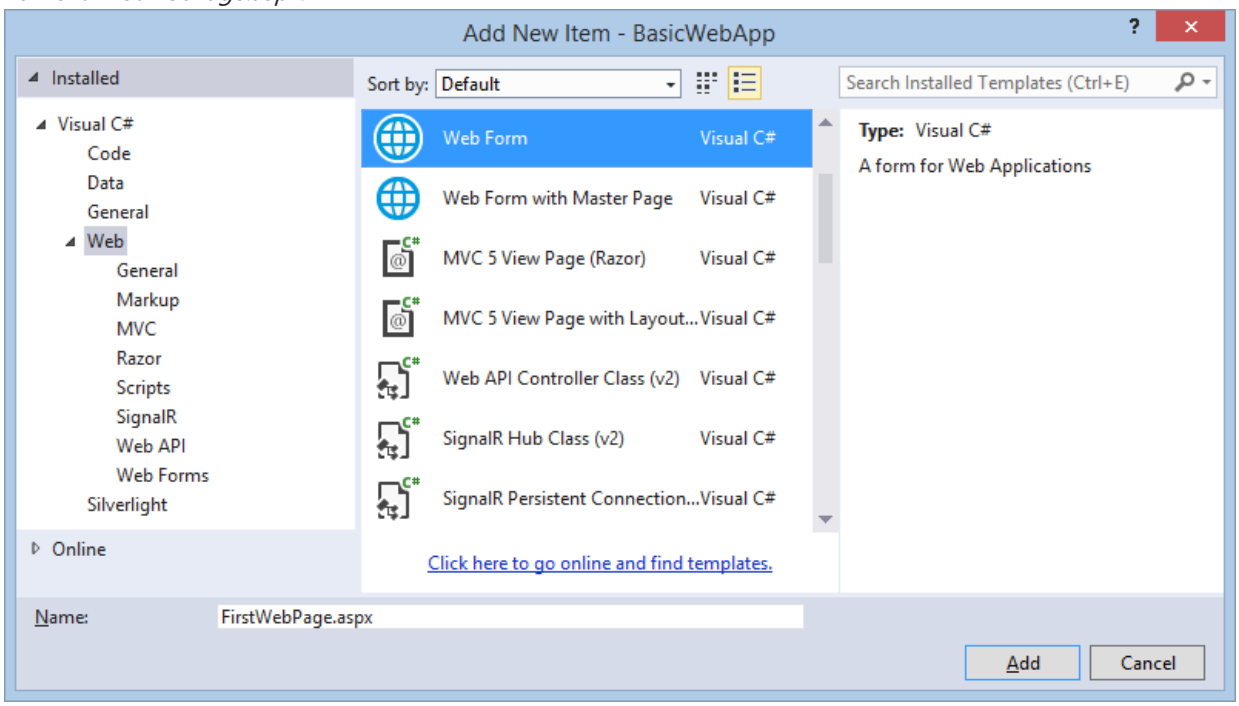
Creating a new ASP.NET Web Forms Page

When you create a new Web Forms application using the **ASP.NET Web Application** project template, Visual Studio adds an ASP.NET page (Web Forms page) named *Default.aspx*, as well as several other files and folders. You can use the *Default.aspx* page as the home page for your Web application. However, for this walkthrough, you will create and work with a new page.

To add a page to the Web application

1. Close the *Default.aspx* page. To do this, click the tab that displays the file name and then click the close option.
2. In **Solution Explorer**, right-click the Web application name (in this tutorial the application name is **BasicWebSite**), and then click **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.
3. Select the **Visual C#** -> **Web** templates group on the left. Then, select **Web Form** from the middle list and

name it *FirstWebPage.aspx*.



4. Click **Add** to add the web page to your project.
   Visual Studio creates the new page and opens it.

Adding HTML to the Page

In this part of the walkthrough, you will add some static text to the page.

To add text to the page

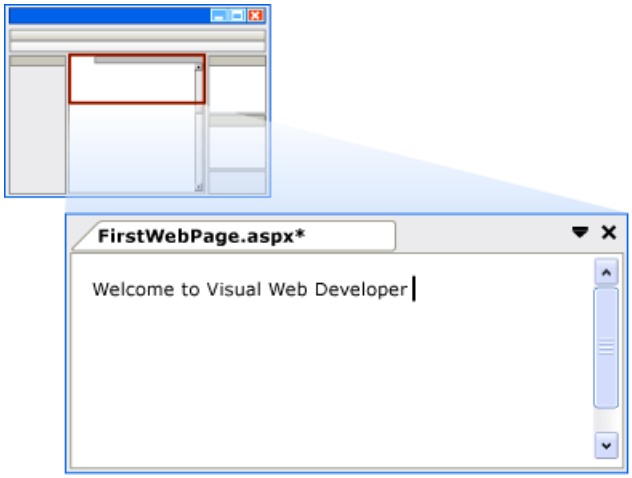
1. At the bottom of the document window, click the **Design** tab to switch to **Design** view.

   Design view displays the current page in a WYSIWYG-like way. At this point, you do not have any text or controls on the page, so the page is blank except for a dashed line that outlines a rectangle. This rectangle represents a **div** element on the page.

2. Click inside the rectangle that is outlined by a dashed line.

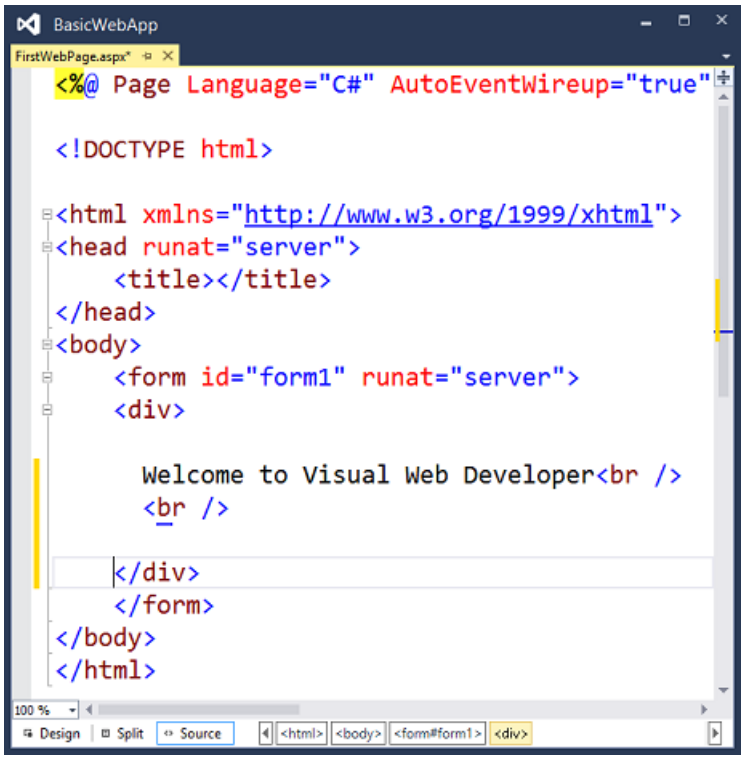
3. Type **Welcome to Visual Web Developer** and press **ENTER** twice.

   The following illustration shows the text you typed in **Design** view.



4. Switch to **Source** view.

   You can see the HTML in **Source** view that you created when you typed in **Design** view.



Running the Page

Before you proceed by adding controls to the page, you can first run it.

To run the page

1. In **Solution Explorer**, right-click *FirstWebPage.aspx* and select **Set as Start Page**.

2. Press **CTRL+F5** to run the page.

   The page is displayed in the browser. Although the page you created has a file-name extension of *.aspx*, it currently runs like any HTML page.

   To display a page in the browser you can also right-click the page in **Solution Explorer** and select **View in Browser**.

3. Close the browser to stop the Web application.

Adding and Programming Controls

You will now add server controls to the page. Server controls, such as buttons, labels, text boxes, and other familiar controls, provide typical form-processing capabilities for your Web Forms pages. However, you can program the controls with code that runs on the server, rather than the client.

You will add a Button control, a TextBox control, and a Label control to the page and write code to handle the Click event for the Button control.

To add controls to the page

1. Click the **Design** tab to switch to **Design** view.

2. Put the insertion point at the end of the **Welcome to Visual Web Developer** text and press **ENTER** five or more times to make some room in the **div** element box.

3. In the **Toolbox**, expand the **Standard** group if it is not already expanded.
   Note that you may need to expand the **Toolbox** window on the left to view it.

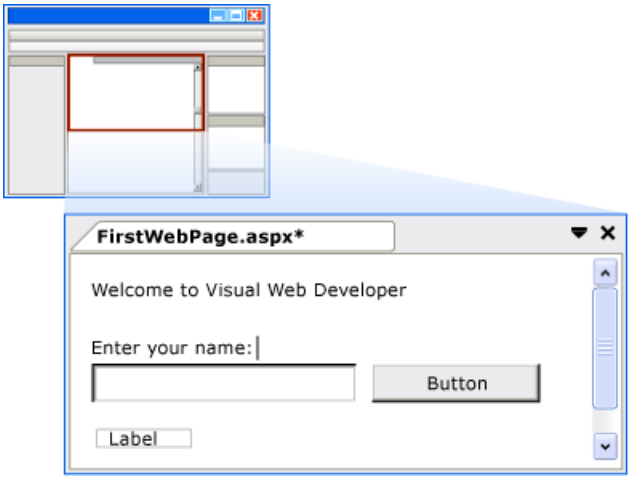
4. Drag a TextBox control onto the page and drop it in the middle of the **div** element box that has **Welcome to Visual Web Developer** in the first line.

5. Drag a Button control onto the page and drop it to the right of the TextBox control.

6. Drag a Label control onto the page and drop it on a separate line below the Button control.

7. Put the insertion point above the TextBox control, and then type **Enter your name:** .

   This static HTML text is the caption for the TextBox control. You can mix static HTML and server controls on the same page. The following illustration shows how the three controls appear in **Design** view.



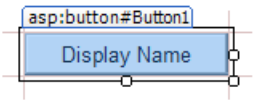
Setting Control Properties

Visual Studio offers you various ways to set the properties of controls on the page. In this part of the walkthrough, you will set properties in both **Design** view and **Source** view.

To set control properties

1. First, display the **Properties** windows by selecting from the **View** menu -> **Other Windows** -> **Properties Window**. You could alternatively select **F4** to display the **Properties** window.

2. Select the Button control, and then in the **Properties** window, set the value of **Text** to **Display Name**. The text you entered appears on the button in the designer, as shown in the following illustration.



3. Switch to **Source** view.

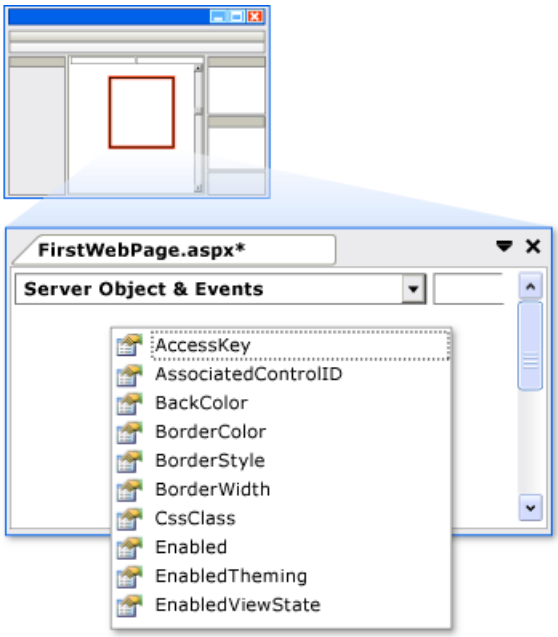
   Source view displays the HTML for the page, including the elements that Visual Studio has created for the server controls. Controls are declared using HTML-like syntax, except that the tags use the prefix **asp:** and include the attribute **runat="server"**.

   Control properties are declared as attributes. For example, when you set the Text property for the Button control, in step 1, you were actually setting the **Text** attribute in the control's markup.

   > **NOTE**
   >
   > All the controls are inside a **form** element, which also has the attribute **runat="server"**. The **runat="server"** attribute and the **asp:** prefix for control tags mark the controls so that they are processed by ASP.NET on the server when the page runs. Code outside of **<form runat="server">** and **<script runat="server">** elements is sent unchanged to the browser, which is why the ASP.NET code must be inside an element whose opening tag contains the **runat="server"** attribute.

4. Next, you will add an additional property to the Label control. Put the insertion point directly after **asp:Label** in the **<asp:Label>** tag, and then press **SPACEBAR**.

A drop-down list appears that displays the list of available properties you can set for a Label control. This feature, referred to as **IntelliSense**, helps you in **Source** view with the syntax of server controls, HTML elements, and other items on the page. The following illustration shows the **IntelliSense** drop-down list for the Label control.



5. Select **ForeColor** and then type an equal sign.

   IntelliSense displays a list of colors.

   > **NOTE**
   >
   > You can display an **IntelliSense** drop-down list at any time by pressing **CTRL+J** when viewing code.

6. Select a color for the **Label** control's text. Make sure you select a color that is dark enough to read against a white background.

   The **ForeColor** attribute is completed with the color that you have selected, including the closing quotation mark.

Programming the Button Control

For this walkthrough, you will write code that reads the name that the user enters into the text box and then displays the name in the Label control.

Add a default button event handler

1. Switch to **Design** view.

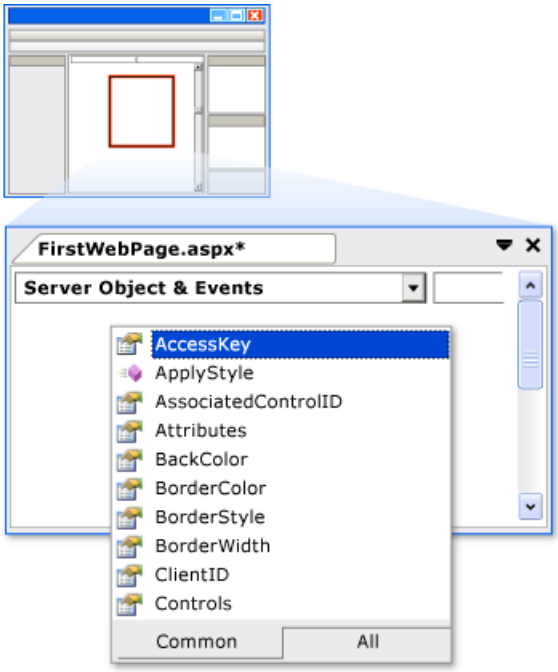
2. Double-click the Button control.

   By default, Visual Studio switches to a code-behind file and creates a skeleton event handler for the Button control's default event, the Click event. The code-behind file separates your UI markup (such as HTML) from your server code (such as C#).
   The cursor is positioned to added code for this event handler.

   > **NOTE**
   >
   > Double-clicking a control in **Design** view is just one of several ways you can create event handlers.

3. Inside the **Button1_Click** event handler, type **Label1** followed by a period (.).

When you type the period after the **ID** of the label (**Label1**), Visual Studio displays a list of available members for the Label control, as shown in the following illustration. A member commonly a property, method, or event.



4. Finish the **Click** event handler for the button so that it reads as shown in the following code example.

```
protected void Button1_Click(object sender, System.EventArgs e)
{
    Label1.Text = TextBox1.Text + ", welcome to Visual Studio!";
}
```

```
Protected Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs)
    Label1.Text = Textbox1.Text & ", welcome to Visual Studio!"
End Sub
```

5. Switch back to viewing the **Source** view of your HTML markup by right-clicking *FirstWebPage.aspx* in the **Solution Explorer** and selecting **View Markup**.

6. Scroll to the **<asp:Button>** element. Note that the **<asp:Button>** element now has the attribute **onclick="Button1_Click"**.

   This attribute binds the button's Click event to the handler method you coded in the previous step.

   Event handler methods can have any name; the name you see is the default name created by Visual Studio. The important point is that the name used for the **OnClick** attribute in the HTML must match the name of a method defined in the code-behind.

Running the Page

You can now test the server controls on the page.

To run the page

1. Press **CTRL+F5** to run the page in the browser. If an error occurs, recheck the steps above.

2. Enter a name into the text box and click the **Display Name** button.

   The name you entered is displayed in the Label control. Note that when you click the button, the page is posted to the Web server. ASP.NET then recreates the page, runs your code (in this case, the Button control's

Click event handler runs), and then sends the new page to the browser. If you watch the status bar in the browser, you can see that the page is making a round trip to the Web server each time you click the button.

3. In the browser, view the source of the page you are running by right-clicking on the page and selecting **View source**.

   In the page source code, you see HTML without any server code. Specifically, you do not see the **<asp:>** elements that you were working with in **Source** view. When the page runs, ASP.NET processes the server controls and renders HTML elements to the page that perform the functions that represent the control. For example, the **<asp:Button>** control is rendered as the HTML **<input type="submit">** element.

4. Close the browser.

Working with Additional Controls

In this part of the walkthrough, you will work with the Calendar control, which displays dates a month at a time. The Calendar control is a more complex control than the button, text box, and label you have been working with and illustrates some further capabilities of server controls.

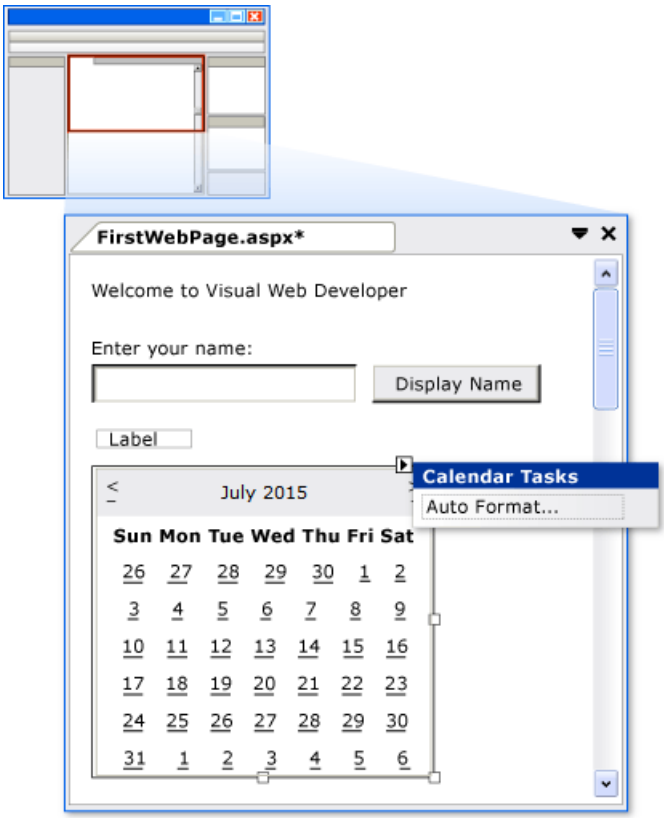
In this section, you will add a System.Web.UI.WebControls.Calendar control to the page and format it.

To add a Calendar control

1. In Visual Studio, switch to **Design** view.

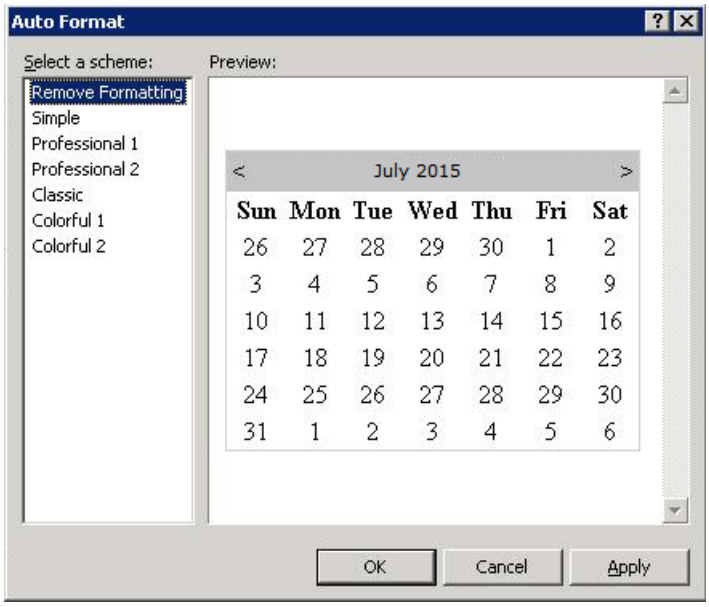
2. From the **Standard** section of the **Toolbox**, drag a Calendar control onto the page and drop it below the **div** element that contains the other controls.

   The calendar's smart tag panel is displayed. The panel displays commands that make it easy for you to perform the most common tasks for the selected control. The following illustration shows the Calendar control as rendered in **Design** view.



3. In the smart tag panel, choose **Auto Format**.

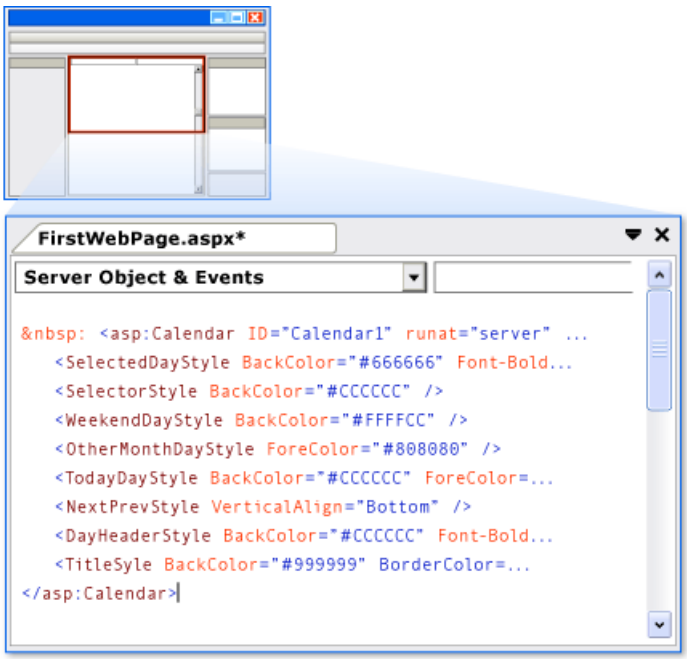
   The **Auto Format** dialog box is displayed, which allows you to select a formatting scheme for the calendar. The following illustration shows the **Auto Format** dialog box for the Calendar control.



4. From the **Select a scheme** list, select **Simple** and then click **OK**.

5. Switch to **Source** view.

   You can see the **<asp:Calendar>** element. This element is much longer than the elements for the simple controls you created earlier. It also includes subelements, such as **<WeekEndDayStyle>**, which represent various formatting settings. The following illustration shows the Calendar control in **Source** view. (The exact markup that you see in **Source** view might differ slightly from the illustration.)



Programming the Calendar Control

In this section, you will program the Calendar control to display the currently selected date.

To program the Calendar control

1. In **Design** view, double-click the Calendar control.

   A new event handler is created and displayed in the code-behind file named *FirstWebPage.aspx.cs*.

2. Finish the SelectionChanged event handler with the following code.

```
protected void Calendar1_SelectionChanged(object sender, System.EventArgs e)
{
    Label1.Text = Calendar1.SelectedDate.ToLongDateString();
}
```

```
Protected Sub Calendar1_SelectionChanged(ByVal sender As Object, ByVal e As System.EventArgs)
    Label1.Text = Calendar1.SelectedDate.ToLongDateString()
End Sub
```

The above code sets the text of the label control to the selected date of the calendar control.

Running the Page

You can now test the calendar.

To run the page

1. Press **CTRL+F5** to run the page in the browser.
2. Click a date in the calendar.

   The date you clicked is displayed in the Label control.
3. In the browser, view the source code for the page.

   Note that the Calendar control has been rendered to the page as a **table**, with each day as a **td** element.
4. Close the browser.

Next Steps

This walkthrough has illustrated the basic features of the Visual Studio page designer. Now that you understand how to create and edit a Web Forms page in Visual Studio, you might want to explore other features. For example, you might want to do the following:

- Learn more about ASP.NET Web Forms by following the step-by-step tutorial series Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013.
- Learn more about Cascading style sheets (CSS). For details, see Working with CSS Overview.

Code Editing ASP.NET Web Forms in Visual Studio 2013

2/20/2020 • 8 minutes to read • Edit Online

by Erik Reitan

In many ASP.NET Web Form pages, you write code in Visual Basic, C#, or another language. The code editor in Visual Studio can help you write code quickly while helping you avoid errors. In addition, the editor provides ways for you to create reusable code to help reduce the amount of work you need to do.

This walkthrough illustrates various features of the Visual Studio code editor.

During this walkthrough, you will learn how to:

- Correct inline coding errors.
- Refactor and rename code.
- Rename variables and objects.
- Insert code snippets.

Prerequisites

In order to complete this walkthrough, you will need:

- Microsoft Visual Studio 2013 or Microsoft Visual Studio Express 2013 for Web. The .NET Framework is installed automatically.

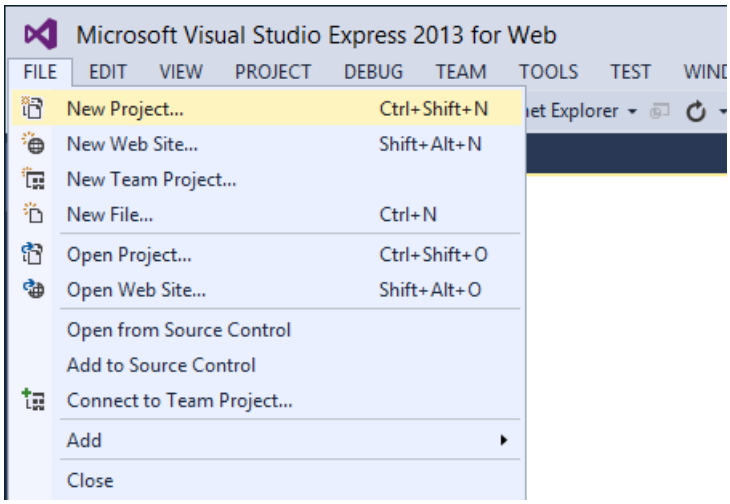
> **NOTE**
>
> Microsoft Visual Studio 2013 and Microsoft Visual Studio Express 2013 for Web will often be referred to as Visual Studio throughout this tutorial series.
>
> If you are using Visual Studio, this walkthrough assumes that you selected the **Web Development** collection of settings the first time that you started Visual Studio. For more information, see How to: Select Web Development Environment Settings.

Creating a Web application project and a Page

In this part of the walkthrough, you will create a Web application project and add a new page to it.

To create a Web application project

1. Open Microsoft Visual Studio.
2. On the **File** menu, select **New Project**.

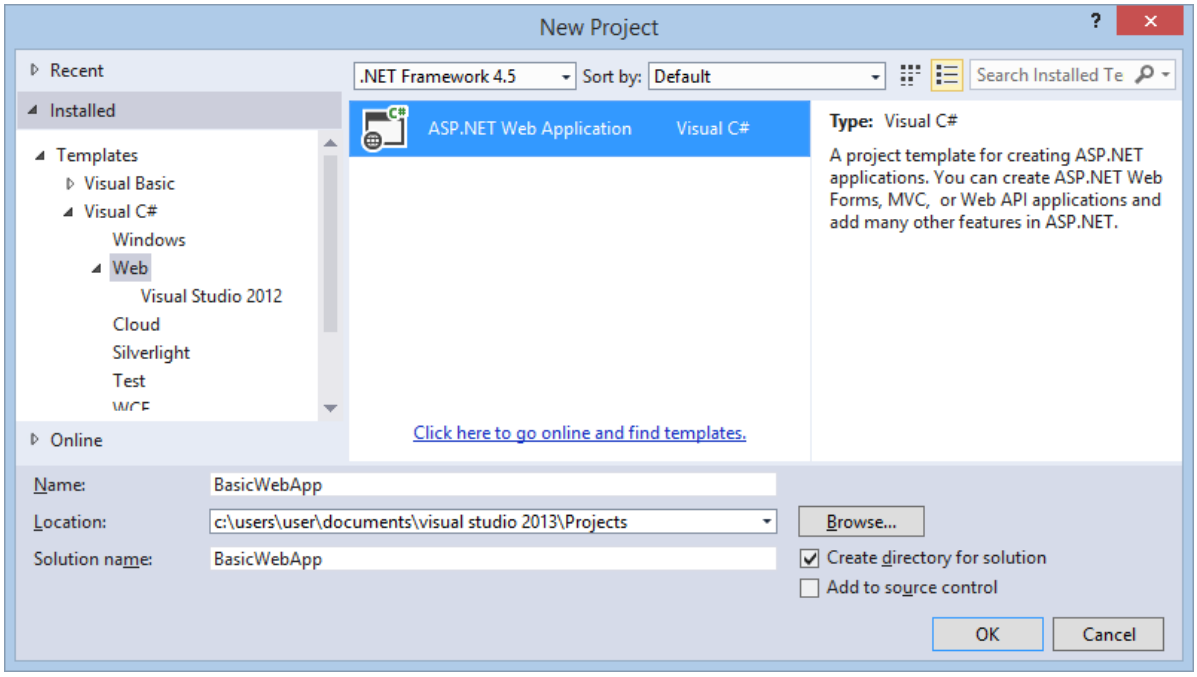


The **New Project** dialog box appears.

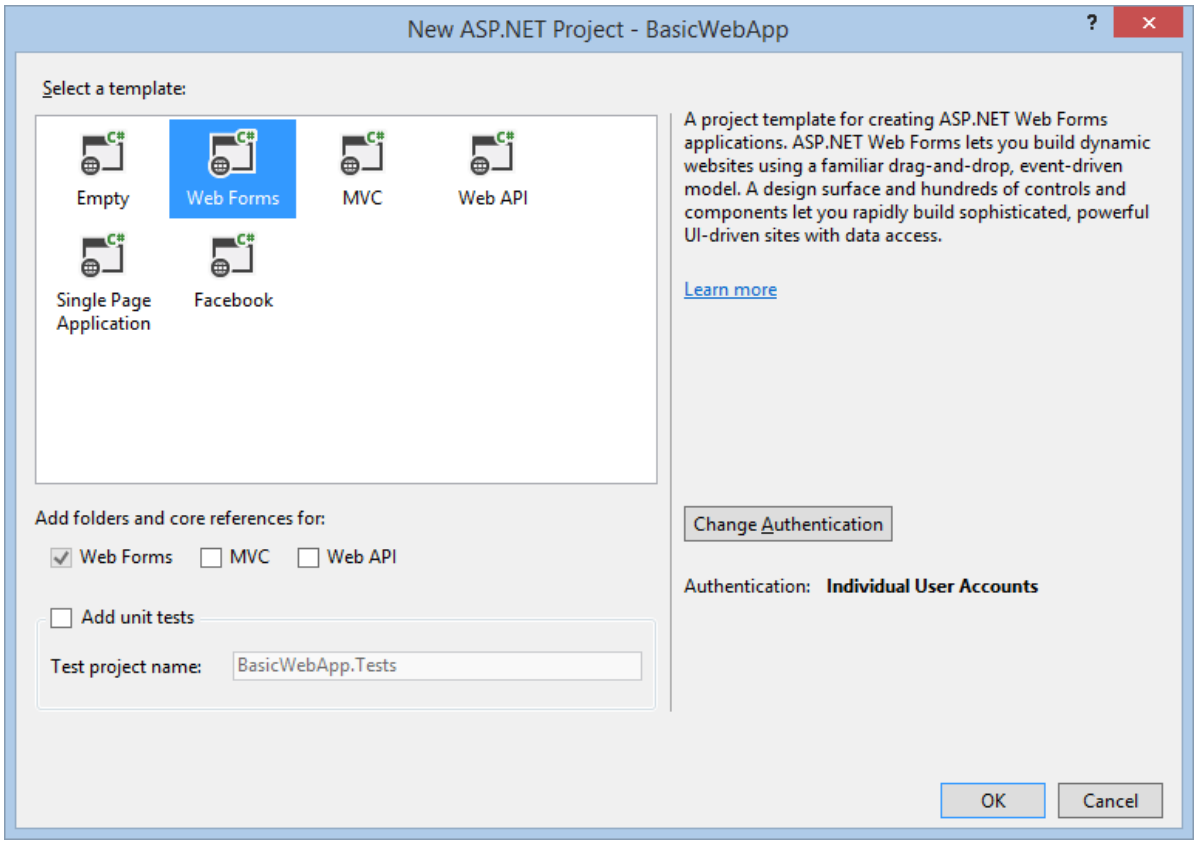
3. Select the **Templates** -> **Visual C#** -> **Web** templates group on the left.

4. Choose the **ASP.NET Web Application** template in the center column.

5. Name your project ***BasicWebApp*** and click the **OK** button.



6. Next, select the **Web Forms** template and click the **OK** button to create the project.



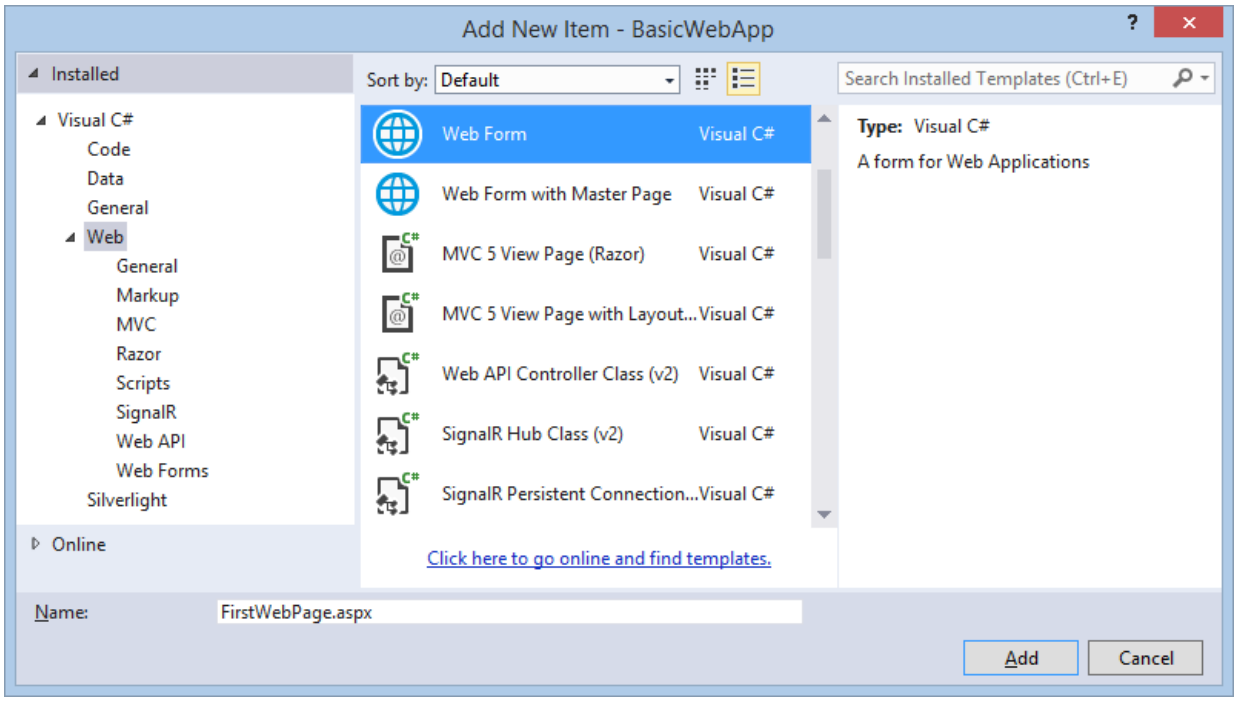
Visual Studio creates a new project that includes prebuilt functionality based on the Web Forms template.

Creating a new ASP.NET Web Forms Page

When you create a new Web Forms application using the **ASP.NET Web Application** project template, Visual Studio adds an ASP.NET page (Web Forms page) named *Default.aspx*, as well as several other files and folders. You can use the *Default.aspx* page as the home page for your Web application. However, for this walkthrough, you will create and work with a new page.

To add a page to the Web application

1. In **Solution Explorer**, right-click the Web application name (in this tutorial the application name is **BasicWebSite**), and then click **Add** -> **New Item**.
   The **Add New Item** dialog box is displayed.
2. Select the **Visual C#** -> **Web** templates group on the left. Then, select **Web Form** from the middle list and name it *FirstWebPage.aspx*.



3. Click **Add** to add the Web Forms page to your project.
   Visual Studio creates the new page and opens it.
4. Next, set this new page as the default startup page. In **Solution Explorer**, right-click the new page named *FirstWebPage.aspx* and select **Set As Start Page**. The next time you run this application to test our progress, you will automatically see this new page in the browser.

Correcting Inline Coding Errors

The code editor in Visual Studio helps you to avoid errors as you write code, and if you have made an error, the code editor helps you to correct the error. In this part of the walkthrough, you will write a line of code that illustrate the error correction features in the editor.

To correct simple coding errors in Visual Studio

1. In **Design** view, double-click the blank page to create a handler for the **Load** event for the page.
   You are using the event handler only as a place to write some code.

2. Inside the handler, type the following line that contains an error and press **ENTER**:

   ```
   string myStr = "Loading..."
   ```

   When you press **ENTER**, the code editor places green and red underlines (commonly call "squiggly" lines) under areas of the code that have issues. A green underline indicates a warning. A red underline indicates an error that you must fix.

   Hold the mouse pointer over `myStr` to see a tooltip that tells you about the warning. Also, hold your mouse pointer over the red underline to see the error message.

   The following image shows the code with the underlines.

   ```
   protected void Page_Load(object sender, EventArgs e)
   {
     string str1 = "Loading..."
   ```

   The error must be fixed by adding a semicolon `;` to the end of the line. The warning simply notifies you that you haven't used the `myStr` variable yet.

> **NOTE**
> You view your current code formatting settings in Visual Studio by selecting **Tools** -> **Options** -> **Fonts and Colors**.

Refactoring and Renaming

Refactoring is a software methodology that involves restructuring your code to make it easier to understand and to maintain, while preserving its functionality. A simple example might be that you write code in an event handler to get data from a database. As you develop your page, you discover that you need to access the data from several different handlers. Therefore, you refactor the page's code by creating a data-access method in the page and inserting calls to the method in the handlers.

The code editor includes tools to help you perform various refactoring tasks. In this walkthrough, you will work with two refactoring techniques: renaming variables and extracting methods. Other refactoring options include encapsulating fields, promoting local variables to method parameters, and managing method parameters. The availability of these refactoring options depends on the location in the code.

Refactoring Code

A common refactoring scenario is to create (extract) a method from code that is inside another member, such as a method. This reduces the size of the original member and makes the extracted code reusable.

In this part of the walkthrough, you will write some simple code, and then extract a method from it. Refactoring is supported for C#, so you will create a page that uses C# as its programming language.

To extract a method in a C# page

1. Switch to **Design** view.
2. In the **Toolbox**, from the **Standard** tab, drag a Button control onto the page.
3. Double-click the **Button** control to create a handler for its Click event, and then add the following highlighted code:

```
protected void Button1_Click(object sender, EventArgs e)
{

    System.Collections.ArrayList alist =
        new System.Collections.ArrayList();
    int i;
    string arrayValue;
    for(i=0; i<5; i++)
    {
        arrayValue = "i = " + i.ToString();
        alist.Add(arrayValue);
    }
    for(i=0; i<alist.Count; i++)
    {
        Response.Write("<br />" + alist[i]);
    };
}
```

   The code creates an **ArrayList** object, uses a loop to load it with values, and then uses another loop to display the contents of the **ArrayList** object.

4. Press **CTRL+F5** to run the page, and then click the **button** to make sure that you see the following output:

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

5. Return to the code editor, and then select the following lines in the event handler.

```
for(i=0; i<alist.Count; i++)
{
      Response.Write("<br />" + alist[i]);
}
```

6. Right-click the selection, click **Refactor**, and then choose **Extract Method**.

   The **Extract Method** dialog box appears.

7. In the **New Method Name** box, type **DisplayArray**, and then click **OK**.

   The code editor creates a new method named `DisplayArray`, and puts a call to the new method in the **Click** handler where the loop was originally.

```
protected void Button1_Click(object sender, EventArgs e)
{
    System.Collections.ArrayList alist =
        new System.Collections.ArrayList();
    int i;
    string arrayValue;
    for(i=0; i<5; i++)
    {
        arrayValue = "i = " + i.ToString();
        alist.Add(arrayValue);
    }
    i = DisplayArray(alist, i);
}
```

8. Press **CTRL+F5** to run the page again, and click the **button**.

   The page functions the same as it did before. The `DisplayArray` method can now be call from anywhere in the page class.

Renaming Variables

When you work with variables, as well as objects, you might want to rename them after they are already referenced in your code. However, renaming variables and objects can cause the code to break if you miss renaming one of the references. Therefore, you can use refactoring to perform the renaming.

To use refactoring to rename a variable

1. In the **Click** event handler, locate the following line:

```
System.Collections.ArrayList alist =
    new System.Collections.ArrayList;
```

2. Right-click the variable name `alist`, choose **Refactor**, and then choose **Rename**.

   The **Rename** dialog box appears.

3. In the **New name** box, type **ArrayList1** and make sure the **Preview reference changes** checkbox has

been selected. Then click **OK**.

The **Preview Changes** dialog box appears, and displays a tree that contains all references to the variable that you are renaming.

4. Click **Apply** to close the **Preview Changes** dialog box.

   The variables that refer specifically to the instance that you selected are renamed. Note, however, that the variable `alist` in the following line is not renamed.

   ```
   private int DisplayArray(System.Collections.ArrayList alist,
       int i)
   ```

   The variable `alist` in this line is not renamed because it does not represent the same value as the variable `alist` that you renamed. The variable `alist` in the `DisplayArray` declaration is a local variable for that method. This illustrates that using refactoring to rename variables is different than simply performing a find-and-replace action in the editor; refactoring renames variables with knowledge of the semantics of the variable that it is working with.

Inserting Snippets

Because there are many coding tasks that Web Forms developers frequently need to perform, the code editor provides a library of snippets, or blocks of prewritten code. You can insert these snippets into your page.

Each language that you use in Visual Studio has slight differences in the way you insert code snippets. For information about inserting snippets, see Visual Basic IntelliSense Code Snippets. For information about inserting snippets in Visual C#, see Visual C# Code Snippets.

Next Steps

This walkthrough has illustrated the basic features of the Visual Studio 2010 code editor for correcting errors in your code, refactoring code, renaming variables, and inserting code snippets into your code. Additional features in the editor can make application development fast and easy. For example, you might want to:

- Learn more about the features of IntelliSense, such as modifying IntelliSense options, managing code snippets, and searching for code snippets online. For more information, see Using IntelliSense.
- Learn how to create your own code snippets. For more information, see Creating and Using IntelliSense Code Snippets
- Learn more about the Visual Basic-specific features of IntelliSense code snippets, such as customizing the snippets and troubleshooting. For more information, see Visual Basic IntelliSense Code Snippets
- Learn more about the C#-specific features of IntelliSense, such as refactoring and code snippets. For more information, see Visual C# IntelliSense.

Using Page Inspector for Visual Studio 2012 in ASP.NET Web Forms

2/20/2020 • 9 minutes to read • Edit Online

by Tim Ammann

Page Inspector for Visual Studio 2012 is a web development tool with an integrated browser. Select any element in the integrated browser, and Page Inspector instantly highlights the element's source and CSS. You can browse any page in your application, quickly find the sources of rendered markup, and use browser tools right within the Visual Studio environment.

This tutorial shows how to enable Inspection Mode and then quickly locate and edit CSS rules and text within your web project. The tutorial uses a Web Forms Application Project, but you can also use Page Inspector for Web Site projects and MVC applications.

The tutorial has the following sections:

Prerequisites

Create a Web Application

Use Page Inspector to View the Application

Enable Inspection Mode

Use Page Inspector to Make Changes to Markup

Inspection Mode and the HTML Window

Preview CSS Changes in the Styles Window

CSS Auto Sync

Using the CSS Color Picker

Prerequisites

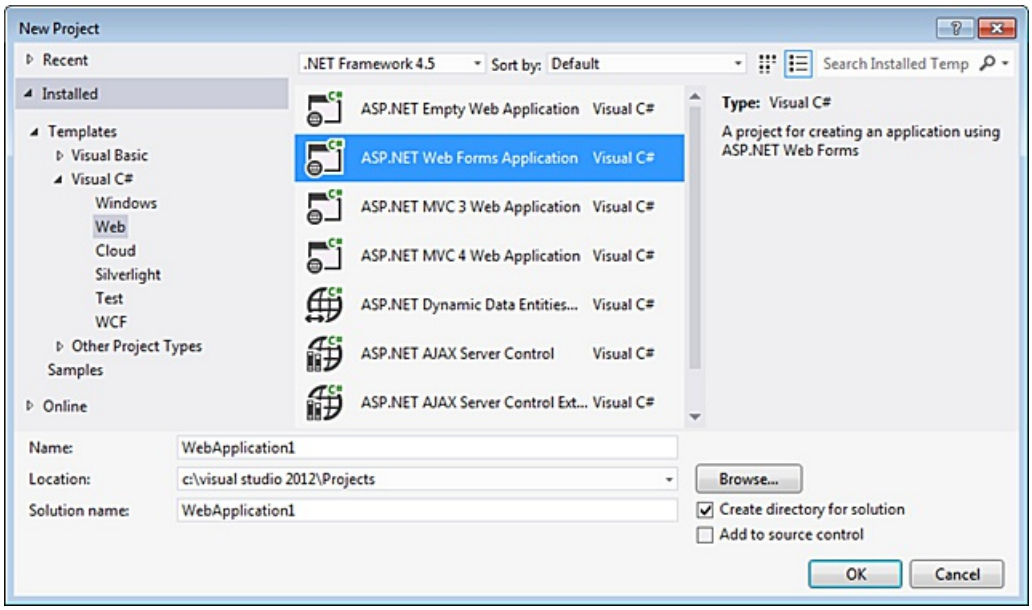
- Visual Studio 2012 or Visual Studio Express 2012 for Web.

> **NOTE**
>
> To get the latest version of Page Inspector, use Web Platform Installer to install the Azure SDK for .NET 2.0.

Page Inspector is bundled with Microsoft Web Developer Tools. The latest version is 1.3. To check which version you have, run Visual Studio and select **About Microsoft Visual Studio** from the **Help** menu.

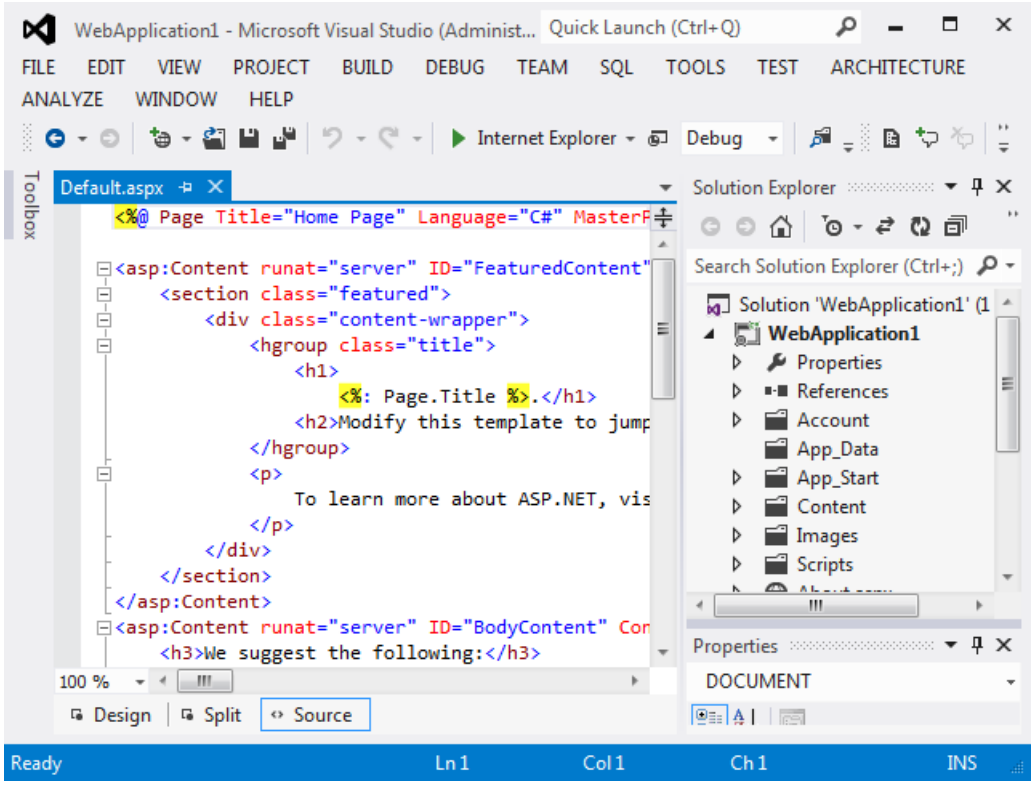
Create a Web Application

First, you will create a web application that you will use Page Inspector with. In Visual Studio, choose **File** > **New Project**. On the left, expand **Visual C#**, select **Web**, and then select **ASP.NET Web Forms Application**.



Click **OK**.

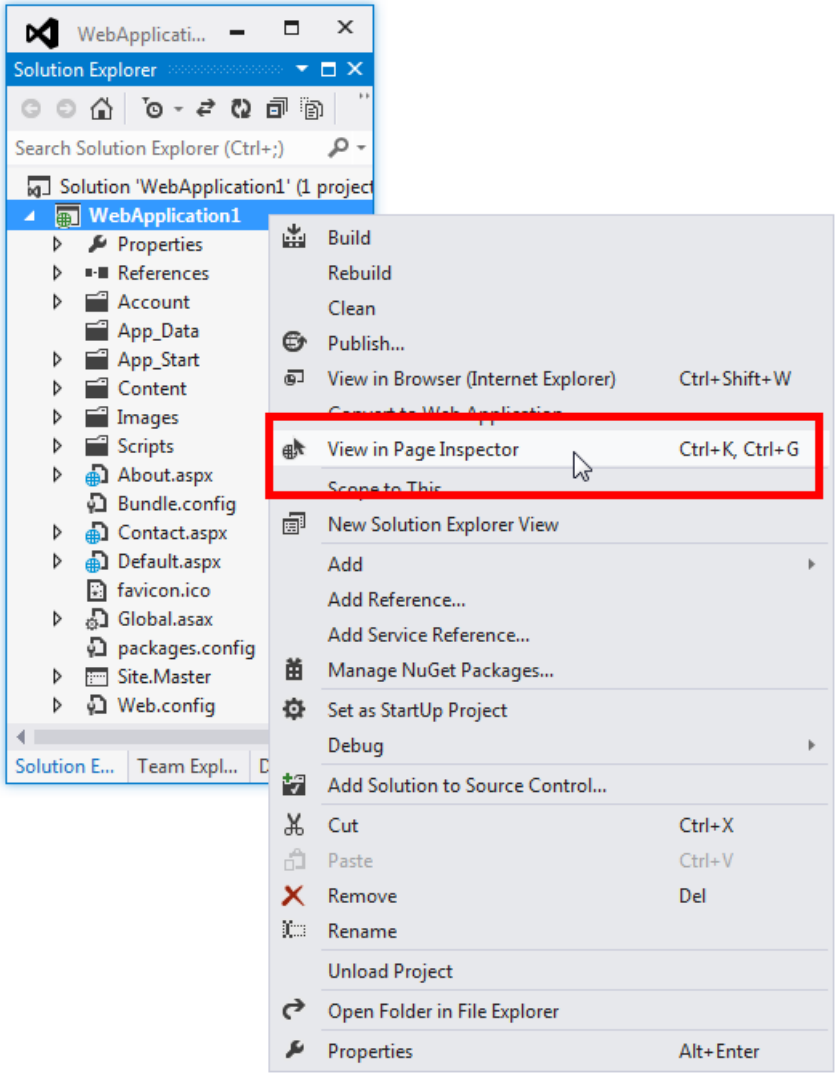
The application opens in **Source** view.



Now that you have an application to work with, you can use Page Inspector to examine and modify it.

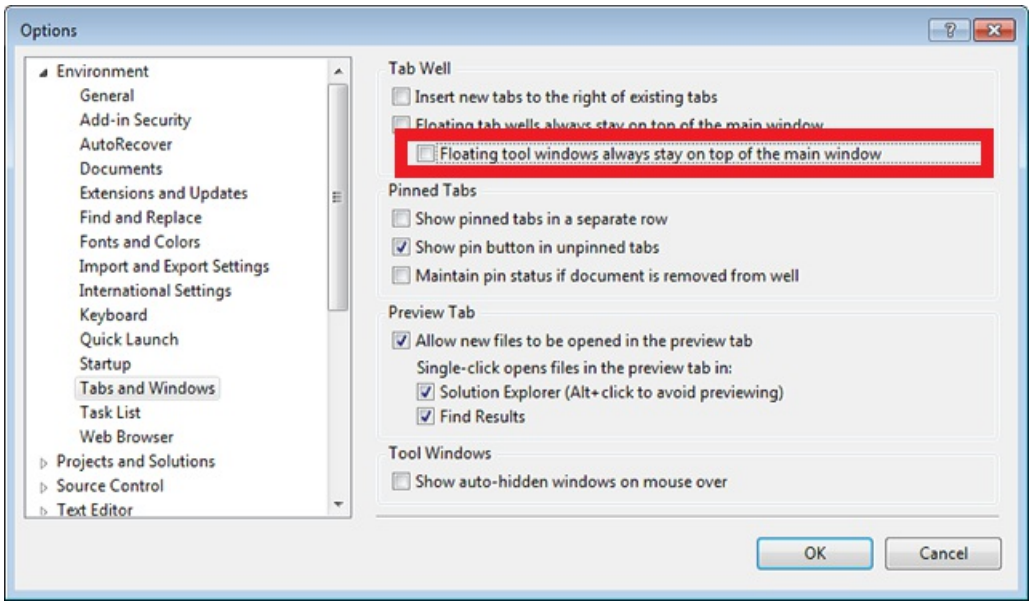
Use Page Inspector to View the Application

Next, you will view the application with Page Inspector. In **Solution Explorer**, right click the project, and then choose **View in Page Inspector**.

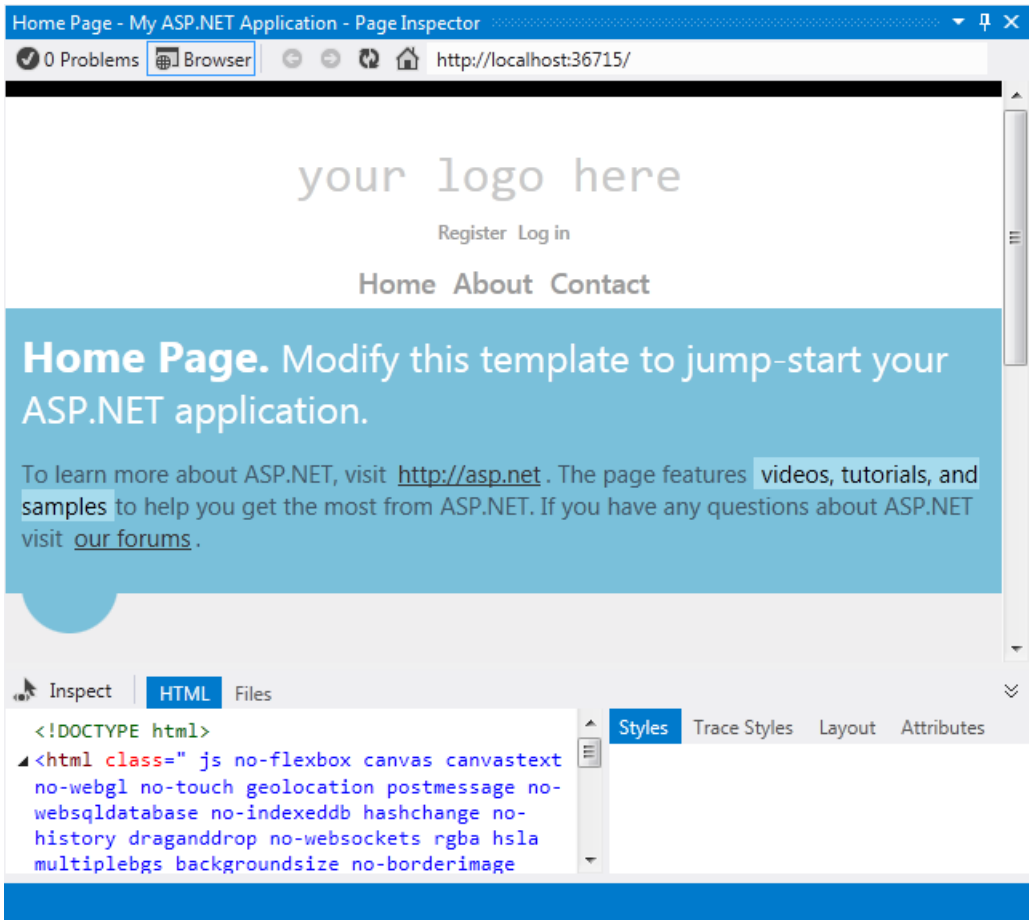


By default, when Page Inspector launches for the first time, it is docked as a narrow window on the left side of the Visual Studio environment. Leave it docked on the left side and set it to a width that is comfortable for you, or dock it in one of the tool areas on the top, bottom, or right:

If you undock the Page Inspector window, you can place it outside Visual Studio, or even on a second monitor if you have one. However, in order to ALT+TAB between Page Inspector and Visual Studio when the Page Inspector window is undocked, go to **Tools** > **Options** > **Environment** > **Tabs and Windows**, and under **Tab Well**, clear the check box called **Floating tool windows always stay on top of the main window**:



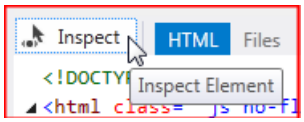
The top pane of the Page Inspector window shows the current page in a browser window. The bottom pane shows the page in HTML markup on the left, and some tabs on the right that let you inspect different aspects of the page. The bottom pane is similar to the F12 Developer Tools in Internet Explorer. (However, unlike the developer tools, you can use Page Inspector right within Visual Studio.)



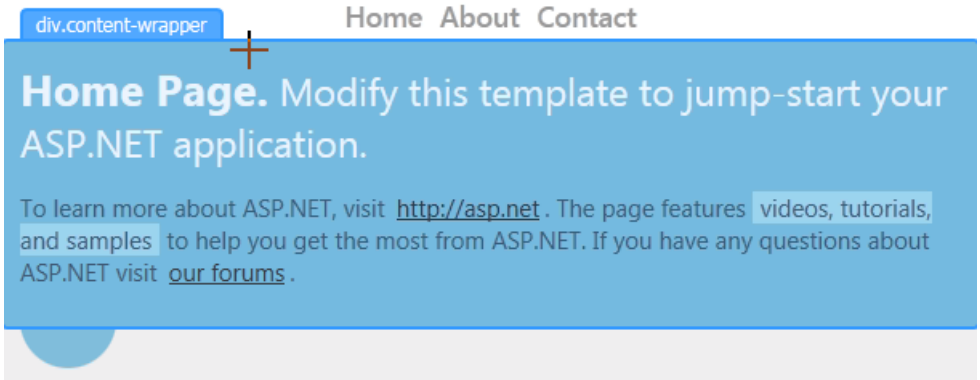
In this tutorial, you will use the Page Inspector browser pane, and the **HTML** and **Styles** tabs to help you rapidly navigate and make changes to the application.

Enable Inspection Mode

Next, you will see how Page Inspector's Inspection Mode works. In the Page Inspector window, click the **Inspect** button.



To see inspection mode in action, move your mouse over different parts of the page within the Page Inspector browser window. As you do, the mouse pointer changes to a large plus sign, and the element underneath is highlighted:



As you move the mouse pointer, note that

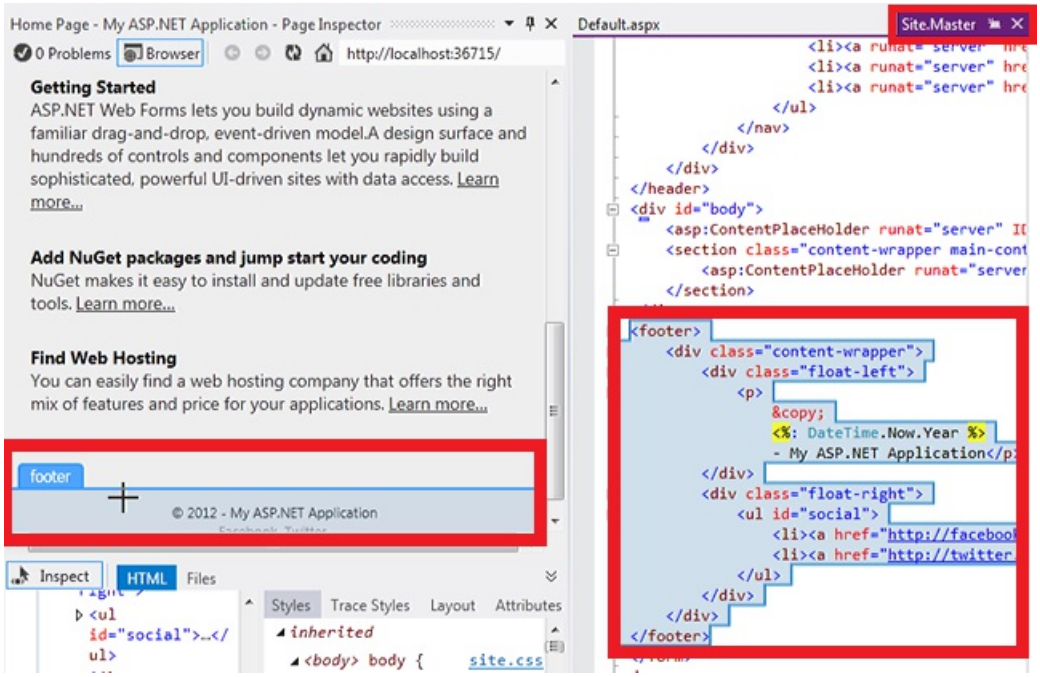
- The content in **Source** view changes to show the markup corresponding to the selected element on the page. The relevant markup is highlighted. If the source is in another file, that file is opened in Source view with the relevant markup highlighted.
- The markup displayed in the **HTML** tab in Page Inspector also changes to correspond to the selected element on the page. In the **HTML** tab, the relevant markup is outlined.
- The **Styles** tab shows the CSS rules relevant to the current selection.

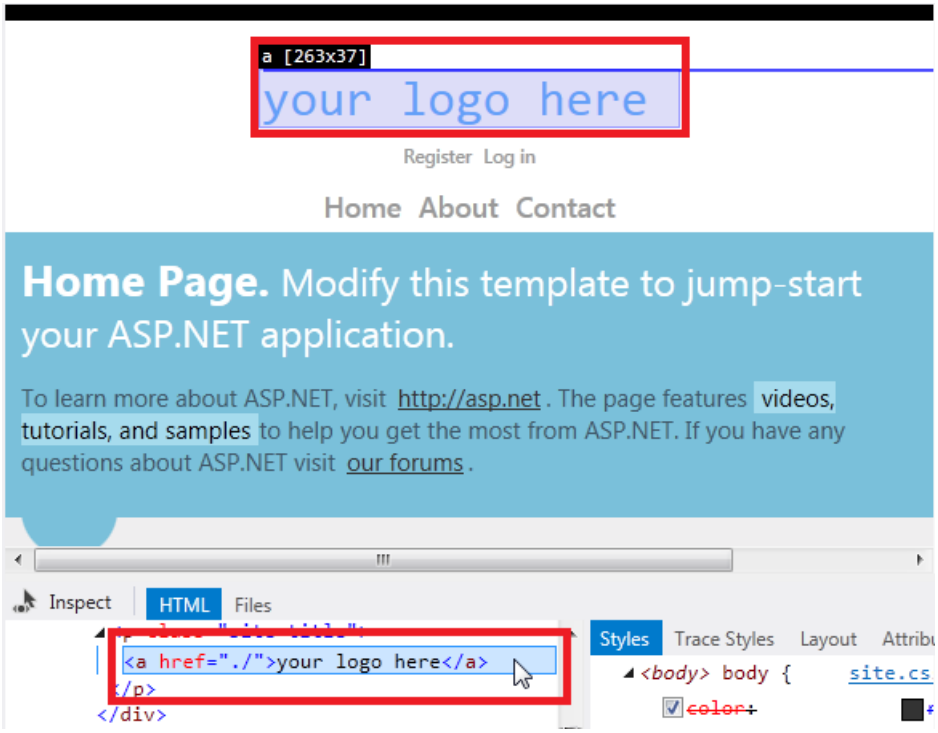
Use Page Inspector to Make Changes to Markup

Now you will see how you can use Page Inspector to find and make changes to markup or text whose location might not be immediately obvious.

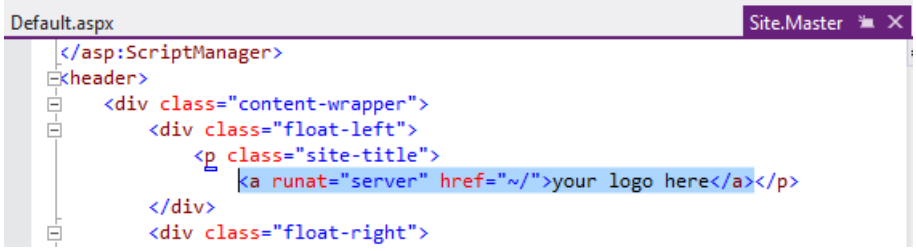
Put Page Inspector in Inspection Mode and then scroll to the bottom of the home page.

As soon as you enter the footer area, Page Inspector opens the *Site.Master* layout file in **Source** view in a temporary tab to the right of the other tabs and highlights the section of the master page that you have selected. This shows you how Page Inspector can find and display content on a page that might actually come from a different file than the one you originally opened.


Home Page - My ASP.NET Application - Page Inspector
0 Problems
Browser
http://localhost:36715/
Getting Started
ASP.NET Web Forms lets you build dynamic websites using a familiar drag-and-drop, event-driven model.A design surface and hundreds of controls and components let you rapidly build sophisticated, powerful UI-driven sites with data access. Learn more...
Add NuGet packages and jump start your coding
NuGet makes it easy to install and update free libraries and tools. Learn more...
Find Web Hosting
You can easily find a web hosting company that offers the right mix of features and price for your applications. Learn more...
footer
© 2012 - My ASP.NET Application
Inspect
HTML
Files
Styles
Trace Styles
Layout
Attributes
inherited
Default.aspx
Site.Master
©
<%: DateTime.Now.Year %>
- My ASP.NET Application</p>
</footer> </footer>



As before, Page Inspector opens the *Site.Master* file for you in a temporary tab. Click the Site.Master tab, and the corresponding markup is highlighted in the <header> section:

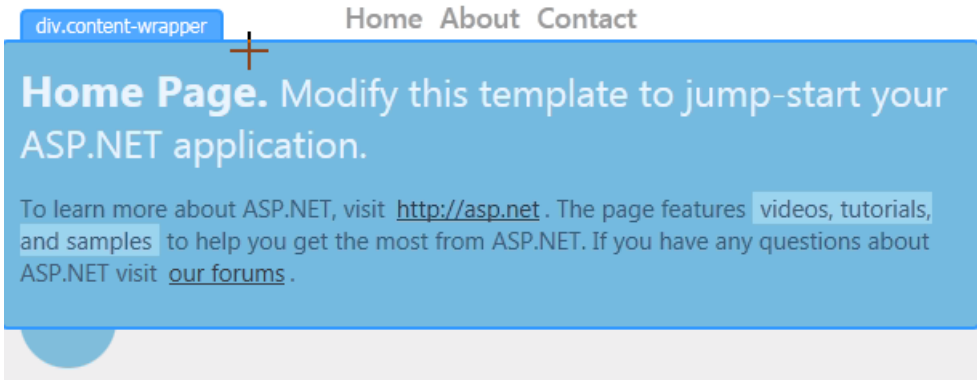


Preview CSS Changes in the Styles Window

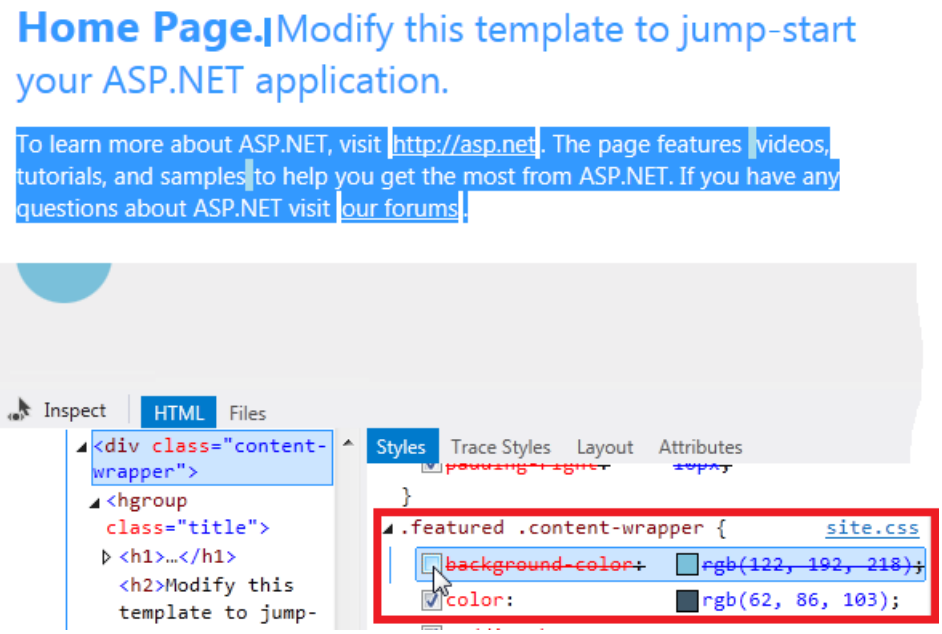
Next, you will see how you can use the Page Inspector **Styles** window to preview changes to CSS.

Click the **Inspect** button to put Page Inspector in Inspection Mode.

In the Page Inspector browser window, move the mouse pointer over the "Home Page" section until the **div.content-wrapper** label appears.

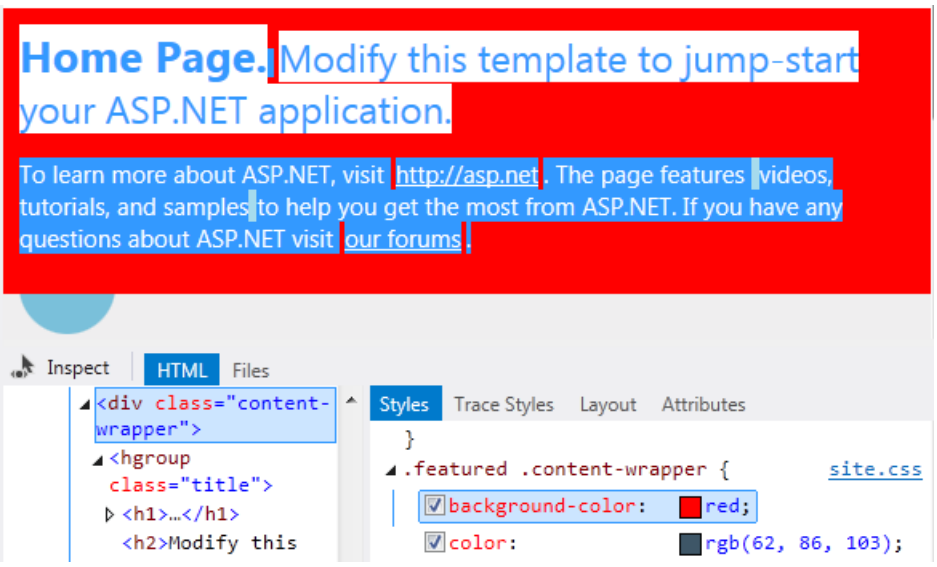


Click within the div.content-wrapper section once, and then move the mouse pointer to the **Styles** window. Under the .featured .content-wrapper class selector, clear and select the checkbox for the background-color property.



Notice how the change previews instantly in the Page Inspector browser window.

Select the checkbox again, then double-click the property value and change it to `red`. The change shows immediately:



The **Styles** window makes it easy to test and preview CSS changes before you commit the changes to the style sheet itself.

CSS Auto Sync

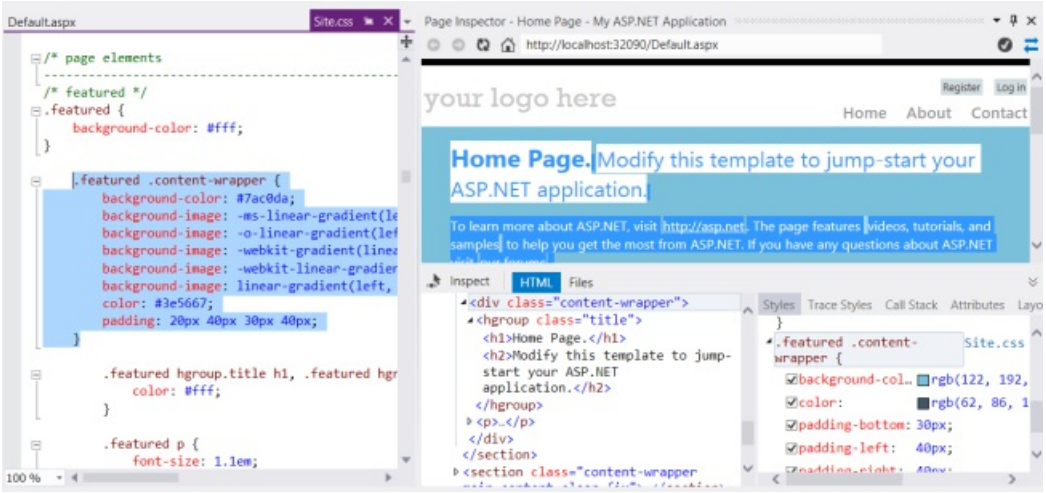
> **NOTE**
>
> This feature requires version 1.3 of Page Inspector.

The CSS Auto-Sync feature allows you to edit a CSS file directly, and see the changes immediately in the Page Inspector browser.

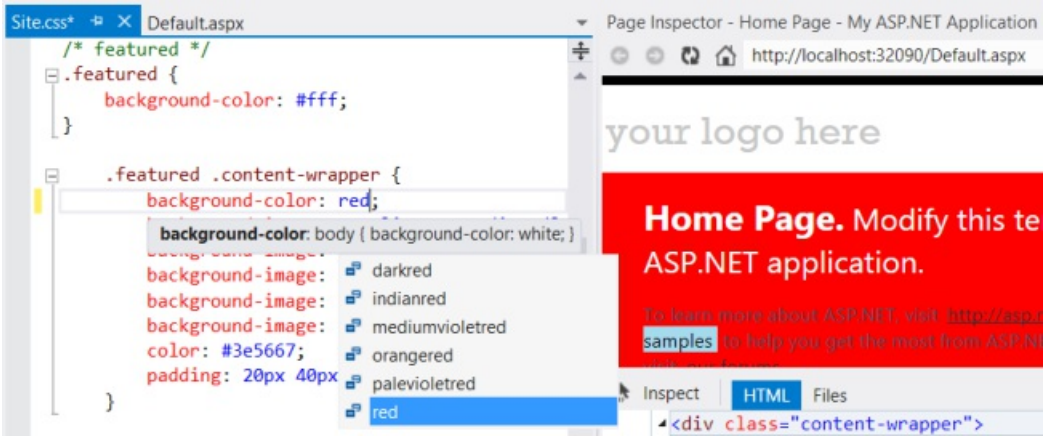
Click **Inspect** to put Page Inspector in Inspection Mode.

In the Page Inspector browser, move the mouse pointer over the "Home Page" section until the **div.content-wrapper** label appears. Click once to select this element.

The **Styles** window shows all of the CSS rules for this element. Scroll down to find the .featured .content-wrapper class selector. Click on ".featured .content-wrapper". Page Inspector opens the CSS file that defines this style (Site.css) and highlights the corresponding CSS style.



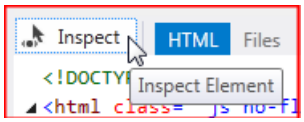
Now change the value for `background-color` to "red". The change appears immediately in the Page Inspector browser.



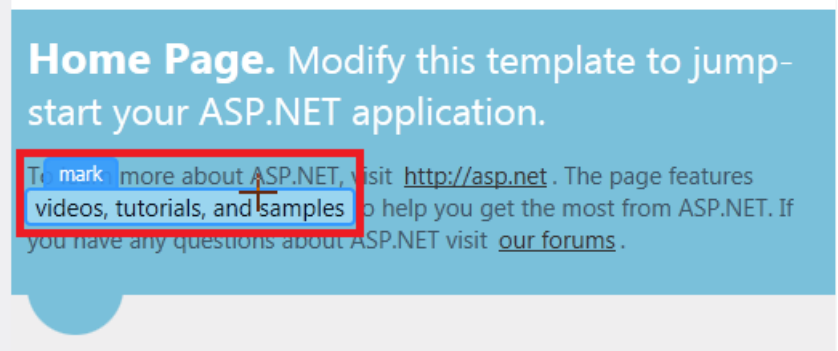
Using the CSS Color Picker

Next, you'll learn how to use Page Inspector to quickly find and change the CSS for highlighted text in the default application. In this example, you've decided that you don't like the blue highlighting and want to change it to another color.

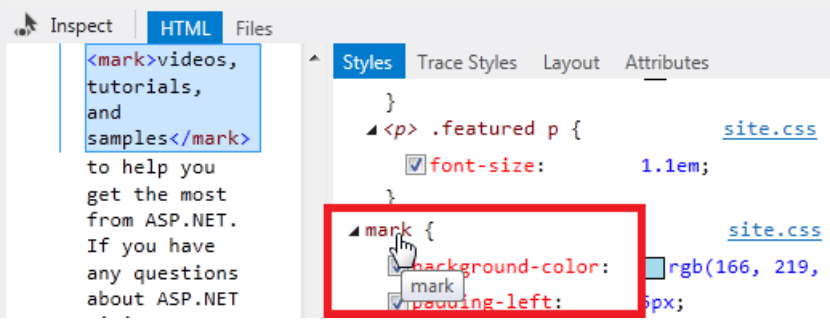
Click the **Inspect** button.



In the Page Inspector browser window, move the mouse pointer over the highlighted "videos, tutorials, and samples" text so that the CSS "mark" label appears.



Click the text to select it. The corresponding CSS mark selector appears at the bottom of the **Styles** window.



Click the mark selector. This opens the *Site.css* file for the web application. Click the Site.css tab, and the corresponding CSS for the selector is highlighted:

```
    display: block;
}

mark {
    background-color: #a6dbed;
    padding-left: 5px;
    padding-right: 5px;
}

.float-left {
    float: left;
}
```

Select and remove the line with the background-color property.

You will now use the new Visual Studio 2012 CSS color picker to choose a new color for the **mark** background-color property.

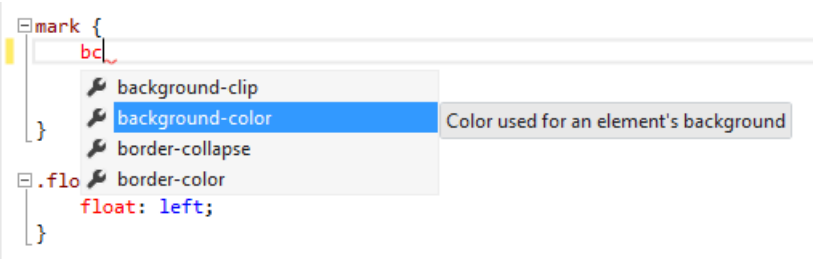
Using the Visual Studio 2012 CSS Color Picker

The CSS editor in Visual Studio 2012 has a color picker that makes it easy to choose and insert colors. It has a simple color bar and a "pop-down" picker that offers finer control.

The color picker includes a standard palette of colors, supports standard color names, hash codes, RGB, RGBA, HSL, and HSLA colors, and maintains a list of the colors you've used most recently in the document.

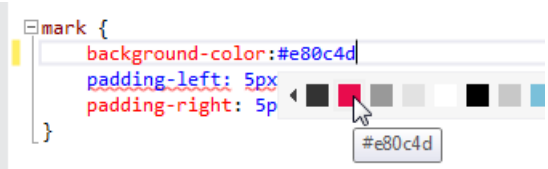
On the line where the background-color property was, type "bc" and press the down arrow once.

When you type the first character of each word in a hyphen-separated property like "background-color", IntelliSense filters the list for you to show only the properties that match:



Now type a colon. When you do, the full background-color property name is inserted. Type **#** or **rgb(**, and the color picker bar appears:

To see how the color picker bar works, click its colors with the mouse pointer, or press the down arrow key and then use the left and right arrow keys to traverse the colors. When you visit a color, the corresponding value for the background-color property is previewed:

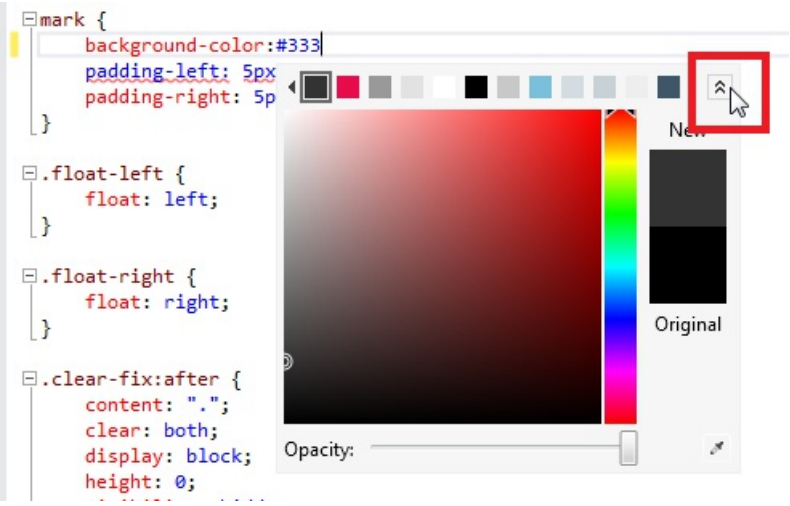


At this point, you could press Enter to select the value and then a semicolon (;) to complete the CSS entry. For now, go on to the next section so that you can see how the color picker pop-down works.

Using the Color Picker Pop-Down

When the color bar doesn't have the exact color that you're looking for, you can use the color picker pop-down.

To open it, click the double chevron at the right end of the color bar, or press the Down Arrow once or twice on the keyboard.



Click a color from the vertical bar on the right. This shows a gradient for that color in the main window. Choose a color directly from the vertical bar by pressing Enter, or click any point in the main window to choose with greater precision.

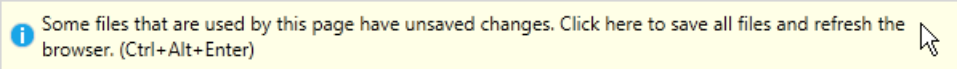
If there is a color on your computer screen that you want to use (it doesn't have to be inside the Visual Studio user interface), you can capture its value by using the eyedropper tool on the lower right.

You also can change the opacity of a color by moving the slider at the bottom of the color picker. Doing so changes color values to RGBA values because the RGBA format can represent opacity.

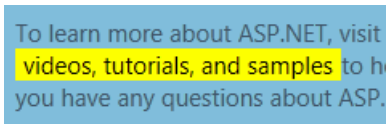
After you have chosen a color, press Enter, and then type a semicolon to complete the background-color entry in the *Site.css* file.

The Page Inspector Update Bar

Page Inspector immediately detects the change to the *Site.css* file (or to any file in the application) and displays an alert in an update bar.



To save all your files and refresh the Page Inspector browser, press Ctrl+Alt+Enter or click the update bar. The change in the highlight color appears in the browser:



Notice that you conveniently refreshed the Page Inspector browser right from within the Visual Studio environment. Using Page Inspector instead of an external browser lets you stay in the editor when you develop your web applications.

See Also

Introducing Page Inspector (Channel 9 video)

Page Inspector Error Messages (MSDN)

Visual Studio 2012 Hands On Labs

2/20/2020 • 2 minutes to read • Edit Online

Learn how to develop with ASP.NET Web Forms with these Hands On Labs

- What's New in ASP.NET and Web Development in Visual Studio 2012
- What's New in Web Forms in ASP.NET 4.5
- Using Page Inspector in Visual Studio 2012

What's New in ASP.NET and Web Development in Visual Studio 2012

2/20/2020 • 33 minutes to read • Edit Online

by Web Camps Team

The new version of Visual Studio introduces a number of enhancements focused on improving the experience and performance when working with Web technologies. Visual Studio Editors for CSS, JavaScript and HTML have been completely revamped to include many of the most in-demand code aids, such as IntelliSense and automatic indentation. Regarding performance, bundling and minification are now integrated as built-in features to easily reduce page load time.

Visual Studio enables you to work with the latest website technologies. You can use cross-browser CSS3 Snippets to make sure your site works regardless of the client platform while also taking advantage of the new HTML5 elements and features.

Writing and profiling JavaScript code should be easier with this Visual Studio version. IntelliSense lists, integrated XML documentation and navigation features are now available for JavaScript code. You now have the JavaScript catalog at your fingertips. Additionally, you can check ECMAScript5 compliance with your scripts and detect syntax errors at an early stage.

Last but not least, this Visual Studio version implements built-in bundling and minification. Your script files and style sheets will be packed and compressed so that the site performs faster.

This lab walks you through the enhancements and new features previously described by applying minor changes to a sample Web application provided in the Source folder.

All sample code and snippets are included in the Web Camps Training Kit, available at https://go.microsoft.com/fwlink/?LinkID=248297&clcid=0x409.

Objectives

In this hands on lab, you will learn how to:

- Use the new features and improvements in the CSS editor
- Use the new features and improvements in the HTML editor
- Use the new features and improvements in the JavaScript editor
- Configure and use bundling and minification

Prerequisites

- Microsoft Visual Studio Express 2012 for Web or superior (read Appendix A for instructions on how to install it).
- Windows PowerShell (for setup scripts - already installed on Windows 8 and Windows Server 2008 R2)
- Internet Explorer 10 - or an HTML5 compliant browser

Exercises

This hands on lab includes the following exercises:

1. Exercise 1: What's New in the CSS Editor
2. Exercise 2: What's New in the HTML Editor

3. Exercise 3: What's New in the JavaScript Editor
4. Exercise 4: Bundling and Minification

Estimated time to complete this lab: **60 minutes**.

Exercise 1: What's New in the CSS Editor

Web developers should be familiar with many of the difficulties related to CSS editing. One of the biggest issues of CSS styling is cross-browser compatibility. It often happens that, after applying styles to your site, you notice that it looks different if you open it in another browser or device. Therefore, you may spend a considerable time on fixing those visual issues to realize that, when you finally make it work in one browser, it is broken in the others.

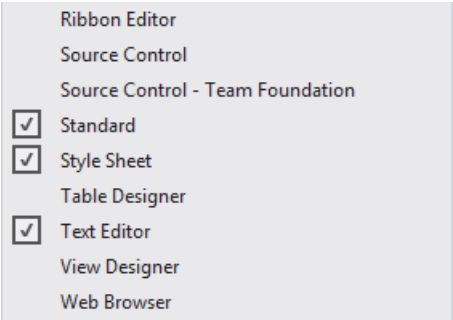
Visual Studio now includes features that help developers access, work and organize CSS style sheets effectively. Throughout this exercise, you will meet the new features for an effective organization and edition, as well as the CSS3 Code Snippets for cross-browser compatibility.

Task 1 - New Editor Features

In this task, you will discover the new features of the CSS Editor. This new editor will help you increase your productivity by taking advantage of the new smart indentation, the improved code comments and the enhanced IntelliSense list.

1. Start **Visual Studio** and open the **WhatsNewASPNET.sln** solution located in the **Source\WhatsNewASPNET** folder of this lab.

2. In Solution Explorer, open the **Site.css** file located under the **Styles** folder. Make sure the **Text Editor** tools are visible on the toolbar. To do that, select the **View | Toolbars** menu option, and check the **Text Editor** options. You will notice that, since this new version, the **Comment** button ( ) and the **Uncomment** button ( ) are also enabled for the CSS editor.



Enabling Editor and CSS Tools

3. Scroll the code and select any CSS class definition. Click the **Comment** ( ) button to comment the selected lines. Then, click the **Uncomment** ( ) button to undo the changes.

4. Click the **Collapse** ( ) and **Expand** ( ) buttons located on the left margin of the text. Notice that you can now hide the styles you don't use to have a cleaner view.

```
/* PRIMARY LAYOUT ELEMENTS
-------------------------------------------------------------*/

.page...

.header...

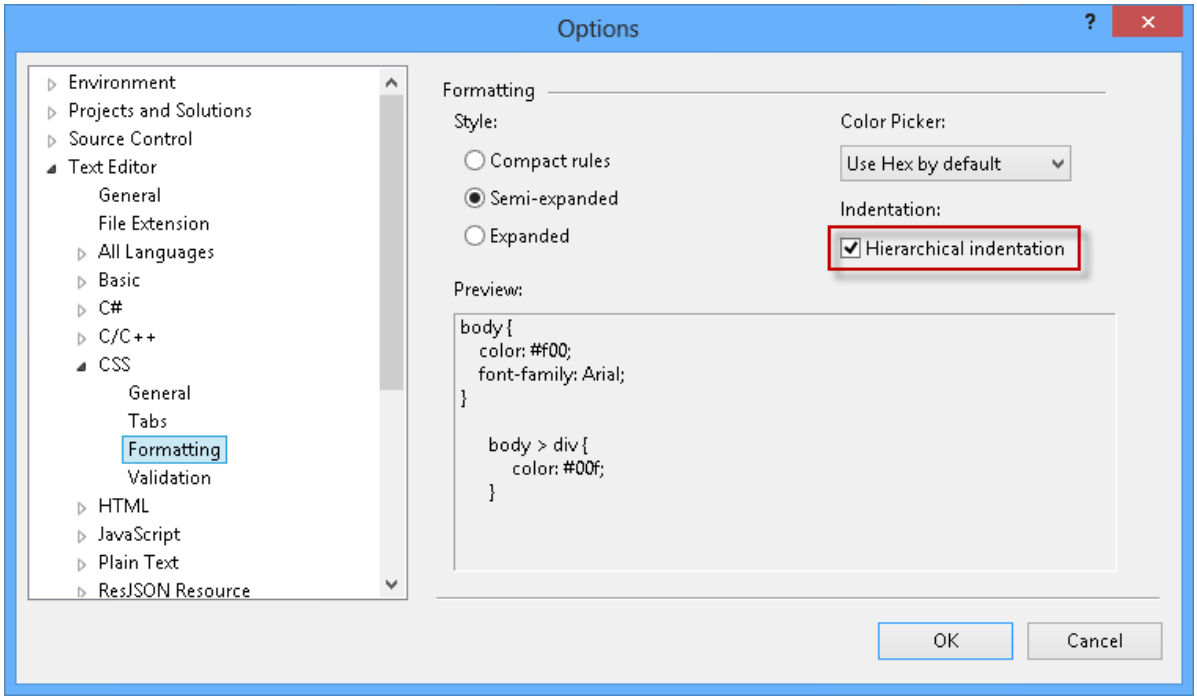
.header h1
{
    font-weight: 700;
    margin: 0px;
    padding: 0px 0px 0px 20px;
    color: #f9f9f9;
    border: none;
    line-height: 2em;
    font-size: 2em;
}

.main
{
    padding: 0px 12px;
    margin: 12px 8px 8px 8px;
    min-height: 420px;

}
```

Collapsing CSS classes

5. Make sure that the smart indentation feature is enabled. Select the **Tools | Options** menu option, and then select the **Text Editor | CSS | Formatting** page in the left pane of the screen. Check the **Hierarchical indentation** option.



Enabling hierarchical indentation

6. Locate the main class definition (.main) and append a style to the div elements. You will notice that the code aligns automatically, helping users to find the parent classes at a glance.

   CSS

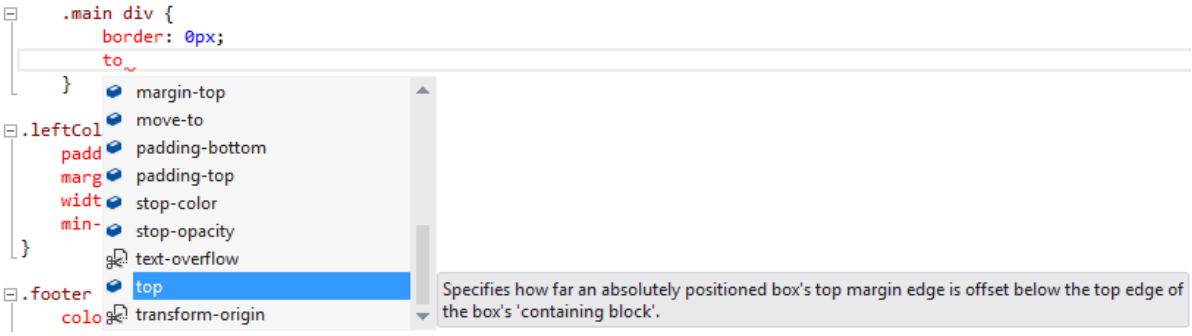
```
.main
{
  padding: 0px 12px;
  margin: 12px 8px 8px 8px;
  min-height: 420px;
}
  .main div
  {
    border: 0px;
  }
```

```
.main {
    padding: 0px 12px;
    margin: 12px 8px 8px 8px;
    min-height: 420px;
}

    .main div {
        border: 0px;
    }
```

Hierarchical alignment in CSS

7. Inside **.main div** class, locate the cursor at the end of **border: 0px;** and press **Enter** to display the IntelliSense list. Start typing **top** and notice how the list is filtered as you type. The list will display the elements that contain **top** at any part of the word (In prior versions of Visual Studio, the list is filtered by the items that *begin* with the term).



IntelliSense enhancements in CSS

Task 2 - The Color Picker

In this task, you will discover the new CSS Color Picker integrated into Visual Studio IntelliSense.

1. In **Site.css,** locate the header class definition (.header) and place the cursor next to **background-color** attribute, between the ":" and "#" characters on that line of code .

```
.header
{
    background-color:#2c5497/*#ef622a*/;
    position: relative;
    margin: 0px;
    padding: 0px;
    width: 100%;
    /*border radius*/

}
```

Locating the cursor

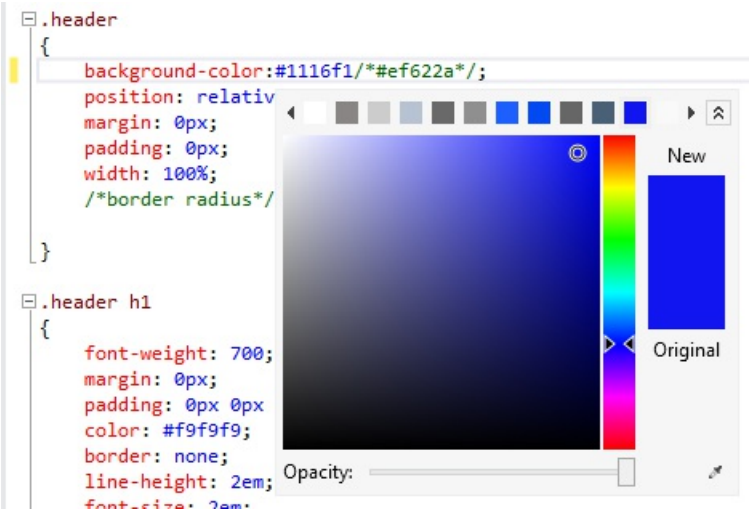
2. Delete the **colon** (:) and write it again to display the color picker. Notice that the first colors you will see are the most frequent colors of your site. If you click the white color, its HTML color code (#fff) will replace the current color code in the stylesheet.

```
.header
{
    background-color:#2c5497/*#ef622a*/;
    position: relativ
    margin: 0px;
    padding: 0px;
    width: 100%;
    /*border radius*/

}
```

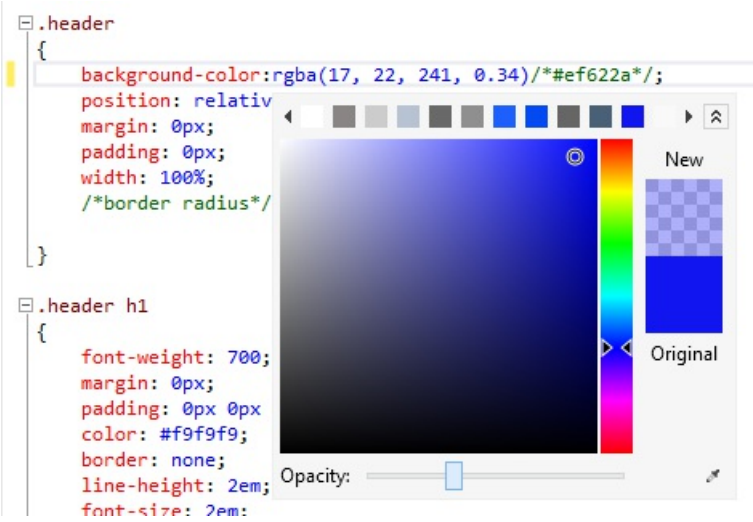
Color picker

3. Press the **Expand** ( ) button on the color picker to display the color gradient, and then drag the gradient cursor to select a different color. After that, click the **Eyedropper** button and select any color from the screen. Notice that background color value changes dynamically while you move the cursor.



Color picker gradient

4. In the **Opacity** slider, move the selector to the center of the bar to reduce the opacity. Notice that background-color value now changes its scale to RGBA.



Color picker Opacity

> **NOTE**
>
> The RGBA (Red, Green, Blue, Alpha) color definition in CSS3 enables you to define the color opacity value for a single item. Unlike **opacity** - a similar CSS attribute - RGBA colors are also compatible with the latest browsers.

Task 3 - CSS Compatible Code Snippets

In this task, you will learn how to use cross-browser compatible CSS3 snippets in order to implement some features in your website.

1. In the **Site.css** file, locate the **header** CSS class definition (.header) and place the cursor below the **/*border radius*/** placeholder to add a new snippet. Press **Enter** to display the IntelliSense list and type **radius** to filter the list. Select the **border-radius** option from the list with a double click, and then press the **TAB** key to insert the snippet. Then, type a radius size in pixels and press **Enter**. For instance, type **15px**.

   The CSS3 attributes added by the snippet will render rounded borders in most HTML5 compliance browsers, including Mozilla and WebKit-based browsers.

   Using a border-radius snippet

2. Apply the same **border** snippets in the page style (.page).

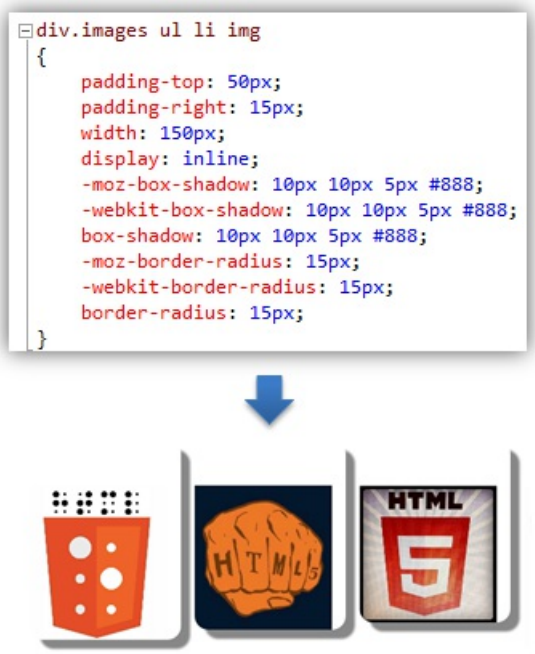
   CSS

```
.page
{
    width: 960px;
    background-color: #fff;
    margin: 20px auto 0px auto;
    border: 1px solid #496077;
    /*border radius*/
    -moz-border-radius: 15px;
    -webkit-border-radius: 15px;
    border-radius: 15px;
}
```

3. Press **F5** to run the solution. Notice that each page now has rounded borders.

Rounded corners

4. Close the browser and return to Visual Studio.
5. Open the **Custom.css** file located under the **Styles** folder and place the cursor inside **div.images ul li img** class definition.
6. Press enter to display the IntelliSense list, type **box-shadow** and press the **TAB** key twice to insert the default shadow code snippet inside the class definition. Set the shadow values to **10px 10px 5px #888**. Then, type **border-radius** and insert the code snippet. Type **15px** to set radius size and press **ENTER**.



```
div.images ul li img
{
    padding-top: 50px;
    padding-right: 15px;
    width: 150px;
    display: inline;
    -moz-box-shadow: 10px 10px 5px #888;
    -webkit-box-shadow: 10px 10px 5px #888;
    box-shadow: 10px 10px 5px #888;
    -moz-border-radius: 15px;
    -webkit-border-radius: 15px;
    border-radius: 15px;
}
```

Rounded corners with shadow

> **NOTE**
>
> At this moment, the shadow attribute is inserted with the corresponding prefix (moz, webkit, o) to support Mozilla and Webkit (Chrome, Safari, Konkeror) browsers.

7. Create a new class **div.images ul li img:hover** below the **div.images ul li img** class definition and place the cursor inside the brackets .

   CSS

```
div.images ul li img
{
  ...
}

div.images ul li img:hover
{

}
```

8. Type **transform** and press the **TAB** key twice in order to insert the transform snippet. Then, enter **rotate(-15deg)** to change the rotation angle value when images are hovered.

   CSS

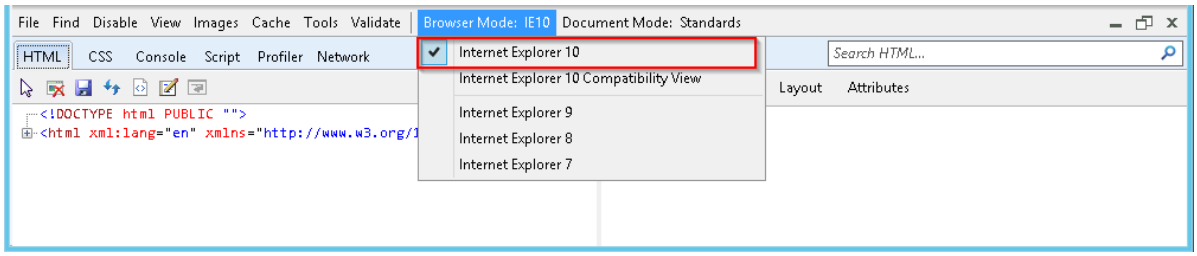
```
div.images ul li img
{
  padding-top: 50px;
  padding-right: 15px;
  width: 150px;
  ...
}

div.images ul li img:hover
{
  -moz-transform: rotate(-15deg);
  -ms-transform: rotate(-15deg);
  -o-transform: rotate(-15deg);
  -webkit-transform: rotate(-15deg);
  transform: rotate(-15deg);
}
```

9. Press **F5** to run the solution and browse to the CSS3 page. Notice that the images have rounded corners and box shadows. Hover the mouse over the images and watch them rotate.

Transform snippet rotating an image

> **NOTE**
>
> If you are using Internet Explorer 10 and cannot see the shadows, make sure the document mode is set to IE10 standards. Press **F12** to open Internet Explorer developer tools and click **Document Mode** to change to IE10 standards.



Exercise 2: What's New in the HTML Editor

Visual Studio has an improved HTML editor. Some of the enhancements included in this version are smart indentation in HTML documents, HTML5 snippets, HTML start and end tag matching, and HTML validation. Throughout this exercise, you will see how these changes improve your fluency when working in the website markup.

Like the CSS editor, the HTML editor was also improved. Most of these improvements are small ones that make the Web developer's life easier. Things like more code snippets for HTML5, smart indentation, matching start and end tags when editing and validation targeting the HTML document DOCTYPE are some of these improvements.

Task 1 - Improved DOCTYPE Validation

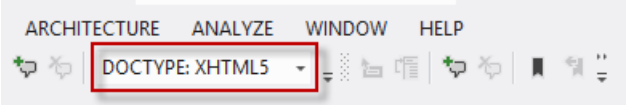
The HTML editor now has the ability to check the DOCTYPE of your page, even though the definition might be in the master page. Depending on the DOCTYPE of your page, the HTML editor will validate with the correct set of rules and will filter the IntelliSense list considering the DOCTYPE elements.

In this task, you will change the DOCTYPE of a page to see how the HTML editor behavior changes accordingly.

1. If not already opened, start **Visual Studio** and open the **WhatsNewASPNET.sln** solution located in the **Source\WhatsNewASPNET** folder of this lab.

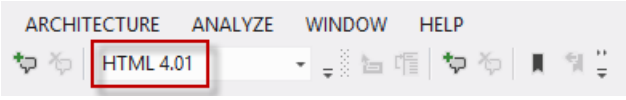
2. Open the **Site.Master** page.

3. Notice the Target Schema for Validation Toolbar. The way the HTML editor behaves (Validation, IntelliSense, etc.) will properly change to fit the Doctype selected.



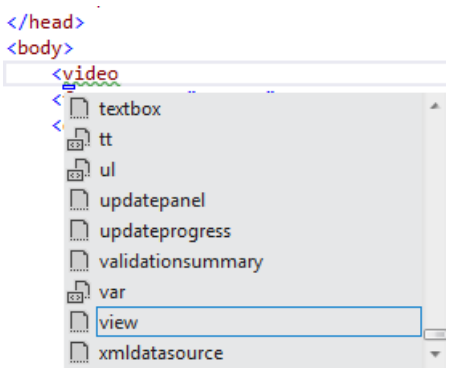
Use Doctype in HTML Source Editing toolbar

4. Change the Target Schema to HTML 4.01.



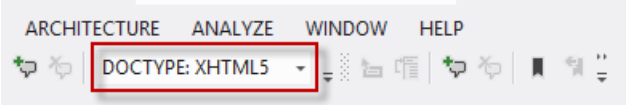
Changing Doctype in HTML Source Editing toolbar

5. Place the cursor under the **body** element, and start typing the name of an HTML5 element (for example, **video**). Notice that the element is not available in the IntelliSense list.



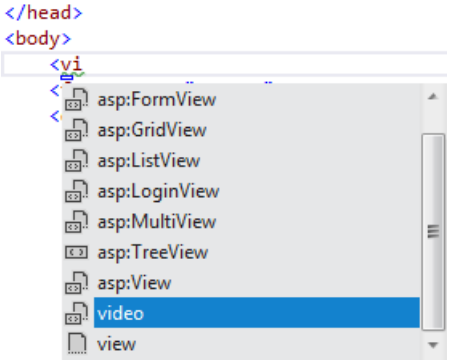
HTML5 elements not listed

6. Undo the changes to the Target Schema for Validation Toolbar, picking DOCTYPE: XHTML5 from the dropdown list.



Reset Doctype in HTML Source Editing toolbar

7. Place the cursor under the **body** element and start typing an HTML5 element again (For example, like **video**). Notice that the HTML5 elements are now available in the IntelliSense list.



HTML5 elements being listed

Task 2 - Start/End Tags Automatic Update

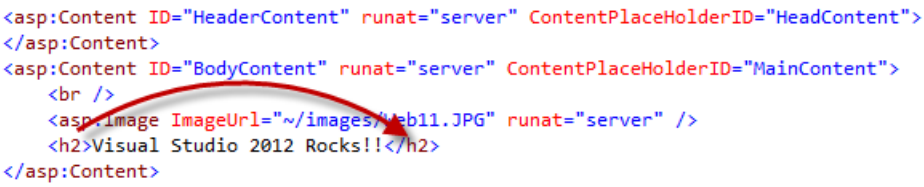
Visual Studio now updates the HTML opening or closing tags of the element that you are editing to match each other. This new feature will improve your productivity when editing HTML tags.

1. On the **Default.aspx** page, add an **H3** element with a title (for example, Visual Studio 2012 Rocks!).

```
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
  <br />
  <asp:Image ImageUrl="~/images/Web11.JPG" runat="server" />
  <h3>Visual Studio 2012 Rocks!!</h3>
</asp:Content>
```

2. Change the **H3** tag and type **H2** or **H1**.

   Notice that the end tag automatically updates. You can also modify the end tag to see that the start tag updates accordingly too.



Automatic update of the end tag

Task 3 - New HTML5 Code Snippets

Visual Studio now includes several HTML5 code snippets. In this task, you will use some of these snippets.

1. Add a new folder named **audio** to the root of the web site folder. Open Windows Explorer and copy any audio file into the **audio** folder of the **WhatsNewASPNET.sln** solution.

2. In the **Default.aspx** page, locate the cursor under the Web11 Rocks!! Header. Type **audio** and press the TAB key.

The new HTML editor includes code snippets for HTML5 content. Remember to use the proper DOCTYPE definition to enable the HTML5 snippets.

Inserting HTML5 Code Snippets

3. Update the audio source to point to an existing audio file.

```
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
  <br />
  <asp:Image ImageUrl="~/images/Web11.JPG" runat="server" />
  <h2>Visual Studio 2012 Rocks!!</h2>
  <audio controls="controls">
    <source src="audio/Kalimba.mp3" />
  </audio>
</asp:Content>
```

> **NOTE**
>
> You will need to add the audio file to the solution.

4. Press **F5** to run the site and play the audio.

Running the audio control

> **NOTE**
>
> You can also try more snippets included in Visual Studio, such as video, figure, etc.

5. Now, try to insert a control in some part of the page. For example, try to insert a **GridView** control, but instead of typing **<Gri,** start typing **<GV**. Notice that the IntelliSense list shows the **asp:GridView** control.

   IntelliSense in the HTML Editor now provides title-casing search, as well as partial matching (retrieving all elements that contains the term).

```
<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
</asp:Content>
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <br />
    <asp:Image ImageUrl="~/images/Web11.JPG" runat="server" />
    <h3>Visual Studio ROCKS!!!!</h3>
    <audio controls="controls">
        <source src="audio/Kalimba.mp3" type="audio/mp3" />
    </audio>
    <GV
</asp
    asp:GridView        asp:gridview
                        Markup snippet for a gridview control
```

Inserting a GridView with IntelliSense lists

If you type **<grid** you will get all the items that match the term, but Visual Studio will suggest the **gridview** control:

```
<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
</asp:Content>
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <br />
    <asp:Image ImageUrl="~/images/Web11.JPG" runat="server" />
    <h3>Visual Studio ROCKS!!!!</h3>
    <audio controls="controls">
        <source src="audio/Kalimba.mp3" type="audio/mp3" />
    </audio>
    <grid
</asp
    asp:DataGrid
    asp:GridView
    asp:PropertyGridEditorPart
    gridview            Markup snippet for a gridview control
```

Inserting a GridView with IntelliSense lists and partial matching

Task 4 - HTML Editor Smart Tags

Another improvement in the HTML Editor is the Smart Tags feature. Smart tags make it easy to perform common or repetitive development tasks on a per-control basis. This feature was already available in the HTML Designer, but not in the HTML Editor.

1. Open **Site.Master** and locate the **asp:Menu** element. Place the cursor on the start tag and notice that the small glyph displayed at the bottom of the element - click it to open the smart tasks menu. Notice that you have quick access to some tasks related to the Menu control.

```
<div class="clear hideSkiplink">
    <asp:Menu ID="NavigationMenu" runat="server" CssClass="menu" EnableViewState="false"
        Menu Tasks
        Format Element                          t.aspx" Text="HTML5"/>
        Auto Format...                          spx" Text="CSS3"/>
        Choose Data Source: (None)              ript.aspx" Text="JavaScript"/>
                                                zation.aspx" Text="Optimization"/>
        Edit Menu Items...
        Convert to DynamicItemTemplate
        Convert to StaticItemTemplate
    </a
</div>
</div>
<div class=
    <asp:ContentPlaceHolder ID="MainContent" runat="server"/>
</div>
```

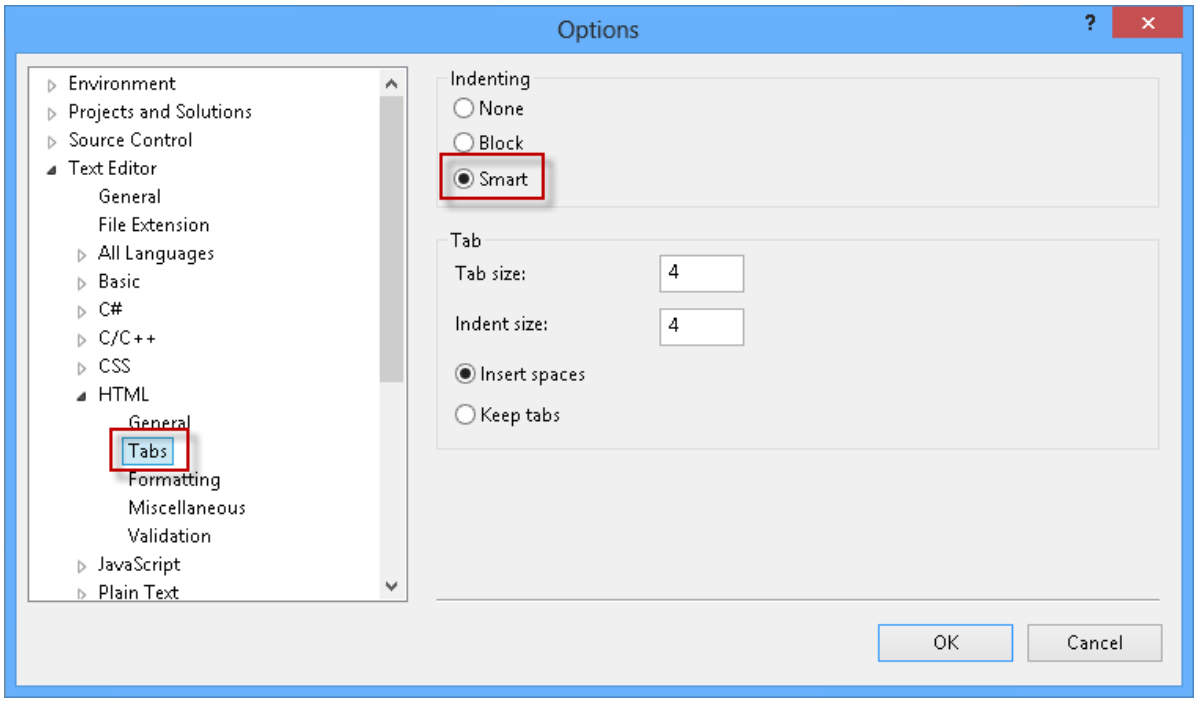
Smart tasks for the Menu control

Task 5 - Smart Indentation

One of the best practices in HTML is indenting the nested elements to keep the code readable. In Visual Studio 2012, you will notice that the editor automatically indents the elements while you are writing the code.

> **NOTE**
>
> In previous version of Visual Studio, smart indentation was available in the XML editor but not in the HTML editor.

1. Make sure that the Indenting configuration on the HTML Editor is set to Smart Indentation. To do that, select the **Tools | Options** menu option and then select the **Text Editor | HTML | Tabs** page in the left pane of the screen. Select the Smart indentation option.



HTML Editor settings

2. On the **Default.aspx** page, remove all the content under the audio element.

3. Place the cursor at the end of the opening **audio** element and hit **ENTER**.

   Notice that the new position of cursor has an additional indentation level.

```
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <br />
    <asp:Image ImageUrl="~/images/Web11.JPG" runat="server" />
    <h2>Visual Studio 2012 Rocks!!</h2>
    <audio controls="controls">
        |
    </audio>
</asp:Content>
```

Smart indentation in the HTML Editor

4. Restore the audio tag with the content you have removed, or close **Default.aspx** without saving the changes.

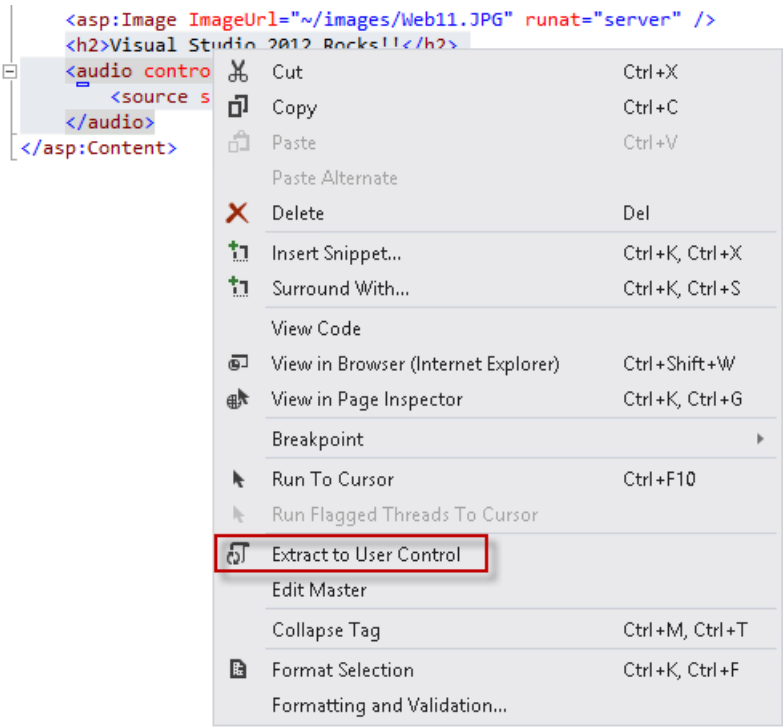
Task 6 - Extract to User Control

The Refactoring tools included in Visual Studio, such as extracting a portion of code to a function, are great features that facilitate the improvement and the refactoring the existing code. The counterpart for ASP.NET pages would be the extraction of HTML code to a User Control. Doing it manually would involve several steps, like creating a new User Control, moving the code section to the User Control, registering a tag prefix for the User Control, and, finally, instantiating the User Control on the pages. Now, the new *Extract to User Control* tool automatically performs all those steps for you.

In this task, you will use the new Extract to User Control contextual operation to generate a new user control from the selected code.

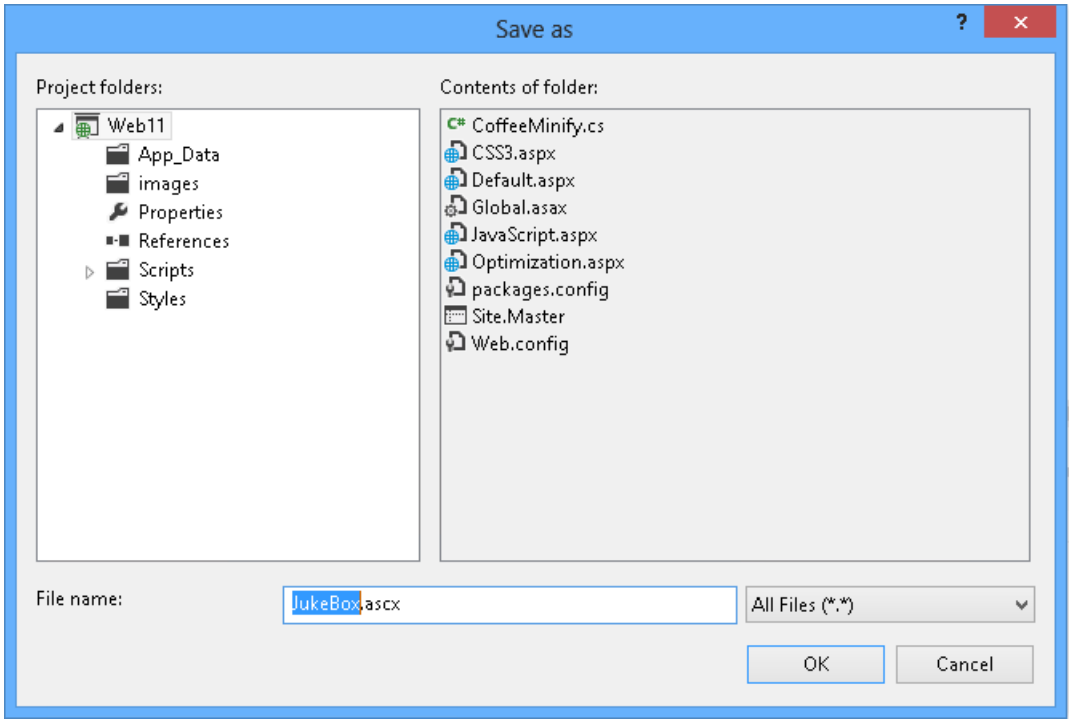
1. On the **Default.aspx** page, select the **H2** and **audio** elements.

2. Right click and select **Extract to User Control**.



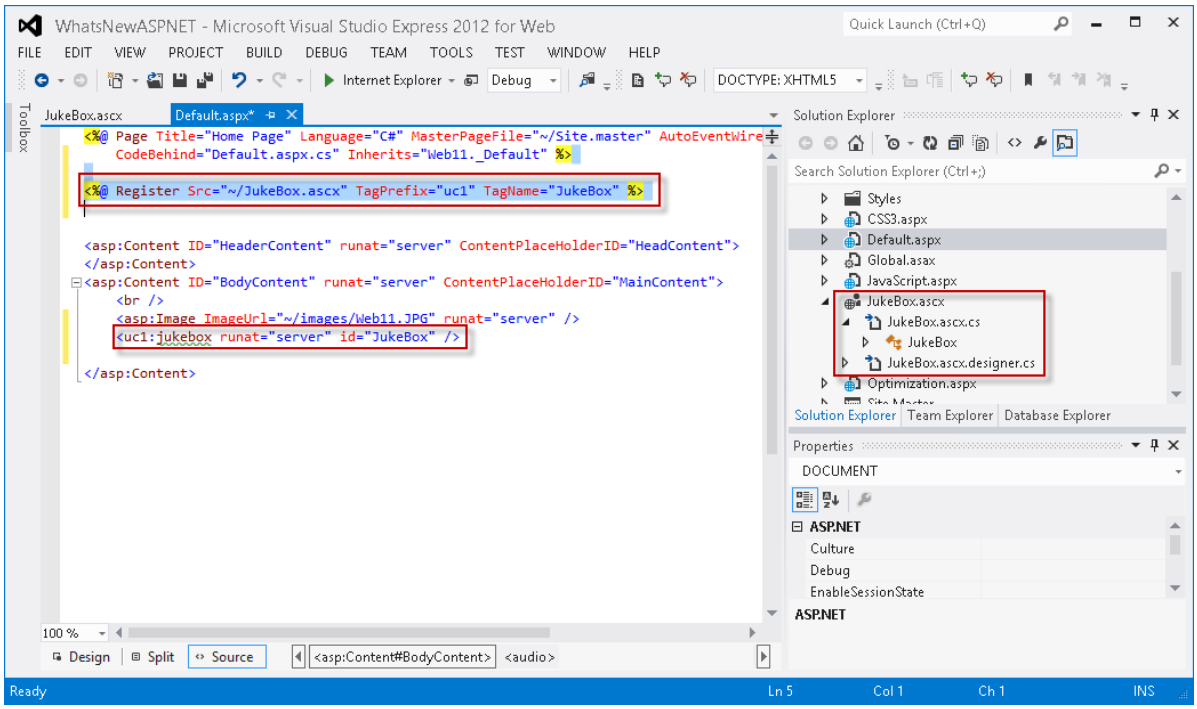
Extract to User Control menu option

3. Type a name for the new user control. For instance, **Jukebox.ascx**, and then click **OK**.



Saving the extracted user control

4. Notice that the selected code was extracted to a user control and the original location of the selected code was replaced with an instance of the new user control.



Page automatically updated to use the new user control

5. Press **F5** to run the page and verify that the control works.

Exercise 3: What's New in the JavaScript Editor

Writing or editing JavaScript code is not an easy task, especially when your application starts to grow in size and you find yourself dealing with long files and hundreds of functions. Script developers usually have to do some extra work to maintain code legibility and navigate across files. With the inclusion of JavaScript libraries like jQuery, script navigation has become a challenge itself because of the code length.

Visual Studio has renewed the JavaScript editor with the promise to make the code mode accessible and organized. Many Visual Studio features that already existed in C# or VB editors are now implemented in the JavaScript editor: Go To Definition, automatic indentation, documentation and validation when you are writing. With the renewed IntelliSense list you will have the JavaScript function catalog at your fingertips.

In this exercise, you will learn some of the new features and improvements of JavaScript editor. You will browse sample files and discover each of the new characteristics that will make your JavaScript programming more efficient within Visual Studio 2012.

Task 1 - JavaScript Editor New Features

This task will introduce you to some of the new JavaScript editor features, which focus on organizing your code and bringing a better user experience.

1. If not already opened, start **Visual Studio** and open the **WhatsNewASPNET.sln** solution located in the **Source\WhatsNewASPNET** folder of this lab.

2. Press **F5** to run the application, then click the JavaScript link in the navigation bar. Refresh the page several times and check how the counter increments.

Page counter

3. Close the browser and go back to Visual Studio.

4. Open the **JavaScript.aspx** page and locate the **<script>** block (shown below).

   The following code uses HTML5 local storage to store a *pageLoadCount* variable that stores the number of times the page has been visited by the current user. Local Storage is a client-side key-value database introduced with the HTML5 standard. The data is saved on the local machine, inside the user's browser.

```
<script>
  addCount(1);
  document.getElementById('count').innerHTML = getCount();

  function getCount() {
    var storage = window.localStorage;
    if (!storage.pageLoadCount)
      storage.pageLoadCount = 0;

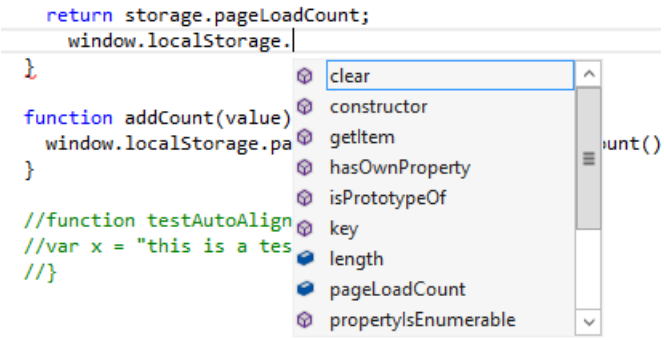
    return storage.pageLoadCount;
  }

  function addCount(value) {
    window.localStorage.pageLoadCount = parseInt(getCount(), 10) + value;
  }

  ...
</script>
```

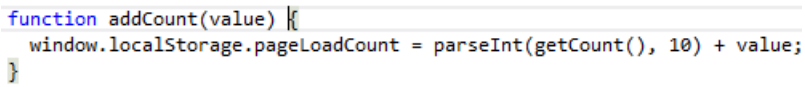
> **NOTE**
>
> Ensure the DOCTYPE is set to XHTML5 before proceeding with the next steps.

5. Edit the code and notice that IntelliSense for JavaScript includes HTML5 features, like local storage, and their inner methods.



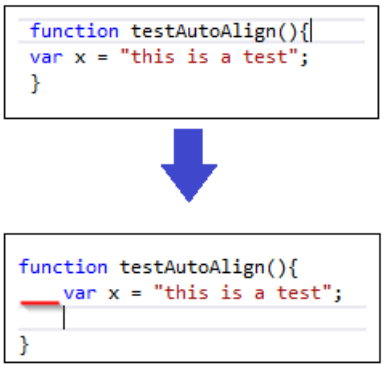
HTML5 JavaScript features in JavaScript

6. Click any opening bracket ({) from the scripting code and notice that the brackets are highlighted.



Brackets are highlighted

7. Uncomment the function **testAutoAlign()** (select the three lines and you can use **CTRL** + **K**; **CTRL** + **U**) and locate the cursor inside the function code. Press enter to append a second line. Notice that the code is now **aligned** and **auto-indented**.



JavaScript code is auto aligned

Task 2 - Validating JavaScript

In this task, you will discover the new JavaScript validation for the ECMAScript5 standard. This feature will help you to write compliant JavaScript code, while preventing scripting issues before site deployment.

> **NOTE**
>
> Visual Studio 2010 implemented ECMAStript3 compliance, while Visual Studio 2012 provides ECMAScript5 compliance.

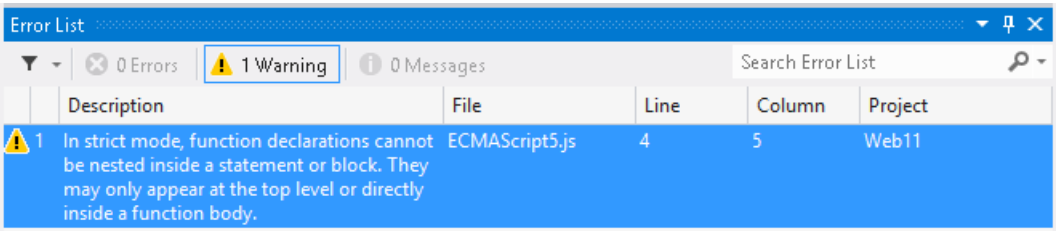
1. Open **ECMA5script5.js** located under the **Scripts\custom** project folder. You will now test validation for ECMAScript5 standard.

   ```
   "use strict";

   if (true) {
       function StrictModeError()
       {
       }
   }
   ```

   You can check out the " **use strict** " direction in the first line of the file, which enables ECMAScript5 **strict mode**. This mode consists in a subset of the language that clarifies ambiguities from the past edition, and adds some new features, such as getters and setters, library support for JSON, and more complete reflection on object properties.

2. Open the **Error List** if not already opened (**View** menu | **Error List**). Notice the **function** declaration is underlined. This is because in ECMA5 standard functions cannot be nested inside language structures. In the error list below you will see the warning details.

   JavaScript validation error message

3. Comment out the **"use strict"** direction and notice that errors disappear, but the warnings remain.

4. In the last line of the file, write any string like **"test"** (include the quotation marks to indicate it is as string). Write a period next to the string to display the IntelliSense list, and select the **trim** option.

   In ECMAScript5 standard, string values and variables also have string methods defined, like trim, uppercase, search and replace.

```
// trim is defined in ECMAScript 5's string
"test".tr
```

constructor
strike
substr
substring
toLocaleString
toString
trim — String trim()

IntelliSense list in JavaScript

Task 3 - XML Documentation for JavaScript

In this task, you will explore Visual Studio features for XML documentation in JavaScript. You will see the JavaScript IntelliSense list now shows the XML documentation of each function. Additionally, you will discover the navigation feature in JavaScript.

1. Open **XMLDoc.js** file located in **Scripts/custom** project folder. This file contains XML documentation on each of the JavaScript functions.

```
function multiply(a, b) {
    /// <summary>multiply two numbers</summary>
    /// <param name="a" type="String">1st number</param>
    /// <param name="b" type="String">2nd number</param>
    /// <returns type="Number" />
    return a * b;
}

function add(p1, p2, p3) {
    ///<signature>
    /// <summary>Add 2 numbers</summary>
    /// <param name="p1" type="Number">1st number</param>
    /// <param name="p2" type="Number">2nd number</param>
    /// </signature>
    ///<signature>
    /// <summary>Add 3 numbers</summary>
    /// <param name="p1" type="Number">1st number</param>
    /// <param name="p2" type="Number">2nd number</param>
    /// <param name="p3" type="Number">3rd number</param>
    /// </signature>
}
```

JavaScript XML documentation integrated to IntelliSense

2. Below **add** function in **XMLDoc.js** file, create a new function named **test**.

3. In the **test** function, call the **multiply** function that receives two parameters. Notice the tooltip box is showing the **multiply** function documentation.

```
function test() {
  multiply(
}
```

```
function test() {
    multiply(
}
```

Number multiply(**String a**, String b)
multiply two numbers
a: *1st number*

XML documentation for JavaScript functions

4. Complete the function call statement and type a *dot* to open the IntelliSense list on the returned value. Notice that Visual Studio is detecting the **return** value in the documentation, treating the value as a number.

```
function test() {
    multiply(1,2).to
    }
```

constructor
isPrototypeOf
toExponential — String toExponential([Number fractionDigits])
toFixed
toLocaleString
toPrecision
toString

XML documentation for return types

5. Now, insert a call to add function. Notice that the JavaScript editor now supports function overloads. When you write a function name, you will be able to select any of the available overloads specified in the documentation.

```
function test() {
    add(
}
```

▲ 1 of 2 ▼ add(**Number p1**, Number p2)
Add 2 numbers
p1: *1st number*

XML documentation for overloads

6. Open **GotoDefinition.js** file and locate the **$().html()** function call. Locate the cursor on **html**.

7. Press **F12** and navigate to the definition. Notice you can now access and browse your JavaScript code without using the **Find** tool.

8. Locate the cursor on the jQuery instruction prior to the signature block at the bottom of the code file. Press **F12**. You will navigate to the jQuery library file. Notice you can also navigate across the jQuery files using **F12**.

GoToDefinition.js

```
$("#abc").html("test");
```

jquery-1.4.1-vsdoc.js | GoToDefinition.js

```
        // Return the cloned set
        return ret;
    },

    html: function( value ) {
        /// <summary>
        ///     Set the html contents of every matched element.
        ///     This property is not available on XML documents.
        ///     Part of DOM/Attributes
        /// </summary>
        /// <returns type="jQuery" />
        /// <param name="value" type="String">
        ///     A string of HTML to set as the content of each matched element.
        /// </param>

        if ( value === undefined ) {
            return this[0] && this[0].nodeType === 1 ?
```

Navigating to jQuery definitions

> **NOTE**
>
> Make sure that GotoDefinition.js has no syntax errors before saving the file.

Exercise 4: Bundling and Minification

How many times do your websites include more than one JavaScript or CSS file? This is a very common scenario where bundling and minification can help to reduce the file size and make the site perform faster. The new bundling feature in ASP.NET 4.5 packs a set of JS or CSS files into a single element, and reduces its size by minifying the content (i.e. removing not required blank spaces, removing comments, reducing identifiers).

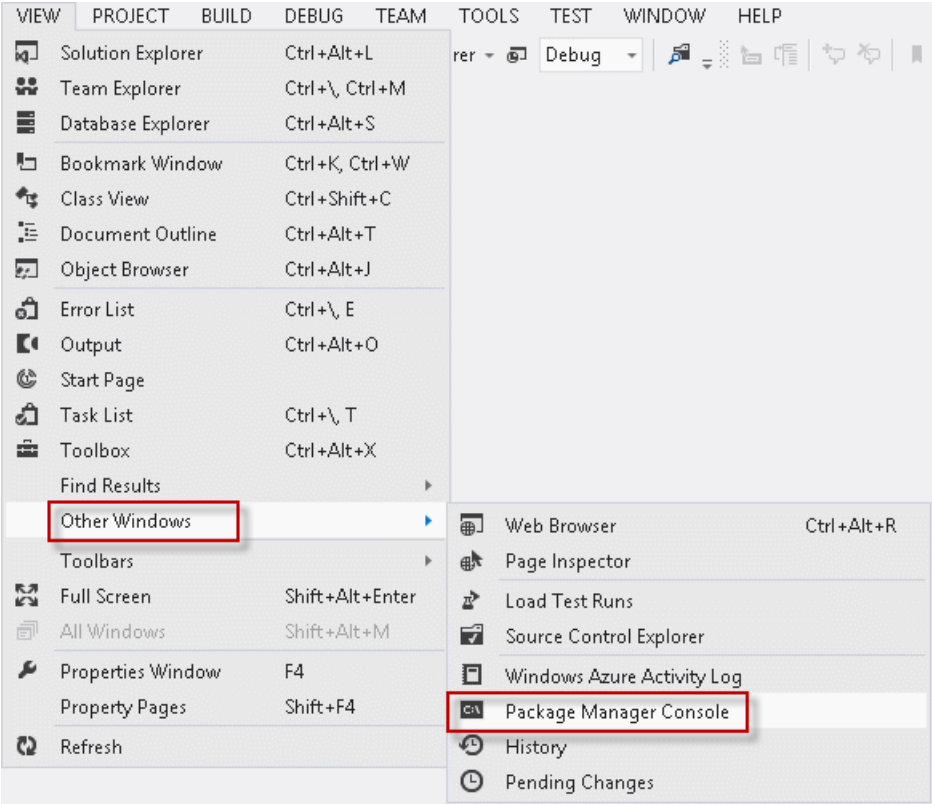
Bundling and minification in ASP.NET 4.5 is performed at runtime, so that the process can identify the user agent (for example IE, Mozilla, etc), and thus, improve the compression by targeting the user browser (for instance, removing stuff that is Mozilla specific when the request comes from IE).

In this exercise, you will learn how to enable and use the different types of bundling and minification in ASP.NET 4.5.

Task 1 - Installing the Bundling and Minification Package from NuGet

1. If not already opened, start **Visual Studio** and open the **WhatsNewASPNET.sln** solution located in the **Source\WhatsNewASPNET** folder of this lab.

2. Open the NuGet Package Manager Console. To do this, use the menu **View | Other Windows | Package Manager Console**.



Opening the package manager console

3. In the **Package Manager Console,** type **Install-Package Microsoft.Web.Optimization** and press **ENTER**.

Task 2 - Default Bundles

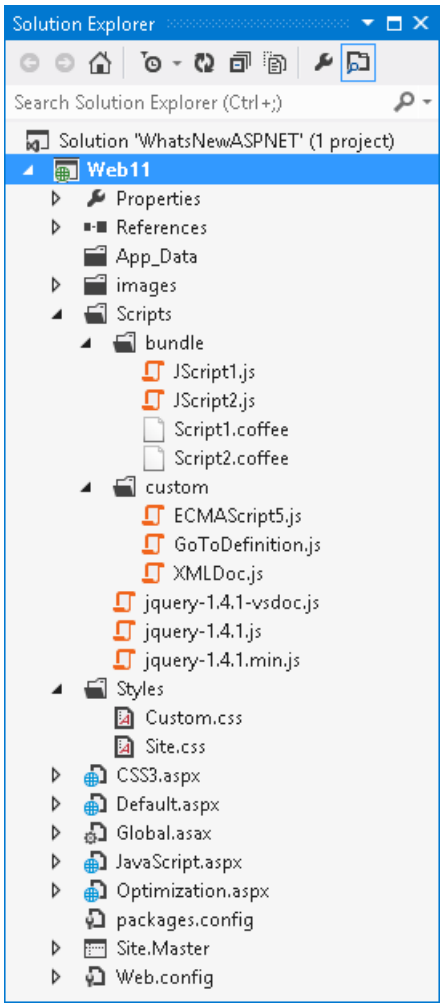
The simplest way to use bundling and minification is to enable the default bundles. This method uses conventions to let you reference the bundled and minified version for the JS and CSS files in a folder.

In this task, you will learn how to enable and reference the bundled and minified JS and CSS files and view the resulting output.

1. If not already opened, start **Visual Studio** and open the **WhatsNewASPNET.sln** solution located in the **Source\WhatsNewASPNET** folder of this lab.

2. In the **Solution Explorer**, expand the **Styles**, **Scripts\custom** and **Scripts\bundle** folders.

Notice that the application is using more than one CSS and JS file.



Multiple Stylesheets and JavaScript files in the application

3. Open the **Global.asax.cs** file.

   Notice that the new **Microsoft.Web.Optimization** namespace is commented out at the beginning of the file. Uncomment the using directive to include the bundling and minification features.

```
using System;
using Microsoft.Web.Optimization;
```

4. Locate the **Application_Start** method.

   In this method, uncomment the EnableDefaultBundles call as shown in the snippet below. This enables us to reference a bundled collection of CSS files in a folder by using the path to that folder, plus the "CSS" or the "JS" suffix.

```
void Application_Start(object sender, EventArgs e)
{
    // Default behavior
    // Bundles all .js files in folders such as "scripts" if URL pointed to it:
http://localhost:54716/scripts/custom/js
    BundleTable.Bundles.EnableDefaultBundles();

    ...
}
```

5. Open the **Optimization.aspx** file and locate the content control for **HeadContent**.

Notice the CSS files and the JS files have a single referenced tag.

```
<asp:Content ID="Content1" ContentPlaceHolderID="HeadContent" runat="server">
  <link href="Styles/CSS" rel="stylesheet" type="text/css" />
  <script src="Scripts/custom/JS"></script>
</asp:Content>
```

> **NOTE**
>
> This code is for demo purposes. Ideally, you will reference the bundles in the Site.Master file. In this sample code, you will find that some of the bundled files are also being referenced by the Site.Master file, making this last reference redundant.

6. Notice that the links are using the bundling conventions in the **href** attribute to get all the CSS or JS files from the Styles and Scripts\custom folder respectively.

   You can use the path **Scripts/custom/JS** as shown below to bundle and minify all the JS files inside a **Scripts/custom** folder. This is the default behavior with the default bundles.

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
  <ul>
    <li>
      <h2><a href="Styles/CSS">CSS Bundle</a></h2>
    </li>
    <li>
      <h2><a href="Scripts/custom/JS">JS Bundle</a></h2>
    </li>
```

7. Open the **Styles\Site.css** file.

   Notice that the original CSS file contains indented code, blank spaces and comments that enlarge the file. (Also the JavaScript file contains blank spaces and comments).

```
/* DEFAULTS
-------------------------------------------------------*/

body
{
    background-color:#b7c2d3 /*rgba(235, 193, 176, 0.80)*/;
    font-size: .80em;
    font-family: "Helvetica Neue", "Lucida Grande", "Segoe UI", Aria
    margin: 0px;
    padding: 0px;
    color: #696969;
}

a:link, a:visited
{
    color:rgba(0, 0, 0, 0.42);
}

a:hover
{
    color: #1d60ff;
    text-decoration: none;
}

a:active
{
    color: #034af3;
}

p
```

One of the original CSS files in the Scripts folder

8. Press **F5** to run the application and navigate to the **Optimization** page.

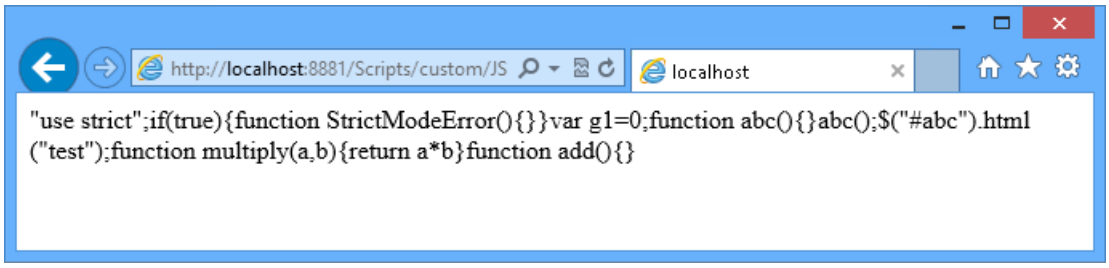
9. Click on the **CSS Bundle** link to download and open the file.

   Check out the minified bundled file. You will notice that all the blank spaces, comments and indentation characters have been removed, generating a smaller file.

```
ul{list-style-type:none;padding:0;margin:0}li{background-repeat:no-
repeat;padding-left:15px;float:left}div.images ul li img{padding-
top:50px;padding-right:15px;width:150px;display:inline}#map
{position:absolute;top:80%;left:50%;width:500px;height:300px;margin-
left:-400px;margin-top:-300px}.tr13{background:url(/images/alert-
overlay.png) repeat-x;background-color:rgba
(75,108,158,.78);display:inline-block;padding:5px 10px
6px;color:#fff;text-decoration:none;font-weight:bold;line-height:1;-
moz-border-radius:5px;-webkit-border-radius:5px;-moz-box-shadow:0 1px
3px #999;-webkit-box-shadow:0 1px 3px #999;text-shadow:0 -1px 1px
#222;border-bottom:1px solid #222;position:relative;cursor:pointer}
```

Bundled CSS files

10. Now click the **JS Bundle** link to open the JavaScript bundled file. You can safely disregard the explorer warning. Notice the JavaScript files under the **custom** folder are also bundled and minified.



Bundled JavaScript files

Enabling compression for CSS or JS files was much more complicated in previous ASP.NET version. Now, as you have seen, you just need to add one line in the *Global.asax* file to enable bundling, and then reference the bundled files from your site.

Task 3 - Static Bundles

The static bundle approach allows you to customize the set of files to bundle, the reference and the minification method that will be used.

In this task, you will configure a static bundle to define a specific set of files to bundle and minify.

1. Close the browser.
2. Open the **Global.asax.cs** file and locate the **Application_Start** method.
3. Uncomment the static bundle code as shown in the code below.

   You are defining a static bundle that will be referenced with the "**~/StaticBundle**" virtual path and use **JsMinify** for minification of all the specified files with the **AddFile** method. Finally, you are adding the static bundle to the **BundleTable** and enabling it.

   Notice that the files are not located in the same place; this is another advantage over the default bundling.

```
void Application_Start(object sender, EventArgs e)
{
    // Default behavior
    // Bundles all .js files in folders such as "scripts" if URL pointed to it:
http://localhost:54716/scripts/custom/js
    BundleTable.Bundles.EnableDefaultBundles();

    // Static bundle.
    // Access on url http://localhosthost:54716/StaticBundle
    Bundle b = new Bundle("~/StaticBundle", typeof(JsMinify));
    b.AddFile("~/scripts/custom/ECMAScript5.js");
    b.AddFile("~/scripts/custom/GoToDefinition.js");
    b.AddFile("~/scripts/bundle/JScript1.js");
    b.AddFile("~/scripts/bundle/JScript2.js");

    BundleTable.Bundles.Add(b);

    // Dynamic bundle
    // Bundles all .coffee files in folders such as "script" when "coffee" is appended to it:
http://localhost:54716/scripts/coffee
    // DynamicFolderBundle fb = new DynamicFolderBundle("coffee", typeof(CoffeeMinify), "*.coffee");
    // BundleTable.Bundles.Add(fb);
}
```

4. Open the **Optimization.aspx** file.

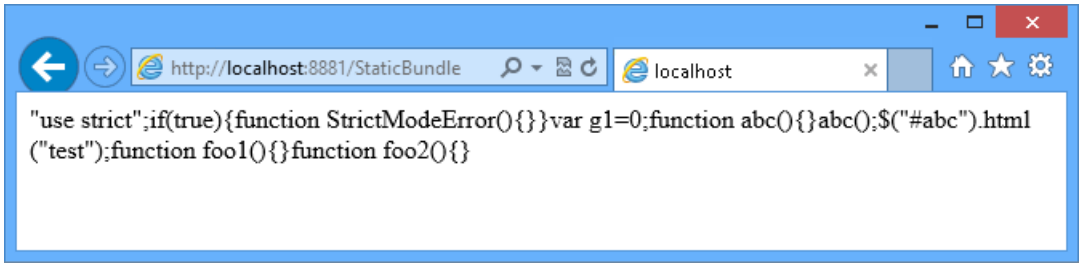
   Notice that the link to **Static JS Bundle** is using the path you have declared when you configured the static bundle in the Global.asax.cs file: **/StaticBundle**.

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
  <ul>
    <li>
      <h2><a href="Styles/CSS">CSS Bundle</a></h2>
    </li>
    <li>
      <h2><a href="Scripts/custom/JS">JS Bundle</a></h2>
    </li>
    <li>
      <h2><a href="StaticBundle">Static JS Bundle</a></h2>
    </li>
    <li>
      <h2><a href="Scripts/bundle/Coffee">Dynamic JS Bundle</a></h2>
    </li>
  </ul>
</asp:Content>
```

5. Press **F5** to run the application, and then navigate to the **Optimization** page.

6. Click on the **Static JS Bundle** link to open the file.

   Notice that the minified bundled JavaScript file is the output for all the JavaScript files configured in the static bundle file under the path "/StaticBundle".



   Static JavaScript files bundle

7. Close the browser and return to Visual Studio.

Task 4 - Dynamic Folder Bundles

In this task, you will learn how to configure dynamic folder bundles. The power of dynamic bundling is that you can include static JavaScript, as well as other files in languages that compiles into JavaScript, and thus, require some processing before the bundling is executed.

In this example, you will learn how to use the **DynamicFolderBundle** class to create a dynamic bundle for files written in CofeeScript. CofeeScript is a programming language that compiles into JavaScript and provides a simpler syntax for writing JavaScript code, enhancing JavaScript's brevity and readability.

1. Open the **Global.asax.cs** file and locate the **Application_Start** method.

2. Uncomment the dynamic bundle code as shown in the code below.

   You are defining a dynamic folder bundle that will use the **CoffeeMinify** custom minification processor that will only apply to the files with the "**.coffee**" extension (CoffeeScript files). Notice that you can use a search pattern to select the files to bundle within a folder, like '*.coffee'.

```
void Application_Start(object sender, EventArgs e)
{
    // Default behavior
    // Bundles all .js files in folders such as "scripts" if URL pointed to it:
http://localhost:54716/scripts/custom/js
    BundleTable.Bundles.EnableDefaultBundles();

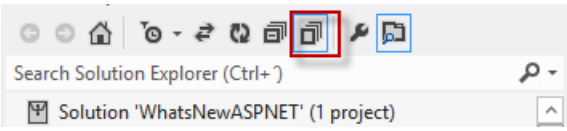
    // Static bundle.
    // Access on url http://localhosthost:54716/StaticBundle
    ...

    // Dynamic bundle
    // Bundles all .coffee files in folders such as "script" when "coffee" is appended to it:
         // http://localhost:54716/scripts/coffee
    DynamicFolderBundle fb = new DynamicFolderBundle("coffee", typeof(CoffeeMinify), "*.coffee");
    BundleTable.Bundles.Add(fb);
}
```

3. Open the NuGet Package Manager Console. To do this, use the menu **View | Other Windows | Package Manager Console**.

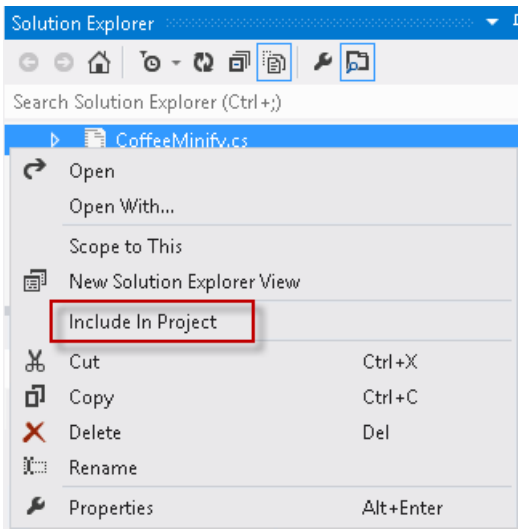
4. In the **Package Manager Console**, type **Install-Package CoffeeSharp** and press **ENTER**.

5. Click the **Show All Files** button in the **Solution Explorer** window



Showing all files

6. Right click the **CoffeeMinify.cs** file in the **Solution Explorer** and select **Include in Project**



Include the CoffeeMinify.cs file in the project

7. Open the **CoffeeMinify.cs** file.

   This class inherits from JsMinify to minify the JavaScript output resulting from the CoffeeScript code compilation. It calls the CoffeeScript compiler to generate the JavaScript code first, and then it sends it to the JsMinify.Process method to minify the resulting code.

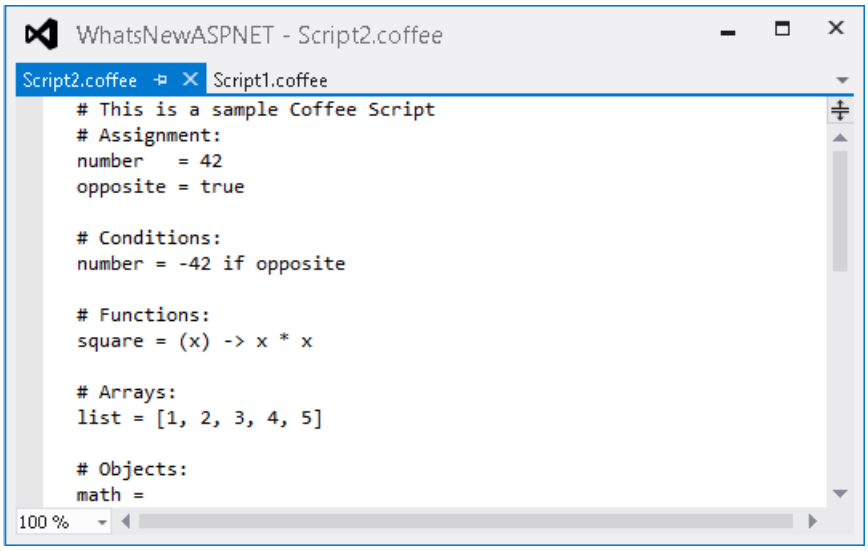
```
public class CoffeeMinify : JsMinify
{
    public override void Process(BundleResponse bundle)
    {
        var compiler = new CoffeeScriptEngine();
        bundle.Content = compiler.Compile(bundle.Content);

        base.Process(bundle);
    }
}
```

8. Open the **Script1.coffee** and **Script2.coffee** files from the **Scripts/bundle** folder.

   These files will include the CoffeScript code to be compiled while performing the bundling with the CoffeeMinify class.

   For simplicity purposes, the CoffeeScript files provided are only including CoffeeScript sample code. The comments are excluded by the JsMinify process.



   CoffeeScript files

> **NOTE**
>
> CofeeScript provides a simpler syntax for writing JavaScript code, enhancing JavaScript's brevity and readability, as well as adding other features like array comprehension and pattern matching.

9. Open the **Optimization.aspx** file and locate the bundle links.

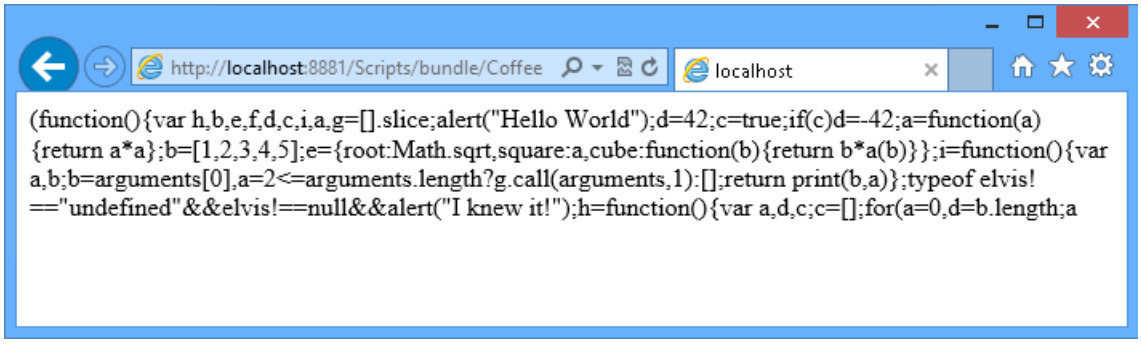
   Notice that the link to **Dynamic JS Bundle** is referencing the **Scripts/bundle** folder by using the **/Coffee** suffix you configured for the dynamic folder bundle.

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
  <ul>
    <li>
      <h2><a href="Styles/CSS">CSS Bundle</a></h2>
    </li>
    <li>
      <h2><a href="Scripts/custom/JS">JS Bundle</a></h2>
    </li>
    <li>
      <h2><a href="StaticBundle">Static JS Bundle</a></h2>
    </li>
    <li>
      <h2><a href="Scripts/bundle/Coffee">Dynamic JS Bundle</a></h2>
    </li>
  </ul>
</asp:Content>
```

10. Press **F5** to run the application, and then navigate to the **Optimization** page.

11. Click on the **Dynamic JS Bundle** link to open the generated file.

    Notice that the content that was included in this bundle only contains **.coffee** files. You can also see that the CoffeeScript code was compiled to JavaScript and the commented-out lines has been removed.



Dynamic JS files bundle

> **NOTE**
>
> Additionally, you can deploy this application to Windows Azure Web Sites following Appendix B: Publishing an ASP.NET MVC 4 Application using Web Deploy.

Summary

This lab helps you to understand what New in ASP.NET and Web Development in Visual Studio 2012 is and how to take advantage of the variety of enhancements in Visual Studio 2012.

By completing this Hands-On Lab, you have learnt how to use the new features and improvements in Visual Studio 2012 Editors for CSS, JavaScript and HTML. In addition, you have learnt how Visual Studio 2012 implements built-in bundling and minification.

Appendix A: Installing Visual Studio Express 2012 for Web

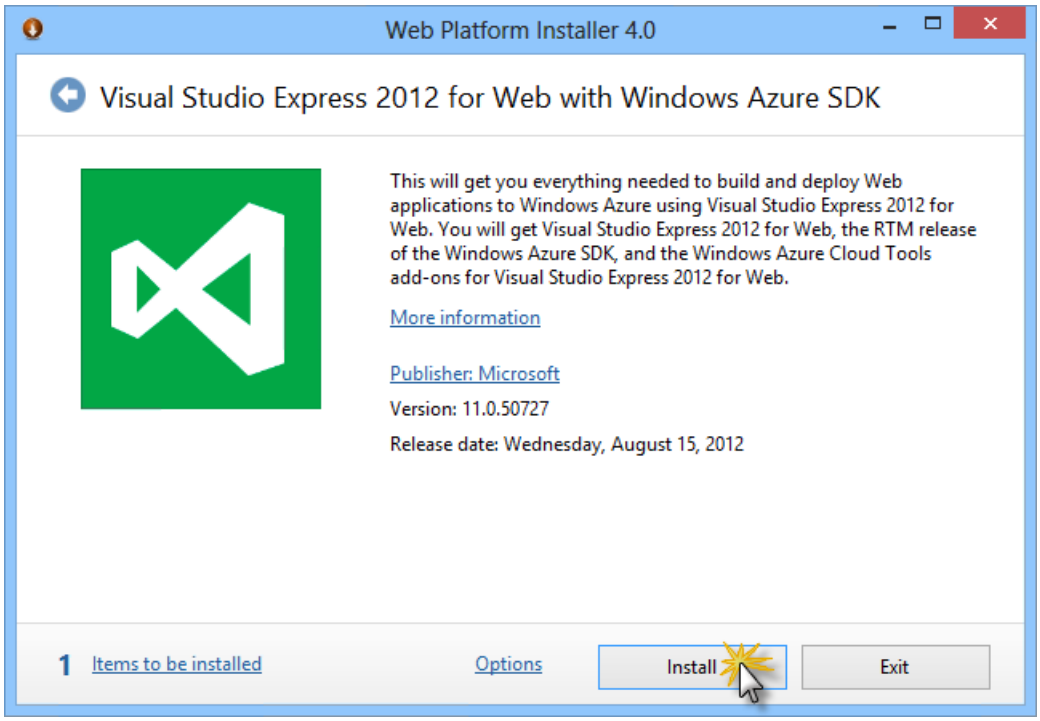
You can install **Microsoft Visual Studio Express 2012 for Web** or another "Express" version using the **Microsoft Web Platform Installer**. The following instructions guide you through the steps required to install *Visual studio Express 2012 for Web* using *Microsoft Web Platform Installer*.

1. Go to https://go.microsoft.com/?linkid=9810169.
   Alternatively, if you already have installed Web Platform Installer, you can open it and search for the product

"*Visual Studio Express 2012 for Web with Windows Azure SDK*".

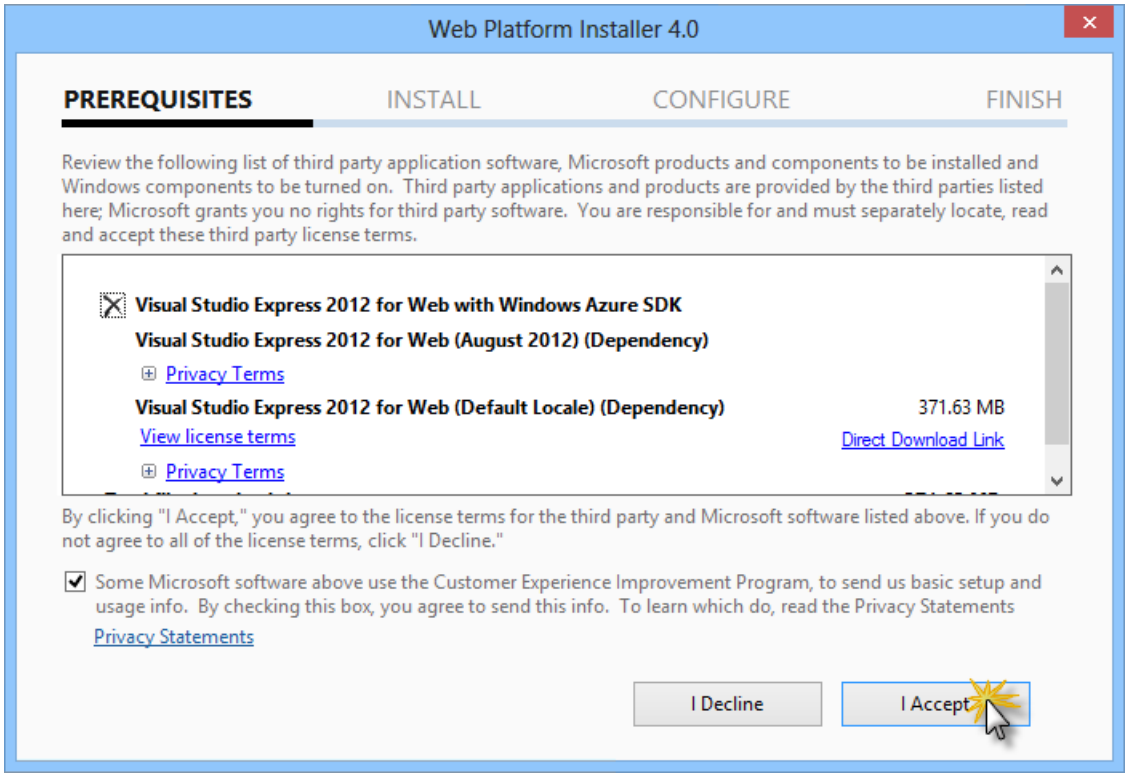
2. Click on **Install Now**. If you do not have **Web Platform Installer** you will be redirected to download and install it first.

3. Once **Web Platform Installer** is open, click **Install** to start the setup.



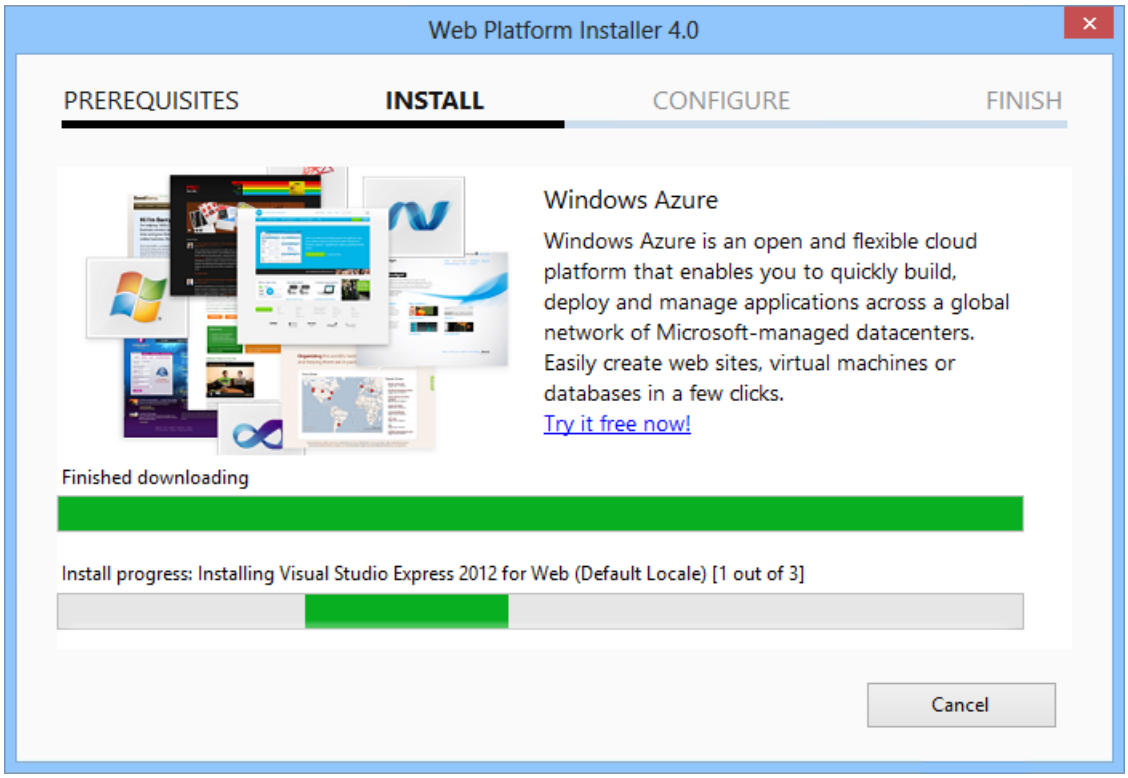
Install Visual Studio Express

4. Read all the products' licenses and terms and click **I Accept** to continue.



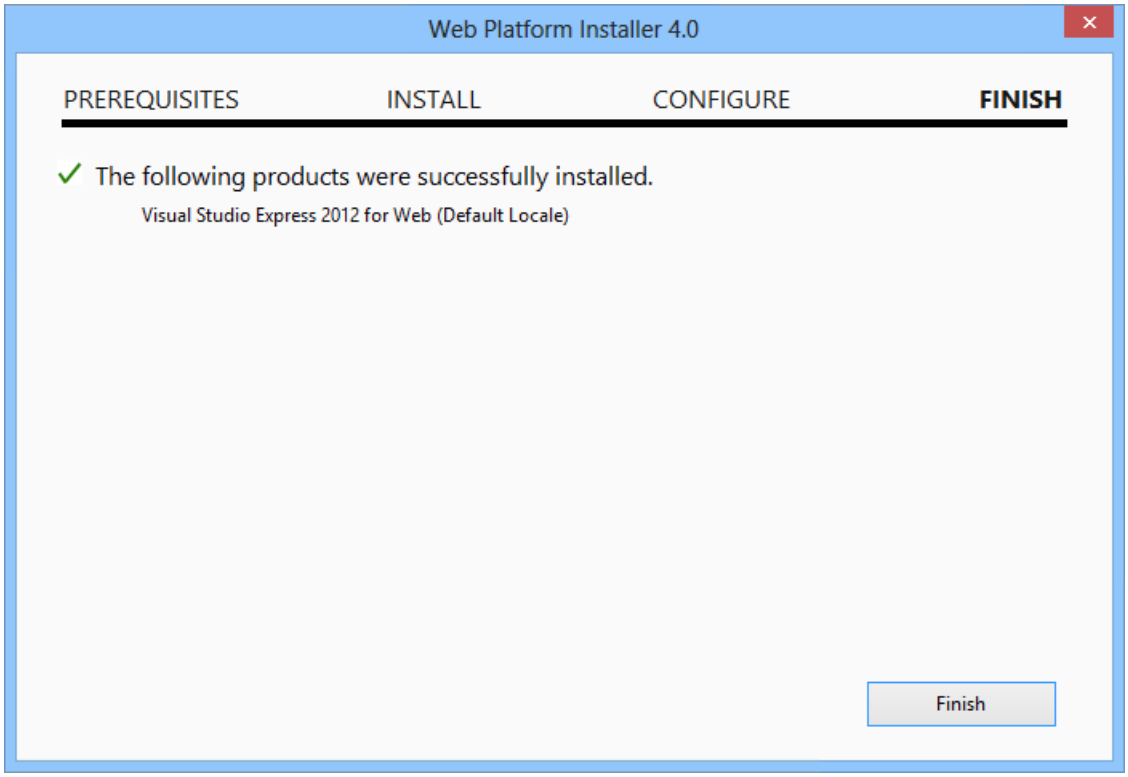
Accepting the license terms

5. Wait until the downloading and installation process completes.



Installation progress

6. When the installation completes, click **Finish**.



Installation completed

7. Click **Exit** to close Web Platform Installer.

8. To open Visual Studio Express for Web, go to the **Start** screen and start writing "**VS Express**", then click on the **VS Express for Web** tile.

VS Express for Web tile

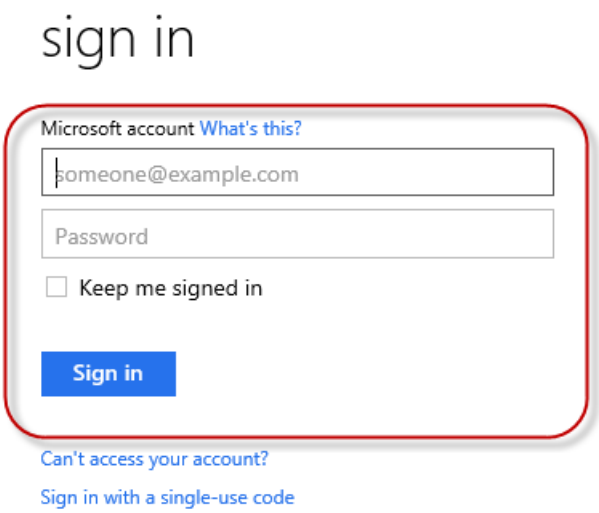
Appendix B: Publishing an ASP.NET MVC 4 Application using Web Deploy

This appendix will show you how to create a new web site from the Windows Azure Management Portal and publish the application you obtained by following the lab, taking advantage of the Web Deploy publishing feature provided by Windows Azure.

Task 1 - Creating a New Web Site from the Windows Azure Portal

1. Go to the Windows Azure Management Portal and sign in using the Microsoft credentials associated with your subscription.

> **NOTE**
> With Windows Azure you can host 10 ASP.NET Web Sites for free and then scale as your traffic grows. You can sign up here.



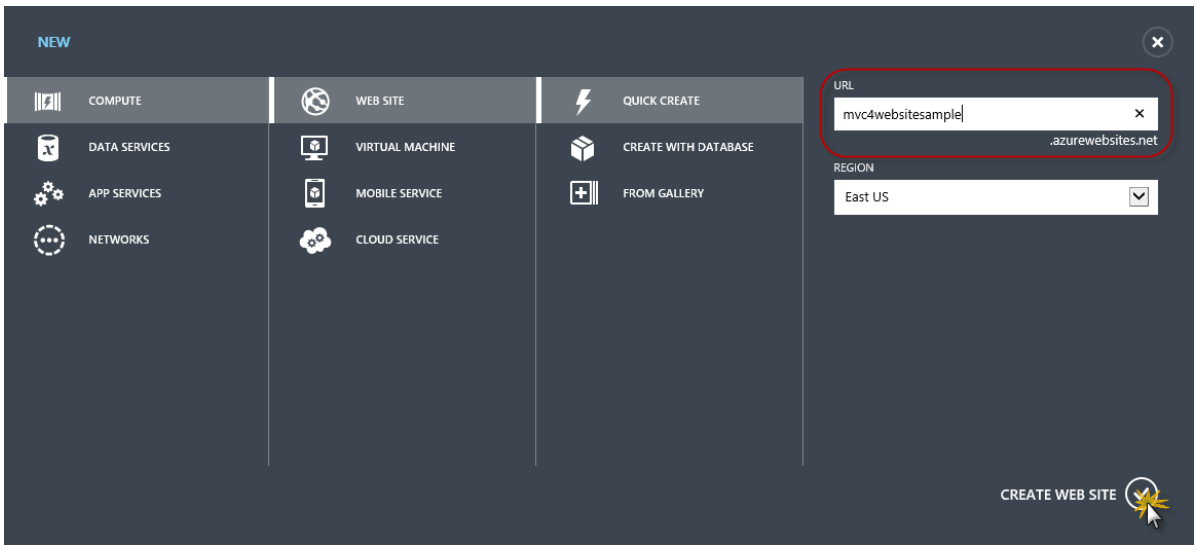
Log on to Windows Azure Management Portal

2. Click **New** on the command bar.

Creating a new Web Site

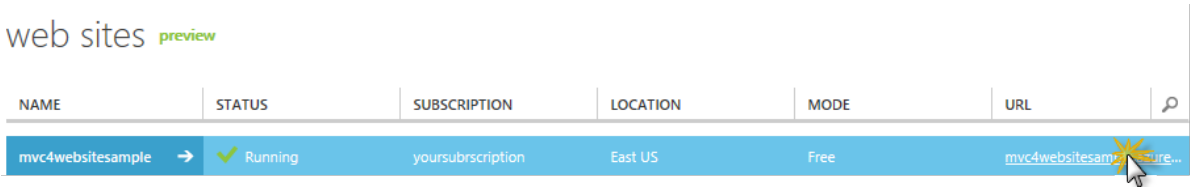
3. Click **Compute | Web Site**. Then select **Quick Create** option. Provide an available URL for the new web site and click **Create Web Site**.

> **NOTE**
>
> A Windows Azure Web Site is the host for a web application running in the cloud that you can control and manage. The Quick Create option allows you to deploy a completed web application to the Windows Azure Web Site from outside the portal. It does not include steps for setting up a database.

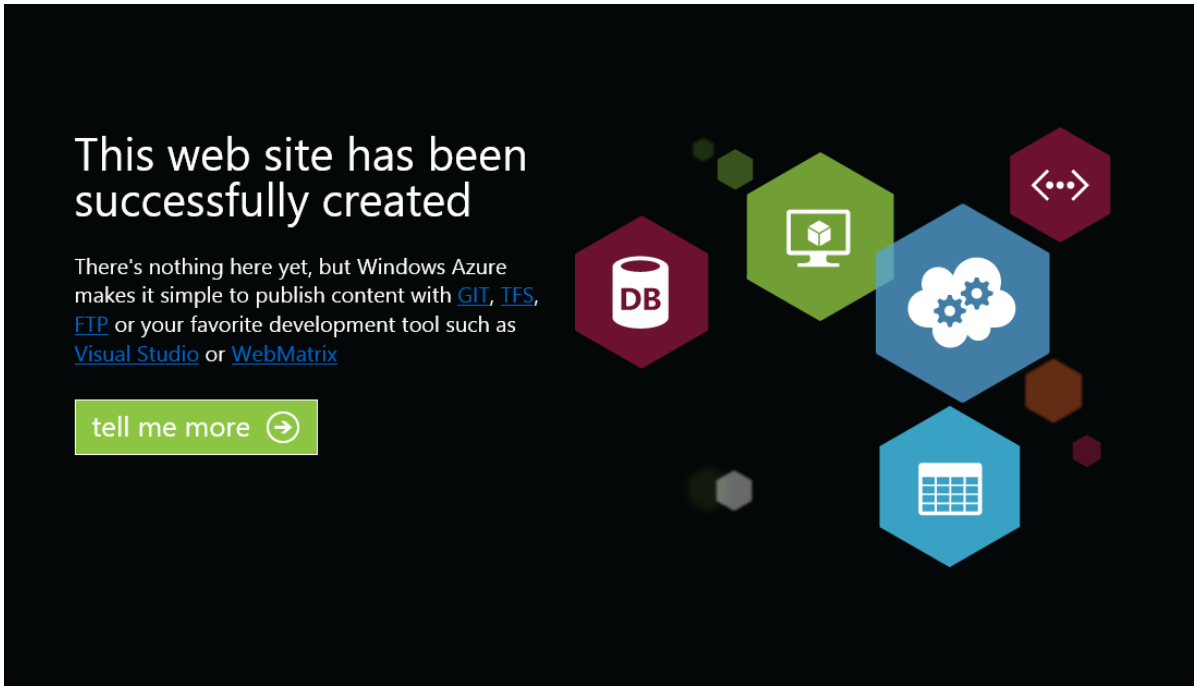


Creating a new Web Site using Quick Create

4. Wait until the new **Web Site** is created.
5. Once the Web Site is created click the link under the **URL** column. Check that the new Web Site is working.

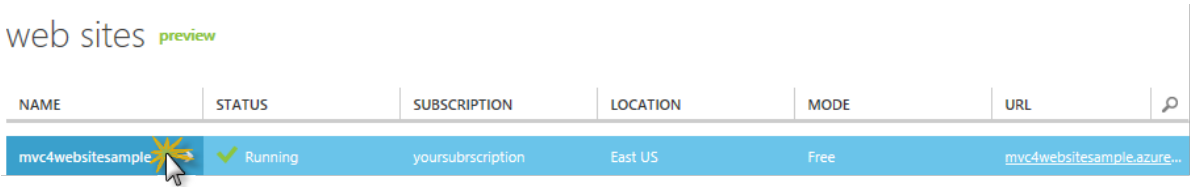


Browsing to the new web site



Web site running

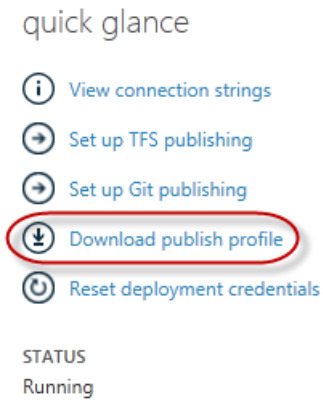
6. Go back to the portal and click the name of the web site under the **Name** column to display the management pages.



Opening the Web Site management pages

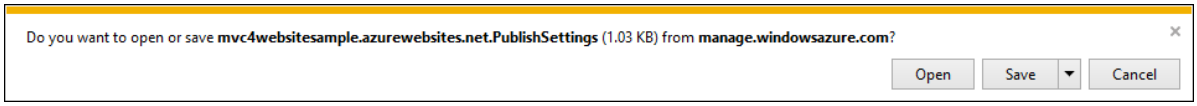
7. In the **Dashboard** page, under the **quick glance** section, click the **Download publish profile** link.

> **NOTE**
>
> The *publish profile* contains all of the information required to publish a web application to a Windows Azure website for each enabled publication method. The publish profile contains the URLs, user credentials and database strings required to connect to and authenticate against each of the endpoints for which a publication method is enabled. **Microsoft WebMatrix 2**, **Microsoft Visual Studio Express for Web** and **Microsoft Visual Studio 2012** support reading publish profiles to automate configuration of these programs for publishing web applications to Windows Azure websites.



Downloading the Web Site publish profile

8. Download the publish profile file to a known location. Further in this exercise you will see how to use this file to publish a web application to a Windows Azure Web Sites from Visual Studio.

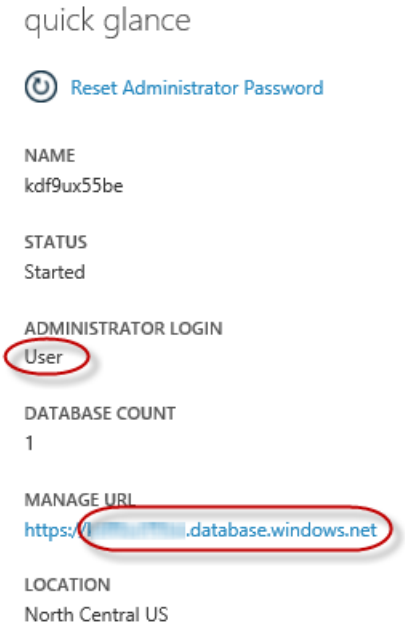


Saving the publish profile file

Task 2 - Configuring the Database Server

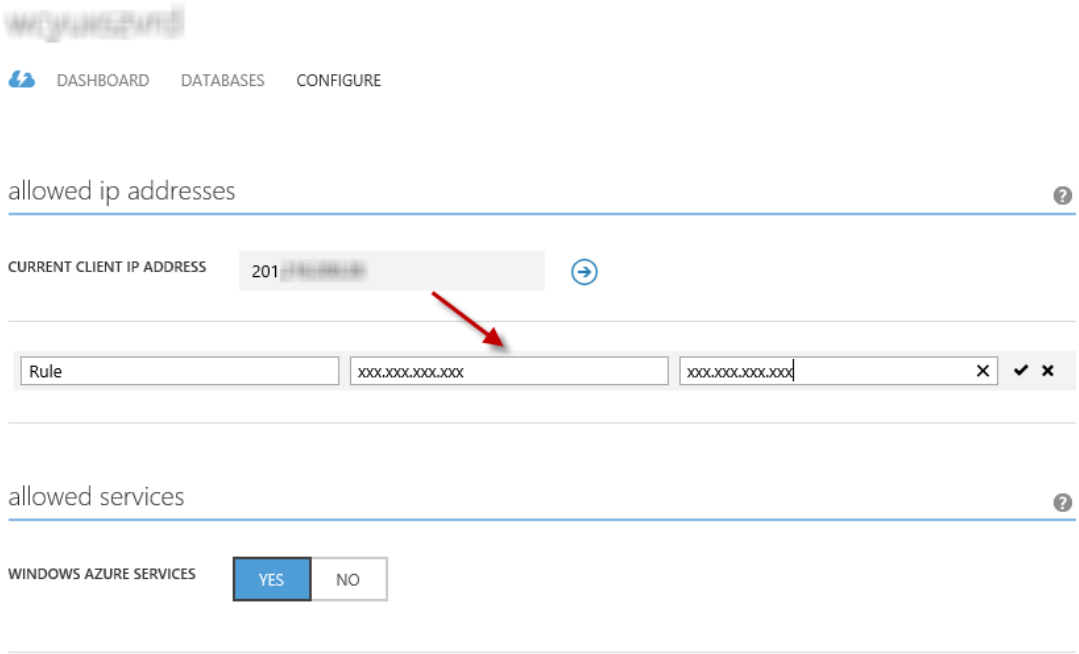
If your application makes use of SQL Server databases you will need to create a SQL Database server. If you want to deploy a simple application that does not use SQL Server you might skip this task.

1. You will need a SQL Database server for storing the application database. You can view the SQL Database servers from your subscription in the Windows Azure Management portal at **Sql Databases | Servers | Server's Dashboard**. If you do not have a server created, you can create one using the **Add** button on the command bar. Take note of the **server name and URL, administrator login name and password**, as you will use them in the next tasks. Do not create the database yet, as it will be created in a later stage.



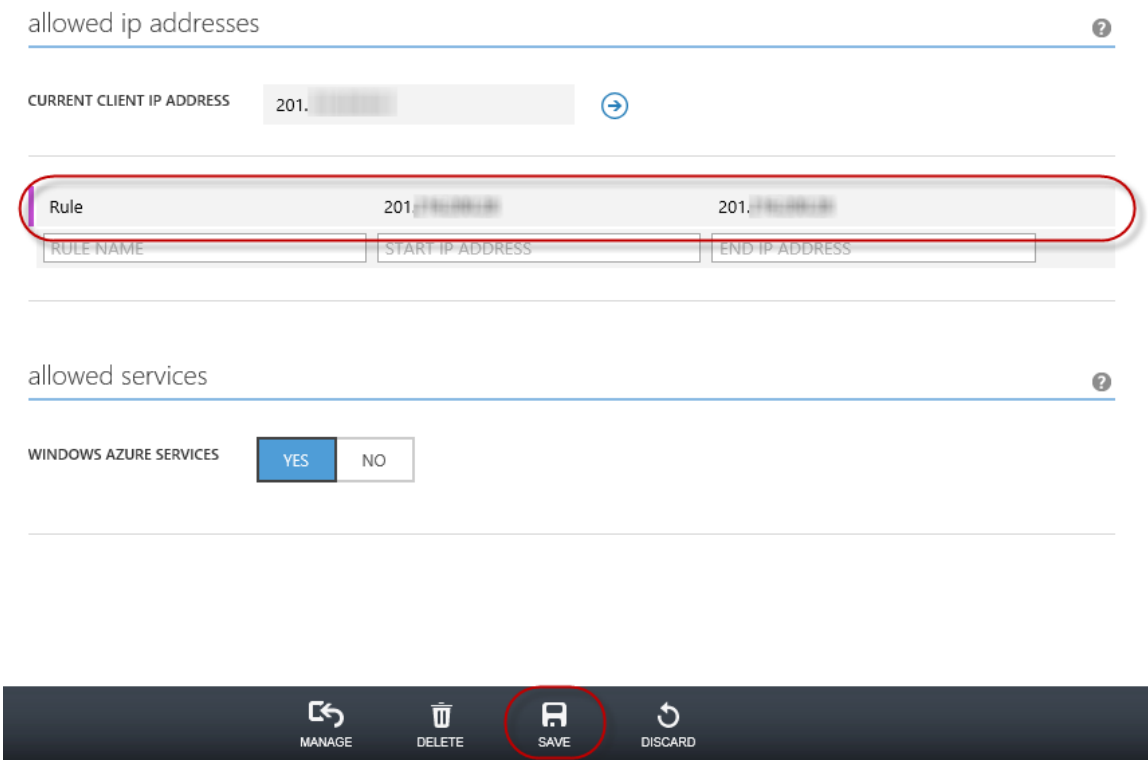
SQL Database Server Dashboard

2. In the next task you will test the database connection from Visual Studio, for that reason you need to include your local IP address in the server's list of **Allowed IP Addresses**. To do that, click **Configure**, select the IP address from **Current Client IP Address** and paste it on the **Start IP Address** and **End IP Address** text boxes. Enter a name for the rule and click the ✔ button.



Adding Client IP Address

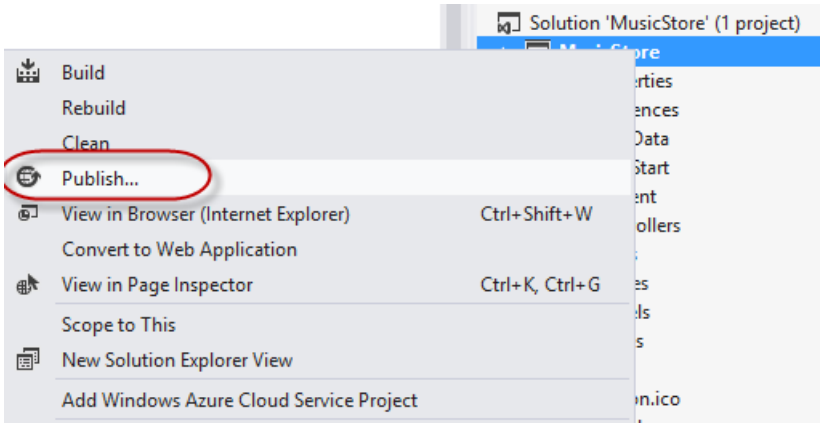
3. Once the **Client IP Address** is added to the allowed IP addresses list, click on **Save** to confirm the changes.



Confirm Changes

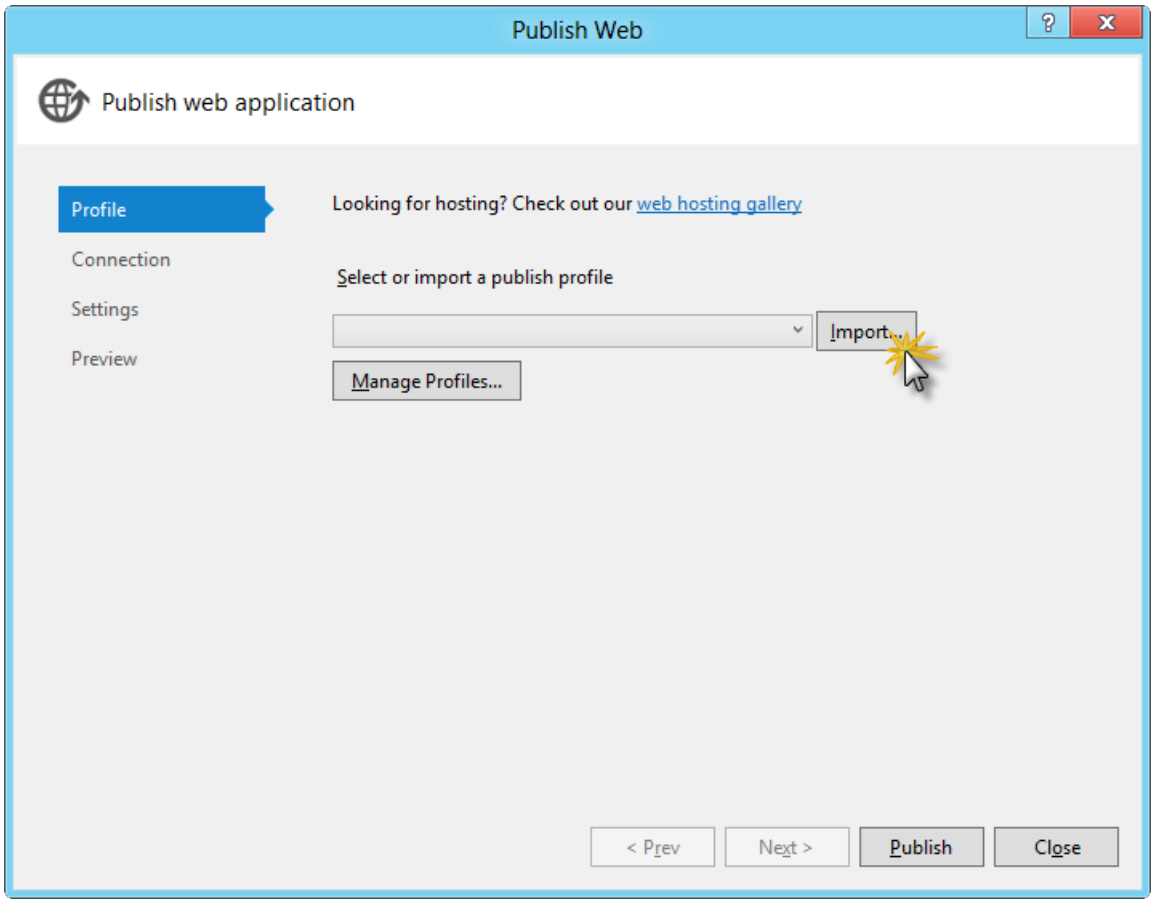
Task 3 - Publishing an ASP.NET MVC 4 Application using Web Deploy

1. Go back to the ASP.NET MVC 4 solution. In the **Solution Explorer**, right-click the web site project and select **Publish**.



Publishing the web site

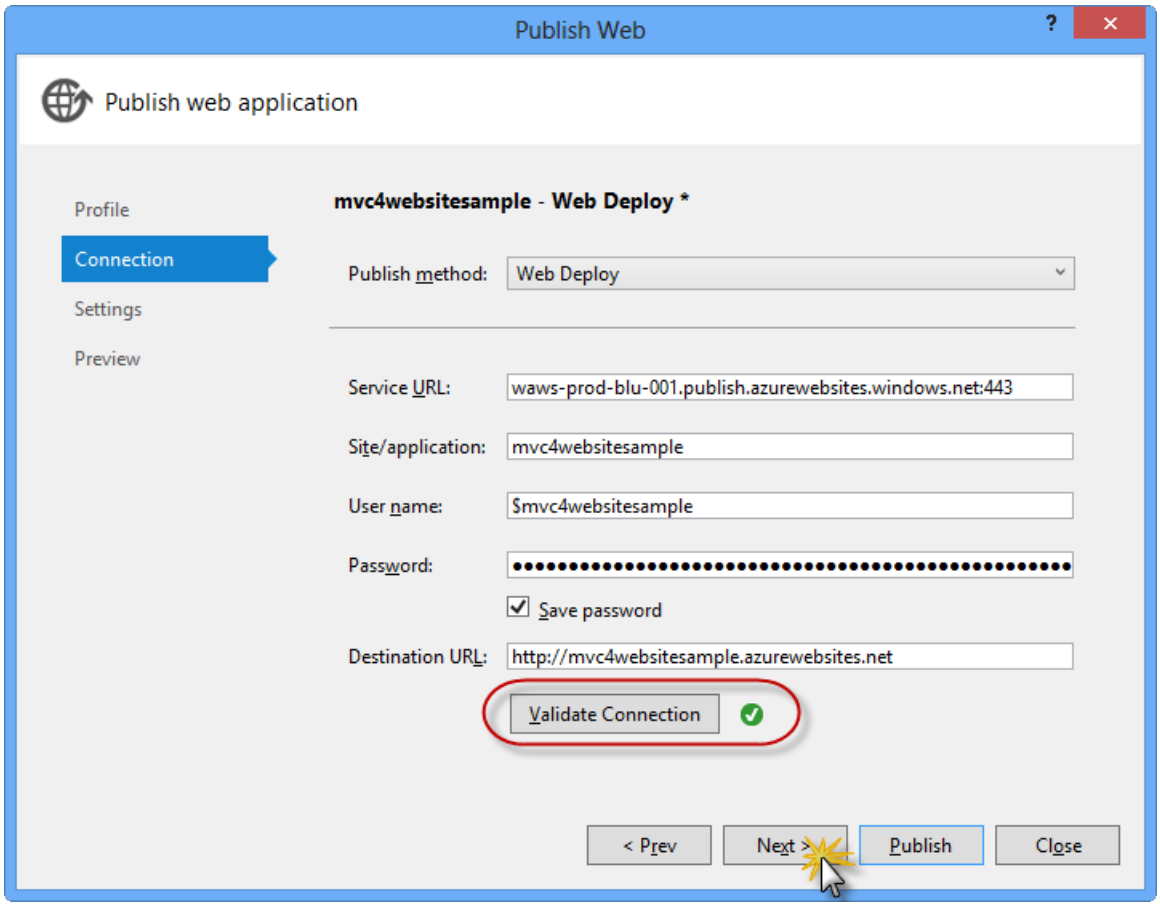
2. Import the publish profile you saved in the first task.



Importing publish profile

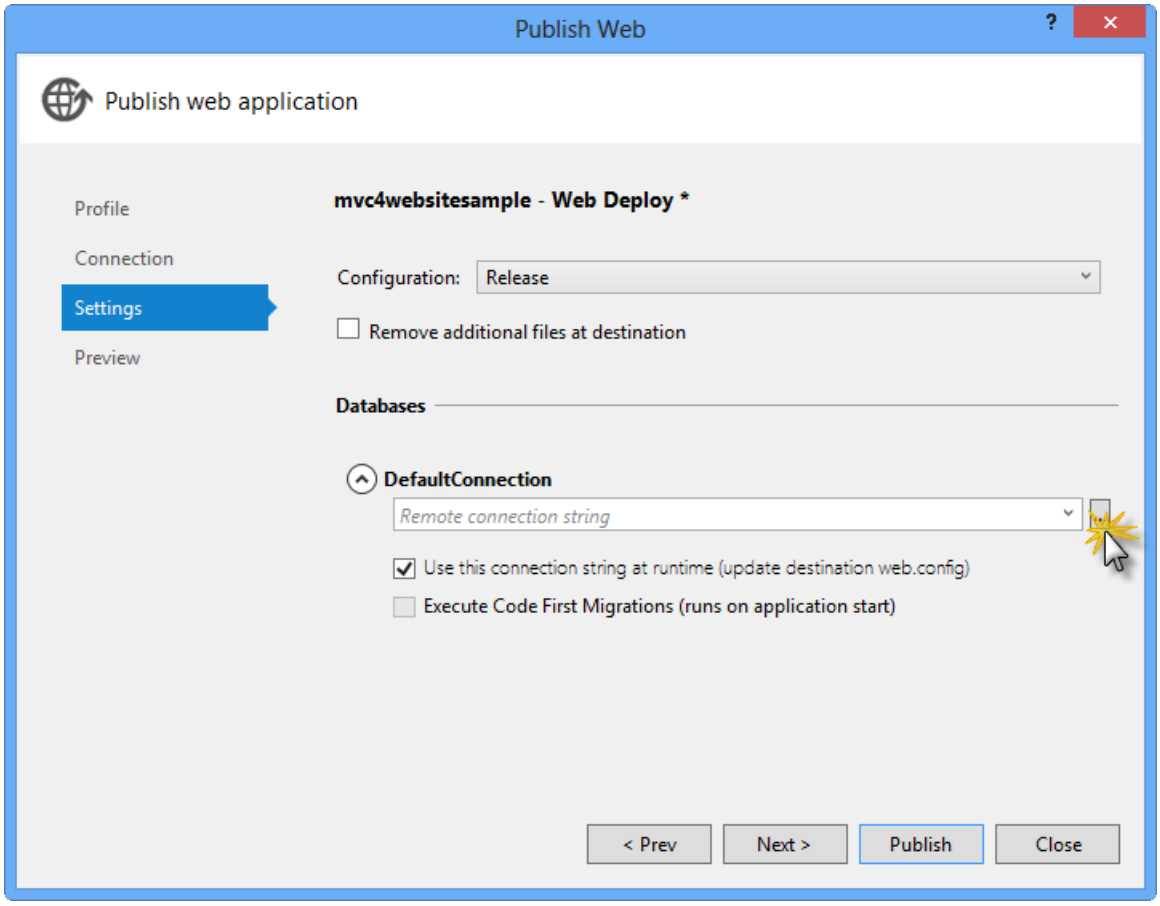
3. Click **Validate Connection**. Once Validation is complete click **Next**.

> **NOTE**
>
> Validation is complete once you see a green checkmark appear next to the Validate Connection button.



Validating connection

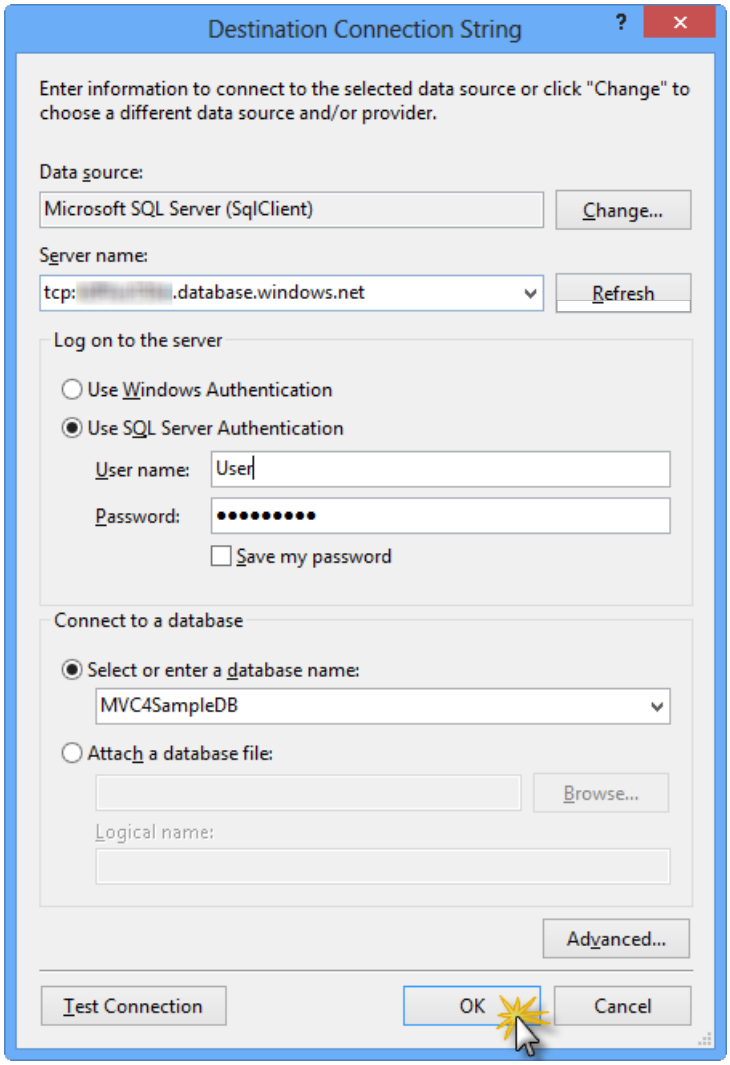
4. In the **Settings** page, under the **Databases** section, click the button next to your database connection's textbox (i.e. **DefaultConnection**).



Web deploy configuration

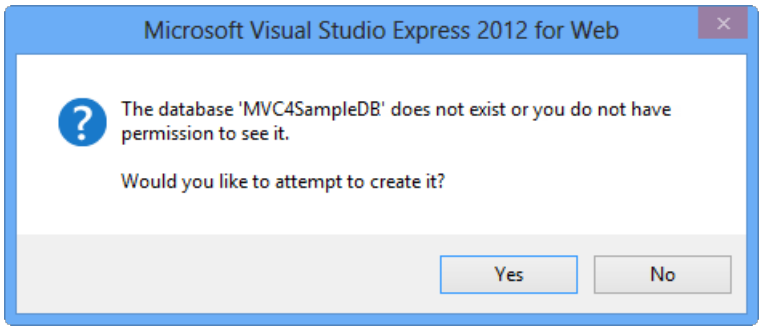
5. Configure the database connection as follows:

   - In the **Server name** type your SQL Database server URL using the *tcp:* prefix.
   - In **User name** type your server administrator login name.
   - In **Password** type your server administrator login password.
   - Type a new database name, for example: *MVC4SampleDB*.



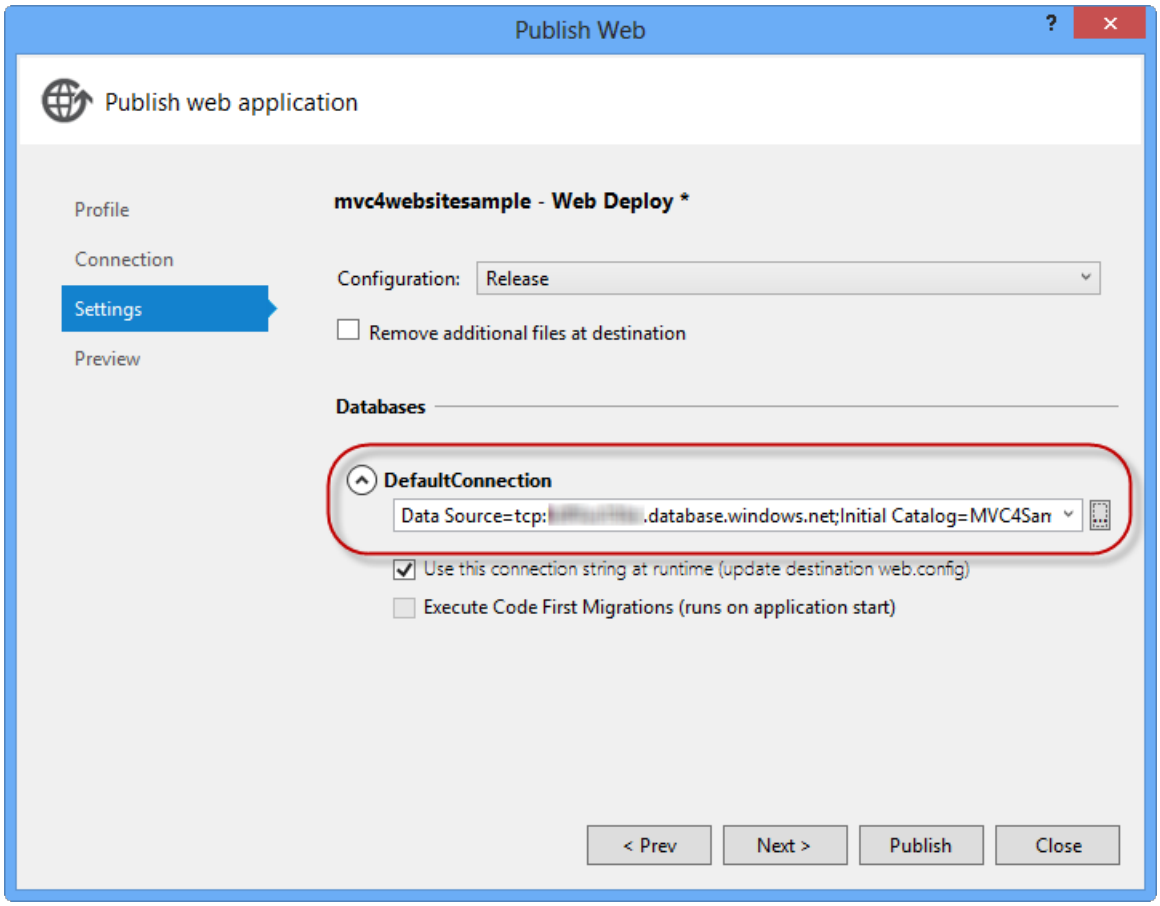
Configuring destination connection string

6. Then click **OK**. When prompted to create the database click **Yes**.



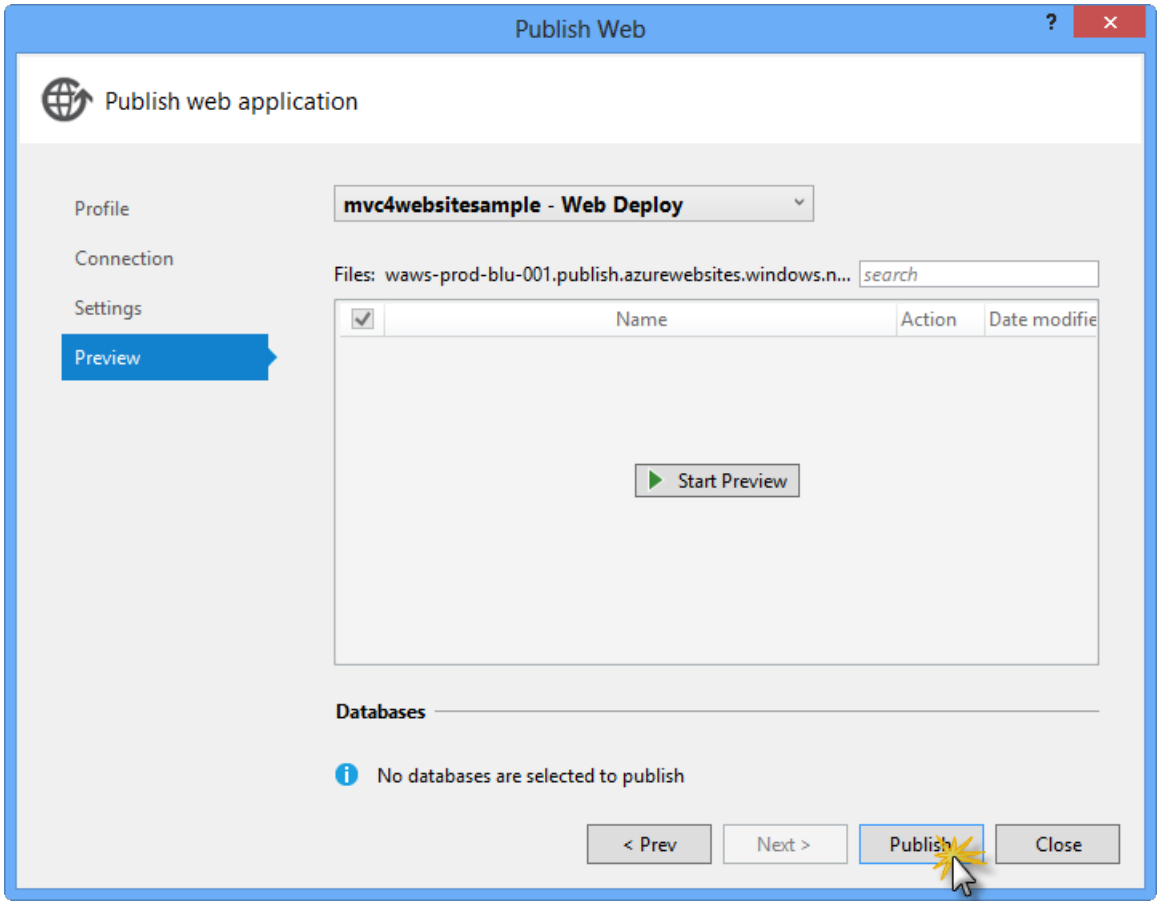
Creating the database

7. The connection string you will use to connect to SQL Database in Windows Azure is shown within Default Connection textbox. Then click **Next**.



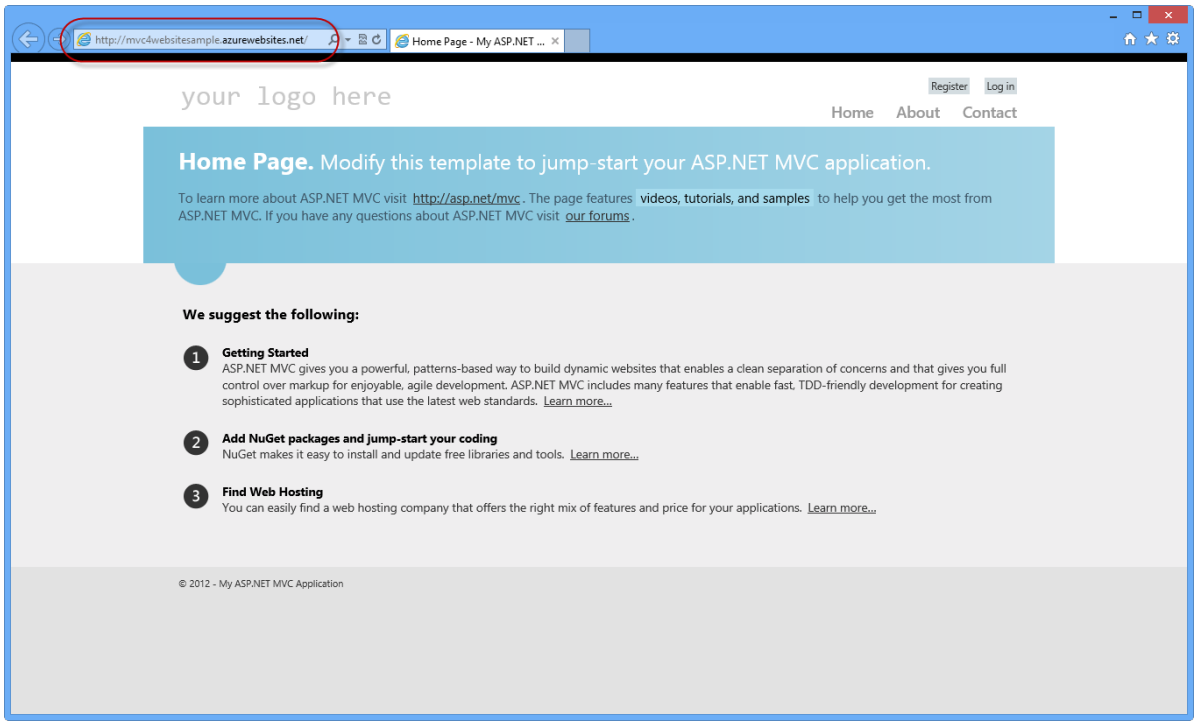
Connection string pointing to SQL Database

8. In the **Preview** page, click **Publish**.



Publishing the web application

9. Once the publishing process finishes, your default browser will open the published web site.



Application published to Windows Azure

What's New in Web Forms in ASP.NET 4.5

2/20/2020 • 40 minutes to read • Edit Online

by Web Camps Team

The new version of ASP.NET Web Forms introduces a number of improvements focused on improving user experience when working with data.

In previous versions of Web Forms, when using data-binding to emit the value of an object member, you used the data-binding expressions Bind() or Eval(). In the new version of ASP.NET, you are able to declare what type of data a control is going to be bound to by using a new ItemType property. Setting this property will enable you to use a strongly-typed variable to receive the full benefits of the Visual Studio development experience, such as IntelliSense, member navigation, and compile-time checking.

With the data-bound controls, you can now also specify your own custom methods for selecting, updating, deleting and inserting data, simplifying the interaction between the page controls and your application logic. Additionally, model binding capabilities have been added to ASP.NET, which means you can map data from the page directly into method type parameters.

Validating user input should also be easier with the latest version of Web Forms. You can now annotate your model classes with validation attributes from the **System.ComponentModel.DataAnnotations** namespace and request that all your site controls validate user input using that information. Client-side validation in Web Forms is now integrated with jQuery, providing cleaner client-side code and unobtrusive JavaScript features.

In the request validation area, improvements have been made to make it easier to selectively turn off request validation for specific parts of your applications or read invalidated request data.

Some improvements have been made to Web Forms server controls to take advantage of new features of HTML5:

- The TextMode property of the TextBox control has been updated to support the new HTML5 input types like email, datetime, and so on.
- The FileUpload control now supports multiple file uploads from browsers that support this HTML5 feature.
- Validator controls now support validating HTML5 input elements.
- New HTML5 elements that have attributes that represent a URL now support runat="server". As a result, you can use ASP.NET conventions in URL paths, like the ~ operator to represent the application root (for example, <video runat="server" src="~/myVideo.wmv"></video>).
- The UpdatePanel control has been fixed to support posting HTML5 input fields.

In the official ASP.NET portal you can find more examples of the new features in ASP.NET WebForms 4.5: What's New in ASP.NET 4.5 and Visual Studio 2012

All sample code and snippets are included in the Web Camps Training Kit, available at https://go.microsoft.com/fwlink/?LinkID=248297&clcid=0x409.

Objectives

In this hands-on lab, you will learn how to:

- Use strongly-typed data-binding expressions
- Use new model binding features in Web Forms
- Use value providers for mapping page data to code-behind methods

- Use Data Annotations for user input validation
- Take advantage of unobtrusive client-side validation with jQuery in Web Forms
- Implement granular request validation
- Implement asynchronous page processing in Web Forms

Prerequisites

You must have the following items to complete this lab:

- Microsoft Visual Studio Express 2012 for Web or superior (read Appendix A for instructions on how to install it).

Setup

Installing Code Snippets

For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. To install the code snippets run **.\Source\Setup\CodeSnippets.vsi** file.

If you are not familiar with the Visual Studio Code Snippets, and want to learn how to use them, you can refer to the appendix from this document "Appendix C: Using Code Snippets".

Exercises

This hands-on lab includes the following exercises:

1. Exercise 1: Model Binding in ASP.NET Web Forms
2. Exercise 2: Data Validation
3. Exercise 3: Asynchronous Page Processing in ASP.NET Web Forms

> **NOTE**
> Each exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

Estimated time to complete this lab: **60 minutes**.

Exercise 1: Model Binding in ASP.NET Web Forms

The new version of ASP.NET Web Forms introduces a number of enhancements focused on improving the experience when working with data. Throughout this exercise, you will learn about strongly typed data-controls and model binding.

Task 1 - Using Strongly-Typed Data-Bindings

In this task, you will discover the new strongly-typed bindings available in ASP.NET 4.5.

1. Open the **Begin** solution located at **Source/Ex1-ModelBinding/Begin/** folder.
   a. You will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.
   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.
   c. Finally, build the solution by clicking **Build | Build Solution**.

> **NOTE**
> One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Open the **Customers.aspx** page. Place an unnumbered list in the main control and include a repeater control inside for listing each customer. Set the repeater name to **customersRepeater** as shown in the following code.

   In previous versions of Web Forms, when using data-binding to emit the value of a member on an object you're data-binding to, you would use a data-binding expression, along with a call to the Eval method, passing in the name of the member as a string.

   At runtime, these calls to Eval will use reflection against the currently bound object to read the value of the member with the given name, and display the result in the HTML. This approach makes it very easy to data-bind against arbitrary, unshaped data.

   Unfortunately, you lose many of the great development-time experience features in Visual Studio, including IntelliSense for member names, support for navigation (like Go To Definition), and compile-time checking.

```
...
<asp:Content ID="Content3" ContentPlaceHolderID="MainContent" runat="server">
  <h3>Customers</h3>
  <ul>
    <asp:Repeater ID="customersRepeater" runat="server">
      <ItemTemplate>
            <li>
                <%# Eval("FirstName") %>
                <%# Eval("LastName") %>
            </li>
      </ItemTemplate>
    </asp:Repeater>
  </ul>
  <a href="CustomerDetails.aspx">Add a New Customer</a>
</asp:Content>
```

3. Open the **Customers.aspx.cs** file.

4. Add the following using statement.

```
using System.Linq;
```

5. In the **Page_Load** method, add code to populate the repeater with the list of customers.

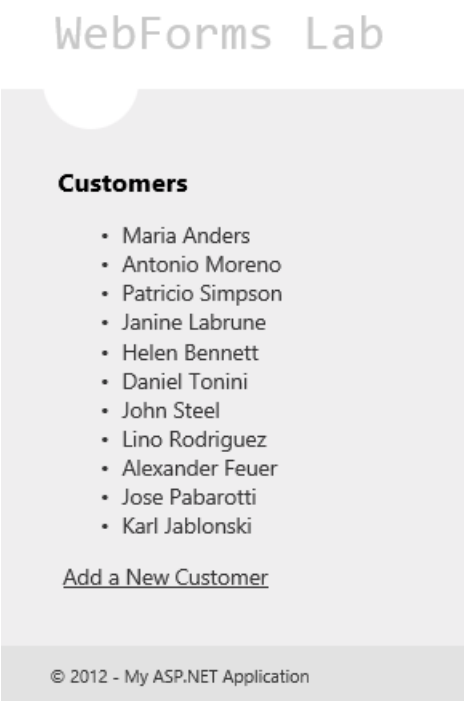
   (Code Snippet - *Web Forms Lab - Ex01 - Bind Customers Data Source*)

```
protected void Page_Load(object sender, EventArgs e)
{
    using (var db = new WebFormsLab.Model.ProductsContext())
    {
        this.customersRepeater.DataSource = db.Customers.ToList();
        this.customersRepeater.DataBind();
    }
}
```

The solution uses EntityFramework together with CodeFirst to create and access the database. In the

following code, the customersRepeater is bound to a materialized query that returns all the customers from the database.

6. Press **F5** to run the solution and go to the **Customers** page to see the repeater in action. As the solution is using CodeFirst, the database will be created and populated in your local SQL Express instance when running the application.



Listing the customers with a repeater

> **NOTE**
>
> In Visual Studio 2012, IIS Express is the default Web development server.

7. Close the browser and go back to Visual Studio.

8. Now replace the implementation to use strongly typed bindings. Open the **Customers.aspx** page and use the new **ItemType** attribute in the repeater to set the **Customer** type as the binding type.

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
  <ul>
    <asp:Repeater ID="customersRepeater"
                  ItemType="WebFormsLab.Model.Customer"
                  runat="server">
      <ItemTemplate>
         ...
      </ItemTemplate>
    </asp:Repeater>
  </ul>
  <a href="CustomerDetails.aspx">Add a New Customer</a>
</asp:Content>
```

The ItemType property enables you to declare which type of data the control is going to be bound to and allows you to use strongly-typed binding inside the data-bound control.

9. Replace the ItemTemplate content with the following code.

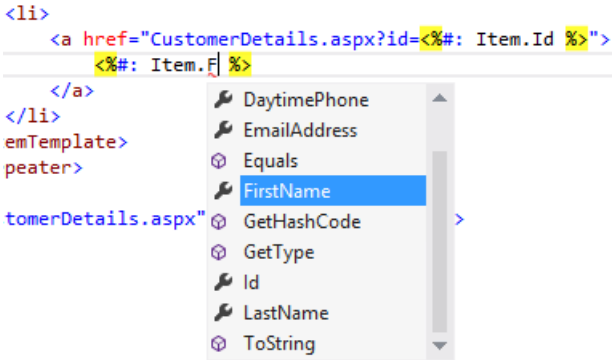
```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
  ...
  <ul>
    <asp:Repeater ID="customersRepeater" ItemType="WebFormsLab.Model.Customer" runat="server">
      <ItemTemplate>
        <li>
          <a href="CustomerDetails.aspx?id=<%#: Item.Id %>">
            <%#: Item.FirstName %> <%#: Item.LastName %>
          </a>
        </li>
      </ItemTemplate>
    </asp:Repeater>
  </ul>
  <a href="CustomerDetails.aspx">Add a New Customer</a>
</asp:Content>
```

One downside with the above approaches is that the calls to Eval() and Bind() are late-bound - meaning you pass strings to represent the property names. This means you don't get Intellisense for the member names, support for code navigation (like Go To Definition), nor compile-time checking support.

Setting the ItemType property causes two new typed variables to be generated in the scope of the data-binding expressions: **Item** and **BindItem**. You can use these strongly typed variables in the data-binding expressions and get the full benefits of the Visual Studio development experience.

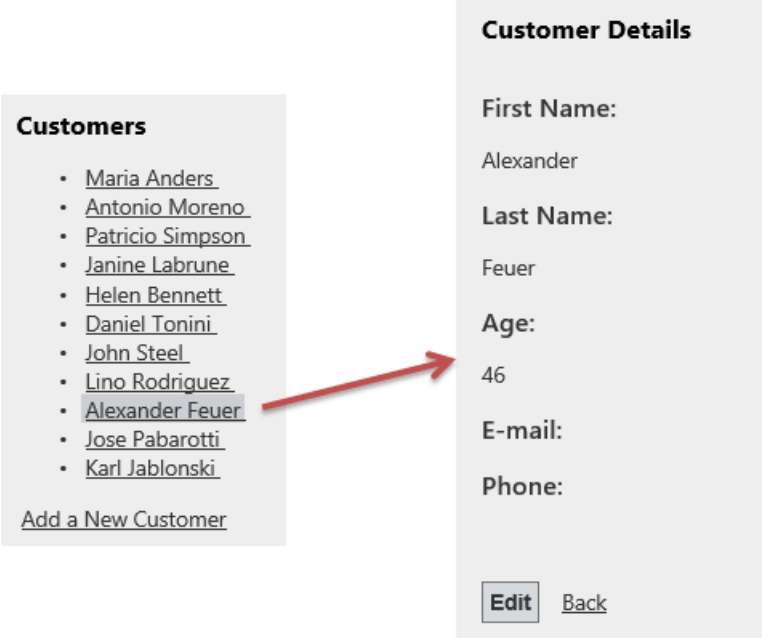
The ": " used in the expression will automatically HTML-encode the output to avoid security issues (for example, cross-site scripting attacks). This notation was available since .NET 4 for response writing, but now is also available in data-binding expressions.

> **NOTE**
>
> The Item member works for one-way binding. If you want to perform two-way binding use the **BindItem** member.



IntelliSense support in strongly-typed binding

10. Press **F5** to run the solution and go to the Customers page to make sure the changes work as expected.



Listing customer details

11. Close the browser and go back to Visual Studio.

Task 2 - Introducing Model Binding in Web Forms

In previous versions of ASP.NET Web Forms, when you wanted to perform two-way data-binding, both retrieving and updating data, you needed to use a Data Source object. This could be an Object Data Source, a SQL Data Source, a LINQ Data Source and so on. However if your scenario required custom code for handling the data, you needed to use the Object Data Source and this brought some drawbacks. For example, you needed to avoid complex types and you needed to handle exceptions when executing validation logic.

In the new version of ASP.NET Web Forms the data-bound controls support model binding. This means that you can specify select, update, insert and delete methods directly in the data-bound control to call logic from your code-behind file or from another class.

To learn about this, you will use a GridView to list the product categories using the new **SelectMethod** attribute. This attribute enables you to specify a method for retrieving the GridView data.

1. Open the **Products.aspx** page and include a **GridView**. Configure the GridView as shown below to use strongly-typed bindings and enable sorting and paging.

```
<asp:Content ID="Content3" ContentPlaceHolderID="MainContent" runat="server">
 <asp:GridView ID="categoriesGrid" runat="server"
    AutoGenerateColumns="false"
    ItemType="WebFormsLab.Model.Category" DataKeyNames="CategoryID">
    <Columns>
      <asp:BoundField DataField="CategoryId" HeaderText="ID" SortExpression="CategoryId" />
      <asp:BoundField DataField="CategoryName" HeaderText="Name" SortExpression="CategoryName" />
      <asp:BoundField DataField="Description" HeaderText="Description" />
      <asp:TemplateField HeaderText="# of Products">
        <ItemTemplate><%#: Item.Products.Count %></ItemTemplate>
      </asp:TemplateField>
    </Columns>
  </asp:GridView>
</asp:Content>
```

2. Use the new **SelectMethod** attribute to configure the GridView to call a **GetCategories** method to select the data.

```
<asp:GridView ID="categoriesGrid" runat="server"
    AutoGenerateColumns="false"
    ItemType="WebFormsLab.Model.Category" DataKeyNames="CategoryId"
    SelectMethod="GetCategories">
  <Columns>
    <asp:BoundField DataField="CategoryId" HeaderText="ID" SortExpression="CategoryId" />
    <asp:BoundField DataField="CategoryName" HeaderText="Name" SortExpression="CategoryName" />
    <asp:BoundField DataField="Description" HeaderText="Description" />
    <asp:TemplateField HeaderText="# of Products">
      <ItemTemplate><%#: Item.Products.Count %></ItemTemplate>
    </asp:TemplateField>
  </Columns>
</asp:GridView>
```

3. Open the **Products.aspx.cs** code-behind file and add the following using statements.

   (Code Snippet - *Web Forms Lab - Ex01 - Namespaces*)

```
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using WebFormsLab.Model;
```

4. Add a private member in the **Products** class and assign a new instance of **ProductsContext**. This property will store the Entity Framework data context that enables you to connect to the database.

```
public partial class Products : System.Web.UI.Page
{
    private ProductsContext db = new ProductsContext();
    ...
```

5. Create a **GetCategories** method to retrieve the list of categories using LINQ. The query will include the **Products** property so the GridView can show the amount of products for each category. Notice that the method returns a raw IQueryable object that represent the query to be executed later on the page lifecycle.

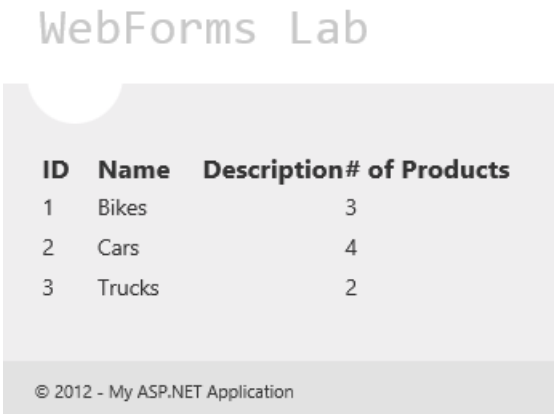
   (Code Snippet - *Web Forms Lab - Ex01 - GetCategories*)

```
public IQueryable<Category> GetCategories()
{
  var query = this.db.Categories
    .Include(c => c.Products);

  return query;
}
```

> **NOTE**
> In previous versions of ASP.NET Web Forms, enabling sorting and paging using your own repository logic within an Object Data Source context, required to write your own custom code and receive all the necessary parameters. Now, as the data-binding methods can return IQueryable and this represents a query still to be executed, ASP.NET can take care of modifying the query to add the proper sorting and paging parameters.

6. Press **F5** to start debugging the site and go to the Products page. You should see that the GridView is populated with the categories returned by the GetCategories method.



Populating a GridView using model binding

7. Press **SHIFT+F5** Stop debugging.

Task 3 - Value Providers in Model Binding

Model binding not only enables you to specify custom methods to work with your data directly in the data-bound control, but also allows you to map data from the page into parameters from these methods. On the method parameter, you can use value provider attributes to specify the value's data source. For example:

- Controls on the page
- Query string values
- View data
- Session state
- Cookies
- Posted form data
- View state
- Custom value providers are supported as well

If you have used ASP.NET MVC 4, you will notice the model binding support is similar. Indeed, these features were taken from ASP.NET MVC and moved into the **System.Web** assembly to be able to use them on Web Forms as well.

In this task, you will update the GridView to filter its results by the amount of products for each category, receiving the filter parameter with model binding.

1. Go back to the **Products.aspx** page.
2. At the top of the GridView, add a **Label** and a **ComboBox** to select the number of products for each category as shown below.

```
<h3>Categories</h3>
<asp:Label ID="Label1" runat="server" AssociatedControlID="minProductsCount">
     Show categories with at least this number of products:
</asp:Label>
<asp:DropDownList runat="server" ID="minProductsCount" AutoPostBack="true">
  <asp:ListItem Value="" Text="-" />
  <asp:ListItem Text="1" />
  <asp:ListItem Text="3" />
  <asp:ListItem Text="5" />
</asp:DropDownList>
<br/>
```

3. Add an **EmptyDataTemplate** to the GridView to show a message when there are no categories with the selected number of products.

```
<asp:GridView ID="categoriesGrid" runat="server"
    AutoGenerateColumns="false"
    ItemType="WebFormsLab.Model.Category" DataKeyNames="CategoryId"
    SelectMethod="GetCategories">
  <Columns>
    <asp:BoundField DataField="CategoryId" HeaderText="ID" />
    <asp:BoundField DataField="CategoryName" HeaderText="Name" />
    <asp:BoundField DataField="Description" HeaderText="Description" />
    <asp:TemplateField HeaderText="# of Products">
      <ItemTemplate><%#: Item.Products.Count %></ItemTemplate>
    </asp:TemplateField>
  </Columns>
  <EmptyDataTemplate>
      No categories found with a product count of <%#: minProductsCount.SelectedValue %>
  </EmptyDataTemplate>
</asp:GridView>
```

4. Open the **Products.aspx.cs** code-behind and add the following using statement.

```
using System.Web.ModelBinding;
```

5. Modify the **GetCategories** method to receive an integer **minProductsCount** argument and filter the returned results. To do this, replace the method with the following code.

   (Code Snippet - *Web Forms Lab - Ex01 - GetCategories 2*)

```
public IQueryable<Category> GetCategories([Control]int? minProductsCount)
{
    var query = this.db.Categories
    .Include(c => c.Products);

    if (minProductsCount.HasValue)
    {
        query = query.Where(c => c.Products.Count >= minProductsCount);
    }

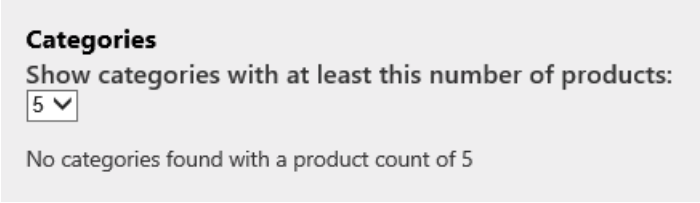
    return query;
}
```

   The new **[Control]** attribute on the **minProductsCount** argument will let ASP.NET know its value must be populated using a control in the page. ASP.NET will look for any control matching the name of the argument (minProductsCount) and perform the necessary mapping and conversion to fill the parameter with the control value.

   Alternatively, the attribute provides an overloaded constructor that enables you to specify the control from where to get the value.

   > **NOTE**
   > One goal of the data-binding features is to reduce the amount of code that needs to be written for page interaction. Apart from the [Control] value provider, you can use other model-binding providers in your method parameters. Some of them are listed in the task introduction.

6. Press **F5** to start debugging the site and go to the Products page. Select a number of products in the drop-down list and notice how the GridView is now updated.



Filtering the GridView with a drop-down list value

7. Stop debugging.

Task 4 - Using Model Binding for Filtering

In this task, you will add a second, child GridView to show the products within the selected category.

1. Open the **Products.aspx** page and update the categories GridView to auto-generate the Select button.

```
<asp:GridView ID="categoriesGrid" runat="server"
  AutoGenerateColumns="false"
  ItemType="WebFormsLab.Model.Category" DataKeyNames="CategoryId"
  SelectMethod="GetCategories"
  AutoGenerateSelectButton="true">
```

2. Add a second **GridView** named **productsGrid** at the bottom. Set the **ItemType** to **WebFormsLab.Model.Product**, the **DataKeyNames** to **ProductId** and the **SelectMethod** to **GetProducts**. Set **AutoGenerateColumns** to **false** and add the columns for ProductId, ProductName, Description and UnitPrice.

```
<h3>Products</h3>
<asp:GridView ID="productsGrid" runat="server"
    CellPadding="4"
    AutoGenerateColumns="false"
    ItemType="WebFormsLab.Model.Product"
    DataKeyNames="ProductId"
    SelectMethod="GetProducts">
    <Columns>
        <asp:BoundField DataField="ProductId" HeaderText="ID" />
        <asp:BoundField DataField="ProductName" HeaderText="Name" />
        <asp:BoundField DataField="Description" HeaderText="Description" HtmlEncode="false" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price" />
    </Columns>
    <EmptyDataTemplate>
        Select a category above to see its products
    </EmptyDataTemplate>
</asp:GridView>
```

3. Open the **Products.aspx.cs** code-behind file. Implement the **GetProducts** method to receive the category ID from the category GridView and filter the products. Model binding will set the parameter value using the selected row in the **categoriesGrid**. Since the argument name and control name do not match, you should specify the name of the control in the Control value provider.

   (Code Snippet - *Web Forms Lab - Ex01 - GetProducts*)

```
public IEnumerable<Product> GetProducts([Control("categoriesGrid")]int? categoryId)
{
    return this.db.Products.Where(p => p.CategoryId == categoryId);
}
```

> **NOTE**
> This approach makes it easier to unit test these methods. On a unit test context, where Web Forms is not executing, the [Control] attribute will not perform any specific action.

4. Open the **Products.aspx** page and locate the products GridView. Update the products GridView to show a link for editing the selected product.

```
<h3>Products</h3>
<asp:GridView ID="productsGrid" runat="server"
  CellPadding="4"
  AutoGenerateColumns="false"
  ItemType="WebFormsLab.Model.Product"
  DataKeyNames="ProductId"
  SelectMethod="GetProducts">
  <Columns>
    <asp:TemplateField>
      <ItemTemplate>
        <a href="ProductDetails.aspx?productId=<%#: Item.ProductId %>">View</a>
      </ItemTemplate>
    </asp:TemplateField>
    <asp:BoundField DataField="ProductId" HeaderText="ID" />
    <asp:BoundField DataField="ProductName" HeaderText="Name" />
    <asp:BoundField DataField="Description" HeaderText="Description" HtmlEncode="false" />
    <asp:BoundField DataField="UnitPrice" HeaderText="Price" />
  </Columns>
  <EmptyDataTemplate>
    Select a category above to see its products
  </EmptyDataTemplate>
</asp:GridView>
```

5. Open the **ProductDetails.aspx** page code-behind and replace the **SelectProduct** method with the following code.

   (Code Snippet - *Web Forms Lab - Ex01 - SelectProduct Method*)

```
public Product SelectProduct([QueryString]int? productId)
{
    return this.db.Products.Find(productId);
}
```

> **NOTE**
> Notice that the **[QueryString]** attribute is used to fill the method parameter from a productId parameter in the query string.

6. Press **F5** to start debugging the site and go to the Products page. Select any category from the categories GridView and notice that the products GridView is updated.

Categories

Show categories with at least this number of products:

	ID	Name	Description	# of Products
Select	1	Bikes		3
Select	2	Cars		4
Select	3	Trucks		2

Products

	ID	Name	Description	Price
View	1	Wooden Bike		350.00
View	2	Quad		10210.00
View	3	Monocycle		210.00

Showing products from the selected category

7. Click the **View** link on a product to open the ProductDetails.aspx page.

   Notice that the page is retrieving the product with the SelectMethod using the productId parameter from the query string.

Product Details

Name:

Wooden Bike

Description (HTML):

Price:

350.00

Edit Back

Viewing the product details

> **NOTE**
>
> The ability to type an HTML description will be implemented in the next exercise.

Task 5 - Using Model Binding for Update Operations

In the previous task, you have used model binding mainly for selecting data, in this task you will learn how to use model binding in update operations.

You will update the categories GridView to let the user update categories.

1. Open the **Products.aspx** page and update the categories GridView to auto-generate the Edit button and use the new **UpdateMethod** attribute to specify an **UpdateCategory** method to update the selected item.

```
<asp:GridView ID="categoriesGrid" runat="server"
    AutoGenerateColumns="false"
    ItemType="WebFormsLab.Model.Category" DataKeyNames="CategoryId"
    SelectMethod="GetCategories"
    AutoGenerateSelectButton="true"
    AutoGenerateEditButton="true"
    UpdateMethod="UpdateCategory">
```

The DataKeyNames attribute in the GridView define which are the members that uniquely identify the model-bound object and therefore, which are the parameters the update method should at least receive.

2. Open the **Products.aspx.cs** code-behind file and implement the **UpdateCategory** method. The method should receive the category ID to load the current category, populate the values from the GridView and then update the category.

   (Code Snippet - *Web Forms Lab - Ex01 - UpdateCategory*)

```
public void UpdateCategory(int categoryId)
{
    var category = this.db.Categories.Find(categoryId);

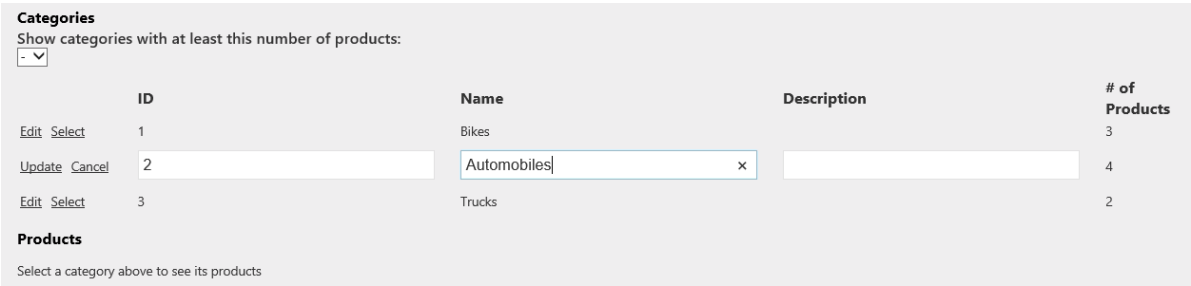
    this.TryUpdateModel(category);

    if (this.ModelState.IsValid)
    {
        this.db.SaveChanges();
    }
}
```

   The new **TryUpdateModel** method in the Page class is responsible of populating the model object using the values from the controls in the page. In this case, it will replace the updated values from the current GridView row being edited into the **category** object.

> **NOTE**
>
> The next exercise will explain the usage of the ModelState.IsValid for validating the data entered by the user when editing the object.

3. Run the site and go to the Products page. Edit a category. Type a new name and then click **Update** to persist the changes.



Editing categories

Exercise 2: Data Validation

In this exercise, you will learn about the new data validation features in ASP.NET 4.5. You will check out the new unobtrusive validation features in Web Forms. You will use data annotations in the application model classes for user input validation, and finally, you will learn how to turn on or off request validation to individual controls in a page.

Task 1 - Unobtrusive Validation

Forms with complex data including validators tend to generate too much JavaScript code in the page, which can represent about 60% of the code. With unobtrusive validation enabled, your HTML code will look cleaner and tidier.

In this section, you will enable unobtrusive validation in ASP.NET to compare the HTML code generated by both configurations.

1. Open **Visual Studio 2012** and open the **Begin** solution located in the **Source\Ex2-Validation\Begin** folder of this lab. Alternatively, you can continue working on your existing solution from the previous exercise.

   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, in the Solution Explorer, click the **WebFormsLab** project **Manage NuGet Packages**.

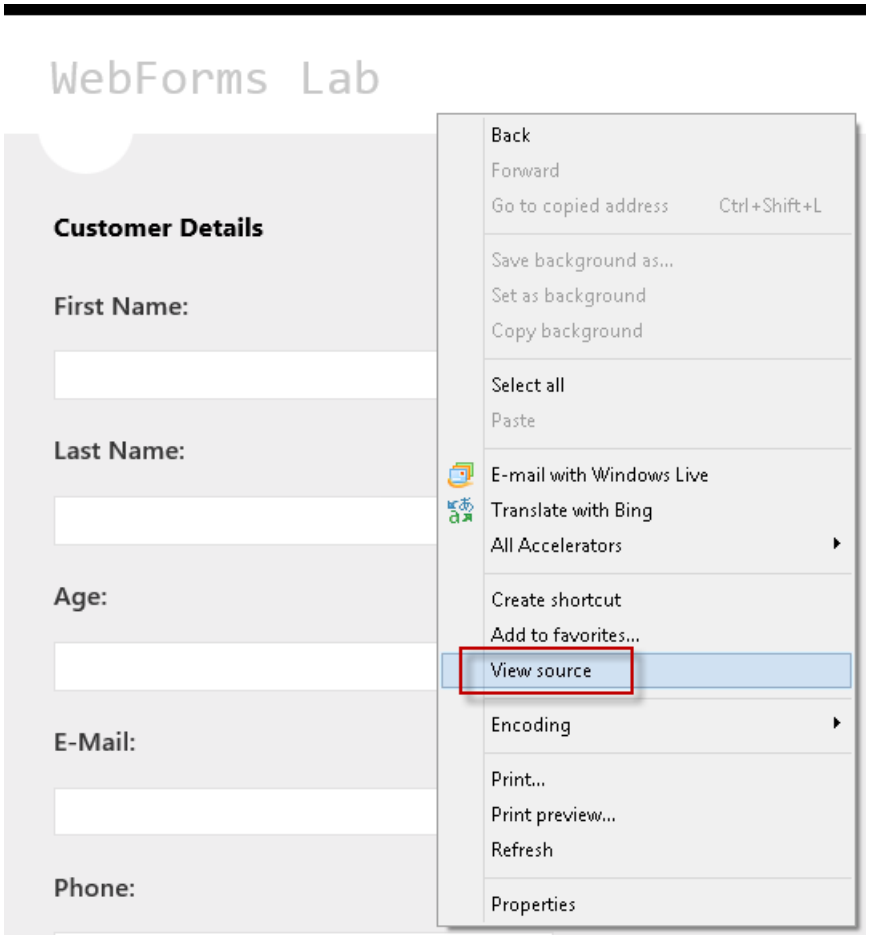
   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Press **F5** to start the web application. Go to the Customers page and click the **Add a New Customer** link.

3. Right-click on the browser page, and select **View Source** option to open the HTML code generated by the application.



Showing the page HTML code

4. Scroll through the page source code and notice that ASP.NET has injected JavaScript code and data validators in the page to perform the validations and show the error list.

Validation JavaScript code in CustomerDetails page

5. Close the browser and go back to Visual Studio.

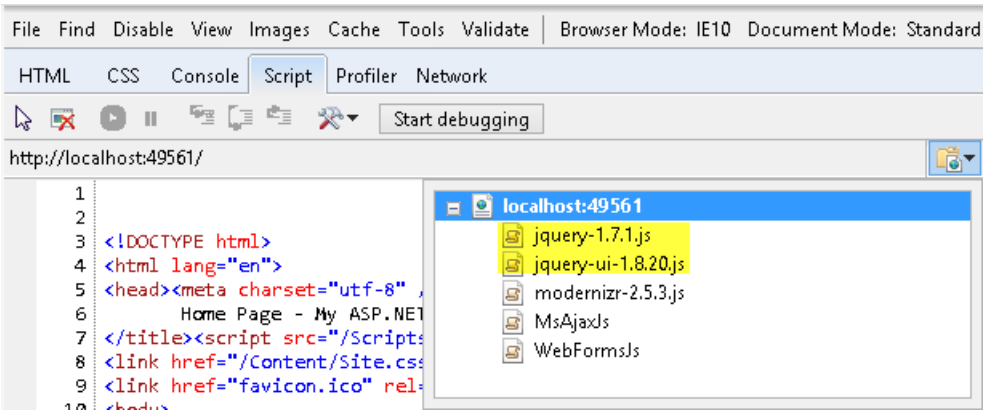
6. Now you will enable unobtrusive validation. Open **Web.Config** and locate **ValidationSettings:UnobtrusiveValidationMode** key in the **AppSettings** section . Set the key value to **WebForms**.

```
<configuration>
  ...
  <appSettings>
    <add key="aspnet:uselegacysynchronizationcontext" value="false" />
    <add key="ValidationSettings:UnobtrusiveValidationMode" value="WebForms"/>
```

> **NOTE**
>
> You can also set this property in the "**Page_Load**" event in case you want to enable Unobtrusive Validation only for some pages.

7. Open **CustomerDetails.aspx** and press **F5** to start the Web application.

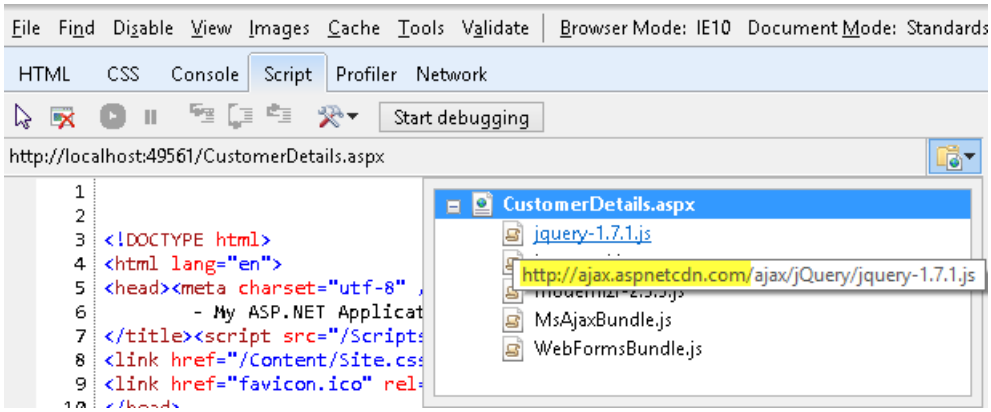
8. Press the F12 key to open the IE developer tools. Once the developer tools is open, select the script tab. Select **CustomerDetails.aspx** from the menu and take note that the scripts required to run jQuery on the page have been loaded into the browser from the local site.



Loading the jQuery JavaScript files directly from the local IIS server

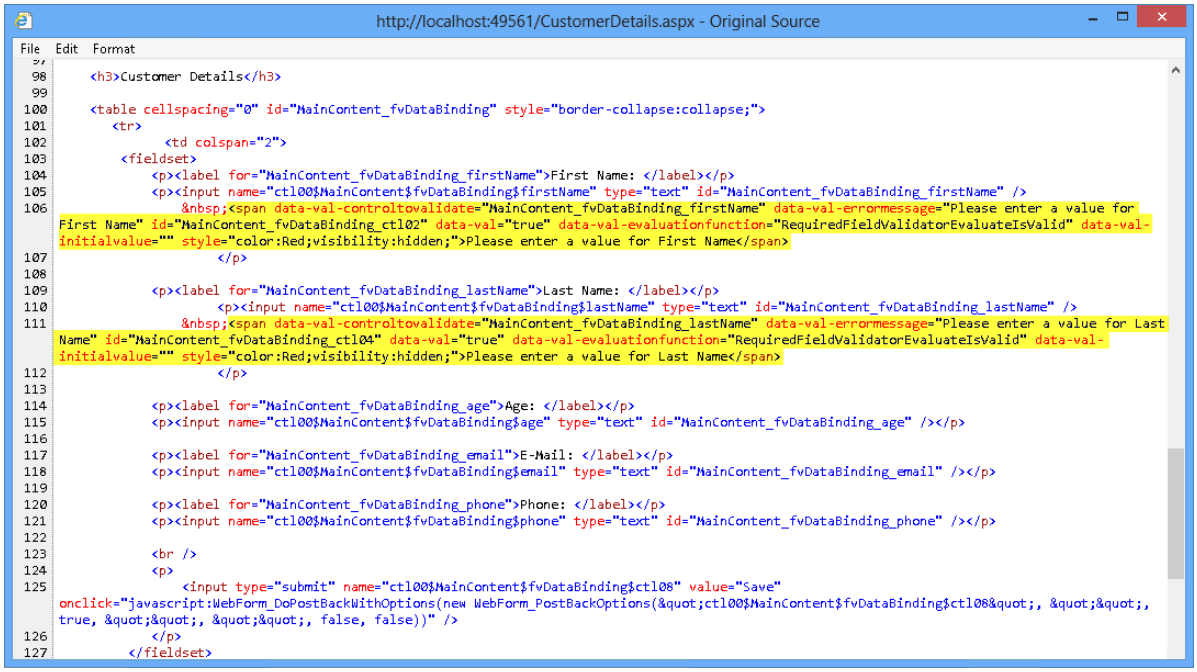
9. Close the browser to return to Visual Studio. Open the **Site.Master** file again and locate the **ScriptManager**. Add the attribute **EnableCdn** property with the value **True**. This will force jQuery to be loaded from the online URL, not from the local site's URL.

10. Open **CustomerDetails.aspx** in Visual Studio. Press the F5 key to run the site. Once Internet Explorer opens, press the F12 key to open the developer tools. Select the **Script** tab, and then take a look at the drop-down list. Note the jQuery JavaScript files are no longer being loaded from the local site, but rather from the online jQuery CDN.



Loading the jQuery JavaScript files from the CDN

11. Open the HTML page source code again using the View source option in the browser. Notice that by enabling the unobtrusive validation ASP.NET has replaced the injected JavaScript code with data-*attributes.



Unobtrusive validation code

> **NOTE**
>
> In this example, you saw how a validation summary with Data annotations was simplified to only a few HTML and JavaScript lines. Previously, without unobtrusive validation, the more validation controls you add, the bigger your JavaScript validation code will grow.

Task 2 - Validating the Model with Data Annotations

ASP.NET 4.5 introduces data annotations validation for Web Forms. Instead of having a validation control on each input, you can now define constraints in your model classes and use them across all your web application. In this section, you will learn how to use data annotations for validating a new/edit customer form.

1. Open **CustomerDetail.aspx** page. Notice that the customer first name and second name in the

EditItemTemplate and **InsertItemTemplate** sections are validated using a RequiredFieldValidator controls. Each validator is associated to a particular condition, so you need to include as many validators as conditions to check.

2. Add data annotations to validate the Customer model class. Open **Customer.cs** class in the **Model** folder and *decorate* each property using data annotation attributes.

   (Code Snippet - *Web Forms Lab - Ex02 - Data Annotations*)

```
namespace WebFormsLab.Model
{
  using System.Collections.Generic;
  using System.ComponentModel.DataAnnotations;

  public class Customer
  {
     [Key]
     public int Id { get; set; }

     [Required]
     public string FirstName { get; set; }

     [Required]
     public string LastName { get; set; }

     [Range(0, 130)]
     public int Age { get; set; }

     public Address Address { get; set; }

     [Phone]
     public string DaytimePhone { get; set; }

     [EmailAddress, StringLength(256)]
     public string EmailAddress { get; set; }
  }
}
```

> **NOTE**
>
> .NET Framework 4.5 has extended the existing data annotation collection. These are some of the data annotations you can use: [CreditCard], [Phone], [EmailAddress], [Range], [Compare], [Url], [FileExtensions], [Required], [Key], [RegularExpression].
>
> Some usage examples:
>
> [Key]: Specifies that an attribute is the unique identifier
>
> [Range(0.4, 0.5, ErrorMessage="{Write an error message}"]: Double range
>
> [EmailAddress(ErrorMessage="Invalid Email"), MaxLength(56)]: Two annotations in the same line.
>
> You can also define your own error messages within each attribute.

3. Open **CustomerDetails.aspx** and remove all the RequiredFieldValidators for the first and last name fields in the in EditItemTemplate and InsertItemTemplate sections of the FormView control.

```
<EditItemTemplate>
  <fieldset>
     <p><asp:Label runat="server" AssociatedControlID="firstName">First Name: </asp:Label></p>
     <p><asp:TextBox runat="server" ID="firstName" Text='<%#: BindItem.FirstName %>' />
         <asp:RequiredFieldValidator runat="server" ControlToValidate="firstName"
ErrorMessage="Please enter a value for First Name" ForeColor="Red" />
    </p>

     <p><asp:Label runat="server" AssociatedControlID="lastName">Last Name: </asp:Label></p>
     <p><asp:TextBox runat="server" ID="lastName" Text='<%#: BindItem.LastName %>' />
           <asp:RequiredFieldValidator runat="server" ControlToValidate="lastName"
ErrorMessage="Please enter a value for Last Name" ForeColor="Red" />
    </p>
  ...
<InsertItemTemplate>
 <fieldset>
   <p><asp:Label runat="server" AssociatedControlID="firstName">First Name: </asp:Label></p>
   <p><asp:TextBox runat="server" ID="firstName" Text='<%#: BindItem.FirstName %>' />
      <asp:RequiredFieldValidator runat="server" ControlToValidate="firstName"
ErrorMessage="Please enter a value for First Name" ForeColor="Red" />
    </p>

   <p><asp:Label runat="server" AssociatedControlID="lastName">Last Name: </asp:Label></p>
    <p><asp:TextBox runat="server" ID="lastName" Text='<%#: BindItem.LastName %>' />
      <asp:RequiredFieldValidator runat="server" ControlToValidate="lastName" ErrorMessage="Please
enter a value for Last Name" ForeColor="Red" />
    </p>
  ...
```

NOTE

One advantage of using data annotations is that validation logic is not duplicated in your application pages. You define it once in the model, and use it across all the application pages that manipulate data.

4. Open **CustomerDetails.aspx** code-behind and locate the SaveCustomer method. This method is called when inserting a new customer and receives the Customer parameter from the FormView control values. When the mapping between the page controls and the parameter object occurs, ASP.NET will execute the model validation against all the data annotation attributes and fill the ModelState dictionary with the errors encountered, if any.

   The ModelState.IsValid will only return true if all the fields on your model are valid after performing the validation.

```
public void SaveCustomer(Customer customer)
{
    if (this.ModelState.IsValid)
    {
        using (var db = new ProductsContext())
        {
            ...
```

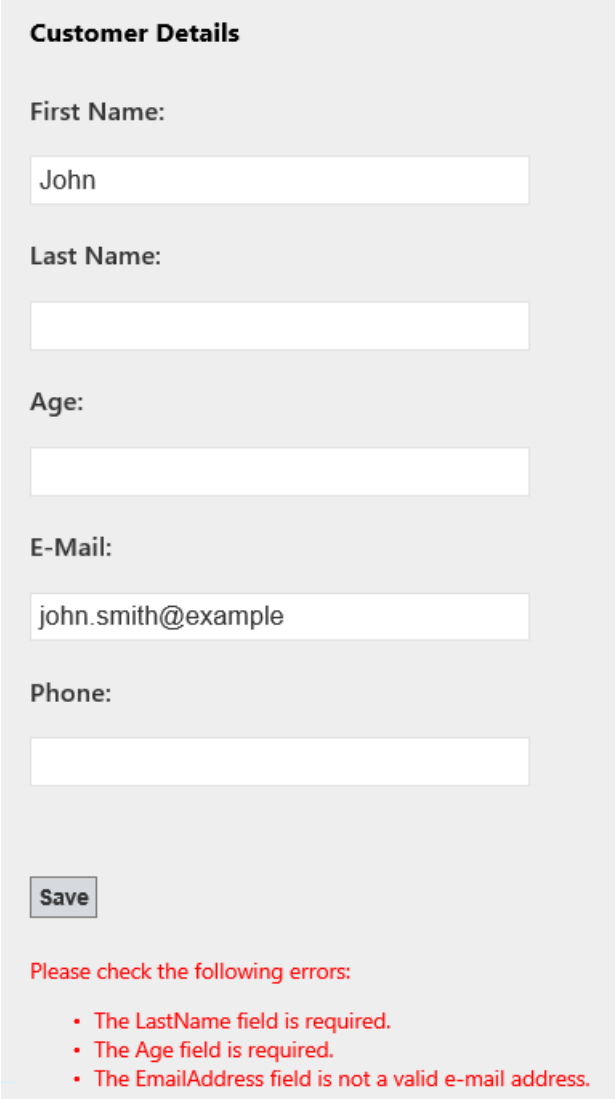
5. Add a **ValidationSummary** control at the end of the CustomerDetails page to show the list of model errors.

```
</fieldset>
    </InsertItemTemplate>
  </asp:FormView>

  <asp:ValidationSummary runat="server" ShowModelStateErrors="true"
      ForeColor="Red" HeaderText="Please check the following errors:"/>
</asp:Content>
```

The **ShowModelStateErrors** is a new property on the ValidationSummary control that when set to **true**, the control will show the errors from the ModelState dictionary. These errors come from the data annotations validation.

6. Press **F5** to run the Web application. Complete the form with some erroneous values and click **Save** to execute validation. Notice the error summary at the bottom.



Validation with Data Annotations

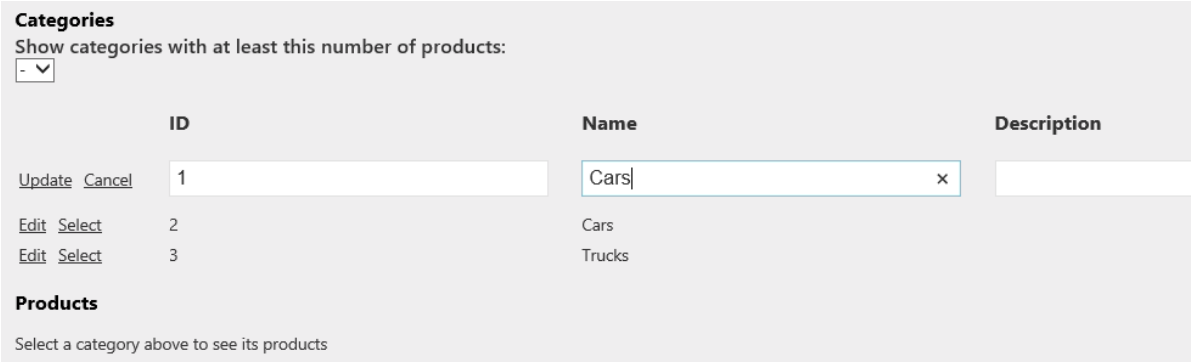
Task 3 - Handling Custom Database Errors with ModelState

In previous version of Web Forms, handling database errors such as a too long string or a unique key violation could involve throwing exceptions in your repository code and then handling the exceptions on your code-behind to display an error. A great amount of code is required to do something relatively simple.

In Web Forms 4.5, the ModelState object can be used to display the errors on the page, either from your model or from the database, in a consistent manner.

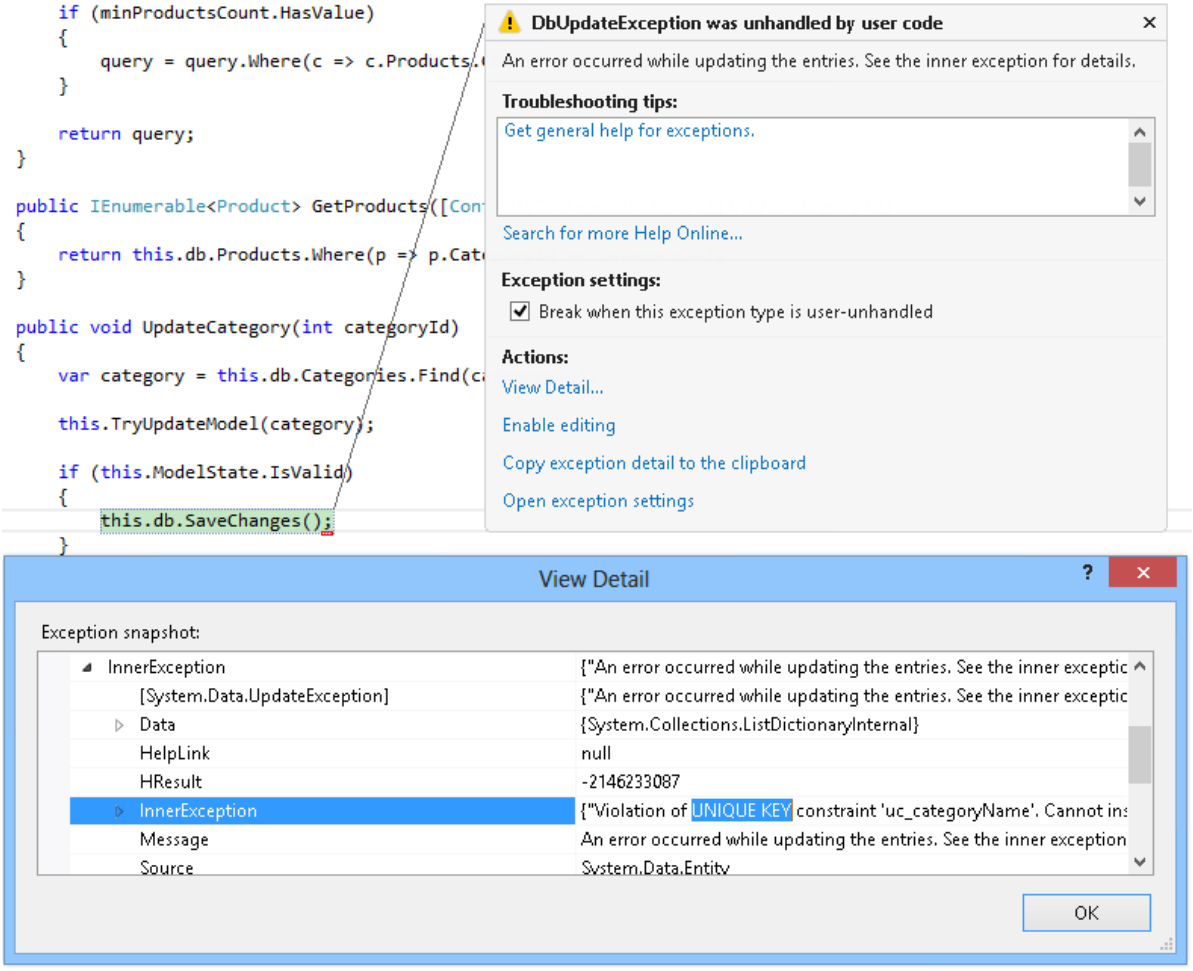
In this task, you will add code to properly handle database exceptions and show the appropriate message to the user using the ModelState object.

1. While the application is still running, try to update the name of a category using a duplicated value.



Updating a category with a duplicated name

Notice that an exception is thrown due to the "unique" constraint of the **CategoryName** column.



Exception for duplicated category names

2. Stop debugging. In the **Products.aspx.cs** code-behind file, update the **UpdateCategory** method to handle the exceptions thrown by the db.SaveChanges() method call and add an error to the **ModelState** object.

   The new **TryUpdateModel** method updates the category object retrieved from the database using the form data provided by the user.

   (Code Snippet - *Web Forms Lab - Ex02 - UpdateCategory Handle Errors*)

```
public void UpdateCategory(int categoryId)
{
  var category = this.db.Categories.Find(categoryId);

  this.TryUpdateModel(category);

  if (this.ModelState.IsValid)
  {
    try
    {
      this.db.SaveChanges();
    }
    catch (DbUpdateException)
    {
      var message = string.Format("A category with the name {0} already exists.",
category.CategoryName);
      this.ModelState.AddModelError("CategoryName", message);
    }
  }
}
```

> **NOTE**
> Ideally, you would have to identify the cause of the DbUpdateException and check if the root cause is the violation of a unique key constraint.

3. Open **Products.aspx** and add a **ValidationSummary** control below the categories GridView to show the list of model errors.

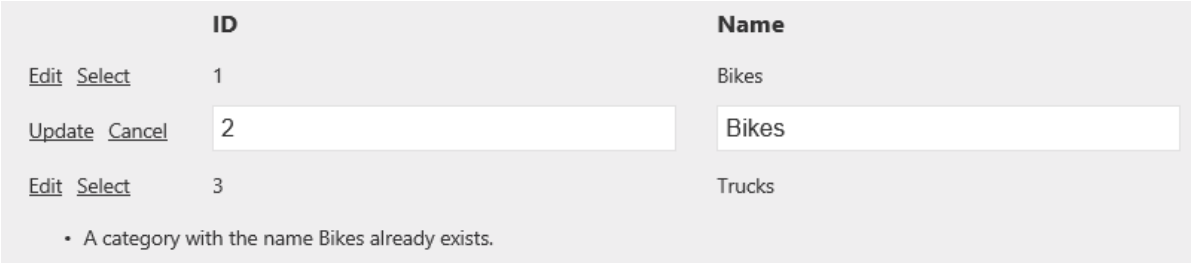
```
<asp:GridView ID="categoriesGrid" runat="server"
  ...
</asp:GridView>

<asp:ValidationSummary ID="ValidationSummary1" runat="server" ShowModelStateErrors="true" />

<h3>Products</h3>
```

4. Run the site and go to the Products page. Try to update the name of a category using an duplicated value.

   Notice that the exception was handled and the error message appears in the **ValidationSummary** control.



Duplicated category error

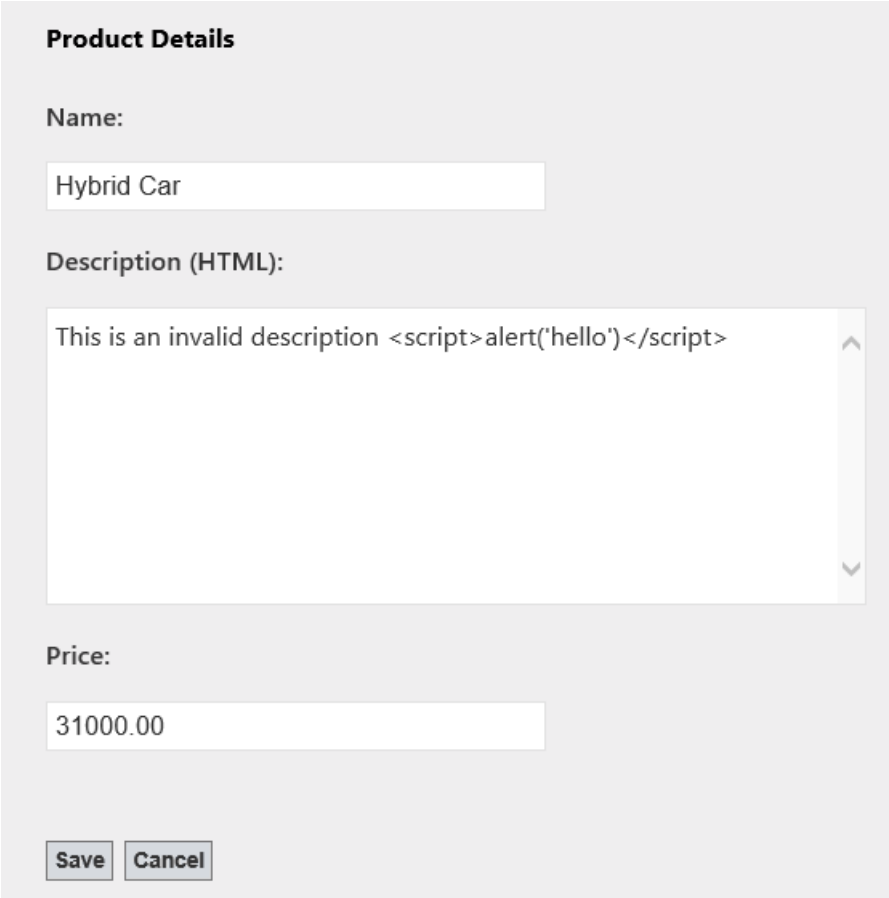
Task 4 - Request Validation in ASP.NET Web Forms 4.5

The request validation feature in ASP.NET provides a certain level of default protection against cross-site scripting (XSS) attacks. In previous versions of ASP.NET, request validation was enabled by default and could only be disabled for an entire page. With the new version of ASP.NET Web Forms you can now disable the request validation for a single control, perform lazy request validation or access un-validated request data (be careful if you do so!).

1. Press **Ctrl+F5** to start the site without debugging and go to the Products page. Select a category and then

click the **Edit** link on any of the products.

2. Type a description containing potentially dangerous content, for instance including HTML tags. Take notice of the exception thrown due to the request validation.



Editing a product with potentially dangerous content

Exception thrown due to request validation

3. Close the page and, in Visual Studio, press **SHIFT+F5** to stop debugging.

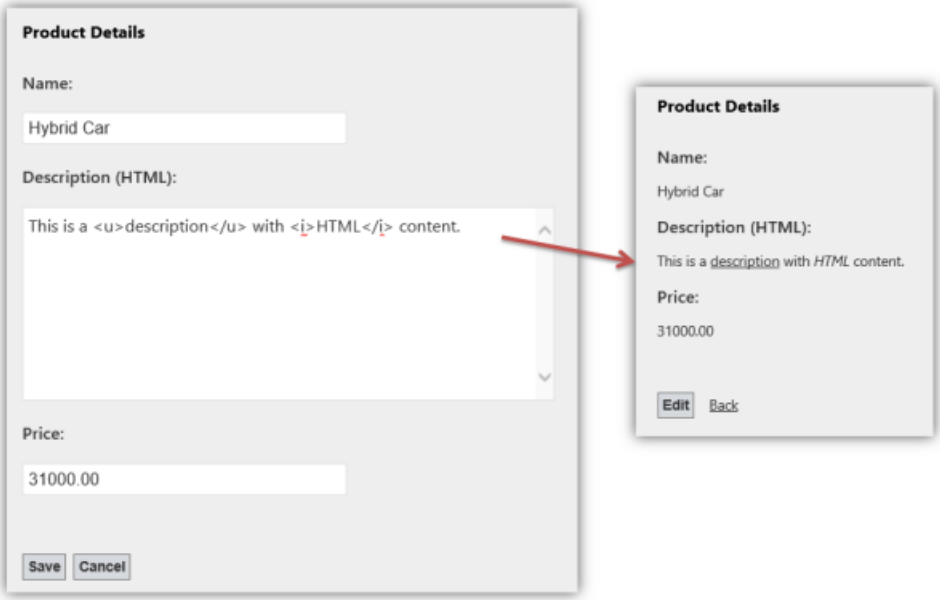
4. Open the **ProductDetails.aspx** page and locate the **Description** TextBox.

5. Add the new **ValidateRequestMode** property to the TextBox and set its value to **Disabled**.

   The new **ValidateRequestMode** attribute allows you to disable the request validation granularly on each control. This is useful when you want to use an input that may receive HTML code, but want to keep the validation working for the rest of the page.

```
<p>
  <asp:TextBox runat="server" ID="Description" TextMode="MultiLine"
            Cols="60" Rows="8" Text='<%# BindItem.Description %>'
    ValidateRequestMode="Disabled" />
</p>
```

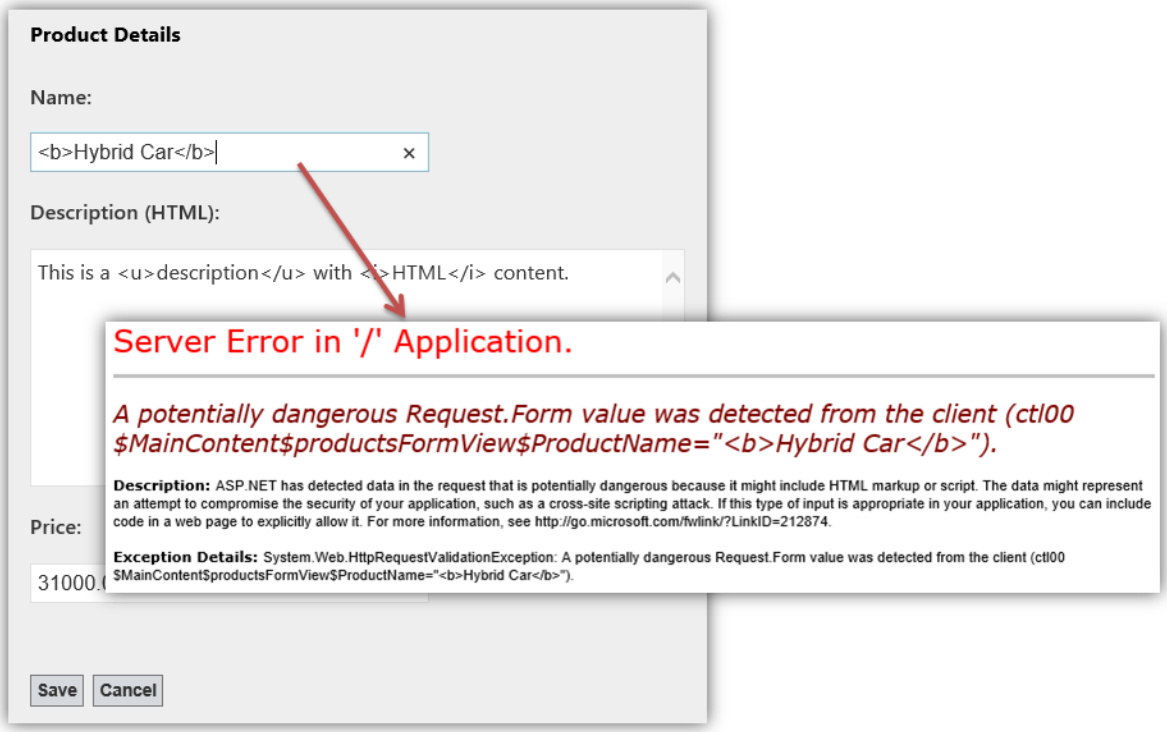
6. Press **F5** to run the web application. Open the edit product page again and complete a product description including HTML tags. Notice that you can now add HTML content to the description.



Request validation disabled for the product description

> **NOTE**
>
> In a production application, you should sanitize the HTML code entered by the user to make sure only safe HTML tags are entered (for example, there are no <script> tags). To do this, you can use Microsoft Web Protection Library.

7. Edit the product again. Type HTML code in the Name field and click **Save**. Notice that Request Validation is only disabled for the Description field and the rest of the fields re still validated against the potentially dangerous content.



Request validation enabled in the rest of the fields

ASP.NET Web Forms 4.5 includes a new request validation mode to perform request validation lazily. With

the request validation mode set to **4.5**, if a piece of code accesses *Request.Form["key"]*, ASP.NET 4.5's request validation will only trigger request validation for that specific element in the form collection.

Additionally, ASP.NET 4.5 now includes core encoding routines from the Microsoft Anti-XSS Library v4.0. The Anti-XSS encoding routines are implemented by the new *AntiXssEncoder* type found in the new **System.Web.Security.AntiXss** namespace. With the **encoderType** parameter configured to use *AntiXssEncoder*, all output encoding within ASP.NET automatically uses the new encoding routines.

8. ASP.NET 4.5 request validation also supports un-validated access to request data. ASP.NET 4.5 adds a new collection property to the **HttpRequest** object called **Unvalidated**. When you navigate into **HttpRequest.Unvalidated** you have access to all of the common pieces of request data, including Forms, QueryStrings, Cookies, URLs, and so on.

```
protected void Page_Load(object sender, EventArgs e)
{
    Request.Unvalidated.
}

public IQueryable<Catego                    ntrol]i
{
    var query = this.db.
      .Include(c => c.Pr

    if (minProductsCount
    {
        query = query.Wh                    ount >=
    }
```

Cookies
Equals
Files
Form
GetHashCode
GetType
Headers
Path
PathInfo

Request.Unvalidated object

> **NOTE**
>
> **Please use the HttpRequest.Unvalidated property with caution!** Make sure you carefully perform custom validation on the raw request data to ensure that dangerous text is not round-tripped and rendered back to unsuspecting customers!

Exercise 3: Asynchronous Page Processing in ASP.NET Web Forms

In this exercise, you will be introduced to the new asynchronous page processing features in ASP.NET Web Forms.

Task 1 - Updating the Product Details Page to Upload and Show Images

In this task, you will update the product details page to allow the user to specify an image URL for the product and display it in the read-only view. You will create a local copy of the specified image by downloading it synchronously. In the next task, you will update this implementation to make it work asynchronously.

1. Open **Visual Studio 2012** and load the **Begin** solution located in **Source\Ex3-Async\Begin** from this lab's folder. Alternatively, you can continue working on your existing solution from the previous exercises.

   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, in the Solution Explorer, click the **WebFormsLab** project and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

> **NOTE**
>
> One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Open the **ProductDetails.aspx** page source and add a field in the FormView's ItemTemplate to show the product image.

```
<ItemTemplate>
     <fieldset>
          <p><b><asp:Label ID="Label2" runat="server" AssociatedControlID="itemProductName">Name:
</asp:Label></b></p>
          <p><asp:Label runat="server" ID="itemProductName" Text='<%#: Item.ProductName %>' /></p>
          <p><b><asp:Label ID="Label3" runat="server" AssociatedControlID="itemDescription">Description
(HTML):</asp:Label></b></p>
          <p><asp:Label runat="server" ID="itemDescription" Text='<%# Item.Description %>' /></p>
          <p><b><asp:Label ID="Label4" runat="server" AssociatedControlID="itemUnitPrice">Price:
</asp:Label></b></p>
          <p><asp:Label runat="server" ID="itemUnitPrice" Text='<%#: Item.UnitPrice %>' /></p>

          <p><b><asp:Label ID="Label5" runat="server" AssociatedControlID="itemUnitPrice">Image:
</asp:Label></b></p>
          <p>
                <img src="<%# string.IsNullOrEmpty(Item.ImagePath) ? "/Images/noimage.jpg" :
                Item.ImagePath %>" alt="Image" />
          </p>

          <br />
          <p>
                <asp:Button ID="Button1" runat="server" CommandName="Edit" Text="Edit" /> 
                <asp:HyperLink NavigateUrl="~/Products.aspx" Text="Back" runat="server" />
          </p>
     </fieldset>
</ItemTemplate>
```

3. Add a field to specify the image URL in the FormView's EditTemplate.

```
<fieldset>
     <p><asp:Label ID="Label2" runat="server" AssociatedControlID="ProductName">Name:</asp:Label></p>
     <p><asp:TextBox runat="server" ID="ProductName" Text='<%#: BindItem.ProductName %>' /></p>
     <p><asp:Label ID="Label3" runat="server" AssociatedControlID="Description">Description (HTML):
</asp:Label></p>
     <p>
          <asp:TextBox runat="server" ID="Description" TextMode="MultiLine" Cols="60" Rows="8"
Text='<%# BindItem.Description %>'
                ValidateRequestMode="Disabled" />
     </p>
     <p><asp:Label ID="Label4" runat="server" AssociatedControlID="UnitPrice">Price:</asp:Label></p>
     <p><asp:TextBox runat="server" ID="UnitPrice" Text='<%#: BindItem.UnitPrice %>' /></p>

     <p><asp:Label ID="Label1" runat="server" AssociatedControlID="ImagePath">Image URL:</asp:Label>
</p>
     <p><asp:TextBox runat="server" ID="ImagePath" Text='<%#:  BindItem.ImagePath %>' /></p>

     <br />
     <p>
          <asp:Button runat="server" CommandName="Update" Text="Save" />
          <asp:Button runat="server" CommandName="Cancel" Text="Cancel" CausesValidation="false" />
     </p>
</fieldset>
```

4. Open the **ProductDetails.aspx.cs** code-behind file and add the following namespace directives.

   (Code Snippet - *Web Forms Lab - Ex03 - Namespaces*)

```
using System.IO;
using System.Net;
using System.Web;
```

5. Create an **UpdateProductImage** method to store remote images in the local **Images** folder and update the product entity with the new image location value.

   (Code Snippet - *Web Forms Lab - Ex03 - UpdateProductImage*)

```
private void UpdateProductImage(Product product)
{
    string imageUrl = product.ImagePath;

    if (!string.IsNullOrEmpty(imageUrl) && !VirtualPathUtility.IsAbsolute(imageUrl))
    {
        product.ImagePath = string.Format(
                                "/Images/{0}{1}",
                                product.ProductId,
                                Path.GetExtension(imageUrl));

        using (var wc = new WebClient())
        {
            wc.DownloadFile(imageUrl, Server.MapPath(product.ImagePath));
        }
    }
}
```

6. Update the **UpdateProduct** method to call the **UpdateProductImage** method.

   (Code Snippet - *Web Forms Lab - Ex03 - UpdateProductImage Call*)

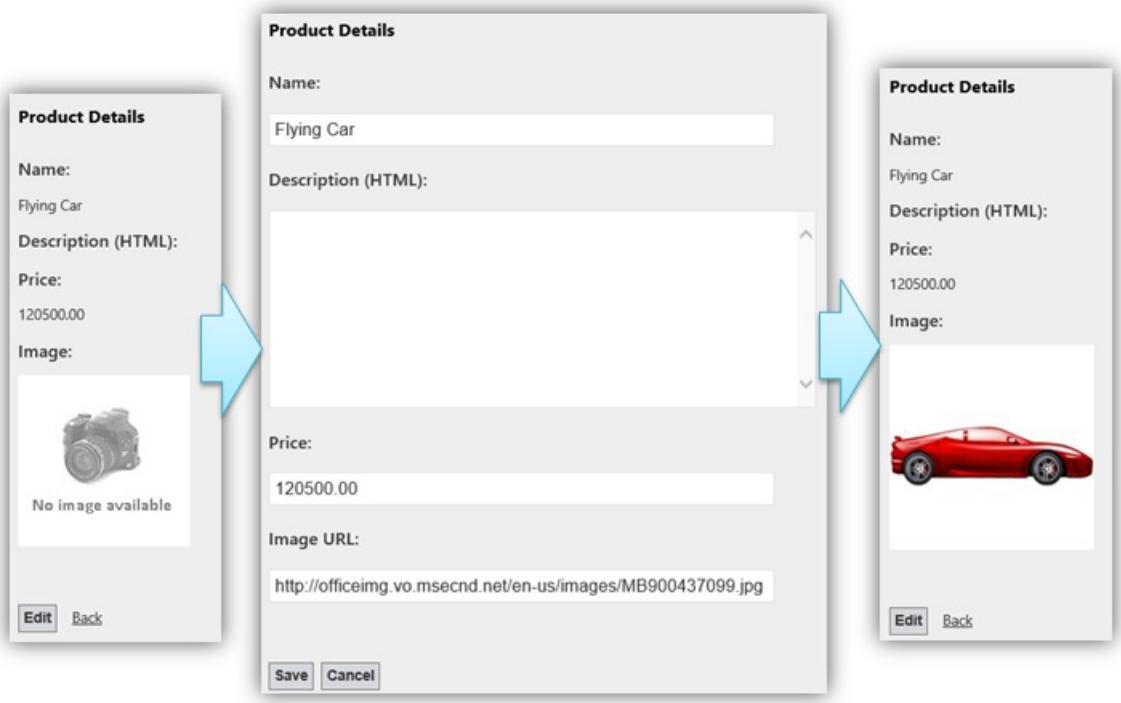
```
public void UpdateProduct(int productId)
{
    var product = this.db.Products.Find(productId);

    this.TryUpdateModel(product);

    this.UpdateProductImage(product);

    if (this.ModelState.IsValid)
    {
        this.db.SaveChanges();
    }
}
```

7. Run the application and try to upload an image for a product. For example, you can use the following image URL from Office Clip Arts: [http://officeimg.vo.msecnd.net/images/MB900437099.jpg] (http://officeimg.vo.msecnd.net/images/MB900437099.jpg)



Setting an image for a product

Task 2 - Adding Asynchronous Processing to the Product Details Page

In this task, you will update the product details page to make it work asynchronously. You will enhance a long running task - the image download process - by using ASP.NET 4.5 asynchronous page processing.

Asynchronous methods in web applications can be used to optimize the way ASP.NET thread pools are used. In ASP.NET there are a limited number of threads in the thread pool for attending requests, thus, when all the threads are busy, ASP.NET starts to reject new requests, sends application error messages and makes your site unavailable.

Time-consuming operations on your web site are great candidates for asynchronous programming because they occupy the assigned thread for a long time. This includes long running requests, pages with lots of different elements and pages that require offline operations, such querying a database or accessing an external web server. The advantage is that if you use asynchronous methods for these operations, while the page is processing, the thread is freed and returned to the thread pool and can be used to attend to a new page request. This means, the page will start processing in one thread from the thread pool and might complete processing in a different one, after the async processing completes.

1. Open the **ProductDetails.aspx** page. Add the **Async** attribute in the **Page** element and set it to **true**. This

attribute tells ASP.NET to implement the IHttpAsyncHandler interface.

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
    CodeBehind="ProductDetails.aspx.cs" Inherits="WebFormsLab.ProductDetails"
    Async="true" %>
```

2. Add a Label at the bottom of the page to show the details of the threads running the page.

```
<EmptyDataTemplate>Product not found</EmptyDataTemplate>
  </asp:FormView>

  <asp:Label ID="threadsMessageLabel" runat="server" />
</asp:Content>
```

3. Open up **ProductDetails.aspx.cs** and add the following namespace directives.

   (Code Snippet - *Web Forms Lab - Ex03 - Namespaces 2*)

```
using System.Web.UI;
using System.Threading;
```

4. Modify the **UpdateProductImage** method to download the image with an asynchronous task. You will replace the **WebClient DownloadFile** method with the **DownloadFileTaskAsync** method and include the **await** keyword.

   (Code Snippet - *Web Forms Lab - Ex03 - UpdateProductImage Async*)

```
private void UpdateProductImage(Product product)
{
    string imageUrl = product.ImagePath;

    if (!string.IsNullOrEmpty(imageUrl) && !VirtualPathUtility.IsAbsolute(imageUrl))
    {
        product.ImagePath = string.Format(
            "/Images/{0}{1}",
            product.ProductId,
            Path.GetExtension(imageUrl));

        this.RegisterAsyncTask(new PageAsyncTask(async (t) =>
        {
            using (var wc = new WebClient())
            {
                await wc.DownloadFileTaskAsync(imageUrl, this.Server.MapPath(product.ImagePath));
            }
        }));
    }
}
```

The RegisterAsyncTask registers a new page asynchronous task to be executed in a different thread. It receives a lambda expression with the Task (t) to be executed. The **await** keyword in the **DownloadFileTaskAsync** method converts the remainder of the method into a callback that is invoked asynchronously after the **DownloadFileTaskAsync** method has completed. ASP.NET will resume the execution of the method by automatically maintaining all the HTTP request original values. The new asynchronous programming model in .NET 4.5 enables you to write asynchronous code that looks very much like synchronous code, and let the compiler handle the complications of callback functions or continuation code.

> **NOTE**
>
> RegisterAsyncTask and PageAsyncTask were already available since .NET 2.0. The await keyword is new from the .NET 4.5 asynchronous programming model and can be used together with the new TaskAsync methods from the .NET WebClient object.

5. Add code to display the threads on which the code started and finished executing. To do this, update the **UpdateProductImage** method with the following code.

   (Code Snippet - *Web Forms Lab - Ex03 - Show threads*)

```
private void UpdateProductImage(Product product)
{
  string imageUrl = product.ImagePath;

  if (!string.IsNullOrEmpty(imageUrl) && !VirtualPathUtility.IsAbsolute(imageUrl))
  {
    product.ImagePath = string.Format(
         "/Images/{0}{1}",
         product.ProductId,
         Path.GetExtension(imageUrl));

    this.RegisterAsyncTask(new PageAsyncTask(async (t) =>
    {
      var startThread = Thread.CurrentThread.ManagedThreadId;

      using (var wc = new WebClient())
      {
        await wc.DownloadFileTaskAsync(imageUrl, this.Server.MapPath(product.ImagePath));
      }

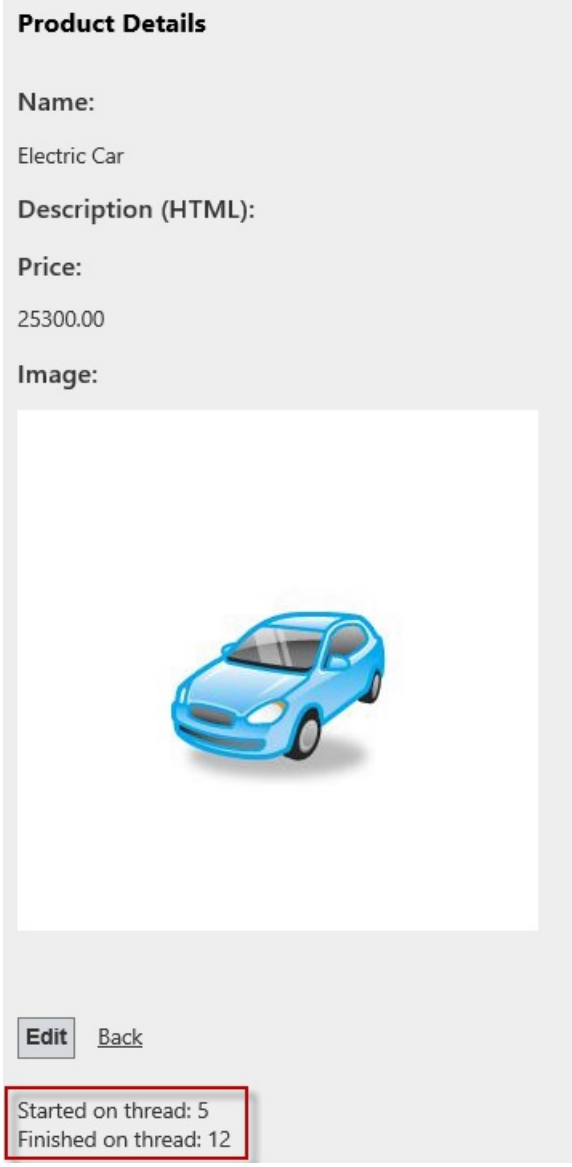
      var endThread = Thread.CurrentThread.ManagedThreadId;

      this.threadsMessageLabel.Text = string.Format("Started on thread: {0}<br /> Finished on thread:
{1}", startThread, endThread);
    }));
  }
}
```

6. Open the web site's **Web.config** file. Add the following appSetting variable.

```
<add key="aspnet:UseTaskFriendlySynchronizationContext" value="true"/>
```

7. Press **F5** to run the application and upload an image for the product. Notice the threads ID where the code started and finished may be different. This is because asynchronous tasks run on a separate thread from ASP.NET thread pool. When the task completes, ASP.NET puts the task back in the queue and assigns any of the available threads.



Downloading an image asynchronously

> **NOTE**
> Additionally, you can deploy this application to Azure following Appendix B: Publishing an ASP.NET MVC 4 Application using Web Deploy.

Summary

In this hands-on lab, the following concepts have been addressed and demonstrated:

- Use strongly-typed data-binding expressions
- Use new model binding features in Web Forms
- Use value providers for mapping page data to code-behind methods
- Use Data Annotations for user input validation
- Take advantage of unobtrusive client-side validation with jQuery in Web Forms
- Implement granular request validation
- Implement asynchronous page processing in Web Forms

Appendix A: Installing Visual Studio Express 2012 for Web

You can install **Microsoft Visual Studio Express 2012 for Web** or another "Express" version using the **Microsoft Web Platform Installer**. The following instructions guide you through the steps required to install *Visual studio Express 2012 for Web* using *Microsoft Web Platform Installer*.

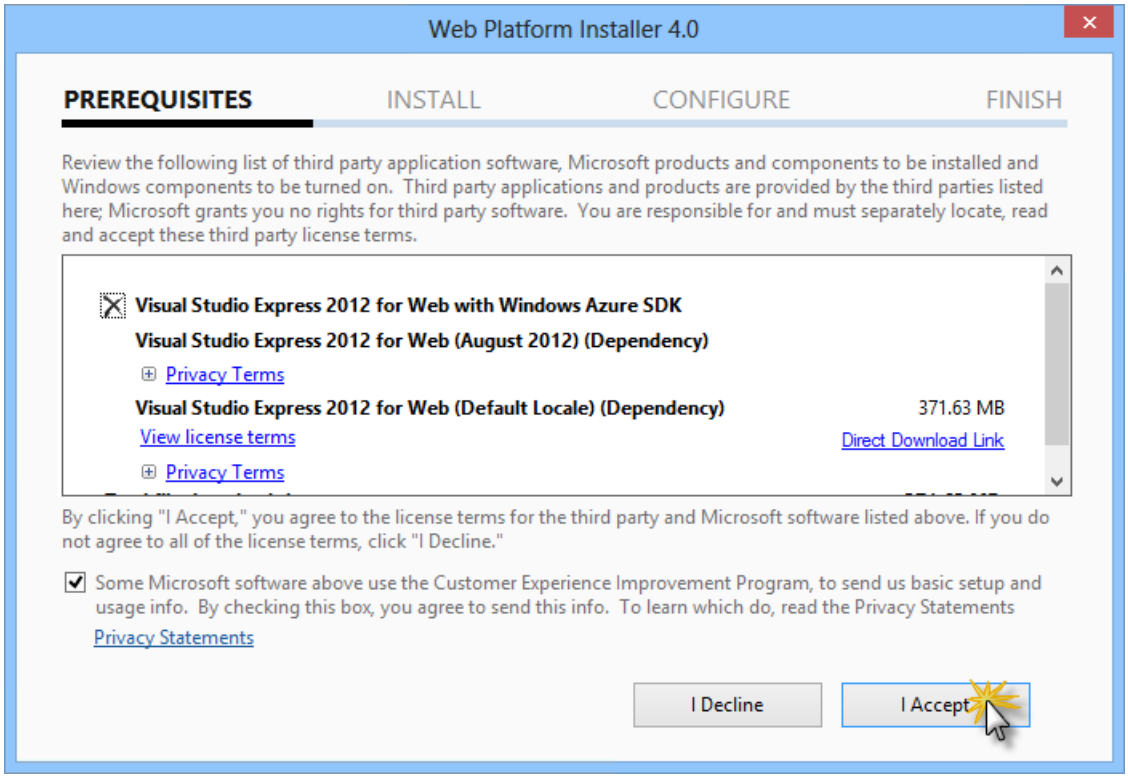
1. Go to https://go.microsoft.com/?linkid=9810169. Alternatively, if you already have installed Web Platform Installer, you can open it and search for the product "*Visual Studio Express 2012 for Web with Azure SDK*".

2. Click on **Install Now**. If you do not have **Web Platform Installer** you will be redirected to download and install it first.

3. Once **Web Platform Installer** is open, click **Install** to start the setup.

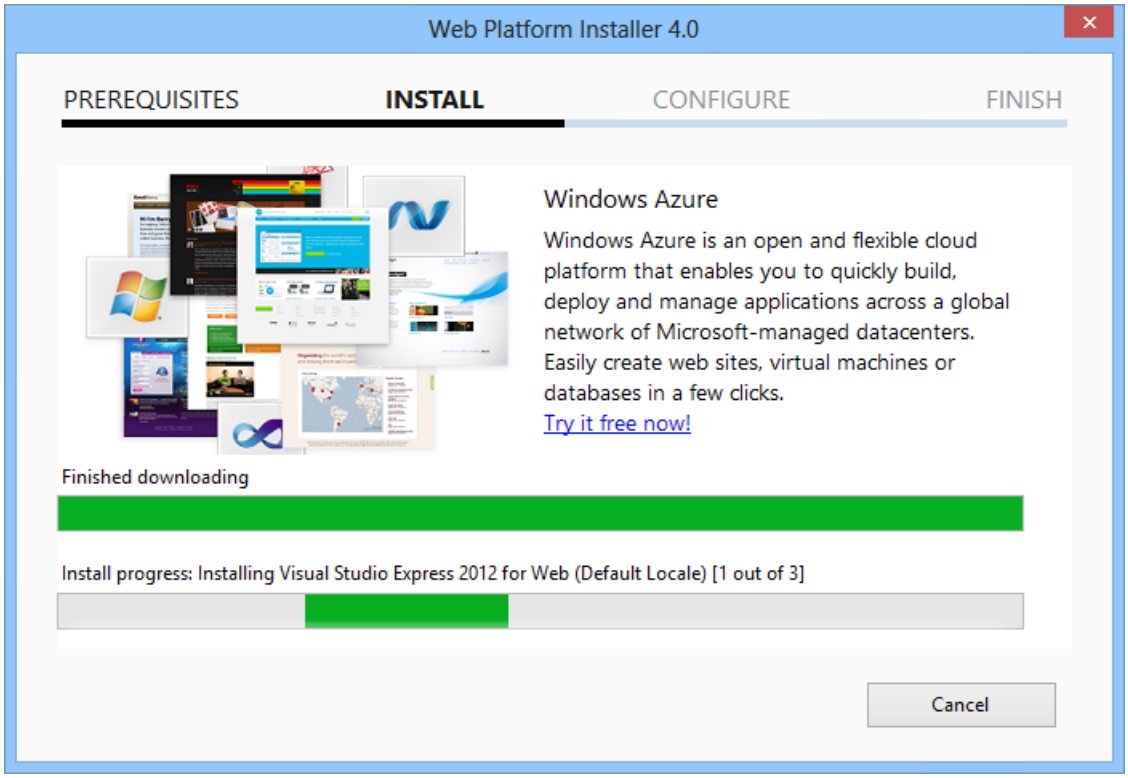
Install Visual Studio Express

4. Read all the products' licenses and terms and click **I Accept** to continue.



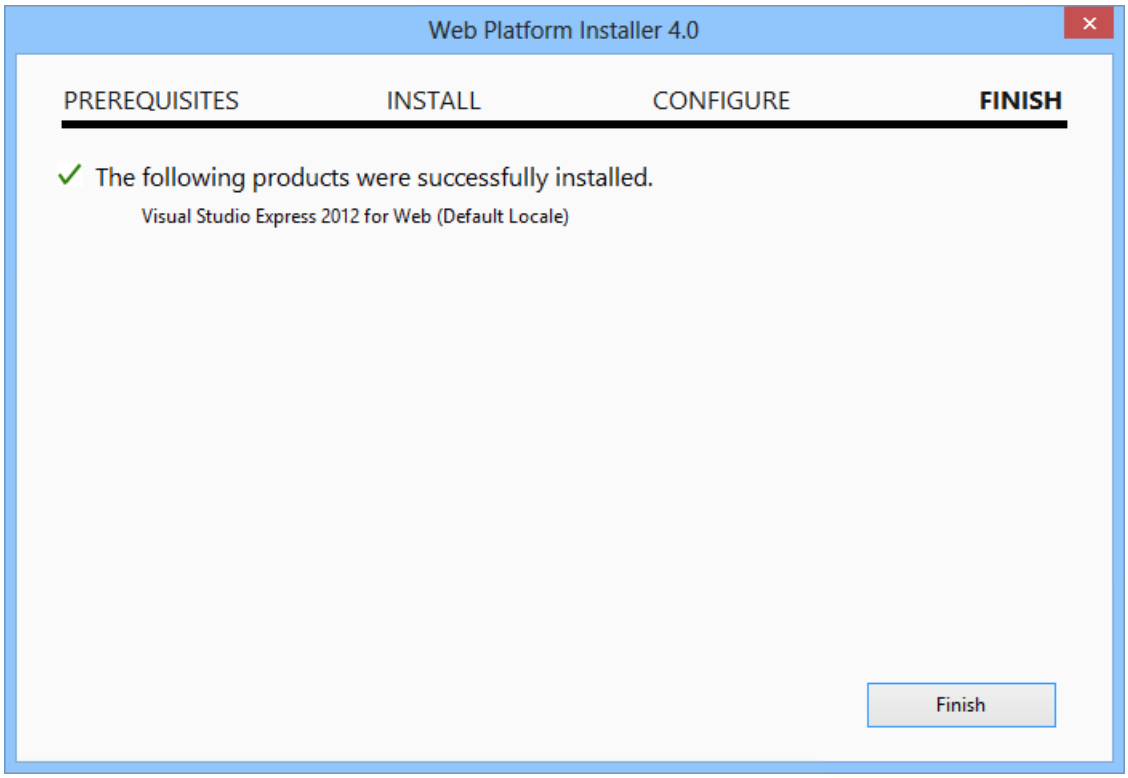
Accepting the license terms

5. Wait until the downloading and installation process completes.



Installation progress

6. When the installation completes, click **Finish**.



Installation completed

7. Click **Exit** to close Web Platform Installer.

8. To open Visual Studio Express for Web, go to the **Start** screen and start writing "**VS Express**", then click on the **VS Express for Web** tile.

VS Express for Web tile

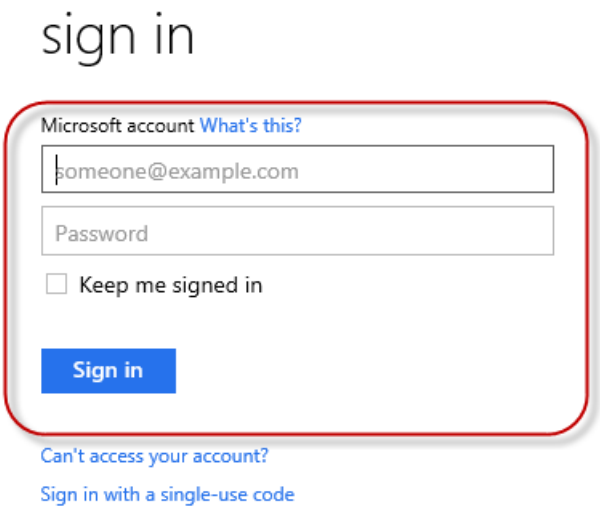
Appendix B: Publishing an ASP.NET MVC 4 Application using Web Deploy

This appendix will show you how to create a new web site from the Azure Portal and publish the application you obtained by following the lab, taking advantage of the Web Deploy publishing feature provided by Azure.

Task 1 - Creating a New Web Site from the Azure Portal

1. Go to the Azure Management Portal and sign in using the Microsoft credentials associated with your subscription.

> **NOTE**
> With Azure you can host 10 ASP.NET Web Sites for free and then scale as your traffic grows. You can sign up here.



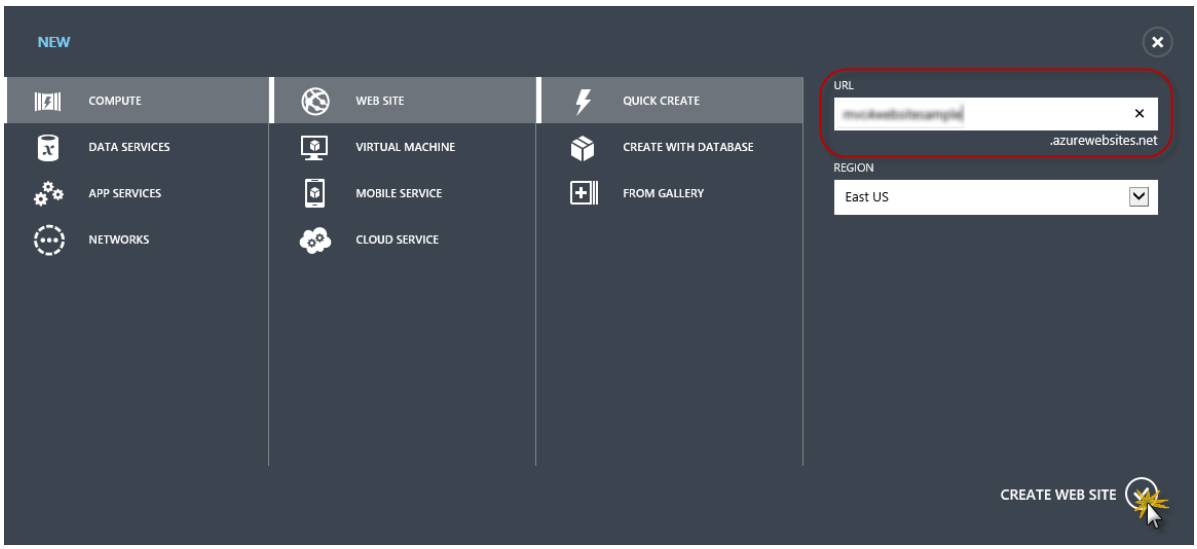
Log on to the Portal

2. Click **New** on the command bar.

Creating a new Web Site

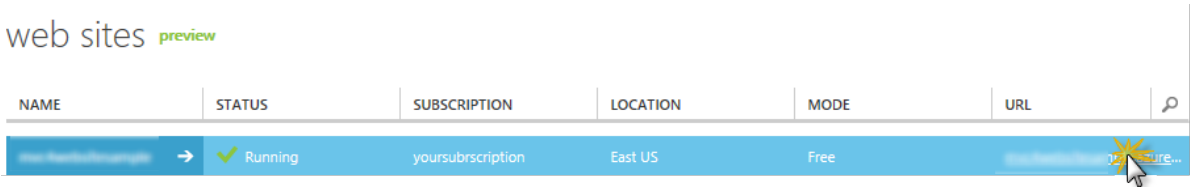
3. Click **Compute | Web Site**. Then select **Quick Create** option. Provide an available URL for the new web site and click **Create Web Site**.

> **NOTE**
>
> Azure is the host for a web application running in the cloud that you can control and manage. The Quick Create option allows you to deploy a completed web application to the Azure from outside the portal. It does not include steps for setting up a database.

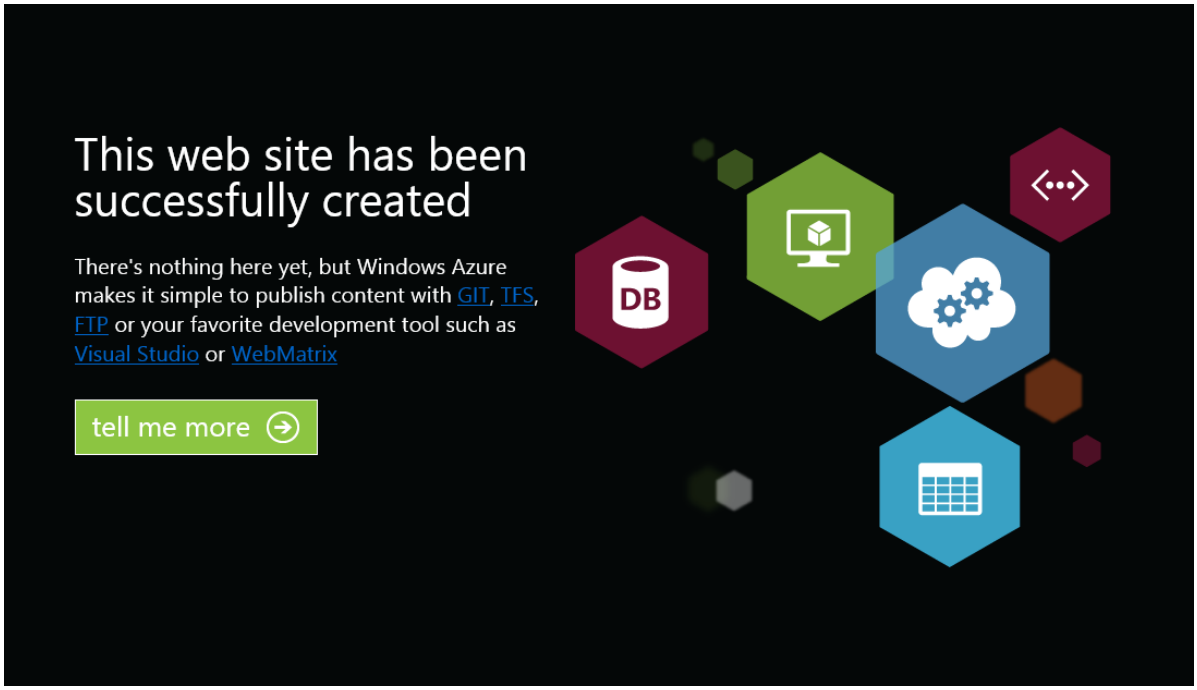


Creating a new Web Site using Quick Create

4. Wait until the new **Web Site** is created.
5. Once the Web Site is created click the link under the **URL** column. Check that the new Web Site is working.

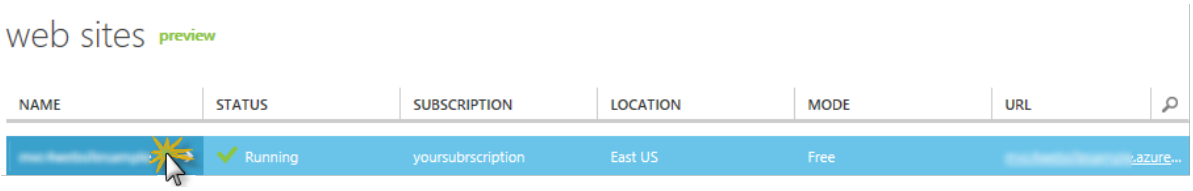


Browsing to the new web site



Web site running

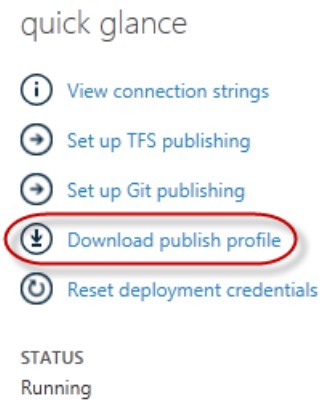
6. Go back to the portal and click the name of the web site under the **Name** column to display the management pages.



Opening the Web Site management pages

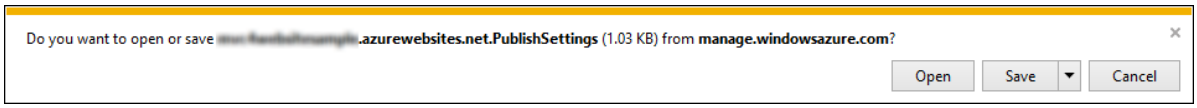
7. In the **Dashboard** page, under the **quick glance** section, click the **Download publish profile** link.

> **NOTE**
>
> The *publish profile* contains all of the information required to publish a web application to Azure for each enabled publication method. The publish profile contains the URLs, user credentials and database strings required to connect to and authenticate against each of the endpoints for which a publication method is enabled. **Microsoft WebMatrix 2**, **Microsoft Visual Studio Express for Web** and **Microsoft Visual Studio 2012** support reading publish profiles to automate configuration of these programs for publishing web applications to Azure.



Downloading the Web Site publish profile

8. Download the publish profile file to a known location. Further in this exercise you will see how to use this file to publish a web application to Azure from Visual Studio.

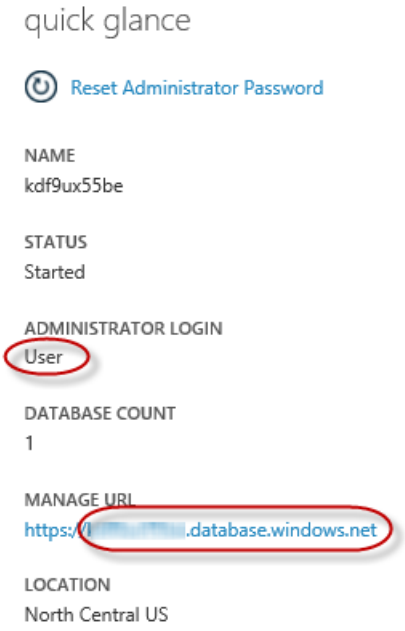


Saving the publish profile file

Task 2 - Configuring the Database Server

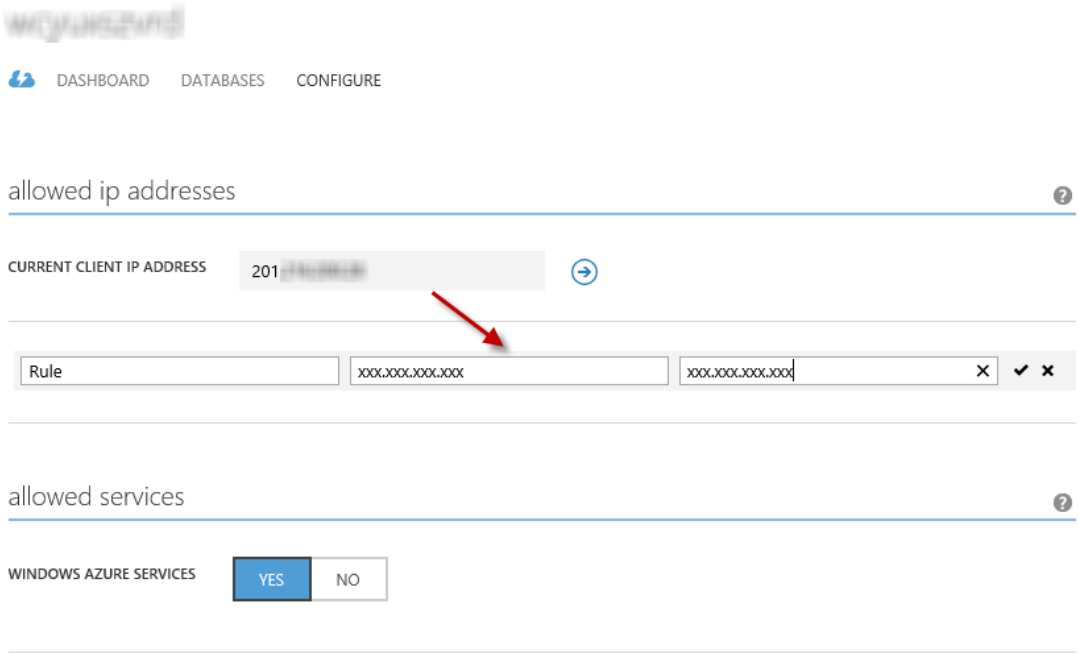
If your application makes use of SQL Server databases you will need to create a SQL Database server. If you want to deploy a simple application that does not use SQL Server you might skip this task.

1. You will need a SQL Database server for storing the application database. You can view the SQL Database servers from your subscription in the Azure Management portal at **Sql Databases | Servers | Server's Dashboard**. If you do not have a server created, you can create one using the **Add** button on the command bar. Take note of the **server name and URL, administrator login name and password**, as you will use them in the next tasks. Do not create the database yet, as it will be created in a later stage.



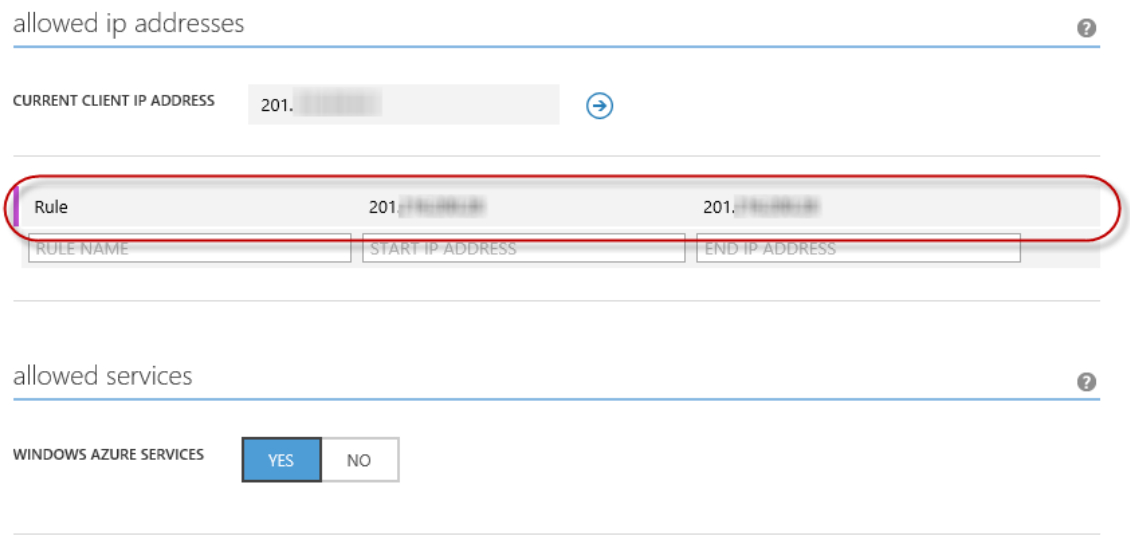
SQL Database Server Dashboard

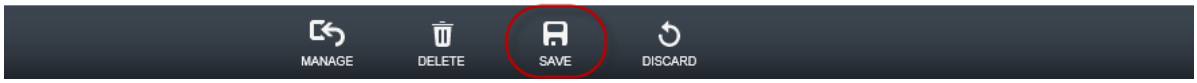
2. In the next task you will test the database connection from Visual Studio, for that reason you need to include your local IP address in the server's list of **Allowed IP Addresses**. To do that, click **Configure**, select the IP address from **Current Client IP Address** and paste it on the **Start IP Address** and **End IP Address** text boxes and click the ✔ button.



Adding Client IP Address

3. Once the **Client IP Address** is added to the allowed IP addresses list, click on **Save** to confirm the changes.

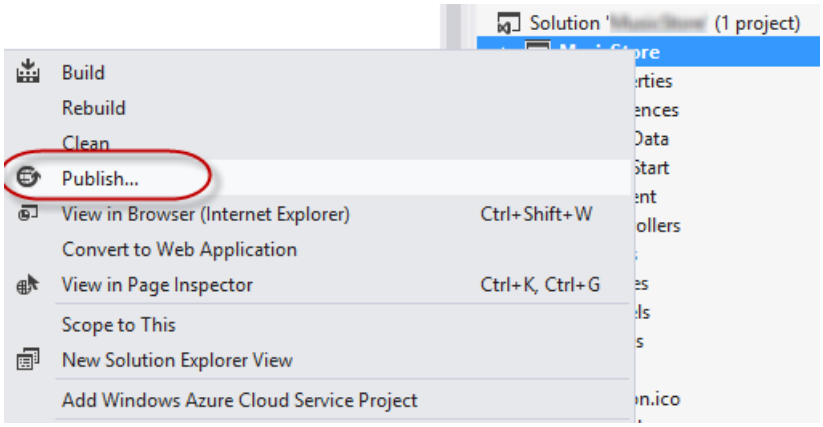




Confirm Changes

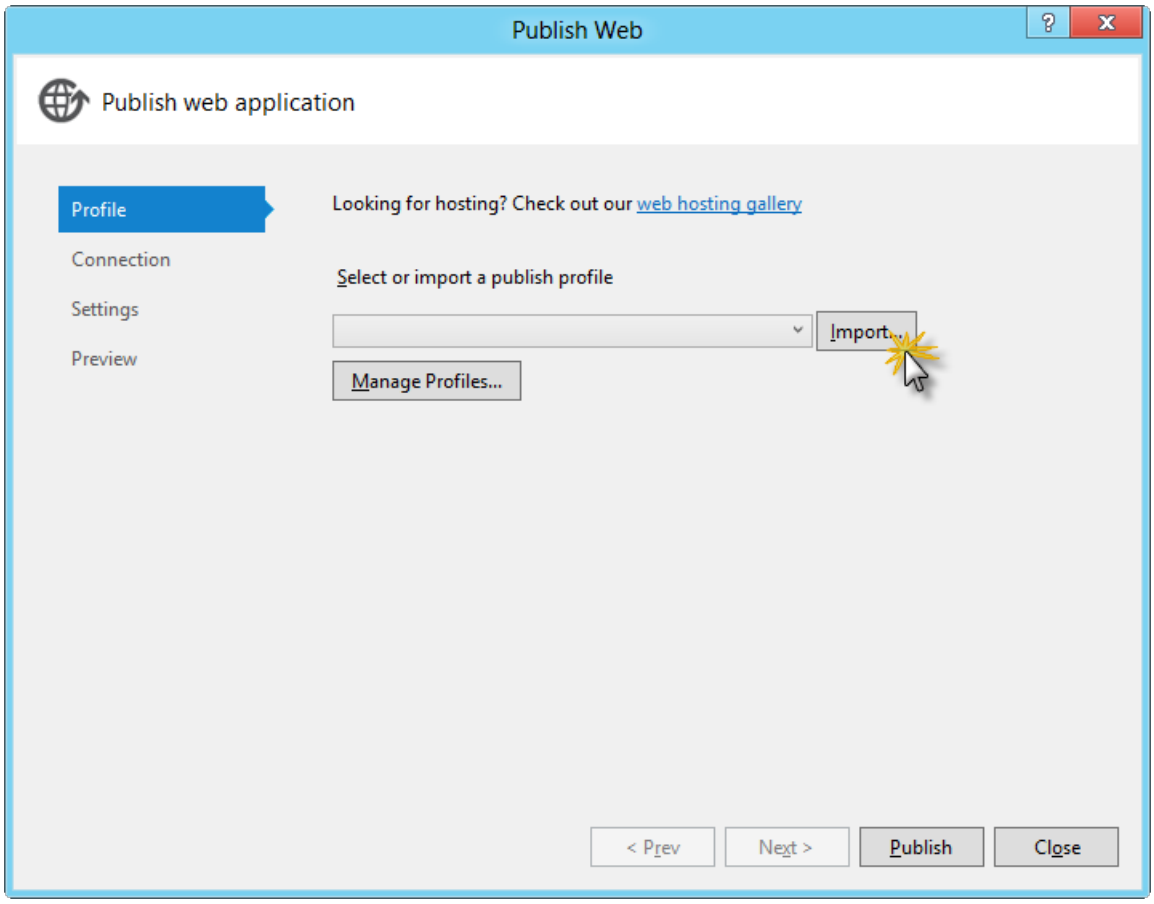
Task 3 - Publishing an ASP.NET MVC 4 Application using Web Deploy

1. Go back to the ASP.NET MVC 4 solution. In the **Solution Explorer**, right-click the web site project and select **Publish**.



Publishing the web site

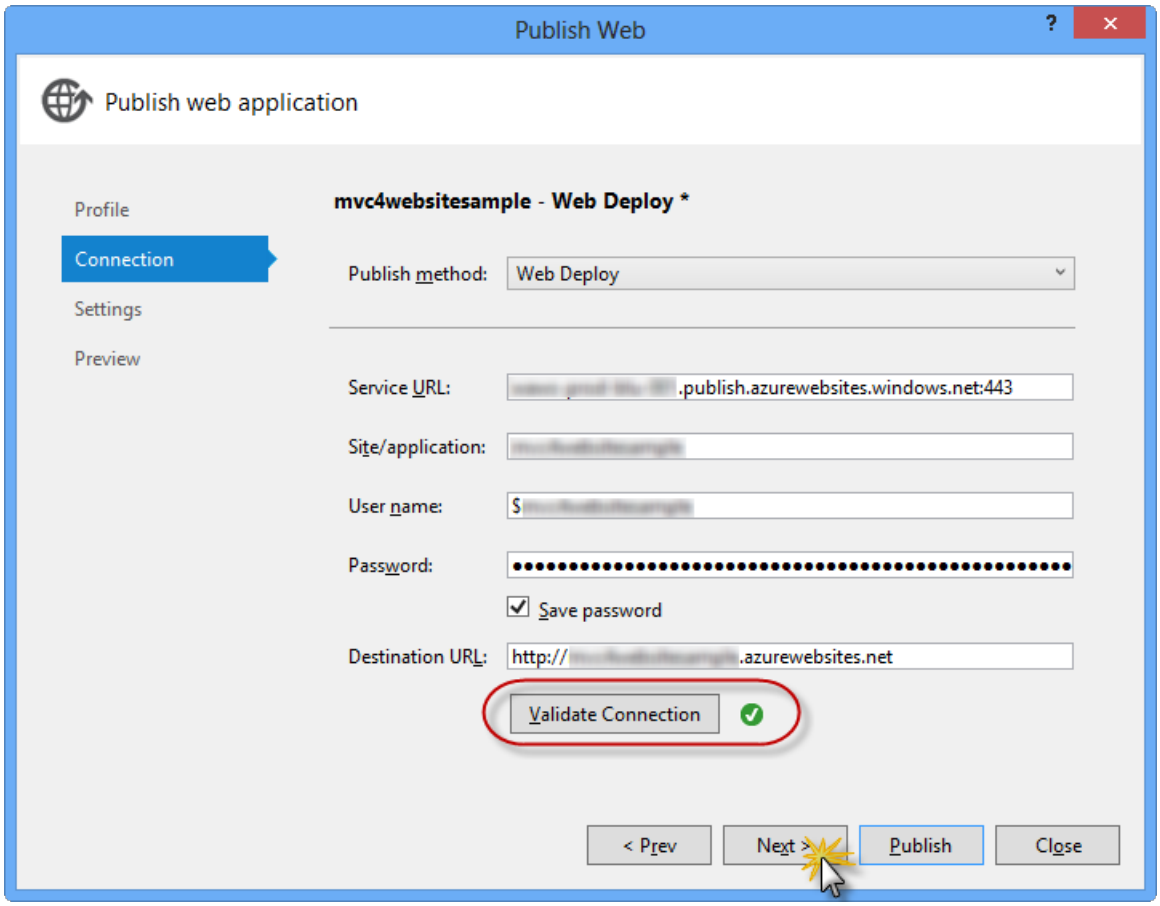
2. Import the publish profile you saved in the first task.



Importing publish profile

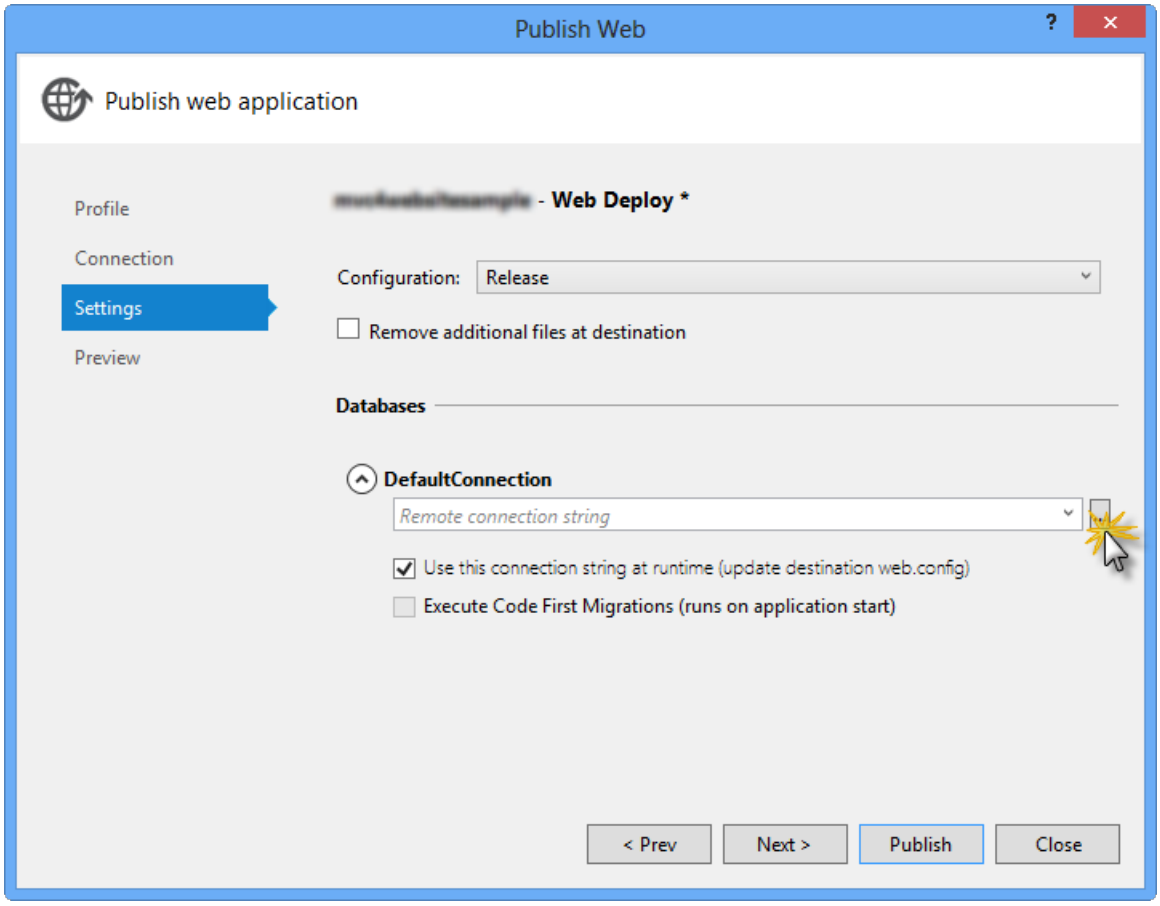
3. Click **Validate Connection**. Once Validation is complete click **Next**.

> **NOTE**
>
> Validation is complete once you see a green checkmark appear next to the Validate Connection button.



Validating connection

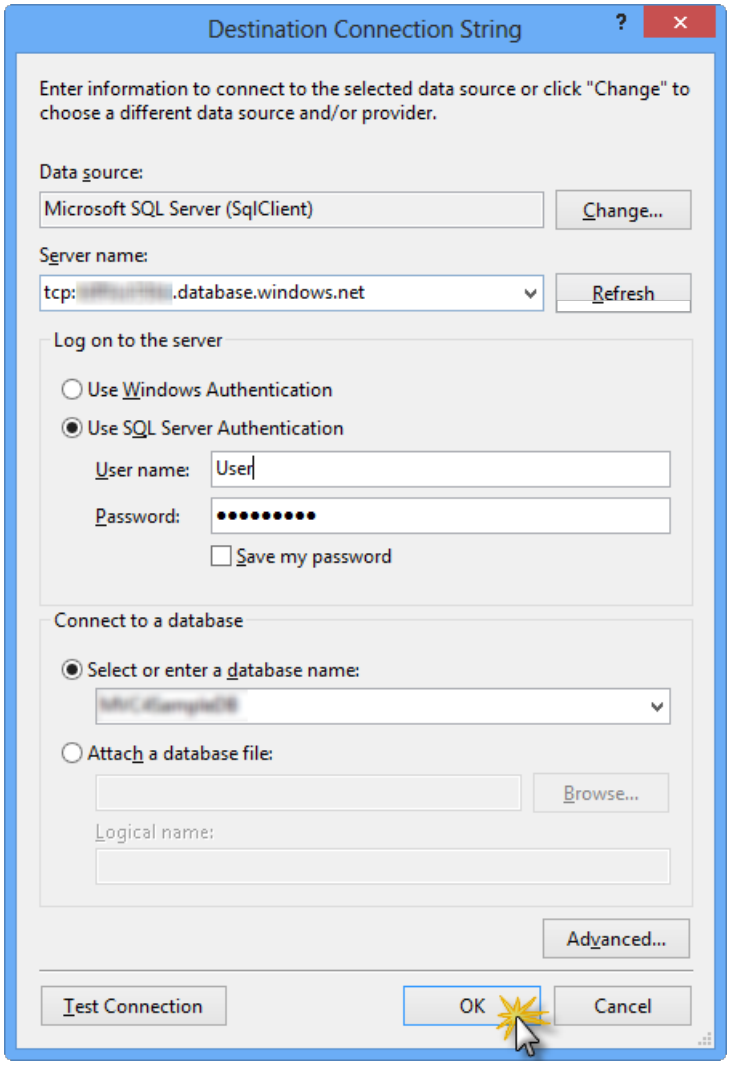
4. In the **Settings** page, under the **Databases** section, click the button next to your database connection's textbox (i.e. **DefaultConnection**).



Web deploy configuration

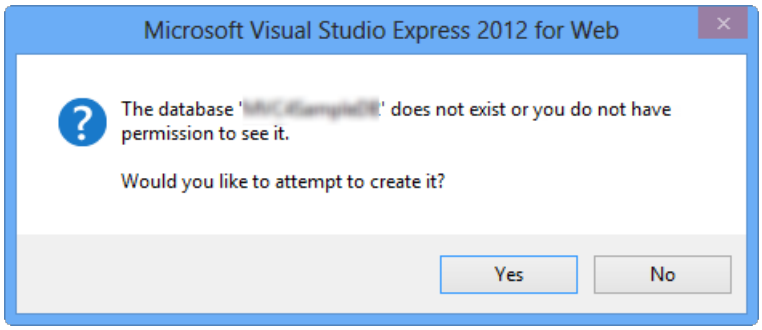
5. Configure the database connection as follows:

   - In the **Server name** type your SQL Database server URL using the *tcp:* prefix.
   - In **User name** type your server administrator login name.
   - In **Password** type your server administrator login password.
   - Type a new database name.



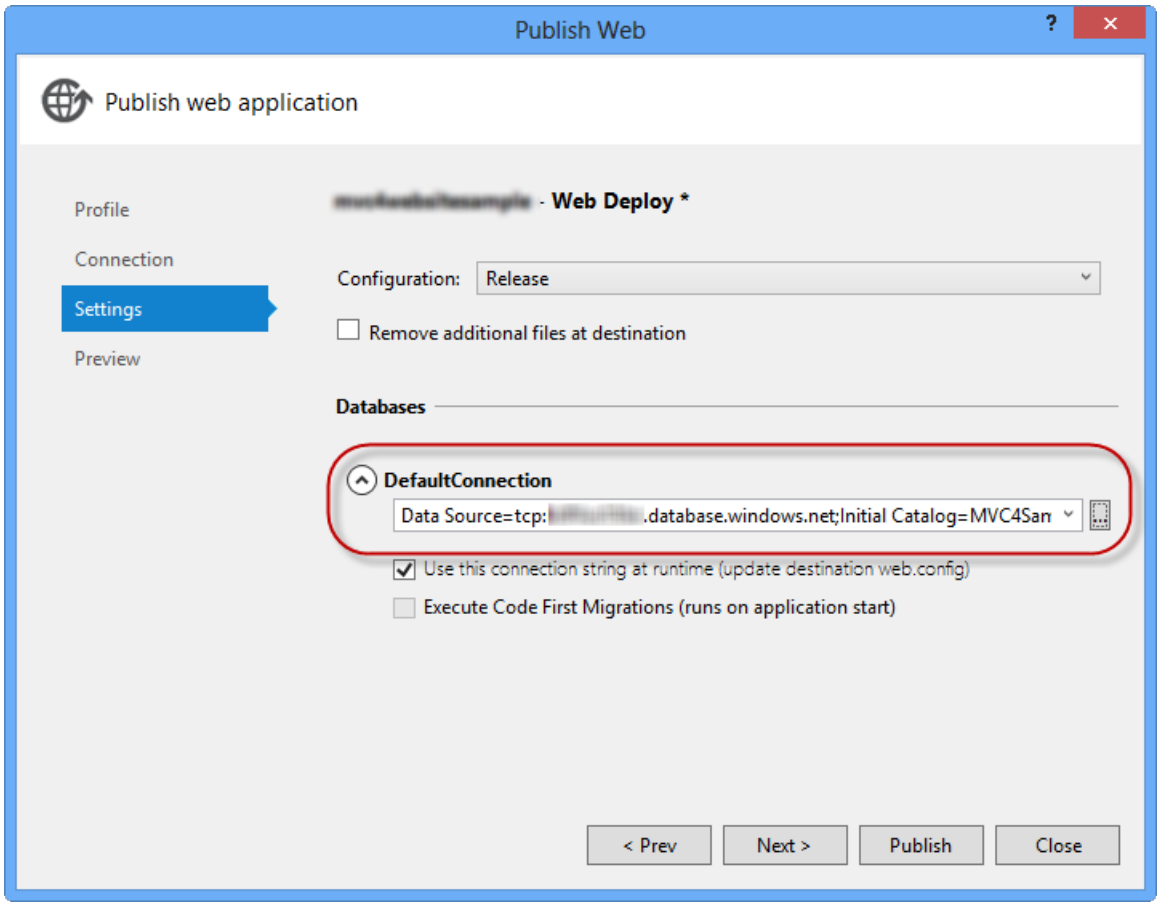
Configuring destination connection string

6. Then click **OK**. When prompted to create the database click **Yes**.



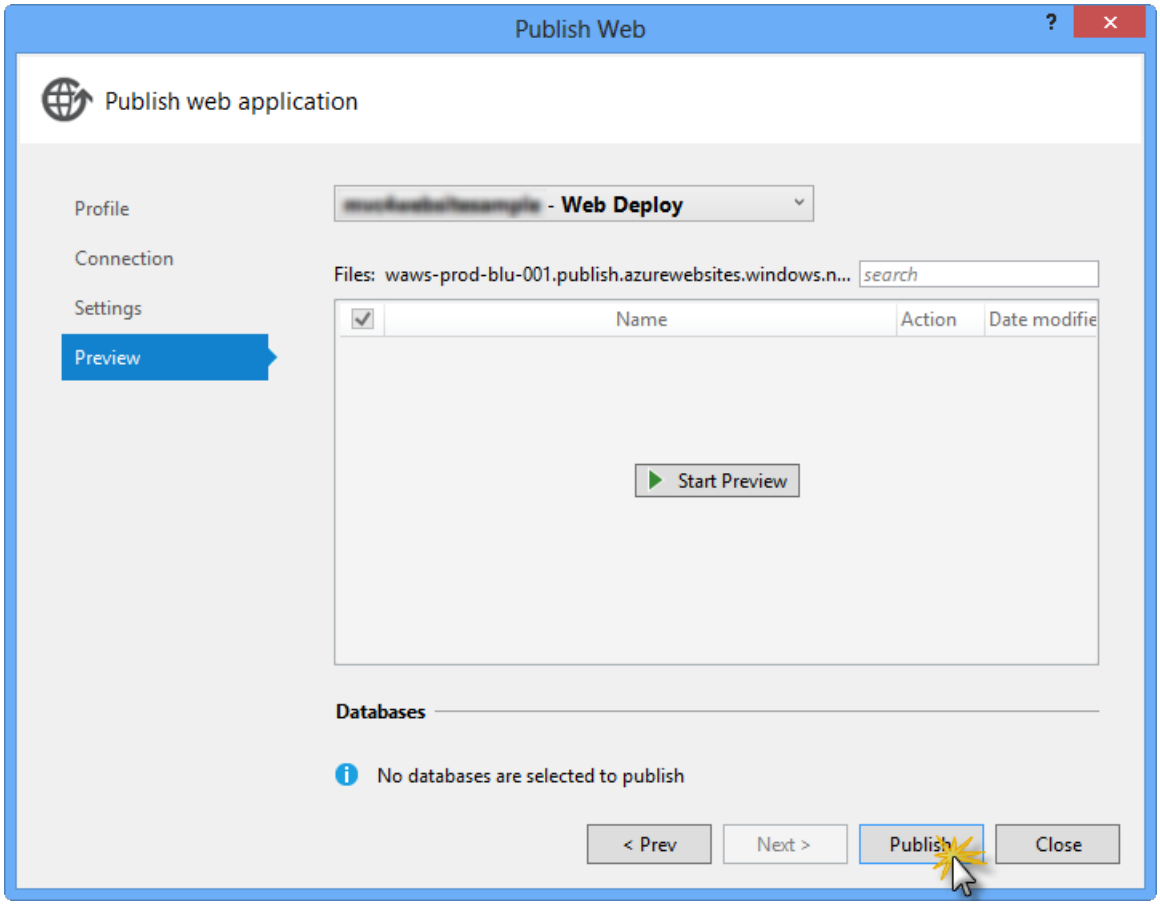
Creating the database

7. The connection string you will use to connect to SQL Database in Azure is shown within Default Connection textbox. Then click **Next**.



Connection string pointing to SQL Database

8. In the **Preview** page, click **Publish**.



Publishing the web application

9. Once the publishing process finishes, your default browser will open the published web site.

Appendix C: Using Code Snippets

With code snippets, you have all the code you need at your fingertips. The lab document will tell you exactly when you can use them, as shown in the following figure.

6. Insert the following code inside the body of the **Run** method.

(Code Snippet – *AzurePlatformTrainingKit - Ex1-Task 1 -Step 5- Run method*)

C#

```
public override void Run()
{
  while (true)
  {
    // retrieve a new message from the queue
    CloudQueueMessage message = queue.GetMessage();
    if (message != null)
    {
      ProcessMessage(message);
      queue.DeleteMessage(msg);
    }
  }
}
```

Using Visual Studio code snippets to insert code into your project

To add a code snippet using the keyboard (C# only)

1. Place the cursor where you would like to insert the code.
2. Start typing the snippet name (without spaces or hyphens).
3. Watch as IntelliSense displays matching snippets' names.
4. Select the correct snippet (or keep typing until the entire snippet's name is selected).
5. Press the Tab key twice to insert the snippet at the cursor location.

```
namespace PhotoGallery.Controllers
{
    public class PhotoController : Controller
    {
        mvc
    }
}
```

IMvcFilter
MVC4LabEx02HomeControllerUsings
MVC4LabEx02IndexAction
MVC4LabEx02PhotoController
MVC4LabEx02Photomodel
MVC4LabEx03iPhoneDisplayMode
MVC4LabEx04Attributes
MVC4LabEx04GetAsync
MVC4LabEx04SendAsyncwithCancellationTok

MVC4 Lab - Ex02 - PhotoController

Start typing the snippet name

```
static void Main(string[] args)
{
    AzureTrainingKitServiceBus
}
```

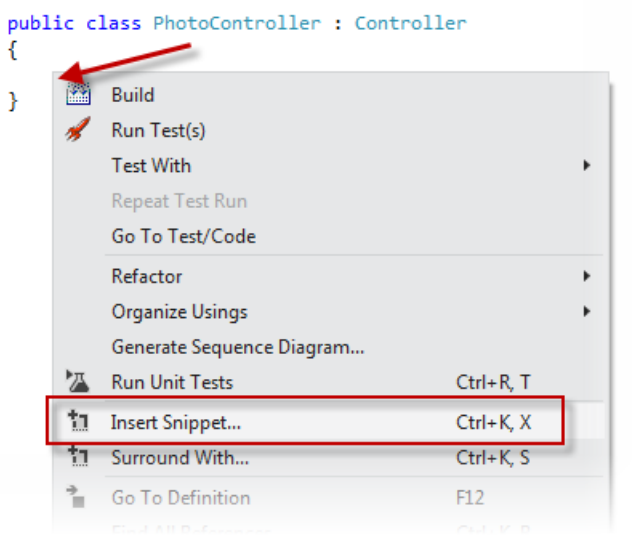
Press Tab to select the highlighted snippet

```
static void Main(string[] args)
{
    request.Headers.Add("$accountKey", accountKey);
    request.Headers.Add("$uniqueUserID", uniqueUserId);
}
```

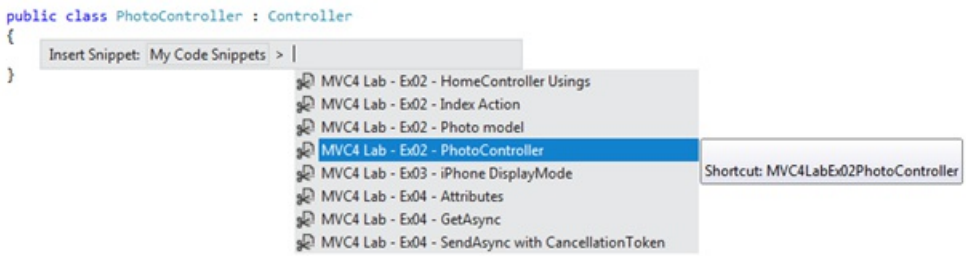
Press Tab again and the snippet will expand

To add a code snippet using the mouse (C#, Visual Basic and XML) 1. Right-click where you want to insert the code snippet.

1. Select **Insert Snippet** followed by **My Code Snippets**.
2. Pick the relevant snippet from the list, by clicking on it.



Right-click where you want to insert the code snippet and select Insert Snippet



Pick the relevant snippet from the list, by clicking on it

Using Page Inspector in Visual Studio 2012

2/20/2020 • 21 minutes to read • Edit Online

by Web Camps Team

In this Hands-on Lab, you will discover a new tool to find and fix web page issues in Visual Studio - the Page Inspector.

Page Inspector is a new tool that brings browser diagnostics tools to Visual Studio and provides an integrated experience among the browser, ASP.NET, and source code. It renders a web page (HTML, Web Forms, ASP.NET MVC, or Web Pages) directly within the Visual Studio IDE and lets you examine both the source code and the resulting output. Page Inspector enables you to easily decompose a website, rapidly build pages from the ground up, and quickly diagnose issues.

Nowadays we have a number of Web frameworks that create flexible and scalable websites in a timely manner, such as ASP.NET MVC and WebForms. On the other hand, it gets harder to find issues on the pages because the IDE does not support the designer view in template-based pages and dynamic content. Therefore, these websites have to be opened in a browser to see how they appear to a user.

Web developers use external tools to find issues that regularly run in the browser. Then, they return to the IDE and start fixing. This back and forth activity among the IDE, the browser and the profiling tools can be inefficient, and sometimes requires a fresh deployment and cache cleaning each time you want to reproduce an issue.

Page Inspector bridges a gap in Web development between the client (browser tools) and the server (ASP.NET and source code) by bringing together the best of both worlds using a combined set of features.

Using Page Inspector, you can see which elements in the source files (including server-side code) have produced the HTML markup to be rendered in the browser. Page Inspector also lets you modify CSS properties and DOM element attributes to see the changes reflected immediately in the browser.

This hands-on lab will walk you through the Page Inspector features and show you how you can use them to fix issues in Web applications. **This lab contains two exercises using similar flows but targeting different technologies. If you are an ASP.NET MVC Developer, follow exercise one; if you are a WebForms developer follow exercise two**.

This lab walks you through the enhancements and new features previously described by applying minor changes to a sample Web application provided in the Source folder.

All sample code and snippets are included in the Web Camps Training Kit, available at https://go.microsoft.com/fwlink/?LinkID=248297&clcid=0x409.

Objectives

In this hands-on lab, you will learn how to:

- Decompose a Web Site using Page Inspector
- Inspect and preview CSS styles changes with Page Inspector
- Detect and fix issues in your web pages using Page Inspector

Prerequisites

You must have the following items to complete this lab:

- Microsoft Visual Studio Express 2012 for Web or superior (read Appendix A for instructions on how to install

it).

- Internet Explorer 9 or higher

Exercises

This hands-on lab includes the following exercises:

1. Exercise 1: Using Page Inspector in ASP.NET MVC Projects
2. Exercise 2: Using Page Inspector in WebForms Projects

> **NOTE**
> Each exercise is accompanied by a starting solution, located in the Begin folder of the exercise, that allows you to follow each exercise independently of the others. Inside the source code for an exercise, you will also find an End folder containing a Visual Studio solution with the code that results from completing the steps in the corresponding exercise. You can use these solutions as guidance if you need additional help as you work through this hands-on lab.

Estimated time to complete this lab: **30 minutes**.

Exercise 1: Using Page Inspector in ASP.NET MVC Projects

In this exercise, you will learn how to preview and debug an **ASP.NET MVC 4** solution using **Page Inspector**. First, you will perform a brief lap around the tool to learn the features that facilitate the Web debugging process. Then, you will work in a web page that contains styling issues. You will learn how to use Page Inspector to find the source code that generates the issue and fix it.

Task 1 - Exploring Page Inspector

In this task, you will learn how to use the Page Inspector in the context of an ASP.NET MVC 4 project that shows a photo gallery.

1. Open the **Begin** solution located at **Source/Ex1-MVC4/Begin/** folder.

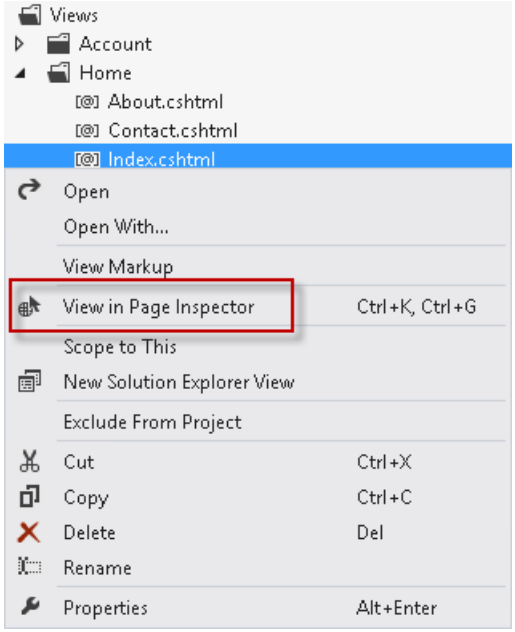
   a. You will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

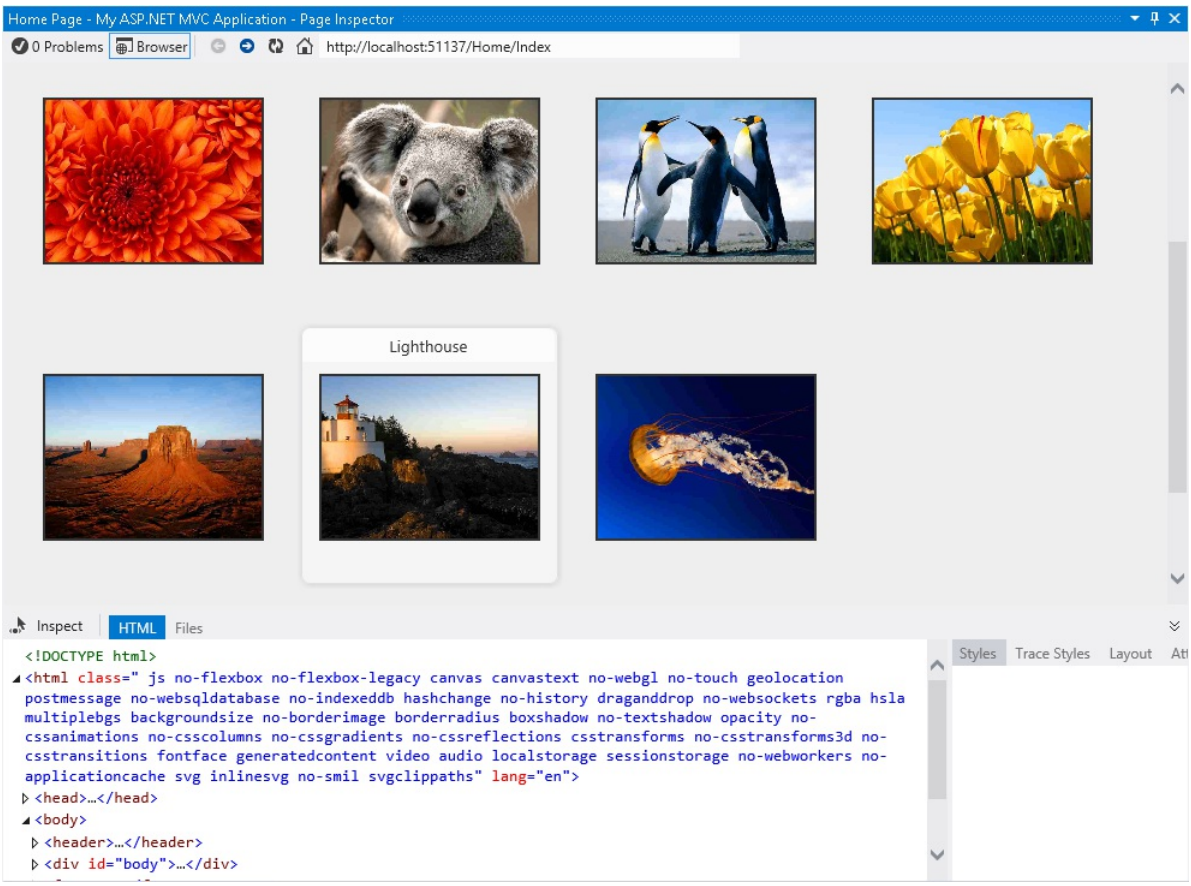
   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. In the Solution Explorer, locate **Index.cshtml** view under the **/Views/Home** project folder, right-click it and select **View in Page Inspector**.



Selecting a file to preview in Page Inspector

3. The Page Inspector window will show the */Home/Index* URL mapped to the source View you selected.



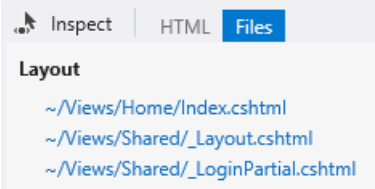
The first contact with Page Inspector

The Page Inspector tool is integrated in your Visual Studio environment. The inspector contains an embedded browser, together with a powerful HTML profiler. Notice that you do not have to run the solution to see how your pages look.

> **NOTE**
>
> When the width of Page Inspector browser is less than the width of the opened page, you will not see the page properly. If that happens, adjust the width of the Page Inspector.

4. Click the **Files** tab in Page Inspector.

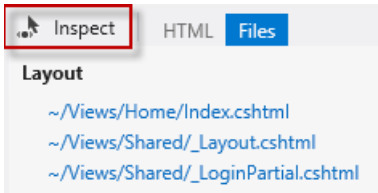
   You will see all the source files that are composing the Index page. This feature helps to identify all the elements at a glance, especially when you are working with partial views and templates. Notice that you can also open each of the files if you click the links.



The Files tab

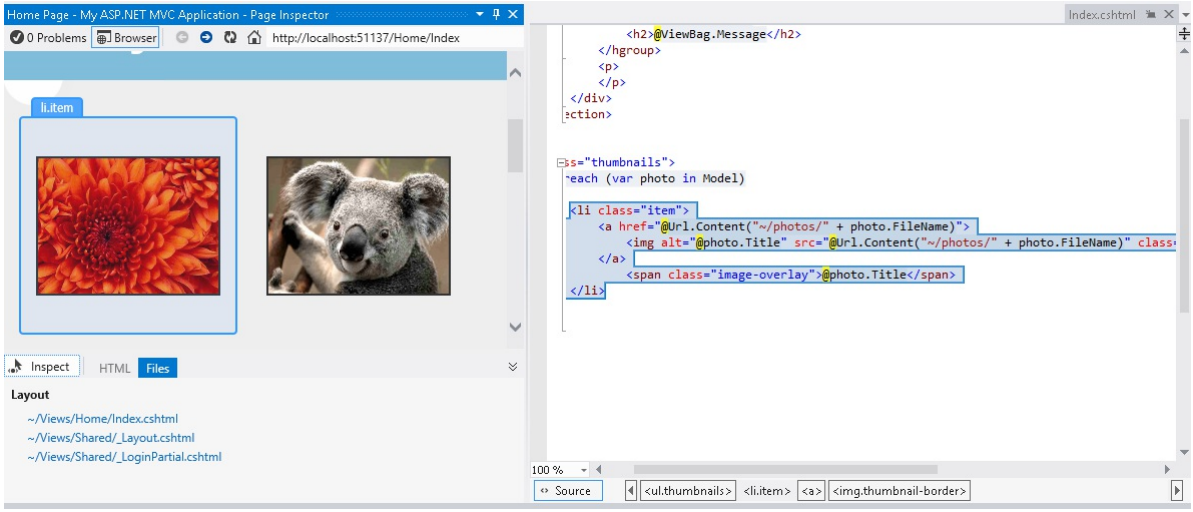
5. Click the **Toggle Inspection Mode** button, located at the left of the tabs.

   This tool will let you select any element of the page and see its HTML and Razor code.



Toggle Inspection Mode button

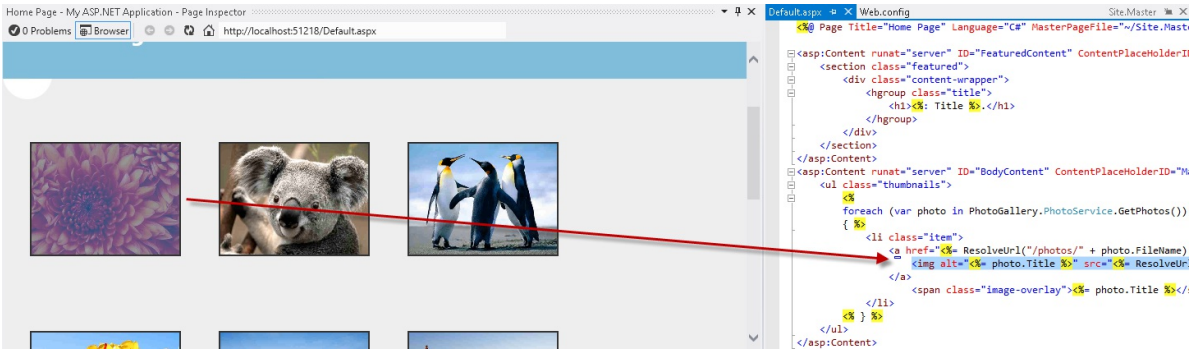
6. In the Page Inspector browser, move the mouse pointer over the page elements. While you move the mouse pointer over any part of the rendered page, the element type is displayed and the corresponding source markup or code is highlighted in the Visual Studio editor.



Inspection mode in action

> **NOTE**
>
> Do not maximize the Page Inspector window or you will not be able to see the preview tab showing the source code. If you click the element in Page Inspector when it is maximized, the source code of the selection will appear but it will hide the Page Inspector window.

If you pay attention to the **Index.cshtml** file, you will notice that the portion of source code that generates the selected element is highlighted. This feature facilitates the editing of long source files, providing a direct and fast way to access the code.



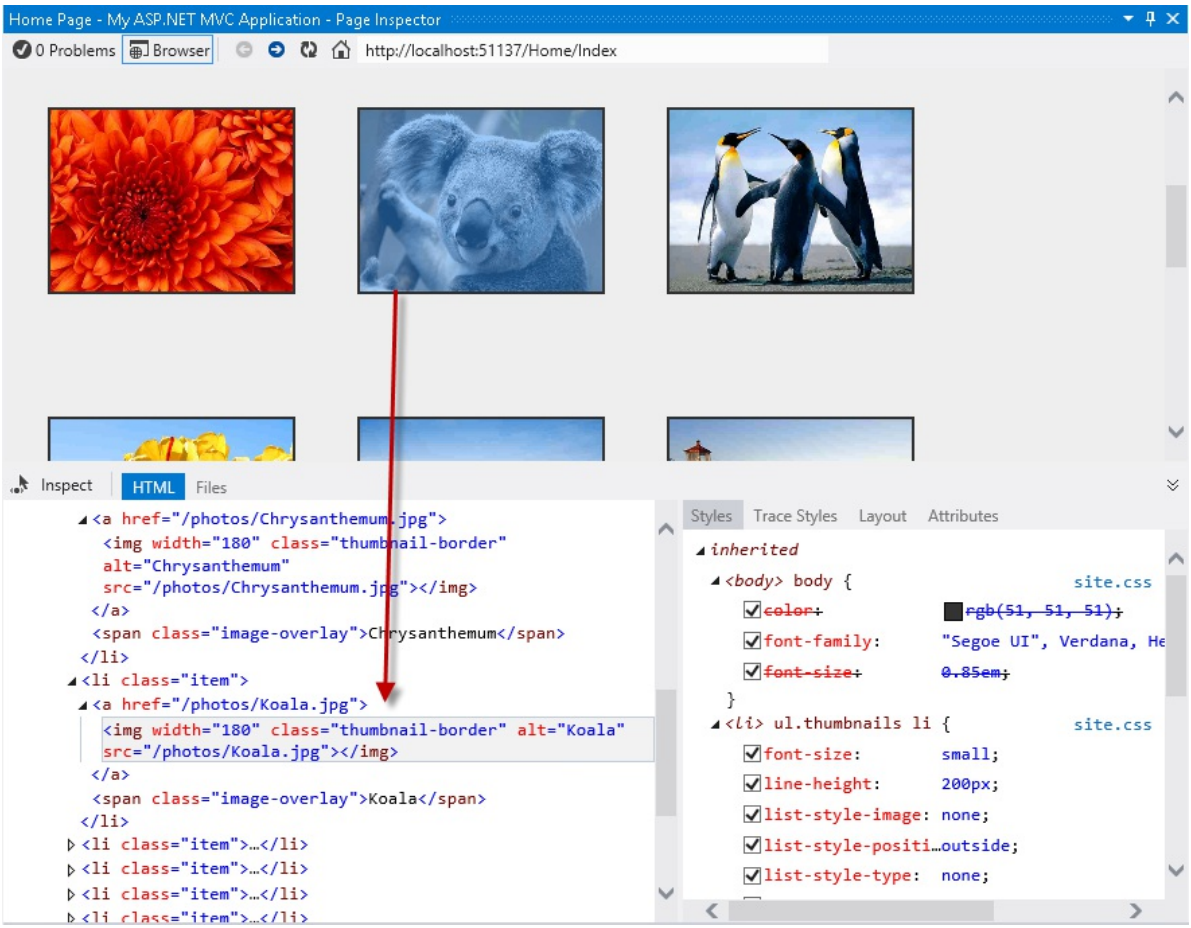
Inspecting elements

7. Click the **Toggle Inspection Mode** button (Inspect) to disable the cursor.

8. Select the **HTML** tab to display the HTML code rendered in the Page Inspector browser.

9. In the HTML markup, locate the list item with the Koala link and select it.

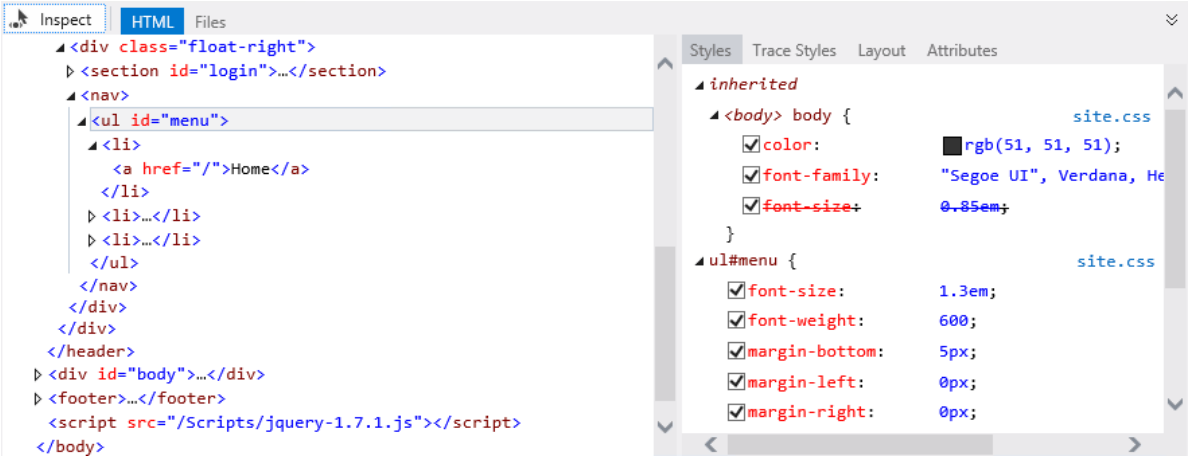
   Notice that when you select the code, the corresponding output is automatically highlighted in the browser. This feature is useful to see how an HTML block is rendered on the page.



Selecting HTML element in the page

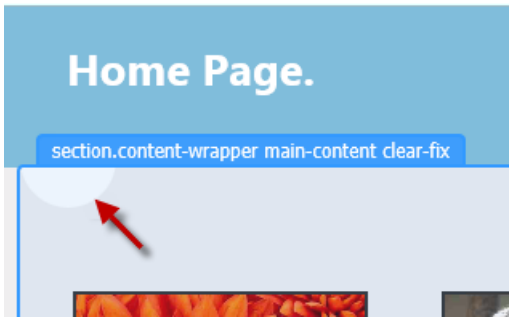
10. Click the **Toggle Inspection Mode** button to enable *Inspection Mode* and click the navigation bar. On the right of the HTML code, in the Styles pane, you will see a list with the CSS styles applied to the selected element.

> **NOTE**
>
> Since the header is a part of the site layout, Page Inspector will also open _Layout.cshtml file and highlight the segment of code affected.



Discovering styles and source files of a selected element

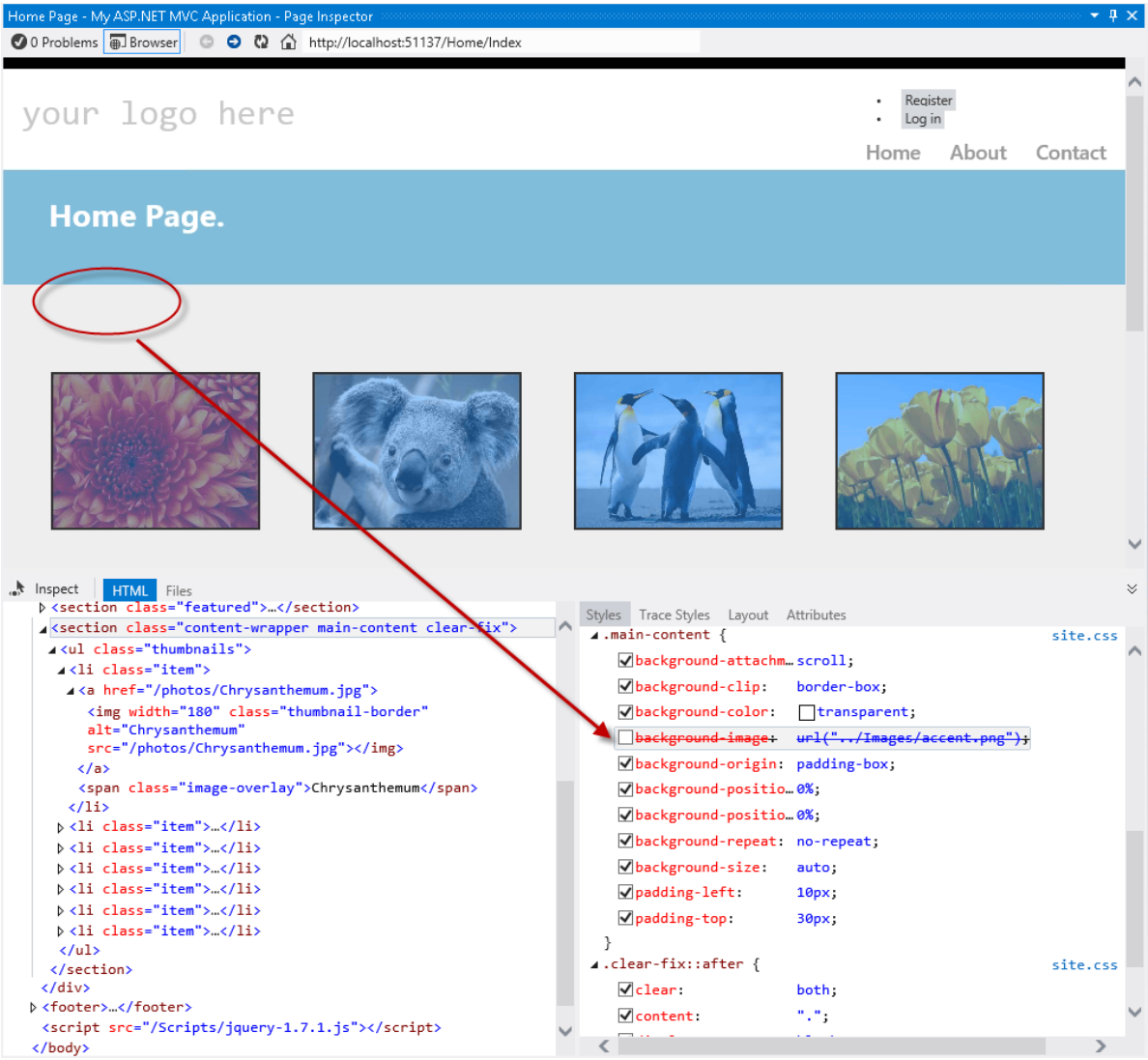
11. With the toggle inspection pointer enabled, move the mouse pointer below the blue featured bar and click the half circle.



Selecting an element

12. In the Styles pane, locate the **background-image** item under the **.main-content** group. **Uncheck** the **background-image** and see what happens. You will notice that the browser will reflect the changes immediately and the circle is hidden.

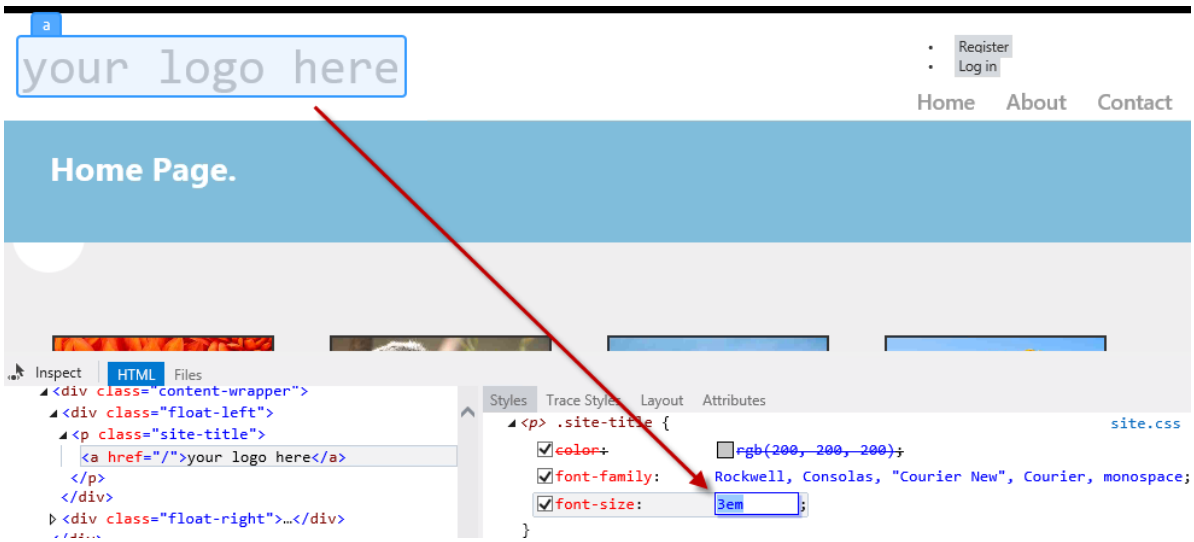
> **NOTE**
>
> The changes you apply on the Page Inspector Styles tab do not affect the original stylesheet. You can uncheck styles or change their values as many times as you want, but they will be restored after refreshing the page.



Enabling and disabling CSS styles

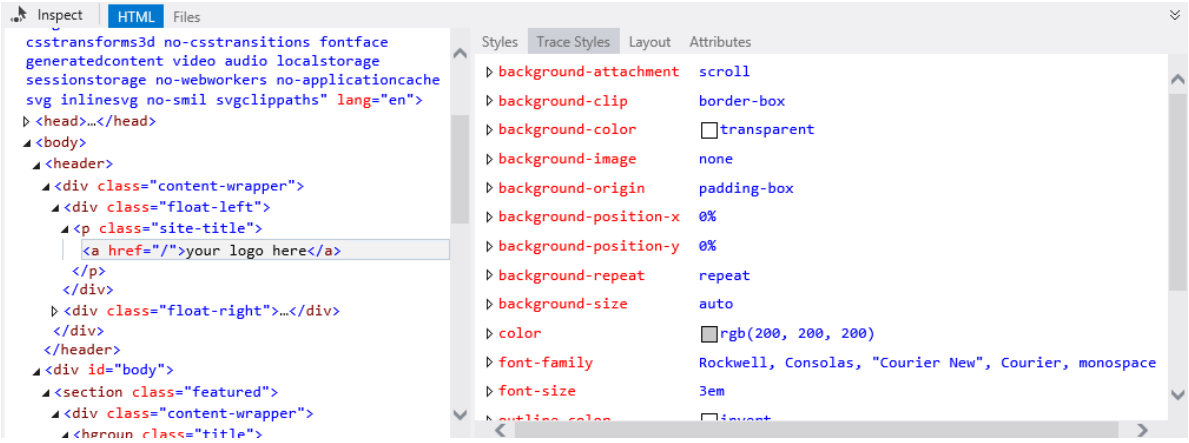
13. Now, click the '**your logo here**' text on the header using the inspection mode.

14. In the **Styles** tab, locate the **font-size** CSS attribute under the **.site-title** group. Double-click the attribute value and replace the 2.3 em value with **3 em**, and then press **ENTER**. Notice that the title looks bigger.



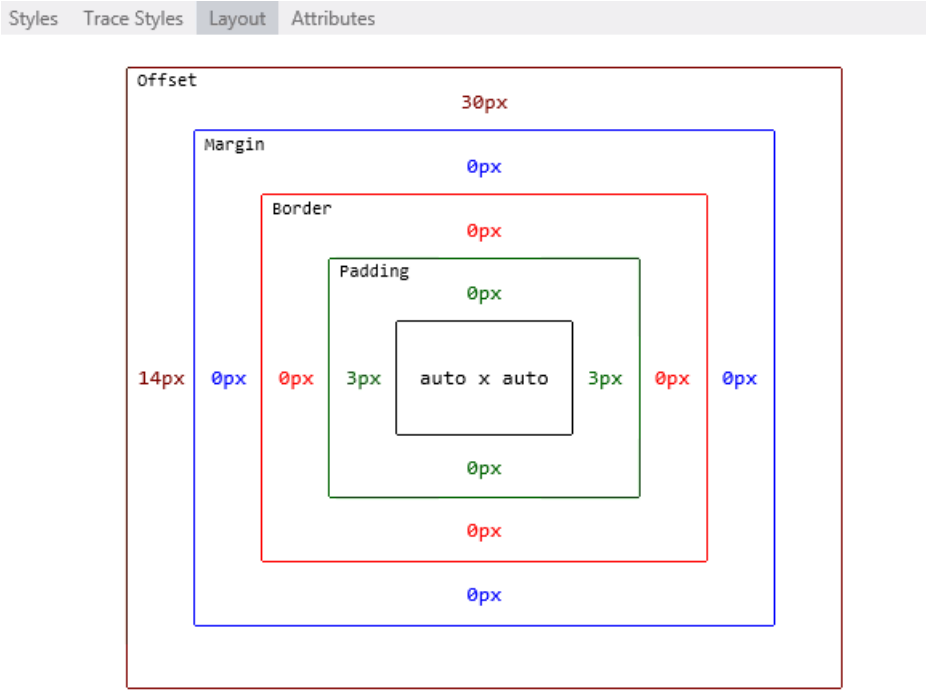
Changing CSS values in the Page Inspector

15. Click the **Trace Styles** tab, located in the right pane of Page Inspector. This is an alternative way to see all the styles applied to the selection, ordered by attribute name.



CSS styles tracing of the selected element

16. Another feature of Page Inspector is the Layout pane. Using the inspection mode, select the navigation bar and then click the **Layout** tab on the right pane. You will see the exact size of the selected element, as well as its offset, margin, padding and border size. Notice that you can also modify the values from this view.



Element layout in Page Inspector

Task 2 - Finding and Fixing Style Issues in the Photo Gallery

How would you diagnose Web pages issues with previous versions of Visual Studio? You are likely familiar with web debugging tools that run outside the Visual Studio IDE, like Internet Explorer Developer Tools or Firebug. Browsers only understand HTML, scripting and styles, while an underlying framework generates the HTML that will be rendered. For that reason, you often need to deploy the whole site to see how web pages look like.

You had probably followed these steps when you wanted to detect and fix an issue in your web site:

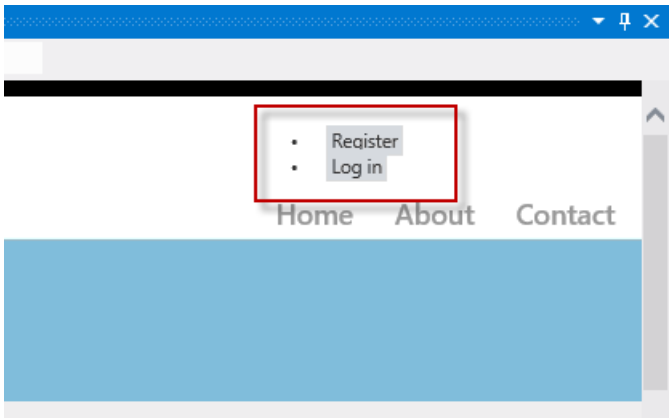
1. Run the Solution from Visual Studio, or deploy the page on the web server.
2. In the browser, open the developer tools you use, or simply open the source code and the styles and try to match the issue. To find the files involved, you would have used the "Search" or "Search in files" features with the name of the style classes.
3. Once the error is detected, stop the Web browser and the server.
4. Clear the browser cache.
5. Return to Visual Studio to apply a fix. Repeat all the steps to test.

As there is no real WYSIWYG in ASP.NET MVC 4, most of the style issues are detected on a later stage, after running or deploying the web application. Now, with Page Inspector, it is possible to preview any page without running the solution.

In this task, you will use the Page inspector and fix some issues the Photo Gallery application.

1. Using Page Inspector, locate the **Register** and the **Log in** links at the left side of the header.

   Notice that the links are not displayed at the expected place on the right, and they are shown like a bulleted list. You will now align the links to the right and restyle them accordingly.



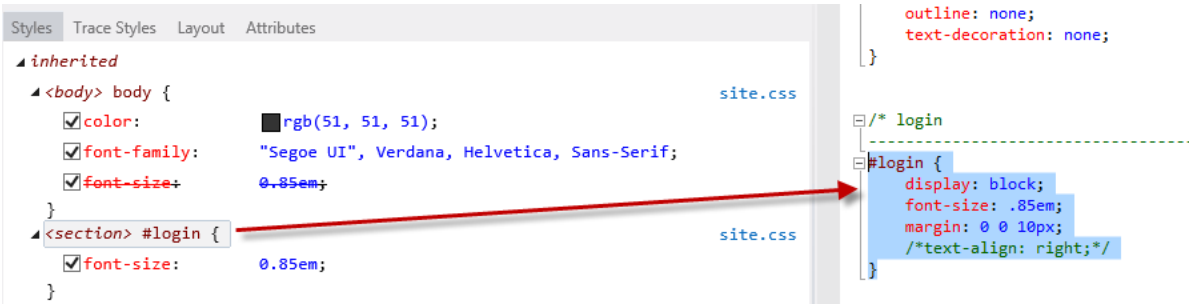
   Locating the Register and Log in links

2. With Toggle Inspection Mode selected, click close to, but not on, the Register link to open its code.

   Notice that the source code of the links is located in the **_LoginPartial.cshtml** file, not the Index.cshtml nor the _Layout.cshtml, which are the places you might look in first place. You have been placed directly in the correct source file.

3. In the **Styles** tab, locate and click the **<section> #login** item, which is the HTML container for these links.

   Notice that the **#login** style is automatically located in **Site.css** after you click. Moreover, the code is now highlighted.

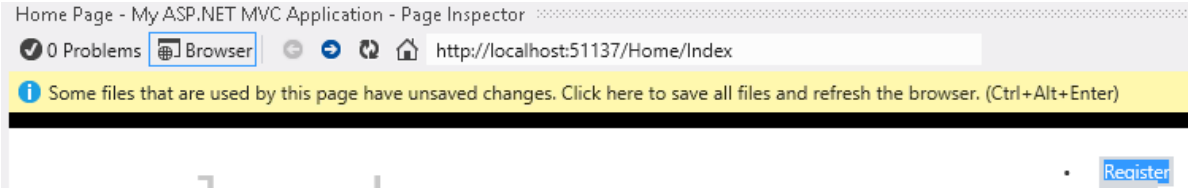


   Selecting the CSS styles

4. Uncomment the **text-align** attribute in the highlighted code by removing the opening and closing characters and save the **Site.css** file.

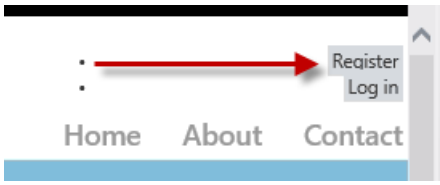
   Page Inspector is aware of all the different files that compose the current page, and it can detect when any of these files change. It alerts you whenever the current page in browser is not in sync with the source files.

5. In the Page Inspector browser, click the bar located below the address bar to reload the page.



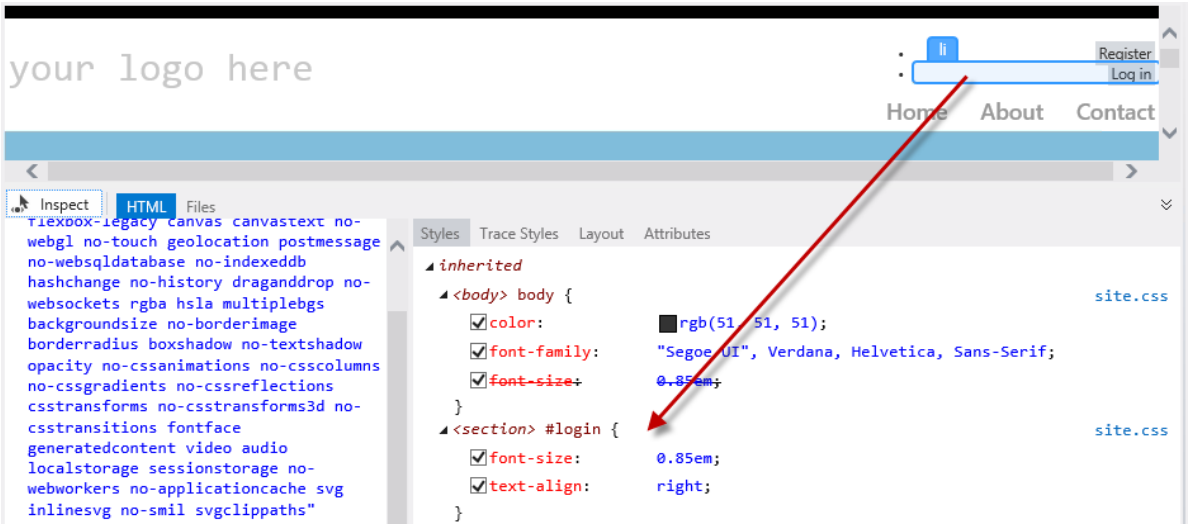
Reloading the page

The links are now at the right, but they still look like a bulleted list. Now, you will remove the bullets and align the links horizontally.



Updated page

6. Using the inspection mode, select any of the **<li>** items that contain the "Register" and "Log in" links. Then, click the **<section> #login** item to access **Styles.css** code.



Finding the style

7. In **Style.css**, uncomment the code for **#login li** items. The style you are adding will hide the bullet and display the items horizontally.

```
#login li
{
    display: inline;
    list-style: none;
}
```

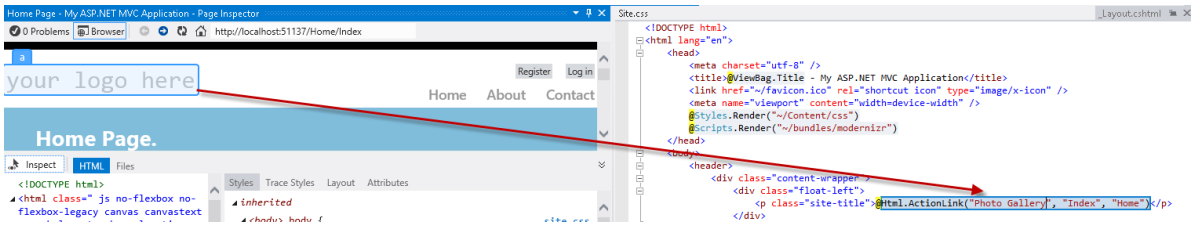
Restyling the login links

8. Save **Style.css** file and click once on the bar located below the address to reload the page. Notice that the links appear correctly.

Links aligned to the right side

9. Finally, you will change the header title. Use the inspection mode to click **your logo here** text and get to the source code that generates it.

10. Now you are in **_Layout.cshtml**, replace '**your logo here**' text with '**Photo Gallery**'. Save and update the Page Inspector browser.



Assigning a new title

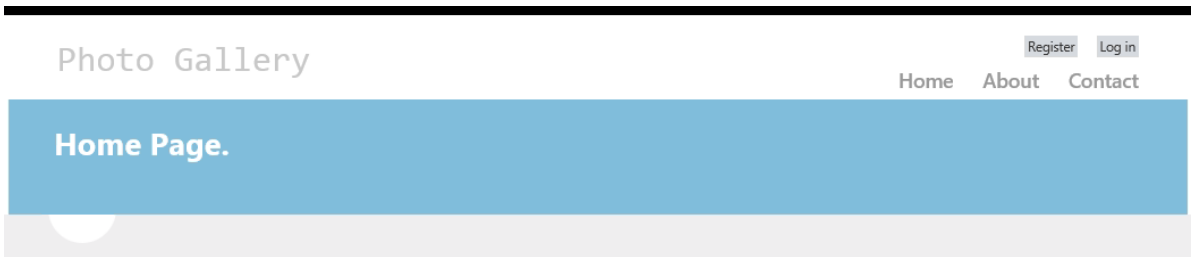


Photo Gallery page updated

11. Finally, select the **PhotoGallery** project and press **F5** to run the app. Check out all the changes work as expected.

Exercise 2: Using Page Inspector in WebForms Projects

In this exercise, you will learn how to preview and debug a WebForms solution using Page Inspector. You will first perform a brief lap around the tool to learn the Page Inspector features that facilitate the Web debugging process. Then, you will work in a web page that contains styling issues. You will learn how to use Page Inspector to find the source code that generates the issue and fix it.

Task 1 - Exploring Page Inspector

In this task, you will learn how to use the Page Inspector features in the context of a WebForms project that shows a photo gallery.

1. Open the **Begin** solution located at **Source/Ex2-WebForms/Begin/** folder.

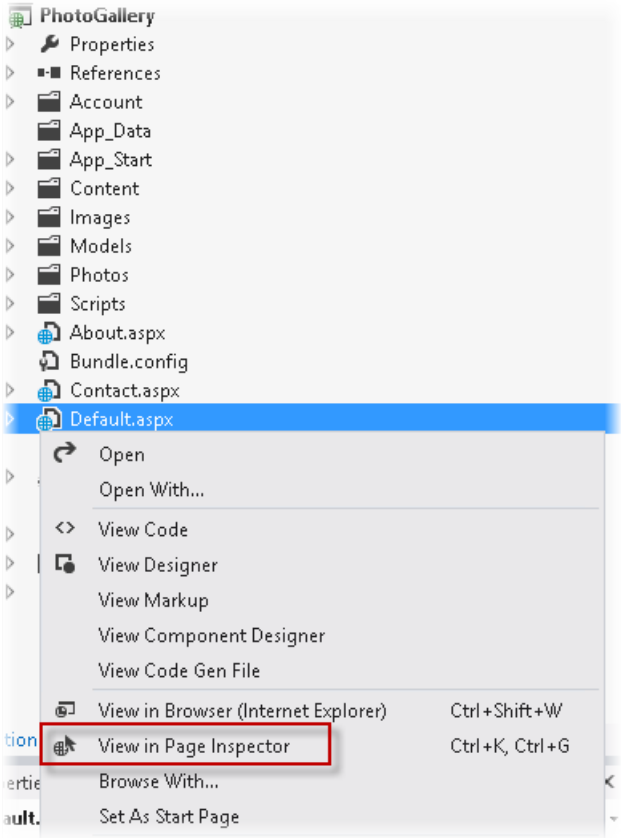
   a. You will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

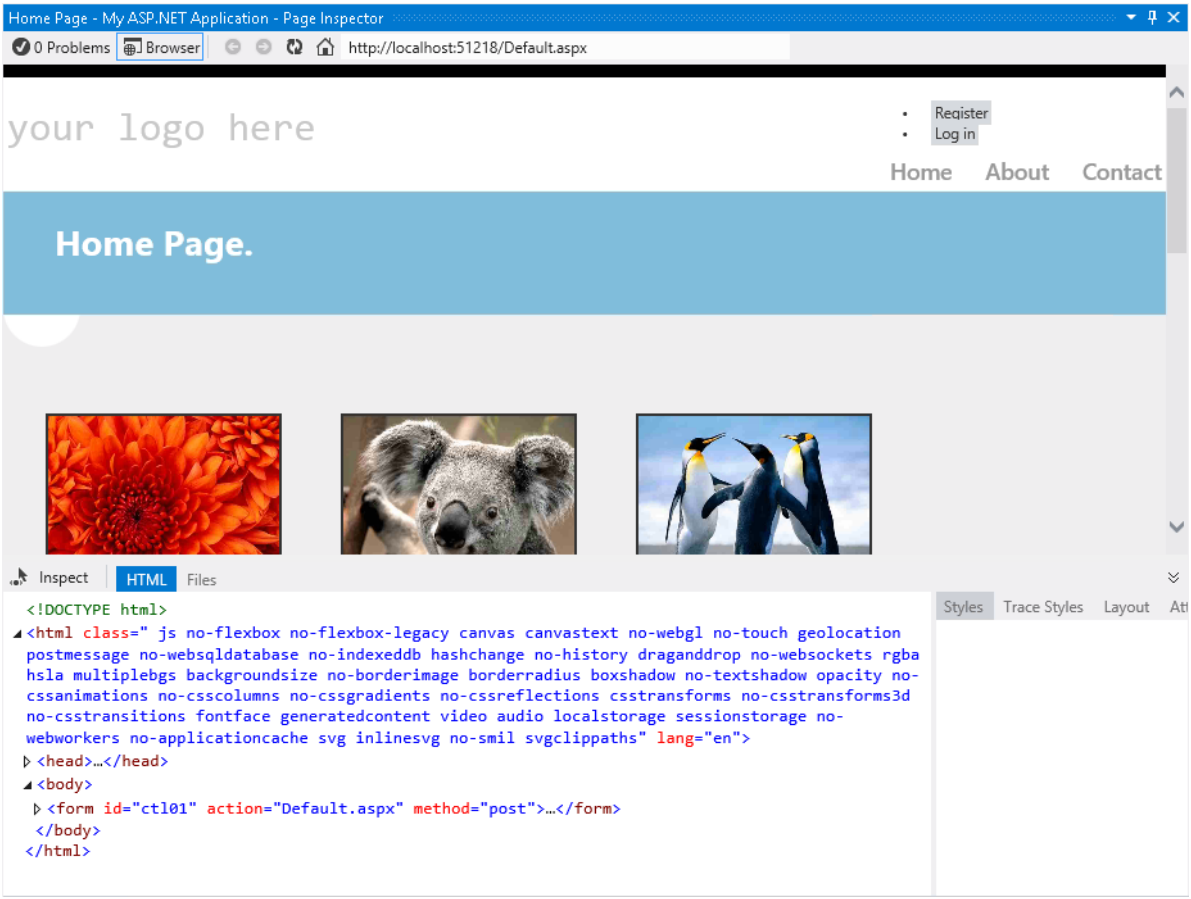
   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. In the Solution Explorer, locate **Default.aspx** page, right-click it and select **View in Page Inspector**.



Opening Default.aspx with Page Inspector

3. The Page Inspector window will show Default.aspx.



Viewing Default.aspx in Page Inspector

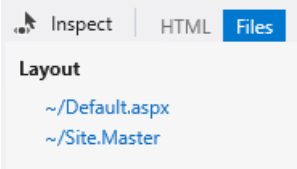
The Page Inspector tool is integrated in your Visual Studio environment. The inspector contains an embedded browser, together with a powerful HTML profiler that will show the selected code. Notice that

you do not have to run the solution to see how your pages look.

> **NOTE**
>
> When the width of Page Inspector browser is less than the width of the opened page, you will not see the page properly. If that happens, adjust the width of the Page Inspector.

4. Click the **Files** tab in Page Inspector.

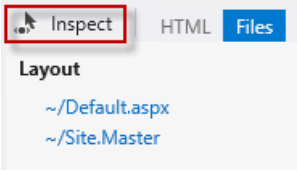
   You will see all the source files that are composing the rendered Default page. This is a useful feature to identify all the elements at a glance, especially when you are working with User Controls and Master Pages. Notice that you can also navigate to each of the files.



The Files tab

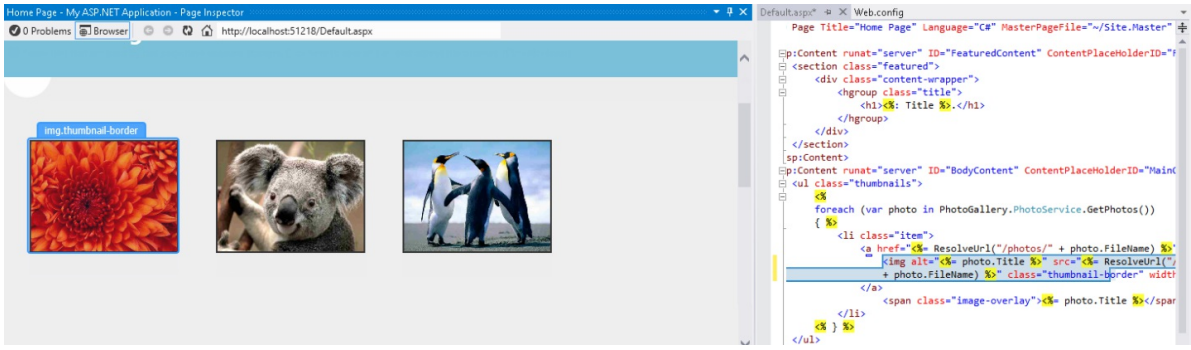
5. Click the **Toggle Inspection Mode** button, located at the left of the tabs.

   This tool will let you select any element of the page and see its HTML code and .aspx source.



Toggle Inspection Mode button

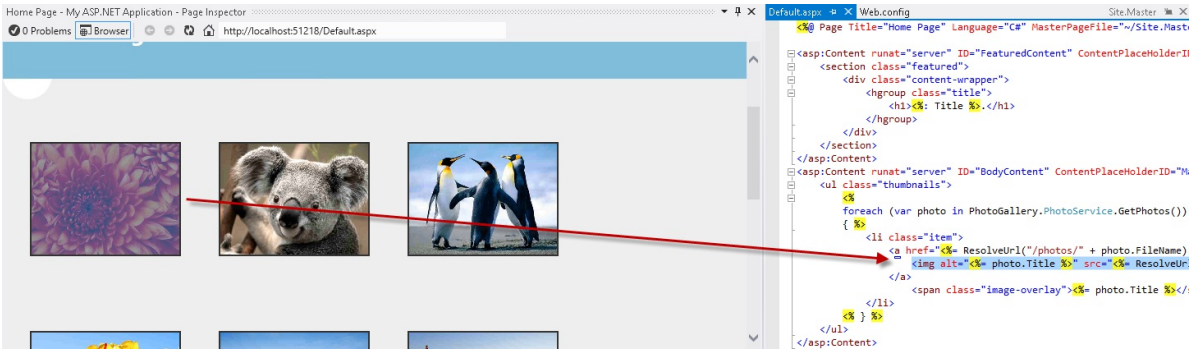
6. In the Page Inspector browser, move the mouse over the page elements. While you move the mouse pointer over any part of the rendered page, the element type is displayed and the corresponding source markup or code is highlighted in the Visual Studio editor.



Inspection mode in action

> **NOTE**
>
> Do not maximize the Page Inspector window or you will not be able to see the preview tab showing the source code. If you click the element in Page Inspector when it is maximized, the source code of the selection will appear but it will hide the Page Inspector window.

If you pay attention to **Default.aspx** file, you will notice that the portion of source code that generates the selected element is highlighted. This feature facilitates the edition of long source files, providing a direct and fast way to access the code.



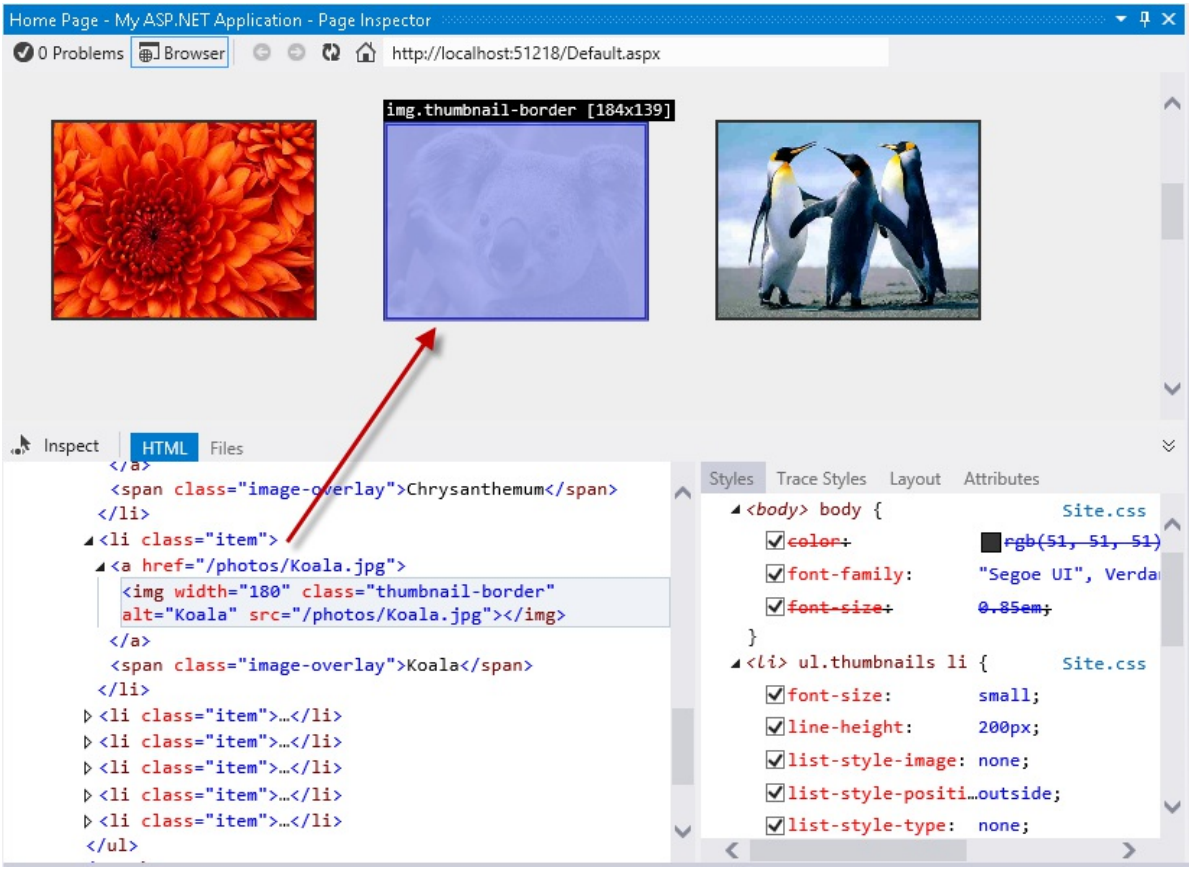
Inspecting elements

7. Click the **Toggle Inspection Mode** button (Inspect), located in Page Inspector tabs, to disable the cursor.

8. Select the **HTML** tab to display the HTML code rendered in the Page Inspector browser.

9. In the HTML code, locate the list item with the Koala link and select it.

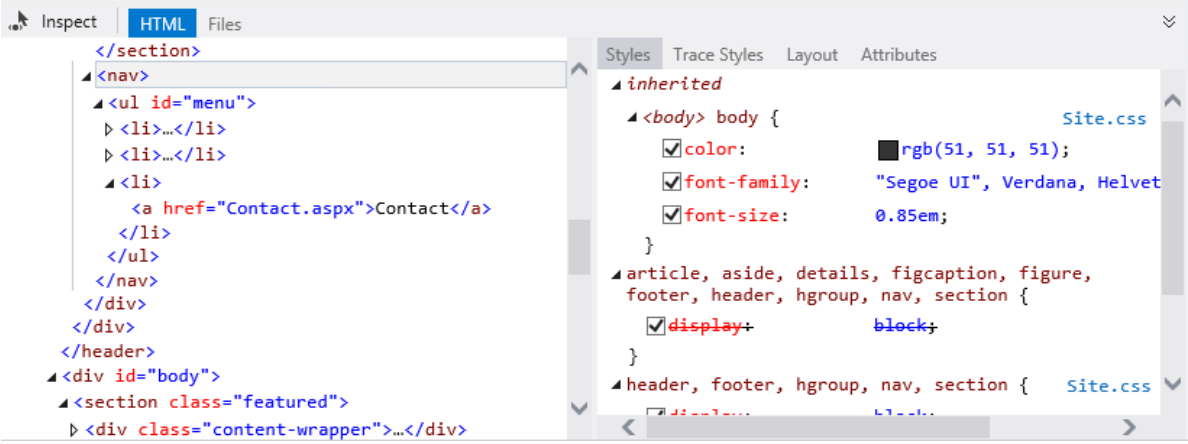
   Notice that when you select the code, the corresponding output is automatically highlighted browser. This feature is useful to see how an HTML block is rendered on the page.



Selecting an HTML element in the page

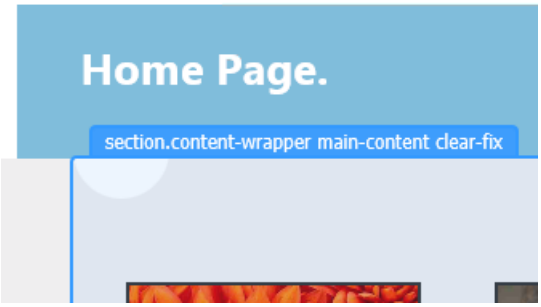
10. Click the **Toggle Inspection Mode** button to enable *Inspection Mode* and click the navigation bar. On the right of the HTML code, in the Styles pane, you will see a list with the CSS styles applied to the selected element.

> **NOTE**
>
> since the header is a part of the site layout, Page Inspector will also open Site.Master file and highlight the segment of code affected.



Discovering styles and source files of a selected element

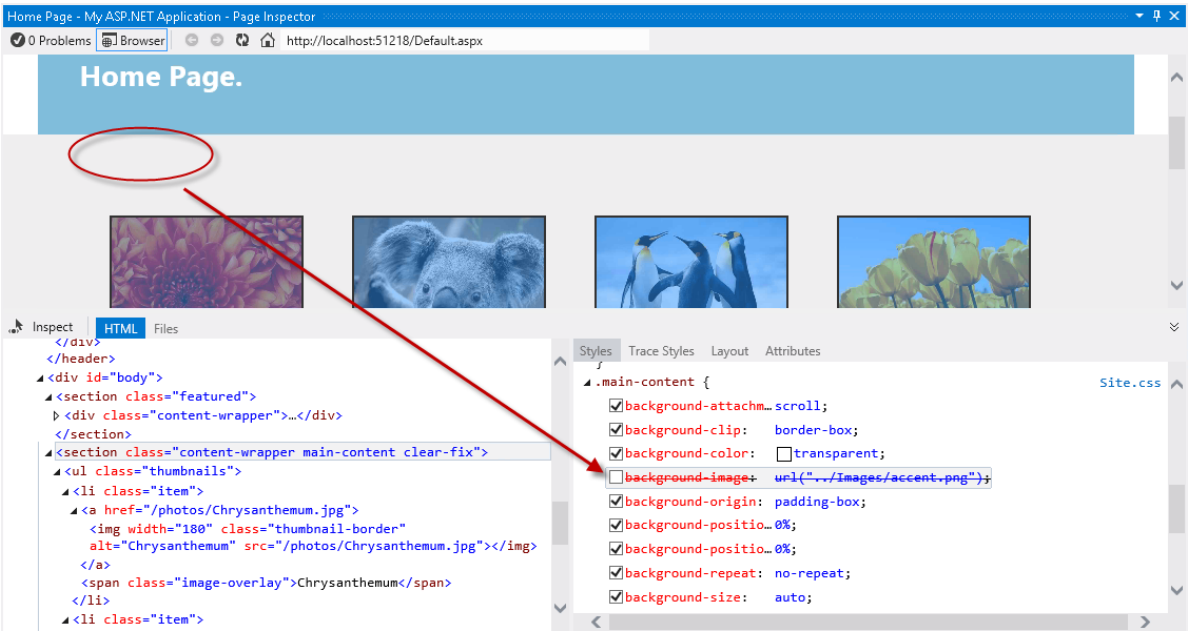
11. With the toggle inspection pointer enabled, move the mouse pointer below the menu bar and click the blank half circle.



Selecting an element

12. In the Styles pane, locate the **background-image** item under the **.main-content** group. **Uncheck** the **background-image** and see what happens. You will notice that the browser will reflect the changes immediately and the circle is hidden.

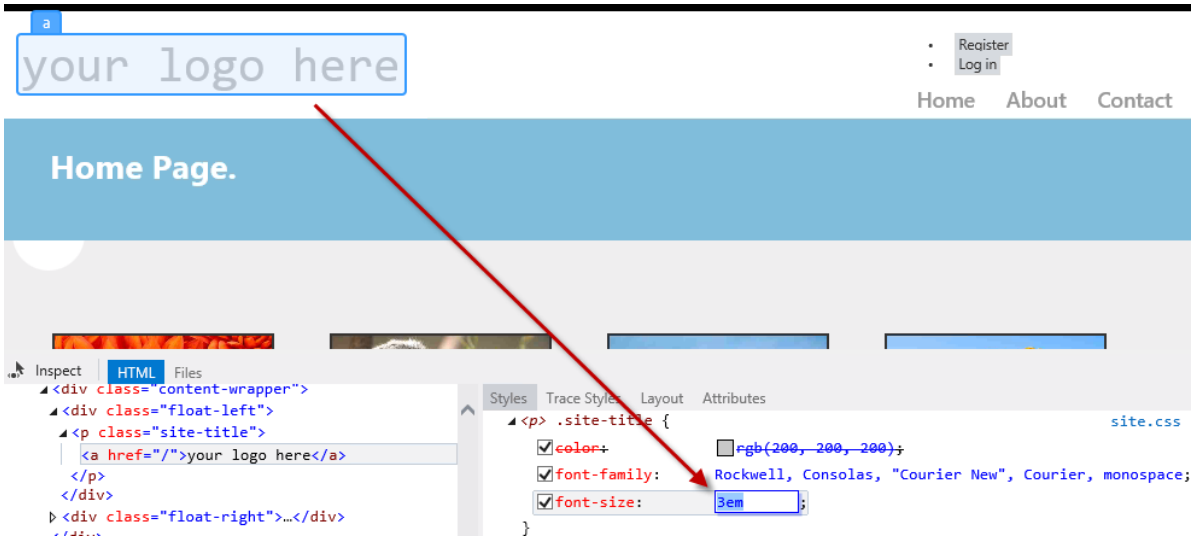
> **NOTE**
>
> The changes you apply on the Page Inspector Styles tab do not affect the original stylesheet. You can uncheck styles or change their values as many times as you want, but they will be restored after refreshing the page.



Enabling and disabling CSS styles

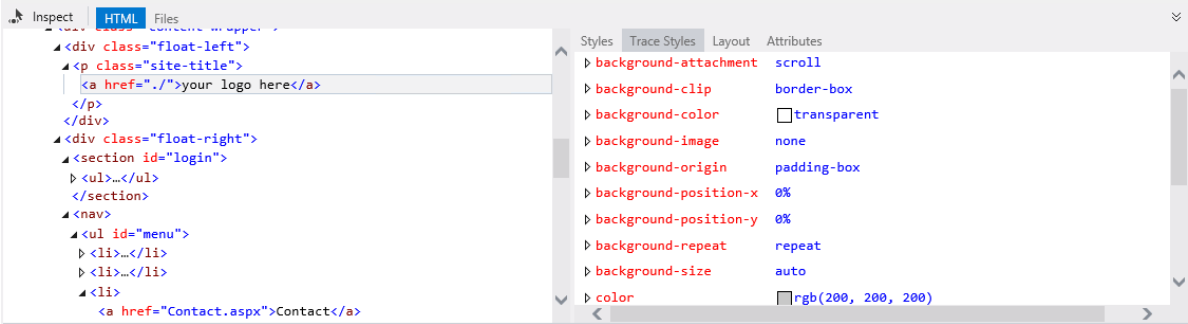
13. Now, click the '**your logo here**' text on the header using the inspection mode.

14. In the **Styles** tab, locate the **font-size** CSS attribute under the **.site-title** group. Double-click the attribute once to edit its value. Replace the 2.3em value with **3em**, and then press ENTER. Notice that the title looks bigger.



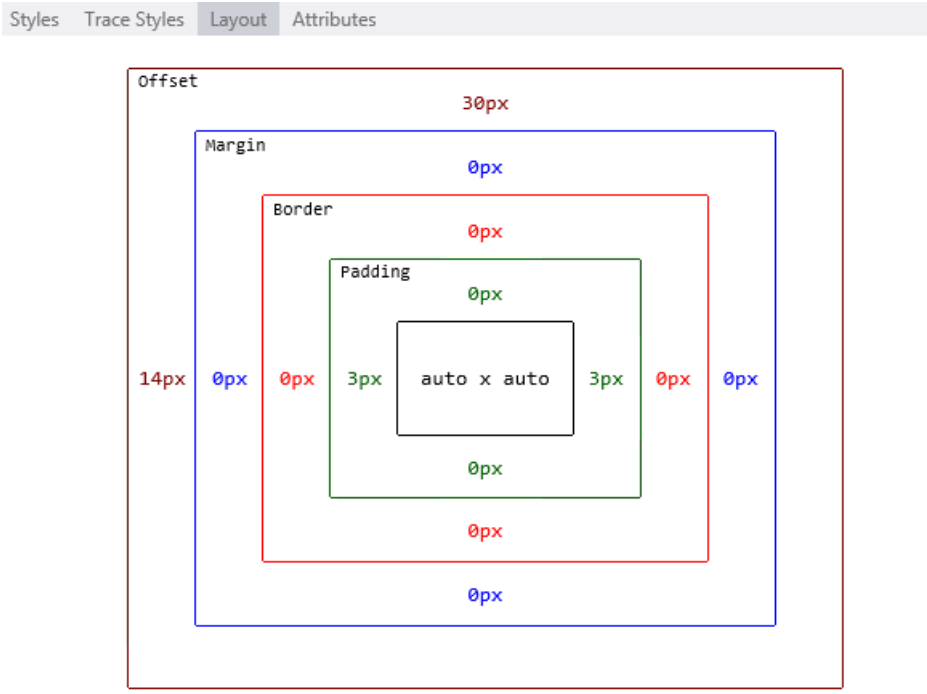
Changing CSS values in the Page Inspector

15. Click the **Trace Styles** tab, located in the right pane of Page Inspector. This is an alternative way to see all the styles applied to the selection, ordered by attribute name.



CSS styles tracing of the selected element

16. Another feature of Page Inspector is the Layout pane. Using the inspection mode, select the navigation bar and then click the **Layout** tab on the right pane. You will see the exact size of the selected element, as well as its offset, margin, padding and border size. Notice that you can also modify the values from this view.



Element layout in Page Inspector

Task 2 - Finding and Fixing Style Issues in the Photo Gallery

How would you diagnose Web pages issues with previous versions of Visual Studio? You are likely familiar with web debugging tools that run outside the Visual Studio IDE, like Internet Explorer Developer Tools or Firebug. Browsers only understand HTML, scripting and styles, while an underlying framework generates the HTML that will be rendered. For that reason, you often need to deploy the whole site to see how web pages look like.

You had probably followed these steps when you wanted to detect and fix an issue in your web site:

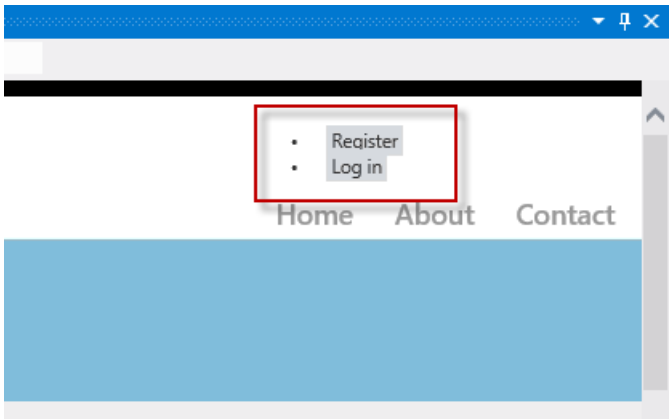
1. Run the Solution from Visual Studio, or deploy the page on the web server.
2. In the browser, open the developer tools you use, or simply open the source code and the styles and try to match the issue. To find the files involved, you'd have used the "Search" or "Search in files" features with the name of the style classes.
3. Once the error is detected, stop the Web browser and the server.
4. Clear the browser cache.
5. Return to Visual Studio to apply a fix. Repeat all the steps to test.

As there is no real WYSIWYG in ASP.NET WebForms, some style issues are detected on a later stage, after running or deploying. Now, with Page Inspector, it is possible to preview any page without running the solution.

In this task, you will use the Page inspector for fixing some issues of the Photo Gallery application. In the following steps, you will detect and quickly fix some simple styling issue in the header.

1. Using Page Inspection, locate the **Register** and the **Log In** links at the left side of the header.

   Notice that the link is not displayed at the expected place on the right. You will now align the link to the right and restyle it accordingly.



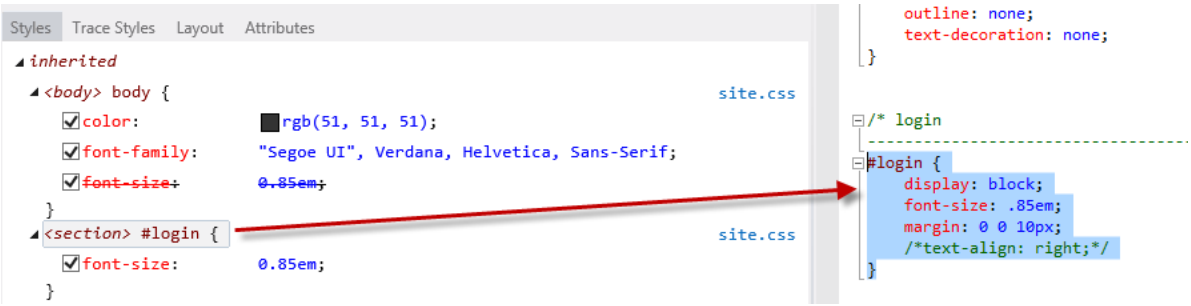
Log In link positioned on the left

2. With Toggle Inspection Mode selected, select the Log In link to open its code.

   Notice that the link source code is located in the **Site.Master** file, not in the Default.aspx page which is the place you might look in first place; you have been placed directly in the correct source file.

3. In the **Styles** tab, locate and click the **<section> #login** item, which is the HTML container for these links.

   Notice that the **#login** style is automatically located in **Site.css** after you click. Moreover, the code is now highlighted.

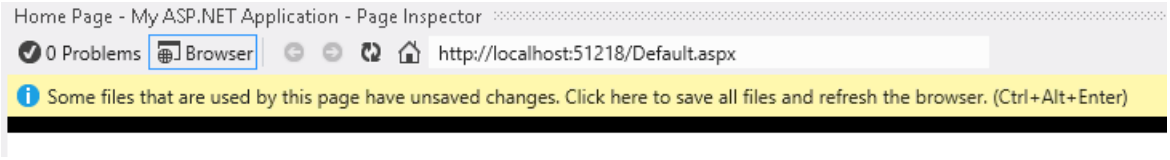


Selecting the CSS styles

4. Uncomment the **text-align** attribute in the highlighted code by removing the opening and closing characters and save the **Site.css** file.

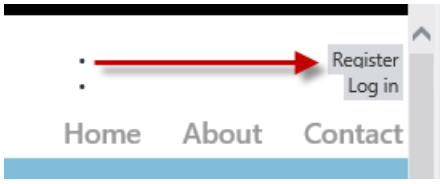
   Page Inspector is aware of all the different files that compose the current page, and it can detect when any of these files change. It alerts you whenever the current page in browser is not in sync with the source files.

5. In the Page Inspector browser, click the bar located below the address bar to save the changes and reload the page.



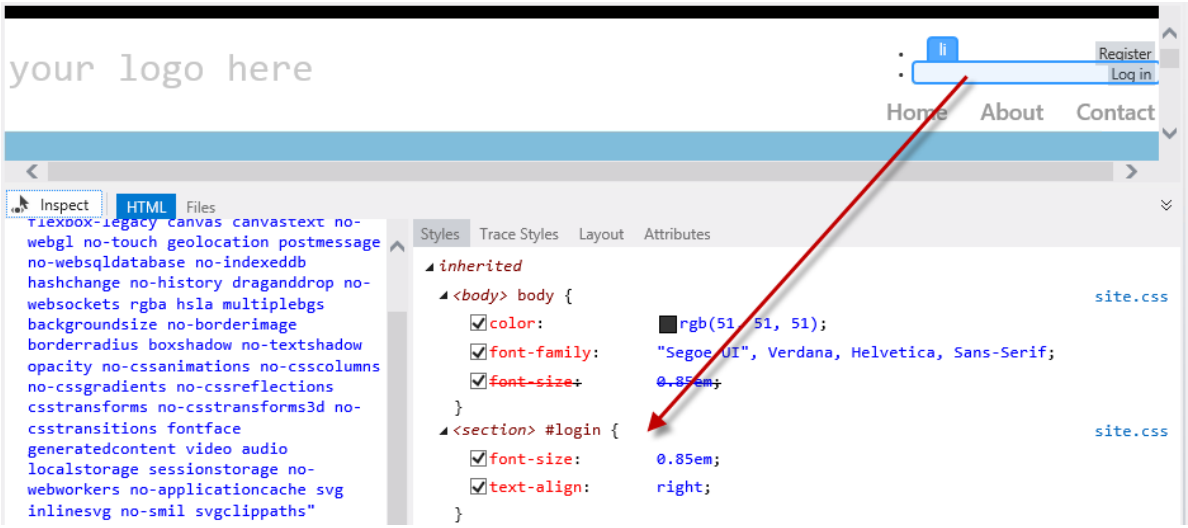
Reloading the page

The links are now at the right, but they still look like a bulleted list. Now, you will remove the bullets and align the links horizontally.



Updated page

6. Using the inspection mode, select any of the **<li>** items that contain the "Register" and "Log in" links. Then, click the **<section> #login** item to access **Styles.css** code.



Finding the style

7. In **Style.css**, uncomment the code for **#login li** items. The style you are adding will hide the bullet and display the items horizontally.

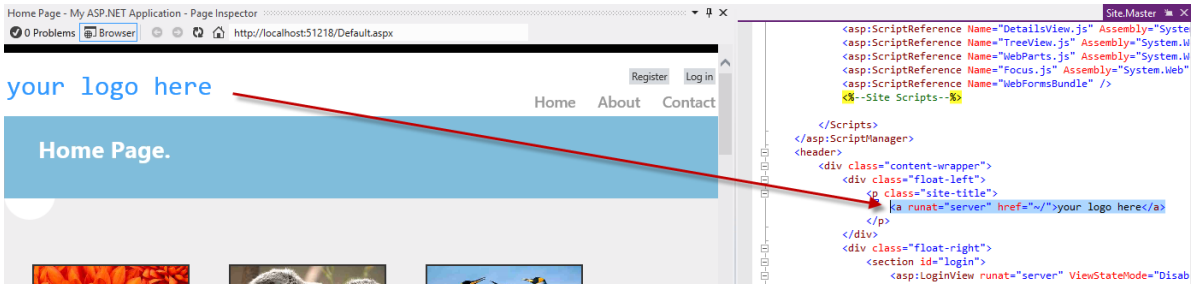
```
#login li
{
    display: inline;
    list-style: none;
}
```

Restyling the login links

8. Save **Style.css** file and click once on the bar located below the address to reload the page. Notice that the links appear correctly.

Links aligned to the right side

9. Finally, you will change the header title. Instead of searching for the '**your logo here**' text in all the files, use the inspection mode to click the text and get to source code that generates it.



Finding the site title

10. Now you are in **Site.Master**, replace the '**your logo here**' text with '**Photo Gallery**'. Save and update the Page Inspector browser.

Photo Gallery page updated

11. Finally press **F5** to run the app the check out all the changes work as expected.

Summary

By completing this Hands-On Lab, you have learnt how to use Page Inspector to preview your Web application without having to rebuild and run the Web site in a browser. In addition, you have learnt how to quickly find and fix bugs by accessing directly from the rendered output to the source code.

Appendix A: Installing Visual Studio Express 2012 for Web

You can install **Microsoft Visual Studio Express 2012 for Web** or another "Express" version using the **Microsoft Web Platform Installer**. The following instructions guide you through the steps required to install *Visual studio Express 2012 for Web* using *Microsoft Web Platform Installer*.

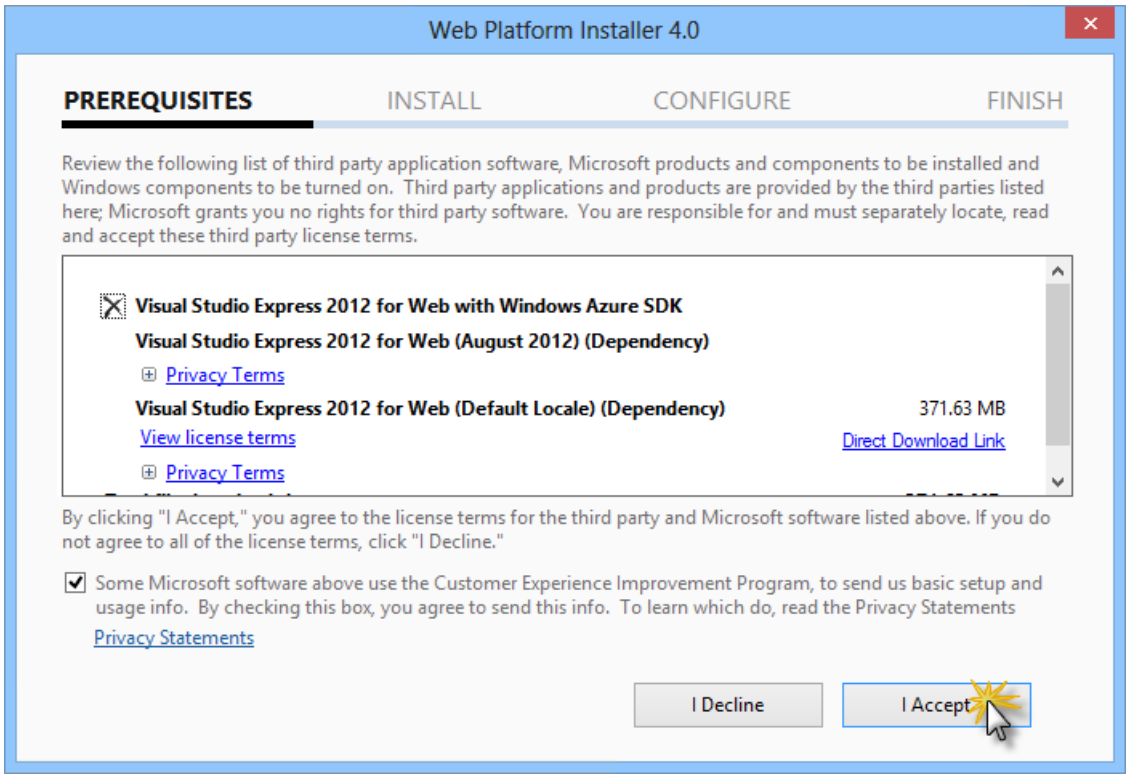
1. Go to https://go.microsoft.com/?linkid=9810169. Alternatively, if you already have installed Web Platform Installer, you can open it and search for the product "*Visual Studio Express 2012 for Web with Windows Azure SDK*".

2. Click on **Install Now**. If you do not have **Web Platform Installer** you will be redirected to download and install it first.

3. Once **Web Platform Installer** is open, click **Install** to start the setup.

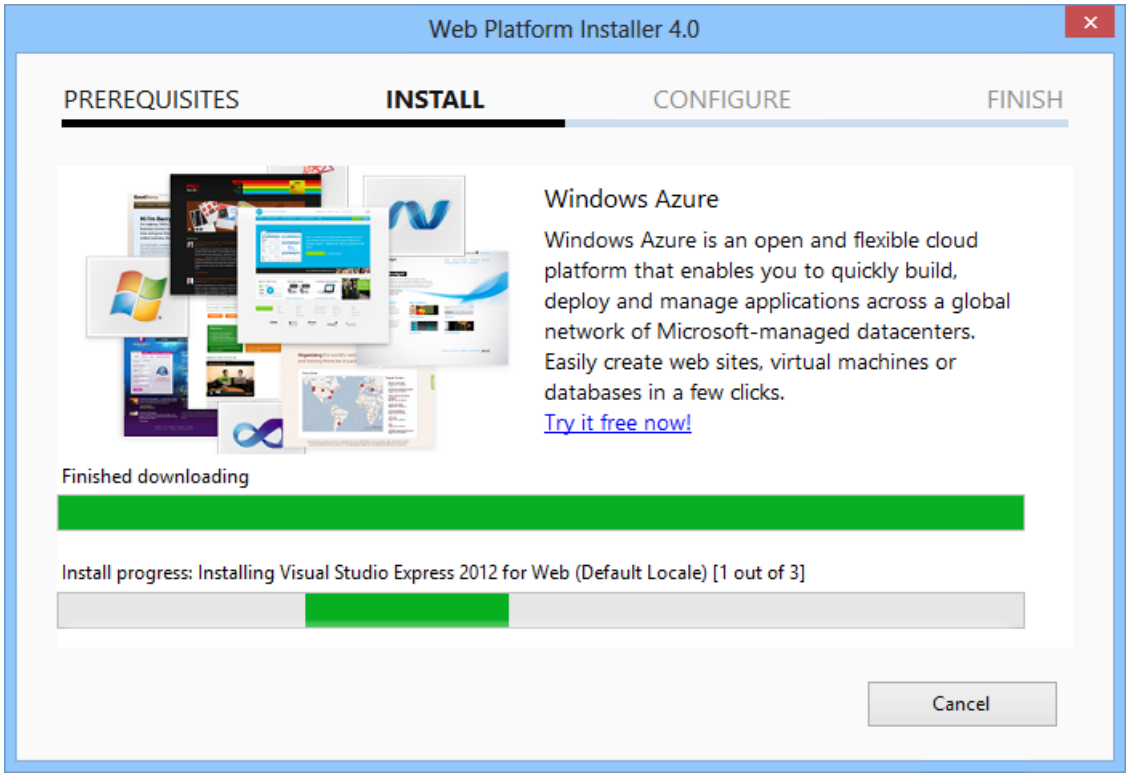
Install Visual Studio Express

4. Read all the products' licenses and terms and click **I Accept** to continue.



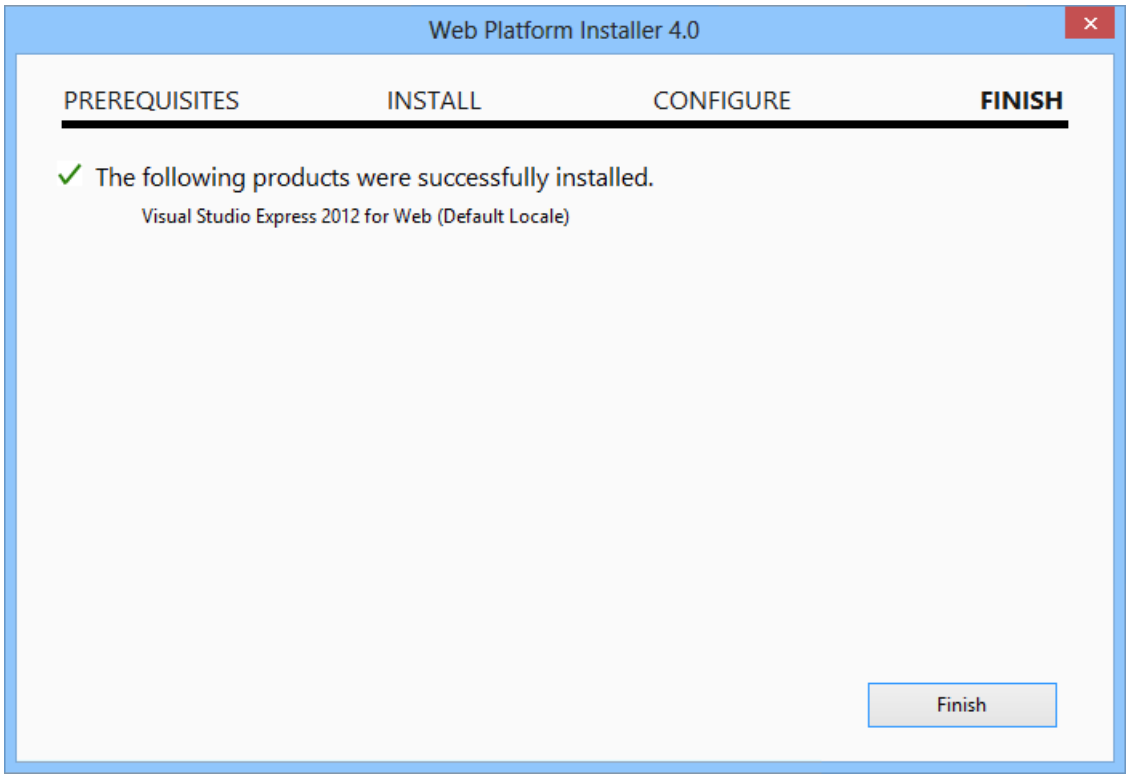
Accepting the license terms

5. Wait until the downloading and installation process completes.



Installation progress

6. When the installation completes, click **Finish**.



Installation completed

7. Click **Exit** to close Web Platform Installer.

8. To open Visual Studio Express for Web, go to the **Start** screen and start writing "**VS Express**", then click on the **VS Express for Web** tile.

VS Express for Web tile

Working with Data in ASP.NET Web Forms

2/20/2020 • 2 minutes to read • Edit Online

How to access data using controls and how to persist information between requests.

- Model Binding and Web Forms in Visual Studio 2013
  - Retrieving and Displaying Data
  - Updating, Deleting, and Creating Data
  - Sorting, Paging, and Filtering Data
  - Integrating JQuery UI Datepicker
  - Using Query String Values to Filter Data
  - Adding Business Logic Layer

Model Binding and Web Forms in Visual Studio 2013

2/20/2020 • 2 minutes to read • Edit Online

How to work with data using Model Binding and Web Forms.

- Retrieving and Displaying Data
- Updating, Deleting, and Creating Data
- Sorting, Paging, and Filtering Data
- Integrating JQuery UI Datepicker
- Using Query String Values to Filter Data
- Adding Business Logic Layer

Retrieving and displaying data with model binding and web forms

9/16/2020 • 10 minutes to read • Edit Online

This tutorial series demonstrates basic aspects of using model binding with an ASP.NET Web Forms project. Model binding makes data interaction more straight-forward than dealing with data source objects (such as ObjectDataSource or SqlDataSource). This series starts with introductory material and moves to more advanced concepts in later tutorials.

The model binding pattern works with any data access technology. In this tutorial, you will use Entity Framework, but you could use the data access technology that is most familiar to you. From a data-bound server control, such as a GridView, ListView, DetailsView, or FormView control, you specify the names of the methods to use for selecting, updating, deleting, and creating data. In this tutorial, you will specify a value for the SelectMethod.

Within that method, you provide the logic for retrieving the data. In the next tutorial, you will set values for UpdateMethod, DeleteMethod and InsertMethod.

You can download the complete project in C# or Visual Basic. The downloadable code works with Visual Studio 2012 and later. It uses the Visual Studio 2012 template, which is slightly different than the Visual Studio 2017 template shown in this tutorial.

In the tutorial you run the application in Visual Studio. You can also deploy the application to a hosting provider and make it available over the internet. Microsoft offers free web hosting for up to 10 web sites in a free Azure trial account. For information about how to deploy a Visual Studio web project to Azure App Service Web Apps, see the ASP.NET Web Deployment using Visual Studio series. That tutorial also shows how to use Entity Framework Code First Migrations to deploy your SQL Server database to Azure SQL Database.

Software versions used in the tutorial

- Microsoft Visual Studio 2017 or Microsoft Visual Studio Community 2017

This tutorial also works with Visual Studio 2012 and Visual Studio 2013, but there are some differences in the user interface and project template.

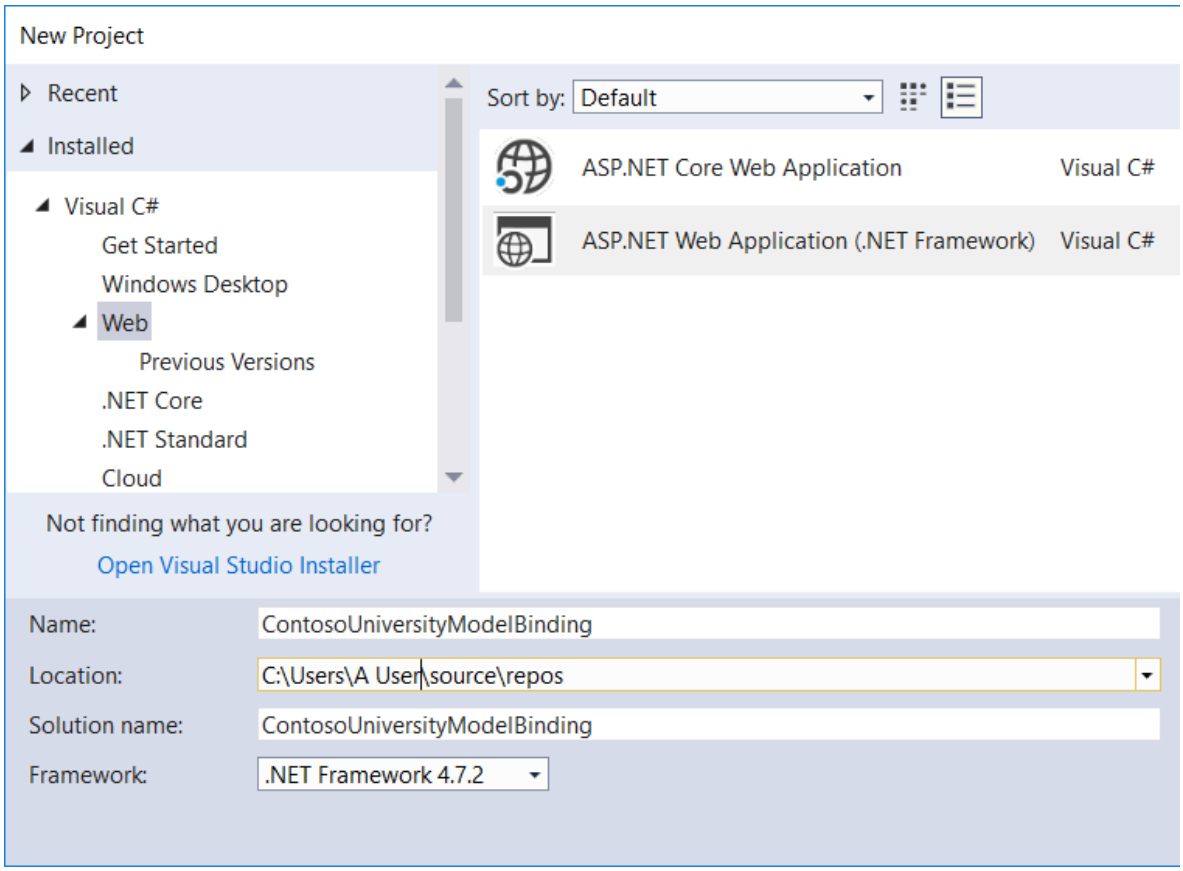
What you'll build

In this tutorial, you'll:

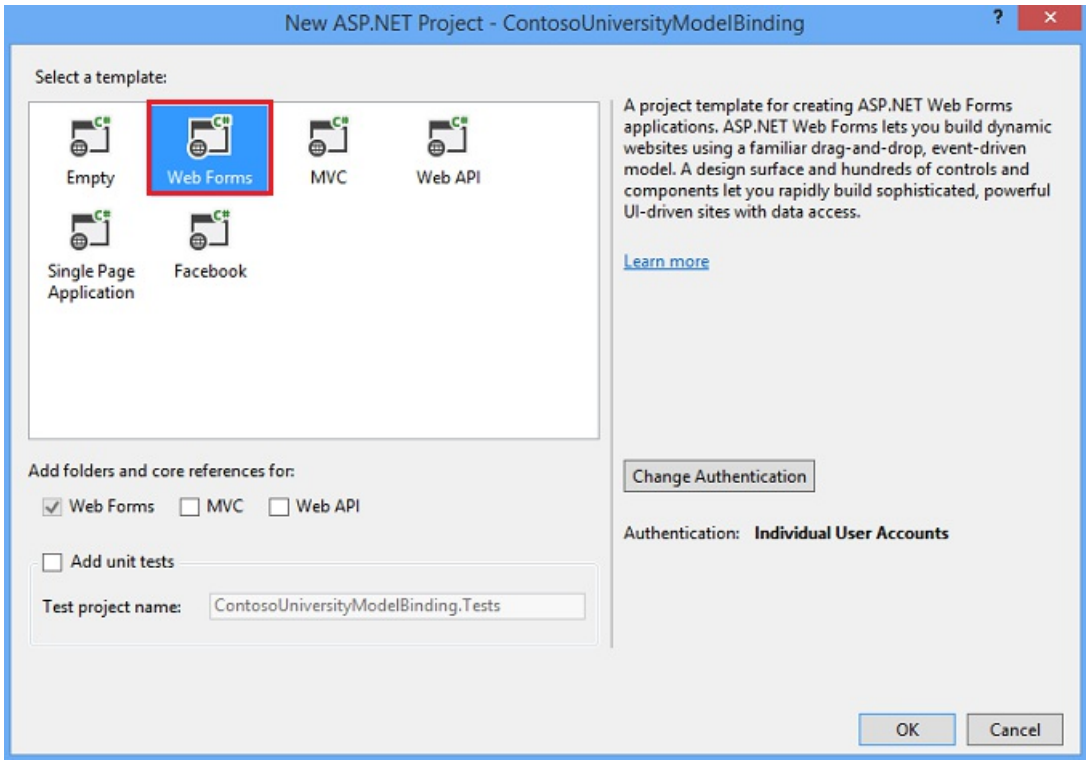
- Build data objects that reflect a university with students enrolled in courses
- Build database tables from the objects
- Populate the database with test data
- Display data in a web form

Create the project

1. In Visual Studio 2017, create a **ASP.NET Web Application (.NET Framework)** project called **ContosoUniversityModelBinding**.



2. Select **OK**. The dialog box to select a template appears.



3. Select the **Web Forms** template.

4. If necessary, change the authentication to **Individual User Accounts**.

5. Select **OK** to create the project.

Modify site appearance

Make a few changes to customize site appearance.

1. Open the Site.Master file.
2. Change the title to display **Contoso University** and not **My ASP.NET Application**.

```
<title><%: Page.Title %> - Contoso University</title>
```

3. Change the header text from **Application name** to **Contoso University**.

```
<div class="navbar-header">
    <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
    </button>
    <a class="navbar-brand" runat="server" href="~/">Contoso University</a>
</div>
```

4. Change the navigation header links to site appropriate ones.

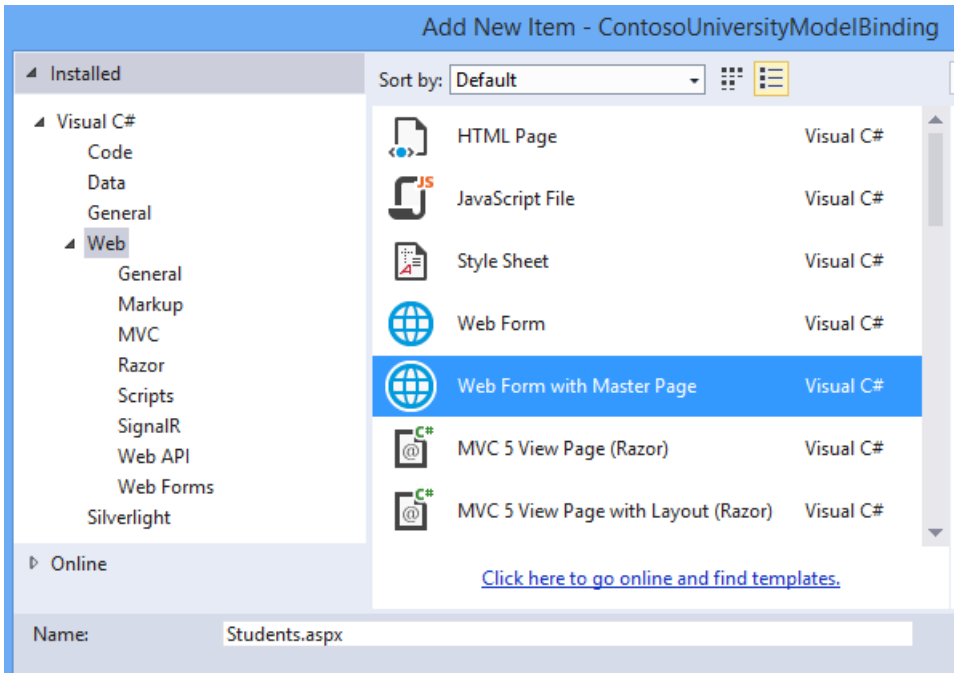
   Remove the links for **About** and **Contact** and, instead, link to a **Students** page, which you will create.

```
<ul class="nav navbar-nav">
    <li><a runat="server" href="~/">Home</a></li>
    <li><a runat="server" href="~/Students">Students</a></li>
</ul>
```

5. Save Site.Master.

Add a web form to display student data

1. In **Solution Explorer**, right-click your project, select **Add** and then **New Item**.
2. In the **Add New Item** dialog box, select the **Web Form with Master Page** template and name it **Students.aspx**.



3. Select **Add**.
4. For the web form's master page, select **Site.Master**.

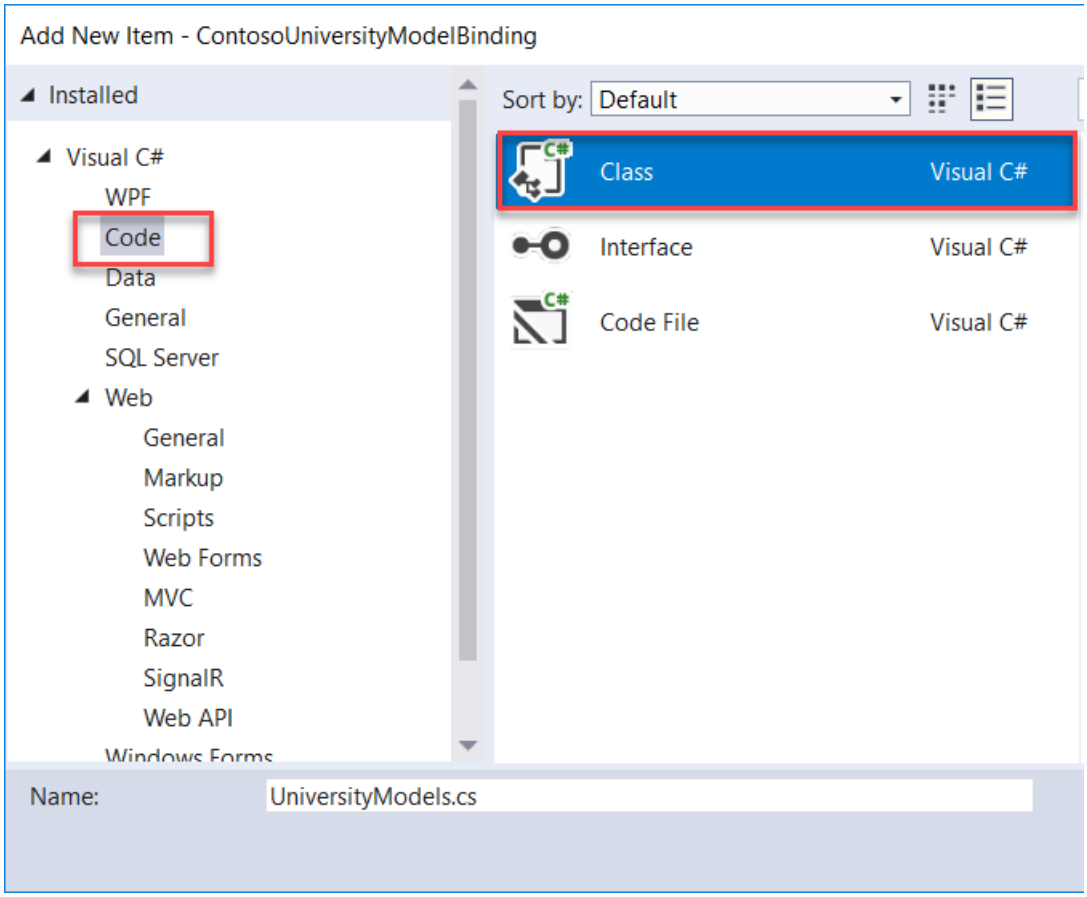
5. Select **OK**.

Add the data model

In the **Models** folder, add a class named **UniversityModels.cs**.

1. Right-click **Models**, select **Add**, and then **New Item**. The **Add New Item** dialog box appears.

2. From the left navigation menu, select **Code**, then **Class**.



3. Name the class **UniversityModels.cs** and select **Add**.

   In this file, define the `SchoolContext`, `Student`, `Enrollment`, and `Course` classes as follows:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversityModelBinding.Models
{
    public class SchoolContext : DbContext
    {
        public DbSet<Student> Students { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Course> Courses { get; set; }
    }

    public class Student
    {
        [Key, Display(Name = "ID")]
        [ScaffoldColumn(false)]
        public int StudentID { get; set; }

        [Required, StringLength(40), Display(Name="Last Name")]
        public string LastName { get; set; }

        [Required, StringLength(20), Display(Name = "First Name")]
        public string FirstName { get; set; }

        [EnumDataType(typeof(AcademicYear)), Display(Name = "Academic Year")]
        public AcademicYear Year { get; set; }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }

    public class Enrollment
    {
        [Key]
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public decimal? Grade { get; set; }
        public virtual Course Course { get; set; }
        public virtual Student Student { get; set; }
    }

    public class Course
    {
        [Key]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }
        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }

    public enum AcademicYear
    {
        Freshman,
        Sophomore,
        Junior,
        Senior
    }
}
```

The `SchoolContext` class derives from `DbContext`, which manages the database connection and changes in the data.

In the `Student` class, notice the attributes applied to the `FirstName`, `LastName`, and `Year` properties. This tutorial uses these attributes for data validation. To simplify the code, only these properties are marked

with data-validation attributes. In a real project, you would apply validation attributes to all properties needing validation.

4. Save UniversityModels.cs.

Set up the database based on classes

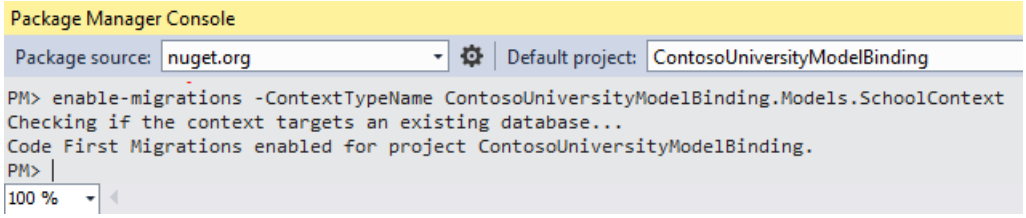
This tutorial uses Code First Migrations to create objects and database tables. These tables store information about the students and their courses.

1. Select **Tools** > **NuGet Package Manager** > **Package Manager Console**.

2. In **Package Manager Console**, run this command:

   `enable-migrations -ContextTypeName ContosoUniversityModelBinding.Models.SchoolContext`

   If the command completes successfully, a message stating migrations have been enabled appears.



   Notice that a file named *Configuration.cs* has been created. The `Configuration` class has a `Seed` method, which can pre-populate the database tables with test data.

Pre-populate the database

1. Open Configuration.cs.

2. Add the following code to the `Seed` method. Also, add a `using` statement for the `ContosoUniversityModelBinding. Models` namespace.

```
namespace ContosoUniversityModelBinding.Migrations
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;
    using ContosoUniversityModelBinding.Models;

    internal sealed class Configuration : DbMigrationsConfiguration<SchoolContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
        }

        protected override void Seed(SchoolContext context)
        {

            context.Students.AddOrUpdate(
                 new Student {
                     FirstName = "Carson",
                     LastName = "Alexander",
                     Year = AcademicYear.Freshman },
                 new Student {
                     FirstName = "Meredith",
                     LastName = "Alonso",
                     Year = AcademicYear.Freshman },
                 new Student {
                     FirstName = "Arturo",
```

```
                FirstName = "Arturo",
                LastName = "Anand",
                Year = AcademicYear.Sophomore },
            new Student {
                FirstName = "Gytis",
                LastName = "Barzdukas",
                Year = AcademicYear.Sophomore },
            new Student {
                FirstName = "Yan",
                LastName = "Li",
                Year = AcademicYear.Junior },
            new Student {
                FirstName = "Peggy",
                LastName = "Justice",
                Year = AcademicYear.Junior },
            new Student {
                FirstName = "Laura",
                LastName = "Norman",
                Year = AcademicYear.Senior },
            new Student {
                FirstName = "Nino",
                LastName = "Olivetto",
                Year = AcademicYear.Senior }
            );

        context.SaveChanges();

        context.Courses.AddOrUpdate(
            new Course { Title = "Chemistry", Credits = 3 },
            new Course { Title = "Microeconomics", Credits = 3 },
            new Course { Title = "Macroeconomics", Credits = 3 },
            new Course { Title = "Calculus", Credits = 4 },
            new Course { Title = "Trigonometry", Credits = 4 },
            new Course { Title = "Composition", Credits = 3 },
            new Course { Title = "Literature", Credits = 4 }
            );

        context.SaveChanges();

        context.Enrollments.AddOrUpdate(
            new Enrollment { StudentID = 1, CourseID = 1, Grade = 1 },
            new Enrollment { StudentID = 1, CourseID = 2, Grade = 3 },
            new Enrollment { StudentID = 1, CourseID = 3, Grade = 1 },
            new Enrollment { StudentID = 2, CourseID = 4, Grade = 2 },
            new Enrollment { StudentID = 2, CourseID = 5, Grade = 4 },
            new Enrollment { StudentID = 2, CourseID = 6, Grade = 4 },
            new Enrollment { StudentID = 3, CourseID = 1 },
            new Enrollment { StudentID = 4, CourseID = 1 },
            new Enrollment { StudentID = 4, CourseID = 2, Grade = 4 },
            new Enrollment { StudentID = 5, CourseID = 3, Grade = 3 },
            new Enrollment { StudentID = 6, CourseID = 4 },
            new Enrollment { StudentID = 7, CourseID = 5, Grade = 2 }
            );

        context.SaveChanges();
    }
  }
}
```

3. Save Configuration.cs.
4. In the Package Manager Console, run the command **add-migration initial**.
5. Run the command **update-database**.

   If you receive an exception when running this command, the `StudentID` and `CourseID` values might be different from the `Seed` method values. Open those database tables and find existing values for

`StudentID` and `CourseID`. Add those values to the code for seeding the `Enrollments` table.

Add a GridView control

With populated database data, you're now ready to retrieve that data and display it.

1. Open Students.aspx.

2. Locate the `MainContent` placeholder. Within that placeholder, add a **GridView** control that includes this code.

```
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <asp:GridView runat="server" ID="studentsGrid"
        ItemType="ContosoUniversityModelBinding.Models.Student" DataKeyNames="StudentID"
        SelectMethod="studentsGrid_GetData"
        AutoGenerateColumns="false">
        <Columns>
            <asp:DynamicField DataField="StudentID" />
            <asp:DynamicField DataField="LastName" />
            <asp:DynamicField DataField="FirstName" />
            <asp:DynamicField DataField="Year" />
            <asp:TemplateField HeaderText="Total Credits">
              <ItemTemplate>
                <asp:Label Text="<%# Item.Enrollments.Sum(en => en.Course.Credits) %>"
                    runat="server" />
              </ItemTemplate>
            </asp:TemplateField>
        </Columns>
    </asp:GridView>
</asp:Content>
```

   Things to note:

   - Notice the value set for the `SelectMethod` property in the GridView element. This value specifies the method used to retrieve GridView data, which you create in the next step.

   - The `ItemType` property is set to the `Student` class created earlier. This setting allows you to reference class properties in the markup. For example, the `Student` class has a collection named `Enrollments`. You can use `Item.Enrollments` to retrieve that collection and then use LINQ syntax to retrieve each student's enrolled credits sum.

3. Save Students.aspx.

Add code to retrieve data

In the Students.aspx code-behind file, add the method specified for the `SelectMethod` value.

1. Open Students.aspx.cs.

2. Add `using` statements for the `ContosoUniversityModelBinding. Models` and `System.Data.Entity` namespaces.

```
using ContosoUniversityModelBinding.Models;
using System.Data.Entity;
```

3. Add the method you specified for `SelectMethod`:

```
public IQueryable<Student> studentsGrid_GetData()
{
    SchoolContext db = new SchoolContext();
    var query = db.Students.Include(s => s.Enrollments.Select(e => e.Course));
    return query;
}
```

The `Include` clause improves query performance but isn't required. Without the `Include` clause, the data is retrieved using *lazy loading*, which involves sending a separate query to the database each time related data is retrieved. With the `Include` clause, data is retrieved using *eager loading*, which means a single database query retrieves all related data. If related data isn't used, eager loading is less efficient because more data is retrieved. However, in this case, eager loading gives you the best performance because the related data is displayed for each record.

For more information about performance considerations when loading related data, see the **Lazy, Eager, and Explicit Loading of Related Data** section in the Reading Related Data with the Entity Framework in an ASP.NET MVC Application article.

By default, the data is sorted by the values of the property marked as the key. You can add an `OrderBy` clause to specify a different sort value. In this example, the default `StudentID` property is used for sorting. In the Sorting, Paging, and Filtering Data article, the user is enabled to select a column for sorting.

4. Save Students.aspx.cs.

Run your application

Run your web application (**F5**) and navigate to the **Students** page, which displays the following:

Contoso University Home Students

ID	Last Name	First Name	Academic Year	Total Credits
1	Alexander	Carson	Freshman	9
2	Alonso	Meredith	Freshman	11
3	Anand	Arturo	Sophomore	3
4	Barzdukas	Gytis	Sophomore	6
5	Li	Yan	Junior	3
6	Justice	Peggy	Junior	4
7	Norman	Laura	Senior	4
8	Olivetto	Nino	Senior	0

Automatic generation of model binding methods

When working through this tutorial series, you can simply copy the code from the tutorial to your project. However, one disadvantage of this approach is that you may not become aware of the feature provided by Visual Studio to automatically generate code for model binding methods. When working on your own projects, automatic code generation can save you time and help you gain a sense of how to implement an operation. This section describes the automatic code generation feature. This section is only informational and does not contain any code you need to implement in your project.

When setting a value for the `SelectMethod`, `UpdateMethod`, `InsertMethod`, or `DeleteMethod` properties in the markup code, you can select the **Create New Method** option.

```
<asp:GridView runat="server" ID="studentsGrid"
    ItemType="ContosoUniversityModelBinding.Models.Student" DataKeyNames="StudentID"
    SelectMethod="studentsGrid_GetData"
    UpdateMethod="|"
    AutoGenerateC  <Create New Method>
    <Columns>
```

Visual Studio not only creates a method in the code-behind with the proper signature, but also generates implementation code to perform the operation. If you first set the `ItemType` property before using the automatic code generation feature, the generated code uses that type for the operations. For example, when setting the `UpdateMethod` property, the following code is automatically generated:

```
// The id parameter name should match the DataKeyNames value set on the control
public void studentsGrid_UpdateItem(int id)
{
    ContosoUniversityModelBinding.Models.Student item = null;
    // Load the item here, e.g. item = MyDataLayer.Find(id);
    if (item == null)
    {
        // The item wasn't found
        ModelState.AddModelError("", String.Format("Item with id {0} was not found", id));
        return;
    }
    TryUpdateModel(item);
    if (ModelState.IsValid)
    {
        // Save changes here, e.g. MyDataLayer.SaveChanges();

    }
}
```

Again, this code doesn't need to be added to your project. In the next tutorial, you'll implement methods for updating, deleting, and adding new data.

Summary

In this tutorial, you created data model classes and generated a database from those classes. You filled the database tables with test data. You used model binding to retrieve data from the database, and then displayed the data in a GridView.

In the next tutorial in this series, you'll enable updating, deleting, and creating data.

NEXT

Updating, deleting, and creating data with model binding and web forms

2/20/2020 • 5 minutes to read • Edit Online

by Tom FitzMacken

This tutorial series demonstrates basic aspects of using model binding with an ASP.NET Web Forms project. Model binding makes data interaction more straight-forward than dealing with data source objects (such as ObjectDataSource or SqlDataSource). This series starts with introductory material and moves to more advanced concepts in later tutorials.

This tutorial shows how to create, update, and delete data with model binding. You will set the following properties:

- DeleteMethod
- InsertMethod
- UpdateMethod

These properties receive the name of the method that handles the corresponding operation. Within that method, you provide the logic for interacting with the data.

This tutorial builds on the project created in the first part of the series.

You can download the complete project in C# or VB. The downloadable code works with either Visual Studio 2012 or Visual Studio 2013. It uses the Visual Studio 2012 template, which is slightly different than the Visual Studio 2013 template shown in this tutorial.

What you'll build

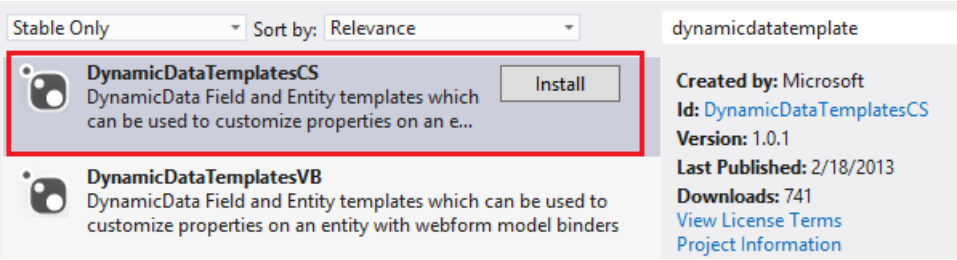
In this tutorial, you'll:

1. Add dynamic data templates
2. Enable updating and deleting data through model binding methods
3. Apply data validation rules - Enable creating a new record in the database

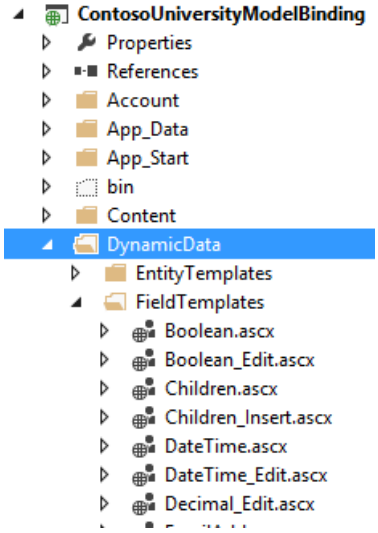
Add dynamic data templates

To provide the best user experience and minimize code repetition, you will use dynamic data templates. You can easily integrate pre-built dynamic data templates into your existing site by installing a NuGet package.

From the **Manage NuGet Packages**, install the **DynamicDataTemplatesCS**.



Notice that your project now includes a folder named **DynamicData**. In that folder, you will find the templates that are automatically applied to dynamic controls in your web forms.



Enable updating and deleting

Enabling users to update and delete records in the database is very similar to the process for retrieving data. In the **UpdateMethod** and **DeleteMethod** properties, you specify the names of the methods that perform those operations. With a GridView control, you can also specify the automatic generation of edit and delete buttons. The following highlighted code shows the additions to the GridView code.

```
<asp:GridView runat="server" ID="studentsGrid"
    ItemType="ContosoUniversityModelBinding.Models.Student" DataKeyNames="StudentID"
    SelectMethod="studentsGrid_GetData"
    UpdateMethod="studentsGrid_UpdateItem" DeleteMethod="studentsGrid_DeleteItem"
    AutoGenerateEditButton="true" AutoGenerateDeleteButton="true"
    AutoGenerateColumns="false">
```

In the code-behind file, add a using statement for **System.Data.Entity.Infrastructure**.

```
using System.Data.Entity.Infrastructure;
```

Then, add the following update and delete methods.

```
public void studentsGrid_UpdateItem(int studentID)
{
    using (SchoolContext db = new SchoolContext())
    {
        Student item = null;
        item = db.Students.Find(studentID);
        if (item == null)
        {
            ModelState.AddModelError("",
              String.Format("Item with id {0} was not found", studentID));
            return;
        }

        TryUpdateModel(item);
        if (ModelState.IsValid)
        {
            db.SaveChanges();
        }
    }
}

public void studentsGrid_DeleteItem(int studentID)
{
    using (SchoolContext db = new SchoolContext())
    {
        var item = new Student { StudentID = studentID };
        db.Entry(item).State = EntityState.Deleted;
        try
        {
            db.SaveChanges();
        }
        catch (DbUpdateConcurrencyException)
        {
            ModelState.AddModelError("",
              String.Format("Item with id {0} no longer exists in the database.", studentID));
        }
    }
}
```

The **TryUpdateModel** method applies the matching data-bound values from the web form to the data item. The data item is retrieved based on the value of the id parameter.

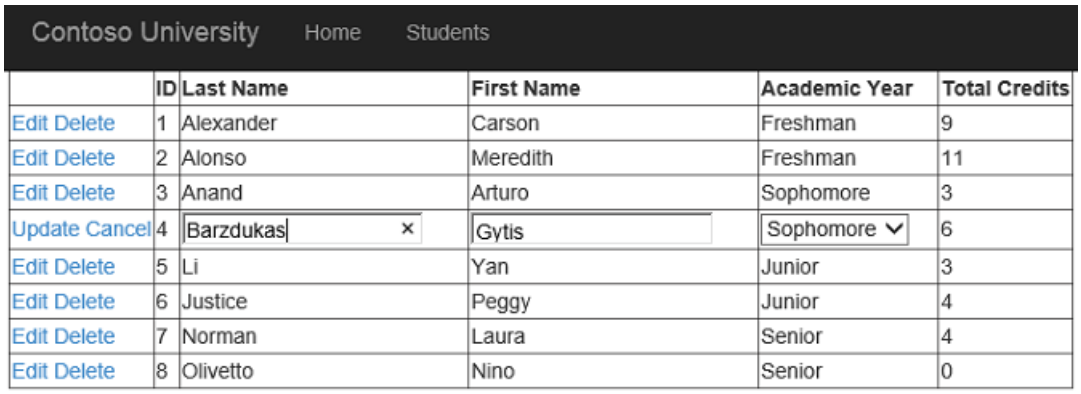
Enforce validation requirements

The validation attributes that you applied to the FirstName, LastName, and Year properties in the Student class are automatically enforced when updating the data. The DynamicField controls add client and server validators based on the validation attributes. The FirstName and LastName properties are both required. FirstName cannot exceed 20 characters in length, and LastName cannot exceed 40 characters. Year must be a valid value for the AcademicYear enumeration.

If the user violates one of the validation requirements, the update does not proceed. To see the error message, add a ValidationSummary control above the GridView. To display the validation errors from model binding, set the **ShowModelStateErrors** property set to **true**.

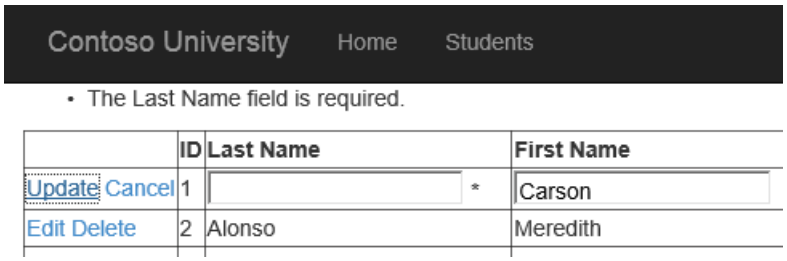
```
<asp:ValidationSummary ShowModelStateErrors="true" runat="server" />
```

Run the web application, and update and delete any of the records.



Notice that when in the edit mode the value for the Year property is automatically rendered as a drop down list. The Year property is an enumeration value, and the dynamic data template for an enumeration value specifies a drop down list for editing. You can find that template by opening the **Enumeration_Edit.ascx** file in the **DynamicData/FieldTemplates** folder.

If you provide valid values, the update completes successfully. If you violate one of the validation requirements, the update does not proceed and an error message is displayed above the grid.



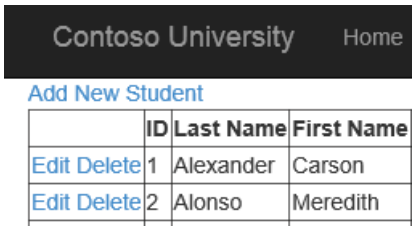
Add new records

The GridView control does not include the **InsertMethod** property and therefore cannot be used for adding a new record with model binding. You can find the InsertMethod property in the **FormView**, **DetailsView**, or **ListView** controls. In this tutorial, you will use a FormView control to add a new record.

First, add a link to the new page you will create for adding a new record. Above the ValidationSummary, add:

```
<asp:HyperLink NavigateUrl="~/AddStudent" Text="Add New Student" runat="server" />
```

The new link will appear at the top of the content for the Students page.



Then, add a new web form using a master page, and name it **AddStudent**. Select Site.Master as the master page.

You will render the fields for adding a new student by using a **DynamicEntity** control. The DynamicEntity control renders that editable properties in the class specified in the ItemType property. The StudentID property was marked with the **[ScaffoldColumn(false)]** attribute so it is not rendered. In the MainContent placeholder of the AddStudent page, add the following code.

```
<asp:ValidationSummary runat="server" ShowModelStateErrors="true" />
<asp:FormView runat="server" ID="addStudentForm"
    ItemType="ContosoUniversityModelBinding.Models.Student"
    InsertMethod="addStudentForm_InsertItem" DefaultMode="Insert"
    RenderOuterTable="false" OnItemInserted="addStudentForm_ItemInserted">
    <InsertItemTemplate>
        <fieldset>
            <ol>
                <asp:DynamicEntity runat="server" Mode="Insert" />
            </ol>
            <asp:Button runat="server" Text="Insert" CommandName="Insert" />
            <asp:Button runat="server" Text="Cancel" CausesValidation="false" OnClick="cancelButton_Click" />
        </fieldset>
    </InsertItemTemplate>
</asp:FormView>
```

In the code-behind file (AddStudent.aspx.cs), add a **using** statement for the **ContosoUniversityModelBinding.Models** namespace.

```
using ContosoUniversityModelBinding.Models;
```

Then, add the following methods to specify how to insert a new record and an event handler for the cancel button.

```
public void addStudentForm_InsertItem()
{
    var item = new Student();

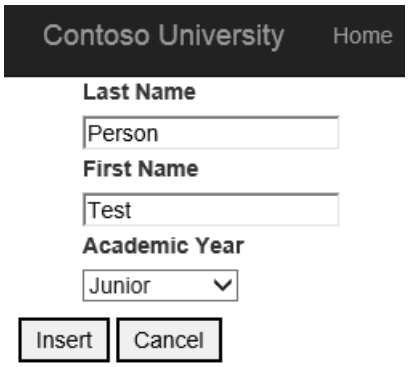
    TryUpdateModel(item);
    if (ModelState.IsValid)
    {
        using (SchoolContext db = new SchoolContext())
        {
            db.Students.Add(item);
            db.SaveChanges();
        }
    }
}

protected void cancelButton_Click(object sender, EventArgs e)
{
    Response.Redirect("~/Students");
}

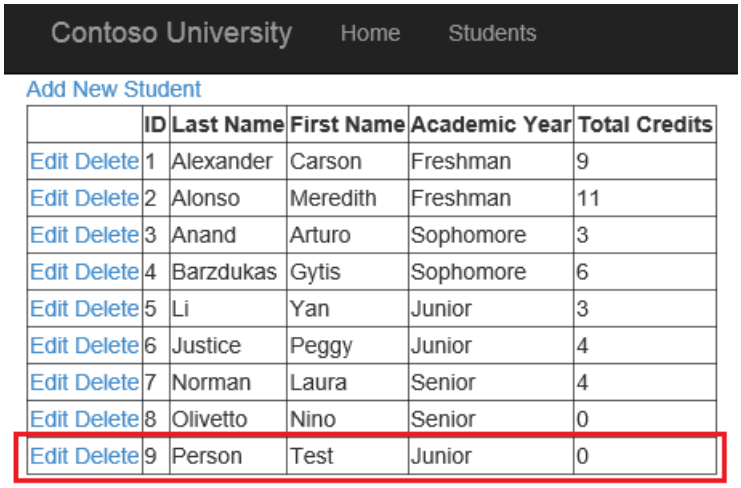
protected void addStudentForm_ItemInserted(object sender, FormViewInsertedEventArgs e)
{
    Response.Redirect("~/Students");
}
```

Save all of the changes.

Run the web application and create a new student.



Click **Insert** and notice the new student has been created.



	ID	Last Name	First Name	Academic Year	Total Credits
Edit Delete	1	Alexander	Carson	Freshman	9
Edit Delete	2	Alonso	Meredith	Freshman	11
Edit Delete	3	Anand	Arturo	Sophomore	3
Edit Delete	4	Barzdukas	Gytis	Sophomore	6
Edit Delete	5	Li	Yan	Junior	3
Edit Delete	6	Justice	Peggy	Junior	4
Edit Delete	7	Norman	Laura	Senior	4
Edit Delete	8	Olivetto	Nino	Senior	0
Edit Delete	9	Person	Test	Junior	0

Conclusion

In this tutorial, you enabled updating, deleting, and creating data. You ensured validation rules are applied when interacting with the data.

In the next tutorial in this series, you will enable sorting, paging, and filtering data.

PREVIOUS NEXT

Sorting, paging, and filtering data with model binding and web forms

2/20/2020 • 3 minutes to read • Edit Online

by Tom FitzMacken

This tutorial series demonstrates basic aspects of using model binding with an ASP.NET Web Forms project. Model binding makes data interaction more straight-forward than dealing with data source objects (such as ObjectDataSource or SqlDataSource). This series starts with introductory material and moves to more advanced concepts in later tutorials.

This tutorial shows how to add sorting, paging, and filtering of the data through model binding.

This tutorial builds on the project created in the first part of the series.

You can download the complete project in C# or VB. The downloadable code works with either Visual Studio 2012 or Visual Studio 2013. It uses the Visual Studio 2012 template, which is slightly different than the Visual Studio 2013 template shown in this tutorial.

What you'll build

In this tutorial, you'll:

1. Enable sorting and paging of the data
2. Enable filtering of the data based on a selection by the user

Add sorting

Enabling sorting in the GridView is very easy. In the Student.aspx file, simply set **AllowSorting** to **true** in the GridView. You do not need to set a **SortExpression** value for each column as the DataField is automatically used. The GridView modifies the query to include ordering the data by the selected value. The highlighted code below shows the addition you need to make to enable sorting.

```
<asp:GridView runat="server" ID="studentsGrid"
    ItemType="ContosoUniversity.Models.Student" DataKeyNames="StudentID"
    SelectMethod="studentsGrid_GetData"
    UpdateMethod="studentsGrid_UpdateItem" DeleteMethod="studentsGrid_DeleteItem"
    AllowSorting="true"
    AutoGenerateEditButton="true" AutoGenerateDeleteButton="true"
    AutoGenerateColumns="false">
  <Columns>
    <asp:DynamicField DataField="StudentID" />
    <asp:DynamicField DataField="LastName" />
    <asp:DynamicField DataField="FirstName" />
    <asp:DynamicField DataField="Year" />
    <asp:TemplateField HeaderText="Total Credits">
      <ItemTemplate>
        <asp:Label Text="<%# Item.Enrollments.Sum(en => en.Course.Credits) %>"
            runat="server" />
      </ItemTemplate>
    </asp:TemplateField>
  </Columns>
</asp:GridView>
```

Run the web application, and test sorting student records by the values in different columns.

Contoso University Home Students

Add New Student

	ID	Last Name	First Name	Academic Year	Total Credits
Edit Delete	3	Anand	Arturo	Sophomore	3
Edit Delete	1	Alexander	Carson	Freshman	9
Edit Delete	4	Barzdukas	Gytis	Sophomore	6
Edit Delete	7	Norman	Laura	Senior	4
Edit Delete	2	Alonso	Meredith	Freshman	11
Edit Delete	8	Olivetto	Nino	Senior	0
Edit Delete	6	Justice	Peggy	Junior	4
Edit Delete	9	Person	Test	Junior	0
Edit Delete	5	Li	Yan	Junior	3

Add paging

Enabling paging is also very easy. In the GridView, set the **AllowPaging** property to **true** and set the **PageSize** property to the number of records you wish to display on each page. In this tutorial, you can set it to 4.

```
<asp:GridView runat="server" ID="studentsGrid"
    ItemType="ContosoUniversity.Models.Student" DataKeyNames="StudentID"
    SelectMethod="studentsGrid_GetData"
    UpdateMethod="studentsGrid_UpdateItem" DeleteMethod="studentsGrid_DeleteItem"
    AllowSorting="true" AllowPaging="true" PageSize="4"
    AutoGenerateEditButton="true" AutoGenerateDeleteButton="true"
    AutoGenerateColumns="false">
```

Run the web application, and notice that now the records are divided over multiple pages with no more than 4 records displayed on a single page.

Contoso University Home Students

Add New Student

	ID	Last Name	First Name	Academic Year	Total Credits
Edit Delete	1	Alexander	Carson	Freshman	9
Edit Delete	2	Alonso	Meredith	Freshman	11
Edit Delete	3	Anand	Arturo	Sophomore	3
Edit Delete	4	Barzdukas	Gytis	Sophomore	6
1 2 3					

Deferred query execution improves the application efficiency. Instead of retrieving the entire data set, the GridView modifies the query to retrieve only the records for the current page.

Filter records by user selection

Model binding adds several attributes which enable you to designate how to set the value for a parameter in a model binding method. These attributes are in the **System.Web.ModelBinding** namespace. They include:

- Control
- Cookie
- Form
- Profile
- QueryString
- RouteData

- Session
- UserProfile
- ViewState

In this tutorial, you will use a control's value to filter which records are displayed in the GridView. You will add the **Control** attribute to the query method you had created earlier. In a later tutorial, you will apply the **QueryString** attribute to a parameter to specify that the parameter value comes from a query string value.

First, above the ValidationSummary, add a drop down list for filtering which students are shown.

```
<asp:HyperLink runat="server" NavigateUrl="~/AddStudent" Text="Add New Student" />

<br /><br />
<asp:Label runat="server" Text="Show:" />
<asp:DropDownList runat="server" AutoPostBack="true" ID="DisplayYear">
    <asp:ListItem Text="All" Value="" />
    <asp:ListItem Text="Freshman" />
    <asp:ListItem Text="Sophomore" />
    <asp:ListItem Text="Junior" />
    <asp:ListItem Text="Senior" />
</asp:DropDownList>

<asp:ValidationSummary runat="server" ShowModelStateErrors="true"/>
```

In the code-behind file, modify the select method to receive a value from the control, and set the name of the parameter to the name of the control that provides the value.

You must add a **using** statement for the **System.Web.ModelBinding** namespace to resolve the Control attribute.

```
using System.Web.ModelBinding;
```

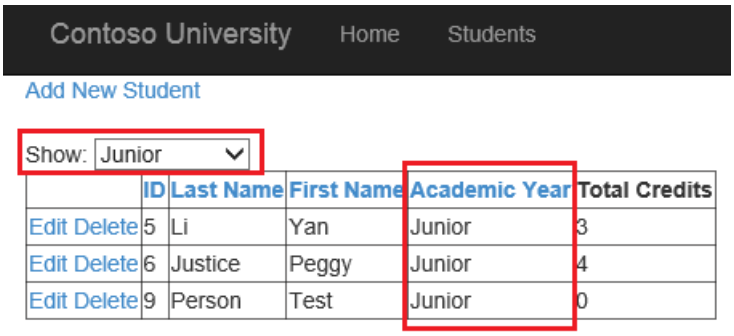
The following code shows the select method re-worked to filter the returned data based on the value of the drop down list. Adding a control attribute before a parameter specifies that the value for this parameter comes from a control with the same name.

```
public IQueryable<Student> studentsGrid_GetData([Control] AcademicYear? displayYear)
{
    SchoolContext db = new SchoolContext();
    var query = db.Students.Include(s => s.Enrollments.Select(e => e.Course));

    if (displayYear != null)
    {
        query = query.Where(s => s.Year == displayYear);
    }

    return query;
}
```

Run the web application and select different values from the drop down list to filter the list of students.



Conclusion

In this tutorial, you enabled sorting and paging of the data. You also enabled filtering the data by the value of a control.

In the next tutorial you will enhance the UI by integrating a JQuery UI widget into the dynamic data template.

PREVIOUS NEXT

Integrating JQuery UI Datepicker with model binding and web forms

2/20/2020 • 5 minutes to read • Edit Online

by Tom FitzMacken

This tutorial series demonstrates basic aspects of using model binding with an ASP.NET Web Forms project. Model binding makes data interaction more straight-forward than dealing with data source objects (such as ObjectDataSource or SqlDataSource). This series starts with introductory material and moves to more advanced concepts in later tutorials.

This tutorial shows how to add the JQuery UI Datepicker widget to a Web Form, and use model binding to update the database with the selected value.

This tutorial builds on the project created in the first and second parts of the series.

You can download the complete project in C# or VB. The downloadable code works with either Visual Studio 2012 or Visual Studio 2013. It uses the Visual Studio 2012 template, which is slightly different than the Visual Studio 2013 template shown in this tutorial.

What you'll build

In this tutorial, you'll:

1. Add a property to your model to record the student's enrollment date
2. Enable the user to select the enrollment date using the JQuery UI Datepicker widget
3. Enforce validation rules for the enrollment date

The JQuery UI Datepicker widget enables users to easily select a date from a calendar that pops up when the user interacts with the field. Using this widget can be more convenient for users than manually typing a date. Integrating the Datepicker widget into a page that uses model binding for data operations requires only a small amount of additional work.

Add a new property to the model

First, you will add a **Datetime** property to your Student model and migrate that change to the database. Open **UniversityModels.cs**, and add the highlighted code to the Student model.

```
public class Student
{
    [Key, Display(Name = "ID")]
    [ScaffoldColumn(false)]
    public int StudentID { get; set; }

    [Required, StringLength(40), Display(Name="Last Name")]
    public string LastName { get; set; }

    [Required, StringLength(20), Display(Name = "First Name")]
    public string FirstName { get; set; }

    [EnumDataType(typeof(YearEnum)), Display(Name = "Academic Year")]
    public YearEnum AcademicYear { get; set; }

    [Range(typeof(DateTime), "1/1/2013", "1/1/3000", ErrorMessage="Please provide an enrollment date after
1/1/2013")]
    [DisplayFormat(ApplyFormatInEditMode=true, DataFormatString="{0:d}")]
    public DateTime EnrollmentDate { get; set; }

    public virtual ICollection Enrollments { get; set; }
}
```

The **RangeAttribute** is included to enforce validation rules for the property. For this tutorial, we will assume that Contoso University was founded on January 1st, 2013 and therefore earlier enrollment dates are not valid.

In the Package Management window, add a migration by running the command **add-migration AddEnrollmentDate**. Notice that the migration code adds the new Datetime column to the Student table. To match the value you specified in the RangeAttribute, add a default value for the new column, as shown in the highlighted code below.

```
namespace ContosoUniversity.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddEnrollmentDate : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.Students", "EnrollmentDate", c =>
              c.DateTime(nullable: false, defaultValue: new DateTime(2013, 1, 1)));
        }

        public override void Down()
        {
            DropColumn("dbo.Students", "EnrollmentDate");
        }
    }
}
```

Save your change to the migration file.

You do not need to seed the data again. Therefore, open **Configuration.cs** in the Migrations folder and remove or comment out the code in the **Seed** method. Save and close the file.

Now, run the command **update-database**. Notice that the column now exists in the database and all of the existing records have the default value for EnrollmentDate.

Add dynamic controls for enrollment date

You will now add controls for displaying and editing the enrollment date. At this point, the value is edited through

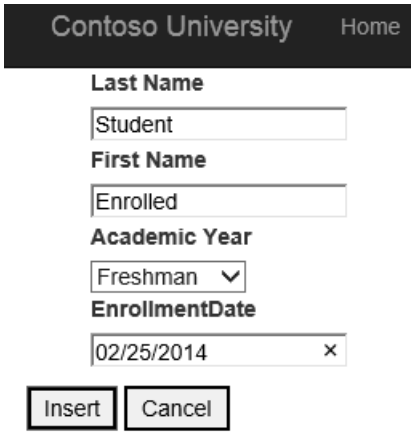
a text box. Later in the tutorial, you will change the text box to the JQuery Datepicker widget.

First, it is important to note that you do not need to make any change to the **AddStudent.aspx** file. The DynamicEntity control will automatically render the new property.

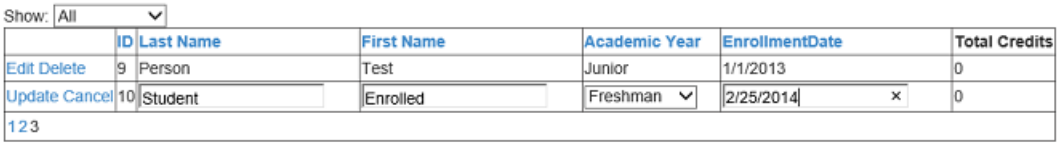
Open **Students.aspx**, and add the following highlighted code.

```
<asp:GridView runat="server" ID="studentsGrid"
        ItemType="ContosoUniversity.Models.Student" DataKeyNames="StudentID"
        SelectMethod="studentsGrid_GetData"
        UpdateMethod="studentsGrid_UpdateItem" DeleteMethod="studentsGrid_DeleteItem"
        AllowSorting="true" AllowPaging="true" PageSize="4"
        AutoGenerateEditButton="true" AutoGenerateDeleteButton="true"
        AutoGenerateColumns="false">
  <Columns>
    <asp:DynamicField DataField="StudentID" />
    <asp:DynamicField DataField="LastName" />
    <asp:DynamicField DataField="FirstName" />
    <asp:DynamicField DataField="Year" />
    <asp:DynamicField DataField="EnrollmentDate" />
    <asp:TemplateField HeaderText="Total Credits">
      <ItemTemplate>
        <asp:Label Text="<%# Item.Enrollments.Sum(en => en.Course.Credits) %>"
            runat="server" />
      </ItemTemplate>
    </asp:TemplateField>
  </Columns>
</asp:GridView>
```

Run the application and notice that you can set the value of the enrollment date by typing a date. When adding a new student:



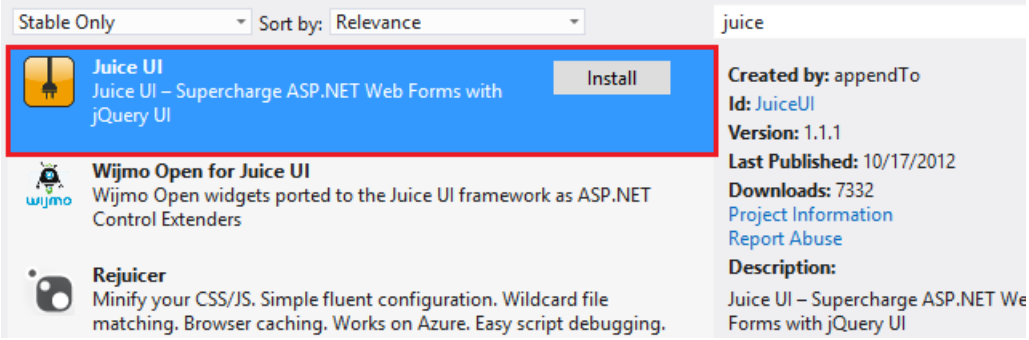
Or, editing an existing value:



Typing the date works, but it might not be the customer experience you wish to provide. In the next section, you will enable selecting a date through a calendar.

Install NuGet package to work with JQuery UI

The **Juice UI** NuGet package enables easy integration of the JQuery UI widgets into your web application. To use this package, install it through NuGet.



The version of Juice UI that you install may conflict with the version of JQuery in your application. Before proceeding with this tutorial, try running your application. If you encounter a JavaScript error, you need to reconcile the JQuery version. You can either add the expected version of JQuery to your Scripts folder (version 1.8.2 at time of writing this tutorial), or in Site.master specify the path to the JQuery file.

```
<asp:ScriptReference Name="jquery" Path="~/Scripts/jquery-1.10.2.js" />
<asp:ScriptReference Path="~/Scripts/jquery-ui-1.9.2.js" />
```

Customize DateTime template to include Datepicker widget

You will add the Datepicker widget to the dynamic data template for editing a datetime value. By adding the widget to the template, it is automatically rendered in both the form for adding a new student and in the grid view for editing students. Open **DateTime_Edit.ascx**, and add the following highlighted code.

```
<%@ Control Language="C#" CodeBehind="DateTime_Edit.ascx.cs"
Inherits="ContosoUniversityModelBinding.DateTime_EditField" %>

<juice:Datepicker runat="server" ID="t1" TargetControlID="TextBox1"/>
<asp:TextBox ID="TextBox1" runat="server" CssClass="DDTextBox" Text='<%# FieldValueEditString %>'
Columns="20"></asp:TextBox>

<asp:RequiredFieldValidator runat="server" ID="RequiredFieldValidator1" CssClass="DDControl DDValidator"
ControlToValidate="TextBox1" Display="Static" Enabled="false" />
<asp:RegularExpressionValidator runat="server" ID="RegularExpressionValidator1" CssClass="DDControl
DDValidator" ControlToValidate="TextBox1" Display="Static" Enabled="false" />
<asp:DynamicValidator runat="server" ID="DynamicValidator1" CssClass="DDControl DDValidator"
ControlToValidate="TextBox1" Display="Static" />
<asp:CustomValidator runat="server" ID="DateValidator" CssClass="DDControl DDValidator"
ControlToValidate="TextBox1" Display="Static" EnableClientScript="false" Enabled="false"
OnServerValidate="DateValidator_ServerValidate" />
```

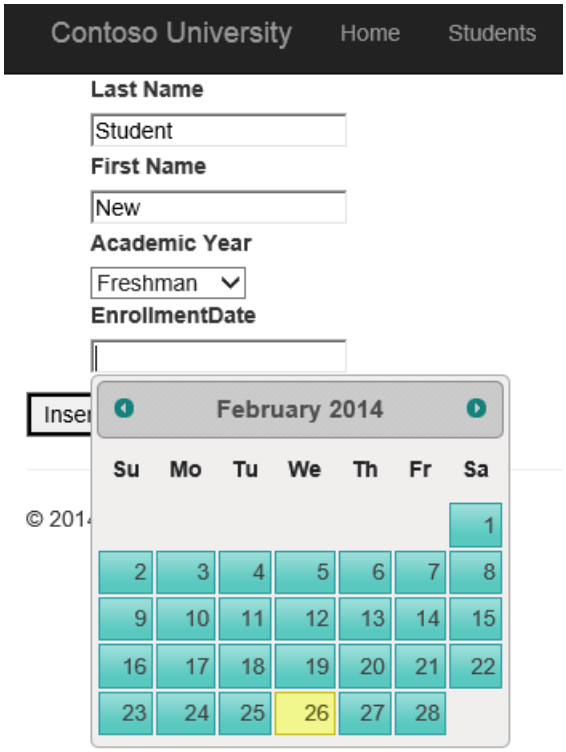
In the code-behind file, you will set the minimum and maximum dates for the DatePicker. By setting these values, you will prevent users from navigating to invalid dates. You will retrieve the minimum and maximum values from the **RangeAttribute** on the DateTime property, if one is provided. Open **DateTime_Edit.ascx.cs**, and add the following highlighted code to the Page_Load method.

```
protected void Page_Load(object sender, EventArgs e) {
    TextBox1.ToolTip = Column.Description;

    SetUpValidator(RequiredFieldValidator1);
    SetUpValidator(RegularExpressionValidator1);
    SetUpValidator(DynamicValidator1);
    SetUpCustomValidator(DateValidator);

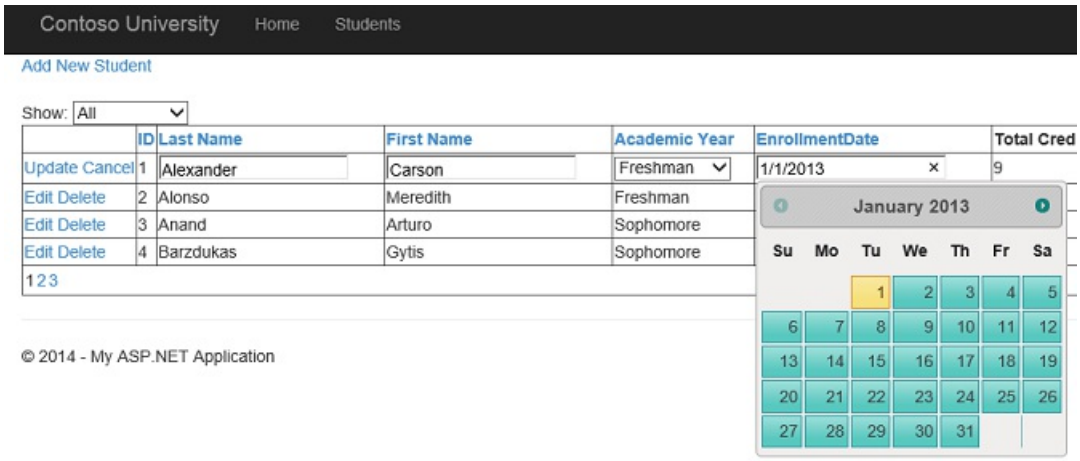
    RangeAttribute ra = (RangeAttribute)Column.Attributes[typeof(RangeAttribute)];
    if (ra != null)
    {
        t1.MinDate = ra.Minimum.ToString();
        t1.MaxDate = ra.Maximum.ToString();
    }
}
```

Run the web application and navigate to the AddStudent page. Provide values for the fields and notice that when you click on the text box for Enrollment Date, the calendar is displayed.



Pick a date, and click **Insert**. The RangeAttribute enforces validation on the server. By setting the minDate property on the Datepicker, you also apply validation on the client. The calendar does not let the user navigate to a date prior to the value of minDate.

When you edit a record in the grid view, the calendar is also displayed.



Conclusion

In this tutorial, you learned how to incorporate a JQuery widget into a web form that uses model binding.

In the next tutorial, you will use a query string value when selecting data.

PREVIOUS NEXT

Using query string values to filter data with model binding and web forms

2/20/2020 • 3 minutes to read • Edit Online

by Tom FitzMacken

This tutorial series demonstrates basic aspects of using model binding with an ASP.NET Web Forms project. Model binding makes data interaction more straight-forward than dealing with data source objects (such as ObjectDataSource or SqlDataSource). This series starts with introductory material and moves to more advanced concepts in later tutorials.

This tutorial shows how to pass a value in the query string and use that value to retrieve data through model binding.

This tutorial builds on the project created in the earlier parts of the series.

You can download the complete project in C# or VB. The downloadable code works with either Visual Studio 2012 or Visual Studio 2013. It uses the Visual Studio 2012 template, which is slightly different than the Visual Studio 2013 template shown in this tutorial.

What you'll build

In this tutorial, you'll:

1. Add a new page to show the enrolled courses for a student
2. Retrieve the enrolled courses for the selected student based on a value in the query string
3. Add a hyperlink with a query string value from the grid view to the new page

The steps in this tutorial are fairly similar to what you did in the earlier tutorial to filter the displayed students based on the user selection in a drop down list. In that tutorial, you used the **Control** attribute in the select method to specify that the parameter value comes from a control. In this tutorial, you'll use the **QueryString** attribute in the select method to specify that the parameter value comes from the query string.

Add new page for displaying a student's courses

Add a new web form that uses the Site.master master page, and name the page **Courses**.

In the **Courses.aspx** file, add a grid view to display the courses for the selected student.

```
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    <asp:GridView runat="server" ID="coursesGrid"
        ItemType="ContosoUniversityModelBinding.Models.Enrollment"
        SelectMethod="coursesGrid_GetData" AutoGenerateColumns="false">
        <Columns>
            <asp:BoundField HeaderText="Title" DataField="Course.Title" />
            <asp:BoundField HeaderText="Credits" DataField="Course.Credits" />
            <asp:BoundField HeaderText="Grade" DataField="Grade" />
        </Columns>
        <EmptyDataTemplate>
            <asp:Label Text="No Enrolled Courses" runat="server" />
        </EmptyDataTemplate>
    </asp:GridView>
</asp:Content>
```

Define the select method

In **Courses.aspx.cs**, you will add the select method with the name you specified in the grid view's **SelectMethod** property. In that method, you'll define the query for retrieving a student's courses, and specify that the parameter comes from a query string value with the same name as the parameter.

First, you must add the following **using** statements.

```
using ContosoUniversityModelBinding.Models;
using System.Web.ModelBinding;
using System.Data.Entity;
```

Then, add the following code to Courses.aspx.cs:

```
public IQueryable<Enrollment> coursesGrid_GetData([QueryString] int? studentID)
{
    SchoolContext db = new SchoolContext();
    var query = db.Enrollments.Include(e => e.Course)
        .Where(e => e.StudentID == studentID);
    return query;
}
```

The QueryString attribute means that a query string value named StudentID is automatically assigned to the parameter in this method.

Add hyperlink with query string value

In the grid view on Students.aspx, you will add a hyperlink field that links to your new Courses page. The hyperlink will include a query string value with the student's id.

In Students.aspx, add the following field to the grid view columns just below the field for Total Credits.

```
<asp:TemplateField HeaderText="Total Credits">
    <ItemTemplate>
        <asp:Label Text="<%# Item.Enrollments.Sum(en => en.Course.Credits) %>"
            runat="server" />
    </ItemTemplate>
</asp:TemplateField>
<asp:HyperLinkField Text="Courses" DataNavigateUrlFormatString="~/Courses.aspx?StudentID={0}"
    DataNavigateUrlFields="StudentID" />
```

Run the application and notice that the grid view now includes the Courses link.

Contoso University Home Students

Add New Student

Show: All

	ID	Last Name	First Name	Academic Year	EnrollmentDate	Total Credits	
Edit Delete	1	Alexander	Carson	Freshman	1/1/2013	9	Courses
Edit Delete	2	Alonso	Meredith	Freshman	1/1/2013	11	Courses
Edit Delete	3	Anand	Arturo	Sophomore	1/1/2013	3	Courses
Edit Delete	4	Barzdukas	Gytis	Sophomore	1/1/2013	6	Courses

1 2 3

When you click one of the links, you'll see that student's enrolled courses.

Contoso University

Title	Credits	Grade
Calculus	4	2.00
Trigonometry	4	4.00
Composition	3	4.00

Conclusion

In this tutorial, you added a link with a query string value. You used that query string value for the parameter value in the select method.

In the next tutorial, you will move the code from the code-behind files into a business logic layer and a data access layer.

PREVIOUS NEXT

Adding business logic layer to a project that uses model binding and web forms

2/20/2020 • 5 minutes to read • Edit Online

by Tom FitzMacken

This tutorial series demonstrates basic aspects of using model binding with an ASP.NET Web Forms project. Model binding makes data interaction more straight-forward than dealing with data source objects (such as ObjectDataSource or SqlDataSource). This series starts with introductory material and moves to more advanced concepts in later tutorials.

This tutorial shows how to use model binding with a business logic layer. You will set the OnCallingDataMethods member to specify that an object other than the current page is used to call the data methods.

This tutorial builds on the project created in the earlier parts of the series.

You can download the complete project in C# or VB. The downloadable code works with either Visual Studio 2012 or Visual Studio 2013. It uses the Visual Studio 2012 template, which is slightly different than the Visual Studio 2013 template shown in this tutorial.

What you'll build

Model binding enables you to put your data interaction code in either the code-behind file for a web page or in a separate business logic class. The previous tutorials have shown how to use the code-behind files for data interaction code. This approach works for small sites but it can lead to code repetition and greater difficulty when maintaining a large site. It can also be very difficult to programmatically test code that resides in code behind files because there is no abstraction layer.

To centralize the data interaction code, you can create a business logic layer that contains all of the logic for interacting with data. You then call the business logic layer from your web pages. This tutorial shows how to move all of the code you have written in the previous tutorials into a business logic layer, and then use that code from the pages.

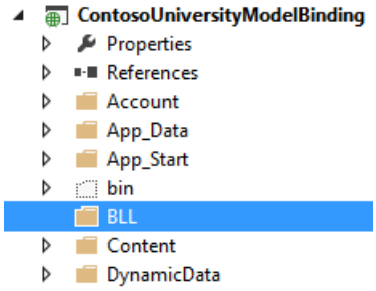
In this tutorial, you'll:

1. Move the code from code-behind files to a business logic layer
2. Change your data bound controls to call the methods in the business logic layer

Create business logic layer

Now, you will create the class that is called from the web pages. The methods in this class look similar to the methods you used in the previous tutorials and include the value provider attributes.

First, add a new folder called **BLL**.



In the BLL folder, create a new class named **SchoolBL.cs**. It will contain all of the data operations that originally resided in code-behind files. The methods are almost the same as the methods in the code-behind file, but will include some changes.

The most important change to note is that you are no longer executing the code from within an instance of **Page** class. The Page class contains the **TryUpdateModel** method and the **ModelState** property. When this code is moved to a business logic layer, you no longer have an instance of the Page class to call these members. To get around this issue, you must add a **ModelMethodContext** parameter to any method that accesses TryUpdateModel or ModelState. You use this ModelMethodContext parameter to call TryUpdateModel or retrieve ModelState. You do not need to change anything in the web page to account for this new parameter.

Replace the code in SchoolBL.cs with the following code.

```
using System;
using System.Linq;
using ContosoUniversityModelBinding.Models;
using System.Web.ModelBinding;
using System.Web.UI.WebControls;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;

namespace ContosoUniversityModelBinding.BLL
{
    public class SchoolBL : IDisposable
    {
        SchoolContext db = new SchoolContext();

        public IQueryable<Student> GetStudents([Control] AcademicYear? displayYear)
        {
            var query = db.Students.Include(s => s.Enrollments.Select(e => e.Course));

            if (displayYear != null)
            {
                query = query.Where(s => s.Year == displayYear);
            }

            return query;
        }

        public void InsertStudent(ModelMethodContext context)
        {
            var item = new Student();

            context.TryUpdateModel(item);
            if (context.ModelState.IsValid)
            {
                db.Students.Add(item);
                db.SaveChanges();
            }
        }

        public void DeleteStudent(int studentID, ModelMethodContext context)
        {
            var item = new Student { StudentID = studentID };
            db.Entry(item).State = EntityState.Deleted;
```

```
            db.Entry(item).State = EntityState.Deleted;
            try
            {
                db.SaveChanges();
            }
            catch (DbUpdateConcurrencyException)
            {
                context.ModelState.AddModelError("",
                    String.Format("Item with id {0} no longer exists in the database.", studentID));
            }
        }

        public void UpdateStudent(int studentID, ModelMethodContext context)
        {
            Student item = null;
            item = db.Students.Find(studentID);
            if (item == null)
            {
                context.ModelState.AddModelError("", String.Format("Item with id {0} was not found",
studentID));
                return;
            }

            context.TryUpdateModel(item);
            if (context.ModelState.IsValid)
            {
                db.SaveChanges();
            }
        }

        public IQueryable<Enrollment> GetCourses([QueryString] int? studentID)
        {
            var query = db.Enrollments.Include(e => e.Course)
                .Where(e => e.StudentID == studentID);
            return query;
        }

        private bool disposedValue = false;

        protected virtual void Dispose(bool disposing)
        {
            if (!this.disposedValue)
            {
                if (disposing)
                {
                    db.Dispose();
                }
            }
            this.disposedValue = true;
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
    }
}
```

Revise existing pages to retrieve data from business logic layer

Finally, you will convert the pages Students.aspx, AddStudent.aspx, and Courses.aspx from using queries in the code-behind file to using the business logic layer.

In the code-behind files for Students, AddStudent, and Courses, delete or comment out the following query methods:

- studentsGrid_GetData
- studentsGrid_UpdateItem
- studentsGrid_DeleteItem
- addStudentForm_InsertItem
- coursesGrid_GetData

You now should have no code in the code-behind file that pertains to data operations.

The **OnCallingDataMethods** event handler enables you to specify an object to use for the data methods. In Students.aspx, add a value for that event handler and change the names of the data methods to the names of the methods in the business logic class.

```
<asp:GridView runat="server" ID="studentsGrid"
    ItemType="ContosoUniversityModelBinding.Models.Student" DataKeyNames="StudentID"
    SelectMethod="GetStudents"
    UpdateMethod="UpdateStudent" DeleteMethod="DeleteStudent"
    AllowSorting="true" AllowPaging="true" PageSize="4"
    AutoGenerateEditButton="true" AutoGenerateDeleteButton="true"
    AutoGenerateColumns="false"
    OnCallingDataMethods="studentsGrid_CallingDataMethods">
```

In the code-behind file for Students.aspx, define the event handler for the CallingDataMethods event. In this event handler, you specify the business logic class for data operations.

```
protected void studentsGrid_CallingDataMethods(object sender, CallingDataMethodsEventArgs e)
{
    e.DataMethodsObject = new ContosoUniversityModelBinding.BLL.SchoolBL();
}
```

In AddStudent.aspx, make similar changes.

```
<asp:FormView runat="server" ID="addStudentForm"
    ItemType="ContosoUniversityModelBinding.Models.Student"
    InsertMethod="InsertStudent" DefaultMode="Insert"
    OnCallingDataMethods="addStudentForm_CallingDataMethods"
    RenderOuterTable="false" OnItemInserted="addStudentForm_ItemInserted">
```

```
protected void addStudentForm_CallingDataMethods(object sender, CallingDataMethodsEventArgs e)
{
    e.DataMethodsObject = new ContosoUniversityModelBinding.BLL.SchoolBL();
}
```

In Courses.aspx, make similar changes.

```
<asp:GridView runat="server" ID="coursesGrid"
    ItemType="ContosoUniversityModelBinding.Models.Enrollment"
    SelectMethod="GetCourses" AutoGenerateColumns="false"
    OnCallingDataMethods="coursesGrid_CallingDataMethods">
```

```
protected void coursesGrid_CallingDataMethods(object sender, CallingDataMethodsEventArgs e)
{
    e.DataMethodsObject = new ContosoUniversityModelBinding.BLL.SchoolBL();
}
```

Run the application and notice that all of the pages function as they had previously. The validation logic also works correctly.

Conclusion

In this tutorial, you re-structured your application to use a data access layer and business logic layer. You specified that the data controls use an object that is not the current page for data operations.

PREVIOUS

Security, Authentication, and Authorization in ASP.NET Web Forms

2/20/2020 • 2 minutes to read • Edit Online

How to let users to log in to your site (and optionally be assigned to roles) using either a login form or Windows authentication.

- Create a secure ASP.NET Web Forms app with user registration, email confirmation and password reset (C#)
- Create an ASP.NET Web Forms app with SMS Two-Factor Authentication (C#)

Create a secure ASP.NET Web Forms app with user registration, email confirmation and password reset (C#)

2/20/2020 • 14 minutes to read • Edit Online

by Erik Reitan

This tutorial shows you how to build an ASP.NET Web Forms app with user registration, email confirmation and password reset using the ASP.NET Identity membership system. This tutorial was based on Rick Anderson's MVC tutorial.

Introduction

This tutorial guides you through the steps required to create an ASP.NET Web Forms application using Visual Studio and ASP.NET 4.5 to create a secure Web Forms app with user registration, email confirmation and password reset.

Tutorial Tasks and Information:

- Create an ASP.NET Web Forms app
- Hook Up SendGrid
- Require Email Confirmation Before Log In
- Password Recovery and Reset
- Resend Email Confirmation Link
- Troubleshooting the App
- Additional Resources

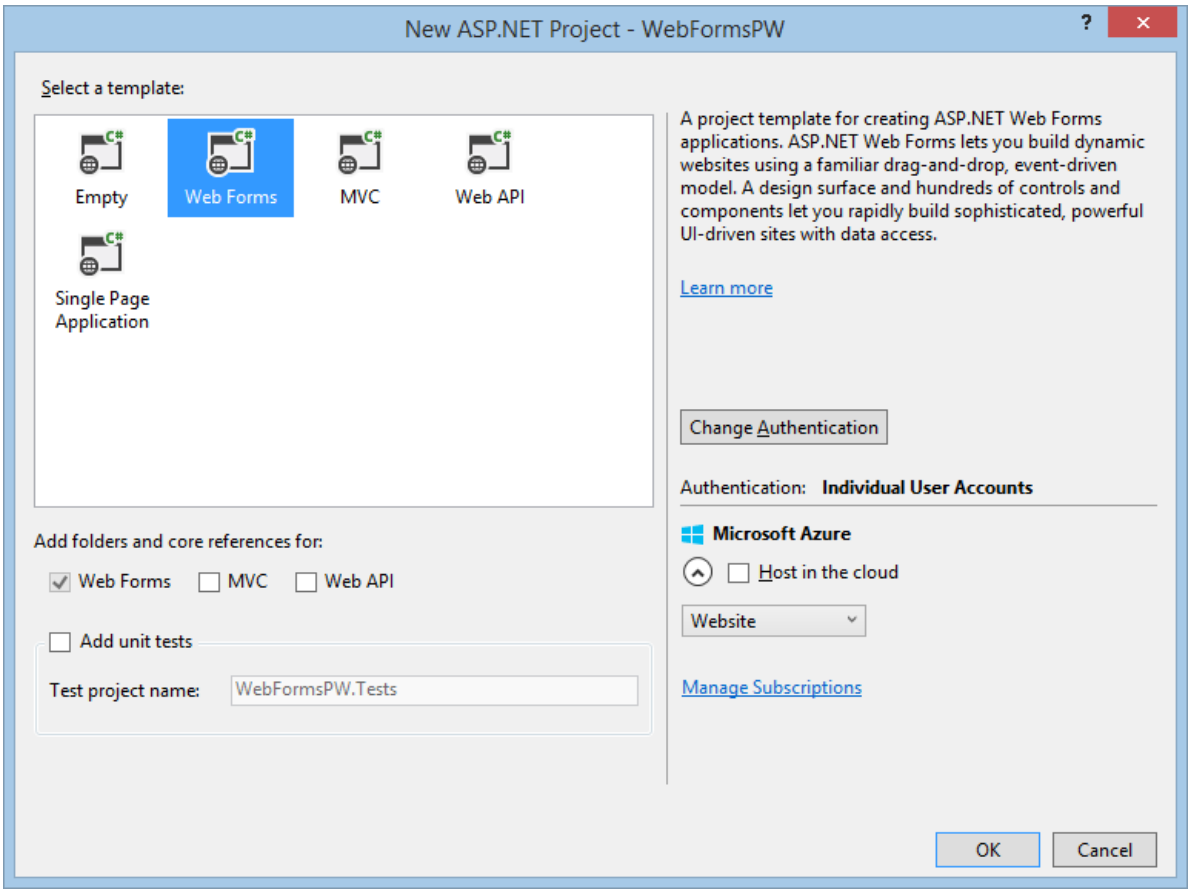
Create an ASP.NET Web Forms App

Start by installing and running Visual Studio Express 2013 for Web or Visual Studio 2013. Install Visual Studio 2013 Update 3 or higher as well.

> **NOTE**
>
> Warning: You must install Visual Studio 2013 Update 3 or higher to complete this tutorial.

1. Create a new project (**File** -> **New Project**) and select the **ASP.NET Web Application** template and the latest .NET Framework version from the **New Project** dialog box.

2. From the **New ASP.NET Project** dialog box, select the **Web Forms** template. Leave the default authentication as **Individual User Accounts**. If you'd like to host the app in Azure, leave the **Host in the cloud** check box checked.
   Then, click **OK** to create the new project.

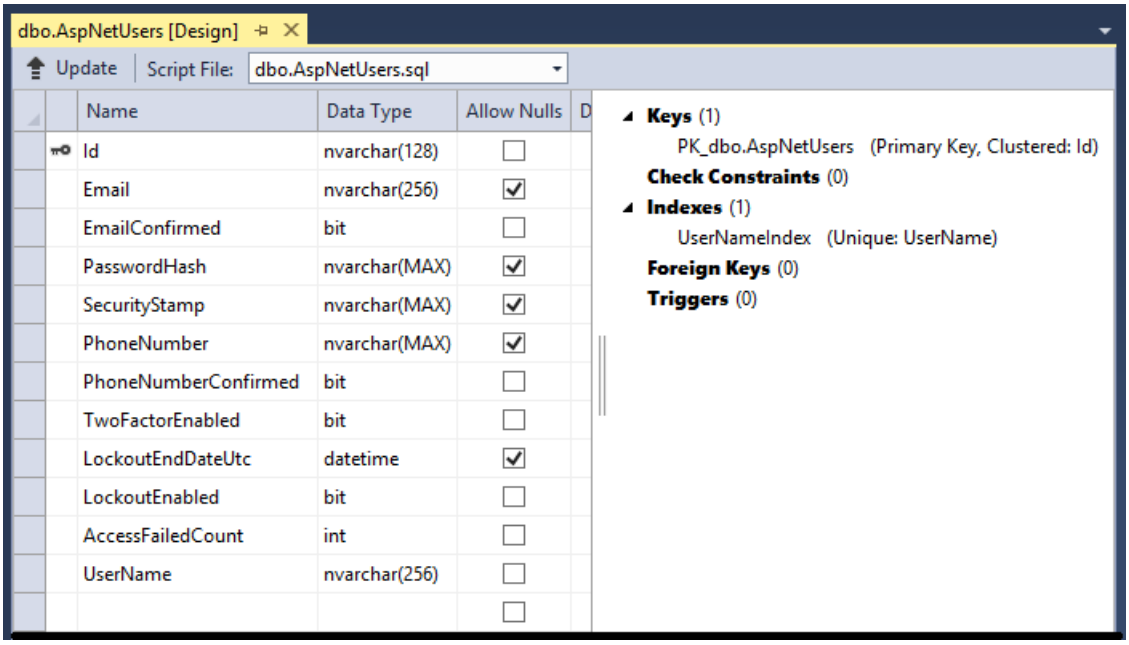


3. Enable Secure Sockets Layer (SSL) for the project. Follow the steps available in the **Enable SSL for the Project** section of the Getting Started with Web Forms tutorial series.

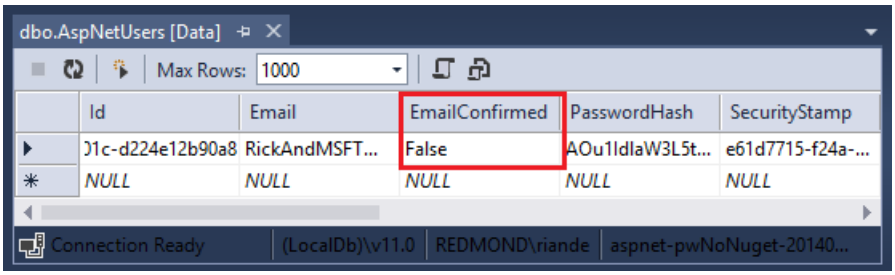
4. Run the app, click the **Register** link and register a new user. At this point, the only validation on the email is based on the [EmailAddress] attribute to ensure the email address is well-formed. You will modify the code to add email confirmation. Close the browser window.

5. In **Server Explorer** of Visual Studio (**View -> Server Explorer**), navigate to **Data Connections\DefaultConnection\Tables\AspNetUsers**, right click and select **Open table definition**.

   The following image shows the `AspNetUsers` table schema:



6. In **Server Explorer**, right-click on the **AspNetUsers** table and select **Show Table Data**.



At this point the email for the registered user has not been confirmed.

7. Click on the row and select delete to delete the user. You'll add this email again in the next step and send a confirmation message to the email address.

Email Confirmation

It's a best practice to confirm the email during the registration of a new user to verify they are not impersonating someone else (that is, they haven't registered with someone else's email). Suppose you had a discussion forum, you would want to prevent `"bob@cpandl.com"` from registering as `"joe@contoso.com"`. Without email confirmation, `"joe@contoso.com"` could get unwanted email from your app. Suppose Bob accidentally registered as `"bib@cpandl.com"` and hadn't noticed it, he wouldn't be able to use password recovery because the app doesn't have his correct email. Email confirmation provides only limited protection from bots and doesn't provide protection from determined spammers.

You generally want to prevent new users from posting any data to your website before they have been confirmed by either email, an SMS text message or another mechanism. In the sections below, we will enable email confirmation and modify the code to prevent newly registered users from logging in until their email has been confirmed. You'll use the email service SendGrid in this tutorial.

Hook up SendGrid

SendGrid has changed it's API since this tutorial was written. For current SendGrid instructions, see SendGrid or Enable account confirmation and password recovery.

Although this tutorial only shows how to add email notification through SendGrid, you can send email using SMTP and other mechanisms (see additional resources).

1. In Visual Studio, open the **Package Manager Console** (**Tools** -> **NuGet Package Manger** -> **Package Manager Console**), and enter the following command:
   `Install-Package SendGrid`

2. Go to the Azure SendGrid sign-up page and register for free SendGrid account. You can also sign-up for a free SendGrid account directly on SendGrid's site.

3. From **Solution Explorer** open the *IdentityConfig.cs* file in the *App_Start* folder and add the following code highlighted in yellow to the `EmailService` class to configure **SendGrid**:

```
public class EmailService : IIdentityMessageService
{
   public async Task SendAsync(IdentityMessage message)
   {
     await configSendGridasync(message);
   }

   // Use NuGet to install SendGrid (Basic C# client lib)
   private async Task configSendGridasync(IdentityMessage message)
   {
      var myMessage = new SendGridMessage();
      myMessage.AddTo(message.Destination);
      myMessage.From = new System.Net.Mail.MailAddress(
                          "Royce@contoso.com", "Royce Sellars (Contoso Admin)");
      myMessage.Subject = message.Subject;
      myMessage.Text = message.Body;
      myMessage.Html = message.Body;

      var credentials = new NetworkCredential(
                 ConfigurationManager.AppSettings["emailServiceUserName"],
                 ConfigurationManager.AppSettings["emailServicePassword"]
                 );

      // Create a Web transport for sending email.
      var transportWeb = new Web(credentials);

      // Send the email.
      if (transportWeb != null)
      {
         await transportWeb.DeliverAsync(myMessage);
      }
      else
      {
         Trace.TraceError("Failed to create Web transport.");
         await Task.FromResult(0);
      }
   }
}
```

4. Also, add the following `using` statements to the beginning of the *IdentityConfig.cs* file:

```
using SendGrid;
using System.Net;
using System.Configuration;
using System.Diagnostics;
```

5. To keep this sample simple, you'll store the email service account values in the `appSettings` section of the *web.config* file. Add the following XML highlighted in yellow to the *web.config* file at the root of your project:

```
</connectionStrings>
   <appSettings>
      <add key="emailServiceUserName" value="[EmailServiceAccountUserName]" />
      <add key="emailServicePassword" value="[EmailServiceAccountPassword]" />
   </appSettings>
  <system.web>
```

> **WARNING**
> Security - Never store sensitive data in your source code. In this example, the account and credentials are stored in the **appSetting** section of the *Web.config* file. On Azure, you can securely store these values on the **Configure** tab in the Azure portal. For related information see Rick Anderson's topic titled Best practices for deploying passwords and other sensitive data to ASP.NET and Azure.

6. Add the email service values to reflect your SendGrid authentication values (User Name and Password) so that you can successful send email from your app. Be sure to use your SendGrid account name rather than the email address you provided SendGrid.

Enable Email Confirmation

To enable email confirmation, you'll modify the registration code using the following steps.

1. In the *Account* folder, open the *Register.aspx.cs* code-behind and update the `CreateUser_Click` method to enable the following highlighted changes:

```
protected void CreateUser_Click(object sender, EventArgs e)
{
    var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
    var user = new ApplicationUser() { UserName = Email.Text, Email = Email.Text };
    IdentityResult result = manager.Create(user, Password.Text);
    if (result.Succeeded)
    {
        // For more information on how to enable account confirmation and password reset please visit
https://go.microsoft.com/fwlink/?LinkID=320771
        string code = manager.GenerateEmailConfirmationToken(user.Id);
        string callbackUrl = IdentityHelper.GetUserConfirmationRedirectUrl(code, user.Id, Request);
        manager.SendEmail(user.Id, "Confirm your account", "Please confirm your account by clicking <a
href=\"" + callbackUrl + "\">here</a>.");

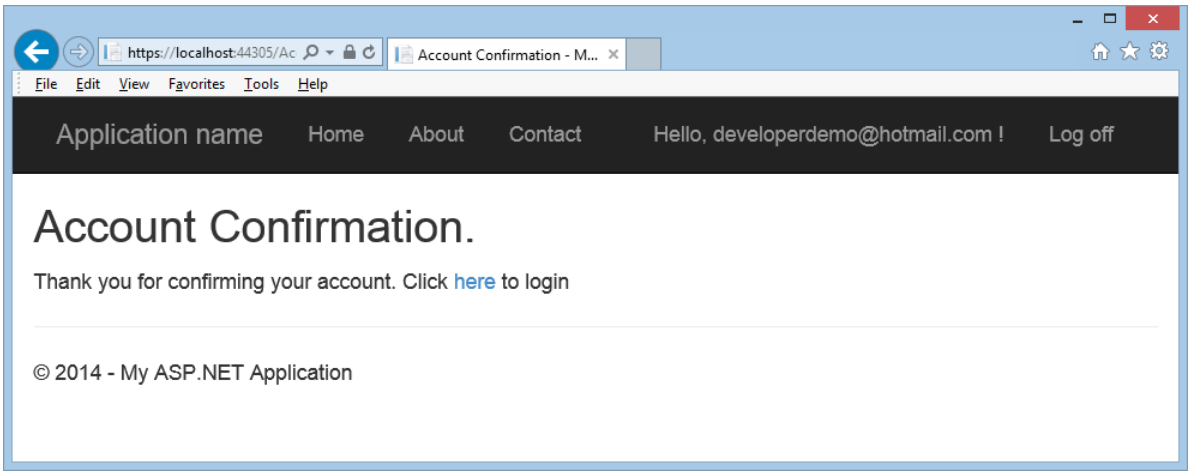
        IdentityHelper.SignIn(manager, user, isPersistent: false);
        IdentityHelper.RedirectToReturnUrl(Request.QueryString["ReturnUrl"], Response);
    }
    else
    {
        ErrorMessage.Text = result.Errors.FirstOrDefault();
    }
}
```

2. In **Solution Explorer**, right-click *Default.aspx* and select **Set As Start Page**.

3. Run the app by pressing **F5**. After the page is displayed, click the **Register** link to display the Register page.

4. Enter your email and password, then click the **Register** button to send an email message via SendGrid. The current state of your project and code will allow the user to log in once they complete the registration form, even though they haven't confirmed their account.

5. Check your email account and click on the link to confirm your email.
   Once you submit the registration form, you will be logged in.



Require Email Confirmation Before Log In

Although you have confirmed the email account, at this point you would not need to click on the link contained in the verification email to be fully signed-in. In the following section, you will modify the code requiring new users to have a confirmed email before they are logged in (authenticated).

1. In **Solution Explorer** of Visual Studio, update the `CreateUser_Click` event in the *Register.aspx.cs* code-behind contained in the *Accounts* folder with the following highlighted changes:

```
protected void CreateUser_Click(object sender, EventArgs e)
{
    var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
    var user = new ApplicationUser() { UserName = Email.Text, Email = Email.Text };
    IdentityResult result = manager.Create(user, Password.Text);
    if (result.Succeeded)
    {
        // For more information on how to enable account confirmation and password reset please visit
https://go.microsoft.com/fwlink/?LinkID=320771
        string code = manager.GenerateEmailConfirmationToken(user.Id);
        string callbackUrl = IdentityHelper.GetUserConfirmationRedirectUrl(code, user.Id, Request);
        manager.SendEmail(user.Id, "Confirm your account", "Please confirm your account by clicking <a
href=\"" + callbackUrl + "\">here</a>.");

        if (user.EmailConfirmed)
        {
          IdentityHelper.SignIn(manager, user, isPersistent: false);
          IdentityHelper.RedirectToReturnUrl(Request.QueryString["ReturnUrl"], Response);
        }
        else
        {
          ErrorMessage.Text = "An email has been sent to your account. Please view the email and
confirm your account to complete the registration process.";
        }
    }
    else
    {
        ErrorMessage.Text = result.Errors.FirstOrDefault();
    }
}
```

2. Update the `LogIn` method in the *Login.aspx.cs* code-behind with the following highlighted changes:

```
protected void LogIn(object sender, EventArgs e)
{
    if (IsValid)
    {
        // Validate the user password
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        var signinManager = Context.GetOwinContext().GetUserManager<ApplicationSignInManager>();

        // Require the user to have a confirmed email before they can log on.
        var user = manager.FindByName(Email.Text);
        if (user != null)
        {
            if (!user.EmailConfirmed)
            {
                FailureText.Text = "Invalid login attempt. You must have a confirmed email account.";
                ErrorMessage.Visible = true;
            }
            else
            {
                // This doen't count login failures towards account lockout
                // To enable password failures to trigger lockout, change to shouldLockout: true
                var result = signinManager.PasswordSignIn(Email.Text, Password.Text,
RememberMe.Checked, shouldLockout: false);

                switch (result)
                {
                    case SignInStatus.Success:
                        IdentityHelper.RedirectToReturnUrl(Request.QueryString["ReturnUrl"], Response);
                        break;
                    case SignInStatus.LockedOut:
                        Response.Redirect("/Account/Lockout");
                        break;
                    case SignInStatus.RequiresVerification:
                        Response.Redirect(String.Format("/Account/TwoFactorAuthenticationSignIn?
ReturnUrl={0}&RememberMe={1}",
                                                  Request.QueryString["ReturnUrl"],
                                                  RememberMe.Checked),
                                       true);
                        break;
                    case SignInStatus.Failure:
                    default:
                        FailureText.Text = "Invalid login attempt";
                        ErrorMessage.Visible = true;
                        break;
                }
            }
        }
    }
}
```

Run the Application

Now that you have implemented the code to check whether a user's email address has been confirmed, you can check the functionality on both the **Register** and **Login** pages.

1. Delete any accounts in the **AspNetUsers** table that contain the email alias you wish to test.
2. Run the app (**F5**) and verify you cannot register as a user until you have confirmed your email address.
3. Before confirming your new account via the email that was just sent, attempt to log in with the new account. You'll see that you are unable to log in and that you must have a confirmed email account.
4. Once you confirm your email address, log in to the app.

Password Recovery and Reset

1. In Visual Studio, remove the comment characters from the `Forgot` method in the *Forgot.aspx.cs* code-behind contained in the *Account* folder, so that the method appears as follows:

```
protected void Forgot(object sender, EventArgs e)
{
    if (IsValid)
    {
        // Validate the user's email address
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        ApplicationUser user = manager.FindByName(Email.Text);
        if (user == null || !manager.IsEmailConfirmed(user.Id))
        {
            FailureText.Text = "The user either does not exist or is not confirmed.";
            ErrorMessage.Visible = true;
            return;
        }
        // For more information on how to enable account confirmation and password reset please visit
https://go.microsoft.com/fwlink/?LinkID=320771
        // Send email with the code and the redirect to reset password page
        string code = manager.GeneratePasswordResetToken(user.Id);
        string callbackUrl = IdentityHelper.GetResetPasswordRedirectUrl(code, Request);
        manager.SendEmail(user.Id, "Reset Password", "Please reset your password by clicking <a
href=\"" + callbackUrl + "\">here</a>.");
        loginForm.Visible = false;
        DisplayEmail.Visible = true;
    }
}
```

2. Open the *Login.aspx* page. Replace the markup near the end of the **loginForm** section as highlighted below:

```
<%@ Page Title="Log in" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="Login.aspx.cs" Inherits="WebForms002.Account.Login" Async="true" %>

<%@ Register Src="~/Account/OpenAuthProviders.ascx" TagPrefix="uc" TagName="OpenAuthProviders" %>

<asp:Content runat="server" ID="BodyContent" ContentPlaceHolderID="MainContent">
    <h2><%: Title %>.</h2>

    <div class="row">
        <div class="col-md-8">
            <section id="loginForm">
                <div class="form-horizontal">
                    <h4>Use a local account to log in.</h4>
                    <hr />
                    <asp:PlaceHolder runat="server" ID="ErrorMessage" Visible="false">
                        <p class="text-danger">
                            <asp:Literal runat="server" ID="FailureText" />
                        </p>
                    </asp:PlaceHolder>
                    <div class="form-group">
                        <asp:Label runat="server" AssociatedControlID="Email" CssClass="col-md-2
control-label">Email</asp:Label>
                        <div class="col-md-10">
                            <asp:TextBox runat="server" ID="Email" CssClass="form-control"
TextMode="Email" />
                            <asp:RequiredFieldValidator runat="server" ControlToValidate="Email"
                                CssClass="text-danger" ErrorMessage="The email field is required." />
                        </div>
                    </div>
                    <div class="form-group">
                        <asp:Label runat="server" AssociatedControlID="Password" CssClass="col-md-2
control-label">Password</asp:Label>
                        <div class="col-md-10">
                            <asp:TextBox runat="server" ID="Password" TextMode="Password"
```

```
CssClass="form-control" />
                            <asp:RequiredFieldValidator runat="server" ControlToValidate="Password"
CssClass="text-danger" ErrorMessage="The password field is required." />
                        </div>
                    </div>
                    <div class="form-group">
                        <div class="col-md-offset-2 col-md-10">
                            <div class="checkbox">
                                <asp:CheckBox runat="server" ID="RememberMe" />
                                <asp:Label runat="server" AssociatedControlID="RememberMe">Remember me?
</asp:Label>
                            </div>
                        </div>
                    </div>
                    <div class="form-group">
                        <div class="col-md-offset-2 col-md-10">
                            <asp:Button runat="server" OnClick="LogIn" Text="Log in" CssClass="btn btn-
default" />
                        </div>
                    </div>
                </div>
                <p>
                    <asp:HyperLink runat="server" ID="RegisterHyperLink"
ViewStateMode="Disabled">Register as a new user</asp:HyperLink>
                </p>
                <p>
                    <%-- Enable this once you have account confirmation enabled for password reset
functionality --%>
                    <asp:HyperLink runat="server" ID="ForgotPasswordHyperLink"
ViewStateMode="Disabled">Forgot your password?</asp:HyperLink>
                </p>
            </section>
        </div>

        <div class="col-md-4">
            <section id="socialLoginForm">
                <uc:OpenAuthProviders runat="server" ID="OpenAuthLogin" />
            </section>
        </div>
    </div>
</asp:Content>
```

3. Open the *Login.aspx.cs* code-behind and uncomment the following line of code highlighted in yellow from the `Page_Load` event handler:

```
protected void Page_Load(object sender, EventArgs e)
{
    RegisterHyperLink.NavigateUrl = "Register";
    // Enable this once you have account confirmation enabled for password reset functionality
    ForgotPasswordHyperLink.NavigateUrl = "Forgot";
    OpenAuthLogin.ReturnUrl = Request.QueryString["ReturnUrl"];
    var returnUrl = HttpUtility.UrlEncode(Request.QueryString["ReturnUrl"]);
    if (!String.IsNullOrEmpty(returnUrl))
    {
        RegisterHyperLink.NavigateUrl += "?ReturnUrl=" + returnUrl;
    }
}
```

4. Run the app by pressing **F5**. After the page is displayed, click the **Log in** link.

5. Click the **Forgot your password?** link to display the **Forgot Password** page.

6. Enter your email address and click the **Submit** button to send an email to your address which will allow you to reset your password.
   Check your email account and click on the link to display the **Reset Password** page.

7. On the **Reset Password** page, enter your email, password, and confirmed password. Then, press the **Reset** button.
   When you successfully reset your password, the **Password Changed** page will be displayed. Now you can log in with your new password.

Resend Email Confirmation Link

Once a user creates a new local account, they are emailed a confirmation link they are required to use before they can log on. If the user accidentally deletes the confirmation email, or the email never arrives, they will need the confirmation link sent again. The following code changes show how to enable this.

1. In Visual Studio, open the **Login.aspx.cs** code-behind and add the following event handler after the `LogIn` event handler:

```
protected void SendEmailConfirmationToken(object sender, EventArgs e)
{
    var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
    var user = manager.FindByName(Email.Text);
    if (user != null)
    {
        if (!user.EmailConfirmed)
        {
            string code = manager.GenerateEmailConfirmationToken(user.Id);
            string callbackUrl = IdentityHelper.GetUserConfirmationRedirectUrl(code, user.Id, Request);
            manager.SendEmail(user.Id, "Confirm your account", "Please confirm your account by clicking
<a href=\"" + callbackUrl + "\">here</a>.");

            FailureText.Text = "Confirmation email sent. Please view the email and confirm your
account.";
            ErrorMessage.Visible = true;
            ResendConfirm.Visible = false;
        }
    }
}
```

2. Modify the `LogIn` event handler in the *Login.aspx.cs* code-behind by changing the code highlighted in yellow as follows:

```
protected void LogIn(object sender, EventArgs e)
{
    if (IsValid)
    {
        // Validate the user password
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        var signinManager = Context.GetOwinContext().GetUserManager<ApplicationSignInManager>();

        // Require the user to have a confirmed email before they can log on.
        var user = manager.FindByName(Email.Text);
        if (user != null)
        {
            if (!user.EmailConfirmed)
            {
                FailureText.Text = "Invalid login attempt. You must have a confirmed email address.
Enter your email and password, then press 'Resend Confirmation'.";
                ErrorMessage.Visible = true;
                ResendConfirm.Visible = true;
            }
            else
            {
                // This doen't count login failures towards account lockout
                // To enable password failures to trigger lockout, change to shouldLockout: true
                var result = signinManager.PasswordSignIn(Email.Text, Password.Text,
RememberMe.Checked, shouldLockout: false);

                switch (result)
                {
                    case SignInStatus.Success:
                        IdentityHelper.RedirectToReturnUrl(Request.QueryString["ReturnUrl"], Response);
                        break;
                    case SignInStatus.LockedOut:
                        Response.Redirect("/Account/Lockout");
                        break;
                    case SignInStatus.RequiresVerification:
                        Response.Redirect(String.Format("/Account/TwoFactorAuthenticationSignIn?
ReturnUrl={0}&RememberMe={1}",
                                                    Request.QueryString["ReturnUrl"],
                                                    RememberMe.Checked),
                                        true);
                        break;
                    case SignInStatus.Failure:
                    default:
                        FailureText.Text = "Invalid login attempt";
                        ErrorMessage.Visible = true;
                        break;
                }
            }
        }
    }
}
```

3. Update the *Login.aspx* page by adding the code highlighted in yellow as follows:

```
<%@ Page Title="Log in" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="Login.aspx.cs" Inherits="WebForms002.Account.Login" Async="true" %>

<%@ Register Src="~/Account/OpenAuthProviders.ascx" TagPrefix="uc" TagName="OpenAuthProviders" %>

<asp:Content runat="server" ID="BodyContent" ContentPlaceHolderID="MainContent">
    <h2><%: Title %>.</h2>

    <div class="row">
        <div class="col-md-8">
            <section id="loginForm">
                <div class="form-horizontal">
```

```
                    <h4>Use a local account to log in.</h4>
                    <hr />
                    <asp:PlaceHolder runat="server" ID="ErrorMessage" Visible="false">
                        <p class="text-danger">
                            <asp:Literal runat="server" ID="FailureText" />
                        </p>
                    </asp:PlaceHolder>
                    <div class="form-group">
                        <asp:Label runat="server" AssociatedControlID="Email" CssClass="col-md-2
control-label">Email</asp:Label>
                        <div class="col-md-10">
                            <asp:TextBox runat="server" ID="Email" CssClass="form-control"
TextMode="Email" />
                            <asp:RequiredFieldValidator runat="server" ControlToValidate="Email"
                                CssClass="text-danger" ErrorMessage="The email field is required." />
                        </div>
                    </div>
                    <div class="form-group">
                        <asp:Label runat="server" AssociatedControlID="Password" CssClass="col-md-2
control-label">Password</asp:Label>
                        <div class="col-md-10">
                            <asp:TextBox runat="server" ID="Password" TextMode="Password"
CssClass="form-control" />
                            <asp:RequiredFieldValidator runat="server" ControlToValidate="Password"
CssClass="text-danger" ErrorMessage="The password field is required." />
                        </div>
                    </div>
                    <div class="form-group">
                        <div class="col-md-offset-2 col-md-10">
                            <div class="checkbox">
                                <asp:CheckBox runat="server" ID="RememberMe" />
                                <asp:Label runat="server" AssociatedControlID="RememberMe">Remember me?
</asp:Label>
                            </div>
                        </div>
                    </div>
                    <div class="form-group">
                        <div class="col-md-offset-2 col-md-10">
                            <asp:Button runat="server" OnClick="LogIn" Text="Log in" CssClass="btn btn-
default" />
                              
                            <asp:Button runat="server" ID="ResendConfirm"
OnClick="SendEmailConfirmationToken" Text="Resend confirmation" Visible="false" CssClass="btn btn-
default" />
                        </div>
                    </div>
                </div>
                <p>
                    <asp:HyperLink runat="server" ID="RegisterHyperLink"
ViewStateMode="Disabled">Register as a new user</asp:HyperLink>
                </p>
                <p>
                    <%-- Enable this once you have account confirmation enabled for password reset
functionality --%>
                    <asp:HyperLink runat="server" ID="ForgotPasswordHyperLink"
ViewStateMode="Disabled">Forgot your password?</asp:HyperLink>
                </p>
            </section>
        </div>

        <div class="col-md-4">
            <section id="socialLoginForm">
                <uc:OpenAuthProviders runat="server" ID="OpenAuthLogin" />
            </section>
        </div>
    </div>
</asp:Content>
```

4. Delete any accounts in the **AspNetUsers** table that contain the email alias you wish to test.

5. Run the app (**F5**) and register your email address.

6. Before confirming your new account via the email that was just sent, attempt to log in with the new account.
   You'll see that you are unable to log in and that you must have a confirmed email account. In addition, you can now resend a confirmation message to your email account.

7. Enter your email address and password, then press the **Resend confirmation** button.

8. Once you confirm your email address based on the newly sent email message, log in to the app.

Troubleshooting the App

If you don't receive an email containing the link to verify your credentials:

- Check your junk or spam folder.
- Log into your SendGrid account and click on the Email Activity link.
- Be certain you used your SendGrid user account name as a *Web.config* value rather than your SendGrid account email address.

Additional Resources

- Links to ASP.NET Identity Recommended Resources
- Account Confirmation and Password Recovery with ASP.NET Identity
- ASP.NET Web Forms tutorial series - Add an OAuth 2.0 Provider
- Deploy a Secure ASP.NET Web Forms App with Membership, OAuth, and SQL Database to Azure App Service
- ASP.NET Web Forms tutorial series - Enable SSL for the Project

Create an ASP.NET Web Forms app with SMS Two-Factor Authentication (C#)

2/20/2020 • 8 minutes to read • Edit Online

by Erik Reitan

Download ASP.NET Web Forms App with Email and SMS Two-Factor Authentication

> This tutorial shows you how to build an ASP.NET Web Forms app with Two-Factor Authentication. This tutorial was designed to complement the tutorial titled Create a secure ASP.NET Web Forms app with user registration, email confirmation and password reset. In addition, this tutorial was based on Rick Anderson's MVC tutorial.

Introduction

This tutorial guides you through the steps required to create an ASP.NET Web Forms application that supports Two-Factor Authentication using Visual Studio. Two-Factor Authentication is an extra user authentication step. This extra step generates a unique personal identification number (PIN) during sign-in. The PIN is commonly sent to the user as an email or SMS message. The user of your app then enters the PIN as an extra authentication measure when signing-in.

Tutorial Tasks and Information:

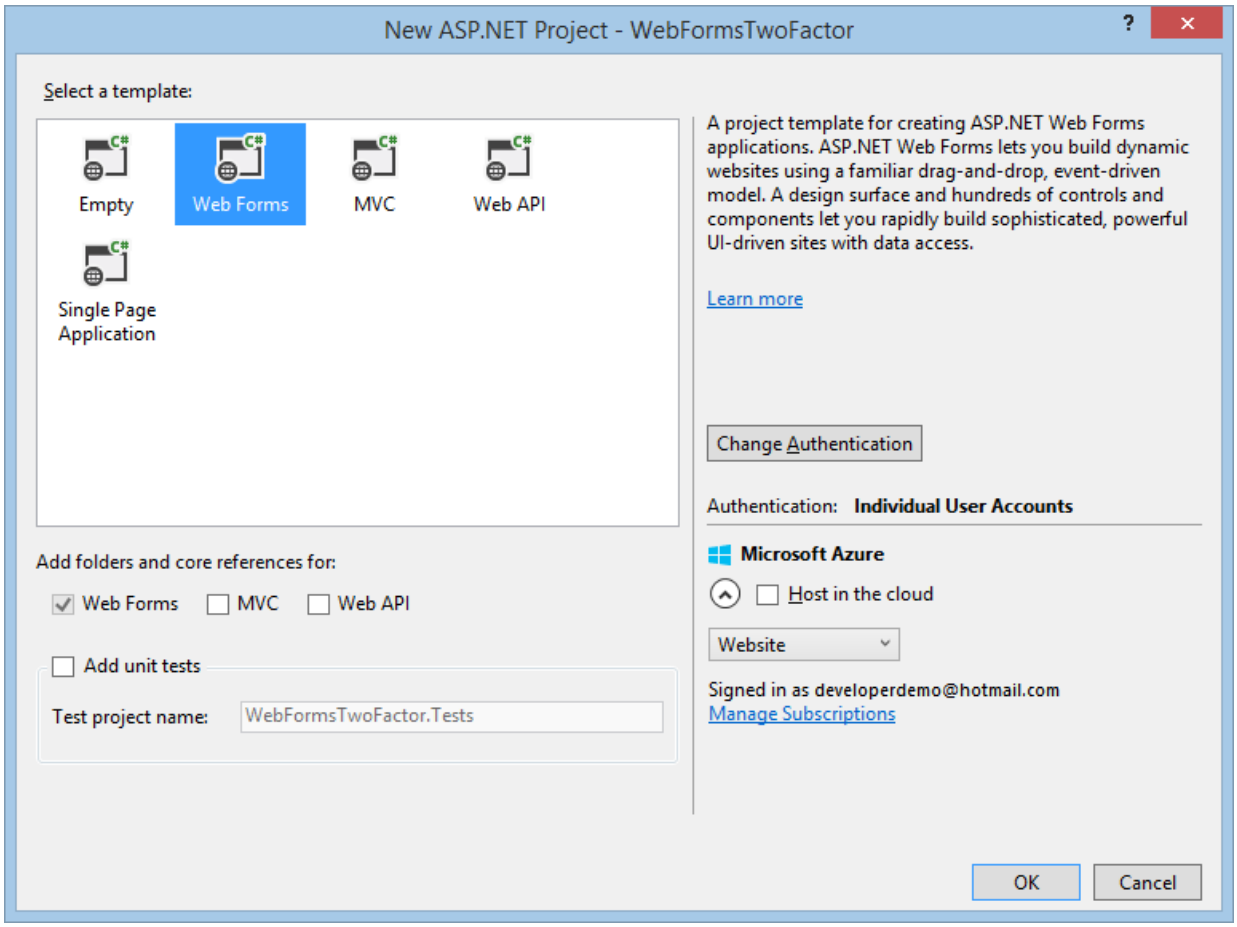
- Create an ASP.NET Web Forms App
- Setup SMS and Two-Factor Authentication
- Enable Two-Factor Authentication for Registered User
- Additional Resources

Create an ASP.NET Web Forms App

Start by installing and running Visual Studio Express 2013 for Web or Visual Studio 2013. Install Visual Studio 2013 Update 3 or higher as well. Also, you will need to create a Twilio account, as explained below.

> **NOTE**
>
> Important: You must install Visual Studio 2013 Update 3 or higher to complete this tutorial.

1. Create a new project (**File** -> **New Project**) and select the **ASP.NET Web Application** template along with the .NET Framework version 4.5.2 from the **New Project** dialog box.
2. From the **New ASP.NET Project** dialog box, select the **Web Forms** template. Leave the default authentication as **Individual User Accounts**. Then, click **OK** to create the new project.



3. Enable Secure Sockets Layer (SSL) for the project. Follow the steps available in the **Enable SSL for the Project** section of the Getting Started with Web Forms tutorial series.
4. In Visual Studio, open the **Package Manager Console** (**Tools** -> **NuGet Package Manger** -> **Package Manager Console**), and enter the following command:

```
Install-Package Twilio
```

Setup SMS and Two-Factor Authentication

This tutorial uses Twilio, but you can use any SMS provider.

1. Create a Twilio account.
2. From the **Dashboard** tab of your Twilio account, copy the **Account SID** and **Auth Token.** You will add them to your app later.
3. From the **Numbers** tab, copy your Twilio **phone number** as well.
4. Make the Twilio **Account SID**, **Auth Token** and **phone number** available to the app. To keep things simple you will store these values in the *web.config* file. When you deploy to Azure, you can store the values securely in the **appSettings** section on the web site configure tab. Also, when adding the phone number, only use numbers.
   Notice that you can also add SendGrid credentials. SendGrid is an email notification service. For details about how to enable SendGrid, see the 'Hook Up SendGrid' section of the tutorial titled Create a Secure ASP.NET Web Forms App with user registration, email confirmation and password reset.

```
</connectionStrings>
  <appSettings>
    <!-- SendGrid Credentials-->
    <add key="emailServiceUserName" value="[EmailServiceAccountUserName]" />
    <add key="emailServicePassword" value="[EmailServiceAccountPassword]" />
    <!-- Twilio Credentials-->
    <add key="SMSSID" value="[SMSServiceAccountSID]" />
    <add key="SMSAuthToken" value="[SMSServiceAuthorizationToken]" />
    <add key="SMSPhoneNumber" value="+[SMSPhoneNumber]" />
  </appSettings>
  <system.web>
```

WARNING

Security - Never store sensitive data in your source code. In this example, the account and credentials are stored in the **appSettings** section of the *Web.config* file. On Azure, you can securely store these values on the **Configure** tab in the Azure portal. For related information, see Rick Anderson's topic titled Best practices for deploying passwords and other sensitive data to ASP.NET and Azure.

5. Configure the `SmsService` class in the *App_Start\IdentityConfig.cs* file by making the following changes highlighted in yellow:

```
public class SmsService : IIdentityMessageService
{
    public Task SendAsync(IdentityMessage message)
    {
        var Twilio = new TwilioRestClient(
           ConfigurationManager.AppSettings["SMSSID"],
           ConfigurationManager.AppSettings["SMSAuthToken"]
        );
        var result = Twilio.SendMessage(
            ConfigurationManager.AppSettings["SMSPhoneNumber"],
           message.Destination, message.Body);

        // Status is one of Queued, Sending, Sent, Failed or null if the number is not valid
        Trace.TraceInformation(result.Status);

        // Twilio doesn't currently have an async API, so return success.
        return Task.FromResult(0);
    }
}
```

6. Add the following `using` statements to the beginning of the *IdentityConfig.cs* file:

```
using Twilio;
using System.Net;
using System.Configuration;
using System.Diagnostics;
```

7. Update the *Account/Manage.aspx* file by removing the lines highlighted in yellow:

```
<%@ Page Title="Manage Account" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="Manage.aspx.cs" Inherits="WebFormsTwoFactor.Account.Manage" %>

<%@ Register Src="~/Account/OpenAuthProviders.ascx" TagPrefix="uc" TagName="OpenAuthProviders" %>

<asp:Content ContentPlaceHolderID="MainContent" runat="server">
    <h2><%: Title %>.</h2>
```

```
    <div>
        <asp:PlaceHolder runat="server" ID="successMessage" Visible="false" ViewStateMode="Disabled">
            <p class="text-success"><%: SuccessMessage %></p>
        </asp:PlaceHolder>
    </div>

    <div class="row">
        <div class="col-md-12">
            <div class="form-horizontal">
                <h4>Change your account settings</h4>
                <hr />
                <dl class="dl-horizontal">
                    <dt>Password:</dt>
                    <dd>
                        <asp:HyperLink NavigateUrl="/Account/ManagePassword" Text="[Change]"
Visible="false" ID="ChangePassword" runat="server" />
                        <asp:HyperLink NavigateUrl="/Account/ManagePassword" Text="[Create]"
Visible="false" ID="CreatePassword" runat="server" />
                    </dd>
                    <dt>External Logins:</dt>
                    <dd><%: LoginsCount %>
                        <asp:HyperLink NavigateUrl="/Account/ManageLogins" Text="[Manage]"
runat="server" />

                    </dd>
                    <%--
                        Phone Numbers can used as a second factor of verification in a two-factor
authentication system.
                        See <a href="https://go.microsoft.com/fwlink/?LinkId=313242">this article</a>
                        for details on setting up this ASP.NET application to support two-factor
authentication using SMS.
                        Uncomment the following block after you have set up two-factor authentication
                    --%>

                    <dt>Phone Number:</dt>
                    <%--
                    <% if (HasPhoneNumber)
                       { %>
                    <dd>
                        <asp:HyperLink NavigateUrl="/Account/AddPhoneNumber" runat="server" Text="
[Add]" />
                    </dd>
                    <% }
                       else
                       { %>
                    <dd>
                        <asp:Label Text="" ID="PhoneNumber" runat="server" />
                        <asp:HyperLink NavigateUrl="/Account/AddPhoneNumber" runat="server" Text="
[Change]" />  | 
                        <asp:LinkButton Text="[Remove]" OnClick="RemovePhone_Click" runat="server" />
                    </dd>
                    <% } %>
                    --%>

                    <dt>Two-Factor Authentication:</dt>
                    <dd>
                        <p>
                            There are no two-factor authentication providers configured. See <a
href="https://go.microsoft.com/fwlink/?LinkId=313242">this article</a>
                            for details on setting up this ASP.NET application to support two-factor
authentication.
                        </p>
                        <% if (TwoFactorEnabled)
                          { %>
                        <%--
                        Enabled
                        <asp:LinkButton Text="[Disable]" runat="server" CommandArgument="false"
OnClick="TwoFactorDisable_Click" />
                        --%>
```

```
                        <% }
                          else
                          { %>
                        <%--
                        Disabled
                        <asp:LinkButton Text="[Enable]" CommandArgument="true"
OnClick="TwoFactorEnable_Click" runat="server" />
                        --%>
                        <% } %>
                    </dd>
                </dl>
            </div>
        </div>
    </div>
</asp:Content>
```

8. In the `Page_Load` handler of the *Manage.aspx.cs* code-behind, uncomment the line of code highlighted in yellow so that it appears as follows:

```
protected void Page_Load()
{
    var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();

    HasPhoneNumber = String.IsNullOrEmpty(manager.GetPhoneNumber(User.Identity.GetUserId()));

    // Enable this after setting up two-factor authentientication
    PhoneNumber.Text = manager.GetPhoneNumber(User.Identity.GetUserId()) ?? String.Empty;

    TwoFactorEnabled = manager.GetTwoFactorEnabled(User.Identity.GetUserId());

    LoginsCount = manager.GetLogins(User.Identity.GetUserId()).Count;

    var authenticationManager = HttpContext.Current.GetOwinContext().Authentication;

    if (!IsPostBack)
    {
        // Determine the sections to render
        if (HasPassword(manager))
        {
            ChangePassword.Visible = true;
        }
        else
        {
            CreatePassword.Visible = true;
            ChangePassword.Visible = false;
        }

        // Render success message
        var message = Request.QueryString["m"];
        if (message != null)
        {
            // Strip the query string from action
            Form.Action = ResolveUrl("~/Account/Manage");

            SuccessMessage =
                message == "ChangePwdSuccess" ? "Your password has been changed."
                : message == "SetPwdSuccess" ? "Your password has been set."
                : message == "RemoveLoginSuccess" ? "The account was removed."
                : message == "AddPhoneNumberSuccess" ? "Phone number has been added"
                : message == "RemovePhoneNumberSuccess" ? "Phone number was removed"
                : String.Empty;
            successMessage.Visible = !String.IsNullOrEmpty(SuccessMessage);
        }
    }
}
```

9. In the codebehind of *Account/TwoFactorAuthenticationSignIn.aspx.cs*, update the `Page_Load` handler by adding the following code highlighted in yellow:

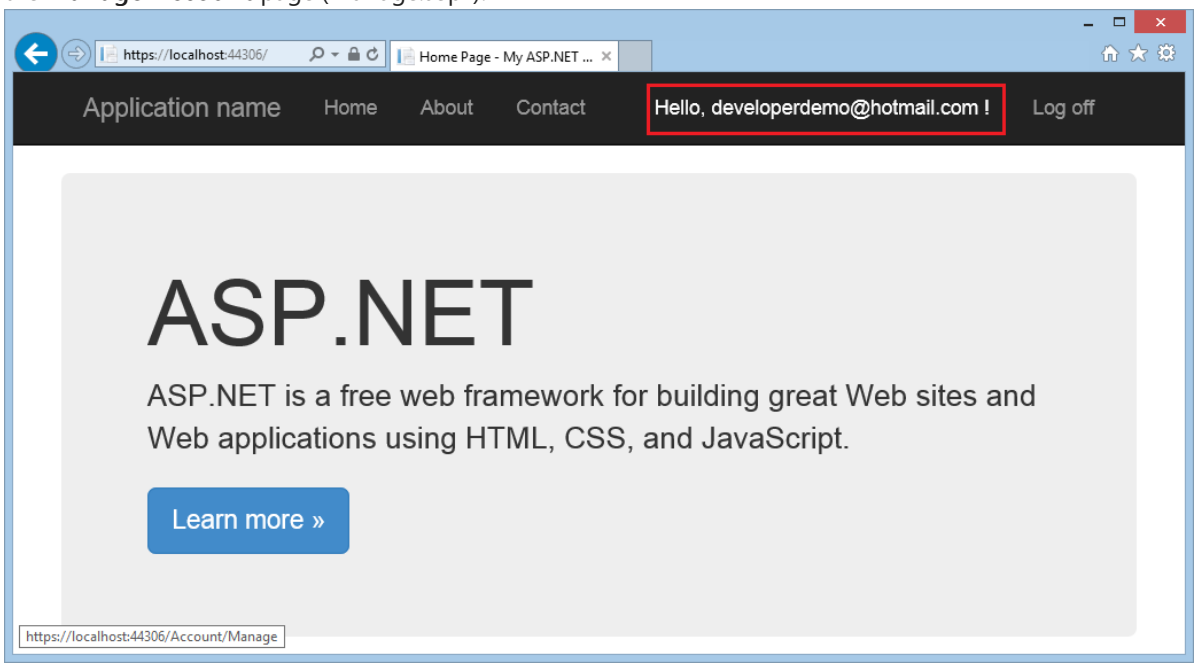
```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        var userId = signinManager.GetVerifiedUserId<ApplicationUser, string>();
        if (userId == null)
        {
            Response.Redirect("/Account/Error", true);
        }
        var userFactors = manager.GetValidTwoFactorProviders(userId);
        Providers.DataSource = userFactors.Select(x => x).ToList();
        Providers.DataBind();
    }
}
```

By making the above code change, the "Providers" DropDownList containing the authentication options will not be reset to the first value. This will allow the user to successfully select all options to use when authenticating, not just the first.

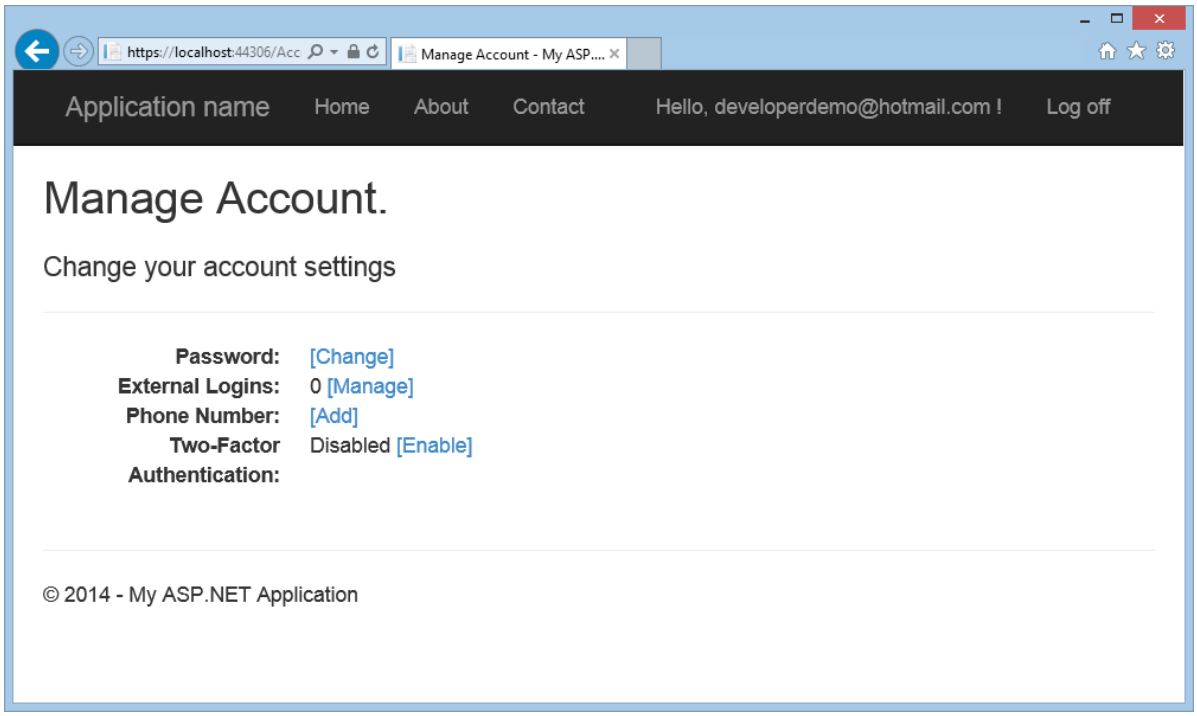
10. In **Solution Explorer**, right-click *Default.aspx* and select **Set As Start Page**.

11. By testing your app, first build the app (**Ctrl+Shift+B**) and then run the app (**F5**) and either select **Register** to create a new user account or select **Log in** if the user account has already been registered.

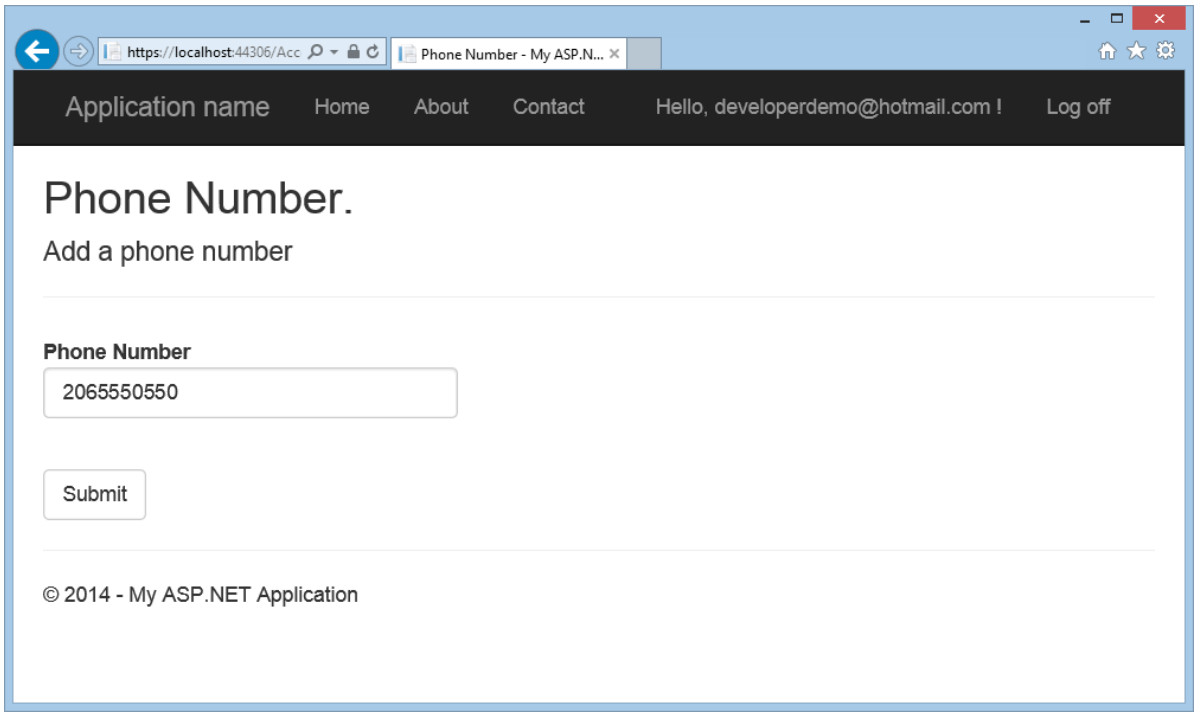
12. Once you (as the user) have logged in, click on the User ID (email address) in the navigation bar to display the **Manage Account** page (Manage.aspx).



13. Click **Add** next to **Phone Number** on the **Manage Account** page.

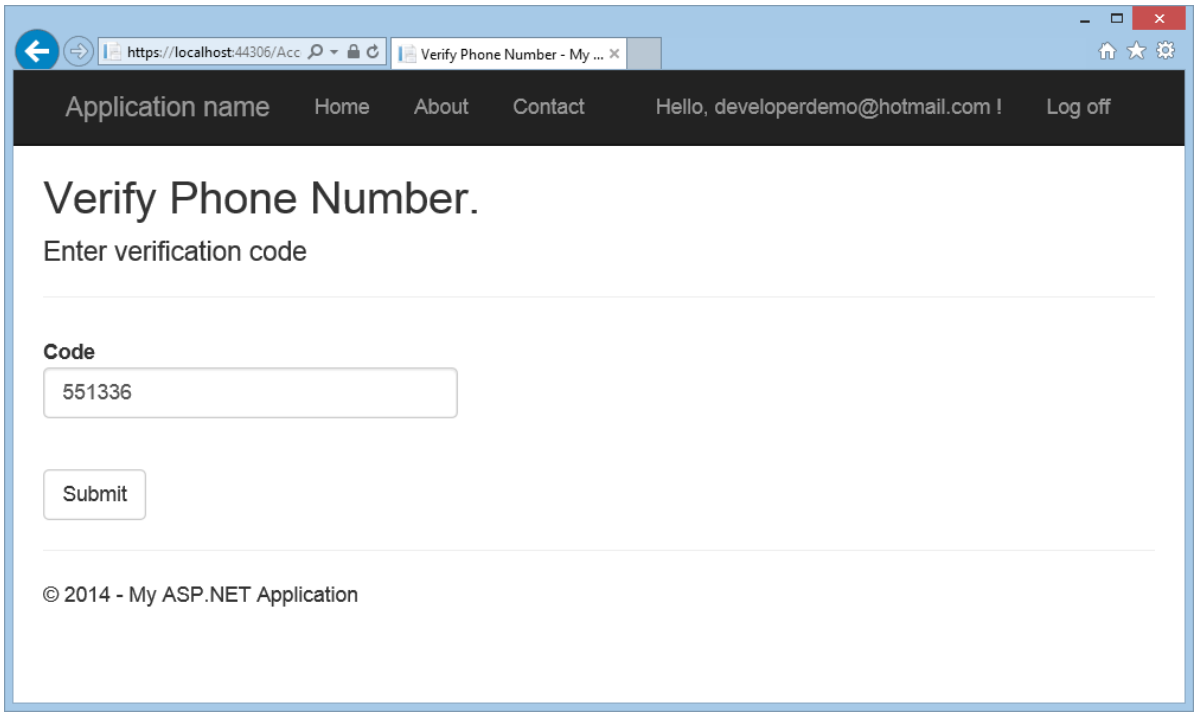


14. Add the phone number where you (as the user) would like to receive SMS messages (text messages) and click the **Submit** button.



At this point, the app will use the credentials from the *Web.config* to contact Twilio. A SMS message (text message) will be sent to the phone associated with the user account. You can verify that the Twilio message was sent by viewing the Twilio dashboard.

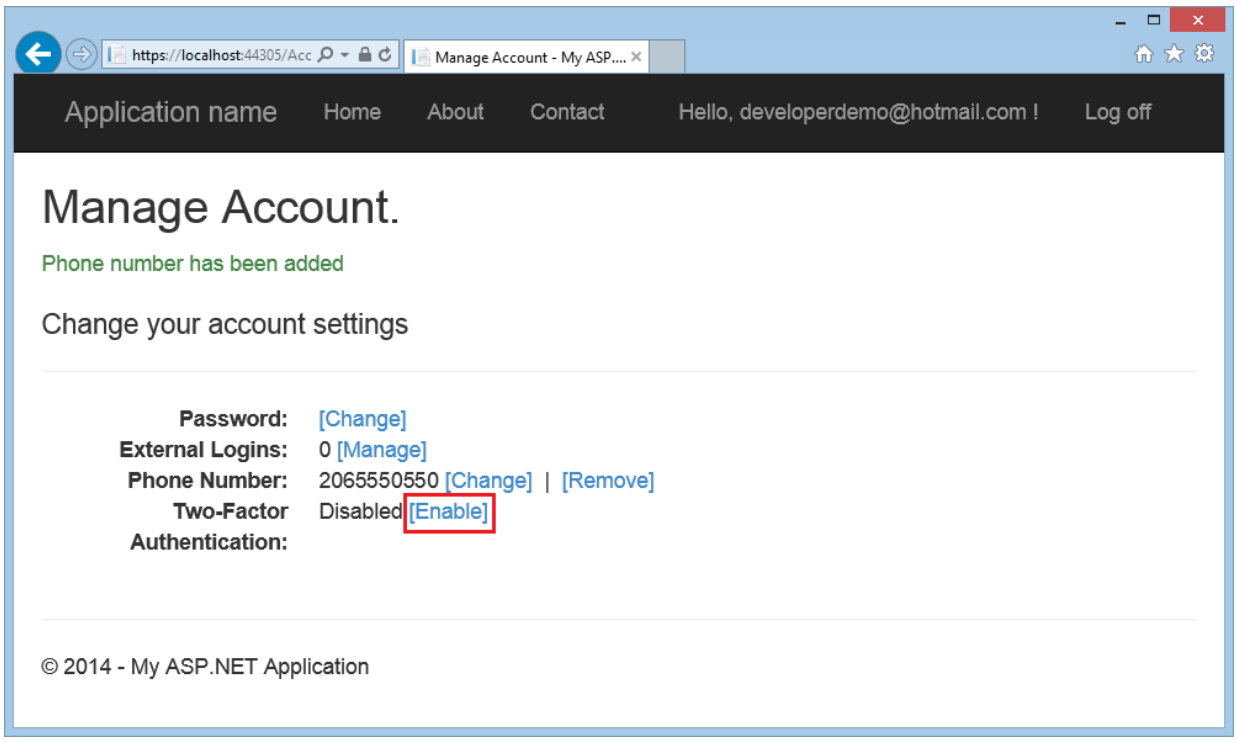
15. In a few seconds, the phone associated with the user account will get a text message containing the verification code. Enter the verification code and press **Submit**.



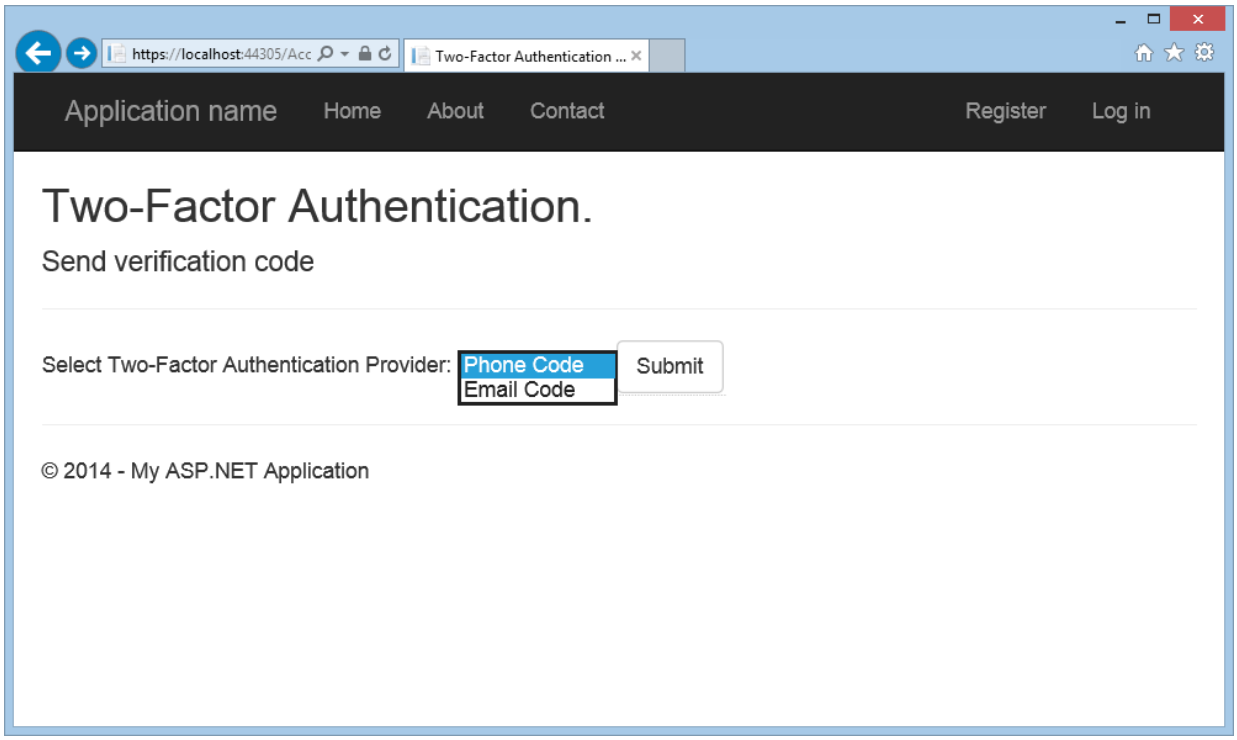
Enable Two-Factor Authentication for a Registered User

At this point, you have enabled two-factor authentication for your app. For a user to use two-factor authentication, they can simply change their settings using the UI.

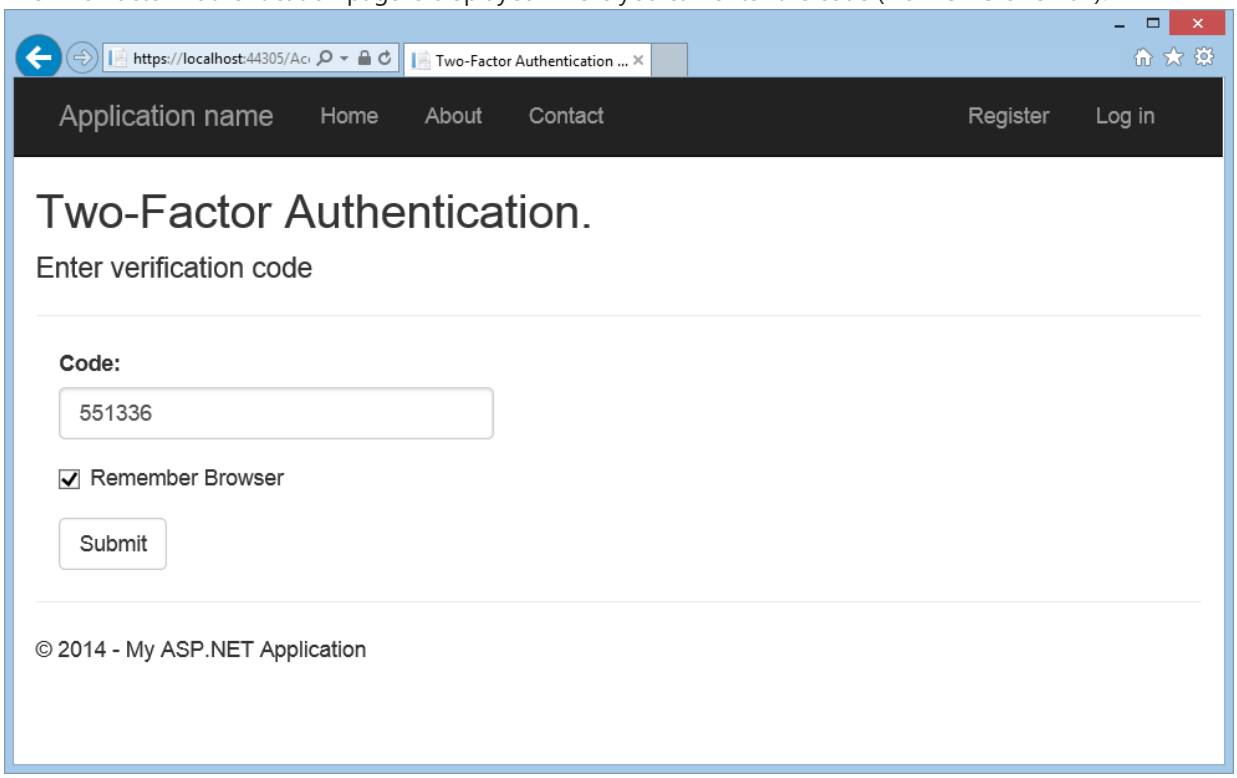
1. As a user of your app you can enable two-factor authentication for your specific account by clicking on the user ID (email alias) in the navigation bar to display the **Manage Account** page.Then, click on the **Enable** link to enable two-factor authentication for the account.



2. Log off, then log back in. If you've enabled email, you can select either SMS or email for two-factor authentication. If you haven't enabled email, see the tutorial titled Create a Secure ASP.NET Web Forms App with User Registration, Email Confirmation and Password Reset.



3. The Two-Factor Authentication page is displayed where you can enter the code (from SMS or email).



Clicking on the **Remember this browser** check box will exempt you from needing to use two-factor authentication to log in when using the browser and device where you checked the box. As long as malicious users can't gain access to your device, enabling two-factor authentication and clicking on the **Remember this browser** will provide you with convenient one step password access, while still retaining strong two-factor authentication protection for all access from non-trusted devices. You can do this on any private device you regularly use.

Additional Resources

- Two-factor authentication using SMS and email with ASP.NET Identity
- Links to ASP.NET Identity Recommended Resources
- Deploy a Secure ASP.NET Web Forms App with Membership, OAuth, and SQL Database to an Azure Website
- ASP.NET Web Forms tutorial series - Add an OAuth 2.0 Provider

- ASP.NET Web Forms tutorial series - Enable SSL for the Project
- Account Confirmation and Password Recovery with ASP.NET Identity
- Creating the app in Facebook and connecting the app to the project

Performance

2/20/2020 • 2 minutes to read • Edit Online

How to cache site content for faster performance.

- Using Asynchronous Methods in ASP.NET 4.5

Using Asynchronous Methods in ASP.NET 4.5

9/16/2020 • 14 minutes to read • Edit Online

by Rick Anderson

This tutorial will teach you the basics of building an asynchronous ASP.NET Web Forms application using Visual Studio Express 2012 for Web, which is a free version of Microsoft Visual Studio. You can also use Visual Studio 2012. The following sections are included in this tutorial.

- How Requests Are Processed by the Thread Pool
- Choosing Synchronous or Asynchronous Methods
- The Sample Application
- The Gizmos Synchronous Page
- Creating an Asynchronous Gizmos Page
- Performing Multiple Operations in Parallel
- Using a Cancellation Token
- Server Configuration for High Concurrency/High Latency Web Service Calls

A complete sample is provided for this tutorial at https://github.com/RickAndMSFT/Async-ASP.NET/ on the GitHub site.

ASP.NET 4.5 Web Pages in combination .NET 4.5 enables you to register asynchronous methods that return an object of type Task. The .NET Framework 4 introduced an asynchronous programming concept referred to as a Task and ASP.NET 4.5 supports Task. Tasks are represented by the **Task** type and related types in the System.Threading.Tasks namespace. The .NET Framework 4.5 builds on this asynchronous support with the await and async keywords that make working with Task objects much less complex than previous asynchronous approaches. The await keyword is syntactical shorthand for indicating that a piece of code should asynchronously wait on some other piece of code. The async keyword represents a hint that you can use to mark methods as task-based asynchronous methods. The combination of **await**, **async**, and the **Task** object makes it much easier for you to write asynchronous code in .NET 4.5. The new model for asynchronous methods is called the *Task-based Asynchronous Pattern* (**TAP**). This tutorial assumes you have some familiarity with asynchronous programing using await and async keywords and the Task namespace.

For more information on the using await and async keywords and the Task namespace, see the following references.

- Whitepaper: Asynchrony in .NET
- Async/Await FAQ
- Visual Studio Asynchronous Programming

How Requests Are Processed by the Thread Pool

On the web server, the .NET Framework maintains a pool of threads that are used to service ASP.NET requests. When a request arrives, a thread from the pool is dispatched to process that request. If the request is processed synchronously, the thread that processes the request is busy while the request is being processed, and that thread cannot service another request.

This might not be a problem, because the thread pool can be made large enough to accommodate many busy threads. However, the number of threads in the thread pool is limited (the default maximum for .NET 4.5 is 5,000). In large applications with high concurrency of long-running requests, all available threads might be busy. This

condition is known as thread starvation. When this condition is reached, the web server queues requests. If the request queue becomes full, the web server rejects requests with an HTTP 503 status (Server Too Busy). The CLR thread pool has limitations on new thread injections. If concurrency is bursty (that is, your web site can suddenly get a large number of requests) and all available request threads are busy because of backend calls with high latency, the limited thread injection rate can make your application respond very poorly. Additionally, each new thread added to the thread pool has overhead (such as 1 MB of stack memory). A web application using synchronous methods to service high latency calls where the thread pool grows to the .NET 4.5 default maximum of 5, 000 threads would consume approximately 5 GB more memory than an application able the service the same requests using asynchronous methods and only 50 threads. When you're doing asynchronous work, you're not always using a thread. For example, when you make an asynchronous web service request, ASP.NET will not be using any threads between the **async** method call and the **await**. Using the thread pool to service requests with high latency can lead to a large memory footprint and poor utilization of the server hardware.

Processing Asynchronous Requests

In web applications that see a large number of concurrent requests at start-up or has a bursty load (where concurrency increases suddenly), making web service calls asynchronous will increase the responsiveness of your application. An asynchronous request takes the same amount of time to process as a synchronous request. For example, if a request makes a web service call that requires two seconds to complete, the request takes two seconds whether it is performed synchronously or asynchronously. However, during an asynchronous call, a thread is not blocked from responding to other requests while it waits for the first request to complete. Therefore, asynchronous requests prevent request queuing and thread pool growth when there are many concurrent requests that invoke long-running operations.

Choosing Synchronous or Asynchronous Methods

This section lists guidelines for when to use synchronous or asynchronous Methods. These are just guidelines; examine each application individually to determine whether asynchronous methods help with performance.

In general, use synchronous methods for the following conditions:

- The operations are simple or short-running.
- Simplicity is more important than efficiency.
- The operations are primarily CPU operations instead of operations that involve extensive disk or network overhead. Using asynchronous methods on CPU-bound operations provides no benefits and results in more overhead.

In general, use asynchronous methods for the following conditions:

- You're calling services that can be consumed through asynchronous methods, and you're using .NET 4.5 or higher.
- The operations are network-bound or I/O-bound instead of CPU-bound.
- Parallelism is more important than simplicity of code.
- You want to provide a mechanism that lets users cancel a long-running request.
- When the benefit of switching threads outweighs the cost of the context switch. In general, you should make a method asynchronous if the synchronous method blocks the ASP.NET request thread while doing no work. By making the call asynchronous, the ASP.NET request thread is not blocked doing no work while it waits for the web service request to complete.
- Testing shows that the blocking operations are a bottleneck in site performance and that IIS can service more requests by using asynchronous methods for these blocking calls.

> The downloadable sample shows how to use asynchronous methods effectively. The sample provided was designed to provide a simple demonstration of asynchronous programming in ASP.NET 4.5. The sample is not intended to be a reference architecture for asynchronous programming in ASP.NET. The sample program calls ASP.NET Web API methods which in turn call Task.Delay to simulate long-running web service calls. Most production applications will not show such obvious benefits to using asynchronous Methods.

Few applications require all methods to be asynchronous. Often, converting a few synchronous methods to asynchronous methods provides the best efficiency increase for the amount of work required.

The Sample Application

You can download the sample application from https://github.com/RickAndMSFT/Async-ASP.NET on the GitHub site. The repository consists of three projects:

- *WebAppAsync*: The ASP.NET Web Forms project that consumes the Web API **WebAPIpwg** service. Most of the code for this tutorial is from the this project.
- *WebAPIpgw*: The ASP.NET MVC 4 Web API project that implements the `Products, Gizmos and Widgets` controllers. It provides the data for the *WebAppAsync* project and the *Mvc4Async* project.
- *Mvc4Async*: The ASP.NET MVC 4 project that contains the code used in another tutorial. It makes Web API calls to the **WebAPIpwg** service.

The Gizmos Synchronous Page

The following code shows the `Page_Load` synchronous method that is used to display a list of gizmos. (For this article, a gizmo is a fictional mechanical device.)

```
public partial class Gizmos : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        var gizmoService = new GizmoService();
        GizmoGridView.DataSource = gizmoService.GetGizmos();
        GizmoGridView.DataBind();
    }
}
```

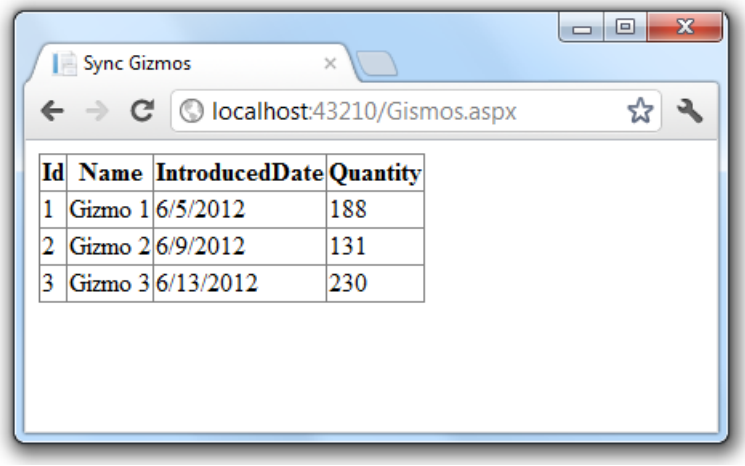
The following code shows the `GetGizmos` method of the gizmo service.

```
public class GizmoService
{
    public async Task<List<Gizmo>> GetGizmosAsync(
        // Implementation removed.

    public List<Gizmo> GetGizmos()
    {
        var uri = Util.getServiceUri("Gizmos");
        using (WebClient webClient = new WebClient())
        {
            return JsonConvert.DeserializeObject<List<Gizmo>>(
                webClient.DownloadString(uri)
            );
        }
    }
}
```

The `GizmoService GetGizmos` method passes a URI to an ASP.NET Web API HTTP service which returns a list of

gizmos data. The *WebAPIpgw* project contains the implementation of the Web API `gizmos, widget` and `product` controllers.
The following image shows the gizmos page from the sample project.



Creating an Asynchronous Gizmos Page

The sample uses the new async and await keywords (available in .NET 4.5 and Visual Studio 2012) to let the compiler be responsible for maintaining the complicated transformations necessary for asynchronous programming. The compiler lets you write code using the C#'s synchronous control flow constructs and the compiler automatically applies the transformations necessary to use callbacks in order to avoid blocking threads.

ASP.NET asynchronous pages must include the Page directive with the `Async` attribute set to "true". The following code shows the Page directive with the `Async` attribute set to "true" for the *GizmosAsync.aspx* page.

```
<%@ Page Async="true"  Language="C#" AutoEventWireup="true"
    CodeBehind="GizmosAsync.aspx.cs" Inherits="WebAppAsync.GizmosAsync" %>
```

The following code shows the `Gizmos` synchronous `Page_Load` method and the `GizmosAsync` asynchronous page. If your browser supports the HTML 5 <mark> element, you'll see the changes in `GizmosAsync` in yellow highlight.

```
protected void Page_Load(object sender, EventArgs e)
{
   var gizmoService = new GizmoService();
   GizmoGridView.DataSource = gizmoService.GetGizmos();
   GizmoGridView.DataBind();
}
```

The asynchronous version:

```
protected void Page_Load(object sender, EventArgs e)
{
    RegisterAsyncTask(new PageAsyncTask(GetGizmosSvcAsync));
}

private async Task GetGizmosSvcAsync()
{
    var gizmoService = new GizmoService();
    GizmosGridView.DataSource = await gizmoService.GetGizmosAsync();
    GizmosGridView.DataBind();
}
```

The following changes were applied to allow the `GizmosAsync` page be asynchronous.

- The Page directive must have the `Async` attribute set to "true".
- The `RegisterAsyncTask` method is used to register an asynchronous task containing the code which runs asynchronously.
- The new `GetGizmosSvcAsync` method is marked with the async keyword, which tells the compiler to generate callbacks for parts of the body and to automatically create a `Task` that is returned.
- "Async" was appended to the asynchronous method name. Appending "Async" is not required but is the convention when writing asynchronous methods.
- The return type of the new `GetGizmosSvcAsync` method is `Task`. The return type of `Task` represents ongoing work and provides callers of the method with a handle through which to wait for the asynchronous operation's completion.
- The await keyword was applied to the web service call.
- The asynchronous web service API was called (`GetGizmosAsync`).

Inside of the `GetGizmosSvcAsync` method body another asynchronous method, `GetGizmosAsync` is called. `GetGizmosAsync` immediately returns a `Task<List<Gizmo>>` that will eventually complete when the data is available. Because you don't want to do anything else until you have the gizmo data, the code awaits the task (using the **await** keyword). You can use the **await** keyword only in methods annotated with the **async** keyword.

The **await** keyword does not block the thread until the task is complete. It signs up the rest of the method as a callback on the task, and immediately returns. When the awaited task eventually completes, it will invoke that callback and thus resume the execution of the method right where it left off. For more information on using the await and async keywords and the Task namespace, see the async references.

The following code shows the `GetGizmos` and `GetGizmosAsync` methods.

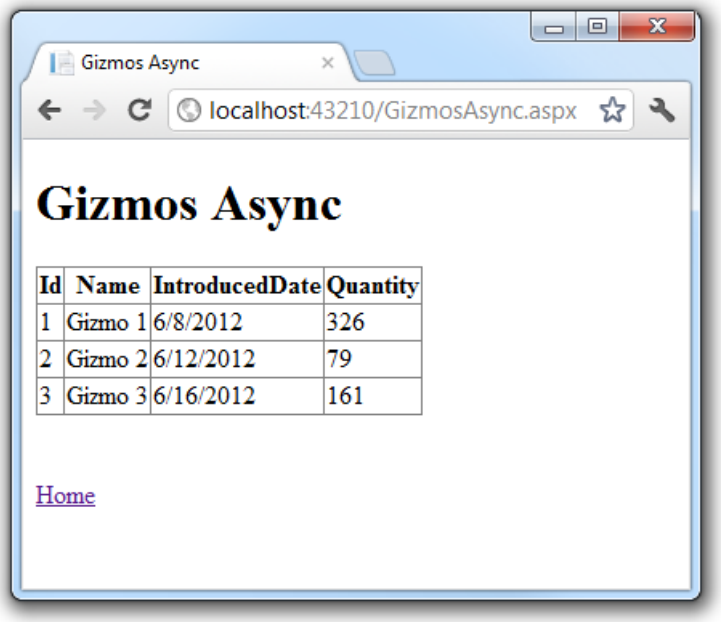
```
public List<Gizmo> GetGizmos()
{
    var uri = Util.getServiceUri("Gizmos");
    using (WebClient webClient = new WebClient())
    {
        return JsonConvert.DeserializeObject<List<Gizmo>>(
            webClient.DownloadString(uri)
        );
    }
}
```

```
public async Task<List<Gizmo>> GetGizmosAsync()
{
    var uri = Util.getServiceUri("Gizmos");
    using (WebClient webClient = new WebClient())
    {
        return JsonConvert.DeserializeObject<List<Gizmo>>(
            await webClient.DownloadStringTaskAsync(uri)
        );
    }
}
```

The asynchronous changes are similar to those made to the **GizmosAsync** above.

- The method signature was annotated with the async keyword, the return type was changed to `Task<List<Gizmo>>`, and *Async* was appended to the method name.
- The asynchronous HttpClient class is used instead of the synchronous WebClient class.
- The await keyword was applied to the HttpClientGetAsync asynchronous method.

The following image shows the asynchronous gizmo view.



The browsers presentation of the gizmos data is identical to the view created by the synchronous call. The only difference is the asynchronous version may be more performant under heavy loads.

RegisterAsyncTask Notes

Methods hooked up with `RegisterAsyncTask` will run immediately after PreRender.

If you use async void page events directly, as shown in the following code:

```
protected async void Page_Load(object sender, EventArgs e) {
    await ...;
    // do work
}
```

you no longer have full control over when events execute. For example, if both an .aspx and a .Master define `Page_Load` events and one or both of them are asynchronous, the order of execution can't be guaranteed. The same indeterminate order for event handlers (such as `async void Button_Click`) applies.

Performing Multiple Operations in Parallel

Asynchronous Methods have a significant advantage over synchronous methods when an action must perform several independent operations. In the sample provided, the synchronous page *PWG.aspx*(for Products, Widgets and Gizmos) displays the results of three web service calls to get a list of products, widgets, and gizmos. The ASP.NET Web API project that provides these services uses Task.Delay to simulate latency or slow network calls. When the delay is set to 500 milliseconds, the asynchronous *PWGasync.aspx* page takes a little over 500 milliseconds to complete while the synchronous `PWG` version takes over 1,500 milliseconds. The synchronous *PWG.aspx* page is shown in the following code.

```
protected void Page_Load(object sender, EventArgs e)
{
    Stopwatch stopWatch = new Stopwatch();
    stopWatch.Start();

    var widgetService = new WidgetService();
    var prodService = new ProductService();
    var gizmoService = new GizmoService();

    var pwgVM = new ProdGizWidgetVM(
        widgetService.GetWidgets(),
        prodService.GetProducts(),
        gizmoService.GetGizmos()
       );
    WidgetGridView.DataSource = pwgVM.widgetList;
    WidgetGridView.DataBind();
    ProductGridView.DataSource = pwgVM.prodList;
    ProductGridView.DataBind();
    GizmoGridView.DataSource = pwgVM.gizmoList;
    GizmoGridView.DataBind();

    stopWatch.Stop();
    ElapsedTimeLabel.Text = String.Format("Elapsed time: {0}",
        stopWatch.Elapsed.Milliseconds / 1000.0);
}
```

The asynchronous `PWGasync` code behind is shown below.

```
protected void Page_Load(object sender, EventArgs e)
{
    Stopwatch stopWatch = new Stopwatch();
    stopWatch.Start();
    RegisterAsyncTask(new PageAsyncTask(GetPWGsrvAsync));
    stopWatch.Stop();
    ElapsedTimeLabel.Text = String.Format("Elapsed time: {0}",
        stopWatch.Elapsed.Milliseconds / 1000.0);
}

private async Task GetPWGsrvAsync()
{
    var widgetService = new WidgetService();
    var prodService = new ProductService();
    var gizmoService = new GizmoService();

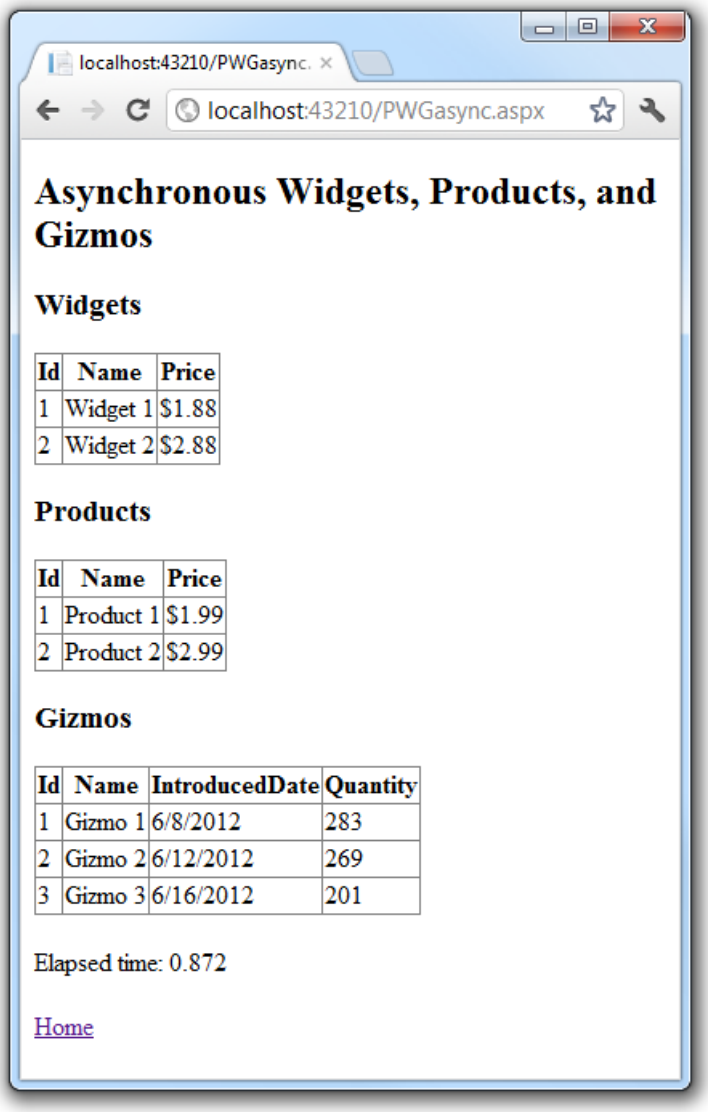
    var widgetTask = widgetService.GetWidgetsAsync();
    var prodTask = prodService.GetProductsAsync();
    var gizmoTask = gizmoService.GetGizmosAsync();

    await Task.WhenAll(widgetTask, prodTask, gizmoTask);

    var pwgVM = new ProdGizWidgetVM(
       widgetTask.Result,
       prodTask.Result,
       gizmoTask.Result
       );

    WidgetGridView.DataSource = pwgVM.widgetList;
    WidgetGridView.DataBind();
    ProductGridView.DataSource = pwgVM.prodList;
    ProductGridView.DataBind();
    GizmoGridView.DataSource = pwgVM.gizmoList;
    GizmoGridView.DataBind();
}
```

The following image shows the view returned from the asynchronous *PWGasync.aspx* page.



Using a Cancellation Token

Asynchronous Methods returning `Task` are cancelable, that is they take a CancellationToken parameter when one is provided with the `AsyncTimeout` attribute of the Page directive. The following code shows the *GizmosCancelAsync.aspx* page with a timeout of on second.

```
<%@ Page  Async="true"  AsyncTimeout="1"
    Language="C#" AutoEventWireup="true"
    CodeBehind="GizmosCancelAsync.aspx.cs"
    Inherits="WebAppAsync.GizmosCancelAsync" %>
```

The following code shows the *GizmosCancelAsync.aspx.cs* file.

```
protected void Page_Load(object sender, EventArgs e)
{
    RegisterAsyncTask(new PageAsyncTask(GetGizmosSvcCancelAsync));
}

private async Task GetGizmosSvcCancelAsync(CancellationToken cancellationToken)
{
    var gizmoService = new GizmoService();
    var gizmoList = await gizmoService.GetGizmosAsync(cancellationToken);
    GizmosGridView.DataSource = gizmoList;
    GizmosGridView.DataBind();
}
private void Page_Error(object sender, EventArgs e)
{
    Exception exc = Server.GetLastError();

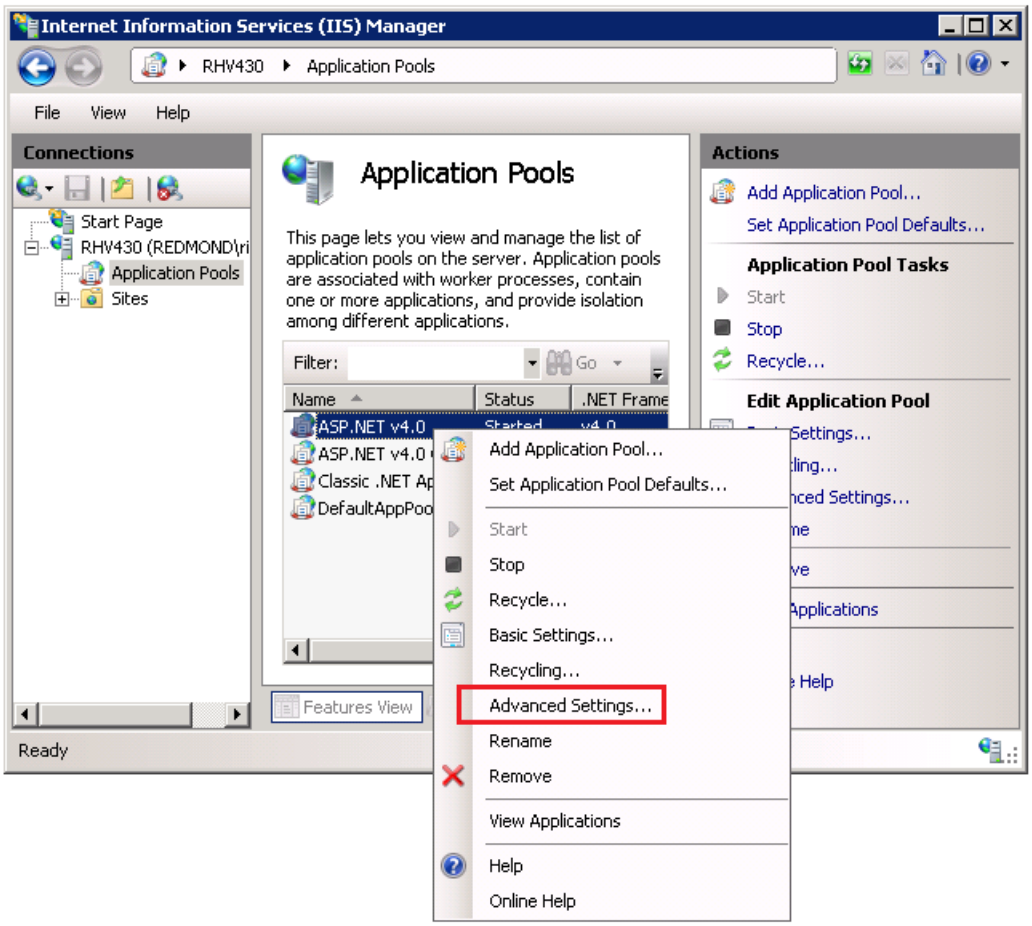
    if (exc is TimeoutException)
    {
        // Pass the error on to the Timeout Error page
        Server.Transfer("TimeoutErrorPage.aspx", true);
    }
}
```

In the sample application provided, selecting the *GizmosCancelAsync* link calls the *GizmosCancelAsync.aspx* page and demonstrates the cancellation (by timing out) of the asynchronous call. Because the delay time is within a random range, you might need to refresh the page a couple times to get the time out error message.

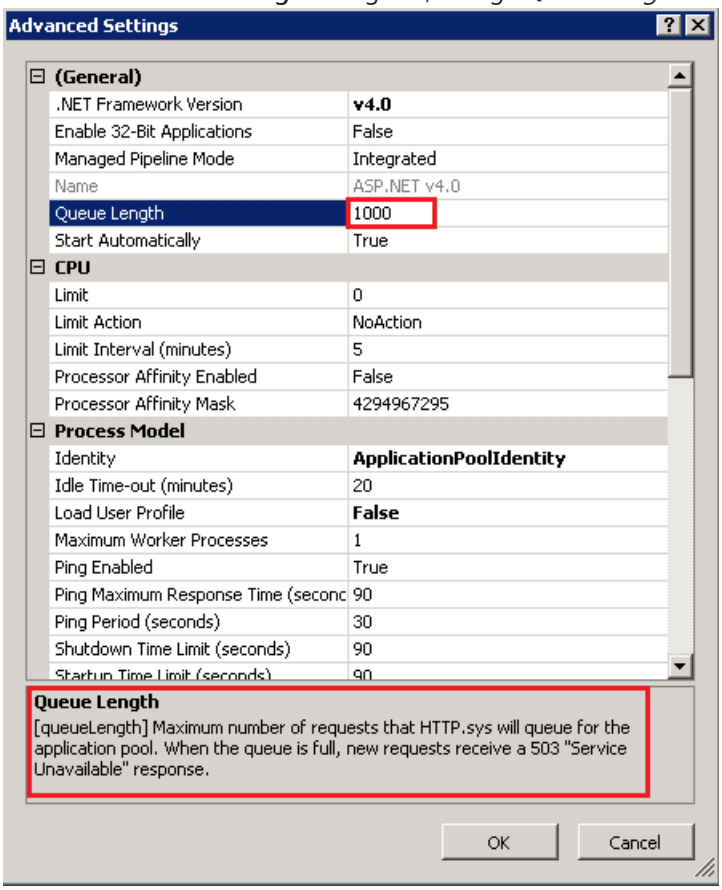
Server Configuration for High Concurrency/High Latency Web Service Calls

To realize the benefits of an asynchronous web application, you might need to make some changes to the default server configuration. Keep the following in mind when configuring and stress testing your asynchronous web application.

- Windows 7, Windows Vista, Window 8, and all Windows client operating systems have a maximum of 10 concurrent requests. You'll need a Windows Server operating system to see the benefits of asynchronous methods under high load.
- Register .NET 4.5 with IIS from an elevated command prompt using the following command:
  %windir%\Microsoft.NET\Framework64 \v4.0.30319\aspnet_regiis -i
  See ASP.NET IIS Registration Tool (Aspnet_regiis.exe)
- You might need to increase the HTTP.sys queue limit from the default value of 1,000 to 5,000. If the setting is too low, you may see HTTP.sys reject requests with a HTTP 503 status. To change the HTTP.sys queue limit:
  - Open IIS manager and navigate to the Application Pools pane.
  - Right click on the target application pool and select **Advanced Settings**.



- In the **Advanced Settings** dialog box, change *Queue Length* from 1,000 to 5,000.



Note in the images above, the .NET framework is listed as v4.0, even though the application pool is using .NET 4.5. To understand this discrepancy, see the following:

- .NET Versioning and Multi-Targeting - .NET 4.5 is an in-place upgrade to .NET 4.0
- How to set an IIS Application or AppPool to use ASP.NET 3.5 rather than 2.0

- .NET Framework Versions and Dependencies
- If your application is using web services or System.NET to communicate with a backend over HTTP you may need to increase the connectionManagement/maxconnection element. For ASP.NET applications, this is limited by the autoConfig feature to 12 times the number of CPUs. That means that on a quad-proc, you can have at most 12 * 4 = 48 concurrent connections to an IP end point. Because this is tied to autoConfig, the easiest way to increase `maxconnection` in an ASP.NET application is to set System.Net.ServicePointManager.DefaultConnectionLimit programmatically in the from `Application_Start` method in the *global.asax* file. See the sample download for an example.
- In .NET 4.5, the default of 5000 for MaxConcurrentRequestsPerCPU should be fine.

Contributors

- Levi Broderick
- Tom Dykstra
- Brad Wilson
- HongMei Ge

Deployment

2/20/2020 • 2 minutes to read • Edit Online

How to deploy ASP.NET Web Forms projects to a server.

- ASP.NET Web Deployment using Visual Studio
  - Introduction
  - Preparing for Database Deployment
  - Web.config File Transformations
  - Project Properties
  - Deploying to Test
  - Setting Folder Permissions
  - Deploying to Production
  - Deploying a Code Update
  - Deploying a Database Update
  - Command Line Deployment
  - Deploying Extra Files
  - Troubleshooting
- ASP.NET 4 - Enterprise Deployment Introduction
  - Deploying Web Applications in Enterprise Scenarios using Visual Studio 2010
  - Enterprise Web Deployment: Scenario Overview
  - Application Lifecycle Management: From Development to Production
- ASP.NET 4 - Enterprise Deployment Series 1
  - Web Deployment in the Enterprise
  - The Contact Manager Solution
  - Setting Up the Contact Manager Solution
  - Understanding the Project File
  - Understanding the Build Process
  - Building and Packaging Web Application Projects
  - Configuring Parameters for Web Package Deployment
  - Deploying Web Packages
  - Deploying Database Projects
  - Creating and Running a Deployment Command File
  - Manually Installing Web Packages
- ASP.NET 4 - Enterprise Deployment Series 2 Configuring Servers
  - Configuring Server Environments for Web Deployment
  - Choosing the Right Approach to Web Deployment
  - Scenario: Configuring a Test Environment for Web Deployment
  - Scenario: Configuring a Staging Environment for Web Deployment
  - Scenario: Configuring a Production Environment for Web Deployment
  - Configuring a Web Server for Web Deploy Publishing (Remote Agent)
  - Configuring a Web Server for Web Deploy Publishing (Web Deploy Handler)

  - Configuring a Web Server for Web Deploy Publishing (Offline Deployment)
  - Configuring a Database Server for Web Deploy Publishing
  - Creating a Server Farm with the Web Farm Framework
  - Configuring Deployment Properties for a Target Environment
- ASP.NET 4 - Enterprise Deployment Series 3 Configuring TFS
  - Configuring Team Foundation Server for Web Deployment
  - Creating a Team Project in TFS
  - Adding Content to Source Control
  - Configuring a TFS Build Server for Web Deployment
  - Creating a Build Definition That Supports Deployment
  - Deploying a Specific Build
  - Configuring Permissions for Team Build Deployment
- ASP.NET 4 - Enterprise Deployment Series 4 Advanced Topics
  - Advanced Enterprise Web Deployment
  - Performing a "What If" Deployment
  - Customizing Database Deployments for Multiple Environments
  - Deploying Database Role Memberships to Test Environments
  - Deploying Membership Databases to Enterprise Environments
  - Excluding Files and Folders from Deployment
  - Taking Web Applications Offline with Web Deploy
  - Running Windows PowerShell Scripts from MSBuild Project Files
  - Troubleshooting the Packaging Process

ASP.NET Web Deployment using Visual Studio

2/20/2020 • 2 minutes to read • Edit Online

This tutorial series shows how to make an ASP.NET web application that uses SQL Server (not SQL Server Compact) available over the internet by deploying it to a Windows Azure Web Site or a third-party hosting provider. Requires Visual Studio 2012 and the ASP.NET and Web Tools 2012.2 Update, or Visual Studio 2010. For information specific to SQL Server Compact, see ASP.NET Web Deployment with SQL Server Compact using Visual Studio.

- Introduction
- Preparing for Database Deployment
- Web.config File Transformations
- Project Properties
- Deploying to Test
- Setting Folder Permissions
- Deploying to Production
- Deploying a Code Update
- Deploying a Database Update
- Command Line Deployment
- Deploying Extra Files
- Troubleshooting

ASP.NET Web Deployment using Visual Studio: Introduction

2/20/2020 • 12 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

This tutorial series shows you how to deploy (publish) an ASP.NET web application to Azure App Service Web Apps or a third-party hosting provider, by using Visual Studio 2012 with the Azure SDK for .NET. Most of the procedures are similar for Visual Studio 2013.

You develop a web application in order to make it available to people over the Internet. But web programming tutorials typically stop right after they've shown you how to get something working on your development computer. This series of tutorials begins where the others leave off: you've built a web app, tested it, and it's ready to go. What's next? These tutorials show you how to deploy first to IIS on your local development computer for testing, and then to Azure or a third-party hosting provider for staging and production. The sample application that you'll deploy is a web application project that uses the Entity Framework, SQL Server, and the ASP.NET membership system. The sample application uses ASP.NET Web Forms, but the procedures shown apply also to ASP.NET MVC and Web API.

These tutorials assume you know how to work with ASP.NET in Visual Studio. If you don't, a good place to start is a basic ASP.NET Web Forms Tutorial or a basic ASP.NET MVC Tutorial.

If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET Deployment forum or StackOverflow.

This content is also available as a free e-book in the TechNet E-Book Gallery.

Overview

These tutorials guide you through deploying an ASP.NET web application that includes SQL Server databases. You'll deploy first to IIS on your local development computer for testing, and then to Web Apps in Azure App Service and Azure SQL Database for staging and production. You'll see how to deploy by using Visual Studio one-click publish, and you'll see how to deploy using the command line.

The number of tutorials might make the deployment process seem daunting. In fact, the basic procedures are simple. However, in real-world situations, you often need to do extra deployment tasks — for example, setting folder permissions on the target server. We've illustrated some of these additional tasks, in the hope that the tutorials don't leave out information that might prevent you from successfully deploying a real application.

The tutorials are designed to run in sequence, and each part builds on the previous part. You can skip parts that aren't relevant to your situation, but then you might have to adjust the procedures in later tutorials.

Intended audience

The tutorials are aimed at ASP.NET developers who work in environments where:

- The production environment is Azure App Service Web Apps or a third-party hosting provider.
- Deployment is not limited to a continuous integration process but may be done directly from Visual Studio.

Deployment from source control using a continuous delivery process is not covered in these tutorials except for one tutorial that shows how to deploy from the command line. For information about continuous delivery, see the following resources:

- Continuous Integration and Continuous Delivery (Building Real-World Cloud Apps with Windows Azure)
- Deploy a web app in Azure App Service
- Deploying Web Applications in Enterprise Scenarios (An older set of tutorials written for Visual Studio 2010, which still has useful information for enterprise environments.)

Using a third-party hosting provider

The tutorials take you through the process of setting up an Azure account and deploying the application to Web Apps in Azure App Service for staging and production. However, you can use the same basic procedures for deploying to a third-party hosting provider of your choice. Where the tutorials go over processes unique to Azure, they explain that and advise what differences you can expect at a third-party hosting provider.

Deploying web app projects

The sample application that you download and deploy for these tutorials is a Visual Studio web application project. However, if you install the latest Web Publish Update for Visual Studio, you can use the same deployment methods and tools for web app projects.

Deploying ASP.NET MVC projects

The sample application is an ASP.NET Web Forms project, but everything you learn how to do is applicable to ASP.NET MVC as well. A Visual Studio MVC project is just another form of web application project. The only difference is that if you're deploying to a hosting provider that does not support ASP.NET MVC or your target version of it, you must make sure that you have installed the appropriate (MVC 3, MVC 4, or MVC 5) NuGet package in your project.

Programming language

The sample application uses C# but the tutorials do not require knowledge of C#, and the deployment techniques shown by the tutorials are not language-specific.

Database deployment methods

There are three ways that you can deploy a SQL Server database along with web deployment in Visual Studio:

- Entity Framework Code First Migrations
- The dbDacFx Web Deploy provider
- The dbFullSql Web Deploy provider

In this tutorial you will use the first two of these methods. The dbFullSql Web Deploy provider is a legacy method that is no longer recommended except for some specific scenarios such as migrating from SQL Server Compact to SQL Server.

The methods shown in this tutorial are for SQL Server databases, not SQL Server Compact. For information about how to deploy a SQL Server Compact database, see Visual Studio Web Deployment With SQL Server Compact.

The methods shown in this tutorial require that you use the Web Deploy publish method. If you prefer a different publish method, such as FTP, File System, or FPSE, see Deploying a database separately from web application deployment in the Web Deployment Content Map for Visual Studio and ASP.NET.

Entity Framework Code First Migrations

In the Entity Framework version 4.3, Microsoft introduced Code First Migrations. Code First Migrations automates the process of making incremental changes to a data model and propagating those changes to the database. In earlier versions of Code First, you typically let the Entity Framework drop and re-create the database each time you change the data model. This is not a problem in development because test data is easily re-created, but in production you usually want to update the database schema without dropping the database. The Migrations feature enables Code First to update the database without dropping and re-creating it. You can let Code First automatically decide how to make the required schema changes, or you can write code that customizes the changes. For an introduction to Code First Migrations, see Code First Migrations.

When you are deploying a web project, Visual Studio can automate the process of deploying a database that is managed by Code First Migrations. When you create the publish profile, you select a check box that is labeled Execute Code First Migrations (runs on application start). This setting causes the deployment process to automatically configure the application Web.config file on the destination server so that Code First uses the `MigrateDatabaseToLatestVersion` initializer class.

Visual Studio does not do anything with the database during the deployment process. When the deployed application accesses the database for the first time after deployment, Code First automatically creates the database or updates the database schema to the latest version. If the application implements a Migrations Seed method, the method runs after the database is created or the schema is updated.

In this tutorial, you'll use Code First Migrations to deploy the application database.

The dbDacFx Web Deploy provider

For a SQL Server database that isn't managed by Entity Framework Code First, you can select a check box that is labeled Update database when you configure the publish profile. During the initial deployment, the dbDacFx provider creates tables and other database objects in the destination database to match the source database. On subsequent deployments, the provider determines what is different between the source and destination databases, and it updates the schema of the destination database to match the source database. By default, the provider won't make any changes that cause data loss, such as when a table or column is dropped.

This method does not automate the deployment of data in database tables, but you can create scripts to do that and configure Visual Studio to run them during deployment. Another reason to run scripts during deployment is to make schema changes that can't be done automatically because they would cause data loss.

In this tutorial, you'll use the dbDacFx provider to deploy the ASP.NET membership database.

Troubleshooting during this tutorial

When an error happens during deployment, or if the deployed site does not run correctly, the error messages don't always provide an obvious solution. To help you with some common problem scenarios, a troubleshooting reference page is available. If you get an error message or something doesn't work as you go through the tutorials, be sure to check the troubleshooting page.

Comments welcome

Comments on the tutorials are welcome, and when the tutorial is updated every effort will be made to take into account corrections or suggestions for improvements that are provided in tutorial comments.

Prerequisites

This tutorial was written for the following products:

- Windows 8 or Windows 7.

- Visual Studio 2012 or Visual Studio 2012 Express for Web with the latest update.
- Azure SDK for Visual Studio 2012

You can follow the tutorial by using Visual Studio 2010 SP1 or Visual Studio 2013, but some screen shots will be different and some features will be different.

If you're using Visual Studio 2013, install Azure SDK for Visual Studio 2013.

If you're using Visual Studio 2010 SP1, install the following software:

- Azure SDK for Visual Studio 2010
- SQL Server Express LocalDB
- SQL Server Data Tools.

Depending on how many of the SDK dependencies you already have on your machine, installing the Azure SDK could take a long time, from several minutes to a half hour or more. You need the Azure SDK even if you plan to publish to a third-party hosting provider instead of to Azure, because the SDK includes the latest updates to Visual Studio web publish features.

> **NOTE**
>
> This tutorial was written with version 1.8.1 of the Azure SDK. Since then newer versions with additional features have been released. The tutorials have been updated to mention these features and link to resources that have more information about them.

The instructions and screen shots are based on Windows 8, but the tutorials explain differences for Windows 7.

Some other software is required in order to complete the tutorial, but you don't have to have that installed yet. The tutorial will walk you through the steps for installing it when you need it.

Download the sample application

The application that you'll deploy is named Contoso University and has already been created for you. It's a simplified version of a university web site, based loosely on the Contoso University application described in the Entity Framework tutorials on the ASP.NET site.

When you have the prerequisites installed, download the Contoso University web application. The *.zip* file contains multiple versions of the project. To work through the steps of the tutorial, start with the project located in the C# folder. To see what the project looks like at the end of the tutorials, open the project in the ContosoUniversity-End folder.

To prepare the project for working through the tutorial steps, perform the following steps:

1. Save the ContosoUniversity solution files from the C# folder in a folder named ContosoUniversity in whatever folder you use for working with Visual Studio projects.

   By default this is the following folder for Visual Studio 2012:

   ```
   C:\Users\<username>\Documents\Visual Studio 2012\Projects
   ```

   (For the screen shots in this tutorial, the project folder is located in the root directory on the `C`: drive.)

2. Start Visual Studio and open the project.

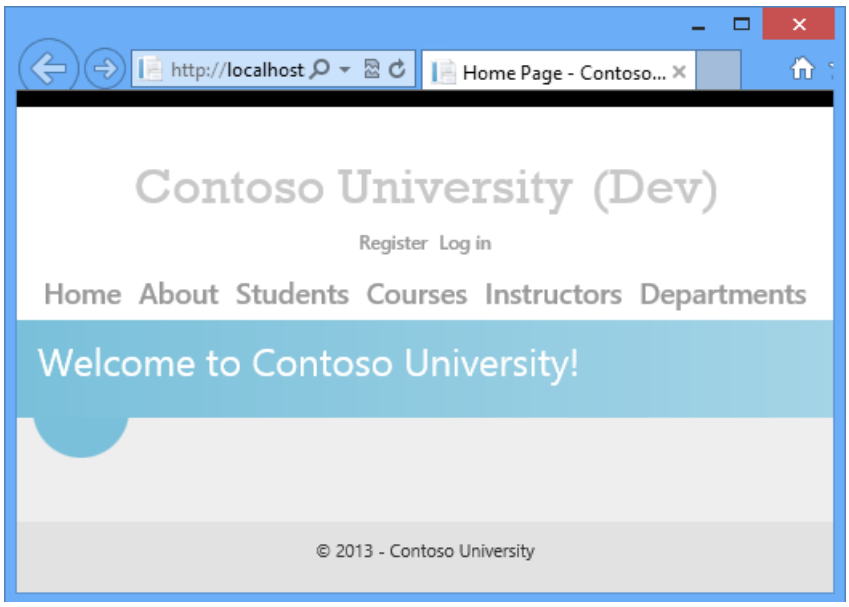
3. In **Solution Explorer**, right-click the solution and click **EnableNuGet Package Restore**.

4. Build the solution.

5. If you get compile errors, manually restore the NuGet packages:

   a. In **Solution Explorer**, right-click the solution, and then click **Manage NuGet Packages for Solution**.
   b. At the top of the **Manage NuGet Packages** dialog box you'll see **Some NuGet packages are missing from this solution. Click to restore.** Click the **Restore** button.
   c. Rebuild the solution.
6. Press CTRL-F5 to run the application.

   The application opens to the Contoso University home page.

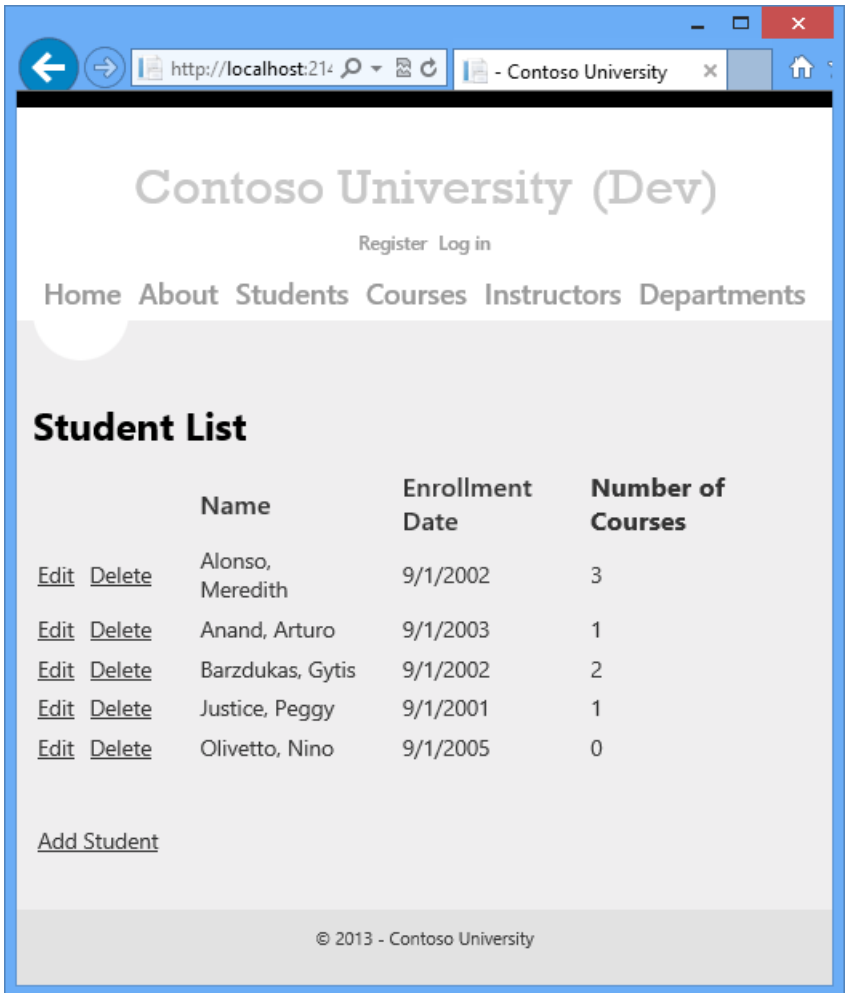


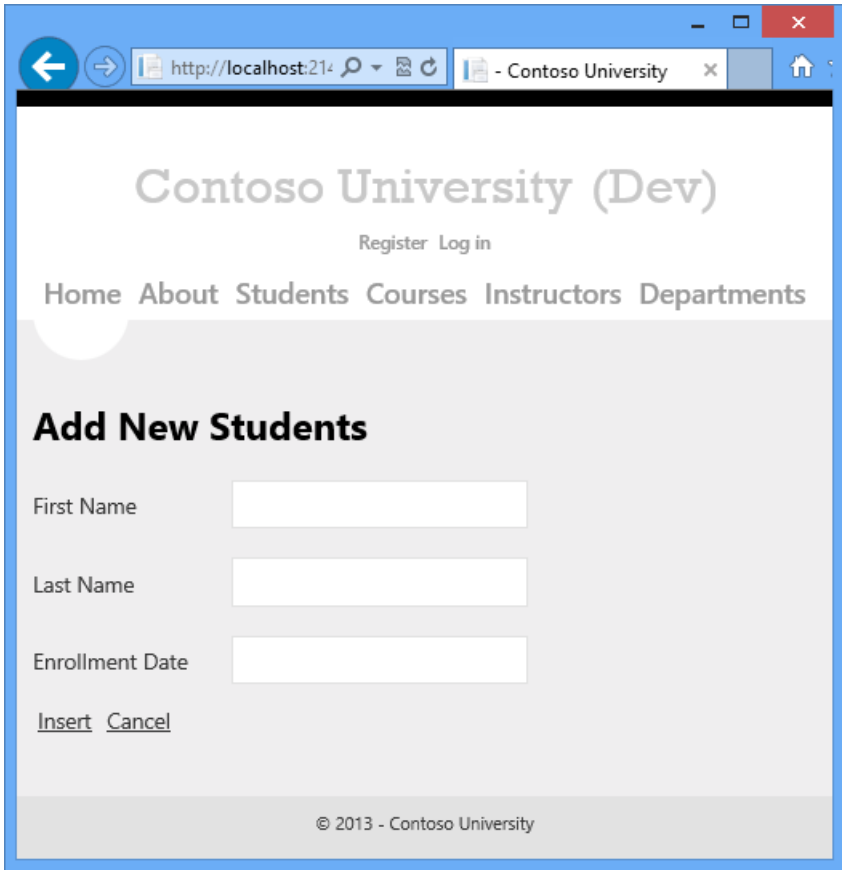
   (There might be a wait time while Visual Studio starts up the SQL Server Express LocalDB instance, and you might get a timeout error if that process takes too long. In that case just start the project again.)

The website pages are accessible from the menu bar and let you perform the following functions:

- Display student statistics (the About page).
- Display, edit, delete, and add students.
- Display and edit courses.
- Display and edit instructors.
- Display and edit departments.

Following are screen shots of a few representative pages.





Review application features that affect deployment

The following features of the application affect how you deploy it or what you have to do to deploy it. Each of these is explained in more detail in the following tutorials in the series.

- Contoso University uses a SQL Server database to store application data such as student and instructor names. The database contains a mix of test data and production data, and when you deploy to production you need to exclude the test data.
- The application uses the ASP.NET membership system, which stores user account information in a SQL Server database. The application defines an administrator user who has access to some restricted information. You need to deploy the membership database without test accounts but with an administrator account.
- The application uses a third-party error logging and reporting utility. This utility is provided in an assembly which must be deployed with the application.
- The error logging utility writes error information in XML files to a file folder. You have to make sure that the account that ASP.NET runs under in the deployed site has write permission to this folder, and you have to exclude this folder from deployment. (Otherwise, error log data from the test environment might be deployed to production and/or production error log files might be deleted.)
- The application includes some settings that must be changed in the deployed *Web.config* file depending on the destination environment (test, staging, or production), and other settings that must be changed depending on the build configuration (Debug or Release).
- The Visual Studio solution includes a class library project. Only the assembly that this project generates should be deployed, not the project itself.

Summary

In this first tutorial in the series, you have downloaded the sample Visual Studio project and reviewed site features that affect how you deploy the application. In the following tutorials, you prepare for deployment by setting up some of these things to be handled automatically. Others you take care of manually.

NEXT

ASP.NET Web Deployment using Visual Studio: Preparing for Database Deployment

2/20/2020 • 15 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This tutorial series shows you how to deploy (publish) an ASP.NET web application to Azure App Service Web Apps or to a third-party hosting provider, by using Visual Studio 2012 or Visual Studio 2010. For information about the series, see the first tutorial in the series.

Overview

This tutorial shows how to get the project ready for database deployment. The database structure and some (not all) of the data in the application's two databases must be deployed to test, staging, and production environments.

Typically, as you develop an application, you enter test data into a database that you don't want to deploy to a live site. However, you might also have some production data that you do want to deploy. In this tutorial you'll configure the Contoso University project and prepare SQL scripts so that the correct data is included when you deploy.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

SQL Server Express LocalDB

The sample application uses SQL Server Express LocalDB. SQL Server Express is the free edition of SQL Server. It is commonly used during development because it is based on the same database engine as full versions of SQL Server. You can test with SQL Server Express and be assured that the application will behave the same in production, with a few exceptions for features that vary between SQL Server Editions.

LocalDB is a special execution mode of SQL Server Express that enables you to work with databases as *.mdf* files. Typically, LocalDB database files are kept in the *App_Data* folder of a web project. The user instance feature in SQL Server Express also enables you to work with *.mdf* files, but the user instance feature is deprecated; therefore, LocalDB is recommended for working with *.mdf* files.

Typically SQL Server Express is not used for production web applications. LocalDB in particular is not recommended for production use with a web application because it is not designed to work with IIS.

In Visual Studio 2012, LocalDB is installed by default with Visual Studio. In Visual Studio 2010 and earlier versions, SQL Server Express (without LocalDB) is installed by default with Visual Studio; that is why you installed it as one of the prerequisites in the first tutorial in this series.

For more information about SQL Server editions, including LocalDB, see the following resources Working with SQL Server Databases.

Entity Framework and Universal Providers

For database access, the Contoso University application requires the following software that must be deployed with the application because it is not included in the .NET Framework:

- ASP.NET Universal Providers (enables the ASP.NET membership system to use Azure SQL Database)
- Entity Framework

Because this software is included in NuGet packages, the project is already set up so that the required assemblies are deployed with the project. (The links point to the current versions of these packages, which might be newer than what is installed in the starter project that you downloaded for this tutorial.)

If you are deploying to a third-party hosting provider instead of Azure, make sure that you use Entity Framework 5.0 or later. Earlier versions of Code First Migrations require Full Trust, and most hosting providers will run your application in Medium Trust. For more information about Medium Trust, see the Deploy to IIS as a Test Environment tutorial.

Configure Code First Migrations for application database deployment

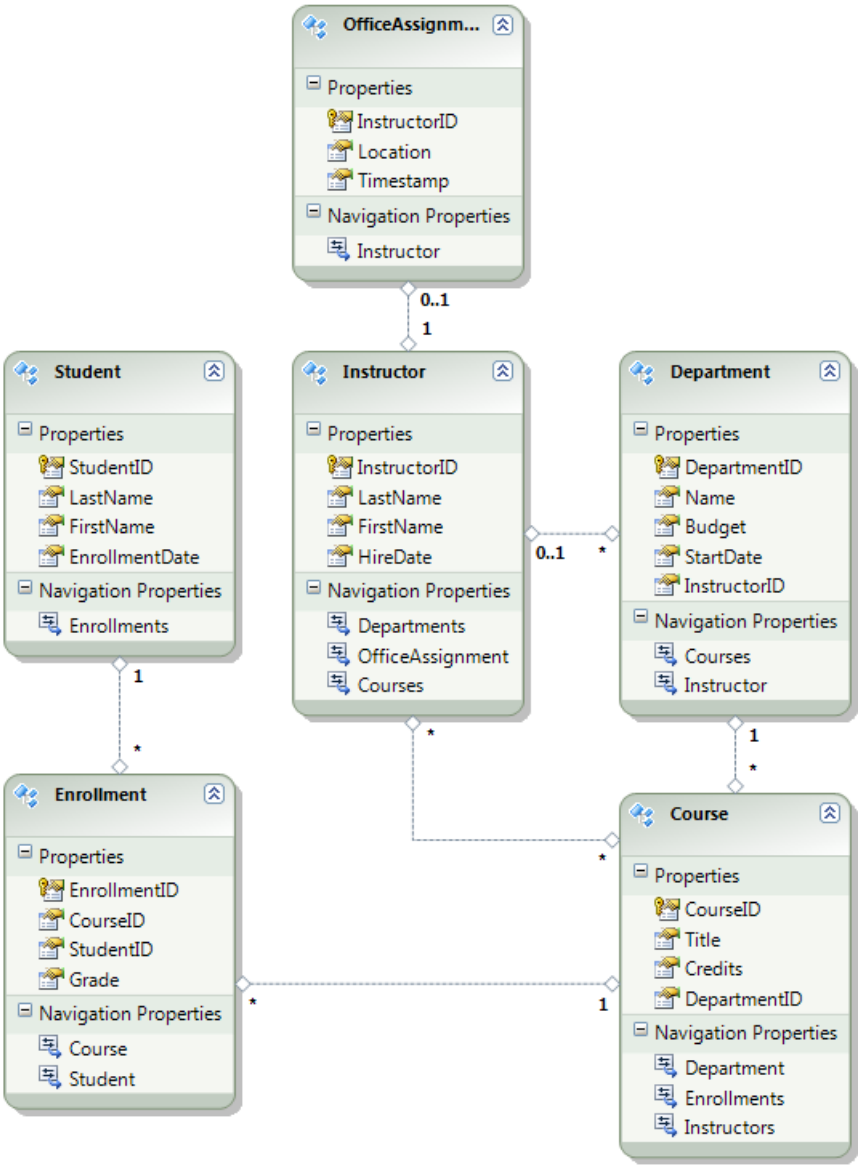
The Contoso University application database is managed by Code First, and you'll deploy it by using Code First Migrations. For an overview of database deployment by using Code First Migrations, see the first tutorial in this series.

When you deploy an application database, typically you don't simply deploy your development database with all of the data in it to production, because much of the data in it is probably there only for testing purposes. For example, the student names in a test database are fictional. On the other hand, you often can't deploy just the database structure with no data in it at all. Some of the data in your test database might be real data and must be there when users begin to use the application. For example, your database might have a table that contains valid grade values or real department names.

To simulate this common scenario, you'll configure a Code First Migrations `Seed` method that inserts into the database only the data that you want to be there in production. This `Seed` method shouldn't insert test data because it will run in production after Code First creates the database in production.

In earlier versions of Code First before Migrations was released, it was common for `Seed` methods to insert test data also, because with every model change during development the database had to be completely deleted and re-created from scratch. With Code First Migrations, test data is retained after database changes, so including test data in the `Seed` method is not necessary. The project that you downloaded uses the method of including all data in the `Seed` method of an initializer class. In this tutorial you'll disable that initializer class and enable Migrations. Then you'll update the `Seed` method in the Migrations configuration class so that it inserts only data that you want to be inserted in production.

The following diagram illustrates the schema of the application database:



For these tutorials, you'll assume that the `Student` and `Enrollment` tables should be empty when the site is first deployed. The other tables contain data that has to be preloaded when the application goes live.

Disable the initializer

Since you will be using Code First Migrations, you do not have to use the `DropCreateDatabaseIfModelChanges` Code First initializer. The code for this initializer is in the *SchoolInitializer.cs* file in the ContosoUniversity.DAL project. A setting in the `appSettings` element of the *Web.config* file causes this initializer to run whenever the application tries to access the database for the first time:

```
<appSettings>
  <add key="Environment" value="Dev" />
  <add key="DatabaseInitializerForType ContosoUniversity.DAL.SchoolContext, ContosoUniversity.DAL"
value="ContosoUniversity.DAL.SchoolInitializer, ContosoUniversity.DAL" />
</appSettings>
```

Open the application *Web.config* file and remove or comment out the `add` element that specifies the Code First initializer class. The `appSettings` element now looks like this:

```
<appSettings>
  <add key="Environment" value="Dev" />
</appSettings>
```

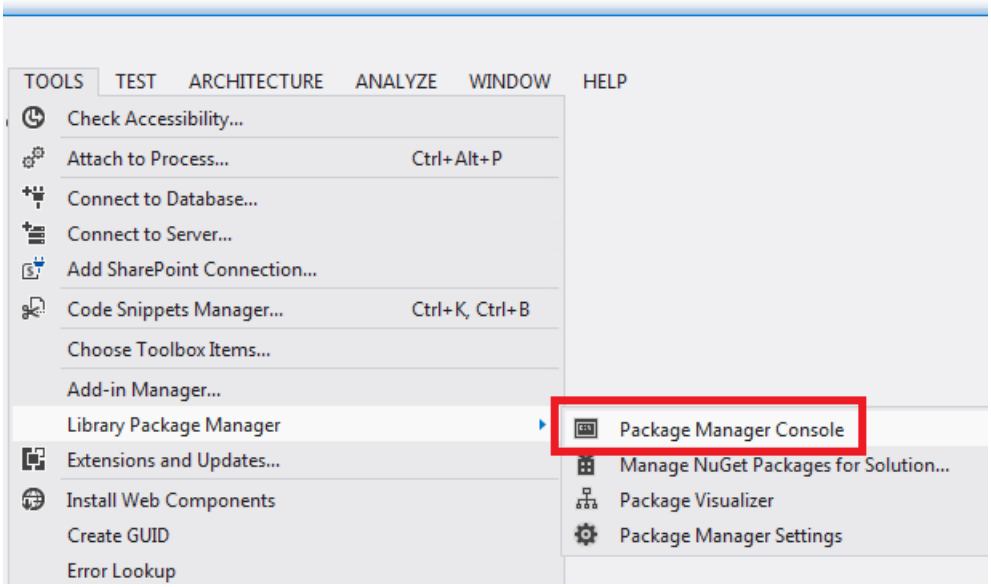
> **NOTE**
> Another way to specify an initializer class is do it by calling `Database.SetInitializer` in the `Application_Start` method in the *Global.asax* file. If you are enabling Migrations in a project that uses that method to specify the initializer, remove that line of code.

> **NOTE**
> If you are using Visual Studio 2013, add the following steps between steps 2 and 3: (a) In PMC enter "update-package entityframework -version 6.1.1" to get the current version of EF. Then (b) build the project to get a list of build errors, and fix them. Delete using statements for namespaces that no longer exist, right-click and click Resolve to add using statements where they're needed, and change occurrences of System.Data.EntityState to System.Data.Entity.EntityState.

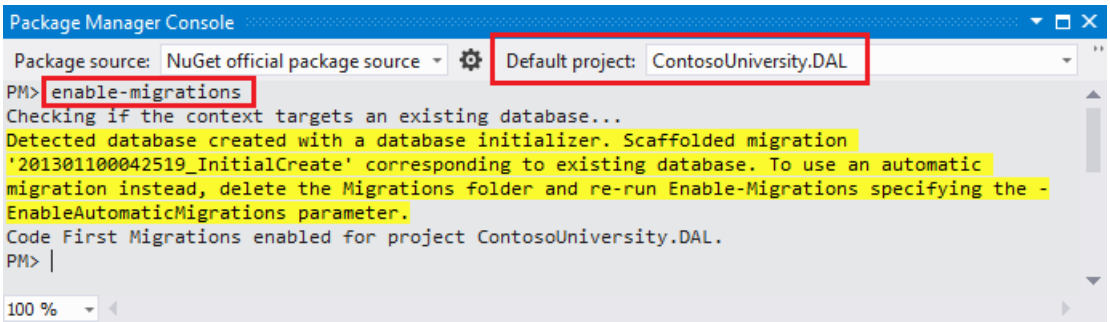
Enable Code First Migrations

1. Make sure that the ContosoUniversity project (not ContosoUniversity.DAL) is set as the startup project. In **Solution Explorer**, right-click the ContosoUniversity project and select **Set as Startup Project**. Code First Migrations will look in the startup project to find the database connection string.

2. From the **Tools** menu, choose **NuGet Package Manager** > **Package Manager Console**.

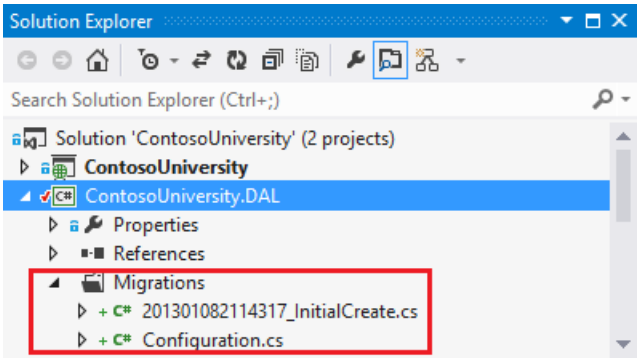


3. At the top of the **Package Manager Console** window select ContosoUniversity.DAL as the default project and then at the `PM>` prompt enter "enable-migrations".



(If you get an error saying the *enable-migrations* command is not recognized, enter the command *update-package EntityFramework -Reinstall* and try again.)

This command creates a *Migrations* folder in the ContosoUniversity.DAL project, and it puts in that folder two files: a *Configuration.cs* file that you can use to configure Migrations, and an *InitialCreate.cs* file for the first migration that creates the database.



You selected the DAL project in the **Default project** drop-down list of the **Package Manager Console** because the `enable-migrations` command must be executed in the project that contains the Code First context class. When that class is in a class library project, Code First Migrations looks for the database connection string in the startup project for the solution. In the ContosoUniversity solution, the web project has been set as the startup project. If you don't want to designate the project that has the connection string as the startup project in Visual Studio, you can specify the startup project in the PowerShell command. To see the command syntax, enter the command `get-help enable-migrations`.

The `enable-migrations` command automatically created the first migration because the database already exists. An alternative is to have Migrations create the database. To do that, use **Server Explorer** or **SQL Server Object Explorer** to delete the ContosoUniversity database before you enable Migrations. After you enable migrations, create the first migration manually by entering the command "add-migration InitialCreate". You can then create the database by entering the command "update-database".

Set up the Seed method

For this tutorial you'll add fixed data by adding code to the `Seed` method of the Code First Migrations `Configuration` class. Code First Migrations calls the `Seed` method after every migration.

Since the `Seed` method runs after every migration, there is data already in the tables after the first migration. To handle this situation you'll use the `AddOrUpdate` method to update rows that have already been inserted, or insert them if they don't exist yet. The `AddOrUpdate` method might not be the best choice for your scenario. For more information, see Take care with EF 4.3 AddOrUpdate Method on Julie Lerman's blog.

1. Open the *Configuration.cs* file and replace the comments in the `Seed` method with the following code:

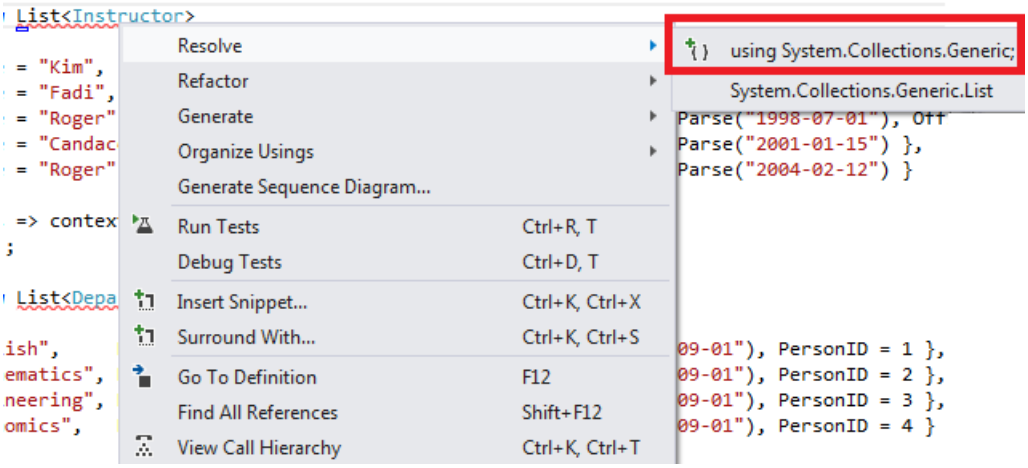
```
var instructors = new List<Instructor>
{
    new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie", HireDate =
DateTime.Parse("1995-03-11"), OfficeAssignment = new OfficeAssignment { Location = "Smith 17" } },
    new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",    HireDate =
DateTime.Parse("2002-07-06"), OfficeAssignment = new OfficeAssignment { Location = "Gowan 27" } },
    new Instructor { FirstMidName = "Roger",   LastName = "Harui",       HireDate =
DateTime.Parse("1998-07-01"), OfficeAssignment = new OfficeAssignment { Location = "Thompson 304" } },
    new Instructor { FirstMidName = "Candace", LastName = "Kapoor",      HireDate =
DateTime.Parse("2001-01-15") },
    new Instructor { FirstMidName = "Roger",   LastName = "Zheng",       HireDate =
DateTime.Parse("2004-02-12") }
};
instructors.ForEach(s => context.Instructors.AddOrUpdate(i => i.LastName, s));
context.SaveChanges();

var departments = new List<Department>
{
    new Department { Name = "English",     Budget = 350000, StartDate = DateTime.Parse("2007-09-01"),
PersonID = 1 },
    new Department { Name = "Mathematics", Budget = 100000, StartDate = DateTime.Parse("2007-09-01"),
PersonID = 2 },
    new Department { Name = "Engineering", Budget = 350000, StartDate = DateTime.Parse("2007-09-01"),
PersonID = 3 },
    new Department { Name = "Economics",   Budget = 100000, StartDate = DateTime.Parse("2007-09-01"),
PersonID = 4 }
};
departments.ForEach(s => context.Departments.AddOrUpdate(d => d.Name, s));
context.SaveChanges();

var courses = new List<Course>
{
    new Course { CourseID = 1050, Title = "Chemistry",      Credits = 3, DepartmentID = 3 },
    new Course { CourseID = 4022, Title = "Microeconomics", Credits = 3, DepartmentID = 4 },
    new Course { CourseID = 4041, Title = "Macroeconomics", Credits = 3, DepartmentID = 4 },
    new Course { CourseID = 1045, Title = "Calculus",       Credits = 4, DepartmentID = 2 },
    new Course { CourseID = 3141, Title = "Trigonometry",   Credits = 4, DepartmentID = 2 },
    new Course { CourseID = 2021, Title = "Composition",    Credits = 3, DepartmentID = 1 },
    new Course { CourseID = 2042, Title = "Literature",     Credits = 4, DepartmentID = 1 }
};
courses.ForEach(s => context.Courses.AddOrUpdate(s));
context.SaveChanges();

courses[0].Instructors.Add(instructors[0]);
courses[0].Instructors.Add(instructors[1]);
courses[1].Instructors.Add(instructors[2]);
courses[2].Instructors.Add(instructors[2]);
courses[3].Instructors.Add(instructors[3]);
courses[4].Instructors.Add(instructors[3]);
courses[5].Instructors.Add(instructors[3]);
courses[6].Instructors.Add(instructors[3]);
context.SaveChanges();
```

2. The references to `List` have red squiggly lines under them because you don't have a `using` statement for its namespace yet. Right-click one of the instances of `List` and click **Resolve**, and then click **using System.Collections.Generic**.



This menu selection adds the following code to the `using` statements near the top of the file.

```
using System.Collections.Generic;
```

3. Press CTRL-SHIFT-B to build the project.

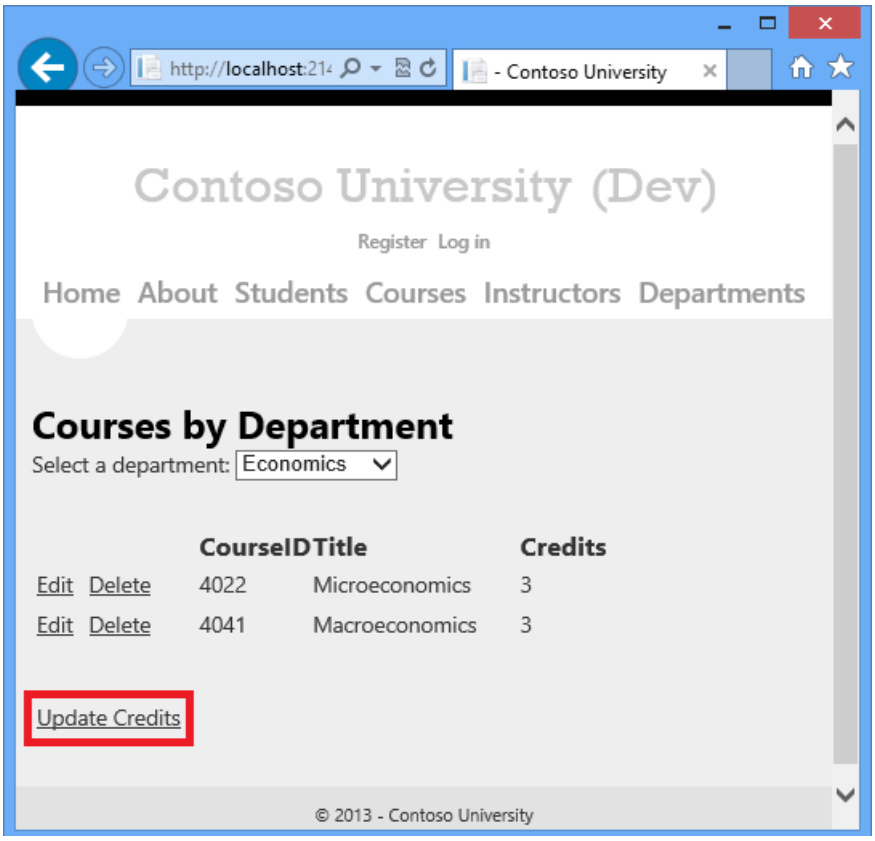
The project is now ready to deploy the *ContosoUniversity* database. After you deploy the application, the first time you run it and navigate to a page that accesses the database, Code First will create the database and run this `Seed` method.

> **NOTE**
>
> Adding code to the `Seed` method is one of many ways that you can insert fixed data into the database. An alternative is to add code to the `Up` and `Down` methods of each migration class. The `Up` and `Down` methods contain code that implements database changes. You'll see examples of them in the Deploying a Database Update tutorial.
>
> You can also write code that executes SQL statements by using the `Sql` method. For example, if you were adding a Budget column to the Department table and wanted to initialize all department budgets to $1,000.00 as part of a migration, you could add the following line of code to the `Up` method for that migration:
>
> `Sql("UPDATE Department SET Budget = 1000");`

Create scripts for membership database deployment

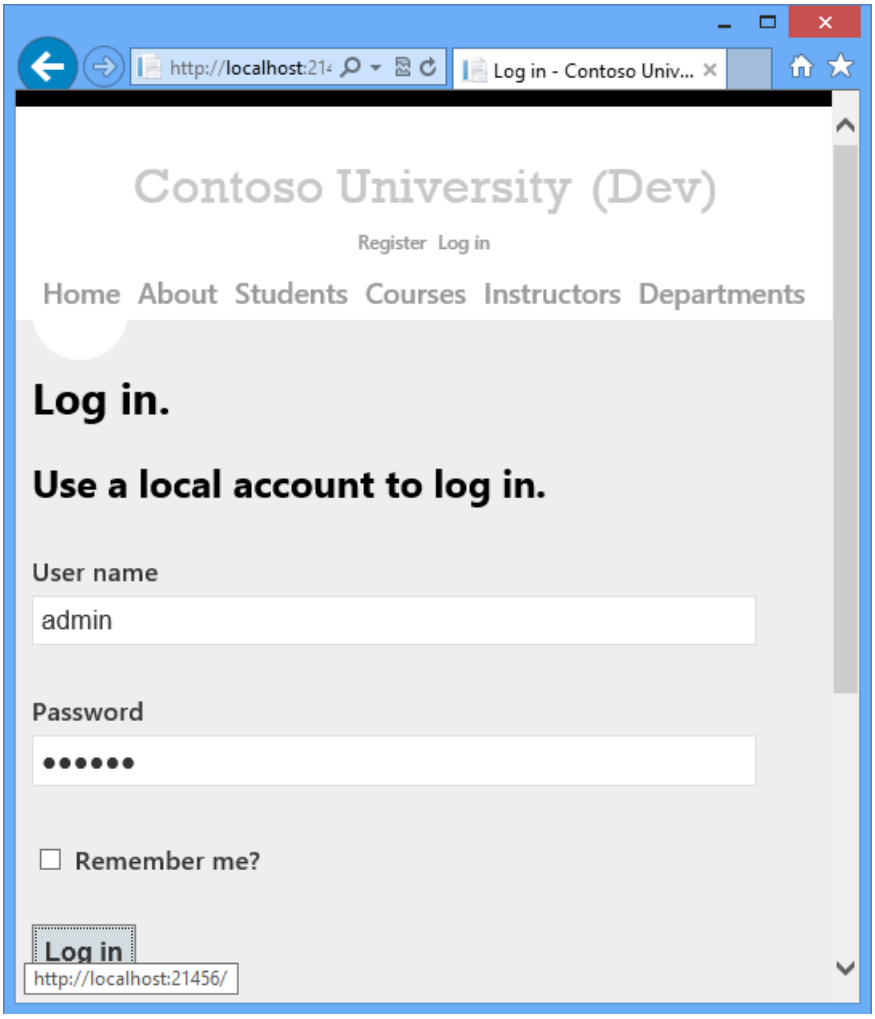
The Contoso University application uses the ASP.NET membership system and forms authentication to authenticate and authorize users. The **Update Credits** page is accessible only to users who are in the Administrator role.

Run the application and click **Courses**, and then click **Update Credits**.

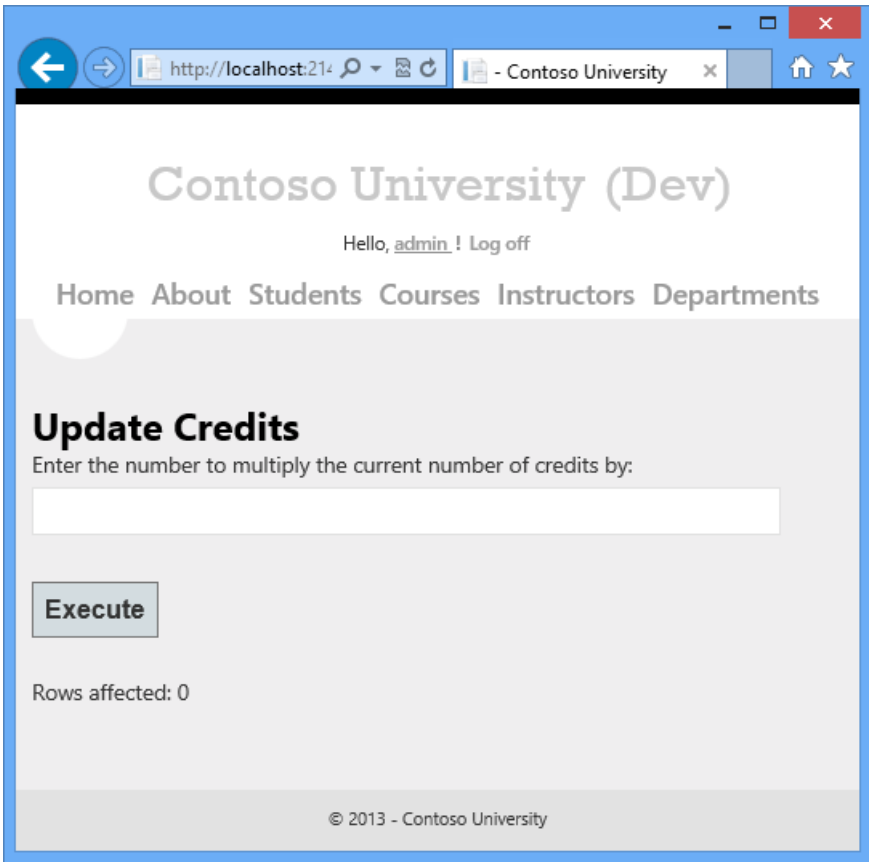


The **Log in** page appears because the **Update Credits** page requires administrative privileges.

Enter *admin* as the user name and *devpwd* as the password and click **Log in**.



The **Update Credits** page appears.



User and role information is in the *aspnet-ContosoUniversity* database that is specified by the **DefaultConnection** connection string in the *Web.config* file.

This database is not managed by Entity Framework Code First, so you can't use Migrations to deploy it. You'll use the dbDacFx provider to deploy the database schema, and you'll configure the publish profile to run a script that will insert initial data into database tables.

> **NOTE**
>
> A new ASP.NET membership system (now named ASP.NET Identity) was introduced with Visual Studio 2013. The new system enables you to keep both application and membership tables in the same database, and you can use Code First Migrations to deploy both. The sample application uses the earlier ASP.NET membership system, which can't be deployed by using Code First Migrations. The procedures for deploying this membership database apply also to any other scenario in which your application needs to deploy a SQL Server database that isn't created by Entity Framework Code First.

Here too, you typically don't want the same data in production that you have in development. When you deploy a site for the first time, it is common to exclude most or all of the user accounts you create for testing. Therefore, the downloaded project has two membership databases: *aspnet-ContosoUniversity.mdf* with development users and *aspnet-ContosoUniversity-Prod.mdf* with production users. For this tutorial the user names are the same in both databases: *admin* and *nonadmin*. Both users have the password *devpwd* in the development database and *prodpwd* in the production database.

You'll deploy the development users to the test environment and the production users to staging and production. To do that you'll create two SQL scripts in this tutorial, one for development and one for production, and in later tutorials you'll configure the publish process to run them.

> **NOTE**
> The membership database stores a hash of account passwords. In order to deploy accounts from one machine to another, you must make sure that hashing routines don't generate different hashes on the destination server than they do on the source computer. They will generate the same hashes when you use the ASP.NET Universal Providers, as long as you don't change the default algorithm. The default algorithm is HMACSHA256 and is specified in the **validation** attribute of the **machineKey** element in the Web.config file.

You can create data deployment scripts manually, by using SQL Server Management Studio (SSMS), or by using a third-party tool. This remainder of this tutorial will show you how to do it in SSMS, but if you don't want to install and use SSMS you can get the scripts from the completed version of the project and skip to the section where you store them in the solution folder.

To install SSMS, install it from Download Center: Microsoft SQL Server 2012 Express by clicking ENU\x64\SQLManagementStudio_x64_ENU.exe or ENU\x86\SQLManagementStudio_x86_ENU.exe. If you choose the wrong one for your system it will fail to install and you can try the other one.

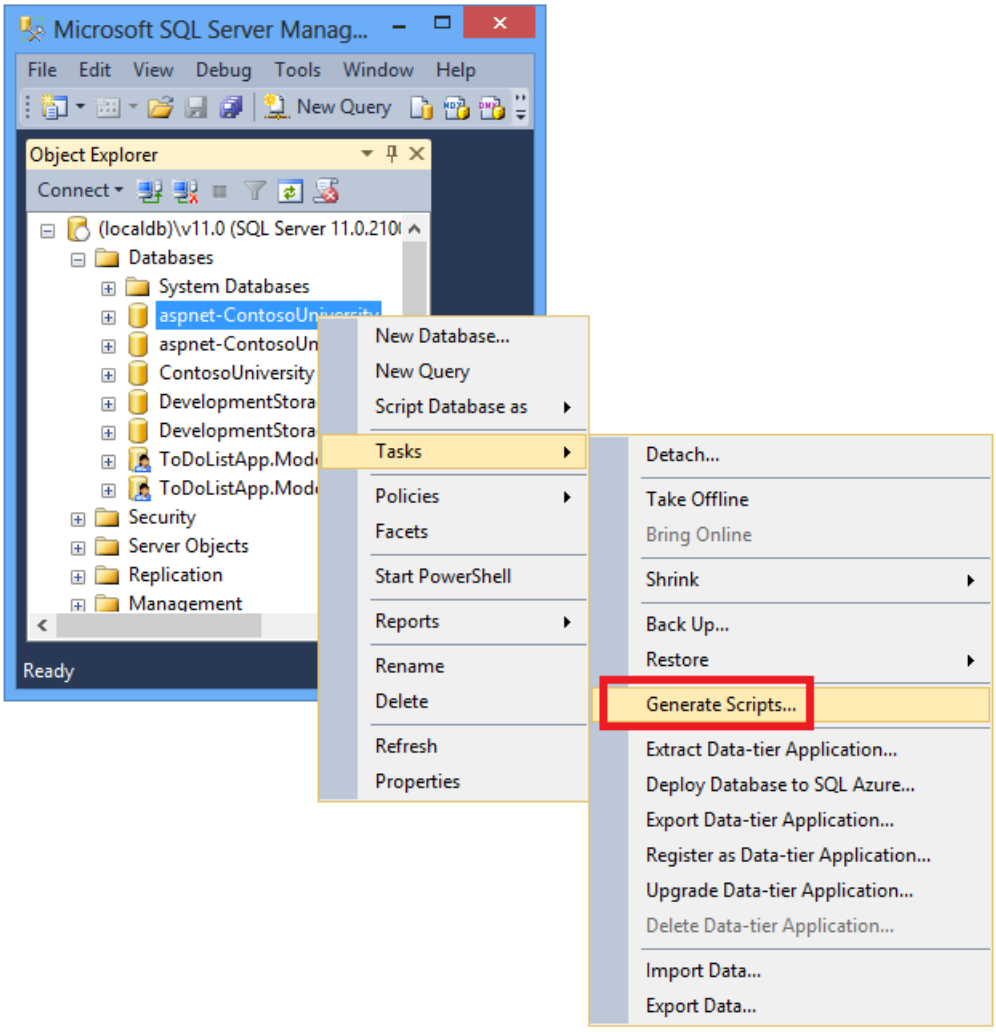
(Note that this is a 600 megabyte download. It may take a long time to install and will require a reboot of your computer.)

On the first page of the SQL Server Installation Center, click **New SQL Server stand-alone installation or add features to an existing installation**, and follow the instructions, accepting the default choices.

Create the development database script

1. Run SSMS.
2. In the **Connect to Server** dialog box, enter *(localdb)\v11.0* as the **Server name**, leave **Authentication** set to **Windows Authentication**, and then click **Connect**.

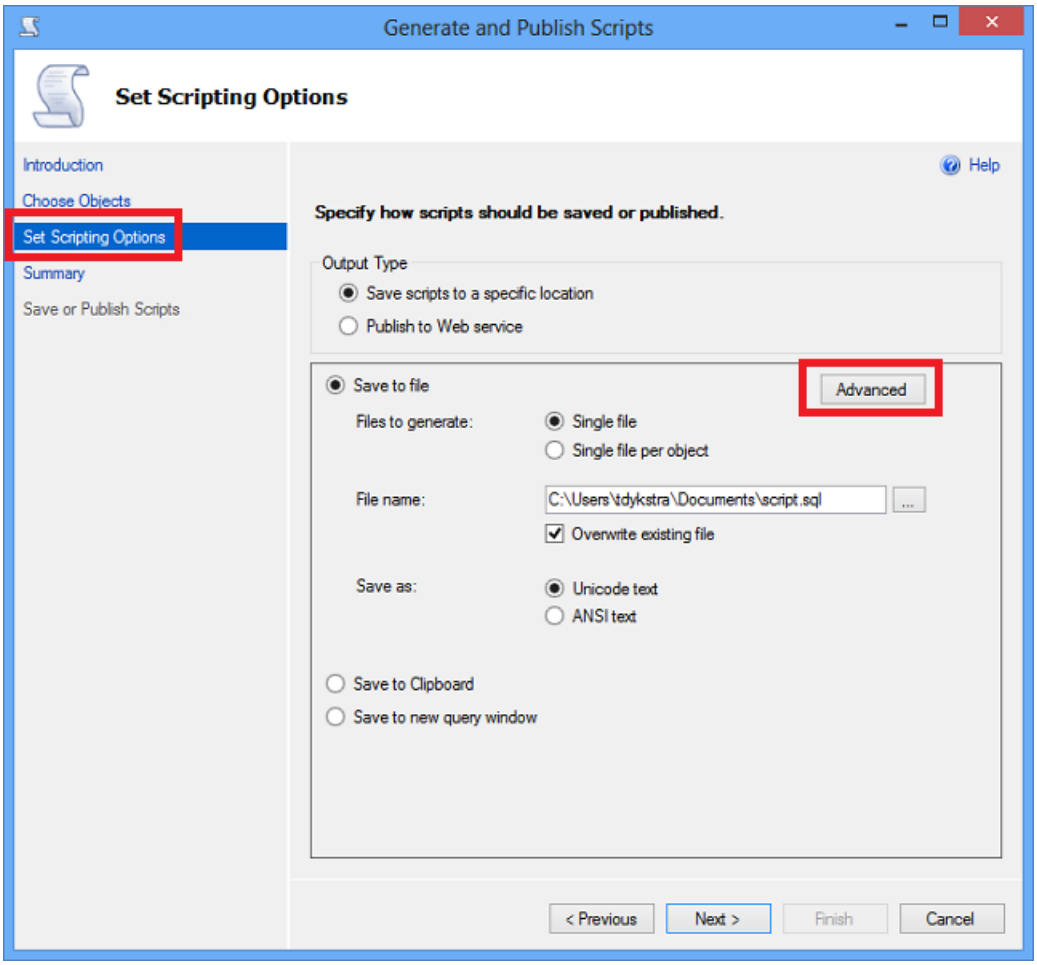
3. In the **Object Explorer** window, expand **Databases**, right-click **aspnet-ContosoUniversity**, click **Tasks**, and then click **Generate Scripts**.



4. In the **Generate and Publish Scripts** dialog box, click **Set Scripting Options**.

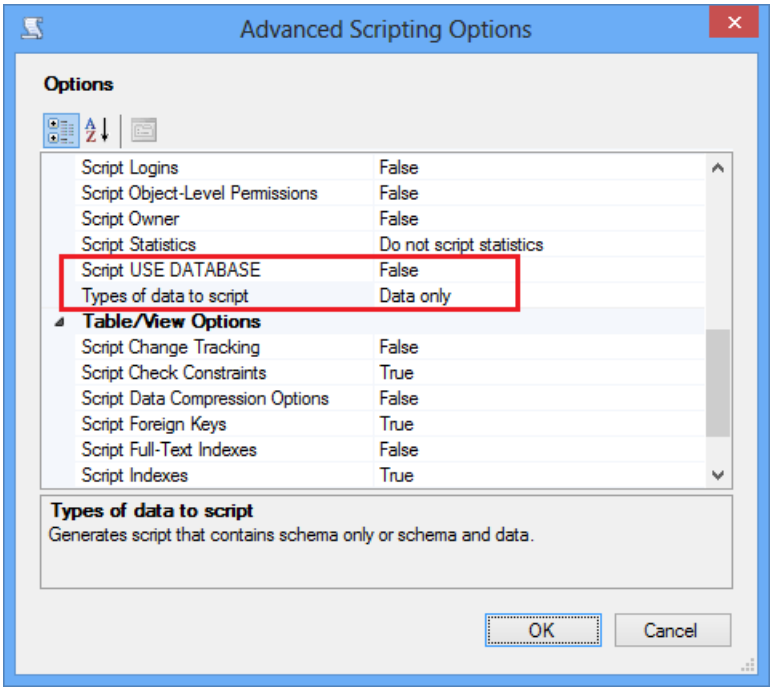
   You can skip the **Choose Objects** step because the default is **Script entire database and all database objects** and that is what you want.

5. Click **Advanced**.



6. In the **Advanced Scripting Options** dialog box, scroll down to **Types of data to script**, and click the **Data only** option in the drop-down list.

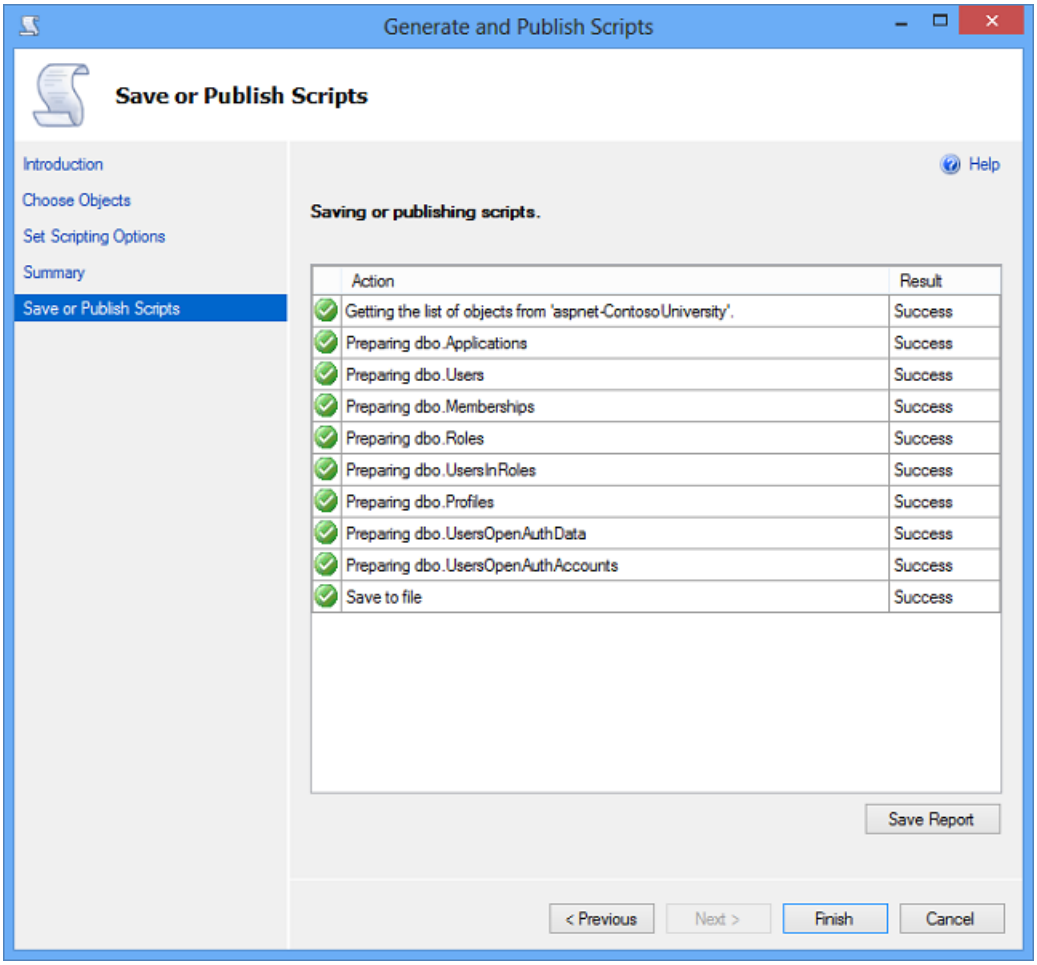
7. Change **Script USE DATABASE** to **False**. USE statements aren't valid for Azure SQL Database and aren't needed for deployment to SQL Server Express in the test environment.



8. Click **OK**.

9. In the **Generate and Publish Scripts** dialog box, the **File name** box specifies where the script will be created. Change the path to your solution folder (the folder that has your ContosoUniversity.sln file) and the file name to *aspnet-data-dev.sql*.

10. Click **Next** to go to the **Summary** tab, and then click **Next** again to create the script.

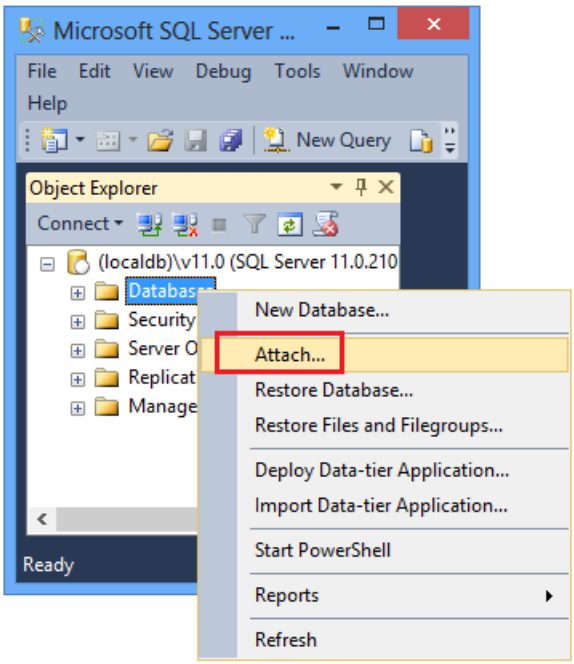


11. Click **Finish**.

Create the production database script

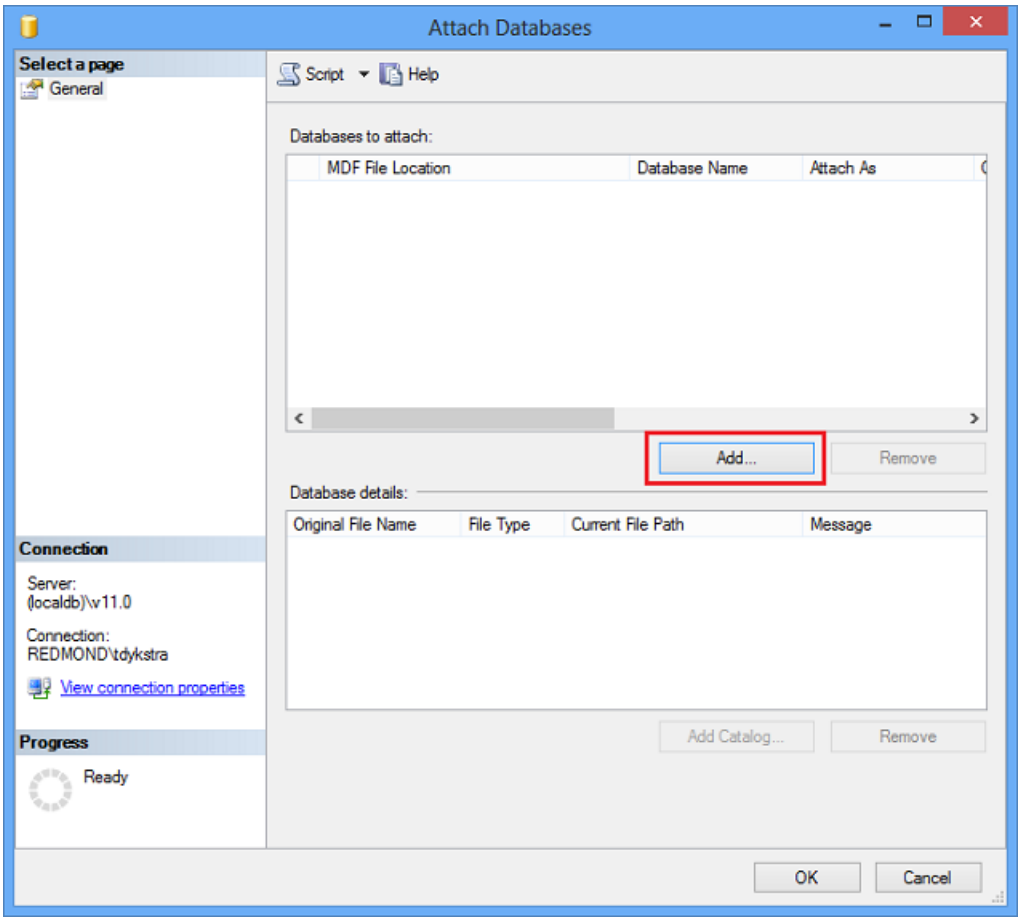
Since you haven't run the project with the production database, it isn't attached yet to the LocalDB instance. Therefore you need to attach the database first.

1. In the SSMS **Object Explorer**, right-click **Databases** and click **Attach**.



2. In the **Attach Databases** dialog box, click **Add** and then navigate to the *aspnet-ContosoUniversity-*

Prod.mdf file in the *App_Data* folder.

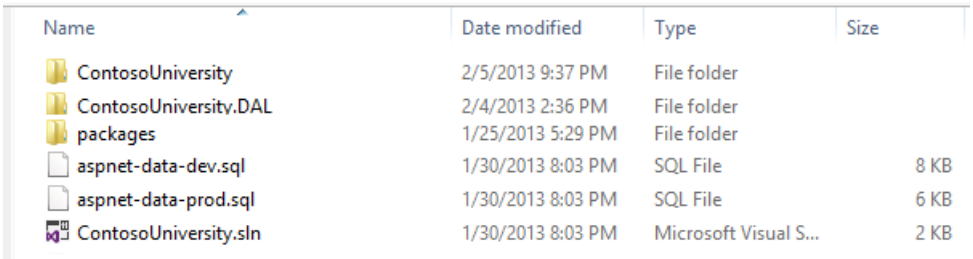


3. Click **OK**.

4. Follow the same procedure you used earlier to create a script for the production file. Name the script file *aspnet-data-prod.sql*.

Summary

Both databases are now ready to be deployed and you have two data deployment scripts in your solution folder.



Name	Date modified	Type	Size
ContosoUniversity	2/5/2013 9:37 PM	File folder	
ContosoUniversity.DAL	2/4/2013 2:36 PM	File folder	
packages	1/25/2013 5:29 PM	File folder	
aspnet-data-dev.sql	1/30/2013 8:03 PM	SQL File	8 KB
aspnet-data-prod.sql	1/30/2013 8:03 PM	SQL File	6 KB
ContosoUniversity.sln	1/30/2013 8:03 PM	Microsoft Visual S...	2 KB

In the following tutorial you configure project settings that affect deployment, and you set up automatic *Web.config* file transformations for settings that must be different in the deployed application.

More Information

For more information on NuGet, see Manage Project Libraries with NuGet and NuGet Documentation. If you don't want to use NuGet, you'll need to learn how to analyze a NuGet package to determine what it does when it is installed. (For example, it might configure *Web.config* transformations, configure PowerShell scripts to run at build time, etc.) To learn more about how NuGet works, see Creating and Publishing a Package and Configuration File and Source Code Transformations.

ASP.NET Web Deployment using Visual Studio: Web.config File Transformations

2/20/2020 • 9 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This tutorial series shows you how to deploy (publish) an ASP.NET web application to Azure App Service Web Apps or to a third-party hosting provider, by using Visual Studio 2012 or Visual Studio 2010. For information about the series, see the first tutorial in the series.

Overview

This tutorial shows you how to automate the process of changing the *Web.config* file when you deploy it to different destination environments. Most applications have settings in the *Web.config* file that must be different when the application is deployed. Automating the process of making these changes keeps you from having to do them manually every time you deploy, which would be tedious and error prone.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

Web.config transformations versus Web Deploy parameters

There are two ways to automate the process of changing *Web.config* file settings: Web.config transformations and Web Deploy parameters. A *Web.config* transformation file contains XML markup that specifies how to change the *Web.config* file when it is deployed. You can specify different changes for specific build configurations and for specific publish profiles. The default build configurations are Debug and Release, and you can create custom build configurations. A publish profile typically corresponds to a destination environment. (You'll learn more about publish profiles in the Deploying to IIS as a Test Environment tutorial.)

Web Deploy parameters can be used to specify many different kinds of settings that must be configured during deployment, including settings that are found in *Web.config* files. When used to specify *Web.config* file changes, Web Deploy parameters are more complex to set up, but they are useful when you do not know the value to be set until you deploy. For example, in an enterprise environment, you might create a *deployment package* and give it to a person in the IT department to install in production, and that person has to be able to enter connection strings or passwords that you do not know.

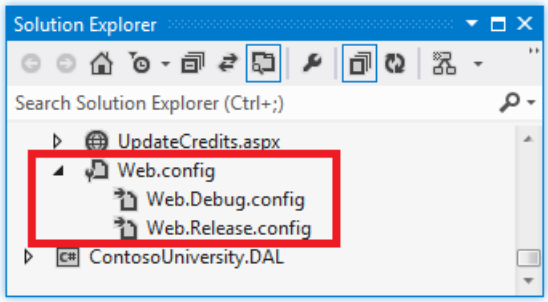
For the scenario that this tutorial series covers, you know in advance everything that has to be done to the *Web.config* file, so you do not need to use Web Deploy parameters. You'll configure some transformations that differ depending on the build configuration used, and some that differ depending on the publish profile used.

Specifying Web.config settings in Azure

If the *Web.config* file settings that you want to change are in the `<connectionStrings>` or the `<appSettings>` element, and if you are deploying to Web Apps in Azure App Service, you have another option for automating changes during deployment. You can enter the settings that you want to take effect in Azure in the **Configure** tab of the management portal page for your web app (scroll down to the **app settings** and **connection strings** sections). When you deploy the project, Azure automatically applies the changes. For more information, see Windows Azure Web Sites: How Application Strings and Connection Strings Work.

Default transformation files

In **Solution Explorer**, expand *Web.config* to see the *Web.Debug.config* and *Web.Release.config* transformation files that are created by default for the two default build configurations.



You can create transformation files for custom build configurations by right-clicking the Web.config file and choosing **Add Config Transforms** from the context menu. For this tutorial you don't need to do that, and the menu option is disabled, because you haven't created any custom build configurations.

Later you'll create three more transformation files, one each for the test, staging, and production publish profiles. A typical example of a setting that you would handle in a publish profile transformation file because it depends on the destination environment is a WCF endpoint that is different for test versus production. You'll create publish profile transformation files in later tutorials after you create the publish profiles that they go with.

Disable debug mode

An example of a setting that depends on build configuration rather than destination environment is the `debug` attribute. For a Release build, you typically want debugging disabled regardless of which environment you are deploying to. Therefore, by default the Visual Studio project templates create *Web.Release.config* transform files with code that removes the `debug` attribute from the `compilation` element. Here is the default *Web.Release.config*: in addition to some sample transformation code that is commented out, it includes code in the `compilation` element that removes the `debug` attribute:

```
<?xml version="1.0" encoding="utf-8"?>

<!-- For more information on using web.config transformation visit https://go.microsoft.com/fwlink/?
LinkId=125889 -->

<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <!--
    In the example below, the "SetAttributes" transform will change the value of
    "connectionString" to use "ReleaseSQLServer" only when the "Match" locator
    finds an attribute "name" that has a value of "MyDB".

    <connectionStrings>
      <add name="MyDB"
        connectionString="Data Source=ReleaseSQLServer;Initial Catalog=MyReleaseDB;Integrated Security=True"
        xdt:Transform="SetAttributes" xdt:Locator="Match(name)"/>
    </connectionStrings>
  -->
  <system.web>
    <compilation xdt:Transform="RemoveAttributes(debug)" />
    <!--
      In the example below, the "Replace" transform will replace the entire
      <customErrors> section of your web.config file.
      Note that because there is only one customErrors section under the
      <system.web> node, there is no need to use the "xdt:Locator" attribute.

      <customErrors defaultRedirect="GenericError.htm"
        mode="RemoteOnly" xdt:Transform="Replace">
        <error statusCode="500" redirect="InternalError.htm"/>
      </customErrors>
    -->
  </system.web>
</configuration>
```

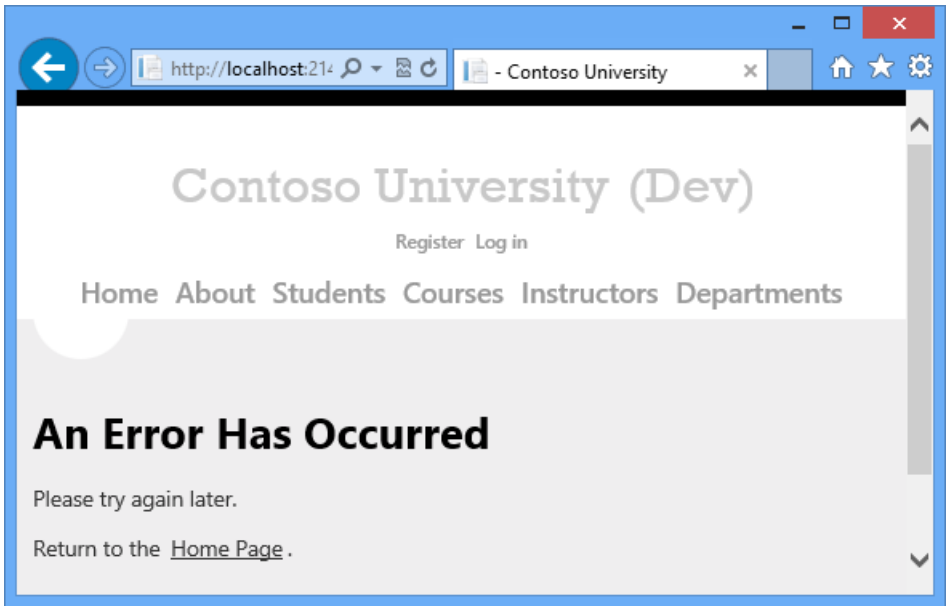
The `xdt:Transform="RemoveAttributes(debug)"` attribute specifies that you want the `debug` attribute to be removed from the `system.web/compilation` element in the deployed *Web.config* file. This will be done every time you deploy a Release build.

Limit error log access to administrators

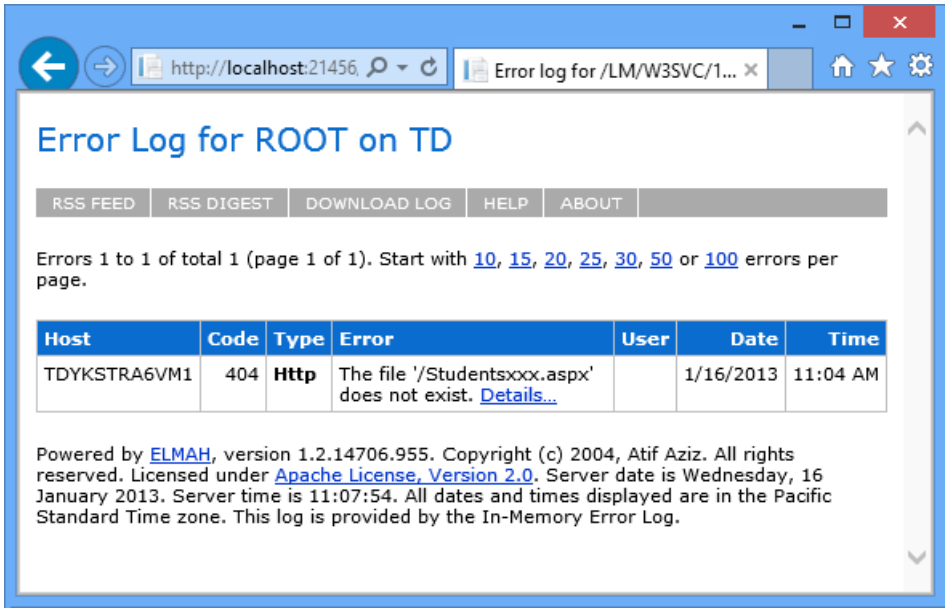
If there's an error while the application runs, the application displays a generic error page in place of the system-generated error page, and it uses the Elmah NuGet package for error logging and reporting. The `customErrors` element in the application *Web.config* file specifies the error page:

```
<customErrors mode="RemoteOnly" defaultRedirect="~/GenericErrorPage.aspx">
  <error statusCode="404" redirect="~/GenericErrorPage.aspx" />
</customErrors>
```

To see the error page, temporarily change the `mode` attribute of the `customErrors` element from "RemoteOnly" to "On" and run the application from Visual Studio. Cause an error by requesting an invalid URL, such as *Studentsxxx.aspx*. Instead of an IIS-generated "The resource cannot be found" error page, you see the *GenericErrorPage.aspx* page.



To see the error log, replace everything in the URL after the port number with *elmah.axd* (for example, `http://localhost:51130/elmah.axd`) and press Enter:



Don't forget to set the `customErrors` element back to "RemoteOnly" mode when you're done.

On your development computer it's convenient to allow free access to the error log page, but in production that would be a security risk. For the production site, you want to add an authorization rule that restricts error log access to administrators, and to make sure that the restriction works you want it in test and staging also. Therefore this is another change that you want to implement every time you deploy a Release build, and so it belongs in the *Web.Release.config* file.

Open *Web.Release.config* and add a new `location` element immediately before the closing `configuration` tag, as shown here.

```
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <!--
    In the example below, the "SetAttributes" transform will change the value of
    "connectionString" to use "ReleaseSQLServer" only when the "Match" locator
    finds an attribute "name" that has a value of "MyDB".

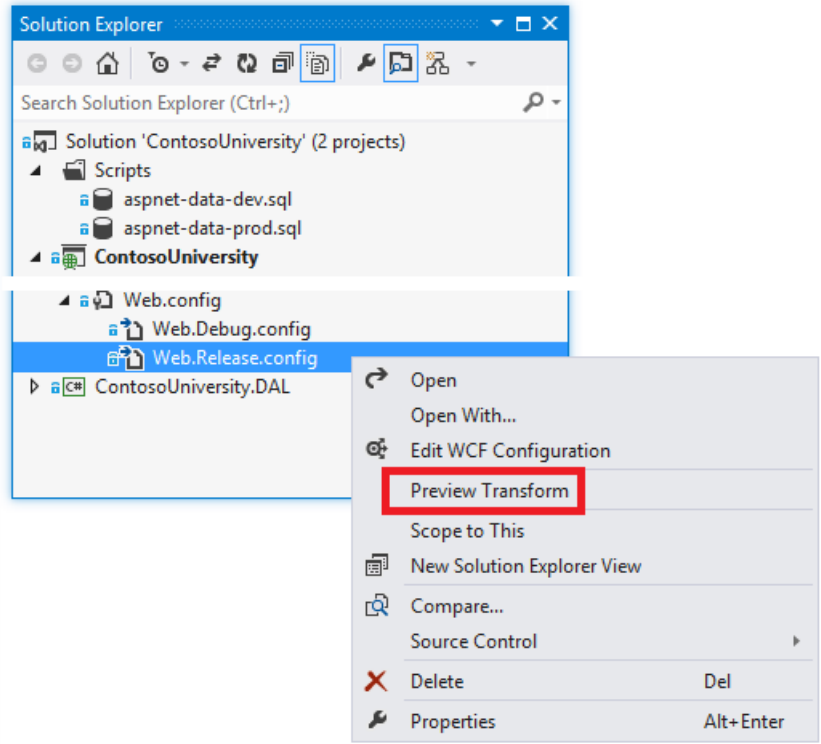
    <connectionStrings>
      <add name="MyDB"
        connectionString="Data Source=ReleaseSQLServer;Initial Catalog=MyReleaseDB;Integrated Security=True"
        xdt:Transform="SetAttributes" xdt:Locator="Match(name)"/>
    </connectionStrings>
  -->
  <system.web>
    <compilation xdt:Transform="RemoveAttributes(debug)" />
    <!--
      In the example below, the "Replace" transform will replace the entire
      <customErrors> section of your web.config file.
      Note that because there is only one customErrors section under the
      <system.web> node, there is no need to use the "xdt:Locator" attribute.

      <customErrors defaultRedirect="GenericError.htm"
        mode="RemoteOnly" xdt:Transform="Replace">
        <error statusCode="500" redirect="InternalError.htm"/>
      </customErrors>
    -->
  </system.web>
  <location path="elmah.axd" xdt:Transform="Insert">
    <system.web>
      <authorization>
        <allow roles="Administrator" />
        <deny users="*" />
      </authorization>
    </system.web>
  </location>
</configuration>
```

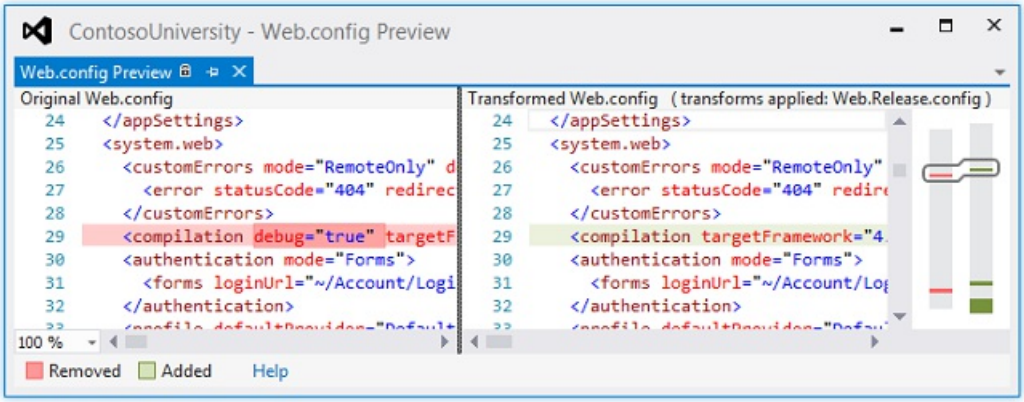
The `Transform` attribute value of "Insert" causes this `location` element to be added as a sibling to any existing `location` elements in the *Web.config* file. (There is already one `location` element that specifies authorization rules for the **Update Credits** page.)

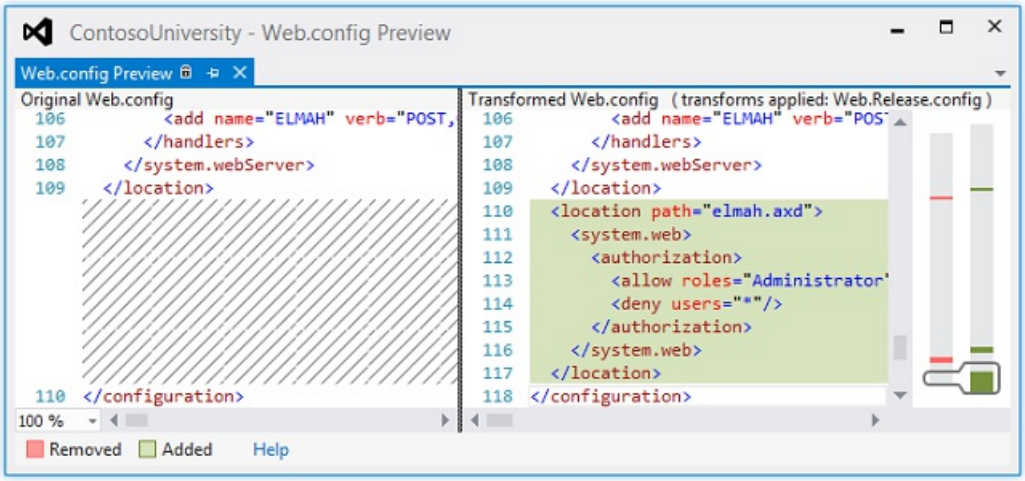
Now you can preview the transform to make sure that you coded it correctly.

In **Solution Explorer**, right-click *Web.Release.config* and click **Preview Transform**.



A page opens that shows you the development *Web.config* file on the left and what the deployed *Web.config* file will look like on the right, with changes highlighted.





(In the preview, you might notice some additional changes that you didn't write transforms for: these typically involve the removal of white space that doesn't affect functionality.)

When you test the site after deployment, you'll also test to verify that the authorization rule is effective.

> **NOTE**
>
> **Security Note** Never display error details to the public in a production application, or store that information in a public location. Attackers can use error information to discover vulnerabilities in a site. If you use ELMAH in your own application, configure ELMAH to minimize security risks. The ELMAH example in this tutorial should not be considered a recommended configuration. It is an example that was chosen in order to illustrate how to handle a folder that the application must be able to create files in. For more information, see securing the ELMAH endpoint.

A setting that you'll handle in publish profile transformation files

A common scenario is to have *Web.config* file settings that must be different in each environment that you deploy to. For example, an application that calls a WCF service might need a different endpoint in test and production environments. The Contoso University application includes a setting of this kind also. This setting controls a visible indicator on a site's pages that tells you which environment you are in, such as development, test, or production. The setting value determines whether the application will append "(Dev)" or "(Test)" to the main heading in the *Site.Master* master page:

The environment indicator is omitted when the application is running in staging or production.

The Contoso University web pages read a value that is set in `appSettings` in the *Web.config* file in order to determine what environment the application is running in:

```
<appSettings>
    <add key="Environment" value="Dev" />
</appSettings>
```

The value should be "Test" in the test environment, and "Prod" for staging and production.

The following code in a transform file will implement this transformation:

```
<appSettings>
    <add key="Environment" value="Test" xdt:Transform="SetAttributes" xdt:Locator="Match(key)"/>
</appSettings>
```

The `xdt:Transform` attribute value "SetAttributes" indicates that the purpose of this transform is to change attribute values of an existing element in the *Web.config* file. The `xdt:Locator` attribute value "Match(key)" indicates that the element to be modified is the one whose `key` attribute matches the `key` attribute specified here. The only other attribute of the `add` element is `value`, and that is what will be changed in the deployed *Web.config* file. The code shown here causes the `value` attribute of the `Environment` `appSettings` element to be set to "Test" in the *Web.config* file that is deployed.

This transform belongs in the publish profile transform files, which you haven't created yet. You'll create and update the transform files that implement this change when you create the publish profiles for the test, staging, and production environments. You'll do that in the deploy to IIS and deploy to production tutorials.

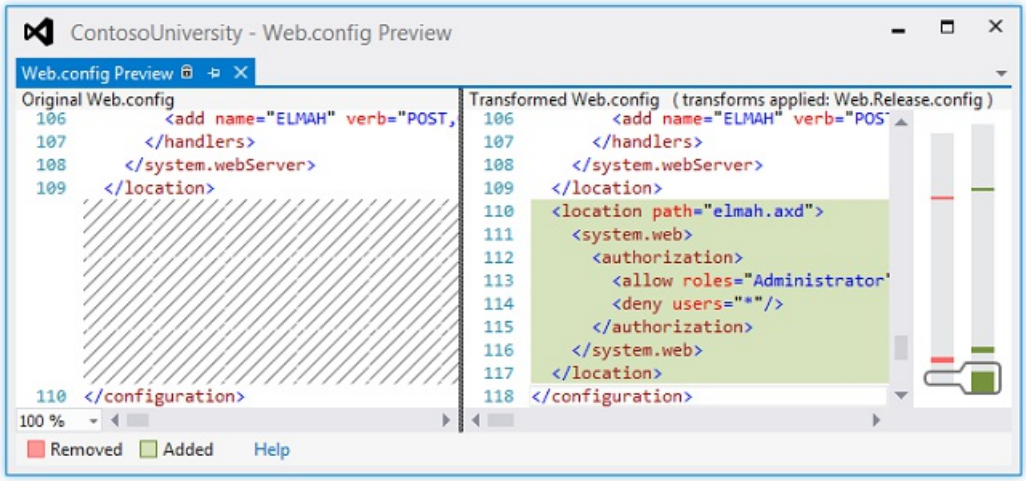
> **NOTE**
>
> Because this setting is in the `<appSettings>` element, you have another alternative for specifying the transformation when you're deploying to Web Apps in Azure App Service See Specifying Web.config settings in Azure earlier in this topic.

Setting connection strings

Although the default transform file contains an example that shows how to update a connection string, in most cases you do not need to set up connection string transformations, because you can specify connection strings in the publish profile. You'll do that in the deploy to IIS and deploy to production tutorials.

Summary

You have now done as much as you can with *Web.config* transformations before you create the publish profiles, and you've seen a preview of what will be in the deployed Web.config file.



In the following tutorial, you'll take care of deployment set-up tasks that require setting project properties.

More Information

For more information about topics covered by this tutorial, see Using Web.config transformations to change settings in the destination Web.config file or app.config file during deployment in the Web Deployment Content Map for Visual Studio and ASP.NET.

PREVIOUS NEXT

ASP.NET Web Deployment using Visual Studio: Project Properties

2/20/2020 • 4 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This tutorial series shows you how to deploy (publish) an ASP.NET web application to Azure App Service Web Apps or to a third-party hosting provider, by using Visual Studio 2012 or Visual Studio 2010. For information about the series, see the first tutorial in the series.

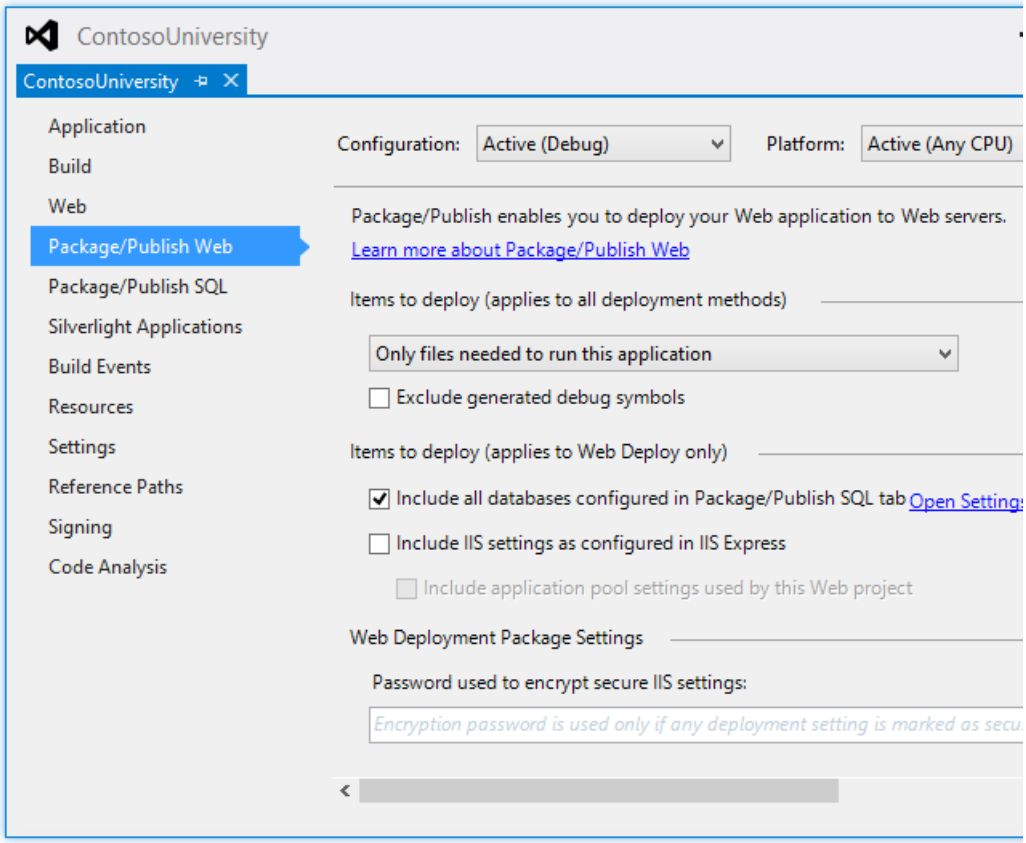
Overview

Some deployment options are configured in project properties that are stored in the project file (the *.csproj* or *.vbproj* file). In most cases, the default values of these settings are what you want, but you can use the **Project Properties** UI built into Visual Studio to work with these settings if you have to change them. In this tutorial you review the deployment settings in **Project Properties**. You also create a placeholder file that causes an empty folder to be deployed.

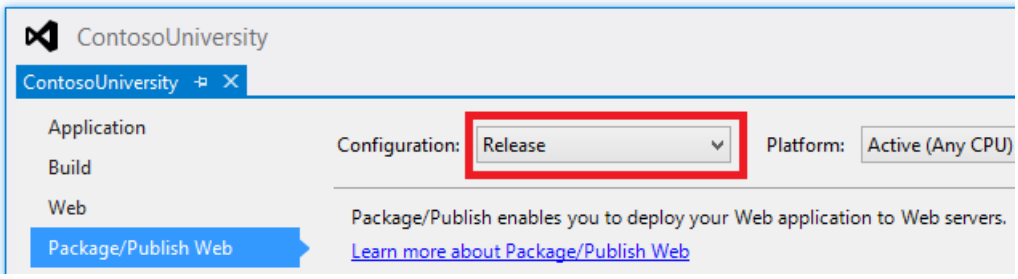
Configure deployment settings in the project properties window

Most settings that affect what happens during deployment are included in the publish profile, as you'll see in the following tutorials. A few settings that you should be aware of are located in the **Package/Publish** tabs of the **Project Properties** window. These settings are specified for each build configuration — that is, you can have different settings for a Release build than you have for a Debug build.

In **Solution Explorer**, right-click the **ContosoUniversity** project, select **Properties**, and then select the **Package/Publish Web** tab.



When the window is displayed, it defaults to showing settings for whichever build configuration is currently active for the solution. If the **Configuration** box does not indicate **Active (Release)**, select **Release** in order to display settings for the Release build configuration. You'll deploy Release builds to both your test and production environments.



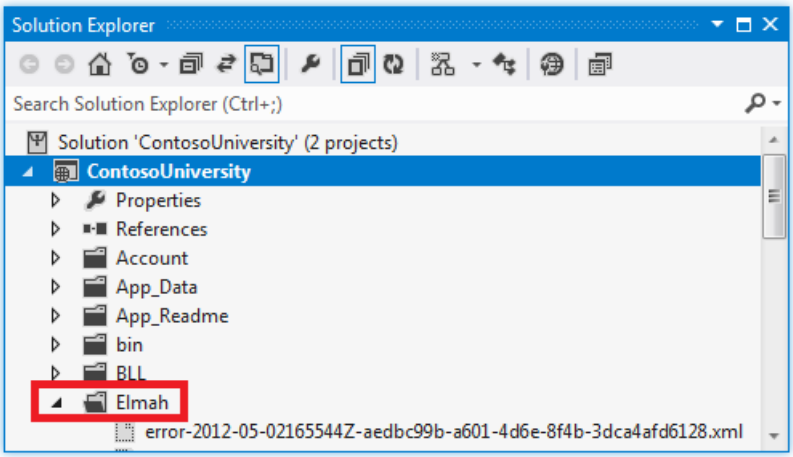
With **Active (Release)** or **Release** selected, you see the values that are effective when you deploy using the Release build configuration:

- In the **Items to deploy** box, **Only files needed to run the application** is selected. Other options are **All files in this project** or **All files in this project folder**. By leaving the default selection unchanged you avoid deploying source code files, for example. This setting is the reason why the folders that contain the SQL Server Compact binary files had to be included in the project. For more information about this setting, see **Why don't all of the files in my project folder get deployed?** in ASP.NET Web Application Project Deployment FAQ.
- **Exclude generated debug symbols** is selected. You won't be debugging when you use this build configuration.
- **Include all databases configured in Package/Publish SQL tab** is selected. Specifies whether Visual Studio will deploy databases as well as files. Although the check box label only mentions the **Package/Publish SQL** tab, clearing this check box would also disable database deployment that is configured in the publish profile. You will be doing that later, so the check box must remain selected. The **Package/Publish SQL** tab is used for a legacy database publishing method that you won't be using in these tutorials.
- The **Web Deployment Package Settings** section does not apply because you're using one-click publish in these tutorials.

Change the **Configuration** drop-down box to Debug to see the default settings for Debug builds. The values are the same, except **Exclude generated debug symbols** is cleared so that you can debug when you deploy a Debug build.

Make sure that the Elmah folder gets deployed

As you saw in the previous tutorial, the Elmah NuGet package provides functionality for error logging and reporting. In the Contoso University application Elmah has been configured to store error details in a folder named *Elmah*:



Excluding specific files or folders from deployment is a common requirement; another example would be a folder that users can upload files to. You don't want log files or uploaded files that were created in your development environment to be deployed to production. And if you are deploying an update to production you don't want the deployment process to delete files that exist in production. (Depending on how you set a deployment option, if a file exists in the destination site but not the source site when you deploy, Web Deploy deletes it from the destination.)

As you saw earlier in this tutorial, the **Items to deploy** option in the **Package/Publish Web** tab is set to **Only Files Needed to run this application**. As a result, log files that are created by Elmah in development will not be deployed, which is what you want to happen. (To be deployed, they would have to be included in the project and their **Build Action** property would have to be set to **Content**. For more information, see **Why don't all of the files in my project folder get deployed?** in ASP.NET Web Application Project Deployment FAQ). However, Web Deploy will not create a folder in the destination site unless there's at least one file to copy to it. Therefore, you'll add a *.txt* file to the folder to act as a placeholder so that the folder will be copied.

In **Solution Explorer**, right-click the *Elmah* folder, select **Add New Item**, and create a text file named *Placeholder.txt*. Put the following text in it: "This is a placeholder file to ensure that the folder gets deployed." and save the file. That's all you have to do in order to make sure that Visual Studio deploys this file and the folder it's in, because the **Build Action** property of *.txt* files is set to **Content** by default.

Summary

You have now completed all of the deployment set-up tasks. In the next tutorial, you'll deploy the Contoso University site to the test environment and test it there.

PREVIOUS NEXT

ASP.NET Web Deployment using Visual Studio: Deploying to Test

2/20/2020 • 18 minutes to read • Edit Online

by Tom Dykstra

This tutorial series shows how to deploy (publish) an ASP.NET web application to Azure App Service Web Apps or to a third-party hosting provider using Visual Studio 2017. For information about the series, see the first tutorial in the series.

For a current version of deploying to Azure, see Create an ASP.NET Core web app in Azure.

Overview

In this tutorial, you'll deploy an ASP.NET web application to Internet Information Server (IIS) on your local computer.

Generally when you develop an application, you run it and test it in Visual Studio. By default, web application projects in Visual Studio 2017 use IIS Express as the development web server. IIS Express behaves more like full IIS than the Visual Studio Development Server (also known as Cassini), which Visual Studio 2017 uses by default. But neither development web server works exactly like IIS. Consequently, an app could run and test correctly in Visual Studio but fail when it's deployed to IIS.

You can reliably test your application in two ways:

1. Deploy your application to IIS on your development computer using the same process that you'll use later to deploy it to your production environment.

   You can configure Visual Studio to use IIS when you run a web project, but that wouldn't test your deployment process. This method validates your deployment process and that your application runs correctly under IIS.

2. Deploy your application to a test environment similar to your production environment.

   The production environment for these tutorials is Web Apps in Azure App Service. The ideal test environment is an additional web app created in the Azure Service. Though it would be set up the same way as a production web app, you would only use it for testing.

Option 2 is the most reliable way to test. If you use option 2, you don't necessarily need to use option 1. However if you're deploying to a third-party hosting provider, option 2 might not be feasible or might be expensive, so this tutorial series shows both methods. Guidance for option 2 is provided in the Deploying to the Production Environment tutorial.

For more information about using web servers in Visual Studio, see Web Servers in Visual Studio for ASP.NET Web Projects.

Reminder: If you receive an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

Download the Contoso University starter project

Download and install the Contoso University Visual Studio starter solution and project. This solution contains the completed tutorial.

Download Starter Project

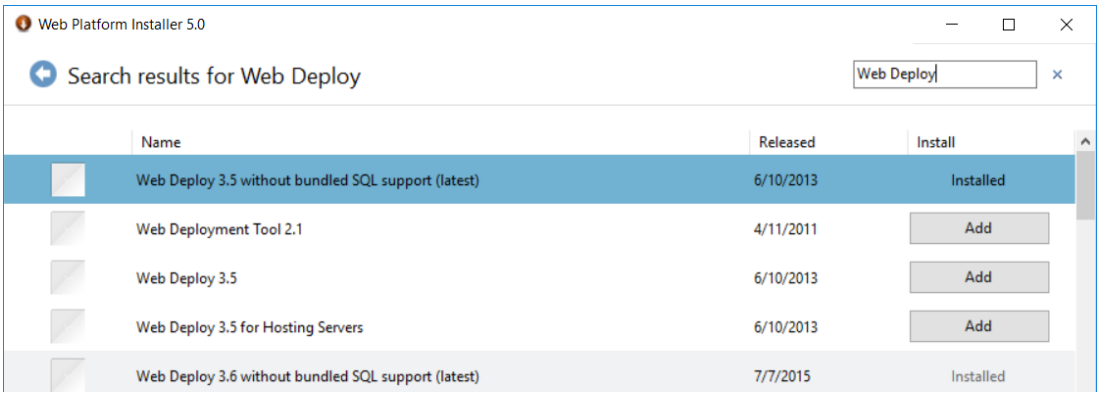
Install IIS

To deploy to IIS on your development computer, confirm that IIS and Web Deploy are installed. By default, Visual Studio installs Web Deploy, but IIS isn't included in the default Windows 10, Windows 8, or Windows 7 configuration. If you've already installed IIS and the default application pool is already set to .NET 4, skip to the next section.

1. It's recommended you use the Web Platform Installer (WPI) to install IIS and Web Deploy. WPI installs a recommended IIS configuration that includes IIS and Web Deploy prerequisites if necessary.

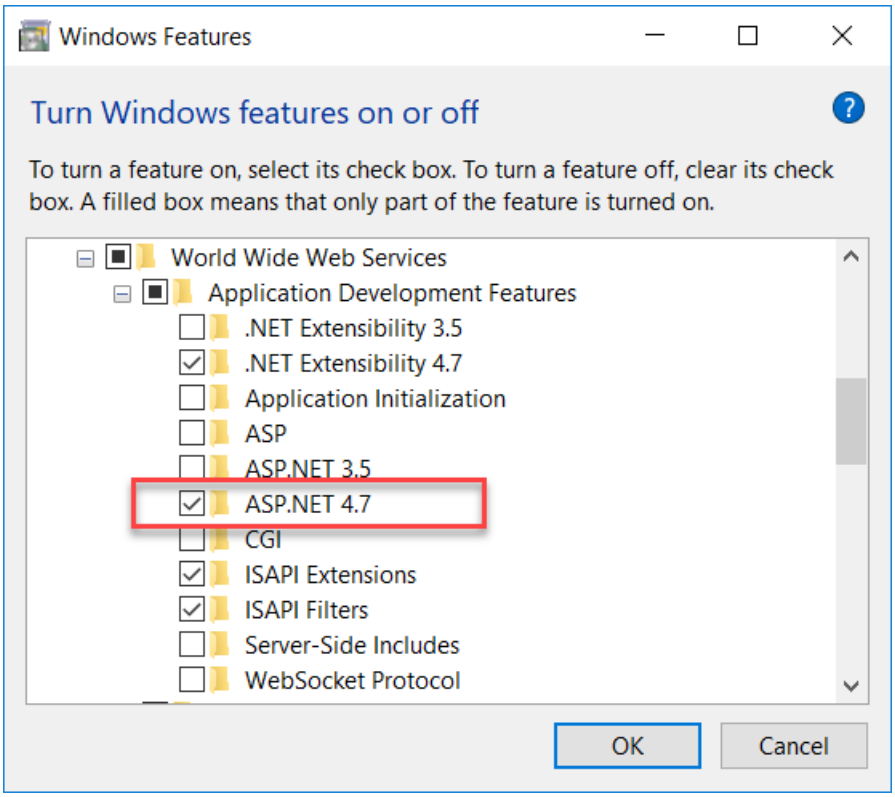
   If you've already installed IIS, Web Deploy, or any of their required components, the WPI installs only what is missing.

   - Use the Web Platform Installer to install IIS and Web Deploy:

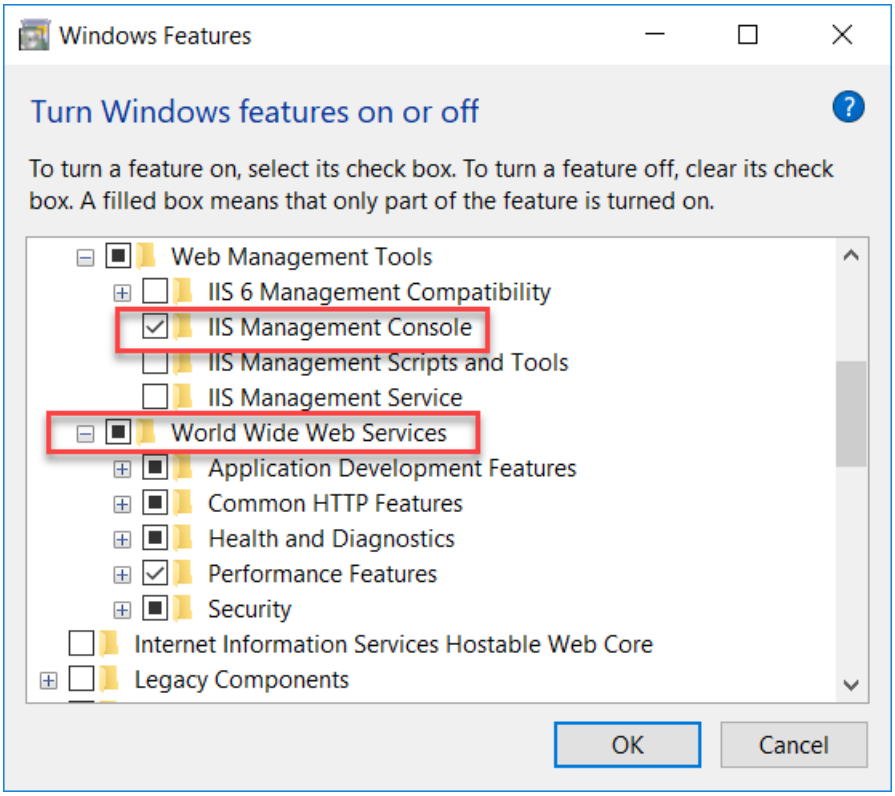


   You'll see messages indicating that IIS 7 will be installed. The link works for IIS 8 in Windows 8; but for Windows 8 and later, go through the following steps to make sure that ASP.NET 4.7 is installed:

   - Open **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off**.
   - Expand **Internet Information Services**, **World Wide Web Services**, and **Application Development Features**.
   - Confirm that **ASP.NET 4.7** is selected.



- Confirm that **World Wide Web Services** and **IIS Management Console** is selected. This installs IIS and IIS Manager.



- Select **OK**. Dialog box messages indicating installation is taking place appear.

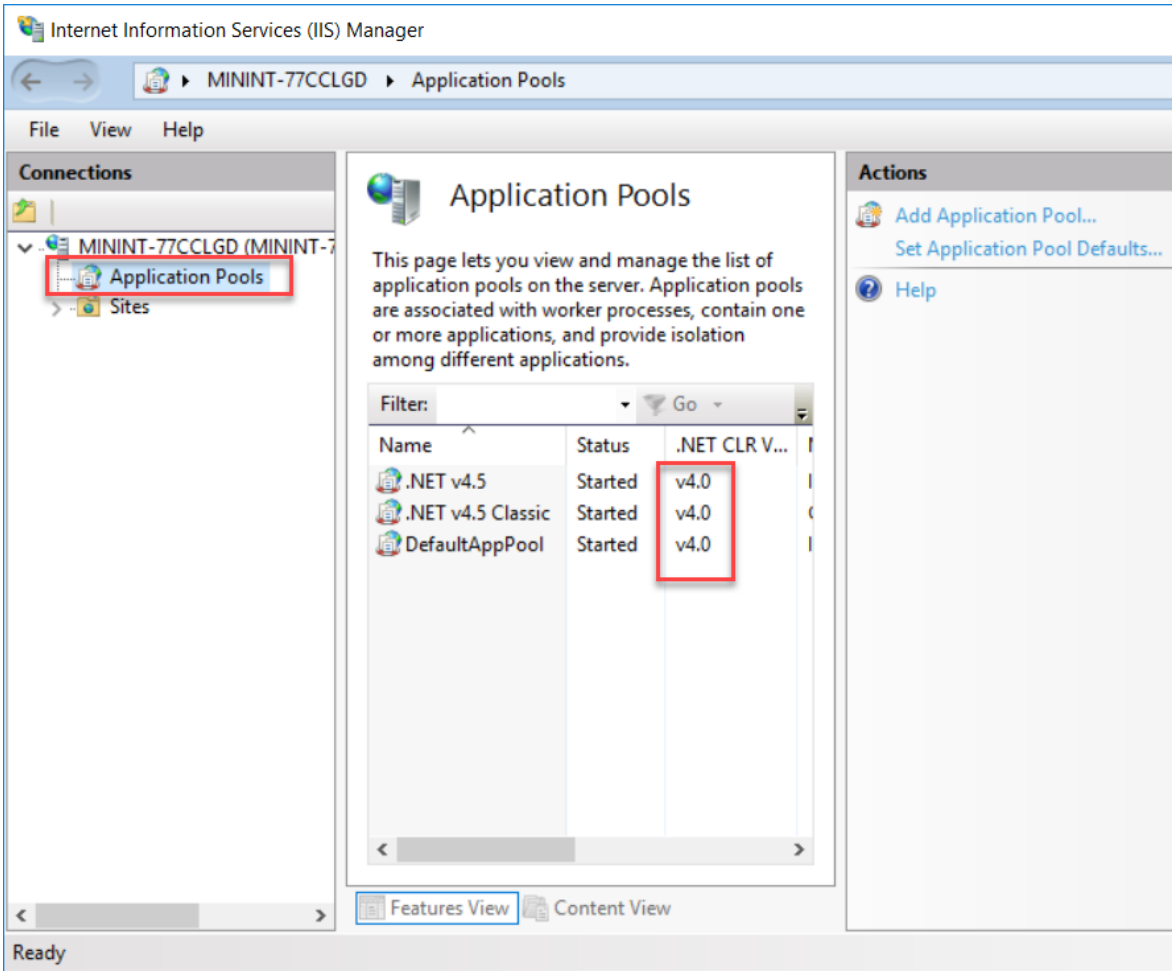
After installing IIS, run **IIS Manager** to make sure that the .NET Framework version 4 is assigned to the default application pool.

1. Press WINDOWS+R to open the **Run** dialog box.

   (On Windows 8 or later, enter "run" on the **Start** page. In Windows 7, select **Run** from the **Start** menu. If **Run** isn't in the **Start** menu, right-click the taskbar, select **Properties**, select the **Start Menu** tab, select **Customize**, and select **Run command**.)

2. Enter "inetmgr" and select **OK**.

3. In the **Connections** pane, expand the server node and select **Application Pools**. In the **Application Pools** pane if **DefaultAppPool** is assigned to the .NET framework version 4 as in the following illustration, skip to the next section.



4. If you see only two application pools and both are set to .NET Framework 2.0, install ASP.NET 4 in IIS.

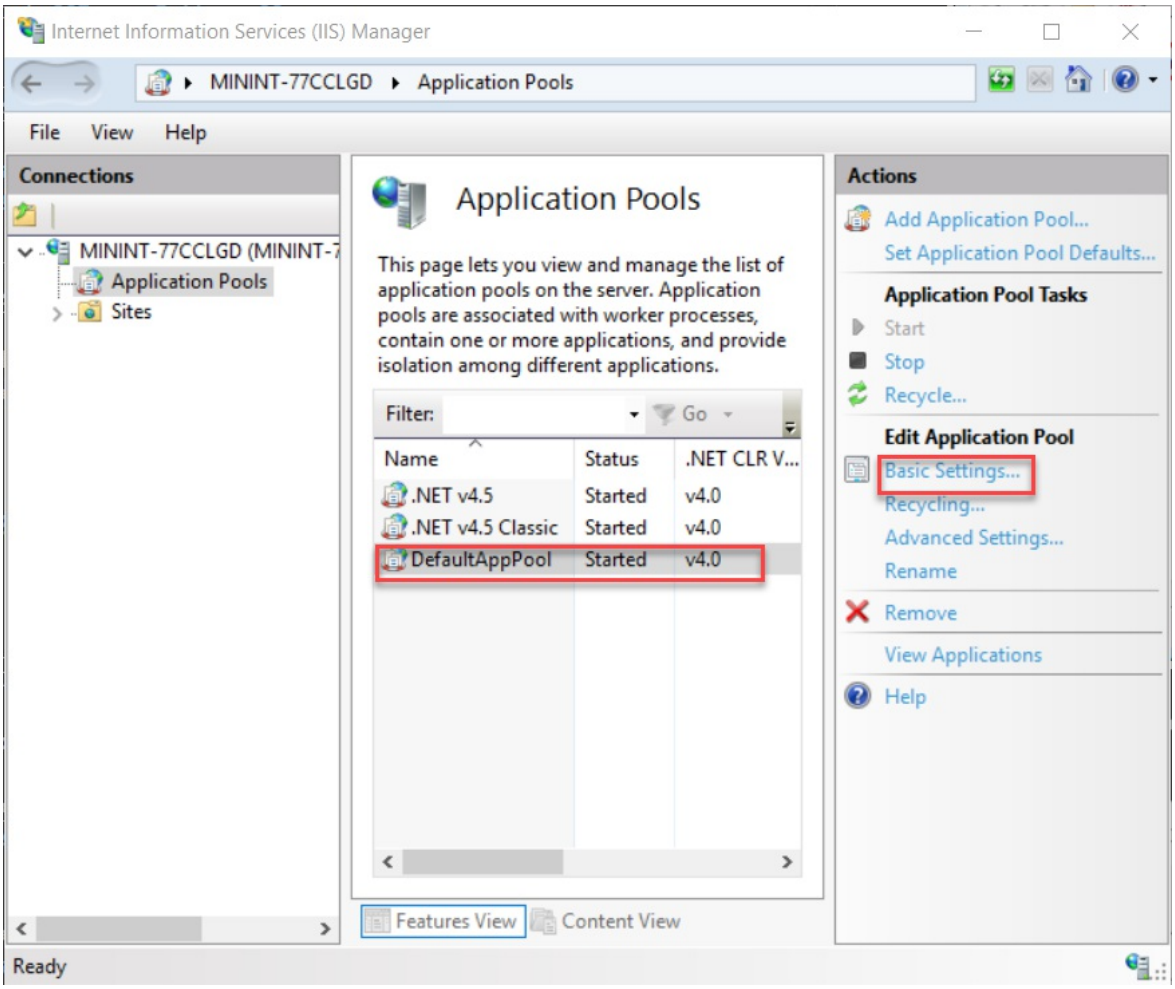
   For Windows 8 or later, see the previous section's instructions for making sure that ASP.NET 4.7 is installed or see How to install ASP.NET 4.5 on Windows 8 and Windows Server 2012. For Windows 7, open a command prompt window by right-clicking **Command Prompt** in the Windows **Start** menu and selecting **Run as Administrator**. Run aspnet_regiis.exe to install ASP.NET 4 in IIS using the following commands. (In 32-bit systems, replace "Framework64" with "Framework".)

```
cd %windir%\Microsoft.NET\Framework64\v4.0.30319
aspnet_regiis.exe -i
```

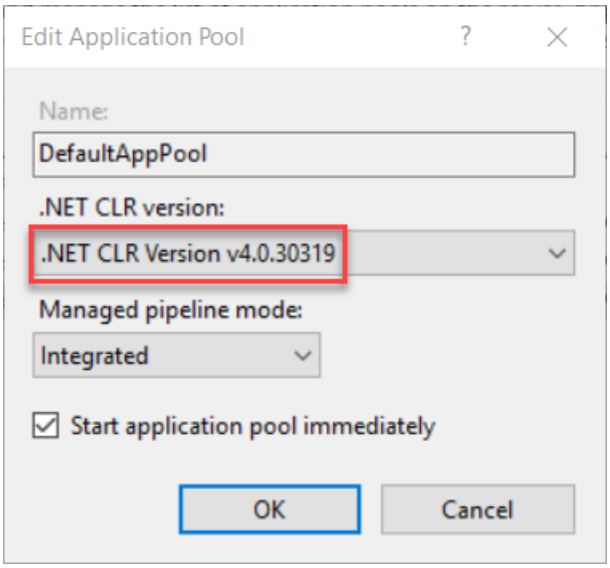
   This command creates new application pools for the .NET Framework 4, but the default application pool will remain set to 2.0. You're deploying an application that targets .NET 4 to that application pool, so change the application pool to .NET 4.

5. If you closed **IIS Manager**, run it again, expand the server node, and select **Application Pools**.

6. In the **Application Pools** pane, select **DefaultAppPool**. In the **Actions** pane, select **Basic Settings**.



7. In the **Edit Application Pool** dialog box, change the **.NET CLR version** to **.NET CLR v4.0.30319**. Select **OK**.



You're now ready to publish a web application to IIS. First, however, create databases for testing.

Install SQL Server Express

LocalDB isn't designed to work in IIS, so your test environment has to have SQL Server Express installed. If you're using Visual Studio 2010 SQL Server Express, it's already installed by default. If you're using Visual Studio 2012 or later, install SQL Server Express.

To install SQL Server Express, download and install it from Download Center: Microsoft SQL Server 2017 Express edition.

On the first page of the SQL Server Installation Center, select **New SQL Server stand-alone installation or add features to an existing installation** and follow the instructions accepting the default choices. In the installation wizard, accept the default settings. For more information about installation options, see Install SQL Server from the Installation Wizard (Setup).

Create SQL Server Express databases for the test environment

The Contoso University application has two databases:

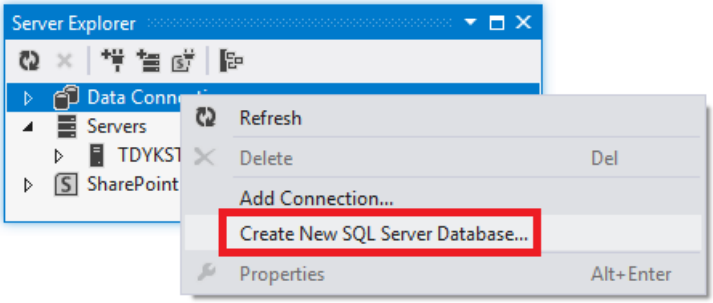
1. Membership database
2. Application database

You can deploy these databases to two separate databases or to a single database. Combining them makes database joins between them easier.

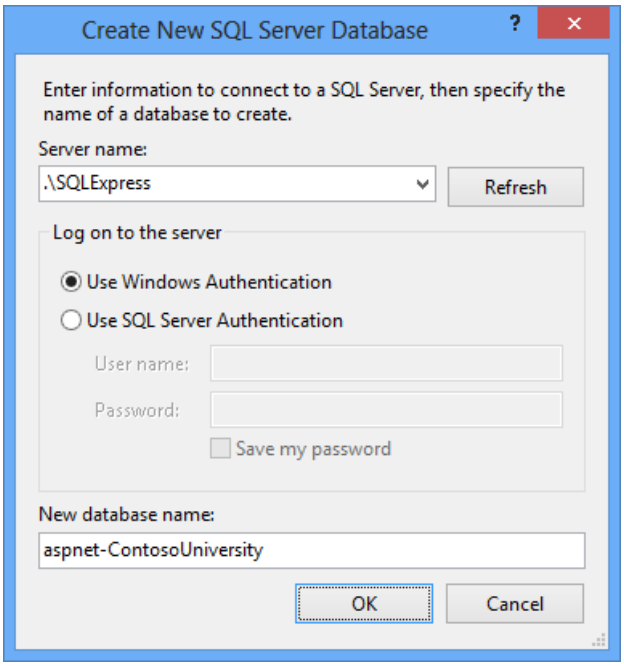
If you're deploying to a third-party hosting provider, your hosting plan might also give you a reason to combine them. For example, the provider might charge more for multiple databases or might not even allow more than one database.

In this tutorial, you'll deploy to two databases in the test environment and to one database in the staging and production environments.

From the **View** menu in Visual Studio, select **Server Explorer** (**Database Explorer** in Visual Web Developer). Right-click **Data Connections** and select **Create New SQL Server Database**.

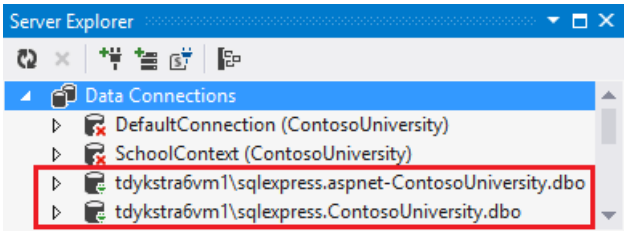


In the **Create New SQL Server Database** dialog box, enter ".\SQLExpress" in the **Server name** box and "aspnet-ContosoUniversity" in the **New database name** box. Select **OK**.



Follow the same procedure to create a new SQL Server Express School database named `ContosoUniversity`.

Server Explorer shows the two new databases.



Create a grant script for the new databases

When the application runs in IIS on your development computer, the application uses the default application pool's credentials to access the database. However, by default, the application pool doesn't have permission to open the databases. This means you need to run a script to grant that permission. In this section, you'll create that script and run it later to make sure that the application can open the databases when it runs in IIS.

In a text editor, copy the following SQL commands into a new file and save it as *Grant.sql*.

```
IF NOT EXISTS (SELECT name FROM sys.server_principals WHERE name = 'IIS APPPOOL\DefaultAppPool')
BEGIN
    CREATE LOGIN [IIS APPPOOL\DefaultAppPool]
      FROM WINDOWS WITH DEFAULT_DATABASE=[master],
      DEFAULT_LANGUAGE=[us_english]
END
GO
CREATE USER [ContosoUniversityUser]
  FOR LOGIN [IIS APPPOOL\DefaultAppPool]
GO
EXEC sp_addrolemember 'db_owner', 'ContosoUniversityUser'
GO
```

In Visual Studio, open the Contoso University solution. Right-click the solution (not one of the projects), and select **Add**. Select **Existing Item**, browse to *Grant.sql*, and open it.

> **NOTE**
>
> This script is designed to work with SQL Server Express 2012 or later and with the IIS settings in Windows 10, Windows 8, or Windows 7 as they are specified in this tutorial. If you're using a different version of SQL Server or Windows, or if you set up IIS on your computer differently, changes to this script might be required. For more information about SQL Server scripts, see SQL Server Books Online.

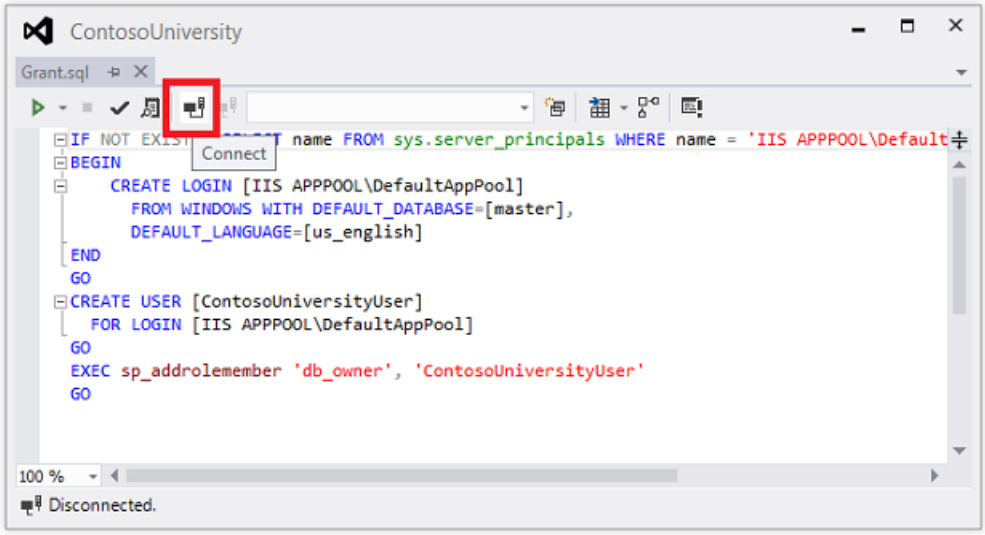
> **NOTE**
>
> **Security Note** This script gives `db_owner` permissions to the user that accesses the database at run time, which is what you'll have in the production environment. In some scenarios, you might want to specify a user that has full database schema update permissions only for deployment and specify for run time a different user that has permissions only to read and write data. For more information, see Reviewing the Automatic Web.config Changes for Code First Migrations later in this tutorial.

Run the grant script in the application database

You can configure the publish profile to run the grant script in the membership database during deployment because that database deployment uses the dbDacFx provider. You can't run scripts during Code First Migrations deployment, which is how you're deploying the application database. This means you have to manually run the script before deployment in the application database.

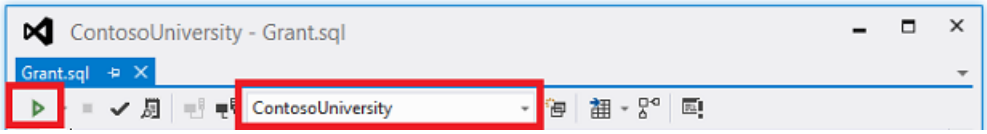
1. In Visual Studio, open the *Grant.sql* file that you created earlier.

2. Select **Connect**.



3. In the **Connect to Server** dialog box, enter *.\SQLExpress* as the **Server Name**. Select **Connect**.

4. In the database drop-down list select **ContosoUniversity**. Select **Execute**.



The default application pool identity now has sufficient permissions in the application database for Code First Migrations to create the database tables when the application runs.

Publish to IIS

There are several ways you can deploy to IIS using Visual Studio and Web Deploy:

- Use Visual Studio one-click publish.
- Publish from the command line.
- Create a deployment package and install it using IIS Manager. The package has a .zip file with all the files and metadata required to install a site in IIS.
- Create a deployment package and install it using the command line.

The process you went through in the previous tutorials to set up Visual Studio to automate deployment tasks applies to all of these methods. In these tutorials, you'll use the first two methods. For information about using deployment packages, see Deploying a web application by creating and installing a web deployment package in the Web Deployment Content Map for Visual Studio and ASP.NET.

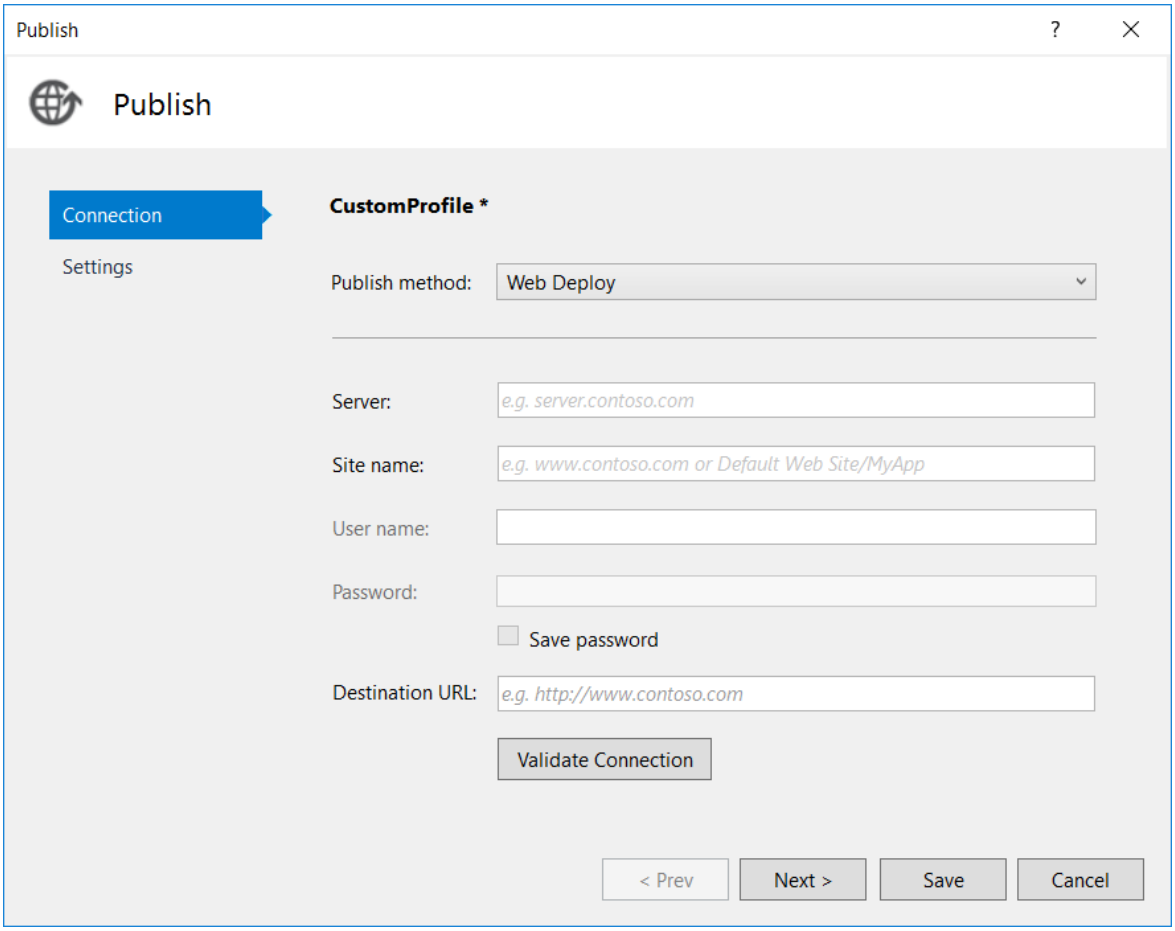
Before publishing, make sure that you're running Visual Studio in administrator mode. If you don't see **(Administrator)** in the title bar, close Visual Studio. In the Windows 8 (or later) **Start** page or the Windows 7 **Start** menu, right-click the Visual Studio icon and select **Run as Administrator**. Administrator mode is only required for publishing when you're publishing to IIS on the local computer.

Create the publish profile

1. In **Solution Explorer**, right-click the **ContosoUniversity** project (not the **ContosoUniversity.DAL** project). Select **Publish**. The **Publish** page appears.

2. Select **New Profile**. The **Pick a publish target** dialog box appears.

3. Select **IIS, FTP, etc**. Select **Create Profile**. The **Publish** wizard appears.



4. From the **Publish method** drop-down menu, select **Web Deploy**.

5. For **Server**, enter *localhost*.

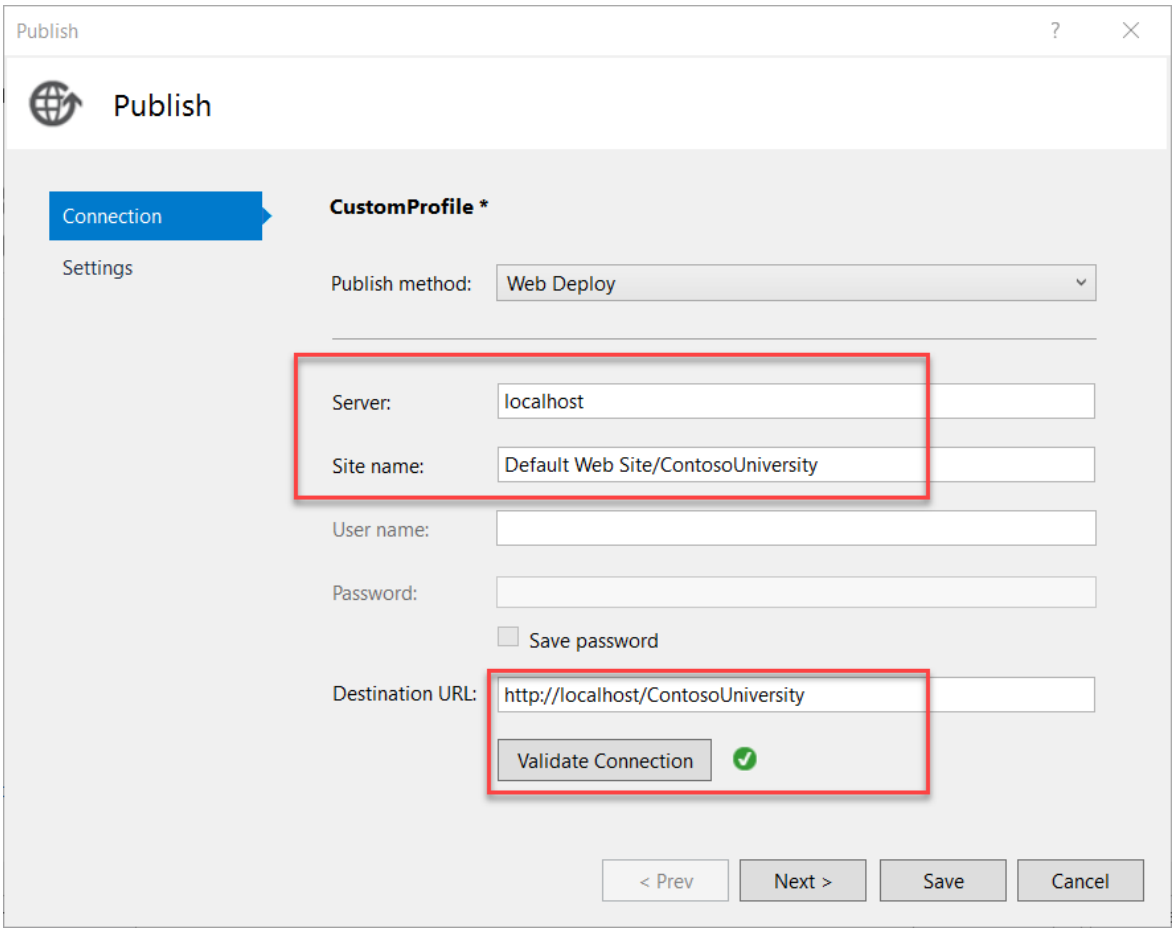
6. For **Site name**, enter *Default Web Site/ContosoUniversity*.

7. For **Destination URL**, enter *http://localhost/ContosoUniversity*.

   The **Destination URL** setting isn't required. When Visual Studio finishes deploying the application, it automatically opens your default browser to this URL. If you don't want the browser to open automatically after deployment, leave this box blank.

8. Select **Validate Connection** to verify that the settings are correct and you can connect to IIS on the local computer.

   A green check mark verifies that the connection is successful.



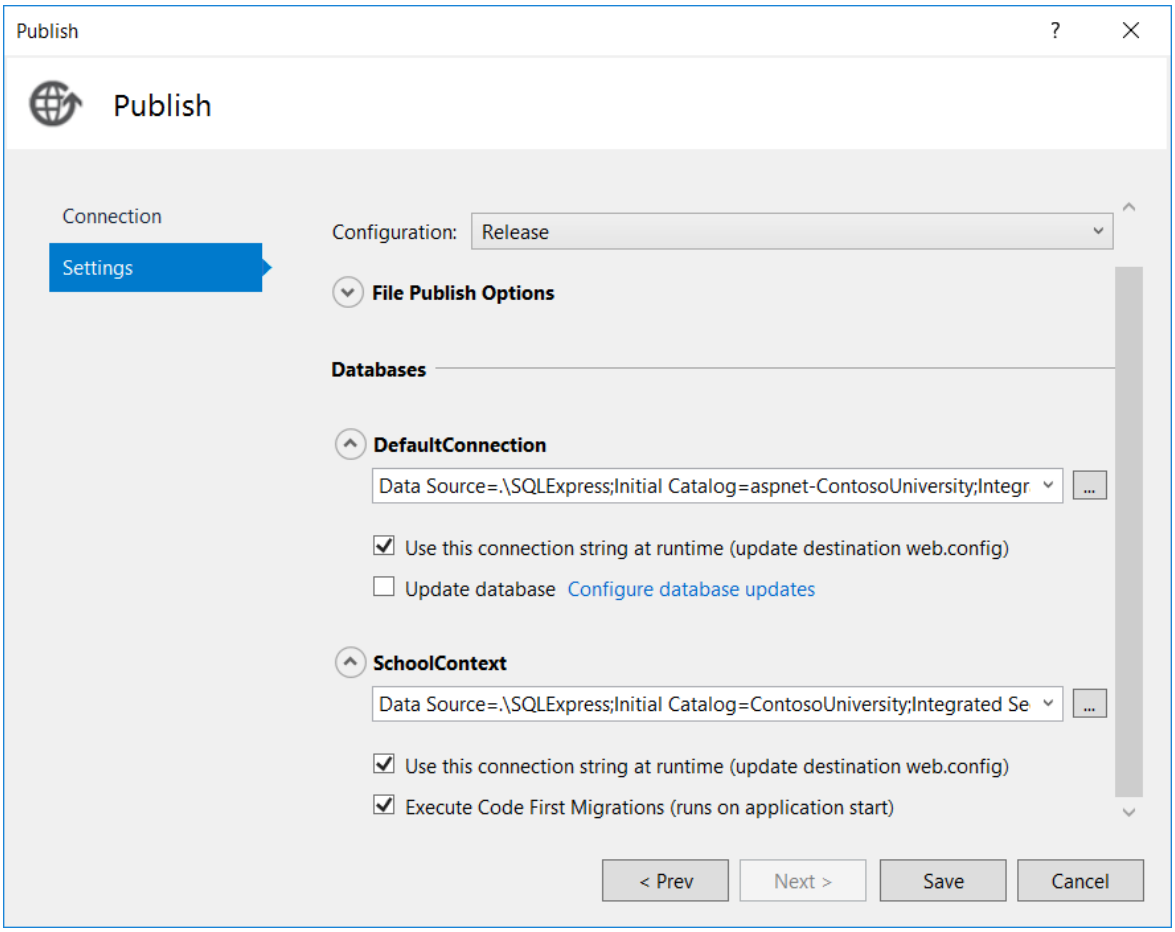
9. Select **Next** to advance to the **Settings** tab.

10. The **Configuration** drop-down box specifies the build configuration to deploy. Leave it set to the default value of **Release**. You won't be deploying Debug builds in this tutorial.

11. Expand **File Publish Options**. Select **Exclude files from the App_Data folder**.

    In the test environment, the application accesses the databases that you created in the local SQL Server Express instance, not the .mdf files in the *App_Data* folder.

12. Leave the **Precompile during publishing** and **Remove additional files at destination** check boxes cleared.



Precompiling is an option that is useful mainly for large sites. It can reduce startup time the first time a page is requested after the site is published.

You don't need to remove additional files since this is your first deployment and there won't be any files in the destination folder yet.

> **NOTE**
>
> If you select **Remove additional files at destination** for a subsequent deployment to the same site, make sure that you use the preview feature so that you see in advance which files will be deleted before you deploy. The expected behavior is that Web Deploy will delete files on the destination server that you have deleted in your project. However, the entire folder structure under the source and destination folders is compared; and in some scenarios, Web Deploy might delete files you don't want to delete.
>
> For example if you have a web application in a subfolder on the server when you deploy a project to the root folder, the subfolder will be deleted. You might have one project for the main site at contoso.com and another project for a blog at contoso.com/blog. The blog application is in a subfolder. If you select **Remove additional files at destination** when you deploy the main site, the blog application will be deleted.
>
> For another example, your App_Data folder might get deleted unexpectedly. Certain databases such as SQL Server Compact store database files in the App_Data folder. After the initial deployment, you don't want to keep copying the database files in subsequent deployments, so you select **Exclude App_Data** on the Package/Publish Web tab. After you do that if you have **Remove additional files at destination** selected, your database files and the App_Data folder itself will be deleted the next time you publish.

Configure deployment for the membership database

The following steps apply to the **DefaultConnection** database in the dialog box's **Databases** section.

1. In the **Remote connection string** box, enter the following connection string that points to the new SQL Server Express membership database.

```
Data Source=.\SQLExpress;Initial Catalog=aspnet-ContosoUniversity;Integrated Security=True
```

The deployment process puts this connection string in the deployed Web.config file because **Use this connection string at runtime** is selected.

You can also get the connection string from **Server Explorer**. In **Server Explorer**, expand **Data Connections** and select the **<machinename>\sqlexpress.aspnet-ContosoUniversity** database, then from the **Properties** window copy the **Connection String** value. That connection string will have one additional setting that you can delete: `Pooling=False`.

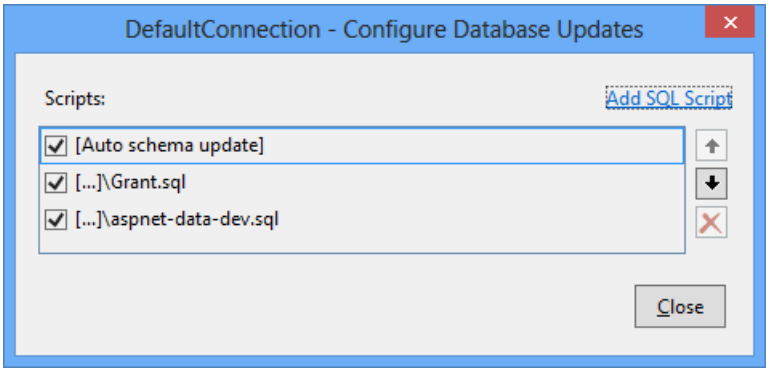
2. Select **Update database**.

   This causes the database schema to be created in the destination database during deployment. In next steps, you specify the additional scripts that you need to run: one to grant database access to the default application pool and one to deploy data.

3. Select **Configure database updates**.

4. In the **Configure Database Updates** dialog box, select **Add SQL Script**. Navigate to the *Grant.sql* script that you saved earlier in the solution folder.

5. Repeat the process to add the *aspnet-data-dev.sql* script.



6. Select **Close**.

Configure deployment for the application database

When Visual Studio detects an Entity Framework `DbContext` class, it creates an entry in the **Databases** section that has an **Execute Code First Migrations** check box instead of an **Update Database** check box. For this tutorial, you'll use that check box to specify Code First Migrations deployment.

In some scenarios, you might be using a `DbContext` database but you want to use the dbDacFx provider instead of Migrations to deploy the database. In that case, see How do I deploy a Code First database without Migrations? in the ASP.NET Web Deployment FAQ on MSDN.

The following steps apply to the **SchoolContext** database in the dialog box's **Databases** section.

1. In the **Remote connection string** box, enter the following connection string that points to the new SQL Server Express application database.

```
Data Source=.\SQLExpress;Initial Catalog=ContosoUniversity;Integrated Security=True
```

   The deployment process puts this connection string in the deployed Web.config file because **Use this connection string at runtime** is selected.

   You can also get the application database connection string from **Server Explorer** in the same way you got the membership database connection string.

2. Select **Execute Code First Migrations (runs on application start)**.

   This option causes the deployment process to configure the deployed Web.config file to specify the `MigrateDatabaseToLatestVersion` initializer. This initializer automatically updates the database to the latest version when the application accesses the database for the first time after deployment.

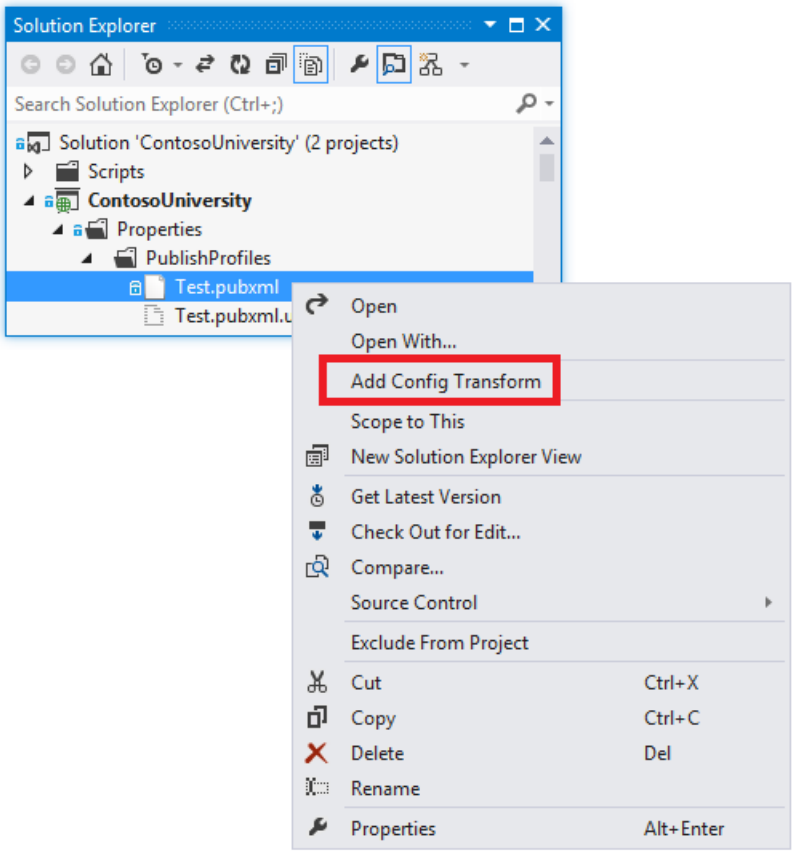
Configure publish profile transforms

1. Select **Close**. Select **Yes** when you are asked if you want to save changes.

2. In **Solution Explorer**, expand **Properties**, expand **PublishProfiles**.

3. Right-click *CustomProfile.pubxml* and rename it *Test.pubxml*.

4. Right-click *Test.pubxml*. Select **Add Config Transform**.



   Visual Studio creates the *Web.Test.config* transform file and opens it.

5. In the *Web.Test.config* transform file, insert the following code immediately after the opening configuration tag.

```
<appSettings>
    <add key="Environment" value="Test" xdt:Transform="SetAttributes" xdt:Locator="Match(key)"/>
</appSettings>
```

   When you use the Test publish profile, this transform sets the environment indicator to "Test". In the deployed site, you'll see "(Test)" after the "Contoso University" H1 heading.

6. Save and close the file.

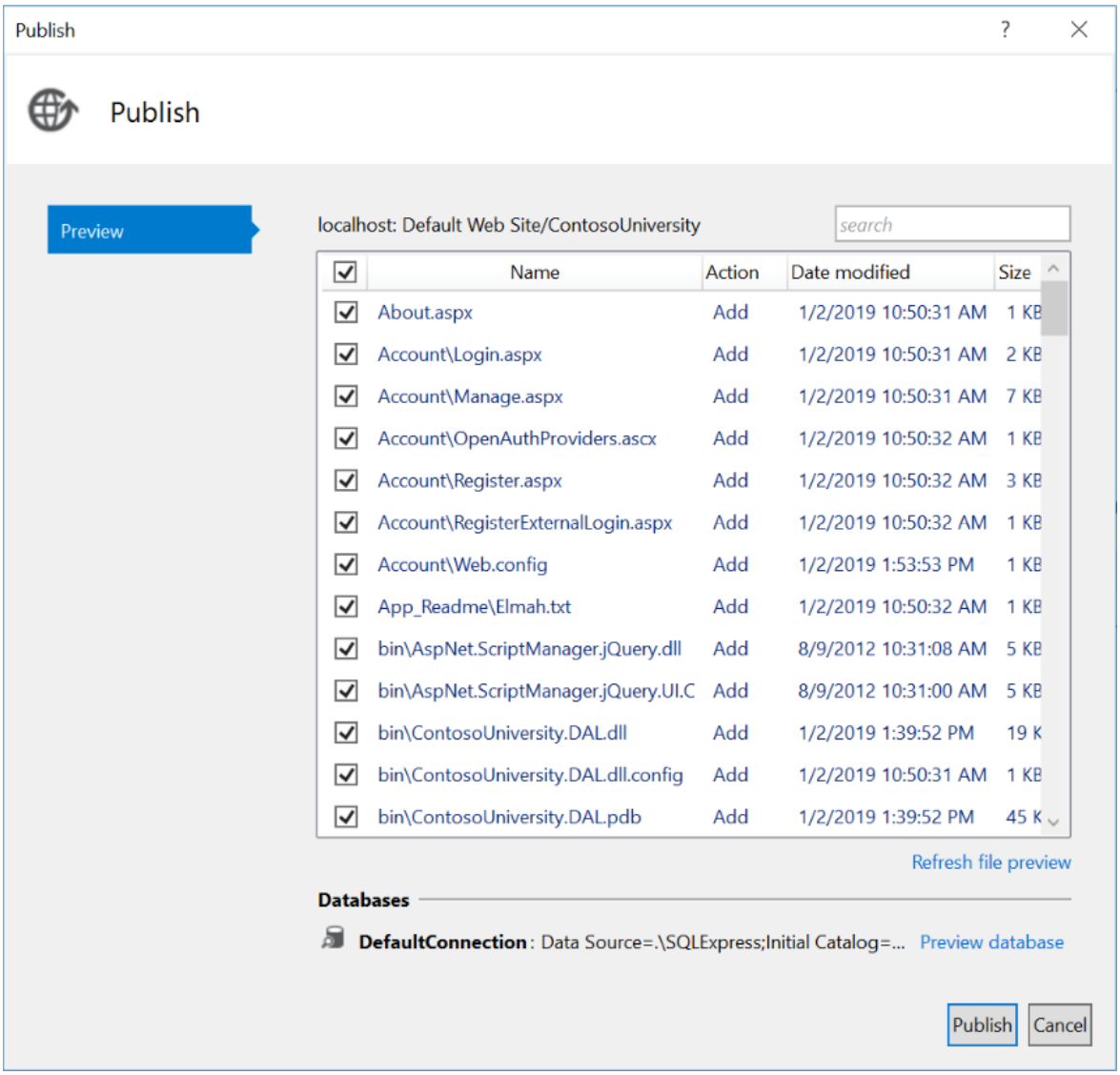
7. Right-click the *Web.Test.config* file and select **Preview Transform** to make sure that the transform you coded produces the expected changes.

   The **Web.config Preview** window shows the result of applying both the *Web.Release.config* transforms and the *Web.Test.config* transforms.

Preview the deployment updates

1. Open the **Publish Web** wizard again (right-click the ContosoUniversity project, select **Publish**, then **Preview**).

2. In the **Preview** dialog box, select **Start Preview** to see a list of the files that will be copied.

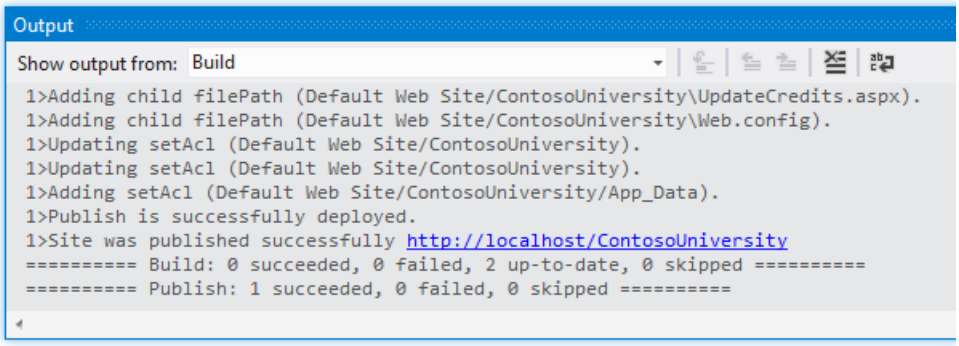


You can also select the **Preview database** link to see the scripts that will run in the membership database. (No scripts are run for Code First Migrations deployment, so there's nothing to preview for the application database.)

3. Select **Publish**.

If Visual Studio is not in administrator mode, you might get a permissions error message. In that case, close Visual Studio, open it in administrator mode, and try to publish again.

If Visual Studio is in administrator mode, the **Output** window reports successful build and publish.



If you entered the URL in the **Destination URL** box on the publish profile **Connection** tab, the browser automatically opens to the Contoso University Home page running in IIS on your computer.

Test in the test environment

Notice that the environment indicator shows "(Test)" instead of "(Dev)," which shows that the *Web.config* transformation for the environment indicator was successful.

Run the **Instructors** page to verify that Code First seeded the database with instructor data. When you select this page, it may take a few minutes to load because Code First creates the database and then runs the `Seed` method. (It didn't do that when you were on the home page because the application didn't try to access the database yet.)

Select the **Students** tab to verify that the deployed database has no students.

Select **Add Students** from the **Students** menu. Add a student, and then view the new student in the **Students** page. This verifies that you can successfully write to the database.

From the **Courses** menu, select **Update Credits**. The **Update Credits** page requires administrator permissions, so the **Log In** page is displayed. Enter the administrator account credentials that you created earlier ("admin" and "devpwd"). The **Update Credits** page is displayed. This verifies that the administrator account that you created in the previous tutorial was correctly deployed to the test environment.

Verify that an *ELMAH* folder exists in the *c:\inetpub\wwwroot\ContosoUniversity* folder with only the placeholder file in it.

Review the automatic Web.config changes for Code First Migrations

Open the *Web.config* file in the deployed application at *C:\inetpub\wwwroot\ContosoUniversity* and you can see where the deployment process configured Code First Migrations to automatically update the database to the latest version.

```
<entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlCeConnectionFactor
        <parameters>
            <parameter value="System.Data.SqlServerCe.4.0" />
        </parameters>
    </defaultConnectionFactory>
    <contexts>
        <context type="ContosoUniversity.DAL.SchoolContext, ContosoUniversity.DAL">
            <databaseInitializer type="System.Data.Entity.MigrateDatabaseToLatestVersion`2[
                <parameters>
                    <parameter value="SchoolContext_DatabasePublish" />
                </parameters>
            </databaseInitializer>
        </context>
    </contexts>
</entityFramework>
```

The deployment process also created a new connection string for Code First Migrations to use exclusively for updating the database schema:

```
<connectionStrings>
  <add name="DefaultConnection" providerName="System.Data.SqlClient" connectionString="Data
  <add name="SchoolContext" providerName="System.Data.SqlClient" connectionString="Data Sou
  <add name="SchoolContext_DatabasePublish" connectionString="Data Source=.\SQLExpress;Init
</connectionStrings>
```

This additional connection string enables you to specify one user account for database schema updates and a different user account for application data access. For example, you could assign the **db_owner** role to Code First Migrations and **db_datareader** with **db_datawriter** roles to the application. This is a common defense-in-depth pattern that prevents potentially malicious code in the application from changing the database schema.

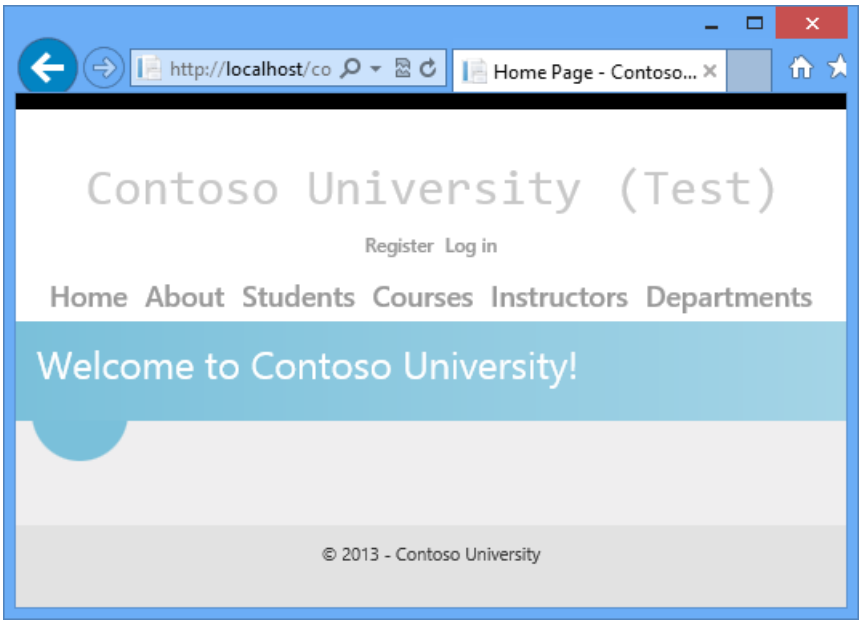
(For example, this might happen in a successful SQL injection attack.) These tutorials don't use this pattern. To implement this pattern in your scenario, take these steps:

1. In the **Publish Web** wizard under the **Settings** tab, enter the connection string that specifies a user with full database schema update permissions. Clear the **Use this connection string at runtime** check box. In the deployed Web.config file, this becomes the `DatabasePublish` connection string.

2. Create a Web.config file transformation for the connection string that you want the application to use at run time.

Summary

You've now deployed your application to IIS on your development computer and tested it there.



This verifies that the deployment process copied the application's content to the right location (excluding the files that you did not want to deploy) and also that Web Deploy configured IIS correctly during deployment. In the next tutorial, you'll run one more test that finds a deployment task that has not yet been done: setting folder permissions on the *Elm ah* folder.

More information

For information about running IIS or IIS Express in Visual Studio, see the following resources:

- IIS Express Overview on the IIS.net site.
- Introducing IIS Express on Scott Guthrie's blog.
- Web Servers in Visual Studio for ASP.NET Web Projects.
- Core Differences Between IIS and the ASP.NET Development Server on the ASP.NET site.

For information about what issues might arise when your application runs in medium trust, see Hosting ASP.NET Applications in Medium Trust on the four Guys from Rolla site.

PREVIOUS NEXT

ASP.NET Web Deployment using Visual Studio: Setting Folder Permissions

2/20/2020 • 3 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This tutorial series shows you how to deploy (publish) an ASP.NET web application to Azure App Service Web Apps or to a third-party hosting provider, by using Visual Studio 2012 or Visual Studio 2010. For information about the series, see the first tutorial in the series.

Overview

In this tutorial, you set folder permissions for the *Elmah* folder in the deployed web site so that the application can create log files in that folder.

When you test a web application in Visual Studio using the Visual Studio Development Server (Cassini) or IIS Express, the application runs under your identity. You are most likely an administrator on your development computer and have full authority to do anything to any file in any folder. But when an application runs under IIS, it runs under the identity defined for the application pool that the site is assigned to. This is typically a system-defined account that has limited permissions. By default it has read and execute permissions on your web application's files and folders, but it doesn't have write access.

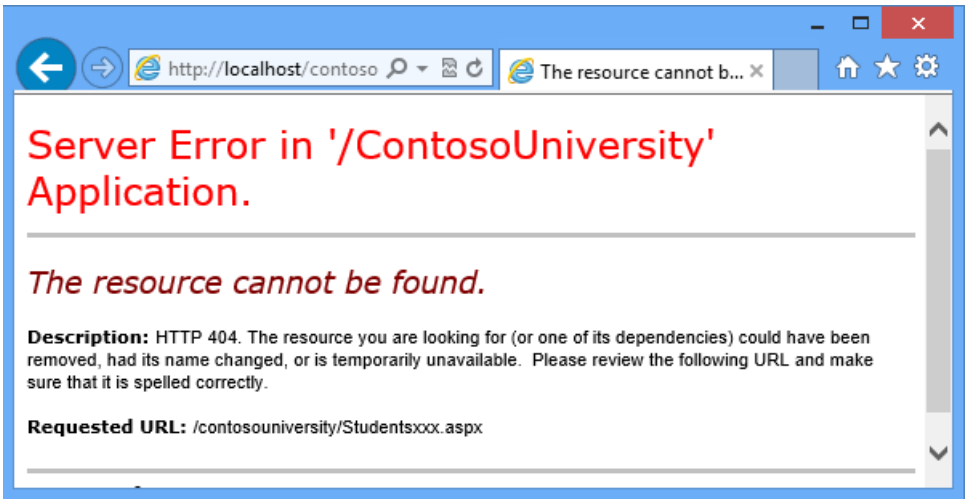
This becomes an issue if your application creates or updates files, which is a common need in web applications. In the Contoso University application, Elmah creates XML files in the *Elmah* folder in order to save details about errors. Even if you don't use something like Elmah, your site might let users upload files or perform other tasks that write data to a folder in your site.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

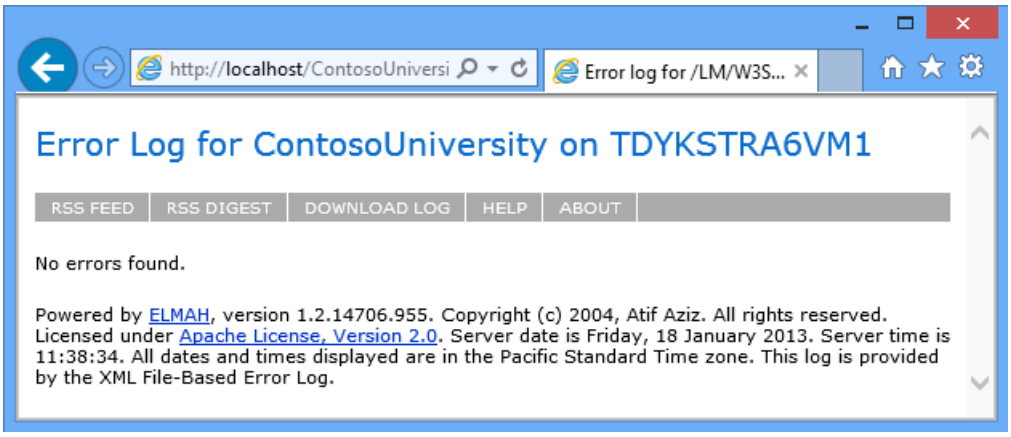
Test error logging and reporting

To see how the application doesn't work correctly in IIS (although it did when you tested it in Visual Studio), you can cause an error that would normally be logged by Elmah, and then open the Elmah error log to see the details. If Elmah was unable to create an XML file and store the error details, you see an empty error report.

Open a browser and go to `http://localhost/ContosoUniversity`, and then request an invalid URL like *Studentsxxx.aspx*. You see a system-generated error page instead of the *GenericErrorPage.aspx* page because the `customErrors` setting in the Web.config file is "RemoteOnly" and you are running IIS locally:



Now run *Elmah.axd* to see the error report. After you log in with the administrator account credentials ("admin" and "devpwd"), you see an empty error log page because Elmah was unable to create an XML file in the *Elmah* folder:



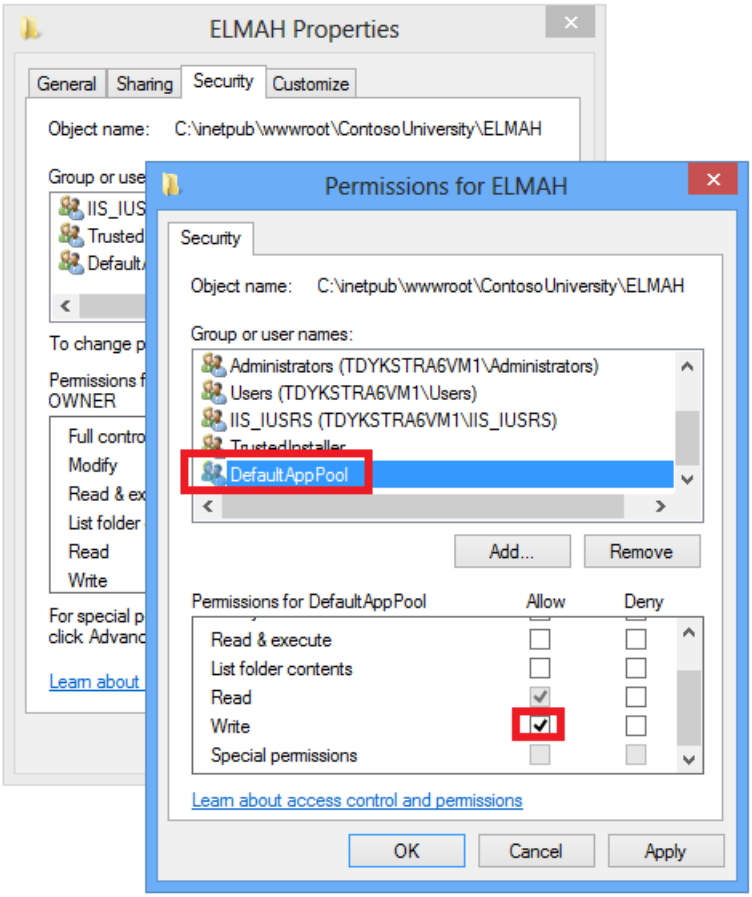
Set write permission on the Elmah folder

You can set folder permissions manually or you can make it an automatic part of the deployment process. Making it automatic requires complex MSBuild code, and since you only have to do this the first time you deploy, the following steps how to do it manually. (For information about how to make this part of the deployment process, see Setting Folder Permissions on Web Publish on Sayed Hashimi's blog.)

1. In **File Explorer**, navigate to *C:\inetpub\wwwroot\ContosoUniversity*. Right-click the *Elmah* folder, select **Properties**, and then select the **Security** tab.

2. Click **Edit**.

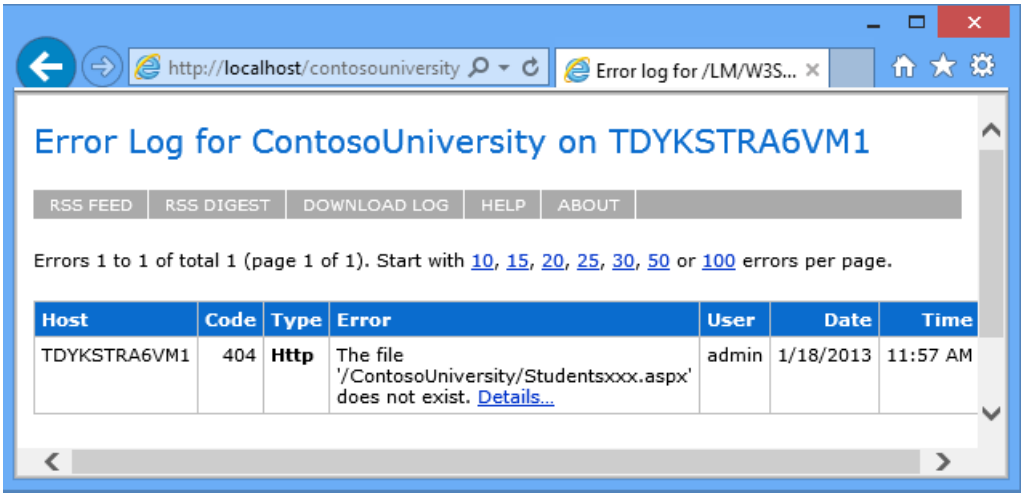
3. In the **Permissions for Elmah** dialog box, select **DefaultAppPool**, and then select the **Write** check box in the **Allow** column.



(If you don't see **DefaultAppPool** in the **Group or user names** list, you probably used some other method than the one specified in this tutorial to set up IIS and ASP.NET 4 on your computer. In that case, find out what identity is used by the application pool assigned to the Contoso University application, and grant write permission to that identity. See the links about application pool identities at the end of this tutorial.) Click **OK** in both dialog boxes.

Retest error logging and reporting

Test by causing an error again in the same way (request a bad URL) and run the **Error Log** page. This time the error appears on the page.



Summary

You have now completed all of the tasks necessary to get Contoso University working correctly in IIS on your local computer. In the next tutorial, you will make the site publicly available by deploying it to Azure.

More information

In this example, the reason why Elmah was unable to save log files was fairly obvious. You can use IIS tracing in cases where the cause of the problem is not so obvious; see Troubleshooting Failed Requests Using Tracing in IIS 7 on the IIS.net site.

For more information about how to grant permissions to application pool identities, see Application Pool Identities and Secure Content in IIS Through File System ACLs on the IIS.net site.

PREVIOUS NEXT

ASP.NET Web Deployment using Visual Studio: Deploying to Production

9/16/2020 • 17 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This tutorial series shows you how to deploy (publish) an ASP.NET web application to Azure App Service Web Apps or to a third-party hosting provider, by using Visual Studio 2012 or Visual Studio 2010. For information about the series, see the first tutorial in the series.

Overview

In this tutorial, you set up a Microsoft Azure account, create staging and production environments, and deploy your ASP.NET web application to the staging and production environments by using the Visual Studio one-click publish feature.

If you prefer, you can deploy to a third-party hosting provider. Most of the procedures described in this tutorial are the same for a hosting provider or for Azure, except that each provider has its own user interface for account and web site management. You can find a hosting provider in the gallery of providers on the Microsoft.com web site.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the Troubleshooting page in this tutorial series.

Get a Microsoft Azure account

If you don't already have an Azure account, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

Create a staging environment

> **NOTE**
>
> Since this tutorial was written, Azure App Service added a new feature to automate many of the processes for creating staging and production environments. See Set up staging environments for web apps in Azure App Service.

As explained in the Deploy to the Test Environment tutorial, the most reliable test environment is a web site at the hosting provider that's just like the production web site. At many hosting providers you would have to weigh the benefits of this against significant additional cost, but in Azure you can create an additional free web app as your staging app. You also need a database, and the additional expense for that over the expense of your production database will be either none or minimal. In Azure you pay for the amount of database storage you use rather than for each database, and the amount of additional storage you'll use in staging will be minimal.

As explained in the Deploy to the Test Environment tutorial, in staging and production you're going to deploy your two databases into one database. If you wanted to keep them separate, the process would be the same except that you'd create an additional database for each environment and you would select the correct destination string for each database when you create the publish profile.

In this section of the tutorial you'll create a web app and database to use for the staging environment, and you'll deploy to staging and test there before creating and deploying to the production environment.

> **NOTE**
>
> The following steps show how to create a web app in Azure App Service by using the Azure management portal. In the latest version of the Azure SDK, you can also do this without leaving Visual Studio, by using Server Explorer. In Visual Studio 2013, you can also create a web app directly from the Publish dialog. For more information, see Create an ASP.NET web app in Azure App Service.

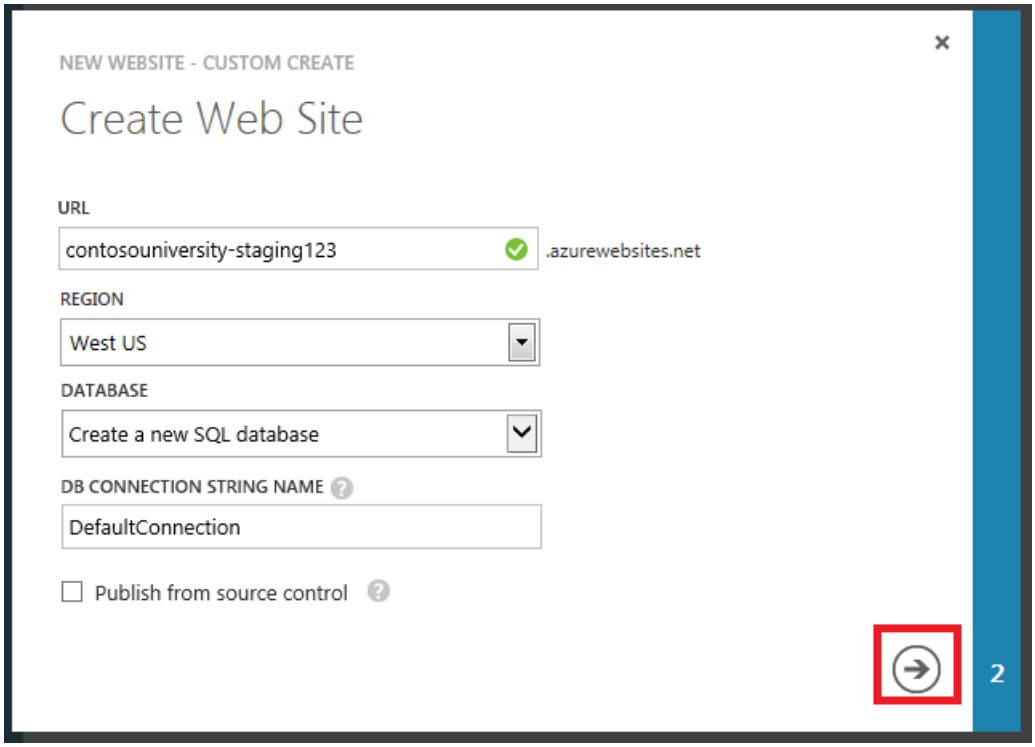
1. In the Azure Management Portal, click **Websites**, and then click **New**.
2. Click **Website**, and then click **Custom Create**.

   The **New Website - Custom Create** wizard opens. The **Custom Create** wizard enables you to create a web site and a database at the same time.
3. In the **Create Website** step of the wizard, enter a string in the **URL** box to use as the unique URL for your application's staging environment. For example, enter ContosoUniversity-staging123 (including random numbers at the end to make it unique in case ContosoUniversity-staging is taken).

   The complete URL will consist of what you enter here plus the suffix that you see next to the text box.
4. In the **Region** drop-down list, choose the region that is closest to you.

   This setting specifies which data center your web app will run in.
5. In the **Database** drop-down list, choose **Create a new SQL database**.
6. In the **DB Connection String Name** box, leave the default value, *DefaultConnection*.
7. Click the arrow that points to the right at the bottom of the box.

   The following illustration shows the **Create Website** dialog with sample values in it. The URL and Region that you have entered will be different.



   The wizard advances to the **Specify database settings** step.

8. In the **Name** box, enter *ContosoUniversity* plus a random number to make it unique, for example *ContosoUniversity123*.

9. In the **Server** box, select **New SQL Database Server**.

10. Enter an administrator name and password.

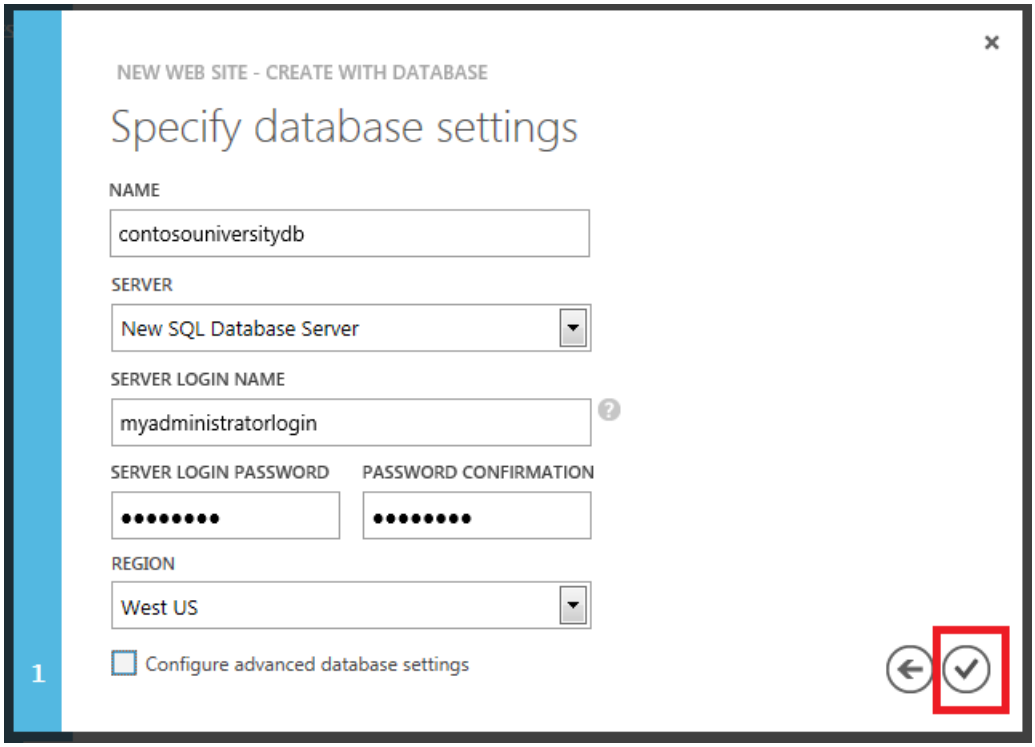
    You aren't entering an existing name and password here. You're entering a new name and password that you're defining now to use later when you access the database.

11. In the **Region** box, choose the same region that you chose for the web app.

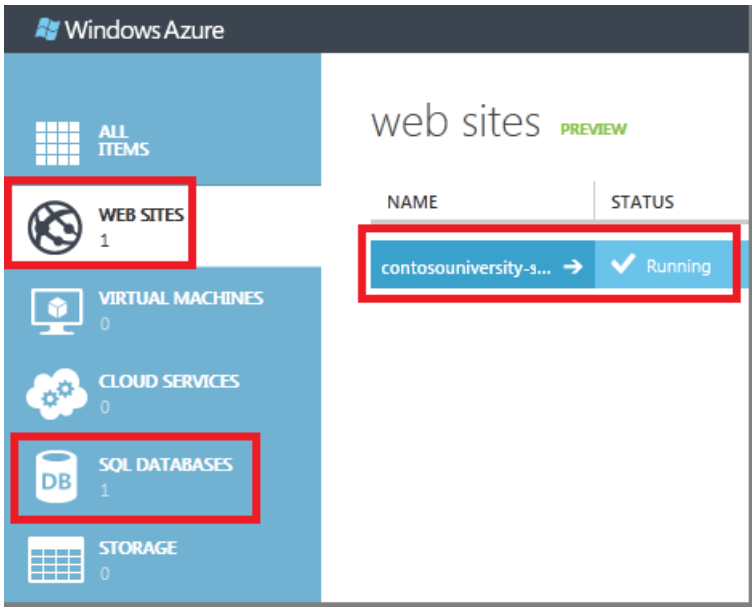
    Keeping the web server and the database server in the same region gives you the best performance and minimizes expenses.

12. Click the check mark at the bottom of the box to indicate that you're finished.

    The following illustration shows the **Specify database settings** dialog with sample values in it. The values you have entered may be different.



The Management Portal returns to the Websites page, and the **Status** column shows that the web app is being created. After a while (typically less than a minute), the **Status** column shows that the web app was successfully created. In the navigation bar at the left, the number of web apps you have in your account appears next to the **Websites** icon, and the number of databases appears next to the **SQL Databases** icon.



Your web app name will be different from the example app in the illustration.

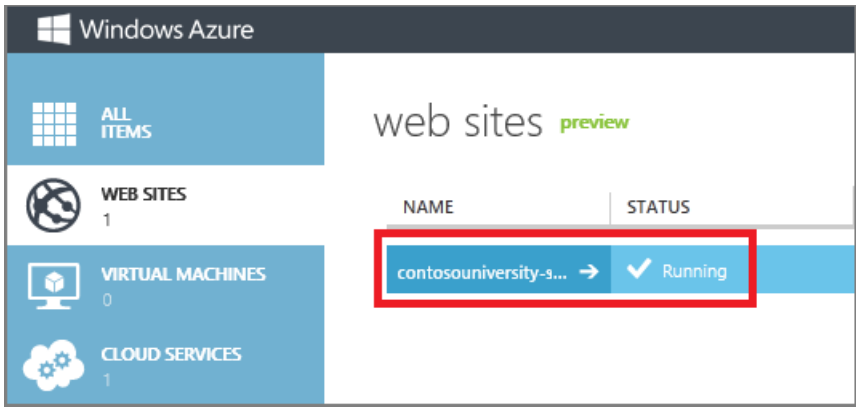
Deploy the application to staging

Now that you have created a web app and database for the staging environment, you can deploy the project to it.

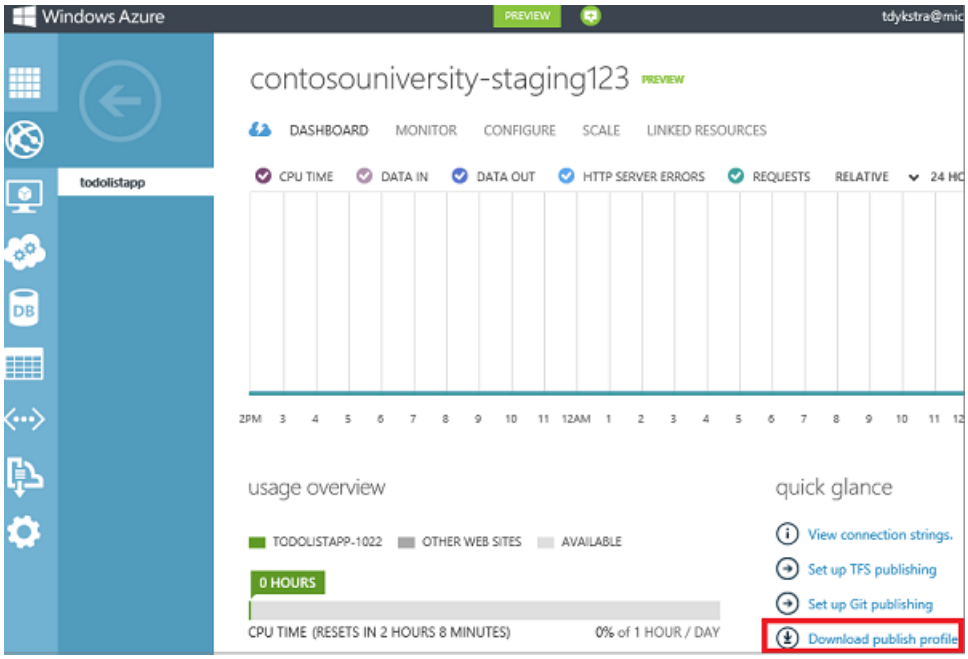
> **NOTE**
>
> These instructions show how to create a publish profile by downloading a *.publishsettings* file, which works not only for Azure but also for third-party hosting providers. The latest Azure SDK also enables you to connect directly to Azure from Visual Studio, and choose from a list of web apps that you have in your Azure account. In Visual Studio 2013, you can sign in to Azure from the **Web Publish** dialog or from the **Server Explorer** window. For more information, see Create an ASP.NET web app in Azure App Service.

Download the .publishsettings file

1. Click the name of the web app that you just created.



2. Under **Quick glance** in the **Dashboard** tab, click **Download publish profile**.



This step downloads a file that contains all of the settings that you need in order to deploy an application to your web app. You'll import this file into Visual Studio so you don't have to enter this information manually.

3. Save the *.publishsettings* file in a folder that you can access from Visual Studio.

> **WARNING**
>
> Security - The *.publishsettings* file contains your credentials (unencoded) that are used to administer your Azure subscriptions and services. The security best practice for this file is to store it temporarily outside your source directories (for example in the Libraries\Documents folder), and then delete it once the import has completed. A malicious user who gains access to the *.publishsettings* file can edit, create, and delete your Azure services.

Create a publish profile

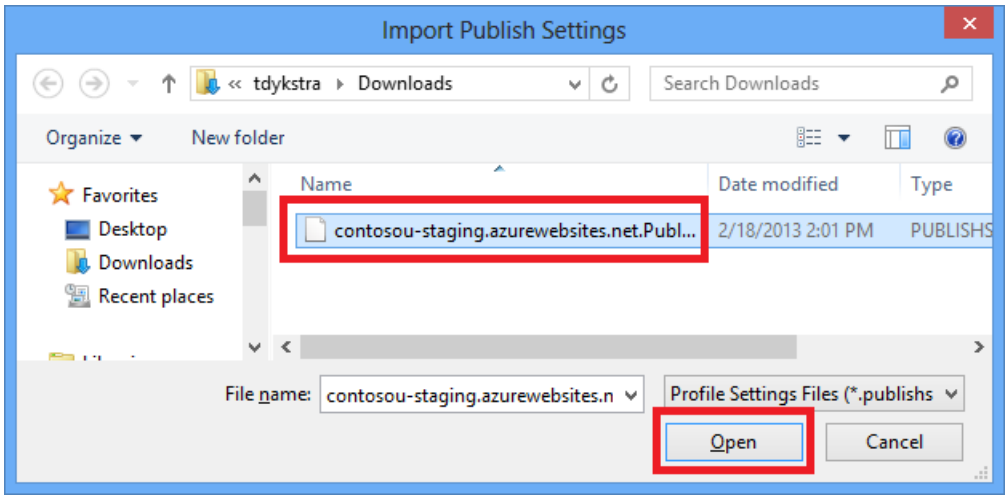
1. In Visual Studio, right-click the ContosoUniversity project in **Solution Explorer** and select **Publish** from the context menu.

   The **Publish Web** wizard opens.

2. Click the **Profile** tab.

3. Click **Import**.

4. Navigate to the *.publishsettings* file you downloaded earlier, and then click **Open**.

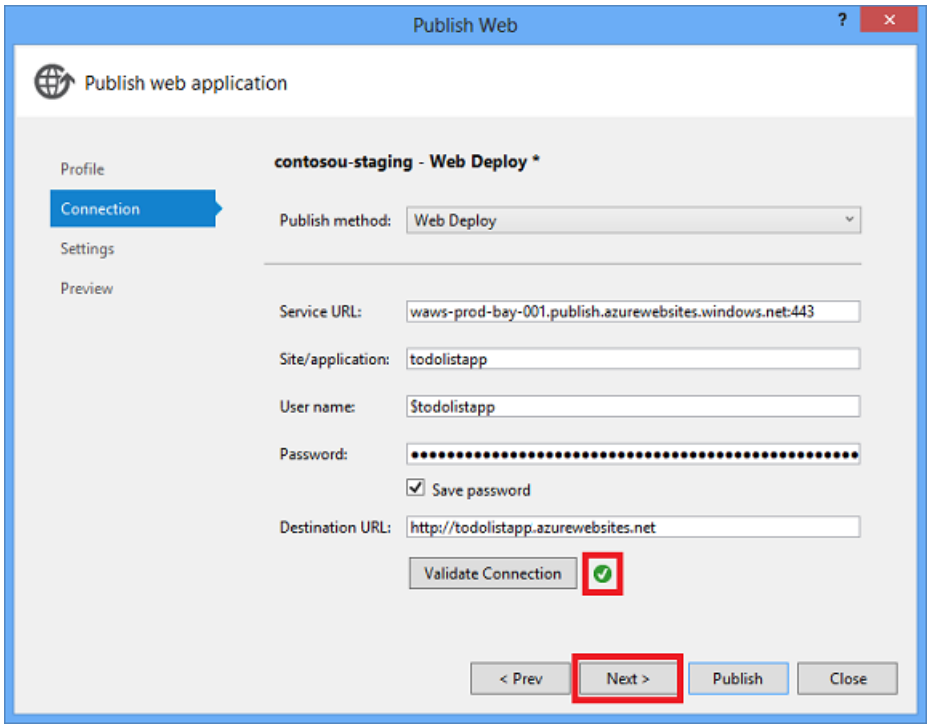


5. In the **Connection** tab, click **Validate Connection** to make sure that the settings are correct.

   When the connection has been validated, a green check mark is shown next to the **Validate Connection** button.

   For some hosting providers, when you click **Validate Connection**, you might see a **Certificate Error** dialog box. If you do, verify that the server name is what you expect. If the server name is correct, select **Save this certificate for future sessions of Visual Studio** and click **Accept**. (This error means that the hosting provider has chosen to avoid the expense of purchasing an SSL certificate for the URL that you are deploying to. If you prefer to establish a secure connection by using a valid certificate, contact your hosting provider.)

6. Click **Next**.



7. In the **Settings** tab, expand **File Publish Options**, and then select **Exclude files from the App_Data folder**.

   For information about the other options under **File Publish Options**, see the deploying to IIS tutorial. The screen shot that shows the result of this step and the following database configuration steps is at the end of the database configuration steps.

8. Under **DefaultConnection** in the **Databases** section, configure database deployment for the membership database.

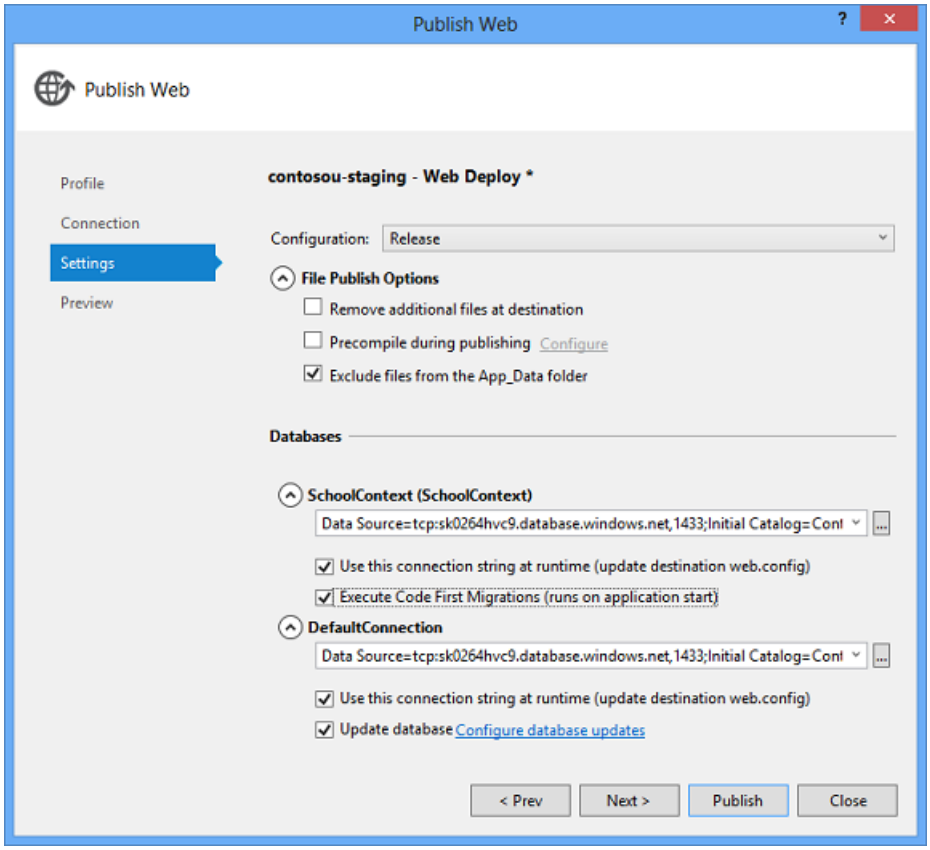
9. a. Select **Update database**.

   The **Remote connection string** box directly below **DefaultConnection** is filled in with the connection string from the .publishsettings file.The connection string includes SQL Server credentials, which are stored in plain text in the *.pubxml* file. If you prefer not to store them permanently there, you can remove them from the publish profile after the database is deployed and store them in Azure instead. For more information, see How to keep your ASP.NET database connection strings secure when deploying to Azure from Source on Scott Hanselman's blog. 2. Click **Configure database updates**. 3. In the **Configure Database Updates** dialog box, click **Add SQL Script**. 4. In the **Add SQL Script** box, navigate to the *aspnet-data-prod.sql* script that you saved earlier in the solution folder, and then click **Open**. 5. Close the **Configure Database Updates** dialog box.

10. Under **SchoolContext** in the **Databases** section, select **Execute Code First Migrations (runs on application start)**.

    Visual Studio displays **Execute Code First Migrations** instead of **Update Database** for `DbContext` classes. If you want to use the dbDacFx provider instead of Migrations to deploy a database that you access by using a `DbContext` class, see How do I deploy a Code First database without Migrations? in the Web Deployment FAQ for Visual Studio and ASP.NET on MSDN.

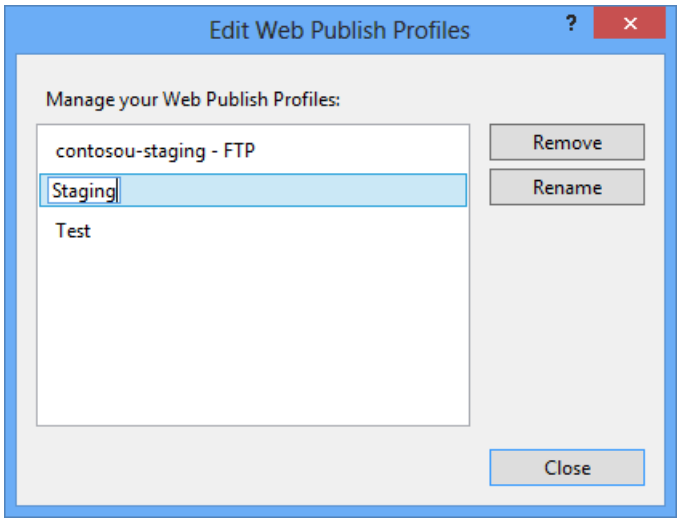
    The **Settings** tab now looks like the following example:



11. Perform the following steps to save the profile and rename it to *Staging*.

    a. Click the **Profile** tab, and then click **Manage Profiles**.

    b. The import created two new profiles, one for FTP and one for Web Deploy. You configured the Web Deploy profile: rename this profile to *Staging*.



c. Close the **Edit Web Publish Profiles** dialog box.

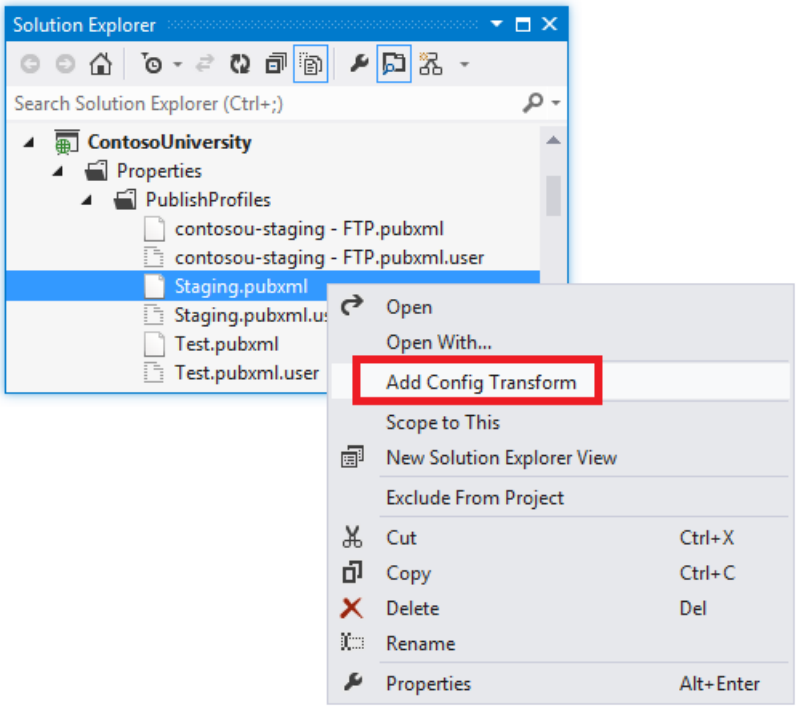
d. Close the **Publish Web** wizard.

Configure a publish profile transform for the environment indicator

> **NOTE**
>
> This section shows how to set up a Web.config transform for the environment indicator. Because the indicator is in the `<appSettings>` element, you have another alternative for specifying the transformation when you're deploying to Azure App Service. For more information, see Specifying Web.config settings in Azure.

1. In **Solution Explorer**, expand **Properties**, and then expand **PublishProfiles**.

2. Right-click *Staging.pubxml*, and then click **Add Config Transform**.



Visual Studio creates the *Web.Staging.config* transform file and opens it.

3. In the *Web.Staging.config* transform file, insert the following code immediately after the opening `configuration` tag.

```
<appSettings>
    <add key="Environment" value="Prod" xdt:Transform="SetAttributes" xdt:Locator="Match(key)"/>
</appSettings>
```

When you use the Staging publish profile, this transform sets the environment indicator to "Prod". In the deployed web app, you won't see any suffix such as "(Dev)" or "(Test)" after the "Contoso University" H1 heading.

4. Right-click the *Web.Staging.config* file and click **Preview Transform** to make sure that the transform you coded produces the expected changes.

   The **Web.config Preview** window shows the result of applying both the *Web.Release.config* transforms and the *Web.Staging.config* transforms.

Prevent public use of the test app

An important consideration for the staging app is that it will be live on the Internet, but you don't want the public to use it. To minimize the likelihood that people will find and use it, you can use one or more of the following methods:

- Set firewall rules that allow access to the staging app only from IP addresses that you use to test staging.
- Use an obfuscated URL that would be impossible to guess.
- Create a *robots.txt* file to ensure that search engines will not crawl the test app and report links to it in search results.

The first of these methods is the most effective but is not covered in this tutorial because it would require that you deploy to an Azure Cloud Service instead of Azure App Service. For more information about Cloud Services and IP restrictions in Azure, see Compute Hosting Options Provided by Azure and Block Specific IP Addresses from Accessing a Web Role. If you are deploying to a third-party hosting provider, contact the provider to find out how to implement IP restrictions.

For this tutorial, you'll create a *robots.txt* file.

1. In **Solution Explorer**, right-click the ContosoUniversity project and click **Add New Item**.

2. Create a new **Text File** named *robots.txt*, and put the following text in it:

   ```
   User-agent: *
   Disallow: /
   ```

   The `User-agent` line tells search engines that the rules in the file apply to all search engine web crawlers (robots), and the `Disallow` line specifies that no pages on the site should be crawled.

   You do want search engines to catalog your production app, so you need to exclude this file from production deployment. To do that, you'll configure a setting in the Production publish profile when you create it.

Deploy to staging

1. Open the **Publish Web** wizard by right-clicking the Contoso University project and clicking **Publish**.

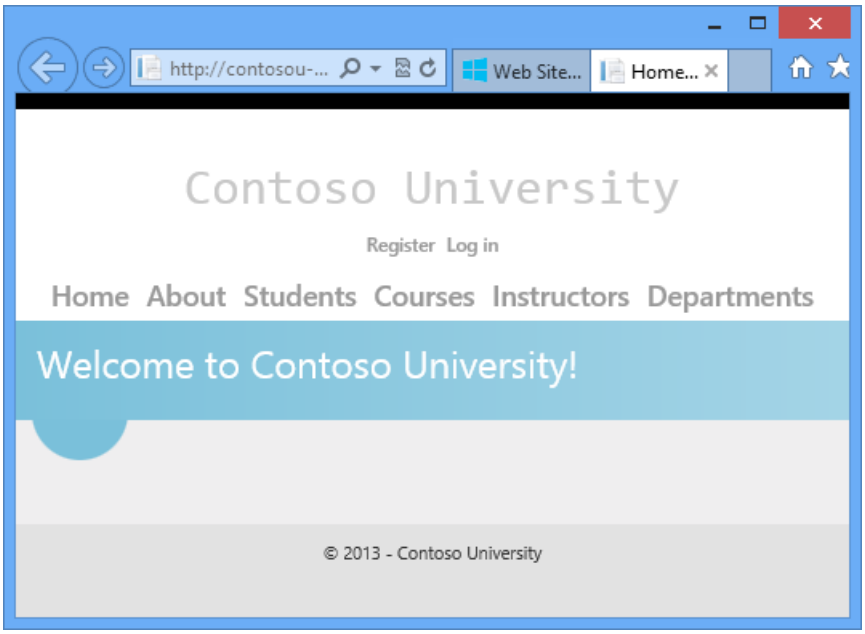
2. Make sure that the **Staging** profile is selected.

3. Click **Publish**.

   The **Output** window shows what deployment actions were taken and reports successful completion of the deployment. The default browser automatically opens to the URL of the deployed web app.

Test in the staging environment

Notice that the environment indicator is absent (there is no "(Test)" or "(Dev)" after the H1 heading, which shows that the *Web.config* transformation for the environment indicator was successful.



Run the **Students** page to verify that the deployed database has no students.

Run the **Instructors** page to verify that Code First seeded the database with instructor data:

Select **Add Students** from the **Students** menu, add a student, and then view the new student in the **Students** page to verify that you can successfully write to the database.

From the **Courses** page, click **Update Credits**. The **Update Credits** page requires administrator permissions, so the **Log In** page is displayed. Enter the administrator account credentials that you created earlier ("admin" and "prodpwd"). The **Update Credits** page is displayed, which verifies that the administrator account that you created in the previous tutorial was correctly deployed to the test environment.

Request an invalid URL to cause an error that ELMAH will track, and then request the ELMAH error report. If you are deploying to a third-party hosting provider, you will probably find that the report is empty for the same reason that it was empty in the previous tutorial. You will have to use the hosting provider's account management tools to configure folder permissions to enable ELMAH to write to the log folder.

The application that you created is now running in the cloud in a web app that is just like what you will use for production. Since everything is working correctly, the next step is to deploy to production.

Deploy to production

The process for creating a production web app and deploying to production is the same as for staging, except that you need to exclude the *robots.txt* from deployment. To do that you'll edit the publish profile file.

Create the production environment and the production publish profile

1. Create the production web app and database in Azure, following the same procedure that you used for staging.

   When you create the database, you can choose to put it on the same server you created earlier, or create a new server.

2. Download the *.publishsettings* file.

3. Create the publish profile by importing the production *.publishsettings* file, following the same procedure

that you used for staging.

Don't forget to configure the data deployment script under **DefaultConnection** in the **Databases** section of the **Settings** tab.

4. Rename the publish profile to *Production*.

5. Configure a publish profile transform for the environment indicator, following the same procedure that you used for staging..

Edit the .pubxml file to exclude robots.txt

Publish profile files are named <profilename>*.pubxml* and are located in the *PublishProfiles* folder. The *PublishProfiles* folder is under the *Properties* folder in a C# web application project, under the *My Project* folder in a VB web application project, or under the *App_Data* folder in a web app project. Each *.pubxml* file contains settings that apply to one publish profile. The values you enter in the Publish Web wizard are stored in these files, and you can edit them to create or change settings that aren't made available in the Visual Studio UI.

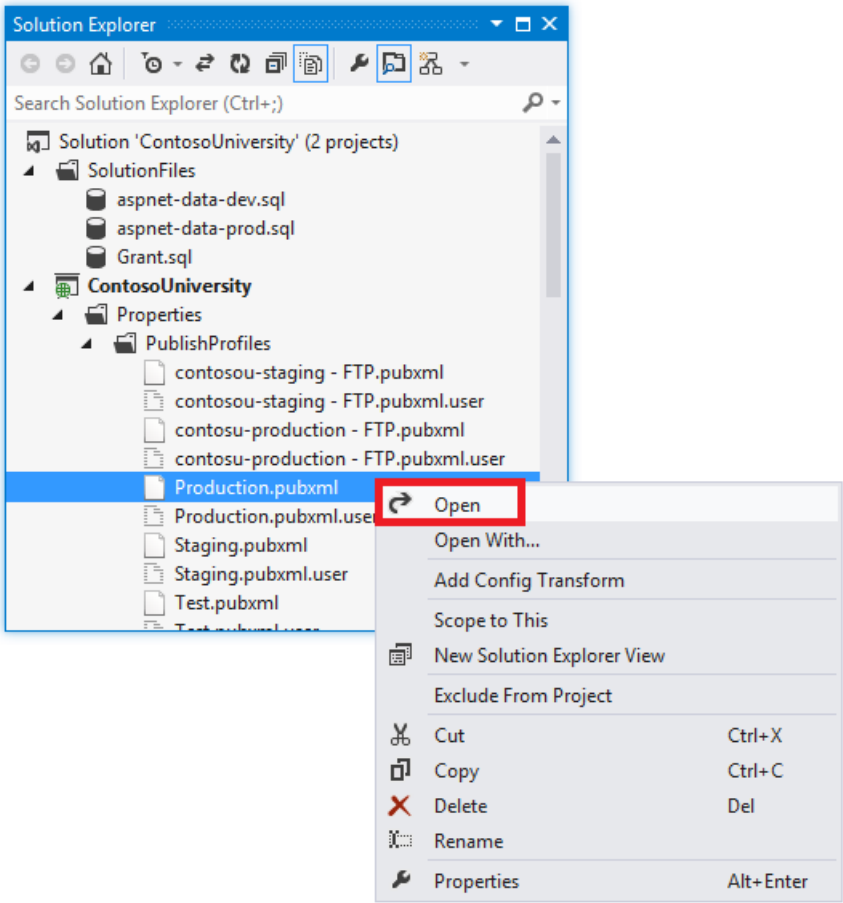
By default, *.pubxml* files are included in the project when you create a publish profile, but you can exclude them from the project and Visual Studio will still use them. Visual Studio looks in the *PublishProfiles* folder for *.pubxml* files, regardless of whether they are included in the project.

For each *.pubxml* file there is a *.pubxml.user* file. The *.pubxml.user* file contains the encrypted password if you selected the **Save password** option, and by default it is excluded from the project.

A *.pubxml* file contains the settings that pertain to a specific publish profile. If you want to configure settings that apply to all profiles, you can create a *.wpp.targets* file. The build process imports these files into the *.csproj* or *.vbproj* project file, so most settings that you can configure in the project file can be configured in these files. For more information about *.pubxml* files and *.wpp.targets* files, see How to: Edit Deployment Settings in Publish Profile (.pubxml) Files and the .wpp.targets File in Visual Studio Web Projects.

1. In **Solution Explorer**, expand **Properties** and expand **PublishProfiles**.

2. Right-click *Production.pubxml* and click **Open**.



3. Right-click *Production.pubxml* and click **Open**.

4. Add the following lines immediately before the closing `PropertyGroup` element:

```
<ExcludeFilesFromDeployment>
    robots.txt
</ExcludeFilesFromDeployment>
```

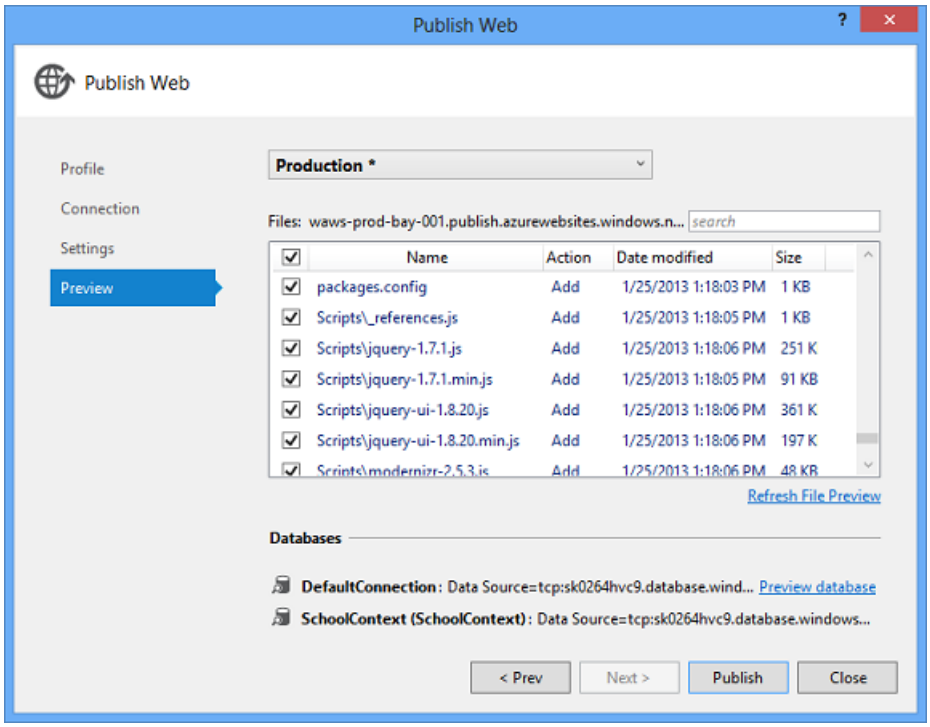
The .pubxml file now looks like the following example:

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>MSDeploy</WebPublishMethod>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <SiteUrlToLaunchAfterPublish>http://contosou-
staging.azurewebsites.net</SiteUrlToLaunchAfterPublish>
    <ExcludeApp_Data>True</ExcludeApp_Data>
    <MSDeployServiceURL>waws-prod-bay-001.publish.azurewebsites.windows.net:443</MSDeployServiceURL>
    <DeployIisAppPath>contosou-staging</DeployIisAppPath>
    <RemoteSitePhysicalPath />
    <SkipExtraFilesOnServer>True</SkipExtraFilesOnServer>
    <MSDeployPublishMethod>WMSVC</MSDeployPublishMethod>
    <UserName>[username]</UserName>
    <_SavePWD>True</_SavePWD>
    <PublishDatabaseSettings>
      <!-- database settings removed -->
    </PublishDatabaseSettings>
    <ExcludeFilesFromDeployment>
        robots.txt
    </ExcludeFilesFromDeployment>
  </PropertyGroup>
  <ItemGroup>
    <MSDeployParameterValue Include="$(DeployParameterPrefix)DefaultConnection-Web.config Connection
String">
      <ParameterValue>Data Source=tcp:sk0264hvc9.database.windows.net,1433;Initial
Catalog=ContosoUniversity-staging;User Id=CU-staging-admin@sk0264hvc9;Password=</ParameterValue>
    </MSDeployParameterValue>
    <MSDeployParameterValue Include="$(DeployParameterPrefix)SchoolContext-Web.config Connection
String">
      <ParameterValue>Data Source=tcp:sk0264hvc9.database.windows.net,1433;Initial
Catalog=ContosoUniversity-staging;User Id=CU-staging-admin@sk0264hvc9;Password=</ParameterValue>
    </MSDeployParameterValue>
  </ItemGroup>
</Project>
```

For more information about how to exclude files and folders, see Can I exclude specific files or folders from deployment? in the **Web Deployment FAQ for Visual Studio and ASP.NET** on MSDN.

Deploy to production

1. Open the **Publish Web** wizard make sure that the **Production** publish profile is selected, and then click **Start Preview** on the **Preview** tab to verify that the *robots.txt* file will not be copied to the production app.



Review the list of files that will be copied. You'll see that all of the *.cs* files, including *.aspx.cs*, *.aspx.designer.cs*, *Master.cs*, and *Master.designer.cs* files are omitted. All of this code has been compiled into the *ContosoUniversity.dll* and *ContosoUniversity.pdb* files that you'll find in the *bin* folder. Because only the *.dll* is needed to run the application, and you specified earlier that only files needed to run the application should be deployed, no *.cs* files were copied to the destination environment. The *obj* folder and the *ContosoUniversity.csproj* and *.csproj.user* files are omitted for the same reason.

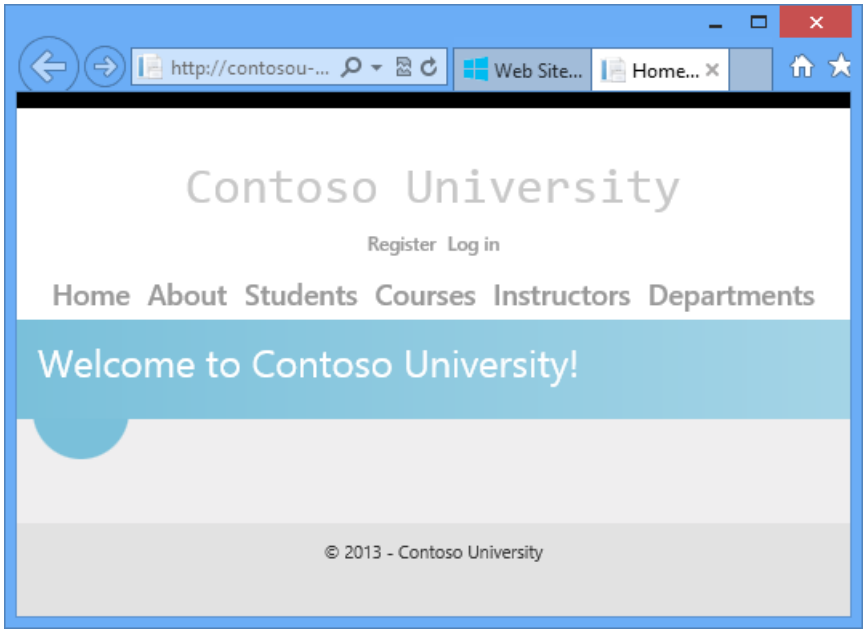
Click **Publish** to deploy to the production environment.

2. Test in production, following the same procedure you used for staging.

   Everything is identical to staging except for the URL and the absence of the *robots.txt* file.

Summary

You have now successfully deployed and tested your web app and it is available publicly over the Internet.



In the next tutorial, you'll update application code and deploy the change to the test, staging, and production environments.

NOTE

While your application is in use in the production environment you should be implementing a recovery plan. That is, you should be periodically backing up your databases from the production app to a secure storage location, and you should be keeping several generations of such backups. When you update the database, you should make a backup copy from immediately before the change. Then, if you make a mistake and don't discover it until after you have deployed it to production, you will still be able to recover the database to the state it was in before it became corrupted. For more information, see Azure SQL Database Backup and Restore.

NOTE

In this tutorial the SQL Server edition that you are deploying to is Azure SQL Database. While the deployment process is similar to other editions of SQL Server, a real production application might require special code for Azure SQL Database in some scenarios. For more information, see Working with Azure SQL Database and Choosing between SQL Server and Azure SQL Database.

PREVIOUS NEXT

ASP.NET Web Deployment using Visual Studio: Deploying a Code Update

2/20/2020 • 9 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This tutorial series shows you how to deploy (publish) an ASP.NET web application to Azure App Service Web Apps or to a third-party hosting provider, by using Visual Studio 2012 or Visual Studio 2010. For information about the series, see the first tutorial in the series.

Overview

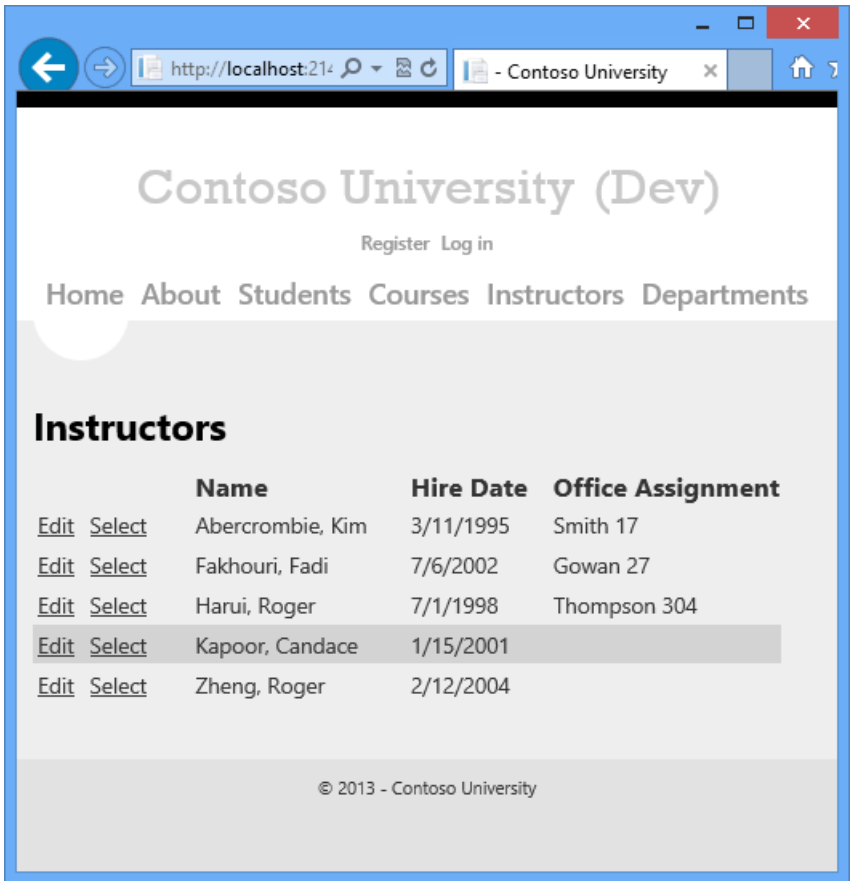
After the initial deployment, your work of maintaining and developing your web site continues, and before long you will want to deploy an update. This tutorial takes you through the process of deploying an update to your application code. The update that you implement and deploy in this tutorial does not involve a database change; you'll see what's different about deploying a database change in the next tutorial.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

Make a code change

As a simple example of an update to your application, you'll add to the **Instructors** page a list of courses taught by the selected instructor.

If you run the **Instructors** page, you'll notice that there are **Select** links in the grid, but they don't do anything other than make the row background turn gray.

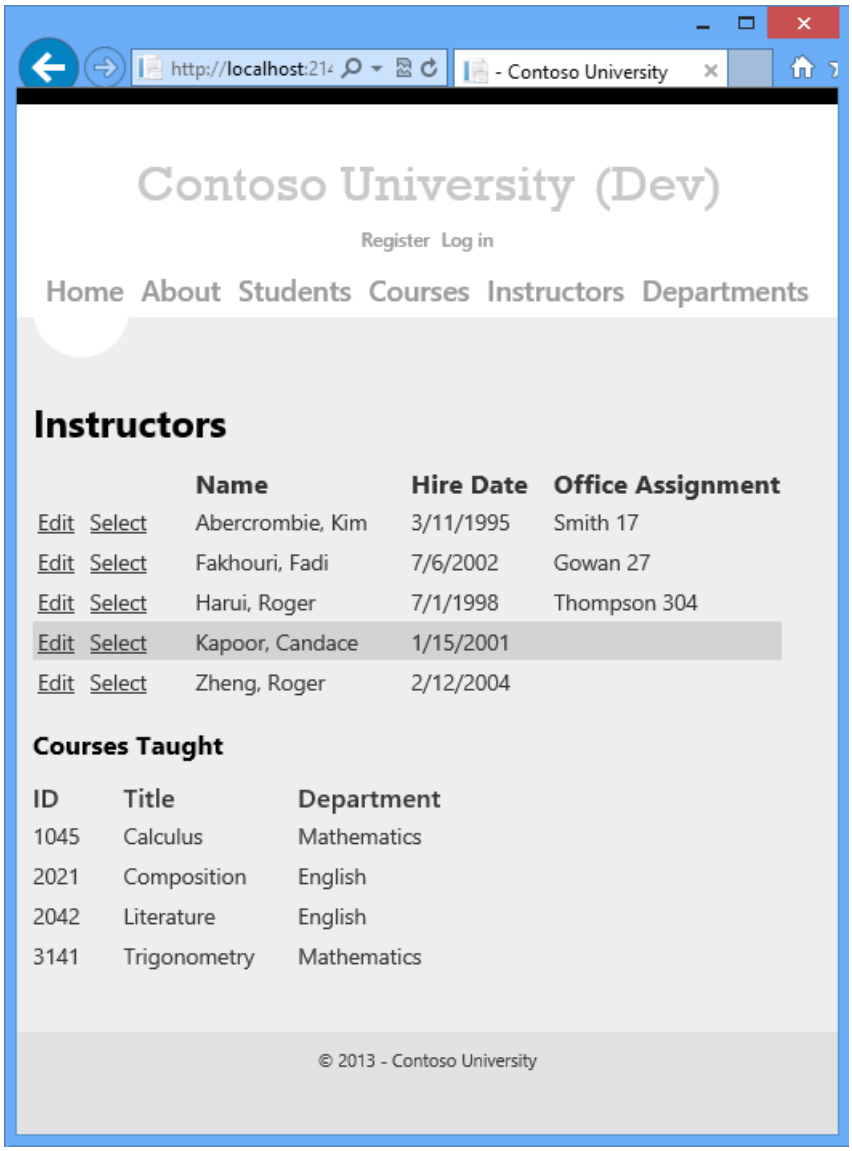


Now you'll add code that runs when the **Select** link is clicked and displays a list of courses taught by the selected instructor .

1. In *Instructors.aspx*, add the following markup immediately after the **ErrorMessageLabel** `Label` control:

```
<h3>Courses Taught</h3>
<asp:ObjectDataSource ID="CoursesObjectDataSource" runat="server"
TypeName="ContosoUniversity.BLL.SchoolBL"
    DataObjectTypeName="ContosoUniversity.DAL.Course" SelectMethod="GetCoursesByInstructor">
    <SelectParameters>
        <asp:ControlParameter ControlID="InstructorsGridView" Name="PersonID"
PropertyName="SelectedDataKey.Value"
            Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
<asp:GridView ID="CoursesGridView" runat="server" DataSourceID="CoursesObjectDataSource"
    AllowSorting="True" AutoGenerateColumns="False" SelectedRowStyle-BackColor="LightGray"
    DataKeyNames="CourseID">
    <EmptyDataTemplate>
        <p>No courses found.</p>
    </EmptyDataTemplate>
    <Columns>
        <asp:BoundField DataField="CourseID" HeaderText="ID" ReadOnly="True" SortExpression="CourseID"
/>
        <asp:BoundField DataField="Title" HeaderText="Title" SortExpression="Title" />
        <asp:TemplateField HeaderText="Department" SortExpression="DepartmentID">
            <ItemTemplate>
                <asp:Label ID="GridViewDepartmentLabel" runat="server" Text='<%#
Eval("Department.Name") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
```

2. Run the page and select an instructor. You see a list of courses taught by that instructor.



3. Close the browser.

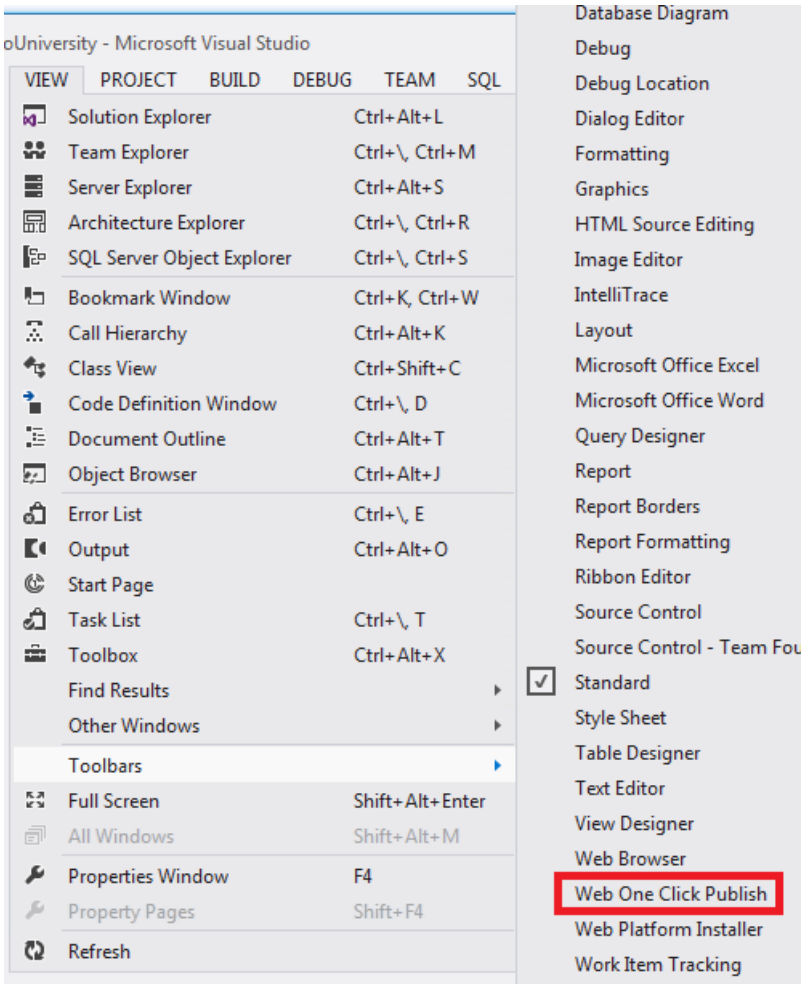
Deploy the code update to the test environment

Before you can use your publish profiles to deploy to test, staging, and production, you need to change database publishing options. You no longer need to run the grant and data deployment scripts for the membership database.

1. Open the **Publish Web** wizard by right-clicking the ContosoUniversity project and clicking **Publish**.
2. Click the **Test** profile in the **Profile** drop-down list.
3. Click the **Settings** tab.
4. Under **DefaultConnection** in the **Databases** section, clear the **Update database** check box.
5. Click the **Profile** tab, and then click the **Staging** profile in the **Profile** drop-down list.
6. When you are asked if you want to save the changes made to the **Test** profile, click **Yes**.
7. Make the same change in the Staging profile.
8. Repeat the process to make the same change in the Production profile.
9. Close the **Publish Web** wizard.

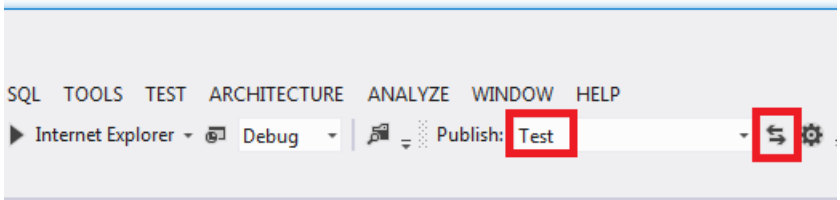
Deploying to the test environment is now a simple matter of running one-click publish again. To make this process quicker, you can use the **Web One Click Publish** toolbar.

1. In the **View** menu, choose **Toolbars** and then select **Web One Click Publish**.



2. In **Solution Explorer**, select the ContosoUniversity project.

3. the **Web One Click Publish** toolbar, choose the **Test** publish profile and then click **Publish Web** (the icon with arrows pointing left and right).



4. Visual Studio deploys the updated application, and the browser automatically opens to the home page.

5. Run the Instructors page and select an instructor to verify that the update was successfully deployed.

You would normally also do regression testing (that is, test the rest of the site to make sure that the new change didn't break any existing functionality). But for this tutorial you'll skip that step and proceed to deploy the update to staging and production.

When you redeploy, Web Deploy automatically determines which files have changed and only copies changed files to the server. By default, Web Deploy uses last-changed dates on files to determine which ones have changed. Some source control systems change file dates even when you don't change the file contents. In that case, you might want to configure Web Deploy to use file checksums to determine which files have changed. For more information, see Why do all of my files get redeployed although I didn't change them? in the ASP.NET Deployment FAQ.

Take the application offline during deployment

The change you're deploying now is a simple change to a single page. But sometimes you deploy larger changes,

or you deploy both code and database changes, and the site might behave incorrectly if a user requests a page before deployment is finished. To prevent users from accessing the site while deployment is in progress, you can use an *app_offline.htm* file. When you put a file named *app_offline.htm* in the root folder of your application, IIS automatically displays that file instead of running your application. So to prevent access during deployment, you put *app_offline.htm* in the root folder, run the deployment process, and then remove *app_offline.htm* after successful deployment.

You can configure Web Deploy to automatically put a default *app_offline.htm* file on the server when it starts deploying and remove it when it finishes. To do that all you have to do is add the following XML element to your publish profile (.pubxml) file:

```
<EnableMSDeployAppOffline>true</EnableMSDeployAppOffline>
```

For this tutorial you'll see how to create and use a custom *app_offline.htm* file.

Using *app_offline.htm* in the staging site isn't required, because you don't have users accessing the staging site. But it's a good practice to use staging to test everything the way you plan to deploy in production.

Create app_offline.htm

1. In **Solution Explorer**, right-click the solution and click **Add**, and then click **New Item**.
2. Create an **HTML Page** named *app_offline.htm* (delete the final "l" in the *.html* extension that Visual Studio creates by default).
3. Replace the template markup with the following markup:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Contoso University - Under Construction</title>
</head>
<body>
  <h1>Contoso University</h1>
  <h2>Under Construction</h2>
  <p>The Contoso University site is temporarily unavailable while we upgrade it. Please try again
later.</p>
</body>
</html>
```

4. Save and close the file.

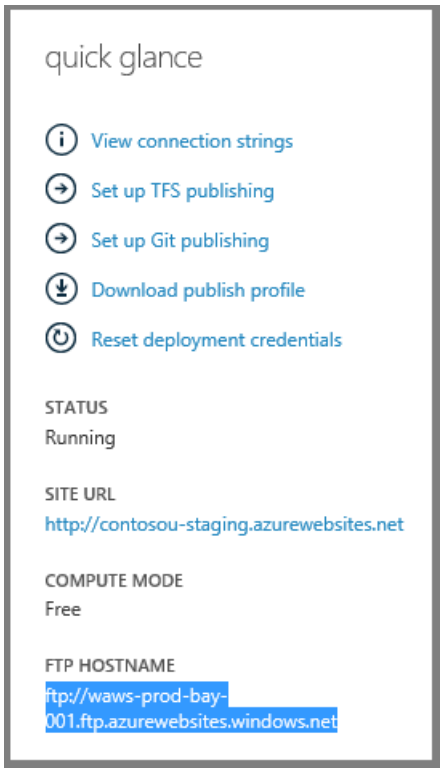
Copy app_offline.htm to the root folder of the web site

You can use any FTP tool to copy files to the web site. FileZilla is a popular FTP tool and is shown in the screen shots.

To use an FTP tool, you need three things: the FTP URL, the user name, and the password.

The URL is shown on the web site's dashboard page in the Azure Management Portal, and the user name and password for FTP can be found in the *.publishsettings* file that you downloaded earlier. The following steps show how to get these values.

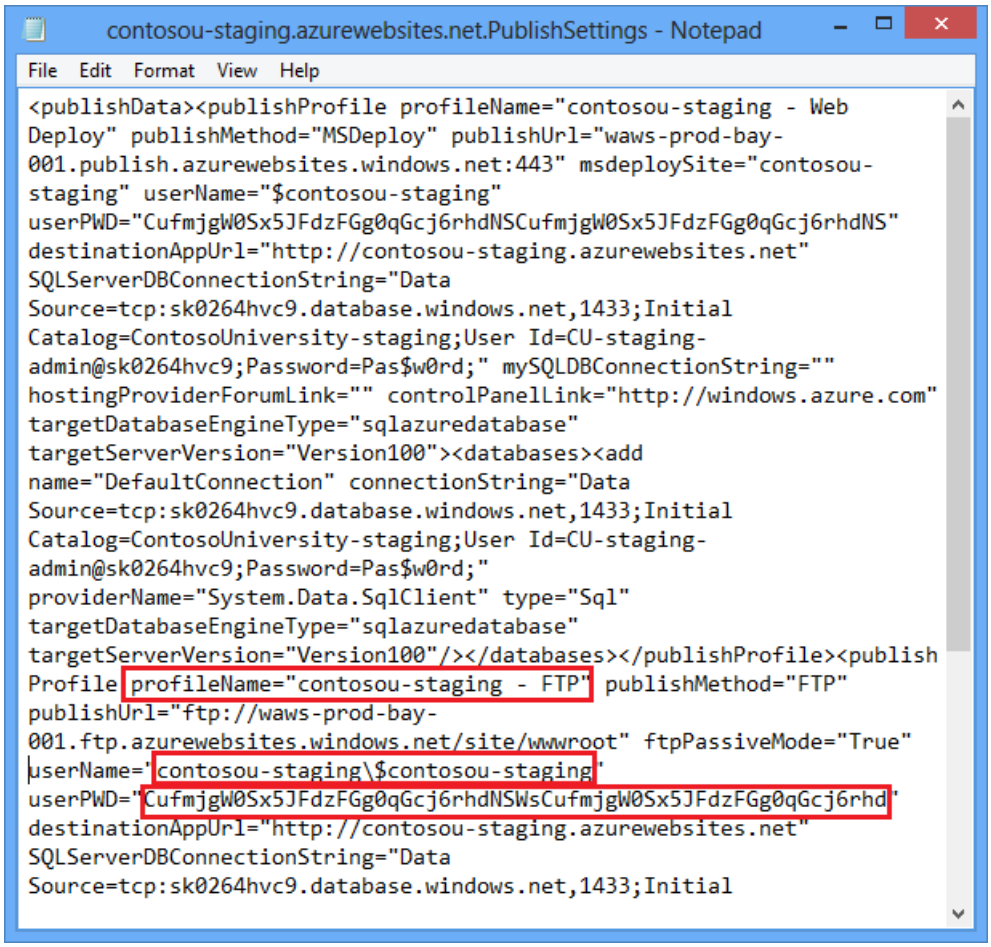
1. In the Azure Management Portal, click **Web Sites** tab and then click the staging web site.
2. On the **Dashboard** page, scroll down to find the FTP host name in the **Quick Glance** section.



3. Open the staging *.publishsettings* file in Notepad or another text editor.

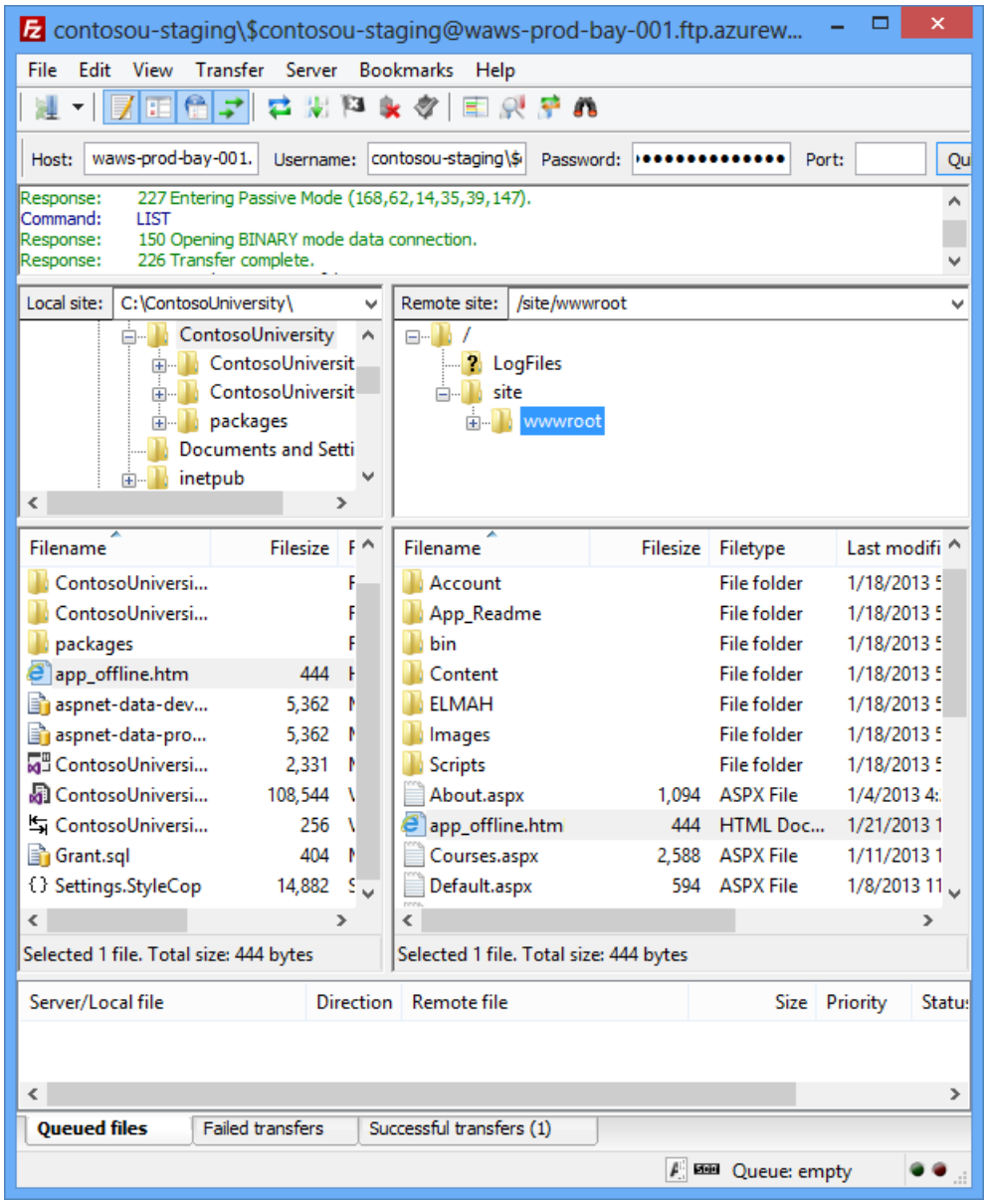
4. Find the `publishProfile` element for the FTP profile.

5. Copy the `userName` and `userPWD` values.

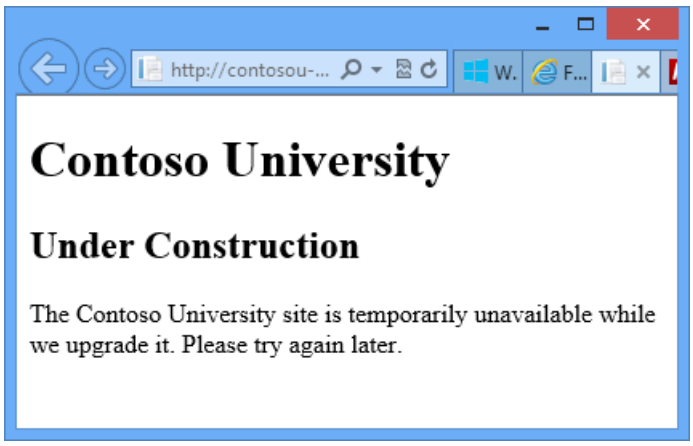


6. Open your FTP tool and log on to the FTP URL.

7. Copy *app_offline.htm* from the solution folder to the */site/wwwroot* folder in the staging site.



8. Browse to your staging site's URL. You see that the *app_offline.htm* page is now displayed instead of your home page.



You are now ready to deploy to staging.

Deploy the code update to staging and production

1. In the **Web One Click Publish** toolbar, choose the **Staging** publish profile and then click **Publish Web**.

   Visual Studio deploys the updated application and opens the browser to the site's home page. The

app_offline.htm file is displayed. Before you can test to verify successful deployment, you must remove the *app_offline.htm* file.

2. Return to your FTP tool, and delete **app_offline.htm** from the staging site.

3. In the browser, open the Instructors page in the staging site, and select an instructor to verify that the update was successfully deployed.

4. Follow the same procedure for production as you did for staging.

Reviewing Changes and Deploying Specific Files

Visual Studio 2012 also gives you the ability to deploy individual files. For a selected file you can view differences between the local version and the deployed version, deploy the file to the destination environment, or copy the file from the destination environment to the local project. In this section of the tutorial you see how to use these features.

Make a change to deploy

1. Open *Content/Site.css*, and find the block for the `body` tag.

2. Change the value for `background-color` from `#fff` to `darkblue`.

```
body {
    background-color: darkblue;
    border-top: solid 10px #000;
    color: #333;
    font-size: .85em;
    font-family: "Segoe UI", Verdana, Helvetica, Sans-Serif;
    margin: 0;
    padding: 0;
}
```

View the change in the Publish Preview window

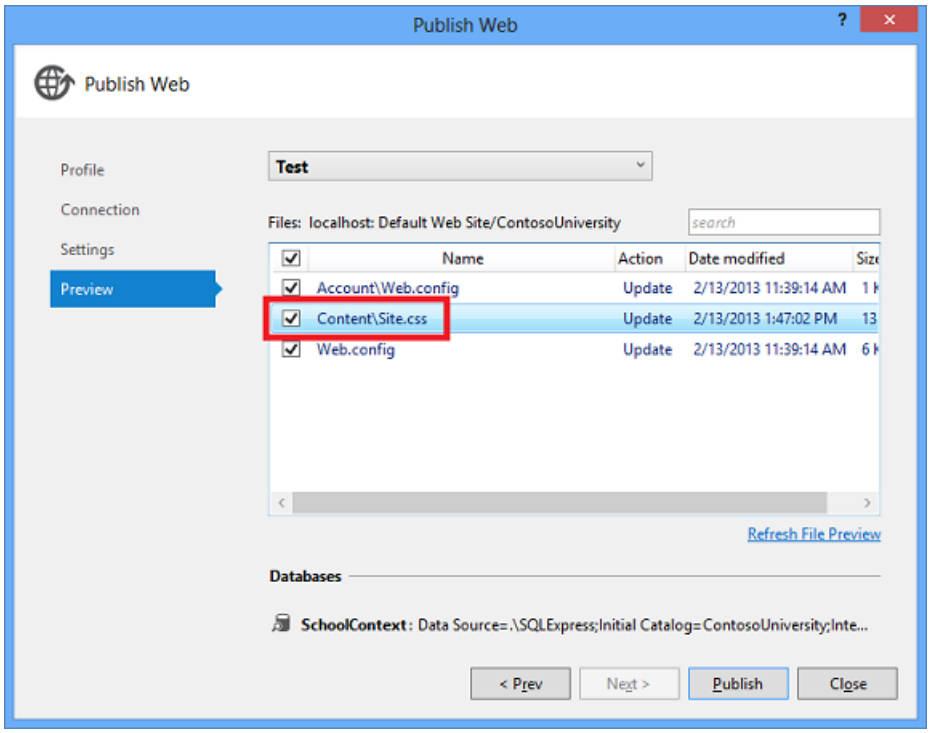
When you use the **Publish Web** wizard to publish the project, you can see what changes are going to be published by double-clicking the file in the **Preview** window.

1. Right-click the ContosoUniversity project and click **Publish**.

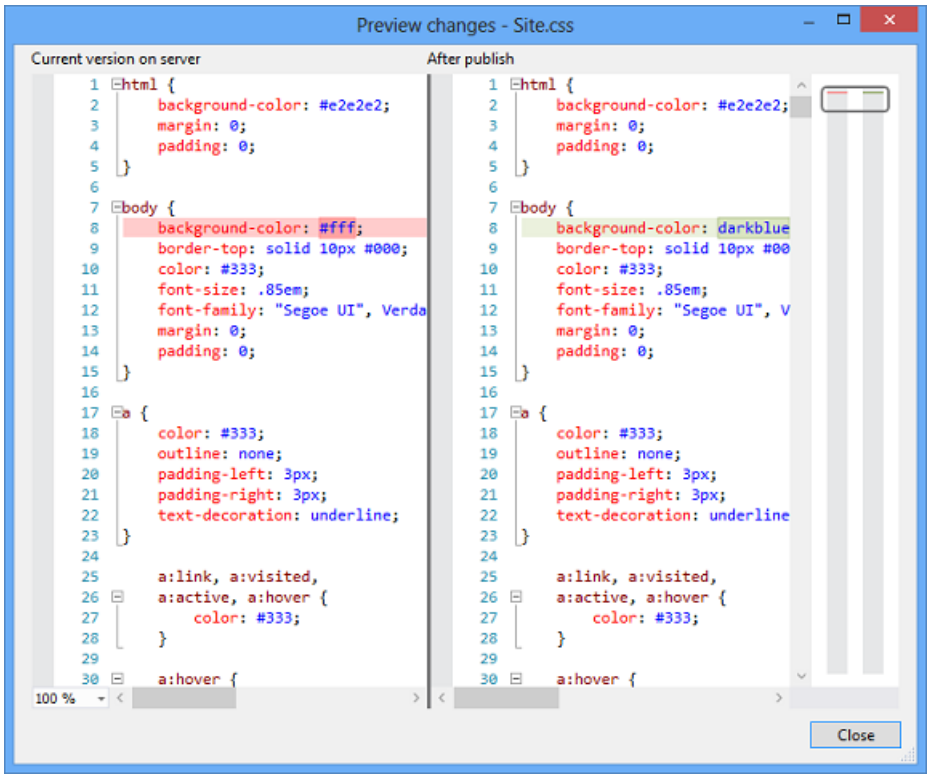
2. From the **Profile** drop-down list, select the **Test** publish profile.

3. Click **Preview**, and then click **Start Preview**.

4. In the **Preview** pane, double-click **Site.css**.



The **Preview changes** dialog shows a preview of the changes that will be deployed.

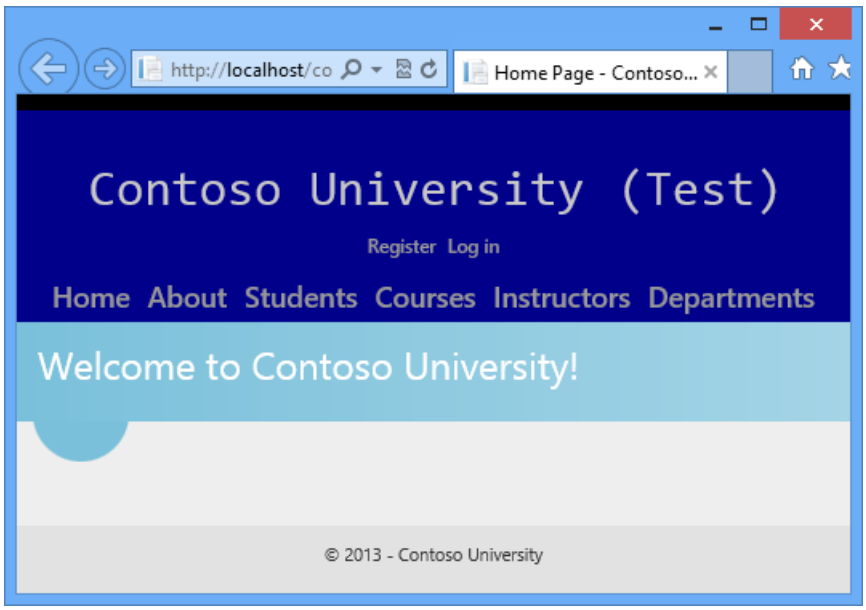


If you double-click the *Web.config* file, the **Preview changes** dialog shows the effect of your build configuration transformations and publish profile transformations. At this point you have not done anything that would cause the *Web.config* file on the server to change, so you expect to see no changes. However, the **Preview changes** window incorrectly shows two changes. It looks like two XML elements will be removed. These elements are added by the publish process when you select **Execute Code First Migrations on application start** for a Code First context class. The comparison is done before the publish process adds those elements, so it looks like they are being removed although they will not be removed. This error will be corrected in a future release.

5. Click **Close**.

6. Click **Publish**.

7. When the browser opens to the home page of the Test site, press CTRL+F5 to cause a hard refresh in order to see the effect of the CSS change.



8. Close the browser.

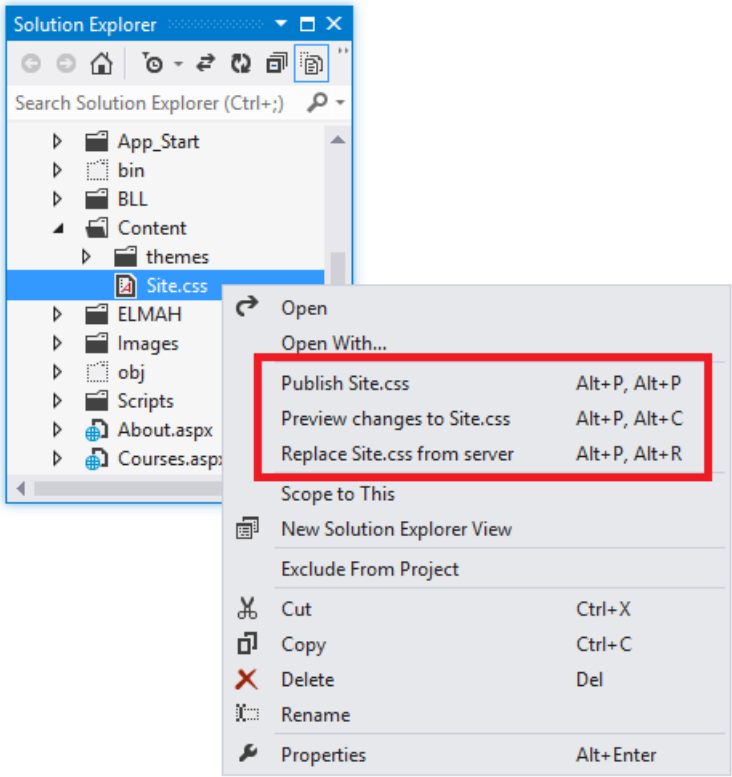
Publish specific files from Solution Explorer

Suppose you don't like the blue background and want to revert to the original color. In this section you'll restore the original settings by publishing a specific file directly from **Solution Explorer**.

1. Open *Content/Site.css* and restore the `background-color` setting to `#fff`.

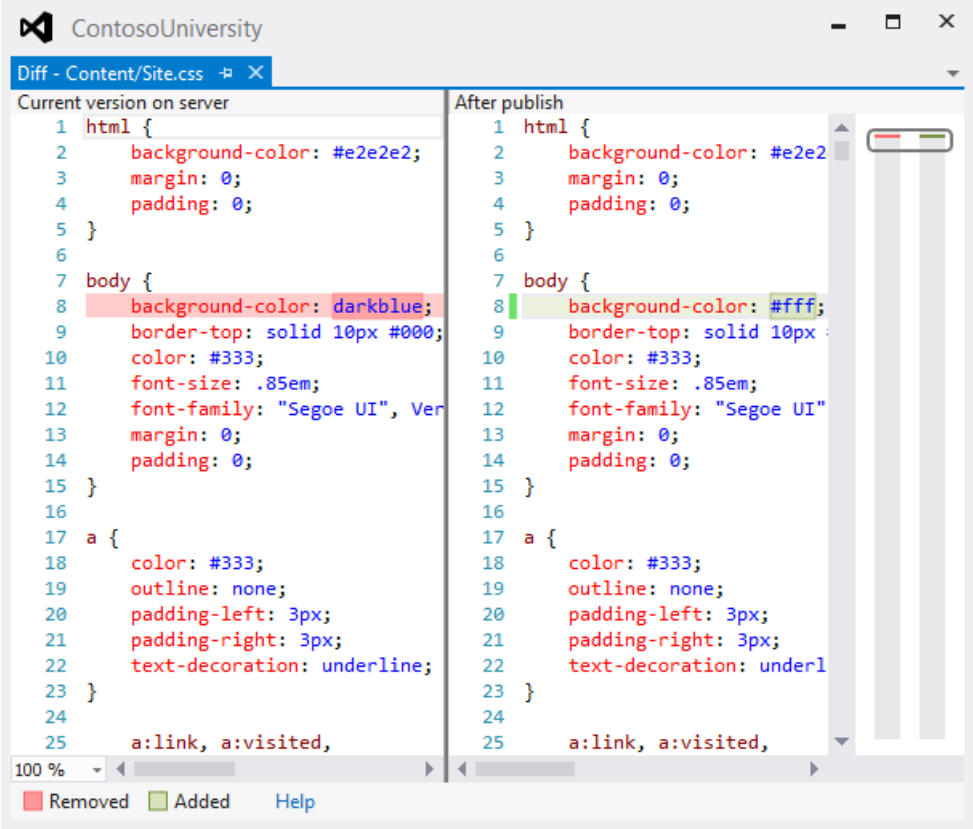
2. In **Solution Explorer**, right-click the *Content/Site.css* file.

   The context menu shows three publish options.



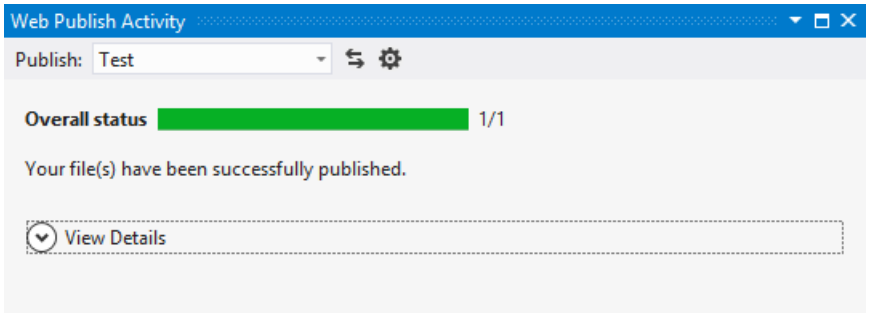
3. Click **Preview changes to Site.css**.

   A window opens to show the differences between the local file and the version of it in the destination environment.

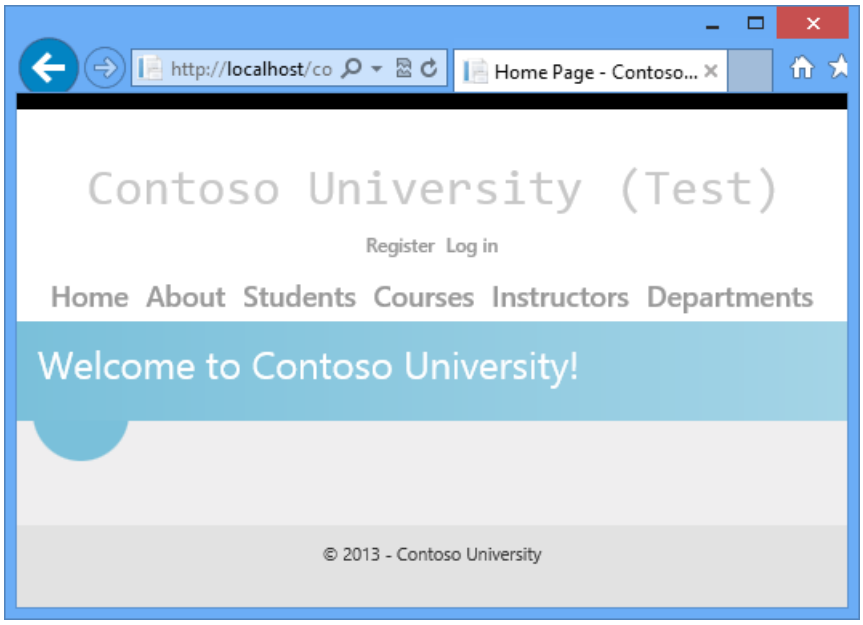


4. In **Solution Explorer**, right-click **Site.css** again and click **Publish Site.css**.

   The **Web Publish Activity** window shows that the file has been published.



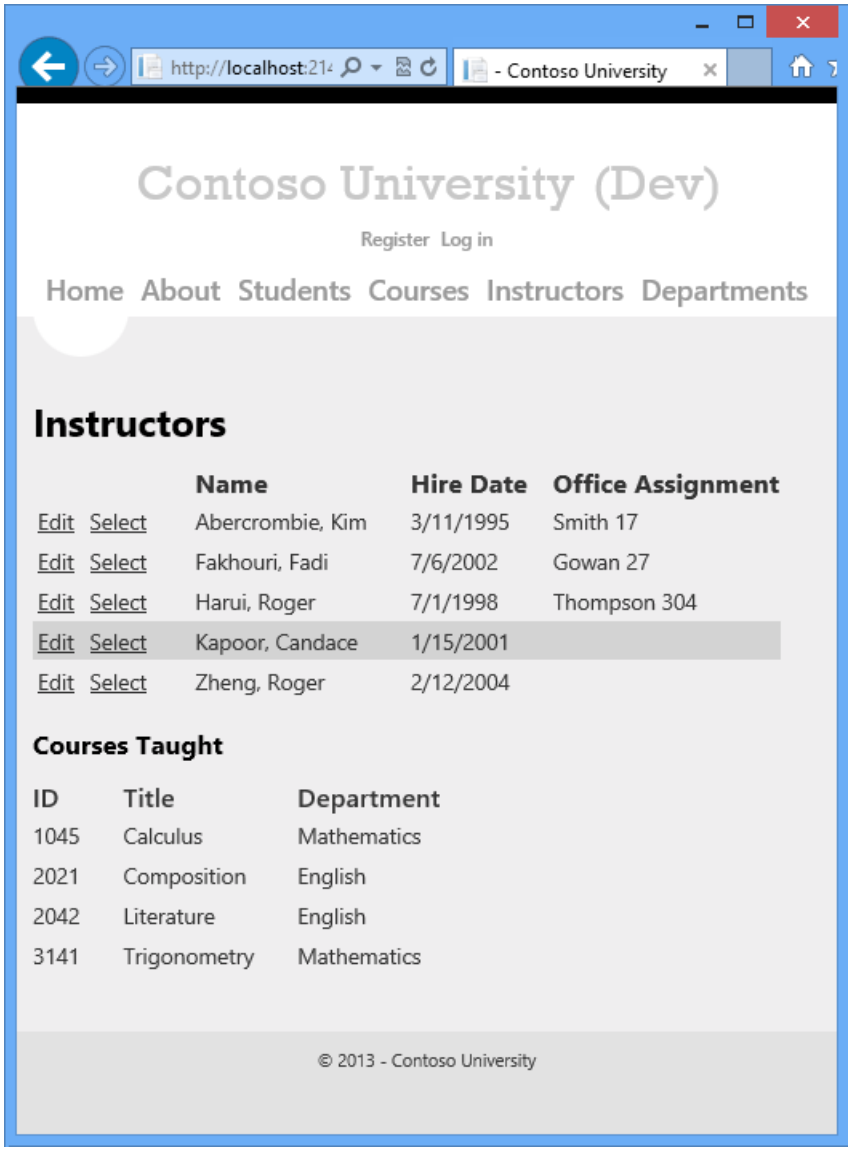
5. Open a browser to the `http://localhost/contosouniversity` URL, and then press CTRL+F5 to cause a hard refresh in order to see the effect of the CSS change.



6. Close the browser.

Summary

You've now seen several ways to deploy an application update that does not involve a database change, and you've seen how to preview the changes to verify that what will be updated is what you expect. The Instructors page now has a **Courses Taught** section.



The next tutorial shows you how to deploy a database change: you'll add a birthdate field to the database and to the Instructors page.

PREVIOUS NEXT

ASP.NET Web Deployment using Visual Studio: Deploying a Database Update

2/20/2020 • 8 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This tutorial series shows you how to deploy (publish) an ASP.NET web application to Azure App Service Web Apps or to a third-party hosting provider, by using Visual Studio 2012 or Visual Studio 2010. For information about the series, see the first tutorial in the series.

Overview

In this tutorial, you make a database change and related code changes, test the changes in Visual Studio, then deploy the update to the test, staging, and production environments.

The tutorial first shows how to update a database that is managed by Code First Migrations, and then later it shows how to update a database by using the dbDacFx provider.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

Deploy a database update by using Code First Migrations

In this section, you add a birth date column to the `Person` base class for the `Student` and `Instructor` entities. Then you update the page that displays instructor data so that it displays the new column. Finally, you deploy the changes to test, staging, and production.

Add a column to a table in the application database

1. In the *ContosoUniversity.DAL* project, open *Person.cs* and add the following property at the end of the `Person` class (there should be two closing curly braces following it):

   ```
   [DisplayFormat(DataFormatString = "{0:d}", ApplyFormatInEditMode = true)]
   [Display(Name = "Birth Date")]
   public DateTime? BirthDate { get; set; }
   ```

   Next, update the `Seed` method so that it provides a value for the new column. Open *Migrations\Configuration.cs* and replace the code block that begins `var instructors = new List<Instructor>` with the following code block which includes birth date information:

```
var instructors = new List<Instructor>
{
    new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie", HireDate =
DateTime.Parse("1995-03-11"), BirthDate = DateTime.Parse("1918-08-12"), OfficeAssignment = new
OfficeAssignment { Location = "Smith 17" } },
    new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",    HireDate =
DateTime.Parse("2002-07-06"), BirthDate = DateTime.Parse("1960-03-15"), OfficeAssignment = new
OfficeAssignment { Location = "Gowan 27" } },
    new Instructor { FirstMidName = "Roger",   LastName = "Harui",       HireDate =
DateTime.Parse("1998-07-01"), BirthDate = DateTime.Parse("1970-01-11"), OfficeAssignment = new
OfficeAssignment { Location = "Thompson 304" } },
    new Instructor { FirstMidName = "Candace", LastName = "Kapoor",      HireDate =
DateTime.Parse("2001-01-15"), BirthDate = DateTime.Parse("1975-04-11") },
    new Instructor { FirstMidName = "Roger",   LastName = "Zheng",       HireDate =
DateTime.Parse("2004-02-12"), BirthDate = DateTime.Parse("1957-10-12") }
};
```

2. Build the solution, and then open the **Package Manager Console** window. Make sure that ContosoUniversity.DAL is still selected as the **Default project**.

3. In the **Package Manager Console** window, select **ContosoUniversity.DAL** as the **Default project**, and then enter the following command:

```
add-migration AddBirthDate
```

When this command finishes, Visual Studio opens the class file that defines the new `DbMigration` class, and in the `Up` method you can see the code that creates the new column. The `Up` method creates the column when you are implementing the change, and the `Down` method deletes the column when you are rolling back the change.

```
public partial class AddBirthDate : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.Person", "BirthDate", c => c.DateTime());
    }

    public override void Down()
    {
        DropColumn("dbo.Person", "BirthDate");
    }
}
```

4. Build the solution, and then enter the following command in the **Package Manager Console** window (make sure the ContosoUniversity.DAL project is still selected):

```
update-database
```

The Entity Framework runs the `Up` method and then runs the `Seed` method.

Display the new column in the Instructors page

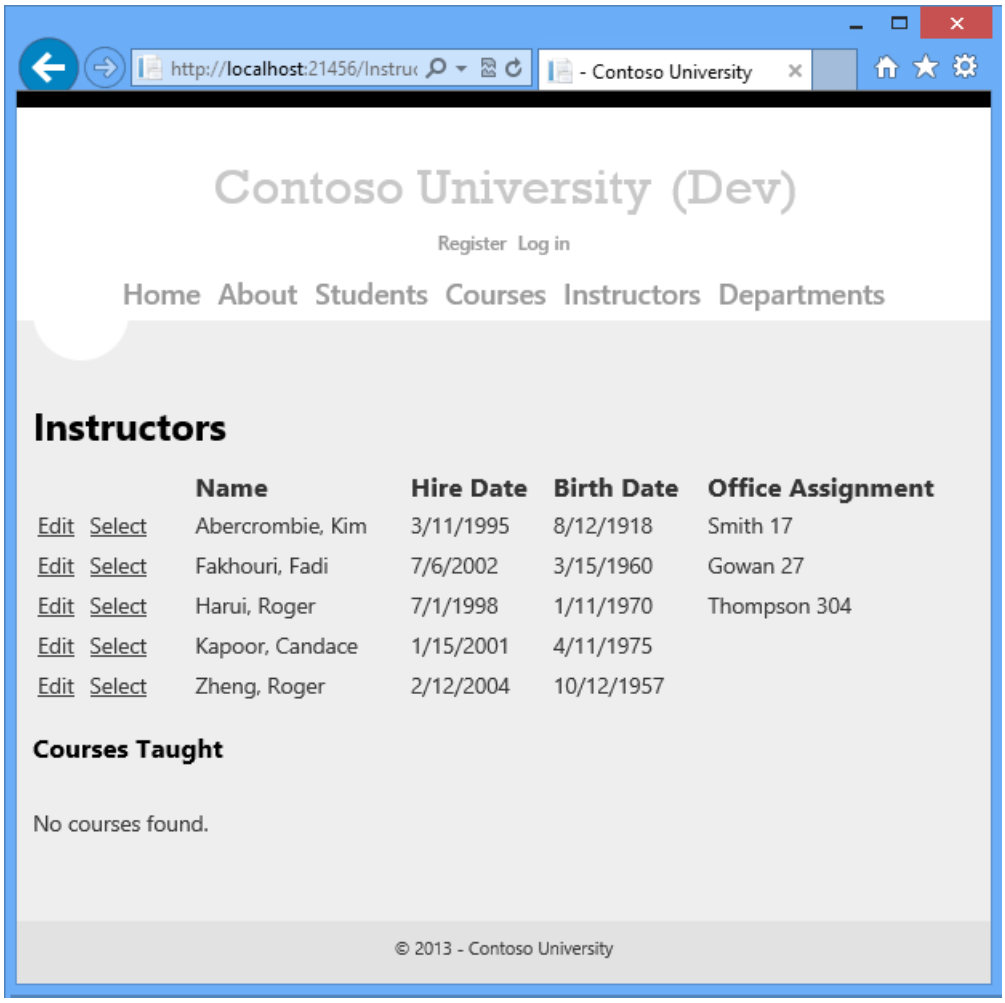
1. In the ContosoUniversity project, open *Instructors.aspx* and add a new template field to display the birth date. Add it between the ones for hire date and office assignment:

```
<asp:TemplateField HeaderText="Hire Date" SortExpression="HireDate">
    <ItemTemplate>
        <asp:Label ID="InstructorHireDateLabel" runat="server" Text='<%# Eval("HireDate", "{0:d}")
%>'></asp:Label>
    </ItemTemplate>
    <EditItemTemplate>
        <asp:TextBox ID="InstructorHireDateTextBox" runat="server" Text='<%# Bind("HireDate", "{0:d}")
%>' Width="7em"></asp:TextBox>
    </EditItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="Birth Date" SortExpression="BirthDate">
    <ItemTemplate>
        <asp:Label ID="InstructorBirthDateLabel" runat="server" Text='<%# Eval("BirthDate", "{0:d}")
%>'></asp:Label>
    </ItemTemplate>
    <EditItemTemplate>
        <asp:TextBox ID="InstructorBirthDateTextBox" runat="server" Text='<%# Bind("BirthDate", "
{0:d}") %>'
            Width="7em"></asp:TextBox>
    </EditItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="Office Assignment" SortExpression="OfficeAssignment.Location">
    <ItemTemplate>
        <asp:Label ID="InstructorOfficeLabel" runat="server" Text='<%#
Eval("OfficeAssignment.Location") %>'></asp:Label>
    </ItemTemplate>
    <EditItemTemplate>
        <asp:TextBox ID="InstructorOfficeTextBox" runat="server"
            Text='<%# Eval("OfficeAssignment.Location") %>' Width="7em"
            OnInit="InstructorOfficeTextBox_Init"></asp:TextBox>
    </EditItemTemplate>
</asp:TemplateField>
```

(If code indentation gets out of sync, you can press CTRL-K and then CTRL-D to automatically reformat the file.)

2. Run the application and click the **Instructors** link.

   When the page loads, you see that it has the new birth date field.



3. Close the browser.

Deploy the database update

1. In **Solution Explorer** select the ContosoUniversity project.
2. In the **Web One Click Publish** toolbar, click the **Test** publish profile, and then click **Publish Web**. (If the toolbar is disabled, select the ContosoUniversity project in **Solution Explorer**.)

   Visual Studio deploys the updated application, and the browser opens to the home page.
3. Run the **Instructors** page to verify that the update was successfully deployed.

   When the application tries to access the database for this page, Code First updates the database schema and runs the `Seed` method. When the page displays, you see the expected **Birth Date** column with dates in it.
4. In the **Web One Click Publish** toolbar, click the **Staging** publish profile, and then click **Publish Web**.
5. Run the **Instructors** page in staging to verify that the update was successfully deployed.
6. In the **Web One Click Publish** toolbar, click the **Production** publish profile, and then click **Publish Web**.
7. Run the **Instructors** page in production to verify that the update was successfully deployed.

   For a real production application update that includes a database change you would also typically take the application offline during deployment by using *app_offline.htm*, as you saw in the previous tutorial.

Deploy a database update by using the dbDacFx provider

In this section, you add a *Comments* column to the *User* table in the membership database and create a page that lets you display and edit comments for each user. Then you deploy the changes to test, staging, and production.

Add a column to a table in the membership database

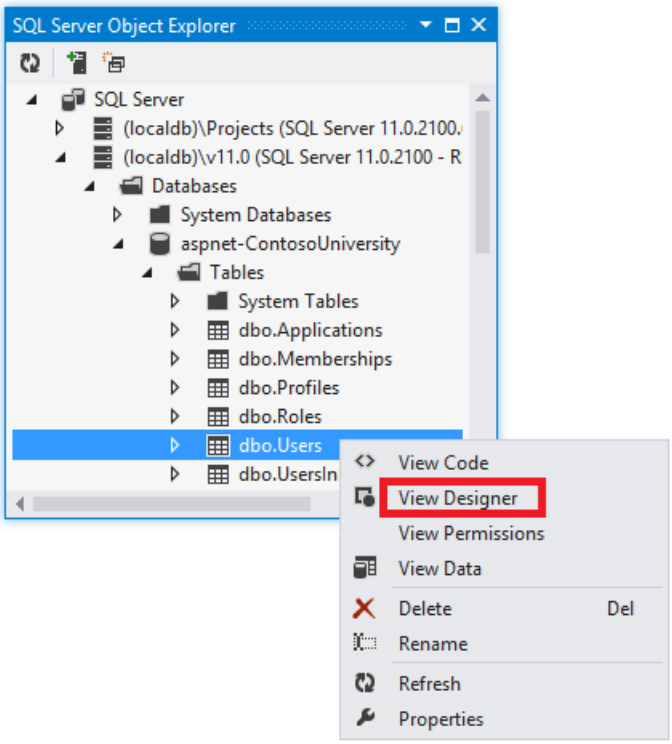
1. In Visual Studio, open **SQL Server Object Explorer**.

2. Expand **(localdb)\v11.0**, expand **Databases**, expand **aspnet-ContosoUniversity** (not **aspnet-ContosoUniversity-Prod**) and then expand **Tables**.

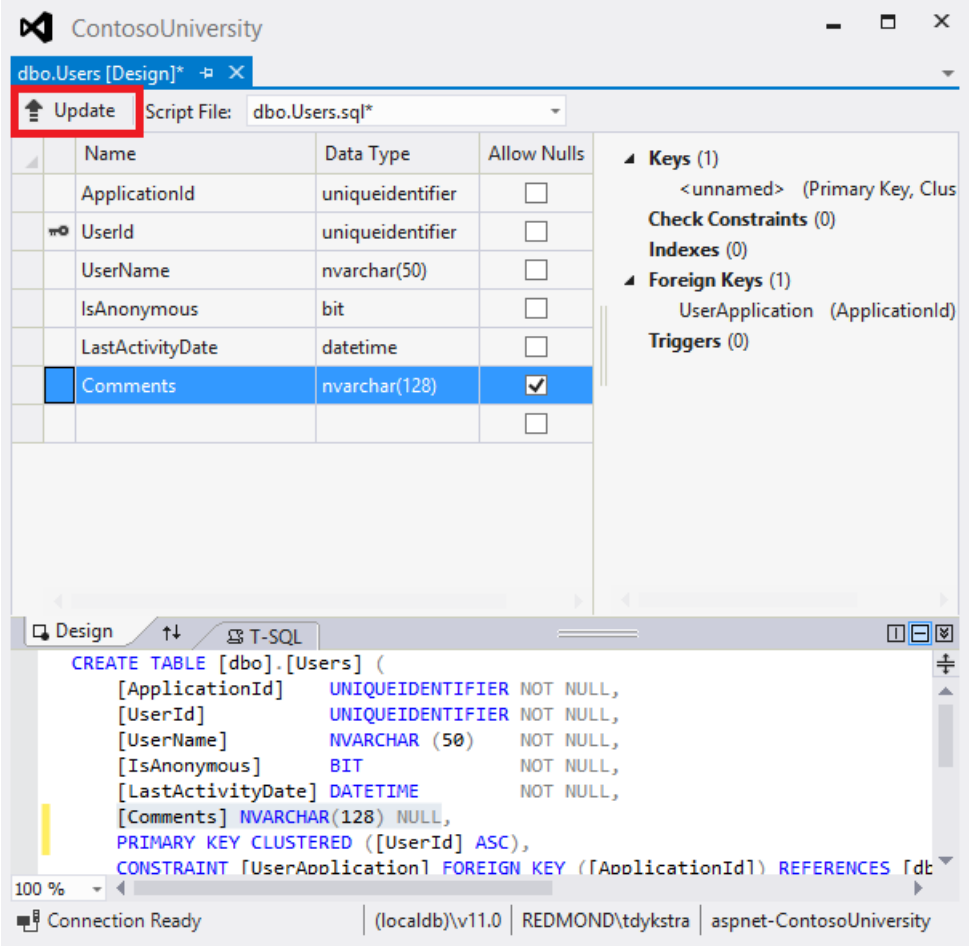
   If you don't see **(localdb)\v11.0** under the **SQL Server** node, right-click the **SQL Server** node and click **Add SQL Server**. In the **Connect to Server** dialog box enter *(localdb)\v11.0* as the **Server name**, and then click **Connect**.

   If you don't see **aspnet-ContosoUniversity**, run the project and log in using the *admin* credentials (password is *devpwd*), and then refresh the **SQL Server Object Explorer** window.

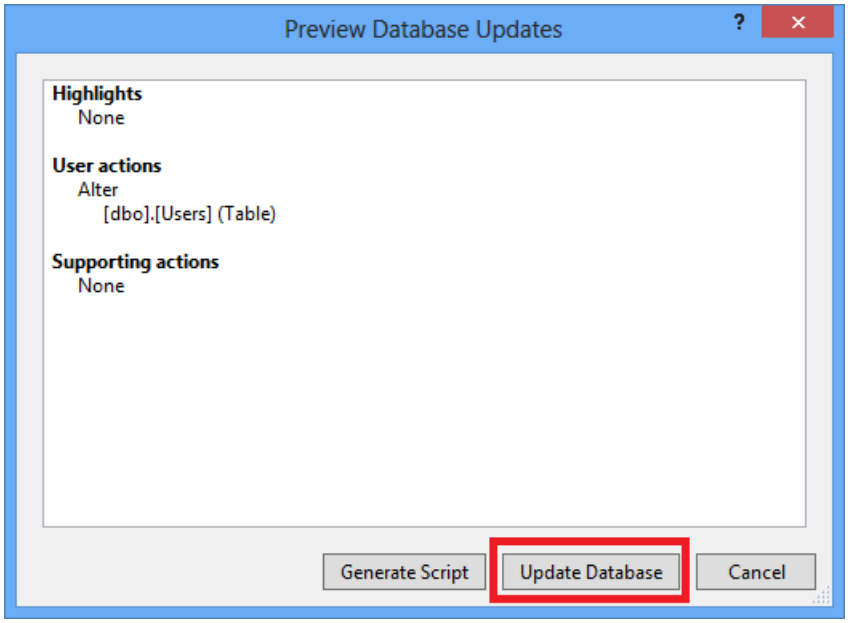
3. Right-click the **Users** table, and then click **View Designer**.



4. In the designer, add a *Comments* column and make it *nvarchar(128)* and nullable, and then click **Update**.



5. In the **Preview Database Updates** box, click **Update Database**.



Create a page to display and edit the new column

1. In **Solution Explorer**, right-click the **Account** folder in the ContosoUniversity project, click **Add**, and then click **New Item**.

2. Create a new **Web Form Using Master Page** and name it *UserInfo.aspx*. Accept the default *Site.Master* file as the master page.

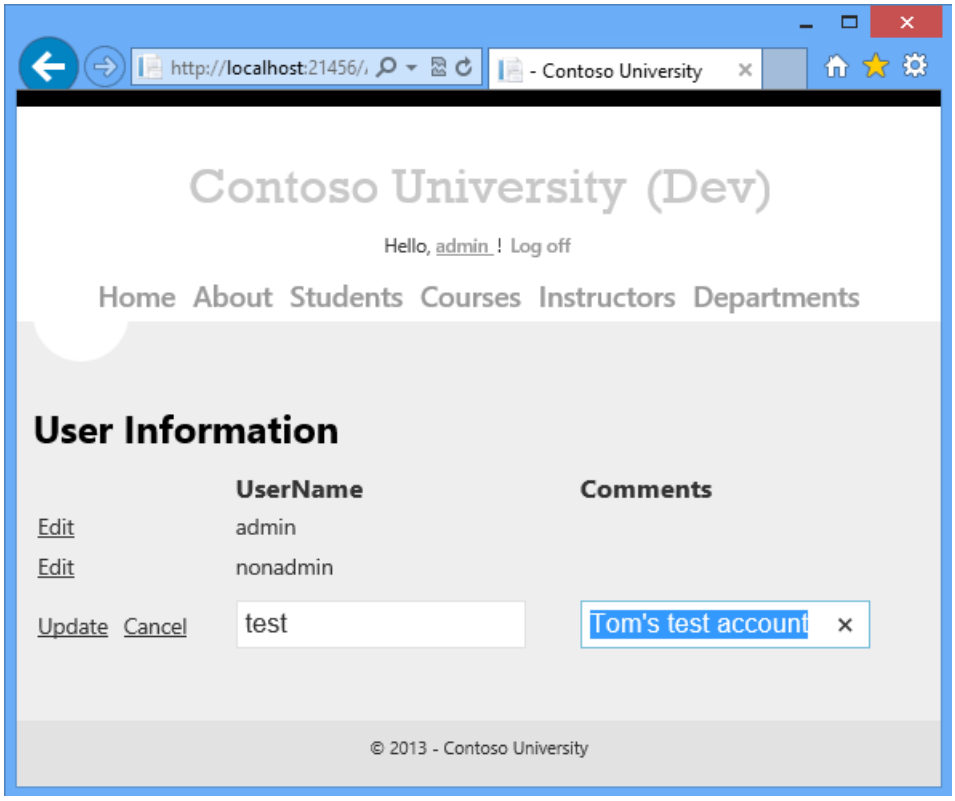
3. Copy the following markup into the `MainContent` `Content` element (the last of the 3 `Content` elements):

```
<h2>User Information</h2>
    <asp:SqlDataSource ID="SqlDataSource1" runat="server" ConnectionString="<%$
ConnectionStrings:DefaultConnection %>"
        SelectCommand="SELECT UserId, UserName, Comments FROM [Users]"
        UpdateCommand="UPDATE [Users] SET [UserName] = @UserName, [Comments] = @Comments WHERE
[UserId] = @UserId">
        <DeleteParameters>
            <asp:Parameter Name="UserId" Type="Object" />
        </DeleteParameters>
        <UpdateParameters>
            <asp:Parameter Name="UserId" Type="Object" />
            <asp:Parameter Name="UserName" Type="String" />
            <asp:Parameter Name="Comments" Type="String" />
        </UpdateParameters>
    </asp:SqlDataSource>

    <asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False" DataKeyNames="UserId"
DataSourceID="SqlDataSource1">
        <Columns>
            <asp:CommandField ShowEditButton="True" />
            <asp:BoundField DataField="UserName" HeaderText="UserName" SortExpression="UserName" />
            <asp:BoundField DataField="Comments" HeaderText="Comments" SortExpression="Comments" />
        </Columns>
    </asp:GridView>
```

4. Right-click the *UserInfo.aspx* page and click **View in Browser**.

5. Log in with your *admin* user credentials (password is *devpwd*) and add some comments to a user to verify that the page works correctly.

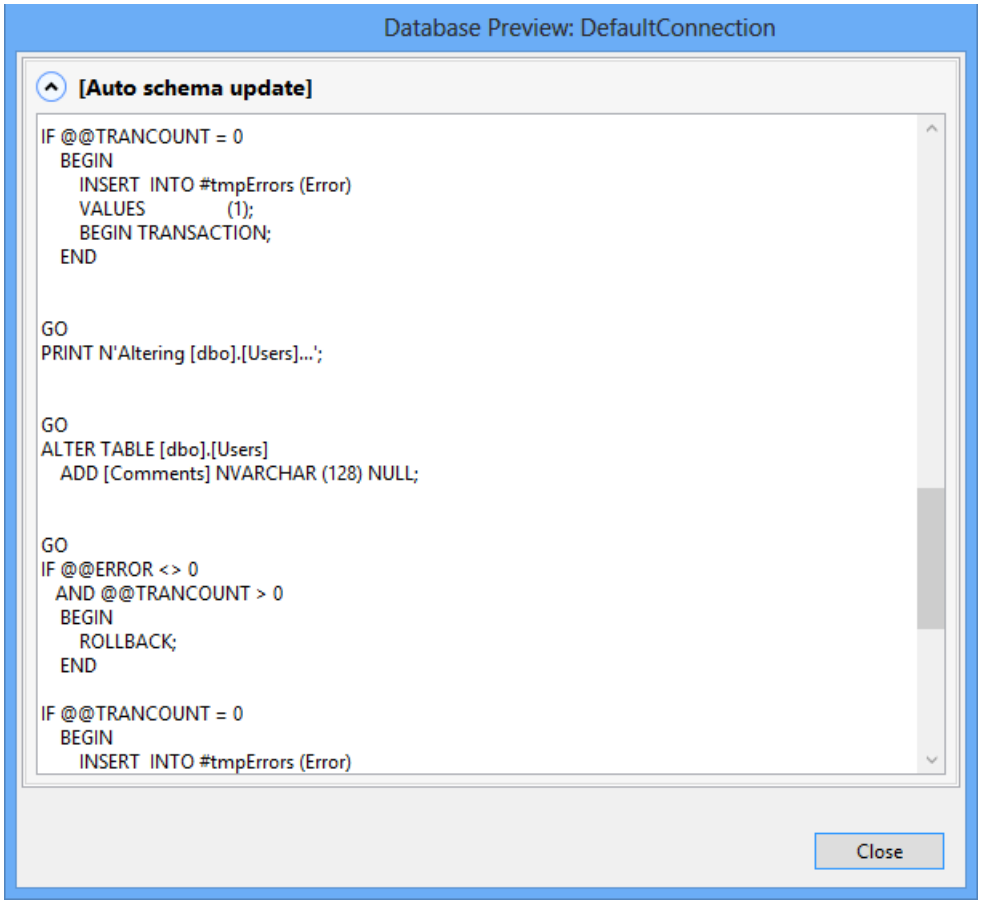


6. Close the browser.

Deploy the database update

To deploy by using the dbDacFx provider, you just need to select the **Update database** option in the publish profile. However, for the initial deployment when you used this option you also configured some additional SQL scripts to run: those are still in the profile and you'll have to prevent them from running again.

1. Open the **Publish Web** wizard by right-clicking the ContosoUniversity project and clicking **Publish**.
2. Select the **Test** profile.
3. Click the **Settings** tab.
4. Under **DefaultConnection**, select **Update database**.
5. Disable the additional scripts that you configured to run for the initial deployment:
   a. Click **Configure database updates**.
   b. In the **Configure Database Updates** dialog box, clear the check boxes next to *Grant.sql* and *aspnet-data-dev.sql*.
   c. Click **Close**.
6. Click the **Preview** tab.
7. Under **Databases** and to the right of **DefaultConnection**, click the **Preview database** link.



   The preview window shows the script that will be run in the destination database to make that database schema match the schema of the source database. The script includes an ALTER TABLE command that adds the new column.

8. Close the **Database Preview** dialog box, and then click **Publish**.

   Visual Studio deploys the updated application, and the browser opens to the home page.

9. Run the UserInfo page (add *Account/UserInfo.aspx* to the home page URL) to verify that the update was successfully deployed. You'll have to log in by entering *admin* and *devpwd*.

   Data in tables is not deployed by default, and you didn't configure a data deployment script to run, so you won't find the comment that you added in development. You can add a new comment now in staging to verify that the change was deployed to the database and the page works correctly.

10. Follow the same procedure to deploy to staging and production.

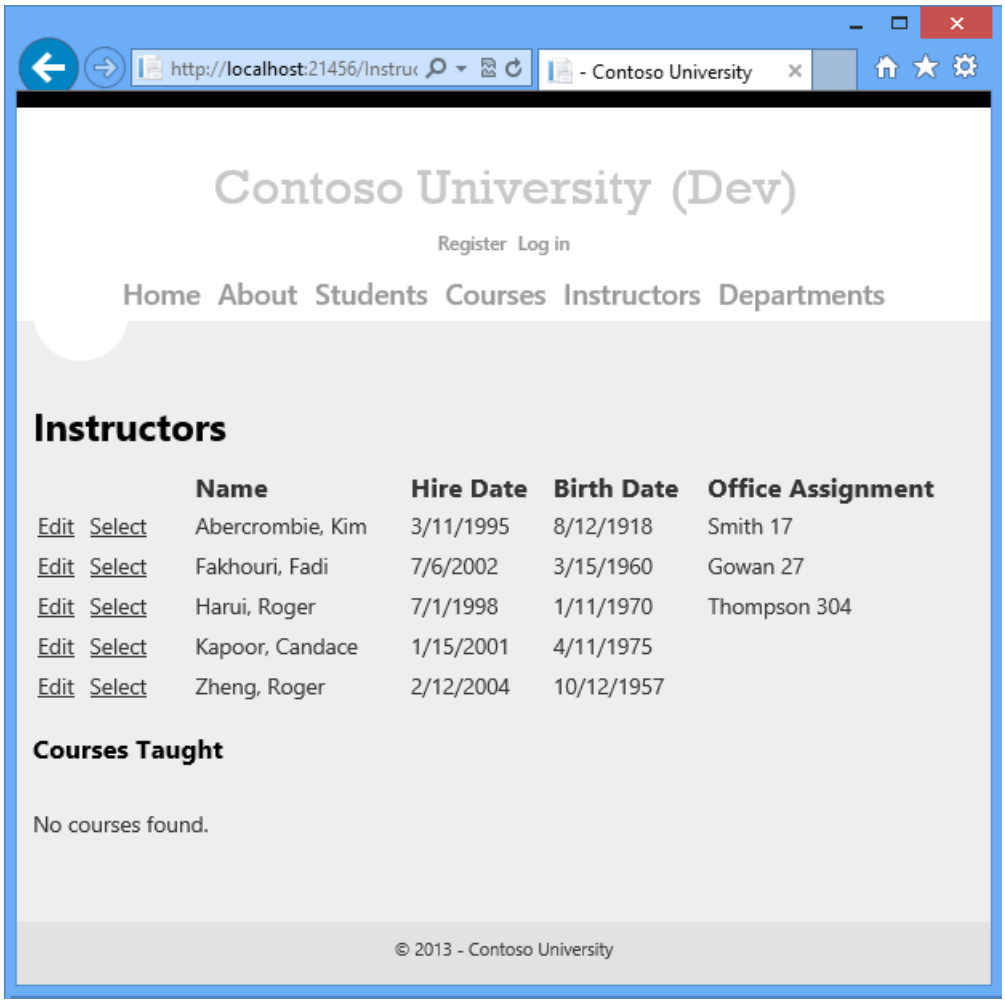
Don't forget to disable the extra scripts. The only difference compared to the Test profile is that you will disable only one script in the Staging and Production profiles because they were configured to run only *aspnet-prod-data.sql*.

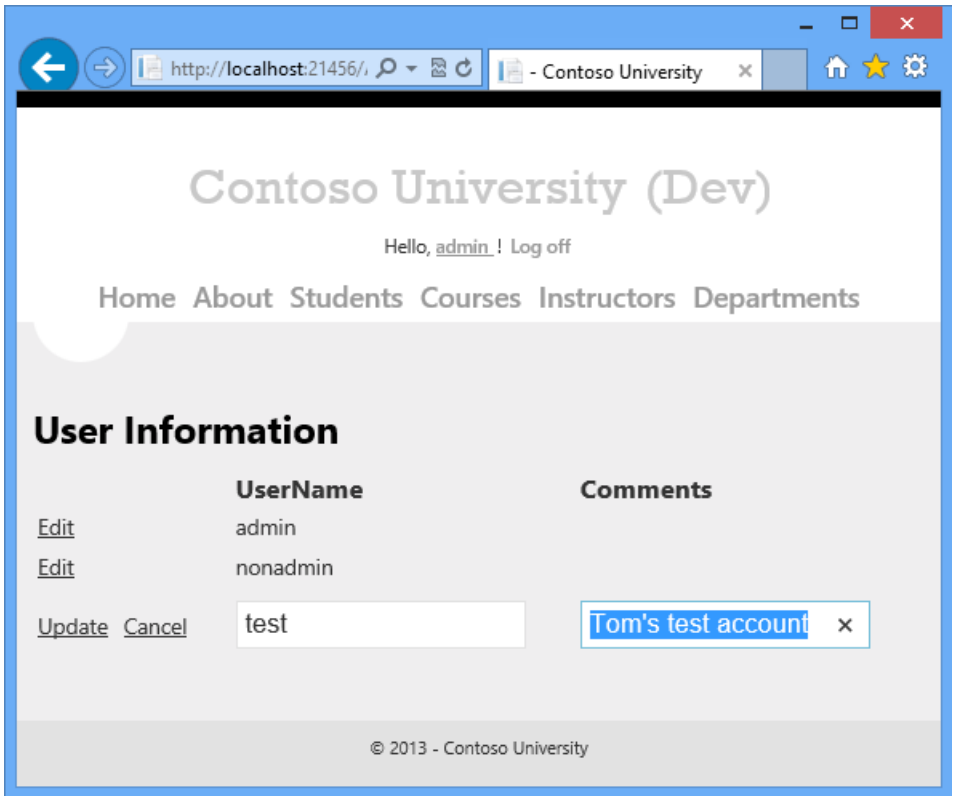
The credentials for staging and production are admin and prodpwd.

For a real production application update that includes a database change you would also typically take the application offline during deployment by uploading *app_offline.htm* before publishing and deleting it afterward, as you saw in the previous tutorial.

Summary

You've now deployed an application update that included a database change using both Code First Migrations and the dbDacFx provider.





The next tutorial shows you how to execute deployments by using the command line.

PREVIOUS NEXT

ASP.NET Web Deployment using Visual Studio: Command Line Deployment

2/20/2020 • 5 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

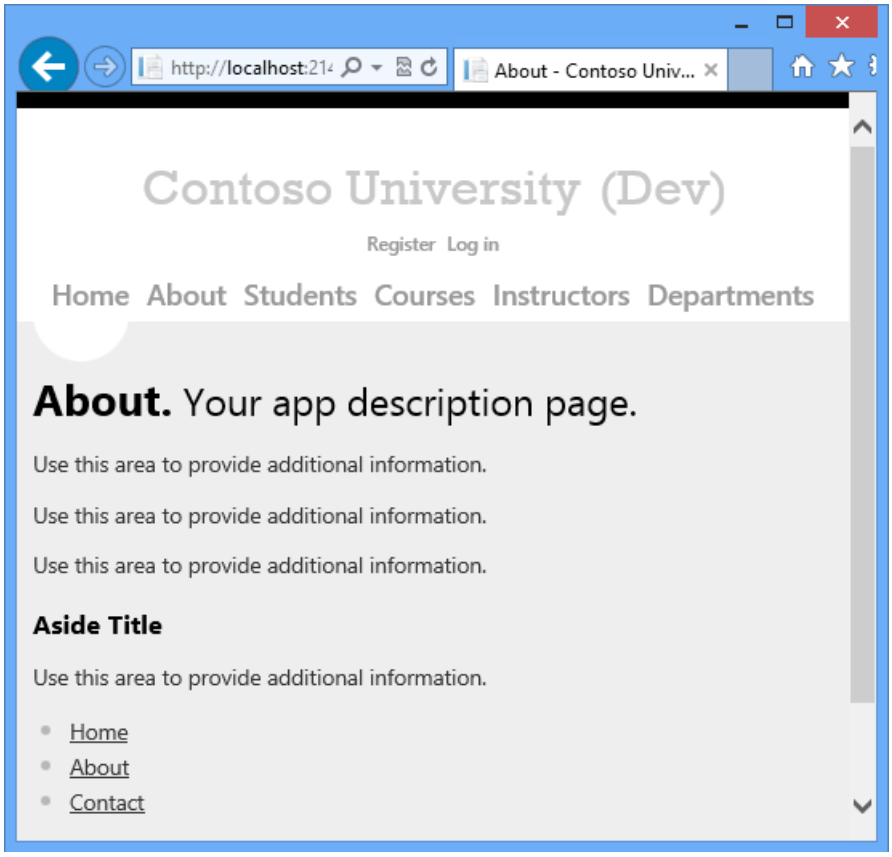
> This tutorial series shows you how to deploy (publish) an ASP.NET web application to Azure App Service Web Apps or to a third-party hosting provider, by using Visual Studio 2012 or Visual Studio 2010. For information about the series, see the first tutorial in the series.

Overview

This tutorial shows you how to invoke the Visual Studio web publish pipeline from the command line. This is useful for scenarios where you want to automate the deployment process instead of doing it manually in Visual Studio, typically by using a source code version control system.

Make a change to deploy

Currently the About page displays the template code.

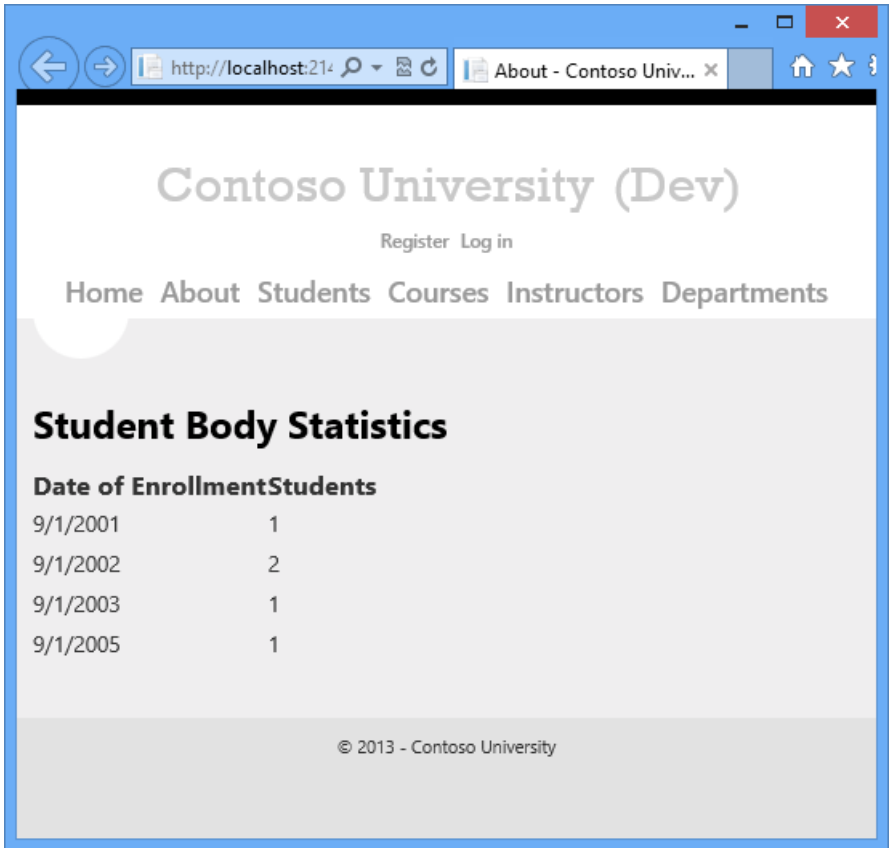


You'll replace that with code that displays a summary of student enrollment.

Open the *About.aspx* page, delete all of the markup inside the `MainContent` `Content` element, and insert the following markup in its place:

```
<h2>Student Body Statistics</h2>
    <asp:ObjectDataSource ID="StudentStatisticsObjectDataSource" runat="server"
TypeName="ContosoUniversity.BLL.SchoolBL"
        SelectMethod="GetStudentStatistics" DataObjectTypeName="ContosoUniversity.DAL.EnrollmentDateGroup">
    </asp:ObjectDataSource>
    <asp:GridView ID="StudentStatisticsGridView" runat="server" AutoGenerateColumns="False"
        DataSourceID="StudentStatisticsObjectDataSource">
        <Columns>
            <asp:BoundField DataField="EnrollmentDate" DataFormatString="{0:d}" HeaderText="Date of
Enrollment"
                ReadOnly="True" SortExpression="EnrollmentDate" />
            <asp:BoundField DataField="StudentCount" HeaderText="Students" ReadOnly="True"
                SortExpression="StudentCount" />
        </Columns>
    </asp:GridView>
```

Run the project and select the **About** page.



Deploy to Test by using the command line

You won't be deploying another database change, so disable dbDacFx database deployment for the aspnet-ContosoUniversity database. Open the **Publish Web** wizard, and in each of the three publish profiles, clear the **Update Database** check box on the **Settings** tab.

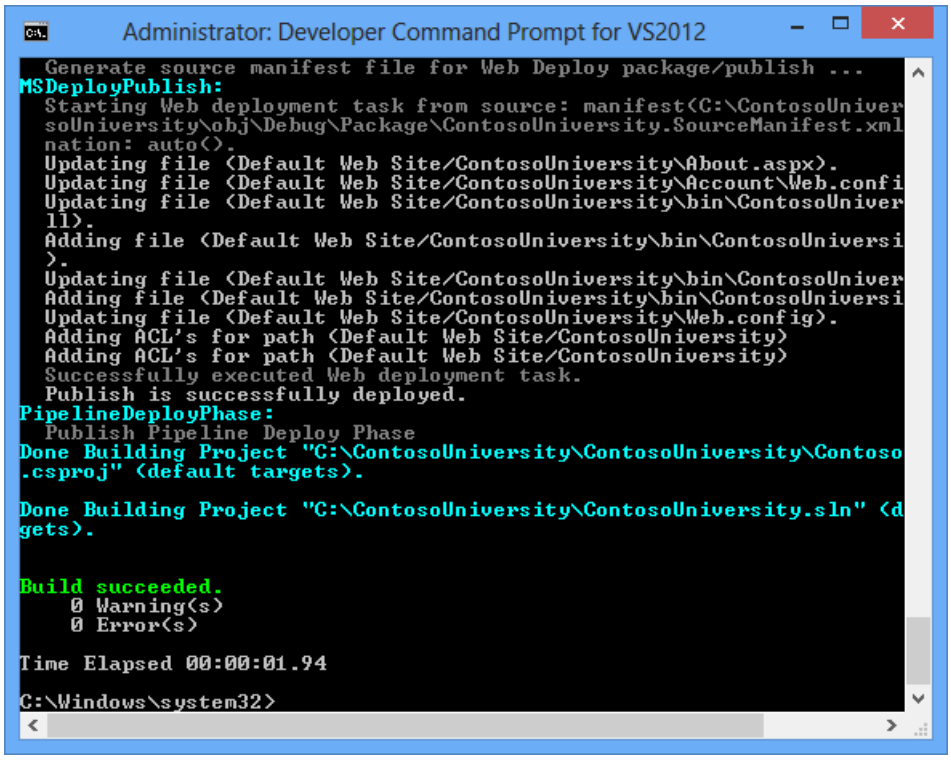
In the Windows 8 Start page, search for **Developer Command Prompt for VS2012**.

Right-click the icon for **Developer Command Prompt for VS2012** and click **Run as administrator**.

Enter the following command at the command prompt, replacing the path to the solution file with the path to your solution file:

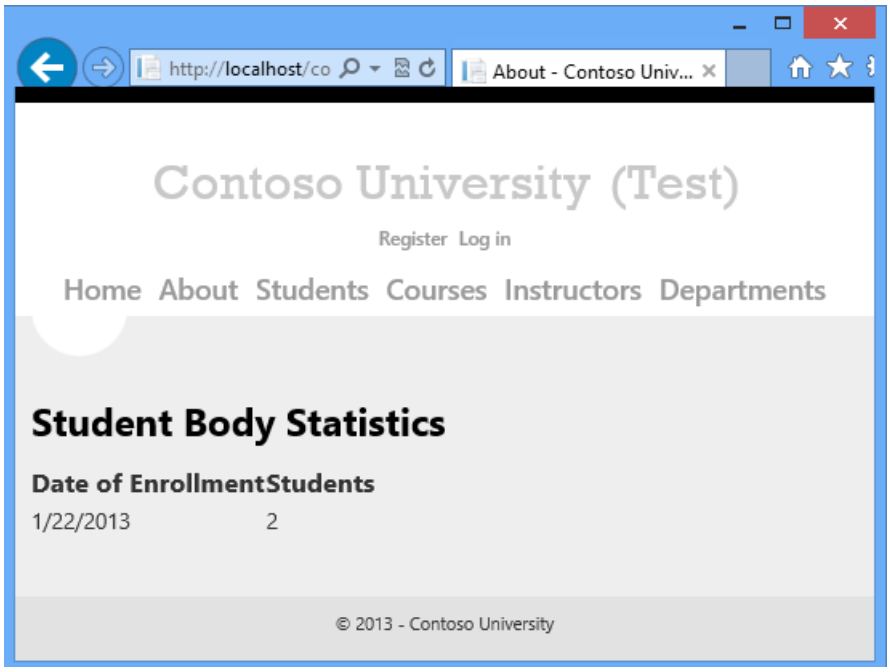
```
msbuild C:\ContosoUniversity\ContosoUniversity.sln /p:DeployOnBuild=true /p:PublishProfile=Test
```

MSBuild builds the solution and deploys it to the test environment.



Open a browser and go to `http://localhost/ContosoUniversity`, then click the **About** page to verify that the deployment was successful.

If you haven't created any students in test, you'll see an empty page under the **Student Body Statistics** heading. Go to the **Students** page, click **Add Student**, and add some students, and then return to the **About** page to see student statistics.



Key command line options

The command that you entered passed the solution file path and two properties to MSBuild:

```
msbuild C:\ContosoUniversity\ContosoUniversity.sln /p:DeployOnBuild=true
/p:PublishProfile=Test
```

Deploying the solution versus deploying individual projects

Specifying the solution file causes all projects in the solution to be built. If you have multiple web projects in the solution, the following MSBuild behavior applies:

- The properties that you specify on the command line are passed to every project. Therefore, each web project must have a publish profile with the name that you specify. If you specify `/p:PublishProfile=Test`, each web project must have a publish profile named *Test*.
- You might successfully publish one project when another one doesn't even build. For more information, see the stackoverflow thread MSBuild fails with two packages.

If you specify an individual project instead of a solution, you have to add a parameter that specifies the Visual Studio version. If you are using Visual Studio 2012 the command line would be similar to the following example:

```
msbuild C:\ContosoUniversity\ContosoUniversity\ContosoUniversity.csproj /p:DeployOnBuild=true
/p:PublishProfile=Test /p:VisualStudioVersion=11.0
```

The version number for Visual Studio 2010 is 10.0. For more information, see Visual Studio project compatibility and VisualStudioVersion on Sayed Hashimi's blog.

Specifying the publish profile

You can specify the publish profile by name or by the full path to the *.pubxml* file, as shown in the following example:

```
msbuild C:\ContosoUniversity\ContosoUniversity.sln /p:DeployOnBuild=true
/p:PublishProfile=C:\ContosoUniversity\ContosoUniversity\Properties\PublishProfiles\Test.pubxml
```

Web publish methods supported for command-line publishing

Three publish methods are supported for command line publishing:

- `MSDeploy` - Publish by using Web Deploy.
- `Package` - Publish by creating a Web Deploy Package. You have to install the package separately from the MSBuild command that creates it.
- `FileSystem` - Publish by copying files to a specified folder.

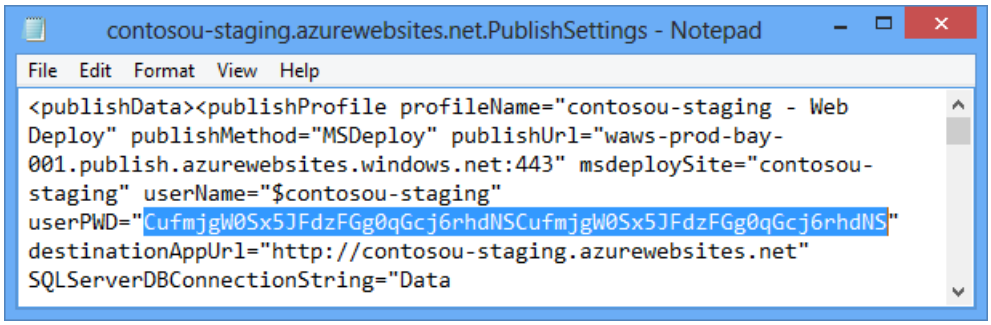
Specifying the build configuration and platform

The build configuration and platform must be set in Visual Studio or on the command line. The publish profiles include properties that are named `LastUsedBuildConfiguration` and `LastUsedPlatform`, but you can't set these properties in order to determine how the project is built. For more information, see MSBuild: how to set the configuration property on Sayed Hashimi's blog.

Deploy to staging

To deploy to Azure, you must add the password to the command line. If you saved the password in the publish profile in Visual Studio, it was stored in encrypted form in the your *.pubxml.user* file. That file is not accessed by MSBuild when you do a command line deployment, so you have to pass in the password in a command line parameter.

1. Copy the password that you need from the *.publishsettings* file that you downloaded earlier for the staging web site. The password is the value of the `userPWD` attribute for the Web Deploy `publishProfile` element.



2. In the Windows 8 Start page, search for **Developer Command Prompt for VS2012**, and click the icon to open the command prompt. (You don't have to open it as administrator this time because you aren't deploying to IIS on the local computer.)

3. Enter the following command at the command prompt, replacing the path to the solution file with the path to your solution file and the password with your password:

```
msbuild C:\ContosoUniversity\ContosoUniversity.sln /p:DeployOnBuild=true /p:PublishProfile=Staging
/p:Password=hdNSWsbuqno7J5uqnwKafwlfNPt1DSco7J5uqnwKafwlfNPt1DSpKHuYgCco7J5
/p:AllowUntrustedCertificate=true
```

   Notice that this command line includes an extra parameter: `/p:AllowUntrustedCertificate=true`. As this tutorial is being written, the `AllowUntrustedCertificate` property must be set when you publish to Azure from the command line. When the fix for this bug is released, you won't need that parameter.

4. Open a browser and go to the URL of your staging site, and then click the **About** page to verify that the deployment was successful.

   As you saw earlier for the test environment, you might have to create some students to see statistics on the **About** page.

Deploy to production

The process for deploying to production is similar to the process for staging.

1. Copy the password that you need from the *.publishsettings* file that you downloaded earlier for the production web site.

2. Open **Developer Command Prompt for VS2012**.

3. Enter the following command at the command prompt, replacing the path to the solution file with the path to your solution file and the password with your password:

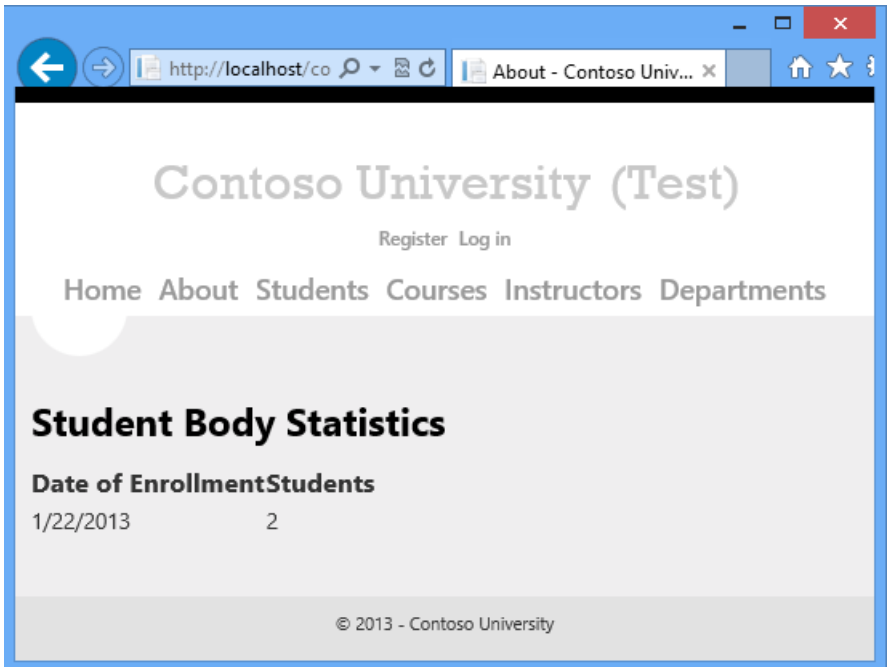
```
msbuild C:\ContosoUniversity\ContosoUniversity.sln /p:DeployOnBuild=true /p:PublishProfile=Production
/p:Password=hdNSWsbuqnwKafwlo7J5uqnwKafwlfNPt1DSqnwKafwlfNPt1DSpKHuYgCco7J5
/p:AllowUntrustedCertificate=true
```

   For a real production site, if there was also a database change, you would typically copy the *app_offline.htm* file to the site before deployment and delete it after successful deployment.

4. Open a browser and go to the URL of your staging site, and then click the **About** page to verify that the deployment was successful.

Summary

You have now deployed an application update by using the command line.



In the next tutorial, you will see an example of how to extend the web publish pipeline. The example will show you how to deploy files that are not included in the project.

PREVIOUS NEXT

ASP.NET Web Deployment using Visual Studio: Deploying Extra Files

2/20/2020 • 5 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This tutorial series shows you how to deploy (publish) an ASP.NET web application to Azure App Service Web Apps or to a third-party hosting provider, by using Visual Studio 2012 or Visual Studio 2010. For information about the series, see the first tutorial in the series.

Overview

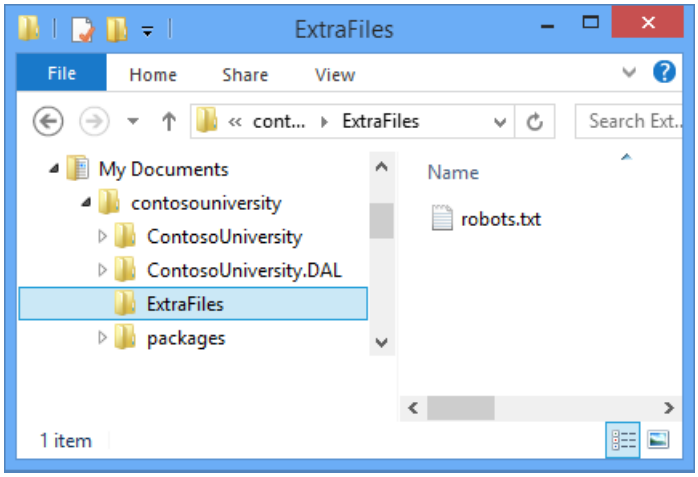
This tutorial shows how to extend the Visual Studio web publish pipeline to do an additional task during deployment. The task is to copy extra files that are not in the project folder to the destination web site.

For this tutorial you'll copy one extra file: *robots.txt*. You want to deploy this file to staging but not to production. In the Deploying to Production tutorial, you added this file to the project and configured the Production publish profile to exclude it. In this tutorial you'll see an alternative method to handle this situation, one that will be useful for any files that you want to deploy but don't want to include in the project.

Move the robots.txt file

To prepare for a different method of handling *robots.txt*, in this section of the tutorial you move the file to a folder that is not included in the project, and you delete *robots.txt* from the staging environment. It is necessary to delete the file from staging so that you can verify that your new method of deploying the file to that environment is working correctly.

1. In **Solution Explorer**, right-click the *robots.txt* file and click **Exclude From Project**.
2. Using Windows File Explorer, create a new folder in the solution folder and name it *ExtraFiles*.
3. Move the *robots.txt* file from the *ContosoUniversity* project folder to the *ExtraFiles* folder.



4. Using your FTP tool, delete the *robots.txt* file from the staging web site.

   As an alternative, you can select **Remove additional files at destination** under **File Publish Options** on the **Settings** tab of the Staging publish profile, and republish to staging.

Update the publish profile file

You only need *robots.txt* in staging, so the only publish profile you need to update in order to deploy it is Staging.

1. In Visual Studio, open *Staging.pubxml*.

2. At the end of the file, before the closing `</Project>` tag, add the following markup:

```
<Target Name="CustomCollectFiles">
    <ItemGroup>
      <_CustomFiles Include="..\ExtraFiles\**\*" />
      <FilesForPackagingFromProject Include="%(_CustomFiles.Identity)">
        <DestinationRelativePath>%(RecursiveDir)%(Filename)%(Extension)</DestinationRelativePath>
      </FilesForPackagingFromProject>
    </ItemGroup>
  </Target>
```

   This code creates a new *target* that will collect additional files to be deployed. A target is composed of one or more tasks that MSBuild will execute based on conditions you specify.

   The `Include` attribute specifies that the folder in which to find the files is *ExtraFiles*, located at the same level as the project folder. MSBuild will collect all files from that folder and recursively from any subfolders (the double asterisk specifies recursive subfolders). With this code you could put multiple files, and files in subfolders inside the *ExtraFiles* folder, and all will be deployed.

   The `DestinationRelativePath` element specifies that the folders and files should be copied to the root folder of the destination web site, in the same file and folder structure as they are found in the *ExtraFiles* folder. If you wanted to copy the *ExtraFiles* folder itself, the `DestinationRelativePath` value would be *ExtraFiles%(RecursiveDir)%(Filename)%(Extension)*.

3. At the end of the file, before the closing `</Project>` tag, add the following markup that specifies when to execute the new target.

```
<PropertyGroup>
    <CopyAllFilesToSingleFolderForPackageDependsOn>
      CustomCollectFiles;
      $(CopyAllFilesToSingleFolderForPackageDependsOn);
    </CopyAllFilesToSingleFolderForPackageDependsOn>

    <CopyAllFilesToSingleFolderForMsdeployDependsOn>
      CustomCollectFiles;
      $(CopyAllFilesToSingleFolderForMsdeployDependsOn);
    </CopyAllFilesToSingleFolderForMsdeployDependsOn>
</PropertyGroup>
```

   This code causes the new `CustomCollectFiles` target to be executed whenever the target that copies files to the destination folder is executed. There is a separate target for publish versus deployment package creation, and the new target is injected in both targets in case you decide to deploy by using a deployment package instead of publishing.

   The *.pubxml* file now looks like the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This file is used by the publish/package process of your Web project. You can customize the behavior
of this process
by editing this MSBuild file. In order to learn more about this please visit
https://go.microsoft.com/fwlink/?LinkID=208121.
```

```
-->
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>MSDeploy</WebPublishMethod>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <SiteUrlToLaunchAfterPublish>http://contosou-
staging.azurewebsites.net</SiteUrlToLaunchAfterPublish>
    <ExcludeApp_Data>True</ExcludeApp_Data>
    <MSDeployServiceURL>waws-prod-bay-001.publish.azurewebsites.windows.net:443</MSDeployServiceURL>
    <DeployIisAppPath>contosou-staging</DeployIisAppPath>
    <RemoteSitePhysicalPath />
    <SkipExtraFilesOnServer>False</SkipExtraFilesOnServer>
    <MSDeployPublishMethod>WMSVC</MSDeployPublishMethod>
    <UserName>$contosou-staging</UserName>
    <_SavePWD>True</_SavePWD>
    <PublishDatabaseSettings>
      <Objects xmlns="">
        <ObjectGroup Name="SchoolContext" Order="1" Enabled="True">
          <Destination Path="Data Source=tcp:sk0264hvc9.database.windows.net,1433;Initial
Catalog=ContosoUniversity-staging;User ID=CU-staging-admin@sk0264hvc9;Password=" Name="Data
Source=tcp:sk0264hvc9.database.windows.net,1433;Initial Catalog=ContosoUniversity-staging;User Id=CU-
staging-admin@sk0264hvc9;Password=" />
          <Object Type="DbCodeFirst">
            <Source Path="DBMigration" DbContext="ContosoUniversity.DAL.SchoolContext,
ContosoUniversity.DAL" MigrationConfiguration="ContosoUniversity.DAL.Migrations.Configuration,
ContosoUniversity.DAL" Origin="Configuration" />
          </Object>
        </ObjectGroup>
        <ObjectGroup Name="DefaultConnection" Order="2" Enabled="False">
          <Destination Path="Data Source=tcp:sk0264hvc9.database.windows.net,1433;Initial
Catalog=ContosoUniversity-staging;User ID=CU-staging-admin@sk0264hvc9;Password=" Name="Data
Source=tcp:sk0264hvc9.database.windows.net,1433;Initial Catalog=ContosoUniversity-staging;User Id=CU-
staging-admin@sk0264hvc9;Password=" />
          <Object Type="DbDacFx">
            <PreSource Path="Data Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\aspnet-
ContosoUniversity.mdf;Initial Catalog=aspnet-ContosoUniversity;Integrated Security=True"
includeData="False" />
            <Source
Path="$(IntermediateOutputPath)AutoScripts\DefaultConnection_IncrementalSchemaOnly.dacpac"
dacpacAction="Deploy" />
          </Object>
          <UpdateFrom Type="Web.Config">
            <Source MatchValue="Data Source=(LocalDb)\v11.0;Integrated Security=SSPI;Initial
Catalog=aspnet-ContosoUniversity;AttachDBFilename=|DataDirectory|\aspnet-ContosoUniversity.mdf"
MatchAttributes="$(UpdateFromConnectionStringAttributes)" />
          </UpdateFrom>
          <Object Type="DbFullSql" Enabled="False">
            <Source Path="..\aspnet-data-prod.sql" Transacted="False" />
          </Object>
        </ObjectGroup>
      </Objects>
    </PublishDatabaseSettings>
    <EnableMSDeployBackup>False</EnableMSDeployBackup>
  </PropertyGroup>
  <ItemGroup>
    <MSDeployParameterValue Include="$(DeployParameterPrefix)DefaultConnection-Web.config Connection
String">
      <ParameterValue>Data Source=tcp:sk0264hvc9.database.windows.net,1433;Initial
Catalog=ContosoUniversity-staging;User Id=CU-staging-admin@sk0264hvc9;Password=</ParameterValue>
    </MSDeployParameterValue>
    <MSDeployParameterValue Include="$(DeployParameterPrefix)SchoolContext-Web.config Connection
String">
      <ParameterValue>Data Source=tcp:sk0264hvc9.database.windows.net,1433;Initial
Catalog=ContosoUniversity-staging;User Id=CU-staging-admin@sk0264hvc9;Password=</ParameterValue>
    </MSDeployParameterValue>
  </ItemGroup>
  <Target Name="CustomCollectFiles">
    <ItemGroup>
```

```
      <_CustomFiles Include="..\ExtraFiles\**\*" />
      <FilesForPackagingFromProject Include="%(_CustomFiles.Identity)">
        <DestinationRelativePath>%(RecursiveDir)%(Filename)%(Extension)</DestinationRelativePath>
      </FilesForPackagingFromProject>
    </ItemGroup>
  </Target>
  <PropertyGroup>
    <CopyAllFilesToSingleFolderForPackageDependsOn>
      CustomCollectFiles;
      $(CopyAllFilesToSingleFolderForPackageDependsOn);
    </CopyAllFilesToSingleFolderForPackageDependsOn>

    <CopyAllFilesToSingleFolderForMsdeployDependsOn>
      CustomCollectFiles;
      $(CopyAllFilesToSingleFolderForMsdeployDependsOn);
    </CopyAllFilesToSingleFolderForMsdeployDependsOn>
  </PropertyGroup>
</Project>
```

4. Save and close the *Staging.pubxml* file.

Publish to staging

Using one-click publish or the command line, publish the application by using the Staging profile.

If you use one-click publish, you can verify in the **Preview** window that *robots.txt* will be copied. Otherwise, use your FTP tool to verify that the *robots.txt* file is in the root folder of the web site after deployment.

Summary

This completes this series of tutorials on deploying an ASP.NET web application to a third-party hosting provider. For more information about any of the topics covered in these tutorials, see the ASP.NET Deployment Content Map.

More information

If you know how to work with MSBuild files, you can automate many other deployment tasks by writing code in *.pubxml* files (for profile-specific tasks) or the project *.wpp.targets* file (for tasks that apply to all profiles). For more information about *.pubxml* and *.wpp.targets* files, see How to: Edit Deployment Settings in Publish Profile (.pubxml) Files and the .wpp.targets File in Visual Studio Web Projects. For a basic introduction to MSBuild code, see **The Anatomy of a Project File** in Enterprise Deployment Series: Understanding the Project File. To learn how to work with MSBuild files to perform tasks for your own scenarios, see this book: Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build by Sayed Ibraham Hashimi and William Bartholomew.

Acknowledgements

I would like to thank the following people who made significant contributions to the content of this tutorial series:

- Alberto Poblacion, MVP & MCT, Spain
- Jarod Ferguson, Data Platform Development MVP, United States
- Harsh Mittal, Microsoft
- Jon Galloway (twitter: @jongalloway)
- Kristina Olson, Microsoft
- Mike Pope, Microsoft
- Mohit Srivastava, Microsoft
- Raffaele Rialdi, Italy
- Rick Anderson, Microsoft

- Sayed Hashimi, Microsoft(twitter: @sayedihashimi)
- Scott Hanselman (twitter: @shanselman)
- Scott Hunter, Microsoft (twitter: @coolcsh)
- Srđan Božović, Serbia
- Vishal Joshi, Microsoft (twitter: @vishalrjoshi)

PREVIOUS NEXT

ASP.NET Web Deployment using Visual Studio: Troubleshooting

2/20/2020 • 20 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This tutorial series shows you how to deploy (publish) an ASP.NET web application to Azure App Service Web Apps or to a third-party hosting provider, by using Visual Studio 2012 or Visual Studio 2010. For information about the series, see the first tutorial in the series.

This page describes some common problems that may arise when you deploy an ASP.NET web application by using Visual Studio. For each one, one or more possible causes and corresponding solutions are provided.

The scenarios shown apply to both Azure and third-party hosting providers. For more information about troubleshooting web apps in Azure App Service, see the following resources:

- Troubleshoot a web app in Azure App Service using Visual Studio
- Monitor Web Apps in Azure App Service
- Announcing the release of Windows Azure SDK 2.0 for .NET (ScottGu's blog, shows how to get diagnostic logs in Visual Studio)

Server Error in '/' Application - Current Custom Error Settings Prevent Details of the Error from Being Viewed Remotely

Scenario

After deploying a site to a remote host, you get an error message that mentions the customErrors setting in the Web.config file but doesn't indicate what the actual cause of the error was:

```
Server Error in '/' Application.
Runtime Error

Description: An application error occurred on the server. The current custom error settings
for this application prevent the details of the application error from being viewed remotely
(for security reasons). It could, however, be viewed by browsers running on the local server
machine.

Details: To enable the details of this specific error message to be viewable on remote machines,
please create a <customErrors> tag within a "web.config" configuration file located in the
root directory of the current web application. This <customErrors> tag should then have its
"mode" attribute set to "Off".
```

Possible Cause and Solution

By default, ASP.NET shows detailed error information only when your web application is running on the local computer. Generally you don't want to display detailed error information when your web application is publicly available over the Internet, because hackers may be able to use this information to find vulnerabilities in the application. However, when you are deploying a site or updates to a site, sometimes something will go wrong and you need to get the actual error message.

To enable the application to display detailed error messages when it runs on the remote host, edit the

Web.config file to set customErrors mode off, redeploy the application, and run the application again:

1. If the application Web.config file has a customErrors element in the system.web element, change the mode attribute to "off". Otherwise add a customErrors element in the system.web element with the mode attribute set to "off", as shown in the following example:

```
<configuration>
  <system.web>
    <customErrors mode="off"/>
  </system.web>
</configuration>
```

2. Deploy the application.
3. Run the application and repeat whatever you did earlier that caused the error to occur. Now you can see what the actual error message is.
4. When you have resolved the error, restore the original customErrors setting and redeploy the application.

Cannot create/shadow copy 'ContosoUniversity' when that file already exists.

Scenario

When you try to run a project in Visual Studio you get an error page with a message like the following example:

Server Error in '/' Application. Cannot create/shadow copy 'ContosoUniversity' when that file already exists.

Possible Cause and Solution

Wait a minute and refresh the browser, or recompile the site and try running it again.

Access is Denied in a Web Page that Uses SQL Server Compact

Scenario

When you deploy a site that uses SQL Server Compact and you run a page in the deployed site that accesses the database, you see the following error message:

Access is denied. (Exception from HRESULT: 0x80070005 (E_ACCESSDENIED))

Possible Cause and Solution

The NETWORK SERVICE account on the server needs to be able to read SQL Service Compact native binaries that are in the *bin\amd64* or *bin\x86* folder, but it does not have read permissions for those folders. Set read permission for NETWORK SERVICE on the *bin* folder, making sure to extend the permissions to subfolders.

Cannot Read Configuration File Due to Insufficient Permissions

Scenario

When you click the Visual Studio publish button to deploy an application to IIS on your local machine, publishing fails and the **Output** window shows an error message similar to this:

An error occurred when reading the IIS Configuration File 'MACHINE/REDIRECTION'. The identity performing this operation was ... Error: Cannot read configuration file due to insufficient permissions.

Possible Cause and Solution

To use one-click publish to IIS on your local machine, you must be running Visual Studio with administrator permissions. Close Visual Studio and restart it with administrator permissions.

Could Not Connect to the Destination Computer ... Using the Specified Process

Scenario

When you click the Visual Studio publish button to deploy an application, publishing fails and the **Output** window shows an error message similar to this:

```
Web deployment task failed.(Could not connect to the destination computer ("<server URL>") using the
specified process
("The Web Management Service"). This can happen if a proxy server is interrupting communication with the
destination server.
Disable the proxy server and try again.) ... The remote server returned an error: (502) Bad Gateway.
```

Possible Cause and Solution

A proxy server is interrupting communication with the destination server. From the Windows Control Panel or in Internet Explorer, select **Internet Options** and select the **Connections** tab. In the **Internet Properties** dialog box, click **LAN Settings**. In the **Local Area Network (LAN) Settings** dialog box, clear the **Automatically detect settings** checkbox. Then click the publish button again.

If the problem persists, contact your system administrator to determine what can be done with proxy or firewall settings. The problem happens because Web Deploy uses a non-standard port for Web Management Service deployment (8172); for other connections, Web Deploy uses port 80. When you are deploying to a third-party hosting provider, you are typically using the Web Management Service.

Default .NET 4.0 Application Pool Does Not Exist

Scenario

When you deploy an application that requires the .NET Framework 4, you see the following error message:

The default .NET 4.0 application pool does not exist or the application could not be added. Please verify that ASP.NET 4.0 is installed on this machine.

Possible Cause and Solution

ASP.NET 4 is not installed in IIS. If the server you are deploying to is your development computer and has Visual Studio 2010 installed on it, ASP.NET 4 is installed on the computer but might not be installed in IIS. On the server that you are deploying to, open an elevated command prompt and install ASP.NET 4 in IIS by running the following commands:

```
cd %windir%\Microsoft.NET\Framework\v4.0.30319
aspnet_regiis.exe –iru
```

You might also need to manually set the .NET Framework version of the default application pool. For more information, see the Deploying to IIS as a Test Environment tutorial in this series.

Format of the initialization string does not conform to specification starting at index 0.

Scenario

After you deploy an application using one-click publish, when you run a page that accesses the database you get the following error message:

Format of the initialization string does not conform to specification starting at index 0.

Possible Cause and Solution

Open the *Web.config* file in the deployed site and check to see whether the connection string values begin with `$(ReplaceableToken_`, as in the following example:

```
<connectionStrings>
  <add name="DefaultConnection" connectionString="$(ReplaceableToken_DefaultConnection-Web.config
Connection String_0)" providerName="System.Data.SqlServerCe.4.0" />
  <add name="SchoolContext" connectionString="$(ReplaceableToken_SchoolContext-Web.config Connection
String_0)" providerName="System.Data.SqlServerCe.4.0" />
</connectionStrings>
```

If the connection strings look like this example, edit the project file and add the following property to the PropertyGroup element that is for all build configurations:

```
<AutoParameterizationWebConfigConnectionStrings>False</AutoParameterizationWebConfigConnectionStrings>
```

Then redeploy the application.

HTTP 500 Internal Server Error

Scenario

When you run the deployed site, you see the following error message without specific information indicating the cause of the error:

HTTP Error 500 - Internal Server Error.

Possible Cause and Solution

There are many causes of 500 errors, but one possible cause if you are following these tutorials is that you put an XML element in the wrong place in one of the Web.config transformation files. For example, you would get this error if you put the transformation that inserts a <location> element under <system.web> instead of directly under <configuration>. You can use the Web.config transform preview feature to verify that transformations are working as intended. The solution if you find a transform that was coded incorrectly is to correct the transformation file and redeploy. If an error isn't obvious, try commenting out transforms and redeploying to see which one is causing the 500 error.

HTTP 500.21 Internal Server Error

Scenario

When you run the deployed site, you see the following error message:

HTTP Error 500.21 - Internal Server Error. Handler "PageHandlerFactory-Integrated" has a bad module "ManagedPipelineHandler" in its module list.

Possible Cause and Solution

The site you have deployed targets ASP.NET 4, but ASP.NET 4 is not registered in IIS on the server. On the server open an elevated command prompt and register ASP.NET 4 by running the following commands:

```
cd %windir%\Microsoft.NET\Framework\v4.0.30319
aspnet_regiis.exe –iru
```

You might also need to manually set the .NET Framework version of the default application pool. For more information, see the Deploying to IIS as a Test Environment tutorial in this series.

Login Failed Opening SQL Server Express Database in App_Data

Scenario

You updated the *Web.config* file connection string to point to a SQL Server Express database as an *.mdf* file in your *App_Data* folder, and the first time you run the application you see the following error message:

System.Data.SqlClient.SqlException: Cannot open database "DatabaseName" requested by the login. The login failed.

Possible Cause and Solution

The name of the *.mdf* file cannot match the name of any SQL Server Express database that has ever existed on your computer, even if you deleted the *.mdf* file of the previously existing database. Change the name of the *.mdf* file to a name that has never been used as a database name and change the *Web.config* file to use the new name. As an alternative, you can use SQL Server Management Studio Express to delete previously existing SQL Server Express databases.

Model Compatibility Cannot be Checked

Scenario

You updated the *Web.config* file connection string to point to a new SQL Server Express database, and the first time you run the application you see the following error message:

Model compatibility cannot be checked because the database does not contain model metadata. Ensure that IncludeMetadataConvention has been added to the DbModelBuilder conventions.

Possible Cause and Solution

If the database name you put in the Web.config file was ever used before on your computer, a database might already exist with some tables in it. Select a new name that has not been used on your computer before and change the *Web.config* file to point to use this new database name. As an alternative, you can use SQL Server Express Utility or SQL Server Management Studio Express to delete the existing database.

SQL Error When a Script Attempts to Create Users or Roles

Scenario

You are using database deployment configured on the **Package/Publish SQL** tab, SQL scripts that run during deployment include Create User or Create Role commands, and script execution fails when those commands are executed. You might see more detailed messages, such as the following:

```
The approximate location of the error was between lines '1' and '3' of the script.
The verbose log may have more information about the error. The command started with:
CREATE USER [user2] FOR LOGIN [user2] WITH DEFAULT
Error: User does not have permission to perform this action.
```

If this error occurs when you have configured database deployment in the **Publish Web** wizard rather than the **Package/Publish SQL** tab, create a thread in the Configuration and Deployment forum, and the solution will be added to this troubleshooting page.

Possible Cause and Solution

The user account you are using to perform deployment does not have permission to create users or roles. For example, the hosting company might assign the db_datareader, db_datawriter, and db_ddladmin roles to the user account that it sets up for you. These are sufficient for creating most database objects, but not for creating users or roles. One way to avoid the error is by excluding users and roles from database deployment. You can do this by editing the PreSource element for the database's automatically generated script so that it includes the following attributes:

```
CopyAllUsers=false, CopyAllRoles=false
```

For information about how to edit the PreSource element in the project file, see How to: Edit Deployment Settings in the Project File. If the users or roles in your development database need to be in the destination database, contact your hosting provider for assistance.

SQL Server Timeout Error When Running Custom Scripts During Deployment

Scenario

You have specified custom SQL scripts to run during deployment, and when Web Deploy runs them, they time out.

Possible Cause and Solution

Running multiple scripts that have different transaction modes can cause time-out errors. By default, automatically generated scripts run in a transaction, but custom scripts do not. If you select the **Pull data and/or schema from an existing database** option on the **Package/Publish SQL** tab, and if you add a custom SQL script, you must change transaction settings on some scripts so that all scripts use the same transaction settings. For more information, see How to: Deploy a Database With a Web Application Project.

If you have configured transaction settings so that all are the same but still get this error, a possible workaround is to run the scripts separately. In the **Database Scripts** grid in the **Package/Publish** SQL tab, clear the **Include** check box for the script that causes the timeout error, then publish the project. Then go back into the **Database Scripts** grid, select that script's **Include** check box, and clear the **Include** check boxes for the other scripts. Then publish the project again. This time when you publish, only the selected custom script runs.

Stream Data of Site Manifest Is Not Yet Available

Scenario

When you are installing a package using the *deploy.cmd* file with the t (test) option, you see the following error message:

Error: The stream data of 'sitemanifest/dbFullSql[@path='C:\TEMP\AdventureWorksGrant.sql']/sqlScript' is not yet available.

Possible Cause and Solution

The error message means that the command cannot produce a test report. However, the command might run if you use the y (actual installation) option. The message indicates only that there is a problem with running the command in test mode.

This Application Requires ManagedRuntimeVersion v4.0

Scenario

When you attempt to deploy, you see the following error message:

The application pool that you are trying to use has the 'managedRuntimeVersion' property set to 'v2.0'. This application requires 'v4.0'.

Possible Cause and Solution

ASP.NET 4 is not installed in IIS. If the server you are deploying to is your development computer and has Visual Studio 2010 installed on it, ASP.NET 4 is installed on the computer but might not be installed in IIS. On the server that you are deploying to, open an elevated command prompt and install ASP.NET 4 in IIS by running the

following commands:

```
cd %windir%\Microsoft.NET\Framework\v4.0.30319
aspnet_regiis.exe -i
```

Unable to cast Microsoft.Web.Deployment.DeploymentProviderOptions

Scenario

When you are deploying a package, you see the following error message:

Unable to cast object of type 'Microsoft.Web.Deployment.DeploymentProviderOptions' to 'Microsoft.Web.Deployment.DeploymentProviderOptions'.

Possible Cause and Solution

You are trying to deploy from IIS Manager using the Web Deploy 1.1 UI to a server that has Web Deploy 2.0 installed. If you are using the IIS Remote Administration Tool to deploy by importing a package, check the **New Features Available** dialog box when you establish the connection. (This dialog box might only be shown once when the connection is first established. To clear the connection and start over, close IIS Manager and start it up again by entering inetmgr /reset at the command prompt.) If one of the features listed is **Web Deploy UI**, and it has a version number lower than 8, the server you are deploying to might have both 1.1 and 2.0 versions of Web Deploy installed. To deploy from a client that has 2.0 installed, the server must have only Web Deploy 2.0 installed. You will have to contact your hosting provider to resolve this problem.

Unable to load the native components of SQL Server Compact

Scenario

When you run the deployed site, you see the following error message:

Unable to load the native components of SQL Server Compact corresponding to the ADO.NET provider of version 8482. Install the correct version of SQL Server Compact. Refer to KB article 974247 for more details.

Possible Cause and Solution

The deployed site does not have *amd64* and *x86* subfolders with the native assemblies in them under the application's *bin* folder. On a computer that has SQL Server Compact installed, the native assemblies are located in *C:\Program Files\Microsoft SQL Server Compact Edition\v4.0\Private*. The best way to get the correct files into the correct folders in a Visual Studio project is to install the NuGet SqlServerCompact package. Package installation adds a post-build script to copy the native assemblies into *amd64* and *x86*. In order for these to be deployed, however, you have to manually include them in the project. For more information, see the Deploying SQL Server Compact tutorial.

"Path is not valid" error after deploying an Entity Framework Code First application

Scenario

You deploy an application that uses Entity Framework Code First Migrations and a DBMS such as SQL Server Compact which stores its database in a file in the App_Data folder. You have Code First Migrations configured to create the database after your first deployment. When you run the application you get an error message like the following example:

The path is not valid. Check the directory for the database. [Path = c:\inetpub\wwwroot\App_Data\DatabaseName.sdf]

Possible Cause and Solution

Code First is attempting to create the database but the App_Data folder does not exist. Either you didn't have any files in the *App_Data* folder when you deployed, or you selected **Exclude App_Data** on the **Package/Publish Web** tab of the Project Properties window. The deployment process won't create a folder on the server if there are no files in the folder to be copied to the server. If you already had the database set up in the site, the deployment process will delete the files and the *App_Data* folder itself if you selected **Remove additional files at destination** in the publish profile. To solve the problem, put a placeholder file such as a .txt file in the *App_Data* folder, make sure you do not have **Exclude App_Data** selected, and redeploy.

"COM object that has been separated from its underlying RCW cannot be used."

Scenario

You have been successfully using one-click publish to deploy your application and then you start getting this error:

Web deployment task failed. (Could not complete the request to remote agent URL 'https://serverurl.com/msdeploy.axd?site=sitename'.)
Could not complete the request to remote agent URL 'https://url/msdeploy.axd?site=sitename'.
The request was aborted: The request was canceled.
COM object that has been separated from its underlying RCW cannot be used.

Possible Cause and Solution

Closing and restarting Visual Studio is usually all that is required to resolve this error.

Deployment Fails Because User Credentials Used for Publishing Don't Have setACL Authority

Scenario

Publishing fails with an error that indicates you don't have authority to set folder permissions (the user account you are using doesn't have setACL authority).

Possible Cause and Solution

By default, Visual Studio sets read permissions on the root folder of the site and write permissions on the App_Data folder. If you know that the default permissions on site folders are correct and do not need to be set, you disable this behavior by adding **<IncludeSetACLProviderOnDestination>False</IncludeSetACLProviderOnDestination>** to the publish profile file (to affect a single profile) or to the wpp.targets file (to affect all profiles). For information about how to edit these files, see How to: Edit Deployment Settings in Profile (.pubxml) Files.

Access Denied Errors when the Application Tries to Write to an Application Folder

Scenario

Your application errors when it tries to create or edit a file in one of the application folders, because it does not have write authority for that folder.

Possible Cause and Solution

By default, Visual Studio sets read permissions on the root folder of the site and write permissions on the App_Data folder. If your application needs write access to a sub-folder, you can set permissions for that folder as shown in the Setting Folder Permissions and Deploying to the Production Environment tutorials in this series. If your application needs write access to the root folder of the site, you have to prevent it from setting read-only

access on the root folder by adding **<IncludeSetACLProviderOnDestination>False</IncludeSetACLProviderOnDestination>** to the publish profile file (to affect a single profile) or to the wpp.targets file (to affect all profiles). For information about how to edit these files, see How to: Edit Deployment Settings in Profile (.pubxml) Files.

Configuration Error - targetFramework attribute references a version that is later than the installed version of the .NET Framework

Scenario

You successfully published a web project that targets ASP.NET 4.5, but when you run the application (with the customErrors mode set to "off" in the Web.config file) you get the following error:

The 'targetFramework' attribute in the <compilation> element of the Web.config file is used only to target version 4.0 and later of the .NET Framework (for example, '<compilation targetFramework="4.0">'). The 'targetFramework' attribute currently references a version that is later than the installed version of the .NET Framework. Specify a valid target version of the .NET Framework, or install the required version of the .NET Framework.

The Source Error box of the error page highlights the following line from Web.config as the cause of the error:

<compilation targetFramework="4.5" />

Possible Cause and Solution

The server does not support ASP.NET 4.5. Contact the hosting provider to determine when and if support for ASP.NET 4.5 can be added. If upgrading the server is not an option, you have to deploy a web project that targets ASP.NET 4 or earlier instead.

If you deploy an ASP.NET 4 or earlier web project to the same destination, select the **Remove additional files at destination** check box on the **Settings** tab of the **Publish Web** wizard. If you don't select **Remove additional files at destination**, you will continue to get the Configuration Error page.

The project **Properties** windows includes a Target framework drop-down list, but you can't resolve this problem by just changing that from **.NET Framework 4.5** to **.NET Framework 4**. If you change the target framework to an earlier framework version, the project will still have references to the later framework version's assemblies and will not run. You have to manually change those references or create a new project that targets .NET Framework 4 or earlier. For more information, see .NET Framework Targeting for Web Sites.

Medium Trust Errors

Scenario

When you run your application in production, it gets an error related to medium trust.

Possible Cause and Solution

Many third-party hosting providers run your web site in medium trust, which means that there are some things it isn't allowed to do. For example, application code can't access the Windows registry and can't read or write files that are outside of your application's folder hierarchy. By default your application runs in *full trust* on your local computer, which means that the application might be able to do things that would fail when you deploy it to production.

You can configure the application to run in medium trust in the local IIS environment in order to troubleshoot. To do that, open the application *Web.config* file, and add a **trust** element in the **system.web** element, as shown in this example.

```
<configuration>
  <!-- Settings -->
  <system.web>
    <trust level="Medium" />
    <!-- Settings -->
  </system.web>
</configuration>
```

The application will now run in medium trust in IIS even on your local computer.

Don't do this if you are deploying to Azure App Service, because Azure does not require medium trust. At the time this tutorial is being written in February, 2012, using this method to make your application run in medium trust will cause an error in Azure.

If you are using Entity Framework Code First Migrations and you are deploying to a hosting provider that runs your application in medium trust, make sure that you have version 5.0 or later installed. In Entity Framework version 4.3, Migrations requires full trust in order to update the database schema.

HTTP 404.17 Not Found Error

Scenario

When you run the deployed site on your development computer in IIS, you see the following error message reporting that the server can't process Default.aspx:

HTTP Error 404.17 - Not Found

The requested content appears to be script and will not be served by the static file handler.

Possible Cause and Solution

ASP.NET 4.5 might not be installed on your computer. See the steps in the Deploying to IIS as a Test Environment tutorial in this series that explain how to install ASP.NET 4.5.

PREVIOUS

ASP.NET 4 - Enterprise Deployment Introduction

2/20/2020 • 2 minutes to read • Edit Online

This set of tutorials describes tools and techniques you can use to deploy web applications in various enterprise scenarios.

- Deploying Web Applications in Enterprise Scenarios using Visual Studio 2010
- Enterprise Web Deployment: Scenario Overview
- Application Lifecycle Management: From Development to Production

Deploying Web Applications in Enterprise Scenarios using Visual Studio 2010

2/20/2020 • 4 minutes to read • Edit Online

by Jason Lee

Download PDF

This set of tutorials describes tools and techniques you can use to deploy web applications in various enterprise scenarios. It explains how to make best use of technologies like Visual Studio 2010, the Microsoft Build Engine (MSBuild), Internet Information Services (IIS) 7.5, the IIS Web Deployment Tool (Web Deploy), the Web Farm Framework (WFF), and utilities like VSDBCMD.exe to simplify and manage the deployment process. It includes conceptual overviews and task-oriented guidance that will help you to:

- Review and establish the deployment requirements for an enterprise-scale web application.
- Configure test, staging, and production web server environments to support web deployment.
- Configure Team Foundation Server (TFS) continuous integration (CI) processes to support automated web deployment.
- Deploy enterprise-scale web applications to different server environments with varying requirements and restrictions.
- Deploy changes to web applications that are running in different server environments.

> **NOTE**
>
> While these tutorials describe the use of TFS as a CI server, the guidance is easily adapted to any CI server. You don't need a detailed knowledge of TFS to understand and leverage the tutorials.

For an Italian translation of these tutorials, visit http://www.lucamorelli.it.

About the Authors

Jason Lee is a principal technologist with Content Master where he has been working with Microsoft products and technologies, especially SharePoint and ASP.NET, for several years. Jason holds a PhD in computing and is currently MCPD and MCTS certified. You can read Jason's technical blog at www.jrjlee.com.

Benjamin Curry is a principal technologist with Content Master who has written whitepapers, SDK documentation, PowerPoint presentations, and instructor-led and online training courses during his career. An original member of the ASP.NET documentation team, he has worked with Microsoft's web technologies for over a decade.

Target Audience

This set of tutorials is for ASP.NET web application developers and solution architects who use Visual Studio 2010 to create enterprise-scale web applications. To get the most value from the content, you should be comfortable using Visual Studio 2010 and have a basic familiarity with TFS, together with an awareness of Microsoft web platform technologies like ASP.NET MVC 3, Windows Communication Foundation (WCF), IIS, SQL Server, and Visual Studio database projects. However, you do not need to be familiar with deployment tools and technologies or need to know how to set up CI systems.

Requirements

To follow the walkthroughs and perform the tasks that these tutorials describe, you'll need to install this software on your development computer:

- Visual Studio 2010 Premium or Ultimate Edition with Service Pack 1
- .NET Framework 4.0
- .NET Framework 3.5 with Service Pack 1
- ASP.NET MVC 3.0
- IIS 7.5 Express
- SQL Server Express 2008 R2

To perform the deployment steps described throughout these walkthroughs, you'll need to have access to sample Web application deployment environments. For best results, these environments should reflect your organization's enterprise deployment pattern. You can then modify the walkthroughs provided in this documentation to reflect the deployment environments and requirements of your own organization.

Series Contents

This introductory section consists of two further topics. These are designed to provide some broader context for the tutorials that follow:

- Enterprise Web Deployment: Scenario Overview. This topic describes the scenario that underpins each of the tutorials in this series. The scenario focuses on the Application Lifecycle Management (ALM) requirements of a fictional company named Fabrikam, Inc. as it develops an enterprise-scale web application.
- Application Lifecycle Management: From Development to Production. This topic provides a high-level, end-to-end overview of a deployment process. It illustrates how Fabrikam,Inc. moves an enterprise-scale ASP.NET web application through test, staging, and production environments as part of a continuous development process.

The series includes four tutorial sets. Each focuses on different aspects of web deployment:

- Web Deployment in the Enterprise. This tutorial provides a conceptual introduction to MSBuild project files, the Web Publishing Pipeline, Web Deploy, and other related technologies. It explains how you can use these tools together to manage complex deployment processes.
- Configuring Server Environments for Web Deployment. This tutorial describes how to configure Windows servers to support various deployment scenarios, including remote web package deployment using the Web Deployment Agent Service (the "remote agent") or the Web Deploy Handler and remote database deployment. It provides guidance on choosing the appropriate deployment method for your own environment, and it describes how to use the WFF to replicate deployed web applications across all the web servers in a server farm.
- Configuring Team Foundation Server for Web Deployment. This tutorial describes how to configure TFS to support various deployment scenarios, including automated deployment as part of a CI process and manually triggered deployments of specific builds.
- Advanced Enterprise Web Deployment. This tutorial describes how to accomplish various more advanced deployment tasks, like customizing database deployments for multiple environments, excluding files and folders from deployment, and taking web applications offline during the deployment process.

Where to Start

This set of tutorials uses a sample solution with a realistic level of complexity, together with a fictional enterprise deployment scenario, to provide a reference implementation and to give the tasks and walkthroughs a common context. The next topic, Enterprise Web Deployment: Scenario Overview, introduces the scenario and the sample

solution. From there you can work through the tutorials and topics that most closely match your needs.

NEXT

Enterprise Web Deployment: Scenario Overview

2/20/2020 • 7 minutes to read • Edit Online

by Jason Lee

Download PDF

This set of tutorials uses a sample solution with a realistic level of complexity, together with a fictional enterprise deployment scenario, to provide a reference implementation and to give the tasks and walkthroughs a common context. This topic describes the tutorial scenario and introduces the sample solution.

Scenario Description

Fabrikam, Inc., a fictitious company, is creating a solution that lets remote sales teams store and retrieve contact information from a web interface.

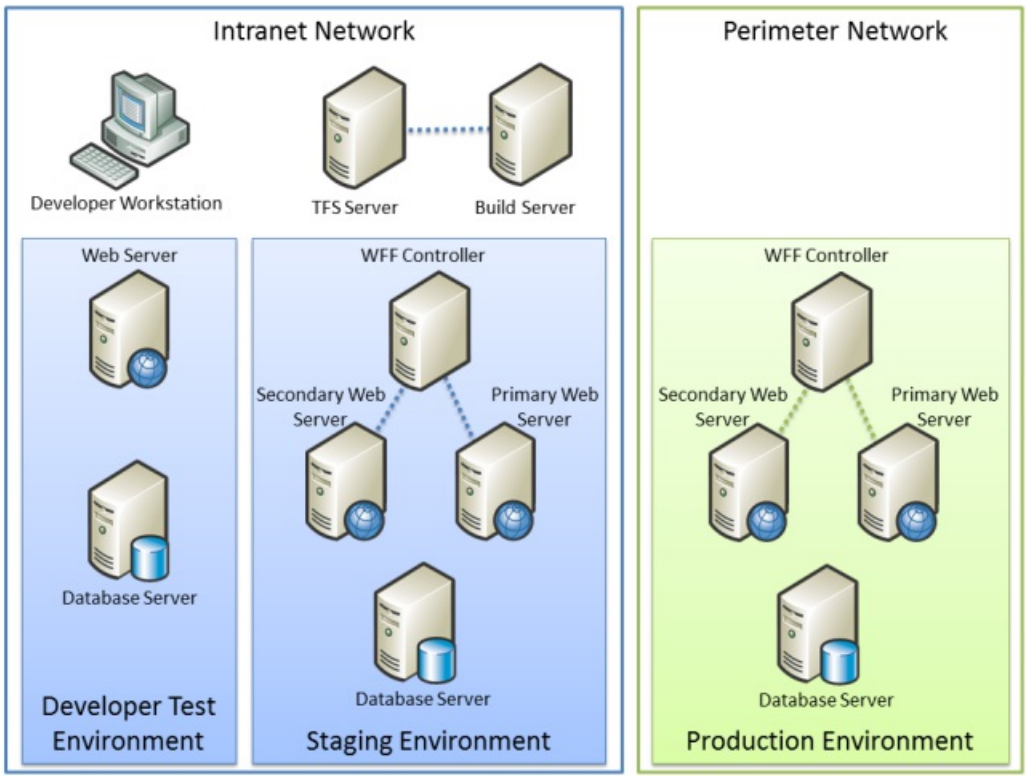
The Application Lifecycle Management (ALM) processes at Fabrikam, Inc. require the solution to be deployed to three server environments at various stages of the software development process:

- A developer test or "sandbox" environment.
- An intranet-based staging environment.
- An Internet-facing production environment.

Each of these environments has different configuration and security requirements, and each poses unique deployment challenges.

The Fabrikam, Inc. Server Infrastructure

This is the high-level development and deployment infrastructure at Fabrikam, Inc.



The developer workstations, the source control infrastructure, the developer test environment, and the staging

environment all reside on the intranet network within the Fabrikam.net domain. The production environment resides on a perimeter network (also known as DMZ, demilitarized zone, and screened subnet), which is isolated from the intranet network by a firewall. This is a common deployment scenario: you typically isolate your Internet-facing web servers from your internal server infrastructure through the use of firewalls or gateway servers.

In this example:

- A Team Foundation Server (TFS) 2010 server with a separate build server provides source control and continuous integration (CI) functionality.
- The developer test environment includes an Internet Information Services (IIS) 7.5 web server and a SQL Server 2008 R2 database server.
- The production environment includes multiple IIS 7.5 web servers synchronized by a Web Farm Framework (WFF) controller server, together with a SQL Server 2008 R2 database server. In practice, the database server may use clustering or mirroring to improve scalability and availability.
- The staging environment is designed to replicate the configuration of the production environment as closely as possible.
- The firewall and network isolation policies do not permit direct, automated deployment from the intranet to the perimeter network.

The configuration of each of these environments is described in more detail in the second tutorial, Configuring Server Environments for Web Deployment.

Team Roles for ALM

These users are involved in creating, managing, building, and publishing the Contact Manager solution:

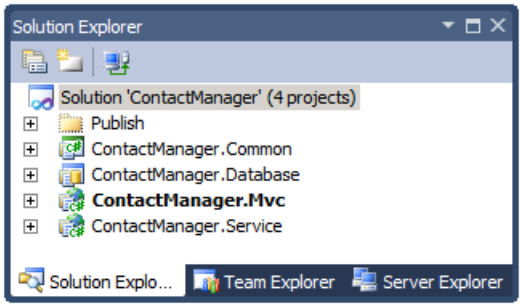
- Matt Hink is a web application developer at Fabrikam, Inc. He is part of the team who developed the Contact Manager solution by using Visual Studio 2010. Matt has full administrator rights on the servers in the developer test environment, which lets him configure the environment to meet his needs. He also has user access to the Visual Studio 2010 TFS instance where he stores the source code for the Contact Manager solution.

- Rob Walters is a server administrator for the Fabrikam, Inc. development team. Rob has administrative access on the TFS server so that he can configure all aspects of TFS and Team Build. Rob also has administrative access to the test and staging web servers and acts as the database administrator (DBA) for the database servers in the test and staging environments. Rob has configured Team Build on the TFS server to carry out these tasks:
  - Build and run unit tests on the application whenever a user checks in a file to TFS. This is called CI.
  - Deploy the Contact Manager application to the test environment automatically once the application passes unit tests. This includes publishing the database to the test servers on initial deployment and any updates to the database after initial deployment.
  - Deploy the Contact Manager application to the staging environment in a single-step process.
  - Create a Web package that a Web server administrator and a DBA can use to publish the application to the production environment.
- Lisa Andrews is a server administrator responsible for deploying applications to the Fabrikam, Inc. production servers. She has read access to the share where the TFS Team Build stores the web deployment package once it builds the Contact Manager application. She also has administrative access to the production web servers so that she can deploy the application to production. Additionally, she acts as the DBA who deploys databases and database updates to the database server in the production environment.

The Contact Manager Solution

The Contact Manager solution is designed to let registered, logged-in users add and edit contact information

through a web interface. The Contact Manager solution consists of four individual projects:

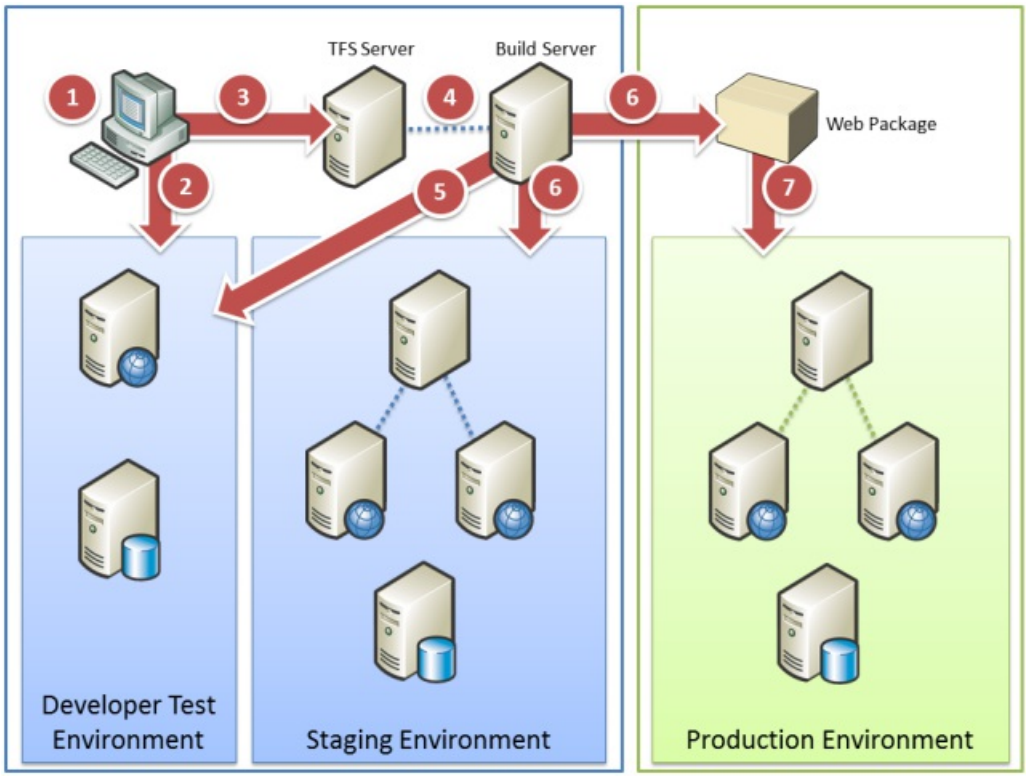


- **ContactManager.Mvc**. This is an ASP.NET MVC3 web application project that represents the entry point for the solution. It offers some basic web application functionality, like providing users with the ability to create and view contact details. The application relies on a Windows Communication Foundation (WCF) service to manage contacts and an ASP.NET application services database to manage authentication and authorization.
- **ContactManager.Database**. This is a Visual Studio 2010 database project. The project defines the schema for a database that stores contact details.
- **ContactManager.Service**. This is a WCF web service project. The WCF exposes an endpoint that allows callers to perform create, retrieve, update, and delete (CRUD) operations on the Contact Manager database. The service relies on the Contact Manager database and the ContactManager.Common.dll assembly.
- **ContactManager.Common**. This is a class library project. The WCF service relies on types defined in this assembly.

A complete review of the solution and its deployment requirements is provided in the first tutorial in this series, Web Deployment in the Enterprise.

Deployment Tasks

There are several distinct tasks involved in deploying applications to different environments in a large organization. These are the key tasks that the tutorials cover:



Here is a list of each step in the deployment process from the perspective of the users described earlier in this document:

1. All members of the team review the Contact Manager solution in Visual Studio 2010 to determine key deployment requirements and issues.
2. Matt Hink may deploy the Contact Manager solution directly from the developer workstation to the developer test environment, to conduct an initial test of the deployment logic.
3. Matt Hink adds the application to source control in TFS.
4. Rob Walters creates various build definitions for the Contact Manager solution in Team Build. One build definition uses CI to deploy the solution to the developer test environment whenever a user checks in new code. Another build definition lets users trigger deployments to the staging environment as required.
5. Every time a user checks in new code, Team Build automatically builds the solution components, runs unit tests, and deploys the solution to the developer test environment if the build was successful and the unit tests pass.
6. When a user triggers a deployment to the staging environment, the solution is packaged and deployed in a single-step process. This process also generates a package for manual deployment to the production environment.
7. Lisa Andrews deploys the application to the production environment by manually importing the web package created in step 6.

Key Deployment Issues

The Contact Manager solution and the Fabrikam, Inc. scenario highlight various common issues and challenges that you may encounter when you deploy complex, enterprise-scale solutions. For example:

- You need to be able to deploy projects to multiple environments, like developer or test environments, staging platforms, and production servers. The solution needs to be deployed with different configuration settings for each environment.
- You need to deploy multiple dependent projects simultaneously as part of a single-step or automated build and deployment process.
- You need to be able to drive deployment from an automated process. For example, you want to use a CI process to deploy web applications to a staging environment when new code is checked in.
- You need to be able to control the deployment process and set deployment variables from outside Visual Studio, as developers are unlikely to have the correct configuration settings or the necessary credentials for every target environment.
- You need to deploy schema-based database projects and preserve existing data on subsequent deployments.
- You need to deploy membership databases on an ad hoc basis without deploying user account data. You may also need to update the schema of deployed membership databases without losing existing user account data.
- You need to exclude certain files or folders when you deploy content to various target environments.

In addition, managing deployment when updates are frequent and incremental throws up some additional challenges. For example:

- You run unit tests every time a developer checks in new code. You only want to deploy the solution if the code passes the unit tests.
- When you deploy a web application to a staging or production environment, you want to redirect users to an *app_offline.htm* file for the duration of the deployment process.
- You want to log deployment activities. The deployment process should send email notifications of successful or failed deployments to designated recipients.
- If an automated deployment fails, the deployment process should retry the current deployment or deploy the previous web package instead.

Application Lifecycle Management: From Development to Production

2/20/2020 • 12 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic illustrates how a fictional company manages the deployment of an ASP.NET web application through test, staging, and production environments as part of a continuous development process. Throughout the topic, links are provided to further information and walkthroughs on how to perform specific tasks.

The topic is designed to provide a high-level overview for a series of tutorials on web deployment in the enterprise. Don't worry if you're not familiar with some of the concepts described here—the tutorials that follow provide detailed information on all of these tasks and techniques.

> **NOTE**
>
> For the sake of simplicity, this topic doesn't discuss updating databases as part of the deployment process. However, making incremental updates to databases features is a requirement of many enterprise deployment scenarios, and you can find guidance on how to accomplish this later in this tutorial series. For more information, see Deploying Database Projects.

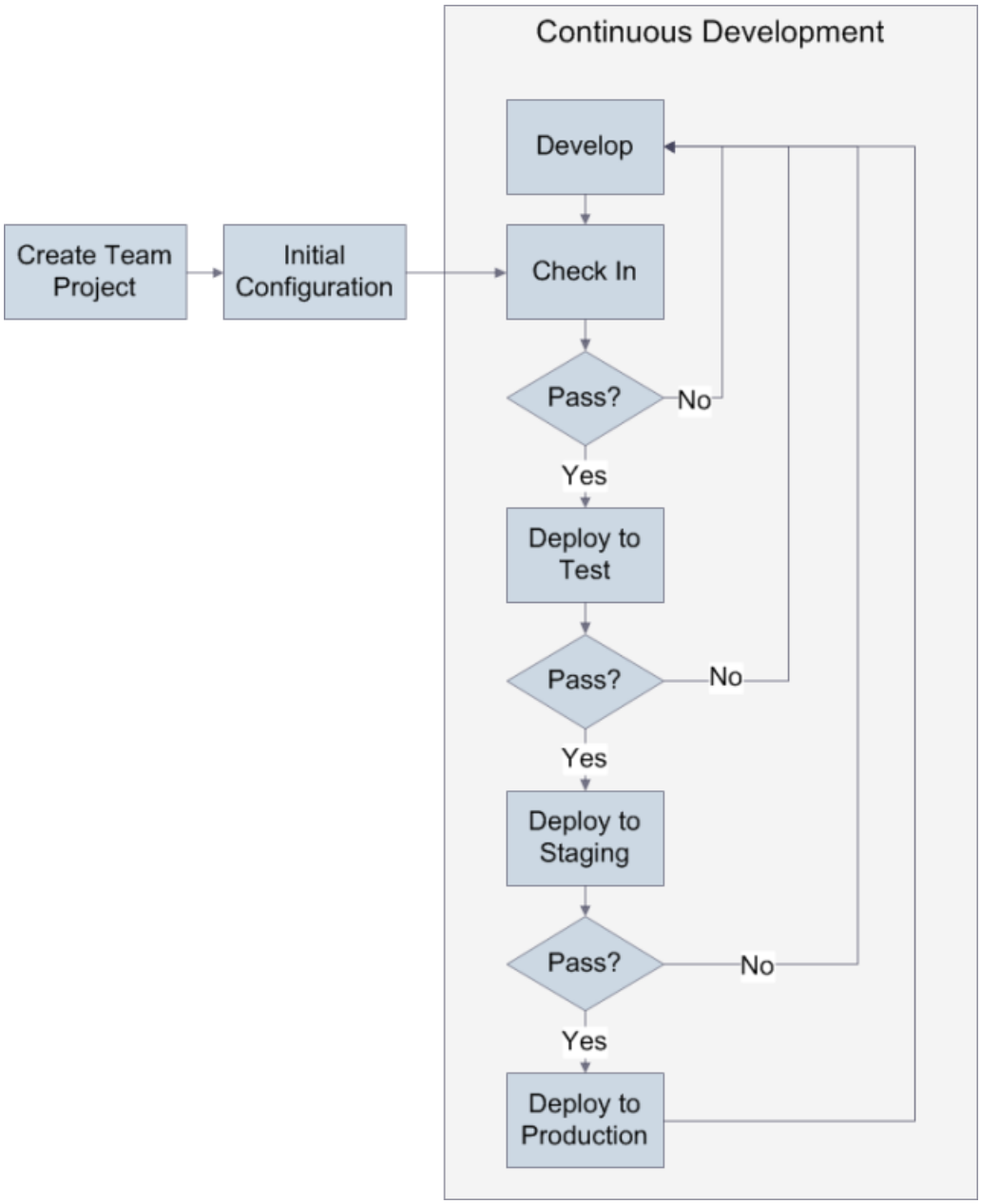
Overview

The deployment process illustrated here is based on the Fabrikam, Inc. deployment scenario described in Enterprise Web Deployment: Scenario Overview. You should read the scenario overview before you study this topic. Essentially, the scenario examines how an organization manages the deployment of a reasonably complex web application, the Contact Manager solution, through various phases in a typical enterprise environment.

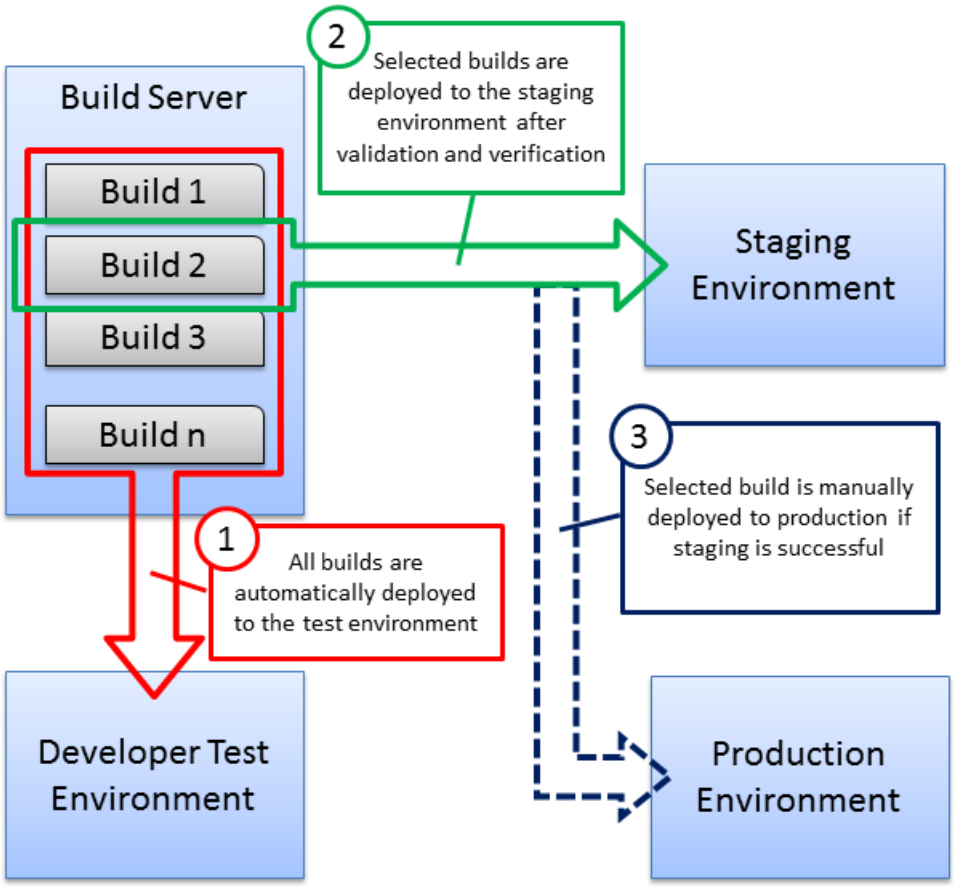
At a high level, the Contact Manager solution goes through these stages as part of the development and deployment process:

1. A developer checks some code into Team Foundation Server (TFS) 2010.
2. TFS builds the code and runs any unit tests associated with the team project.
3. TFS deploys the solution to the test environment.
4. The developer team verifies and validates the solution in the test environment.
5. The staging environment administrator performs a "what if" deployment to the staging environment, to establish whether the deployment will cause any problems.
6. The staging environment administrator performs a live deployment to the staging environment.
7. The solution undergoes user acceptance testing in the staging environment.
8. The web deployment packages are manually imported into the production environment.

These stages form part of a continuous development cycle.



In practice, the process is slightly more complicated than this, as you'll see when we look at each stage in more detail. Fabrikam, Inc. uses a different approach to deployment for each target environment.



The rest of this topic examines these key stages of this deployment lifecycle:

- **Prerequisites**: How you need to configure your server infrastructure before you put your deployment logic in place.
- **Initial development and deployment**: What you need to do before you deploy your solution for the first time.
- **Deployment to test**: How to package and deploy content to a test environment automatically when a developer checks in new code.
- **Deployment to staging**: How to deploy specific builds to a staging environment and how to perform "what if" deployments to ensure that a deployment won't cause any problems.
- **Deployment to production**: How to import web packages into a production environment when network infrastructure prevents remote deployment.

Prerequisites

The first task in any deployment scenario is to ensure that your server infrastructure meets the requirements of your deployment tools and techniques. In this case, Fabrikam, Inc. has configured its server infrastructure like this:

- TFS is configured to include a team project collection, build controllers, and build agents. See Configuring Team Foundation Server for Automated Web Deployment for more information.
- The test environment is configured to accept remote deployments using the Web Deployment Agent Service (the "remote agent"), as described in Scenario: Configuring a Test Environment for Web Deployment and Configure a Web Server for Web Deploy Publishing (Remote Agent).
- The staging environment is configured to accept remote deployments using the Web Deploy Handler endpoint, as described in Scenario: Configuring a Staging Environment for Web Deployment and Configure a Web Server for Web Deploy Publishing (Web Deploy Handler).
- The production environment is configured to allow an administrator to manually import web deployment packages into Internet Information Services (IIS), as described in Scenario: Configuring a Production Environment for Web Deployment and Configure a Web Server for Web Deploy Publishing (Offline

Deployment).

Initial Development and Deployment

Before the Fabrikam, Inc. development team can deploy the Contact Manager solution for the first time, it needs to perform these tasks:

- Create a new team project in TFS.
- Create the Microsoft Build Engine (MSBuild) project files that contain the deployment logic.
- Create the TFS build definitions that trigger the deployment processes.

Create a New Team Project

- The TFS administrator, Rob Walters, creates a new team project for the application, as described in Creating a Team Project in TFS. Next, the lead developer, Matt Hink, creates a skeleton solution. He checks his files into the new team project in TFS, as described in Adding Content to Source Control.

Create the Deployment Logic

Matt Hink creates various custom MSBuild project files, using the split project file approach described in Understanding the Project File. Matt creates:

- A project file named *Publish.proj* that runs the deployment process. This file contains MSBuild targets that build the projects in the solution, create web packages, and deploy the packages to a destination server environment.
- Environment-specific project files named *Env-Dev.proj* and *Env-Stage.proj*. These contain settings that are specific to the test environment and the staging environment respectively, like connection strings, service endpoints, and the details of the remote service that will receive the web package. For guidance on choosing the right settings for specific destination environments, see Configure Deployment Properties for a Target Environment.

To run the deployment, a user executes the *Publish.proj* file using MSBuild or Team Build and specifies the location of the relevant environment-specific project file (*Env-Dev.proj* or *Env-Stage.proj*) as a command-line argument. The *Publish.proj* file then imports the environment-specific project file to create a complete set of publishing instructions for each target environment.

> **NOTE**
>
> The way these custom project files work is independent of the mechanism you use to invoke MSBuild. For example, you can use the MSBuild command line directly, as described in Understanding the Project File. You can run the project files from a command file, as described in Create and Run a Deployment Command File. Alternatively, you can run the project files from a build definition in TFS, as described in Creating a Build Definition that Supports Deployment.
> In each case the end result is the same—MSBuild executes the merged project file and deploys your solution to the target environment. This provides you with a great deal of flexibility in how you trigger your publishing process.

Once he has created the custom project files, Matt adds them to a solution folder and checks them into source control.

Create Build Definitions

As a final preparation task, Matt and Rob work together to create three build definitions for the new team project:

- **DeployToTest**. This builds the Contact Manager solution and deploys it to the test environment every time a check-in occurs.
- **DeployToStaging**. This deploys resources from a specified previous build to the staging environment when a developer queues the build.
- **DeployToStaging-WhatIf**. This performs a "what if" deployment to the staging environment when a

developer queues the build.

The sections that follow provide more detail on each of these build definitions.

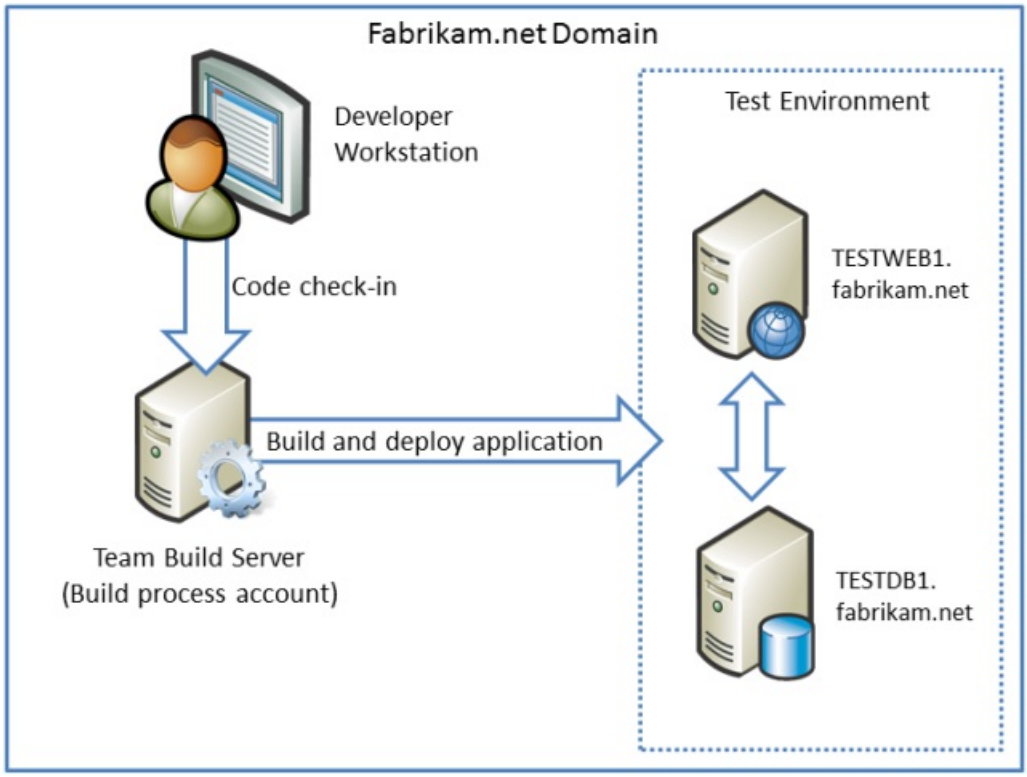
Deployment to Test

The development team at Fabrikam, Inc. maintains test environments to conduct a variety of software testing activities, like verification and validation, usability testing, compatibility testing, and ad hoc or exploratory testing.

The development team has created a build definition in TFS named **DeployToTest**. This build definition uses a continuous integration trigger, which means the build process runs every time a member of the Fabrikam, Inc. development team performs a check-in. When a build is triggered, the build definition will:

- Build the ContactManager.sln solution. This in turn builds every project within the solution.
- Run any unit tests in the solution folder structure (if the solution builds successfully).
- Run the custom project files that control the deployment process (if the solution builds successfully and passes any unit tests).

The end result is that if the solution builds successfully and passes unit tests, the web packages and any other deployment resources are deployed to the test environment.



How Does the Deployment Process Work?

The **DeployToTest** build definition supplies these arguments to MSBuild:

```
/p:DeployOnBuild=true;DeployTarget=package;TargetEnvPropsFile=[path]\Env-Dev.proj
```

The **DeployOnBuild=true** and **DeployTarget=package** properties are used when Team Build builds the projects within the solution. When the project is a web application project, these properties instruct MSBuild to create a web deployment package for the project. The **TargetEnvPropsFile** property tells the *Publish.proj* file where to find the environment-specific project file to import.

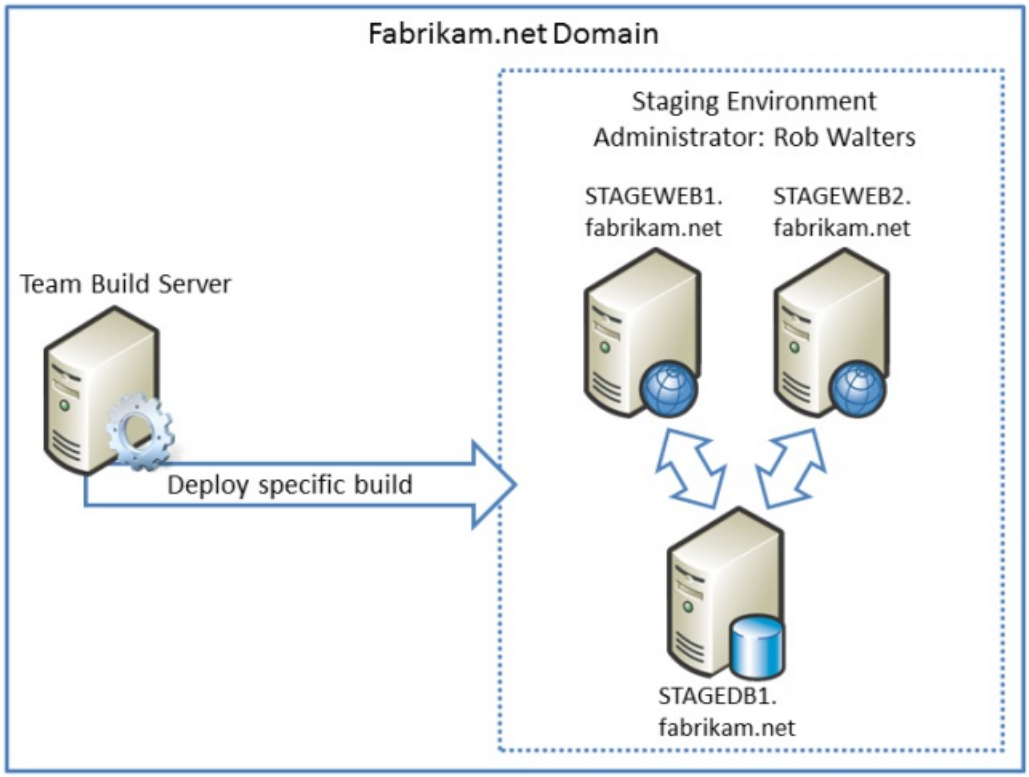
> **NOTE**
> For a detailed walkthrough on how to create a build definition like this, see Creating a Build Definition that Supports Deployment.

The *Publish.proj* file contains targets that build each project in the solution. However, it also includes conditional logic that skips these build targets if you're executing the file in Team Build. This lets you take advantage of the additional build functionality that Team Build offers, like the ability to run unit tests. If the solution build or the unit tests fail, the *Publish.proj* file will not be executed and the application will not be deployed.

The conditional logic is accomplished by evaluating the **BuildingInTeamBuild** property. This is an MSBuild property that is automatically set to **true** when you use Team Build to build your projects.

Deployment to Staging

When a build meets all of the requirements of the developer team in the test environment, the team may want to deploy the same build to a staging environment. Staging environments are typically configured to match the characteristics of the production or "live" environment as closely as possible, for example, in terms of server specifications, operating systems and software, and network configuration. Staging environments are often used for load testing, user acceptance testing, and broader internal reviews. Builds are deployed to the staging environment directly from the build server.



The build definitions used to deploy the solution to the staging environment, **DeployToStaging-WhatIf** and **DeployToStaging**, share these characteristics:

- They don't actually build anything. When Rob deploys the solution to the staging environment, he wants to deploy a specific, existing build that's already been verified and validated in the test environment. The build definitions just need to run the custom project files that control the deployment process.
- When Rob triggers a build, he uses the build parameters to specify which build contains the resources he wants to deploy from the build server.
- The build definitions are not triggered automatically. Rob manually queues a build when he wants to deploy the solution to the staging environment.

This is the high-level process for a deployment to the staging environment:

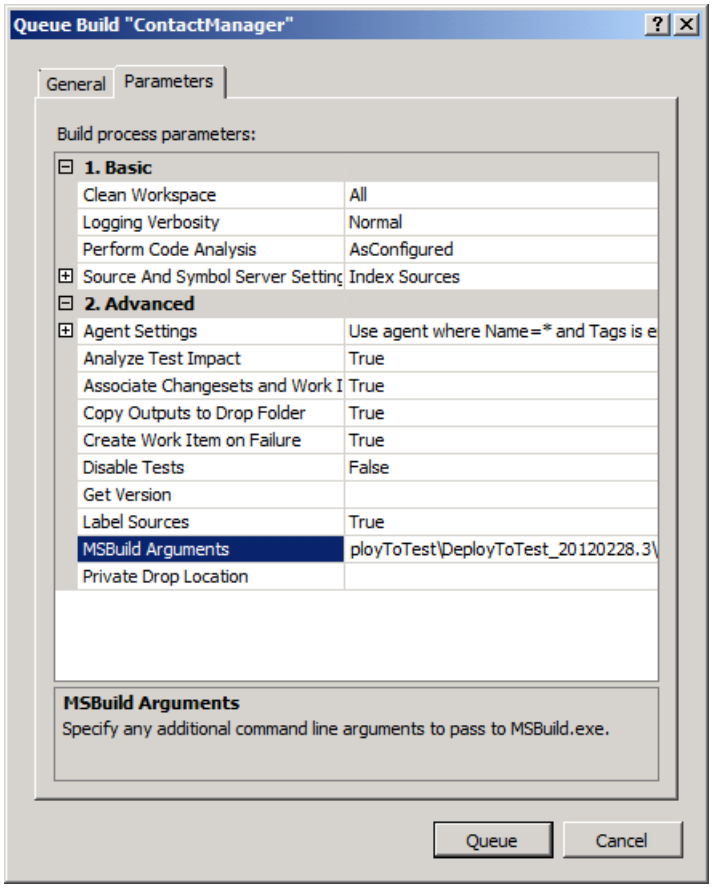
1. The staging environment administrator, Rob Walters, queues a build using the **DeployToStaging-WhatIf** build definition. Rob uses the build definition parameters to specify which build he wants to deploy.
2. The **DeployToStaging-WhatIf** build definition runs the custom project files in "what if" mode. This generates log files as if Rob was performing a live deployment, but it doesn't actually make any changes to the destination environment.
3. Rob reviews the log files to ascertain the effects of the deployment on the staging environment. In particular, Rob wants to check what will be added, what will be updated, and what will be deleted.
4. If Rob is satisfied that the deployment won't make any undesirable changes to existing resources or data, he queues a build using the **DeployToStaging** build definition.
5. The **DeployToStaging** build definition runs the custom project files. These publish the deployment resources to the primary web server in the staging environment.
6. The Web Farm Framework (WFF) controller synchronizes the web servers in the staging environment. This makes the application available on all the web servers in the server farm.

How Does the Deployment Process Work?

The **DeployToStaging** build definition supplies these arguments to MSBuild:

```
/p:TargetEnvPropsFile=[path]\Env-Stage.proj;OutputRoot=[path to build folder]
```

The **TargetEnvPropsFile** property tells the *Publish.proj* file where to find the environment-specific project file to import. The **OutputRoot** property overrides the built-in value and indicates the location of the build folder that contains the resources you want to deploy. When Rob queues the build, he uses the **Parameters** tab to provide an updated value for the **OutputRoot** property.



> **NOTE**
> For more information on how to create a build definition like this, see Deploy a Specific Build.

The **DeployToStaging-WhatIf** build definition contains the same deployment logic as the **DeployToStaging** build definition. However, it includes the additional argument **WhatIf=true**:

```
/p:TargetEnvPropsFile=[path]\Env-Stage.proj;
   OutputRoot=[path to build folder];
   WhatIf=true
```

Within the *Publish.proj* file, the **WhatIf** property indicates that all deployment resources should be published in "what if" mode. In other words, log files are generated as if the deployment had gone ahead, but nothing is actually changed in the destination environment. This lets you evaluate the impact of a proposed deployment—in particular, what will get added, what will get updated, and what will get deleted—before you actually make any changes.

> **NOTE**
> For more information on how to configure "what if" deployments, see Performing a "What If" Deployment.

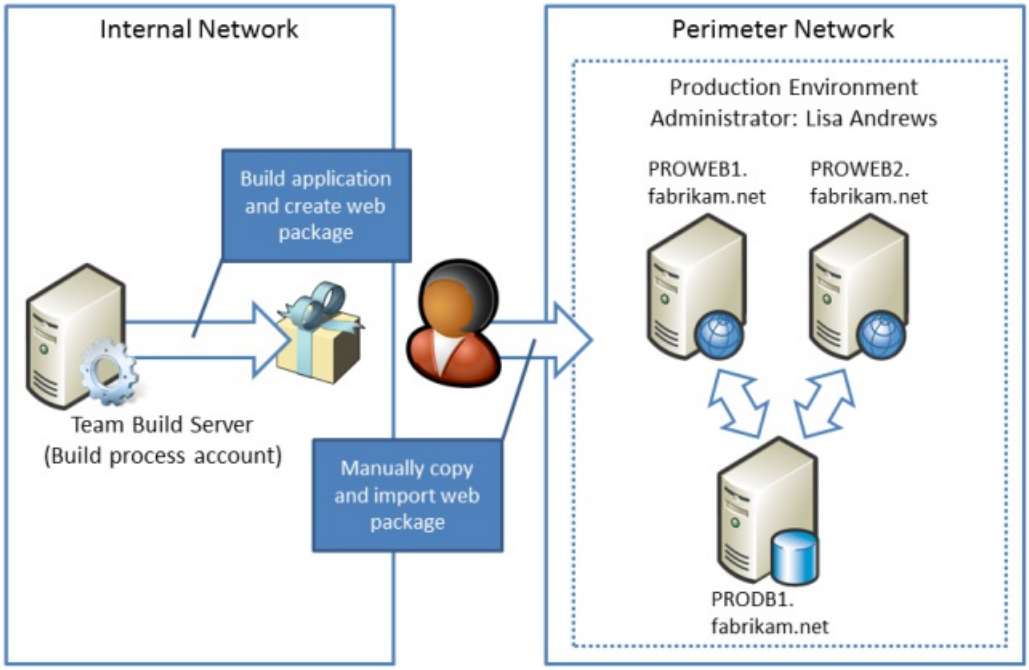
Once you've deployed your application to the primary web server in the staging environment, the WFF will automatically synchronize the application across all the servers in the server farm.

> **NOTE**
> For more information on configuring the WFF to synchronize web servers, see Create a Server Farm with the Web Farm Framework.

Deployment to Production

When a build has been approved in the staging environment, the Fabrikam, Inc. team can publish the application to the production environment. The production environment is where the application goes "live" and reaches its target audience of end users.

The production environment is in an Internet-facing perimeter network. This is isolated from the internal network that contains the build server. The production environment administrator, Lisa Andrews, must manually copy the web deployment packages from the build server and import them into IIS on the primary production web server.



This is the high-level process for a deployment to the production environment:

1. The developer team advises Lisa that a build is ready for deployment to production. The team advises Lisa of the location of the web deployment packages within the drop folder on the build server.
2. Lisa collects the web packages from the build server and copies them to the primary web server in the production environment.
3. Lisa uses IIS Manager to import and publish the web packages on the primary web server.
4. The WFF controller synchronizes the web servers in the production environment. This makes the application available on all the web servers in the server farm.

How Does the Deployment Process Work?

IIS Manager includes an Import Application Package Wizard that makes it easy to publish web packages to an IIS website. For a walkthrough on how to perform this procedure, see Manually Installing Web Packages.

Conclusion

This topic provided an illustration of the deployment lifecycle for a typical enterprise-scale web application.

This topic forms part of a series of tutorials that provide guidance on various aspects of web application deployment. In practice, there are lots of additional tasks and considerations at each stage of the deployment process, and it's not possible to cover them all in a single walkthrough. For more information, consult these tutorials:

- Web Deployment in the Enterprise. This tutorial provides a comprehensive introduction to web deployment techniques using MSBuild and the IIS Web Deployment Tool (Web Deploy).
- Configuring Server Environments for Web Deployment. This tutorial provides guidance on how to configure Windows server environments to support various deployment scenarios.
- Configuring Team Foundation Server for Automated Web Deployment. This tutorial provides guidance on how to integrate deployment logic into TFS build processes.
- Advanced Enterprise Web Deployment. This tutorial provides guidance on how to meet some of the more complex deployment challenges that organizations face.

ASP.NET 4 - Enterprise Deployment Series 1

2/20/2020 • 2 minutes to read • Edit Online

This tutorial describes how to meet lots of the challenges you'll encounter when you manage the deployment of enterprise-scale web applications to development, test, staging, and production environments. The tutorial includes a reference solution together with a mixture of conceptual and task-oriented content to guide you through various common tasks and procedures.

- Web Deployment in the Enterprise
- The Contact Manager Solution
- Setting Up the Contact Manager Solution
- Understanding the Project File
- Understanding the Build Process
- Building and Packaging Web Application Projects
- Configuring Parameters for Web Package Deployment
- Deploying Web Packages
- Deploying Database Projects
- Creating and Running a Deployment Command File
- Manually Installing Web Packages

Web Deployment in the Enterprise

2/20/2020 • 5 minutes to read • Edit Online

by Jason Lee

Download PDF

This tutorial describes how to meet lots of the challenges you'll encounter when you manage the deployment of enterprise-scale web applications to development, test, staging, and production environments. The tutorial includes a reference solution together with a mixture of conceptual and task-oriented content to guide you through various common tasks and procedures.

For an Italian translation of these tutorials, visit http://www.lucamorelli.it.

Enterprise Deployment Challenges

Organizations often encounter these challenges when they look to manage the deployment of complex, enterprise-scale solutions:

- You need to be able to deploy projects to multiple environments, like developer or test environments, staging platforms, and production servers. The solution needs to be deployed with different configuration settings for each environment.
- You need to deploy multiple dependent projects simultaneously as part of a single-step or automated build and deployment process.
- You need to be able to drive deployment from an automated process. For example, you want to use a continuous integration (CI) process to deploy web applications to a test environment when new code is checked in.
- You need to be able to control the deployment process and set deployment variables from outside Visual Studio, as developers are unlikely to have the correct configuration settings or the necessary credentials for every target environment.
- You need to deploy schema-based database projects and preserve existing data on subsequent deployments.
- You need to deploy membership databases on an ad hoc basis without deploying user account data. You may also need to update the schema of deployed membership databases without losing existing user account data.
- You need to exclude certain files or folders when you deploy content to various target environments.

Overview of Approach

This tutorial, together with the other tutorials in this series, uses this high-level approach to meet the challenges described above.

- **Use custom Microsoft Build Engine (MSBuild) project files to control the overall build and deployment process.**
- This lets you build and deploy every project in the solution as part of a single, scriptable operation.
- Environment-specific settings are configured using simple environment-specific project files. In contrast to the Visual Studio–centric approach of using solution configurations and publish profiles to configure deployments for different environments, this approach lets you configure and manage the deployment process from outside Visual Studio. This means that developers don't need advance knowledge of connection strings, service endpoints, server credentials, and other deployment variables for destination environments.

- The custom project files can be invoked by Team Build as part of a Team Foundation Server (TFS) workflow. This lets you configure automated deployment for CI scenarios.

Use the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) to package and deploy web application projects.

- Web Deploy provides a framework that lets you package and deploy your web application content to a destination IIS web server, together with dependencies, configuration settings, security settings, and any other requirements.
- You can control the entire packaging and deployment process from within your custom MSBuild project files. You can also manipulate the configuration settings that accompany your web deployment package, like connection strings, service endpoints, and IIS destination details.
- Web Deploy, together with the Web Publishing Pipeline, offers lots of extensibility points that let you customize your deployments. For example, it's easy to exclude unwanted files and folders from your web deployment packages.

Use the VSDBCMD.exe utility to deploy and update database schemas.

- VSDBCMD allows you to deploy databases from a database schema file (.dbschema), which is generated when you build a Visual Studio database project. In contrast, the database deployment functionality included in Web Deploy is more suited to deploying existing databases from a local SQL Server instance.
- Unlike Visual Studio's functionality for deploying database projects, VSDBCMD lets you deploy differential updates to an existing target database. This allows you to preserve any existing data while you upgrade the database schema.
- You can execute VSDBCMD commands from within your custom MSBuild project files.

Content Map

This tutorial includes topics that fall into four main areas.

These topics introduce the reference solution—the Contact Manager solution—and describe how to download it and configure it on your local machine:

- The Contact Manager Solution
- Setting Up the Contact Manager Solution

These topics introduce MSBuild project files, describe how you can create and use custom project files, and walk through the deployment process for the Contact Manager solution:

- Understanding the Project File
- Understanding the Build Process

These topics describe web application deployment, including how the build and packaging process works, how the build process integrates with the Web Publishing Pipeline, how to modify deployment parameters, and how to deploy web packages to destination environments:

- Building and Packaging Web Application Projects
- Configuring Parameters for Web Package Deployment
- Deploying Web Packages
- Deploying Database Projects describes the different techniques you can use to deploy Visual Studio database projects, together with the advantages and disadvantages of each approach. Creating and Running a Deployment Command File describes how to create a simple command file that encapsulates your deployment logic and lets you deploy complex solutions as a single-step process.

- Finally, Manually Installing Web Packages concludes the tutorial by showing you to import web packages into IIS.

Key Technologies

The topics in this tutorial primarily use these technologies to manage build and deployment:

- Visual Studio 2010
- MSBuild
- IIS 7.5
- Web Deploy 2.0
- The VSDBCMD.exe database deployment utility

Other Tutorials in This Series

This forms part of a series of five tutorials on enterprise-scale web deployment. These are the other tutorials in the series:

- Deploying Web Applications in Enterprise Scenarios. This introductory content provides the contextual background for the tutorial series. It describes the tutorial scenario, and it illustrates how the tasks and walkthroughs described throughout the series fit into a broader Application Lifecycle Management (ALM) process.
- Configuring Server Environments for Web Deployment. This tutorial describes how to configure Windows servers to support various deployment scenarios, including remote web package deployment using the Web Deployment Agent Service (the remote agent) or the Web Deploy Handler and remote database deployment. It provides guidance on choosing the appropriate deployment method for your own environment, and it describes how to use the Web Farm Framework (WFF) to replicate deployed web applications across all the web servers in a server farm.
- Configuring Team Foundation Server for Web Deployment. This tutorial describes how to configure TFS to support various deployment scenarios, including automated deployment as part of a CI process and manually triggered deployments of specific builds.
- Advanced Enterprise Web Deployment. This tutorial describes how to accomplish various more advanced deployment tasks, like customizing database deployments for multiple environments, excluding files and folders from deployment, and taking web applications offline during the deployment process.

NEXT

The Contact Manager Solution

2/20/2020 • 2 minutes to read • Edit Online

by Jason Lee

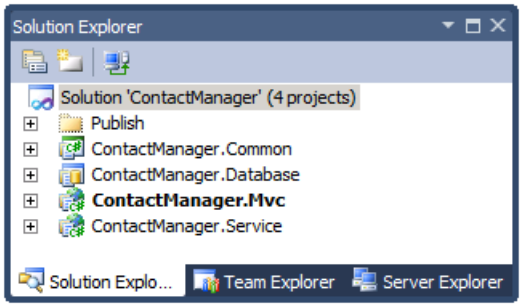
Download PDF

This series of tutorials uses a sample solution—the Contact Manager solution—to represent an enterprise-scale application with a realistic level of complexity. This topic introduces the Contact Manager solution, describes the key components of the solution, and identifies the challenges in deploying this kind of application to various destination platforms in an enterprise environment.

As you work through the topics in these tutorials, you can use the Contact Manager solution as a reference implementation that demonstrates how you can meet specific challenges in enterprise deployment scenarios. The next topic, Setting Up the Contact Manager Solution, describes how to download and run the solution on your developer workstation.

Solution Overview

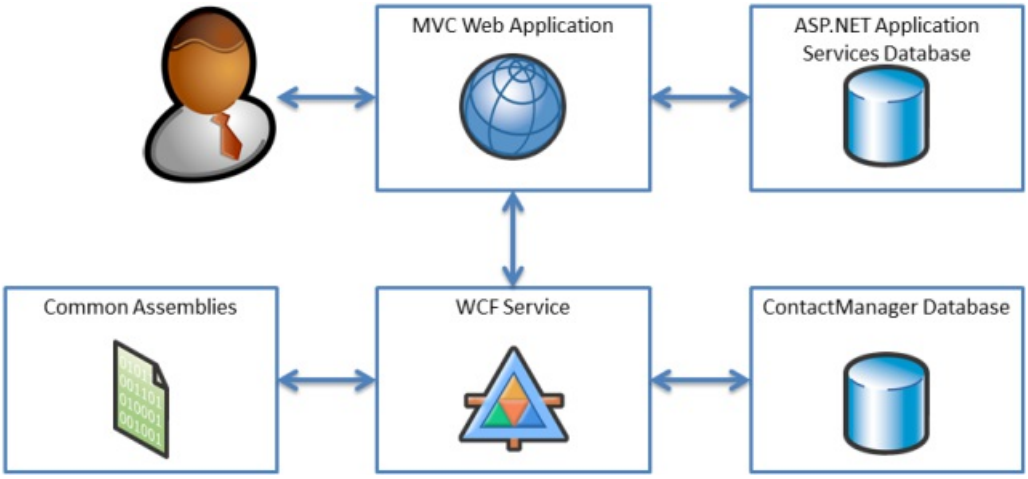
The Contact Manager solution consists of four individual projects:



- **ContactManager.Mvc**. This is an ASP.NET MVC 3 web application project that represents the entry point for the solution. It offers some basic web application functionality, like providing users with the ability to create and view contact details. The application relies on a Windows Communication Foundation (WCF) service to manage contacts and an ASP.NET application services database to manage authentication and authorization.
- **ContactManager.Database**. This is a Visual Studio database project. The project defines the schema for a database that stores contact details.
- **ContactManager.Service**. This is a WCF web service project. The WCF service exposes an endpoint that allows callers to perform create, retrieve, update, and delete (CRUD) operations on the **ContactManager** database. The service relies on the **ContactManager** database and the **ContactManager.Common.dll** assembly.
- **ContactManager.Common**. This is a class library project. The WCF service relies on types defined in this assembly.

The solution also includes a solution folder named Publish. This contains various custom project files and command files that demonstrate how you can control and manipulate the build and deployment process. These are covered in more detail later in this tutorial.

At a conceptual level, the components of the solution fit together like this:



NOTE

While the ASP.NET MVC 3 web application uses the ASP.NET membership provider, all the pages within the web application allow anonymous access. This is clearly not a realistic configuration. However, the solution is set up in this way to make it easier for you to deploy and test the solution without configuring user accounts and roles.

Deployment Challenges

The Contact Manager solution illustrates several deployment challenges that are common to lots of enterprise deployment scenarios:

- The solution consists of multiple dependent projects. You need to deploy these projects simultaneously.
- Connection strings and service endpoints need to be updated for each environment, and in a lot of cases this information will not be available to the developer.
- When you deploy the **ContactManager** database to staging and production environments, you need to preserve existing data on subsequent deployments.
- When you deploy the ASP.NET application services database, you need to deploy some configuration data but omit any user account data.
- The projects include some files and folders that should not be deployed. You need to exclude these files and folders from the deployment process.
- The solution needs to support automated deployment from a Team Foundation Server (TFS) build server.

Conclusion

This topic provided a high-level overview of the Contact Manager solution and identified some of the inherent deployment challenges that are common to lots of enterprise deployment scenarios. The remaining topics in this tutorial describe some of the techniques you can use to meet these challenges.

The next topic, Setting Up the Contact Manager Solution, describes how to download and run the solution on your developer workstation.

PREVIOUS NEXT

Setting Up the Contact Manager Solution

2/20/2020 • 4 minutes to read • Edit Online

by Jason Lee

Download PDF

> This topic describes how to download and configure the Contact Manager solution to run locally on a developer workstation.

System Requirements

To run the Contact Manager solution locally and to perform the other tasks described in this tutorial, you'll need to install this software on your developer workstation:

- Visual Studio 2010 Service Pack 1, Premium or Ultimate Edition
- Internet Information Services (IIS) 7.5 Express
- SQL Server Express 2008 R2
- IIS Web Deployment Tool (Web Deploy) 2.1 or later
- ASP.NET 4.0
- ASP.NET MVC 3
- .NET Framework 4
- .NET Framework 3.5 SP1

With the exception of Visual Studio 2010, you can download and install the latest versions of all of these products and components through the Web Platform Installer.

Download and Extract the Solution

You can download the Contact Manager sample application from the MSDN Code Gallery here.

Configure and Run the Solution

To configure and run the Contact Manager solution on your local machine, you'll need to perform these high-level steps:

1. If you don't have one already, create a local ASP.NET application services database with the membership and role management features enabled.
2. Edit connection strings in the *web.config* files to point to your local SQL Server Express instance.
3. Run the solution from Visual Studio 2010.

The remainder of this section provides more guidance on how to complete each of these tasks.

To create the application services database

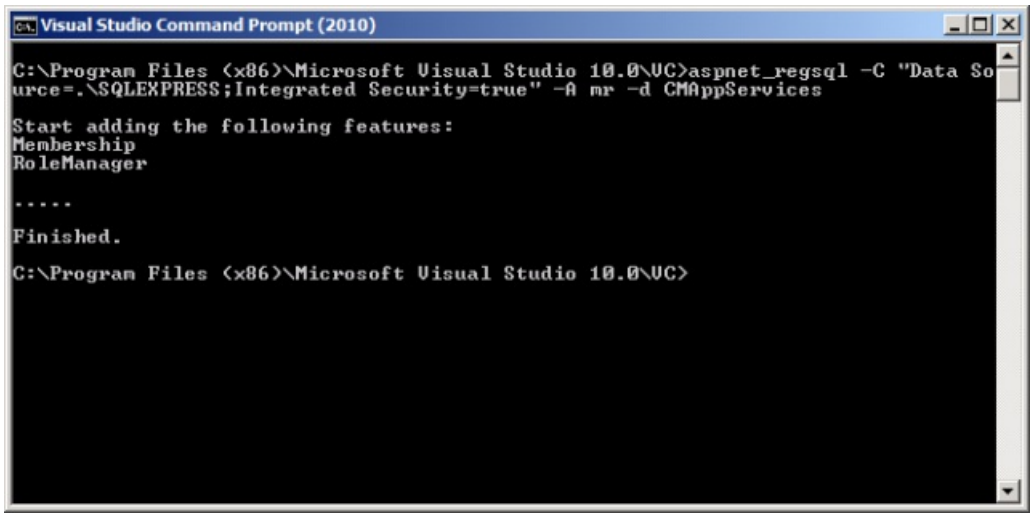
1. Open a Visual Studio 2010 command prompt. To do this, on the **Start** menu, point to **All Programs**, click **Microsoft Visual Studio 2010**, click **Visual Studio Tools**, and then click **Visual Studio Command Prompt (2010)**.

2. At the command prompt, type this command, and then press Enter:

```
aspnet_regsql -C "Data Source=.\SQLEXPRESS;Integrated Security=true" -A mr -d CMAppServices
```

a. Use the **–C** switch to specify the connection string for your database server.
b. Use the **–A** switch to specify the application services features you want to add to the database. In this case, **m** indicates that you want to add support for the membership provider and **r** indicates that you want to add support for the role manager.
c. Use the **–d** switch to specify a name for your application services database. If you omit this switch, the utility will create a database with the default name of **aspnetdb**.

3. When the database has been created successfully, the command prompt will show a confirmation.



> **NOTE**
> For more information on the aspnet_regsql utility, see ASP.NET SQL Server Registration Tool (Aspnet_regsql.exe).

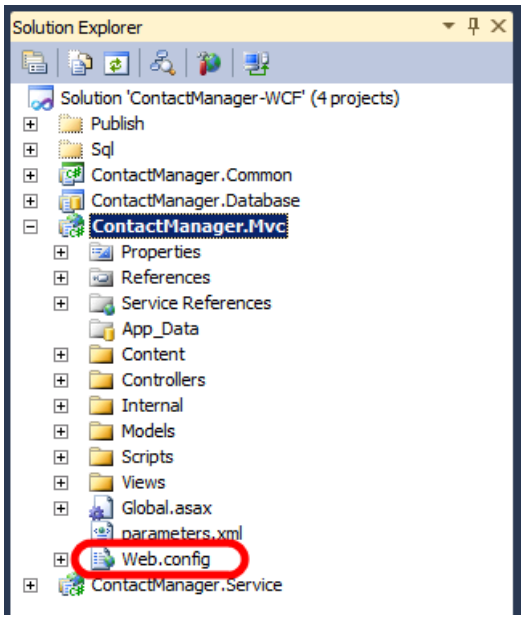
The next step is to make sure that the connection strings in the Contact Manager solution point to your local instance of SQL Server Express.

To update the connection strings

1. Open the Contact Manager solution in Visual Studio 2010.

2. In the **Solution Explorer** window, expand the **ContactManager.Mvc** project, and then double-click the **Web.config** node.

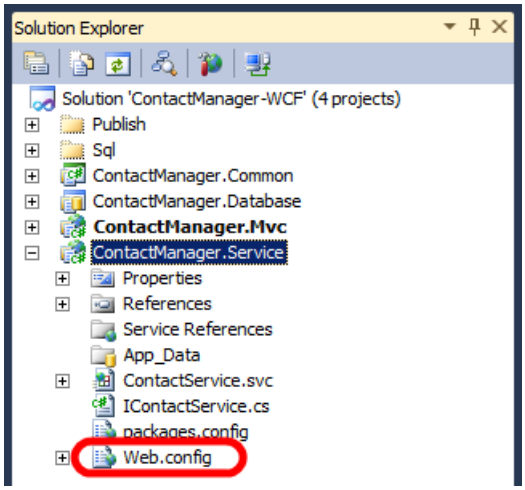
> **NOTE**
> The ContactManager.Mvc project includes two *web.config* files. You need to edit the project-level file.



3. In the **connectionStrings** element, verify that the connection string named **ApplicationServices** points to your local ASP.NET application services database.

```
<connectionStrings>
  <add name="ApplicationServices"
       connectionString="Data Source=.\SQLEXPRESS;
                         Integrated Security=true;
                         Initial Catalog=CMAppServices"
       providerName="System.Data.SqlClient" />
</connectionStrings>
```

4. In the **Solution Explorer** window, expand the **ContactManager.Service** project, and then double-click the **Web.config** node.



5. In the **connectionStrings** element, in the connection string named **ContactManagerContext**, verify that the **Data Source** property is set to your local instance of SQL Server Express. You don't need to change anything else in the connection string.

```
<connectionStrings>
  <add name="ContactManagerContext"
       connectionString="Data Source=.\SQLExpress;
                         Initial Catalog=ContactManager;
                         Integrated Security=true;
                         multipleactiveresultsets=true"
       providerName="System.Data.SqlClient" />
</connectionStrings>
```

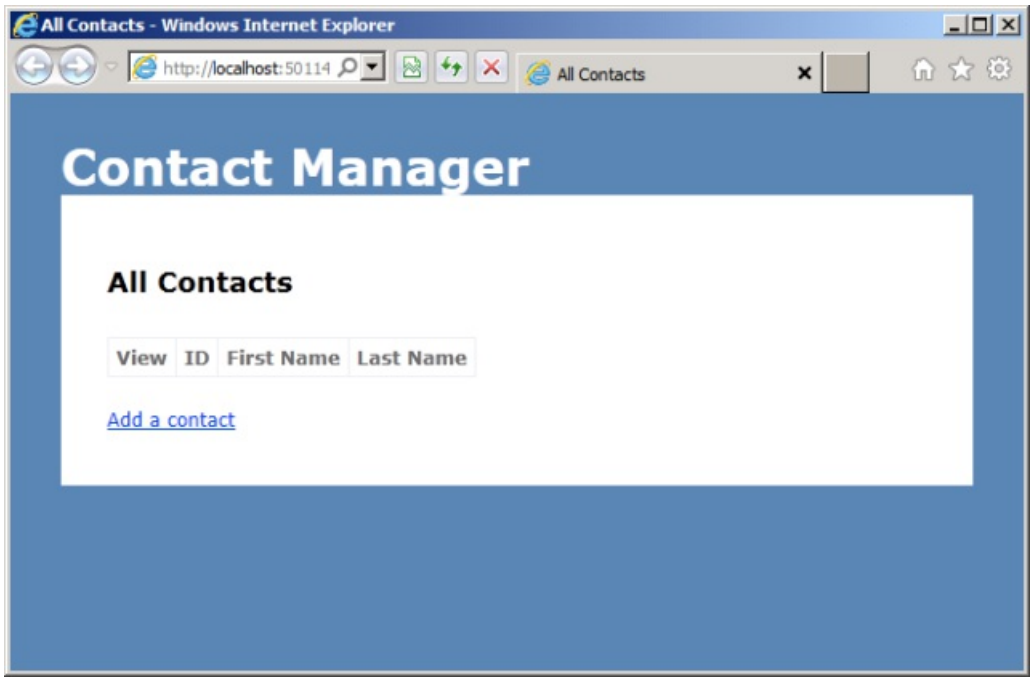
6. Save all open files.

You should now be ready to run the Contact Manager solution on your local machine.

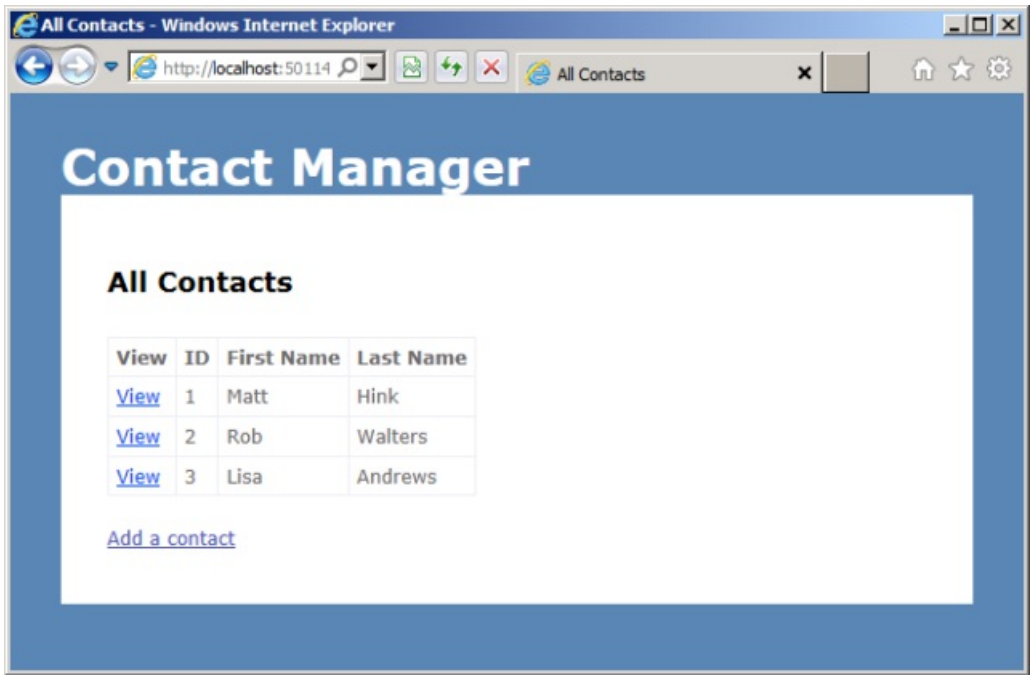
> **NOTE**
> If you follow these steps without first creating an application services database, ASP.NET will create the database the first time you attempt to create a user. However, manually creating the database gives you a lot more control over the application services feature set you want to support.

To run the Contact Manager solution

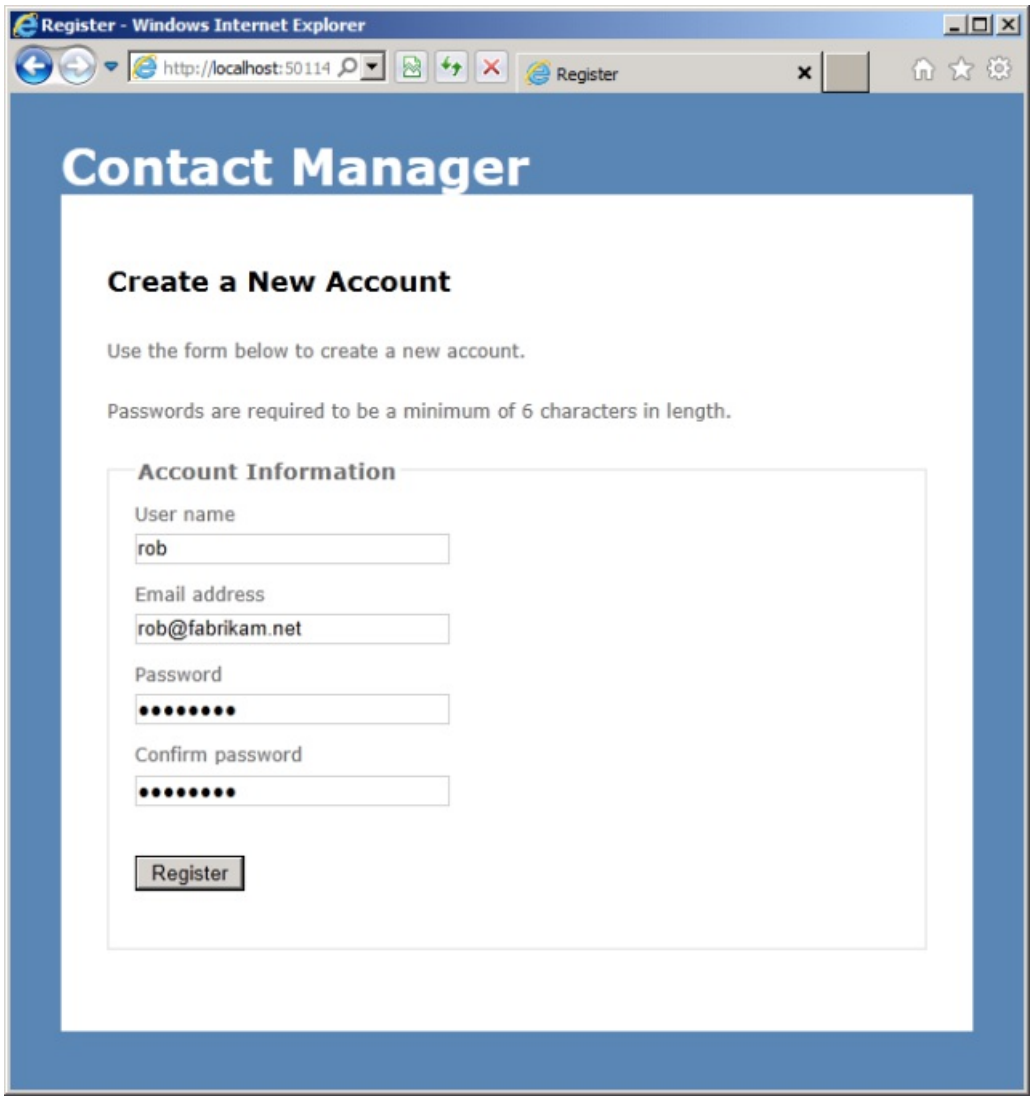
1. In Visual Studio 2010, press F5.
2. Internet Explorer starts up and requests the URL of the Contact Manager ASP.NET MVC 3 application. By default, the application displays the **All Contacts** page.



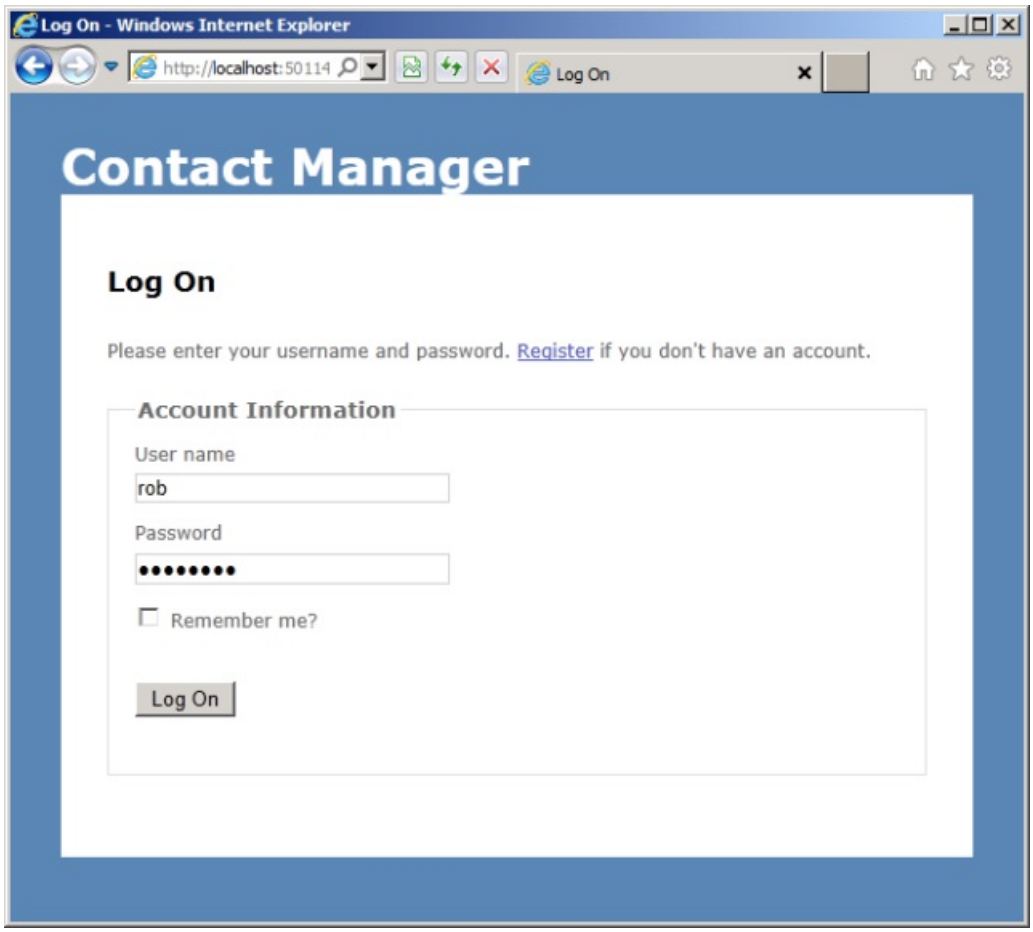
3. Add a few contacts, and then verify that the application works as expected.



4. Browse to `http://localhost:50114/Account/Register` (adjust the URL if you're hosting the application on a different port). Add a user name, email address, and password, and verify that you're able to register an account successfully.



5. Browse to `http://localhost:50114/Account/LogOn` (adjust the URL if you're hosting the application on a different port). Verify that you're able to log on using the account you just created.



6. Close Internet Explorer to stop debugging.

Conclusion

At this point, the Contact Manager solution should be fully configured to run on your local machine. You can use the solution as a reference when you work through the other topics in this tutorial.

The next topic, Understanding the Project File, explains how you can use the custom Microsoft Build Engine (MSBuild) project files within the Contact Manager solution to control the deployment process.

PREVIOUS NEXT

Understanding the project file

2/20/2020 • 14 minutes to read • Edit Online

by Jason Lee

Download PDF

Microsoft Build Engine (MSBuild) project files lie at the heart of the build and deployment process. This topic starts with a conceptual overview of MSBuild and the project file. It describes the key components you'll come across when you work with project files, and it works through an example of how you can use project files to deploy real-world applications.

What you'll learn:

- How MSBuild uses MSBuild project files to build projects.
- How MSBuild integrates with deployment technologies, like the Internet Information Services (IIS) Web Deployment Tool (Web Deploy).
- How to understand the key components of a project file.
- How you can use project files to build and deploy complex applications.

MSBuild and the Project File

When you create and build solutions in Visual Studio, Visual Studio uses MSBuild to build each project in your solution. Every Visual Studio project includes an MSBuild project file, with a file extension that reflects the type of project—for example, a C# project (.csproj), a Visual Basic.NET project (.vbproj), or a database project (.dbproj). In order to build a project, MSBuild must process the project file associated with the project. The project file is an XML document that contains all the information and instructions that MSBuild needs in order to build your project, like the content to include, the platform requirements, versioning information, web server or database server settings, and the tasks that must be performed.

MSBuild project files are based on the MSBuild XML schema, and as a result the build process is entirely open and transparent. In addition, you don't need to install Visual Studio in order to use the MSBuild engine—the MSBuild.exe executable is part of the .NET Framework, and you can run it from a command prompt. As a developer, you can craft your own MSBuild project files, using the MSBuild XML schema, to impose sophisticated and fine-grained control over how your projects are built and deployed. These custom project files work in exactly the same way as the project files that Visual Studio generates automatically.

> **NOTE**
>
> You can also use MSBuild project files with the Team Build service in Team Foundation Server (TFS). For example, you can use project files in continuous integration (CI) scenarios to automate deployment to a test environment when new code is checked in. For more information, see Configuring Team Foundation Server for Automated Web Deployment.

Project File Naming Conventions

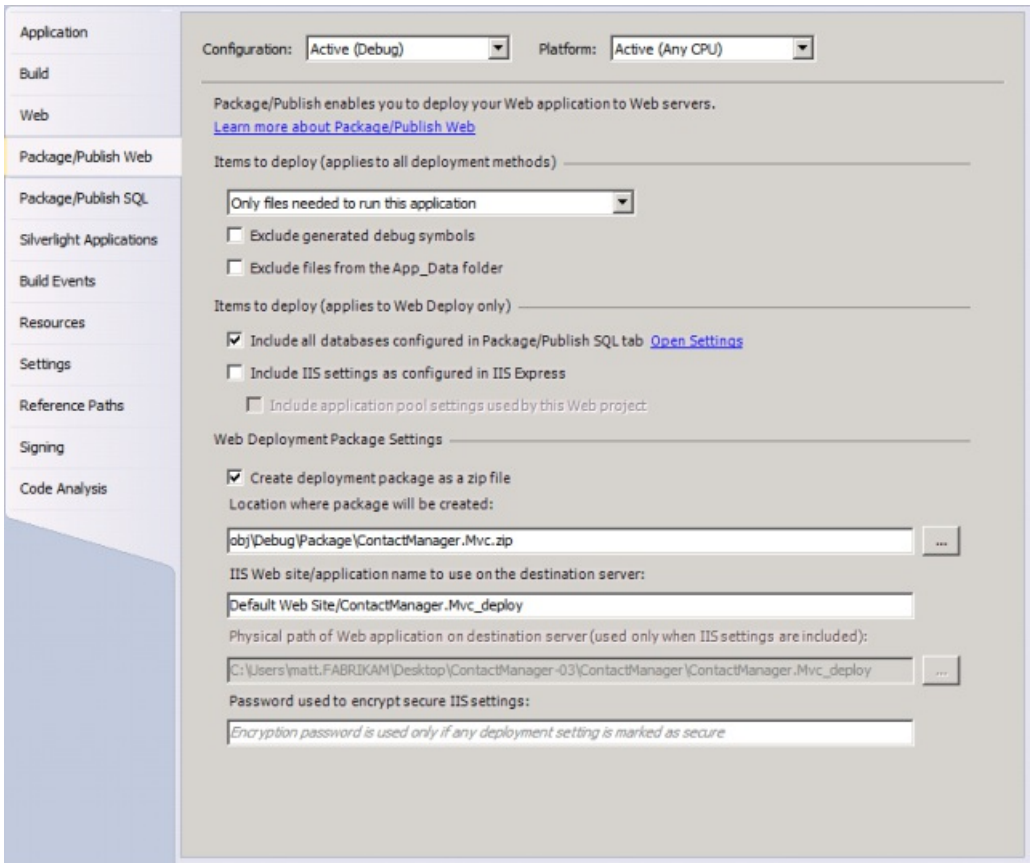
When you create your own project files, you can use any file extension you like. However, to make your solutions easier for others to understand, you should use these common conventions:

- Use the .proj extension when you create a project file that builds projects.
- Use the .targets extension when you create a reusable project file to import into other project files. Files

with a .targets extension typically don't build anything themselves, they simply contain instructions that you can import into your .proj files.

Integration with Deployment Technologies

If you've worked with web application projects in Visual Studio 2010, like ASP.NET web applications and ASP.NET MVC web applications, you'll know that these projects include built-in support for packaging and deploying the web application to a target environment. The **Properties** pages for these projects include **Package/Publish Web** and **Package/Publish SQL** tabs that you can use to configure how the components of your application are packaged and deployed. This shows the **Package/Publish Web** tab:



The underlying technology behind these capabilities is known as the Web Publishing Pipeline (WPP). The WPP essentially brings MSBuild and Web Deploy together to provide a complete build, package, and deployment process for your web applications.

The good news is that you can take advantage of the integration points that the WPP provides when you create custom project files for web projects. You can include deployment instructions in your project file, which allows you to build your projects, create web deployment packages, and install these packages on remote servers through a single project file and a single call to MSBuild. You can also call any other executables as part of your build process. For example, you can run the VSDBCMD.exe command-line tool to deploy a database from a schema file. Over the course of this topic, you'll see how you can take advantage of these capabilities to meet the requirements of your enterprise deployment scenarios.

> **NOTE**
>
> For more information on how the web application deployment process works, see ASP.NET Web Application Project Deployment Overview.

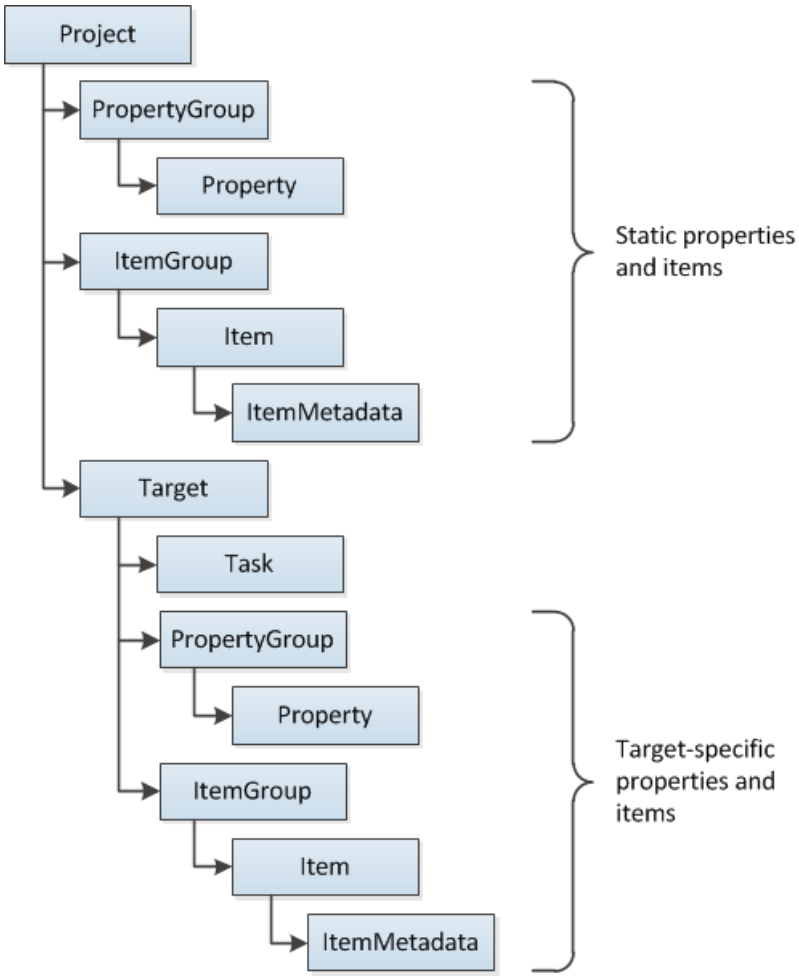
The Anatomy of a Project File

Before you look at the build process in more detail, it's worth taking a few moments to familiarize yourself with the basic structure of an MSBuild project file. This section provides an overview of the more common

elements that you'll encounter when you review, edit, or create a project file. In particular, you'll learn:

- How to use *properties* to manage variables for the build process.
- How to use *items* to identify the inputs to the build process, like code files.
- How to use *targets* and *tasks* to provide execution instructions to MSBuild, using *properties* and *items* defined elsewhere in the project file.

This shows the relationship between the key elements in an MSBuild project file:



The Project Element

The Project element is the root element of every project file. In addition to identifying the XML schema for the project file, the **Project** element can include attributes to specify the entry points for the build process. For example, in the Contact Manager sample solution, the *Publish.proj* file specifies that the build should start by calling the target named **FullPublish**.

```
<Project ToolsVersion="4.0" DefaultTargets="FullPublish"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

</Project>
```

Properties and Conditions

A project file typically needs to provide lots of different pieces of information in order to successfully build and deploy your projects. These pieces of information could include server names, connection strings, credentials, build configurations, source and destination file paths, and any other information you want to include to support customization. In a project file, properties must be defined within a PropertyGroup element. MSBuild properties consist of key-value pairs. Within the **PropertyGroup** element, the element name defines the property key and the content of the element defines the property value. For example, you could define properties named **ServerName** and **ConnectionString** to store a static server name and

connection string.

```
<PropertyGroup>
   <ServerName>FABRIKAM\TEST1</ServerName>
   <ConnectionString>
     Data Source=FABRIKAM\TESTDB;InitialCatalog=ContactManager,...
   </ConnectionString>
</PropertyGroup>
```

To retrieve a property value, use the format *$(PropertyName)*. For example, to retrieve the value of the **ServerName** property, you would type:

```
$(ServerName)
```

> **NOTE**
>
> You'll see examples of how and when to use property values later in this topic.

Embedding information as static properties in a project file is not always the ideal approach to managing the build process. In a lot of scenarios, you'll want to obtain the information from other sources or empower the user to provide the information from the command prompt. MSBuild allows you to specify any property value as a command-line parameter. For example, the user could provide a value for **ServerName** when he or she runs MSBuild.exe from the command line.

```
msbuild.exe Publish.proj /p:ServerName=FABRIKAM\TESTWEB1
```

> **NOTE**
>
> For more information on the arguments and switches you can use with MSBuild.exe, see MSBuild Command Line Reference.

You can use the same property syntax to obtain the values of environment variables and built-in project properties. Lots of commonly used properties are defined for you, and you can use them in your project files by including the relevant parameter name. For example, to retrieve the current project platform—for example, **x86** or **AnyCpu**—you can include the **$(Platform)** property reference in your project file. For more information, see Macros for Build Commands and Properties, Common MSBuild Project Properties, and Reserved Properties.

Properties are often used in conjunction with *conditions*. Most MSBuild elements support the **Condition** attribute, which lets you specify the criteria upon which MSBuild should evaluate the element. For example, consider this property definition:

```
<PropertyGroup>
   <OutputRoot Condition=" '$(OutputRoot)'=='' ">..\Publish\Out\</OutputRoot>
   ...
</PropertyGroup>
```

When MSBuild processes this property definition, it first checks to see whether an **$(OutputRoot)** property value is available. If the property value is blank—in other words, the user hasn't provided a value for this property—the condition evaluates to **true** and the property value is set to **..\Publish\Out**. If the user has provided a value for this property, the condition evaluates to **false** and the static property value is not used.

For more information on the different ways in which you can specify conditions, see MSBuild Conditions.

Items and Item Groups

One of the important roles of the project file is to define the inputs to the build process. Typically, these inputs are files—code files, configuration files, command files, and any other files that you need to process or copy as part of the build process. In the MSBuild project schema, these inputs are represented by Item elements. In a project file, items must be defined within an ItemGroup element. Just like **Property** elements, you can name an **Item** element however you like. However, you must specify an **Include** attribute to identify the file or wildcard that the item represents.

```
<ItemGroup>
   <ProjectsToBuild Include="$(SourceRoot)ContactManager-WCF.sln"/>
</ItemGroup>
```

By specifying multiple **Item** elements with the same name, you're effectively creating a named list of resources. A good way to see this in action is to take a look inside one of the project files that Visual Studio creates. For example, the *ContactManager.Mvc.csproj* file in the sample solution includes a lot of item groups, each with several identically named **Item** elements.

```
<ItemGroup>
   <Reference Include="Microsoft.CSharp" />
   <Reference Include="System.Runtime.Serialization" />
   <Reference Include="System.ServiceModel" />
   ...
</ItemGroup>
<ItemGroup>
   <Compile Include="Controllers\AccountController.cs" />
   <Compile Include="Controllers\ContactsController.cs" />
   <Compile Include="Controllers\HomeController.cs" />
   ...
</ItemGroup>
<ItemGroup>
   <Content Include="Content\Custom.css" />
   <Content Include="CreateDatabase.sql" />
   <Content Include="DropDatabase.sql" />
   ...
</ItemGroup>
```

In this way, the project file is instructing MSBuild to construct lists of files that need to be processed in the same way—the **Reference** list includes assemblies that must be in place for a successful build, the **Compile** list includes code files that must be compiled, and the **Content** list includes resources that must be copied unaltered. We'll look at how the build process references and uses these items later in this topic.

Item elements can also include ItemMetadata child elements. These are user-defined key-value pairs and essentially represent properties that are specific to that item. For example, a lot of the **Compile** item elements in the project file include **DependentUpon** child elements.

```
<Compile Include="Global.asax.cs">
   <DependentUpon>Global.asax</DependentUpon>
</Compile>
```

> **NOTE**
>
> In addition to user-created item metadata, all items are assigned various common metadata on creation. For more information, see Well-known Item Metadata.

You can create **ItemGroup** elements within the root-level **Project** element or within specific **Target** elements. **ItemGroup** elements also support **Condition** attributes, which lets you tailor the inputs to the build process according to conditions like the project configuration or platform.

Targets and Tasks

In the MSBuild schema, a Task element represents an individual build instruction (or task). MSBuild includes a multitude of predefined tasks. For example:

- The **Copy** task copies files to a new location.
- The **Csc** task invokes the Visual C# compiler.
- The **Vbc** task invokes the Visual Basic compiler.
- The **Exec** task runs a specified program.
- The **Message** task writes a message to a logger.

> **NOTE**
> For full details of the tasks that are available out of the box, see MSBuild Task Reference. For more information on tasks, including how to create your own custom tasks, see MSBuild Tasks.

Tasks must always be contained within Target elements. A **Target** element is a set of one or more tasks that are executed sequentially, and a project file can contain multiple targets. When you want to run a task, or a set of tasks, you invoke the target that contains them. For example, suppose you have a simple project file that logs a message.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   <Target Name="LogMessage">
      <Message Text="Hello world!" />
   </Target>
</Project>
```

You can invoke the target from the command line, by using the **/t** switch to specify the target.

```
msbuild.exe Publish.proj /t:LogMessage
```

Alternatively, you can add a **DefaultTargets** attribute to the **Project** element, to specify the targets that you want to invoke.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         DefaultTargets="FullPublish">
   <Target Name="LogMessage">
      <Message Text="Hello world!" />
   </Target>
</Project>
```

In this case, you don't need to specify the target from the command line. You can simply specify the project file, and MSBuild will invoke the **FullPublish** target for you.

```
msbuild.exe Publish.proj
```

Both targets and tasks can include **Condition** attributes. As such, you can choose to omit entire targets or individual tasks if certain conditions are met.

Generally speaking, when you create useful tasks and targets, you'll need to refer to the properties and

items that you've defined elsewhere in the project file:

- To use a property value, type **$(*PropertyName*)**, where *PropertyName* is the name of the **Property** element or the name of the parameter.
- To use an item, type **@(*ItemName*)**, where *ItemName* is the name of the **Item** element.

> **NOTE**
>
> Remember that if you create multiple items with the same name, you're building a list. In contrast, if you create multiple properties with the same name, the last property value you provide will overwrite any previous properties with the same name—a property can only contain a single value.

For example, in the *Publish.proj* file in the sample solution, take a look at the **BuildProjects** target.

```
<Target Name="BuildProjects" Condition=" '$(BuildingInTeamBuild)'!='true' ">
   <MSBuild Projects="@(ProjectsToBuild)"
            Properties="OutDir=$(OutputRoot);
                        Configuration=$(Configuration);
                        DeployOnBuild=true;
                        DeployTarget=Package"
            Targets="Build" />
</Target>
```

In this sample, you can observe these key points:

- If the **BuildingInTeamBuild** parameter is specified and has a value of **true**, none of the tasks within this target will be executed.
- The target contains a single instance of the MSBuild task. This task lets you build other MSBuild projects.
- The **ProjectsToBuild** item is passed to the task. This item could represent a list of project or solution files, all defined by **ProjectsToBuild** item elements within an item group. In this case, the **ProjectsToBuild** item refers to a single solution file.

  ```
  <ItemGroup>
     <ProjectsToBuild Include="$(SourceRoot)ContactManager-WCF.sln"/>
  </ItemGroup>
  ```

- The property values passed to the **MSBuild** task include parameters named **OutputRoot** and **Configuration**. These are set to parameter values if they are provided, or static property values if they are not.

  ```
  <PropertyGroup>
     ...
     <Configuration Condition=" '$(Configuration)'==='' ">Release
     </Configuration>
     <OutputRoot Condition=" '$(OutputRoot)'==='' ">..\Publish\Out\
     </OutputRoot>
     ...
  </PropertyGroup>
  ```

You can also see that the **MSBuild** task invokes a target named **Build**. This is one of several built-in targets that are widely used in Visual Studio project files and are available to you in your custom project files, like **Build**, **Clean**, **Rebuild**, and **Publish**. You'll learn more about using targets and tasks to control the build process, and about the **MSBuild** task in particular, later in this topic.

> **NOTE**
>
> For more information on targets, see MSBuild Targets.

Splitting Project Files to Support Multiple Environments

Suppose you want to be able to deploy a solution to multiple environments, like test servers, staging platforms, and production environments. The configuration may vary substantially between these environments—not just in terms of server names, connection strings, and so on, but also potentially in terms of credentials, security settings, and lots of other factors. If you need to do this regularly, it's not really expedient to edit multiple properties in your project file every time you switch the target environment. Nor is it an ideal solution to require an endless list of property values to be provided to the build process.

Fortunately there is an alternative. MSBuild lets you split your build configuration across multiple project files. To see how this works, in the sample solution, notice that there are two custom project files:

- *Publish.proj*, which contains properties, items, and targets that are common to all environments.
- *Env-Dev.proj*, which contains properties that are specific to a developer environment.

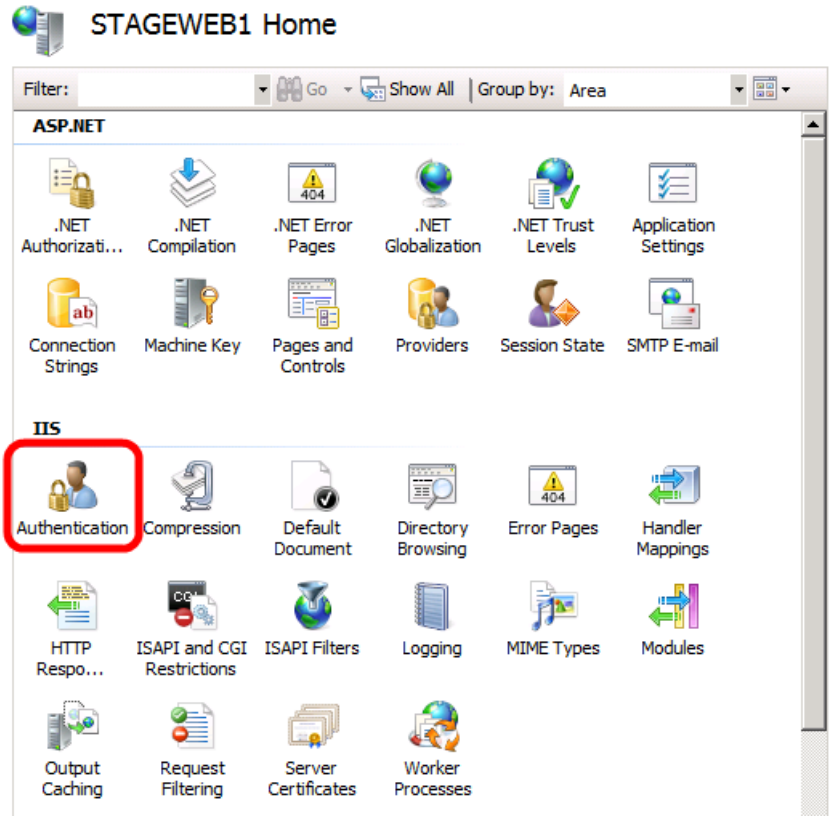
Now notice that the *Publish.proj* file includes an Import element, immediately beneath the opening **Project** tag.

```
<Import Project="$(TargetEnvPropsFile)"/>
```

The **Import** element is used to import the contents of another MSBuild project file into the current MSBuild project file. In this case, the **TargetEnvPropsFile** parameter provides the filename of the project file you want to import. You can provide a value for this parameter when you run MSBuild.

```
msbuild.exe Publish.proj /p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
```

This effectively merges the contents of the two files into a single project file. Using this approach, you can create one project file containing your universal build configuration and multiple supplementary project files containing environment-specific properties. As a result, simply running a command with a different parameter value lets you deploy your solution to a different environment.



Splitting your project files in this way is a good practice to follow. It allows developers to deploy to multiple environments by running a single command, while avoiding the duplication of universal build properties across multiple project files.

> **NOTE**
>
> For guidance on how to customize the environment-specific project files for your own server environments, see Configuring Deployment Properties for a Target Environment.

Conclusion

This topic provided a general introduction to MSBuild project files and explained how you can create your own custom project files to control the build process. It also introduced the concept of splitting project files into universal build instructions and environment-specific build properties, to make it easy to build and deploy projects to multiple destinations.

The next topic, Understanding the Build Process, provides more insight into how you can use project files to control build and deployment by walking you through the deployment of a solution with a realistic level of complexity.

Further Reading

For a more in-depth introduction to project files and the WPP, see Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build by Sayed Ibrahim Hashimi and William Bartholomew, ISBN: 978-0-7356-4524-0.

PREVIOUS NEXT

Understanding the Build Process

2/20/2020 • 12 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic provides a walkthrough of an enterprise-scale build and deployment process. The approach described in this topic uses custom Microsoft Build Engine (MSBuild) project files to provide fine-grained control over every aspect of the process. Within the project files, custom MSBuild targets are used to run deployment utilities like the Internet Information Services (IIS) Web Deployment Tool (MSDeploy.exe) and the database deployment utility VSDBCMD.exe.

> **NOTE**
>
> The previous topic, Understanding the Project File, described the key components of an MSBuild project file and introduced the concept of split project files to support deployment to multiple target environments. If you're not already familiar with these concepts, you should review Understanding the Project File before you work through this topic.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

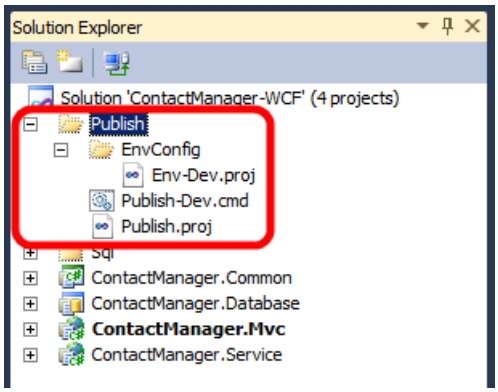
The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Project File, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Build and Deployment Overview

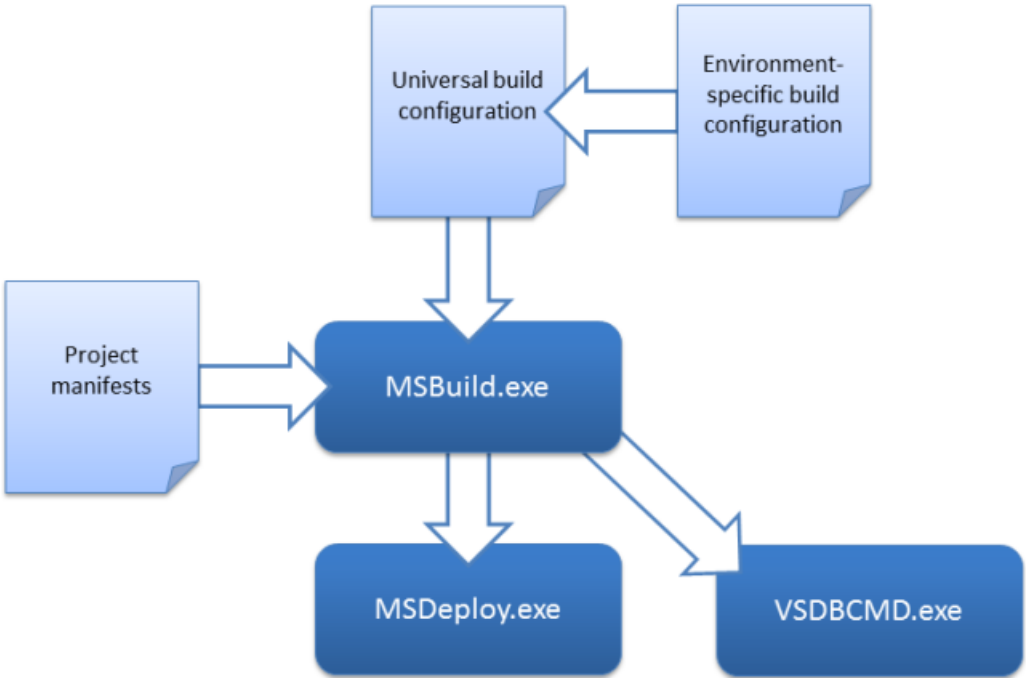
In the Contact Manager solution, three files control the build and deployment process:

- A *universal project file* (*Publish.proj*). This contains build and deployment instructions that do not change between destination environments.
- An *environment-specific project file* (*Env-Dev.proj*). This contains build and deployment settings that are specific to a particular destination environment. For example, you could use the *Env-Dev.proj* file to provide settings for a developer or test environment and create an alternative file named *Env-Stage.proj* to provide settings for a staging environment.
- A *command file* (*Publish-Dev.cmd*). This contains an MSBuild.exe command that specifies which project files you want to execute. You can create a command file for every destination environment, where each file contains an MSBuild.exe command that specifies a different environment-specific project file. This lets the developer deploy to different environments simply by running the appropriate command file.

In the sample solution, you can find these three files in the Publish solution folder.



Before you look at these files in more detail, let's take a look at how the overall build process works when you use this approach. At a high level, the build and deployment process looks like this:



The first thing that happens is that the two project files—one containing universal build and deployment instructions, and one containing environment-specific settings—are merged into a single project file. MSBuild then works through the instructions in the project file. It builds each of the projects in the solution, using the project file for each project. It then calls out to other tools, like Web Deploy (MSDeploy.exe) and the VSDBCMD utility to deploy your web content and databases to the target environment.

From start to finish, the build and deployment process performs these tasks:

1. It deletes the contents of the output directory, in preparation for a fresh build.
2. It builds each project in the solution:
   a. For web projects—in this case, an ASP.NET MVC web application and a WCF web service—the build process creates a web deployment package for each project.
   b. For database projects, the build process creates a deployment manifest (.deploymanifest file) for each project.
3. It uses the VSDBCMD.exe utility to deploy each database project in the solution, using various properties from the project files—a target connection string and a database name—together with the .deploymanifest file.
4. It uses the MSDeploy.exe utility to deploy each web project in the solution, using various properties from the project files to control the deployment process.

You can use the sample solution to trace this process in more detail.

> **NOTE**
> For guidance on how to customize the environment-specific project files for your own server environments, see Configure Deployment Properties for a Target Environment.

Invoking the Build and Deployment Process

To deploy the Contact Manager solution to a developer test environment, the developer runs the *Publish-Dev.cmd* command file. This invokes MSBuild.exe, specifying *Publish.proj* as the project file to execute and *Env-Dev.proj* as a parameter value.

```
msbuild.exe Publish.proj /fl /p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
```

> **NOTE**
> The **/fl** switch (short for **/fileLogger**) logs the build output to a file named *msbuild.log* in the current directory. For more information, see the MSBuild Command Line Reference.

At this point, MSBuild starts running, loads the *Publish.proj* file, and starts processing the instructions within it. The first instruction tells MSBuild to import the project file that the **TargetEnvPropsFile** parameter specifies.

```
<Import Project="$(TargetEnvPropsFile)" />
```

The **TargetEnvPropsFile** parameter specifies the *Env-Dev.proj* file, so MSBuild merges the contents of the *Env-Dev.proj* file into the *Publish.proj* file.

The next elements that MSBuild encounters in the merged project file are property groups. Properties are processed in the order in which they appear in the file. MSBuild creates a key-value pair for each property, providing that any specified conditions are met. Properties defined later in the file will overwrite any properties with the same name defined earlier in the file. For example, consider the **OutputRoot** properties.

```
<OutputRoot Condition=" '$(OutputRoot)'==''  ">..\Publish\Out\</OutputRoot>
<OutputRoot Condition=" '$(BuildingInTeamBuild)'=='true' ">$(OutDir)</OutputRoot>
```

When MSBuild processes the first **OutputRoot** element, providing a similarly named parameter has not been provided, it sets the value of the **OutputRoot** property to **..\Publish\Out**. When it encounters the second **OutputRoot** element, if the condition evaluates to **true**, it will overwrite the value of the **OutputRoot** property with the value of the **OutDir** parameter.

The next element that MSBuild encounters is a single item group, containing an item named **ProjectsToBuild**.

```
<ItemGroup>
   <ProjectsToBuild Include="$(SourceRoot)ContactManager-WCF.sln"/>
</ItemGroup>
```

MSBuild processes this instruction by building an item list named **ProjectsToBuild**. In this case, the item list contains a single value—the path and filename of the solution file.

At this point, the remaining elements are targets. Targets are processed differently from properties and items—essentially, targets are not processed unless they are either explicitly specified by the user or invoked by another construct within the project file. Recall that the opening **Project** tag includes a **DefaultTargets**

attribute.

```
<Project ToolsVersion="4.0"
         DefaultTargets="FullPublish"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
```

This instructs MSBuild to invoke the **FullPublish** target, if targets are not specified when MSBuild.exe is invoked. The **FullPublish** target doesn't contain any tasks; instead it simply specifies a list of dependencies.

```
<PropertyGroup>
  <FullPublishDependsOn>
     Clean;
     BuildProjects;
     GatherPackagesForPublishing;
     PublishDbPackages;
     PublishWebPackages;
  </FullPublishDependsOn>
</PropertyGroup>
<Target Name="FullPublish" DependsOnTargets="$(FullPublishDependsOn)" />
```

This dependency tells MSBuild that in order to execute the **FullPublish** target, it needs to invoke this list of targets in the order provided:

1. It must invoke the **Clean** target.
2. It must invoke the **BuildProjects** target.
3. It must invoke the **GatherPackagesForPublishing** target.
4. It must invoke the **PublishDbPackages** target.
5. It must invoke the **PublishWebPackages** target.

The Clean Target

The **Clean** target basically deletes the output directory and all its contents, as preparation for a fresh build.

```
<Target Name="Clean" Condition=" '$(BuildingInTeamBuild)'!='true' ">
  <Message Text="Cleaning up the output directory [$(OutputRoot)]"/>
  <ItemGroup>
     <_FilesToDelete Include="$(OutputRoot)**\*"/>
  </ItemGroup>
  <Delete Files="@(_FilesToDelete)"/>
  <RemoveDir Directories="$(OutputRoot)"/>
</Target>
```

Notice that the target includes an **ItemGroup** element. When you define properties or items within a **Target** element, you're creating *dynamic* properties and items. In other words, the properties or items aren't processed until the target is executed. The output directory might not exist or contain any files until the build process begins, so you can't build the **_FilesToDelete** list as a static item; you have to wait until execution is underway. As such, you build the list as a dynamic item within the target.

> **NOTE**
>
> In this case, because the **Clean** target is the first to be executed, there's no real need to use a dynamic item group. However, it's good practice to use dynamic properties and items in this type of scenario, as you might want to execute targets in a different order at some point.
> You should also aim to avoid declaring items that will never be used. If you have items that will only be used by a specific target, consider placing them inside the target to remove any unnecessary overhead on the build process.

Dynamic items aside, the **Clean** target is fairly straightforward and makes use of the built-in **Message**, **Delete**, and **RemoveDir** tasks to:

1. Send a message to the logger.
2. Build a list of files to delete.
3. Delete the files.
4. Remove the output directory.

The BuildProjects Target

The **BuildProjects** target basically builds all the projects in the sample solution.

```
<Target Name="BuildProjects" Condition=" '$(BuildingInTeamBuild)'!='true' ">
   <MSBuild Projects="@(ProjectsToBuild)"
            Properties="OutDir=$(OutputRoot);
                        Configuration=$(Configuration);
                        DeployOnBuild=true;
                        DeployTarget=Package"
            Targets="Build" />
  </Target>
```

This target was described in some detail in the previous topic, Understanding the Project File, to illustrate how tasks and targets reference properties and items. At this point, you're mainly interested in the **MSBuild** task. You can use this task to build multiple projects. The task does not create a new instance of MSBuild.exe; it uses the current running instance to build each project. The key points of interest in this example are the deployment properties:

- The **DeployOnBuild** property instructs MSBuild to run any deployment instructions in the project settings when the build of each project is complete.
- The **DeployTarget** property identifies the target that you want to invoke after the project is built. In this case, the **Package** target builds the project output into a deployable web package.

> **NOTE**
>
> The **Package** target invokes the Web Publishing Pipeline (WPP), which provides integration between MSBuild and Web Deploy. If you want to take a look at the built-in targets that the WPP provides, review the *Microsoft.Web.Publishing.targets* file in the %PROGRAMFILES(x86)%\MSBuild\Microsoft\VisualStudio\v10.0\Web folder.

The GatherPackagesForPublishing Target

If you study the **GatherPackagesForPublishing** target, you'll notice that it doesn't actually contain any tasks. Instead, it contains a single item group that defines three dynamic items.

```
<Target Name="GatherPackagesForPublishing">
   <ItemGroup>
      <PublishPackages
         Include="$(_ContactManagerDest)ContactManager.Mvc.deploy.cmd">
         <WebPackage>true</WebPackage>
         <!-- More item metadata -->
      </PublishPackages>
      <PublishPackages
         Include="$(_ContactManagerSvcDest)ContactManager.Service.deploy.cmd">
         <WebPackage>true</WebPackage>
         <!-- More item metadata -->
      </PublishPackages>
      <DbPublishPackages Include="$(_DbDeployManifestPath)">
         <DbPackage>true</DbPackage>
         <!-- More item metadata -->
      </DbPublishPackages>
   </ItemGroup>
</Target>
```

These items refer to the deployment packages that were created when the **BuildProjects** target was executed. You couldn't define these items statically in the project file, because the files to which the items refer don't exist until the **BuildProjects** target is executed. Instead, the items must be defined dynamically within a target that is not invoked until after the **BuildProjects** target is executed.

The items are not used within this target—this target simply builds the items and the metadata associated with each item value. Once these elements are processed, the **PublishPackages** item will contain two values, the path to the *ContactManager.Mvc.deploy.cmd* file and the path to the *ContactManager.Service.deploy.cmd* file. Web Deploy creates these files as part of the web package for each project, and these are the files that you must invoke on the destination server in order to deploy the packages. If you open up one of these files, you'll basically see an MSDeploy.exe command with various build-specific parameter values.

The **DbPublishPackages** item will contain a single value, the path to the *ContactManager.Database.deploymanifest* file.

> **NOTE**
> A .deploymanifest file is generated when you build a database project, and it uses the same schema as an MSBuild project file. It contains all the information required to deploy a database, including the location of the database schema (.dbschema) and details of any pre-deployment and post-deployment scripts. For more information, see An Overview of Database Build and Deployment.

You'll learn more about how deployment packages and database deployment manifests are created and used in Building and Packaging Web Application Projects and Deploying Database Projects.

The PublishDbPackages Target

Briefly speaking, the **PublishDbPackages** target invokes the VSDBCMD utility to deploy the **ContactManager** database to a target environment. Configuring database deployment involves lots of decisions and nuances, and you'll learn more about this in Deploying Database Projects and Customizing Database Deployments for Multiple Environments. In this topic, we'll focus on how this target actually functions.

First, notice that the opening tag includes an **Outputs** attribute.

```
<Target Name="PublishDbPackages" Outputs="%(DbPublishPackages.Identity)">
```

This is an example of *target batching*. In MSBuild project files, batching is a technique for iterating over

collections. The value of the **Outputs** attribute, **"%(DbPublishPackages.Identity)"**, refers to the **Identity** metadata property of the **DbPublishPackages** item list. This notation, **Outputs=%***(ItemList.ItemMetadataName)*, is translated as:

- Split the items in **DbPublishPackages** into batches of items that contain the same **Identity** metadata value.
- Execute the target once per batch.

> **NOTE**
>
> **Identity** is one of the built-in metadata values that is assigned to every item on creation. It refers to the value of the **Include** attribute in the **Item** element—in other words, the path and filename of the item.

In this case, because there should never be more than one item with the same path and filename, we're essentially working with batch sizes of one. The target is executed once for every database package.

You can see a similar notation in the **_Cmd** property, which builds a VSDBCMD command with the appropriate switches.

```
<_Cmd>"$(VsdbCmdExe)"
   /a:Deploy
   /cs:"%(DbPublishPackages.DatabaseConnectionString)"
   /p:TargetDatabase=%(DbPublishPackages.TargetDatabase)
   /manifest:"%(DbPublishPackages.FullPath)"
   /script:"$(_CmDbScriptPath)"
   $(_DbDeployOrScript)
</_Cmd>
```

In this case, **%(DbPublishPackages.DatabaseConnectionString)**, **%(DbPublishPackages.TargetDatabase)**, and **%(DbPublishPackages.FullPath)** all refer to metadata values of the **DbPublishPackages** item collection. The **_Cmd** property is used by the **Exec** task, which invokes the command.

```
<Exec Command="$(_Cmd)"/>
```

As a result of this notation, the **Exec** task will create batches based on unique combinations of the **DatabaseConnectionString**, **TargetDatabase**, and **FullPath** metadata values, and the task will execute once for each batch. This is an example of *task batching*. However, because the target-level batching has already divided our item collection into single-item batches, the **Exec** task will run once and only once for each iteration of the target. In other words, this task invokes the VSDBCMD utility once for each database package in the solution.

> **NOTE**
>
> For more information on target and task batching, see MSBuild Batching, Item Metadata in Target Batching, and Item Metadata in Task Batching.

The PublishWebPackages Target

By this point, you've invoked the **BuildProjects** target, which generates a web deployment package for each project in the sample solution. Accompanying each package is a *deploy.cmd* file, which contains the MSDeploy.exe commands required to deploy the package to the target environment, and a *SetParameters.xml* file, which specifies the necessary details of the target environment. You've also invoked the **GatherPackagesForPublishing** target, which generates an item collection containing the *deploy.cmd* files

you're interested in. Essentially, the **PublishWebPackages** target performs these functions:

- It manipulates the *SetParameters.xml* file for each package to include the correct details for the target environment, using the **XmlPoke** task.
- It invokes the *deploy.cmd* file for each package, using the appropriate switches.

Just like the **PublishDbPackages** target, the **PublishWebPackages** target uses target batching to ensure that the target is executed once for each web package.

```
<Target Name="PublishWebPackages" Outputs="%(PublishPackages.Identity)">
```

Within the target, the **Exec** task is used to run the *deploy.cmd* file for each web package.

```
<PropertyGroup>
   <_Cmd>
      %(PublishPackages.FullPath)
      $(_WhatifSwitch)
      /M:$(MSDeployComputerName)
      %(PublishPackages.AdditionalMSDeployParameters)
   </_Cmd>
</PropertyGroup>
<Exec Command="$(_Cmd)"/>
```

For more information on configuring the deployment of web packages, see Building and Packaging Web Application Projects.

Conclusion

This topic provided a walkthrough of how split project files are used to control the build and deployment process from start to finish for the Contact Manager sample solution. Using this approach lets you run complex, enterprise-scale deployments in a single, repeatable step, simply by running an environment-specific command file.

Further Reading

For a more in-depth introduction to project files and the WPP, see Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build by Sayed Ibrahim Hashimi and William Bartholomew, ISBN: 978-0-7356-4524-0.

PREVIOUS NEXT

Building and Packaging Web Application Projects

2/20/2020 • 6 minutes to read • Edit Online

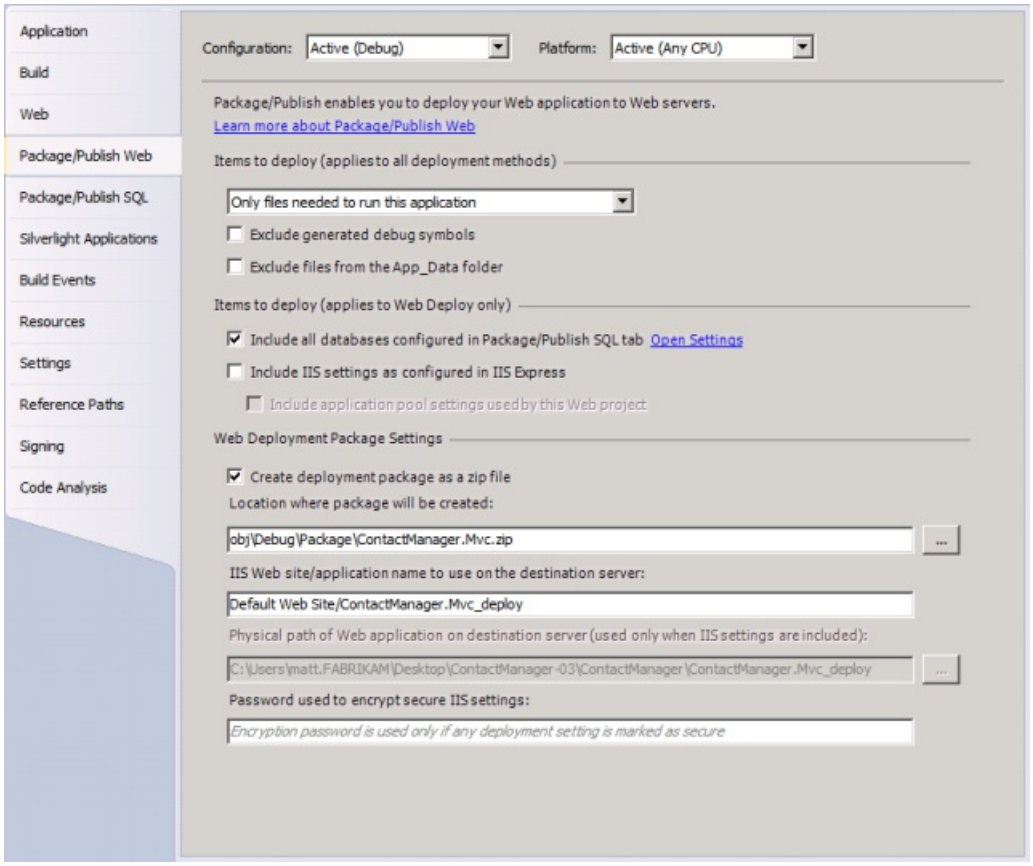
by Jason Lee

Download PDF

When you want to deploy a web application project to a remote server environment, your first task is to build the project and generate a web deployment package. This topic describes how the build process works for web application projects. In particular, it explains:

- How the Web Publishing Pipeline (WPP) extends the build process to include deployment functionality.
- How the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) turns your web application into a deployment package.
- How the build and packaging process works and what files are created.

In Visual Studio 2010, the build and deployment process for web application projects is supported by the WPP. The WPP provides a set of Microsoft Build Engine (MSBuild) targets that extend the functionality of MSBuild and enable it to integrate with Web Deploy. Within Visual Studio, you can see this extended functionality on the property pages for your web application project. The **Package/Publish Web** page, together with the **Package/Publish SQL** page, lets you configure how your web application project is packaged for deployment when the build process is complete.



How Does the WPP Work?

If you take a look at the project file for a C#-based web application project, you can see that it imports two .targets files.

```
<Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
<Import Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\
                 v10.0\WebApplications\Microsoft.WebApplication.targets" />
```

The first **Import** statement is common to all Visual C# projects. This file, *Microsoft.CSharp.targets*, contains targets and tasks that are specific to Visual C#. For example, the C# compiler (**Csc**) task is invoked here. The *Microsoft.CSharp.targets* file in turn imports the *Microsoft.Common.targets* file. This defines targets that are common to all projects, like **Build**, **Rebuild**, **Run**, **Compile**, and **Clean**. The second **Import** statement is specific to web application projects. The *Microsoft.WebApplication.targets* file in turn imports the *Microsoft.Web.Publishing.targets* file. The *Microsoft.Web.Publishing.targets* file essentially *is* the WPP. It defines targets, like **Package** and **MSDeployPublish**, that invoke Web Deploy to complete various deployment tasks.

To understand how these additional targets are used, in the Contact Manager sample solution, open the *Publish.proj* file and take a look at the **BuildProjects** target.

```
<Target Name="BuildProjects" Condition=" '$(BuildingInTeamBuild)'!='true' ">
  <MSBuild Projects="@(ProjectsToBuild)"
           Properties="OutDir=$(OutputRoot);
                       Configuration=$(Configuration);
                       DeployOnBuild=true;
                       DeployTarget=Package"
           Targets="Build" />
</Target>
```

This target uses the **MSBuild** task to build various projects. Notice the **DeployOnBuild** and **DeployTarget** properties:

- The **DeployOnBuild=true** property essentially means "I want to execute an additional target when build completes successfully."
- The **DeployTarget** property identifies the name of the target you want to execute when the **DeployOnBuild** property is equal to **true**. In this case, you're specifying that you want MSBuild to execute the **Package** target after building the project.

The **Package** target is defined in the *Microsoft.Web.Publishing.targets* file. Essentially, this target takes the build output of your web application project and turns it into a web deployment package that can be published to an IIS web server.

> **NOTE**
>
> To view a project file (for example, *ContactManager.Mvc.csproj*) in Visual Studio 2010, you first need to unload the project from your solution. In the **Solution Explorer** window, right-click the project node, and then click **Unload Project**. Right-click the project node again, and then click **Edit** *[project file]*. The project file will open in its raw XML form. Remember to reload the project when you're done.
> For more information on MSBuild targets, tasks, and **Import** statements, see Understanding the Project File. For a more in-depth introduction to project files and the WPP, see Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build by Sayed Ibrahim Hashimi and William Bartholomew, ISBN: 978-0-7356-4524-0.

What Is a Web Deployment Package?

When you build and deploy a web application project, either by using Visual Studio 2010 or by using MSBuild directly, the end result is typically a *web deployment package*. The web deployment package is a .zip file. It contains everything that IIS and Web Deploy need in order to recreate your web application, including:

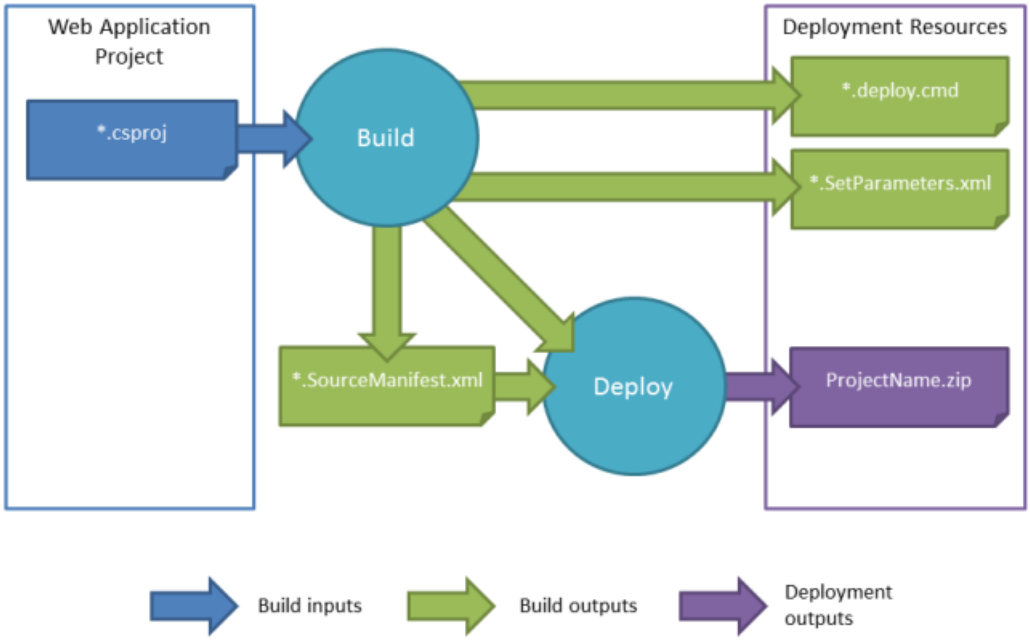
- The compiled output of your web application, including content, resource files, configuration files, JavaScript and cascading style sheets (CSS) resources, and so on.

- Assemblies for your web application project and for any referenced projects within your solution.
- SQL scripts to generate any databases that you're deploying with your web application.

Once the web deployment package has been generated, you can publish it to an IIS web server in various ways. For example, you can deploy it remotely by targeting the Web Deploy Remote Agent service or the Web Deploy Handler on the destination web server, or you can use IIS Manager to manually import the package on the destination web server. For more information on these approaches to deployment, see Choosing the Right Approach to Web Deployment.

How Does the Build Process Work?

This shows what happens when you build and package a web application project:



When you build a web application project, the build process generates a file named *[project name].SourceManifest.xml*. Along with the project file and the build output, this *.SourceManifest.xml* file tells Web Deploy what it needs to include in the web deployment package. Using these inputs, Web Deploy generates a web deployment package named *[project name].zip*.

Alongside the web deployment package, the build process generates two files that can help you to use the package:

- The *.deploy.cmd* file includes a set of parameterized Web Deploy (MSDeploy.exe) commands that publish your web deployment package to a remote IIS web server. Running the *.deploy.cmd* file, with appropriate parameters, typically provides a quicker and easier alternative to manually constructing the MSDeploy.exe commands yourself.
- The *SetParameters.xml* file provides a set of parameter values to the MSDeploy.exe command. These values include properties like the name of the IIS web application to which you want to deploy the package, the values of any service endpoints and connection strings defined in the *web.config* file, and any deployment property values defined on the project properties pages.

The *SetParameters.xml* file is key to managing the deployment process. This file is generated dynamically according to the contents of your web application project. For example, if you add a connection string to your *web.config* file, the build process will automatically detect the connection string, parameterize the deployment accordingly, and create an entry in the *SetParameters.xml* file to allow you to modify the connection string as part of the deployment process. The next topic, Configuring Parameters for Web Package Deployment, explains the role of this file in more detail and describes the different ways in which you can modify it during build and deployment.

NOTE

In Visual Studio 2010, the WPP does not support precompiling the pages in a web application prior to packaging. The next version of Visual Studio and the WPP will include the ability to precompile a web application as a packaging option.

Conclusion

This topic provided an overview of the build and packaging process for web application projects in Visual Studio 2010. It described how the WPP lets you invoke Web Deploy commands from MSBuild, and it explained how the build and packaging process works.

Once you've created a web deployment package, your next step is to deploy it. For more information on this, see Configuring Parameters for Web Package Deployment and Deploying Web Packages.

Further Reading

The next topics in this tutorial, Configuring Parameters for Web Package Deployment and Deploying Web Packages, provide guidance on how to use the web package you've created. The final tutorial in this series, Advanced Enterprise Web Deployment, provides guidance on how to customize and troubleshoot the packaging process.

For a more in-depth introduction to project files and the WPP, see Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build by Sayed Ibrahim Hashimi and William Bartholomew, ISBN: 978-0-7356-4524-0.

PREVIOUS NEXT

Configuring Parameters for Web Package Deployment

2/20/2020 • 8 minutes to read • Edit Online

by Jason Lee

Download PDF

> This topic describes how to set parameter values, like Internet Information Services (IIS) web application names, connection strings, and service endpoints, when you deploy a web package to a remote IIS web server.

When you build a web application project, the build and packaging process generates three key files:

- A *[project name].zip* file. This is the web deployment package for your web application project. This package contains all the assemblies, files, database scripts, and resources required to recreate your web application on a remote IIS web server.
- A *[project name].deploy.cmd* file. This contains a set of parameterized Web Deploy (MSDeploy.exe) commands that publish your web deployment package to a remote IIS web server.
- A *[project name].SetParameters.xml* file. This provides a set of parameter values to the MSDeploy.exe command. You can update the values in this file and pass it to Web Deploy as a command-line parameter when you deploy your web package.

> **NOTE**
>
> For more information on the build and packaging process, see Building and Packaging Web Application Projects.

The *SetParameters.xml* file is dynamically generated from your web application project file and any configuration files within your project. When you build and package your project, the Web Publishing Pipeline (WPP) will automatically detect lots of the variables that are likely to change between deployment environments, like the destination IIS web application and any database connection strings. These values are automatically parameterized in the web deployment package and added to the *SetParameters.xml* file. For example, if you add a connection string to the *web.config* file in your web application project, the build process will detect this change and will add an entry to the *SetParameters.xml* file accordingly.

In a lot of cases, this automatic parameterization will be sufficient. However, if your users need to vary other settings between deployment environments, like application settings or service endpoint URLs, you need to tell the WPP to parameterize these values in the deployment package and add corresponding entries to the *SetParameters.xml* file. The sections that follow explain how to do this.

Automatic Parameterization

When you build and package a web application, the WPP will automatically parameterize these things:

- The destination IIS web application path and name.
- Any connection strings in your *web.config* file.
- Connection strings for any databases you add to the **Package/Publish SQL** tab in the project property pages.

For example, if you were to build and package the Contact Manager sample solution without touching the

parameterization process in any way, the WPP would generate this *ContactManager.Mvc.SetParameters.xml* file:

```
<parameters>
  <setParameter
    name="IIS Web Application Name"
    value="Default Web Site/ContactManager.Mvc_deploy" />
  <setParameter
    name="ApplicationServices-Web.config Connection String"
    value="Data Source=DEVWORKSTATION\SQLEXPRESS;Initial Catalog=CMAppServices;
           Integrated Security=true;" />
</parameters>
```

In this case:

- The **IIS Web Application Name** parameter is the IIS path where you want to deploy the web application. The default value is taken from the **Package/Publish Web** page in the project property pages.
- The **ApplicationServices-Web.config Connection String** parameter was generated from a **connectionStrings/add** element in the *web.config* file. It represents the connection string that the application should use to contact the membership database. The value you provide here will be substituted into the deployed *web.config* file. The default value is taken from the pre-deployment *web.config* file.

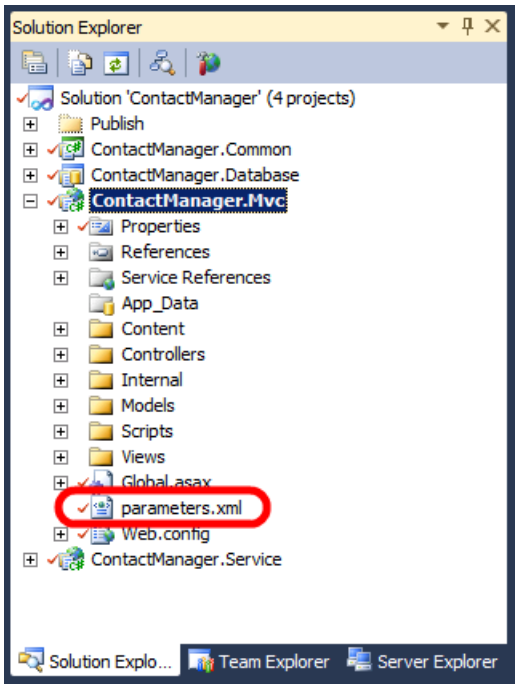
The WPP also parameterizes these properties in the deployment package it generates. You can provide values for these properties when you install the deployment package. If you install the package manually through IIS Manager, as described in Manually Installing Web Packages, the installation wizard prompts you to provide values for any parameters. If you install the package remotely using the *.deploy.cmd* file, as described in Deploying Web Packages, Web Deploy will look to this *SetParameters.xml* file to provide the parameter values. You can edit the values in the *SetParameters.xml* file manually, or you can customize the file as part of an automated build and deployment process. This process is described in more detail later in this topic.

Custom Parameterization

In more complex deployment scenarios, you'll often want to parameterize additional properties before you deploy your project. Generally speaking, you should parameterize any properties and settings that will vary between destination environments. These can include:

- Service endpoints in the *web.config* file.
- Application settings in the *web.config* file.
- Any other declarative properties that you want to prompt users to specify.

The easiest way to parameterize these properties is to add a *parameters.xml* file to the root folder of your web application project. For example, in the Contact Manager solution, the ContactManager.Mvc project includes a *parameters.xml* file in the root folder.



If you open this file, you'll see that it contains a single **parameter** entry. The entry uses an XML Path Language (XPath) query to locate and parameterize the endpoint URL of the ContactService Windows Communication Foundation (WCF) service in the *web.config* file.

```
<parameters>
  <parameter name="ContactService Service Endpoint Address"
             description="Specify the endpoint URL for the ContactService WCF
                          service in the destination environment"
             defaultValue="http://localhost/ContactManagerService">
    <parameterEntry kind="XmlFile" scope="Web.config"
                    match="/configuration/system.serviceModel/client
                           /endpoint[@name='BasicHttpBinding_IContactService']
                           /@address" />
  </parameter>
</parameters>
```

In addition to parameterizing the endpoint URL in the deployment package, the WPP also adds a corresponding entry to the *SetParameters.xml* file that gets generated alongside the deployment package.

```
<parameters>
  ...
  <setParameter
    name="ContactService Service Endpoint Address"
    value="http://localhost/ContactManagerService" />
  ...
</parameters>
```

If you install the deployment package manually, IIS Manager will prompt you for the service endpoint address alongside the properties that were parameterized automatically. If you install the deployment package by running the *.deploy.cmd* file, you can edit the *SetParameters.xml* file to provide a value for the service endpoint address together with values for the properties that were parameterized automatically.

For full details on how to create a *parameters.xml* file, see How to: Use Parameters to Configure Deployment Settings When a Package is Installed. The procedure named **To use deployment parameters for Web.config file settings** provides step-by-step instructions.

Modifying the SetParameters.xml File

If you plan to deploy the web application package manually—either by running the *.deploy.cmd* file or by running MSDeploy.exe from the command line—there's nothing to stop you manually editing the *SetParameters.xml* file prior to the deployment. However, if you're working on an enterprise-scale solution, you may need to deploy a web application package as part of a larger, automated build and deployment process. In this scenario, you need the Microsoft Build Engine (MSBuild) to modify the *SetParameters.xml* file for you. You can do this by using the MSBuild **XmlPoke** task.

The Contact Manager sample solution illustrates this process. The code examples that follow have been edited to show only the details that are relevant to this example.

> **NOTE**
>
> For a broader overview of the project file model in the sample solution, and an introduction to custom project files in general, see Understanding the Project File and Understanding the Build Process.

First, the parameter values of interest are defined as properties in the environment-specific project file (for example, *Env-Dev.proj*).

```
<PropertyGroup>
  <ContactManagerIisPath Condition=" '$(ContactManagerIisPath)'==''  ">
    DemoSite/ContactManager
  </ContactManagerIisPath>
  <ContactManagerTargetUrl Condition =" '$(ContactManagerTargetUrl)'==''  ">
    http://localhost:85/ContactManagerService/ContactService.svc
  </ContactManagerTargetUrl>
  <MembershipConnectionString Condition=" '$(MembershipConnectionString)'==''  ">
    Data Source=TESTDB1;Integrated Security=true;Initial Catalog=CMAppServices
  </MembershipConnectionString>
</PropertyGroup>
```

> **NOTE**
>
> For guidance on how to customize the environment-specific project files for your own server environments, see Configure Deployment Properties for a Target Environment.

Next, the *Publish.proj* file imports these properties. Because each *SetParameters.xml* file is associated with a *.deploy.cmd* file, and we ultimately want the project file to invoke each *.deploy.cmd* file, the project file creates an MSBuild *item* for each *.deploy.cmd* file and defines the properties of interest as *item metadata*.

```
<ItemGroup>
  <PublishPackages Include="$(_ContactManagerDest)ContactManager.Mvc.deploy.cmd">
    <ParametersXml>
      $(_ContactManagerDest)ContactManager.Mvc.SetParameters.xml
    </ParametersXml>
    <IisWebAppName>
      $(ContactManagerIisPath)
    </IisWebAppName>
    <MembershipDBConnectionName>
      ApplicationServices-Web.config Connection String
    </MembershipDBConnectionName>
    <MembershipDBConnectionString>
      $(MembershipConnectionString.Replace(";","%3b"))
    </MembershipDBConnectionString>
    <ServiceEndpointParamName>
      ContactService Service Endpoint Address
    </ServiceEndpointParamName>
    <ServiceEndpointValue>
      $(ContactManagerTargetUrl)
    </ServiceEndpointValue>
  </PublishPackages>
  ...
</ItemGroup>
```

In this case:

- The **ParametersXml** metadata value indicates the location of the *SetParameters.xml* file.
- The **IisWebAppName** value is the IIS path to which you want to deploy the web application.
- The **MembershipDBConnectionString** value is the connection string for the membership database, and the **MembershipDBConnectionName** value is the **name** attribute of the corresponding parameter in the *SetParameters.xml* file.
- The **ServiceEndpointValue** value is the endpoint address for the WCF service on the destination server, and the **ServiceEndpointParamName** value is the name attribute of the corresponding parameter in the *SetParameters.xml* file.

Finally, in the *Publish.proj* file, the **PublishWebPackages** target uses the **XmlPoke** task to modify these values in the *SetParameters.xml* file.

```
<Target Name="PublishWebPackages" Outputs="%(PublishPackages.Identity)">
  <XmlPoke
    XmlInputPath="%(PublishPackages.ParametersXml)"
    Query="//parameters/setParameter[@name='%(PublishPackages.ConnectionName)']
           /@value"
    Value="%(PublishPackages.ConnectionString)"
    Condition =" '%(PublishPackages.ConnectionName)'!='' "
  />
  <XmlPoke
    XmlInputPath="%(PublishPackages.ParametersXml)"
    Query="//parameters/setParameter
           [@name='%(PublishPackages.MembershipDBConnectionName)']/@value"
    Value='%(PublishPackages.MembershipDBConnectionString)'
    Condition =" '%(PublishPackages.MembershipDBConnectionName)'!='' "
  />
  <XmlPoke
    XmlInputPath="%(PublishPackages.ParametersXml)"
    Query="//parameters/setParameter[@name='IIS Web Application Name']/@value"
    Value="%(PublishPackages.IisWebAppName)"
    Condition =" '%(PublishPackages.IisWebAppName)'!='' "
  />
  <XmlPoke
    XmlInputPath="%(PublishPackages.ParametersXml)"
    Query="//parameters/setParameter
           [@name='%(PublishPackages.ServiceEndpointParamName)']/@value"
    Value="%(PublishPackages.ServiceEndpointValue)"
    Condition =" '%(PublishPackages.ServiceEndpointParamName)'!='' "
  />
  <!--Execute the .deploy.cmd file-->
  ...
</Target>
```

You'll notice that each **XmlPoke** task specifies four attribute values:

- The **XmlInputPath** attribute tells the task where to find the file you want to modify.
- The **Query** attribute is an XPath query that identifies the XML node you want to change.
- The **Value** attribute is the new value you want to insert into the selected XML node.
- The **Condition** attribute is the criteria on which the task should run or not run. In these cases, the condition ensures that you don't try to insert a null or empty value into the *SetParameters.xml* file.

Conclusion

This topic described the role of the *SetParameters.xml* file and explained how it's generated when you build a web application project. It explained how you can parameterize additional settings by adding a *parameters.xml* file to your project. It also described how you can modify the *SetParameters.xml* file as part of a larger, automated build process, by using the **XmlPoke** task in your project files.

The next topic, Deploying Web Packages, describes how you can deploy a web package either by running the *.deploy.cmd* file or by using MSDeploy.exe commands directly. In both cases, you can specify your *SetParameters.xml* file as a deployment parameter.

Further Reading

For information on how to create web packages, see Building and Packaging Web Application Projects. For guidance on how to actually deploy a web package, see Deploying Web Packages. For a step-by-step walkthrough on how to create a *parameters.xml* file, see How to: Use Parameters to Configure Deployment Settings When a Package is Installed.

For more general information on parameterization in Web Deploy, see Web Deploy Parameterization in Action (blog post).

Deploying Web Packages

2/20/2020 • 12 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic describes how you can publish web deployment packages to a remote server by using the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) 2.0.

There are two main ways in which you can deploy a web package to a remote server:

- You can use the MSDeploy.exe command-line utility directly.
- You can run the *[project name].deploy.cmd* file that the build process generates.

The end result is the same regardless of which approach you use. Essentially, all the *.deploy.cmd* file does is to run MSDeploy.exe with some predetermined values, so that you don't have to provide as much information in order to deploy the package. This simplifies the deployment process. On the other hand, using MSDeploy.exe directly gives you a lot more flexibility over exactly how your package is deployed.

Which approach you use will depend on a variety of factors, including how much control you require over the deployment process and whether you're targeting the Web Deploy Remote Agent service or the Web Deploy Handler. This topic explains how to use each approach and identifies when each approach is appropriate.

The tasks and walkthroughs in this topic assume that:

- You've built and packaged your web application, as described in Building and Packaging Web Application Projects.
- You've modified the *SetParameters.xml* file to provide the right parameter values for your target environment, as described in Configuring Parameters for Web Package Deployment.

Running the [*project name*].*deploy.cmd* file is the simplest way to deploy a web package. In particular, using the *.deploy.cmd* file offers these advantages over using MSDeploy.exe directly:

- You don't need to specify the location of the web deployment package—the *.deploy.cmd* file already knows where it is.
- You don't need to specify the location of the *SetParameters.xml* file—the *.deploy.cmd* file already knows where it is.
- You don't need to specify source and destination MSDeploy providers—the *.deploy.cmd* file already knows which values to use.
- You don't need to specify MSDeploy operation settings—the *.deploy.cmd* file adds the commonly required values to the MSDeploy.exe command automatically.

Before you use the *.deploy.cmd* file to deploy a web package, you should ensure that:

- The *.deploy.cmd* file, the [*project name*].*SetParameters.xml* file, and the web package ([*project name*].*zip*) are in the same folder.
- Web Deploy (MSDeploy.exe) is installed on the computer that runs the *.deploy.cmd* file.

The *.deploy.cmd* file supports various command-line options. When you run the file from a command prompt, this is the basic syntax:

```
[project name].deploy.cmd [/T | /Y]
                          [/M:<computer name>]
                          [/A:<Basic | NTLM>]
                          [/U:<user name>]
                          [/P:<password>]
                          [/L]
                          [/G:<true | false>]
                          [Additional MSDeploy.exe flags]
```

You must specify either a **/T** flag or a **/Y** flag, to indicate whether you want to perform a trial run or a live deployment respectively (don't use both flags in the same command). This table explains the purpose of each of these flags.

FLAG	DESCRIPTION
/T	Calls MSDeploy.exe with the **–whatif** flag, which indicates a trial run. Rather than deploying the package, it creates a report of what would happen if you did deploy the package.
/Y	Calls MSDeploy.exe without the **–whatif** flag. This deploys the package to the local computer or the specified destination server.
/M	Specifies the destination server name or service URL. For more information on the values you can provide here, see the **Endpoint Considerations** section in this topic. If you omit the **/M** flag, the package will be deployed to the local computer.
/A	Specifies the authentication type that MSDeploy.exe should use to perform the deployment. Possible values are **NTLM** and **Basic**. If you omit the **/A** flag, the authentication type defaults to **NTLM** for deployment to the Web Deploy Remote Agent service and to **Basic** for deployment to the Web Deploy Handler.
/U	Specifies the user name. This applies only if you're using basic authentication.
/P	Specifies the password. This applies only if you're using basic authentication.
/L	Indicates that the package should be deployed to the local IIS Express instance.
/G	Specifies that the package is deployed using the tempAgent provider setting. If you omit the **/G** flag, the value defaults to **false**.

NOTE

Every time the build process creates a web package, it also creates a file named *[project name].deploy-readme.txt* that explains these deployment options.

In addition to these flags, you can specify Web Deploy operation settings as additional *.deploy.cmd* parameters. Any additional settings you specify are simply passed through to the underlying MSDeploy.exe command. For

more information on these settings, see Web Deploy Operation Settings.

Suppose you want to deploy the ContactManager.Mvc web application project to a test environment by running the *.deploy.cmd* file. Your test environment is configured to use the Web Deploy Remote Agent service, as described in Configure a Web Server for Web Deploy Publishing (Remote Agent). To deploy the web application, you need to complete the next steps.

To deploy a web application using the .deploy.cmd file

1. Build and package the web application project, as described in Building and Packaging Web Application Projects.
2. Modify the *ContactManager.Mvc.SetParameters.xml* file to contain the correct parameter values for your test environment, as described in Configuring Parameters for Web Package Deployment.
3. Open a Command Prompt window and navigate to the location of the *ContactManager.Mvc.deploy.cmd* file.
4. Type this command, and then press Enter:

```
ContactManager.Mvc.deploy.cmd /Y /M:TESTWEB1 /A:NTLM
```

In this example:

- The **/Y** flag indicates that you want to actually deploy the package, rather than doing a trial run.
- The **/M** flag indicates that you want to deploy the package to the server named TESTWEB1. From this value, MSDeploy.exe will attempt to deploy the package to the Web Deploy Remote Agent service at http://TESTWEB1/MSDeployAgentService.
- The **/A** flag indicates that you want to use NTLM authentication. As such, you don't need to specify a user name and password.

To illustrate how using the *.deploy.cmd* file simplifies the deployment process, take a look at the MSDeploy.exe command that gets generated and executed when you run *ContactManager.Mvc.deploy.cmd* using the options shown above.

```
msdeploy.exe
-source:package='C:\Users\matt.FABRIKAM\Desktop\ContactManager-03\ContactManager\
 Publish\Out\_PublishedWebsites\ContactManager.Mvc_Package\ContactManager.Mvc.zip' -
dest:auto,computerName='TESTWEB1.fabrikam.net', authtype='NTLM',
 includeAcls='False'
-verb:sync
-disableLink:AppPoolExtension
-disableLink:ContentExtension
-disableLink:CertificateExtension
-setParamFile:"C:\Users\matt.FABRIKAM\Desktop\ContactManager-03\ContactManager\
 Publish\Out\_PublishedWebsites\ContactManager.Mvc_Package\
 ContactManager.Mvc.SetParameters.xml"
```

For more information on using the *.deploy.cmd* file to deploy a web package, see How to: Install a Deployment Package Using the deploy.cmd File.

Using MSDeploy.exe

Although using the *.deploy.cmd* file generally simplifies the deployment process, there are some situations when it's preferable to use MSDeploy.exe directly. For example:

- If you want to deploy to the Web Deploy Handler as a non-administrator user, you can't use the *.deploy.cmd*

file. This is due to a bug in Web Deploy 2.0, as described under **Endpoint Considerations**.

- If you want to manually switch between different *SetParameters.xml* files in different locations, you may prefer to use MSDeploy.exe directly.
- If you want to override several MSDeploy.exe command-line arguments, you may prefer to use MSDeploy.exe directly.

When you use MSDeploy.exe, you need to provide three key pieces of information:

- A **–source** parameter that indicates where your data is coming from.
- A **–dest** parameter that indicates where your data is going to.
- A **–verb** parameter that indicates the operation you want to perform.

MSDeploy.exe relies on Web Deploy providers to process source and destination data. Web Deploy includes a lot of providers that represent the range of applications and data sources it can work with—for example, there are providers for SQL Server databases, IIS web servers, certificates, global assembly cache (GAC) assemblies, various different configuration files, and lots of other types of data. Both the **–source** parameter and the **–dest** parameter must specify a provider, in the form **–source**:[*providerName*]=[*location*]. When you're deploying a web package to an IIS website, you should use these values:

- The **–source** provider is always package. For example:

```
-source:package='[path to web package]'
```

- The **–dest** provider is always auto. For example:

```
-dest:auto='[server name or service URL]'
```

- The **–verb** is always **sync**.

```
-verb:sync
```

In addition, you'll need to specify various other provider-specific settings and general operation settings. For example, suppose you want to deploy the ContactManager.Mvc web application to a staging environment. The deployment will target the Web Deploy Handler and must use basic authentication. To deploy the web application, you need to complete the next steps.

To deploy a web application using MSDeploy.exe

1. Build and package the web application project, as described in Building and Packaging Web Application Projects.

2. Modify the *ContactManager.Mvc.SetParameters.xml* file to contain the correct parameter values for your staging environment, as described in Configuring Parameters for Web Package Deployment.

3. Open a Command Prompt window and browse to the location of MSDeploy.exe. This is typically at %PROGRAMFILES%\IIS\Microsoft Web Deploy V2\msdeploy.exe.

4. Type this command, and then press Enter (disregard the line breaks):

```
MSDeploy.exe
  -source:package="[path]\ContactManager.Mvc.zip"
  -dest:auto,
        computerName="https://stageweb1:8172/MSDeploy.axd?site=DemoSite",
        username="FABRIKAM\stagingdeployer",
        password="Pa$$w0rd",
        authtype="Basic",
        includeAcls="False"
  -verb:sync
  -disableLink:AppPoolExtension
  -disableLink:ContentExtension
  -disableLink:CertificateExtension
  -setParamFile:"[path]\ContactManager.Mvc.SetParameters.xml"
  -allowUntrusted
```

In this example:

- The **–source** parameter specifies the **package** provider and indicates the location of the web package.
- The **–dest** parameter specifies the **auto** provider. The **computerName** setting provides the service URL of the Web Deploy Handler on the destination server. The **authtype** setting indicates that you want to use basic authentication, and as such you need to provide a **username** and a **password**. Finally, the **includeAcls="False"** setting indicates that you don't want to copy the access control lists (ACLs) of the files in your source web application to the destination server.
- The **–verb:sync** argument indicates that you want to replicate the source content on the destination server.
- The **–disableLink** arguments indicate that you don't want to replicate application pools, virtual directory configuration, or Secure Sockets Layer (SSL) certificates on the destination server. For more information, see Web Deploy Link Extensions.
- The **–setParamFile** parameter provides the location of the *SetParameters.xml* file.
- The **–allowUntrusted** switch indicates that Web Deploy should accept SSL certificates that were not issued by a trusted certification authority. If you're deploying to the Web Deploy Handler, and you've used a self-signed certificate to secure the service URL, you need to include this switch.

Automating Web Package Deployment

In a lot of enterprise scenarios, you'll want to deploy your web packages as part of a larger single-step or automated deployment. Regardless of whether you choose to deploy your web packages by running the *.deploy.cmd* file or by using MSDeploy.exe directly, you can parameterize your commands and call them from a target in a Microsoft Build Engine (MSBuild) project file.

In the Contact Manager sample solution, take a look at the **PublishWebPackages** target in the *Publish.proj* file. This target runs once for each *.deploy.cmd* file identified by an item list named **PublishPackages**. The target uses properties and item metadata to build up a full set of argument values for each *.deploy.cmd* file and then uses the **Exec** task to run the command.

```
<Target Name="PublishWebPackages" Outputs="%(PublishPackages.Identity)">
  ...
  <PropertyGroup>
    <_WhatIfSwitch>/Y</_WhatIfSwitch>
    <_WhatIfSwitch Condition=" '$(_WhatIf)'=='true' ">/T</_WhatIfSwitch>
    <_Cmd>
      %(PublishPackages.FullPath) $(_WhatifSwitch) /M:$(MSDeployComputerName)
      /U:$(MSDeployUsername) /P:$(Password) /A:$(MSDeployAuth)
      %(PublishPackages.AdditionalMSDeployParameters)
    </_Cmd>
  </PropertyGroup>
  <Exec Command="$(_Cmd)"/>
</Target>
```

NOTE

For a broader overview of the project file model in the sample solution, and an introduction to custom project files in general, see Understanding the Project File and Understanding the Build Process.

Endpoint Considerations

Regardless of whether you deploy your web package by running the *.deploy.cmd* file or by using MSDeploy.exe directly, you need to specify a computer name or a service endpoint for your deployment.

If the destination web server is configured for deployment using the Web Deploy Remote Agent service, you specify the target service URL as your destination.

```
http://[server name]/MSDeployAgentService
```

Alternatively, you can specify the server name alone as your destination, and Web Deploy will infer the remote agent service URL.

```
[server name]
```

If the destination web server is configured for deployment using the Web Deploy Handler, you need to specify the endpoint address of the IIS Web Management Service (WMSvc) as your destination. By default, this takes the form:

```
https://[server name]:8172/MSDeploy.axd
```

You can target any of these endpoints using either the *.deploy.cmd* file or MSDeploy.exe directly. However, if you want to deploy to the Web Deploy Handler as a non-administrator user, as described in Configure a Web Server for Web Deploy Publishing (Web Deploy Handler), you need to add a query string to the service endpoint address.

```
https://[server name]:8172/MSDeploy.axd?site=[IIS website name]
```

This is because the non-administrator user doesn't have server-level access to IIS; he or she only has access to a specific IIS website. At the time of writing, due to a bug in the Web Publishing Pipeline (WPP), you can't run the *.deploy.cmd* file using an endpoint address that includes a query string. In this scenario, you need to deploy your web package by using MSDeploy.exe directly.

NOTE

For more information on the Web Deploy Remote Agent service and the Web Deploy Handler, see Choosing the Right Approach to Web Deployment. For guidance on how to configure your environment-specific project files to deploy to these endpoints, see Configure Deployment Properties for a Target Environment.

Authentication Considerations

Regardless of whether you deploy your web package by running the *.deploy.cmd* file or by using MSDeploy.exe directly, you need to specify an authentication type. Web Deploy accepts two possible values: **NTLM** or **Basic**. If you specify basic authentication, you also need to provide a user name and password. There are various factors you need to be aware of when you select an authentication type:

- If you're deploying to the Web Deploy Remote Agent service, you must use NTLM authentication. The remote agent service doesn't accept basic authentication credentials.
- If you're deploying to the Web Deploy Handler, you can use either NTLM or basic authentication. The default setting is basic authentication. Although basic authentication relies on user names and passwords being transmitted in plain text, your credentials are protected as the Web Deploy Handler always uses SSL encryption.
- If your web package includes a database, and the web server and database server are separate machines, you won't be able to deploy the database using NTLM authentication due to the NTLM "double-hop" limitation. You need to either use SQL Server credentials in your deployment connection string or supply basic authentication credentials to Web Deploy. This issue is described in more detail in Deploying Membership Databases to Enterprise Environments.

Conclusion

This topic described how you can deploy a web package either by running the *.deploy.cmd* file or by using MSDeploy.exe directly. It explained when each approach might be appropriate, and it described how you can parameterize and run a deployment command as part of a larger single-step or automated build process.

Further Reading

For guidance on how to create and parameterize a web deployment package, see Building and Packaging Web Application Projects and Configuring Parameters for Web Package Deployment. For guidance on how to build and deploy web packages from a Team Foundation Server (TFS) instance, see Configuring Team Foundation Server for Automated Web Deployment. For information on how to customize and troubleshoot the deployment process, see Excluding Files and Folders from Deployment.

PREVIOUS NEXT

Deploying Database Projects

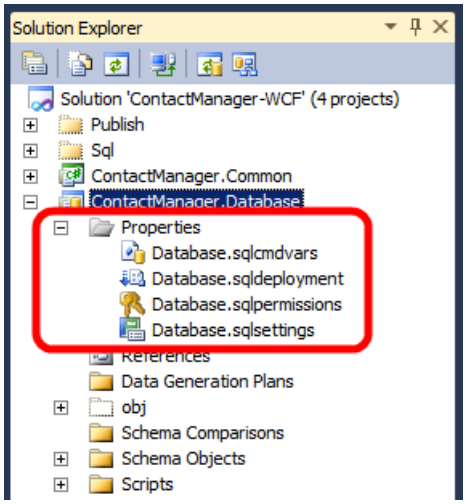
2/20/2020 • 9 minutes to read • Edit Online

by Jason Lee

Download PDF

> **NOTE**
>
> In lots of enterprise deployment scenarios, you need the ability to publish incremental updates to a deployed database. The alternative is to recreate the database on every deployment, which means you lose any data in the existing database. When you work with Visual Studio 2010, using VSDBCMD is the recommended approach to incremental database publishing. However, the next version of Visual Studio and the Web Publishing Pipeline (WPP) will include tooling that supports incremental publishing directly.

If you open the Contact Manager sample solution in Visual Studio 2010, you'll see that the database project includes a Properties folder that contains four files.



Together with the project file (*ContactManager.Database.dbproj* in this case), these files control various aspects of the build and deployment process:

- The *Database.sqlcmdvars* file provides values for any SQLCMD variables you use when you deploy the project. Each solution configuration (for example, debug and release) can specify a different .sqlcmdvars file.
- The *Database.sqldeployment* file provides deployment-specific settings, like whether to use the collation defined in your project or the collation of the destination server, whether to recreate the destination database every time or simply amend the existing database to bring it up to date, and so on. Each solution configuration can specify a different .sqldeployment file.
- The *Database.sqlpermissions* file is an XML document that you can use to define any permissions you want to add to the target database. All solution configurations share the same .sqlpermissions file.
- The *Database.sqlsettings* file specifies the database-level properties to use when creating the database, like the collation to use, the behavior of comparison operators, and so on. All solution configurations share the same .sqlsettings file.

It's worth taking a moment to open these files in Visual Studio and familiarize yourself with the contents.

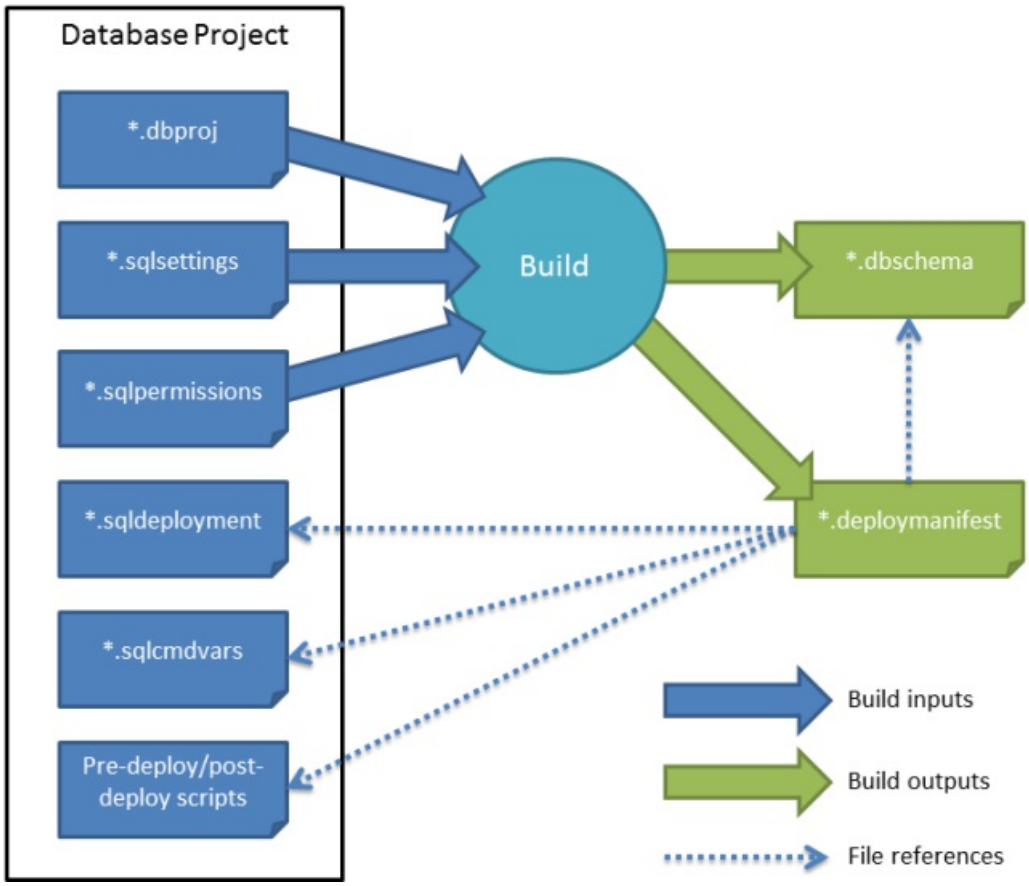
When you build a database project, the build process creates two files:

- A *database schema* (.dbschema file). This describes the schema of the database you want to create in XML

format.

- A *deployment manifest* (.deploymanifest file). This contains all the information required to create and deploy your database. It references the .dbschema file along with other resources, like the deployment instructions (the .sqldeployment file) and any pre-deployment or post-deployment SQL scripts.

This shows the relationship between these resources:



As you can see, the .sqlsettings file and the .sqlpermissions file are inputs to the build process. Along with the database project file, these files are used to create the database schema file. The .sqldeployment file and the .sqlcmdvars file pass through the build process unchanged. The deployment manifest indicates the location of the database schema, the .sqldeployment file, the .sqlcmdvars file, and any pre-deployment or post-deployment SQL scripts.

Why Use VSDBCMD to Deploy a Database Project?

There are various different approaches to deploying database projects. However, not all of them are suitable for deploying a database project to remote servers in an enterprise environment. Consider what you want from a database project deployment. In enterprise deployment scenarios, you're likely to want:

- The ability to deploy the database project from a remote location.
- The ability to make incremental updates to an existing database.
- The ability to include pre-deployment scripts or post-deployment scripts.
- The ability to tailor the deployment to multiple destination environments.
- The ability to deploy the database project as part of a larger, typically scripted, single-step solution deployment.

There are three main approaches you can use to deploy a database project:

- You can use the deployment functionality with the database project type in Visual Studio 2010. When you build and deploy a database project in Visual Studio 2010, the deployment process uses the deployment manifest to generate a SQL-based deployment file specific to the build configuration. This will create the

database if it doesn't already exist or make any necessary changes to the database if it does already exist. You can use SQLCMD.exe to run this file on your destination server, or you can set Visual Studio to create and run the file. The disadvantage of this approach is that you have only limited control over the deployment settings. You may often also need to modify the SQL deployment file to provide environment-specific variable values. You can only use this approach from a computer with Visual Studio 2010 installed, and the developer would need to know and provide connection strings and credentials for all destination environments.

- You can use the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) to deploy a database as part of a web application project. However, this approach is a lot more complex if you want to deploy a database project rather than simply replicate an existing local database on a destination server. You can configure Web Deploy to run the SQL deployment script that the database project generates, but in order to do this, you need to create a custom WPP targets file for your web application project. This adds a substantial amount of complexity to the deployment process. In addition, Web Deploy does not directly support incremental updates to existing databases. For more information on this approach, see Extending the Web Publishing Pipeline to package database project deployed SQL file.
- You can use the VSDBCMD utility to deploy the database, using either the database schema or the deployment manifest. You can call VSDBCMD.exe from an MSBuild target, which lets you publish databases as part of a larger, scripted deployment process. You can override the variables in your .sqlcmdvars file and lots of other database properties from a VSDBCMD command, which allows you to customize your deployment for different environments without creating multiple build configurations. VSDBCMD provides differentiation functionality, which means it will make only the necessary changes to align a destination database with your database schema. VSDBCMD also offers a wide range of command-line options, which give you fine-grained control over the deployment process.

From this overview, you can see that using VSDBCMD with MSBuild is the approach best suited to a typical enterprise deployment scenario:

	VISUAL STUDIO 2010	WEB DEPLOY 2.0	VSDBCMD.EXE
Supports remote deployment?	Yes	Yes	Yes
Supports incremental updates?	Yes	No	Yes
Supports pre/post-deployment scripts?	Yes	Yes	Yes
Supports multi-environment deployment?	Limited	Limited	Yes
Supports scripted deployment?	Limited	Yes	Yes

The remainder of this topic describes the use of VSDBCMD with MSBuild to deploy database projects.

Understanding the Deployment Process

The VSDBCMD utility lets you deploy a database using either the database schema (the .dbschema file) or the deployment manifest (the .deploymanifest file). In practice, you'll almost always use the deployment manifest, as the deployment manifest lets you provide default values for various deployment properties and identify any pre-deployment or post-deployment SQL scripts you want to run. For example, this VSDBCMD command is used to deploy the **ContactManager** database to a database server in a test environment:

```
vsdbcmd.exe /a:Deploy
            /manifest:"…\ContactManager.Database.deploymanifest"
            /cs:"Data Source=TESTDB1;Integrated Security=true"
            /p:TargetDatabase=ContactManager
            /dd+
            /script:"…\Publish-ContactManager-Db.sql"
```

In this case:

- The **/a** (or **/Action**) switch specifies what you want VSDBCMD to do. You can set this to **Import** or **Deploy**. The **Import** option is used to generate a .dbschema file from an existing database, and the **Deploy** option is used to deploy a .dbschema file to a target database.
- The **/manifest** (or **/ManifestFile**) switch identifies the .deploymanifest file you want to deploy. If you wanted to use the .dbschema file instead, you'd use the **/model** (or **/ModelFile**) switch.
- The **/cs** (or **/ConnectionString**) switch provides the connection string for the target database server. Note that this doesn't include the name of the database—VSDBCMD needs to connect to the server to create the database; it doesn't need to connect to an individual database. If your .deploymanifest file includes a connection string, you can omit this switch. If you use the switch anyway, the switch value will override the .deploymanifest value.
- The **/p:TargetDatabase** property provides the name you want to assign to the target database on creation. This overrides the value of the **TargetDatabase** property in the .deploymanifest file. You can use the **/p:** *[property name]*syntax to set a wide variety of deployment properties and to override any SQLCMD variables declared in your .sqlcmdvars file.
- The **/dd+** (or **/DeployToDatabase+**) switch indicates that you want to create a deployment and deploy it to the target environment. If you specify **/dd-**, or omit the switch, VSDBCMD will generate a deployment script but will not deploy it to the target environment. This switch is often the source of confusion and is explained in more detail in the next section.
- The **/script** (or **/DeploymentScriptFile**) switch specifies where you want to generate the deployment script. This value does not affect the deployment process.

For more information on VSDBCMD, see Command-Line Reference for VSDBCMD.EXE (Deployment and Schema Import) and How to: Prepare a Database for Deployment From a Command Prompt by Using VSDBCMD.EXE.

For an example of how you can use VSDBCMD from an MSBuild project file, see Understanding the Build Process. For examples of how to configure database deployment settings for multiple environments, see Customizing Database Deployments for Multiple Environments.

Understanding the DeployToDatabase Switch

The behavior of the **/dd** or **/DeployToDatabase** switch depends on whether you're using VSDBCMD with a .dbschema file or a .deploymanifest file. If you're using a .dbschema file, the behavior is fairly straightforward:

- If you specify **/dd+** or **/dd**, VSDBCMD will generate a deployment script and deploy the database.
- If you specify **/dd-** or omit the switch, VSDBCMD will generate a deployment script only.

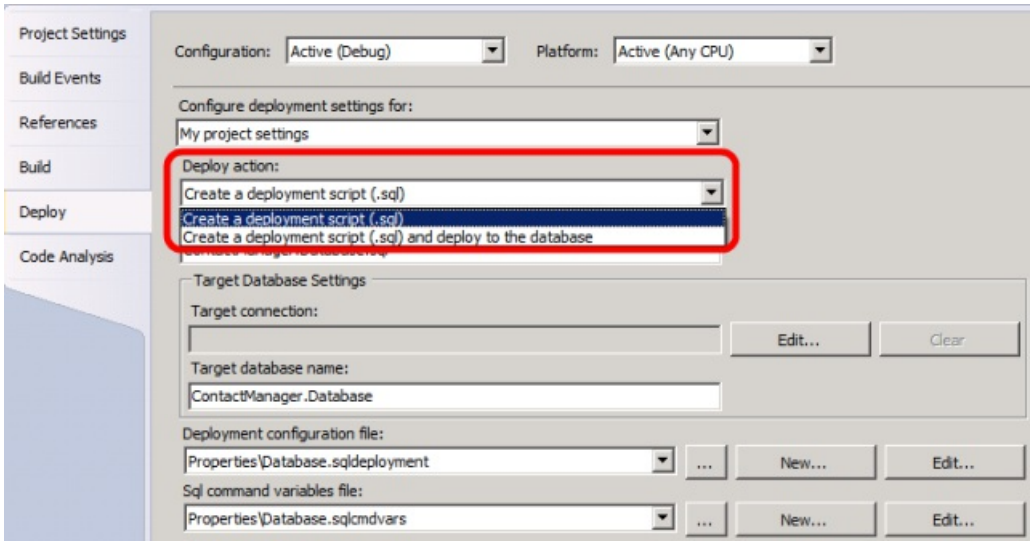
If you're using a .deploymanifest file, the behavior is a lot more complicated. This is because the .deploymanifest file contains a property name **DeployToDatabase** that also determines whether the database is deployed.

```
<DeployToDatabase>False</DeployToDatabase>
```

The value of this property is set according to the properties of the database project. If you set the **Deploy action** to **Create a deployment script (.sql)**, the value will be **False**. If you set the **Deploy action** to **Create**

a deployment script (.sql) and deploy to the database, the value will be **True**.

> **NOTE**
>
> These settings are associated with a specific build configuration and platform. For example, if you configure settings for the **Debug** configuration and then publish using the **Release** configuration, your settings will not be used.



> **NOTE**
>
> In this scenario, the **Deploy action** should always be set to **Create a deployment script (.sql)**, because you don't want Visual Studio 2010 to deploy your database. In other words, the **DeployToDatabase** property should always be **False**.

When a **DeployToDatabase** property is specified, the **/dd** switch will only override the property if the property value is **false**:

- If the **DeployToDatabase** property is **False**, and you specify **/dd+** or **/dd**, VSDBCMD will override the **DeployToDatabase** property and deploy the database.
- If the **DeployToDatabase** property is **False**, and you specify **/dd-** or omit the switch, VSDBCMD will not deploy the database.
- If the **DeployToDatabase** property is **True**, VSDBCMD will ignore the switch and deploy the database.
- A deployment script is generated in each case, regardless of whether you're deploying the database as well.

Conclusion

This topic provided an overview of the build and deployment process for database projects in Visual Studio 2010. It also described how you can use VSDBCMD.exe with MSBuild to support enterprise-scale database deployment.

For more information on how this works in practice, see Customizing Database Deployments for Multiple Environments.

Further Reading

For information on how to customize database deployments by creating a separate deployment configuration file for each environment, see Customizing Database Deployments for Multiple Environments. For guidance on how to configure database role memberships by running a post-deployment script, see Deploying Database Role Memberships to Test Environments. For guidance on managing some of the unique challenges that

membership databases impose, see Deploying Membership Databases to Enterprise Environments.

These topics on MSDN provide broader guidance and background information on Visual Studio database projects and the database deployment process:

- Visual Studio 2010 SQL Server Database Projects
- Managing Database Change
- How to: Prepare a Database for Deployment From a Command Prompt by Using VSDBCMD.EXE
- An Overview of Database Build and Deployment

PREVIOUS NEXT

Creating and Running a Deployment Command File

2/20/2020 • 4 minutes to read • Edit Online

by Jason Lee

Download PDF

> This topic describes how to build a command file that will let you run a deployment using Microsoft Build Engine (MSBuild) project files as a single-step, repeatable process.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Build Process, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Process Overview

In this topic, you'll learn how to create and run a command file that uses these project files to perform a repeatable deployment to your target environment. Essentially, the command file simply needs to contain an MSBuild command that:

- Tells MSBuild to execute the environment-agnostic *Publish.proj* file.
- Tells the *Publish.proj* file which file contains the environment-specific project settings and where to find it.

Create an MSBuild Command

As described in Understanding the Build Process, the environment-specific project file—for example, *Env-Dev.proj*—is designed to be imported into the environment-agnostic *Publish.proj* file at build time. Together, these two files provide a complete set of instructions that tell MSBuild how to build and deploy your solution.

The *Publish.proj* file uses an **Import** element to import the environment-specific project file.

```
<Import Project="$(TargetEnvPropsFile)"/>
```

As such, when you use MSBuild.exe to build and deploy the Contact Manager solution, you need to:

- Run MSBuild.exe on the *Publish.proj* file.
- Specify the location of the environment-specific project file by supplying a command-line parameter named **TargetEnvPropsFile**.

To do this, your MSBuild command should resemble this:

```
msbuild.exe Publish.proj /p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
```

From here, it's a simple step to move to a repeatable, single-step deployment. All you need to do is to add your MSBuild command to a .cmd file. In the Contact Manager solution, the Publish folder includes a file named *Publish-Dev.cmd* that does exactly this.

```
%windir%\Microsoft.NET\Framework\v4.0.30319\msbuild.exe Publish.proj /fl /p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
echo
pause
```

> **NOTE**
>
> The **/fl** switch instructs MSBuild to create a log file named *msbuild.log* in the working directory in which MSBuild.exe was invoked.

To deploy or redeploy the Contact Manager solution, all you need to do is run the *Publish-Dev.cmd* file. When you run the file, MSBuild will:

- Build all the projects in the solution.
- Generate deployable web packages for the web application projects.
- Generate .dbschema and .deploymanifest files for the database projects.
- Deploy the web packages to the web server.
- Deploy the database to the database server.

Run the Deployment

When you've created a command file for your target environment, you should be able to complete the entire deployment by simply running the file.

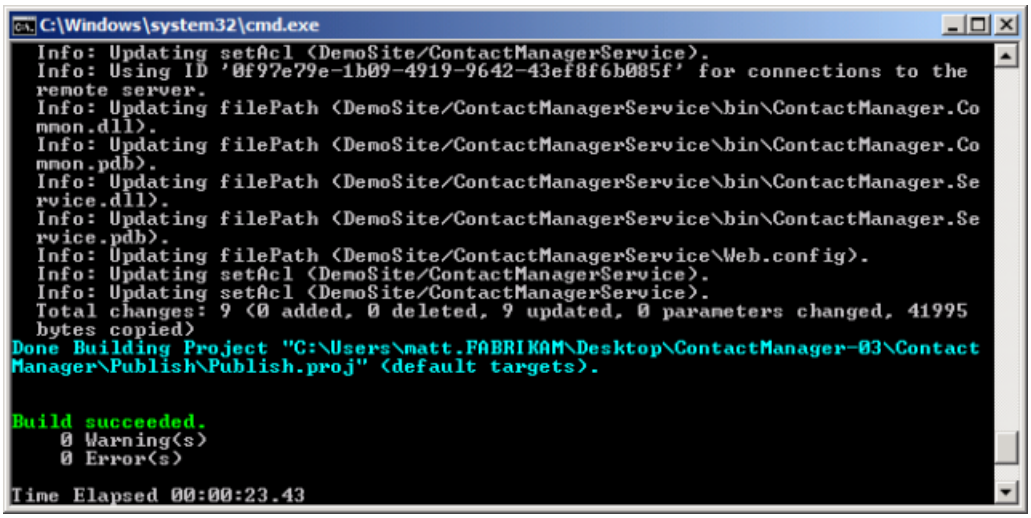
To deploy the Contact Manager solution to your test environment

1. On your developer workstation, open Windows Explorer, and then browse to the location of the *Publish-Dev.cmd* file.

2. Double-click the file to run it.

3. If an **Open File – Security Warning** dialog box appears, click **Run**.

4. If your configuration settings and test servers are set up correctly, the Command Prompt window will show a **Build succeeded** message when MSBuild has finished processing the project files.



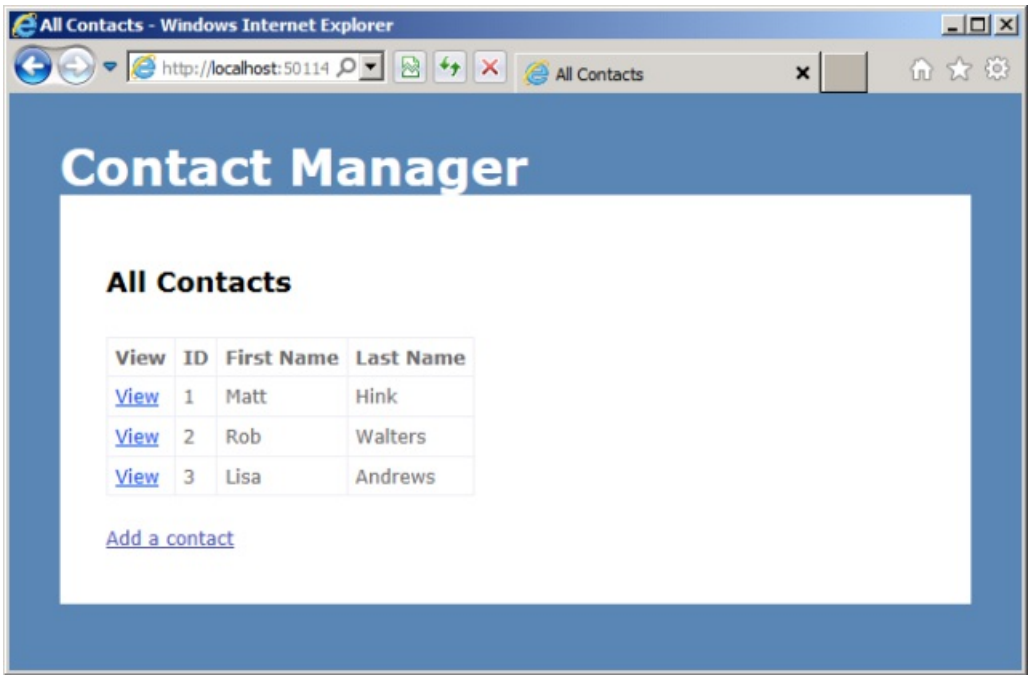
5. If this is the first time you've deployed the solution to this environment, you'll need to add the test web

server machine account to the **db_datawriter** and **db_datareader** roles on the **ContactManager** database. This procedure is described in Configure a Database Server for Web Deploy Publishing.

> **NOTE**
> You only need to assign these permissions when you create the database. By default, the build process will not recreate the database on every deployment—instead, it will compare the existing database to the latest schema and make only the changes required. As a result, you should only need to map these database roles the first time you deploy the solution.

6. Open Internet Explorer and browse to the URL of the Contact Manager application (for example, `http://testweb1:85/ContactManager/`).

7. Verify that the application works as expected and you're able to add contacts.



Conclusion

Creating a command file containing your MSBuild instructions provides you with a quick and easy way of building and deploying a multi-project solution to a specific destination environment. If you need to repeatedly deploy your solution to multiple destination environments, you can create multiple command files. In each command file, the MSBuild command will build the same universal project file, but it will specify a different environment-specific project file. For example, a command file to publish to a developer or test environment might contain this MSBuild command:

```
msbuild.exe Publish.proj /p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
```

A command file to publish to a staging environment might contain this MSBuild command:

```
msbuild.exe Publish.proj /p:TargetEnvPropsFile=EnvConfig\Env-Stage.proj
```

NOTE

For guidance on how to customize the environment-specific project files for your own server environments, see Configure Deployment Properties for a Target Environment.

You can also customize the build process for each environment by overriding properties or setting various other switches in your MSBuild command. For more information, see MSBuild Command Line Reference.

PREVIOUS NEXT

Manually Installing Web Packages

2/20/2020 • 7 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic describes how to manually import a web deployment package into Internet Information Services (IIS).

The topic Building and Packaging Web Application Projects described how the IIS Web Deployment Tool (Web Deploy), in conjunction with the Microsoft Build Engine (MSBuild) and the Web Publishing Pipeline (WPP), lets you package your web application projects into a single zip file. This file, commonly known as a web deployment package (or simply a deployment package), contains all the content and configuration information that IIS needs in order to re-create your web application on a web server.

Once you've created a web deployment package, you can publish it to an IIS server in various ways. In a lot of scenarios, you'll want to take advantage of the integration points between MSBuild, the WPP, and Web Deploy to create and install web packages remotely as part of an automated or single-step build and deployment process. This process is described in Deploying Web Packages. However, this isn't always possible. Suppose you want to deploy a web application to an Internet-facing production environment. For security reasons, such a production environment is at the very least likely to be behind a firewall on a subnet that is separate from the build server, in a perimeter network (also known as DMZ, demilitarized zone, and screened subnet). In lots of cases, the production environment will be on a separate domain or on a physically isolated network.

In these scenarios, your only option may be to port the web package onto the destination server and manually import it into IIS. Although this approach precludes automated deployment, it's still a highly effective technique for publishing a web application—you simply copy a single zip file to your web server and use a wizard to guide you through the import process.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

Task Overview

You'll need to complete these high-level tasks to import a web deployment package into IIS:

- Create a web deployment package using the MSBuild command line, Team Build, or Visual Studio 2010.
- Copy the web package to the destination web server.
- Use the Import Application Package Wizard in IIS Manager to install the web package and provide values for variables like connection strings and service endpoints.

This topic will show you how to perform these procedures. The tasks and walkthroughs in this topic assume that you're already familiar with the concepts behind web packages, Web Deploy, and the WPP. For more information, see Building and Packaging Web Application Projects.

> **NOTE**
>
> This topic is best used in conjunction with Configure a Web Server for Web Deploy Publishing (Offline Deployment), which explains how to install the required components and prepare an IIS website for package import.

Create a Web Deployment Package

The first task is to create a web deployment package for the web application project you want to deploy. You can create web packages in a variety of ways.

Approach 1: Create a package as part of the build process with Visual Studio

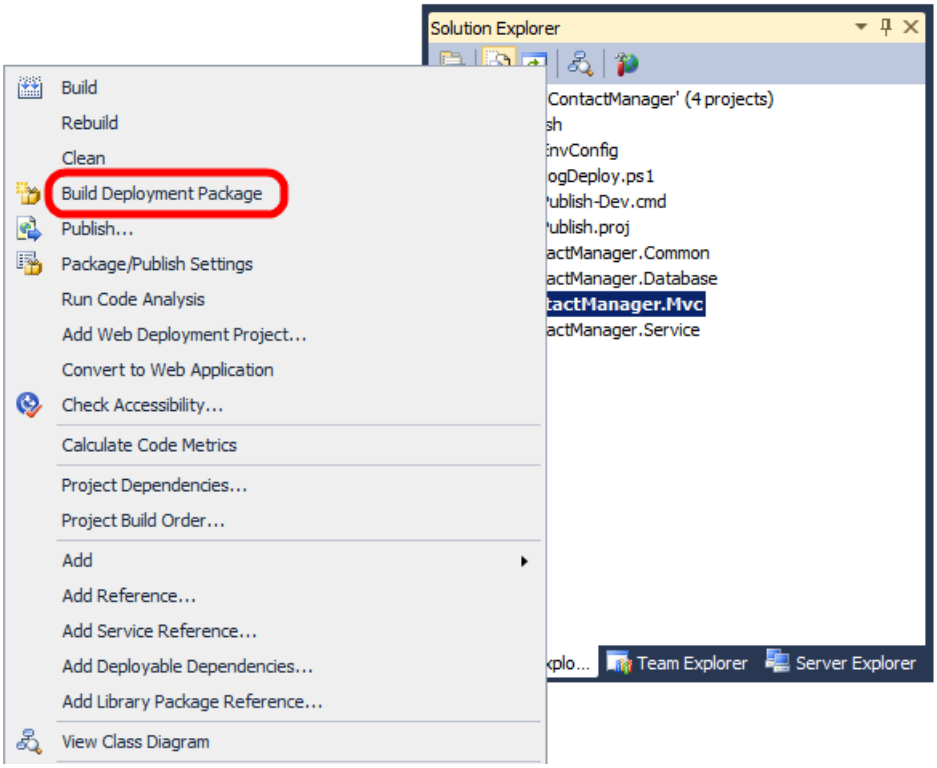
You can configure your web application project to create a web deployment package after every build through the **Package/Publish Web** tab on the project property pages. This process is described in Building and Packaging Web Application Projects.

Approach 2: Create a package as part of the build process with MSBuild

If you build your web application project by using MSBuild directly, either through a custom MSBuild project file or from the command line, you can create a web deployment package as part of the build process by including the **DeployOnBuild=true** and **DeployTarget=Package** properties in your command. This process is described in Understanding the Build Process.

Approach 3: Create a package on demand in Visual Studio

You can create a web deployment package for a web application project at any time in Visual Studio 2010. To do this, in the **Solution Explorer** window, right-click your web application project, and then click **Build Deployment Package**.



Approach 4: Create a package on demand from the command line

You can create a web deployment package from the command line by invoking the **Package** target on your web application project using MSBuild. The command should resemble this:

```
MSBuild.exe [Path to your project].[csproj/vbproj] /T:Package
```

Whichever approach you use, the end result is the same. The WPP creates a web deployment package as a zip file, together with various supporting resources, in the output folder for your web application project.

PackageTmp	16/02/2012 18:59	File folder	
ContactManager.Mvc.deploy.cmd	16/02/2012 19:00	Windows Command Script	14 KB
ContactManager.Mvc.deploy-readme.txt	16/02/2012 19:00	Text Document	4 KB
ContactManager.Mvc.SetParameters.xml	16/02/2012 19:00	XML Document	1 KB
ContactManager.Mvc.SourceManifest.xml	16/02/2012 19:00	XML Document	1 KB
ContactManager.Mvc.zip	16/02/2012 19:00	Compressed (zipped) Folder	554 KB

When you're planning to import the web package manually, you require only the zip file. Copy this file to your target web server and you can begin the import process.

Import a Web Package into IIS

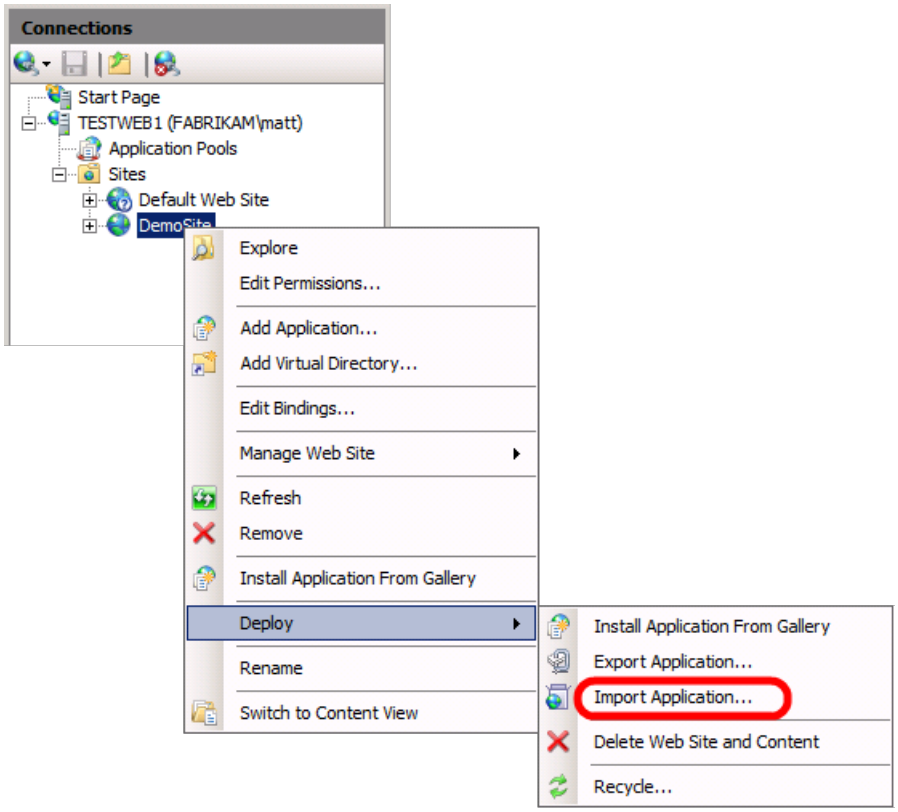
You can use the next procedure to import a web deployment package from the local file system into an IIS website. Before you perform this procedure, ensure that you have:

- Copied the web deployment package to the web server.
- Configured an IIS web server to host your application.

For more information on configuring an IIS web server to support web deployment packages, see Configure a Web Server for Web Deploy Publishing (Offline Deployment).

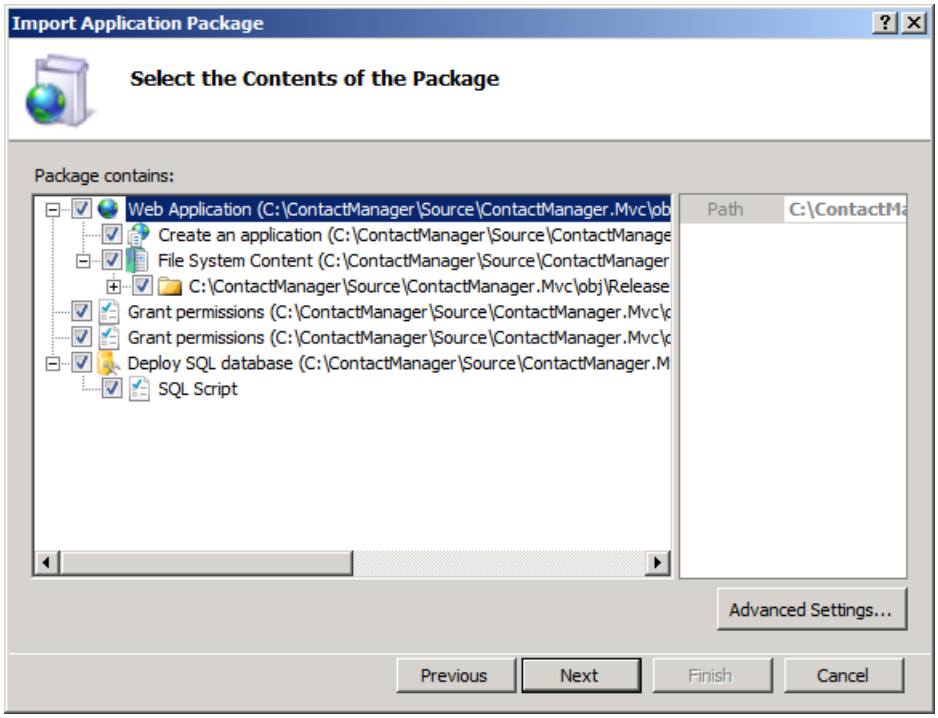
To import a web deployment package using IIS Manager

1. In IIS Manager, in the **Connections** pane, right-click your IIS website, point to **Deploy**, and then click **Import Application**.



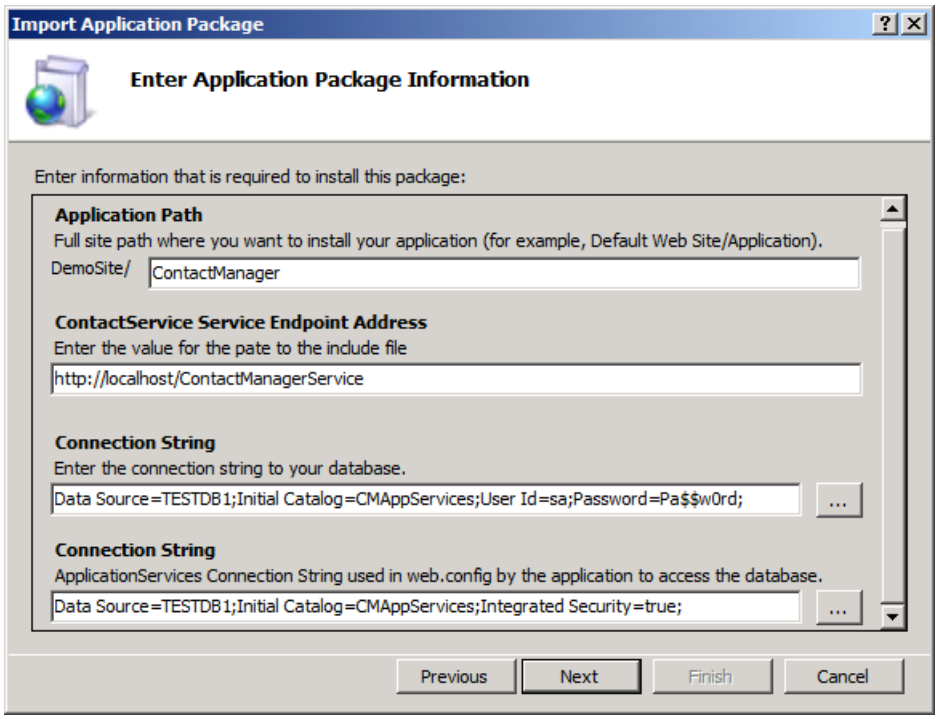
2. In the Import Application Package Wizard, on the **Select the Package** page, browse to the location of your web deployment package, and then click **Next**.

3. On the **Select the Contents of the Package** page, clear any content that you don't require, and then click **Next**.



> **NOTE**
>
> In a lot of cases, you may not want to import everything that comes with a web deployment package. For example, you may not want to allow Web Deploy to replace the associated database.
>
> The **Grant permissions** entries set permissions on the destination file system to ensure that the application pool identity can access the physical folder that stores the website content. In addition, the anonymous authentication user is granted read permission to the folder to let the application serve Multipurpose Internet Mail Extensions (MIME) type files. If you prefer, you can remove these entries and configure permissions manually.

4. On the **Enter Application Package Information** page, provide the requested information.



5. When you create a web package, the WPP analyzes the configuration file for your application and detects any variables, like connection strings and service endpoints. In this case:

a. **Application Path** is the IIS path where you want to install your application. This setting is common to all deployment packages that the WPP creates.

b. **ContactService Service Endpoint Address** is the address that the application should use to communicate with the deployed WCF service. This setting corresponds to an entry in the *web.config* file.

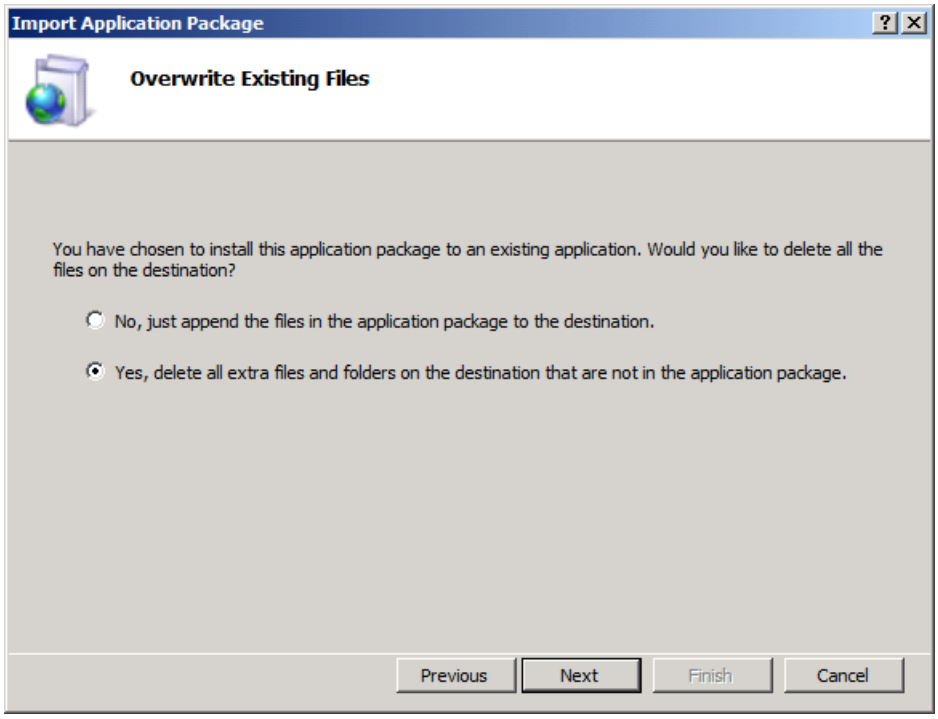
c. The first **Connection String** setting is the connection string that Web Deploy should use to deploy the database associated with the application (in this case an ASP.NET membership database). This setting corresponds to the setting on the **Package/Publish SQL** tab in Visual Studio.

d. The second **Connection String** setting is the connection string that your application will actually use to communicate with the database when it's up and running. This corresponds to a connection string entry in the *web.config* file.

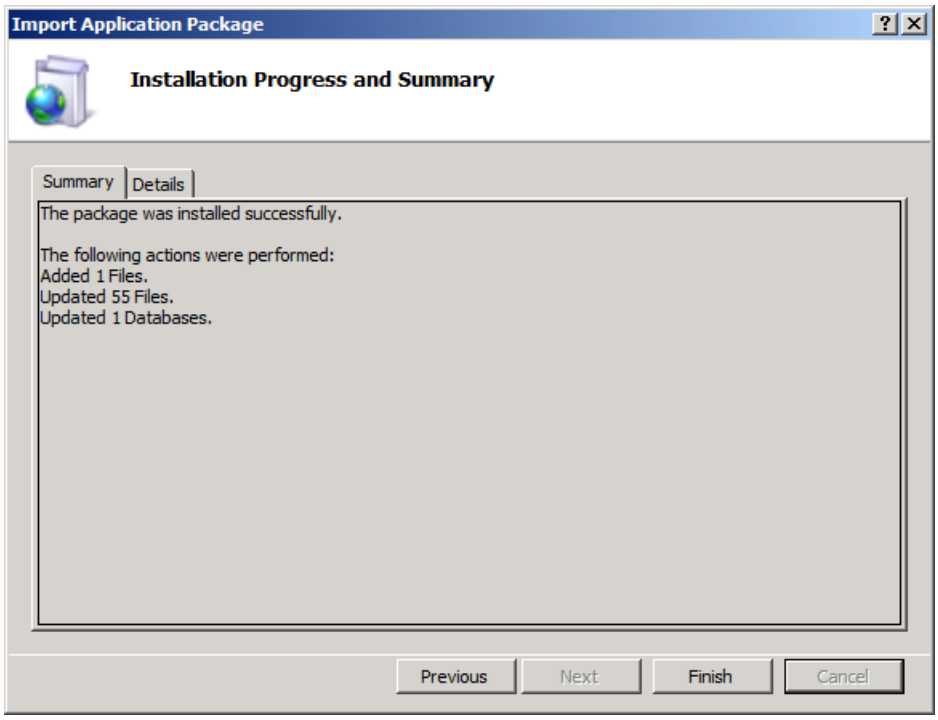
> **NOTE**
>
> For more information on where these parameters come from, see Configuring Parameters for Web Package Deployment.

6. Click **Next**.

7. If this is not the first time you've deployed the application to this website, you'll be prompted to specify whether you want to delete all existing content prior to installation. Choose the option that's appropriate for your requirements, and then click **Next**.



8. When IIS has finished installing the package, click **Finish**.



At this point, you've successfully published your web application to IIS.

Conclusion

This topic described how to import a web deployment package into an IIS website using IIS Manager. This approach to web application publishing is appropriate when security or infrastructure constraints make remote deployment impossible or undesirable.

Further Reading

For guidance on how to configure an IIS web server to support manually importing a web package, see Configure a Web Server for Web Deploy Publishing (Offline Deployment). For more general guidance on deploying web packages, see Walkthrough: Deploying a Web Application Project Using a Web Deployment Package (Part 1 of 4).

PREVIOUS

ASP.NET 4 - Enterprise Deployment Series 2 Configuring Servers

2/20/2020 • 2 minutes to read • Edit Online

This tutorial will show you how to set up server environments to support one-click, or automated, website deployment and publishing in various different scenarios.

- Configuring Server Environments for Web Deployment
- Choosing the Right Approach to Web Deployment
- Scenario: Configuring a Test Environment for Web Deployment
- Scenario: Configuring a Staging Environment for Web Deployment
- Scenario: Configuring a Production Environment for Web Deployment
- Configuring a Web Server for Web Deploy Publishing (Remote Agent)
- Configuring a Web Server for Web Deploy Publishing (Web Deploy Handler)
- Configuring a Web Server for Web Deploy Publishing (Offline Deployment)
- Configuring a Database Server for Web Deploy Publishing
- Creating a Server Farm with the Web Farm Framework
- Configuring Deployment Properties for a Target Environment

Configuring Server Environments for Web Deployment

2/20/2020 • 2 minutes to read • Edit Online

by Jason Lee

Download PDF

> This tutorial will show you how to set up server environments to support one-click, or automated, website deployment and publishing in various different scenarios. The tutorial includes topics to walk you through completing various tasks, like configuring a web server to support specific approaches to deployment and setting up a Web Farm Framework (WFF) server farm, together with scenario-based overviews that provide higher-level end-to-end guidance.
>
> The tutorial uses the Fabrikam, Inc. deployment scenario described in Enterprise Web Deployment: Scenario Overview as a reference point for examples and network infrastructure.
>
> For an Italian translation of these tutorials, visit http://www.lucamorelli.it.

This tutorial includes these topics:

- Choosing the Right Approach to Web Deployment
- Scenario: Configuring a Test Environment for Web Deployment
- Scenario: Configuring a Staging Environment for Web Deployment
- Scenario: Configuring a Production Environment for Web Deployment
- Configuring a Web Server for Web Deploy Publishing (Remote Agent)
- Configuring a Web Server for Web Deploy Publishing (Web Deploy Handler)
- Configuring a Web Server for Web Deploy Publishing (Offline Deployment)
- Configuring a Database Server for Web Deploy Publishing
- Creating a Server Farm with the Web Farm Framework
- Configuring Deployment Properties for a Target Environment

The first topic, Choosing the Right Approach to Web Deployment, describes the main approaches you can use to publish web applications by using the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) 2.0. It also identifies the scenarios that map to each approach. From here, each scenario topic provides a high-level overview of the tasks you need to complete and identifies the topics you'll need to work through to help you complete these tasks.

If you're using the split project file approach described in Understanding the Build Process to build and deploy your solution, the final topic, Configuring Deployment Properties for a Target Environment, describes how to configure environment-specific project files for deployment to different destination environments.

Key Technologies

This tutorial focuses on how to use these products and technologies to support web deployment:

- IIS 7.5
- Web Deploy 2.x
- WFF 2.x

- IIS Web Management Service (WMSvc)

The tutorial also touches on the use of Windows Server 2008 R2, SQL Server 2008 R2, ASP.NET 4.0, and ASP.NET MVC 3.

Other Tutorials in This Series

This forms part of a series of five tutorials on enterprise-scale web deployment. These are the other tutorials in the series:

- Deploying Web Applications in Enterprise Scenarios. This introductory content provides the contextual background for the tutorial series. It describes the tutorial scenario, and it illustrates how the tasks and walkthroughs described throughout the series fit into a broader Application Lifecycle Management (ALM) process.
- Web Deployment in the Enterprise. This tutorial provides a conceptual introduction to Microsoft Build Engine (MSBuild) project files, the Web Publishing Pipeline, Web Deploy, and other related technologies. It explains how you can use these tools together to manage complex deployment processes.
- Configuring Team Foundation Server for Web Deployment. This tutorial describes how to configure Team Foundation Server (TFS) to support various deployment scenarios, including automated deployment as part of a continuous integration (CI) process and manually triggered deployments of specific builds.
- Advanced Enterprise Web Deployment. This tutorial describes how to accomplish various more advanced deployment tasks, like customizing database deployments for multiple environments, excluding files and folders from deployment, and taking web applications offline during the deployment process.

NEXT

Choosing the Right Approach to Web Deployment

2/20/2020 • 7 minutes to read • Edit Online

by Jason Lee

Download PDF

When you work with the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) 2.0 or later, there are three main approaches you can use to get your packaged web applications onto a web server. You can either:

- Deploy the application from a remote location by targeting the *Web Deployment Agent Service* (also known as the "remote agent") on the destination server.
- Deploy the application from a remote location using Web Deploy On Demand (also known as the "temp agent").
- Deploy the application from a remote location by targeting the *IIS Web Deploy Handler* on the destination server.
- Deploy the application by manually copying the web package to the destination server and importing it through IIS Manager.

How you configure your destination web servers will depend on which approach to deployment you want to use. This topic will help you decide which approach to deployment is right for you.

This table shows the main advantages and disadvantages of each deployment approach, together with the scenarios that most typically suit each approach.

APPROACH	ADVANTAGES	DISADVANTAGES	TYPICAL SCENARIOS
Remote Agent	It is easy to set up. It is suitable for regular updates to web applications and content.	The user must be an administrator on the target server. the user can't supply alternative credentials.	Development environments. Test environments.
Temp Agent	There is no need to install Web Deploy on the target computer. The latest version of Web Deploy is automatically used.	The user must be an administrator on the target server. The user can't supply alternative credentials.	Development environments. Test environments.
Web Deploy Handler	Non-administrator users can deploy content. It is suitable for regular updates to web applications and content.	It is a lot more complex to set up.	Staging environments. Intranet production environments. Hosted environments.
Offline Deployment	It is very easy to set up. It is suitable for isolated environments.	The server administrator must manually copy and import the web package every time.	Internet-facing production environments. Isolated network environments.

Using the Remote Agent

When you install Web Deploy using the default settings on a destination server, the Web Deployment Agent Service (the "remote agent") is automatically installed and started. By default, the remote agent exposes an HTTP endpoint at this address:

```
http://[server]/MSDEPLOYAGENTSERVICE
```

> **NOTE**
> You can replace [*server*] with the machine name of your web server, an IP address for your web server, or a hostname that resolves to your web server.

Server administrators can deploy web packages from a remote location, like a developer machine or a build server, by specifying this endpoint address. For example, suppose Matt Hink at Fabrikam, Inc. has built the ContactManager.Mvc web application project on his developer machine. The build process generates a web package, together with a *.deploy.cmd* file that contains the Web Deploy commands required to install the package. If Matt is a server administrator on the TESTWEB1 server, he can deploy the web application to the test web server by running this command on his developer machine:

```
ContactManager.Mvc.deploy.cmd /y /m:http://TESTWEB1/MSDEPLOYAGENTSERVICE a/:NTLM
```

In actual fact, the Web Deploy executable can infer the endpoint address of the remote agent if you provide the machine name, so Matt only needs to type this:

```
ContactManager.Mvc.deploy.cmd /y /m:TESTWEB1 /a:NTLM
```

> **NOTE**
> For more information on Web Deploy command-line syntax and *.deploy.cmd* files, see How to: Install a Deployment Package Using the deploy.cmd File.

The remote agent offers a straightforward way to deploy content from a remote location, and this approach can work well with one-click or automated deployment. However, the user who runs the deployment command must also be either a domain administrator or a member of the local administrators group on the destination server. In addition, the remote agent doesn't support basic authentication, so you can't pass alternative credentials on the command line.

The remote agent provides a useful approach to deployment in development or test scenarios, where it's not uncommon for developers to have full administrator control over a test server environment, and applications are typically rebuilt and redeployed very frequently. However, this approach is usually less acceptable for staging or production environments.

For an end-to-end example of a scenario that uses the remote agent approach, see Scenario: Configuring a Test Environment for Web Deployment.

Using the Temp Agent

The temp agent approach to deployment is similar to the remote agent approach. However, in contrast to the remote agent approach, you don't need to install Web Deploy on the destination web server. Instead, when you perform the deployment, Web Deploy will install a temporary version of the web deployment agent service on the destination server and will use this to deploy your content to IIS. When the deployment is complete, all temporary files are removed.

If you want to use the temp agent provider setting, add the **/g** flag to your deployment command:

```
ContactManager.Mvc.deploy.cmd /y /m:TESTWEB1 /g:true
```

> **NOTE**
> You can't use the temp agent if the web deployment agent service is installed on the destination computer, even if the service is not running.

The advantage of this approach is that you don't need to maintain installations of Web Deploy on your destination servers. Furthermore, you don't need to ensure that the source and destination computers are running the same version of Web Deploy. However, this approach suffers from the same principal limitations as the remote agent approach, namely that you must be a local administrator on the destination server in order to deploy content, and only NTLM authentication is supported. The temp agent approach also requires a lot more initial configuration of the destination environment.

For more information on using the temp agent, see How to: Install a Deployment Package Using the deploy.cmd File and Web Deploy On Demand.

Using the Web Deploy Handler

For IIS 7 onwards, Web Deploy offers an alternative deployment approach through the IIS Web Deploy Handler. The Web Deploy Handler is closely integrated with the IIS Web Management Service (WMSvc), which is designed to allow users to manage IIS websites from remote locations.

By default, the remote agent exposes an HTTP endpoint at this address:

```
https://[server]:8172/MSDeploy.axd
```

> **NOTE**
> You can replace [*server*] with the machine name of your web server, an IP address for your web server, or a hostname that resolves to your web server.

The big advantage of the Web Deploy Handler over the remote agent, and the temp agent, is that you can configure IIS to allow non-administrator users to deploy applications and content to specific IIS websites. The Web Deploy Handler also supports basic authentication, so you can provide alternative credentials as parameters in your Web Deploy commands. The major drawback is that the Web Deploy Handler is initially a lot more complicated to set up and configure.

In the case of non-administrator users, the Web Management Service (WMSvc) will only allow the user to connect to IIS using a site-level connection, rather than a server-level connection. To access a particular site, you can include a site-specific query string in the endpoint address:

```
https://[server]:8172/MSDeploy.axd?site=DemoSite
```

For example, suppose a build process is configured to automatically deploy a web application to a staging environment after every successful build. If you used the remote agent approach, you'd need to make the build process identity an administrator on your destination servers. In contrast, using the Web Deploy Handler approach you can give a non-administrator user—**FABRIKAM\stagingdeployer** in this case—permission to a specific IIS website only, and the build process can provide these credentials to deploy the web package.

```
msdeploy.exe
  -source:package='…\ContactManager.Mvc.zip'
  -dest:auto,
        computerName='https://STAGEWEB1:8172/MSDeploy.axd?site=DemoSite',
        userName='FABRIKAM\stagingdeployer',
        password='Pa$$w0rd',
        authtype='Basic',
  -verb:sync
  -setParamFile:"…\ContactManager.Mvc.SetParameters.xml"
  -allowUntrusted
```

NOTE

For more information on Web Deploy command-line operations and syntax, see Web Deploy Command Line Reference.
For more information on using the *.deploy.cmd* file, see How to: Install a Deployment Package Using the deploy.cmd File.

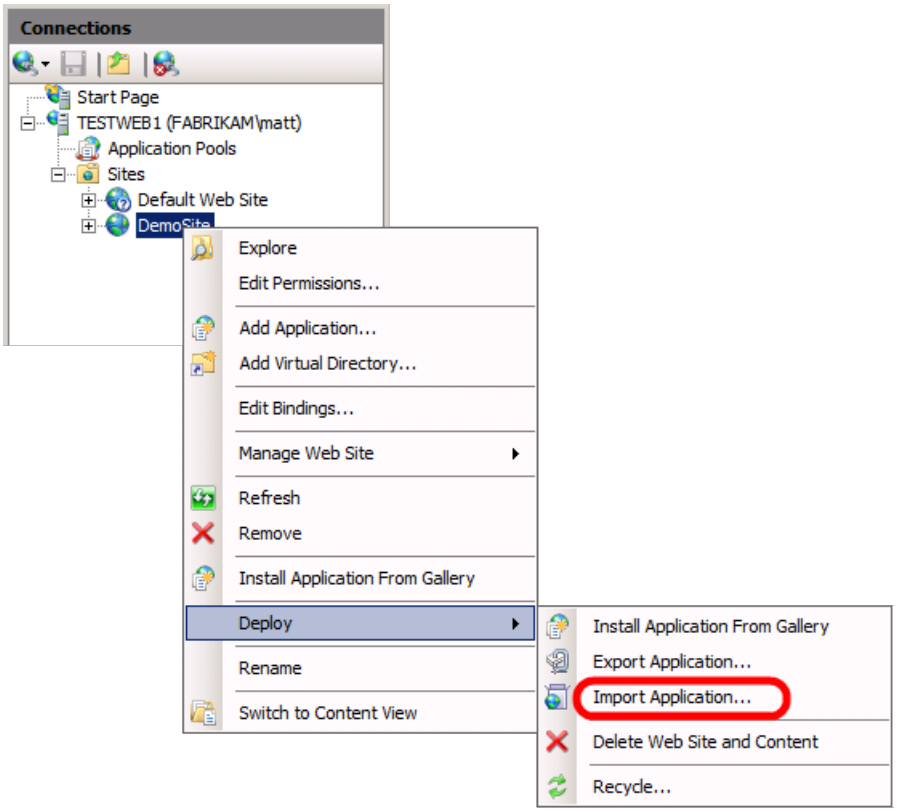
The Web Deploy Handler provides a useful approach to deployment in staging environments, hosted environments, and intranet-based production environments, where remote access to the server is available but administrator credentials are not.

For an end-to-end example of a scenario that uses the Web Deploy Handler approach, see Scenario: Configuring a Staging Environment for Web Deployment.

Using Offline Deployment

In some cases, it's not possible or practical to deploy applications and content to an IIS website from a remote location. For example, the source and destination computers may be in isolated networks or network segments, or firewall policy may not permit remote access.

In scenarios like these, you can still use the packaging and publishing capabilities of Web Deploy; you just can't use them from a remote location. Instead, an administrator on the destination server must copy the web package onto the server and import it through IIS Manager.



The offline deployment approach is typically useful in Internet-facing production environments, where servers in a perimeter network may have restricted connectivity with computers in the internal network.

For an end-to-end example of a scenario that uses the offline deployment approach, see Scenario: Configuring a Production Environment for Web Deployment.

Further Reading

For more information on Web Deploy command-line operations and syntax, see Web Deploy Command Line Reference. For more information on using the *.deploy.cmd* file, see How to: Install a Deployment Package Using the deploy.cmd File.

For more general guidance on the different ways in which you can deploy web packages from a remote computer, see Using Web Deploy Remotely. For more information on using Web Deploy On Demand, see Web Deploy On Demand.

PREVIOUS NEXT

Scenario: Configuring a Test Environment for Web Deployment

2/20/2020 • 2 minutes to read • Edit Online

by Jason Lee

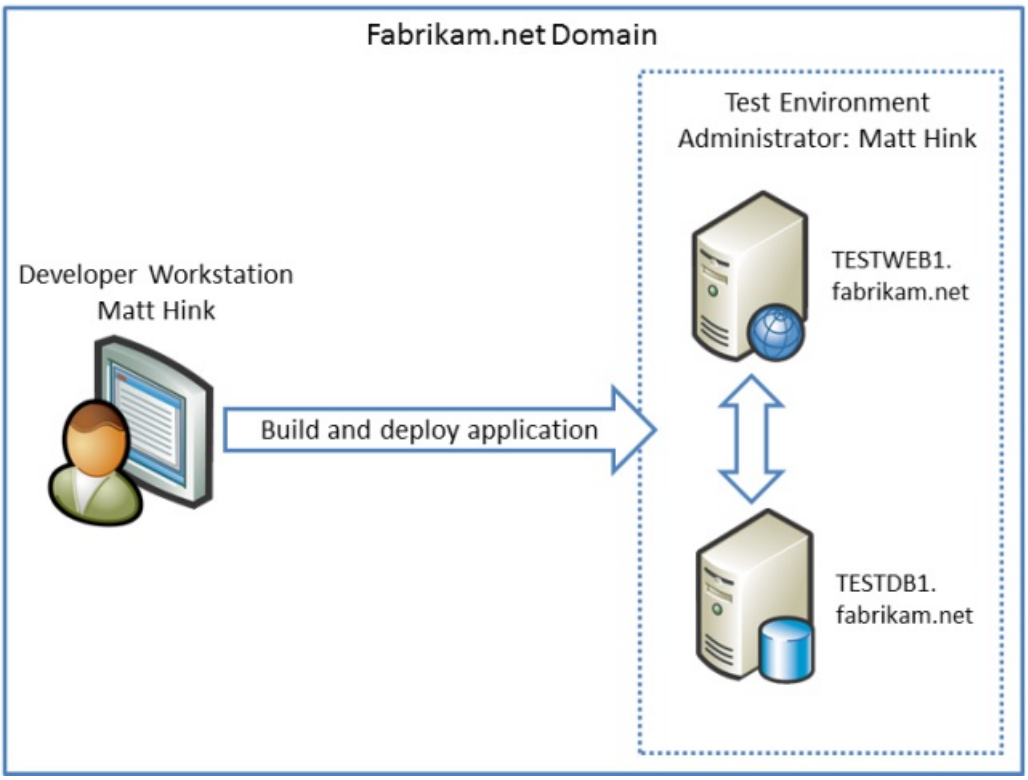
Download PDF

> This topic describes a typical web deployment scenario for developer or test environments and explains the tasks you need to complete in order to set up a similar environment.

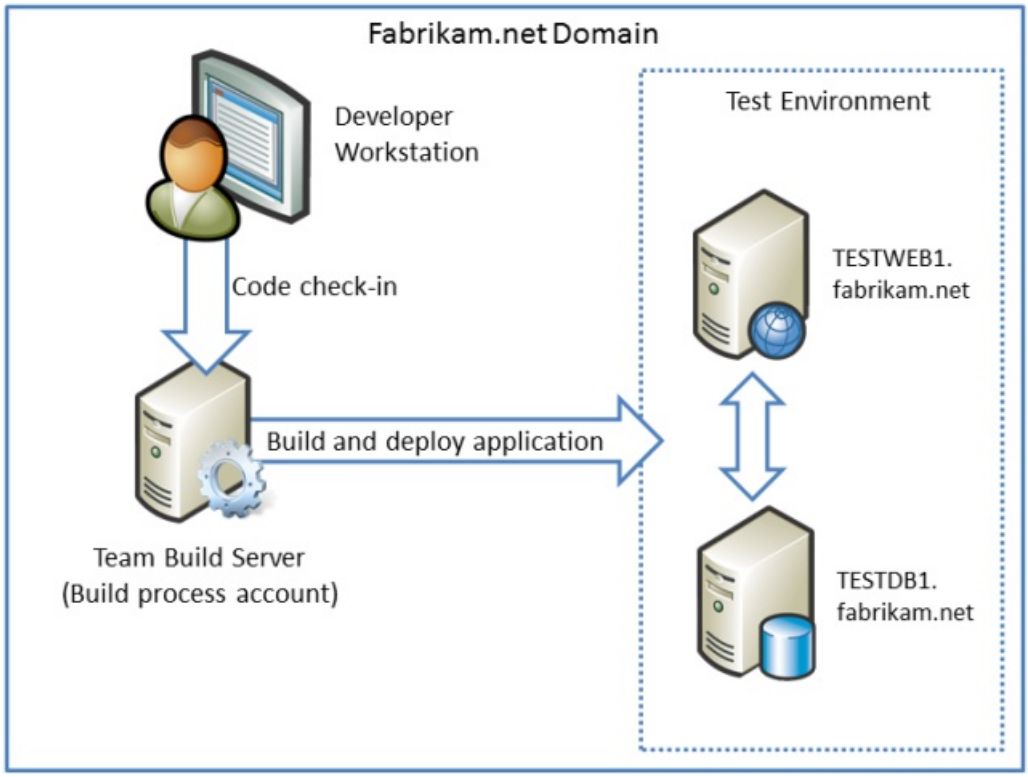
When developers work on web applications, they're often given access to a server environment that they can use to test changes to their applications in a realistic setting. This kind of development or test environment typically has these characteristics:

- The environment consists of a single web server and a single database server.
- The developers usually have administrator privileges on the servers, to let them configure the environment to the requirements of their applications.
- Changes to applications are deployed on a frequent basis, so the environment needs to support single-step or automated deployment.

For example, in our tutorial scenario, Matt Hink is a developer at Fabrikam, Inc. Matt is working on the Contact Manager solution and regularly needs to deploy changes to a test environment. Matt is an administrator on the test web server and the test database server. Initially, Matt needs to be able to deploy the solution to the test environment directly.



As work progresses and more developers join the team, the Contact Manager solution is configured for continuous integration (CI) in Team Foundation Server (TFS). Whenever a developer checks in content, Team Build should build the solution, run any unit tests, and automatically deploy the solution to the test environment.



Solution Overview

The test environment needs to support single-step or automated deployment from a remote computer, so you have a choice of two main approaches. You can:

- Configure the test web server to support deployment using the Web Deployment Agent Service (the "remote agent").
- Configure the test web server to support deployment using the Web Deploy handler.

> **NOTE**
>
> You could also use Web Deploy On Demand (the "temp agent"). This is similar to the remote agent approach in terms of requirements and constraints.

In this case, the developers have administrator privileges on the destination servers, and the test environment is not subject to strict security constraints, so the logical choice is to configure the test web server to support deployment using the remote agent. This is less complex and requires less initial configuration than the Web Deploy Handler approach. You'll also need to configure your database server to support remote access and deployment.

These topics provide all the information you need in order to complete these tasks:

- Configure a Web Server for Web Deploy Publishing (Remote Agent). This topic describes how to build a web server that supports Web Deploy publishing, using the remote agent approach, starting from a clean Windows Server 2008 R2 build.
- Configure a Database Server for Web Deploy Publishing. This topic describes how to configure a database server to support remote access and deployment, starting from a default installation of SQL Server 2008 R2.

Further Reading

For guidance on configuring a typical staging environment, see Scenario: Configuring a Staging Environment for Web Deployment. For guidance on configuring a typical production environment, see Scenario: Configuring a Production Environment for Web Deployment.

Scenario: Configuring a Staging Environment for Web Deployment

2/20/2020 • 3 minutes to read • Edit Online

by Jason Lee

Download PDF

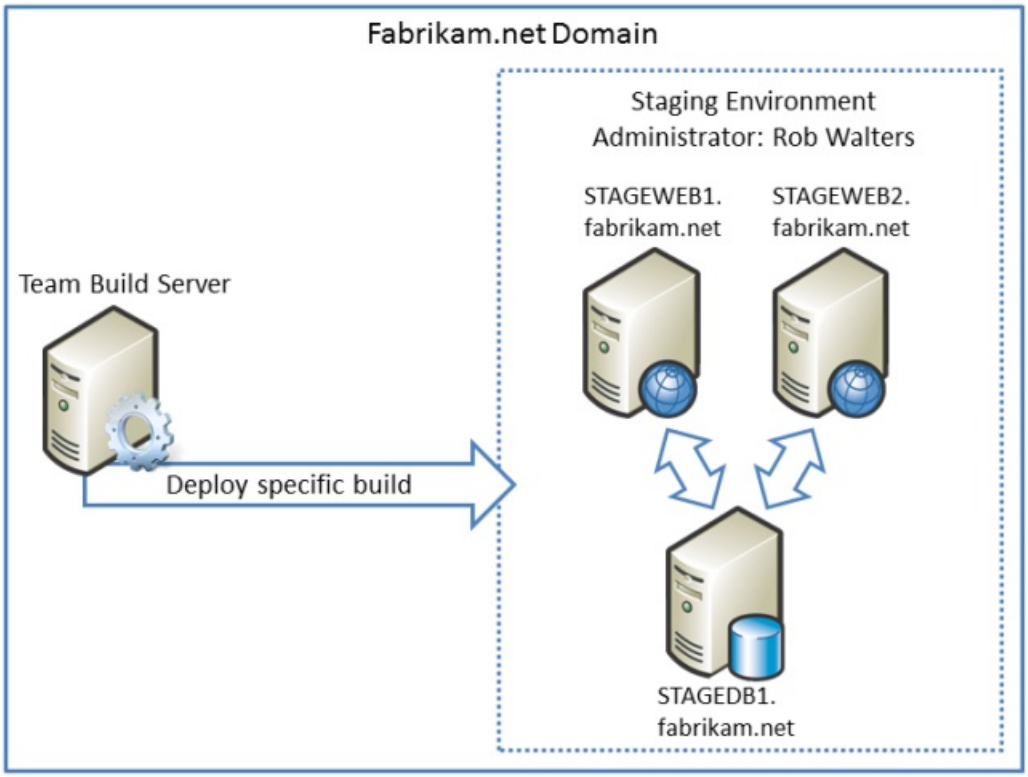
> This topic describes a typical web deployment scenario for a staging environment and explains the tasks you need to complete in order to set up a similar environment.

Lots of organizations use staging environments to preview updates to web applications or websites. This gives people within the organization a chance to explore and review new functionality or content before the site "goes live," or in other words is deployed to a production environment. The staging environment is designed to replicate the production environment as closely as possible, in order to provide a realistic preview. This kind of staging environment typically has these characteristics:

- The environment consists of multiple load-balanced web servers and one or more database servers, often with failover clustering and database mirroring.
- Applications may be deployed manually by a development team or automatically by a Team Build server.
- The users or process accounts that deploy applications are unlikely to have administrator privileges on the staging servers.
- Changes to applications are deployed on a frequent basis, so the environment needs to support single-step or automated deployment.

> **NOTE**
>
> Scaling out a database deployment across multiple servers is beyond the scope of this tutorial. For more information on this area, please consult SQL Server Books Online.

For example, in our tutorial scenario, Team Foundation Server (TFS) manages the Contact Manager solution. The TFS administrator, Rob Walters, has created a build definition that lets developers trigger a deployment to the staging environment as required.



Note that in most cases, you won't necessarily want to deploy the latest build to the staging environment. Instead, you're a lot more likely to want to deploy a specific build that has already undergone validation and verification in the test environment.

Solution Overview

In this scenario, you can deduce these facts from an analysis of the deployment requirements:

- The user or process account that performs the deployment won't have administrator privileges on the staging servers, so the staging web servers must support non-administrator deployment. As such, you'll need to configure the staging web servers to use the Web Deploy Handler rather than the remote agent.
- The staging environment includes multiple web servers, but it needs to support one-click or automated deployment, so you'll need to use the Web Farm Framework (WFF) to create a server farm. Using this approach, you can deploy an application to one web server (the primary server), and WFF will replicate the deployment on all the other web servers in the staging environment.
- The user or process account that performs the deployment must have permissions to create databases. As such, you'll need to add the account to the **dbcreator** server role on the database server, in addition to configuring the database server to support remote access and deployment.

These topics provide all the information you need in order to complete these tasks:

- Create a Server Farm with the Web Farm Framework. This topic describes how to create and configure a server farm using WFF, so that web platform products and components, configuration settings, and websites and applications are replicated across multiple load-balanced web servers.
- Configure a Web Server for Web Deploy Publishing (Web Deploy Handler). This topic describes how to build a web server that supports Web Deploy publishing, using the remote agent approach, starting from a clean Windows Server 2008 R2 build.
- Configure a Database Server for Web Deploy Publishing. This topic describes how to configure a database server to support remote access and deployment, starting from a default installation of SQL Server 2008 R2.

Further Reading

For guidance on configuring a typical developer test environment, see Scenario: Configuring a Test Environment

for Web Deployment. For guidance on configuring a typical production environment, see Scenario: Configuring a Production Environment for Web Deployment.

PREVIOUS NEXT

Scenario: Configuring a Production Environment for Web Deployment

2/20/2020 • 2 minutes to read • Edit Online

by Jason Lee

Download PDF

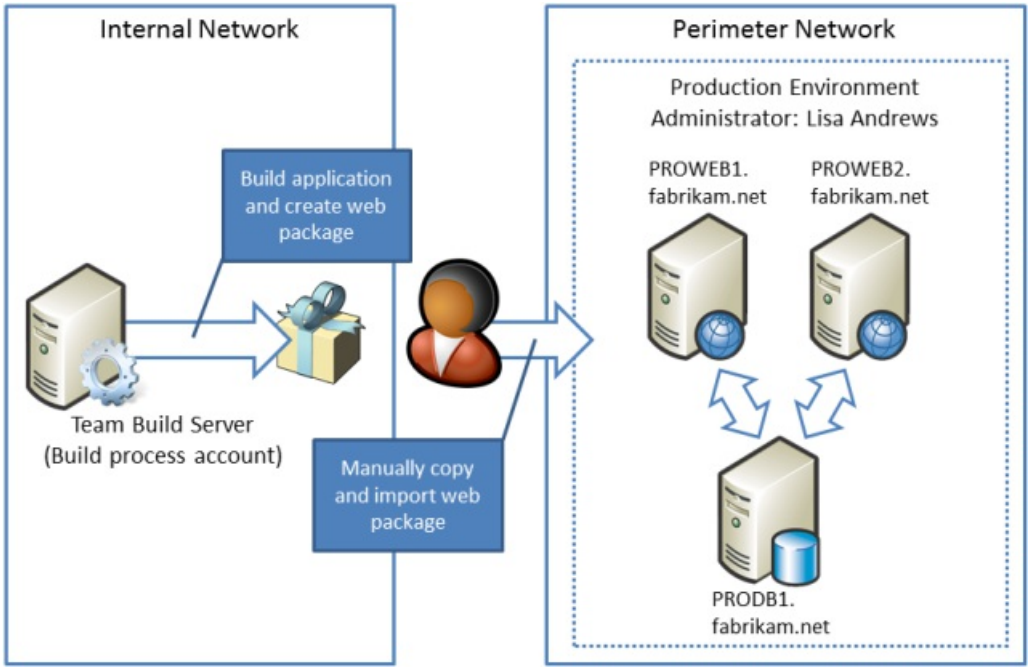
> This topic describes a typical web deployment scenario for a production environment and explains the tasks you need to complete in order to set up a similar environment.

The production environment is the final destination for a web application or a website. By this point, your application has been through testing, has been deployed to a staging environment, and is ready to "go live." The characteristics of a production environment can vary widely according to the nature and purpose of your web content, the size of your organization, your target audience, and lots of other factors. In an enterprise-scale scenario, the production environment may have these characteristics:

- The environment consists of multiple load-balanced web servers and one or more database servers, often with failover clustering and database mirroring.
- If the environment is Internet-facing, it's likely to be segregated from your internal network. It may be on a different subnet in a perimeter network, it may be on a different domain, and it may be on an entirely different network infrastructure.
- Developers and build server process accounts are highly unlikely to have administrator privileges on the production servers.
- Changes to applications are deployed on a less frequent basis than test or staging deployments.

> **NOTE**
>
> Scaling out a database deployment across multiple servers is beyond the scope of this tutorial. For more information on this area, please consult SQL Server Books Online.

For example, in our tutorial scenario, a Team Build server includes build definitions that let users build the Contact Manager solution and deploy it to a staging environment in a single step. When the application is ready to be deployed to production, due to the constraints imposed by security requirements and the network infrastructure, the production environment administrator must manually copy the web package onto a production web server and import it through Internet Information Services (IIS) Manager.



Solution Overview

In this scenario, you can deduce these facts from an analysis of the deployment requirements:

- Due to security restrictions and the network configuration, you can't configure the production environment to support one-click or automated deployment. Offline deployment is the only viable approach in this scenario.
- The production environment includes multiple web servers, so you can use the Web Farm Framework (WFF) to create a server farm. Using this approach, the administrator only needs to import the application onto one web server (the primary server), and WFF will replicate the deployment on all the other web servers in the production environment.

These topics provide all the information you need in order to complete these tasks:

- Create a Server Farm with the Web Farm Framework. This topic describes how to create and configure a server farm using WFF, so that web platform products and components, configuration settings, and websites and applications are replicated across multiple load-balanced web servers.
- Configure a Web Server for Web Deploy Publishing (Offline Deployment). This topic describes how to build a web server that lets administrators import and deploy web packages manually, starting from a clean Windows Server 2008 R2 build.
- Configure a Database Server for Web Deploy Publishing. This topic describes how to configure a database server to support remote access and deployment, starting from a default installation of SQL Server 2008 R2.

Further Reading

For guidance on configuring a typical developer test environment, see Scenario: Configuring a Test Environment for Web Deployment. For guidance on configuring a typical staging environment, see Scenario: Configuring a Staging Environment for Web Deployment.

PREVIOUS NEXT

Configuring a Web Server for Web Deploy Publishing (Remote Agent)

9/16/2020 • 12 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic describes how to configure an Internet Information Services (IIS) web server to support web publishing and deployment using the IIS Web Deployment Tool (Web Deploy) Remote Agent Service.

When you work with Web Deploy 2.0 or later, there are three main approaches you can use to get your applications or sites onto a web server. You can:

- Use the *Web Deploy Remote Agent Service*. This approach requires less configuration of the web server, but you need to provide the credentials of a local server administrator in order to deploy anything to the server.
- Use the *Web Deploy Handler*. This approach is a lot more complex and requires more initial effort to set up the web server. However, when you use this approach, you can configure IIS to allow non-administrator users to perform the deployment. The Web Deploy Handler is only available in IIS version 7 or later.
- Use *offline deployment*. This approach requires the least configuration of the web server, but a server administrator must manually copy the web package onto the server and import it through IIS Manager.

For more information on the key features, advantages, and disadvantages of these approaches, see Choosing the Right Approach to Web Deployment.

Is the Web Deploy Remote Agent the Right Approach for You?

Yes, if the user who will deploy the content can supply the credentials of an administrator on the destination server. This approach is often desirable in these types of scenarios:

- Development or test environments, where the developer has full control over the destination web server and database server.
- Smaller organizations in which a single user or a small group of users has control over the entire application lifecycle.

In lots of larger organizations, and particularly for staging or production environments, it's often not realistic to give users administrator rights on web servers. In the case of hosted web servers, this is especially unlikely to be the case. In addition, if you're planning to automate deployment from a build server, you may not want to use administrator credentials for the deployment process. In these scenarios, configuring your web servers to support deployment using the Web Deploy Handler may provide a more satisfactory choice.

Task Overview

This topic describes how to configure an Internet Information Services (IIS) 7.5 web server to accept and deploy web packages from a remote computer using the Web Deploy Remote Agent approach. You'll need to:

- Install IIS 7.5 and the IIS 7 recommended configuration.
- Install Web Deploy 2.1 or later.

- Create an IIS website to host the deployed content.
- Ensure that the Web Deployment Agent Service is running.

To host the sample solution specifically, you'll also need to:

- Install the .NET Framework 4.0.
- Install ASP.NET MVC 3.

This topic will show you how to perform each of these procedures. The tasks and walkthroughs in this topic assume that you're starting with a clean server build running Windows Server 2008 R2. Before you continue, ensure that:

- Windows Server 2008 R2 Service Pack 1 and all available updates are installed.
- The server is domain-joined.
- The server has a static IP address.

> **NOTE**
> For more information on joining computers to a domain, see Joining Computers to the Domain and Logging On. For more information on configuring static IP addresses, see Configure a Static IP Address. The Remote Agent service is supported by IIS 6 onwards and does not require you to be joined to a domain. However, the steps in this tutorial were developed and tested on IIS 7.5 and procedures for other versions may vary.

Install Products and Components

This section will guide you through installing the required products and components on the web server. Before you begin, a good practice is to run Windows Update to ensure that your server is fully up to date.

In this case, you need to install these things:

- **IIS 7 Recommended Configuration**. This enables the **Web Server (IIS)** role on your web server and installs the set of IIS modules and components that you need in order to host an ASP.NET application.
- **.NET Framework 4.0**. This is required to run applications that were built on this version of the .NET Framework.
- **Web Deployment Tool 2.1 or later**. This installs Web Deploy (and its underlying executable, MSDeploy.exe) on your server. As part of this process, it installs and starts the Web Deployment Agent Service. This service lets you deploy web packages from a remote computer.
- **ASP.NET MVC 3**. This installs the assemblies you need to run MVC 3 applications.

> **NOTE**
> This walkthrough describes the use of the Web Platform Installer to install and configure the required components. Although you don't have to use the Web Platform Installer, it simplifies the installation process by automatically detecting dependencies and ensuring that you always get the latest product versions. For more information, see Microsoft Web Platform Installer 3.0.

To install the required products and components

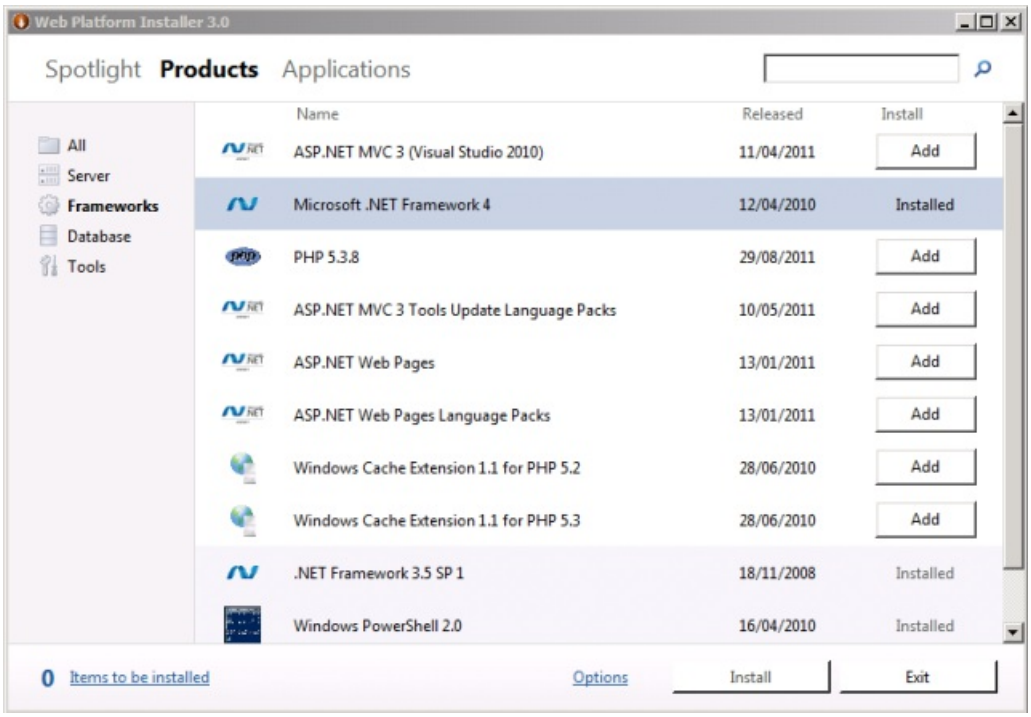
1. Download and install the Web Platform Installer.

2. When installation is complete, the Web Platform Installer will launch automatically.

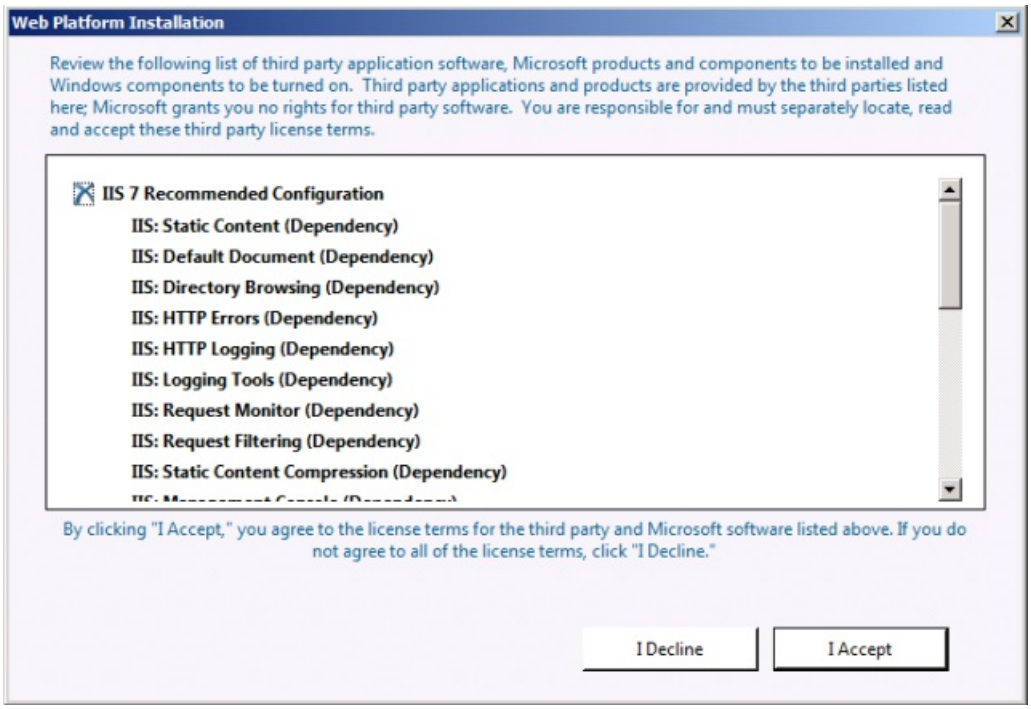
> **NOTE**
>
> You can now launch the Web Platform Installer at any time from the **Start** menu. To do this, on the **Start** menu, click **All Programs**, and then click **Microsoft Web Platform Installer**.

3. At the top of the **Web Platform Installer 3.0** window, click **Products**.
4. On the left side of the window, in the navigation pane, click **Frameworks**.
5. In the **Microsoft .NET Framework 4** row, if the .NET Framework is not already installed, click **Add**.

> **NOTE**
>
> You may have already installed the .NET Framework 4.0 through Windows Update. If a product or component is already installed, the Web Platform Installer will indicate this by replacing the **Add** button with the text **Installed**.



6. In the **ASP.NET MVC 3 (Visual Studio 2010)** row, click **Add**.
7. In the navigation pane, click **Server**.
8. In the **IIS 7 Recommended Configuration** row, click **Add**.
9. In the **Web Deployment Tool 2.1** row, click **Add**.
10. Click **Install**. The Web Platform Installer will show you a list of products—together with any associated dependencies—to be installed and will prompt you to accept the license terms.



11. Review the license terms, and if you consent to the terms, click **I Accept**.
12. When the installation is complete, click **Finish**, and then close the **Web Platform Installer 3.0** window.

If you installed the .NET Framework 4.0 before you installed IIS, you'll need to run the ASP.NET IIS Registration Tool (aspnet_regiis.exe) to register the latest version of ASP.NET with IIS. If you don't do this, you'll find that IIS will serve static content (like HTML files) without any problems, but it will return **HTTP Error 404.0 – Not Found** when you attempt to browse to ASP.NET content. You can use this procedure to ensure that ASP.NET 4.0 is registered.

To register ASP.NET 4.0 with IIS

1. Click **Start**, and then type **Command Prompt**.
2. In the search results, right-click **Command Prompt**, and then click **Run as administrator**.
3. In the Command Prompt window, navigate to the **%WINDIR%\Microsoft.NET\Framework\v4.0.30319** directory.
4. Type this command, and then press Enter:

```
aspnet_regiis -iru
```

5. If you plan to host 64-bit web applications at any point, you should also register the 64-bit version of ASP.NET with IIS. To do this, in the Command Prompt window, navigate to the **%WINDIR%\Microsoft.NET\Framework64\v4.0.30319** directory.
6. Type this command, and then press Enter:

```
aspnet_regiis -iru
```

As a good practice, use Windows Update again at this point to download and install any available updates for the new products and components you've installed.

Configure the IIS Website

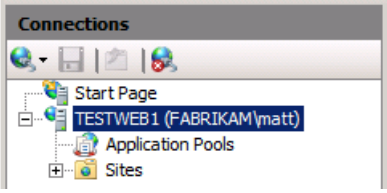
Before you can deploy web content to your server, you need to create and configure an IIS website to host the content. Web Deploy can only deploy web packages to an existing IIS website; it can't create the website for you. At a high level, you'll need to complete these tasks:

- Create a folder on the file system to host your content.
- Create an IIS website to serve the content, and associate it with the local folder.
- Grant read permissions to the application pool identity on the local folder.

Although there's nothing stopping you from deploying content to the default website in IIS, this approach is not recommended for anything other than test or demonstration scenarios. To simulate a production environment, you should create a new IIS website with settings that are specific to the requirements of your application.

To create and configure an IIS website

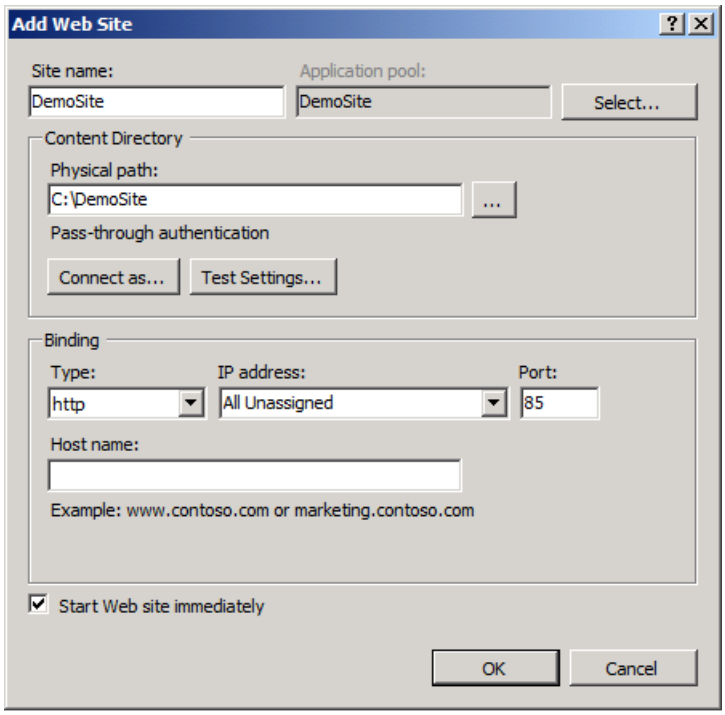
1. On the local file system, create a folder to store your content (for example, **C:\DemoSite**).
2. On the **Start** menu, point to **Administrative Tools**, and then click **Internet Information Services (IIS) Manager**.
3. In IIS Manager, in the **Connections** pane, expand the server node (for example, **TESTWEB1**).



4. Right-click the **Sites** node, and then click **Add Web Site**.
5. In the **Site name** box, type a name for the IIS website (for example, **DemoSite**).
6. In the **Physical path** box, type (or browse to) the path to your local folder (for example, **C:\DemoSite**).
7. In the **Port** box, type the port number on which you want to host the website (for example, **85**).

> **NOTE**
> The standard port numbers are 80 for HTTP and 443 for HTTPS. However, if you host this website on port 80, you'll need to stop the default website before you can access your site.

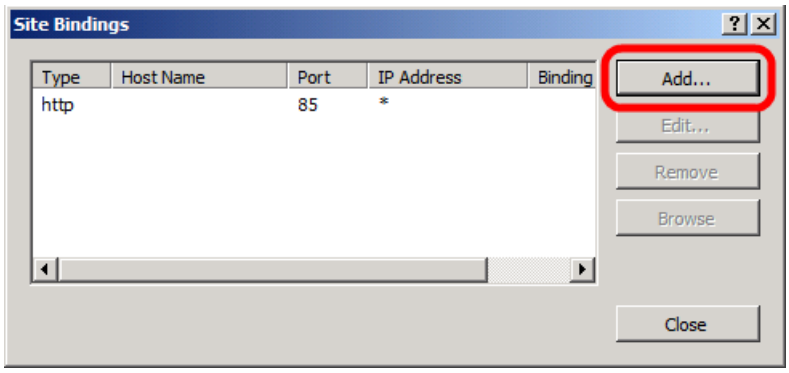
8. Leave the **Host name** box blank, unless you want to configure a Domain Name System (DNS) record for the website, and then click **OK**.



> **NOTE**
>
> In a production environment, you'll likely want to host your website on port 80 and configure a host header, together with matching DNS records. For more information on configuring host headers in IIS 7, see Configure a Host Header for a Web Site (IIS 7). For more information on the DNS Server role in Windows Server 2008 R2, see DNS Server Overview and DNS Server.

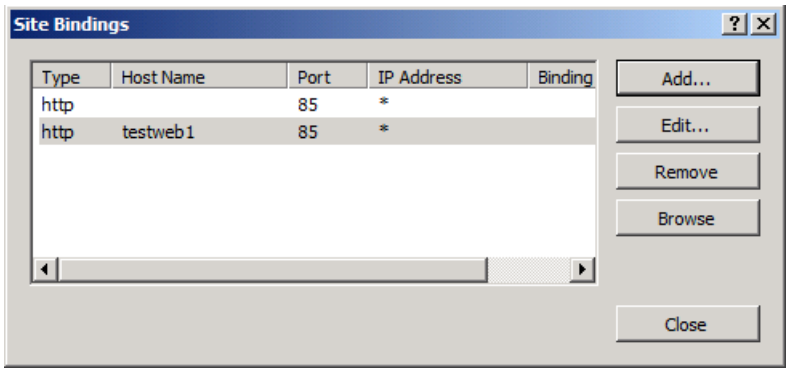
9. In the **Actions** pane, under **Edit Site**, click **Bindings**.

10. In the **Site Bindings** dialog box, click **Add**.



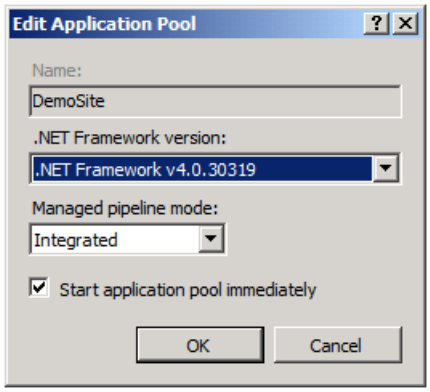
11. In the **Add Site Binding** dialog box, set the **IP address** and **Port** to match your existing site configuration.

12. In the **Host name** box, type the name of your web server (for example, **TESTWEB1**), and then click **OK**.



> **NOTE**
> The first site binding allows you to access the site locally using the IP address and port or `http://localhost:85`. The second site binding allows you to access the site from other computers on the domain using the machine name (for example, http://testweb1:85).

13. In the **Site Bindings** dialog box, click **Close**.
14. In the **Connections** pane, click **Application Pools**.
15. In the **Application Pools** pane, right-click the name of your application pool, and then click **Basic Settings**. By default, the name of your application pool will match the name of your website (for example, **DemoSite**).
16. In the **.NET Framework version** list, select **.NET Framework v4.0.30319**, and then click **OK**.



> **NOTE**
> The sample solution requires .NET Framework 4.0. This is not a requirement for Web Deploy in general.

In order for your website to serve content, the application pool identity must have read permissions on the local folder that stores the content. In IIS 7.5, application pools run with a unique application pool identity by default (in contrast to previous versions of IIS, where application pools would typically run using the Network Service account). The application pool identity is not a real user account and does not show up on any lists of users or groups—instead, it's created dynamically when the application pool is started. Each application pool identity is added to the local **IIS_IUSRS** security group as a hidden item.

To grant permissions to an application pool identity on a file or folder, you have two options:

- Assign permissions to the application pool identity directly, using the format **IIS AppPool[application pool name]** (for example, **IIS AppPool\DemoSite**).
- Assign permissions to the **IIS_IUSRS** group.

The most common approach is to assign permissions to the local **IIS_IUSRS** group because this approach lets you change application pools without reconfiguring file system permissions. The next procedure uses this group-based approach.

> **NOTE**
> For more information on application pool identities in IIS 7.5, see Application Pool Identities.

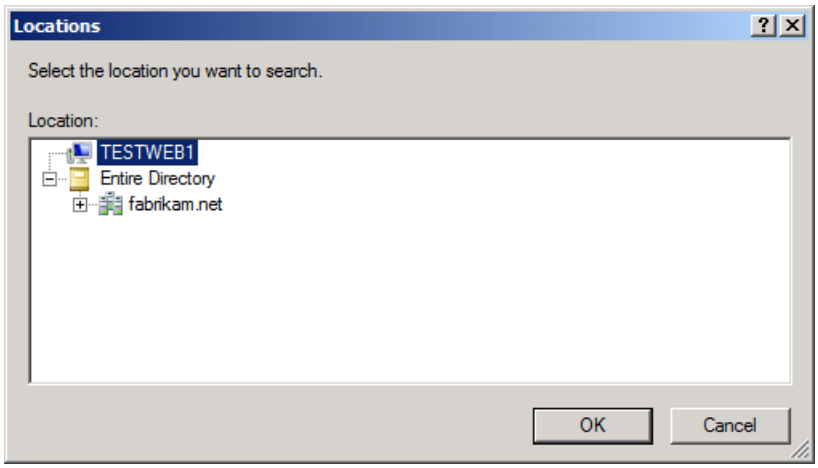
To configure folder permissions for an IIS website

1. In Windows Explorer, browse to the location of your local folder.

2. Right-click the folder, and then click **Properties**.

3. On the **Security** tab, click **Edit**, and then click **Add**.

4. Click **Locations**. In the **Locations** dialog box, select the local server, and then click **OK**.



5. In the **Select Users or Groups** dialog box, type **IIS_IUSRS**, click **Check Names**, and then click **OK**.

6. In the **Permissions for (folder name) dialog box**, notice that the new group has been assigned the **Read & execute**, **List folder contents**, and **Read** permissions by default. Leave this unchanged and click **OK**.

7. Click **OK** to close the **(folder name) Properties** dialog box.

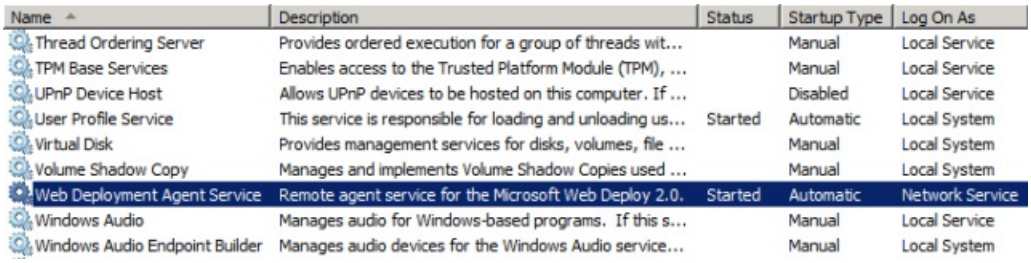
As a final task before you attempt to deploy any web packages to your server, you should ensure that the Web Deployment Agent Service is running. When you deploy a package from a remote computer, the Web Deployment Agent Service is responsible for extracting and installing the contents of the package. The service is started by default when you install the Web Deployment Tool and runs under the Network Service identity.

You can check whether a service is running in multiple different ways, using various command-line utilities or Windows PowerShell cmdlets. This procedure describes a straightforward UI-based approach.

To check that the Web Deployment Agent Service is running

1. On the **Start** menu, point to **Administrative Tools**, and then click **Services**.

2. Locate the **Web Deployment Agent Service** row, and verify that the **Status** is set to **Started**.



Name	Description	Status	Startup Type	Log On As
Thread Ordering Server	Provides ordered execution for a group of threads wit...		Manual	Local Service
TPM Base Services	Enables access to the Trusted Platform Module (TPM), ...		Manual	Local Service
UPnP Device Host	Allows UPnP devices to be hosted on this computer. If ...		Disabled	Local Service
User Profile Service	This service is responsible for loading and unloading us...	Started	Automatic	Local System
Virtual Disk	Provides management services for disks, volumes, file ...		Manual	Local System
Volume Shadow Copy	Manages and implements Volume Shadow Copies used ...		Manual	Local System
Web Deployment Agent Service	Remote agent service for the Microsoft Web Deploy 2.0.	Started	Automatic	Network Service
Windows Audio	Manages audio for Windows-based programs. If this s...		Manual	Local Service
Windows Audio Endpoint Builder	Manages audio devices for the Windows Audio service...		Manual	Local System

3. If the service is not already started, click **Start**.

Configure Firewall Exceptions

By default, the Remote Agent Service listens on TCP port 80, at this URL:

http://servername.com/MSDEPLOYAGENTSERVICE

In most cases, you won't need to configure any additional firewall rules for the Remote Agent Service because web servers typically listen for HTTP requests on port 80. If you customized your installation to listen on a

nonstandard port, you'll need to configure firewall exceptions as required.

Conclusion

At this point, your web server is ready to accept and install web packages from a remote computer. Before you attempt to deploy a web application to the server, you may want to check these key points:

- Have you registered ASP.NET 4.0 with IIS?
- Does the application pool identity have read access to the source folder for your website?
- Is the Web Deployment Agent Service running?

Further Reading

For guidance on how to configure custom Microsoft Build Engine (MSBuild) project files to deploy web packages to the Remote Agent Service, see Configuring Deployment Properties for a Target Environment.

PREVIOUS NEXT

Configuring a Web Server for Web Deploy Publishing (Web Deploy Handler)

9/16/2020 • 14 minutes to read • Edit Online

PDF Download

This topic describes how to configure an Internet Information Services (IIS) web server to support web publishing and deployment using the IIS Web Deploy Handler.

When you work with Web Deploy 2.0 or later, there are three main approaches you can use to get your applications or sites onto a web server. You can:

- Use the *Web Deploy Remote Agent Service*. This approach requires less configuration of the web server, but you need to provide the credentials of a local server administrator in order to deploy anything to the server.
- Use the *Web Deploy Handler*. This approach is a lot more complex and requires more initial effort to set up the web server. However, when you use this approach, you can configure IIS to allow non-administrator users to perform the deployment. The Web Deploy Handler is only available in IIS version 7 or later.
- Use *offline deployment*. This approach requires the least configuration of the web server, but a server administrator must manually copy the web package onto the server and import it through IIS Manager.

For more information on the key features, advantages, and disadvantages of these approaches, see Choosing the Right Approach to Web Deployment.

Yes, if you want to allow non-administrator users to deploy content to specific IIS websites. This approach is often desirable in these types of scenarios:

- Staging or production environments, where the person or service account that triggers the remote deployment is unlikely to have access to the credentials of a server administrator.
- Hosted environments, where you want to give remote users the ability to update their websites without giving them full control of your web servers (or access to anyone else's websites).

In development or test scenarios, or in smaller organizations, deploying content using server administrator credentials is often less contentious. In these scenarios, configuring your web servers to support deployment using the Web Deploy Remote Agent Service offers a more straightforward approach.

Task Overview

To configure the web server to accept and deploy web packages from a remote computer using the Web Deploy Handler approach, you'll need to:

- Create, or choose, a domain user account (the "non-administrator user") whose credentials you'll use to perform deployments.
- Install IIS 7.5, including the Web Management Service and the Basic Authentication module.
- Install Web Deploy 2.1 or later.
- Configure the Web Management Service to allow remote connections, and start the service.
- Create an IIS website to host the deployed content.
- Grant your non-administrator user permissions on your website in IIS Manager.

- Ensure that the Web Management Service delegation rules permit the service to add and change website content using your non-administrator user account.
- Configure any firewalls to allow incoming connections on port 8172.

To host the ContactManager sample solution specifically, you'll also need to:

- Install the .NET Framework 4.0.
- Install ASP.NET MVC 3.

This topic will show you how to perform each of these procedures. The tasks and walkthroughs in this topic assume that you're starting with a clean server build running Windows Server 2016. Before you continue, ensure that:

- Windows Server 2016
- The server is domain-joined.
- The server has a static IP address.

> **NOTE**
> For more information on joining computers to a domain, see Joining Computers to the Domain and Logging On. For more information on configuring static IP addresses, see Configure a Static IP Address.

Install Products and Components

This section will guide you through installing the required products and components on the web server. Before you begin, a good practice is to run Windows Update to ensure that your server is fully up to date.

In this case, you need to install these things:

- **IIS 7 Recommended Configuration**. This enables the **Web Server (IIS)** role on your web server and installs the set of IIS modules and components that you need in order to host an ASP.NET application.
- **IIS: Management Service**. This installs the Web Management Service (WMSvc) in IIS. This service enables remote management of IIS websites and exposes the Web Deploy Handler endpoint to clients.
- **IIS: Basic Authentication**. This installs the IIS Basic Authentication module. This lets the Web Management Service (WMSvc) authenticate the credentials you provide.
- **Web Deployment Tool 2.1 or later**. This installs Web Deploy (and its underlying executable, MSDeploy.exe) on your server. As part of this process, it installs the Web Deploy Handler and integrates it with the Web Management Service.
- **.NET Framework 4.0**. This is required to run applications that were built on this version of the .NET Framework.
- **ASP.NET MVC 3**. This installs the assemblies you need to run MVC 3 applications.

> **NOTE**
> This walkthrough describes the use of the Web Platform Installer to install and configure various components. Although you don't have to use the Web Platform Installer, it simplifies the installation process by automatically detecting dependencies and ensuring that you always get the latest product versions. For more information, see Microsoft Web Platform Installer.

To install the required products and components

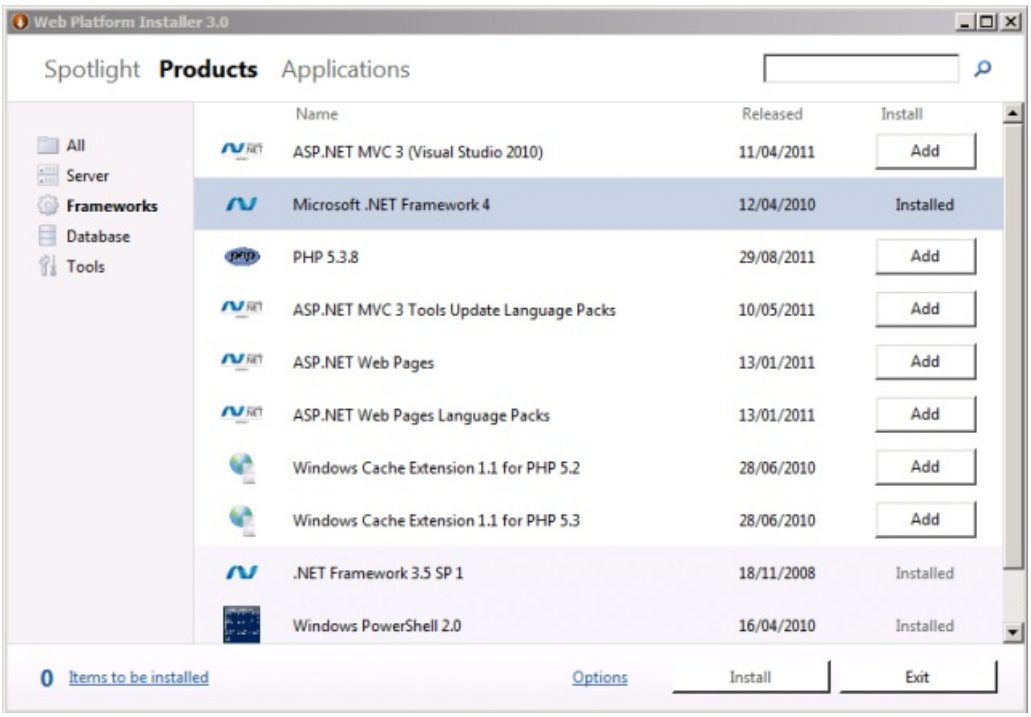
1. Download and install the Web Platform Installer.

2. When installation is complete, the Web Platform Installer will launch automatically.

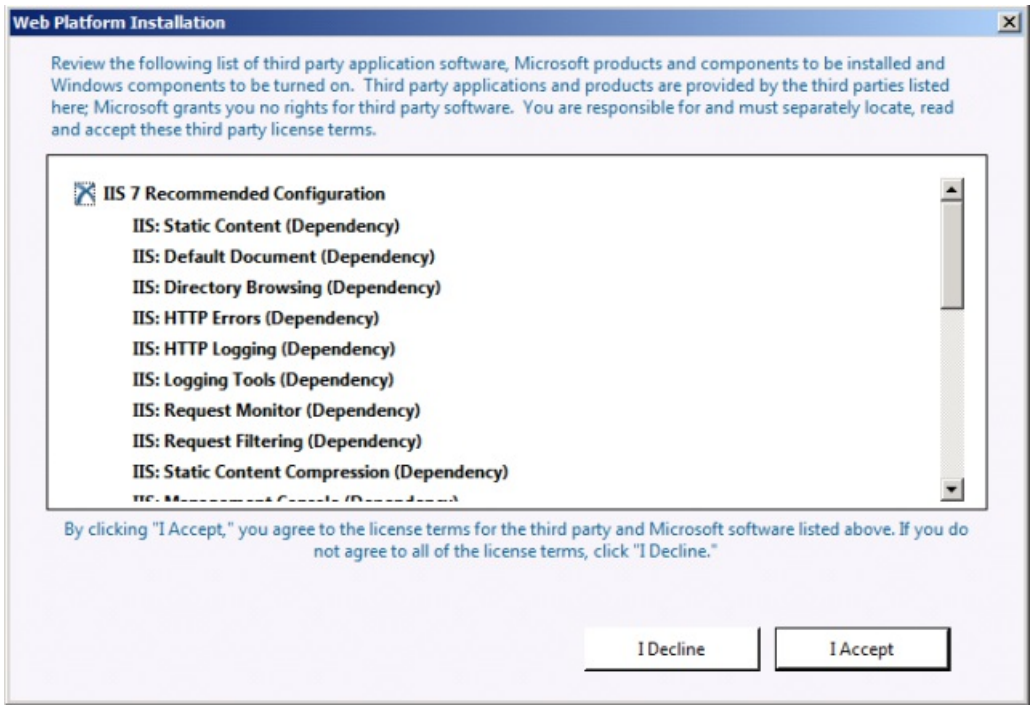
> **NOTE**
>
> You can now launch the Web Platform Installer at any time from the **Start** menu. To do this, on the **Start** menu, click **All Programs**, and then click **Microsoft Web Platform Installer**.

3. At the top of the **Web Platform Installer** window, click **Products**.
4. On the left side of the window, in the navigation pane, click **Frameworks**.
5. In the **Microsoft .NET Framework 4** row, if the .NET Framework is not already installed, click **Add**.

> **NOTE**
>
> You may have already installed the .NET Framework 4.0 through Windows Update. If a product or component is already installed, the Web Platform Installer will indicate this by replacing the **Add** button with the text **Installed**.



6. In the **ASP.NET MVC 3 (Visual Studio 2010)** row, click **Add**.
7. In the navigation pane, click **Server**.
8. In the **IIS 7 Recommended Configuration** row, click **Add**.
9. In the **Web Deployment Tool 2.1** row, click **Add**.
10. In the **IIS: Basic Authentication** row, click **Add**.
11. In the **IIS: Management Service** row, click **Add**.
12. Click **Install**. The Web Platform Installer will show you a list of products—together with any associated dependencies—to be installed and will prompt you to accept the license terms.



13. Review the license terms, and if you consent to the terms, click **I Accept**.
14. When the installation is complete, click **Finish**, and then close the **Web Platform Installer** window.

If you installed the .NET Framework 4.0 before you installed IIS, you'll need to run the ASP.NET IIS Registration Tool (aspnet_regiis.exe) to register the latest version of ASP.NET with IIS. If you don't do this, you'll find that IIS will serve static content (like HTML files) without any problems, but it will return **HTTP Error 404.0 – Not Found** when you attempt to browse to ASP.NET content. You can use the next procedure to ensure that ASP.NET 4.0 is registered.

To register ASP.NET 4.0 with IIS

1. Click **Start**, and then type **Command Prompt**.
2. In the search results, right-click **Command Prompt**, and then click **Run as administrator**.
3. In the Command Prompt window, navigate to the **%WINDIR%\Microsoft.NET\Framework\v4.0.30319** directory.
4. Type this command, and then press Enter:

```
aspnet_regiis -iru
```

5. If you plan to host 64-bit web applications at any point, you should also register the 64-bit version of ASP.NET with IIS. To do this, in the Command Prompt window, navigate to the **%WINDIR%\Microsoft.NET\Framework64\v4.0.30319** directory.
6. Type this command, and then press Enter:

```
aspnet_regiis -iru
```

As a good practice, use Windows Update again at this point to download and install any available updates for the new products and components you've installed.

Configure the Web Management Service

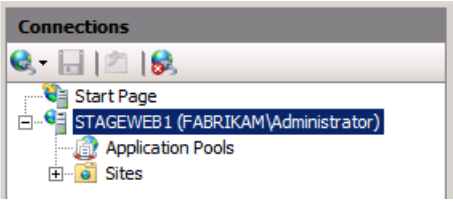
Now that you've installed everything you need, the next step is to configure the Web Management Service in IIS. At a high level, you'll need to complete these tasks:

- Enable basic authentication at the server level.
- Configure the Web Management Service to accept remote connections.
- Start the Web Management Service.
- Check that the required Web Management Service delegation rules are in place.

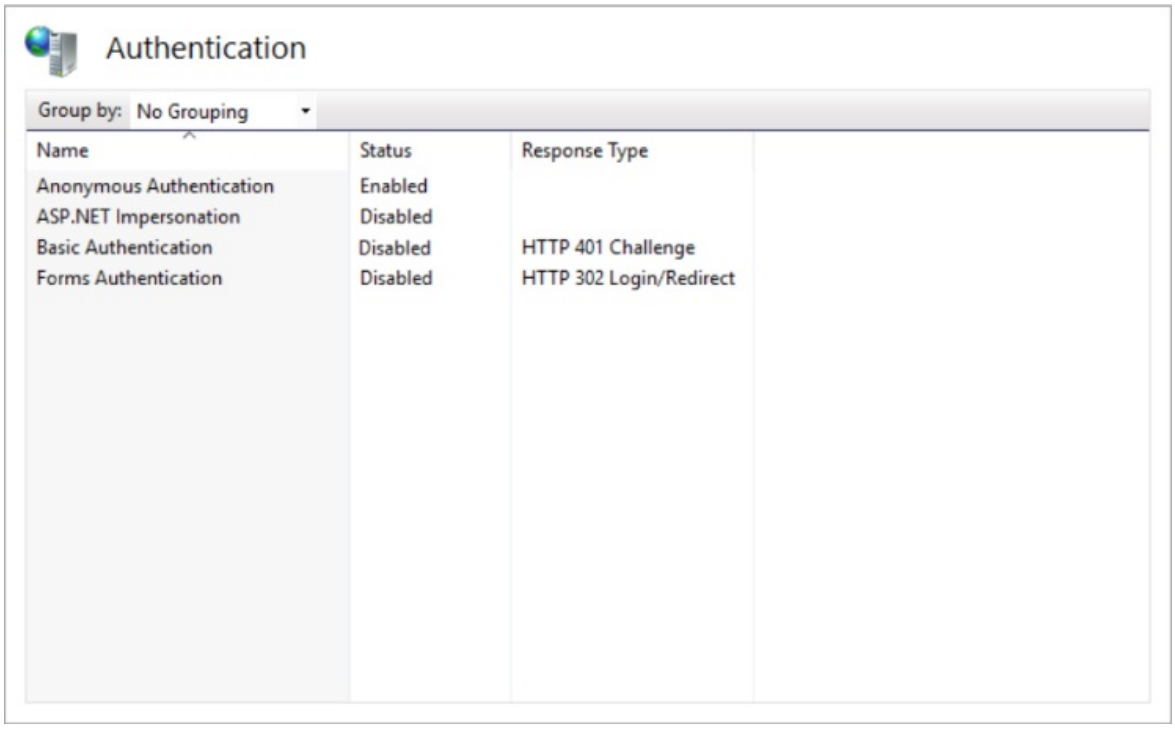
To configure the Web Management Service

1. On the **Start** menu, point to **Administrative Tools**, and then click **Internet Information Services (IIS) Manager**.

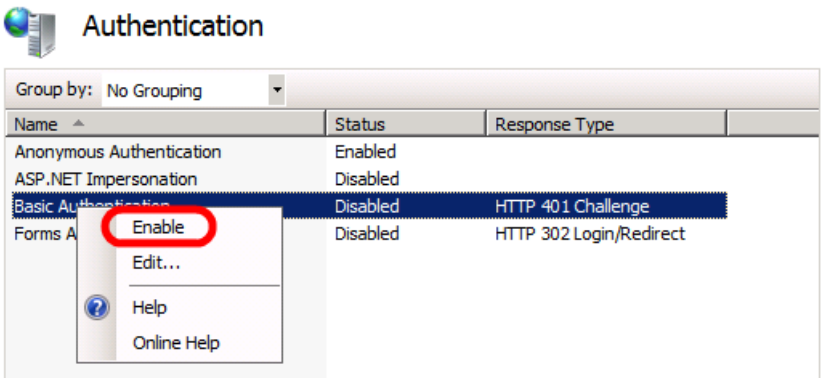
2. In IIS Manager, in the **Connections** pane, click the server node (for example, **STAGEWEB1**).



3. In the center pane, under **IIS**, double-click **Authentication**.

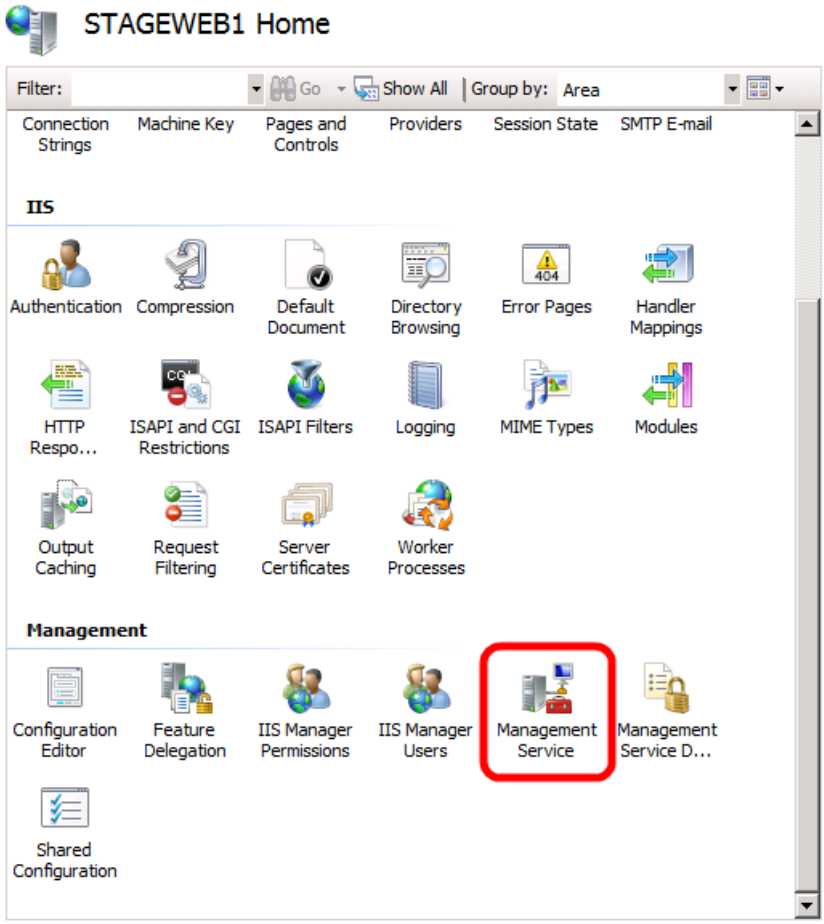


4. Right-click **Basic Authentication**, and then click **Enable**.



5. In the **Connections** pane, click the server node again to return to the top-level settings.

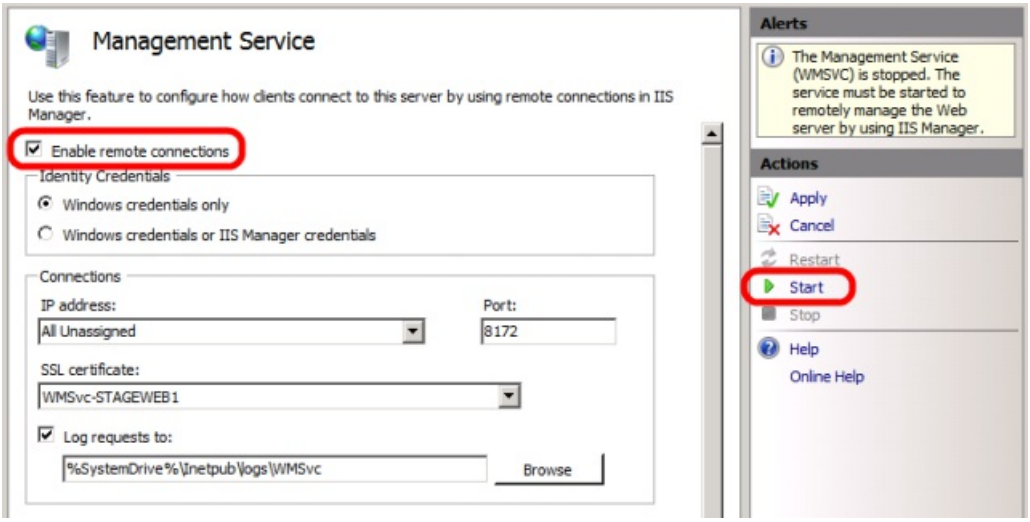
6. In the center pane, under **Management**, double-click **Management Service**.



7. In the center pane, select **Enable remote connections**.

> **NOTE**
>
> If the Web Management Service is already running, you'll need to stop it first.

8. In the **Actions** pane, click **Start** to start the Web Management Service.

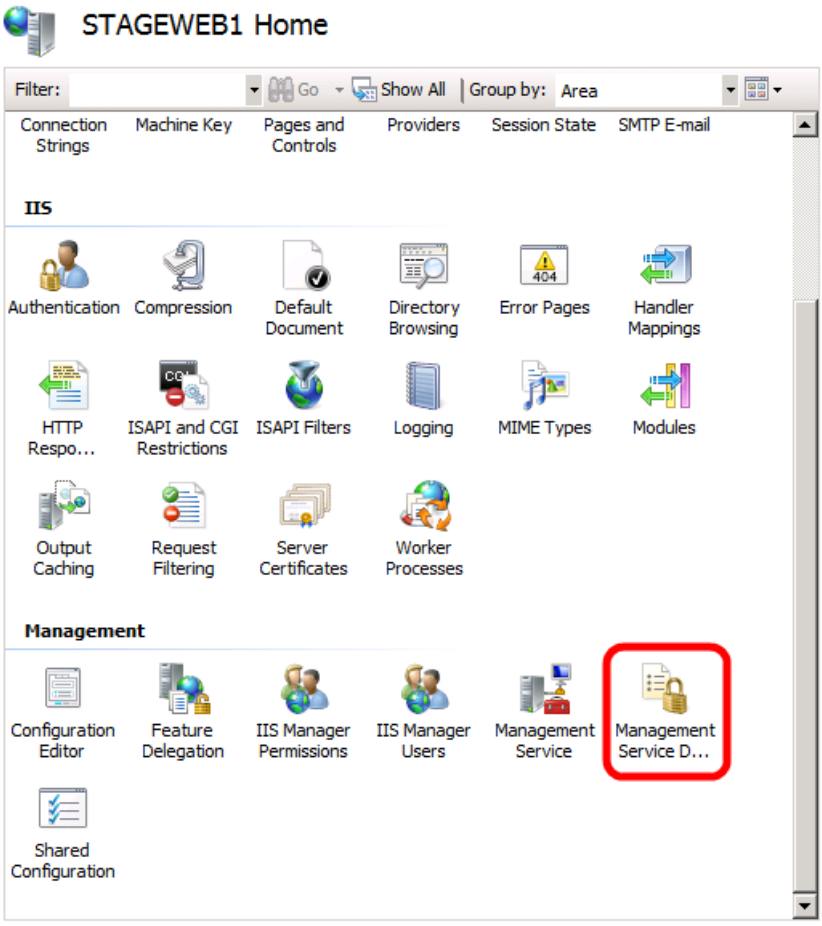


9. If you're prompted to save your settings, click **Yes**.

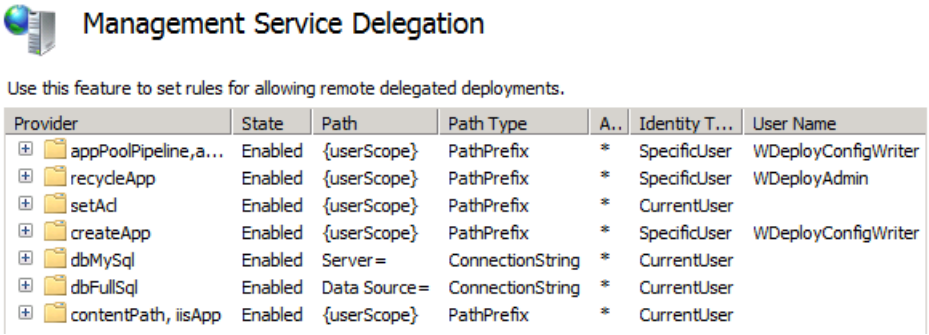
> **NOTE**
>
> You may also want to configure the service to start automatically. To do this, open the Services console, right-click **Web Management Service**, and then click **Properties**. In the **Startup type** dropdown list, select **Automatic**, and then click **OK**.

10. In the **Connections** pane, click the server node again to return to the top-level settings.

11. In the center pane, under **Management**, double-click **Management Service Delegation**.



12. Verify that the center pane contains a set of rules.



Provider	State	Path	Path Type	A..	Identity T...	User Name
appPoolPipeline,a...	Enabled	{userScope}	PathPrefix	*	SpecificUser	WDeployConfigWriter
recycleApp	Enabled	{userScope}	PathPrefix	*	SpecificUser	WDeployAdmin
setAcl	Enabled	{userScope}	PathPrefix	*	CurrentUser	
createApp	Enabled	{userScope}	PathPrefix	*	SpecificUser	WDeployConfigWriter
dbMySql	Enabled	Server=	ConnectionString	*	CurrentUser	
dbFullSql	Enabled	Data Source=	ConnectionString	*	CurrentUser	
contentPath, iisApp	Enabled	{userScope}	PathPrefix	*	CurrentUser	

These rules allow authorized Web Management Service users to use various Web Deploy providers. For example, to deploy web applications and content to IIS through the Web Deploy Handler, there must be a delegation rule that allows all authenticated Web Management Service users to use the **contentPath** and **iisApp** providers (the last rule that you can see in the screenshot).

If you installed products and components in the order described in this topic, the latest version of Web Deploy should automatically add all the required delegation rules to the Web Management Service. If the Management Service Delegation page does not show any rules, you'll need to create them yourself. For

instructions on how to do this, see Configure the Web Deployment Handler.

13. In the **Connections** pane, click the server node again to return to the top-level settings.

Create and Configure an IIS Website

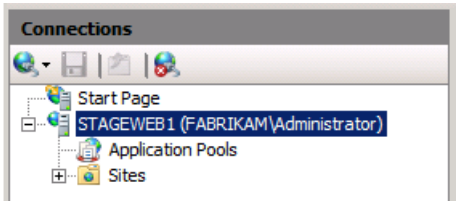
Before you can deploy web content to your server, you need to create and configure an IIS website to host the content. Web Deploy can only deploy web packages to an existing IIS website; it can't create the website for you. You also need to do a little extra configuration to allow your non-administrator account to deploy content remotely. At a high level, you'll need to complete these tasks:

- Create a folder on the file system to host your content.
- Create an IIS website to serve the content, and associate it with the local folder.
- Grant read permissions to the application pool identity on the local folder.
- Grant the necessary IIS permissions to the domain account that will deploy your web application.

Although there's nothing stopping you from deploying content to the default website in IIS, this approach is not recommended for anything other than test or demonstration scenarios. To simulate a production environment, you should create a new IIS website with settings that are specific to the requirements of your application.

To create an IIS website

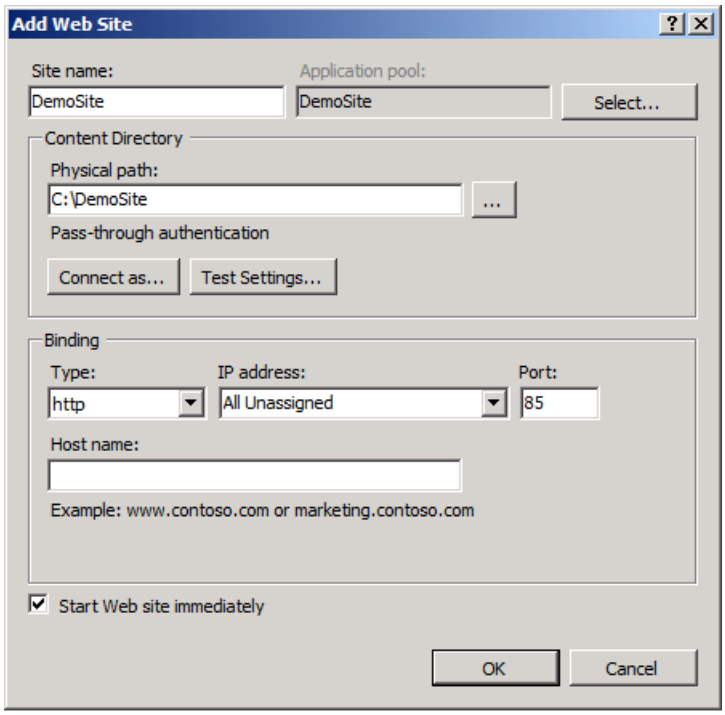
1. On the local file system, create a folder to store your content (for example, **C:\DemoSite**).
2. On the **Start** menu, point to **Administrative Tools**, and then click **Internet Information Services (IIS) Manager**.
3. In IIS Manager, in the **Connections** pane, expand the server node (for example, **STAGEWEB1**).



4. Right-click the **Sites** node, and then click **Add Web Site**.
5. In the **Site name** box, type a name for the IIS website (for example, **DemoSite**).
6. In the **Physical path** box, type (or browse to) the path to your local folder (for example, **C:\DemoSite**).
7. In the **Port** box, type the port number on which you want to host the website (for example, **85**).

> **NOTE**
> The standard port numbers are 80 for HTTP and 443 for HTTPS. However, if you host this website on port 80, you'll need to stop the default website before you can access your site.

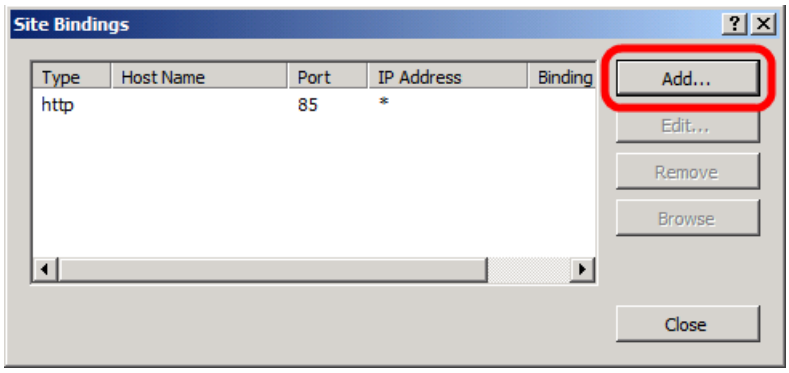
8. Leave the **Host name** box blank, unless you want to configure a Domain Name System (DNS) record for the website, and then click **OK**.



> **NOTE**
>
> In a production environment, you'll likely want to host your website on port 80 and configure a host header, together with matching DNS records. For more information on configuring host headers in IIS 7, see Configure a Host Header for a Web Site (IIS 7). For more information on the DNS Server role in Windows Server, see DNS Server Overview and DNS Server.

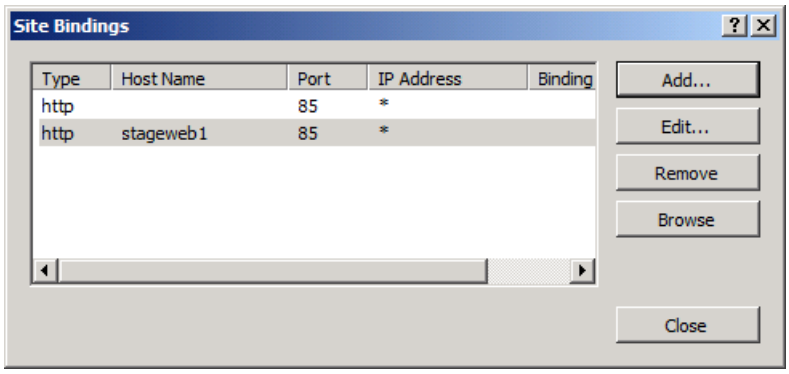
9. In the **Actions** pane, under **Edit Site**, click **Bindings**.

10. In the **Site Bindings** dialog box, click **Add**.



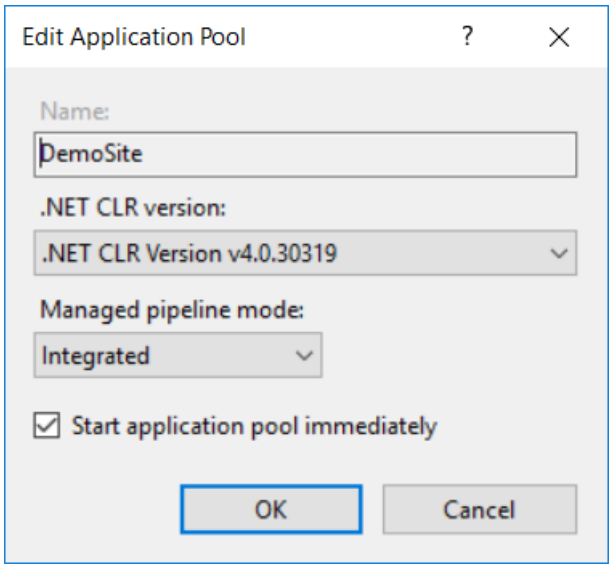
11. In the **Add Site Binding** dialog box, set the **IP address** and **Port** to match your existing site configuration.

12. In the **Host name** box, type the name of your web server (for example, **STAGEWEB1**), and then click **OK**.



> **NOTE**
> The first site binding allows you to access the site locally using the IP address and port or `http://localhost:85`. The second site binding allows you to access the site from other computers on the domain using the machine name (for example, http://stageweb1:85).

13. In the **Site Bindings** dialog box, click **Close**.
14. In the **Connections** pane, click **Application Pools**.
15. In the **Application Pools** pane, right-click the name of your application pool, and then click **Basic Settings**. By default, the name of your application pool will match the name of your website (for example, **DemoSite**).
16. In the **.NET CLR version** list, select **.NET CLR v4.0.30319**, and then click **OK**.



> **NOTE**
> The sample solution requires .NET Framework 4.0. This is not a requirement for Web Deploy in general.

In order for your website to serve content, the application pool identity must have read permissions on the local folder that stores the content. In IIS 7.5, application pools run with a unique application pool identity by default (in contrast to previous versions of IIS, where application pools would typically run using the Network Service account). The application pool identity is not a real user account and does not show up on any lists of users or groups—instead, it's created dynamically when the application pool is started. Each application pool identity is added to the local **IIS_IUSRS** security group as a hidden item.

To grant permissions to an application pool identity on a file or folder, you have two options:

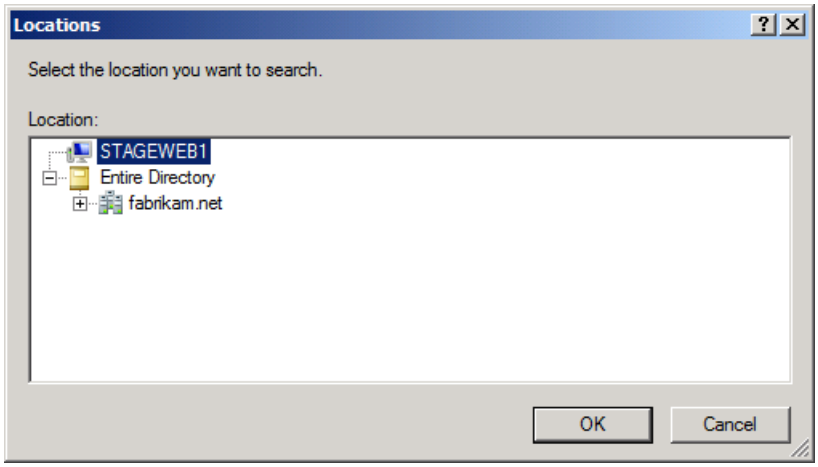
- Assign permissions to the application pool identity directly, using the format **IIS AppPool(application pool name)** (for example, **IIS AppPool\DemoSite**).
- Assign permissions to the **IIS_IUSRS** group.

The most common approach is to assign permissions to the local **IIS_IUSRS** group, because this approach lets you change application pools without reconfiguring file system permissions. The next procedure uses this group-based approach.

> **NOTE**
>
> For more information on application pool identities in IIS 7.5, see Application Pool Identities.

To configure folder permissions for an IIS website

1. In Windows Explorer, browse to the location of your local folder.
2. Right-click the folder, and then click **Properties**.
3. On the **Security** tab, click **Edit**, and then click **Add**.
4. Click **Locations**. In the **Locations** dialog box, select the local server, and then click **OK**.

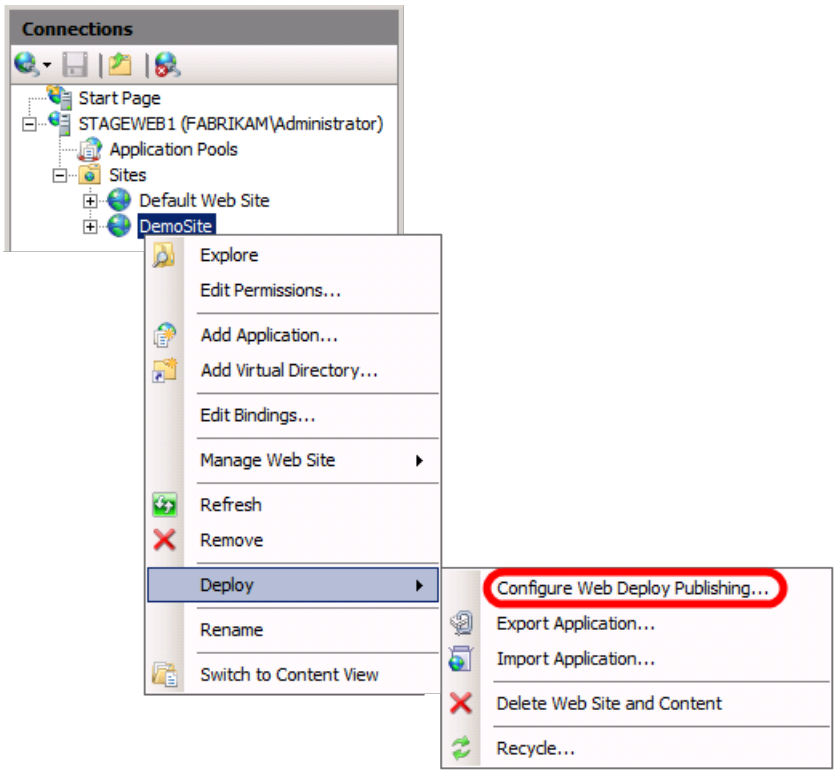


5. In the **Select Users or Groups** dialog box, type **IIS_IUSRS**, click **Check Names**, and then click **OK**.
6. In the **Permissions for (folder name)** dialog box, notice that the new group has been assigned the **Read & execute**, **List folder contents**, and **Read** permissions by default. Leave this unchanged and click **OK**.
7. Click **OK** to close the **(folder name) Properties** dialog box.

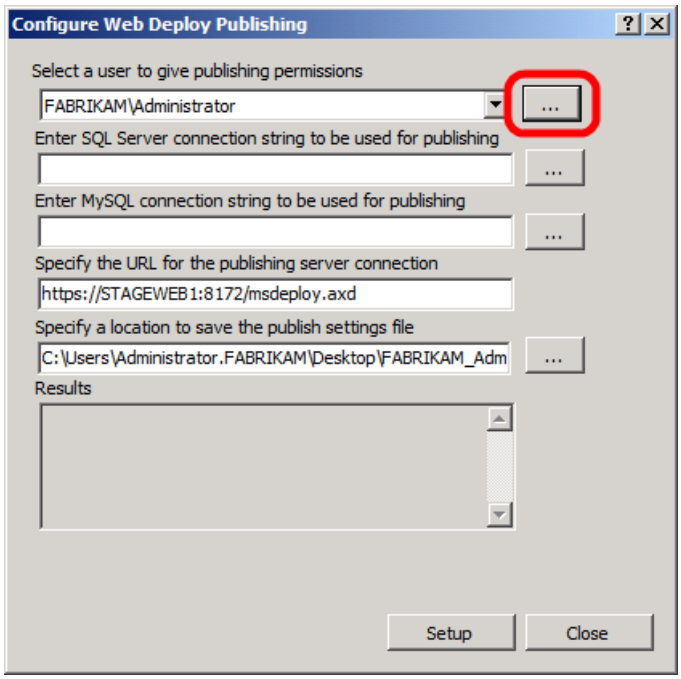
As a final task, you must grant the appropriate permissions to the non-administrator user whose credentials you'll use to deploy content. This user requires the permissions to deploy content remotely to your website.

To configure IIS website permissions for a non-administrator domain user

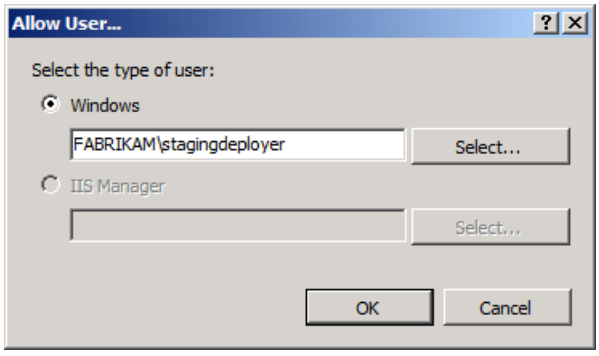
1. In IIS Manager, in the **Connections** pane, right-click your website node (for example, **DemoSite**), point to **Deploy**, and then click **Configure Web Deploy Publishing**.



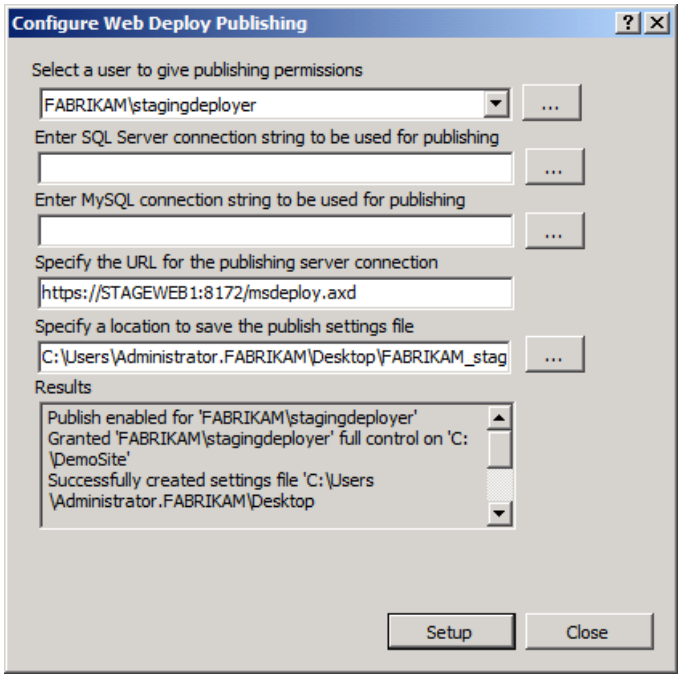
2. In the **Configure Web Deploy Publishing** dialog box, to the right of the **Select a user to give publishing permissions** list, click the ellipsis button.



3. In the **Allow User** dialog box, type the domain and user name of the account you want to use to deploy content, and then click **OK**.



4. In the **Configure Web Deploy Publishing** dialog box, click **Setup**.



> **NOTE**
>
> This operation performs two key functions in one step. First, it grants the user permission to modify the website remotely through the Web Management Service, according to the delegation rules you examined in the previous section. Second, it grants the user full control of the source folder for the website, which allows the user to add, modify, and set permissions on the website content.

5. In the **Configure Web Deploy Publishing** dialog box, click **Close**.

Configure Firewall Exceptions

By default, the IIS Web Management Service listens on TCP port 8172. If Windows Firewall is enabled on your web server, you'll need to create a new inbound rule to allow TCP traffic on port 8172 (all outbound traffic is permitted by default in Windows Firewall). If you use a third-party firewall, you'll need to create rules to allow traffic.

DIRECTION	FROM PORT	TO PORT	PORT TYPE
Inbound	Any	8172	TCP
Outbound	8172	Any	TCP

For more information on configuring rules in Windows Firewall, see Configuring Firewall Rules. For third-party firewalls, please consult your product documentation.

Conclusion

Your web server should now be ready to accept remote deployments to the Web Deploy Handler through the Web Management Service. Before you attempt to deploy a web application to the server, you may want to check these key points:

- Have you enabled basic authentication at the server level in IIS?
- Have you enabled remote connections to the Web Management Service?

- Have you started the Web Management Service?
- Are there management service delegation rules in place?
- Does the application pool identity have read access to the source folder for your website?
- Does the non-administrator user account have site-level permissions in IIS?
- Does your firewall allow incoming connections to the server on TCP port 8172?

Further Reading

For guidance on how to configure custom Microsoft Build Engine (MSBuild) project files to deploy web packages to the Web Deploy Handler, see Configuring Deployment Properties for a Target Environment.

PREVIOUS NEXT

Configuring a Web Server for Web Deploy Publishing (Offline Deployment)

9/16/2020 • 10 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic describes how to configure an IIS web server to support offline web publishing and deployment.

When you work with Internet Information Services (IIS) Web Deployment Tool (Web Deploy) 2.0 or later, there are three main approaches you can use to get your applications or sites onto a web server. You can:

- Use the *Web Deploy Remote Agent Service*. This approach requires less configuration of the web server, but you need to provide the credentials of a local server administrator in order to deploy anything to the server.
- Use the *Web Deploy Handler*. This approach is a lot more complex and requires more initial effort to set up the web server. However, when you use this approach, you can configure IIS to allow non-administrator users to perform the deployment. The Web Deploy Handler is only available in IIS version 7 or later.
- Use *offline deployment*. This approach requires the least configuration of the web server, but a server administrator must manually copy the web package onto the server and import it through IIS Manager.

For more information on the key features, advantages, and disadvantages of these approaches, see Choosing the Right Approach to Web Deployment.

Yes, if your network infrastructure or security restrictions prevent remote deployment. This is most likely to be the case in Internet-facing production environments, where the web servers are isolated—either physically or by firewalls and subnets—from the rest of your server infrastructure.

Obviously, this approach becomes less desirable if your web applications are updated on a regular basis. If your infrastructure allows it, you may want to consider enabling remote deployment, using either the Web Deploy Handler or the Web Deploy Remote Agent Service.

Task Overview

To configure the web server to support offline import and deployment of web packages, you'll need to:

- Install IIS 7.5 and the IIS 7 recommended configuration.
- Install Web Deploy 2.1 or later.
- Create an IIS website to host the deployed content.
- Disable the Web Deployment Agent Service.

To host the sample solution specifically, you'll also need to:

- Install the .NET Framework 4.0.
- Install ASP.NET MVC 3.

This topic will show you how to perform each of these procedures. The tasks and walkthroughs in this topic assume that you're starting with a clean server build running Windows Server 2008 R2. Before you continue, ensure that:

- Windows Server 2008 R2 Service Pack 1 and all available updates are installed.
- The server is domain-joined.
- The server has a static IP address.

> **NOTE**
> For more information on joining computers to a domain, see Joining Computers to the Domain and Logging On. For more information on configuring static IP addresses, see Configure a Static IP Address.

Install Products and Components

This section will guide you through installing the required products and components on the web server. Before you begin, a good practice is to run Windows Update to ensure that your server is fully up to date.

In this case, you need to install these things:

- **IIS 7 Recommended Configuration**. This enables the **Web Server (IIS)** role on your web server and installs the set of IIS modules and components that you need in order to host an ASP.NET application.
- **.NET Framework 4.0**. This is required to run applications that were built on this version of the .NET Framework.
- **Web Deployment Tool 2.1 or later**. This installs Web Deploy (and its underlying executable, MSDeploy.exe) on your server. Web Deploy integrates with IIS and lets you import and export web packages.
- **ASP.NET MVC 3**. This installs the assemblies you need to run MVC 3 applications.

> **NOTE**
> This walkthrough describes the use of the Web Platform Installer to install and configure various components. Although you don't have to use the Web Platform Installer, it simplifies the installation process by automatically detecting dependencies and ensuring that you always get the latest product versions. For more information, see Microsoft Web Platform Installer 3.0.

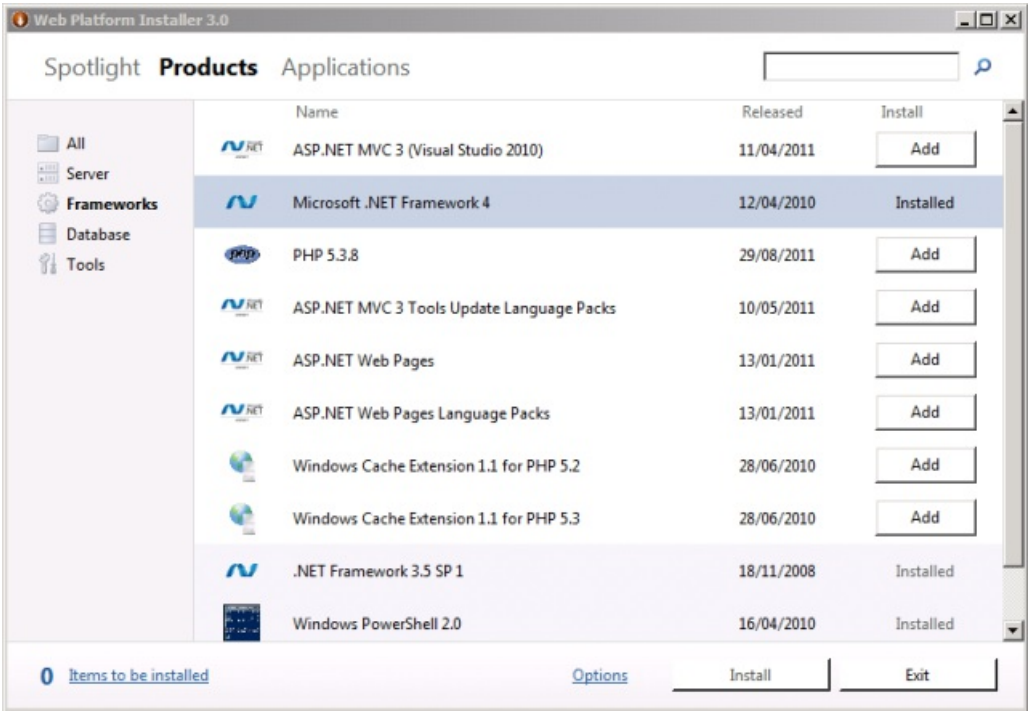
To install the required products and components

1. Download and install the Web Platform Installer.
2. When installation is complete, the Web Platform Installer will launch automatically.

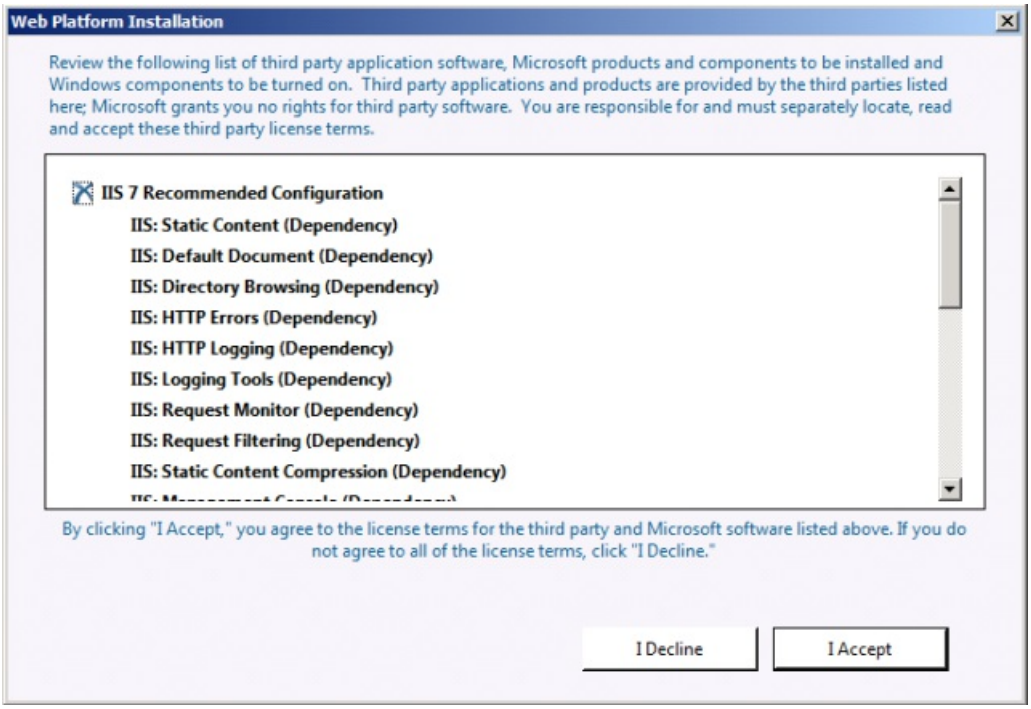
   > **NOTE**
   > You can now launch the Web Platform Installer at any time from the **Start** menu. To do this, on the **Start** menu, click **All Programs**, and then click **Microsoft Web Platform Installer**.

3. At the top of the **Web Platform Installer 3.0** window, click **Products**.
4. On the left side of the window, in the navigation pane, click **Frameworks**.
5. In the **Microsoft .NET Framework 4** row, if the .NET Framework is not already installed, click **Add**.

   > **NOTE**
   > You may have already installed the .NET Framework 4.0 through Windows Update. If a product or component is already installed, the Web Platform Installer will indicate this by replacing the **Add** button with the text **Installed**.



6. In the **ASP.NET MVC 3 (Visual Studio 2010)** row, click **Add**.
7. In the navigation pane, click **Server**.
8. In the **IIS 7 Recommended Configuration** row, click **Add**.
9. In the **Web Deployment Tool 2.1** row, click **Add**.
10. Click **Install**. The Web Platform Installer will show you a list of products—together with any associated dependencies—to be installed and will prompt you to accept the license terms.



11. Review the license terms, and if you consent to the terms, click **I Accept**.
12. When the installation is complete, click **Finish**, and then close the **Web Platform Installer 3.0** window.

If you installed the .NET Framework 4.0 before you installed IIS, you'll need to run the ASP.NET IIS Registration Tool (aspnet_regiis.exe) to register the latest version of ASP.NET with IIS. If you don't do this, you'll find that IIS will serve static content (like HTML files) without any problems, but it will return **HTTP Error 404.0 – Not**

Found when you attempt to browse to ASP.NET content. You can use the next procedure to ensure that ASP.NET 4.0 is registered.

To register ASP.NET 4.0 with IIS

1. Click **Start**, and then type **Command Prompt**.
2. In the search results, right-click **Command Prompt**, and then click **Run as administrator**.
3. In the Command Prompt window, navigate to the **%WINDIR%\Microsoft.NET\Framework\v4.0.30319** directory.
4. Type this command, and then press Enter:

```
aspnet_regiis -iru
```

5. If you plan to host 64-bit web applications at any point, you should also register the 64-bit version of ASP.NET with IIS. To do this, in the Command Prompt window, navigate to the **%WINDIR%\Microsoft.NET\Framework64\v4.0.30319** directory.
6. Type this command, and then press Enter:

```
aspnet_regiis -iru
```

As a good practice, use Windows Update again at this point to download and install any available updates for the new products and components you've installed.

Configure the IIS Website

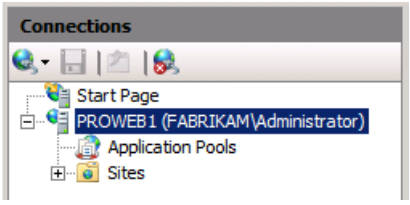
Before you can deploy web content to your server, you need to create and configure an IIS website to host the content. Web Deploy can only deploy web packages to an existing IIS website; it can't create the website for you. At a high level, you'll need to complete these tasks:

- Create a folder on the file system to host your content.
- Create an IIS website to serve the content, and associate it with the local folder.
- Grant read permissions to the application pool identity on the local folder.

Although there's nothing stopping you from deploying content to the default website in IIS, this approach is not recommended for anything other than test or demonstration scenarios. To simulate a production environment, you should create a new IIS website with settings that are specific to the requirements of your application.

To create and configure an IIS website

1. On the local file system, create a folder to store your content (for example, **C:\DemoSite**).
2. On the **Start** menu, point to **Administrative Tools**, and then click **Internet Information Services (IIS) Manager**.
3. In IIS Manager, in the **Connections** pane, expand the server node (for example, **PROWEB1**).

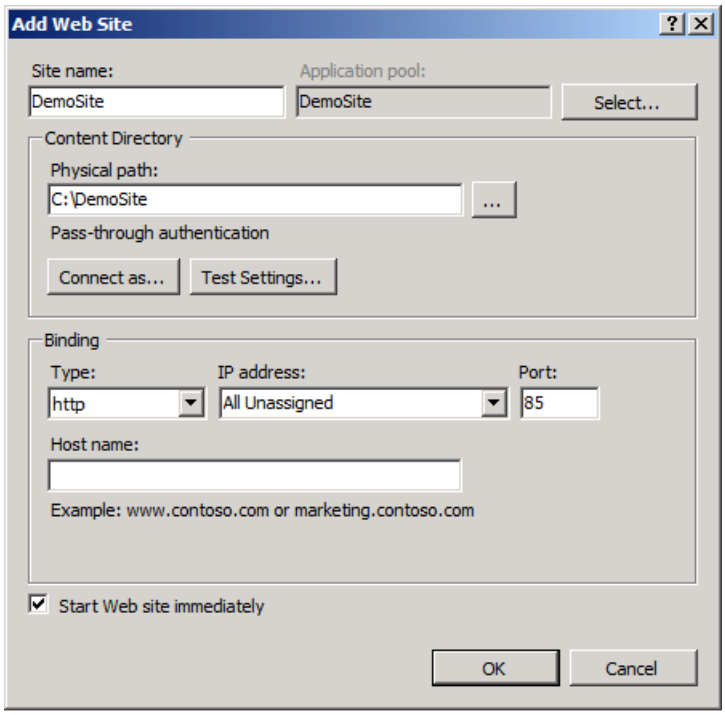


4. Right-click the **Sites** node, and then click **Add Web Site**.

5. In the **Site name** box, type a name for the IIS website (for example, **DemoSite**).
6. In the **Physical path** box, type (or browse to) the path to your local folder (for example, **C:\DemoSite**).
7. In the **Port** box, type the port number on which you want to host the website (for example, **85**).

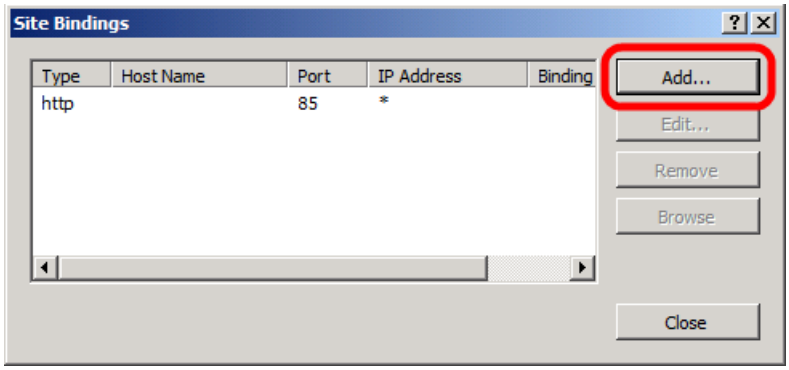
> **NOTE**
> The standard port numbers are 80 for HTTP and 443 for HTTPS. However, if you host this website on port 80, you'll need to stop the default website before you can access your site.

8. Leave the **Host name** box blank, unless you want to configure a Domain Name System (DNS) record for the website, and then click **OK**.



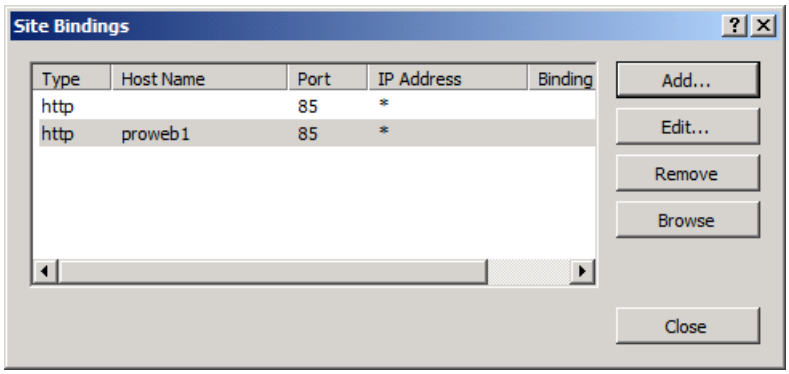
> **NOTE**
> In a production environment, you'll likely want to host your website on port 80 and configure a host header, together with matching DNS records. For more information on configuring host headers in IIS 7, see Configure a Host Header for a Web Site (IIS 7). For more information on the DNS Server role in Windows Server 2008 R2, see DNS Server Overview and DNS Server.

9. In the **Actions** pane, under **Edit Site**, click **Bindings**.
10. In the **Site Bindings** dialog box, click **Add**.



11. In the **Add Site Binding** dialog box, set the **IP address** and **Port** to match your existing site configuration.

12. In the **Host name** box, type the name of your web server (for example, **PROWEB1**), and then click **OK**.



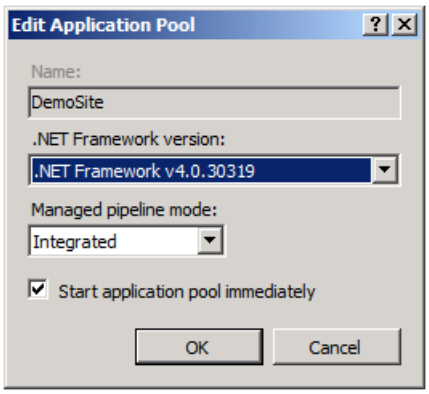
> **NOTE**
>
> The first site binding allows you to access the site locally using the IP address and port or `http://localhost:85`. The second site binding allows you to access the site from other computers on the domain using the machine name (for example, http://proweb1:85).

13. In the **Site Bindings** dialog box, click **Close**.

14. In the **Connections** pane, click **Application Pools**.

15. In the **Application Pools** pane, right-click the name of your application pool, and then click **Basic Settings**. By default, the name of your application pool will match the name of your website (for example, **DemoSite**).

16. In the **.NET Framework version** list, select **.NET Framework v4.0.30319**, and then click **OK**.



> **NOTE**
>
> The sample solution requires .NET Framework 4.0. This is not a requirement for Web Deploy in general.

In order for your website to serve content, the application pool identity must have read permissions on the local folder that stores the content. In IIS 7.5, application pools run with a unique application pool identity by default (in contrast to previous versions of IIS, where application pools would typically run using the Network Service account). The application pool identity is not a real user account and does not show up on any lists of users or groups—instead, it's created dynamically when the application pool is started. Each application pool identity is added to the local **IIS_IUSRS** security group as a hidden item.

To grant permissions to an application pool identity on a file or folder, you have two options:

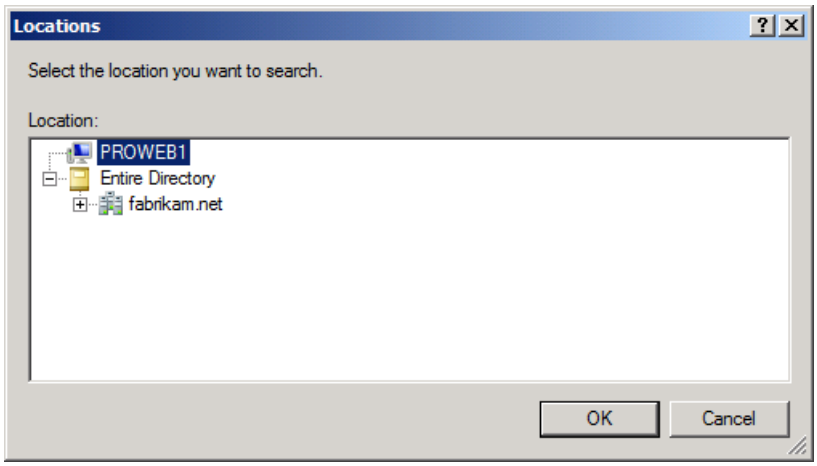
- Assign permissions to the application pool identity directly, using the format **IIS AppPool(application pool name)** (for example, **IIS AppPool\DemoSite**).
- Assign permissions to the **IIS_IUSRS** group.

The most common approach is to assign permissions to the local **IIS_IUSRS** group, because this approach lets you change application pools without reconfiguring file system permissions. The next procedure uses this group-based approach.

> **NOTE**
> For more information on application pool identities in IIS 7.5, see Application Pool Identities.

To configure folder permissions for an IIS website

1. In Windows Explorer, browse to the location of your local folder.
2. Right-click the folder, and then click **Properties**.
3. On the **Security** tab, click **Edit**, and then click **Add**.
4. Click **Locations**. In the **Locations** dialog box, select the local server, and then click **OK**.



5. In the **Select Users or Groups** dialog box, type **IIS_IUSRS**, click **Check Names**, and then click **OK**.
6. In the **Permissions for (folder name)** dialog box, notice that the new group has been assigned the **Read & execute**, **List folder contents**, and **Read** permissions by default. Leave this unchanged and click **OK**.
7. Click **OK** to close the **(folder name) Properties** dialog box.

Disable the Remote Agent Service

When you install Web Deploy, the Web Deployment Agent Service is installed and started automatically. This service allows you to deploy and publish web packages from a remote location. You won't be using the remote deployment capability in this scenario, so you should stop and disable the service.

> **NOTE**
> You don't need to stop the remote agent service in order to import and deploy a web package manually. However, it's a good practice to stop and disable the service if you don't plan to use it.

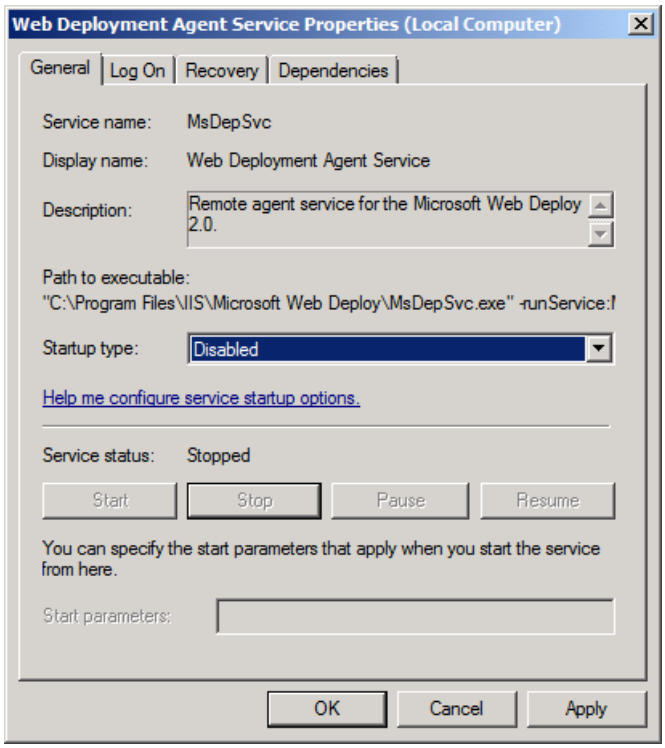
You can stop and disable a service in multiple ways, using various command-line utilities or Windows PowerShell cmdlets. This procedure describes a straightforward UI-based approach.

To stop and disable the remote agent service

1. On the **Start** menu, point to **Administrative Tools**, and then click **Services**.
2. In the Services console, locate the **Web Deployment Agent Service** row.

Name	Description	Status	Startup Type	Log On As
Thread Ordering Server	Provides ordered execution for a group of threads wit...		Manual	Local Service
TPM Base Services	Enables access to the Trusted Platform Module (TPM), ...		Manual	Local Service
UPnP Device Host	Allows UPnP devices to be hosted on this computer. If ...		Disabled	Local Service
User Profile Service	This service is responsible for loading and unloading us...	Started	Automatic	Local System
Virtual Disk	Provides management services for disks, volumes, file ...		Manual	Local System
Volume Shadow Copy	Manages and implements Volume Shadow Copies used ...		Manual	Local System
Web Deployment Agent Service	Remote agent service for the Microsoft Web Deploy 2.0.	Started	Automatic	Network Service
Windows Audio	Manages audio for Windows-based programs. If this s...		Manual	Local Service
Windows Audio Endpoint Builder	Manages audio devices for the Windows Audio service...		Manual	Local System

3. Right-click **Web Deployment Agent Service**, and then click **Properties**.
4. In the **Web Deployment Agent Service Properties** dialog box, click **Stop**.
5. In the **Startup type** list, select **Disabled**, and then click **OK**.



Conclusion

At this point, your web server is ready for offline web package deployment. Before you attempt to import web packages to an IIS website, you may want to check these key points:

- Have you registered ASP.NET 4.0 with IIS?
- Does the application pool identity have read access to the source folder for your website?
- Have you stopped the Web Deployment Agent Service?

PREVIOUS NEXT

Configuring a Database Server for Web Deploy Publishing

2/20/2020 • 11 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic describes how to configure a SQL Server 2008 R2 database server to support web deployment and publishing.

The tasks described in this topic are common to every deployment scenario—it doesn't matter whether your web servers are configured to use the IIS Web Deployment Tool (Web Deploy) Remote Agent Service, the Web Deploy Handler, or offline deployment or your application is running on a single web server or a server farm. The way you deploy the database may change according to security requirements and other considerations. For example, you might deploy the database with or without sample data, and you might deploy user role mappings or configure them manually after deployment. However, the way you configure the database server remains the same.

You don't have to install any additional products or tools to configuring a database server to support web deployment. Assuming that your database server and your web server run on different machines, you simply need to:

- Permit SQL Server to communicate using TCP/IP.
- Allow SQL Server traffic through any firewalls.
- Give the web server machine account a SQL Server login.
- Map the machine account login to any required database roles.
- Give the account that will run the deployment a SQL Server login and database creator permissions.
- To support repeat deployments, map the deployment account login to the **db_owner** database role.

This topic will show you how to perform each of these procedures. The tasks and walkthroughs in this topic assume that you're starting with a default instance of SQL Server 2008 R2 running on Windows Server 2008 R2. Before you continue, ensure that:

- Windows Server 2008 R2 Service Pack 1 and all available updates are installed.
- The server is domain-joined.
- The server has a static IP address.
- SQL Server 2008 R2 Service Pack 1 and all available updates are installed.

The SQL Server instance only needs to include the **Database Engine Services** role, which is included automatically in any SQL Server installation. However, for ease of configuration and maintenance, we recommend that you include the **Management Tools – Basic** and **Management Tools – Complete** server roles.

NOTE

For more information on joining computers to a domain, see Joining Computers to the Domain and Logging On. For more information on configuring static IP addresses, see Configure a Static IP Address. For more information on installing SQL Server, see Installing SQL Server 2008 R2.

Enable Remote Access to SQL Server

SQL Server uses TCP/IP to communicate with remote computers. If your database server and your web server are on different machines, you need to:

- Configure SQL Server networking settings to allow communication over TCP/IP.
- Configure any hardware or software firewalls to allow TCP traffic (and in some cases User Datagram Protocol (UDP) traffic) on the ports that the SQL Server instance uses.

To enable SQL Server to communicate over TCP/IP, use SQL Server Configuration Manager to change the network configuration for your SQL Server instance.

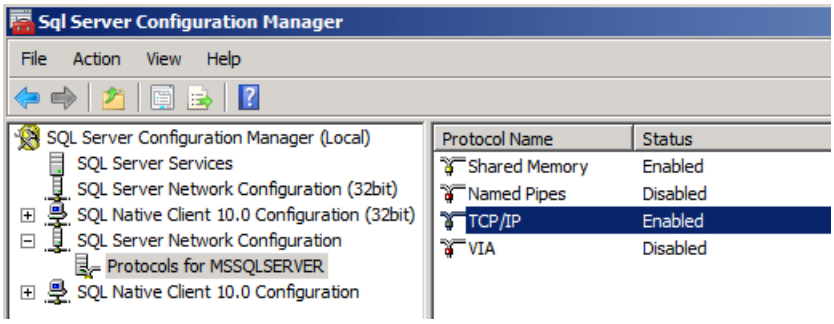
To enable SQL Server to communicate using TCP/IP

1. On the **Start** menu, point to **All Programs**, click **Microsoft SQL Server 2008 R2**, click **Configuration Tools**, and then click **SQL Server Configuration Manager**.

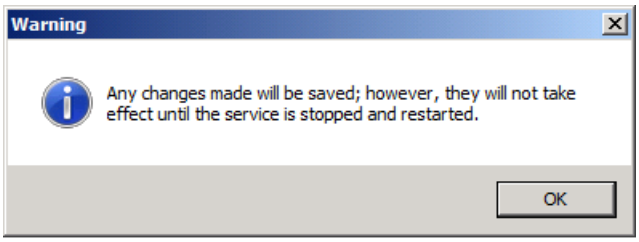
2. In the tree view pane, expand **SQL Server Network Configuration**, and then click **Protocols for MSSQLSERVER**.

   > **NOTE**
   > If you have installed multiple instances of SQL Server, you'll see a **Protocols for** *[instance name]* item for each instance. You need to configure network settings on an instance-by-instance basis.

3. In the details pane, right-click the **TCP/IP** row, and then click **Enable**.



4. In the **Warning** dialog box, click **OK**.



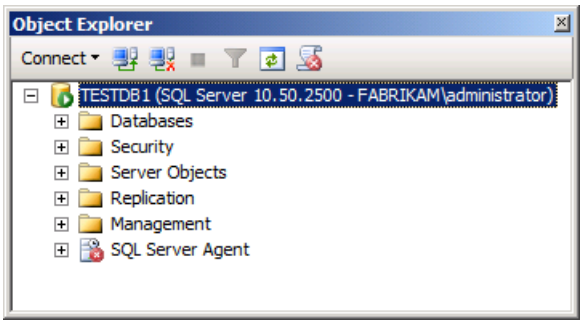
5. You need to restart the MSSQLSERVER service before your new network configuration will take effect. You can do that at a command prompt, from the Services console, or from SQL Server Management Studio. In this procedure, you'll use SQL Server Management Studio.

6. Close SQL Server Configuration Manager.

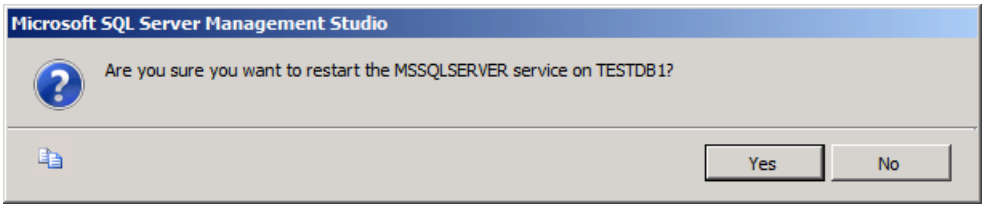
7. On the **Start** menu, point to **All Programs**, click **Microsoft SQL Server 2008 R2**, and then click **SQL Server Management Studio**.

8. In the **Connect to Server** dialog box, in the **Server name** box, type the name of the database server, and then click **Connect**.

9. In the **Object Explorer** pane, right-click the parent server node (for example, **TESTDB1**), and then click **Restart**.



10. In the **Microsoft SQL Server Management Studio** dialog box, click **Yes**.



11. When the service has restarted, close SQL Server Management Studio.

To allow SQL Server traffic through a firewall, you first need to know which ports your SQL Server instance is using. This will depend on how the SQL Server instance was created and configured:

- A *default instance* of SQL Server listens for (and responds to) requests on TCP port 1433.
- A *named instance* of SQL Server listens for (and responds to) requests on a dynamically assigned TCP port.
- If the SQL Server Browser service is enabled, clients can query the service on UDP port 1434 to find out which TCP port to use for a particular SQL Server instance. However, this service is often disabled for security reasons.

Assuming that you're using a default instance of SQL Server, you need to configure your firewall to allow traffic.

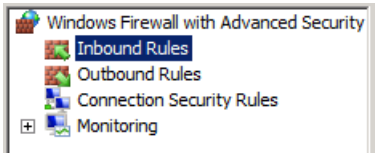
DIRECTION	FROM PORT	TO PORT	PORT TYPE
Inbound	Any	1433	TCP
Outbound	1433	Any	TCP

> **NOTE**
>
> Technically, a client computer will use a randomly assigned TCP port between 1024 and 5000 to communicate with SQL Server, and you can restrict your firewall rules accordingly. For more information on SQL Server ports and firewalls, see TCP/IP port numbers required to communicate to SQL over a firewall and How to: Configure a Server to Listen on a Specific TCP Port (SQL Server Configuration Manager).

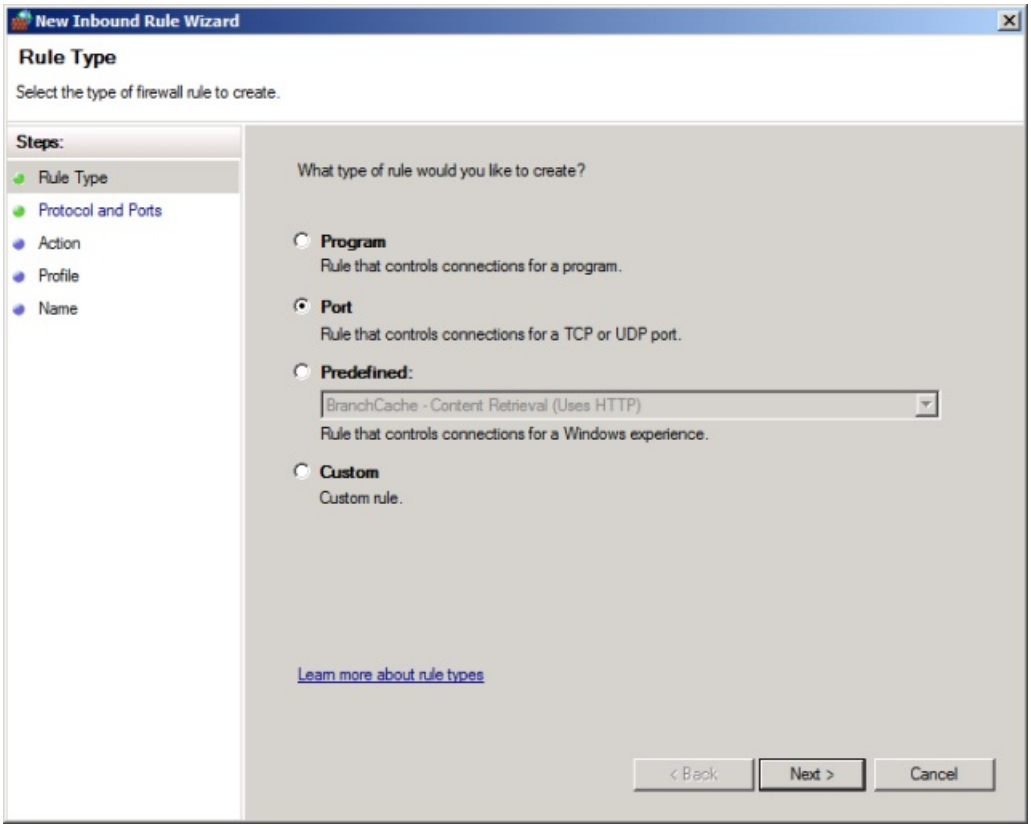
In most Windows Server environments, you'll likely have to configure Windows Firewall on the database server. By default, Windows Firewall allows all outbound traffic unless a rule specifically prohibits it. To enable your web server to reach your database, you need to configure an inbound rule that allows TCP traffic on the port number that the SQL Server instance uses. If you're using a default instance of SQL Server, you can use the next procedure to configure this rule.

To configure Windows Firewall to allow communication with a default SQL Server instance

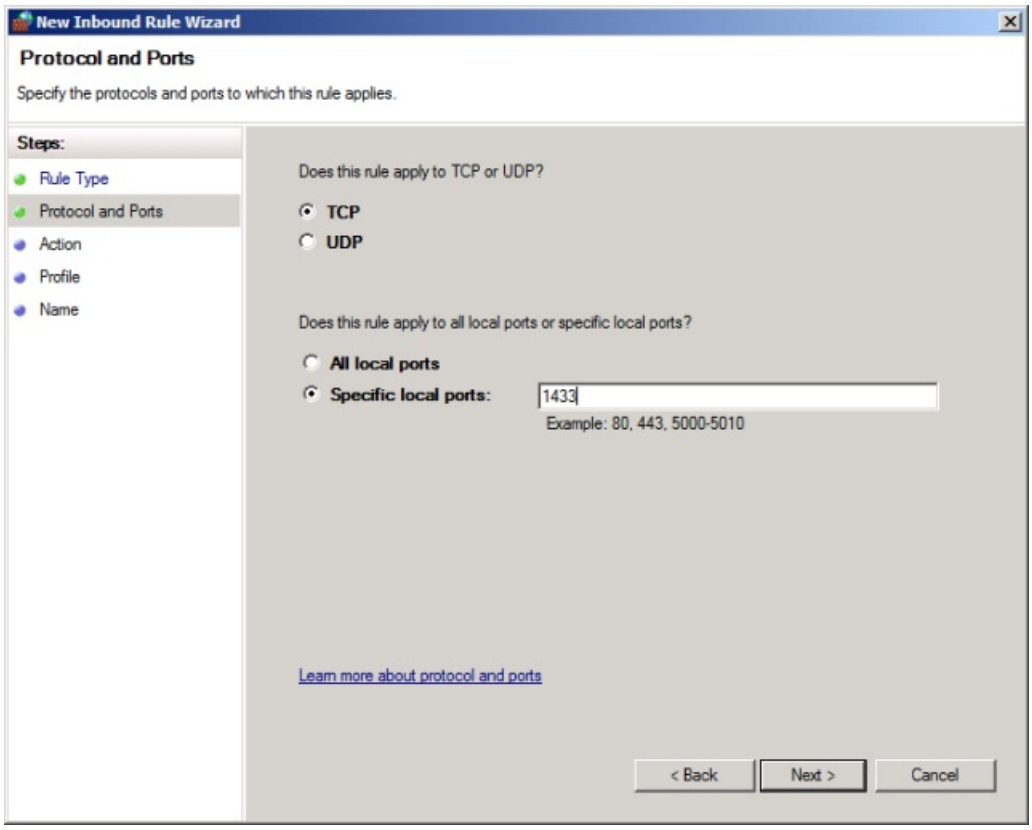
1. On the database server, on the **Start** menu, point to **Administrative Tools**, and then click **Windows Firewall with Advanced Security**.
2. In the tree view pane, click **Inbound Rules**.



3. In the **Actions** pane, under **Inbound Rules**, click **New Rule**.
4. In the New Inbound Rule Wizard, on the **Rule Type** page, select **Port**, and then click **Next**.

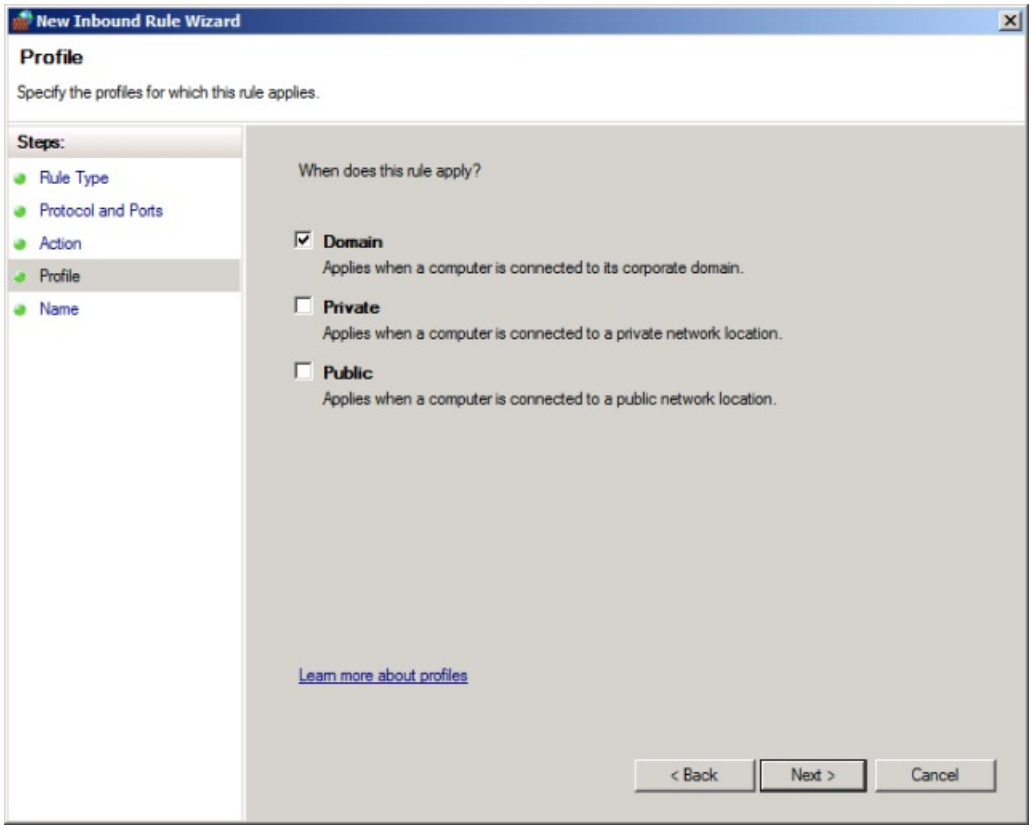


5. On the **Protocol and Ports** page, ensure that **TCP** is selected, and in the **Specific local ports** box, type **1433**, and then click **Next**.



6. On the **Action** page, leave **Allow the connection** selected and click **Next**.

7. On the **Profile** page, leave **Domain** selected, clear the **Private** and **Public** check boxes, and then click **Next**.



8. On the **Name** page, give the rule a suitably descriptive name (for example, **SQL Server default instance – network access**), and then click **Finish**.

For more information on configuring Windows Firewall for SQL Server, particularly if you need to communicate with SQL Server over non-standard or dynamic ports, see How to: Configure a Windows Firewall for Database Engine Access.

Configure Logins and Database Permissions

When you deploy a web application to Internet Information Services (IIS), the application runs using the identity of the application pool. In a domain environment, application pool identities use the machine account of the server on which they run to access network resources. Machine accounts take the form *[domain name]**[machine name]*$—for example, FABRIKAM\TESTWEB1$. To allow your web application to access a database across the network, you need to:

- Add a login for the web server machine account to the SQL Server instance.
- Map the machine account login to any required database roles (typically db_datareader and db_datawriter).

If your web application is running on a server farm, rather than a single server, you'll need to repeat these procedures for every web server in the server farm.

> **NOTE**
>
> For more information on application pool identities and accessing network resources, see Application Pool Identities.

You can approach these tasks in various ways. To create the login, you can either:

- Create the login manually on the database server, using Transact-SQL or SQL Server Management Studio.
- Use a SQL Server 2008 Server Project in Visual Studio to create and deploy the login.

A SQL Server login is a server-level object, rather than a database-level object, so it's not dependent on the database you want to deploy. As such, you can create the login at any point, and the easiest approach is often to create the login manually on the database server before you start deploying databases. You can use the next procedure to create a login in SQL Server Management Studio.

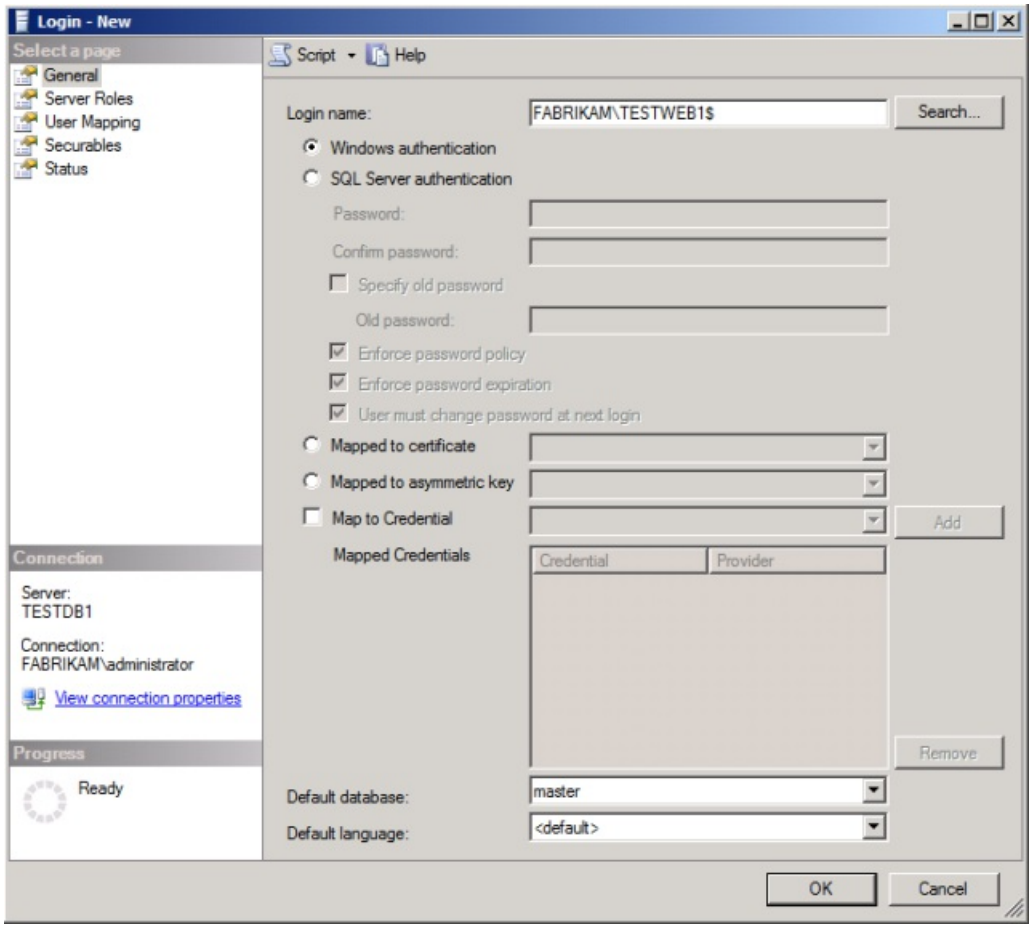
To create a SQL Server login for the web server machine account

1. On the database server, on the Start menu, point to All Programs, click Microsoft SQL Server 2008 R2, and then click SQL Server Management Studio.

2. In the Connect to Server dialog box, in the Server name box, type the name of the database server, and then click Connect.

3. In the Object Explorer pane, right-click Security, point to New, and then click Login.

4. In the Login – New dialog box, in the Login name box, type the name of your web server machine account (for example, FABRIKAM\TESTWEB1$).



5. Click OK.

At this point, your database server is ready for Web Deploy publishing. However, any solutions you deploy won't work until you map the machine account login to the required database roles. Mapping the login to database roles requires a lot more thought, as you can't map roles until after you've deployed the database. To map the machine account login to the required database roles, you can either:

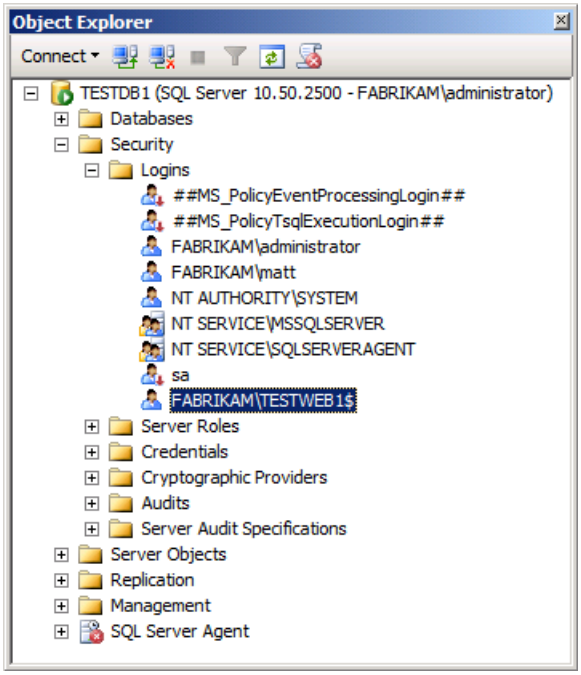
- Assign the database roles to the login manually, after you've deployed the database for the first time.
- Use a post-deployment script to assign the database roles to the login.

For more information on automating the creation of logins and database role mappings, see Deploying Database Role Memberships to Test Environments. Alternatively, you can use the next procedure to map the machine account login to the required database roles manually. Remember that you can't perform this procedure until *after* you've deployed the database.

To map database roles to the web server machine account login

1. Open SQL Server Management Studio as before.

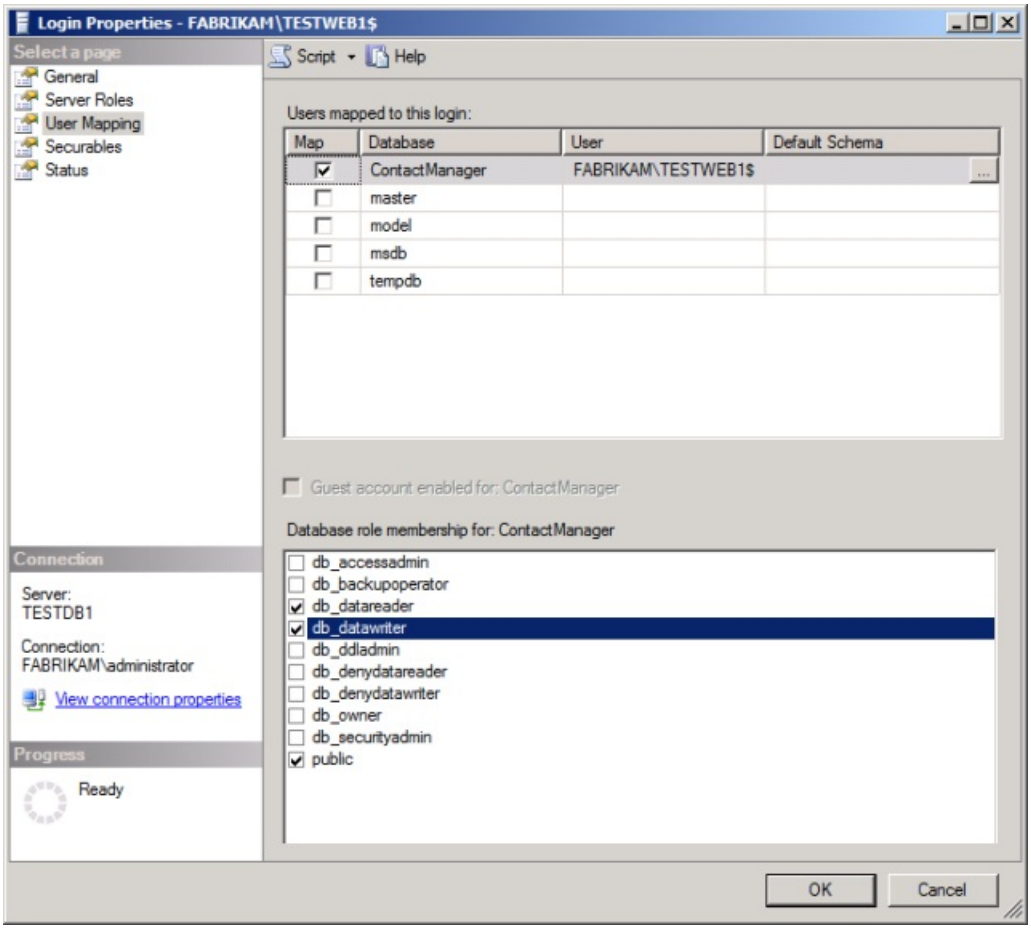
2. In the Object Explorer pane, expand the Security node, expand the Logins node, and then double-click the machine account login (for example, FABRIKAM\TESTWEB1$).



3. In the Login Properties dialog box, click User Mapping.

4. In the Users mapped to this login table, select the name of your database (for example, ContactManager).

5. In the Database role membership for: *[database name]* list, select the permissions required. In the case of the Contact Manager sample solution, you must select the db_datareader and db_datawriter roles.



6. Click OK.

While manually mapping database roles is often more than adequate for test environments, it's

less desirable for automated or one-click deployments to staging or production environments. You can find more information on automating this kind of task using post-deployment scripts in Deploying Database Role Memberships to Test Environments.

> **NOTE**
> For more information on server projects and database projects, see Visual Studio 2010 SQL Server Database Projects.

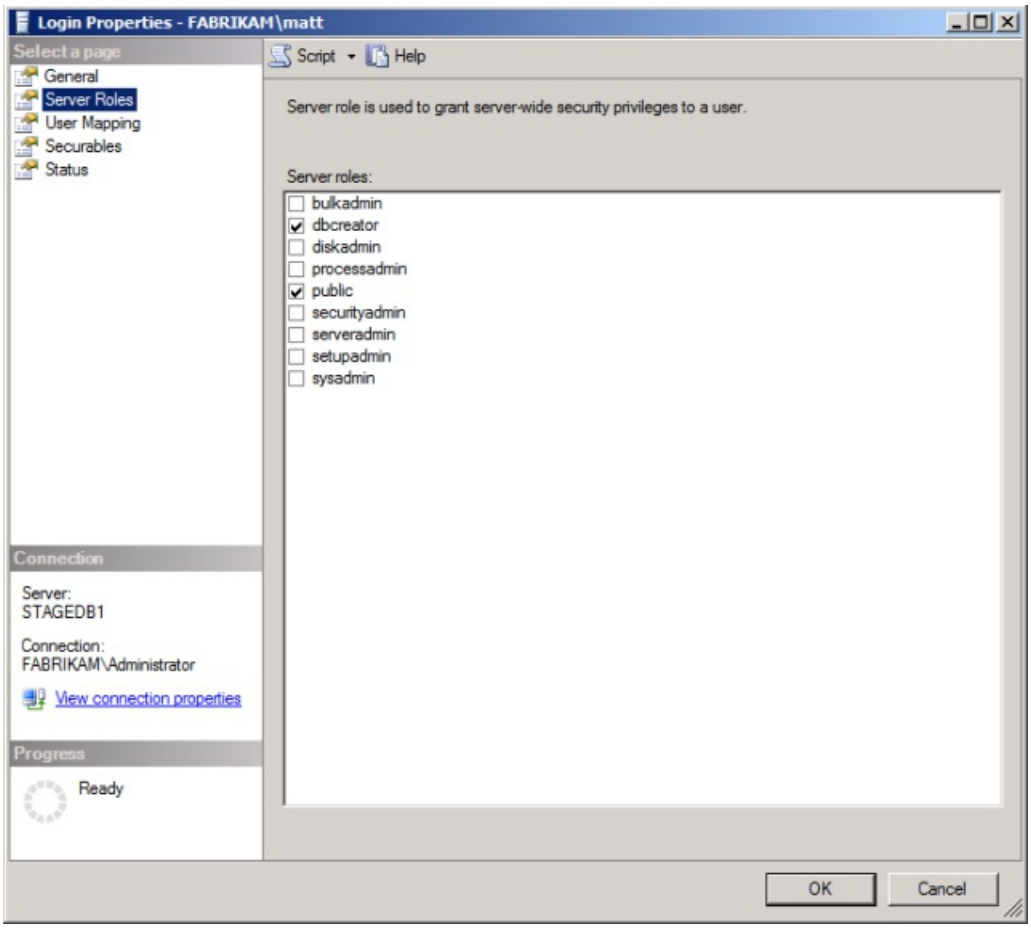
Configure Permissions for the Deployment Account

If the account that you'll use to run the deployment is not a SQL Server administrator, you'll also need to create a login for this account. In order to create the database, the account must be a member of the dbcreator server role or have equivalent permissions.

> **NOTE**
> When you use Web Deploy or VSDBCMD to deploy a database, you can use Windows credentials or SQL Server credentials (if your SQL Server instance is configured to support mixed mode authentication). The next procedure assumes that you want to use Windows credentials, but there's nothing stopping you from specifying a SQL Server user name and password in your connection string when you configure the deployment.

To set up permissions for the deployment account

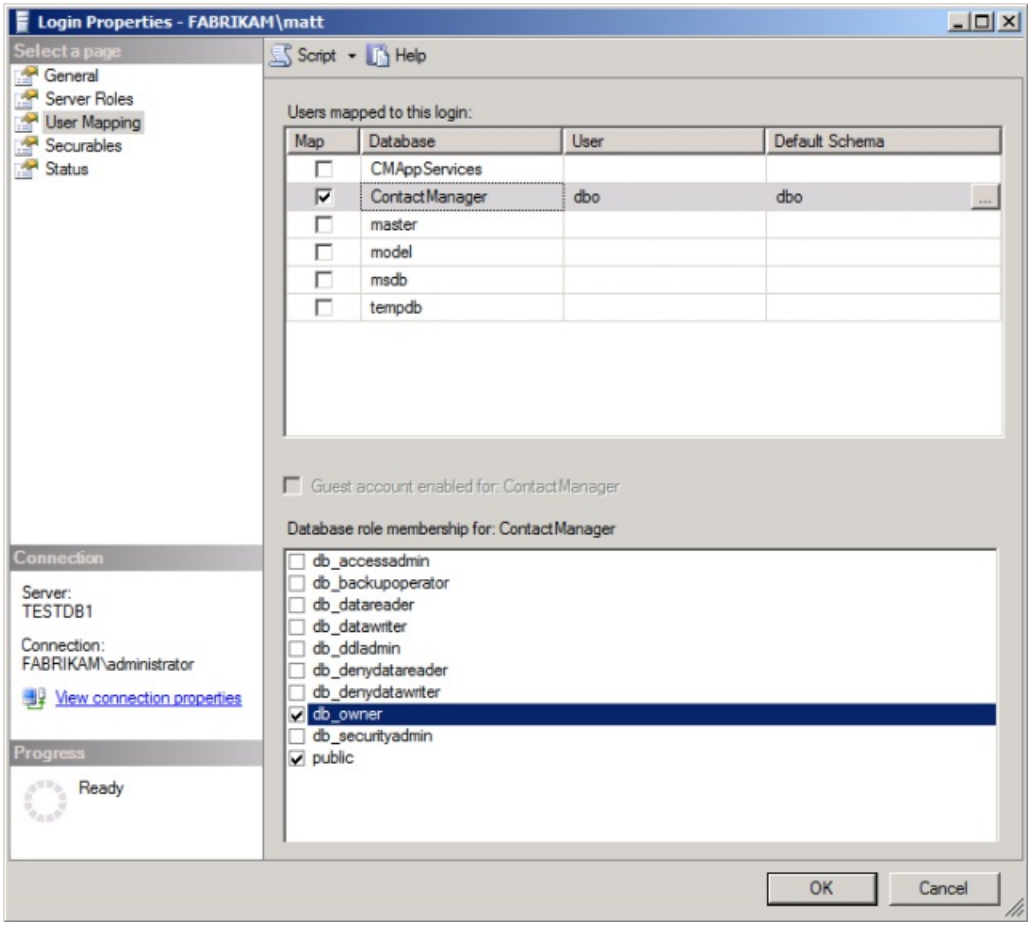
1. Open SQL Server Management Studio as before.
2. In the Object Explorer pane, right-click Security, point to New, and then click Login.
3. In the Login – New dialog box, in the Login name box, type the name of your deployment account (for example, FABRIKAM\matt).
4. In the Select a page pane, click Server Roles.
5. Select dbcreator, and then click OK.



To support subsequent deployments, you'll also need to add the deploying account to the db_owner role on the database after the first deployment. This is because on subsequent deployments you're modifying the schema of an existing database, rather than creating a new database. As described in the previous section, you can't add a user to a database role until you've created the database, for obvious reasons.

To map the deployment account login to the db_owner database role

1. Open SQL Server Management Studio as before.
2. In the Object Explorer window, expand the Security node, expand the Logins node, and then double-click the machine account login (for example, FABRIKAM\matt).
3. In the Login Properties dialog box, click User Mapping.
4. In the Users mapped to this login table, select the name of your database (for example, ContactManager).
5. In the Database role membership for: *[database name]* list, select the db_owner role.



6. Click OK.

Conclusion

Your database server should now be ready to accept remote database deployments and to allow remote IIS web servers to access your databases. Before you attempt to deploy and use databases, you may want to check these key points:

- Have you configured SQL Server to accept remote TCP/IP connections?
- Have you configured any firewalls to permit SQL Server traffic?
- Have you created a machine account login for every web server that will access SQL Server?
- Does your database deployment include a script to create user role mappings, or do you need to create these manually after you deploy the database for the first time?
- Have you created a login for the deployment account and added it to the dbcreator server role?

Further Reading

For guidance on deploying database projects, see Deploying Database Projects. For guidance on creating database role memberships by running a post-deployment script, see Deploying Database Role Memberships to Test Environments. For guidance on how to meet the unique deployment challenges that membership databases pose, see Deploying Membership Databases to Enterprise Environments.

PREVIOUS NEXT

Creating a Server Farm with the Web Farm Framework

2/20/2020 • 12 minutes to read • Edit Online

by Jason Lee

Download PDF

> This topic describes how to use the Web Farm Framework (WFF) 2.0 to create and configure a web server farm from a collection of servers.

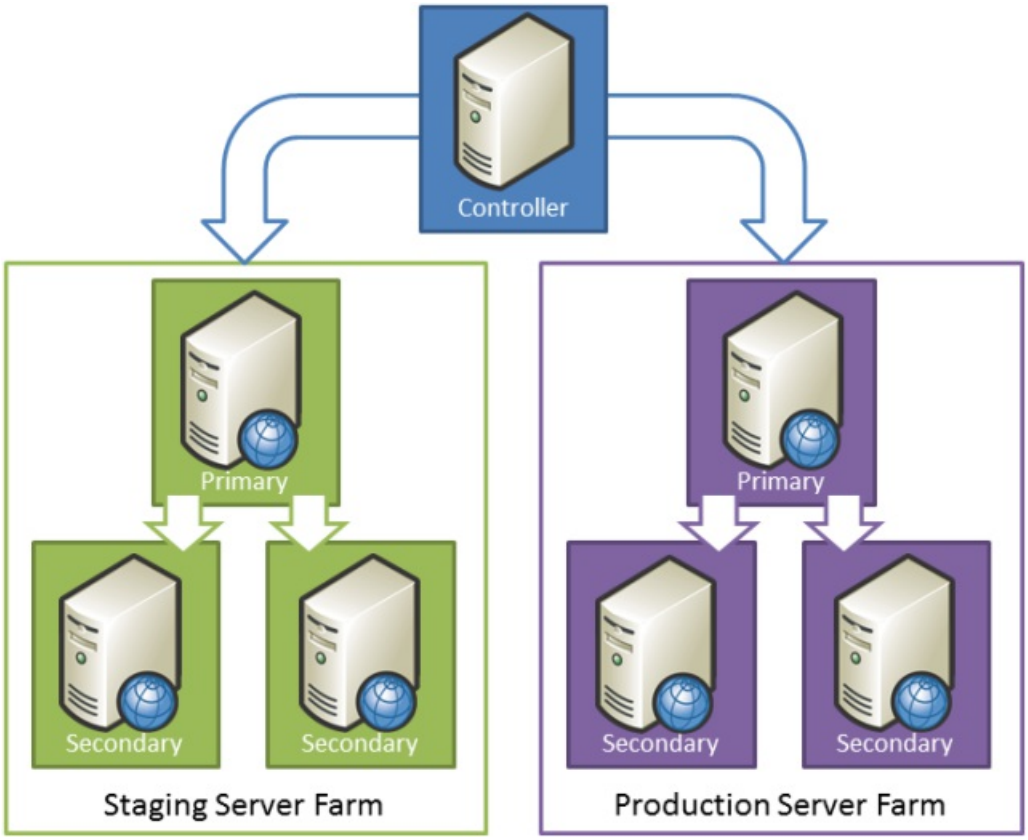
WFF lets you synchronize web platform products and components, web applications, websites, and configuration settings across multiple load-balanced web servers. In scenarios where you need more than one web server, like staging and production environments, this can vastly simplify your deployment and configuration process. You can deploy a web application to a single server—the *primary server*—and WFF will automatically replicate that web application on all the other web servers in the server farm.

Understanding the Web Farm Framework

You can use WFF 2.0 to provision, manage, and deploy content to a group of web servers. A WFF deployment consists of three key server roles:

- The *controller server*. You use this server to create and configure WFF server farms. The controller server manages the synchronization of web platform components, configuration settings, and applications between the web servers in a server farm. You install WFF 2.0 on the controller server, and the controller server will in turn install the WFF agent on each of the servers in a server farm. The controller server does not conceptually belong to any WFF server farm, and a single controller server can manage multiple server farms. In this scenario, you use a single WFF controller server to create and manage the staging server farm and the production server farm.
- The *primary server*. Each WFF server farm includes a single primary server. When you install web platform components or deploy applications to the primary server, the WFF synchronizes your changes to all the other servers in the server farm.
- The *secondary server*. Each WFF server farm includes one or more secondary servers. Any changes you make to the primary server are replicated to every secondary server within the server farm.

This shows how these server roles relate to the Fabrikam, Inc. staging and production environments:



In this scenario, the staging environment and the production environment are both configured as WFF server farms. A single WFF controller server manages both farms. Within each server farm, any changes to the primary server are replicated to every secondary server.

Before you start to configure your staging and production environments, we recommend that you read these articles to familiarize yourself with the key concepts of WFF 2.0:

- Overview of the Web Farm Framework 2.0 for IIS 7
- Setting up a Server Farm with the Web Farm Framework 2.0 for IIS 7
- System and Platform Requirements for the Web Farm Framework 2.0 for IIS 7

Task Overview

To complete the tasks and walkthroughs in this topic, you'll need at least three servers—one WFF controller, one primary web server for the server farm, and one or more secondary web servers for the server farm. You can add more secondary servers to a WFF server farm at any time. At a high level, to create and configure a WFF server farm for your staging or production environment you'll need to:

- Create a controller server by installing Internet Information Services (IIS) 7.5 and WFF 2.0.
- Prepare primary and secondary servers by creating a common administrator account and configuring firewall exceptions.
- Configure the server farm by using IIS Manager on the controller server.
- Configure load balancing using IIS Application Request Routing (ARR) or an alternative load-balancing technology.

The tasks and walkthroughs in this topic assume that you're starting with clean server builds running Windows Server 2008 R2. Before you begin, for each server, ensure that:

- Windows Server 2008 R2 Service Pack 1 and all available updates are installed.
- The server is domain-joined.
- The server has a static IP address.

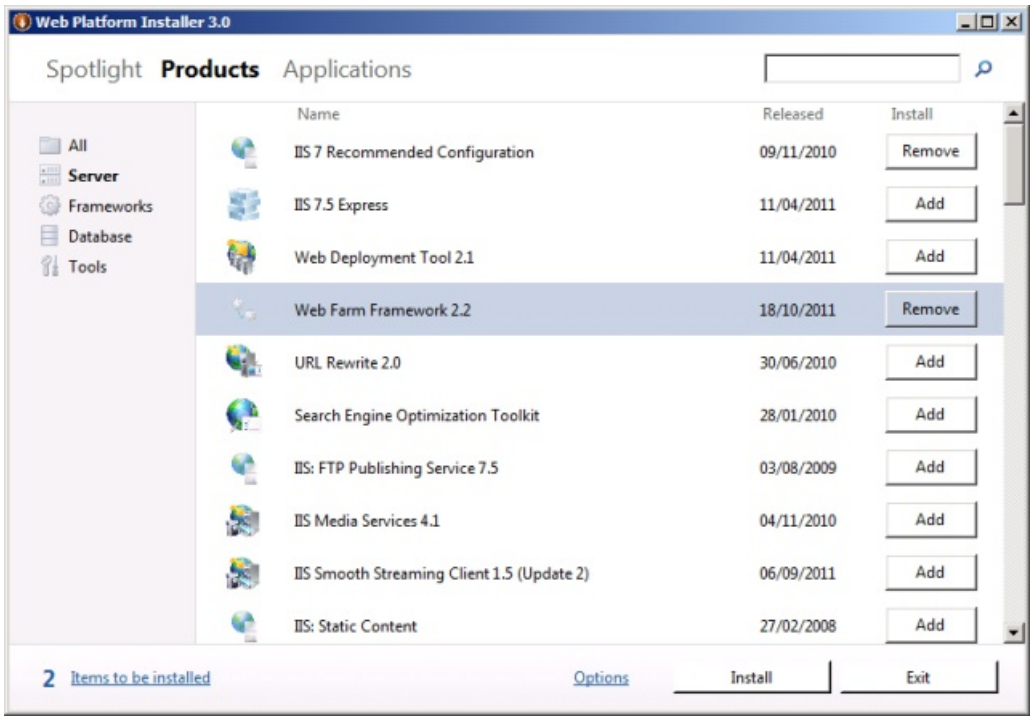
> **NOTE**
>
> For more information on joining computers to a domain, see Joining Computers to the Domain and Logging On. For more information on configuring static IP addresses, see Configure a Static IP Address.

Create the WFF Controller Server

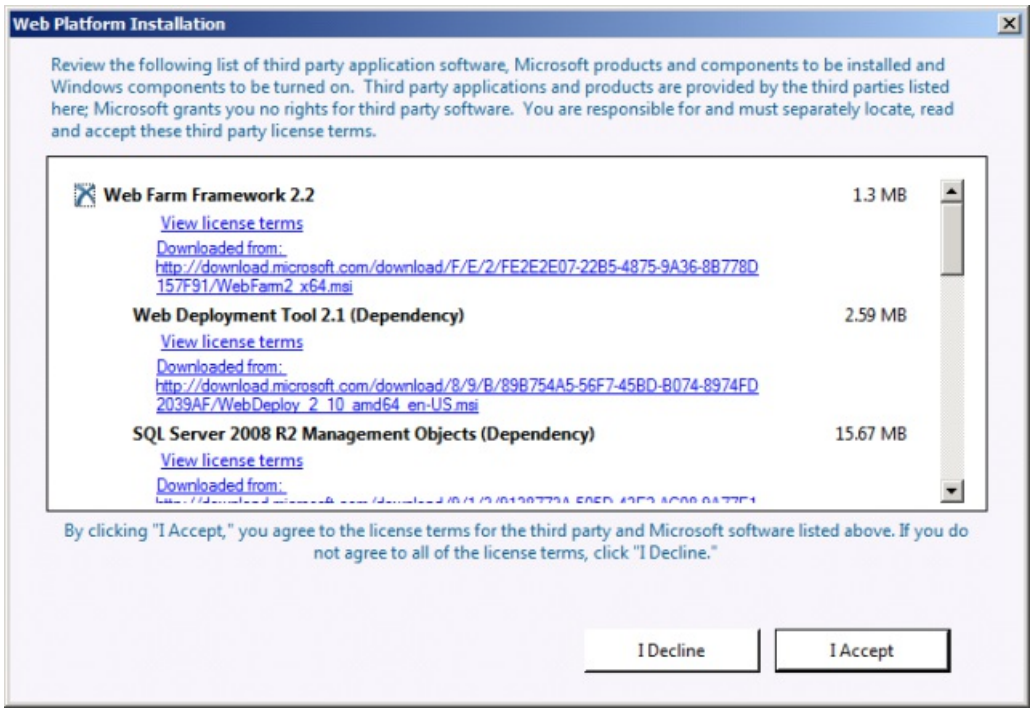
To create a WFF controller server, you'll need to install both IIS 7 or later and WFF 2.0 or later. Under the covers, WFF uses the IIS Web Deployment Tool (Web Deploy) 2.x to synchronize the servers in your farm. If you use the Web Platform Installer to install WFF, the installer will automatically download and install Web Deploy for you.

To create the WFF controller server

1. Download and install the Web Platform Installer.
2. At the top of the **Web Platform Installer 3.0** window, click **Products**.
3. On the left side of the window, in the navigation pane, click **Server**.
4. In the **IIS 7 Recommended Configuration** row, click **Add**.
5. In the **Web Farm Framework 2.*x*** row, click **Add**.



6. Click **Install**. Notice that the Web Platform Installer has added the Web Deployment Tool, along with various other dependencies, to the installation list.



7. Review the license terms, and if you consent to the terms, click **I Accept**.

8. When the installation is complete, click **Finish**, and then close the **Web Platform Installer 3.0** window.

Configure the Primary and Secondary Servers

Before you create a WFF server farm, you should complete some preparation tasks on the web servers that will make up the farm:

- Add firewall exceptions to allow the **Core Networking**, **Remote Administration**, and **File and Printer Sharing** features to communicate with the WFF controller server.
- Create a domain account (for example, **FABRIKAM\stagingfarm**) in Active Directory and add it to the local administrators group on each server. You'll use this account as the server farm administrator account when you create the server farm.

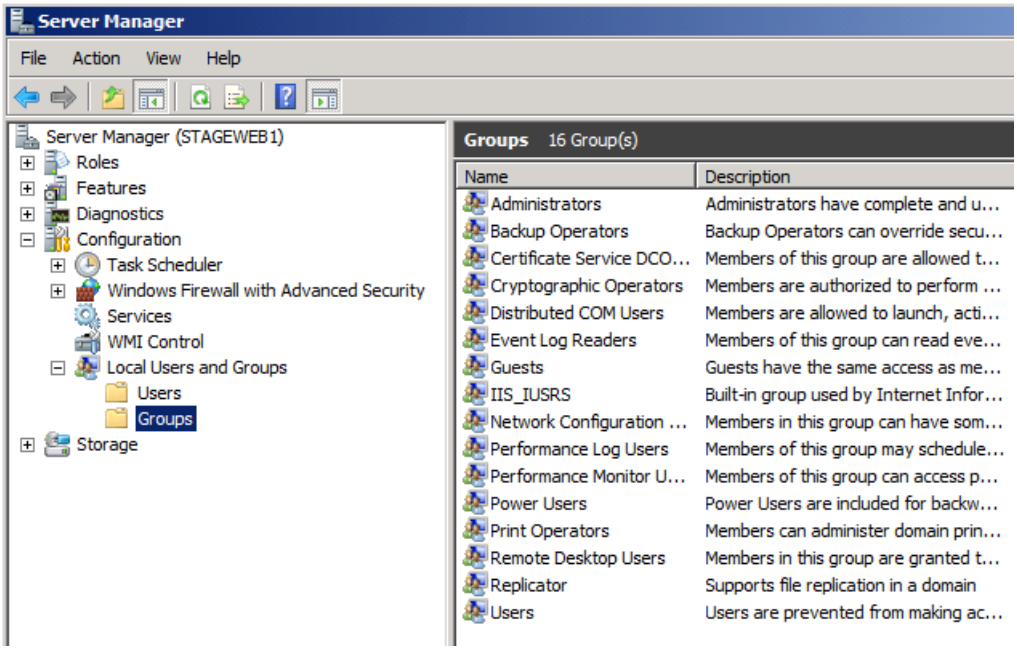
For more information on how to configure these firewall exceptions in Windows Firewall, see System and Platform Requirements for the Web Farm Framework 2.0 for IIS 7. For other firewall systems, consult your product documentation.

You can use the next procedure to add a domain account to the local administrators group in Windows Server 2008 R2. You should perform this procedure on every server that you want to add to the server farm—in other words, add the same domain account to the local administrators group on the primary server and on each secondary server.

To add a domain account to the local administrators group

1. On the **Start** menu, point to **Administrative Tools**, and then click **Server Manager**.

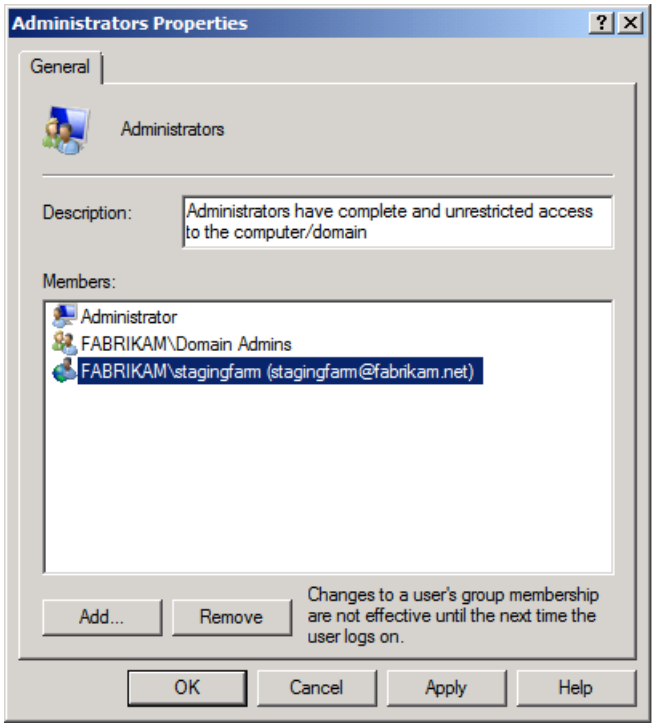
2. In the **Server Manager** window, in the tree view pane, expand **Configuration**, expand **Local Users and Groups**, and then click **Groups**.



3. In the **Groups** pane, double-click **Administrators**.

4. In the **Administrators Properties** dialog box, click **Add**.

5. In the **Select Users, Computers, Service Accounts, or Groups** dialog box, type (or browse) to your domain account (for example, **FABRIKAM\stagingfarm**), and then click **OK**.



6. In the **Administrators Properties** dialog box, click **OK**.

Your servers are now ready to be added to a server farm. In the case of the primary server, you can configure the server to meet your application requirements before or after you create the server farm—in both cases, the WFF will synchronize the servers by deploying the same products, components, or configuration to your secondary servers. For the sake of simplicity, this tutorial assumes that you'll configure the primary server when you've finished creating the server farm.

Create the WFF Server Farm

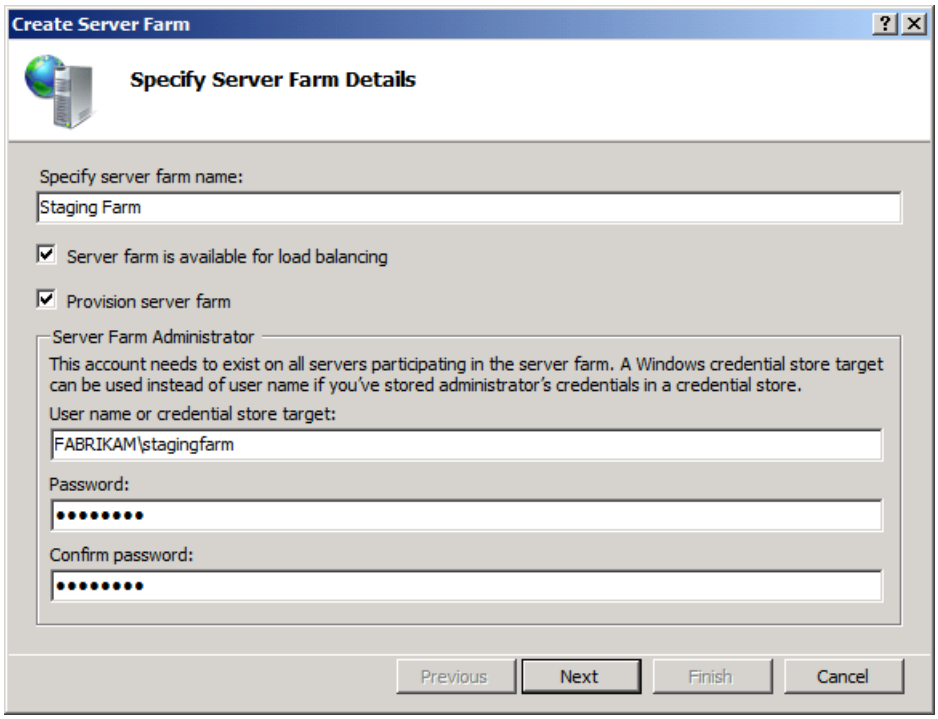
At this point, all your servers are ready to be added to a WFF server farm:

- You've installed WFF on the controller server.
- You've configured firewall exceptions on your primary and secondary web servers.
- You've added a domain account to the local administrators group on your primary and secondary web servers.

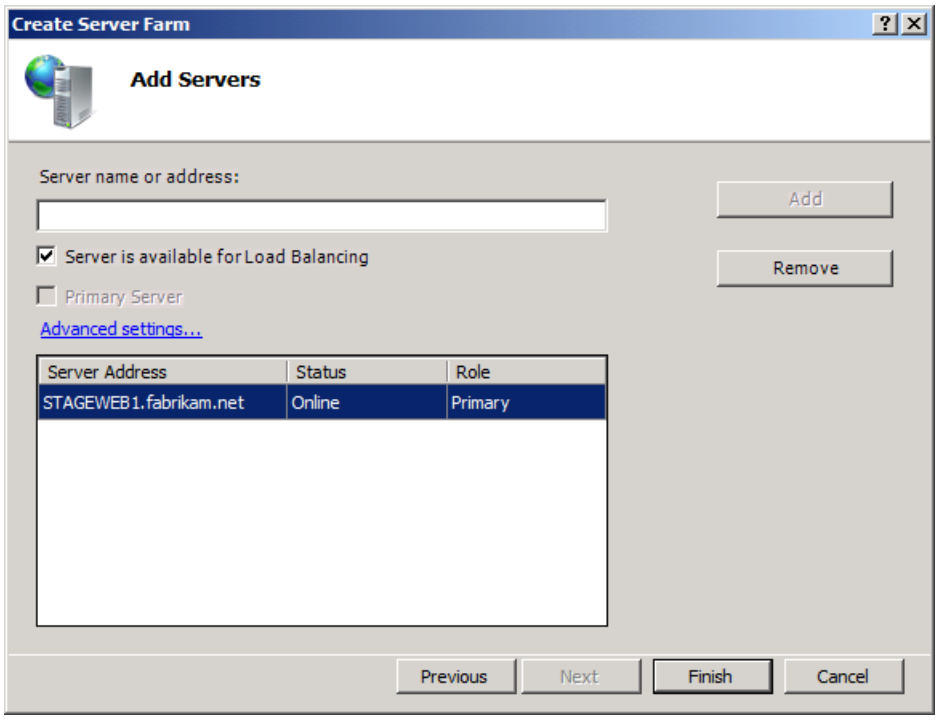
The next step is to create the server farm in WFF. You can do this from IIS Manager on the WFF controller server.

To create a WFF server farm

1. On the WFF controller server, on the **Start** menu, point to **Administrative Tools**, and then click **Internet Information Services (IIS) Manager**.
2. In the **Connections** pane, expand the local server node, right-click **Server Farms**, and then click **Create Server Farm**.
3. In the **Create Server Farm** dialog box, type a meaningful name for the server farm (for example, **Staging Farm**), and then select **Provision server farm**.
4. Type the user name and password of the domain account that you added to the local administrators group on each server.

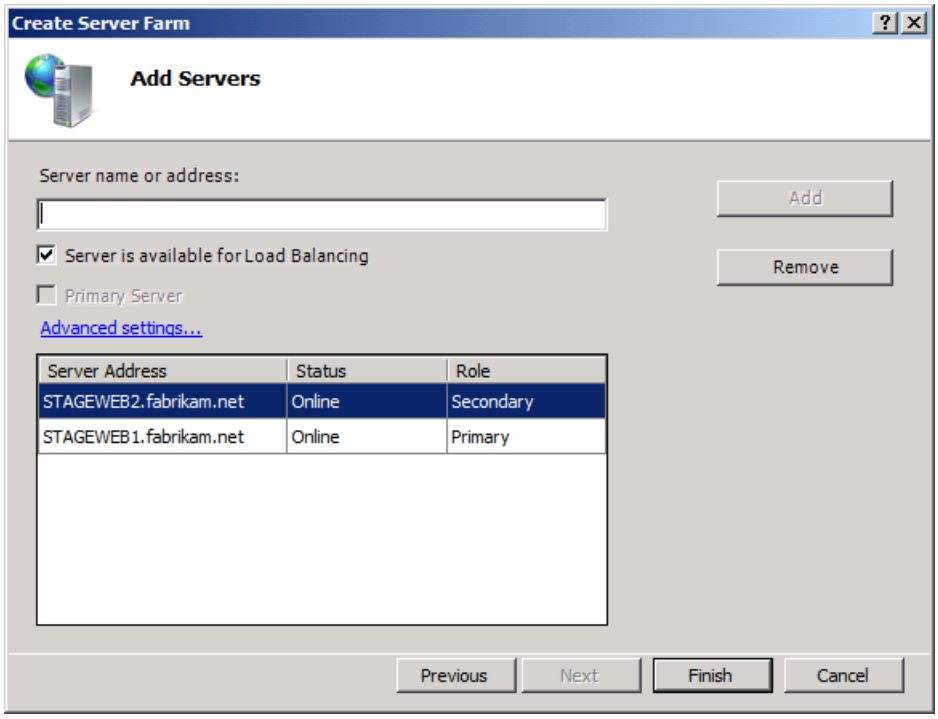


5. Click **Next**.
6. On the **Add Servers** page, type the fully qualified domain name (FQDN) of the primary server, select **Primary Server**, and then click **Add**.
7. At this point, WFF will attempt to contact the primary server using the credentials you provided. If the connection succeeds, the primary server will be added to the table on the **Add Servers** page.



> **NOTE**
>
> You might have noticed that **Server is available for Load Balancing** is selected by default. WFF uses the IIS ARR module to implement load balancing and thereby distribute requests across the web servers in your server farm. In most scenarios, you'd only clear the **Server is available for Load Balancing** option if you wanted to use a third-party load balancing solution instead.

8. On the **Add Servers** page, type the FQDN of your first secondary server, and then click **Add**.



9. Repeat step 7 for any additional secondary servers in your farm, and then click **Finish**.

Your WFF server farm is now up and running. Any web platform products or components that you install on the primary server, and any web applications or content that you deploy to the primary server, will be automatically provisioned on all your secondary servers.

WFF is a broad and complex topic, and you can learn more about it on the Microsoft Web Farm Framework 2.0

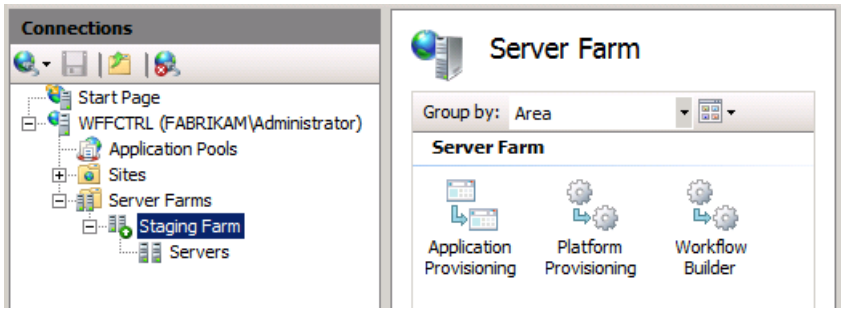
for IIS 7 website. For the time being, however, there are two features areas that you need to be aware of:

- *Application provisioning* is the process that replicates content from the primary server, like web applications and configuration settings, across all the secondary servers in the server farm. For example, if you deploy the Contact Manager sample solution to your primary staging server, the WFF application provisioning process will deploy this solution to all your secondary staging servers. By default, the application provisioning process runs every 30 seconds.
- *Platform provisioning* is the process that synchronizes web platform products and components from the primary server to all the secondary servers in the server farm. For example, if you install ASP.NET MVC 3 on your primary staging server, the platform provisioning process will use the Web Platform Installer to install ASP.NET MVC 3 on all your secondary staging servers. By default, the platform provisioning process runs every five minutes.

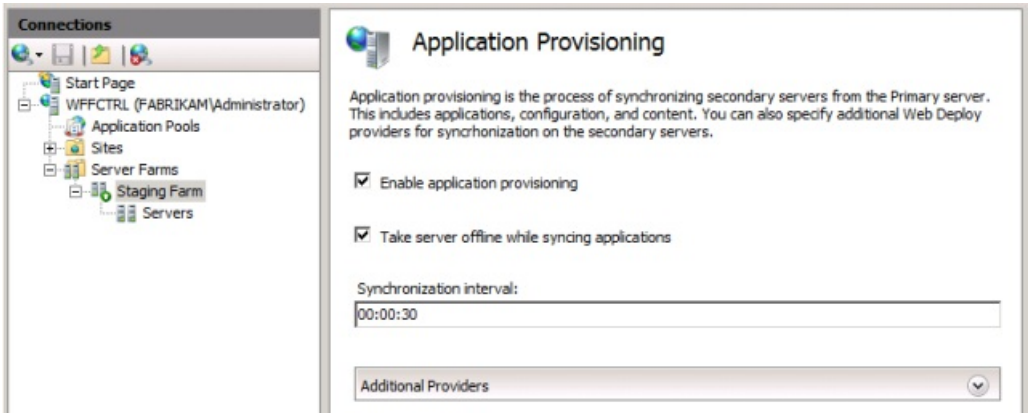
You can manage basic application and platform provisioning settings from IIS Manager on your WFF controller server.

Explore application and platform provisioning settings

1. In IIS Manager, in the **Connections** pane, select your server farm.

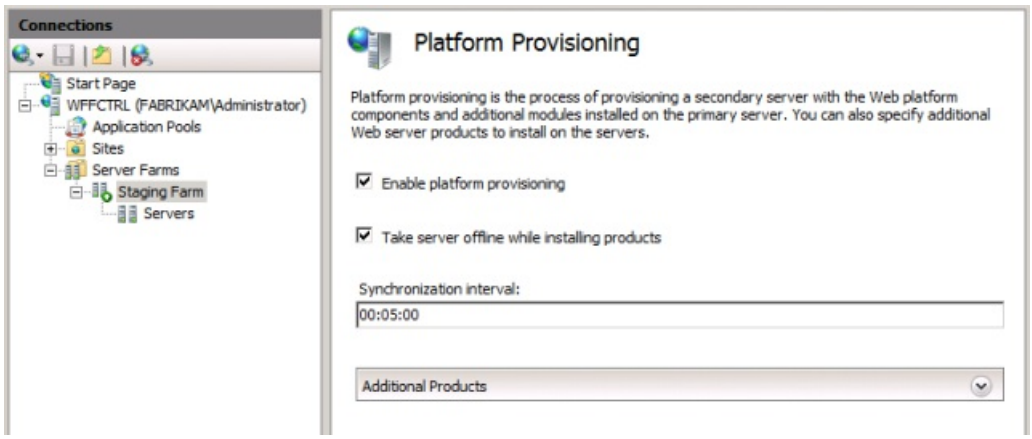


2. In the **Server Farm** pane, double-click **Application Provisioning**.



3. As you can see, the server farm is currently configured to synchronize web content and configuration settings between the primary server and the secondary servers every 30 seconds.

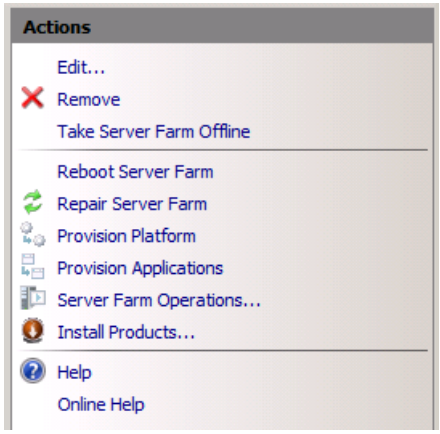
4. Click **Back**, and then double-click **Platform Provisioning**.



5. As you can see, the server farm is currently configured to synchronize web platform products and components between the primary server and the secondary servers every five minutes.

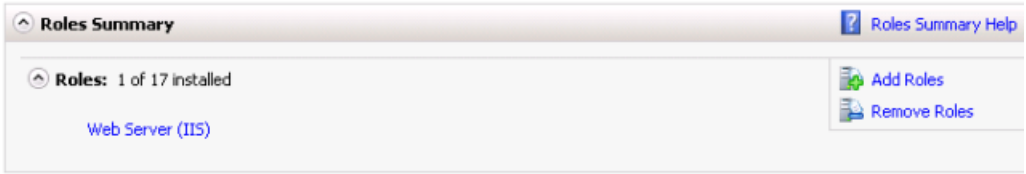
6. Click **Back**.

7. To force the server farm to synchronize web platform products immediately, in the **Actions** pane, click **Provision Platform**.

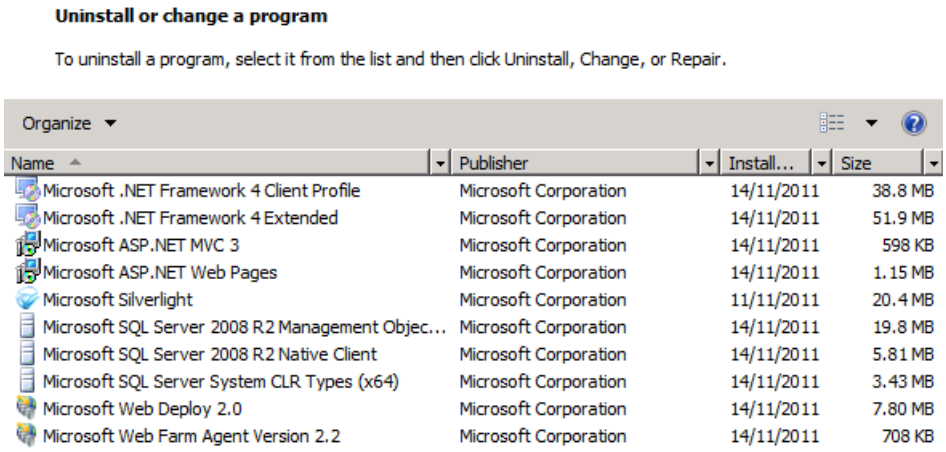


> **NOTE**
>
> Platform provisioning may take some time. The installer process runs in the background on the secondary servers in your server farm.

8. Once you've allowed sufficient time for the provisioning process to complete, you can verify that the products and components that you added to the primary server have now been replicated on the secondary servers. For example, you can log on to a secondary server and use the **Server Manager** window to verify that the web server role has been installed.



9. You can also check the installed programs list to verify that various web platform components have been added.



Configure Load Balancing

When you create a web farm, you need to set up some form of load balancing to distribute HTTP requests between your web servers. This could be Windows Server 2008 network load balancing, IIS ARR, or a third-party software-based or hardware-based load balancing solution.

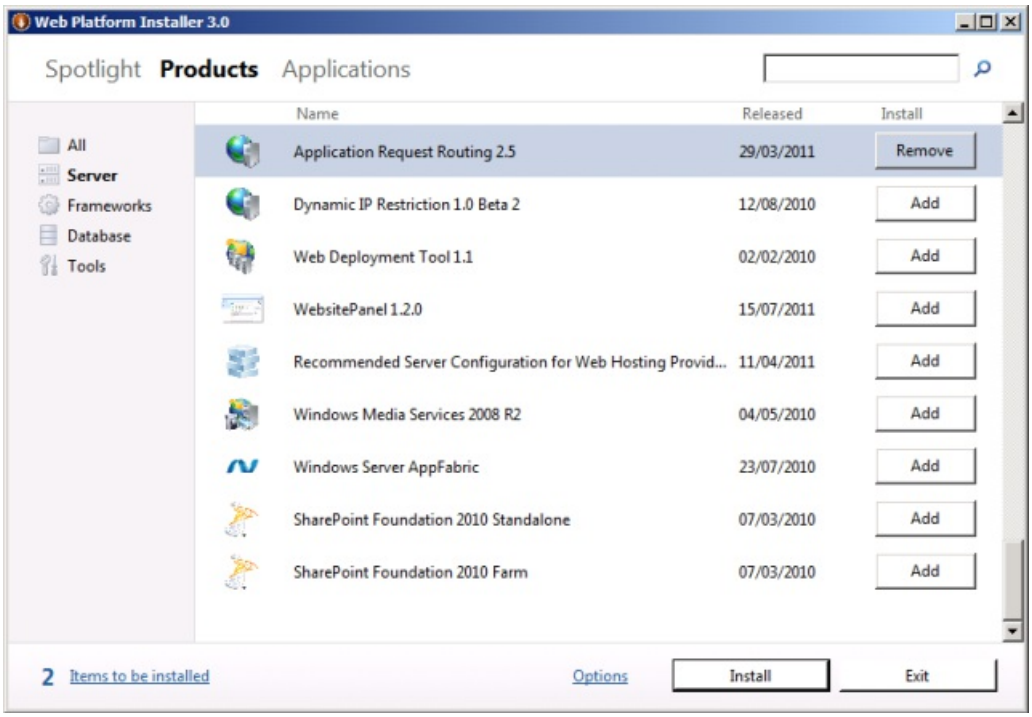
WFF is designed to integrate closely with IIS ARR. To take advantage of this integration, you need to install the ARR module on the WFF controller server. You then direct all your web traffic to the controller server, typically by configuring Domain Name System (DNS) records. The controller server will then distribute incoming requests among the servers in your farm, based on server availability and various other criteria.

> **NOTE**
>
> You don't have to use ARR with WFF; you can configure WFF to work with third-party load balancing solutions. For more information, see Overview of the Web Farm Framework 2.0 for IIS 7.

Load balancing using ARR is a complex topic, most of which is beyond the scope of this tutorial. However, you can use the next procedure to install the ARR module and get started with load balancing.

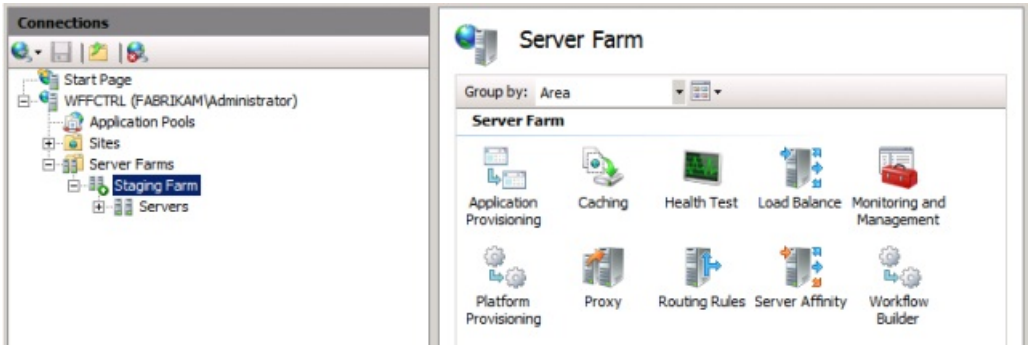
To set up load balancing on the WFF controller server

1. On the WFF controller server, launch the Web Platform Installer.
2. At the top of the **Web Platform Installer 3.0** window, click **Products**.
3. On the left side of the window, in the navigation pane, click **Server**.
4. In the **Application Request Routing 2.5** row, click **Add**.



5. Click **Install**, and then follow the instructions in the **Web Platform Installation** window.

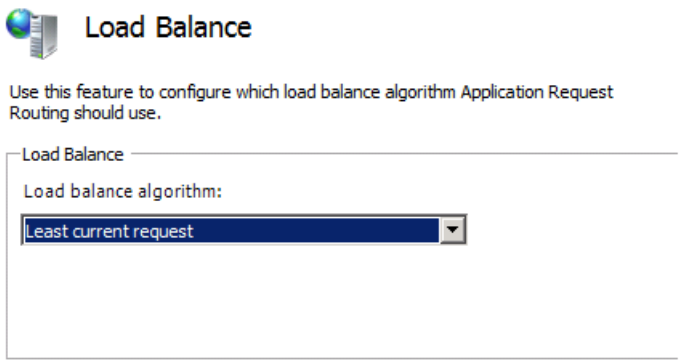
6. When the installation is complete, launch IIS Manager, and in the **Connections** pane, click your server farm node. Notice that several new icons have been added to the **Server Farm** pane.



7. In the **Server Farm** pane, double-click **Load Balance**.

8. In the **Load Balance** pane, select a load balance algorithm (for example, **Least current request**).

> **NOTE**
>
> For more information on load balancing algorithms and other configuration settings, see Application Request Routing Module.



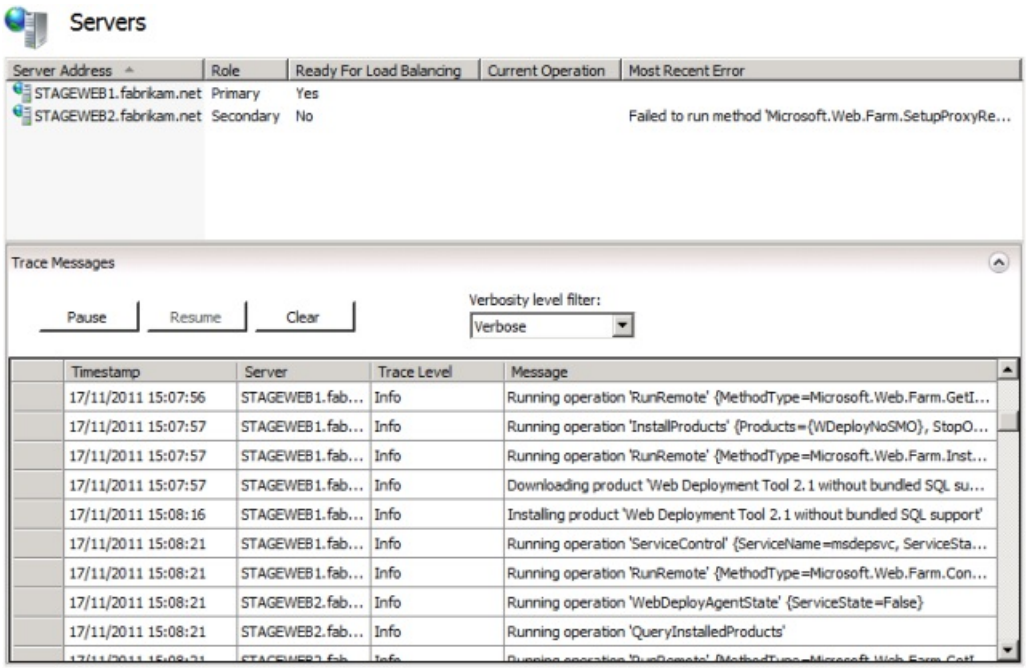
9. In the **Actions** pane, click **Apply**.

You have now configured basic load balancing for the servers in your server farm. If you direct all your web farm traffic to the controller server, the requests will be distributed between the servers in your farm according to availability and the load balancing algorithm you selected.

For more information on how to configure load balancing with ARR, see Application Request Routing Module.

Monitor the Server Farm

You can monitor the health of your server farm at any time through IIS Manager on the controller server. In the **Connections** pane, expand your server farm, and then click **Servers**. The center pane will show a summary of each server in the farm together with a trace log of recent activity.



Conclusion

Your WFF server farm should now be up and running. You can configure the primary server to support whichever deployment approach you prefer—see the Further Reading section for details—and your configuration will be replicated on each secondary server in the server farm.

Further Reading

For more guidance on all aspects of configuring and using the WFF, see the Microsoft Web Farm Framework 2.0 for IIS 7 website.

PREVIOUS NEXT

Configuring Deployment Properties for a Target Environment

2/20/2020 • 9 minutes to read • Edit Online

by Jason Lee

Download PDF

> This topic describes how to configure environment-specific properties in order to deploy the sample Contact Manager solution to a specific target environment.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Build Process, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Process Overview

The project file that you'll use to build and deploy the Contact Manager solution is split into two physical files:

- One that contains universal build settings and instructions (the *Publish.proj* file).
- One that contains environment-specific build settings (*Env-Dev.proj*, *Env-Stage.proj*, and so on).

At build time, the appropriate environment-specific project file is merged into the universal *Publish.proj* file to form a complete set of build instructions. You can configure deployment to specific destination environments by creating or customizing environment-specific project files with settings that describe your own deployment scenario.

Lots of these values are determined by how your destination environment is configured—in particular, whether your target web server is configured to use the Web Deployment Agent Service (the remote agent) or the Web Deploy Handler. For more information on these approaches, and for guidance on choosing the right approach for your own environment, see Choosing the Right Approach to Web Deployment.

The Contact Manager scenario requires two environment-specific project files:

- Deployment to a developer test environment (*Env-Dev.proj*). The developer test environment is configured to accept remote deployments using the remote agent, as described in Scenario: Configuring a Test Environment for Web Deployment. This file needs to provide the remote agent endpoint address as well as location-specific settings like connection strings and service endpoints.
- Deployment to a staging environment (*Env-Stage.proj*). The staging environment is configured to accept remote deployments using the Web Deploy Handler, as described in Scenario: Configuring a Staging Environment for Web Deployment. This file needs to provide the Web Deploy Handler endpoint address as well as location-specific settings like connection strings and service endpoints.

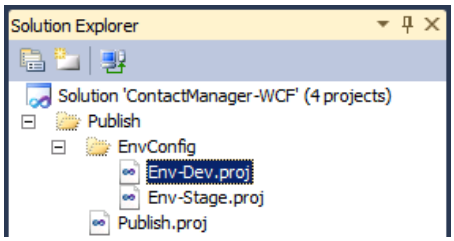
It's important to note that the settings you configure in the environment-specific project file don't affect the contents of the web package itself—instead, they control how the package is deployed and what parameter

values are provided when the package is extracted. You're importing the web package into the production environment manually, as described in Scenario: Configuring a Production Environment for Web Deployment and Manually Installing Web Packages, so it doesn't matter what settings you used in the environment-specific project file when you generated the package. Internet Information Services (IIS) Manager will prompt you for any parameterized values, like connection strings and service endpoints, when you import the package.

To deploy the Contact Manager solution to your own target environment, you can either customize this file or use it as a template and create your own file.

To configure environment-specific deployment settings for the Contact Manager solution

1. Open the ContactManager-WCF solution in Visual Studio 2010.
2. In the **Solution Explorer** window, expand the **Publish** folder, expand the **EnvConfig** folder, and then double-click **Env-Dev.proj**.



3. Replace the property values in the *Env-Dev.proj* file with the correct values for your own test environment.

> **NOTE**
> The table that follows this procedure provides more information on each of these properties.

4. Save your work, and then close the *Env-Dev.proj* file.

Choosing the Right Deployment Properties

This table describes the purpose of each property in the sample environment-specific project file, *Env-Dev.proj*, and provides some guidance on the values you should provide.

PROPERTY NAME	DETAILS
MSDeployComputerName The name of the destination web server or service endpoint.	If you're deploying to the remote agent service on the destination web server, you can specify the target computer name (for example, **TESTWEB1** or **TESTWEB1.fabrikam.net**), or you can specify the remote agent endpoint (for example, `http://TESTWEB1/MSDEPLOYAGENTSERVICE`). The deployment works the same way in each case. If you're deploying to the Web Deploy Handler on the destination web server, you should specify the service endpoint and include the name of the IIS website as a query string parameter (for example, `https://STAGEWEB1:8172/MSDeploy.axd?site=DemoSite`).

PROPERTY NAME	DETAILS
MSDeployAuth The method that Web Deploy should use to authenticate to the remote computer.	This should be set to **NTLM** or **Basic**. Typically, you'll use **NTLM** if you're deploying to the remote agent service and **Basic** if you're deploying to the Web Deploy Handler. If you use basic authentication, you also need to specify the user name and password that the IIS Web Deployment Tool (Web Deploy) should impersonate in order to perform the deployment. In this example, these values are provided through the **MSDeployUsername** and **MSDeployPassword** properties. If you use NTLM authentication, you can omit these properties or leave them blank.
MSDeployUsername If you use basic authentication, Web Deploy will use this account on the remote computer.	This should take the form *DOMAIN**username* (for example, **FABRIKAM\matt**). This value is only used if you specify basic authentication. If you use NTLM authentication, the property can be omitted. If a value is supplied, it will be ignored.
MSDeployPassword If you use basic authentication, Web Deploy will use this password on the remote computer.	This is the password for the user account you specified in the **MSDeployUsername** property. This value is only used if you specify basic authentication. If you use NTLM authentication, the property can be omitted. If a value is supplied, it will be ignored.
ContactManagerIisPath The IIS path on which you want to deploy the Contact Manager MVC application.	This should be the path as it appears in IIS Manager, in the form [*IIS website name*]/[*webapplication name*]. Remember that the IIS website needs to exist before you deploy your application. For example, if you've created an IIS website named DemoSite, you could specify the IIS path for the MVC application as DemoSite/ContactManager.
ContactManagerServiceIisPath The IIS path on which you want to deploy the Contact Manager WCF service.	For example, if you've created an IIS website named DemoSite, you could specify the IIS path for the WCF service as **DemoSite/ContactManagerService**.
ContactManagerTargetUrl The URL at which the WCF service can be reached.	This will take the form [*IIS website root URL*]/[*service application name*]/[*service endpoint*]. For example, if you've created an IIS website on port 85, the URL would take the form `http://localhost:85/ContactManagerService/ContactService.svc`. Remember that the MVC application and the WCF service are deployed to the same server. As a result, this URL is only ever accessed from the machine on which it's installed. Because of this, it's better to use localhost or the IP address, rather than the machine name or a host header, in the URL. If you use the machine name or a host header, the loopback check security feature in IIS may block the URL and return an **HTTP 401.1 - Unauthorized** error.

PROPERTY NAME	DETAILS
CmDatabaseConnectionString The connection string for the database server.	The connection string determines both the credentials that VSDBCMD will use to contact the database server and create the database and the credentials that the web server application pool will use to contact the database server and interact with the database. Essentially you have two choices here. You can specify **Integrated Security=true**, in which case integrated Windows authentication is used: **Data Source=TESTDB1;Integrated Security=true** In this case, the database will be created using the credentials of the user who runs the VSDBCMD executable, and the application will access the database using the identity of the web server machine account. Alternatively, you can specify the user name and password of a SQL Server account. In this case, the SQL Server credentials are used both by VSDBCMD to create the database and by the application pool to interact with the database: **Data Source=TESTDB1;User Id=ASqlUser; Password=Pa$$w0rd** The walkthroughs in this topic assume that you'll use integrated Windows authentication.
CmTargetDatabase The name you want to give the database you'll create on the database server.	The value you provide here is added to the VSDBCMD command as a parameter. It's also used to build a full connection string that the application pool on the web server can use to interact with the database.

These examples show how you might configure these properties for specific deployment scenarios.

Example 1—Deployment to the Remote Agent Service

In this example:

- You're deploying to the remote agent service on TESTWEB1.
- You're instructing Web Deploy to use NTLM authentication. Web Deploy will run using the credentials you used to invoke the Microsoft Build Engine (MSBuild).
- You're using integrated authentication to deploy the **ContactManager** database to TESTDB1. The database will be deployed using the credentials you used to invoke MSBuild.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <MSDeployComputerName Condition=" '$(MSDeployComputerName)'==='' ">
      TESTWEB1.fabrikam.net
    </MSDeployComputerName>
    <MSDeployAuth Condition=" '$(MSDeployAuth)'==='' ">NTLM</MSDeployAuth>
    <ContactManagerTargetUrl Condition =" '$(ContactManagerTargetUrl)'==='' ">
      http://localhost:85/ContactManagerService/ContactService.svc
    </ContactManagerTargetUrl>
    <ContactManagerIisPath Condition=" '$(ContactManagerIisPath)'==='' ">
      DemoSite/ContactManager
    </ContactManagerIisPath>
    <ContactManagerServiceIisPath
      Condition=" '$(ContactManagerServiceIisPath)'==='' ">
        DemoSite/ContactManagerService
    </ContactManagerServiceIisPath>
    <CmDatabaseConnectionString Condition=" '$(CmDatabaseConnectionString)'==='' ">
      Data Source=TESTDB1;Integrated Security=true</CmDatabaseConnectionString>
    <CmTargetDatabase Condition=" '$(CmTargetDatabase)'==='' ">
      ContactManager
    </CmTargetDatabase>
  </PropertyGroup>
</Project>
```

Example 2—Deployment to the Web Deploy Handler Endpoint

In this example:

- You're deploying to the Web Deploy Handler service endpoint on STAGEWEB1.
- You're instructing Web Deploy to use basic authentication.
- You're specifying that Web Deploy should impersonate the FABRIKAM\stagingdeployer account on the remote computer.
- You're using SQL Server authentication to deploy the **ContactManager** database to STAGEDB1.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <MSDeployComputerName Condition=" '$(MSDeployComputerName)'==''  ">
      https://STAGEWEB1:8172/MSDeploy.axd?site=DemoSite
    </MSDeployComputerName>
    <MSDeployAuth Condition=" '$(MSDeployAuth)'==''  ">Basic</MSDeployAuth>
    <MSDeployUsername Condition=" '$(MSDeployUsername)'==''  ">
      FABRIKAM\stagingdeployer
    </MSDeployUsername>
    <MSDeployPassword Condition=" '$(MSDeployPassword)'==''  ">
      Pa$$w0rd
    </MSDeployPassword>
    <ContactManagerTargetUrl Condition =" '$(ContactManagerTargetUrl)'==''  ">
      http://localhost:85/ContactManagerService/ContactService.svc
    </ContactManagerTargetUrl>
    <ContactManagerIisPath Condition=" '$(ContactManagerIisPath)'==''  ">
      DemoSite/ContactManager
    </ContactManagerIisPath>
    <ContactManagerServiceIisPath
      Condition=" '$(ContactManagerServiceIisPath)'==''  ">
        DemoSite/ContactManagerService
    </ContactManagerServiceIisPath>
    <CmDatabaseConnectionString Condition=" '$(CmDatabaseConnectionString)'==''  ">
      Data Source=STAGEDB1;User ID=sa;Password=Pa$$w0rd;
    </CmDatabaseConnectionString>
    <CmTargetDatabase Condition=" '$(CmTargetDatabase)'==''  ">
      ContactManager
    </CmTargetDatabase>
  </PropertyGroup>
</Project>
```

Conclusion

At this point, your project files are fully configured to build and deploy the Contact Manager solution to one or more destination environments.

To use these project files as part of a single-step, repeatable deployment process, you need to execute the *Publish.proj* file using MSBuild and pass in the location of the environment-specific project file as a parameter. You can do this in various ways:

- For an overview of MSBuild and an introduction to custom project files, see Understanding the Project File.
- For information on how to formulate an MSBuild command that executes your custom project files, see Deploying Web Packages.
- For information on how to incorporate your MSBuild commands into a command file for single-step, repeatable deployments, see Creating and Running a Deployment Command File.
- For information on how to execute your custom project files from Team Build, see Creating a Build Definition that Supports Deployment.

ASP.NET 4 - Enterprise Deployment Series 3 Configuring TFS

2/20/2020 • 2 minutes to read • Edit Online

This tutorial will show you how to configure Team Foundation Server (TFS) 2010 to build solutions and deploy web content to various target environments.

- Configuring Team Foundation Server for Web Deployment
- Creating a Team Project in TFS
- Adding Content to Source Control
- Configuring a TFS Build Server for Web Deployment
- Creating a Build Definition That Supports Deployment
- Deploying a Specific Build
- Configuring Permissions for Team Build Deployment

Configuring Team Foundation Server for Web Deployment

2/20/2020 • 5 minutes to read • Edit Online

by Jason Lee

Download PDF

This tutorial will show you how to configure Team Foundation Server (TFS) 2010 to build solutions and deploy web content to various target environments. This includes continuous integration (CI) scenarios, where you deploy content automatically every time a developer makes a change. It can also include manual trigger scenarios, where an administrator may want to trigger deployment of a specific build to a staging environment once the build has been verified and validated in the test environment. The topics in this tutorial will guide you through the entire configuration process, including:

- How to create a new team project in TFS.
- How to add content to source control.
- How to configure a build server to support CI and deployment.
- How to create a build definition that includes deployment logic.
- How to configure permissions for automated deployment.

For an Italian translation of these tutorials, visit http://www.lucamorelli.it.

This tutorial assumes that you have installed TFS 2010 and created a team project collection as part of the initial configuration process. The Team Foundation Installation Guide for Visual Studio 2010 provides comprehensive guidance on these tasks.

Context

This forms part of a series of tutorials based on the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Build Process, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Scenario Overview

The high-level scenario for these tutorials is described in Enterprise Web Deployment: Scenario Overview. We recommend that you review this topic before you get started on this tutorial.

How to Use This Tutorial

If this is the first time you've performed the tasks described in this tutorial, or if you want to follow the examples using the sample solution, you should work through the tutorial topics in order. Alternatively, you can use

individual topics as guidance for specific tasks. This tutorial includes these topics:

- Creating a Team Project in TFS. A team project is the core unit for source control, process management, and build in TFS. You need to create a team project before you can add content to source control or create build definitions.
- Adding Content to Source Control. Once you've created a team project, you can start adding content to source control. You'll need to add your projects and solutions, together with any external dependencies, before you can start configuring builds.
- Configuring a TFS Build Server for Web Deployment. If you want to build your team project content, you'll need to configure a build server. In most cases, this should be on a separate machine from your TFS installation. To configure a build server, you need to install and configure the TFS build service, install Visual Studio 2010, create build controllers and build agents, install any products or components that your code needs in order to build successfully, and install the Internet Information Services (IIS) Web Deployment Tool (Web Deploy).
- Creating a Build Definition That Supports Deployment. Before you can start queuing or triggering builds in TFS, you need to create at least one build definition for your team project. The build definition defines every aspect of the build, including which things should be included in the build, what should trigger the build, and where Team Build should send the build outputs. You can configure a build definition to run custom Microsoft Build Engine (MSBuild) project files, which lets you include deployment logic in your automated builds.
- Deploying a Specific Build. In a lot of scenarios, you'll want to deploy a specific build, rather than the latest build, to a target environment. In this case, you can configure a build definition that deploys content from a specific drop folder.
- Configuring Permissions for Team Build Deployment. If the build service is to deploy content as part of an automated build process, you need to grant various permissions to the build service account on any destination web servers and database servers.

Key Technologies

This tutorial focuses on how to use these products and technologies to support automated build and web deployment:

- Visual Studio Team Foundation Server 2010
- Team Build and MSBuild
- Web Deploy

The tutorial also touches on the use of Windows Server 2008 R2, IIS 7.5, SQL Server 2008 R2, ASP.NET 4.0, and ASP.NET MVC 3.

Other Tutorials in This Series

This forms part of a series of five tutorials on enterprise-scale web deployment. These are the other tutorials in the series:

- Deploying Web Applications in Enterprise Scenarios. This introductory content provides the contextual background for the tutorial series. It describes the tutorial scenario, and it illustrates how the tasks and walkthroughs described throughout the series fit into a broader Application Lifecycle Management (ALM) process.
- Web Deployment in the Enterprise. This tutorial provides a conceptual introduction to MSBuild project files, the Web Publishing Pipeline (WPP), Web Deploy, and other related technologies. It explains how you can use these tools together to manage complex deployment processes.
- Configuring Server Environments for Web Deployment. This tutorial describes how to configure Windows servers to support various deployment scenarios, including remote web package deployment using the Web

Deployment Agent Service (the remote agent) or the Web Deploy Handler and remote database deployment. It provides guidance on choosing the appropriate deployment method for your own environment, and it describes how to use the Web Farm Framework (WFF) to replicate deployed web applications across all the web servers in a server farm.

- Advanced Enterprise Web Deployment. This tutorial describes how to accomplish various more advanced deployment tasks, like customizing database deployments for multiple environments, excluding files and folders from deployment, and taking web applications offline during the deployment process.

NEXT

Creating a Team Project in TFS

2/20/2020 • 8 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic describes how to create a new team project in Team Foundation Server (TFS) 2010.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

Task Overview

To provision and use a new team project in TFS, you'll need to complete these high-level steps:

- Grant permissions to the user who will create the new team project.
- Create the team project.
- Grant permissions to the team members who will work on the project.
- Check in some content.

This topic will show you how to perform these procedures, and it will identify the users and job roles that are likely to be responsible for each procedure. Be aware that, depending on the structure of your organization, each of these tasks may be the responsibility of a different person.

The tasks and walkthroughs in this topic assume that you've installed and configured TFS, and that you've created a team project collection as part of the configuration process. For more information on these assumptions, and for more general background information on the scenario, see Configure a TFS Build Server for Web Deployment.

Grant Permissions to the Team Project Creator

In order to create a new team project, you need these permissions:

- You must have the **Create new projects** permission on the TFS application tier. You typically grant this permission by adding users to the **Project Collection Administrators** TFS group. The **Team Foundation Administrators** global group also includes this permission.
- You must have permission to create new team sites within the SharePoint site collection that corresponds to the TFS team project collection. You typically grant this permission by adding the user to a SharePoint group with **Full Control** rights on the SharePoint site collection.
- If you're using SQL Server Reporting Services features, you must be a member of the **Team Foundation Content Manager** role in Reporting Services.

Who Performs These Procedures?

Typically, the person or group who administers the TFS deployment also performs these procedures.

Because this is a highly privileged set of permissions, new team projects are typically created by a small subset of users with responsibility for administering a TFS deployment. Developers will not usually be granted the permissions required to create new team projects.

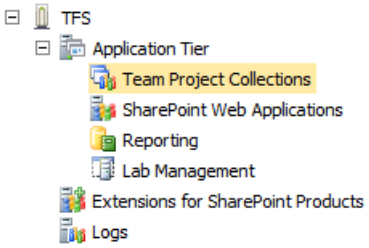
Grant Permissions in TFS

If you want to enable a user to create new team projects, the first high-level task is to add the user to the **Project Collection Administrators** group for the team project collection.

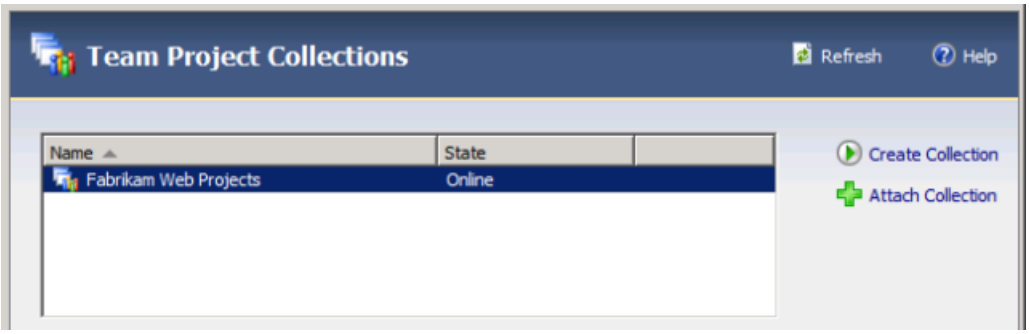
To add a user to the Project Collection Administrators group

1. On the TFS server, on the **Start** menu, point to **All Programs**, click **Microsoft Team Foundation Server 2010**, and then click **Team Foundation Administration Console**.

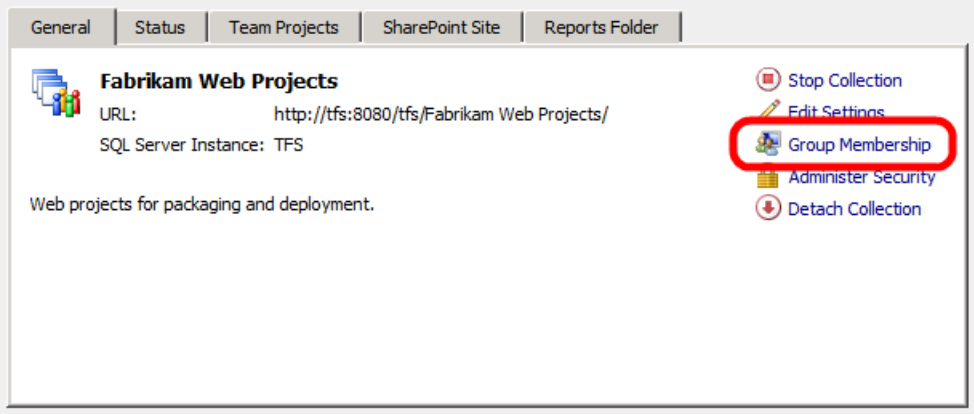
2. In the navigation tree view, expand **Application Tier**, and then click **Team Project Collections**.



3. In the **Team Project Collections** pane, select the team project collection you want to manage.

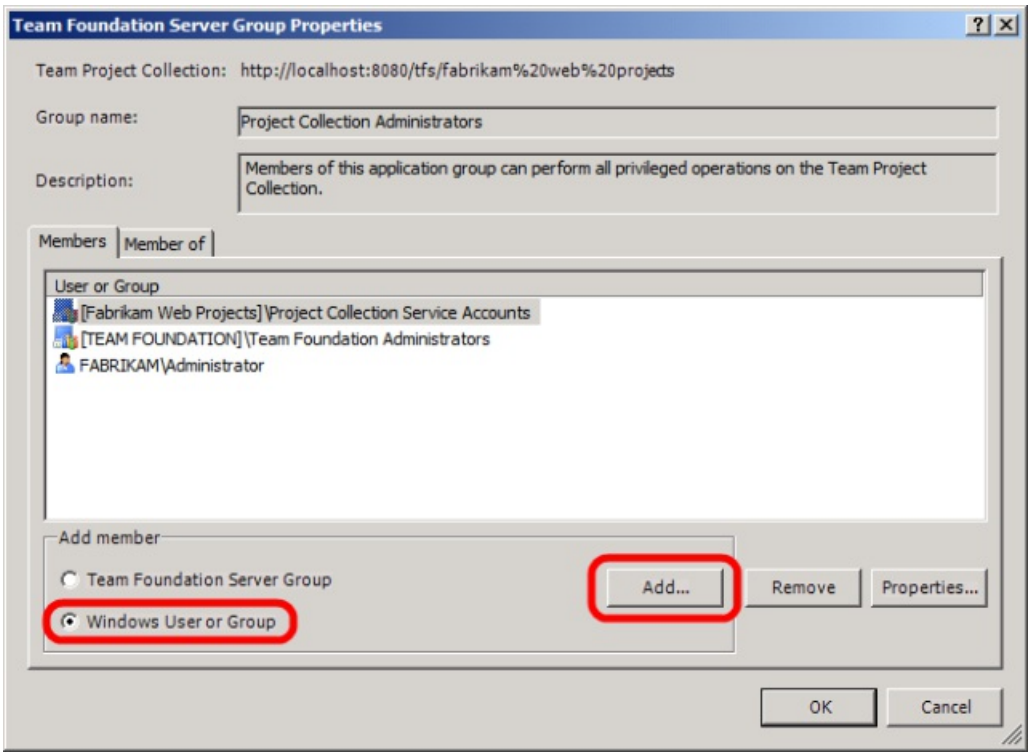


4. On the **General** tab, click **Group Membership**.

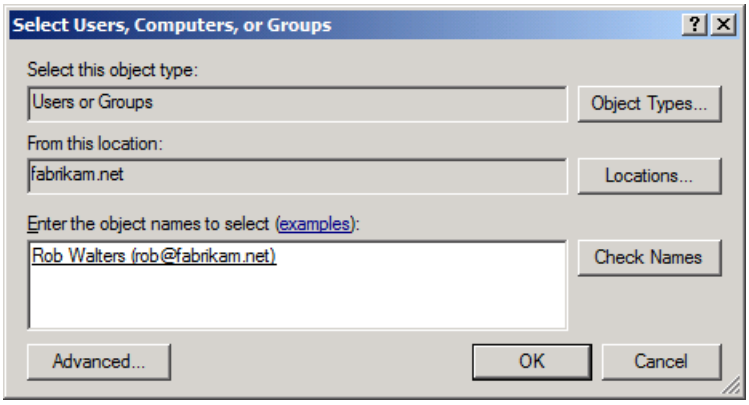


5. In the **Global Groups** dialog box, select the **Project Collection Administrators** group, and then click **Properties**.

6. In the **Team Foundation Server Group Properties** dialog box, select **Windows User or Group**, and then click **Add**.



7. In the **Select Users, Computers, or Groups** dialog box, type the user name of the user you want to be able to create new team projects, click **Check Names**, and then click **OK**.



8. In the **Team Foundation Server Group Properties** dialog box, click **OK**.

9. In the **Global Groups** dialog box, click **Close**.

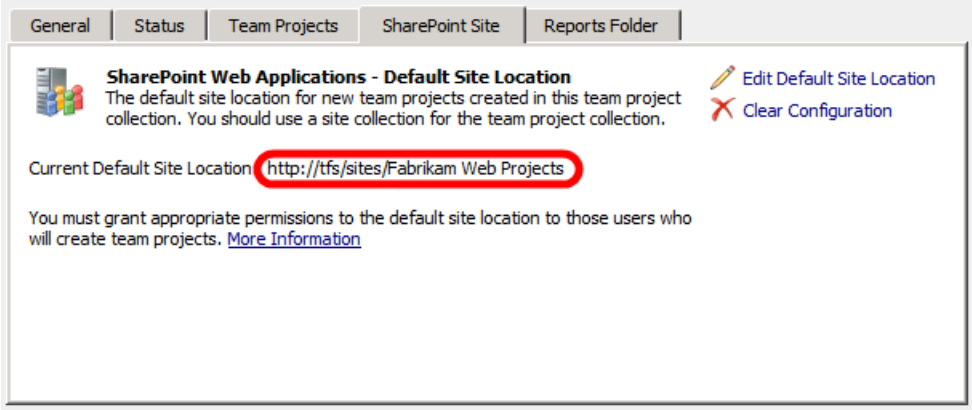
Grant Permissions in SharePoint Services

Next, you need to give the user permission to create new team sites in the SharePoint site collection that corresponds to your TFS team project collection.

To grant Full Control permissions on the SharePoint site collection

1. In the Team Foundation Server Administration Console, on the **Team Project Collections** page, select the team project collection you want to manage.

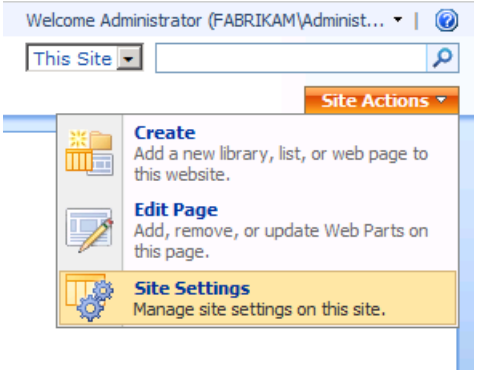
2. On the **SharePoint Site** tab, note the value of the **Current Default Site Location** URL.



3. Open Internet Explorer, and then go to the URL you noted in step 2.

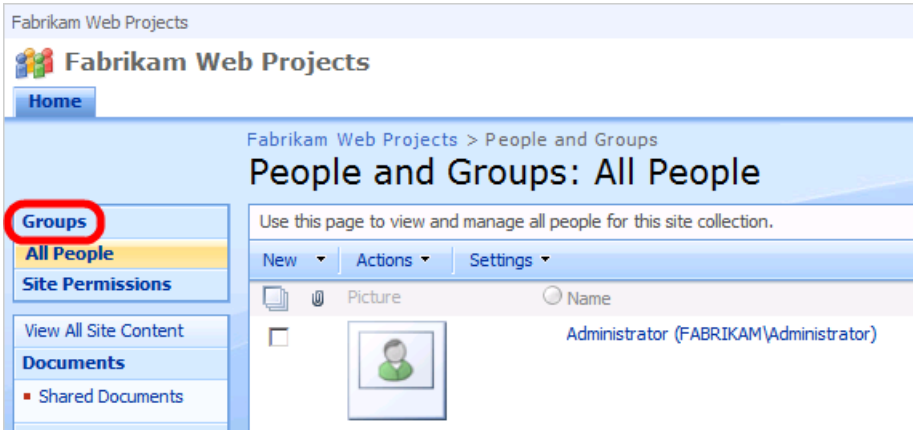
> **NOTE**
>
> If you're not logged on to Windows as the user who created the team project collection, you'll need to sign in to SharePoint as this user in order to continue.

4. On the **Site Actions** menu, click **Site Settings**.

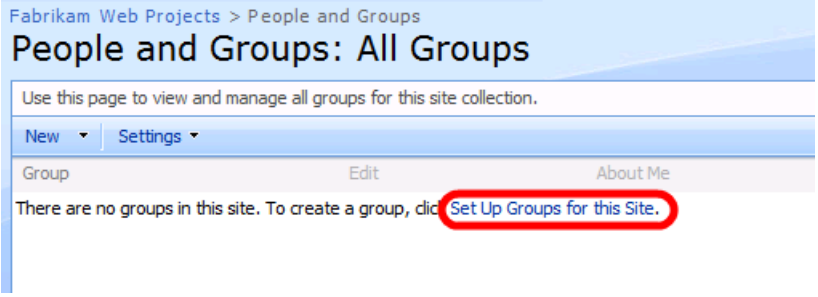


5. On the **Site Settings** page, under **Users and Permissions**, click **People and groups**.

6. In the left navigation panel, click **Groups**.

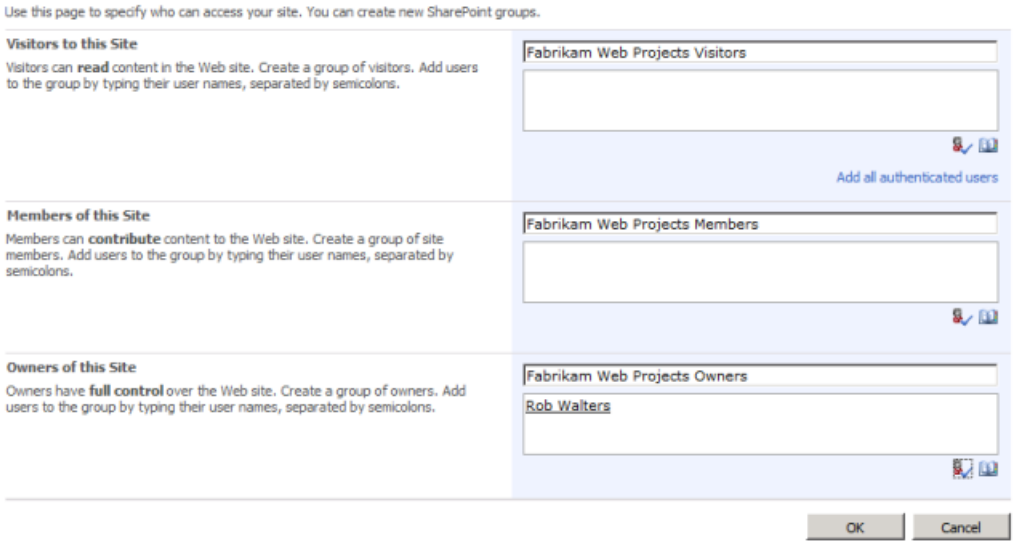


7. On the **People and Groups: All Groups** page, click **Set Up Groups for this Site**.



> **NOTE**
>
> You may receive an **HTTP 404 Not Found** error due to a double HTTP encoding bug. If this occurs, replace the URL with this:
>
> `[site_collection_URL]/_layouts/permsetup.aspx` For example:
>
> `http://tfs/sites/Fabrikam%20Web%20Projects/_layouts/permsetup.aspx`

8. On the **Set Up Groups for this Site** page, add the user who will create team projects to the **Owners** group, and then click **OK**.



For more information on enabling users to create new team projects within a team project collection, see Set Administrator Permissions for Team Project Collections.

Create a New Team Project and Add Users

Once you have the necessary permissions, you can use the **Team Explorer** window in Visual Studio 2010 to create a new team project. This approach provides a wizard that collects all the required information and performs the necessary tasks in TFS, SharePoint, and SQL Server Reporting Services. You'll also need to grant permissions on the new team project to members of the developer team, to enable them to add and modify content.

Who Performs These Procedures?

Usually either a TFS administrator or a developer team leader performs these procedures.

Create a New Team Project

The next procedure describes how to create a new team project in TFS 2010.

To create a new team project

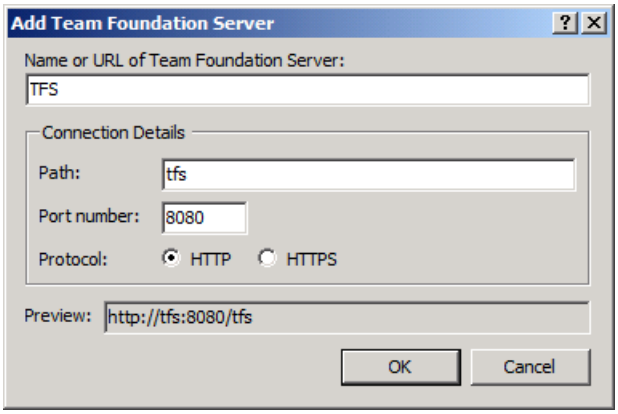
1. On the **Start** menu, point to **All Programs**, click **Microsoft Visual Studio 2010**, right-click **Microsoft Visual Studio 2010**, and then click **Run as administrator**.

> **NOTE**
>
> If you don't run Visual Studio 2010 as an administrator, the New Team Project Wizard will fail on the last step.

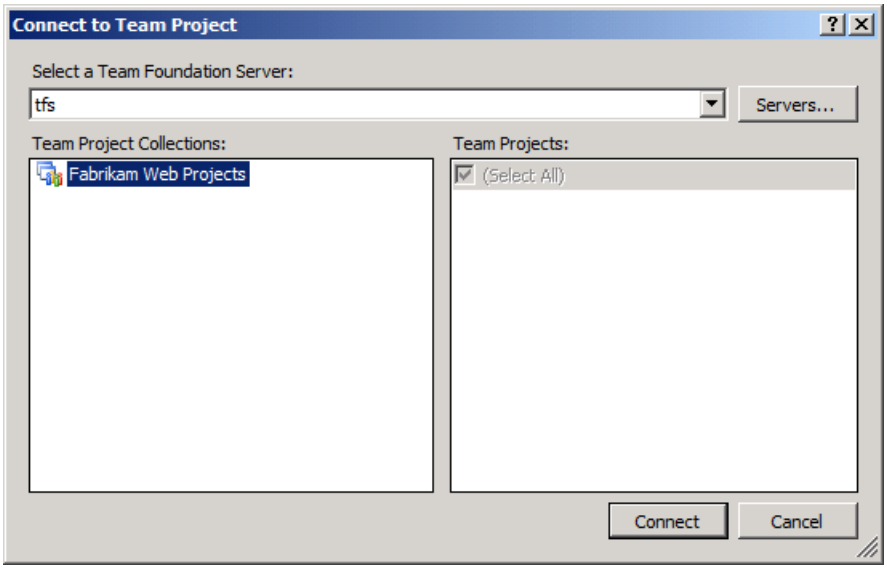
2. If the **User Account Control** dialog box appears, click **Yes**.
3. In Visual Studio, on the **Team** menu, click **Connect to Team Foundation Server**.

> **NOTE**
>
> If you have already configured a connection to a TFS server, you can omit steps 4-7.

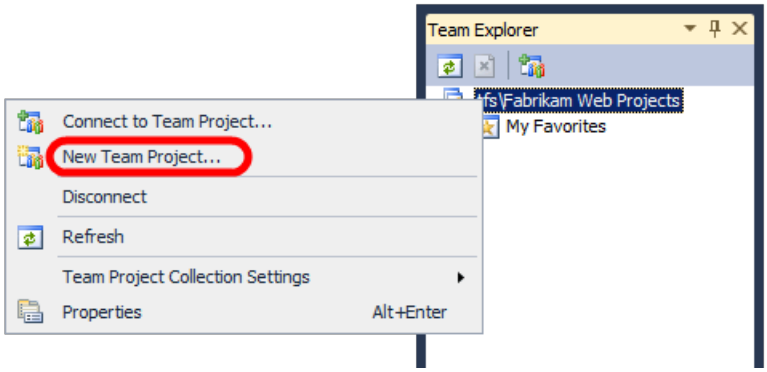
4. In the **Connection to Team Project** dialog box, click **Servers**.
5. In the **Add/Remove Team Foundation Server** dialog box, click **Add**.
6. In the **Add Team Foundation Server** dialog box, provide the details of your TFS instance, and then click **OK**.



7. In the **Add/Remove Team Foundation Server** dialog box, click **Close**.
8. In the **Connect to Team Project** dialog box, select the TFS instance you want to connect to, select the team project collection you want to add to, and then click **Connect**.

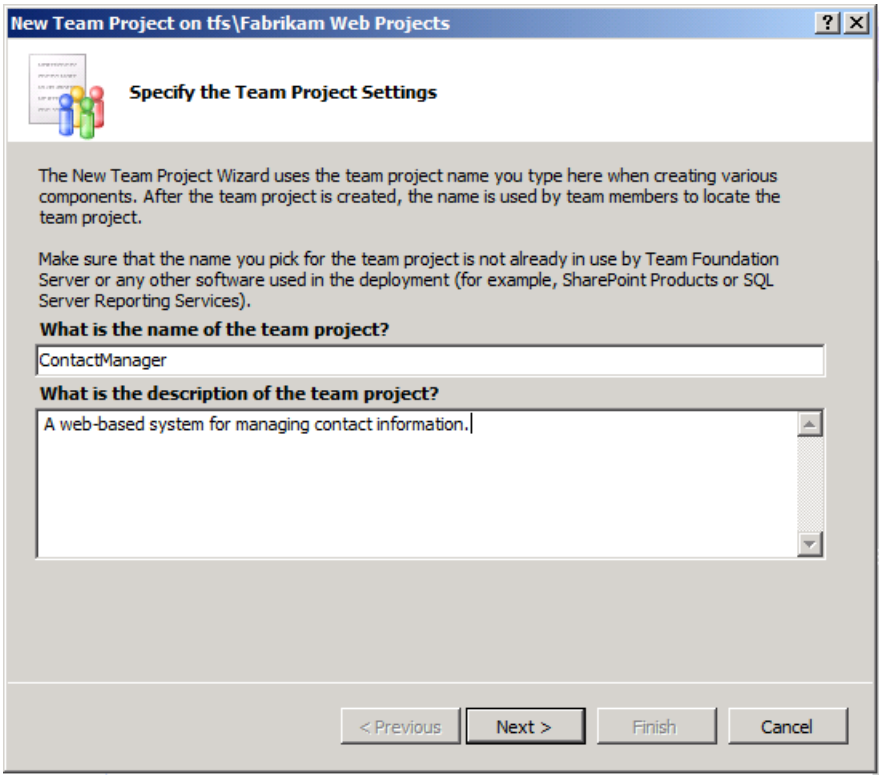


9. In the **Team Explorer** window, right-click the team project collection, and then click **New Team Project**.



10. In the **New Team Project** dialog box, provide a name and a description for the team project, and then click **Next**.

> **NOTE**
>
> If your team project includes spaces, you may face some issues when you come to use the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) to deploy packages from the output path. Spaces in the path can make it a lot more difficult to run Web Deploy commands.



11. On the **Select a Process Template** page, select the process template that you want to use to manage the development process, and then click **Next**.

> **NOTE**
>
> For more information on process templates for TFS, see Process Templates and Tools.

12. On the **Team Site Settings** page, leave the default settings unchanged, and then click **Next**.

13. This setting creates, or identifies, a SharePoint team site that is associated with the TFS team project. Your development team can use this site to manage documentation, participate in discussion threads, create wiki pages, and perform various other tasks that are not related to code. For more information, see Interactions Between SharePoint Products and Team Foundation Server.

14. On the **Specify Source Control Settings** page, leave the default settings unchanged, and then click **Next**.

15. This setting identifies or creates the location in the TFS folder hierarchy that will act as a root folder for your content.

16. On the **Confirm Team Project Settings** page, click **Finish**.

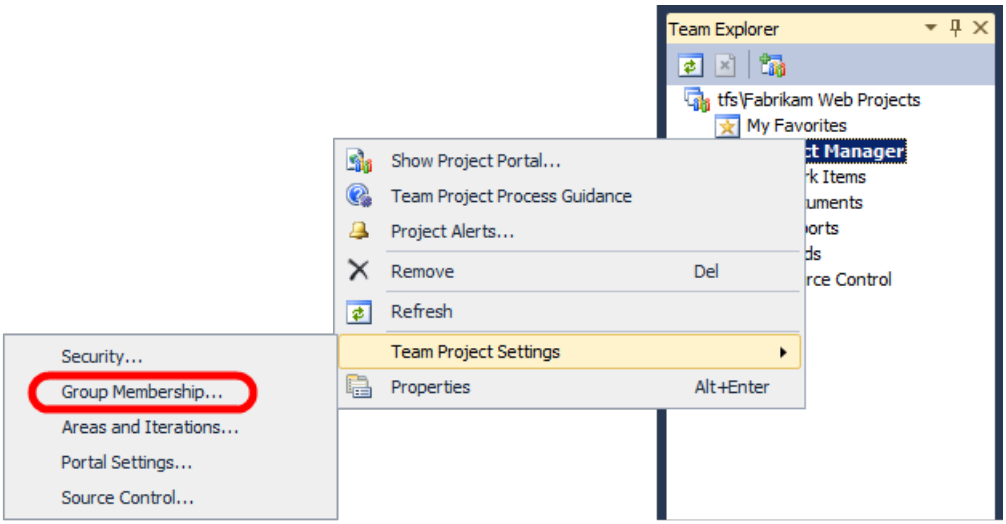
17. When the new team project is successfully created, on the **Team Project Created** page, click **Close**.

Add Users to a Team Project

Now that you've created the new team project, you can grant permissions to users to enable them to start adding and collaborating on content.

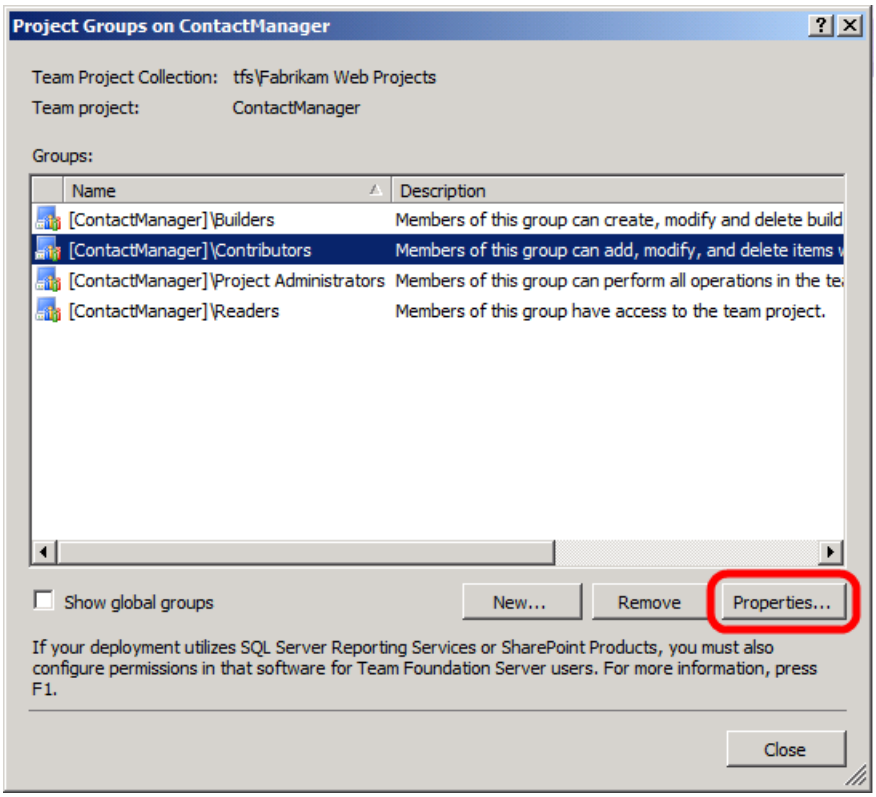
To add users to a team project

1. In Visual Studio 2010, in the **Team Explorer** window, right-click the team project, point to **Team Project Settings**, and then click **Group Membership**.

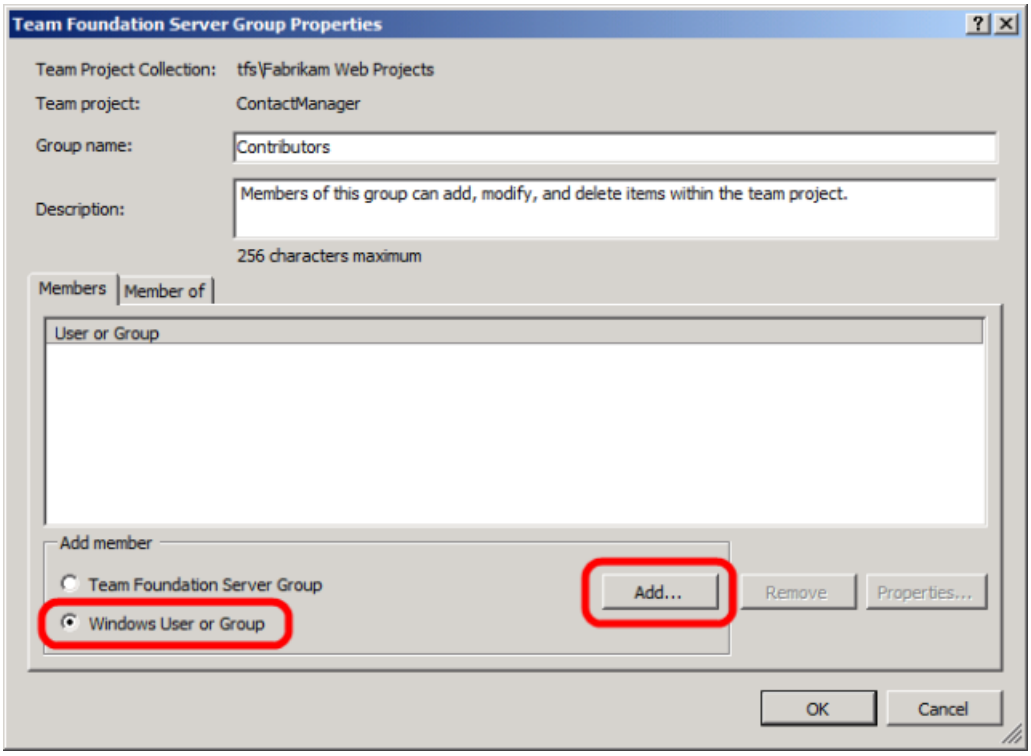


2. To enable a user to add, modify, and remove code under source control, you need to add him or her to the **Contributors** group.

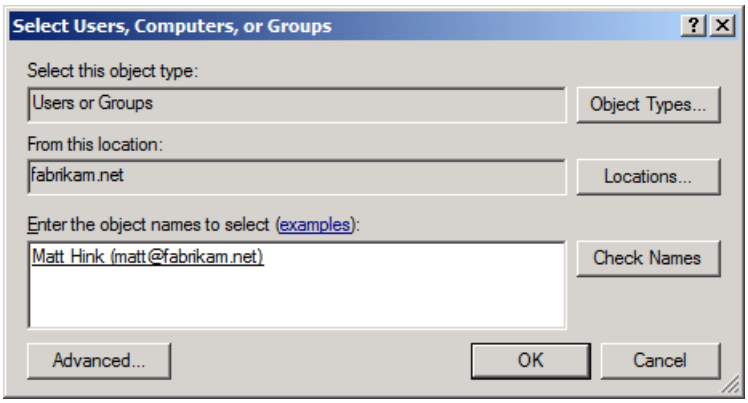
3. In the **Project Groups** dialog box, select the **Contributors** group, and then click **Properties**.



4. In the **Team Foundation Server Group Properties** dialog box, select **Windows User or Group**, and then click **Add**.



5. In the **Select Users, Computers, or Groups** dialog box, type the user name of the user you want to add to the team project, click **Check Names**, and then click **OK**.



6. In the **Team Foundation Server Group Properties** dialog box, click **OK**.

7. In the **Project Groups** dialog box, click **Close**.

Conclusion

At this point, your new team project is ready to use, and your developer team can start adding content and collaborating on the development process.

The next topic, Adding Content to Source Control, describes how to add content to source control.

Further Reading

For broader guidance on creating team projects in TFS, see Create a Team Project. For more information on enabling users to create new team projects within a team project collection, see Set Administrator Permissions for Team Project Collections. For more information on adding users to team projects, see Add Users to Team Projects.

PREVIOUS NEXT

Adding Content to Source Control

2/20/2020 • 6 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic explains how to add content to source control in Team Foundation Server (TFS) 2010. It describes how to add solutions and projects to a team project in TFS, and it explains how to add external dependencies like frameworks or assemblies to source control.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

Task Overview

In most cases, every member of the developer team should be able to add content to source control. To add a solution to source control in TFS, you'll need to complete these high-level steps:

- Connect to a team project.
- Map the team project folder structure on the server to a folder structure on your local computer.
- Add the solution and its contents to source control.
- Add any external dependencies to source control.

This topic will show you how to perform these procedures.

The tasks and walkthroughs in this topic assume that you've already created a new TFS team project to manage your content. For more information on creating a new team project, see Creating a Team Project in TFS.

Who Performs These Procedures?

In most cases, every member of the developer team should be able to add and modify content within specific team projects.

Connect to a Team Project and Create a Folder Mapping

Before you add any content to source control, you need to connect to a team project and create a mapping between the folder structure on the server and the file system on your local machine.

To connect to a team project and map a local path

1. On your developer workstation, open Visual Studio 2010.

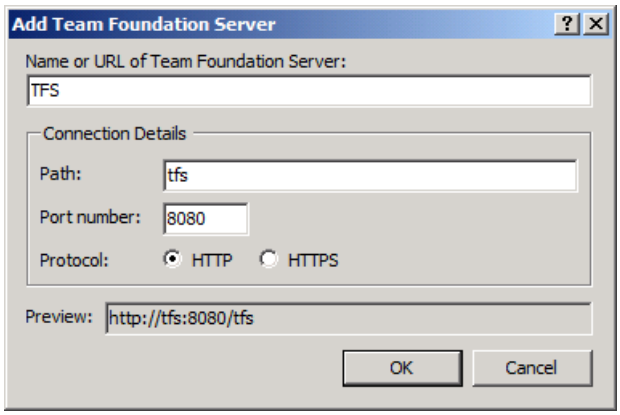
2. In Visual Studio, on the **Team** menu, click **Connect to Team Foundation Server**.

> **NOTE**
>
> If you have already configured a connection to a TFS server, you can omit steps 3-6.

3. In the **Connection to Team Project** dialog box, click **Servers**.

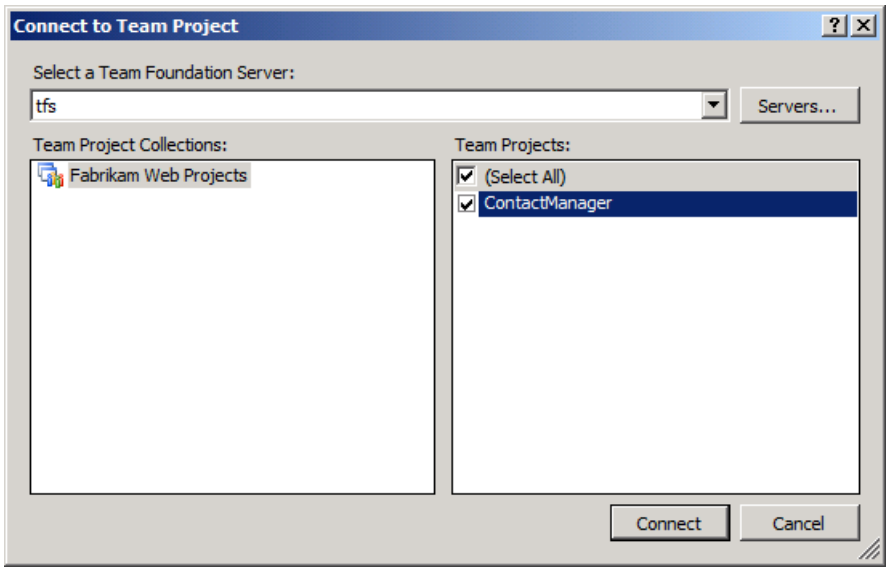
4. In the **Add/Remove Team Foundation Server** dialog box, click **Add**.

5. In the **Add Team Foundation Server** dialog box, provide the details of your TFS instance, and then click **OK**.

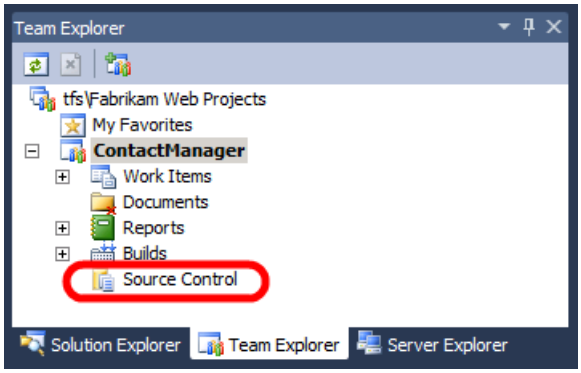


6. In the **Add/Remove Team Foundation Server** dialog box, click **Close**.

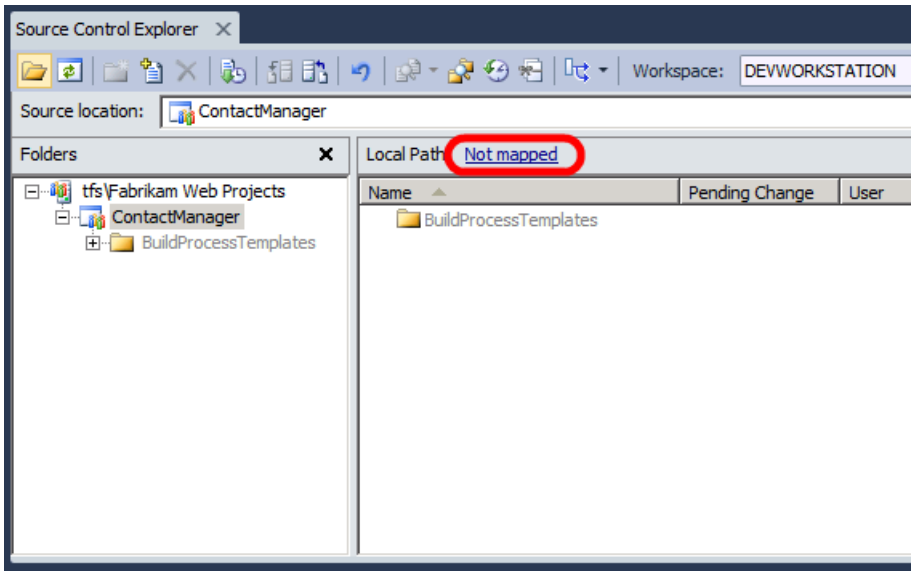
7. In the **Connect to Team Project** dialog box, select the TFS instance you want to connect to, select the team project collection, select the team project you want to add to, and then click **Connect**.



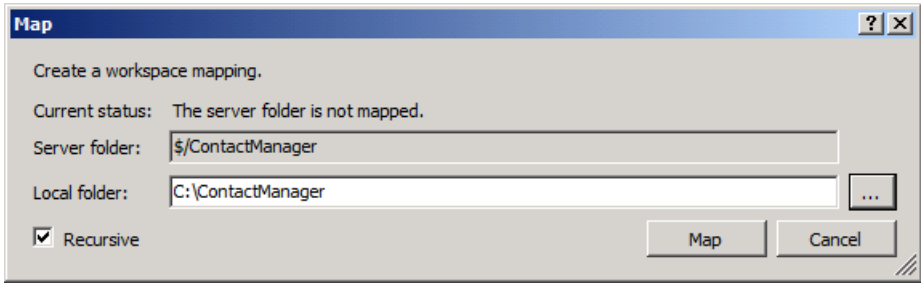
8. In the **Team Explorer** window, expand your team project, and then double-click **Source Control**.



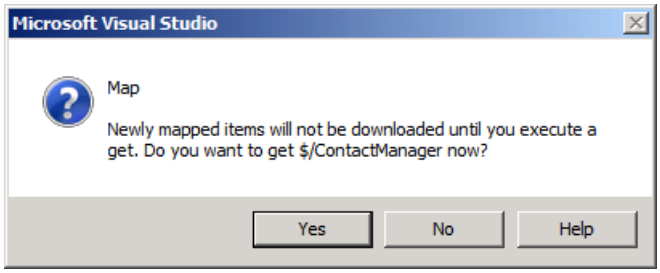
9. On the **Source Control Explorer** tab, click **Not mapped**.



10. In the **Map** dialog box, in the **Local folder** box, browse to (or create) a local folder to act as the root folder for the team project, and then click **Map**.



11. When you're prompted to download source files, click **Yes**.



At this point, you have mapped the server-side folder for the team project to a local folder on your developer workstation. You've also downloaded any existing content from the team project to your local folder structure. You can now start to add your own content to source control.

Add Projects and Solutions to Source Control

To add projects and solutions to source control, you first need to move them to the mapped folder for the team project on your local machine. You can then check in the content to synchronize your additions with the server.

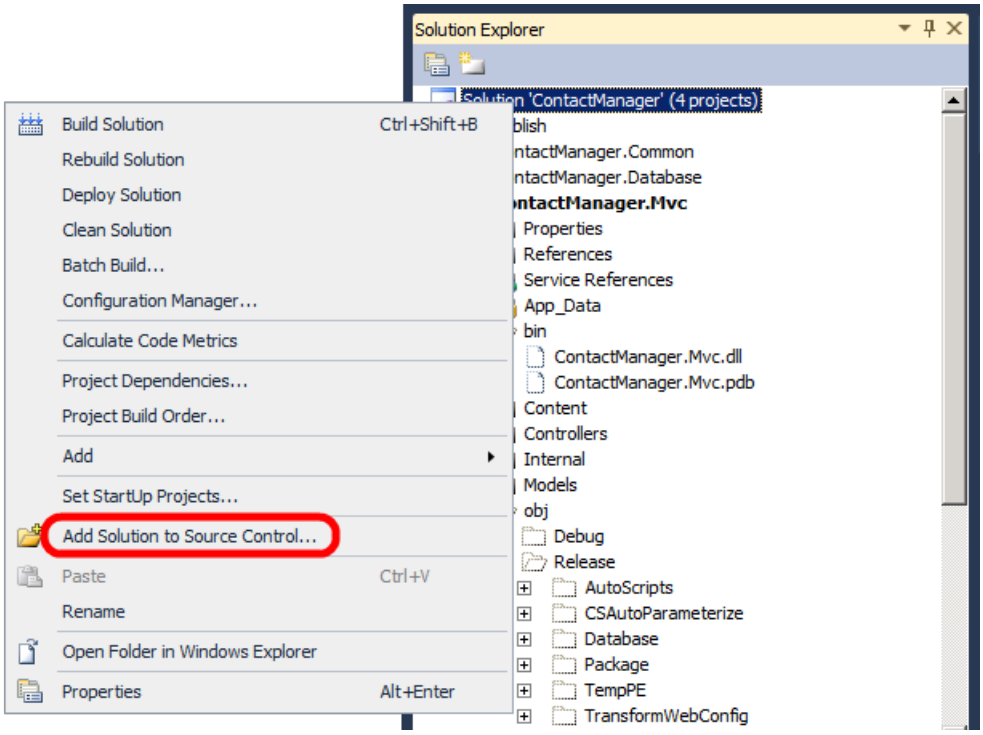
To add projects to source control

1. On your developer workstation, move your projects and solutions to an appropriate location within the mapped folder structure for the team project.

> **NOTE**
>
> Many organizations will have a preferred approach to how projects and solutions should be organized in source control. For guidance on how to structure folders, see How To: Structure Your Source Control Folders in Team Foundation Server.

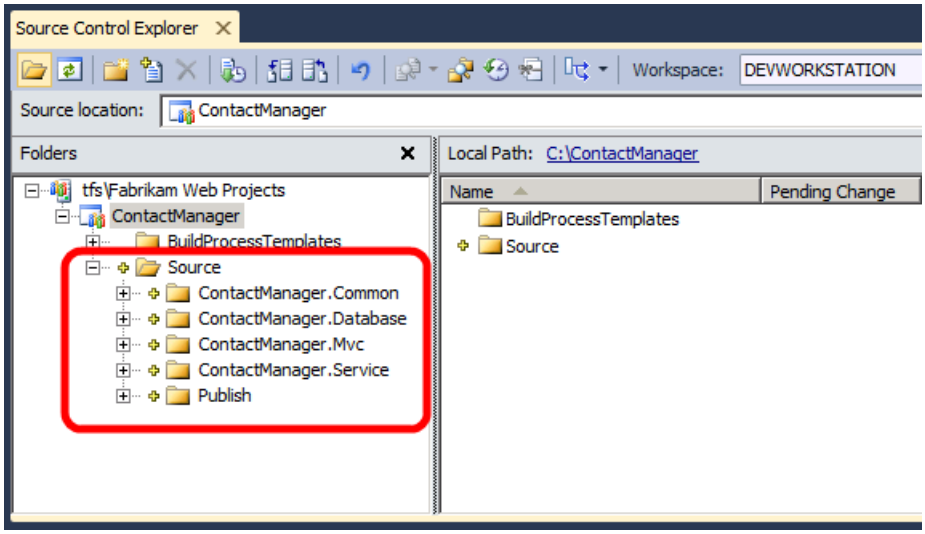
2. Open the solution in Visual Studio 2010.

3. In the **Solution Explorer** window, right-click the solution, and then click **Add Solution to Source Control**.



> **NOTE**
>
> In some cases, depending on how your organization likes to structure content in TFS, you may need to add projects to source control individually to provide more fine-grained control over how your source code is organized.

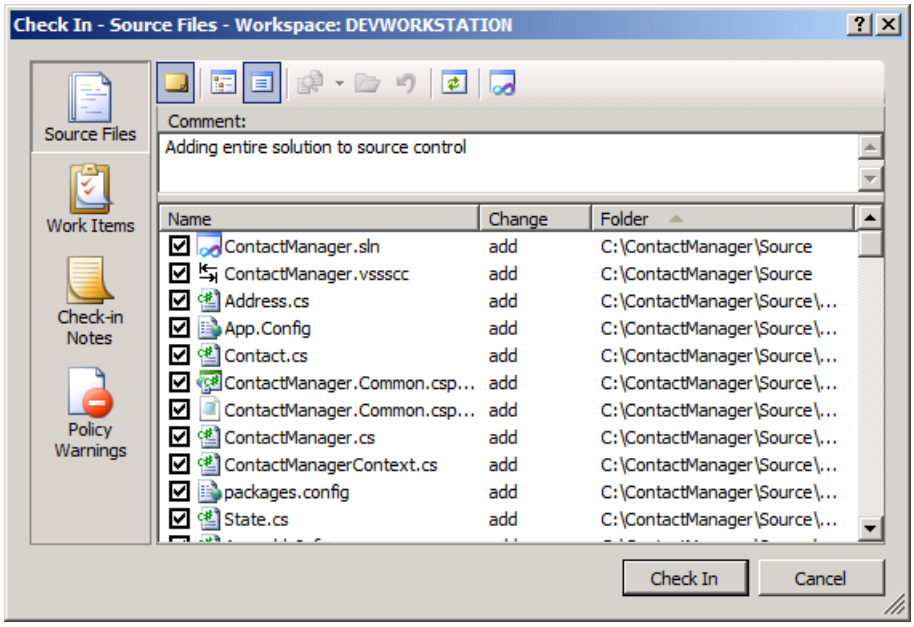
4. Verify that the **Source Control Explorer** tab displays the content you've added within the server folder structure for the team project.



> **NOTE**
>
> The **Source Control Explorer** tab displays your content with no further prompting because you added your solution to a mapped folder on the local file system. If your solution was in an unmapped location, you'd be prompted to specify folder locations in both TFS and your local file system.

5. On the **Source Control Explorer** tab, in the **Folders** pane, right-click the team project (for example, **ContactManager**), and then click **Check In Pending Changes**.

6. In the **Check In – Source Files** dialog box, type a comment, and then click **Check In**.



At this point you have added your solution to source control in TFS.

Add External Dependencies to Source Control

When you add a project or solution to source control, any files and folders within your project or solution will also be added. However, in a lot of cases, projects and solutions also rely on external dependencies, like local assemblies, to function properly. You need to add any such resources to source control to let both Team Build and other members of the developer team build your code successfully.

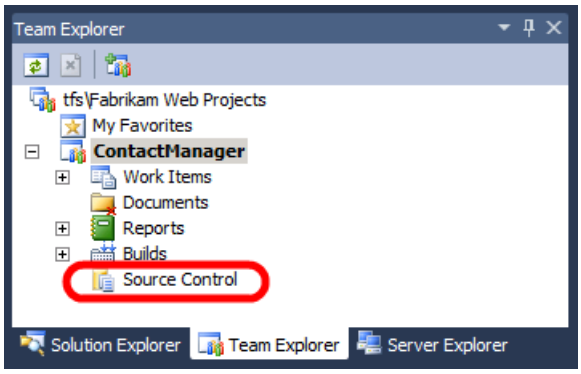
For example, the folder structure for the Contact Manager sample solution includes a folder named packages. This contains the assembly and various supporting resources for the ADO.NET Entity Framework 4.1. The packages folder is not part of the Contact Manager solution, but the solution will not build successfully without it. To enable Team Build to build the solution, you need to add the packages folder to source control.

> **NOTE**
>
> The inclusion of a packages folder is typical of what happens when you add the Entity Framework, or similar resources, to your solution using the NuGet extension for Visual Studio 2010.

To add non-project content to source control

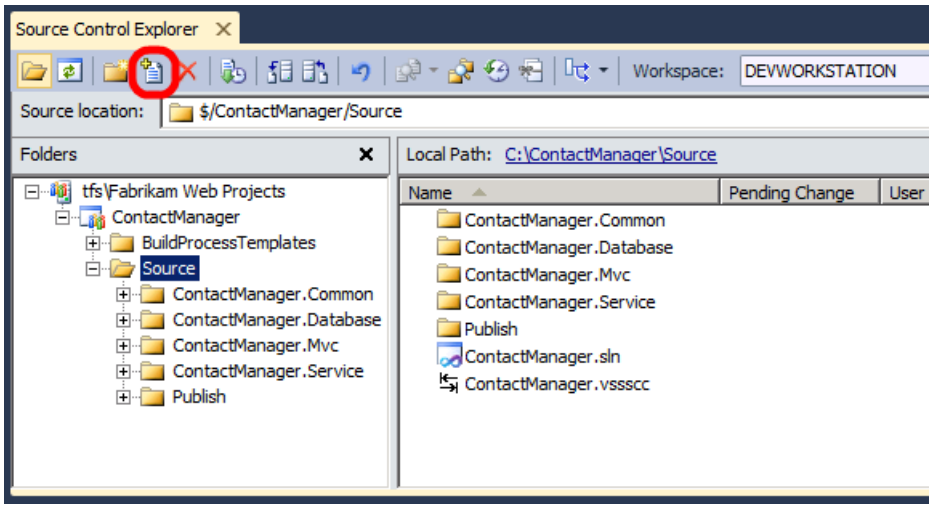
1. Ensure that the items you want to add (for example, the packages folder) are in an appropriate location within a mapped folder on your local file system.

2. In Visual Studio 2010, In the **Team Explorer** window, expand your team project, and then double-click **Source Control**.

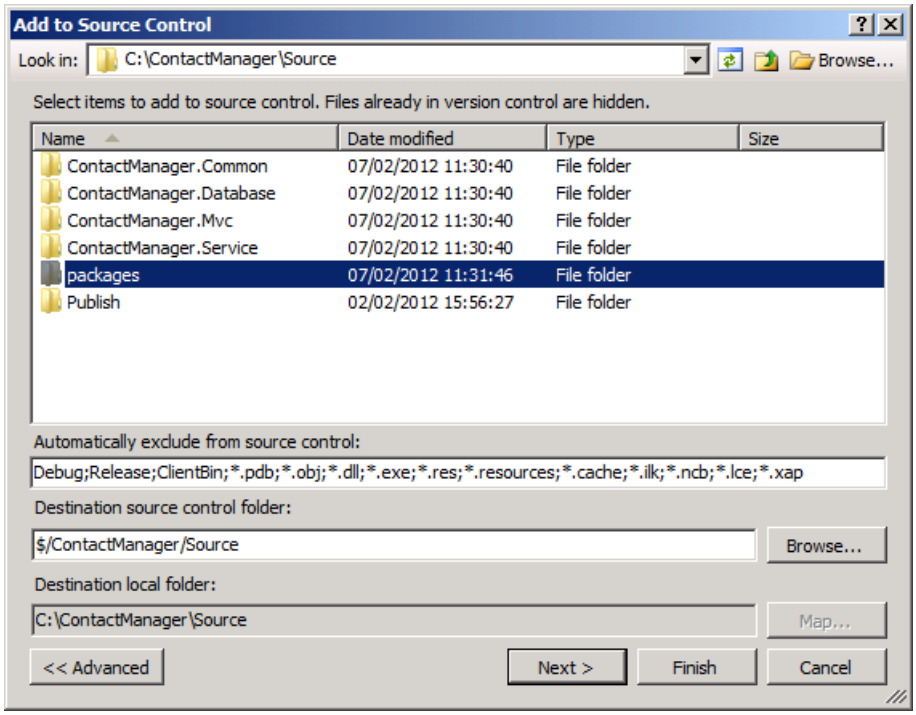


3. On the **Source Control Explorer** tab, in the **Folders** pane, select the folder that contains the item or items you want to add.

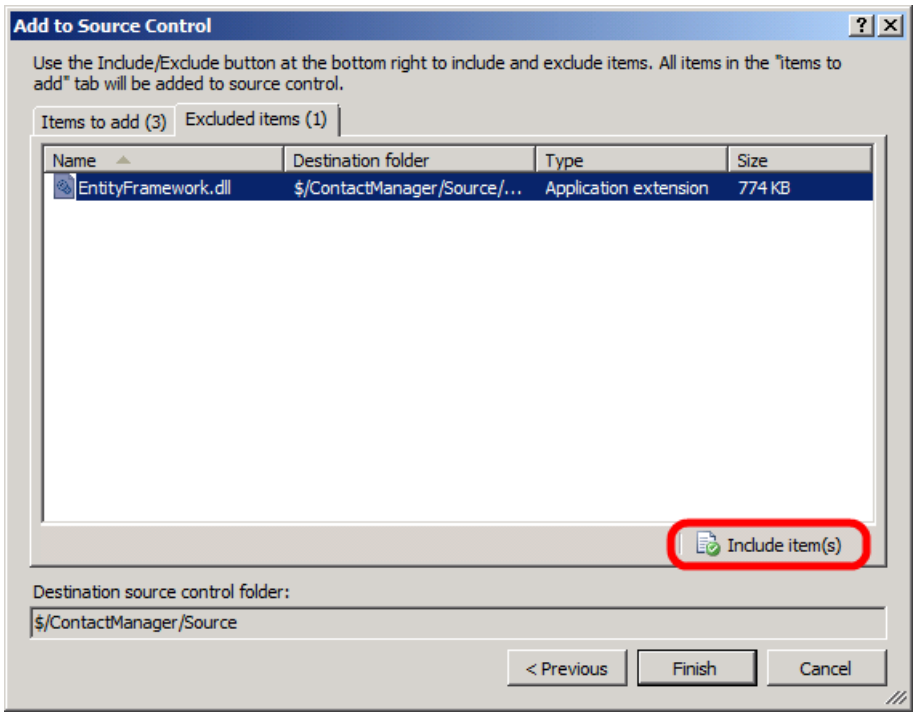
4. Click the **Add Items to Folder** button.



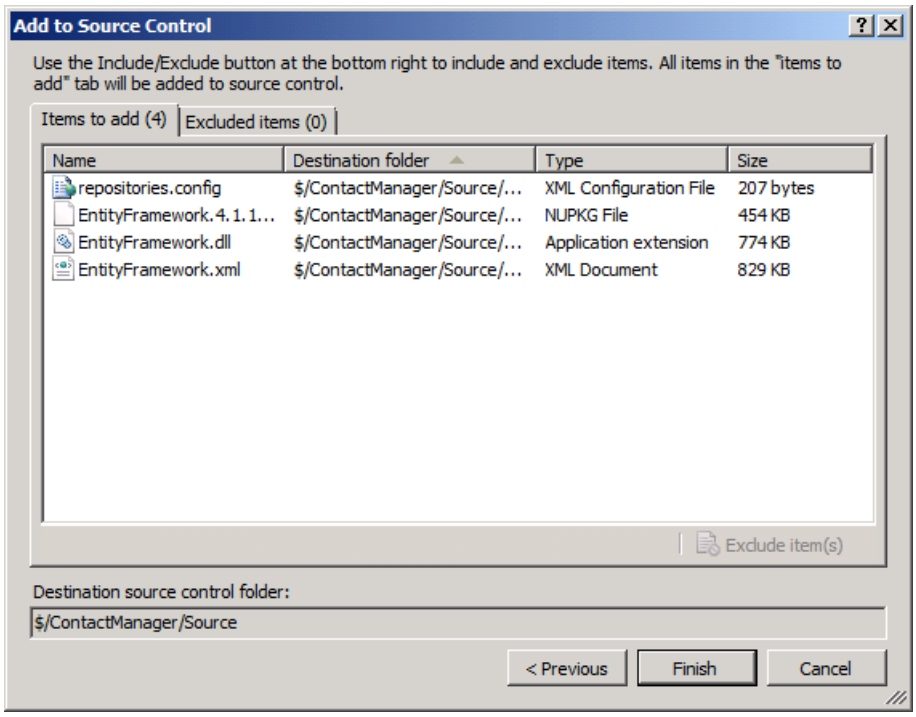
5. In the **Add to Source Control** dialog box, select the folder or items you want to add, and then click **Next**.



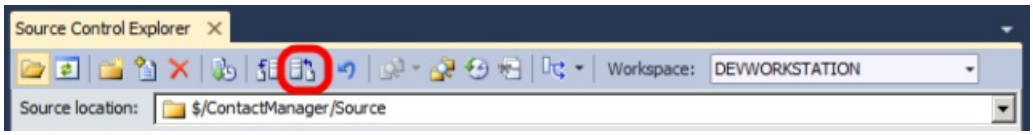
6. On the **Excluded items** tab, select any required items that have been automatically excluded (for example, assemblies), and then click **Include item(s)**.



7. On the **Items to add** tab, verify that all the files you want to include are listed, and then click **Finish**.



8. In the **Source Control Explorer** window, click the **Check In** button.



9. In the **Check In – Source Files** dialog box, type a comment, and then click **Check In**.

At this point, you have added the external dependencies for your solution to source control.

Conclusion

This topic described how to connect to a team project, map a folder structure, and add content to source control. For more information on how to work with items under source control, see Using Version Control.

The next topic, Configuring a TFS Build Server for Web Deployment, describes how to prepare a TFS Team Build server to build and deploy your solution.

Further Reading

For more comprehensive information on working with source control in TFS, see Using Version Control.

PREVIOUS NEXT

Configuring a TFS Build Server for Web Deployment

2/20/2020 • 5 minutes to read • Edit Online

by Jason Lee

Download PDF

> This topic describes how to prepare a Team Foundation Server (TFS) build server to build and deploy your solutions using Team Build and the Internet Information Services (IIS) Web Deployment Tool (Web Deploy).

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Project File, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Task Overview

To prepare a build server to build and deploy your solutions, you'll need to:

- Install and configure the TFS build service.
- Install Visual Studio 2010.
- Install any products or components that are required to build your solution, like versions of the .NET Framework or ASP.NET MVC.
- Install Web Deploy 2.0 or later.

This topic will show you how to perform these procedures or point to other resources where they exist. The tasks and walkthroughs in this topic assume that:

- You're starting with a clean server build running Windows Server 2008 R2 Service Pack 1.
- The server is domain-joined with a static IP address.
- You've installed the TFS application tier on a separate server, as described in Enterprise Web Deployment: Scenario Overview.

Who Performs These Procedures?

In most cases, a TFS administrator will be responsible for configuring build servers. In some cases, the developer team may take ownership of specific build servers.

Install and Configure the TFS Build Service

When you configure a build server, your first task is to install and configure the TFS build service. As part of this process, you'll need to:

- Install the TFS build service and configure a service account. Any build tasks, including deployment, will run using the identity of the build service account.
- Create a *build controller* and one or more *build agents*. Each build controller manages a set of build agents.

When you queue a build, the build controller assigns the build task to an available build agent. Each team project collection in TFS is mapped to a single build controller.

- Configure a drop folder for your build outputs. This is a network share. Any build outputs, like web deployment packages, are sent to the drop folder.

The Administering Team Foundation Build chapter on MSDN contains all the resources you need in order to perform these tasks:

- For a conceptual overview of Team Foundation Build, including the build service, build controllers, and build agents, see Understanding a Team Foundation Build System.
- For information on installing and configuring the build service, see Configure a Build Machine.
- For information on creating build controllers, see Create and Work with a Build Controller.
- For information on creating build agents, see Create and Work with Build Agents.
- For information on creating and configuring drop folders, see Set Up Drop Folders.

Install Required Products and Components

To enable the build server to build your solutions, you must install any products, components, or assemblies that your solution requires. Before you install any web platform components, you should install Visual Studio 2010 (any version) on the build server. This ensures that the core Microsoft Build Engine (MSBuild) target files and the Web Publishing Pipeline (WPP) target files are available to the build service. The Visual Studio installer should also install Web Deploy, which you'll need if you plan to deploy web packages as part of your build process.

The best way to install common web platform components is to use the Web Platform Installer. This ensures that you're installing the latest version of each product, and it also automatically detects and installs any prerequisites for each product. In the case of the Contact Manager solution, you should use the Web Platform Installer to install these products and components:

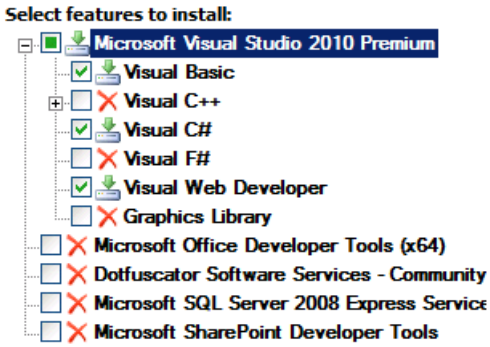
- **.NET Framework 4.0**. This is required to run applications that were built on this version of the .NET Framework.
- **Web Deployment Tool 2.1 or later**. This installs Web Deploy (and its underlying executable, MSDeploy.exe) on your server. As part of this process, it installs and starts the Web Deployment Agent Service. This service lets you deploy web packages from a remote computer.
- **ASP.NET MVC 3**. This installs the assemblies you need to run ASP.NET MVC 3 applications.

To install the required products and components

1. Install Visual Studio 2010. When prompted to select features to install, you should include:

   a. Any programming languages that you need to compile.

   b. Visual Web Developer. This ensures that the WPP targets are added to your build server.

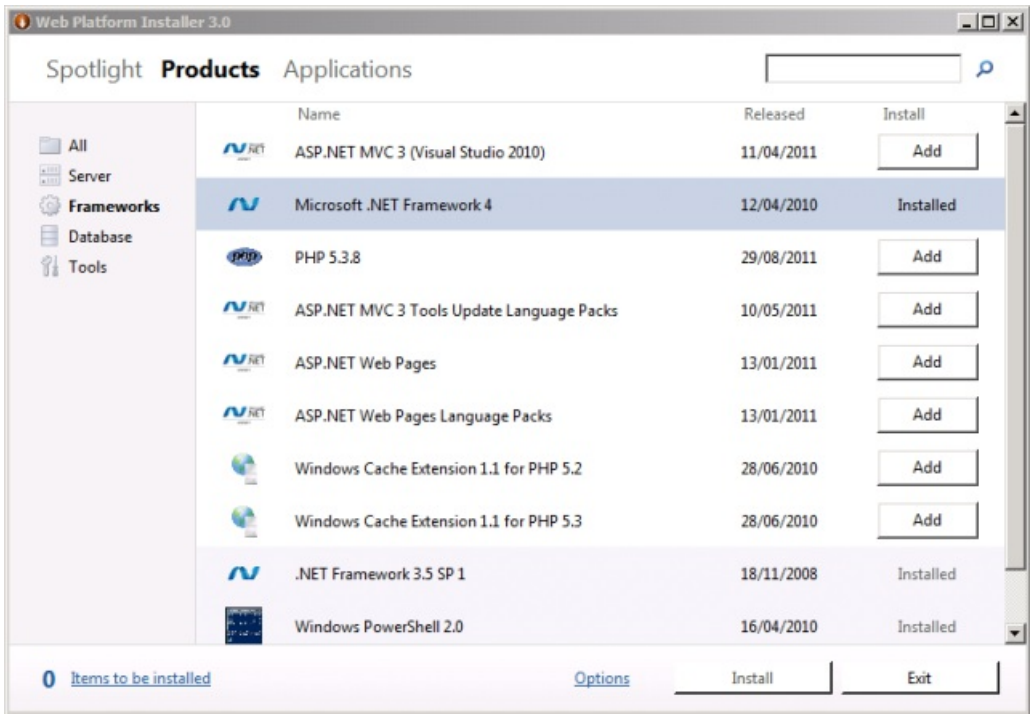


2. When the installation of Visual Studio 2010 is complete, download and install Visual Studio 2010 Service Pack 1 (if it's not already included in your installation media).

> **NOTE**
> Visual Studio 2010 Service Pack 1 resolves a bug that can prevent MSBuild from locating the MSDeploy executable.

3. Download and launch the Web Platform Installer.
4. At the top of the **Web Platform Installer 3.0** window, click **Products**.
5. On the left side of the window, in the navigation pane, click **Frameworks**.
6. In the **Microsoft .NET Framework 4** row, if the .NET Framework is not already installed, click **Add**.

> **NOTE**
> You may have already installed the .NET Framework 4.0 through Windows Update. If a product or component is already installed, the Web Platform Installer will indicate this by replacing the **Add** button with the text **Installed**.



7. In the **ASP.NET MVC 3 (Visual Studio 2010)** row, click **Add**.
8. In the navigation pane, click **Server**.
9. In the **Web Deployment Tool 2.1** row, click **Add**.
10. Click **Install**. The Web Platform Installer will show you a list of products—together with any associated dependencies—to be installed and will prompt you to accept the license terms.
11. Review the license terms, and if you consent to the terms, click **I Accept**.
12. When the installation is complete, click **Finish**, and then close the **Web Platform Installer 3.0** window.

> **NOTE**
> If your deployment process includes the use of tools like VSDBCMD.exe or SQLCMD.exe, you'll need to ensure that these are installed on your build server. VSDBCMD.exe is a Visual Studio tool and is typically added to the server when you install Team Foundation Build. SQLCMD.exe is a SQL Server tool. You can download a stand-alone version of SQLCMD.exe from the Microsoft SQL Server 2008 R2 Feature Pack page.

Conclusion

At this point, your build server is ready to start building and deploying your web application projects. The next topic, Creating a Build Definition That Supports Deployment, describes how to create and configure a build definition to control when and how your projects are built and deployed.

Further Reading

For more general guidance on working with Team Build, see Administering Team Foundation Build.

PREVIOUS NEXT

Creating a Build Definition That Supports Deployment

2/20/2020 • 8 minutes to read • Edit Online

by Jason Lee

Download PDF

> If you want to perform any kind of build in Team Foundation Server (TFS) 2010, you need to create a build definition within your team project. This topic describes how to create a new build definition in TFS and how to control web deployment as part of the build process in Team Build.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Project File, in which the build and deployment process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Task Overview

A build definition is the mechanism that controls how and when builds occur for team projects in TFS. Each build definition specifies:

- The things you want to build, like Visual Studio solution files or custom Microsoft Build Engine (MSBuild) project files.
- The criteria that determine when a build should take place, like manual triggers, continuous integration (CI), or gated check-ins.
- The location to which Team Build should send build outputs, including deployment artifacts like web packages and database scripts.
- The amount of time that each build should be retained.
- Various other parameters of the build process.

> **NOTE**
>
> For more information on build definitions, see Define Your Build Process.

This topic will show you how to create a build definition that uses CI, so that a build is triggered when a developer checks in new content. If the build succeeds, the build service runs a custom project file to deploy the solution to a test environment.

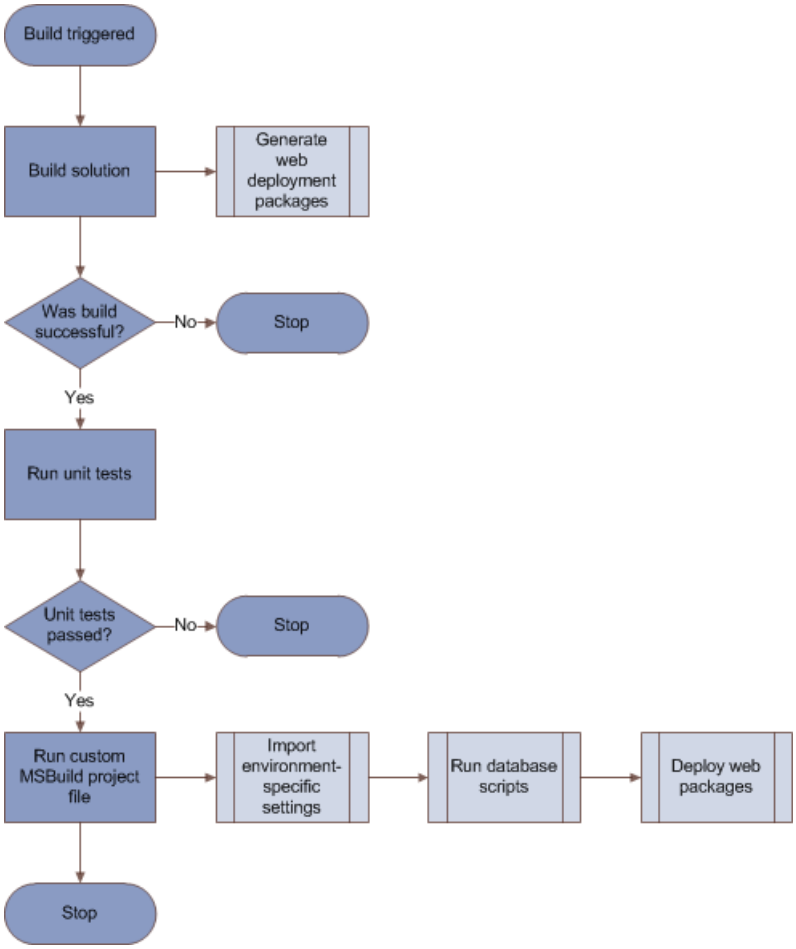
When you trigger a build, these actions need to happen:

- First, Team Build should build the solution. As part of this process, Team Build will invoke the Web Publishing Pipeline (WPP) to generate web deployment packages for each of the web application projects in the

solution. Team Build will also run any unit tests associated with the solution.

- If the solution build fails, Team Build should take no further action. Unit test failures should be treated as a build failure.
- If the solution build succeeds, Team Build should run the custom project file that controls the deployment of the solution. As part of this process, Team Build will invoke the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) to install the packaged web applications on the destination web servers, and it will invoke the VSDBCMD.exe utility to run database creation scripts on the destination database servers.

This illustrates the process:



The Contact Manager sample solution includes a custom MSBuild project file, *Publish.proj*, that you can run from MSBuild or Team Build. As described in Understanding the Build Process, this project file defines the logic that deploys your web packages and databases to a target environment. The file includes logic that omits the building and packaging process if it's running in Team Build, leaving just the deployment tasks to run. This is because when you automate deployment in this way, you'll typically want to ensure that the solution builds successfully and passes any unit tests before the deployment process commences.

The next section explains how to implement this process by creating a new build definition.

> **NOTE**
>
> This procedure—in which a single automated process builds, tests, and deploys a solution—is likely to be most suited to deployment to test environments. For staging and production environments you're a lot more likely to want to deploy content from a previous build that you've already verified and validated in a test environment. This approach is described in the next topic, Deploying a Specific Build.

Who Performs This Procedure?

Typically, a TFS administrator performs this procedure. In some cases, a developer team leader may take responsibility for the team project collection in TFS. In order to create a new build definition, you need to be a

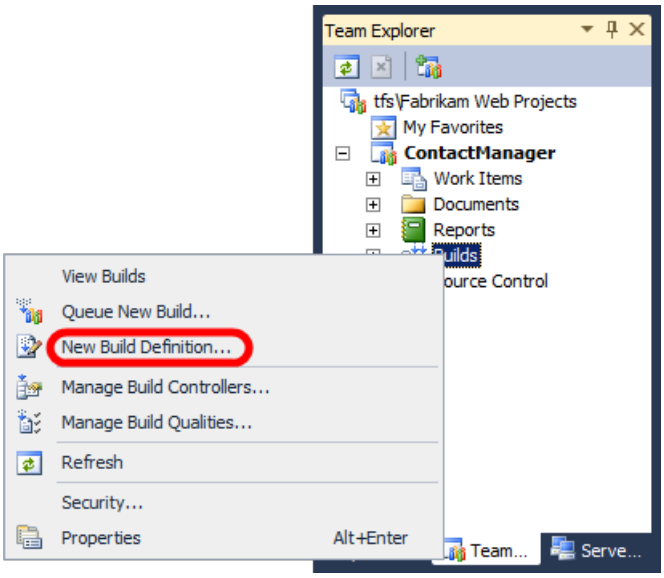
member of the **Project Collection Build Administrators** group for the team project collection that contains your solution.

Create a Build Definition for CI and Deployment

The next procedure describes how to create a build definition that CI triggers. If the build succeeds, the solution is deployed using the logic in a custom MSBuild project file.

To create a build definition for CI and deployment

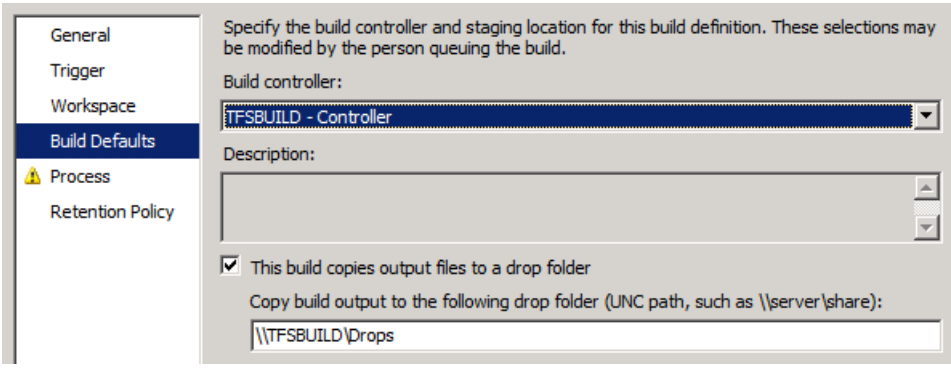
1. In Visual Studio 2010, in the **Team Explorer** window, expand your team project node, right-click **Builds**, and then click **New Build Definition**.



2. On the **General** tab, give the build definition a name (for example, **DeployToTest**) and an optional description.

3. On the **Trigger** tab, select the criteria on which you want to trigger a new build. For example, if you want to build the solution and deploy to the test environment every time a developer checks in new code, select **Continuous Integration**.

4. On the **Build Defaults** tab, in the **Copy build output to the following drop folder** box, type the Universal Naming Convention (UNC) path of your drop folder (for example, **\TFSBUILD\Drops**).

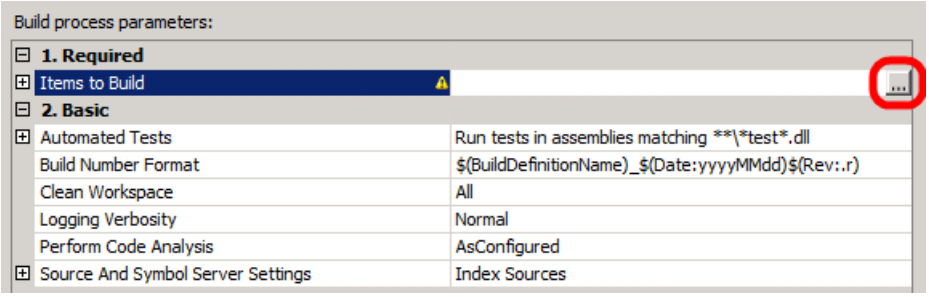


> **NOTE**
>
> This drop location stores several builds, depending on the retention policy you configure. When you want to publish deployment artifacts from a specific build to a staging or production environment, this is where you'll find them.

5. On the **Process** tab, in the **Build process file** dropdown list, leave **DefaultTemplate.xaml** selected.

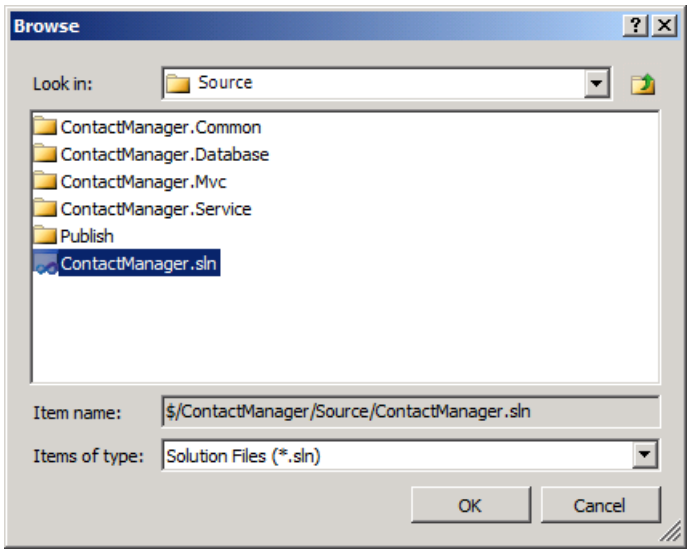
This is one of the default build process templates that get added to all new team projects.

6. In the **Build process parameters** table, click in the **Items to Build** row, and then click the **ellipsis** button.



7. In the **Items to Build** dialog box, click **Add**.

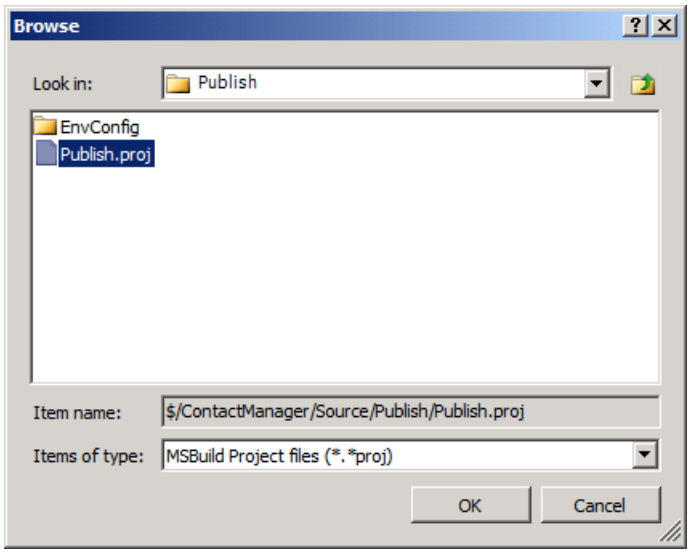
8. Browse to the location of your solution file, and then click **OK**.



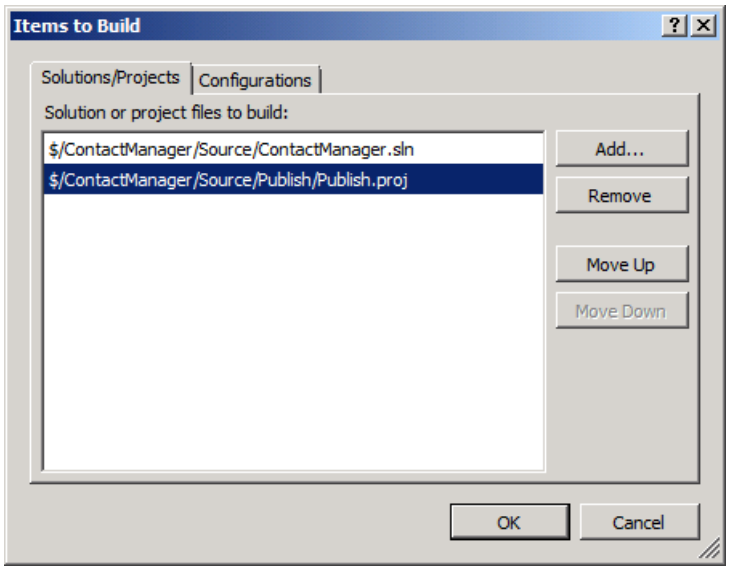
9. In the **Items to Build** dialog box, click **Add**.

10. In the **Items of type** dropdown list, select **MSBuild Project files**.

11. Browse to the location of the custom project file with which you control the deployment process, select the file, and then click **OK**.



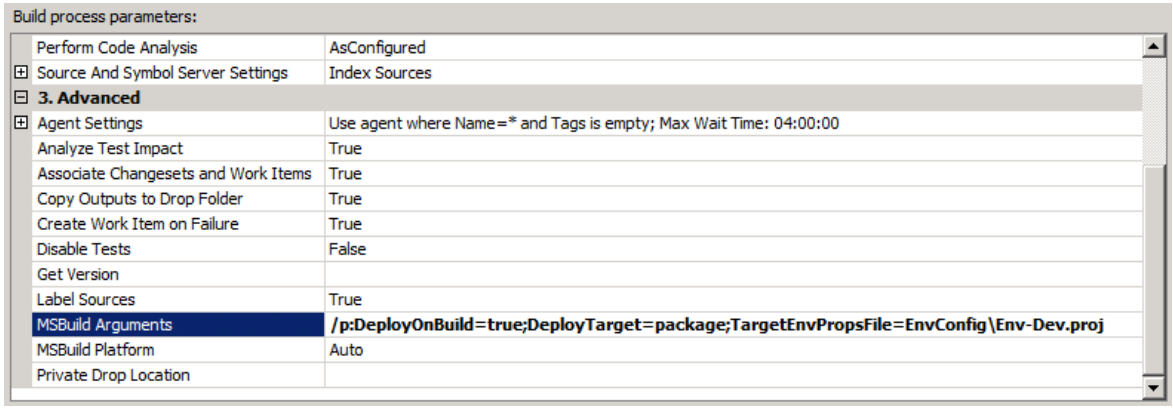
12. The **Items to Build** dialog box should now show two items. Click **OK**.



13. On the **Process** tab, in the **Build process parameters** table, expand the **Advanced** section.

14. In the **MSBuild Arguments** row, add any MSBuild command-line arguments that *either* of your items to build requires. In the Contact Manager solution scenario, these arguments are required:

```
/p:DeployOnBuild=true;DeployTarget=Package;
   TargetEnvPropsFile=EnvConfig\Env-Dev.proj
```



15. In this example:

    a. The **DeployOnBuild=true** and **DeployTarget=package** arguments are required when you build the Contact Manager solution. This instructs MSBuild to create web deployment packages after building each web application project, as described in Building and Packaging Web Application Projects.

    b. The **TargetEnvPropsFile** argument is required when you build the *Publish.proj* file. This property indicates the location of the environment-specific configuration file, as described in Understanding the Build Process.

16. On the **Retention Policy** tab, configure how many builds of each type you want to retain as required.

17. Click **Save**.

Queue a Build

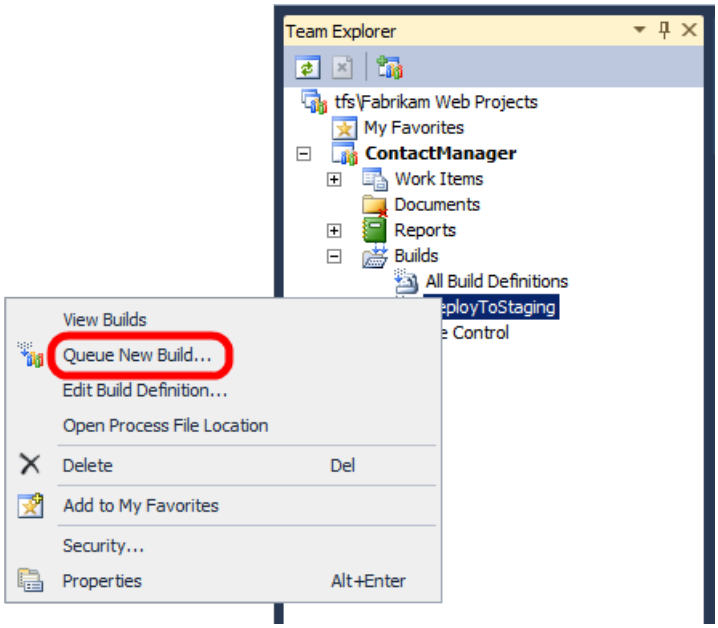
At this point, you have created at least one new build definition. The build process you defined will now run according to the triggers you specified in the build definition.

If you've configured your build definition to use CI, you can test your build definition in two ways:

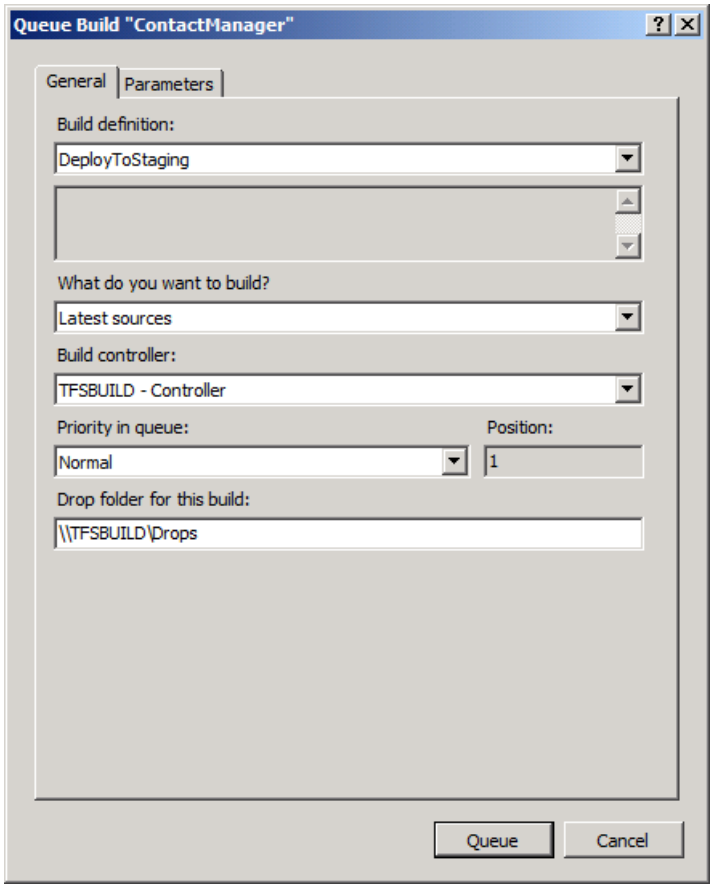
- Check in some content to the team project to trigger an automatic build.
- Queue a build manually.

To queue a build manually

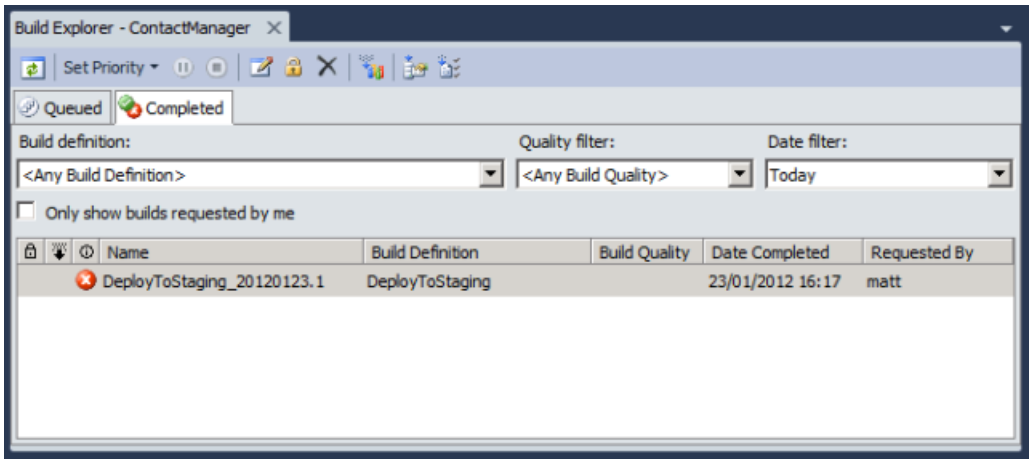
1. In the **Team Explorer** window, right-click the build definition, and then click **Queue New Build**.



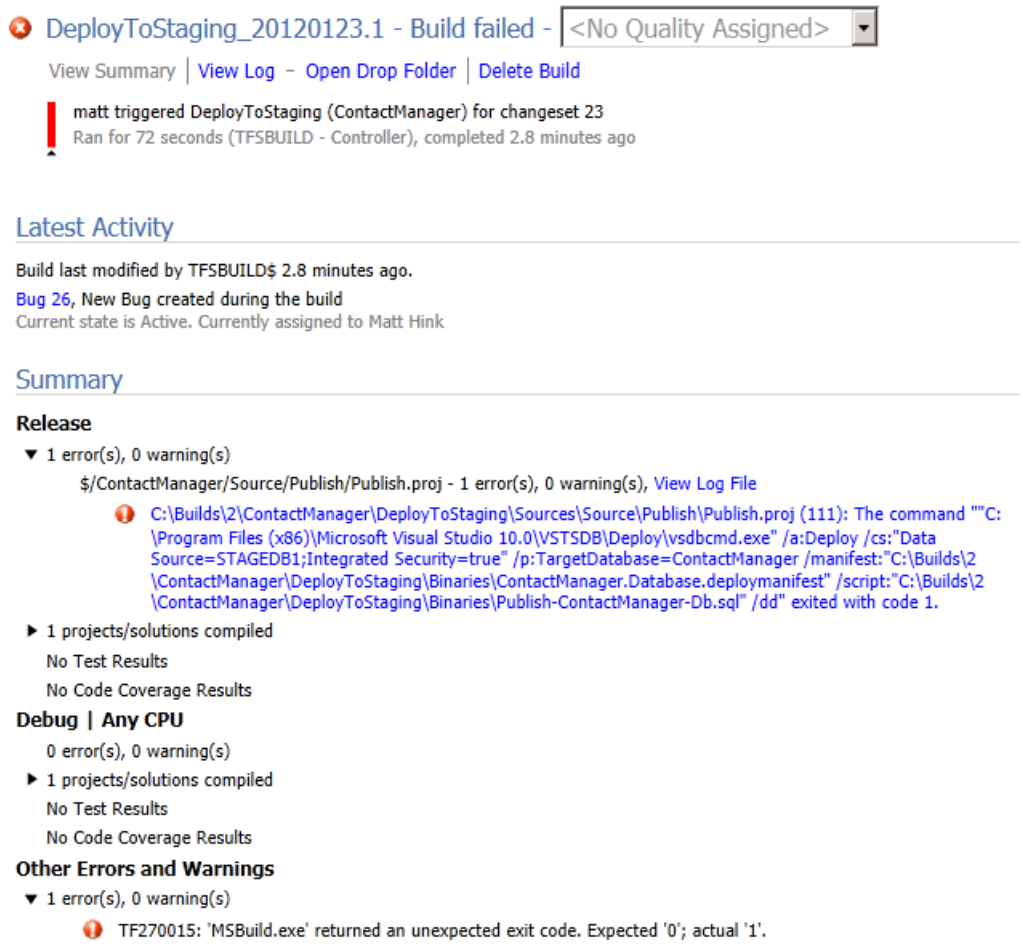
2. In the **Queue Build** dialog box, review the build properties, and then click **Queue**.



To review the progress and the outcome of a build—regardless of whether it was triggered manually or automatically—double-click the build definition in the **Team Explorer** window. This will open a **Build Explorer** tab.



From here, you can troubleshoot failed builds. If you double-click an individual build, you can view summary information and click through to detailed log files.



You can use this information to troubleshoot failed builds and address any problems before you attempt another build.

> **NOTE**
>
> Builds that execute deployment logic are likely to fail until you have granted the build server any permissions required in the destination environment. For more information, see Configuring Permissions for Team Build Deployment.

Monitor the Build Process

TFS provides a broad range of functionality to help you monitor the build process. For example, TFS can send you an email or display alerts in your taskbar notification area when a build has completed. For more

information, see Run and Monitor Builds.

Conclusion

This topic described how to create a build definition in TFS. The build definition is configured for CI, so the build process runs whenever a developer checks in content to the team project. The build definition executes a custom MSBuild project file to deploy web packages and database scripts to a target server environment.

In order for an automated deployment to succeed as part of a build process, you'll need to grant appropriate permissions to the build service account on the target web servers and the target database server. The final topic in this tutorial, Configuring Permissions for Team Build Deployment, describes how to identify and configure the permissions required for automated deployment from a Team Build server.

Further Reading

For more information on creating build definitions, see Create a Basic Build Definition and Define Your Build Process. For more guidance on queuing builds, see Queue a Build.

PREVIOUS NEXT

Deploying a Specific Build

2/20/2020 • 6 minutes to read • Edit Online

by Jason Lee

Download PDF

> This topic describes how to deploy web packages and database scripts from a specific previous build to a new destination, like a staging or production environment.

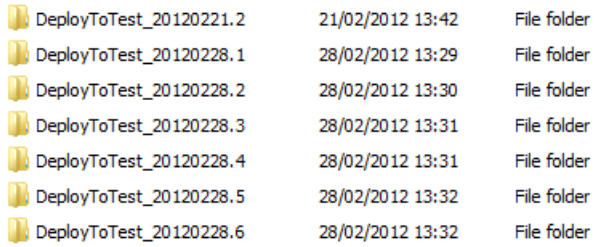
This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Project File, in which the build and deployment process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Task Overview

Until now, the topics in this tutorial set have focused on how to build, package, and deploy web applications and databases as part of a single-step or automated process. However, in some common scenarios, you'll want to select the resources that you deploy from a list of builds in a drop folder. In other words, the latest build may not be the build you want to deploy.

Consider the continuous integration (CI) scenario described in the previous topic, Creating a Build Definition That Supports Deployment. You've created a build definition in Team Foundation Server (TFS) 2010. Every time a developer checks code into TFS, Team Build will build your code, create web packages and database scripts as part of the build process, run any unit tests, and deploy your resources to a test environment. Depending on the retention policy you configured when you created the build definition, TFS will retain a certain number of previous builds.



DeployToTest_20120221.2	21/02/2012 13:42	File folder
DeployToTest_20120228.1	28/02/2012 13:29	File folder
DeployToTest_20120228.2	28/02/2012 13:30	File folder
DeployToTest_20120228.3	28/02/2012 13:31	File folder
DeployToTest_20120228.4	28/02/2012 13:31	File folder
DeployToTest_20120228.5	28/02/2012 13:32	File folder
DeployToTest_20120228.6	28/02/2012 13:32	File folder

Now, suppose you've performed verification and validation testing against one of these builds in your test environment, and you're ready to deploy your application to a staging environment. In the meantime, developers may have checked in new code. You don't want to rebuild the solution and deploy to the staging environment, and you don't want to deploy the latest build to the staging environment. Instead, you want to deploy the specific build that you've verified and validated on the test servers.

To accomplish this, you need to tell the Microsoft Build Engine (MSBuild) where to find the web packages and database scripts that a specific build generated.

Overriding the OutputRoot Property

In the sample solution, the *Publish.proj* file declares a property named **OutputRoot**. As the name suggests, this is the root folder that contains everything that the build process generates. In the *Publish.proj* file, you can see that the **OutputRoot** property refers to the root location for all deployment resources.

> **NOTE**
>
> **OutputRoot** is a commonly used property name. Visual C# and Visual Basic project files also declare this property to store the root location for all build outputs.

```
<PropertyGroup>
  <!--This is where the .deploymanifest file will be written to during a build-->
  <_DbDeployManifestPath>
    $(OutputRoot)ContactManager.Database.deploymanifest
  </_DbDeployManifestPath>

  <!-- The folder where the .zip and .cmd file will be located for
               ContactManager.Mvc Web project -->
  <_ContactManagerDest>
    $(OutputRoot)_PublishedWebsites\ContactManager.Mvc_Package\
  </_ContactManagerDest>

  <!-- The folder where the .zip and .cmd file will be located for
               ContactManager.Service Web project -->
   <_ContactManagerSvcDest>
    $(OutputRoot)_PublishedWebsites\ContactManager.Service_Package\
  </_ContactManagerSvcDest>

  <!-- ... -->
</PropertyGroup>
```

If you want your project file to deploy web packages and database scripts from a different location—like the outputs of a previous TFS build—you simply need to override the **OutputRoot** property. You should set the property value to the relevant build folder on the Team Build server. If you were running MSBuild from the command line, you could specify a value for **OutputRoot** as a command-line argument:

```
msbuild.exe Publish.proj /p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
  /p:OutputRoot=\\TFSBUILD\Drops\DeployToTest\DeployToTest_20120228.3\
```

In practice, however, you'd also want to skip the **Build** target—there's no point in building your solution if you don't plan to use the build outputs. You could do this by specifying the targets you want to execute from the command line:

```
msbuild.exe Publish.proj /p:TargetEnvPropsFile=EnvConfig\Env-Dev.proj
  /p:OutputRoot=\\TFSBUILD\Drops\DeployToTest\DeployToTest_20120228.3\
  /target:GatherPackagesForPublishing;PublishDBPackages;PublishWebPackages
```

However, in most cases, you'll want to build your deployment logic into a TFS build definition. This enables users with the **Queue builds** permission to trigger the deployment from any Visual Studio installation with a connection to the TFS server.

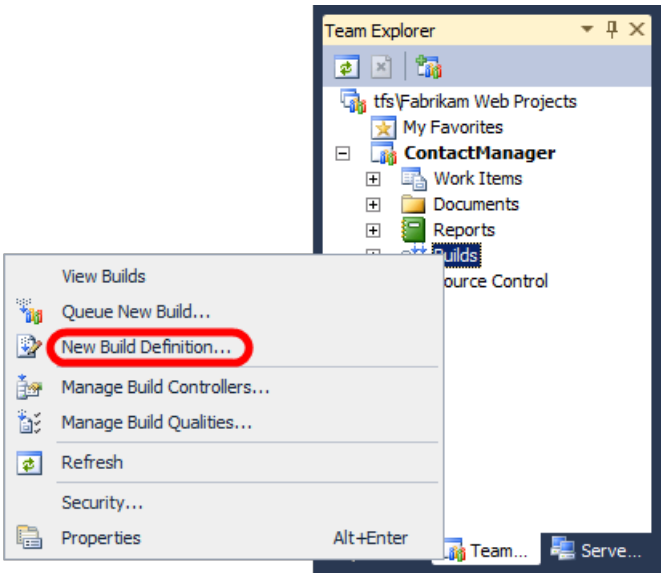
Creating a Build Definition to Deploy Specific Builds

The next procedure describes how to create a build definition that enables users to trigger deployments to a staging environment with a single command.

In this case, you don't want the build definition to actually build anything—you just want it to execute the deployment logic in your custom project file. The *Publish.proj* file includes conditional logic that skips the **Build** target if the file is running in Team Build. It does this by evaluating the built-in **BuildingInTeamBuild** property, which is automatically set to **true** if you run your project file in Team Build. As a result, you can skip the build process and simply run the project file to deploy an existing build.

To create a build definition to trigger deployment manually

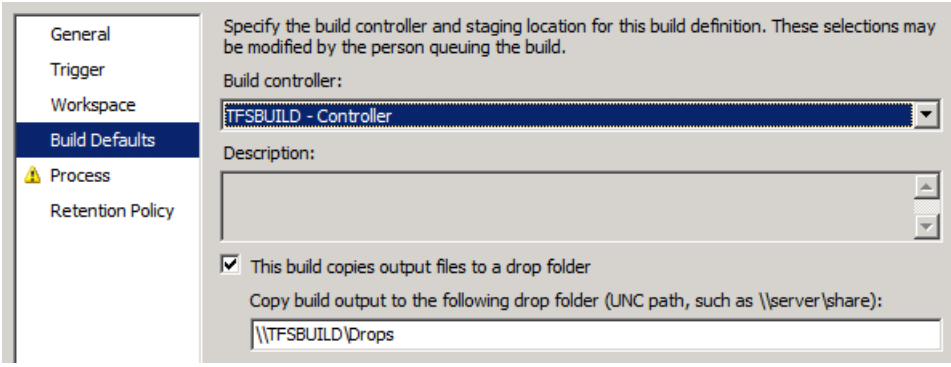
1. In Visual Studio 2010, in the **Team Explorer** window, expand your team project node, right-click **Builds**, and then click **New Build Definition**.



2. On the **General** tab, give the build definition a name (for example, **DeployToStaging**) and an optional description.

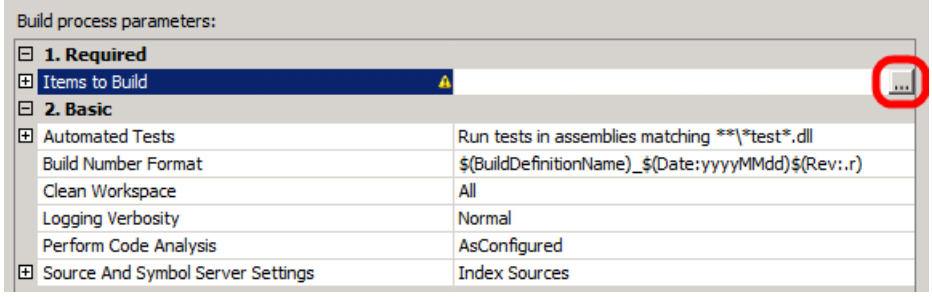
3. On the **Trigger** tab, select **Manual – Check-ins do not trigger a new build**.

4. On the **Build Defaults** tab, in the **Copy build output to the following drop folder** box, type the Universal Naming Convention (UNC) path of your drop folder (for example, **\TFSBUILD\Drops**).



5. On the **Process** tab, in the **Build process file** dropdown list, leave **DefaultTemplate.xaml** selected. This is one of the default build process templates that get added to all new team projects.

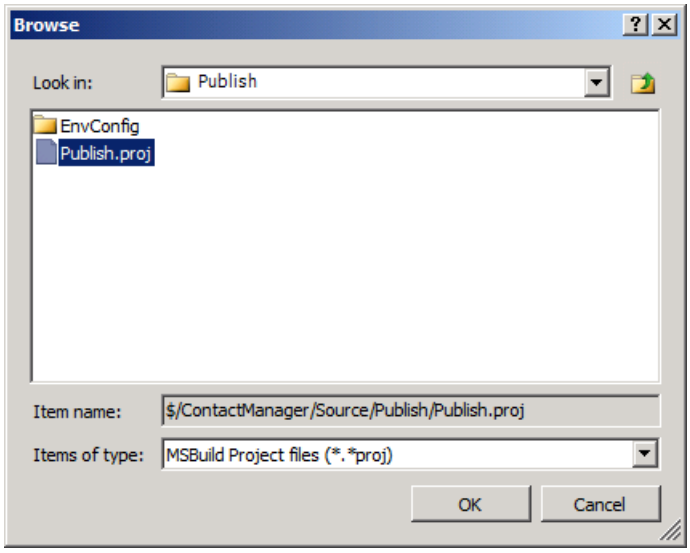
6. In the **Build process parameters** table, click in the **Items to Build** row, and then click the **ellipsis** button.



7. In the **Items to Build** dialog box, click **Add**.

8. In the **Items of type** dropdown list, select **MSBuild Project files**.

9. Browse to the location of the custom project file with which you control the deployment process, select the file, and then click **OK**.

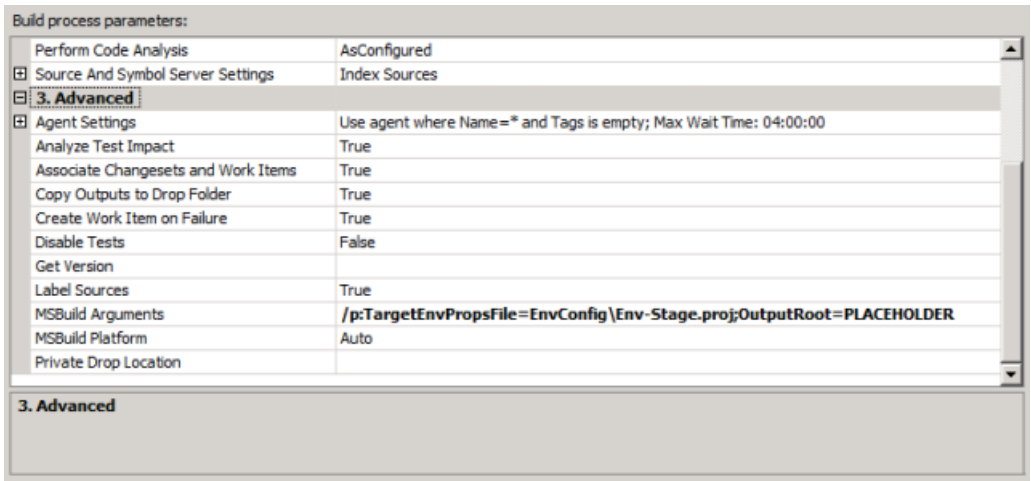


10. In the **Items to Build** dialog box, click **OK**.

11. In the **Build process parameters** table, expand the **Advanced** section.

12. In the **MSBuild Arguments** row, specify the location of your environment-specific project file and add a placeholder for the location of your build folder:

```
/p:TargetEnvPropsFile=EnvConfig\Env-Stage.proj;
OutputRoot=PLACEHOLDER
```



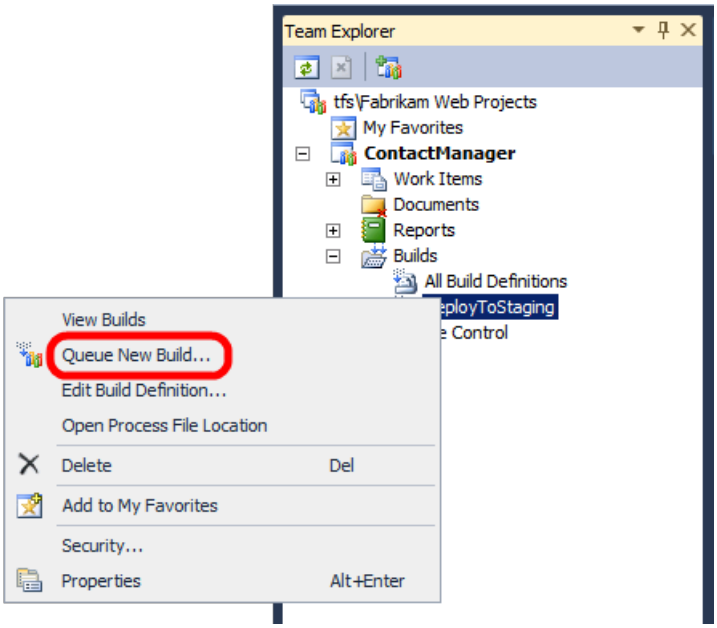
> **NOTE**
>
> You'll need to override the **OutputRoot** value every time you queue a build. This is covered in the next procedure.

13. Click **Save**.

When you trigger a build, you need to update the **OutputRoot** property to point to the build you want to deploy.

To deploy a specific build from a build definition

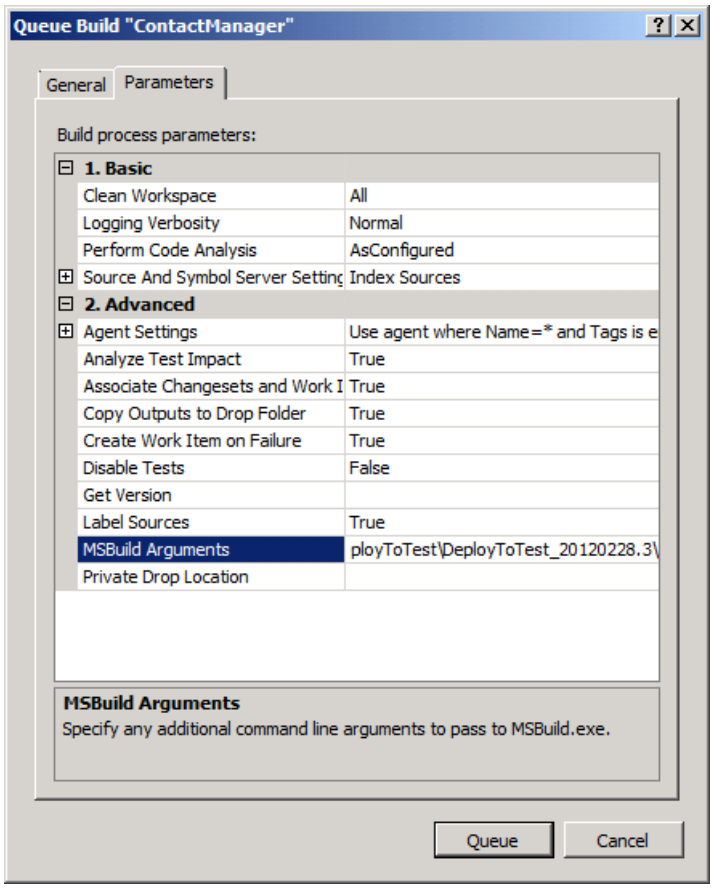
1. In the **Team Explorer** window, right-click the build definition, and then click **Queue New Build**.



2. In the **Queue Build** dialog box, on the **Parameters** tab, expand the **Advanced** section.

3. In the **MSBuild Arguments** row, replace the value of the **OutputRoot** property with the location of your build folder. For example:

```
/p:TargetEnvPropsFile=EnvConfig\Env-Stage.proj;
   OutputRoot=\\TFSBUILD\Drops\DeployToTest\DeployToTest_20120228.3\
```



> **NOTE**
>
> Be sure to include a trailing slash at the end of the path to your build folder.

4. Click **Queue**.

When you queue the build, the project file will deploy the database scripts and web packages from the build drop folder you specified in the **OutputRoot** property.

Conclusion

This topic described how to publish deployment resources, like web packages and database scripts, from a specific previous build using the split project file deployment model. It explained how to override the **OutputRoot** property and how to incorporate the deployment logic into a TFS build definition.

Further Reading

For more information on creating build definitions, see Create a Basic Build Definition and Define Your Build Process. For more guidance on queuing builds, see Queue a Build.

PREVIOUS NEXT

Configuring Permissions for Team Build Deployment

2/20/2020 • 4 minutes to read • Edit Online

by Jason Lee

Download PDF

> This topic describes how to configure permissions to enable your build server to deploy content to web servers and database servers as part of an automated build process.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Project File, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Task Overview

When you install the Team Foundation Server (TFS) 2010 build service, you specify the identity with which you want the service to run. By default, this is the Network Service account. Alternatively, you can configure the build service to run using a domain account.

Any deployment tasks that require Windows authentication, and that you plan to automate using Team Build, will run using the build service identity. As such, you'll need to grant the build service identity any required permissions on your web servers and your database servers.

> **NOTE**
>
> The Network Service account uses the machine account to authenticate to other computers. Machine accounts take the form *[domain name][machine name]*$—for example, **FABRIKAM\TFSBUILD$**. As such, if your build service runs using the Network Service identity, you should grant any required permissions to the machine account identity for your build server.

Configuring Web Server Permissions

As described in Choosing the Right Approach to Web Deployment, there are two main approaches you can use if you want to deploy web packages to a remote web server:

- Deploy the application from a remote location by targeting the *Web Deployment Agent Service* (also known as the remote agent) on the destination server.
- Deploy the application from a remote location by targeting the *Internet Information Services (IIS) Web Deploy Handler* on the destination server.

The remote agent has two key limitations in this case:

- The remote agent supports only NTLM authentication. In other words, the deployment must use the build service identity—you can't impersonate another account.

- To use the remote agent, the account that performs the deployment must be an administrator on the target server.

Together, these two limitations make the remote agent approach undesirable for an automated Team Build deployment. To use this approach, you'd need to make the build service account an administrator on any target web servers.

In contrast, the Web Deploy Handler approach offers various advantages:

- The Web Deploy Handler supports basic authentication over HTTPS, which allows you to pass the credentials of an alternative account to the IIS Web Deployment Tool (Web Deploy).
- You can configure target web servers to allow non-administrator users to deploy content to specific IIS websites using the Web Deploy Handler.

As a result, it's clearly preferable to target the Web Deploy Handler when you automate web package deployment from Team Build. This is the recommended process:

1. Create a low-privileged domain account to use for the deployment.
2. Configure the Web Deploy Handler and grant the account the required permissions to deploy content to a specific IIS website, as described in Configuring a Web Server for Web Deploy Publishing (Web Deploy Handler).
3. Invoke Web Deploy and target the Web Deploy Handler, using basic authentication and supplying the credentials of the domain account you created, to perform the deployment.

In the Contact Manager sample solution, you specify the authentication type (basic or NTLM), the Web Deploy credentials, and the endpoint address (remote agent or Web Deploy Handler) in the environment-specific project file. These values are used to formulate and run a Web Deploy command when the project file is executed. For more information, see Deploying Web Packages.

For more information on configuring the Web Deploy Handler, including how to configure permissions, see Configuring a Web Server for Web Deploy Publishing (Web Deploy Handler). For more information on configuring the remote agent, see Configuring a Web Server for Web Deploy Publishing (Remote Agent).

Configuring Database Server Permissions

To deploy a database to SQL Server, you must:

- Create a login for the deploying account on the SQL Server instance.
- Grant the login **DBCreator** permissions on the SQL Server instance.
- After the initial deployment, add the login to the **db_owner** role on the target database. This is required because on subsequent deployments, you're modifying an existing database rather than creating a new database.

You can authenticate to a SQL Server instance using either NTLM authentication or SQL Server authentication:

- If you use NTLM authentication, you need to grant the permissions described above to the build service account.
- If you use SQL Server authentication, you need to grant the permissions described above to the SQL Server account. You also need to include the SQL Server user name and password in the connection string you use to deploy the database.

For step-by-step details on how to configure permissions for database deployment, see Configuring a Database Server for Web Deploy Publishing.

Conclusion

At this point, you should understand the permissions required, together with the authentication options open to you, when you automate web application and database deployments from Team Build. You should also be able to implement the necessary permissions on IIS web servers and SQL Server database servers.

Further Reading

For more information on configuring Windows server environments to support remote deployment, see Configuring Server Environments for Web Deployment.

PREVIOUS

ASP.NET 4 - Enterprise Deployment Series 4 Advanced Topics

2/20/2020 • 2 minutes to read • Edit Online

This tutorial will show you how to perform various tasks that are required or desirable in a lot of enterprise deployment scenarios.

- Advanced Enterprise Web Deployment
- Performing a "What If" Deployment
- Customizing Database Deployments for Multiple Environments
- Deploying Database Role Memberships to Test Environments
- Deploying Membership Databases to Enterprise Environments
- Excluding Files and Folders from Deployment
- Taking Web Applications Offline with Web Deploy
- Running Windows PowerShell Scripts from MSBuild Project Files
- Troubleshooting the Packaging Process

Advanced Enterprise Web Deployment

2/20/2020 • 5 minutes to read • Edit Online

by Jason Lee

Download PDF

> This tutorial will show you how to perform various tasks that are required or desirable in a lot of enterprise deployment scenarios.
>
> For an Italian translation of these tutorials, visit http://www.lucamorelli.it.

This forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Build Process, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Scenario Overview

The high-level scenario for these tutorials is described in Enterprise Web Deployment: Scenario Overview. We recommend that you review this topic before you get started on this tutorial.

How to Use This Tutorial

- Each of the topics in this tutorial is self-contained and addresses a particular challenge or problem that occurs in enterprise deployment scenarios. You don't need to work through these topics in any particular order. However, this tutorial covers some advanced tasks. As such, you should familiarize yourself with the concepts and techniques that the Web Deployment in the Enterprise tutorial covers in order to gain the most benefit from this content.
- This tutorial includes these topics:
- Performing a "What If" Deployment. In a lot of scenarios, you'll want to determine the impact of a proposed deployment on a target environment or any existing content before you actually make any changes. This topic describes how you can run a "what if" deployment to generate log files and database update scripts as if you had deployed content to a target environment, without actually making any changes. Analyzing these resources can help you to spot any potential problems in advance of a live deployment.
- Customizing Database Deployments for Multiple Environments. When you deploy a database project to multiple destinations, you'll often want to customize the deployment properties for each target environment. For example, in test environments you'd typically recreate the database on every deployment, whereas in staging or production environments you'd be a lot more likely to make incremental updates to preserve your data. This topic describes how you can incorporate these property changes into your deployment logic by creating an environment-specific deployment configuration (.sqldeployment) file for each target environment.
- Deploying Database Role Memberships to Test Environments. When you recreate a database on every deployment—for example, as part of a continuous integration (CI) build and deploy to a test environment—you'll typically need to configure database role memberships every time. For example, you'll usually need to

grant permissions to the application pool identity associated with your web application. This topic describes how you can automate this process by adding a post-deployment SQL script to your deployment logic.

- Deploying Membership Databases to Enterprise Environments. ASP.NET membership databases have various characteristics that can complicate the deployment process. For example, a schema-only deployment will leave the database in a non-operational state. In most scenarios, it's preferable to create a membership database directly in each destination environment. However, if you do have to deploy a membership database, this topic describes some of the approaches you can use to meet the inherent challenges.
- Excluding Files and Folders from Deployment. In some scenarios, you'll want to tailor the contents of your web package to specific destination environments. For example, you might want to include full versions of JavaScript libraries when you deploy to a test environment, to support client-side debugging, but use minified versions of the libraries when you deploy to a staging or production environment. This topic describes how you can exclude specific files and folders from the package creation process.
- Taking Web Applications Offline with Web Deploy. When you deploy solutions to a staging or production environment, you'll often want to take your web applications offline for the duration of the deployment process. This topic describes how you can add an *App_offline.htm* file to your web application at the start of the deployment process and remove it at the end. While the *App_offline.htm* file is in place, any users who browse to the web application are automatically redirected to the *App_offline.htm* file.
- Running Windows PowerShell Scripts from MSBuild. Many deployment scenarios require more complex post-deployment actions, like adding custom event sources to the registry or configuring replication between SQL Server instances. These actions are often accomplished through Windows PowerShell scripts. This topic describes how to run Windows PowerShell scripts from a Microsoft Build Engine (MSBuild) project file as part of the build and deployment process.
- Troubleshooting the Packaging Process. The Web Publishing Pipeline (WPP) defines an MSBuild property named **EnablePackageProcessLoggingAndAssert** that you can use to generate in-depth information about the packaging process for web application projects. This topic describes what the property does and how to use it.

Key Technologies

This tutorial focuses on how to use these products and technologies to support automated build and web deployment:

- Visual Studio 2010 and Team Foundation Server (TFS) 2010
- MSBuild and TFS Team Build
- Internet Information Services (IIS) 7.5
- IIS Web Deployment Tool (Web Deploy) 2.1
- The VSDBCMD.exe database deployment utility

Other Tutorials in This Series

This forms part of a series of five tutorials on enterprise-scale web deployment. These are other tutorials in the series:

- Deploying Web Applications in Enterprise Scenarios. This introductory content provides the contextual background for the tutorial series. It describes the tutorial scenario, and it illustrates how the tasks and walkthroughs described throughout the series fit into a broader Application Lifecycle Management (ALM) process.
- Web Deployment in the Enterprise. This tutorial provides a conceptual introduction to MSBuild project files, the WPP, Web Deploy, and other related technologies. It explains how you can use these tools together to manage complex deployment processes.
- Configuring Server Environments for Web Deployment. This tutorial describes how to configure Windows servers to support various deployment scenarios, including remote web package deployment using the Web

Deployment Agent Service (the remote agent) or the Web Deploy Handler and remote database deployment. It provides guidance on choosing the appropriate deployment method for your own environment, and it describes how to use the Web Farm Framework (WFF) to replicate deployed web applications across all the web servers in a server farm.

- Configuring Team Foundation Server for Web Deployment. This tutorial describes how to configure TFS to support various deployment scenarios, including automated deployment as part of a CI process and manually triggered deployments of specific builds.

NEXT

Performing a "What If" Deployment

2/20/2020 • 7 minutes to read • Edit Online

by Jason Lee

Download PDF

> This topic describes how to perform "what if" (or simulated) deployments using the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) and VSDBCMD. This lets you determine the effects of your deployment logic on a particular target environment before you actually deploy your application.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Project File, in which the build and deployment process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Performing a "What If" Deployment for Web Packages

Web Deploy includes functionality that lets you perform deployments in "what if" (or trial) mode. When you deploy artifacts in "what if" mode, Web Deploy generates a log file as if you had performed the deployment, but it doesn't actually change anything on the destination server. Reviewing the log file can help you to understand what impact your deployment will have on the destination server, in particular:

- What will get added.
- What will get updated.
- What will get deleted.

Because a "what if" deployment doesn't actually change anything on the destination server, what it can't always do is predict whether a deployment will succeed.

As described in Deploying Web Packages, you can deploy web packages using Web Deploy in two ways—by using the MSDeploy.exe command-line utility directly or by running the *.deploy.cmd* file that the build process generates.

If you're using MSDeploy.exe directly, you can run a "what if" deployment by adding the **–whatif** flag to your command. For example, to evaluate what would happen if you deployed the ContactManager.Mvc.zip package to a staging environment, the MSDeploy command should resemble this:

```
MSDeploy.exe
  -whatif
  -source:package="[path]\ContactManager.Mvc.zip"
  -dest:auto,
        computerName="https://stageweb1:8172/MSDeploy.axd?site=DemoSite",
        username="FABRIKAM\stagingdeployer",
        password="Pa$$w0rd",
        authtype="Basic",
        includeAcls="False"
  -verb:sync
  -disableLink:AppPoolExtension
  -disableLink:ContentExtension
  -disableLink:CertificateExtension
  -setParamFile:"[path]\ContactManager.Mvc.SetParameters.xml"
  -allowUntrusted
```

When you're satisfied with the results of your "what if" deployment, you can remove the **–whatif** flag to run a live deployment.

> **NOTE**
>
> For more information on command-line options for MSDeploy.exe, see Web Deploy Operation Settings.

If you're using the *.deploy.cmd* file, you can run a "what if" deployment by including the **/t** flag (trial mode) flag instead of the **/y** flag ("yes," or update mode) in your command. For example, to evaluate what would happen if you deployed the ContactManager.Mvc.zip package by running the *.deploy.cmd* file, your command should resemble this:

```
ContactManager.Mvc.deploy.cmd /t /m:TESTWEB1 /a:NTLM
```

When you're satisfied with the results of your "trial mode" deployment, you can replace the **/t** flag with a **/y** flag to run a live deployment:

```
ContactManager.Mvc.deploy.cmd /y /m:TESTWEB1 /a:NTLM
```

> **NOTE**
>
> For more information on command-line options for *.deploy.cmd* files, see How to: Install a Deployment Package Using the deploy.cmd File. If you run the *.deploy.cmd* file without specifying any flags, the command prompt will display a list of available flags.

Performing a "What If" Deployment for Databases

This section assumes that you're using the VSDBCMD utility to perform incremental, schema-based database deployment. This approach is described in more detail in Deploying Database Projects. We recommend that you familiarize yourself with this topic before you apply the concepts described here.

When you use VSDBCMD in **Deploy** mode, you can use the **/dd** (or **/DeployToDatabase**) flag to control whether VSDBCMD actually deploys the database or just generates a deployment script. If you're deploying a .dbschema file, this is the behavior:

- If you specify **/dd+** or **/dd**, VSDBCMD will generate a deployment script and deploy the database.
- If you specify **/dd-** or omit the switch, VSDBCMD will generate a deployment script only.

> **NOTE**
> If you're deploying a .deploymanifest file rather than a .dbschema file, the behavior of the **/dd** switch is a lot more complicated. Essentially, VSDBCMD will ignore the value of the **/dd** switch if the .deploymanifest file includes a **DeployToDatabase** element with a value of **True**. Deploying Database Projects describes this behavior in full.

For example, to generate a deployment script for the **ContactManager** database without actually deploying the database, your VSDBCMD command should resemble this:

```
vsdbcmd.exe /a:Deploy
            /manifest:"…\ContactManager.Database.deploymanifest"
            /cs:"Data Source=TESTDB1;Integrated Security=true"
            /p:TargetDatabase=ContactManager
            /dd-
            /script:"…\Publish-ContactManager-Db.sql"
```

VSDBCMD is a differential database deployment tool, and as such the deployment script is dynamically generated to contain all the SQL commands necessary to update the current database, if one exists, to the specified schema. Reviewing the deployment script is a useful way to determine what impact your deployment will have on the current database and the data it contains. For example, you might want to determine:

- Whether any existing tables will be removed, and whether that will result in data loss.
- Whether the order of operations carries a risk of data loss, for example, if you're splitting or merging tables.

If you're happy with the deployment script, you can repeat the VSDBCMD with a **/dd+** flag to make the changes. Alternatively, you can edit the deployment script to meet your requirements and then execute it manually on the database server.

Integrating "What If" Functionality into Custom Project Files

In more complex deployment scenarios, you'll want to use a custom Microsoft Build Engine (MSBuild) project file to encapsulate your build and deployment logic, as described in Understanding the Project File. For example, in the Contact Manager sample solution, the *Publish.proj* file:

- Builds the solution.
- Uses Web Deploy to package and deploy the ContactManager.Mvc application.
- Uses Web Deploy to package and deploy the ContactManager.Service application.
- Deploys the **ContactManager** database.

When you integrate the deployment of multiple web packages and/or databases into a single-step process in this way, you may also want the option of performing the entire deployment in a "what if" mode.

The *Publish.proj* file demonstrates how you can do this. First, you need to create a property to store the "what if" value:

```
<PropertyGroup>
  <WhatIf Condition=" '$(WhatIf)'==''  ">false</WhatIf>
</PropertyGroup>
```

In this case, you've created a property named **WhatIf** with a default value of **false**. Users can override this value by setting the property to **true** in a command-line parameter, as you'll see shortly.

The next stage is to parameterize any Web Deploy and VSDBCMD commands so that the flags reflect the **WhatIf** property value. For example, the next target (taken from the *Publish.proj* file and simplified) runs the *.deploy.cmd* file to deploy a web package. By default, the command includes a **/Y** switch ("yes," or update mode). If **WhatIf** is

set to **true**, this is replaced by a **/T** switch (trial, or "what if" mode).

```
<Target Name="PublishWebPackages" Outputs="%(PublishPackages.Identity)">
  <PropertyGroup>
    <_WhatIfSwitch>/Y</_WhatIfSwitch>
    <_WhatIfSwitch Condition=" '$(WhatIf)'=='true' ">/T</_WhatIfSwitch>
    <_Cmd>%(PublishPackages.FullPath) $(_WhatifSwitch)
         /M:$(MSDeployComputerName)
         /U:$(MSDeployUsername)
         /P:$(MSDeployPassword)
         /A:$(MSDeployAuth)
         %(PublishPackages.AdditionalMSDeployParameters)
    </_Cmd>
  </PropertyGroup>
  <Exec Command="$(_Cmd)"/>
</Target>
```

Similarly, the next target uses the VSDBCMD utility to deploy a database. By default, a **/dd** switch is not included. This means that VSDBCMD will generate a deployment script but will not deploy the database—in other words, a "what if" scenario. If the **WhatIf** property is not set to **true**, a **/dd** switch is added and VSDBCMD will deploy the database.

```
<Target Name="PublishDbPackages" Outputs="%(DbPublishPackages.Identity)">
  <PropertyGroup>
    <_DbDeployOrScript></_DbDeployOrScript>
    <_DbDeployOrScript Condition=" '$(Whatif)'!='true' ">/dd</_DbDeployOrScript>
    <_Cmd>"$(VsdbCmdExe)" /a:Deploy
           /cs:"%(DbPublishPackages.DatabaseConnectionString)"
           /p:TargetDatabase=%(DbPublishPackages.TargetDatabase)
           /manifest:"%(DbPublishPackages.FullPath)"
           /script:"$(_CmDbScriptPath)"
           $(_DbDeployOrScript)
    </_Cmd>
  </PropertyGroup>
  <Exec Command="$(_Cmd)"/>
</Target>
```

You can use the same approach to parameterize all the relevant commands in your project file. When you want to run a "what if" deployment, you can then simply provide a **WhatIf** property value from the command line:

```
MSBuild.exe Publish.proj /p:WhatIf=true;TargetEnvPropsFile=EnvConfig\Env-Dev.proj
```

In this way, you can run a "what if" deployment for all your project components in a single step.

Conclusion

This topic described how to run "what if" deployments using Web Deploy, VSDBCMD, and MSBuild. A "what if" deployment lets you evaluate the impact of a proposed deployment before you actually make any changes to the destination environment.

Further Reading

For more information on Web Deploy command-line syntax, see Web Deploy Operation Settings. For guidance on command-line options when you use the *.deploy.cmd* file, see How to: Install a Deployment Package Using the deploy.cmd File. For guidance on VSDBCMD command-line syntax, see Command-Line Reference for VSDBCMD.EXE (Deployment and Schema Import).

Customizing Database Deployments for Multiple Environments

2/20/2020 • 9 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic describes how to tailor the properties of a database to specific target environments as part of the deployment process.

> **NOTE**
> The topic assumes that you're deploying a Visual Studio 2010 database project using MSBuild.exe and VSDBCMD.exe. For more information on why you might choose this approach, see Web Deployment in the Enterprise and Deploying Database Projects.

When you deploy a database project to multiple destinations, you'll often want to customize the database deployment properties for each target environment. For example, in test environments you'd typically recreate the database on every deployment, whereas in staging or production environments you'd be a lot more likely to make incremental updates to preserve your data.

In a Visual Studio 2010 database project, deployment settings are contained within a deployment configuration (.sqldeployment) file. This topic will show you how to create environment-specific deployment configuration files and specify the one you want to use as a VSDBCMD parameter.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Project File, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Task Overview

This topic assumes that:

- You use the split project file approach to solution deployment, as described in Understanding the Project File.
- You call VSDBCMD from the project file to deploy your database project, as described in Understanding the Build Process.

To create a deployment system that supports varying the database deployment properties between target environments, you'll need to:

- Create a deployment configuration (.sqldeployment) file for each target environment.
- Create a VSDBCMD command that specifies the deployment configuration file as a command-line switch.

- Parameterize the VSDBCMD command in a Microsoft Build Engine (MSBuild) project file, so that the VSDBCMD options are appropriate to the target environment.

This topic will show you how to perform each of these procedures.

Creating Environment-Specific Deployment Configuration Files

By default, a database project contains a single deployment configuration file named *Database.sqldeployment*. If you open this file in Visual Studio 2010, you can see the different deployment options that are available to you:

- **Deployment comparison collation**. This lets you choose whether to use the database collation of your project (the *source* collation) or the database collation of your destination server (the *target* collation). In most cases, you'll want to use the source collation when you deploy to a development or test environment. When you deploy to a staging or production environment, you'll usually want to leave the target collation unchanged to avoid any interoperability issues.
- **Deploy database properties**. This lets you choose whether to apply the database properties, as defined in the *Database.sqlsettings* file. When you deploy a database for the first time, you should deploy the database properties. If you're updating an existing database, the properties should already be in place, and you shouldn't need to deploy them again.
- **Always re-create database**. This lets you choose whether to re-create the target database every time you deploy or make incremental changes to bring the target database up to date with your schema. If you re-create the database, you'll lose any data in the existing database. As such, you should usually set this to **false** for deployments to staging or production environments.
- **Block incremental deployment if data loss might occur**. This lets you choose whether deployment should stop if a change to the database schema will cause the loss of data. You typically set this to **true** for a deployment to a production environment, to give you the opportunity to intervene and protect any important data. If you have set **Always re-create database** to **false**, this setting will have no effect.
- **Execute deployment in single-user mode**. This is not usually an issue in development or test environments. However, you should typically set this to **true** for deployments to staging or production environments. This prevents users from making changes to the database while the deployment is underway.
- **Back up database before deployment**. You typically set this to **true** when you deploy to a production environment, as a precaution against data loss. You may also want to set it to **true** when you deploy to a staging environment, if your staging database contains a lot of data.
- **Generate DROP statements for objects that are in the target database but that are not in the database project**. In most cases, this is an integral and essential part of making incremental changes to a database. If you have set **Always re-create database** to **false**, this setting will have no effect.
- **Do not use ALTER ASSEMBLY statements to update CLR types**. This setting determines how SQL Server should update common language runtime (CLR) types to newer assembly versions. This should be set to **false** in most scenarios.

This table shows typical deployment settings for different destination environments. However, your settings may be different depending on your exact requirements.

	DEVELOPER/TEST	STAGING/INTEGRATION	PRODUCTION
Deployment comparison collation	Source	Target	Target
Deploy database properties	True	First time only	First time only
Always re-create database	True	False	False

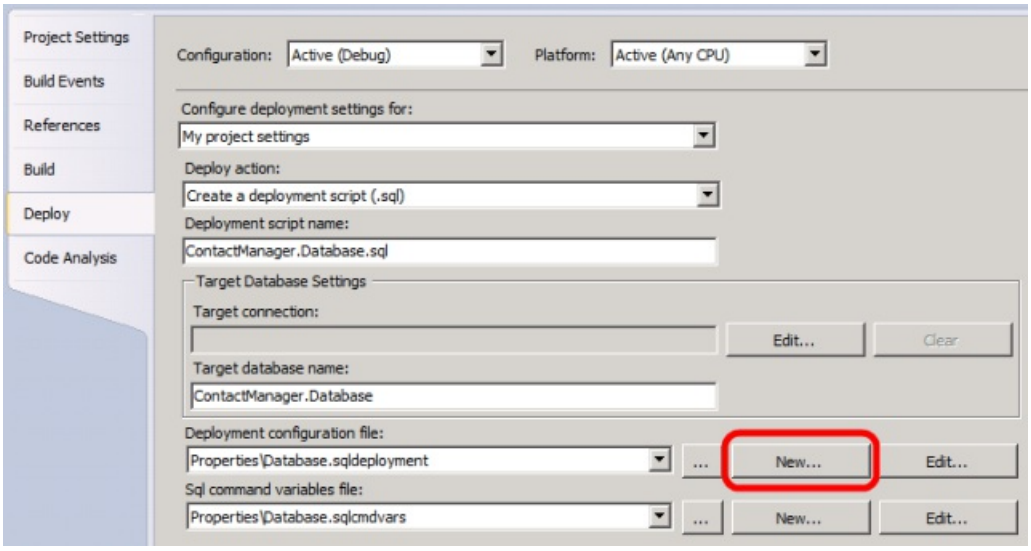
	DEVELOPER/TEST	STAGING/INTEGRATION	PRODUCTION
Block incremental deployment if data loss might occur	False	Maybe	True
Execute deployment script in single-user mode	False	True	True
Back up database before deployment	False	Maybe	True
Generate DROP statements for objects that are in the target database but that are not in the database project	False	True	True
Do not use ALTER ASSEMBLY statements to update CLR types	False	False	False

> **NOTE**
>
> For more information on database deployment properties and environment considerations, see An Overview of Database Project Settings, How to: Configure Properties for Deployment Details, Build and Deploy Database to an Isolated Development Environment, and Build and Deploy Databases to a Staging or Production Environment.

To support the deployment of a database project to multiple destinations, you should create a deployment configuration file for each target environment.

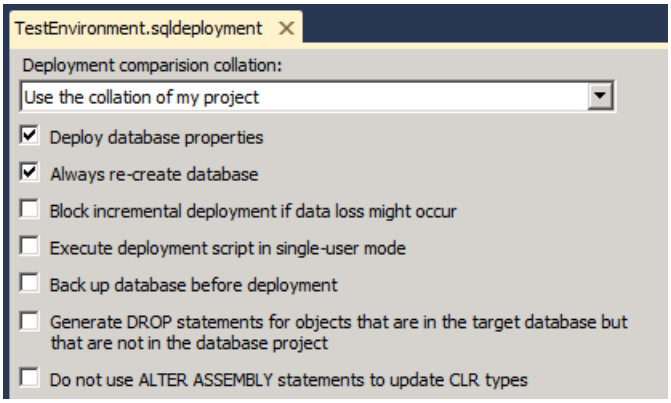
To create an environment-specific configuration file

1. In Visual Studio 2010, in the **Solution Explorer** window, right-click your database project, and then click **Properties**.
2. On the database project properties page, on the **Deploy** tab, in the **Deployment configuration file** row, click **New**.

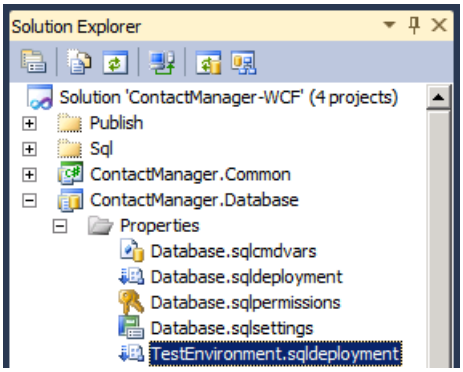


3. In the **New Deployment Configuration File** dialog box, give the file a meaningful name (for example, **TestEnvironment.sqldeployment**), and then click **Save**.

4. On the *[Filename]*.**sqldeployment** page, set the deployment properties to match the requirements of your destination environment, and then save the file.



5. Notice that the new file is added to the Properties folder in your database project.



Specifying the Deployment Configuration File in VSDBCMD

When you use solution configurations (like Debug and Release) within Visual Studio 2010, you can associate a deployment configuration file with each configuration. When you build a particular configuration, the build process generates a configuration-specific deployment manifest file that points to the configuration-specific deployment configuration file. However, one of the main aims of the approach to deployment described in these tutorials is to give people the ability to control the deployment process without using Visual Studio 2010 and solution configurations. In this approach, the solution configuration is the same regardless of the target deployment environment. To tailor your database deployment to a specific destination environment, you can use the VSDBCMD command-line options to specify your deployment configuration file.

To specify a deployment configuration file in your VSDBCMD, use the **p:/DeploymentConfigurationFile** switch and provide the full path to your file. This will override the deployment configuration file that the deployment manifest identifies. For example, you could use this VSDBCMD command to deploy the **ContactManager** database to a test environment:

```
vsdbcmd.exe /a:Deploy
            /manifest:"…\ContactManager.Database.deploymanifest"
            /cs:"Data Source=TESTDB1;Integrated Security=true"
            /p:TargetDatabase=ContactManager
            /p:DeploymentConfigurationFile=
              "…\ContactManager.Database_TestEnvironment.sqldeployment"
            /dd+
            /script:"…\Publish-ContactManager-Db.sql"
```

> **NOTE**
> Note that the build process may rename your .sqldeployment file when it copies the file to the output directory.

If you use SQL command variables in your pre-deployment or post-deployment SQL scripts, you can use a similar approach to associate an environment-specific .sqlcmdvars file with your deployment. In this case, you use the **p:/SqlCommandVariablesFile** switch to identify your .sqlcmdvars file.

Running the VSDBCMD Command from an MSBuild Project File

You can invoke a VSDBCMD command from an MSBuild project file by using an **Exec** task within an MSBuild target. In its simplest form, it would look like this:

```
<Target Name="DeployDatabase">
   <PropertyGroup>
      <_Cmd>
         Add your VSDBCMD command here
      </_Cmd>
   </PropertyGroup>
   <Exec Command="$(_Cmd)"/>
 </Target>
```

- In practice, to make your project files easy to read and reuse, you'll want to create properties to store the various command-line parameters. This makes it easier for users to provide property values in an environment-specific project file or to override default values from the MSBuild command line. If you use the split project file approach described in Understanding the Project File, you should divide your build instructions and properties between the two files accordingly:
- Environment-specific settings, like the deployment configuration filename, the database connection string, and the target database name, should go in the environment-specific project file.
- The MSBuild target that runs the VSDBCMD command, together with any universal properties like the location of the VSDBCMD executable, should go in the universal project file.

You should also ensure that you build the database project before you invoke VSDBCMD so that the .deploymanifest file is created and ready to use. You can see a full example of this approach in the topic Understanding the Build Process, which walks you through the project files in the Contact Manager sample solution.

Conclusion

This topic described how you can tailor database properties to different destination environments when you deploy database projects using MSBuild and VSDBCMD. This approach is useful when you need to deploy database projects as part of larger, enterprise-scale solutions. These solutions are often deployed to multiple destinations, like sandboxed development or test environments, staging or integration platforms, and production or live environments. Each of these target environments typically requires a unique set of database deployment properties.

Further Reading

For more information on deploying database projects using VSDBCMD.exe, see Deploying Database Projects. For more information on using custom MSBuild project files to control the deployment process, see Understanding the Project File and Understanding the Build Process.

These articles on MSDN provide more general guidance on database deployment:

- An Overview of Database Project Settings
- How to: Configure Properties for Deployment Details
- Build and Deploy Databases to an Isolated Development Environment
- Build and Deploy Databases to a Staging or Production Environment

PREVIOUS NEXT

Deploying Database Role Memberships to Test Environments

2/20/2020 • 6 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic describes how to add user accounts to database roles as part of a solution deployment to a test environment.

When you deploy a solution containing a database project to a staging or production environment, you typically don't want the developer to automate the addition of user accounts to database roles. In most cases, the developer won't know which user accounts need to be added to which database roles, and these requirements could change at any time. However, when you deploy a solution containing a database project to a development or test environment, the situation is usually rather different:

- The developer typically re-deploys the solution on a regular basis, often several times a day.
- The database is typically re-created on every deployment, which means that database users must be created and added to roles after every deployment.
- The developer typically has full control over the target development or test environment.

In this scenario, it's often beneficial to automatically create database users and assign database role memberships as part of the deployment process.

The key factor is that this operation needs to be conditional based on the target environment. If you're deploying to a staging or a production environment, you want to skip the operation. If you're deploying to a developer or test environment, you want to deploy role memberships without further intervention. This topic describes one approach you can use to address this challenge.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Project File, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Task Overview

This topic assumes that:

- You use the split project file approach to solution deployment, as described in Understanding the Project File.
- You call VSDBCMD from the project file to deploy your database project, as described in Understanding the Build Process.

To create database users and assign role memberships when you deploy a database project to a test environment, you'll need to:

- Create a Transact Structured Query Language (Transact-SQL) script that makes the necessary database changes.
- Create a Microsoft Build Engine (MSBuild) target that uses the sqlcmd.exe utility to run the SQL script.
- Configure your project files to invoke the target when you're deploying your solution to a test environment.

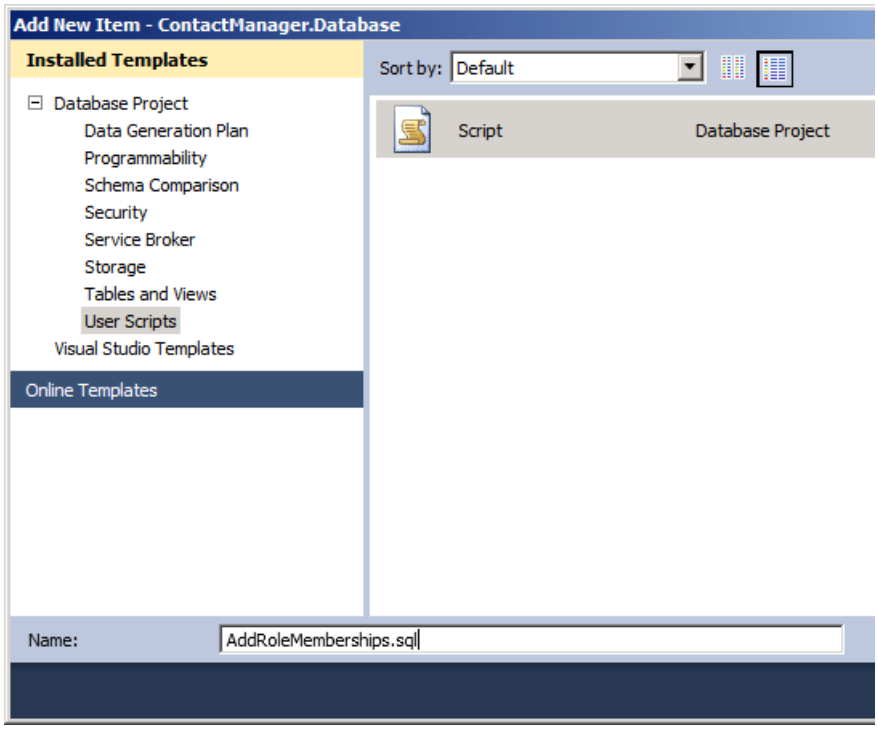
This topic will show you how to perform each of these procedures.

Scripting the Database Role Memberships

You can create a Transact-SQL script in a lot of different ways, and in any location you choose. The easiest approach is to create the script within your solution in Visual Studio 2010.

To create a SQL script

1. In the **Solution Explorer** window, expand your database project node.
2. Right-click the **Scripts** folder, point to **Add**, and then click **New Folder**.
3. Type **Test** as the folder name, and then press Enter.
4. Right-click the **Test** folder, point to **Add**, and then click **Script**.
5. In the **Add New Item** dialog box, give your script a meaningful name (for example, **AddRoleMemberships.sql**), and then click **Add**.



6. In the *AddRoleMemberships.sql* file, add Transact-SQL statements that:
   a. Create a database user for the SQL Server login that will access your database.
   b. Add the database user to any required database roles.
7. The file should resemble this:

```
USE $(DatabaseName)
GO
CREATE USER [FABRIKAM\TESTWEB1$] FOR LOGIN[FABRIKAM\TESTWEB1$]
GO
USE [ContactManager]
GO
EXEC sp_addrolemember N'db_datareader', N'FABRIKAM\TESTWEB1$'
GO
USE [ContactManager]
GO
EXEC sp_addrolemember N'db_datawriter', N'FABRIKAM\TESTWEB1$'
GO
```

8. Save the file.

Executing the Script on the Target Database

Ideally, you'd run any required Transact-SQL scripts as part of a post-deployment script when you deploy your database project. However, post-deployment scripts don't allow you to execute logic conditionally based on solution configurations or build properties. The alternative is to run your SQL scripts directly from the MSBuild project file, by creating a **Target** element that executes a sqlcmd.exe command. You can use this command to run your script on the target database:

```
sqlcmd.exe -S [Database server] -d [Database name] -i [SQL script]
```

> **NOTE**
>
> For more information on sqlcmd command-line options, see sqlcmd Utility.

Before you embed this command in an MSBuild target, you need to consider under what conditions you want the script to run:

- The target database must exist before you change its role memberships. As such, you need to run this script *after* the database deployment.
- You need to include a condition so that the script is only executed for test environments.
- If you're running a "what if" deployment—in other words, if you're generating deployment scripts but not actually running them—you shouldn't run the SQL script.

If you're using the split project file approach described in Understanding the Project File, as demonstrated by the Contact Manager sample solution, you can split the build instructions for your SQL script like this:

- Any required environment-specific properties, together with the property that determines whether to deploy permissions, should go in the environment-specific project file (for example, *Env-Dev.proj*).
- The MSBuild target itself, together with any properties that will not change between destination environments, should go in the universal project file (for example, *Publish.proj*).

In the environment-specific project file, you need to define the database server name, the target database name, and a Boolean property that lets the user specify whether to deploy role memberships.

```
<PropertyGroup>
   <CmTargetDatabase Condition=" '$(CmTargetDatabase)'=='' ">
      ContactManager
   </CmTargetDatabase>
   <DatabaseServer Condition=" '$(DatabaseServer)'=='' ">
      TESTDB1
   </DatabaseServer>
   <DeployTestDBRoleMemberships Condition="'$(DeployTestDBRoleMemberships)'==''">
      true
   </DeployTestDBRoleMemberships>
</PropertyGroup>
```

In the universal project file, you need to provide the location of the sqlcmd executable and the location of the SQL script you want to run. These properties will remain the same regardless of the destination environment. You also need to create an MSBuild target to execute the sqlcmd command.

```
<PropertyGroup>
   <SqlCmdExe Condition=" '$(SqlCmdExe)'=='' ">
      C:\Program Files\Microsoft SQL Server\100\Tools\Binn\sqlcmd.exe
   </SqlCmdExe>
</PropertyGroup>

<Target Name="DeployTestDBPermissions"
        Condition=" '$(DeployTestDBRoleMemberships)'=='true' AND
                    '$(Whatif)'!='true' ">
   <PropertyGroup>
     <SqlScript>
        $(SourceRoot)ContactManager.Database\Scripts\Test\AddRoleMemberships.sql
     </SqlScript>
     <_Cmd>"$(SqlCmdExe)" -S "$(DatabaseServer)"
                          -d "$(CmTargetDatabase)"
                          -i "$(SqlScript)"
     </_Cmd>
   </PropertyGroup>
   <Exec Command="$(_Cmd)" ContinueOnError="false" />
</Target>
```

Notice that you add the location of the sqlcmd executable as a static property, as this could be useful to other targets. In contrast, you define the location of your SQL script and the syntax of the sqlcmd command as dynamic properties within the target, as they will not be required before the target is executed. In this case, the **DeployTestDBPermissions** target will only be executed if these conditions are met:

- The **DeployTestDBRoleMemberships** property is set to **true**.
- The user hasn't specified a **WhatIf=true** flag.

Finally, don't forget to invoke the target. In the *Publish.proj* file, you can do this by adding the target to the dependency list for the default **FullPublish** target. You need to ensure that the **DeployTestDBPermissions** target is not executed until the **PublishDbPackages** target has been executed.

```
<Project ToolsVersion="4.0"
         DefaultTargets="FullPublish"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   ...
   <PropertyGroup>
      <FullPublishDependsOn>
         Clean;
         BuildProjects;
         GatherPackagesForPublishing;
         PublishDbPackages;
         DeployTestDBPermissions;
         PublishWebPackages;
      </FullPublishDependsOn>
   </PropertyGroup>
   <Target Name="FullPublish" DependsOnTargets="$(FullPublishDependsOn)" />
</Project>
```

Conclusion

This topic described one way in which you can add database users and role memberships as a post-deployment action when you deploy a database project. This is typically useful when you regularly re-create a database in a test environment, but it should usually be avoided when you deploy databases to staging or production environments. As such, you should ensure that you use the necessary conditional logic so that database users and role memberships are only created when it's appropriate to do so.

Further Reading

For more information on using VSDBCMD to deploy database projects, see Deploying Database Projects. For guidance on customizing database deployments for different target environments, see Customizing Database Deployments for Multiple Environments. For more information on using custom MSBuild project files to control the deployment process, see Understanding the Project File and Understanding the Build Process. For more information on sqlcmd command-line options, see sqlcmd Utility.

PREVIOUS NEXT

Deploying Membership Databases to Enterprise Environments

2/20/2020 • 4 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic explains the key considerations and challenges you'll need to overcome when you provision ASP.NET application services databases (more commonly referred to as membership databases) in test, staging, or production environments. It also describes approaches you can use to meet these challenges.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Project File, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

What Are the Issues When You Deploy a Membership Database?

In most cases, when you devise a deployment strategy for a database, the first thing you need to consider is what data you want to deploy. In a development or test environment, you might want to deploy user account data to facilitate quick and easy testing. In a staging or production environment, it's very unlikely that you'd want to deploy user account data.

Unfortunately, ASP.NET membership databases introduce some specific challenges that make this decision a lot more complex:

- A schema-only deployment will leave the membership database in a non-operational state. This is because the membership database includes some configuration data (in the **aspnet_SchemaVersions** table) that the database requires in order to function. As such, if you perform a schema-only deployment of your membership database in order to exclude user account data, you'll need to run a post-deployment script to add the essential configuration data.
- Depending on how your membership database is configured, the membership provider may use the machine key to encrypt passwords and store them in the database. In this case, any user account data you deploy with the database will become unusable on the destination server. For this reason, deploying user account data is not a supported scenario.

Choosing a Membership Database Strategy

Use these guidelines when you choose how to provision a membership database in an enterprise server environment:

- Wherever possible, do not deploy membership databases. Instead, create the membership database manually on the target database server. If you haven't customized your membership database schema, you can simply

create a new one in situ at the destination using the ASP.NET SQL Server Registration Tool (aspnet_regsql.exe).

- If you have no option but to deploy a membership database—for example, if you've made extensive modifications to the database schema—you should perform a schema-only deployment of the membership database, to exclude user account data, and then run a post-deployment script to add any required configuration data. You can find broad guidance on these approaches in How to: Deploy the ASP.NET Membership Database Without Including User Accounts.

It's important to remember that *the schema of your membership database is likely to be fairly static*. Even if you've customized the membership database, it's unlikely that you'll need to update the schema on a regular basis—it's not going to change with the same frequency as the code in a web application or a database project. As such, you shouldn't need to include the membership database in any automated or single-step deployment processes.

Using VSDBCMD to Update a Membership Database Schema

If you modify the structure of your membership database after your first deployment, you may not want to use the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) to redeploy the database. The database deployment functionality in Web Deploy doesn't include the capability to make differential updates to a destination database—instead, Web Deploy must drop and re-create the database. This means that you lose any existing user account data, which is typically undesirable in staging or production environments.

The alternative is to use the VSDBCMD utility to update the schema of your destination database. VSDBCMD includes two important capabilities. First, it can import the schema of an existing database into a .dbschema file. Second, it can deploy a .dbschema file to an existing database as a differential update, which means that it only makes the changes required to bring the target database up to date and you don't lose any data.

You can use these high-level steps to update a membership database schema:

1. Use the VSDBCMD **Import** action to generate a .dbschema file for your source membership database. This procedure is described in How to: Import a Schema from a Command Prompt.
2. Use the VSDBCMD **Deploy** action to deploy the .dbschema file to your destination membership database. This procedure is described in Command-Line Reference for VSDBCMD.EXE (Deployment and Schema Import).

Conclusion

This topic described some of the challenges you may face when you need to provision ASP.NET membership databases in various target environments. In particular, it explained why schema-only deployments will leave the membership database in a non-operational state and why deploying user account data is not supported. The topic also presented guidance on how to provision, deploy, and update membership databases in different scenarios.

Further Reading

For more guidance and examples of how to use VSDBCMD, see Command-Line Reference for VSDBCMD.EXE (Deployment and Schema Import) and How to: Import a Schema from a Command Prompt. For more information on using aspnet_regsql.exe to create membership databases, see ASP.NET SQL Server Registration Tool (aspnet_regsql.exe). For more general guidance on deploying membership databases, see How to: Deploy the ASP.NET Membership Database Without Including User Accounts.

Excluding Files and Folders from Deployment

2/20/2020 • 6 minutes to read • Edit Online

by Jason Lee

Download PDF

> This topic describes how you can exclude files and folders from a web deployment package when you build and package a web application project.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Project File, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

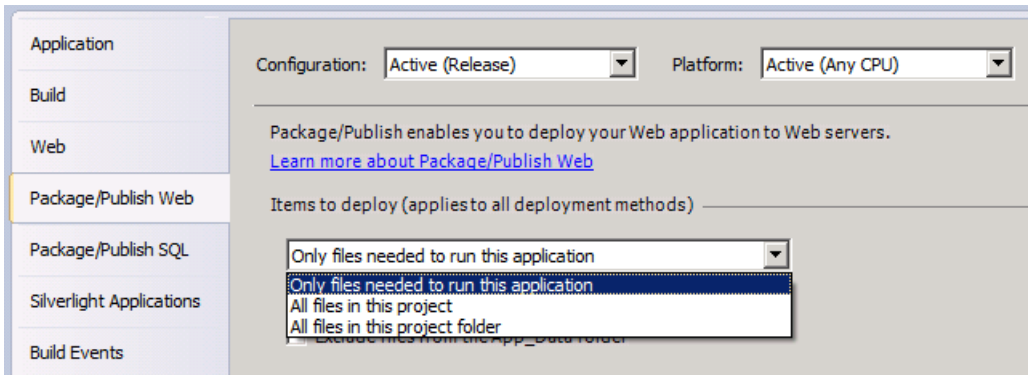
Overview

When you build a web application project in Visual Studio 2010, the Web Publishing Pipeline (WPP) lets you extend this build process by packaging your compiled web application into a deployable web package. You can then use the Internet Information Services (IIS) Web Deployment Tool (Web Deploy) to deploy this web package to a remote IIS web server, or import the web package manually through IIS Manager. This packaging process is explained in Building and Packaging Web Application Projects.

So how do you control what gets included in your web package? The project settings in Visual Studio, through the underlying project file, provide sufficient control for a lot of scenarios. However, in some cases you may want to tailor the contents of your web package to specific destination environments. For example, you might want to include a folder for log files when you deploy your application to a test environment but exclude the folder when you deploy the application to a staging or production environment. This topic will show you how to do this.

What Gets Included by Default?

When you configure your web application project properties in Visual Studio, the **Items to deploy** list on the **Package/Publish Web** page lets you specify what you want to include in your web deployment package. By default, this is set to **Only files needed to run this application**.



When you choose **Only files needed to run this application**, the WPP will try to determine which files should be added to the web package. This includes:

- All the build outputs for the project.
- Any files marked with a build action of **Content**.

> **NOTE**
> The logic that determines which files to include is contained in this file:
> *%PROGRAMFILES%\MSBuild\Microsoft\VisualStudio\v10.0\Web\ Microsoft.Web.Publishing.OnlyFilesToRunTheApp.targets*

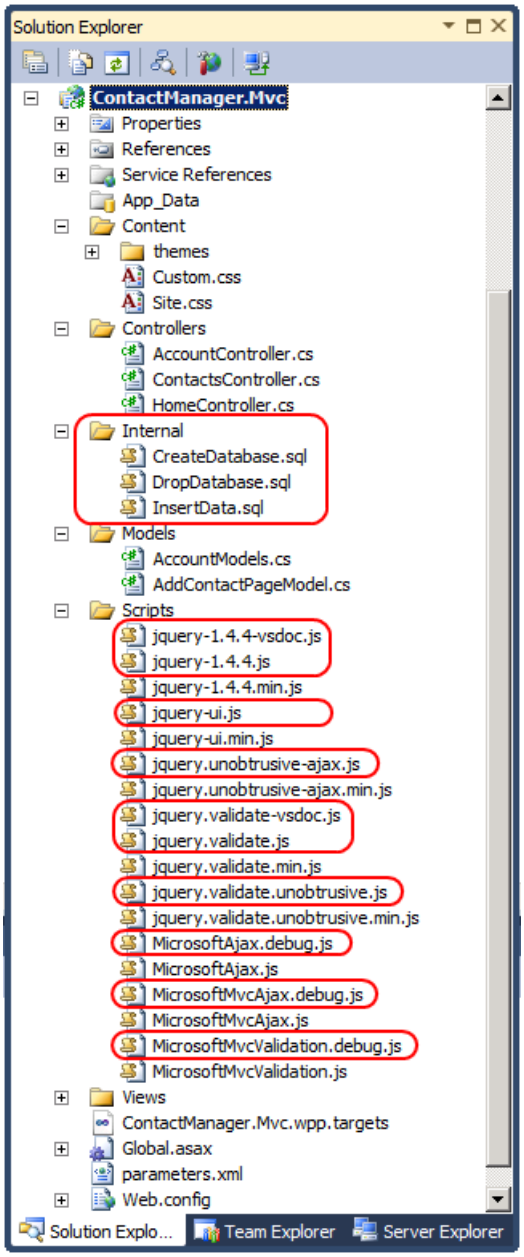
Excluding Specific Files and Folders

In some cases, you'll want more fine-grained control over which files and folders are deployed. If you know which files you want to exclude ahead of time, and the exclusion applies to all destination environments, you can simply set the **Build Action** of each file to **None**.

To exclude specific files from deployment

1. In the **Solution Explorer** window, right-click the file, and then click **Properties**.
2. In the **Properties** window, in the **Build Action** row, select **None**.

However, this approach is not always convenient. For example, you may want to vary which files and folders are included according to your destination environment, and from outside Visual Studio. For example, in the Contact Manager sample solution, take a look at the contents of the ContactManager.Mvc project:



- The Internal folder contains some SQL scripts that the developer uses to create, drop, and populate local databases for development purposes. Nothing in this folder should be deployed to a staging or production environment.
- The Scripts folder contains several JavaScript files. A lot of these files are included purely to support debugging or provide IntelliSense in Visual Studio. Some of these files should not be deployed to staging or production environments. However, you may want to deploy them to a developer test environment to facilitate troubleshooting.

Although you could manipulate your project files to exclude specific files and folders, there is an easier way. The WPP includes a mechanism to exclude files and folders by building item lists named **ExcludeFromPackageFolders** and **ExcludeFromPackageFiles**. You can extend this mechanism by adding your own items to these lists. To do this, you need to complete these high-level steps:

1. Create a custom project file named *[project name].wpp.targets* in the same folder as your project file.

> **NOTE**
>
> The *.wpp.targets* file needs to go in the same folder as your web application project file—for example, *ContactManager.Mvc.csproj*—rather than in the same folder as any custom project files you use to control the build and deployment process.

2. In the *.wpp.targets* file, add an **ItemGroup** element.

3. In the **ItemGroup** element, add **ExcludeFromPackageFolders** and **ExcludeFromPackageFiles** items to exclude specific files and folders as required.

This is the basic structure of this *.wpp.targets* file:

```
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <ExcludeFromPackageFolders Include="[semi-colon-separated folder list]">
      <FromTarget>[arbitrary metadata value]</FromTarget>
    </ExcludeFromPackageFolders>
    <ExcludeFromPackageFiles Include="[semi-colon-separated file list]">
      <FromTarget>[arbitrary metadata value]</FromTarget>
    </ExcludeFromPackageFiles>
  </ItemGroup>
</Project>
```

Note that each item includes an item metadata element named **FromTarget**. This is an optional value that doesn't affect the build process; it simply serves to indicate why particular files or folders were omitted if someone reviews the build logs.

Excluding Files and Folders from a Web Package

The next procedure shows you how to add a *.wpp.targets* file to a web application project and how to use the file to exclude specific files and folders from the web package when you build your project.

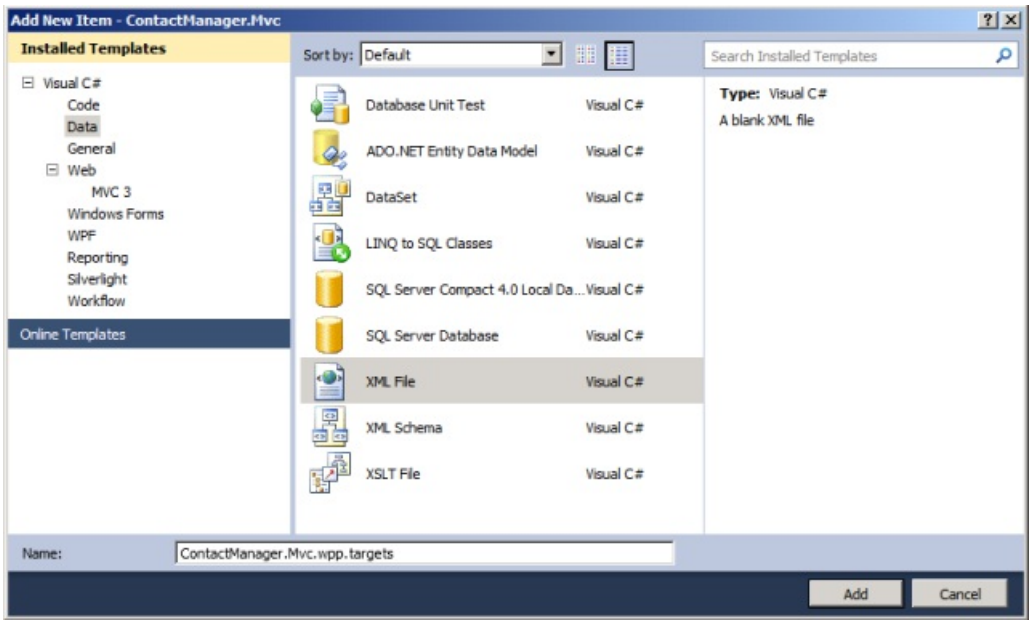
To exclude files and folders from a web deployment package

1. Open your solution in Visual Studio 2010.

2. In the **Solution Explorer** window, right-click your web application project node (for example, **ContactManager.Mvc**), point to **Add**, and then click **New Item**.

3. In the **Add New Item** dialog box, select the **XML File** template.

4. In the **Name** box, type *[project name]*.**wpp.targets** (for example, **ContactManager.Mvc.wpp.targets**), and then click **Add**.



> **NOTE**
> If you add a new item to the root node of a project, the file is created in the same folder as the project file. You can verify this by opening the folder in Windows Explorer.

5. In the file, add a **Project** element and an **ItemGroup** element:

```
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
  </ItemGroup>
</Project>
```

6. If you want to exclude folders from the web package, add an **ExcludeFromPackageFolders** element to the **ItemGroup** element:

   a. In the **Include** attribute, provide a semicolon-separated list of the folders you want to exclude.

   b. In the **FromTarget** metadata element, provide a meaningful value to indicate why the folders are being excluded, like the name of the *.wpp.targets* file.

```
<ExcludeFromPackageFolders Include="Internal">
  <FromTarget>ContactManager.Mvc.wpp.targets</FromTarget>
</ExcludeFromPackageFolders>
```

7. If you want to exclude files from the web package, add an **ExcludeFromPackageFiles** element to the **ItemGroup** element:

   a. In the **Include** attribute, provide a semicolon-separated list of the files you want to exclude.

   b. In the **FromTarget** metadata element, provide a meaningful value to indicate why the files are being excluded, like the name of the *.wpp.targets* file.

```
<ExcludeFromPackageFiles Include="Scripts\jquery-1.4.4-vsdoc.js;Scripts\jquery-
1.4.4.js;Scripts\jquery-ui.js;Scripts\jquery.unobtrusive-ajax.js;Scripts\jquery.validate-
vsdoc.js;Scripts\jquery.validate.js;Scripts\jquery.validate.unobtrusive.js;Scripts\MicrosoftAja
x.debug.js;Scripts\MicrosoftMvcValidation.debug.js">
  <FromTarget>ContactManager.Mvc.wpp.targets</FromTarget>
</ExcludeFromPackageFiles>
```

8. The *[project name].wpp.targets* file should now resemble this:

```
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <ExcludeFromPackageFolders Include="Internal">
      <FromTarget>ContactManager.Mvc.wpp.targets</FromTarget>
    </ExcludeFromPackageFolders>
    <ExcludeFromPackageFiles Include="Scripts\jquery-1.4.4-
vsdoc.js;Scripts\jquery-1.4.4.js;Scripts\jquery-ui.js;Scripts\jquery.unobtrusive-
ajax.js;Scripts\jquery.validate-
vsdoc.js;Scripts\jquery.validate.js;Scripts\jquery.validate.unobtrusive.js;Scripts\MicrosoftAjax.debug
.js;Scripts\MicrosoftMvcValidation.debug.js">
      <FromTarget>ContactManager.Mvc.wpp.targets</FromTarget>
    </ExcludeFromPackageFiles>
  </ItemGroup>
</Project>
```

9. Save and close the *[project name].wpp.targets* file.

The next time you build and package your web application project, the WPP will automatically detect the *.wpp.targets* file. Any files and folders you specified will not be included in the web package.

Conclusion

This topic described how to exclude specific files and folders when you build a web package, by creating a custom *.wpp.targets* file in the same folder as your web application project file.

Further Reading

For more information on using custom Microsoft Build Engine (MSBuild) project files to control the deployment process, see Understanding the Project File and Understanding the Build Process. For more information on the packaging and deployment process, see Building and Packaging Web Application Projects, Configuring Parameters for Web Package Deployment, and Deploying Web Packages.

PREVIOUS NEXT

Taking Web Applications Offline with Web Deploy

2/20/2020 • 10 minutes to read • Edit Online

by Jason Lee

Download PDF

> This topic describes how to take a web application offline for the duration of an automated deployment using the Internet Information Services (IIS) Web Deployment Tool (Web Deploy). Users who browse to the web application are redirected to an *App_offline.htm* file until the deployment is complete.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Project File, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Task Overview

In a lot of scenarios, you'll want to take a web application offline while you make changes to related components, like databases or web services. Typically, in IIS and ASP.NET, you accomplish this by placing a file named *App_offline.htm* in the root folder of the IIS website or web application. The *App_offline.htm* file is a standard HTML file and will usually contain a simple message advising the user that the site is temporarily unavailable due to maintenance. While the *App_offline.htm* file exists in the root folder of the website, IIS will automatically redirect any requests to the file. When you've finished making updates, you remove the *App_offline.htm* file and the website resumes serving requests as usual.

When you use Web Deploy to perform automated or single-step deployments to a target environment, you may want to incorporate adding and removing the *App_offline.htm* file into your deployment process. To do this, you'll need to complete these high-level tasks:

- In the Microsoft Build Engine (MSBuild) project file that you use to control the deployment process, create an MSBuild target that copies an *App_offline.htm* file to the destination server before any deployment tasks begin.
- Add another MSBuild target that removes the *App_offline.htm* file from the destination server when all deployment tasks are complete.
- In the web application project, create a *.wpp.targets* file that ensures that an *App_offline.htm* file is added to the deployment package when Web Deploy is invoked.

This topic will show you how to perform these procedures. The tasks and walkthroughs in this topic assume that you've already created a solution that contains at least one web application project, and that you use a custom project file to control the deployment process as described in Web Deployment in the Enterprise. Alternatively, you can use the Contact Manager sample solution to follow the examples in the topic.

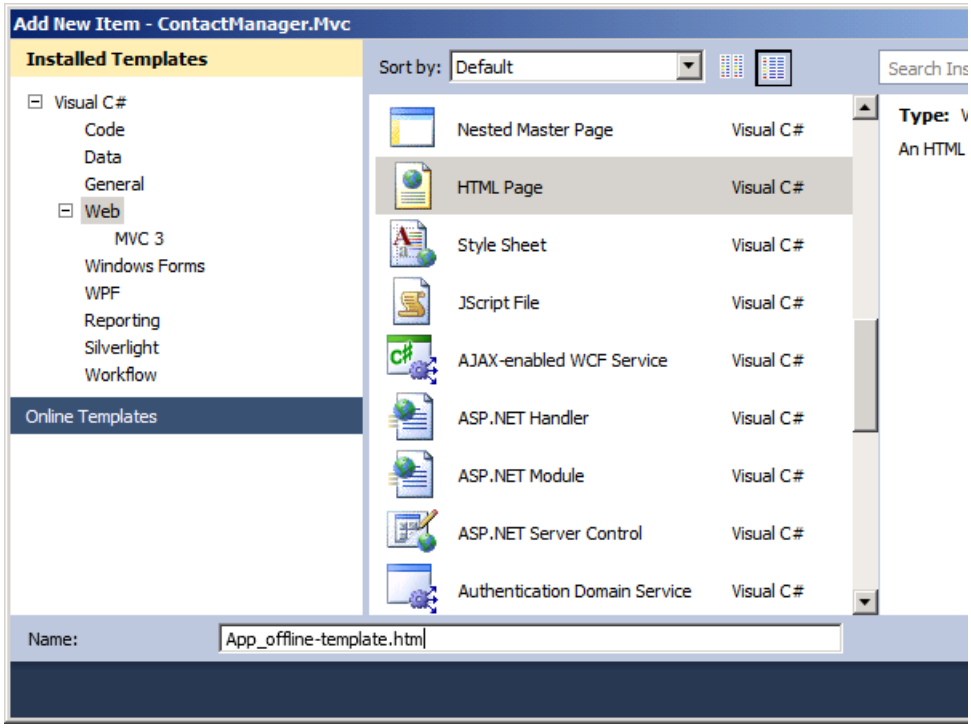
Adding an App_Offline File to a Web Application Project

The first task you need to complete is to add an *App_offline* file to your web application project:

- To prevent the file from interfering with the development process (you don't want your application to be permanently offline), you should call it something other than *App_offline.htm*. For example, you could name the file *App_offline-template.htm*.
- To prevent the file from being deployed as-is, you should set the build action to **None**.

To add an App_offline file to a web application project

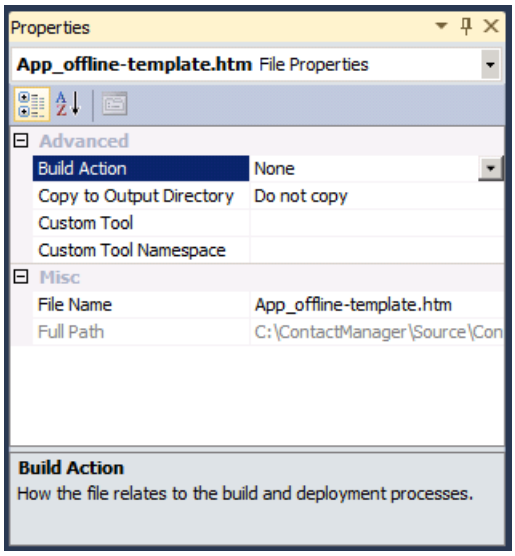
1. Open your solution in Visual Studio 2010.
2. In the **Solution Explorer** window, right-click your web application project, point to **Add**, and then click **New Item**.
3. In the **Add New Item** dialog box, select **HTML Page**.
4. In the **Name** box, type **App_offline-template.htm**, and then click **Add**.



5. Add some simple HTML to inform users that the application is unavailable, and then save the file. Do not include any server-side tags (for example, any tags that are prefixed with "asp:").

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Application Offline</title>
</head>
<body>
    <p>This web application is temporarily offline for maintenance.</p>
</body>
</html>
```

6. In the **Solution Explorer** window, right-click the new file, and then click **Properties**.
7. In the **Properties** window, in the **Build Action** row, select **None**.



Deploying and Deleting an App_Offline File

The next step is to modify your deployment logic to copy the file to the destination server at the start of the deployment process and remove it at the end.

> **NOTE**
>
> The next procedure assumes that you're using a custom MSBuild project file to control your deployment process, as described in Understanding the Project File. If you're deploying direct from Visual Studio, you'll need to use a different approach. Sayed Ibrahim Hashimi describes one such approach in How to Take Your Web App Offline During Publishing.

To deploy an *App_offline* file to a destination IIS website, you need to invoke MSDeploy.exe using the Web Deploy **contentPath** provider. The **contentPath** provider supports both physical directory paths and IIS website or application paths, which makes it the ideal choice for synchronizing a file between a Visual Studio project folder and an IIS web application. To deploy the file, your MSDeploy command should resemble this:

```
msdeploy.exe -verb:sync
             -source:contentPath="[Project folder]\App_offline.template.htm"
             -dest:contentPath="[IIS application path]/App_offline.htm",
              computerName="[Destination web server]"
```

To remove the file from the destination site at the end of the deployment process, your MSDeploy command should resemble this:

```
msdeploy.exe -verb:delete
             -dest:contentPath="[IIS application path]/App_offline.htm",
              computerName="[Destination web server]"
```

To automate these commands as part of a build and deployment process, you need to integrate them into your custom MSBuild project file. The next procedure describes how to do this.

To deploy and delete an App_offline file

1. In Visual Studio 2010, open the MSBuild project file that controls your deployment process. In the Contact Manager sample solution, this is the *Publish.proj* file.

2. In the root **Project** element, create a new **PropertyGroup** element to store variables for the *App_offline* deployment:

```
<PropertyGroup>
  <AppOfflineTemplateFilename
    Condition=" '$(AppOfflineTemplateFilename)'==''  ">
      app_offline-template.htm
  </AppOfflineTemplateFilename>
  <AppOfflineSourcePath
    Condition=" '$(AppOfflineSourcePath)'==''">
      $(SourceRoot)ContactManager.Mvc\$(AppOfflineTemplateFilename)
  </AppOfflineSourcePath>
</PropertyGroup>
```

3. The **SourceRoot** property is defined elsewhere in the *Publish.proj* file. It indicates the location of the root folder for the source content relative to the current path—in other words, relative to the location of the *Publish.proj* file.

4. The **contentPath** provider will not accept relative file paths, so you need to get an absolute path to your source file before you can deploy it. You can use the ConvertToAbsolutePath task to do this.

5. Add a new **Target** element named **GetAppOfflineAbsolutePath**. Within this target, use the **ConvertToAbsolutePath** task to get an absolute path to the *App_offline-template* file in your project folder.

```
<Target Name="GetAppOfflineAbsolutePath" BeforeTargets="DeployAppOffline">
  <ConvertToAbsolutePath Paths="$(AppOfflineSourcePath)">
    <Output TaskParameter="AbsolutePaths"
            PropertyName="AppOfflineAbsoluteSourcePath" />
  </ConvertToAbsolutePath>
</Target>
```

6. This target takes the relative path to the *App_offline-template* file in your project folder and saves it to a new property as an absolute file path. The **BeforeTargets** attribute specifies that you want this target to execute before the **DeployAppOffline** target, which you'll create in the next step.

7. Add a new target named **DeployAppOffline**. Within this target, invoke the MSDeploy.exe command that deploys your *App_offline* file to the destination web server.

```
<Target Name="DeployAppOffline"
        Condition=" '$(EnableAppOffline'!='false' ">
  <PropertyGroup>
    <_Cmd>"$(MSDeployPath)\msdeploy.exe" -verb:sync
           -source:contentPath="$(AppOfflineAbsoluteSourcePath)"
           -dest:contentPath="$(ContactManagerIisPath)/App_offline.htm",
            computerName="$(MSDeployComputerName)"
    </_Cmd>
  </PropertyGroup>
  <Exec Command="$(_Cmd)"/>
</Target>
```

8. In this example, the **ContactManagerIisPath** property is defined elsewhere in the project file. This is simply an IIS application path, in the form *[IIS Website Name]/[Application Name]*. Including a condition in the target enables users to switch the *App_offline* deployment on or off by changing a property value or providing a command-line parameter.

9. Add a new target named **DeleteAppOffline**. Within this target, invoke the MSDeploy.exe command that removes your *App_offline* file from the destination web server.

```
<Target Name="DeleteAppOffline"
        Condition=" '$(EnableAppOffline'!='false' ">
  <PropertyGroup>
    <_Cmd>"$(MSDeployPath)\msdeploy.exe" -verb:delete
           -dest:contentPath="$(ContactManagerIisPath)/App_offline.htm",
            computerName="$(MSDeployComputerName)"
    </_Cmd>
  </PropertyGroup>
  <Exec Command="$(_Cmd)"/>
</Target>
```

10. The final task is to invoke these new targets at appropriate points during the execution of your project file. You can do this in various ways. For example, in the *Publish.proj* file, the **FullPublishDependsOn** property specifies a list of targets that must be executed in order when the **FullPublish** default target is invoked.

11. Modify your MSBuild project file to invoke the **DeployAppOffline** and **DeleteAppOffline** targets at appropriate points in the publishing process.

```
<PropertyGroup>
  <FullPublishDependsOn>
    Clean;
    BuildProjects;
    DeployAppOffline;
    GatherPackagesForPublishing;
    PublishDbPackages;
    DeployTestDBPermissions;
    PublishWebPackages;
    DeleteAppOffline;
  </FullPublishDependsOn>
</PropertyGroup>
<Target Name="FullPublish" DependsOnTargets="$(FullPublishDependsOn)" />
```

When you run your custom MSBuild project file, the *App_offline* file will be deployed to the server immediately after a successful build. It will then be deleted from the server once all the deployment tasks are complete.

Adding an App_Offline File to Deployment Packages

Depending on how you configure your deployment, any existing content at the destination IIS web application—like the *App_offline.htm* file—may be deleted automatically when you deploy a web package to the destination. To ensure that the *App_offline.htm* file remains in place for the duration of the deployment, you need to include the file within the web deployment package itself in addition to deploying the file directly at the start of the deployment process.

- If you've followed the previous tasks in this topic, you'll have added the *App_offline.htm* file to your web application project under a different filename (we used *App_offline-template.htm*) and you'll have set the build action to **None**. These changes are necessary to prevent the file from interfering with development and debugging. As a result, you need to customize the packaging process to ensure that the *App_offline.htm* file is included in the web deployment package.

The Web Publishing Pipeline (WPP) uses an item list named **FilesForPackagingFromProject** to build a list of files that should be included in the web deployment package. You can customize the contents of your web packages by adding your own items to this list. To do this, you need to complete these high-level steps:

1. Create a custom project file named *[project name].wpp.targets* in the same folder as your project file.

> **NOTE**
> The *.wpp.targets* file needs to go in the same folder as your web application project file—for example, *ContactManager.Mvc.csproj*—rather than in the same folder as any custom project files you use to control the build and deployment process.

2. In the *.wpp.targets* file, create a new MSBuild target that executes *before* the **CopyAllFilesToSingleFolderForPackage** target. This is the WPP target that builds a list of things to include in the package.

3. In the new target, create an **ItemGroup** element.

4. In the **ItemGroup** element, add a **FilesForPackagingFromProject** item and specify the *App_offline.htm* file.

The *.wpp.targets* file should resemble this:

```
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="AddAppOfflineToPackage"
          BeforeTargets="CopyAllFilesToSingleFolderForPackage">
    <ItemGroup>
      <FilesForPackagingFromProject Include="App_offline-template.htm">
        <DestinationRelativePath>App_offline.htm</DestinationRelativePath>
    </FilesForPackagingFromProject>
  </ItemGroup>
  </Target>
</Project>
```

These are the key points of note in this example:

- The **BeforeTargets** attribute inserts this target into the WPP by specifying that it should be executed immediately before the **CopyAllFilesToSingleFolderForPackage** target.
- The **FilesForPackagingFromProject** item uses the **DestinationRelativePath** metadata value to rename the file from *App_offline-template.htm* to *App_offline.htm* as it's added to the list.

The next procedure shows you how to add this *.wpp.targets* file to a web application project.

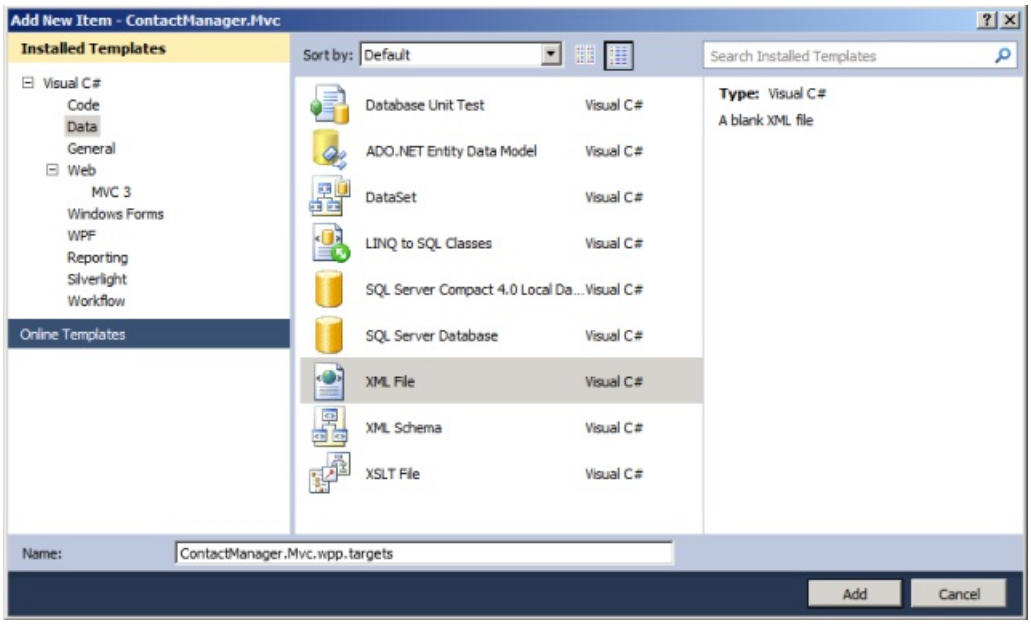
To add a .wpp.targets file to a web deployment package

1. Open your solution in Visual Studio 2010.

2. In the **Solution Explorer** window, right-click your web application project node (for example, **ContactManager.Mvc**), point to **Add**, and then click **New Item**.

3. In the **Add New Item** dialog box, select the **XML File** template.

4. In the **Name** box, type *[project name]*.**wpp.targets** (for example, **ContactManager.Mvc.wpp.targets**), and then click **Add**.



> **NOTE**
>
> If you add a new item to the root node of a project, the file is created in the same folder as the project file. You can verify this by opening the folder in Windows Explorer.

5. In the file, add the MSBuild markup described previously.

```
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="AddAppOfflineToPackage"
          BeforeTargets="CopyAllFilesToSingleFolderForPackage">
    <ItemGroup>
      <FilesForPackagingFromProject Include="App_offline-template.htm">
        <DestinationRelativePath>App_offline.htm</DestinationRelativePath>
    </FilesForPackagingFromProject>
  </ItemGroup>
  </Target>
</Project>
```

6. Save and close the *[project name].wpp.targets* file.

The next time you build and package your web application project, the WPP will automatically detect the *.wpp.targets* file. The *App_offline-template.htm* file will be included in the resulting web deployment package as *App_offline.htm*.

> **NOTE**
>
> If your deployment fails, the *App_offline.htm* file will remain in place and your application will remain offline. This is typically the desired behavior. To bring your application back online, you can delete the *App_offline.htm* file from your web server. Alternatively, if you correct any errors and run a successful deployment, the *App_offline.htm* file will be removed.

Conclusion

This topic described how to take a web application offline for the duration of a deployment, by publishing an *App_offline.htm* file to the destination server at the start of the deployment process and removing it at the end. It also covered how to include an *App_offline.htm* file in a web deployment package.

Further Reading

For more information on the packaging and deployment process, see Building and Packaging Web Application Projects, Configuring Parameters for Web Package Deployment, and Deploying Web Packages.

If you publish your web applications directly from Visual Studio, rather than using the custom MSBuild project file approach described in these tutorials, you'll need to use a slightly different approach to take your application offline during the publishing process.

PREVIOUS NEXT

Running Windows PowerShell Scripts from MSBuild Project Files

2/20/2020 • 8 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic describes how to run a Windows PowerShell script as part of a build and deployment process. You can run a script locally (in other words, on the build server) or remotely, like on a destination web server or database server.

There are lots of reasons why you might want to run a post-deployment Windows PowerShell script. For example, you might want to:

- Add a custom event source to the registry.
- Generate a file system directory for uploads.
- Clean up build directories.
- Write entries to a custom log file.
- Send emails inviting users to a newly provisioned web application.
- Create user accounts with the appropriate permissions.
- Configure replication between SQL Server instances.

This topic will show you how to run Windows PowerShell scripts both locally and remotely from a custom target in a Microsoft Build Engine (MSBuild) project file.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Project File, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Task Overview

To run a Windows PowerShell script as part of an automated or single-step deployment process, you'll need to complete these high-level tasks:

- Add the Windows PowerShell script to your solution and to source control.
- Create a command that invokes your Windows PowerShell script.
- Escape any reserved XML characters in your command.
- Create a target in your custom MSBuild project file and use the **Exec** task to run your command.

This topic will show you how to perform these procedures. The tasks and walkthroughs in this topic assume that you're already familiar with MSBuild targets and properties, and that you understand how to use a custom MSBuild project file to drive a build and deployment process. For more information, see Understanding the

Project File and Understanding the Build Process.

Creating and Adding Windows PowerShell Scripts

The tasks in this topic use a sample Windows PowerShell script named **LogDeploy.ps1** to illustrate how to run scripts from MSBuild. The **LogDeploy.ps1** script contains a simple function that writes a single-line entry to a log file:

```
function LogDeployment
{
  param([string]$filepath,[string]$deployDestination)
  $datetime = Get-Date
  $filetext = "Deployed package to " + $deployDestination + " on " + $datetime
  $filetext | Out-File -filepath $filepath -Append
}

LogDeployment $args[0] $args[1]
```

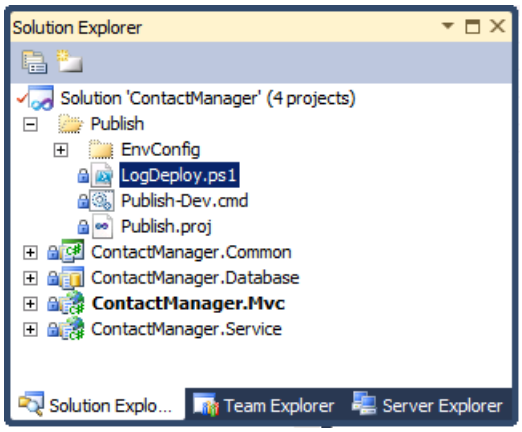
The **LogDeploy.ps1** script accepts two parameters. The first parameter represents the full path to the log file to which you want to add an entry, and the second parameter represents the deployment destination that you want to record in the log file. When you run the script, it adds a line to the log file in this format:

```
Deployed package to TESTWEB1 on 02/11/2012 09:28:18
```

To make the **LogDeploy.ps1** script available to MSBuild, you need to:

- Add the script to source control.
- Add the script to your solution in Visual Studio 2010.

You don't need to deploy the script with your solution content, regardless of whether you plan to run the script on the build server or on a remote computer. One option is to add the script to a solution folder. In the Contact Manager example, because you want to use the Windows PowerShell script as part of the deployment process, it makes sense to add the script to the Publish solution folder.



The contents of solution folders are copied to build servers as source material. However, they form no part of any project output.

Executing a Windows PowerShell Script on the Build Server

In some scenarios, you may want to run Windows PowerShell scripts on the computer that builds your projects. For example, you might use a Windows PowerShell script to clean up build folders or write entries to a custom log file.

In terms of syntax, running a Windows PowerShell script from an MSBuild project file is the same as running a

Windows PowerShell script from a regular command prompt. You need to invoke the powershell.exe executable and use the **–command** switch to provide the commands you want Windows PowerShell to run. (In Windows PowerShell v2, you can also use the **–file** switch). The command should take this format:

```
powershell.exe –command "& { [Path to script] 'parameter1' 'parameter2' ... }"
```

For example:

```
powershell.exe –command
  "& { C:\LogDeploy.ps1 'C:\DeployLogs\log.txt' 'TESTWEB1' }"
```

If the path to your script includes spaces, you need to enclose the file path in single quotes preceded by an ampersand. You can't use double quotes, because you've already used them to enclose the command:

```
powershell.exe –command
  "& { &'C:\Path With Spaces\LogDeploy.ps1'
       'C:\Path With Spaces\log.txt'
       'TESTWEB1' }"
```

There are a few additional considerations when you invoke this command from MSBuild. First, you should include the **–NonInteractive** flag to ensure that the script executes quietly. Next, you should include the **–ExecutionPolicy** flag with an appropriate argument value. This specifies the execution policy that Windows PowerShell will apply to your script and allows you to override the default execution policy, which may prevent execution of your script. You can choose from these argument values:

- A value of **Unrestricted** will allow Windows PowerShell to execute your script, regardless of whether the script is signed.
- A value of **RemoteSigned** will allow Windows PowerShell to execute unsigned scripts that were created on the local machine. However, scripts that were created elsewhere must be signed. (In practice, you're very unlikely to have created a Windows PowerShell script locally on a build server).
- A value of **AllSigned** will allow Windows PowerShell to execute signed scripts only.

The default execution policy is **Restricted**, which prevents Windows PowerShell from running any script files.

Finally, you need to escape any reserved XML characters that occur in your Windows PowerShell command:

- Replace single quotes with **'**
- Replace double quotes with **"**
- Replace ampersands with **&**
- When you make these changes, your command will resemble this:

```
powershell.exe –NonInteractive –ExecutionPolicy Unrestricted
               –command &quot;&amp; { &amp;&apos;[Path to script]&apos;
                        &apos;[parameter1]&apos;
                        &apos;[parameter2]&apos; } &quot;
```

Within your custom MSBuild project file, you can create a new target and use the **Exec** task to run this command:

```
<Target Name="WriteLogEntry" Condition=" '$(WriteLogEntry)'!='false' ">
  <PropertyGroup>
    <PowerShellExe Condition=" '$(PowerShellExe)'==''  ">
      %WINDIR%\System32\WindowsPowerShell\v1.0\powershell.exe
    </PowerShellExe>
    <ScriptLocation Condition=" '$(ScriptLocation)'==''  ">
      C:\Path With Spaces\LogDeploy.ps1
    </ScriptLocation>
    <LogFileLocation Condition=" '$(LogFileLocation)'==''  ">
      C:\Path With Spaces\ContactManagerDeployLog.txt
    </LogFileLocation>
  </PropertyGroup>
  <Exec Command="$(PowerShellExe) -NonInteractive -executionpolicy Unrestricted
                -command &quot;&amp; {
                        &amp;&apos;$(ScriptLocation)&apos;
                        &apos;$(LogFileLocation)&apos;
                        &apos;$(MSDeployComputerName)&apos;} &quot;" />
</Target>
```

In this example, note that:

- Any variables, like parameter values and the location of the Windows PowerShell executable, are declared as MSBuild properties.
- Conditions are included to enable users to override these values from the command line.
- The **MSDeployComputerName** property is declared elsewhere in the project file.

When you execute this target as part of your build process, Windows PowerShell will run your command and write a log entry to the file you specified.

Executing a Windows PowerShell Script on a Remote Computer

Windows PowerShell is capable of running scripts on remote computers through Windows Remote Management (WinRM). To do this, you need to use the Invoke-Command cmdlet. This lets you execute your script against one or more remote computers without copying the script to the remote computers. Any results are returned to the local computer from which you ran the script.

> **NOTE**
> Before you use the **Invoke-Command** cmdlet to execute Windows PowerShell scripts on a remote computer, you need to configure a WinRM listener to accept remote messages. You can do this by running the command **winrm quickconfig** on the remote computer. For more information, see Installation and Configuration for Windows Remote Management.

From a Windows PowerShell window, you'd use this syntax to run the **LogDeploy.ps1** script on a remote computer:

```
Invoke-Command –ComputerName 'REMOTESERVER1'
               –ScriptBlock { &"C:\Path With Spaces\LogDeploy.ps1"
                              'C:\Path With Spaces\Log.txt'
                              'TESTWEB1' }
```

> **NOTE**
> There are various other ways of using **Invoke-Command** to run a script file, but this approach is the most straightforward when you need to provide parameter values and manage paths with spaces.

When you run this from a command prompt, you need to invoke the Windows PowerShell executable and use the

–command parameter to provide your instructions:

```
powershell.exe –command
  "& {Invoke-Command –ComputerName 'REMOTESERVER1'
                     –ScriptBlock { &'C:\Path With Spaces\LogDeploy.ps1'
                                    'C:\Path With Spaces\Log.txt'
                                    'TESTWEB1' } "
```

As before, you need to provide some additional switches and escape any reserved XML characters when you run the command from MSBuild:

```
powershell.exe -NonInteractive -executionpolicy Unrestricted
                -command &quot;&amp; Invoke-Command
                  –ComputerName &apos;REMOTESERVER1&apos;
                  -ScriptBlock { &amp;&apos;C:\Path With Spaces\LogDeploy.ps1&apos;
                                 &apos; C:\Path With Spaces\Log.txt &apos;
                                 &apos;TESTWEB1&apos; } &quot;
```

Finally, as before, you can use the **Exec** task within a custom MSBuild target to execute your command:

```
<Target Name="WriteLogEntry" Condition=" '$(WriteLogEntry)'!='false' ">
  <PropertyGroup>
    <PowerShellExe Condition=" '$(PowerShellExe)'=='' ">
      %WINDIR%\System32\WindowsPowerShell\v1.0\powershell.exe
    </PowerShellExe>
    <ScriptLocation Condition=" '$(ScriptLocation)'=='' ">
      C:\Path With Spaces\LogDeploy.ps1
    </ScriptLocation>
    <LogFileLocation Condition=" '$(LogFileLocation)'=='' ">
      C:\Path With Spaces\ContactManagerDeployLog.txt
    </LogFileLocation>
  </PropertyGroup>
  <Exec Command="$(PowerShellExe) -NonInteractive -executionpolicy Unrestricted
                -command &quot;&amp; invoke-command -scriptblock {
                         &amp;&apos;$(ScriptLocation)&apos;
                         &apos;$(LogFileLocation)&apos;
                         &apos;$(MSDeployComputerName)&apos;}
                         &quot;"/>
</Target>
```

When you execute this target as part of your build process, Windows PowerShell will run your script on the computer you specified in the **–computername** argument.

Conclusion

This topic described how to run a Windows PowerShell script from an MSBuild project file. You can use this approach to run a Windows PowerShell script, either locally or on a remote computer, as part of an automated or single-step build and deployment process.

Further Reading

For guidance on signing Windows PowerShell scripts and managing execution policies, see Running Windows PowerShell Scripts. For guidance on running Windows PowerShell commands from a remote computer, see Running Remote Commands.

For more information on using custom MSBuild project files to control the deployment process, see Understanding the Project File and Understanding the Build Process.

Troubleshooting the Packaging Process

2/20/2020 • 5 minutes to read • Edit Online

by Jason Lee

Download PDF

This topic describes how you can collect detailed information about the packaging process by using the **EnablePackageProcessLoggingAndAssert** property in the Microsoft Build Engine (MSBuild).

When you set the **EnablePackageProcessLoggingAndAssert** property to **true**, MSBuild will:

- Add additional information about the packaging process to the build logs.
- Log errors under certain conditions, for example, if duplicate files are found in the packaging list.
- Create a Log directory in the *ProjectName*_Package folder and use it to record information about the files you're packaging.

If the packaging process is failing, or your web deployment packages don't contain the files that you expect, you can use this information to troubleshoot the process and pinpoint where things are going wrong.

> **NOTE**
>
> The **EnablePackageProcessLoggingAndAssert** property only works if you build your project using the **Debug** configuration. The property is ignored in other configurations.

This topic forms part of a series of tutorials based around the enterprise deployment requirements of a fictional company named Fabrikam, Inc. This tutorial series uses a sample solution—the Contact Manager solution—to represent a web application with a realistic level of complexity, including an ASP.NET MVC 3 application, a Windows Communication Foundation (WCF) service, and a database project.

The deployment method at the heart of these tutorials is based on the split project file approach described in Understanding the Project File, in which the build process is controlled by two project files—one containing build instructions that apply to every destination environment, and one containing environment-specific build and deployment settings. At build time, the environment-specific project file is merged into the environment-agnostic project file to form a complete set of build instructions.

Understanding the EnablePackageProcessLoggingAndAssert Property

Building and Packaging Web Application Projects described how the Web Publishing Pipeline (WPP) provides a set of MSBuild targets that extend the functionality of MSBuild and enable it to integrate with the Internet Information Services (IIS) Web Deployment Tool (Web Deploy). When you package a web application project, you're invoking WPP targets.

Lots of these WPP targets include conditional logic that logs additional information when the **EnablePackageProcessLoggingAndAssert** property is set to **true**. For example, if you review the **Package** target, you can see that it creates an additional log directory and writes a list of files to a text file if **EnablePackageProcessLoggingAndAssert** is equal to **true**.

```
<Target Name="Package"
        Condition="$(_CreatePackage)"
        DependsOnTargets="$(PackageDependsOn)">

    <!--Log the information  Set $(EnablePackageProcessLoggingAndAssert) to True
                 if you want to see this information-->
    <MakeDir Condition="$(EnablePackageProcessLoggingAndAssert) And
                        !Exists('$(PackageLogDir)')"
             Directories="$(PackageLogDir)" />
    <WriteLinesToFile Condition="$(EnablePackageProcessLoggingAndAssert)"
                      Encoding="utf-8"
                      File="$(PackageLogDir)\Prepackage.txt"
                      Lines="@(FilesForPackagingFromProject->'
                             From:%(Identity)
                             DestinationRelativePath:%(DestinationRelativePath)
                             Exclude:%(Exclude)
                             FromTarget:%(FromTarget)
                             Category:%(Category)
                             ProjectFileType:%(ProjectFileType)')"
                      Overwrite="True" />
```

NOTE

The WPP targets are defined in the *Microsoft.Web.Publishing.targets* file in the %PROGRAMFILES(x86)%\MSBuild\Microsoft\VisualStudio\v10.0\Web folder. You can open this file and review the targets in Visual Studio 2010 or any XML editor. Take care not to modify the contents of the file.

Enabling the Additional Logging

You can supply a value for the **EnablePackageProcessLoggingAndAssert** property in various ways, depending on how you build your project.

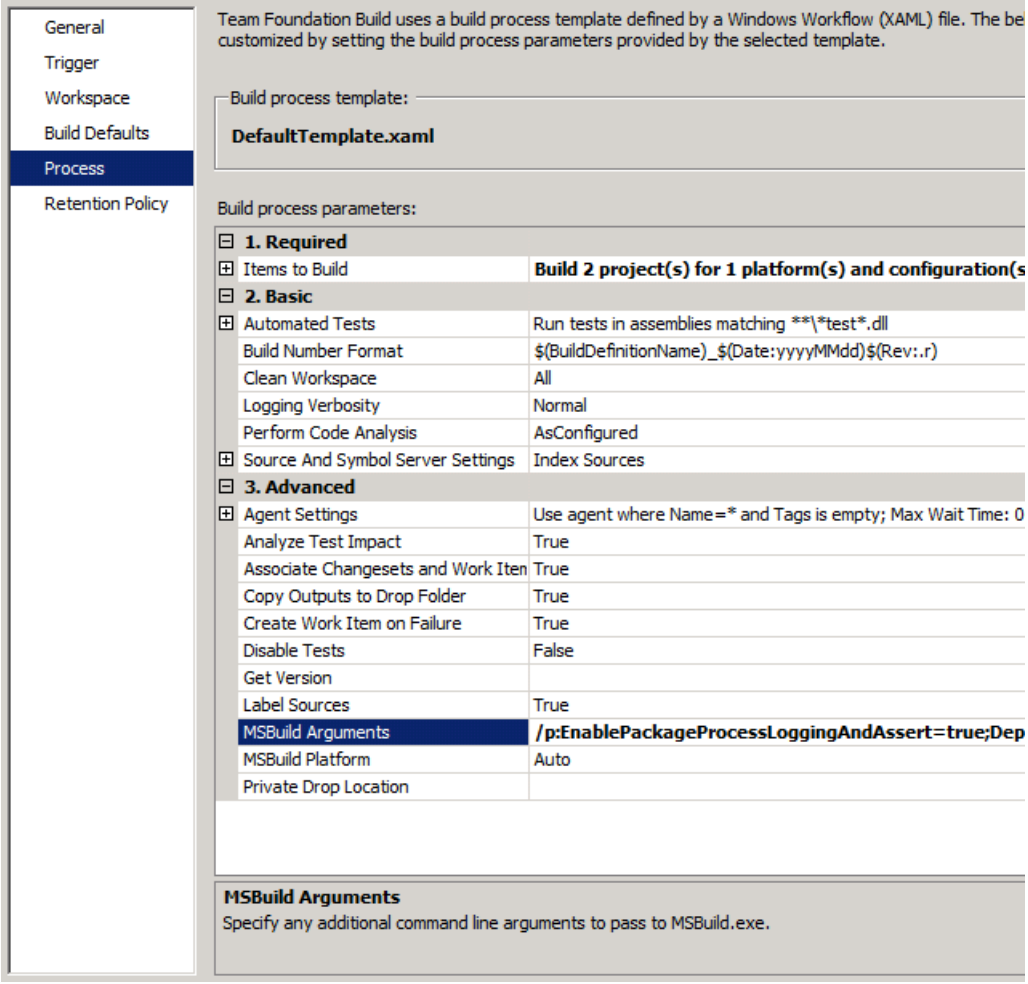
If you build your project from the command line, you can supply a value for the **EnablePackageProcessLoggingAndAssert** property as a command-line argument:

```
MSBuild.exe /t:Build
            /p:Configuration=DEBUG
            /p:DeployOnBuild=true
            /p:DeployTarget=Package
            /p:EnablePackageProcessLoggingAndAssert=true
            [Your project].csproj
```

If you're using a custom project file to build your projects, you can include the **EnablePackageProcessLoggingAndAssert** value in the **Properties** attribute of the **MSBuild** task:

```
<Target Name="BuildProjects" Condition=" '$(BuildingInTeamBuild)'!='true' ">
   <MSBuild Projects="@(ProjectsToBuild)"
            Properties="OutDir=$(OutputRoot);
                        Configuration=$(Configuration);
                        DeployOnBuild=true;
                        DeployTarget=Package;
                        EnablePackageProcessLoggingAndAssert=true"
            Targets="Build" />
  </Target>
```

If you're using a Team Foundation Server (TFS) build definition to build your projects, you can supply a value for the **EnablePackageProcessLoggingAndAssert** property in the **MSBuild Arguments** row:



> **NOTE**
>
> For more information on creating and configuring build definitions, see Creating a Build Definition That Supports Deployment.

Alternatively, if you want to include the package in every build, you can modify the project file for your web application project to set the **EnablePackageProcessLoggingAndAssert** property to **true**. You should add the property to the first **PropertyGroup** element within your .csproj or .vbproj file.

```
<Project ToolsVersion="4.0" DefaultTargets="Build" xmlns="...">
  <PropertyGroup>
    <EnablePackageProcessLoggingAndAssert
       Condition=" '$(EnablePackageProcessLoggingAndAssert)' == '' ">
         true
    </EnablePackageProcessLoggingAndAssert>
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
```

Reviewing the Log Files

When you build and package a web application project with **EnablePackageProcessLoggingAndAssert** set to **true**, MSBuild creates an additional folder named Log in the *ProjectName*_Package folder. The Log folder contains various files:

AfterExcludeFilesFilesList.txt	17/02/2012 16:24	Text Document	19 KB
AfterTransformWebConfig.txt	17/02/2012 16:24	Text Document	19 KB
PackageUsingManifest.parameters.xml	17/02/2012 16:24	XML Document	2 KB
PostAutoParameterizationWebConfigConnectionStrings.txt	17/02/2012 16:24	Text Document	19 KB
PreAutoParameterizationWebConfigConnectionStrings.Log	17/02/2012 16:24	Text Document	1 KB
PreExcludePipelineCollectFilesPhaseFileList.txt	17/02/2012 16:24	Text Document	19 KB
Prepackage.txt	17/02/2012 16:24	Text Document	21 KB
PreTransformWebConfig.Log	17/02/2012 16:24	Text Document	1 KB

The list of files that you see will vary according to the things in your project and your build process. However, these files are typically used to record the list of files that the WPP is collecting for packaging, at various stages of the process:

- The *PreExcludePipelineCollectFilesPhaseFileList.txt* file lists the files that MSBuild collects for packaging before any files that are specified for exclusion are removed.
- The *AfterExcludeFilesFilesList.txt* file contains the modified file list after any files that are specified for exclusion are removed.

  > **NOTE**
  >
  > For more information on excluding files and folders from the packaging process, see Excluding Files and Folders from Deployment.

- The *AfterTransformWebConfig.txt* file lists the files collected for packaging after any *Web.config* transforms have been performed. In this list, any configuration-specific *Web.config* transform files, like *Web.Debug.config* and *Web.Release.config*, are excluded from the list of files for packaging. A single transformed *Web.config* is included in their place.
- The *PostAutoParameterizationWebConfigConnectionStrings.txt* file contains the list of files after the connection strings in the *Web.config* file have been parameterized. This is the process that lets you replace your connection strings with the right settings for your target environment when you deploy the package.
- The *Prepackage.txt* file contains the finalized pre-build list of files to be included in the package.

> **NOTE**
>
> The names of the additional log files typically correspond to WPP targets. You can review these targets by examining the *Microsoft.Web.Publishing.targets* file in the %PROGRAMFILES(x86)%\MSBuild\Microsoft\VisualStudio\v10.0\Web folder.

If the contents of your web package aren't what you expected, reviewing these files can be a useful way to identify at what point in the process things went wrong.

Conclusion

This topic described how you can use the **EnablePackageProcessLoggingAndAssert** property in MSBuild to troubleshoot the packaging process. It explained the different ways in which you can supply the property value to the build process, and it described the additional information that is recorded when you set the property to **true**.

Further Reading

For more information on using custom MSBuild project files to control the deployment process, see Understanding the Project File and Understanding the Build Process. For more information on the WPP and how it manages the packaging process, see Building and Packaging Web Application Projects. For guidance on how to exclude specific files and folders from web deployment packages, see Excluding Files and Folders from

Deployment.

PREVIOUS

ASP.NET Web Forms Advanced Topics

2/20/2020 • 2 minutes to read • Edit Online

Advanced topics for programming ASP.NET Web Forms applications, including Internationalization, accessibility, and extensibility.

- ASP.NET Web Forms Connection Resiliency and Command Interception

ASP.NET Web Forms Connection Resiliency and Command Interception

2/20/2020 • 16 minutes to read • Edit Online

by Erik Reitan

In this tutorial, you will modify the Wingtip Toys sample application to support connection resiliency and command interception. By enabling connection resiliency, the Wingtip Toys sample application will automatically retry data calls when transient errors typical of a cloud environment occur. Also, by implementing command interception, the Wingtip Toys sample application will catch all SQL queries sent to the database in order to log or change them.

> **NOTE**
>
> This Web Forms tutorial was based on Tom Dykstra's following MVC tutorial:
> Connection Resiliency and Command Interception with the Entity Framework in an ASP.NET MVC Application

What you'll learn:

- How to provide connection resiliency.
- How to implement command interception.

Prerequisites

Before you start, make sure that you have the following software installed on your computer:

- Microsoft Visual Studio 2013 or Microsoft Visual Studio Express 2013 for Web. The .NET Framework is installed automatically.
- The Wingtip Toys sample project, so that you can implement the functionality mentioned in this tutorial within the Wingtip Toys project. The following link provides download details:
  - Getting Started with ASP.NET 4.5.1 Web Forms - Wingtip Toys (C#)
- Prior to completing this tutorial, consider reviewing the related tutorial series, Getting Started with ASP.NET 4.5 Web Forms and Visual Studio 2013. The tutorial series will help you become familiar with the **WingtipToys** project and code.

Connection resiliency

When you consider deploying an application to Windows Azure, one option to consider is deploying the database to **Windows Azure SQL Database**, a cloud database service. Transient connection errors are typically more frequent when you connect to a cloud database service than when your web server and your database server are directly connected together in the same data center. Even if a cloud web server and a cloud database service are hosted in the same data center, there are more network connections between them that can have problems, such as load balancers.

Also a cloud service is typically shared by other users, which means its responsiveness can be affected by them. And your access to the database might be subject to throttling. Throttling means the database service throws exceptions when you try to access it more frequently than is allowed in your *Service Level Agreement* (SLA).

Many or most connection problems that occur when you're accessing a cloud service are transient, that is, they resolve themselves in a short period of time. So when you try a database operation and get a type of error that is typically transient, you could try the operation again after a short wait, and the operation might be successful. You can provide a much better experience for your users if you handle transient errors by automatically trying again, making most of them invisible to the customer. The connection resiliency feature in Entity Framework 6 automates that process of retrying failed SQL queries.

The connection resiliency feature must be configured appropriately for a particular database service:

1. It has to know which exceptions are likely to be transient. You want to retry errors caused by a temporary loss in network connectivity, not errors caused by program bugs, for example.
2. It has to wait an appropriate amount of time between retries of a failed operation. You can wait longer between retries for a batch process than you can for an online web page where a user is waiting for a response.
3. It has to retry an appropriate number of times before it gives up. You might want to retry more times in a batch process that you would in an online application.

You can configure these settings manually for any database environment supported by an Entity Framework provider.

All you have to do to enable connection resiliency is create a class in your assembly that derives from the `DbConfiguration` class, and in that class set the SQL Database execution strategy, which in Entity Framework is another term for retry policy.

Implementing connection resiliency

1. Download and open the WingtipToys sample Web Forms application in Visual Studio.

2. In the *Logic* folder of the **WingtipToys** application, add a class file named *WingtipToysConfiguration.cs*.

3. Replace the existing code with the following code:

```
using System.Data.Entity;
using System.Data.Entity.SqlServer;

namespace WingtipToys.Logic
{
    public class WingtipToysConfiguration : DbConfiguration
    {
        public WingtipToysConfiguration()
        {
          SetExecutionStrategy("System.Data.SqlClient", () => new SqlAzureExecutionStrategy());
        }
    }
}
```

The Entity Framework automatically runs the code it finds in a class that derives from `DbConfiguration`. You can use the `DbConfiguration` class to do configuration tasks in code that you would otherwise do in the *Web.config* file. For more information, see EntityFramework Code-Based Configuration.

1. In the *Logic* folder, open the *AddProducts.cs* file.

2. Add a `using` statement for `System.Data.Entity.Infrastructure` as shown highlighted in yellow:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using WingtipToys.Models;
using System.Data.Entity.Infrastructure;
```

3. Add a `catch` block to the `AddProduct` method so that the `RetryLimitExceededException` is logged as highlighted in yellow:

```
public bool AddProduct(string ProductName, string ProductDesc, string ProductPrice, string
ProductCategory, string ProductImagePath)
{
    var myProduct = new Product();
    myProduct.ProductName = ProductName;
    myProduct.Description = ProductDesc;
    myProduct.UnitPrice = Convert.ToDouble(ProductPrice);
    myProduct.ImagePath = ProductImagePath;
    myProduct.CategoryID = Convert.ToInt32(ProductCategory);

    using (ProductContext _db = new ProductContext())
    {
        // Add product to DB.
        _db.Products.Add(myProduct);
        try
        {
            _db.SaveChanges();
        }
        catch (RetryLimitExceededException ex)
        {
            // Log the RetryLimitExceededException.
            WingtipToys.Logic.ExceptionUtility.LogException(ex, "Error: RetryLimitExceededException ->
RemoveProductButton_Click in AdminPage.aspx.cs");
        }
    }
    // Success.
    return true;
}
```

By adding the `RetryLimitExceededException` exception, you can provide better logging or display an error message to the user where they can choose to try the process again. By catching the `RetryLimitExceededException` exception, the only errors likely to be transient will already have been tried and failed several times. The actual exception returned will be wrapped in the `RetryLimitExceededException` exception. In addition, you also added a general catch block. For more information about the `RetryLimitExceededException` exception, see Entity Framework Connection Resiliency / Retry Logic.

Command interception

Now that you've turned on a retry policy, how do you test to verify that it is working as expected? It's not so easy to force a transient error to happen, especially when you're running locally, and it would be especially difficult to integrate actual transient errors into an automated unit test. To test the connection resiliency feature, you need a way to intercept queries that Entity Framework sends to SQL Server and replace the SQL Server response with an exception type that is typically transient.

You can also use query interception in order to implement a best practice for cloud applications: log the latency and success or failure of all calls to external services such as database services.

In this section of the tutorial you'll use the Entity Framework's *interception feature* both for logging and for simulating transient errors.

Create a logging interface and class

A best practice for logging is to do it by using an `interface` rather than hard-coding calls to `System.Diagnostics.Trace` or a logging class. That makes it easier to change your logging mechanism later if you ever need to do that. So in this section, you'll create the logging interface and a class to implement it.

Based on the above procedure, you have downloaded and opened the **WingtipToys** sample application in Visual

Studio.

1. Create a folder in the **WingtipToys** project and name it *Logging*.

2. In the *Logging* folder, create a class file named *ILogger.cs* and replace the default code with the following code:

```
using System;

namespace WingtipToys.Logging
{
    public interface ILogger
    {
        void Information(string message);
        void Information(string fmt, params object[] vars);
        void Information(Exception exception, string fmt, params object[] vars);

        void Warning(string message);
        void Warning(string fmt, params object[] vars);
        void Warning(Exception exception, string fmt, params object[] vars);

        void Error(string message);
        void Error(string fmt, params object[] vars);
        void Error(Exception exception, string fmt, params object[] vars);

        void TraceApi(string componentName, string method, TimeSpan timespan);
        void TraceApi(string componentName, string method, TimeSpan timespan, string properties);
        void TraceApi(string componentName, string method, TimeSpan timespan, string fmt, params
object[] vars);

    }
}
```

   The interface provides three tracing levels to indicate the relative importance of logs, and one designed to provide latency information for external service calls such as database queries. The logging methods have overloads that let you pass in an exception. This is so that exception information including stack trace and inner exceptions is reliably logged by the class that implements the interface, instead of relying on that being done in each logging method call throughout the application.

   The `TraceApi` methods enable you to track the latency of each call to an external service such as SQL Database.

3. In the *Logging* folder, create a class file named *Logger.cs* and replace the default code with the following code:

```
using System;
using System.Diagnostics;
using System.Text;

namespace WingtipToys.Logging
{
  public class Logger : ILogger
  {

    public void Information(string message)
    {
      Trace.TraceInformation(message);
    }

    public void Information(string fmt, params object[] vars)
    {
      Trace.TraceInformation(fmt, vars);
    }
```

```
    public void Information(Exception exception, string fmt, params object[] vars)
    {
      Trace.TraceInformation(FormatExceptionMessage(exception, fmt, vars));
    }

    public void Warning(string message)
    {
      Trace.TraceWarning(message);
    }

    public void Warning(string fmt, params object[] vars)
    {
      Trace.TraceWarning(fmt, vars);
    }

    public void Warning(Exception exception, string fmt, params object[] vars)
    {
      Trace.TraceWarning(FormatExceptionMessage(exception, fmt, vars));
    }

    public void Error(string message)
    {
      Trace.TraceError(message);
    }

    public void Error(string fmt, params object[] vars)
    {
      Trace.TraceError(fmt, vars);
    }

    public void Error(Exception exception, string fmt, params object[] vars)
    {
      Trace.TraceError(FormatExceptionMessage(exception, fmt, vars));
    }

    public void TraceApi(string componentName, string method, TimeSpan timespan)
    {
      TraceApi(componentName, method, timespan, "");
    }

    public void TraceApi(string componentName, string method, TimeSpan timespan, string fmt, params
object[] vars)
    {
      TraceApi(componentName, method, timespan, string.Format(fmt, vars));
    }
    public void TraceApi(string componentName, string method, TimeSpan timespan, string properties)
    {
      string message = String.Concat("Component:", componentName, ";Method:", method, ";Timespan:",
timespan.ToString(), ";Properties:", properties);
      Trace.TraceInformation(message);
    }

    private static string FormatExceptionMessage(Exception exception, string fmt, object[] vars)
    {
      var sb = new StringBuilder();
      sb.Append(string.Format(fmt, vars));
      sb.Append(" Exception: ");
      sb.Append(exception.ToString());
      return sb.ToString();
    }
  }
}
```

The implementation uses `System.Diagnostics` to do the tracing. This is a built-in feature of .NET which makes it easy to generate and use tracing information. There are many "listeners" you can use with `System.Diagnostics` tracing, to write logs to files, for example, or to write them to blob storage in Windows Azure. See some of the

options, and links to other resources for more information, in Troubleshooting Windows Azure Web Sites in Visual Studio. For this tutorial, you'll only look at logs in the Visual Studio **Output** window.

In a production application you might want to consider using tracing frameworks other than `System.Diagnostics`, and the `ILogger` interface makes it relatively easy to switch to a different tracing mechanism if you decide to do that.

Create interceptor classes

Next, you'll create the classes that the Entity Framework will call into every time it is going to send a query to the database, one to simulate transient errors and one to do logging. These interceptor classes must derive from the `DbCommandInterceptor` class. In them, you write method overrides that are automatically called when the query is about to be executed. In these methods you can examine or log the query that is being sent to the database, and you can change the query before it's sent to the database or return something to Entity Framework yourself without even passing the query to the database.

1. To create the interceptor class that will log every SQL query before it is sent to the database, create a class file named *InterceptorLogging.cs* in the *Logic* folder and replace the default code with the following code:

```
using System;
using System.Data.Common;
using System.Data.Entity;
using System.Data.Entity.Infrastructure.Interception;
using System.Data.Entity.SqlServer;
using System.Data.SqlClient;
using System.Diagnostics;
using System.Reflection;
using System.Linq;
using WingtipToys.Logging;

namespace WingtipToys.Logic
{
  public class InterceptorLogging : DbCommandInterceptor
  {
    private ILogger _logger = new Logger();
    private readonly Stopwatch _stopwatch = new Stopwatch();

    public override void ScalarExecuting(DbCommand command, DbCommandInterceptionContext<object>
interceptionContext)
    {
      base.ScalarExecuting(command, interceptionContext);
      _stopwatch.Restart();
    }

    public override void ScalarExecuted(DbCommand command, DbCommandInterceptionContext<object>
interceptionContext)
    {
      _stopwatch.Stop();
      if (interceptionContext.Exception != null)
      {
        _logger.Error(interceptionContext.Exception, "Error executing command: {0}",
command.CommandText);
      }
      else
      {
        _logger.TraceApi("SQL Database", "Interceptor.ScalarExecuted", _stopwatch.Elapsed, "Command:
{0}: ", command.CommandText);
      }
      base.ScalarExecuted(command, interceptionContext);
    }

    public override void NonQueryExecuting(DbCommand command, DbCommandInterceptionContext<int>
interceptionContext)
    {
      base.NonQueryExecuting(command, interceptionContext);
```

```
      _stopwatch.Restart();
    }

    public override void NonQueryExecuted(DbCommand command, DbCommandInterceptionContext<int>
interceptionContext)
    {
      _stopwatch.Stop();
      if (interceptionContext.Exception != null)
      {
        _logger.Error(interceptionContext.Exception, "Error executing command: {0}",
command.CommandText);
      }
      else
      {
        _logger.TraceApi("SQL Database", "Interceptor.NonQueryExecuted", _stopwatch.Elapsed, "Command:
{0}: ", command.CommandText);
      }
      base.NonQueryExecuted(command, interceptionContext);
    }

    public override void ReaderExecuting(DbCommand command, DbCommandInterceptionContext<DbDataReader>
interceptionContext)
    {
      base.ReaderExecuting(command, interceptionContext);
      _stopwatch.Restart();
    }
    public override void ReaderExecuted(DbCommand command, DbCommandInterceptionContext<DbDataReader>
interceptionContext)
    {
      _stopwatch.Stop();
      if (interceptionContext.Exception != null)
      {
        _logger.Error(interceptionContext.Exception, "Error executing command: {0}",
command.CommandText);
      }
      else
      {
        _logger.TraceApi("SQL Database", "Interceptor.ReaderExecuted", _stopwatch.Elapsed, "Command:
{0}: ", command.CommandText);
      }
      base.ReaderExecuted(command, interceptionContext);
    }
  }
}
```

For successful queries or commands, this code writes an Information log with latency information. For exceptions, it creates an Error log.

2. To create the interceptor class that will generate dummy transient errors when you enter "Throw" in the **Name** textbox on the page named *AdminPage.aspx*, create a class file named *InterceptorTransientErrors.cs* in the *Logic* folder and replace the default code with the following code:

```
using System;
using System.Data.Common;
using System.Data.Entity;
using System.Data.Entity.Infrastructure.Interception;
using System.Data.Entity.SqlServer;
using System.Data.SqlClient;
using System.Diagnostics;
using System.Reflection;
using System.Linq;
using WingtipToys.Logging;

namespace WingtipToys.Logic
{
  public class InterceptorTransientErrors : DbCommandInterceptor
  {
    private int _counter = 0;
    private ILogger _logger = new Logger();

    public override void ReaderExecuting(DbCommand command, DbCommandInterceptionContext<DbDataReader>
interceptionContext)
    {
      bool throwTransientErrors = false;
      if (command.Parameters.Count > 0 && command.Parameters[0].Value.ToString() == "Throw")
      {
        throwTransientErrors = true;
        command.Parameters[0].Value = "TransientErrorExample";
        command.Parameters[1].Value = "TransientErrorExample";
      }

      if (throwTransientErrors && _counter < 4)
      {
        _logger.Information("Returning transient error for command: {0}", command.CommandText);
        _counter++;
        interceptionContext.Exception = CreateDummySqlException();
      }
    }

    private SqlException CreateDummySqlException()
    {
      // The instance of SQL Server you attempted to connect to does not support encryption
      var sqlErrorNumber = 20;

      var sqlErrorCtor = typeof(SqlError).GetConstructors(BindingFlags.Instance |
BindingFlags.NonPublic).Where(c => c.GetParameters().Count() == 7).Single();
      var sqlError = sqlErrorCtor.Invoke(new object[] { sqlErrorNumber, (byte)0, (byte)0, "", "", "", 1
});

      var errorCollection = Activator.CreateInstance(typeof(SqlErrorCollection), true);
      var addMethod = typeof(SqlErrorCollection).GetMethod("Add", BindingFlags.Instance |
BindingFlags.NonPublic);
      addMethod.Invoke(errorCollection, new[] { sqlError });

      var sqlExceptionCtor = typeof(SqlException).GetConstructors(BindingFlags.Instance |
BindingFlags.NonPublic).Where(c => c.GetParameters().Count() == 4).Single();
      var sqlException = (SqlException)sqlExceptionCtor.Invoke(new object[] { "Dummy", errorCollection,
null, Guid.NewGuid() });

      return sqlException;
    }
  }
}
```

This code only overrides the `ReaderExecuting` method, which is called for queries that can return multiple rows of data. If you wanted to check connection resiliency for other types of queries, you could also override the `NonQueryExecuting` and `ScalarExecuting` methods, as the logging interceptor does.

Later, you will log in as the "Admin" and select the **Admin** link on the top navigation bar. Then, on the *AdminPage.aspx* page you will add a product named "Throw". The code creates a dummy SQL Database exception for error number 20, a type known to be typically transient. Other error numbers currently recognized as transient are 64, 233, 10053, 10054, 10060, 10928, 10929, 40197, 40501, and 40613, but these are subject to change in new versions of SQL Database. The product will be renamed to "TransientErrorExample", which you can follow in the code of the *InterceptorTransientErrors.cs* file.

The code returns the exception to Entity Framework instead of running the query and passing back results. The transient exception is returned *four* times, and then the code reverts to the normal procedure of passing the query to the database.

Because everything is logged, you'll be able to see that Entity Framework tries to execute the query four times before finally succeeding, and the only difference in the application is that it takes longer to render a page with query results.

The number of times the Entity Framework will retry is configurable; the code specifies four times because that's the default value for the SQL Database execution policy. If you change the execution policy, you'd also change the code here that specifies how many times transient errors are generated. You could also change the code to generate more exceptions so that Entity Framework will throw the `RetryLimitExceededException` exception.

3. In *Global.asax*, add the following using statements:

```
using System.Data.Entity.Infrastructure.Interception;
```

4. Then, add the highlighted lines to the `Application_Start` method:

```
void Application_Start(object sender, EventArgs e)
{
    // Code that runs on application startup
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);

    // Initialize the product database.
    Database.SetInitializer(new ProductDatabaseInitializer());

    // Create administrator role and user.
    RoleActions roleActions = new RoleActions();
    roleActions.createAdmin();

    // Add Routes.
    RegisterRoutes(RouteTable.Routes);

    // Logging.
    DbInterception.Add(new InterceptorTransientErrors());
    DbInterception.Add(new InterceptorLogging());

}
```

These lines of code are what causes your interceptor code to be run when Entity Framework sends queries to the database. Notice that because you created separate interceptor classes for transient error simulation and logging, you can independently enable and disable them.

You can add interceptors using the `DbInterception.Add` method anywhere in your code; it doesn't have to be in the `Application_Start` method. Another option, if you didn't add interceptors in the `Application_Start` method, would be to update or add the class named *WingtipToysConfiguration.cs* and put the above code at the end of the constructor of the `WingtipToysConfiguration` class.

Wherever you put this code, be careful not to execute `DbInterception.Add` for the same interceptor more than once, or you'll get additional interceptor instances. For example, if you add the logging interceptor twice, you'll see two logs for every SQL query.

Interceptors are executed in the order of registration (the order in which the `DbInterception.Add` method is called). The order might matter depending on what you're doing in the interceptor. For example, an interceptor might change the SQL command that it gets in the `CommandText` property. If it does change the SQL command, the next interceptor will get the changed SQL command, not the original SQL command.

You've written the transient error simulation code in a way that lets you cause transient errors by entering a different value in the UI. As an alternative, you could write the interceptor code to always generate the sequence of transient exceptions without checking for a particular parameter value. You could then add the interceptor only when you want to generate transient errors. If you do this, however, don't add the interceptor until after database initialization has completed. In other words, do at least one database operation such as a query on one of your entity sets before you start generating transient errors. The Entity Framework executes several queries during database initialization, and they aren't executed in a transaction, so errors during initialization could cause the context to get into an inconsistent state.

Test logging and connection resiliency

1. In Visual Studio, press **F5** to run the application in debug mode, and then login as "Admin" using "Pa$$word" as the password.

2. Select **Admin** from the navigation bar at the top.

3. Enter a new product named "Throw" with appropriate description, price and image file.

4. Press the **Add Product** button.
   You'll notice that the browser seems to hang for several seconds while Entity Framework is retrying the query several times. The first retry happens very quickly, then the wait increases before each additional retry. This process of waiting longer before each retry is called *exponential backoff*.

5. Wait until the page is no longer attempting to load.

6. Stop the project and look at the Visual Studio **Output** window to see the tracing output. You can find the **Output** window by selecting **Debug** -> **Windows** -> **Output**. You might have to scroll past several other logs written by your logger.

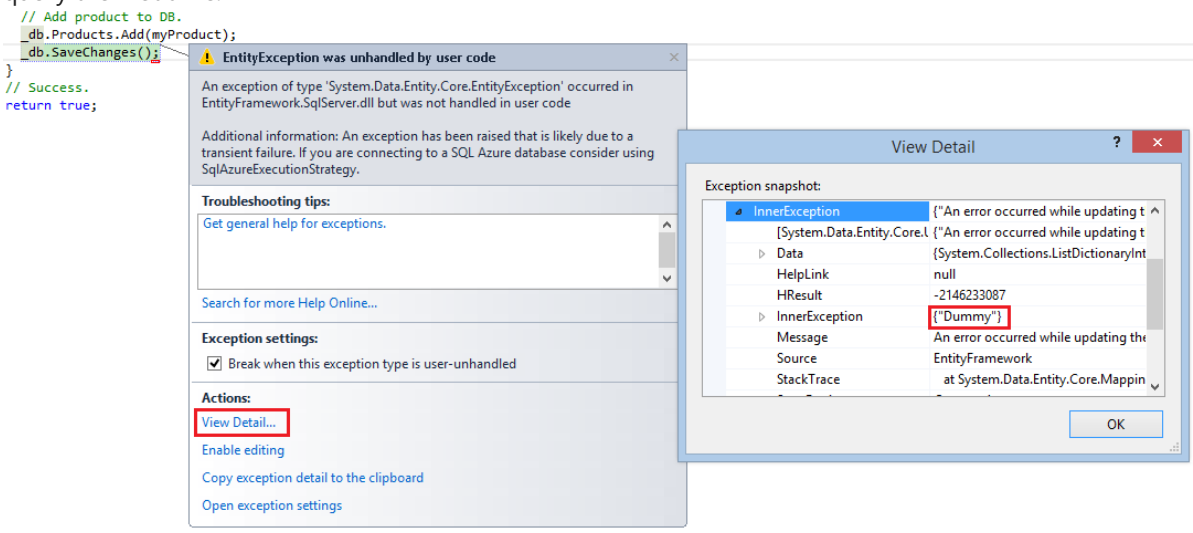
   Notice that you can see the actual SQL queries sent to the database. You see some initial queries and commands that Entity Framework does to get started, checking the database version and migration history table.

```
Output
Show output from: Debug
SELECT [ProductID]
FROM [dbo].[Products]
WHERE @@ROWCOUNT > 0 AND [ProductID] = scope_identity() Exception: System.Data.SqlClient.SqlException (0x80131904): Dummy
ClientConnectionId:28b113d8-add8-435b-ac6e-466b7b93cd1b
iisexpress.exe Information: 0 : Returning transient error for command: INSERT [dbo].[Products]([ProductName], [Description], [ImagePath], [Uni
VALUES (@0, @1, @2, @3, @4)
SELECT [ProductID]
FROM [dbo].[Products]
WHERE @@ROWCOUNT > 0 AND [ProductID] = scope_identity()
iisexpress.exe Error: 0 : Error executing command: INSERT [dbo].[Products]([ProductName], [Description], [ImagePath], [UnitPrice], [CategoryID
VALUES (@0, @1, @2, @3, @4)
SELECT [ProductID]
FROM [dbo].[Products]
WHERE @@ROWCOUNT > 0 AND [ProductID] = scope_identity() Exception: System.Data.SqlClient.SqlException (0x80131904): Dummy
ClientConnectionId:f3a12c13-7664-471a-987d-2bb2ad9cd6cb
iisexpress.exe Information: 0 : Component:SQL Database;Method:Interceptor.ReaderExecuting;Timespan:00:00:00.0000013;Properties:Command: INSERT
VALUES (@0, @1, @2, @3, @4)
SELECT [ProductID]
FROM [dbo].[Products]
WHERE @@ROWCOUNT > 0 AND [ProductID] = scope_identity():
A first chance exception of type 'System.Threading.ThreadAbortException' occurred in mscorlib.dll
iisexpress.exe Information: 0 : Component:SQL Database;Method:Interceptor.ReaderExecuting;Timespan:00:00:00.0000010;Properties:Command: SELECT
    [GroupBy1].[A1] AS [C1]
    FROM ( SELECT
        SUM([Extent1].[Quantity]) AS [A1]
        FROM [dbo].[CartItems] AS [Extent1]
        WHERE (([Extent1].[CartId] = @p__linq__0) AND ( NOT ([Extent1].[CartId] IS NULL OR @p__linq__0 IS NULL))) OR (([Extent1].[CartId] IS N
    )  AS [GroupBy1]:
iisexpress.exe Information: 0 : Component:SQL Database;Method:Interceptor.ReaderExecuting;Timespan:00:00:00.0000006;Properties:Command: SELECT
    [Extent1].[CategoryID] AS [CategoryID],
    [Extent1].[CategoryName] AS [CategoryName],
    [Extent1].[Description] AS [Description]
    FROM [dbo].[Categories] AS [Extent1]:
iisexpress.exe Information: 0 : Component:SQL Database;Method:Interceptor.ReaderExecuting;Timespan:00:00:00.0000010;Properties:Command: SELECT
    [Extent1].[CategoryID] AS [CategoryID],
    [Extent1].[CategoryName] AS [CategoryName],
    [Extent1].[Description] AS [Description]
```

Note that you can't repeat this test unless you stop the application and restart it. If you wanted to be able to test connection resiliency multiple times in a single run of the application, you could write code to reset the error counter in `InterceptorTransientErrors` .

7. To see the difference the execution strategy (retry policy) makes, comment out the `SetExecutionStrategy` line in *WingtipToysConfiguration.cs* file in the *Logic* folder, run the **Admin** page in debug mode again, and add the product named "Throw" again.

   This time the debugger stops on the first generated exception immediately when it tries to execute the query the first time.



8. Uncomment the `SetExecutionStrategy` line in the *WingtipToysConfiguration.cs* file.

Summary

In this tutorial you've seen how to modify a Web Forms sample application to support connection resiliency and command interception.

Next Steps

After you have reviewed connection resiliency and command interception in ASP.NET Web Forms, review the ASP.NET Web Forms topic Asynchronous Methods in ASP.NET 4.5. The topic will teach you the basics of building an

asynchronous ASP.NET Web Forms application using Visual Studio.

Older Versions - Web Forms Getting Started

2/20/2020 • 4 minutes to read • Edit Online

Documentation on ASP.NET Web Forms 4 and earlier versions.

- ASP.NET 4 - Tailspin Spyworks
  - Part 1: File-> New Project
  - Part 2: Data Access Layer
  - Part 3: Layout and Category Menu
  - Part 4: Listing Products
  - Part 5: Business Logic
  - Part 6: ASP.NET Membership
  - Part 7: Adding Features
  - Part 8: Final Pages, Exception Handling, and Conclusion
- ASP.NET 4 - Getting Started with Entity Framework 4 and Web Forms
  - Getting Started with Entity Framework 4.0 Database First
  - Getting Started with Entity Framework 4.0 Database First - Part 2
  - Getting Started with Entity Framework 4.0 Database First - Part 3
  - Getting Started with Entity Framework 4.0 Database First - Part 4
  - Getting Started with Entity Framework 4.0 Database First - Part 5
  - Getting Started with Entity Framework 4.0 Database First - Part 6
  - Getting Started with Entity Framework 4.0 Database First - Part 7
  - Getting Started with Entity Framework 4.0 Database First - Part 8
- ASP.NET 4 - Continuing with Entity Framework 4 and Web Forms
  - Entity Framework 4.0 and the ObjectDataSource Control, Part 1: Getting Started
  - Entity Framework 4.0 and the ObjectDataSource Control, Part 2: Adding a Business Logic Layer and Unit Tests
  - Entity Framework 4.0 and the ObjectDataSource Control, Part 3: Sorting and Filtering
  - Handling Concurrency with the Entity Framework 4.0 in ASP.NET 4
  - Maximizing Performance with the Entity Framework 4.0 in ASP.NET 4
  - What's New in the Entity Framework 4.0
- ASP.NET 4 - Web Deployment with SQL Server Compact using Visual Studio
  - Visual Studio Web Deployment with SQL Server Compact - Introduction
  - Visual Studio Web Deployment with SQL Server Compact- Deploying SQL Server Compact Databases
  - Visual Studio Web Deployment with SQL Server Compact - Web.Config File Transformations
  - Visual Studio Web Deployment with SQL Server Compact - Configuring Project Properties
  - Visual Studio Web Deployment with SQL Server Compact - Deploying to IIS as a Test Environment
  - Visual Studio Web Deployment with SQL Server Compact - Setting Folder Permissions
  - Visual Studio Web Deployment with SQL Server Compact - Deploying to the Production Environment
  - Visual Studio Web Deployment with SQL Server Compact - Deploying a Code-Only Update
  - Visual Studio Web Deployment with SQL Server Compact - Deploying a Database Update
  - Visual Studio Web Deployment with SQL Server Compact - Migrating to SQL Server

  - Visual Studio Web Deployment with SQL Server Compact - Deploying a SQL Server Database Update
  - Visual Studio Web Deployment with SQL Server Compact - Troubleshooting
- ASP.NET 3.5 - AJAX
  - Understanding Partial Page Updates with ASP.NET AJAX
  - Understanding ASP.NET AJAX UpdatePanel Triggers
  - Understanding ASP.NET AJAX Authentication and Profile Application Services
  - Understanding ASP.NET AJAX Localization
  - Understanding ASP.NET AJAX Web Services
  - Understanding ASP.NET AJAX Debugging Capabilities
- ASP.NET 3.5 - Web Forms Master Pages
  - Creating a Site-Wide Layout Using Master Pages (C#)
  - Multiple ContentPlaceHolders and Default Content (C#)
  - Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page (C#)
  - URLs in Master Pages (C#)
  - Control ID Naming in Content Pages (C#)
  - Interacting with the Master Page from the Content Page (C#)
  - Interacting with the Content Page from the Master Page (C#)
  - Master Pages and ASP.NET AJAX (C#)
  - Specifying the Master Page Programmatically (C#)
  - Nested Master Pages (C#)
  - Creating a Site-Wide Layout Using Master Pages (VB)
  - Multiple ContentPlaceHolders and Default Content (VB)
  - Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page (VB)
  - URLs in Master Pages (VB)
  - Control ID Naming in Content Pages (VB)
  - Interacting with the Master Page from the Content Page (VB)
  - Interacting with the Content Page from the Master Page (VB)
  - Master Pages and ASP.NET AJAX (VB)
  - Specifying the Master Page Programmatically (VB)
  - Nested Master Pages (VB)
- Deploying Web Site Projects in Visual Studio 2008 and earlier
  - ASP.NET Hosting Options (C#)
  - Determining What Files Need to Be Deployed (C#)
  - Deploying Your Site Using an FTP Client (C#)
  - Deploying Your Site Using Visual Studio (C#)
  - Common Configuration Differences Between Development and Production (C#)
  - Core Differences Between IIS and the ASP.NET Development Server (C#)
  - Deploying a Database (C#)
  - Configuring the Production Web Application to Use the Production Database (C#)
  - Configuring a Website that Uses Application Services (C#)
  - Strategies for Database Development and Deployment (C#)
  - Displaying a Custom Error Page (C#)
  - Processing Unhandled Exceptions (C#)
  - Logging Error Details with ASP.NET Health Monitoring (C#)
  - Logging Error Details with ELMAH (C#)

- Precompiling Your Website (C#)
- Users and Roles On Production Website (C#)
- ASP.NET Hosting Options (VB)
- Determining What Files Need to Be Deployed (VB)
- Deploying Your Site Using an FTP Client (VB)
- Deploying Your Site Using Visual Studio (VB)
- Common Configuration Differences Between Development and Production (VB)
- Core Differences Between IIS and the ASP.NET Development Server (VB)
- Deploying a Database (VB)
- Configuring the Production Web Application to Use the Production Database (VB)
- Configuring a Website that Uses Application Services (VB)
- Strategies for Database Development and Deployment (VB)
- Displaying a Custom Error Page (VB)
- Processing Unhandled Exceptions (VB)
- Logging Error Details with ASP.NET Health Monitoring (VB)
- Logging Error Details with ELMAH (VB)
- Precompiling Your Website (VB)
- Users and Roles On Production Website (VB)

ASP.NET 4 - Tailspin Spyworks

2/20/2020 • 2 minutes to read • Edit Online

How to use new features in ASP.NET 4 to build an online store, including shopping, checkout, and administration.

- Part 1: File-> New Project
- Part 2: Data Access Layer
- Part 3: Layout and Category Menu
- Part 4: Listing Products
- Part 5: Business Logic
- Part 6: ASP.NET Membership
- Part 7: Adding Features
- Part 8: Final Pages, Exception Handling, and Conclusion

Part 1: File-> New Project

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

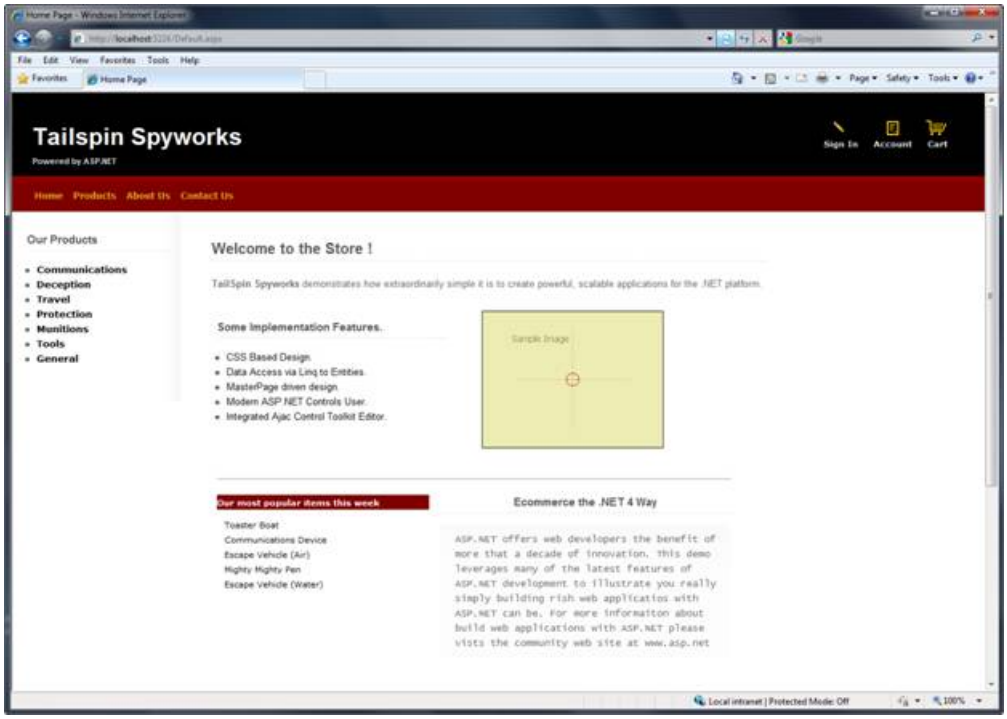
Tailspin Spyworks demonstrates how extraordinarily simple it is to create powerful, scalable applications for the .NET platform. It shows off how to use the great new features in ASP.NET 4 to build an online store, including shopping, checkout, and administration.

This tutorial series details all of the steps taken to build the Tailspin Spyworks sample application. Part 1 covers Overview and File/New Project.

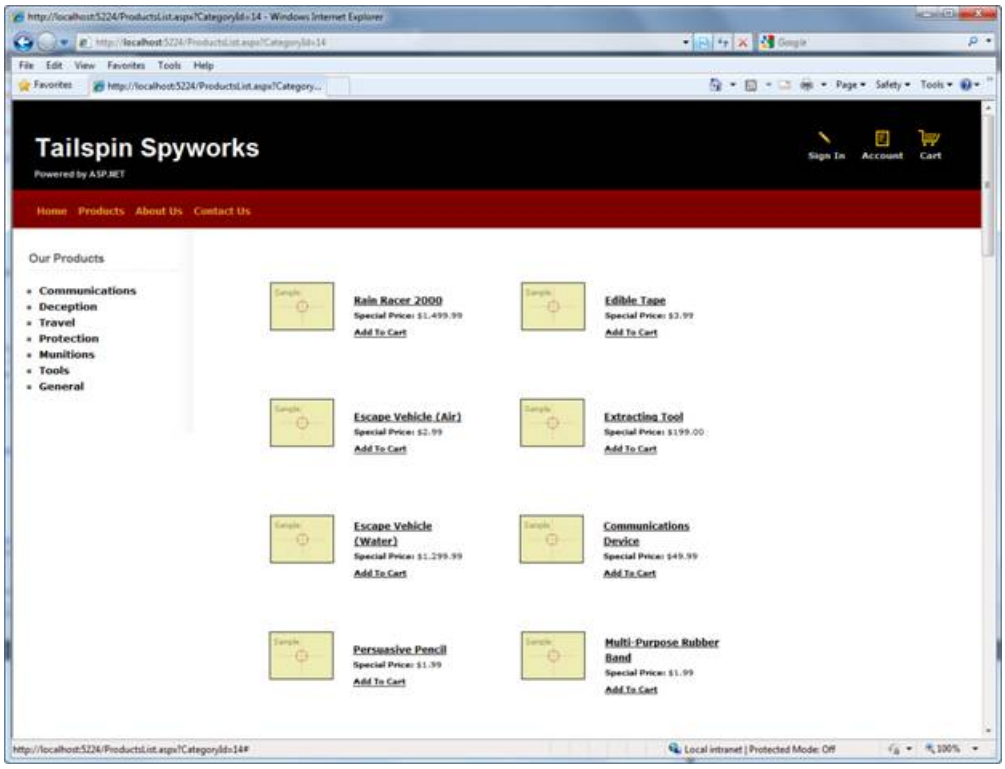
Overview

This tutorial is an introduction to ASP.NET WebForms. We'll be starting slowly, so beginner level web development experience is okay.

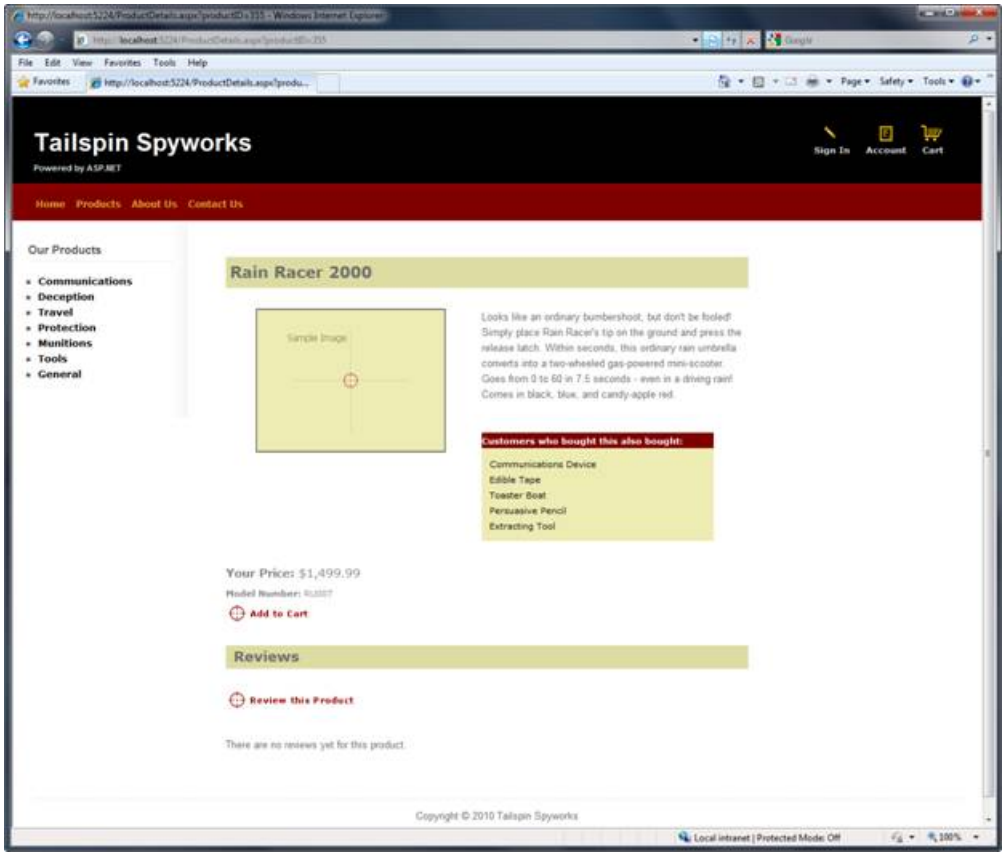
The application we'll be building is a simple on-line store.



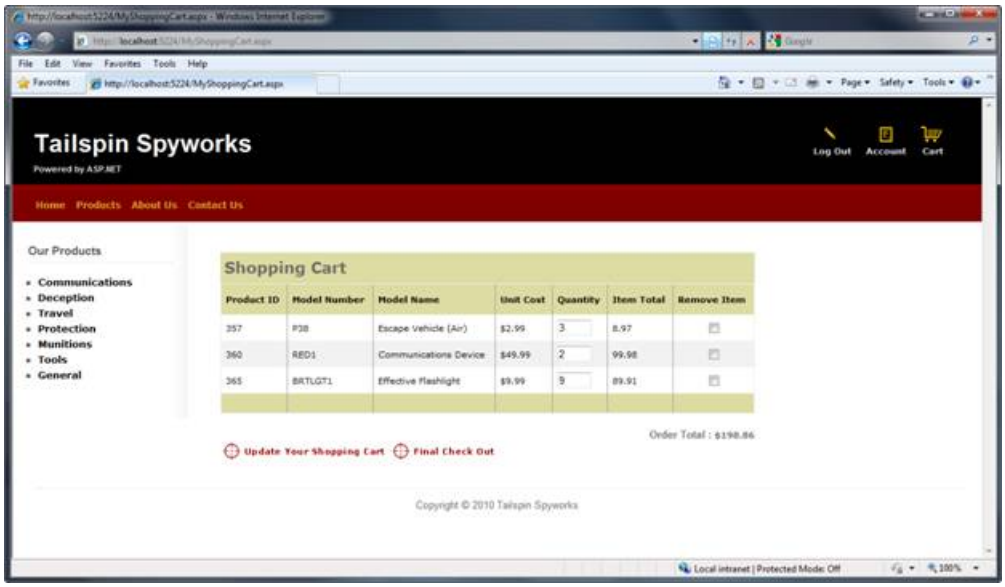
Visitors can browse Products by Category:



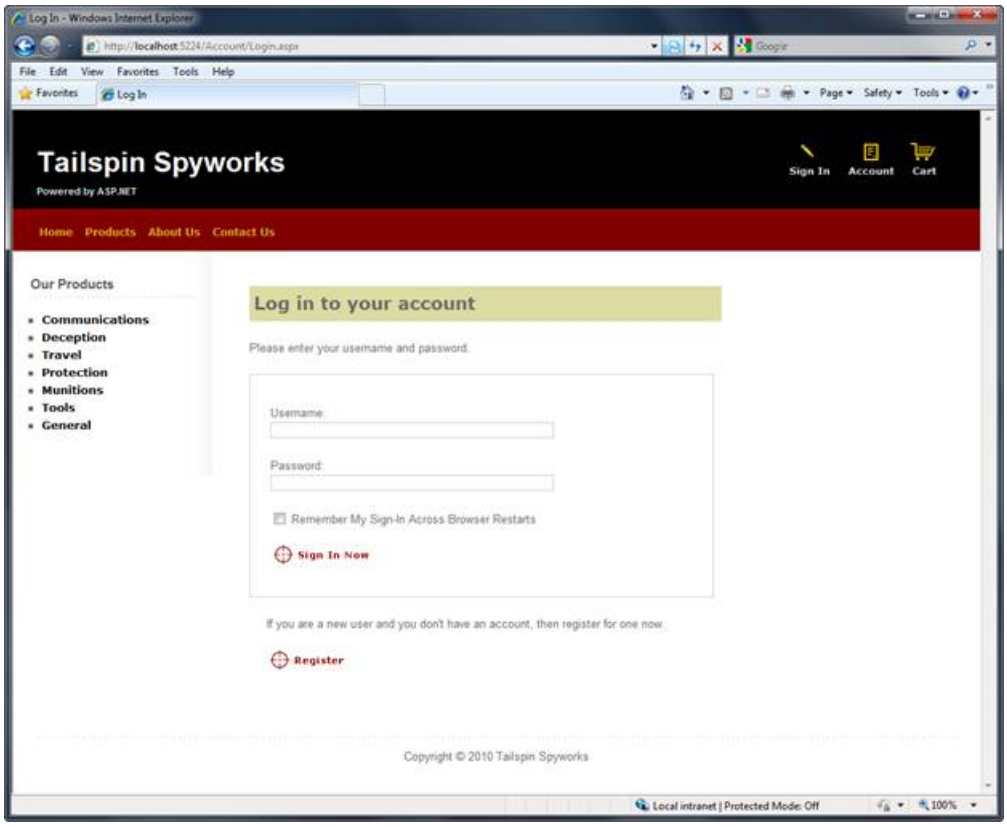
They can view a single product and add it to their cart:

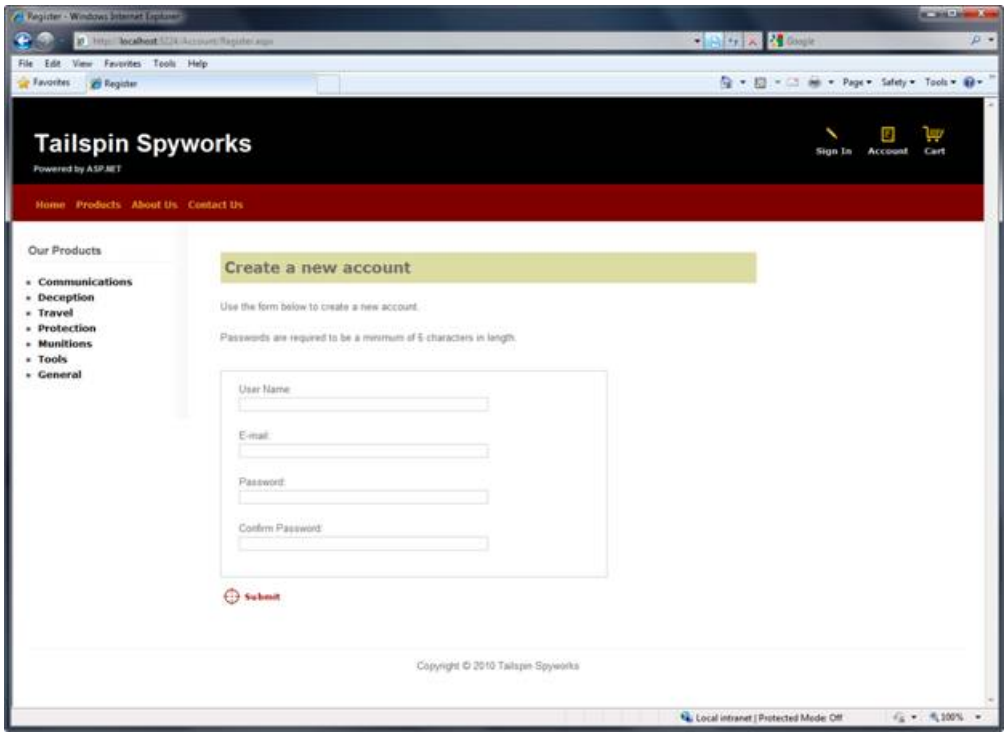


They can review their cart, removing any items they no longer want:

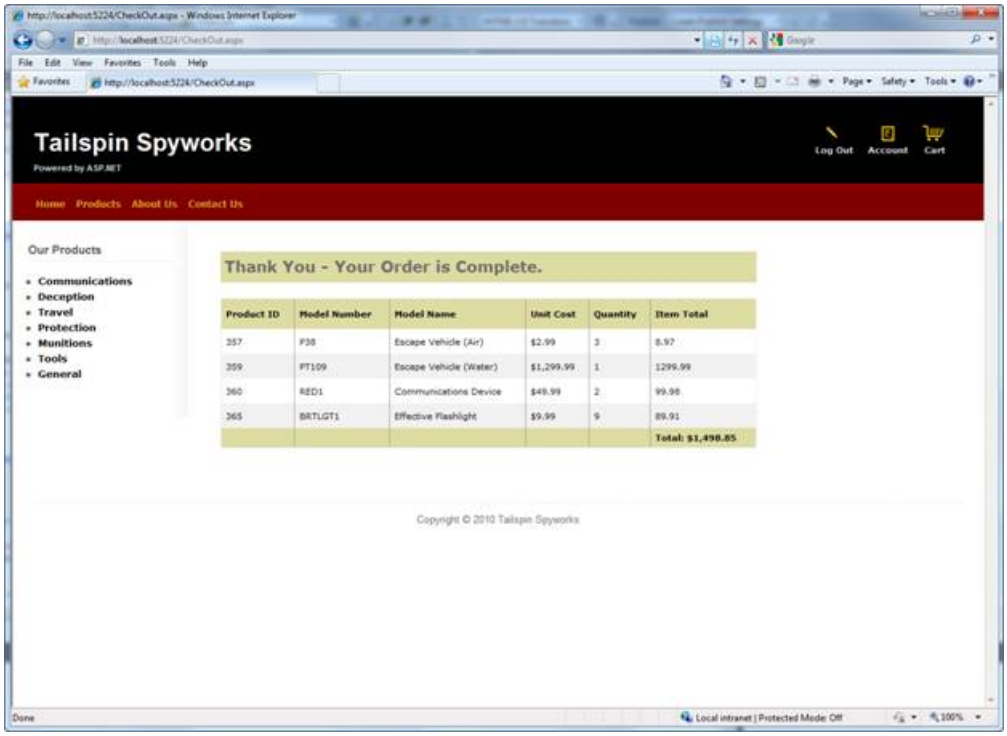


Proceeding to Checkout will prompt them to





After ordering, they see a simple confirmation screen:



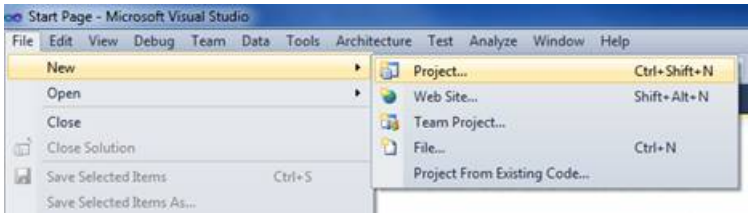
We'll begin by creating a new ASP.NET WebForms project in Visual Studio 2010, and we'll incrementally add features to create a complete functioning application. Along the way, we'll cover database access, list and grid views, data update pages, data validation, using master pages for consistent page layout, AJAX, validation, user membership, and more.

You can follow along step by step, or you can download the completed application from http://tailspinspyworks.codeplex.com/

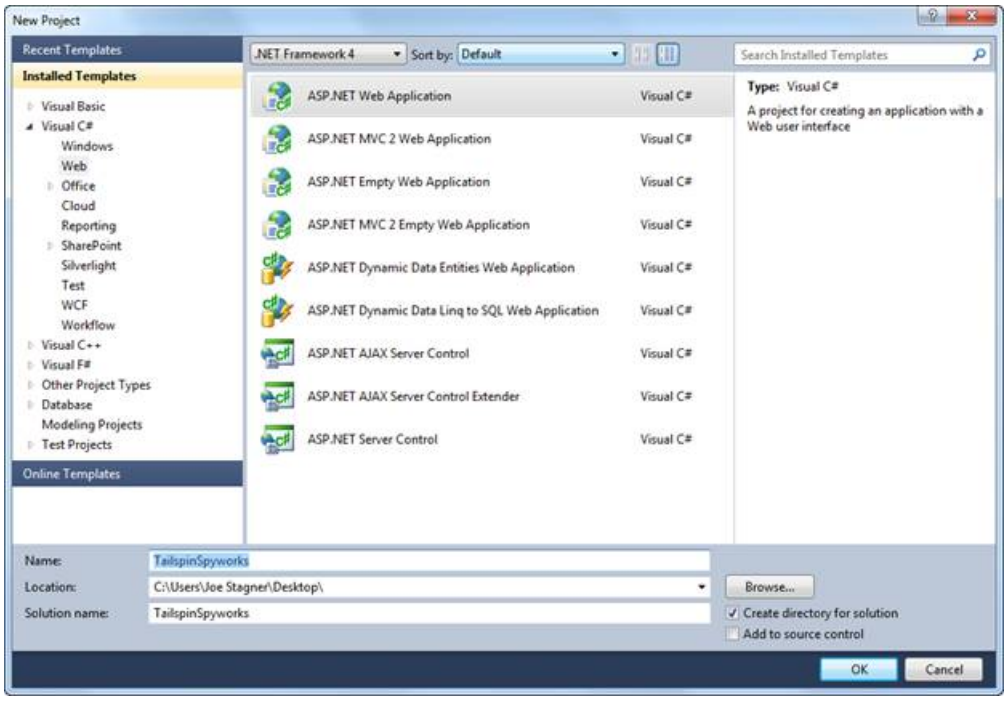
You can use either Visual Studio 2010 or the free Visual Web Developer 2010 from https://www.microsoft.com/express/Web/. To build the application, you can use either SQL Server or the free SQL Server Express to host the database.

File / New Project

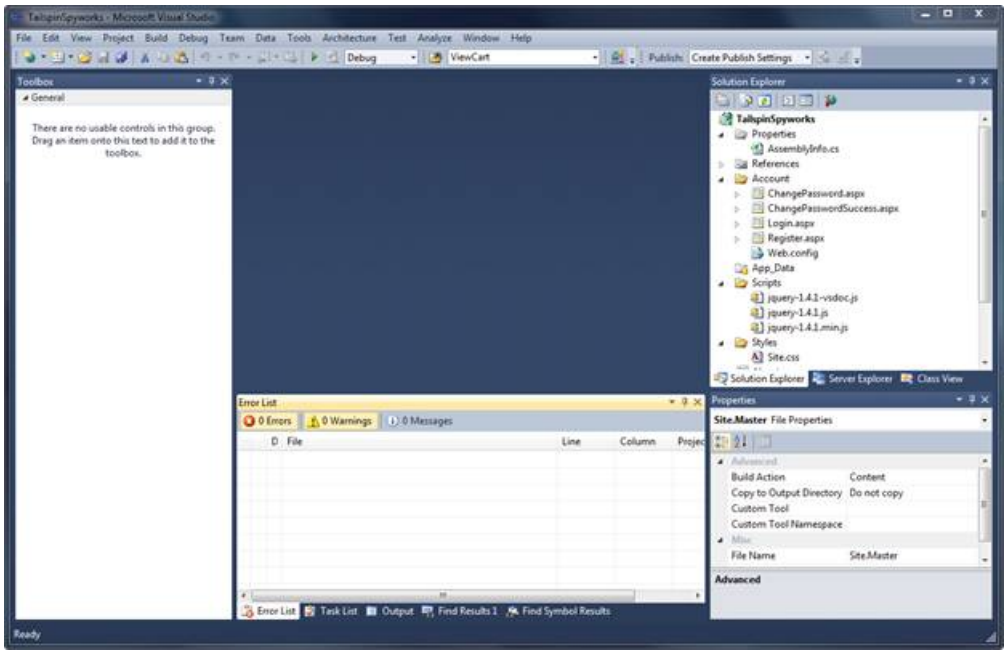
We'll start by selecting the New Project from the File menu in Visual Studio. This brings up the New Project dialog.



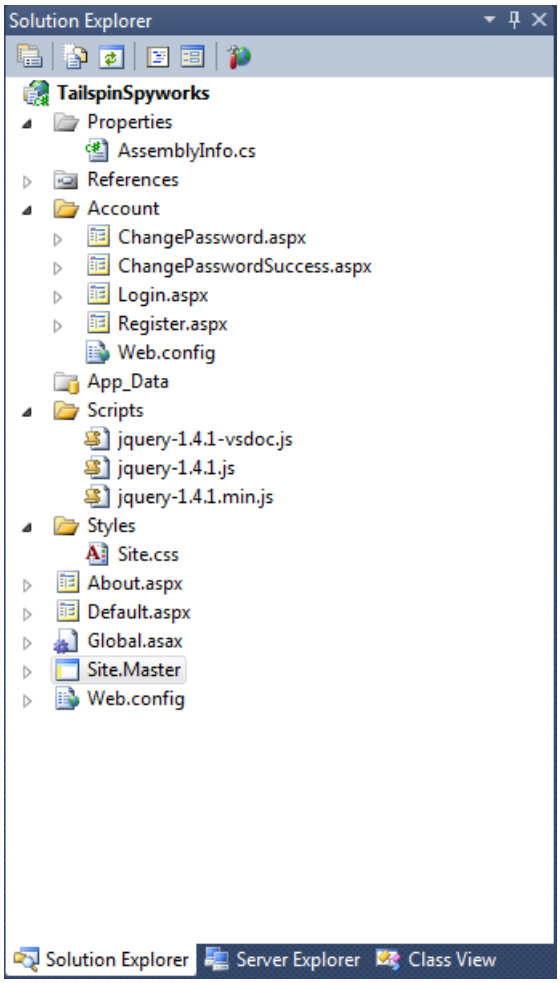
We'll select the Visual C# / Web Templates group on the left, and then choose the "ASP.NET Web Application" template in the center column. Name your project TailspinSpyworks and press the OK button.



This will create our project. Let's take a look at the folders that are included in our application in the Solution Explorer on the right side.



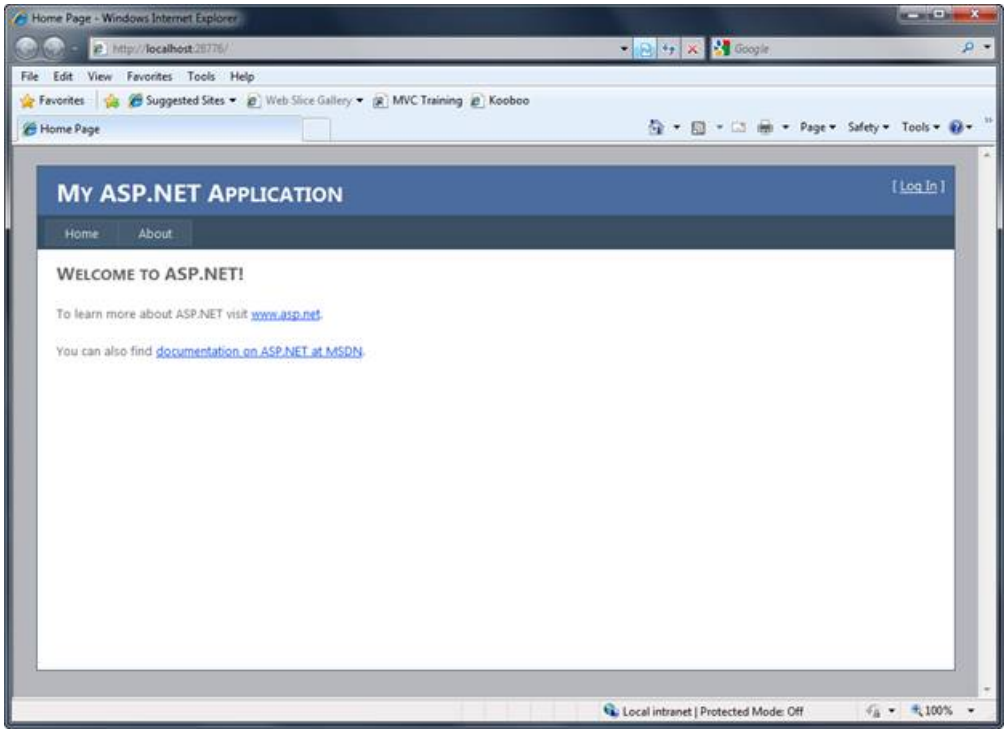
The Empty Solution isn't completely empty – it adds a basic folder structure:



Note the conventions implemented by the ASP.NET 4 default project template.

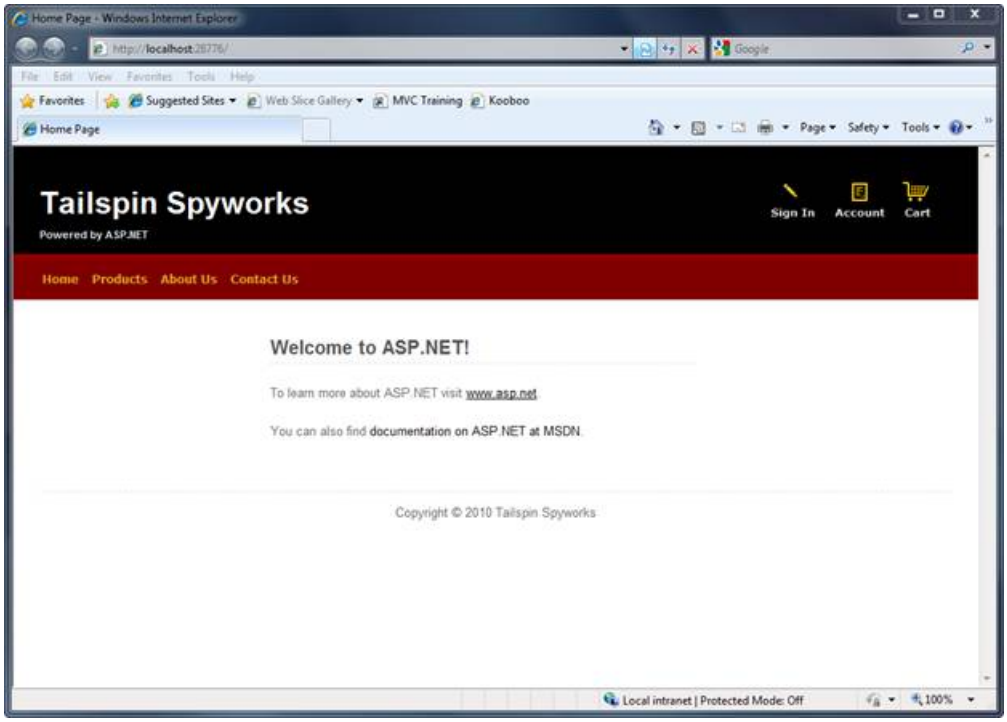
- The "Account" folder implements a basic user interface for ASP.NET's membership subsystem.
- The "Scripts" folder serves as the repository for client side JavaScript files and the core jQuery .js files are made available by default.
- The "Styles" folder is used to organize our web site visuals (CSS Style Sheets)

When we press F5 to run our application and render the default.aspx page we see the following.



Our first application enhancement will be to replace the Style.css file from the default WebForms template with the CSS classes and associated image files that will render the visual asthetics that we want for our Tailspin Spyworks application.

After doing so our default.aspx page renders like this.



Notice the image links at the top right of the page and the menu items that have been added to the master page. Only the "Sign In" and "Account" links point to pages that exist (generated by the default template) and the rest of the pages we will implement as we build our application.

We're also going to relocate the Master Page to the Styles directory. Though this is only a preference it may make things a little easier if we decide to make our application "skinable" in the future.

After doing this we'll need to change the master page references in all the .aspx files generated by the default ASP.NET WebForms pages.

NEXT

Part 2: Data Access Layer

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

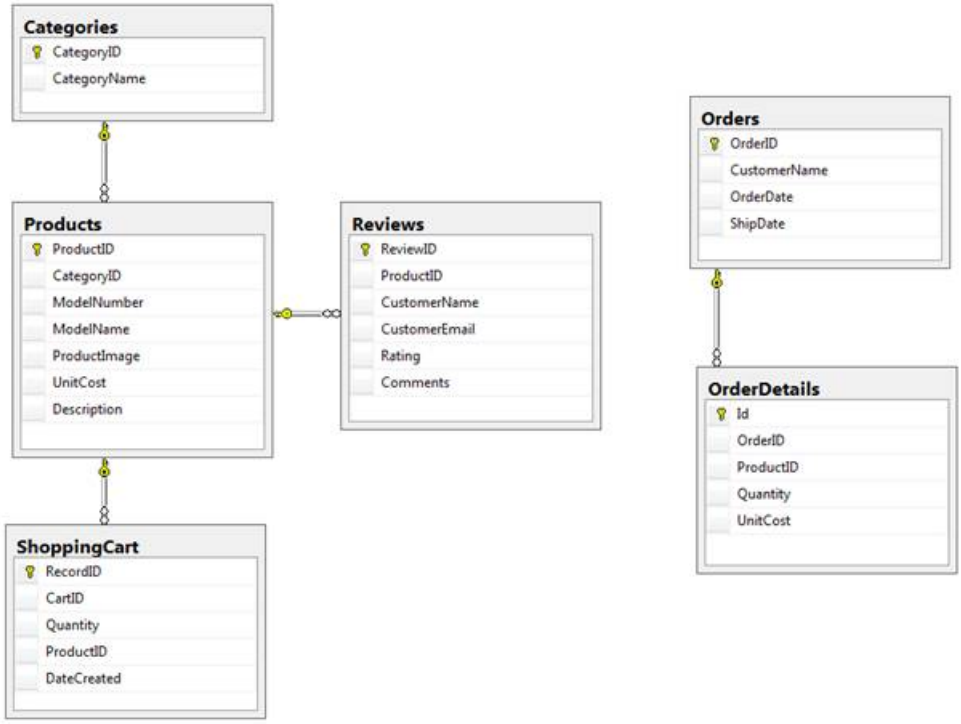
Tailspin Spyworks demonstrates how extraordinarily simple it is to create powerful, scalable applications for the .NET platform. It shows off how to use the great new features in ASP.NET 4 to build an online store, including shopping, checkout, and administration.

This tutorial series details all of the steps taken to build the Tailspin Spyworks sample application. Part 2 covers adding the data access layer.

Adding the Data Access Layer

Our ecommerce application will depend on two databases.

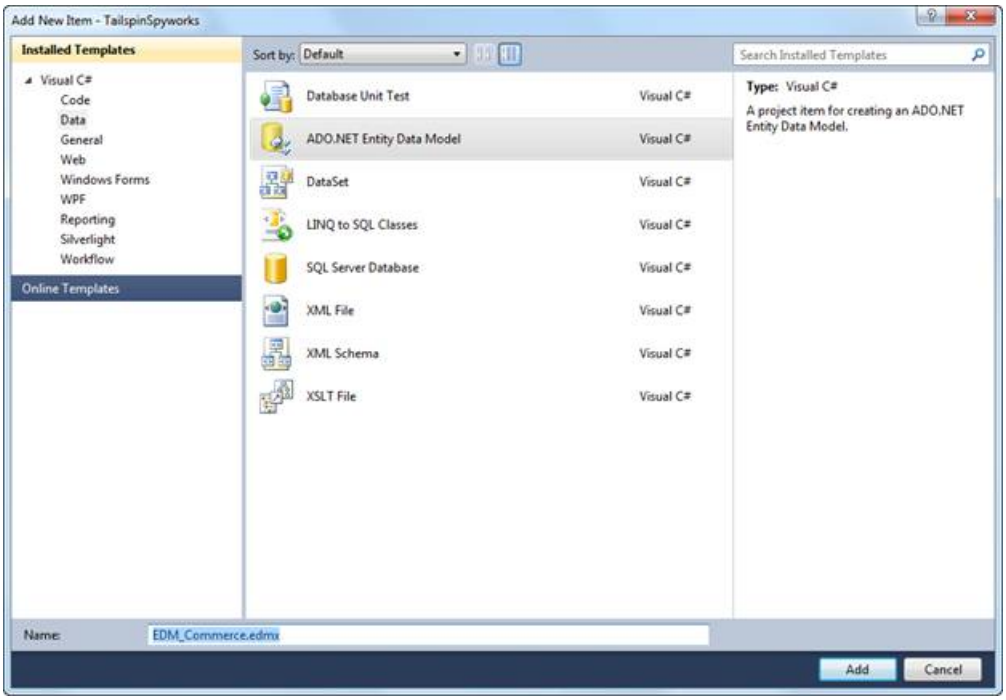
For customer information we'll use the standard ASP.NET Membership database. For our shopping cart and product catalog we'll implement a SQL Express database as follows.



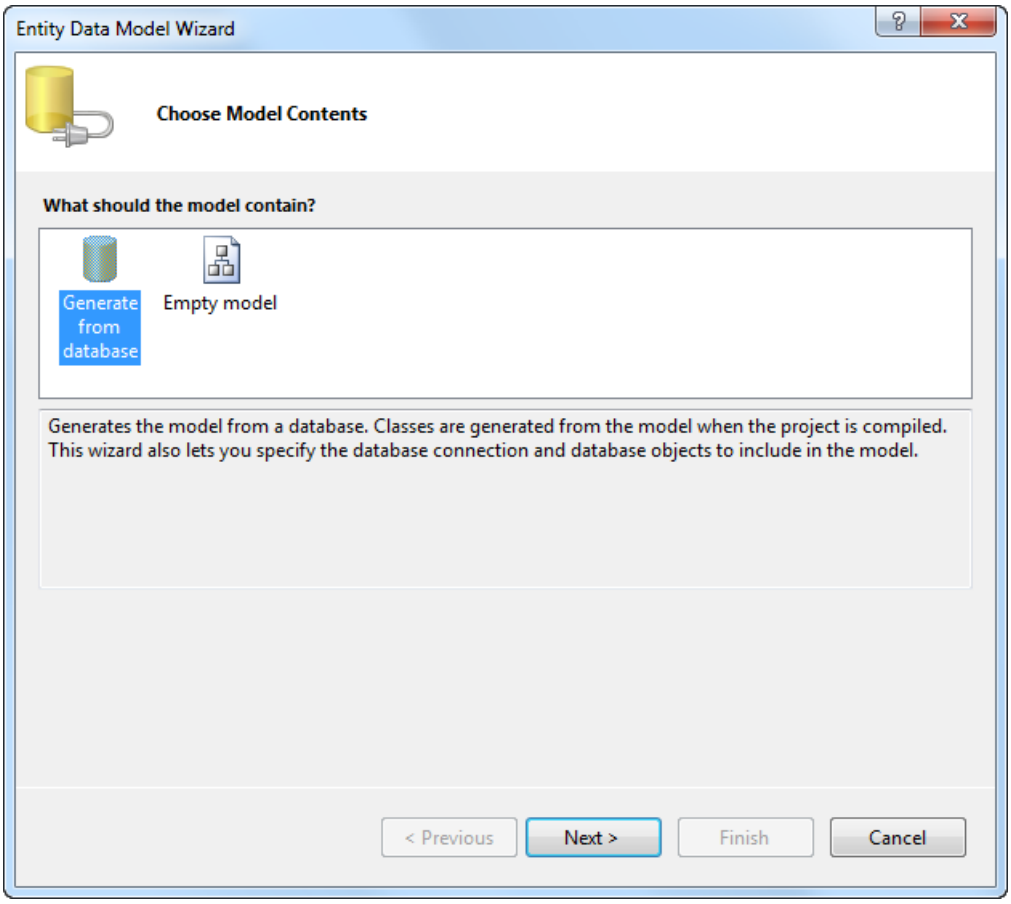
Having created the database (Commerce.mdf) in the application's App_Data folder we can proceed to create our Data Access Layer using the .NET Entity Framework.

We'll create a folder named "Data_Access" and them right click on that folder and select "Add New Item".

In the "Installed Templates" item and then select "ADO.NET Entity Data Model" enter EDM_Commerce.edmx as the name and click the "Add" button.



Choose "Generate from Database".



Entity Data Model Wizard

Choose Your Data Connection

Which data connection should your application use to connect to the database?

Commerce.mdf | New Connection...

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

- No, exclude sensitive data from the connection string. I will set it in my application code.
- Yes, include the sensitive data in the connection string.

Entity connection string:

```
metadata=res://*/Data_Access.EDM_Commerce.csdl|res://*/Data_Access.EDM_Commerce.ssdl|
res://*/Data_Access.EDM_Commerce.msl;provider=System.Data.SqlClient;provider connection
string="Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\Commerce.mdf;Integrated
Security=True;User Instance=True"
```

☑ Save entity connection settings in Web.Config as:

CommerceEntities

< Previous | Next > | Finish | Cancel

Entity Data Model Wizard

Choose Your Database Objects

Which database objects do you want to include in your model?

- ☑ Tables
  - ☑ Categories (dbo)
  - ☑ OrderDetails (dbo)
  - ☑ Orders (dbo)
  - ☑ Products (dbo)
  - ☑ Reviews (dbo)
  - ☑ ShoppingCart (dbo)
  - ☐ sysdiagrams (dbo)
- ☑ Views
  - ☑ VewOrderDetails (dbo)
  - ☑ ViewAlsoPurchased (dbo)
  - ☑ ViewCart (dbo)
- ☑ Stored Procedures
  - ☐ fn_diagramobjects (dbo)
  - ☑ SelectPurchasedWithProducts (dbo)
  - ☐ sp_alterdiagram (dbo)
  - ☐ sp_creatediagram (dbo)
  - ☐ sp_dropdiagram (dbo)
  - ☐ sp_helpdiagramdefinition (dbo)
  - ☐ sp_helpdiagrams (dbo)
  - ☐ sp_renamediagram (dbo)
  - ☐ sp_upgraddiagrams (dbo)

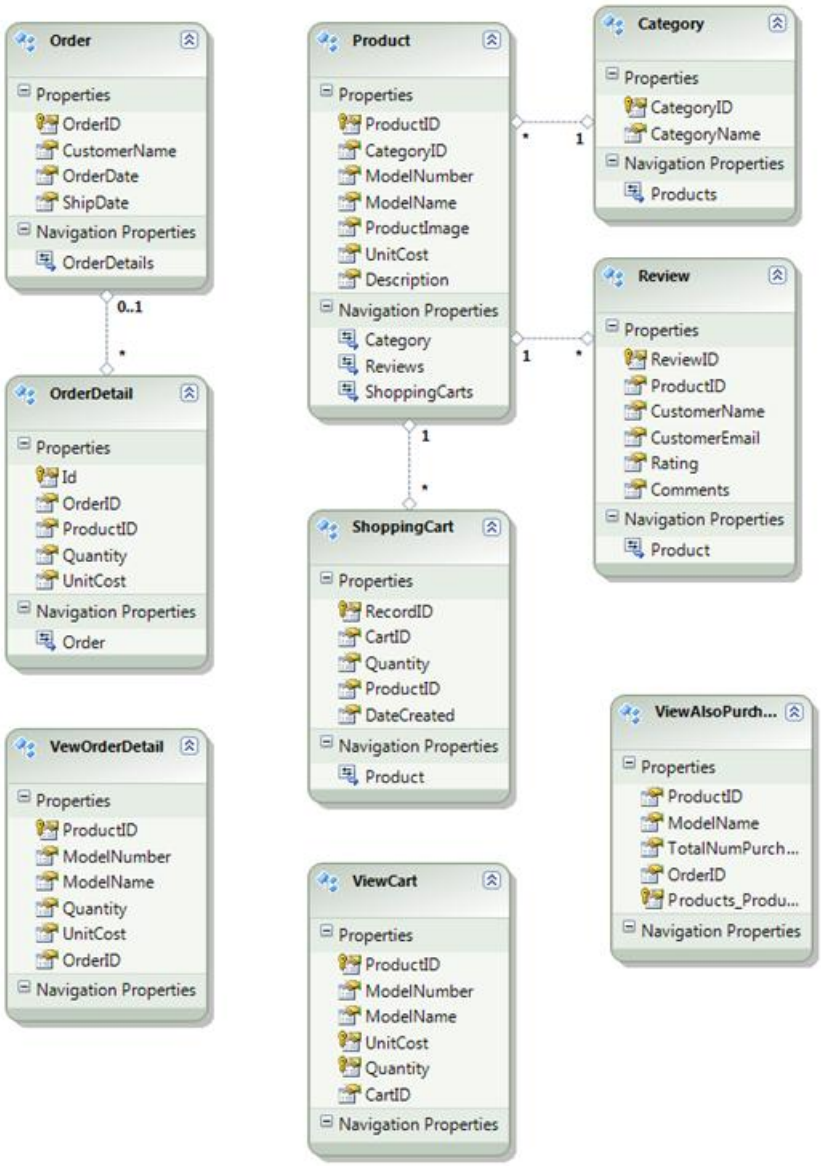
☑ Pluralize or singularize generated object names

☑ Include foreign key columns in the model

Model Namespace:

CommerceModel

< Previous | Next > | Finish | Cancel



Save and build.

Now we are ready to add our first feature – a product category menu.

PREVIOUS NEXT

Part 3: Layout and Category Menu

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Tailspin Spyworks demonstrates how extraordinarily simple it is to create powerful, scalable applications for the .NET platform. It shows off how to use the great new features in ASP.NET 4 to build an online store, including shopping, checkout, and administration.

This tutorial series details all of the steps taken to build the Tailspin Spyworks sample application. Part 3 covers adding layout and a category menu.

Adding Some Layout and a Category Menu

In our site master page we'll add a div for the left side column that will contain our product category menu.

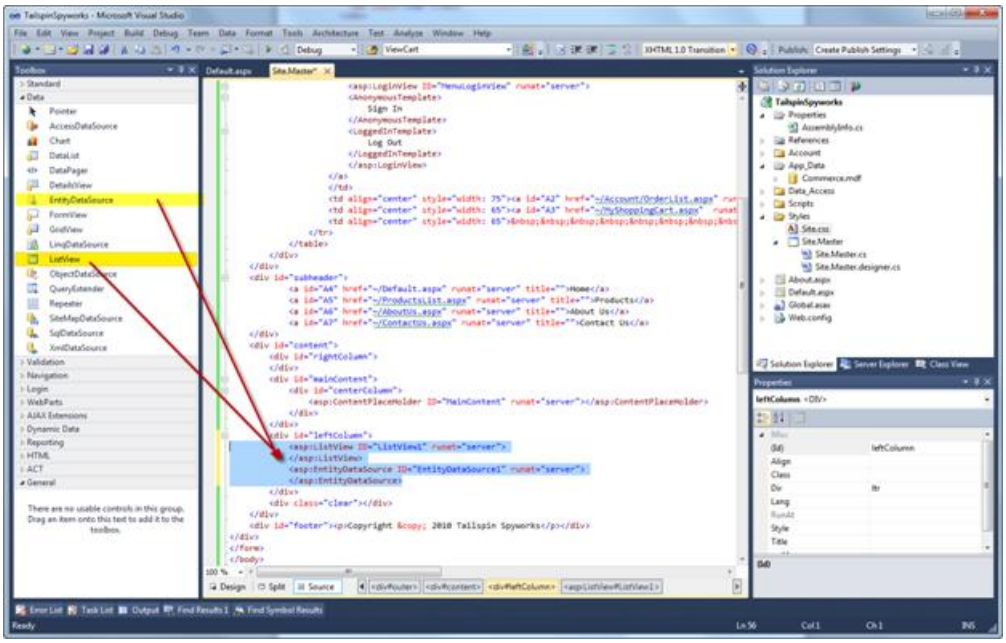
```
<div id="content">
  <div id="rightColumn"></div>
  <div id="mainContent">
    <div id="centerColumn">
       <asp:ContentPlaceHolder ID="MainContent" runat="server"></asp:ContentPlaceHolder>
    </div>
  </div>
  <div id="leftColumn">
<!-- Our menu will go here -->
  </div>
  <div class="clear"></div>
</div>
```

Note that the desired aligning and other formatting will be provided by the CSS class that we added to our Style.css file.

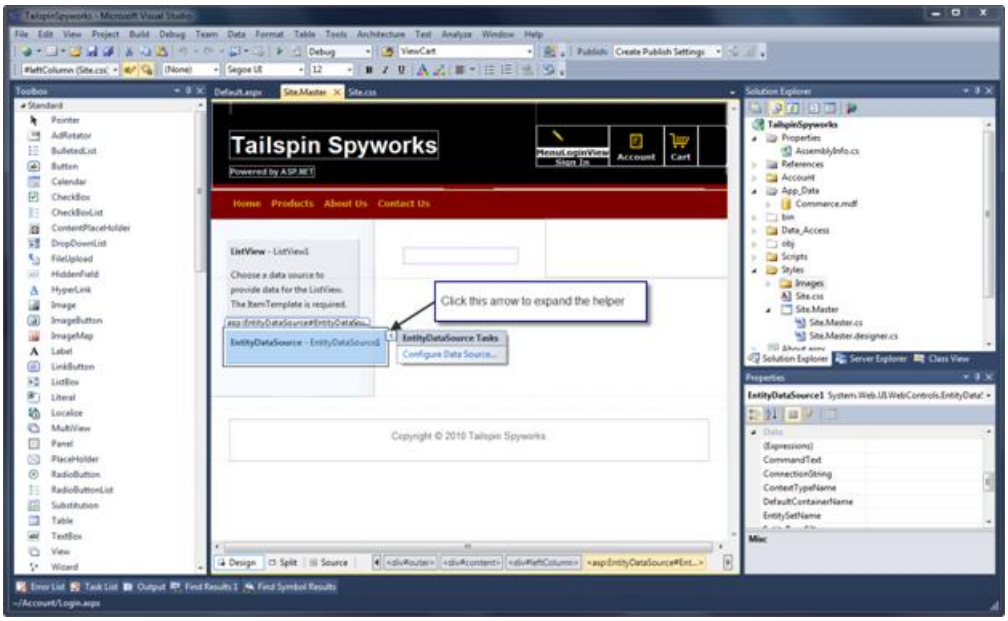
```
#leftColumn
{
position: relative;float: left;width: 14em;padding: 2em 1.5em 2em;background: #fff url('images/a1.gif')
repeat-y right top;
        top: 1px;
        left: 0px;
        height: 100%;
}
```

The product category menu will be dynamically created at runtime by querying the Commerce database for existing product categories and creating the menu items and corresponding links.

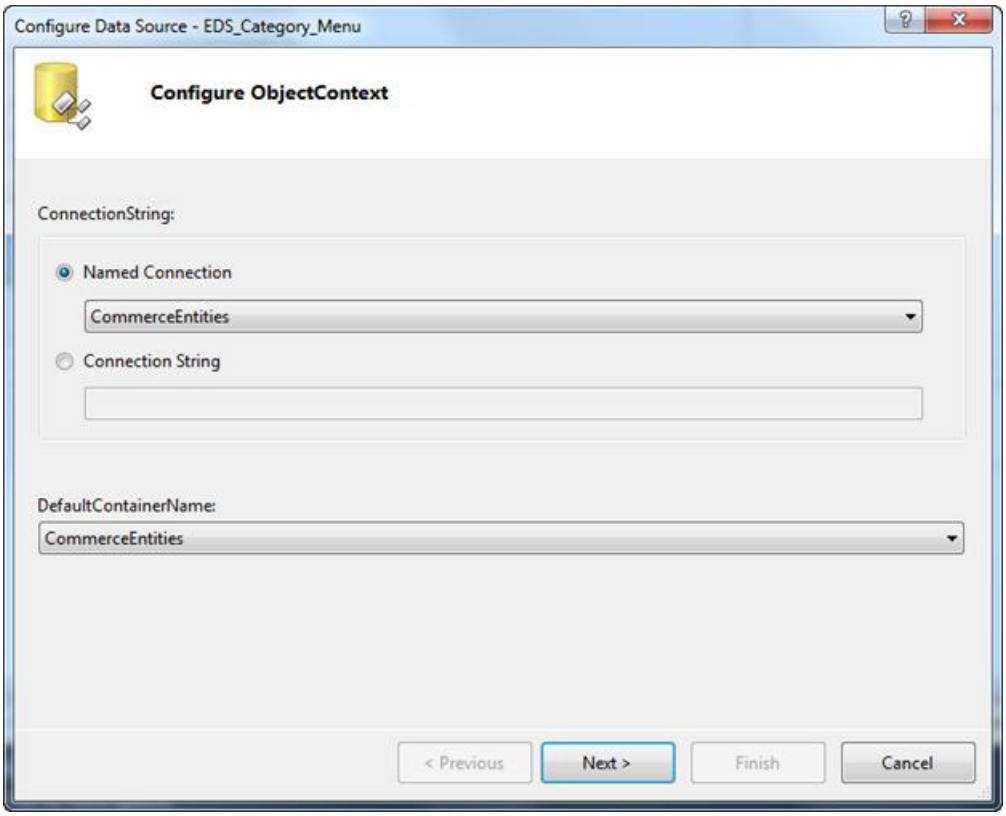
To accomplish this we will use two of ASP.NET's powerful data controls. The "Entity Data Source" control and the "ListView" control.



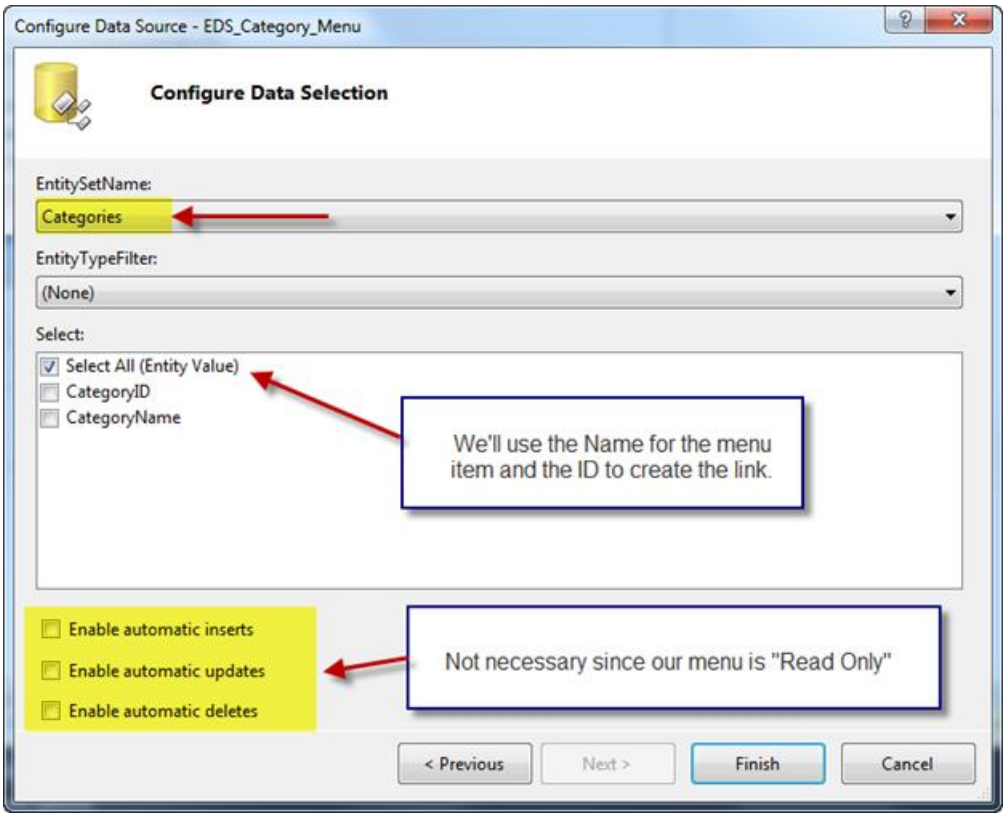
Let's switch to "Design View" and use the helpers to configure our controls.



Let's set the EntityDataSource ID property to EDS_Category_Menu and click on "Configure Data Source".

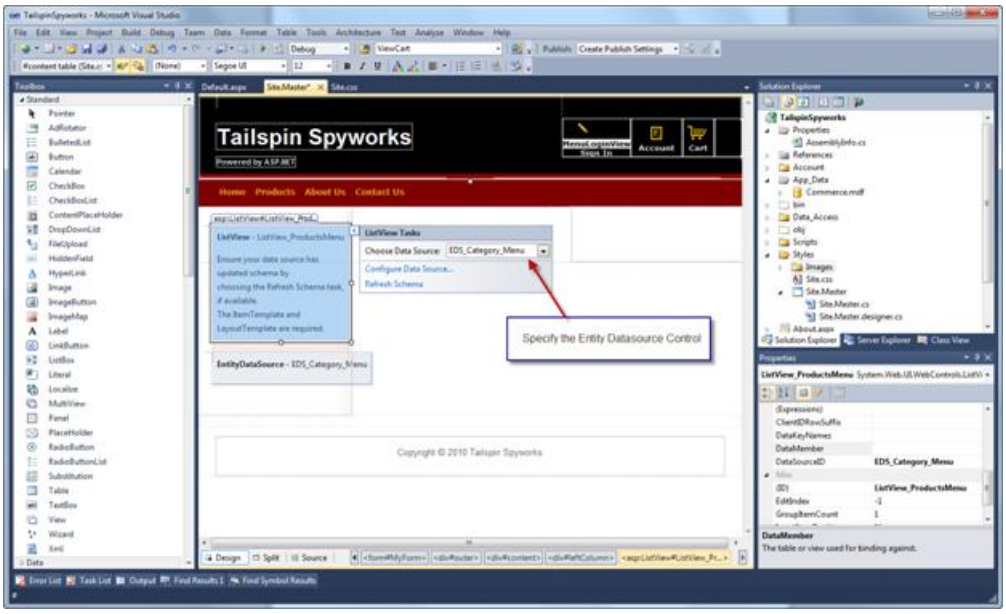


Select the CommerceEntities Connection that was created for us when we created the Entity Data Source Model for our Commerce Database and click "Next".



Select the "Categories" Entity set name and leave the rest of the options as default. Click "Finish".

Now let's set the ID property of the ListView control instance that we placed on our page to ListView_ProductsMenu and activate its helper.



Though we could use control options to format the data item display and formatting, our menu creation will only require simple markup so we will enter the code in the source view.

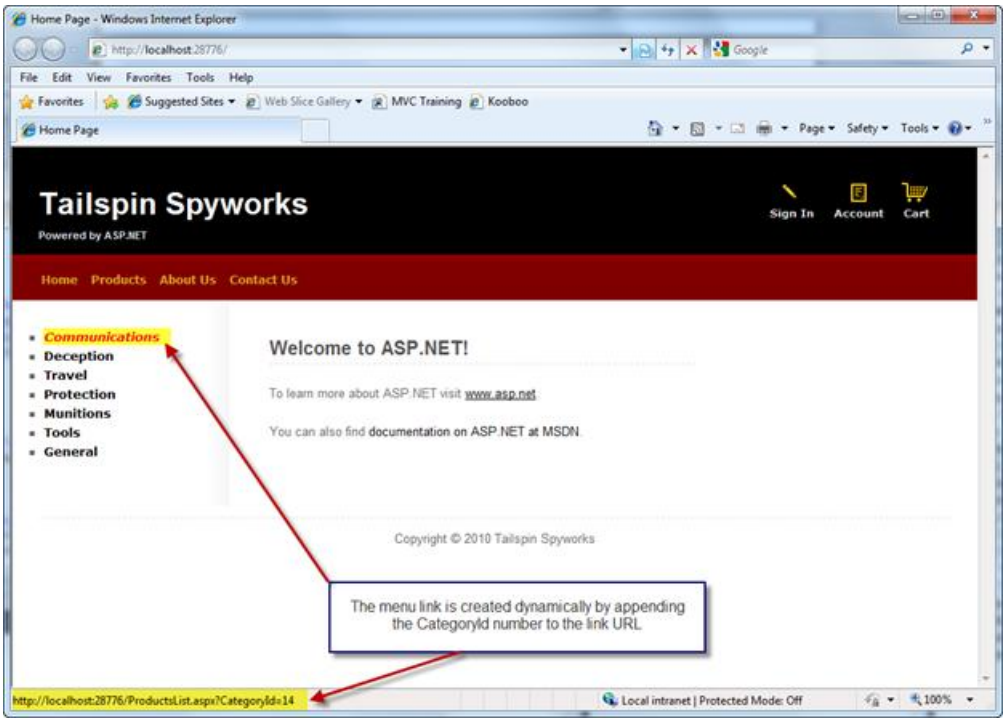
```
<asp:ListView ID="ListView_ProductsMenu" runat="server" DataKeyNames="CategoryID"
DataSourceID="EDS_Category_Menu">
   <EmptyDataTemplate>No Menu Items.</EmptyDataTemplate>
   <ItemSeparatorTemplate></ItemSeparatorTemplate>
   <ItemTemplate>
      <li>
         <a href='<%# VirtualPathUtility.ToAbsolute("~/ProductsList.aspx?CategoryId=" +
                                 Eval("CategoryID")) %>'><%# Eval("CategoryName") %></a>
      </li>
   </ItemTemplate>
   <LayoutTemplate>
      <ul ID="itemPlaceholderContainer" runat="server">
         <li runat="server" id="itemPlaceholder" />
      </ul>
      <div>
      </div>
   </LayoutTemplate>
```

Please note the "Eval" statement : <%# Eval("CategoryName") %>

The ASP.NET syntax <%# %> is a shorthand convention that instructs the runtime to execute whatever is contained within and output the results "in Line".

The statement Eval("CategoryName") instructs that, for the current entry in the bound collection of data items, fetch the value of the Entity Model item names "CategoryName". This is concise syntax for a very powerful feature.

Lets run the application now.



Note that our product category menu is now displayed and when we hover over one of the category menu items we can see the menu item link points to a page we have yet to implement named ProductsList.aspx and that we have built a dynamic query string argument that contains the category id.

PREVIOUS NEXT

Part 4: Listing Products

2/20/2020 • 4 minutes to read • Edit Online

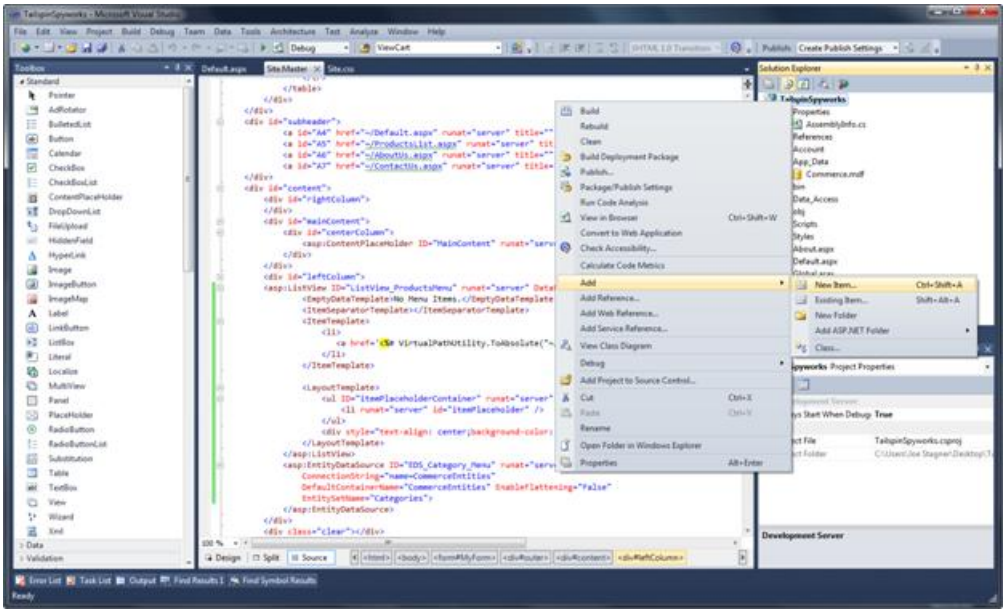
by Joe Stagner

Tailspin Spyworks demonstrates how extraordinarily simple it is to create powerful, scalable applications for the .NET platform. It shows off how to use the great new features in ASP.NET 4 to build an online store, including shopping, checkout, and administration.

This tutorial series details all of the steps taken to build the Tailspin Spyworks sample application. Part 4 covers listing products with the GridView control.

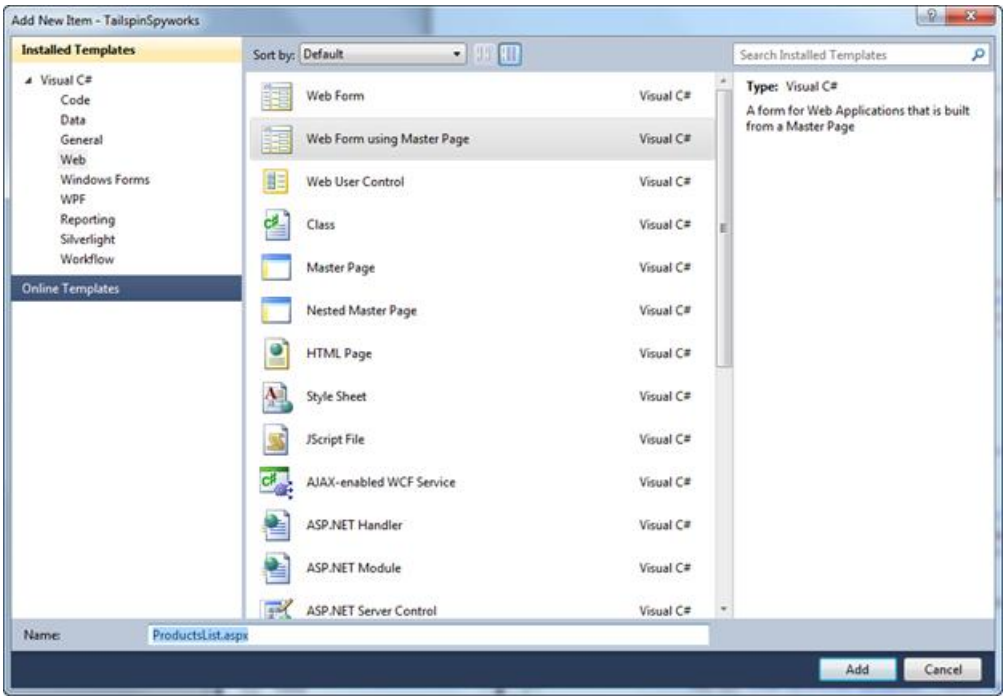
Listing Products with the GridView Control

Let's begin implementing our ProductsList.aspx page by "Right Clicking" on our solution and selecting "Add" and "New Item".

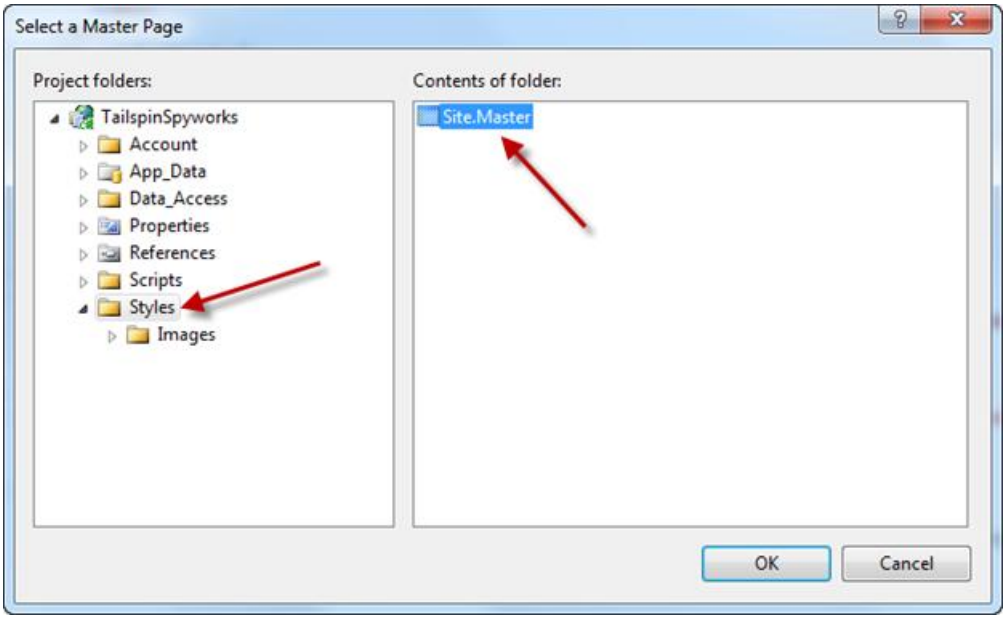


Choose "Web Form Using Master Page" and enter a page name of ProductsList.aspx".

Click "Add".

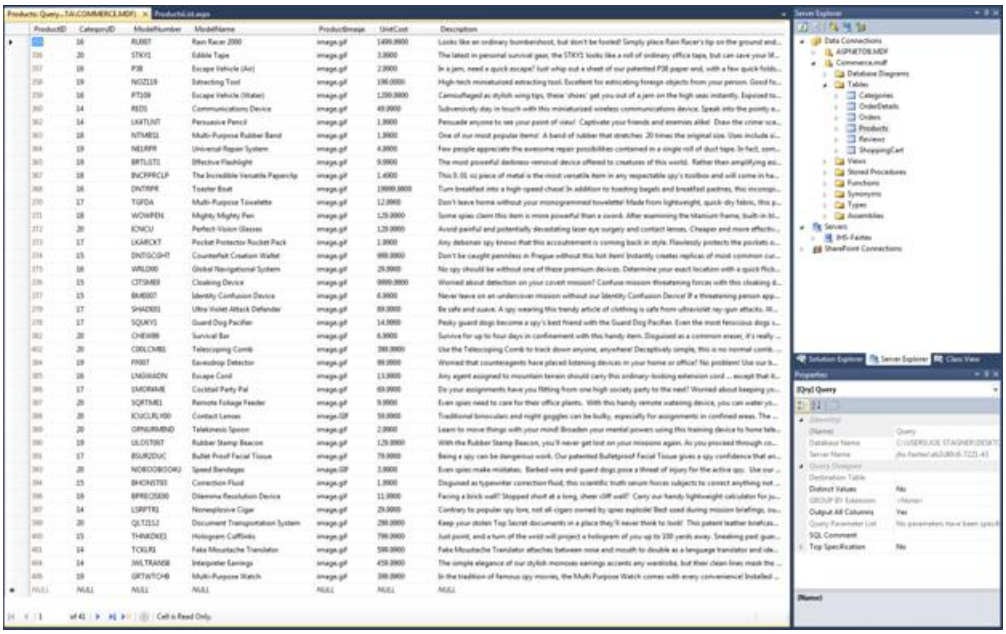


Next choose the "Styles" folder where we placed the Site.Master page and select it from the "Contents of folder" window.



Click "Ok" to create the page.

Our database is populated with product data as seen below.



After our page is created we'll again use an Entity Data Source to access that product data, but in this instance we need to select the Product Entities and we need to restrict the items that are returned to only those for the selected Category.

To accomplish this we'll tell the EntityDataSource to Auto Generate the WHERE clause and we'll specify the WhereParameter.

You'll recall that when we created the Menu Items in our "Product Category Menu" we dynamically built the link by adding the CategoryID to the QueryString for each link. We will tell the Entity Data Source to derive the WHERE parameter from that QueryString parameter.

```
<asp:EntityDataSource ID="EDS_ProductsByCategory" runat="server"
                      EnableFlattening="False" AutoGenerateWhereClause="True"
                      ConnectionString="name=CommerceEntities"
                      DefaultContainerName="CommerceEntities"
                      EntitySetName="Products">
<WhereParameters>
        <asp:QueryStringParameter Name="CategoryID"
                                  QueryStringField="Category Id"
                                  Type="Int32" />
       </WhereParameters>
</asp:EntityDataSource>
```

Next, we'll configure the ListView control to display a list of products. To create an optimal shopping experience we'll compact several concise features into each individual product displayed in our ListVew.

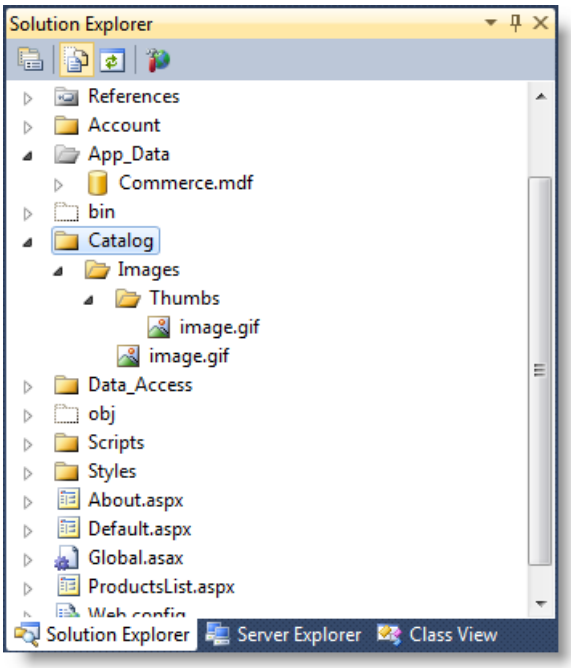
- The product name will be a link to the product's detail view.
- The product's price will be displayed.
- An image of the product will be displayed and we'll dynamically select the image from a catalog images directory in our application.
- We will include a link to immediately add the specific product to the shopping cart.

Here is the markup for our ListView control instance.

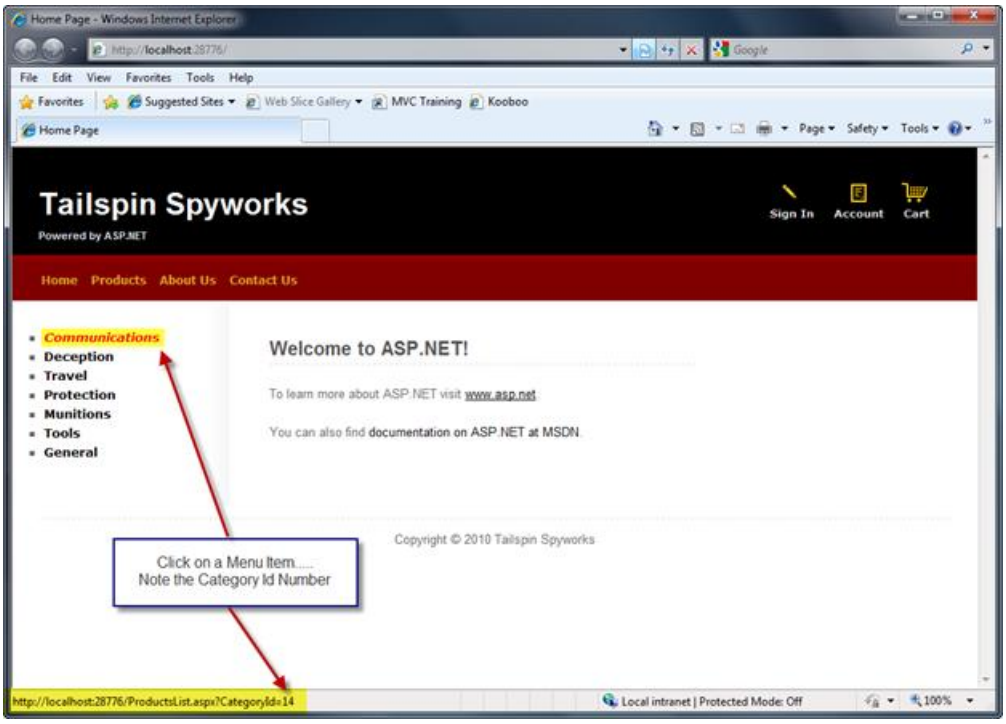
```
<asp:ListView ID="ListView_Products" runat="server"
              DataKeyNames="ProductID"
              DataSourceID="EDS_ProductsByCategory"
              GroupItemCount="2">
   <EmptyDataTemplate>
      <table runat="server">
        <tr>
          <td>No data was returned.</td>
        </tr>
     </table>
  </EmptyDataTemplate>
  <EmptyItemTemplate>
     <td runat="server" />
  </EmptyItemTemplate>
  <GroupTemplate>
    <tr ID="itemPlaceholderContainer" runat="server">
      <td ID="itemPlaceholder" runat="server"></td>
    </tr>
  </GroupTemplate>
  <ItemTemplate>
    <td runat="server">
      <table border="0" width="300">
        <tr>
          <td> </td>
          <td>
            <a href='ProductDetails.aspx?productID=<%# Eval("ProductID") %>'>
               <image src='Catalog/Images/Thumbs/<%# Eval("ProductImage") %>'
                      width="100" height="75" border="0">
            </a>  
          </td>
          <td>
            <a href='ProductDetails.aspx?productID=<%# Eval("ProductID") %>'><span
               class="ProductListHead"><%# Eval("ModelName") %></span><br>
            </a>
            <span class="ProductListItem">
              <b>Special Price: </b><%# Eval("UnitCost", "{0:c}")%>
            </span><br />
            <a href='AddToCart.aspx?productID=<%# Eval("ProductID") %>'>
               <span class="ProductListItem"><b>Add To Cart<b></font></span>
            </a>
          </td>
        </tr>
      </table>
    </td>
  </ItemTemplate>
  <LayoutTemplate>
    <table runat="server">
      <tr runat="server">
        <td runat="server">
          <table ID="groupPlaceholderContainer" runat="server">
            <tr ID="groupPlaceholder" runat="server">></tr>
          </table>
        </td>
      </tr>
      <tr runat="server">><td runat="server">></td>></tr>
    </table>
  </LayoutTemplate>
</asp:ListView>
```

We are dynamically building several links for each displayed product.

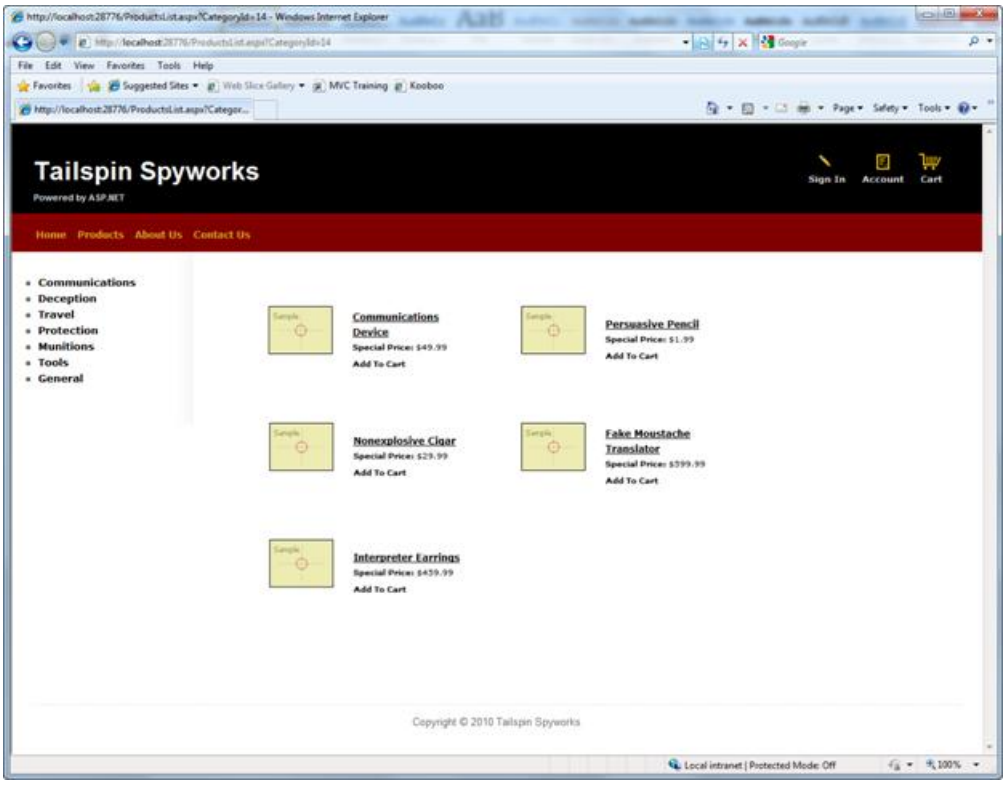
Also, before we test own new page we need to create the directory structure for the product catalog images as follows.



Once our product images are accessible we can test our product list page.



From the site's home page, click on one of the Category List Links.



Now we need to implement the ProductDetails.aspx page and the AddToCart functionality.

Use File->New to create a page name ProductDetails.aspx using the site Master Page as we did previously.

We will again use an EntityDataSource control to access the specific product record in the database and we will use an ASP.NET FormView control to display the product data as follows.

```
<asp:FormView ID="FormView_Product" runat="server" DataKeyNames="ProductID"
                                                          DataSourceID="EDS_Product">
  <ItemTemplate>
    <div class="ContentHead"><%# Eval("ModelName") %></div><br />
      <table  border="0">
        <tr>
          <td>
            <img src='Catalog/Images/<%# Eval("ProductImage") %>'  border="0"
                                                     alt='<%# Eval("ModelName") %>' />
          </td>
          <td><%# Eval("Description") %>
            <br /><br /><br />
          </td>
        </tr>
      </table>
      <span class="UnitCost"><b>Your Price:</b> <%# Eval("UnitCost", "{0:c}")%>
      <br />
      <span class="ModelNumber">
        <b>Model Number:</b> <%# Eval("ModelNumber") %>
      </span><br />
      <a href='AddToCart.aspx?ProductID=
        <%# Eval("ProductID") %>'>
        <img id="Img1" src="~/Styles/Images/add_to_cart.gif" runat="server"
             alt="" />
      </a>
      <br /><br />
    </ItemTemplate>
  </asp:FormView>
  <asp:EntityDataSource ID="EDS_Product" runat="server" AutoGenerateWhereClause="True"
                        EnableFlattening="False"
                        ConnectionString="name=CommerceEntities"
                        DefaultContainerName="CommerceEntities"
                        EntitySetName="Products"
                        EntityTypeFilter=""
                        Select="" Where="">
    <WhereParameters>
      <asp:QueryStringParameter Name="ProductID"
                                QueryStringField="productID"  Type="Int32" />
    </WhereParameters>
  </asp:EntityDataSource>
```

Don't worry if the formatting looks a bit funny to you. The markup above leaves room in the display layout for a couple of features we'll implement later on.

The Shopping Cart will represent the more complex logic in our application. To get started, use File->New to create a page called MyShoppingCart.aspx.

Note that we are not choosing the name ShoppingCart.aspx.

Our database contains a table named "ShoppingCart". When we generated an Entity Data Model a class was created for each table in the database. Therefore, the Entity Data Model generated an Entity Class named "ShoppingCart". We could edit the model so that we could use that name for our shopping cart implementation or extend it for our needs, but we will opt instead to simply select a name that will avoid the conflict.

It's also worth noting that we will be creating a simple shopping cart and embedding the shopping cart logic with the shopping cart display. We might also choose to implement our shopping cart in a completely separate Business Layer.

Part 5: Business Logic

2/20/2020 • 10 minutes to read • Edit Online

by Joe Stagner

Tailspin Spyworks demonstrates how extraordinarily simple it is to create powerful, scalable applications for the .NET platform. It shows off how to use the great new features in ASP.NET 4 to build an online store, including shopping, checkout, and administration.

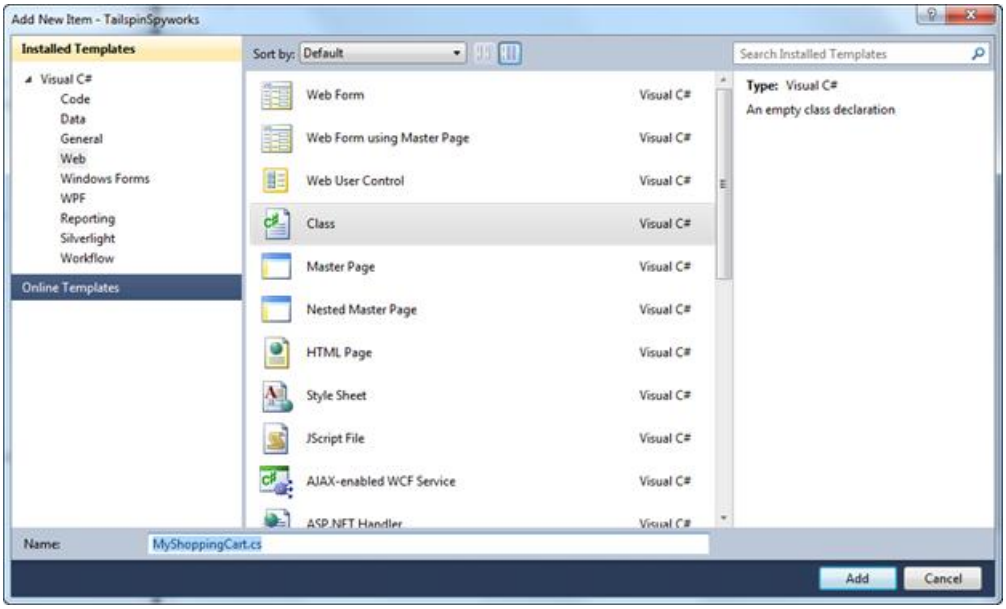
This tutorial series details all of the steps taken to build the Tailspin Spyworks sample application. Part 5 adds some business logic.

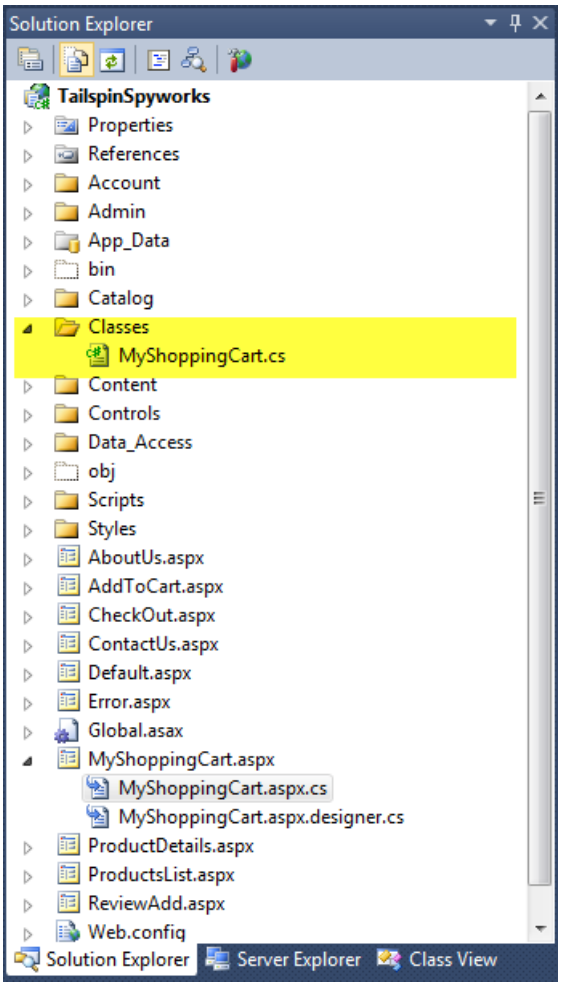
Adding Some Business Logic

We want our shopping experience to be available whenever someone visits our web site. Visitors will be able to browse and add items to the shopping cart even if they are not registered or logged in. When they are ready to check out they will be given the option to authenticate and if they are not yet members they will be able to create an account.

This means that we will need to implement the logic to convert the shopping cart from an anonymous state to a "Registered User" state.

Let's create a directory named "Classes" then Right-Click on the folder and create a new "Class" file named MyShoppingCart.cs





As previously mentioned we will be extending the class that implements the MyShoppingCart.aspx page and we will do this using .NET's powerful "Partial Class" construct.

The generated call for our MyShoppingCart.aspx.cf file looks like this.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace TailspinSpyworks
{
    public partial class MyShoppingCart : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }
    }
}
```

Note the use of the "partial" keyword.

The class file that we just generated looks like this.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace TailspinSpyworks.Classes
{
    public class MyShoppingCart
    {
    }
}
```

We will merge our implementations by adding the partial keyword to this file as well.

Our new class file now looks like this.

```
namespace TailspinSpyworks.Classes
{
    public partial class MyShoppingCart
    {
    }
}
```

The first method that we will add to our class is the "AddItem" method. This is the method that will ultimately be called when the user clicks on the "Add to Art" links on the Product List and Product Details pages.

Append the following to the using statements at the top of the page.

```
using TailspinSpyworks.Data_Access;
```

And add this method to the MyShoppingCart class.

```
//-------------------------------------------------------------------------------------+
public void AddItem(string cartID, int productID, int quantity)
{
  using (CommerceEntities db = new CommerceEntities())
    {
    try
      {
      var myItem = (from c in db.ShoppingCarts where c.CartID == cartID &&
                              c.ProductID == productID select c).FirstOrDefault();
      if(myItem == null)
        {
        ShoppingCart cartadd = new ShoppingCart();
        cartadd.CartID = cartID;
        cartadd.Quantity = quantity;
        cartadd.ProductID = productID;
        cartadd.DateCreated = DateTime.Now;
        db.ShoppingCarts.AddObject(cartadd);
        }
      else
        {
        myItem.Quantity += quantity;
        }
      db.SaveChanges();
      }
    catch (Exception exp)
      {
      throw new Exception("ERROR: Unable to Add Item to Cart - " +
                                                        exp.Message.ToString(), exp);
      }
   }
}
```

We are using LINQ to Entities to see if the item is already in the cart. If so, we update the order quantity of the item, otherwise we create a new entry for the selected item

In order to call this method we will implement an AddToCart.aspx page that not only class this method but then displayed the current shopping a=cart after the item has been added.

Right-Click on the solution name in the solution explorer and add and new page named AddToCart.aspx as we have done previously.

While we could use this page to display interim results like low stock issues, etc, in our implementation, the page will not actually render, but rather call the "Add" logic and redirect.

To accomplish this we'll add the following code to the Page_Load event.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Diagnostics;

namespace TailspinSpyworks
{
    public partial class AddToCart : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            string rawId = Request.QueryString["ProductID"];
            int productId;
            if (!String.IsNullOrEmpty(rawId) && Int32.TryParse(rawId, out productId))
            {
                MyShoppingCart usersShoppingCart = new MyShoppingCart();
                String cartId = usersShoppingCart.GetShoppingCartId();
                usersShoppingCart.AddItem(cartId, productId, 1);
            }
            else
            {
                Debug.Fail("ERROR : We should never get to AddToCart.aspx
                                                    without a ProductId.");
                throw new Exception("ERROR : It is illegal to load AddToCart.aspx
                                                    without setting a ProductId.");
            }
            Response.Redirect("MyShoppingCart.aspx");
        }
    }
}
```

Note that we are retrieving the product to add to the shopping cart from a QueryString parameter and calling the AddItem method of our class.

Assuming no errors are encountered control is passed to the SHoppingCart.aspx page which we will fully implement next. If there should be an error we throw an exception.

Currently we have not yet implemented a global error handler so this exception would go unhandled by our application but we will remedy this shortly.

Note also the use of the statement Debug.Fail() (available via `using System.Diagnostics;)`

Is the application is running inside the debugger, this method will display a detailed dialog with information about the applications state along with the error message that we specify.

When running in production the Debug.Fail() statement is ignored.

You will note in the code above a call to a method in our shopping cart class names "GetShoppingCartId".

Add the code to implement the method as follows.

Note that we've also added update and checkout buttons and a label where we can display the cart "total".

```
public const string CartId = "TailSpinSpyWorks_CartID";

//-----------------------------------------------------------------------------------------+
public String GetShoppingCartId()
{
  if (Session[CartId] == null)
     {
     Session[CartId] = System.Web.HttpContext.Current.Request.IsAuthenticated ?
                                        User.Identity.Name : Guid.NewGuid().ToString();
     }
  return Session[CartId].ToString();
}
```

We can now add items to our shopping cart but we have not implemented the logic to display the cart after a product has been added.

So, in the MyShoppingCart.aspx page we'll add an EntityDataSource control and a GridVire control as follows.

```
<div id="ShoppingCartTitle" runat="server" class="ContentHead">Shopping Cart</div>
<asp:GridView ID="MyList" runat="server" AutoGenerateColumns="False" ShowFooter="True"
                          GridLines="Vertical" CellPadding="4"
                          DataSourceID="EDS_Cart"
                          DataKeyNames="ProductID,UnitCost,Quantity"
                          CssClass="CartListItem">
  <AlternatingRowStyle CssClass="CartListItemAlt" />
  <Columns>
    <asp:BoundField DataField="ProductID" HeaderText="Product ID" ReadOnly="True"
                                          SortExpression="ProductID"  />
    <asp:BoundField DataField="ModelNumber" HeaderText="Model Number"
                                            SortExpression="ModelNumber" />
    <asp:BoundField DataField="ModelName" HeaderText="Model Name"
                                          SortExpression="ModelName"  />
    <asp:BoundField DataField="UnitCost" HeaderText="Unit Cost" ReadOnly="True"
                                         SortExpression="UnitCost"
                                         DataFormatString="{0:c}" />
    <asp:TemplateField>
      <HeaderTemplate>Quantity</HeaderTemplate>
      <ItemTemplate>
         <asp:TextBox ID="PurchaseQuantity" Width="40" runat="server"
                      Text='<%# Bind("Quantity") %>'></asp:TextBox>
      </ItemTemplate>
    </asp:TemplateField>
    <asp:TemplateField>
      <HeaderTemplate>Item Total</HeaderTemplate>
      <ItemTemplate>
        <%# (Convert.ToDouble(Eval("Quantity")) *
             Convert.ToDouble(Eval("UnitCost")))%>
      </ItemTemplate>
    </asp:TemplateField>
    <asp:TemplateField>
    <HeaderTemplate>Remove Item</HeaderTemplate>
      <ItemTemplate>
        <center>
          <asp:CheckBox id="Remove" runat="server" />
        </center>
      </ItemTemplate>
    </asp:TemplateField>
  </Columns>
  <FooterStyle CssClass="CartListFooter"/>
  <HeaderStyle  CssClass="CartListHead" />
</asp:GridView>

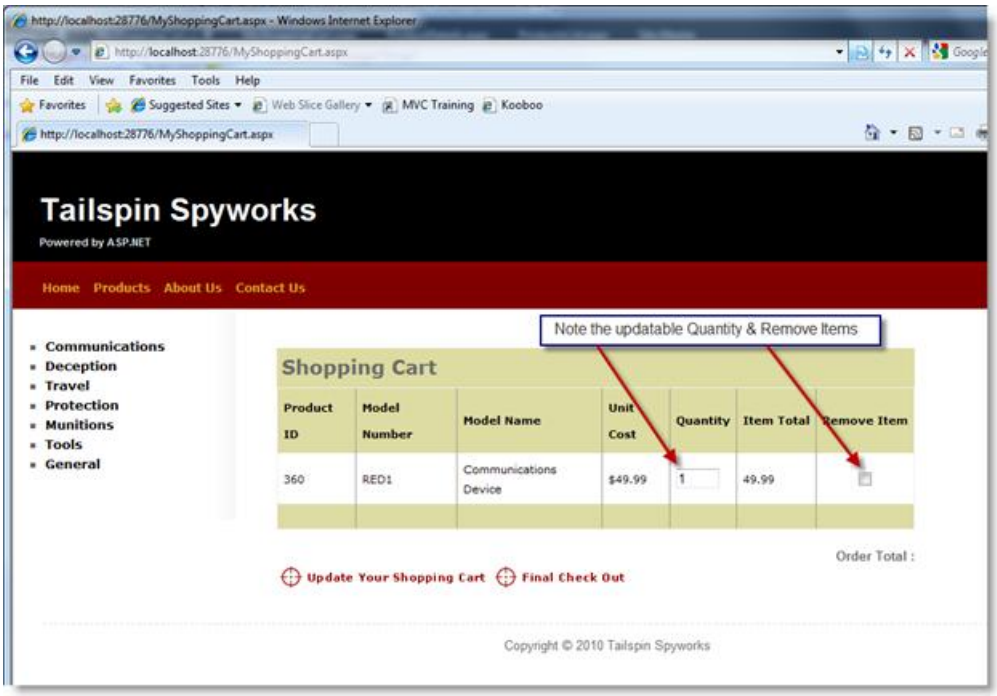
<div>
  <strong>
    <asp:Label ID="LabelTotalText" runat="server" Text="Order Total : ">
    </asp:Label>
```

```
    <asp:Label CssClass="NormalBold" id="lblTotal" runat="server"
                                                EnableViewState="false">
    </asp:Label>
  </strong>
</div>
<br />
<asp:imagebutton id="UpdateBtn" runat="server" ImageURL="Styles/Images/update_cart.gif"
                               onclick="UpdateBtn_Click"></asp:imagebutton>
<asp:imagebutton id="CheckoutBtn" runat="server"
                                 ImageURL="Styles/Images/final_checkout.gif"
                                 PostBackUrl="~/CheckOut.aspx">
</asp:imagebutton>
<asp:EntityDataSource ID="EDS_Cart" runat="server"
                      ConnectionString="name=CommerceEntities"
                      DefaultContainerName="CommerceEntities" EnableFlattening="False"
                      EnableUpdate="True" EntitySetName="ViewCarts"
                      AutoGenerateWhereClause="True" EntityTypeFilter="" Select=""
                      Where="">
  <WhereParameters>
    <asp:SessionParameter Name="CartID" DefaultValue="0"
                                       SessionField="TailSpinSpyWorks_CartID" />
  </WhereParameters>
</asp:EntityDataSource>
```

Call up the form in the designer so that you can double click on the Update Cart button and generate the click event handler that is specified in the declaration in the markup.

We'll implement the details later but doing this will let us build and run our application without errors.

When you run the application and add an item to the shopping cart you will see this.



Note that we have deviated from the "default" grid display by implementing three custom columns.

The first is an Editable, "Bound" field for the Quantity:

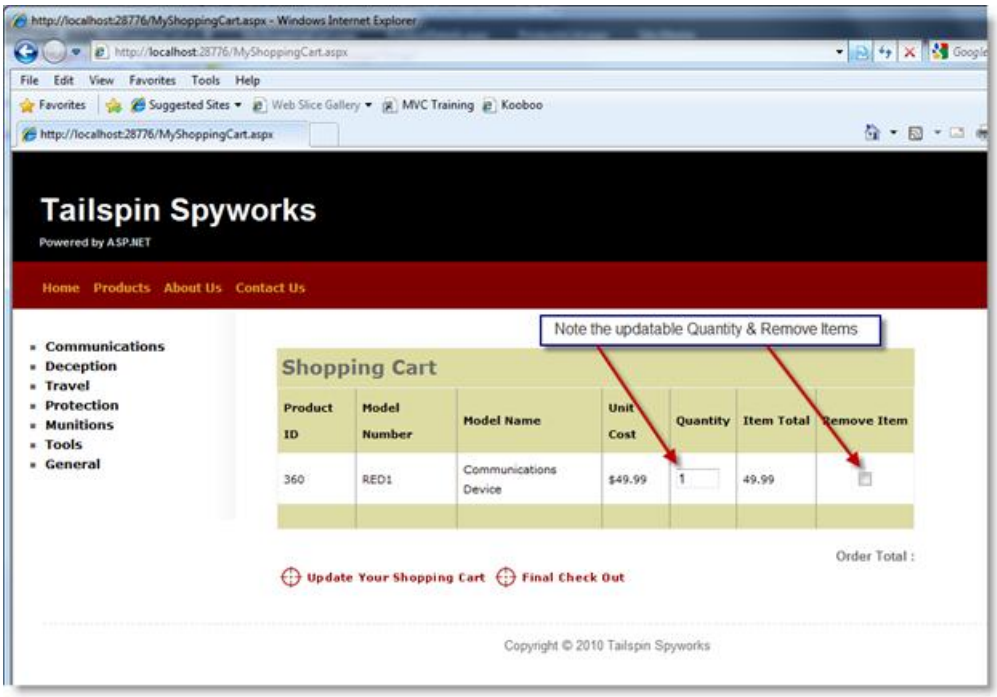
```
<asp:TemplateField>
      <HeaderTemplate>Quantity</HeaderTemplate>
      <ItemTemplate>
         <asp:TextBox ID="PurchaseQuantity" Width="40" runat="server"
                      Text='<%# Bind("Quantity") %>'></asp:TextBox>
      </ItemTemplate>
    </asp:TemplateField>
```

The next is a "calculated" column that displays the line item total (the item cost times the quantity to be ordered):

```
<asp:TemplateField>
      <HeaderTemplate>Item Total</HeaderTemplate>
      <ItemTemplate>
        <%# (Convert.ToDouble(Eval("Quantity")) *
             Convert.ToDouble(Eval("UnitCost")))%>
      </ItemTemplate>
    </asp:TemplateField>
```

Lastly we have a custom column that contains a CheckBox control that the user will use to indicate that the item should be removed from the shopping chart.

```
<asp:TemplateField>
    <HeaderTemplate>Remove Item</HeaderTemplate>
      <ItemTemplate>
        <center>
          <asp:CheckBox id="Remove" runat="server" />
        </center>
      </ItemTemplate>
    </asp:TemplateField>
```



As you can see, the Order Total line is empty so let's add some logic to calculate the Order Total.

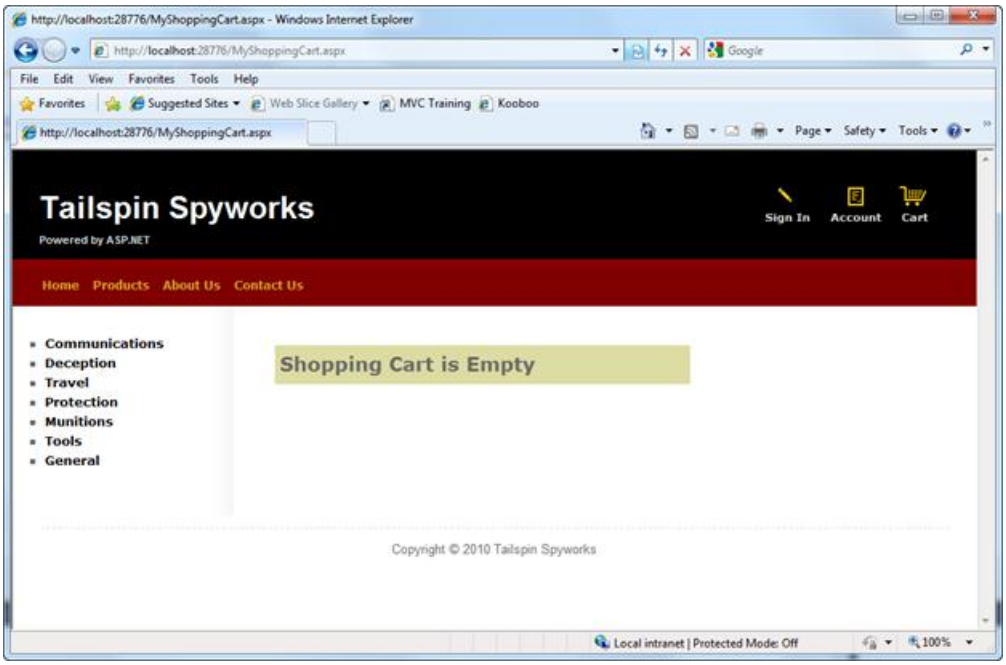
We'll first implement a "GetTotal" method to our MyShoppingCart Class.

In the MyShoppingCart.cs file add the following code.

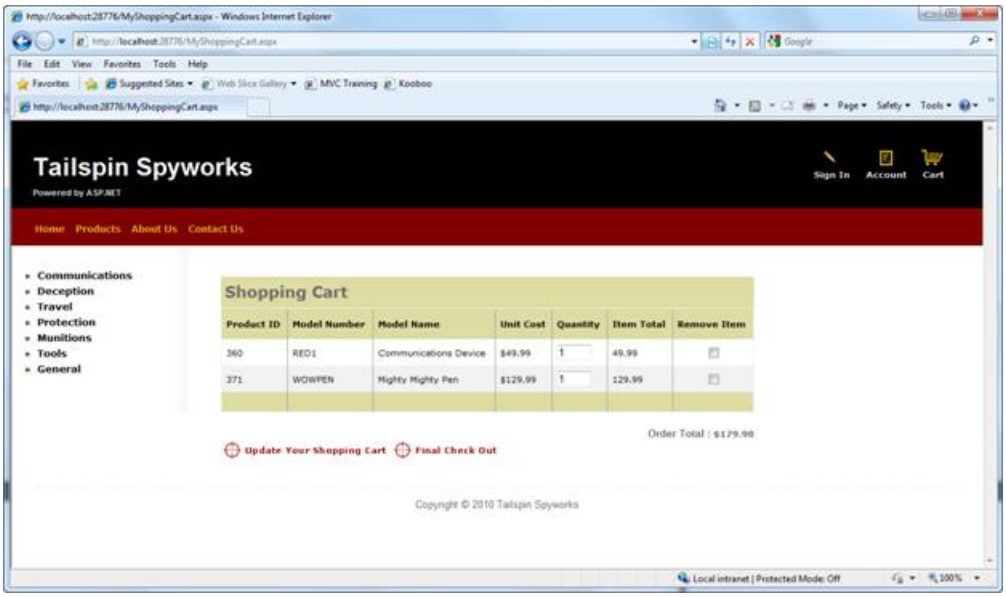
```
//-----------------------------------------------------------------------------------------+
public decimal GetTotal(string cartID)
{
    using (CommerceEntities db = new CommerceEntities())
    {
        decimal cartTotal = 0;
        try
        {
            var myCart = (from c in db.ViewCarts where c.CartID == cartID select c);
            if (myCart.Count() > 0)
            {
                cartTotal = myCart.Sum(od => (decimal)od.Quantity * (decimal)od.UnitCost);
            }
        }
        catch (Exception exp)
        {
            throw new Exception("ERROR: Unable to Calculate Order Total - " +
            exp.Message.ToString(), exp);
        }
        return (cartTotal);
     }
}
```

Then in the Page_Load event handler we'll can call our GetTotal method. At the same time we'll add a test to see if the shopping cart is empty and adjust the display accordingly if it is.

Now if the shopping cart is empty we get this:



And if not, we see our total.



However, this page is not yet complete.

We will need additional logic to recalculate the shopping cart by removing items marked for removal and by determining new quantity values as some may have been changed in the grid by the user.

Lets add a "RemoveItem" method to our shopping cart class in MyShoppingCart.cs to handle the case when a user marks an item for removal.

```
//-------------------------------------------------------------------------------------+
public void RemoveItem(string cartID, int  productID)
{
    using (CommerceEntities db = new CommerceEntities())
    {
        try
        {
            var myItem = (from c in db.ShoppingCarts where c.CartID == cartID &&
                          c.ProductID == productID select c).FirstOrDefault();
            if (myItem != null)
            {
                db.DeleteObject(myItem);
                db.SaveChanges();
            }
        }
        catch (Exception exp)
        {
            throw new Exception("ERROR: Unable to Remove Cart Item - " +
                                exp.Message.ToString(), exp);
        }
    }
}
```

Now let's ad a method to handle the circumstance when a user simply changes the quality to be ordered in the GridView.

```
//-----------------------------------------------------------------------------------------+
public void UpdateItem(string cartID, int productID, int quantity)
{
    using (CommerceEntities db = new CommerceEntities())
    {
        try
        {
            var myItem = (from c in db.ShoppingCarts where c.CartID == cartID &&
                    c.ProductID == productID select c).FirstOrDefault();
            if (myItem != null)
            {
                myItem.Quantity = quantity;
                db.SaveChanges();
            }
        }
        catch (Exception exp)
        {
            throw new Exception("ERROR: Unable to Update Cart Item - " +
                                exp.Message.ToString(), exp);
        }
    }
}
```

With the basic Remove and Update features in place we can implement the logic that actually updates the shopping cart in the database. (In MyShoppingCart.cs)

```
//-----------------------------------------------------------------------------------------+
public void UpdateShoppingCartDatabase(String cartId,
                                       ShoppingCartUpdates[] CartItemUpdates)
{
  using (CommerceEntities db = new CommerceEntities())
    {
    try
      {
      int CartItemCOunt = CartItemUpdates.Count();
      var myCart = (from c in db.ViewCarts where c.CartID == cartId select c);
      foreach (var cartItem in myCart)
        {
        // Iterate through all rows within shopping cart list
        for (int i = 0; i < CartItemCOunt; i++)
          {
          if (cartItem.ProductID == CartItemUpdates[i].ProductId)
             {
             if (CartItemUpdates[i].PurchaseQantity < 1 ||
   CartItemUpdates[i].RemoveItem == true)
                {
                RemoveItem(cartId, cartItem.ProductID);
                }
             else
                {
                UpdateItem(cartId, cartItem.ProductID,
                                   CartItemUpdates[i].PurchaseQantity);
                }
              }
            }
          }
        }
      catch (Exception exp)
        {
        throw new Exception("ERROR: Unable to Update Cart Database - " +
                            exp.Message.ToString(), exp);
        }
    }
}
```

You'll note that this method expects two parameters. One is the shopping cart Id and the other is an array of objects of user defined type.

So as to minimize the dependency of our logic on user interface specifics, we've defined a data structure that we can use to pass the shopping cart items to our code without our method needing to directly access the GridView control.

```
public struct ShoppingCartUpdates
{
    public int ProductId;
    public int PurchaseQantity;
    public bool RemoveItem;
}
```

In our MyShoppingCart.aspx.cs file we can use this structure in our Update Button Click Event handler as follows. Note that in addition to updating the cart we will update the cart total as well.

```
//-----------------------------------------------------------------------------------+
protected void UpdateBtn_Click(object sender, ImageClickEventArgs e)
{
  MyShoppingCart usersShoppingCart = new MyShoppingCart();
  String cartId = usersShoppingCart.GetShoppingCartId();

  ShoppingCartUpdates[] cartUpdates = new ShoppingCartUpdates[MyList.Rows.Count];
  for (int i = 0; i < MyList.Rows.Count; i++)
    {
    IOrderedDictionary rowValues = new OrderedDictionary();
    rowValues = GetValues(MyList.Rows[i]);
    cartUpdates[i].ProductId =  Convert.ToInt32(rowValues["ProductID"]);
    cartUpdates[i].PurchaseQantity = Convert.ToInt32(rowValues["Quantity"]);

    CheckBox cbRemove = new CheckBox();
    cbRemove = (CheckBox)MyList.Rows[i].FindControl("Remove");
    cartUpdates[i].RemoveItem = cbRemove.Checked;
    }

   usersShoppingCart.UpdateShoppingCartDatabase(cartId, cartUpdates);
   MyList.DataBind();
   lblTotal.Text = String.Format("{0:c}", usersShoppingCart.GetTotal(cartId));
}
```

Note with particular interest this line of code:

```
rowValues = GetValues(MyList.Rows[i]);
```

GetValues() is a special helper function that we will implement in MyShoppingCart.aspx.cs as follows.

```
//-----------------------------------------------------------------------------------------+
public static IOrderedDictionary GetValues(GridViewRow row)
{
  IOrderedDictionary values = new OrderedDictionary();
  foreach (DataControlFieldCell cell in row.Cells)
    {
    if (cell.Visible)
      {
      // Extract values from the cell
      cell.ContainingField.ExtractValuesFromCell(values, cell, row.RowState, true);
      }
    }
    return values;
}
```

This provides a clean way to access the values of the bound elements in our GridView control. Since our "Remove Item" CheckBox Control is not bound we'll access it via the FindControl() method.

At this stage in your project's development we are getting ready to implement the checkout process.

Before doing so let's use Visual Studio to generate the membership database and add a user to the membership repository.

PREVIOUS NEXT

Part 6: ASP.NET Membership

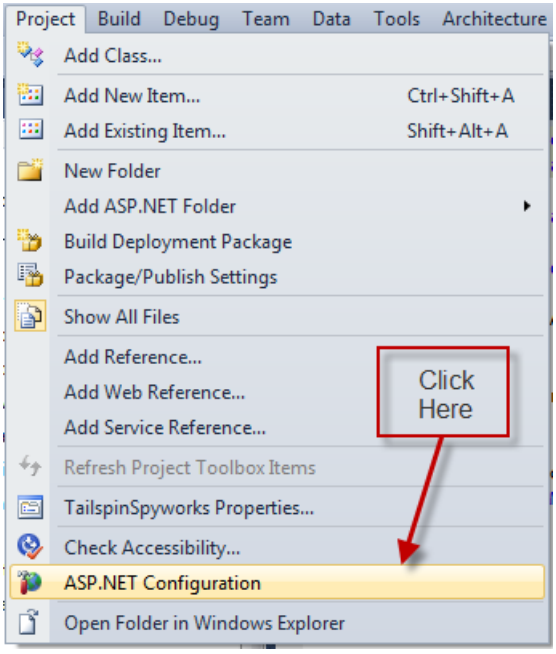
2/20/2020 • 5 minutes to read • Edit Online

by Joe Stagner

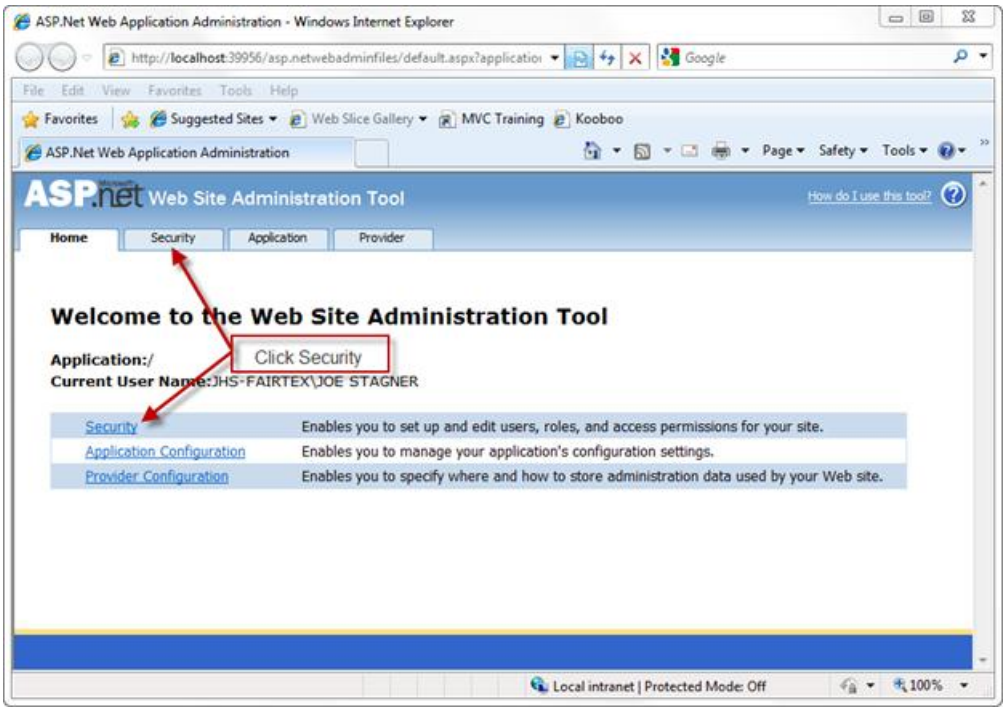
Tailspin Spyworks demonstrates how extraordinarily simple it is to create powerful, scalable applications for the .NET platform. It shows off how to use the great new features in ASP.NET 4 to build an online store, including shopping, checkout, and administration.

This tutorial series details all of the steps taken to build the Tailspin Spyworks sample application. Part 6 adds ASP.NET Membership.

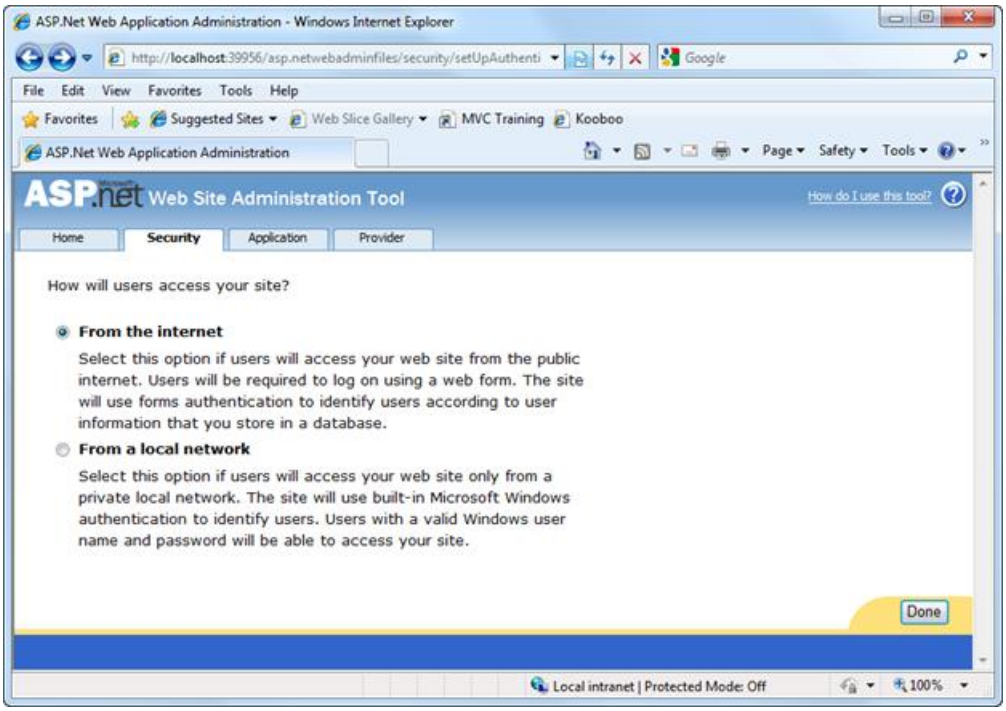
Working with ASP.NET Membership



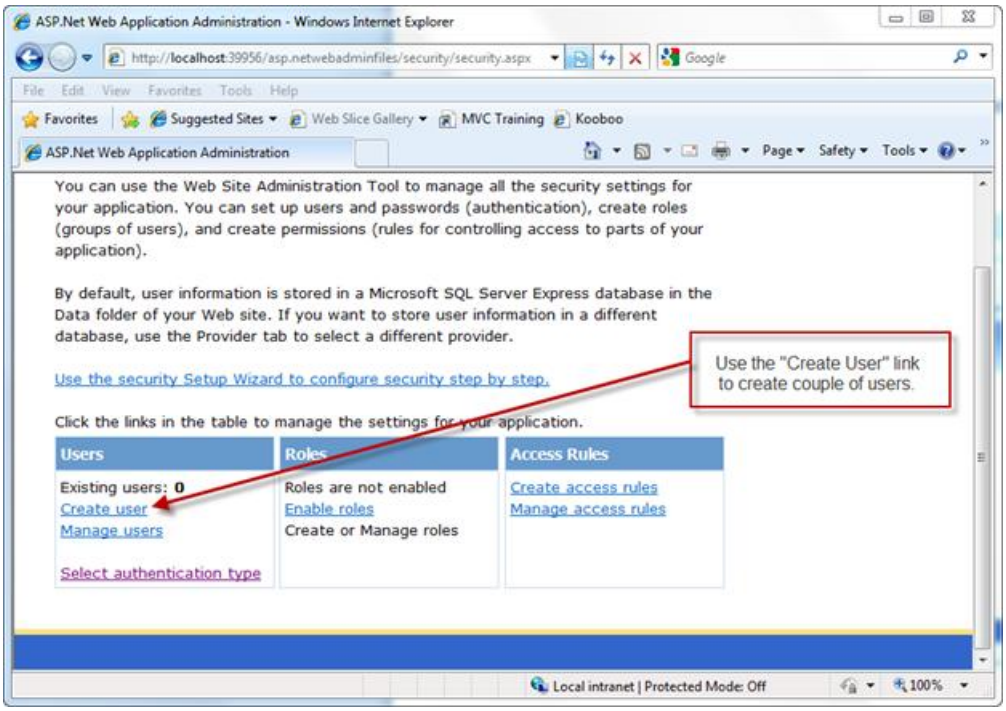
Click Security



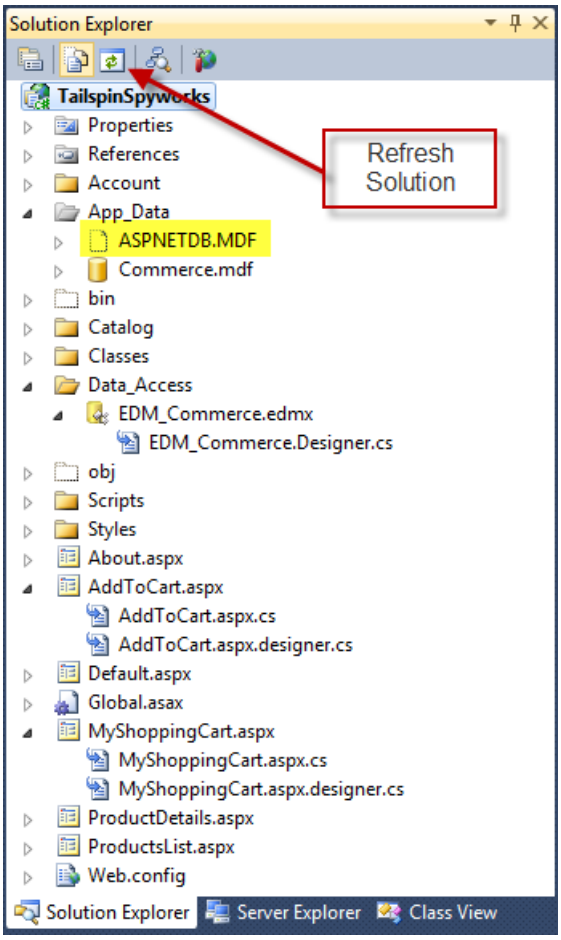
Make sure that we are using forms authentication.



Use the "Create User" link to create a couple of users.



When done, refer to the Solution Explorer window and refresh the view.



Note that the ASPNETDB.MDF fine has been created. This file contains the tables to support the core ASP.NET services like membership.

Now we can begin implementing the checkout process.

Begin by creating a CheckOut.aspx page.

The CheckOut.aspx page should only be available to users who are logged in so we will restrict access to logged in users and redirect users who are not logged in to the LogIn page.

To do this we'll add the following to the configuration section of our web.config file.

```
<location path="Checkout.aspx">
    <system.web>
      <authorization>
        <deny users="?" />
      </authorization>
    </system.web>
  </location>
```

The template for ASP.NET Web Forms applications automatically added an authentication section to our web.config file and established the default login page.

```
<authentication mode="Forms">
      <forms loginUrl="~/Account/Login.aspx" timeout="2880" />
    </authentication>
```

We must modify the Login.aspx code behind file to migrate an anonymous shopping cart when the user logs in. Change the Page_Load event as follows.

```
using System.Web.Security;

protected void Page_Load(object sender, EventArgs e)
{
  // If the user is not submitting their credentials
  // save refferer
  if (!Page.IsPostBack)
     {
     if (Page.Request.UrlReferrer != null)
        {
        Session["LoginReferrer"] = Page.Request.UrlReferrer.ToString();
        }
      }

  // User is logged in so log them out.
  if (User.Identity.IsAuthenticated)
     {
     FormsAuthentication.SignOut();
     Response.Redirect("~/");
     }
}
```

Then add a "LoggedIn" event handler like this to set the session name to the newly logged in user and change the temporary session id in the shopping cart to that of the user by calling the MigrateCart method in our MyShoppingCart class. (Implemented in the .cs file)

```
protected void LoginUser_LoggedIn(object sender, EventArgs e)
{
  MyShoppingCart usersShoppingCart = new MyShoppingCart();
  String cartId = usersShoppingCart.GetShoppingCartId();
  usersShoppingCart.MigrateCart(cartId, LoginUser.UserName);

  if(Session["LoginReferrer"] != null)
    {
    Response.Redirect(Session["LoginReferrer"].ToString());
    }

  Session["UserName"] = LoginUser.UserName;
}
```

Implement the MigrateCart() method like this.

```
//-------------------------------------------------------------------------------------+
public void MigrateCart(String oldCartId, String UserName)
{
  using (CommerceEntities db = new CommerceEntities())
    {
    try
      {
      var myShoppingCart = from cart in db.ShoppingCarts
                           where cart.CartID == oldCartId
                           select cart;

      foreach (ShoppingCart item in myShoppingCart)
        {
        item.CartID = UserName;
        }
      db.SaveChanges();
      Session[CartId] = UserName;
      }
    catch (Exception exp)
      {
      throw new Exception("ERROR: Unable to Migrate Shopping Cart - " +
                           exp.Message.ToString(), exp);
      }
    }
}
```

In checkout.aspx we'll use an EntityDataSource and a GridView in our check out page much as we did in our shopping cart page.

```
<div id="CheckOutHeader" runat="server" class="ContentHead">
  Review and Submit Your Order
</div>
<span id="Message" runat="server"><br />
   <asp:Label ID="LabelCartHeader" runat="server"
              Text="Please check all the information below to be sure it&#39;s correct.">
   </asp:Label>
</span><br />
<asp:GridView ID="MyList" runat="server" AutoGenerateColumns="False"
              DataKeyNames="ProductID,UnitCost,Quantity"
              DataSourceID="EDS_Cart"
              CellPadding="4" GridLines="Vertical" CssClass="CartListItem"
              onrowdatabound="MyList_RowDataBound" ShowFooter="True">
  <AlternatingRowStyle CssClass="CartListItemAlt" />
  <Columns>
    <asp:BoundField DataField="ProductID" HeaderText="Product ID" ReadOnly="True"
                    SortExpression="ProductID"  />
    <asp:BoundField DataField="ModelNumber" HeaderText="Model Number"
                    SortExpression="ModelNumber" />
    <asp:BoundField DataField="ModelName" HeaderText="Model Name"
                    SortExpression="ModelName" />
    <asp:BoundField DataField="UnitCost" HeaderText="Unit Cost" ReadOnly="True"
                    SortExpression="UnitCost" DataFormatString="{0:c}" />
    <asp:BoundField DataField="Quantity" HeaderText="Quantity" ReadOnly="True"
                    SortExpression="Quantity" />
    <asp:TemplateField>
      <HeaderTemplate>Item Total</HeaderTemplate>
      <ItemTemplate>
        <%# (Convert.ToDouble(Eval("Quantity")) * Convert.ToDouble(Eval("UnitCost")))%>
      </ItemTemplate>
    </asp:TemplateField>
  </Columns>
  <FooterStyle CssClass="CartListFooter"/>
  <HeaderStyle  CssClass="CartListHead" />
</asp:GridView>

<br />
<asp:imagebutton id="CheckoutBtn" runat="server" ImageURL="Styles/Images/submit.gif"
                                 onclick="CheckoutBtn_Click">
</asp:imagebutton>
<asp:EntityDataSource ID="EDS_Cart" runat="server"
                      ConnectionString="name=CommerceEntities"
                      DefaultContainerName="CommerceEntities"
                      EnableFlattening="False"
                      EnableUpdate="True"
                      EntitySetName="ViewCarts"
                      AutoGenerateWhereClause="True"
                      EntityTypeFilter=""
                      Select="" Where="">
   <WhereParameters>
      <asp:SessionParameter Name="CartID" DefaultValue="0"
                                          SessionField="TailSpinSpyWorks_CartID" />
   </WhereParameters>
</asp:EntityDataSource>
```

Note that our GridView control specifies an "ondatabound" event handler named MyList_RowDataBound so let's implement that event handler like this.

```
decimal _CartTotal = 0;

//-----------------------------------------------------------------------------------------+
protected void MyList_RowDataBound(object sender, GridViewRowEventArgs e)
{
  if (e.Row.RowType == DataControlRowType.DataRow)
     {
     TailspinSpyworks.Data_Access.ViewCart myCart = new Data_Access.ViewCart();
     myCart = (TailspinSpyworks.Data_Access.ViewCart)e.Row.DataItem;
     _CartTotal += myCart.UnitCost * myCart.Quantity;
     }
   else if (e.Row.RowType == DataControlRowType.Footer)
     {
     if (_CartTotal > 0)
        {
        CheckOutHeader.InnerText = "Review and Submit Your Order";
        LabelCartHeader.Text = "Please check all the information below to be sure
                                                                  it&#39;s correct.";
        CheckoutBtn.Visible = true;
        e.Row.Cells[5].Text = "Total: " + _CartTotal.ToString("C");
        }
     }
}
```

This method keeps a running total of the shopping cart as each row is bound and updates the bottom row of the GridView.

At this stage we have implemented a "review" presentation of the order to be placed.

Let's handle an empty cart scenario by adding a few lines of code to our Page_Load event:

```
protected void Page_Load(object sender, EventArgs e)
{
   CheckOutHeader.InnerText = "Your Shopping Cart is Empty";
   LabelCartHeader.Text = "";
   CheckoutBtn.Visible = false;
}
```

When the user clicks on the "Submit" button we will execute the following code in the Submit Button Click Event handler.

```
protected void CheckoutBtn_Click(object sender, ImageClickEventArgs e)
{
  MyShoppingCart usersShoppingCart = new MyShoppingCart();
  if (usersShoppingCart.SubmitOrder(User.Identity.Name) == true)
    {
    CheckOutHeader.InnerText = "Thank You - Your Order is Complete.";
    Message.Visible = false;
    CheckoutBtn.Visible = false;
    }
  else
    {
    CheckOutHeader.InnerText = "Order Submission Failed - Please try again. ";
    }
}
```

The "meat" of the order submission process is to be implemented in the SubmitOrder() method of our MyShoppingCart class.

SubmitOrder will:

- Take all the line items in the shopping cart and use them to create a new Order Record and the associated

OrderDetails records.

- Calculate Shipping Date.
- Clear the shopping cart.

```
//-----------------------------------------------------------------------------------+
public bool SubmitOrder(string UserName)
{
  using (CommerceEntities db = new CommerceEntities())
    {
    try
      {
      //-------------------------------------------------------------------+
      //  Add New Order Record                                             |
      //-------------------------------------------------------------------+
      Order newOrder = new Order();
      newOrder.CustomerName = UserName;
      newOrder.OrderDate = DateTime.Now;
      newOrder.ShipDate = CalculateShipDate();
      db.Orders.AddObject(newOrder);
      db.SaveChanges();

      //-------------------------------------------------------------------+
      //  Create a new OderDetail Record for each item in the Shopping Cart     |
      //-------------------------------------------------------------------+
      String cartId = GetShoppingCartId();
      var myCart = (from c in db.ViewCarts where c.CartID == cartId select c);
      foreach (ViewCart item in myCart)
        {
        int i = 0;
        if (i < 1)
          {
          OrderDetail od = new OrderDetail();
          od.OrderID = newOrder.OrderID;
          od.ProductID = item.ProductID;
          od.Quantity = item.Quantity;
          od.UnitCost = item.UnitCost;
          db.OrderDetails.AddObject(od);
          i++;
          }

        var myItem = (from c in db.ShoppingCarts where c.CartID == item.CartID &&
                         c.ProductID == item.ProductID select c).FirstOrDefault();
        if (myItem != null)
          {
          db.DeleteObject(myItem);
          }
        }
      db.SaveChanges();
      }
    catch (Exception exp)
      {
      throw new Exception("ERROR: Unable to Submit Order - " + exp.Message.ToString(),
                                                                 exp);
      }
    }
  return(true);
}
```

For the purposes of this sample application we'll calculate a ship date by simply adding two days to the current date.

```
//-----------------------------------------------------------------------------------+
DateTime CalculateShipDate()
{
   DateTime shipDate = DateTime.Now.AddDays(2);
   return (shipDate);
}
```

Running the application now will permit us to test the shopping process from start to finish.

PREVIOUS NEXT

Part 7: Adding Features

2/20/2020 • 14 minutes to read • Edit Online

by Joe Stagner

Tailspin Spyworks demonstrates how extraordinarily simple it is to create powerful, scalable applications for the .NET platform. It shows off how to use the great new features in ASP.NET 4 to build an online store, including shopping, checkout, and administration.

This tutorial series details all of the steps taken to build the Tailspin Spyworks sample application. Part 7 adds additional features, such as account review, product reviews, and "popular items" and "also purchased" user controls.

Adding Features

Though users can browse our catalog, place items in their shopping cart, and complete the checkout process, there are a number of supporting features that we will include to improve our site.

1. Account Review (List orders placed and view details.)
2. Add some context specific content to the front page.
3. Add a feature to let users Review the products in the catalog.
4. Create a User Control to display Popular Items and Place that control on the front page.
5. Create an "Also Purchased" user control and add it to the product details page.
6. Add a Contact Page.
7. Add an About Page.
8. Global Error

Account Review

In the "Account" folder create two .aspx pages one named OrderList.aspx and the other named OrderDetails.aspx

OrderList.aspx will leverage the GridView and EntityDataSource controls much as we have previously.

```
<div class="ContentHead">Order History</div><br />

<asp:GridView ID="GridView_OrderList" runat="server" AllowPaging="True"
              ForeColor="#333333" GridLines="None" CellPadding="4" Width="100%"
              AutoGenerateColumns="False" DataKeyNames="OrderID"
              DataSourceID="EDS_Orders" AllowSorting="True" ViewStateMode="Disabled" >
  <AlternatingRowStyle BackColor="White" />
  <Columns>
    <asp:BoundField DataField="OrderID" HeaderText="OrderID" ReadOnly="True"
                    SortExpression="OrderID" />
    <asp:BoundField DataField="CustomerName" HeaderText="Customer"
                    SortExpression="CustomerName" />
    <asp:BoundField DataField="OrderDate" HeaderText="Order Date"
                    SortExpression="OrderDate" />
    <asp:BoundField DataField="ShipDate" HeaderText="Ship Date"
                    SortExpression="ShipDate" />
    <asp:HyperLinkField HeaderText="Show Details" Text="Show Details"
                 DataNavigateUrlFields="OrderID"
                 DataNavigateUrlFormatString="~/Account/OrderDetails.aspx?OrderID={0}" />
  </Columns>
  <FooterStyle BackColor="#990000" Font-Bold="True" ForeColor="White" />
  <HeaderStyle BackColor="#990000" Font-Bold="True" ForeColor="White" />
  <PagerStyle BackColor="#FFCC66" ForeColor="#333333" HorizontalAlign="Center" />
  <RowStyle BackColor="#FFFBD6" ForeColor="#333333" />
  <SelectedRowStyle BackColor="#FFCC66" Font-Bold="True" ForeColor="Navy" />
  <SortedAscendingCellStyle BackColor="#FDF5AC" />
  <SortedAscendingHeaderStyle BackColor="#4D0000" />
  <SortedDescendingCellStyle BackColor="#FCF6C0" />
  <SortedDescendingHeaderStyle BackColor="#820000" />
  <SortedAscendingCellStyle BackColor="#FDF5AC"></SortedAscendingCellStyle>
  <SortedAscendingHeaderStyle BackColor="#4D0000"></SortedAscendingHeaderStyle>
  <SortedDescendingCellStyle BackColor="#FCF6C0"></SortedDescendingCellStyle>
  <SortedDescendingHeaderStyle BackColor="#820000"></SortedDescendingHeaderStyle>
</asp:GridView>

<asp:EntityDataSource ID="EDS_Orders" runat="server" EnableFlattening="False"
                      AutoGenerateWhereClause="True"
                      Where=""
                      OrderBy="it.OrderDate DESC"
                      ConnectionString="name=CommerceEntities"
                      DefaultContainerName="CommerceEntities"
                      EntitySetName="Orders" >
   <WhereParameters>
      <asp:SessionParameter Name="CustomerName" SessionField="UserName" />
   </WhereParameters>
</asp:EntityDataSource>
```

The EntityDataSource selects records from the Orders table filtered on the UserName (see the WhereParameter) which we set in a session variable when the user log's in.

Note also these parameters in the HyperlinkField of the GridView:

```
DataNavigateUrlFields="OrderID" DataNavigateUrlFormatString="~/Account/OrderDetails.aspx?OrderID={0}"
```

These specify the link to the Order details view for each product specifying the OrderID field as a QueryString parameter to the OrderDetails.aspx page.

OrderDetails.aspx

We will use an EntityDataSource control to access the Orders and a FormView to display the Order data and another EntityDataSource with a GridView to display all the Order's line items.

```
<asp:FormView ID="FormView1" runat="server" CellPadding="4"
                            DataKeyNames="OrderID"
                            DataSourceID="EDS_Order" ForeColor="#333333" Width="250px">
   <FooterStyle BackColor="#990000" Font-Bold="True" ForeColor="White" />
   <HeaderStyle BackColor="#990000" Font-Bold="True" ForeColor="White" />
   <ItemTemplate>
      OrderID : <%# Eval("OrderID") %><br />
      CustomerName : <%# Eval("CustomerName") %><br />
      Order Date : <%# Eval("OrderDate") %><br />
      Ship Date : <%# Eval("ShipDate") %><br />
   </ItemTemplate>
   <PagerStyle BackColor="#FFCC66" ForeColor="#333333" HorizontalAlign="Center" />
   <RowStyle BackColor="#FFFBD6" ForeColor="#333333" />
</asp:FormView>
<asp:EntityDataSource ID="EDS_Order" runat="server"  EnableFlattening="False"
                      ConnectionString="name=CommerceEntities"
                      DefaultContainerName="CommerceEntities"
                      EntitySetName="Orders"
                      AutoGenerateWhereClause="True"
                      Where=""
                      EntityTypeFilter="" Select="">
   <WhereParameters>
      <asp:QueryStringParameter Name="OrderID" QueryStringField="OrderID" Type="Int32" />
   </WhereParameters>
</asp:EntityDataSource>

<asp:GridView ID="GridView_OrderDetails" runat="server"
              AutoGenerateColumns="False"
              DataKeyNames="ProductID,UnitCost,Quantity"
              DataSourceID="EDS_OrderDetails"
              CellPadding="4" GridLines="Vertical" CssClass="CartListItem"
              onrowdatabound="MyList_RowDataBound" ShowFooter="True"
              ViewStateMode="Disabled">
   <AlternatingRowStyle CssClass="CartListItemAlt" />
   <Columns>
     <asp:BoundField DataField="ProductID" HeaderText="Product ID" ReadOnly="True"
                     SortExpression="ProductID"  />
     <asp:BoundField DataField="ModelNumber" HeaderText="Model Number"
                     SortExpression="ModelNumber" />
     <asp:BoundField DataField="ModelName" HeaderText="Model Name"
                     SortExpression="ModelName" />
     <asp:BoundField DataField="UnitCost" HeaderText="Unit Cost" ReadOnly="True"
                     SortExpression="UnitCost" DataFormatString="{0:c}" />
     <asp:BoundField DataField="Quantity" HeaderText="Quantity" ReadOnly="True"
                     SortExpression="Quantity" />
     <asp:TemplateField>
       <HeaderTemplate>Item Total</HeaderTemplate>
       <ItemTemplate>
         <%# (Convert.ToDouble(Eval("Quantity")) *  Convert.ToDouble(Eval("UnitCost")))%>
       </ItemTemplate>
     </asp:TemplateField>
   </Columns>
   <FooterStyle CssClass="CartListFooter"/>
   <HeaderStyle  CssClass="CartListHead" />
 </asp:GridView>
 <asp:EntityDataSource ID="EDS_OrderDetails" runat="server"
                       ConnectionString="name=CommerceEntities"
                       DefaultContainerName="CommerceEntities"
                       EnableFlattening="False"
                       EntitySetName="VewOrderDetails"
                       AutoGenerateWhereClause="True"
                       Where="">
   <WhereParameters>
     <asp:QueryStringParameter Name="OrderID" QueryStringField="OrderID" Type="Int32" />
   </WhereParameters>
</asp:EntityDataSource>
```

In the Code Behind file (OrderDetails.aspx.cs) we have two little bits of housekeeping.

First we need to make sure that OrderDetails always gets an OrderId.

```
protected void Page_Load(object sender, EventArgs e)
{
  if (String.IsNullOrEmpty(Request.QueryString["OrderId"]))
     {
     Response.Redirect("~/Account/OrderList.aspx");
     }
}
```

We also need to calculate and display the order total from the line items.

```
decimal _CartTotal = 0;

protected void MyList_RowDataBound(object sender, GridViewRowEventArgs e)
{
  if (e.Row.RowType == DataControlRowType.DataRow)
     {
     TailspinSpyworks.Data_Access.VewOrderDetail myCart = new
                                                    Data_Access.VewOrderDetail();
     myCart = (TailspinSpyworks.Data_Access.VewOrderDetail)e.Row.DataItem;
     _CartTotal += Convert.ToDecimal(myCart.UnitCost * myCart.Quantity);
     }
   else if (e.Row.RowType == DataControlRowType.Footer)
     {
     e.Row.Cells[5].Text = "Total: " + _CartTotal.ToString("C");
   }
}
```

The Home Page

Let's add some static content to the Default.aspx page.

First I'll create a "Content" folder and within it an Images folder (and I'll include an image to be used on the home page.)

Into the bottom placeholder of the Default.aspx page, add the following markup.

```
<h2>
  <asp:LoginView ID="LoginView_VisitorGreeting" runat="server">
    <AnonymousTemplate>
       Welcome to the Store !
    </AnonymousTemplate>
    <LoggedInTemplate>
      Hi <asp:LoginName ID="LoginName_Welcome" runat="server" />. Thanks for coming back.
    </LoggedInTemplate>
  </asp:LoginView>
</h2>

<p><strong>TailSpin Spyworks</strong> demonstrates how extraordinarily simple it is to create powerful,
scalable applications for the .NET platform. </p>
<table>
  <tr>
    <td>
      <h3>Some Implementation Features.</h3>
      <ul>
                <li><a href="#">CSS Based Design.</a></li>
                <li><a href="#">Data Access via Linq to Entities.</a></li>
                <li><a href="#">MasterPage driven design.</a></li>
                <li><a href="#">Modern ASP.NET Controls User.</a></li>
                <li><a href="#">Integrated Ajac Control Toolkit Editor.</a></li>
```

```
        </ul>
    </td>
    <td>
        <img src="Content/Images/SampleProductImage.gif" alt=""/>
    </td>
  </tr>
</table>

<table>
  <tr>
    <td colspan="2"><hr /></td>
  </tr>
  <tr>
    <td>
        <!-- Popular Items -->
    </td>
    <td>
      <center><h3>Ecommerce the .NET 4 Way</h3></center>
      <blockquote>
        <p>
        ASP.NET offers web developers the benefit of more that a decade of innovation.
        This   demo leverages many of the latest features of ASP.NET development to
        illustrate really simply building rich web applications with ASP.NET can be.
        For more information about build web applications with ASP.NET please visit the
        community web site at www.asp.net
        </p>
      </blockquote>
    </td>
  </tr>
</table>

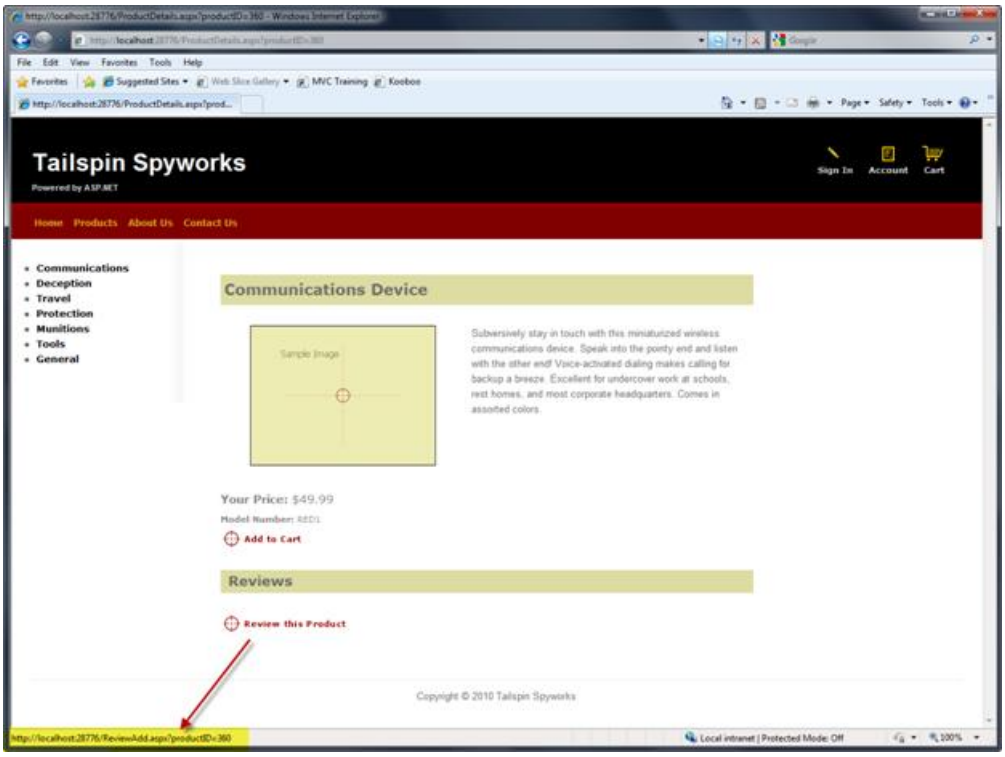
<h3>Spyworks Event Calendar</h3>
<table>
  <tr class="rowH">
    <th>Date</th>
    <th>Title</th>
    <th>Description</th>
  </tr>
  <tr class="rowA">
    <td>June 01, 2011</td>
    <td>Sed vestibulum blandit</td>
    <td>
    Come and check out demos of all the newest Tailspin Spyworks products and experience
    them hands on.
    </td>
  </tr>
  <tr class="rowB">
    <td>November 28, 2011</td>
    <td>Spyworks Product Demo</td>
    <td>
    Come and check out demos of all the newest Tailspin Spyworks products and experience
    them hands on.
    </td>
  </tr>
  <tr class="rowA">
    <td>November 23, 2011</td>
    <td>Spyworks Product Demo</td>
    <td>
     Come and check out demos of all the newest Tailspin Spyworks products and experience
     them hands on.
    </td>
  </tr>
  <tr class="rowB">
    <td>November 21, 2011</td>
    <td>Spyworks Product Demo</td>
    <td>
    Come and check out demos of all the newest Tailspin Spyworks products and experience
    them hands on.
    </td>
```

```
    </td>
  </tr>
</table>
```

Product Reviews

First we'll add a button with a link to a form that we can use to enter a product review.

```
<div class="SubContentHead">Reviews</div><br />
<a id="ReviewList_AddReview" href="ReviewAdd.aspx?productID=<%# Eval("ProductID") %>">
   <img id="Img2" runat="server"
        src="~/Styles/Images/review_this_product.gif" alt="" />
</a>
```

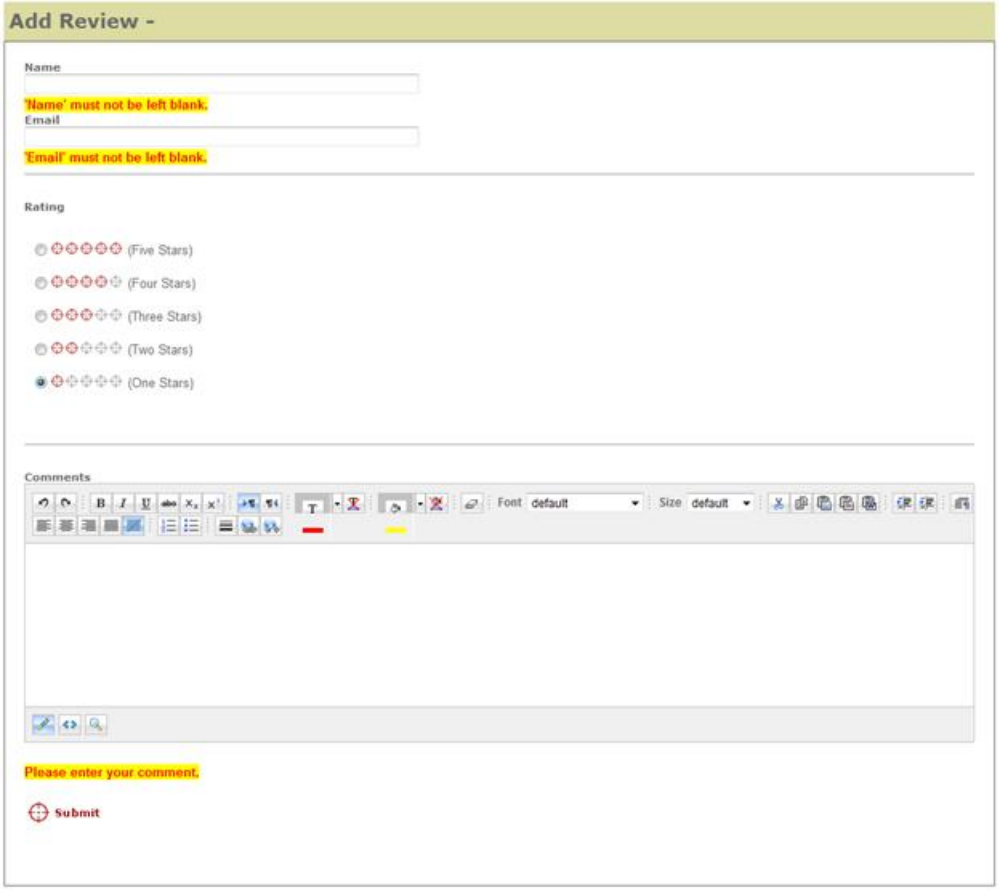


Note that we are passing the ProductID in the query string

Next let's add page named ReviewAdd.aspx

This page will use the ASP.NET AJAX Control Toolkit. If you have not already done so you can download it from DevExpress and there is guidance on setting up the toolkit for use with Visual Studio here https://www.asp.net/learn/ajax-videos/video-76.aspx.

In design mode, drag controls and validators from the toolbox and build a form like the one below.



The markup will look something like this.

```
<asp:ToolkitScriptManager ID="ToolkitScriptManager1" runat="server">
</asp:ToolkitScriptManager>
<div class="ContentHead">Add Review - <asp:label id="ModelName" runat="server" /></div>
<div>
  <span class="NormalBold">Name</span><br />
  <asp:TextBox id="Name" runat="server" Width="400px" /><br />
  <asp:RequiredFieldValidator runat="server" id="RequiredFieldValidator1"
                              ControlToValidate="Name"
                              Display="Dynamic"
                              CssClass="ValidationError"
                              ErrorMessage="'Name' must not be left blank."  /><br />
  <span class="NormalBold">Email</span><br />
  <asp:TextBox id="Email" runat="server" Width="400px" /><br />
  <asp:RequiredFieldValidator runat="server" id="RequiredFieldValidator2"
                              ControlToValidate="Email" Display="Dynamic"
                              CssClass="ValidationError"
                              ErrorMessage="'Email' must not be left blank." />
  <br /><hr /><br />
  <span class="NormalBold">Rating</span><br /><br />
  <asp:RadioButtonList ID="Rating" runat="server">
    <asp:ListItem value="5" selected="True"
             Text='<img src="Styles/Images/reviewrating5.gif" alt=""> (Five Stars) '  />
    <asp:ListItem value="4" selected="True"
             Text='<img src="Styles/Images/reviewrating4.gif" alt=""> (Four Stars) '  />
    <asp:ListItem value="3" selected="True"
             Text='<img src="Styles/Images/reviewrating3.gif" alt=""> (Three Stars) '  />
    <asp:ListItem value="2" selected="True"
             Text='<img src="Styles/Images/reviewrating2.gif" alt=""> (Two Stars) '  />
    <asp:ListItem value="1" selected="True"
             Text='<img src="Styles/Images/reviewrating1.gif" alt=""> (One Stars) '  />
  </asp:RadioButtonList>
  <br /><hr /><br />
  <span class="NormalBold">Comments</span><br />
  <cc1:Editor ID="UserComment" runat="server" />
  <asp:RequiredFieldValidator runat="server" id="RequiredFieldValidator3"
                              ControlToValidate="UserComment" Display="Dynamic"
                              CssClass="ValidationError"
                              ErrorMessage="Please enter your comment." /><br /><br />
  <asp:ImageButton ImageURL="Styles/Images/submit.gif" runat="server"
                   id="ReviewAddBtn" onclick="ReviewAddBtn_Click" />
  <br /><br /><br />
</div>
```

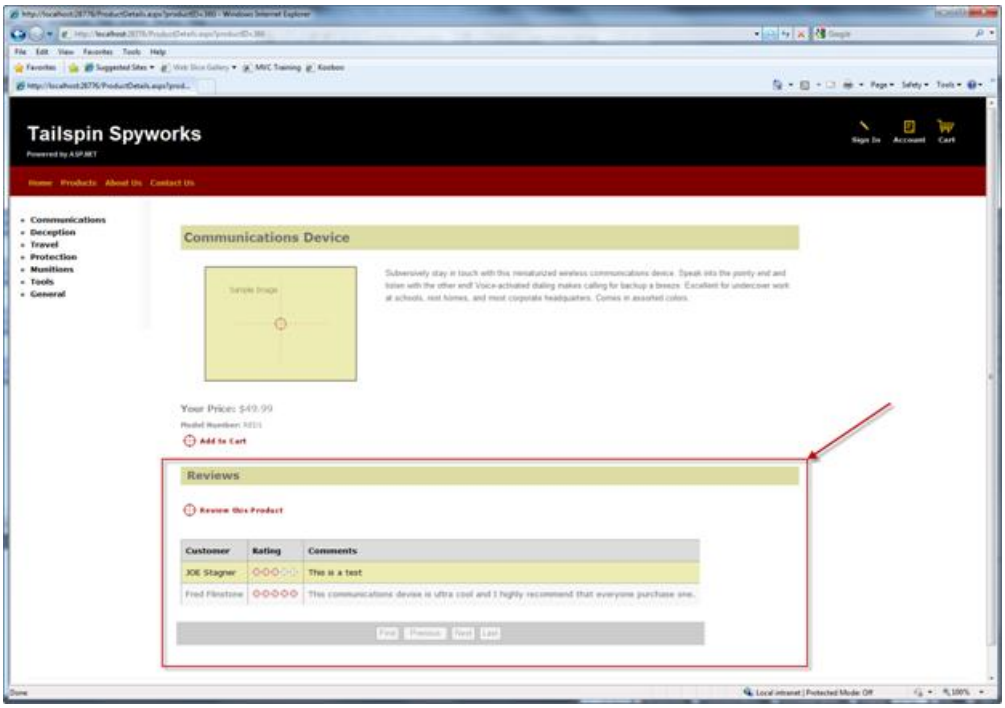
Now that we can enter reviews, lets display those reviews on the product page.

Add this markup to the ProductDetails.aspx page.

```
<asp:ListView ID="ListView_Comments" runat="server"
              DataKeyNames="ReviewID,ProductID,Rating" DataSourceID="EDS_CommentsList">
  <ItemTemplate>
    <tr>
      <td><%# Eval("CustomerName") %></td>
      <td>
        <img src='Styles/Images/ReviewRating_d<%# Eval("Rating") %>.gif' alt="">
        <br />
      </td>
      <td>
        <%# Eval("Comments") %>
      </td>
    </tr>
  </ItemTemplate>
  <AlternatingItemTemplate>
    <tr>
      <td><%# Eval("CustomerName") %></td>
      <td>
        <img src='Styles/Images/ReviewRating_da<%# Eval("Rating") %>.gif' alt="">
```

```
        <br />
      </td>
      <td><%# Eval("Comments") %></td>
    </tr>
  </AlternatingItemTemplate>
   <EmptyDataTemplate>
     <table runat="server">
       <tr><td>There are no reviews yet for this product.</td></tr>
     </table>
  </EmptyDataTemplate>
  <LayoutTemplate>
    <table runat="server">
      <tr runat="server">
        <td runat="server">
          <table ID="itemPlaceholderContainer" runat="server" border="1">
            <tr runat="server">
              <th runat="server">Customer</th>
              <th runat="server">Rating</th>
              <th runat="server">Comments</th>
             </tr>
             <tr ID="itemPlaceholder" runat="server"></tr>
           </table>
         </td>
       </tr>
       <tr runat="server">
         <td runat="server">
           <asp:DataPager ID="DataPager1" runat="server">
             <Fields>
               <asp:NextPreviousPagerField ButtonType="Button"
                                           ShowFirstPageButton="True"
                                           ShowLastPageButton="True" />
             </Fields>
           </asp:DataPager>
         </td>
       </tr>
     </table>
  </LayoutTemplate>
</asp:ListView>
<asp:EntityDataSource ID="EDS_CommentsList" runat="server"  EnableFlattening="False"
                      AutoGenerateWhereClause="True"
                      EntityTypeFilter=""
                      Select="" Where=""
                      ConnectionString="name=CommerceEntities"
                      DefaultContainerName="CommerceEntities"
                      EntitySetName="Reviews">
   <WhereParameters>
    <asp:QueryStringParameter Name="ProductID" QueryStringField="productID"
                                               Type="Int32" />
  </WhereParameters>
</asp:EntityDataSource>
```

Running our application now and navigating to a product shows the product information including customer reviews.



Popular Items Control (Creating User Controls)

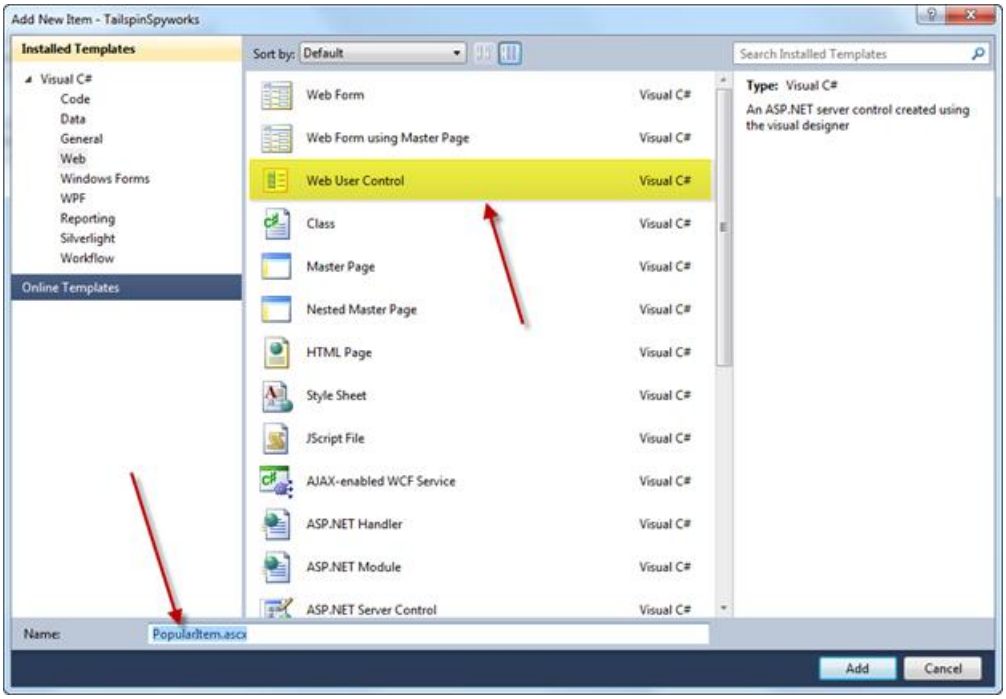
In order to increase sales on your web site we will add a couple of features to "suggestive sell" popular or related products.

The first of these features will be a list of the more popular product in our product catalog.

We will create a "User Control" to display the top selling items on the home page of our application. Since this will be a control, we can use it on any page by simply dragging and dropping the control in Visual Studio's designer onto any page that we like.

In Visual Studio's solutions explorer, right-click on the solution name and create a new directory named "Controls". While it is not necessary to do so, we will help keep our project organized by creating all our user controls in the "Controls" directory.

Right-click on the controls folder and choose "New Item" :



Specify a name for our control of "PopularItems". Note that the file extension for user controls is .ascx not .aspx.

Our Popular Items User control will be defined as follows.

```
<%@ OutputCache Duration="3600" VaryByParam="None" %>
<div class="MostPopularHead">Our most popular items this week</div>
<div id="PanelPopularItems" runat="server">
  <asp:Repeater ID="RepeaterItemsList" runat="server">
    <HeaderTemplate></HeaderTemplate>
      <ItemTemplate>
        <a class='MostPopularItemText'
           href='ProductDetails.aspx?productID=<%# Eval("ProductId") %>'>
                                                <%# Eval("ModelName") %></a><br />
      </ItemTemplate>
    <FooterTemplate></FooterTemplate>
  </asp:Repeater>
</div>
```

Here we're using a method we have not used yet in this application. We're using the repeater control and instead of using a data source control we're binding the Repeater Control to the results of a LINQ to Entities query.

In the code behind of our control we do that as follows.

```
using TailspinSpyworks.Data_Access;

protected void Page_Load(object sender, EventArgs e)
{
  using (CommerceEntities db = new CommerceEntities())
    {
    try
      {
      var query = (from ProductOrders in db.OrderDetails
                        join SelectedProducts in db.Products on ProductOrders.ProductID
                        equals SelectedProducts.ProductID
                        group ProductOrders by new
                            {
                            ProductId = SelectedProducts.ProductID,
                            ModelName = SelectedProducts.ModelName
                            } into grp
                        select new
                            {
                            ModelName = grp.Key.ModelName,
                            ProductId = grp.Key.ProductId,
                            Quantity = grp.Sum(o => o.Quantity)
                            } into orderdgrp where orderdgrp.Quantity > 0
                        orderby orderdgrp.Quantity descending select orderdgrp).Take(5);

                    RepeaterItemsList.DataSource = query;
                    RepeaterItemsList.DataBind();
      }
    catch (Exception exp)
      {
      throw new Exception("ERROR: Unable to Load Popular Items - " +
                          exp.Message.ToString(), exp);
      }
    }
}
```

Note also this important line at the top of our control's markup.

```
<%@ OutputCache Duration="3600" VaryByParam="None" %>
```

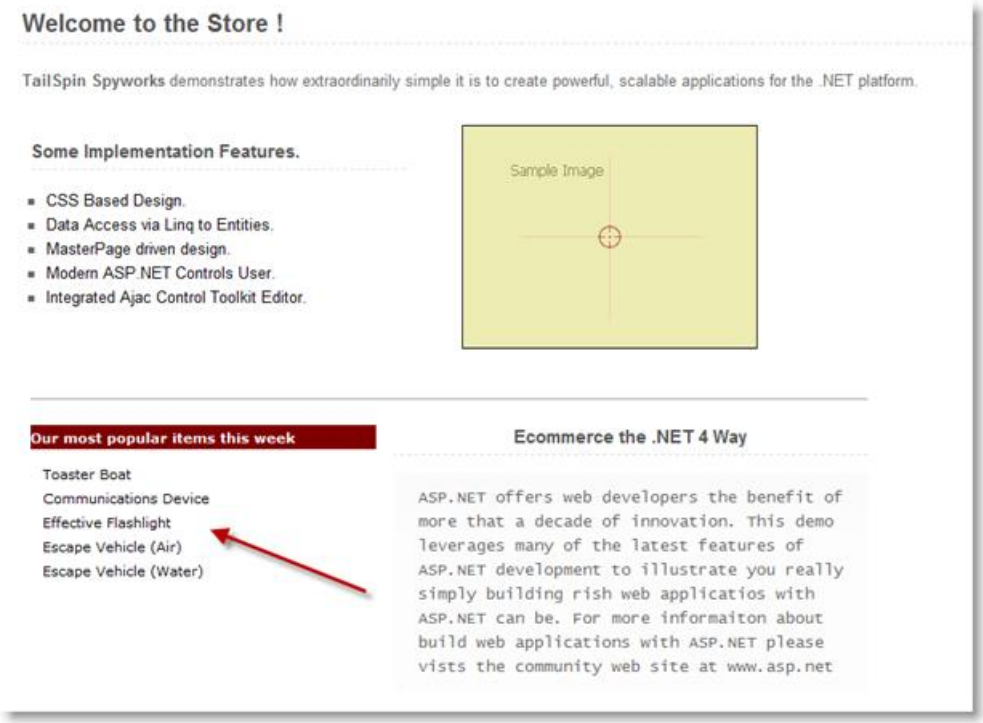
Since the most popular items won't be changing on a minute to minute basis we can add a aching directive to improve the performance of our application. This directive will cause the controls code to only be executed when

the cached output of the control expires. Otherwise, the cached version of the control's output will be used.

Now all we have to do is include our new control in our Default.aspx page.

Use drag and drop to place an instance of the control in the open column of our Default form.

Now when we run our application the home page displays the most popular items.



"Also Purchased" Control (User Controls with Parameters)

The second User Control that we'll create will take suggestive selling to the next level by adding context specificity.

The logic to calculate the top "Also Purchased" items is non-trivial.

Our "Also Purchased" control will select the OrderDetails records (previously purchased) for the currently selected ProductID and grab the OrderIDs for each unique order that is found.

Then we will select al the products from all those Orders and sum the quantities purchased. We'll sort the products by that quantity sum and display the top five items.

Given the complexity of this logic, we will implement this algorithm as a stored procedure.

The T-SQL for the stored procedure is as follows.

```
ALTER PROCEDURE dbo.SelectPurchasedWithProducts
 @ProductID int
AS
        SELECT  TOP 5
    OrderDetails.ProductID,
    Products.ModelName,
    SUM(OrderDetails.Quantity) as TotalNum

FROM
    OrderDetails
  INNER JOIN Products ON OrderDetails.ProductID = Products.ProductID

WHERE   OrderID IN
(
    /* This inner query should retrieve all orders that have contained the productID */
    SELECT DISTINCT OrderID
    FROM OrderDetails
    WHERE ProductID = @ProductID
)
AND OrderDetails.ProductID != @ProductID

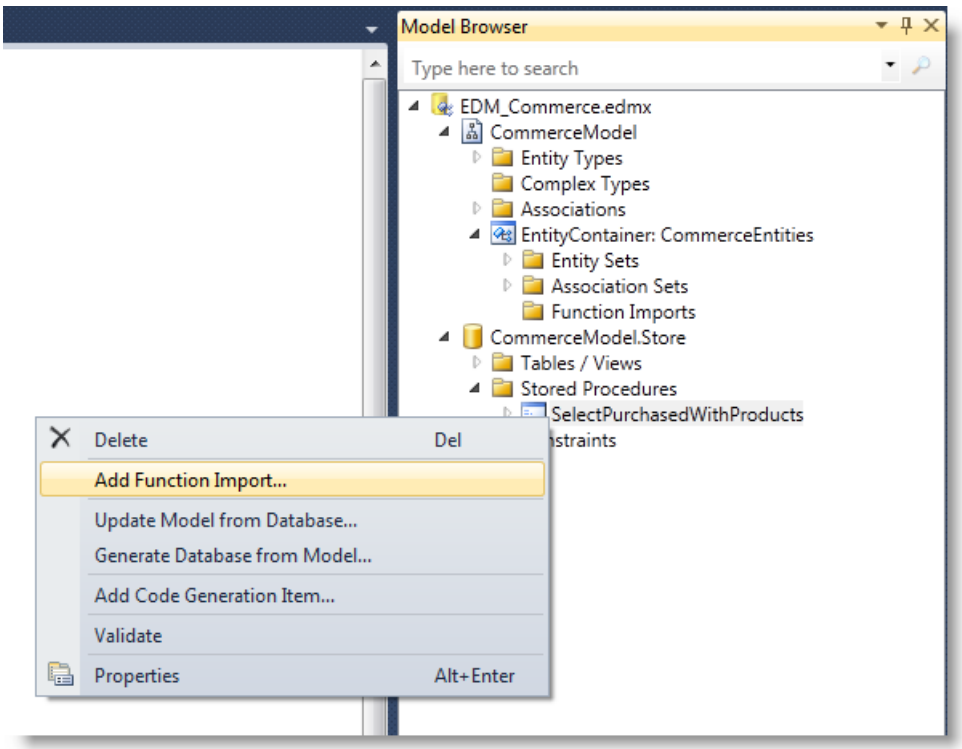
GROUP BY OrderDetails.ProductID, Products.ModelName

ORDER BY TotalNum DESC
RETURN
```

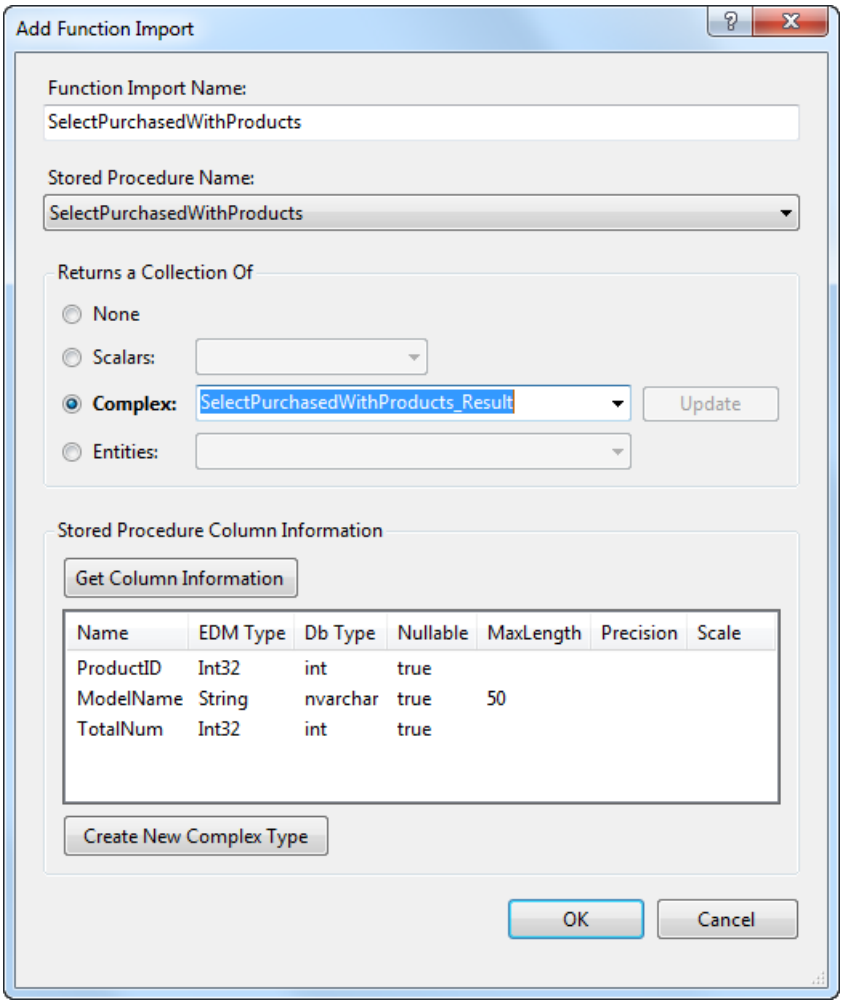
Note that this stored procedure (SelectPurchasedWithProducts) existed in the database when we included it in our application and when we generated the Entity Data Model we specified that, in addition to the Tables and Views that we needed, the Entity Data Model should include this stored procedure.

To access the stored procedure from the Entity Data Model we need to import the function.

Double Click on the Entity Data Model in the Solutions Explorer to open it in the designer and open the Model Browser, then right-click in the designer and select "Add Function Import".



Doing so will open this dialog.



Fill out the fields as you see above, selecting the "SelectPurchasedWithProducts" and use the procedure name for the name of our imported function.

Click "Ok".

Having done this we can simply program against the stored procedure as we might any other item in the model.

So, in our "Controls" folder create a new user control named AlsoPurchased.ascx.

The markup for this control will look very familiar to the PopularItems control.

```
<div>
<div class="MostPopularHead">
<asp:Label ID="LabelTitle" runat="server" Text=" Customers who bought this also bought:"></asp:Label></div>
<div id="PanelAlsoBoughtItems" runat="server">
    <asp:Repeater ID="RepeaterItemsList" runat="server">
       <HeaderTemplate></HeaderTemplate>
          <ItemTemplate>
             <a class='MostPopularItemText' href='ProductDetails.aspx?productID=<%# Eval("ProductId") %>'><%#
Eval("ModelName") %></a><br />
          </ItemTemplate>
       <FooterTemplate></FooterTemplate>
    </asp:Repeater>
</div>
</div>
```

The notable difference is that are not caching the output since the item's to be rendered will differ by product.

The ProductId will be a "property" to the control.

```
private int _ProductId;

public int ProductId
{
get { return _ProductId ; }
set { _ProductId = Convert.ToInt32(value); }
}
```

In the control's PreRender event handler we eed to do three things.

1. Make sure the ProductID is set.
2. See if there are any products that have been purchased with the current one.
3. Output some items as determined in #2.

Note how easy it is to call the stored procedure through the model.

```
//-------------------------------------------------------------------------------------+
protected void Page_PreRender(object sender, EventArgs e)
{
  if (_ProductId < 1)
     {
     // This should never happen but we could expand the use of this control by reducing
     // the dependency on the query string by selecting a few RANDOME products here.
     Debug.Fail("ERROR : The Also Purchased Control Can not be used without
                         setting the ProductId.");
     throw new Exception("ERROR : It is illegal to load the AlsoPurchased COntrol
                                  without setting a ProductId.");
     }

  int ProductCount = 0;
  using (CommerceEntities db = new CommerceEntities())
    {
    try
      {
      var v = db.SelectPurchasedWithProducts(_ProductId);
      ProductCount = v.Count();
      }
    catch (Exception exp)
      {
      throw new Exception("ERROR: Unable to Retrieve Also Purchased Items - " +
                                  exp.Message.ToString(), exp);
      }
    }

  if (ProductCount > 0)
     {
     WriteAlsoPurchased(_ProductId);
     }
  else
     {
     WritePopularItems();
     }
}
```

After determining that there ARE "also purchased" we can simply bind the repeater to the results returned by the query.

```
//-----------------------------------------------------------------------------------------+
private void WriteAlsoPurchased(int currentProduct)
{
  using (CommerceEntities db = new CommerceEntities())
        {
        try
          {
          var v = db.SelectPurchasedWithProducts(currentProduct);
          RepeaterItemsList.DataSource = v;
          RepeaterItemsList.DataBind();
          }
         catch (Exception exp)
          {
          throw new Exception("ERROR: Unable to Write Also Purchased - " +
                                                          exp.Message.ToString(), exp);
          }
        }
}
```

If there were not any "also purchased" items we'll simply display other popular items from our catalog.

```
//-----------------------------------------------------------------------------------------+
private void WritePopularItems()
{
  using (CommerceEntities db = new CommerceEntities())
    {
    try
      {
      var query = (from ProductOrders in db.OrderDetails
                   join SelectedProducts in db.Products on ProductOrders.ProductID
                   equals SelectedProducts.ProductID
                   group ProductOrders by new
                         {
                         ProductId = SelectedProducts.ProductID,
                         ModelName = SelectedProducts.ModelName
                         } into grp
                   select new
                         {
                         ModelName = grp.Key.ModelName,
                         ProductId = grp.Key.ProductId,
                         Quantity = grp.Sum(o => o.Quantity)
                         } into orderdgrp
                   where orderdgrp.Quantity > 0
                   orderby orderdgrp.Quantity descending
                   select orderdgrp).Take(5);

      LabelTitle.Text = "Other items you might be interested in: ";
      RepeaterItemsList.DataSource = query;
      RepeaterItemsList.DataBind();
      }
    catch (Exception exp)
      {
      throw new Exception("ERROR: Unable to Load Popular Items - " +
                                                        exp.Message.ToString(), exp);
      }
    }
}
```

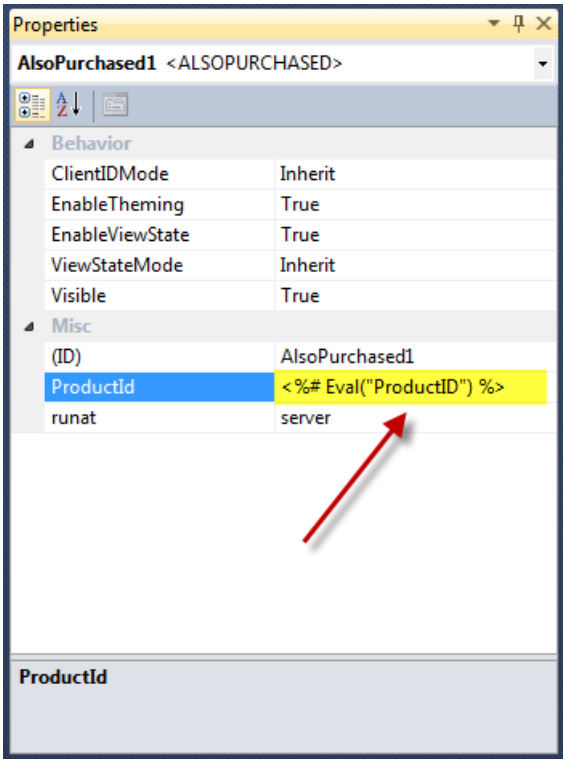
To view the "Also Purchased" items, open the ProductDetails.aspx page and drag the AlsoPurchased control from the Solutions Explorer so that it appears in this position in the markup.

```
<table  border="0">
  <tr>
     <td>
       <img src='Catalog/Images/<%# Eval("ProductImage") %>'  border="0"
                                                   alt='<%# Eval("ModelName") %>' />
     </td>
     <td><%# Eval("Description") %><br /><br /><br />
         <uc1:AlsoPurchased ID="AlsoPurchased1" runat="server" />
     </td>
   </tr>
</table>
```

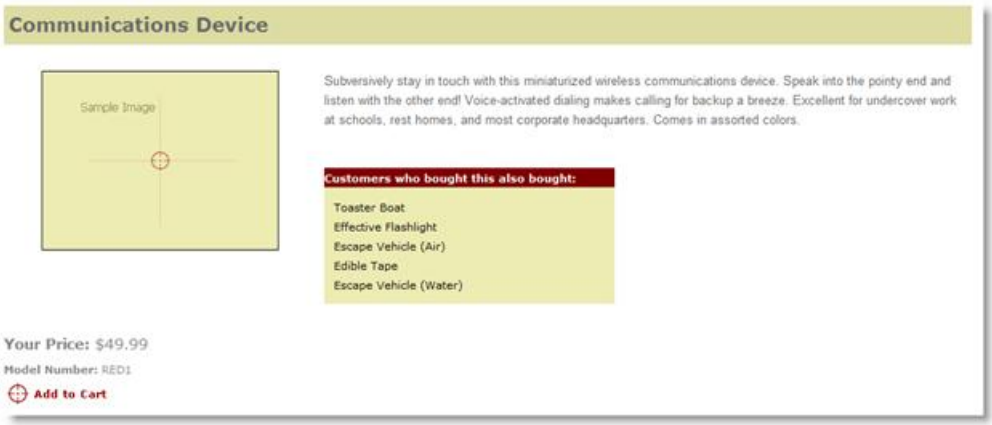
Doing so will create a reference to the control at the top of the ProductDetails page.

```
<%@ Register src="Controls/AlsoPurchased.ascx" tagname="AlsoPurchased" tagprefix="uc1" %>
```

Since the AlsoPurchased user control requires a ProductId number we will set the ProductID property of our control by using an Eval statement against the current data model item of the page.



When we build and run now and browse to a product we see the "Also Purchased" items.



Part 8: Final Pages, Exception Handling, and Conclusion

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

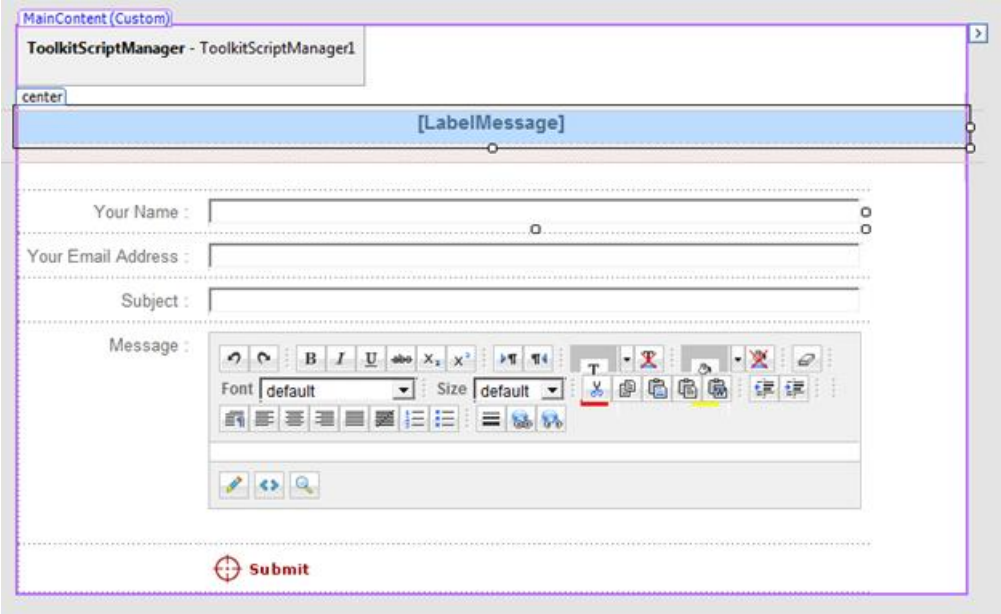
Tailspin Spyworks demonstrates how extraordinarily simple it is to create powerful, scalable applications for the .NET platform. It shows off how to use the great new features in ASP.NET 4 to build an online store, including shopping, checkout, and administration.

This tutorial series details all of the steps taken to build the Tailspin Spyworks sample application. Part 8 adds a contact page, about page, and exception handling. This is the conclusion of the series.

Contact Page (Sending email from ASP.NET)

Create a new page named ContactUs.aspx

Using the designer, create the following form taking special note to include the ToolkitScriptManager and the Editor control from the AjaxControlToolkit. .



Double click on the "Submit" button to generate a click event handler in the code behind file and implement a method to send the contact information as an email.

```
protected void ImageButton_Submit_Click(object sender, ImageClickEventArgs e)
  {
  try
    {
    MailMessage mMailMessage = new MailMessage();
    mMailMessage.From = new MailAddress(HttpUtility.HtmlEncode(TextBoxEmail.Text));
    mMailMessage.To.Add(new MailAddress("Your Email Here"));

    // mMailMessage.Bcc.Add(new MailAddress(bcc));
    // mMailMessage.CC.Add(new MailAddress(cc));

   mMailMessage.Subject = "From:" + HttpUtility.HtmlEncode(TextBoxYourName.Text) + "-" +
                                    HttpUtility.HtmlEncode(TextBoxSubject.Text);
   mMailMessage.Body = HttpUtility.HtmlEncode(EditorEmailMessageBody.Content);
   mMailMessage.IsBodyHtml = true;
   mMailMessage.Priority = MailPriority.Normal;
   SmtpClient mSmtpClient = new SmtpClient();
   mSmtpClient.Send(mMailMessage);
   LabelMessage.Text = "Thank You - Your Message was sent.";
   }
 catch (Exception exp)
   {
   throw new Exception("ERROR: Unable to Send Contact - " + exp.Message.ToString(), exp);
   }
}
```

This code requires that your web.config file contain an entry in the configuration section that specifies the SMTP server to use for sending mail.

```
<system.net>
        <mailSettings>
            <smtp>
                <network
                     host="mail..com"
                     port="25"
                     userName=""
                     password="" />
            </smtp>
        </mailSettings>
    </system.net>
```

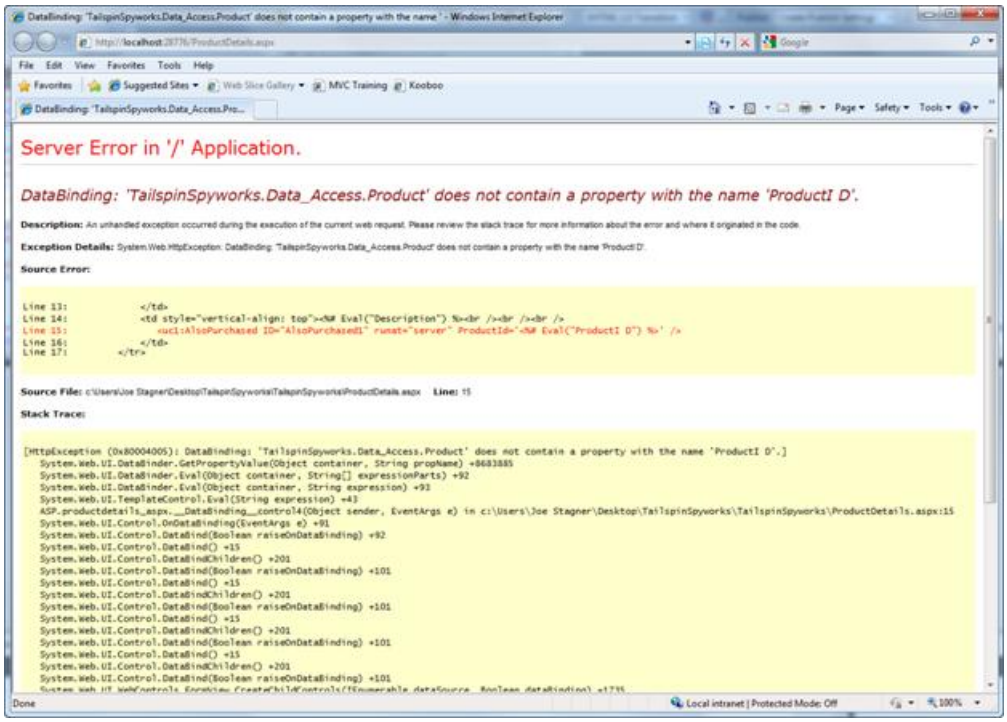
About Page

Create a page named AboutUs.aspx and add whatever content you like.

Global Exception Handler

Lastly, throughout the application we have thrown exceptions and there are unforeseen circumstances that cold also cause unhandled exceptions in our web application.

We never want an unhandled exception to be displayed to a web site visitor.



Apart from being a terrible user experience unhandled exceptions can also be a security problem.

To solve this problem we will implement a global exception handler.

To do this, open the Global.asax file and note the following pre-generated event handler.

```
void Application_Error(object sender, EventArgs e)
 {
 // Code that runs when an unhandled error occurs
 }
```

Add code to implement the Application_Error handler as follows.

```
void Application_Error(object sender, EventArgs e)
 {
    Exception myEx =  Server.GetLastError();
    String RedirectUrlString = "~/Error.aspx?InnerErr=" +
        myEx.InnerException.Message.ToString() + "&Err=" + myEx.Message.ToString();
    Response.Redirect(RedirectUrlString);
 }
```

Then add a page named Error.aspx to the solution and add this markup snippet.

```
<center>
  <div class="ContentHead">ERROR</div><br /><br />
  <asp:Label ID="Label_ErrorFrom" runat="server" Text="Label"></asp:Label><br /><br />
  <asp:Label ID="Label_ErrorMessage" runat="server" Text="Label"></asp:Label><br /><br />
</center>
```

Now in the Page_Load event handler extract the error messages from the Request Object.

```
protected void Page_Load(object sender, EventArgs e)
{
    Label_ErrorFrom.Text = Request["Err"].ToString();
    Label_ErrorMessage.Text = Request["InnerErr"].ToString();
}
```

Conclusion

We've seen that ASP.NET WebForms makes it easy to create a sophisticated website with database access, membership, AJAX, etc. pretty quickly.

Hopefully this tutorial has given you the tools you need to get started building your own ASP.NET WebForms applications!

PREVIOUS

ASP.NET 4 - Getting Started with Entity Framework 4 and Web Forms

2/20/2020 • 2 minutes to read • Edit Online

The basics of using Entity Framework to display and edit data in an ASP.NET Web Forms application.

- Getting Started with Entity Framework 4.0 Database First
- Getting Started with Entity Framework 4.0 Database First - Part 2
- Getting Started with Entity Framework 4.0 Database First - Part 3
- Getting Started with Entity Framework 4.0 Database First - Part 4
- Getting Started with Entity Framework 4.0 Database First - Part 5
- Getting Started with Entity Framework 4.0 Database First - Part 6
- Getting Started with Entity Framework 4.0 Database First - Part 7
- Getting Started with Entity Framework 4.0 Database First - Part 8

Getting Started with Entity Framework 4.0 Database First and ASP.NET 4 Web Forms

2/20/2020 • 9 minutes to read • Edit Online

by Tom Dykstra

The Contoso University sample web application demonstrates how to create ASP.NET Web Forms applications using the Entity Framework 4.0 and Visual Studio 2010. The sample application is a website for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments.

The tutorial shows examples in C#. The downloadable sample contains code in both C# and Visual Basic.

Database First

There are three ways you can work with data in the Entity Framework: *Database First*, *Model First*, and *Code First*. This tutorial is for Database First. For information about the differences between these workflows and guidance on how to choose the best one for your scenario, see Entity Framework Development Workflows.

Web Forms

This tutorial series uses the ASP.NET Web Forms model and assumes you know how to work with ASP.NET Web Forms in Visual Studio. If you don't, see Getting Started with ASP.NET 4.5 Web Forms. If you prefer to work with the ASP.NET MVC framework, see Getting Started with the Entity Framework using ASP.NET MVC.

Software versions

SHOWN IN THE TUTORIAL	ALSO WORKS WITH
Windows 7	Windows 8
Visual Studio 2010	Visual Studio 2010 Express for Web. The tutorial has not been tested with later versions of Visual Studio. There are many differences in menu selections, dialog boxes, and templates.
.NET 4	.NET 4.5 is backward compatible with .NET 4, but the tutorial has not been tested with .NET 4.5.
Entity Framework 4	The tutorial has not been tested with later versions of Entity Framework. Starting with Entity Framework 5, EF uses by default the `DbContext API` that was introduced with EF 4.1. The EntityDataSource control was designed to use the `ObjectContext` API. For information about how to use the EntityDataSource control with the `DbContext` API, see this blog post.

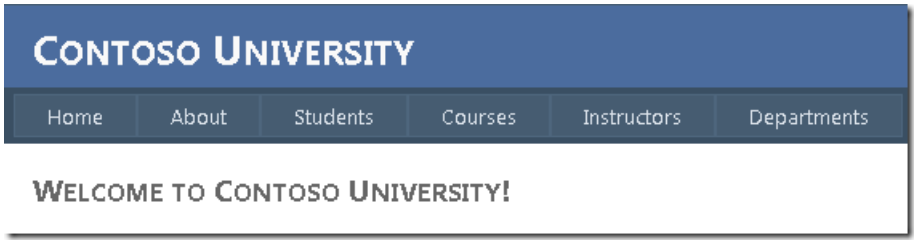
Questions

If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET Entity

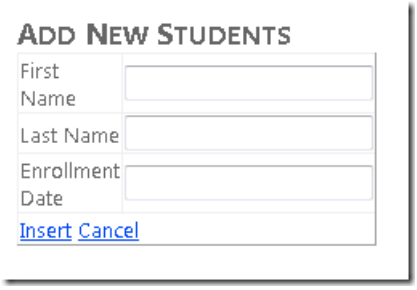
Framework forum, the Entity Framework and LINQ to Entities forum, or StackOverflow.com.

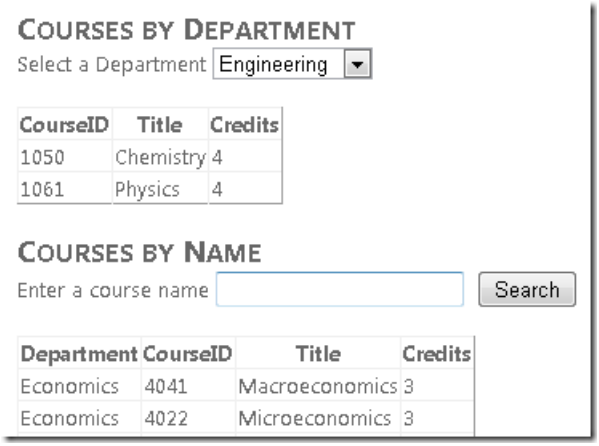
Overview

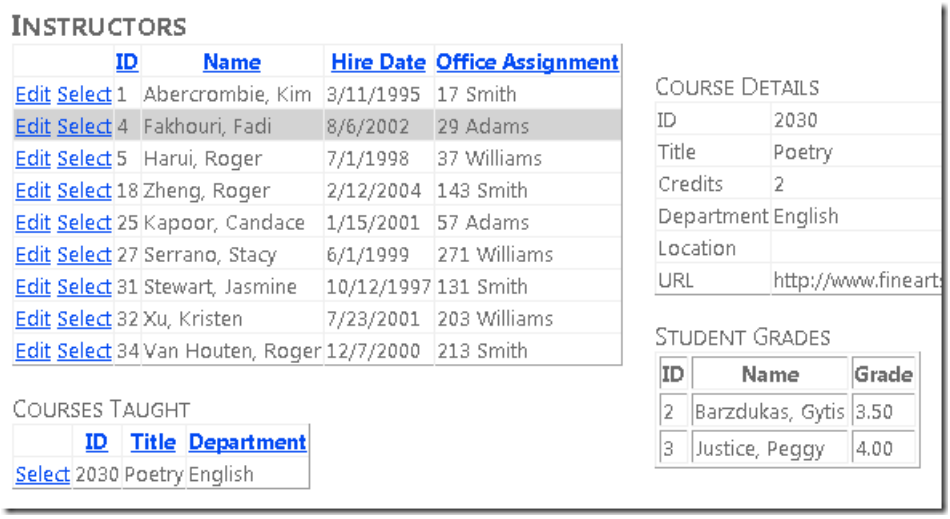
The application you'll be building in these tutorials is a simple university website.



Users can view and update student, course, and instructor information. A few of the screens you'll create are shown below.

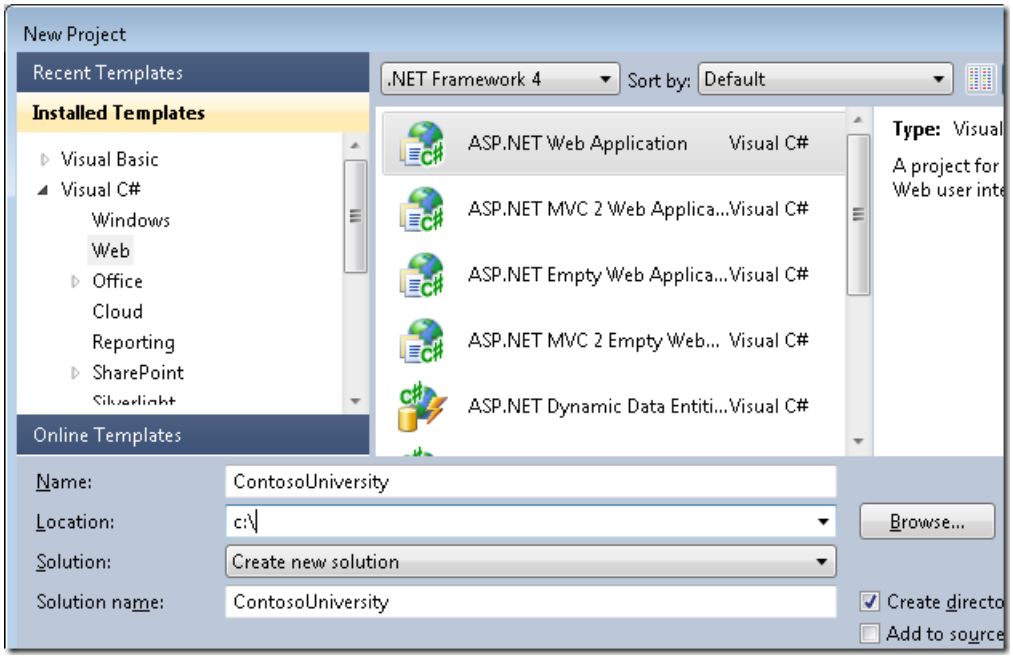




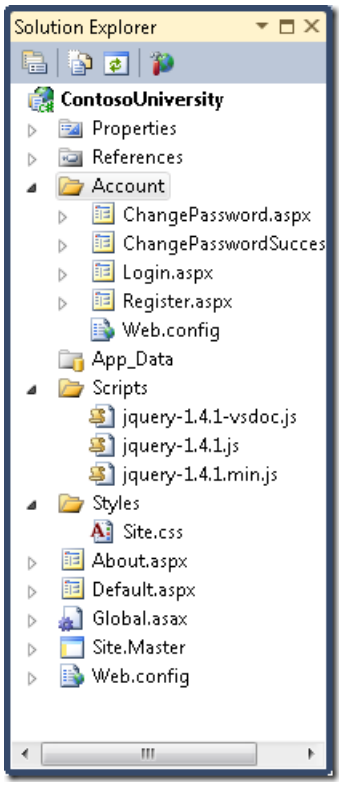


Creating the Web Application

To start the tutorial, open Visual Studio and then create a new ASP.NET Web Application Project using the **ASP.NET Web Application** template:



This template creates a web application project that already includes a style sheet and master pages:



Open the *Site.Master* file and change "My ASP.NET Application" to "Contoso University".

```
<h1>
    Contoso University
</h1>
```

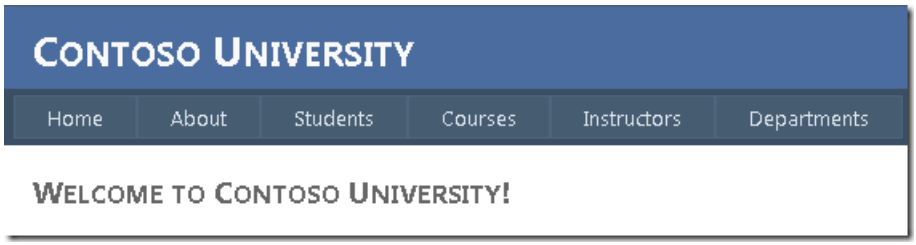
Find the *Menu* control named `NavigationMenu` and replace it with the following markup, which adds menu items for the pages you'll be creating.

```
<asp:Menu ID="NavigationMenu" runat="server" CssClass="menu" EnableViewState="false"
                    IncludeStyleBlock="false" Orientation="Horizontal">
                    <Items>
                        <asp:MenuItem NavigateUrl="~/Default.aspx" Text="Home" />
                        <asp:MenuItem NavigateUrl="~/About.aspx" Text="About" />
                        <asp:MenuItem NavigateUrl="~/Students.aspx" Text="Students">
                            <asp:MenuItem NavigateUrl="~/StudentsAdd.aspx" Text="Add Students" />
                        </asp:MenuItem>
                        <asp:MenuItem NavigateUrl="~/Courses.aspx" Text="Courses">
                            <asp:MenuItem NavigateUrl="~/CoursesAdd.aspx" Text="Add Courses" />
                        </asp:MenuItem>
                        <asp:MenuItem NavigateUrl="~/Instructors.aspx" Text="Instructors">
                            <asp:MenuItem NavigateUrl="~/InstructorsCourses.aspx" Text="Course Assignments"
/>
                            <asp:MenuItem NavigateUrl="~/OfficeAssignments.aspx" Text="Office Assignments"
/>
                        </asp:MenuItem>
                        <asp:MenuItem NavigateUrl="~/Departments.aspx" Text="Departments">
                            <asp:MenuItem NavigateUrl="~/DepartmentsAdd.aspx" Text="Add Departments" />
                        </asp:MenuItem>
                    </Items>
                </asp:Menu>
```

Open the *Default.aspx* page and change the `Content` control named `BodyContent` to this:

```
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <h2>
        Welcome to Contoso University!
   </h2>
</asp:Content>
```

You now have a simple home page with links to the various pages that you'll be creating:



Creating the Database

For these tutorials, you'll use the Entity Framework data model designer to automatically create the data model based on an existing database (often called the *database-first* approach). An alternative that's not covered in this tutorial series is to create the data model manually and then have the designer generate scripts that create the database (the *model-first* approach).

For the database-first method used in this tutorial, the next step is to add a database to the site. The easiest way is to first download the project that goes with this tutorial. Then right-click the *App_Data* folder, select **Add Existing Item**, and select the *School.mdf* database file from the downloaded project.

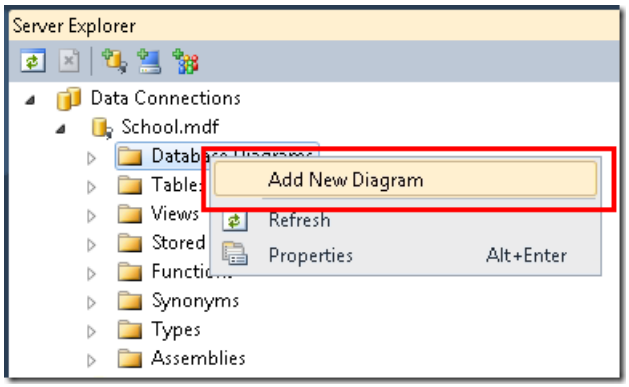
An alternative is to follow the instructions at Creating the School Sample Database. Whether you download the database or create it, copy the *School.mdf* file from the following folder to your application's *App_Data* folder:

```
%PROGRAMFILES%\Microsoft SQL Server\MSSQL10.SQLEXPRESS\MSSQL\DATA
```

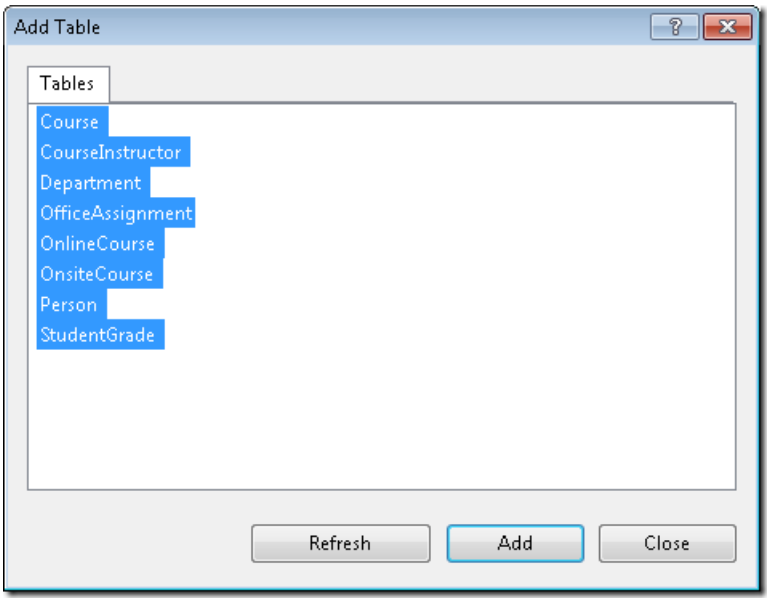
(This location of the *.mdf* file assumes you're using SQL Server 2008 Express.)

If you create the database from a script, perform the following steps to create a database diagram:

1. In **Server Explorer**, expand **Data Connections**, expand *School.mdf*, right-click **Database Diagrams**, and select **Add New Diagram**.



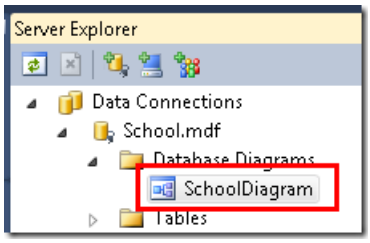
2. Select all of the tables and then click **Add**.



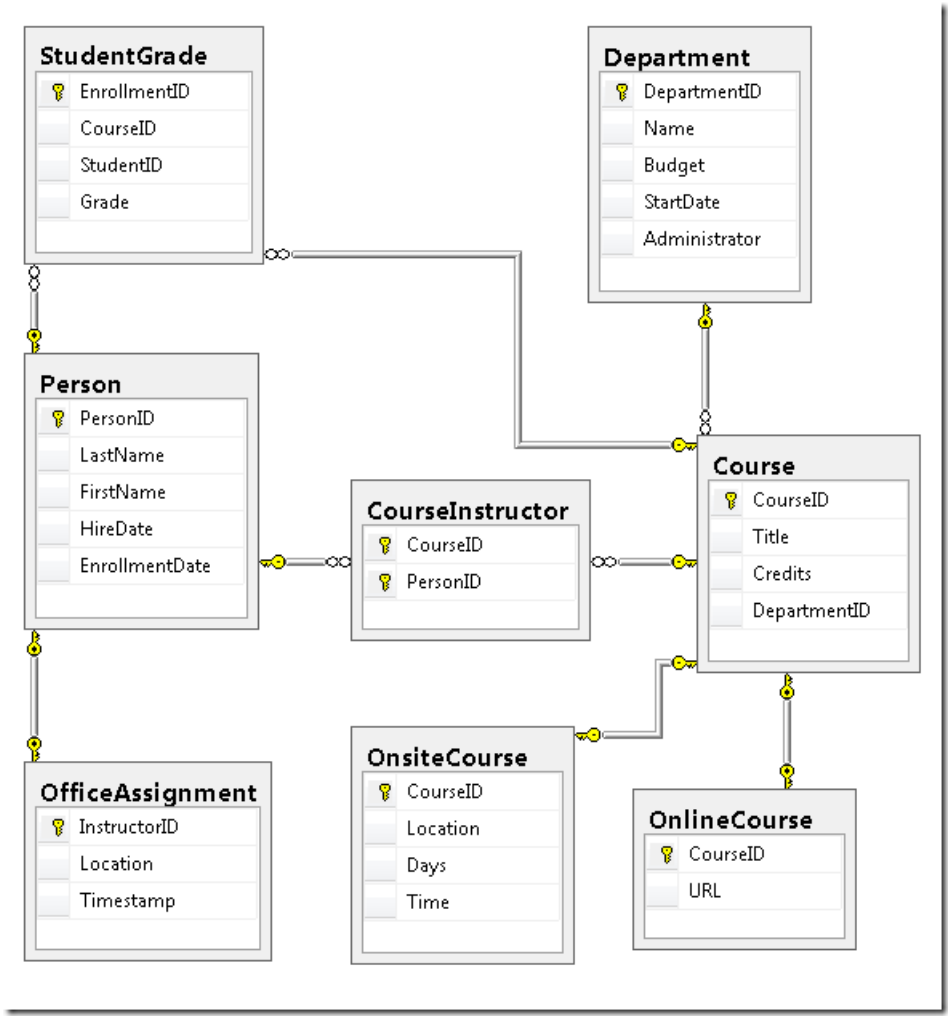
SQL Server creates a database diagram that shows tables, columns in the tables, and relationships between the tables. You can move the tables around to organize them however you like.

3. Save the diagram as "SchoolDiagram" and close it.

If you download the *School.mdf* file that goes with this tutorial, you can view the database diagram by double-clicking **SchoolDiagram** under **Database Diagrams** in **Server Explorer**.



The diagram looks something like this (the tables might be in different locations from what's shown here):

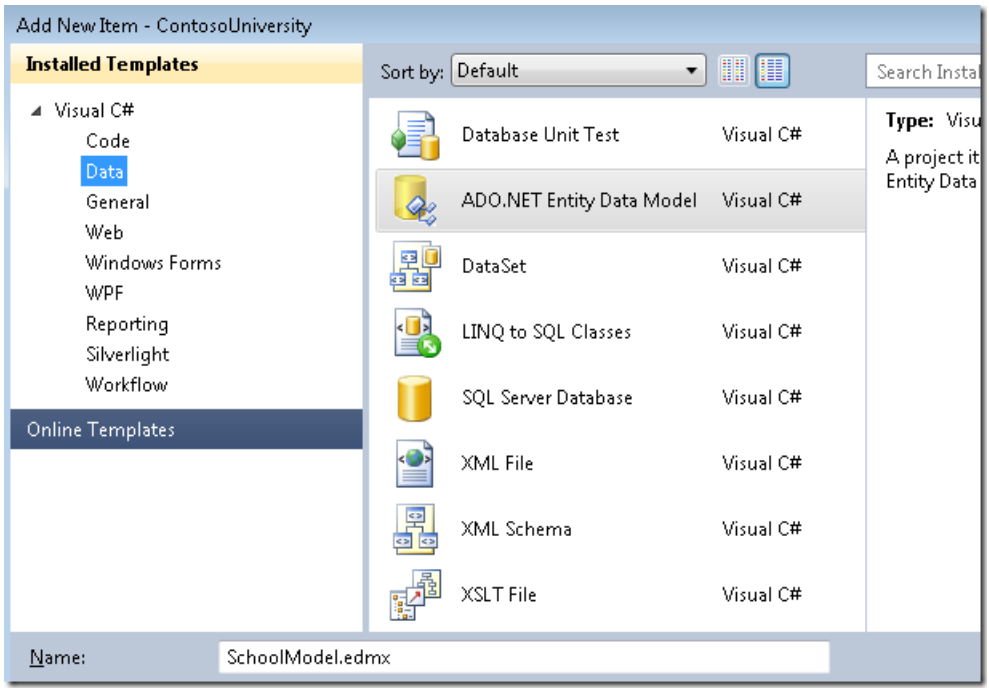


Creating the Entity Framework Data Model

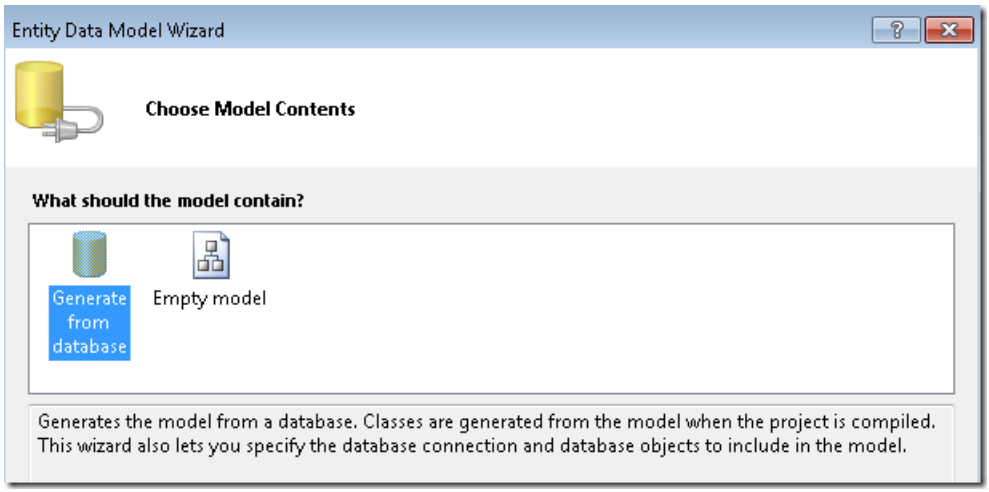
Now you can create an Entity Framework data model from this database. You could create the data model in the root folder of the application, but for this tutorial you'll place it in a folder named *DAL* (for Data Access Layer).

In **Solution Explorer**, add a project folder named *DAL* (make sure it's under the project, not under the solution).

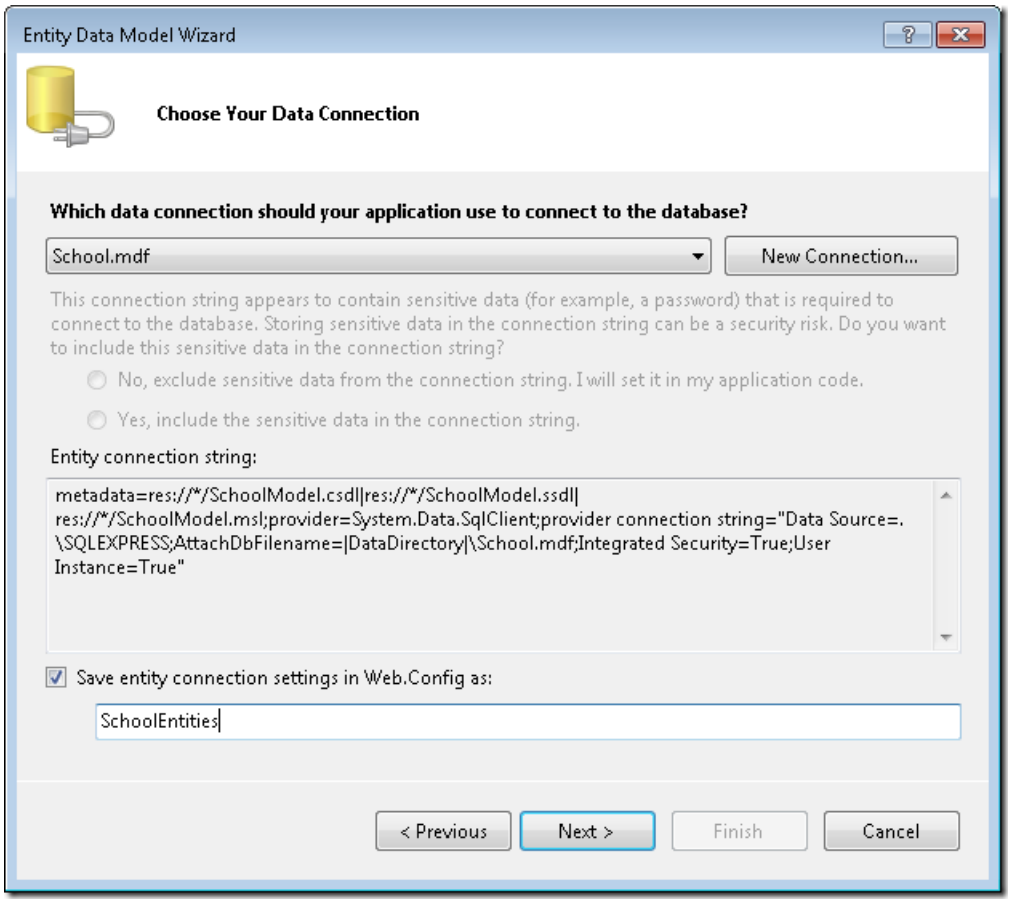
Right-click the *DAL* folder and then select **Add** and **New Item**. Under **Installed Templates**, select **Data**, select the **ADO.NET Entity Data Model** template, name it *SchoolModel.edmx*, and then click **Add**.



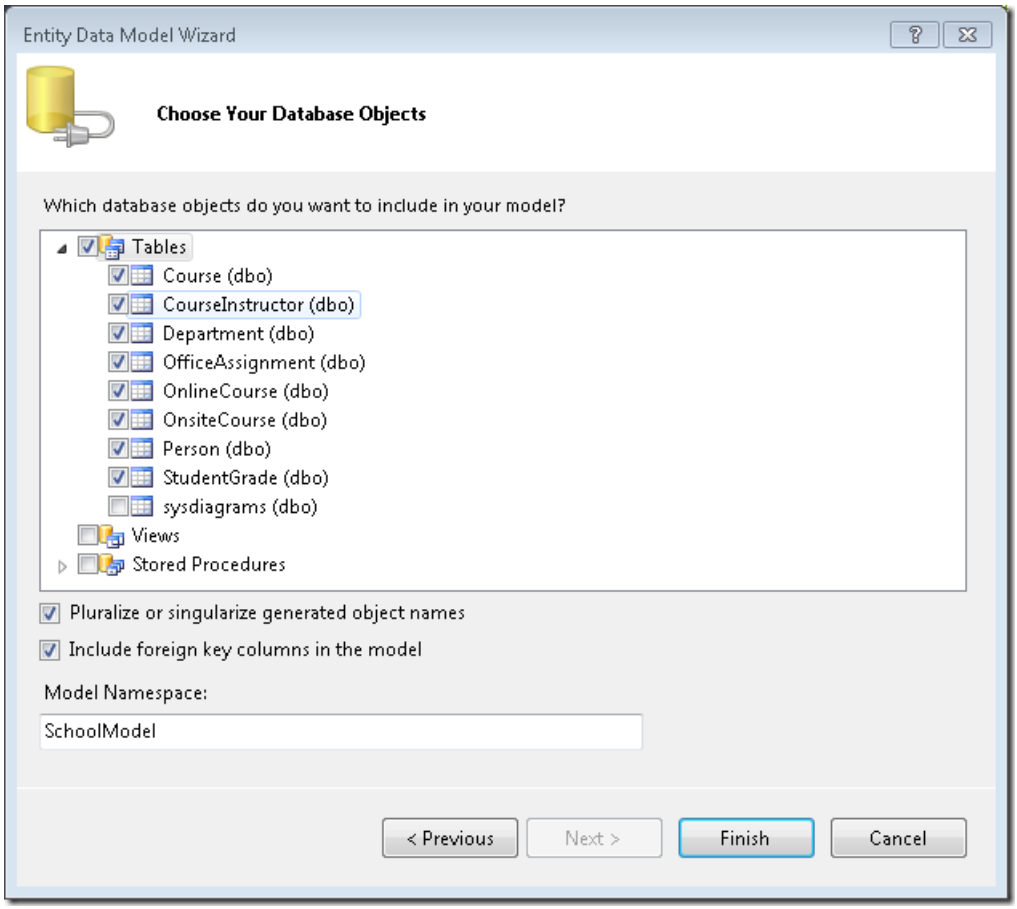
This starts the Entity Data Model Wizard. In the first wizard step, the **Generate from database** option is selected by default. Click **Next**.



In the **Choose Your Data Connection** step, leave the default values and click **Next**. The School database is selected by default and the connection setting is saved in the *Web.config* file as **SchoolEntities**.

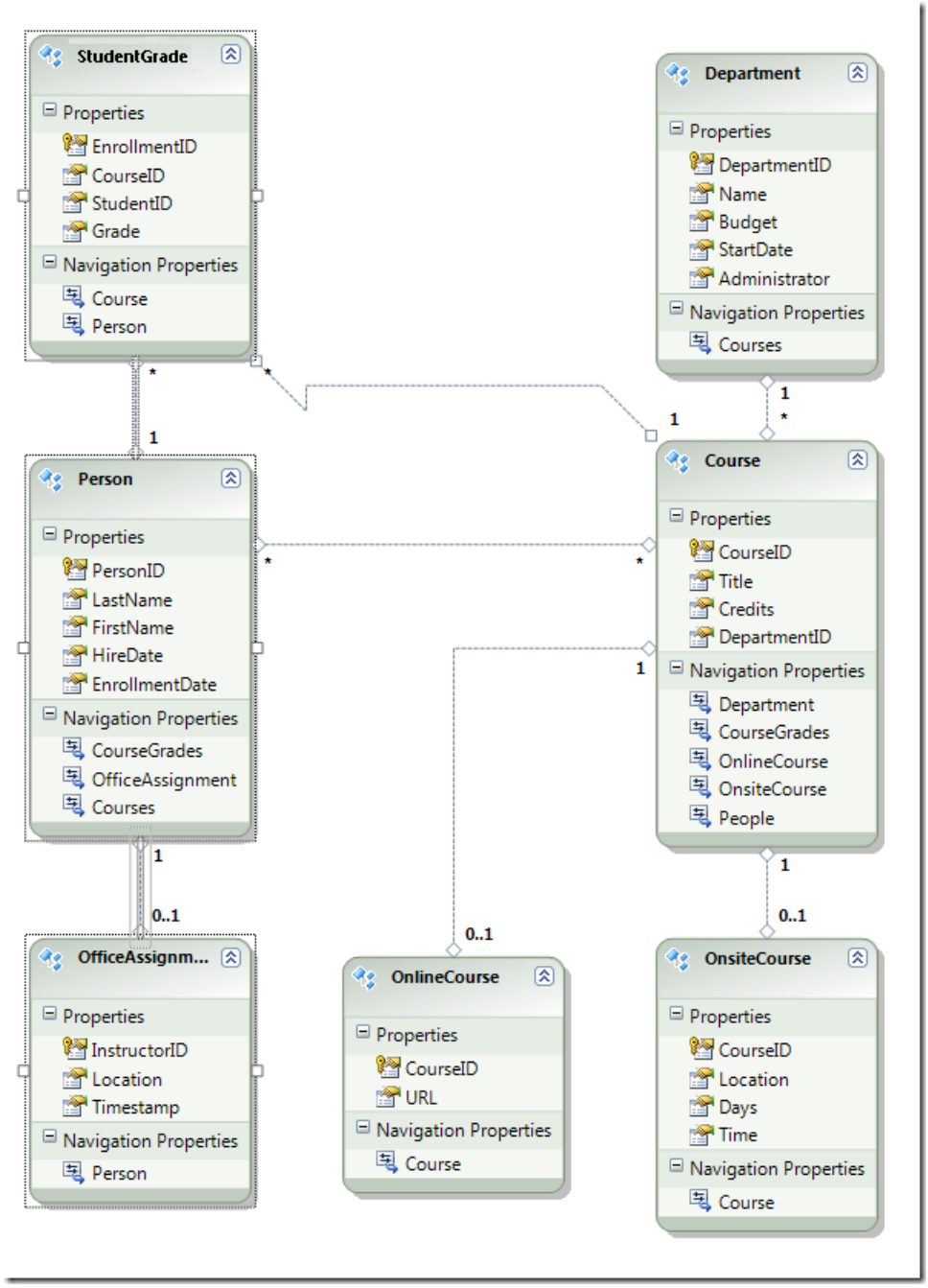


In the **Choose Your Database Objects** wizard step, select all of the tables except `sysdiagrams` (which was created for the diagram you generated earlier) and then click **Finish**.



After it's finished creating the model, Visual Studio shows you a graphical representation of the Entity Framework objects (entities) that correspond to your database tables. (As with the database diagram, the location of individual elements might be different from what you see in this illustration. You can drag the

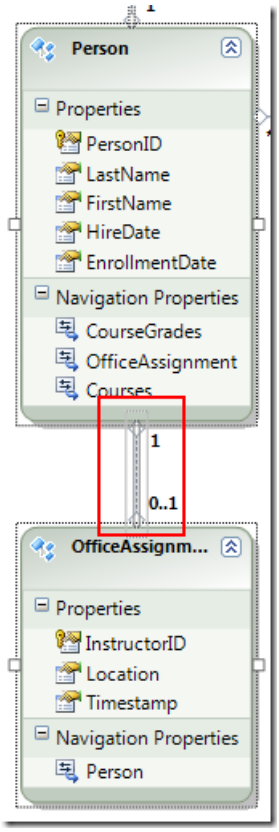
elements around to match the illustration if you want.)



Exploring the Entity Framework Data Model

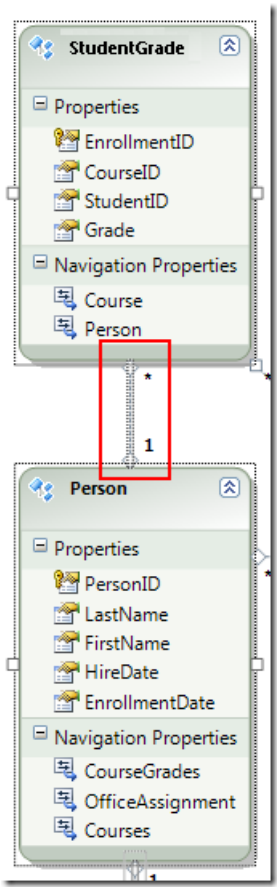
You can see that the entity diagram looks very similar to the database diagram, with a couple of differences. One difference is the addition of symbols at the end of each association that indicate the type of association (table relationships are called entity associations in the data model):

- A one-to-zero-or-one association is represented by "1" and "0..1".



In this case, a `Person` entity may or may not be associated with an `OfficeAssignment` entity. An `OfficeAssignment` entity must be associated with a `Person` entity. In other words, an instructor may or may not be assigned to an office, and any office can be assigned to only one instructor.

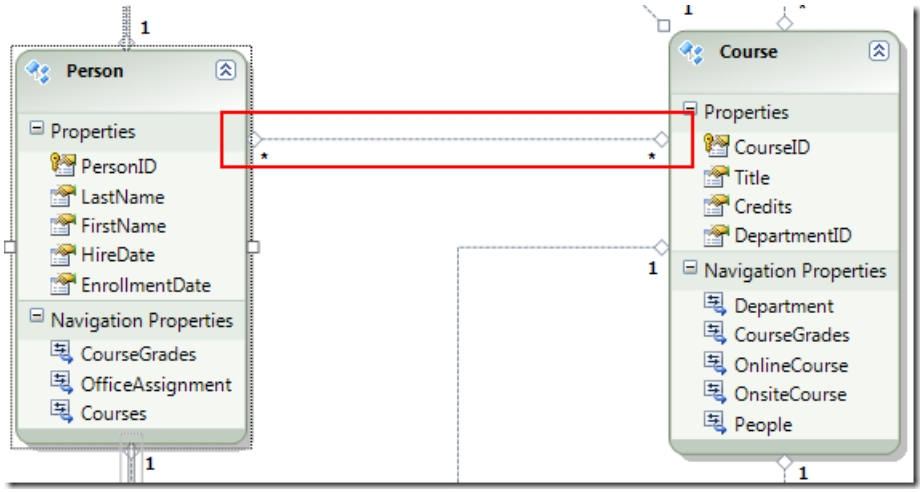
- A one-to-many association is represented by "1" and "*".



In this case, a `Person` entity may or may not have associated `StudentGrade` entities. A `StudentGrade` entity must be associated with one `Person` entity. `StudentGrade` entities actually represent enrolled

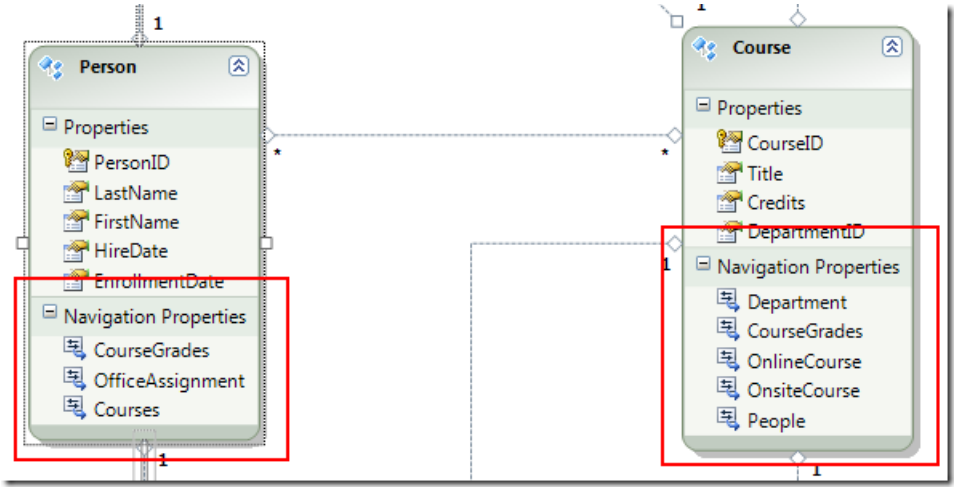
courses in this database; if a student is enrolled in a course and there's no grade yet, the `Grade` property is null. In other words, a student may not be enrolled in any courses yet, may be enrolled in one course, or may be enrolled in multiple courses. Each grade in an enrolled course applies to only one student.

- A many-to-many association is represented by "*" and "*".

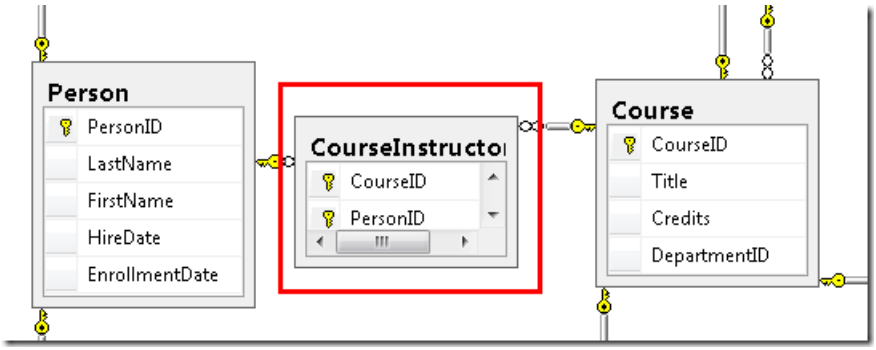


In this case, a `Person` entity may or may not have associated `Course` entities, and the reverse is also true: a `Course` entity may or may not have associated `Person` entities. In other words, an instructor may teach multiple courses, and a course may be taught by multiple instructors. (In this database, this relationship applies only to instructors; it does not link students to courses. Students are linked to courses by the StudentGrades table.)

Another difference between the database diagram and the data model is the additional **Navigation Properties** section for each entity. A navigation property of an entity references related entities. For example, the `Courses` property in a `Person` entity contains a collection of all the `Course` entities that are related to that `Person` entity.

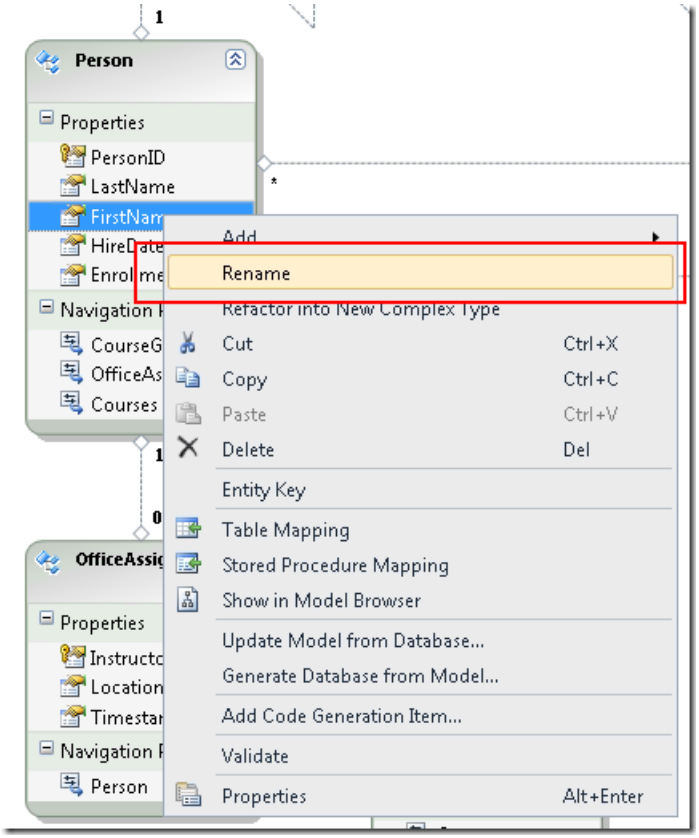


Yet another difference between the database and data model is the absence of the `CourseInstructor` association table that's used in the database to link the `Person` and `Course` tables in a many-to-many relationship. The navigation properties enable you to get related `Course` entities from the `Person` entity and related `Person` entities from the `Course` entity, so there's no need to represent the association table in the data model.

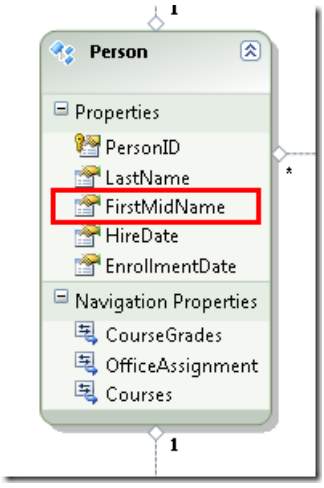


For purposes of this tutorial, suppose the `FirstName` column of the `Person` table actually contains both a person's first name and middle name. You want to change the name of the field to reflect this, but the database administrator (DBA) might not want to change the database. You can change the name of the `FirstName` property in the data model, while leaving its database equivalent unchanged.

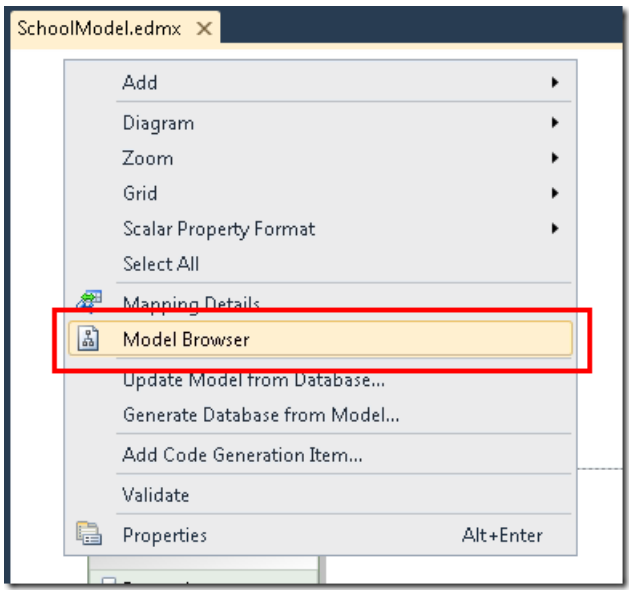
In the designer, right-click **FirstName** in the `Person` entity, and then select **Rename**.



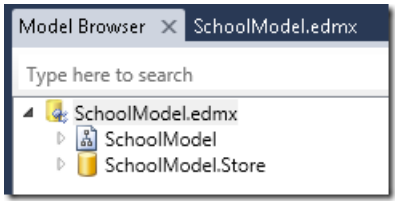
Type in the new name "FirstMidName". This changes the way you refer to the column in code without changing the database.



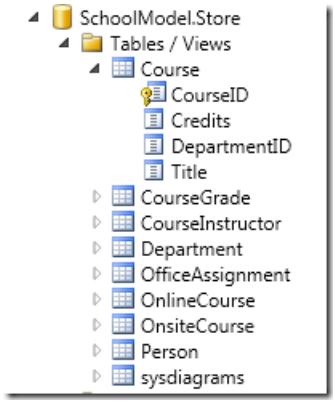
The model browser provides another way to view the database structure, the data model structure, and the mapping between them. To see it, right-click a blank area in the entity designer and then click **Model Browser**.



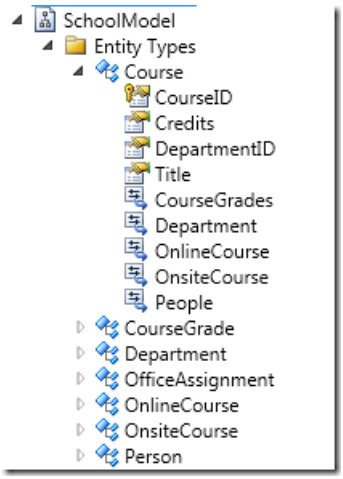
The **Model Browser** pane displays a tree view. (The **Model Browser** pane might be docked with the **Solution Explorer** pane.) The **SchoolModel** node represents the data model structure, and the **SchoolModel.Store** node represents the database structure.



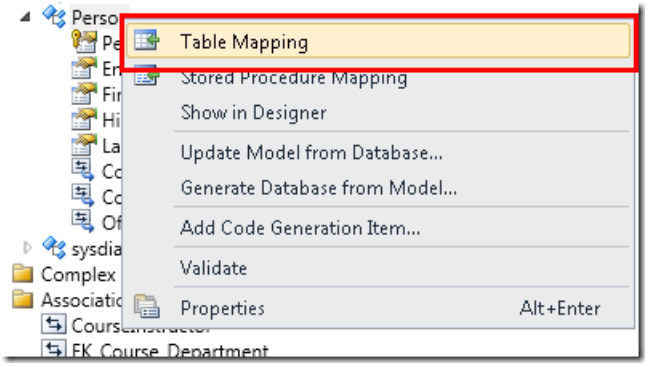
Expand **SchoolModel.Store** to see the tables, expand **Tables / Views** to see tables, and then expand **Course** to see the columns within a table.



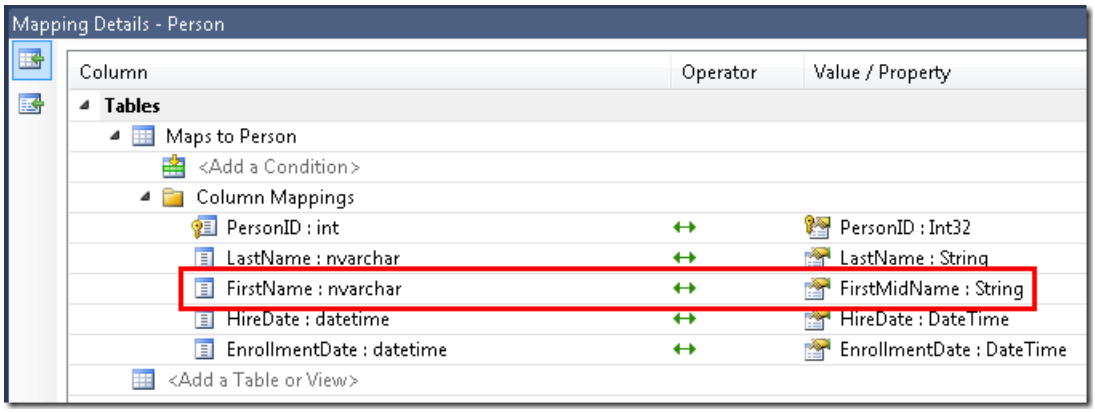
Expand **SchoolModel**, expand **Entity Types**, and then expand the **Course** node to see the entities and the properties within the entities.



In either the designer or the **Model Browser** pane you can see how the Entity Framework relates the objects of the two models. Right-click the `Person` entity and select **Table Mapping**.



This opens the **Mapping Details** window. Notice that this window lets you see that the database column `FirstName` is mapped to `FirstMidName`, which is what you renamed it to in the data model.



The Entity Framework uses XML to store information about the database, the data model, and the mappings between them. The *SchoolModel.edmx* file is actually an XML file that contains this information. The designer renders the information in a graphical format, but you can also view the file as XML by right-clicking the *.edmx* file in **Solution Explorer**, clicking **Open With**, and selecting **XML (Text) Editor**. (The data model designer and an XML editor are just two different ways of opening and working with the same file, so you cannot have the designer open and open the file in an XML editor at the same time.)

You've now created a website, a database, and a data model. In the next walkthrough you'll begin working with data using the data model and the ASP.NET `EntityDataSource` control.

NEXT

Getting Started with Entity Framework 4.0 Database First and ASP.NET 4 Web Forms - Part 2

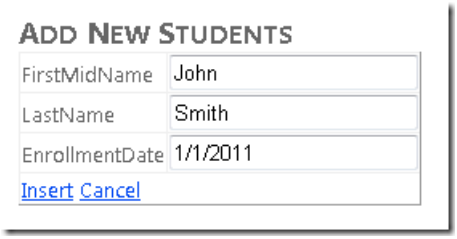
2/20/2020 • 14 minutes to read • Edit Online

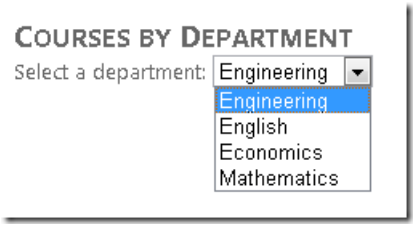
by Tom Dykstra

> The Contoso University sample web application demonstrates how to create ASP.NET Web Forms applications using the Entity Framework 4.0 and Visual Studio 2010. For information about the tutorial series, see the first tutorial in the series

The EntityDataSource Control

In the previous tutorial you created a web site, a database, and a data model. In this tutorial you work with the `EntityDataSource` control that ASP.NET provides in order to make it easy to work with an Entity Framework data model. You'll create a `GridView` control for displaying and editing student data, a `DetailsView` control for adding new students, and a `DropDownList` control for selecting a department (which you'll use later for displaying associated courses).





Note that in this application you won't be adding input validation to pages that update the database, and some of the error handling will not be as robust as would be required in a production application. That keeps the tutorial focused on the Entity Framework and keeps it from getting too long. For details about how to add these features to your application, see Validating User Input in ASP.NET Web Pages and Error Handling in ASP.NET Pages and Applications.

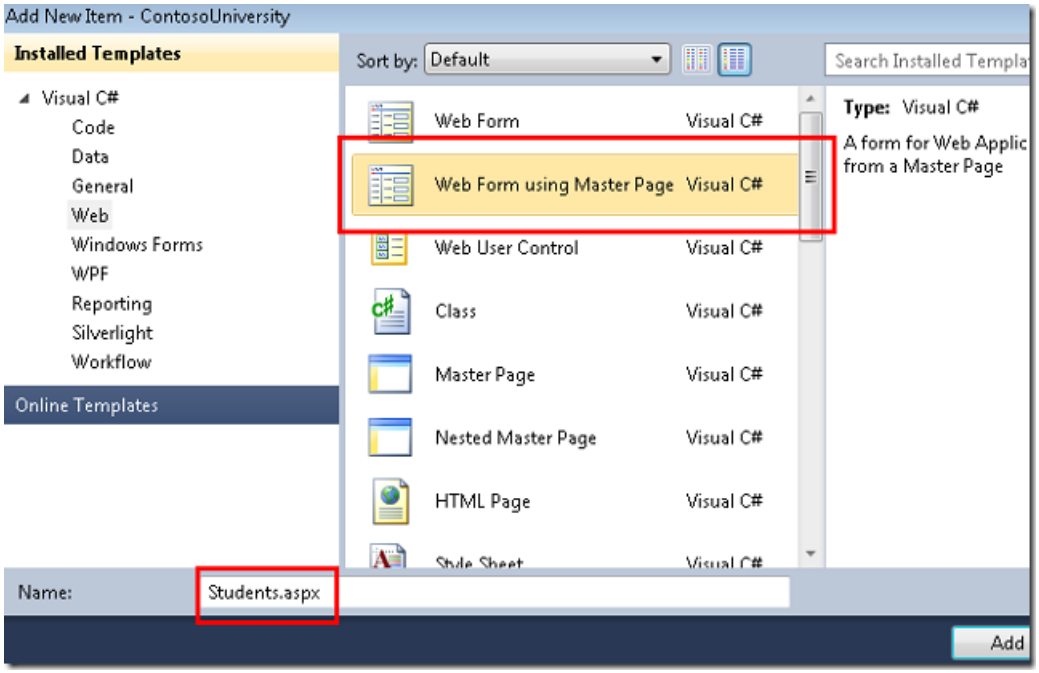
Adding and Configuring the EntityDataSource Control

You'll begin by configuring an `EntityDataSource` control to read `Person` entities from the `People` entity set.

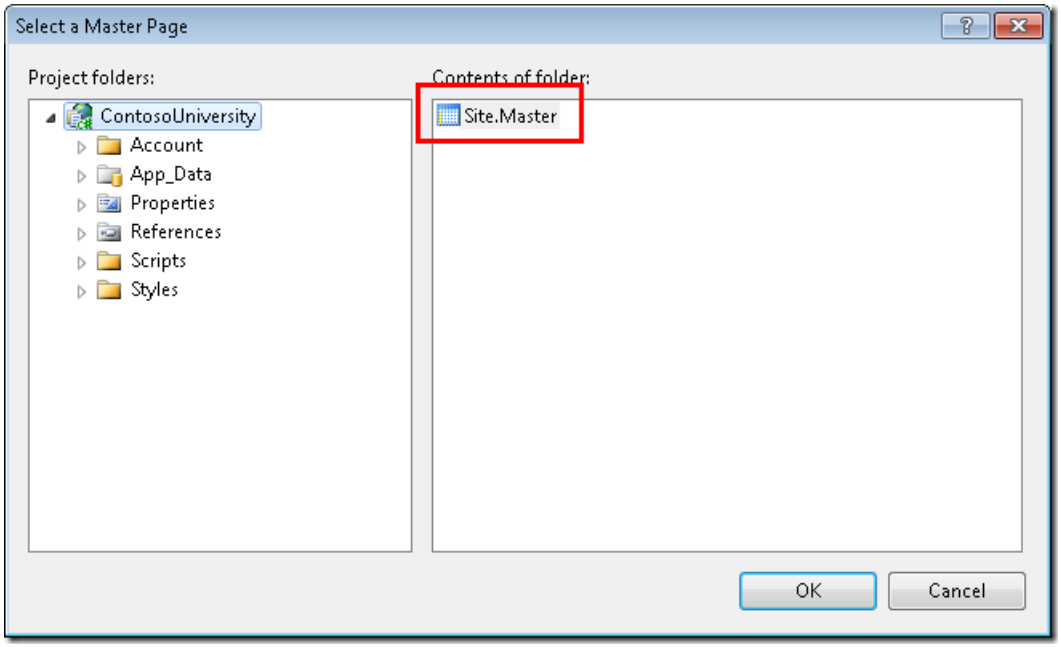
Make sure you have Visual Studio open and that you're working with the project you created in part 1. If you haven't built the project since you created the data model or since the last change you made to it, build the project

now. Changes to the data model are not made available to the designer until the project is built.

Create a new web page using the **Web Form using Master Page** template, and name it *Students.aspx*.



Specify *Site.Master* as the master page. All of the pages you create for these tutorials will use this master page.



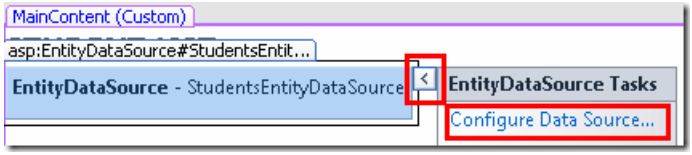
In **Source** view, add an `h2` heading to the `Content` control named `Content2`, as shown in the following example:

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
   <h2>Student List</h2>
</asp:Content>
```

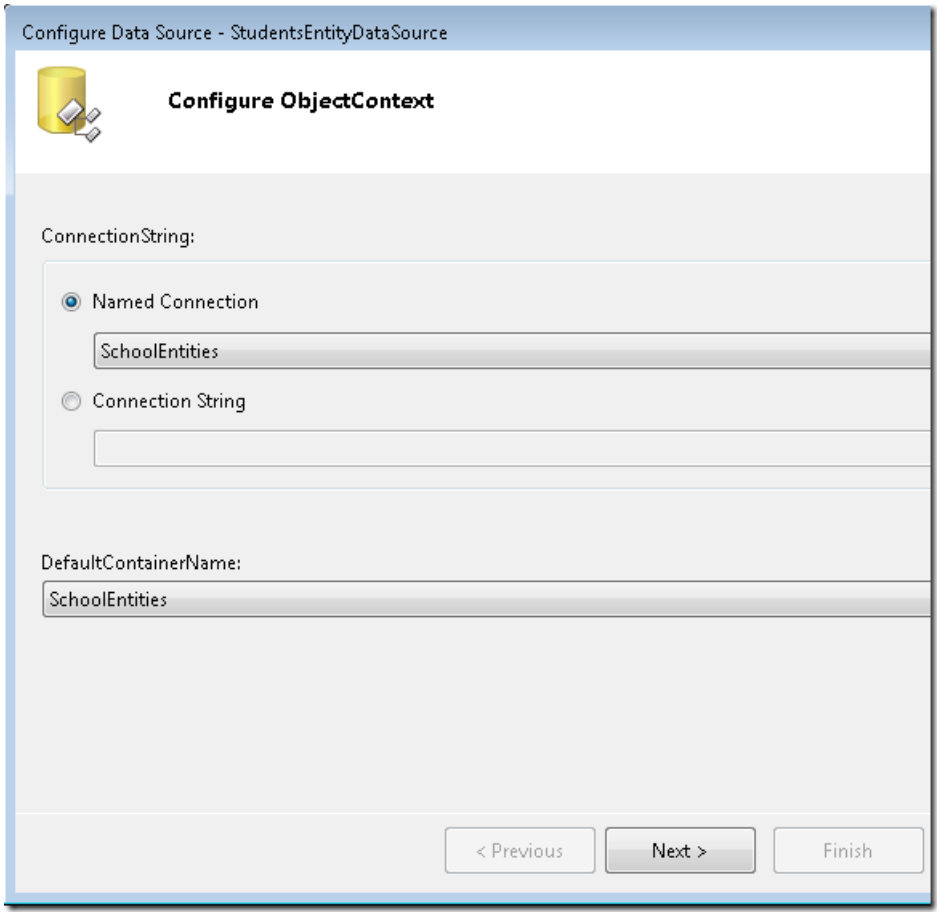
From the **Data** tab of the **Toolbox**, drag an `EntityDataSource` control to the page, drop it below the heading, and change the ID to `StudentsEntityDataSource`:

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Student List</h2>
    <asp:EntityDataSource ID="StudentsEntityDataSource" runat="server">
    </asp:EntityDataSource>
</asp:Content>
```

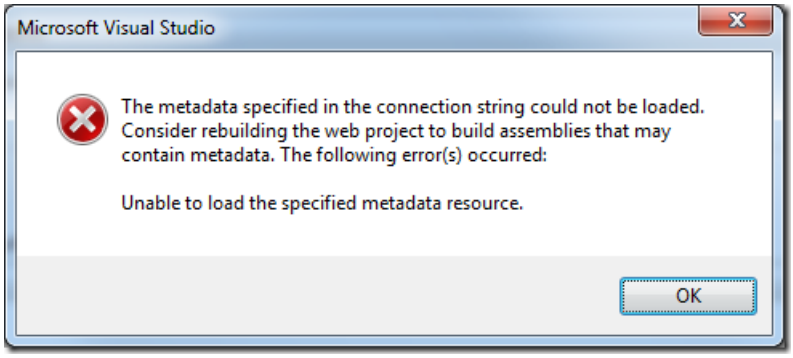
Switch to **Design** view, click the data source control's smart tag, and then click **Configure Data Source** to launch the **Configure Data Source** wizard.



In the **Configure ObjectContext** wizard step, select **SchoolEntities** as the value for **Named Connection**, and select **SchoolEntities** as the **DefaultContainerName** value. Then click **Next**.

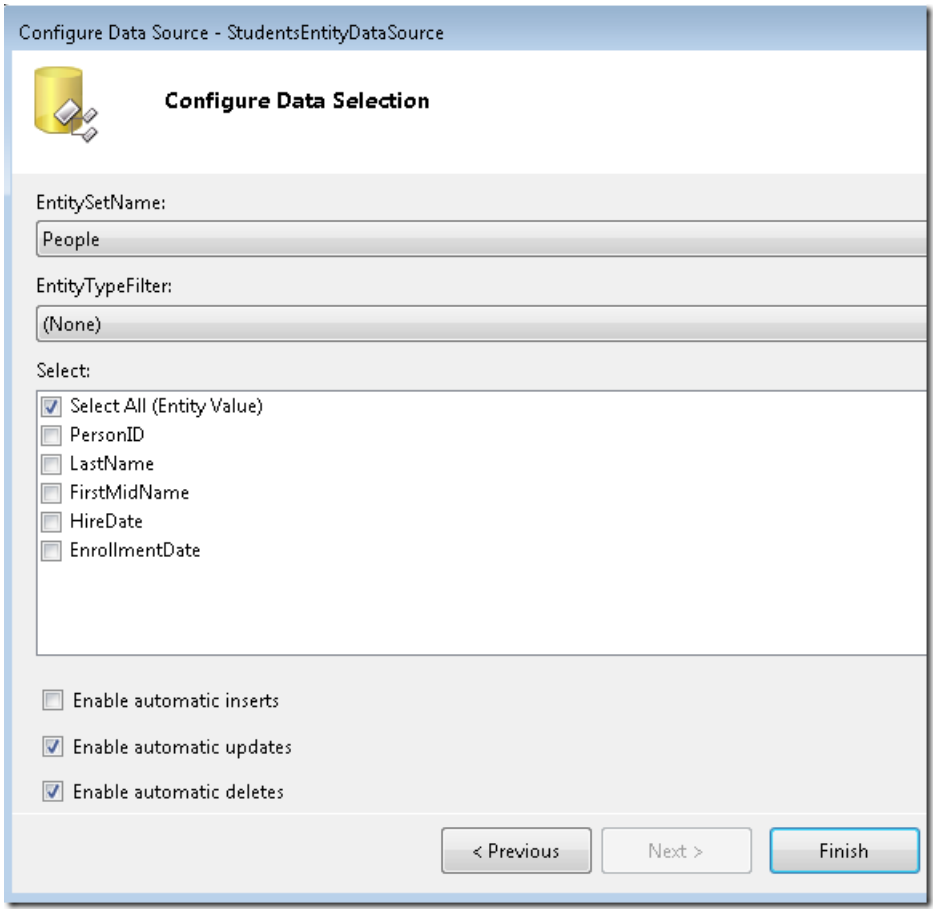


Note: If you get the following dialog box at this point, you have to build the project before proceeding.



In the **Configure Data Selection** step, select **People** as the value for **EntitySetName**. Under **Select**, make sure the **Select A** ll check box is selected. Then select the options to enable update and delete. When you're done, click

Finish.

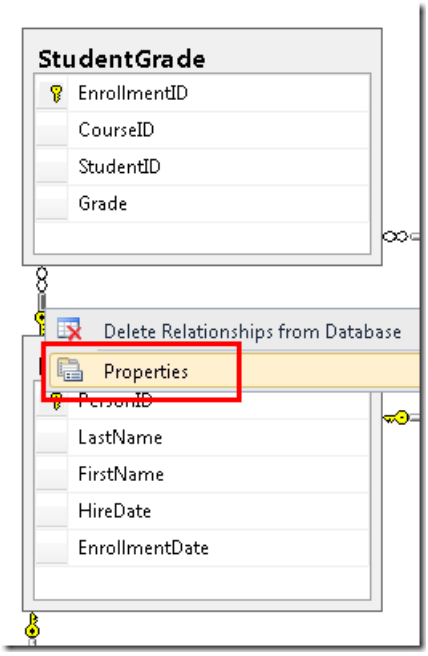


Configuring Database Rules to Allow Deletion

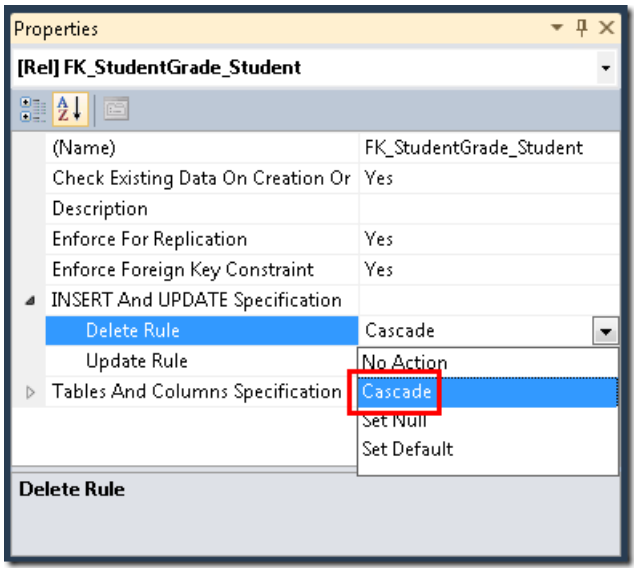
You'll be creating a page that lets users delete students from the `Person` table, which has three relationships with other tables (`Course`, `StudentGrade`, and `OfficeAssignment`). By default, the database will prevent you from deleting a row in `Person` if there are related rows in one of the other tables. You can manually delete the related rows first, or you can configure the database to delete them automatically when you delete a `Person` row. For student records in this tutorial, you'll configure the database to delete the related data automatically. Because students can have related rows only in the `StudentGrade` table, you need to configure only one of the three relationships.

If you're using the *School.mdf* file that you downloaded from the project that goes with this tutorial, you can skip this section because these configuration changes have already been done. If you created the database by running a script, configure the database by performing the following procedures.

In **Server Explorer**, open the database diagram that you created in part 1. Right-click the relationship between `Person` and `StudentGrade` (the line between tables), and then select **Properties**.

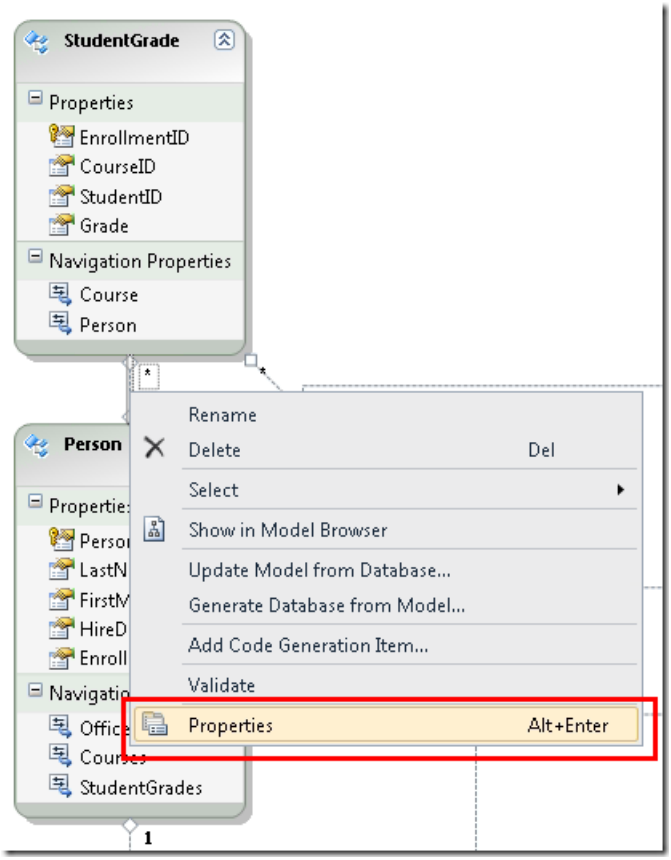


In the **Properties** window, expand **INSERT and UPDATE Specification** and set the **DeleteRule** property to **Cascade**.

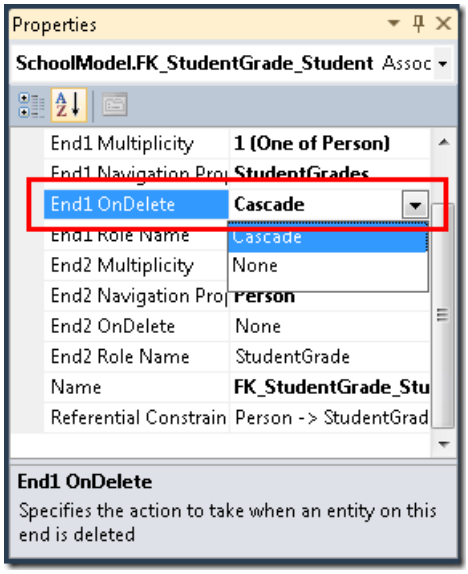


Save and close the diagram. If you're asked whether you want to update the database, click **Yes**.

To make sure that the model keeps entities that are in memory in sync with what the database is doing, you must set corresponding rules in the data model. Open *SchoolModel.edmx*, right-click the association line between `Person` and `StudentGrade`, and then select **Properties**.



In the **Properties** window, set **End1 OnDelete** to **Cascade**.



Save and close the *SchoolModel.edmx* file, and then rebuild the project.

In general, when the database changes, you have several choices for how to sync up the model:

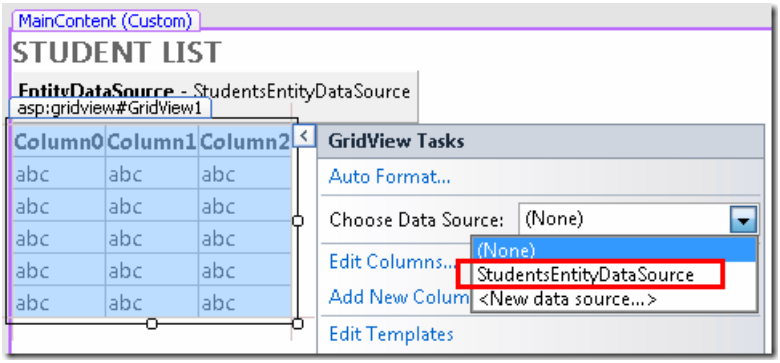
- For certain kinds of changes (such as adding or refreshing tables, views, or stored procedures), right-click in the designer and select **Update Model from Database** to have the designer make the changes automatically.
- Regenerate the data model.
- Make manual updates like this one.

In this case, you could have regenerated the model or refreshed the tables affected by the relationship change, but then you'd have to make the field-name change again (from `FirstName` to `FirstMidName`).

Using a GridView Control to Read and Update Entities

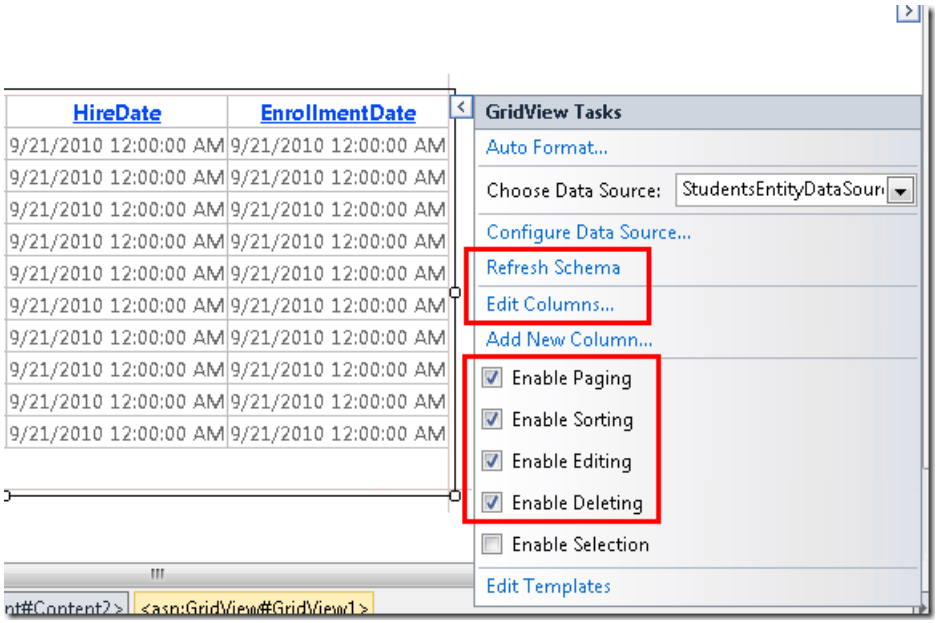
In this section you'll use a `GridView` control to display, update, or delete students.

Open or switch to *Students.aspx* and switch to **Design** view. From the **Data** tab of the **Toolbox**, drag a `GridView` control to the right of the `EntityDataSource` control, name it `StudentsGridView`, click the smart tag, and then select **StudentsEntityDataSource** as the data source.

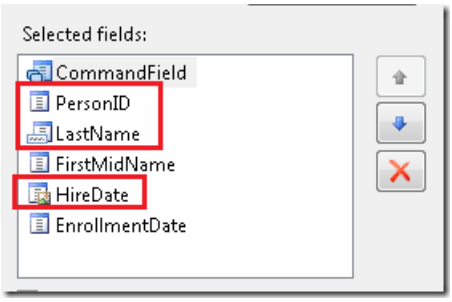


Click **Refresh Schema** (click **Yes** if you're prompted to confirm), then click **Enable Paging**, **Enable Sorting**, **Enable Editing**, and **Enable Deleting**.

Click **Edit Columns**.

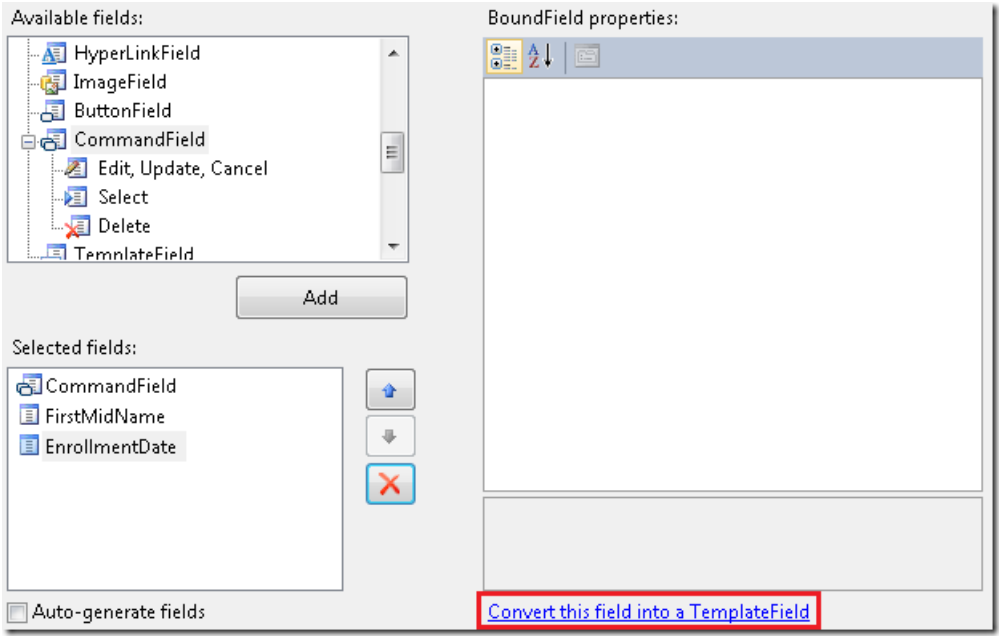


In the **Selected fields** box, delete **PersonID**, **LastName**, and **HireDate**. You typically don't display a record key to users, hire date is not relevant to students, and you'll put both parts of the name in one field, so you only need one of the name fields.)



Select the **FirstMidName** field and then click **Convert this field into a TemplateField**.

Do the same for **EnrollmentDate**.



Click **OK** and then switch to **Source** view. The remaining changes will be easier to do directly in markup. The `GridView` control markup now looks like the following example.

```
<asp:GridView ID="StudentsGridView" runat="server" AllowPaging="True"
        AllowSorting="True" AutoGenerateColumns="False" DataKeyNames="PersonID"
        DataSourceID="StudentsEntityDataSource">
        <Columns>
            <asp:CommandField ShowDeleteButton="True" ShowEditButton="True" />
            <asp:TemplateField HeaderText="FirstMidName" SortExpression="FirstMidName">
                <EditItemTemplate>
                    <asp:TextBox ID="TextBox1" runat="server" Text='<%# Bind("FirstMidName") %>'>
</asp:TextBox>
                </EditItemTemplate>
                <ItemTemplate>
                    <asp:Label ID="Label1" runat="server" Text='<%# Bind("FirstMidName") %>'></asp:Label>
                </ItemTemplate>
            </asp:TemplateField>
            <asp:TemplateField HeaderText="EnrollmentDate" SortExpression="EnrollmentDate">
                <EditItemTemplate>
                    <asp:TextBox ID="TextBox2" runat="server" Text='<%# Bind("EnrollmentDate") %>'>
</asp:TextBox>
                </EditItemTemplate>
                <ItemTemplate>
                    <asp:Label ID="Label2" runat="server" Text='<%# Bind("EnrollmentDate") %>'></asp:Label>
                </ItemTemplate>
            </asp:TemplateField>
        </Columns>
    </asp:GridView>
```

The first column after the command field is a template field that currently displays the first name. Change the markup for this template field to look like the following example:

```
<asp:TemplateField HeaderText="Name" SortExpression="LastName">
                <EditItemTemplate>
                    <asp:TextBox ID="LastNameTextBox" runat="server" Text='<%# Bind("LastName") %>'>
</asp:TextBox>
                    <asp:TextBox ID="FirstNameTextBox" runat="server" Text='<%# Bind("FirstMidName") %>'>
</asp:TextBox>
                </EditItemTemplate>
                <ItemTemplate>
                    <asp:Label ID="LastNameLabel" runat="server" Text='<%# Eval("LastName") %>'></asp:Label>,
                    <asp:Label ID="FirstNameLabel" runat="server" Text='<%# Eval("FirstMidName") %>'>
</asp:Label>
                </ItemTemplate>
            </asp:TemplateField>
```

In display mode, two `Label` controls display the first and last name. In edit mode, two text boxes are provided so you can change the first and last name. As with the `Label` controls in display mode, you use `Bind` and `Eval` expressions exactly as you would with ASP.NET data source controls that connect directly to databases. The only difference is that you're specifying entity properties instead of database columns.

The last column is a template field that displays the enrollment date. Change the markup for this field to look like the following example:

```
<asp:TemplateField HeaderText="Enrollment Date" SortExpression="EnrollmentDate">
                <EditItemTemplate>
                    <asp:TextBox ID="EnrollmentDateTextBox" runat="server" Text='<%# Bind("EnrollmentDate", "
{0:d}") %>'></asp:TextBox>
                </EditItemTemplate>
                <ItemTemplate>
                    <asp:Label ID="EnrollmentDateLabel" runat="server" Text='<%# Eval("EnrollmentDate", "
{0:d}") %>'></asp:Label>
                </ItemTemplate>
            </asp:TemplateField>
```

In both display and edit mode, the format string "{0,d}" causes the date to be displayed in the "short date" format. (Your computer might be configured to display this format differently from the screen images shown in this tutorial.)

Notice that in each of these template fields, the designer used a `Bind` expression by default, but you've changed that to an `Eval` expression in the `ItemTemplate` elements. The `Bind` expression makes the data available in `GridView` control properties in case you need to access the data in code. In this page you don't need to access this data in code, so you can use `Eval`, which is more efficient. For more information, see Getting your data out of the data controls.

Revising EntityDataSource Control Markup to Improve Performance

In the markup for the `EntityDataSource` control, remove the `ConnectionString` and `DefaultContainerName` attributes and replace them with a `ContextTypeName="ContosoUniversity.DAL.SchoolEntities"` attribute. This is a change you should make every time you create an `EntityDataSource` control, unless you need to use a connection that is different from the one that's hard-coded in the object context class. Using the `ContextTypeName` attribute provides the following benefits:

- Better performance. When the `EntityDataSource` control initializes the data model using the `ConnectionString` and `DefaultContainerName` attributes, it performs additional work to load metadata on every request. This isn't necessary if you specify the `ContextTypeName` attribute.
- Lazy loading is turned on by default in generated object context classes (such as `SchoolEntities` in this tutorial) in Entity Framework 4.0. This means that navigation properties are loaded with related data automatically right when you need it. Lazy loading is explained in more detail later in this tutorial.

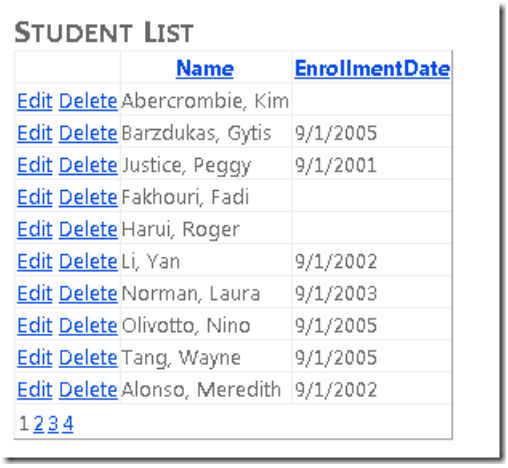
- Any customizations that you've applied to the object context class (in this case, the `SchoolEntities` class) will be available to controls that use the `EntityDataSource` control. Customizing the object context class is an advanced topic that is not covered in this tutorial series. For more information, see Extending Entity Framework Generated Types.

The markup will now resemble the following example (the order of the properties might be different):

```
<asp:EntityDataSource ID="StudentsEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
        EntitySetName="People"
        EnableDelete="True" EnableUpdate="True">
    </asp:EntityDataSource>
```

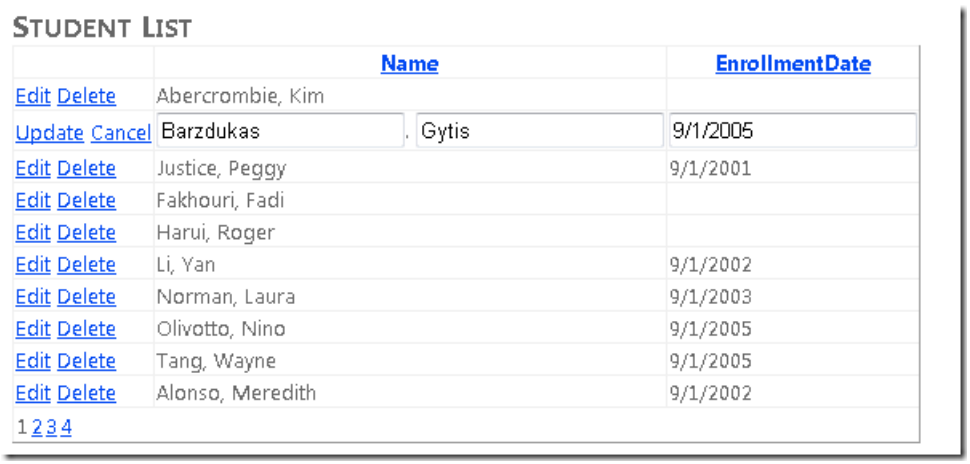
The `EnableFlattening` attribute refers to a feature that was needed in earlier versions of the Entity Framework because foreign key columns were not exposed as entity properties. The current version makes it possible to use *foreign key associations*, which means foreign key properties are exposed for all but many-to-many associations. If your entities have foreign key properties and no complex types, you can leave this attribute set to `False`. Don't remove the attribute from the markup, because the default value is `True`. For more information, see Flattening Objects (EntityDataSource).

Run the page and you see a list of students and employees (you'll filter for just students in the next tutorial). The first name and last name are displayed together.



To sort the display, click a column name.

Click **Edit** in any row. Text boxes are displayed where you can change the first and last name.



The **Delete** button also works. Click delete for a row that has an enrollment date and the row disappears. (Rows without an enrollment date represent instructors and you may get a referential integrity error. In the next tutorial you'll filter this list to include just students.)

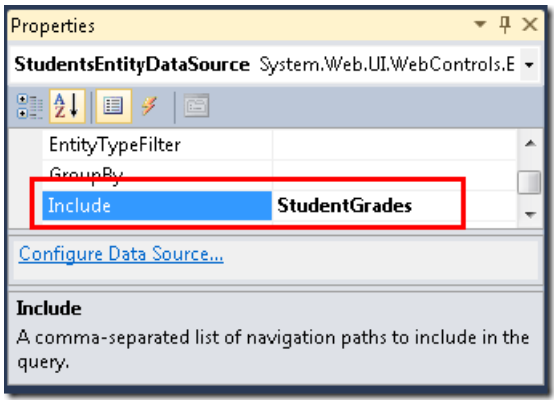
Displaying Data from a Navigation Property

Now suppose you want to know how many courses each student is enrolled in. The Entity Framework provides that information in the `StudentGrades` navigation property of the `Person` entity. Because the database design does not allow a student to be enrolled in a course without having a grade assigned, for this tutorial you can assume that having a row in the `StudentGrade` table row that is associated with a course is the same as being enrolled in the course. (The `Courses` navigation property is only for instructors.)

When you use the `ContextTypeName` attribute of the `EntityDataSource` control, the Entity Framework automatically retrieves information for a navigation property when you access that property. This is called *lazy loading*. However, this can be inefficient, because it results in a separate call to the database each time additional information is needed. If you need data from the navigation property for every entity returned by the `EntityDataSource` control, it's more efficient to retrieve the related data along with the entity itself in a single call to the database. This is called *eager loading*, and you specify eager loading for a navigation property by setting the `Include` property of the `EntityDataSource` control.

In *Students.aspx*, you want to show the number of courses for every student, so eager loading is the best choice. If you were displaying all students but showing the number of courses only for a few of them (which would require writing some code in addition to the markup), lazy loading might be a better choice.

Open or switch to *Students.aspx*, switch to **Design** view, select `StudentsEntityDataSource`, and in the **Properties** window set the **Include** property to **StudentGrades**. (If you wanted to get multiple navigation properties, you could specify their names separated by commas — for example, **StudentGrades, Courses**.)



Switch to **Source** view. In the `StudentsGridView` control, after the last `asp:TemplateField` element, add the following new template field:

```
<asp:TemplateField HeaderText="Number of Courses">
                <ItemTemplate>
                    <asp:Label ID="Label1" runat="server" Text='<%# Eval("StudentGrades.Count") %>'>
</asp:Label>
                </ItemTemplate>
            </asp:TemplateField>
```

In the `Eval` expression, you can reference the navigation property `StudentGrades`. Because this property contains a collection, it has a `Count` property that you can use to display the number of courses in which the student is enrolled. In a later tutorial you'll see how to display data from navigation properties that contain single entities instead of collections. (Note that you cannot use `BoundField` elements to display data from navigation properties.)

Run the page and you now see how many courses each student is enrolled in.

Using a DetailsView Control to Insert Entities

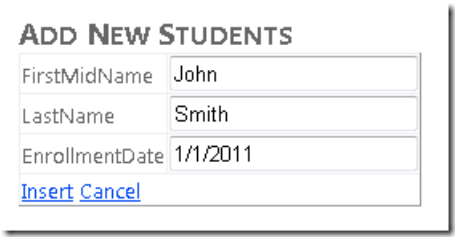
The next step is to create a page that has a `DetailsView` control that will let you add new students. Close the browser and then create a new web page using the *Site.Master* master page. Name the page *StudentsAdd.aspx*, and then switch to **Source** view.

Add the following markup to replace the existing markup for the `Content` control named `Content2`:

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Add New Students</h2>
    <asp:EntityDataSource ID="StudentsEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
        EnableInsert="True" EntitySetName="People">
    </asp:EntityDataSource>
    <asp:DetailsView ID="StudentsDetailsView" runat="server"
        DataSourceID="StudentsEntityDataSource" AutoGenerateRows="False"
        DefaultMode="Insert">
        <Fields>
            <asp:BoundField DataField="FirstMidName" HeaderText="First Name"
                SortExpression="FirstMidName" />
            <asp:BoundField DataField="LastName" HeaderText="Last Name"
                SortExpression="LastName" />
            <asp:BoundField DataField="EnrollmentDate" HeaderText="Enrollment Date"
                SortExpression="EnrollmentDate" />
             <asp:CommandField ShowInsertButton="True" />
       </Fields>
    </asp:DetailsView>
</asp:Content>
```

This markup creates an `EntityDataSource` control that is similar to the one you created in *Students.aspx*, except it enables insertion. As with the `GridView` control, the bound fields of the `DetailsView` control are coded exactly as they would be for a data control that connects directly to a database, except that they reference entity properties. In this case, the `DetailsView` control is used only for inserting rows, so you have set the default mode to `Insert`.

Run the page and add a new student.



Nothing will happen after you insert a new student, but if you now run *Students.aspx*, you'll see the new student information.

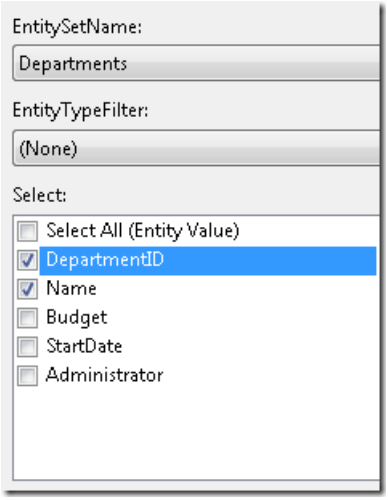
Displaying Data in a Drop-Down List

In the following steps you'll databind a `DropDownList` control to an entity set using an `EntityDataSource` control. In this part of the tutorial, you won't do much with this list. In subsequent parts, though, you'll use the list to let users select a department to display courses associated with the department.

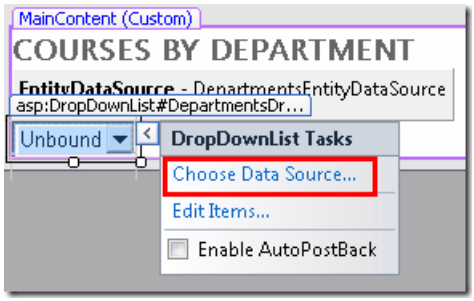
Create a new web page named *Courses.aspx*. In **Source** view, add a heading to the `Content` control that's named `Content2`:

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Courses by Department</h2>
</asp:Content>
```

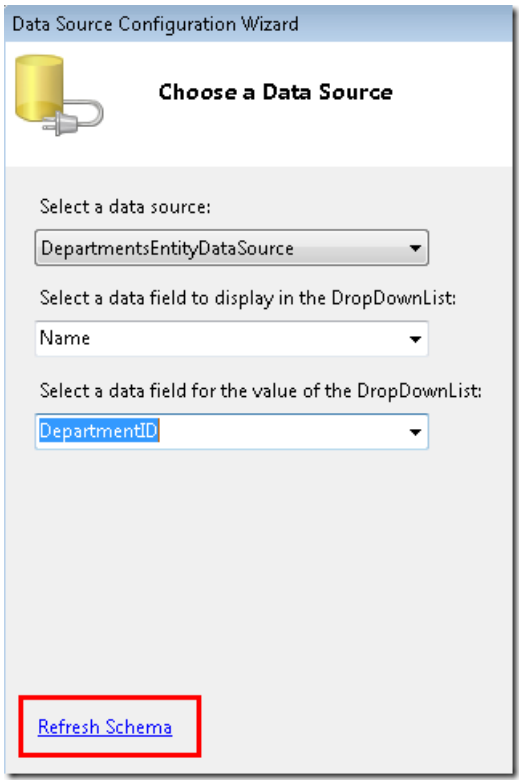
In **Design** view, add an `EntityDataSource` control to the page as you did before, except this time name it `DepartmentsEntityDataSource`. Select **Departments** as the **EntitySetName** value, and select only the **DepartmentID** and **Name** properties.



From the **Standard** tab of the **Toolbox**, drag a `DropDownList` control to the page, name it `DepartmentsDropDownList`, click the smart tag, and select **Choose Data Source** to start the **DataSource Configuration Wizard**.



In the **Choose a Data Source** step, select **DepartmentsEntityDataSource** as the data source, click **Refresh Schema**, and then select **Name** as the data field to display and **DepartmentID** as the value data field. Click **OK**.



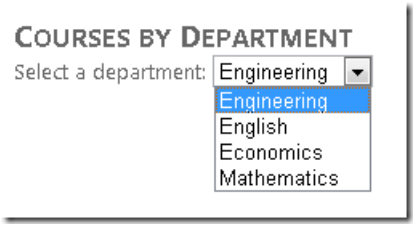
The method you use to databind the control using the Entity Framework is the same as with other ASP.NET data source controls except you're specifying entities and entity properties.

Switch to **Source** view and add "Select a department:" immediately before the `DropDownList` control.

```
Select a department:
    <asp:DropDownList ID="DropDownList1" runat="server"
        DataSourceID="EntityDataSource1" DataTextField="Name"
        DataValueField="DepartmentID">
    </asp:DropDownList>
```

As a reminder, change the markup for the `EntityDataSource` control at this point by replacing the `ConnectionString` and `DefaultContainerName` attributes with a `ContextTypeName="ContosoUniversity.DAL.SchoolEntities"` attribute. It's often best to wait until after you've created the data-bound control that is linked to the data source control before you change the `EntityDataSource` control markup, because after you make the change, the designer will not provide you with a **Refresh Schema** option in the data-bound control.

Run the page and you can select a department from the drop-down list.



This completes the introduction to using the `EntityDataSource` control. Working with this control is generally no different from working with other ASP.NET data source controls, except that you reference entities and properties instead of tables and columns. The only exception is when you want to access navigation properties. In the next tutorial you'll see that the syntax you use with `EntityDataSource` control might also differ from other data source controls when you filter, group, and order data.

Getting Started with Entity Framework 4.0 Database First and ASP.NET 4 Web Forms - Part 3

2/20/2020 • 9 minutes to read • Edit Online

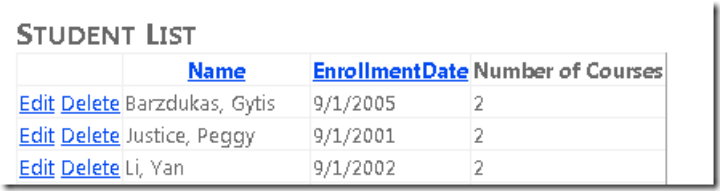
by Tom Dykstra

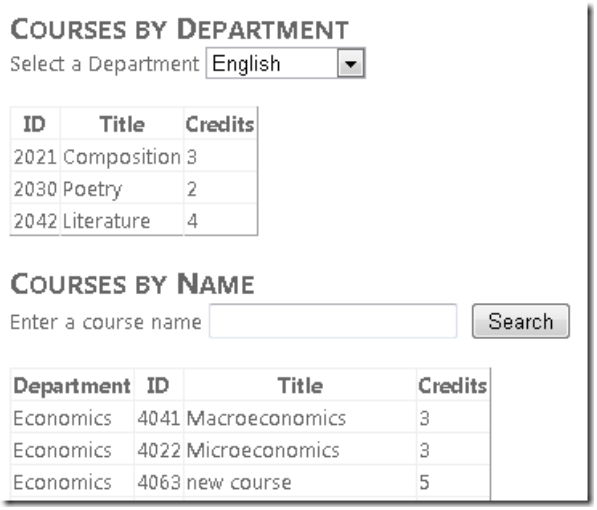
> The Contoso University sample web application demonstrates how to create ASP.NET Web Forms applications using the Entity Framework 4.0 and Visual Studio 2010. For information about the tutorial series, see the first tutorial in the series

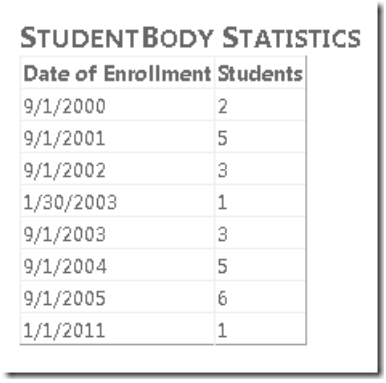
Filtering, Ordering, and Grouping Data

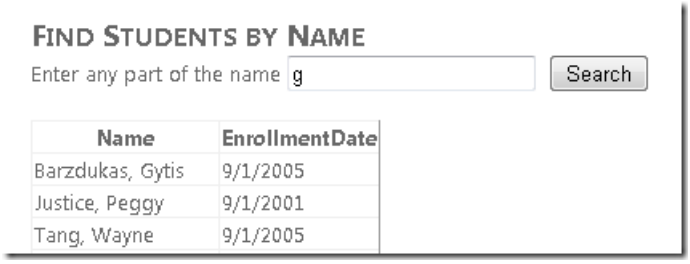
In the previous tutorial you used the `EntityDataSource` control to display and edit data. In this tutorial you'll filter, order, and group data. When you do this by setting properties of the `EntityDataSource` control, the syntax is different from other data source controls. As you'll see, however, you can use the `QueryExtender` control to minimize these differences.

You'll change the *Students.aspx* page to filter for students, sort by name, and search on name. You'll also change the *Courses.aspx* page to display courses for the selected department and search for courses by name. Finally, you'll add student statistics to the *About.aspx* page.





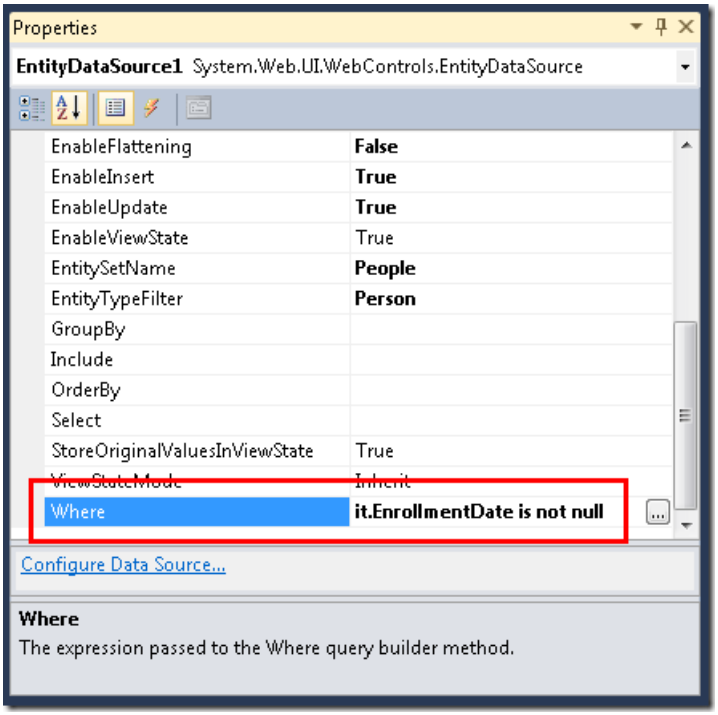




Using the EntityDataSource "Where" Property to Filter Data

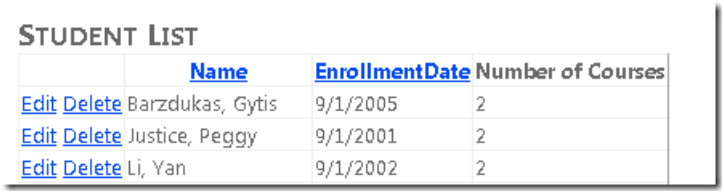
Open the *Students.aspx* page that you created in the previous tutorial. As currently configured, the `GridView` control in the page displays all the names from the `People` entity set. However, you want to show only students, which you can find by selecting `Person` entities that have non-null enrollment dates.

Switch to **Design** view and select the `EntityDataSource` control. In the **Properties** window, set the `Where` property to `it.EnrollmentDate is not null`.



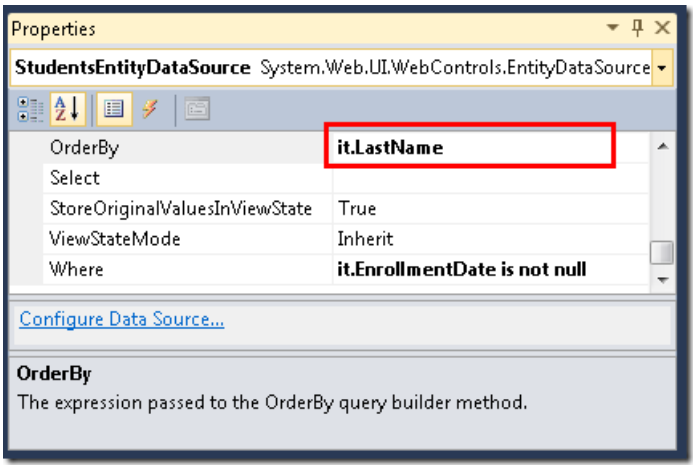
The syntax you use in the `Where` property of the `EntityDataSource` control is Entity SQL. Entity SQL is similar to Transact-SQL, but it's customized for use with entities rather than database objects. In the expression `it.EnrollmentDate is not null`, the word `it` represents a reference to the entity returned by the query. Therefore, `it.EnrollmentDate` refers to the `EnrollmentDate` property of the `Person` entity that the `EntityDataSource` control returns.

Run the page. The students list now contains only students. (There are no rows displayed where there's no enrollment date.)

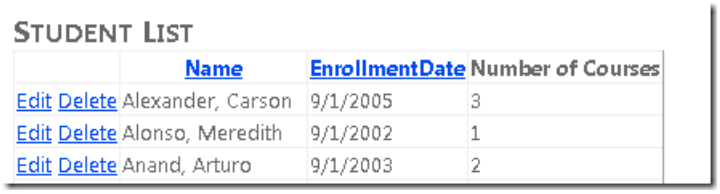


Using the EntityDataSource "OrderBy" Property to Order Data

You also want this list to be in name order when it's first displayed. With the *Students.aspx* page still open in **Design** view, and with the `EntityDataSource` control still selected, in the **Properties** window set the **OrderBy** property to `it.LastName`.



Run the page. The students list is now in order by last name.

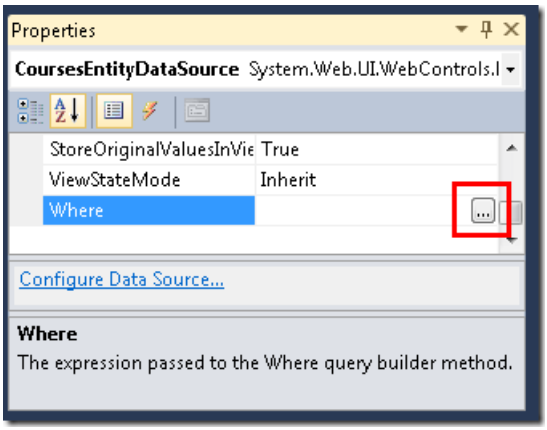


Using a Control Parameter to Set the "Where" Property

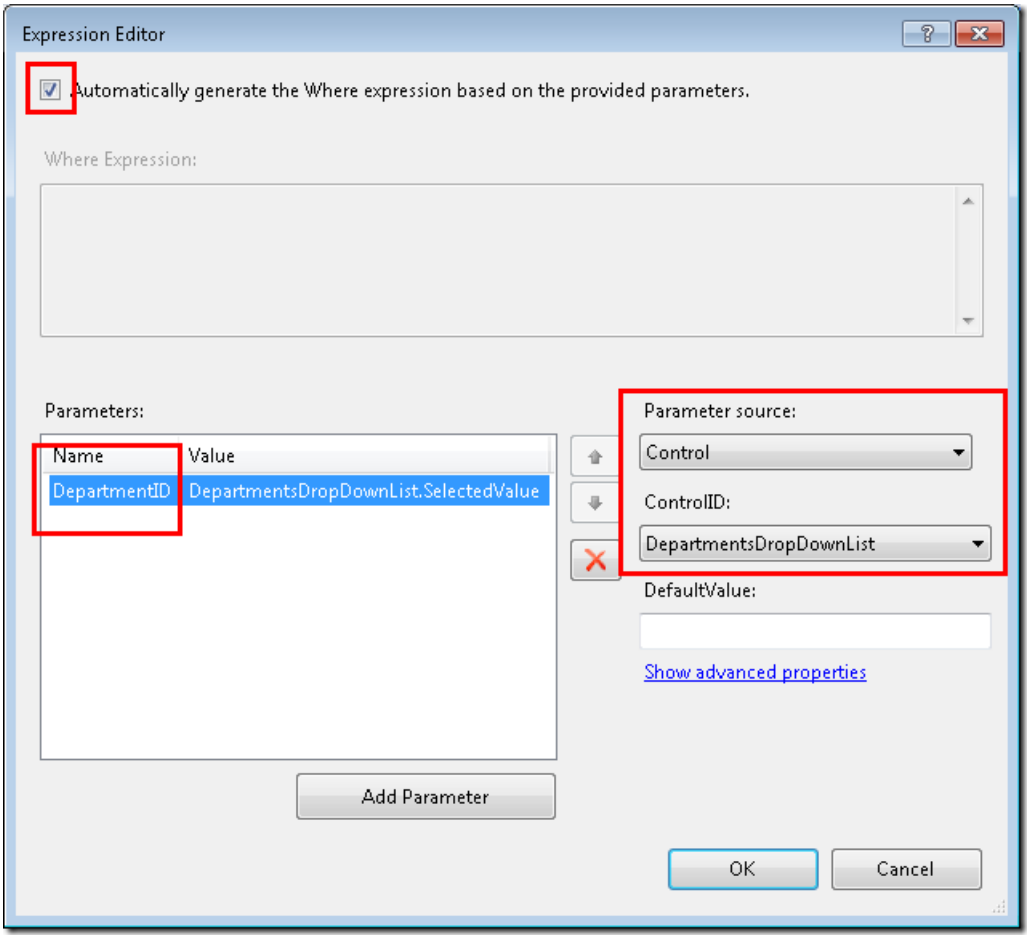
As with other data source controls, you can pass parameter values to the `Where` property. On the *Courses.aspx* page that you created in part 2 of the tutorial, you can use this method to display courses that are associated with the department that a user selects from the drop-down list.

Open *Courses.aspx* and switch to **Design** view. Add a second `EntityDataSource` control to the page, and name it `CoursesEntityDataSource`. Connect it to the `SchoolEntities` model, and select `Courses` as the **EntitySetName** value.

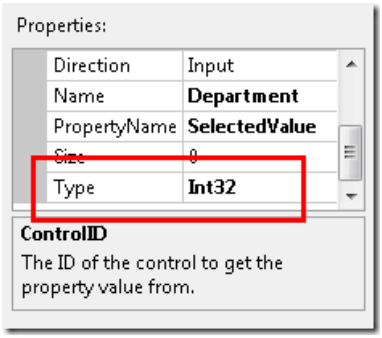
In the **Properties** window, click the ellipsis in the **Where** property box. (Make sure the `CoursesEntityDataSource` control is still selected before using the **Properties** window.)



The **Expression Editor** dialog box is displayed. In this dialog box, select **Automatically generate the Where expression based on the provided parameters**, and then click **Add Parameter**. Name the parameter `DepartmentID`, select **Control** as the **Parameter source** value, and select **DepartmentsDropDownList** as the **ControlID** value.



Click **Show advanced properties**, and in the **Properties** window of the **Expression Editor** dialog box, change the `Type` property to `Int32`.

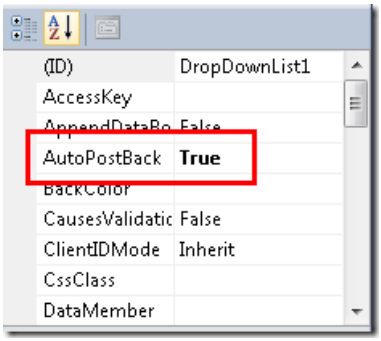


When you're done, click **OK**.

Below the drop-down list, add a `GridView` control to the page and name it `CoursesGridView`. Connect it to the `CoursesEntityDataSource` data source control, click **Refresh Schema**, click **Edit Columns**, and remove the `DepartmentID` column. The `GridView` control markup resembles the following example.

```
<asp:GridView ID="CoursesGridView" runat="server" AutoGenerateColumns="False"
        DataKeyNames="CourseID" DataSourceID="CoursesEntityDataSource">
        <Columns>
            <asp:BoundField DataField="CourseID" HeaderText="ID" ReadOnly="True"
                SortExpression="CourseID" />
            <asp:BoundField DataField="Title" HeaderText="Title" SortExpression="Title" />
            <asp:BoundField DataField="Credits" HeaderText="Credits"
                SortExpression="Credits" />
        </Columns>
    </asp:GridView>
```

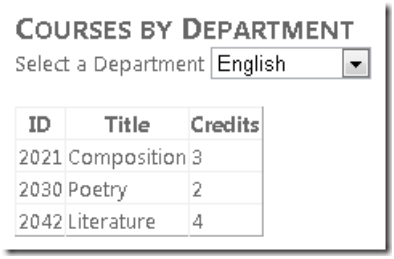
When the user changes the selected department in the drop-down list, you want the list of associated courses to change automatically. To make this happen, select the drop-down list, and in the **Properties** window set the `AutoPostBack` property to `True`.



Now that you're finished using the designer, switch to **Source** view and replace the `ConnectionString` and `DefaultContainer` name properties of the `CoursesEntityDataSource` control with the `ContextTypeName="ContosoUniversity.DAL.SchoolEntities"` attribute. When you're done, the markup for the control will look like the following example.

```
<asp:EntityDataSource ID="CoursesEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="false"
        EntitySetName="Courses"
        AutoGenerateWhereClause="true" Where="">
        <WhereParameters>
            <asp:ControlParameter ControlID="DepartmentsDropDownList" Type="Int32"
                Name="DepartmentID" PropertyName="SelectedValue" />
        </WhereParameters>
    </asp:EntityDataSource>
```

Run the page and use the drop-down list to select different departments. Only courses that are offered by the selected department are displayed in the `GridView` control.



Using the EntityDataSource "GroupBy" Property to Group Data

Suppose Contoso University wants to put some student-body statistics on its About page. Specifically, it wants to show a breakdown of numbers of students by the date they enrolled.

Open *About.aspx*, and in **Source** view, replace the existing contents of the `BodyContent` control with "Student Body Statistics" between `h2` tags:

```
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <h2>Student Body Statistics</h2>
</asp:Content>
```

After the heading, add an `EntityDataSource` control and name it `StudentStatisticsEntityDataSource`. Connect it to `SchoolEntities`, select the `People` entity set, and leave the **Select** box in the wizard unchanged. Set the following properties in the **Properties** window:

- To filter for students only, set the `Where` property to `it.EnrollmentDate is not null`.
- To group the results by the enrollment date, set the `GroupBy` property to `it.EnrollmentDate`.
- To select the enrollment date and the number of students, set the `Select` property to `it.EnrollmentDate, Count(it.EnrollmentDate) AS NumberOfStudents`.
- To order the results by the enrollment date, set the `OrderBy` property to `it.EnrollmentDate`.

In **Source** view, replace the `ConnectionString` and `DefaultContainer` name properties with a `ContextTypeName` property. The `EntityDataSource` control markup now resembles the following example.

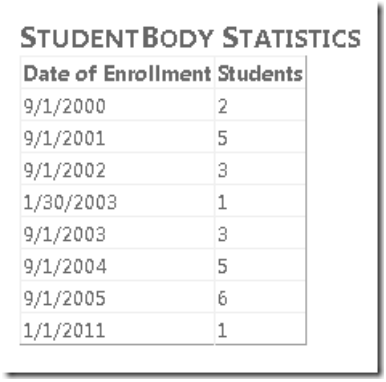
```
<asp:EntityDataSource ID="StudentStatisticsEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
        EntitySetName="People"
        Select="it.EnrollmentDate, Count(it.EnrollmentDate) AS NumberOfStudents"
        OrderBy="it.EnrollmentDate" GroupBy="it.EnrollmentDate"
        Where="it.EnrollmentDate is not null" >
    </asp:EntityDataSource>
```

The syntax of the `Select`, `GroupBy`, and `Where` properties resembles Transact-SQL except for the `it` keyword that specifies the current entity.

Add the following markup to create a `GridView` control to display the data.

```
<asp:GridView ID="StudentStatisticsGridView" runat="server" AutoGenerateColumns="False"
        DataSourceID="StudentStatisticsEntityDataSource">
        <Columns>
            <asp:BoundField DataField="EnrollmentDate" DataFormatString="{0:d}"
                HeaderText="Date of Enrollment"
                ReadOnly="True" SortExpression="EnrollmentDate" />
            <asp:BoundField DataField="NumberOfStudents" HeaderText="Students"
                ReadOnly="True" SortExpression="NumberOfStudents" />
        </Columns>
    </asp:GridView>
```

Run the page to see a list showing the number of students by enrollment date.



STUDENTBODY STATISTICS

Date of Enrollment	Students
9/1/2000	2
9/1/2001	5
9/1/2002	3
1/30/2003	1
9/1/2003	3
9/1/2004	5
9/1/2005	6
1/1/2011	1

Using the QueryExtender Control for Filtering and Ordering

The `QueryExtender` control provides a way to specify filtering and sorting in markup. The syntax is independent of the database management system (DBMS) you're using. It's also generally independent of the Entity Framework, with the exception that syntax you use for navigation properties is unique to the Entity Framework.

In this part of the tutorial you'll use a `QueryExtender` control to filter and order data, and one of the order-by fields will be a navigation property.

(If you prefer to use code instead of markup to extend the queries that are automatically generated by the `EntityDataSource` control, you can do that by handling the `QueryCreated` event. This is how the `QueryExtender` control extends `EntityDataSource` control queries also.)

Open the *Courses.aspx* page, and below the markup you added previously, insert the following markup to create a heading, a text box for entering search strings, a search button, and an `EntityDataSource` control that's bound to the `Courses` entity set.

```
<h2>Courses by Name</h2>
    Enter a course name
    <asp:TextBox ID="SearchTextBox" runat="server"></asp:TextBox>
     <asp:Button ID="SearchButton" runat="server" Text="Search" />
    <br /><br />
    <asp:EntityDataSource ID="SearchEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
        EntitySetName="Courses"
        Include="Department" >
    </asp:EntityDataSource>
```

Notice that the `EntityDataSource` control's `Include` property is set to `Department`. In the database, the `Course` table does not contain the department name; it contains a `DepartmentID` foreign key column. If you were querying the database directly, to get the department name along with course data, you would have to join the `Course` and `Department` tables. By setting the `Include` property to `Department`, you specify that the Entity Framework should do the work of getting the related `Department` entity when it gets a `Course` entity. The `Department` entity is then stored in the `Department` navigation property of the `Course` entity. (By default, the `SchoolEntities` class that was generated by the data model designer retrieves related data when it's needed, and you've bound the data source control to that class, so setting the `Include` property is not necessary. However, setting it improves performance of the page, because otherwise the Entity Framework would make separate calls to the database to retrieve data for the `Course` entities and for the related `Department` entities.)

After the `EntityDataSource` control you just created, insert the following markup to create a `QueryExtender` control that's bound to that `EntityDataSource` control.

```
<asp:QueryExtender ID="SearchQueryExtender" runat="server"
        TargetControlID="SearchEntityDataSource" >
        <asp:SearchExpression SearchType="StartsWith" DataFields="Title">
            <asp:ControlParameter ControlID="SearchTextBox" />
        </asp:SearchExpression>
        <asp:OrderByExpression DataField="Department.Name" Direction="Ascending">
            <asp:ThenBy DataField="Title" Direction="Ascending" />
        </asp:OrderByExpression>
    </asp:QueryExtender>
```

The `SearchExpression` element specifies that you want to select courses whose titles match the value entered in the text box. Only as many characters as are entered in the text box will be compared, because the `SearchType` property specifies `StartsWith`.

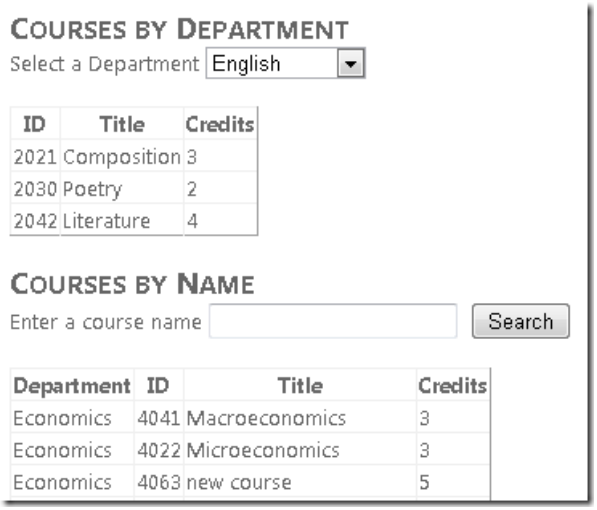
The `OrderByExpression` element specifies that the result set will be ordered by course title within department name. Notice how department name is specified: `Department.Name`. Because the association between the `Course` entity and the `Department` entity is one-to-one, the `Department` navigation property contains a `Department` entity. (If this were a one-to-many relationship, the property would contain a collection.) To get the department name, you must specify the `Name` property of the `Department` entity.

Finally, add a `GridView` control to display the list of courses:

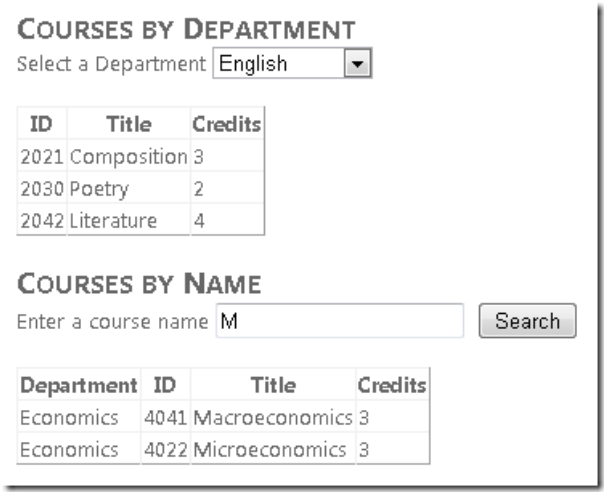
```
<asp:GridView ID="SearchGridView" runat="server" AutoGenerateColumns="False"
        DataKeyNames="CourseID" DataSourceID="SearchEntityDataSource"  AllowPaging="true">
        <Columns>
            <asp:TemplateField HeaderText="Department">
                <ItemTemplate>
                    <asp:Label ID="Label2" runat="server" Text='<%# Eval("Department.Name") %>'></asp:Label>
                </ItemTemplate>
            </asp:TemplateField>
            <asp:BoundField DataField="CourseID" HeaderText="ID"/>
            <asp:BoundField DataField="Title" HeaderText="Title" />
            <asp:BoundField DataField="Credits" HeaderText="Credits" />
        </Columns>
    </asp:GridView>
```

The first column is a template field that displays the department name. The databinding expression specifies `Department.Name`, just as you saw in the `QueryExtender` control.

Run the page. The initial display shows a list of all courses in order by department and then by course title.



Enter an "m" and click **Search** to see all courses whose titles begin with "m" (the search is not case sensitive).



Using the "Like" Operator to Filter Data

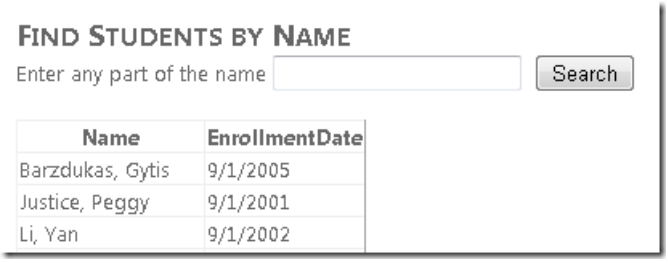
You can achieve an effect similar to the `QueryExtender` control's `StartsWith`, `Contains`, and `EndsWith` search types by using a `Like` operator in the `EntityDataSource` control's `Where` property. In this part of the tutorial, you'll see how to use the `Like` operator to search for a student by name.

Open *Students.aspx* in **Source** view. After the `GridView` control, add the following markup:

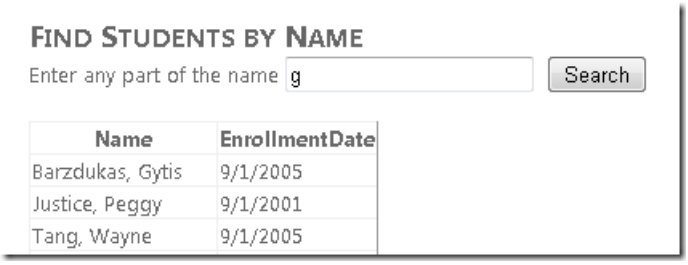
```
<h2>Find Students by Name</h2>
    Enter any part of the name
    <asp:TextBox ID="SearchTextBox" runat="server" AutoPostBack="true"></asp:TextBox>
     <asp:Button ID="SearchButton" runat="server" Text="Search" />
    <br />
    <br />
    <asp:EntityDataSource ID="SearchEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
        EntitySetName="People"
        Where="it.EnrollmentDate is not null and (it.FirstMidName Like '%' + @StudentName + '%' or
it.LastName Like '%' + @StudentName + '%')" >
        <WhereParameters>
            <asp:ControlParameter ControlID="SearchTextBox" Name="StudentName" PropertyName="Text"
             Type="String" DefaultValue="%"/>
        </WhereParameters>
    </asp:EntityDataSource>
    <asp:GridView ID="SearchGridView" runat="server" AutoGenerateColumns="False" DataKeyNames="PersonID"
        DataSourceID="SearchEntityDataSource" AllowPaging="true">
        <Columns>
            <asp:TemplateField HeaderText="Name" SortExpression="LastName, FirstMidName">
                <ItemTemplate>
                    <asp:Label ID="LastNameFoundLabel" runat="server" Text='<%# Eval("LastName") %>'>
</asp:Label>,
                    <asp:Label ID="FirstNameFoundLabel" runat="server" Text='<%# Eval("FirstMidName") %>'>
</asp:Label>
                </ItemTemplate>
            </asp:TemplateField>
            <asp:TemplateField HeaderText="Enrollment Date" SortExpression="EnrollmentDate">
                <ItemTemplate>
                    <asp:Label ID="EnrollmentDateFoundLabel" runat="server" Text='<%# Eval("EnrollmentDate",
"{0:d}") %>'></asp:Label>
                </ItemTemplate>
            </asp:TemplateField>
        </Columns>
    </asp:GridView>
```

This markup is similar to what you've seen earlier except for the `Where` property value. The second part of the `Where` expression defines a substring search (`LIKE %FirstMidName% or LIKE %LastName%`) that searches both the first and last names for whatever is entered in the text box.

Run the page. Initially you see all of the students because the default value for the `StudentName` parameter is "%".



Enter the letter "g" in the text box and click **Search**. You see a list of students that have a "g" in either the first or last name.



You've now displayed, updated, filtered, ordered, and grouped data from individual tables. In the next tutorial you'll begin to work with related data (master-detail scenarios).

PREVIOUS NEXT

Getting Started with Entity Framework 4.0 Database First and ASP.NET 4 Web Forms - Part 4

2/20/2020 • 10 minutes to read • Edit Online

by Tom Dykstra

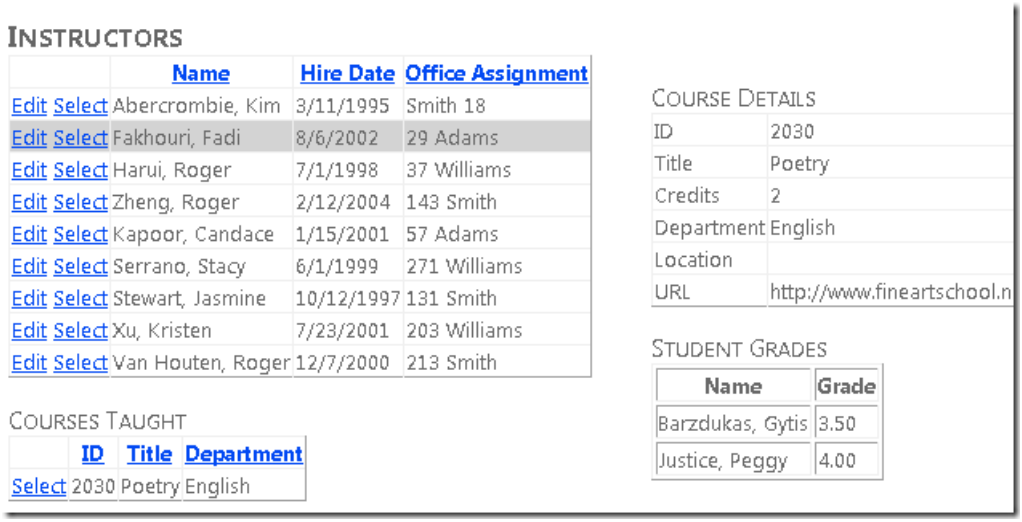
> The Contoso University sample web application demonstrates how to create ASP.NET Web Forms applications using the Entity Framework 4.0 and Visual Studio 2010. For information about the tutorial series, see the first tutorial in the series

Working with Related Data

In the previous tutorial you used the `EntityDataSource` control to filter, sort, and group data. In this tutorial you'll display and update related data.

You'll create the Instructors page that shows a list of instructors. When you select an instructor, you see a list of courses taught by that instructor. When you select a course, you see details for the course and a list of students enrolled in the course. You can edit the instructor name, hire date, and office assignment. The office assignment is a separate entity set that you access through a navigation property.

You can link master data to detail data in markup or in code. In this part of the tutorial, you'll use both methods.



Displaying and Updating Related Entities in a GridView Control

Create a new web page named *Instructors.aspx* that uses the *Site.Master* master page, and add the following markup to the `Content` control named `Content2`:

```
<h2>Instructors</h2>
    <div>
        <asp:EntityDataSource ID="InstructorsEntityDataSource" runat="server"
            ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
            EntitySetName="People"
            Where="it.HireDate is not null" Include="OfficeAssignment" EnableUpdate="True">
        </asp:EntityDataSource>
    </div>
```

This markup creates an `EntityDataSource` control that selects instructors and enables updates. The `div` element configures markup to render on the left so that you can add a column on the right later.

Between the `EntityDataSource` markup and the closing `</div>` tag, add the following markup that creates a `GridView` control and a `Label` control that you'll use for error messages:

```
<asp:GridView ID="InstructorsGridView" runat="server" AllowPaging="True" AllowSorting="True"
            AutoGenerateColumns="False" DataKeyNames="PersonID" DataSourceID="InstructorsEntityDataSource"
            OnSelectedIndexChanged="InstructorsGridView_SelectedIndexChanged"
            SelectedRowStyle-BackColor="LightGray"
            onrowupdating="InstructorsGridView_RowUpdating">
            <Columns>
                <asp:CommandField ShowSelectButton="True" ShowEditButton="True" />
                <asp:TemplateField HeaderText="Name" SortExpression="LastName">
                    <ItemTemplate>
                        <asp:Label ID="InstructorLastNameLabel" runat="server" Text='<%# Eval("LastName")
%>'></asp:Label>,
                        <asp:Label ID="InstructorFirstNameLabel" runat="server" Text='<%#
Eval("FirstMidName") %>'></asp:Label>
                    </ItemTemplate>
                    <EditItemTemplate>
                        <asp:TextBox ID="InstructorLastNameTextBox" runat="server" Text='<%#
Bind("FirstMidName") %>' Width="7em"></asp:TextBox>
                        <asp:TextBox ID="InstructorFirstNameTextBox" runat="server" Text='<%#
Bind("LastName") %>' Width="7em"></asp:TextBox>
                    </EditItemTemplate>
                </asp:TemplateField>
                <asp:TemplateField HeaderText="Hire Date" SortExpression="HireDate">
                    <ItemTemplate>
                        <asp:Label ID="InstructorHireDateLabel" runat="server" Text='<%# Eval("HireDate", "
{0:d}") %>'></asp:Label>
                    </ItemTemplate>
                    <EditItemTemplate>
                        <asp:TextBox ID="InstructorHireDateTextBox" runat="server" Text='<%# Bind("HireDate",
"{0:d}") %>' Width="7em"></asp:TextBox>
                    </EditItemTemplate>
                </asp:TemplateField>
                <asp:TemplateField HeaderText="Office Assignment" SortExpression="OfficeAssignment.Location">
                    <ItemTemplate>
                        <asp:Label ID="InstructorOfficeLabel" runat="server" Text='<%#
Eval("OfficeAssignment.Location") %>'></asp:Label>
                    </ItemTemplate>
                    <EditItemTemplate>
                        <asp:TextBox ID="InstructorOfficeTextBox" runat="server"
                        Text='<%# Eval("OfficeAssignment.Location") %>' Width="7em"
                        oninit="InstructorOfficeTextBox_Init"></asp:TextBox>
                    </EditItemTemplate>
                </asp:TemplateField>
            </Columns>
            <SelectedRowStyle BackColor="LightGray"></SelectedRowStyle>
        </asp:GridView>
        <asp:Label ID="ErrorMessageLabel" runat="server" Text="" Visible="false" ViewStateMode="Disabled">
</asp:Label>
```

This `GridView` control enables row selection, highlights the selected row with a light gray background color, and specifies handlers (which you'll create later) for the `SelectedIndexChanged` and `Updating` events. It also specifies `PersonID` for the `DataKeyNames` property, so that the key value of the selected row can be passed to another control that you'll add later.

The last column contains the instructor's office assignment, which is stored in a navigation property of the `Person` entity because it comes from an associated entity. Notice that the `EditItemTemplate` element specifies `Eval` instead of `Bind`, because the `GridView` control cannot directly bind to navigation properties in order to update them. You'll update the office assignment in code. To do that, you'll need a reference to the `TextBox`

control, and you'll get and save that in the `TextBox` control's `Init` event.

Following the `GridView` control is a `Label` control that's used for error messages. The control's `Visible` property is `false`, and view state is turned off, so that the label will appear only when code makes it visible in response to an error.

Open the *Instructors.aspx.cs* file and add the following `using` statement:

```
using ContosoUniversity.DAL;
```

Add a private class field immediately after the partial-class name declaration to hold a reference to the office assignment text box.

```
private TextBox instructorOfficeTextBox;
```

Add a stub for the `SelectedIndexChanged` event handler that you'll add code for later. Also add a handler for the office assignment `TextBox` control's `Init` event so that you can store a reference to the `TextBox` control. You'll use this reference to get the value the user entered in order to update the entity associated with the navigation property.

```
protected void InstructorsGridView_SelectedIndexChanged(object sender, EventArgs e)
{
}

protected void InstructorOfficeTextBox_Init(object sender, EventArgs e)
{
    instructorOfficeTextBox = sender as TextBox;
}
```

You'll use the `GridView` control's `Updating` event to update the `Location` property of the associated `OfficeAssignment` entity. Add the following handler for the `Updating` event:

```
protected void InstructorsGridView_RowUpdating(object sender, GridViewUpdateEventArgs e)
{
    using (var context = new SchoolEntities())
    {
        var instructorBeingUpdated = Convert.ToInt32(e.Keys[0]);
        var officeAssignment = (from o in context.OfficeAssignments
                                where o.InstructorID == instructorBeingUpdated
                                select o).FirstOrDefault();

        try
        {
            if (String.IsNullOrWhiteSpace(instructorOfficeTextBox.Text) == false)
            {
                if (officeAssignment == null)
                {

context.OfficeAssignments.AddObject(OfficeAssignment.CreateOfficeAssignment(instructorBeingUpdated,
instructorOfficeTextBox.Text, null));
                }
                else
                {
                    officeAssignment.Location = instructorOfficeTextBox.Text;
                }
            }
            else
            {
                if (officeAssignment != null)
                {
                    context.DeleteObject(officeAssignment);
                }
            }
            context.SaveChanges();
        }
        catch (Exception)
        {
            e.Cancel = true;
            ErrorMessageLabel.Visible = true;
            ErrorMessageLabel.Text = "Update failed.";
            //Add code to log the error.
        }
    }
}
```

This code is run when the user clicks **Update** in a `GridView` row. The code uses LINQ to Entities to retrieve the `OfficeAssignment` entity that's associated with the current `Person` entity, using the `PersonID` of the selected row from the event argument.

The code then takes one of the following actions depending on the value in the `InstructorOfficeTextBox` control:

- If the text box has a value and there's no `OfficeAssignment` entity to update, it creates one.
- If the text box has a value and there's an `OfficeAssignment` entity, it updates the `Location` property value.
- If the text box is empty and an `OfficeAssignment` entity exists, it deletes the entity.

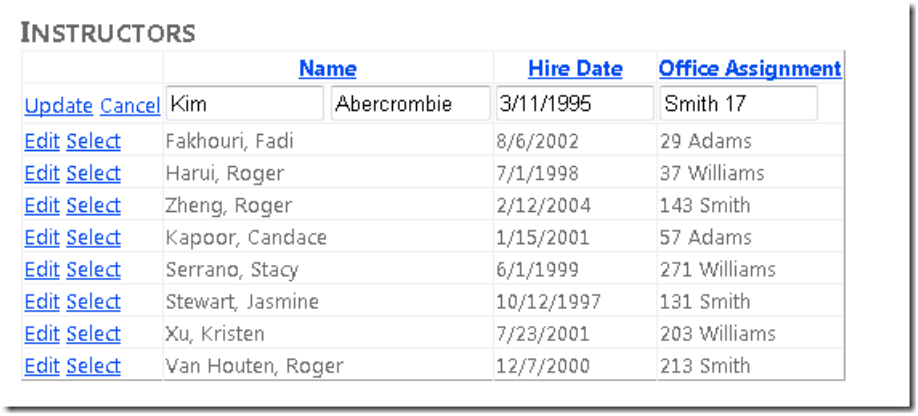
After this, it saves the changes to the database. If an exception occurs, it displays an error message.

Run the page.

INSTRUCTORS

	Name	Hire Date	Office Assignment
Edit Select	Abercrombie, Kim	3/11/1995	Smith 17
Edit Select	Fakhouri, Fadi	8/6/2002	29 Adams
Edit Select	Harui, Roger	7/1/1998	37 Williams
Edit Select	Zheng, Roger	2/12/2004	143 Smith
Edit Select	Kapoor, Candace	1/15/2001	57 Adams
Edit Select	Serrano, Stacy	6/1/1999	271 Williams
Edit Select	Stewart, Jasmine	10/12/1997	131 Smith
Edit Select	Xu, Kristen	7/23/2001	203 Williams
Edit Select	Van Houten, Roger	12/7/2000	213 Smith

Click **Edit** and all fields change to text boxes.



INSTRUCTORS

	Name		Hire Date	Office Assignment
Update Cancel	Kim	Abercrombie	3/11/1995	Smith 17
Edit Select	Fakhouri, Fadi		8/6/2002	29 Adams
Edit Select	Harui, Roger		7/1/1998	37 Williams
Edit Select	Zheng, Roger		2/12/2004	143 Smith
Edit Select	Kapoor, Candace		1/15/2001	57 Adams
Edit Select	Serrano, Stacy		6/1/1999	271 Williams
Edit Select	Stewart, Jasmine		10/12/1997	131 Smith
Edit Select	Xu, Kristen		7/23/2001	203 Williams
Edit Select	Van Houten, Roger		12/7/2000	213 Smith

Change any of these values, including **Office Assignment**. Click **Update** and you'll see the changes reflected in the list.

Displaying Related Entities in a Separate Control

Each instructor can teach one or more courses, so you'll add an `EntityDataSource` control and a `GridView` control to list the courses associated with whichever instructor is selected in the instructors `GridView` control. To create a heading and the `EntityDataSource` control for courses entities, add the following markup between the error message `Label` control and the closing `</div>` tag:

```
<h3>Courses Taught</h3>
        <asp:EntityDataSource ID="CoursesEntityDataSource" runat="server"
            ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
            EntitySetName="Courses"
            Where="@PersonID IN (SELECT VALUE instructor.PersonID FROM it.People AS instructor)">
            <WhereParameters>
                <asp:ControlParameter ControlID="InstructorsGridView" Type="Int32" Name="PersonID"
PropertyName="SelectedValue" />
            </WhereParameters>
        </asp:EntityDataSource>
```

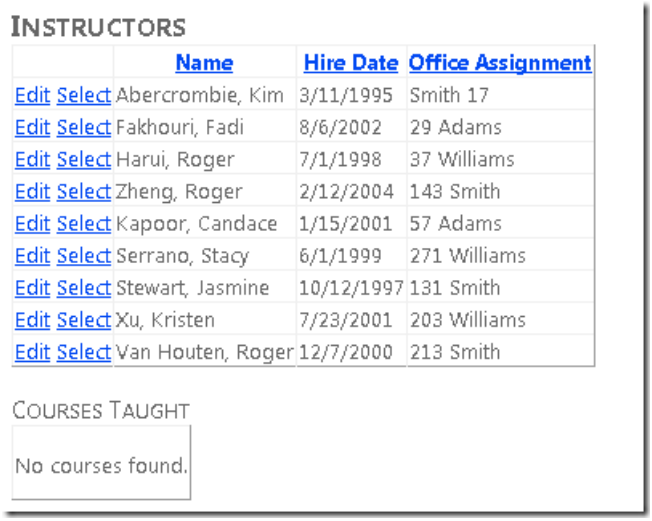
The `Where` parameter contains the value of the `PersonID` of the instructor whose row is selected in the `InstructorsGridView` control. The `Where` property contains a subselect command that gets all associated `Person` entities from a `Course` entity's `People` navigation property and selects the `Course` entity only if one of the associated `Person` entities contains the selected `PersonID` value.

To create the `GridView` control., add the following markup immediately following the `CoursesEntityDataSource` control (before the closing `</div>` tag):

```
<asp:GridView ID="CoursesGridView" runat="server"
            DataSourceID="CoursesEntityDataSource"
            AllowSorting="True" AutoGenerateColumns="False"
            SelectedRowStyle-BackColor="LightGray"
            DataKeyNames="CourseID">
            <EmptyDataTemplate>
                <p>No courses found.</p>
            </EmptyDataTemplate>
            <Columns>
                <asp:CommandField ShowSelectButton="True" />
                <asp:BoundField DataField="CourseID" HeaderText="ID" ReadOnly="True"
SortExpression="CourseID" />
                <asp:BoundField DataField="Title" HeaderText="Title" SortExpression="Title" />
                <asp:TemplateField HeaderText="Department" SortExpression="DepartmentID">
                    <ItemTemplate>
                        <asp:Label ID="GridViewDepartmentLabel" runat="server" Text='<%#
Eval("Department.Name") %>'></asp:Label>
                    </ItemTemplate>
                </asp:TemplateField>
            </Columns>
        </asp:GridView>
```

Because no courses will be displayed if no instructor is selected, an `EmptyDataTemplate` element is included.

Run the page.



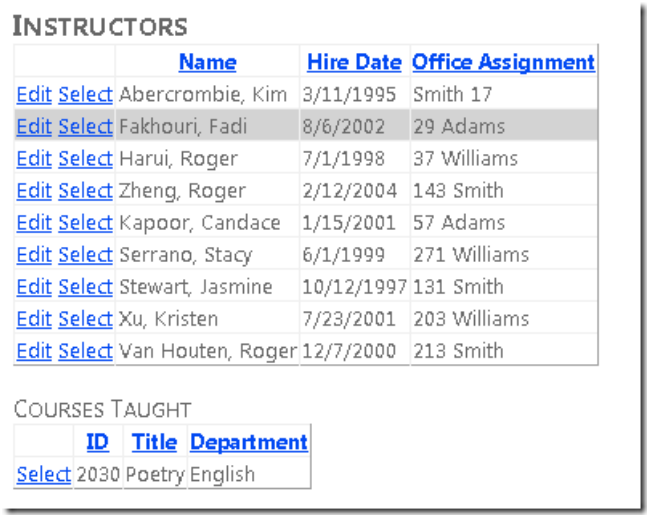
INSTRUCTORS

	Name	Hire Date	Office Assignment
Edit Select	Abercrombie, Kim	3/11/1995	Smith 17
Edit Select	Fakhouri, Fadi	8/6/2002	29 Adams
Edit Select	Harui, Roger	7/1/1998	37 Williams
Edit Select	Zheng, Roger	2/12/2004	143 Smith
Edit Select	Kapoor, Candace	1/15/2001	57 Adams
Edit Select	Serrano, Stacy	6/1/1999	271 Williams
Edit Select	Stewart, Jasmine	10/12/1997	131 Smith
Edit Select	Xu, Kristen	7/23/2001	203 Williams
Edit Select	Van Houten, Roger	12/7/2000	213 Smith

COURSES TAUGHT

No courses found.

Select an instructor who has one or more courses assigned, and the course or courses appear in the list. (Note: although the database schema allows multiple courses, in the test data supplied with the database no instructor actually has more than one course. You can add courses to the database yourself using the **Server Explorer** window or the *CoursesAdd.aspx* page, which you'll add in a later tutorial.)



The `CoursesGridView` control shows only a few course fields. To display all the details for a course, you'll use a `DetailsView` control for the course that the user selects. In *Instructors.aspx*, add the following markup after the closing `</div>` tag (make sure you place this markup **after** the closing div tag, not before it):

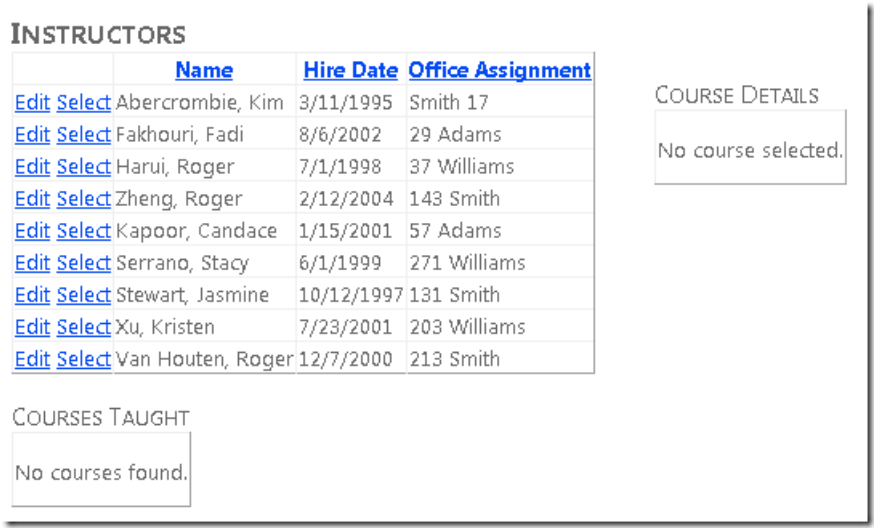
```
<div>
        <h3>Course Details</h3>
        <asp:EntityDataSource ID="CourseDetailsEntityDataSource" runat="server"
            ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
            EntitySetName="Courses"
            AutoGenerateWhereClause="False" Where="it.CourseID = @CourseID"
Include="Department,OnlineCourse,OnsiteCourse,StudentGrades.Person"
            OnSelected="CourseDetailsEntityDataSource_Selected">
            <WhereParameters>
                <asp:ControlParameter ControlID="CoursesGridView" Type="Int32" Name="CourseID"
PropertyName="SelectedValue" />
            </WhereParameters>
        </asp:EntityDataSource>
        <asp:DetailsView ID="CourseDetailsView" runat="server" AutoGenerateRows="False"
            DataSourceID="CourseDetailsEntityDataSource">
            <EmptyDataTemplate>
                <p>
                    No course selected.</p>
            </EmptyDataTemplate>
            <Fields>
                <asp:BoundField DataField="CourseID" HeaderText="ID" ReadOnly="True"
SortExpression="CourseID" />
                <asp:BoundField DataField="Title" HeaderText="Title" SortExpression="Title" />
                <asp:BoundField DataField="Credits" HeaderText="Credits" SortExpression="Credits" />
                <asp:TemplateField HeaderText="Department">
                    <ItemTemplate>
                        <asp:Label ID="DetailsViewDepartmentLabel" runat="server" Text='<%#
Eval("Department.Name") %>'></asp:Label>
                    </ItemTemplate>
                </asp:TemplateField>
                <asp:TemplateField HeaderText="Location">
                    <ItemTemplate>
                        <asp:Label ID="LocationLabel" runat="server" Text='<%# Eval("OnsiteCourse.Location")
%>'></asp:Label>
                    </ItemTemplate>
                </asp:TemplateField>
                <asp:TemplateField HeaderText="URL">
                    <ItemTemplate>
                        <asp:Label ID="URLLabel" runat="server" Text='<%# Eval("OnlineCourse.URL") %>'>
</asp:Label>
                    </ItemTemplate>
                </asp:TemplateField>
            </Fields>
        </asp:DetailsView>
    </div>
```

This markup creates an `EntityDataSource` control that's bound to the `Courses` entity set. The `Where` property selects a course using the `CourseID` value of the selected row in the courses `GridView` control. The markup specifies a handler for the `Selected` event, which you'll use later for displaying student grades, which is another level lower in the hierarchy.

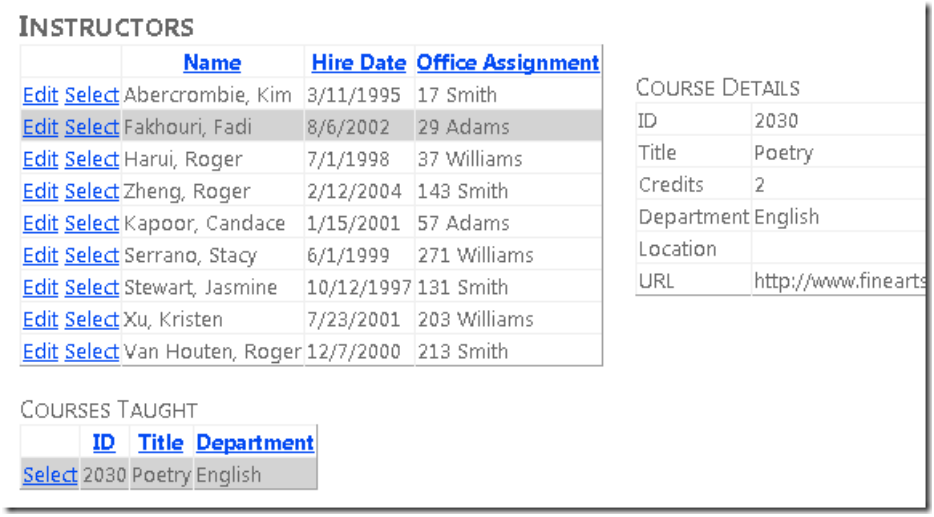
In *Instructors.aspx.cs*, create the following stub for the `CourseDetailsEntityDataSource_Selected` method. (You'll fill this stub out later in the tutorial; for now, you need it so that the page will compile and run.)

```
protected void CourseDetailsEntityDataSource_Selected(object sender, EntityDataSourceSelectedEventArgs e)
{
}
```

Run the page.



Initially there are no course details because no course is selected. Select an instructor who has a course assigned, and then select a course to see the details.



Using the EntityDataSource "Selected" Event to Display Related Data

Finally, you want to show all of the enrolled students and their grades for the selected course. To do this, you'll use the `Selected` event of the `EntityDataSource` control bound to the course `DetailsView`.

In *Instructors.aspx*, add the following markup after the `DetailsView` control:

```
<h3>Student Grades</h3>
        <asp:ListView ID="GradesListView" runat="server">
            <EmptyDataTemplate>
                <p>No student grades found.</p>
            </EmptyDataTemplate>
            <LayoutTemplate>
                <table border="1" runat="server" id="itemPlaceholderContainer">
                    <tr runat="server">
                        <th runat="server">
                            Name
                        </th>
                        <th runat="server">
                            Grade
                        </th>
                    </tr>
                    <tr id="itemPlaceholder" runat="server">
                    </tr>
                </table>
            </LayoutTemplate>
            <ItemTemplate>
                <tr>
                    <td>
                        <asp:Label ID="StudentLastNameLabel" runat="server" Text='<%# Eval("Person.LastName")
%>' />,
                        <asp:Label ID="StudentFirstNameLabel" runat="server" Text='<%#
Eval("Person.FirstMidName") %>' />
                    </td>
                    <td>
                        <asp:Label ID="StudentGradeLabel" runat="server" Text='<%# Eval("Grade") %>' />
                    </td>
                </tr>
            </ItemTemplate>
        </asp:ListView>
```

This markup creates a `ListView` control that displays a list of students and their grades for the selected course. No data source is specified because you'll databind the control in code. The `EmptyDataTemplate` element provides a message to display when no course is selected—in that case, there are no students to display. The `LayoutTemplate` element creates an HTML table to display the list, and the `ItemTemplate` specifies the columns to display. The student ID and the student grade are from the `StudentGrade` entity, and the student name is from the `Person` entity that the Entity Framework makes available in the `Person` navigation property of the `StudentGrade` entity.

In *Instructors.aspx.cs*, replace the stubbed-out `CourseDetailsEntityDataSource_Selected` method with the following code:

```
protected void CourseDetailsEntityDataSource_Selected(object sender, EntityDataSourceSelectedEventArgs e)
{
    var course = e.Results.Cast<Course>().FirstOrDefault();
    if (course != null)
    {
        var studentGrades = course.StudentGrades.ToList();
        GradesListView.DataSource = studentGrades;
        GradesListView.DataBind();
    }
}
```

The event argument for this event provides the selected data in the form of a collection, which will have zero items if nothing is selected or one item if a `Course` entity is selected. If a `Course` entity is selected, the code uses the `First` method to convert the collection to a single object. It then gets `StudentGrade` entities from the navigation property, converts them to a collection, and binds the `GradesListView` control to the collection.

This is sufficient to display grades, but you want to make sure that the message in the empty data template is

displayed the first time the page is displayed and whenever a course is not selected. To do that, create the following method, which you'll call from two places:

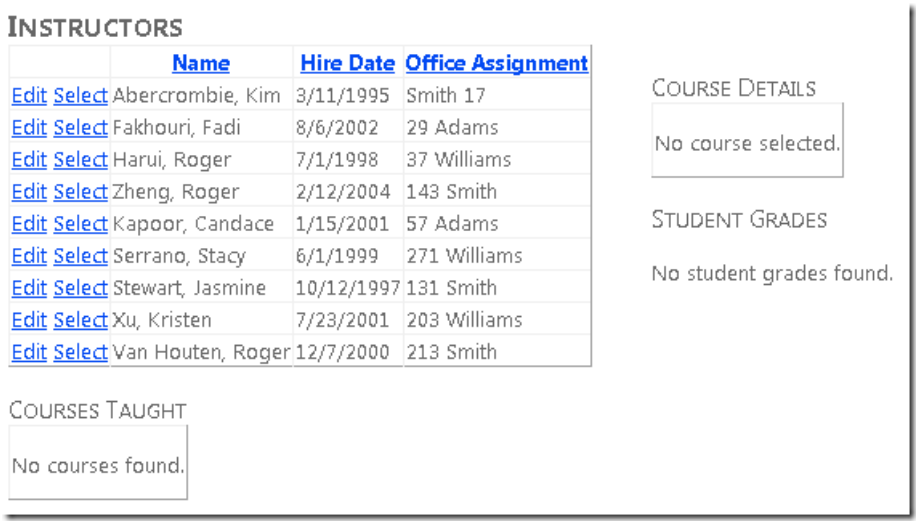
```
private void ClearStudentGradesDataSource()
{
    var emptyStudentGradesList = new List<StudentGrade>();
    GradesListView.DataSource = emptyStudentGradesList;
    GradesListView.DataBind();
}
```

Call this new method from the `Page_Load` method to display the empty data template the first time the page is displayed. And call it from the `InstructorsGridView_SelectedIndexChanged` method because that event is raised when an instructor is selected, which means new courses are loaded into the courses `GridView` control and none is selected yet. Here are the two calls:

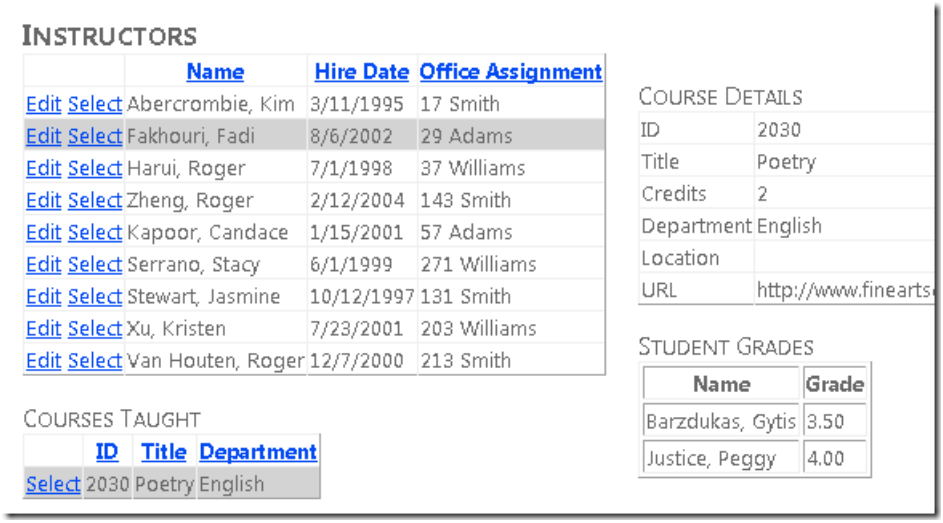
```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        ClearStudentGradesDataSource();
    }
}
```

```
protected void InstructorsGridView_SelectedIndexChanged(object sender, EventArgs e)
{
    ClearStudentGradesDataSource();
}
```

Run the page.



Select an instructor that has a course assigned, and then select the course.



You have now seen a few ways to work with related data. In the following tutorial, you'll learn how to add relationships between existing entities, how to remove relationships, and how to add a new entity that has a relationship to an existing entity.

PREVIOUS NEXT

Getting Started with Entity Framework 4.0 Database First and ASP.NET 4 Web Forms - Part 5

2/20/2020 • 8 minutes to read • Edit Online

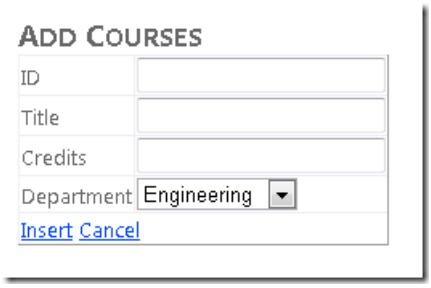
by Tom Dykstra

> The Contoso University sample web application demonstrates how to create ASP.NET Web Forms applications using the Entity Framework 4.0 and Visual Studio 2010. For information about the tutorial series, see the first tutorial in the series

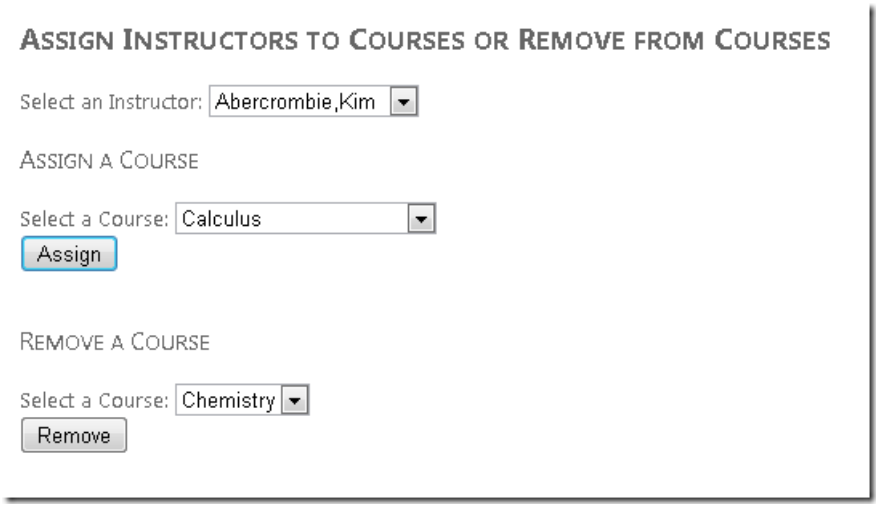
Working with Related Data, Continued

In the previous tutorial you began to use the `EntityDataSource` control to work with related data. You displayed multiple levels of hierarchy and edited data in navigation properties. In this tutorial you'll continue to work with related data by adding and deleting relationships and by adding a new entity that has a relationship to an existing entity.

You'll create a page that adds courses that are assigned to departments. The departments already exist, and when you create a new course, at the same time you'll establish a relationship between it and an existing department.



You'll also create a page that works with a many-to-many relationship by assigning an instructor to a course (adding a relationship between two entities that you select) or removing an instructor from a course (removing a relationship between two entities that you select). In the database, adding a relationship between an instructor and a course results in a new row being added to the `CourseInstructor` association table; removing a relationship involves deleting a row from the `CourseInstructor` association table. However, you do this in the Entity Framework by setting navigation properties, without referring to the `CourseInstructor` table explicitly.



Adding an Entity with a Relationship to an Existing Entity

Create a new web page named *CoursesAdd.aspx* that uses the *Site.Master* master page, and add the following markup to the `Content` control named `Content2`:

```
<h2>Add Courses</h2>
    <asp:EntityDataSource ID="CoursesEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
        EntitySetName="Courses"
        EnableInsert="True" EnableDelete="True" >
    </asp:EntityDataSource>
    <asp:DetailsView ID="CoursesDetailsView" runat="server" AutoGenerateRows="False"
        DataSourceID="CoursesEntityDataSource" DataKeyNames="CourseID"
        DefaultMode="Insert" oniteminserting="CoursesDetailsView_ItemInserting">
        <Fields>
            <asp:BoundField DataField="CourseID" HeaderText="ID" />
            <asp:BoundField DataField="Title" HeaderText="Title" />
            <asp:BoundField DataField="Credits" HeaderText="Credits" />
            <asp:TemplateField HeaderText="Department">
                <InsertItemTemplate>
                    <asp:EntityDataSource ID="DepartmentsEntityDataSource" runat="server"
ConnectionString="name=SchoolEntities"
                        DefaultContainerName="SchoolEntities" EnableDelete="True" EnableFlattening="False"
                        EntitySetName="Departments" EntityTypeFilter="Department">
                    </asp:EntityDataSource>
                    <asp:DropDownList ID="DepartmentsDropDownList" runat="server"
DataSourceID="DepartmentsEntityDataSource"
                        DataTextField="Name" DataValueField="DepartmentID"
                        oninit="DepartmentsDropDownList_Init">
                    </asp:DropDownList>
                </InsertItemTemplate>
            </asp:TemplateField>
            <asp:CommandField ShowInsertButton="True" />
        </Fields>
    </asp:DetailsView>
```

This markup creates an `EntityDataSource` control that selects courses, that enables inserting, and that specifies a handler for the `Inserting` event. You'll use the handler to update the `Department` navigation property when a new `Course` entity is created.

The markup also creates a `DetailsView` control to use for adding new `Course` entities. The markup uses bound fields for `Course` entity properties. You have to enter the `CourseID` value because this is not a system-generated ID field. Instead, it's a course number that must be specified manually when the course is created.

You use a template field for the `Department` navigation property because navigation properties cannot be used with `BoundField` controls. The template field provides a drop-down list to select the department. The drop-down list is bound to the `Departments` entity set by using `Eval` rather than `Bind`, again because you cannot directly bind navigation properties in order to update them. You specify a handler for the `DropDownList` control's `Init` event so that you can store a reference to the control for use by the code that updates the `DepartmentID` foreign key.

In *CoursesAdd.aspx.cs* just after the partial-class declaration, add a class field to hold a reference to the `DepartmentsDropDownList` control:

```
private DropDownList departmentDropDownList;
```

Add a handler for the `DepartmentsDropDownList` control's `Init` event so that you can store a reference to the control. This lets you get the value the user has entered and use it to update the `DepartmentID` value of the `Course` entity.

```
protected void DepartmentsDropDownList_Init(object sender, EventArgs e)
{
    departmentDropDownList = sender as DropDownList;
}
```

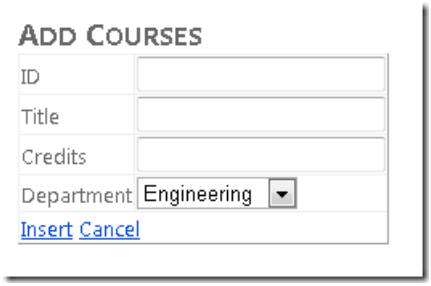
Add a handler for the `DetailsView` control's `Inserting` event:

```
protected void CoursesDetailsView_ItemInserting(object sender, DetailsViewInsertEventArgs e)
{
    var departmentID = Convert.ToInt32(departmentDropDownList.SelectedValue);
    e.Values["DepartmentID"] = departmentID;
}
```

When the user clicks `Insert`, the `Inserting` event is raised before the new record is inserted. The code in the handler gets the `DepartmentID` from the `DropDownList` control and uses it to set the value that will be used for the `DepartmentID` property of the `Course` entity.

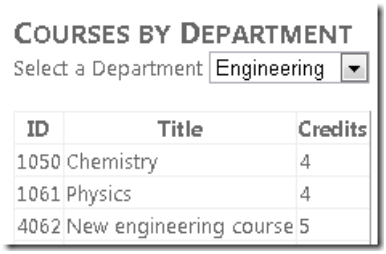
The Entity Framework will take care of adding this course to the `Courses` navigation property of the associated `Department` entity. It also adds the department to the `Department` navigation property of the `Course` entity.

Run the page.



Enter an ID, a title, a number of credits, and select a department, then click **Insert**.

Run the *Courses.aspx* page, and select the same department to see the new course.



Working with Many-to-Many Relationships

The relationship between the `Courses` entity set and the `People` entity set is a many-to-many relationship. A `Course` entity has a navigation property named `People` that can contain zero, one, or more related `Person` entities (representing instructors assigned to teach that course). And a `Person` entity has a navigation property named `Courses` that can contain zero, one, or more related `Course` entities (representing courses that instructor is assigned to teach). One instructor might teach multiple courses, and one course might be taught by multiple instructors. In this section of the walkthrough, you'll add and remove relationships between `Person` and `Course` entities by updating the navigation properties of the related entities.

Create a new web page named *InstructorsCourses.aspx* that uses the *Site.Master* master page, and add the following markup to the `Content` control named `Content2`:

```
<h2>Assign Instructors to Courses or Remove from Courses</h2>
    <br />
    <asp:EntityDataSource ID="InstructorsEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
        EntitySetName="People"
        Where="it.HireDate is not null" Select="it.LastName + ', ' + it.FirstMidName AS Name, it.PersonID">
    </asp:EntityDataSource>
    Select an Instructor:
    <asp:DropDownList ID="InstructorsDropDownList" runat="server" DataSourceID="InstructorsEntityDataSource"
        AutoPostBack="true" DataTextField="Name" DataValueField="PersonID"
        OnSelectedIndexChanged="InstructorsDropDownList_SelectedIndexChanged"
        OnDataBound="InstructorsDropDownList_DataBound">
    </asp:DropDownList>
    <h3>
        Assign a Course</h3>
    <br />
    Select a Course:
    <asp:DropDownList ID="UnassignedCoursesDropDownList" runat="server"
        DataTextField="Title" DataValueField="CourseID">
    </asp:DropDownList>
    <br />
    <asp:Button ID="AssignCourseButton" runat="server" Text="Assign" OnClick="AssignCourseButton_Click" />
    <br />
    <asp:Label ID="CourseAssignedLabel" runat="server" Visible="false" Text="Assignment successful">
</asp:Label>
    <br />
    <h3>
        Remove a Course</h3>
    <br />
    Select a Course:
    <asp:DropDownList ID="AssignedCoursesDropDownList" runat="server"
        DataTextField="title" DataValueField="courseiD">
    </asp:DropDownList>
    <br />
    <asp:Button ID="RemoveCourseButton" runat="server" Text="Remove" OnClick="RemoveCourseButton_Click" />
    <br />
    <asp:Label ID="CourseRemovedLabel" runat="server" Visible="false" Text="Removal successful"></asp:Label>
```

This markup creates an `EntityDataSource` control that retrieves the name and `PersonID` of `Person` entities for instructors. A `DropDrownList` control is bound to the `EntityDataSource` control. The `DropDownList` control specifies a handler for the `DataBound` event. You'll use this handler to databind the two drop-down lists that display courses.

The markup also creates the following group of controls to use for assigning a course to the selected instructor:

- A `DropDownList` control for selecting a course to assign. This control will be populated with courses that are currently not assigned to the selected instructor.
- A `Button` control to initiate the assignment.
- A `Label` control to display an error message if the assignment fails.

Finally, the markup also creates a group of controls to use for removing a course from the selected instructor.

In *InstructorsCourses.aspx.cs*, add a using statement:

```
using ContosoUniversity.DAL;
```

Add a method for populating the two drop-down lists that display courses:

```
private void PopulateDropDownLists()
{
    using (var context = new SchoolEntities())
    {
        var allCourses = (from c in context.Courses
                          select c).ToList();

        var instructorID = Convert.ToInt32(InstructorsDropDownList.SelectedValue);
        var instructor = (from p in context.People.Include("Courses")
                          where p.PersonID == instructorID
                          select p).First();

        var assignedCourses = instructor.Courses.ToList();
        var unassignedCourses = allCourses.Except(assignedCourses.AsEnumerable()).ToList();

        UnassignedCoursesDropDownList.DataSource = unassignedCourses;
        UnassignedCoursesDropDownList.DataBind();
        UnassignedCoursesDropDownList.Visible = true;

        AssignedCoursesDropDownList.DataSource = assignedCourses;
        AssignedCoursesDropDownList.DataBind();
        AssignedCoursesDropDownList.Visible = true;
    }
}
```

This code gets all courses from the `Courses` entity set and gets the courses from the `Courses` navigation property of the `Person` entity for the selected instructor. It then determines which courses are assigned to that instructor and populates the drop-down lists accordingly.

Add a handler for the `Assign` button's `Click` event:

```
protected void AssignCourseButton_Click(object sender, EventArgs e)
{
    using (var context = new SchoolEntities())
    {
        var instructorID = Convert.ToInt32(InstructorsDropDownList.SelectedValue);
        var instructor = (from p in context.People
                          where p.PersonID == instructorID
                          select p).First();
        var courseID = Convert.ToInt32(UnassignedCoursesDropDownList.SelectedValue);
        var course = (from c in context.Courses
                      where c.CourseID == courseID
                      select c).First();
        instructor.Courses.Add(course);
        try
        {
            context.SaveChanges();
            PopulateDropDownLists();
            CourseAssignedLabel.Text = "Assignment successful.";
        }
        catch (Exception)
        {
            CourseAssignedLabel.Text = "Assignment unsuccessful.";
            //Add code to log the error.
        }
        CourseAssignedLabel.Visible = true;
    }
}
```

This code gets the `Person` entity for the selected instructor, gets the `Course` entity for the selected course, and adds the selected course to the `Courses` navigation property of the instructor's `Person` entity. It then saves the changes to the database and repopulates the drop-down lists so the results can be seen immediately.

Add a handler for the `Remove` button's `Click` event:

```
protected void RemoveCourseButton_Click(object sender, EventArgs e)
{
    using (var context = new SchoolEntities())
    {
        var instructorID = Convert.ToInt32(InstructorsDropDownList.SelectedValue);
        var instructor = (from p in context.People
                          where p.PersonID == instructorID
                          select p).First();
        var courseID = Convert.ToInt32(AssignedCoursesDropDownList.SelectedValue);
        var courses = instructor.Courses;
        var courseToRemove = new Course();
        foreach (Course c in courses)
        {
            if (c.CourseID == courseID)
            {
                courseToRemove = c;
                break;
            }
        }
        try
        {
            courses.Remove(courseToRemove);
            context.SaveChanges();
            PopulateDropDownLists();
            CourseRemovedLabel.Text = "Removal successful.";
        }
        catch (Exception)
        {
            CourseRemovedLabel.Text = "Removal unsuccessful.";
            //Add code to log the error.
        }
        CourseRemovedLabel.Visible = true;
    }
}
```

This code gets the `Person` entity for the selected instructor, gets the `Course` entity for the selected course, and removes the selected course from the `Person` entity's `Courses` navigation property. It then saves the changes to the database and repopulates the drop-down lists so the results can be seen immediately.

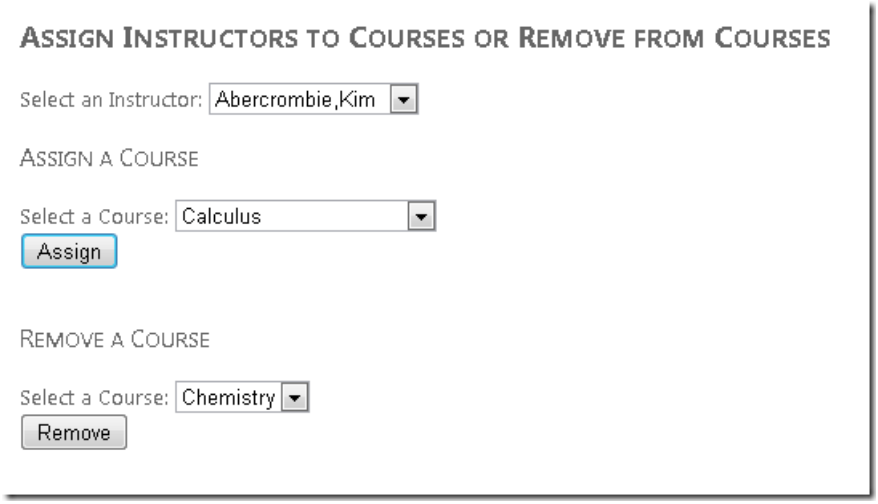
Add code to the `Page_Load` method that makes sure the error messages are not visible when there's no error to report, and add handlers for the `DataBound` and `SelectedIndexChanged` events of the instructors drop-down list to populate the courses drop-down lists:

```
protected void Page_Load(object sender, EventArgs e)
{
    CourseAssignedLabel.Visible = false;
    CourseRemovedLabel.Visible = false;
}

protected void InstructorsDropDownList_DataBound(object sender, EventArgs e)
{
    PopulateDropDownLists();
}

protected void InstructorsDropDownList_SelectedIndexChanged(object sender, EventArgs e)
{
    PopulateDropDownLists();
}
```

Run the page.



Select an instructor. The **Assign a Course** drop-down list displays the courses that the instructor doesn't teach, and the **Remove a Course** drop-down list displays the courses that the instructor is already assigned to. In the **Assign a Course** section, select a course and then click **Assign**. The course moves to the **Remove a Course** drop-down list. Select a course in the **Remove a Course** section and click **Remove**. The course moves to the **Assign a Course** drop-down list.

You have now seen some more ways to work with related data. In the following tutorial, you'll learn how to use inheritance in the data model to improve the maintainability of your application.

PREVIOUS NEXT

Getting Started with Entity Framework 4.0 Database First and ASP.NET 4 Web Forms - Part 6

2/20/2020 • 7 minutes to read • Edit Online

by Tom Dykstra

> The Contoso University sample web application demonstrates how to create ASP.NET Web Forms applications using the Entity Framework 4.0 and Visual Studio 2010. For information about the tutorial series, see the first tutorial in the series

Implementing Table-per-Hierarchy Inheritance

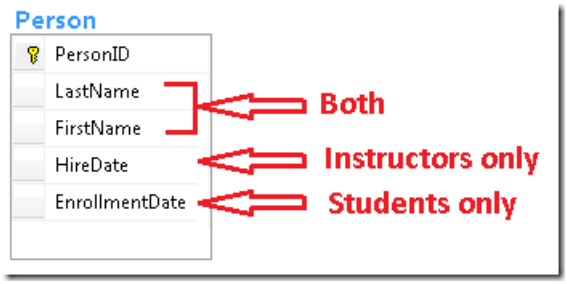
In the previous tutorial you worked with related data by adding and deleting relationships and by adding a new entity that had a relationship to an existing entity. This tutorial will show you how to implement inheritance in the data model.

In object-oriented programming, you can use inheritance to make it easier to work with related classes. For example, you could create `Instructor` and `Student` classes that derive from a `Person` base class. You can create the same kinds of inheritance structures among entities in the Entity Framework.

In this part of the tutorial, you won't create any new web pages. Instead, you'll add derived entities to the data model and modify existing pages to use the new entities.

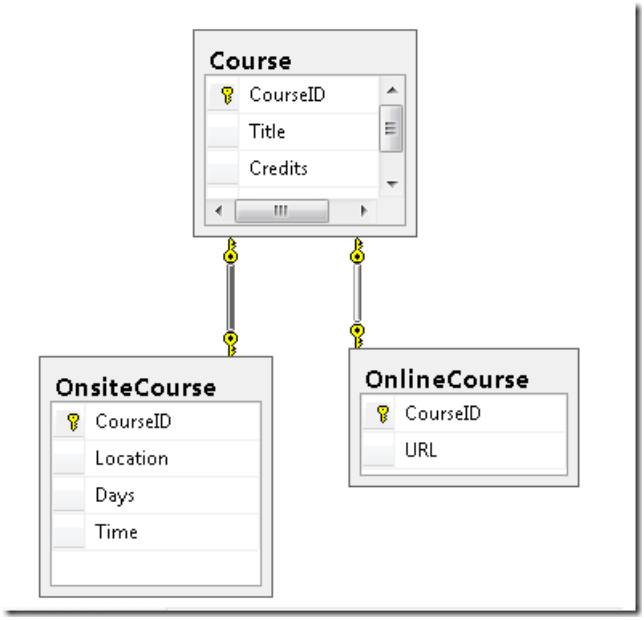
Table-per-Hierarchy versus Table-per-Type Inheritance

A database can store information about related objects in one table or in multiple tables. For example, in the `School` database, the `Person` table includes information about both students and instructors in a single table. Some of the columns apply only to instructors (`HireDate`), some only to students (`EnrollmentDate`), and some to both (`LastName`, `FirstName`).



You can configure the Entity Framework to create `Instructor` and `Student` entities that inherit from the `Person` entity. This pattern of generating an entity inheritance structure from a single database table is called *table-per-hierarchy* (TPH) inheritance.

For courses, the `School` database uses a different pattern. Online courses and onsite courses are stored in separate tables, each of which has a foreign key that points to the `Course` table. Information common to both course types is stored only in the `Course` table.



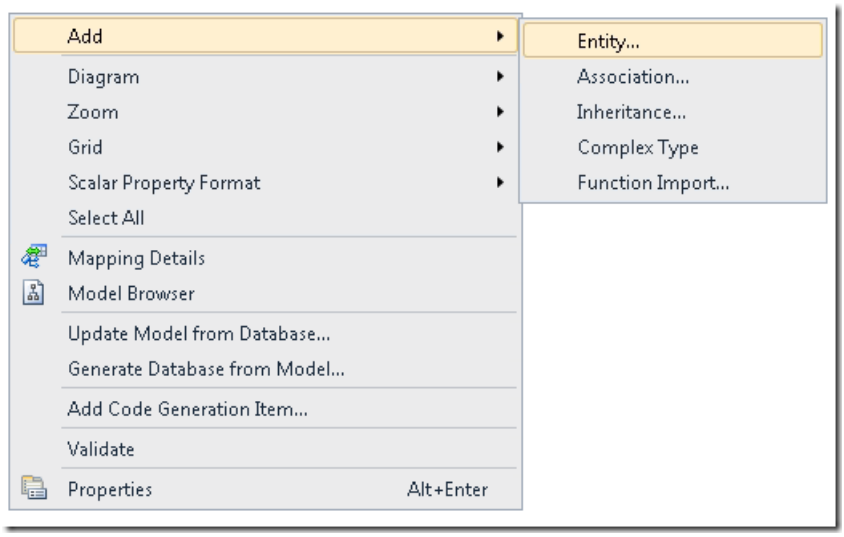
You can configure the Entity Framework data model so that `OnlineCourse` and `OnsiteCourse` entities inherit from the `Course` entity. This pattern of generating an entity inheritance structure from separate tables for each type, with each separate table referring back to a table that stores data common to all types, is called *table per type* (TPT) inheritance.

TPH inheritance patterns generally deliver better performance in the Entity Framework than TPT inheritance patterns, because TPT patterns can result in complex join queries. This walkthrough demonstrates how to implement TPH inheritance. You'll do that by performing the following steps:

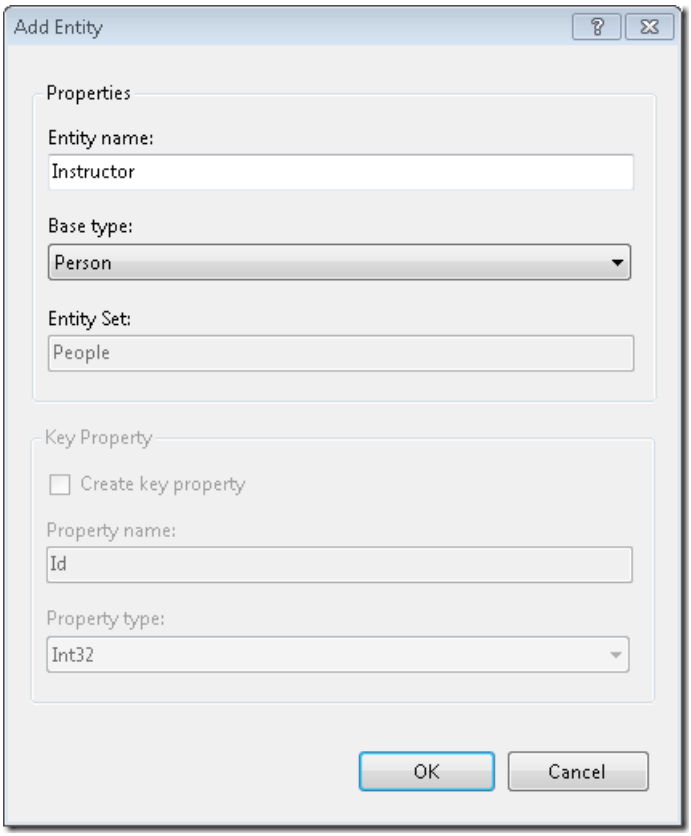
- Create `Instructor` and `Student` entity types that derive from `Person`.
- Move properties that pertain to the derived entities from the `Person` entity to the derived entities.
- Set constraints on properties in the derived types.
- Make the `Person` entity an abstract entity.
- Map each derived entity to the `Person` table with a condition that specifies how to determine whether a `Person` row represents that derived type.

Adding Instructor and Student Entities

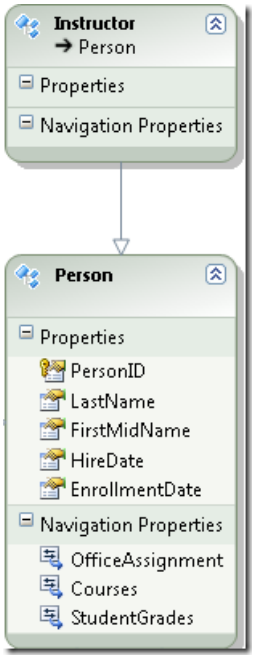
Open the *SchoolModel.edmx* file, right-click an unoccupied area in the designer, select **Add**, then select **Entity**.



In the **Add Entity** dialog box, name the entity `Instructor` and set its **Base type** option to `Person`.

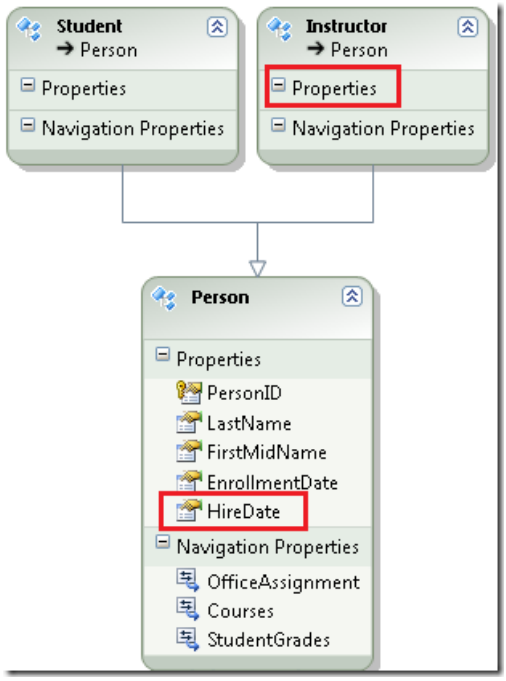


Click **OK**. The designer creates an `Instructor` entity that derives from the `Person` entity. The new entity does not yet have any properties.

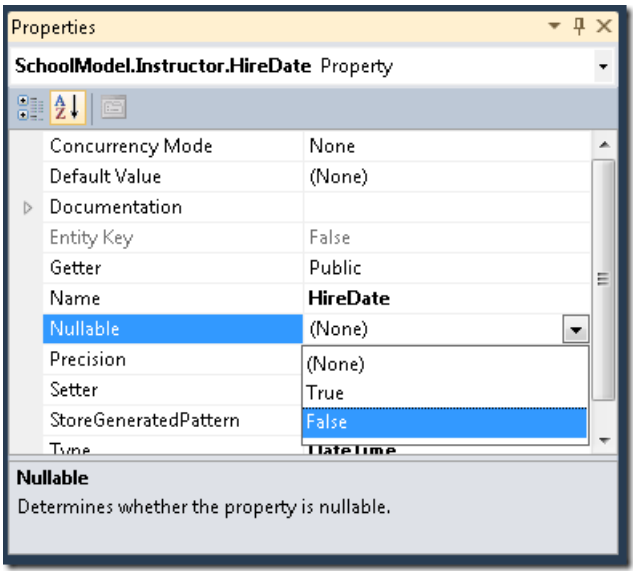


Repeat the procedure to create a `Student` entity that also derives from `Person`.

Only instructors have hire dates, so you need to move that property from the `Person` entity to the `Instructor` entity. In the `Person` entity, right-click the `HireDate` property and click **Cut**. Then right-click **Properties** in the `Instructor` entity and click **Paste**.

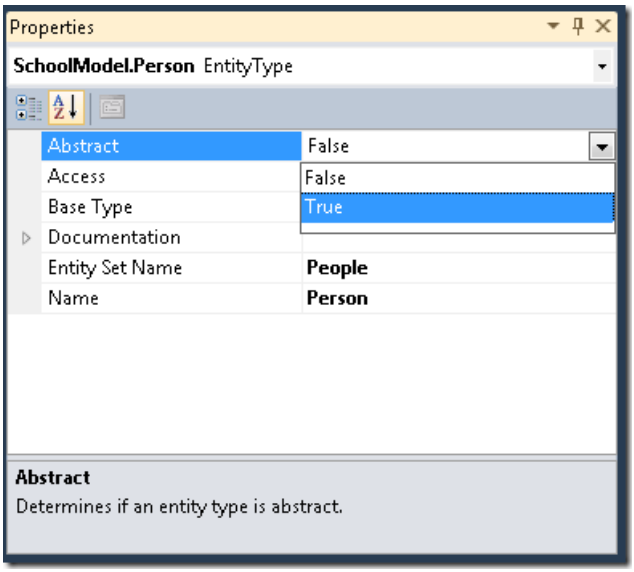


The hire date of an `Instructor` entity cannot be null. Right-click the `HireDate` property, click **Properties**, and then in the **Properties** window change `Nullable` to `False`.



Repeat the procedure to move the `EnrollmentDate` property from the `Person` entity to the `Student` entity. Make sure that you also set `Nullable` to `False` for the `EnrollmentDate` property.

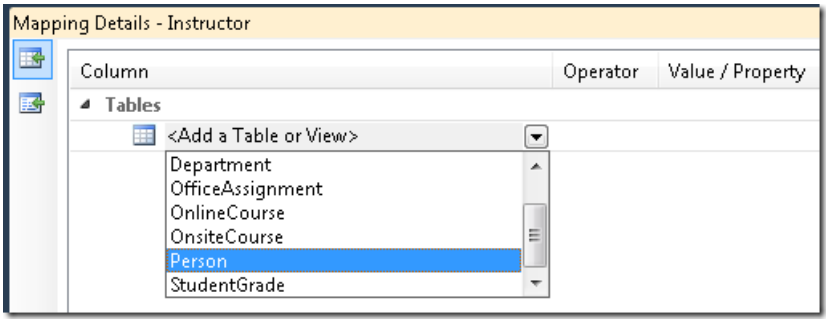
Now that a `Person` entity has only the properties that are common to `Instructor` and `Student` entities (aside from navigation properties, which you're not moving), the entity can only be used as a base entity in the inheritance structure. Therefore, you need to ensure that it's never treated as an independent entity. Right-click the `Person` entity, select **Properties**, and then in the **Properties** window change the value of the **Abstract** property to **True**.



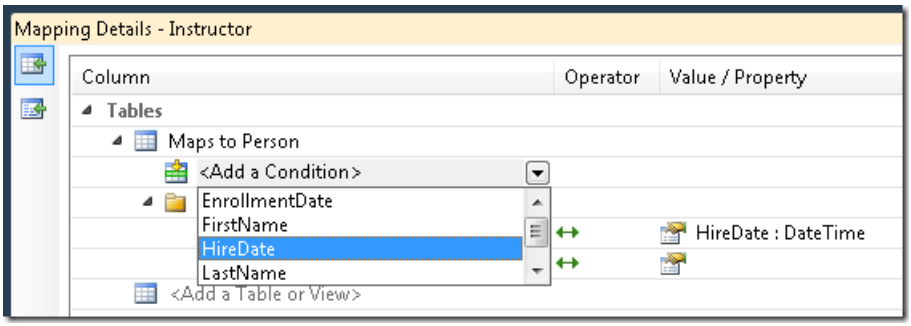
Mapping Instructor and Student Entities to the Person Table

Now you need to tell the Entity Framework how to differentiate between `Instructor` and `Student` entities in the database.

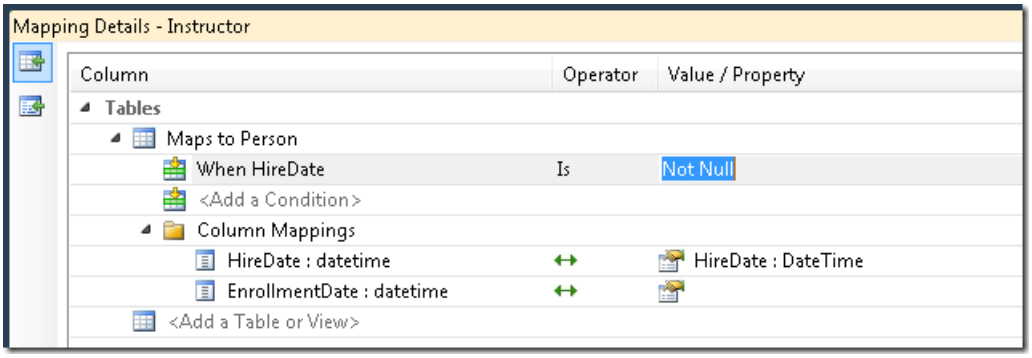
Right-click the `Instructor` entity and select **Table Mapping**. In the **Mapping Details** window, click **Add a Table or View** and select **Person**.



Click **Add a Condition**, and then select **HireDate**.



Change **Operator** to **Is** and **Value / Property** to **Not Null**.



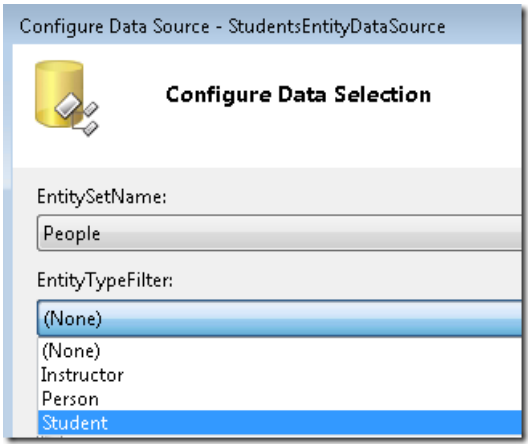
Repeat the procedure for the `Students` entity, specifying that this entity maps to the `Person` table when the `EnrollmentDate` column is not null. Then save and close the data model.

Build the project in order to create the new entities as classes and make them available in the designer.

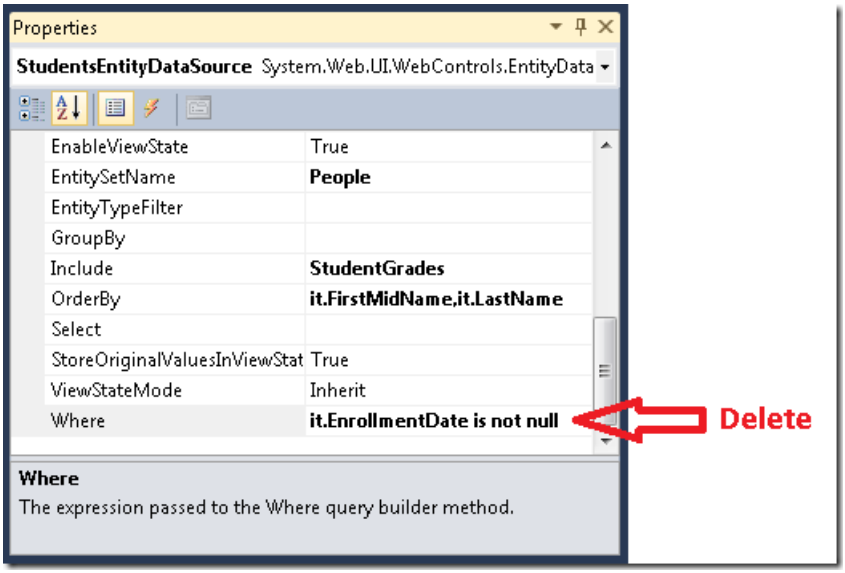
Using the Instructor and Student Entities

When you created the web pages that work with student and instructor data, you databound them to the `Person` entity set, and you filtered on the `HireDate` or `EnrollmentDate` property to restrict the returned data to students or instructors. However, now when you bind each data source control to the `Person` entity set, you can specify that only `Student` or `Instructor` entity types should be selected. Because the Entity Framework knows how to differentiate students and instructors in the `Person` entity set, you can remove the `Where` property settings you entered manually to do that.

In the Visual Studio Designer, you can specify the entity type that an `EntityDataSource` control should select in the **EntityTypeFilter** drop-down box of the `Configure Data Source` wizard, as shown in the following example.



And in the **Properties** window you can remove `Where` clause values that are no longer needed, as shown in the following example.



However, because you've changed the markup for `EntityDataSource` controls to use the `ContextTypeName` attribute, you cannot run the **Configure Data Source** wizard on `EntityDataSource` controls that you've already created. Therefore, you'll make the required changes by changing markup instead.

Open the *Students.aspx* page. In the `StudentsEntityDataSource` control, remove the `Where` attribute and add an `EntityTypeFilter="Student"` attribute. The markup will now resemble the following example:

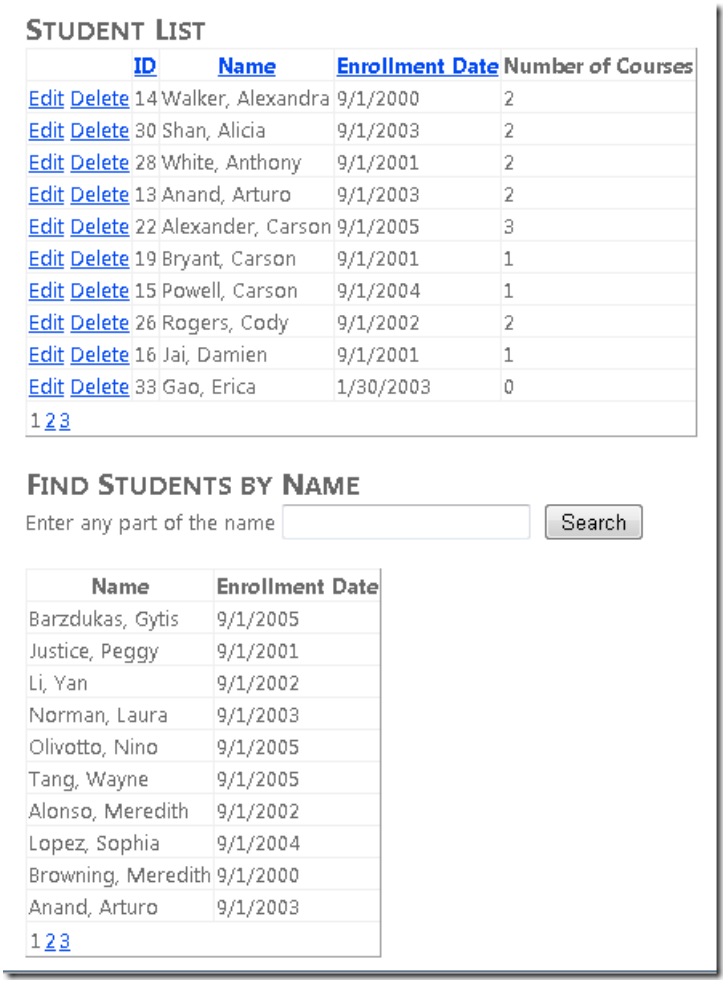
```
<asp:EntityDataSource ID="StudentsEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
        EntitySetName="People" EntityTypeFilter="Student"
        Include="StudentGrades"
        EnableDelete="True" EnableUpdate="True"
        OrderBy="it.LastName" >
    </asp:EntityDataSource>
```

Setting the `EntityTypeFilter` attribute ensures that the `EntityDataSource` control will select only the specified entity type. If you wanted to retrieve both `Student` and `Instructor` entity types, you would not set this attribute. (You have the option of retrieving multiple entity types with one `EntityDataSource` control only if you're using the control for read-only data access. If you're using an `EntityDataSource` control to insert, update, or delete entities, and if the entity set it's bound to can contain multiple types, you can only work with one entity type, and you have to set this attribute.)

Repeat the procedure for the `SearchEntityDataSource` control, except remove only the part of the `Where` attribute that selects `Student` entities instead of removing the property altogether. The opening tag of the control will now resemble the following example:

```
<asp:EntityDataSource ID="SearchEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
        EntitySetName="People" EntityTypeFilter="Student"
        Where="it.FirstMidName Like '%' + @StudentName + '%' or it.LastName Like '%' + @StudentName + '%'" >
```

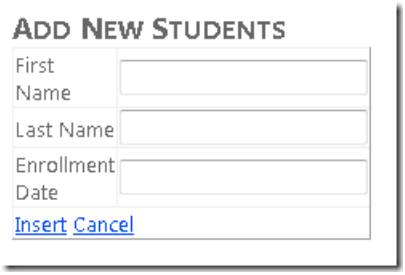
Run the page to verify that it still works as it did before.



Update the following pages that you created in earlier tutorials so that they use the new `Student` and `Instructor` entities instead of `Person` entities, then run them to verify that they work as they did before:

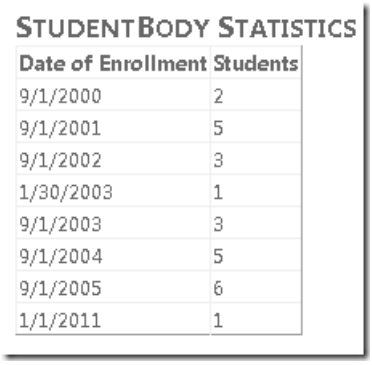
- In *StudentsAdd.aspx*, add `EntityTypeFilter="Student"` to the `StudentsEntityDataSource` control. The markup will now resemble the following example:

```
<asp:EntityDataSource ID="StudentsEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
        EntitySetName="People" EntityTypeFilter="Student"
        EnableInsert="True"
    </asp:EntityDataSource>
```



- In *About.aspx*, add `EntityTypeFilter="Student"` to the `StudentStatisticsEntityDataSource` control and remove `Where="it.EnrollmentDate is not null"`. The markup will now resemble the following example:

```
<asp:EntityDataSource ID="StudentStatisticsEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
        EntitySetName="People" EntityTypeFilter="Student"
        Select="it.EnrollmentDate, Count(it.EnrollmentDate) AS NumberOfStudents"
        OrderBy="it.EnrollmentDate" GroupBy="it.EnrollmentDate" >
    </asp:EntityDataSource>
```



- In *Instructors.aspx* and *InstructorsCourses.aspx*, add `EntityTypeFilter="Instructor"` to the `InstructorsEntityDataSource` control and remove `Where="it.HireDate is not null"`. The markup in *Instructors.aspx* now resembles the following example:

```
<asp:EntityDataSource ID="InstructorsEntityDataSource" runat="server"
            ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="false"
            EntitySetName="People" EntityTypeFilter="Instructor"
            Include="OfficeAssignment"
            EnableUpdate="True">
        </asp:EntityDataSource>
```

INSTRUCTORS

	ID	Name	Hire Date	Office Assignment
Edit Select	1	Abercrombie, Kim	3/11/1995	17 Smith
Edit Select	4	Fakhouri, Fadi	8/6/2002	29 Adams
Edit Select	5	Harui, Roger	7/1/1998	37 Williams
Edit Select	18	Zheng, Roger	2/12/2004	143 Smith
Edit Select	25	Kapoor, Candace	1/15/2001	57 Adams
Edit Select	27	Serrano, Stacy	6/1/1999	271 Williams
Edit Select	31	Stewart, Jasmine	10/12/1997	131 Smith
Edit Select	32	Xu, Kristen	7/23/2001	203 Williams
Edit Select	34	Van Houten, Roger	12/7/2000	213 Smith

COURSES TAUGHT

	ID	Title	Department
Select	2042	Literature	English
Select	3141	Trigonometry	Mathematics
Select	4022	Microeconomics	Economics
Select	4041	Macroeconomics	Economics
Select	4061	Quantitative	Economics
Select	4062	New engineering course	Engineering
Select	4063	new course	Economics

COURSE DETAILS

ID	2042
Title	Literature
Credits	4
Department	English
Location	225 Adams
URL	

STUDENT GRADES

ID	Name	Grade
6	Li, Yan	3.50
7	Norman, Laura	4.00
8	Olivotto, Nino	3.00

The markup in *InstructorsCourses.aspx* will now resemble the following example:

```
<asp:EntityDataSource ID="InstructorsEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="False"
        EntitySetName="People" EntityTypeFilter="Instructor"
        Select="it.LastName + ',' + it.FirstMidName AS Name, it.PersonID">
    </asp:EntityDataSource>
```

ASSIGN INSTRUCTORS TO COURSES OR REMOVE FROM COURSES

Select an Instructor: Abercrombie, Kim

ASSIGN A COURSE

Select a Course: Calculus

Assign

REMOVE A COURSE

Select a Course: Literature

Remove

As a result of these changes, you've improved the Contoso University application's maintainability in several ways. You've moved selection and validation logic out of the UI layer (*.aspx* markup) and made it an integral part of the data access layer. This helps to isolate your application code from changes that you might make in the future to the database schema or the data model. For example, you could decide that students might be hired as teachers' aids and therefore would get a hire date. You could then add a new property to differentiate students from instructors and update the data model. No code in the web application would need to change except where you wanted to show a hire date for students. Another benefit of adding `Instructor` and `Student` entities is that your code is more readily understandable than when it referred to `Person` objects that were actually students or instructors.

You've now seen one way to implement an inheritance pattern in the Entity Framework. In the following tutorial, you'll learn how to use stored procedures in order to have more control over how the Entity Framework accesses the database.

Getting Started with Entity Framework 4.0 Database First and ASP.NET 4 Web Forms - Part 7

2/20/2020 • 7 minutes to read • Edit Online

by Tom Dykstra

> The Contoso University sample web application demonstrates how to create ASP.NET Web Forms applications using the Entity Framework 4.0 and Visual Studio 2010. For information about the tutorial series, see the first tutorial in the series

Using Stored Procedures

In the previous tutorial you implemented a table-per-hierarchy inheritance pattern. This tutorial will show you how to use stored procedures to gain more control over database access.

The Entity Framework lets you specify that it should use stored procedures for database access. For any entity type, you can specify a stored procedure to use for creating, updating, or deleting entities of that type. Then in the data model you can add references to stored procedures that you can use to perform tasks such as retrieving sets of entities.

Using stored procedures is a common requirement for database access. In some cases a database administrator may require that all database access go through stored procedures for security reasons. In other cases you may want to build business logic into some of the processes that the Entity Framework uses when it updates the database. For example, whenever an entity is deleted you might want to copy it to an archive database. Or whenever a row is updated you might want to write a row to a logging table that records who made the change. You can perform these kinds of tasks in a stored procedure that's called whenever the Entity Framework deletes an entity or updates an entity.

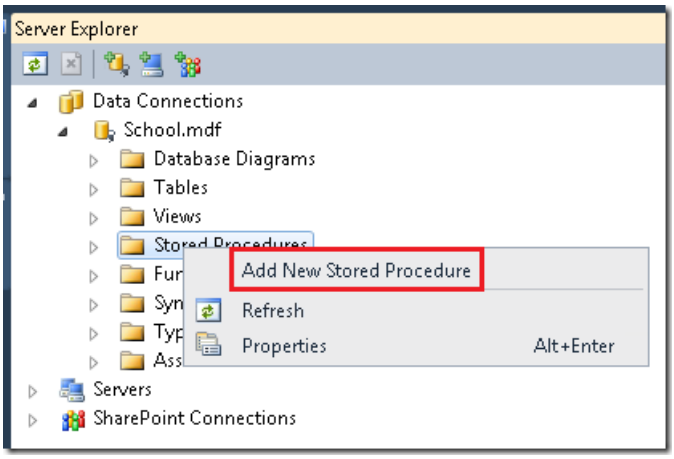
As in the previous tutorial, you'll not create any new pages. Instead, you'll change the way the Entity Framework accesses the database for some of the pages you already created.

In this tutorial you'll create stored procedures in the database for inserting `Student` and `Instructor` entities. You'll add them to the data model, and you'll specify that the Entity Framework should use them for adding `Student` and `Instructor` entities to the database. You'll also create a stored procedure that you can use to retrieve `Course` entities.

Creating Stored Procedures in the Database

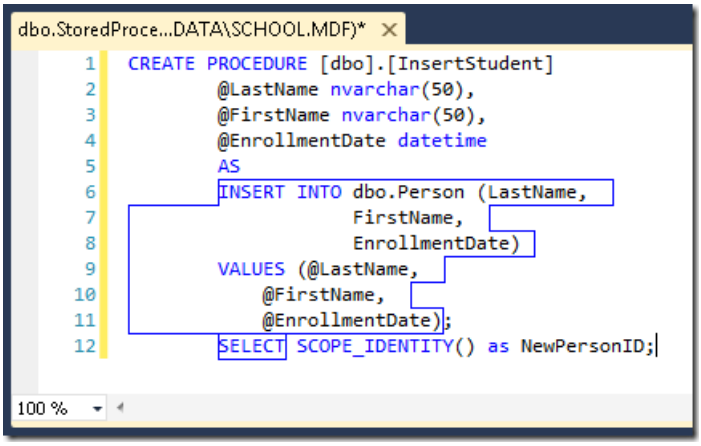
(If you're using the *School.mdf* file from the project available for download with this tutorial, you can skip this section because the stored procedures already exist.)

In **Server Explorer**, expand *School.mdf*, right-click **Stored Procedures**, and select **Add New Stored Procedure**.



Copy the following SQL statements and paste them into the stored procedure window, replacing the skeleton stored procedure.

```
CREATE PROCEDURE [dbo].[InsertStudent]
    @LastName nvarchar(50),
    @FirstName nvarchar(50),
    @EnrollmentDate datetime
    AS
    INSERT INTO dbo.Person (LastName,
                FirstName,
                EnrollmentDate)
    VALUES (@LastName,
        @FirstName,
        @EnrollmentDate);
    SELECT SCOPE_IDENTITY() as NewPersonID;
```



`Student` entities have four properties: `PersonID`, `LastName`, `FirstName`, and `EnrollmentDate`. The database generates the ID value automatically, and the stored procedure accepts parameters for the other three. The stored procedure returns the value of the new row's record key so that the Entity Framework can keep track of that in the version of the entity it keeps in memory.

Save and close the stored procedure window.

Create an `InsertInstructor` stored procedure in the same manner, using the following SQL statements:

```
CREATE PROCEDURE [dbo].[InsertInstructor]
        @LastName nvarchar(50),
    @FirstName nvarchar(50),
    @HireDate datetime
    AS
    INSERT INTO dbo.Person (LastName,
                FirstName,
                HireDate)
    VALUES (@LastName,
        @FirstName,
        @HireDate);
    SELECT SCOPE_IDENTITY() as NewPersonID;
```

Create `Update` stored procedures for `Student` and `Instructor` entities also. (The database already has a `DeletePerson` stored procedure which will work for both `Instructor` and `Student` entities.)

```
CREATE PROCEDURE [dbo].[UpdateStudent]
    @PersonID int,
    @LastName nvarchar(50),
    @FirstName nvarchar(50),
    @EnrollmentDate datetime
    AS
    UPDATE Person SET LastName=@LastName,
            FirstName=@FirstName,
            EnrollmentDate=@EnrollmentDate
    WHERE PersonID=@PersonID;
```

```
CREATE PROCEDURE [dbo].[UpdateInstructor]
    @PersonID int,
    @LastName nvarchar(50),
    @FirstName nvarchar(50),
    @HireDate datetime
    AS
    UPDATE Person SET LastName=@LastName,
            FirstName=@FirstName,
            HireDate=@HireDate
    WHERE PersonID=@PersonID;
```

In this tutorial you'll map all three functions -- insert, update, and delete -- for each entity type. The Entity Framework version 4 allows you to map just one or two of these functions to stored procedures without mapping the others, with one exception: if you map the update function but not the delete function, the Entity Framework will throw an exception when you attempt to delete an entity. In the Entity Framework version 3.5, you did not have this much flexibility in mapping stored procedures: if you mapped one function you were required to map all three.

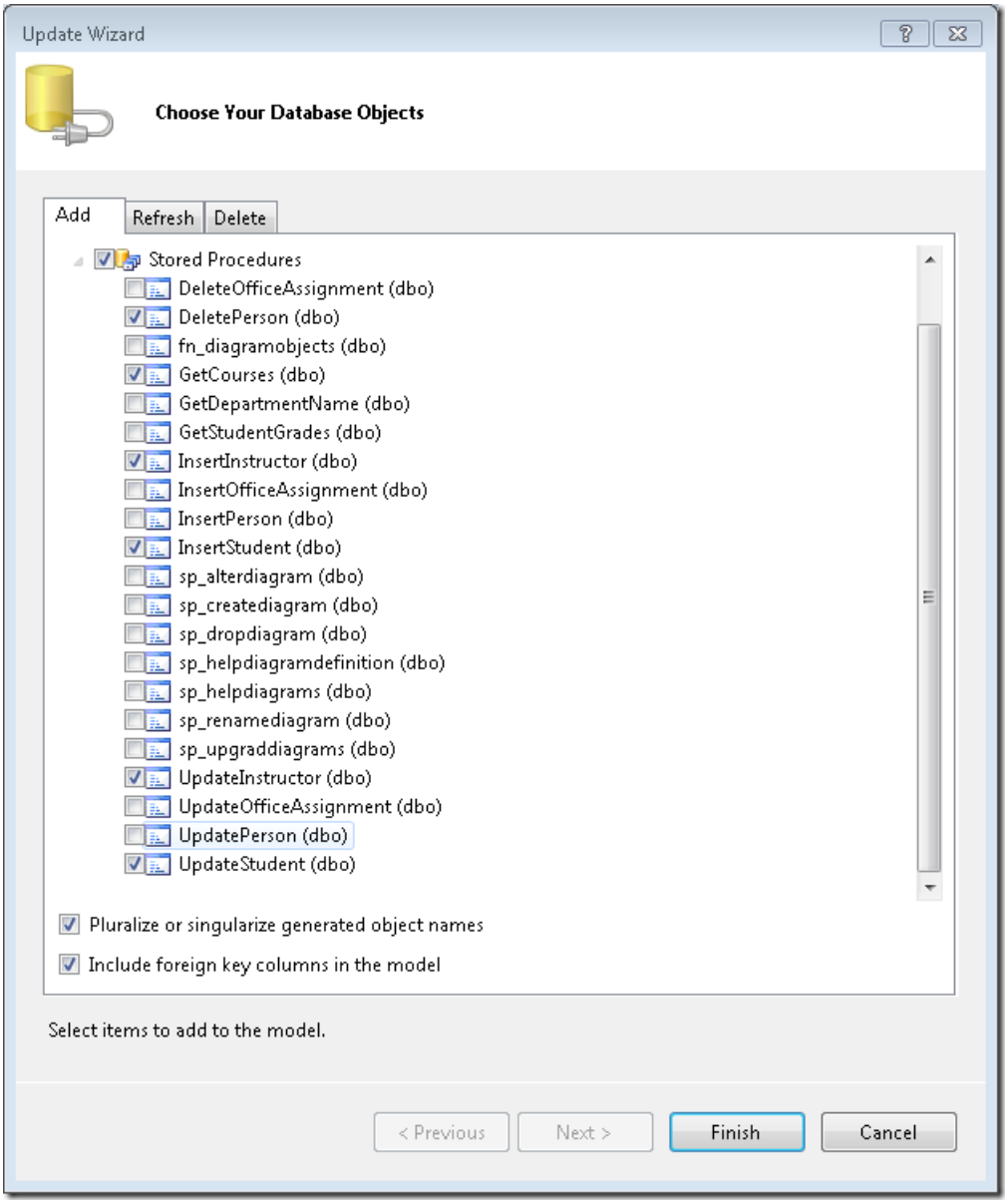
To create a stored procedure that reads rather than updates data, create one that selects all `Course` entities, using the following SQL statements:

```
CREATE PROCEDURE [dbo].[GetCourses]
            AS
            SELECT CourseID, Title, Credits, DepartmentID FROM dbo.Course
```

Adding the Stored Procedures to the Data Model

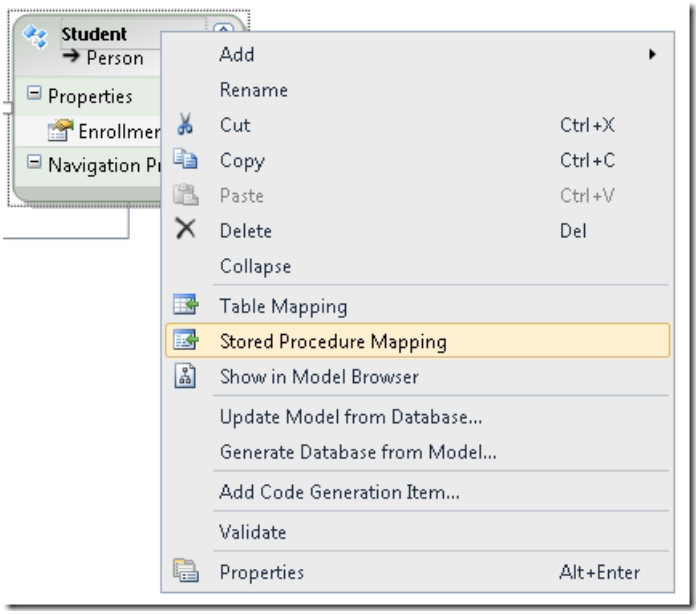
The stored procedures are now defined in the database, but they must be added to the data model to make them available to the Entity Framework. Open *SchoolModel.edmx*, right-click the design surface, and select **Update Model from Database**. In the **Add** tab of the **Choose Your Database Objects** dialog box, expand **Stored**

Procedures, select the newly created stored procedures and the `DeletePerson` stored procedure, and then click **Finish**.

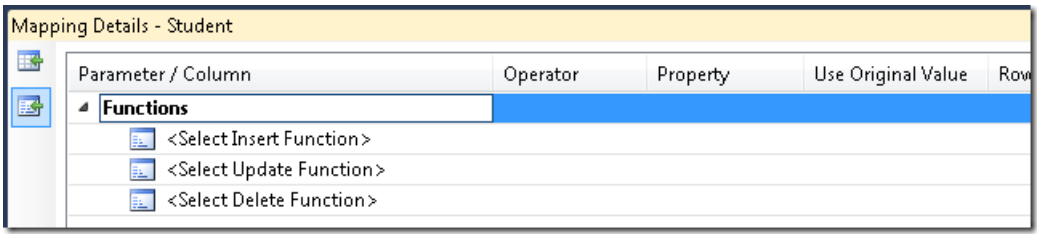


Mapping the Stored Procedures

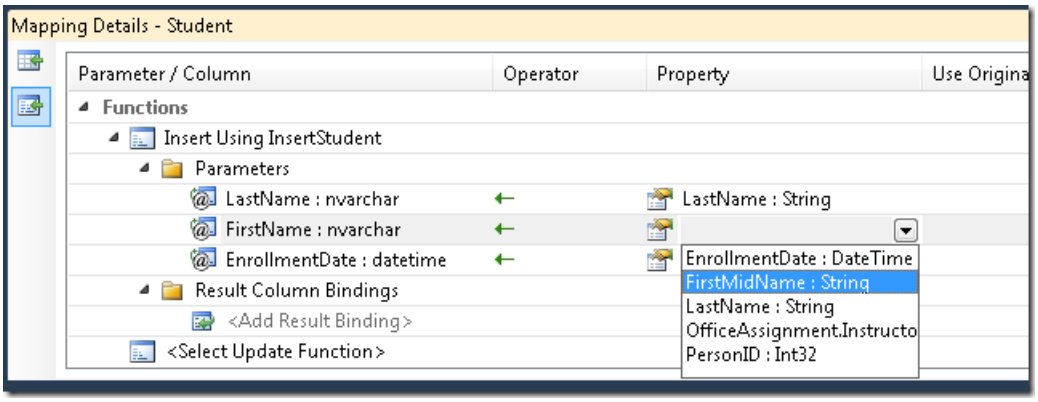
In the data model designer, right-click the `Student` entity and select **Stored Procedure Mapping**.



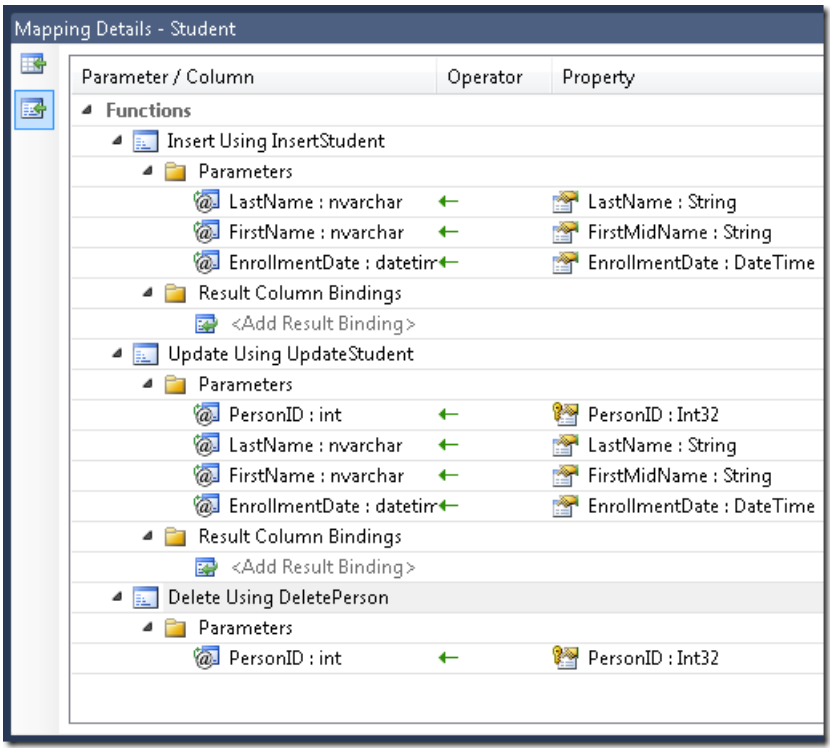
The **Mapping Details** window appears, in which you can specify stored procedures that the Entity Framework should use for inserting, updating, and deleting entities of this type.



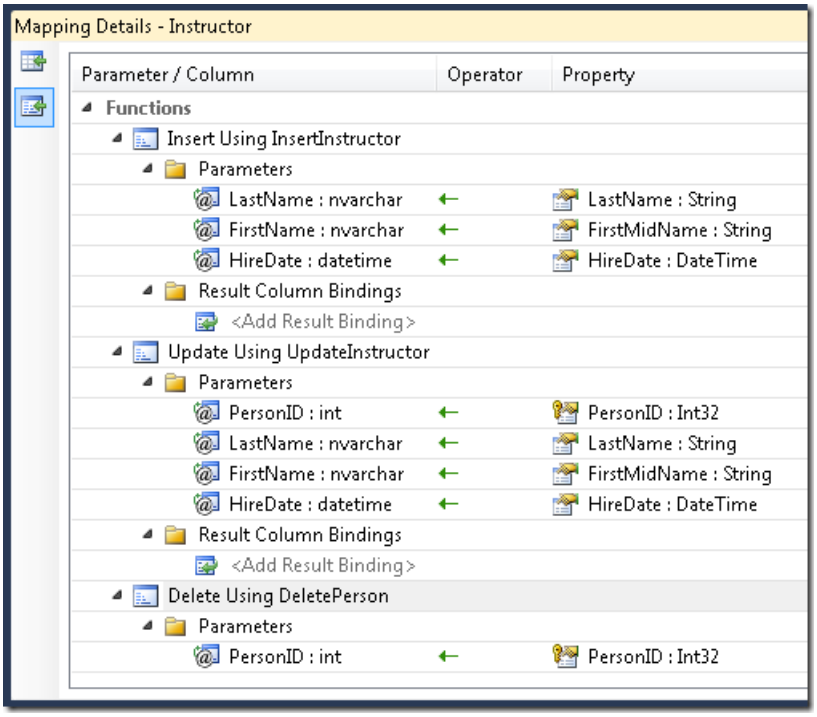
Set the **Insert** function to **InsertStudent**. The window shows a list of stored procedure parameters, each of which must be mapped to an entity property. Two of these are mapped automatically because the names are the same. There's no entity property named `FirstName`, so you must manually select `FirstMidName` from a drop-down list that shows available entity properties. (This is because you changed the name of the `FirstName` property to `FirstMidName` in the first tutorial.)



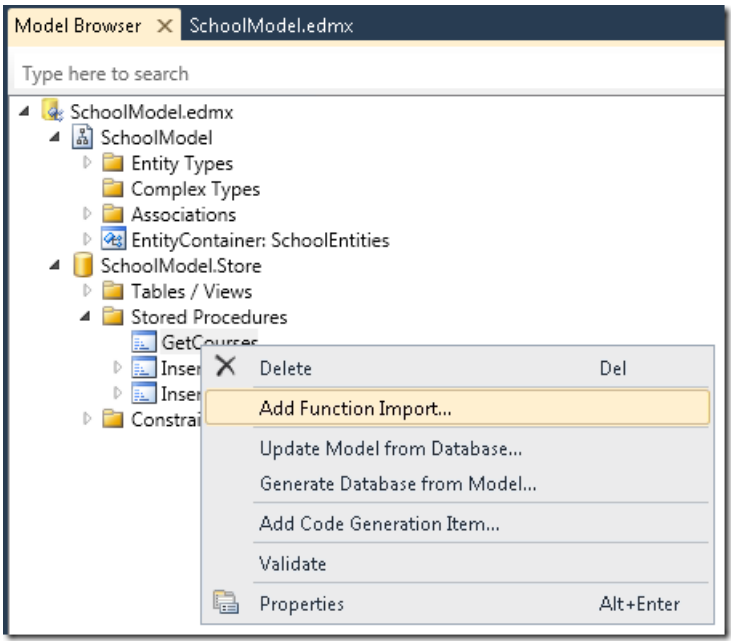
In the same **Mapping Details** window, map the `Update` function to the `UpdateStudent` stored procedure (make sure you specify `FirstMidName` as the parameter value for `FirstName`, as you did for the `Insert` stored procedure) and the `Delete` function to the `DeletePerson` stored procedure.



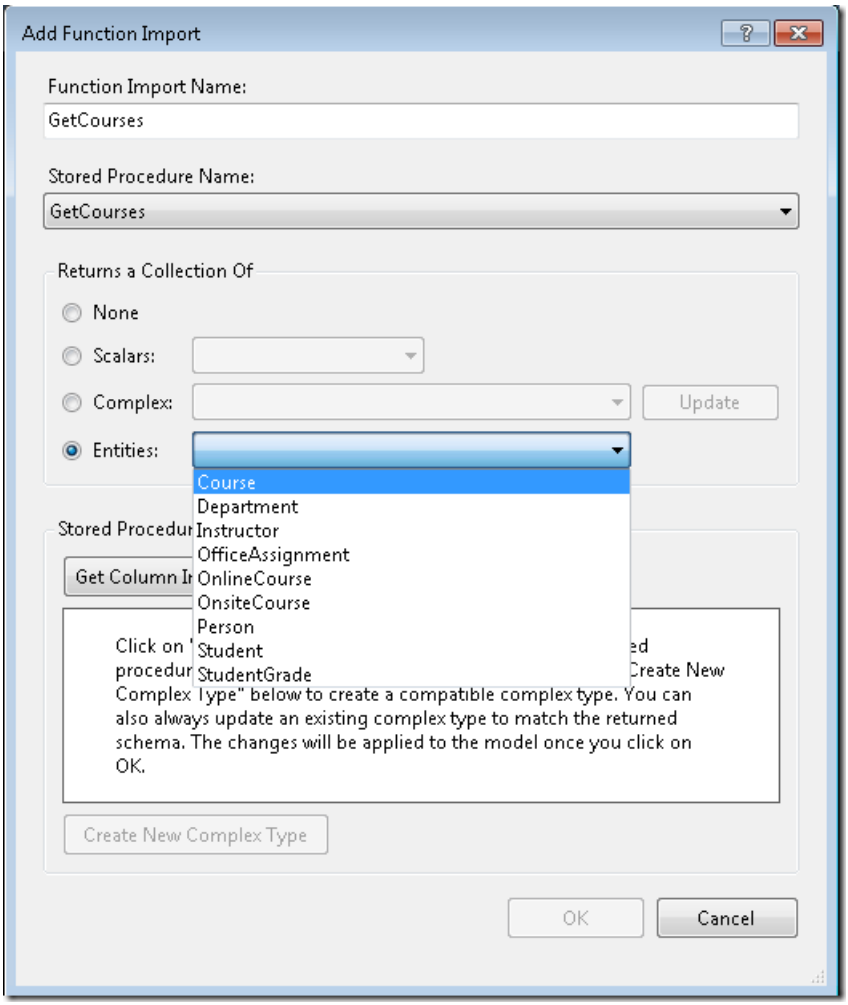
Follow the same procedure to map the insert, update, and delete stored procedures for instructors to the `Instructor` entity.



For stored procedures that read rather than update data, you use the **Model Browser** window to map the stored procedure to the entity type it returns. In the data model designer, right-click the design surface and select **Model Browser**. Open the **SchoolModel.Store** node and then open the **Stored Procedures** node. Then right-click the `GetCourses` stored procedure and select **Add Function Import**.

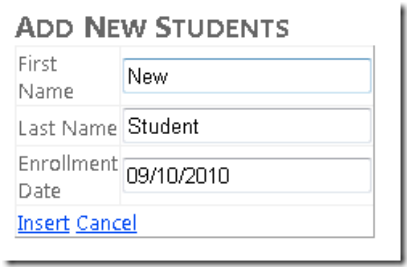


In the **Add Function Import** dialog box, under **Returns a Collection Of** select **Entities**, and then select `Course` as the entity type returned. When you're done, click **OK**. Save and close the *.edmx* file.

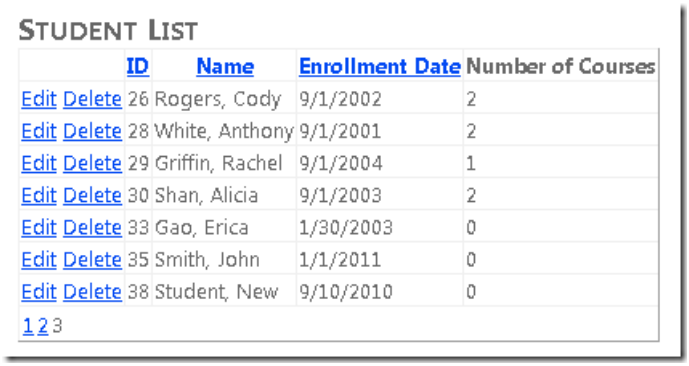


Using Insert, Update, and Delete Stored Procedures

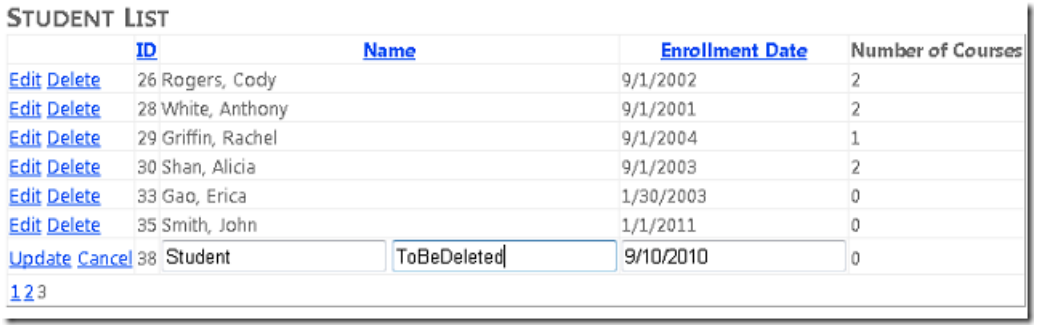
Stored procedures to insert, update, and delete data are used by the Entity Framework automatically after you've added them to the data model and mapped them to the appropriate entities. You can now run the *StudentsAdd.aspx* page, and every time you create a new student, the Entity Framework will use the `InsertStudent` stored procedure to add the new row to the `Student` table.



Run the *Students.aspx* page and the new student appears in the list.



Change the name to verify that the update function works, and then delete the student to verify that the delete function works.



Using Select Stored Procedures

The Entity Framework does not automatically run stored procedures such as `GetCourses`, and you cannot use them with the `EntityDataSource` control. To use them, you call them from code.

Open the *InstructorsCourses.aspx.cs* file. The `PopulateDropDownLists` method uses a LINQ-to-Entities query to retrieve all course entities so that it can loop through the list and determine which ones an instructor is assigned to and which ones are unassigned:

```
var allCourses = (from c in context.Courses
                  select c).ToList();
```

Replace this with the following code:

```
var allCourses = context.GetCourses();
```

The page now uses the `GetCourses` stored procedure to retrieve the list of all courses. Run the page to verify that it works as it did before.

(Navigation properties of entities retrieved by a stored procedure might not be automatically populated with the data related to those entities, depending on `ObjectContext` default settings. For more information, see Loading Related Objects in the MSDN Library.)

In the next tutorial, you'll learn how to use Dynamic Data functionality to make it easier to program and test data formatting and validation rules. Instead of specifying on each web page rules such as data format strings and whether or not a field is required, you can specify such rules in data model metadata and they're automatically applied on every page.

PREVIOUS NEXT

Getting Started with Entity Framework 4.0 Database First and ASP.NET 4 Web Forms - Part 8

2/20/2020 • 5 minutes to read • Edit Online

by Tom Dykstra

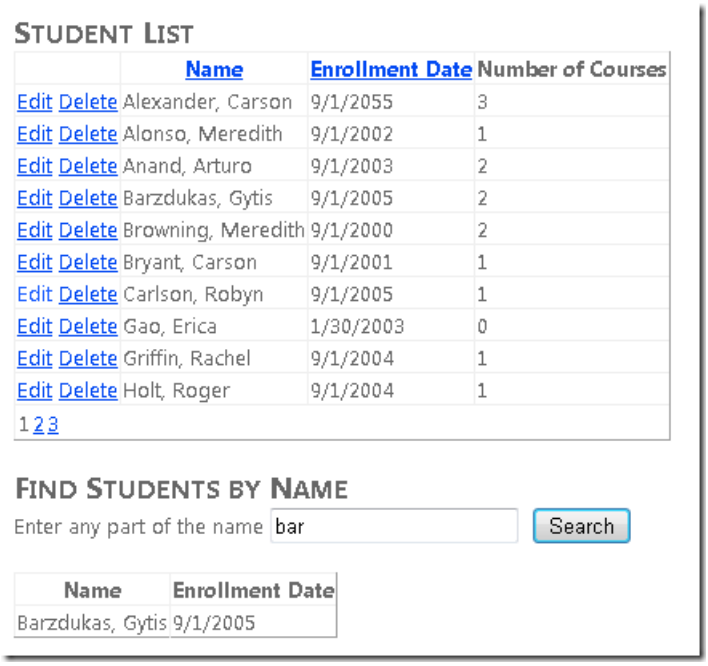
> The Contoso University sample web application demonstrates how to create ASP.NET Web Forms applications using the Entity Framework 4.0 and Visual Studio 2010. For information about the tutorial series, see the first tutorial in the series

Using Dynamic Data Functionality to Format and Validate Data

In the previous tutorial you implemented stored procedures. This tutorial will show you how Dynamic Data functionality can provide the following benefits:

- Fields are automatically formatted for display based on their data type.
- Fields are automatically validated based on their data type.
- You can add metadata to the data model to customize formatting and validation behavior. When you do this, you can add the formatting and validation rules in just one place, and they're automatically applied everywhere you access the fields using Dynamic Data controls.

To see how this works, you'll change the controls you use to display and edit fields in the existing *Students.aspx* page, and you'll add formatting and validation metadata to the name and date fields of the `Student` entity type.



Using DynamicField and DynamicControl Controls

Open the *Students.aspx* page and in the `StudentsGridView` control replace the **Name** and **Enrollment Date** `TemplateField` elements with the following markup:

```
<asp:TemplateField HeaderText="Name" SortExpression="LastName">
                <EditItemTemplate>
                    <asp:DynamicControl ID="LastNameTextBox" runat="server" DataField="LastName" Mode="Edit"
/>
                    <asp:DynamicControl ID="FirstNameTextBox" runat="server" DataField="FirstMidName"
Mode="Edit" />
                </EditItemTemplate>
                <ItemTemplate>
                    <asp:DynamicControl ID="LastNameLabel" runat="server" DataField="LastName"
Mode="ReadOnly" />,
                    <asp:DynamicControl ID="FirstNameLabel" runat="server" DataField="FirstMidName"
Mode="ReadOnly" />
                </ItemTemplate>
            </asp:TemplateField>
            <asp:DynamicField DataField="EnrollmentDate" HeaderText="Enrollment Date"
SortExpression="EnrollmentDate" />
```

This markup uses `DynamicControl` controls in place of `TextBox` and `Label` controls in the student name template field, and it uses a `DynamicField` control for the enrollment date. No format strings are specified.

Add a `ValidationSummary` control after the `StudentsGridView` control.

```
<asp:ValidationSummary ID="StudentsValidationSummary" runat="server" ShowSummary="true"
        DisplayMode="BulletList" Style="color: Red" />
```

In the `SearchGridView` control replace the markup for the **Name** and **Enrollment Date** columns as you did in the `StudentsGridView` control, except omit the `EditItemTemplate` element. The `Columns` element of the `SearchGridView` control now contains the following markup:

```
<asp:TemplateField HeaderText="Name" SortExpression="LastName">
                <ItemTemplate>
                    <asp:DynamicControl ID="LastNameLabel" runat="server" DataField="LastName"
Mode="ReadOnly" />,
                    <asp:DynamicControl ID="FirstNameLabel" runat="server" DataField="FirstMidName"
Mode="ReadOnly" />
                </ItemTemplate>
            </asp:TemplateField>
            <asp:DynamicField DataField="EnrollmentDate" HeaderText="Enrollment Date"
SortExpression="EnrollmentDate" />
```

Open *Students.aspx.cs* and add the following `using` statement:

```
using ContosoUniversity.DAL;
```

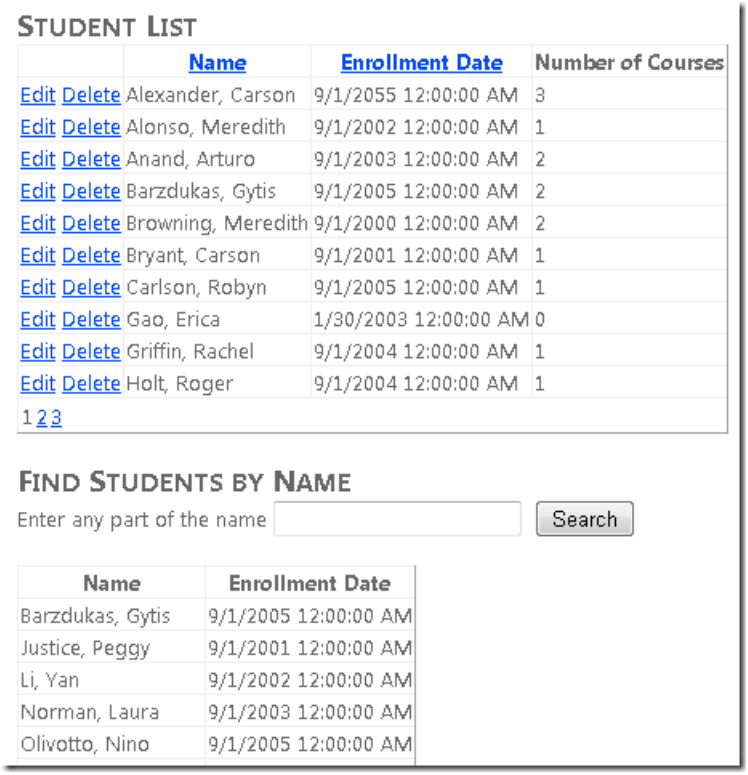
Add a handler for the page's `Init` event:

```
protected void Page_Init(object sender, EventArgs e)
{
    StudentsGridView.EnableDynamicData(typeof(Student));
    SearchGridView.EnableDynamicData(typeof(Student));
}
```

This code specifies that Dynamic Data will provide formatting and validation in these data-bound controls for fields of the `Student` entity. If you get an error message like the following example when you run the page, it typically means you've forgotten to call the `EnableDynamicData` method in `Page_Init`:

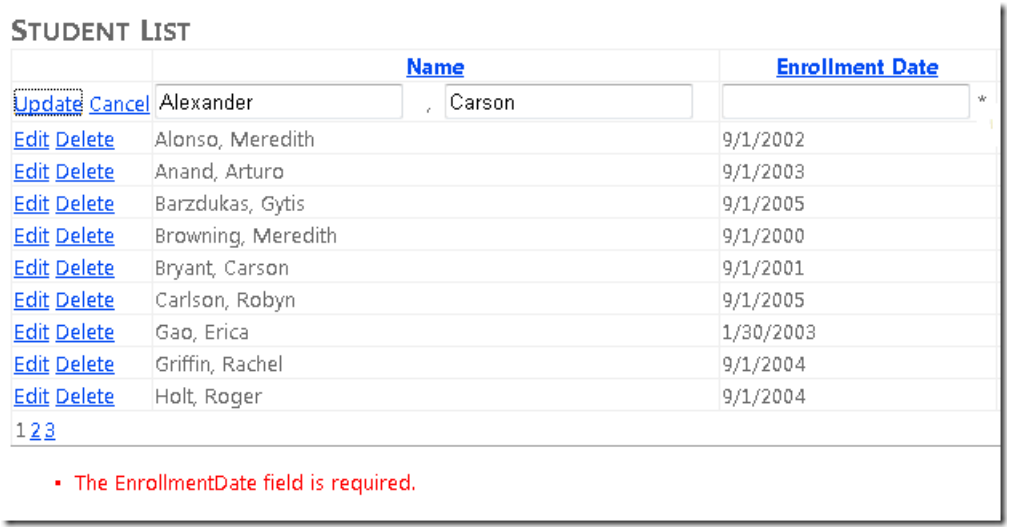
```
Could not determine a MetaTable. A MetaTable could not be determined for the data source
'StudentsEntityDataSource' and one could not be inferred from the request URL.
```

Run the page.

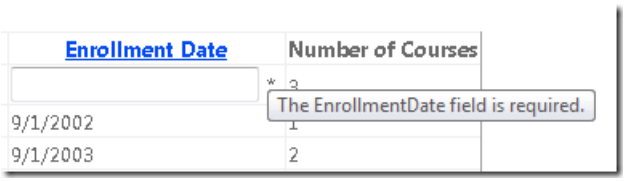


In the **Enrollment Date** column, the time is displayed along with the date because the property type is `DateTime`. You'll fix that later.

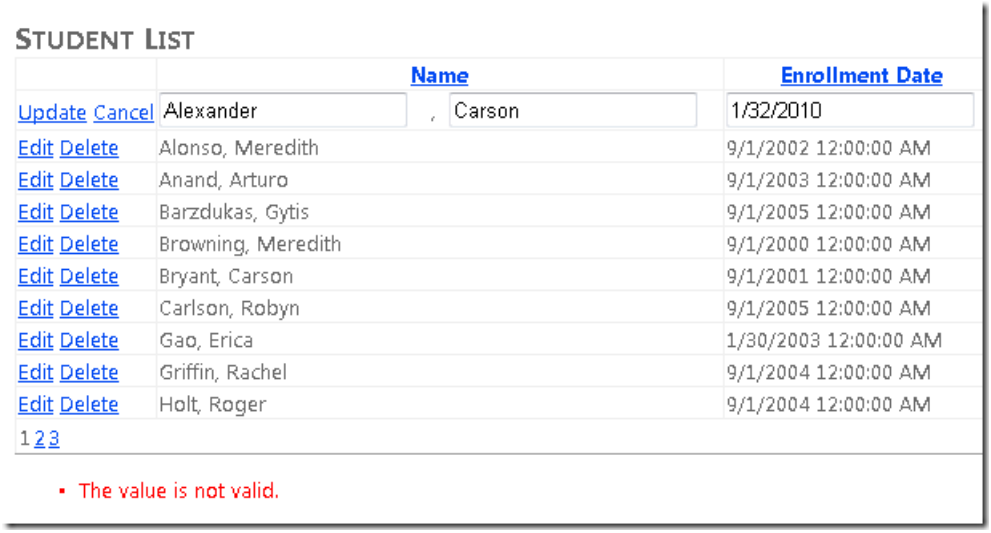
For now, notice that Dynamic Data automatically provides basic data validation. For example, click **Edit**, clear the date field, click **Update**, and you see that Dynamic Data automatically makes this a required field because the value is not nullable in the data model. The page displays an asterisk after the field and an error message in the `ValidationSummary` control:



You could omit the `ValidationSummary` control, because you can also hold the mouse pointer over the asterisk to see the error message:



Dynamic Data will also validate that data entered in the **Enrollment Date** field is a valid date:

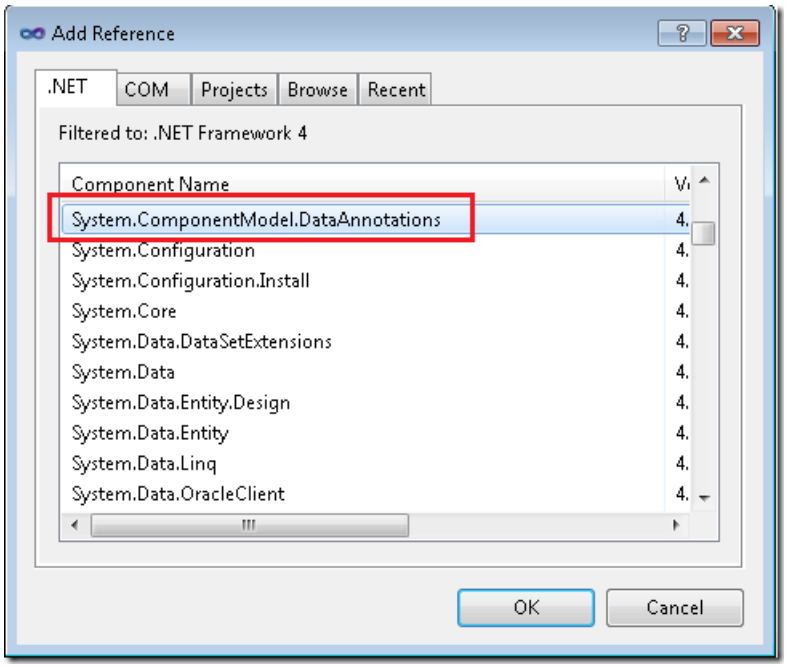


As you can see, this is a generic error message. In the next section you'll see how to customize messages as well as validation and formatting rules.

Adding Metadata to the Data Model

Typically, you want to customize the functionality provided by Dynamic Data. For example, you might change how data is displayed and the content of error messages. You typically also customize data validation rules to provide more functionality than what Dynamic Data provides automatically based on data types. To do this, you create partial classes that correspond to entity types.

In **Solution Explorer**, right-click the **ContosoUniversity** project, select **Add Reference**, and add a reference to `System.ComponentModel.DataAnnotations`.



In the *DAL* folder, create a new class file, name it *Student.cs*, and replace the template code in it with the following code.

```
using System;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.DAL
{
    [MetadataType(typeof(StudentMetadata))]
    public partial class Student
    {
    }

    public class StudentMetadata
    {
        [DisplayFormat(DataFormatString="{0:d}", ApplyFormatInEditMode=true)]
        public DateTime EnrollmentDate { get; set; }

        [StringLength(25, ErrorMessage = "First name must be 25 characters or less in length.")]
        [Required(ErrorMessage="First name is required.")]
        public String FirstMidName { get; set; }

        [StringLength(25, ErrorMessage = "Last name must be 25 characters or less in length.")]
        [Required(ErrorMessage = "Last name is required.")]
        public String LastName { get; set; }
    }
}
```

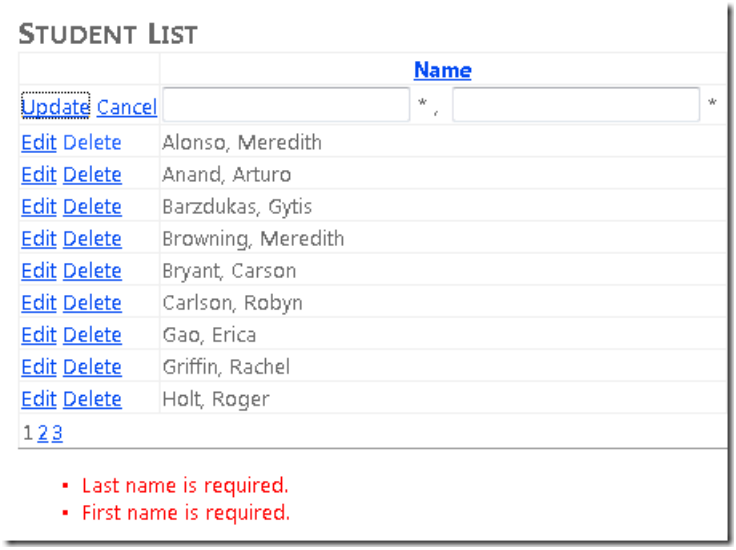
This code creates a partial class for the `Student` entity. The `MetadataType` attribute applied to this partial class identifies the class that you're using to specify metadata. The metadata class can have any name, but using the entity name plus "Metadata" is a common practice.

The attributes applied to properties in the metadata class specify formatting, validation, rules, and error messages. The attributes shown here will have the following results:

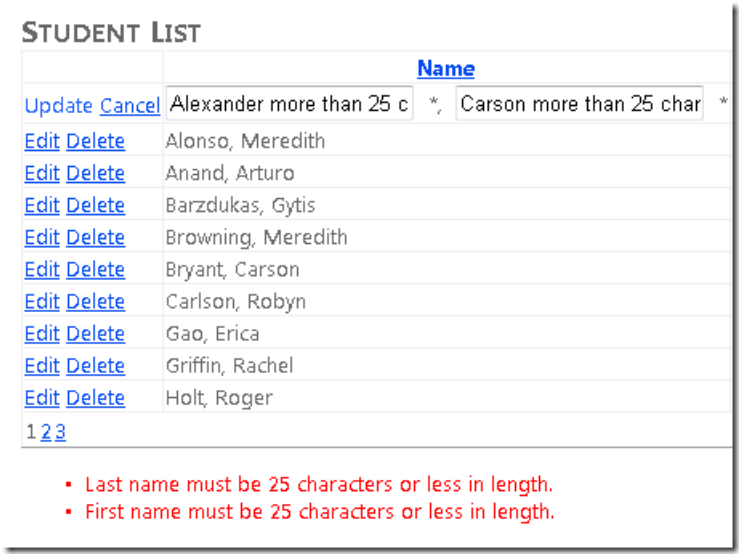
- `EnrollmentDate` will display as a date (without a time).
- Both name fields must be 25 characters or less in length, and a custom error message is provided.
- Both name fields are required, and a custom error message is provided.

Run the *Students.aspx* page again, and you see that the dates are now displayed without times:

Edit a row and try to clear the values in the name fields. The asterisks indicating field errors appear as soon as you leave a field, before you click **Update**. When you click **Update**, the page displays the error message text you specified.



Try to enter names that are longer than 25 characters, click **Update**, and the page displays the error message text you specified.



Now that you've set up these formatting and validation rules in the data model metadata, the rules will automatically be applied on every page that displays or allows changes to these fields, so long as you use `DynamicControl` or `DynamicField` controls. This reduces the amount of redundant code you have to write, which makes programming and testing easier, and it ensures that data formatting and validation are consistent throughout an application.

More Information

This concludes this series of tutorials on Getting Started with the Entity Framework. For more resources to help you learn how to use the Entity Framework, continue with the first tutorial in the next Entity Framework tutorial series or visit the following sites:

- Entity Framework FAQ
- The Entity Framework Team Blog
- Entity Framework in the MSDN Library
- Entity Framework in the MSDN Data Developer Center
- EntityDataSource Web Server Control Overview in the MSDN Library
- EntityDataSource control API reference in the MSDN Library
- Entity Framework Forums on MSDN
- Julie Lerman's blog

PREVIOUS

ASP.NET 4 - Continuing with Entity Framework 4 and Web Forms

2/20/2020 • 2 minutes to read • Edit Online

How to build the Contoso U application as an n-tier application using the ObjectDataSource control and how to handle concurrency and maximize performance.

- Entity Framework 4.0 and the ObjectDataSource Control, Part 1: Getting Started
- Entity Framework 4.0 and the ObjectDataSource Control, Part 2: Adding a Business Logic Layer and Unit Tests
- Entity Framework 4.0 and the ObjectDataSource Control, Part 3: Sorting and Filtering
- Handling Concurrency with the Entity Framework 4.0 in ASP.NET 4
- Maximizing Performance with the Entity Framework 4.0 in ASP.NET 4
- What's New in the Entity Framework 4.0

Using the Entity Framework 4.0 and the ObjectDataSource Control, Part 1: Getting Started

2/20/2020 • 22 minutes to read • Edit Online

by Tom Dykstra

This tutorial series builds on the Contoso University web application that is created by the Getting Started with the Entity Framework 4.0 tutorial series. If you didn't complete the earlier tutorials, as a starting point for this tutorial you can download the application that you would have created. You can also download the application that is created by the complete tutorial series.

The Contoso University sample web application demonstrates how to create ASP.NET Web Forms applications using the Entity Framework 4.0 and Visual Studio 2010. The sample application is a website for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments.

The tutorial shows examples in C#. The downloadable sample contains code in both C# and Visual Basic.

Database First

There are three ways you can work with data in the Entity Framework: *Database First*, *Model First*, and *Code First*. This tutorial is for Database First. For information about the differences between these workflows and guidance on how to choose the best one for your scenario, see Entity Framework Development Workflows.

Web Forms

Like the Getting Started series, this tutorial series uses the ASP.NET Web Forms model and assumes you know how to work with ASP.NET Web Forms in Visual Studio. If you don't, see Getting Started with ASP.NET 4.5 Web Forms. If you prefer to work with the ASP.NET MVC framework, see Getting Started with the Entity Framework using ASP.NET MVC.

Software versions

SHOWN IN THE TUTORIAL	ALSO WORKS WITH
Windows 7	Windows 8
Visual Studio 2010	Visual Studio 2010 Express for Web. The tutorial has not been tested with later versions of Visual Studio. There are many differences in menu selections, dialog boxes, and templates.
.NET 4	.NET 4.5 is backward compatible with .NET 4, but the tutorial has not been tested with .NET 4.5.

SHOWN IN THE TUTORIAL	ALSO WORKS WITH
Entity Framework 4	The tutorial has not been tested with later versions of Entity Framework. Starting with Entity Framework 5, EF uses by default the `DbContext API` that was introduced with EF 4.1. The EntityDataSource control was designed to use the `ObjectContext` API. For information about how to use the EntityDataSource control with the `DbContext` API, see this blog post.

Questions

If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET Entity Framework forum, the Entity Framework and LINQ to Entities forum, or StackOverflow.com.

The `EntityDataSource` control enables you to create an application very quickly, but it typically requires you to keep a significant amount of business logic and data-access logic in your *.aspx* pages. If you expect your application to grow in complexity and to require ongoing maintenance, you can invest more development time up front in order to create an *n-tier* or *layered* application structure that's more maintainable. To implement this architecture, you separate the presentation layer from the business logic layer (BLL) and the data access layer (DAL). One way to implement this structure is to use the `ObjectDataSource` control instead of the `EntityDataSource` control. When you use the `ObjectDataSource` control, you implement your own data-access code and then invoke it in *.aspx* pages using a control that has many of the same features as other data-source controls. This lets you combine the advantages of an n-tier approach with the benefits of using a Web Forms control for data access.

The `ObjectDataSource` control gives you more flexibility in other ways as well. Because you write your own data-access code, it's easier to do more than just read, insert, update, or delete a specific entity type, which are the tasks that the `EntityDataSource` control is designed to perform. For example, you can perform logging every time an entity is updated, archive data whenever an entity is deleted, or automatically check and update related data as needed when inserting a row with a foreign key value.

Business Logic and Repository Classes

An `ObjectDataSource` control works by invoking a class that you create. The class includes methods that retrieve and update data, and you provide the names of those methods to the `ObjectDataSource` control in markup. During rendering or postback processing, the `ObjectDataSource` calls the methods that you've specified.

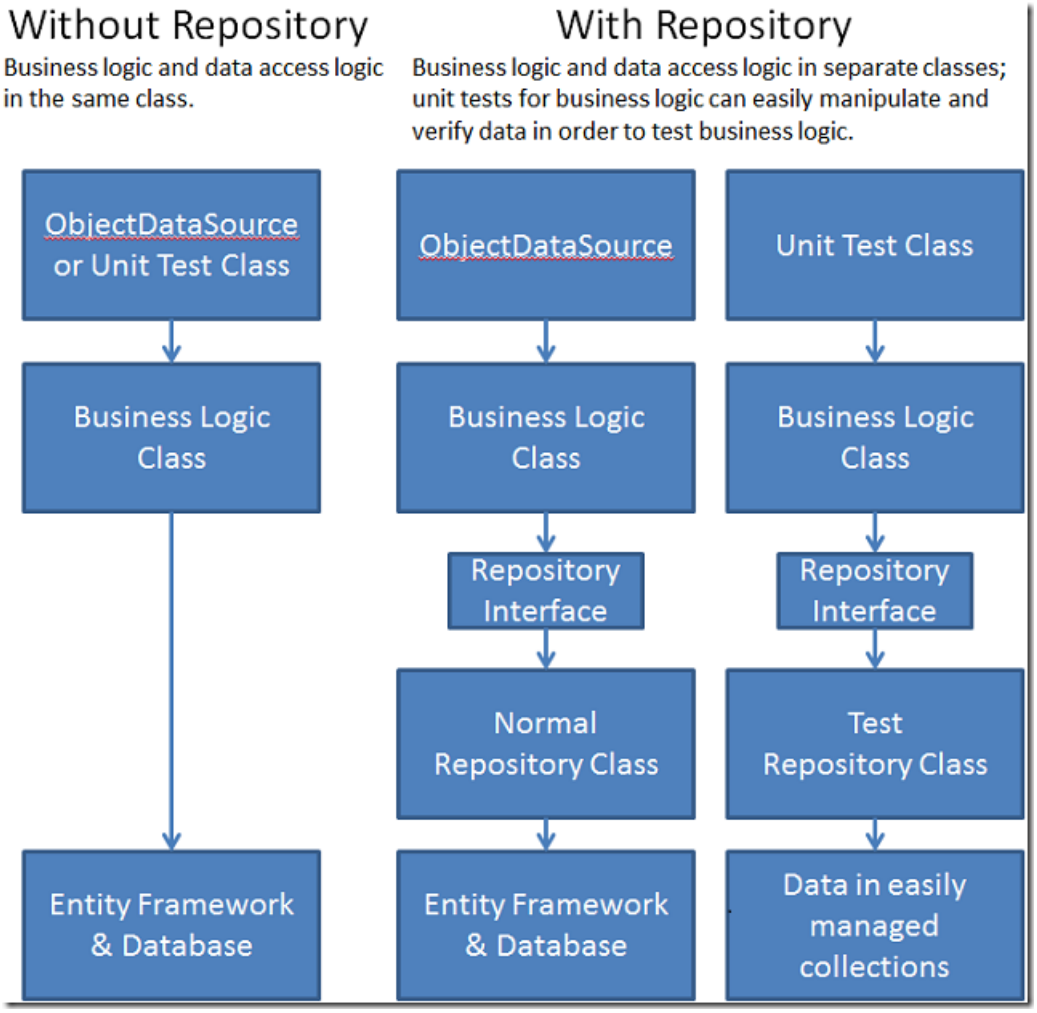
Besides basic CRUD operations, the class that you create to use with the `ObjectDataSource` control might need to execute business logic when the `ObjectDataSource` reads or updates data. For example, when you update a department, you might need to validate that no other departments have the same administrator because one person cannot be administrator of more than one department.

In some `ObjectDataSource` documentation, such as the ObjectDataSource Class overview, the control calls a class referred to as a *business object* that includes both business logic and data-access logic. In this tutorial you will create separate classes for business logic and for data-access logic. The class that encapsulates data-access logic is called a *repository*. The business logic class includes both business-logic methods and data-access methods, but the data-access methods call the repository to perform data-access tasks.

You will also create an abstraction layer between your BLL and DAL that facilitates automated unit testing of the BLL. This abstraction layer is implemented by creating an interface and using the interface when you instantiate the repository in the business-logic class. This makes it possible for you to provide the business-logic class with a reference to any object that implements the repository interface. For normal operation, you provide a repository object that works with the Entity Framework. For testing, you provide a repository object that works with data

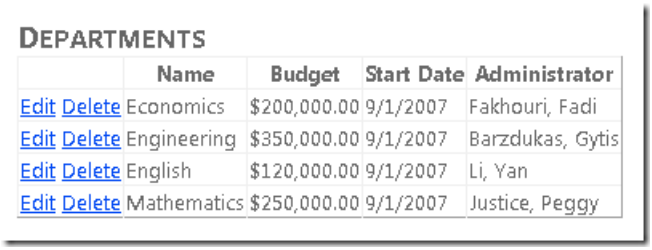
stored in a way that you can easily manipulate, such as class variables defined as collections.

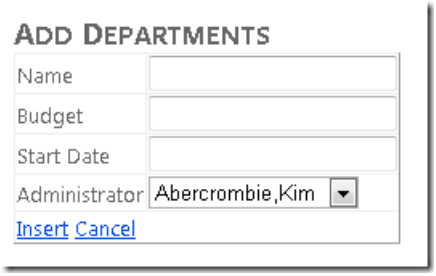
The following illustration shows the difference between a business-logic class that includes data-access logic without a repository and one that uses a repository.



You will begin by creating web pages in which the `ObjectDataSource` control is bound directly to a repository because it only performs basic data-access tasks. In the next tutorial you will create a business logic class with validation logic and bind the `ObjectDataSource` control to that class instead of to the repository class. You will also create unit tests for the validation logic. In the third tutorial in this series you will add sorting and filtering functionality to the application.

The pages you create in this tutorial work with the `Departments` entity set of the data model that you created in the Getting Started tutorial series.





Updating the Database and the Data Model

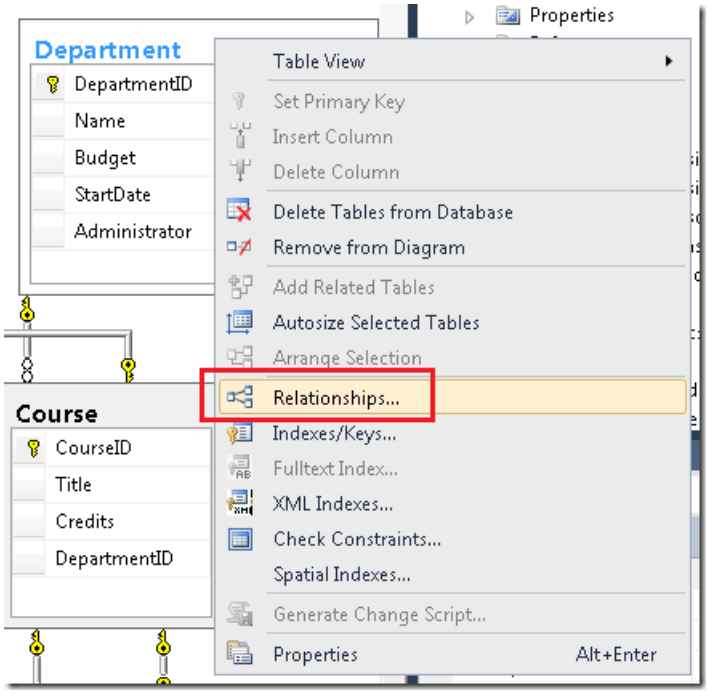
You will begin this tutorial by making two changes to the database, both of which require corresponding changes to the data model that you created in the Getting Started with the Entity Framework and Web Forms tutorials. In one of those tutorials, you made changes in the designer manually to synchronize the data model with the database after a database change. In this tutorial, you will use the designer's **Update Model From Database** tool to update the data model automatically.

Adding a Relationship to the Database

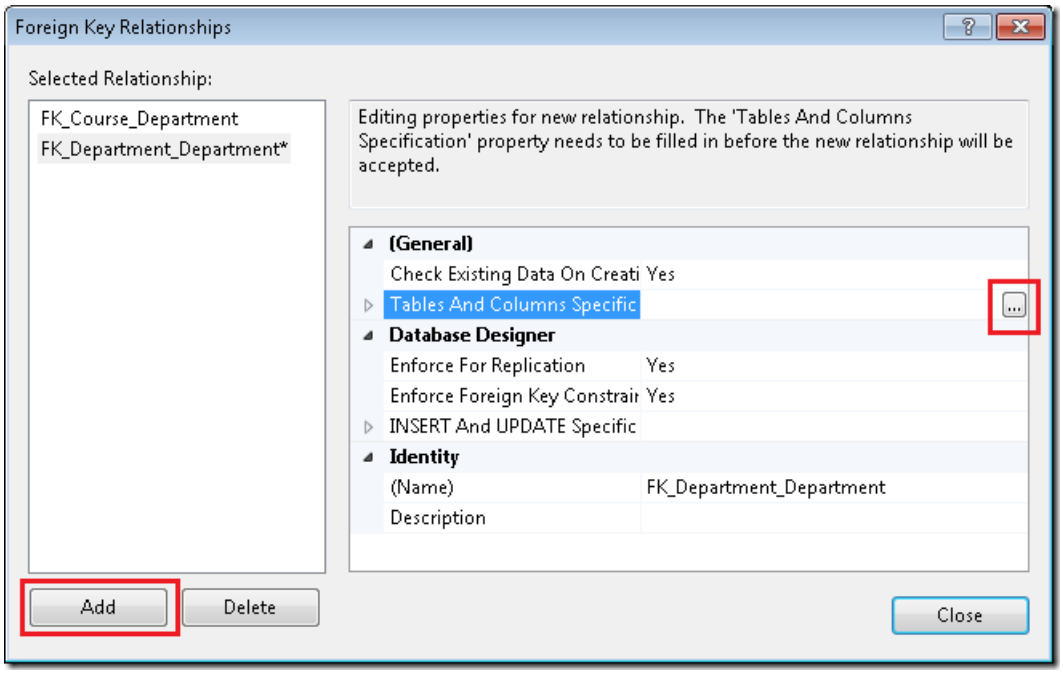
In Visual Studio, open the Contoso University web application you created in the Getting Started with the Entity Framework and Web Forms tutorial series, and then open the `SchoolDiagram` database diagram.

If you look at the `Department` table in the database diagram, you will see that it has an `Administrator` column. This column is a foreign key to the `Person` table, but no foreign key relationship is defined in the database. You need to create the relationship and update the data model so that the Entity Framework can automatically handle this relationship.

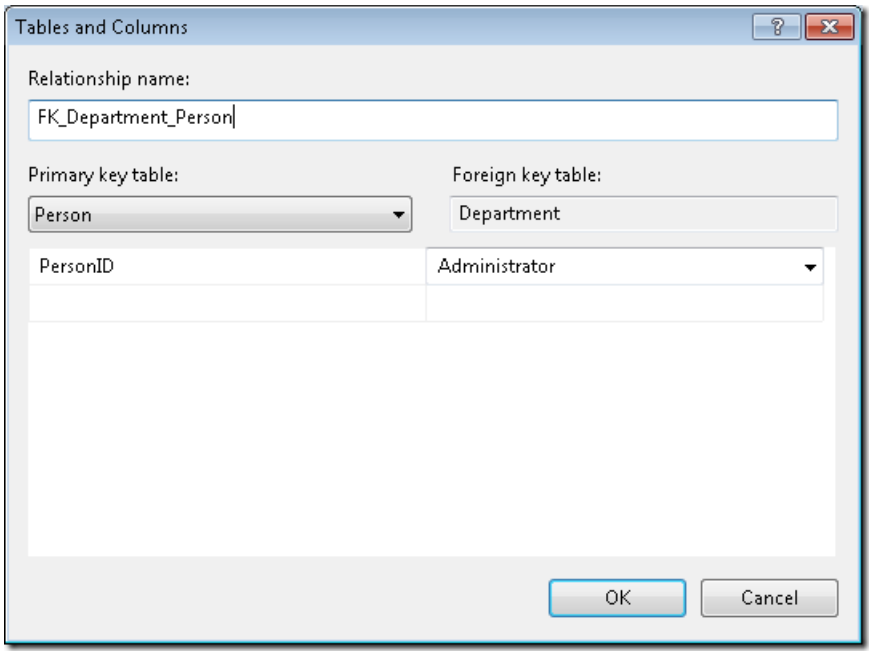
In the database diagram, right-click the `Department` table, and select **Relationships**.



In the **Foreign Key Relationships** box click **Add**, then click the ellipsis for **Tables and Columns Specification**.



In the **Tables and Columns** dialog box, set the primary key table and field to `Person` and `PersonID`, and set the foreign key table and field to `Department` and `Administrator`. (When you do this, the relationship name will change from `FK_Department_Department` to `FK_Department_Person`.)



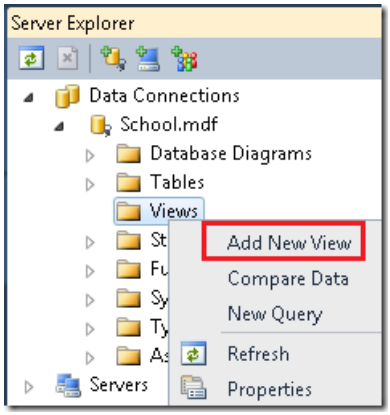
Click **OK** in the **Tables and Columns** box, click **Close** in the **Foreign Key Relationships** box, and save the changes. If you're asked if you want to save the `Person` and `Department` tables, click **Yes**.

> **NOTE**
>
> If you've deleted `Person` rows that correspond to data that's already in the `Administrator` column, you will not be able to save this change. In that case, use the table editor in **Server Explorer** to make sure that the `Administrator` value in every `Department` row contains the ID of a record that actually exists in the `Person` table.
>
> After you save the change, you will not be able to delete a row from the `Person` table if that person is a department administrator. In a production application, you would provide a specific error message when a database constraint prevents a deletion, or you would specify a cascading delete. For an example of how to specify a cascading delete, see The Entity Framework and ASP.NET – Getting Started Part 2.

Adding a View to the Database

In the new *Departments.aspx* page that you will be creating, you want to provide a drop-down list of instructors, with names in "last, first" format so that users can select department administrators. To make it easier to do that, you will create a view in the database. The view will consist of just the data needed by the drop-down list: the full name (properly formatted) and the record key.

In **Server Explorer**, expand *School.mdf*, right-click the **Views** folder, and select **Add New View**.



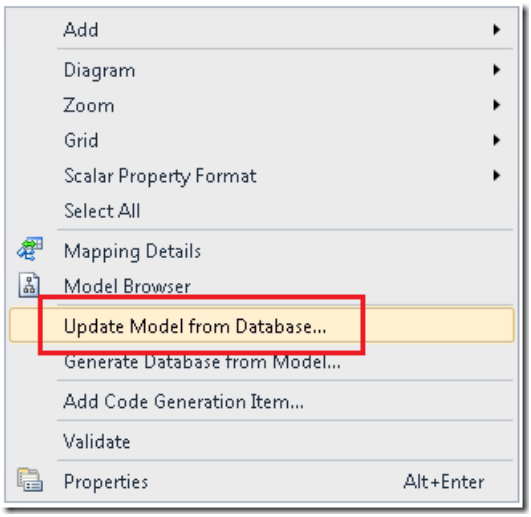
Click **Close** when the **Add Table** dialog box appears, and paste the following SQL statement into the SQL pane:

```
SELECT        LastName + ',' + FirstName AS FullName, PersonID
FROM          dbo.Person
WHERE        (HireDate IS NOT NULL)
```

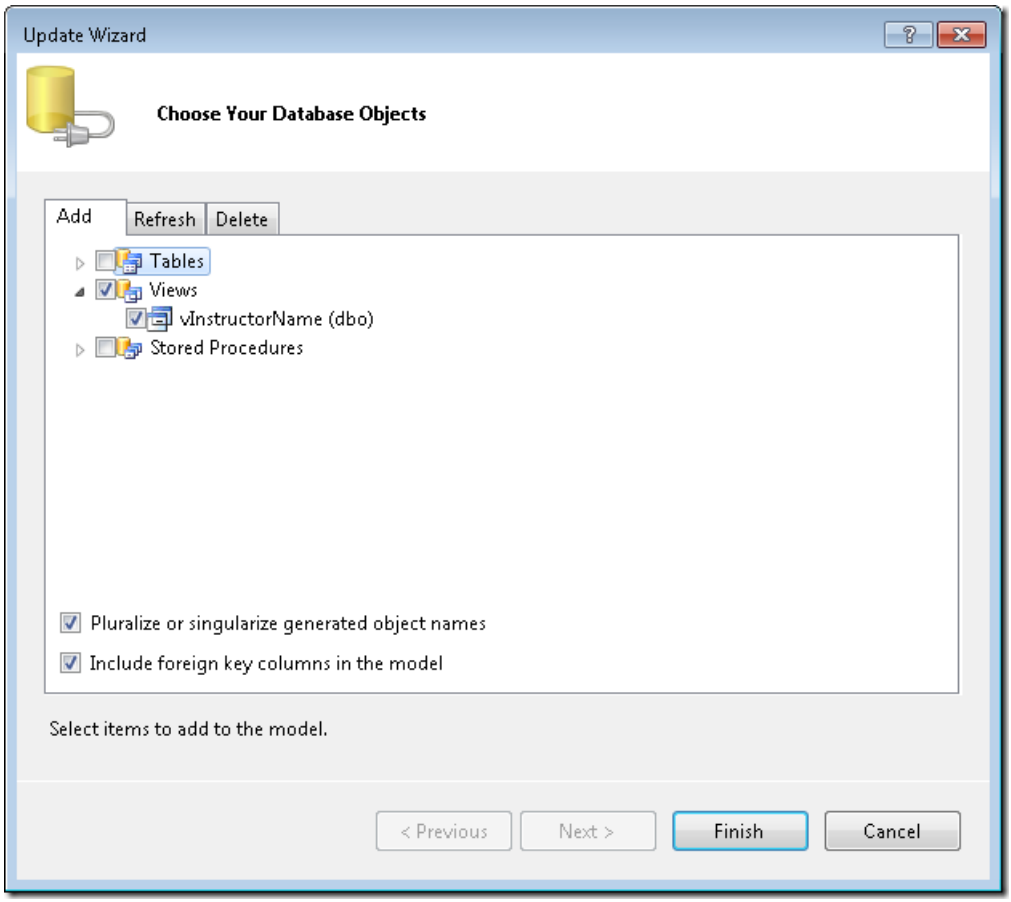
Save the view as `vInstructorName`.

Updating the Data Model

In the *DAL* folder, open the *SchoolModel.edmx* file, right-click the design surface, and select **Update Model from Database**.

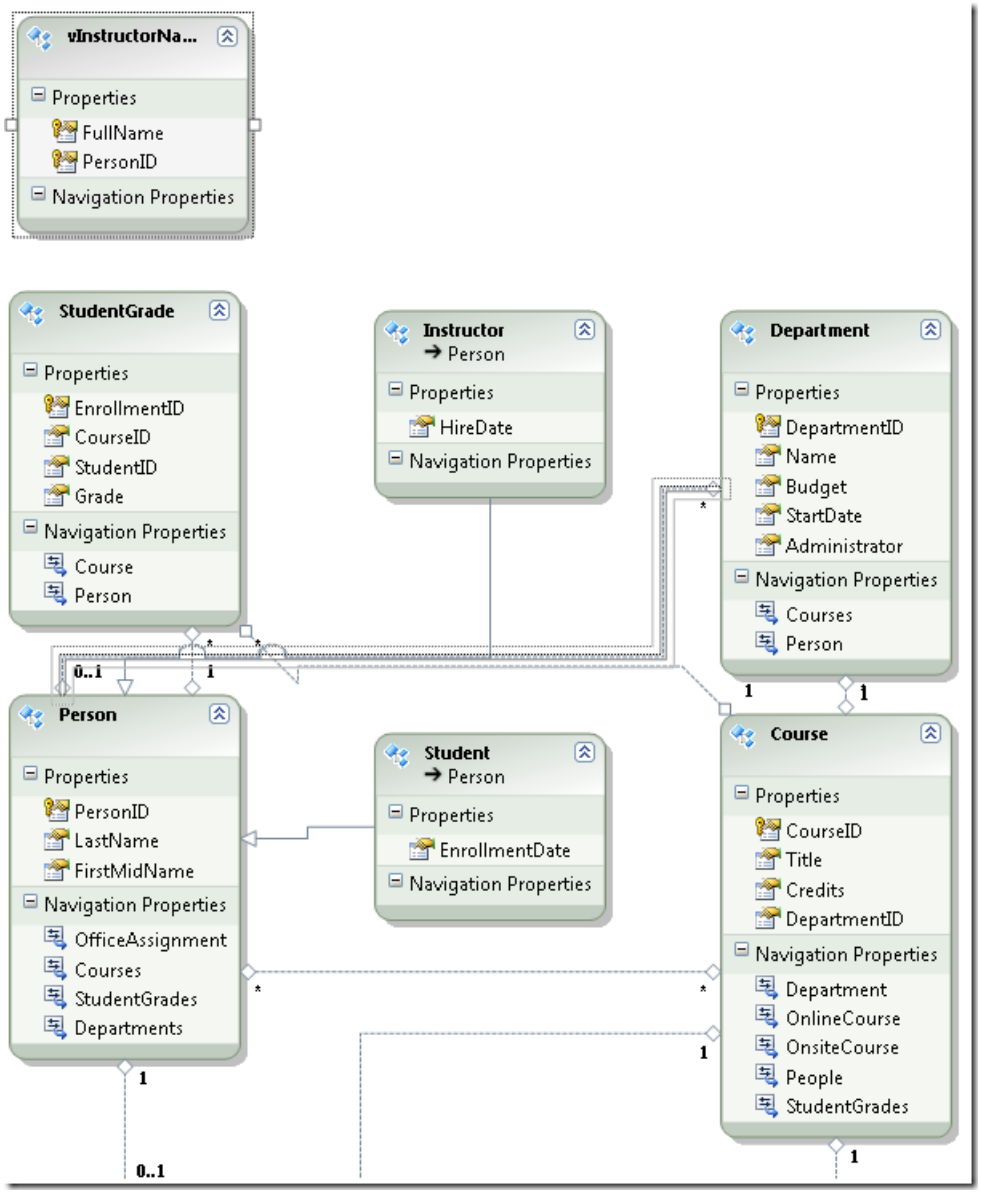


In the **Choose Your Database Objects** dialog box, select the **Add** tab and select the view you just created.



Click **Finish**.

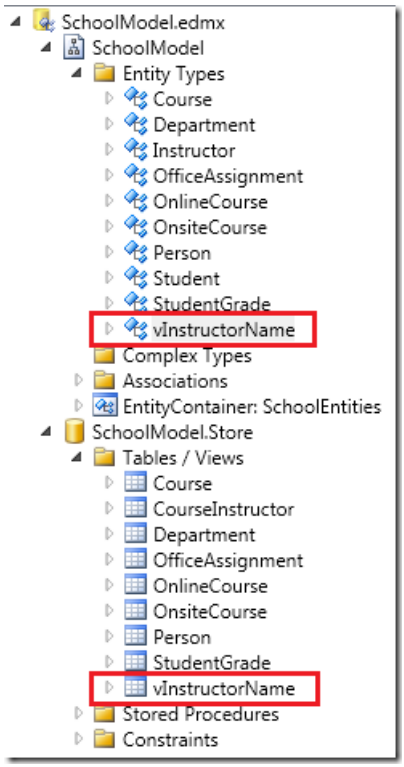
In the designer, you see that the tool created a `vInstructorName` entity and a new association between the `Department` and `Person` entities.



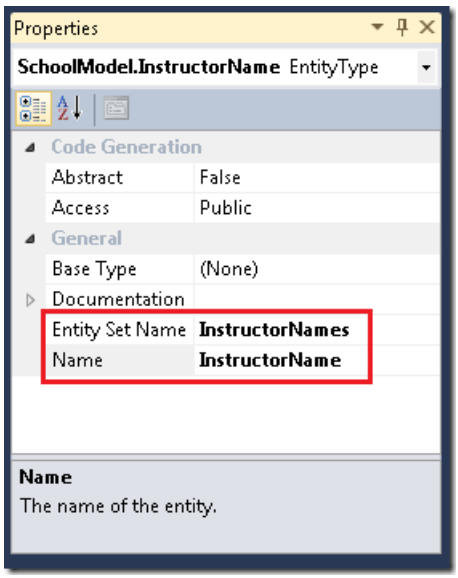
> **NOTE**
>
> In the **Output** and **Error List** windows you might see a warning message informing you that the tool automatically created a primary key for the new `vInstructorName` view. This is expected behavior.

When you refer to the new `vInstructorName` entity in code, you don't want to use the database convention of prefixing a lower-case "v" to it. Therefore, you will rename the entity and entity set in the model.

Open the **Model Browser**. You see `vInstructorName` listed as an entity type and a view.



Under **SchoolModel** (not **SchoolModel.Store**), right-click **vInstructorName** and select **Properties**. In the **Properties** window, change the **Name** property to "InstructorName" and change the **Entity Set Name** property to "InstructorNames".



Save and close the data model, and then rebuild the project.

Using a Repository Class and an ObjectDataSource Control

Create a new class file in the *DAL* folder, name it *SchoolRepository.cs*, and replace the existing code with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using ContosoUniversity.DAL;

namespace ContosoUniversity.DAL
{
    public class SchoolRepository : IDisposable
    {
        private SchoolEntities context = new SchoolEntities();

        public IEnumerable<Department> GetDepartments()
        {
            return context.Departments.Include("Person").ToList();
        }

        private bool disposedValue = false;

        protected virtual void Dispose(bool disposing)
        {
            if (!this.disposedValue)
            {
                if (disposing)
                {
                    context.Dispose();
                }
            }
            this.disposedValue = true;
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

    }
}
```

This code provides a single `GetDepartments` method that returns all of the entities in the `Departments` entity set. Because you know that you will be accessing the `Person` navigation property for every row returned, you specify eager loading for that property by using the `Include` method. The class also implements the `IDisposable` interface to ensure that the database connection is released when the object is disposed.

> **NOTE**
> A common practice is to create a repository class for each entity type. In this tutorial, one repository class for multiple entity types is used. For more information about the repository pattern, see the posts in the Entity Framework team's blog and Julie Lerman's blog.

The `GetDepartments` method returns an `IEnumerable` object rather than an `IQueryable` object in order to ensure that the returned collection is usable even after the repository object itself is disposed. An `IQueryable` object can cause database access whenever it's accessed, but the repository object might be disposed by the time a databound control attempts to render the data. You could return another collection type, such as an `IList` object instead of an `IEnumerable` object. However, returning an `IEnumerable` object ensures that you can perform typical read-only list processing tasks such as `foreach` loops and LINQ queries, but you cannot add to or remove items in the collection, which might imply that such changes would be persisted to the database.

Create a *Departments.aspx* page that uses the *Site.Master* master page, and add the following markup in the `Content` control named `Content2`:

```
<h2>Departments</h2>
    <asp:ObjectDataSource ID="DepartmentsObjectDataSource" runat="server"
        TypeName="ContosoUniversity.DAL.SchoolRepository"
        DataObjectTypeName="ContosoUniversity.DAL.Department"
        SelectMethod="GetDepartments" >
    </asp:ObjectDataSource>
    <asp:GridView ID="DepartmentsGridView" runat="server" AutoGenerateColumns="False"
        DataSourceID="DepartmentsObjectDataSource"  >
        <Columns>
            <asp:CommandField ShowEditButton="True" ShowDeleteButton="True"
                ItemStyle-VerticalAlign="Top">
            </asp:CommandField>
            <asp:DynamicField DataField="Name" HeaderText="Name" SortExpression="Name" ItemStyle-
VerticalAlign="Top" />
            <asp:DynamicField DataField="Budget" HeaderText="Budget" SortExpression="Budget" ItemStyle-
VerticalAlign="Top" />
            <asp:DynamicField DataField="StartDate" HeaderText="Start Date" ItemStyle-VerticalAlign="Top" />
            <asp:TemplateField HeaderText="Administrator" SortExpression="Person.LastName" ItemStyle-
VerticalAlign="Top" >
                <ItemTemplate>
                    <asp:Label ID="AdministratorLastNameLabel" runat="server" Text='<%#
Eval("Person.LastName") %>'></asp:Label>,
                    <asp:Label ID="AdministratorFirstNameLabel" runat="server" Text='<%#
Eval("Person.FirstMidName") %>'></asp:Label>
                </ItemTemplate>
            </asp:TemplateField>
        </Columns>
    </asp:GridView>
```

This markup creates an `ObjectDataSource` control that uses the repository class you just created, and a `GridView` control to display the data. The `GridView` control specifies **Edit** and **Delete** commands, but you haven't added code to support them yet.

Several columns use `DynamicField` controls so that you can take advantage of automatic data formatting and validation functionality. For these to work, you will have to call the `EnableDynamicData` method in the `Page_Init` event handler. (`DynamicControl` controls are not used in the `Administrator` field because they don't work with navigation properties.)

The `Vertical-Align="Top"` attributes will become important later when you add a column that has a nested `GridView` control to the grid.

Open the *Departments.aspx.cs* file and add the following `using` statement:

```
using ContosoUniversity.DAL;
```

Then add the following handler for the page's `Init` event:

```
protected void Page_Init(object sender, EventArgs e)
{
    DepartmentsGridView.EnableDynamicData(typeof(Department));
}
```

In the *DAL* folder, create a new class file named *Department.cs* and replace the existing code with the following code:

```
using System;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.DAL
{
    [MetadataType(typeof(DepartmentMetaData))]
    public partial class Department
    {
    }

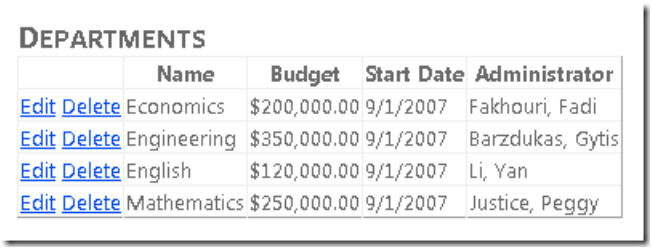
    public class DepartmentMetaData
    {
        [DataType(DataType.Currency)]
        [Range(0, 1000000, ErrorMessage = "Budget must be less than $1,000,000.00")]
        public Decimal Budget { get; set; }

        [DisplayFormat(DataFormatString="{0:d}",ApplyFormatInEditMode=true)]
        public DateTime StartDate { get; set; }

    }
}
```

This code adds metadata to the data model. It specifies that the `Budget` property of the `Department` entity actually represents currency although its data type is `Decimal`, and it specifies that the value must be between 0 and $1,000,000.00. It also specifies that the `StartDate` property should be formatted as a date in the format mm/dd/yyyy.

Run the *Departments.aspx* page.



DEPARTMENTS

	Name	Budget	Start Date	Administrator
Edit Delete	Economics	$200,000.00	9/1/2007	Fakhouri, Fadi
Edit Delete	Engineering	$350,000.00	9/1/2007	Barzdukas, Gytis
Edit Delete	English	$120,000.00	9/1/2007	Li, Yan
Edit Delete	Mathematics	$250,000.00	9/1/2007	Justice, Peggy

Notice that although you did not specify a format string in the *Departments.aspx* page markup for the **Budget** or **Start Date** columns, default currency and date formatting has been applied to them by the `DynamicField` controls, using the metadata that you supplied in the *Department.cs* file.

Adding Insert and Delete Functionality

Open *SchoolRepository.cs*, add the following code in order to create an `Insert` method and a `Delete` method. The code also includes a method named `GenerateDepartmentID` that calculates the next available record key value for use by the `Insert` method. This is required because the database is not configured to calculate this automatically for the `Department` table.

```
public void InsertDepartment(Department department)
{
    try
    {
        department.DepartmentID = GenerateDepartmentID();
        context.Departments.AddObject(department);
        context.SaveChanges();
    }
    catch (Exception ex)
    {
        //Include catch blocks for specific exceptions first,
        //and handle or log the error as appropriate in each.
        //Include a generic catch block like this one last.
        throw ex;
    }
}

public void DeleteDepartment(Department department)
{
    try
    {
        context.Departments.Attach(department);
        context.Departments.DeleteObject(department);
        context.SaveChanges();
    }
    catch (Exception ex)
    {
        //Include catch blocks for specific exceptions first,
        //and handle or log the error as appropriate in each.
        //Include a generic catch block like this one last.
        throw ex;
    }
}

private Int32 GenerateDepartmentID()
{
    Int32 maxDepartmentID = 0;
    var department = (from d in GetDepartments()
                      orderby d.DepartmentID descending
                      select d).FirstOrDefault();
    if (department != null)
    {
        maxDepartmentID = department.DepartmentID + 1;
    }
    return maxDepartmentID;
}
```

The Attach Method

The `DeleteDepartment` method calls the `Attach` method in order to re-establish the link that's maintained in the object context's object state manager between the entity in memory and the database row it represents. This must occur before the method calls the `SaveChanges` method.

The term *object context* refers to the Entity Framework class that derives from the `ObjectContext` class that you use to access your entity sets and entities. In the code for this project, the class is named `SchoolEntities`, and an instance of it is always named `context`. The object context's *object state manager* is a class that derives from the `ObjectStateManager` class. The object contact uses the object state manager to store entity objects and to keep track of whether each one is in sync with its corresponding table row or rows in the database.

When you read an entity, the object context stores it in the object state manager and keeps track of whether that representation of the object is in sync with the database. For example, if you change a property value, a flag is set to indicate that the property you changed is no longer in sync with the database. Then when you call the `SaveChanges` method, the object context knows what to do in the database because the object state manager

knows exactly what's different between the current state of the entity and the state of the database.

However, this process typically does not work in a web application, because the object context instance that reads an entity, along with everything in its object state manager, is disposed after a page is rendered. The object context instance that must apply changes is a new one that's instantiated for postback processing. In the case of the `DeleteDepartment` method, the `ObjectDataSource` control re-creates the original version of the entity for you from values in view state, but this re-created `Department` entity does not exist in the object state manager. If you called the `DeleteObject` method on this re-created entity, the call would fail because the object context does not know whether the entity is in sync with the database. However, calling the `Attach` method re-establishes the same tracking between the re-created entity and the values in the database that was originally done automatically when the entity was read in an earlier instance of the object context.

There are times when you don't want the object context to track entities in the object state manager, and you can set flags to prevent it from doing that. Examples of this are shown in later tutorials in this series.

The SaveChanges Method

This simple repository class illustrates basic principles of how to perform CRUD operations. In this example, the `SaveChanges` method is called immediately after each update. In a production application you might want to call the `SaveChanges` method from a separate method to give you more control over when the database is updated. (At the end of the next tutorial you will find a link to a white paper that discusses the unit of work pattern which is one approach to coordinating related updates.) Notice also that in the example, the `DeleteDepartment` method does not include code for handling concurrency conflicts; code to do that will be added in a later tutorial in this series.

Retrieving Instructor Names to Select When Inserting

Users must be able to select an administrator from a list of instructors in a drop-down list when creating new departments. Therefore, add the following code to *SchoolRepository.cs* to create a method to retrieve the list of instructors using the view that you created earlier:

```
public IEnumerable<InstructorName> GetInstructorNames()
{
    return context.InstructorNames.OrderBy("it.FullName").ToList();
}
```

Creating a Page for Inserting Departments

Create a *DepartmentsAdd.aspx* page that uses the *Site.Master* page, and add the following markup in the `Content` control named `Content2`:

```
<h2>Departments</h2>
    <asp:ObjectDataSource ID="DepartmentsObjectDataSource" runat="server"
        TypeName="ContosoUniversity.DAL.SchoolRepository"
DataObjectTypeName="ContosoUniversity.DAL.Department"
        InsertMethod="InsertDepartment" >
    </asp:ObjectDataSource>
    <asp:DetailsView ID="DepartmentsDetailsView" runat="server"
        DataSourceID="DepartmentsObjectDataSource" AutoGenerateRows="False"
        DefaultMode="Insert" OnItemInserting="DepartmentsDetailsView_ItemInserting">
        <Fields>
            <asp:DynamicField DataField="Name" HeaderText="Name" />
            <asp:DynamicField DataField="Budget" HeaderText="Budget" />
            <asp:DynamicField DataField="StartDate" HeaderText="Start Date" />
            <asp:TemplateField HeaderText="Administrator">
                <InsertItemTemplate>
                    <asp:ObjectDataSource ID="InstructorsObjectDataSource" runat="server"
                        TypeName="ContosoUniversity.DAL.SchoolRepository"
                        DataObjectTypeName="ContosoUniversity.DAL.InstructorName"
                        SelectMethod="GetInstructorNames" >
                    </asp:ObjectDataSource>
                    <asp:DropDownList ID="InstructorsDropDownList" runat="server"
                        DataSourceID="InstructorsObjectDataSource"
                        DataTextField="FullName" DataValueField="PersonID"
OnInit="DepartmentsDropDownList_Init">
                    </asp:DropDownList>
                </InsertItemTemplate>
            </asp:TemplateField>
            <asp:CommandField ShowInsertButton="True" />
        </Fields>
    </asp:DetailsView>
   <asp:ValidationSummary ID="DepartmentsValidationSummary" runat="server"
        ShowSummary="true" DisplayMode="BulletList"  />
```

This markup creates two `ObjectDataSource` controls, one for inserting new `Department` entities and one for retrieving instructor names for the `DropDownList` control that's used for selecting department administrators. The markup creates a `DetailsView` control for entering new departments, and it specifies a handler for the control's `ItemInserting` event so that you can set the `Administrator` foreign key value. At the end is a `ValidationSummary` control to display error messages.

Open *DepartmentsAdd.aspx.cs* and add the following `using` statement:

```
using ContosoUniversity.DAL;
```

Add the following class variable and methods:

```
private DropDownList administratorsDropDownList;

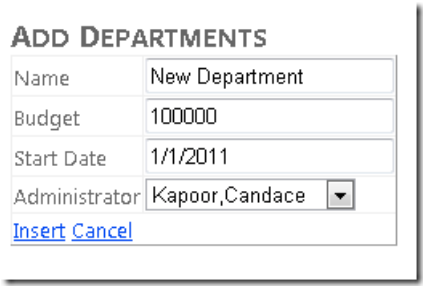
protected void Page_Init(object sender, EventArgs e)
{
    DepartmentsDetailsView.EnableDynamicData(typeof(Department));
}

protected void DepartmentsDropDownList_Init(object sender, EventArgs e)
{
    administratorsDropDownList = sender as DropDownList;
}

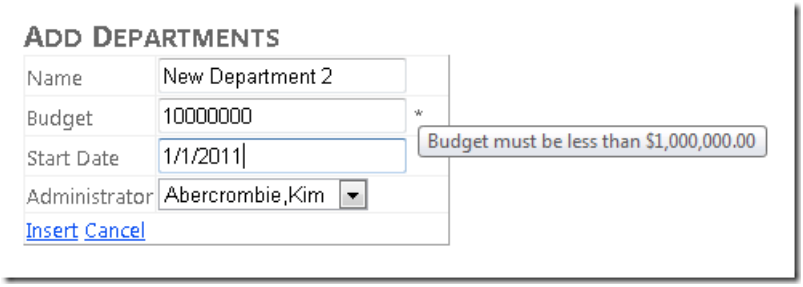
protected void DepartmentsDetailsView_ItemInserting(object sender, DetailsViewInsertEventArgs e)
{
    e.Values["Administrator"] = administratorsDropDownList.SelectedValue;
}
```

The `Page_Init` method enables Dynamic Data functionality. The handler for the `DropDownList` control's `Init` event saves a reference to the control, and the handler for the `DetailsView` control's `Inserting` event uses that reference to get the `PersonID` value of the selected instructor and update the `Administrator` foreign key property of the `Department` entity.

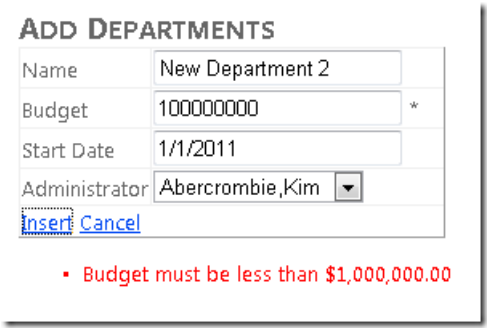
Run the page, add information for a new department, and then click the **Insert** link.



Enter values for another new department. Enter a number greater than 1,000,000.00 in the **Budget** field and tab to the next field. An asterisk appears in the field, and if you hold the mouse pointer over it, you can see the error message that you entered in the metadata for that field.



Click **Insert**, and you see the error message displayed by the `ValidationSummary` control at the bottom of the page.



Next, close the browser and open the *Departments.aspx* page. Add delete capability to the *Departments.aspx* page by adding a `DeleteMethod` attribute to the `ObjectDataSource` control, and a `DataKeyNames` attribute to the `GridView` control. The opening tags for these controls will now resemble the following example:

```
<asp:ObjectDataSource ID="DepartmentsObjectDataSource" runat="server"
        TypeName="ContosoUniversity.DAL.SchoolRepository"
        DataObjectTypeName="ContosoUniversity.DAL.Department"
        SelectMethod="GetDepartments"
        DeleteMethod="DeleteDepartment" >

    <asp:GridView ID="DepartmentsGridView" runat="server" AutoGenerateColumns="False"
        DataSourceID="DepartmentsObjectDataSource" DataKeyNames="DepartmentID" >
```

Run the page.

DEPARTMENTS

	Name	Budget	Start Date	Administrator
Edit Delete	Economics	$200,000.00	9/1/2007	Fakhouri, Fadi
Edit Delete	Engineering	$350,000.00	9/1/2007	Barzdukas, Gytis
Edit Delete	English	$120,000.00	9/1/2007	Li, Yan
Edit Delete	Mathematics	$250,000.00	9/1/2007	Justice, Peggy
Edit Delete	New Department	$100,000.00	1/1/2011	Kapoor, Candace

Delete the department you added when you ran the *DepartmentsAdd.aspx* page.

Adding Update Functionality

Open *SchoolRepository.cs* and add the following `Update` method:

```
public void UpdateDepartment(Department department, Department origDepartment)
{
    try
    {
        context.Departments.Attach(origDepartment);
        context.ApplyCurrentValues("Departments", department);
        context.SaveChanges();
    }
    catch (Exception ex)
    {
        //Include catch blocks for specific exceptions first,
        //and handle or log the error as appropriate in each.
        //Include a generic catch block like this one last.
        throw ex;
    }
}
```

When you click **Update** in the *Departments.aspx* page, the `ObjectDataSource` control creates two `Department` entities to pass to the `UpdateDepartment` method. One contains the original values that have been stored in view state, and the other contains the new values that were entered in the `GridView` control. The code in the `UpdateDepartment` method passes the `Department` entity that has the original values to the `Attach` method in order to establish the tracking between the entity and what's in the database. Then the code passes the `Department` entity that has the new values to the `ApplyCurrentValues` method. The object context compares the old and new values. If a new value is different from an old value, the object context changes the property value. The `SaveChanges` method then updates only the changed columns in the database. (However, if the update function for this entity were mapped to a stored procedure, the entire row would be updated regardless of which columns were changed.)

Open the *Departments.aspx* file and add the following attributes to the `DepartmentsObjectDataSource` control:

- `UpdateMethod="UpdateDepartment"`
- `ConflictDetection="CompareAllValues"`
  This causes old values to be stored in view state so that they can be compared with the new values in the `Update` method.
- `OldValuesParameterFormatString="orig{0}"`
  This informs the control that the name of the original values parameter is `origDepartment` .

The markup for the opening tag of the `ObjectDataSource` control now resembles the following example:

```
<asp:ObjectDataSource ID="DepartmentsObjectDataSource" runat="server"
        TypeName="ContosoUniversity.DAL.SchoolRepository"
        DataObjectTypeName="ContosoUniversity.DAL.Department"
        SelectMethod="GetDepartments" DeleteMethod="DeleteDepartment"
        UpdateMethod="UpdateDepartment"
        ConflictDetection="CompareAllValues"
        OldValuesParameterFormatString="orig{0}" >
```

Add an `OnRowUpdating="DepartmentsGridView_RowUpdating"` attribute to the `GridView` control. You will use this to set the `Administrator` property value based on the row the user selects in a drop-down list. The `GridView` opening tag now resembles the following example:

```
<asp:GridView ID="DepartmentsGridView" runat="server" AutoGenerateColumns="False"
        DataSourceID="DepartmentsObjectDataSource" DataKeyNames="DepartmentID"
        OnRowUpdating="DepartmentsGridView_RowUpdating">
```

Add an `EditItemTemplate` control for the `Administrator` column to the `GridView` control, immediately after the `ItemTemplate` control for that column:

```
<EditItemTemplate>
                    <asp:ObjectDataSource ID="InstructorsObjectDataSource" runat="server"
DataObjectTypeName="ContosoUniversity.DAL.InstructorName"
                        SelectMethod="GetInstructorNames" TypeName="ContosoUniversity.DAL.SchoolRepository">
                    </asp:ObjectDataSource>
                    <asp:DropDownList ID="InstructorsDropDownList" runat="server"
DataSourceID="InstructorsObjectDataSource"
                        SelectedValue='<%# Eval("Administrator")  %>'
                        DataTextField="FullName" DataValueField="PersonID"
OnInit="DepartmentsDropDownList_Init" >
                    </asp:DropDownList>
                </EditItemTemplate>
```

This `EditItemTemplate` control is similar to the `InsertItemTemplate` control in the *DepartmentsAdd.aspx* page. The difference is that the initial value of the control is set using the `SelectedValue` attribute.

Before the `GridView` control, add a `ValidationSummary` control as you did in the *DepartmentsAdd.aspx* page.

```
<asp:ValidationSummary ID="DepartmentsValidationSummary" runat="server"
        ShowSummary="true" DisplayMode="BulletList"  />
```

Open *Departments.aspx.cs* and immediately after the partial-class declaration, add the following code to create a private field to reference the `DropDownList` control:

```
private DropDownList administratorsDropDownList;
```

Then add handlers for the `DropDownList` control's `Init` event and the `GridView` control's `RowUpdating` event:

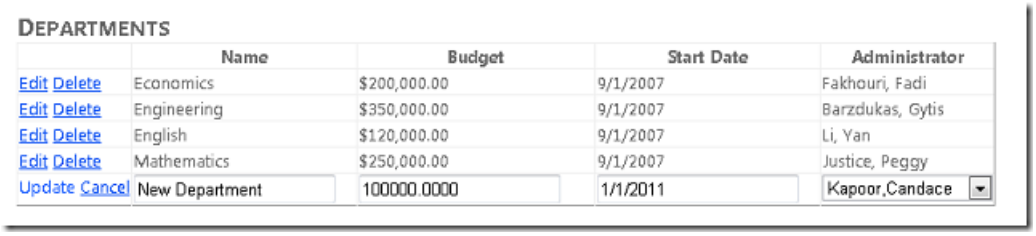
```
protected void DepartmentsDropDownList_Init(object sender, EventArgs e)
{
    administratorsDropDownList = sender as DropDownList;
}

protected void DepartmentsGridView_RowUpdating(object sender, GridViewUpdateEventArgs e)
{
    e.NewValues["Administrator"] = administratorsDropDownList.SelectedValue;
}
```

The handler for the `Init` event saves a reference to the `DropDownList` control in the class field. The handler for the `RowUpdating` event uses the reference to get the value the user entered and apply it to the `Administrator` property of the `Department` entity.

Use the *DepartmentsAdd.aspx* page to add a new department, then run the *Departments.aspx* page and click **Edit** on the row that you added.

> **NOTE**
>
> You will not be able to edit rows that you did not add (that is, that were already in the database), because of invalid data in the database; the administrators for the rows that were created with the database are students. If you try to edit one of them, you will get an error page that reports an error like
>
> ```
> 'InstructorsDropDownList' has a SelectedValue which is invalid because it does not exist in the list of items.
> ```

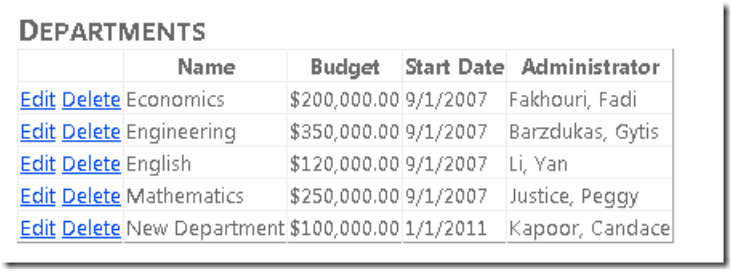


DEPARTMENTS

	Name	Budget	Start Date	Administrator
Edit Delete	Economics	$200,000.00	9/1/2007	Fakhouri, Fadi
Edit Delete	Engineering	$350,000.00	9/1/2007	Barzdukas, Gytis
Edit Delete	English	$120,000.00	9/1/2007	Li, Yan
Edit Delete	Mathematics	$250,000.00	9/1/2007	Justice, Peggy
Update Cancel	New Department	100000.0000	1/1/2011	Kapoor,Candace

If you enter an invalid **Budget** amount and then click **Update**, you see the same asterisk and error message that you saw in the *Departments.aspx* page.

Change a field value or select a different administrator and click **Update**. The change is displayed.



DEPARTMENTS

	Name	Budget	Start Date	Administrator
Edit Delete	Economics	$200,000.00	9/1/2007	Fakhouri, Fadi
Edit Delete	Engineering	$350,000.00	9/1/2007	Barzdukas, Gytis
Edit Delete	English	$120,000.00	9/1/2007	Li, Yan
Edit Delete	Mathematics	$250,000.00	9/1/2007	Justice, Peggy
Edit Delete	New Department	$100,000.00	1/1/2011	Kapoor, Candace

This completes the introduction to using the `ObjectDataSource` control for basic CRUD (create, read, update, delete) operations with the Entity Framework. You've built a simple n-tier application, but the business-logic layer is still tightly coupled to the data-access layer, which complicates automated unit testing. In the following tutorial you'll see how to implement the repository pattern to facilitate unit testing.

NEXT

Using the Entity Framework 4.0 and the ObjectDataSource Control, Part 2: Adding a Business Logic Layer and Unit Tests

2/20/2020 • 14 minutes to read • Edit Online

by Tom Dykstra

> This tutorial series builds on the Contoso University web application that is created by the Getting Started with the Entity Framework 4.0 tutorial series. If you didn't complete the earlier tutorials, as a starting point for this tutorial you can download the application that you would have created. You can also download the application that is created by the complete tutorial series. If you have questions about the tutorials, you can post them to the ASP.NET Entity Framework forum.

In the previous tutorial you created an n-tier web application using the Entity Framework and the `ObjectDataSource` control. This tutorial shows how to add business logic while keeping the business-logic layer (BLL) and the data-access layer (DAL) separate, and it shows how to create automated unit tests for the BLL.

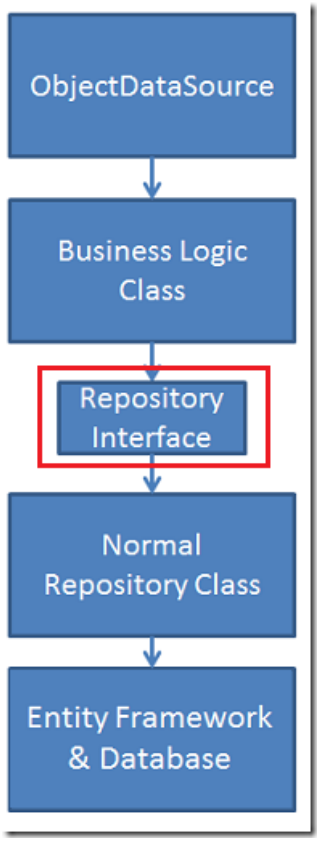
In this tutorial you'll complete the following tasks:

- Create a repository interface that declares the data-access methods you need.
- Implement the repository interface in the repository class.
- Create a business-logic class that calls the repository class to perform data-access functions.
- Connect the `ObjectDataSource` control to the business-logic class instead of to the repository class.
- Create a unit-test project and a repository class that uses in-memory collections for its data store.
- Create a unit test for business logic that you want to add to the business-logic class, then run the test and see it fail.
- Implement the business logic in the business-logic class, then re-run the unit test and see it pass.

You'll work with the *Departments.aspx* and *DepartmentsAdd.aspx* pages that you created in the previous tutorial.

Creating a Repository Interface

You'll begin by creating the repository interface.



In the *DAL* folder, create a new class file, name it *ISchoolRepository.cs*, and replace the existing code with the following code:

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.DAL
{
    public interface ISchoolRepository : IDisposable
    {
        IEnumerable<Department> GetDepartments();
        void InsertDepartment(Department department);
        void DeleteDepartment(Department department);
        void UpdateDepartment(Department department, Department origDepartment);
        IEnumerable<InstructorName> GetInstructorNames();
    }
}
```

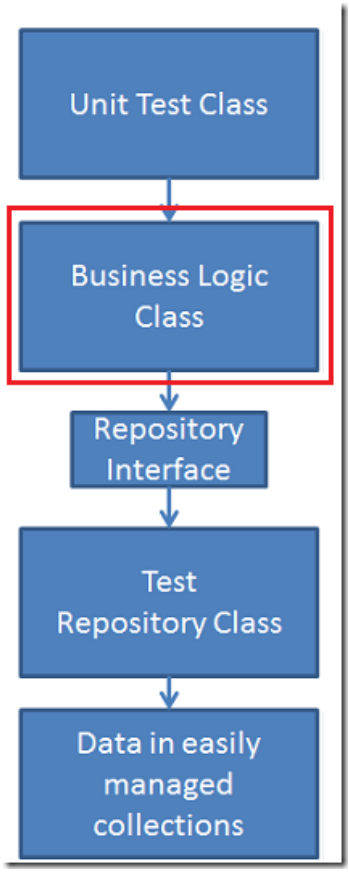
The interface defines one method for each of the CRUD (create, read, update, delete) methods that you created in the repository class.

In the `SchoolRepository` class in *SchoolRepository.cs*, indicate that this class implements the `ISchoolRepository` interface:

```
public class SchoolRepository : IDisposable, ISchoolRepository
```

Creating a Business-Logic Class

Next, you'll create the business-logic class. You do this so that you can add business logic that will be executed by the `ObjectDataSource` control, although you will not do that yet. For now, the new business-logic class will only perform the same CRUD operations that the repository does.



Create a new folder and name it *BLL*. (In a real-world application, the business-logic layer would typically be implemented as a class library — a separate project — but to keep this tutorial simple, BLL classes will be kept in a project folder.)

In the *BLL* folder, create a new class file, name it *SchoolBL.cs*, and replace the existing code with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using ContosoUniversity.DAL;

namespace ContosoUniversity.BLL
{
    public class SchoolBL : IDisposable
    {
        private ISchoolRepository schoolRepository;

        public SchoolBL()
        {
            this.schoolRepository = new SchoolRepository();
        }

        public SchoolBL(ISchoolRepository schoolRepository)
        {
            this.schoolRepository = schoolRepository;
        }

        public IEnumerable<Department> GetDepartments()
        {
            return schoolRepository.GetDepartments();
        }

        public void InsertDepartment(Department department)
        {
            try
            {
                schoolRepository.InsertDepartment(department);
```

```
            schoolRepository.InsertDepartment(department);
        }
        catch (Exception ex)
        {
            //Include catch blocks for specific exceptions first,
            //and handle or log the error as appropriate in each.
            //Include a generic catch block like this one last.
            throw ex;
        }
    }

    public void DeleteDepartment(Department department)
    {
        try
        {
            schoolRepository.DeleteDepartment(department);
        }
        catch (Exception ex)
        {
            //Include catch blocks for specific exceptions first,
            //and handle or log the error as appropriate in each.
            //Include a generic catch block like this one last.
            throw ex;
        }
    }

    public void UpdateDepartment(Department department, Department origDepartment)
    {
        try
        {
            schoolRepository.UpdateDepartment(department, origDepartment);
        }
        catch (Exception ex)
        {
            //Include catch blocks for specific exceptions first,
            //and handle or log the error as appropriate in each.
            //Include a generic catch block like this one last.
            throw ex;
        }

    }

    public IEnumerable<InstructorName> GetInstructorNames()
    {
        return schoolRepository.GetInstructorNames();
    }

    private bool disposedValue = false;

    protected virtual void Dispose(bool disposing)
    {
        if (!this.disposedValue)
        {
            if (disposing)
            {
                schoolRepository.Dispose();
            }
        }
        this.disposedValue = true;
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

  }
}
```

This code creates the same CRUD methods you saw earlier in the repository class, but instead of accessing the Entity Framework methods directly, it calls the repository class methods.

The class variable that holds a reference to the repository class is defined as an interface type, and the code that instantiates the repository class is contained in two constructors. The parameterless constructor will be used by the `ObjectDataSource` control. It creates an instance of the `SchoolRepository` class that you created earlier. The other constructor allows whatever code that instantiates the business-logic class to pass in any object that implements the repository interface.

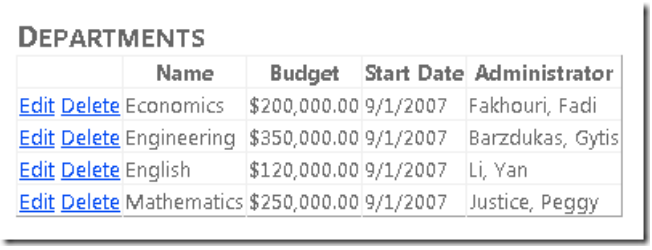
The CRUD methods that call the repository class and the two constructors make it possible to use the business-logic class with whatever back-end data store you choose. The business-logic class does not need to be aware of how the class that it's calling persists the data. (This is often called *persistence ignorance*.) This facilitates unit testing, because you can connect the business-logic class to a repository implementation that uses something as simple as in-memory `List` collections to store data.

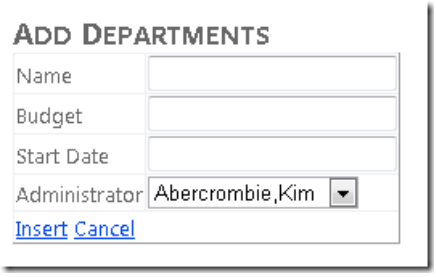
> **NOTE**
> Technically, the entity objects are still not persistence-ignorant, because they're instantiated from classes that inherit from the Entity Framework's `EntityObject` class. For complete persistence ignorance, you can use *plain old CLR objects*, or *POCOs*, in place of objects that inherit from the `EntityObject` class. Using POCOs is beyond the scope of this tutorial. For more information, see Testability and Entity Framework 4.0 on the MSDN website.)

Now you can connect the `ObjectDataSource` controls to the business-logic class instead of to the repository and verify that everything works as it did before.

In *Departments.aspx* and *DepartmentsAdd.aspx*, change each occurrence of `TypeName="ContosoUniversity.DAL.SchoolRepository"` to `TypeName="ContosoUniversity.BLL.SchoolBL"`. (There are four instances in all.)

Run the *Departments.aspx* and *DepartmentsAdd.aspx* pages to verify that they still work as they did before.





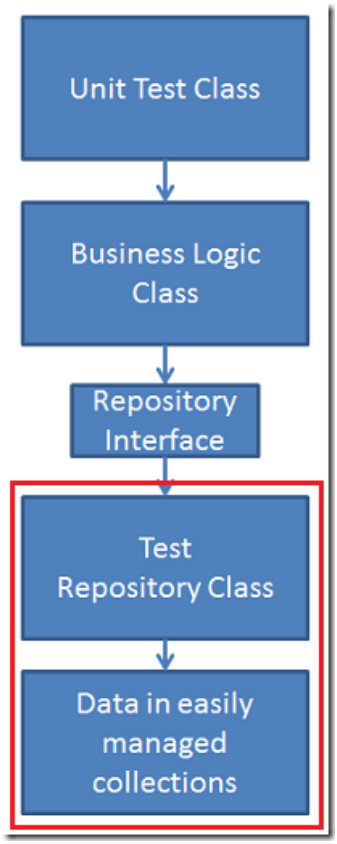
Creating a Unit-Test Project and Repository Implementation

Add a new project to the solution using the **Test Project** template, and name it `ContosoUniversity.Tests`.

In the test project, add a reference to `System.Data.Entity` and add a project reference to the `ContosoUniversity` project.

You can now create the repository class that you'll use with unit tests. The data store for this repository will be

within the class.



In the test project, create a new class file, name it *MockSchoolRepository.cs*, and replace the existing code with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ContosoUniversity.DAL;
using ContosoUniversity.BLL;

namespace ContosoUniversity.Tests
{
    class MockSchoolRepository : ISchoolRepository, IDisposable
    {
        List<Department> departments = new List<Department>();
        List<InstructorName> instructors = new List<InstructorName>();

        public IEnumerable<Department> GetDepartments()
        {
            return departments;
        }

        public void InsertDepartment(Department department)
        {
            departments.Add(department);
        }

        public void DeleteDepartment(Department department)
        {
            departments.Remove(department);
        }

        public void UpdateDepartment(Department department, Department origDepartment)
        {
            departments.Remove(origDepartment);
            departments.Add(department);
        }

        public IEnumerable<InstructorName> GetInstructorNames()
        {
            return instructors;
        }

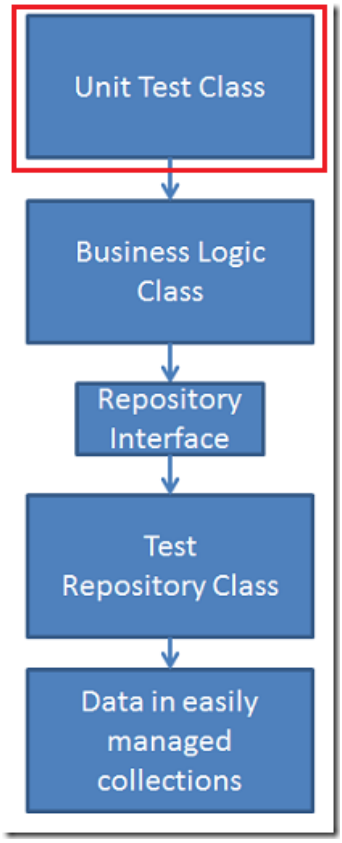
        public void Dispose()
        {

        }
    }
}
```

This repository class has the same CRUD methods as the one that accesses the Entity Framework directly, but they work with `List` collections in memory instead of with a database. This makes it easier for a test class to set up and validate unit tests for the business-logic class.

Creating Unit Tests

The **Test** project template created a stub unit test class for you, and your next task is to modify this class by adding unit test methods to it for business logic that you want to add to the business-logic class.



At Contoso University, any individual instructor can only be the administrator of a single department, and you need to add business logic to enforce this rule. You will start by adding tests and running the tests to see them fail. You'll then add the code and rerun the tests, expecting to see them pass.

Open the *UnitTest1.cs* file and add `using` statements for the business logic and data-access layers that you created in the ContosoUniversity project:

```
using ContosoUniversity.BLL;
using ContosoUniversity.DAL;
```

Replace the `TestMethod1` method with the following methods:

```
private SchoolBL CreateSchoolBL()
{
    var schoolRepository = new MockSchoolRepository();
    var schoolBL = new SchoolBL(schoolRepository);
    schoolBL.InsertDepartment(new Department() { Name = "First Department", DepartmentID = 0, Administrator =
1, Person = new Instructor () { FirstMidName = "Admin", LastName = "One" } });
    schoolBL.InsertDepartment(new Department() { Name = "Second Department", DepartmentID = 1, Administrator
= 2, Person = new Instructor() { FirstMidName = "Admin", LastName = "Two" } });
    schoolBL.InsertDepartment(new Department() { Name = "Third Department", DepartmentID = 2, Administrator =
3, Person = new Instructor() { FirstMidName = "Admin", LastName = "Three" } });
    return schoolBL;
}

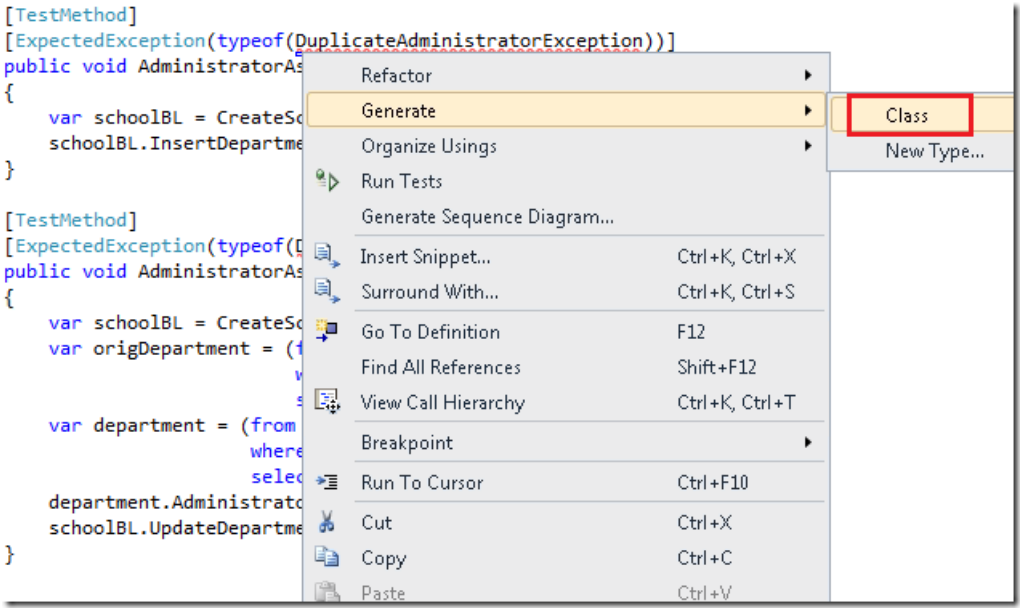
[TestMethod]
[ExpectedException(typeof(DuplicateAdministratorException))]
public void AdministratorAssignmentRestrictionOnInsert()
{
    var schoolBL = CreateSchoolBL();
    schoolBL.InsertDepartment(new Department() { Name = "Fourth Department", DepartmentID = 3, Administrator
= 2, Person = new Instructor() { FirstMidName = "Admin", LastName = "Two" } });
}

[TestMethod]
[ExpectedException(typeof(DuplicateAdministratorException))]
public void AdministratorAssignmentRestrictionOnUpdate()
{
    var schoolBL = CreateSchoolBL();
    var origDepartment = (from d in schoolBL.GetDepartments()
                          where d.Name == "Second Department"
                          select d).First();
    var department = (from d in schoolBL.GetDepartments()
                          where d.Name == "Second Department"
                          select d).First();
    department.Administrator = 1;
    schoolBL.UpdateDepartment(department, origDepartment);
}
```

The `CreateSchoolBL` method creates an instance of the repository class that you created for the unit test project, which it then passes to a new instance of the business-logic class. The method then uses the business-logic class to insert three departments that you can use in test methods.

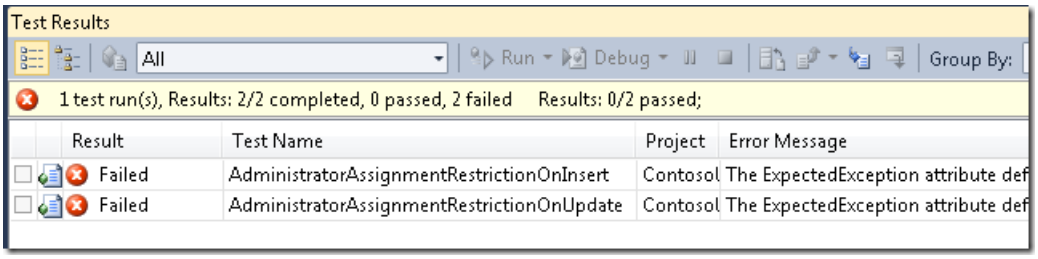
The test methods verify that the business-logic class throws an exception if someone tries to insert a new department with the same administrator as an existing department, or if someone tries to update a department's administrator by setting it to the ID of a person who is already the administrator of another department.

You haven't created the exception class yet, so this code will not compile. To get it to compile, right-click `DuplicateAdministratorException` and select **Generate**, and then **Class**.



This creates a class in the test project which you can delete after you've created the exception class in the main project. and implemented the business logic.

Run the test project. As expected, the tests fail.



Adding Business Logic to Make a Test Pass

Next, you'll implement the business logic that makes it impossible to set as the administrator of a department someone who is already administrator of another department. You'll throw an exception from the business-logic layer, and then catch it in the presentation layer if a user edits a department and clicks **Update** after selecting someone who is already an administrator. (You could also remove instructors from the drop-down list who are already administrators before you render the page, but the purpose here is to work with the business-logic layer.)

Start by creating the exception class that you'll throw when a user tries to make an instructor the administrator of more than one department. In the main project, create a new class file in the *BLL* folder, name it *DuplicateAdministratorException.cs*, and replace the existing code with the following code:

```
using System;

namespace ContosoUniversity.BLL
{
    public class DuplicateAdministratorException : Exception
    {
        public DuplicateAdministratorException(string message)
            : base(message)
        {
        }
    }
}
```

Now delete the temporary *DuplicateAdministratorException.cs* file that you created in the test project earlier in order to be able to compile.

In the main project, open the *SchoolBL.cs* file and add the following method that contains the validation logic. (The code refers to a method that you'll create later.)

```
private void ValidateOneAdministratorAssignmentPerInstructor(Department department)
{
    if (department.Administrator != null)
    {
        var duplicateDepartment =
schoolRepository.GetDepartmentsByAdministrator(department.Administrator.GetValueOrDefault()).FirstOrDefault()
;
        if (duplicateDepartment != null && duplicateDepartment.DepartmentID != department.DepartmentID)
        {
            throw new DuplicateAdministratorException(String.Format(
                "Instructor {0} {1} is already administrator of the {2} department.",
                duplicateDepartment.Person.FirstMidName,
                duplicateDepartment.Person.LastName,
                duplicateDepartment.Name));
        }
    }
}
```

You'll call this method when you're inserting or updating `Department` entities in order to check whether another department already has the same administrator.

The code calls a method to search the database for a `Department` entity that has the same `Administrator` property value as the entity being inserted or updated. If one is found, the code throws an exception. No validation check is required if the entity being inserted or updated has no `Administrator` value, and no exception is thrown if the method is called during an update and the `Department` entity found matches the `Department` entity being updated.

Call the new method from the `Insert` and `Update` methods:

```
public void InsertDepartment(Department department)
{
    ValidateOneAdministratorAssignmentPerInstructor(department);
    try
    ...

public void UpdateDepartment(Department department, Department origDepartment)
{
    ValidateOneAdministratorAssignmentPerInstructor(department);
    try
    ...
```

In *ISchoolRepository.cs*, add the following declaration for the new data-access method:

```
IEnumerable<Department> GetDepartmentsByAdministrator(Int32 administrator);
```

In *SchoolRepository.cs*, add the following `using` statement:

```
using System.Data.Objects;
```

In *SchoolRepository.cs*, add the following new data-access method:

```
public IEnumerable<Department> GetDepartmentsByAdministrator(Int32 administrator)
{
    return new ObjectQuery<Department>("SELECT VALUE d FROM Departments as d", context,
MergeOption.NoTracking).Include("Person").Where(d => d.Administrator == administrator).ToList();
}
```

This code retrieves `Department` entities that have a specified administrator. Only one department should be found (if any). However, because no constraint is built into the database, the return type is a collection in case multiple departments are found.

By default, when the object context retrieves entities from the database, it keeps track of them in its object state manager. The `MergeOption.NoTracking` parameter specifies that this tracking will not be done for this query. This is necessary because the query might return the exact entity that you're trying to update, and then you would not be able to attach that entity. For example, if you edit the History department in the *Departments.aspx* page and leave the administrator unchanged, this query will return the History department. If `NoTracking` is not set, the object context would already have the History department entity in its object state manager. Then when you attach the History department entity that's re-created from view state, the object context would throw an exception that says

```
"An object with the same key already exists in the ObjectStateManager. The ObjectStateManager cannot track
multiple objects with the same key"
```

.

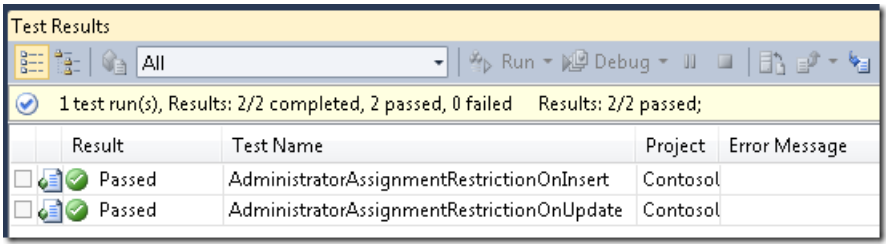
(As an alternative to specifying `MergeOption.NoTracking`, you could create a new object context just for this query. Because the new object context would have its own object state manager, there would be no conflict when you call the `Attach` method. The new object context would share metadata and database connection with the original object context, so the performance penalty of this alternate approach would be minimal. The approach shown here, however, introduces the `NoTracking` option, which you'll find useful in other contexts. The `NoTracking` option is discussed further in a later tutorial in this series.)

In the test project, add the new data-access method to *MockSchoolRepository.cs*:

```
public IEnumerable<Department> GetDepartmentsByAdministrator(Int32 administrator)
{
    return (from d in departments
            where d.Administrator == administrator
            select d);
}
```

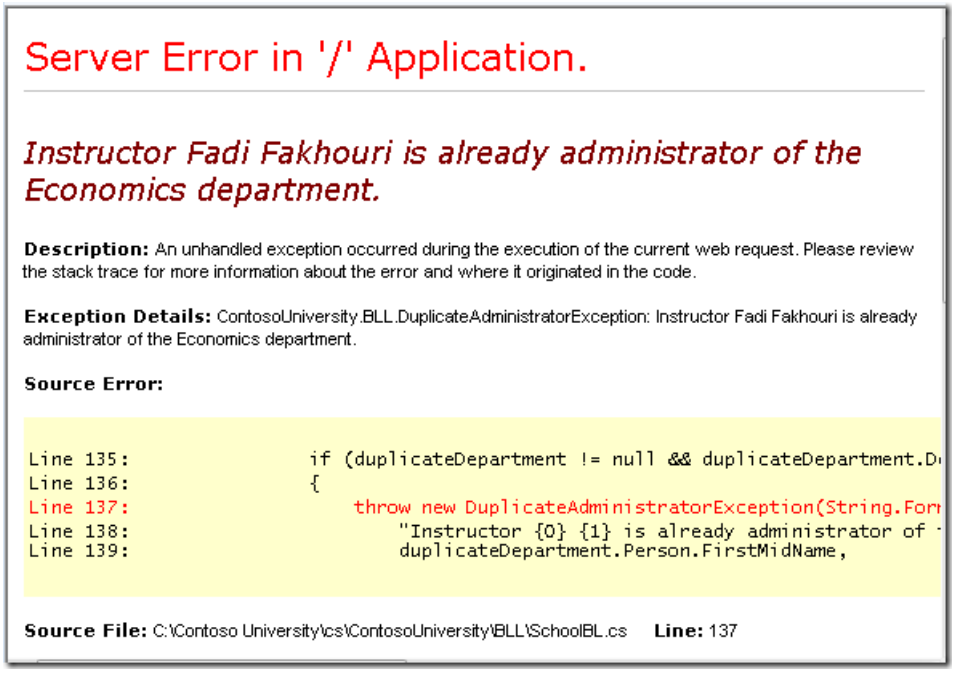
This code uses LINQ to perform the same data selection that the `ContosoUniversity` project repository uses LINQ to Entities for.

Run the test project again. This time the tests pass.



Handling ObjectDataSource Exceptions

In the `ContosoUniversity` project, run the *Departments.aspx* page and try to change the administrator for a department to someone who is already administrator for another department. (Remember that you can only edit departments that you added during this tutorial, because the database comes preloaded with invalid data.) You get the following server error page:



You don't want users to see this kind of error page, so you need to add error-handling code. Open *Departments.aspx* and specify a handler for the `OnUpdated` event of the `DepartmentsObjectDataSource`. The `ObjectDataSource` opening tag now resembles the following example.

```
<asp:ObjectDataSource ID="DepartmentsObjectDataSource" runat="server"
        TypeName="ContosoUniversity.BLL.SchoolBL"
        DataObjectTypeName="ContosoUniversity.DAL.Department"
        SelectMethod="GetDepartments"
        DeleteMethod="DeleteDepartment"
        UpdateMethod="UpdateDepartment"
        ConflictDetection="CompareAllValues"
        OldValuesParameterFormatString="orig{0}"
        OnUpdated="DepartmentsObjectDataSource_Updated" >
```

In *Departments.aspx.cs*, add the following `using` statement:

```
using ContosoUniversity.BLL;
```

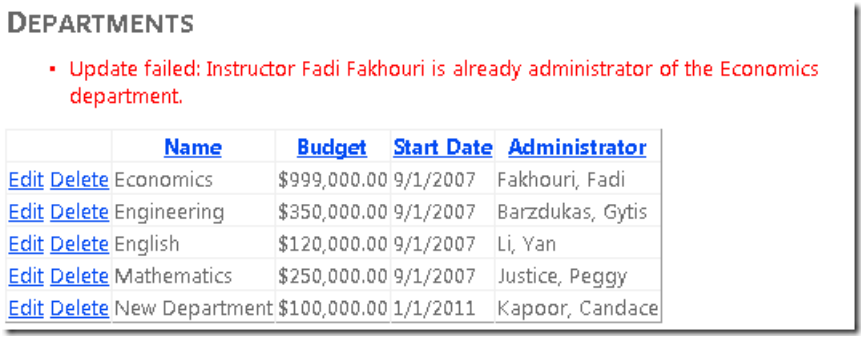
Add the following handler for the `Updated` event:

```
protected void DepartmentsObjectDataSource_Updated(object sender, ObjectDataSourceStatusEventArgs e)
{
    if (e.Exception != null)
    {
        if (e.Exception.InnerException is DuplicateAdministratorException)
        {
            var duplicateAdministratorValidator = new CustomValidator();
            duplicateAdministratorValidator.IsValid = false;
            duplicateAdministratorValidator.ErrorMessage = "Update failed: " +
e.Exception.InnerException.Message;
            Page.Validators.Add(duplicateAdministratorValidator);
            e.ExceptionHandled = true;
        }
    }
}
```

If the `ObjectDataSource` control catches an exception when it tries to perform the update, it passes the exception in the event argument (`e`) to this handler. The code in the handler checks to see if the exception is the duplicate

administrator exception. If it is, the code creates a validator control that contains an error message for the `ValidationSummary` control to display.

Run the page and attempt to make someone the administrator of two departments again. This time the `ValidationSummary` control displays an error message.



Make similar changes to the *DepartmentsAdd.aspx* page. In *DepartmentsAdd.aspx*, specify a handler for the `OnInserted` event of the `DepartmentsObjectDataSource`. The resulting markup will resemble the following example.

```
<asp:ObjectDataSource ID="DepartmentsObjectDataSource" runat="server"
        TypeName="ContosoUniversity.BLL.SchoolBL" DataObjectTypeName="ContosoUniversity.DAL.Department"
        InsertMethod="InsertDepartment"
        OnInserted="DepartmentsObjectDataSource_Inserted">
```

In *DepartmentsAdd.aspx.cs*, add the same `using` statement:

```
using ContosoUniversity.BLL;
```

Add the following event handler:

```
protected void DepartmentsObjectDataSource_Inserted(object sender, ObjectDataSourceStatusEventArgs e)
{
    if (e.Exception != null)
    {
        if (e.Exception.InnerException is DuplicateAdministratorException)
        {
            var duplicateAdministratorValidator = new CustomValidator();
            duplicateAdministratorValidator.IsValid = false;
            duplicateAdministratorValidator.ErrorMessage = "Insert failed: " +
e.Exception.InnerException.Message;
            Page.Validators.Add(duplicateAdministratorValidator);
            e.ExceptionHandled = true;
        }
    }
}
```

You can now test the *DepartmentsAdd.aspx.cs* page to verify that it also correctly handles attempts to make one person the administrator of more than one department.

This completes the introduction to implementing the repository pattern for using the `ObjectDataSource` control with the Entity Framework. For more information about the repository pattern and testability, see the MSDN whitepaper Testability and Entity Framework 4.0.

In the following tutorial you'll see how to add sorting and filtering functionality to the application.

PREVIOUS NEXT

Using the Entity Framework 4.0 and the ObjectDataSource Control, Part 3: Sorting and Filtering

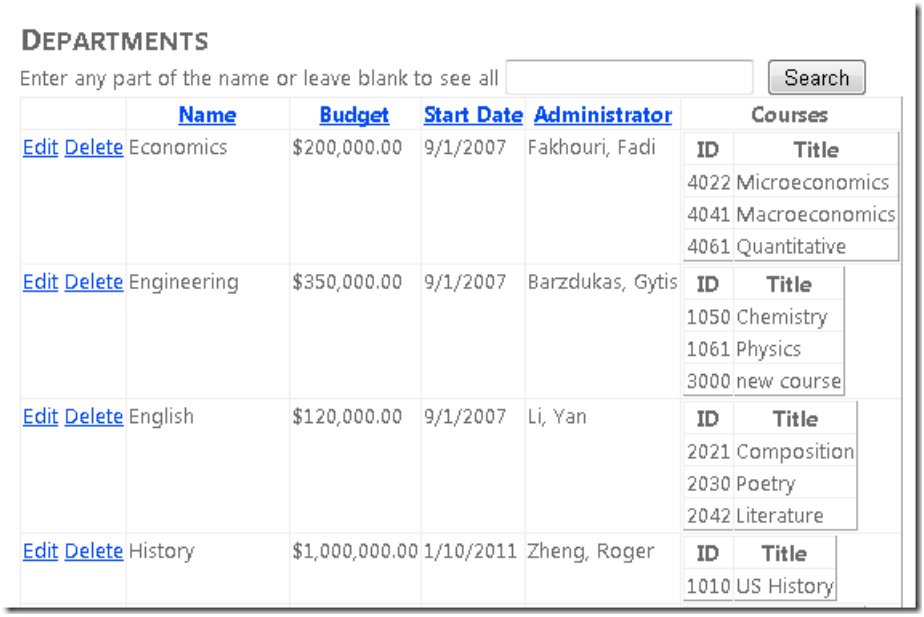
2/20/2020 • 6 minutes to read • Edit Online

by Tom Dykstra

> This tutorial series builds on the Contoso University web application that is created by the Getting Started with the Entity Framework 4.0 tutorial series. If you didn't complete the earlier tutorials, as a starting point for this tutorial you can download the application that you would have created. You can also download the application that is created by the complete tutorial series. If you have questions about the tutorials, you can post them to the ASP.NET Entity Framework forum.

In the previous tutorial you implemented the repository pattern in an n-tier web application that uses the Entity Framework and the `ObjectDataSource` control. This tutorial shows how to do sorting and filtering and handle master-detail scenarios. You'll add the following enhancements to the *Departments.aspx* page:

- A text box to allow users to select departments by name.
- A list of courses for each department that's shown in the grid.
- The ability to sort by clicking column headings.



Adding the Ability to Sort GridView Columns

Open the *Departments.aspx* page and add a `SortParameterName="sortExpression"` attribute to the `ObjectDataSource` control named `DepartmentsObjectDataSource`. (Later you'll create a `GetDepartments` method that takes a parameter named `sortExpression`.) The markup for the opening tag of the control now resembles the following example.

```
<asp:ObjectDataSource ID="DepartmentsObjectDataSource" runat="server"
        TypeName="ContosoUniversity.BLL.SchoolBL" DataObjectTypeName="ContosoUniversity.DAL.Department"
        SelectMethod="GetDepartments" DeleteMethod="DeleteDepartment" UpdateMethod="UpdateDepartment"
        ConflictDetection="CompareAllValues" OldValuesParameterFormatString="orig{0}"
        OnUpdated="DepartmentsObjectDataSource_Updated" SortParameterName="sortExpression" >
```

Add the `AllowSorting="true"` attribute to the opening tag of the `GridView` control. The markup for the opening tag of the control now resembles the following example.

```
<asp:GridView ID="DepartmentsGridView" runat="server" AutoGenerateColumns="False"
        DataSourceID="DepartmentsObjectDataSource" DataKeyNames="DepartmentID"
        OnRowUpdating="DepartmentsGridView_RowUpdating"
        AllowSorting="true" >
```

In *Departments.aspx.cs*, set the default sort order by calling the `GridView` control's `Sort` method from the `Page_Load` method:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        DepartmentsGridView.Sort("Name", SortDirection.Ascending);
    }
}
```

You can add code that sorts or filters in either the business logic class or the repository class. If you do it in the business logic class, the sorting or filtering work will be done after the data is retrieved from the database, because the business logic class is working with an `IEnumerable` object returned by the repository. If you add sorting and filtering code in the repository class and you do it before a LINQ expression or object query has been converted to an `IEnumerable` object, your commands will be passed through to the database for processing, which is typically more efficient. In this tutorial you'll implement sorting and filtering in a way that causes the processing to be done by the database — that is, in the repository.

To add sorting capability, you must add a new method to the repository interface and repository classes as well as to the business logic class. In the *ISchoolRepository.cs* file, add a new `GetDepartments` method that takes a `sortExpression` parameter that will be used to sort the list of departments that's returned:

```
IEnumerable<Department> GetDepartments(string sortExpression);
```

The `sortExpression` parameter will specify the column to sort on and the sort direction.

Add code for the new method to the *SchoolRepository.cs* file:

```
public IEnumerable<Department> GetDepartments(string sortExpression)
{
    if (String.IsNullOrWhiteSpace(sortExpression))
    {
        sortExpression = "Name";
    }
    return context.Departments.Include("Person").OrderBy("it." + sortExpression).ToList();
}
```

Change the existing parameterless `GetDepartments` method to call the new method:

```
public IEnumerable<Department> GetDepartments()
{
    return GetDepartments("");
}
```

In the test project, add the following new method to *MockSchoolRepository.cs*:

```
public IEnumerable<Department> GetDepartments(string sortExpression)
{
    return departments;
}
```

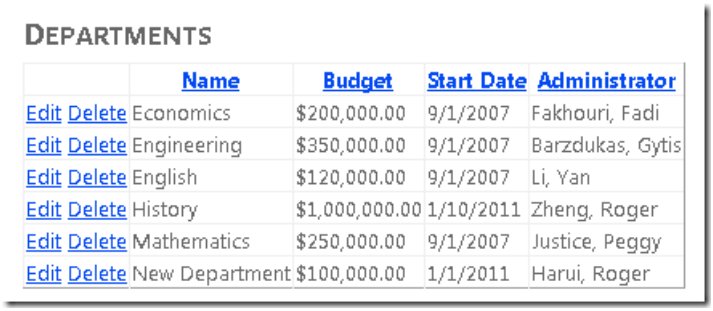
If you were going to create any unit tests that depended on this method returning a sorted list, you would need to sort the list before returning it. You won't be creating tests like that in this tutorial, so the method can just return the unsorted list of departments.

In the *SchoolBL.cs* file, add the following new method to the business logic class:

```
public IEnumerable<Department> GetDepartments(string sortExpression)
{
    return schoolRepository.GetDepartments(sortExpression);
}
```

This code passes the sort parameter to the repository method.

Run the *Departments.aspx* page.



DEPARTMENTS

	Name	Budget	Start Date	Administrator
Edit Delete	Economics	$200,000.00	9/1/2007	Fakhouri, Fadi
Edit Delete	Engineering	$350,000.00	9/1/2007	Barzdukas, Gytis
Edit Delete	English	$120,000.00	9/1/2007	Li, Yan
Edit Delete	History	$1,000,000.00	1/10/2011	Zheng, Roger
Edit Delete	Mathematics	$250,000.00	9/1/2007	Justice, Peggy
Edit Delete	New Department	$100,000.00	1/1/2011	Harui, Roger

You can now click any column heading to sort by that column. If the column is already sorted, clicking the heading reverses the sort direction.

Adding a Search Box

In this section you'll add a search text box, link it to the `ObjectDataSource` control using a control parameter, and add a method to the business logic class to support filtering.

Open the *Departments.aspx* page and add the following markup between the heading and the first `ObjectDataSource` control:

```
Enter any part of the name or leave the box blank to see all names:
    <asp:TextBox ID="SearchTextBox" runat="server" AutoPostBack="true"></asp:TextBox>
     <asp:Button ID="SearchButton" runat="server" Text="Search" />
```

In the `ObjectDataSource` control named `DepartmentsObjectDataSource`, do the following:

- Add a `SelectParameters` element for a parameter named `nameSearchString` that gets the value entered in the `SearchTextBox` control.

- Change the `SelectMethod` attribute value to `GetDepartmentsByName`. (You'll create this method later.)

The markup for the `ObjectDataSource` control now resembles the following example:

```
<asp:ObjectDataSource ID="DepartmentsObjectDataSource" runat="server"
TypeName="ContosoUniversity.BLL.SchoolBL"
        SelectMethod="GetDepartmentsByName" DeleteMethod="DeleteDepartment" UpdateMethod="UpdateDepartment"
        DataObjectTypeName="ContosoUniversity.DAL.Department" ConflictDetection="CompareAllValues"
        SortParameterName="sortExpression" OldValuesParameterFormatString="orig{0}"
        OnUpdated="DepartmentsObjectDataSource_Updated">
        <SelectParameters>
            <asp:ControlParameter ControlID="SearchTextBox" Name="nameSearchString" PropertyName="Text"
                Type="String" />
        </SelectParameters>
    </asp:ObjectDataSource>
```

In *ISchoolRepository.cs*, add a `GetDepartmentsByName` method that takes both `sortExpression` and `nameSearchString` parameters:

```
IEnumerable<Department> GetDepartmentsByName(string sortExpression, string nameSearchString);
```

In *SchoolRepository.cs*, add the following new method:

```
public IEnumerable<Department> GetDepartmentsByName(string sortExpression, string nameSearchString)
{
    if (String.IsNullOrWhiteSpace(sortExpression))
    {
        sortExpression = "Name";
    }
    if (String.IsNullOrWhiteSpace(nameSearchString))
    {
        nameSearchString = "";
    }
    return context.Departments.Include("Person").OrderBy("it." + sortExpression).Where(d =>
d.Name.Contains(nameSearchString)).ToList();
}
```

This code uses a `Where` method to select items that contain the search string. If the search string is empty, all records will be selected. Note that when you specify method calls together in one statement like this (`Include`, then `OrderBy`, then `Where`), the `Where` method must always be last.

Change the existing `GetDepartments` method that takes a `sortExpression` parameter to call the new method:

```
public IEnumerable<Department> GetDepartments(string sortExpression)
{
    return GetDepartmentsByName(sortExpression, "");
}
```

In *MockSchoolRepository.cs* in the test project, add the following new method:

```
public IEnumerable<Department> GetDepartmentsByName(string sortExpression, string nameSearchString)
{
    return departments;
}
```

In *SchoolBL.cs*, add the following new method:

```
public IEnumerable<Department> GetDepartmentsByName(string sortExpression, string nameSearchString)
{
    return schoolRepository.GetDepartmentsByName(sortExpression, nameSearchString);
}
```

Run the *Departments.aspx* page and enter a search string to make sure that the selection logic works. Leave the text box empty and try a search to make sure that all records are returned.

Adding a Details Column for Each Grid Row

Next, you want to see all of the courses for each department displayed in the right-hand cell of the grid. To do this, you'll use a nested `GridView` control and databind it to data from the `Courses` navigation property of the `Department` entity.

Open *Departments.aspx* and in the markup for the `GridView` control, specify a handler for the `RowDataBound` event. The markup for the opening tag of the control now resembles the following example.

```
<asp:GridView ID="DepartmentsGridView" runat="server" AutoGenerateColumns="False"
        DataSourceID="DepartmentsObjectDataSource" DataKeyNames="DepartmentID"
        OnRowUpdating="DepartmentsGridView_RowUpdating"
        OnRowDataBound="DepartmentsGridView_RowDataBound"
        AllowSorting="True" >
```

Add a new `TemplateField` element after the `Administrator` template field:

```
<asp:TemplateField HeaderText="Courses">
                <ItemTemplate>
                    <asp:GridView ID="CoursesGridView" runat="server" AutoGenerateColumns="False">
                        <Columns>
                            <asp:BoundField DataField="CourseID" HeaderText="ID" />
                            <asp:BoundField DataField="Title" HeaderText="Title" />
                        </Columns>
                    </asp:GridView>
                </ItemTemplate>
            </asp:TemplateField>
```

This markup creates a nested `GridView` control that shows the course number and title of a list of courses. It does not specify a data source because you'll databind it in code in the `RowDataBound` handler.

Open *Departments.aspx.cs* and add the following handler for the `RowDataBound` event:

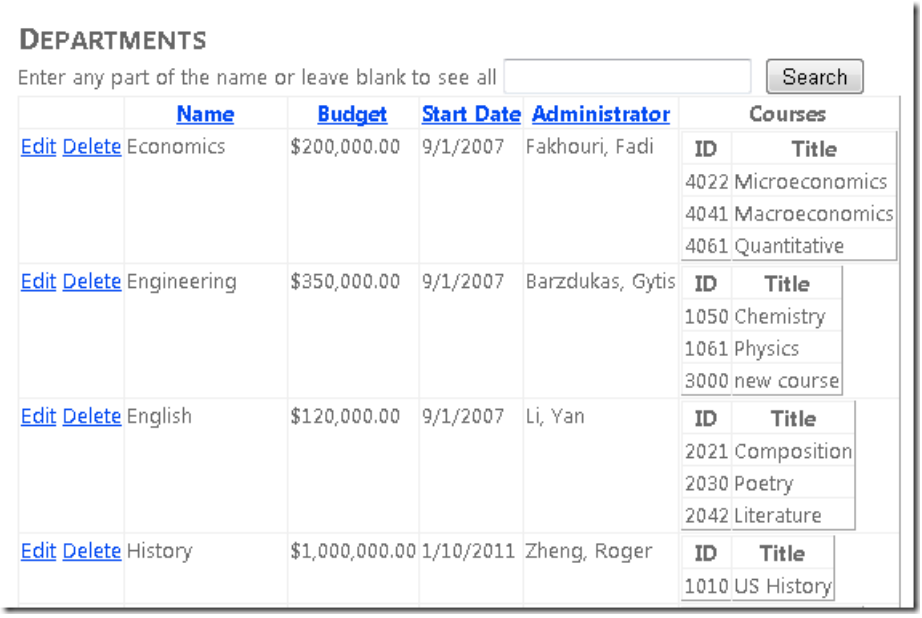
```
protected void DepartmentsGridView_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        var department = e.Row.DataItem as Department;
        var coursesGridView = (GridView)e.Row.FindControl("CoursesGridView");
        coursesGridView.DataSource = department.Courses.ToList();
        coursesGridView.DataBind();
    }
}
```

This code gets the `Department` entity from the event arguments, converts the `Courses` navigation property to a `List` collection, and databinds the nested `GridView` to the collection.

Open the *SchoolRepository.cs* file and specify eager loading for the `Courses` navigation property by calling the `Include` method in the object query that you create in the `GetDepartmentsByName` method. The `return` statement in the `GetDepartmentsByName` method now resembles the following example.

```
return context.Departments.Include("Person").Include("Courses").
    OrderBy("it." + sortExpression).Where(d => d.Name.Contains(nameSearchString)).ToList();
```

Run the page. In addition to the sorting and filtering capability that you added earlier, the GridView control now shows nested course details for each department.



This completes the introduction to sorting, filtering, and master-detail scenarios. In the next tutorial, you'll see how to handle concurrency.

PREVIOUS NEXT

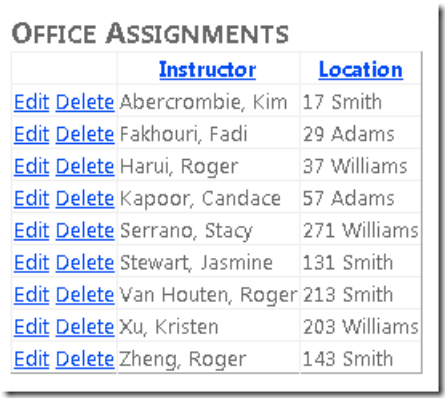
Handling Concurrency with the Entity Framework 4.0 in an ASP.NET 4 Web Application

2/20/2020 • 18 minutes to read • Edit Online

by Tom Dykstra

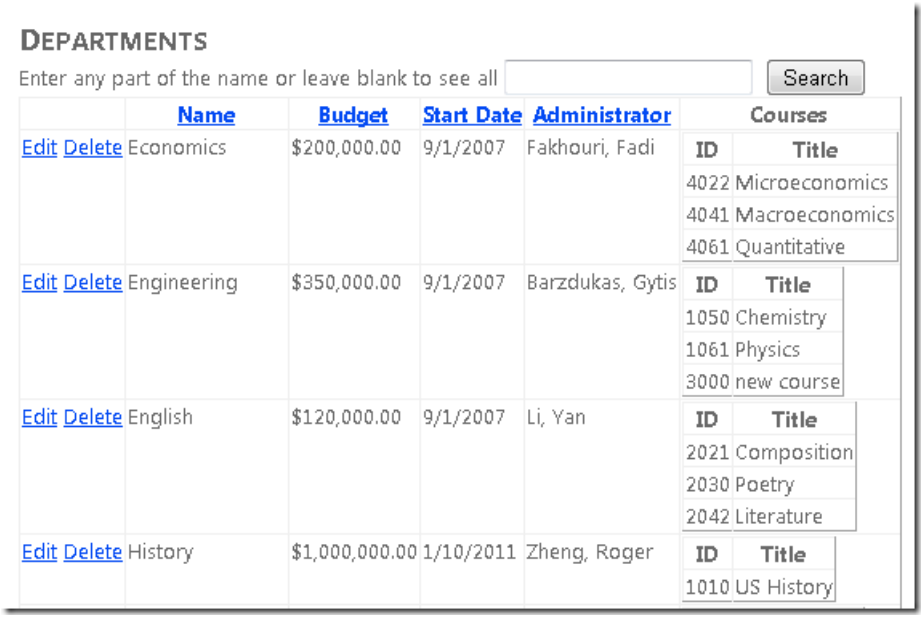
> This tutorial series builds on the Contoso University web application that is created by the Getting Started with the Entity Framework 4.0 tutorial series. If you didn't complete the earlier tutorials, as a starting point for this tutorial you can download the application that you would have created. You can also download the application that is created by the complete tutorial series. If you have questions about the tutorials, you can post them to the ASP.NET Entity Framework forum.

In the previous tutorial you learned how to sort and filter data using the `ObjectDataSource` control and the Entity Framework. This tutorial shows options for handling concurrency in an ASP.NET web application that uses the Entity Framework. You will create a new web page that's dedicated to updating instructor office assignments. You'll handle concurrency issues in that page and in the Departments page that you created earlier.



OFFICE ASSIGNMENTS

	Instructor	Location
Edit Delete	Abercrombie, Kim	17 Smith
Edit Delete	Fakhouri, Fadi	29 Adams
Edit Delete	Harui, Roger	37 Williams
Edit Delete	Kapoor, Candace	57 Adams
Edit Delete	Serrano, Stacy	271 Williams
Edit Delete	Stewart, Jasmine	131 Smith
Edit Delete	Van Houten, Roger	213 Smith
Edit Delete	Xu, Kristen	203 Williams
Edit Delete	Zheng, Roger	143 Smith



DEPARTMENTS

Enter any part of the name or leave blank to see all Search

	Name	Budget	Start Date	Administrator	Courses	
					ID	Title
Edit Delete	Economics	$200,000.00	9/1/2007	Fakhouri, Fadi	4022	Microeconomics
					4041	Macroeconomics
					4061	Quantitative
Edit Delete	Engineering	$350,000.00	9/1/2007	Barzdukas, Gytis	1050	Chemistry
					1061	Physics
					3000	new course
Edit Delete	English	$120,000.00	9/1/2007	Li, Yan	2021	Composition
					2030	Poetry
					2042	Literature
Edit Delete	History	$1,000,000.00	1/10/2011	Zheng, Roger	1010	US History

Concurrency Conflicts

A concurrency conflict occurs when one user edits a record and another user edits the same record before the first user's change is written to the database. If you don't set up the Entity Framework to detect such conflicts,

whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable, and you don't have to configure the application to handle possible concurrency conflicts. (If there are few users, or few updates, or if isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit.) If you don't need to worry about concurrency conflicts, you can skip this tutorial; the remaining two tutorials in the series don't depend on anything you build in this one.

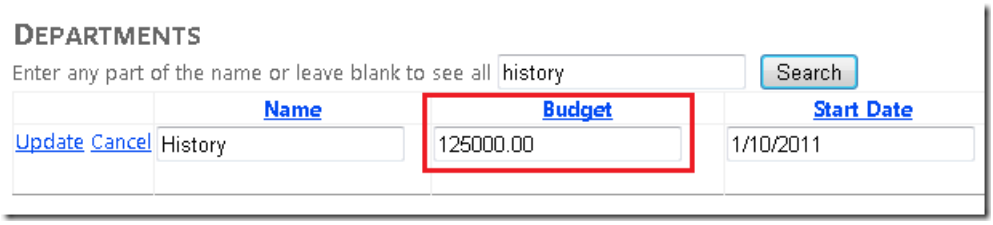
Pessimistic Concurrency (Locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called *pessimistic concurrency*. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

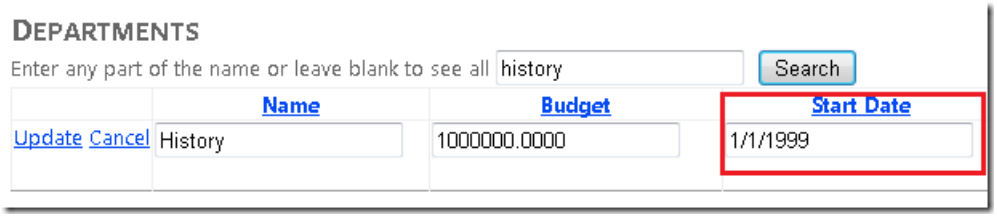
Managing locks has some disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases (that is, it doesn't scale well). For these reasons, not all database management systems support pessimistic concurrency. The Entity Framework provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

The alternative to pessimistic concurrency is *optimistic concurrency*. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, John runs the *Department.aspx* page, clicks the **Edit** link for the History department, and reduces the **Budget** amount from $1,000,000.00 to $125,000.00. (John administers a competing department and wants to free up money for his own department.)



Before John clicks **Update**, Jane runs the same page, clicks the **Edit** link for the History department, and then changes the **Start Date** field from 1/10/2011 to 1/1/1999. (Jane administers the History department and wants to give it more seniority.)



John clicks **Update** first, then Jane clicks **Update**. Jane's browser now lists the **Budget** amount as $1,000,000.00, but this is incorrect because the amount has been changed by John to $125,000.00.

Some of the actions you can take in this scenario include the following:

- You can keep track of which property a user has modified and update only the corresponding columns in the database. In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the History department, they will see 1/1/1999 and $125,000.00.

  This is the default behavior in the Entity Framework, and it can substantially reduce the number of conflicts that could result in data loss. However, this behavior doesn't avoid data loss if competing changes are

made to the same property of an entity. In addition, this behavior isn't always possible; when you map stored procedures to an entity type, all of an entity's properties are updated when any changes to the entity are made in the database.

- You can let Jane's change overwrite John's change. After Jane clicks **Update**, the **Budget** amount goes back to $1,000,000.00. This is called a *Client Wins* or *Last in Wins* scenario. (The client's values take precedence over what's in the data store.)

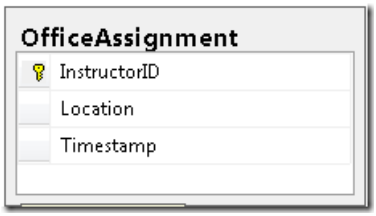
- You can prevent Jane's change from being updated in the database. Typically, you would display an error message, show her the current state of the data, and allow her to reenter her changes if she still wants to make them. You could further automate the process by saving her input and giving her an opportunity to reapply it without having to reenter it. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.)

Detecting Concurrency Conflicts

In the Entity Framework, you can resolve conflicts by handling `OptimisticConcurrencyException` exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database and the data model appropriately. Some options for enabling conflict detection include the following:

- In the database, include a table column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the `Where` clause of SQL `Update` or `Delete` commands.

  That's the purpose of the `Timestamp` column in the `OfficeAssignment` table.

  The data type of the `Timestamp` column is also called `Timestamp`. However, the column doesn't actually contain a date or time value. Instead, the value is a sequential number that's incremented each time the row is updated. In an `Update` or `Delete` command, the `Where` clause includes the original `Timestamp` value. If the row being updated has been changed by another user, the value in `Timestamp` is different than the original value, so the `Where` clause returns no row to update. When the Entity Framework finds that no rows have been updated by the current `Update` or `Delete` command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

- Configure the Entity Framework to include the original values of every column in the table in the `Where` clause of `Update` and `Delete` commands.

  As in the first option, if anything in the row has changed since the row was first read, the `Where` clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. This method is as effective as using a `Timestamp` field, but can be inefficient. For database tables that have many columns, it can result in very large `Where` clauses, and in a web application it can require that you maintain large amounts of state. Maintaining large amounts of state can affect application performance because it either requires server resources (for example, session state) or must be included in the web page itself (for example, view state).

In this tutorial you will add error handling for optimistic concurrency conflicts for an entity that doesn't have a tracking property (the `Department` entity) and for an entity that does have a tracking property (the `OfficeAssignment` entity).

Handling Optimistic Concurrency Without a Tracking Property

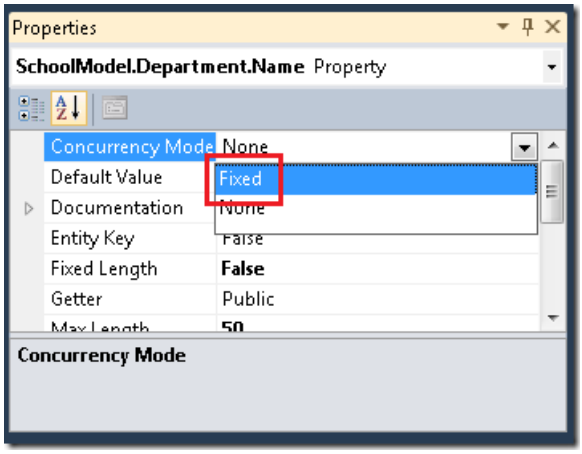
To implement optimistic concurrency for the `Department` entity, which doesn't have a tracking (`Timestamp`) property, you will complete the following tasks:

- Change the data model to enable concurrency tracking for `Department` entities.
- In the `SchoolRepository` class, handle concurrency exceptions in the `SaveChanges` method.
- In the *Departments.aspx* page, handle concurrency exceptions by displaying a message to the user warning that the attempted changes were unsuccessful. The user can then see the current values and retry the changes if they are still needed.

Enabling Concurrency Tracking in the Data Model

In Visual Studio, open the Contoso University web application that you were working with in the previous tutorial in this series.

Open *SchoolModel.edmx*, and in the data model designer, right-click the `Name` property in the `Department` entity and then click **Properties**. In the **Properties** window, change the `ConcurrencyMode` property to `Fixed`.



Do the same for the other non-primary-key scalar properties (`Budget`, `StartDate`, and `Administrator`.) (You can't do this for navigation properties.) This specifies that whenever the Entity Framework generates a `Update` or `Delete` SQL command to update the `Department` entity in the database, these columns (with original values) must be included in the `Where` clause. If no row is found when the `Update` or `Delete` command executes, the Entity Framework will throw an optimistic-concurrency exception.

Save and close the data model.

Handling Concurrency Exceptions in the DAL

Open *SchoolRepository.cs* and add the following `using` statement for the `System.Data` namespace:

```
using System.Data;
```

Add the following new `SaveChanges` method, which handles optimistic concurrency exceptions:

```
public void SaveChanges()
{
    try
    {
        context.SaveChanges();
    }
    catch (OptimisticConcurrencyException ocex)
    {
        context.Refresh(RefreshMode.StoreWins, ocex.StateEntries[0].Entity);
        throw ocex;
    }
}
```

If a concurrency error occurs when this method is called, the property values of the entity in memory are replaced with the values currently in the database. The concurrency exception is rethrown so that the web page can handle it.

In the `DeleteDepartment` and `UpdateDepartment` methods, replace the existing call to `context.SaveChanges()` with a call to `SaveChanges()` in order to invoke the new method.

Handling Concurrency Exceptions in the Presentation Layer

Open *Departments.aspx* and add an `OnDeleted="DepartmentsObjectDataSource_Deleted"` attribute to the `DepartmentsObjectDataSource` control. The opening tag for the control will now resemble the following example.

```
<asp:ObjectDataSource ID="DepartmentsObjectDataSource" runat="server"
        TypeName="ContosoUniversity.BLL.SchoolBL" DataObjectTypeName="ContosoUniversity.DAL.Department"
        SelectMethod="GetDepartmentsByName" DeleteMethod="DeleteDepartment" UpdateMethod="UpdateDepartment"
        ConflictDetection="CompareAllValues" OldValuesParameterFormatString="orig{0}"
        OnUpdated="DepartmentsObjectDataSource_Updated" SortParameterName="sortExpression"
        OnDeleted="DepartmentsObjectDataSource_Deleted" >
```

In the `DepartmentsGridView` control, specify all of the table columns in the `DataKeyNames` attribute, as shown in the following example. Note that this will create very large view state fields, which is one reason why using a tracking field is generally the preferred way to track concurrency conflicts.

```
<asp:GridView ID="DepartmentsGridView" runat="server" AutoGenerateColumns="False"
        DataSourceID="DepartmentsObjectDataSource"
        DataKeyNames="DepartmentID,Name,Budget,StartDate,Administrator"
        OnRowUpdating="DepartmentsGridView_RowUpdating"
        OnRowDataBound="DepartmentsGridView_RowDataBound"
        AllowSorting="True" >
```

Open *Departments.aspx.cs* and add the following `using` statement for the `System.Data` namespace:

```
using System.Data;
```

Add the following new method, which you will call from the data source control's `Updated` and `Deleted` event handlers for handling concurrency exceptions:

```
private void CheckForOptimisticConcurrencyException(ObjectDataSourceStatusEventArgs e, string function)
{
    if (e.Exception.InnerException is OptimisticConcurrencyException)
    {
        var concurrencyExceptionValidator = new CustomValidator();
        concurrencyExceptionValidator.IsValid = false;
        concurrencyExceptionValidator.ErrorMessage =
            "The record you attempted to edit or delete was modified by another " +
            "user after you got the original value. The edit or delete operation was canceled " +
            "and the other user's values have been displayed so you can " +
            "determine whether you still want to edit or delete this record.";
        Page.Validators.Add(concurrencyExceptionValidator);
        e.ExceptionHandled = true;
    }
}
```

This code checks the exception type, and if it's a concurrency exception, the code dynamically creates a `CustomValidator` control that in turn displays a message in the `ValidationSummary` control.

Call the new method from the `Updated` event handler that you added earlier. In addition, create a new `Deleted` event handler that calls the same method (but doesn't do anything else):

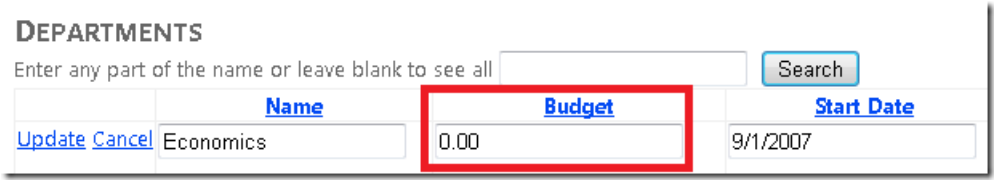
```
protected void DepartmentsObjectDataSource_Updated(object sender, ObjectDataSourceStatusEventArgs e)
{
    if (e.Exception != null)
    {
        CheckForOptimisticConcurrencyException(e, "update");
        // ...
    }
}

protected void DepartmentsObjectDataSource_Deleted(object sender, ObjectDataSourceStatusEventArgs e)
{
    if (e.Exception != null)
    {
        CheckForOptimisticConcurrencyException(e, "delete");
    }
}
```

Testing Optimistic Concurrency in the Departments Page

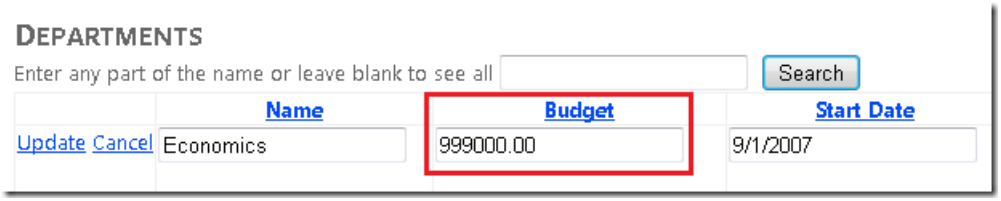
Run the *Departments.aspx* page.

Click **Edit** in a row and change the value in the **Budget** column. (Remember that you can only edit records that you've created for this tutorial, because the existing `School` database records contain some invalid data. The record for the Economics department is a safe one to experiment with.)

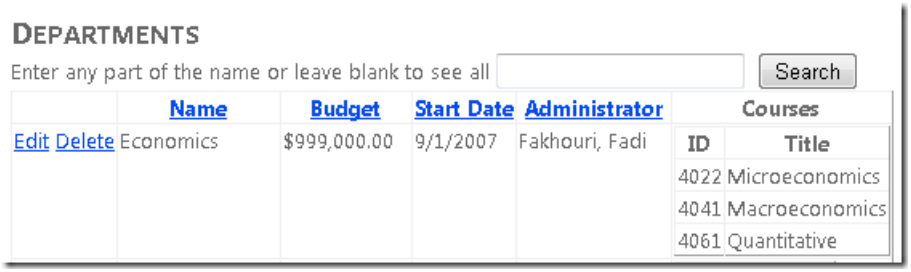


Open a new browser window and run the page again (copy the URL from the first browser window's address box to the second browser window).

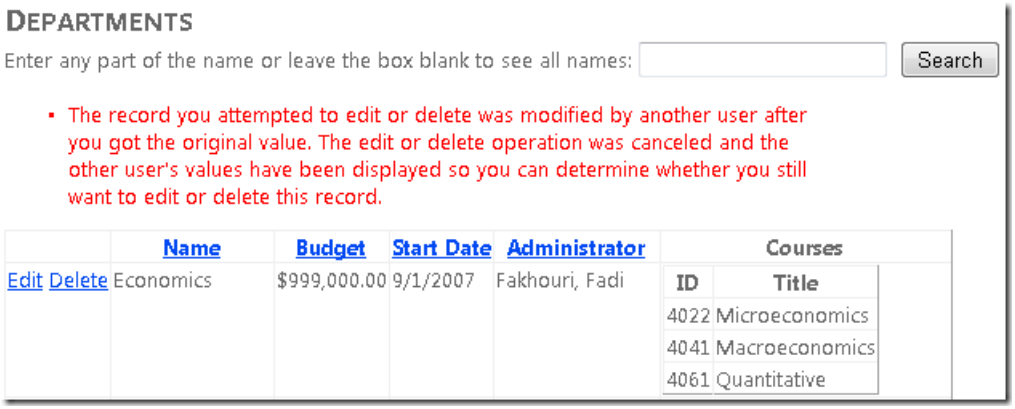
Click **Edit** in the same row you edited earlier and change the **Budget** value to something different.



In the second browser window, click **Update**. The **Budget** amount is successfully changed to this new value.



In the first browser window, click **Update**. The update fails. The **Budget** amount is redisplayed using the value you set in the second browser window, and you see an error message.



Handling Optimistic Concurrency Using a Tracking Property

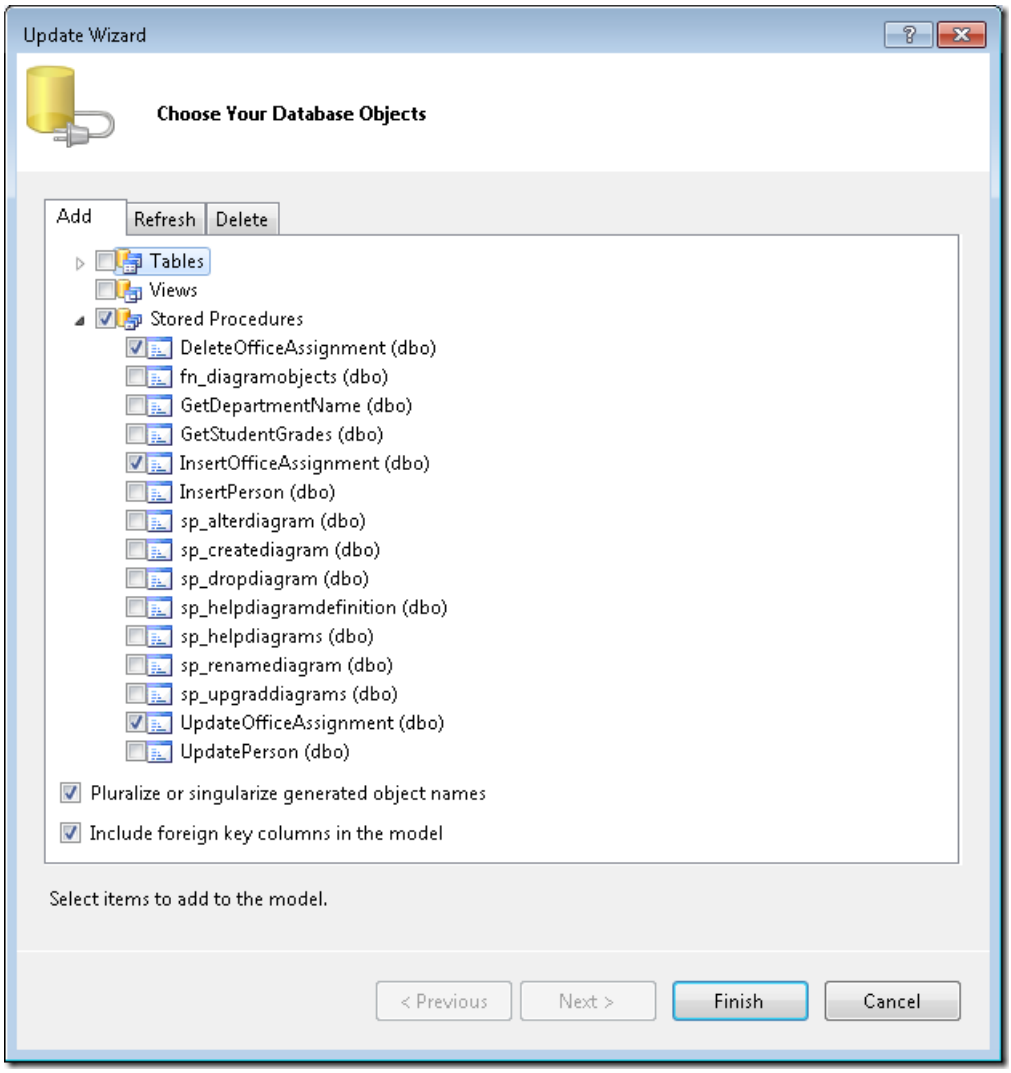
To handle optimistic concurrency for an entity that has a tracking property, you will complete the following tasks:

- Add stored procedures to the data model to manage `OfficeAssignment` entities. (Tracking properties and stored procedures don't have to be used together; they're just grouped together here for illustration.)
- Add CRUD methods to the DAL and the BLL for `OfficeAssignment` entities, including code to handle optimistic concurrency exceptions in the DAL.
- Create an office-assignments web page.
- Test optimistic concurrency in the new web page.

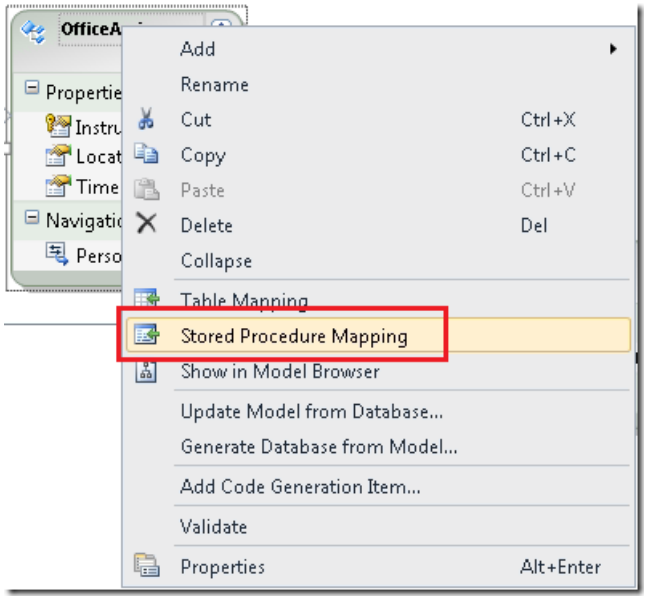
Adding OfficeAssignment Stored Procedures to the Data Model

Open the *SchoolModel.edmx* file in the model designer, right-click the design surface, and click **Update Model**

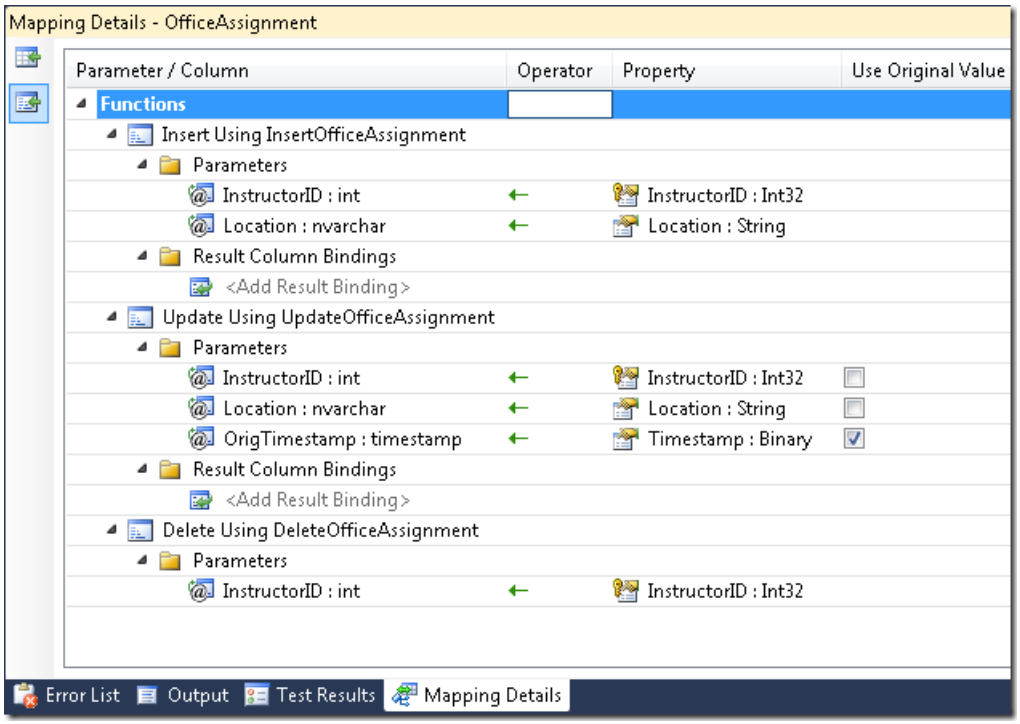
from Database. In the **Add** tab of the **Choose Your Database Objects** dialog box, expand **Stored Procedures** and select the three `OfficeAssignment` stored procedures (see the following screenshot), and then click **Finish**. (These stored procedures were already in the database when you downloaded or created it using a script.)



Right-click the `OfficeAssignment` entity and select **Stored Procedure Mapping**.



Set the **Insert**, **Update**, and **Delete** functions to use their corresponding stored procedures. For the `OrigTimestamp` parameter of the `Update` function, set the **Property** to `Timestamp` and select the **Use Original Value** option.



When the Entity Framework calls the `UpdateOfficeAssignment` stored procedure, it will pass the original value of the `Timestamp` column in the `OrigTimestamp` parameter. The stored procedure uses this parameter in its `Where` clause:

```
ALTER PROCEDURE [dbo].[UpdateOfficeAssignment]
    @InstructorID int,
    @Location nvarchar(50),
    @OrigTimestamp timestamp
    AS
    UPDATE OfficeAssignment SET Location=@Location
    WHERE InstructorID=@InstructorID AND [Timestamp]=@OrigTimestamp;
    IF @@ROWCOUNT > 0
    BEGIN
        SELECT [Timestamp] FROM OfficeAssignment
            WHERE InstructorID=@InstructorID;
    END
```

The stored procedure also selects the new value of the `Timestamp` column after the update so that the Entity Framework can keep the `OfficeAssignment` entity that's in memory in sync with the corresponding database row.

(Note that the stored procedure for deleting an office assignment doesn't have an `OrigTimestamp` parameter. Because of this, the Entity Framework can't verify that an entity is unchanged before deleting it.)

Save and close the data model.

Adding OfficeAssignment Methods to the DAL

Open *ISchoolRepository.cs* and add the following CRUD methods for the `OfficeAssignment` entity set:

```
IEnumerable<OfficeAssignment> GetOfficeAssignments(string sortExpression);
void InsertOfficeAssignment(OfficeAssignment OfficeAssignment);
void DeleteOfficeAssignment(OfficeAssignment OfficeAssignment);
void UpdateOfficeAssignment(OfficeAssignment OfficeAssignment, OfficeAssignment origOfficeAssignment);
```

Add the following new methods to *SchoolRepository.cs*. In the `UpdateOfficeAssignment` method, you call the local `SaveChanges` method instead of `context.SaveChanges`.

```
public IEnumerable<OfficeAssignment> GetOfficeAssignments(string sortExpression)
{
    return new ObjectQuery<OfficeAssignment>("SELECT VALUE o FROM OfficeAssignments AS o",
context).Include("Person").OrderBy("it." + sortExpression).ToList();
}

public void InsertOfficeAssignment(OfficeAssignment officeAssignment)
{
    context.OfficeAssignments.AddObject(officeAssignment);
    context.SaveChanges();
}

public void DeleteOfficeAssignment(OfficeAssignment officeAssignment)
{
    context.OfficeAssignments.Attach(officeAssignment);
    context.OfficeAssignments.DeleteObject(officeAssignment);
    context.SaveChanges();
}

public void UpdateOfficeAssignment(OfficeAssignment officeAssignment, OfficeAssignment origOfficeAssignment)
{
    context.OfficeAssignments.Attach(origOfficeAssignment);
    context.ApplyCurrentValues("OfficeAssignments", officeAssignment);
    SaveChanges();
}
```

In the test project, open *MockSchoolRepository.cs* and add the following `OfficeAssignment` collection and CRUD methods to it. (The mock repository must implement the repository interface, or the solution won't compile.)

```
List<OfficeAssignment> officeAssignments = new List<OfficeAssignment>();

public IEnumerable<OfficeAssignment> GetOfficeAssignments(string sortExpression)
{
    return officeAssignments;
}

public void InsertOfficeAssignment(OfficeAssignment officeAssignment)
{
    officeAssignments.Add(officeAssignment);
}

public void DeleteOfficeAssignment(OfficeAssignment officeAssignment)
{
    officeAssignments.Remove(officeAssignment);
}

public void UpdateOfficeAssignment(OfficeAssignment officeAssignment, OfficeAssignment origOfficeAssignment)
{
    officeAssignments.Remove(origOfficeAssignment);
    officeAssignments.Add(officeAssignment);
}
```

Adding OfficeAssignment Methods to the BLL

In the main project, open *SchoolBL.cs* and add the following CRUD methods for the `OfficeAssignment` entity set to it:

```
public IEnumerable<OfficeAssignment> GetOfficeAssignments(string sortExpression)
{
    if (string.IsNullOrEmpty(sortExpression)) sortExpression = "Person.LastName";
    return schoolRepository.GetOfficeAssignments(sortExpression);
}

public void InsertOfficeAssignment(OfficeAssignment officeAssignment)
{
    try
    {
        schoolRepository.InsertOfficeAssignment(officeAssignment);
    }
    catch (Exception ex)
    {
        //Include catch blocks for specific exceptions first,
        //and handle or log the error as appropriate in each.
        //Include a generic catch block like this one last.
        throw ex;
    }
}

public void DeleteOfficeAssignment(OfficeAssignment officeAssignment)
{
    try
    {
        schoolRepository.DeleteOfficeAssignment(officeAssignment);
    }
    catch (Exception ex)
    {
        //Include catch blocks for specific exceptions first,
        //and handle or log the error as appropriate in each.
        //Include a generic catch block like this one last.
        throw ex;
    }
}

public void UpdateOfficeAssignment(OfficeAssignment officeAssignment, OfficeAssignment origOfficeAssignment)
{
    try
    {
        schoolRepository.UpdateOfficeAssignment(officeAssignment, origOfficeAssignment);
    }
    catch (Exception ex)
    {
        //Include catch blocks for specific exceptions first,
        //and handle or log the error as appropriate in each.
        //Include a generic catch block like this one last.
        throw ex;
    }
}
```

Creating an OfficeAssignments Web Page

Create a new web page that uses the *Site.Master* master page and name it *OfficeAssignments.aspx*. Add the following markup to the `Content` control named `Content2`:

```
<h2>Office Assignments</h2>
    <asp:ObjectDataSource ID="OfficeAssignmentsObjectDataSource" runat="server"
TypeName="ContosoUniversity.BLL.SchoolBL"
        DataObjectTypeName="ContosoUniversity.DAL.OfficeAssignment" SelectMethod="GetOfficeAssignments"
        DeleteMethod="DeleteOfficeAssignment" UpdateMethod="UpdateOfficeAssignment"
ConflictDetection="CompareAllValues"
        OldValuesParameterFormatString="orig{0}"
        SortParameterName="sortExpression"  OnUpdated="OfficeAssignmentsObjectDataSource_Updated">
    </asp:ObjectDataSource>
    <asp:ValidationSummary ID="OfficeAssignmentsValidationSummary" runat="server" ShowSummary="true"
        DisplayMode="BulletList" Style="color: Red; width: 40em;" />
    <asp:GridView ID="OfficeAssignmentsGridView" runat="server" AutoGenerateColumns="False"
        DataSourceID="OfficeAssignmentsObjectDataSource" DataKeyNames="InstructorID,Timestamp"
        AllowSorting="True">
        <Columns>
            <asp:CommandField ShowEditButton="True" ShowDeleteButton="True" ItemStyle-VerticalAlign="Top">
                <ItemStyle VerticalAlign="Top"></ItemStyle>
            </asp:CommandField>
            <asp:TemplateField HeaderText="Instructor" SortExpression="Person.LastName">
                <ItemTemplate>
                    <asp:Label ID="InstructorLastNameLabel" runat="server" Text='<%# Eval("Person.LastName")
%>'></asp:Label>,
                    <asp:Label ID="InstructorFirstNameLabel" runat="server" Text='<%#
Eval("Person.FirstMidName") %>'></asp:Label>
                </ItemTemplate>
            </asp:TemplateField>
            <asp:DynamicField DataField="Location" HeaderText="Location" SortExpression="Location"/>
        </Columns>
        <SelectedRowStyle BackColor="LightGray"></SelectedRowStyle>
    </asp:GridView>
```

Notice that in the `DataKeyNames` attribute, the markup specifies the `Timestamp` property as well as the record key (`InstructorID`). Specifying properties in the `DataKeyNames` attribute causes the control to save them in control state (which is similar to view state) so that the original values are available during postback processing.

If you didn't save the `Timestamp` value, the Entity Framework would not have it for the `Where` clause of the SQL `Update` command. Consequently nothing would be found to update. As a result, the Entity Framework would throw an optimistic concurrency exception every time an `OfficeAssignment` entity is updated.

Open *OfficeAssignments.aspx.cs* and add the following `using` statement for the data access layer:

```
using ContosoUniversity.DAL;
```

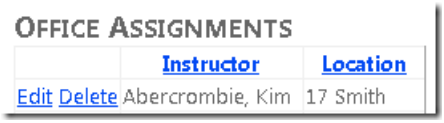
Add the following `Page_Init` method, which enables Dynamic Data functionality. Also add the following handler for the `ObjectDataSource` control's `Updated` event in order to check for concurrency errors:

```
protected void Page_Init(object sender, EventArgs e)
{
    OfficeAssignmentsGridView.EnableDynamicData(typeof(OfficeAssignment));
}

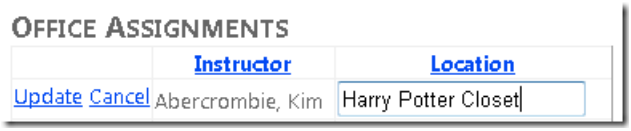
protected void OfficeAssignmentsObjectDataSource_Updated(object sender, ObjectDataSourceStatusEventArgs e)
{
    if (e.Exception != null)
    {
        var concurrencyExceptionValidator = new CustomValidator();
        concurrencyExceptionValidator.IsValid = false;
        concurrencyExceptionValidator.ErrorMessage = "The record you attempted to " +
            "update has been modified by another user since you last visited this page. " +
            "Your update was canceled to allow you to review the other user's " +
            "changes and determine if you still want to update this record.";
        Page.Validators.Add(concurrencyExceptionValidator);
        e.ExceptionHandled = true;
    }
}
```

Testing Optimistic Concurrency in the OfficeAssignments Page

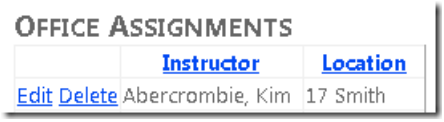
Run the *OfficeAssignments.aspx* page.



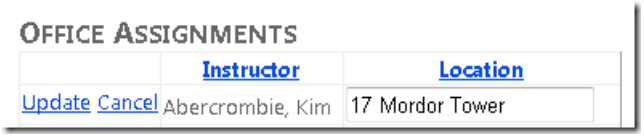
Click **Edit** in a row and change the value in the **Location** column.



Open a new browser window and run the page again (copy the URL from the first browser window to the second browser window).

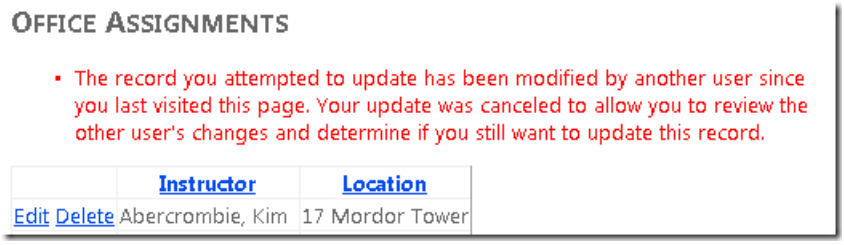


Click **Edit** in the same row you edited earlier and change the **Location** value to something different.



In the second browser window, click **Update**.

Switch to the first browser window and click **Update**.



You see an error message and the **Location** value has been updated to show the value you changed it to in the second browser window.

Handling Concurrency with the EntityDataSource Control

The `EntityDataSource` control includes built-in logic that recognizes the concurrency settings in the data model and handles update and delete operations accordingly. However, as with all exceptions, you must handle `OptimisticConcurrencyException` exceptions yourself in order to provide a user-friendly error message.

Next, you will configure the *Courses.aspx* page (which uses an `EntityDataSource` control) to allow update and delete operations and to display an error message if a concurrency conflict occurs. The `Course` entity doesn't have a concurrency tracking column, so you will use the same method that you did with the `Department` entity: track the values of all non-key properties.

Open the *SchoolModel.edmx* file. For the non-key properties of the `Course` entity (`Title`, `Credits`, and `DepartmentID`), set the **Concurrency Mode** property to `Fixed`. Then save and close the data model.

Open the *Courses.aspx* page and make the following changes:

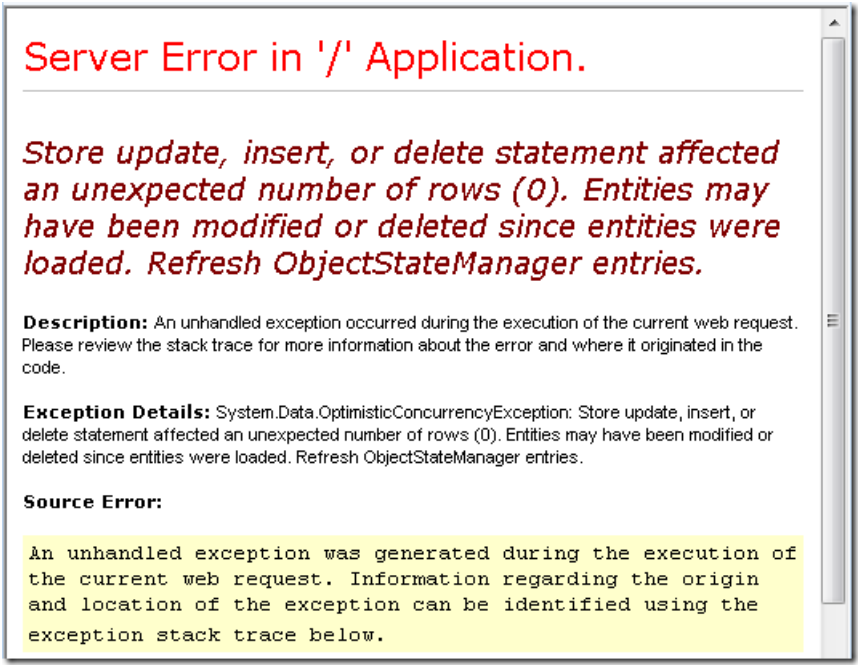
- In the `CoursesEntityDataSource` control, add `EnableUpdate="true"` and `EnableDelete="true"` attributes. The opening tag for that control now resembles the following example:

```
<asp:EntityDataSource ID="CoursesEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="false"
        AutoGenerateWhereClause="True" EntitySetName="Courses"
        EnableUpdate="true" EnableDelete="true">
```

- In the `CoursesGridView` control, change the `DataKeyNames` attribute value to `"CourseID,Title,Credits,DepartmentID"`. Then add a `CommandField` element to the `Columns` element that shows **Edit** and **Delete** buttons (`<asp:CommandField ShowEditButton="True" ShowDeleteButton="True" />`). The `GridView` control now resembles the following example:

```
<asp:GridView ID="CoursesGridView" runat="server" AutoGenerateColumns="False"
        DataKeyNames="CourseID,Title,Credits,DepartmentID"
        DataSourceID="CoursesEntityDataSource" >
        <Columns>
            <asp:CommandField ShowEditButton="True" ShowDeleteButton="True" />
            <asp:BoundField DataField="CourseID" HeaderText="CourseID" ReadOnly="True"
SortExpression="CourseID" />
            <asp:BoundField DataField="Title" HeaderText="Title" SortExpression="Title" />
            <asp:BoundField DataField="Credits" HeaderText="Credits" SortExpression="Credits" />
        </Columns>
    </asp:GridView>
```

Run the page and create a conflict situation as you did before in the Departments page. Run the page in two browser windows, click **Edit** in the same line in each window, and make a different change in each one. Click **Update** in one window and then click **Update** in the other window. When you click **Update** the second time, you see the error page that results from an unhandled concurrency exception.



You handle this error in a manner very similar to how you handled it for the `ObjectDataSource` control. Open the *Courses.aspx* page, and in the `CoursesEntityDataSource` control, specify handlers for the `Deleted` and `Updated` events. The opening tag of the control now resembles the following example:

```
<asp:EntityDataSource ID="CoursesEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.SchoolEntities" EnableFlattening="false"
        AutoGenerateWhereClause="true" EntitySetName="Courses"
        EnableUpdate="true" EnableDelete="true"
        OnDeleted="CoursesEntityDataSource_Deleted"
        OnUpdated="CoursesEntityDataSource_Updated">
```

Before the `CoursesGridView` control, add the following `ValidationSummary` control:

```
<asp:ValidationSummary ID="CoursesValidationSummary" runat="server"
        ShowSummary="true" DisplayMode="BulletList"  />
```

In *Courses.aspx.cs*, add a `using` statement for the `System.Data` namespace, add a method that checks for concurrency exceptions, and add handlers for the `EntityDataSource` control's `Updated` and `Deleted` handlers. The code will look like the following:

```
using System.Data;
```

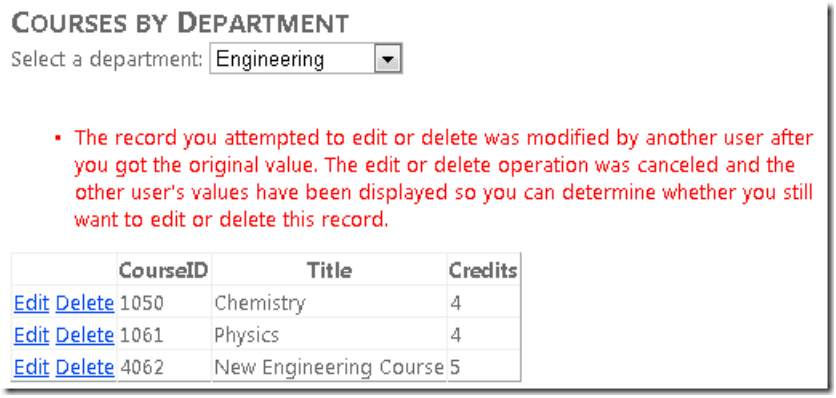
```
protected void CoursesEntityDataSource_Updated(object sender, EntityDataSourceChangedEventArgs e)
{
    CheckForOptimisticConcurrencyException(e, "update");
}

protected void CoursesEntityDataSource_Deleted(object sender, EntityDataSourceChangedEventArgs e)
{
    CheckForOptimisticConcurrencyException(e, "delete");
}

private void CheckForOptimisticConcurrencyException(EntityDataSourceChangedEventArgs e, string function)
{
    if (e.Exception != null && e.Exception is OptimisticConcurrencyException)
    {
        var concurrencyExceptionValidator = new CustomValidator();
        concurrencyExceptionValidator.IsValid = false;
        concurrencyExceptionValidator.ErrorMessage =
            "The record you attempted to edit or delete was modified by another " +
            "user after you got the original value. The edit or delete operation was canceled " +
            "and the other user's values have been displayed so you can " +
            "determine whether you still want to edit or delete this record.";
        Page.Validators.Add(concurrencyExceptionValidator);
        e.ExceptionHandled = true;
    }
}
```

The only difference between this code and what you did for the `ObjectDataSource` control is that in this case the concurrency exception is in the `Exception` property of the event arguments object rather than in that exception's `InnerException` property.

Run the page and create a concurrency conflict again. This time you see an error message:



This completes the introduction to handling concurrency conflicts. The next tutorial will provide guidance on how to improve performance in a web application that uses the Entity Framework.

PREVIOUS NEXT

Maximizing Performance with the Entity Framework 4.0 in an ASP.NET 4 Web Application

2/20/2020 • 19 minutes to read • Edit Online

by Tom Dykstra

> This tutorial series builds on the Contoso University web application that is created by the Getting Started with the Entity Framework 4.0 tutorial series. If you didn't complete the earlier tutorials, as a starting point for this tutorial you can download the application that you would have created. You can also download the application that is created by the complete tutorial series. If you have questions about the tutorials, you can post them to the ASP.NET Entity Framework forum.

In the previous tutorial, you saw how to handle concurrency conflicts. This tutorial shows options for improving the performance of an ASP.NET web application that uses the Entity Framework. You'll learn several methods for maximizing performance or for diagnosing performance problems.

Information presented in the following sections is likely to be useful in a broad variety of scenarios:

- Efficiently load related data.
- Manage view state.

Information presented in the following sections might be useful if you have individual queries that present performance problems:

- Use the `NoTracking` merge option.
- Pre-compile LINQ queries.
- Examine query commands sent to the database.

Information presented in the following section is potentially useful for applications that have extremely large data models:

- Pre-generate views.

> **NOTE**
>
> Web application performance is affected by many factors, including things like the size of request and response data, the speed of database queries, how many requests the server can queue and how quickly it can service them, and even the efficiency of any client-script libraries you might be using. If performance is critical in your application, or if testing or experience shows that application performance isn't satisfactory, you should follow normal protocol for performance tuning. Measure to determine where performance bottlenecks are occurring, and then address the areas that will have the greatest impact on overall application performance.
>
> This topic focuses mainly on ways in which you can potentially improve the performance specifically of the Entity Framework in ASP.NET. The suggestions here are useful if you determine that data access is one of the performance bottlenecks in your application. Except as noted, the methods explained here shouldn't be considered "best practices" in general — many of them are appropriate only in exceptional situations or to address very specific kinds of performance bottlenecks.

To start the tutorial, start Visual Studio and open the Contoso University web application that you were working with in the previous tutorial.

Efficiently Loading Related Data

There are several ways that the Entity Framework can load related data into the navigation properties of an entity:

- *Lazy loading*. When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. This results in multiple queries sent to the database — one for the entity itself and one each time that related data for the entity must be retrieved.

```
departments = context.Departments.ToList();
foreach (Department d in departments)
{
    if (d.Name == "History")
    {
        adminLastName = d.Person.LastName;
    }

}
```

Read Department rows

Read one Person row

Eager loading. When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading by using the `Include` method, as you've seen already in these tutorials.

```
departments = context.Departments.Include("Person").ToList();
foreach (Department d in departments)
{
    if (d.Name == "History")
    {
        adminLastName = d.Person.LastName;
    }
}
```

Read Department rows and all related Person rows

- *Explicit loading*. This is similar to lazy loading, except that you explicitly retrieve the related data in code; it doesn't happen automatically when you access a navigation property. You load related data manually using the `Load` method of the navigation property for collections, or you use the `Load` method of the reference property for properties that hold a single object. (For example, you call the `PersonReference.Load` method to load the `Person` navigation property of a `Department` entity.)

```
departments = context.Departments.ToList();
foreach (Department d in departments)
{
    if (d.Name == "History")
    {
        d.PersonReference.Load();
        adminLastName = d.Person.LastName;
    }
}
```

Read Department rows

Read one Person row

Because they don't immediately retrieve the property values, lazy loading and explicit loading are also both known as *deferred loading*.

Lazy loading is the default behavior for an object context that has been generated by the designer. If you open the *SchoolModel.Designer.cs* file that defines the object context class, you'll find three constructor methods, and each of them includes the following statement:

```
this.ContextOptions.LazyLoadingEnabled = true;
```

In general, if you know you need related data for every entity retrieved, eager loading offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. On the other hand, if you need to access an entity's navigation properties only infrequently or only for a small set of the entities, lazy loading or explicit loading may be more efficient, because eager loading would retrieve more data than you need.

In a web application, lazy loading may be of relatively little value anyway, because user actions that affect the need for related data take place in the browser, which has no connection to the object context that rendered the page. On the other hand, when you databind a control, you typically know what data you need, and so it's generally best to choose eager loading or deferred loading based on what's appropriate in each scenario.

In addition, a databound control might use an entity object after the object context is disposed. In that case, an attempt to lazy-load a navigation property would fail. The error message you receive is clear: "

```
The ObjectContext instance has been disposed and can no longer be used for operations that require a
connection.
```

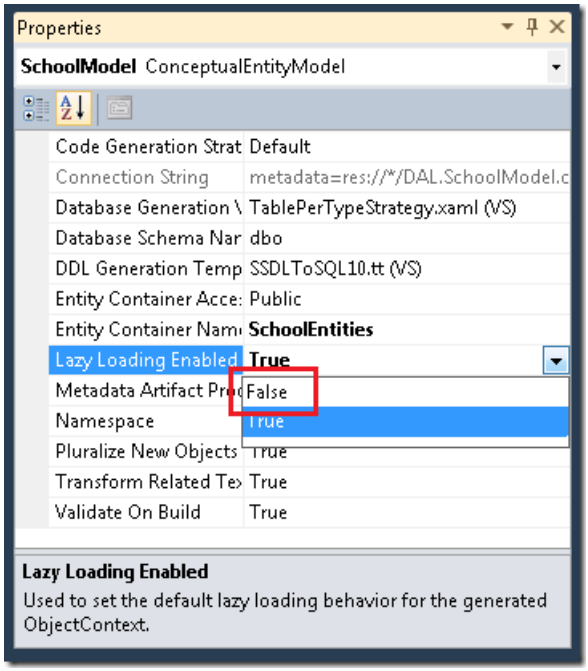
"

The `EntityDataSource` control disables lazy loading by default. For the `ObjectDataSource` control that you're using for the current tutorial (or if you access the object context from page code), there are several ways you can make lazy loading disabled by default. You can disable it when you instantiate an object context. For example, you can add the following line to the constructor method of the `SchoolRepository` class:

```
context.ContextOptions.LazyLoadingEnabled = false;
```

For the Contoso University application, you'll make the object context automatically disable lazy loading so that this property doesn't have to be set whenever a context is instantiated.

Open the *SchoolModel.edmx* data model, click the design surface, and then in the properties pane set the **Lazy Loading Enabled** property to `False`. Save and close the data model.



Managing View State

In order to provide update functionality, an ASP.NET web page must store the original property values of an entity when a page is rendered. During postback processing the control can re-create the original state of the entity and call the entity's `Attach` method before applying changes and calling the `SaveChanges` method. By default, ASP.NET Web Forms data controls use view state to store the original values. However, view state can affect performance, because it's stored in hidden fields that can substantially increase the size of the page that's sent to and from the browser.

Techniques for managing view state, or alternatives such as session state, aren't unique to the Entity Framework, so this tutorial doesn't go into this topic in detail. For more information see the links at the end of the tutorial.

However, version 4 of ASP.NET provides a new way of working with view state that every ASP.NET developer of

Web Forms applications should be aware of: the `ViewStateMode` property. This new property can be set at the page or control level, and it enables you to disable view state by default for a page and enable it only for controls that need it.

For applications where performance is critical, a good practice is to always disable view state at the page level and enable it only for controls that require it. The size of view state in the Contoso University pages wouldn't be substantially decreased by this method, but to see how it works, you'll do it for the *Instructors.aspx* page. That page contains many controls, including a `Label` control that has view state disabled. None of the controls on this page actually need to have view state enabled. (The `DataKeyNames` property of the `GridView` control specifies state that must be maintained between postbacks, but these values are kept in control state, which isn't affected by the `ViewStateMode` property.)

The `Page` directive and `Label` control markup currently resembles the following example:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
    CodeBehind="Instructors.aspx.cs" Inherits="ContosoUniversity.Instructors" %>
    ...
    <asp:Label ID="ErrorMessageLabel" runat="server" Text="" Visible="false" ViewStateMode="Disabled">
</asp:Label>
    ...
```

Make the following changes:

- Add `ViewStateMode="Disabled"` to the `Page` directive.
- Remove `ViewStateMode="Disabled"` from the `Label` control.

The markup now resembles the following example:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
    CodeBehind="Instructors.aspx.cs" Inherits="ContosoUniversity.Instructors"
    ViewStateMode="Disabled" %>
    ...
    <asp:Label ID="ErrorMessageLabel" runat="server" Text="" Visible="false"></asp:Label>
    ...
```

View state is now disabled for all controls. If you later add a control that does need to use view state, all you need to do is include the `ViewStateMode="Enabled"` attribute for that control.

Using The NoTracking Merge Option

When an object context retrieves database rows and creates entity objects that represent them, by default it also tracks those entity objects using its object state manager. This tracking data acts as a cache and is used when you update an entity. Because a web application typically has short-lived object context instances, queries often return data that doesn't need to be tracked, because the object context that reads them will be disposed before any of the entities it reads are used again or updated.

In the Entity Framework, you can specify whether the object context tracks entity objects by setting a *merge option*. You can set the merge option for individual queries or for entity sets. If you set it for an entity set, that means that you're setting the default merge option for all queries that are created for that entity set.

For the Contoso University application, tracking isn't needed for any of the entity sets that you access from the repository, so you can set the merge option to `NoTracking` for those entity sets when you instantiate the object context in the repository class. (Note that in this tutorial, setting the merge option won't have a noticeable effect on the application's performance. The `NoTracking` option is likely to make an observable performance improvement only in certain high-data-volume scenarios.)

In the DAL folder, open the *SchoolRepository.cs* file and add a constructor method that sets the merge option for the entity sets that the repository accesses:

```
public SchoolRepository()
{
    context.Departments.MergeOption = MergeOption.NoTracking;
    context.InstructorNames.MergeOption = MergeOption.NoTracking;
    context.OfficeAssignments.MergeOption = MergeOption.NoTracking;
}
```

Pre-Compiling LINQ Queries

The first time that the Entity Framework executes an Entity SQL query within the life of a given `ObjectContext` instance, it takes some time to compile the query. The result of compilation is cached, which means that subsequent executions of the query are much quicker. LINQ queries follow a similar pattern, except that some of the work required to compile the query is done every time the query is executed. In other words, for LINQ queries, by default not all of the results of compilation are cached.

If you have a LINQ query that you expect to run repeatedly in the life of an object context, you can write code that causes all of the results of compilation to be cached the first time the LINQ query is run.

As an illustration, you'll do this for two `Get` methods in the `SchoolRepository` class, one of which doesn't take any parameters (the `GetInstructorNames` method), and one that does require a parameter (the `GetDepartmentsByAdministrator` method). These methods as they stand now actually don't need to be compiled because they aren't LINQ queries:

```
public IEnumerable<InstructorName> GetInstructorNames()
{
    return context.InstructorNames.OrderBy("it.FullName").ToList();
}
```

```
public IEnumerable<Department> GetDepartmentsByAdministrator(Int32 administrator)
{
    return new ObjectQuery<Department>("SELECT VALUE d FROM Departments as d", context,
MergeOption.NoTracking).Include("Person").Where(d => d.Administrator == administrator).ToList();
}
```

However, so that you can try out compiled queries, you'll proceed as if these had been written as the following LINQ queries:

```
public IEnumerable<InstructorName> GetInstructorNames()
{
    return (from i in context.InstructorNames orderby i.FullName select i).ToList();
}
```

```
public IEnumerable<Department> GetDepartmentsByAdministrator(Int32 administrator)
{
    context.Departments.MergeOption = MergeOption.NoTracking;
    return (from d in context.Departments where d.Administrator == administrator select d).ToList();
}
```

You could change the code in these methods to what's shown above and run the application to verify that it works before continuing. But the following instructions jump right into creating pre-compiled versions of them.

Create a class file in the *DAL* folder, name it *SchoolEntities.cs*, and replace the existing code with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data.Objects;

namespace ContosoUniversity.DAL
{
    public partial class SchoolEntities
    {
        private static readonly Func<SchoolEntities, IQueryable<InstructorName>> compiledInstructorNamesQuery
=
            CompiledQuery.Compile((SchoolEntities context) => from i in context.InstructorNames orderby
i.FullName select i);

        public IEnumerable<InstructorName> CompiledInstructorNamesQuery()
        {
            return compiledInstructorNamesQuery(this).ToList();
        }

        private static readonly Func<SchoolEntities, Int32, IQueryable<Department>>
compiledDepartmentsByAdministratorQuery =
            CompiledQuery.Compile((SchoolEntities context, Int32 administrator) => from d in
context.Departments.Include("Person") where d.Administrator == administrator select d);

        public IEnumerable<Department> CompiledDepartmentsByAdministratorQuery(Int32 administrator)
        {
            return compiledDepartmentsByAdministratorQuery(this, administrator).ToList();
        }
    }
}
```

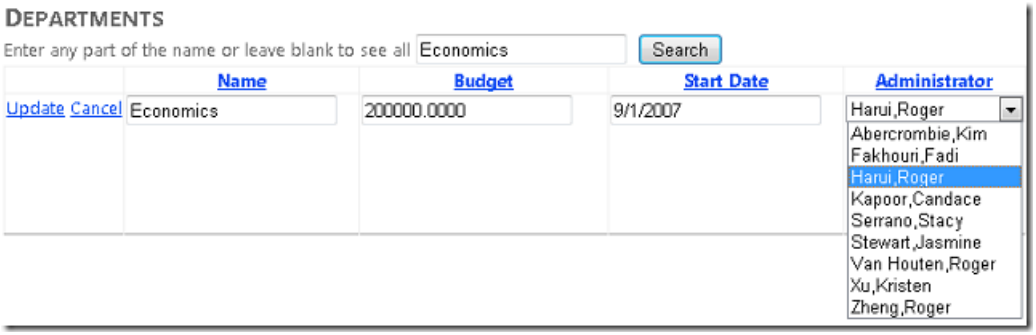
This code creates a partial class that extends the automatically generated object context class. The partial class includes two compiled LINQ queries using the `Compile` method of the `CompiledQuery` class. It also creates methods that you can use to call the queries. Save and close this file.

Next, in *SchoolRepository.cs*, change the existing `GetInstructorNames` and `GetDepartmentsByAdministrator` methods in the repository class so that they call the compiled queries:

```
public IEnumerable<InstructorName> GetInstructorNames()
{
    return context.CompiledInstructorNamesQuery();
}
```

```
public IEnumerable<Department> GetDepartmentsByAdministrator(Int32 administrator)
{
    return context.CompiledDepartmentsByAdministratorQuery(administrator);
}
```

Run the *Departments.aspx* page to verify that it works as it did before. The `GetInstructorNames` method is called in order to populate the administrator drop-down list, and the `GetDepartmentsByAdministrator` method is called when you click **Update** in order to verify that no instructor is an administrator of more than one department.



You've pre-compiled queries in the Contoso University application only to see how to do it, not because it would measurably improve performance. Pre-compiling LINQ queries does add a level of complexity to your code, so make sure you do it only for queries that actually represent performance bottlenecks in your application.

Examining Queries Sent to the Database

When you're investigating performance issues, sometimes it's helpful to know the exact SQL commands that the Entity Framework is sending to the database. If you're working with an `IQueryable` object, one way to do this is to use the `ToTraceString` method.

In *SchoolRepository.cs*, change the code in the `GetDepartmentsByName` method to match the following example:

```
public IEnumerable<Department> GetDepartmentsByName(string sortExpression, string nameSearchString)
{
    ...
    var departments = new ObjectQuery<Department>("SELECT VALUE d FROM Departments AS d",
context).OrderBy("it." + sortExpression).Include("Person").Include("Courses").Where(d =>
d.Name.Contains(nameSearchString));
    string commandText = ((ObjectQuery)departments).ToTraceString();
    return departments.ToList();
}
```

The `departments` variable must be cast to an `ObjectQuery` type only because the `Where` method at the end of the preceding line creates an `IQueryable` object; without the `Where` method, the cast would not be necessary.

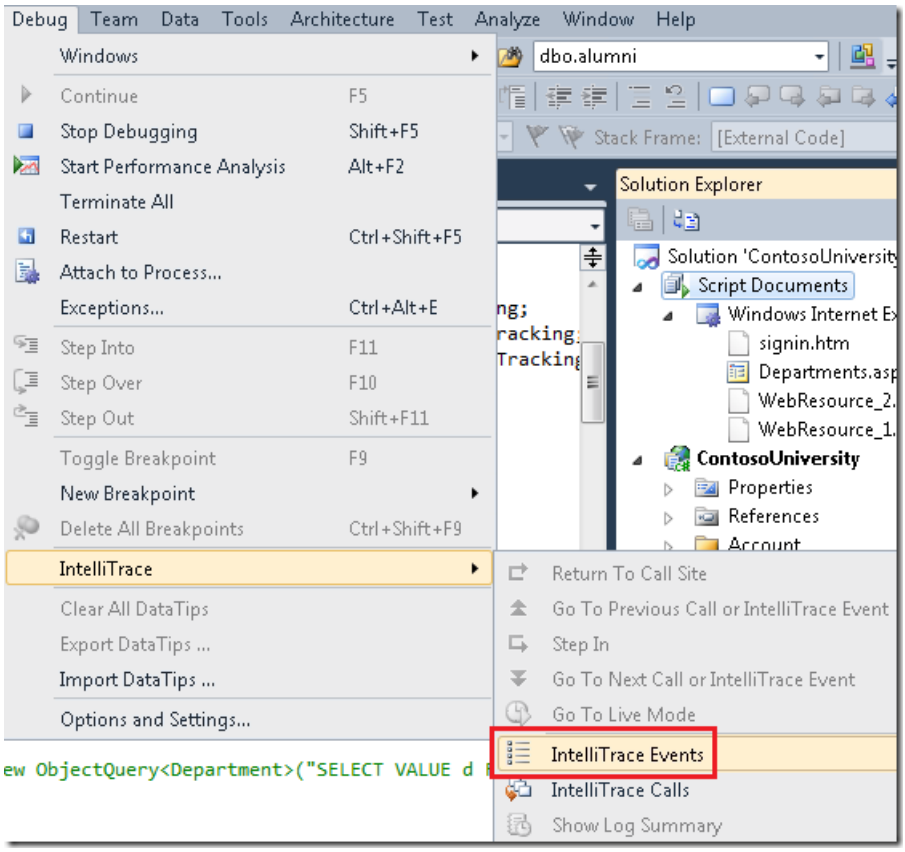
Set a breakpoint on the `return` line, and then run the *Departments.aspx* page in the debugger. When you hit the breakpoint, examine the `commandText` variable in the **Locals** window and use the text visualizer (the magnifying glass in the **Value** column) to display its value in the **Text Visualizer** window. You can see the entire SQL command that results from this code:

As an alternative, the IntelliTrace feature in Visual Studio Ultimate provides a way to view SQL commands generated by the Entity Framework that doesn't require you to change your code or even set a breakpoint.

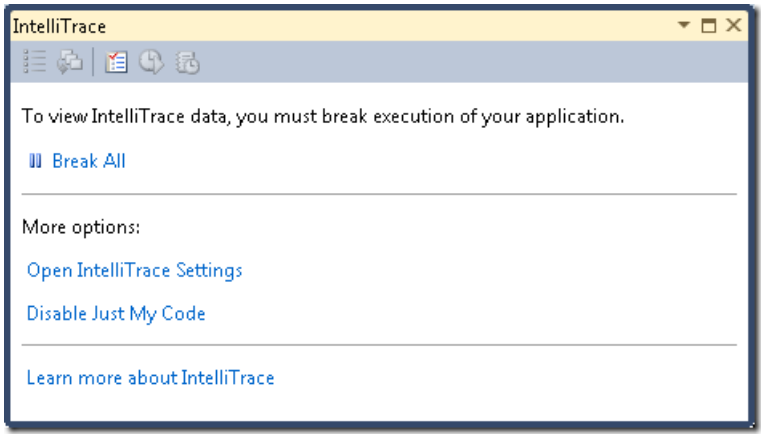
> **NOTE**
>
> You can perform the following procedures only if you have Visual Studio Ultimate.

Restore the original code in the `GetDepartmentsByName` method, and then run the *Departments.aspx* page in the debugger.

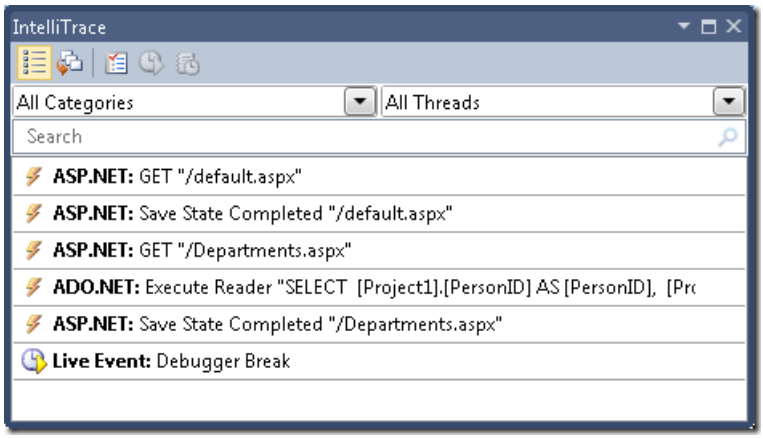
In Visual Studio, select the **Debug** menu, then **IntelliTrace**, and then **IntelliTrace Events**.



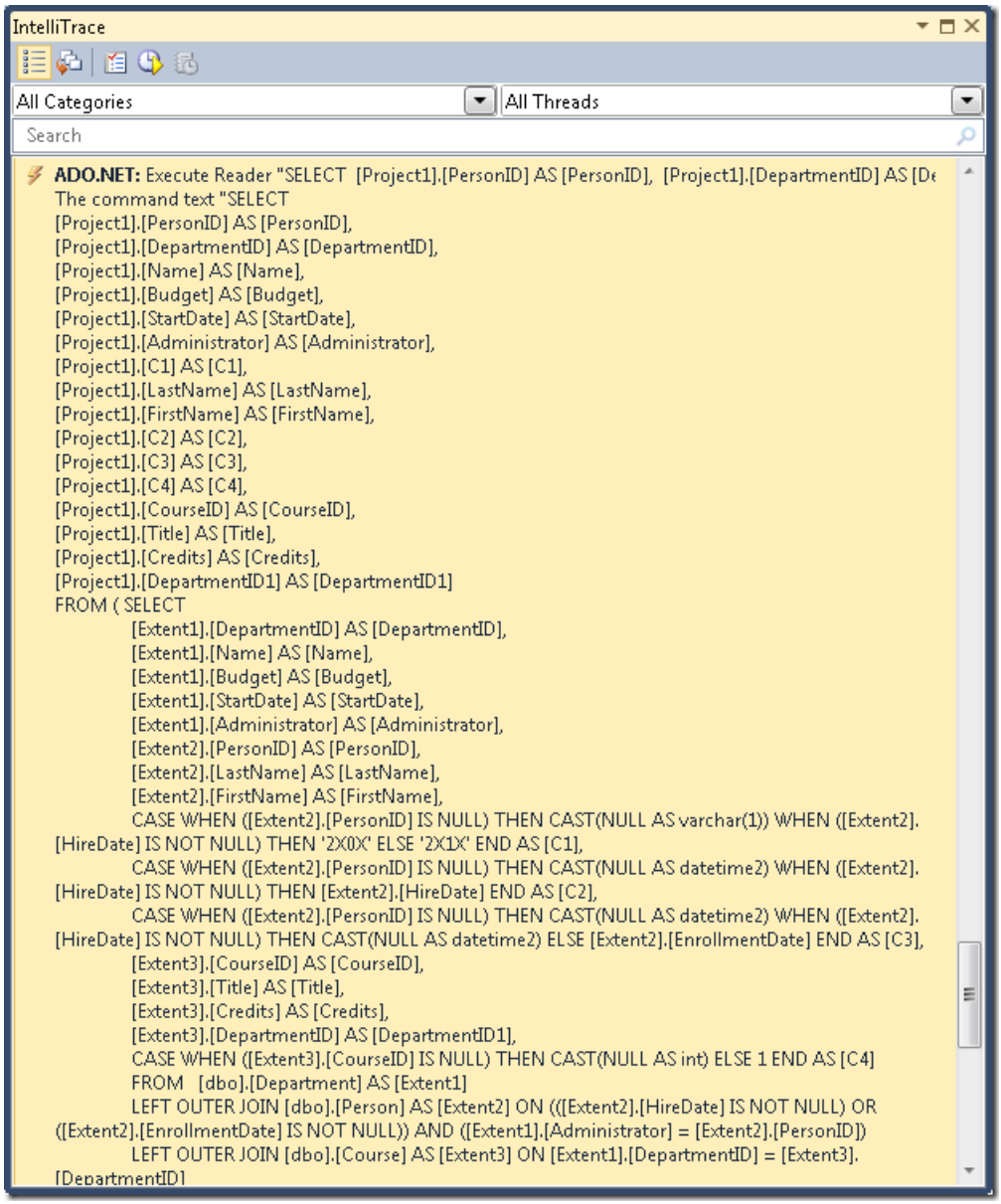
In the **IntelliTrace** window, click **Break All**.



The **IntelliTrace** window displays a list of recent events:



Click the **ADO.NET** line. It expands to show you the command text:



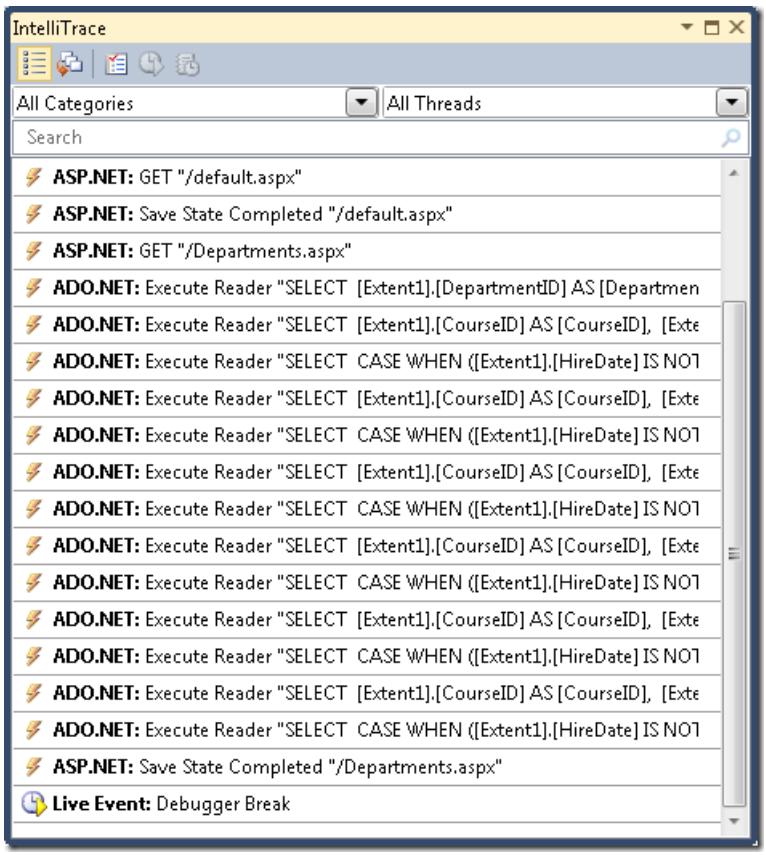
You can copy the entire command text string to the clipboard from the **Locals** window.

Suppose you were working with a database with more tables, relationships, and columns than the simple `School` database. You might find that a query that gathers all the information you need in a single `Select` statement containing multiple `Join` clauses becomes too complex to work efficiently. In that case you can switch from eager loading to explicit loading to simplify the query.

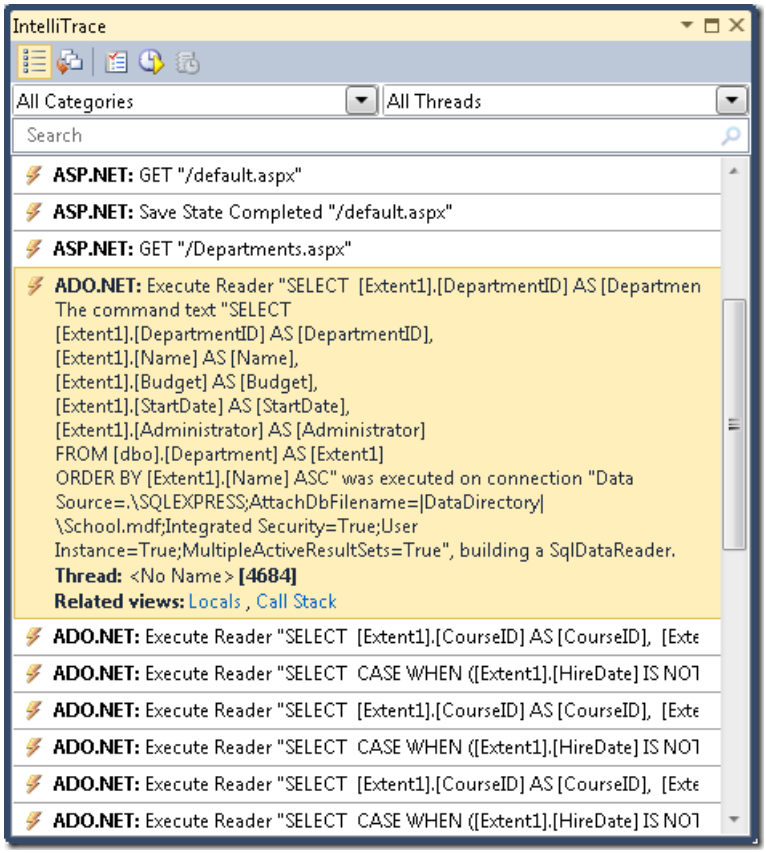
For example, try changing the code in the `GetDepartmentsByName` method in *SchoolRepository.cs*. Currently in that method you have an object query that has `Include` methods for the `Person` and `Courses` navigation properties. Replace the `return` statement with code that performs explicit loading, as shown in the following example:

```
public IEnumerable<Department> GetDepartmentsByName(string sortExpression, string nameSearchString)
{
    ...
    var departments = new ObjectQuery<Department>("SELECT VALUE d FROM Departments AS d",
context).OrderBy("it." + sortExpression).Where(d => d.Name.Contains(nameSearchString)).ToList();
    foreach (Department d in departments)
    {
        d.Courses.Load();
        d.PersonReference.Load();
    }
    return departments;
}
```

Run the *Departments.aspx* page in the debugger and check the **IntelliTrace** window again as you did before. Now, where there was a single query before, you see a long sequence of them.



Click the first **ADO.NET** line to see what has happened to the complex query you viewed earlier.



The query from Departments has become a simple `Select` query with no `Join` clause, but it's followed by separate queries that retrieve related courses and an administrator, using a set of two queries for each department returned by the original query.

NOTE

If you leave lazy loading enabled, the pattern you see here, with the same query repeated many times, might result from lazy loading. A pattern that you typically want to avoid is lazy-loading related data for every row of the primary table. Unless you've verified that a single join query is too complex to be efficient, you'd typically be able to improve performance in such cases by changing the primary query to use eager loading.

Pre-Generating Views

When an `ObjectContext` object is first created in a new application domain, the Entity Framework generates a set of classes that it uses to access the database. These classes are called *views*, and if you have a very large data model, generating these views can delay the web site's response to the first request for a page after a new application domain is initialized. You can reduce this first-request delay by creating the views at compile time rather than at run time.

NOTE

If your application doesn't have an extremely large data model, or if it does have a large data model but you aren't concerned about a performance problem that affects only the very first page request after IIS is recycled, you can skip this section. View creation doesn't happen every time you instantiate an `ObjectContext` object, because the views are cached in the application domain. Therefore, unless you're frequently recycling your application in IIS, very few page requests would benefit from pre-generated views.

You can pre-generate views using the *EdmGen.exe* command-line tool or by using a *Text Template Transformation Toolkit* (T4) template. In this tutorial you'll use a T4 template.

In the *DAL* folder, add a file using the **Text Template** template (it's under the **General** node in the **Installed Templates** list), and name it *SchoolModel.Views.tt*. Replace the existing code in the file with the following code:

```
<#
/*******************************************************************************

Copyright (c) Microsoft Corporation. All rights reserved.

THIS CODE IS PROVIDED *AS IS* WITHOUT WARRANTY OF
ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING ANY
IMPLIED WARRANTIES OF FITNESS FOR A PARTICULAR
PURPOSE, MERCHANTABILITY, OR NON-INFRINGEMENT.

*******************************************************************************/
#>

<#
    //
    // TITLE: T4 template to generate views for an EDMX file in a C# project
    //
    // DESCRIPTION:
    // This is a T4 template to generate views in C# for an EDMX file in C# projects.
    // The generated views are automatically compiled into the project's output assembly.
    //
    // This template follows a simple file naming convention to determine the EDMX file to process:
    // - It assumes that [edmx-file-name].Views.tt will process and generate views for [edmx-file-name].EDMX
    // - The views are generated in the code behind file [edmx-file-name].Views.cs
    //
    // USAGE:
    // Do the following to generate views for an EDMX file (e.g. Model1.edmx) in a C# project
    // 1. In Solution Explorer, right-click the project node and choose "Add...Existing...Item" from the
context menu
    // 2. Browse to and choose this .tt file to include it in the project
```

```
    // 3. Ensure this .tt file is in the same directory as the EDMX file to process
    // 4. In Solution Explorer, rename this .tt file to the form [edmx-file-name].Views.tt (e.g.
Model1.Views.tt)
    // 5. In Solution Explorer, right-click Model1.Views.tt and choose "Run Custom Tool" to generate the
views
    // 6. The views are generated in the code behind file Model1.Views.cs
    //
    // TIPS:
    // If you have multiple EDMX files in your project then make as many copies of this .tt file and rename
appropriately
    // to pair each with each EDMX file.
    //
    // To generate views for all EDMX files in the solution, click the "Transform All Templates" button in
the Solution Explorer toolbar
    // (its the rightmost button in the toolbar)
    //
#>
<#
    //
    // T4 template code follows
    //
#>
<#@ template language="C#" hostspecific="true"#>
<#@ include file="EF.Utility.CS.ttinclude"#>
<#@ output extension=".cs" #>
<#
    // Find EDMX file to process: Model1.Views.tt generates views for Model1.EDMX
    string edmxFileName =
Path.GetFileNameWithoutExtension(this.Host.TemplateFile).ToLowerInvariant().Replace(".views", "") + ".edmx";
    string edmxFilePath = Path.Combine(Path.GetDirectoryName(this.Host.TemplateFile), edmxFileName);
    if (File.Exists(edmxFilePath))
    {
        // Call helper class to generate pre-compiled views and write to output
        this.WriteLine(GenerateViews(edmxFilePath));
    }
    else
    {
        this.Error(String.Format("No views were generated. Cannot find file {0}. Ensure the project has an
EDMX file and the file name of the .tt file is of the form [edmx-file-name].Views.tt", edmxFilePath));
    }

    // All done!
#>

<#+
    private String GenerateViews(string edmxFilePath)
    {
        MetadataLoader loader = new MetadataLoader(this);
        MetadataWorkspace workspace;
        if(!loader.TryLoadAllMetadata(edmxFilePath, out workspace))
        {
            this.Error("Error in the metadata");
            return String.Empty;
        }

        String generatedViews = String.Empty;
        try
        {
            using (StreamWriter writer = new StreamWriter(new MemoryStream()))
            {
                StorageMappingItemCollection mappingItems =
(StorageMappingItemCollection)workspace.GetItemCollection(DataSpace.CSSpace);

                // Initialize the view generator to generate views in C#
                EntityViewGenerator viewGenerator = new EntityViewGenerator();
                viewGenerator.LanguageOption = LanguageOption.GenerateCSharpCode;
                IList<EdmSchemaError> errors = viewGenerator.GenerateViews(mappingItems, writer);

                foreach (EdmSchemaError e in errors)
```

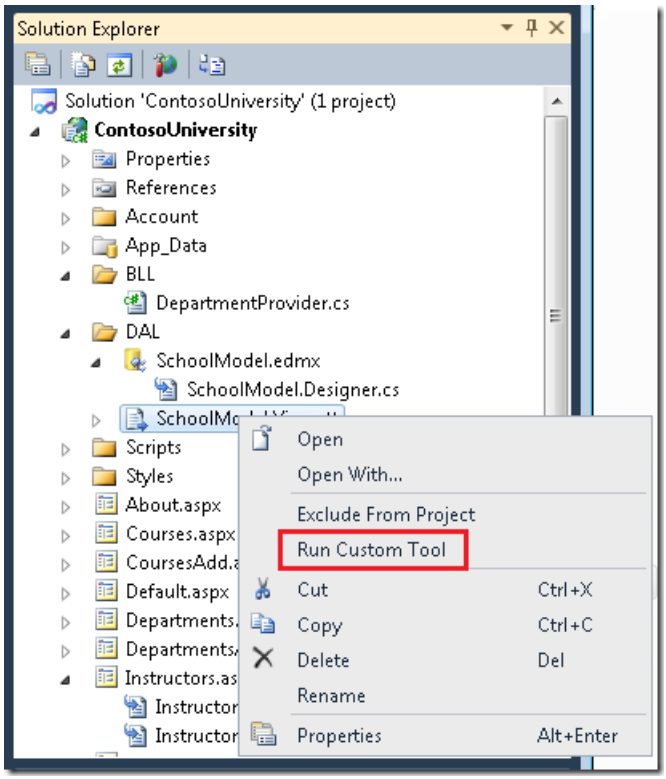
```
                {
                    // log error
                    this.Error(e.Message);
                }

                MemoryStream memStream = writer.BaseStream as MemoryStream;
                generatedViews = Encoding.UTF8.GetString(memStream.ToArray());
            }
        }
        catch (Exception ex)
        {
            // log error
            this.Error(ex.ToString());
        }

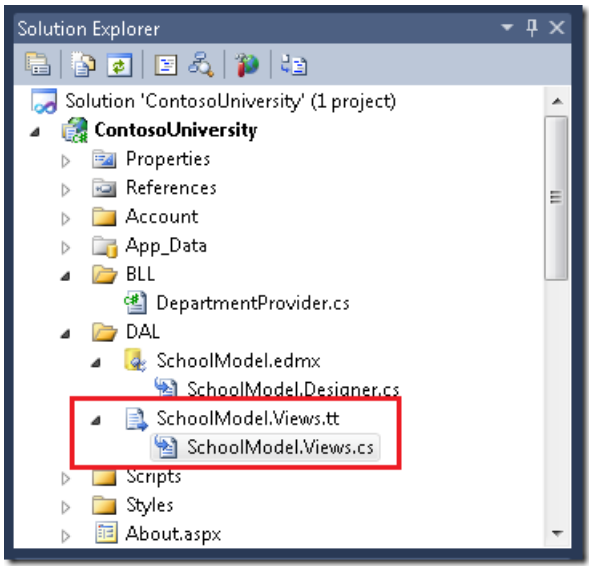
        return generatedViews;
    }
#>
```

This code generates views for an *.edmx* file that's located in the same folder as the template and that has the same name as the template file. For example, if your template file is named *SchoolModel.Views.tt*, it will look for a data model file named *SchoolModel.edmx*.

Save the file, then right-click the file in **Solution Explorer** and select **Run Custom Tool**.



Visual Studio generates a code file that creates the views, which is named *SchoolModel.Views.cs* based on the template. (You might have noticed that the code file is generated even before you select **Run Custom Tool**, as soon as you save the template file.)



You can now run the application and verify that it works as it did before.

For more information about pre-generated views, see the following resources:

- How to: Pre-Generate Views to Improve Query Performance on the MSDN web site. Explains how to use the `EdmGen.exe` command-line tool to pre-generate views.
- Isolating Performance with Precompiled/Pre-generated Views in the Entity Framework 4 on the Windows Server AppFabric Customer Advisory Team blog.

This completes the introduction to improving performance in an ASP.NET web application that uses the Entity Framework. For more information, see the following resources:

- Performance Considerations (Entity Framework) on the MSDN web site.
- Performance-related posts on the Entity Framework Team blog.
- EF Merge Options and Compiled Queries. Blog post that explains unexpected behaviors of compiled queries and merge options such as `NoTracking`. If you plan to use compiled queries or manipulate merge option settings in your application, read this first.
- Entity Framework-related posts in the Data and Modeling Customer Advisory Team blog. Includes posts on compiled queries and using the Visual Studio 2010 Profiler to discover performance issues.
- Entity Framework forum thread with advice on improving performance of highly complex queries.
- ASP.NET State Management Recommendations.
- Using the Entity Framework and the ObjectDataSource: Custom Paging. Blog post that builds on the ContosoUniversity application created in these tutorials to explain how to implement paging in the *Departments.aspx* page.

The next tutorial reviews some of the important enhancements to the Entity Framework that are new in version 4.

PREVIOUS NEXT

What's New in the Entity Framework 4.0

2/20/2020 • 11 minutes to read • Edit Online

by Tom Dykstra

This tutorial series builds on the Contoso University web application that is created by the Getting Started with the Entity Framework tutorial series. If you didn't complete the earlier tutorials, as a starting point for this tutorial you can download the application that you would have created. You can also download the application that is created by the complete tutorial series. If you have questions about the tutorials, you can post them to the ASP.NET Entity Framework forum.

In the previous tutorial you saw some methods for maximizing the performance of a web application that uses the Entity Framework. This tutorial reviews some of the most important new features in version 4 of the Entity Framework, and it links to resources that provide a more complete introduction to all of the new features. The features highlighted in this tutorial include the following:

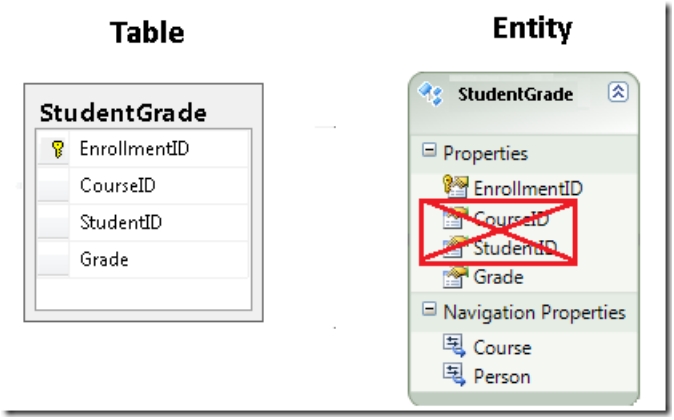
- Foreign-key associations.
- Executing user-defined SQL commands.
- Model-first development.
- POCO support.

In addition, the tutorial will briefly introduce *code-first development*, a feature that's coming in the next release of the Entity Framework.

To start the tutorial, start Visual Studio and open the Contoso University web application that you were working with in the previous tutorial.

Foreign-Key Associations

Version 3.5 of the Entity Framework included navigation properties, but it didn't include foreign-key properties in the data model. For example, the `CourseID` and `StudentID` columns of the `StudentGrade` table would be omitted from the `StudentGrade` entity.



The reason for this approach was that, strictly speaking, foreign keys are a physical implementation detail and don't belong in a conceptual data model. However, as a practical matter, it's often easier to work with entities in code when you have direct access to the foreign keys.

For an example of how foreign keys in the data model can simplify your code, consider how you would have had to code the *DepartmentsAdd.aspx* page without them. In the `Department` entity, the `Administrator` property is a

foreign key that corresponds to `PersonID` in the `Person` entity. In order to establish the association between a new department and its administrator, all you had to do was set the value for the `Administrator` property in the `ItemInserting` event handler of the databound control:

```
protected void DepartmentsDetailsView_ItemInserting(object sender, DetailsViewInsertEventArgs e)
{
    e.Values["Administrator"] = administratorsDropDownList.SelectedValue;
}
```

Without foreign keys in the data model, you'd handle the `Inserting` event of the data source control instead of the `ItemInserting` event of the databound control, in order to get a reference to the entity itself before the entity is added to the entity set. When you have that reference, you establish the association using code like that in the following examples:

```
departmentEntityToBeInserted.PersonReference.EntityKey = new
System.Data.EntityKey("SchoolEntities.Departments", "PersonID",
Convert.ToInt32(administratorsDropDownList.SelectedValue));
```

```
departmentEntityToBeInserted.Person = context.People.Single(p => p.PersonID ==
Convert.ToInt32(administratorsDropDownList.SelectedValue));
```

As you can see in the Entity Framework team's blog post on Foreign Key associations, there are other cases where the difference in code complexity is much greater. To meet the needs of those who prefer to live with implementation details in the conceptual data model for the sake of simpler code, the Entity Framework now gives you the option of including foreign keys in the data model.

In Entity Framework terminology, if you include foreign keys in the data model you're using *foreign key associations*, and if you exclude foreign keys you're using *independent associations*.

Executing User-Defined SQL Commands

In earlier versions of the Entity Framework, there was no easy way to create your own SQL commands on the fly and run them. Either the Entity Framework dynamically generated SQL commands for you, or you had to create a stored procedure and import it as a function. Version 4 adds `ExecuteStoreQuery` and `ExecuteStoreCommand` methods the `ObjectContext` class that make it easier for you to pass any query directly to the database.

Suppose Contoso University administrators want to be able to perform bulk changes in the database without having to go through the process of creating a stored procedure and importing it into the data model. Their first request is for a page that lets them change the number of credits for all courses in the database. On the web page, they want to be able to enter a number to use to multiply the value of every `Course` row's `Credits` column.

Create a new page that uses the *Site.Master* master page and name it *UpdateCredits.aspx*. Then add the following markup to the `Content` control named `Content2`:

```
<h2>Update Credits</h2>
    Enter the number to multiply the current number of credits by:
    <asp:TextBox ID="CreditsMultiplierTextBox" runat="server"></asp:TextBox>
    <br /><br />
    <asp:Button ID="ExecuteButton" runat="server" Text="Execute" OnClick="ExecuteButton_Click" /><br /><br />
    Rows affected:
    <asp:Label ID="RowsAffectedLabel" runat="server" Text="0" ViewStateMode="Disabled"></asp:Label><br /><br
/>
```

This markup creates a `TextBox` control in which the user can enter the multiplier value, a `Button` control to click

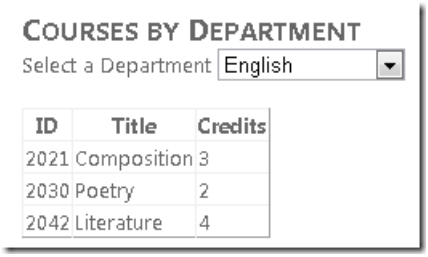
in order to execute the command, and a `Label` control for indicating the number of rows affected.

Open *UpdateCredits.aspx.cs*, and add the following `using` statement and a handler for the button's `Click` event:

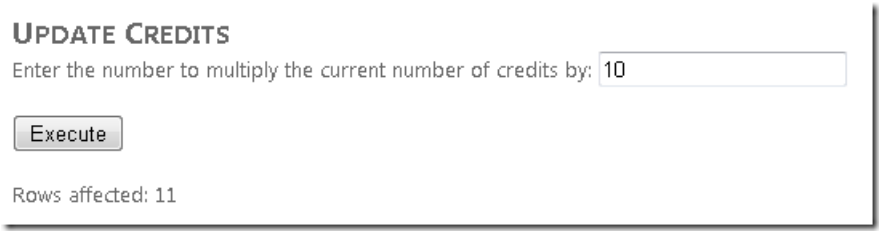
```
using ContosoUniversity.DAL;
```

```
protected void ExecuteButton_Click(object sender, EventArgs e)
{
    using (SchoolEntities context = new SchoolEntities())
    {
        RowsAffectedLabel.Text = context.ExecuteStoreCommand("UPDATE Course SET Credits = Credits * {0}",
CreditsMultiplierTextBox.Text).ToString();
    }
}
```

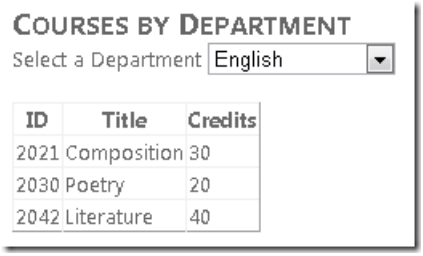
This code executes the SQL `Update` command using the value in the text box and uses the label to display the number of rows affected. Before you run the page, run the *Courses.aspx* page to get a "before" picture of some data.



Run *UpdateCredits.aspx*, enter "10" as the multiplier, and then click **Execute**.



Run the *Courses.aspx* page again to see the changed data.



(If you want to set the number of credits back to their original values, in *UpdateCredits.aspx.cs* change `Credits * {0}` to `Credits / {0}` and re-run the page, entering 10 as the divisor.)

For more information about executing queries that you define in code, see How to: Directly Execute Commands Against the Data Source.

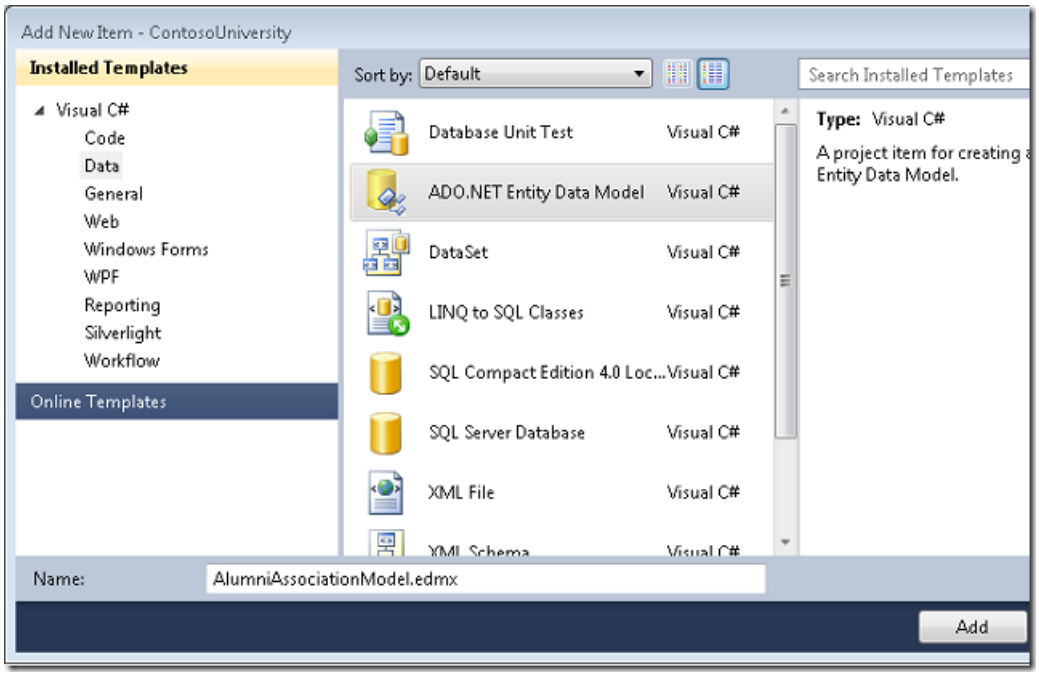
Model-First Development

In these walkthroughs you created the database first and then generated the data model based on the database structure. In the Entity Framework 4 you can start with the data model instead and generate the database based on the data model structure. If you're creating an application for which the database doesn't already exist, the

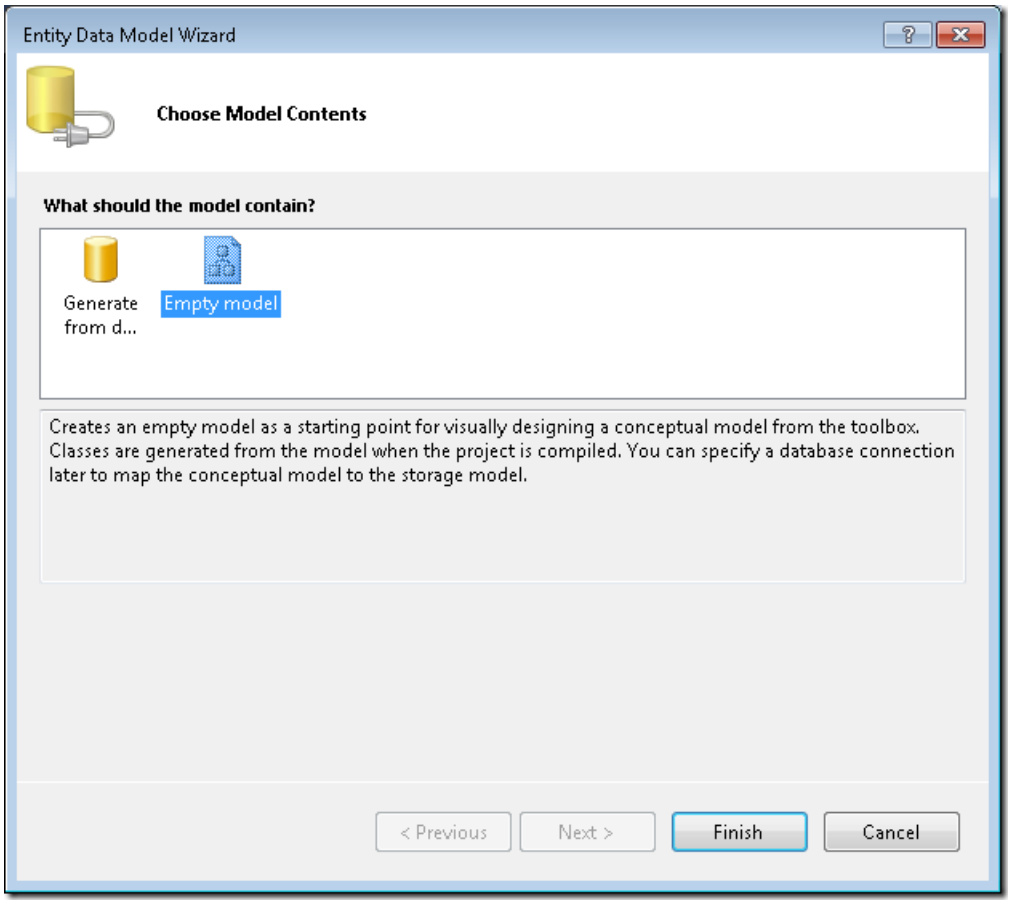
model-first approach enables you to create entities and relationships that make sense conceptually for the application, while not worrying about physical implementation details. (This remains true only through the initial stages of development, however. Eventually the database will be created and will have production data in it, and recreating it from the model will no longer be practical; at that point you'll be back to the database-first approach.)

In this section of the tutorial, you'll create a simple data model and generate the database from it.

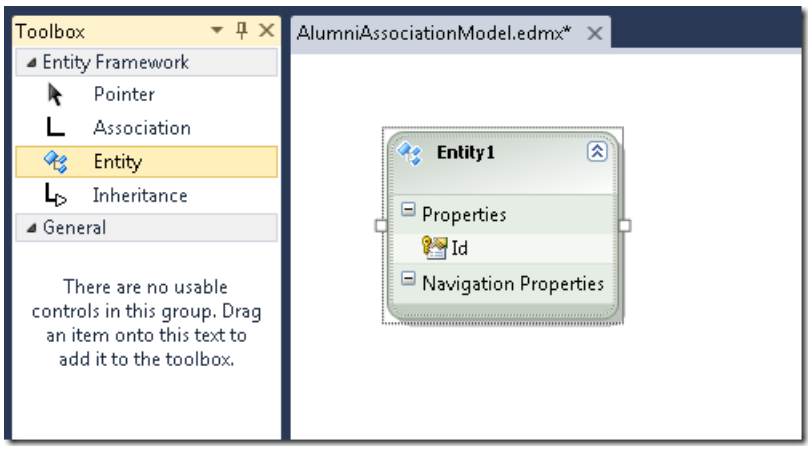
In **Solution Explorer**, right-click the *DAL* folder and select **Add New Item**. In the **Add New Item** dialog box, under **Installed Templates** select **Data** and then select the **ADO.NET Entity Data Model** template. Name the new file *AlumniAssociationModel.edmx* and click **Add**.



This launches the Entity Data Model Wizard. In the **Choose Model Contents** step, select **Empty Model** and then click **Finish**.

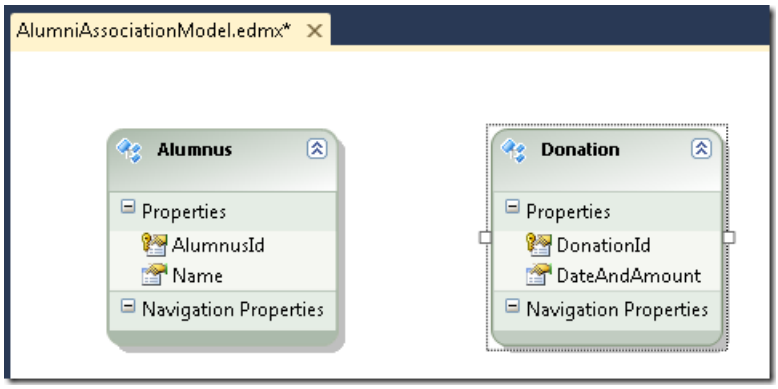


The **Entity Data Model Designer** opens with a blank design surface. Drag an **Entity** item from the **Toolbox** onto the design surface.

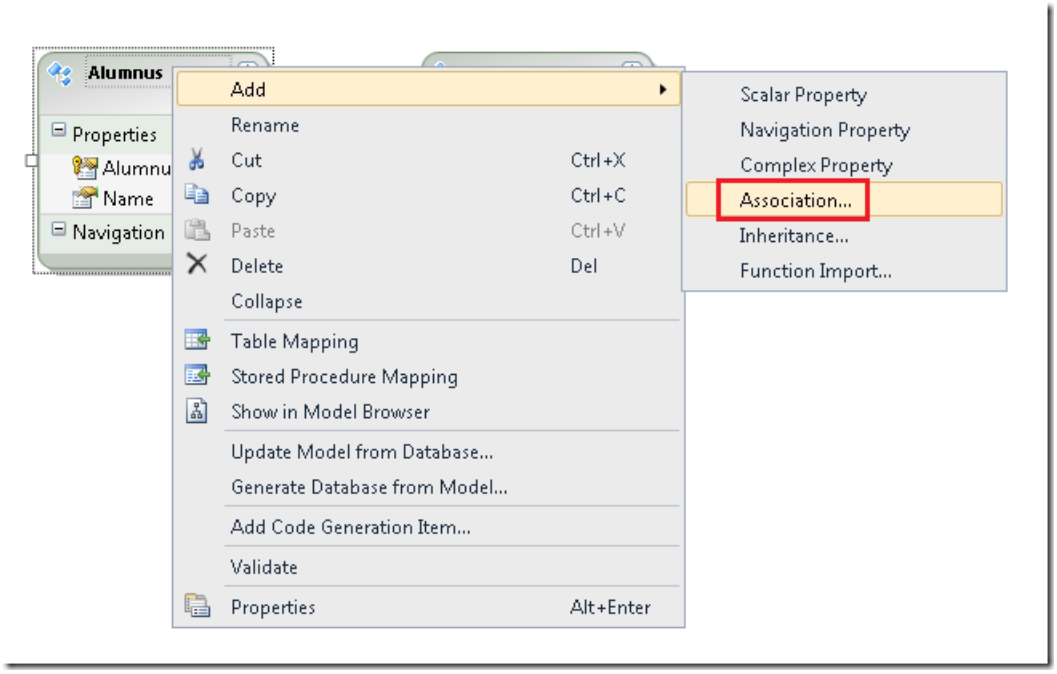


Change the entity name from `Entity1` to `Alumnus`, change the `Id` property name to `AlumnusId`, and add a new scalar property named `Name`. To add new properties you can press Enter after changing the name of the `Id` column, or right-click the entity and select **Add Scalar Property**. The default type for new properties is `String`, which is fine for this simple demonstration, but of course you can change things like data type in the **Properties** window.

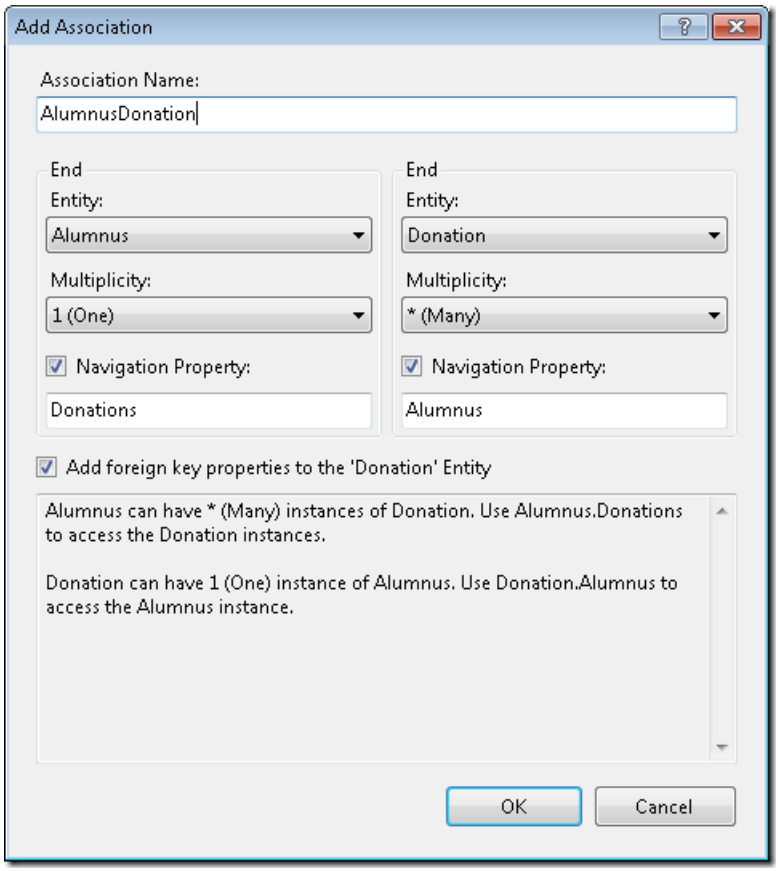
Create another entity the same way and name it `Donation`. Change the `Id` property to `DonationId` and add a scalar property named `DateAndAmount`.



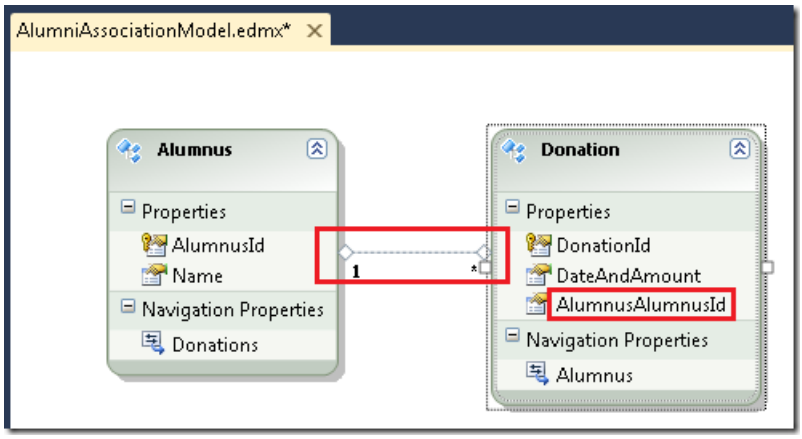
To add an association between these two entities, right-click the `Alumnus` entity, select **Add**, and then select **Association**.



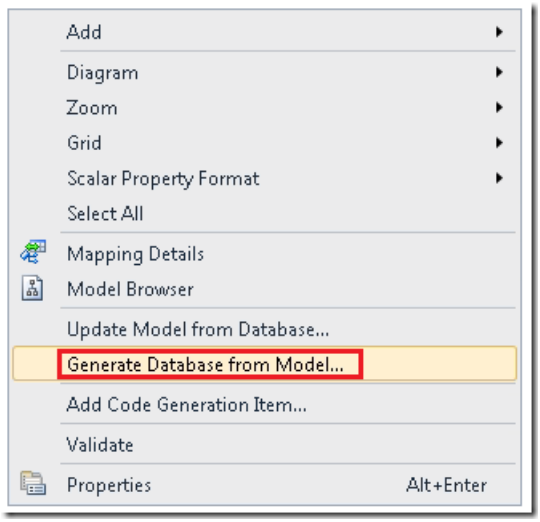
The default values in the **Add Association** dialog box are what you want (one-to-many, include navigation properties, include foreign keys), so just click **OK**.



The designer adds an association line and a foreign-key property.

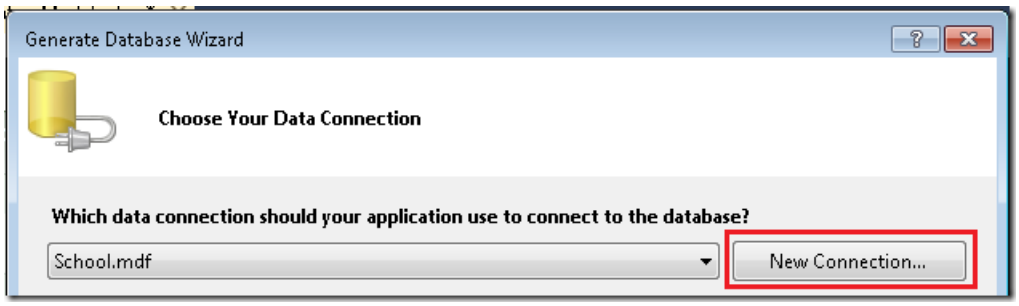


Now you're ready to create the database. Right-click the design surface and select **Generate Database from Model**.

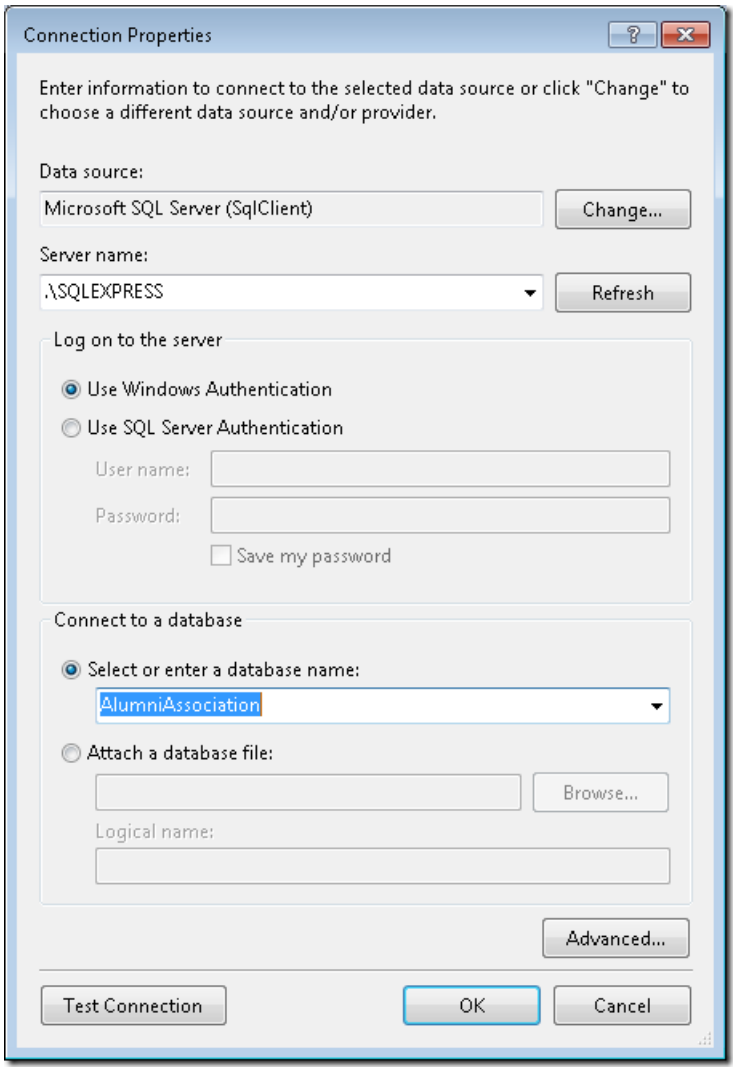


This launches the Generate Database Wizard. (If you see warnings that indicate that the entities aren't mapped, you can ignore those for the time being.)

In the **Choose Your Data Connection** step, click **New Connection**.

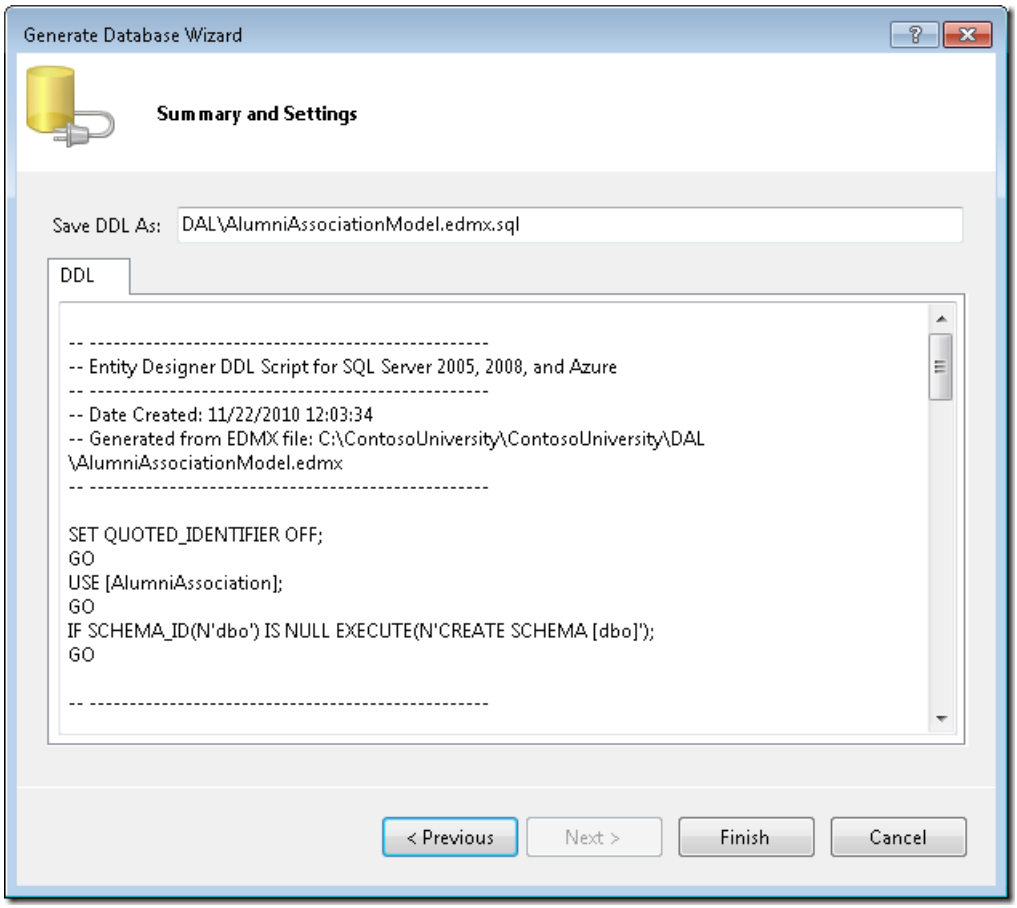


In the **Connection Properties** dialog box, select the local SQL Server Express instance and name the database `AlumniAssociation`.

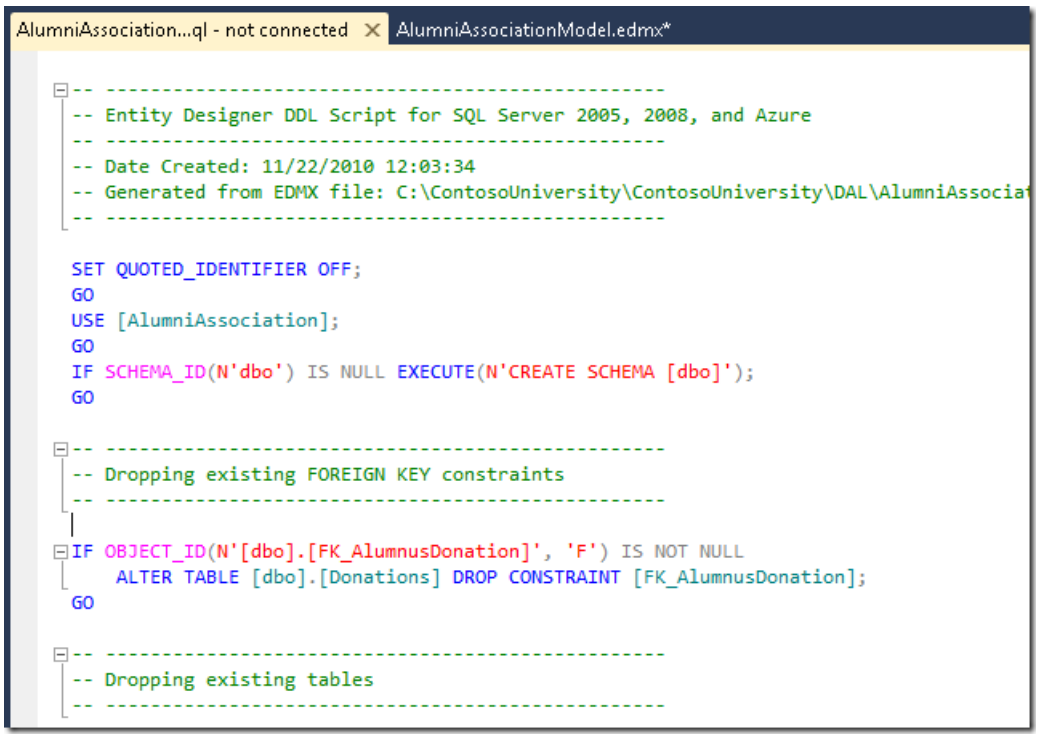


Click **Yes** when you're asked if you want to create the database. When the **Choose Your Data Connection** step is displayed again, click **Next**.

In the **Summary and Settings** step, click **Finish**.



A *.sql* file with the data definition language (DDL) commands is created, but the commands haven't been run yet.



Use a tool such as **SQL Server Management Studio** to run the script and create the tables, as you might have done when you created the `School` database for the first tutorial in the Getting Started tutorial series. (Unless you downloaded the database.)

You can now use the `AlumniAssociation` data model in your web pages the same way you've been using the `School` model. To try this out, add some data to the tables and create a web page that displays the data.

Using **Server Explorer**, add the following rows to the `Alumnus` and `Donation` tables.

AlumnusId	Name
1	Roger Zheng
2	Candace Kapoor

DonationId	DateAndAmo...	AlumnusAlumnusId
1	1/1/2010 $100	1
2	1/2/2010 $200	1
3	1/3/2010 $300	2

Create a new web page named *Alumni.aspx* that uses the *Site.Master* master page. Add the following markup to the `Content` control named `Content2`:

```
<h2>Alumni</h2>
    <asp:EntityDataSource ID="AlumniEntityDataSource" runat="server"
        ContextTypeName="ContosoUniversity.DAL.AlumniAssociationModelContainer" EnableFlattening="False"
        EntitySetName="Alumni">
    </asp:EntityDataSource>
    <asp:GridView ID="AlumniGridView" runat="server"
        DataSourceID="AlumniEntityDataSource" AutoGenerateColumns="False"
        OnRowDataBound="AlumniGridView_RowDataBound"
        DataKeyNames="AlumnusId">
        <Columns>
            <asp:BoundField DataField="Name" HeaderText="Name" SortExpression="Name" />
            <asp:TemplateField HeaderText="Donations">
                <ItemTemplate>
                    <asp:GridView ID="DonationsGridView" runat="server" AutoGenerateColumns="False">
                        <Columns>
                            <asp:BoundField DataField="DateAndAmount" HeaderText="Date and Amount" />
                        </Columns>
                    </asp:GridView>
                </ItemTemplate>
            </asp:TemplateField>
        </Columns>
    </asp:GridView>
```

This markup creates nested `GridView` controls, the outer one to display alumni names and the inner one to display donation dates and amounts.

Open *Alumni.aspx.cs*. Add a `using` statement for the data access layer and a handler for the outer `GridView` control's `RowDataBound` event:

```
using ContosoUniversity.DAL;

// ...

protected void AlumniGridView_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        var alumnus = e.Row.DataItem as Alumnus;
        var donationsGridView = (GridView)e.Row.FindControl("DonationsGridView");
        donationsGridView.DataSource = alumnus.Donations.ToList();
        donationsGridView.DataBind();
    }
}
```

This code databinds the inner `GridView` control using the `Donations` navigation property of the current row's `Alumnus` entity.

Run the page.

ALUMNI

Name	Donations
Roger Zheng	Date and Amount 1/1/2010 $100 1/2/2010 $200
Candace Kapoor	Date and Amount 1/3/2010 $300

(Note: This page is included in the downloadable project, but to make it work you must create the database in your local SQL Server Express instance; the database isn't included as an *.mdf* file in the *App_Data* folder.)

For more information about using the model-first feature of the Entity Framework, see Model-First in the Entity Framework 4.

POCO Support

When you use domain-driven design methodology, you design data classes that represent data and behavior that's relevant to the business domain. These classes should be independent of any specific technology used to store (persist) the data; in other words, they should be *persistence ignorant*. Persistence ignorance can also make a class easier to unit test because the unit test project can use whatever persistence technology is most convenient for testing. Earlier versions of the Entity Framework offered limited support for persistence ignorance because entity classes had to inherit from the `EntityObject` class and thus included a great deal of Entity Framework-specific functionality.

The Entity Framework 4 introduces the ability to use entity classes that don't inherit from the `EntityObject` class and therefore are persistence ignorant. In the context of the Entity Framework, classes like this are typically called *plain-old CLR objects* (POCO, or POCOs). You can write POCO classes manually, or you can automatically generate them based on an existing data model using Text Template Transformation Toolkit (T4) templates provided by the Entity Framework.

For more information about using POCOs in the Entity Framework, see the following resources:

- Working with POCO Entities. This is an MSDN document that's an overview of POCOs, with links to other documents that have more detailed information.
- Walkthrough: POCO Template for the Entity Framework This is a blog post from the Entity Framework development team, with links to other blog posts about POCOs.

Code-First Development

POCO support in the Entity Framework 4 still requires that you create a data model and link your entity classes to the data model. The next release of the Entity Framework will include a feature called *code-first development*. This feature enables you to use the Entity Framework with your own POCO classes without needing to use either the data model designer or a data model XML file. (Therefore, this option has also been called *code-only*, *code-first* and *code-only* both refer to the same Entity Framework feature.)

For more information about using the code-first approach to development, see the following resources:

- Code-First Development with Entity Framework 4. This is a blog post by Scott Guthrie introducing code-first development.
- Entity Framework Development Team Blog - posts tagged CodeOnly
- Entity Framework Development Team Blog - posts tagged Code First
- MVC Music Store tutorial - Part 4: Models and Data Access
- Getting Started with MVC 3 - Part 4: Entity Framework Code-First Development

In addition, a new MVC Code-First tutorial that builds an application similar to the Contoso University application

is projected to be published in the spring of 2011 at https://asp.net/entity-framework/tutorials

More Information

This completes the overview to what's new in the Entity Framework and this Continuing with the Entity Framework tutorial series. For more information about new features in the Entity Framework 4 that aren't covered here, see the following resources:

- What's New in ADO.NET MSDN topic on new features in version 4 of the Entity Framework.
- Announcing the release of Entity Framework 4 The Entity Framework development team's blog post about new features in version 4.

PREVIOUS

ASP.NET 4 - Web Deployment with SQL Server Compact using Visual Studio

2/20/2020 • 2 minutes to read • Edit Online

This tutorial series shows how to make an ASP.NET web application that uses SQL Server Compact available over the internet by deploying it to a third-party hosting provider. Requires Visual Studio 2012 RC or Visual Studio 2010. For more up-to-date information about deployment features, or for information about how to deploy SQL Server editions other than SQL Server Compact, see ASP.NET Web Deployment using Visual Studio.

- Visual Studio Web Deployment with SQL Server Compact - Introduction
- Visual Studio Web Deployment with SQL Server Compact- Deploying SQL Server Compact Databases
- Visual Studio Web Deployment with SQL Server Compact - Web.Config File Transformations
- Visual Studio Web Deployment with SQL Server Compact - Configuring Project Properties
- Visual Studio Web Deployment with SQL Server Compact - Deploying to IIS as a Test Environment
- Visual Studio Web Deployment with SQL Server Compact - Setting Folder Permissions
- Visual Studio Web Deployment with SQL Server Compact - Deploying to the Production Environment
- Visual Studio Web Deployment with SQL Server Compact - Deploying a Code-Only Update
- Visual Studio Web Deployment with SQL Server Compact - Deploying a Database Update
- Visual Studio Web Deployment with SQL Server Compact - Migrating to SQL Server
- Visual Studio Web Deployment with SQL Server Compact - Deploying a SQL Server Database Update
- Visual Studio Web Deployment with SQL Server Compact - Troubleshooting

Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio: Introduction - 1 of 12

2/20/2020 • 8 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

This series of tutorials shows you how to deploy (publish) an ASP.NET web application project that includes a SQL Server Compact database by using Visual Studio 2012 RC or Visual Studio Express 2012 RC for Web. You can also use Visual Studio 2010 if you install the Web Publish Update.

For a tutorial that shows deployment features introduced after the RC release of Visual Studio 2012, shows how to deploy SQL Server editions other than SQL Server Compact, and shows how to deploy to Azure App Service Web Apps, see ASP.NET Web Deployment using Visual Studio.

These tutorials guide you through deploying first to IIS on your local development computer for testing, and then to a third-party hosting provider. The application that you'll deploy uses an application database and an ASP.NET membership database. You start off using SQL Server Compact and deploying to SQL Server Compact, and later tutorials show you how to deploy database changes and how to migrate to SQL Server.

The tutorials assume you know how to work with ASP.NET in Visual Studio. If you don't, a good place to start is a basic ASP.NET Web Forms Tutorial or a basic ASP.NET MVC Tutorial.

If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET Deployment forum.

Overview

These tutorials guide you through deploying first to IIS on your local development computer for testing, and then to a third-party hosting provider. The application that you'll deploy uses an application database and an ASP.NET membership database. You start off using SQL Server Compact and deploying to SQL Server Compact, and later tutorials show you how to deploy database changes and how to migrate to SQL Server.

The number of tutorials – 11 in all plus a troubleshooting page – might make the deployment process seem daunting. In fact, the basic procedures for deploying a site make up a relatively small part of the tutorial set. However, in real-world situations, you often need information about some small but important extra aspect of deployment — for example, setting folder permissions on the target server. We've included many of these additional techniques in the tutorials, with the hope that the tutorials don't leave out information that might prevent you from successfully deploying a real application.

The tutorials are designed to run in sequence, and each part builds on the previous part. However, you can skip parts that aren't relevant to your situation. (Skipping parts might require you to adjust the procedures in later tutorials.)

Intended Audience

The tutorials are aimed at ASP.NET developers who work in small organizations or other environments where:

- A continuous integration process (automated builds and deployment) is not used.

- The production environment is a third-party hosting provider.
- One person typically fills multiple roles (the same person develops, tests, and deploys).

In enterprise environments, it's more typical to implement continuous integration processes, and the production environment is usually hosted by the company's own servers. Different people also typically perform different roles. For information about enterprise deployment, see Deploying Web Applications in Enterprise Scenarios.

Organizations of all sizes can also deploy web applications to Azure, and most of the procedures shown in these tutorials apply also to Azure App Services Web Apps. For an introduction to Azure, see https://azure.microsoft.com.

The Hosting Provider Shown in the Tutorials

The tutorials take you through the process of setting up an account with a hosting company and deploying the application to that hosting provider. A specific hosting company was chosen so that the tutorials could illustrate the complete experience of deploying to a live website. Each hosting company provides different features, and the experience of deploying to their servers varies somewhat; however, the process described in this tutorial is typical for the overall process.

The hosting provider used for this tutorial, Cytanium.com, is one of many that are available, and its use in this tutorial does not constitute an endorsement or recommendation.

Deploying Web Site Projects

Contoso University is a Visual Studio web application project. Most of the deployment methods and tools demonstrated in this tutorial do not apply to Web Site Projects. For information about how to deploy web site projects, see ASP.NET Deployment Content Map.

Deploying ASP.NET MVC Projects

For this tutorial you deploy an ASP.NET Web Forms project, but everything you learn how to do is applicable to ASP.NET MVC as well. A Visual Studio MVC project is just another form of web application project. The only difference is that if you're deploying to a hosting provider that does not support ASP.NET MVC or your target version of it, you must make sure that you have installed the appropriate (MVC 3 or MVC 4) NuGet package in your project.

Programming Language

The sample application uses C# but the tutorials do not require knowledge of C#, and the deployment techniques shown by the tutorials are not language-specific.

Troubleshooting During this Tutorial

When an error happens during deployment, or if the deployed site does not run correctly, the error messages don't always provide a solution. To help you with some common problem scenarios, a troubleshooting reference page is available. If you get an error message or something doesn't work as you go through the tutorials, be sure to check the troubleshooting page.

Comments Welcome

Comments on the tutorials are welcome, and when the tutorial is updated every effort will be made to take into account corrections or suggestions for improvements that are provided in tutorial comments.

Prerequisites

Before you start, make sure that you have Windows 7 or later and one of the following products installed on your computer:

- Visual Studio 2010 SP1
- Visual Web Developer Express 2010 SP1
- Visual Studio 2012 RC or Visual Studio Express 2012 RC for Web

If you have Visual Studio 2010 SP1 or Visual Web Developer Express 2010 SP1, install the following products also:

- Azure SDK for .NET (VS 2010 SP1) (includes the Web Publish Update)
- Microsoft Visual Studio 2010 SP1 Tools for SQL Server Compact 4.0

Some other software is required in order to complete the tutorial, but you don't have to have that loaded yet. The tutorial will walk you through the steps for installing it when you need it.

Downloading the Sample Application

The application you'll deploy is named Contoso University and has already been created for you. It's a simplified version of a university web site, based loosely on the Contoso University application described in the Entity Framework tutorials on the ASP.NET site.

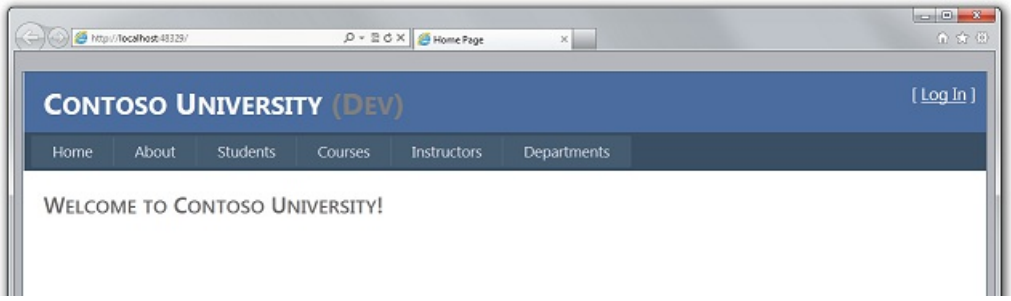
When you have the prerequisites installed, download the Contoso University web application. The *.zip* file contains multiple versions of the project and a PDF file that contains all 12 tutorials. To work through the steps of the tutorial, start with ContosoUniversity-Begin. To see what the project looks like at the end of the tutorials, open ContosoUniversity-End. To see what the project looks like before the migration to full SQL Server in tutorial 10, open ContosoUniversity-AfterTutorial09.

To prepare to work through the tutorial steps, save ContosoUniversity-Begin to whatever folder you use for working with Visual Studio projects. By default this is the following folder:

```
C:\Users\<username>\Documents\Visual Studio 2012\Projects
```

(For the screen shots in this tutorial, the project folder is located in the root directory on the `C`: drive.)

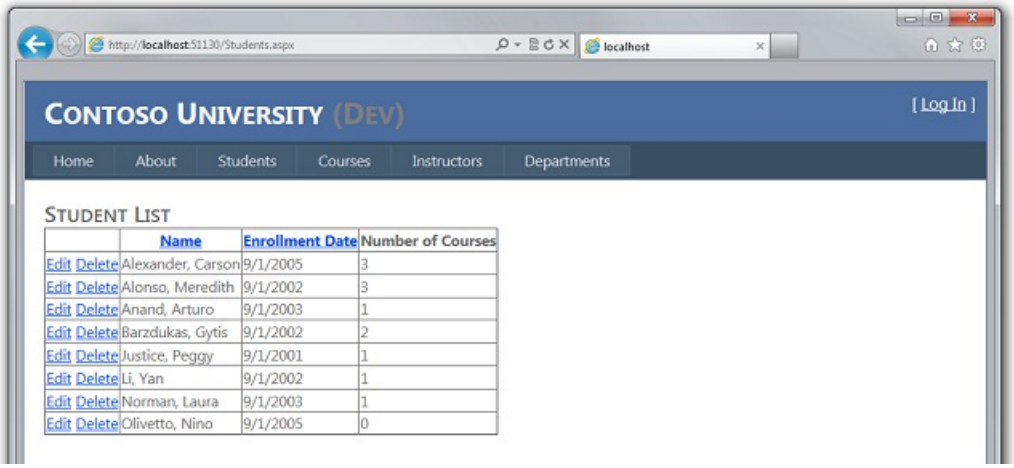
Start Visual Studio, open the project, and press CTRL-F5 to run it.

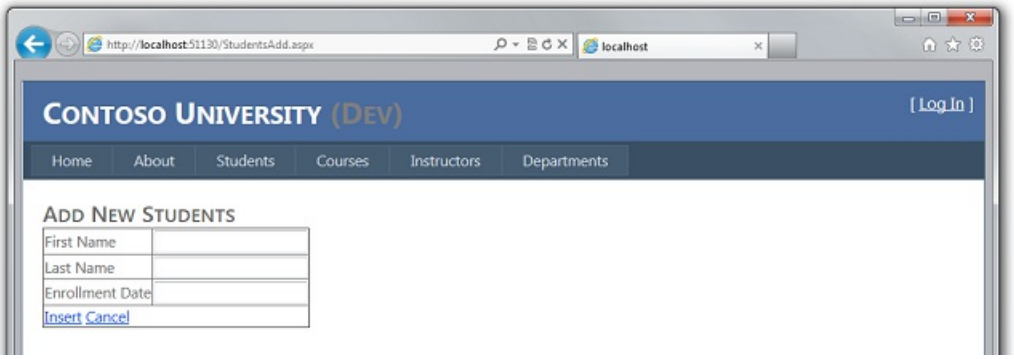


The website pages are accessible from the menu bar and let you perform the following functions:

- Display student statistics (the About page).
- Display, edit, delete, and add students.
- Display and edit courses.
- Display and edit instructors.
- Display and edit departments.

Following are screen shots of a few representative pages.





Reviewing Application Features that Affect Deployment

The following features of the application affect how you deploy it or what you have to do to deploy it. Each of these is explained in more detail in the following tutorials in the series.

- Contoso University uses a SQL Server Compact database to store application data such as student and instructor names. The database contains a mix of test data and production data, and when you deploy to production you need to exclude the test data. Later in the tutorial series you'll migrate from SQL Server Compact to SQL Server.
- The application uses the ASP.NET membership system, which stores user account information in a SQL Server Compact database. The application defines an administrator user who has access to some restricted information. You need to deploy the membership database without test accounts but with one administrator account.
- Because the application database and the membership database use SQL Server Compact as the database engine, you have to deploy the database engine to the hosting provider, as well as the databases themselves.
- The application uses ASP.NET universal membership providers so that the membership system can store its data in a SQL Server Compact database. The assembly that contains the universal membership providers must be deployed with the application.
- The application uses the Entity Framework 5.0 to access data in the application database. The assembly that contains Entity Framework 5.0 must be deployed with the application.
- The application uses a third-party error logging and reporting utility. This utility is provided in an assembly which must be deployed with the application.
- The error logging utility writes error information in XML files to a file folder. You have to make sure that the account that ASP.NET runs under in the deployed site has write permission to this folder, and you have to exclude this folder from deployment. (Otherwise, error log data from the test environment might be deployed to production and/or production error log files might be deleted.)
- The application includes some settings that must be changed in the deployed *Web.config* file depending on the destination environment (test or production), and other settings that must be changed depending on the build configuration (Debug or Release).

- The Visual Studio solution includes a class library project. Only the assembly that this project generates should be deployed, not the project itself.

In this first tutorial in the series, you have downloaded the sample Visual Studio project and reviewed site features that affect how you deploy the application. In the following tutorials, you prepare for deployment by setting up some of these things to be handled automatically. Others you take care of manually.

NEXT

Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio or Visual Web Developer: Deploying SQL Server Compact Databases - 2 of 12

2/20/2020 • 17 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This series of tutorials shows you how to deploy (publish) an ASP.NET web application project that includes a SQL Server Compact database by using Visual Studio 2012 RC or Visual Studio Express 2012 RC for Web. You can also use Visual Studio 2010 if you install the Web Publish Update. For an introduction to the series, see the first tutorial in the series.
>
> For a tutorial that shows deployment features introduced after the RC release of Visual Studio 2012, shows how to deploy SQL Server editions other than SQL Server Compact, and shows how to deploy to Azure App Service Web Apps, see ASP.NET Web Deployment using Visual Studio.

Overview

This tutorial shows how to set up two SQL Server Compact databases and the database engine for deployment.

For database access, the Contoso University application requires the following software that must be deployed with the application because it is not included in the .NET Framework:

- SQL Server Compact (the database engine).
- ASP.NET Universal Providers (which enable the ASP.NET membership system to use SQL Server Compact)
- Entity Framework 5.0(Code First with Migrations).

The database structure and some (not all) of the data in the application's two databases must also be deployed. Typically, as you develop an application, you enter test data into a database that you don't want to deploy to a live site. However, you might also enter some production data that you do want to deploy. In this tutorial you'll configure the Contoso University project so that the required software and the correct data are included when you deploy.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

SQL Server Compact versus SQL Server Express

The sample application uses SQL Server Compact 4.0. This database engine is a relatively new option for websites; earlier versions of SQL Server Compact do not work in a web hosting environment. SQL Server Compact offers a few benefits compared to the more common scenario of developing with SQL Server Express and deploying to full SQL Server. Depending on the hosting provider you choose, SQL Server Compact might be cheaper to deploy, because some providers charge extra to support a full SQL Server database. There is no extra charge for SQL Server Compact because you can deploy the database engine itself as part of your web application.

However, you should also be aware of its limitations. SQL Server Compact does not support stored procedures, triggers, views, or replication. (For a complete list of SQL Server features that are not supported by SQL Server Compact, see Differences Between SQL Server Compact and SQL Server.) Also, some of the tools that you can use to manipulate schemas and data in SQL Server Express and SQL Server databases do not work with SQL Server Compact. For example, you cannot use SQL Server Management Studio or SQL Server Data Tools in Visual Studio with SQL Server Compact databases. You do have other options for working with SQL Server Compact databases:

- You can use Server Explorer in Visual Studio, which offers limited database manipulation functionality for SQL Server Compact.
- You can use the database manipulation feature of WebMatrix, which has more features than Server Explorer.
- You can use relatively full-featured third-party or open source tools, such as the SQL Server Compact Toolbox and SQL Compact data and schema script utility.
- You can write and run your own DDL (data definition language) scripts to manipulate the database schema.

You can start with SQL Server Compact and then upgrade later as your needs evolve. Later tutorials in this series show you how to migrate from SQL Server Compact to SQL Server Express and to SQL Server. However, if you're creating a new application and expect to need SQL Server in the near future, it's probably best to start with SQL Server or SQL Server Express.

Configuring the SQL Server Compact Database Engine for Deployment

The software required for data access in the Contoso University application was added by installing the following NuGet packages:

- SqlServerCompact
- System.Web.Providers (ASP.NET universal providers)
- EntityFramework
- EntityFramework.SqlServerCompact

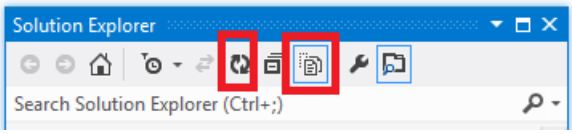
The links point to the current versions of these packages, which might be newer than what is installed in the starter project that you downloaded for this tutorial. For deployment to a hosting provider, make sure that you use Entity Framework 5.0 or later. Earlier versions of Code First Migrations require Full Trust, and at many hosting providers your application will run in Medium Trust. For more information about Medium Trust, see the Deploying to IIS as a Test Environment tutorial.

NuGet package installation generally takes care of everything that you need in order to deploy this software with the application. In some cases, this involves tasks such as changing the Web.config file and adding PowerShell scripts that run whenever you build the solution. **If you want to add support for any of these features (such as SQL Server Compact and Entity Framework) without using NuGet, make sure that you know what NuGet package installation does so that you can do the same work manually.**

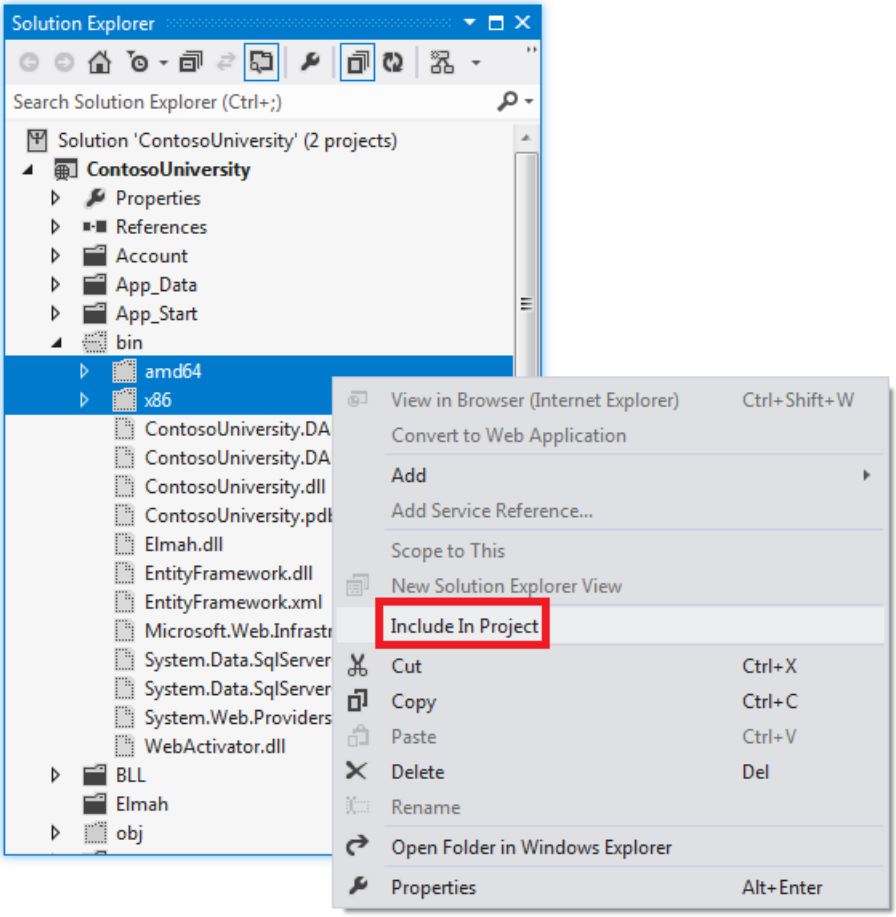
There is one exception where NuGet doesn't take care of everything you have to do in order to ensure successful deployment. The SqlServerCompact NuGet package adds a post-build script to your project that copies the native assemblies to *x86* and *amd64* subfolders under the project *bin* folder, but the script does not include those folders in the project. As a result, Web Deploy will not copy them to the destination web site unless you manually include them in the project. (This behavior results from the default deployment configuration; another option, which you won't use in these tutorials, is to change the setting that controls this behavior. The setting that you can change is **Only files needed to run the application** under **Items to deploy** on the **Package/Publish Web** tab of the **Project Properties** window. Changing this setting is not generally recommended because it can result in the deployment of many more files to the production environment than are needed there. For more information about the alternatives, see the Configuring Project Properties tutorial.)

Build the project, and then in **Solution Explorer** click **Show all Files** if you have not already done so. You might

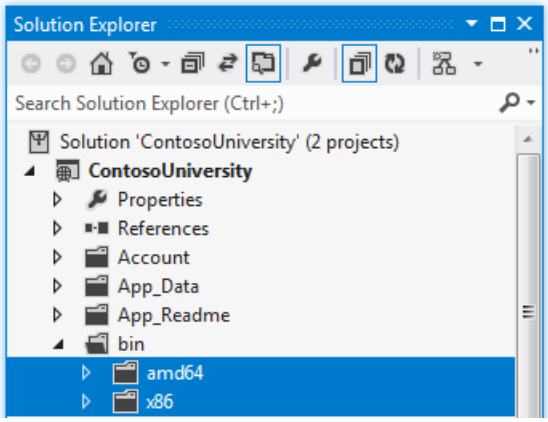
also have to click **Refresh**.



Expand the **bin** folder to see the **amd64** and **x86** folders, and then select those folders, right-click, and select **Include in Project**.



The folder icons change to show that the folder has been included in the project.



Configuring Code First Migrations for Application Database Deployment

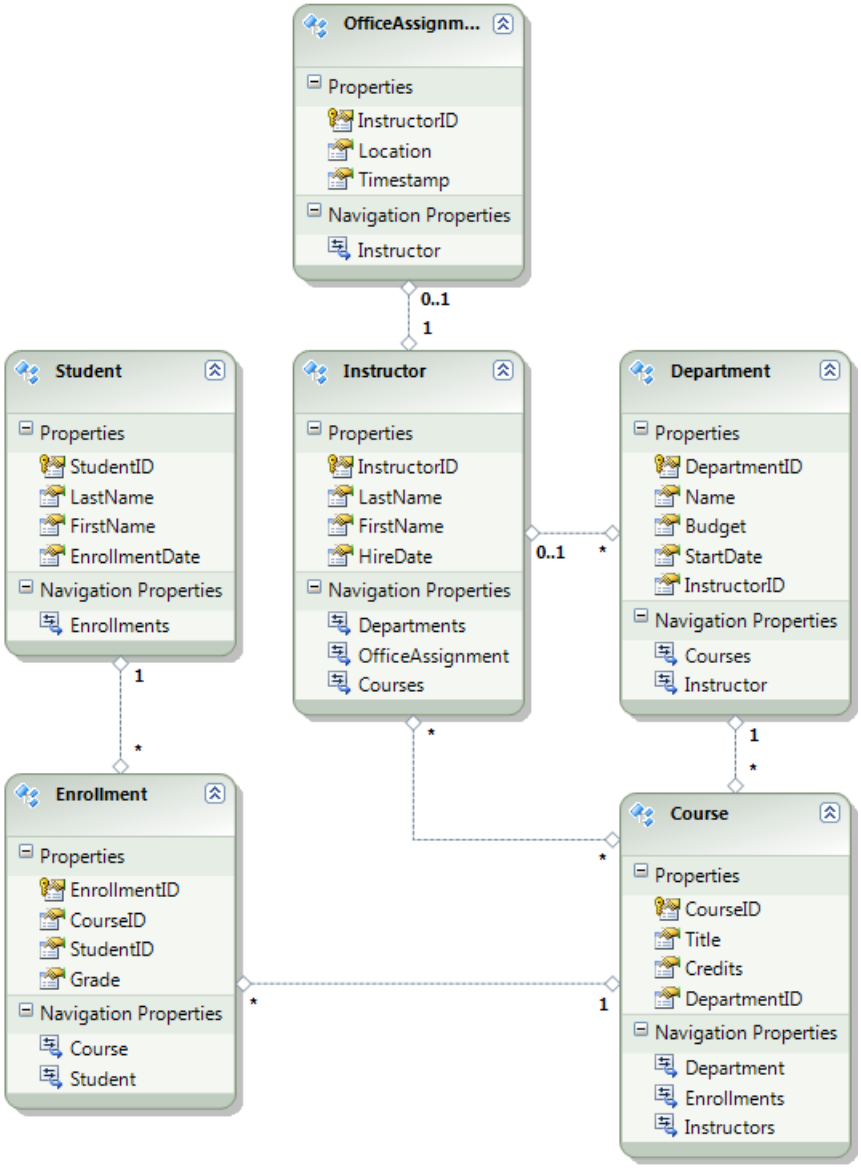
When you deploy an application database, typically you don't simply deploy your development database with all of the data in it to production, because much of the data in it is probably there only for testing purposes. For example, the student names in a test database are fictional. On the other hand, you often can't deploy just the

database structure with no data in it at all. Some of the data in your test database might be real data and must be there when users begin to use the application. For example, your database might have a table that contains valid grade values or real department names.

To simulate this common scenario, you'll configure a Code First Migrations Seed method that inserts into the database only the data that you want to be there in production. This Seed method won't insert test data because it will run in production after Code First creates the database in production.

In earlier versions of Code First before Migrations was released, it was common for Seed methods to insert test data also, because with every model change during development the database had to be completely deleted and re-created from scratch. With Code First Migrations, test data is retained after database changes, so including test data in the Seed method is not necessary. The project you downloaded uses the pre-Migrations method of including all data in the Seed method of an initializer class. In this tutorial you'll disable the initializer class and enable Migrations. Then you'll update the Seed method in the Migrations configuration class so that it inserts only data that you want to be inserted in production.

The following diagram illustrates the schema of the application database:



For these tutorials, you'll assume that the `Student` and `Enrollment` tables should be empty when the site is first deployed. The other tables contain data that has to be preloaded when the application goes live.

Since you will be using Code First Migrations, you no longer have to use the **DropCreateDatabaseIfModelChanges** Code First initializer. The code for this initializer is in the SchoolInitializer.cs file in the ContosoUniversity.DAL project. A setting in the **appSettings** element of the

Web.config file causes this initializer to run whenever the application tries to access the database for the first time:

```
<appSettings>
  <add key="Environment" value="Dev" />
  <add key="DatabaseInitializerForType ContosoUniversity.DAL.SchoolContext, ContosoUniversity.DAL"
value="ContosoUniversity.DAL.SchoolInitializer, ContosoUniversity.DAL" />
</appSettings>
```

Open the application Web.config file and remove the element that specifies the Code First initializer class from the appSettings element. The appSettings element now looks like this:

```
<appSettings>
  <add key="Environment" value="Dev" />
</appSettings>
```

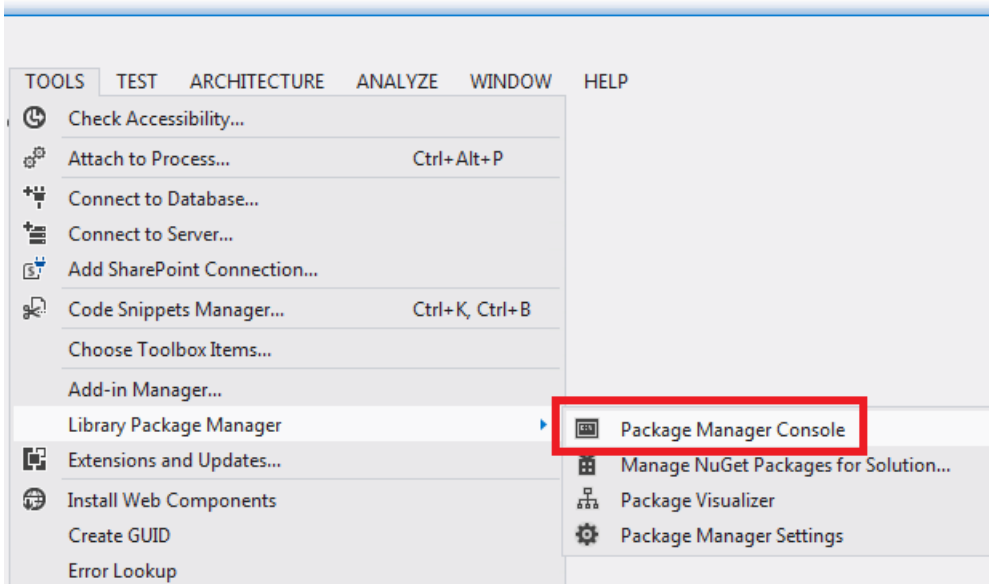
NOTE

Another way to specify an initializer class is do it by calling `Database.SetInitializer` in the `Application_Start` method in the *Global.asax* file. If you are enabling Migrations in a project that uses that method to specify the initializer, remove that line of code.

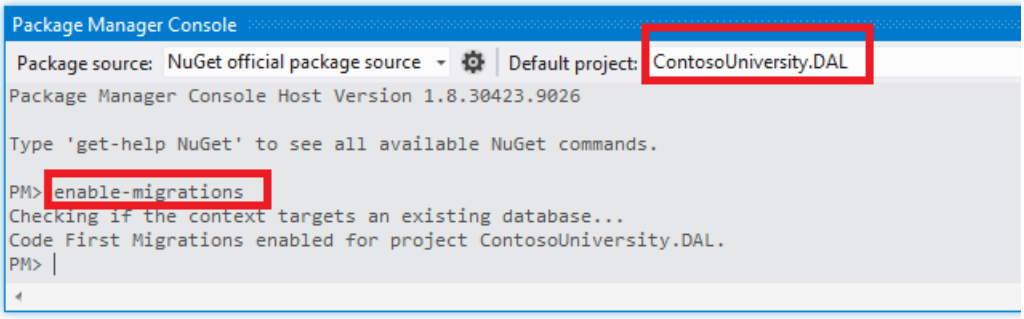
Next, enable Code First Migrations.

The first step is to make sure that the ContosoUniversity project is set as the startup project. In **Solution Explorer**, right-click the ContosoUniversity project and select **Set as Startup Project**. Code First Migrations will look in the startup project to find the database connection string.

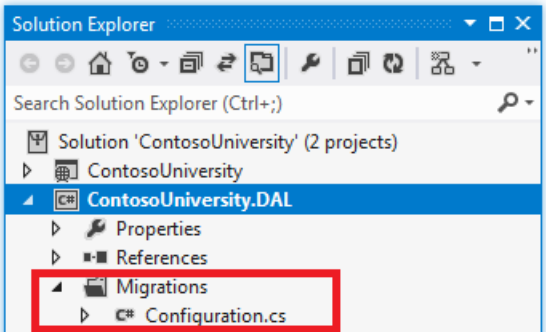
From the **Tools** menu, click **NuGet Package Manager** and then **Package Manager Console**.



At the top of the **Package Manager Console** window select ContosoUniversity.DAL as the default project and then at the `PM>` prompt enter "enable-migrations".



This command creates a *Configuration.cs* file in a new *Migrations* folder in the ContosoUniversity.DAL project.



You selected the DAL project because the "enable-migrations" command must be executed in the project that contains the Code First context class. When that class is in a class library project, Code First Migrations looks for the database connection string in the startup project for the solution. In the ContosoUniversity solution, the web project has been set as the startup project. (If you did not want to designate the project that has the connection string as the startup project in Visual Studio, you can specify the startup project in the PowerShell command. To see the command syntax for the enable-migrations command, you can enter the command "get-help enable-migrations".)

Open the Configuration.cs file and replace the comments in the `Seed` method with the following code:

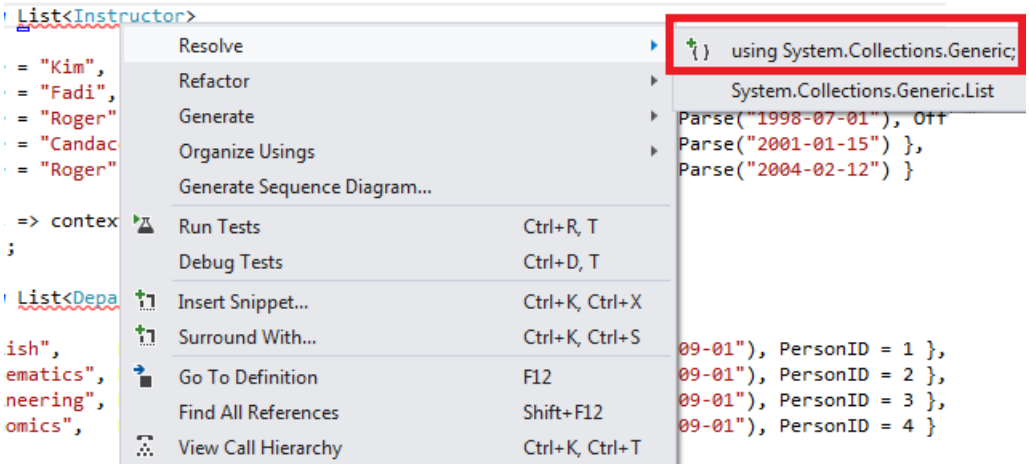
```
var instructors = new List<Instructor>
{
    new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie", HireDate = DateTime.Parse("1995-03-
11"), OfficeAssignment = new OfficeAssignment { Location = "Smith 17" } },
    new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",    HireDate = DateTime.Parse("2002-07-
06"), OfficeAssignment = new OfficeAssignment { Location = "Gowan 27" } },
    new Instructor { FirstMidName = "Roger",   LastName = "Harui",       HireDate = DateTime.Parse("1998-07-
01"), OfficeAssignment = new OfficeAssignment { Location = "Thompson 304" } },
    new Instructor { FirstMidName = "Candace", LastName = "Kapoor",      HireDate = DateTime.Parse("2001-01-
15") },
    new Instructor { FirstMidName = "Roger",   LastName = "Zheng",       HireDate = DateTime.Parse("2004-02-
12") }
};
instructors.ForEach(s => context.Instructors.AddOrUpdate(i => i.LastName, s));
context.SaveChanges();

var departments = new List<Department>
{
    new Department { Name = "English",     Budget = 350000, StartDate = DateTime.Parse("2007-09-01"),
PersonID = 1 },
    new Department { Name = "Mathematics", Budget = 100000, StartDate = DateTime.Parse("2007-09-01"),
PersonID = 2 },
    new Department { Name = "Engineering", Budget = 350000, StartDate = DateTime.Parse("2007-09-01"),
PersonID = 3 },
    new Department { Name = "Economics",   Budget = 100000, StartDate = DateTime.Parse("2007-09-01"),
PersonID = 4 }
};
departments.ForEach(s => context.Departments.AddOrUpdate(d => d.Name, s));
context.SaveChanges();

var courses = new List<Course>
{
    new Course { CourseID = 1050, Title = "Chemistry",      Credits = 3, DepartmentID = 3, Instructors = new
List<Instructor>() },
    new Course { CourseID = 4022, Title = "Microeconomics", Credits = 3, DepartmentID = 4, Instructors = new
List<Instructor>() },
    new Course { CourseID = 4041, Title = "Macroeconomics", Credits = 3, DepartmentID = 4, Instructors = new
List<Instructor>() },
    new Course { CourseID = 1045, Title = "Calculus",       Credits = 4, DepartmentID = 2, Instructors = new
List<Instructor>() },
    new Course { CourseID = 3141, Title = "Trigonometry",   Credits = 4, DepartmentID = 2, Instructors = new
List<Instructor>() },
    new Course { CourseID = 2021, Title = "Composition",    Credits = 3, DepartmentID = 1, Instructors = new
List<Instructor>() },
    new Course { CourseID = 2042, Title = "Literature",     Credits = 4, DepartmentID = 1, Instructors = new
List<Instructor>() }
};
courses.ForEach(s => context.Courses.AddOrUpdate(s));
context.SaveChanges();

courses[0].Instructors.Add(instructors[0]);
courses[0].Instructors.Add(instructors[1]);
courses[1].Instructors.Add(instructors[2]);
courses[2].Instructors.Add(instructors[2]);
courses[3].Instructors.Add(instructors[3]);
courses[4].Instructors.Add(instructors[3]);
courses[5].Instructors.Add(instructors[3]);
courses[6].Instructors.Add(instructors[3]);
context.SaveChanges();
```

The references to `List` have red squiggly lines under them because you don't have a `using` statement for its namespace yet. Right-click one of the instances of `List` and click **Resolve**, and then click **using System.Collections.Generic**.



This menu selection adds the following code to the `using` statements near the top of the file.

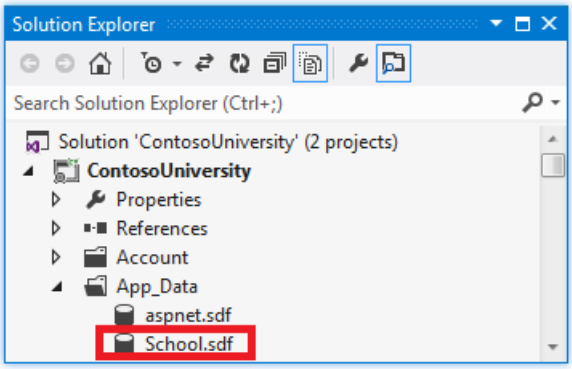
```
using System.Collections.Generic;
```

> **NOTE**
>
> Adding code to the `Seed` method is one of many ways that you can insert fixed data into the database. An alternative is to add code to the `Up` and `Down` methods of each migration class. The `Up` and `Down` methods contain code that implements database changes. You'll see examples of them in the Deploying a Database Update tutorial.
>
> You can also write code that executes SQL statements by using the `Sql` method. For example, if you were adding a Budget column to the Department table and wanted to initialize all department budgets to $1,000.00 as part of a migration, you could add the following line of code to the `Up` method for that migration:
>
> `Sql("UPDATE Department SET Budget = 1000");`
>
> This example shown for this tutorial uses the `AddOrUpdate` method in the `Seed` method of the Code First Migrations `Configuration` class. Code First Migrations calls the `Seed` method after every migration, and this method updates rows that have already been inserted, or inserts them if they don't exist yet. The `AddOrUpdate` method might not be the best choice for your scenario. For more information, see Take care with EF 4.3 AddOrUpdate Method on Julie Lerman's blog.

Press CTRL-SHIFT-B to build the project.

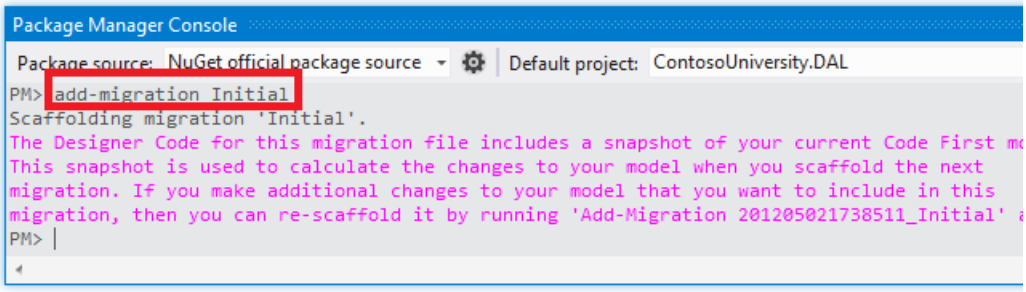
The next step is to create a `DbMigration` class for the initial migration. You want this migration to create a new database, so you have to delete the database that already exists. SQL Server Compact databases are contained in *.sdf* files in the *App_Data* folder. In **Solution Explorer**, expand *App_Data* in the ContosoUniversity project to see the two SQL Server Compact databases, which are represented by *.sdf* files.

Right-click the *School.sdf* file and click **Delete**.



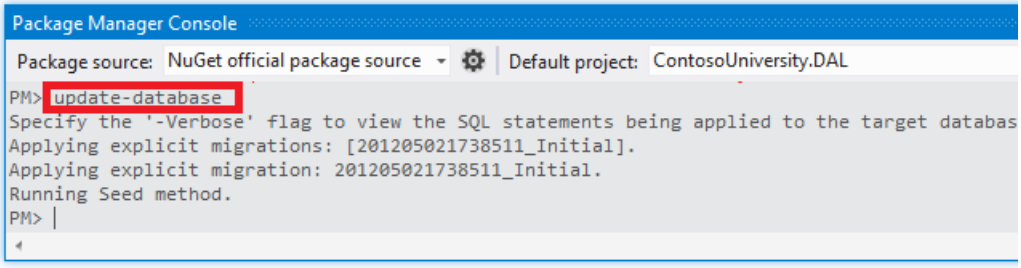
In the **Package Manager Console** window, enter the command "add-migration Initial" to create the initial

migration and name it "Initial".



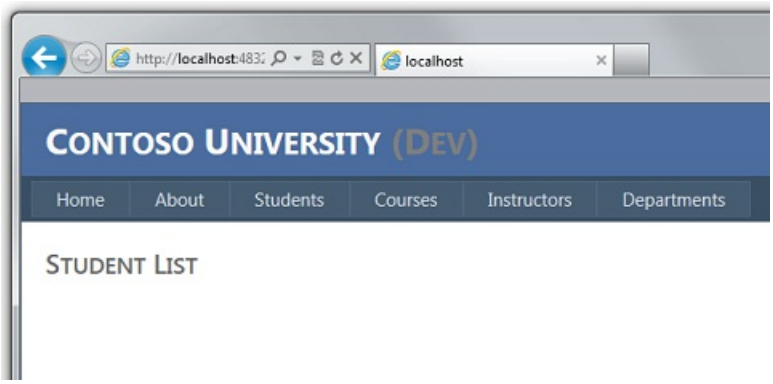
Code First Migrations creates another class file in the *Migrations* folder, and this class contains code that creates the database schema.

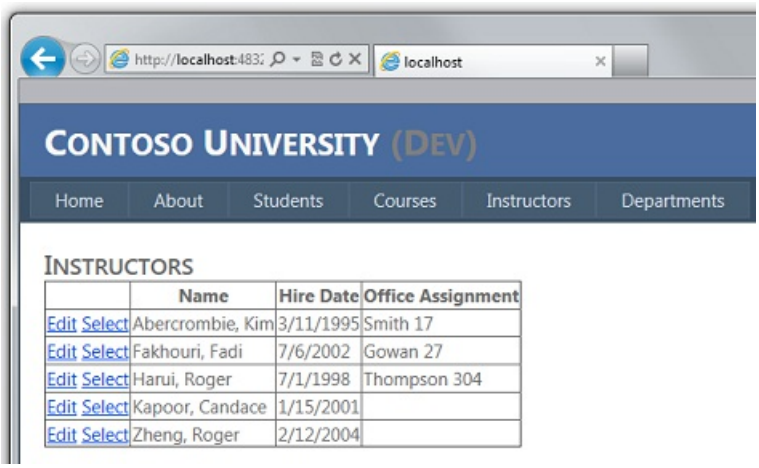
In the **Package Manager Console**, enter the command "update-database" to create the database and run the **Seed** method.



(If you get an error that indicates a table already exists and can't be created, it is probably because you ran the application after you deleted the database and before you executed `update-database`. In that case, delete the *School.sdf* file again and retry the `update-database` command.)

Run the application. Now the Students page is empty but the Instructors page contains instructors. This is what you will get in production after you deploy the application.

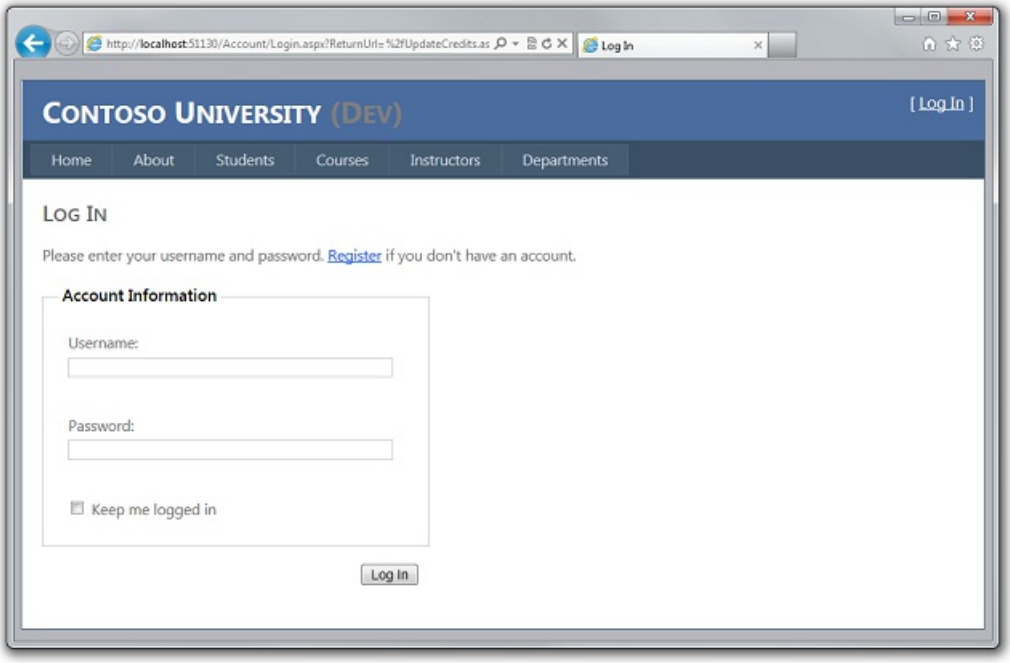




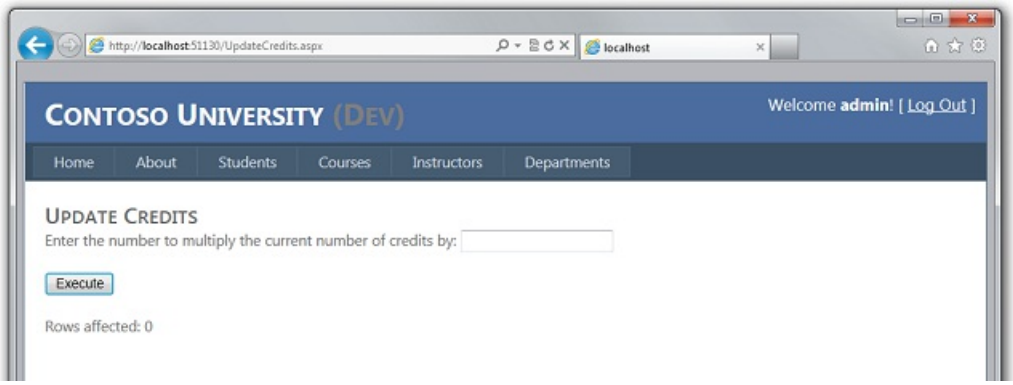
The project is now ready to deploy the *School* database.

Creating a Membership Database for Deployment

The Contoso University application uses the ASP.NET membership system and forms authentication to authenticate and authorize users. One of its pages is accessible only to administrators. To see this page, run the application and select **Update Credits** from the flyout menu under **Courses**. The application displays the **Log In** page, because only administrators are authorized to use the **Update Credits** page.



Log in as "admin" using the password "Pas$w0rd" (notice the number zero in place of the letter "o" in "w0rd"). After you log in, the **Update Credits** page is displayed.



When you deploy a site for the first time, it is common to exclude most or all of the user accounts you create for testing. In this case, you'll deploy an administrator account and no user accounts. Rather than manually deleting test accounts, you'll create a new membership database that has only the one administrator user account that you need in production.

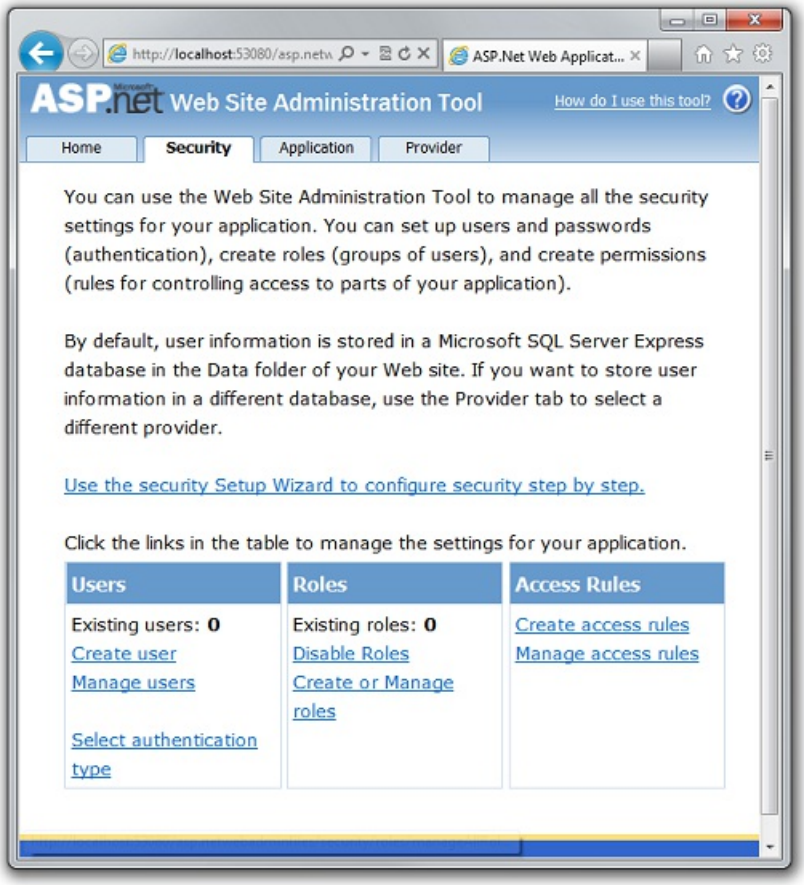
> **NOTE**
>
> The membership database stores a hash of account passwords. In order to deploy accounts from one machine to another, you must make sure that hashing routines don't generate different hashes on the destination server than they do on the source computer. They will generate the same hashes when you use the ASP.NET Universal Providers, as long as you don't change the default algorithm. The default algorithm is HMACSHA256 and is specified in the **validation** attribute of the **machineKey** element in the Web.config file.

The membership database is not maintained by Code First Migrations, and there is no automatic initializer that seeds the database with test accounts (as there is for the School database). Therefore, to keep test data available you'll make a copy of the test database before you create a new one.

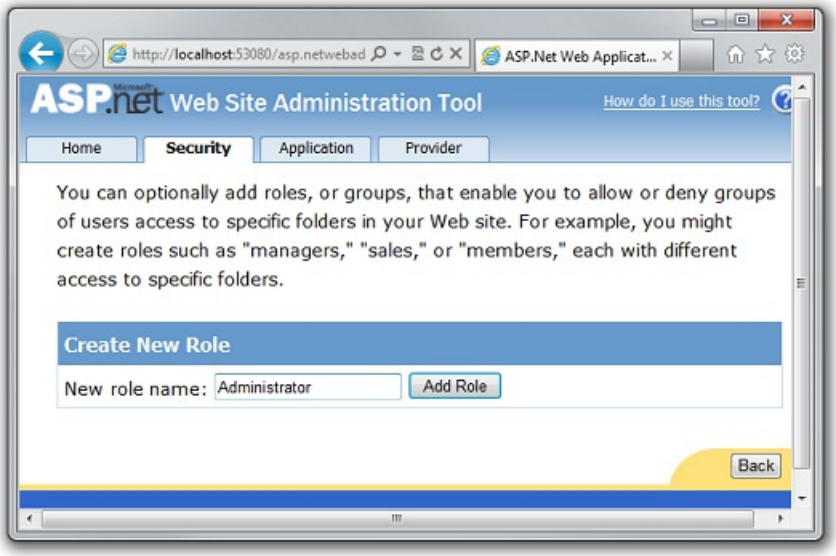
In **Solution Explorer**, rename the *aspnet.sdf* file in the *App_Data* folder to *aspnet-Dev.sdf*. (Don't make a copy, just rename it — you'll create a new database in a moment.)

In **Solution Explorer**, make sure that the web project (ContosoUniversity, not ContosoUniversity.DAL) is selected. Then in the **Project** menu, select **ASP.NET Configuration** to run the **Web Site Administration Tool** (WAT).

Select the **Security** tab.

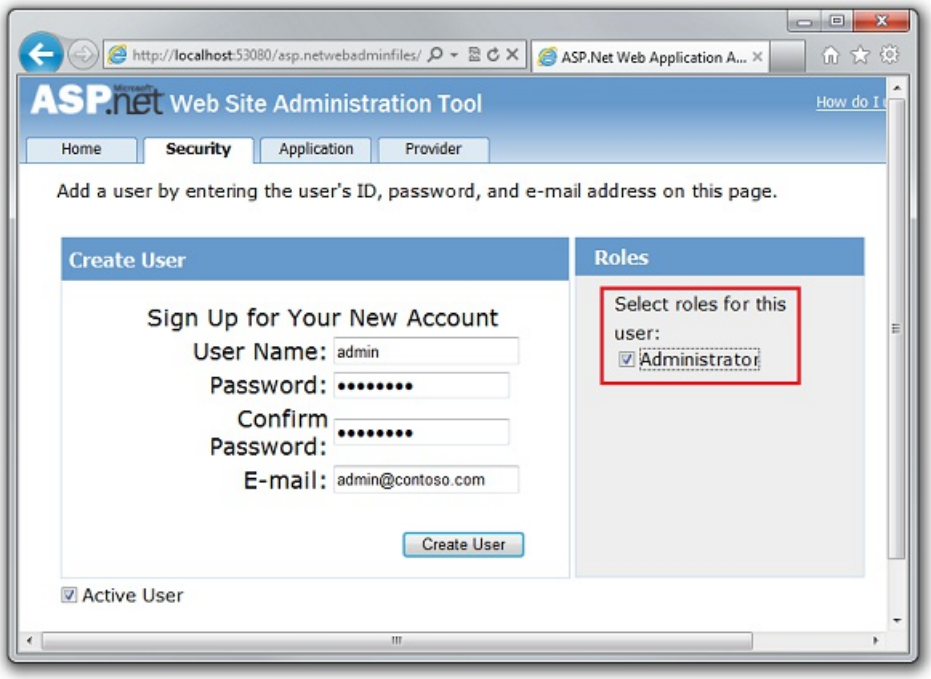


Click **Create or Manage Roles** and add an **Administrator** role.

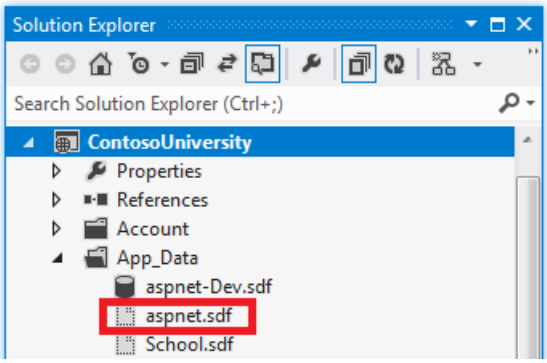


Navigate back to the **Security** tab, click **Create User**, and add user "admin" as an administrator. Before you click

the **Create User** button on the **Create User** page, make sure that you select the **Administrator** check box. The password used in this tutorial is "Pas$w0rd", and you can enter any email address.



Close the browser. In **Solution Explorer**, click the refresh button to see the new *aspnet.sdf* file.

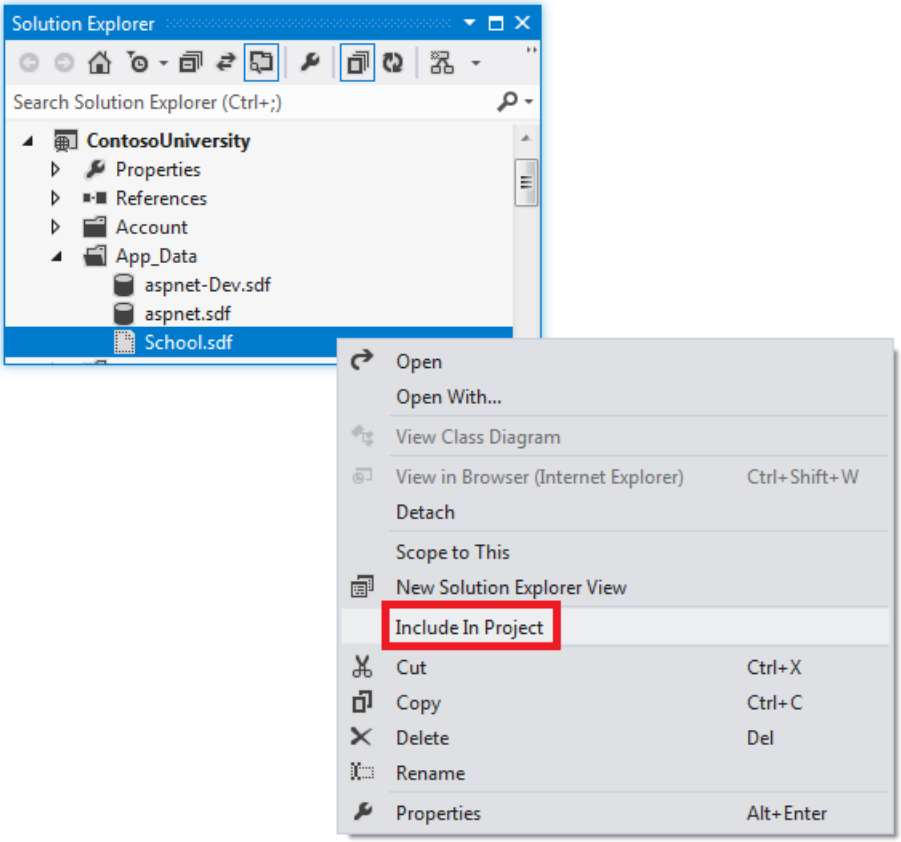


Right-click **aspnet.sdf** and select **Include in Project**.

Distinguishing Development from Production Databases

In this section, you'll rename the databases so that the development versions are School-Dev.sdf and aspnet-Dev.sdf and the production versions are School-Prod.sdf and aspnet-Prod.sdf. This isn't necessary, but doing so will help keep you from getting test and production versions of the databases confused.

In **Solution Explorer**, click **Refresh** and expand the App_Data folder to see the School database that you created earlier; right-click it and select **Include in project**.



Rename *aspnet.sdf* to *aspnet-Prod.sdf*.

Rename *School.sdf* to *School-Dev.sdf*.

When you run the application in Visual Studio you don't want to use the *-Prod* versions of the database files, you want to use *-Dev* versions. Therefore you have to change the connection strings in the Web.config file so that they point to the *-Dev* versions of the databases. (You haven't created a School-Prod.sdf file, but that's OK because Code First will create that database in production the first time you run your app there.)

Open the application Web.config file, and locate the connection strings:

```
<configuration>
  <!-- Settings -->
  <connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=|DataDirectory|aspnet.sdf"
providerName="System.Data.SqlServerCe.4.0" />
    <add name="SchoolContext" connectionString="Data Source=|DataDirectory|School.sdf"
providerName="System.Data.SqlServerCe.4.0" />
  </connectionStrings>
  <!-- Settings -->
</configuration>
```

Change "aspnet.sdf" to "aspnet-Dev.sdf", and change "School.sdf" to "School-Dev.sdf":

```
<configuration>
  <!-- Settings -->
  <connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=|DataDirectory|aspnet-Dev.sdf"
providerName="System.Data.SqlServerCe.4.0" />
    <add name="SchoolContext" connectionString="Data Source=|DataDirectory|School-Dev.sdf"
providerName="System.Data.SqlServerCe.4.0" />
  </connectionStrings>
  <!-- Settings -->
</configuration>
```

The SQL Server Compact database engine and both databases are now ready to be deployed. In the following tutorial you set up automatic *Web.config* file transformations for settings that must be different in the development, test, and production environments. (Among the settings that must be changed are the connection strings, but you'll set up those changes later when you create a publish profile.)

More Information

For more information on NuGet, see Manage Project Libraries with NuGet and NuGet Documentation. If you don't want to use NuGet, you'll need to learn how to analyze a NuGet package to determine what it does when it is installed. (For example, it might configure *Web.config* transformations, configure PowerShell scripts to run at build time, etc.) To learn more about how NuGet works, see especially Creating and Publishing a Package and Configuration File and Source Code Transformations.

PREVIOUS NEXT

Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio or Visual Web Developer: Web.Config File Transformations - 3 of 12

2/20/2020 • 9 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This series of tutorials shows you how to deploy (publish) an ASP.NET web application project that includes a SQL Server Compact database by using Visual Studio 2012 RC or Visual Studio Express 2012 RC for Web. You can also use Visual Studio 2010 if you install the Web Publish Update. For an introduction to the series, see the first tutorial in the series.
>
> For a tutorial that shows deployment features introduced after the RC release of Visual Studio 2012, shows how to deploy SQL Server editions other than SQL Server Compact, and shows how to deploy to Azure App Service Web Apps, see ASP.NET Web Deployment using Visual Studio.

Overview

This tutorial shows you how to automate the process of changing the *Web.config* file when you deploy it to different destination environments. Most applications have settings in the *Web.config* file that must be different when the application is deployed. Automating the process of making these changes keeps you from having to do them manually every time you deploy, which would be tedious and error prone.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

Web.config Transformations versus Web Deploy Parameters

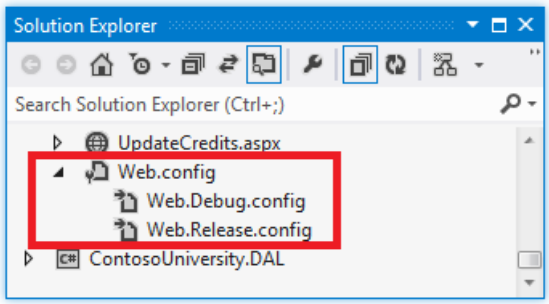
There are two ways to automate the process of changing *Web.config* file settings: Web.config transformations and Web Deploy parameters. A *Web.config* transformation file contains XML markup that specifies how to change the *Web.config* file when it is deployed. You can specify different changes for specific build configurations and for specific publish profiles. The default build configurations are Debug and Release, and you can create custom build configurations. A publish profile typically corresponds to a destination environment. (You'll learn more about publish profiles in the Deploying to IIS as a Test Environment tutorial.)

Web Deploy parameters can be used to specify many different kinds of settings that must be configured during deployment, including settings that are found in *Web.config* files. When used to specify *Web.config* file changes, Web Deploy parameters are more complex to set up, but they are useful when you do not know the value to be set until you deploy. For example, in an enterprise environment, you might create a *deployment package* and give it to a person in the IT department to install in production, and that person has to be able to enter connection strings or passwords that you do not know.

For the scenario that this tutorial covers, you know everything that has to be done to the *Web.config* file, so you do not need to use Web Deploy parameters. You'll configure some transformations that differ depending on the build configuration used, and some that differ depending on the publish profile used.

Creating Transformation Files for Publish Profiles

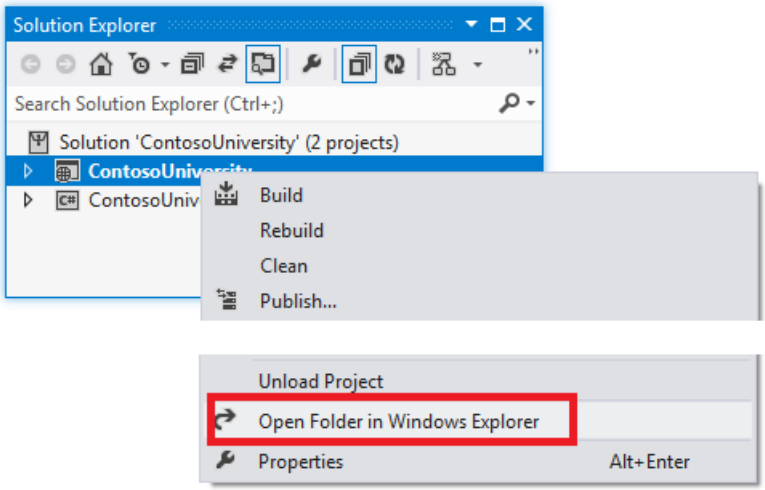
In **Solution Explorer**, expand *Web.config* to see the *Web.Debug.config* and *Web.Release.config* transformation files that are created by default for the two default build configurations.



You can create transformation files for custom build configurations by right-clicking the Web.config file and choosing **Add Config Transforms** from the context menu, but for this tutorial you don't need to do that.

You do need two more transformation files, for configuring changes that are related to the deployment destination rather than to the build configuration. A typical example of this kind of setting is a WCF endpoint that is different for test versus production. In later tutorials you'll create publish profiles named Test and Production, so you need a *Web.Test.config* file and a *Web.Production.config* file.

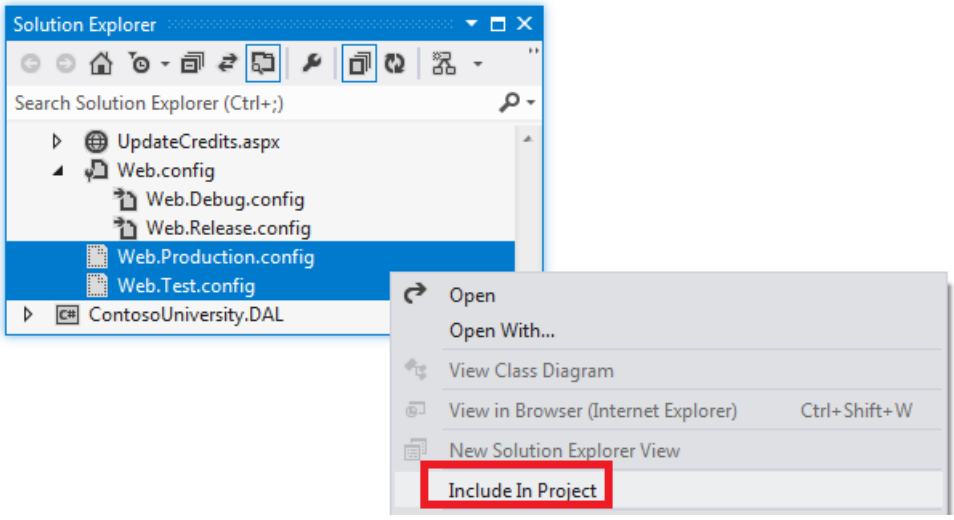
Transformation files that are tied to publish profiles must be created manually. In **Solution Explorer**, right-click the ContosoUniversity project and select **Open Folder in Windows Explorer**.



In **Windows Explorer**, select the *Web.Release.config* file, copy the file, and then paste two copies of it. Rename these copies *Web.Production.config* and *Web.Test.config*, then close **Windows Explorer**.

In **Solution Explorer**, click **Refresh** to see the new files.

Select the new files, right-click, and then click **Include in Project** in the context menu.



To prevent these files from being deployed, select them in **Solution Explorer**, and then in the **Properties** window change the **Build Action** property from **Content** to **None**. (The transformation files that are based on build configurations are automatically prevented from deploying.)

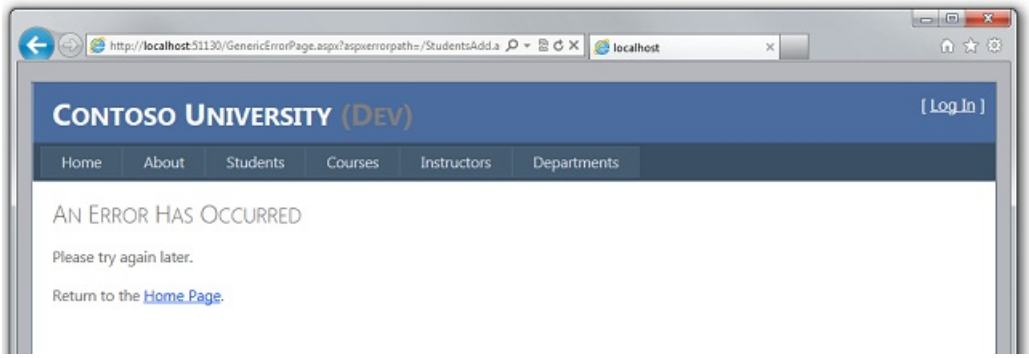
You are now ready to enter *Web.config* transformations into the *Web.config* transformation files.

Limiting Error Log Access to Administrators

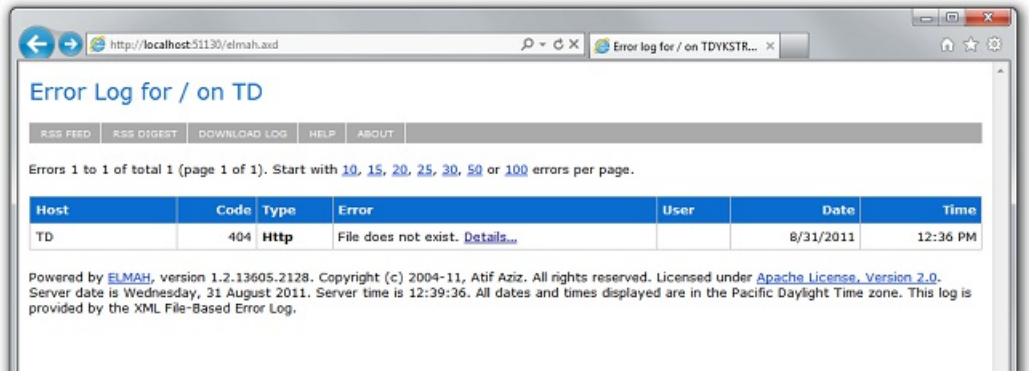
If there's an error while the application runs, the application displays a generic error page in place of the system-generated error page, and it uses Elmah NuGet package for error logging and reporting. The `customErrors` element in the *Web.config* file specifies the error page:

```
<customErrors mode="RemoteOnly" defaultRedirect="~/GenericErrorPage.aspx">
  <error statusCode="404" redirect="~/GenericErrorPage.aspx" />
</customErrors>
```

To see the error page, temporarily change the `mode` attribute of the `customErrors` element from "RemoteOnly" to "On" and run the application from Visual Studio. Cause an error by requesting an invalid URL, such as *Studentsxxx.aspx*. Instead of an IIS-generated "page not found" error page, you see the *GenericErrorPage.aspx* page.



To see the error log, replace everything in the URL after the port number with *elmah.axd* (for the example in the screen shot, `http://localhost:51130/elmah.axd`) and press Enter:



Don't forget to set the `customErrors` element back to "RemoteOnly" mode when you're done.

On your development computer it's convenient to allow free access to the error log page, but in production that would be a security risk. For the production site, you can add an authorization rule that restricts error log access just to administrators by configuring a transform in the *Web.Production.config* file.

Open *Web.Production.config* and add a new `location` element immediately after the opening `configuration` tag, as shown here. (Make sure that you add only the `location` element and not the surrounding markup which is shown here only to provide some context.)

```
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
    <location path="elmah.axd" xdt:Transform="Insert">
      <system.web>
        <authorization>
          <allow roles="Administrator" />
          <deny users="*" />
        </authorization>
      </system.web>
    </location>
</configuration>
```

The `Transform` attribute value of "Insert" causes this `location` element to be added as a sibling to any existing `location` elements in the *Web.config* file. (There is already one `location` element that specifies authorization rules for the **Update Credits** page.) When you test the production site after deployment, you'll test to verify that this authorization rule is effective.

You don't have to restrict error log access in the test environment, so you don't have to add this code to the *Web.Test.config* file.

> **NOTE**
>
> **Security Note** Never display error details to the public in a production application, or store that information in a public location. Attackers can use error information to discover vulnerabilities in a site. If you use ELMAH in your own application, be sure to investigate ways in which ELMAH can be configured to minimize security risks. The ELMAH example in this tutorial should not be considered a recommended configuration. It is an example that was chosen in order to illustrate how to handle a folder that the application must be able to create files in.

Setting an Environment Indicator

A common scenario is to have *Web.config* file settings that must be different in each environment that you deploy to. For example, an application that calls a WCF service might need a different endpoint in test and production environments. The Contoso University application includes a setting of this kind also. This setting controls a visible indicator on a site's pages that tells you which environment you are in, such as development, test, or production. The setting value determines whether the application will append "(Dev)" or "(Test)" to the main heading in the *Site.Master* master page:

CONTOSO UNIVERSITY (DEV)

The environment indicator is omitted when the application is running in production.

The Contoso University web pages read a value that is set in `appSettings` in the *Web.config* file in order to determine what environment the application is running in:

```
<appSettings>
    <add key="Environment" value="Dev" />
</appSettings>
```

The value should be "Test" in the test environment, and "Prod" in the production environment.

Open *Web.Production.config* and add an `appSettings` element immediately before the opening tag of the `location` element you added earlier:

```
<appSettings>
    <add key="Environment" value="Prod" xdt:Transform="SetAttributes" xdt:Locator="Match(key)"/>
</appSettings>
```

The `xdt:Transform` attribute value "SetAttributes" indicates that the purpose of this transform is to change attribute values of an existing element in the *Web.config* file. The `xdt:Locator` attribute value "Match(key)" indicates that the element to be modified is the one whose `key` attribute matches the `key` attribute specified here. The only other attribute of the `add` element is `value`, and that is what will be changed in the deployed *Web.config* file. This code causes the `value` attribute of the `Environment` `appSettings` element to be set to "Prod" in the *Web.config* file that is deployed to production.

Next, apply the same change to *Web.Test.config* file, except set the `value` to "Test" instead of "Prod". When you are done, the `appSettings` section in *Web.Test.config* will look like the following example:

```
<appSettings>
    <add key="Environment" value="Test" xdt:Transform="SetAttributes" xdt:Locator="Match(key)"/>
</appSettings>
```

Disabling Debug Mode

For a Release build, you do not want debugging enabled regardless of which environment you are deploying to. By default the *Web.Release.config* transform file is automatically created with code that removes the `debug` attribute from the `compilation` element:

```
<system.web>
  <compilation xdt:Transform="RemoveAttributes(debug)" />
</system.web>
```

The `Transform` attribute causes the `debug` attribute to be omitted from the deployed *Web.config* file whenever you deploy a Release build.

This same transformation is in Test and Production transform files because you created them by copying the Release transform file. You don't need it duplicated there, so open each of those files, remove the **compilation** element, and save and close each file.

Setting Connection Strings

In most cases you do not need to set up connection string transformations, because you can specify connection strings in the publish profile. But there is an exception when you're deploying a SQL Server Compact database and you're using Entity Framework Code First Migrations to update the database on the destination server. For this scenario, you have to specify an additional connection string that will be used on the server for updating the database schema. To set up this transformation, add a **<connectionStrings>** element immediately after the opening **<configuration>** tag in both the *Web.Test.config* and the *Web.Production.config* transform files:

```
<connectionStrings>
  <add name="SchoolContext_DatabasePublish" connectionString="Data Source=|DataDirectory|School-Prod.sdf"
providerName="System.Data.SqlServerCe.4.0" xdt:Transform="Insert"/>
</connectionStrings>
```

The `Transform` attribute specifies that this connection string will be added to the *connectionStrings* element in the deployed *Web.config* file. (The publish process creates this additional connection string automatically for you if it doesn't exist, but by default the **providerName** attribute gets set to `System.Data.SqlClient`, which does not work for SQL Server Compact. By adding the connection string manually, you keep the deployment process from creating a connection string element with the wrong provider name.)

You have now specified all of the *Web.config* transformations that you need for deploying the Contoso University application to test and production. In the following tutorial, you'll take care of deployment set-up tasks that require setting project properties.

More Information

For more information about topics covered by this tutorial, see the Web.config transformation scenario in ASP.NET Deployment Content Map.

PREVIOUS NEXT

Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio or Visual Web Developer: Configuring Project Properties - 4 of 12

2/20/2020 • 5 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

This series of tutorials shows you how to deploy (publish) an ASP.NET web application project that includes a SQL Server Compact database by using Visual Studio 2012 RC or Visual Studio Express 2012 RC for Web. You can also use Visual Studio 2010 if you install the Web Publish Update. For an introduction to the series, see the first tutorial in the series.

For a tutorial that shows deployment features introduced after the RC release of Visual Studio 2012, shows how to deploy SQL Server editions other than SQL Server Compact, and shows how to deploy to Azure App Service Web Apps, see ASP.NET Web Deployment using Visual Studio.

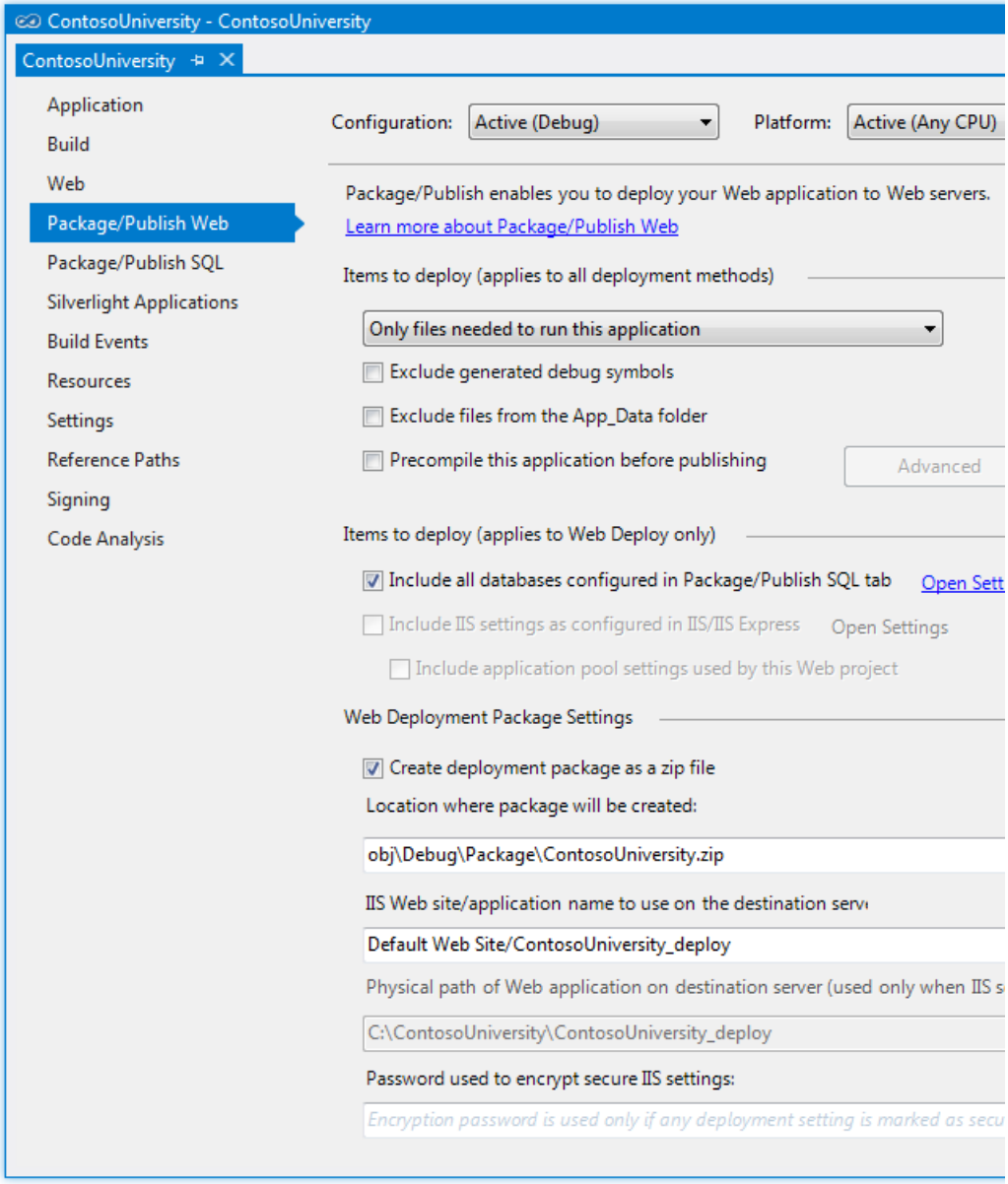
Overview

Some deployment options are configured in project properties that are stored in the project file (the *.csproj* or *.vbproj* file). In most cases, the default values of these settings are what you want, but you can use the **Project Properties** UI built into Visual Studio to work with these settings if you have to change them. In this tutorial you review the deployment settings in **Project Properties**. You also create a placeholder file that causes an empty folder to be deployed.

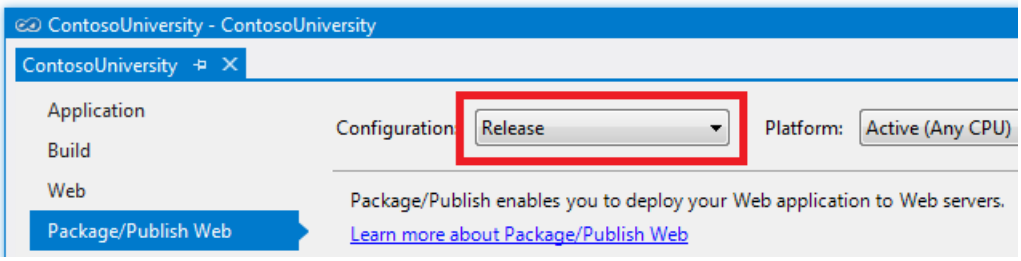
Configuring Deployment Settings in the Project Properties Window

Most settings that affect what happens during deployment are included in the publish profile, as you'll see in the following tutorials. A few settings that you should be aware of are located in the **Package/Publish** tabs of the **Project Properties** window. These settings are specified for each build configuration — that is, you can have different settings for a Release build than you have for a Debug build.

In **Solution Explorer**, right-click the **ContosoUniversity** project, select **Properties**, and then select the **Package/Publish Web** tab.



When the window is displayed, it defaults to showing settings for whichever build configuration is currently active for the solution. If the **Configuration** box does not indicate **Active (Release)**, select **Release** in order to display settings for the Release build configuration. You'll deploy Release builds to both your test and production environments.



With **Active (Release)** or **Release** selected, you see the values that are effective when you deploy using the Release build configuration:

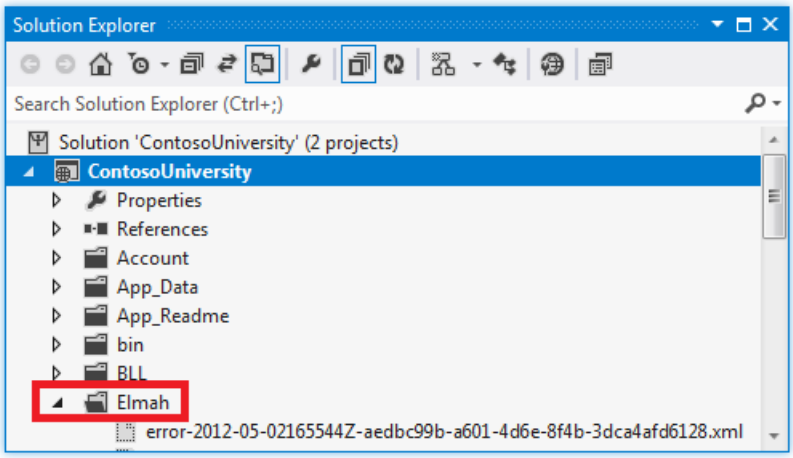
- In the **Items to deploy** box, **Only files needed to run the application** is selected. Other options are **All files in this project** or **All files in this project folder**. By leaving the default selection unchanged you avoid deploying source code files, for example. This setting is the reason why the folders that contain the SQL Server Compact binary files had to be included in the project. For more information about this setting, see **Why don't all of the files in my project folder get deployed?** in ASP.NET Web Application Project Deployment FAQ.

- **Exclude generated debug symbols** is selected. You won't be debugging when you use this build configuration.
- **Exclude files from the App_Data folder** is not selected. Your SQL Server Compact file for the membership database is in that folder and you have to deploy it. When you deploy updates that do not include database changes, you'll select this checkbox.
- **Precompile this application before publishing** is not selected. In most scenarios, there's no need to precompile web application projects. For more information about this option, see Package/Publish Web Tab, Project Properties and Advanced Precompile Settings Dialog.
- **Include all databases configured in Package/Publish SQL tab** is selected, but this option has no effect now because you aren't configuring the **Package/Publish SQL** tab. That tab is for a legacy database deployment method that used to be the only option for deploying SQL Server databases. You'll use the **Package/Publish SQL** tab in the Migrating to SQL Server tutorial.
- The **Web Deployment Package Settings** section does not apply because you're using one-click publish in these tutorials.

Change the **Configuration** drop-down box to Debug to see the default settings for Debug builds. The values are the same, except **Exclude generated debug symbols** is cleared so that you can debug when you deploy a Debug build.

Making Sure that the Elmah Folder gets Deployed

As you saw in the previous tutorial, the Elmah NuGet package provides functionality for error logging and reporting. In the Contoso University application Elmah has been configured to store error details in a folder named *Elmah*:



Excluding specific files or folders from deployment is a common requirement; another example would be a folder that users can upload files to. You don't want log files or uploaded files that were created in your development environment to be deployed to production. And if you are deploying an update to production you don't want the deployment process to delete files that exist in production. (Depending on how you set a deployment option, if a file exists in the destination site but not the source site when you deploy, Web Deploy deletes it from the destination.)

As you saw earlier in this tutorial, the **Items to deploy** option in the **Package/Publish Web** tab is set to **Only Files Needed to run this application**. As a result, log files that are created by Elmah in development will not be deployed, which is what you want to happen. (To be deployed, they would have to be included in the project and their **Build Action** property would have to be set to **Content**. For more information, see **Why don't all of the files in my project folder get deployed?** in ASP.NET Web Application Project Deployment FAQ). However, Web Deploy will not create a folder in the destination site unless there's at least one file to copy to it. Therefore, you'll add a *.txt* file to the folder to act as a placeholder so that the folder will be copied.

In **Solution Explorer**, right-click the *Elmah* folder, select **Add New Item**, and create a file named *Placeholder.txt*.

Put the following text in it: "This is a placeholder file to ensure that the folder gets deployed." and save the file. That's all you have to do in order to make sure that Visual Studio deploys this file and the folder it's in, because the **Build Action** property of *.txt* files is set to **Content** by default.

You have now completed all of the deployment set-up tasks. In the next tutorial, you'll deploy the Contoso University site to the test environment and test it there.

PREVIOUS NEXT

Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio or Visual Web Developer: Deploying to IIS as a Test Environment - 5 of 12

2/20/2020 • 12 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This series of tutorials shows you how to deploy (publish) an ASP.NET web application project that includes a SQL Server Compact database by using Visual Studio 2012 RC or Visual Studio Express 2012 RC for Web. You can also use Visual Studio 2010 if you install the Web Publish Update. For an introduction to the series, see the first tutorial in the series.
>
> For a tutorial that shows deployment features introduced after the RC release of Visual Studio 2012, shows how to deploy SQL Server editions other than SQL Server Compact, and shows how to deploy to Azure App Service Web Apps, see ASP.NET Web Deployment using Visual Studio.

Overview

This tutorial shows how to deploy an ASP.NET web application to IIS on the local computer.

When you develop an application, you generally test by running it in Visual Studio. By default, this means you're using the Visual Studio Development Server (also known as Cassini). The Visual Studio Development Server makes it easy to test during development in Visual Studio, but it doesn't work exactly like IIS. As a result, it's possible that an application will run correctly when you test it in Visual Studio, but fail when it's deployed to IIS in a hosting environment.

You can test your application more reliably in these ways:

1. Use IIS Express or full IIS instead of the Visual Studio Development Server when you test in Visual Studio during development. This method generally emulates more accurately how your site will run under IIS. However, this method does not test your deployment process or validate that the result of the deployment process will run correctly.
2. Deploy the application to IIS on your development computer by using the same process that you'll use later to deploy it to your production environment. This method validates your deployment process in addition to validating that your application will run correctly under IIS.
3. Deploy the application to a test environment that is as close as possible to your production environment. Since the production environment for these tutorials is a third-party hosting provider, the ideal test environment would be a second account with the hosting provider. You would use this second account only for testing, but it would be set up the same way as the production account.

This tutorial shows the steps for option 2. Guidance for option 3 is provided at the end of the Deploying to the Production Environment tutorial, and at the end of this tutorial there are links to resources for option 1.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

Configuring the Application to Run in Medium Trust

Before installing IIS and deploying to it, you'll change a Web.config file setting in order to make the site run more like it will in a typical shared hosting environment.

Hosting providers typically run your web site in *medium trust*, which means there are some things it is not allowed to do. For example, application code can't access the Windows registry and can't read or write files that are outside of your application's folder hierarchy. By default your application runs in *high trust* on your local computer, which means that the application might be able to do things that would fail when you deploy it to production. Therefore, to make the test environment more accurately reflect the production environment, you'll configure the application to run in medium trust.

In the application Web.config file, add a **trust** element in the **system.web** element, as shown in this example.

```
<configuration>
  <!-- Settings -->
  <system.web>
    <trust level="Medium" />
    <!-- Settings -->
  </system.web>
</configuration>
```

The application will now run in medium trust in IIS even on your local computer. This setting enables you to catch as early as possible any attempts by application code to do something that would fail in production.

> **NOTE**
> If you are using Entity Framework Code First Migrations, make sure that you have version 5.0 or later installed. In Entity Framework version 4.3, Migrations requires full trust in order to update the database schema.

Installing IIS and Web Deploy

To deploy to IIS on your development computer, you must have IIS and Web Deploy installed. These are not included in the default Windows 7 configuration. If you have already installed both IIS and Web Deploy, skip to the next section.

Using the Web Platform Installer is the preferred way to install IIS and Web Deploy, because the Web Platform Installer installs a recommended configuration for IIS and it automatically installs the prerequisites for IIS and Web Deploy if necessary.

To run Web Platform Installer to install IIS and Web Deploy, use the following link. If you already have installed IIS, Web Deploy or any of their required components, the Web Platform Installer installs only what is missing.

- Install IIS and Web Deploy using WebPI

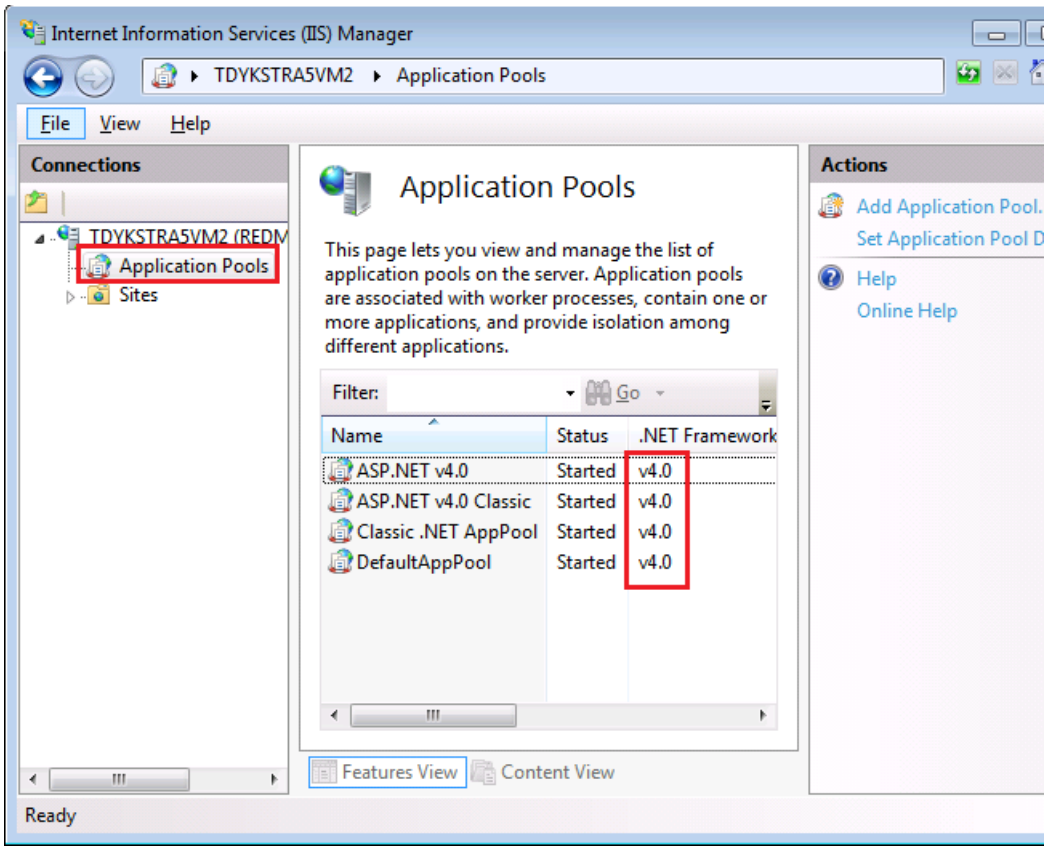
Setting the Default Application Pool to .NET 4

After installing IIS, run **IIS Manager** to make sure that the .NET Framework version 4 is assigned to the default application pool.

From the Windows **Start** menu, select **Run**, enter "inetmgr", and then click **OK**. (If the **Run** command is not in your **Start** menu, you can press the Windows Key and R to open it. Or right-click the taskbar, click **Properties**, select the **Start Menu** tab, click **Customize**, and select **Run command**.)

In the **Connections** pane, expand the server node and select **Application Pools**. In the **Application Pools** pane, if **DefaultAppPool** is assigned to the .NET framework version 4 as in the following illustration, skip to the

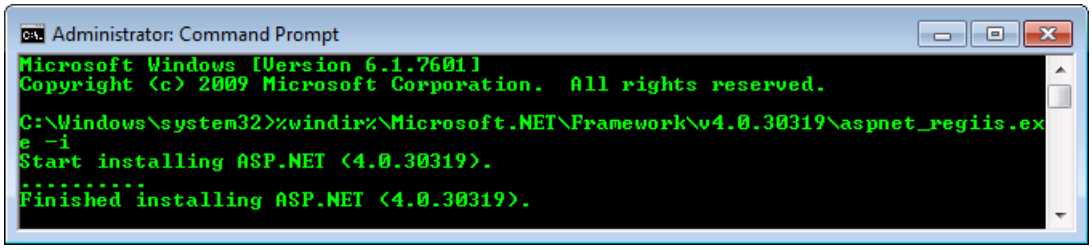
next section.



If you see only two application pools and both of them are set to the .NET Framework 2.0, you have to install ASP.NET 4 in IIS:

- Open a command prompt window by right-clicking **Command Prompt** in the Windows **Start** menu and selecting **Run as Administrator**. Then run aspnet_regiis.exe to install ASP.NET 4 in IIS, using the following commands. (In 64-bit systems, replace "Framework" with "Framework64".)

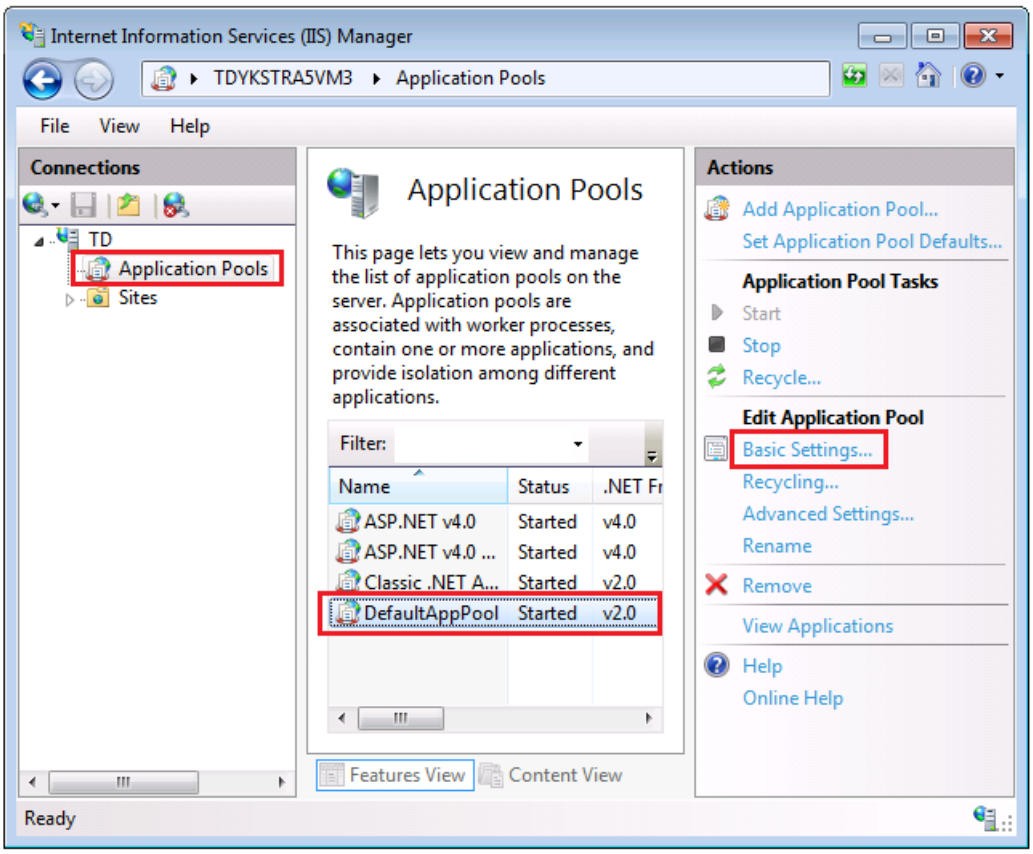
```
cd %windir%\Microsoft.NET\Framework\v4.0.30319
aspnet_regiis.exe -iru
```



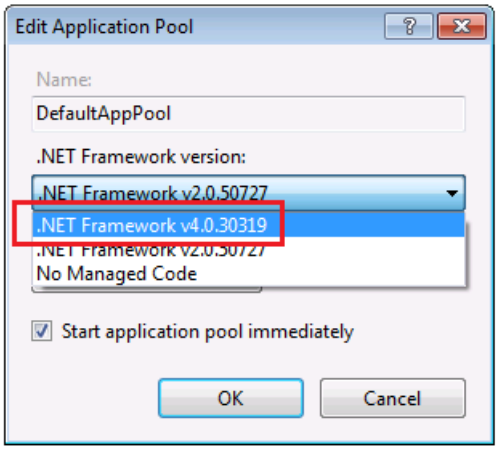
  This command creates new application pools for the .NET Framework 4, but the default application pool will still be set to 2.0. You'll be deploying an application that targets .NET 4 to that application pool, so you have to change the application pool to .NET 4.

If you closed **IIS Manager**, run it again, expand the server node, and click **Application Pools** to display the **Application Pools** pane again.

In the **Application Pools** pane, click **DefaultAppPool**, and then in the **Actions** pane click **Basic Settings**.



In the **Edit Application Pool** dialog box, change **.NET Framework version** to **.NET Framework v4.0.30319** and click **OK**.



You are now ready to publish to IIS.

Publishing to IIS

There are several ways you can deploy using Visual Studio 2010 and Web Deploy:

- Use Visual Studio one-click publish.
- Create a *deployment package* and install it using the IIS Manager UI. The deployment package consists of a *.zip* file that contains all the files and metadata needed to install a site in IIS.
- Create a deployment package and install it using the command line.

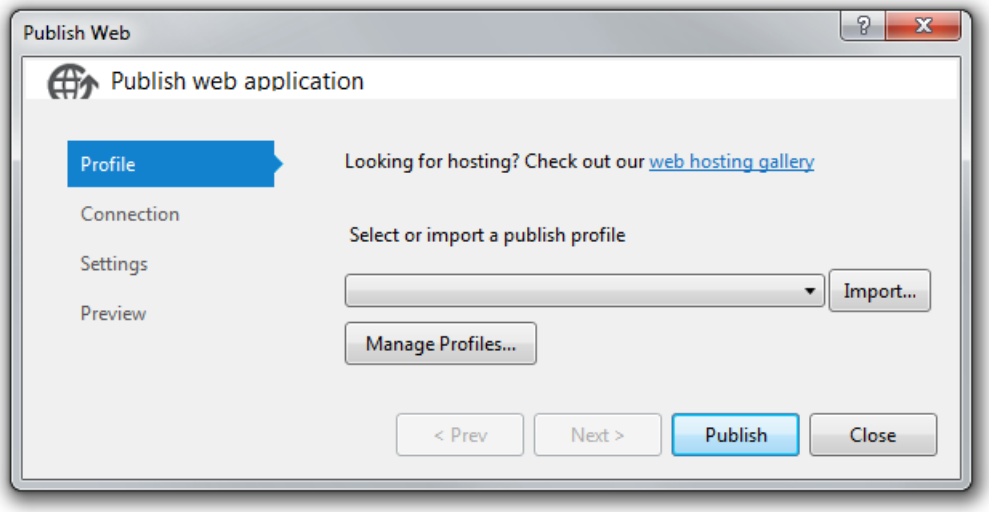
The process you went through in the previous tutorials to set up Visual Studio to automate deployment tasks applies to all of these three methods. In these tutorials you'll use the first of these methods. For information about using deployment packages, see ASP.NET Deployment Content Map.

Before publishing, make sure that you are running Visual Studio in administrator mode. (In the Windows 7 **Start**

menu, right-click the icon for the version of Visual Studio you're using and select **Run as Administrator**.) Administrator mode is required for publishing only when you are publishing to IIS on the local computer.

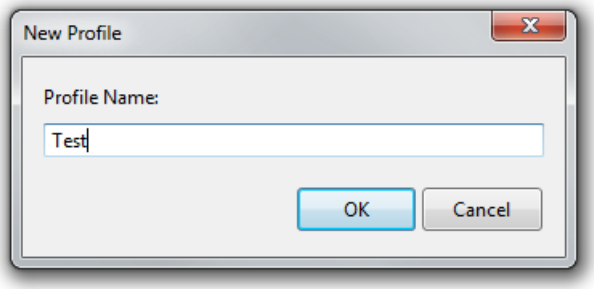
In **Solution Explorer**, right-click the ContosoUniversity project (not the ContosoUniversity.DAL project) and select **Publish**.

The **Publish Web** wizard appears.



In the drop-down list, select **<New...>**.

In the **New Profile** dialog box, enter "Test", and then click **OK**.



This name is the same as the middle node of the Web.Test.config transform file that you created earlier. This correspondence is what causes the Web.Test.config transformations to be applied when you publish by using this profile.

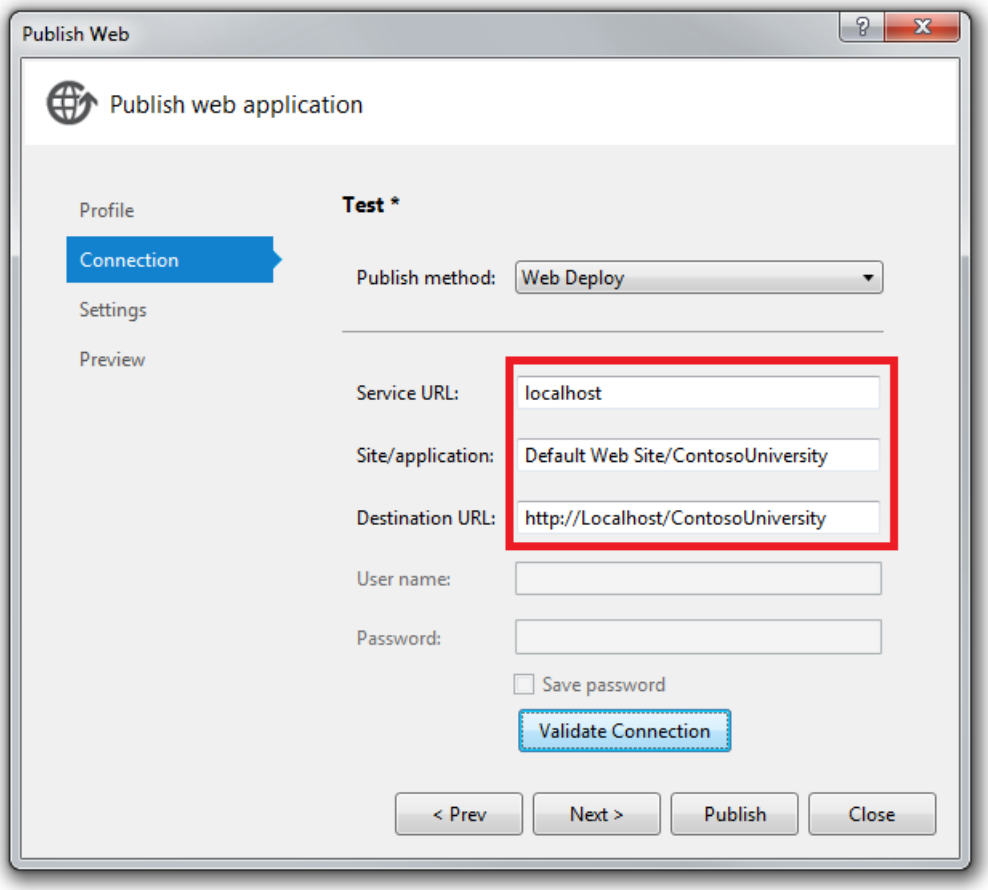
The wizard automatically advances to the **Connection** tab.

In the **Service URL** box, enter *localhost*.

In the **Site/application** box, enter *Default Web Site/ContosoUniversity*.

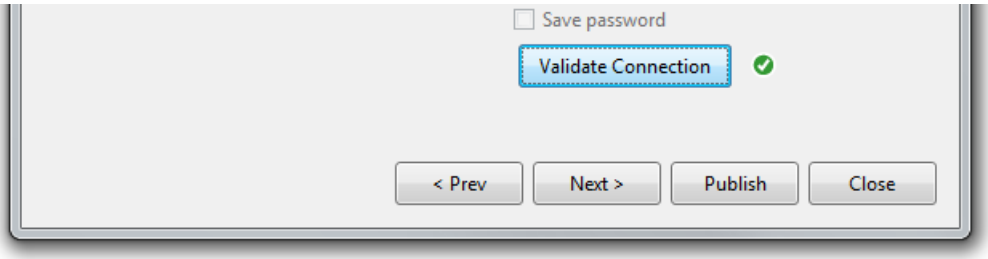
In the **Destination URL** box, enter `http://localhost/ContosoUniversity`.

The **Destination URL** setting isn't required. When Visual Studio finishes deploying the application, it automatically opens your default browser to this URL. If you don't want the browser to open automatically after deployment, leave this box blank.



Click **Validate Connection** to verify that the settings are correct and you can connect to IIS on the local computer.

A green check mark verifies that the connection is successful.



Click **Next** to advance to the **Settings** tab.

The **Configuration** drop-down box specifies the build configuration to deploy. The default value is Release, which is what you want.

Leave the **Remove additional files at destination** check box cleared. Since this is your first deployment, there won't be any files in the destination folder yet.

In the **Databases** section, enter the following value in the connection string box for **SchoolContext**:

```
Data Source=|DataDirectory|School-Prod.sdf
```

The deployment process will put this connection string in the deployed Web.config file because **Use this connection string at runtime** is selected.

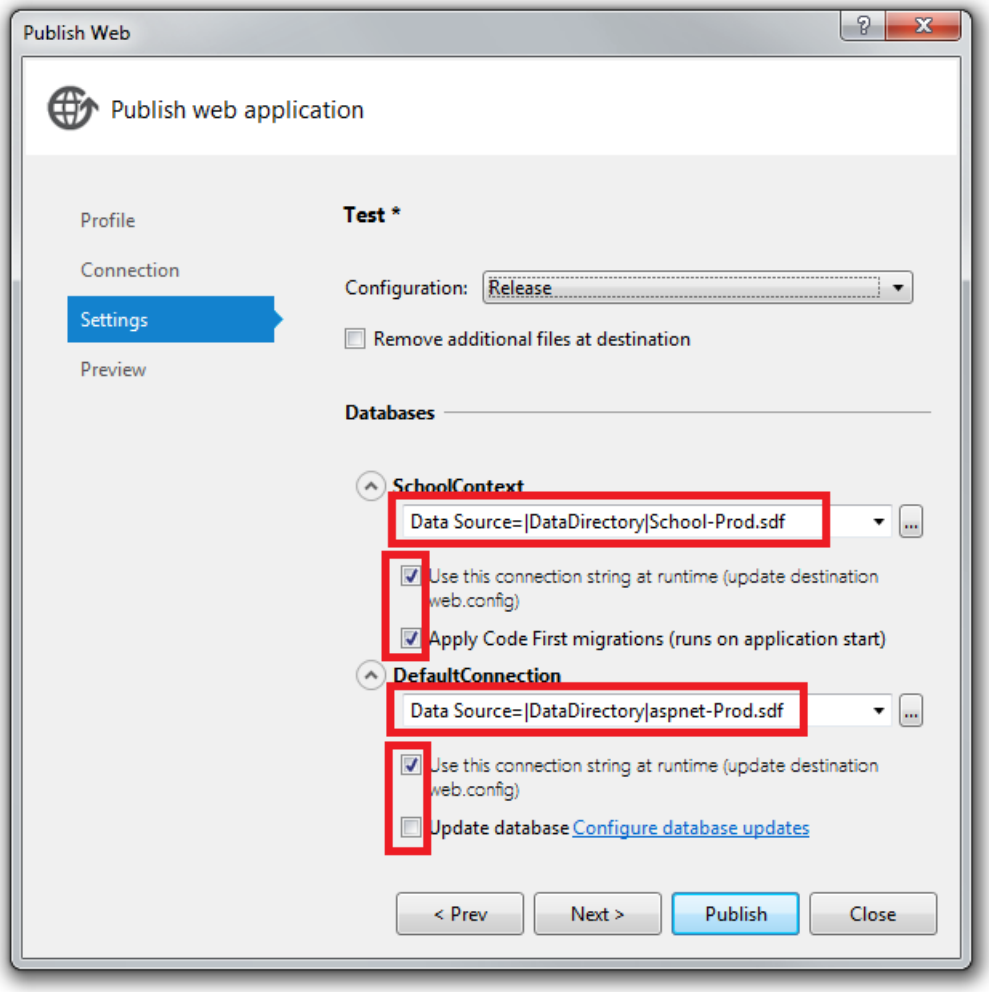
Also under **SchoolContext**, select **Apply Code First Migrations**. This option causes the deployment process to configure the deployed Web.config file to specify the `MigrateDatabaseToLatestVersion` initializer. This initializer automatically updates the database to the latest version when the application accesses the database for the first

time after deployment.

In the connection string box for **DefaultConnection**, enter the following value:

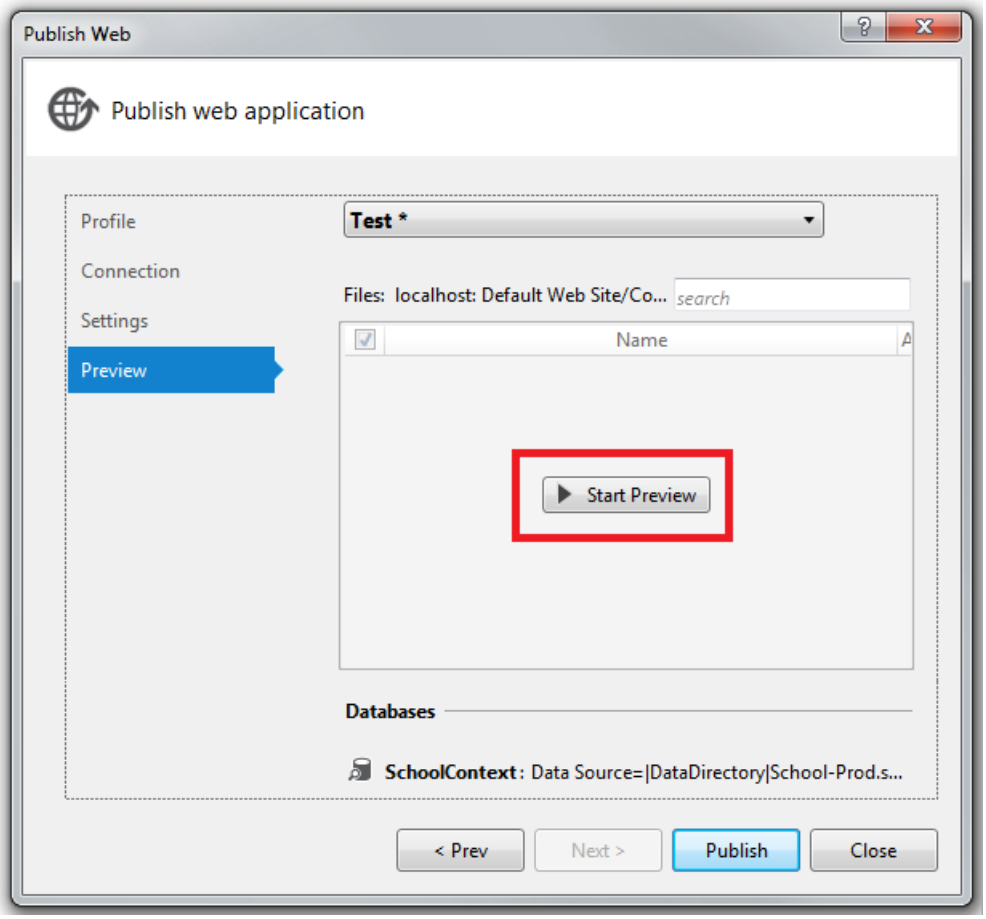
```
Data Source=|DataDirectory|aspnet-Prod.sdf
```

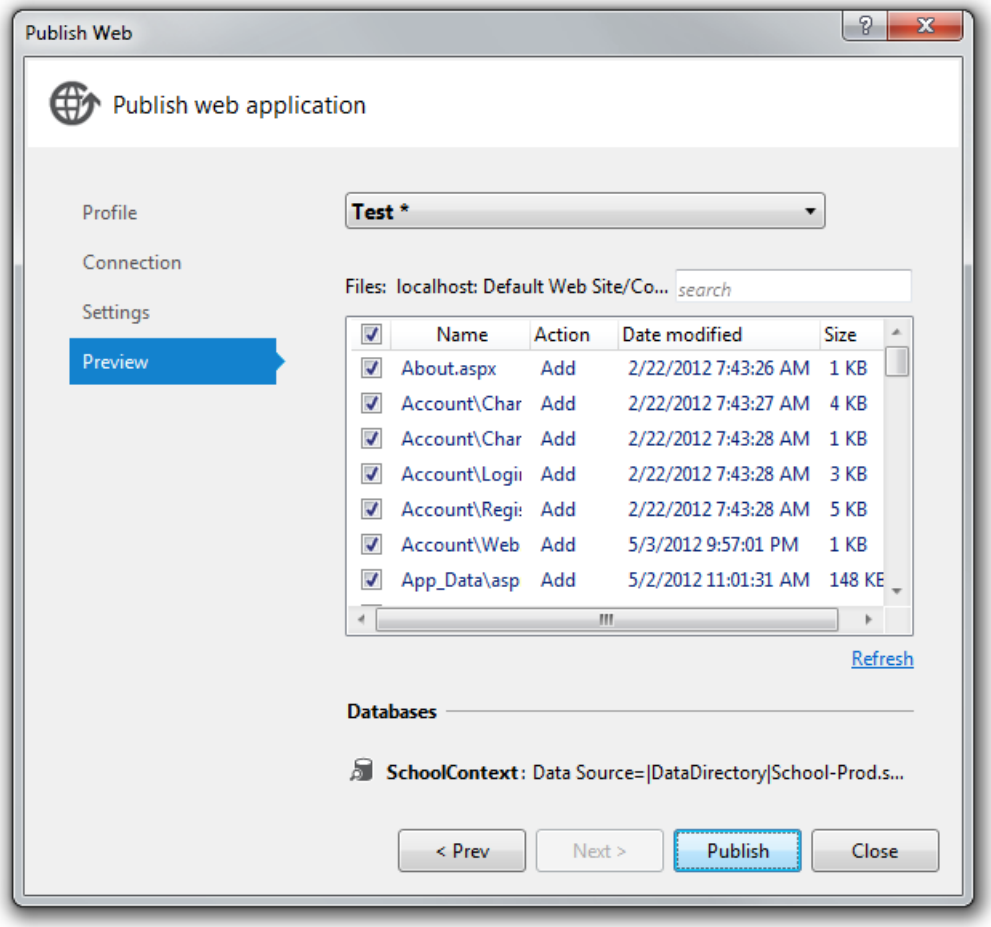
Leave **Update database** cleared. The membership database will be deployed by copying the .sdf file in App_Data, and you don't want the deployment process to do anything else with this database.



Click **Next** to advance to the **Preview** tab.

In the **Preview** tab, click **Start Preview** to see a list of the files that will be copied.

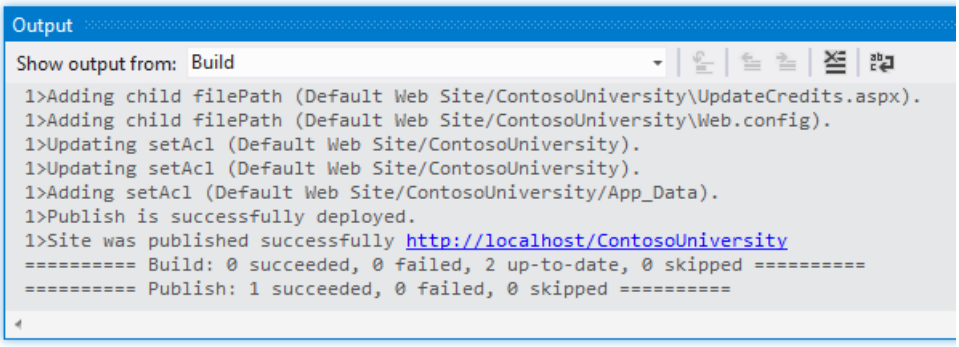




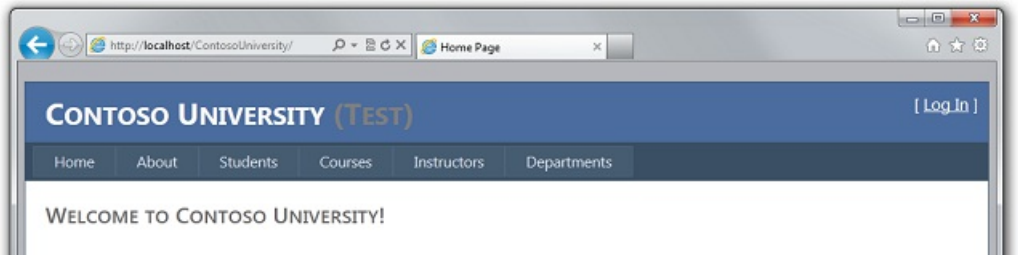
Click **Publish**.

If Visual Studio is not in administrator mode, you might get an error message that indicates a permissions error. In that case, close Visual Studio, open it in administrator mode, and try to publish again.

If Visual Studio is in administrator mode, the **Output** window reports successful build and publish.

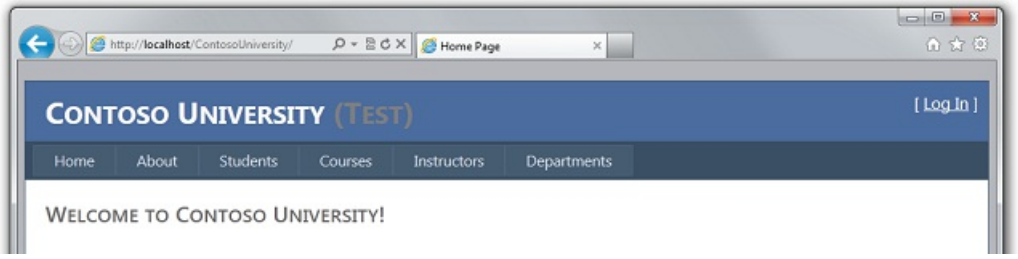


The browser automatically opens to the Contoso University Home page running in IIS on the local computer.

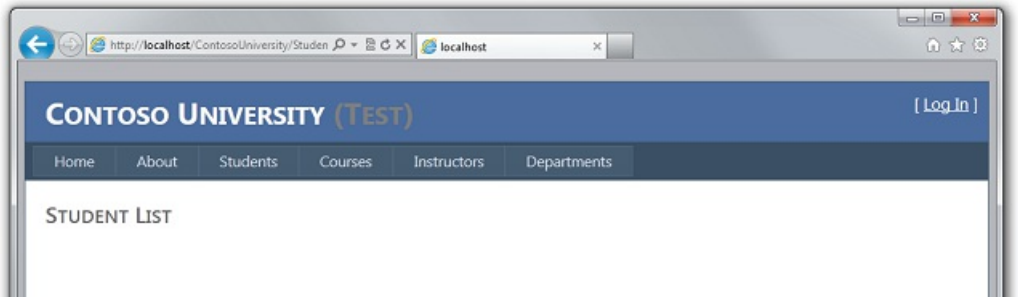


Testing in the Test Environment

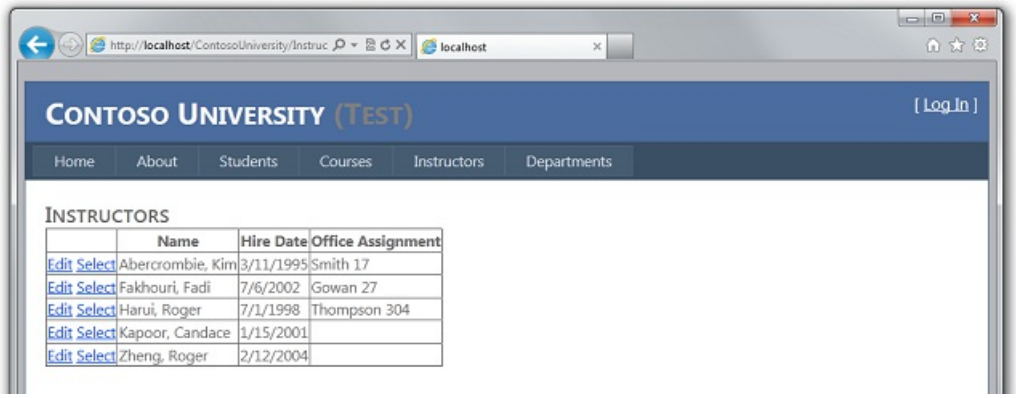
Notice that the environment indicator shows "(Test)" instead of "(Dev)", which shows that the *Web.config* transformation for the environment indicator was successful.



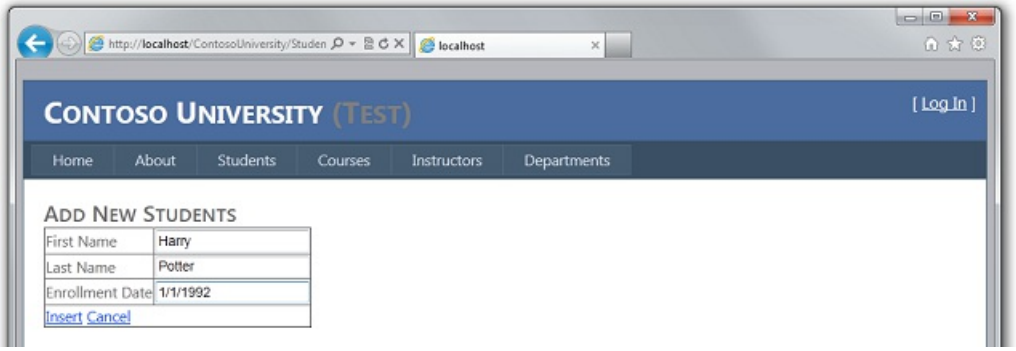
Run the **Students** page to verify that the deployed database has no students. When you select this page it may take a few minutes to load because Code First creates the database and then runs the `Seed` method. (It didn't do that when you were on the home page because the application didn't try to access the database yet.)

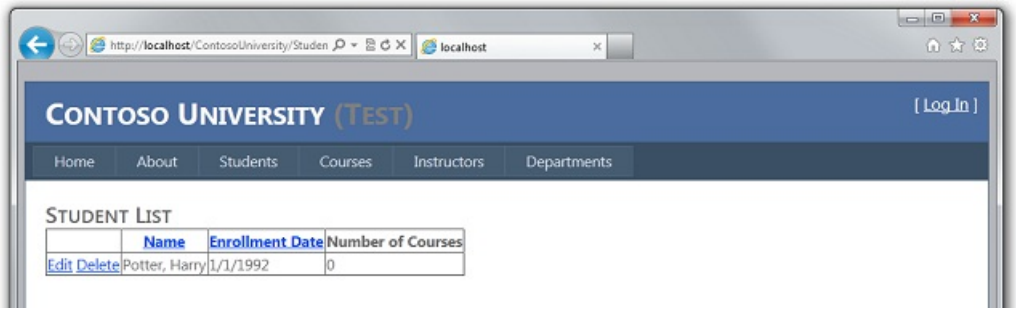


Run the **Instructors** page to verify that Code First seeded the database with instructor data:

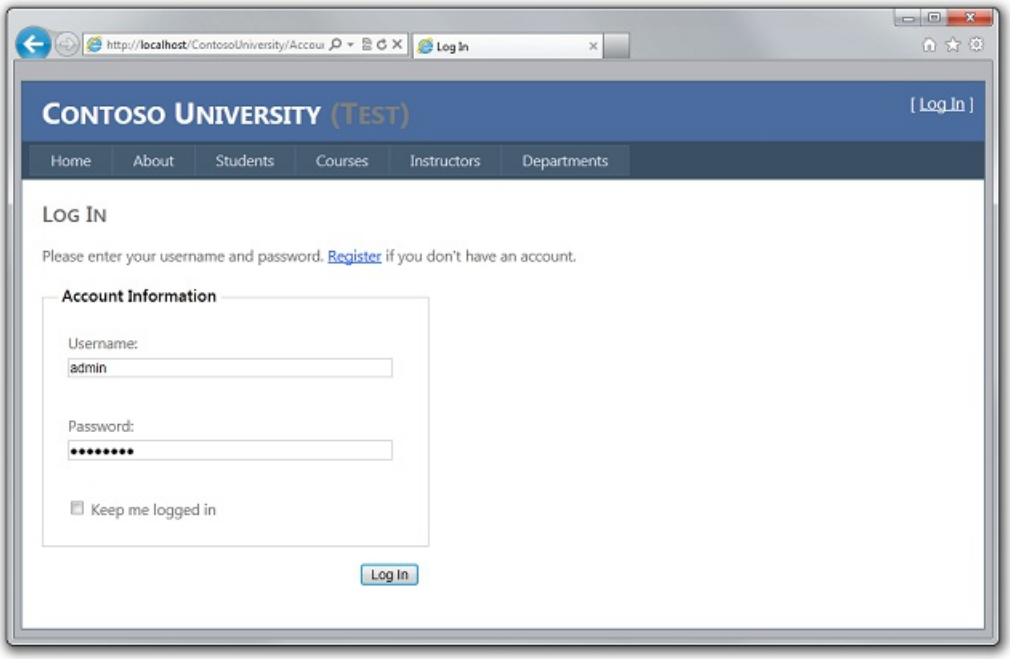


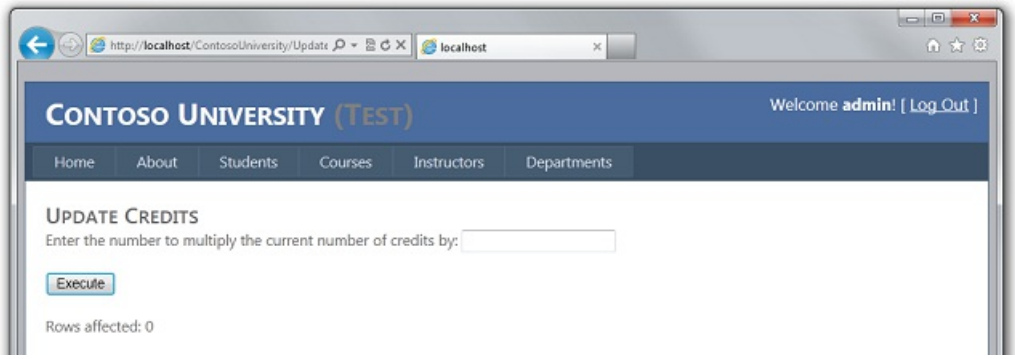
Select **Add Students** from the **Students** menu, add a student, and then view the new student in the **Students** page to verify that you can successfully write to the database:



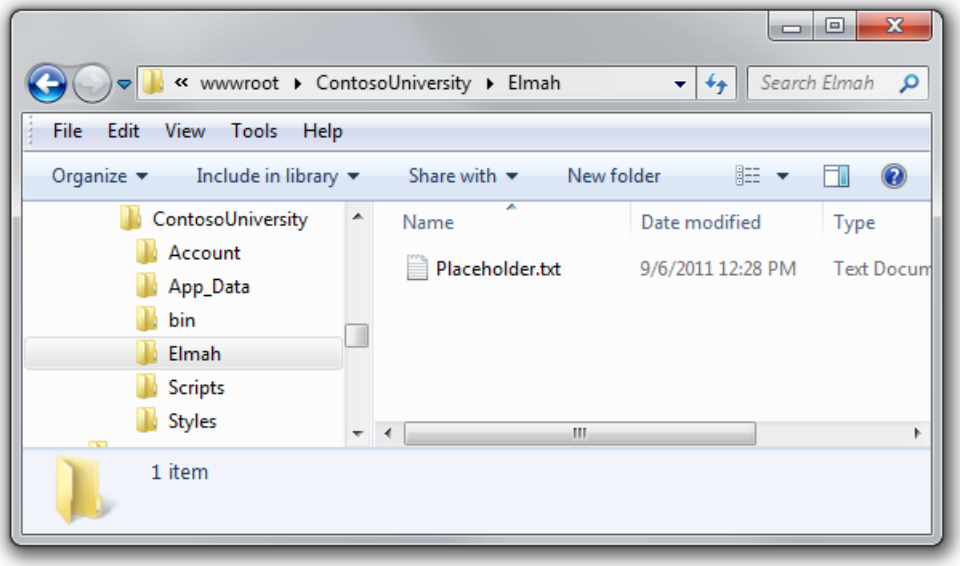


From the **Courses** menu, select **Update Credits**. The **Update Credits** page requires administrator permissions, so the **Log In** page is displayed. Enter the administrator account credentials that you created earlier ("admin" and "Pas$w0rd"). The **Update Credits** page is displayed, which verifies that the administrator account that you created in the previous tutorial was correctly deployed to the test environment.





Verify that an *Elmah* folder exists with only the placeholder file in it.



Reviewing the Automatic Web.config Changes for Code First Migrations

Open the *Web.config* file in the deployed application at *C:\inetpub\wwwroot\ContosoUniversity* and you can see where the deployment process configured Code First Migrations to automatically update the database to the latest version.

```
<entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlCeConnectionFactor
        <parameters>
            <parameter value="System.Data.SqlServerCe.4.0" />
        </parameters>
    </defaultConnectionFactory>
    <contexts>
        <context type="ContosoUniversity.DAL.SchoolContext, ContosoUniversity.DAL">
            <databaseInitializer type="System.Data.Entity.MigrateDatabaseToLatestVersion`2[
                <parameters>
                    <parameter value="SchoolContext_DatabasePublish" />
                </parameters>
            </databaseInitializer>
        </context>
    </contexts>
</entityFramework>
```

The deployment process also created a new connection string for Code First Migrations to use exclusively for updating the database schema:

```
<connectionStrings>
  <add name="DefaultConnection" connectionString="|DataDirectory|aspnet.sdf" providerName=
  <add name="SchoolContext" connectionString="Data Source=|DataDirectory|School.sdf" provi
  <add name="SchoolContext_DatabasePublish" connectionString="Data Source=|DataDirectory|S
</connectionStrings>
```

This additional connection string enables you to specify one user account for database schema updates, and a different user account for application data access. For example, you could assign the db_owner role to Code First Migrations, and db_datareader and db_datawriter roles to the application. This is a common defense-in-depth pattern that prevents potentially malicious code in the application from changing the database schema. (For example, this might happen in a successful SQL injection attack.) This pattern is not used by these tutorials. It does not apply to SQL Server Compact, and it does not apply when you migrate to SQL Server in a later tutorial in this series. The Cytanium site offers just one user account for accessing the SQL Server database that you create at Cytanium. If you are able to implement this pattern in your scenario, you can do it by performing the following steps:

1. In the **Settings** tab of the **Publish Web** wizard, enter the connection string that specifies a user with full database schema update permissions, and clear the **Use this connection string at runtime** check box. In the deployed Web.config file, this becomes the `DatabasePublish` connection string.
2. Create a Web.config file transformation for the connection string that you want the application to use at run time.

You have now deployed your application to IIS on your development computer and tested it there. This verifies that the deployment process copied the application's content to the right location (excluding the files that you did not want to deploy), and also that Web Deploy configured IIS correctly during deployment. In the next tutorial, you'll run one more test that finds a deployment task that has not yet been done: setting folder permissions on the *Elmah* folder.

More Information

For information about running IIS or IIS Express in Visual Studio, see the following resources:

- IIS Express Overview on the IIS.net site.
- Introducing IIS Express on Scott Guthrie's blog.
- How to: Specify the Web Server for Web Projects in Visual Studio.
- Core Differences Between IIS and the ASP.NET Development Server on the ASP.NET site.
- Test your ASP.NET MVC or Web Forms Application on IIS 7 in 30 seconds on Rick Anderson's blog. This entry provides examples of why testing with the Visual Studio Development Server (Cassini) is not as reliable as testing in IIS Express, and why testing in IIS Express is not as reliable as testing in IIS.

For information about what issues might arise when your application runs in medium trust, see Hosting ASP.NET Applications in Medium Trust on the 4 Guys from Rolla site.

PREVIOUS NEXT

Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio or Visual Web Developer: Setting Folder Permissions - 6 of 12

2/20/2020 • 4 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This series of tutorials shows you how to deploy (publish) an ASP.NET web application project that includes a SQL Server Compact database by using Visual Studio 2012 RC or Visual Studio Express 2012 RC for Web. You can also use Visual Studio 2010 if you install the Web Publish Update. For an introduction to the series, see the first tutorial in the series.
>
> For a tutorial that shows deployment features introduced after the RC release of Visual Studio 2012, shows how to deploy SQL Server editions other than SQL Server Compact, and shows how to deploy to Azure App Service Web Apps, see ASP.NET Web Deployment using Visual Studio.

Overview

In this tutorial, you set folder permissions for the *Elmah* folder in the deployed web site so that the application can create log files in that folder.

When you test a web application in Visual Studio using the Visual Studio Development Server (Cassini), the application runs under your identity. You are most likely an administrator on your development computer and have full authority to do anything to any file in any folder. But when an application runs under IIS, it runs under the identity defined for the application pool that the site is assigned to. This is typically a system-defined account that has limited permissions. By default it has read and execute permissions on your web application's files and folders, but it doesn't have write access.

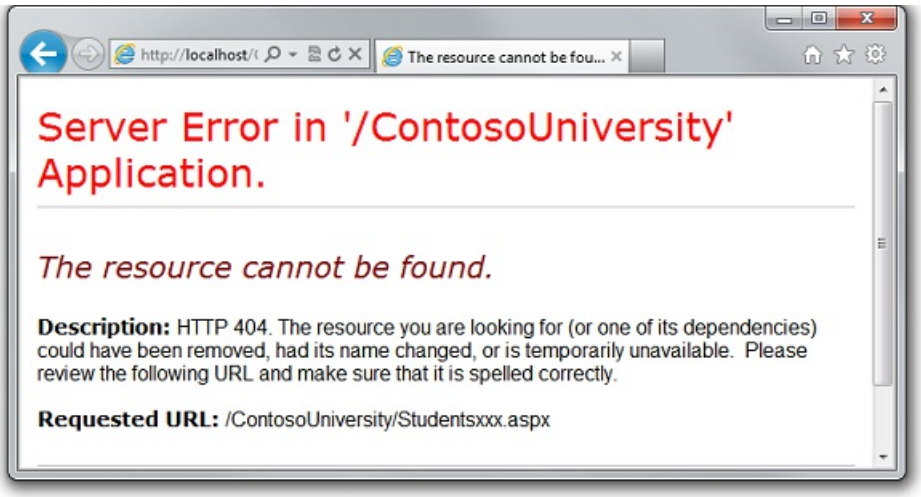
This becomes an issue if your application creates or updates files, which is a common need in web applications. In the Contoso University application, Elmah creates XML files in the *Elmah* folder in order to save details about errors. Even if you don't use something like Elmah, your site might let users upload files or perform other tasks that write data to a folder in your site.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

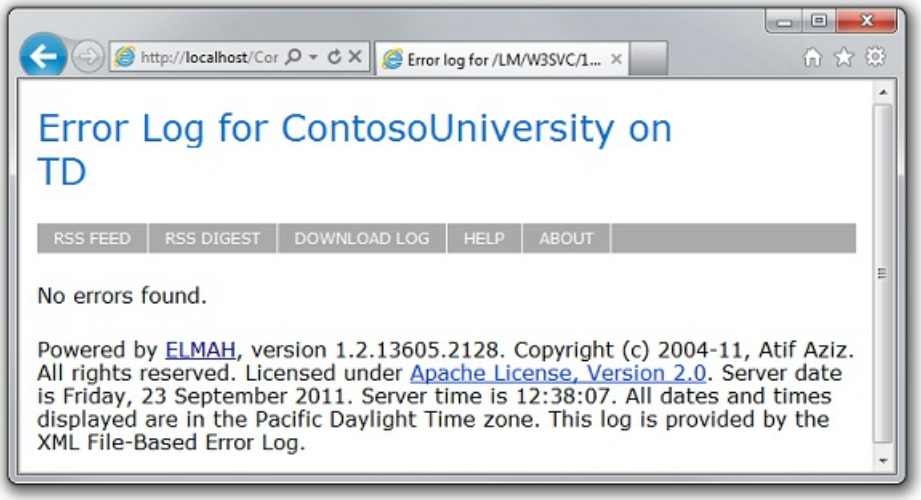
Testing Error Logging and Reporting

To see how the application doesn't work correctly in IIS (although it did when you tested it in Visual Studio), you can cause an error that would normally be logged by Elmah, and then open the Elmah error log to see the details. If Elmah was unable to create an XML file and store the error details, you see an empty error report.

Open a browser and go to `http://localhost/ContosoUniversity`, and then request an invalid URL like *Studentsxxx.aspx*. You see a system-generated error page instead of the *GenericErrorPage.aspx* page because the `customErrors` setting in the Web.config file is "RemoteOnly" and you are running IIS locally:



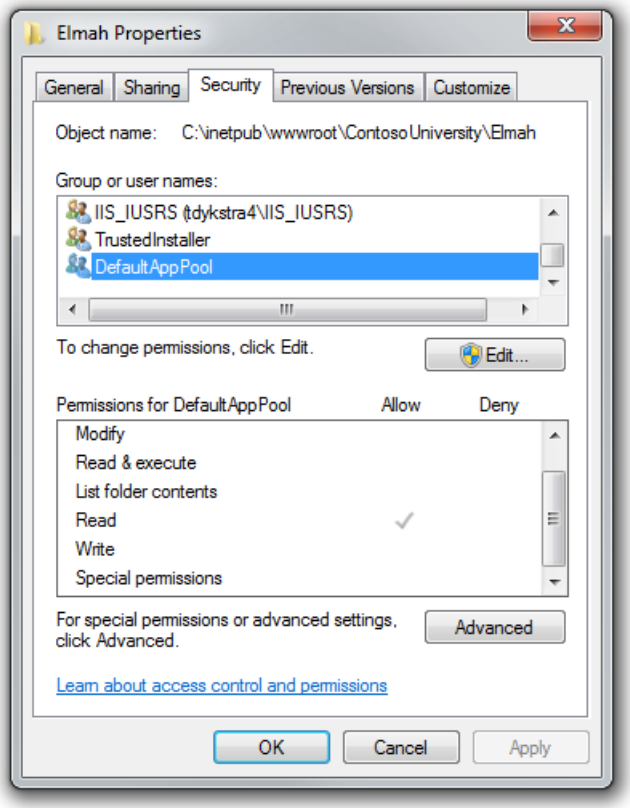
Now run *Elmah.axd* to see the error report. You see an empty error log page because Elmah was unable to create an XML file in the *Elmah* folder:



Setting Write Permission on the Elmah Folder

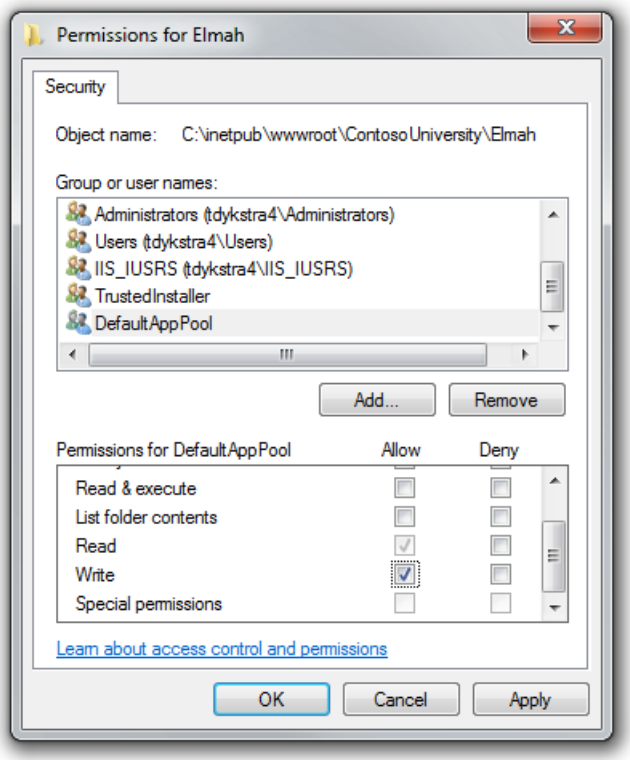
You can set folder permissions manually or you can make it an automatic part of the deployment process. Making it automatic requires complex MSBuild code, and since you only have to do this the first time you deploy, this tutorial only shows how to do it manually. (For information about how to make this part of the deployment process, see Setting Folder Permissions on Web Publish on Sayed Hashimi's blog.)

In **Windows Explorer**, navigate to *C:\inetpub\wwwroot\ContosoUniversity*. Right-click the *Elmah* folder, select **Properties**, and then select the **Security** tab.



(If you don't see **DefaultAppPool** in the **Group or user names** list, you probably used some other method than the one specified in this tutorial to set up IIS and ASP.NET 4 on your computer. In that case, find out what identity is used by the application pool assigned to the Contoso University application, and grant write permission to that identity. See the links about application pool identities at the end of this tutorial.)

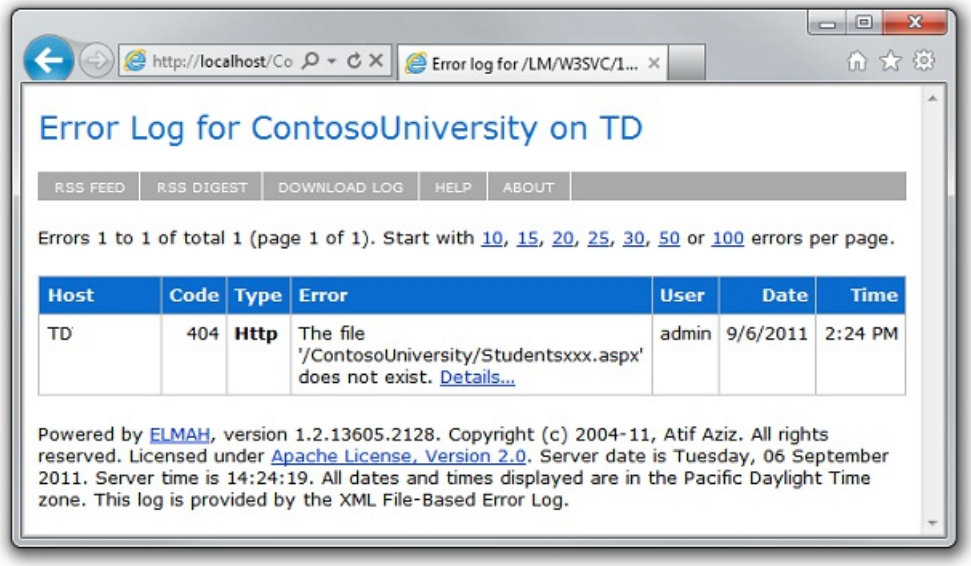
Click **Edit**. In the **Permissions for Elmah** dialog box, select **DefaultAppPool**, and then select the **Write** check box in the **Allow** column.



Click **OK** in both dialog boxes.

Retesting Error Logging and Reporting

Test by causing an error again in the same way (request a bad URL) and run the **Error Log** page. This time the error appears on the page.



You also need write permission on the *App_Data* folder because you have SQL Server Compact database files in that folder, and you want to be able to update data in those databases. In that case, however, you don't have to do anything extra because the deployment process automatically sets write permission on the *App_Data* folder.

You have now completed all of the tasks necessary to get Contoso University working correctly in IIS on your local computer. In the next tutorial, you will make the site publicly available by deploying it to a hosting provider.

More Information

In this example, the reason why Elmah was unable to save log files was fairly obvious. You can use IIS tracing in cases where the cause of the problem is not so obvious; see Troubleshooting Failed Requests Using Tracing in IIS 7 on the IIS.net site.

For more information about how to grant permissions to application pool identities, see Application Pool Identities and Secure Content in IIS Through File System ACLs on the IIS.net site.

PREVIOUS NEXT

Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio or Visual Web Developer: Deploying to the Production Environment - 7 of 12

2/20/2020 • 12 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This series of tutorials shows you how to deploy (publish) an ASP.NET web application project that includes a SQL Server Compact database by using Visual Studio 2012 RC or Visual Studio Express 2012 RC for Web. You can also use Visual Studio 2010 if you install the Web Publish Update. For an introduction to the series, see the first tutorial in the series.
>
> For a tutorial that shows deployment features introduced after the RC release of Visual Studio 2012, shows how to deploy SQL Server editions other than SQL Server Compact, and shows how to deploy to Azure App Service Web Apps, see ASP.NET Web Deployment using Visual Studio.

Overview

In this tutorial, you set up an account with a hosting provider and deploy your ASP.NET web application to the production environment by using the Visual Studio one-click publish feature.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

Selecting a Hosting Provider

For the Contoso University application and this tutorial series, you need a provider that supports ASP.NET 4 and Web Deploy. A specific hosting company was chosen so that the tutorials could illustrate the complete experience of deploying to a live website. Each hosting company provides different features, and the experience of deploying to their servers varies somewhat. However, the process described in this tutorial is typical for the overall process. The hosting provider used for this tutorial, Cytanium.com, is one of many that are available, and its use in this tutorial does not constitute an endorsement or recommendation.

When you are ready to select your own hosting provider, you can compare features and prices in the gallery of providers on the Microsoft.com/web site.

Creating an Account

Create an account at your selected provider. If support for a full SQL Server database is an added extra, you do not need to select it for this tutorial, but you'll need it for the Migrating to SQL Server tutorial later in this series.

For these tutorials, you don't have to register a new domain name. You can test to verify successful deployment by using the temporary URL assigned to the site by the provider.

After the account has been created, you typically receive a welcome email that contains all the information you need in order to deploy and manage your site. The information that your hosting provider sends you will be

similar to what is shown here. The Cytanium welcome email that's sent to new account owners includes the following information:

- The URL to the provider's control panel site, where you can manage settings for your site. The ID and password you specified are included in this part of the welcome email for easy reference. (Both have been changed to a demo value for this illustration.)

  Control Panel URL

Control Panel URL	Username	Password
http://panel.cytanium.com	contoso	contoso

- The default .NET Framework version and information about how to change it. Many hosting sites default to 2.0, which works with ASP.NET applications that target the .NET Framework 2.0, 3.0, or 3.5. However Contoso University is a .NET Framework 4 application, so you have to change this setting. (For an ASP.NET 4.5 application you would use the .NET 4.0 setting.)

  Web

  Web Settings

  This account is setup for ASP.NET 2.0 (3.5 SP1) in IIS 7 Integrated mode.
  Instructions on how to change framework version can be found here.

  Need help on selecting the correct .NET version? See guidance below:

  - **.NET 3.5 – works with Web Sites built using "Site from Gallery" option (e.g. – DotNetNuke, Umbraco, etc...)**
  - **.NET 4.0 – works with applications built using "Site from Template" option (e.g. – Starter Site, Bakery, etc...)**

- The temporary URL that you can use to access your web site. When this account was created, "contosouniversity.com" was entered as the existing domain name. Therefore the temporary URL is `http://contosouniversity.com.vserver01.cytanium.com`.

  Temporary URL

  You can access your web sites right now using their respective temporary URLs (instant aliases). Temporary URL is a sub-domain of the form http://<yourdomain.com>.vserver01.cytanium.com where <yourdomain.com> is your domain.

- Information about how to set up databases, and the connection strings that you need in order to access them:

Databases

SQL CE databases are automatically available on any hosting plan. You can create/upload any number of SQL CE databases from File Manager in Control Panel.

If you purchased a MySQL or MS SQL database, you can create the database via the Control Panel. Further instructions can be found here.

After creating a SQL Server or MySQL database you can access it using one of the following methods:

- **Database Manager using IIS Manager**
- **Remotely via SQL Management Stuido or MySQL Workbench**
- **directly from code**

We've provided two example connection strings below:

- **SQL Server: Data Source=vserver01.cytanium.com;Initial Catalog={myDataBase};User Id={myUsername};Password={myPassword};**
- **MySQL: Server=vserver01.cytanium.com;Database={myDataBase};Uid={myUsername};Pwd={myPassword};**

- Information about tools and settings for deploying your site. (The email from Cytanium also mentions WebMatrix, which is omitted here.)

Deploy and Manage Your Site

You can deploy and manage your site using:

- **WebMatrix**
- **Visual Studio 2010**
- **IIS 7 Manager 32-bit**
- **IIS 7 Manager 64-bit**
- **WebDeploy (MSDeploy)**
- **Cytanium Control Panel**
- **FTP**

Following are field instructions for WebMatrix and Visual Studio 2010:

Visual Studio 2010
  Publish method: Web Deploy
  Service URL: https://vserver01.cytanium.com:8172/MsDeploy.axd
  Site/application: <yourdomain.com>
  Allow untrusted certificate (check)
  User name: contoso
  Password: contoso

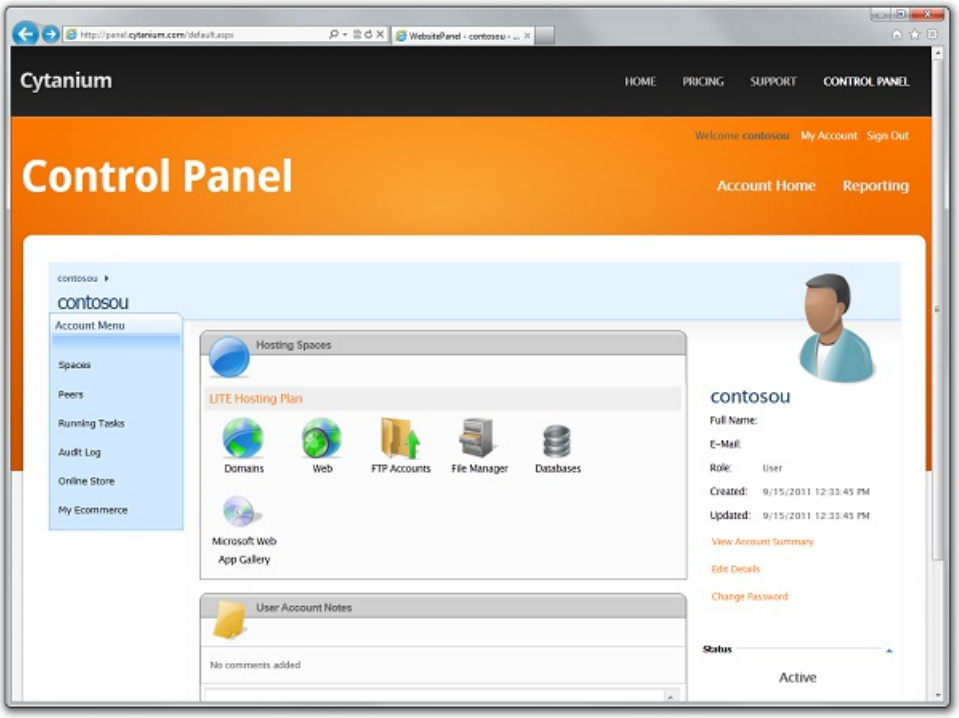
FTP Standalone client (FileZilla example)
  Host: vserver01.cytanium.com
  Server Type: FTPES - FTP over explicit TLS/SSL
  Logon Type: Normal
  User: contoso
  Password: contoso
  Transfer Settings tab: Set Transfer mode to Passive

Setting the .NET Framework Version

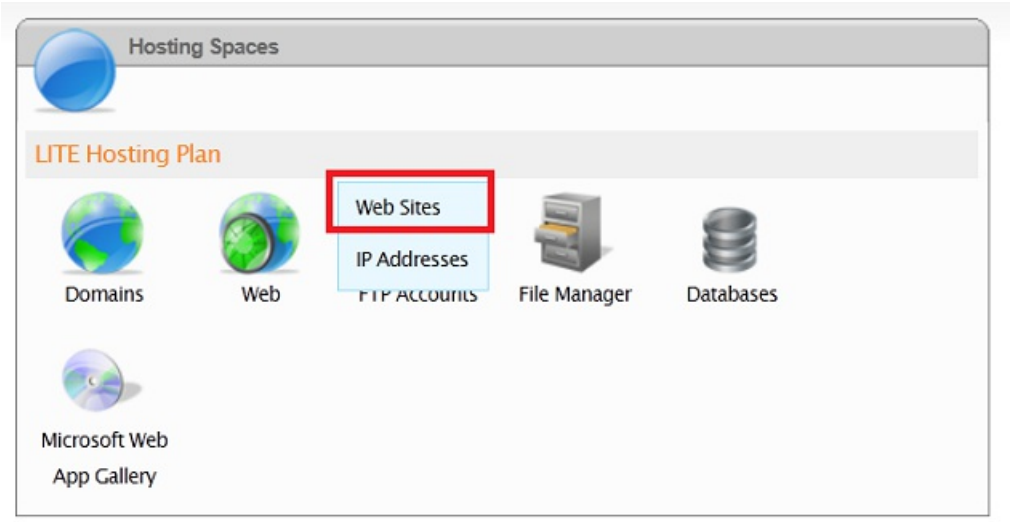
The Cytanium welcome email includes a link to instructions on how to change the version of the .NET Framework. These instructions explain that this can be done through the Cytanium control panel. Other

providers have control panel sites that look different, or they may instruct you to do this in a different way.

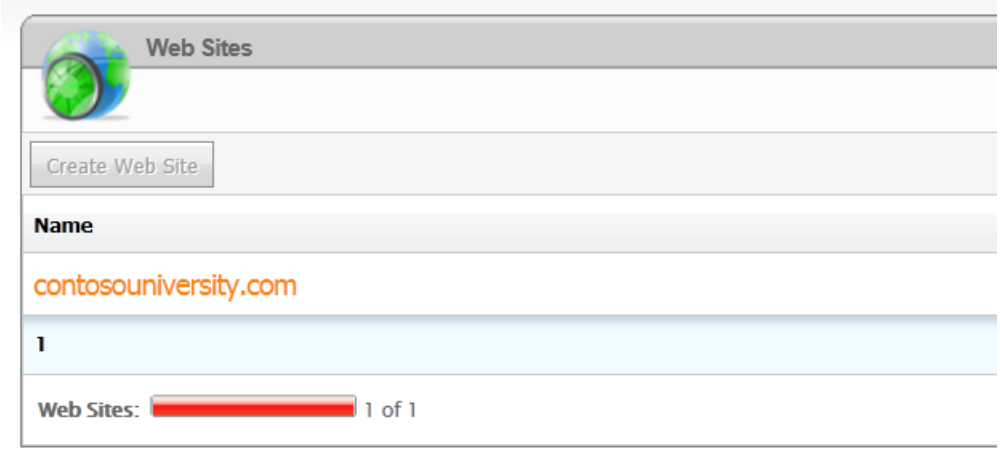
Go to the control panel URL. After logging in with your user name and password, you see the control panel.



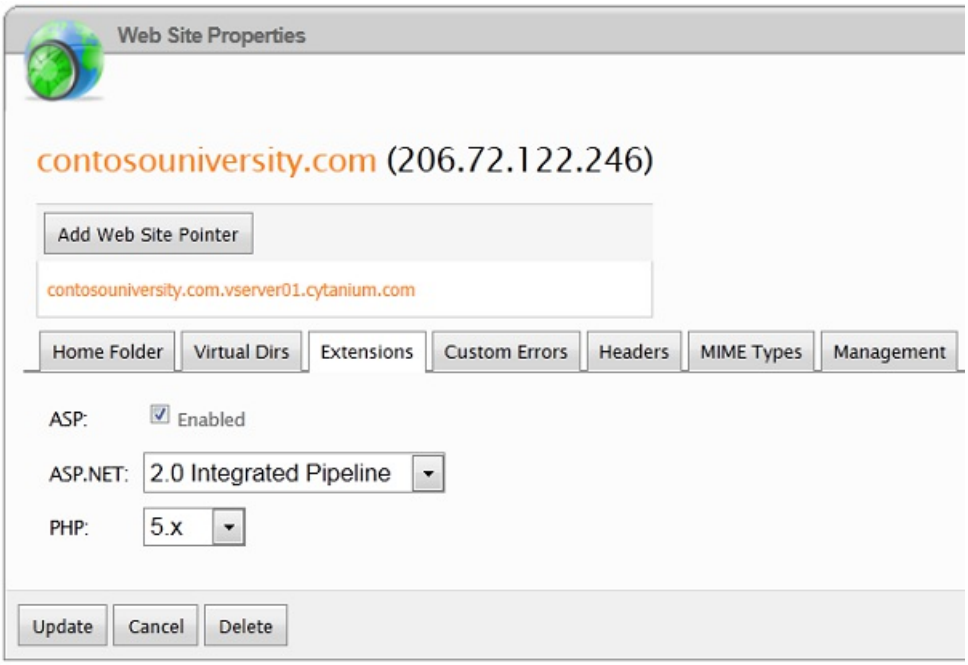
In the **Hosting Spaces** box, hold the pointer over the Web icon and select **Web Sites** from the menu.



In the **Web Sites** box, click **contosouniversity.com** (the name of the site that you used when you created the account).



In the **Web Site Properties** box, select the **Extensions** tab.

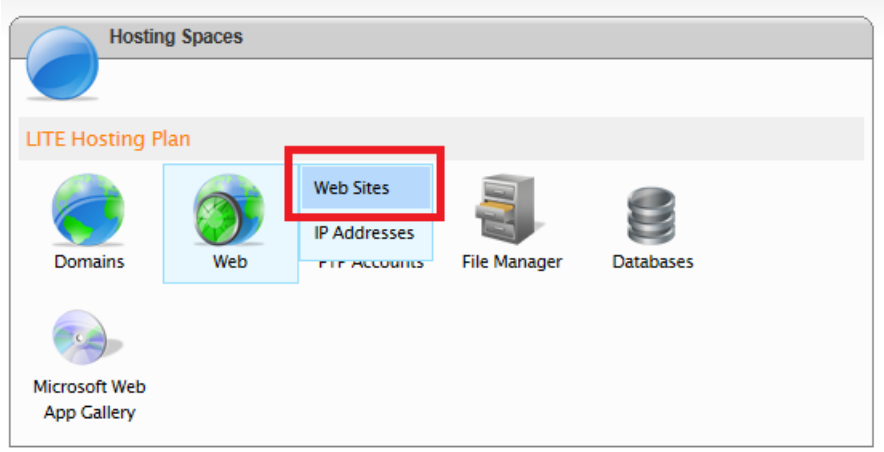


Change ASP.NET from **2.0 Integrated Pipeline** to **4.0 (Integrated Pipeline)**, and then click **Update**.

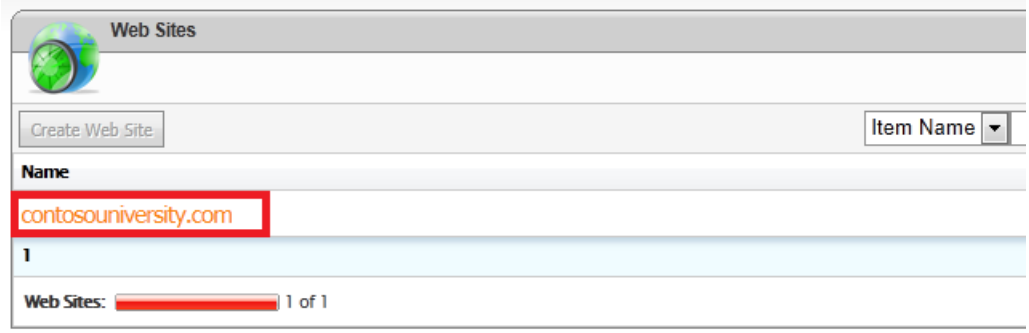
Publishing to the Hosting Provider

The welcome email from the hosting provider includes all of the settings you need in order to publish the project, and you can enter that information manually into a publish profile. But you'll use an easier and less error-prone method to configure deployment to the provider: you'll download a *.publishsettings* file and import it into a publish profile.

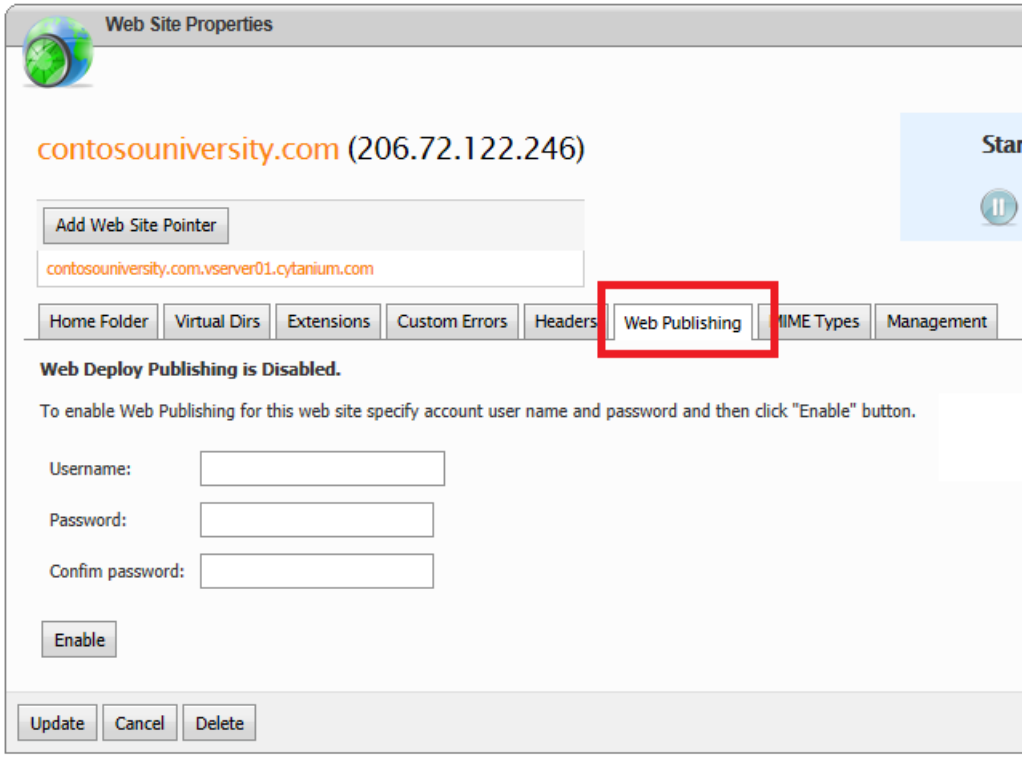
In your browser, go to the Cytanium control panel and select **Web** and then select **Web Sites**.



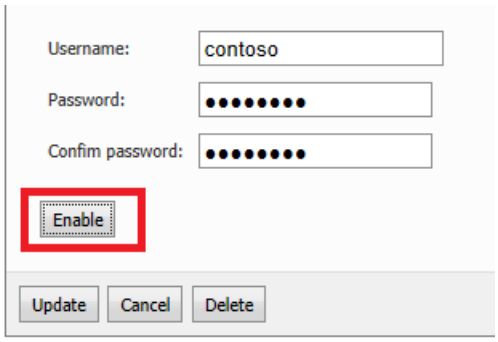
Select the **contosouniversity.com** web site.



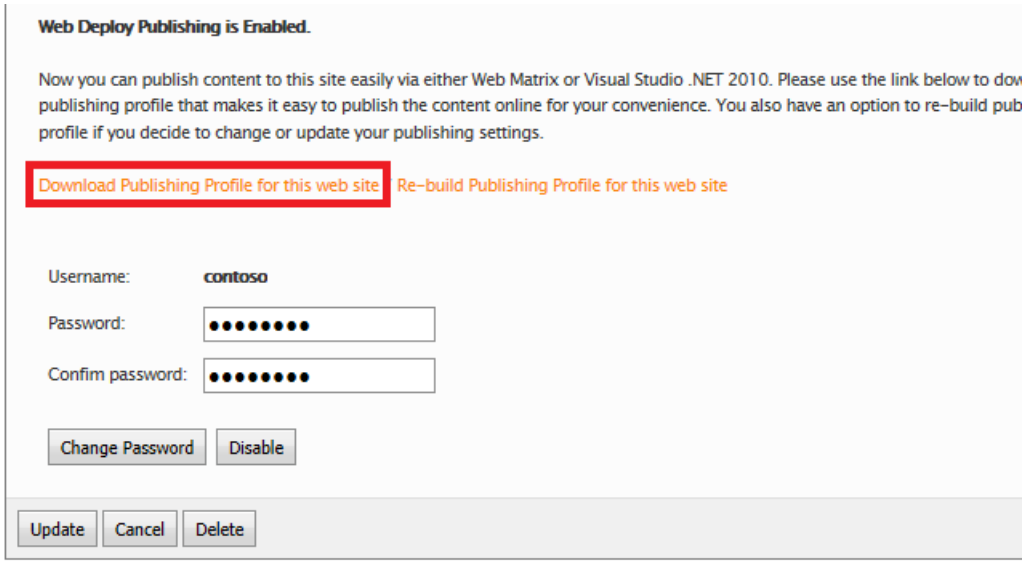
Select the **Web Publishing** tab.



Create credentials to use for web publishing by entering a user name and password. You can enter the same credentials that you use to log on to the control panel. Then click **Enable**.



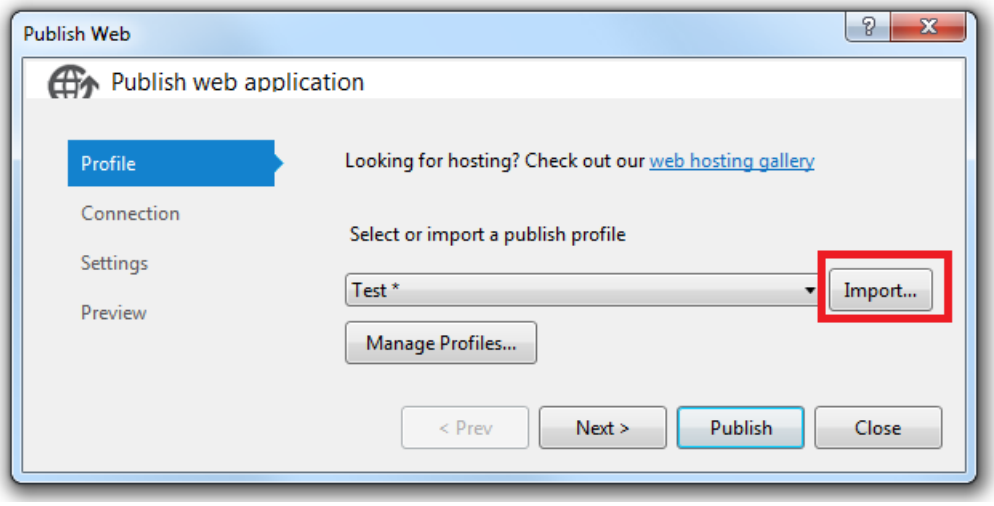
Click **Download Publishing Profile for this web site**.



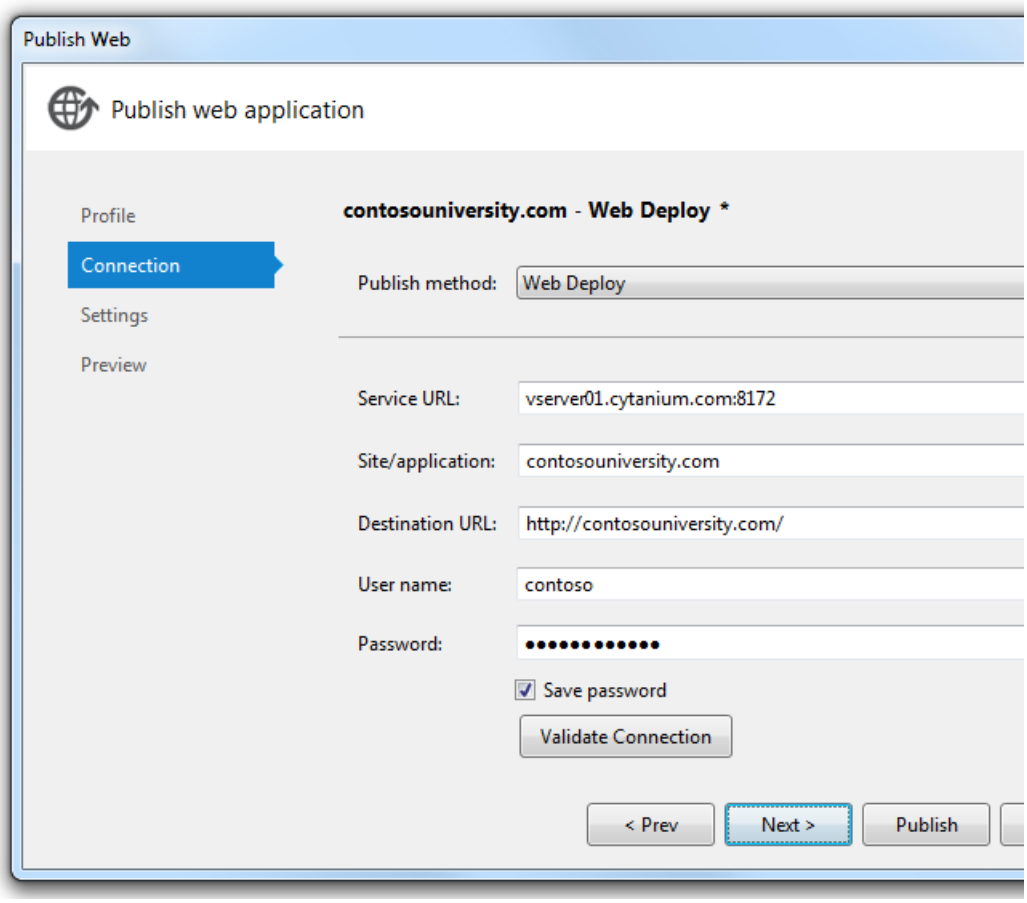
When you are prompted to open or save the file, save it.

In **Solution Explorer** in Visual Studio, right-click the ContosoUniversity project and select **Publish**. The **Publish Web** dialog box opens on the **Preview** tab with the **Test** profile selected because that is the last profile you used.

Select the **Profile** tab and then Click **Import**.



In the **Import Publish Settings** dialog box, select the *.publishsettings* file that you downloaded, and click **Open**. The wizard advances to the Connection tab with all of the fields filled in.

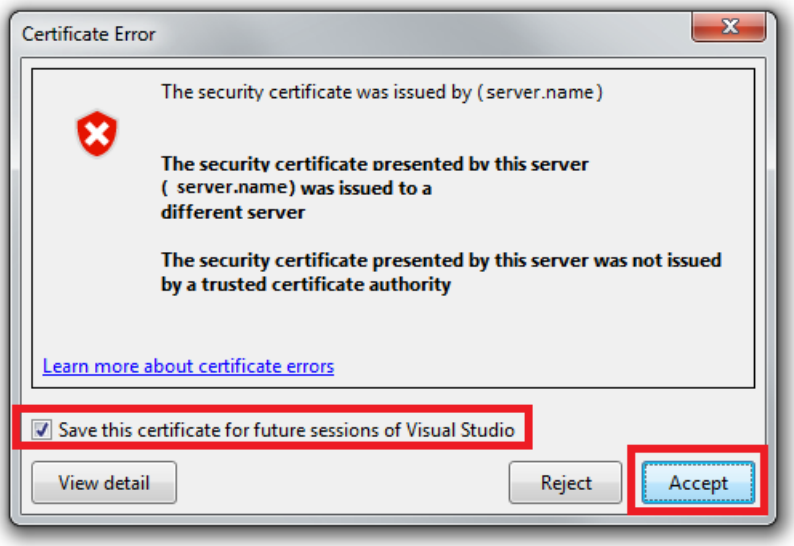


The .publishsettings file puts the planned permanent URL for the site in the Destination URL box, but if you haven't purchased that domain yet, replace the value with the temporary URL. For this example, the URL is *http://contosouniversity.com.vserver01.cytanium.com*. The only purpose of this box is to specify what URL the browser will open to automatically after successfully after deployment. If you leave it blank, the only

consequence is that the browser won't start automatically after deployment.

Click **Validate Connection** to verify that the settings are correct and you can connect to the server. As you saw earlier, a green check mark verifies that the connection is successful.

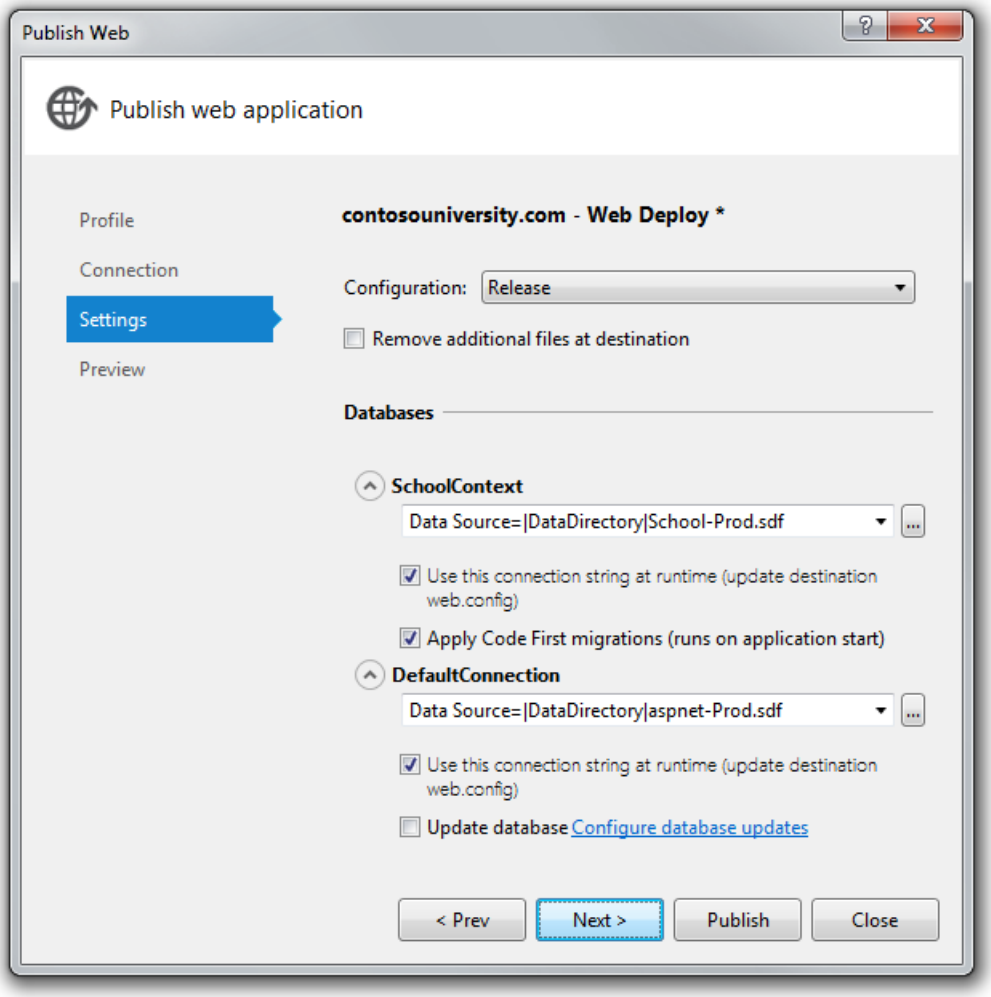
When you click Validate Connection, you might see a **Certificate Error** dialog box. If you do, verify that the server name is what you expect. If it is, select **Save this certificate for future sessions of Visual Studio** and click **Accept**. (This error means that the hosting provider has chosen to avoid the expense of purchasing an SSL certificate for the URL that you are deploying to. If you prefer to establish a secure connection by using a valid certificate, contact your hosting provider.)



Click **Next**.

In the **Databases** section of the **Settings** tab, enter the same values that you entered for the Test publish profile. You'll find the connection strings you need in the drop-down lists.

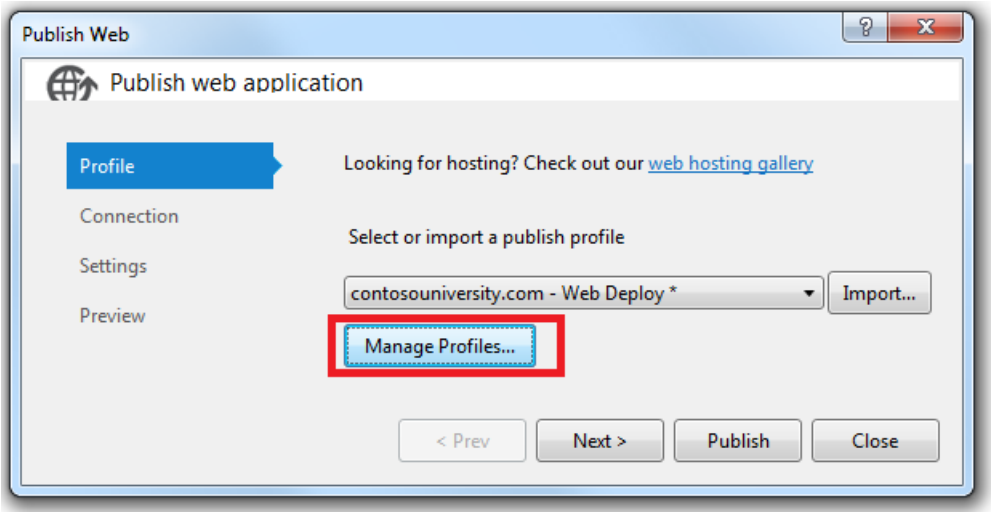
- In the connection string box for **SchoolContext,** select `Data Source=|DataDirectory|School-Prod.sdf`
- Under **SchoolContext**, select **Apply Code First Migrations**.
- In the connection string box for **DefaultConnection**, select `Data Source=|DataDirectory|aspnet-Prod.sdf`
- Under **DefaultConnection**, leave **Update database** cleared.



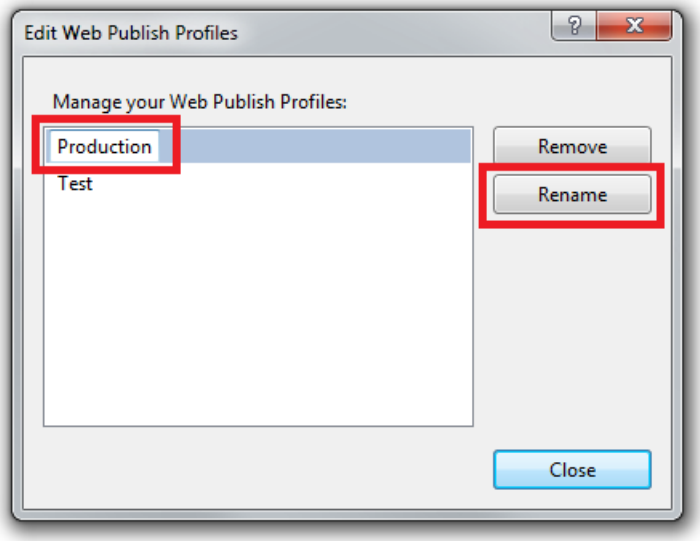
Click **Next**.

In the **Preview** tab, click **Start Preview** to see a list of the files that will be copied. You see the same list that you saw earlier when you deployed to IIS on the local computer.

Before you publish, change the name of the profile so that your Web.Production.config transformation file will be applied. Select the **Profile** tab and click **Manage Profiles**.

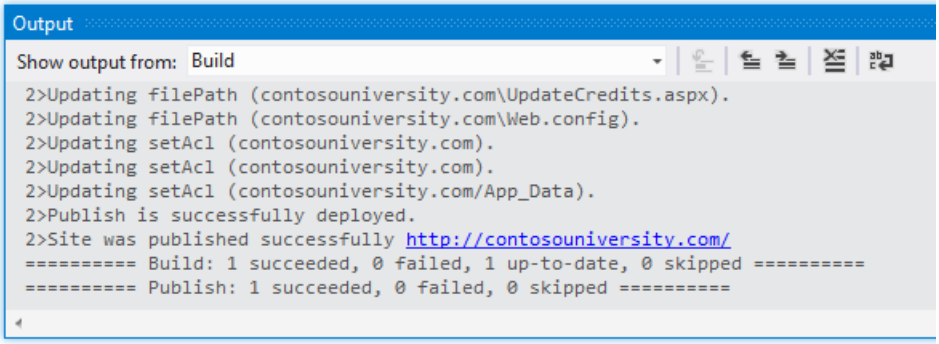


In the **Edit Web Publish Profiles** dialog box, select the production profile, click **Rename**, and change the profile name to Production. Then click **Close**.



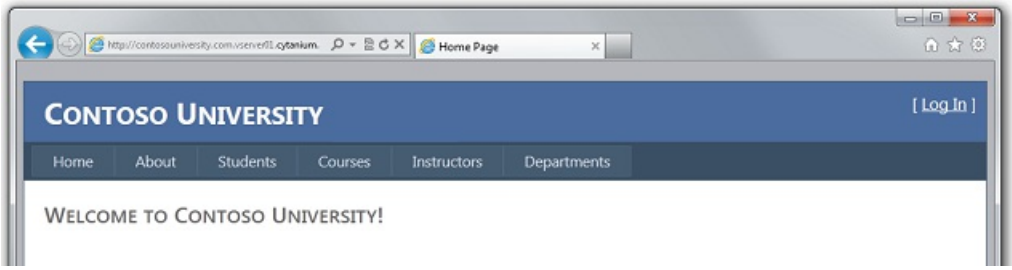
Click **Publish**.

The application is published to the hosting provider. The result shows in the **Output** window.



The browser automatically opens to the URL that you entered in **Destination URL** box on the **Connection** tab of the **Publish Web** wizard. You see the same home page as when you run the site in Visual Studio, except now there is no "(Test)" or "(Dev)" environment indicator in the title bar. This indicates that the environment indicator *Web.config* transformation worked correctly.

> **NOTE**
>
> If you still see "(Test)" in the heading, delete the *obj* folder from the ContosoUniversity project and redeploy. In pre-release versions of the software, the previously applied transformation file (Web.Test.config) might get applied again although you are using the Production profile.



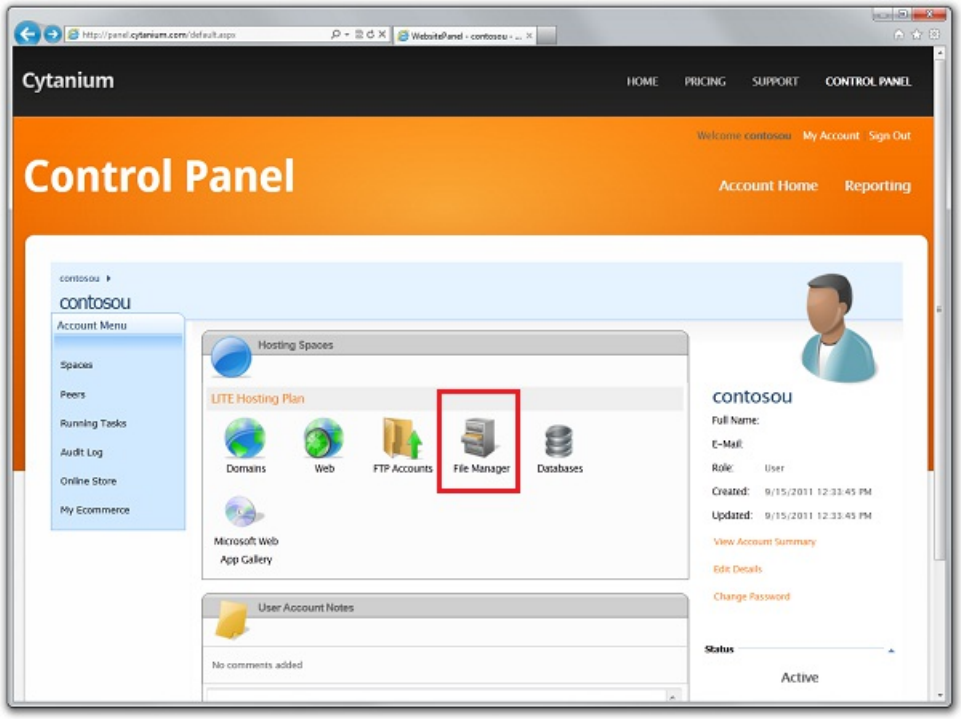
Before you run a page that causes database access, make sure that Elmah will be able to log any errors that occur.

Setting Folder Permissions for Elmah

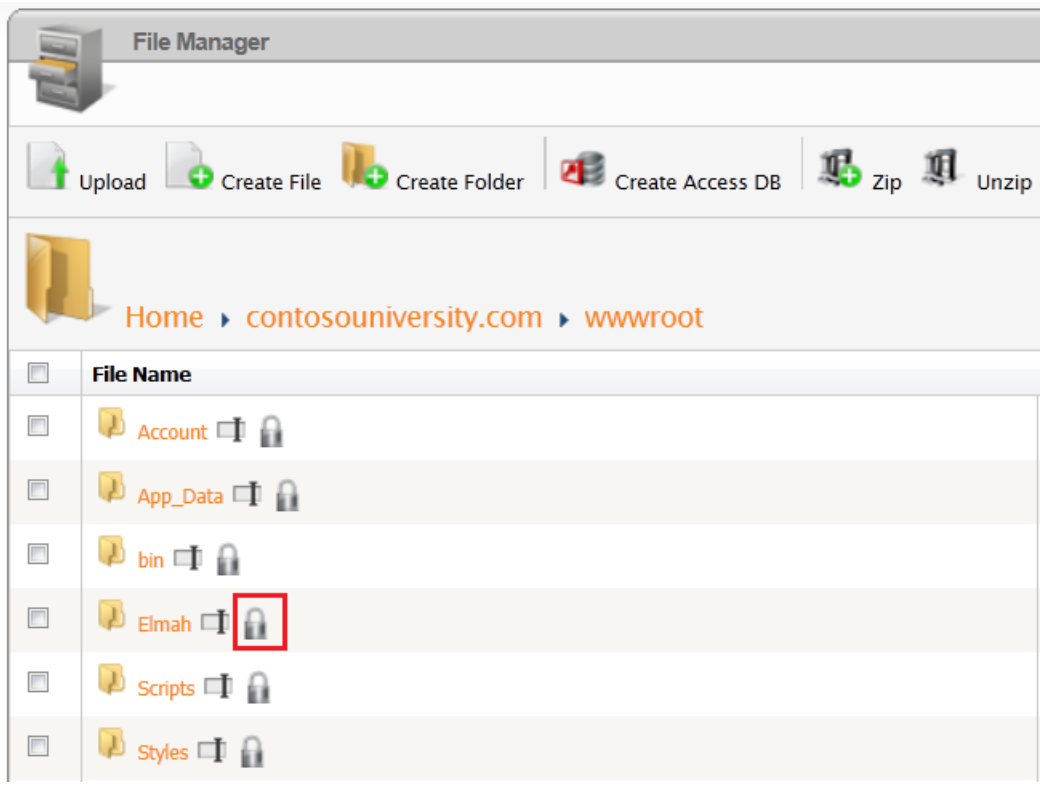
As you remember from the previous tutorial in this series, you must make sure that the application has Write permissions for the folder in your application where Elmah stores error log files. When you deployed to IIS

locally on your computer, you set those permissions manually. In this section, you'll see how to set permissions at Cytanium. (Some hosting providers may not enable you to do this; they may offer one or more predefined folders with Write permissions. In that case you would have to modify your application to use the specified folders.)

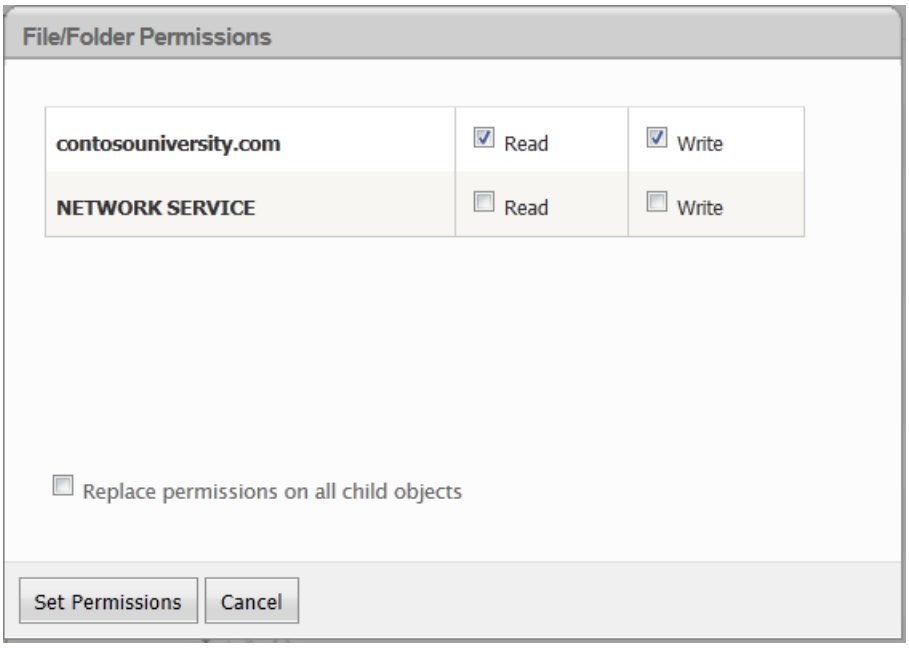
You can set folder permissions in the Cytanium control panel. Go to the control panel URL and select **File Manager**.



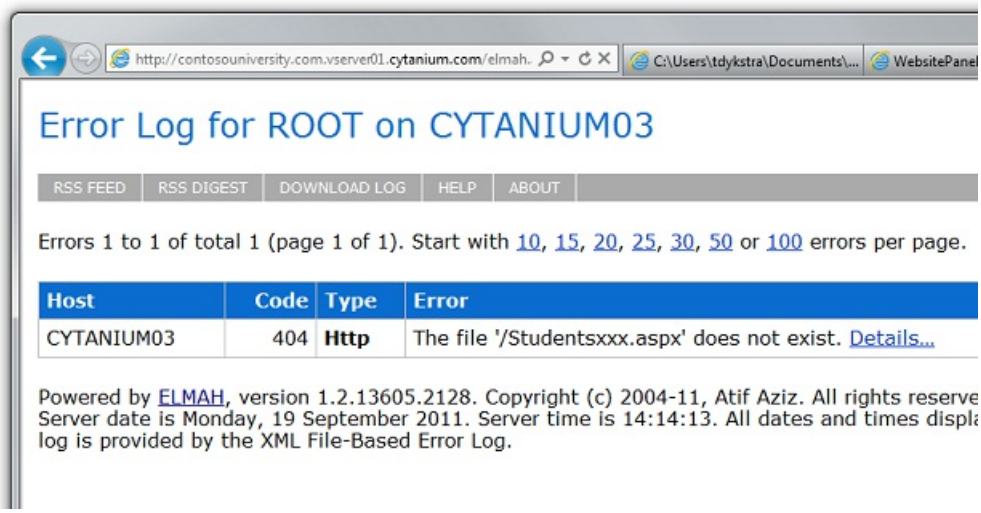
In the **File Manager** box, select **contosouniversity.com** and then **wwwroot** to see the application's root folder. Click the padlock icon next to **Elmah**.



In the **File/Folder Permissions** window, select the **Read** and **Write** checkboxes for **contosouniversity.com** and click **Set Permissions**.

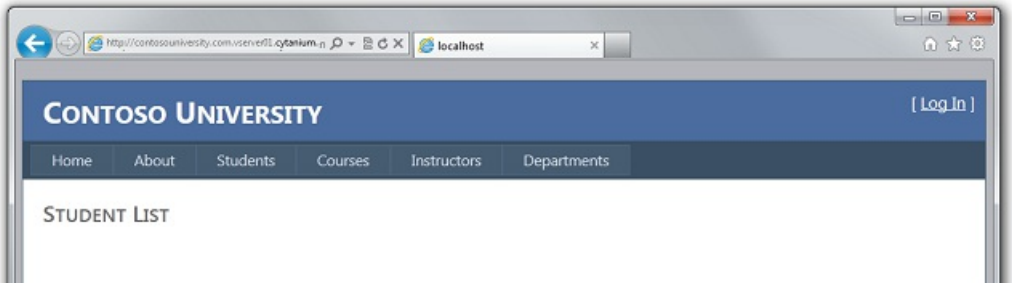


Make sure that Elmah has write access to the *Elmah* folder by causing an error and then displaying the Elmah error report. Request an invalid URL like *Studentsxxx.aspx*. As before, you see the *GenericErrorPage.aspx* page. Click the **Log Out** link, and then run *Elmah.axd*. You get the **Log In** page first, which validates that the *Web.config* transform successfully added Elmah authorization. After you log in, you see the report that shows the error you just caused.

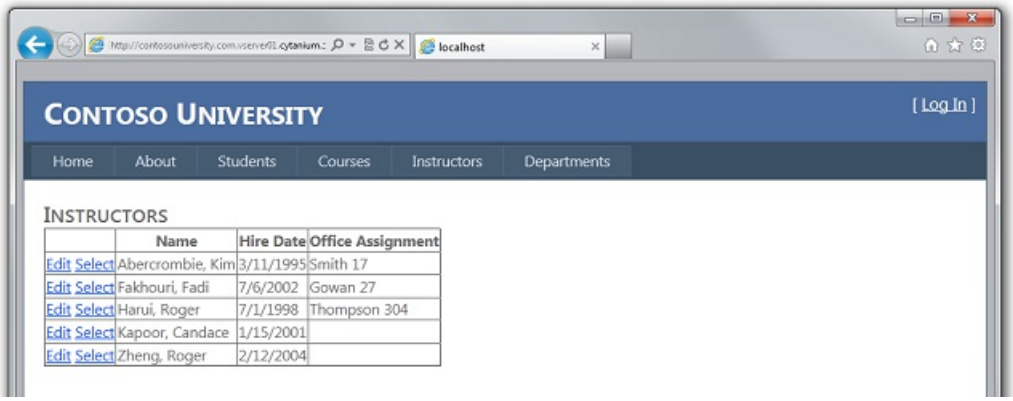


Testing in the Production Environment

Run the **Students** page. The application tries to access the School database for the first time, which triggers Code First Migrations to create the database. When the page displays after a moment's delay, it shows that there are no students.

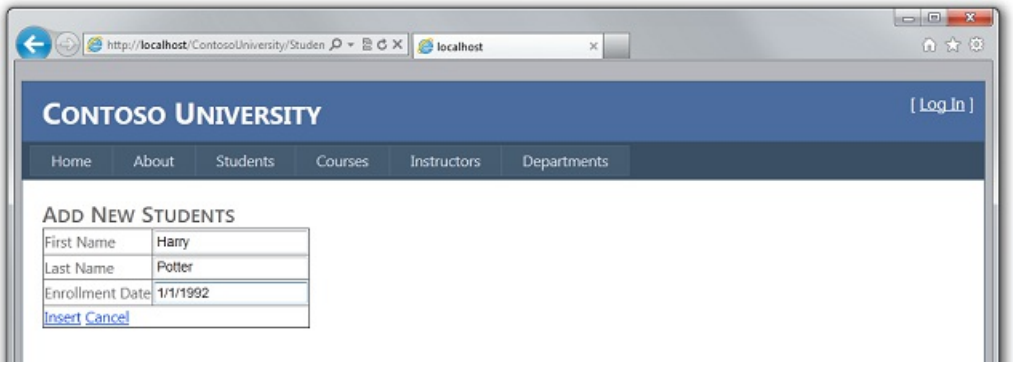


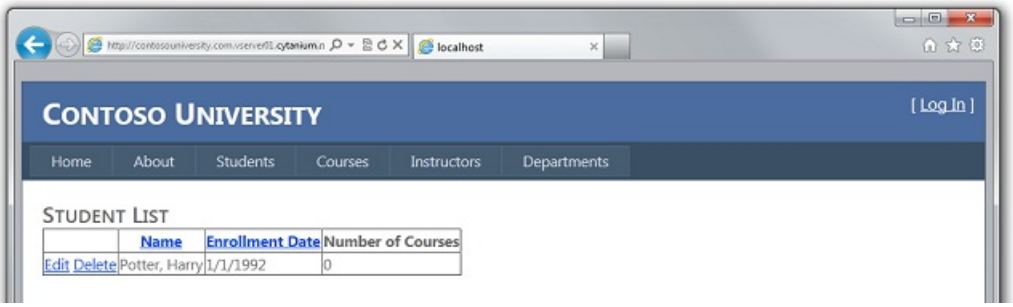
Run the **Instructors** page to verify that the Seed data successfully inserted instructor data in the database.



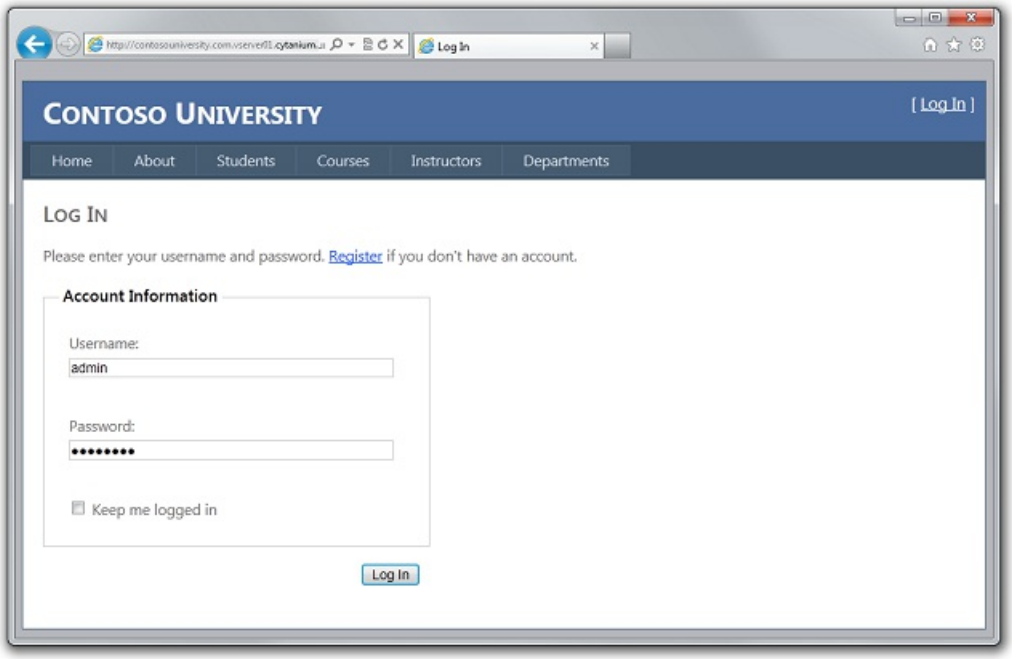
As you did in the test environment, you want to verify that database updates work in the production environment, but you typically do not want to enter test data into your production database. For this tutorial, you'll use the same method you did in test. But in a real application you might want to find a method that validates that database updates are successful without introducing test data into the production database. In some applications, it might be practical to add something and then delete it.

Add a student and then view the data you entered in the **Students** page to verify that you can update data in the database.

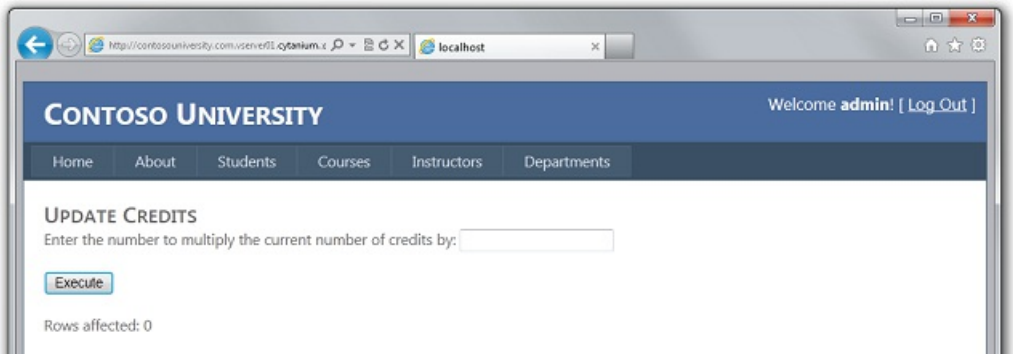




Validate that authorization rules are working correctly by selecting **Update Credits** from the **Courses** menu. The **Log In** page is displayed. Enter your administrator account credentials, click **Log In**, and the **Update Credits** page is displayed.



If login is successful, the **Update Credits** page is displayed. This indicates that the ASP.NET membership database (with the single administrator account) was successfully deployed.



You have now successfully deployed and tested your site and it is available publicly over the Internet.

Creating a More Reliable Test Environment

As explained in the Deploying to the Test Environment tutorial, the most reliable test environment would be a second account at the hosting provider that's just like the production account. This would be more expensive than using local IIS as your test environment, since you would have to sign up for a second hosting account. But if it prevents production site errors or outages, you might decide that it's worth the cost.

Most of the process for creating and deploying to a test account is similar to what you've already done to deploy to production:

- Create a *Web.config* transformation file.
- Create an account at the hosting provider.
- Create a new publish profile and deploy to the test account.

Preventing Public Access to the Test Site

An important consideration for the test account is that it will be live on the Internet, but you don't want the public to use it. To keep the site private you can use one or more of the following methods:

- Contact the hosting provider to set firewall rules that allow access to the testing site only from IP addresses that you use for testing.
- Disguise the URL so that it is not similar to the public site's URL.

- Use a *robots.txt* file to ensure that search engines will not crawl the test site and report links to it in search results.

The first of these methods is obviously the most secure, but the procedure for that is specific to each hosting provider and will not be covered in this tutorial. If you do arrange with your hosting provider to allow only your IP address to browse to the test account URL, you theoretically don't have to worry about search engines crawling it. But even in that case, deploying a *robots.txt* file is a good idea as a backup in case that firewall rule is ever accidentally turned off.

The *robots.txt* file goes in your project folder and should have the following text in it:

```
User-agent: *
Disallow: /
```

The `User-agent` line tells search engines that the rules in the file apply to all search engine web crawlers (robots), and the `Disallow` line specifies that no pages on the site should be crawled.

You probably do want search engines to catalog your production site, so you need to exclude this file from production deployment. To do that, see **Can I exclude specific files or folders from deployment?** in ASP.NET Web Application Project Deployment FAQ. Make sure that you specify the exclusion only for the Production publish profile.

Creating a second hosting account is an approach to working with a test environment that is not required but might be worth the added expense. In the following tutorials, you'll continue to use IIS as your test environment.

In the next tutorial, you'll update application code and deploy your change to the test and production environments.

PREVIOUS NEXT

Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio or Visual Web Developer: Deploying a Code-Only Update - 8 of 12

2/20/2020 • 5 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

This series of tutorials shows you how to deploy (publish) an ASP.NET web application project that includes a SQL Server Compact database by using Visual Studio 2012 RC or Visual Studio Express 2012 RC for Web. You can also use Visual Studio 2010 if you install the Web Publish Update. For an introduction to the series, see the first tutorial in the series.

For a tutorial that shows deployment features introduced after the RC release of Visual Studio 2012, shows how to deploy SQL Server editions other than SQL Server Compact, and shows how to deploy to Azure App Service Web Apps, see ASP.NET Web Deployment using Visual Studio.

Overview

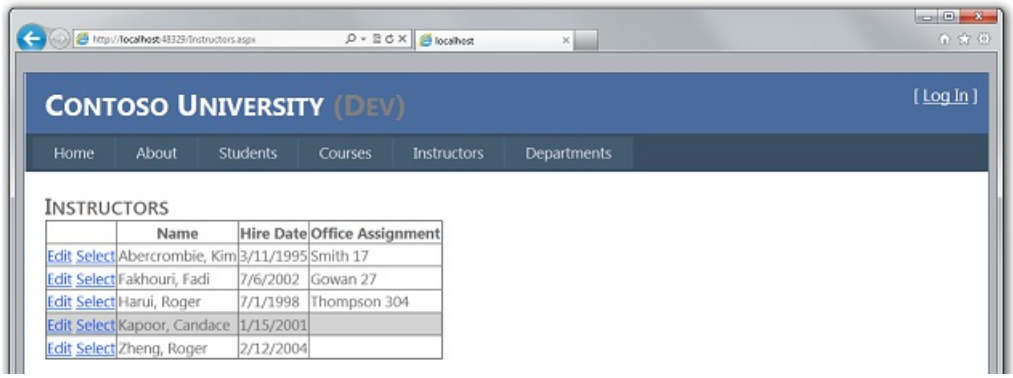
After the initial deployment, your work of maintaining and developing your web site continues, and before long you will want to deploy an update. This tutorial takes you through the process of deploying an update to your application code. This update does not involve a database change; you'll see what's different about deploying a database change in the next tutorial.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

Making a Code Change

As a simple example of an update to your application, you'll add to the **Instructors** page a list of courses taught by the selected instructor.

If you run the **Instructors** page, you'll notice that there are **Select** links in the grid, but they don't do anything other than make the row background turn gray.

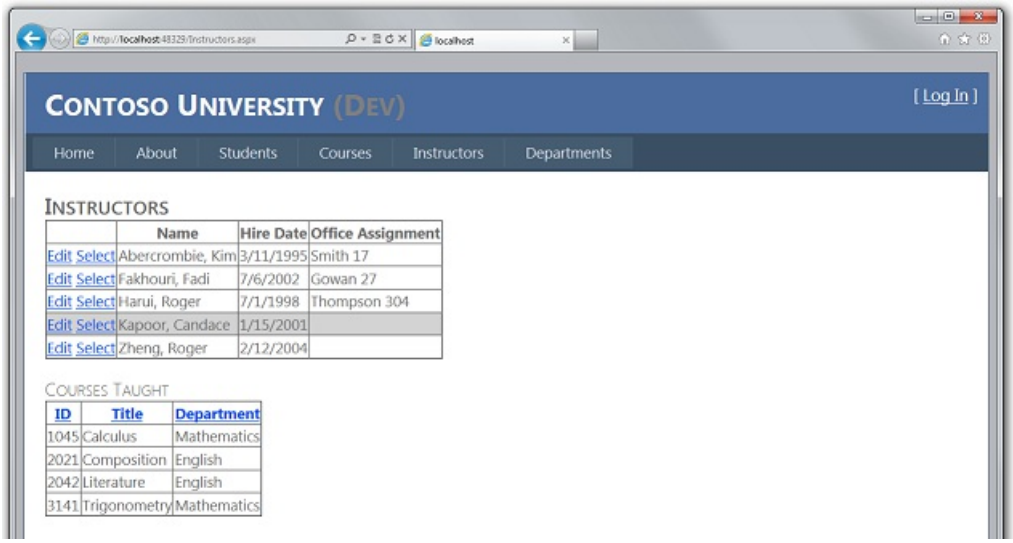


Now you'll add code that runs when the **Select** link is clicked and displays a list of courses taught by the selected instructor .

In *Instructors.aspx*, add the following markup immediately after the **ErrorMessageLabel** `Label` control:

```
<h3>Courses Taught</h3>
<asp:ObjectDataSource ID="CoursesObjectDataSource" runat="server" TypeName="ContosoUniversity.BLL.SchoolBL"
    DataObjectTypeName="ContosoUniversity.DAL.Course" SelectMethod="GetCoursesByInstructor">
    <SelectParameters>
        <asp:ControlParameter ControlID="InstructorsGridView" Name="PersonID"
PropertyName="SelectedDataKey.Value"
            Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
<asp:GridView ID="CoursesGridView" runat="server" DataSourceID="CoursesObjectDataSource"
    AllowSorting="True" AutoGenerateColumns="False" SelectedRowStyle-BackColor="LightGray"
    DataKeyNames="CourseID">
    <EmptyDataTemplate>
        <p>No courses found.</p>
    </EmptyDataTemplate>
    <Columns>
        <asp:BoundField DataField="CourseID" HeaderText="ID" ReadOnly="True" SortExpression="CourseID" />
        <asp:BoundField DataField="Title" HeaderText="Title" SortExpression="Title" />
        <asp:TemplateField HeaderText="Department" SortExpression="DepartmentID">
            <ItemTemplate>
                <asp:Label ID="GridViewDepartmentLabel" runat="server" Text='<%# Eval("Department.Name") %>'>
</asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
```

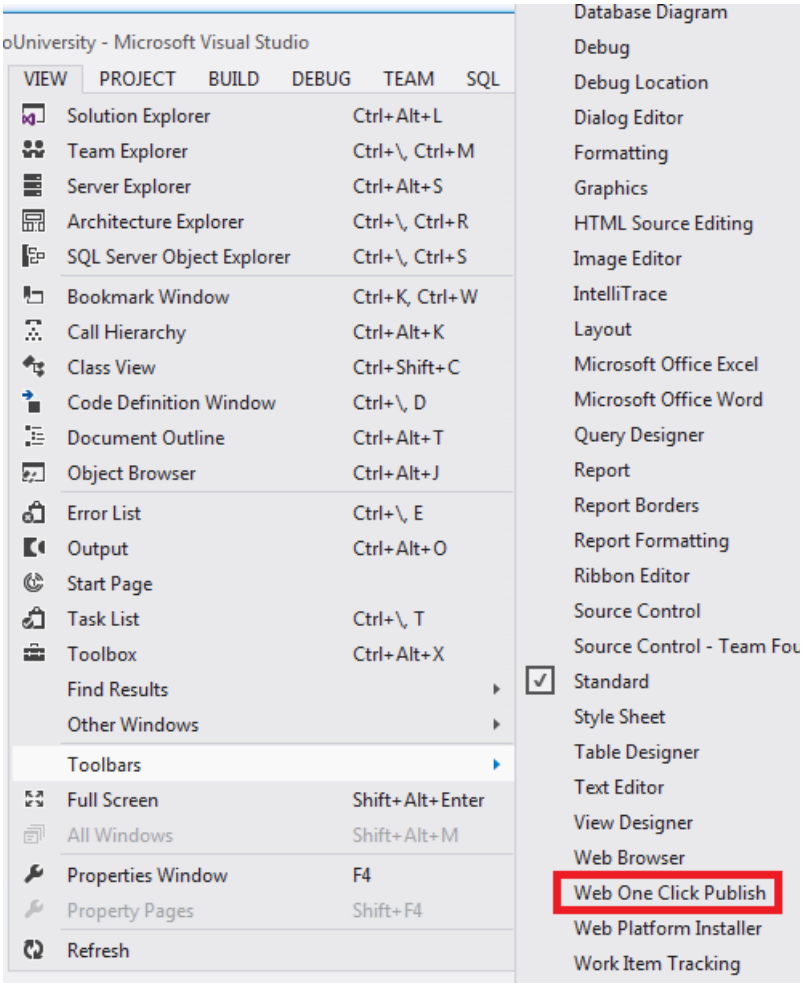
Run the page and select an instructor. You see a list of courses taught by that instructor.



Deploying the Code Update to the Test Environment

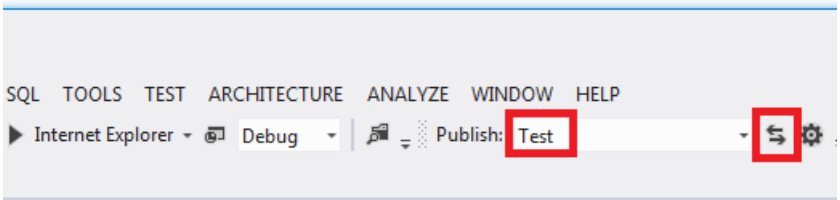
Deploying to the test environment is a simple matter of running one-click publish again. To make this process quicker, you can use the **Web One Click Publish** toolbar.

In the **View** menu, choose **Toolbars** and then select **Web One Click Publish**.

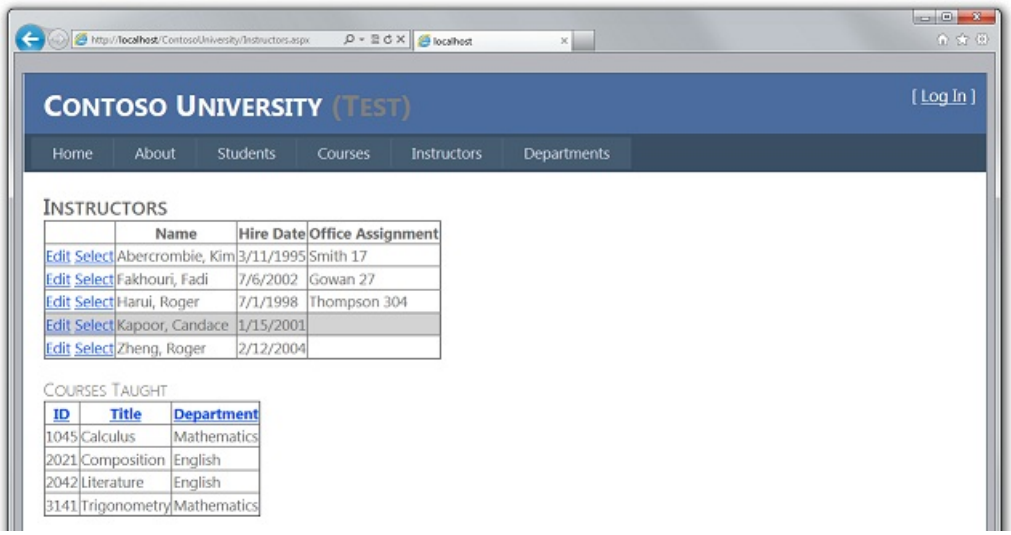


In **Solution Explorer**, select the ContosoUniversity project.

the **Web One Click Publish** toolbar, choose the **Test** publish profile and then click **Publish Web** (the icon with arrows pointing left and right).



Visual Studio deploys the updated application, and the browser automatically opens to the home page. Run the Instructors page and select an instructor to verify that the update was successfully deployed.

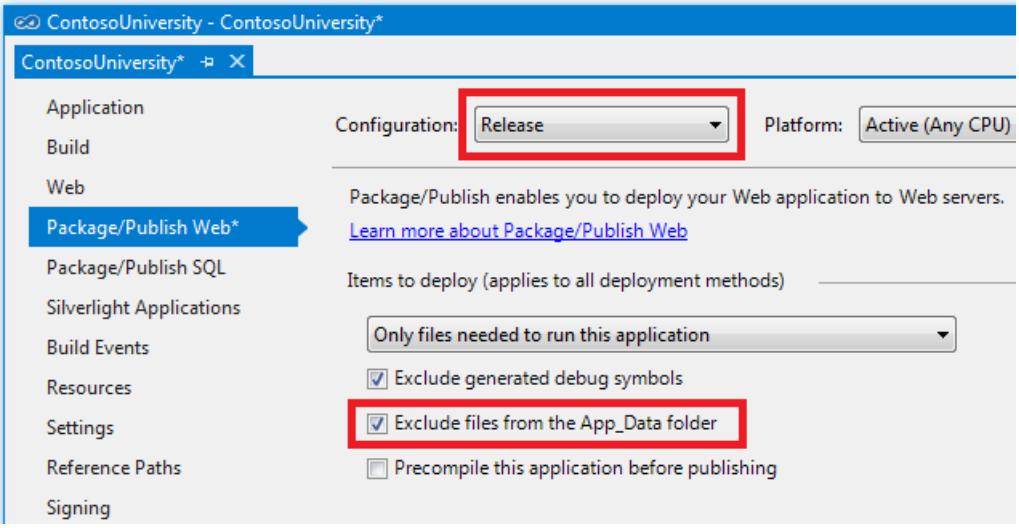


You would normally also do regression testing (that is, test the rest of the site to make sure that the new change didn't break any existing functionality). But for this tutorial you'll skip that step and proceed to deploy the update to production.

Preventing Redeployment of the Initial Database State to Production

In a real application, users interact with your production site after your initial deployment, and the databases are populated with live data. Therefore, you don't want to redeploy the membership database in its initial state, which would wipe out all of the live data. Since SQL Server Compact databases are files in the *App_Data* folder, you have to prevent this by changing deployment settings so that files in the *App_Data* folder aren't deployed.

Open the **Project Properties** window for the ContosoUniversity project, and select the **Package/Publish Web** tab. Make sure that the **Configuration** drop-down box has either **Active (Release)** or **Release** selected, select **Exclude files from the App_Data folder**.



In case you decide to deploy a debug build in the future, it's a good idea to make the same change for the Debug build configuration: change **Configuration** to **Debug** and then select **Exclude files from the App_Data folder**.

Save and close the **Package/Publish Web** tab.

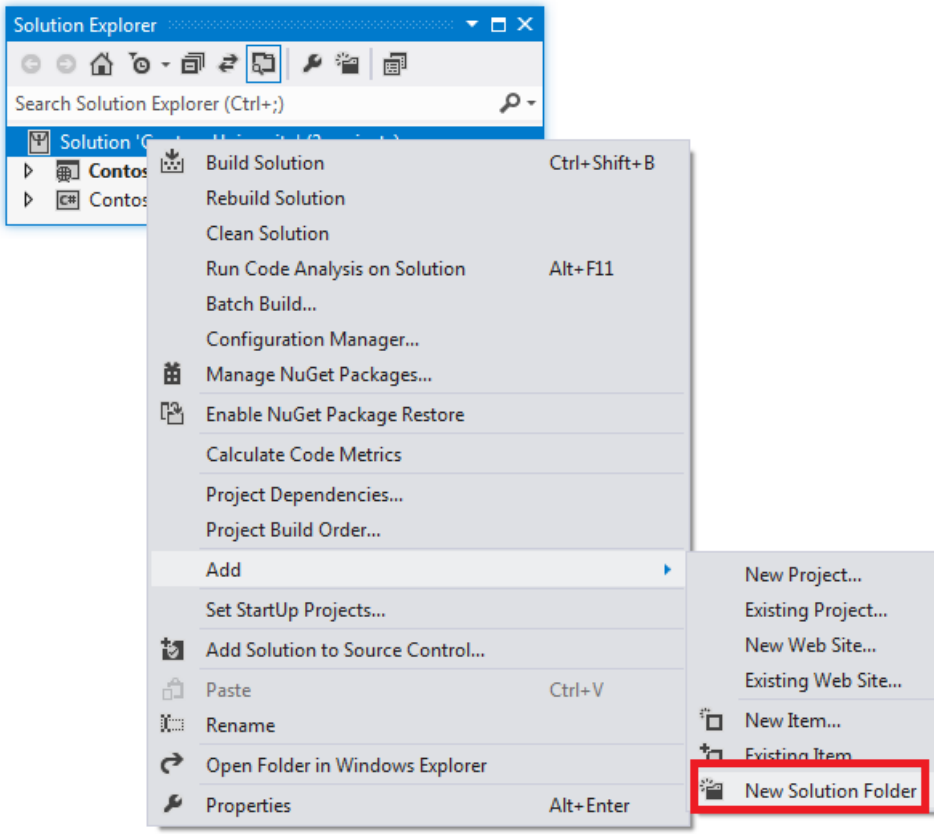
> **NOTE**

> **IMPORTANT**
>
> Make sure that you don't have **Remove additional files at destination** selected in your publish profiles. If you select that option, the deployment process will delete the databases that you have in App_Data in the deployed site, and it will delete the App_Data folder itself.

Preventing User Access to the Production Site During Update

The change you're deploying now is a simple change to a single page. But sometimes you deploy larger changes, and in that case the site can behave strangely if a user requests a page before deployment is finished. To prevent this, you can use an *app_offline.htm* file. When you put a file named *app_offline.htm* in the root folder of your application, IIS automatically displays that file instead of running your application. So to prevent access during deployment, you put *app_offline.htm* in the root folder, run the deployment process, and then remove *app_offline.htm*.

In **Solution Explorer**, right-click the solution (not one of the projects) and select **New Solution Folder**.



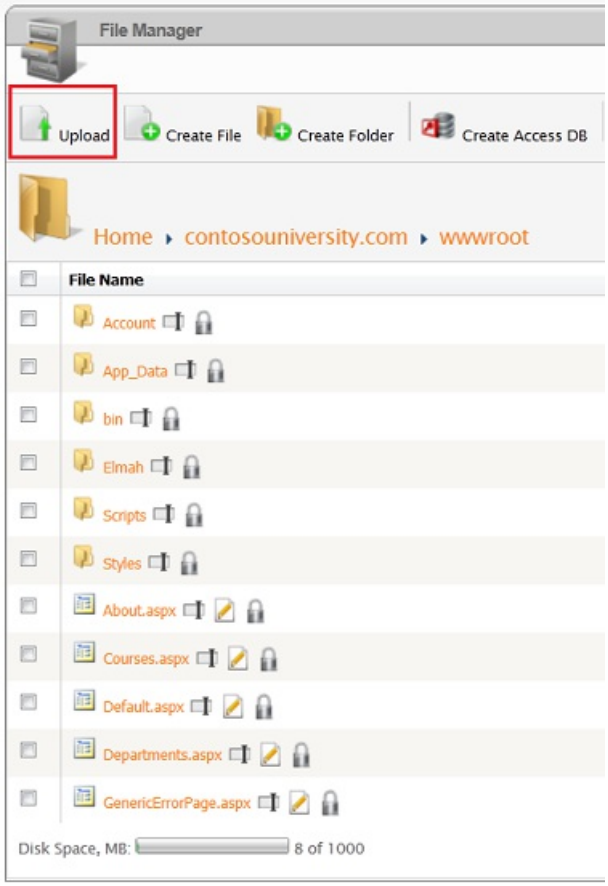
Name the folder *SolutionFiles*.

In the new folder create an HTML page named *app_offline.htm*. Replace the existing contents with the following markup:

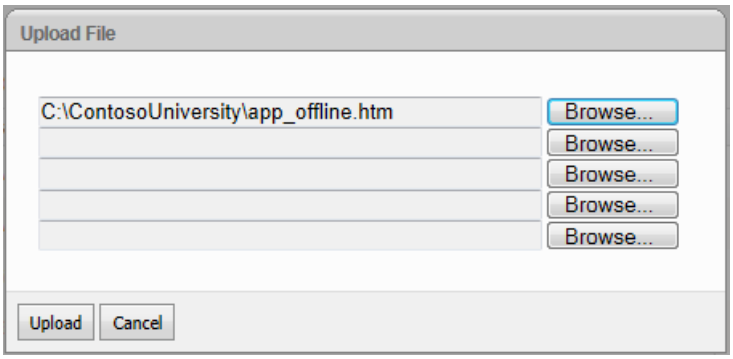
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Contoso University - Under Construction</title>
</head>
<body>
  <h1>Contoso University</h1>
  <h2>Under Construction</h2>
  <p>The Contoso University site is temporarily unavailable while we upgrade it. Please try again later.</p>
</body>
</html>
```

You can copy the *app_offline.htm* file to the site by using an FTP connection or the **File Manager** utility in the hosting provider's control panel. For this tutorial, you'll use the **File Manager**.

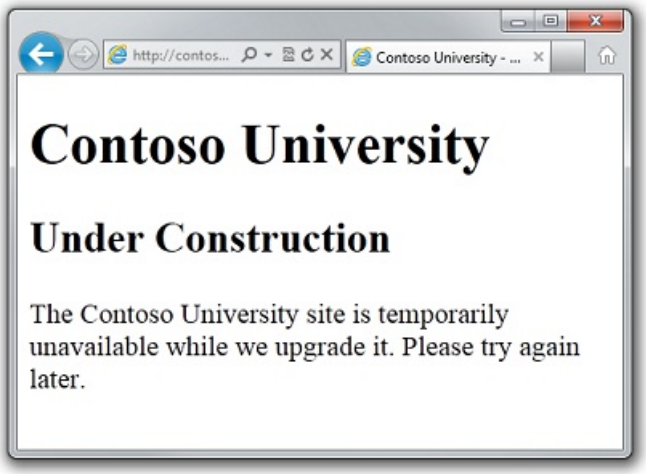
Open the control panel and select **File Manager** as you did in the Deploying to the Production Environment tutorial. Select **contosouniversity.com** and then **wwwroot** to get to your application's root folder, and then click **Upload**.



In the **Upload File** dialog box, select the *app_offline.htm* file and then click **Upload**.



Browse to your site's URL. You see that the *app_offline.htm* page is now displayed instead of your home page.



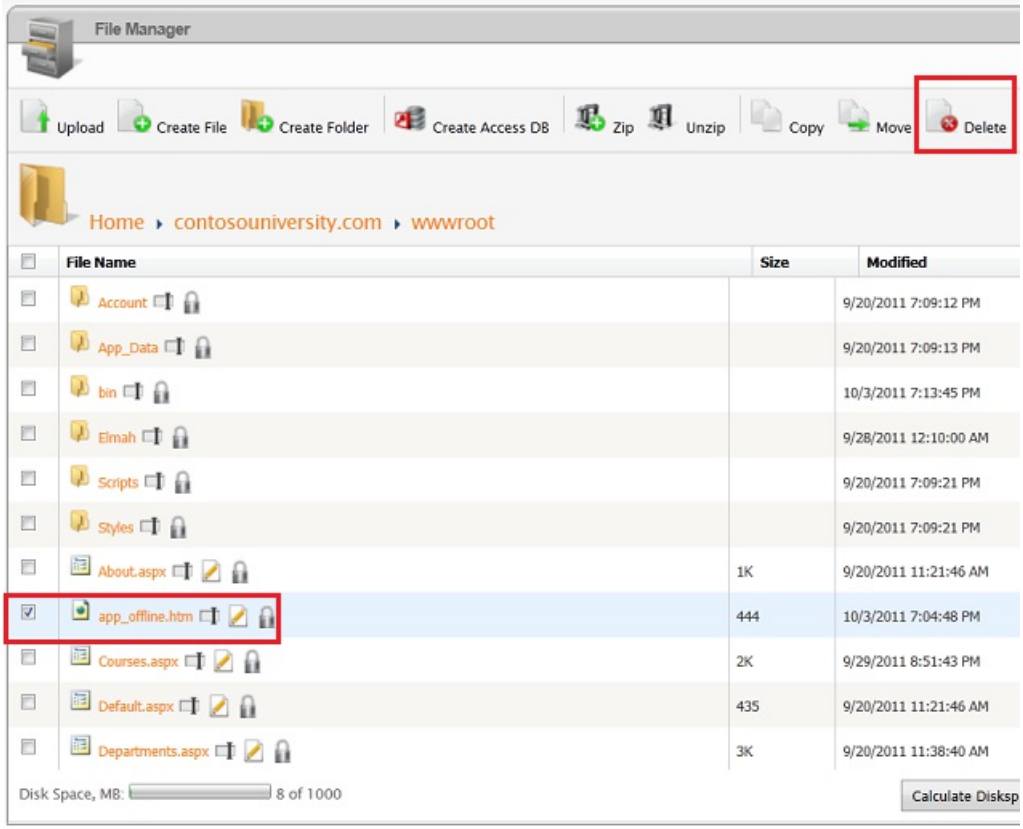
You are now ready to deploy to production.

Deploying the Code Update to the Production Environment

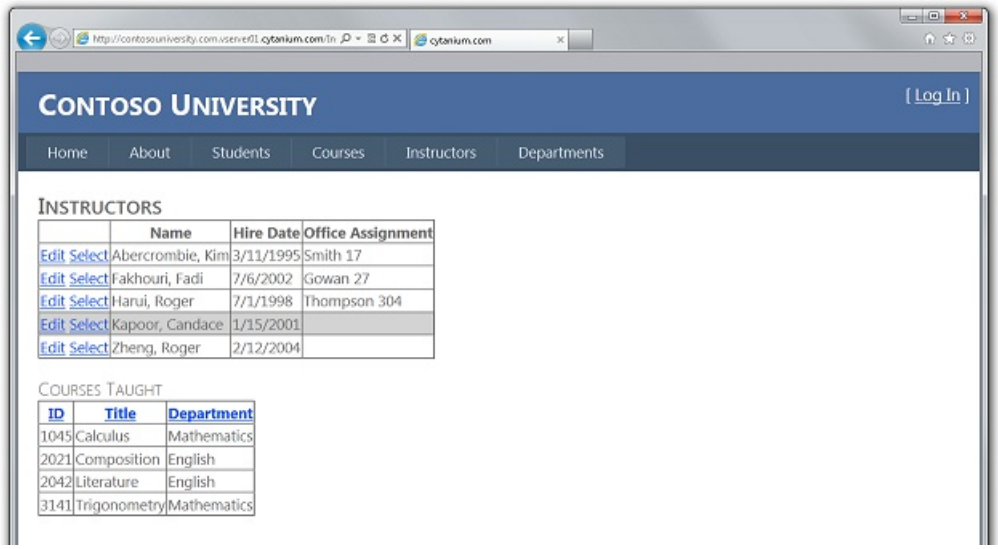
In the **Web One Click Publish** toolbar, choose the **Production** publish profile and then click **Publish Web**.

Visual Studio deploys the updated application and opens the browser to the site's home page. The *app_offline.htm* file is displayed. Before you can test to verify successful deployment, you must remove the *app_offline.htm* file.

Return to the **File Manager** application in the control panel. Select **contosouniversity.com** and **wwwroot**, select **app_offline.htm**, and then click **Delete**.



In the browser, open the Instructors page in the public site, and select an instructor to verify that the update was successfully deployed.



You've now deployed an application update that did not involve a database change. The next tutorial shows you how to deploy a database change.

Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio or Visual Web Developer: Deploying a Database Update - 9 of 12

2/20/2020 • 4 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This series of tutorials shows you how to deploy (publish) an ASP.NET web application project that includes a SQL Server Compact database by using Visual Studio 2012 RC or Visual Studio Express 2012 RC for Web. You can also use Visual Studio 2010 if you install the Web Publish Update. For an introduction to the series, see the first tutorial in the series.
>
> For a tutorial that shows deployment features introduced after the RC release of Visual Studio 2012, shows how to deploy SQL Server editions other than SQL Server Compact, and shows how to deploy to Azure App Service Web Apps, see ASP.NET Web Deployment using Visual Studio.

Overview

In this tutorial, you make a database change and related code changes, test the changes in Visual Studio, then deploy the update to both the test and production environments.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

Adding a New Column to a Table

In this section, you add a birth date column to the `Person` base class for the `Student` and `Instructor` entities. Then you update the page that displays instructor data so that it displays the new column.

In the *ContosoUniversity.DAL* project, open *Person.cs* and add the following property at the end of the `Person` class (there should be two closing curly braces following it):

```
[DisplayFormat(DataFormatString = "{0:d}", ApplyFormatInEditMode = true)]
[Display(Name = "Birth Date")]
public DateTime? BirthDate { get; set; }
```

Next, update the Seed method so that it provides a value for the new column. Open *Migrations\Configuration.cs* and replace the code block that begins `var instructors = new List<Instructor>` with the following code block which includes birth date information:

```
var instructors = new List<Instructor>
{
    new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie", HireDate = DateTime.Parse("1995-03-
11"), BirthDate = DateTime.Parse("1918-08-12"), OfficeAssignment = new OfficeAssignment { Location = "Smith
17" } },
    new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",    HireDate = DateTime.Parse("2002-07-
06"), BirthDate = DateTime.Parse("1960-03-15"), OfficeAssignment = new OfficeAssignment { Location = "Gowan
27" } },
    new Instructor { FirstMidName = "Roger",   LastName = "Harui",       HireDate = DateTime.Parse("1998-07-
01"), BirthDate = DateTime.Parse("1970-01-11"), OfficeAssignment = new OfficeAssignment { Location =
"Thompson 304" } },
    new Instructor { FirstMidName = "Candace", LastName = "Kapoor",      HireDate = DateTime.Parse("2001-01-
15"), BirthDate = DateTime.Parse("1975-04-11") },
    new Instructor { FirstMidName = "Roger",   LastName = "Zheng",       HireDate = DateTime.Parse("2004-02-
12"), BirthDate = DateTime.Parse("1957-10-12") }
};
```

In the ContosoUniversity project, open *Instructors.aspx* and add a new template field to display the birth date. Add it between the ones for hire date and office assignment:

```
<asp:TemplateField HeaderText="Birth Date" SortExpression="BirthDate">
    <ItemTemplate>
        <asp:Label ID="InstructorBirthDateLabel" runat="server" Text='<%# Eval("BirthDate", "{0:d}") %>'>
</asp:Label>
    </ItemTemplate>
    <EditItemTemplate>
        <asp:TextBox ID="InstructorBirthDateTextBox" runat="server" Text='<%# Bind("BirthDate", "{0:d}") %>'
            Width="7em"></asp:TextBox>
    </EditItemTemplate>
</asp:TemplateField>
```

(If code indentation gets out of sync, you can press CTRL-K and then CTRL-D to automatically reformat the file.)

Build the solution, and then open the **Package Manager Console** window. Make sure that ContosoUniversity.DAL is still selected as the **Default project**.

In the **Package Manager Console** window, select **ContosoUniversity.DAL** as the **Default project**, and then enter the following command:

```
add-migration AddBirthDate
```

When this command finishes, Visual Studio opens the class file that defines the new `DbMigration` class, and in the `Up` method you can see the code that creates the new column.

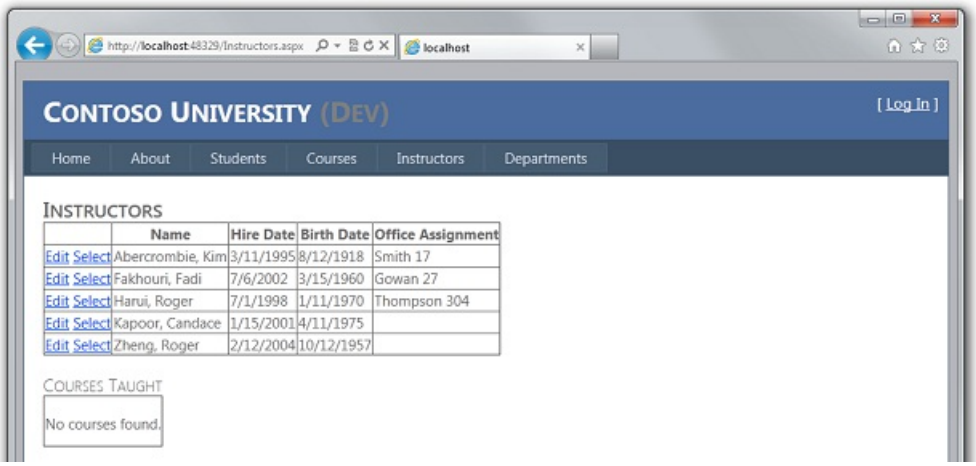
```
public partial class AddBirthDate : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.Person", "BirthDate", c => c.DateTime());
    }

    public override void Down()
    {
        DropColumn("dbo.Person", "BirthDate");
    }
}
```

Build the solution, and then enter the following command in the **Package Manager Console** window (make sure the ContosoUniversity.DAL project is still selected):

```
update-database
```

When the command finishes, run the application and select the Instructors page. When the page loads, you see that it has the new birth date field.

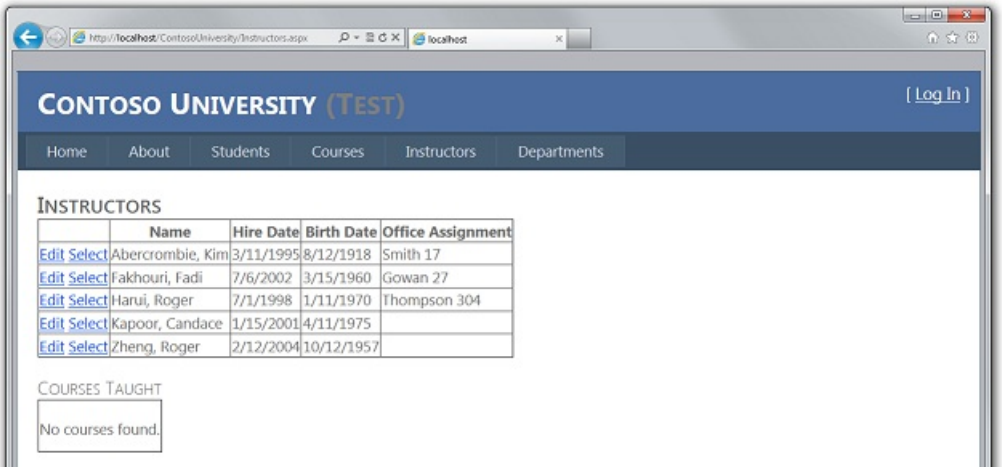


Deploying the Database Update to the Test Environment

In **Solution Explorer** select the ContosoUniversity project.

In the **Web One Click Publish** toolbar, select the **Test** publish profile, and then click **Publish Web**. (If the toolbar is disabled, select the ContosoUniversity project in **Solution Explorer**.)

Visual Studio deploys the updated application, and the browser opens to the home page. Run the Instructors page to verify that the update was successfully deployed. When the application tries to access the database for this page, Code First updates the database schema and runs the `Seed` method. When the page displays, you see the expected **Birth Date** column with dates in it.



Deploying the Database Update to the Production Environment

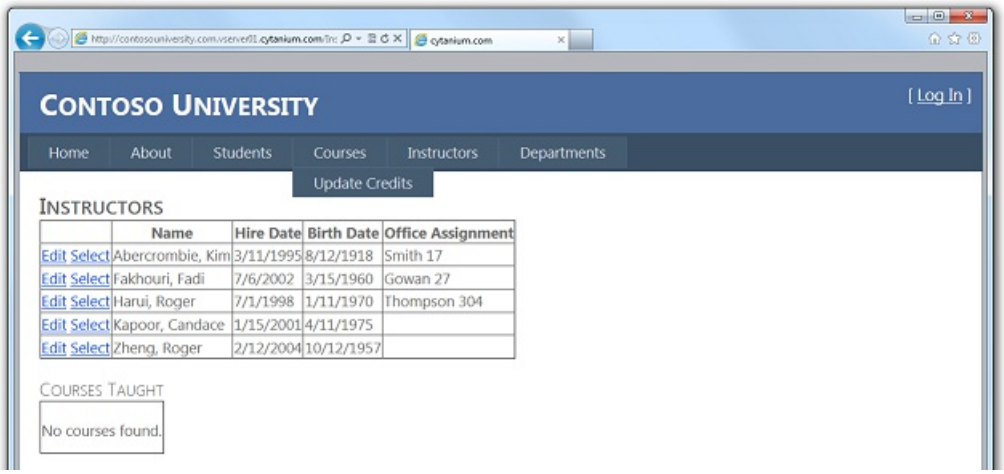
You can now deploy to production. The only difference is that you'll use *app_offline.htm* to prevent users from accessing the site and thus updating the database while you're deploying changes. For production deployment perform the following steps:

- Upload the *app_offline.htm* file to the production site.
- In Visual Studio, choose the Production profile in the **Web One Click Publish** toolbar and click **Publish Web**.
- Delete the *app_offline.htm* file from the production site.

NOTE

While your application is in use in the production environment you should be implementing a backup plan. That is, you should be periodically copying the *School-Prod.sdf* and *aspnet-Prod.sdf* files from the production site to a secure storage location, and you should be keeping several generations of such backups. When you update the database, you should make a backup copy from immediately before the change. Then, if you make a mistake and don't discover it until after you have deployed it to production, you will still be able to recover the database to the state it was in before it became corrupted.

When Visual Studio opens the home page URL in the browser, the *app_offline.htm* page is displayed. After you delete the *app_offline.htm* file, you can browse to your home page again to verify that the update was successfully deployed.



You've now deployed an application update that included a database change to both test and production. The next tutorial shows you how to migrate your database from SQL Server Compact to SQL Server Express and SQL Server.

PREVIOUS NEXT

Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio or Visual Web Developer: Migrating to SQL Server - 10 of 12

2/20/2020 • 22 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This series of tutorials shows you how to deploy (publish) an ASP.NET web application project that includes a SQL Server Compact database by using Visual Studio 2012 RC or Visual Studio Express 2012 RC for Web. You can also use Visual Studio 2010 if you install the Web Publish Update. For an introduction to the series, see the first tutorial in the series.
>
> For a tutorial that shows deployment features introduced after the RC release of Visual Studio 2012, shows how to deploy SQL Server editions other than SQL Server Compact, and shows how to deploy to Azure App Service Web Apps, see ASP.NET Web Deployment using Visual Studio.

Overview

This tutorial shows you how to migrate from SQL Server Compact to SQL Server. One reason you might want to do that is to take advantage of SQL Server features that SQL Server Compact does not support, such as stored procedures, triggers, views, or replication. For more information about the differences between SQL Server Compact and SQL Server, see the Deploying SQL Server Compact tutorial.

SQL Server Express versus full SQL Server for Development

Once you've decided to upgrade to SQL Server, you might want to use SQL Server or SQL Server Express in your development and test environments. In addition to the differences in tool support and in database engine features, there are differences in provider implementations between SQL Server Compact and other versions of SQL Server. These differences can cause the same code to generate different results. Therefore, if you decide to keep SQL Server Compact as your development database, you should thoroughly test your site in SQL Server or SQL Server Express in a test environment before each deployment to production.

Unlike SQL Server Compact, SQL Server Express is essentially the same database engine and uses the same .NET provider as full SQL Server. When you test with SQL Server Express, you can be confident of getting the same results as you will with SQL Server. You can use most of the same database tools with SQL Server Express that you can use with SQL Server (a notable exception being SQL Server Profiler), and it supports other features of SQL Server like stored procedures, views, triggers, and replication. (You typically have to use full SQL Server in a production website, however. SQL Server Express can run in a shared hosting environment, but it was not designed for that, and many hosting providers do not support it.)

If you are using Visual Studio 2012, you typically choose SQL Server Express LocalDB for your development environment because that is what is installed by default with Visual Studio. However, LocalDB does not work in IIS, so for your test environment you have to use either SQL Server or SQL Server Express.

Combining Databases versus Keeping Them Separate

The Contoso University application has two SQL Server Compact databases: the membership database (*aspnet.sdf*) and the application database (*School.sdf*). When you migrate, you can migrate these databases to two separate databases or to a single database. You might want to combine them in order to facilitate database

joins between your application database and your membership database. Your hosting plan might also provide a reason to combine them. For example, the hosting provider might charge more for multiple databases or might not even allow more than one database. That's the case with the Cytanium Lite hosting account that's used for this tutorial, which allows only a single SQL Server database.

In this tutorial, you'll migrate your two databases this way:

- Migrate to two LocalDB databases in the development environment.
- Migrate to two SQL Server Express databases in the test environment.
- Migrate to one combined full SQL Server database in the production environment.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

Installing SQL Server Express

SQL Server Express is automatically installed by default with Visual Studio 2010, but by default it is not installed with Visual Studio 2012. To install SQL Server 2012 Express, click the following link

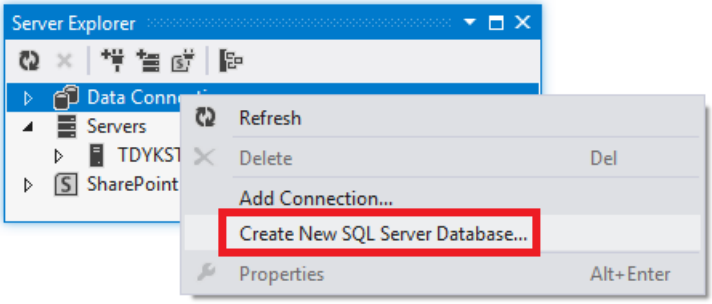
- SQL Server Express 2012

Choose *ENU/x64/SQLEXPR_x64_ENU.exe* or *ENU/x86/SQLEXPR_x86_ENU.exe*, and in the installation wizard accept the default settings. For more information about installation options, see Install SQL Server 2012 from the Installation Wizard (Setup).

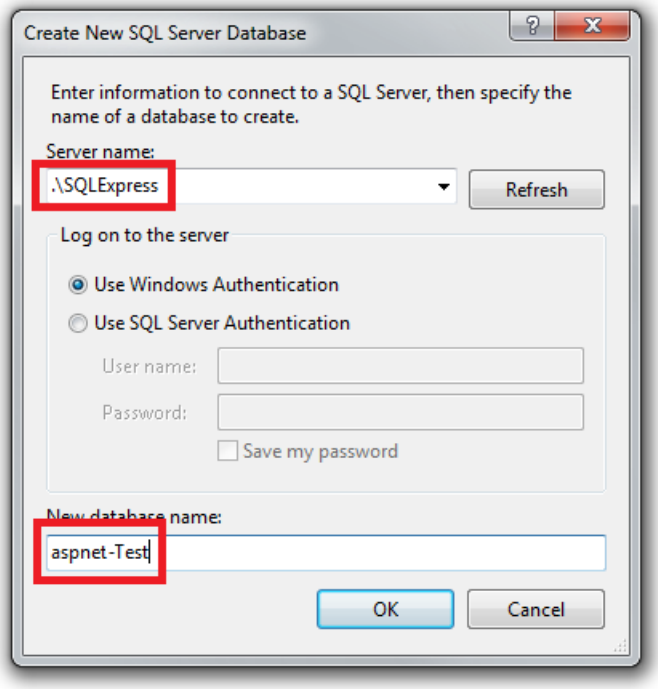
Creating SQL Server Express Databases for the Test Environment

The next step is to create the ASP.NET membership and School databases.

From the **View** menu select **Server Explorer** (**Database Explorer** in Visual Web Developer), and then right-click **Data Connections** and select **Create New SQL Server Database**.



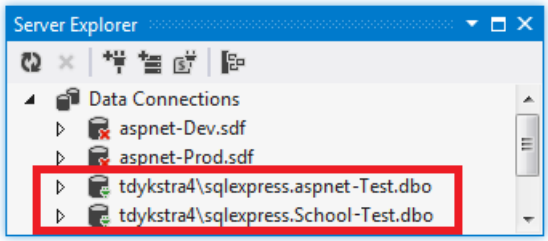
In the **Create New SQL Server Database** dialog box, enter ".\SQLExpress" in the **Server name** box and "aspnet-Test" in the **New database name** box, then click **OK**.



Follow the same procedure to create a new SQL Server Express School database named "School-Test".

(You're appending "Test" to these database names because later you'll create an additional instance of each database for the development environment, and you need to be able to differentiate the two sets of databases.)

Server Explorer now shows the two new databases.



Creating a Grant Script for the New Databases

When the application runs in IIS on your development computer, the application accesses the database by using the default application pool's credentials. However, by default, the application pool identity does not have permission to open the databases. So you have to run a script to grant that permission. In this section you create the script that you'll run later to make sure that the application can open the databases when it runs in IIS.

In the solution's *SolutionFiles* folder that you created in the Deploying to the Production Environment tutorial, create a new SQL file named *Grant.sql*. Copy the following SQL commands into the file, and then save and close the file:

```
IF NOT EXISTS (SELECT name FROM sys.server_principals WHERE name = 'IIS APPPOOL\DefaultAppPool')
BEGIN
    CREATE LOGIN [IIS APPPOOL\DefaultAppPool]
      FROM WINDOWS WITH DEFAULT_DATABASE=[master],
      DEFAULT_LANGUAGE=[us_english]
END
GO
CREATE USER [ContosoUniversityUser]
  FOR LOGIN [IIS APPPOOL\DefaultAppPool]
GO
EXEC sp_addrolemember 'db_owner', 'ContosoUniversityUser'
GO
```

NOTE

This script is designed to work with SQL Server 2008 and with the IIS settings in Windows 7 as they are specified in this tutorial. If you're using a different version of SQL Server or of Windows, or if you set up IIS on your computer differently, changes to this script might be required. For more information about SQL Server scripts, see SQL Server Books Online.

NOTE

Security Note This script gives db_owner permissions to the user that accesses the database at run time, which is what you'll have in the production environment. In some scenarios you might want to specify a user that has full database schema update permissions only for deployment, and specify for run time a different user that has permissions only to read and write data. For more information, see **Reviewing the Automatic Web.config Changes for Code First Migrations** in Deploying to IIS as a Test Environment.

Configuring Database Deployment for the Test Environment

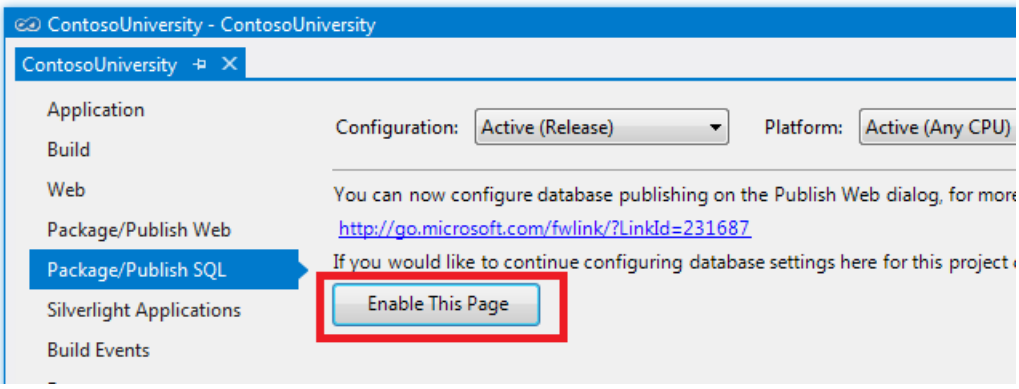
Next, you'll configure Visual Studio so that it will do the following tasks for each database:

- Generate a SQL script that creates the source database's structure (tables, columns, constraints, etc.) in the destination database.
- Generate a SQL script that inserts the source database's data into the tables in the destination database.
- Run the generated scripts, and the Grant script that you created, in the destination database.

Open the **Project Properties** window and select the **Package/Publish SQL** tab.

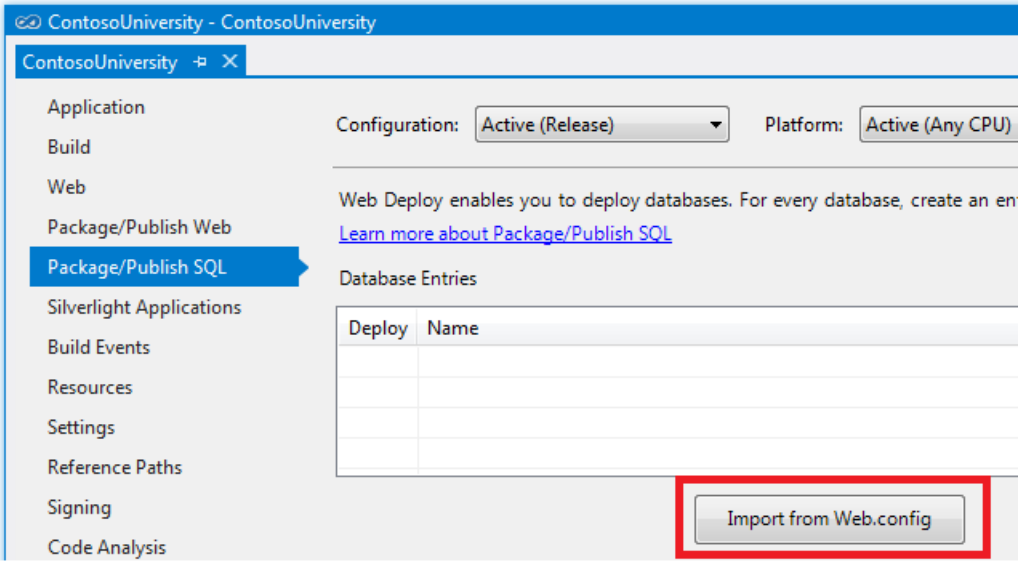
Make sure that **Active (Release)** or **Release** is selected in the **Configuration** drop-down list.

Click **Enable this Page**.



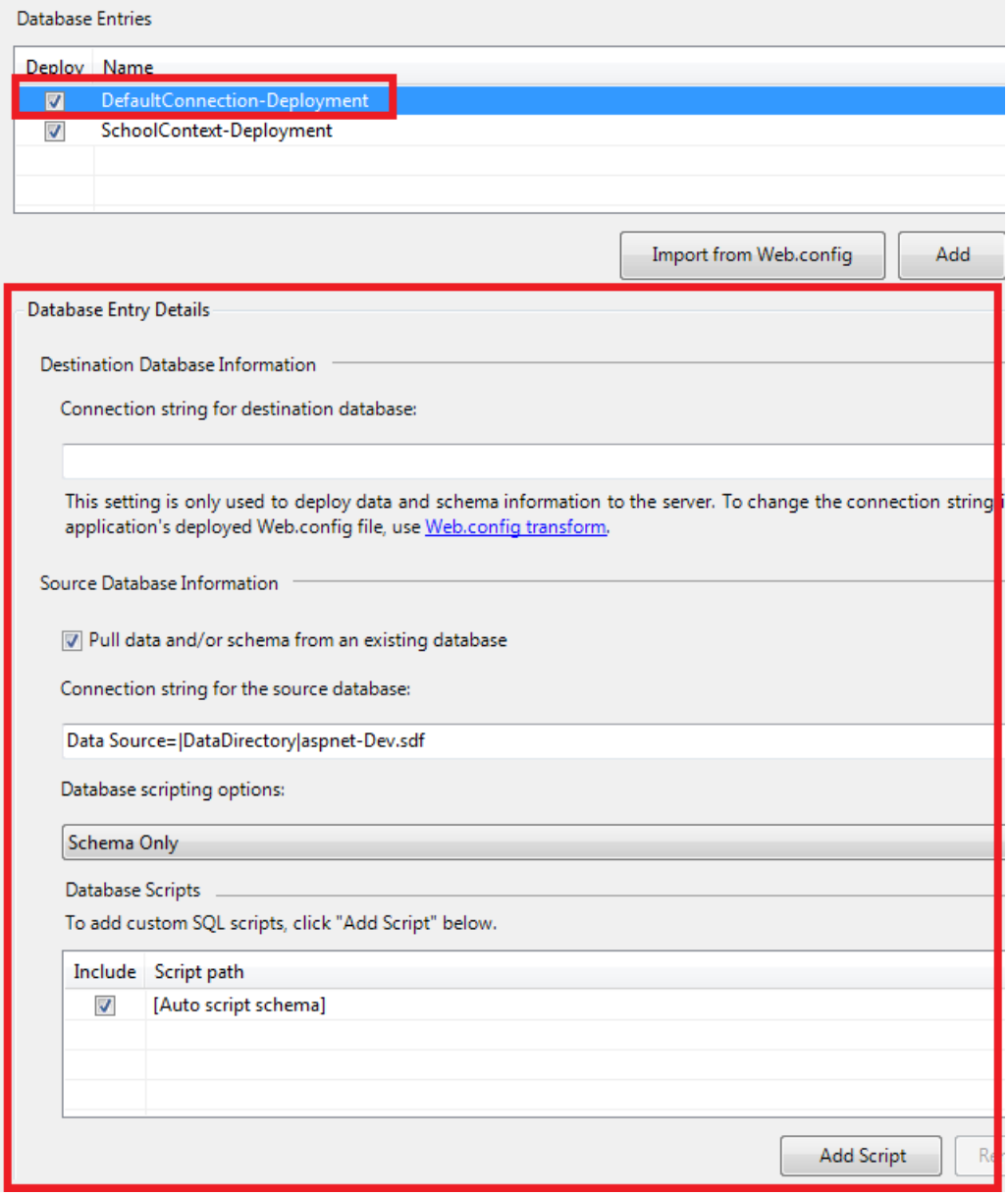
The **Package/Publish SQL** tab is normally disabled because it specifies a legacy deployment method. For most scenarios, you should configure database deployment in the **Publish Web** wizard. Migrating from SQL Server Compact to SQL Server or SQL Server Express is a special case for which this method is a good choice.

Click **Import from Web.config**.



Visual Studio looks for connection strings in the *Web.config* file, finds one for the membership database and one for the School database, and adds a row corresponding to each connection string in the **Database Entries** table. The connection strings it finds are for the existing SQL Server Compact databases, and your next step will be to configure how and where to deploy these databases.

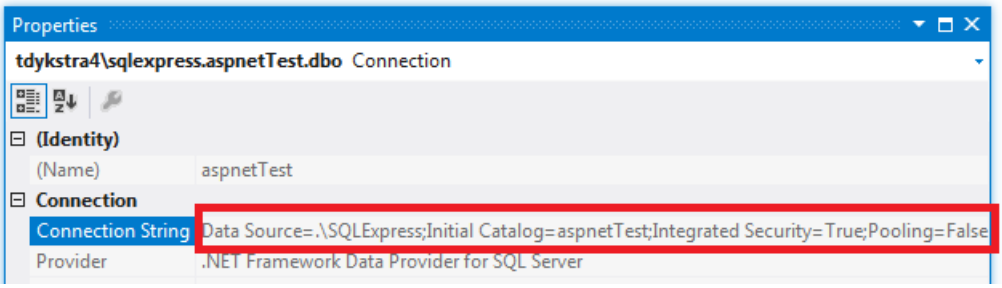
You enter database deployment settings in the **Database Entry Details** section below the **Database Entries** table. The settings shown in the **Database Entry Details** section pertain to whichever row in the **Database Entries** table is selected, as shown in the following illustration.



Configuring Deployment Settings for the Membership Database

Select the **DefaultConnection-Deployment** row in the **Database Entries** table in order to configure settings that apply to the membership database.

In **Connection string for destination database**, enter a connection string that points to the new SQL Server Express membership database. You can get the connection string you need from **Server Explorer**. In **Server Explorer**, expand **Data Connections** and select the **aspnetTest** database, then from the **Properties** window copy the **Connection String** value.



The same connection string is reproduced here:

```
Data Source=.\SQLExpress;Initial Catalog=aspnet-Test;Integrated Security=True;Pooling=False
```

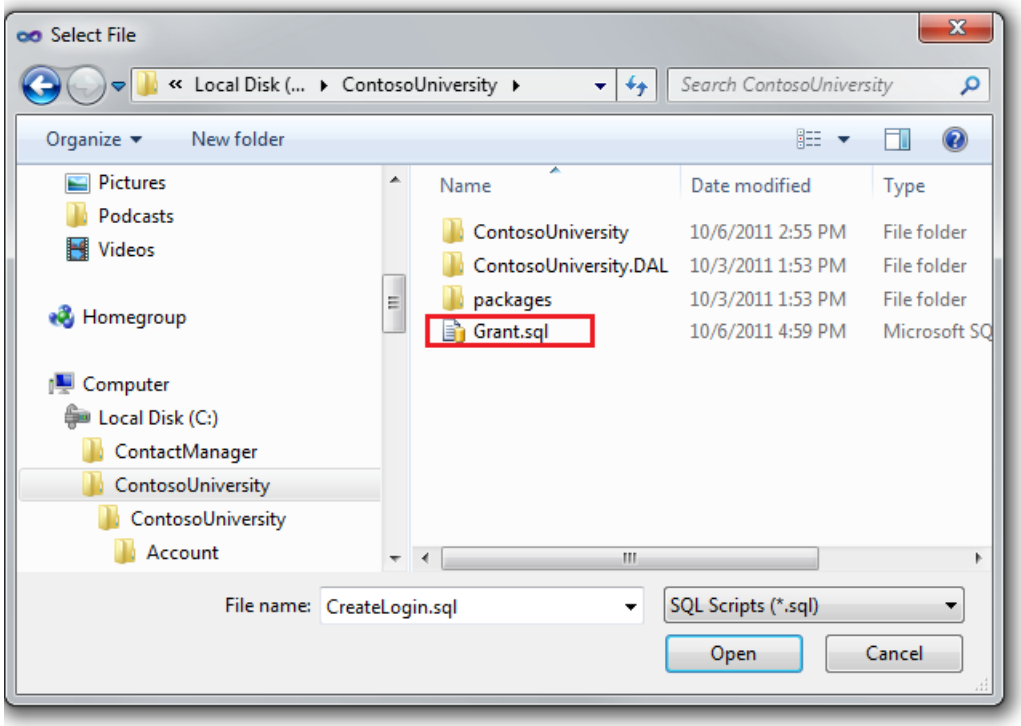
Copy and paste this connection string into **Connection string for destination database** in the **Package/Publish SQL** tab.

Make sure that **Pull data and/or schema from an existing database** is selected. This is what causes SQL scripts to be automatically generated and run in the destination database.

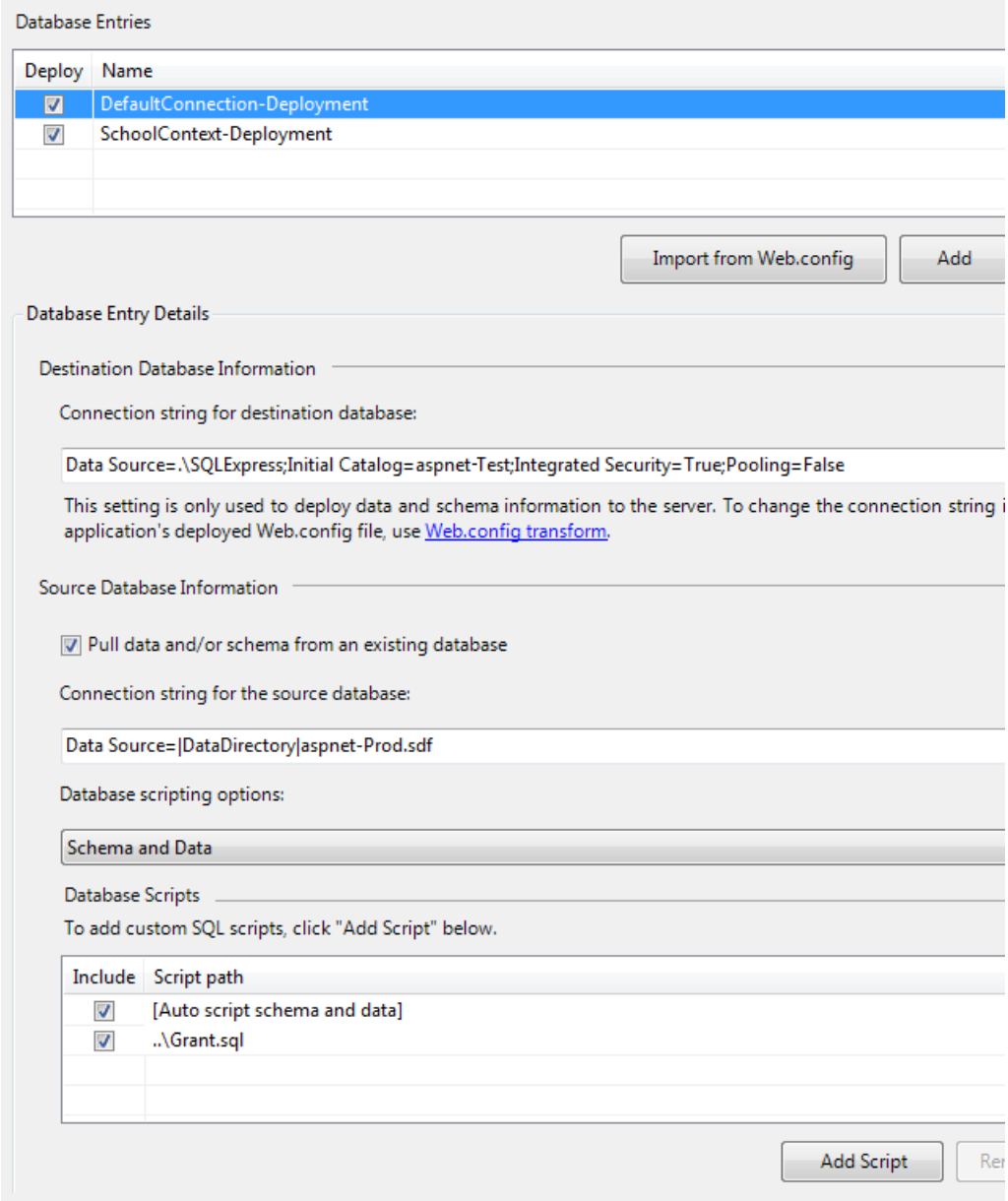
The **Connection string for the source database** value is extracted from the *Web.config* file and points to the development SQL Server Compact database. This is the source database that will be used to generate the scripts that will run later in the destination database. Since you want to deploy the production version of the database, change "aspnet-Dev.sdf" to "aspnet-Prod.sdf".

Change **Database scripting options** from **Schema Only** to **Schema and data**, since you want to copy your data (user accounts and roles) as well as the database structure.

To configure deployment to run the grant scripts that you created earlier, you have to add them to the **Database Scripts** section. Click **Add Script**, and in the **Add SQL Scripts** dialog box, navigate to the folder where you stored the grant script (this is the folder that contains your solution file). Select the file named *Grant.sql*, and click **Open**.



The settings for the **DefaultConnection-Deployment** row in **Database Entries** now look like the following illustration:



Configuring Deployment Settings for the School Database

Next, select the **SchoolContext-Deployment** row in the **Database Entries** table in order to configure deployment settings for the School database.

You can use the same method you used earlier to get the connection string for the new SQL Server Express database. Copy this connection string into **Connection string for destination database** in the **Package/Publish SQL** tab.

```
Data Source=.\SQLExpress;Initial Catalog=School-Test;Integrated Security=True;Pooling=False
```

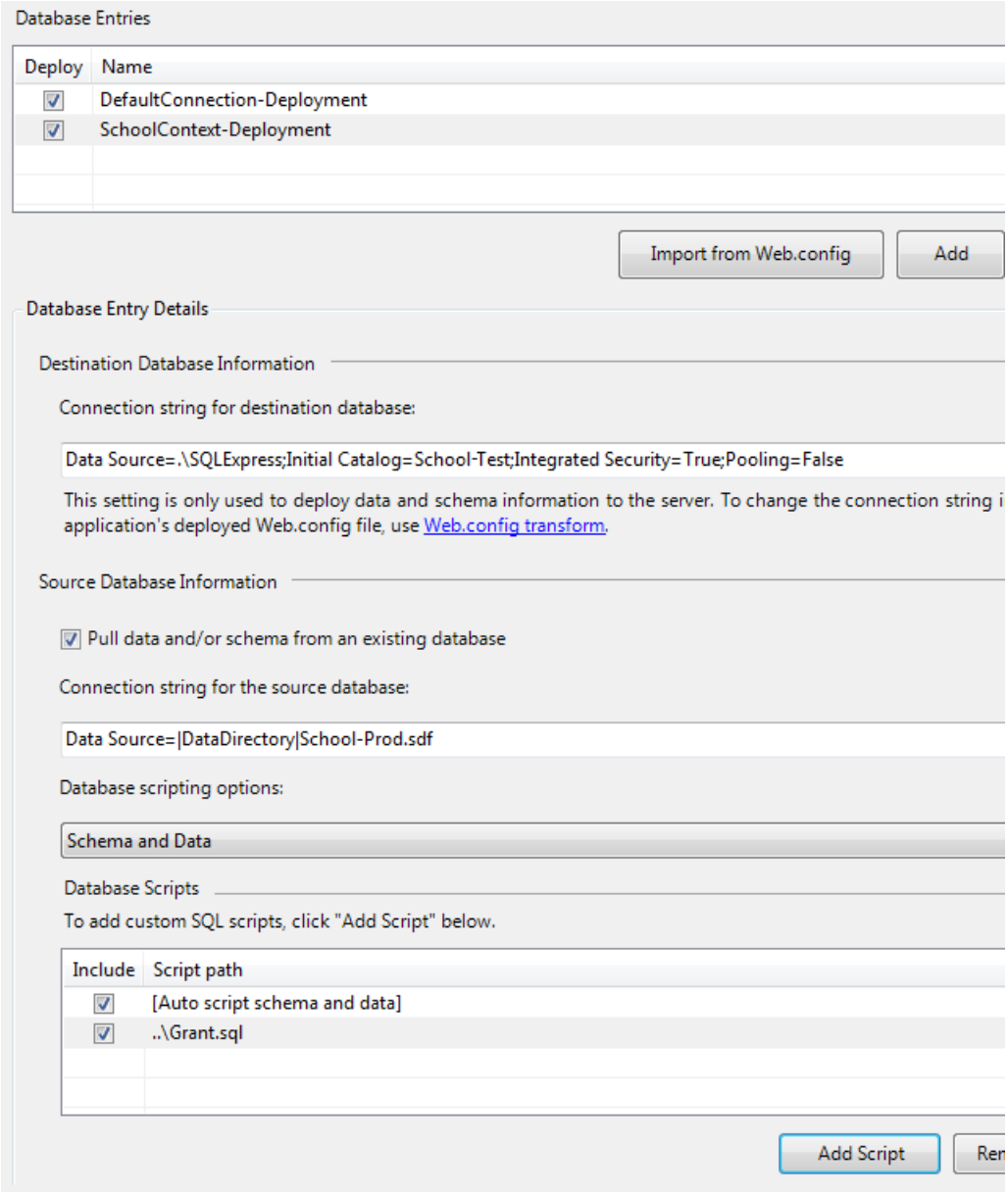
Make sure that **Pull data and/or schema from an existing database** is selected.

The **Connection string for the source database** value is extracted from the *Web.config* file and points to the development SQL Server Compact database. Change "School-Dev.sdf" to "School-Prod.sdf" to deploy the production version of the database. (You never created a School-Prod.sdf file in the App_Data folder, so you'll copy that file from the test environment to the App_Data folder in the ContosoUniversity project folder later.)

Change **Database scripting options** to **Schema and data**.

You also want to run the script to grant read and write permission for this database to the application pool identity, so add the *Grant.sql* script file as you did for the membership database.

When you're done, the settings for the **SchoolContext-Deployment** row in **Database Entries** look like the following illustration:



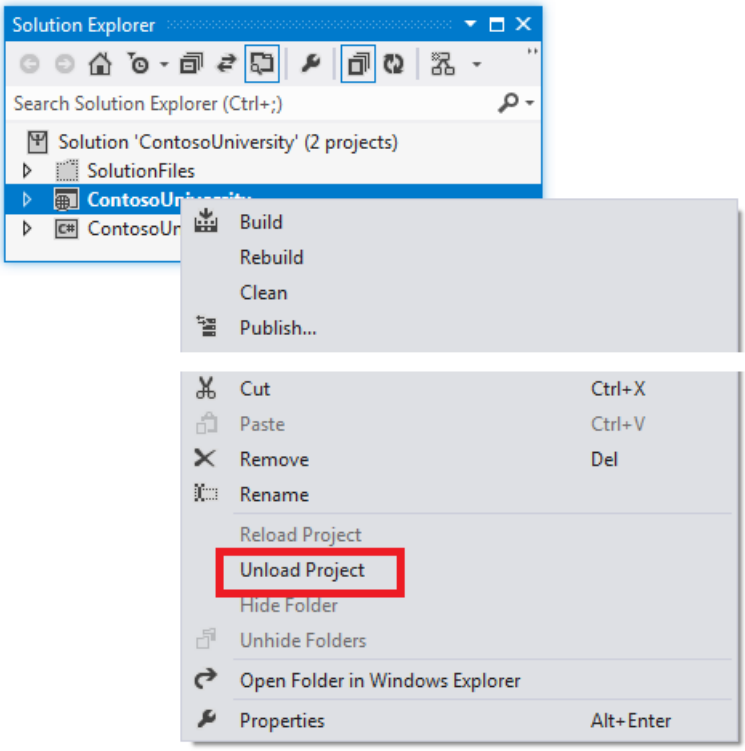
Save the changes to the **Package/Publish SQL** tab.

Copy the *School-Prod.sdf* file from the *c:\inetpub\wwwroot\ContosoUniversity\App_Data* folder to the *App_Data* folder in the ContosoUniversity project.

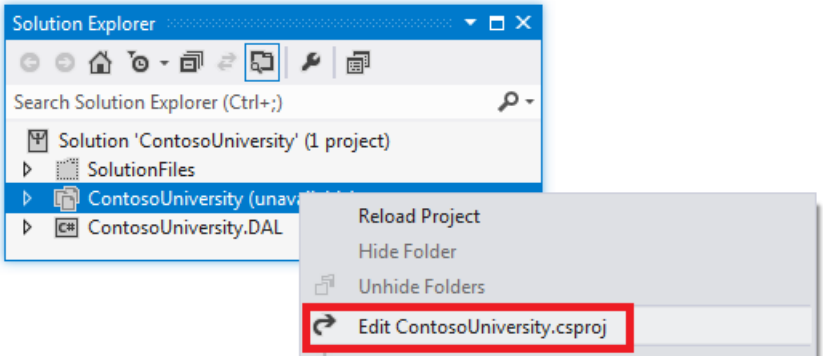
Specifying Transacted Mode for the Grant Script

The deployment process generates scripts that deploy the database schema and data. By default, these scripts run in a transaction. However, custom scripts (like the grant scripts) by default do not run in a transaction. If the deployment process mixes transaction modes, you might get a timeout error when the scripts run during deployment. In this section, you edit the project file in order to configure the custom scripts to run in a transaction.

In **Solution Explorer**, right-click the **ContosoUniversity** project and select **Unload Project**.



Then right-click the project again and select **Edit ContosoUniversity.csproj**.



The Visual Studio editor shows you the XML content of the project file. Notice that there are several `PropertyGroup` elements. (In the image, the contents of the `PropertyGroup` elements have been omitted.)

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <Project ToolsVersion="4.0" DefaultTargets="Build" xmlns="http://schemas.microsof
3     <PropertyGroup>
17    </PropertyGroup>
18    <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
26    </PropertyGroup>
27    <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' '
58    </PropertyGroup>
59    <PropertyGroup Condition="'$(Configuration)|$(Platform)' == 'Test|AnyCPU'">
99    </PropertyGroup>
```

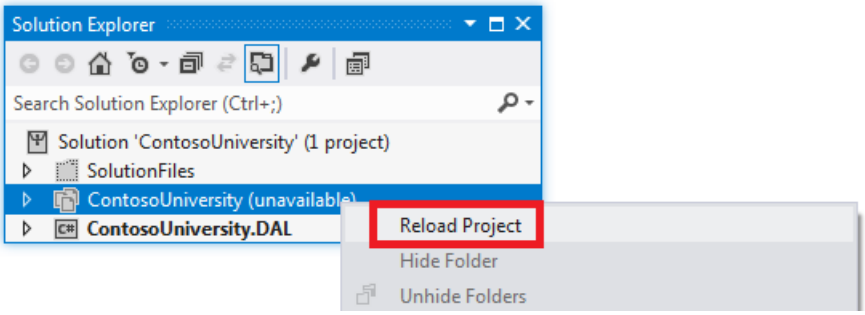
The first one, which has no `Condition` attribute, is for settings that apply regardless of build configuration. One `PropertyGroup` element applies only to the Debug build configuration (note the `Condition` attribute), one applies only to the Release build configuration, and one applies only to the Test build configuration. Within the `PropertyGroup` element for the Release build configuration, you'll see a `PublishDatabaseSettings` element that contains the settings you entered on the **Package/Publish SQL** tab. There is an `Object` element that corresponds to each of the grant scripts you specified (notice the two instances of "Grant.sql"). By default, the `Transacted` attribute of the `Source` element for each grant script is `False`.

```
<PublishDatabaseSettings>
  <Objects>
    <ObjectGroup Name="DefaultConnection-Deployment" Order="1" Enabled="True">
      <Destination Path="Data Source=localhost\SQLExpress%3bInitial Catalog=asp
      <Object Type="dbFullSql" Enabled="True">
        <PreSource Path="data source=.\SQLEXPRESS%3bIntegrated Security=SSPI%3b
        <Source Path="obj\Test\AutoScripts\DefaultConnection-Deployment_SchemaA
      </Object>
      <Object Type="dbFullSql" xmlns="">
        <Source Path="..\Grant.sql" Transacted="False" />
      </Object>
    </ObjectGroup>
    <ObjectGroup Name="SchoolContext-Deployment" Order="2" Enabled="True">
      <Destination Path="Data Source=localhost\SQLExpress%3bInitial Catalog=Sch
      <Object Type="dbFullSql" Enabled="True">
        <PreSource Path="data source=.\SQLEXPRESS%3bIntegrated Security=SSPI%3b
        <Source Path="obj\Test\AutoScripts\SchoolContext-Deployment_SchemaAndDa
      </Object>
      <Object Type="dbFullSql" xmlns="">
        <Source Path="..\Grant.sql" Transacted="False" />
      </Object>
    </ObjectGroup>
  </Objects>
</PublishDatabaseSettings>
```

Change the value of the `Transacted` attribute of the `Source` element to `True`.

```
<PublishDatabaseSettings>
  <Objects>
    <ObjectGroup Name="DefaultConnection-Deployment" Order="1" Enabled="True">
      <Destination Path="Data Source=localhost\SQLExpress%3bInitial Catalog=as
      <Object Type="dbFullSql" Enabled="True">
        <PreSource Path="data source=.\SQLEXPRESS%3bIntegrated Security=SSPI%3
        <Source Path="obj\Test\AutoScripts\DefaultConnection-Deployment_Schema
      </Object>
      <Object Type="dbFullSql" xmlns="">
        <Source Path="..\Grant.sql" Transacted="True" />
      </Object>
    </ObjectGroup>
    <ObjectGroup Name="SchoolContext-Deployment" Order="2" Enabled="True">
      <Destination Path="Data Source=localhost\SQLExpress%3bInitial Catalog=Sc
      <Object Type="dbFullSql" Enabled="True">
        <PreSource Path="data source=.\SQLEXPRESS%3bIntegrated Security=SSPI%3
        <Source Path="obj\Test\AutoScripts\SchoolContext-Deployment_SchemaAndD
      </Object>
      <Object Type="dbFullSql" xmlns="">
        <Source Path="..\Grant.sql" Transacted="True" />
      </Object>
    </ObjectGroup>
  </Objects>
</PublishDatabaseSettings>
```

Save and close the project file, and then right-click the project in **Solution Explorer** and select **Reload Project**.



Setting up Web.Config Transformations for the Connection Strings

The connection strings for the new SQL Express databases that you entered on the **Package/Publish SQL** tab are used by Web Deploy only for updating the destination database during deployment. You still have to set up *Web.config* transformations so that the connection strings in the deployed *Web.config* file point to the new SQL Server Express databases. (When you use the **Package/Publish SQL** tab, you can't configure connection strings in the publish profile.)

Open *Web.Test.config* and replace the `connectionStrings` element with the `connectionStrings` element in the following example. (Make sure you only copy the connectionStrings element, not the surrounding code that is shown here to provide context.)

```
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <connectionStrings>
    <add name="DefaultConnection"
         connectionString="Data Source=.\SQLExpress;Initial Catalog=aspnet-Test;Integrated
Security=True;Pooling=False;MultipleActiveResultSets=True"
         providerName="System.Data.SqlClient"
         xdt:Transform="SetAttributes" xdt:Locator="Match(name)"/>
    <add name="SchoolContext"
         connectionString="Data Source=.\SQLExpress;Initial Catalog=School-Test;Integrated
Security=True;Pooling=False;MultipleActiveResultSets=True"
         providerName="System.Data.SqlClient"
         xdt:Transform="SetAttributes" xdt:Locator="Match(name)"/>
  </connectionStrings>
  <!-- appSettings element, comments, and system.web element -->
</configuration>
```

This code causes the `connectionString` and `providerName` attributes of each `add` element to be replaced in the deployed *Web.config* file. These connection strings are not identical to the ones you entered in the **Package/Publish SQL** tab. The setting "MultipleActiveResultSets=True" has been added to them because it's required for the Entity Framework and the Universal Providers.

Installing SQL Server Compact

The SqlServerCompact NuGet package provides the SQL Server Compact database engine assemblies for the Contoso University application. But now it is not the application but Web Deploy that must be able to read the SQL Server Compact databases, in order to create scripts to run in the SQL Server databases. To enable Web Deploy to read SQL Server Compact databases, install SQL Server Compact on the development computer by using the following link: Microsoft SQL Server Compact 4.0.

Deploying to the Test Environment

In order to publish to the Test environment, you have to create a publish profile that is configured to use the **Package/Publish SQL** tab for database publishing instead of the publish profile database settings.

First, delete the existing Test profile.

In **Solution Explorer**, right-click the ContosoUniversity project, and click **Publish**.

Select the **Profile** tab.

Click **Manage Profiles**.

Select **Test**, click **Remove**, and then click **Close**.

Close the **Publish Web** wizard to save this change.

Next, create a new Test profile and use it to publish the project.

In **Solution Explorer**, right-click the ContosoUniversity project, and click **Publish**.

Select the **Profile** tab.

Select **<New...>** from the drop-down list, and enter "Test" as the profile name.

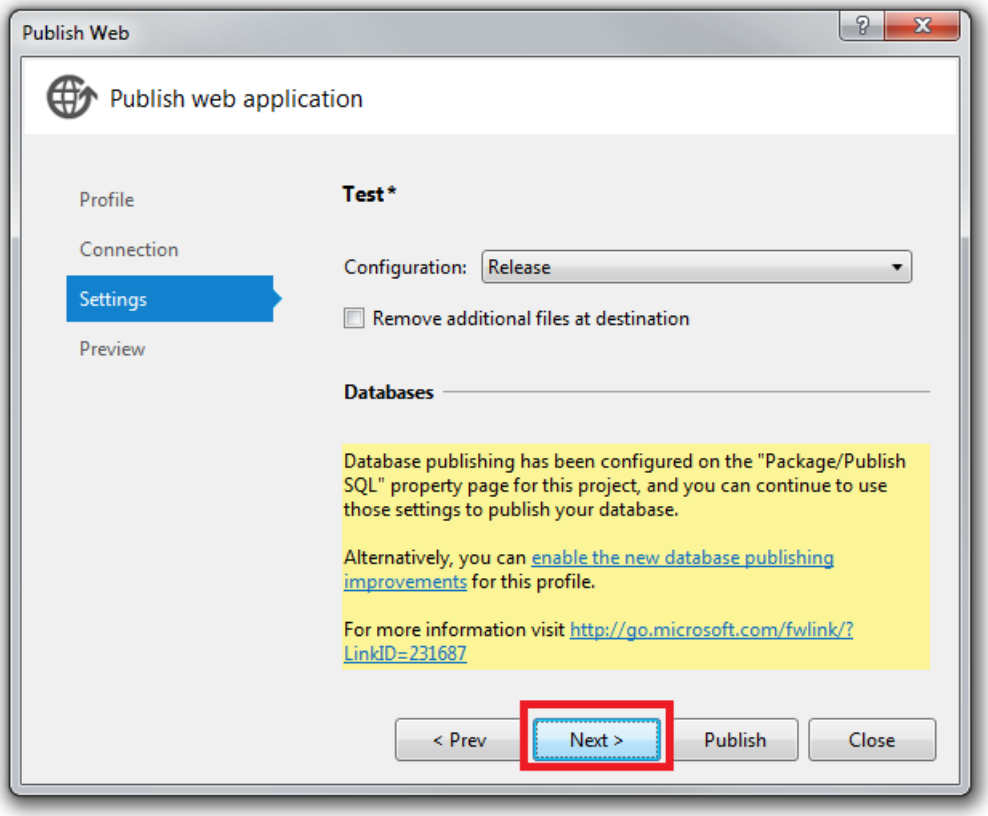
In the **Service URL** box, enter *localhost*.

In the **Site/application** box, enter *Default Web Site/ContosoUniversity*.

In the **Destination URL** box, enter `http://localhost/ContosoUniversity/`.

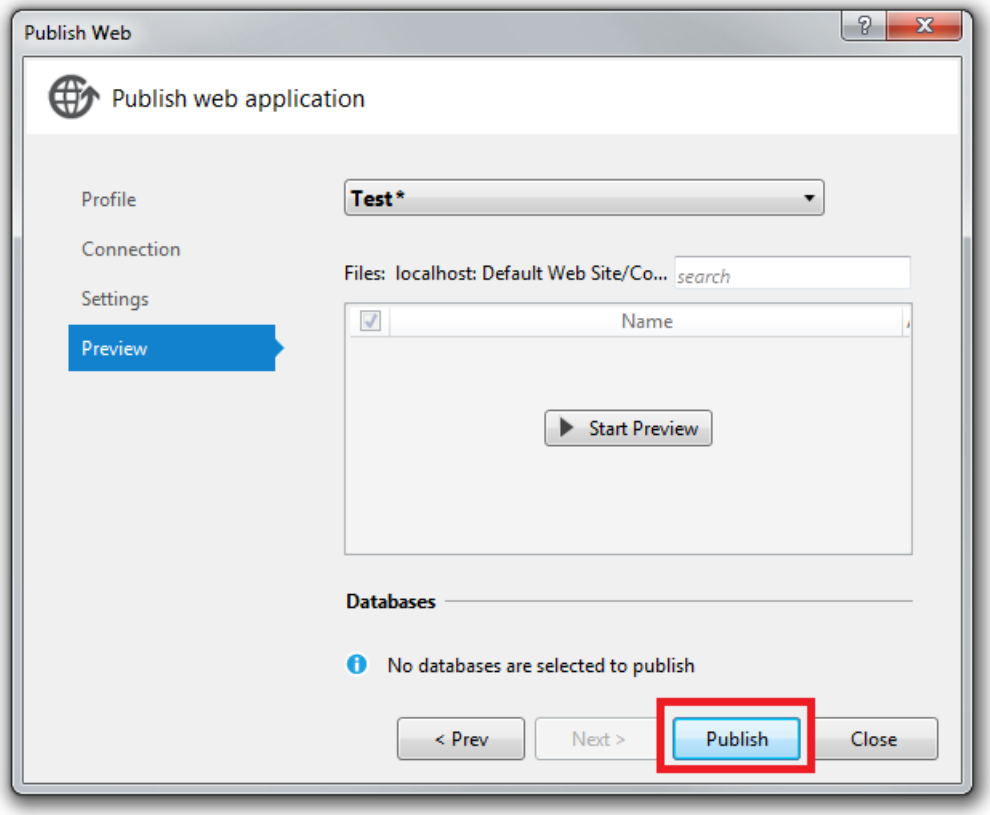
Click **Next**.

The **Settings** tab warns you that the **Package/Publish SQL** tab has been configured, and it gives you an opportunity to override them by clicking enable the new database publishing improvements. For this deployment you don't want to override the **Package/Publish SQL** tab settings, so just click **Next**.



A message on the **Preview** tab indicates that **No databases are selected to publish**, but this only means that database publishing is not configured in the publish profile.

Click **Publish**.



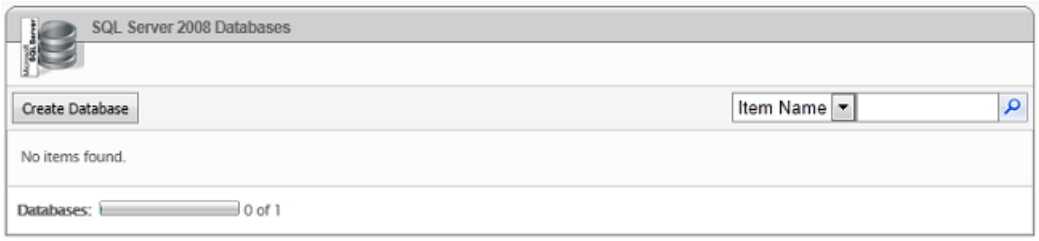
Visual Studio deploys the application and opens the browser to the home page of the site in the test environment. Run the Instructors page to see that it displays the same data that you saw earlier. Run the **Add Students** page, add a new student, and then view the new student in the **Students** page. This verifies that the you can update the database. Select the **Update Credits** page (you'll need to log in) to verify that the membership database was deployed and you have access to it.

Creating a SQL Server Database for the Production Environment

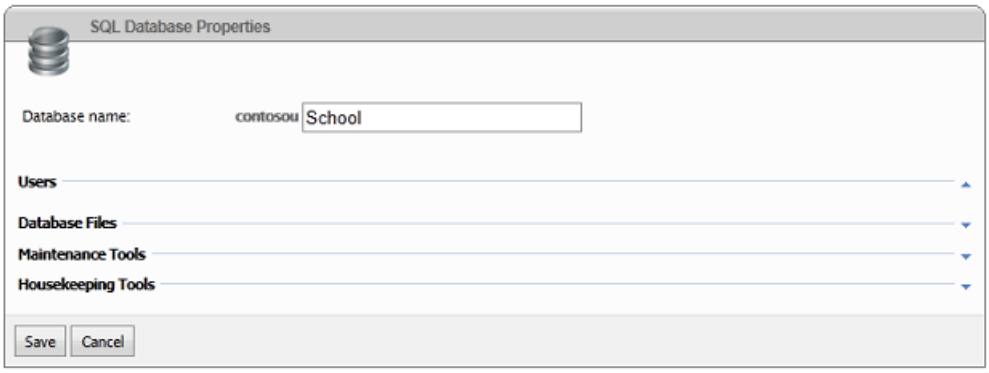
Now that you've deployed to the test environment, you're ready to set up deployment to production. You begin as you did for the test environment, by creating a database to deploy to. As you recall from the Overview, the Cytanium Lite hosting plan only allows a single SQL Server database, so you will set up only one database, not two. All of the tables and data from the membership and School SQL Server Compact databases will be deployed into one SQL Server database in production.

Go to the Cytanium control panel at http://panel.cytanium.com. Hold the mouse over **Databases** and then click **SQL Server 2008**.

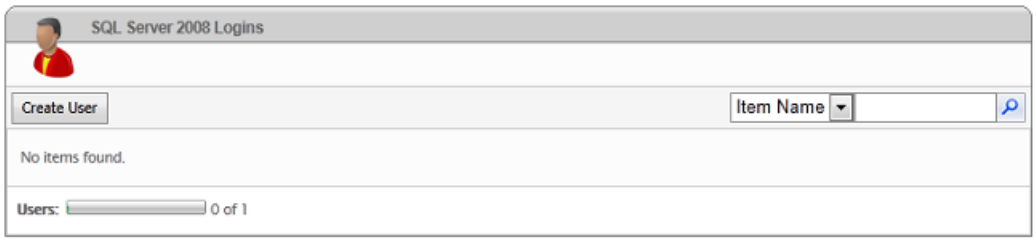
In the **SQL Server 2008** page, click **Create Database**.



Name the database "School" and click **Save**. (The page automatically adds the prefix "contosou", so the effective name will be "contosouSchool".)



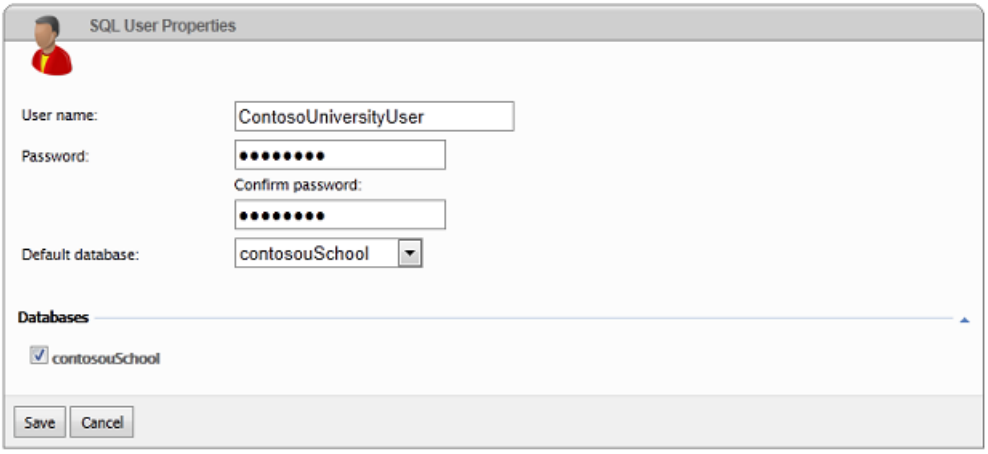
On the same page, click **Create User**. On Cytanium's servers, rather than using integrated Windows security and letting the application pool identity open your database, you'll create a user that has authority to open your database. You'll add the user's credentials to the connection strings that go in the production *Web.config* file. In this step you create those credentials.



Fill in the required fields in the **SQL User Properties** page:

- Enter "ContosoUniversityUser" as the name.
- Enter a password.

- Select **contosouSchool** as the default database.
- Select the **contosouSchool** check box.



Configuring Database Deployment for the Production Environment

Now you're ready to set up database deployment settings in the **Package/Publish SQL** tab, as you did earlier for the test environment.

Open the **Project Properties** window, select the **Package/Publish SQL** tab, and make sure that **Active (Release)** or **Release** is selected in the **Configuration** drop-down list.

When you configure deployment settings for each database, the key difference between what you do for production and test environments is in how you configure connection strings. For the test environment you entered different destination database connection strings, but for the production environment the destination connection string will be the same for both databases. This is because you are deploying both databases to one database in production.

Configuring Deployment Settings for the Membership Database

To configure settings that apply to the membership database, select the **DefaultConnection-Deployment** row in the **Database Entries** table.

In **Connection string for destination database**, enter a connection string that points to the new production SQL Server database that you just created. You can get the connection string from your welcome email. The relevant part of the email contains the following sample connection string:

```
Data Source=vserver01.cytanium.com;Initial Catalog={myDataBase};User Id={myUsername};Password={myPassword};
```

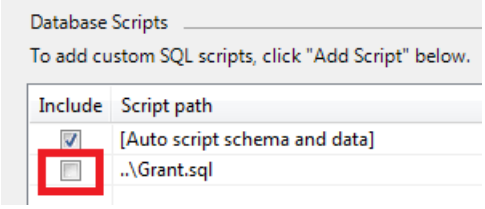
After you replace the three variables, the connection string you need looks like this example:

```
Data Source=vserver01.cytanium.com;Initial Catalog=contosouSchool;User
Id=ContosoUniversityUser;Password=Password;
```

Copy and paste this connection string into **Connection string for destination database** in the **Package/Publish SQL** tab.

Make sure that **Pull data and/or schema from an existing database** is still selected, and that **Database scripting options** is still **Schema and Data**.

In the **Database Scripts** box, clear the check box next to the Grant.sql script.



Configuring Deployment Settings for the School Database

Next, select the **SchoolContext-Deployment** row in the **Database Entries** table in order to configure the School database settings.

Copy the same connection string into **Connection string for destination database** that you copied into that field for the membership database.

Make sure that **Pull data and/or schema from an existing database** is still selected, and that **Database scripting options** is still **Schema and Data**.

In the **Database Scripts** box, clear the check box next to the Grant.sql script.

Save the changes to the **Package/Publish SQL** tab.

Setting Up Web.Config Transforms for the Connection Strings to Production Databases

Next, you'll set up *Web.config* transformations so that the connection strings in the deployed *Web.config* file to point to the new production database. The connection string that you entered on the **Package/Publish SQL** tab for Web Deploy to use is the same as the one the application needs to use, except for the addition of the `MultipleResultSets` option.

Open *Web.Production.config* and replace the `connectionStrings` element with a `connectionStrings` element that looks like the following example. (Only copy the `connectionStrings` element, not the surrounding tags that are provided to show the context.)

```
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <connectionStrings>
    <add name="DefaultConnection"
         connectionString="Data Source=vserver01.cytanium.com;Initial Catalog=contosouSchool;User
Id=ContosoUniversityUser;Password=Password;MultipleActiveResultSets=True"
         providerName="System.Data.SqlClient"
         xdt:Transform="SetAttributes" xdt:Locator="Match(name)"/>
    <add name="SchoolContext"
         connectionString="Data Source=vserver01.cytanium.com;Initial Catalog=contosouSchool;User
Id=ContosoUniversityUser;Password=Password;MultipleActiveResultSets=True"
         providerName="System.Data.SqlClient"
         xdt:Transform="SetAttributes" xdt:Locator="Match(name)"/>
  </connectionStrings>
  <!-- appSettings element, comments, and system.web element -->
</configuration>
```

You sometimes see advice that tells you to always encrypt connection strings in the *Web.config* file. This might be appropriate if you were deploying to servers on your own company's network. When you are deploying to a shared hosting environment, though, you're trusting the security practices of the hosting provider, and it's not necessary or practical to encrypt the connection strings.

Deploying to the Production Environment

Now you're ready to deploy to production. Web Deploy will read the SQL Server Compact databases in your project's *App_Data* folder and re-create all of their tables and data in the production SQL Server database. In

order to publish by using the **Package/Publish Web** tab settings, you have to create a new publish profile for production.

First, delete the existing Production profile as you did the Test profile earlier.

In **Solution Explorer**, right-click the ContosoUniversity project, and click **Publish**.

Select the **Profile** tab.

Click **Manage Profiles**.

Select **Production**, click **Remove**, and then click **Close**.

Close the **Publish Web** wizard to save this change.

Next, create a new Production profile and use it to publish the project.

In **Solution Explorer**, right-click the ContosoUniversity project, and click **Publish**.

Select the **Profile** tab.

Click **Import**, and select the .publishsettings file that you downloaded earlier.

On the **Connection** tab, change the **Destination URL** to the correct temporary URL, which in this example is http://contosouniversity.com.vserver01.cytanium.com.

Rename the profile to Production. (Select the **Profile** tab and click **Manage Profiles** to do that).

Close the **Publish Web** wizard to save your changes.

In a real application in which the database was being updated in production, you would do two additional steps now before you publish:

1. Upload *app_offline.htm*, as shown in the Deploying to the Production Environment tutorial.
2. Use the **File Manager** feature of the Cytanium control panel to copy the *aspnet-Prod.sdf* and *School-Prod.sdf* files from the production site to the *App_Data* folder of the ContosoUniversity project. This ensures that the data you're deploying to the new SQL Server database includes the latest updates made by your production website.

In the **Web One Click Publish** toolbar, make sure that the **Production** profile is selected, and then click **Publish**.

If you uploaded *app_offline.htm* before publishing, you have to use the **File Manager** utility in the Cytanium control panel to delete *app_offline*.htm before you test. You can also at the same time delete the *.sdf* files from the *App_Data* folder.

You can now open a browser and go to the URL of your public site to test the application the same way you did after deploying to the test environment.

Switching to SQL Server Express LocalDB in Development

As was explained in the Overview, it's generally best to use the same database engine in development that you use in test and production. (Remember that the advantage to using SQL Server Express in development is that the database will work the same in your development, test, and production environments.) In this section you'll set up the ContosoUniversity project to use SQL Server Express LocalDB when you run the application from Visual Studio.

The simplest way to perform this migration is to let Code First and the membership system create both new development databases for you. Using this method to migrate requires three steps:

1. Change the connection strings to specify new SQL Express LocalDB databases.

2. Run the Web Site Administration Tool to create an administrator user. This creates the membership database.
3. Use the Code First Migrations update-database command to create and seed the application database.

Updating Connection Strings in the Web.config file

Open the *Web.config* file and replace the `connectionStrings` element with the following code:

```
<connectionStrings>
  <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\v11.0;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|\aspnet-Dev.mdf;MultipleActiveResultSets=True;"
providerName="System.Data.SqlClient" />
  <add name="SchoolContext" connectionString="Data Source=(LocalDb)\v11.0;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|\School-Dev.mdf;MultipleActiveResultSets=True;"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

Creating the Membership Database

In **Solution Explorer**, select the ContosoUniversity project, and then click **ASP.NET Configuration** in the **Project** menu.

Select the Security tab.

Click **Create or Manage Roles**, and then create an **Administrator** role.

Return to the Security tab.

Click **Create user**, and then select the **Administrator** check box and create a user named admin.

Close the **Web Site Administration Tool**.

Creating the School Database

Open the Package Manager Console window.

In the **Default project** drop-down list, select the ContosoUniversity.DAL project.

Enter the following command:

```
update-database
```

Code First Migrations applies the Initial migration that creates the database and then applies the AddBirthDate migration, then it runs the Seed method.

Run the site by pressing Control-F5. As you did for the test and production environments, run the **Add Students** page, add a new student, and then view the new student in the **Students** page. This verifies that the School database was created and initialized and that you have read and write access to it.

Select the **Update Credits** page and log in to verify that the membership database was deployed and that you have access to it. If you did not migrate your user accounts, create an administrator account and then select the **Update Credits** page to verify that it works.

Cleaning Up SQL Server Compact Files

You no longer need files and NuGet packages that were included to support SQL Server Compact. If you want (this step is not required), you can clean up unneeded files and references.

In **Solution Explorer**, delete the *.sdf* files from the *App_Data* folder and the *amd64* and *x86* folders from the *bin* folder.

In **Solution Explorer**, right-click the solution (not one of the projects), and then click **Manage NuGet Packages**

for Solution.

In the left pane of the **Manage NuGet Packages** dialog box, select **Installed packages**.

Select the **EntityFramework.SqlServerCompact** package and click **Manage**.

In the **Select Projects** dialog box, both projects are selected. To uninstall the package in both projects, clear both check boxes, then click **OK**.

In the dialog box that asks if you want to uninstall the dependent packages also, click No. One of these is the Entity Framework package that you have to keep.

Follow the same procedure to uninstall the **SqlServerCompact** package. (The packages must be uninstalled in this order because the **EntityFramework.SqlServerCompact** package depends on the **SqlServerCompact** package.)

You have now successfully migrated to SQL Server Express and full SQL Server. In the next tutorial you'll make another database change, and you'll see how to deploy database changes when your test and production databases use SQL Server Express and full SQL Server.

PREVIOUS NEXT

Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio or Visual Web Developer: Deploying a SQL Server Database Update - 11 of 12

2/20/2020 • 7 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This series of tutorials shows you how to deploy (publish) an ASP.NET web application project that includes a SQL Server Compact database by using Visual Studio 2012 RC or Visual Studio Express 2012 RC for Web. You can also use Visual Studio 2010 if you install the Web Publish Update. For an introduction to the series, see the first tutorial in the series.
>
> For a tutorial that shows deployment features introduced after the RC release of Visual Studio 2012, shows how to deploy SQL Server editions other than SQL Server Compact, and shows how to deploy to Windows Azure Web Sites, see ASP.NET Web Deployment using Visual Studio.

Overview

This tutorial shows you how to deploy a database update to a full SQL Server database. Because Code First Migrations does all the work of updating the database, the process is almost identical to what you did for SQL Server Compact in the Deploying a Database Update tutorial.

Reminder: If you get an error message or something doesn't work as you go through the tutorial, be sure to check the troubleshooting page.

Adding a New Column to a Table

In this section of the tutorial you'll make a database change and corresponding code changes, then test them in Visual Studio in preparation for deploying them to the test and production environments. The change involves adding an `OfficeHours` column to the `Instructor` entity and displaying the new information in the **Instructors** web page.

In the ContosoUniversity.DAL project, open *Instructor.cs* and add the following property between the `HireDate` and `Courses` properties:

```
[MaxLength(50)]
public string OfficeHours { get; set; }
```

Update the initializer class so that it seeds the new column with test data. Open *Migrations\Configuration.cs* and replace the code block that begins `var instructors = new List<Instructor>` with the following code block which includes the new column:

```
var instructors = new List<Instructor>
{
    new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie", HireDate = DateTime.Parse("1995-03-
11"), BirthDate = DateTime.Parse("1918-08-12"), OfficeHours = "8-9AM, 4-5PM", OfficeAssignment = new
OfficeAssignment { Location = "Smith 17" } },
    new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",    HireDate = DateTime.Parse("2002-07-
06"), BirthDate = DateTime.Parse("1960-03-15"), OfficeAssignment = new OfficeAssignment { Location = "Gowan
27" } },
    new Instructor { FirstMidName = "Roger",   LastName = "Harui",       HireDate = DateTime.Parse("1998-07-
01"), BirthDate = DateTime.Parse("1970-01-11"), OfficeHours = "6AM-6PM", OfficeAssignment = new
OfficeAssignment { Location = "Thompson 304" } },
    new Instructor { FirstMidName = "Candace", LastName = "Kapoor",      HireDate = DateTime.Parse("2001-01-
15"), BirthDate = DateTime.Parse("1975-04-11") },
    new Instructor { FirstMidName = "Roger",   LastName = "Zheng",       HireDate = DateTime.Parse("2004-02-
12"), BirthDate = DateTime.Parse("1957-10-12"), OfficeHours = "By appointment only" }
};
```

In the ContosoUniversity project, open *Instructors.aspx* and add a new template field for office hours just before the closing `</Columns>` tag in the first `GridView` control:

```
<asp:TemplateField HeaderText="Office Hours">
    <ItemTemplate>
        <asp:Label ID="InstructorOfficeHoursLabel" runat="server" Text='<%# Eval("OfficeHours") %>'>
</asp:Label>
    </ItemTemplate>
    <EditItemTemplate>
        <asp:TextBox ID="InstructorOfficeHoursTextBox" runat="server" Text='<%# Bind("OfficeHours") %>'
            Width="14em"></asp:TextBox>
    </EditItemTemplate>
</asp:TemplateField>
```

Build the solution.

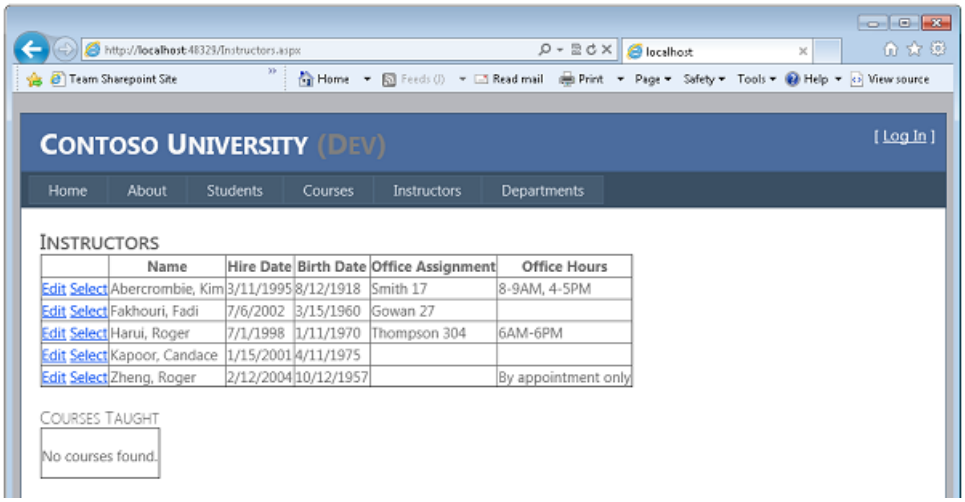
Open the **Package Manager Console** window, and select ContosoUniversity.DAL as the **Default project**.

Enter the following commands:

```
add-migration AddOfficeHoursColumn

update-database
```

Run the application and select the **Instructors** page. The page takes a little longer than usual to load, because the Entity Framework re-creates the database and seeds it with test data.



Deploying the Database Update to the Test Environment

When you use Code First Migrations, the method for deploying a database change to SQL Server is the same as for SQL Server Compact. However, you have to change the Test publish profile because it is still set up to migrate from SQL Server Compact to SQL Server.

The first step is to remove the connection string transformations that you created in the previous tutorial. These are no longer needed because you'll specify connection string transformations in the publish profile, as you did before you configured the **Package/Publish SQL** tab for migration to SQL Server.

Open the *Web.Test.config* file and remove the `connectionStrings` element. The only remaining transformation in the *Web.Test.config* file is for the `Environment` value in the `appSettings` element.

Now you can update the publish profile and publish to the test environment.

Open the **Publish Web** wizard, and then switch to the **Profile** tab.

Select the **Test** publish profile.

Select the **Settings** tab.

Click **enable the new database publishing improvements**.

In the connection string box for **SchoolContext**, enter the same value that you used in the *Web.Test.config* transformation file in the previous tutorial:

```
Data Source=.\SQLExpress;Initial Catalog=School-Test;Integrated
Security=True;Pooling=False;MultipleActiveResultSets=True
```

Select **Execute Code First Migrations (runs on application start)**. (In your version of Visual Studio, the check box might be labeled **Apply Code First Migrations**.)

In the connection string box for **DefaultConnection**, enter the same value that you used in the *Web.Test.config* transformation file in the previous tutorial:

```
Data Source=.\SQLExpress;Initial Catalog=aspnet-Test;Integrated
Security=True;Pooling=False;MultipleActiveResultSets=True
```

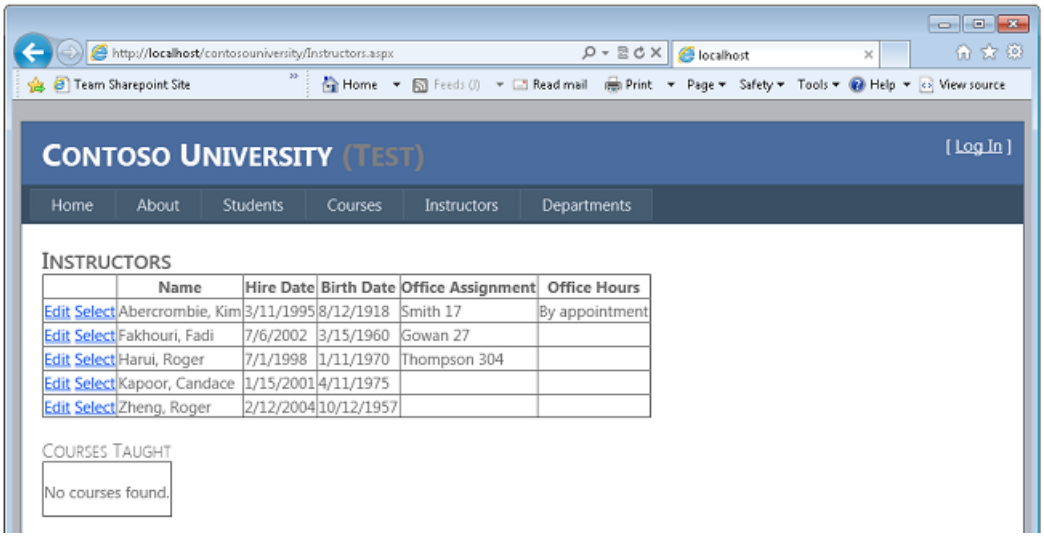
Leave **Update database** cleared.

Click **Publish**.

Visual Studio deploys the code changes to the test environment and opens the browser to the Contoso University home page.

Select the Instructors page.

When the application runs this page, it tries to access the database. Code First Migrations checks if the database is current, and finds that the latest migration has not been applied yet. Code First Migrations applies the latest migration, runs the `Seed` method, and then the page runs normally. You see the new Office Hours column with the seeded data.



Deploying the Database Update to the Production Environment

You have to change the publish profile for the production environment also. In this case you'll remove the existing profile and create a new one by importing an updated .publishsettings file. The updated file will include the connection string for the SQL Server database at Cytanium.

As you saw when you deployed to the test environment, you no longer need connection string transforms in the *Web.Production.config* transformation file. Open that file and remove the `connectionStrings` element. The remaining transformations are for the `Environment` value in the `appSettings` element and the `location` element that restricts access to Elmah error reports.

Before you create a new publish profile for production, download an updated .publishsettings file the same way you did earlier in the Deploying to the Production Environment tutorial. (In the Cytanium control panel, click **Web Sites**, and then click the **contosouniversity.com** website. Select the **Web Publishing** tab, and then click **Download Publishing Profile for this web site**.) The reason you are doing this is to pick up the database connection string in the .publishsettings file. The connection string wasn't available the first time you downloaded the file, because you were still using SQL Server Compact and hadn't created the SQL Server database at Cytanium yet.

Now you can update the publish profile and publish to the production environment.

Open the **Publish Web** wizard, and then switch to the **Profile** tab.

Click **Manage Profiles**, and then delete the Production profile.

Close the **Publish Web** wizard to save this change.

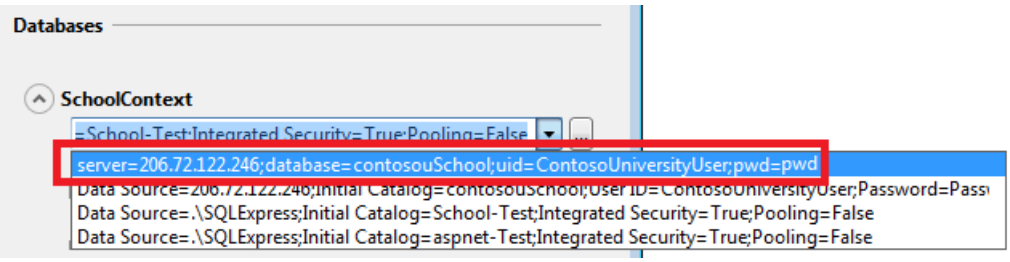
Open the **Publish Web** wizard again, and then click **Import**.

On the **Connection** tab, change **Destination URL** to the appropriate value if you are using a temporary URL.

Click **Next**.

On the **Settings** tab, click **enable the new database publishing improvements**.

In the connection string drop-down list for **SchoolContext**, select the Cytanium connection string.



Select **Execute Code First migrations (runs on application start)**.

In the connection string drop-down list for **DefaultConnection**, select the Cytanium connection string.

Select the **Profile** tab, click **Manage Profiles**, and rename the profile from "contosouniversity.com - Web Deploy" to "Production".

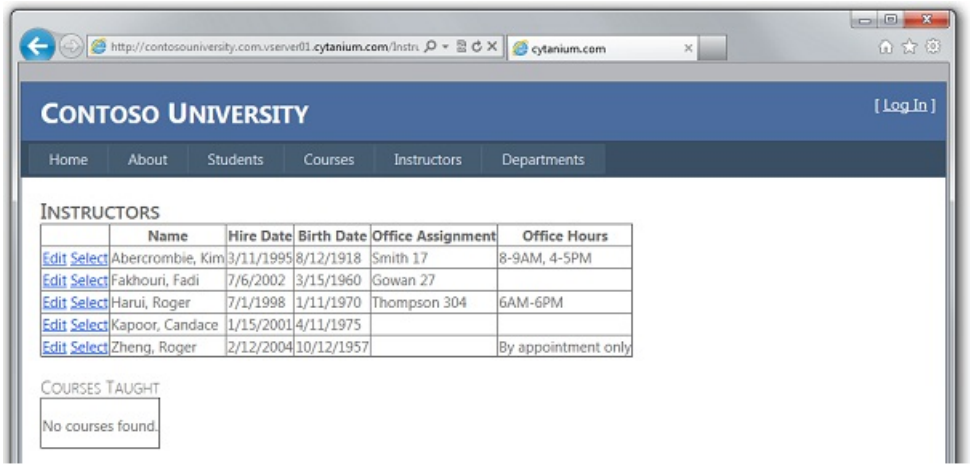
Close the publish profile to save the change, then open it again.

Click **Publish**. (For a real production website, you would copy *app_offline.htm* to production and put it in your project folder before publishing, then remove it when deployment is complete.)

Visual Studio deploys the code changes to the test environment and opens the browser to the Contoso University home page.

Select the Instructors page.

Code First Migrations updates the database the same way it did in the Test environment. You see the new Office Hours column with the seeded data.



You have now successfully deployed an application update that included a database change, using a SQL Server database.

More Information

This completes this series of tutorials on deploying an ASP.NET web application to a third-party hosting provider. For more information about any of the topics covered in these tutorials, see the ASP.NET Deployment Content Map on the MSDN web site.

Acknowledgements

I would like to thank the following people who made significant contributions to the content of this tutorial series:

- Alberto Poblacion, MVP & MCT, Spain
- Jarod Ferguson, Data Platform Development MVP, United States
- Harsh Mittal, Microsoft

- Kristina Olson, Microsoft
- Mike Pope, Microsoft
- Mohit Srivastava, Microsoft
- Raffaele Rialdi, Italy
- Rick Anderson, Microsoft
- Sayed Hashimi, Microsoft(twitter: @sayedihashimi)
- Scott Hanselman (twitter: @shanselman)
- Scott Hunter, Microsoft (twitter: @coolcsh)
- Srđan Božović, Serbia
- Vishal Joshi, Microsoft (twitter: @vishalrjoshi)

PREVIOUS NEXT

Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio or Visual Web Developer: Troubleshooting (12 of 12)

2/20/2020 • 18 minutes to read • Edit Online

by Tom Dykstra

Download Starter Project

> This series of tutorials shows you how to deploy (publish) an ASP.NET web application project that includes a SQL Server Compact database by using Visual Studio 2012 RC or Visual Studio Express 2012 RC for Web. You can also use Visual Studio 2010 if you install the Web Publish Update. For an introduction to the series, see the first tutorial in the series.
>
> For a tutorial that shows deployment features introduced after the RC release of Visual Studio 2012, shows how to deploy SQL Server editions other than SQL Server Compact, and shows how to deploy to Windows Azure Web Sites, see ASP.NET Web Deployment using Visual Studio.

This page describes some common problems that may arise when you deploy an ASP.NET web application by using Visual Studio. For each one, one or more possible causes and corresponding solutions are provided.

Server Error in '/' Application - Current Custom Error Settings Prevent Details of the Error from Being Viewed Remotely

Scenario

After deploying a site to a remote host, you get an error message that mentions the customErrors setting in the Web.config file but doesn't indicate what the actual cause of the error was:

```
Server Error in '/' Application.
Runtime Error

Description: An application error occurred on the server. The current custom error settings
for this application prevent the details of the application error from being viewed remotely
(for security reasons). It could, however, be viewed by browsers running on the local server
machine.

Details: To enable the details of this specific error message to be viewable on remote machines,
please create a <customErrors> tag within a "web.config" configuration file located in the
root directory of the current web application. This <customErrors> tag should then have its
"mode" attribute set to "Off".
```

Possible Cause and Solution

By default, ASP.NET shows detailed error information only when your web application is running on the local computer. Generally you don't want to display detailed error information when your web application is publicly available over the Internet, because hackers may be able to use this information to find vulnerabilities in the application. However, when you are deploying a site or updates to a site, sometimes something will go wrong and you need to get the actual error message.

To enable the application to display detailed error messages when it runs on the remote host, edit the Web.config file to set `customErrors` mode off, redeploy the application, and run the application again:

1. If the application Web.config file has a `customErrors` element in the `system.web` element, change the `mode` attribute to "off". Otherwise add a `customErrors` element in the `system.web` element with the `mode` attribute set to "off", as shown in the following example:

   ```
   <configuration>
     <system.web>
       <customErrors mode="off"/>
     </system.web>
   </configuration>
   ```

2. Deploy the application.

3. Run the application and repeat whatever you did earlier that caused the error to occur. Now you can see what the actual error message is.

4. When you have resolved the error, restore the original `customErrors` setting and redeploy the application.

Access is Denied in a Web Page that Uses SQL Server Compact

Scenario

When you deploy a site that uses SQL Server Compact and you run a page in the deployed site that accesses the database, you see the following error message:

```
Access is denied. (Exception from HRESULT: 0x80070005 (E_ACCESSDENIED))
```

Possible Cause and Solution

The NETWORK SERVICE account on the server needs to be able to read SQL Service Compact native binaries that are in the *bin\amd64* or *bin\x86* folder, but it does not have read permissions for those folders. Set read permission for NETWORK SERVICE on the *bin* folder, making sure to extend the permissions to subfolders.

Cannot Read Configuration File Due to Insufficient Permissions

Scenario

When you click the Visual Studio publish button to deploy an application to IIS on your local machine, publishing fails and the **Output** window shows an error message similar to this:

```
An error occurred when reading the IIS Configuration File 'MACHINE/REDIRECTION'.
The identity performing this operation was ... Error: Cannot read configuration file due to insufficient
permissions.
```

Possible Cause and Solution

To use one-click publish to IIS on your local machine, you must be running Visual Studio with administrator permissions. Close Visual Studio and restart it with administrator permissions.

Could Not Connect to the Destination Computer ... Using the Specified Process

Scenario

When you click the Visual Studio publish button to deploy an application, publishing fails and the **Output** window shows an error message similar to this:

```
Web deployment task failed.(Could not connect to the destination computer ("<server URL>") using the
specified process
("The Web Management Service"). This can happen if a proxy server is interrupting communication with the
destination server.
Disable the proxy server and try again.) ... The remote server returned an error: (502) Bad Gateway.
```

Possible Cause and Solution

A proxy server is interrupting communication with the destination server. From the Windows Control Panel or in Internet Explorer, select **Internet Options** and select the **Connections** tab. In the **Internet Properties** dialog box, click **LAN Settings**. In the **Local Area Network (LAN) Settings** dialog box, clear the **Automatically detect settings** checkbox. Then click the publish button again.

If the problem persists, contact your system administrator to determine what can be done with proxy or firewall settings. The problem happens because Web Deploy uses a non-standard port for Web Management Service deployment (8172); for other connections, Web Deploy uses port 80. When you are deploying to a third-party hosting provider, you are typically using the Web Management Service.

Default .NET 4.0 Application Pool Does Not Exist

Scenario

When you deploy an application that requires the .NET Framework 4, you see the following error message:

```
The default .NET 4.0 application pool does not exist or the application could not be added.
Please verify that ASP.NET 4.0 is installed on this machine.
```

Possible Cause and Solution

ASP.NET 4 is not installed in IIS. If the server you are deploying to is your development computer and has Visual Studio 2010 installed on it, ASP.NET 4 is installed on the computer but might not be installed in IIS. On the server that you are deploying to, open an elevated command prompt and install ASP.NET 4 in IIS by running the following commands:

```
cd %windir%\Microsoft.NET\Framework\v4.0.30319
aspnet_regiis.exe -iru
```

You might also need to manually set the .NET Framework version of the default application pool. For more information, see the Deploying to IIS as a Test Environment tutorial.

Format of the initialization string does not conform to specification starting at index 0.

Scenario

After you deploy an application using one-click publish, when you run a page that accesses the database you get the following error message:

```
Format of the initialization string does not conform to specification starting at index 0.
```

Possible Cause and Solution

Open the *Web.config* file in the deployed site and check to see whether the connection string values begin with `$(ReplaceableToken_`, as in the following example:

```
<connectionStrings>
  <add name="DefaultConnection" connectionString="$(ReplaceableToken_DefaultConnection-Web.config
Connection String_0)" providerName="System.Data.SqlServerCe.4.0" />
  <add name="SchoolContext" connectionString="$(ReplaceableToken_SchoolContext-Web.config Connection
String_0)" providerName="System.Data.SqlServerCe.4.0" />
</connectionStrings>
```

If the connection strings look like this example, edit the project file and add the following property to the `PropertyGroup` element that is for all build configurations:

```
<AutoParameterizationWebConfigConnectionStrings>False</AutoParameterizationWebConfigConnectionStrings>
```

Then redeploy the application.

HTTP 500 Internal Server Error

Scenario

When you run the deployed site, you see the following error message without specific information indicating the cause of the error:

```
HTTP Error 500 - Internal Server Error.
```

Possible Cause and Solution

There are many causes of 500 errors, but one possible cause if you are following these tutorials is that you put an XML element in the wrong place in one of the XML transformation files. For example, you would get this error if you put the transformation that inserts a `<location>` element under `<system.web>` instead of directly under `<configuration>`. The solution in that case is to correct the XML transformation file and redeploy.

HTTP 500.21 Internal Server Error

Scenario

When you run the deployed site, you see the following error message:

```
HTTP Error 500.21 - Internal Server Error.
Handler "PageHandlerFactory-Integrated" has a bad module "ManagedPipelineHandler" in its module list.
```

Possible Cause and Solution

The site you have deployed targets ASP.NET 4, but ASP.NET 4 is not registered in IIS on the server. On the server open an elevated command prompt and register ASP.NET 4 by running the following commands:

```
cd %windir%\Microsoft.NET\Framework\v4.0.30319
aspnet_regiis.exe –iru
```

You might also need to manually set the .NET Framework version of the default application pool. For more information, see the Deploying to IIS as a Test Environment tutorial.

Login Failed Opening SQL Server Express Database in App_Data

Scenario

You updated the *Web.config* file connection string to point to a SQL Server Express database as an *.mdf* file in

your *App_Data* folder, and the first time you run the application you see the following error message:

```
System.Data.SqlClient.SqlException: Cannot open database "DatabaseName" requested by the login. The login
failed.
```

Possible Cause and Solution

The name of the *.mdf* file cannot match the name of any SQL Server Express database that has ever existed on your computer, even if you deleted the *.mdf* file of the previously existing database. Change the name of the *.mdf* file to a name that has never been used as a database name and change the *Web.config* file to use the new name. As an alternative, you can use SQL Server Management Studio Express to delete previously existing SQL Server Express databases.

Model Compatibility Cannot be Checked

Scenario

You updated the *Web.config* file connection string to point to a new SQL Server Express database, and the first time you run the application you see the following error message:

```
Model compatibility cannot be checked because the database does not contain model metadata.
Ensure that IncludeMetadataConvention has been added to the DbModelBuilder conventions.
```

Possible Cause and Solution

If the database name you put in the Web.config file was ever used before on your computer, a database might already exist with some tables in it. Select a new name that has not been used on your computer before and change the *Web.config* file to point to use this new database name. As an alternative, you can use SQL Server Express Utility or SQL Server Management Studio Express to delete the existing database.

SQL Error When a Script Attempts to Create Users or Roles

Scenario

You are using database deployment configured on the **Package/Publish SQL** tab, SQL scripts that run during deployment include Create User or Create Role commands, and script execution fails when those commands are executed. You might see more detailed messages, such as the following:

```
The approximate location of the error was between lines '1' and '3' of the script.
The verbose log may have more information about the error. The command started with:
CREATE USER [user2] FOR LOGIN [user2] WITH DEFAULT
Error: User does not have permission to perform this action.
```

If this error occurs when you have configured database deployment in the **Publish Web** wizard rather than the **Package/Publish SQL** tab, create a thread in the Configuration and Deployment forum, and the solution will be added to this troubleshooting page.

Possible Cause and Solution

The user account you are using to perform deployment does not have permission to create users or roles. For example, the hosting company might assign the `db_datareader`, `db_datawriter`, and `db_ddladmin` roles to the user account that it sets up for you. These are sufficient for creating most database objects, but not for creating users or roles. One way to avoid the error is by excluding users and roles from database deployment. You can do this by editing the `PreSource` element for the database's automatically generated script so that it includes the following attributes:

```
CopyAllUsers=false, CopyAllRoles=false
```

For information about how to edit the `PreSource` element in the project file, see How to: Edit Deployment Settings in the Project File. If the users or roles in your development database need to be in the destination database, contact your hosting provider for assistance.

SQL Server Timeout Error When Running Custom Scripts During Deployment

Scenario

You have specified custom SQL scripts to run during deployment, and when Web Deploy runs them, they time out.

Possible Cause and Solution

Running multiple scripts that have different transaction modes can cause time-out errors. By default, automatically generated scripts run in a transaction, but custom scripts do not. If you select the **Pull data and/or schema from an existing database** option on the **Package/Publish SQL** tab, and if you add a custom SQL script, you must change transaction settings on some scripts so that all scripts use the same transaction settings. For more information, see How to: Deploy a Database With a Web Application Project.

If you have configured transaction settings so that all are the same but still get this error, a possible workaround is to run the scripts separately. In the **Database Scripts** grid in the **Package/Publish** SQL tab, clear the **Include** check box for the script that causes the timeout error, then publish the project. Then go back into the **Database Scripts** grid, select that script's **Include** check box, and clear the **Include** check boxes for the other scripts. Then publish the project again. This time when you publish, only the selected custom script runs.

Stream Data of Site Manifest Is Not Yet Available

Scenario

When you are installing a package using the *deploy.cmd* file with the `t` (test) option, you see the following error message:

```
Error: The stream data of 'sitemanifest/dbFullSql[@path='C:\TEMP\AdventureWorksGrant.sql']/sqlScript' is
not yet available.
```

Possible Cause and Solution

The error message means that the command cannot produce a test report. However, the command might run if you use the `y` (actual installation) option. The message indicates only that there is a problem with running the command in test mode.

This Application Requires ManagedRuntimeVersion v4.0

Scenario

When you attempt to deploy, you see the following error message:

Error: The stream data of 'sitemanifest/dbFullSql[@path='C:\TEMP\AdventureWorksGrant.sql']/sqlScript' is not yet available. The application pool that you are trying to use has the 'managedRuntimeVersion' property set to 'v2.0'. This application requires 'v4.0'.

Possible Cause and Solution

ASP.NET 4 is not installed in IIS. If the server you are deploying to is your development computer and has Visual Studio 2010 installed on it, ASP.NET 4 is installed on the computer but might not be installed in IIS. On the server that you are deploying to, open an elevated command prompt and install ASP.NET 4 in IIS by running the following commands:

```
cd %windir%\Microsoft.NET\Framework\v4.0.30319
aspnet_regiis.exe -i
```

Unable to cast Microsoft.Web.Deployment.DeploymentProviderOptions

Scenario

When you are deploying a package, you see the following error message:

```
Unable to cast object of type 'Microsoft.Web.Deployment.DeploymentProviderOptions' to
'Microsoft.Web.Deployment.DeploymentProviderOptions'.
```

Possible Cause and Solution

You are trying to deploy from IIS Manager using the Web Deploy 1.1 UI to a server that has Web Deploy 2.0 installed. If you are using the IIS Remote Administration Tool to deploy by importing a package, check the **New Features Available** dialog box when you establish the connection. (This dialog box might only be shown once when the connection is first established. To clear the connection and start over, close IIS Manager and start it up again by entering `inetmgr /reset` at the command prompt.) If one of the features listed is **Web Deploy UI**, and it has a version number lower than 8, the server you are deploying to might have both 1.1 and 2.0 versions of Web Deploy installed. To deploy from a client that has 2.0 installed, the server must have only Web Deploy 2.0 installed. You will have to contact your hosting provider to resolve this problem.

Unable to load the native components of SQL Server Compact

Scenario

When you run the deployed site, you see the following error message:

```
Unable to load the native components of SQL Server Compact corresponding to the ADO.NET provider of version
8482.
Install the correct version of SQL Server Compact. Refer to KB article 974247 for more details.
```

Possible Cause and Solution

The deployed site does not have *amd64* and *x86* subfolders with the native assemblies in them under the application's *bin* folder. On a computer that has SQL Server Compact installed, the native assemblies are located in *C:\Program Files\Microsoft SQL Server Compact Edition\v4.0\Private*. The best way to get the correct files into the correct folders in a Visual Studio project is to install the NuGet SqlServerCompact package. Package installation adds a post-build script to copy the native assemblies into *amd64* and *x86*. In order for these to be deployed, however, you have to manually include them in the project. For more information, see the Deploying SQL Server Compact tutorial.

"Path is not valid" error after deploying an Entity Framework Code First application

Scenario

You deploy an application that uses Entity Framework Code First Migrations and a DBMS such as SQL Server

Compact which stores its database in a file in the App_Data folder. You have Code First Migrations configured to create the database after your first deployment. When you run the application you get an error message like the following example:

```
The path is not valid. Check the directory for the database. [Path =
c:\inetpub\wwwroot\App_Data\DatabaseName.sdf ]
```

Possible Cause and Solution

Code First is attempting to create the database but the App_Data folder does not exist. Either you didn't have any files in the *App_Data* folder when you deployed, or you selected **Exclude App_Data** on the **Package/Publish Web** tab of the **Project Properties** window. The deployment process won't create a folder on the server if there are no files in the folder to be copied to the server. If you already had the database set up in the site, the deployment process will delete the files and the *App_Data* folder itself if you selected **Remove additional files at destination** in the publish profile. To solve the problem, put a placeholder file such as a .txt file in the *App_Data* folder, make sure you do not have **Exclude App_Data** selected, and redeploy.

"COM object that has been separated from its underlying RCW cannot be used."

Scenario

You have been successfully using one-click publish to deploy your application and then you start getting this error:

```
Web deployment task failed. (Could not complete the request to remote agent URL
'https://serverurl.com/msdeploy.axd?site=sitename'.)
Could not complete the request to remote agent URL 'https://url/msdeploy.axd?site=sitename'.
The request was aborted: The request was canceled.
COM object that has been separated from its underlying RCW cannot be used.
```

Possible Cause and Solution

Closing and restarting Visual Studio is usually all that is required to resolve this error.

Deployment Fails Because User Credentials Used for Publishing Don't Have setACL Authority

Scenario

Publishing fails with an error that indicates you don't have authority to set folder permissions (the user account you are using doesn't have setACL authority).

Possible Cause and Solution

By default, Visual Studio sets read permissions on the root folder of the site and write permissions on the App_Data folder. If you know that the default permissions on site folders are correct and do not need to be set, you disable this behavior by adding **<IncludeSetACLProviderOnDestination>False</IncludeSetACLProviderOnDestination>** to the publish profile file (to affect a single profile) or to the wpp.targets file (to affect all profiles). For information about how to edit these files, see How to: Edit Deployment Settings in Profile (.pubxml) Files.

Access Denied Errors when the Application Tries to Write to an Application Folder

Scenario

Your application errors when it tries to create or edit a file in one of the application folders, because it does not have write authority for that folder.

Possible Cause and Solution

By default, Visual Studio sets read permissions on the root folder of the site and write permissions on the App_Data folder. If your application needs write access to a sub-folder, you can set permissions for that folder as shown in the Setting Folder Permissions and Deploying to the Production Environment tutorials. If your application needs write access to the root folder of the site, you have to prevent it from setting read-only access on the root folder by adding **<IncludeSetACLProviderOnDestination>False</IncludeSetACLProviderOnDestination>** to the publish profile file (to affect a single profile) or to the wpp.targets file (to affect all profiles). For information about how to edit these files, see How to: Edit Deployment Settings in Profile (.pubxml) Files.

Configuration Error - targetFramework attribute references a version that is later than the installed version of the .NET Framework

Scenario

You successfully published a web project that targets ASP.NET 4.5, but when you run the application (with the `customErrors` mode set to "off" in the Web.config file) you get the following error:

```
The 'targetFramework' attribute in the <compilation> element of the Web.config
file is used only to target version 4.0 and later of the .NET Framework (for
example, '<compilation targetFramework="4.0">'). The 'targetFramework' attribute
currently references a version that is later than the installed version of the
.NET Framework. Specify a valid target version of the .NET Framework, or install
the required version of the .NET Framework.
```

The Source Error box of the error page highlights the following line from Web.config as the cause of the error:

```
<compilation targetFramework="4.5" />
```

Possible Cause and Solution

The server does not support ASP.NET 4.5. Contact the hosting provider to determine when and if support for ASP.NET 4.5 can be added. If upgrading the server is not an option, you have to deploy a web project that targets ASP.NET 4 or earlier instead.If you deploy an ASP.NET 4 or earlier web project to the same destination, select the **Remove additional files at destination** check box on the **Settings** tab of the **Publish Web** wizard. If you don't select **Remove additional files at destination**, you will continue to get the Configuration Error page.

The project **Properties** windows includes a Target framework drop-down list, but you can't resolve this problem by just changing that from **.NET Framework 4.5** to **.NET Framework 4**. If you change the target framework to an earlier framework version, the project will still have references to the later framework version's assemblies and will not run. You have to manually change those references or create a new project that targets .NET Framework 4 or earlier. For more information, see .NET Framework Targeting for Web Sites.

PREVIOUS

ASP.NET 3.5 - AJAX

2/20/2020 • 2 minutes to read • Edit Online

About the ASP.NET AJAX library, including UpdatePanels and partial page updates, authentication, localization, and debugging.

- Understanding Partial Page Updates with ASP.NET AJAX
- Understanding ASP.NET AJAX UpdatePanel Triggers
- Understanding ASP.NET AJAX Authentication and Profile Application Services
- Understanding ASP.NET AJAX Localization
- Understanding ASP.NET AJAX Web Services
- Understanding ASP.NET AJAX Debugging Capabilities

Understanding Partial Page Updates with ASP.NET AJAX

2/20/2020 • 21 minutes to read • Edit Online

by Scott Cate

Download PDF

Perhaps the most visible feature of the ASP.NET AJAX Extensions is the ability to do a partial or incremental page updates without doing a full postback to the server, with no code changes and minimal markup changes. The advantages are extensive – the state of your multimedia (such as Adobe Flash or Windows Media) is unchanged, bandwidth costs are reduced, and the client does not experience the flicker usually associated with a postback.

Introduction

Microsoft's ASP.NET technology brings an object-oriented and event-driven programming model and unites it with the benefits of compiled code. However, its server-side processing model has several drawbacks inherent in the technology:

- Page updates require a round-trip to the server, which requires a page refresh.
- Round-trips do not persist any effects generated by Javascript or other client-side technology (such as Adobe Flash)
- During postback, browsers other than Microsoft Internet Explorer do not support automatically restoring the scroll position. And even in Internet Explorer, there is still a flicker as the page is refreshed.
- Postbacks may involve a high amount of bandwidth as the __VIEWSTATE form field may grow, especially when dealing with controls such as the GridView control or repeaters.
- There is no unified model for accessing Web Services through JavaScript or other client-side technology.

Enter Microsoft's ASP.NET AJAX extensions. AJAX, which stands for **A** synchronous **J** avaScript **A** nd **X** ML, is an integrated framework for providing incremental page updates via cross-platform JavaScript, composed of server-side code comprising the Microsoft AJAX Framework, and a script component called the Microsoft AJAX Script Library. The ASP.NET AJAX extensions also provide cross-platform support for accessing ASP.NET Web Services via JavaScript.

This whitepaper examines the partial page updates functionality of the ASP.NET AJAX Extensions, which includes the ScriptManager component, the UpdatePanel control, and the UpdateProgress control, and considers scenarios in which they should or should not be utilized.

This whitepaper is based on the Beta 2 release of the Visual Studio 2008 and the .NET Framework 3.5, which integrates the ASP.NET AJAX Extensions into the Base Class Library (where it was previously an add-on component available for ASP.NET 2.0). This whitepaper also assumes that you are using Visual Studio 2008 and not Visual Web Developer Express Edition; some project templates that are referenced may not be available to Visual Web Developer Express users.

Partial Page Updates

Perhaps the most visible feature of the ASP.NET AJAX Extensions is the ability to do a partial or incremental page updates without doing a full postback to the server, with no code changes and minimal markup changes. The advantages are extensive - the state of your multimedia (such as Adobe Flash or Windows Media) is unchanged,

bandwidth costs are reduced, and the client does not experience the flicker usually associated with a postback.

The ability to integrate partial page rendering is integrated into ASP.NET with minimal changes into your project.

Walkthrough: Integrating Partial Rendering into an Existing Project

1. In Microsoft Visual Studio 2008, create a new ASP.NET Web Site project by going to *File-> New-> Web Site* and selecting ASP.NET Web Site from the dialog. You can name it whatever you like, and you may install it either to the file system or into Internet Information Services (IIS).
2. You will be presented with the blank default page with basic ASP.NET markup (a server-side form and an `@Page` directive). Drop a Label called `Label1` and a Button called `Button1` onto the page within the form element. You may set their text properties to whatever you like.
3. In Design view, double-click `Button1` to generate a code-behind event handler. Within this event handler, set `Label1.Text` to You clicked the button! .

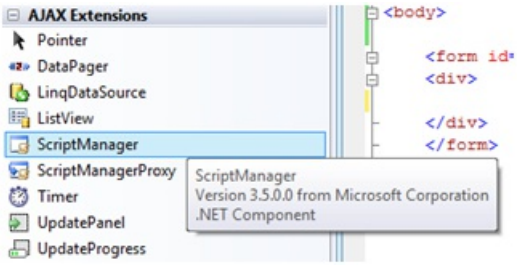
Listing 1: Markup for default.aspx before partial rendering is enabled

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
 <head runat="server">
 <title>Untitled Page</title>
 </head>
 <body>
 <form id="form1" runat="server">
 <div>
 <asp:Label ID="Label1" runat="server" Text="This is a label!"></asp:Label>
 <asp:Button ID="Button1" runat="server" Text="Click Me" OnClick="Button1_Click" />
 </div>
 </form>
 </body>
</html>
```

Listing 2: Codebehind (trimmed) in default.aspx.cs

```
public partial class _Default : System.Web.UI.Page
{
    protected void Button1_Click(object sender, EventArgs e)
    {
        Label1.Text = "You clicked the button!";
    }
}
```

1. Press F5 to launch your web site. Visual Studio will prompt you to add a web.config file to enable debugging; do so. When you click the button, notice that the page refreshes to change the text in the label, and there is a brief flicker as the page is redrawn.
2. After closing your browser window, return to Visual Studio and to the markup page. Scroll down in the Visual Studio toolbox, and find the tab labeled AJAX Extensions. (If you do not have this tab because you are using an older version of AJAX or Atlas extensions, refer to the walkthrough for registering the AJAX Extensions toolbox items later in this whitepaper, or install the current version with the Windows Installer downloadable from the website).



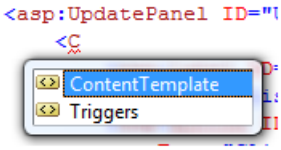
(Click to view full-size image)

1. *Known Issue*:If you install Visual Studio 2008 onto a computer that already has Visual Studio 2005 installed with the ASP.NET 2.0 AJAX Extensions, Visual Studio 2008 will import the AJAX Extensions toolbox items. You can determine whether this is the case by examining the tooltip of the components; they should say Version 3.5.0.0. If they say Version 2.0.0.0, then you have imported your old toolbox items, and will need to manually import them by using the Choose Toolbox Items dialog in Visual Studio. You will be unable to add Version 2 controls via the designer.

2. Before the `<asp:Label>` tag begins, create a line of whitespace, and double-click on the UpdatePanel control in the toolbox. Note that a new `@Register` directive is included at the top of the page, indicating that controls within the System.Web.UI namespace should be imported using the `asp:` prefix.

3. Drag the closing `</asp:UpdatePanel>` tag past the end of the Button element, so that the element is well-formed with the Label and Button controls wrapped.

4. After the opening `<asp:UpdatePanel>` tag, begin opening a new tag. Note that IntelliSense prompts you with two options. In this case, create a `<ContentTemplate>` tag. Be sure to wrap this tag around your Label and Button so that the markup is well-formed.



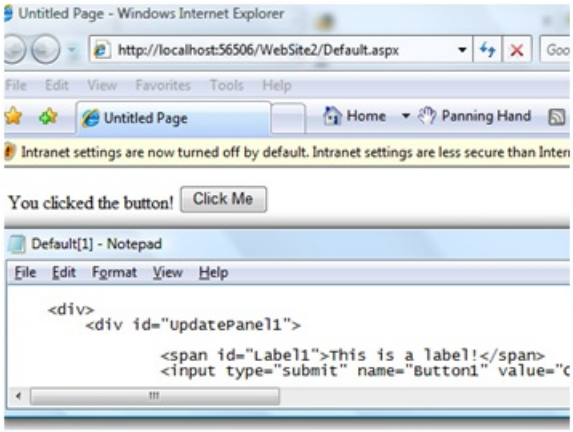
(Click to view full-size image)

1. Anywhere within the `<form>` element, include a ScriptManager control by double-clicking on the `ScriptManager` item in the toolbox.
2. Edit the `<asp:ScriptManager>` tag so that it includes the attribute `EnablePartialRendering= true`.

Listing 3: Markup for default.aspx with partial rendering enabled

```
<%@ Page Language="C#" AutoEventWireup="true"
 CodeFile="Default.aspx.cs" Inherits="_Default" %>
<%@ Register Assembly="System.Web.Extensions, Version=1.0.61025.0, Culture=neutral,
 PublicKeyToken=31bf3856ad364e35"
 Namespace="System.Web.UI" TagPrefix="asp" %>
<!DOCTYPE html PUBLIC
 "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
 <head runat="server">
 <title>Untitled Page</title>
 </head>
 <body>
 <form id="form1" runat="server">
 <asp:ScriptManager EnablePartialRendering="true"
 ID="ScriptManager1" runat="server"></asp:ScriptManager>
 <div>
 <asp:UpdatePanel ID="UpdatePanel1" runat="server">
 <ContentTemplate>
 <asp:Label ID="Label1" runat="server"
 Text="This is a label!"></asp:Label>
 <asp:Button ID="Button1" runat="server"
 Text="Click Me" OnClick="Button1_Click" />
 </ContentTemplate>
 </asp:UpdatePanel>
 </div>
 </form>
 </body>
</html>
```

1. Open your web.config file. Notice that Visual Studio has automatically added a compilation reference to System.Web.Extensions.dll.
2. What's New in Visual Studio 2008: The web.config that comes with the ASP.NET Web Site project templates automatically includes all necessary references to the ASP.NET AJAX Extensions, and includes commented sections of configuration information that can be un-commented to enable additional functionality. Visual Studio 2005 had similar templates when ASP.NET 2.0 AJAX Extensions were installed. However, in Visual Studio 2008, the AJAX Extensions are opt-out by default (that is, they are referenced by default, but can be removed as references).



(Click to view full-size image)

1. Press F5 to launch your website. Note how no source code changes were required to support partial rendering - only markup was changed.

When you launch your website, you should see that partial rendering is now enabled, because when you click on the button there will be no flicker, nor will there be any change in the page scroll position (this example does not demonstrate that). If you were to look at the rendered source of the page after clicking the button, it will confirm

that in fact a post-back has not occurred - the original label text is still part of the source markup, and the label has changed through JavaScript.

Visual Studio 2008 does not appear to come with a pre-defined template for an ASP.NET AJAX-Enabled web site. However, such a template was available within Visual Studio 2005 if the Visual Studio 2005 and ASP.NET 2.0 AJAX Extensions were installed. Consequently, configuring a web site and starting with the AJAX-Enabled Web Site template will likely be even easier, as the template should include a fully-configured web.config file (supporting all of the ASP.NET AJAX Extensions, including Web Services access and JSON serialization - JavaScript Object Notation) and includes an UpdatePanel and ContentTemplate within the main Web Forms page by default. Enabling partial rendering with this default page is as simple as revisiting Step 10 of this walkthrough and dropping controls onto the page.

The ScriptManager Control

ScriptManager Control Reference

Markup-Enabled Properties:

PROPERTY NAME	TYPE	DESCRIPTION
AllowCustomErrors-Redirect	Bool	Specifies whether to use the custom error section of the web.config file to handle errors.
AsyncPostBackError-Message	String	Gets or sets the error message sent to the client if an error is raised.
AsyncPostBack-Timeout	Int32	Gets or sets the default amount of a time a client should wait for the asynchronous request to complete.
EnableScript-Globalization	Bool	Gets or sets whether script globalization is enabled.
EnableScript-Localization	Bool	Gets or sets whether script localization is enabled.
ScriptLoadTimeout	Int32	Determines the number of seconds allowed for loading scripts into the client
ScriptMode	Enum (Auto, Debug, Release, Inherit)	Gets or sets whether to render release versions of scripts
ScriptPath	String	Gets or sets the root path to the location of script files to be sent to the client.

Code-Only Properties:

PROPERTY NAME	TYPE	DESCRIPTION
AuthenticationService	AuthenticationService-Manager	Gets details about the ASP.NET Authentication Service proxy that will be sent to the client.

PROPERTY NAME	TYPE	DESCRIPTION
IsDebuggingEnabled	Bool	Gets whether script and code debugging is enabled.
IsInAsyncPostback	Bool	Gets whether the page is currently in an asynchronous post back request.
ProfileService	ProfileService-Manager	Gets details about the ASP.NET Profiling Service proxy that will be sent to the client.
Scripts	Collection<Script-Reference>	Gets a collection of script references that will be sent to the client.
Services	Collection<Service-Reference>	Gets a collection of Web Service proxy references that will be sent to the client.
SupportsPartialRendering	Bool	Gets whether the current client supports partial rendering. If this property returns **false**, then all page requests will be standard postbacks.

Public Code Methods:

METHOD NAME	TYPE	DESCRIPTION
SetFocus(string)	Void	Sets the focus of the client to a particular control when the request has completed.

Markup Descendants:

TAG	DESCRIPTION
<AuthenticationService>	Provides details about the proxy to the ASP.NET authentication service.
<ProfileService>	Provides details about the proxy to the ASP.NET profiling service.
<Scripts>	Provides additional script references.
<asp:ScriptReference>	Denotes a specific script reference.
<Service>	Provides additional Web Service references which will have proxy classes generated.
<asp:ServiceReference>	Denotes a specific Web Service reference.

The ScriptManager control is the essential core for the ASP.NET AJAX Extensions. It provides access to the script library (including the extensive client-side script type system), supports partial rendering, and provides extensive support for additional ASP.NET services (such as authentication and profiling, but also other Web Services). The ScriptManager control also provides globalization and localization support for the client scripts.

Providing Alternative and Supplemental Scripts

While the Microsoft ASP.NET 2.0 AJAX Extensions include the entire script code in both debug and release editions as resources embedded in the referenced assemblies, developers are free to redirect the ScriptManager to customized script files, as well as register additional necessary scripts.

To override the default binding for the typically-included scripts (such as those which support the Sys.WebForms namespace and the custom typing system), you can register for the `ResolveScriptReference` event of the ScriptManager class. When this method is called, the event handler has the opportunity to change the path to the script file in question; the script manager will then send a different or customized copy of the scripts to the client.

Additionally, script references (represented by the `ScriptReference` class) can be included programmatically or via markup. To do so, either programmatically modify the `ScriptManager.Scripts` collection, or include `<asp:ScriptReference>` tags under the `<Scripts>` tag, which is a first-level child of the ScriptManager control.

Custom Error Handling for UpdatePanels

Although updates are handled by triggers specified by UpdatePanel controls, the support for error handling and custom error messages is handled by a page's ScriptManager control instance. This is done by exposing an event, `AsyncPostBackError`, to the page which can then provide custom exception-handling logic.

By consuming the AsyncPostBackError event, you may specify the `AsyncPostBackErrorMessage` property, which then causes an alert box to be raised upon completion of the callback.

Client-side customization is also possible instead of using the default alert box; for instance, you may want to display a customized `<div>` element rather than the default browser modal dialog. In this case, you can handle the error in client script:

Listing 5: Client-side script to display custom errors

```
<script type= text/javascript >
 <!--
 Sys.WebForms.PageRequestManager.getInstance().add_EndRequest(Request_End);
 function Request_End(sender, args)
 {
 if (args.get_error() != undefined)
 {
 var errorMessage =  ;
 if (args.get_response().get_statusCode() ==  200 )
 {
 errorMessage = args.get_error().message;
 }
 else
 {
 // the server wasn't the problem...
 errorMessage =  An unknown error occurred... ;
 }
 // do something with the errorMessage here.
 // now make sure the system knows we handled the error.
 args.set_errorHandled(true);
 }
 }
 // -->
</script>
```

Quite simply, the above script registers a callback with the client-side AJAX runtime for when the asynchronous request has been completed. It then checks to see whether an error was reported, and if so, processes the details of it, finally indicating to the runtime that the error was handled in custom script.

Globalization and Localization Support

The ScriptManager control provides extensive support for localization of script strings and user interface components; however, that topic is outside of the scope of this whitepaper. For more information, see the whitepaper, Globalization Support in ASP.NET AJAX Extensions.

The UpdatePanel Control

UpdatePanel Control Reference

Markup-Enabled Properties:

PROPERTY NAME	TYPE	DESCRIPTION
ChildrenAsTriggers	bool	Specifies whether child controls automatically invoke refresh on postback.
RenderMode	enum (Block, Inline)	Specifies the way the content will be visually presented.
UpdateMode	enum (Always, Conditional)	Specifies whether the UpdatePanel is always refreshed during a partial render or if it is only refreshed when a trigger is hit.

Code-Only Properties:

PROPERTY NAME	TYPE	DESCRIPTION
IsInPartialRendering	bool	Gets whether the UpdatePanel is supporting partial rendering for the current request.
ContentTemplate	ITemplate	Gets the markup template for the update request.
ContentTemplateContainer	Control	Gets the programmatic template for the update request.
Triggers	UpdatePanel- TriggerCollection	Gets the list of triggers associated with the current UpdatePanel.

Public Code Methods:

METHOD NAME	TYPE	DESCRIPTION
Update()	Void	Updates the specified UpdatePanel programmatically. Allows a server request to trigger a partial render of an otherwise-untriggered UpdatePanel.

Markup Descendants:

TAG	DESCRIPTION
<ContentTemplate>	Specifies the markup to be used to render the partial rendering result. Child of <asp:UpdatePanel>.
<Triggers>	Specifies a collection of *n* controls associated with updating this UpdatePanel. Child of <asp:UpdatePanel>.
<asp:AsyncPostBackTrigger>	Specifies a trigger that invokes a partial page render for the given UpdatePanel. This may or may not be a control as a descendant of the UpdatePanel in question. Granular to the event name.Child of <Triggers>.
<asp:PostBackTrigger>	Specifies a control that causes the entire page to refresh. This may or may not be a control as a descendant of the UpdatePanel in question. Granular to the object. Child of <Triggers>.

The `UpdatePanel` control is the control that delimits the server-side content that will take part in the partial rendering functionality of the AJAX Extensions. There is no limit to the number of UpdatePanel controls that can be on a page, and they can be nested. Each UpdatePanel is isolated, so that each can work independently (you can have two UpdatePanels running at the same time, rendering different parts of the page, independent of the page's postback).

The UpdatePanel control primarily deals with control triggers - by default, any control contained within an UpdatePanel's `ContentTemplate` that creates a postback is registered as a trigger for the UpdatePanel. This means that the UpdatePanel can work with the default data-bound controls (such as the GridView), with user controls, and they can be programmed in script.

By default, when a partial page render is triggered, all UpdatePanel controls on the page will be refreshed, whether or not the UpdatePanel controls defined triggers for such action. For example, if one UpdatePanel defines a Button control, and that Button control is clicked, all UpdatePanel controls on that page will be refreshed by default. This is because, by default, the `UpdateMode` property of the UpdatePanel is set to `Always`. Alternatively, you may set the UpdateMode property to `Conditional`, which means that the UpdatePanel will only be refreshed if a specific trigger is hit.

Custom Control Notes

An UpdatePanel can be added to any user control or custom control; however, the page on which these controls are included must also include a ScriptManager control with the property EnablePartialRendering set to **true**.

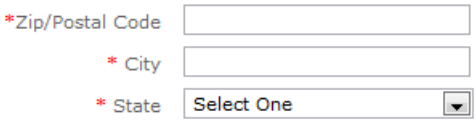
One way in which you might account for this when using Web Custom Controls is to override the protected `CreateChildControls()` method of the `CompositeControl` class. By doing so, you can inject an UpdatePanel between the control's children and the outside world if you determine the page supports partial rendering; otherwise, you can simply layer the child controls into a container `Control` instance.

UpdatePanel Considerations

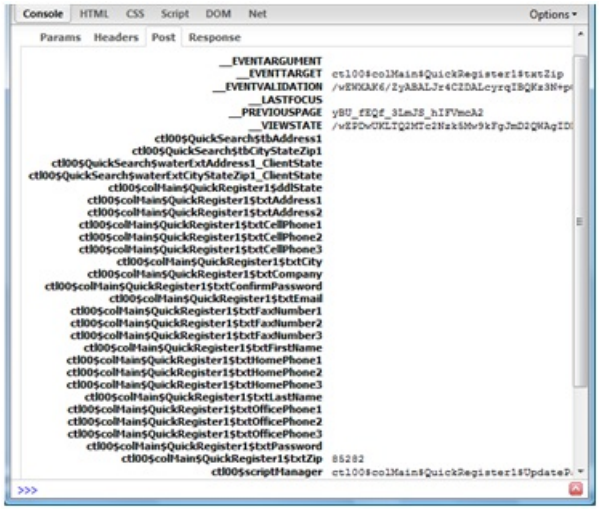
The UpdatePanel operates as something of a black-box, wrapping ASP.NET postbacks within the context of a JavaScript XMLHttpRequest. However, there are significant performance considerations to bear in mind, both in terms of behavior and speed. To understand how the UpdatePanel works, so that you can best decide when its use is appropriate, you should examine the AJAX exchange. The following example uses an existing site and, Mozilla Firefox with the Firebug extension (Firebug captures XMLHttpRequest data).

Consider a form that, among other things, has a postal code textbox which is supposed to populate a city and

state field on a form or control. This form ultimately collects membership information, including a user's name, address, and contact information. There are many design considerations to take into account, based on the requirements of a specific project.



(Click to view full-size image)



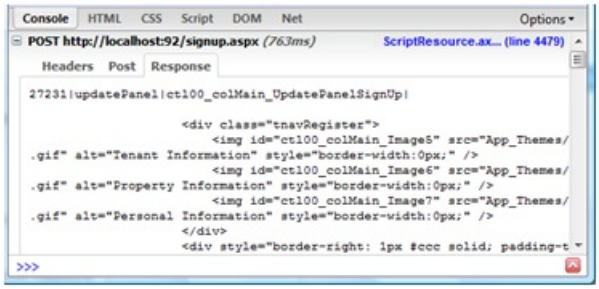
(Click to view full-size image)

In the original iteration of this application, a control was built that incorporated the entirety of the user registration data, including the postal code, city, and state. The entire control was wrapped within an UpdatePanel and dropped onto a Web Form. When the postal code is entered by the user, the UpdatePanel detects the event (the corresponding TextChanged event in the back-end, either by specifying triggers or by using the ChildrenAsTriggers property set to true). AJAX posts all of the fields within the UpdatePanel, as captured by FireBug (see the diagram on the right).

As the screen capture indicates, values from every control within the UpdatePanel are delivered (in this case, they are all empty), as well as the ViewState field. All told, over 9kb of data is sent, when in fact only five bytes of data were needed to make this particular request. The response is even more bloated: in total, 57kb is sent to the client, simply to update a text field and a drop-down field.

It may also be of interest to see how ASP.NET AJAX updates the presentation. The response portion of the UpdatePanel's update request is shown in the Firebug console display on the left; it is a specially-formulated pipe-delimited string that is broken up by the client script and then reassembled on the page. Specifically, ASP.NET AJAX sets the *innerHTML* property of the HTML element on the client that represents your UpdatePanel. As the browser re-generates the DOM, there is a slight delay, depending on the amount of information that needs to be processed.

The regeneration of the DOM triggers a number of additional issues:



(Click to view full-size image)

- If the focused HTML element is within the UpdatePanel, it will lose focus. So, for users who pressed the Tab key to exit the postal code text box, their next destination would have been the City text box. However, once the UpdatePanel refreshed the display, the form would no longer have had focus, and pressing Tab would have started highlighting the focus elements (such as links).
- If any type of custom client-side script is in use that accesses DOM elements, references persisted by functions may become defunct after a partial postback.

UpdatePanels are not intended to be catch-all solutions. Rather, they provide a quick solution for certain situations, including prototyping, small control updates, and provide a familiar interface to ASP.NET developers who might be familiar with the .NET object model but less-so with the DOM. There are a number of alternatives that may result in better performance, depending on the application scenario:

- Consider using PageMethods and JSON (JavaScript Object Notation) allows the developer to invoke static methods on a page as if a web service call was being invoked. Because the methods are static, no state is necessary; the script caller supplies the parameters, and the result is returned asynchronously.
- Consider using a Web Service and JSON if a single control needs to be used in several places across an application. This again requires very little special work, and works asynchronously.

Incorporating functionality through Web Services or Page Methods has drawbacks as well. First and foremost, ASP.NET developers typically tend to build small components of functionality into user controls (.ascx files). Page methods cannot be hosted in these files; they must be hosted within the actual .aspx page class. Web services, similarly, must be hosted within the .asmx class. Depending on the application, this architecture may violate the Single Responsibility Principle, in that the functionality for a single component is now spread across two or more physical components which may have little or no cohesive ties.

Finally, if an application requires that UpdatePanels are used, the following guidelines should assist with troubleshooting and maintenance.

- **Nest UpdatePanels as little as possible, not only within-units, but also across units of code.** For example, having an UpdatePanel on a Page that wraps a Control, while that Control also contains an UpdatePanel, which contains another Control that contains an UpdatePanel, is cross-unit nesting. This helps to keep clear which elements should be refreshing, and prevents unexpected refreshes to child UpdatePanels.
- **Keep the *ChildrenAsTriggers* property set to false, and explicitly set triggering events.** Utilizing the `<Triggers>` collection is a much clearer way to handle events, and may prevent unexpected behavior, helping with maintenance tasks and forcing a developer to opt-in for an event.
- **Use the smallest possible unit to achieve functionality.** As noted in the discussion of the postal code service, wrapping only the bare minimum reduces time to the server, total processing, and the footprint of the client-server exchange, enhancing performance.

The UpdateProgress Control

UpdateProgress Control Reference

Markup-Enabled Properties:

PROPERTY NAME	TYPE	DESCRIPTION
AssociatedUpdate-PanelID	String	Specifies the ID of the UpdatePanel that this UpdateProgress should report on.
DisplayAfter	Int	Specifies the timeout in milliseconds before this control is displayed after the asynchronous request begins.

PROPERTY NAME	TYPE	DESCRIPTION
DynamicLayout	bool	Specifies whether the progress is rendered dynamically.

Markup Descendants:

TAG	DESCRIPTION
<ProgressTemplate>	Contains the control template set for the content that will be displayed with this control.

The UpdateProgress control gives you a measure of feedback to keep your users' interest while doing the necessary work to transport to the server. This can help your users know that you're doing something even though it may not be apparent, especially since most users are used to the page refreshing and seeing the status bar highlight.

As a note, UpdateProgress controls can appear anywhere on a page hierarchy. However, in cases in which a partial postback is initiated from a child UpdatePanel (where an UpdatePanel is nested within another UpdatePanel), postbacks that trigger the child UpdatePanel will cause UpdateProgress templates to be displayed for the child UpdatePanel as well as the parent UpdatePanel. But, if the trigger is a direct child of the parent UpdatePanel, then only the UpdateProgress templates associated with the parent will be displayed.

Summary

The Microsoft ASP.NET AJAX extensions are sophisticated products designed to assist in making web content more accessible and to provide a richer user experience to your web applications. As part of the ASP.NET AJAX Extensions, the partial page rendering controls, including the ScriptManager, the UpdatePanel, and UpdateProgress controls are some of the most visible components of the toolkit.

The ScriptManager component integrates the provision of client JavaScript for the Extensions, as well as enables the various server- and client-side components to work together with minimal development investment.

The UpdatePanel control is the apparent magic box - markup within the UpdatePanel can have server-side Codebehind and not trigger a page refresh. UpdatePanel controls can be nested, and can be dependent on controls in other UpdatePanels. By default, UpdatePanels handle any postbacks invoked by their descendant controls, although this functionality can be finely tuned, either declaratively or programmatically.

When using the UpdatePanel control, developers should be aware of the performance impact that could potentially arise. Potential alternatives include web services and page methods, though the design of the application should be considered.

The UpdateProgress control allows the user to know that she or he is not being ignored, and that the behind-the-scenes request is going on while the page is otherwise not doing anything to respond to the user input. It also includes the ability to abort partial rendering results.

Together, these tools assist creating a rich and seamless user experience by making server work less apparent to the user and interrupting workflow less.

Bio

Scott Cate has been working with Microsoft Web technologies since 1997 and is the President of myKB.com (www.myKB.com) where he specializes in writing ASP.NET based applications focused on Knowledge Base Software solutions. Scott can be contacted via email at scott.cate@myKB.com or his blog at ScottCate.com

NEXT

Understanding ASP.NET AJAX UpdatePanel Triggers

2/20/2020 • 12 minutes to read • Edit Online

by Scott Cate

Download PDF

> When working in the markup editor in Visual Studio, you may notice (from IntelliSense) that there are two child elements of an UpdatePanel control. One of which is the Triggers element, which specifies the controls on the page (or the user control, if you are using one) that will trigger a partial render of the UpdatePanel control in which the element resides.

Introduction

Microsoft's ASP.NET technology brings an object-oriented and event-driven programming model and unites it with the benefits of compiled code. However, its server-side processing model has several drawbacks inherent in the technology, many of which can be addressed by the new features included in the Microsoft ASP.NET 3.5 AJAX Extensions. These extensions enable many new rich client features, including partial rendering of pages without requiring a full page refresh, the ability to access Web Services via client script (including the ASP.NET profiling API), and an extensive client-side API designed to mirror many of the control schemes seen in the ASP.NET server-side control set.

This whitepaper examines the XML Triggers functionality of the ASP.NET AJAX `UpdatePanel` component. XML Triggers give granular control over the components that can cause partial rendering for specific UpdatePanel controls.

This whitepaper is based on the Beta 2 release of the .NET Framework 3.5 and Visual Studio 2008. The ASP.NET AJAX Extensions, previously an add-on assembly targeted at ASP.NET 2.0, are now integrated into the .NET Framework Base Class Library. This whitepaper also assumes that you will be working with Visual Studio 2008, not Visual Web Developer Express, and will provide walkthroughs according to the user interface of Visual Studio (although code listings will be entirely compatible regardless of development environment).

Triggers

Triggers for a given UpdatePanel, by default, automatically include any child controls that invoke a postback, including (for example) TextBox controls that have their `AutoPostBack` property set to **true**. However, triggers can also be included declaratively using markup; this is done within the `<triggers>` section of the UpdatePanel control declaration. Although triggers can be accessed via the `Triggers` collection property, it is recommended that you register any partial render triggers at run-time (for instance, if a control is not available at design time) by using the `RegisterAsyncPostBackControl(Control)` method of the ScriptManager object for your page, within the `Page_Load` event. Remember that Pages are stateless, and so you should re-register these controls every time they are created.

Automatic child trigger inclusion can also be disabled (so that child controls that create postbacks do not automatically trigger partial renders) by setting the `ChildrenAsTriggers` property to **false**. This allows you the greatest flexibility in assigning which specific controls may invoke a page render, and is recommended, so that a developer will opt-in to respond to an event, rather than handling any events that may arise.

Note that when UpdatePanel controls are nested, when the UpdateMode is set to **Conditional**, if the child UpdatePanel is triggered, but the parent is not, then only the child UpdatePanel will refresh. However, if the parent UpdatePanel is refreshed, then the child UpdatePanel will also be refreshed.

The <Triggers> Element

When working in the markup editor in Visual Studio, you may notice (from IntelliSense) that there are two child elements of an `UpdatePanel` control. The most-frequently seen element is the `<ContentTemplate>` element, which essentially encapsulates the content that will be held by the update panel (the content for which we are enabling partial rendering). The other element is the `<Triggers>` element, which specifies the controls on the page (or the user control, if you are using one) that will trigger a partial render of the UpdatePanel control in which the <Triggers> element resides.

The `<Triggers>` element can contain any number each of two child nodes: `<asp:AsyncPostBackTrigger>` and `<asp:PostBackTrigger>`. They both accept two attributes, `ControlID` and `EventName`, and can specify any Control within the current unit of encapsulation (for instance, if your UpdatePanel control resides within a Web User Control, you should not attempt to reference a Control on the Page on which the User Control will reside).

The `<asp:AsyncPostBackTrigger>` element is particularly useful in that it can target any event from a Control that exists as a child of *any* UpdatePanel control in the unit of encapsulation, not just the UpdatePanel under which this trigger is a child. Thus, any control can be made to trigger a partial page update.

Similarly, the `<asp:PostBackTrigger>` element can be used to trigger a partial page render, but one that requires a full round-trip to the server. This trigger element can also be used to force a full page render when a control would otherwise normally trigger a partial page render (for instance, when a `Button` control exists in the `<ContentTemplate>` element of an UpdatePanel control). Again, the PostBackTrigger element can specify any control that is a child of any UpdatePanel control in the current unit of encapsulation.

<Triggers> Element Reference

Markup Descendants:

TAG	DESCRIPTION
<asp:AsyncPostBackTrigger>	Specifies a control and event that will cause a partial page update for the UpdatePanel that contains this trigger reference.
<asp:PostBackTrigger>	Specifies a control and event that will cause a full page update (a full page refresh). This tag can be used to force a full refresh when a control would otherwise trigger partial rendering.

Walkthrough: Cross-UpdatePanel Triggers

1. Create a new ASP.NET page with a ScriptManager object set to enable partial rendering. Add two UpdatePanels to this page - in the first, include a Label control (Label1) and two Button controls (Button1 and Button2). Button1 should say Click to Update Both and Button2 should say Click to Update This, or something along those lines. In the second UpdatePanel, include only a Label control (Label2), but set its ForeColor property to something other than the default to differentiate it.
2. Set the UpdateMode property of both UpdatePanel tags to **Conditional**.

Listing 1: Markup for default.aspx:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
   <head runat="server">
      <title>Untitled Page</title>
   </head>
   <body>
      <form id="form1" runat="server">
         <asp:ScriptManager EnablePartialRendering="true"
            ID="ScriptManager1" runat="server"></asp:ScriptManager>
         <div>
            <asp:UpdatePanel ID="UpdatePanel1" runat="server"
               UpdateMode="Conditional">
               <ContentTemplate>
                  <asp:Label ID="Label1" runat="server" />
                  <br />
                  <asp:Button ID="Button1" runat="server"
                     Text="Update Both Panels" OnClick="Button1_Click" />
                  <asp:Button ID="Button2" runat="server"
                     Text="Update This Panel" OnClick="Button2_Click" />
               </ContentTemplate>
            </asp:UpdatePanel>
            <asp:UpdatePanel ID="UpdatePanel2" runat="server"
               UpdateMode="Conditional">
               <ContentTemplate>
                  <asp:Label ID="Label2" runat="server" ForeColor="red" />
               </ContentTemplate>
               <Triggers>
                  <asp:AsyncPostBackTrigger ControlID="Button1" EventName="Click" />
               </Triggers>
            </asp:UpdatePanel>
         </div>
      </form>
   </body>
</html>
```

1. In the Click event handler for Button1, set Label1.Text and Label2.Text to something time-dependent (such as DateTime.Now.ToLongTimeString()). For the Click event handler for Button2, set only Label1.Text to the time-dependent value.

Listing 2: Codebehind (trimmed) in default.aspx.cs:

```
public partial class _Default : System.Web.UI.Page
{
    protected void Button1_Click(object sender, EventArgs e)
    {
        Label1.Text = DateTime.Now.ToLongTimeString();
        Label2.Text = DateTime.Now.ToLongTimeString();
    }
    protected void Button2_Click(object sender, EventArgs e)
    {
        Label1.Text = DateTime.Now.ToLongTimeString();
    }
}
```

1. Press F5 to build and run the project. Note that, when you click Update Both Panels, both labels change text; however, when you click Update This Panel, only Label1 updates.

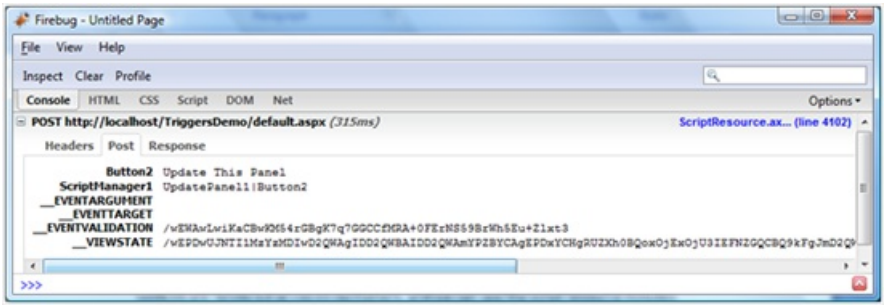
(Click to view full-size image)

Under the Hood

Utilizing the example we just constructed, we can take a look at what ASP.NET AJAX is doing and how our UpdatePanel cross-panel triggers work. To do so, we will work with the generated page source HTML, as well as the Mozilla Firefox extension called FireBug - with it, we can easily examine the AJAX postbacks. We will also use the .NET Reflector tool by Lutz Roeder. Both of these tools are freely available online, and can be found with an internet search.

An examination of the page source code shows almost nothing out of the ordinary; the UpdatePanel controls are rendered as `<div>` containers, and we can see the script resource includes provided by the `<asp:ScriptManager>`. There are also some new AJAX-specific calls to the PageRequestManager that are internal to the AJAX client script library. Finally, we see the two UpdatePanel containers - one with the rendered `<input>` buttons with the two `<asp:Label>` controls rendered as `<span>` containers. (If you inspect the DOM tree in FireBug, you will notice that the labels are dimmed to indicate that they are not producing visible content).

Click the Update This Panel button, and notice the top UpdatePanel will be updated with the current server time. In FireBug, choose the Console tab so that you can examine the request. Examine the POST request parameters first:



(Click to view full-size image)

Note that the UpdatePanel has indicated to the server-side AJAX code precisely which control tree was fired via the ScriptManager1 parameter: `Button1` of the `UpdatePanel1` control. Now, click on the Update Both Panels button. Then, examining the response, we see a pipe-delimited series of variables set in a string; specifically, we see the top UpdatePanel, `UpdatePanel1`, has the entirety of its HTML sent to the browser. The AJAX client script library substitutes the UpdatePanel's original HTML content with the new content via the `.innerHTML` property, and so the server sends the changed content from the server as HTML.

Now, click on the Update Both Panels button and examine the results from the server. The results are very similar - both UpdatePanels receive new HTML from the server. As with the previous callback, additional page state is sent.

As we can see, because no special code is utilized to perform an AJAX postback, the AJAX client script library is able to intercept form postbacks without any additional code. Server controls automatically utilize JavaScript so that they do not automatically submit the form - ASP.NET automatically injects code for form validation and state already, primarily achieved by automatic script resource inclusion, the PostBackOptions class, and the ClientScriptManager class.

For instance, consider a CheckBox control; examine the class disassembly in .NET Reflector. To do so, ensure that your System.Web assembly is open, and navigate to the `System.Web.UI.WebControls.CheckBox` class, opening the `RenderInputTag` method. Look for a conditional that checks the `AutoPostBack` property:

```
    onClick = Util.MergeScript(onClick, this.Page.ClientScript.GetPostBackEventReference(options, true));
    writer.AddAttribute(HtmlTextWriterAttribute.Onclick, onClick);
    if (base.EnableLegacyRendering)
    {
        writer.AddAttribute("language", "javascript", false);
    }
}
else
{
    if (this.Page != null)
    {
        this.Page.ClientScript.RegisterForEventValidation(this.UniqueID);
    }
    if (onClick != null)
    {
        writer.AddAttribute(HtmlTextWriterAttribute.Onclick, onClick);
    }
```

(Click to view full-size image)

When automatic postback is enabled on a `CheckBox` control (via the AutoPostBack property being true), the resultant `<input>` tag is therefore rendered with an ASP.NET event handling script in its `onclick` attribute. The interception of the form's submission, then, allows ASP.NET AJAX to be injected into the page nonintrusively, helping to avoid any potential breaking changes that might occur by utilizing a possibly-imprecise string replacement. Furthermore, this enables *any* custom ASP.NET control to utilize the power of ASP.NET AJAX without any additional code to support its use within an UpdatePanel container.

The `<triggers>` functionality corresponds to the values initialized in the PageRequestManager call to _updateControls (note that the ASP.NET AJAX client script library utilizes the convention that methods, events, and field names that begin with an underscore are marked as internal, and are not meant for use outside of the library itself). With it, we can observe which controls are intended to cause AJAX postbacks.

For example, let's add two additional controls to the page, leaving one control outside of the UpdatePanels entirely, and leaving one within an UpdatePanel. We will add a CheckBox control within the upper UpdatePanel, and drop a DropDownList with a number of colors defined within the list. Here is the new markup:

Listing 3: New Markup

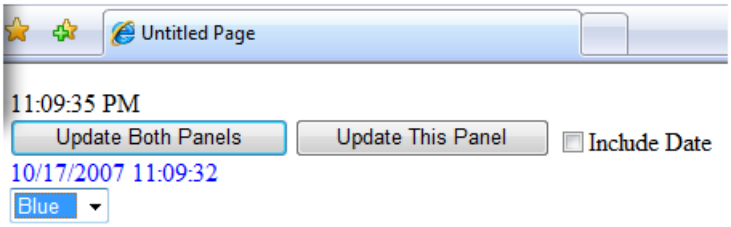
```
<%@ Page Language="C#" AutoEventWireup="true"
 CodeFile="Default.aspx.cs" Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
 <head id="Head1" runat="server">
 <title>Untitled Page</title>
 </head>
 <body>
 <form id="form1" runat="server">
 <asp:ScriptManager EnablePartialRendering="true"
 ID="ScriptManager1" runat="server"></asp:ScriptManager>
 <div>
 <asp:UpdatePanel ID="UpdatePanel1" runat="server"
 UpdateMode="Conditional">
 <ContentTemplate>
 <asp:Label ID="Label1" runat="server" /><br />
 <asp:Button ID="Button1" runat="server"
 Text="Update Both Panels" OnClick="Button1_Click" />
 <asp:Button ID="Button2" runat="server"
 Text="Update This Panel" OnClick="Button2_Click" />
 <asp:CheckBox ID="cbDate" runat="server"
 Text="Include Date" AutoPostBack="false"
 OnCheckedChanged="cbDate_CheckedChanged" />
 </ContentTemplate>
 </asp:UpdatePanel>
 <asp:UpdatePanel ID="UpdatePanel2" runat="server"
 UpdateMode="Conditional">
 <ContentTemplate>
 <asp:Label ID="Label2" runat="server"
 ForeColor="red" />
 </ContentTemplate>
 <Triggers>
 <asp:AsyncPostBackTrigger ControlID="Button1"
 EventName="Click" />
 <asp:AsyncPostBackTrigger ControlID="ddlColor"
 EventName="SelectedIndexChanged" />
 </Triggers>
 </asp:UpdatePanel>
 <asp:DropDownList ID="ddlColor" runat="server"
 AutoPostBack="true"
 OnSelectedIndexChanged="ddlColor_SelectedIndexChanged">
 <asp:ListItem Selected="true" Value="Red" />
 <asp:ListItem Value="Blue" />
 <asp:ListItem Value="Green" />
 </asp:DropDownList>
 </div>
 </form>
 </body>
</html>
```

And here is the new code-behind:

Listing 4: Codebehind

```
public partial class _Default : System.Web.UI.Page
{
    protected void Button1_Click(object sender, EventArgs e)
    {
        if (cbDate.Checked)
        {
            Label1.Text = DateTime.Now.ToString("MM/dd/yyyy hh:mm:ss");
            Label2.Text = DateTime.Now.ToString("MM/dd/yyyy hh:mm:ss");
        }
        else
        {
            Label1.Text = DateTime.Now.ToLongTimeString();
            Label2.Text = DateTime.Now.ToLongTimeString();
        }
    }
    protected void Button2_Click(object sender, EventArgs e)
    {
        if (cbDate.Checked)
        {
            Label1.Text = DateTime.Now.ToString("MM/dd/yyyy hh:mm:ss");
        }
        else
        {
            Label1.Text = DateTime.Now.ToLongTimeString();
        }
    }
    protected void cbDate_CheckedChanged(object sender, EventArgs e)
    {
        cbDate.Font.Bold = cbDate.Checked;
    }
    protected void ddlColor_SelectedIndexChanged(object sender, EventArgs e)
    {
        Color c = Color.FromName(ddlColor.SelectedValue);
        Label2.ForeColor = c;
    }
}
```

The idea behind this page is that the drop-down list selects one of three colors to show the second label, that the check box determines both whether it is bold, and whether the labels display the date as well as the time. The check box should not cause an AJAX update, but the drop-down list should, even though it is not housed within an UpdatePanel.



(Click to view full-size image)

As is apparent in the above screen shot, the most-recent button to be clicked was the right button Update This Panel, which updated the top time independent of the bottom time. The date was also switched off between clicks, as the date is visible in the bottom label. Finally of interest is the bottom label's color: it was updated more recently than the label's text, which demonstrates that control state is important, and users expect it to be preserved through AJAX postbacks. *However*, the time was not updated. The time was automatically repopulated through the persistence of the __VIEWSTATE field of the page being interpreted by the ASP.NET runtime when the control was being re-rendered on the server. The ASP.NET AJAX server code does not recognize in which methods the controls are changing state; it simply repopulates from view state and then runs the events that are appropriate.

It should be pointed out, however, that had I initialized the time within the Page_Load event, the time would have been incremented correctly. Consequently, developers should be wary that the appropriate code is being run during the appropriate event handlers, and avoid use of Page_Load when a control event handler would be appropriate.

Summary

The ASP.NET AJAX Extensions UpdatePanel control is versatile, and can utilize a number of methods for identifying control events that should cause it to be updated. It supports being updated automatically by its child controls, but can also respond to control events elsewhere on the page.

To reduce potential for server processing load, it is recommended that the `ChildrenAsTriggers` property of an UpdatePanel be set to `false`, and that events be opted-into rather than included by default. This also prevents any unneeded events from causing potentially-unwanted effects, including validation, and changes to input fields. These types of bugs may be difficult to isolate, because the page updates transparently to the user, and the cause may therefore not be immediately obvious.

By examining the inner workings of the ASP.NET AJAX form post interception model, we were able to determine that it utilizes the framework already provided by ASP.NET. In doing so, it preserves maximum compatibility with controls designed using the same framework, and intrudes minimally on any additional JavaScript written for the page.

Bio

Rob Paveza is a senior .NET application developer at Terralever (www.terralever.com), a leading interactive marketing firm in Tempe, AZ. He can be reached at robpaveza@gmail.com, and his blog is located at http://geekswithblogs.net/robp/.

Scott Cate has been working with Microsoft Web technologies since 1997 and is the President of myKB.com (www.myKB.com) where he specializes in writing ASP.NET based applications focused on Knowledge Base Software solutions. Scott can be contacted via email at scott.cate@myKB.com or his blog at ScottCate.com

PREVIOUS NEXT

Understanding ASP.NET AJAX Authentication and Profile Application Services

2/20/2020 • 16 minutes to read • Edit Online

by Scott Cate

Download PDF

The Authentication service allows users to provide credentials in order to receive an authentication cookie, and is the gateway service to allow custom user profiles provided by ASP.NET. Use of the ASP.NET AJAX authentication service is compatible with standard ASP.NET Forms authentication, so applications currently using Forms authentication (such as with the Login control) will not be broken by upgrading to the AJAX authentication service.

Introduction

As part of the .NET Framework 3.5, Microsoft is delivering a sizable environment upgrade; not only is a new development environment available, but the new Language-Integrated Query (LINQ) features and other language enhancements are forthcoming. In addition, some familiar features of other toolsets, notably the ASP.NET AJAX Extensions, are being included as first-class members of the .NET Framework Base Class Library. These extensions enable many new rich client features, including partial rendering of pages without requiring a full page refresh, the ability to access Web Services via client script (including the ASP.NET profiling API), and an extensive client-side API designed to mirror many of the control schemes seen in the ASP.NET server-side control set.

This whitepaper looks at the implementation and use of the ASP.NET Profiling and Forms Authentication services as they are exposed by the Microsoft ASP.NET AJAX ExtensionsThe AJAX Extensions make Forms authentication incredibly easy to support, as it (as well as the Profiling Service) is exposed through a Web Service proxy script. The AJAX Extensions also support custom authentication through the AuthenticationServiceManager class.

This whitepaper is based on the Beta 2 release of the Visual Studio 2008 and the .NET Framework 3.5. This whitepaper also assumes that you will be working with Visual Studio 2008 Beta 2, not Visual Web Developer Express, and will provide walkthroughs according to the user interface of Visual Studio. Some code samples may utilize project templates unavailable in Visual Web Developer Express.

Profiles and Authentication

The Microsoft ASP.NET Profiles and Authentication services are provided by the ASP.NET Forms Authentication system, and are standard components of ASP.NET. The ASP.NET AJAX Extensions provide script access to these services via script proxies, through a fairly straightforward model under the Sys.Services namespace of the client AJAX library.

The Authentication service allows users to provide credentials in order to receive an authentication cookie, and is the gateway service to allow custom user profiles provided by ASP.NET. Use of the ASP.NET AJAX authentication service is compatible with standard ASP.NET Forms authentication, so applications currently using Forms authentication (such as with the Login control) will not be broken by upgrading to the AJAX authentication service.

The Profile service allows the automatic integration and storage of user data based on membership as provided by the Authentication service. The stored data is specified by the web.config file, and the various profiling service providers handle the data management. As with the Authentication service, the AJAX Profile service is compatible

with the standard ASP.NET profile service, so that pages currently incorporating features of the ASP.NET Profile service should not be broken by including AJAX support.

Incorporating the ASP.NET Authentication and Profiling services themselves into an application is outside of the scope of this whitepaper. For more information on the topic, see the MSDN Library reference article Managing Users by Using Membership at https://msdn.microsoft.com/library/tw292whz.aspx. ASP.NET also includes a utility to automatically set up Membership with a SQL Server, which is the default authentication service provider for ASP.NET Membership. For more information, see the article ASP.NET SQL Server Registration Tool (Aspnet_regsql.exe) at https://msdn.microsoft.com/library/ms229862(vs.80).aspx.

Using the ASP.NET AJAX Authentication Service

The ASP.NET AJAX Authentication service must be enabled in the web.config file:

```
<system.web.extensions>
 <scripting>
 <webServices>
 <authenticationService enabled="true" />
 </webServices>
 </scripting>
</system.web.extensions>
```

The Authentication service requires ASP.NET Forms authentication to be enabled and requires cookies to be enabled on the client browser (a script cannot enable a cookieless session since cookieless sessions require URL parameters).

Once the AJAX authentication service is enabled and configured, client script can immediately take advantage of the Sys.Services.AuthenticationService object. Primarily, client script will want to take advantage of the `login` method and `isLoggedIn` property. Several properties exist to provide defaults for the login method, which can accept a large number of parameters.

Sys.Services.AuthenticationService members

login method:

The login() method begins a request to authenticate the user's credentials. This method is asynchronous and does not block execution.

Parameters:

PARAMETER NAME	MEANING
userName	Required. The username to authenticate.
password	Optional (defaults to null). The user's password.
isPersistent	Optional (defaults to false). Whether the user's authentication cookie should persist across sessions. If false, the user will log out when the browser is closed or the session expires.
redirectUrl	Optional (defaults to null).The URL to redirect the browser to upon successful authentication. If this parameter is null or an empty string, no redirection occurs.
customInfo	Optional (defaults to null). This parameter is currently unused and is reserved for future use.

PARAMETER NAME	MEANING
loginCompletedCallback	Optional (defaults to null).The function to call when the login has successfully completed. If specified, this parameter overrides the defaultLoginCompleted property.
failedCallback	Optional (defaults to null).The function to call when the login has failed. If specified, this parameter overrides the defaultFailedCallback property.
userContext	Optional (defaults to null). Custom user context data that should be passed to the callback functions.

Return Value:

This function does not include a return value. However, a number of behaviors are included upon completion of a call to this function:

- The current page will either be refreshed or be changed if the `redirectUrl` parameter was neither null nor an empty string.
- However, if the parameter was null or an empty string, the `loginCompletedCallback` parameter, or `defaultLoginCompletedCallback` property is called.
- If the call to the web service fails, the `failedCallback` parameter of `defaultFailedCallback` property is called.

logout method:

The logout() method removes the credentials cookie and logs out the current user from the web application.

Parameters:

PARAMETER NAME	MEANING
redirectUrl	Optional (defaults to null).The URL to redirect the browser to upon successful authentication. If this parameter is null or an empty string, no redirection occurs.
logoutCompletedCallback	Optional (defaults to null).The function to call when the logout has successfully completed. If specified, this parameter overrides the defaultLogoutCompleted property.
failedCallback	Optional (defaults to null).The function to call when the login has failed. If specified, this parameter overrides the defaultFailedCallback property.
userContext	Optional (defaults to null). Custom user context data that should be passed to the callback functions.

Return Value:

This function does not include a return value. However, a number of behaviors are included upon completion of a call to this function:

- The current page will either be refreshed or be changed if the `redirectUrl` parameter was neither null nor an empty string.
- However, if the parameter was null or an empty string, the `logoutCompletedCallback` parameter, or `defaultLogoutCompletedCallback` property is called.
- If the call to the web service fails, the `failedCallback` parameter of `defaultFailedCallback` property is called.

defaultFailedCallback property (get, set):

This property specifies a function that should be called if a failure to communicate with the web service occurs. It should receive a delegate (or function reference).

The function reference specified by this property should have the following signature:

```
function AuthenticationFailureCallback(error, userContext, methodName);
```

Parameters:

PARAMETER NAME	MEANING
error	Specifies the error information.
userContext	Specifies the user context information provided when the login or logout function was called.
methodName	The name of the calling method.

defaultLoginCompletedCallback property (get, set):

This property specifies a function that should be called when the login web service call has completed. It should receive a delegate (or function reference).

The function reference specified by this property should have the following signature:

```
function AuthenticationLoginCompletedCallback(validCredentials, userContext, methodName);
```

Parameters:

PARAMETER NAME	MEANING
validCredentials	Specifies whether the user provided valid credentials. `true` if the user successfully logged in; otherwise `false`.
userContext	Specifies the user context information provided when the login function was called.
methodName	The name of the calling method.

defaultLogoutCompletedCallback property (get, set):

This property specifies a function that should be called when the logout web service call has completed. It should receive a delegate (or function reference).

The function reference specified by this property should have the following signature:

```
function AuthenticationLogoutCompletedCallback(result, userContext, methodName);
```

Parameters:

PARAMETER NAME	MEANING
result	This parameter will always be `null`; it is reserved for future use.
userContext	Specifies the user context information provided when the login function was called.
methodName	The name of the calling method.

isLoggedIn property (get):

This property gets the current authentication state of the user; it is set by the ScriptManager object during a page request.

This property returns `true` if the user is currently logged in; otherwise, it returns `false`.

path property (get, set):

This property programmatically determines the location of the authentication web service. It can be used to override the default authentication provider, as well as one set declaratively in the Path property of the ScriptManager control's AuthenticationService child node (for more information, see the Using a Custom Authentication Service Provider topic below).

Note that the location of the default authentication service does not change. However, ASP.NET AJAX allows you to specify the location of a web service that provides the same class interface as the ASP.NET AJAX authentication service proxy.

Note also that this property should not be set to a value that directs the script request off of the current site. Because the current application would not receive the authentication credentials, it would be useless; also, the technology underlying AJAX should not post cross-site requests, and may generate a security exception in the client browser.

This property is a `String` object representing the path to the authentication web service.

timeout property (get, set):

This property determines the length of time to wait for the authentication service before assuming the login request has failed. If the timeout expires while waiting for a call to complete, the request-failed callback will be called, and the call will not complete.

This property is a `Number` object representing the number of milliseconds to wait for results from the authentication service.

Code Sample: Logging into the Authentication Service

The following markup is an example ASP.NET page with a simple script call to the login and logout methods of the AuthenticationService class.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
 <head runat="server">
 <title>Login Example</title>
 <script type="text/javascript">
 function Login()
 {
 var userTextbox = $get("txtUser");
 var passTextbox = $get("txtPassword");
 Sys.Services.AuthenticationService.login(userTextbox.value,
 passTextbox.value, false, null, null, LoginServiceCompleted,
 LoginServiceFailed, "Context Info");
 }
 function Logout()
 {
 Sys.Services.AuthenticationService.logout(null, LogoutServiceCompleted,
 LoginServiceFailed, "Context Info");
 }
 function LoginServiceFailed(error, userContext, methodName)
 {
 alert('There was an error with the authentication service:\n\n' + error);
 }
 function LoginServiceCompleted(validCredentials, userContext, methodName)
 {
 if (validCredentials)
 {
 alert('Great! You successfully logged in.');
 }
 else
 {
 alert('Oops! You gave us bad credentials!');
 }
 }
 function LogoutServiceCompleted(result, userContext, methodName)
 {
 alert('You have been logged out from the web site.');
 }
 </script>
 </head>
 <body>
 <form id="form1" runat="server">
 <asp:ScriptManager ID="ScriptManager1" runat="server"
 EnableScriptLocalization="true">
 </asp:ScriptManager>
 <div>
 <asp:TextBox ID="txtUser" runat="Server"></asp:TextBox>
 <br />
 <asp:TextBox ID="txtPassword" runat="Server" TextMode="Password"/>
 <br />
 <asp:Button Text="Log in" ID="btnLogin" runat="server"
 OnClientClick="Login(); return false;" />
 </div>
 </form>
 </body>
</html>
```

Accessing ASP.NET Profiling Data via AJAX

The ASP.NET profiling service is also exposed through the ASP.NET AJAX Extensions. Since the ASP.NET profiling service provides a rich, granular API by which to store and retrieve user data, this can be an excellent productivity tool.

The profile service must be enabled in web.config; it is not by default. To do so, ensure that the `profileService`

child element has enabled= true specified in web.config, and that you have specified which properties can be read or written as follows:

```
<system.web.extensions>
 <scripting>
 <webServices>
 <profileService enabled="true"
 readAccessProperties= Name,Address,BackgroundColor
 writeAccessProperties= BackgroundColor />
 </webServices>
 </scripting>
</system.web.extensions>
```

The profile service must also be configured. Although configuration of the profiling service is outside of the scope of this whitepaper, it is worthwhile to note that groups as defined in profile configuration settings will be accessible as sub-properties of the group name. For example, with the following profile section specified:

```
<profile enabled="true">
 <properties>
 <add name="Name" type="System.String"/>
 <group name="Address">
 <add name="Line1" type="System.String"/>
 <add name="Line2" type="System.String"/>
 <add name="City" type="System.String"/>
 <add name="State" type="System.String"/>
 <add name="Zip" type="System.String"/>
 </group>
 <add name="BackgroundColor" type="System.Drawing.Color"/>
 </properties>
</profile>
```

Client script would be able to access Name, Address.Line1, Address.Line2, Address.City, Address.State, Address.Zip, and BackgroundColor as properties of the properties field of the ProfileService class.

Once the AJAX Profiling Service is configured, it will be immediately available in pages; however, it will have to be loaded once before use.

Sys.Services.ProfileService members

properties field:

The properties field exposes all configured profile data as child properties that can be referenced by the dot-operator-name convention. Properties that are children of property groups are referred to as GroupName.PropertyName. In the example profile configuration presented above, to get the state of the user, you could use the following identifier:

```
Sys.Services.ProfileService.properties.Address.State
```

load method:

Loads a selected list or all properties from the server.

Parameters:

PARAMETER NAME	MEANING
propertyNames	Optional (defaults to null). The properties to be loaded from the server.

PARAMETER NAME	MEANING
loadCompletedCallback	Optional (defaults to null). The function to call when loading has completed.
failedCallback	Optional (defaults to null). The function to call if an error occurs.
userContext	Optional (defaults to null). Context information to be passed to the callback function.

The load function does not have a return value. If the call completed successfully, it will call either the `loadCompletedCallback` parameter or `defaultLoadCompletedCallback` property. If the call failed, or the timeout expired, either the `failedCallback` parameter or `defaultFailedCallback` property will be called.

If the `propertyNames` parameter is not supplied, all read-configured properties are retrieved from the server.

save method:

The save() method saves the specified property list (or all properties) to the user's ASP.NET profile.

Parameters:

PARAMETER NAME	MEANING
propertyNames	Optional (defaults to null). The properties to be saved to the server.
saveCompletedCallback	Optional (defaults to null). The function to call when saving has completed.
failedCallback	Optional (defaults to null). The function to call if an error occurs.
userContext	Optional (defaults to null). Context information to be passed to the callback function.

The save function does not have a return value. If the call completes successfully, it will call either the `saveCompletedCallback` parameter or `defaultSaveCompletedCallback` property. If the call failed, or the timeout expired, either the `failedCallback` or `defaultFailedCallback` property will be called.

If the `propertyNames` parameter is null, all profile properties will be sent to the server, and the server will decide which properties can be saved and which cannot.

defaultFailedCallback property (get, set):

This property specifies a function that should be called if a failure to communicate with the web service occurs. It should receive a delegate (or function reference).

The function reference specified by this property should have the following signature:

```
function AuthenticationFailureCallback(error, userContext, methodName);
```

Parameters:

PARAMETER NAME	MEANING
Error	Specifies the error information.
userContext	Specifies the user context information provided when the load or save function was called.
methodName	The name of the calling method.

defaultSaveCompleted property (get, set):

This property specifies a function that should be called upon the completion of saving the user's profile data. It should receive a delegate (or function reference).

The function reference specified by this property should have the following signature:

```
function ProfileSaveComplete(numPropsSaved, userContext, methodName);
```

Parameters:

PARAMETER NAME	MEANING
numPropsSaved	Specifies the number of properties that were saved.
userContext	Specifies the user context information provided when the load or save function was called.
methodName	The name of the calling method.

defaultLoadCompleted property (get, set):

This property specifies a function that should be called upon the completion of loading of the user's profile data. It should receive a delegate (or function reference).

The function reference specified by this property should have the following signature:

```
function ProfileLoadComplete(numPropsLoaded, userContext, methodName);
```

Parameters:

PARAMETER NAME	MEANING
numPropsLoaded	Specifies the number of properties loaded.
userContext	Specifies the user context information provided when the load or save function was called.
methodName	The name of the calling method.

path property (get, set):

This property programmatically determines the location of the profile web service. It can be used to override the default profile service provider, as well as one set declaratively in the Path property of the ScriptManager control's ProfileService child node.

Note that the location of the default profile service does not change. However, ASP.NET AJAX allows you to specify the location of a web service that provides the same class interface as the ASP.NET AJAX authentication service proxy.

Note also that this property should not be set to a value that directs the script request off of the current site. The technology underlying AJAX should not post cross-site requests, and may generate a security exception in the client browser.

This property is a `String` object representing the path to the profile web service.

timeout property (get, set):

This property determines the length of time to wait for the profile service before assuming the load or save request has failed. If the timeout expires while waiting for a call to complete, the request-failed callback will be called, and the call will not complete.

This property is a `Number` object representing the number of milliseconds to wait for results from the profile service.

Code sample: Loading profile data at page load

The following code will check to see whether a user is authenticated, and if so, will load the user's preferred background color as the page's.

```
function Page_Load()
{
 if (Sys.Services.AuthenticationService.get_isLoggedIn())
 {
 Sys.Services.ProfileService.load();
 }
}
function ProfileLoaded(numPropsLoaded, userContext, methodName)
{
 document.documentElement.style.backgroundColor = Sys.Services.ProfileService.properties.BackgroundColor;
}
```

Using a Custom Authentication Service Provider

The ASP.NET AJAX Extensions allow you to create a custom script authentication service provider by exposing your functionality through a custom web service. In order to be used, your web service must expose two methods, `Login` and `Logout`; and these methods must be specified with the same method signatures as the default ASP.NET AJAX Authentication web service.

Once you have created the custom web service, you will need to specify the path to it, either declaratively on your page, programmatically in code, or via the client script.

To set the path declaratively:

To set the path declaratively, include the AuthenticationService child of the ScriptManager object on your ASP.NET page:

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
 <AuthenticationService Path="~/AuthService.asmx" />
</asp:ScriptManager>
```

To set the path in code:

To set the path programmatically, specify the path via the instance of your script manager:

```
protected void Page_Load(object sender, EventArgs e)
{
    this.ScriptManager1.AuthenticationService.Path = "~/AuthService.asmx";
}
```

To set the path in script:

To set the path programmatically in script, utilize the `path` property of the AuthenticationService class:

```
function Login()
{
 var userTextbox = $get("txtUser");
 var passTextbox = $get("txtPassword");
 Sys.Services.AuthenticationService.set_path("./AuthService.asmx");
 Sys.Services.AuthenticationService.login(userTextbox.value, passTextbox.value, false, null, null,
LoginServiceCompleted, LoginServiceFailed, "Context Info");
}
```

Sample Web Service for Custom Authentication

```
<%@ WebService Language="C#" Class="AuthService" %>
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Web.Script.Services;
[ScriptService]
[WebService]
public class AuthService : WebService
{
 [WebMethod]
 public bool Login(string userName, string password, bool createCookie)
 {
 Session["LoggedInUser"] = userName;
 return true;
 }
 [WebMethod]
 public void Logout()
 {
 Session.Abandon();
 }
}
```

Summary

ASP.NET services - specifically the profiling, membership, and authentication services - are easily exposed to JavaScript on the client browser. This allows developers to integrate their client-side code with the authentication mechanism seamlessly, without depending on controls such as UpdatePanels to do the heavy lifting. Profile data can be protected from the client as well, by utilizing web configuration settings; no data is available by default, and developers must opt-in to profile properties.

Furthermore, by creating simplified web service implementations with equivalent method signatures, developers can create custom script providers for these intrinsic ASP.NET services. Support for these techniques simplifies the development of rich client applications, while providing developers with a wide range of flexibility to meet specific needs.

Bio

Scott Cate has been working with Microsoft Web technologies since 1997 and is the President of myKB.com (www.myKB.com) where he specializes in writing ASP.NET based applications focused on Knowledge Base Software solutions. Scott can be contacted via email at scott.cate@myKB.com or his blog at ScottCate.com

PREVIOUS NEXT

Understanding ASP.NET AJAX Localization

2/20/2020 • 8 minutes to read • Edit Online

by Scott Cate

Download PDF

Localization is the process of designing and integrating support for a specific language and culture into an application or an application component. The Microsoft ASP.NET platform provides extensive support for localization for standard ASP.NET applications by integrating the standard .NET localization model; the Microsoft AJAX Framework utilize the integrated model to support the diverse scenarios in which localization can be performed.

Introduction

Microsoft's ASP.NET technology brings an object-oriented and event-driven programming model and unites it with the benefits of compiled code. However, its server-side processing model has several drawbacks inherent in the technology, many of which can be addressed by the new features included in the System.Web.Extensions namespace, which encapsulates the Microsoft AJAX Services in the .NET Framework 3.5. These extensions enable many rich client features, previously available as part of the ASP.NET 2.0 AJAX Extensions, but now part of the Framework Base Class Library. Controls and features in this namespace include partial rendering of pages without requiring a full page refresh, the ability to access Web Services via client script (including the ASP.NET profiling API), and an extensive client-side API designed to mirror many of the control schemes seen in the ASP.NET server-side control set.

This whitepaper examines the localization features present in the Microsoft AJAX Framework and Microsoft AJAX Script Library, in the context of business need for localization support and reviewing already-integrated support for localization in web applications provided by the .NET Framework. The Microsoft AJAX Script Library utilizes the .resx file format already used by .NET applications, which provides integrated IDE support and a shareable resource type.

This whitepaper is based on the Beta 2 release of Microsoft Visual Studio 2008. This whitepaper also assumes that you will be working with Visual Studio 2008, not Visual Web Developer Express, and will provide walkthroughs according to the user interface of Visual Studio. Some code samples will utilize project templates that may be unavailable in Visual Web Developer Express.

The Need for Localization

Particularly for enterprise application developers and component developers, the ability to create tools that can be aware of the differences between cultures and languages has become increasingly necessary. Designing components with the ability to adapt to the locale of the client increases developer productivity and reduces the amount of work required for the adaptation of a component to function globally.

Localization is the process of designing and integrating support for a specific language and culture into an application or an application component. The Microsoft ASP.NET platform provides extensive support for localization for standard ASP.NET applications by integrating the standard .NET localization model; the Microsoft AJAX Framework utilize the integrated model to support the diverse scenarios in which localization can be performed. With the Microsoft AJAX Framework, scripts can either be localized by being deployed into satellite assemblies, or by utilizing a static file system structure.

Embedding Scripts with Satellite Assemblies

Consistent with the standard .NET Framework localization strategy, resources can be included in satellite assemblies. Satellite assemblies provide several advantages over traditional resource inclusion in binaries - any given localization can be updated without updating the larger image, additional localizations can be deployed simply by installing satellite assemblies into the project folder, and satellite assemblies can be deployed without causing a reload of the main project assembly. Particularly in ASP.NET projects, this is beneficial because it can significantly reduce the amount of system resources used by incremental updates, and minimally disrupts production website usage.

Scripts are embedded into assemblies by including them in managed .resx (or compiled .resources) files, which are included into the assembly at compile-time. Their resources are then made available to the script application through AJAX runtime-generated code, via assembly-level attributes

Naming Conventions for Embedded Script Files

The Microsoft AJAX Framework script management supports a variety of options for use in deployment and testing of scripts, and guidelines are provided to facilitate these options.

To facilitate debugging:

Release (production) scripts should not include the `.debug` qualifier in the filename. Scripts designed for debugging should include `.debug` in the filename.

To facilitate localization:

Neutral-culture scripts should not include any culture identifier in the name of the file. For scripts that contain localized resources, the ISO language code should be specified in the file name. For example, `es-CO` stands for Spanish, Columbia.

The following table summarizes the file naming conventions with examples:

FILENAME	MEANING
Script.js	A release-version culture-neutral script.
Script.debug.js	A debug-version culture-neutral script.
Script.en-US.js	A release version English, United States script.
Script.debug.es-CO.js	A debug-version Spanish, Columbia script.

Walkthrough: Create an Localized, Embedded Script

Please note: this walkthrough requires the use of Visual Studio 2008 as Visual Web Developer Express does not include a project template for class library projects.

1. Create a new Web Site project with ASP.NET AJAX Extensions integrated. Create another project, a Class Library project, within the solution called LocalizingResources.
2. Add a Jscript file called VerifyDeletion.js to the LocalizingResources project, as well as .resx resources files called DeletionResources.resx and DeletionResources.es.resx. The former will contain culture-neutral resources; the latter will contain Spanish-language resources.
3. Add the following code to VerifyDeletion.js:

```
function VerifyDeletion(fileName)
{
 if (confirm(Message.VerifyDelete.replace(/FILENAME/, fileName)))
 {
 Delete(fileName);
 return true;
 }
 return false;
}
function Delete(fileName)
{
 alert (Message.Deleted.replace(/FILENAME/, fileName));
}
```

For those unfamiliar with JavaScript Regex syntax, text within single forward slashes (in the previous example, /FILENAME/ is an example) denotes a RegExp object. The MSDN Library contains an extensive JavaScript reference, and resources on JavaScript native objects can be found online.

1. Add the following resource strings to DeletionResources.resx:

   VerifyDelete: Are you sure you want to delete FILENAME?

   Deleted: FILENAME has been deleted.

2. Add the following resource strings to DeletionResources.es.resx:

   VerifyDelete: Est seguro que desee quitar FILENAME?

   Deleted: FILENAME se ha quitado.

3. Add the following lines of code to the AssemblyInfo file:

```
[assembly: System.Web.UI.WebResource("LocalizingResources.VerifyDeletion.js",
 "text/javascript")]
[assembly: System.Web.UI.ScriptResource("LocalizingResources.VerifyDeletion.js",
 "LocalizingResources.DeletionResources", "Message")]
```

1. Add references to System.Web and System.Web.Extensions to the LocalizingResources project.
2. Add a reference to the LocalizingResources project from the Web Site project.
3. In default.aspx, under the Web Site project, update the ScriptManager control with the following additional markup:

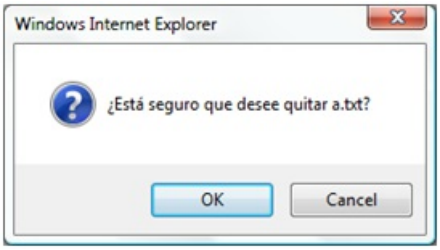
```
<asp:ScriptManager ID="ScriptManager1" runat="server" EnableScriptLocalization="true">
 <Scripts>
 <asp:ScriptReference Assembly="LocalizingResources" Name="LocalizingResources.VerifyDeletion.js"/>
 </Scripts>
</asp:ScriptManager>
```

1. In default.aspx, anywhere on the page, include this markup:

```
<asp:Button ID="btnDelete" runat="Server" OnClientClick="VerifyDeletion('a.txt');" Text="Delete" />
```

1. Press F5. If prompted, enable debugging. When the page is loaded, press the Delete button. Note that you are prompted in English (unless your computer is set to prefer Spanish-language resources by default) for confirmation.
2. Close the browser window and return to default.aspx. In the @Page header directive, replace auto for Culture and UICulture with es-ES. Press F5 again to launch the web application in the browser again. This time, note

that you are prompted to delete the file in Spanish:



(Click to view full-size image)

(Click to view full-size image)

Note that there are several variations for this walkthrough. For instance, scripts could be registered with the ScriptManager control programmatically during page load.

Including a Static Script File Structure

When using static script files for deployment, you lose some of the benefits of using the inherent .NET localization scheme. Primarily visible is that you lose the automatic type generated from including script resource files; in the above walkthrough, for example, resources were exposed by an automatically-generated type called Message from the ScriptManager control.

There are, however, some benefits to using a static script file structure. Updates can be performed without recompiling and redeploying satellite assemblies, and the use of a static file structure can also be done to override embedded script, to integrate a minor piece of functionality that may not have been shipped with a component.

Microsoft recommends avoiding a version control issue by automatically generating your script resources during project compilation. When maintaining an extensive script code base, it can become increasingly difficult to ensure that code changes are reflected in each localized script. As an alternative, you can simply maintain one logic script and multiple localization scripts, merging the files while building the project.

Because there are not resources to declaratively include, static script files should be referenced either by adding `<asp:ScriptElement>` elements as a child of the `<Scripts>` tag of the ScriptManager control, or by programmatically adding `ScriptReference` objects to the `Scripts` property of the `ScriptManager` control on the page at runtime.

The ScriptManager and its Role in Localization

The ScriptManager enables several automatic behaviors for localized applications:

- It automatically locates script files based on settings and naming conventions; for instance, it loads debug-enabled scripts when in debugging mode, and loads localized scripts based on the browser's user interface selection.
- It enables the definition of cultures, including custom cultures.
- It enables the compression of script files over HTTP.
- It caches scripts to efficiently manage many requests.
- It adds a layer of indirection to scripts by piping them through an encrypted URL.

Script references can be added to the ScriptManager control either programmatically or by declarative markup. Declarative markup is particularly useful when working with scripts embedded in assemblies other than the web site project itself, as the name of the script will likely not change as revisions are pushed through.

Summary

As web applications grow to reach a larger audience, the need to be able to reach broader cultures and communities becomes core to a business model; e-commerce web applications need to be able to deal with foreign currencies, content management systems need to be able to not only present their content but also their navigation hints and form fields in other languages, and companies need to know that this need is accessible.

The .NET Framework intrinsically supports a rich localization framework, utilizing satellite assemblies and XML resource (.resx) files to present a uniform way to look up resource strings and images. The ASP.NET AJAX Extensions, including the Microsoft AJAX Framework and Microsoft AJAX Script Library, provide support for this programming model into client-side code, enabling easy resource string lookups. Satellite assemblies support the automatic inclusion of script resources (actual .js files) through ScriptResource.axd as long as the filenames follow a given naming scheme. With this support, the ASP.NET AJAX Extensions simplify the localization of scripts and the globalization of applications.

Bio

Scott Cate has been working with Microsoft Web technologies since 1997 and is the President of myKB.com (www.myKB.com) where he specializes in writing ASP.NET based applications focused on Knowledge Base Software solutions. Scott can be contacted via email at scott.cate@myKB.com or his blog at ScottCate.com

PREVIOUS NEXT

Understanding ASP.NET AJAX Web Services

2/20/2020 • 25 minutes to read • Edit Online

by Scott Cate

Download PDF

> Web Services are an integral part of the .NET framework that provide a cross-platform solution for exchanging data between distributed systems. Although Web Services are normally used to allow different operating systems, object models and programming languages to send and receive data, they can also be used to dynamically inject data into an ASP.NET AJAX page or send data from a page to a back-end system. All of this can be done without resorting to postback operations.

Calling Web Services with ASP.NET AJAX

Dan Wahlin

Web Services are an integral part of the .NET framework that provide a cross-platform solution for exchanging data between distributed systems. Although Web Services are normally used to allow different operating systems, object models and programming languages to send and receive data, they can also be used to dynamically inject data into an ASP.NET AJAX page or send data from a page to a back-end system. All of this can be done without resorting to postback operations.

While the ASP.NET AJAX UpdatePanel control provides a simple way to AJAX enable any ASP.NET page, there may be times when you need to dynamically access data on the server without using an UpdatePanel. In this article you'll see how to accomplish this by creating and consuming Web Services within ASP.NET AJAX pages.

This article will focus on functionality available in the core ASP.NET AJAX Extensions as well as a Web Service enabled control in the ASP.NET AJAX Toolkit called the AutoCompleteExtender. Topics covered include defining AJAX-enabled Web Services, creating client proxies and calling Web Services with JavaScript. You'll also see how Web Service calls can be made directly to ASP.NET page methods.

Web Services Configuration

When a new Web Site project is created with Visual Studio 2008, the web.config file has a number of new additions that may be unfamiliar to users of previous versions of Visual Studio. Some of these modifications map the "asp" prefix to ASP.NET AJAX controls so they can be used in pages while others define required HttpHandlers and HttpModules. Listing 1 shows modifications made to the `<httpHandlers>` element in web.config that affects Web Service calls. The default HttpHandler used to process .asmx calls is removed and replaced with a ScriptHandlerFactory class located in the System.Web.Extensions.dll assembly. System.Web.Extensions.dll contains all of the core functionality used by ASP.NET AJAX.

Listing 1. ASP.NET AJAX Web Service Handler Configuration

```
<httpHandlers>
     <remove verb="*" path="*.asmx"/>
     <add verb="*" path="*.asmx" validate="false"
          type="System.Web.Script.Services.ScriptHandlerFactory,
          System.Web.Extensions, Version=1.0.61025.0, Culture=neutral,
          PublicKeyToken=31bf3856ad364e35"/>
</httpHandlers>
```

This HttpHandler replacement is made in order to allow JavaScript Object Notation (JSON) calls to be made from ASP.NET AJAX pages to .NET Web Services using a JavaScript Web Service proxy. ASP.NET AJAX sends JSON messages to Web Services as opposed to the standard Simple Object Access Protocol (SOAP) calls typically associated with Web Services. This results in smaller request and response messages overall. It also allows for more efficient client-side processing of data since the ASP.NET AJAX JavaScript library is optimized to work with JSON objects. Listing 2 and Listing 3 show examples of Web Service request and response messages serialized to JSON format. The request message shown in Listing 2 passes a country parameter with a value of "Belgium" while the response message in Listing 3 passes an array of Customer objects and their associated properties.

Listing 2. Web Service Request Message Serialized to JSON

```
{"country":"Belgium"}
```

> *[!NOTE] the operation name is defined as part of the URL to the web service; additionally, request messages are not always submitted via JSON. Web Services can utilize the ScriptMethod attribute with the UseHttpGet parameter set to true, which causes parameters to be passed via a the query string parameters.*

Listing 3. Web Service Response Message Serialized to JSON

```
[{"__type":"Model.Customer","Country":"Belgium","CompanyName":"Maison
     Dewey","CustomerID":"MAISD","ContactName":"Catherine
     Dewey"},{"__type":"Model.Customer","Country":"Belgium","CompanyName":"Suprêmes
     délices","CustomerID":"SUPRD","ContactName":"Pascale
     Cartrain"}]
```

In the next section you'll see how to create Web Services capable of handling JSON request messages and responding with both simple and complex types.

Creating AJAX-Enabled Web Services

The ASP.NET AJAX framework provides several different ways to call Web Services. You can use the AutoCompleteExtender control (available in the ASP.NET AJAX Toolkit) or JavaScript. However, before calling a service you have to AJAX-enable it so that it can be called by client-script code.

Whether or not you're new to ASP.NET Web Services, you'll find it straightforward to create and AJAX-enable services. The .NET framework has supported the creation of ASP.NET Web Services since its initial release in 2002 and the ASP.NET AJAX Extensions provide additional AJAX functionality that builds upon the .NET framework's default set of features. Visual Studio .NET 2008 Beta 2 has built-in support for creating .asmx Web Service files and automatically derives associated code beside classes from the System.Web.Services.WebService class. As you add methods into the class you must apply the WebMethod attribute in order for them to be called by Web Service consumers.

Listing 4 shows an example of applying the WebMethod attribute to a method named GetCustomersByCountry().

Listing 4. Using the WebMethod Attribute in a Web Service

```
[WebMethod]
public Customer[] GetCustomersByCountry(string country)
{
     return Biz.BAL.GetCustomersByCountry(country);
}
```

The GetCustomersByCountry() method accepts a country parameter and returns a Customer object array. The country value passed into the method is forwarded to a business layer class which in turn calls a data layer class

to retrieve the data from the database, fill the Customer object properties with data and return the array.

Using the ScriptService Attribute

While adding the WebMethod attribute allows the GetCustomersByCountry() method to be called by clients that send standard SOAP messages to the Web Service, it doesn't allow JSON calls to be made from ASP.NET AJAX applications out of the box. To allow JSON calls to be made you have to apply the ASP.NET AJAX Extension's `ScriptService` attribute to the Web Service class. This enables a Web Service to send response messages formatted using JSON and allows client-side script to call a service by sending JSON messages.

Listing 5 shows an example of applying the ScriptService attribute to a Web Service class named CustomersService.

Listing 5. Using the ScriptService Attribute to AJAX-enable a Web Service

```
[System.Web.Script.Services.ScriptService]
[WebService(Namespace = "http://xmlforasp.net")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class CustomersService : System.Web.Services.WebService
{
     [WebMethod]
     public Customer[] GetCustomersByCountry(string country)
     {
          return Biz.BAL.GetCustomersByCountry(country);
     }
}
```

The ScriptService attribute acts as a marker that indicates it can be called from AJAX script code. It doesn't actually handle any of the JSON serialization or deserialization tasks that occur behind the scenes. The ScriptHandlerFactory (configured in web.config) and other related classes do the bulk of JSON processing.

Using the ScriptMethod Attribute

The ScriptService attribute is the only ASP.NET AJAX attribute that has to be defined in a .NET Web Service in order for it to be used by ASP.NET AJAX pages. However, another attribute named ScriptMethod can also be applied directly to Web Methods in a service. ScriptMethod defines three properties including `UseHttpGet`, `ResponseFormat` and `XmlSerializeString`. Changing the values of these properties can be useful in cases where the type of request accepted by a Web Method needs to be changed to GET, when a Web Method needs to return raw XML data in the form of an `XmlDocument` or `XmlElement` object or when data returned from a service should always be serialized as XML instead of JSON.

The UseHttpGet property can be used when a Web Method should accept GET requests as opposed to POST requests. Requests are sent using a URL with Web Method input parameters converted to QueryString parameters. The UseHttpGet property defaults to false and should only be set to `true` when operations are known to be safe and when sensitive data is not passed to a Web Service. Listing 6 shows an example of using the ScriptMethod attribute with the UseHttpGet property.

Listing 6. Using the ScriptMethod attribute with the UseHttpGet property.

```
[WebMethod]
[ScriptMethod(UseHttpGet = true)]
public string HttpGetEcho(string input)
{
     return input;
}
```

An example of the headers sent when the HttpGetEcho Web Method shown in Listing 6 is called are shown next:

```
GET /CustomerViewer/DemoService.asmx/HttpGetEcho?input=%22Input Value%22 HTTP/1.1
```

In addition to allowing Web Methods to accept HTTP GET requests, the ScriptMethod attribute can also be used when XML responses need to be returned from a service rather than JSON. For example, a Web Service may retrieve an RSS feed from a remote site and return it as an XmlDocument or XmlElement object. Processing of the XML data can then occur on the client.

Listing 7 shows an example of using the ResponseFormat property to specify that XML data should be returned from a Web Method.

Listing 7. Using the ScriptMethod attribute with the ResponseFormat property.

```
[WebMethod]
[ScriptMethod(ResponseFormat = ResponseFormat.Xml)]
public XmlElement GetRssFeed(string url)
{
     XmlDocument doc = new XmlDocument();
     doc.Load(url);
     return doc.DocumentElement;
}
```

The ResponseFormat property can also be used along with the XmlSerializeString property. The XmlSerializeString property has a default value of false which means that all return types except strings returned from a Web Method are serialized as XML when the `ResponseFormat` property is set to `ResponseFormat.Xml`. When `XmlSerializeString` is set to `true`, all types returned from a Web Method are serialized as XML including string types. If the ResponseFormat property has a value of `ResponseFormat.Json` the XmlSerializeString property is ignored.

Listing 8 shows an example of using the XmlSerializeString property to force strings to be serialized as XML.

Listing 8. Using the ScriptMethod attribute with the XmlSerializeString property

```
[WebMethod]
[ScriptMethod(ResponseFormat = ResponseFormat.Xml,XmlSerializeString=true)]
public string GetXmlString(string input)
{
     return input;
}
```

The value returned from calling the GetXmlString Web Method shown in Listing 8 is shown next:

```
<?xml version="1.0"?>
     <string>Test</string>
```

Although the default JSON format minimizes the overall size of request and response messages and is more readily consumed by ASP.NET AJAX clients in a cross-browser manner, the ResponseFormat and XmlSerializeString properties can be utilized when client applications such as Internet Explorer 5 or higher expect XML data to be returned from a Web Method.

Working with Complex Types

Listing 5 showed an example of returning a complex type named Customer from a Web Service. The Customer class defines several different simple types internally as properties such as FirstName and LastName. Complex types used as an input parameter or return type on an AJAX-enabled Web Method are automatically serialized into JSON before being sent to the client-side. However, nested complex types (those defined internally within another type) are not made available to the client as standalone objects by default.

In cases where a nested complex type used by a Web Service must also be used in a client page, the ASP.NET AJAX GenerateScriptType attribute can be added to the Web Service. For example, the CustomerDetails class shown in Listing 9 contains Address and Gender properties which ***represent nested complex types.***

Listing 9. The CustomerDetails class shown here contains two nested complex types.

```
public class CustomerDetails : Customer
{
     public CustomerDetails()
     {
     }
     Address _Address;
     Gender _Gender = Gender.Unknown;
     public Address Address
     {
          get { return _Address; }
          set { _Address = value; }
     }
     public Gender Gender
     {
          get { return _Gender; }
          set { _Gender = value; }
     }
}
```

The Address and Gender objects defined within the CustomerDetails class shown in Listing 9 won't automatically be made available for use on the client-side via JavaScript since they are nested types (Address is a class and Gender is an enumeration). In situations where a nested type used within a Web Service must be available on the client-side, the GenerateScriptType attribute mentioned earlier can be used (see Listing 10). This attribute can be added multiple times in cases where different nested complex types are returned from a service. It can be applied directly to the Web Service class or above specific Web Methods.

Listing 10. Using the GenerateScriptService attribute to define nested types that should be available to the client.

```
[System.Web.Script.Services.ScriptService]
[System.Web.Script.Services.GenerateScriptType(typeof(Address))]
[System.Web.Script.Services.GenerateScriptType(typeof(Gender))]
[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class NestedComplexTypeService : System.Web.Services.WebService
{
     //Web Methods
}
```

By applying the `GenerateScriptType` attribute to the Web Service, the Address and Gender types will automatically be made available for use by client-side ASP.NET AJAX JavaScript code. An example of the JavaScript that is automatically generated and sent to the client by adding the GenerateScriptType attribute on a Web Service is shown in Listing 11. You'll see how to use nested complex types later in the article.

Listing 11. Nested complex types made available to an ASP.NET AJAX page.

```
if (typeof(Model.Address) === 'undefined')
{
     Model.Address=gtc("Model.Address");
     Model.Address.registerClass('Model.Address');
}
Model.Gender = function() { throw Error.invalidOperation(); }
Model.Gender.prototype = {Unknown: 0,Male: 1,Female: 2}
Model.Gender.registerEnum('Model.Gender', true);
```

Now that you've seen how to create Web Services and make them accessible to ASP.NET AJAX pages, let's take a look at how to create and use JavaScript proxies so that data can be retrieved or sent to Web Services.

Creating JavaScript Proxies

Calling a standard Web Service (.NET or another platform) typically involves creating a proxy object that shields you from the complexities of sending SOAP request and response messages. With ASP.NET AJAX Web Service calls, JavaScript proxies can be created and used to easily call services without worrying about serializing and deserializing JSON messages. JavaScript proxies can be automatically generated by using the ASP.NET AJAX ScriptManager control.

Creating a JavaScript proxy that can call Web Services is accomplished by using the ScriptManager's Services property. This property allows you to define one or more services that an ASP.NET AJAX page can call asynchronously to send or receive data without requiring postback operations. You define a service by using the ASP.NET AJAX `ServiceReference` control and assigning the Web Service URL to the control's `Path` property. Listing 12 shows an example of referencing a service named CustomersService.asmx.

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
     <Services>
          <asp:ServiceReference Path="~/CustomersService.asmx" />
     </Services>
</asp:ScriptManager>
```

Listing 12. Defining a Web Service used in an ASP.NET AJAX page.

Adding a reference to the CustomersService.asmx through the ScriptManager control causes a JavaScript proxy to be dynamically generated and referenced by the page. The proxy is embedded by using the <script> tag and dynamically loaded by calling the CustomersService.asmx file and appending /js to the end of it. The following example shows how the JavaScript proxy is embedded in the page when debugging is disabled in web.config:

```
<script src="CustomersService.asmx/js" type="text/javascript"></script>
```

> *[!NOTE] If you'd like to see the actual JavaScript proxy code that is generated you can type the URL to the desired .NET Web Service into Internet Explorer's address box and append /js to the end of it.*

If debugging is enabled in web.config a debug version of the JavaScript proxy will be embedded in the page as shown next:

```
<script src="CustomersService.asmx/jsdebug" type="text/javascript"></script>
```

The JavaScript proxy created by the ScriptManager can also be embedded directly into the page rather than referenced using the <script> tag's src attribute. This can be done by setting the ServiceReference control's InlineScript property to true (the default is false). This can be useful when a proxy isn't shared across multiple pages and when you'd like to reduce the number of network calls made to the server. When InlineScript is set to

true the proxy script won't be cached by the browser so the default value of false is recommended in cases where the proxy is used by multiple pages in an ASP.NET AJAX application. An example of using the InlineScript property is shown next:

```
<asp:ServiceReference InlineScript="true" Path="~/CustomersService.asmx"/>
```

Using JavaScript Proxies

Once a Web Service is referenced by an ASP.NET AJAX page using the ScriptManager control, a call can be made to the Web Service and the returned data can be handled using callback functions. A Web Service is called by referencing its namespace (if one exists), class name and Web Method name. Any parameters passed to the Web Service can be defined along with a callback function that handles the returned data.

An example of using a JavaScript proxy to call a Web Method named GetCustomersByCountry() is shown in Listing 13. The GetCustomersByCountry() function is called when an end user clicks a button on the page.

Listing 13. Calling a Web Service with a JavaScript proxy.

```
function GetCustomerByCountry()
{
     var country = $get("txtCountry").value;
     InterfaceTraining.CustomersService.GetCustomersByCountry(country, OnWSRequestComplete);
}
function OnWSRequestComplete(results)
{
     if (results != null)
     {
          CreateCustomersTable(results);
          GetMap(results);
     }
}
```

This call references the InterfaceTraining namespace, CustomersService class and GetCustomersByCountry Web Method defined in the service. It passes a country value obtained from a textbox as well as a callback function named OnWSRequestComplete that should be invoked when the asynchronous Web Service call returns. OnWSRequestComplete handles the array of Customer objects returned from the service and converts them into a table that is displayed in the page. The output generated from the call is shown in Figure 1.

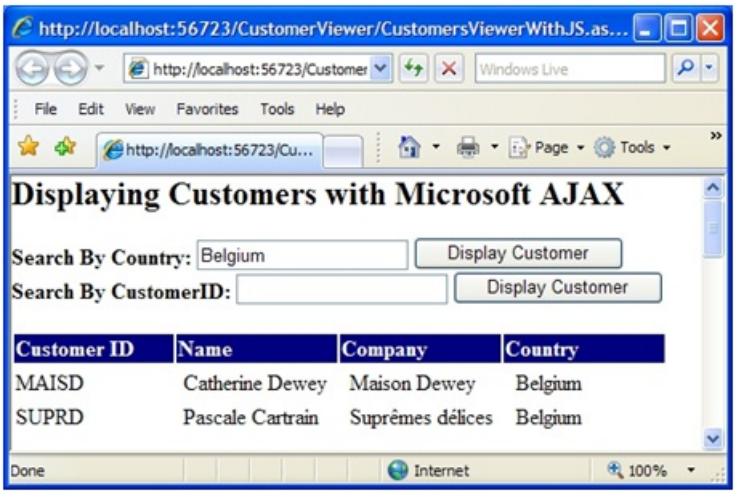


Figure 1: Binding data obtained by making an asynchronous AJAX call to a Web Service. (Click to view full-size image)

JavaScript proxies can also make one-way calls to Web Services in cases where a Web Method should be called

but the proxy shouldn't wait for a response. For example, you may want to call a Web Service to start a process such as a work-flow but not wait for a return value from the service. In cases where a one-way call needs to be made to a service, the callback function shown in Listing 13 can simply be omitted. Since no callback function is defined the proxy object will not wait for the Web Service to return data.

Handling Errors

Asynchronous callbacks to Web Services can encounter different types of errors such as the network being down, the Web Service being unavailable or an exception being returned. Fortunately, JavaScript proxy objects generated by the ScriptManager allow multiple callbacks to be defined to handle errors and failures in addition to the success callback shown earlier. An error callback function can be defined immediately after the standard callback function in the call to the Web Method as shown in Listing 14.

Listing 14. Defining an error callback function and displaying errors.

```
function GetCustomersByCountry()
{
     var country = $get("txtCountry").value;
     InterfaceTraining.CustomersService.GetCustomersByCountry(country,
          OnWSRequestComplete, OnWSRequestFailed);
}
function OnWSRequestFailed(error)
{
     alert("Stack Trace: " + error.get_stackTrace() + "/r/n" +
          "Error: " + error.get_message() + "/r/n" +
          "Status Code: " + error.get_statusCode() + "/r/n" +
          "Exception Type: " + error.get_exceptionType() + "/r/n" +
          "Timed Out: " + error.get_timedOut());
}
```

Any errors that occur when the Web Service is called will trigger the OnWSRequestFailed() callback function to be called which accepts an object representing the error as a parameter. The error object exposes several different functions to determine the cause of the error as well as whether or not the call timed out. Listing 14 shows an example of using the different error functions and Figure 2 shows an example of the output generated by the functions.

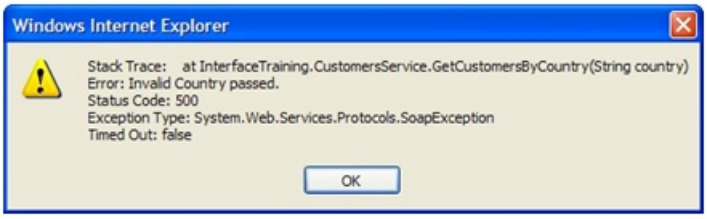


Figure 2: Output generated by calling ASP.NET AJAX error functions. (Click to view full-size image)

Handling XML Data Returned from a Web Service

Earlier you saw how a Web Method could return raw XML data by using the ScriptMethod attribute along with its ResponseFormat property. When ResponseFormat is set to ResponseFormat.Xml, data returned from the Web Service is serialized as XML rather than JSON. This can be useful when XML data needs to be passed directly to the client for processing using JavaScript or XSLT. At the current time, Internet Explorer 5 or higher provides the best client-side object model for parsing and filtering XML data due to its built-in support for MSXML.

Retrieving XML data from a Web Service is no different than retrieving other data types. Start by invoking the JavaScript proxy to call the appropriate function and define a callback function. Once the call returns you can then process the data in the callback function.

Listing 15 shows an example of calling a Web Method named GetRssFeed() that returns an XmlElement object.

GetRssFeed() accepts a single parameter representing the URL for the RSS feed to retrieve.

Listing 15. Working with XML data returned from a Web Service.

```
function GetRss()
{
     InterfaceTraining.DemoService.GetRssFeed(
          "https://blogs.interfacett.com/dan-wahlins-blog/rss.xml",
          OnWSRequestComplete);
}
function OnWSRequestComplete(result)
{
     if (document.all) //Filter for IE DOM since other browsers are limited
     {
          var items = result.selectNodes("//item");
          for (var i=0;i<items.length;i++)
          {
               var title = items[i].selectSingleNode("title").text;
               var href = items[i].selectSingleNode("link").text;
               $get("divOutput").innerHTML +=
               "<a href='" + href + "'>" + title + "</a><br/>";
          }
     }
     else
     {
          $get("divOutput").innerHTML = "RSS only available in IE5+";
     }
}
```

This example passes a URL to an RSS feed and processes the returned XML data in the OnWSRequestComplete() function. OnWSRequestComplete() first checks to see if the browser is Internet Explorer to know whether or not the MSXML parser is available. If it is, an XPath statement is used to locate all <item> tags within the RSS feed. Each item is then iterated through and the associated <title> and <link> tags are located and processed to display each item's data. Figure 3 shows an example of the output generated from making an ASP.NET AJAX call through a JavaScript proxy to the GetRssFeed() Web Method.

Handling Complex Types

Complex types accepted or returned by a Web Service are automatically exposed through a JavaScript proxy. However, nested complex types are not directly accessible on the client-side unless the GenerateScriptType attribute is applied to the service as discussed earlier. Why would you want to use a nested complex type on the client-side?

To answer this question, assume that an ASP.NET AJAX page displays customer data and allows end users to update a customer's address. If the Web Service specifies that the Address type (a complex type defined within a CustomerDetails class) can be sent to the client then the update process can be divided into separate functions for better code re-use.

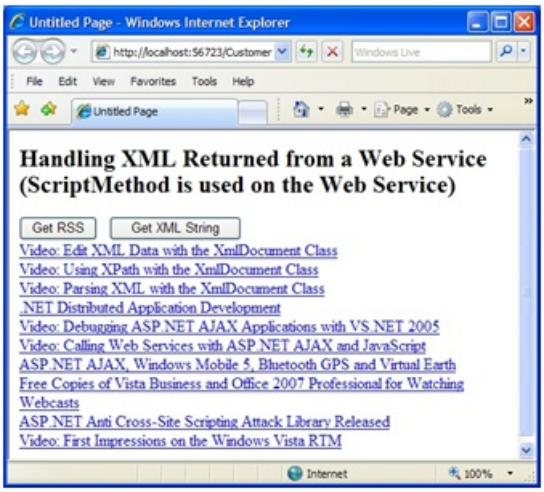


Figure 3: Output creating from calling a Web Service that returns RSS data. (Click to view full-size image)

Listing 16 shows an example of client-side code that invokes an Address object defined in a Model namespace, fills it with updated data and assigns it to a CustomerDetails object's Address property. The CustomerDetails object is then passed to the Web Service for processing.

Listing 16. Using nested complex types

```
function UpdateAddress()
{
     var cust = new Model.CustomerDetails();
     cust.CustomerID = $get("hidCustomerID").value;
     cust.Address = CreateAddress();
     InterfaceTraining.DemoService.UpdateAddress(cust,OnWSUpdateComplete);
}
function CreateAddress()
{
     var addr = new Model.Address();
     addr.Street = $get("txtStreet").value;
     addr.City = $get("txtCity").value;
     addr.State = $get("txtState").value;
     return addr;
}
function OnWSUpdateComplete(result)
{
     alert("Update " + ((result)?"succeeded":"failed")+ "!");
}
```

Creating and Using Page Methods

Web Services provide an excellent way to expose re-useable services to a variety of clients including ASP.NET AJAX pages. However, there may be cases where a page needs to retrieve data that won't ever be used or shared by other pages. In this case, making an .asmx file to allow the page to access the data may seem like overkill since the service is only used by a single page.

ASP.NET AJAX provides another mechanism for making Web Service-like calls without creating standalone .asmx files. This is done by using a technique referred to as "page methods". Page methods are static (shared in VB.NET) methods embedded directly in a page or code-beside file that have the WebMethod attribute applied to them. By applying the WebMethod attribute they can be called using a special JavaScript object named PageMethods that gets dynamically created at runtime. The PageMethods object acts as a proxy that shields you from the JSON serialization/deserialization process. Note that in order to use the PageMethods object you must set the ScriptManager's EnablePageMethods property to true.

```
<asp:ScriptManager ID="ScriptManager1" runat="server" EnablePageMethods="true">
</asp:ScriptManager>
```

Listing 17 shows an example of defining two page methods in an ASP.NET code-beside class. These methods retrieve data from a business layer class located in the App_Code folder of the Website.

Listing 17. Defining page methods.

```
[WebMethod]
public static Customer[] GetCustomersByCountry(string country)
{
     return Biz.BAL.GetCustomersByCountry(country);
}
[WebMethod]
public static Customer[] GetCustomersByID(string id)
{
     return Biz.BAL.GetCustomersByID(id);
}
```

When ScriptManager detects the presence of Web Methods in the page it generates a dynamic reference to the PageMethods object mentioned earlier. Calling a Web Method is accomplished by referencing the PageMethods class followed by the name of the method and any necessary parameter data that should be passed. Listing 18 shows examples of calling the two page methods shown earlier.

Listing 18. Calling page methods with the PageMethods JavaScript object.

```
function GetCustomerByCountry()
{
     var country = $get("txtCountry").value;
     PageMethods.GetCustomersByCountry(country, OnWSRequestComplete);
}
function GetCustomerByID()
{
     var custID = $get("txtCustomerID").value;
     PageMethods.GetCustomersByID(custID, OnWSRequestComplete);
}
function OnWSRequestComplete(results)
{
     var searchResults = $get("searchResults");
     searchResults.control.set_data(results);
     if (results != null) GetMap(results[0].Country,results);
}
```

Using the PageMethods object is very similar to using a JavaScript proxy object. You first specify all of the parameter data that should be passed to the page method and then define the callback function that should be called when the asynchronous call returns. A failure callback can also be specified (refer to Listing 14 for an example of handling failures).

The AutoCompleteExtender and the ASP.NET AJAX Toolkit

The ASP.NET AJAX Toolkit (available from http://ajax.asp.net) offers several controls that can be used to access Web Services. Specifically, the toolkit contains a useful control named `AutoCompleteExtender` that can be used to call Web Services and show data in pages without writing any JavaScript code at all.

The AutoCompleteExtender control can be used to extend existing functionality of a textbox and help users more easily locate data they're looking for. As they type into a textbox the control can be used to query a Web Service and shows results below the textbox dynamically. Figure 4 shows an example of using the AutoCompleteExtender control to display customer ids for a support application. As the user types different characters into the textbox,

different items will be shown below it based upon their input. Users can then select the desired customer id.

Using the AutoCompleteExtender within an ASP.NET AJAX page requires that the AjaxControlToolkit.dll assembly be added to the Website's bin folder. Once the toolkit assembly has been added, you'll want to reference it in web.config so that the controls it contains are available to all pages in an application. This can be done by adding the following tag within web.config's <controls> tag:

```
<add namespace="AjaxControlToolkit" assembly="AjaxControlToolkit" tagPrefix="ajaxToolkit"/>
```

In cases where you only need to use the control in a specific page you can reference it by adding the Reference directive to the top of a page as shown next rather than updating web.config:

```
<%@ Register Assembly="AjaxControlToolkit" Namespace="AjaxControlToolkit"
     TagPrefix="ajaxToolkit" %>
```

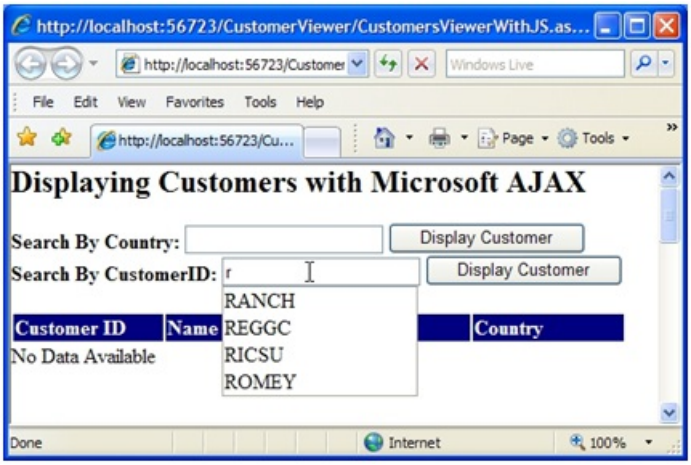


Figure 4: Using the AutoCompleteExtender control. (Click to view full-size image)

Once the Website has been configured to use the ASP.NET AJAX Toolkit, an AutoCompleteExtender control can be added into the page much like you'd add a regular ASP.NET server control. Listing 19 shows an example of using the control to call a Web Service.

Listing 19. Using the ASP.NET AJAX Toolkit AutoCompleteExtender control.

```
<ajaxToolkit:AutoCompleteExtender ID="extTxtCustomerID" runat="server"
     MinimumPrefixLength="1" ServiceMethod="GetCustomerIDs"
     ServicePath="~/CustomersService.asmx"
     TargetControlID="txtCustomerID" />
```

The AutoCompleteExtender has several different properties including the standard ID and runat properties found on server controls. In addition to these, it allows you to define how many characters an end user types before the Web Service is queried for data. The MinimumPrefixLength property shown in Listing 19 causes the service to be called each time a character is typed into the textbox. You'll want to be careful setting this value since each time the user types a character the Web Service will be called to search for values that match the characters in the textbox. The Web Service to call as well as the target Web Method are defined using the ServicePath and ServiceMethod properties respectively. Finally, the TargetControlID property identifies which textbox to hook the AutoCompleteExtender control to.

The Web Service being called must have the ScriptService attribute applied as discussed earlier and the target Web Method must accept two parameters named prefixText and count. The prefixText parameter represents the characters typed by the end user and the count parameter represents how many items to return (the default is 10). Listing 20 shows an example of the GetCustomerIDs Web Method called by the AutoCompleteExtender

control shown earlier in Listing 19. The Web Method calls a business layer method that in turn calls a data layer method that handles filtering the data and returning the matching results. The code for the data layer method is shown in Listing 21.

Listing 20. Filtering data sent from the AutoCompleteExtender control.

```
[WebMethod]
public string[] GetCustomerIDs(string prefixText, int count)
{
     return Biz.BAL.GetCustomerIDs(prefixText, count);
}
```

Listing 21. Filtering results based upon end user input.

```
public static string[] GetCustomerIDs(string prefixText, int count)
{
     //Customer IDs cached in _CustomerIDs field to improve performance
     if (_CustomerIDs == null)
     {
          List<string> ids = new List<string>();
          //SQL text used for simplicity...recommend using sprocs
          string sql = "SELECT CustomerID FROM Customers";
          DbConnection conn = GetDBConnection();
          conn.Open();
          DbCommand cmd = conn.CreateCommand();
          cmd.CommandText = sql;
          DbDataReader reader = cmd.ExecuteReader();
          while (reader.Read())
          {
               ids.Add(reader["CustomerID"].ToString());
          }
          reader.Close();
          conn.Close();
          _CustomerIDs = ids.ToArray();
     }
     int index = Array.BinarySearch(_CustomerIDs, prefixText, new CaseInsensitiveComparer());
     //~ is bitwise complement (reverse each bit)
     if (index < 0) index = ~index;
     int matchingCount;
     for (matchingCount = 0; matchingCount < count && index + matchingCount < _CustomerIDs.Length;
matchingCount++)
     {
          if (!_CustomerIDs[index + matchingCount].StartsWith(prefixText,
StringComparison.CurrentCultureIgnoreCase))
          {
               break;
          }
     }
     String[] returnValue = new string[matchingCount];
     if (matchingCount > 0)
     {
          Array.Copy(_CustomerIDs, index, returnValue, 0, matchingCount);
     }
     return returnValue;
}
```

Conclusion

ASP.NET AJAX provides excellent support for calling Web Services without writing a lot of custom JavaScript code to handle the request and response messages. In this article you've seen how to AJAX-enable .NET Web Services to enable them to process JSON messages and how to define JavaScript proxies using the ScriptManager control. You've also seen how JavaScript proxies can be used to call Web Services, handle simple and complex types and

deal with failures. Finally, you've seen how page methods can be used to simplify the process of creating and making Web Service calls and how the AutoCompleteExtender control can provide help to end users as they type. Although the UpdatePanel available in ASP.NET AJAX will certainly be the control of choice for many AJAX programmers due to its simplicity, knowing how to call Web Services through JavaScript proxies can be useful in many applications.

Bio

Dan Wahlin (Microsoft Most Valuable Professional for ASP.NET and XML Web Services) is a .NET development instructor and architecture consultant at Interface Technical Training (http://www.interfacett.com). Dan founded the XML for ASP.NET Developers Web site (www.XMLforASP.NET), is on the INETA Speaker's Bureau and speaks at several conferences. Dan co-authored Professional Windows DNA (Wrox), ASP.NET: Tips, Tutorials and Code (Sams), ASP.NET 1.1 Insider Solutions, Professional ASP.NET 2.0 AJAX (Wrox), ASP.NET 2.0 MVP Hacks and authored XML for ASP.NET Developers (Sams). When he's not writing code, articles or books, Dan enjoys writing and recording music and playing golf and basketball with his wife and kids.

Scott Cate has been working with Microsoft Web technologies since 1997 and is the President of myKB.com (www.myKB.com) where he specializes in writing ASP.NET based applications focused on Knowledge Base Software solutions. Scott can be contacted via email at scott.cate@myKB.com or his blog at ScottCate.com

PREVIOUS NEXT

Understanding ASP.NET AJAX Debugging Capabilities

2/20/2020 • 28 minutes to read • Edit Online

by Scott Cate

Download PDF

The ability to debug code is a skill that every developer should have in their arsenal regardless of the technology they're using. While many developers are accustomed to using Visual Studio .NET or Web Developer Express to debug ASP.NET applications that use VB.NET or C# code, some aren't aware that it's also extremely useful for debugging client-side code such as JavaScript. The same type of techniques used to debug .NET applications can also be applied to AJAX-enabled applications and more specifically ASP.NET AJAX applications.

Debugging ASP.NET AJAX Applications

Dan Wahlin

The ability to debug code is a skill that every developer should have in their arsenal regardless of the technology they're using. It goes without saying that understanding the different debugging options that are available can save a tremendous amount of time on a project and perhaps even a few headaches. While many developers are accustomed to using Visual Studio .NET or Web Developer Express to debug ASP.NET applications that use VB.NET or C# code, some aren't aware that it's also extremely useful for debugging client-side code such as JavaScript. The same type of techniques used to debug .NET applications can also be applied to AJAX-enabled applications and more specifically ASP.NET AJAX applications.

In this article you'll see how Visual Studio 2008 and several other tools can be used to debug ASP.NET AJAX applications to quickly locate bugs and other issues. This discussion will include information about enabling Internet Explorer 6 or higher for debugging, using Visual Studio 2008 and the Script Explorer to step through code as well as using other free tools such as Web Development Helper. You'll also learn how to debug ASP.NET AJAX applications in Firefox using an extension named Firebug which lets you step through JavaScript code directly in the browser without any other tools. Finally, you'll be introduced to classes in the ASP.NET AJAX Library that can help with various debugging tasks such as tracing and code assertion statements.

Before you try to debug pages viewed in Internet Explorer there are a few basic steps you'll need to perform to enable it for debugging. Let's take a look at some basic setup requirements that need to be performed to get started.

Configuring Internet Explorer for Debugging

Most people aren't interested in seeing JavaScript issues encountered on a Website viewed with Internet Explorer. In fact, the average user wouldn't even know what to do if they saw an error message. As a result, debugging options are turned off by default in the browser. However, it's very straightforward to turn debugging on and put it to use as you develop new AJAX applications.

To enable debugging functionality, go to Tools Internet Options on the Internet Explorer menu and select the Advanced tab. Within the Browsing section ensure that the following items are unchecked:

- Disable script debugging (Internet Explorer)

- Disable script debugging (Other)

Although not required, if you're trying to debug an application you'll probably want any JavaScript errors in the page to be immediately visible and obvious. You can force all errors to be shown with a message box by checking the "Display a notification about every script error" checkbox. While this is a great option to turn on while you're developing an application, it can quickly become annoying if you're just perusing other Websites since your chances of encountering JavaScript errors are pretty good.

Figure 1 shows what the Internet Explorer advanced dialog should look after it has been properly configured for debugging.

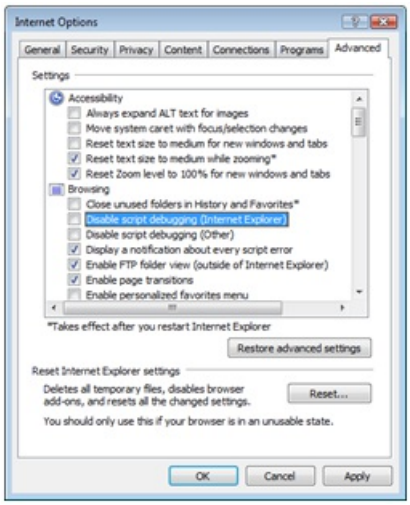


Figure 1: Configuring Internet Explorer for debugging. (Click to view full-size image)

Once debugging has been turned on, you'll see a new menu item appear in the View menu named Script Debugger. It has two options available including Open and Break at Next Statement. When Open is selected you'll be prompted to debug the page in Visual Studio 2008 (note that Visual Web Developer Express can also be used for debugging). If Visual Studio .NET is currently running you can choose to use that instance or to create a new instance. When Break at Next Statement is selected you'll be prompted to debug the page when JavaScript code is executed. If JavaScript code executes in the onLoad event of the page you can refresh the page to trigger a debug session. If JavaScript code is run after a button is clicked then the debugger will run immediately after the button is clicked.

> **NOTE**
>
> If you're running on Windows Vista with User Access Control (UAC) enabled, and you have Visual Studio 2008 set to run as an administrator, Visual Studio will fail to attach to the process when you are prompted to attach. To work around this issue, start Visual Studio first, and use that instance to debug.

Although the next section will demonstrate how to debug an ASP.NET AJAX page directly from within Visual Studio 2008, using the Internet Explorer Script Debugger option is useful when a page is already open and you'd like to more fully inspect it.

Debugging with Visual Studio 2008

Visual Studio 2008 provides debugging functionality that developers around the world rely on everyday to debug .NET applications. The built-in debugger allows you to step through code, view object data, watch for specific variables, monitor the call stack plus much more. In addition to debugging VB.NET or C# code, the debugger is also helpful for debugging ASP.NET AJAX applications and will allow you to step through JavaScript code line by line. The details that follow focus on techniques that can be used to debug client-side script files rather than providing a discourse on the overall process of debugging applications using Visual Studio 2008.

The process of debugging a page in Visual Studio 2008 can be started in several different ways. First, you can use the Internet Explorer Script Debugger option mentioned in the previous section. This works well when a page is already loaded in the browser and you'd like to start debugging it. Alternatively, you can right-click on an .aspx page in the Solution Explorer and select Set As Start Page from the menu. If you're accustomed to debugging ASP.NET pages then you've probably done this before. Once F5 is pressed the page can be debugged. However, while you can generally set a breakpoint anywhere you'd like in VB.NET or C# code, that's not always the case with JavaScript as you'll see next.

Embedded Versus External Scripts

The Visual Studio 2008 debugger treats JavaScript embedded in a page different than external JavaScript files. With external script files, you can open the file and set a breakpoint on any line you choose. Breakpoints can be set by clicking in the grey tray area to the left of the code editor window. When JavaScript is embedded directly into a page using the `<script>` tag, setting a breakpoint by clicking in the grey tray area isn't an option. Attempts to set a breakpoint on a line of embedded script will result in a warning that states "This is not a valid location for a breakpoint".

You can get around this issue by moving the code into an external .js file and referencing it using the src attribute of the <script> tag:

```
<script type="text/javascript" src="Scripts/YourScript.js"></script>
```

What if moving the code into an external file isn't an option or requires more work than it's worth? While you can't set a breakpoint using the editor, you can add the debugger statement directly into the code where you'd like to start debugging. You can also use the Sys.Debug class available in the ASP.NET AJAX library to force debugging to start. You'll learn more about the Sys.Debug class later in this article.

An example of using the `debugger` keyword is shown in Listing 1. This example forces the debugger to break right before a call to an update function is made.

Listing 1. Using the debugger keyword to force the Visual Studio .NET debugger to break.

```
function BuildPerson()
{
 var person =
 {
 FirstName: $get("txtFirstName").value,
 LastName: $get("txtLastName").value,
 Address:
 {
 Street: $get("txtStreet").value,
 City: $get("txtCity").value,
 State: $get("txtState").value
 }
 };
 debugger;
 UpdatePerson(person);
}
```

Once the debugger statement is hit you will be prompted to debug the page using Visual Studio .NET and can begin stepping through the code. While doing this you may encounter an issue with accessing ASP.NET AJAX library script files used in the page so let's take a look at using Visual Studio .NET's Script Explorer.

Using Visual Studio .NET Windows to Debug

Once a debug session is started and you begin walking through code using the default F11 key, you may encounter the error dialog shown in see Figure 2 unless all script files used in the page are open and available for

debugging.

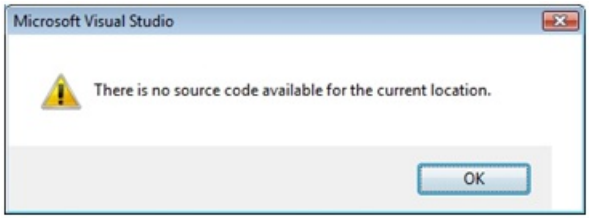


Figure 2: Error dialog shown when no source code is available for debugging. (Click to view full-size image)

This dialog is shown because Visual Studio .NET isn't sure how to get to the source code of some of the scripts referenced by the page. While this can be quite frustrating at first, there's a simple fix. Once you have started a debug session and hit a breakpoint, go to the Debug Windows Script Explorer window on the Visual Studio 2008 menu or use the Ctrl+Alt+N hotkey.

> **NOTE**
>
> If you can't see the Script Explorer menu listed, go to **Tools** > **Customize** > **Commands** on the Visual Studio .NET menu. Locate the **Debug** entry in the Categories section and click it to show all available menu entries. In the Commands list, scroll down to Script Explorer and then drag it up onto the Debug Windows menu in mentioned earlier. Doing this will make the Script Explorer menu entry available each time you run Visual Studio .NET.

The Script Explorer can be used to view all scripts used in a page and open them in the code editor. Once the Script Explorer is open, double-click on the .aspx page currently being debugged to open it in the code editor window. Perform the same action for all of the other scripts shown in the Script Explorer. Once all of the scripts are open in the code window you can press F11 (and use the other debug hotkeys) to step through your code. Figure 3 shows an example of the Script Explorer. It lists the current file being debugged (Demo.aspx) as well as two custom scripts and two scripts dynamically injected into the page by the ASP.NET AJAX ScriptManager.

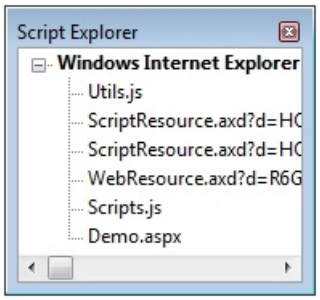


Figure 3. The Script Explorer provides easy access to scripts used in a page. (Click to view full-size image)

Several others windows can also be used to provide useful information as you step through code in a page. For example, you can use the Locals window to see the values of different variables used in the page, the Immediate window to evaluate specific variables or conditions and view the output. You can also use the Output window to view trace statements written out using the Sys.Debug.trace function (which will be covered later in this article) or Internet Explorer's Debug.writeln function.

As you step through code using the debugger you can mouse over variables in the code to view the value that they are assigned. However, the script debugger occasionally won't show anything as you mouse over a given JavaScript variable. To see the value, highlight the statement or variable you're trying to see in the code editor window and then mouse over it. Although this technique doesn't work in every situation, many times you will be able to see the value without having to look in a different debug window such as the Locals window.

A video tutorial demonstrating some of the features discussed here can be viewed at http://www.xmlforasp.net.

Debugging With Web Development Helper

Although Visual Studio 2008 (and Visual Web Developer Express 2008) are very capable debugging tools, there are additional options that can be used as well which are more light-weight. One of the latest tools to be released is the Web Development Helper. Microsoft's Nikhil Kothari (one of the key ASP.NET AJAX architects at Microsoft) wrote this excellent tool which can perform many different tasks from simple debugging to viewing HTTP request and response messages. Web Development Helper can be downloaded at http://projects.nikhilk.net/Projects/WebDevHelper.aspx.

Web Development helper can be used directly inside of Internet Explorer which makes it convenient to use. It's started by selecting Tools Web Development Helper from the Internet Explorer menu. This will open the tool in the bottom portion of the browser which is nice since you don't have to leave the browser to perform several tasks such as HTTP request and response message logging. Figure 4 shows what Web Development Helper looks like in action.

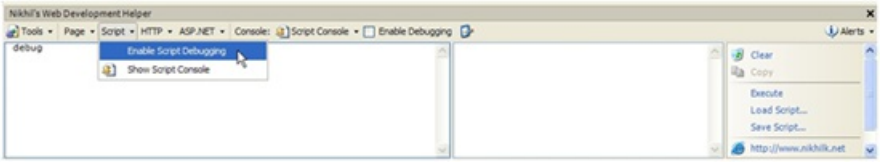


Figure 4: Web Development Helper (Click to view full-size image)

Web Development helper isn't a tool you'll use to step through code line by line as with Visual Studio 2008. However, it can be used to view trace output, easily evaluate variables in a script or explore data is inside of a JSON object. It's also very useful for viewing data that is passed to and from an ASP.NET AJAX page and a server.

Once Web Development Helper is open in Internet Explorer, script debugging must be enabled by selecting Script Enable Script Debugging from the Web Development helper menu as shown earlier in Figure 4. This enables the tool to intercept errors that occur as a page is run. It also allows easy access to trace messages that are output in the page. To view trace information or execute script commands to test different functions within a page, select Script Show Script Console from the Web Development Helper menu. This provides access to a command window and a simple immediate window.

Viewing Trace Messages and JSON Object Data

The immediate window can be used to execute script commands or even load or save scripts that are used to test different functions in a page. The command window displays trace or debug messages written out by the page being viewed. Listing 2 shows how to write a trace message using Internet Explorer's Debug.writeln function.

Listing 2. Writing out a client-side trace message using the Debug class.

```
function BuildPerson()
{
 var person =
 {
 FirstName: $get("txtFirstName").value,
 LastName: $get("txtLastName").value,
 Address:
 {
 Street: $get("txtStreet").value,
 City: $get("txtCity").value,
 State: $get("txtState").value
 }
 };
 Debug.writeln("Person name: " + person.LastName);
 UpdatePerson(person);
}
```

If the LastName property contains a value of Doe, Web Development Helper will display the message "Person name: Doe" in the script console's command window (assuming that debugging has been enabled). Web

Development Helper also adds a top-level debugService object into pages that can be used to write out trace information or view the content of JSON objects. Listing 3 shows an example of using the debugService class's trace function.

Listing 3. Using Web Development Helper's debugService class to write a trace message.

```
function BuildPerson()
{
 var person =
 {
 FirstName: $get("txtFirstName").value,
 LastName: $get("txtLastName").value,
 Address:
 {
 Street: $get("txtStreet").value,
 City: $get("txtCity").value,
 State: $get("txtState").value
 }
 };
 if (window.debugService)
 {
 window.debugService.trace("Person name: " + person.LastName);
 }
 UpdatePerson(person);
}
```

A nice feature of the debugService class is that it will work even if debugging isn't enabled in Internet Explorer making it easy to always access trace data when Web Development Helper is running. When the tool isn't being used to debug a page, trace statements will be ignored since the call to window.debugService will return false.

The debugService class also allows JSON object data to be viewed using Web Development Helper's inspector window. Listing 4 creates a simple JSON object containing person data. Once the object is created, a call is made to the debugService class's inspect function to allow the JSON object to be visually inspected.

Listing 4. Using the debugService.inspect function to view JSON object data.

```
function BuildPerson()
{
 var person =
 {
 FirstName: $get("txtFirstName").value,
 LastName: $get("txtLastName").value,
 Address:
 {
 Street: $get("txtStreet").value,
 City: $get("txtCity").value,
 State: $get("txtState").value
 }
 };
 if (window.debugService)
 {
 window.debugService.inspect("Person Object",person);
 }
 UpdatePerson(person);
}
```

Calling the GetPerson() function in the page or through the immediate window will result in the Object Inspector dialog window appearing as shown in Figure 5. Properties within the object can be changed dynamically by highlighting them, changing the value shown in the Value text box and then clicking the Update link. Using the Object Inspector makes it straightforward to view JSON object data and experiment with applying different values to properties.

Debugging Errors

In addition to allowing trace data and JSON objects to be displayed, Web Development helper can also aid in debugging errors in a page. If an error is encountered, you will be prompted to continue to the next line of code or debug the script (see Figure 6). The Script Error dialog window shows the complete call stack as well as line numbers so you can easily identify where issues are within a script.

Figure 5: Using the Object Inspector window to view a JSON object. (Click to view full-size image)

Selecting the debug option allows you to execute script statements directly in Web Development Helper's immediate window to view the value of variables, write out JSON objects, plus more. If the same action that triggered the error is performed again and Visual Studio 2008 is available on the machine, you will be prompted to start a debug session so that you can step through the code line by line as discussed in the previous section.

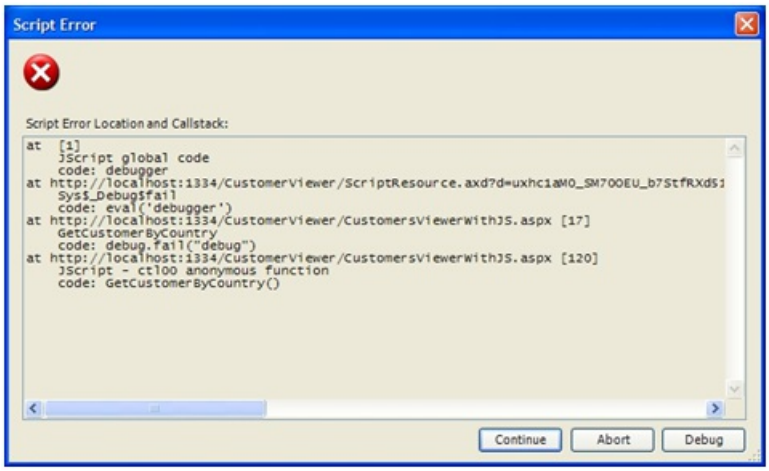


Figure 6: Web Development Helper's Script Error Dialog (Click to view full-size image)

Inspecting Request and Response Messages

While debugging ASP.NET AJAX pages it is often useful to see request and response messages sent between a page and server. Viewing the content within messages allows you to see if the proper data is being passed as well as the size of the messages. Web Development Helper provides an excellent HTTP message logger feature that makes it easy to view data as raw text or in a more readable format.

To view ASP.NET AJAX request and response messages, the HTTP logger must be enabled by selecting HTTP Enable HTTP Logging from the Web Development Helper menu. Once enabled, all messages sent from the current page can be viewed in the HTTP log viewer which can be accessed by selecting HTTP Show HTTP Logs.

Although viewing the raw text sent in each request/response message is certainly useful (and an option in Web Development Helper), it is often easier to view message data in a more graphical format. Once HTTP logging has been enabled and messages have been logged, message data can be viewed by double-clicking on the message in the HTTP log viewer. Doing this allows you to view all headers associated with a message as well as the actual message content. Figure 7 shows an example of a request message and response message viewed in the HTTP

Log Viewer window.

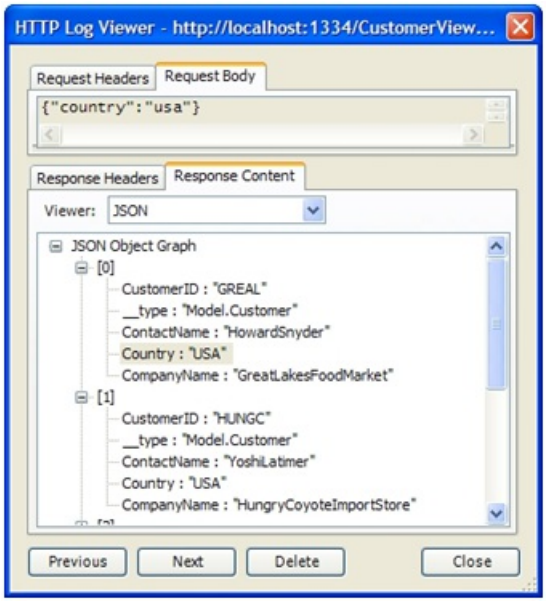


Figure 7: Using the HTTP Log Viewer to view request and response message data. (Click to view full-size image)

The HTTP Log Viewer automatically parses JSON objects and displays them using a tree view making it quick and easy to view the object's property data. When an UpdatePanel is being used in an ASP.NET AJAX page, the viewer breaks out each portion of the message into individual parts as shown in Figure 8. This is a great feature that makes it much easier to see and understand what is in the message as compared to viewing the raw message data.

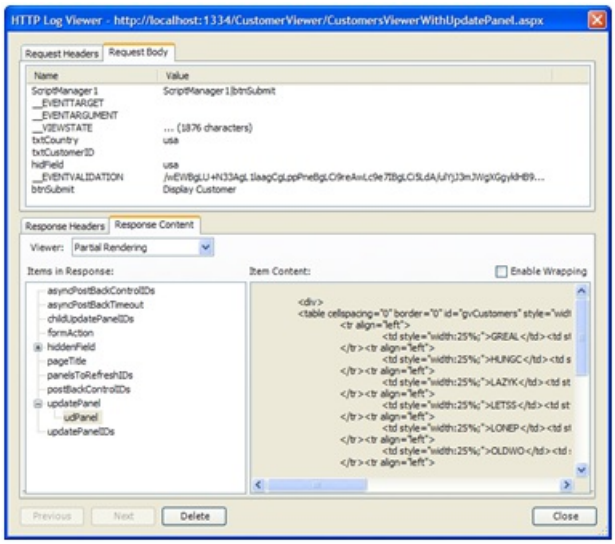


Figure 8: An UpdatePanel response message viewed using the HTTP Log Viewer. (Click to view full-size image)

There are several other tools that can be used to view request and response messages in addition to Web Development Helper. Another good option is Fiddler which is available for free at http://www.fiddlertool.com. Although Fiddler will not be discussed here, it is also a good option when you need to thoroughly inspect message headers and data.

Debugging with Firefox and Firebug

While Internet Explorer is still the most widely used browser, other browsers such as Firefox have become quite popular and are being used more and more. As a result, you'll want to view and debug your ASP.NET AJAX pages in Firefox as well as Internet Explorer to ensure that your applications work properly. Although Firefox can't tie directly into Visual Studio 2008 for debugging, it has an extension called Firebug that can be used to debug pages. Firebug can be downloaded for free by going to http://www.getfirebug.com.

Firebug provides a full-featured debugging environment that can be used to step through code line by line, access all scripts used within a page, view DOM structures, display CSS styles and even track events that occur in a page. Once installed, Firebug can be accessed by selecting Tools Firebug Open Firebug from the Firefox menu. Like Web Development Helper, Firebug is used directly in the browser although it can also be used as a stand-alone application.

Once Firebug is running, breakpoints can be set on any line of a JavaScript file whether the script is embedded in a page or not. To set a breakpoint, first load the appropriate page you'd like to debug in Firefox. Once the page is loaded, select the script to debug from Firebug's Scripts drop-down list. All scripts used by the page will be shown. A breakpoint is set by clicking in Firebug's grey tray area on the line where the breakpoint should go must like you would do in Visual Studio 2008.

Once a breakpoint has been set in Firebug you can perform the action required to execute the script that needs to be debugged such as clicking a button or refreshing the browser to trigger the onLoad event. Execution will automatically stop on the line containing the breakpoint. Figure 9 shows an example of a breakpoint that has been triggered in Firebug.

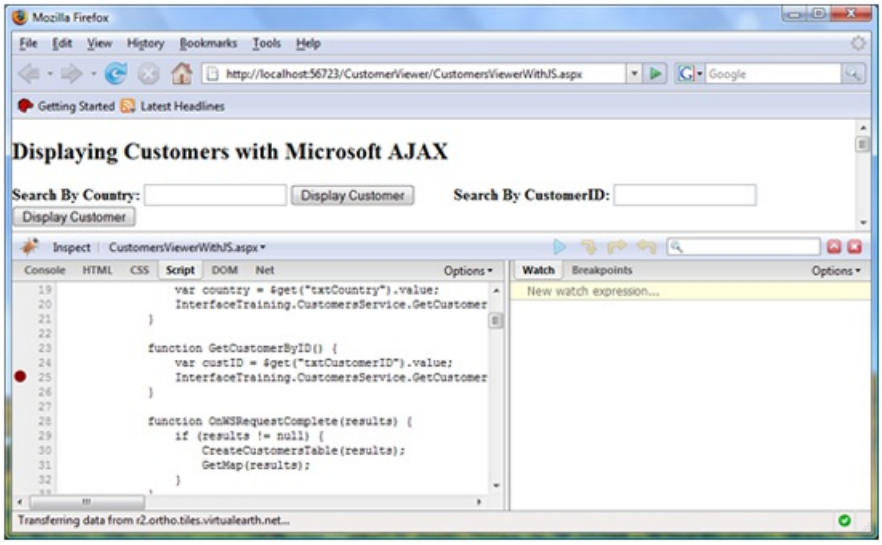


Figure 9: Handling breakpoints in Firebug. (Click to view full-size image)

Once a breakpoint is hit you can step into, step over or step out of code using the arrow buttons. As you step through code, script variables are displayed in the right-hand portion of the debugger allowing you to see values and drill-down into objects. Firebug also includes a Call Stack drop-down list to view the script's execution steps that led up to the current line being debugged.

Firebug also includes a console window that can be used to test different script statements, evaluate variables and view trace output. It is accessed by clicking on the Console tab at the top of the Firebug window. The page being debugged can also be "inspected" to see its DOM structure and contents by clicking on the Inspect tab. As you mouse over the different DOM elements shown in the inspector window the appropriate portion of the page will be highlighted making it easy to see where the element is used in the page. Attribute values associated with a given element can be changed "live" to experiment with applying different widths, styles, etc. to an element. This is a nice feature that saves you from having to constantly switch between the source code editor and the Firefox browser to view how simple changes affect a page.

Figure 10 shows an example of using the DOM inspector to locate a textbox named txtCountry in the page. The Firebug inspector can also be used to view CSS styles used in a page as well as events that occur such as tracking mouse movements, button clicks, plus more.

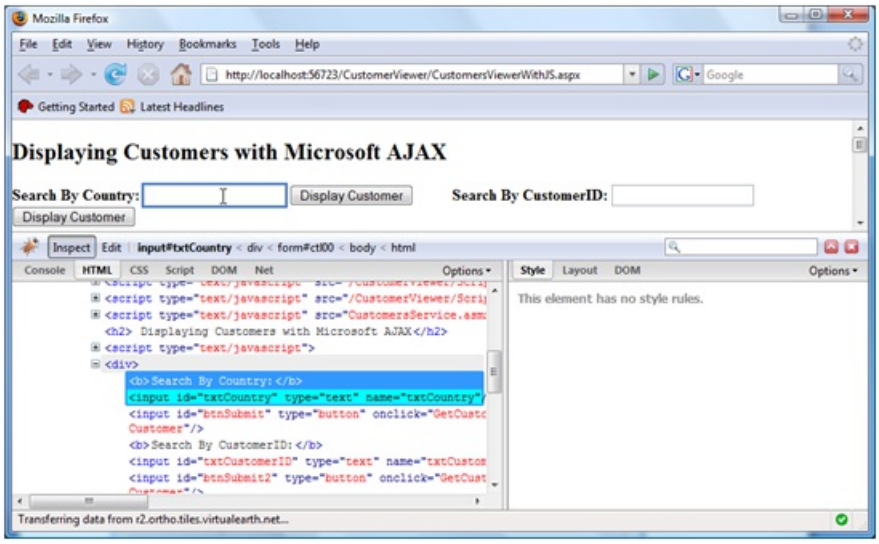


Figure 10: Using Firebug's DOM inspector. (Click to view full-size image)

Firebug provides a light-weight way to quickly debug a page directly in Firefox as well as an excellent tool for inspecting different elements within the page.

Debugging Support in ASP.NET AJAX

The ASP.NET AJAX library includes many different classes that can be used to simplify the process of adding AJAX capabilities into a Webpage. You can use these classes to locate elements within a page and manipulate them, add new controls, call Web Services and even handle events. The ASP.NET AJAX library also contains classes that can be used to enhance the process of debugging pages. In this section you'll be introduced to the Sys.Debug class and see how it can be used in applications.

Using the Sys.Debug class

The Sys.Debug class (a JavaScript class located in the Sys namespace) can be used to perform several different functions including writing trace output, performing code assertions and forcing code to fail so that it can be debugged. It is used extensively in the ASP.NET AJAX library's debug files (installed at C:\Program Files\Microsoft ASP.NET\ASP.NET 2.0 AJAX Extensions\v1.0.61025\MicrosoftAjaxLibrary\System.Web.Extensions\1.0.61025.0 by default) to perform conditional tests (called assertions) that ensure parameters are passed properly to functions, that objects contain the expected data and to write trace statements.

The Sys.Debug class exposes several different functions that can be used to handle tracing, code assertions or failures as shown in Table 1.

Table 1. Sys.Debug class functions.

FUNCTION NAME	DESCRIPTION
assert(condition, message, displayCaller)	Asserts that the condition parameter is true. If the condition being tested is false, a message box will be used to display the message parameter value. If the displayCaller parameter is true, the method also displays information about the caller.
clearTrace()	Erases statements output from tracing operations.
fail(message)	Causes the program to stop execution and break into the debugger. The message parameter can be used to provide a reason for the failure.
trace(message)	Writes the message parameter to the trace output.

FUNCTION NAME	DESCRIPTION
traceDump(object, name)	Outputs an object's data in a readable format. The name parameter can be used to provide a label for the trace dump. Any sub-objects within the object being dumped will be written out by default.

Client-side tracing can be used in much the same way as the tracing functionality available in ASP.NET. It allows different messages to easily be seen without interrupting the flow of the application. Listing 5 shows an example of using the Sys.Debug.trace function to write to the trace log. This function simply takes the message that should be written out as a parameter.

Listing 5. Using the Sys.Debug.trace function.

```
function BuildPerson()
{
 var address = new XmlForAsp.Address($get("txtStreet").value, $get("txtCity").value, $get("txtState").value,
$get("txtZip").value);
 var person = new XmlForAsp.Person(null, $get("txtFirstName").value, $get("txtLastName").value, address);
 Sys.Debug.trace("Person's name: " + person.get_firstName() + " " + person.get_lastName());
 UpdatePerson(person);
}
```

If you execute the code shown in Listing 5 you won't see any trace output in the page. The only way to see it is to use a console window available in Visual Studio .NET, Web Development Helper or Firebug. If you do want to see the trace output in the page then you'll need to add a TextArea tag and give it an id of TraceConsole as shown next:

```
<textArea id="TraceConsole" rows="10" cols="50"></textArea>
```

Any Sys.Debug.trace statements in the page will be written to the TraceConsole TextArea.

In cases where you want to see the data contained within a JSON object you can use the Sys.Debug class's traceDump function. This function takes two parameters including the object that should be dumped to the trace console and a name that can be used to identify the object in the trace output. Listing 6 shows an example of using the traceDump function.

Listing 6. Using the Sys.Debug.traceDump function.

```
function UpdatePerson(person)
{
 //Dump contents of the person object to the trace output
 Sys.Debug.traceDump(person,"Person Data");

 alert("Person updated! " + person.get_firstName() + " " + person.get_lastName());
}
```

Figure 11 shows the output from calling the Sys.Debug.traceDump function. Notice that in addition to writing out the Person object's data, it also writes out the Address sub-object's data.

In addition to tracing, the Sys.Debug class can also be used to perform code assertions. Assertions are used to test that specific conditions are met while an application is running. The debug version of the ASP.NET AJAX library scripts contain several assert statements to test a variety of conditions.

Listing 7 shows an example of using the Sys.Debug.assert function to test a condition. The code tests whether or not the Address object is null before updating a Person object.

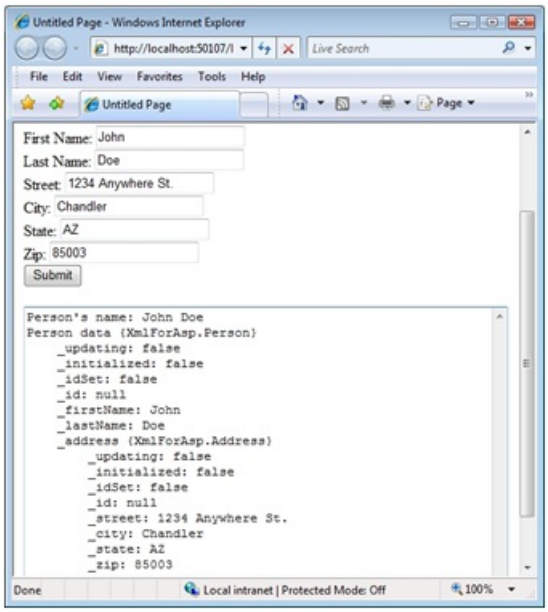


Figure 11: Output of the Sys.Debug.traceDump function. (Click to view full-size image)

Listing 7. Using the debug.assert function.

```
function UpdatePerson(person)
{
 //Check if address is null
 Sys.Debug.assert(person.get_address() == null,"Address is null!",true);

 alert("Person updated! " + person.get_firstName() + " " + person.get_lastName());
}
```

Three parameters are passed including the condition to evaluate, the message to display if the assertion returns false and whether or not information about the caller should be displayed. In cases where an assertion fails, the message will be displayed as well as caller information if the third parameter was true. Figure 12 shows an example of the failure dialog that appears if the assertion shown in Listing 7 fails.

The final function to cover is Sys.Debug.fail. When you want to force code to fail on a particular line in a script you can add a Sys.Debug.fail call rather than the debugger statement typically used in JavaScript applications. The Sys.Debug.fail function accepts a single string parameter that represents the reason for the failure as shown next:

```
Sys.Debug.fail("My forced failure of script.");
```

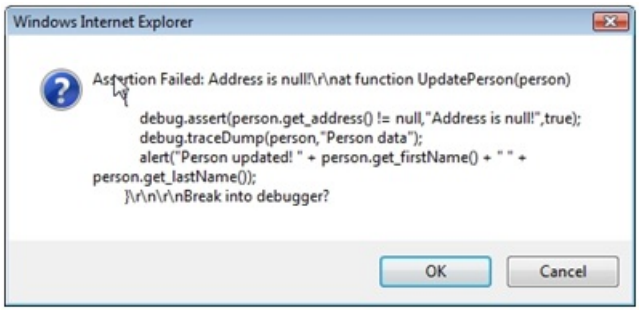


Figure 12: A Sys.Debug.assert failure message. (Click to view full-size image)

When a Sys.Debug.fail statement is encountered while a script is executing, the value of the message parameter will be displayed in the console of a debug application such as Visual Studio 2008 and you'll be prompted to debug the application. One case where this can be quite useful is when you can't set a breakpoint with Visual Studio 2008 on an inline script but would like the code to stop on particular line so you can inspect the value of

variables.

Understanding the ScriptManager Control's ScriptMode Property

The ASP.NET AJAX library includes debug and release script versions that are installed at C:\Program Files\Microsoft ASP.NET\ASP.NET 2.0 AJAX Extensions\v1.0.61025\MicrosoftAjaxLibrary\System.Web.Extensions\1.0.61025.0 by default. The debug scripts are nicely formatted, easy to read and have several Sys.Debug.assert calls scattered throughout them while the release scripts have whitespace stripped out and use the Sys.Debug class sparingly to minimize their overall size.

The ScriptManager control added to ASP.NET AJAX pages reads the compilation element's debug attribute in web.config to determine which versions of library scripts to load. However, you can control if debug or release scripts are loaded (library scripts or your own custom scripts) by changing the ScriptMode property. ScriptMode accepts a ScriptMode enumeration whose members include Auto, Debug, Release and Inherit.

ScriptMode defaults to a value of Auto which means that the ScriptManager will check the debug attribute in web.config. When debug is false the ScriptManager will load the release version of ASP.NET AJAX library scripts. When debug is true the debug version of the scripts will be loaded. Changing the ScriptMode property to Release or Debug will force the ScriptManager to load the appropriate scripts regardless of what value the debug attribute has in web.config. Listing 8 shows an example of using the ScriptManager control to load debug scripts from the ASP.NET AJAX library.

Listing 8. Loading debug scripts using the ScriptManager.

```
<asp:ScriptManager ID="ScriptManager1" runat="server" ScriptMode="Debug"></asp:ScriptManager>
```

You can also load different versions (debug or release) of your own custom scripts by using the ScriptManager's Scripts property along with the ScriptReference component as shown in Listing 9.

Listing 9. Loading custom scripts using the ScriptManager.

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
 <Scripts>
 <asp:ScriptReference Path="~/Scripts/Person.js" ScriptMode="Debug"/>
 </Scripts>
</asp:ScriptManager>
```

NOTE

If you're loading custom scripts using the ScriptReference component you must notify the ScriptManager when the script has finished loading by adding the following code at the bottom of the script:

```
if (typeof(Sys) !== 'undefined') Sys.Application.notifyScriptLoaded();
```

The code shown in Listing 9 tells the ScriptManager to look for a debug version of the Person script so it will automatically look for Person.debug.js instead of Person.js. If the Person.debug.js file is not found an error will be raised.

In cases where you want a debug or release version of a custom script to be loaded based upon the value of the ScriptMode property set on the ScriptManager control, you can set the ScriptReference control's ScriptMode property to Inherit. This will cause the proper version of the custom script to be loaded based upon the ScriptManager's ScriptMode property as shown in Listing 10. Because the ScriptMode property of the ScriptManager control is set to Debug, the Person.debug.js script will be loaded and used in the page.

Listing 10. Inheriting the ScriptMode from the ScriptManager for custom scripts.

```
<asp:ScriptManager ID="ScriptManager1" runat="server" ScriptMode="Debug">
 <Scripts>
 <asp:ScriptReference Path="~/Scripts/Person.js" ScriptMode="Inherit"/>
 </Scripts>
</asp:ScriptManager>
```

By using the ScriptMode property appropriately you can more easily debug applications and simplify the overall process. The ASP.NET AJAX library's release scripts are rather difficult to step through and read since code formatting has been removed while the debug scripts are formatted specifically for debugging purposes.

Conclusion

Microsoft's ASP.NET AJAX technology provides a solid foundation for building AJAX-enabled applications that can enhance the end user's overall experience. However, as with any programming technology, bugs and other application issues will certainly arise. Knowing about the different debugging options available can save a lot of time and result in a more stable product.

In this article you've been introduced to several different techniques for debugging ASP.NET AJAX pages including Internet Explorer with Visual Studio 2008, Web Development Helper and Firebug. These tools can simplify the overall debugging process since you can access variable data, walk through code line by line and view trace statements. In addition to the different debugging tools discussed, you also saw how the ASP.NET AJAX library's Sys.Debug class can be used in an application and how the ScriptManager class can be used to load debug or release versions of scripts.

Bio

Dan Wahlin (Microsoft Most Valuable Professional for ASP.NET and XML Web Services) is a .NET development instructor and architecture consultant at Interface Technical Training (www.interfacett.com). Dan founded the XML for ASP.NET Developers Web site (www.XMLforASP.NET), is on the INETA Speaker's Bureau and speaks at several conferences. Dan co-authored Professional Windows DNA (Wrox), ASP.NET: Tips, Tutorials and Code (Sams), ASP.NET 1.1 Insider Solutions, Professional ASP.NET 2.0 AJAX (Wrox), ASP.NET 2.0 MVP Hacks and authored XML for ASP.NET Developers (Sams). When he's not writing code, articles or books, Dan enjoys writing and recording music and playing golf and basketball with his wife and kids.

Scott Cate has been working with Microsoft Web technologies since 1997 and is the President of myKB.com (www.myKB.com) where he specializes in writing ASP.NET based applications focused on Knowledge Base Software solutions. Scott can be contacted via email at scott.cate@myKB.com or his blog at ScottCate.com

PREVIOUS

ASP.NET 3.5 - Web Forms Master Pages

2/20/2020 • 2 minutes to read • Edit Online

How to create and use ASP.NET master pages and content pages to create a consistent layout in Web Forms applications.

- Creating a Site-Wide Layout Using Master Pages (C#)
- Multiple ContentPlaceHolders and Default Content (C#)
- Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page (C#)
- URLs in Master Pages (C#)
- Control ID Naming in Content Pages (C#)
- Interacting with the Master Page from the Content Page (C#)
- Interacting with the Content Page from the Master Page (C#)
- Master Pages and ASP.NET AJAX (C#)
- Specifying the Master Page Programmatically (C#)
- Nested Master Pages (C#)
- Creating a Site-Wide Layout Using Master Pages (VB)
- Multiple ContentPlaceHolders and Default Content (VB)
- Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page (VB)
- URLs in Master Pages (VB)
- Control ID Naming in Content Pages (VB)
- Interacting with the Master Page from the Content Page (VB)
- Interacting with the Content Page from the Master Page (VB)
- Master Pages and ASP.NET AJAX (VB)
- Specifying the Master Page Programmatically (VB)
- Nested Master Pages (VB)

Creating a Site-Wide Layout Using Master Pages (C#)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

This tutorial will show master page basics. Namely, what are master pages, how does one create a master page, what are content place holders, how does one create an ASP.NET page that uses a master page, how modifying the master page is automatically reflected in its associated content pages, and so on.

Introduction

One attribute of a well-designed website is a consistent site-wide page layout. Take the www.asp.net website, for example. At the time of this writing, every page has the same content at the top and bottom of the page. As Figure 1 shows, the very top of each page displays a gray bar with a list of Microsoft Communities. Beneath that is the site logo, the list of languages into which the site has been translated, and the core sections: Home, Get Started, Learn, Downloads, and so forth. Likewise, the bottom of the page includes information about advertising on www.asp.net, a copyright statement, and a link to the privacy statement.

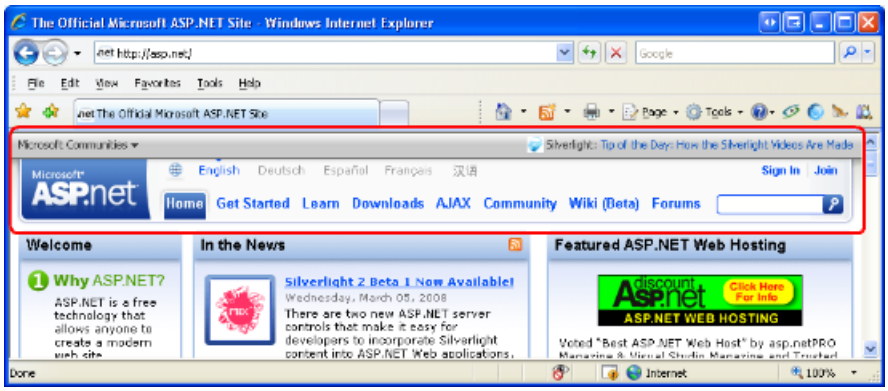


Figure 01: The www.asp.net Website Employs a Consistent Look and Feel Across All Pages (Click to view full-size image)

Another attribute of a well-designed site is the ease with which the site's appearance can be changed. Figure 1 shows the www.asp.net homepage as of March 2008, but between now and this tutorial's publication, the look and feel may have changed. Perhaps the menu items along the top will expand to include a new section for the MVC framework. Or maybe a radically new design with different colors, fonts, and layout will be unveiled. Applying such changes to the entire site should be a fast and simple process that does not require modifying the thousands of web pages that make up the site.

Creating a site-wide page template in ASP.NET is possible through the use of *master pages*. In a nutshell, a master page is a special type of ASP.NET page that defines the markup that is common among all *content pages* as well as regions that are customizable on a content page-by-content page basis. (A content page is an ASP.NET page that is bound to the master page.) Whenever a master page's layout or formatting is changed, all of its content pages' output is likewise immediately updated, which makes applying site-wide appearance changes as easy as updating and deploying a single file (namely, the master page).

This is the first tutorial in a series of tutorials that explore using master pages. Over the course of this tutorial series we:

- Examine creating master pages and their associated content pages,
- Discuss a variety of tips, tricks, and traps,
- Identify common master page pitfalls and explore workarounds,
- See how to access the master page from a content page and vice-a-versa,
- Learn how to specify a content page's master page at runtime, and
- Other advanced master page topics.

These tutorials are geared to be concise and provide step-by-step instructions with plenty of screen shots to walk you through the process visually. Each tutorial is available in C# and Visual Basic versions and includes a download of the complete code used.

This inaugural tutorial starts with a look at master page basics. We discuss how master pages work, look at creating a master page and associated content pages using Visual Web Developer, and see how changes to a master page are immediately reflected in its content pages. Let's get started!

Understanding How Master Pages Work

Building a website with a consistent site-wide page layout requires that each web page emit common formatting markup in addition to its custom content. For example, while each tutorial or forum post on www.asp.net have their own unique content, each of these pages also render a series of common `<div>` elements that display the top-level section links: Home, Get Started, Learn, and so on.

There are a variety of techniques for creating web pages with a consistent look and feel. A naive approach is to simply copy and paste the common layout markup into all web pages, but this approach has a number of downsides. For starters, every time a new page is created, you must remember to copy and paste the shared content into the page. Such copying and pasting operations are ripe for error as you may accidentally copy only a subset of the shared markup into a new page. And to top it off, this approach makes replacing the existing site-wide appearance with a new one a real pain because every single page in the site must be edited in order to use the new look and feel.

Prior to ASP.NET version 2.0, page developers often placed common markup in User Controls and then added these User Controls to each and every page. This approach required that the page developer remember to manually add the User Controls to every new page, but allowed for easier site-wide modifications because when updating the common markup only the User Controls needed to be modified. Unfortunately, Visual Studio .NET 2002 and 2003 - the versions of Visual Studio used to create ASP.NET 1.x applications - rendered User Controls in the Design view as gray boxes. Consequently, page developers using this approach did not enjoy a WYSIWYG design-time environment.

The shortcomings of using User Controls were addressed in ASP.NET version 2.0 and Visual Studio 2005 with the introduction of *master pages*. A master page is a special type of ASP.NET page that defines both the site-wide markup and the *regions* where associated *content pages* define their custom markup. As we will see in Step 1, these regions are defined by ContentPlaceHolder controls. The ContentPlaceHolder control simply denotes a position in the master page's control hierarchy where custom content can be injected by a content page.

> **NOTE**
> The core concepts and functionality of master pages has not changed since ASP.NET version 2.0. However, Visual Studio 2008 offers design-time support for nested master pages, a feature that was lacking in Visual Studio 2005. We will look at using nested master pages in a future tutorial.

Figure 2 shows what the master page for www.asp.net might look like. Note that the master page defines the common site-wide layout - the markup at the top, bottom, and right of every page - as well as a ContentPlaceHolder in the middle-left, where the unique content for each individual web page is located.

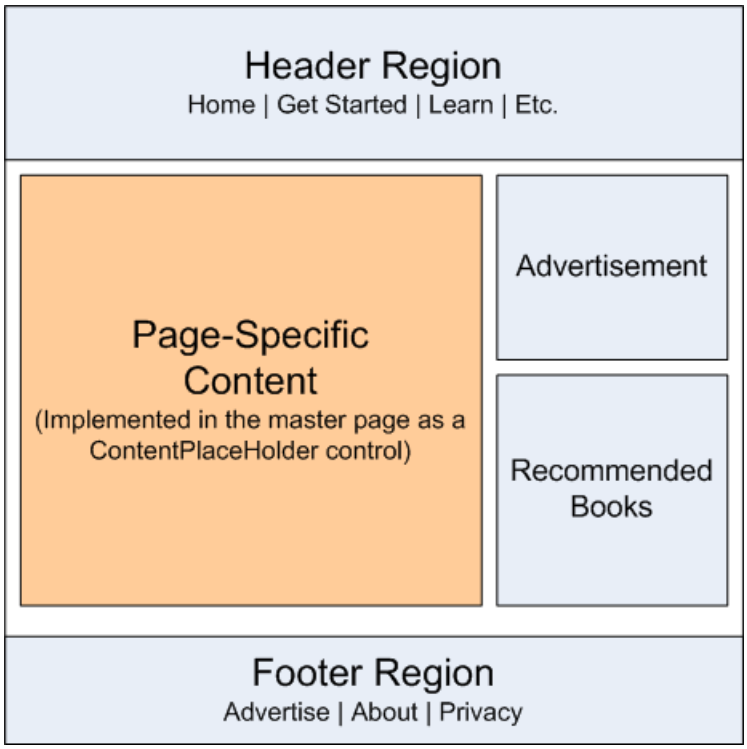


Figure 02: A Master Page Defines the Site-Wide Layout and the Regions Editable on a Content Page-by-Content Page Basis

Once a master page has been defined it can be bound to new ASP.NET pages through the tick of a checkbox. These ASP.NET pages - called content pages - include a Content control for each of the master page's ContentPlaceHolder controls. When the content page is visited through a browser the ASP.NET engine creates the master page's control hierarchy and injects the content page's control hierarchy into the appropriate places. This combined control hierarchy is rendered and the resulting HTML is returned to the end user's browser. Consequently, the content page emits both the common markup defined in its master page outside of the ContentPlaceHolder controls and the page-specific markup defined within its own Content controls. Figure 3 illustrates this concept.

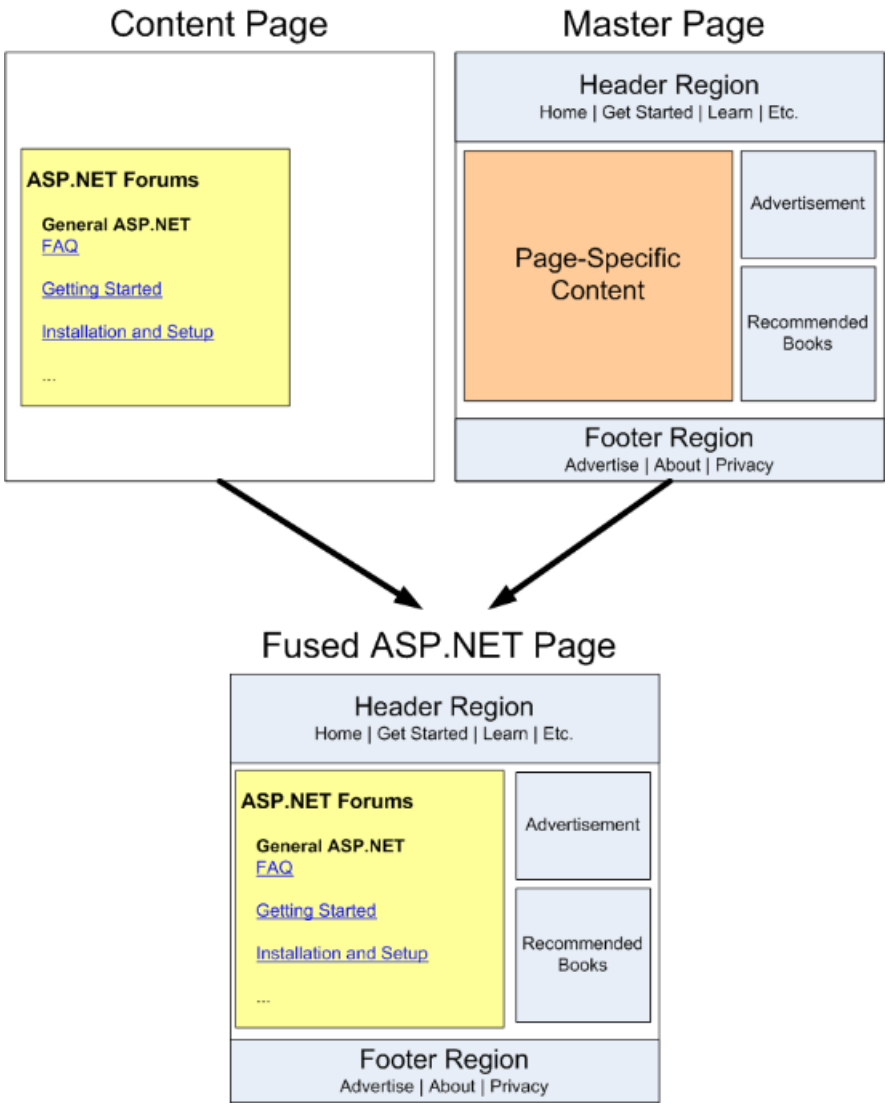


Figure 03: The Requested Page's Markup is Fused into the Master Page (Click to view full-size image)

Now that we have discussed how master pages work, let's take a look at creating a master page and associated content pages using Visual Web Developer.

> **NOTE**
>
> In order to reach the widest possible audience, the ASP.NET website we build throughout this tutorial series will be created using ASP.NET 3.5 with Microsoft's free version of Visual Studio 2008, Visual Web Developer 2008. If you have not yet upgraded to ASP.NET 3.5, don't worry - the concepts discussed in these tutorials work equally well with ASP.NET 2.0 and Visual Studio 2005. However, some demo applications may use features new to the .NET Framework version 3.5; when 3.5-specific features are used, I include a note that discusses how to implement similar functionality in version 2.0. Do keep in mind that the demo applications available for download from each tutorial target the .NET Framework version 3.5, which results in a `Web.config` file that includes 3.5-specific configuration elements and references to 3.5-specific namespaces in the `using` statements in ASP.NET pages' code-behind classes. Long story short, if you have yet to install .NET 3.5 on your computer then the downloadable web application will not work without first removing the 3.5-specific markup from `Web.config`. See Dissecting ASP.NET Version 3.5's `Web.config` File for more information on this topic. You will also need to remove the `using` statements that reference 3.5-specific namespaces.

Step 1: Creating a Master Page

Before we can explore creating and using master and content pages, we first need an ASP.NET website. Start by creating a new file system-based ASP.NET website. To accomplish this, launch Visual Web Developer and then go to the File menu and choose New Web Site, displaying the New Web Site dialog box (see Figure 4). Choose the

ASP.NET Web Site template, set the Location drop-down list to File System, choose a folder to place the web site, and set the language to C#. This will create a new web site with a `Default.aspx` ASP.NET page, an `App_Data` folder, and a `Web.config` file.

> **NOTE**
>
> Visual Studio supports two modes of project management: Web Site Projects and Web Application Projects. Web Site Projects lack a project file, whereas Web Application Projects mimic the project architecture in Visual Studio .NET 2002/2003 - they include a project file and compile the project's source code into a single assembly, which is placed in the `/bin` folder. Visual Studio 2005 initially only supported Web Site Projects, although the Web Application Project model was reintroduced with Service Pack 1; Visual Studio 2008 offers both project models. The Visual Web Developer 2005 and 2008 editions, however, only support Web Site Projects. I use the Web Site Project model for my demos in this tutorial series. If you are using a non-Express edition and want to use the Web Application Project model instead, feel free to do so but be aware that there may be some discrepancies between what you see on your screen and the steps you must take versus the screen shots shown and instructions provided in these tutorials.

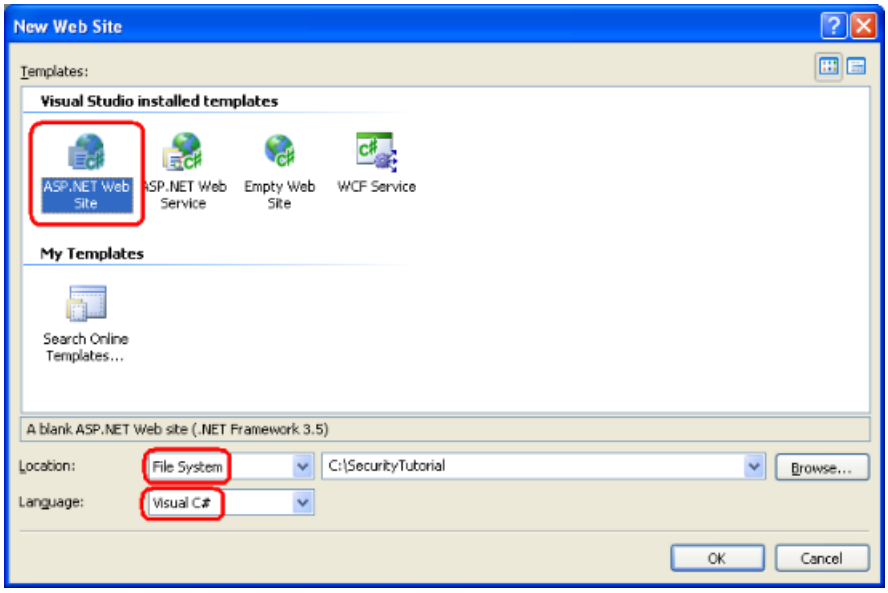


Figure 04: Create a New File System-Based Web Site (Click to view full-size image)

Next, add a master page to the site in the root directory by right-clicking on the Project name, choosing Add New Item, and selecting the Master Page template. Note that master pages end with the extension `.master`. Name this new master page `Site.master` and click Add.

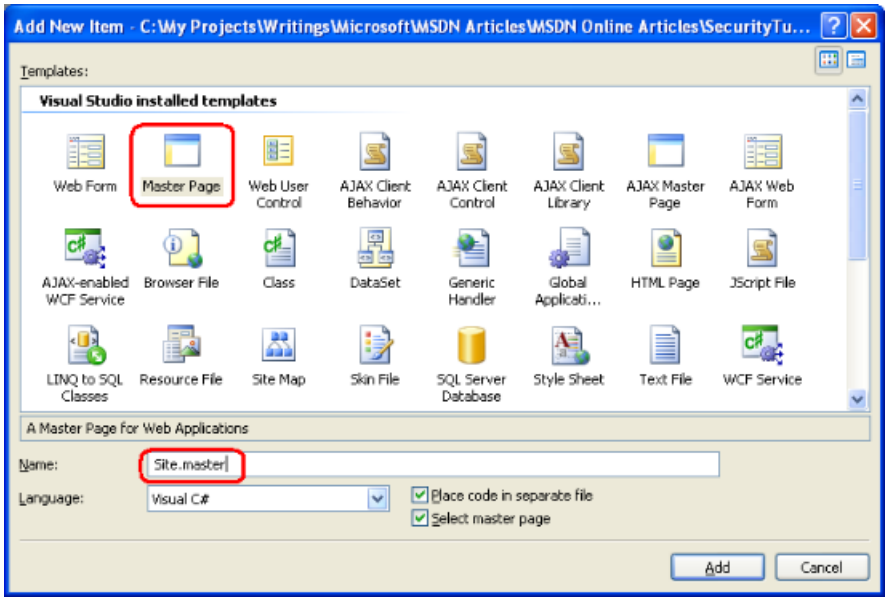


Figure 05: Add a Master Page Named `Site.master` to the Website (Click to view full-size image)

Adding a new master page file through Visual Web Developer creates a master page with the following declarative markup:

```
<%@ Master Language="C#" AutoEventWireup="true" CodeFile="Site.master.cs" Inherits="Site" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
</head>
<body>
 <form id="form1" runat="server">
 <div>
 <asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server">

 </asp:ContentPlaceHolder>
 </div>
 </form>
</body>
</html>
```

The first line in the declarative markup is the `@Master` directive. The `@Master` directive is similar to the `@Page` directive that appears in ASP.NET pages. It defines the server-side language (C#) and information about the location and inheritance of the master page's code-behind class.

The `DOCTYPE` and the page's declarative markup appears beneath the `@Master` directive. The page includes static HTML along with four server-side controls:

- **A Web Form (the `<form runat="server">`)** - because all ASP.NET pages typically have a Web Form - and because the master page may include Web controls that must appear within a Web Form - be sure to add the Web Form to your master page (rather than adding a Web Form to each content page).
- **A ContentPlaceHolder control named `ContentPlaceHolder1`** - this ContentPlaceHolder control appears within the Web Form and serves as the region for the content page's user interface.
- **A server-side `<head>` element** - the `<head>` element has the `runat="server"` attribute, making it accessible through server-side code. The `<head>` element is implemented this way so that the page's title and other `<head>`-related markup may be added or adjusted programmatically. For example, setting an ASP.NET page's `Title` property changes the `<title>` element rendered by the `<head>` server control.
- **A ContentPlaceHolder control named `head`** - this ContentPlaceHolder control appears within the `<head>` server control and can be used to declaratively add content to the `<head>` element.

This default master page declarative markup serves as a starting point for designing your own master pages. Feel free to edit the HTML or to add additional Web controls or ContentPlaceHolders to the master page.

> **NOTE**
> When designing a master page make sure that the master page contains a Web Form and that at least one ContentPlaceHolder control appears within this Web Form.

Creating a Simple Site Layout

Let's expand `Site.master`'s default declarative markup to create a site layout where all pages share: a common header; a left column with navigation, news and other site-wide content; and a footer that displays the "Powered by Microsoft ASP.NET" icon. Figure 6 shows the end result of the master page when one of its content pages is

viewed through a browser. The red circled region in Figure 6 is specific to the page being visited (`Default.aspx`); the other content is defined in the master page and therefore consistent across all content pages.

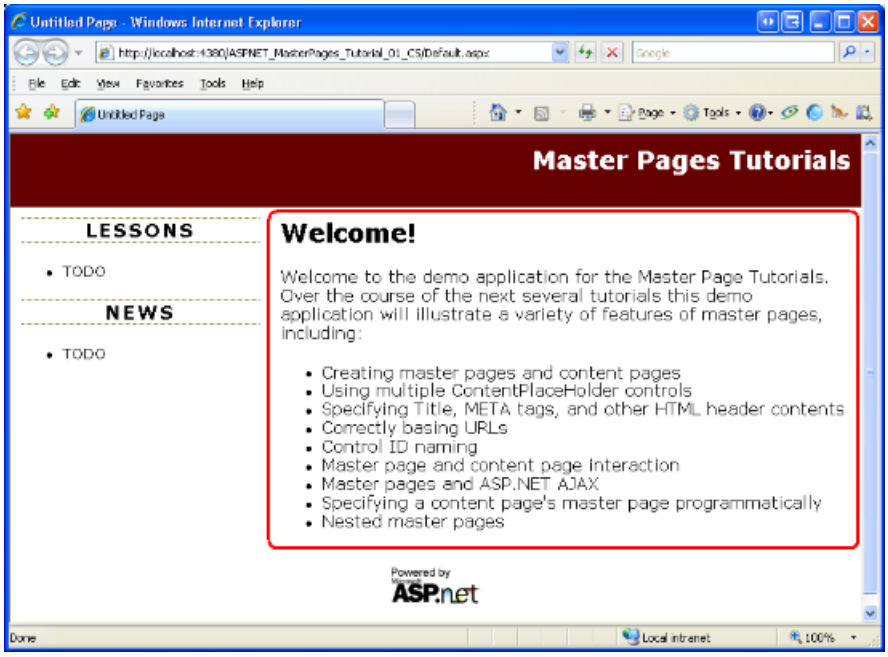


Figure 06: The Master Page Defines the Markup for the Top, Left, and Bottom Portions (Click to view full-size image)

To achieve the site layout shown in Figure 6, start by updating the `Site.master` master page so that it contains the following declarative markup:

```
<%@ Master Language="C#" AutoEventWireup="true" CodeFile="Site.master.cs" Inherits="Site" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
 <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
 <form id="form1" runat="server">
 <div id="topContent">
 <a href="Default.aspx">Master Pages Tutorials</a>
 </div>

 <div id="mainContent">
 <asp:ContentPlaceHolder id="MainContent" runat="server">
 </asp:ContentPlaceHolder>
 </div>

 <div id="leftContent">
 <h3>Lessons</h3>
 <ul>
 <li>TODO</li>
 </ul>
 <h3>News</h3>
 <ul>
 <li>TODO</li>
 </ul>
 </div>

 <div id="footerContent">
 <img src="Images/PoweredByASPNET.gif" alt="Powered by ASP.NET!" />
 </div>
 </form>
</body>
</html>
```

The master page's layout is defined using a series of `<div>` HTML elements. The `topContent` `<div>` contains the markup that appears at the top of each page, while the `mainContent`, `leftContent`, and `footerContent` `<div>` s are used to display the page's content, the left column, and the "Powered by Microsoft ASP.NET" icon, respectively. In addition to adding these `<div>` elements, I also renamed the `ID` property of the primary ContentPlaceHolder control from `ContentPlaceHolder1` to `MainContent`.

The formatting and layout rules for these assorted `<div>` elements is spelled out in the Cascading Stylesheet (CSS) file `Styles.css`, which is specified via a <link> element in the master page's <head> element. These various rules define the look and feel of each `<div>` element noted above. For example, the `topContent` `<div>` element, which displays the "Master Pages Tutorials" text and link, has its formatting rules specified in `Styles.css` as follows:

```
#topContent {
 text-align: right;
 background-color: #600;
 color: White;
 font-size: x-large;
 text-decoration: none;
 font-weight: bold;
 padding: 10px;
 height: 50px;
}
```

If you are following along at your computer, you will need to download this tutorial's accompanying code and add the `Styles.css` file to your project. Similarly, you will also need to create a folder named Images and copy the "Powered by Microsoft ASP.NET" icon from the downloaded demo website to your project.

> **NOTE**
> A discussion of CSS and web page formatting is beyond the scope of this article. For more on CSS, check out the CSS Tutorials at W3Schools.com. I also encourage you to download this tutorial's accompanying code and play with the CSS settings in `Styles.css` to see the effects of different formatting rules.

Creating a Master Page Using an Existing Design Template

Over the years I've built a number of ASP.NET web applications for small- to medium-sized companies. Some of my clients had an existing site layout they wanted to use; others hired a competent graphics designer. A few entrusted me to design the website layout. As you can tell by Figure 6, tasking a programmer to design a website's layout is usually as wise as having your accountant perform open-heart surgery while your doctor does your taxes.

Fortunately, there are innumerous websites that offer free HTML design templates - Google returned more than six million results for the search term "free website templates." One of my favorite ones is OpenDesigns.org. Once you find a website template you like, add the CSS files and images to your website project and integrate the template's HTML into your master page.

> **NOTE**
> Microsoft also offers a number of free ASP.NET Design Start Kit Templates that integrate into the New Web Site dialog box in Visual Studio.

Step 2: Creating Associated Content Pages

With the master page created, we are ready to start creating ASP.NET pages that are bound to the master page. Such pages are referred to as *content pages*.

Let's add a new ASP.NET page to the project and bind it to the `Site.master` master page. Right-click on the project name in Solution Explorer and choose the Add New Item option. Select the Web Form template, enter the name `About.aspx`, and then check the "Select master page" checkbox as shown in Figure 7. Doing so will display the Select a Master Page dialog box (see Figure 8) from where you can choose the master page to use.

> **NOTE**
> If you created your ASP.NET website using the Web Application Project model instead of the Web Site Project model you will not see the "Select master page" checkbox in the Add New Item dialog box shown in Figure 7. To create a content page when using the Web Application Project model you must choose the Web Content Form template instead of the Web Form template. After selecting the Web Content Form template and clicking Add, the same Select a Master Page dialog box shown in Figure 8 will appear.

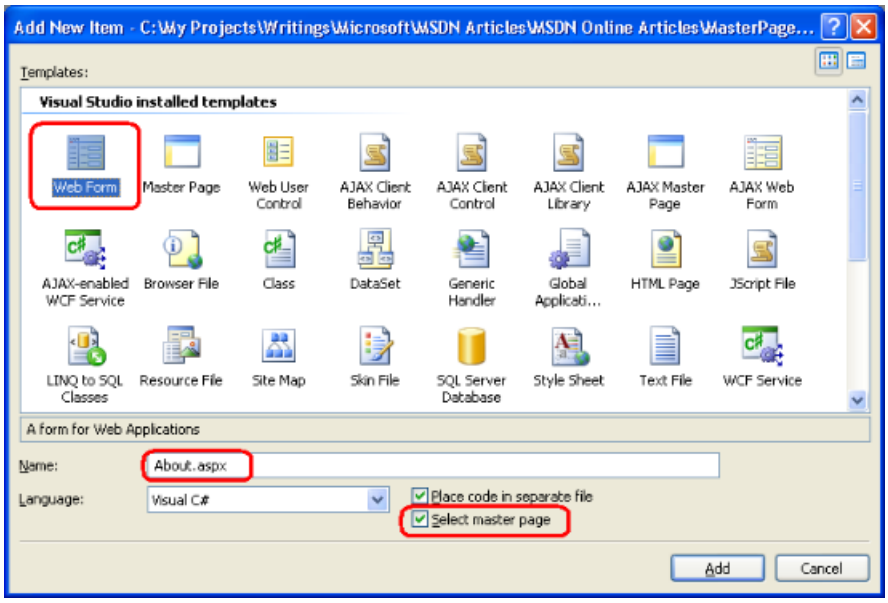


Figure 07: Add a New Content Page (Click to view full-size image)

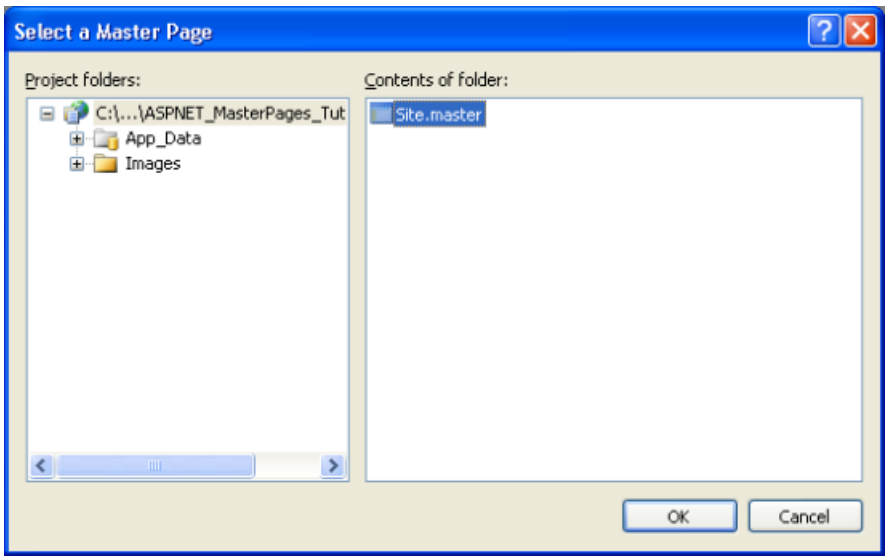


Figure 08: Select the `Site.master` Master Page (Click to view full-size image)

As the following declarative markup shows, a new content page contains a `@Page` directive that points back to its master page and a Content control for each of the master page's ContentPlaceHolder controls.

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true" CodeFile="About.aspx.cs"
Inherits="About" Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
</asp:Content>
```

NOTE

In the "Creating a Simple Site Layout" section in Step 1 I renamed `ContentPlaceHolder1` to `MainContent`. If you did not rename this ContentPlaceHolder control's `ID` in the same way, your content page's declarative markup will differ slightly from the markup shown above. Namely, the second Content control's `ContentPlaceHolderID` will reflect the `ID` of the corresponding ContentPlaceHolder control in your master page.

When rendering a content page, the ASP.NET engine must fuse the page's Content controls with its master page's ContentPlaceHolder controls. The ASP.NET engine determines the content page's master page from the `@Page`

directive's `MasterPageFile` attribute. As the above markup shows, this content page is bound to `~/Site.master`.

Because the master page has two ContentPlaceHolder controls - `head` and `MainContent` - Visual Web Developer generated two Content controls. Each Content control references a particular ContentPlaceHolder via its `ContentPlaceHolderID` property.

Where master pages shine over previous site-wide template techniques is with their design-time support. Figure 9 shows the `About.aspx` content page when viewed through Visual Web Developer's Design view. Note that while the master page content is visible, it is grayed out and cannot be modified. The Content controls corresponding to the master page's ContentPlaceHolders are, however, editable. And just like with any other ASP.NET page, you can create the content page's interface by adding Web controls through the Source or Design views.

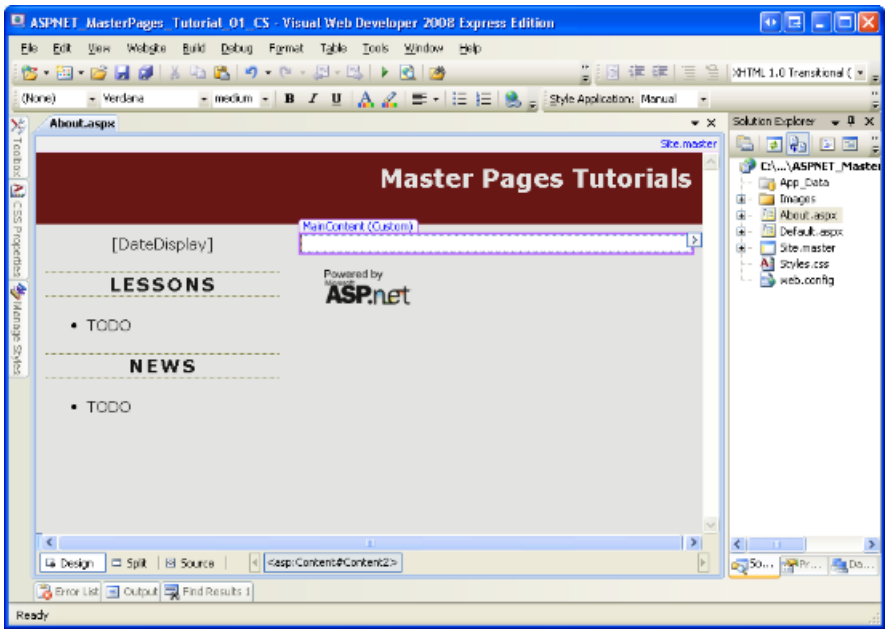


Figure 09: The Content Page's Design View Displays Both the Page-Specific and Master Page Contents (Click to view full-size image)

Adding Markup and Web Controls to the Content Page

Take a moment to create some content for the `About.aspx` page. As you can see in Figure 10, I entered an "About the Author" heading and a couple of paragraphs of text, but feel free to add Web controls, too. After creating this interface, visit the `About.aspx` page through a browser.

Figure 10: Visit the `About.aspx` Page Through a Browser (Click to view full-size image)

It is important to understand that the requested content page and its associated master page are fused and rendered as a whole entirely on the web server. The end user's browser is then sent the resulting, fused HTML. To verify this, view the HTML received by the browser by going to the View menu and choosing Source. Note that there are no frames or any other specialized techniques for displaying two different web pages in a single window.

Binding a Master Page to an Existing ASP.NET Page

As we saw in this step, adding a new content page to an ASP.NET web application is as easy as checking the "Select master page" checkbox and picking the master page. Unfortunately, converting an existing ASP.NET page to a master page is not as easy.

To bind a master page to an existing ASP.NET page you need to perform the following steps:

1. Add the `MasterPageFile` attribute to the ASP.NET page's `@Page` directive, pointing it to the appropriate master page.
2. Add Content controls for each of the ContentPlaceHolders in the master page.
3. Selectively cut and paste the ASP.NET page's existing content into the appropriate Content controls. I say "selectively" here because the ASP.NET page likely contains markup that's already expressed by the master page, such as the `DOCTYPE`, the `<html>` element, and the Web Form.

For step-by-step instructions on this process along with screen shots, check out Scott Guthrie's Using Master Pages and Site Navigation tutorial. The "Update `Default.aspx` and `DataSample.aspx` to use the Master Page" section details these steps.

Because it is much easier to create new content pages than it is to convert existing ASP.NET pages into content pages, I recommend that whenever you create a new ASP.NET website add a master page to the site. Bind all new ASP.NET pages to this master page. Don't worry if the initial master page is very simple or plain; you can update the master page later.

NOTE

When creating a new ASP.NET application, Visual Web Developer adds a `Default.aspx` page that isn't bound to a master page. If you want to practice converting an existing ASP.NET page into a content page, go ahead and do so with `Default.aspx`. Alternatively, you can delete `Default.aspx` and then re-add it, but this time checking the "Select master page" checkbox.

Step 3: Updating the Master Page's Markup

One of the primary benefits of master pages is that a single master page may be used to define the overall layout for numerous pages on the site. Therefore, updating the site's look and feel requires updating a single file - the master page.

To illustrate this behavior, let's update our master page to include the current date in at the top of the left column. Add a Label named `DateDisplay` to the `leftContent` `<div>`.

```
<div id="leftContent">
 <p>
 <asp:Label ID="DateDisplay" runat="server"></asp:Label>
 </p>

 <h3>Lessons</h3>
 <ul>
 <li>TODO</li>
 </ul>
 <h3>News</h3>
 <ul>
 <li>TODO</li>
 </ul>
</div>
```

Next, create a `Page_Load` event handler for the master page and add the following code:

```
protected void Page_Load(object sender, EventArgs e)
{
    DateDisplay.Text = DateTime.Now.ToString("dddd, MMMM dd");
}
```

The above code sets the Label's `Text` property to the current date and time formatted as the day of the week, the name of the month, and the two-digit day (see Figure 11). With this change, revisit one of your content pages. As Figure 11 shows, the resulting markup is immediately updated to include the change to the master page.

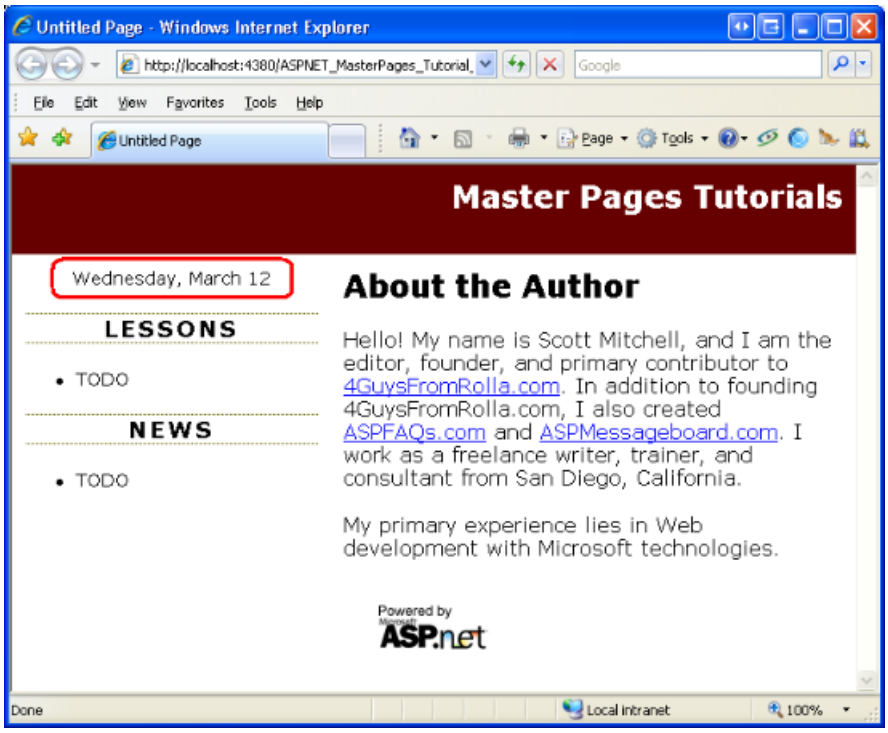


Figure 11: The Changes to the Master Page are Reflected When Viewing the a Content Page (Click to view full-size image)

> **NOTE**
>
> As this example illustrates, master pages may contain server-side Web controls, code, and event handlers.

Summary

Master pages enable ASP.NET developers to design a consistent site-wide layout that is easily updateable. Creating master pages and their associated content pages is as simple as creating standard ASP.NET pages, as Visual Web Developer offers rich design-time support.

The master page example we created in this tutorial had two ContentPlaceHolder controls, `head` and `MainContent`. We only specified markup for the `MainContent` ContentPlaceHolder control in our content page, however. In the next tutorial we look at using multiple Content controls in the content page. We also see how to define default markup for Content controls within the master page, as well as how to toggle between using the default markup defined in the master page and providing custom markup from the content page.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET for Designers: Free Design Templates and Guidance on Building ASP.NET Websites Using Web Standards
- ASP.NET Master Pages Overview
- Cascading Stylesheets (CSS) Tutorials
- Dynamically Setting the Page's Title
- Master Pages in ASP.NET
- Master Pages QuickStart Tutorials

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working

with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Multiple ContentPlaceHolders and Default Content (C#)

2/20/2020 • 13 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> Examines how to add multiple content place holders to a master page as well as how to specify default content in the content place holders.

Introduction

In the preceding tutorial we examined how master pages enable ASP.NET developers to create a consistent site-wide layout. Master pages define both markup that is common to all of its content pages and regions that are customizable on a page-by-page basis. In the previous tutorial we created a simple master page (`Site.master`) and two content pages (`Default.aspx` and `About.aspx`). Our master page consisted of two ContentPlaceHolders named `head` and `MainContent`, which were located in the `<head>` element and Web Form, respectively. While the content pages each had two Content controls, we only specified markup for the one corresponding to `MainContent`.

As evidenced by the two ContentPlaceHolder controls in `Site.master`, a master page may contain multiple ContentPlaceHolders. What's more, the master page may specify default markup for the ContentPlaceHolder controls. A content page, then, can optionally specify its own markup or use the default markup. In this tutorial we look at using multiple content controls in the master page and see how to define default markup in the ContentPlaceHolder controls.

Step 1: Adding Additional ContentPlaceHolder Controls to the Master Page

Many website designs contain several areas on the screen that are customized on a page-by-page basis. `Site.master`, the master page we created in the preceding tutorial, contains a single ContentPlaceHolder within the Web Form named `MainContent`. Specifically, this ContentPlaceHolder is located within the `mainContent` `<div>` element.

Figure 1 shows `Default.aspx` when viewed through a browser. The region circled in red is the page-specific markup corresponding to `MainContent`.

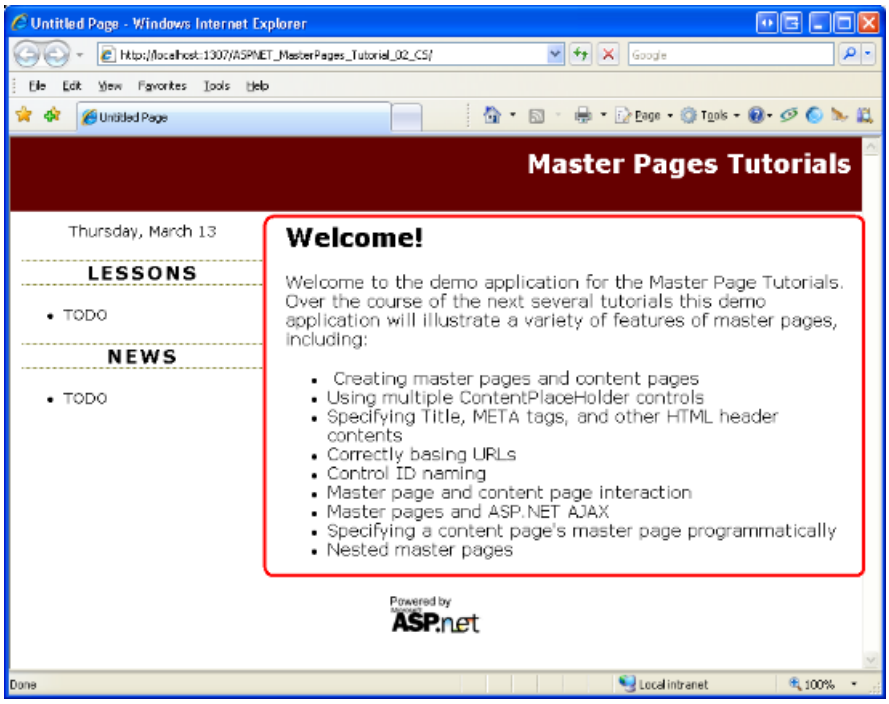


Figure 01: The Circled Region Shows the Area Currently Customizable on a Page-by-Page Basis (Click to view full-size image)

Imagine that in addition to the region shown in Figure 1, we also need to add page-specific items to the left column beneath the Lessons and News sections. To accomplish this, we add another ContentPlaceHolder control to the master page. To follow along, open the `Site.master` master page in Visual Web Developer and then drag a ContentPlaceHolder control from the Toolbox onto the designer after the News section. Set the ContentPlaceHolder's `ID` to `LeftColumnContent`.

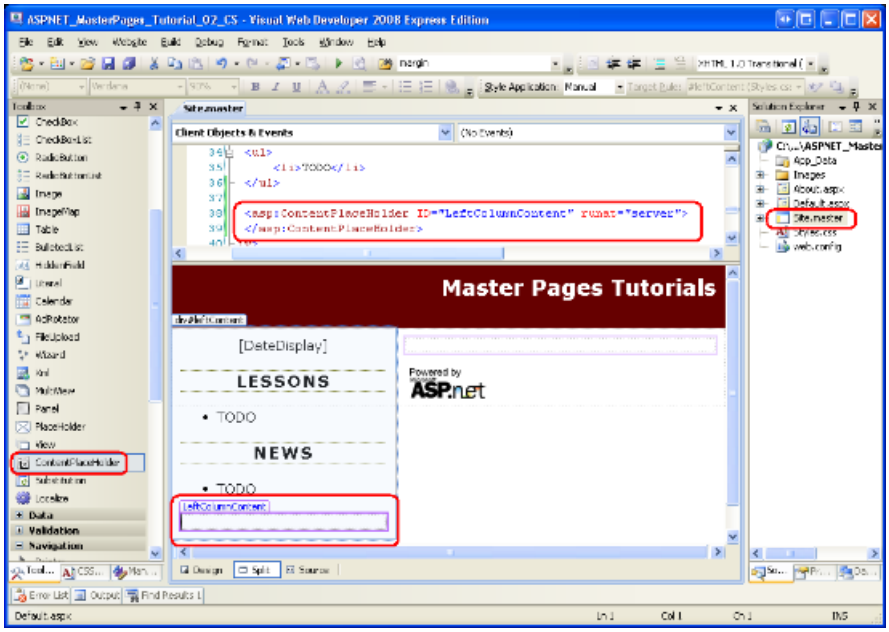


Figure 02: Add a ContentPlaceHolder Control to the Master Page's Left Column (Click to view full-size image)

With the addition of the `LeftColumnContent` ContentPlaceHolder to the master page, we can define content for this region on a page-by-page basis by including a Content control in the page whose `ContentPlaceHolderID` is set to `LeftColumnContent`. We examine this process in Step 2.

Step 2: Defining Content for the New ContentPlaceHolder in the Content Pages

When adding a new content page to the website, Visual Web Developer automatically creates a Content control in the page for each ContentPlaceHolder in the selected master page. Having added a the `LeftColumnContent` ContentPlaceHolder to our master page in Step 1, new ASP.NET pages will now have three Content controls.

To illustrate this, add a new content page to the root directory named `MultipleContentPlaceHolders.aspx` that is bound to the `Site.master` master page. Visual Web Developer creates this page with the following declarative markup:

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true"
CodeFile="MultipleContentPlaceHolders.aspx.cs" Inherits="MultipleContentPlaceHolders" Title="Untitled Page"
%>
<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
</asp:Content>
<asp:Content ID="Content3" ContentPlaceHolderID="LeftColumnContent" Runat="Server">
</asp:Content>
```

Enter some content into the Content control referencing the `MainContent` ContentPlaceHolders (`Content2`). Next, add the following markup to the `Content3` Content control (which references the `LeftColumnContent` ContentPlaceHolder):

```
<h3>Page-Specific Content</h3>
<ul>
 <li>This content is defined in the content page.</li>
 <li>The master page has two regions in the Web Form that are editable on a
 page-by-page basis.</li>
</ul>
```

After adding this markup, visit the page through a browser. As Figure 3 shows, the markup placed in the `Content3` Content control is displayed in the left column beneath the News section (circled in red). The markup placed in `Content2` is displayed in the right portion of the page (circled in blue).

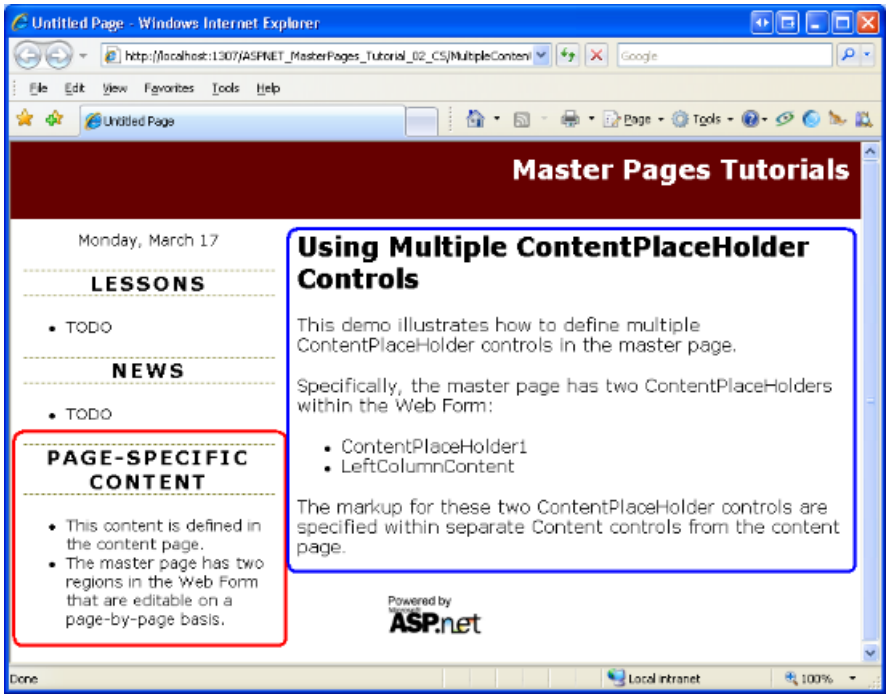


Figure 03: The Left Column Now Includes Page-Specific Content Beneath the News Section (Click to view full-size image)

Defining Content in Existing Content Pages

Creating a new content page automatically incorporates the ContentPlaceHolder control we added in Step 1. But our two existing content pages - `About.aspx` and `Default.aspx` - don't have a Content control for the `LeftColumnContent` ContentPlaceHolder. To specify content for this ContentPlaceHolder in these two existing pages, we need to add a Content control ourselves.

Unlike most ASP.NET Web controls, the Visual Web Developer Toolbox does not include a Content control item. We can manually type in the Content control's declarative markup into the Source view, but an easier and quicker approach is to use the Design view. Open the `About.aspx` page and switch to the Design view. As Figure 4 illustrates, the `LeftColumnContent` ContentPlaceHolder appears in the Design view; if you mouse over it, the title displayed reads: "LeftColumnContent (Master)." The inclusion of "Master" in the title indicates that there is no Content control defined in the page for this ContentPlaceHolder. If there exists a Content control for the ContentPlaceHolder, as in the case for `MainContent`, the title will read: "*ContentPlaceHolderID* (Custom)."

To add a Content control for the `LeftColumnContent` ContentPlaceHolder to `About.aspx`, expand the ContentPlaceHolder's smart tag and click the Create Custom Content link.

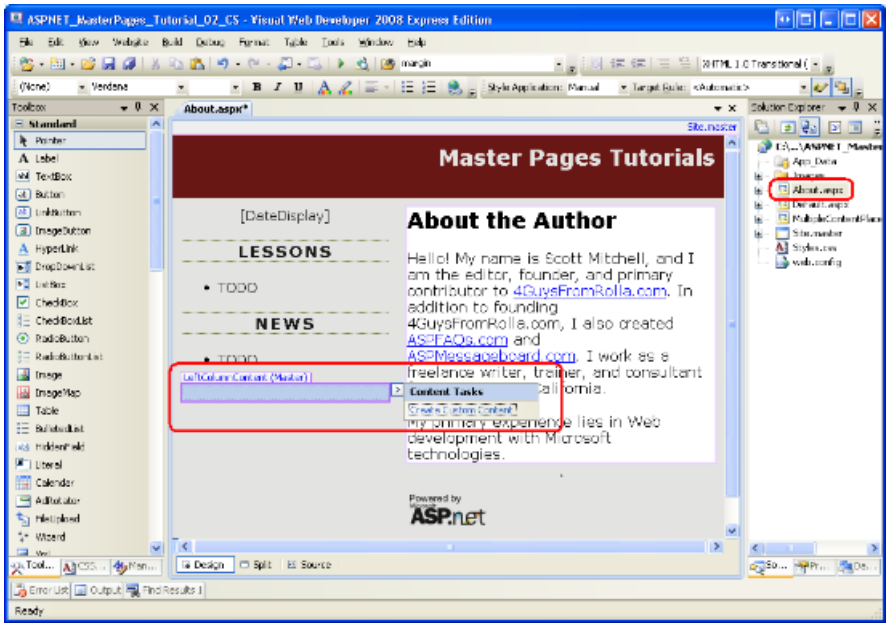


Figure 04: The Design View for `About.aspx` Shows the `LeftColumnContent` ContentPlaceHolder (Click to view full-size image)

Clicking the Create Custom Content link generates the necessary Content control in the page and sets its `ContentPlaceHolderID` property to the ContentPlaceHolder's `ID`. For example, clicking the Create Custom Content link for `LeftColumnContent` region in `About.aspx` adds the following declarative markup to the page:

```
<asp:Content ID="Content3" runat="server" contentplaceholderid="LeftColumnContent">

</asp:Content>
```

Omitting Content Controls

ASP.NET does not require that all content pages include Content controls for each and every ContentPlaceHolder defined in the master page. If a Content control is omitted, the ASP.NET engine uses the markup defined within the ContentPlaceHolder in the master page. This markup is referred to as the ContentPlaceHolder's *default content* and is useful in scenarios where the content for some region is common among the majority of pages, but needs to be customized for a small number of pages. Step 3 explores specifying default content in the master page.

Currently, `Default.aspx` contains two Content controls for the `head` and `MainContent` ContentPlaceHolders; it does not have a Content control for `LeftColumnContent`. Consequently, when `Default.aspx` is rendered the

`LeftColumnContent` ContentPlaceHolder's default content is used. Because we have yet to define any default content for this ContentPlaceHolder, the net effect is that no markup is emitted for this region. To verify this behavior, visit `Default.aspx` through a browser. As Figure 5 shows, no markup is emitted in the left column beneath the News section.

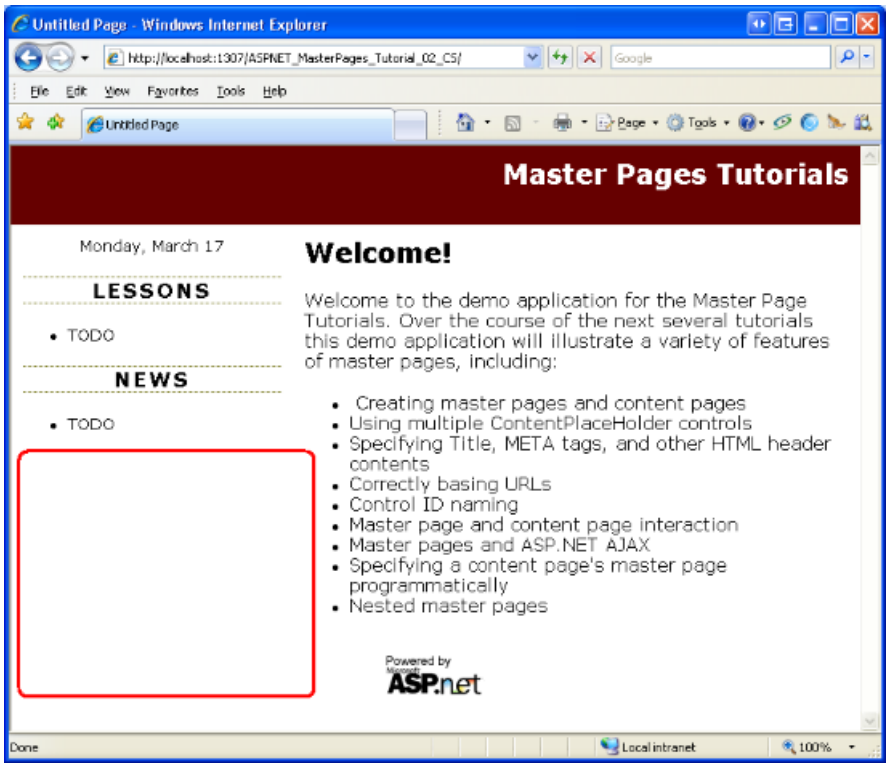


Figure 05: No Content is Rendered for the `LeftColumnContent` ContentPlaceHolder (Click to view full-size image)

Step 3: Specifying Default Content in the Master Page

Some website designs include a region whose content is the same for all pages in the site except for one or two exceptions. Consider a website that supports user accounts. Such a site requires a login page where visitors can enter their credentials to sign into the site. To expedite the sign in process, the website designers might include username and password textboxes in the upper left corner of every page to allow users to sign in without having to explicitly visit the login page. While these username and password textboxes are helpful in most pages, they are redundant in the login page, which already contains textboxes for the user's credentials.

To implement this design, you could create a ContentPlaceHolder control in the upper left corner of the master page. Each page that needed to display the username and password textboxes in their upper left corner would create a Content control for this ContentPlaceHolder and add the necessary interface. The login page, on the other hand, would either omit adding a Content control for this ContentPlaceHolder or would create a Content control with no markup defined. The downside of this approach is that we have to remember to add the username and password textboxes to every page we add to the site (except for the login page). This is asking for trouble. We're likely to forget to add these textboxes to a page or two or, worse, we might not implement the interface correctly (perhaps adding just one textbox instead of two).

A better solution is to define the username and password textboxes as the ContentPlaceHolder's default content. By doing so, we only need to override this default content in those few pages that do not display the username and password textboxes (the login page, for instance). To illustrate specifying default content for a ContentPlaceHolder control, let's implement the scenario just discussed.

> **NOTE**
> The remainder of this tutorial updates our website to include a login interface in the left column for all pages but the login page. However, this tutorial does not examine how to configure the website to support user accounts. For more information on this topic, refer to my Forms Authentication, Authorization, User Accounts and Roles tutorials.

Adding a ContentPlaceHolder and Specifying Its Default Content

Open the `Site.master` master page and add the following markup to the left column between the `DateDisplay` Label and Lessons section:

```
<asp:ContentPlaceHolder ID="QuickLoginUI" runat="server">
 <asp:Login ID="QuickLogin" runat="server"
    TitleText="<h3>Sign In</h3>"
    FailureAction="RedirectToLoginPage">
 </asp:Login>
</asp:ContentPlaceHolder>
```

After adding this markup your master page's Design view should look similar to Figure 6.

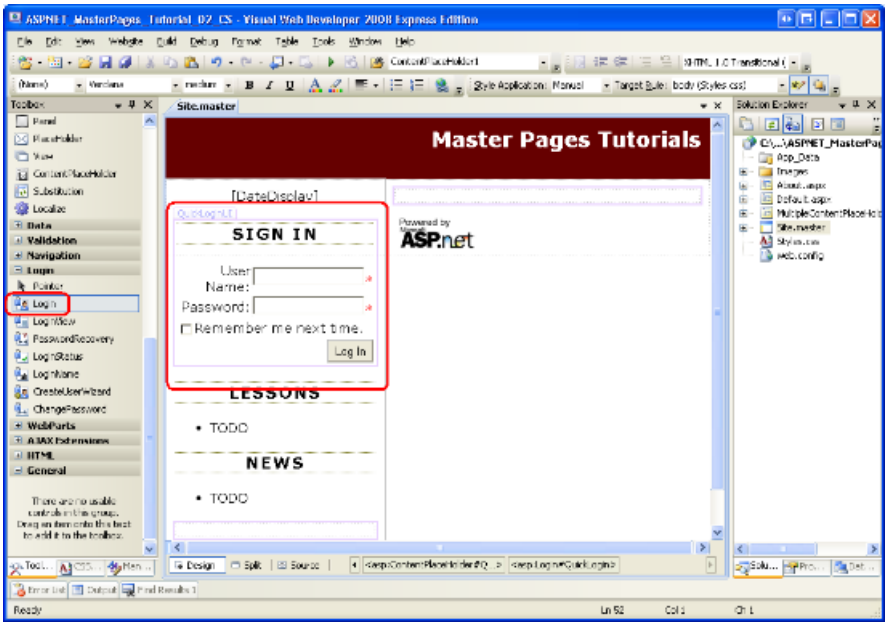


Figure 06: The Master Page Includes a Login Control (Click to view full-size image)

This ContentPlaceHolder, `QuickLoginUI`, has a Login Web control as its default content. The Login control displays a user interface that prompts the user for their username and password along with a Log In button. Upon clicking the Log In button, the Login control internally validates the user's credentials against the Membership API. To use this Login control in practice, then, you need to configure your site to use Membership. This topic is beyond the scope of this tutorial; refer to my Forms Authentication, Authorization, User Accounts and Roles tutorials for more information on building a web application that supports user accounts.

Feel free to customize the Login control's behavior or appearance. I have set two of its properties: `TitleText` and `FailureAction`. The `TitleText` property value, which defaults to "Log In", is displayed at the top of the control's user interface. I have set this property so that it displays the text "Sign In" as an `<h3>` element. The `FailureAction` property indicates what to do if the user's credentials are invalid. It defaults to a value of `Refresh`, which leaves the user on the same page and displays a failure message within the Login control. I've changed it to `RedirectToLoginPage`, which sends the user to the login page in the event of invalid credentials. I prefer to send the user to the login page when a user attempts to login from some other page, but fails, because the login page can contain additional instructions and options that would not easily fit into the left column. For example, the login page might include options to retrieve a forgotten password or to create a new account.

Creating the Login Page and Overriding the Default Content

With the master page complete, our next step is to create the login page. Add an ASP.NET page to your site's root directory named `Login.aspx`, binding it to the `Site.master` master page. Doing so will create a page with four Content controls, one for each of the ContentPlaceHolders defined in `Site.master`.

Add a Login control to the `MainContent` Content control. Likewise, feel free to add any content to the `LeftColumnContent` region. However, make sure to leave the Content control for the `QuickLoginUI` ContentPlaceHolder empty. This will ensure that the Login control does not appear in the left column of the login page.

After defining the content for the `MainContent` and `LeftColumnContent` regions, your login page's declarative markup should look similar to the following:

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true" CodeFile="Login.aspx.cs"
Inherits="Login" Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
 <h2>
 Sign In</h2>
 <p>
 <asp:Login ID="Login1" runat="server" TitleText="">
 </asp:Login>
 </p>
</asp:Content>
<asp:Content ID="Content3" ContentPlaceHolderID="QuickLoginUI" Runat="Server">
</asp:Content>
<asp:Content ID="Content4" ContentPlaceHolderID="LeftColumnContent" Runat="Server">
 <h3>Sign In Tasks</h3>
 <ul>
 <li>Create a New Account</li>
 <li>Recover Forgotten Password</li>
 </ul>
 <p>TODO: Turn the above text into links...</p>
</asp:Content>
```

Figure 7 shows this page when viewed through a browser. Because this page specifies a Content control for the `QuickLoginUI` ContentPlaceHolder, it overrides the default content specified in the master page. The net effect is that the Login control displayed in the master page's Design view (see Figure 6) is not rendered in this page.

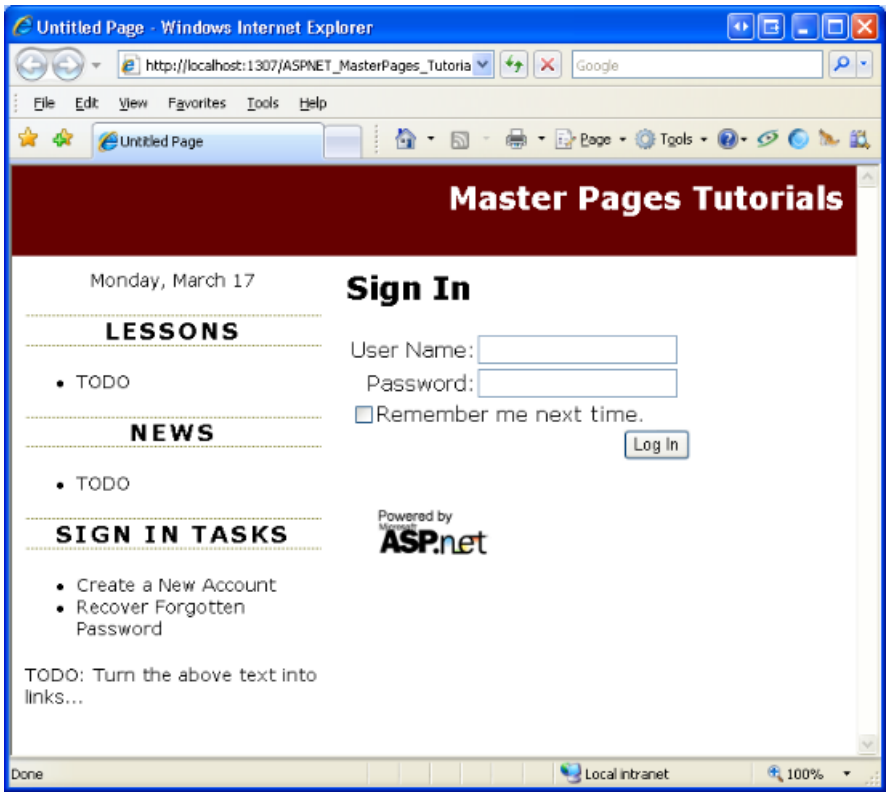


Figure 07: The Login Page Represses the `QuickLoginUI` ContentPlaceHolder's Default Content (Click to view full-size image)

Using the Default Content in New Pages

We want to show the Login control in the left column for all pages except the Login page. To achieve this, all the content pages except for the login page should omit a Content control for the `QuickLoginUI` ContentPlaceHolder. By omitting a Content control, the ContentPlaceHolder's default content will be used instead.

Our existing content pages - `Default.aspx`, `About.aspx`, and `MultipleContentPlaceHolders.aspx` - do not include a Content control for `QuickLoginUI` because they were created before we added that ContentPlaceHolder control to the master page. Therefore, these existing pages do not need to be updated. However, new pages added to the website include a Content control for the `QuickLoginUI` ContentPlaceHolder, by default. Therefore, we have to remember to remove these Content controls each time we add a new content page (unless we want to override the ContentPlaceHolder's default content, as in the case of the login page).

To remove the Content control, you can either manually delete its declarative markup from the Source view or, from the Design view, choose the Default to Master's Content link from its smart tag. Either approach removes the Content control from the page and produces the same net effect.

Figure 8 shows `Default.aspx` when viewed through a browser. Recall that `Default.aspx` only has two Content controls specified in its declarative markup - one for `head` and one for `MainContent`. As a result, the default content for the `LeftColumnContent` and `QuickLoginUI` ContentPlaceHolders are displayed.

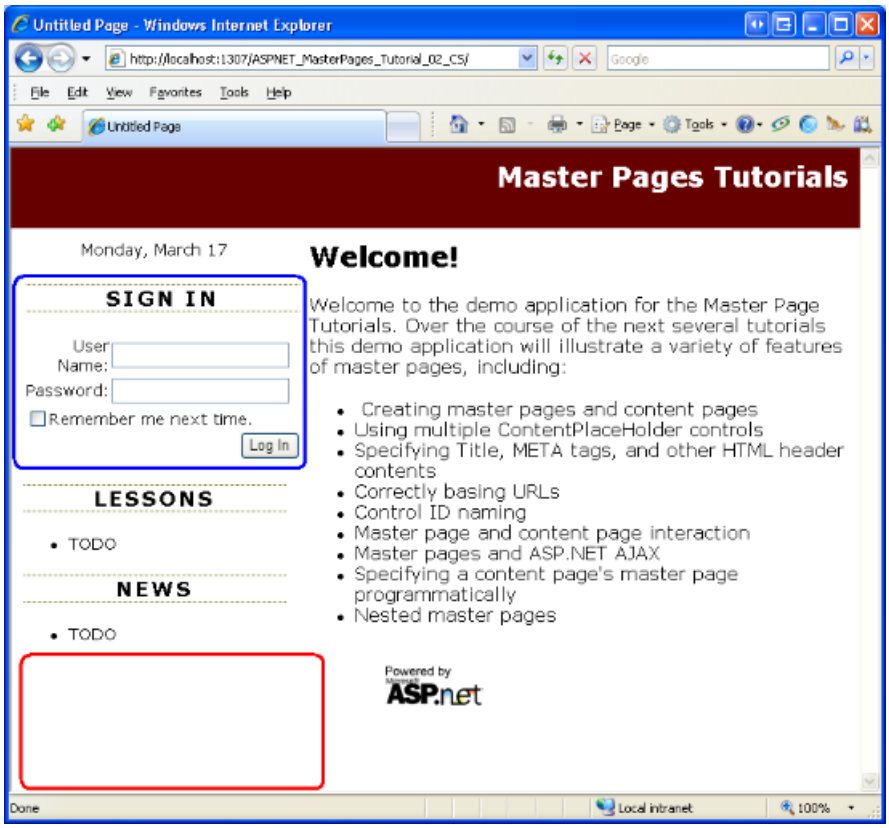


Figure 08: The Default Content for the `LeftColumnContent` and `QuickLoginUI` ContentPlaceHolders are Displayed (Click to view full-size image)

Summary

The ASP.NET master page model allows for an arbitrary number of ContentPlaceHolders in the master page. What's more, ContentPlaceHolders include default content, which is emitted in the case that there is no corresponding Content control in the content page. In this tutorial we saw how to include additional ContentPlaceHolder controls in the master page and how to define Content controls for these new ContentPlaceHolders in both new and existing ASP.NET pages. We also looked at specifying default content in a ContentPlaceHolder, which is useful in scenarios where only a minority of pages needs to customize the otherwise standardized content within a certain region.

In the next tutorial we'll examine the `head` ContentPlaceHolder in more detail, seeing how to declaratively and programmatically define the title, meta tags, and other HTML headers on a page-by-page basis.

Happy Programming!

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Suchi Banerjee. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page (C#)

2/20/2020 • 20 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> Looks at different techniques for defining assorted <head> elements in the Master Page from the content page.

Introduction

New master pages created in Visual Studio 2008 have, by default, two ContentPlaceHolder controls: one named head, and located in the `<head>` element; and one named `ContentPlaceHolder1`, placed within the Web Form. The purpose of `ContentPlaceHolder1` is to define a region in the Web Form that can be customized on a page-by-page basis. The `head` ContentPlaceHolder enables pages to add custom content to the `<head>` section. (Of course, these two ContentPlaceHolders can be modified or removed, and additional ContentPlaceHolder may be added to the master page. Our master page, `Site.master`, currently has four ContentPlaceHolder controls.)

The HTML `<head>` element serves as a repository for information about the web page document that is not part of the document itself. This includes information such as the web page's title, meta-information used by search engines or internal crawlers, and links to external resources, such as RSS feeds, JavaScript, and CSS files. Some of this information may be pertinent to all pages in the website. For example, you might want to globally import the same CSS rules and JavaScript files for every ASP.NET page. However, there are portions of the `<head>` element that are page-specific. The page title is a prime example.

In this tutorial we examine how to define global and page-specific `<head>` section markup in the master page and in its content pages.

Examining the Master Page's `<head>` Section

The default master page file created by Visual Studio 2008 contains the following markup in its `<head>` section:

```
<head runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
</head>
```

Notice that the `<head>` element contains a `runat="server"` attribute, which indicates that it is a server control (rather than static HTML). All ASP.NET pages derive from the `Page` class, which is located in the `System.Web.UI` namespace. This class contains a `Header` property that provides access to the page's `<head>` region. Using the `Header` property we can set an ASP.NET page's title or add additional markup to the rendered `<head>` section. It is possible, then, to customize a content page's `<head>` element by writing a bit of code in the page's `Page_Load` event handler. We examine how to programmatically set the page's title in Step 1.

The markup shown in the `<head>` element above also includes a ContentPlaceHolder control named head. This ContentPlaceHolder control is not necessary, as content pages can add custom content to the `<head>` element

programmatically. It's useful, however, in situations where a content page needs to add static markup to the `<head>` element as the static markup can be added declaratively to the corresponding Content control rather than programmatically.

In addition to the `<title>` element and head ContentPlaceHolder, the master page's `<head>` element should contain any `<head>`-level markup that's common to all pages. In our website, all pages use the CSS rules defined in the `Styles.css` file. Consequently, we updated the `<head>` element in the *Creating a Site-Wide Layout with Master Pages* tutorial to include a corresponding `<link>` element. Our `Site.master` master page's current `<head>` markup is shown below.

```
<head runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
 <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
```

Step 1: Setting a Content Page's Title

The web page's title is specified via the `<title>` element. It is important to set each page's title to an appropriate value. When visiting a page, its title is displayed in the browser's Title bar. Additionally, when bookmarking a page, browsers use the page's title as the suggested name for the bookmark. Also, many search engines show the page's title when displaying search results.

> **NOTE**
> By default, Visual Studio sets the `<title>` element in the master page to "Untitled Page". Similarly, new ASP.NET pages have their `<title>` set to "Untitled Page", too. Because it can be easy to forget to set the page's title to an appropriate value, there are many pages on the Internet with the title "Untitled Page". Searching Google for web pages with this title returns roughly 2,460,000 results. Even Microsoft is susceptible to publishing web pages with the title "Untitled Page". At the time of this writing, a Google search reported 236 such web pages in the Microsoft.com domain.

An ASP.NET page can specify its title in one of the following ways:

- By placing the value directly within the `<title>` element
- Using the `Title` attribute in the `<%@ Page %>` directive
- Programmatically setting the page's `Title` property using code like `Page.Title="title"` or `Page.Header.Title="title"`.

Content pages don't have a `<title>` element, as it's defined in the master page. Therefore, to set a content page's title you can either use the `<%@ Page %>` directive's `Title` attribute or set it programmatically.

Setting the Page's Title Declaratively

A content page's title can be set declaratively through the `Title` attribute of the `<%@ Page %>` directive. This property can be set by directly modifying the `<%@ Page %>` directive or through the Properties window. Let's look at both approaches.

From the Source view, locate the `<%@ Page %>` directive, which is at the top of the page's declarative markup. The `<%@ Page %>` directive for `Default.aspx` follows:

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" Title="Untitled Page" %>
```

The `<%@ Page %>` directive specifies page-specific attributes used by the ASP.NET engine when parsing and compiling the page. This includes its master page file, the location of its code file, and its title, among other information.

By default, when creating a new content page Visual Studio sets the `Title` attribute to Untitled Page. Change `Default.aspx`'s `Title` attribute from "Untitled Page" to "Master Page Tutorials" and then view the page through a browser. Figure 1 shows the browser's Title bar, which reflects the new page title.

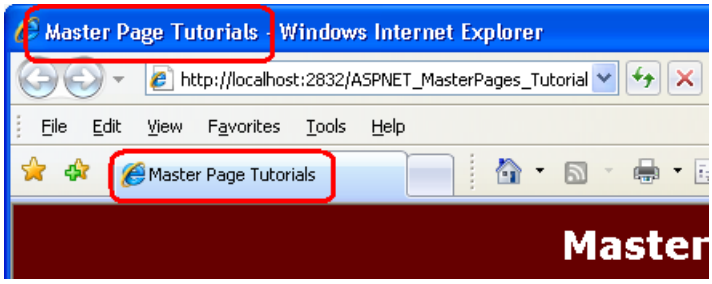


Figure 01: The Browser's Title Bar Now Shows "Master Page Tutorials" Instead of "Untitled Page"

The page's title may also be set from the Properties window. From the Properties window, select DOCUMENT from the drop-down list to load the page-level properties, which includes the `Title` property. Figure 2 shows the Properties window after `Title` has been set to "Master Page Tutorials".

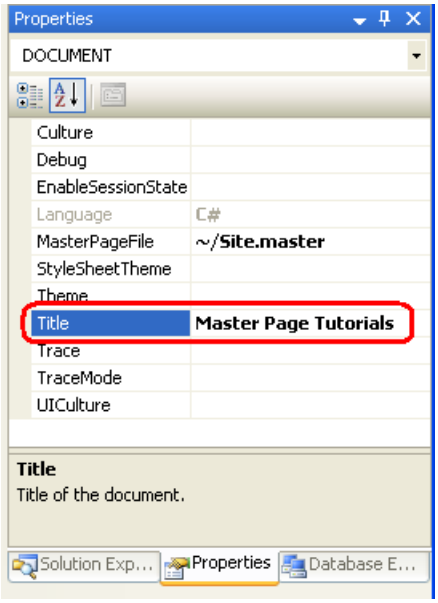


Figure 02: You May Configure the Title from the Properties Window, Too

Setting the Page's Title Programmatically

The master page's `<head runat="server">` markup is translated into an `HtmlHead` class instance when the page is rendered by the ASP.NET engine. The `HtmlHead` class has a `Title` property whose value is reflected in the rendered `<title>` element. This property is accessible from an ASP.NET page's code-behind class via `Page.Header.Title`; this same property can also be accessed via `Page.Title`.

To practice setting the page's title programmatically, navigate to the `About.aspx` page's code-behind class and create an event handler for the page's `Load` event. Next, set the page's title to "Master Page Tutorials :: About :: *date*", where *date* is the current date. After adding this code your `Page_Load` event handler should look similar to the following:

```
protected void Page_Load(object sender, EventArgs e)
{
    Page.Title = string.Format("Master Page Tutorials :: About :: {0:d}", DateTime.Now);
}
```

Figure 3 shows the browser's title bar when visiting the `About.aspx` page.

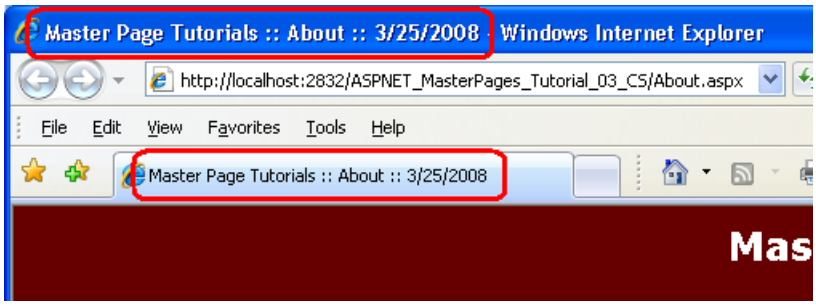


Figure 03: The Page's Title is Programmatically Set and Includes the Current Date

Step 2: Automatically Assigning a Page Title

As we saw in Step 1, a page's title can be set declaratively or programmatically. If you forget to explicitly change the title to something more descriptive, however, your page will have the default title, "Untitled Page". Ideally, the page's title would be set automatically for us in the event that we don't explicitly specify its value. For example, if at runtime the page's title is "Untitled Page", we might want to have the title automatically updated to be the same as the ASP.NET page's filename. The good news is that with a little bit of upfront work it is possible to have the title automatically assigned.

All ASP.NET web pages derive from the `Page` class in the `System.Web.UI` namespace. The `Page` class defines the minimal functionality needed by an ASP.NET page and exposes essential properties like `IsPostBack`, `IsValid`, `Request`, and `Response`, among many others. Oftentimes, every page in a web application requires additional features or functionality. A common way of providing this is to create a custom base page class. A custom base page class is a class you create that derives from the `Page` class and includes additional functionality. Once this base class has been created, you can have your ASP.NET pages derive from it (rather than the `Page` class), thereby offering the extended functionality to your ASP.NET pages.

In this step we create a base page that automatically sets the page's title to the ASP.NET page's filename if the title has not otherwise been explicitly set. Step 3 looks at setting the page's title based on the site map.

> **NOTE**
> A thorough examination of creating and using custom base page classes is beyond the scope of this tutorial series. For more information, read Using a Custom Base Class for Your ASP.NET Pages' Code-Behind Classes.

Creating the Base Page Class

Our first task is to create a base page class, which is a class that extends the `Page` class. Start by adding an `App_Code` folder to your project by right-clicking on the project name in the Solution Explorer, choosing Add ASP.NET Folder, and then selecting `App_Code`. Next, right-click on the `App_Code` folder and add a new class named `BasePage.cs`. Figure 4 shows the Solution Explorer after the `App_Code` folder and `BasePage.cs` class have been added.

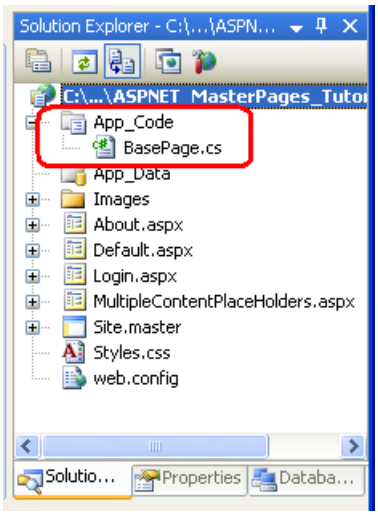


Figure 04: Add an `App_Code` Folder and a Class Named `BasePage`

> **NOTE**
> Visual Studio supports two modes of project management: Web Site Projects and Web Application Projects. The `App_Code` folder is designed to be used with the Web Site Project model. If you are using the Web Application Project model, place the `BasePage.cs` class in a folder named something other than `App_Code`, such as `Classes`. For more information on this topic, refer to Migrating a Web Site Project to a Web Application Project.

Because the custom base page serves as the base class for ASP.NET pages' code-behind classes, it needs to extend the `Page` class.

```
public class BasePage : System.Web.UI.Page
{

}
```

Whenever an ASP.NET page is requested it proceeds through a series of stages, culminating in the requested page being rendered into HTML. We can tap into a stage by overriding the `Page` class's `OnEvent` method. For our base page let's automatically set the title if it has not been explicitly specified by the `LoadComplete` stage (which, as you might have guessed, occurs after the `Load` stage).

To accomplish this, override the `OnLoadComplete` method and enter the following code:

```
protected override void OnLoadComplete(EventArgs e)
{
    // Set the page's title, if necessary
    if (string.IsNullOrEmpty(Page.Title) || Page.Title == "Untitled Page")
    {
        // Determine the filename for this page
        string fileName = System.IO.Path.GetFileNameWithoutExtension(Request.PhysicalPath);

        Page.Title = fileName;
    }

    base.OnLoadComplete(e);
}
```

The `OnLoadComplete` method starts by determining if the `Title` property has not yet been explicitly set. If the `Title` property is `null`, an empty string, or has the value "Untitled Page", it is assigned to the filename of the requested ASP.NET page. The physical path to the requested ASP.NET page - `C:\MySites\Tutorial03\Login.aspx`,

for example - is accessible via the `Request.PhysicalPath` property. The `Path.GetFileNameWithoutExtension` method is used to pull out just the filename portion, and this filename is then assigned to the `Page.Title` property.

> **NOTE**
> I invite you to enhance this logic to improve the format of the title. For example, if the page's filename is `Company-Products.aspx`, the above code will produce the title "Company-Products", but ideally the dash would be replaced with a space, as in "Company Products". Also, consider adding a space whenever there's a case change. That is, consider adding code that transforms the filename `OurBusinessHours.aspx` to a title of "Our Business Hours".

Having the Content Pages Inherit the Base Page Class

We now need to update the ASP.NET pages in our site to derive from the custom base page (`BasePage`) instead of the `Page` class. To accomplish this go to each code-behind class and change the class declaration from:

```
public partial class ClassName : System.Web.UI.Page
```

To:

```
public partial class ClassName : BasePage
```

After doing so, visit the site through a browser. If you visit a page whose title is explicitly set, such as `Default.aspx` or `About.aspx`, the explicitly specified title is used. If, however, you visit a page whose title has not been changed from the default ("Untitled Page"), the base page class sets the title to the page's filename.

Figure 5 shows the `MultipleContentPlaceHolders.aspx` page when viewed through a browser. Note that the title is precisely the page's filename (less the extension), "MultipleContentPlaceHolders".

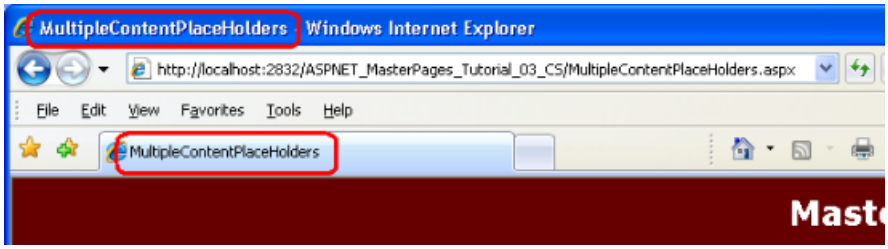


Figure 05: If a Title is Not Explicitly Specified, the Page's Filename is Automatically Used (Click to view full-size image)

Step 3: Basing the Page Title on the Site Map

ASP.NET offers a robust site map framework that enables page developers to define a hierarchical site map in an external resource (such as an XML file or database table) along with Web controls for displaying information about the site map (such as the SiteMapPath, Menu, and TreeView controls).

The site map structure can also be accessed programmatically from an ASP.NET page's code-behind class. In this manner we can automatically set a page's title to the title of its corresponding node in the site map. Let's enhance the `BasePage` class created in Step 2 so that it offers this functionality. But first we need to create a site map for our site.

> **NOTE**
> This tutorial assumes the reader already is familiar with ASP.NET's site map features. For more information on using the site map, consult my multi-part article series, Examining ASP.NET's Site Navigation.

Creating the Site Map

The site map system is built atop the provider model, which decouples the site map API from the logic that serializes site map information between memory and a persistent store. The .NET Framework ships with the `XmlSiteMapProvider` class, which is the default site map provider. As its name implies, `XmlSiteMapProvider` uses an XML file as its site map store. Let's use this provider for defining our site map.

Start by creating a site map file in the website's root folder named `Web.sitemap`. To accomplish this, right-click on the website name in Solution Explorer, choose Add New Item, and select the Site Map template. Ensure that the file is named `Web.sitemap` and click Add.

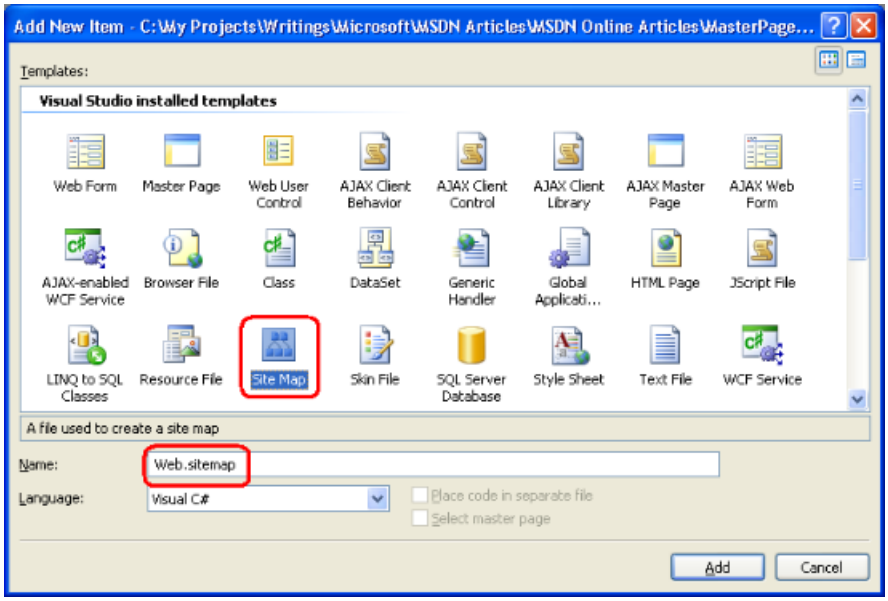


Figure 06: Add a File Named `Web.sitemap` to the Website's Root Folder (Click to view full-size image)

Add the following XML to the `Web.sitemap` file:

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
 <siteMapNode url="~/Default.aspx" title="Home">
 <siteMapNode url="~/About.aspx" title="About the Author" />
 <siteMapNode url="~/MultipleContentPlaceHolders.aspx" title="Using Multiple ContentPlaceHolder Controls" />
 </siteMapNode>
</siteMap>
```

This XML defines the hierarchical site map structure shown in Figure 7.

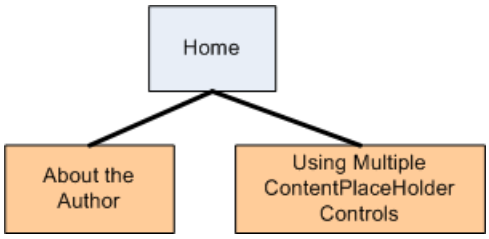


Figure 07: The Site Map is Currently Composed of Three Site Map Nodes

We will update the site map structure in future tutorials as we add new examples.

Updating the Master Page to Include Navigation Web Controls

Now that we have a site map defined, let's update the master page to include navigation Web controls. Specifically, let's add a ListView control to the left column in the Lessons section that renders an unordered list with a list item for each node defined in the site map.

> **NOTE**
> The ListView control is new to ASP.NET version 3.5. If you are using a prior version of ASP.NET, use the Repeater control instead. For more information on the ListView control, see Using ASP.NET 3.5's ListView and DataPager Controls.

Start by removing the existing unordered list markup from the Lessons section. Next, drag a ListView control from the Toolbox and drop it beneath the Lessons heading. The ListView is located in the Data section of the Toolbox, alongside the other view controls: the GridView, DetailsView, and FormView. Set the ListView's ID property to `LessonsList`.

From the Data Source Configuration Wizard choose to bind the ListView to a new SiteMapDataSource control named `LessonsDataSource`. The SiteMapDataSource control returns the hierarchical structure from the site map system.

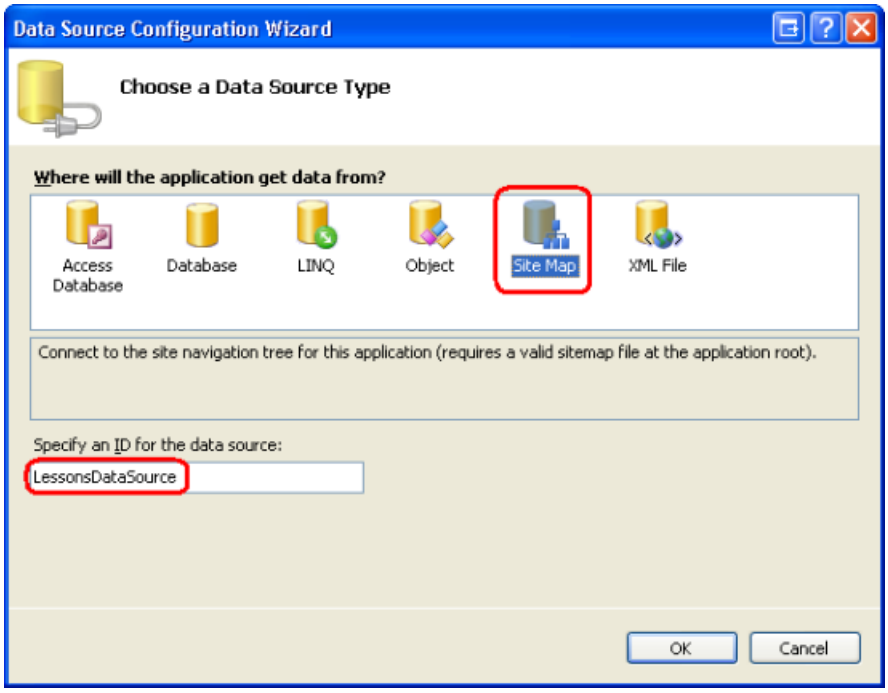


Figure 08: Bind a SiteMapDataSource Control to the `LessonsList` ListView Control (Click to view full-size image)

After creating the SiteMapDataSource control, we need to define the ListView's templates so that it renders an unordered list with a list item for each node returned by the SiteMapDataSource control. This can be accomplished using the following template markup:

```
<asp:ListView ID="LessonsList" runat="server" DataSourceID="LessonsDataSource">
 <LayoutTemplate>
 <ul>
 <asp:PlaceHolder runat="server" ID="itemPlaceholder" />
 </ul>
 </LayoutTemplate>

 <ItemTemplate>
 <li><asp:HyperLink runat="server" ID="lnkLesson" NavigateUrl='<%# Eval("Url") %>'
 Text='<%# Eval("Title") %>' /></li>
 </ItemTemplate>
</asp:ListView>
```

The `LayoutTemplate` generates the markup for an unordered list (`<ul>...</ul>`) while the `ItemTemplate` renders each item returned by the SiteMapDataSource as a list item (`<li>`) that contains a link to the particular lesson.

After configuring the ListView's templates, visit the website. As Figure 9 shows, the Lessons section contains a single bulleted item, Home. Where are the About and Using Multiple ContentPlaceHolder Controls lessons? The SiteMapDataSource is designed to return a hierarchical set of data, but the ListView control can only display a single level of the hierarchy. Consequently, only the first level of site map nodes returned by the SiteMapDataSource is displayed.

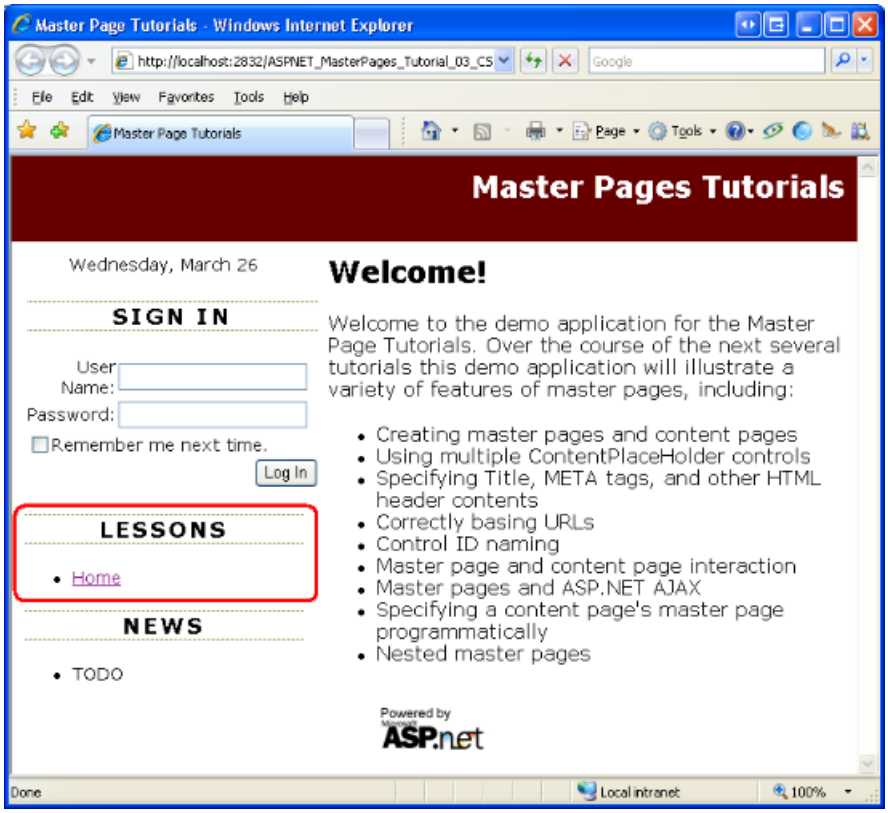


Figure 09: The Lessons Section Contains a Single List Item (Click to view full-size image)

To display multiple levels we could nest multiple ListViews within the `ItemTemplate`. This technique was examined in the *Master Pages and Site Navigation* tutorial of my Working with Data tutorial series. However, for this tutorial series our site map will contain just a two levels: Home (the top level); and each lesson as a child of Home. Rather than crafting a nested ListView, we can instead instruct the SiteMapDataSource to not return the starting node by setting its `ShowStartingNode` property to `false`. The net effect is that the SiteMapDataSource starts by returning the second tier of site map nodes.

With this change, the ListView displays bullet items for the About and Using Multiple ContentPlaceHolder Controls lessons, but omits a bullet item for Home. To remedy this, we can explicitly add a bullet item for Home in the `LayoutTemplate`:

```
<LayoutTemplate>
 <ul>
 <li><asp:HyperLink runat="server" ID="lnkLesson"
 NavigateUrl="~/Default.aspx" Text="Home" /></li>
 <asp:PlaceHolder runat="server" ID="itemPlaceholder" />
 </ul>
</LayoutTemplate>
```

By configuring the SiteMapDataSource to omit the starting node and explicitly adding a Home bullet item, the Lessons section now displays the intended output.

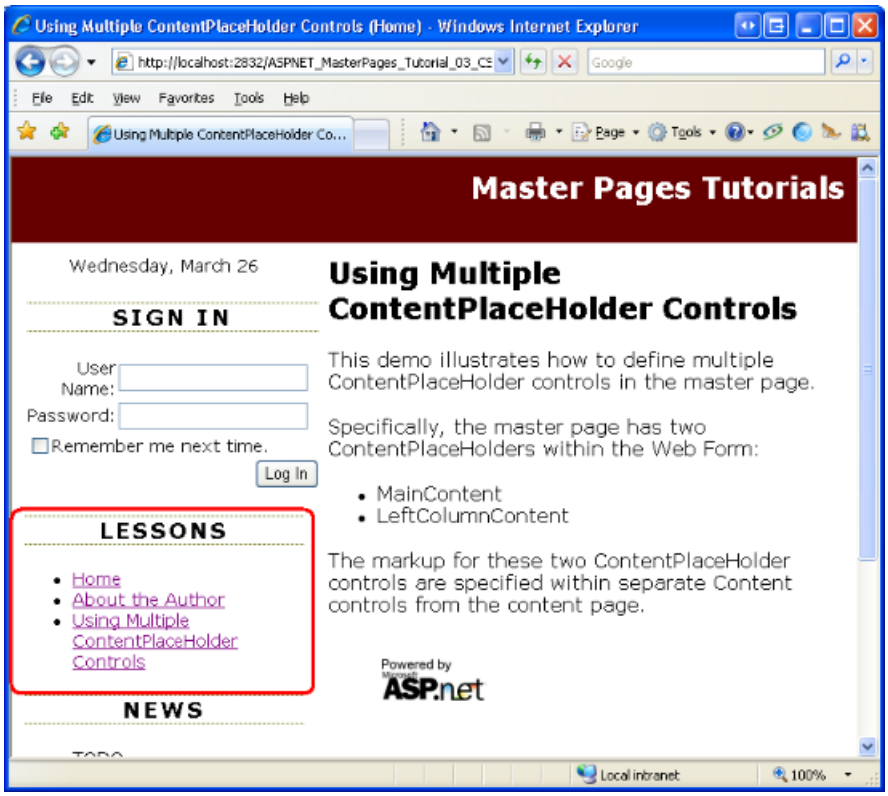


Figure 10: The Lessons Section Contains a Bullet Item for Home and Each Child Node (Click to view full-size image)

Setting the Title Based on the Site Map

With the site map in place, we can update our `BasePage` class to use the title specified in the site map. As we did in Step 2, we only want to use the site map node's title if the page's title has not been explicitly set by the page developer. If the page being requested does not have an explicitly set page title and is not found in the site map, then we'll fall back to using the requested page's filename (less the extension), as we did in Step 2. Figure 11 illustrates this decision process.

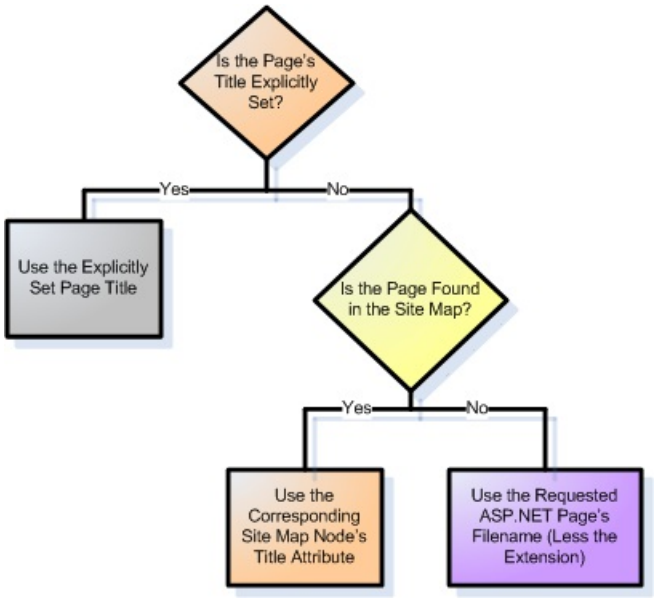


Figure 11: In the Absence of an Explicitly Set Page Title, the Corresponding Site Map Node's Title is Used

Update the `BasePage` class's `OnLoadComplete` method to include the following code:

```
protected override void OnLoadComplete(EventArgs e)
{
    // Set the page's title, if necessary
    if (string.IsNullOrEmpty(Page.Title) || Page.Title == "Untitled Page")
    {
        // Is this page defined in the site map?
        string newTitle = null;

        SiteMapNode current = SiteMap.CurrentNode;
        if (current != null)
        {
            newTitle = current.Title;
        }
        else
        {
            // Determine the filename for this page
            newTitle = System.IO.Path.GetFileNameWithoutExtension(Request.PhysicalPath);
        }

        Page.Title = newTitle;
    }

    base.OnLoadComplete(e);
}
```

As before, the `OnLoadComplete` method starts by determining whether the page's title has been explicitly set. If `Page.Title` is `null`, an empty string, or is assigned the value "Untitled Page" then the code automatically assigns a value to `Page.Title`.

To determine the title to use, the code starts by referencing the `SiteMap` class's `CurrentNode` property. `CurrentNode` returns the `SiteMapNode` instance in the site map that corresponds to the currently requested page. Assuming the currently requested page is found within the site map, the `SiteMapNode`'s `Title` property is assigned to the page's title. If the currently requested page is not in the site map, `CurrentNode` returns `null` and the requested page's filename is used as the title (as was done in Step 2).

Figure 12 shows the `MultipleContentPlaceHolders.aspx` page when viewed through a browser. Because this page's title is not explicitly set, its corresponding site map node's title is used instead.

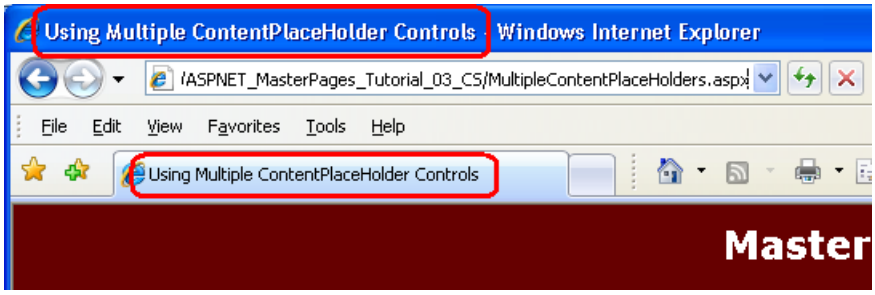


Figure 12: The `MultipleContentPlaceHolders.aspx` Page's Title is Pulled from the Site Map

Step 4: Adding Other Page-Specific Markup to the `<head>` Section

Steps 1, 2, and 3 looked at customizing the `<title>` element on a page-by-page basis. In addition to `<title>`, the `<head>` section may contain `<meta>` elements and `<link>` elements. As noted earlier in this tutorial, `Site.master`'s `<head>` section includes a `<link>` element to `Styles.css`. Because this `<link>` element is defined within the master page, it is included in the `<head>` section for all content pages. But how do we go about adding `<meta>` and `<link>` elements on a page-by-page basis?

The easiest way to add page-specific content to the `<head>` section is by creating a ContentPlaceHolder control in the master page. We already have such a ContentPlaceHolder (named `head`). Therefore, to add custom

`<head>` markup, create a corresponding Content control in the page and place the markup there.

To illustrate adding custom `<head>` markup to a page, let's include a `<meta>` description element to our current set of content pages. The `<meta>` description element provides a brief description about the web page; most search engines incorporate this information in some form when displaying search results.

A `<meta>` description element has the following form:

```
<meta name="description" content="description of the web page" />
```

To add this markup to a content page, add the above text to the Content control that maps to the master page's head ContentPlaceHolder. For example, to define a `<meta>` description element for `Default.aspx`, add the following markup:

```
<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
 <meta name="description" content="Welcome to Scott Mitchell's Master Page Tutorials series." />
</asp:Content>
```

Because the head ContentPlaceHolder is not within the HTML page's body, the markup added to the Content control is not displayed in the Design view. To see the `<meta>` description element visit `Default.aspx` through a browser. After the page has been loaded, view the source and note that the `<head>` section includes the markup specified in the Content control.

Take a moment to add `<meta>` description elements to `About.aspx`, `MultipleContentPlaceHolders.aspx`, and `Login.aspx`.

Programmatically Adding Markup to the `<head>` Region

The head ContentPlaceHolder allows us to declaratively add custom markup to the master page's `<head>` region. Custom markup may also be added programmatically. Recall that the `Page` class's `Header` property returns the `HtmlHead` instance defined in the master page (the `<head runat="server">`).

Being able to programmatically add content to the `<head>` region is useful when the content to add is dynamic. Perhaps it's based on the user visiting the page; maybe it's being pulled from a database. Regardless of the reason, you can add content to the `HtmlHead` by adding controls to its Controls collection like so:

```
// Programmatically add a <meta> element to the Header
HtmlMeta keywords = new HtmlMeta();
keywords.Name = "keywords";
keywords.Content = "master page,asp.net,tutorial";

Page.Header.Controls.Add(keywords);
```

The above code adds the `<meta>` keywords element to the `<head>` region, which provides a comma-delimited list of keywords that describe the page. Note that to add a `<meta>` tag you create an `HtmlMeta` instance, set its `Name` and `Content` properties, and then add it to the `Header`'s `Controls` collection. Similarly, to programmatically add a `<link>` element, create an `HtmlLink` object, set its properties, and then add it to the `Header`'s `Controls` collection.

> **NOTE**
>
> To add arbitrary markup, create a `LiteralControl` instance, set its `Text` property, and then add it to the `Header`'s `Controls` collection.

Summary

In this tutorial we looked at a variety of ways to add `<head>` region markup on a page-by-page basis. A master page should include an `HtmlHead` instance (`<head runat="server">`) with a ContentPlaceHolder. The `HtmlHead` instance allows content pages to programmatically access the `<head>` region and to declaratively and programmatically set the page's title; the ContentPlaceHolder control enables custom markup to be added to the `<head>` section declaratively through a Content control.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Dynamically Setting the Page's Title in ASP.NET
- Examining ASP.NET's Site Navigation
- How To Use HTML Meta Tags
- Master Pages in ASP.NET
- Using ASP.NET 3.5's ListView and DataPager Controls
- Using a Custom Base Class for Your ASP.NET Pages' Code-Behind Classes

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Zack Jones and Suchi Banerjee. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

URLs in Master Pages (C#)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Addresses how URLs in the master page can break due to the master page file being in a different relative directory than the content page. Looks at rebasing URLs via ~ in the declarative syntax and using ResolveUrl and ResolveClientUrl programmatically. (Also look at

Introduction

In all the examples we've seen thus far, the master and content pages have been located in the same folder (the website's root folder). But there's no reason why the master and content pages must be in the same folder. You can certainly create content pages in subfolders. Similarly, you might create a `~/MasterPages/` folder where you place your site's master pages.

One potential issue with placing master and content pages in different folders involves broken URLs. If the master page contains relative URLs in hyperlinks, images, or other elements, the link will be invalid for content pages that reside in a different folder. In this tutorial we examine the source of this problem as well as workarounds.

The Problem with Relative URLs

A URL on a web page is said to be a *relative URL* if the location of the resource it points to is relative to the web page's location in the website's folder structure. Any URL that does not start with a leading forward slash (`/`) or a protocol (such as `http://`) is relative because it is resolved by the browser based on the location of the web page that contains the URL.

For example, our website has an `~/Images/` folder with a single image file, `PoweredByASPNET.gif`. The master page file `Site.master` has an `<img>` element in the `footerContent` region with the following markup:

```
<div id="footerContent">
 <img src="Images/PoweredByASPNET.gif" alt="Powered by ASP.NET!" />
</div>
```

The `src` attribute value in the `<img>` element is a relative URL because it does not start with `/` or `http://`. In short, the `src` attribute value tells the browser to look in the `Images` subfolder for a file named `PoweredByASPNET.gif`.

When visiting a content page, the above markup is sent directly to the browser. Take a moment to visit `About.aspx` and view the HTML source sent to the browser. You will find that the exact same markup in the master page was sent to the browser.

```
<div id="footerContent">
 <img src="Images/PoweredByASPNET.gif" alt="Powered by ASP.NET!" />
</div>
```

If the content page is in the root folder (as is `About.aspx`) everything works as expected because there is an `Images` subfolder relative to the root folder. However, things break down if the content page is in a different folder

than the master page. To illustrate this, create a subfolder named `Admin`. Next, add a content page named `Default.aspx` to the `Admin` folder, making sure to bind the new page to the `Site.master` master page.

> **NOTE**
>
> In the *Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page* tutorial we created a custom base page class named `BasePage` that automatically set the content page's title (if it was not explicitly assigned). Don't forget to have the newly created page's code-behind class derive from `BasePage` so that it can utilize this functionality.

After you have created this content page, your Solution Explorer should look similar to Figure 1.

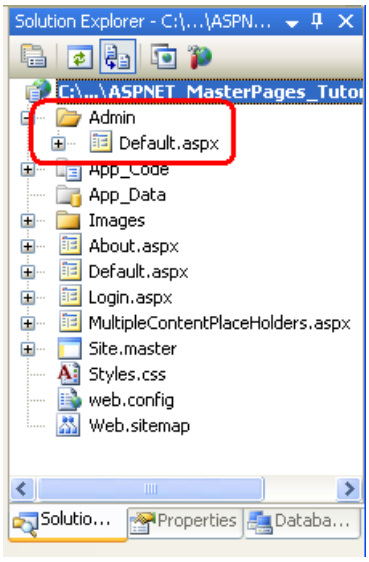


Figure 01: A New Folder and ASP.NET Page Have Been Added to the Project

Next, update the `Web.sitemap` file to include a new `<siteMapNode>` entry for this lesson. The following XML shows the complete `Web.sitemap` markup, which now includes the addition of a third `<siteMapNode>` element.

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
 <siteMapNode url="~/Default.aspx" title="Home">
 <siteMapNode url="~/About.aspx" title="About the Author" />
 <siteMapNode url="~/MultipleContentPlaceHolders.aspx" title="Using Multiple ContentPlaceHolder Controls" />
 <siteMapNode url="~/Admin/Default.aspx" title="Rebasing URLs" />
 </siteMapNode>
</siteMap>
```

The newly created `Default.aspx` page should have four Content controls corresponding to the four ContentPlaceHolders in `Site.master`. Add some text to the Content control referencing the `MainContent` ContentPlaceHolder and then visit the page through a browser. As Figure 2 shows, the browser cannot find the `PoweredByASPNET.gif` image file. What's going on here?

The `~/Admin/Default.aspx` content page is sent the same HTML for the `footerContent` region as was the `About.aspx` page:

```
<div id="footerContent">
 <img src="Images/PoweredByASPNET.gif" alt="Powered by ASP.NET!" />
</div>
```

Because the `<img>` element's `src` attribute is a relative URL, the browser attempts to look for an `Images` folder relative to the web page's folder location. In other words, the browser is looking for the image file `Admin/Images/PoweredByASPNET.gif`.

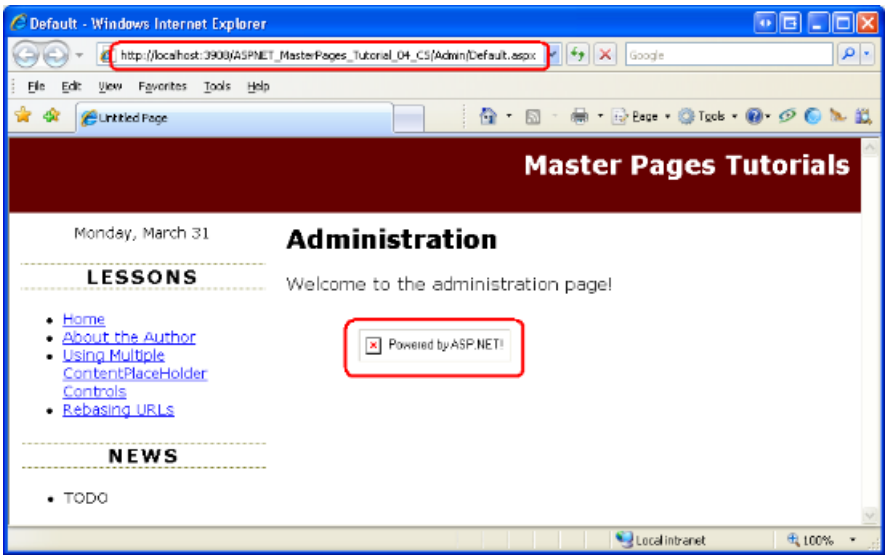


Figure 02: The `PoweredByASPNET.gif` Image File Cannot Be Found (Click to view full-size image)

Replacing Relative URLs with Absolute URLs

The opposite of a relative URL is an *absolute URL*, which is one that starts with a forward slash (`/`) or a protocol such as `http://`. Because an absolute URL specifies the location of a resource from a known fixed point, the same absolute URL is valid in any web page, regardless of the web page's location in the website's folder structure.

To remedy the broken image shown in Figure 2, we need to update the `<img>` element's `src` attribute so that it uses an absolute URL instead of a relative one. To determine the correct absolute URL, visit one of the web pages in your website and examine the Address bar. As the Address bar in Figure 2 shows, the fully qualified path to the web application is `http://localhost:3908/ASPNET_MasterPages_Tutorial_04_CS/`. Therefore, we could update the `<img>` element's `src` attribute to either of the following two absolute URLs:

- `/ASPNET_MasterPages_Tutorial_04_CS/Images/PoweredByASPNET.gif`
- `http://localhost:3908/ASPNET_MasterPages_Tutorial_04_CS/Images/PoweredByASPNET.gif`

Take a moment to update the `<img>` element's `src` attribute to an absolute URL using one of the forms shown above and then visit the `~/Admin/Default.aspx` page through a browser. This time the browser will correctly find and display the `PoweredByASPNET.gif` image file (see Figure 3).

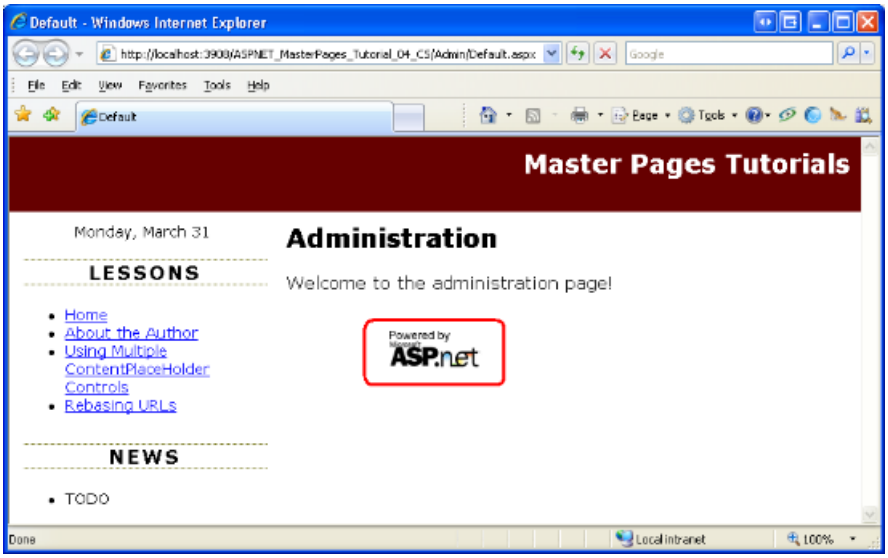


Figure 03: The `PoweredByASPNET.gif` Image is Now Displayed (Click to view full-size image)

While hard coding in the absolute URL works, it tightly couples your HTML to the website's server and folder location, which may change. Using an absolute URL of the form `http://localhost:3908/...` is brittle because the

port number preceding `localhost` is selected automatically each time Visual Studio's built-in ASP.NET Development Web Server is started. Similarly, the `http://localhost` part is only valid when testing locally. Once the code is deployed to a production server, the URL base will change to something else, like `http://www.yourserver.com`. The absolute URL in the form `/ASPNET_MasterPages_Tutorial_04_CS/...` also suffers from the same brittleness because oftentimes this application path differs between development and production servers.

The good news is that ASP.NET offers a method for generating a valid relative URL at runtime.

Using `~` and `ResolveClientUrl`

Rather than hard code an absolute URL, ASP.NET allows page developers to use the tilde (`~`) to indicate the root of the web application. For example, earlier in this tutorial I used the notation `~/Admin/Default.aspx` in the text to refer to the `Default.aspx` page in the `Admin` folder. The `~` indicates that the `Admin` folder is a subfolder of the web application's root.

The `Control` class's `ResolveClientUrl` method takes a URL and modifies it to a relative URL appropriate for the web page on which the control resides. For example, calling `ResolveClientUrl("~/Images/PoweredByASPNET.gif")` from `About.aspx` returns `Images/PoweredByASPNET.gif`. Calling it from `~/Admin/Default.aspx`, however, returns `../Images/PoweredByASPNET.gif`.

> **NOTE**
> Because all ASP.NET server controls derive from the `Control` class, all server controls have access to the `ResolveClientUrl` method. Even the `Page` class derives from the `Control` class, meaning that you can use this method directly from your ASP.NET pages' code-behind classes.

Using `~` in the Declarative Markup

Several ASP.NET Web controls include URL-related properties: the HyperLink control has a `NavigateUrl` property; the Image control has an `ImageUrl` property; and so on. When rendered, these controls pass their URL-related property values to `ResolveClientUrl`. Consequently, if these properties contain a `~` to indicate the root of the web application, the URL will be modified to a valid relative URL.

Keep in mind that only ASP.NET server controls transform the `~` in their URL-related properties. If a `~` appears in static HTML markup, such as `<img src="~/Images/PoweredByASPNET.gif" />`, the ASP.NET engine sends the `~` to the browser along with the rest of the HTML content. The browser assumes that the `~` is part of the URL. For example, if the browser receives the markup `<img src="~/Images/PoweredByASPNET.gif" />` it assumes there is a subfolder named `~` with a subfolder `Images` that contains the image file `PoweredByASPNET.gif`.

To fix the image markup in `Site.master`, replace the existing `<img>` element with an ASP.NET Image Web control. Set the Image Web control's `ID` to `PoweredByImage`, its `ImageUrl` property to `~/Images/PoweredByASPNET.gif`, and its `AlternateText` property to "Powered by ASP.NET!"

```
<div id="footerContent">
 <asp:Image ID="PoweredByImage" runat="server" ImageUrl="~/Images/PoweredByASPNET.gif"
    AlternateText="Powered by ASP.NET!" />
</div>
```

After making this change to the master page, revisit the `~/Admin/Default.aspx` page again. This time the `PoweredByASPNET.gif` image file appears in the page (see Figure 3). When the Image Web control is rendered it uses the `ResolveClientUrl` method to resolve its `ImageUrl` property value. In `~/Admin/Default.aspx` the `ImageUrl` is converted into an appropriate relative URL, as the following snippet of HTML source shows:

```
<div id="footerContent">
 <img id="ctl00_PoweredByImage" src="../Images/PoweredByASPNET.gif" alt="Powered by ASP.NET!" />
</div>
```

NOTE

In addition to being used in URL-based Web control properties, the `~` can also be used when calling the `Response.Redirect` and `Server.MapPath` methods, among others. Also, the `ResolveClientUrl` method may be invoked directly from an ASP.NET or master page's declarative markup, if needed; see Fritz Onion's blog entry Using `ResolveClientUrl` in Markup.

Fixing the Master Page's Remaining Relative URLs

In addition to the `<img>` element in the `footerContent` that we just fixed, the master page contains one more relative URL that requires our attention. The `topContent` region includes the link "Master Pages Tutorials," which points to `Default.aspx`.

```
<div id="topContent">
 <a href="Default.aspx">Master Pages Tutorials</a>
</div>
```

Because this URL is relative, it will send the user to the `Default.aspx` page in the folder of the content page they are visiting. To have this link always point to `Default.aspx` in the root folder we need to replace the `<a>` element with a HyperLink Web control so that we can use the `~` notation.

Remove the `<a>` element markup and add a HyperLink control in its place. Set the HyperLink's `ID` to `lnkHome`, its `NavigateUrl` property to `~/Default.aspx`, and its `Text` property to "Master Pages Tutorials."

```
<div id="topContent">
 <asp:HyperLink ID="lnkHome" runat="server" NavigateUrl="~/Default.aspx"
    Text="Master Pages Tutorials" />
</div>
```

That's it! At this point all the URLs in our master page are properly based when rendered by a content page regardless of what folders the master page and content page are located in.

Automatic URL Resolution in the `<head>` Section

In the *Creating a Site-Wide Layout Using Master Pages* tutorial we added a `<link>` to the `Styles.css` file in the `<head>` region:

```
<head runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
 <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
```

While the `<link>` element's `href` attribute is relative, it's automatically converted to an appropriate path at runtime. As we discussed in the *Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page* tutorial, the `<head>` region is actually a server-side control, which enables it to modify the contents of its inner controls when it is rendered.

To verify this, revisit the `~/Admin/Default.aspx` page and view the HTML source sent to the browser. As the snippet

below illustrates, the `<link>` element's `href` attribute has been automatically modified to an appropriate relative URL, `../Styles.css`.

```
<head>
 <title>
 Default
 </title>
 <link href="../Styles.css" rel="stylesheet" type="text/css" />
</head>
```

Summary

Master pages very often include links, images, and other external resources that must be specified via a URL. Because the master page and content pages might not exist in the same folder, it is important to abstain from using relative URLs. While it is possible to use hard coded absolute URLs, doing so tightly couples the absolute URL to the web application. If the absolute URL changes - as it often does when moving or deploying a web application - you'll have to remember to go back and update the absolute URLs.

The ideal approach is to use the tilde (`~`) to indicate the application root. ASP.NET Web controls that contain URL-related properties map the `~` to the application root at runtime. Internally, the Web controls use the `Control` class's `ResolveClientUrl` method to generate a valid relative URL. This method is public and available from every server control (including the `Page` class), so you can use it programmatically from your code-behind classes, if needed.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Master Pages in ASP.NET
- URL Rebasing in a Master Page
- Using `ResolveClientUrl` in Markup

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Control ID Naming in Content Pages (C#)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Illustrates how ContentPlaceHolder controls serve as a naming container and therefore make programmatically working with a control difficult (via FindControl). Looks at this issue and workarounds. Also discusses how to programmatically access the resulting ClientID value.

Introduction

All ASP.NET server controls include an `ID` property that uniquely identifies the control and is the means by which the control is programmatically accessed in the code-behind class. Similarly, the elements in an HTML document may include an `id` attribute that uniquely identifies the element; these `id` values are often used in client-side script to programmatically reference a particular HTML element. Given this, you may assume that when an ASP.NET server control is rendered into HTML, its `ID` value is used as the `id` value of the rendered HTML element. This is not necessarily the case because in certain circumstances a single control with a single `ID` value may appear multiple times in the rendered markup. Consider a GridView control that includes a TemplateField with a Label Web control with an `ID` value of ProductName. When the GridView is bound to its data source at runtime, this Label is repeated once for every GridView row. Each rendered Label needs a unique `id` value.

To handle such scenarios, ASP.NET allows certain controls to be denoted as naming containers. A naming container serves as a new `ID` namespace. Any server controls that appear within the naming container have their rendered `id` value prefixed with the `ID` of the naming container control. For example, the `GridView` and `GridViewRow` classes are both naming containers. Consequently, a Label control defined in a GridView TemplateField with `ID` ProductName is given a rendered `id` value of `GridViewID_GridViewRowID_ProductName`. Because *GridViewRowID* is unique for each GridView row, the resulting `id` values are unique.

> **NOTE**
>
> The `INamingContainer` interface is used to indicate that a particular ASP.NET server control should function as a naming container. The `INamingContainer` interface does not spell out any methods that the server control must implement; rather, it's used as a marker. In generating the rendered markup, if a control implements this interface then the ASP.NET engine automatically prefixes its `ID` value to its descendents' rendered `id` attribute values. This process is discussed in more detail in Step 2.

Naming containers not only change the rendered `id` attribute value, but also affect how the control may be programmatically referenced from the ASP.NET page's code-behind class. The `FindControl("controlID")` method is commonly used to programmatically reference a Web control. However, `FindControl` does not penetrate through naming containers. Consequently, you cannot directly use the `Page.FindControl` method to reference controls within a GridView or other naming container.

As you may have surmised, master pages and ContentPlaceHolders are both implemented as naming containers. In this tutorial we examine how master pages affect HTML element `id` values and ways to programmatically reference Web controls within a content page using `FindControl`.

Step 1: Adding a New ASP.NET Page

To demonstrate the concepts discussed in this tutorial, let's add a new ASP.NET page to our website. Create a new content page named `IDIssues.aspx` in the root folder, binding it to the `Site.master` master page.

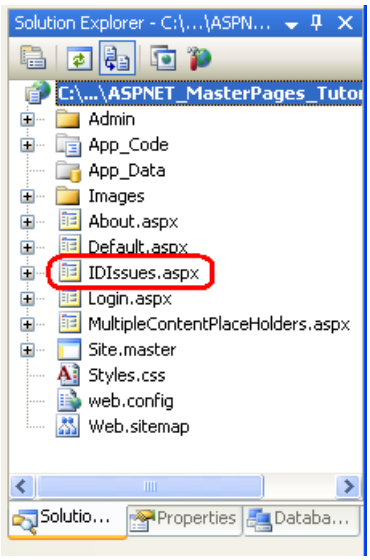


Figure 01: Add the Content Page `IDIssues.aspx` to the Root Folder

Visual Studio automatically creates a Content control for each of the master page's four ContentPlaceHolders. As noted in the *Multiple ContentPlaceHolders and Default Content* tutorial, if a Content control is not present the master page's default ContentPlaceHolder content is emitted instead. Because the `QuickLoginUI` and `LeftColumnContent` ContentPlaceHolders contain suitable default markup for this page, go ahead and remove their corresponding Content controls from `IDIssues.aspx`. At this point, the content page's declarative markup should look like the following:

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true" CodeFile="IDIssues.aspx.cs"
Inherits="IDIssues" Title="Untitled Page" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
</asp:Content>
```

In the *Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page* tutorial we created a custom base page class (`BasePage`) that automatically configures the page's title if it is not explicitly set. For the `IDIssues.aspx` page to employ this functionality, the page's code-behind class must derive from the `BasePage` class (instead of `System.Web.UI.Page`). Modify the code-behind class's definition so that it looks like the following:

```
public partial class IDIssues : BasePage
{
}
```

Finally, update the `Web.sitemap` file to include an entry for this new lesson. Add a `<siteMapNode>` element and set its `title` and `url` attributes to "Control ID Naming Issues" and `~/IDIssues.aspx`, respectively. After making this addition your `Web.sitemap` file's markup should look similar to the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
 <siteMapNode url="~/Default.aspx" title="Home">
 <siteMapNode url="~/About.aspx" title="About the Author" />
 <siteMapNode url="~/MultipleContentPlaceHolders.aspx" title="Using Multiple ContentPlaceHolder Controls" />
 <siteMapNode url="~/Admin/Default.aspx" title="Rebasing URLs" />
 <siteMapNode url="~/IDIssues.aspx" title="Control ID Naming Issues" />
 </siteMapNode>
</siteMap>
```

As Figure 2 illustrates, the new site map entry in `Web.sitemap` is immediately reflected in the Lessons section in the left column.

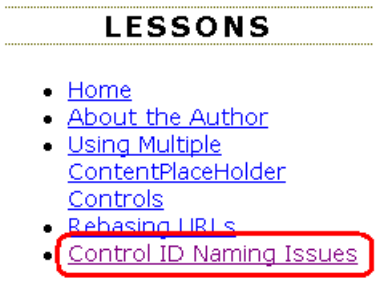


Figure 02: The Lessons Section Now Includes a Link to "Control ID Naming Issues"

Step 2: Examining the Rendered `ID` Changes

To better understand the modifications the ASP.NET engine makes to the rendered `id` values of server controls, let's add a few Web controls to the `IDIssues.aspx` page and then view the rendered markup sent to the browser. Specifically, type in the text "Please enter your age:" followed by a TextBox Web control. Further down on the page add a Button Web control and a Label Web control. Set the TextBox's `ID` and `Columns` properties to `Age` and 3, respectively. Set the Button's `Text` and `ID` properties to "Submit" and `SubmitButton`. Clear out the Label's `Text` property and set its `ID` to `Results`.

At this point your Content control's declarative markup should look similar to the following:

```
<p>
 Please enter your age:
 <asp:TextBox ID="Age" Columns="3" runat="server"></asp:TextBox>
</p>
<p>
 <asp:Button ID="SubmitButton" runat="server" Text="Submit" />
</p>
<p>
 <asp:Label ID="Results" runat="server"></asp:Label>
</p>
```

Figure 3 shows the page when viewed through Visual Studio's designer.

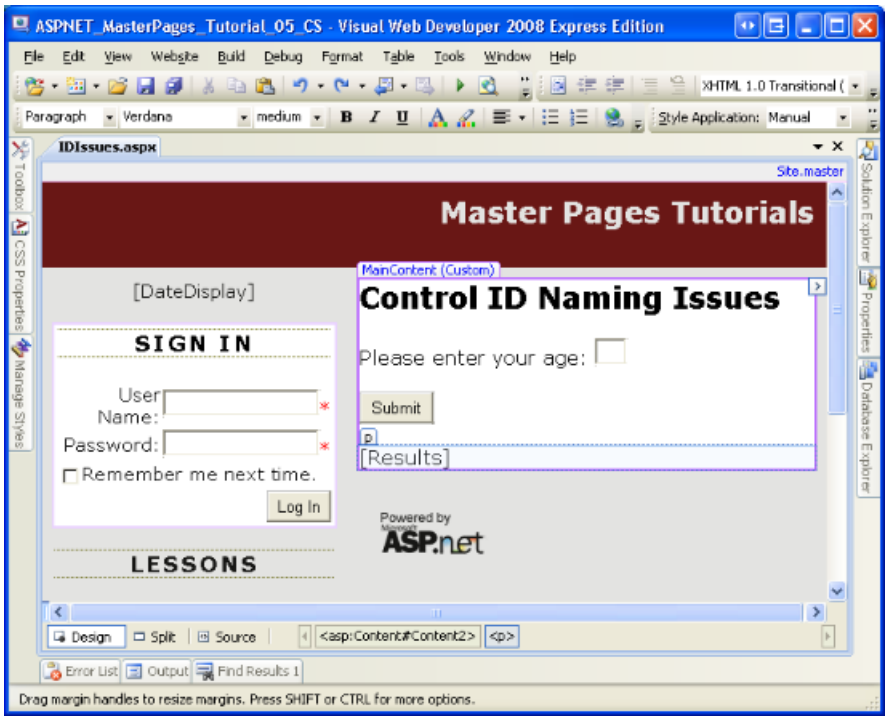


Figure 03: The Page Includes Three Web Controls: a TextBox, Button, and Label (Click to view full-size image)

Visit the page through a browser and then view the HTML source. As the markup below shows, the `id` values of the HTML elements for the TextBox, Button, and Label Web controls are a combination of the `ID` values of the Web controls and the `ID` values of the naming containers in the page.

```
<p>
 Please enter your age:
 <input name="ctl00$MainContent$Age" type="text" size="3" id="ctl00_MainContent_Age" />
</p>
<p>

 <input type="submit" name="ctl00$MainContent$SubmitButton" value="Submit"
id="ctl00_MainContent_SubmitButton" />
</p>
<p>
 <span id="ctl00_MainContent_Results"></span>
</p>
```

As noted earlier in this tutorial, both the master page and its ContentPlaceHolders serve as naming containers. Consequently, both contribute the rendered `ID` values of their nested controls. Take the TextBox's `id` attribute, for instance: `ctl00_MainContent_Age`. Recall that the TextBox control's `ID` value was `Age`. This is prefixed with its ContentPlaceHolder control's `ID` value, `MainContent`. Furthermore, this value is prefixed with the master page's `ID` value, `ctl00`. The net effect is an `id` attribute value consisting of the `ID` values of the master page, the ContentPlaceHolder control, and the TextBox itself.

Figure 4 illustrates this behavior. To determine the rendered `id` of the `Age` TextBox, start with the `ID` value of the TextBox control, `Age`. Next, work your way up the control hierarchy. At each naming container (those nodes with a peach color), prefix the current rendered `id` with the naming container's `id`.

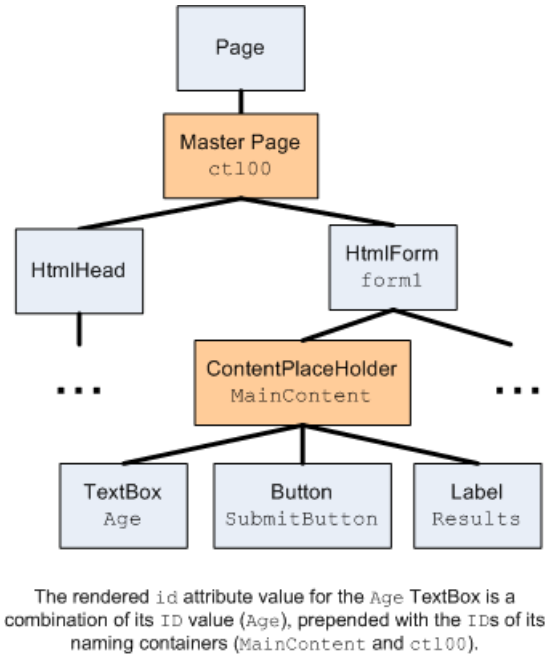


Figure 04: The Rendered `id` Attributes are Based On the `ID` Values of the Naming Containers

> **NOTE**
> As we discussed, the `ctl00` portion of the rendered `id` attribute constitutes the `ID` value of the master page, but you may be wondering how this `ID` value came about. We did not specify it anywhere in our master or content page. Most server controls in an ASP.NET page are added explicitly through the page's declarative markup. The `MainContent` ContentPlaceHolder control was explicitly specified in the markup of `Site.master`; the `Age` TextBox was defined `IDIssues.aspx`'s markup. We can specify the `ID` values for these types of controls through the Properties window or from the declarative syntax. Other controls, like the master page itself, are not defined in the declarative markup. Consequently, their `ID` values must be automatically generated for us. The ASP.NET engine sets the `ID` values at runtime for those controls whose IDs have not been explicitly set. It uses the naming pattern `ctlXX`, where *XX* is a sequentially increasing integer value.

Because the master page itself serves as a naming container, the Web controls defined in the master page also have altered rendered `id` attribute values. For example, the `DisplayDate` Label we added to the master page in the *Creating a Site-Wide Layout with Master Pages* tutorial has the following rendered markup:

```
<span id="ctl00_DateDisplay">current date</span>
```

Note that the `id` attribute includes both the master page's `ID` value (`ctl00`) and the `ID` value of the Label Web control (`DateDisplay`).

Step 3: Programmatically Referencing Web Controls via `FindControl`

Every ASP.NET server control includes a `FindControl("controlID")` method that searches the control's descendents for a control named *controlID*. If such a control is found, it is returned; if no matching control is found, `FindControl` returns `null`.

`FindControl` is useful in scenarios where you need to access a control but you don't have a direct reference to it. When working with data Web controls like the GridView, for example, the controls within the GridView's fields are defined once in the declarative syntax, but at runtime an instance of the control is created for each GridView row. Consequently, the controls generated at runtime exist, but we do not have a direct reference available from the code-behind class. As a result we need to use `FindControl` to programmatically work with a specific control within the GridView's fields. (For more information on using `FindControl` to access the controls within a data

Web control's templates, see Custom Formatting Based Upon Data.) This same scenario occurs when dynamically adding Web controls to a Web Form, a topic discussed in Creating Dynamic Data Entry User Interfaces.

To illustrate using the `FindControl` method to search for controls within a content page, create an event handler for the `SubmitButton`'s `Click` event. In the event handler, add the following code, which programmatically references the `Age` TextBox and `Results` Label using the `FindControl` method and then displays a message in `Results` based on the user's input.

> **NOTE**
>
> Of course, we don't need to use `FindControl` to reference the Label and TextBox controls for this example. We could reference them directly via their `ID` property values. I use `FindControl` here to illustrate what happens when using `FindControl` from a content page.

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
    Label ResultsLabel = FindControl("Results") as Label;
    TextBox AgeTextBox = Page.FindControl("Age") as TextBox;

    ResultsLabel.Text = string.Format("You are {0} years old!", AgeTextBox.Text);
}
```

While the syntax used to call the `FindControl` method differs slightly in the first two lines of `SubmitButton_Click`, they are semantically equivalent. Recall that all ASP.NET server controls include a `FindControl` method. This includes the `Page` class, from which all ASP.NET code-behind classes must derive from. Therefore, calling `FindControl("controlID")` is equivalent to calling `Page.FindControl("controlID")`, assuming you haven't overridden the `FindControl` method in your code-behind class or in a custom base class.

After entering this code, visit the `IDIssues.aspx` page through a browser, enter your age, and click the "Submit" button. Upon clicking the "Submit" button a `NullReferenceException` is raised (see Figure 5).

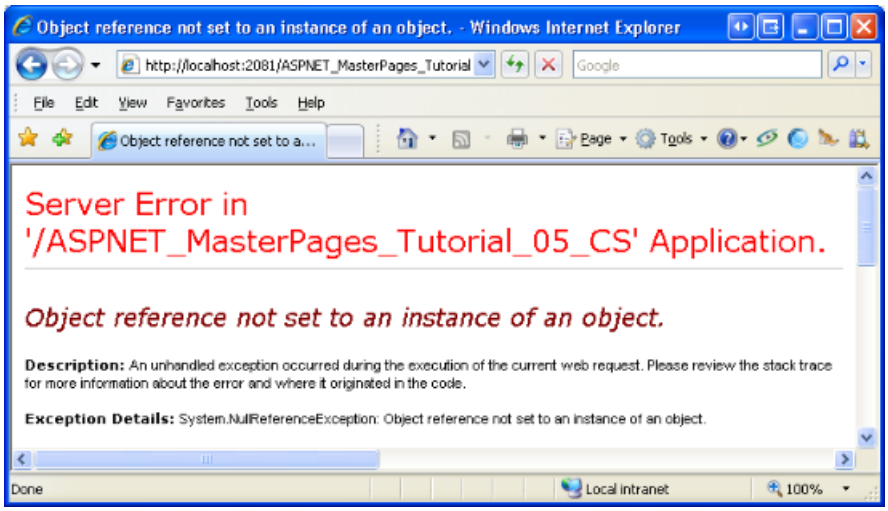


Figure 05: A `NullReferenceException` is Raised (Click to view full-size image)

If you set a breakpoint in the `SubmitButton_Click` event handler you will see that both calls to `FindControl` return a `null` value. The `NullReferenceException` is raised when we attempt to access the `Age` TextBox's `Text` property.

The problem is that `Control.FindControl` only searches *Control*'s descendents that are *in the same naming container*. Because the master page constitutes a new naming container, a call to `Page.FindControl("controlID")` never permeates the master page object `ctl00`. (Refer back to Figure 4 to view the control hierarchy, which shows the `Page` object as the parent of the master page object `ctl00`.) Therefore, the `Results` Label and `Age`

TextBox are not found and `ResultsLabel` and `AgeTextBox` are assigned values of `null`.

There are two workarounds to this challenge: we can drill down, one naming container at a time, to the appropriate control; or we can create our own `FindControl` method that permeates naming containers. Let's examine each of these options.

Drilling Into the Appropriate Naming Container

To use `FindControl` to reference the `Results` Label or `Age` TextBox, we need to call `FindControl` from an ancestor control in the same naming container. As Figure 4 showed, the `MainContent` ContentPlaceHolder control is the only ancestor of `Results` or `Age` that is within the same naming container. In other words, calling the `FindControl` method from the `MainContent` control, as shown in the code snippet below, correctly returns a reference to the `Results` or `Age` controls.

```
Label ResultsLabel = MainContent.FindControl("Results") as Label;
TextBox AgeTextBox = MainContent.FindControl("Age") as TextBox;
```

However, we cannot work with the `MainContent` ContentPlaceHolder from our content page's code-behind class using the above syntax because the ContentPlaceHolder is defined in the master page. Instead, we have to use `FindControl` to get a reference to `MainContent`. Replace the code in the `SubmitButton_Click` event handler with the following modifications:

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
    ContentPlaceHolder MainContent = FindControl("MainContent") as ContentPlaceHolder;

    Label ResultsLabel = MainContent.FindControl("Results") as Label;
    TextBox AgeTextBox = MainContent.FindControl("Age") as TextBox;

    ResultsLabel.Text = string.Format("You are {0} years old!", AgeTextBox.Text);
}
```

If you visit the page through a browser, enter your age, and click the "Submit" button, a `NullReferenceException` is raised. If you set a breakpoint in the `SubmitButton_Click` event handler you will see that this exception occurs when attempting to call the `MainContent` object's `FindControl` method. The `MainContent` object is `null` because the `FindControl` method cannot locate an object named "MainContent". The underlying reason is the same as with the `Results` Label and `Age` TextBox controls: `FindControl` starts its search from the top of the control hierarchy and does not penetrate naming containers, but the `MainContent` ContentPlaceHolder is within the master page, which is a naming container.

Before we can use `FindControl` to get a reference to `MainContent`, we first need a reference to the master page control. Once we have a reference to the master page we can get a reference to the `MainContent` ContentPlaceHolder via `FindControl` and, from there, references to the `Results` Label and `Age` TextBox (again, through using `FindControl`). But how do we get a reference to the master page? By inspecting the `id` attributes in the rendered markup it's evident that the master page's `ID` value is `ctl00`. Therefore, we could use `Page.FindControl("ctl00")` to get a reference to the master page, then use that object to get a reference to `MainContent`, and so on. The following snippet illustrates this logic:

```
// Get a reference to the master page
MasterPage ctl00 = FindControl("ctl00") as MasterPage;

// Get a reference to the ContentPlaceHolder
ContentPlaceHolder MainContent = ctl00.FindControl("MainContent") as ContentPlaceHolder;

// Reference the Label and TextBox controls
Label ResultsLabel = MainContent.FindControl("Results") as Label;
TextBox AgeTextBox = MainContent.FindControl("Age") as TextBox;
```

While this code will certainly work, it assumes that the master page's autogenerated `ID` will always be `ctl00`. It's never a good idea to make assumptions about autogenerated values.

Fortunately, a reference to the master page is accessible through the `Page` class's `Master` property. Therefore, instead of having to use `FindControl("ctl00")` to get a reference of the master page in order to access the `MainContent` ContentPlaceHolder, we can instead use `Page.Master.FindControl("MainContent")`. Update the `SubmitButton_Click` event handler with the following code:

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
    ContentPlaceHolder MainContent = Page.Master.FindControl("MainContent") as ContentPlaceHolder;

    Label ResultsLabel = MainContent.FindControl("Results") as Label;
    TextBox AgeTextBox = MainContent.FindControl("Age") as TextBox;

    ResultsLabel.Text = string.Format("You are {0} years old!", AgeTextBox.Text);
}
```

This time, visiting the page through a browser, entering your age, and clicking the "Submit" button displays the message in the `Results` Label, as expected.

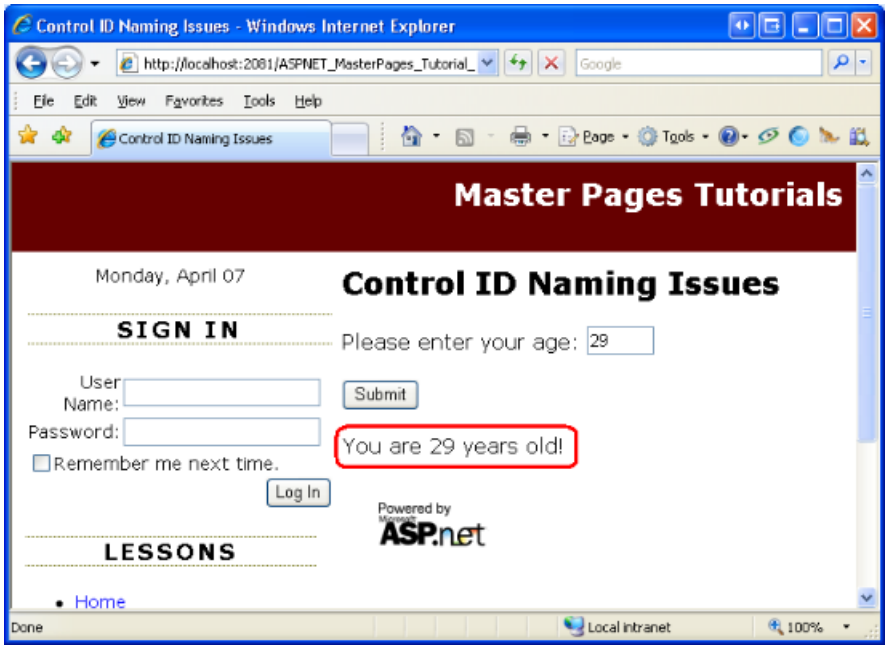


Figure 06: The User's Age is Displayed in the Label (Click to view full-size image)

Recursively Searching Through Naming Containers

The reason the previous code example referenced the `MainContent` ContentPlaceHolder control from the master page, and then the `Results` Label and `Age` TextBox controls from `MainContent`, is because the `Control.FindControl` method only searches within *Control*'s naming container. Having `FindControl` stay within the naming container makes sense in most scenarios because two controls in two different naming containers

may have the same `ID` values. Consider the case of a GridView that defines a Label Web control named `ProductName` within one of its TemplateFields. When the data is bound to the GridView at runtime, a `ProductName` Label is created for each GridView row. If `FindControl` searched through all naming containers and we called `Page.FindControl("ProductName")`, what Label instance should the `FindControl` return? The `ProductName` Label in the first GridView row? The one in the last row?

So having `Control.FindControl` search just *Control*'s naming container makes sense in most cases. But there are other cases, such as the one facing us, where we have a unique `ID` across all naming containers and want to avoid having to meticulously reference each naming container in the control hierarchy to access a control. Having a `FindControl` variant that recursively searches all naming containers makes sense, too. Unfortunately, the .NET Framework does not include such a method.

The good news is that we can create our own `FindControl` method that recursively searches all naming containers. In fact, using *extension methods* we can tack on a `FindControlRecursive` method to the `Control` class to accompany its existing `FindControl` method.

> **NOTE**
>
> Extension methods are a feature new to C# 3.0 and Visual Basic 9, which are the languages that ship with the .NET Framework version 3.5 and Visual Studio 2008. In short, extension methods allow for a developer to create a new method for an existing class type through a special syntax. For more information on this helpful feature, refer to my article, Extending Base Type Functionality with Extension Methods.

To create the extension method, add a new file to the `App_Code` folder named `PageExtensionMethods.cs`. Add an extension method named `FindControlRecursive` that takes as an input a `string` parameter named `controlID`. For extension methods to work properly, it is vital that the class itself and its extension methods be marked `static`. Moreover, all extension methods must accept as their first parameter an object of the type to which the extension method applies, and this input parameter must be preceded with the keyword `this`.

Add the following code to the `PageExtensionMethods.cs` class file to define this class and the `FindControlRecursive` extension method:

```
using System;
using System.Web;
using System.Web.UI;

public static class PageExtensionMethods
{
    public static Control FindControlRecursive(this Control ctrl, string controlID)
    {
        if (string.Compare(ctrl.ID, controlID, true) == 0)
        {
            // We found the control!
            return ctrl;
        }
        else
        {
            // Recurse through ctrl's Controls collections
            foreach (Control child in ctrl.Controls)
            {
                Control lookFor = FindControlRecursive(child, controlID);

                if (lookFor != null)
                    return lookFor;  // We found the control
            }

            // If we reach here, control was not found
            return null;
        }
    }
}
```

With this code in place, return to the `IDIssues.aspx` page's code-behind class and comment out the current `FindControl` method calls. Replace them with calls to `Page.FindControlRecursive("controlID")`. What's neat about extension methods is that they appear directly within the IntelliSense drop-down lists. As Figure 7 shows, when you type Page and then hit period, the `FindControlRecursive` method is included in the IntelliSense drop-down along with the other `Control` class methods.

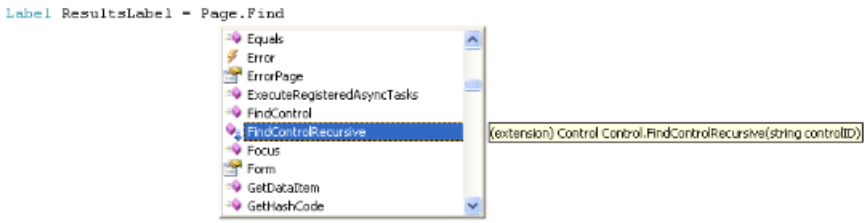


Figure 07: Extension Methods are Included in the IntelliSense Drop-Downs (Click to view full-size image)

Enter the following code into the `SubmitButton_Click` event handler and then test it by visiting the page, entering your age, and clicking the "Submit" button. As shown back in Figure 6, the resulting output will be the message, "You are age years old!"

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
    Label ResultsLabel = Page.FindControlRecursive("Results") as Label;
    TextBox AgeTextBox = Page.FindControlRecursive("Age") as TextBox;

    ResultsLabel.Text = string.Format("You are {0} years old!", AgeTextBox.Text);
}
```

> **NOTE**
> Because extension methods are new to C# 3.0 and Visual Basic 9, if you are using Visual Studio 2005 you cannot use extension methods. Instead, you'll need to implement the `FindControlRecursive` method in a helper class. Rick Strahl has such an example in his blog post, ASP.NET Maser Pages and `FindControl`.

Step 4: Using the Correct `id` Attribute Value in Client-Side Script

As noted in this tutorial's introduction, a Web control's rendered `id` attribute is oftentimes used in client-side script to programmatically reference a particular HTML element. For example, the following JavaScript references an HTML element by its `id` and then displays its value in a modal message box:

```
var elem = document.getElementById("Age");
if (elem != null)
    alert("You entered " + elem.value + " into the Age text box.");
```

Recall that in ASP.NET pages that do not include a naming container, the rendered HTML element's `id` attribute is identical to the Web control's `ID` property value. Because of this, it is tempting to hard code in `id` attribute values into JavaScript code. That is, if you know you want to access the `Age` TextBox Web control through client-side script, do so via a call to `document.getElementById("Age")`.

The problem with this approach is that when using master pages (or other naming container controls), the rendered HTML `id` is not synonymous with the Web control's `ID` property. Your first inclination may be to visit the page through a browser and view the source to determine the actual `id` attribute. Once you know the rendered `id` value, you can paste it into the call to `getElementById` to access the HTML element you need to work with through client-side script. This approach is less than ideal because certain changes to the page's control hierarchy or changes to the `ID` properties of the naming controls will alter the resulting `id` attribute, thereby breaking your JavaScript code.

The good news is that the `id` attribute value that is rendered is accessible in server-side code through the Web control's `ClientID` property. You should use this property to determine the `id` attribute value used in client-side script. For example, to add a JavaScript function to the page that, when called, displays the value of the `Age` TextBox in a modal message box, add the following code to the `Page_Load` event handler:

```
ClientScript.RegisterClientScriptBlock(this.GetType(), "ShowAgeTextBoxScript",
 string.Format(@"function ShowAge()
 {{
 var elem = document.getElementById('{0}');
 if (elem != null)
 alert('You entered ' + elem.value + ' into the Age text box.');
 }}", AgeTextBox.ClientID), true);
```

The above code injects the value of the `Age` TextBox's ClientID property into the JavaScript call to `getElementById`. If you visit this page through a browser and view the HTML source, you'll find the following JavaScript code:

```
<script type="text/javascript">
//<![CDATA[
function ShowAge()
{
 var elem = document.getElementById('ctl00_MainContent_Age');
 if (elem != null)
 alert('You entered ' + elem.value + ' into the Age text box.');
}//]]>
</script>
```

Notice how the correct `id` attribute value, `ctl00_MainContent_Age`, appears within the call to `getElementById`. Because this value is calculated at runtime, it works regardless of later changes to the page control hierarchy.

> **NOTE**
> This JavaScript example merely shows how to add a JavaScript function that correctly references the HTML element rendered by a server control. To use this function you would need to author additional JavaScript to call the function when the document loads or when some specific user action transpires. For more information on these and related topics, read Working with Client-Side Script.

Summary

Certain ASP.NET server controls act as naming containers, which affects the rendered `id` attribute values of their descendent controls as well as the scope of controls canvassed by the `FindControl` method. With regards to master pages, both the master page itself and its ContentPlaceHolder controls are naming containers. Consequently, we need to put forth a bit more work to programmatically reference controls within the content page using `FindControl`. In this tutorial we examined two techniques: drilling into the ContentPlaceHolder control and calling its `FindControl` method; and rolling our own `FindControl` implementation that recursively searches through all naming containers.

In addition to the server-side issues naming containers introduce with regards to referencing Web controls, there are also client-side issues. In the absence of naming containers, the Web control's `ID` property value and rendered `id` attribute value are one in the same. But with the addition of naming container, the rendered `id` attribute includes both the `ID` values of the Web control and the naming container(s) in its control hierarchy's ancestry. These naming concerns are a non-issue as long as you use the Web control's `ClientID` property to determine the rendered `id` attribute value in your client-side script.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Master Pages and `FindControl`
- Creating Dynamic Data Entry User Interfaces
- Extending Base Type Functionality with Extension Methods
- How To: Reference ASP.NET Master Page Content
- Mater Pages: Tips, Tricks, and Traps
- Working with Client-Side Script

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Zack Jones and Suchi Barnerjee. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Interacting with the Master Page from the Content Page (C#)

2/20/2020 • 19 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> Examines how to call methods, set properties, etc. of the Master Page from code in the Content Page.

Introduction

Over the course of the past five tutorials we have looked at how to create a master page, define content regions, bind ASP.NET pages to a master page, and define page-specific content. When a visitor requests a particular content page, the content and master pages' markup are fused at runtime, resulting in the rendering of a unified control hierarchy. Therefore, we have already seen one way in which the master page and one of its content pages can interact: the content page spells out the markup to transfuse into the master page's ContentPlaceHolder controls.

What we have yet to examine is how the master page and content page can interact programmatically. In addition to defining the markup for the master page's ContentPlaceHolder controls, a content page can also assign values to its master page's public properties and invoke its public methods. Similarly, a master page may interact with its content pages. While programmatic interaction between a master and content page is less common than the interaction between their declarative markups, there are many scenarios where such programmatic interaction is needed.

In this tutorial we examine how a content page can programmatically interact with its master page; in the next tutorial we will look at how the master page can similarly interact with its content pages.

Examples of Programmatic Interaction Between a Content Page and its Master Page

When a particular region of a page needs to be configured on a page-by-page basis, we use a ContentPlaceHolder control. But what about situations where the majority of pages need to emit a certain output, but a small number of pages need to customize it to show something else? One such example, which we examined in the *Multiple ContentPlaceHolders and Default Content* tutorial, involves displaying a login interface on each page. While most pages should include a login interface, it should be suppressed for a handful of pages, such as: the main login page (`Login.aspx`); the Create Account page; and other pages that are only accessible to authenticated users. The *Multiple ContentPlaceHolders and Default Content* tutorial showed how to define the default content for a ContentPlaceHolder in the master page and then how to override it in those pages where the default content was not wanted.

Another option is to create a public property or method within the master page that indicates whether to show or hide the login interface. For example, the master page might include a public property named `ShowLoginUI` whose value was used to set the `Visible` property of the Login control in the master page. Those content pages where the login user interface should be suppressed could then programmatically set the `ShowLoginUI` property to `false`.

Perhaps the most common example of content and master page interaction occurs when data displayed in the master page needs to be refreshed after some action has transpired in the content page. Consider a master page

that includes a GridView that displays the five most recently added records from a particular database table, and that one of its content pages includes an interface for adding new records to that same table.

When a user visits the page to add a new record, she sees the five most recently added records displayed in the master page. After filling in the values for the new record's columns, she submits the form. Assuming that the GridView in the master page has its `EnableViewState` property set to true (the default), its content is reloaded from view state and, consequently, the five same records are displayed even though a newer record was just added to the database. This may confuse the user.

> **NOTE**
> Even if you disable the GridView's view state so that it rebinds to its underlying data source on every postback, it still won't show the just-added record because the data is bound to the GridView earlier in the page lifecycle than when the new record is added to the database.

To remedy this so that the just-added record is displayed in the master page's GridView on postback we need to instruct the GridView to rebind to its data source *after* the new record has been added to the database. This requires interaction between the content and master pages because the interface for adding the new record (and its event handlers) are in the content page but the GridView that needs to be refreshed is in the master page.

Because refreshing the master page's display from an event handler in the content page is one of the most common needs for content and master page interaction, let's explore this topic in more detail. The download for this tutorial includes a Microsoft SQL Server 2005 Express Edition database named `NORTHWIND.MDF` in the website's `App_Data` folder. The Northwind database stores product, employee, and sales information for a fictitious company, Northwind Traders.

Step 1 walks through displaying the five most recently added products in a GridView in the master page. Step 2 creates a content page for adding new products. Step 3 looks at how to create public properties and methods in the master page, and Step 4 illustrates how to programmatically interface with these properties and methods from the content page.

> **NOTE**
> This tutorial does not delve into the specifics of working with data in ASP.NET. The steps for setting up the master page to display data and the content page for inserting data are complete, yet breezy. For a more in-depth look at displaying and inserting data and using the SqlDataSource and GridView controls, consult the resources in the Further Readings section at the end of this tutorial.

Step 1: Displaying the Five Most Recently Added Products in the Master Page

Open the `Site.master` master page and add a Label and a GridView control to the `leftContent` `<div>`. Clear out the Label's `Text` property, set its `EnableViewState` property to false, and its `ID` property to `GridMessage`; set the GridView's `ID` property to `RecentProducts`. Next, from the Designer, expand the GridView's smart tag and choose to bind it to a new data source. This launches the Data Source Configuration wizard. Because the Northwind database in the `App_Data` folder is a Microsoft SQL Server database, choose to create a SqlDataSource by selecting (see Figure 1); name the SqlDataSource `RecentProductsDataSource`.

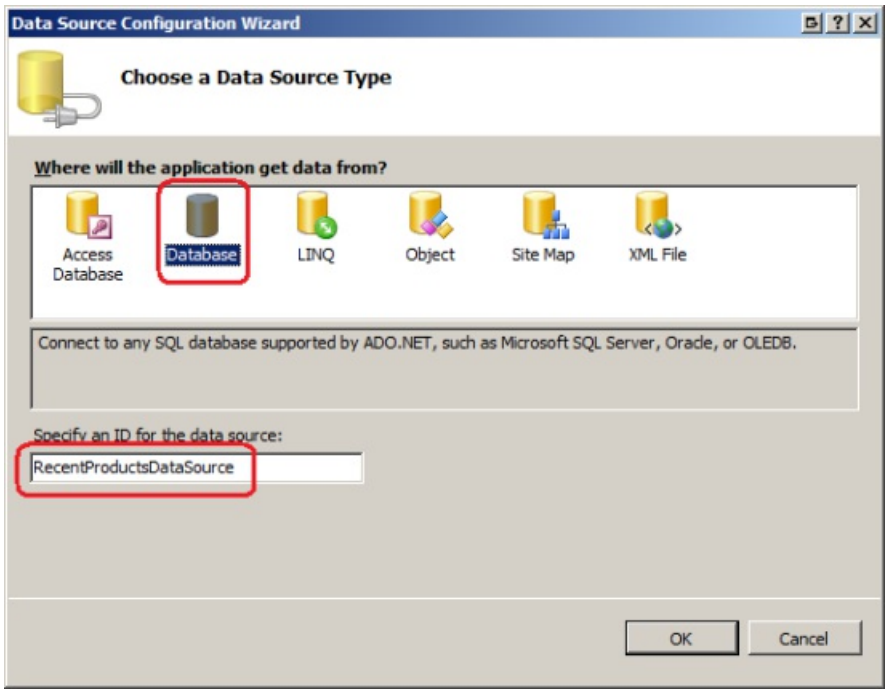


Figure 01: Bind the GridView to a SqlDataSource Control Named `RecentProductsDataSource` (Click to view full-size image)

The next step asks us to specify what database to connect to. Choose the `NORTHWIND.MDF` database file from the drop-down list and click Next. Because this is the first time we've used this database, the wizard will offer to store the connection string in `Web.config`. Have it store the connection string using the name `NorthwindConnectionString`.

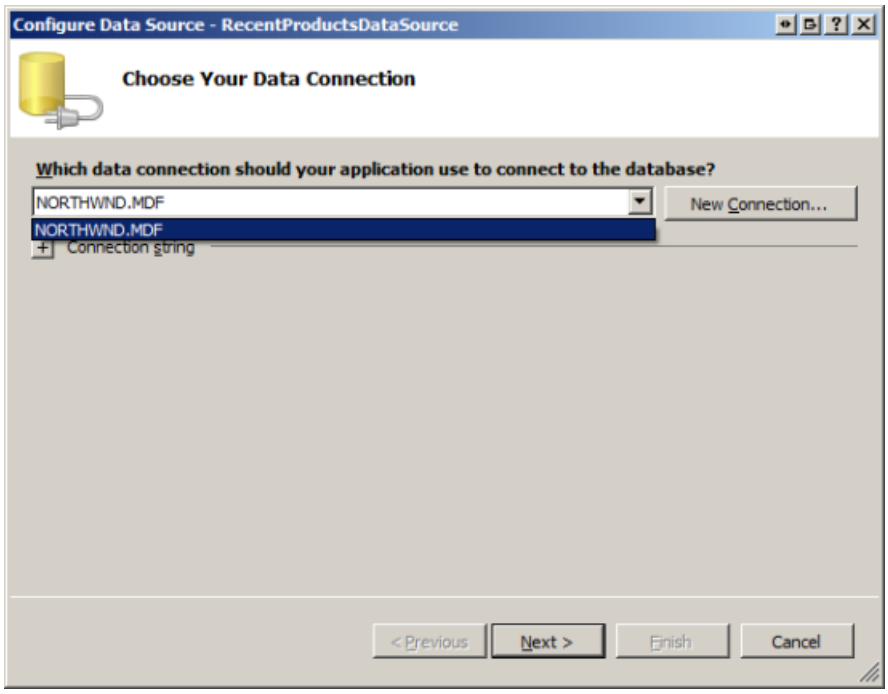


Figure 02: Connect to the Northwind Database (Click to view full-size image)

The Configure Data Source wizard provides two means by which we can specify the query used to retrieve data:

- By specifying a custom SQL statement or stored procedure, or
- By picking a table or view and then specifying the columns to return

Because we want to return just the five most recently added products, we need to specify a custom SQL statement. Use the following SELECT query:

```
SELECT TOP 5 ProductName, UnitPrice FROM Products ORDER BY ProductID DESC
```

The `TOP 5` keyword returns only the first five records from the query. The `Products` table's primary key, `ProductID`, is an `IDENTITY` column, which assures us that each new product added to the table will have a larger value than the previous entry. Therefore, sorting the results by `ProductID` in descending order returns the products starting with the most recently created ones.

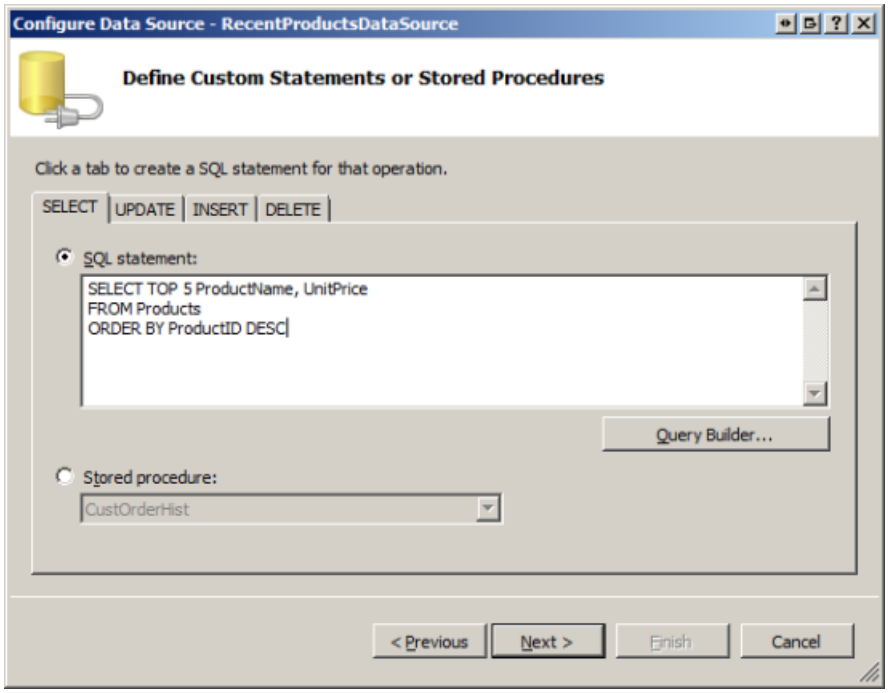


Figure 03: Return the Five Most Recently Added Products (Click to view full-size image)

After completing the wizard, Visual Studio generates two BoundFields for the GridView to display the `ProductName` and `UnitPrice` fields returned from the database. At this point your master page's declarative markup should include markup similar to the following:

```
<asp:Label ID="GridMessage" runat="server" EnableViewState="false"></asp:Label>
<asp:GridView ID="RecentProducts" runat="server" AutoGenerateColumns="False"
 DataSourceID="RecentProductsDataSource">
 <Columns>
 <asp:BoundField DataField="ProductName" HeaderText="ProductName"
 SortExpression="ProductName"/>
 <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
 SortExpression="UnitPrice"/>
 </Columns>
</asp:GridView>

<asp:SqlDataSource ID="RecentProductsDataSource" runat="server"
 ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
 SelectCommand="SELECT TOP 5 ProductName, UnitPrice FROM Products ORDER BY ProductID DESC">
</asp:SqlDataSource>
```

As you can see, the markup contains: the Label Web control (`GridMessage`); the GridView `RecentProducts`, with two BoundFields; and a SqlDataSource control that returns the five most recently added products.

With this GridView created and its SqlDataSource control configured, visit the website through a browser. As Figure 4 shows, you will see a grid in the lower left corner that lists the five most recently added products.

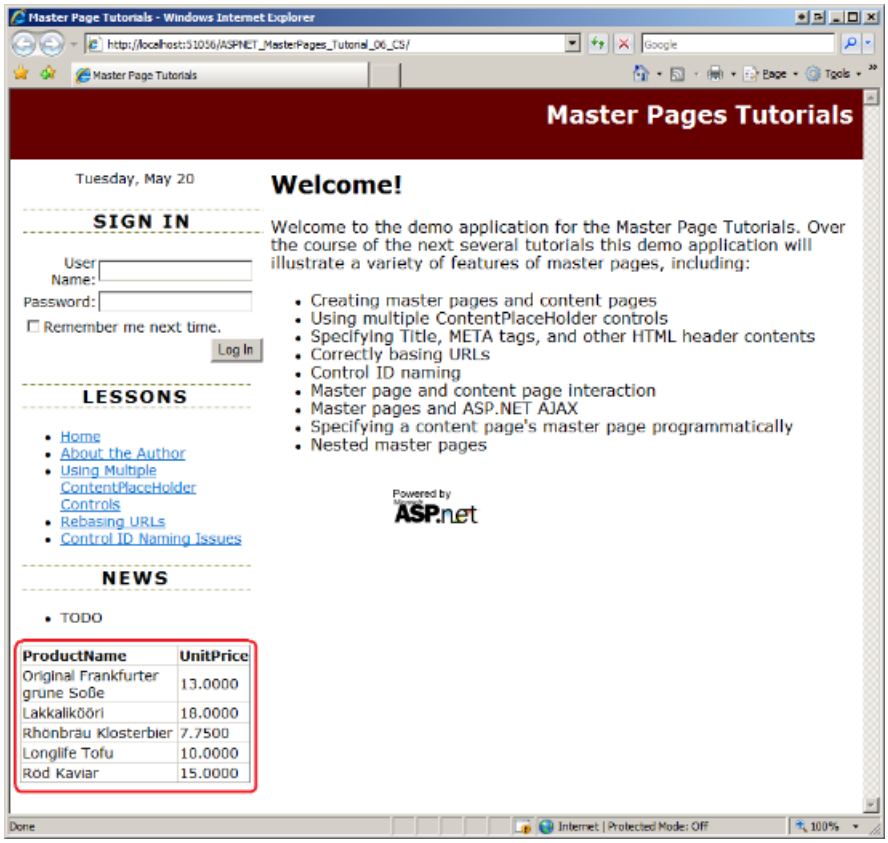


Figure 04: The GridView Displays the Five Most Recently Added Products (Click to view full-size image)

> **NOTE**
>
> Feel free to clean up the appearance of the GridView. Some suggestions include formatting the displayed `UnitPrice` value as a currency and using background colors and fonts to improve the grid's appearance.

Step 2: Creating a Content Page to Add New Products

Our next task is to create a content page from which a user can add a new product to the `Products` table. Add a new content page to the `Admin` folder named `AddProduct.aspx`, making sure to bind it to the `Site.master` master page. Figure 5 shows the Solution Explorer after this page has been added to the website.

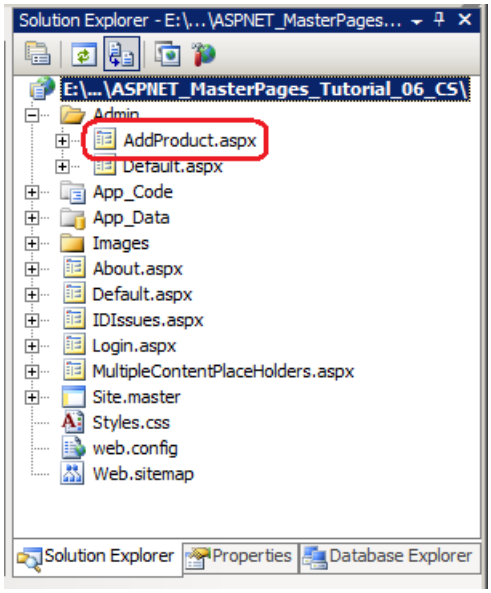


Figure 05: Add a New ASP.NET Page to the `Admin` Folder(Click to view full-size image)

Recall that in the *Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page* tutorial we created a custom base page class named `BasePage` that generated the page's title if it was not explicitly set. Go to the `AddProduct.aspx` page's code-behind class and have it derive from `BasePage` (instead of from `System.Web.UI.Page`).

Finally, update the `Web.sitemap` file to include an entry for this lesson. Add the following markup beneath the `<siteMapNode>` for the Control ID Naming Issues lesson:

```
<siteMapNode url="~/Admin/AddProduct.aspx" title="Content to Master Page Interaction" />
```

As shown in Figure 6, the addition of this `<siteMapNode>` element is reflected in the Lessons list.

Return to `AddProduct.aspx`. In the Content control for the `MainContent` ContentPlaceHolder, add a DetailsView control and name it `NewProduct`. Bind the DetailsView to a new SqlDataSource control named `NewProductDataSource`. Like with the SqlDataSource in Step 1, configure the wizard so that it uses the Northwind database and choose to specify a custom SQL statement. Because the DetailsView will be used to add items to the database, we need to specify both a `SELECT` statement and an `INSERT` statement. Use the following `SELECT` query:

```
SELECT ProductName, UnitPrice FROM Products
```

Then, from the INSERT tab, add the following `INSERT` statement:

```
INSERT INTO Products(ProductName, UnitPrice) VALUES(@ProductName, @UnitPrice)
```

After completing the wizard go to the DetailsView's smart tag and check the "Enable Inserting" checkbox. This adds a CommandField to the DetailsView with its `ShowInsertButton` property set to true. Because this DetailsView will be used solely for inserting data, set the DetailsView's `DefaultMode` property to `Insert`.

That's all there is to it! Let's test this page. Visit `AddProduct.aspx` through a browser, enter a name and price (see Figure 6).

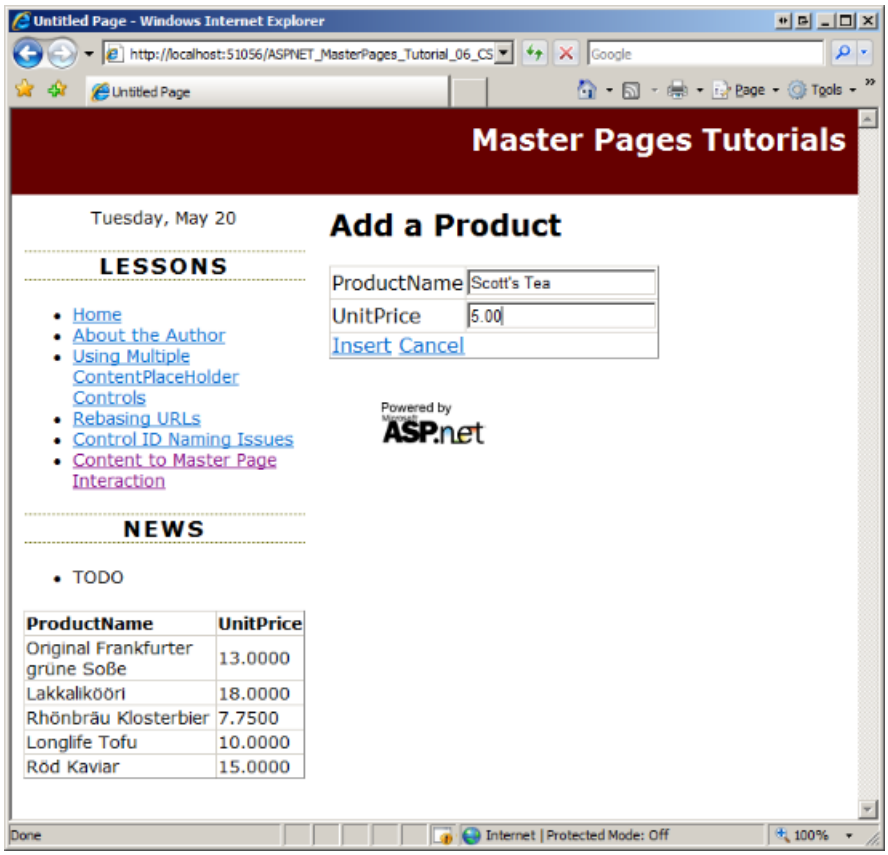


Figure 06: Add a New Product to the Database (Click to view full-size image)

After typing in the name and price for your new product, click the Insert button. This causes the form to postback. On postback, the SqlDataSource control's `INSERT` statement is executed; its two parameters are populated with the user-entered values in the DetailsView's two TextBox controls. Unfortunately, there is no visual feedback that an insert has occurred. It would be nice to have a message displayed, confirming that a new record has been added. I leave this as an exercise for the reader. Also, after adding a new record from the DetailsView the GridView in the master page still shows the same five records as before; it does not include the just-added record. We'll examine how to remedy this in the upcoming steps.

> **NOTE**
>
> In addition to adding some form of visual feedback that the insert has succeeded, I'd encourage you to also update the DetailsView's inserting interface to include validation. Currently, there is no validation. If a user enters an invalid value for the `UnitPrice` field, such as "Too expensive," an exception will be thrown on postback when the system attempts to convert that string into a decimal. For more information on customizing the inserting interface, refer to the *Customizing the Data Modification Interface* tutorial from my Working with Data tutorial series.

Step 3: Creating Public Properties and Methods in the Master Page

In Step 1 we added a Label Web control named `GridMessage` above the GridView in the master page. This Label is intended to optionally display a message. For example, after adding a new record to the `Products` table, we might want to show a message that reads: "*ProductName* has been added to the database." Rather than hard-code the text for this Label in the master page, we might want the message to be customizable by the content page.

Because the Label control is implemented as a protected member variable within the master page it cannot be accessed directly from content pages. In order to work with the Label within a master page from the content page (or, for that matter, any Web control in the master page) we need to create a public property in the master page that exposes the Web control or serves as a proxy by which one of its properties can be accessed. Add the following syntax to the master page's code-behind class to expose the Label's `Text` property:

```
public string GridMessageText
{
    get
    {
        return GridMessage.Text;
    }
    set
    {
        GridMessage.Text = value;
    }
}
```

When a new record is added to the `Products` table from a content page the `RecentProducts` GridView in the master page needs to rebind to its underlying data source. To rebind the GridView call its `DataBind` method. Because the GridView in the master page is not programmatically accessible to the content pages, we need to create a public method in the master page that, when called, rebinds the data to the GridView. Add the following method to the master page's code-behind class:

```
public void RefreshRecentProductsGrid()
{
    RecentProducts.DataBind();
}
```

With the `GridMessageText` property and `RefreshRecentProductsGrid` method in place, any content page can programmatically set or read the value of the `GridMessage` Label's `Text` property or rebind the data to the `RecentProducts` GridView. Step 4 examines how to access the master page's public properties and methods from a content page.

> **NOTE**
> Don't forget to mark the master page's properties and methods as `public`. If you do not explicitly denote these properties and methods as `public`, they will not be accessible from the content page.

Step 4: Calling the Master Page's Public Members from a Content Page

Now that the master page has the necessary public properties and methods, we're ready to invoke these properties and methods from the `AddProduct.aspx` content page. Specifically, we need to set the master page's `GridMessageText` property and call its `RefreshRecentProductsGrid` method after the new product has been added to the database. All the ASP.NET data Web controls fire events immediately before and after completing various tasks, which make it easy for page developers to take some programmatic action either before or after the task. For example, when the end user clicks the DetailsView's Insert button, on postback the DetailsView raises its `ItemInserting` event before beginning the inserting workflow. It then inserts the record into the database. Following that, the DetailsView raises its `ItemInserted` event. Therefore, in order to work with the master page after the new product has been added, create an event handler for the DetailsView's `ItemInserted` event.

There are two ways that a content page can programmatically interface with its master page:

- Using the `Page.Master` property, which returns a loosely-typed reference to the master page, or
- Specify the page's master page type or file path via a `@MasterType` directive; this automatically adds a strongly-typed property to the page named `Master`.

Let's examine both approaches.

Using the Loosely-Typed `Page.Master` Property

All ASP.NET web pages must derive from the `Page` class, which is located in the `System.Web.UI` namespace. The `Page` class includes a `Master` property that returns a reference to the page's master page. If the page does not have a master page `Master` returns `null`.

The `Master` property returns an object of type `MasterPage` (also located in the `System.Web.UI` namespace) which is the base type from which all master pages derive from. Therefore, to use public properties or methods defined in our website's master page we must cast the `MasterPage` object returned from the `Master` property to the appropriate type. Because we named our master page file `Site.master`, the code-behind class was named `Site`. Therefore, the following code casts the `Page.Master` property to an instance of the Site class.

```
// Cast the loosely-typed Page.Master property and then set the GridMessageText property
Site myMasterPage = Page.Master as Site;
```

Now that we have casted the loosely-typed `Page.Master` property to the `Site` type we can reference the properties and methods specific to Site. As Figure 7 shows, the public property `GridMessageText` appears in the IntelliSense drop-down.

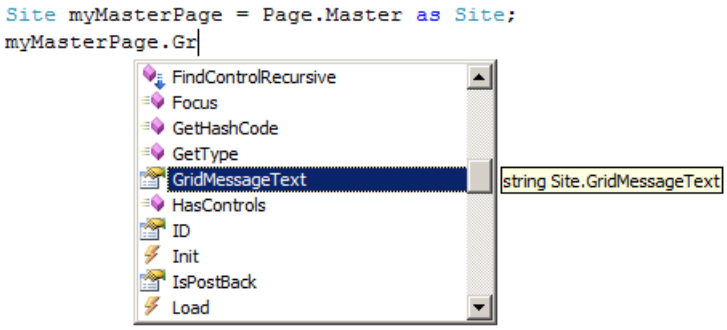


Figure 07: IntelliSense Shows our Master Page's Public Properties and Methods (Click to view full-size image)

> **NOTE**
> If you named your master page file `MasterPage.master` then the master page's code-behind class name is `MasterPage`. This can lead to ambiguous code when casting from the type `System.Web.UI.MasterPage` to your `MasterPage` class. In short, you need to fully qualify the type you are casting to, which can be a little tricky when using the Web Site Project model. My suggestion would be to either make sure that when you create your master page you name it something other than `MasterPage.master` or, even better, create a strongly-typed reference to the master page.

Creating a Strongly-Typed Reference with the `@MasterType` Directive

If you look closely you can see that an ASP.NET page's code-behind class is a partial class (note the `partial` keyword in the class definition). Partial classes were introduced in C# and Visual Basic with.NET Framework 2.0 and, in a nutshell, allow for a class's members to be defined across multiple files. The code-behind class file - `AddProduct.aspx.cs`, for example - contains the code that we, the page developer, create. In addition to our code, the ASP.NET engine automatically creates a separate class file with properties and event handlers in that translate the declarative markup into the page's class hierarchy.

The automatic code generation that occurs whenever an ASP.NET page is visited paves the way for some rather interesting and useful possibilities. In the case of master pages, if we tell the ASP.NET engine what master page is being used by our content page it generates a strongly-typed `Master` property for us.

Use the `@MasterType` directive to inform the ASP.NET engine of the content page's master page type. The `@MasterType` directive can accept either the type name of the master page or its file path. To specify that the `AddProduct.aspx` page uses `Site.master` as its master page, add the following directive to the top of `AddProduct.aspx`:

```
<%@ MasterType VirtualPath="~/Site.master" %>
```

This directive instructs the ASP.NET engine to add a strongly-typed reference to the master page through a property named `Master`. With the `@MasterType` directive in place, we can call the `Site.master` master page's public properties and methods directly through the `Master` property without any casts.

> **NOTE**
> If you omit the `@MasterType` directive, the syntax `Page.Master` and `Master` return the same thing: a loosely-typed object to the page's master page. If you include the `@MasterType` directive then `Master` returns a strongly-typed reference to the specified master page. `Page.Master`, however, still returns a loosely-typed reference. For a more thorough look at why this is the case and how the `Master` property is constructed when the `@MasterType` directive is included, see K. Scott Allen's blog entry `@MasterType` in ASP.NET 2.0.

Updating the Master Page After Adding a New Product

Now that we know how to invoke a master page's public properties and methods from a content page, we're ready to update the `AddProduct.aspx` page so that the master page is refreshed after adding a new product. At the beginning of Step 4 we created an event handler for the DetailsView control's `ItemInserting` event, which executes immediately after the new product has been added to the database. Add the following code to that event handler:

```
protected void NewProduct_ItemInserted(object sender, DetailsViewInsertedEventArgs e)
{
    // Cast the loosely-typed Page.Master property and then set the GridMessageText property
    Site myMasterPage = Page.Master as Site;
    myMasterPage.GridMessageText = string.Format("{0} added to grid...", e.Values["ProductName"]);
    // Use the strongly-typed Master property
    Master.RefreshRecentProductsGrid();
}
```

The above code uses both the loosely-typed `Page.Master` property and the strongly-typed `Master` property. Note that the `GridMessageText` property is set to "*ProductName* added to grid..." The just-added product's values are accessible through the `e.Values` collection; as you can see, the just-added `ProductName` value is accessed via `e.Values["ProductName"]`.

Figure 8 shows the `AddProduct.aspx` page immediately after a new product - Scott's Soda - has been added to the database. Note that the just-added product name is noted in the master page's Label and that the GridView has been refreshed to include the product and its price.

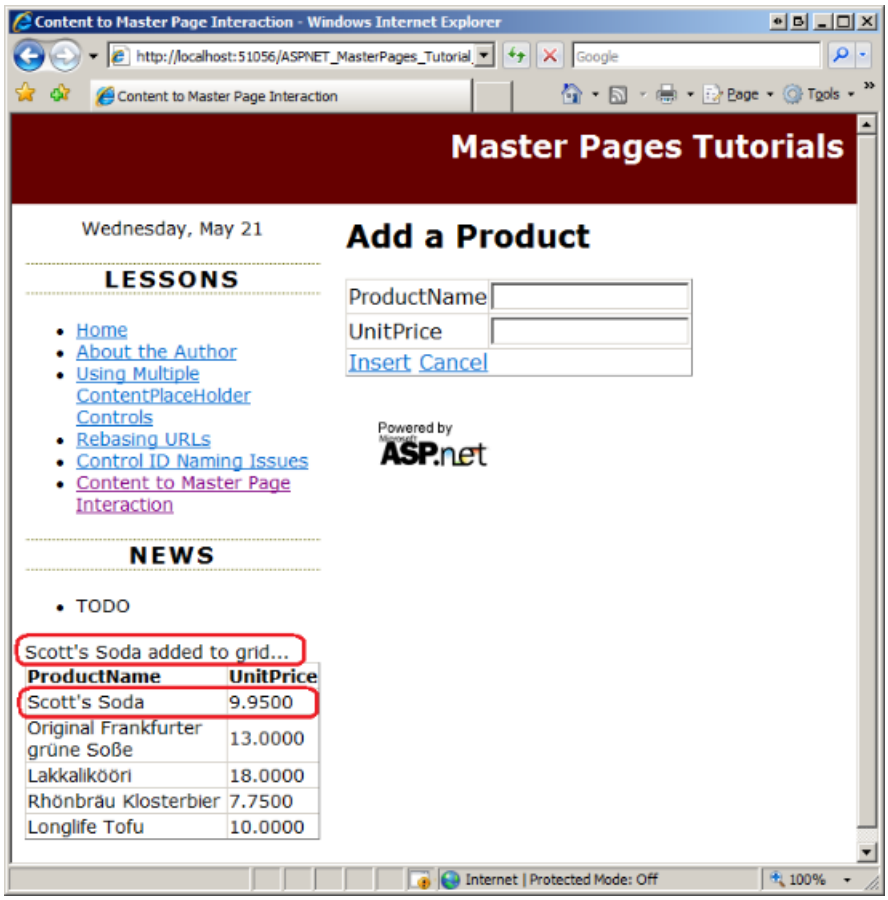


Figure 08: The Master Page's Label and GridView Show the Just-Added Product (Click to view full-size image)

Summary

Ideally, a master page and its content pages are completely separate from one another and require no level of interaction. While master pages and content pages should be designed with that goal in mind, there are a number of common scenarios in which a content page must interface with its master page. One of the most common reasons centers around updating a particular portion of the master page display based on some action that transpired in the content page.

The good news is that it's relatively straightforward to have a content page programmatically interact with its master page. Start by creating public properties or methods in the master page that encapsulate the functionality that needs to be invoked by a content page. Then, in the content page, access the master page's properties and methods through the loosely-typed `Page.Master` property or use the `@MasterType` directive to create a strongly-typed reference to the master page.

In the next tutorial we examine how to have the master page programmatically interact with one of its content pages.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Accessing and Updating Data in ASP.NET
- ASP.NET Master Pages: Tips, Tricks, and Traps
- `@MasterType` in ASP.NET 2.0
- Passing Information Between Content and Master Pages
- Working with Data in ASP.NET Tutorials

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Zack Jones. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Interacting with the Content Page from the Master Page (C#)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> Examines how to call methods, set properties, etc. of the Content Page from code in the Master Page.

Introduction

The preceding tutorial examined how to have the content page programmatically interact with its master page. Recall that we updated the master page to include a GridView control that listed the five most recently added products. We then created a content page from which the user could add a new product. Upon adding a new product, the content page needed to instruct the master page to refresh its GridView so that it would include the just-added product. This functionality was accomplished by adding a public method to the master page that refreshed the data bound to the GridView, and then invoking that method from the content page.

The most common form of content and master page interaction originates from the content page. However, it is possible for the master page to rouse the current content page into action, and such functionality may be needed if the master page contains user interface elements that enable users to modify data that is also displayed on the content page. Consider a content page that displays the products information in a GridView control and a master page that includes a Button control that, when clicked, doubles the prices of all products. Much like the example in the preceding tutorial, the GridView needs to be refreshed after the double price Button is clicked so that it displays the new prices, but in this scenario it's the master page that needs to rouse the content page into action.

This tutorial explores how to have the master page invoke functionality defined in the content page.

Instigating Programmatic Interaction via an Event and Event Handlers

Invoking content page functionality from a master page is more challenging than the other way around. Because a content page has a single master page, when instigating the programmatic interaction from the content page we know what public methods and properties are at our disposal. A master page, however, can have many different content pages, each with its own set of properties and methods. How, then, can we write code in the master page to perform some action in its content page when we don't know what content page will be invoked until runtime?

Consider an ASP.NET Web control, such as the Button control. A Button control can appear on any number of ASP.NET pages and needs a mechanism by which it can alert the page that it has been clicked. This is accomplished using *events*. In particular, the Button control raises its `Click` event when it is clicked; the ASP.NET page that contains the Button can optionally respond to that notification via an *event handler*.

This same pattern can be used to have a master page trigger functionality in its content pages:

1. Add an event to the master page.
2. Raise the event whenever the master page needs to communicate with its content page. For example, if the master page needs to alert its content page that the user has doubled the prices, its event would be raised immediately after the prices have been doubled.
3. Create an event handler in those content pages that need to take some action.

This remainder of this tutorial implements the example outlined in the Introduction; namely, a content page that

lists the products in the database and a master page that includes a Button control to double the prices.

Step 1: Displaying Products in a Content Page

Our first order of business is to create a content page that lists the products from the Northwind database. (We added the Northwind database to the project in the preceding tutorial, *Interacting with the Master Page from the Content Page*.) Start by adding a new ASP.NET page to the `~/Admin` folder named `Products.aspx`, making sure to bind it to the `Site.master` master page. Figure 1 shows the Solution Explorer after this page has been added to the website.

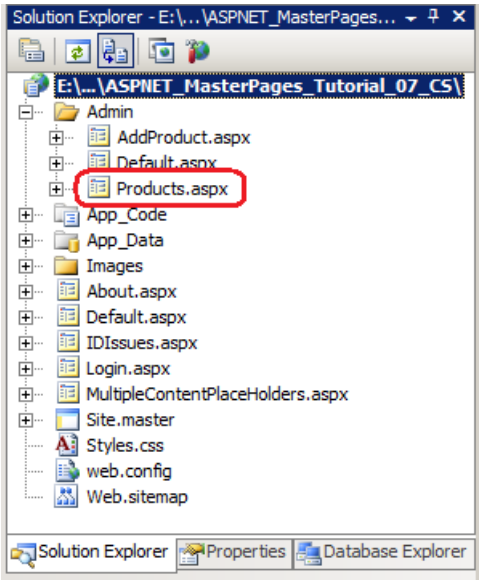


Figure 01: Add a New ASP.NET Page to the `Admin` Folder (Click to view full-size image)

Recall that in the *Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page* tutorial we created a custom base page class named `BasePage` that generates the page's title if it is not explicitly set. Go to the `Products.aspx` page's code-behind class and have it derive from `BasePage` (instead of from `System.Web.UI.Page`).

Finally, update the `Web.sitemap` file to include an entry for this lesson. Add the following markup beneath the `<siteMapNode>` for the Content to Master Page Interaction lesson:

```
<siteMapNode url="~/Admin/Products.aspx" title="Master to Content Page Interaction" />
```

The addition of this `<siteMapNode>` element is reflected in the Lessons list (see Figure 5).

Return to `Products.aspx`. In the Content control for `MainContent`, add a GridView control and name it `ProductsGrid`. Bind the GridView to a new SqlDataSource control named `ProductsDataSource`.

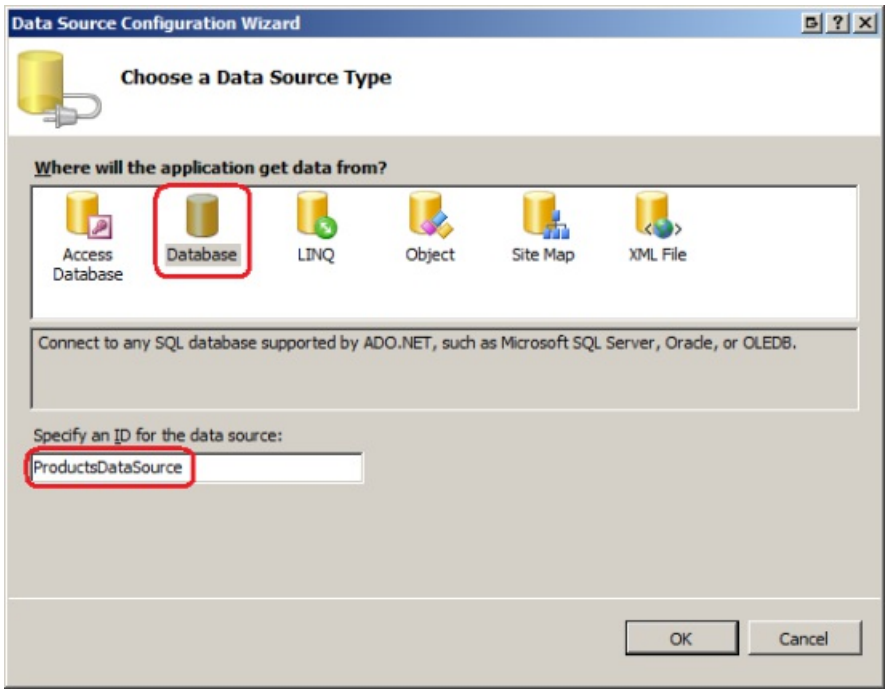


Figure 02: Bind the GridView to a New SqlDataSource Control (Click to view full-size image)

Configure the wizard so that it uses the Northwind database. If you worked through the previous tutorial then you should already have a connection string named `NorthwindConnectionString` in `Web.config`. Choose this connection string from the drop-down list, as shown in Figure 3.

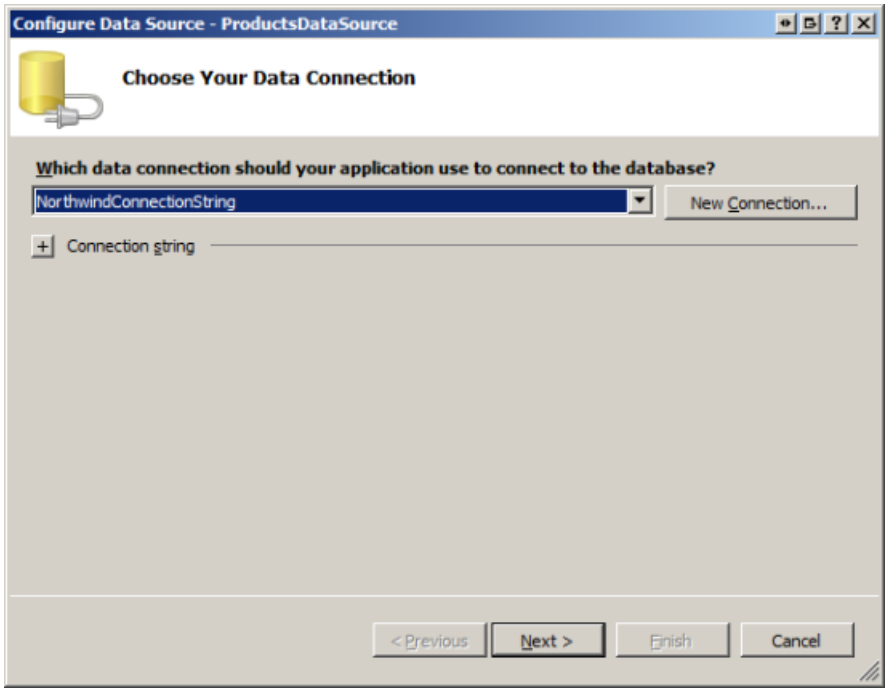


Figure 03: Configure the SqlDataSource to Use the Northwind Database (Click to view full-size image)

Next, specify the data source control's `SELECT` statement by choosing the Products table from the drop-down list and returning the `ProductName` and `UnitPrice` columns (see Figure 4). Click Next and then Finish to complete the Configure Data Source wizard.

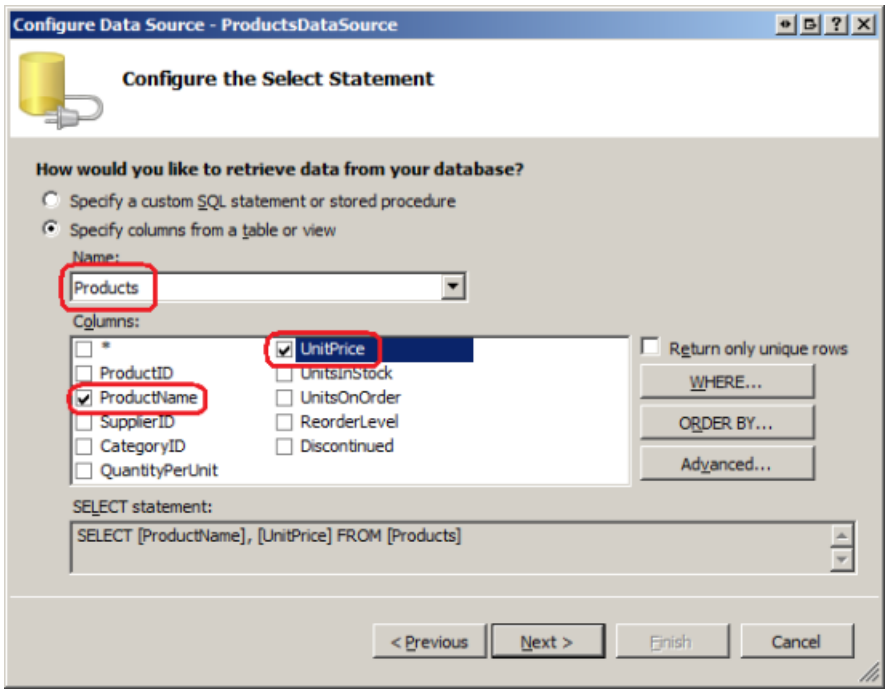


Figure 04: Return the `ProductName` and `UnitPrice` Fields from the `Products` Table (Click to view full-size image)

That's all there is to it! After completing the wizard Visual Studio adds two BoundFields to the GridView to mirror the two fields returned by the SqlDataSource control. The GridView and SqlDataSource controls' markup follows. Figure 5 shows the results when viewed through a browser.

```
<asp:GridView ID="ProductsGrid" runat="server" AutoGenerateColumns="False"
 DataSourceID="ProductsDataSource">
 <Columns>
 <asp:BoundField DataField="ProductName" HeaderText="ProductName"
 SortExpression="ProductName" />
 <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
 SortExpression="UnitPrice" />
 </Columns>
</asp:GridView>

<asp:SqlDataSource ID="ProductsDataSource" runat="server"
 ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
 SelectCommand="SELECT [ProductName], [UnitPrice] FROM [Products]">
</asp:SqlDataSource>
```

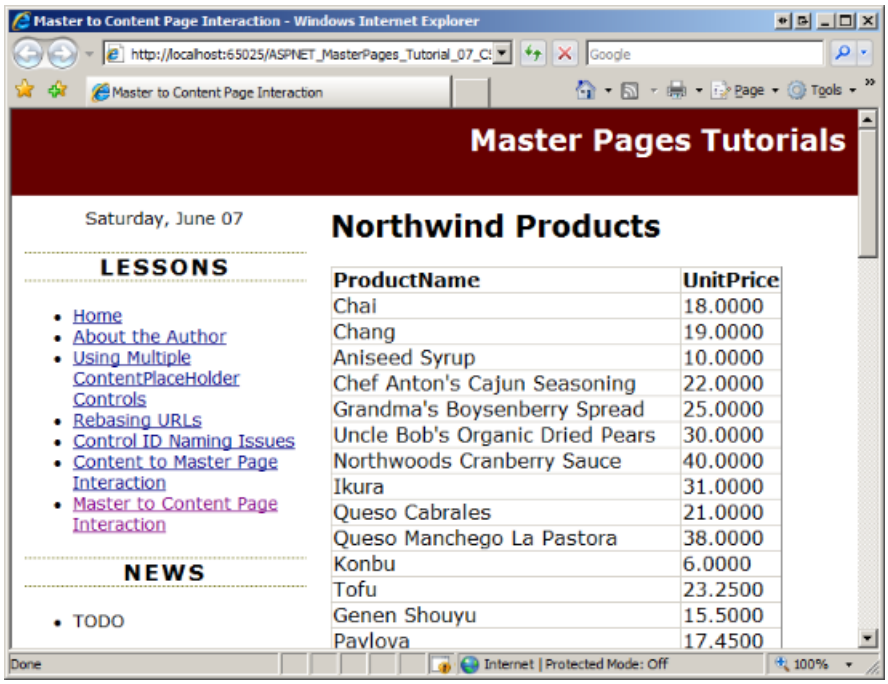


Figure 05: Each Product and its Price is Listed in the GridView (Click to view full-size image)

> **NOTE**
>
> Feel free to clean up the appearance of the GridView. Some suggestions include formatting the displayed UnitPrice value as a currency and using background colors and fonts to improve the grid's appearance. For more information on displaying and formatting data in ASP.NET, refer to my Working with Data tutorial series.

Step 2: Adding a Double Prices Button to the Master Page

Our next task is to add a Button Web control to the master page that, when clicked, will double the price of all products in the database. Open the `Site.master` master page and drag a Button from the Toolbox onto the Designer, placing it beneath the `RecentProductsDataSource` SqlDataSource control we added in the previous tutorial. Set the Button's `ID` property to `DoublePrice` and its `Text` property to "Double Product Prices".

Next, add a SqlDataSource control to the master page, naming it `DoublePricesDataSource`. This SqlDataSource will be used to execute the `UPDATE` statement to double all prices. Specifically, we need to set its `ConnectionString` and `UpdateCommand` properties to the appropriate connection string and `UPDATE` statement. Then we need to call this SqlDataSource control's `Update` method when the `DoublePrice` Button is clicked. To set the `ConnectionString` and `UpdateCommand` properties, select the SqlDataSource control and then go to the Properties window. The `ConnectionString` property lists those connection strings already stored in `Web.config` in a drop-down list; choose the `NorthwindConnectionString` option as shown in Figure 6.

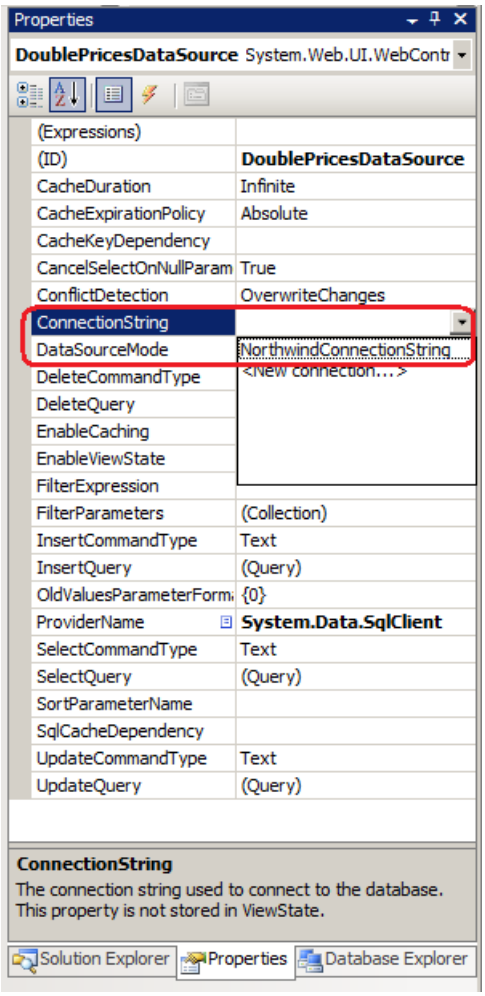


Figure 06: Configure the SqlDataSource to Use the `NorthwindConnectionString` (Click to view full-size image)

To set the `UpdateCommand` property, locate the UpdateQuery option in the Properties window. This property, when selected, displays a button with ellipses; click this button to display the Command and Parameter Editor dialog box shown in Figure 7. Type the following `UPDATE` statement into the dialog box's textbox:

```
UPDATE Products SET UnitPrice = UnitPrice * 2
```

This statement, when executed, will double the `UnitPrice` value for each record in the `Products` table.

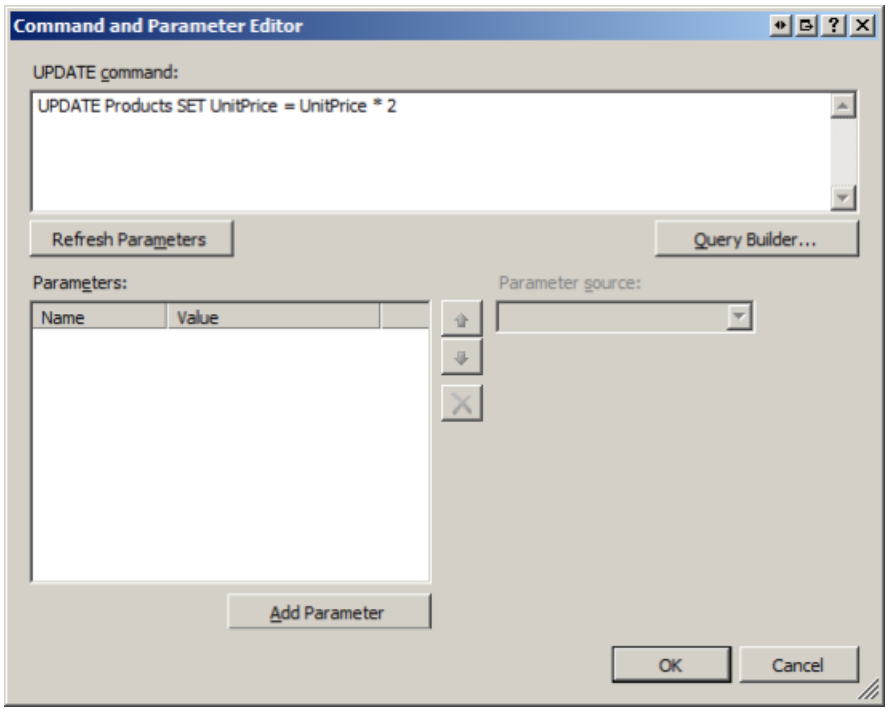


Figure 07: Set SqlDataSource's `UpdateCommand` Property (Click to view full-size image)

After setting these properties, your Button and SqlDataSource controls' declarative markup should look similar to the following:

```
<asp:Button ID="DoublePrice" runat="server"
 Text="Double Product Prices" />

<asp:SqlDataSource ID="DoublePricesDataSource" runat="server"
 UpdateCommand="UPDATE Products SET UnitPrice = UnitPrice * 2"
 ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
 ProviderName="<%$ ConnectionStrings:NorthwindConnectionString.ProviderName %>">
</asp:SqlDataSource>
```

All that remains is to call its `Update` method when the `DoublePrice` Button is clicked. Create a `Click` event handler for the `DoublePrice` Button and add the following code:

```
protected void DoublePrice_Click(object sender, EventArgs e)
{
    // Double the prices
    DoublePricesDataSource.Update();
}
```

To test this functionality, visit the `~/Admin/Products.aspx` page we created in Step 1 and click the "Double Product Prices" button. Clicking the button causes a postback and executes the `DoublePrice` Button's `Click` event handler, doubling the prices of all products. The page is then re-rendered and the markup is returned and re-displayed in the browser. The GridView in the content page, however, lists the same prices as before the "Double Product Prices" button was clicked. This is because the data initially loaded in the GridView had its state stored in view state, so it's not reloaded on postbacks unless instructed otherwise. If you visit a different page and then return to the `~/Admin/Products.aspx` page you'll see the updated prices.

Step 3: Raising an Event When the Prices are Doubled

Because the GridView in the `~/Admin/Products.aspx` page does not immediately reflect the price doubling, a user may understandably think that they did not click the "Double Product Prices" button, or that it didn't work. They may try clicking the button a few more times, doubling the prices again and again. To fix this we need to have the

grid in the content page display the new prices immediately after they are doubled.

As discussed earlier in this tutorial, we need to raise an event in the master page whenever the user clicks the `DoublePrice` Button. An event is a way for one class (an event publisher) to notify another a set of other classes (the event subscribers) that something interesting has occurred. In this example, the master page is the event publisher; those content pages that care about when the `DoublePrice` Button is clicked are the subscribers.

A class subscribes to an event by creating an *event handler*, which is a method that is executed in response to the event being raised. The publisher defines the events he raises by defining an *event delegate*. The event delegate specifies what input parameters the event handler must accept. In the .NET Framework, event delegates do not return any value and accept two input parameters:

- An `Object`, which identifies the event source, and
- A class derived from `System.EventArgs`

The second parameter passed to an event handler can include additional information about the event. While the base `EventArgs` class does not pass along any information, the .NET Framework includes a number of classes that extend `EventArgs` and encompass additional properties. For example, a `CommandEventArgs` instance is passed to event handlers that respond to the `Command` event, and includes two informational properties: `CommandArgument` and `CommandName`.

> **NOTE**
> For more information on creating, raising, and handling events, see Events and Delegates and Event Delegates in Simple English.

To define an event use the following syntax:

```
public event eventDelegate eventName;
```

Because we only need to alert the content page when the user has clicked the `DoublePrice` Button and do not need to pass along any other additional information, we can use the event delegate `EventHandler`, which defines an event handler that accepts as its second parameter an object of type `System.EventArgs`. To create the event in the master page, add the following line of code to the master page's code-behind class:

```
public partial class Site : System.Web.UI.MasterPage
{
    public event EventHandler PricesDoubled;

    ...
}
```

The above code adds a public event to the master page named `PricesDoubled`. We now need to raise this event after the prices have been doubled. To raise an event use the following syntax:

```
if (eventName != null)
    eventName(sender, eventArgs);
```

Where *sender* and *eventArgs* are the values you want to pass to the subscriber's event handler.

Update the `DoublePrice` `Click` event handler with the following code:

```
protected void DoublePrice_Click(object sender, EventArgs e)
{
    // Double the prices
    DoublePricesDataSource.Update();

    // Refresh RecentProducts
    RecentProducts.DataBind();

    // Raise the PricesDoubled event
    if (PricesDoubled != null)
    PricesDoubled(this, EventArgs.Empty);
}
```

As before, the `Click` event handler starts by calling the `DoublePricesDataSource` SqlDataSource control's `Update` method to double the prices of all products. Following that there are two additions to the event handler. First, the `RecentProducts` GridView's data is refreshed. This GridView was added to the master page in the preceding tutorial and displays the five most recently-added products. We need to refresh this grid so that it shows the just-doubled prices for these five products. Following that, the `PricesDoubled` event is raised. A reference to the master page itself (`this`) is sent to the event handler as the event source and an empty `EventArgs` object is sent as the event arguments.

Step 4: Handling the Event in the Content Page

At this point the master page raises its `PricesDoubled` event whenever the `DoublePrice` Button control is clicked. However, this is only half the battle - we still need to handle the event in the subscriber. This involves two steps: creating the event handler and adding event wiring code so that when the event is raised the event handler is executed.

Start by creating an event handler named `Master_PricesDoubled`. Because of how we defined the `PricesDoubled` event in the master page the event handler's two input parameters must be of types `Object` and `EventArgs`, respectively. In the event handler call the `ProductsGrid` GridView's `DataBind` method to rebind the data to the grid.

```
private void Master_PricesDoubled(object sender, EventArgs e)
{
    // Rebind data to ProductsGrid
    ProductsGrid.DataBind();
}
```

The code for the event handler is complete but we've yet to wire the master page's `PricesDoubled` event to this event handler. The subscriber wires an event to an event handler via the following syntax:

```
publisher.eventName += new eventDelegate(methodName);
```

publisher is a reference to the object that offers the event *eventName*, and *methodName* is the name of the event handler defined in the subscriber that has a signature corresponding to the *eventDelegate*. In other words, if the event delegate is `EventHandler`, then *methodName* must be the name of a method in the subscriber that does not return a value and accepts two input parameters of types `Object` and `EventArgs`, respectively.

This event wiring code must be executed on the first page visit and subsequent postbacks and should occur at a point in the page lifecycle that precedes when the event may be raised. A good time to add event wiring code is in the PreInit stage, which occurs very early in the page lifecycle.

Open `~/Admin/Products.aspx` and create a `Page_PreInit` event handler:

```
protected void Page_PreInit(object sender, EventArgs e)
{
    // TODO: Put event wiring logic here
}
```

In order to complete this wiring code we need a programmatic reference to the master page from the content page. As noted in the previous tutorial, there are two ways to do this:

- By casting the loosely-typed `Page.Master` property to the appropriate master page type, or
- By adding a `@MasterType` directive in the `.aspx` page and then using the strongly-typed `Master` property.

Let's use the latter approach. Add the following `@MasterType` directive to the top of the page's declarative markup:

```
<%@ MasterType VirtualPath="~/Site.master" %>
```

Then add the following event wiring code in the `Page_PreInit` event handler:

```
protected void Page_PreInit(object sender, EventArgs e)
{
    // Create an event handler for the master page's PricesDoubled event
    Master.PricesDoubled += new EventHandler(Master_PricesDoubled);
}
```

With this code in place, the GridView in the content page is refreshed whenever the `DoublePrice` Button is clicked.

Figures 8 and 9 illustrate this behavior. Figure 8 shows the page when first visited. Note that price values in both the `RecentProducts` GridView (in the left column of the master page) and the `ProductsGrid` GridView (in the content page). Figure 9 shows the same screen immediately after the `DoublePrice` Button has been clicked. As you can see, the new prices are instantaneously reflected in both GridViews.

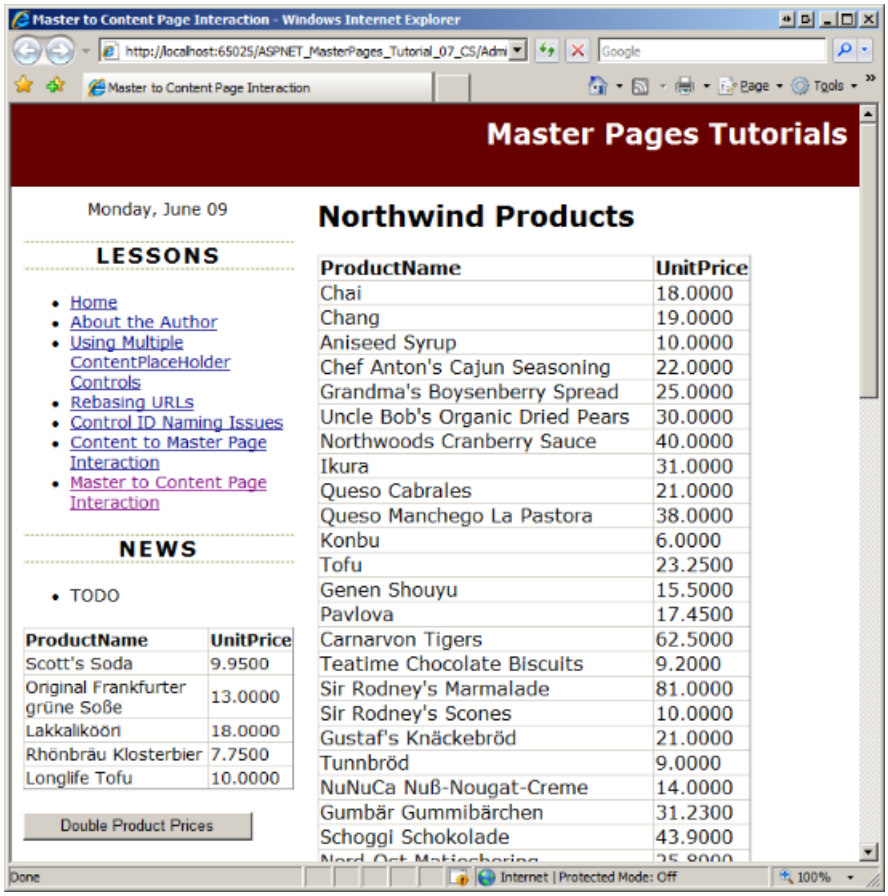


Figure 08: The Initial Price Values (Click to view full-size image)

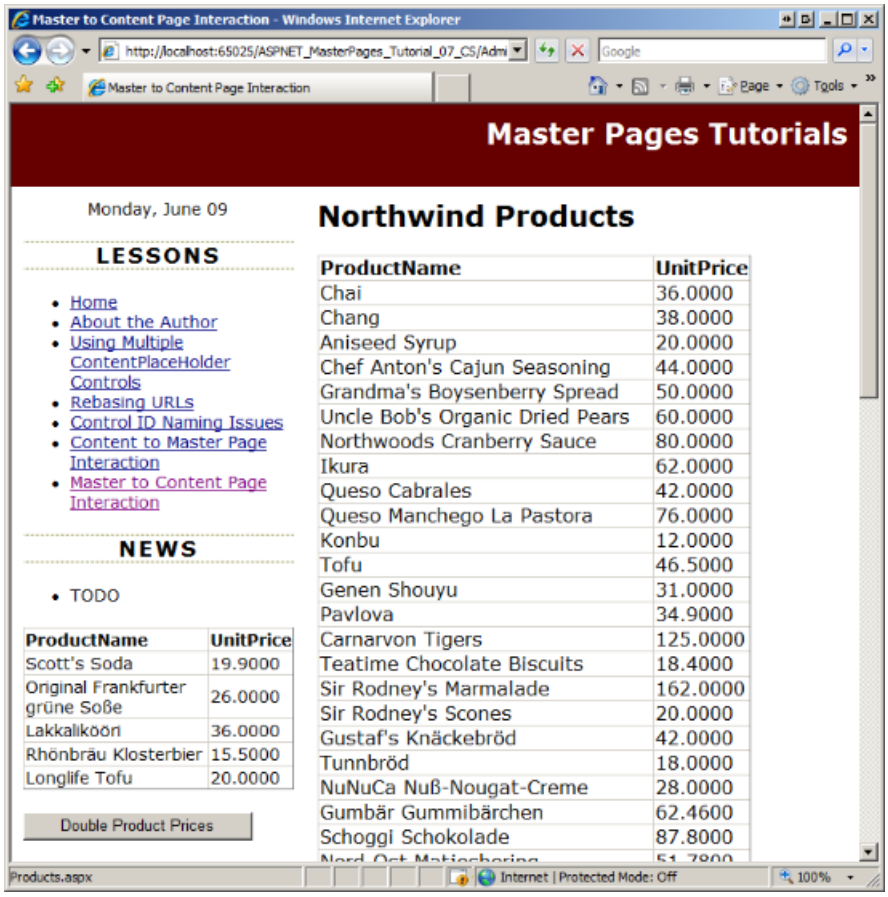


Figure 09: The Just-Doubled Prices are Displayed in the GridViews (Click to view full-size image)

Summary

Ideally, a master page and its content pages are completely separate from one another and require no level of interaction. However, if you have a master page or content page that displays data that can be modified from the master page or content page, then you may need to have the master page alert the content page (or vice-a-versa) when data is modified so that the display can be updated. In the preceding tutorial we saw how to have a content page programmatically interact with its master page; in this tutorial we looked at how to have a master page initiate the interaction.

While programmatic interaction between a content and master page can originate from either the content or master page, the interaction pattern used depends on the origination. The differences are due to the fact that a content page has a single master page, but a master page can have many different content pages. Rather than having a master page directly interact with a content page, a better approach is to have the master page raise an event to signal that some action has taken place. Those content pages that care about the action can create event handlers.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Accessing and Updating Data in ASP.NET
- Events and Delegates
- Passing Information Between Content and Master Pages
- Working with Data in ASP.NET Tutorials

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Suchi Banerjee. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Master Pages and ASP.NET AJAX (C#)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Discusses options for using ASP.NET AJAX and master pages. Looks at using the ScriptManagerProxy class; discusses how the various JS files are loaded depending on whether the ScriptManager is used in the Master page or Content page.

Introduction

Over the past several years, more and more developers have been building AJAX-enabled web applications. An AJAX-enabled website uses a number of related web technologies to offer a more responsive user experience. Creating AJAX-enabled ASP.NET applications is amazingly easy thanks to Microsoft's ASP.NET AJAX framework. ASP.NET AJAX is built into ASP.NET 3.5 and Visual Studio 2008; it is also available as a separate download for ASP.NET 2.0 applications.

When building AJAX-enabled web pages with the ASP.NET AJAX framework, you must add precisely one ScriptManager control to each and every page that uses the framework. As its name implies, the ScriptManager manages the client-side script used in AJAX-enabled web pages. At a minimum, the ScriptManager emits HTML that instructs the browser to download the JavaScript files that makeup the ASP.NET AJAX Client Library. It can also be used to register custom JavaScript files, script-enabled web services, and custom application service functionality.

If your site uses master pages (as it should), you do not necessarily need to add a ScriptManager control to every single content page; rather, you can add a ScriptManager control to the master page. This tutorial shows how to add the ScriptManager control to the master page. It also looks at how to use the ScriptManagerProxy control to register custom scripts and script services in a specific content page.

> **NOTE**
>
> This tutorial does not explore designing or building AJAX-enabled web applications with the ASP.NET AJAX framework. For more information on using AJAX consult the ASP.NET AJAX videos and tutorials, as well as those resources listed in the Further Reading section at the end of this tutorial.

Examining the Markup Emitted by the ScriptManager Control

The ScriptManager control emits markup that instructs the browser to download the JavaScript files that makeup the ASP.NET AJAX Client Library. It also adds a bit of inline JavaScript to the page that initializes this library. The following markup shows the content that is added to the rendered output of a page that includes a ScriptManager control:

```
<script src="/ASPNET_MasterPages_Tutorial_08_CS/WebResource.axd?d=T8EVk6SsA8mgPKu7_sBX5w2
t=633363040378379010" type="text/javascript"></script>

<script src="/ASPNET_MasterPages_Tutorial_08_CS/ScriptResource.axd?
d=SCE1TCh8B24VkQIU5a8iJFizuPBIqs6Lka7GEkxo-6ROKNw5LVPCpF_pmLFR-R-
p_Uf42Ahmr_SKd8lwgZUWb2uPJmfX0X_H6oLA50bniyQ1 t=633465688673751480" type="text/javascript"></script>

<script type="text/javascript">
//<![CDATA[
if (typeof(Sys) === 'undefined') throw new Error('ASP.NET Ajax client-side framework failed to load.');
//]]>
</script>

<script src="/ASPNET_MasterPages_Tutorial_08_CS/ScriptResource.axd?
d=SCE1TCh8B24VkQIU5a8iJFizuPBIqs6Lka7GEkxo-6ROKNw5LVPCpF_pmLFR-R-
phT96yZPngppiP_VXlN4Vz2RuVtvwDiQzF9xt42dUCiYjL0UylAJoyYzStwvObx0U0 t=633465688673751480"
type="text/javascript"></script>

<script type="text/javascript">
//<![CDATA[
Sys.WebForms.PageRequestManager._initialize('ScriptManager1', document.getElementById('form1'));
Sys.WebForms.PageRequestManager.getInstance()._updateControls([], [], [], 90);
//]]>
</script>

<script type="text/javascript">
//<![CDATA[
Sys.Application.initialize();
//]]>
</script>
```

The `<script src="url"></script>` tags instruct the browser to download and execute the JavaScript file at *url*. The ScriptManager emits three such tags; one references the file `WebResource.axd`, while the other two reference the file `ScriptResource.axd`. These files do not actually exist as files in your website. Instead, when a request for either one of these files arrives at the web server, the ASP.NET engine examines the querystring and returns the appropriate JavaScript content. The script provided by these three external JavaScript files constitute the ASP.NET AJAX framework's Client Library. The other `<script>` tags emitted by the ScriptManager include inline script that initializes this library.

The external script references and inline script emitted by the ScriptManager are essential for a page that uses the ASP.NET AJAX framework, but is not needed for pages that do not use the framework. Therefore, you might reason that it is ideal to only add a ScriptManager to those pages that use the ASP.NET AJAX framework. And this is sufficient, but if you have many pages that use the framework you'll end up adding the ScriptManager control to all pages - a repetitive task, to say the least. Alternatively, you can add a ScriptManager to the master page, which then injects this necessary script into all content pages. With this approach, you do not need to remember to add a ScriptManager to a new page that uses the ASP.NET AJAX framework because it is already included by the master page. Step 1 walks through adding a ScriptManager to the master page.

> **NOTE**
> If you plan on including AJAX functionality within the user interface of your master page, then you have no choice in the matter - you must include the ScriptManager in the master page.

One downside of adding the ScriptManager to the master page is that the above script is emitted in *every* page, regardless of whether its needed. This clearly leads to wasted bandwidth for those pages that have the ScriptManager included (via the master page) yet don't use any features of the ASP.NET AJAX framework. But just how much bandwidth is wasted?

- The actual content emitted by the ScriptManager (shown above) totals a little over 1KB.

- The three external script files referenced by the `<script>` element, however, comprise roughly 450KB of data uncompressed; in a website that uses gzip compression, this total bandwidth can be reduced near 100KB. However, these script files are cached by the browser for one year, meaning that they only need to be downloaded once and then can be reused in other pages on the site.

In the best case, then, when the script files are cached, the total cost is 1KB, which is negligible. In the worst case, however - which is when the script files have not yet been downloaded and the web server is not using any form of compression - the bandwidth hit is around 450KB, which can add anywhere from a second or two over a broadband connection to up to a minute for users over dial-up modems. The good news is that because the external script files are cached by the browser, this worst case scenario occurs infrequently.

> **NOTE**
> If you still feel uncomfortable placing the ScriptManager control in the master page, consider the Web Form (the `<form runat="server">` markup in the master page). Every ASP.NET page that uses the postback model must include precisely one Web Form. Adding a Web Form adds additional content: a number of hidden form fields, the `<form>` tag itself, and, if necessary, a JavaScript function for initiating a postback from script. This markup is unnecessary for pages that don't postback. This extraneous markup could be eliminated by removing the Web Form from the master page and manually adding it to each content page that needs one. However, the benefits of having the Web Form in the master page outweigh the disadvantages from having it added unnecessarily to certain content pages.

Step 1: Adding a ScriptManager Control to the Master Page

Every web page that uses the ASP.NET AJAX framework must contain precisely one ScriptManager control. Because of this requirement, it usually makes sense to place a single ScriptManager control on the master page so that all content pages have the ScriptManager control automatically included. Furthermore, the ScriptManager must come before any of the ASP.NET AJAX server controls, such as the UpdatePanel and UpdateProgress controls. Therefore, it's best to put the ScriptManager before any ContentPlaceHolder controls within the Web Form.

Open the `Site.master` master page and add a ScriptManager control to the page within the Web Form, but before the `<div id="topContent">` element (see Figure 1). If you are using Visual Web Developer 2008 or Visual Studio 2008, the ScriptManager control is located in the Toolbox in the AJAX Extensions tab. If you are using Visual Studio 2005, you will need to first install the ASP.NET AJAX framework and add the controls to the Toolbox. Visit the ASP.NET AJAX Wiki to get the framework for ASP.NET 2.0.

After adding the ScriptManager to the page, change its `ID` from `ScriptManager1` to `MyManager`.

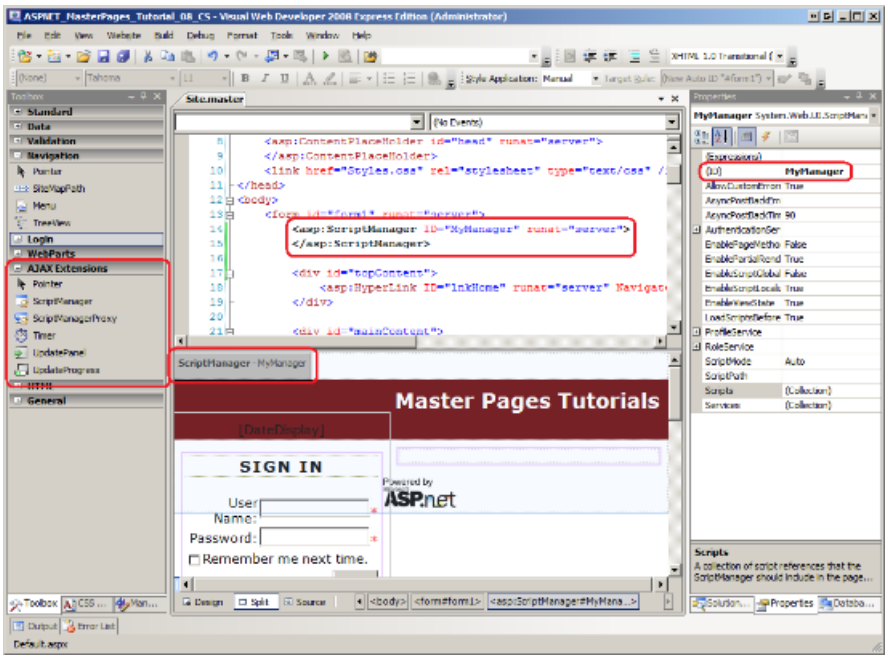


Figure 01: Add the ScriptManager to the Master Page (Click to view full-size image)

Step 2: Using the ASP.NET AJAX Framework from a Content Page

With the ScriptManager control added to the master page we can now add ASP.NET AJAX framework functionality to any content page. Let's create a new ASP.NET page that displays a randomly selected product from the Northwind database. We'll use the ASP.NET AJAX framework's Timer control to update this display every 15 seconds, showing a new product.

Start by creating a new page in the root directory named `ShowRandomProduct.aspx`. Don't forget to bind this new page to the `Site.master` master page.

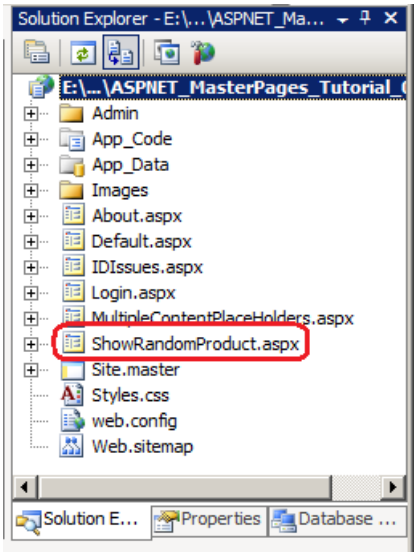


Figure 02: Add a New ASP.NET Page to the Website (Click to view full-size image)

Recall that in the *Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page* tutorial we created a custom base page class named `BasePage` that generated the page's title if it was not explicitly set. Go to the `ShowRandomProduct.aspx` page's code-behind class and have it derive from `BasePage` (instead of from `System.Web.UI.Page`).

Finally, update the `Web.sitemap` file to include an entry for this lesson. Add the following markup beneath the `<siteMapNode>` for the Master to Content Page Interaction lesson:

```
<siteMapNode url="~/ShowRandomProduct.aspx" title="Master Pages and ASP.NET AJAX" />
```

The addition of this `<siteMapNode>` element is reflected in the Lessons list (see Figure 5).

Displaying a Randomly Selected Product

Return to `ShowRandomProduct.aspx`. From the Designer, drag an UpdatePanel control from the Toolbox into the `MainContent` Content control and set its `ID` property to `ProductPanel`. The UpdatePanel represents a region on the screen that can be asynchronously updated through a partial page postback.

Our first task is to display information about a randomly selected product within the UpdatePanel. Start by dragging a DetailsView control into the UpdatePanel. Set the DetailsView control's `ID` property to `ProductInfo` and clear out its `Height` and `Width` properties. Expand the DetailsView's smart tag and, from the Choose Data Source drop-down list, choose to bind the DetailsView to a new SqlDataSource control named `RandomProductDataSource`.

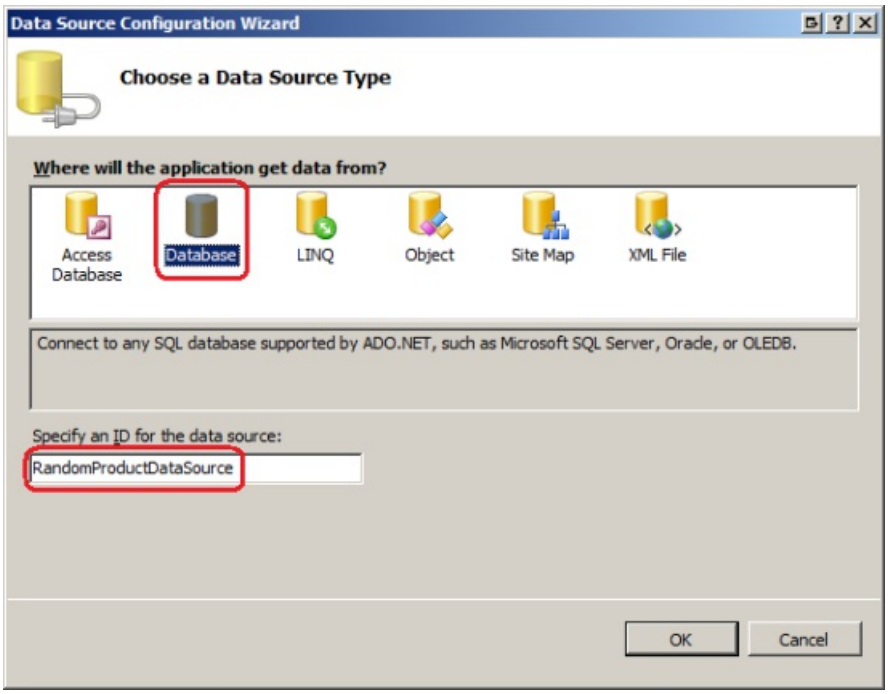


Figure 03: Bind the DetailsView to a New SqlDataSource Control (Click to view full-size image)

Configure the SqlDataSource control to connect to the Northwind database via the `NorthwindConnectionString` (which we created in the *Interacting with the Master Page from the Content Page* tutorial). When configuring the select statement choose to specify a custom SQL statement and then enter the following query:

```
SELECT TOP 1 ProductName, UnitPrice
FROM Products
ORDER BY NEWID()
```

The `TOP 1` keyword in the `SELECT` clause returns only the first record returned by the query. The `NEWID()` function generates a new globally unique identifier value (GUID) and can be used in an `ORDER BY` clause to return the table's records in a random order.

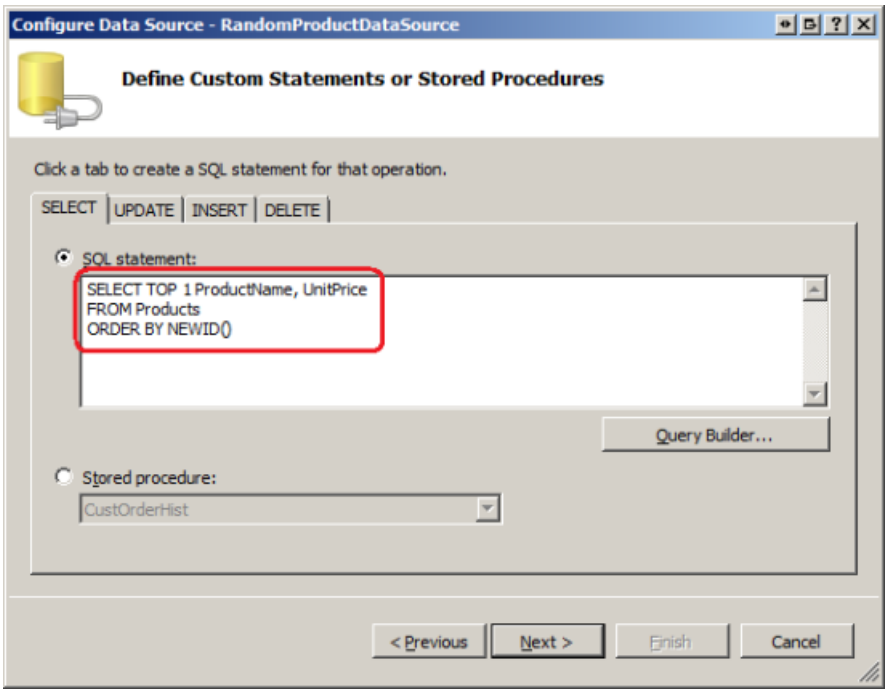


Figure 04: Configure the SqlDataSource to Return a Single, Randomly Selected Record (Click to view full-size image)

After completing the wizard, Visual Studio creates a BoundField for the two columns returned by the above query. At this point your page's declarative markup should look similar to the following:

```
<asp:UpdatePanel ID="ProductPanel" runat="server">
 <ContentTemplate>
 <asp:DetailsView ID="ProductInfo" runat="server" AutoGenerateRows="False"
 DataSourceID="RandomProductDataSource">
 <Fields>
 <asp:BoundField DataField="ProductName" HeaderText="ProductName"
 SortExpression="ProductName" />
 <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
 SortExpression="UnitPrice" />
 </Fields>
 </asp:DetailsView>
 <asp:SqlDataSource ID="RandomProductDataSource" runat="server"
 ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>" SelectCommand="SELECT TOP 1
ProductName, UnitPrice FROM Products ORDER BY NEWID()"></asp:SqlDataSource>
 </ContentTemplate>
</asp:UpdatePanel>
```

Figure 5 shows the `ShowRandomProduct.aspx` page when viewed through a browser. Click your browser's Refresh button to reload the page; you should see the `ProductName` and `UnitPrice` values for a new randomly selected record.

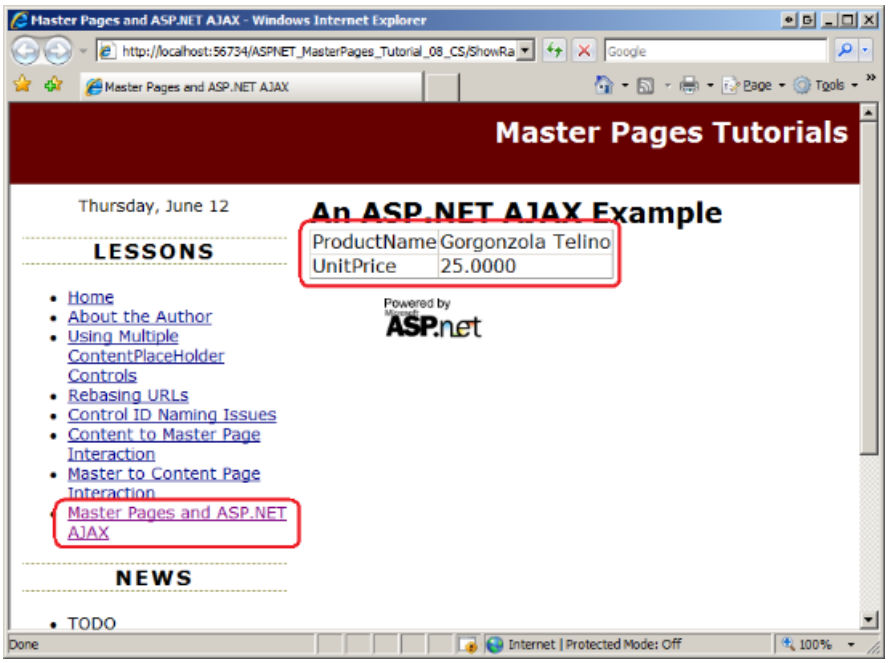


Figure 05: A Random Product's Name and Price is Displayed (Click to view full-size image)

Automatically Displaying a New Product Every 15 Seconds

The ASP.NET AJAX framework includes a Timer control that performs a postback at a specified time; on postback the Timer's `Tick` event is raised. If the Timer control is placed within an UpdatePanel it triggers a partial page postback, during which we can rebind the data to the DetailsView to display a new randomly selected product.

To accomplish this, drag a Timer from the Toolbox and drop it into the UpdatePanel. Change the Timer's `ID` from `Timer1` to `ProductTimer` and its `Interval` property from 60000 to 15000. The `Interval` property indicates the number of milliseconds between postbacks; setting it to 15000 causes the Timer to trigger a partial page postback every 15 seconds. At this point your Timer's declarative markup should look similar to the following:

```
<asp:UpdatePanel ID="ProductPanel" runat="server" onload="ProductPanel_Load">
 <ContentTemplate>
 ...

 <asp:Timer ID="ProductTimer" runat="server" Interval="15000">
 </asp:Timer>
 </ContentTemplate>
</asp:UpdatePanel>
```

Create an event handler for the Timer's `Tick` event. In this event handler we need to rebind the data to the DetailsView by calling the DetailsView's `DataBind` method. Doing so instructs the DetailsView to re-retrieve the data from its data source control, which will select and display a new randomly selected record (just like when reloading the page by clicking the browser's Refresh button).

```
protected void ProductTimer_Tick(object sender, EventArgs e)
{
    ProductInfo.DataBind();
}
```

That's all there is to it! Revisit the page through a browser. Initially, a random product's information is displayed. If you patiently watch the screen you'll notice that, after 15 seconds, information about a new product magically replaces the existing display.

To better see what's happening here, let's add a Label control to the UpdatePanel that displays the time the display was last updated. Add a Label Web control within the UpdatePanel, set its `ID` to `LastUpdateTime`, and clear its

`Text` property. Next, create an event handler for the UpdatePanel's `Load` event and display the current time in the Label. (The UpdatePanel's `Load` event is fired on every full or partial page postback.)

```
protected void ProductPanel_Load(object sender, EventArgs e)
{
    LastUpdateTime.Text = "Last updated at " + DateTime.Now.ToLongTimeString();
}
```

With this change complete, the page includes the time the currently displayed product was loaded. Figure 6 shows the page when first visited. Figure 7 shows the page 15 seconds later after the Timer control has "ticked" and the UpdatePanel has been refreshed to display information about a new product.

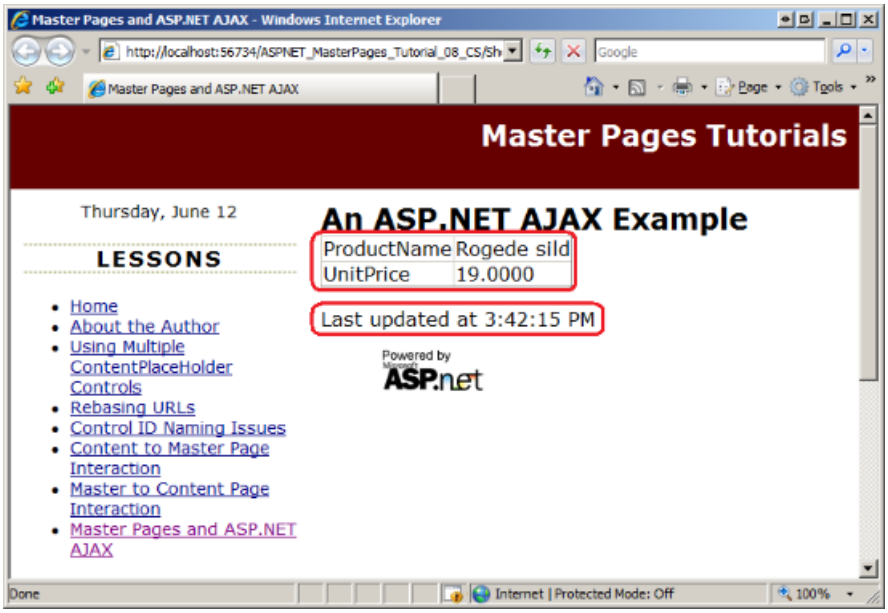


Figure 06: A Randomly Selected Product is Displayed on Page Load (Click to view full-size image)

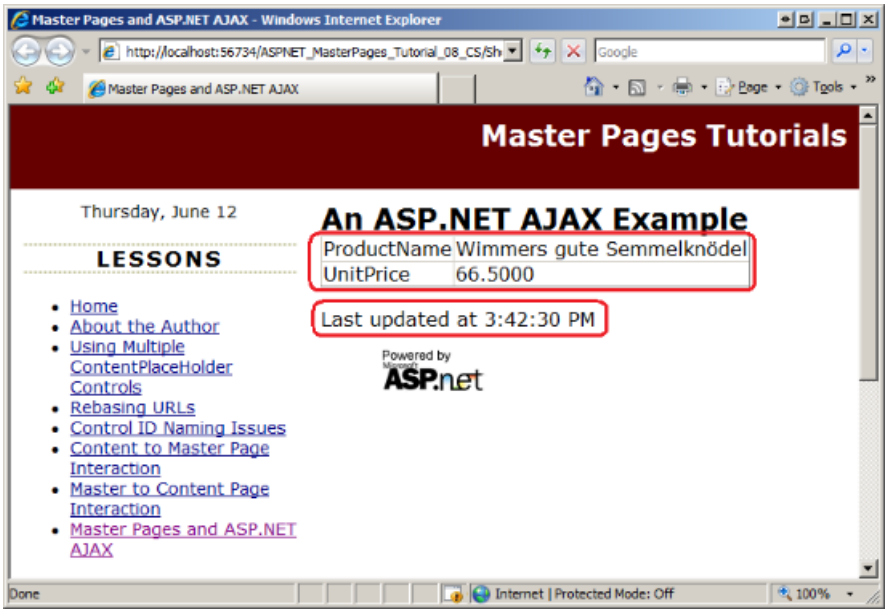


Figure 07: Every 15 Seconds a New Randomly Selected Product is Displayed (Click to view full-size image)

Step 3: Using the ScriptManagerProxy Control

Along with including the necessary script for the ASP.NET AJAX framework Client Library, the ScriptManager can also register custom JavaScript files, references to script-enabled Web Services, and custom authentication, authorization, and profile services. Usually such customizations are specific to a certain page. However, if the custom script files, Web Service references, or authentication, authorization, or profile services are referenced in

the ScriptManager in the master page then they are included in *all* pages in the website.

To add ScriptManager-related customizations on a page-by-page basis use the ScriptManagerProxy control. You can add a ScriptManagerProxy to a content page and then register the custom JavaScript file, Web Service reference, or authentication, authorization, or profile service from the ScriptManagerProxy; this has the effect of registering these services for the particular content page.

> **NOTE**
>
> An ASP.NET page can only have no more than one ScriptManager control present. Therefore, you cannot add a ScriptManager control to a content page if the ScriptManager control is already defined in the master page. The sole purpose of the ScriptManagerProxy is to provide a way for developers to define the ScriptManager in the master page, but still have the ability to add ScriptManager customizations on a page-by-page basis.

To see the ScriptManagerProxy control in action, let's augment the UpdatePanel in `ShowRandomProduct.aspx` to include a button that uses client-side script to pause or resume the Timer control. The Timer control has three client-side methods that we can use to achieve this desired functionality:

- `_startTimer()` - starts the Timer control
- `_raiseTick()` - causes the Timer control to "tick," thereby posting back and raising its `Tick` event on the server
- `_stopTimer()` - stops the Timer control

Let's create a JavaScript file with a variable named `timerEnabled` and a function named `ToggleTimer`. The `timerEnabled` variable indicates whether the Timer control is currently enabled or disabled; it defaults to true. The `ToggleTimer` function accepts two input parameters: a reference to the Pause/Resume button and the client-side `id` value of the Timer control. This function toggles the value of `timerEnabled`, gets a reference to the Timer control, starts or stops the Timer (depending on the value of `timerEnabled`), and updates the button's display text to "Pause" or "Resume". This function will be called whenever the Pause/Resume button is clicked.

Start by creating a new folder in the website named `Scripts`. Next, add a new file to the Scripts folder named `TimerScript.js` of type JScript File.

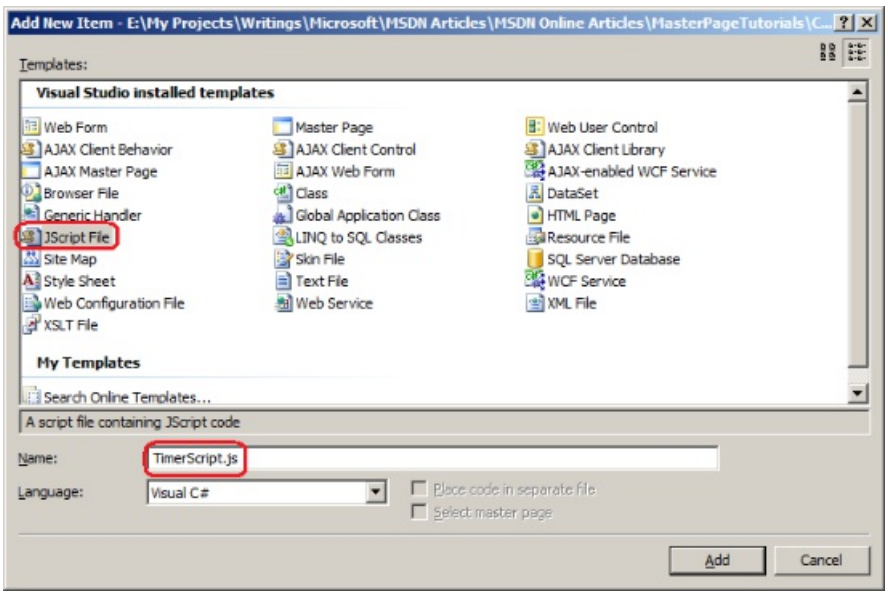


Figure 08: Add a New JavaScript File to the `Scripts` Folder (Click to view full-size image)

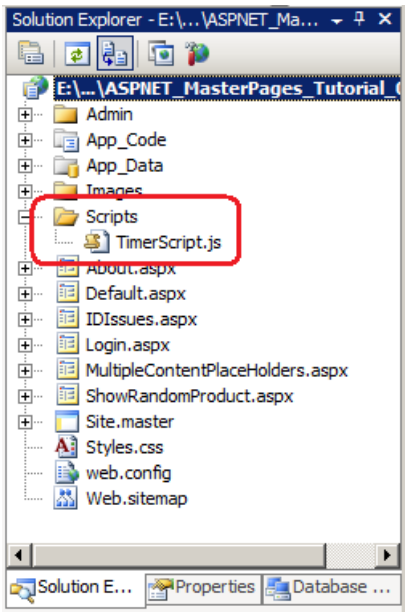


Figure 09: A New JavaScript File has been Added to the Website (Click to view full-size image)

Next, add the following scrip to the TimerScript.js file:

```
var timerEnabled = true;
function ToggleTimer(btn, timerID)
{
    // Toggle the timer enabled state
    timerEnabled = !timerEnabled;

    // Get a reference to the Timer
    var timer = $find(timerID);

    if (timerEnabled)
    {
        // Start timer
        timer._startTimer();

        // Immediately raise a tick
        timer._raiseTick();

        btn.value = 'Pause';
    }
    else
    {
        // Stop timer
        timer._stopTimer();

        btn.value = 'Resume';
    }
}
```

We now need to register this custom JavaScript file in `ShowRandomProduct.aspx`. Return to `ShowRandomProduct.aspx` and add a ScriptManagerProxy control to the page; set its `ID` to `MyManagerProxy`. To register a custom JavaScript file select the ScriptManagerProxy control in the Designer and then go to the Properties window. One of the properties is titled Scripts. Selecting this property displays the ScriptReference Collection Editor shown in Figure 10. Click the Add button to include a new script reference and then enter the path to the script file in the Path property: `~/Scripts/TimerScript.js`.

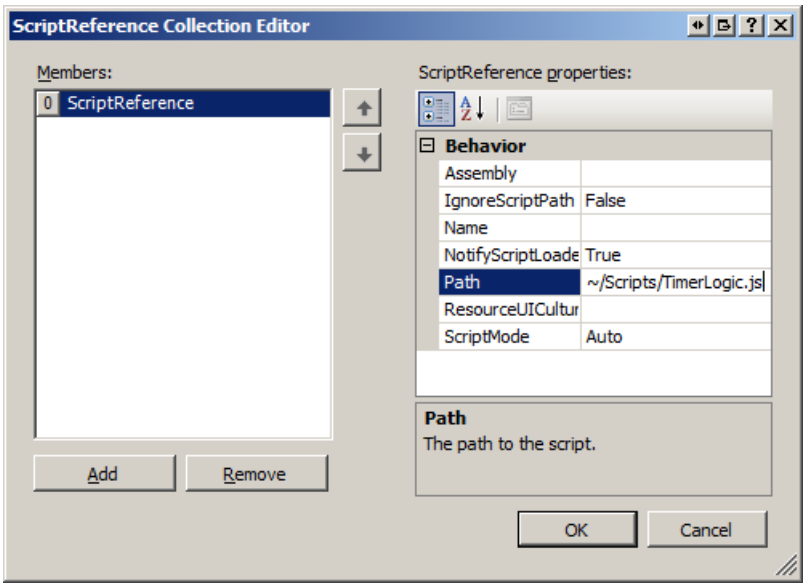


Figure 10: Add a Script Reference to the ScriptManagerProxy Control (Click to view full-size image)

After adding the script reference the ScriptManagerProxy control's declarative markup is updated to include a `<Scripts>` collection with a single `ScriptReference` entry, as the following snippet of markup illustrates:

```
<asp:ScriptManagerProxy ID="MyManagerProxy" runat="server">
 <Scripts>
 <asp:ScriptReference Path="~/Scripts/TimerScript.js" />
 </Scripts>
</asp:ScriptManagerProxy>
```

The `ScriptReference` entry instructs the ScriptManagerProxy to include a reference to the JavaScript file in its rendered markup. That is, by registering the custom script in the ScriptManagerProxy the `ShowRandomProduct.aspx` page's rendered output now includes another `<script src="url"></script>` tag: `<script src="Scripts/TimerScript.js" type="text/javascript"></script>`.

We can now call the `ToggleTimer` function defined in `TimerScript.js` from the client script in the `ShowRandomProduct.aspx` page. Add the following HTML within the UpdatePanel:

```
<input type="button" id="PauseResumeButton"
    value="Pause"
    onclick="ToggleTimer(this, '<%=ProductTimer.ClientID %>');" />
```

This displays a button with the text "Pause". Whenever it is clicked, the JavaScript function `ToggleTimer` is called, passing in a reference to the button and the id value of the Timer control (`ProductTimer`). Note the syntax for obtaining the `id` value of the Timer control. `<%=ProductTimer.ClientID%>` emits the value of the `ProductTimer` Timer control's `ClientID` property. In the *Control ID Naming in Content Pages* tutorial we discussed the differences between the server-side `ID` value and the resulting client-side `id` value, and how `ClientID` returns the client-side `id`.

Figure 11 shows this page when first visited through a browser. The Timer is currently running and updates the displayed product information every 15 seconds. Figure 12 shows the screen after the Pause button has been clicked. Clicking the Pause button stops the Timer and updates the button's text to "Resume". The product information will refresh (and continue to refresh every 15 seconds) once the user clicks Resume.

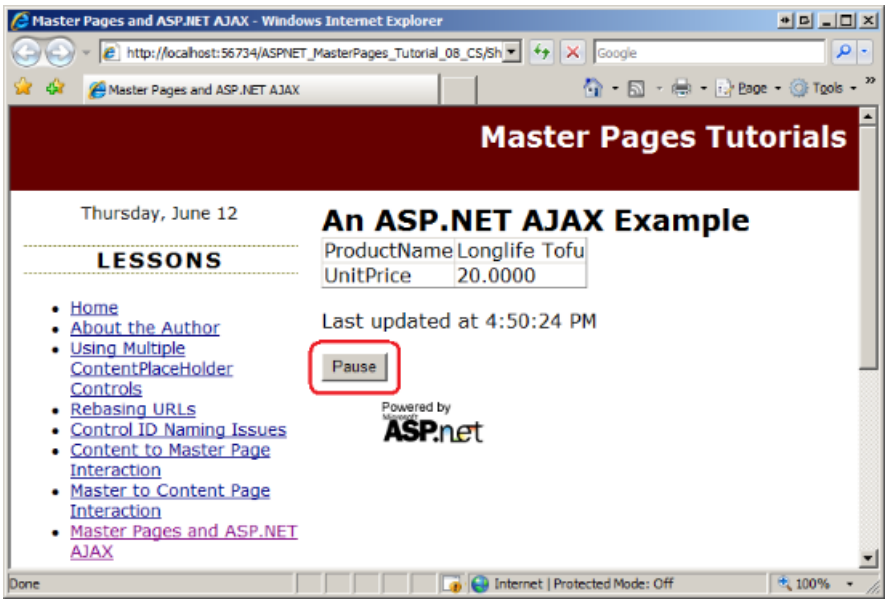


Figure 11: Click the Pause Button to Stop the Timer Control (Click to view full-size image)

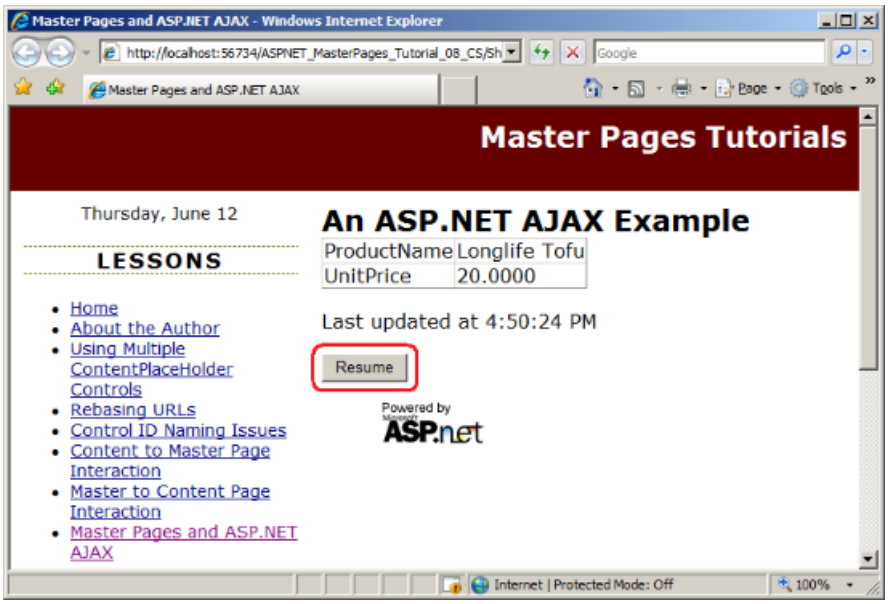


Figure 12: Click the Resume Button to Restart the Timer (Click to view full-size image)

Summary

When building AJAX-enabled web applications using the ASP.NET AJAX framework it is imperative that every AJAX-enabled web page include a ScriptManager control. To facilitate this process, we can add a ScriptManager to the master page rather than having to remember to add a ScriptManager to each and every content page. Step 1 showed how to add the ScriptManager to the master page while Step 2 looked at implementing AJAX functionality in a content page.

If you need to add custom scripts, references to script-enabled Web Services, or customized authentication, authorization, or profile services to a particular content page, add a ScriptManagerProxy control to the content page and then configure the customizations there. Step 3 examined how to use the ScriptManagerProxy to register a custom JavaScript file in a specific content page.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET AJAX Framework

- ASP.NET AJAX Tutorials
- ASP.NET AJAX Videos
- Building Interactive User Interface with Microsoft ASP.NET AJAX
- Using NEWID to Randomly Sort Records
- Using the Timer Control

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Specifying the Master Page Programmatically (C#)

2/20/2020 • 19 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> Looks at setting the content page's master page programmatically via the PreInit event handler.

Introduction

Since the inaugural example in *Creating a Site-Wide Layout Using Master Pages*, all content pages have referenced their master page declaratively via the `MasterPageFile` attribute in the `@Page` directive. For example, the following `@Page` directive links the content page to the master page `Site.master`:

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" ... %>
```

The `Page` class in the `System.Web.UI` namespace includes a `MasterPageFile` property that returns the path to the content page's master page; it is this property that is set by the `@Page` directive. This property can also be used to programmatically specify the content page's master page. This approach is useful if you want to dynamically assign the master page based on external factors, such as the user visiting the page.

In this tutorial we add a second master page to our website and dynamically decide which master page to use at runtime.

Step 1: A Look at the Page Lifecycle

Whenever a request arrives at the web server for an ASP.NET page that is a content page, the ASP.NET engine must fuse the page's Content controls into the master page's corresponding ContentPlaceHolder controls. This fusion creates a single control hierarchy that can then proceed through the typical page lifecycle.

Figure 1 illustrates this fusion. Step 1 in Figure 1 shows the initial content and master page control hierarchies. At the tail end of the PreInit stage the Content controls in the page are added to the corresponding ContentPlaceHolders in the master page (Step 2). After this fusion, the master page serves as the root of the fused control hierarchy. This fused control hierarchy is then added to the page to produce the finalized control hierarchy (Step 3). The net result is that the page's control hierarchy includes the fused control hierarchy.

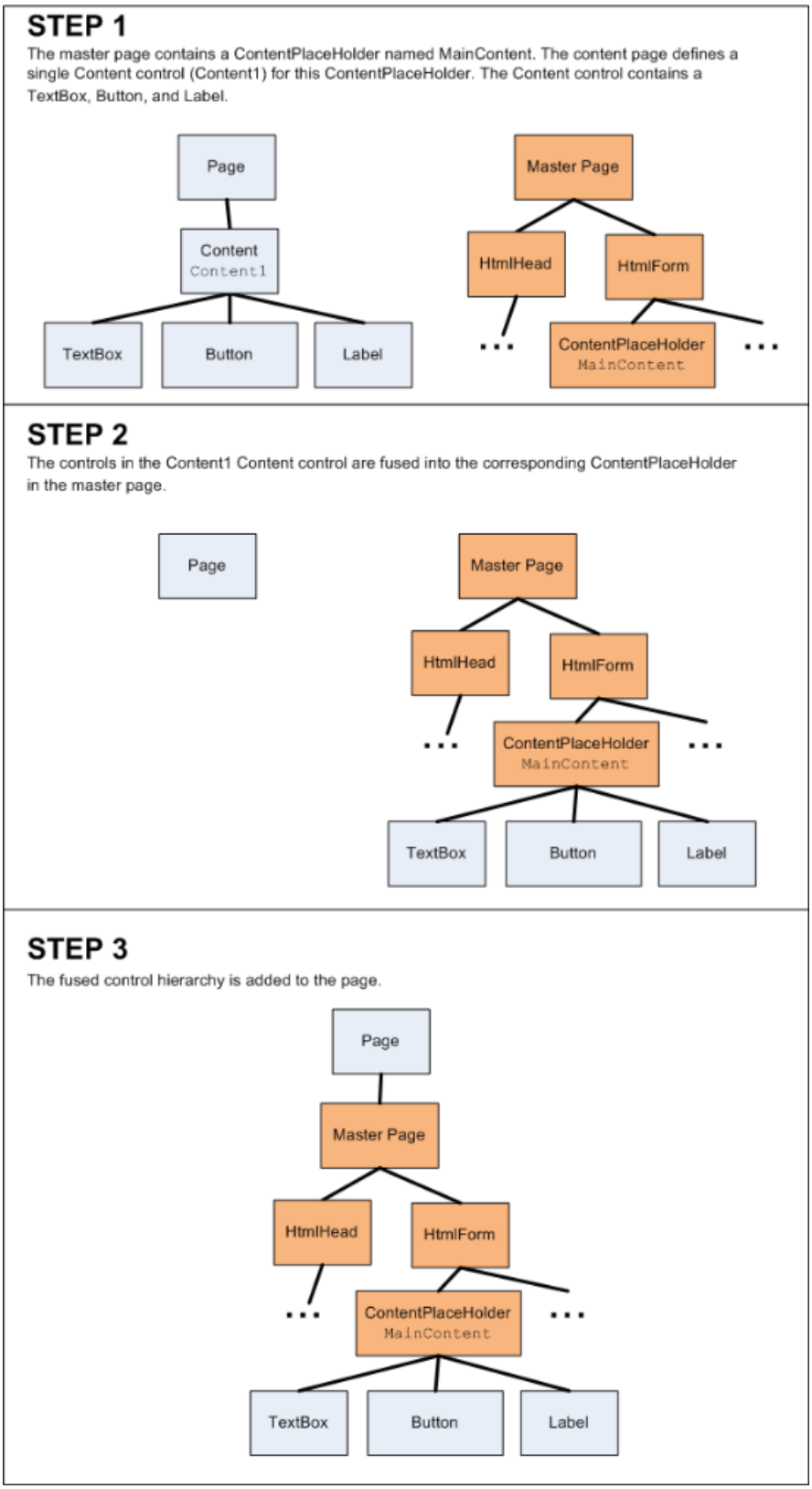


Figure 01: The Master Page and Content Page's Control Hierarchies are Fused Together during the PreInit Stage (Click to view full-size image)

Step 2: Setting the `MasterPageFile` Property from Code

What master page partakes in this fusion depends on the value of the `Page` object's `MasterPageFile` property. Setting the `MasterPageFile` attribute in the `@Page` directive has the net effect of assigning the `Page`'s `MasterPageFile` property during the Initialization stage, which is the very first stage of the page's lifecycle. We can alternatively set this property programmatically. However, it is imperative that this property be set before the fusion in Figure 1 takes place.

At the start of the PreInit stage the `Page` object raises its `PreInit` event and calls its `OnPreInit` method. To set the master page programmatically, then, we can either create an event handler for the `PreInit` event or override the `OnPreInit` method. Let's look at both approaches.

Start by opening `Default.aspx.cs`, the code-behind class file for our site's homepage. Add an event handler for the page's `PreInit` event by typing in the following code:

```
protected void Page_PreInit(object sender, EventArgs e)
{
}
```

From here we can set the `MasterPageFile` property. Update the code so that it assigns the value "~/Site.master" to the `MasterPageFile` property.

```
protected void Page_PreInit(object sender, EventArgs e)
{
    this.MasterPageFile = "~/Site.master";
}
```

If you set a breakpoint and start with debugging you'll see that whenever the `Default.aspx` page is visited or whenever there's a postback to this page, the `Page_PreInit` event handler executes and the `MasterPageFile` property is assigned to "~/Site.master".

Alternatively, you can override the `Page` class's `OnPreInit` method and set the `MasterPageFile` property there. For this example, let's not set the master page in a particular page, but rather from `BasePage`. Recall that we created a custom base page class (`BasePage`) back in the *Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page* tutorial. Currently `BasePage` overrides the `Page` class's `OnLoadComplete` method, where it sets the page's `Title` property based on the site map data. Let's update `BasePage` to also override the `OnPreInit` method to programmatically specify the master page.

```
protected override void OnPreInit(EventArgs e)
{
    this.MasterPageFile = "~/Site.master";
    base.OnPreInit(e);
}
```

Because all our content pages derive from `BasePage`, all of them now have their master page programmatically assigned. At this point the `PreInit` event handler in `Default.aspx.cs` is superfluous; feel free to remove it.

What About the `@Page` Directive?

What may be a little confusing is that the content pages' `MasterPageFile` properties are now being specified in two places: programmatically in the `BasePage` class's `OnPreInit` method as well as through the `MasterPageFile` attribute in each content page's `@Page` directive.

The first stage in the page lifecycle is the Initialization stage. During this stage the `Page` object's `MasterPageFile` property is assigned the value of the `MasterPageFile` attribute in the `@Page` directive (if it is provided). The PreInit stage follows the Initialization stage, and it is here where we programmatically set the `Page` object's `MasterPageFile` property, thereby overwriting the value assigned from the `@Page` directive. Because we are setting the `Page` object's `MasterPageFile` property programmatically, we could remove the `MasterPageFile` attribute from the `@Page` directive without affecting the end user's experience. To convince yourself of this, go ahead and remove the `MasterPageFile` attribute from the `@Page` directive in `Default.aspx` and then visit the page through a browser. As you would expect, the output is the same as before the attribute was removed.

Whether the `MasterPageFile` property is set via the `@Page` directive or programmatically is inconsequential to the

end user's experience. However, the `MasterPageFile` attribute in the `@Page` directive is used by Visual Studio during design-time to produce the WYSIWYG view in the Designer. If you return to `Default.aspx` in Visual Studio and navigate to the Designer you'll see the message, "Master Page error: The page has controls that require a Master Page reference, but none is specified" (see Figure 2).

In short, you need to leave the `MasterPageFile` attribute in the `@Page` directive to enjoy a rich design-time experience in Visual Studio.

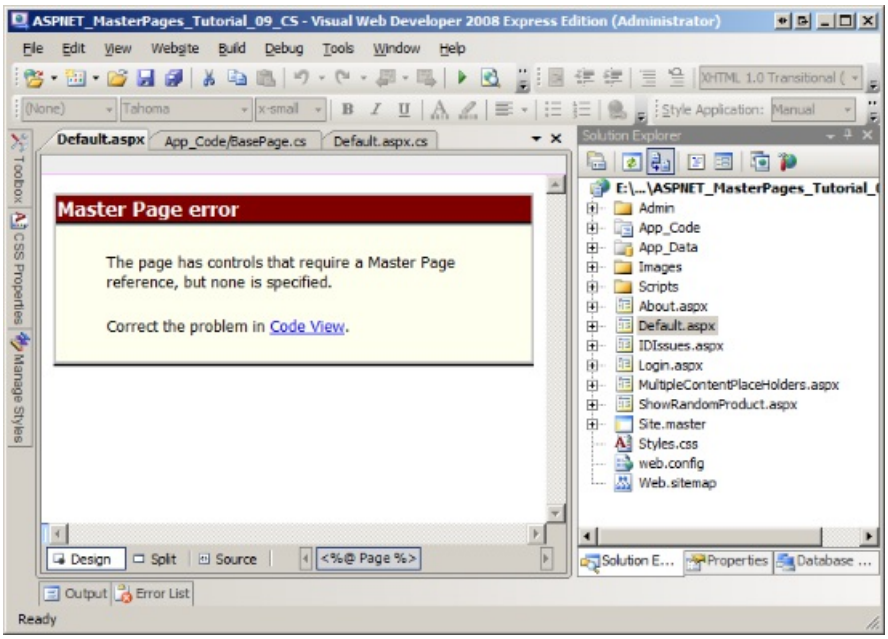


Figure 02: Visual Studio Uses the `@Page` Directive's `MasterPageFile` Attribute to Render the Design View (Click to view full-size image)

Step 3: Creating an Alternative Master Page

Because a content page's master page can be set programmatically at runtime it's possible to dynamically load a particular master page based on some external criteria. This functionality can be useful in situations where the site's layout needs to vary based on the user. For instance, a blog engine web application may allow its users to choose a layout for their blog, where each layout is associated with a different master page. At runtime, when a visitor is viewing a user's blog, the web application would need to determine the blog's layout and dynamically associate the corresponding master page with the content page.

Let's examine how to dynamically load a master page at runtime based on some external criteria. Our website currently contains just one master page (`Site.master`). We need another master page to illustrate choosing a master page at runtime. This step focuses on creating and configuring the new master page. Step 4 looks at determining what master page to use at runtime.

Create a new master page in the root folder named `Alternate.master`. Also add a new style sheet to the website named `AlternateStyles.css`.

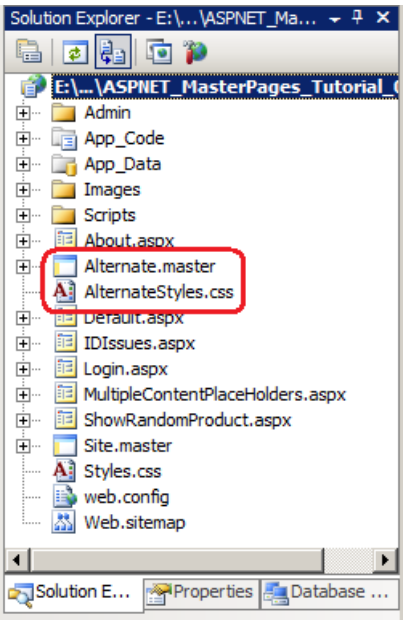


Figure 03: Add Another Master Page and CSS File to the Website (Click to view full-size image)

I've designed the `Alternate.master` master page to have the title displayed at the top of the page, centered and on a navy background. I've dispensed of the left column and moved that content beneath the `MainContent` ContentPlaceHolder control, which now spans the entire width of the page. Furthermore, I nixed the unordered Lessons list and replaced it with a horizontal list above `MainContent`. I also updated the fonts and colors used by the master page (and, by extension, its content pages). Figure 4 shows `Default.aspx` when using the `Alternate.master` master page.

> **NOTE**
>
> ASP.NET includes the ability to define *Themes*. A Theme is a collection of images, CSS files, and style-related Web control property settings that can be applied to a page at runtime. Themes are the way to go if your site's layouts differ only in the images displayed and by their CSS rules. If the layouts differ more substantially, such as using different Web controls or having a radically different layout, then you will need to use separate master pages. Consult the Further Reading section at the end of this tutorial for more information on Themes.

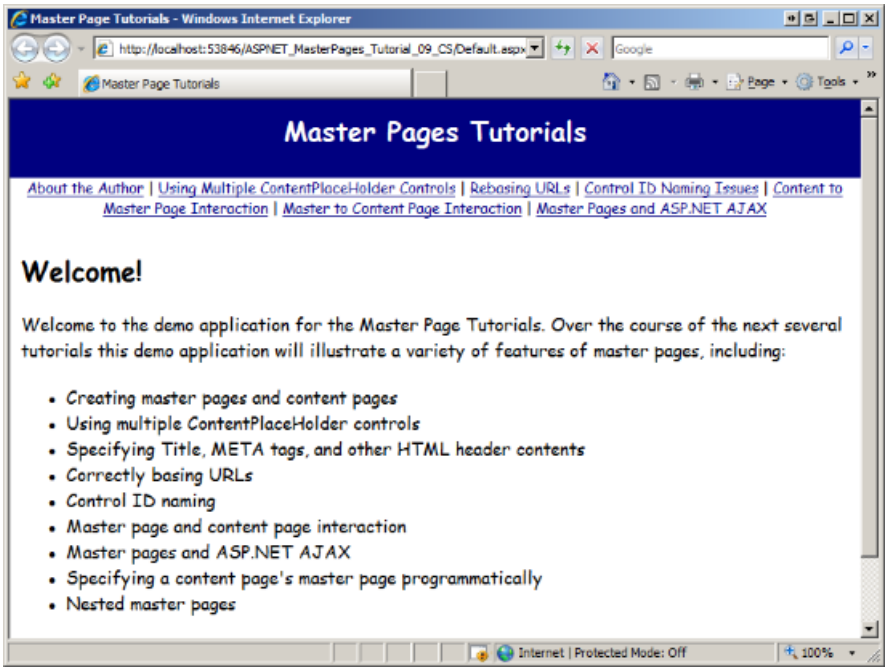


Figure 04: Our Content Pages Can Now Use a New Look and Feel (Click to view full-size image)

When the master and content pages' markup are fused, the `MasterPage` class checks to ensure that every Content control in the content page references a ContentPlaceHolder in the master page. An exception is thrown if a Content control that references a non-existent ContentPlaceHolder is found. In other words, it is imperative that the master page being assigned to the content page have a ContentPlaceHolder for each Content control in the content page.

The `Site.master` master page includes four ContentPlaceHolder controls:

- `head`
- `MainContent`
- `QuickLoginUI`
- `LeftColumnContent`

Some of the content pages in our website include just one or two Content controls; others include a Content control for each of the available ContentPlaceHolders. If our new master page (`Alternate.master`) may ever be assigned to those content pages that have Content controls for all of the ContentPlaceHolders in `Site.master` then it is essential that `Alternate.master` also include the same ContentPlaceHolder controls as `Site.master`.

To get your `Alternate.master` master page to look similar to mine (see Figure 4), start by defining the master page's styles in the `AlternateStyles.css` style sheet. Add the following rules into `AlternateStyles.css`:

```
body
{
 font-family: Comic Sans MS, Arial;
 font-size: medium;
 margin: 0px;
}
#topContent
{
 text-align: center;
 background-color: Navy;
 color: White;
 font-size: x-large;
 text-decoration: none;
 font-weight: bold;
 padding: 10px;
 height: 50px;
}
#topContent a
{
 text-decoration: none;
 color: White;
}
#navContent
{
 font-size: small;
 text-align: center;
}
#footerContent
{
 padding: 10px;
 font-size: 90%;
 text-align: center;
 border-top: solid 1px black;
}
#mainContent
{
 text-align: left;
 padding: 10px;
}
```

Next, add the following declarative markup to `Alternate.master`. As you can see, `Alternate.master` contains four ContentPlaceHolder controls with the same `ID` values as the ContentPlaceHolder controls in `Site.master`. Moreover, it includes a ScriptManager control, which is necessary for those pages in our website that use the ASP.NET AJAX framework.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
 <link href="AlternateStyles.css" rel="stylesheet" type="text/css" />
</head>
<body>
 <form id="form1" runat="server">
 <asp:ScriptManager ID="MyManager" runat="server">
 </asp:ScriptManager>
 <div id="topContent">
 <asp:HyperLink ID="lnkHome" runat="server" NavigateUrl="~/Default.aspx"
 Text="Master Pages Tutorials" />
 </div>
 <div id="navContent">
 <asp:ListView ID="LessonsList" runat="server"
 DataSourceID="LessonsDataSource">
 <LayoutTemplate>
 <asp:PlaceHolder runat="server" ID="itemPlaceholder" />
 </LayoutTemplate>
 <ItemTemplate>
 <asp:HyperLink runat="server" ID="lnkLesson"
 NavigateUrl='<%# Eval("Url") %>'
 Text='<%# Eval("Title") %>' />
 </ItemTemplate>
 <ItemSeparatorTemplate> | </ItemSeparatorTemplate>
 </asp:ListView>
 <asp:SiteMapDataSource ID="LessonsDataSource" runat="server"
 ShowStartingNode="false" />
 </div>
 <div id="mainContent">
 <asp:ContentPlaceHolder id="MainContent" runat="server">
 </asp:ContentPlaceHolder>
 </div>
 <div id="footerContent">
 <p>
 <asp:Label ID="DateDisplay" runat="server"></asp:Label>
 </p>
 <asp:ContentPlaceHolder ID="QuickLoginUI" runat="server">
 </asp:ContentPlaceHolder>
 <asp:ContentPlaceHolder ID="LeftColumnContent" runat="server">
 </asp:ContentPlaceHolder>
 </div>
 </form>
</body>
</html>
```

Testing the New Master Page

To test this new master page update the `BasePage` class's `OnPreInit` method so that the `MasterPageFile` property is assigned the value "~/Alternate.master" and then visit the website. Every page should function without error except for two: `~/Admin/AddProduct.aspx` and `~/Admin/Products.aspx`. Adding a product to the DetailsView in `~/Admin/AddProduct.aspx` results in a `NullReferenceException` from the line of code that attempts to set the master page's `GridMessageText` property. When visiting `~/Admin/Products.aspx` an `InvalidCastException` is thrown on page load with the message: "Unable to cast object of type 'ASP.alternate_master' to type 'ASP.site_master'."

These errors occur because the `Site.master` code-behind class includes public events, properties, and methods that are not defined in `Alternate.master`. The markup portion of these two pages have a `@MasterType` directive that references the `Site.master` master page.

```
<%@ MasterType VirtualPath="~/Site.master" %>
```

Also, the DetailsView's `ItemInserted` event handler in `~/Admin/AddProduct.aspx` includes code that casts the loosely-typed `Page.Master` property to an object of type `Site`. The `@MasterType` directive (used this way) and the cast in the `ItemInserted` event handler tightly couples the `~/Admin/AddProduct.aspx` and `~/Admin/Products.aspx` pages to the `Site.master` master page.

To break this tight coupling we can have `Site.master` and `Alternate.master` derive from a common base class that contains definitions for the public members. Following that, we can update the `@MasterType` directive to reference this common base type.

Creating a Custom Base Master Page Class

Add a new class file to the `App_Code` folder named `BaseMasterPage.cs` and have it derive from `System.Web.UI.MasterPage`. We need to define the `RefreshRecentProductsGrid` method and the `GridMessageText` property in `BaseMasterPage`, but we can't simply move them there from `Site.master` because these members work with Web controls that are specific to the `Site.master` master page (the `RecentProducts` GridView and `GridMessage` Label).

What we need to do is configure `BaseMasterPage` in such a way that these members are defined there, but are actually implemented by `BaseMasterPage`'s derived classes (`Site.master` and `Alternate.master`). This type of inheritance is possible by marking the class and its members as `abstract`. In short, adding the `abstract` keyword to these two members announces that `BaseMasterPage` hasn't implemented `RefreshRecentProductsGrid` and `GridMessageText`, but that its derived classes will.

We also need to define the `PricesDoubled` event in `BaseMasterPage` and provide a means by the derived classes to raise the event. The pattern used in the .NET Framework to facilitate this behavior is to create a public event in the base class and add a protected, `virtual` method named `OnEventName`. Derived classes can then call this method to raise the event or can override it to execute code immediately before or after the event is raised.

Update your `BaseMasterPage` class so that it contains the following code:

```
using System; public abstract class BaseMasterPage : System.Web.UI.MasterPage
{
    public event EventHandler PricesDoubled;
    protected virtual void OnPricesDoubled(EventArgs e)
    {
        if (PricesDoubled != null)
        PricesDoubled(this, e);
    }
    public abstract void RefreshRecentProductsGrid();
    public abstract string GridMessageText
    {
        get;
        set;
    }
}
```

Next, go to the `Site.master` code-behind class and have it derive from `BaseMasterPage`. Because `BaseMasterPage` is `abstract` we need to override those `abstract` members here in `Site.master`. Add the `override` keyword to the method and property definitions. Also update the code that raises the `PricesDoubled` event in the `DoublePrice` Button's `Click` event handler with a call to the base class's `OnPricesDoubled` method.

After these modifications the `Site.master` code-behind class should contain the following code:

```
public partial class Site : BaseMasterPage {
    protected void Page_Load(object sender, EventArgs e)
    {
        DateDisplay.Text = DateTime.Now.ToString("dddd, MMMM dd");
    }
    public override void RefreshRecentProductsGrid()
    {
        RecentProducts.DataBind();
    }
    public override string GridMessageText
    {
        get
        {
            return GridMessage.Text;
        }
        set
        {
            GridMessage.Text = value;
        }
    }
    protected void DoublePrice_Click(object sender, EventArgs e)
    {
        // Double the prices
        DoublePricesDataSource.Update();
        // Refresh RecentProducts
        RecentProducts.DataBind();
        // Raise the PricesDoubled event
        base.OnPricesDoubled(EventArgs.Empty);
    }
}
```

We also need to update `Alternate.master`'s code-behind class to derive from `BaseMasterPage` and override the two `abstract` members. But because `Alternate.master` does not contain a GridView that lists the most recent products nor a Label that displays a message after a new product is added to the database, these methods do not need to do anything.

```
public partial class Alternate : BaseMasterPage
{
    public override void RefreshRecentProductsGrid()
    {
        // Do nothing
    }
    public override string GridMessageText
    {
        get
        {
            return string.Empty;
        }
        set
        {
            // Do nothing
        }
    }
}
```

Referencing the Base Master Page Class

Now that we have completed the `BaseMasterPage` class and have our two master pages extending it, our final step is to update the `~/Admin/AddProduct.aspx` and `~/Admin/Products.aspx` pages to refer to this common type. Start by changing the `@MasterType` directive in both pages from:

```
<%@ MasterType VirtualPath="~/Site.master" %>
```

To:

```
<%@ MasterType TypeName="BaseMasterPage" %>
```

Rather than referencing a file path, the `@MasterType` property now references the base type (`BaseMasterPage`). Consequently, the strongly-typed `Master` property used in both pages' code-behind classes is now of type `BaseMasterPage` (instead of type `Site`). With this change in place revisit `~/Admin/Products.aspx`. Previously, this resulted in a casting error because the page is configured to use the `Alternate.master` master page, but the `@MasterType` directive referenced the `Site.master` file. But now the page renders without error. This is because the `Alternate.master` master page can be cast to an object of type `BaseMasterPage` (since it extends it).

There's one small change that needs to be made in `~/Admin/AddProduct.aspx`. The DetailsView control's `ItemInserted` event handler uses both the strongly-typed `Master` property and the loosely-typed `Page.Master` property. We fixed the strongly-typed reference when we updated the `@MasterType` directive, but we still need to update the loosely-typed reference. Replace the following line of code:

```
Site myMasterPage = Page.Master as Site;
```

With the following, which casts `Page.Master` to the base type:

```
BaseMasterPage myMasterPage = Page.Master as BaseMasterPage;
```

Step 4: Determining What Master Page to Bind to the Content Pages

Our `BasePage` class currently sets all content pages' `MasterPageFile` properties to a hard-coded value in the PreInit stage of the page lifecycle. We can update this code to base the master page on some external factor. Perhaps the master page to load depends on the preferences of the currently logged on user. In that case, we'd need to write code in the `OnPreInit` method in `BasePage` that looks up the currently visiting user's master page preferences.

Let's create a web page that allows the user to choose which master page to use - `Site.master` or `Alternate.master` - and save this choice in a Session variable. Start by creating a new web page in the root directory named `ChooseMasterPage.aspx`. When creating this page (or any other content pages henceforth) you don't need to bind it to a master page because the master page is set programmatically in `BasePage`. However, if you do not bind the new page to a master page then the new page's default declarative markup contains a Web Form and other content supplied by the master page. You'll need to manually replace this markup with the appropriate Content controls. For that reason, I find it easier to bind the new ASP.NET page to a master page.

> **NOTE**
> Because `Site.master` and `Alternate.master` have the same set of ContentPlaceHolder controls it doesn't matter what master page you choose when creating the new content page. For consistency, I'd suggest using `Site.master`.

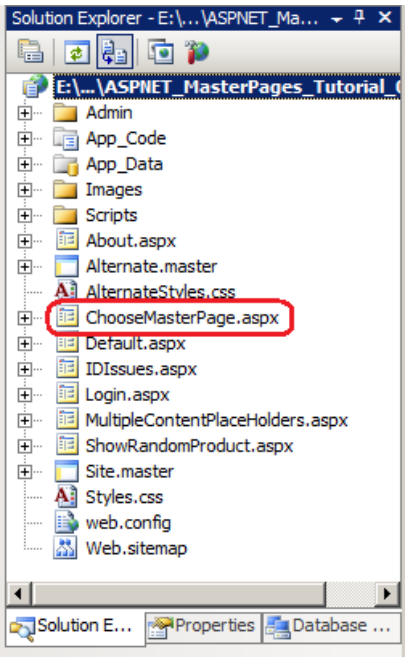


Figure 05: Add a New Content Page to the Website (Click to view full-size image)

Update the `Web.sitemap` file to include an entry for this lesson. Add the following markup beneath the `<siteMapNode>` for the Master Pages and ASP.NET AJAX lesson:

```
<siteMapNode url="~/ChooseMasterPage.aspx" title="Choose a Master Page" />
```

Before adding any content to the `ChooseMasterPage.aspx` page take a moment to update the page's code-behind class so that it derives from `BasePage` (rather than `System.Web.UI.Page`). Next, add a DropDownList control to the page, set its `ID` property to `MasterPageChoice`, and add two ListItems with the `Text` values of "~/Site.master" and "~/Alternate.master".

Add a Button Web control to the page and set its `ID` and `Text` properties to `SaveLayout` and "Save Layout Choice", respectively. At this point your page's declarative markup should look similar to the following:

```
<p>
 Your layout choice:
 <asp:DropDownList ID="MasterPageChoice" runat="server">
 <asp:ListItem>~/Site.master</asp:ListItem>
 <asp:ListItem>~/Alternate.master</asp:ListItem>
 </asp:DropDownList>
</p>
<p>
 <asp:Button ID="SaveLayout" runat="server" Text="Save Layout Choice" />
</p>
```

When the page is first visited we need to display the user's currently selected master page choice. Create a `Page_Load` event handler and add the following code:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        if (Session["MyMasterPage"] != null)
        {
            ListItem li = MasterPageChoice.Items.FindByText(Session["MyMasterPage"].ToString());
            if (li != null)
                li.Selected = true;
        }
    }
}
```

The above code executes only on the first page visit (and not on subsequent postbacks). It first checks to see if the Session variable `MyMasterPage` exists. If it does, it attempts to find the matching ListItem in the `MasterPageChoice` DropDownList. If a matching ListItem is found, its `Selected` property is set to `true`.

We also need code that saves the user's choice into the `MyMasterPage` Session variable. Create an event handler for the `SaveLayout` Button's `Click` event and add the following code:

```
protected void SaveLayout_Click(object sender, EventArgs e)
{
    Session["MyMasterPage"] = MasterPageChoice.SelectedValue;
    Response.Redirect("ChooseMasterPage.aspx");
}
```

NOTE

By the time the `Click` event handler executes on postback, the master page has already been selected. Therefore, the user's drop-down list selection won't be in effect until the next page visit. The `Response.Redirect` forces the browser to re-request `ChooseMasterPage.aspx`.

With the `ChooseMasterPage.aspx` page complete, our final task is to have `BasePage` assign the `MasterPageFile` property based on the value of the `MyMasterPage` Session variable. If the Session variable is not set have `BasePage` default to `Site.master`.

```
protected override void OnPreInit(EventArgs e)
{
    SetMasterPageFile();
    base.OnPreInit(e);
}
protected virtual void SetMasterPageFile()
{
    this.MasterPageFile = GetMasterPageFileFromSession();
}
protected string GetMasterPageFileFromSession()
{
    if (Session["MyMasterPage"] == null)
        return "~/Site.master";
    else
        return Session["MyMasterPage"].ToString();
}
```

NOTE

I moved the code that assigns the `Page` object's `MasterPageFile` property out of the `OnPreInit` event handler and into two separate methods. This first method, `SetMasterPageFile`, assigns the `MasterPageFile` property to the value returned by the second method, `GetMasterPageFileFromSession`. I made the `SetMasterPageFile` method `virtual` so that future classes that extend `BasePage` can optionally override it to implement custom logic, if needed. We'll see an example of overriding `BasePage`'s `SetMasterPageFile` property in the next tutorial.

With this code in place, visit the `ChooseMasterPage.aspx` page. Initially, the `Site.master` master page is selected (see Figure 6), but the user can pick a different master page from the drop-down list.

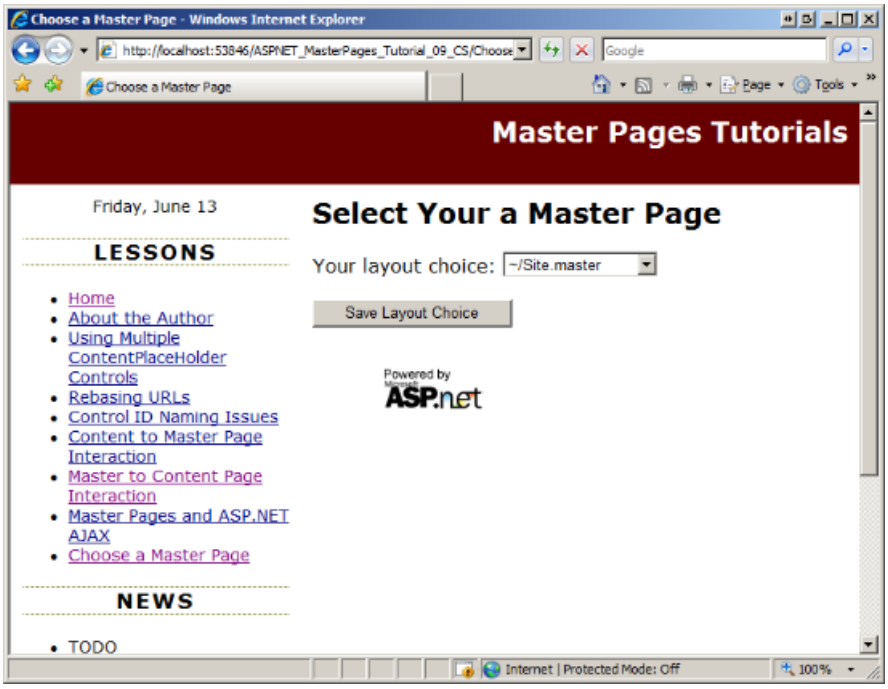


Figure 06: Content Pages are Displayed Using the `Site.master` Master Page (Click to view full-size image)

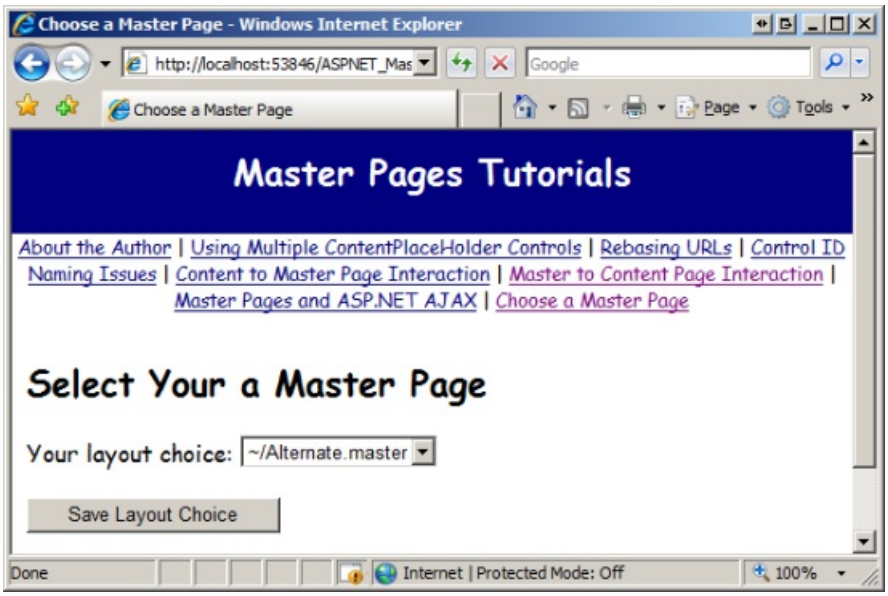


Figure 07: Content Pages are Now Displayed Using the `Alternate.master` Master Page (Click to view full-size image)

Summary

When a content page is visited, its Content controls are fused with its master page's ContentPlaceHolder controls.

The content page's master page is denoted by the `Page` class's `MasterPageFile` property, which is assigned to the `@Page` directive's `MasterPageFile` attribute during the Initialization stage. As this tutorial showed, we can assign a value to the `MasterPageFile` property as long as we do so before the end of the PreInit stage. Being able to programmatically specify the master page opens the door for more advanced scenarios, such as dynamically binding a content page to a master page based on external factors.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Page Lifecycle Diagram
- ASP.NET Page Lifecycle Overview
- ASP.NET Themes and Skins Overview
- Master Pages: Tips, Tricks, and Traps
- Themes in ASP.NET

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Suchi Banerjee. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Nested Master Pages (C#)

2/20/2020 • 26 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> Shows how to nest one master page within another.

Introduction

Over the course of the past nine tutorials we have seen how to implement a site-wide layout with master pages. In a nutshell, master pages allow us, the page developer, to define common markup in the master page along with specific regions that can be customized on a content page-by-content page basis. The ContentPlaceHolder controls in a master page indicate the customizable regions; the customized markup for the ContentPlaceHolder controls are defined in the content page via Content controls.

The master page techniques we've explored thus far are great if you have a single layout used across the entire site. However, many large websites have a site layout that is customized across various sections. For example, consider a health care application used by hospital staff to manage patient information, activities, and billing. There may be three types of web pages in this application:

- Staff member-specific pages where staff members can update availability, view schedules, or request vacation time.
- Patient-specific pages where staff members view or edit information for a specific patient.
- Billing-specific pages where accountants review current claim statuses and financial reports.

Every page might share a common layout, such as a menu across the top and a series of frequently used links along the bottom. But the staff-, patient-, and billing-specific pages may need to customize this generic layout. For example, perhaps all staff-specific pages should include a calendar and task list showing the currently logged on user's availability and daily schedule. Perhaps all patient-specific pages need to show the name, address, and insurance information for the patient whose information is being edited.

It's possible to create such customized layouts by using *nested master pages*. To implement the above scenario, we would start by creating a master page that defined the site-wide layout, the menu and footer content, with ContentPlaceHolders defining the customizable regions. We would then create three nested master pages, one for each type of web page. Each nested master page would define the content among the type of content pages that use the master page. In other words, the nested master page for patient-specific content pages would include markup and programmatic logic for displaying information about the patient being edited. When creating a new patient-specific page we would bind it to this nested master page.

This tutorial starts by highlighting the benefits of nested master pages. It then shows how to create and use nested master pages.

> **NOTE**
>
> Nested master pages have been possible since version 2.0 of the .NET Framework. However, Visual Studio 2005 did not include design-time support for nested master pages. The good news is that Visual Studio 2008 offers a rich design-time experience for nested master pages. If you are interested in using nested master pages but are still using Visual Studio 2005, check out Scott Guthrie's blog entry, Tips for Nested Master Pages in VS 2005 Design-Time.

The Benefits of Nested Master Pages

Many websites have an overarching site design as well as more customized designs specific to certain types of pages. For instance, in our demo web application we have created a rudimentary Administration section (the pages in the `~/Admin` folder). Currently the web pages in the `~/Admin` folder use the same master page as those pages not in the administration section (namely, `Site.master` or `Alternate.master`, depending on the user's selection).

> **NOTE**
> For now, pretend that our site has just one master page, `Site.master`. We'll address using nested master pages with two (or more) master pages starting with "Using a Nested Master Page for the Administration Section" later in this tutorial.

Imagine that we were asked to customize the layout of the Administration pages to include additional information or links that would not otherwise be present in other pages in the site. There are four techniques to implement this requirement:

1. Manually add the Administration-specific information and links to every content page in the `~/Admin` folder.
2. Update the `Site.master` master page to include the Administration section-specific information and links, and then add code to the master page to show or hide these sections based on whether one of the Administration pages is being visited.
3. Create a new master page specifically for the Administration section, copy over the markup from `Site.master`, add the Administration section-specific information and links, and then update the content pages in the `~/Admin` folder to use this new master page.
4. Create a nested master page that binds to `Site.master` and have the content pages in the `~/Admin` folder use this new nested master page. This nested master page would include just the additional information and links specific to the Administration pages and would not need to repeat the markup already defined in `Site.master`.

The first option is the least palatable. The whole point of using master pages is to move away from having to manually copy and paste common markup to new ASP.NET pages. The second option is acceptable, but makes the application less maintainable as it bulks up the master pages with markup that is only occasionally displayed and requires developers editing the master page to work around this markup and to have to remember when, exactly, certain markup is displayed versus when it is hidden. This approach would be less tenable as customizations from more and more types of web pages needed to be accommodated by this single master page.

The third option removes the clutter and complexity issues the surfaced with the second option. However, option three's main drawback is that it requires us to copy and paste the common layout from `Site.master` to the new Administration section-specific master page. If we later decide to change the site-wide layout we have to remember to change it in two places.

The fourth option, nested master pages, give us the best of the second and third options. The site-wide layout information is maintained in one file - the top-level master page - while the content specific to particular regions is separated out into different files.

This tutorial starts with a look at creating and using a simple nested master page. We create a brand new top-level master page, two nested master pages, and two content pages. Starting with "Using a Nested Master Page for the Administration Section," we look at updating our existing master page architecture to include the use of nested master pages. Specifically, we create a nested master page and use it to include additional custom content for the content pages in the `~/Admin` folder.

Step 1: Creating a Simple Top-Level Master Page

Creating a nested master based on one of the existing master pages and then updating an existing content page to use this new nested master page instead of the top-level master page entails some complexity because the

existing content pages already expect certain ContentPlaceHolder controls defined in the top-level master page. Therefore, the nested master page must also include the same ContentPlaceHolder controls with the same names. Furthermore, our particular demo application has two master pages (`Site.master` and `Alternate.master`) that are dynamically assigned to a content page based on a user's preferences, which further adds to this complexity. We will look at updating the existing application to use nested master pages later in this tutorial, but let's first focus on a simple nested master pages example.

Create a new folder named `NestedMasterPages` and then add a new master page file to that folder named `Simple.master`. (See Figure 1 for a screen shot of the Solution Explorer after this folder and file have been added.) Drag the `AlternateStyles.css` style sheet file from the Solution Explorer onto the Designer. This adds a `<link>` element to the style sheet file in the `<head>` element, after which the master page's `<head>` element's markup should look like:

```
<head runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
 <link href="../AlternateStyles.css" rel="stylesheet" type="text/css" />
</head>
```

Next, add the following markup within the Web Form of `Simple.master`:

```
<div id="topContent">
 <asp:HyperLink ID="lnkHome" runat="server"
 NavigateUrl="~/NestedMasterPages/Default.aspx"
 Text="Nested Master Pages Tutorial (Simple)" />
</div>
<div id="mainContent">
 <asp:ContentPlaceHolder id="MainContent" runat="server">
 </asp:ContentPlaceHolder>
</div>
```

This markup displays a link titled "Nested Master Pages (Simple)" at the top of the page in a large white font on a navy background. Beneath that is the `MainContent` ContentPlaceHolder. Figure 1 shows the `Simple.master` master page when loaded in the Visual Studio Designer.

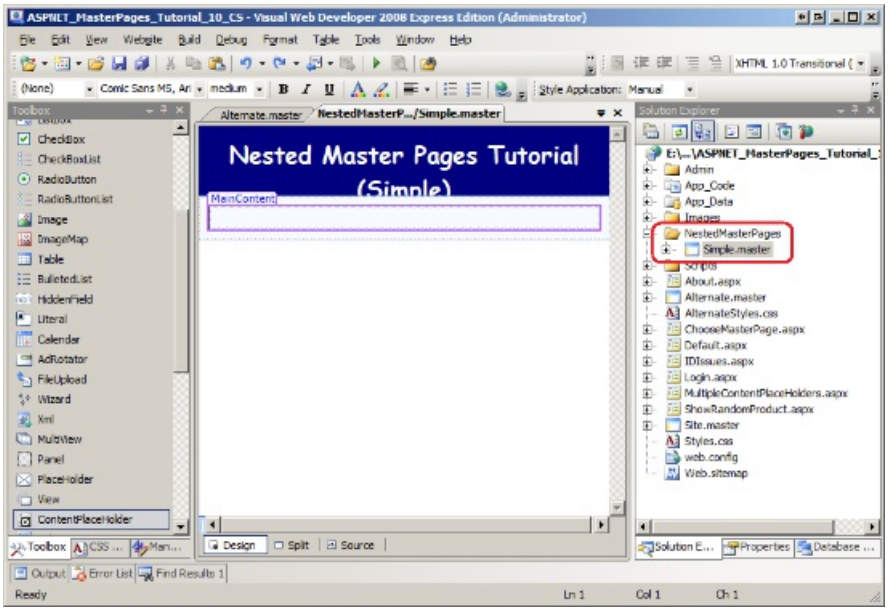


Figure 01: The Nested Master Page Defines Content Specific to the Pages in the Administration Section (Click to view full-size image)

Step 2: Creating a Simple Nested Master Page

`Simple.master` contains two ContentPlaceHolder controls: the `MainContent` ContentPlaceHolder we added within the Web Form along with the `head` ContentPlaceHolder in the `<head>` element. If we were to create a content page and bind it to `Simple.master` the content page would have two Content controls referencing the two ContentPlaceHolders. Similarly, if we create a nested master page and bind it to `Simple.master` then the nested master page will have two Content controls.

Let's add a new nested master page to the `NestedMasterPages` folder named `SimpleNested.master`. Right-click on the `NestedMasterPages` folder and choose Add New Item. This brings up the Add New Item dialog box shown in Figure 2. Select the Master Page template type and type in the name of the new master page. To indicate that the new master page should be a nested master page, check the "Select master page" checkbox.

Next, click the Add button. This will display the same Select a Master Page dialog box you see when binding a content page to a master page (see Figure 3). Choose the `Simple.master` master page in the `NestedMasterPages` folder and click OK.

> **NOTE**
>
> If you created your ASP.NET website using the Web Application Project model instead of the Web Site Project model you will not see the "Select master page" checkbox in the Add New Item dialog box shown in Figure 2. To create a nested master page when using the Web Application Project model you must choose the Nested Master Page template (instead of the Master Page template). After selecting the Nested Master Page template and clicking Add, the same Select a Master Page dialog box shown in Figure 3 will appear.

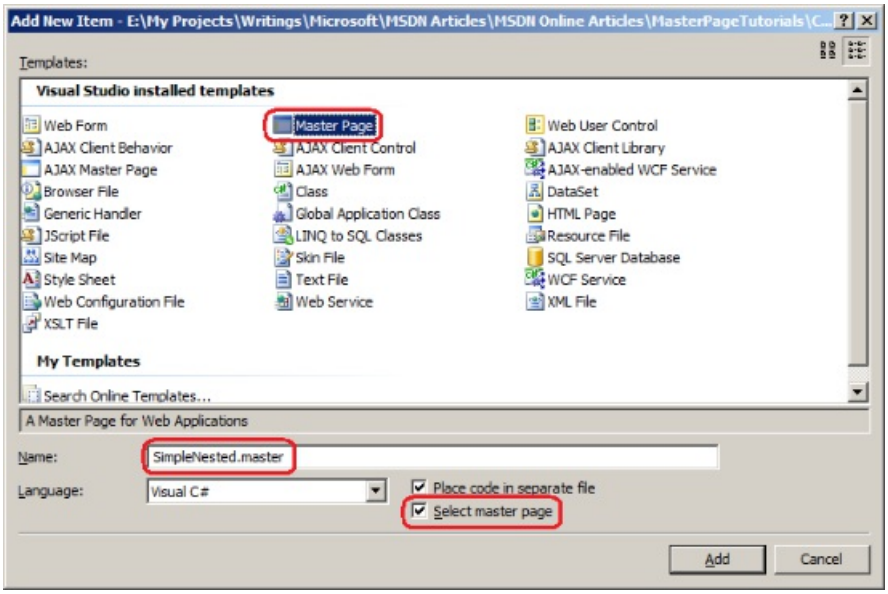


Figure 02: Check the "Select master page" Checkbox to Add a Nested Master Page (Click to view full-size image)

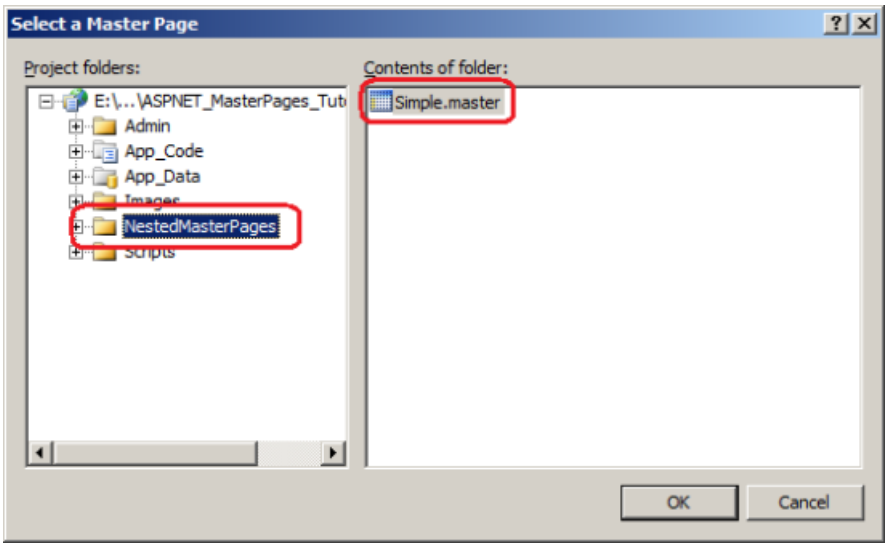


Figure 03: Bind the Nested Master Page to the `Simple.master` Master Page (Click to view full-size image)

The nested master page's declarative markup, shown below, contains two Content controls referencing the top-level master page's two ContentPlaceHolder controls.

```
<%@ Master Language="C#" MasterPageFile="~/NestedMasterPages/Simple.master" AutoEventWireup="false"
CodeFile="SimpleNested.master.cs" Inherits="NestedMasterPages_SimpleNested" %>
 <asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
 </asp:Content>
 <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
 </asp:Content>
```

Except for the `<%@ Master %>` directive, the nested master page's initial declarative markup is identical to the markup that is initially generated when binding a content page to the same top-level master page. Like a content page's `<%@ Page %>` directive, the `<%@ Master %>` directive here includes a `MasterPageFile` attribute that specifies the nested master page's parent master page. The main difference between the nested master page and a content page bound to the same top-level master page is that the nested master page can include ContentPlaceHolder controls. The nested master page's ContentPlaceHolder controls define the regions where the content pages can customize the markup.

Update this nested master page so that it displays the text "Hello, from SimpleNested!" in the Content control that corresponds to the `MainContent` ContentPlaceHolder control.

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
 <p>Hello, from SimpleNested!</p>
</asp:Content>
```

After making this addition, save the nested master page and then add a new content page to the `NestedMasterPages` folder named `Default.aspx`, and bind it to the `SimpleNested.master` master page. Upon adding this page you may be surprised to see that it contains no Content controls (see Figure 4)! A content page can only access its *parent* master page's ContentPlaceHolders. `SimpleNested.master` does not contain any ContentPlaceHolder controls; therefore, any content page bound to this master page cannot contain any Content controls.

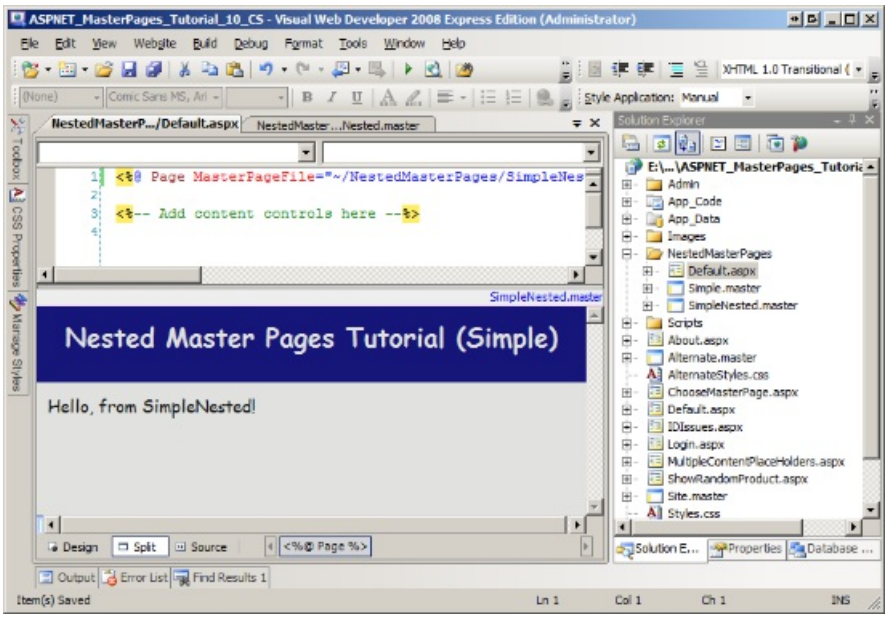


Figure 04: The New Content Page Contains No Content Controls (Click to view full-size image)

What we need to do is update the nested master page (`SimpleNested.master`) to include ContentPlaceHolder controls. Typically you'll want your nested master pages to include a ContentPlaceHolder for each ContentPlaceHolder defined by its parent master page, thereby allowing its child master page or content page to work with any of the top-level master page's ContentPlaceHolder controls.

Update the `SimpleNested.master` master page to include a ContentPlaceHolder in its two Content controls. Give the ContentPlaceHolder controls the same name as the ContentPlaceHolder control their Content control refers to. That is, add a ContentPlaceHolder control named `MainContent` to the Content control in `SimpleNested.master` that references the `MainContent` ContentPlaceHolder in `Simple.master`. Do the same thing in the Content control that references the `head` ContentPlaceHolder.

> **NOTE**
>
> While I recommend naming the ContentPlaceHolder controls in the nested master page the same as the ContentPlaceHolders in the top-level master page, this naming symmetry is not required. You can give the ContentPlaceHolder controls in your nested master page any name you like. However, I find it easier to remember what ContentPlaceHolders correspond with what regions of the page if my top-level master page and nested master pages use the same names.

After making these additions your `SimpleNested.master` master page's declarative markup should look similar to the following:

```
<%@ Master Language="C#" MasterPageFile="~/NestedMasterPages/Simple.master"AutoEventWireup="false"
CodeFile="SimpleNested.master.cs" Inherits="NestedMasterPages_SimpleNested" %>
 <asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
 <asp:ContentPlaceHolder ID="head" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
 <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
 <p>Hello, from SimpleNested!</p>
 <asp:ContentPlaceHolder ID="MainContent" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
```

Delete the `Default.aspx` content page we just created and then re-add it, binding it to the `SimpleNested.master` master page. This time Visual Studio adds two Content controls to the `Default.aspx`, referencing the ContentPlaceHolders now defined in `SimpleNested.master` (see Figure 6). Add the text, "Hello, from Default.aspx!"

in the Content control that referenced `MainContent`.

Figure 5 shows the three entities involved here - `Simple.master`, `SimpleNested.master`, and `Default.aspx` - and how they relate to one another. As the diagram shows, the nested master page implements Content controls for its parent's ContentPlaceHolder. If these regions need to be accessible to the content page, the nested master page must add its own ContentPlaceHolders to the Content controls.

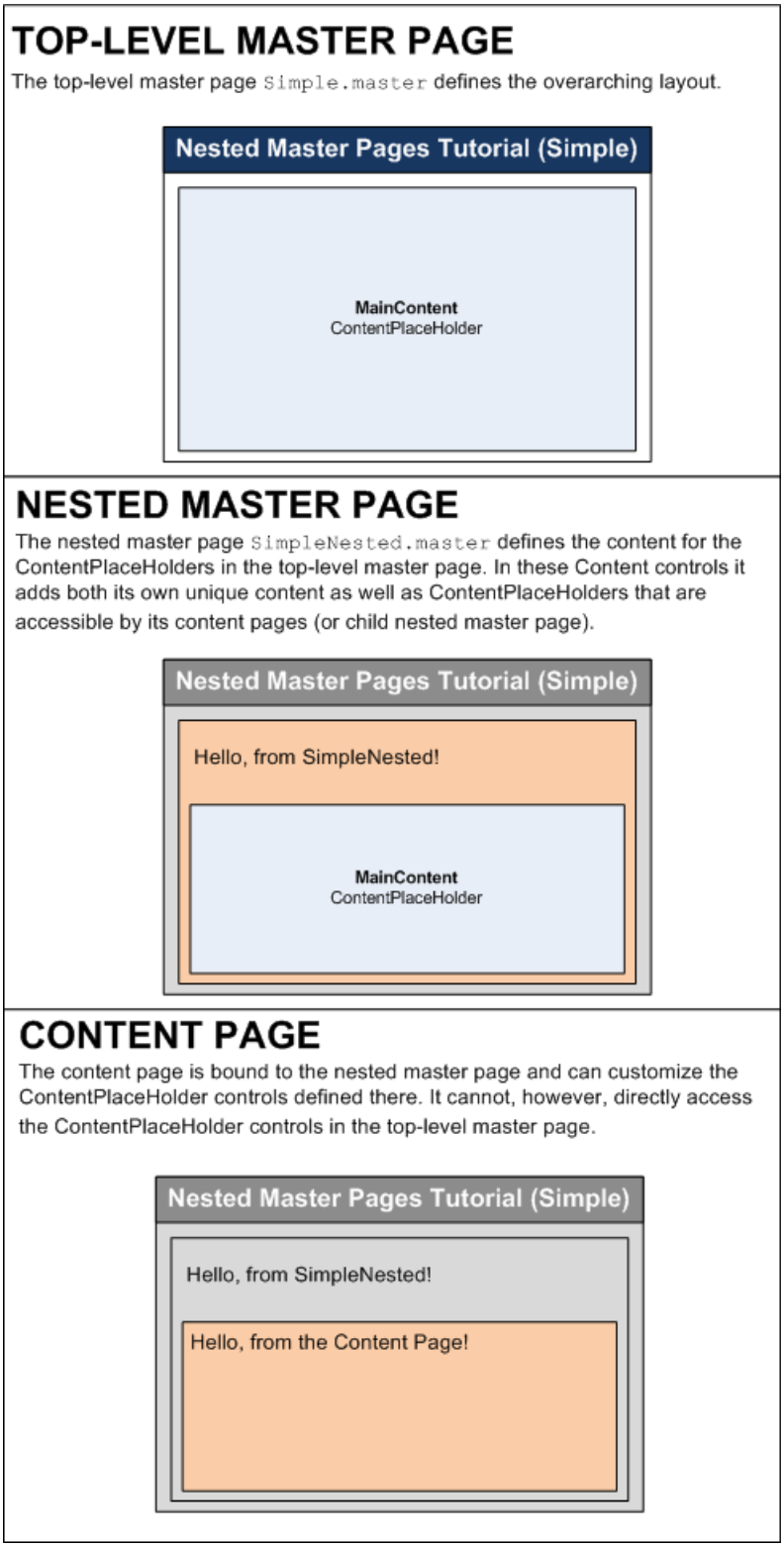


Figure 05: The Top-Level and Nested Master Pages Dictate the Content Page's Layout (Click to view full-size image)

This behavior illustrates how a content page or master page is only cognizant of its parent master page. This behavior is also indicated by the Visual Studio Designer. Figure 6 shows the Designer for `Default.aspx`. While the Designer clearly shows what regions are editable from the content page and what portions aren't, it doesn't disambiguate what non-editable regions are from the nested master page and what regions are from the top-

level master page.

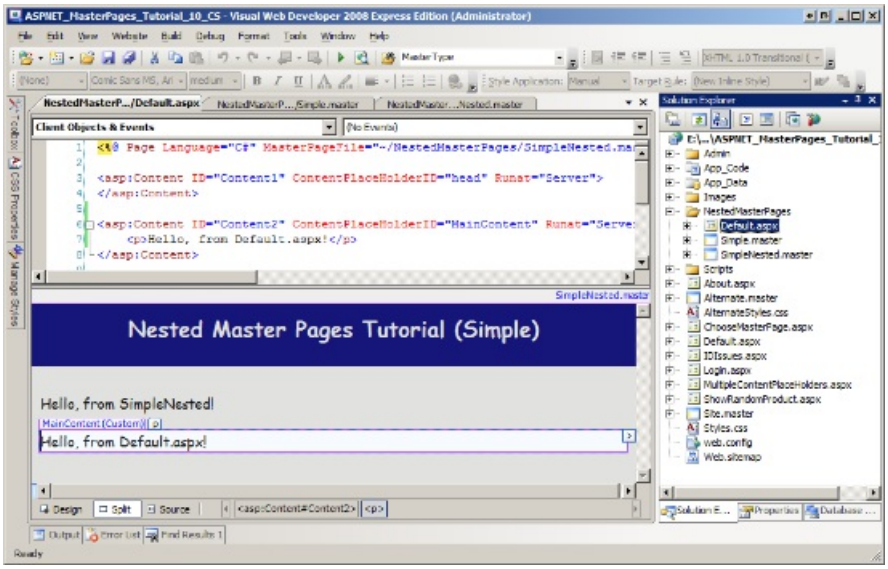


Figure 06: The Content Page Now Includes Content Controls for the Nested Master Page's ContentPlaceHolders (Click to view full-size image)

Step 3: Adding a Second Simple Nested Master Page

The benefit of nested master pages is more evident when there are multiple nested master pages. To illustrate this benefit, create another nested master page in the `NestedMasterPages` folder; name this new nested master page `SimpleNestedAlternate.master` and bind it to the `Simple.master` master page. Add ContentPlaceHolder controls in the nested master page's two Content controls like we did in Step 2. Also add the text, "Hello, from SimpleNestedAlternate!" in the Content control that corresponds to the top-level master page's `MainContent` ContentPlaceHolder. After making these changes your new nested master page's declarative markup should look similar to the following:

```
<%@ Master Language="C#" MasterPageFile="~/NestedMasterPages/Simple.master" AutoEventWireup="false"
CodeFile="SimpleNestedAlternate.master.cs" Inherits="NestedMasterPages_SimpleNestedAlternate" %>
 <asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
 <asp:ContentPlaceHolder ID="head" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
 <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
 <p>Hello, from SimpleNestedAlternate!</p>
 <asp:ContentPlaceHolder ID="MainContent" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
```

Create a content page named `Alternate.aspx` in the `NestedMasterPages` folder and bind it to the `SimpleNestedAlternate.master` nested master page. Add the text, "Hello, from Alternate!" in the Content control that corresponds to `MainContent`. Figure 7 shows `Alternate.aspx` when viewed through the Visual Studio Designer.

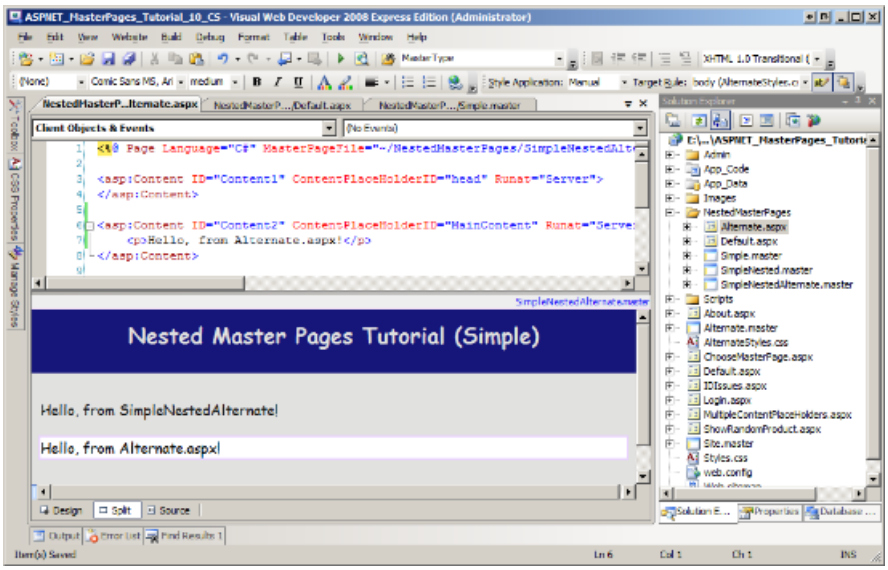


Figure 07: `Alternate.aspx` is Bound to the `SimpleNestedAlternate.master` Master Page (Click to view full-size image)

Compare the Designer in Figure 7 to the Designer in Figure 6. Both content pages share the same layout defined in the top-level master page (`Simple.master`), namely the "Nested Master Pages Tutorial (Simple)" title. Yet both have distinct content defined in their parent master pages - the text "Hello, from SimpleNested!" in Figure 6 and "Hello, from SimpleNestedAlternate!" in Figure 7. Granted, these differences here are trivial, but you could extend this example to include more meaningful differences. For instance, the `SimpleNested.master` page might include a menu with options specific to its content pages, whereas `SimpleNestedAlternate.master` might have information pertinent to the content pages that bind to it.

Now, imagine that we needed to make a change to the overarching site layout. For example, imagine that we wanted to add a list of common links to all content pages. To accomplish this we update the top-level master page, `Simple.master` . Any changes there are immediately reflected in its nested master pages and, by extension, their content pages.

To demonstrate the ease with which we can change the overarching site layout, open the `Simple.master` master page and add the following markup between the `topContent` and `mainContent` `<div>` elements:

```
<div id="navContent">
 <asp:HyperLink ID="lnkDefault" runat="server"
 NavigateUrl="~/NestedMasterPages/Default.aspx"
 Text="Nested Master Page Example 1" />
 |
 <asp:HyperLink ID="lnkAlternate" runat="server"
 NavigateUrl="~/NestedMasterPages/Alternate.aspx"
 Text="Nested Master Page Example 2" />
</div>
```

This adds two links to the top of every page that binds to `Simple.master` , `SimpleNested.master` , or `SimpleNestedAlternate.master` ; these changes apply to all nested master pages and their content pages immediately. Figure 8 shows `Alternate.aspx` when viewed through a browser. Note the addition of the links at the top of the page (compared to Figure 7).

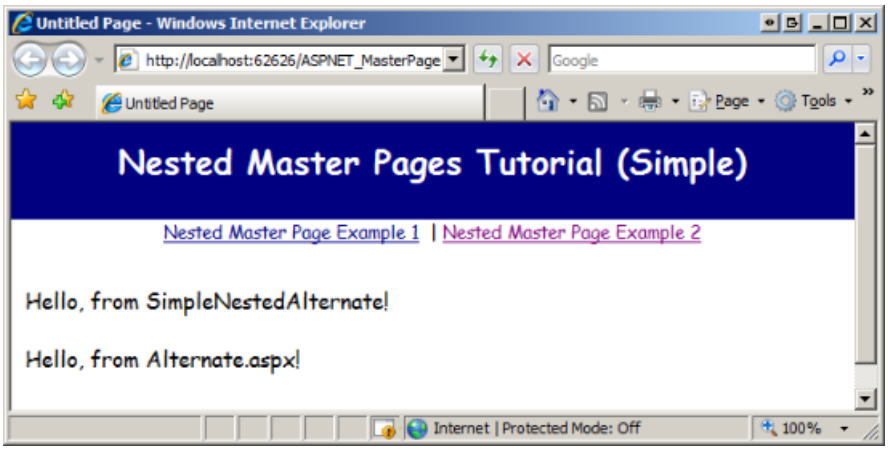


Figure 08: Changed to the Top-Level Master Page are Immediately Reflected in its Nested Master Pages and Their Content Pages (Click to view full-size image)

Using a Nested Master Page for the Administration Section

At this point we have looked at the advantages of nested master pages and have seen how to create and use them in an ASP.NET application. The examples in Steps 1, 2, and 3, however, involved creating a new top-level master page, new nested master pages, and new content pages. What about adding a new nested master page to a website with an existing top-level master page and content pages?

Integrating a nested master page into an existing website and associating it with existing content pages requires a bit more effort than starting from scratch. Steps 4, 5, 6, and 7 explore these challenges as we augment our demo application to include a new nested master page named `AdminNested.master` that contains instructions for the administrator and is used by the ASP.NET pages in the `~/Admin` folder.

Integrating a nested master page into our demo application introduces the following hurdles:

- The existing content pages in the `~/Admin` folder have certain expectations from their master page. For starters, they expect certain ContentPlaceHolder controls to be present. Furthermore, the `~/Admin/AddProduct.aspx` and `~/Admin/Products.aspx` pages call the master page's public `RefreshRecentProductsGrid` method, set its `GridMessageText` property, or have an event handler for its `PricesDoubled` event. Consequently, our nested master page must provide the same ContentPlaceHolders and public members.
- In the preceding tutorial we enhanced the `BasePage` class to dynamically set the `Page` object's `MasterPageFile` property based on a Session variable. How to we support dynamic master pages when using nested master pages?

These two challenges will surface as we build the nested master page and use it from our existing content pages. We'll investigate and surmount these issues as they arise.

Step 4: Creating the Nested Master Page

Our first task is to create the nested master page to be used by the pages in the Administration section. As we saw in Step 2, when adding a new nested master page we need to specify the nested master page's parent master page. But we have two top-level master pages: `Site.master` and `Alternate.master`. Recall that we created `Alternate.master` in the preceding tutorial and wrote code in the `BasePage` class that set the Page object's `MasterPageFile` property at runtime to either `Site.master` or `Alternate.master` depending on the value of the `MyMasterPage` Session variable.

How do we configure our nested master page so that it uses the appropriate top-level master page? We have two options:

- Create two nested master pages, `AdminNestedSite.master` and `AdminNestedAlternate.master`, and bind them to the top-level master pages `Site.master` and `Alternate.master`, respectively. In `BasePage`, then, we'd set the `Page` object's `MasterPageFile` to the appropriate nested master page.
- Create a single nested master page and have the content pages use this particular master page. Then, at runtime, we'd need to set the nested master page's `MasterPageFile` property to the appropriate top-level master page at runtime. (As you might have figured out by now, master pages also have a `MasterPageFile` property.)

Let's use the second option. Create a single nested master page file in the `~/Admin` folder named `AdminNested.master`. Because both `Site.master` and `Alternate.master` have the same set of ContentPlaceHolder controls, it doesn't matter what master page you bind it to, although I encourage you to bind it to `Site.master` for consistency's sake.

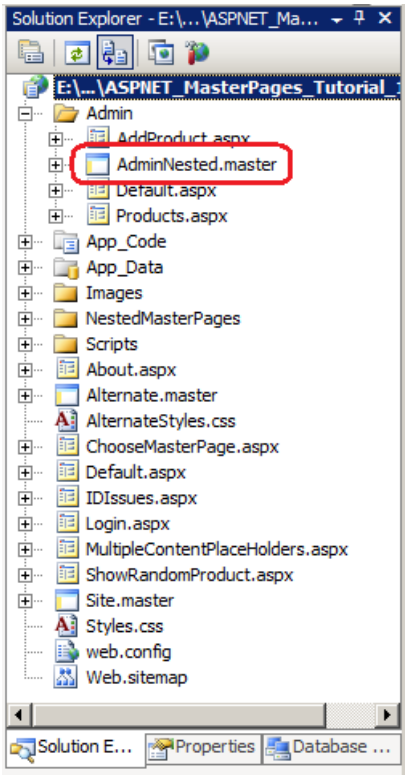


Figure 09: Add a Nested Master Page to the `~/Admin` Folder. (Click to view full-size image)

Because the nested master page is bound to a master page with four ContentPlaceHolder controls, Visual Studio adds four Content controls to the new nested master page file's initial markup. Like we did in Steps 2 and 3, add a ContentPlaceHolder control in each Content control, giving it the same name as the top-level master page's ContentPlaceHolder control. Also add the following markup to the Content control that corresponds to the `MainContent` ContentPlaceHolder:

```
<div class="instructions">
 <b>Administration Instructions:</b>
 <br />
 The pages in the Administration section allow you, the Administrator, to
 add new products and view existing products.
</div>
```

Next, define the `instructions` CSS class in the `Styles.css` and `AlternateStyles.css` CSS files. The following CSS rules cause HTML elements styled with the `instructions` class to be displayed with a light yellow background color and a black, solid border:

```
.instructions
{
 padding: 6px;
 border: dashed 1px black;
 background-color: #ffb;
 margin-bottom: 10px;
}
```

Because this markup has been added to the nested master page, it will only appear in those pages that use this nested master page (namely, the pages in the Administration section).

After making these additions to your nested master page, its declarative markup should look similar to the following:

```
<%@ Master Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="false"
CodeFile="AdminNested.master.cs" Inherits="Admin_AdminNested" %>
 <asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
 <asp:ContentPlaceHolder ID="head" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
 <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
 <div class="instructions">
 <b>Administration Instructions:</b>
 <br />
 The pages in the Administration section allow you, the Administrator, to
 add new products and view existing products.
 </div>
 <asp:ContentPlaceHolder ID="MainContent" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
 <asp:Content ID="Content3" ContentPlaceHolderID="QuickLoginUI" Runat="Server">
 <asp:ContentPlaceHolder ID="QuickLoginUI" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
 <asp:Content ID="Content4" ContentPlaceHolderID="LeftColumnContent" Runat="Server">
 <asp:ContentPlaceHolder ID="LeftColumnContent" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
```

Note that each Content control has a ContentPlaceHolder control and that the ContentPlaceHolder controls' `ID` properties are assigned the same values as the corresponding ContentPlaceHolder controls in the top-level master page. Moreover, the Administration section-specific markup appears in the `MainContent` ContentPlaceHolder.

Figure 10 shows the `AdminNested.master` nested master page when viewed through Visual Studio's Designer. You can see the instructions in the yellow box at the top of the `MainContent` Content control.

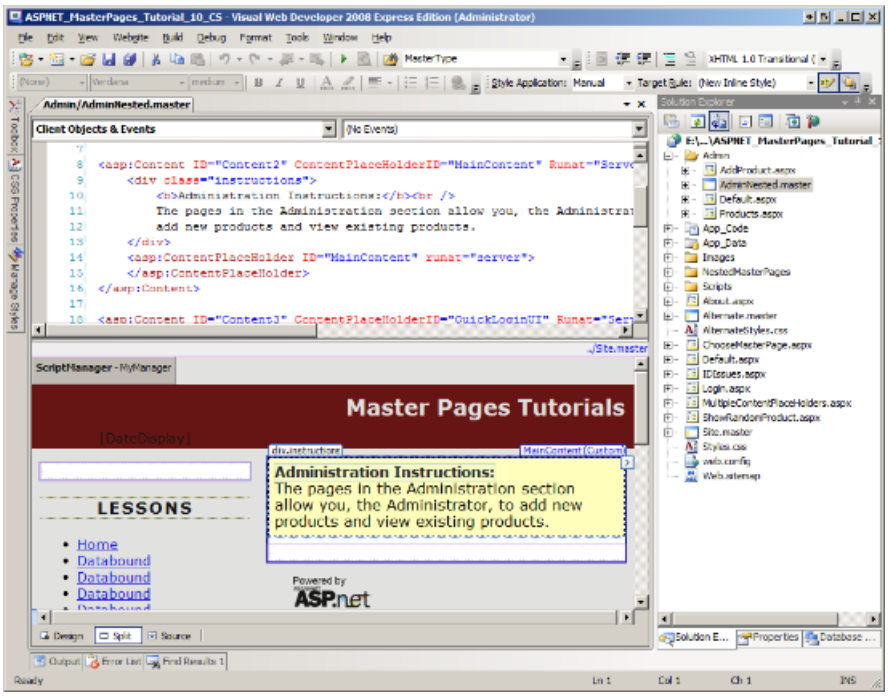


Figure 10: The Nested Master Page Extends the Top-Level Master Page to Include Instructions for the Administrator. (Click to view full-size image)

Step 5: Updating the Existing Content Pages to Use the New Nested Master Page

Anytime we add a new content page to the Administration section we need to bind it to the `AdminNested.master` master page we just created. But what about the existing content pages? Currently, all content pages in the site derive from the `BasePage` class, which programmatically sets the content page's master page at runtime. This is not the behavior we want for the content pages in the Administration section. Instead, we want these content pages to always use the `AdminNested.master` page. It will be the responsibility of the nested master page to choose the right top-level content page at runtime.

To best way to achieve this desired behavior is to create a new custom base page class named `AdminBasePage` that extends the `BasePage` class. `AdminBasePage` can then override the `SetMasterPageFile` and set the `Page` object's `MasterPageFile` to the hard-coded value "~/Admin/AdminNested.master". In this way, any page that derives from `AdminBasePage` will use `AdminNested.master`, whereas any page that derives from `BasePage` will have its `MasterPageFile` property set dynamically to either "~/Site.master" or "~/Alternate.master" based on the value of the `MyMasterPage` Session variable.

Start by adding a new class file to the `App_Code` folder named `AdminBasePage.cs`. Have `AdminBasePage` extend `BasePage` and then override the `SetMasterPageFile` method. In that method assign the `MasterPageFile` the value "~/Admin/AdminNested.master". After making these changes your class file should look similar to the following:

```
public class AdminBasePage : BasePage
{
    protected override void SetMasterPageFile()
    {
        this.MasterPageFile = "~/Admin/AdminNested.master";
    }
}
```

We now need to have the existing content pages in the Administration section derive from `AdminBasePage` instead of `BasePage`. Go to the code-behind class file for each content page in the `~/Admin` folder and make this change. For example, in `~/Admin/Default.aspx` you'd change the code-behind class declaration from:

```
public partial class Admin_Default : BasePage
```

To:

```
public partial class Admin_Default : AdminBasePage
```

Figure 11 depicts how the top-level master page (`Site.master` or `Alternate.master`), the nested master page (`AdminNested.master`), and the Administration section content pages relate to one another.

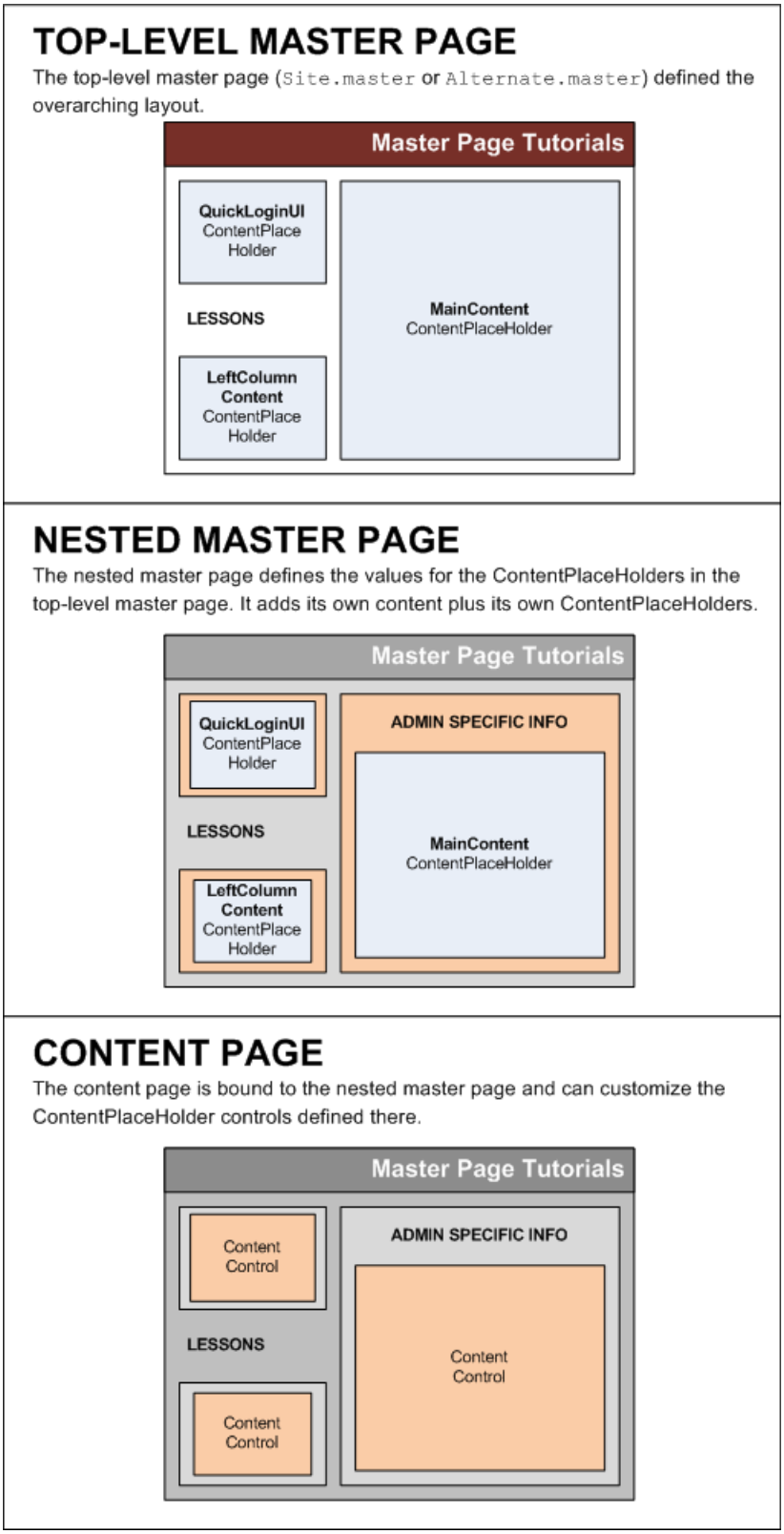


Figure 11: The Nested Master Page Defines Content Specific to the Pages in the Administration Section (Click to

view full-size image)

Step 6: Mirroring the Master Page's Public Methods and Properties

Recall that the `~/Admin/AddProduct.aspx` and `~/Admin/Products.aspx` pages interact programmatically with the master page: `~/Admin/AddProduct.aspx` calls the master page's public `RefreshRecentProductsGrid` method and sets its `GridMessageText` property; `~/Admin/Products.aspx` has an event handler for the `PricesDoubled` event. In the preceding tutorial we created an abstract `BaseMasterPage` class that defined these public members.

The `~/Admin/AddProduct.aspx` and `~/Admin/Products.aspx` pages assume that their master page derives from the `BaseMasterPage` class. The `AdminNested.master` page, however, currently extends the `System.Web.UI.MasterPage` class. As a result, when visiting `~/Admin/Products.aspx` an `InvalidCastException` is thrown with the message: "Unable to cast object of type 'ASP.admin_adminnested_master' to type 'BaseMasterPage'."

To fix this we need to have the `AdminNested.master` code-behind class extend `BaseMasterPage`. Update the nested master page's code-behind class declaration from:

```
public partial class Admin_AdminNested : System.Web.UI.MasterPage
```

To:

```
public partial class Admin_AdminNested : BaseMasterPage
```

We're not done yet. Because the `BaseMasterPage` class is abstract, we need to override the `abstract` members, `RefreshRecentProductsGrid` and `GridMessageText`. These members are used by the top-level master pages to update their user interfaces. (Actually, only the `Site.master` master page uses these methods, although both top-level master pages implement these methods, since both extend `BaseMasterPage`.)

While we need to implement these members in `AdminNested.master`, all these implementations need to do is simply call the same member in the top-level master page used by the nested master page. For instance, when a content page in the Administration section calls the nested master page's `RefreshRecentProductsGrid` method, all the nested master page needs to do is, in turn, call `Site.master` or `Alternate.master`'s `RefreshRecentProductsGrid` method.

To achieve this, start by adding the following `@MasterType` directive to the top of `AdminNested.master`:

```
<%@ MasterType TypeName="BaseMasterPage" %>
```

Recall that the `@MasterType` directive adds a strongly-typed property to the code-behind class named `Master`. Then override the `RefreshRecentProductsGrid` and `GridMessageText` members and simply delegate the call to the `Master`'s corresponding method:

```
public partial class Admin_AdminNested : BaseMasterPage
{
    public override void RefreshRecentProductsGrid()
    {
        Master.RefreshRecentProductsGrid();
    }
    public override string GridMessageText
    {
        get
        {
            return Master.GridMessageText;
        }
        set
        {
            Master.GridMessageText = value;
        }
    }
}
```

With this code in place, you should be able to visit and use the content pages in the Administration section. Figure 12 shows the `~/Admin/Products.aspx` page when viewed through a browser. As you can see, the page includes the Administration Instructions box, which is defined in the nested master page.

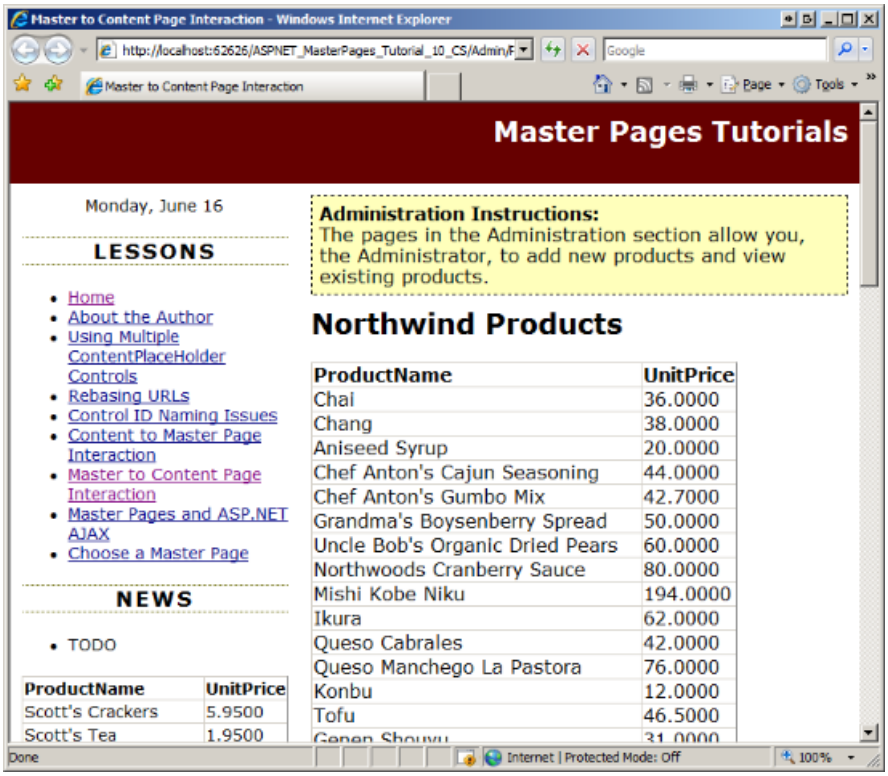


Figure 12: The Content Pages in the Administration Section Include Instructions at the Top of Each Page (Click to view full-size image)

Step 7: Using the Appropriate Top-Level Master Page at Runtime

While all the content pages in the Administration section are fully functional, they all use the same top-level master page and ignore the master page selected by the user at `ChooseMasterPage.aspx`. This behavior is due to the fact that the nested master page has its `MasterPageFile` property statically set to `Site.master` in its `<%@ Master %>` directive.

To use the top-level master page selected by the end user we need to set the `AdminNested.master`'s `MasterPageFile` property to the value in the `MyMasterPage` Session variable. Because we set the content pages' `MasterPageFile` properties in `BasePage`, you may think that we would set the nested master page's

`MasterPageFile` property in `BaseMasterPage` or in the `AdminNested.master`'s code-behind class. This won't work, however, because we need to have set the `MasterPageFile` property by the end of the PreInit stage. The earliest time that we can programmatically tap into the page lifecycle from a master page is the Init stage (which occurs after the PreInit stage).

Therefore, we need to set the nested master page's `MasterPageFile` property from the content pages. The only content pages that use the `AdminNested.master` master page derive from `AdminBasePage`. Therefore, we can put this logic there. In Step 5 we overrode the `SetMasterPageFile` method, setting the `Page` object's `MasterPageFile` property to "~/Admin/AdminNested.master". Update `SetMasterPageFile` to also set the master page's `MasterPageFile` property to the result stored in Session:

```
public class AdminBasePage : BasePage
{
    protected override void SetMasterPageFile()
    {
        this.MasterPageFile = "~/Admin/AdminNested.master";
        Page.Master.MasterPageFile = base.GetMasterPageFileFromSession();
    }
}
```

The `GetMasterPageFileFromSession` method, which we added to the `BasePage` class in the preceding tutorial, returns the appropriate master page file path based on the Session variable value.

With this change in place, the user's master page selection carries over to the Administration section. Figure 13 shows the same page as Figure 12, but after the user has changed their master page selection to `Alternate.master`.

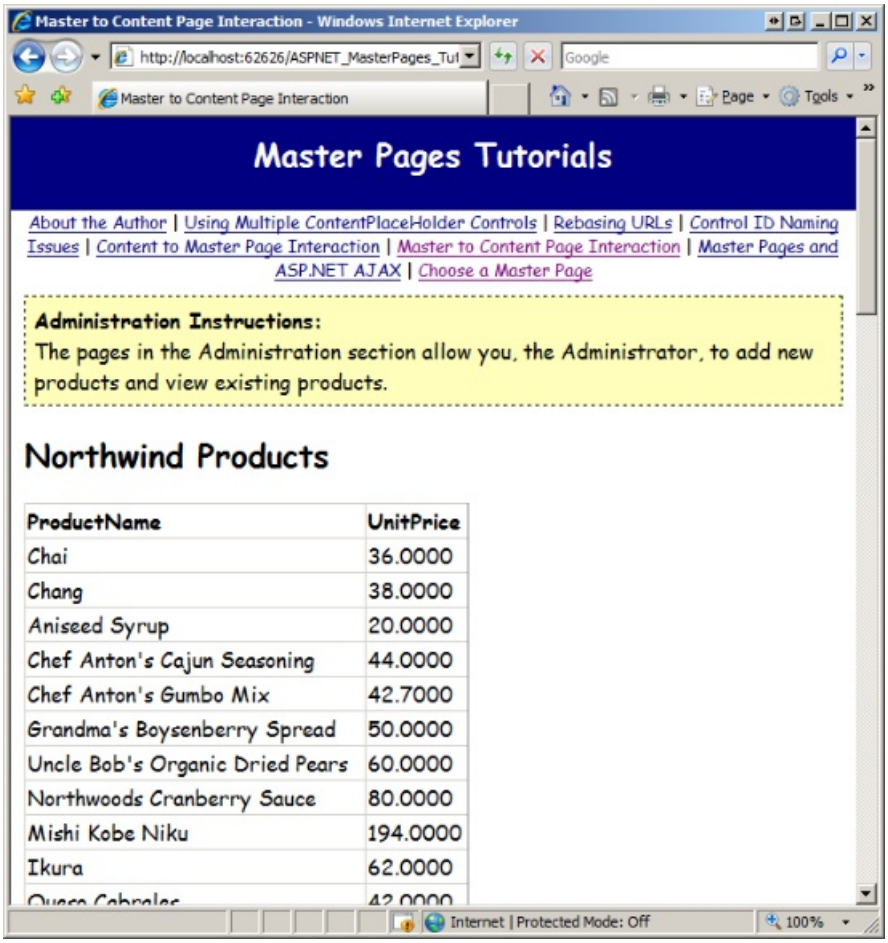


Figure 13: The Nested Administration Page Uses the Top-Level Master Page Selected by the User (Click to view full-size image)

Summary

Much like how content pages can bind to a master page, it is possible to create nested master pages by having a child master page bind to a parent master page. The child master page may define Content controls for each of its parent's ContentPlaceHolders; it can then add its own ContentPlaceHolder controls (as well as other markup) to these Content controls. Nested master pages are quite useful in large web applications where all pages share an overarching look and feel, yet certain sections of the site require unique customizations.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Nested ASP.NET Master Pages
- Tips for Nested Master Pages and VS 2005 Design-Time
- VS 2008 Nested Master Page Support

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Creating a Site-Wide Layout Using Master Pages (VB)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

This tutorial will show master page basics. Namely, what are master pages, how does one create a master page, what are content place holders, how does one create an ASP.NET page that uses a master page, how modifying the master page is automatically reflected in its associated content pages, and so on.

Introduction

One attribute of a well-designed website is a consistent site-wide page layout. Take the www.asp.net website, for example. At the time of this writing, every page has the same content at the top and bottom of the page. As Figure 1 shows, the very top of each page displays a gray bar with a list of Microsoft Communities. Beneath that is the site logo, the list of languages into which the site has been translated, and the core sections: Home, Get Started, Learn, Downloads, and so forth. Likewise, the bottom of the page includes information about advertising on www.asp.net, a copyright statement, and a link to the privacy statement.

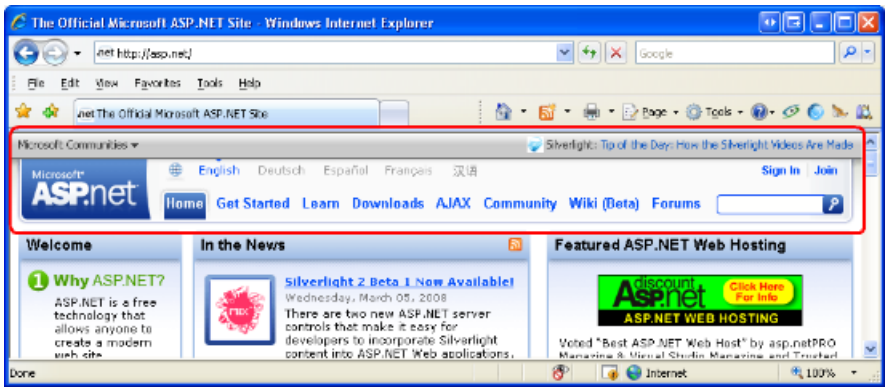


Figure 01: The www.asp.net Website Employs a Consistent Look and Feel Across All Pages (Click to view full-size image)

Another attribute of a well-designed site is the ease with which the site's appearance can be changed. Figure 1 shows the www.asp.net homepage as of March 2008, but between now and this tutorial's publication, the look and feel may have changed. Perhaps the menu items along the top will expand to include a new section for the MVC framework. Or maybe a radically new design with different colors, fonts, and layout will be unveiled. Applying such changes to the entire site should be a fast and simple process that does not require modifying the thousands of web pages that make up the site.

Creating a site-wide page template in ASP.NET is possible through the use of *master pages*. In a nutshell, a master page is a special type of ASP.NET page that defines the markup that is common among all *content pages* as well as regions that are customizable on a content page-by-content page basis. (A content page is an ASP.NET page that is bound to the master page.) Whenever a master page's layout or formatting is changed, all of its content pages' output is likewise immediately updated, which makes applying site-wide appearance changes as easy as updating and deploying a single file (namely, the master page).

This is the first tutorial in a series of tutorials that explore using master pages. Over the course of this tutorial series we:

- Examine creating master pages and their associated content pages,
- Discuss a variety of tips, tricks, and traps,
- Identify common master page pitfalls and explore workarounds,
- See how to access the master page from a content page and vice-a-versa,
- Learn how to specify a content page's master page at runtime, and
- Other advanced master page topics.

These tutorials are geared to be concise and provide step-by-step instructions with plenty of screen shots to walk you through the process visually. Each tutorial is available in C# and Visual Basic versions and includes a download of the complete code used.

This inaugural tutorial starts with a look at master page basics. We discuss how master pages work, look at creating a master page and associated content pages using Visual Web Developer, and see how changes to a master page are immediately reflected in its content pages. Let's get started!

Understanding How Master Pages Work

Building a website with a consistent site-wide page layout requires that each web page emit common formatting markup in addition to its custom content. For example, while each tutorial or forum post on www.asp.net have their own unique content, each of these pages also render a series of common `<div>` elements that display the top-level section links: Home, Get Started, Learn, and so on.

There are a variety of techniques for creating web pages with a consistent look and feel. A naive approach is to simply copy and paste the common layout markup into all web pages, but this approach has a number of downsides. For starters, every time a new page is created, you must remember to copy and paste the shared content into the page. Such copying and pasting operations are ripe for error as you may accidentally copy only a subset of the shared markup into a new page. And to top it off, this approach makes replacing the existing site-wide appearance with a new one a real pain because every single page in the site must be edited in order to use the new look and feel.

Prior to ASP.NET version 2.0, page developers often placed common markup in User Controls and then added these User Controls to each and every page. This approach required that the page developer remember to manually add the User Controls to every new page, but allowed for easier site-wide modifications because when updating the common markup only the User Controls needed to be modified. Unfortunately, Visual Studio .NET 2002 and 2003 - the versions of Visual Studio used to create ASP.NET 1.x applications - rendered User Controls in the Design view as gray boxes. Consequently, page developers using this approach did not enjoy a WYSIWYG design-time environment.

The shortcomings of using User Controls were addressed in ASP.NET version 2.0 and Visual Studio 2005 with the introduction of *master pages*. A master page is a special type of ASP.NET page that defines both the site-wide markup and the *regions* where associated *content pages* define their custom markup. As we will see in Step 1, these regions are defined by ContentPlaceHolder controls. The ContentPlaceHolder control simply denotes a position in the master page's control hierarchy where custom content can be injected by a content page.

> **NOTE**
> The core concepts and functionality of master pages has not changed since ASP.NET version 2.0. However, Visual Studio 2008 offers design-time support for nested master pages, a feature that was lacking in Visual Studio 2005. We will look at using nested master pages in a future tutorial.

Figure 2 shows what the master page for www.asp.net might look like. Note that the master page defines the common site-wide layout - the markup at the top, bottom, and right of every page - as well as a ContentPlaceHolder in the middle-left, where the unique content for each individual web page is located.

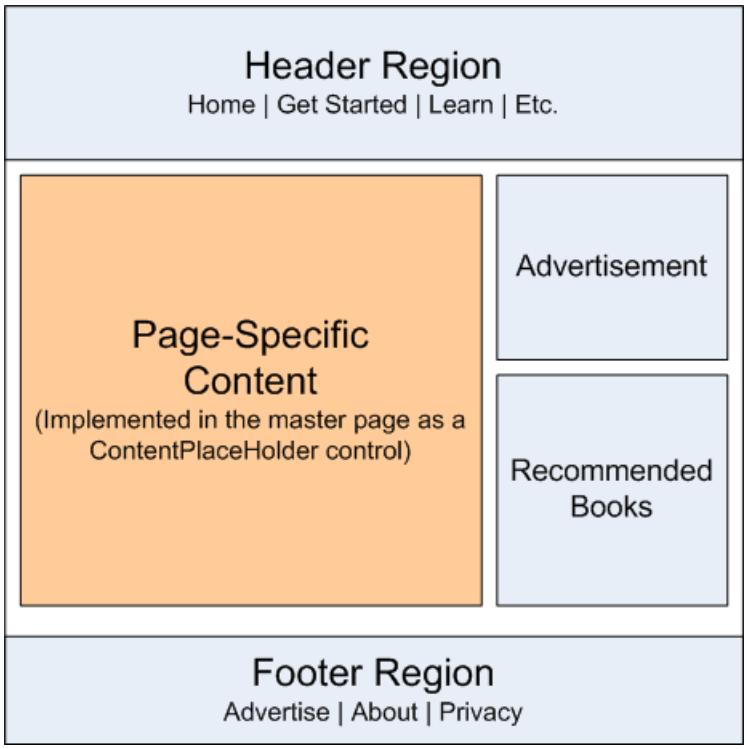


Figure 02: A Master Page Defines the Site-Wide Layout and the Regions Editable on a Content Page-by-Content Page Basis

Once a master page has been defined it can be bound to new ASP.NET pages through the tick of a checkbox. These ASP.NET pages - called content pages - include a Content control for each of the master page's ContentPlaceHolder controls. When the content page is visited through a browser the ASP.NET engine creates the master page's control hierarchy and injects the content page's control hierarchy into the appropriate places. This combined control hierarchy is rendered and the resulting HTML is returned to the end user's browser. Consequently, the content page emits both the common markup defined in its master page outside of the ContentPlaceHolder controls and the page-specific markup defined within its own Content controls. Figure 3 illustrates this concept.

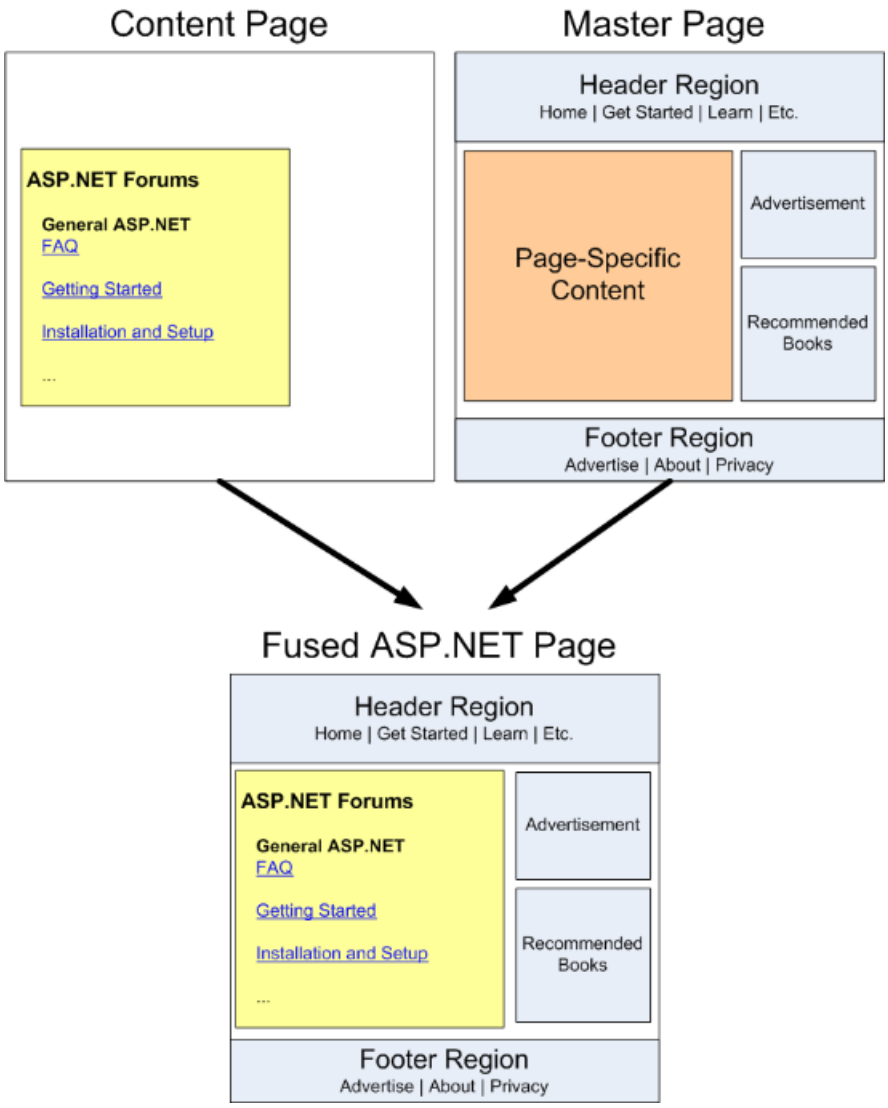


Figure 03: The Requested Page's Markup is Fused into the Master Page (Click to view full-size image)

Now that we have discussed how master pages work, let's take a look at creating a master page and associated content pages using Visual Web Developer.

> **NOTE**
>
> In order to reach the widest possible audience, the ASP.NET website we build throughout this tutorial series will be created using ASP.NET 3.5 with Microsoft's free version of Visual Studio 2008, Visual Web Developer 2008. If you have not yet upgraded to ASP.NET 3.5, don't worry - the concepts discussed in these tutorials work equally well with ASP.NET 2.0 and Visual Studio 2005. However, some demo applications may use features new to the .NET Framework version 3.5; when 3.5-specific features are used, I include a note that discusses how to implement similar functionality in version 2.0. Do keep in mind that the demo applications available for download from each tutorial target the .NET Framework version 3.5, which results in a `Web.config` file that includes 3.5-specific configuration elements. Long story short, if you have yet to install .NET 3.5 on your computer then the downloadable web application will not work without first removing the 3.5-specific markup from `Web.config`. See Dissecting ASP.NET Version 3.5's `Web.config` File for more information on this topic.

Step 1: Creating a Master Page

Before we can explore creating and using master and content pages, we first need an ASP.NET website. Start by creating a new file system-based ASP.NET website. To accomplish this, launch Visual Web Developer and then go to the File menu and choose New Web Site, displaying the New Web Site dialog box (see Figure 4). Choose the ASP.NET Web Site template, set the Location drop-down list to File System, choose a folder to place the web site, and set the language to Visual Basic. This will create a new web site with a `Default.aspx` ASP.NET page, an

`App_Data` folder, and a `Web.config` file.

> **NOTE**
>
> Visual Studio supports two modes of project management: Web Site Projects and Web Application Projects. Web Site Projects lack a project file, whereas Web Application Projects mimic the project architecture in Visual Studio .NET 2002/2003 - they include a project file and compile the project's source code into a single assembly, which is placed in the `/bin` folder. Visual Studio 2005 initially only supported Web Site Projects, although the Web Application Project model was reintroduced with Service Pack 1; Visual Studio 2008 offers both project models. The Visual Web Developer 2005 and 2008 editions, however, only support Web Site Projects. I use the Web Site Project model for my demos in this tutorial series. If you are using a non-Express edition and want to use the Web Application Project model instead, feel free to do so but be aware that there may be some discrepancies between what you see on your screen and the steps you must take versus the screen shots shown and instructions provided in these tutorials.

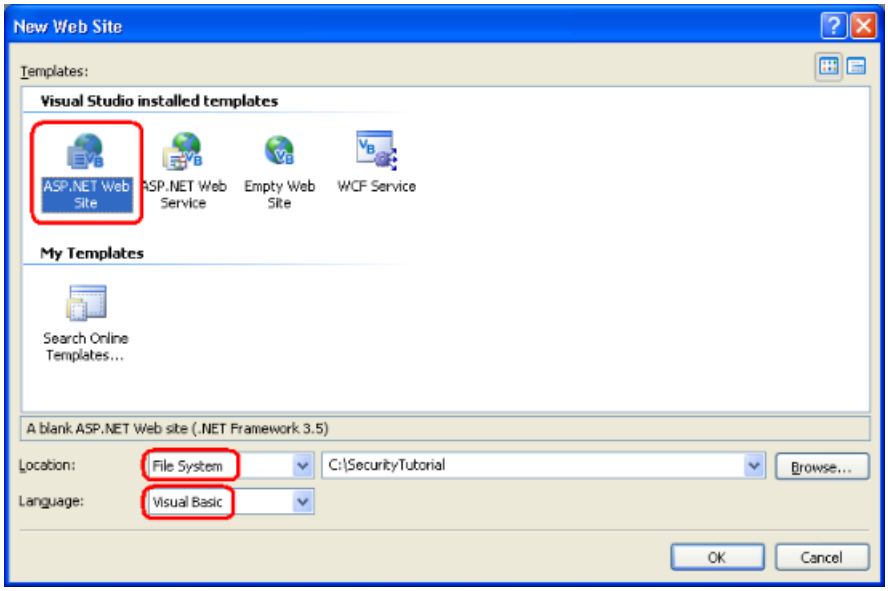


Figure 04: Create a New File System-Based Web Site (Click to view full-size image)

Next, add a master page to the site in the root directory by right-clicking on the Project name, choosing Add New Item, and selecting the Master Page template. Note that master pages end with the extension `.master`. Name this new master page `Site.master` and click Add.

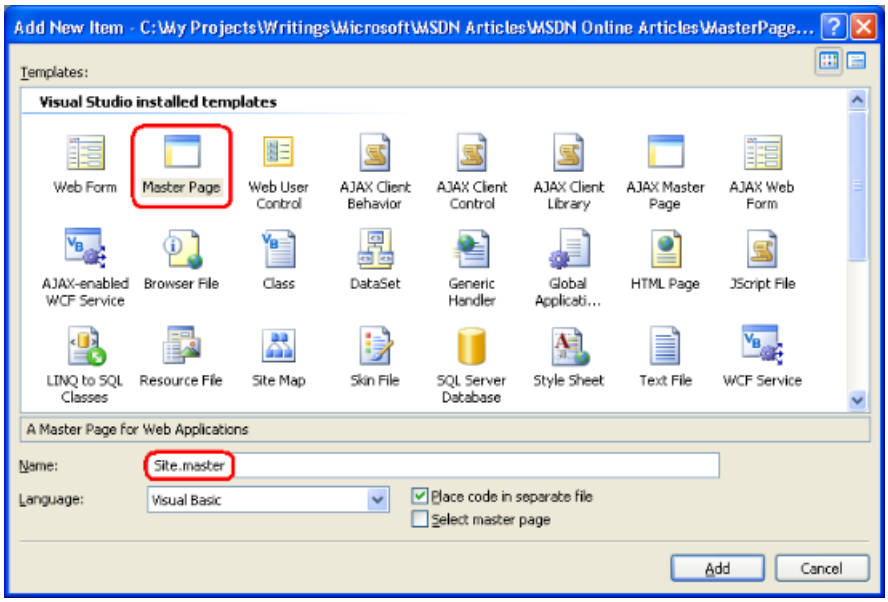


Figure 05: Add a Master Page Named `Site.master` to the Website (Click to view full-size image)

Adding a new master page file through Visual Web Developer creates a master page with the following

declarative markup:

```
<%@ Master Language="VB" CodeFile="Site.master.vb" Inherits="Site" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
</head>
<body>
 <form id="form1" runat="server">
 <div>
 <asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server">

 </asp:ContentPlaceHolder>
 </div>
 </form>
</body>
</html>
```

The first line in the declarative markup is the `@Master` directive. The `@Master` directive is similar to the `@Page` directive that appears in ASP.NET pages. It defines the server-side language (VB) and information about the location and inheritance of the master page's code-behind class.

The `DOCTYPE` and the page's declarative markup appears beneath the `@Master` directive. The page includes static HTML along with four server-side controls:

- **A Web Form (the `<form runat="server">`)** - because all ASP.NET pages typically have a Web Form - and because the master page may include Web controls that must appear within a Web Form - be sure to add the Web Form to your master page (rather than adding a Web Form to each content page).
- **A ContentPlaceHolder control named `ContentPlaceHolder1`** - this ContentPlaceHolder control appears within the Web Form and serves as the region for the content page's user interface.
- **A server-side `<head>` element** - the `<head>` element has the `runat="server"` attribute, making it accessible through server-side code. The `<head>` element is implemented this way so that the page's title and other `<head>`-related markup may be added or adjusted programmatically. For example, setting an ASP.NET page's `Title` property changes the `<title>` element rendered by the `<head>` server control.
- **A ContentPlaceHolder control named `head`** - this ContentPlaceHolder control appears within the `<head>` server control and can be used to declaratively add content to the `<head>` element.

This default master page declarative markup serves as a starting point for designing your own master pages. Feel free to edit the HTML or to add additional Web controls or ContentPlaceHolders to the master page.

> **NOTE**
> When designing a master page make sure that the master page contains a Web Form and that at least one ContentPlaceHolder control appears within this Web Form.

Creating a Simple Site Layout

Let's expand `Site.master`'s default declarative markup to create a site layout where all pages share: a common header; a left column with navigation, news and other site-wide content; and a footer that displays the "Powered by Microsoft ASP.NET" icon. Figure 6 shows the end result of the master page when one of its content pages is viewed through a browser. The red circled region in Figure 6 is specific to the page being visited (`Default.aspx`); the other content is defined in the master page and therefore consistent across all content pages.

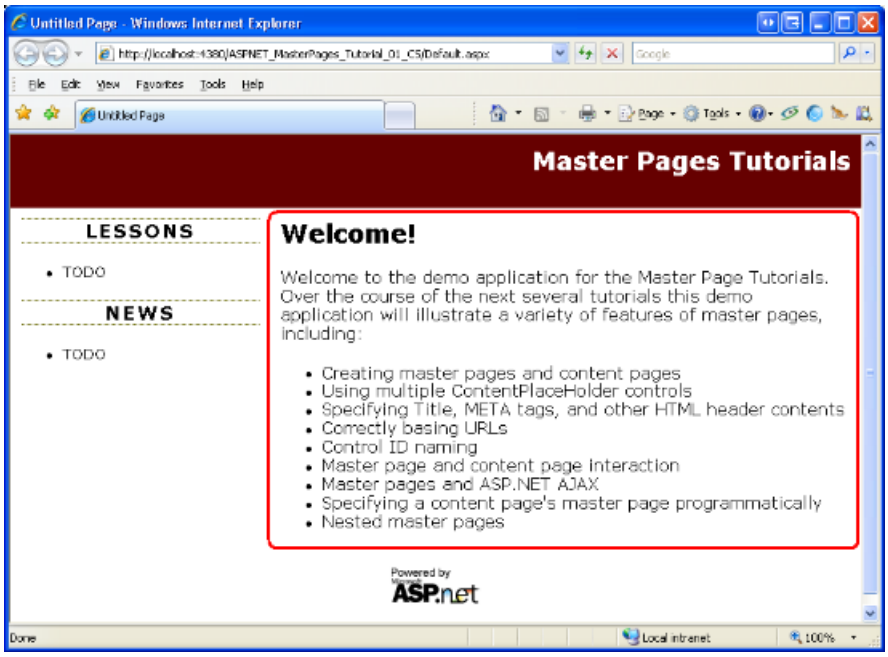


Figure 06: The Master Page Defines the Markup for the Top, Left, and Bottom Portions (Click to view full-size image)

To achieve the site layout shown in Figure 6, start by updating the `Site.master` master page so that it contains the following declarative markup:

```
<%@ Master Language="VB" CodeFile="Site.master.vb" Inherits="Site" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
 <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
 <form id="form1" runat="server">
 <div id="topContent">
 <a href="Default.aspx">Master Pages Tutorials</a>
 </div>

 <div id="mainContent">
 <asp:ContentPlaceHolder id="MainContent" runat="server">
 </asp:ContentPlaceHolder>
 </div>

 <div id="leftContent">
 <h3>Lessons</h3>
 <ul>
 <li>TODO</li>
 </ul>
 <h3>News</h3>
 <ul>
 <li>TODO</li>
 </ul>
 </div>

 <div id="footerContent">
 <img src="Images/PoweredByASPNET.gif" alt="Powered by ASP.NET!" />
 </div>
 </form>
  </body>
</html>
```

The master page's layout is defined using a series of `<div>` HTML elements. The `topContent` `<div>` contains the markup that appears at the top of each page, while the `mainContent`, `leftContent`, and `footerContent` `<div>` s are used to display the page's content, the left column, and the "Powered by Microsoft ASP.NET" icon, respectively. In addition to adding these `<div>` elements, I also renamed the `ID` property of the primary ContentPlaceHolder control from `ContentPlaceHolder1` to `MainContent`.

The formatting and layout rules for these assorted `<div>` elements is spelled out in the Cascading Stylesheet (CSS) file `Styles.css`, which is specified via a `<link>` element in the master page's `<head>` element. These various rules define the look and feel of each `<div>` element noted above. For example, the `topContent` `<div>` element, which displays the "Master Pages Tutorials" text and link, has its formatting rules specified in `Styles.css` as follows:

```
#topContent {
 text-align: right;
 background-color: #600;
 color: White;
 font-size: x-large;
 text-decoration: none;
 font-weight: bold;
 padding: 10px;
 height: 50px;
}
```

If you are following along at your computer, you will need to download this tutorial's accompanying code and add the `Styles.css` file to your project. Similarly, you will also need to create a folder named `Images` and copy the "Powered by Microsoft ASP.NET" icon from the downloaded demo website to your project.

> **NOTE**
> A discussion of CSS and web page formatting is beyond the scope of this article. For more on CSS, check out the CSS Tutorials at W3Schools.com. I also encourage you to download this tutorial's accompanying code and play with the CSS settings in `Styles.css` to see the effects of different formatting rules.

Creating a Master Page Using an Existing Design Template

Over the years I've built a number of ASP.NET web applications for small- to medium-sized companies. Some of my clients had an existing site layout they wanted to use; others hired a competent graphics designer. A few entrusted me to design the website layout. As you can tell by Figure 6, tasking a programmer to design a website's layout is usually as wise as having your accountant perform open-heart surgery while your doctor does your taxes.

Fortunately, there are innumerous websites that offer free HTML design templates - Google returned more than six million results for the search term "free website templates." One of my favorite ones is OpenDesigns.org. Once you find a website template you like, add the CSS files and images to your website project and integrate the template's HTML into your master page.

> **NOTE**
> Microsoft also offers a number of free ASP.NET Design Start Kit Templates that integrate into the New Web Site dialog box in Visual Studio.

Step 2: Creating Associated Content Pages

With the master page created, we are ready to start creating ASP.NET pages that are bound to the master page. Such pages are referred to as *content pages*.

Let's add a new ASP.NET page to the project and bind it to the `Site.master` master page. Right-click on the project name in Solution Explorer and choose the Add New Item option. Select the Web Form template, enter the name `About.aspx`, and then check the "Select master page" checkbox as shown in Figure 7. Doing so will display the Select a Master Page dialog box (see Figure 8) from where you can choose the master page to use.

> **NOTE**
> If you created your ASP.NET website using the Web Application Project model instead of the Web Site Project model you will not see the "Select master page" checkbox in the Add New Item dialog box shown in Figure 7. To create a content page when using the Web Application Project model you must choose the Web Content Form template instead of the Web Form template. After selecting the Web Content Form template and clicking Add, the same Select a Master Page dialog box shown in Figure 8 will appear.

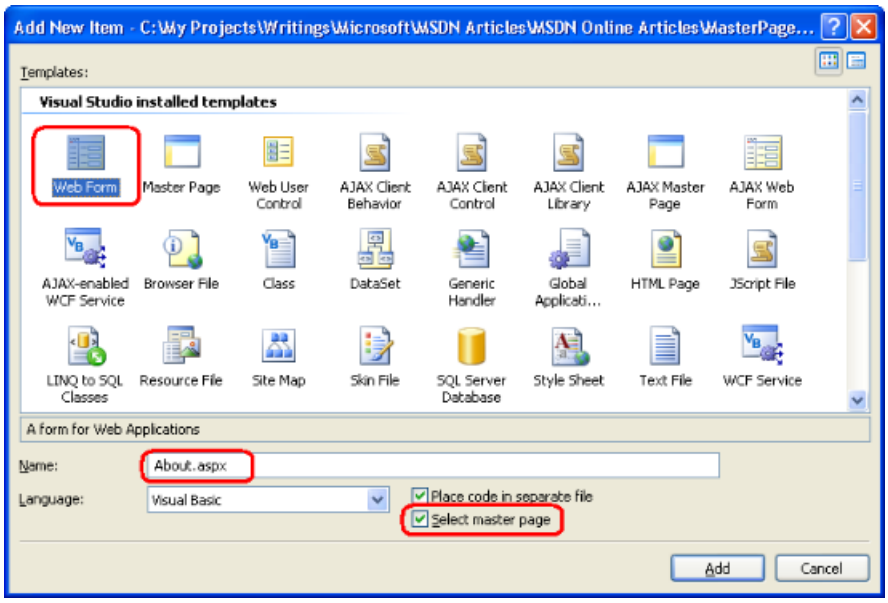


Figure 07: Add a New Content Page (Click to view full-size image)

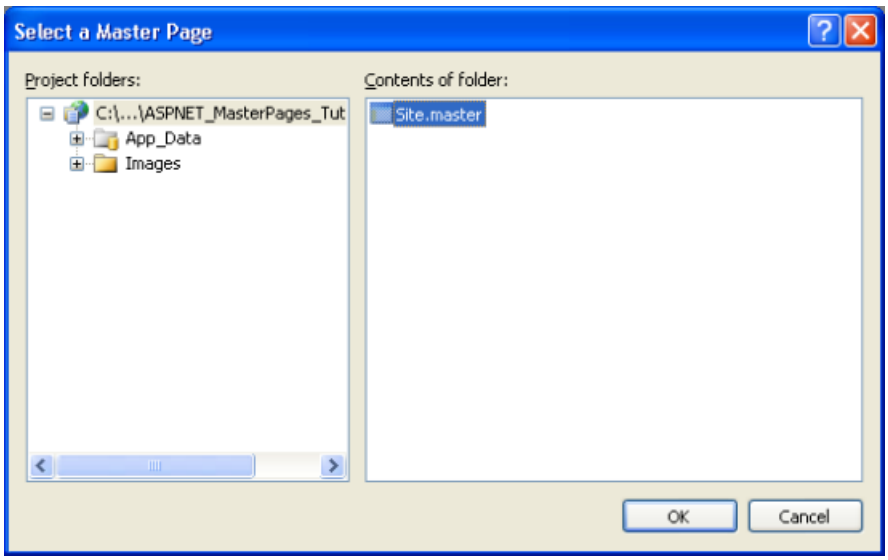


Figure 08: Select the `Site.master` Master Page (Click to view full-size image)

As the following declarative markup shows, a new content page contains a `@Page` directive that points back to its master page and a Content control for each of the master page's ContentPlaceHolder controls.

```
<%@ Page Language="VB" MasterPageFile="~/Site.master" AutoEventWireup="false" CodeFile="About.aspx.vb"
Inherits="About" Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
</asp:Content>
```

NOTE

In the "Creating a Simple Site Layout" section in Step 1 I renamed `ContentPlaceHolder1` to `MainContent`. If you did not rename this ContentPlaceHolder control's `ID` in the same way, your content page's declarative markup will differ slightly from the markup shown above. Namely, the second Content control's `ContentPlaceHolderID` will reflect the `ID` of the corresponding ContentPlaceHolder control in your master page.

When rendering a content page, the ASP.NET engine must fuse the page's Content controls with its master page's ContentPlaceHolder controls. The ASP.NET engine determines the content page's master page from the `@Page`

directive's `MasterPageFile` attribute. As the above markup shows, this content page is bound to `~/Site.master`.

Because the master page has two ContentPlaceHolder controls - `head` and `MainContent` - Visual Web Developer generated two Content controls. Each Content control references a particular ContentPlaceHolder via its `ContentPlaceHolderID` property.

Where master pages shine over previous site-wide template techniques is with their design-time support. Figure 9 shows the `About.aspx` content page when viewed through Visual Web Developer's Design view. Note that while the master page content is visible, it is grayed out and cannot be modified. The Content controls corresponding to the master page's ContentPlaceHolders are, however, editable. And just like with any other ASP.NET page, you can create the content page's interface by adding Web controls through the Source or Design views.

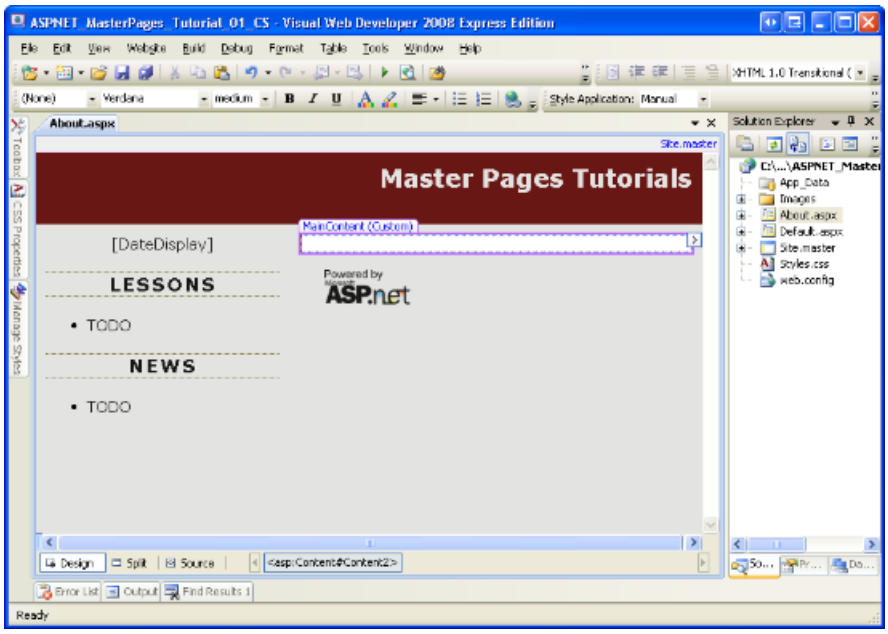


Figure 09: The Content Page's Design View Displays Both the Page-Specific and Master Page Contents (Click to view full-size image)

Adding Markup and Web Controls to the Content Page

Take a moment to create some content for the `About.aspx` page. As you can see in Figure 10, I entered an "About the Author" heading and a couple of paragraphs of text, but feel free to add Web controls, too. After creating this interface, visit the `About.aspx` page through a browser.

Figure 10: Visit the `About.aspx` Page Through a Browser (Click to view full-size image)

It is important to understand that the requested content page and its associated master page are fused and rendered as a whole entirely on the web server. The end user's browser is then sent the resulting, fused HTML. To verify this, view the HTML received by the browser by going to the View menu and choosing Source. Note that there are no frames or any other specialized techniques for displaying two different web pages in a single window.

Binding a Master Page to an Existing ASP.NET Page

As we saw in this step, adding a new content page to an ASP.NET web application is as easy as checking the "Select master page" checkbox and picking the master page. Unfortunately, converting an existing ASP.NET page to a master page is not as easy.

To bind a master page to an existing ASP.NET page you need to perform the following steps:

1. Add the `MasterPageFile` attribute to the ASP.NET page's `@Page` directive, pointing it to the appropriate master page.
2. Add Content controls for each of the ContentPlaceHolders in the master page.
3. Selectively cut and paste the ASP.NET page's existing content into the appropriate Content controls. I say "selectively" here because the ASP.NET page likely contains markup that's already expressed by the master page, such as the `DOCTYPE`, the `<html>` element, and the Web Form.

For step-by-step instructions on this process along with screen shots, check out Scott Guthrie's Using Master Pages and Site Navigation tutorial. The "Update `Default.aspx` and `DataSample.aspx` to use the Master Page" section details these steps.

Because it is much easier to create new content pages than it is to convert existing ASP.NET pages into content pages, I recommend that whenever you create a new ASP.NET website add a master page to the site. Bind all new ASP.NET pages to this master page. Don't worry if the initial master page is very simple or plain; you can update the master page later.

NOTE

When creating a new ASP.NET application, Visual Web Developer adds a `Default.aspx` page that isn't bound to a master page. If you want to practice converting an existing ASP.NET page into a content page, go ahead and do so with `Default.aspx`. Alternatively, you can delete `Default.aspx` and then re-add it, but this time checking the "Select master page" checkbox.

Step 3: Updating the Master Page's Markup

One of the primary benefits of master pages is that a single master page may be used to define the overall layout for numerous pages on the site. Therefore, updating the site's look and feel requires updating a single file - the master page.

To illustrate this behavior, let's update our master page to include the current date in at the top of the left column. Add a Label named `DateDisplay` to the `leftContent` `<div>`.

```
<div id="leftContent">
 <p>
 <asp:Label ID="DateDisplay" runat="server"></asp:Label>
 </p>

 <h3>Lessons</h3>
 <ul>
 <li>TODO</li>
 </ul>
 <h3>News</h3>
 <ul>
 <li>TODO</li>
 </ul>
</div>
```

Next, create a `Page_Load` event handler for the master page and add the following code:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
 DateDisplay.Text = DateTime.Now.ToString("dddd, MMMM dd")
End Sub
```

The above code sets the Label's `Text` property to the current date and time formatted as the day of the week, the name of the month, and the two-digit day (see Figure 11). With this change, revisit one of your content pages. As Figure 11 shows, the resulting markup is immediately updated to include the change to the master page.

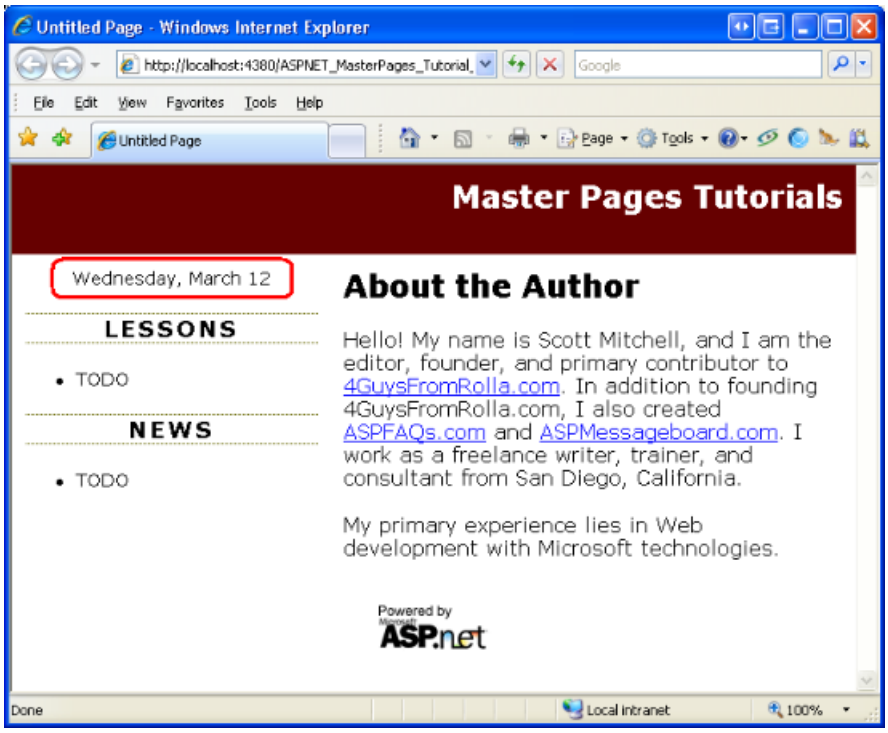


Figure 11: The Changes to the Master Page are Reflected When Viewing the a Content Page (Click to view full-size image)

> **NOTE**
>
> As this example illustrates, master pages may contain server-side Web controls, code, and event handlers.

Summary

Master pages enable ASP.NET developers to design a consistent site-wide layout that is easily updateable. Creating master pages and their associated content pages is as simple as creating standard ASP.NET pages, as Visual Web Developer offers rich design-time support.

The master page example we created in this tutorial had two ContentPlaceHolder controls, head and MainContent. We only specified markup for the MainContent ContentPlaceHolder control in our content page, however. In the next tutorial we look at using multiple Content controls in the content page. We also see how to define default markup for Content controls within the master page, as well as how to toggle between using the default markup defined in the master page and providing custom markup from the content page.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET for Designers: Free Design Templates and Guidance on Building ASP.NET Websites Using Web Standards
- ASP.NET Master Pages Overview
- Cascading Stylesheets (CSS) Tutorials
- Dynamically Setting the Page's Title
- Master Pages in ASP.NET
- Master Pages QuickStart Tutorials

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working

with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Multiple ContentPlaceHolders and Default Content (VB)

2/20/2020 • 13 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> Examines how to add multiple content place holders to a master page as well as how to specify default content in the content place holders.

Introduction

In the preceding tutorial we examined how master pages enable ASP.NET developers to create a consistent site-wide layout. Master pages define both markup that is common to all of its content pages and regions that are customizable on a page-by-page basis. In the previous tutorial we created a simple master page (`Site.master`) and two content pages (`Default.aspx` and `About.aspx`). Our master page consisted of two ContentPlaceHolders named `head` and `MainContent`, which were located in the `<head>` element and Web Form, respectively. While the content pages each had two Content controls, we only specified markup for the one corresponding to `MainContent`.

As evidenced by the two ContentPlaceHolder controls in `Site.master`, a master page may contain multiple ContentPlaceHolders. What's more, the master page may specify default markup for the ContentPlaceHolder controls. A content page, then, can optionally specify its own markup or use the default markup. In this tutorial we look at using multiple content controls in the master page and see how to define default markup in the ContentPlaceHolder controls.

Step 1: Adding Additional ContentPlaceHolder Controls to the Master Page

Many website designs contain several areas on the screen that are customized on a page-by-page basis. `Site.master`, the master page we created in the preceding tutorial, contains a single ContentPlaceHolder within the Web Form named `MainContent`. Specifically, this ContentPlaceHolder is located within the `mainContent` `<div>` element.

Figure 1 shows `Default.aspx` when viewed through a browser. The region circled in red is the page-specific markup corresponding to `MainContent`.

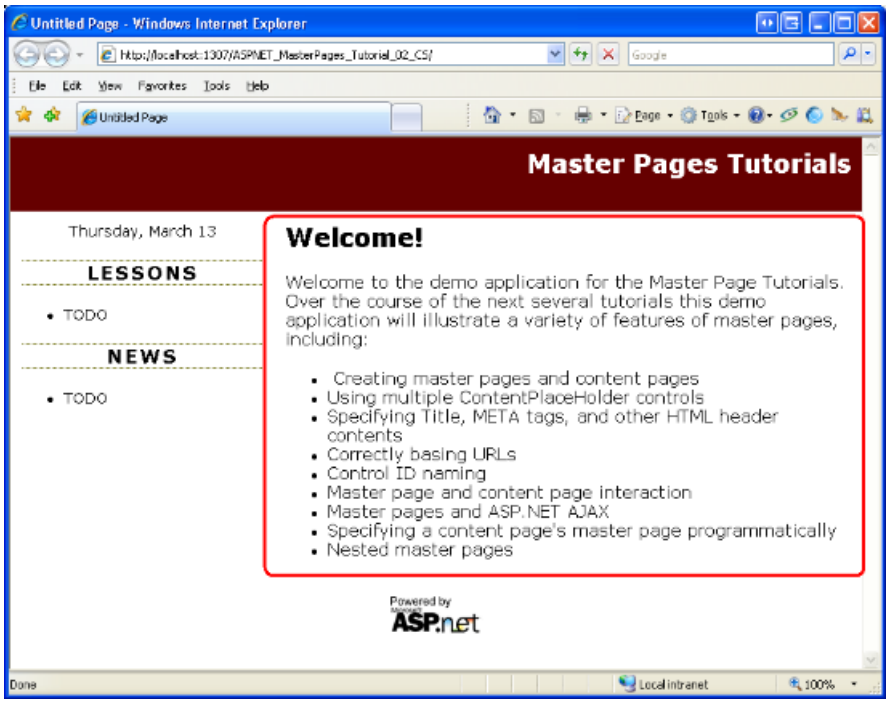


Figure 01: The Circled Region Shows the Area Currently Customizable on a Page-by-Page Basis (Click to view full-size image)

Imagine that in addition to the region shown in Figure 1, we also need to add page-specific items to the left column beneath the Lessons and News sections. To accomplish this, we add another ContentPlaceHolder control to the master page. To follow along, open the `Site.master` master page in Visual Web Developer and then drag a ContentPlaceHolder control from the Toolbox onto the designer after the News section. Set the ContentPlaceHolder's `ID` to `LeftColumnContent`.

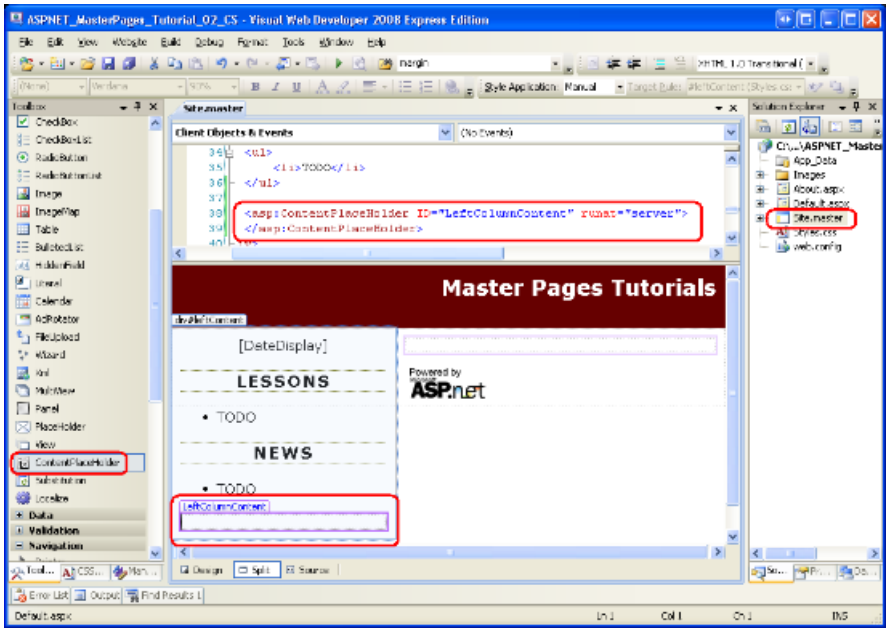


Figure 02: Add a ContentPlaceHolder Control to the Master Page's Left Column (Click to view full-size image)

With the addition of the `LeftColumnContent` ContentPlaceHolder to the master page, we can define content for this region on a page-by-page basis by including a Content control in the page whose `ContentPlaceHolderID` is set to `LeftColumnContent`. We examine this process in Step 2.

Step 2: Defining Content for the New ContentPlaceHolder in the Content Pages

When adding a new content page to the website, Visual Web Developer automatically creates a Content control in the page for each ContentPlaceHolder in the selected master page. Having added a the `LeftColumnContent` ContentPlaceHolder to our master page in Step 1, new ASP.NET pages will now have three Content controls.

To illustrate this, add a new content page to the root directory named `MultipleContentPlaceHolders.aspx` that is bound to the `Site.master` master page. Visual Web Developer creates this page with the following declarative markup:

```
<%@ Page Language="VB" MasterPageFile="~/Site.master" AutoEventWireup="false"
CodeFile="MultipleContentPlaceHolders.aspx.vb" Inherits="MultipleContentPlaceHolders" Title="Untitled Page"
%>
<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
</asp:Content>
<asp:Content ID="Content3" ContentPlaceHolderID="LeftColumnContent" Runat="Server">
</asp:Content>
```

Enter some content into the Content control referencing the `MainContent` ContentPlaceHolders (`Content2`). Next, add the following markup to the `Content3` Content control (which references the `LeftColumnContent` ContentPlaceHolder):

```
<h3>Page-Specific Content</h3>
<ul>
 <li>This content is defined in the content page.</li>
 <li>The master page has two regions in the Web Form that are editable on a
 page-by-page basis.</li>
</ul>
```

After adding this markup, visit the page through a browser. As Figure 3 shows, the markup placed in the `Content3` Content control is displayed in the left column beneath the News section (circled in red). The markup placed in `Content2` is displayed in the right portion of the page (circled in blue).

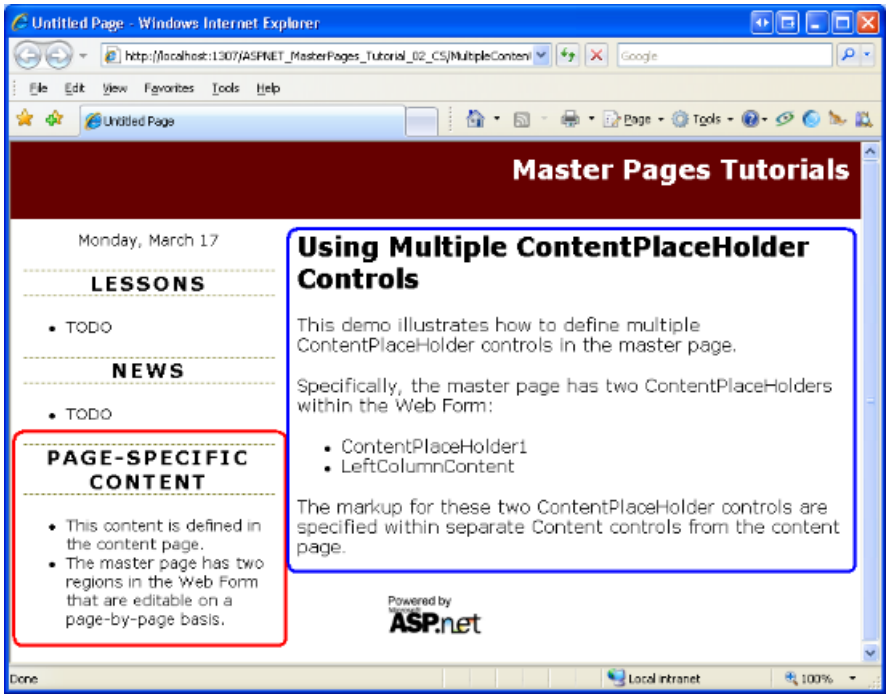


Figure 03: The Left Column Now Includes Page-Specific Content Beneath the News Section (Click to view full-size image)

Defining Content in Existing Content Pages

Creating a new content page automatically incorporates the ContentPlaceHolder control we added in Step 1. But our two existing content pages - `About.aspx` and `Default.aspx` - don't have a Content control for the `LeftColumnContent` ContentPlaceHolder. To specify content for this ContentPlaceHolder in these two existing pages, we need to add a Content control ourselves.

Unlike most ASP.NET Web controls, the Visual Web Developer Toolbox does not include a Content control item. We can manually type in the Content control's declarative markup into the Source view, but an easier and quicker approach is to use the Design view. Open the `About.aspx` page and switch to the Design view. As Figure 4 illustrates, the `LeftColumnContent` ContentPlaceHolder appears in the Design view; if you mouse over it, the title displayed reads: "LeftColumnContent (Master)." The inclusion of "Master" in the title indicates that there is no Content control defined in the page for this ContentPlaceHolder. If there exists a Content control for the ContentPlaceHolder, as in the case for `MainContent`, the title will read: "*ContentPlaceHolderID* (Custom)."

To add a Content control for the `LeftColumnContent` ContentPlaceHolder to `About.aspx`, expand the ContentPlaceHolder's smart tag and click the Create Custom Content link.

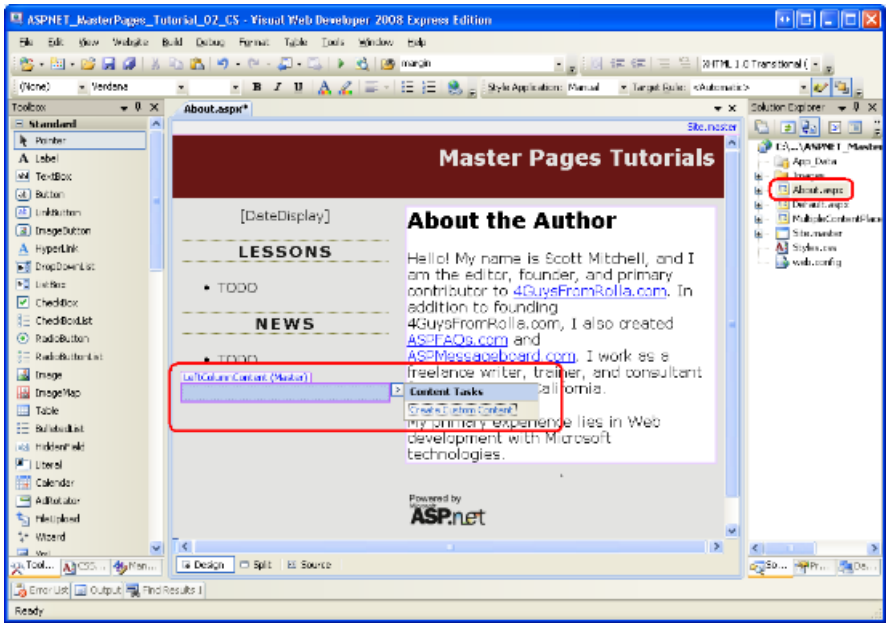


Figure 04: The Design View for `About.aspx` Shows the `LeftColumnContent` ContentPlaceHolder (Click to view full-size image)

Clicking the Create Custom Content link generates the necessary Content control in the page and sets its `ContentPlaceHolderID` property to the ContentPlaceHolder's `ID`. For example, clicking the Create Custom Content link for `LeftColumnContent` region in `About.aspx` adds the following declarative markup to the page:

```
<asp:Content ID="Content3" runat="server"
 contentplaceholderid="LeftColumnContent">

</asp:Content>
```

Omitting Content Controls

ASP.NET does not require that all content pages include Content controls for each and every ContentPlaceHolder defined in the master page. If a Content control is omitted, the ASP.NET engine uses the markup defined within the ContentPlaceHolder in the master page. This markup is referred to as the ContentPlaceHolder's *default content* and is useful in scenarios where the content for some region is common among the majority of pages, but needs to be customized for a small number of pages. Step 3 explores specifying default content in the master page.

Currently, `Default.aspx` contains two Content controls for the `head` and `MainContent` ContentPlaceHolders; it

does not have a Content control for `LeftColumnContent`. Consequently, when `Default.aspx` is rendered the `LeftColumnContent` ContentPlaceHolder's default content is used. Because we have yet to define any default content for this ContentPlaceHolder, the net effect is that no markup is emitted for this region. To verify this behavior, visit `Default.aspx` through a browser. As Figure 5 shows, no markup is emitted in the left column beneath the News section.

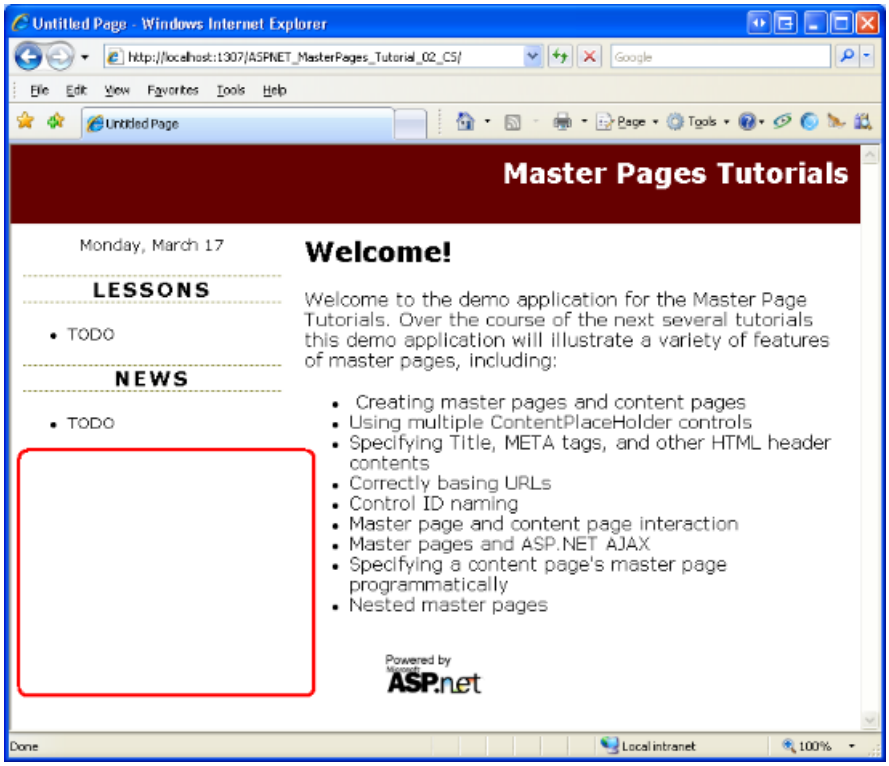


Figure 05: No Content is Rendered for the `LeftColumnContent` ContentPlaceHolder (Click to view full-size image)

Step 3: Specifying Default Content in the Master Page

Some website designs include a region whose content is the same for all pages in the site except for one or two exceptions. Consider a website that supports user accounts. Such a site requires a login page where visitors can enter their credentials to sign into the site. To expedite the sign in process, the website designers might include username and password textboxes in the upper left corner of every page to allow users to sign in without having to explicitly visit the login page. While these username and password textboxes are helpful in most pages, they are redundant in the login page, which already contains textboxes for the user's credentials.

To implement this design, you could create a ContentPlaceHolder control in the upper left corner of the master page. Each page that needed to display the username and password textboxes in their upper left corner would create a Content control for this ContentPlaceHolder and add the necessary interface. The login page, on the other hand, would either omit adding a Content control for this ContentPlaceHolder or would create a Content control with no markup defined. The downside of this approach is that we have to remember to add the username and password textboxes to every page we add to the site (except for the login page). This is asking for trouble. We're likely to forget to add these textboxes to a page or two or, worse, we might not implement the interface correctly (perhaps adding just one textbox instead of two).

A better solution is to define the username and password textboxes as the ContentPlaceHolder's default content. By doing so, we only need to override this default content in those few pages that do not display the username and password textboxes (the login page, for instance). To illustrate specifying default content for a ContentPlaceHolder control, let's implement the scenario just discussed.

> **NOTE**
> The remainder of this tutorial updates our website to include a login interface in the left column for all pages but the login page. However, this tutorial does not examine how to configure the website to support user accounts. For more information on this topic, refer to my Forms Authentication, Authorization, User Accounts and Roles tutorials.

Adding a ContentPlaceHolder and Specifying Its Default Content

Open the `Site.master` master page and add the following markup to the left column between the `DateDisplay` Label and Lessons section:

```
<asp:ContentPlaceHolder ID="QuickLoginUI" runat="server">
 <asp:Login ID="QuickLogin" runat="server"
    TitleText="<h3>Sign In</h3>"
    FailureAction="RedirectToLoginPage">
 </asp:Login>
</asp:ContentPlaceHolder>
```

After adding this markup your master page's Design view should look similar to Figure 6.

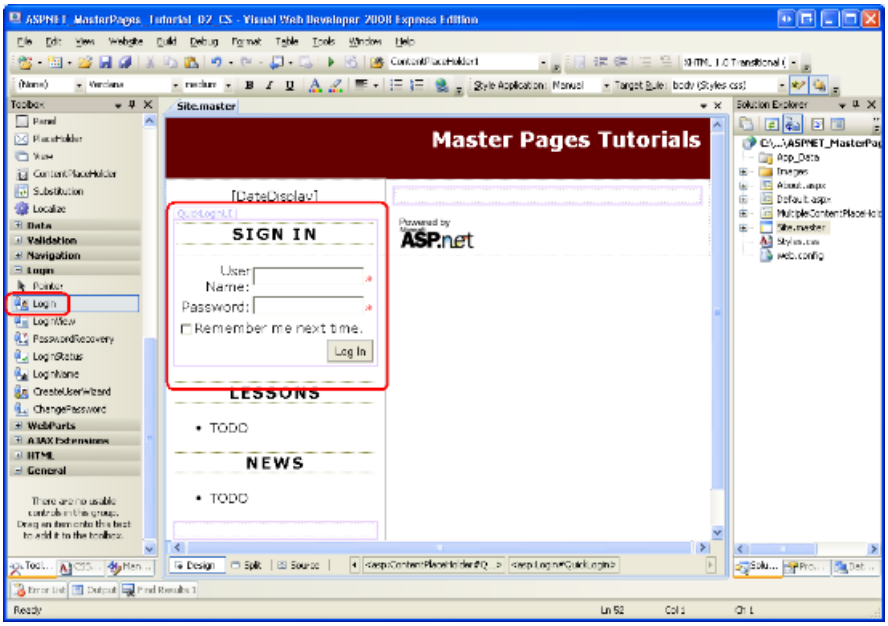


Figure 06: The Master Page Includes a Login Control (Click to view full-size image)

This ContentPlaceHolder, `QuickLoginUI`, has a Login Web control as its default content. The Login control displays a user interface that prompts the user for their username and password along with a Log In button. Upon clicking the Log In button, the Login control internally validates the user's credentials against the Membership API. To use this Login control in practice, then, you need to configure your site to use Membership. This topic is beyond the scope of this tutorial; refer to my Forms Authentication, Authorization, User Accounts and Roles tutorials for more information on building a web application that supports user accounts.

Feel free to customize the Login control's behavior or appearance. I have set two of its properties: `TitleText` and `FailureAction`. The `TitleText` property value, which defaults to "Log In", is displayed at the top of the control's user interface. I have set this property so that it displays the text "Sign In" as an `<h3>` element. The `FailureAction` property indicates what to do if the user's credentials are invalid. It defaults to a value of `Refresh`, which leaves the user on the same page and displays a failure message within the Login control. I've changed it to `RedirectToLoginPage`, which sends the user to the login page in the event of invalid credentials. I prefer to send the user to the login page when a user attempts to login from some other page, but fails, because the login page can contain additional instructions and options that would not easily fit into the left column. For example, the login page might include options to retrieve a forgotten password or to create a new account.

Creating the Login Page and Overriding the Default Content

With the master page complete, our next step is to create the login page. Add an ASP.NET page to your site's root directory named `Login.aspx`, binding it to the `Site.master` master page. Doing so will create a page with four Content controls, one for each of the ContentPlaceHolders defined in `Site.master`.

Add a Login control to the `MainContent` Content control. Likewise, feel free to add any content to the `LeftColumnContent` region. However, make sure to leave the Content control for the `QuickLoginUI` ContentPlaceHolder empty. This will ensure that the Login control does not appear in the left column of the login page.

After defining the content for the `MainContent` and `LeftColumnContent` regions, your login page's declarative markup should look similar to the following:

```
<%@ Page Language="VB" MasterPageFile="~/Site.master" AutoEventWireup="false" CodeFile="Login.aspx.vb"
Inherits="Login" Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
 <h2>
 Sign In</h2>
 <p>
 <asp:Login ID="Login1" runat="server" TitleText="">
 </asp:Login>
 </p>
</asp:Content>
<asp:Content ID="Content3" ContentPlaceHolderID="QuickLoginUI" Runat="Server">
</asp:Content>
<asp:Content ID="Content4" ContentPlaceHolderID="LeftColumnContent" Runat="Server">
 <h3>Sign In Tasks</h3>
 <ul>
 <li>Create a New Account</li>
 <li>Recover Forgotten Password</li>
 </ul>
 <p>TODO: Turn the above text into links...</p>
</asp:Content>
```

Figure 7 shows this page when viewed through a browser. Because this page specifies a Content control for the `QuickLoginUI` ContentPlaceHolder, it overrides the default content specified in the master page. The net effect is that the Login control displayed in the master page's Design view (see Figure 6) is not rendered in this page.

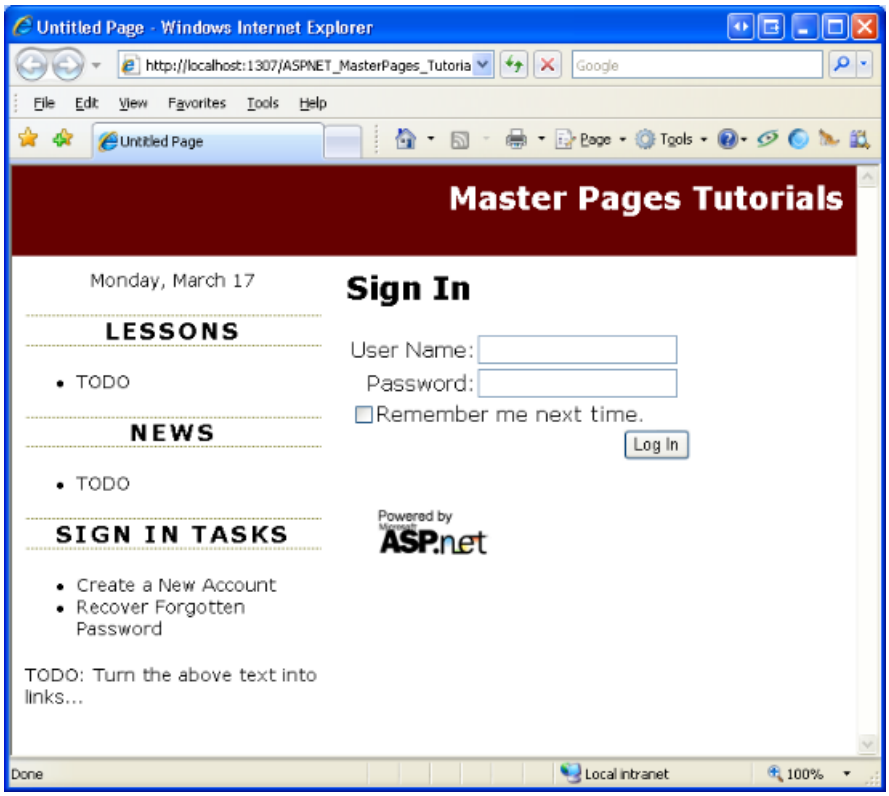


Figure 07: The Login Page Represses the `QuickLoginUI` ContentPlaceHolder's Default Content (Click to view full-size image)

Using the Default Content in New Pages

We want to show the Login control in the left column for all pages except the Login page. To achieve this, all the content pages except for the login page should omit a Content control for the `QuickLoginUI` ContentPlaceHolder. By omitting a Content control, the ContentPlaceHolder's default content will be used instead.

Our existing content pages - `Default.aspx`, `About.aspx`, and `MultipleContentPlaceHolders.aspx` - do not include a Content control for `QuickLoginUI` because they were created before we added that ContentPlaceHolder control to the master page. Therefore, these existing pages do not need to be updated. However, new pages added to the website include a Content control for the `QuickLoginUI` ContentPlaceHolder, by default. Therefore, we have to remember to remove these Content controls each time we add a new content page (unless we want to override the ContentPlaceHolder's default content, as in the case of the login page).

To remove the Content control, you can either manually delete its declarative markup from the Source view or, from the Design view, choose the Default to Master's Content link from its smart tag. Either approach removes the Content control from the page and produces the same net effect.

Figure 8 shows `Default.aspx` when viewed through a browser. Recall that `Default.aspx` only has two Content controls specified in its declarative markup - one for `head` and one for `MainContent`. As a result, the default content for the `LeftColumnContent` and `QuickLoginUI` ContentPlaceHolders are displayed.

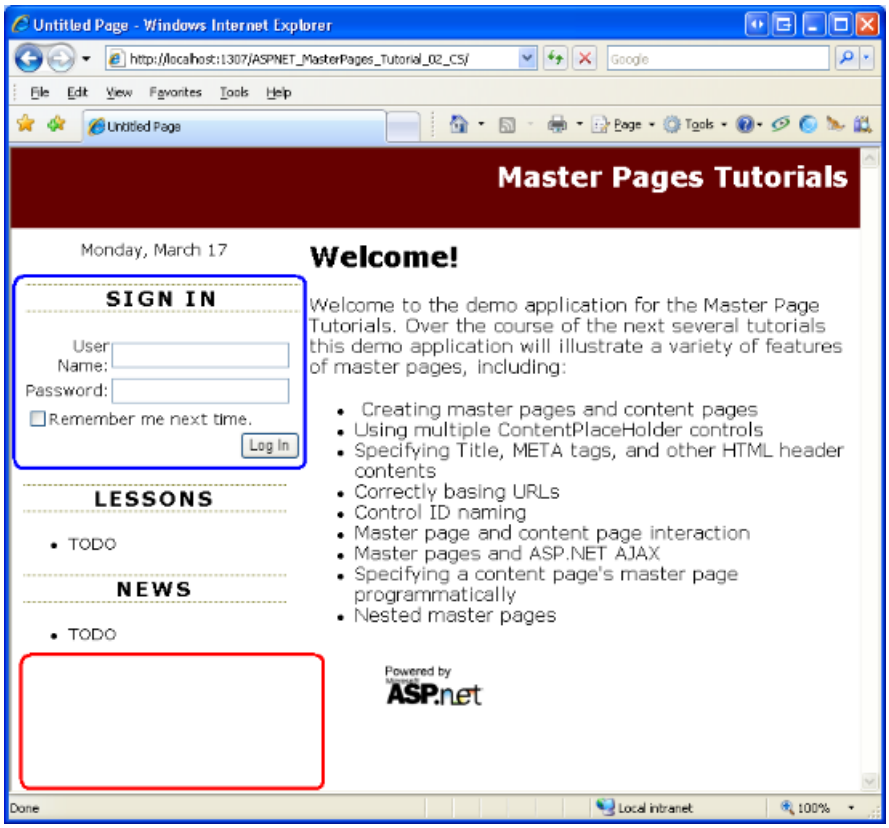


Figure 08: The Default Content for the `LeftColumnContent` and `QuickLoginUI` ContentPlaceHolders are Displayed (Click to view full-size image)

Summary

The ASP.NET master page model allows for an arbitrary number of ContentPlaceHolders in the master page. What's more, ContentPlaceHolders include default content, which is emitted in the case that there is no corresponding Content control in the content page. In this tutorial we saw how to include additional ContentPlaceHolder controls in the master page and how to define Content controls for these new ContentPlaceHolders in both new and existing ASP.NET pages. We also looked at specifying default content in a ContentPlaceHolder, which is useful in scenarios where only a minority of pages needs to customize the otherwise standardized content within a certain region.

In the next tutorial we'll examine the `head` ContentPlaceHolder in more detail, seeing how to declaratively and programmatically define the title, meta tags, and other HTML headers on a page-by-page basis.

Happy Programming!

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Suchi Banerjee. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page (VB)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> Looks at different techniques for defining assorted <head> elements in the Master Page from the content page.

Introduction

New master pages created in Visual Studio 2008 have, by default, two ContentPlaceHolder controls: one named `head`, and located in the `<head>` element; and one named `ContentPlaceHolder1`, placed within the Web Form. The purpose of `ContentPlaceHolder1` is to define a region in the Web Form that can be customized on a page-by-page basis. The `head` ContentPlaceHolder enables pages to add custom content to the `<head>` section. (Of course, these two ContentPlaceHolders can be modified or removed, and additional ContentPlaceHolder may be added to the master page. Our master page, `Site.master`, currently has four ContentPlaceHolder controls.)

The HTML `<head>` element serves as a repository for information about the web page document that is not part of the document itself. This includes information such as the web page's title, meta-information used by search engines or internal crawlers, and links to external resources, such as RSS feeds, JavaScript, and CSS files. Some of this information may be pertinent to all pages in the website. For example, you might want to globally import the same CSS rules and JavaScript files for every ASP.NET page. However, there are portions of the `<head>` element that are page-specific. The page title is a prime example.

In this tutorial we examine how to define global and page-specific `<head>` section markup in the master page and in its content pages.

Examining the Master Page's `<head>` Section

The default master page file created by Visual Studio 2008 contains the following markup in its `<head>` section:

```
<head runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
</head>
```

Notice that the `<head>` element contains a `runat="server"` attribute, which indicates that it is a server control (rather than static HTML). All ASP.NET pages derive from the `Page` class, which is located in the `System.Web.UI` namespace. This class contains a `Header` property that provides access to the page's `<head>` region. Using the `Header` property we can set an ASP.NET page's title or add additional markup to the rendered `<head>` section. It is possible, then, to customize a content page's `<head>` element by writing a bit of code in the page's `Page_Load` event handler. We examine how to programmatically set the page's title in Step 1.

The markup shown in the `<head>` element above also includes a ContentPlaceHolder control named `head`. This ContentPlaceHolder control is not necessary, as content pages can add custom content to the `<head>` element

programmatically. It's useful, however, in situations where a content page needs to add static markup to the `<head>` element as the static markup can be added declaratively to the corresponding Content control rather than programmatically.

In addition to the `<title>` element and `head` ContentPlaceHolder, the master page's `<head>` element should contain any `<head>`-level markup that's common to all pages. In our website, all pages use the CSS rules defined in the `Styles.css` file. Consequently, we updated the `<head>` element in the *Creating a Site-Wide Layout with Master Pages* tutorial to include a corresponding `<link>` element. Our `Site.master` master page's current `<head>` markup is shown below.

```
<head runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
 <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
```

Step 1: Setting a Content Page's Title

The web page's title is specified via the `<title>` element. It is important to set each page's title to an appropriate value. When visiting a page, its title is displayed in the browser's Title bar. Additionally, when bookmarking a page, browsers use the page's title as the suggested name for the bookmark. Also, many search engines show the page's title when displaying search results.

> **NOTE**
> By default, Visual Studio sets the `<title>` element in the master page to "Untitled Page". Similarly, new ASP.NET pages have their `<title>` set to "Untitled Page", too. Because it can be easy to forget to set the page's title to an appropriate value, there are many pages on the Internet with the title "Untitled Page". Searching Google for web pages with this title returns roughly 2,460,000 results. Even Microsoft is susceptible to publishing web pages with the title "Untitled Page". At the time of this writing, a Google search reported 236 such web pages in the Microsoft.com domain.

An ASP.NET page can specify its title in one of the following ways:

- By placing the value directly within the `<title>` element
- Using the `Title` attribute in the `<%@ Page %>` directive
- Programmatically setting the page's `Title` property using code like `Page.Title="title"` or `Page.Header.Title="title"`.

Content pages don't have a `<title>` element, as it's defined in the master page. Therefore, to set a content page's title you can either use the `<%@ Page %>` directive's `Title` attribute or set it programmatically.

Setting the Page's Title Declaratively

A content page's title can be set declaratively through the `Title` attribute of the `<%@ Page %>` directive. This property can be set by directly modifying the `<%@ Page %>` directive or through the Properties window. Let's look at both approaches.

From the Source view, locate the `<%@ Page %>` directive, which is at the top of the page's declarative markup. The `<%@ Page %>` directive for `Default.aspx` follows:

```
<%@ Page Language="VB" MasterPageFile="~/Site.master" AutoEventWireup="false" CodeFile="Default.aspx.vb"
Inherits="_Default" Title="Untitled Page" %>
```

The `<%@ Page %>` directive specifies page-specific attributes used by the ASP.NET engine when parsing and compiling the page. This includes its master page file, the location of its code file, and its title, among other information.

By default, when creating a new content page Visual Studio sets the `Title` attribute to "Untitled Page". Change `Default.aspx`'s `Title` attribute from "Untitled Page" to "Master Page Tutorials" and then view the page through a browser. Figure 1 shows the browser's Title bar, which reflects the new page title.

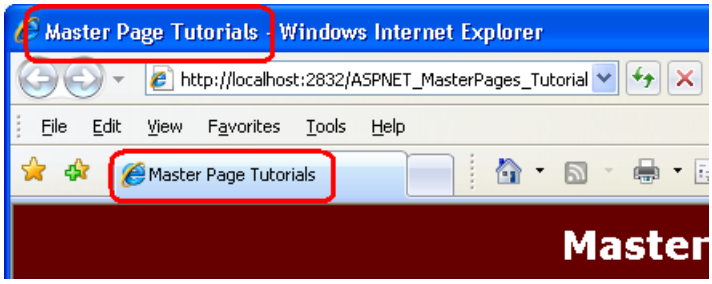


Figure 01: The Browser's Title Bar Now Shows "Master Page Tutorials" Instead of "Untitled Page"

The page's title may also be set from the Properties window. From the Properties window, select DOCUMENT from the drop-down list to load the page-level properties, which includes the `Title` property. Figure 2 shows the Properties window after `Title` has been set to "Master Page Tutorials".

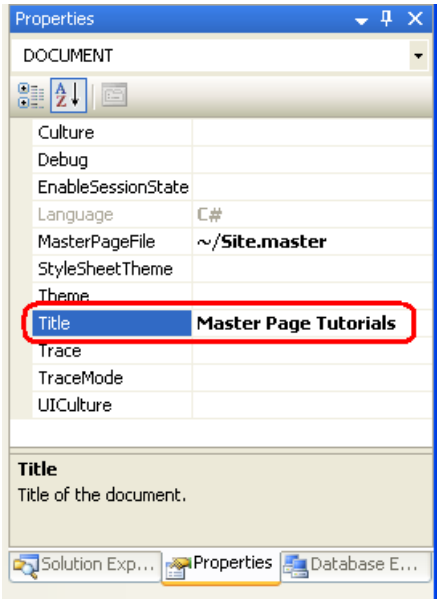


Figure 02: You May Configure the Title from the Properties Window, Too

Setting the Page's Title Programmatically

The master page's `<head runat="server">` markup is translated into an `HtmlHead` class instance when the page is rendered by the ASP.NET engine. The `HtmlHead` class has a `Title` property whose value is reflected in the rendered `<title>` element. This property is accessible from an ASP.NET page's code-behind class via `Page.Header.Title`; this same property can also be accessed via `Page.Title`.

To practice setting the page's title programmatically, navigate to the `About.aspx` page's code-behind class and create an event handler for the page's `Load` event. Next, set the page's title to "Master Page Tutorials :: About :: *date*", where *date* is the current date. After adding this code your `Page_Load` event handler should look similar to the following:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
 Page.Title = String.Format("Master Page Tutorials :: About :: {0:d}", DateTime.Now)
End Sub
```

Figure 3 shows the browser's title bar when visiting the `About.aspx` page.

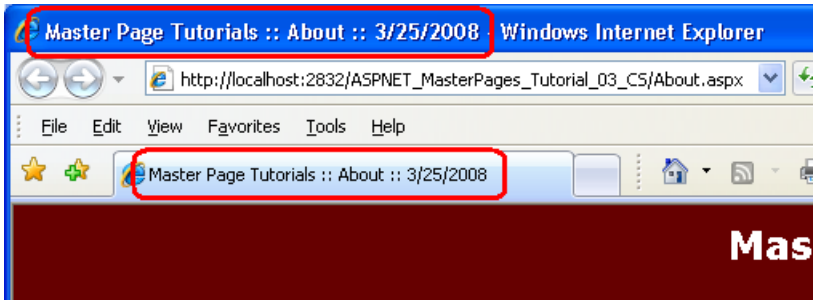


Figure 03: The Page's Title is Programmatically Set and Includes the Current Date

Step 2: Automatically Assigning a Page Title

As we saw in Step 1, a page's title can be set declaratively or programmatically. If you forget to explicitly change the title to something more descriptive, however, your page will have the default title, "Untitled Page". Ideally, the page's title would be set automatically for us in the event that we don't explicitly specify its value. For example, if at runtime the page's title is "Untitled Page", we might want to have the title automatically updated to be the same as the ASP.NET page's filename. The good news is that with a little bit of upfront work it is possible to have the title automatically assigned.

All ASP.NET web pages derive from the `Page` class in the System.Web.UI namespace. The `Page` class defines the minimal functionality needed by an ASP.NET page and exposes essential properties like `IsPostBack`, `IsValid`, `Request`, and `Response`, among many others. Oftentimes, every page in a web application requires additional features or functionality. A common way of providing this is to create a custom base page class. A custom base page class is a class you create that derives from the `Page` class and includes additional functionality. Once this base class has been created, you can have your ASP.NET pages derive from it (rather than the `Page` class), thereby offering the extended functionality to your ASP.NET pages.

In this step we create a base page that automatically sets the page's title to the ASP.NET page's filename if the title has not otherwise been explicitly set. Step 3 looks at setting the page's title based on the site map.

> **NOTE**
> A thorough examination of creating and using custom base page classes is beyond the scope of this tutorial series. For more information, read Using a Custom Base Class for Your ASP.NET Pages' Code-Behind Classes.

Creating the Base Page Class

Our first task is to create a base page class, which is a class that extends the `Page` class. Start by adding an `App_Code` folder to your project by right-clicking on the project name in the Solution Explorer, choosing Add ASP.NET Folder, and then selecting `App_Code`. Next, right-click on the `App_Code` folder and add a new class named `BasePage.vb`. Figure 4 shows the Solution Explorer after the `App_Code` folder and `BasePage.vb` class have been added.

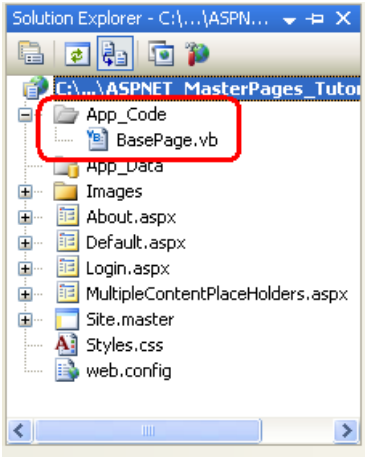


Figure 04: Add an `App_Code` Folder and a Class Named `BasePage`

> **NOTE**
>
> Visual Studio supports two modes of project management: Web Site Projects and Web Application Projects. The `App_Code` folder is designed to be used with the Web Site Project model. If you are using the Web Application Project model, place the `BasePage.vb` class in a folder named something other than `App_Code`, such as `Classes`. For more information on this topic, refer to Migrating a Web Site Project to a Web Application Project.

Because the custom base page serves as the base class for ASP.NET pages' code-behind classes, it needs to extend the `Page` class.

```
Public Class BasePage
 Inherits System.Web.UI.Page

End Class
```

Whenever an ASP.NET page is requested it proceeds through a series of stages, culminating in the requested page being rendered into HTML. We can tap into a stage by overriding the `Page` class's `OnEvent` method. For our base page let's automatically set the title if it has not been explicitly specified by the `LoadComplete` stage (which, as you might have guessed, occurs after the `Load` stage).

To accomplish this, override the `OnLoadComplete` method and enter the following code:

```
Protected Overrides Sub OnLoadComplete(ByVal e As EventArgs)
 ' Set the page's title, if necessary
 If String.IsNullOrEmpty(Page.Title) OrElse Page.Title = "Untitled Page" Then
 ' Determine the filename for this page
 Dim fileName As String = System.IO.Path.GetFileNameWithoutExtension(Request.PhysicalPath)

 Page.Title = fileName
 End If

 MyBase.OnLoadComplete(e)
End Sub
```

The `OnLoadComplete` method starts by determining if the `Title` property has not yet been explicitly set. If the `Title` property is `Nothing`, an empty string, or has the value "Untitled Page", it is assigned to the filename of the requested ASP.NET page. The physical path to the requested ASP.NET page - `C:\MySites\Tutorial03\Login.aspx`, for example - is accessible via the `Request.PhysicalPath` property. The `Path.GetFileNameWithoutExtension` method is used to pull out just the filename portion, and this filename is then assigned to the `Page.Title` property.

> **NOTE**
> I invite you to enhance this logic to improve the format of the title. For example, if the page's filename is `Company-Products.aspx`, the above code will produce the title "Company-Products", but ideally the dash would be replaced with a space, as in "Company Products". Also, consider adding a space whenever there's a case change. That is, consider adding code that transforms the filename `OurBusinessHours.aspx` to a title of "Our Business Hours".

Having the Content Pages Inherit the Base Page Class

We now need to update the ASP.NET pages in our site to derive from the custom base page (`BasePage`) instead of the `Page` class. To accomplish this go to each code-behind class and change the class declaration from:

```
Partial Class About
 Inherits System.Web.UI.Page
   ...
End Class
```

To:

```
Partial Class About
 Inherits BasePage
   ...
End Class
```

After doing so, visit the site through a browser. If you visit a page whose title is explicitly set, such as `Default.aspx` or `About.aspx`, the explicitly specified title is used. If, however, you visit a page whose title has not been changed from the default ("Untitled Page"), the base page class sets the title to the page's filename.

Figure 5 shows the `MultipleContentPlaceHolders.aspx` page when viewed through a browser. Note that the title is precisely the page's filename (less the extension), "MultipleContentPlaceHolders".

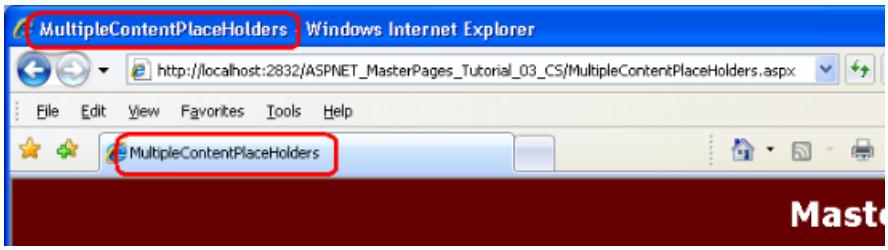


Figure 05: If a Title is Not Explicitly Specified, the Page's Filename is Automatically Used (Click to view full-size image)

Step 3: Basing the Page Title on the Site Map

ASP.NET offers a robust site map framework that enables page developers to define a hierarchical site map in an external resource (such as an XML file or database table) along with Web controls for displaying information about the site map (such as the SiteMapPath, Menu, and TreeView controls).

The site map structure can also be accessed programmatically from an ASP.NET page's code-behind class. In this manner we can automatically set a page's title to the title of its corresponding node in the site map. Let's enhance the `BasePage` class created in Step 2 so that it offers this functionality. But first we need to create a site map for our site.

> **NOTE**
>
> This tutorial assumes the reader already is familiar with ASP.NET's site map features. For more information on using the site map, consult my multi-part article series, Examining ASP.NET's Site Navigation.

Creating the Site Map

The site map system is built atop the provider model, which decouples the site map API from the logic that serializes site map information between memory and a persistent store. The .NET Framework ships with the `XmlSiteMapProvider` class, which is the default site map provider. As its name implies, `XmlSiteMapProvider` uses an XML file as its site map store. Let's use this provider for defining our site map.

Start by creating a site map file in the website's root folder named `Web.sitemap`. To accomplish this, right-click on the website name in Solution Explorer, choose Add New Item, and select the Site Map template. Ensure that the file is named `Web.sitemap` and click Add.

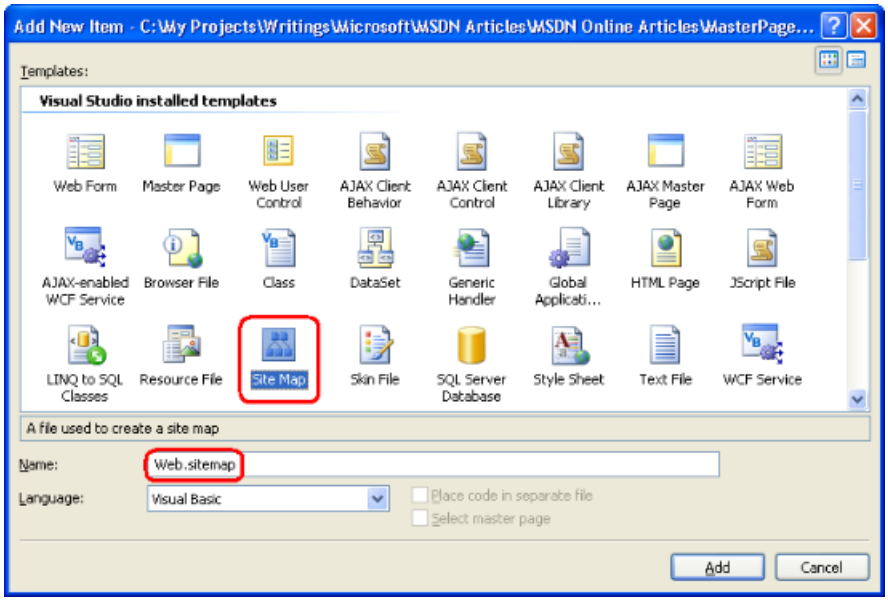


Figure 06: Add a File Named `Web.sitemap` to the Website's Root Folder (Click to view full-size image)

Add the following XML to the `Web.sitemap` file:

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
 <siteMapNode url="~/Default.aspx" title="Home">
 <siteMapNode url="~/About.aspx" title="About the Author" />
 <siteMapNode url="~/MultipleContentPlaceHolders.aspx" title="Using Multiple ContentPlaceHolder Controls" />
 </siteMapNode>
</siteMap>
```

This XML defines the hierarchical site map structure shown in Figure 7.

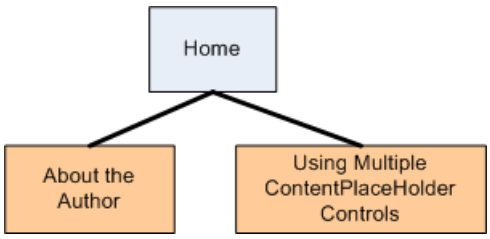


Figure 07: The Site Map is Currently Composed of Three Site Map Nodes

We will update the site map structure in future tutorials as we add new examples.

Updating the Master Page to Include Navigation Web Controls

Now that we have a site map defined, let's update the master page to include navigation Web controls. Specifically, let's add a ListView control to the left column in the Lessons section that renders an unordered list with a list item for each node defined in the site map.

> **NOTE**
>
> The ListView control is new to ASP.NET version 3.5. If you are using a prior version of ASP.NET, use the Repeater control instead. For more information on the ListView control, see Using ASP.NET 3.5's ListView and DataPager Controls.

Start by removing the existing unordered list markup from the Lessons section. Next, drag a ListView control from the Toolbox and drop it beneath the Lessons heading. The ListView is located in the Data section of the Toolbox, alongside the other view controls: the GridView, DetailsView, and FormView. Set the ListView's `ID` property to `LessonsList`.

From the Data Source Configuration Wizard choose to bind the ListView to a new SiteMapDataSource control named `LessonsDataSource`. The SiteMapDataSource control returns the hierarchical structure from the site map system.

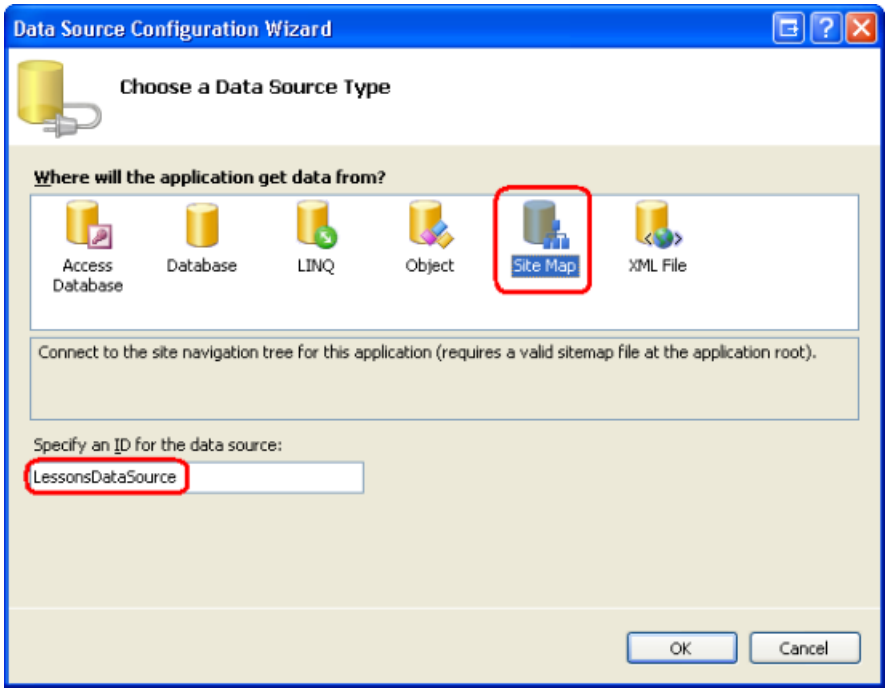


Figure 08: Bind a SiteMapDataSource Control to the LessonsList ListView Control (Click to view full-size image)

After creating the SiteMapDataSource control, we need to define the ListView's templates so that it renders an unordered list with a list item for each node returned by the SiteMapDataSource control. This can be accomplished using the following template markup:

```
<asp:ListView ID="LessonsList" runat="server" DataSourceID="LessonsDataSource">
 <LayoutTemplate>
 <ul>
 <asp:PlaceHolder runat="server" ID="itemPlaceholder" />
 </ul>
 </LayoutTemplate>

 <ItemTemplate>
 <li><asp:HyperLink runat="server" ID="lnkLesson" NavigateUrl='<%# Eval("Url") %>'
 Text='<%# Eval("Title") %>' /></li>
 </ItemTemplate>
</asp:ListView>
```

The `LayoutTemplate` generates the markup for an unordered list (`<ul>...</ul>`) while the `ItemTemplate` renders each item returned by the SiteMapDataSource as a list item (`<li>`) that contains a link to the particular lesson.

After configuring the ListView's templates, visit the website. As Figure 9 shows, the Lessons section contains a single bulleted item, Home. Where are the About and Using Multiple ContentPlaceHolder Controls lessons? The SiteMapDataSource is designed to return a hierarchical set of data, but the ListView control can only display a single level of the hierarchy. Consequently, only the first level of site map nodes returned by the SiteMapDataSource is displayed.

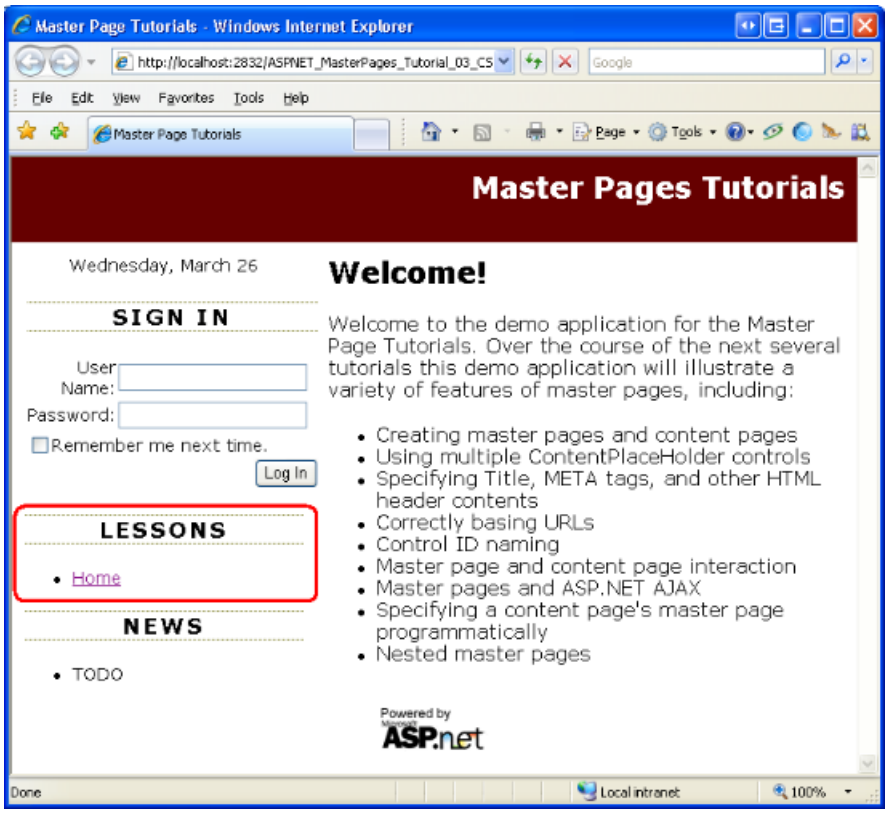


Figure 09: The Lessons Section Contains a Single List Item (Click to view full-size image)

To display multiple levels we could nest multiple ListViews within the `ItemTemplate`. This technique was examined in the *Master Pages and Site Navigation* tutorial of my Working with Data tutorial series. However, for this tutorial series our site map will contain just a two levels: Home (the top level); and each lesson as a child of Home. Rather than crafting a nested ListView, we can instead instruct the SiteMapDataSource to not return the starting node by setting its `ShowStartingNode` property to `False`. The net effect is that the SiteMapDataSource starts by returning the second tier of site map nodes.

With this change, the ListView displays bullet items for the About and Using Multiple ContentPlaceHolder Controls lessons, but omits a bullet item for Home. To remedy this, we can explicitly add a bullet item for Home in the `LayoutTemplate`:

```
<LayoutTemplate>
 <ul>
 <li><asp:HyperLink runat="server" ID="lnkLesson"
 NavigateUrl="~/Default.aspx" Text="Home" /></li>
 <asp:PlaceHolder runat="server" ID="itemPlaceholder" />
 </ul>
</LayoutTemplate>
```

By configuring the SiteMapDataSource to omit the starting node and explicitly adding a Home bullet item, the Lessons section now displays the intended output.

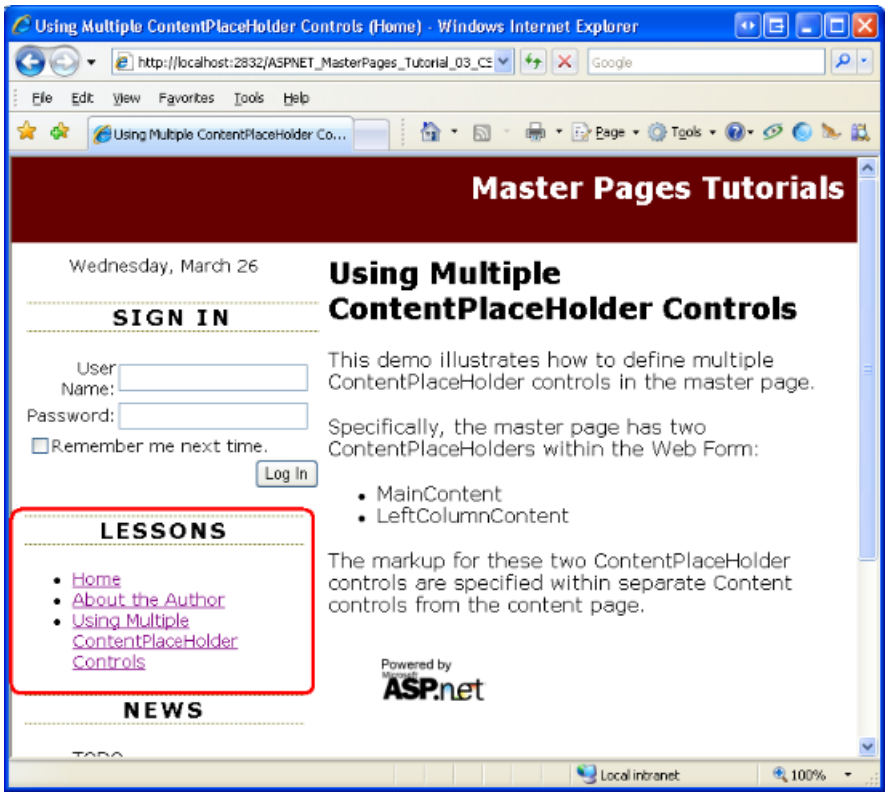


Figure 10: The Lessons Section Contains a Bullet Item for Home and Each Child Node (Click to view full-size image)

Setting the Title Based on the Site Map

With the site map in place, we can update our `BasePage` class to use the title specified in the site map. As we did in Step 2, we only want to use the site map node's title if the page's title has not been explicitly set by the page developer. If the page being requested does not have an explicitly set page title and is not found in the site map, then we'll fall back to using the requested page's filename (less the extension), as we did in Step 2. Figure 11 illustrates this decision process.

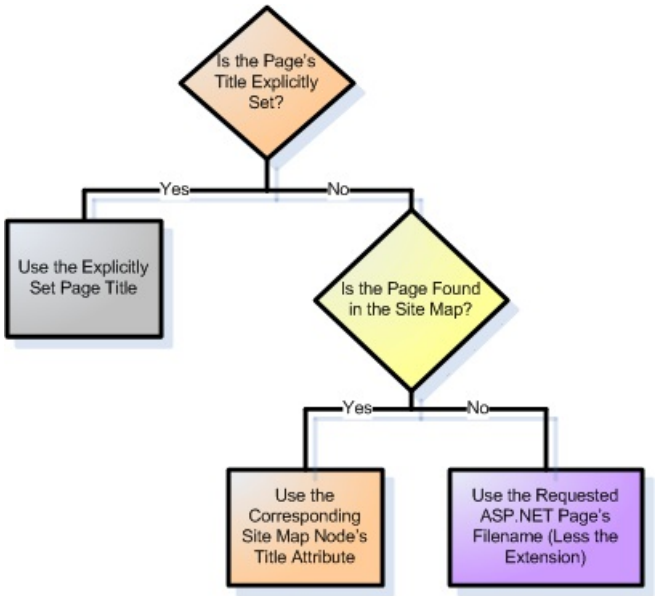


Figure 11: In the Absence of an Explicitly Set Page Title, the Corresponding Site Map Node's Title is Used

Update the `BasePage` class's `OnLoadComplete` method to include the following code:

```
Protected Overrides Sub OnLoadComplete(ByVal e As EventArgs)
 ' Set the page's title, if necessary
 If String.IsNullOrEmpty(Page.Title) OrElse Page.Title = "Untitled Page" Then
 ' Is this page defined in the site map?
 Dim newTitle As String = Nothing

 Dim current As SiteMapNode = SiteMap.CurrentNode
 If current IsNot Nothing Then
 newTitle = current.Title
 Else
 ' Determine the filename for this page
 newTitle = System.IO.Path.GetFileNameWithoutExtension(Request.PhysicalPath)
 End If

 Page.Title = newTitle
 End If

 MyBase.OnLoadComplete(e)
End Sub
```

As before, the `OnLoadComplete` method starts by determining whether the page's title has been explicitly set. If `Page.Title` is `Nothing`, an empty string, or is assigned the value "Untitled Page" then the code automatically assigns a value to `Page.Title`.

To determine the title to use, the code starts by referencing the `SiteMap` class's `CurrentNode` property. `CurrentNode` returns the `SiteMapNode` instance in the site map that corresponds to the currently requested page. Assuming the currently requested page is found within the site map, the `SiteMapNode`'s `Title` property is assigned to the page's title. If the currently requested page is not in the site map, `CurrentNode` returns `Nothing` and the requested page's filename is used as the title (as was done in Step 2).

Figure 12 shows the `MultipleContentPlaceHolders.aspx` page when viewed through a browser. Because this page's title is not explicitly set, its corresponding site map node's title is used instead.

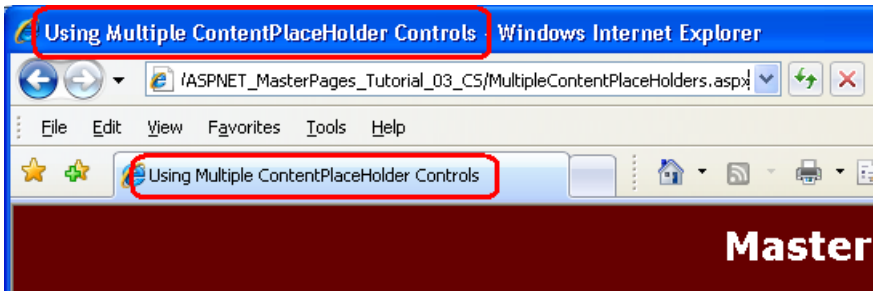


Figure 12: The MultipleContentPlaceHolders.aspx Page's Title is Pulled from the Site Map

Step 4: Adding Other Page-Specific Markup to the `<head>` Section

Steps 1, 2, and 3 looked at customizing the `<title>` element on a page-by-page basis. In addition to `<title>`, the `<head>` section may contain `<meta>` elements and `<link>` elements. As noted earlier in this tutorial, `Site.master`'s `<head>` section includes a `<link>` element to `Styles.css`. Because this `<link>` element is defined within the master page, it is included in the `<head>` section for all content pages. But how do we go about adding `<meta>` and `<link>` elements on a page-by-page basis?

The easiest way to add page-specific content to the `<head>` section is by creating a ContentPlaceHolder control in the master page. We already have such a ContentPlaceHolder (named `head`). Therefore, to add custom `<head>` markup, create a corresponding Content control in the page and place the markup there.

To illustrate adding custom `<head>` markup to a page, let's include a `<meta>` description element to our current set of content pages. The `<meta>` description element provides a brief description about the web page; most

search engines incorporate this information in some form when displaying search results.

A `<meta>` description element has the following form:

```
<meta name="description" content="description of the web page" />
```

To add this markup to a content page, add the above text to the Content control that maps to the master page's `head` ContentPlaceHolder. For example, to define a `<meta>` description element for `Default.aspx`, add the following markup:

```
<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
 <meta name="description" content="Welcome to Scott Mitchell's Master Page Tutorials series." />
</asp:Content>
```

Because the `head` ContentPlaceHolder is not within the HTML page's body, the markup added to the Content control is not displayed in the Design view. To see the `<meta>` description element visit `Default.aspx` through a browser. After the page has been loaded, view the source and note that the `<head>` section includes the markup specified in the Content control.

Take a moment to add `<meta>` description elements to `About.aspx`, `MultipleContentPlaceHolders.aspx`, and `Login.aspx`.

Programmatically Adding Markup to the `<head>` Region

The `head` ContentPlaceHolder allows us to declaratively add custom markup to the master page's `<head>` region. Custom markup may also be added programmatically. Recall that the `Page` class's `Header` property returns the `HtmlHead` instance defined in the master page (the `<head runat="server">`).

Being able to programmatically add content to the `<head>` region is useful when the content to add is dynamic. Perhaps it's based on the user visiting the page; maybe it's being pulled from a database. Regardless of the reason, you can add content to the `HtmlHead` by adding controls to its `Controls` collection like so:

```
' Programmatically add a <meta> element to the Header
Dim keywords As New HtmlMeta()
keywords.Name = "keywords"
keywords.Content = "master page,asp.net,tutorial"

Page.Header.Controls.Add(keywords)
```

The above code adds the `<meta>` keywords element to the `<head>` region, which provides a comma-delimited list of keywords that describe the page. Note that to add a `<meta>` tag you create an `HtmlMeta` instance, set its `Name` and `Content` properties, and then add it to the `Header`'s `Controls` collection. Similarly, to programmatically add a `<link>` element, create an `HtmlLink` object, set its properties, and then add it to the `Header`'s `Controls` collection.

> **NOTE**
>
> To add arbitrary markup, create a `LiteralControl` instance, set its `Text` property, and then add it to the `Header`'s `Controls` collection.

Summary

In this tutorial we looked at a variety of ways to add `<head>` region markup on a page-by-page basis. A master page should include an `HtmlHead` instance (`<head runat="server">`) with a ContentPlaceHolder. The `HtmlHead`

instance allows content pages to programmatically access the `<head>` region and to declaratively and programmatically set the page's title; the ContentPlaceHolder control enables custom markup to be added to the `<head>` section declaratively through a Content control.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Dynamically Setting the Page's Title in ASP.NET
- Examining ASP.NET's Site Navigation
- How To Use HTML Meta Tags
- Master Pages in ASP.NET
- Using ASP.NET 3.5's ListView and DataPager Controls
- Using a Custom Base Class for Your ASP.NET Pages' Code-Behind Classes

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Zack Jones and Suchi Banerjee. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

URLs in Master Pages (VB)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Addresses how URLs in the master page can break due to the master page file being in a different relative directory than the content page. Looks at rebasing URLs via ~ in the declarative syntax and using ResolveUrl and ResolveClientUrl programmatically. (Also look at

Introduction

In all the examples we've seen thus far, the master and content pages have been located in the same folder (the website's root folder). But there's no reason why the master and content pages must be in the same folder. You can certainly create content pages in subfolders. Similarly, you might create a `~/MasterPages/` folder where you place your site's master pages.

One potential issue with placing master and content pages in different folders involves broken URLs. If the master page contains relative URLs in hyperlinks, images, or other elements, the link will be invalid for content pages that reside in a different folder. In this tutorial we examine the source of this problem as well as workarounds.

The Problem with Relative URLs

A URL on a web page is said to be a *relative URL* if the location of the resource it points to is relative to the web page's location in the website's folder structure. Any URL that does not start with a leading forward slash (`/`) or a protocol (such as `http://`) is relative because it is resolved by the browser based on the location of the web page that contains the URL.

For example, our website has an `~/Images/` folder with a single image file, `PoweredByASPNET.gif`. The master page file `Site.master` has an `<img>` element in the `footerContent` region with the following markup:

```
<div id="footerContent">
 <img src="Images/PoweredByASPNET.gif" alt="Powered by ASP.NET!" />
</div>
```

The `src` attribute value in the `<img>` element is a relative URL because it does not start with `/` or `http://`. In short, the `src` attribute value tells the browser to look in the `Images` subfolder for a file named `PoweredByASPNET.gif`.

When visiting a content page, the above markup is sent directly to the browser. Take a moment to visit `About.aspx` and view the HTML source sent to the browser. You will find that the exact same markup in the master page was sent to the browser.

```
<div id="footerContent">
 <img src="Images/PoweredByASPNET.gif" alt="Powered by ASP.NET!" />
</div>
```

If the content page is in the root folder (as is `About.aspx`) everything works as expected because there is an `Images` subfolder relative to the root folder. However, things break down if the content page is in a different folder

than the master page. To illustrate this, create a subfolder named `Admin`. Next, add a content page named `Default.aspx` to the `Admin` folder, making sure to bind the new page to the `Site.master` master page.

> **NOTE**
>
> In the *Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page* tutorial we created a custom base page class named `BasePage` that automatically set the content page's title (if it was not explicitly assigned). Don't forget to have the newly created page's code-behind class derive from `BasePage` so that it can utilize this functionality.

After you have created this content page, your Solution Explorer should look similar to Figure 1.

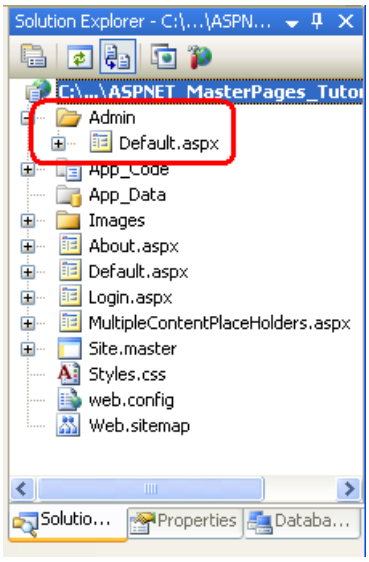


Figure 01: A New Folder and ASP.NET Page Have Been Added to the Project

Next, update the `Web.sitemap` file to include a new `<siteMapNode>` entry for this lesson. The following XML shows the complete `Web.sitemap` markup, which now includes the addition of a third `<siteMapNode>` element.

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
 <siteMapNode url="~/Default.aspx" title="Home">
 <siteMapNode url="~/About.aspx" title="About the Author" />
 <siteMapNode url="~/MultipleContentPlaceHolders.aspx" title="Using Multiple ContentPlaceHolder Controls" />
 <siteMapNode url="~/Admin/Default.aspx" title="Rebasing URLs" />
 </siteMapNode>
</siteMap>
```

The newly created `Default.aspx` page should have four Content controls corresponding to the four ContentPlaceHolders in `Site.master`. Add some text to the Content control referencing the `MainContent` ContentPlaceHolder and then visit the page through a browser. As Figure 2 shows, the browser cannot find the `PoweredByASPNET.gif` image file. What's going on here?

The `~/Admin/Default.aspx` content page is sent the same HTML for the `footerContent` region as was the `About.aspx` page:

```
<div id="footerContent">
 <img src="Images/PoweredByASPNET.gif" alt="Powered by ASP.NET!" />
</div>
```

Because the `<img>` element's `src` attribute is a relative URL, the browser attempts to look for an `Images` folder relative to the web page's folder location. In other words, the browser is looking for the image file `Admin/Images/PoweredByASPNET.gif`.

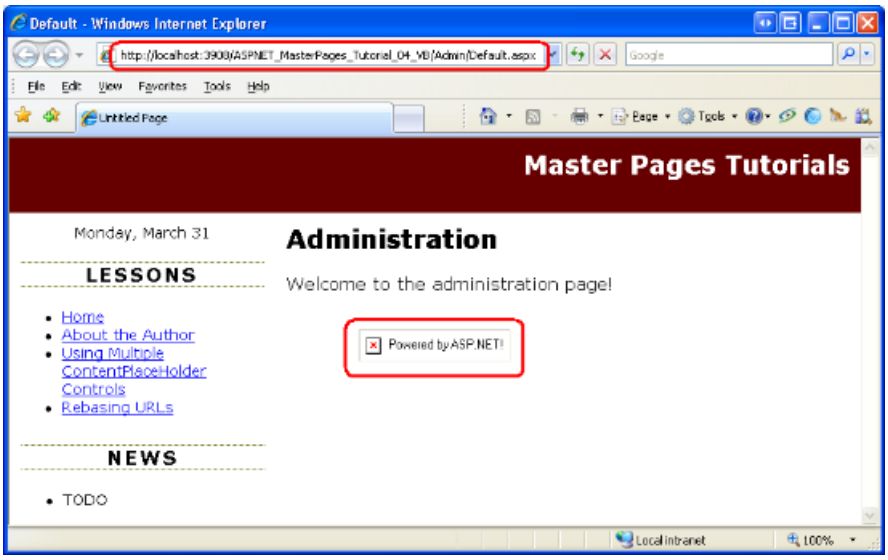


Figure 02: The `PoweredByASPNET.gif` Image File Cannot Be Found (Click to view full-size image)

Replacing Relative URLs with Absolute URLs

The opposite of a relative URL is an *absolute URL*, which is one that starts with a forward slash (`/`) or a protocol such as `http://`. Because an absolute URL specifies the location of a resource from a known fixed point, the same absolute URL is valid in any web page, regardless of the web page's location in the website's folder structure.

To remedy the broken image shown in Figure 2, we need to update the `<img>` element's `src` attribute so that it uses an absolute URL instead of a relative one. To determine the correct absolute URL, visit one of the web pages in your website and examine the Address bar. As the Address bar in Figure 2 shows, the fully qualified path to the web application is `http://localhost:3908/ASPNET_MasterPages_Tutorial_04_VB/`. Therefore, we could update the `<img>` element's `src` attribute to either of the following two absolute URLs:

- `/ASPNET_MasterPages_Tutorial_04_VB/Images/PoweredByASPNET.gif`
- `http://localhost:3908/ASPNET_MasterPages_Tutorial_04_VB/Images/PoweredByASPNET.gif`

Take a moment to update the `<img>` element's `src` attribute to an absolute URL using one of the forms shown above and then visit the `~/Admin/Default.aspx` page through a browser. This time the browser will correctly find and display the `PoweredByASPNET.gif` image file (see Figure 3).

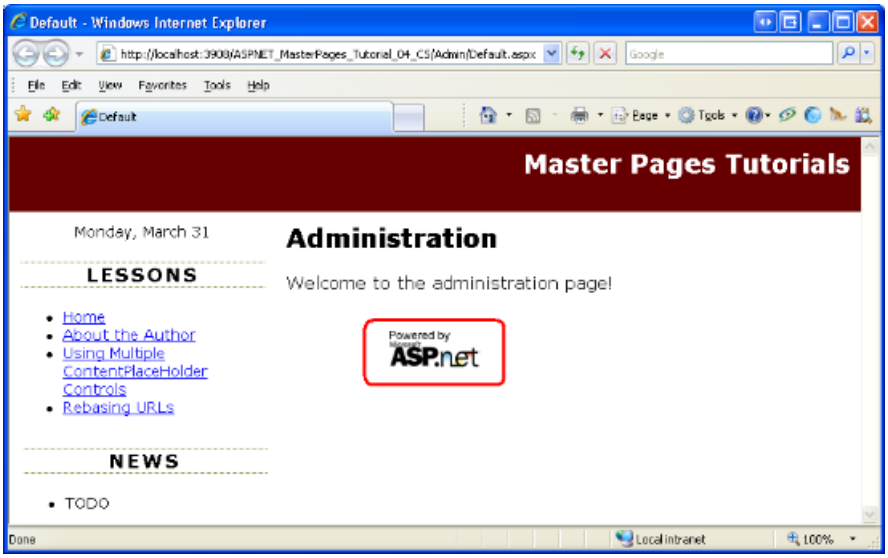


Figure 03: The `PoweredByASPNET.gif` Image is Now Displayed (Click to view full-size image)

While hard coding in the absolute URL works, it tightly couples your HTML to the website's server and folder location, which may change. Using an absolute URL of the form `http://localhost:3908/...` is brittle because the port number preceding localhost is selected automatically each time Visual Studio's built-in ASP.NET Development

Web Server is started. Similarly, the `http://localhost` part is only valid when testing locally. Once the code is deployed to a production server, the URL base will change to something else, like `http://www.yourserver.com`. The absolute URL in the form `/ASPNET_MasterPages_Tutorial_04_VB/...` also suffers from the same brittleness because oftentimes this application path differs between development and production servers.

The good news is that ASP.NET offers a method for generating a valid relative URL at runtime.

Using `~` and `ResolveClientUrl`

Rather than hard code an absolute URL, ASP.NET allows page developers to use the tilde (`~`) to indicate the root of the web application. For example, earlier in this tutorial I used the notation `~/Admin/Default.aspx` in the text to refer to the `Default.aspx` page in the `Admin` folder. The `~` indicates that the `Admin` folder is a subfolder of the web application's root.

The `Control` class's `ResolveClientUrl` method takes a URL and modifies it to a relative URL appropriate for the web page on which the control resides. For example, calling `ResolveClientUrl("~/Images/PoweredByASPNET.gif")` from `About.aspx` returns `Images/PoweredByASPNET.gif`. Calling it from `~/Admin/Default.aspx`, however, returns `../Images/PoweredByASPNET.gif`.

> **NOTE**
> Because all ASP.NET server controls derive from the `Control` class, all server controls have access to the `ResolveClientUrl` method. Even the `Page` class derives from the `Control` class, meaning that you can use this method directly from your ASP.NET pages' code-behind classes.

Using `~` in the Declarative Markup

Several ASP.NET Web controls include URL-related properties: the HyperLink control has a `NavigateUrl` property; the Image control has an `ImageUrl` property; and so on. When rendered, these controls pass their URL-related property values to `ResolveClientUrl`. Consequently, if these properties contain a `~` to indicate the root of the web application, the URL will be modified to a valid relative URL.

Keep in mind that only ASP.NET server controls transform the `~` in their URL-related properties. If a `~` appears in static HTML markup, such as `<img src="~/Images/PoweredByASPNET.gif" />`, the ASP.NET engine sends the `~` to the browser along with the rest of the HTML content. The browser assumes that the `~` is part of the URL. For example, if the browser receives the markup `<img src="~/Images/PoweredByASPNET.gif" />` it assumes there is a subfolder named `~` with a subfolder `Images` that contains the image file `PoweredByASPNET.gif`.

To fix the image markup in `Site.master`, replace the existing `<img>` element with an ASP.NET Image Web control. Set the Image Web control's `ID` to `PoweredByImage`, its `ImageUrl` property to `~/Images/PoweredByASPNET.gif`, and its `AlternateText` property to "Powered by ASP.NET!"

```
<div id="footerContent">
 <asp:Image ID="PoweredByImage" runat="server" ImageUrl="~/Images/PoweredByASPNET.gif"
    AlternateText="Powered by ASP.NET!" />
</div>
```

After making this change to the master page, revisit the `~/Admin/Default.aspx` page again. This time the `PoweredByASPNET.gif` image file appears in the page (see Figure 3). When the Image Web control is rendered it uses the `ResolveClientUrl` method to resolve its `ImageUrl` property value. In `~/Admin/Default.aspx` the `ImageUrl` is converted into an appropriate relative URL, as the following snippet of HTML source shows:

```
<div id="footerContent">
 <img id="ctl00_PoweredByImage" src="../Images/PoweredByASPNET.gif" alt="Powered by ASP.NET!" />
</div>
```

NOTE

In addition to being used in URL-based Web control properties, the `~` can also be used when calling the `Response.Redirect` and `Server.MapPath` methods, among others. Also, the `ResolveClientUrl` method may be invoked directly from an ASP.NET or master page's declarative markup, if needed; see Fritz Onion's blog entry Using `ResolveClientUrl` in Markup.

Fixing the Master Page's Remaining Relative URLs

In addition to the `<img>` element in the `footerContent` that we just fixed, the master page contains one more relative URL that requires our attention. The `topContent` region includes the link "Master Pages Tutorials," which points to `Default.aspx`.

```
<div id="topContent">
 <a href="Default.aspx">Master Pages Tutorials</a>
</div>
```

Because this URL is relative, it will send the user to the `Default.aspx` page in the folder of the content page they are visiting. To have this link always point to `Default.aspx` in the root folder we need to replace the `<a>` element with a HyperLink Web control so that we can use the `~` notation.

Remove the `<a>` element markup and add a HyperLink control in its place. Set the HyperLink's `ID` to `lnkHome`, its `NavigateUrl` property to `~/Default.aspx`, and its `Text` property to "Master Pages Tutorials."

```
<div id="topContent">
 <asp:HyperLink ID="lnkHome" runat="server" NavigateUrl="~/Default.aspx"
    Text="Master Pages Tutorials" />
</div>
```

That's it! At this point all the URLs in our master page are properly based when rendered by a content page regardless of what folders the master page and content page are located in.

Automatic URL Resolution in the `<head>` Section

In the *Creating a Site-Wide Layout Using Master Pages* tutorial we added a `<link>` to the `Styles.css` file in the `<head>` region:

```
<head runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
 <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
```

While the `<link>` element's `href` attribute is relative, it's automatically converted to an appropriate path at runtime. As we discussed in the *Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page* tutorial, the `<head>` region is actually a server-side control, which enables it to modify the contents of its inner controls when it is rendered.

To verify this, revisit the `~/Admin/Default.aspx` page and view the HTML source sent to the browser. As the snippet

below illustrates, the `<link>` element's `href` attribute has been automatically modified to an appropriate relative URL, `../Styles.css`.

```
<head>
 <title>
 Default
 </title>
 <link href="../Styles.css" rel="stylesheet" type="text/css" />
</head>
```

Summary

Master pages very often include links, images, and other external resources that must be specified via a URL. Because the master page and content pages might not exist in the same folder, it is important to abstain from using relative URLs. While it is possible to use hard coded absolute URLs, doing so tightly couples the absolute URL to the web application. If the absolute URL changes - as it often does when moving or deploying a web application - you'll have to remember to go back and update the absolute URLs.

The ideal approach is to use the tilde (`~`) to indicate the application root. ASP.NET Web controls that contain URL-related properties map the `~` to the application root at runtime. Internally, the Web controls use the `Control` class's `ResolveClientUrl` method to generate a valid relative URL. This method is public and available from every server control (including the `Page` class), so you can use it programmatically from your code-behind classes, if needed.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Master Pages in ASP.NET
- URL Rebasing in a Master Page
- Using `ResolveClientUrl` in Markup

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Control ID Naming in Content Pages (VB)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Illustrates how ContentPlaceHolder controls serve as a naming container and therefore make programmatically working with a control difficult (via FindControl). Looks at this issue and workarounds. Also discusses how to programmatically access the resulting ClientID value.

Introduction

All ASP.NET server controls include an `ID` property that uniquely identifies the control and is the means by which the control is programmatically accessed in the code-behind class. Similarly, the elements in an HTML document may include an `id` attribute that uniquely identifies the element; these `id` values are often used in client-side script to programmatically reference a particular HTML element. Given this, you may assume that when an ASP.NET server control is rendered into HTML, its `ID` value is used as the `id` value of the rendered HTML element. This is not necessarily the case because in certain circumstances a single control with a single `ID` value may appear multiple times in the rendered markup. Consider a GridView control that includes a TemplateField with a Label Web control with an `ID` value of `ProductName`. When the GridView is bound to its data source at runtime, this Label is repeated once for every GridView row. Each rendered Label needs a unique `id` value.

To handle such scenarios, ASP.NET allows certain controls to be denoted as naming containers. A naming container serves as a new `ID` namespace. Any server controls that appear within the naming container have their rendered `id` value prefixed with the `ID` of the naming container control. For example, the `GridView` and `GridViewRow` classes are both naming containers. Consequently, a Label control defined in a GridView TemplateField with `ID` `ProductName` is given a rendered `id` value of `GridViewID_GridViewRowID_ProductName`. Because *GridViewRowID* is unique for each GridView row, the resulting `id` values are unique.

> **NOTE**
>
> The `INamingContainer` interface is used to indicate that a particular ASP.NET server control should function as a naming container. The `INamingContainer` interface does not spell out any methods that the server control must implement; rather, it's used as a marker. In generating the rendered markup, if a control implements this interface then the ASP.NET engine automatically prefixes its `ID` value to its descendents' rendered `id` attribute values. This process is discussed in more detail in Step 2.

Naming containers not only change the rendered `id` attribute value, but also affect how the control may be programmatically referenced from the ASP.NET page's code-behind class. The `FindControl("controlID")` method is commonly used to programmatically reference a Web control. However, `FindControl` does not penetrate through naming containers. Consequently, you cannot directly use the `Page.FindControl` method to reference controls within a GridView or other naming container.

As you may have surmised, master pages and ContentPlaceHolders are both implemented as naming containers. In this tutorial we examine how master pages affect HTML element `id` values and ways to programmatically reference Web controls within a content page using `FindControl`.

Step 1: Adding a New ASP.NET Page

To demonstrate the concepts discussed in this tutorial, let's add a new ASP.NET page to our website. Create a new content page named `IDIssues.aspx` in the root folder, binding it to the `Site.master` master page.

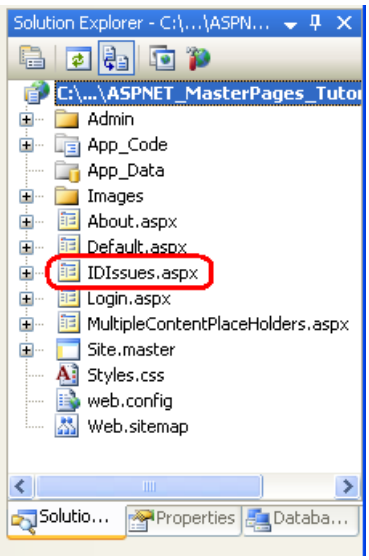


Figure 01: Add the Content Page `IDIssues.aspx` to the Root Folder

Visual Studio automatically creates a Content control for each of the master page's four ContentPlaceHolders. As noted in the *Multiple ContentPlaceHolders and Default Content* tutorial, if a Content control is not present the master page's default ContentPlaceHolder content is emitted instead. Because the `QuickLoginUI` and `LeftColumnContent` ContentPlaceHolders contain suitable default markup for this page, go ahead and remove their corresponding Content controls from `IDIssues.aspx`. At this point, the content page's declarative markup should look like the following:

```
<%@ Page Language="VB" MasterPageFile="~/Site.master" AutoEventWireup="false" CodeFile="IDIssues.aspx.vb"
Inherits="IDIssues" Title="Untitled Page" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
</asp:Content>
```

In the *Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page* tutorial we created a custom base page class (`BasePage`) that automatically configures the page's title if it is not explicitly set. For the `IDIssues.aspx` page to employ this functionality, the page's code-behind class must derive from the `BasePage` class (instead of `System.Web.UI.Page`). Modify the code-behind class's definition so that it looks like the following:

```
Partial Class IDIssues
 Inherits BasePage

End Class
```

Finally, update the `Web.sitemap` file to include an entry for this new lesson. Add a `<siteMapNode>` element and set its `title` and `url` attributes to "Control ID Naming Issues" and `~/IDIssues.aspx`, respectively. After making this addition your `Web.sitemap` file's markup should look similar to the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
 <siteMapNode url="~/Default.aspx" title="Home">
 <siteMapNode url="~/About.aspx" title="About the Author" />
 <siteMapNode url="~/MultipleContentPlaceHolders.aspx" title="Using Multiple ContentPlaceHolder Controls" />
 <siteMapNode url="~/Admin/Default.aspx" title="Rebasing URLs" />
 <siteMapNode url="~/IDIssues.aspx" title="Control ID Naming Issues" />
 </siteMapNode>
</siteMap>
```

As Figure 2 illustrates, the new site map entry in `Web.sitemap` is immediately reflected in the Lessons section in the left column.

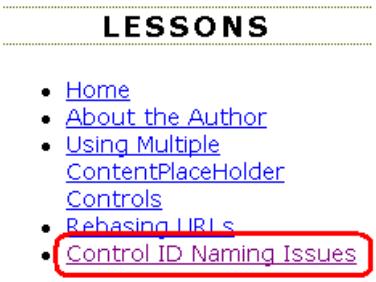


Figure 02: The Lessons Section Now Includes a Link to "Control ID Naming Issues"

Step 2: Examining the Rendered `ID` Changes

To better understand the modifications the ASP.NET engine makes to the rendered `id` values of server controls, let's add a few Web controls to the `IDIssues.aspx` page and then view the rendered markup sent to the browser. Specifically, type in the text "Please enter your age:" followed by a TextBox Web control. Further down on the page add a Button Web control and a Label Web control. Set the TextBox's `ID` and `Columns` properties to `Age` and 3, respectively. Set the Button's `Text` and `ID` properties to "Submit" and `SubmitButton`. Clear out the Label's `Text` property and set its `ID` to `Results`.

At this point your Content control's declarative markup should look similar to the following:

```
<p>
 Please enter your age:
 <asp:TextBox ID="Age" Columns="3" runat="server"></asp:TextBox>
</p>
<p>
 <asp:Button ID="SubmitButton" runat="server" Text="Submit" />
</p>
<p>
 <asp:Label ID="Results" runat="server"></asp:Label>
</p>
```

Figure 3 shows the page when viewed through Visual Studio's designer.

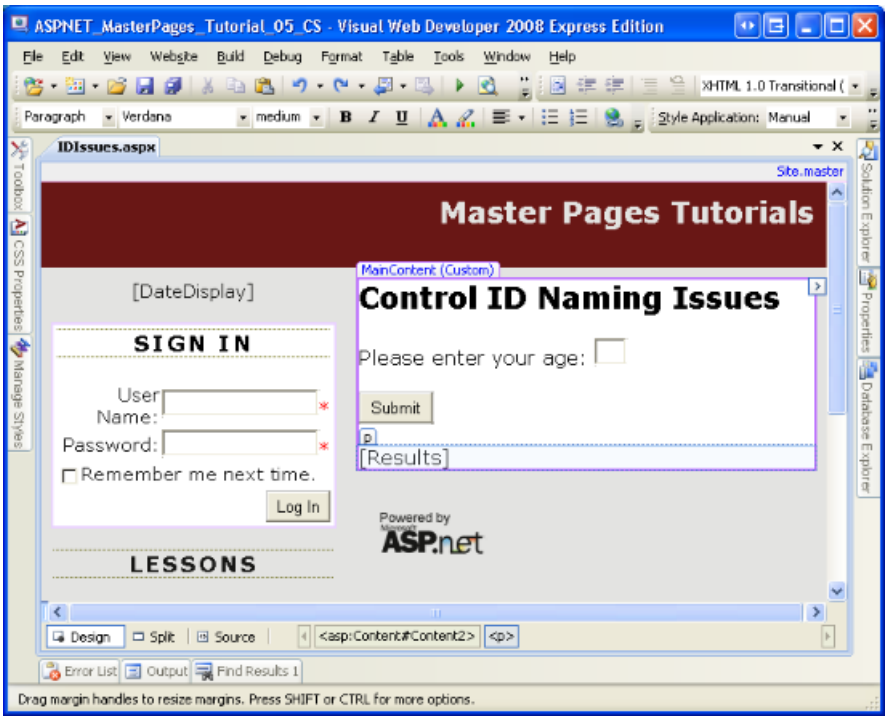


Figure 03: The Page Includes Three Web Controls: a TextBox, Button, and Label (Click to view full-size image)

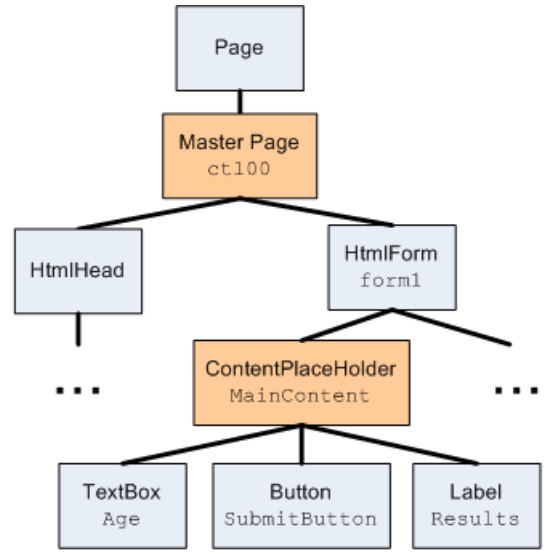
Visit the page through a browser and then view the HTML source. As the markup below shows, the `id` values of the HTML elements for the TextBox, Button, and Label Web controls are a combination of the `ID` values of the Web controls and the `ID` values of the naming containers in the page.

```
<p>
 Please enter your age:
 <input name="ctl00$MainContent$Age" type="text" size="3" id="ctl00_MainContent_Age" />
</p>
<p>

 <input type="submit" name="ctl00$MainContent$SubmitButton" value="Submit"
id="ctl00_MainContent_SubmitButton" />
</p>
<p>
 <span id="ctl00_MainContent_Results"></span>
</p>
```

As noted earlier in this tutorial, both the master page and its ContentPlaceHolders serve as naming containers. Consequently, both contribute the rendered `ID` values of their nested controls. Take the TextBox's `id` attribute, for instance: `ctl00_MainContent_Age`. Recall that the TextBox control's `ID` value was `Age`. This is prefixed with its ContentPlaceHolder control's `ID` value, `MainContent`. Furthermore, this value is prefixed with the master page's `ID` value, `ctl00`. The net effect is an `id` attribute value consisting of the `ID` values of the master page, the ContentPlaceHolder control, and the TextBox itself.

Figure 4 illustrates this behavior. To determine the rendered `id` of the `Age` TextBox, start with the `ID` value of the TextBox control, `Age`. Next, work your way up the control hierarchy. At each naming container (those nodes with a peach color), prefix the current rendered `id` with the naming container's `id`.



The rendered `id` attribute value for the `Age` TextBox is a combination of its `ID` value (`Age`), prepended with the `IDs` of its naming containers (`MainContent` and `ctl00`).

Figure 04: The Rendered `id` Attributes are Based On the `ID` Values of the Naming Containers

> **NOTE**
>
> As we discussed, the `ctl00` portion of the rendered `id` attribute constitutes the `ID` value of the master page, but you may be wondering how this `ID` value came about. We did not specify it anywhere in our master or content page. Most server controls in an ASP.NET page are added explicitly through the page's declarative markup. The `MainContent` ContentPlaceHolder control was explicitly specified in the markup of `Site.master`; the `Age` TextBox was defined `IDIssues.aspx`'s markup. We can specify the `ID` values for these types of controls through the Properties window or from the declarative syntax. Other controls, like the master page itself, are not defined in the declarative markup. Consequently, their `ID` values must be automatically generated for us. The ASP.NET engine sets the `ID` values at runtime for those controls whose IDs have not been explicitly set. It uses the naming pattern `ctlXX`, where *XX* is a sequentially increasing integer value.

Because the master page itself serves as a naming container, the Web controls defined in the master page also have altered rendered `id` attribute values. For example, the `DisplayDate` Label we added to the master page in the *Creating a Site-Wide Layout with Master Pages* tutorial has the following rendered markup:

```
<span id="ctl00_DateDisplay">current date</span>
```

Note that the `id` attribute includes both the master page's `ID` value (`ctl00`) and the `ID` value of the Label Web control (`DateDisplay`).

Step 3: Programmatically Referencing Web Controls via `FindControl`

Every ASP.NET server control includes a `FindControl("controlID")` method that searches the control's descendents for a control named *controlID*. If such a control is found, it is returned; if no matching control is found, `FindControl` returns `Nothing`.

`FindControl` is useful in scenarios where you need to access a control but you don't have a direct reference to it. When working with data Web controls like the GridView, for example, the controls within the GridView's fields are defined once in the declarative syntax, but at runtime an instance of the control is created for each GridView row. Consequently, the controls generated at runtime exist, but we do not have a direct reference available from the code-behind class. As a result we need to use `FindControl` to programmatically work with a specific control within the GridView's fields. (For more information on using `FindControl` to access the controls within a data

Web control's templates, see Custom Formatting Based Upon Data.) This same scenario occurs when dynamically adding Web controls to a Web Form, a topic discussed in Creating Dynamic Data Entry User Interfaces.

To illustrate using the `FindControl` method to search for controls within a content page, create an event handler for the `SubmitButton`'s `Click` event. In the event handler, add the following code, which programmatically references the `Age` TextBox and `Results` Label using the `FindControl` method and then displays a message in `Results` based on the user's input.

> **NOTE**
>
> Of course, we don't need to use `FindControl` to reference the Label and TextBox controls for this example. We could reference them directly via their `ID` property values. I use `FindControl` here to illustrate what happens when using `FindControl` from a content page.

```
Protected Sub SubmitButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
SubmitButton.Click
 Dim ResultsLabel As Label = CType(FindControl("Results"), Label)
 Dim AgeTextBox As TextBox = CType(Page.FindControl("Age"), TextBox)

 ResultsLabel.Text = String.Format("You are {0} years old!", AgeTextBox.Text)
End Sub
```

While the syntax used to call the `FindControl` method differs slightly in the first two lines of `SubmitButton_Click`, they are semantically equivalent. Recall that all ASP.NET server controls include a `FindControl` method. This includes the `Page` class, from which all ASP.NET code-behind classes must derive from. Therefore, calling `FindControl("controlID")` is equivalent to calling `Page.FindControl("controlID")`, assuming you haven't overridden the `FindControl` method in your code-behind class or in a custom base class.

After entering this code, visit the `IDIssues.aspx` page through a browser, enter your age, and click the "Submit" button. Upon clicking the "Submit" button a `NullReferenceException` is raised (see Figure 5).

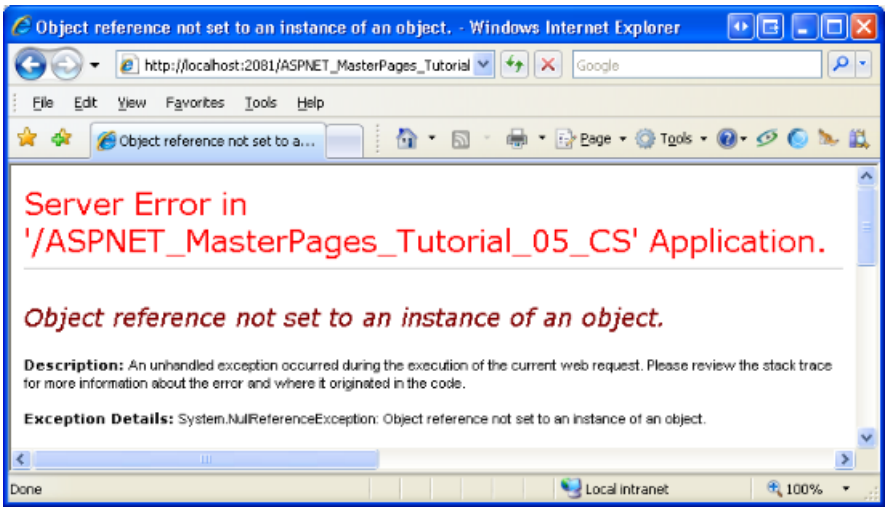


Figure 05: A `NullReferenceException` is Raised (Click to view full-size image)

If you set a breakpoint in the `SubmitButton_Click` event handler you will see that both calls to `FindControl` return `Nothing`. The `NullReferenceException` is raised when we attempt to access the `Age` TextBox's `Text` property.

The problem is that `Control.FindControl` only searches *Control*'s descendents that are in the same naming container. Because the master page constitutes a new naming container, a call to `Page.FindControl("controlID")` never permeates the master page object `ctl00`. (Refer back to Figure 4 to view the control hierarchy, which shows the `Page` object as the parent of the master page object `ctl00`.) Therefore, the `Results` Label and `Age`

TextBox are not found and `ResultsLabel` and `AgeTextBox` are assigned values of `Nothing`.

There are two workarounds to this challenge: we can drill down, one naming container at a time, to the appropriate control; or we can create our own `FindControl` method that permeates naming containers. Let's examine each of these options.

Drilling Into the Appropriate Naming Container

To use `FindControl` to reference the `Results` Label or `Age` TextBox, we need to call `FindControl` from an ancestor control in the same naming container. As Figure 4 showed, the `MainContent` ContentPlaceHolder control is the only ancestor of `Results` or `Age` that is within the same naming container. In other words, calling the `FindControl` method from the `MainContent` control, as shown in the code snippet below, correctly returns a reference to the `Results` or `Age` controls.

```
Dim ResultsLabel As Label = CType(MainContent.FindControl("Results"), Label)
Dim AgeTextBox As TextBox = CType(MainContent.FindControl("Age"), TextBox)
```

However, we cannot work with the `MainContent` ContentPlaceHolder from our content page's code-behind class using the above syntax because the ContentPlaceHolder is defined in the master page. Instead, we have to use `FindControl` to get a reference to `MainContent`. Replace the code in the `SubmitButton_Click` event handler with the following modifications:

```
Protected Sub SubmitButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
SubmitButton.Click
 Dim MainContent As ContentPlaceHolder = CType(FindControl("MainContent"), ContentPlaceHolder)

 Dim ResultsLabel As Label = CType(MainContent.FindControl("Results"), Label)
 Dim AgeTextBox As TextBox = CType(MainContent.FindControl("Age"), TextBox)

 ResultsLabel.Text = String.Format("You are {0} years old!", AgeTextBox.Text)
End Sub
```

If you visit the page through a browser, enter your age, and click the "Submit" button, a `NullReferenceException` is raised. If you set a breakpoint in the `SubmitButton_Click` event handler you will see that this exception occurs when attempting to call the `MainContent` object's `FindControl` method. The `MainContent` object is equal to `Nothing` because the `FindControl` method cannot locate an object named "MainContent". The underlying reason is the same as with the `Results` Label and `Age` TextBox controls: `FindControl` starts its search from the top of the control hierarchy and does not penetrate naming containers, but the `MainContent` ContentPlaceHolder is within the master page, which is a naming container.

Before we can use `FindControl` to get a reference to `MainContent`, we first need a reference to the master page control. Once we have a reference to the master page we can get a reference to the `MainContent` ContentPlaceHolder via `FindControl` and, from there, references to the `Results` Label and `Age` TextBox (again, through using `FindControl`). But how do we get a reference to the master page? By inspecting the `id` attributes in the rendered markup it's evident that the master page's `ID` value is `ctl00`. Therefore, we could use `Page.FindControl("ctl00")` to get a reference to the master page, then use that object to get a reference to `MainContent`, and so on. The following snippet illustrates this logic:

```
'Get a reference to the master page
Dim ctl00 As MasterPage = CType(FindControl("ctl00"), MasterPage)

'Get a reference to the ContentPlaceHolder
Dim MainContent As ContentPlaceHolder = CType(ctl00.FindControl("MainContent"), ContentPlaceHolder)

'Reference the Label and TextBox controls
Dim ResultsLabel As Label = CType(MainContent.FindControl("Results"), Label)
Dim AgeTextBox As TextBox = CType(MainContent.FindControl("Age"), TextBox)
```

While this code will certainly work, it assumes that the master page's autogenerated `ID` will always be `ctl00`. It's never a good idea to make assumptions about autogenerated values.

Fortunately, a reference to the master page is accessible through the `Page` class's `Master` property. Therefore, instead of having to use `FindControl("ctl00")` to get a reference of the master page in order to access the `MainContent` ContentPlaceHolder, we can instead use `Page.Master.FindControl("MainContent")`. Update the `SubmitButton_Click` event handler with the following code:

```
Protected Sub SubmitButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
SubmitButton.Click
 'Get a reference to the ContentPlaceHolder
 Dim MainContent As ContentPlaceHolder = CType(Page.Master.FindControl("MainContent"), ContentPlaceHolder)

 'Reference the Label and TextBox controls
 Dim ResultsLabel As Label = CType(MainContent.FindControl("Results"), Label)
 Dim AgeTextBox As TextBox = CType(MainContent.FindControl("Age"), TextBox)

 ResultsLabel.Text = String.Format("You are {0} years old!", AgeTextBox.Text)
End Sub
```

This time, visiting the page through a browser, entering your age, and clicking the "Submit" button displays the message in the `Results` Label, as expected.

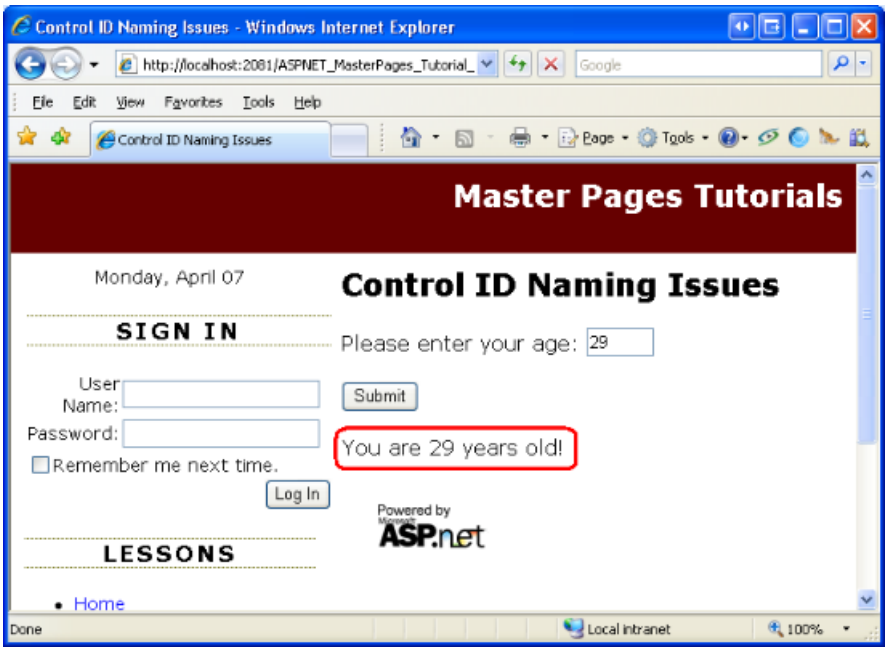


Figure 06: The User's Age is Displayed in the Label (Click to view full-size image)

Recursively Searching Through Naming Containers

The reason the previous code example referenced the `MainContent` ContentPlaceHolder control from the master page, and then the `Results` Label and `Age` TextBox controls from `MainContent`, is because the `Control.FindControl` method only searches within *Control*'s naming container. Having `FindControl` stay within

the naming container makes sense in most scenarios because two controls in two different naming containers may have the same `ID` values. Consider the case of a GridView that defines a Label Web control named `ProductName` within one of its TemplateFields. When the data is bound to the GridView at runtime, a `ProductName` Label is created for each GridView row. If `FindControl` searched through all naming containers and we called `Page.FindControl("ProductName")`, what Label instance should the `FindControl` return? The `ProductName` Label in the first GridView row? The one in the last row?

So having `Control.FindControl` search just *Control*'s naming container makes sense in most cases. But there are other cases, such as the one facing us, where we have a unique `ID` across all naming containers and want to avoid having to meticulously reference each naming container in the control hierarchy to access a control. Having a `FindControl` variant that recursively searches all naming containers makes sense, too. Unfortunately, the .NET Framework does not include such a method.

The good news is that we can create our own `FindControl` method that recursively searches all naming containers. In fact, using *extension methods* we can tack on a `FindControlRecursive` method to the `Control` class to accompany its existing `FindControl` method.

> **NOTE**
> Extension methods are a feature new to C# 3.0 and Visual Basic 9, which are the languages that ship with the .NET Framework version 3.5 and Visual Studio 2008. In short, extension methods allow for a developer to create a new method for an existing class type through a special syntax. For more information on this helpful feature, refer to my article, Extending Base Type Functionality with Extension Methods.

To create the extension method, add a new file to the `App_Code` folder named `PageExtensionMethods.vb`. Add an extension method named `FindControlRecursive` that takes as an input a `String` parameter named `controlID`. For extension methods to work properly, it is vital that the class be marked as a `Module` and that the extension methods be prefixed with the `<Extension()>` attribute. Moreover, all extension methods must accept as their first parameter an object of the type to which the extension method applies.

Add the following code to the `PageExtensionMethods.vb` file to define this `Module` and the `FindControlRecursive` extension method:

```
Imports System.Runtime.CompilerServices

Public Module PageExtensionMethods
 <Extension()> _
  Public Function FindControlRecursive(ByVal ctrl As Control, ByVal controlID As String) As Control
 If String.Compare(ctrl.ID, controlID, True) = 0 Then
 ' We found the control!
 Return ctrl
 Else
 ' Recurse through ctrl's Controls collections
 For Each child As Control In ctrl.Controls
 Dim lookFor As Control = FindControlRecursive(child, controlID)

 If lookFor IsNot Nothing Then
 Return lookFor  ' We found the control
 End If
 Next

 ' If we reach here, control was not found
 Return Nothing
 End If
 End Function
End Module
```

With this code in place, return to the `IDIssues.aspx` page's code-behind class and comment out the current

`FindControl` method calls. Replace them with calls to `Page.FindControlRecursive("controlID")`. What's neat about extension methods is that they appear directly within the IntelliSense drop-down lists. As Figure 7 shows, when you type `Page` and then hit period, the `FindControlRecursive` method is included in the IntelliSense drop-down along with the other `Control` class methods.

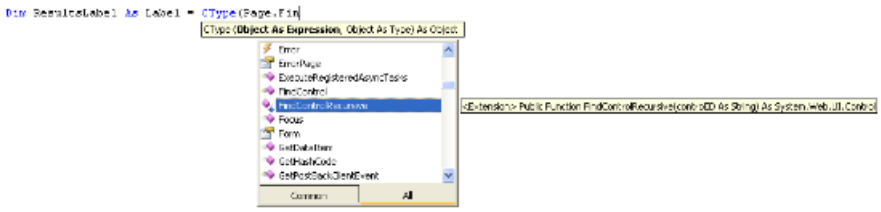


Figure 07: Extension Methods are Included in the IntelliSense Drop-Downs (Click to view full-size image)

Enter the following code into the `SubmitButton_Click` event handler and then test it by visiting the page, entering your age, and clicking the "Submit" button. As shown back in Figure 6, the resulting output will be the message, "You are age years old!"

```
Protected Sub SubmitButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
SubmitButton.Click
 Dim ResultsLabel As Label = CType(Page.FindControlRecursive("Results"), Label)
 Dim AgeTextBox As TextBox = CType(Page.FindControlRecursive("Age"), TextBox)

 ResultsLabel.Text = String.Format("You are {0} years old!", AgeTextBox.Text)
End Sub
```

> **NOTE**
>
> Because extension methods are new to C# 3.0 and Visual Basic 9, if you are using Visual Studio 2005 you cannot use extension methods. Instead, you'll need to implement the `FindControlRecursive` method in a helper class. Rick Strahl has such an example in his blog post, ASP.NET Maser Pages and `FindControl`.

Step 4: Using the Correct `id` Attribute Value in Client-Side Script

As noted in this tutorial's introduction, a Web control's rendered `id` attribute is oftentimes used in client-side script to programmatically reference a particular HTML element. For example, the following JavaScript references an HTML element by its `id` and then displays its value in a modal message box:

```
var elem = document.getElementById("Age");
if (elem != null)
    alert("You entered " + elem.value + " into the Age text box.");
```

Recall that in ASP.NET pages that do not include a naming container, the rendered HTML element's `id` attribute is identical to the Web control's `ID` property value. Because of this, it is tempting to hard code in `id` attribute values into JavaScript code. That is, if you know you want to access the `Age` TextBox Web control through client-side script, do so via a call to `document.getElementById("Age")`.

The problem with this approach is that when using master pages (or other naming container controls), the rendered HTML `id` is not synonymous with the Web control's `ID` property. Your first inclination may be to visit the page through a browser and view the source to determine the actual `id` attribute. Once you know the rendered `id` value, you can paste it into the call to `getElementById` to access the HTML element you need to work with through client-side script. This approach is less than ideal because certain changes to the page's control hierarchy or changes to the `ID` properties of the naming controls will alter the resulting `id` attribute, thereby breaking your JavaScript code.

The good news is that the `id` attribute value that is rendered is accessible in server-side code through the Web control's `ClientID` property. You should use this property to determine the `id` attribute value used in client-side script. For example, to add a JavaScript function to the page that, when called, displays the value of the `Age` TextBox in a modal message box, add the following code to the `Page_Load` event handler:

```
ClientScript.RegisterClientScriptBlock(Me.GetType(), "ShowAgeTextBoxScript", _
 "function ShowAge() " & vbCrLf & _
 "{" & vbCrLf & _
 " var elem = document.getElementById('" & AgeTextBox.ClientID & "');" & vbCrLf & _
 " if (elem != null)" & vbCrLf & _
 " alert('You entered ' + elem.value + ' into the Age text box.');" & vbCrLf & _
 "}", True)
```

The above code injects the value of the `Age` TextBox's `ClientID` property into the JavaScript call to `getElementById`. If you visit this page through a browser and view the HTML source, you'll find the following JavaScript code:

```
<script type="text/javascript">
//<![CDATA[
function ShowAge()
{
 var elem = document.getElementById('ctl00_MainContent_Age');
 if (elem != null)
 alert('You entered ' + elem.value + ' into the Age text box.');
}//]]>
</script>
```

Notice how the correct `id` attribute value, `ctl00_MainContent_Age`, appears within the call to `getElementById`. Because this value is calculated at runtime, it works regardless of later changes to the page control hierarchy.

> **NOTE**
> This JavaScript example merely shows how to add a JavaScript function that correctly references the HTML element rendered by a server control. To use this function you would need to author additional JavaScript to call the function when the document loads or when some specific user action transpires. For more information on these and related topics, read Working with Client-Side Script.

Summary

Certain ASP.NET server controls act as naming containers, which affects the rendered `id` attribute values of their descendent controls as well as the scope of controls canvassed by the `FindControl` method. With regards to master pages, both the master page itself and its ContentPlaceHolder controls are naming containers. Consequently, we need to put forth a bit more work to programmatically reference controls within the content page using `FindControl`. In this tutorial we examined two techniques: drilling into the ContentPlaceHolder control and calling its `FindControl` method; and rolling our own `FindControl` implementation that recursively searches through all naming containers.

In addition to the server-side issues naming containers introduce with regards to referencing Web controls, there are also client-side issues. In the absence of naming containers, the Web control's `ID` property value and rendered `id` attribute value are one in the same. But with the addition of naming container, the rendered `id` attribute includes both the `ID` values of the Web control and the naming container(s) in its control hierarchy's ancestry. These naming concerns are a non-issue as long as you use the Web control's `ClientID` property to determine the rendered `id` attribute value in your client-side script.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Master Pages and `FindControl`
- Creating Dynamic Data Entry User Interfaces
- Extending Base Type Functionality with Extension Methods
- How To: Reference ASP.NET Master Page Content
- Mater Pages: Tips, Tricks, and Traps
- Working with Client-Side Script

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Zack Jones and Suchi Barnerjee. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Interacting with the Master Page from the Content Page (VB)

2/20/2020 • 19 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> Examines how to call methods, set properties, etc. of the Master Page from code in the Content Page.

Introduction

Over the course of the past five tutorials we have looked at how to create a master page, define content regions, bind ASP.NET pages to a master page, and define page-specific content. When a visitor requests a particular content page, the content and master pages' markup are fused at runtime, resulting in the rendering of a unified control hierarchy. Therefore, we have already seen one way in which the master page and one of its content pages can interact: the content page spells out the markup to transfuse into the master page's ContentPlaceHolder controls.

What we have yet to examine is how the master page and content page can interact programmatically. In addition to defining the markup for the master page's ContentPlaceHolder controls, a content page can also assign values to its master page's public properties and invoke its public methods. Similarly, a master page may interact with its content pages. While programmatic interaction between a master and content page is less common than the interaction between their declarative markups, there are many scenarios where such programmatic interaction is needed.

In this tutorial we examine how a content page can programmatically interact with its master page; in the next tutorial we will look at how the master page can similarly interact with its content pages.

Examples of Programmatic Interaction Between a Content Page and its Master Page

When a particular region of a page needs to be configured on a page-by-page basis, we use a ContentPlaceHolder control. But what about situations where the majority of pages need to emit a certain output, but a small number of pages need to customize it to show something else? One such example, which we examined in the *Multiple ContentPlaceHolders and Default Content* tutorial, involves displaying a login interface on each page. While most pages should include a login interface, it should be suppressed for a handful of pages, such as: the main login page (`Login.aspx`); the Create Account page; and other pages that are only accessible to authenticated users. The *Multiple ContentPlaceHolders and Default Content* tutorial showed how to define the default content for a ContentPlaceHolder in the master page and then how to override it in those pages where the default content was not wanted.

Another option is to create a public property or method within the master page that indicates whether to show or hide the login interface. For example, the master page might include a public property named `ShowLoginUI` whose value was used to set the `Visible` property of the Login control in the master page. Those content pages where the login user interface should be suppressed could then programmatically set the `ShowLoginUI` property to `False`.

Perhaps the most common example of content and master page interaction occurs when data displayed in the master page needs to be refreshed after some action has transpired in the content page. Consider a master page

that includes a GridView that displays the five most recently added records from a particular database table, and that one of its content pages includes an interface for adding new records to that same table.

When a user visits the page to add a new record, she sees the five most recently added records displayed in the master page. After filling in the values for the new record's columns, she submits the form. Assuming that the GridView in the master page has its `EnableViewState` property set to True (the default), its content is reloaded from view state and, consequently, the five same records are displayed even though a newer record was just added to the database. This may confuse the user.

> **NOTE**
> Even if you disable the GridView's view state so that it rebinds to its underlying data source on every postback, it still won't show the just-added record because the data is bound to the GridView earlier in the page lifecycle than when the new record is added to the database.

To remedy this so that the just-added record is displayed in the master page's GridView on postback we need to instruct the GridView to rebind to its data source *after* the new record has been added to the database. This requires interaction between the content and master pages because the interface for adding the new record (and its event handlers) are in the content page but the GridView that needs to be refreshed is in the master page.

Because refreshing the master page's display from an event handler in the content page is one of the most common needs for content and master page interaction, let's explore this topic in more detail. The download for this tutorial includes a Microsoft SQL Server 2005 Express Edition database named `NORTHWIND.MDF` in the website's `App_Data` folder. The Northwind database stores product, employee, and sales information for a fictitious company, Northwind Traders.

Step 1 walks through displaying the five most recently added products in a GridView in the master page. Step 2 creates a content page for adding new products. Step 3 looks at how to create public properties and methods in the master page, and Step 4 illustrates how to programmatically interface with these properties and methods from the content page.

> **NOTE**
> This tutorial does not delve into the specifics of working with data in ASP.NET. The steps for setting up the master page to display data and the content page for inserting data are complete, yet breezy. For a more in-depth look at displaying and inserting data and using the SqlDataSource and GridView controls, consult the resources in the Further Readings section at the end of this tutorial.

Step 1: Displaying the Five Most Recently Added Products in the Master Page

Open the Site.master master page and add a Label and a GridView control to the `leftContent` `<div>`. Clear out the Label's `Text` property, set its `EnableViewState` property to `False`, and its `ID` property to `GridMessage`; set the GridView's `ID` property to `RecentProducts`. Next, from the Designer, expand the GridView's smart tag and choose to bind it to a new data source. This launches the Data Source Configuration wizard. Because the Northwind database in the `App_Data` folder is a Microsoft SQL Server database, choose to create a SqlDataSource by selecting (see Figure 1); name the SqlDataSource `RecentProductsDataSource`.

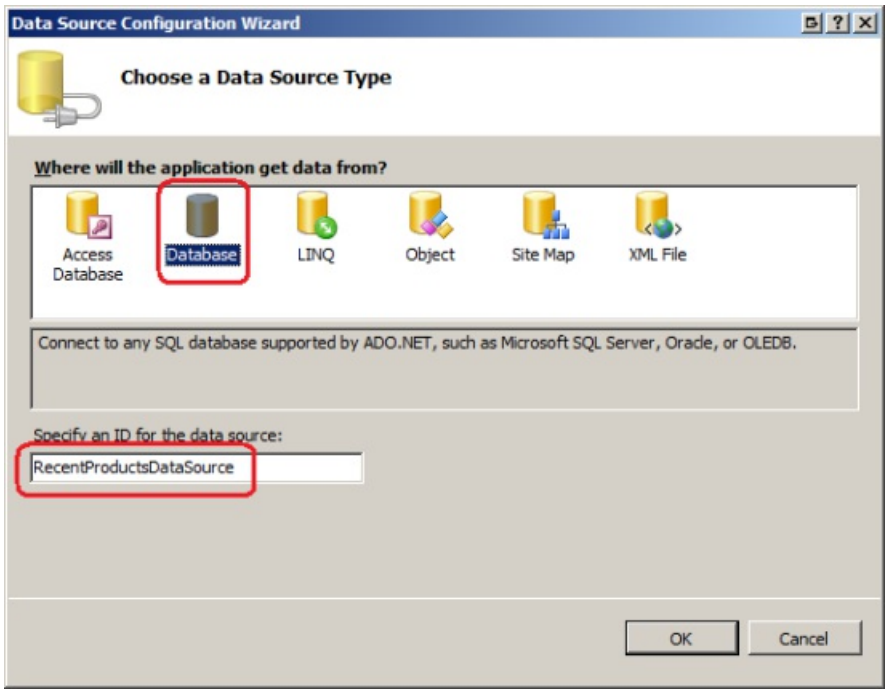


Figure 01: Bind the GridView to a SqlDataSource Control Named `RecentProductsDataSource` (Click to view full-size image)

The next step asks us to specify what database to connect to. Choose the `NORTHWIND.MDF` database file from the drop-down list and click Next. Because this is the first time we've used this database, the wizard will offer to store the connection string in `Web.config`. Have it store the connection string using the name `NorthwindConnectionString`.

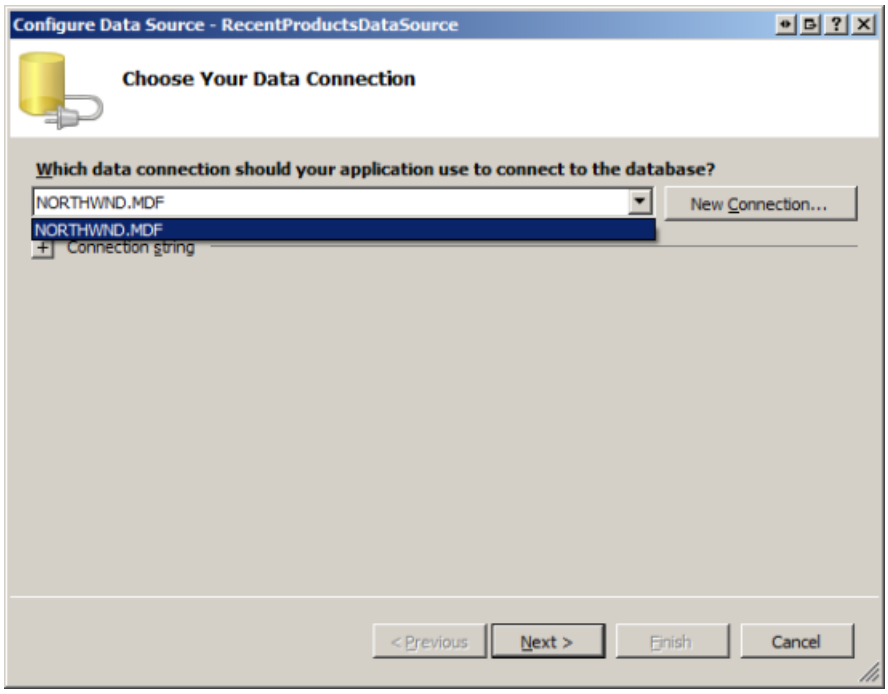


Figure 02: Connect to the Northwind Database (Click to view full-size image)

The Configure Data Source wizard provides two means by which we can specify the query used to retrieve data:

- By specifying a custom SQL statement or stored procedure, or
- By picking a table or view and then specifying the columns to return

Because we want to return just the five most recently added products, we need to specify a custom SQL statement. Use the following `SELECT` query:

```
SELECT TOP 5 ProductName, UnitPrice FROM Products ORDER BY ProductID DESC
```

The `TOP 5` keyword returns only the first five records from the query. The `Products` table's primary key, `ProductID`, is an `IDENTITY` column, which assures us that each new product added to the table will have a larger value than the previous entry. Therefore, sorting the results by `ProductID` in descending order returns the products starting with the most recently created ones.

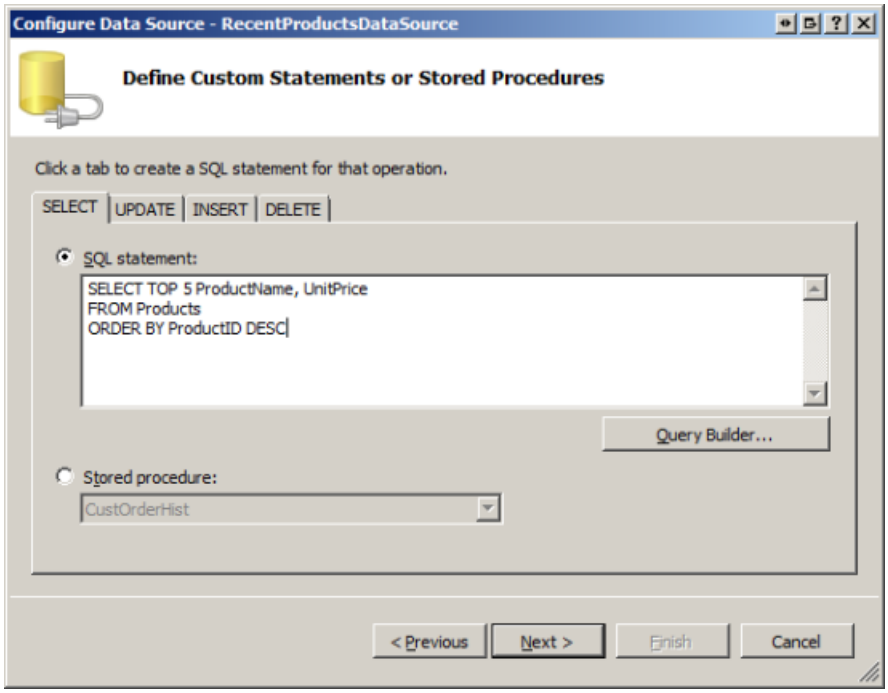


Figure 03: Return the Five Most Recently Added Products (Click to view full-size image)

After completing the wizard, Visual Studio generates two BoundFields for the GridView to display the `ProductName` and `UnitPrice` fields returned from the database. At this point your master page's declarative markup should include markup similar to the following:

```
<asp:Label ID="GridMessage" runat="server" EnableViewState="false"></asp:Label>
<asp:GridView ID="RecentProducts" runat="server" AutoGenerateColumns="False"
 DataSourceID="RecentProductsDataSource">
 <Columns>
 <asp:BoundField DataField="ProductName" HeaderText="ProductName"
 SortExpression="ProductName" />
 <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
 SortExpression="UnitPrice" />
 </Columns>
</asp:GridView>
<asp:SqlDataSource ID="RecentProductsDataSource" runat="server"
 ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
 SelectCommand="SELECT TOP 5 ProductName, UnitPrice FROM Products ORDER BY ProductID DESC">
</asp:SqlDataSource>
```

As you can see, the markup contains: the Label Web control (`GridMessage`); the GridView `RecentProducts`, with two BoundFields; and a SqlDataSource control that returns the five most recently added products.

With this GridView created and its SqlDataSource control configured, visit the website through a browser. As Figure 4 shows, you will see a grid in the lower left corner that lists the five most recently added products.

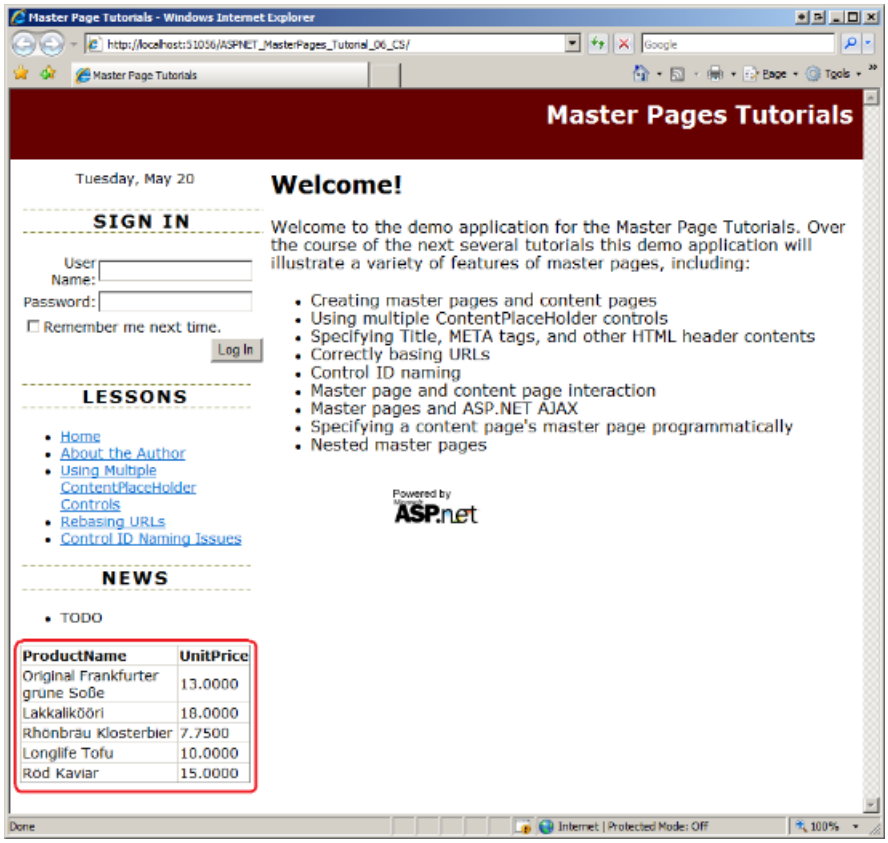


Figure 04: The GridView Displays the Five Most Recently Added Products (Click to view full-size image)

> **NOTE**
>
> Feel free to clean up the appearance of the GridView. Some suggestions include formatting the displayed `UnitPrice` value as a currency and using background colors and fonts to improve the grid's appearance.

Step 2: Creating a Content Page to Add New Products

Our next task is to create a content page from which a user can add a new product to the `Products` table. Add a new content page to the `Admin` folder named `AddProduct.aspx`, making sure to bind it to the `Site.master` master page. Figure 5 shows the Solution Explorer after this page has been added to the website.

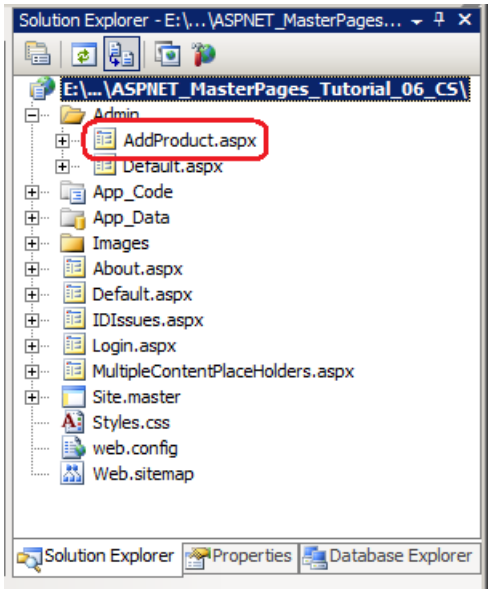


Figure 05: Add a New ASP.NET Page to the `Admin` Folder (Click to view full-size image)

Recall that in the *Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page* tutorial we created a custom base page class named `BasePage` that generated the page's title if it was not explicitly set. Go to the `AddProduct.aspx` page's code-behind class and have it derive from `BasePage` (instead of from `System.Web.UI.Page`).

Finally, update the `Web.sitemap` file to include an entry for this lesson. Add the following markup beneath the `<siteMapNode>` for the Control ID Naming Issues lesson:

```
<siteMapNode url="~/Admin/AddProduct.aspx" title="Content to Master Page Interaction" />
```

As shown in Figure 6, the addition of this `<siteMapNode>` element is reflected in the Lessons list.

Return to `AddProduct.aspx`. In the Content control for the `MainContent` ContentPlaceHolder, add a DetailsView control and name it `NewProduct`. Bind the DetailsView to a new SqlDataSource control named `NewProductDataSource`. Like with the SqlDataSource in Step 1, configure the wizard so that it uses the Northwind database and choose to specify a custom SQL statement. Because the DetailsView will be used to add items to the database, we need to specify both a `SELECT` statement and an `INSERT` statement. Use the following `SELECT` query:

```
SELECT ProductName, UnitPrice FROM Products
```

Then, from the INSERT tab, add the following `INSERT` statement:

```
INSERT INTO Products(ProductName, UnitPrice) VALUES(@ProductName, @UnitPrice)
```

After completing the wizard go to the DetailsView's smart tag and check the "Enable Inserting" checkbox. This adds a CommandField to the DetailsView with its `ShowInsertButton` property set to True. Because this DetailsView will be used solely for inserting data, set the DetailsView's `DefaultMode` property to `Insert`.

That's all there is to it! Let's test this page. Visit `AddProduct.aspx` through a browser, enter a name and price (see Figure 6).

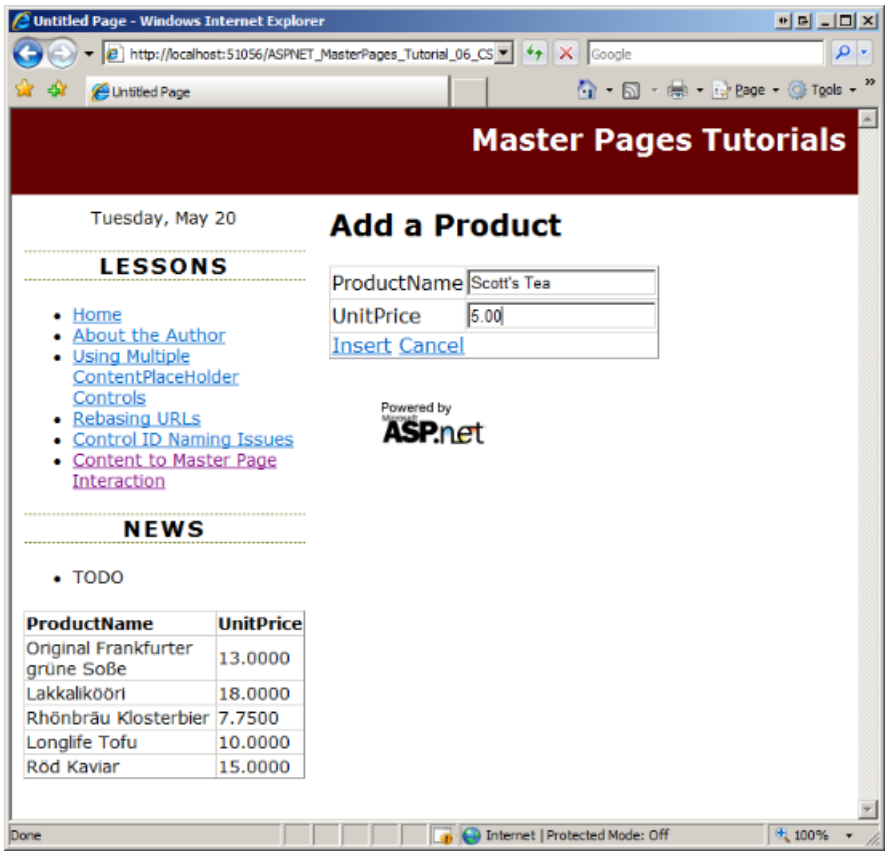


Figure 06: Add a New Product to the Database (Click to view full-size image)

After typing in the name and price for your new product, click the Insert button. This causes the form to postback. On postback, the SqlDataSource control's `INSERT` statement is executed; its two parameters are populated with the user-entered values in the DetailsView's two TextBox controls. Unfortunately, there is no visual feedback that an insert has occurred. It would be nice to have a message displayed, confirming that a new record has been added. I leave this as an exercise for the reader. Also, after adding a new record from the DetailsView the GridView in the master page still shows the same five records as before; it does not include the just-added record. We'll examine how to remedy this in the upcoming steps.

> **NOTE**
>
> In addition to adding some form of visual feedback that the insert has succeeded, I'd encourage you to also update the DetailsView's inserting interface to include validation. Currently, there is no validation. If a user enters an invalid value for the `UnitPrice` field, such as "Too expensive," an exception will be thrown on postback when the system attempts to convert that string into a decimal. For more information on customizing the inserting interface, refer to the *Customizing the Data Modification Interface* tutorial from my Working with Data tutorial series.

Step 3: Creating Public Properties and Methods in the Master Page

In Step 1 we added a Label Web control named `GridMessage` above the GridView in the master page. This Label is intended to optionally display a message. For example, after adding a new record to the `Products` table, we might want to show a message that reads: "*ProductName* has been added to the database." Rather than hard-code the text for this Label in the master page, we might want the message to be customizable by the content page.

Because the Label control is implemented as a protected member variable within the master page it cannot be accessed directly from content pages. In order to work with the Label within a master page from the content page (or, for that matter, any Web control in the master page) we need to create a public property in the master page that exposes the Web control or serves as a proxy by which one of its properties can be accessed. Add the following syntax to the master page's code-behind class to expose the Label's `Text` property:

```
Public Property GridMessageText() As String
 Get
 Return GridMessage.Text
 End Get
 Set(ByVal Value As String)
 GridMessage.Text = Value
 End Set
End Property
```

When a new record is added to the `Products` table from a content page the `RecentProducts` GridView in the master page needs to rebind to its underlying data source. To rebind the GridView call its `DataBind` method. Because the GridView in the master page is not programmatically accessible to the content pages, we need to create a public method in the master page that, when called, rebinds the data to the GridView. Add the following method to the master page's code-behind class:

```
Public Sub RefreshRecentProductsGrid()
 RecentProducts.DataBind()
End Sub
```

With the `GridMessageText` property and `RefreshRecentProductsGrid` method in place, any content page can programmatically set or read the value of the `GridMessage` Label's `Text` property or rebind the data to the `RecentProducts` GridView. Step 4 examines how to access the master page's public properties and methods from a content page.

> **NOTE**
> Don't forget to mark the master page's properties and methods as `Public`. If you do not explicitly denote these properties and methods as `Public`, they will not be accessible from the content page.

Step 4: Calling the Master Page's Public Members from a Content Page

Now that the master page has the necessary public properties and methods, we're ready to invoke these properties and methods from the `AddProduct.aspx` content page. Specifically, we need to set the master page's `GridMessageText` property and call its `RefreshRecentProductsGrid` method after the new product has been added to the database. All the ASP.NET data Web controls fire events immediately before and after completing various tasks, which make it easy for page developers to take some programmatic action either before or after the task. For example, when the end user clicks the DetailsView's Insert button, on postback the DetailsView raises its `ItemInserting` event before beginning the inserting workflow. It then inserts the record into the database. Following that, the DetailsView raises its `ItemInserted` event. Therefore, in order to work with the master page after the new product has been added, create an event handler for the DetailsView's `ItemInserted` event.

There are two ways that a content page can programmatically interface with its master page:

- Using the `Page.Master` property, which returns a loosely-typed reference to the master page, or
- Specify the page's master page type or file path via a `@MasterType` directive; this automatically adds a strongly-typed property to the page named `Master`.

Let's examine both approaches.

Using the Loosely-Typed `Page.Master` Property

All ASP.NET web pages must derive from the `Page` class, which is located in the `System.Web.UI` namespace. The `Page` class includes a `Master` property that returns a reference to the page's master page. If the page does not have a master page `Master` returns `Nothing`.

The `Master` property returns an object of type `MasterPage` (also located in the `System.Web.UI` namespace) which is the base type from which all master pages derive from. Therefore, to use public properties or methods defined in our website's master page we must cast the `MasterPage` object returned from the `Master` property to the appropriate type. Because we named our master page file `Site.master`, the code-behind class was named `Site`. Therefore, the following code casts the `Page.Master` property to an instance of the `Site` class.

```
' Cast the loosely-typed Page.Master property and then set the GridMessageText property
Dim myMasterPage As Site = CType(Page.Master, Site)
```

Now that we have casted the loosely-typed `Page.Master` property to the Site type we can reference the properties and methods specific to Site. As Figure 7 shows, the public property `GridMessageText` appears in the IntelliSense drop-down.

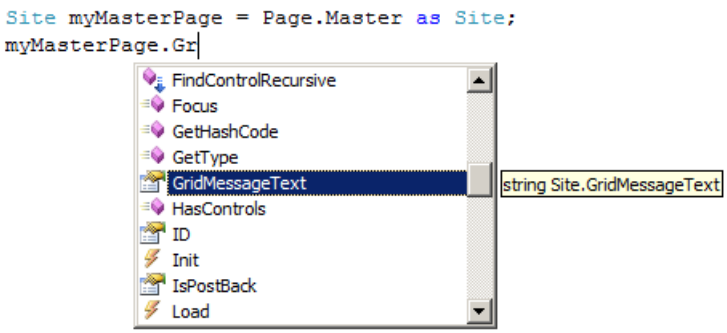


Figure 07: IntelliSense Shows our Master Page's Public Properties and Methods (Click to view full-size image)

> **NOTE**
> If you named your master page file `MasterPage.master` then the master page's code-behind class name is `MasterPage`. This can lead to ambiguous code when casting from the type `System.Web.UI.MasterPage` to your `MasterPage` class. In short, you need to fully qualify the type you are casting to, which can be a little tricky when using the Web Site Project model. My suggestion would be to either make sure that when you create your master page you name it something other than `MasterPage.master` or, even better, create a strongly-typed reference to the master page.

Creating a Strongly-Typed Reference with the `@MasterType` Directive

If you look closely you can see that an ASP.NET page's code-behind class is a partial class (note the `Partial` keyword in the class definition). Partial classes were introduced in C# and Visual Basic with.NET Framework 2.0 and, in a nutshell, allow for a class's members to be defined across multiple files. The code-behind class file - `AddProduct.aspx.vb`, for example - contains the code that we, the page developer, create. In addition to our code, the ASP.NET engine automatically creates a separate class file with properties and event handlers in that translate the declarative markup into the page's class hierarchy.

The automatic code generation that occurs whenever an ASP.NET page is visited paves the way for some rather interesting and useful possibilities. In the case of master pages, if we tell the ASP.NET engine what master page is being used by our content page it generates a strongly-typed `Master` property for us.

Use the `@MasterType` directive to inform the ASP.NET engine of the content page's master page type. The `@MasterType` directive can accept either the type name of the master page or its file path. To specify that the `AddProduct.aspx` page uses `Site.master` as its master page, add the following directive to the top of `AddProduct.aspx`:

```
<%@ MasterType VirtualPath="~/Site.master" %>
```

This directive instructs the ASP.NET engine to add a strongly-typed reference to the master page through a property named `Master`. With the `@MasterType` directive in place, we can call the `Site.master` master page's public properties and methods directly through the `Master` property without any casts.

> **NOTE**
> If you omit the `@MasterType` directive, the syntax `Page.Master` and `Master` return the same thing: a loosely-typed object to the page's master page. If you include the `@MasterType` directive then `Master` returns a strongly-typed reference to the specified master page. `Page.Master`, however, still returns a loosely-typed reference. For a more thorough look at why this is the case and how the `Master` property is constructed when the `@MasterType` directive is included, see K. Scott Allen's blog entry `@MasterType` in ASP.NET 2.0.

Updating the Master Page After Adding a New Product

Now that we know how to invoke a master page's public properties and methods from a content page, we're ready to update the `AddProduct.aspx` page so that the master page is refreshed after adding a new product. At the beginning of Step 4 we created an event handler for the DetailsView control's `ItemInserting` event, which executes immediately after the new product has been added to the database. Add the following code to that event handler:

```
Protected Sub NewProduct_ItemInserted(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.DetailsViewInsertedEventArgs) Handles NewProduct.ItemInserted
 ' Cast the loosely-typed Page.Master property and then set the GridMessageText property
 Dim myMasterPage As Site = CType(Page.Master, Site)
 myMasterPage.GridMessageText = String.Format("{0} added to grid...", e.Values("ProductName"))
 ' Use the strongly-typed Master property
 Master.RefreshRecentProductsGrid()
End Sub
```

The above code uses both the loosely-typed `Page.Master` property and the strongly-typed `Master` property. Note that the `GridMessageText` property is set to "*ProductName* added to grid..." The just-added product's values are accessible through the `e.Values` collection; as you can see, the just-added `ProductName` value is accessed via `e.Values("ProductName")`.

Figure 8 shows the `AddProduct.aspx` page immediately after a new product - Scott's Soda - has been added to the database. Note that the just-added product name is noted in the master page's Label and that the GridView has been refreshed to include the product and its price.

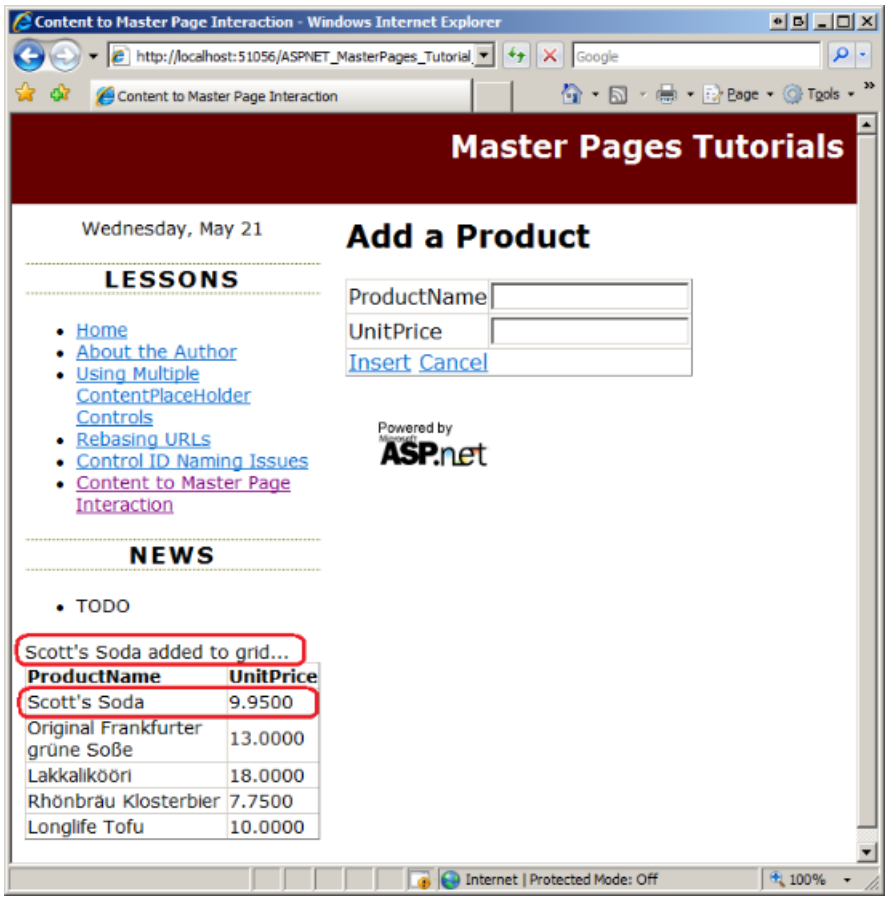


Figure 08: The Master Page's Label and GridView Show the Just-Added Product (Click to view full-size image)

Summary

Ideally, a master page and its content pages are completely separate from one another and require no level of interaction. While master pages and content pages should be designed with that goal in mind, there are a number of common scenarios in which a content page must interface with its master page. One of the most common reasons centers around updating a particular portion of the master page display based on some action that transpired in the content page.

The good news is that it's relatively straightforward to have a content page programmatically interact with its master page. Start by creating public properties or methods in the master page that encapsulate the functionality that needs to be invoked by a content page. Then, in the content page, access the master page's properties and methods through the loosely-typed `Page.Master` property or use the `@MasterType` directive to create a strongly-typed reference to the master page.

In the next tutorial we examine how to have the master page programmatically interact with one of its content pages.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Accessing and Updating Data in ASP.NET
- ASP.NET Master Pages: Tips, Tricks, and Traps
- `@MasterType` in ASP.NET 2.0
- Passing Information Between Content and Master Pages
- Working with Data in ASP.NET Tutorials

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Zack Jones. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Interacting with the Content Page from the Master Page (VB)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> Examines how to call methods, set properties, etc. of the Content Page from code in the Master Page.

Introduction

The preceding tutorial examined how to have the content page programmatically interact with its master page. Recall that we updated the master page to include a GridView control that listed the five most recently added products. We then created a content page from which the user could add a new product. Upon adding a new product, the content page needed to instruct the master page to refresh its GridView so that it would include the just-added product. This functionality was accomplished by adding a public method to the master page that refreshed the data bound to the GridView, and then invoking that method from the content page.

The most common form of content and master page interaction originates from the content page. However, it is possible for the master page to rouse the current content page into action, and such functionality may be needed if the master page contains user interface elements that enable users to modify data that is also displayed on the content page. Consider a content page that displays the products information in a GridView control and a master page that includes a Button control that, when clicked, doubles the prices of all products. Much like the example in the preceding tutorial, the GridView needs to be refreshed after the double price Button is clicked so that it displays the new prices, but in this scenario it's the master page that needs to rouse the content page into action.

This tutorial explores how to have the master page invoke functionality defined in the content page.

Instigating Programmatic Interaction via an Event and Event Handlers

Invoking content page functionality from a master page is more challenging than the other way around. Because a content page has a single master page, when instigating the programmatic interaction from the content page we know what public methods and properties are at our disposal. A master page, however, can have many different content pages, each with its own set of properties and methods. How, then, can we write code in the master page to perform some action in its content page when we don't know what content page will be invoked until runtime?

Consider an ASP.NET Web control, such as the Button control. A Button control can appear on any number of ASP.NET pages and needs a mechanism by which it can alert the page that it has been clicked. This is accomplished using *events*. In particular, the Button control raises its `Click` event when it is clicked; the ASP.NET page that contains the Button can optionally respond to that notification via an *event handler*.

This same pattern can be used to have a master page trigger functionality in its content pages:

1. Add an event to the master page.
2. Raise the event whenever the master page needs to communicate with its content page. For example, if the master page needs to alert its content page that the user has doubled the prices, its event would be raised immediately after the prices have been doubled.
3. Create an event handler in those content pages that need to take some action.

This remainder of this tutorial implements the example outlined in the Introduction; namely, a content page that lists the products in the database and a master page that includes a Button control to double the prices.

Step 1: Displaying Products in a Content Page

Our first order of business is to create a content page that lists the products from the Northwind database. (We added the Northwind database to the project in the preceding tutorial, *Interacting with the Master Page from the Content Page*.) Start by adding a new ASP.NET page to the `~/Admin` folder named `Products.aspx`, making sure to bind it to the `Site.master` master page. Figure 1 shows the Solution Explorer after this page has been added to the website.

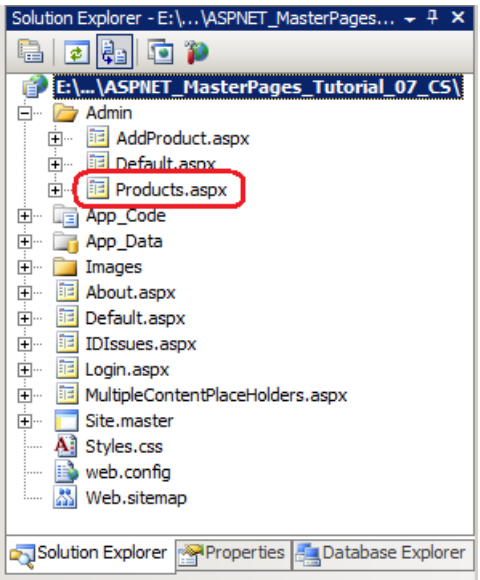


Figure 01: Add a New ASP.NET Page to the `Admin` Folder (Click to view full-size image)

Recall that in the *Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page* tutorial we created a custom base page class named `BasePage` that generates the page's title if it is not explicitly set. Go to the `Products.aspx` page's code-behind class and have it derive from `BasePage` (instead of from `System.Web.UI.Page`).

Finally, update the `Web.sitemap` file to include an entry for this lesson. Add the following markup beneath the `<siteMapNode>` for the Content to Master Page Interaction lesson:

```
<siteMapNode url="~/Admin/Products.aspx" title="Master to Content Page Interaction" />
```

The addition of this `<siteMapNode>` element is reflected in the Lessons list (see Figure 5).

Return to `Products.aspx`. In the Content control for `MainContent`, add a GridView control and name it `ProductsGrid`. Bind the GridView to a new SqlDataSource control named `ProductsDataSource`.

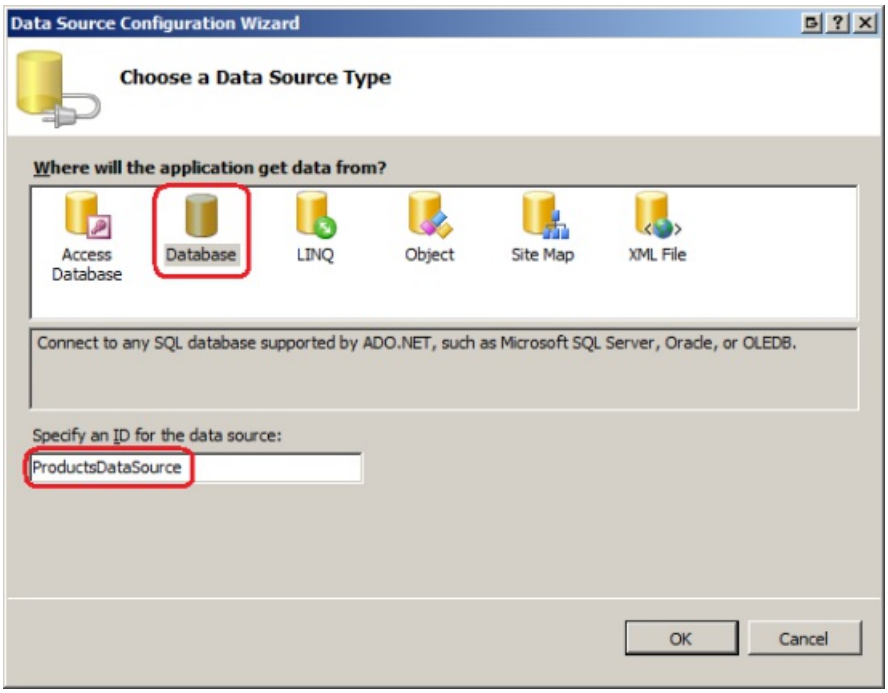


Figure 02: Bind the GridView to a New SqlDataSource Control (Click to view full-size image)

Configure the wizard so that it uses the Northwind database. If you worked through the previous tutorial then you should already have a connection string named `NorthwindConnectionString` in `Web.config`. Choose this connection string from the drop-down list, as shown in Figure 3.

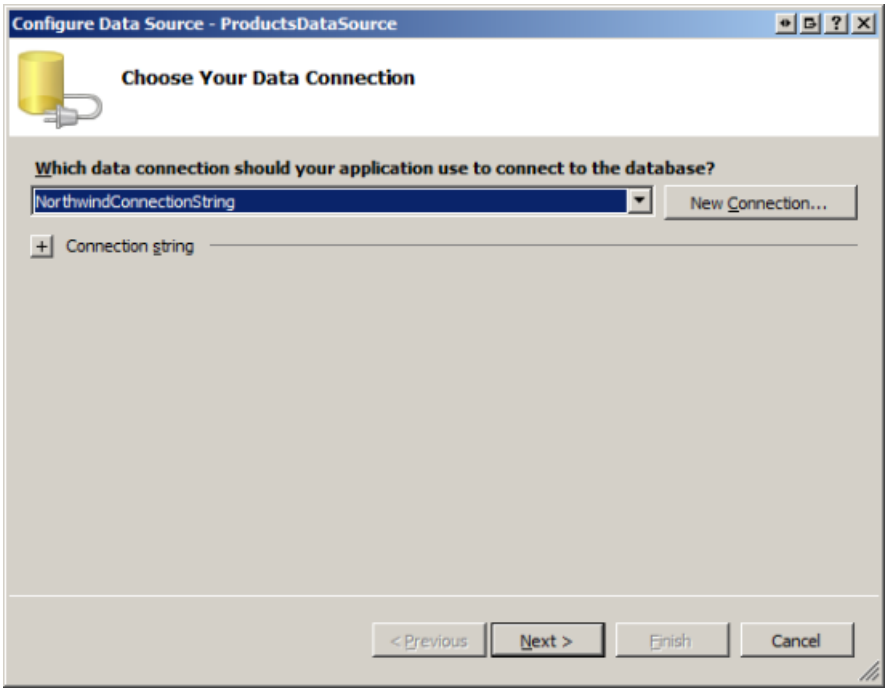


Figure 03: Configure the SqlDataSource to Use the Northwind Database (Click to view full-size image)

Next, specify the data source control's `SELECT` statement by choosing the Products table from the drop-down list and returning the `ProductName` and `UnitPrice` columns (see Figure 4). Click Next and then Finish to complete the Configure Data Source wizard.

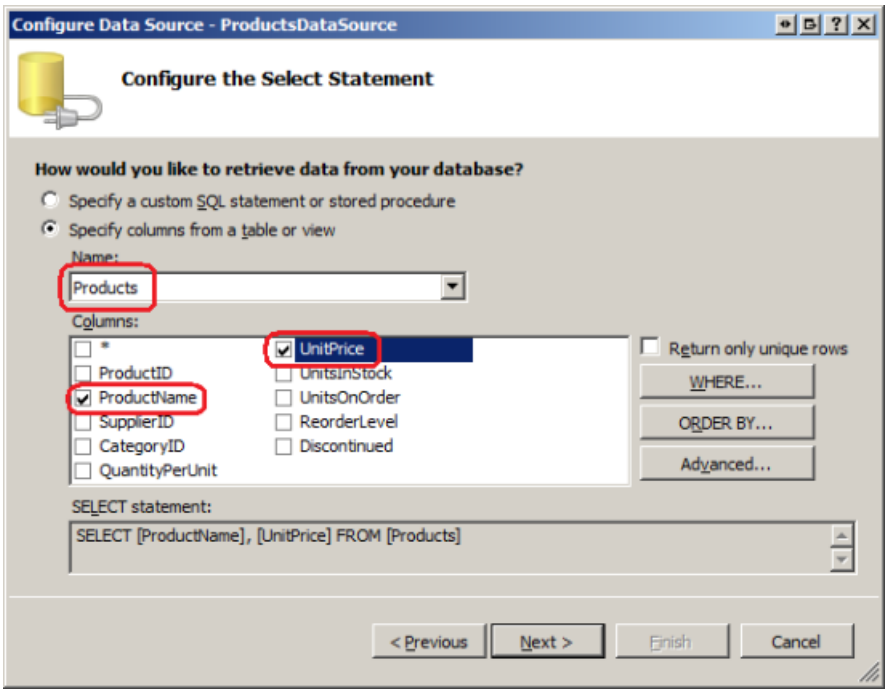


Figure 04: Return the `ProductName` and `UnitPrice` Fields from the `Products` Table (Click to view full-size image)

That's all there is to it! After completing the wizard Visual Studio adds two BoundFields to the GridView to mirror the two fields returned by the SqlDataSource control. The GridView and SqlDataSource controls' markup follows. Figure 5 shows the results when viewed through a browser.

```
<asp:GridView ID="ProductsGrid" runat="server" AutoGenerateColumns="False"
 DataSourceID="ProductsDataSource">
 <Columns>
 <asp:BoundField DataField="ProductName" HeaderText="ProductName"
 SortExpression="ProductName" />
 <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
 SortExpression="UnitPrice" />
 </Columns>
</asp:GridView>

<asp:SqlDataSource ID="ProductsDataSource" runat="server"
 ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
 SelectCommand="SELECT [ProductName], [UnitPrice] FROM [Products]">
</asp:SqlDataSource>
```

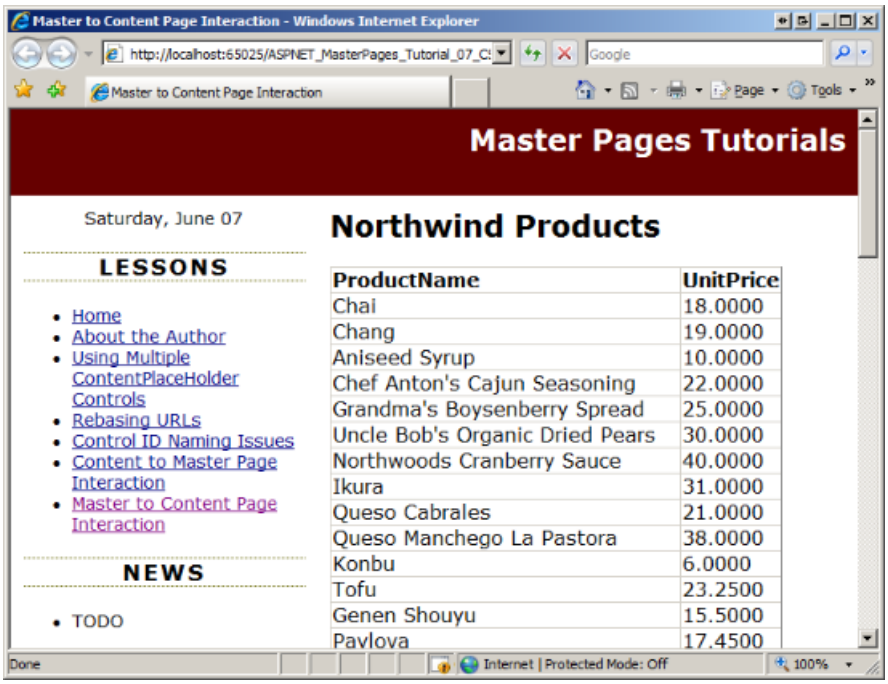


Figure 05: Each Product and its Price is Listed in the GridView (Click to view full-size image)

> **NOTE**
>
> Feel free to clean up the appearance of the GridView. Some suggestions include formatting the displayed UnitPrice value as a currency and using background colors and fonts to improve the grid's appearance. For more information on displaying and formatting data in ASP.NET, refer to my Working with Data tutorial series.

Step 2: Adding a Double Prices Button to the Master Page

Our next task is to add a Button Web control to the master page that, when clicked, will double the price of all products in the database. Open the `Site.master` master page and drag a Button from the Toolbox onto the Designer, placing it beneath the `RecentProductsDataSource` SqlDataSource control we added in the previous tutorial. Set the Button's `ID` property to `DoublePrice` and its `Text` property to "Double Product Prices".

Next, add a SqlDataSource control to the master page, naming it `DoublePricesDataSource`. This SqlDataSource will be used to execute the `UPDATE` statement to double all prices. Specifically, we need to set its `ConnectionString` and `UpdateCommand` properties to the appropriate connection string and `UPDATE` statement. Then we need to call this SqlDataSource control's `Update` method when the `DoublePrice` Button is clicked. To set the `ConnectionString` and `UpdateCommand` properties, select the SqlDataSource control and then go to the Properties window. The `ConnectionString` property lists those connection strings already stored in `Web.config` in a drop-down list; choose the `NorthwindConnectionString` option as shown in Figure 6.

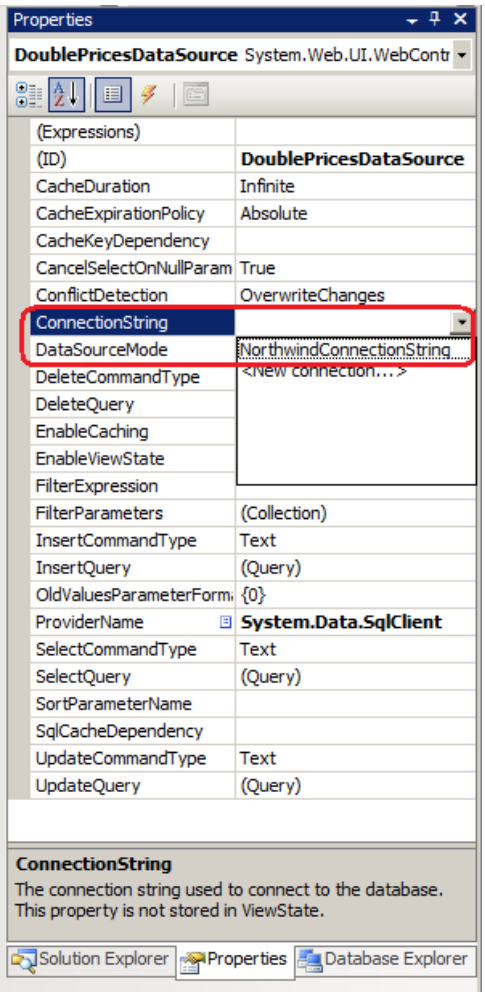


Figure 06: Configure the SqlDataSource to Use the `NorthwindConnectionString` (Click to view full-size image)

To set the `UpdateCommand` property, locate the UpdateQuery option in the Properties window. This property, when selected, displays a button with ellipses; click this button to display the Command and Parameter Editor dialog box shown in Figure 7. Type the following `UPDATE` statement into the dialog box's textbox:

```
UPDATE Products SET UnitPrice = UnitPrice * 2
```

This statement, when executed, will double the `UnitPrice` value for each record in the `Products` table.

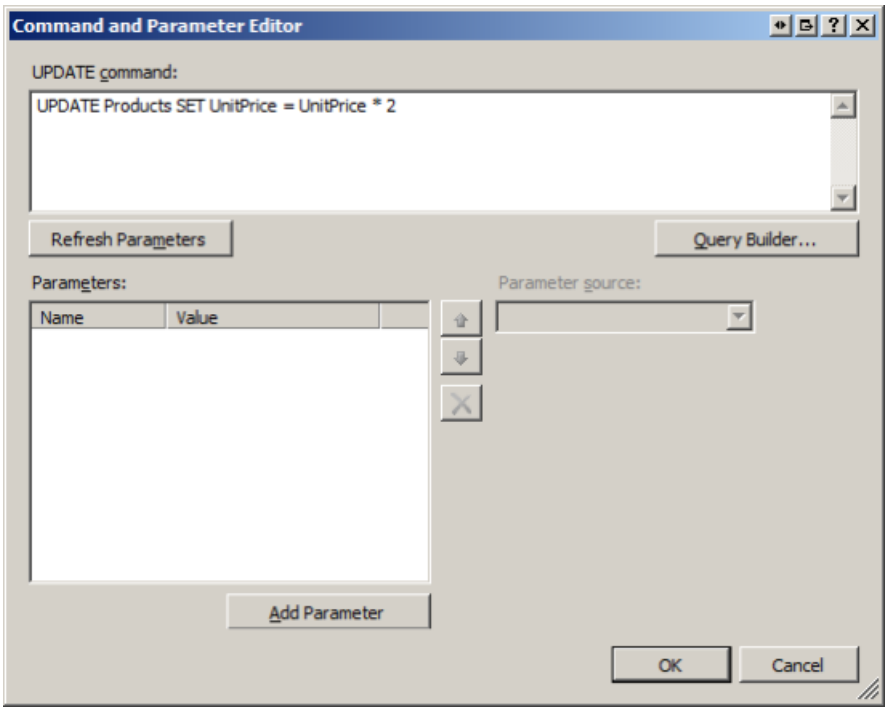


Figure 07: Set SqlDataSource's `UpdateCommand` Property (Click to view full-size image)

After setting these properties, your Button and SqlDataSource controls' declarative markup should look similar to the following:

```
<asp:Button ID="DoublePrice" runat="server"
 Text="Double Product Prices" />

<asp:SqlDataSource ID="DoublePricesDataSource" runat="server"
 UpdateCommand="UPDATE Products SET UnitPrice = UnitPrice * 2"
 ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
 ProviderName="<%$ ConnectionStrings:NorthwindConnectionString.ProviderName %>">
</asp:SqlDataSource>
```

All that remains is to call its `Update` method when the `DoublePrice` Button is clicked. Create a `Click` event handler for the `DoublePrice` Button and add the following code:

```
Protected Sub DoublePrice_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
DoublePrice.Click
 ' Double the prices
 DoublePricesDataSource.Update()
End Sub
```

To test this functionality, visit the `~/Admin/Products.aspx` page we created in Step 1 and click the "Double Product Prices" button. Clicking the button causes a postback and executes the `DoublePrice` Button's `Click` event handler, doubling the prices of all products. The page is then re-rendered and the markup is returned and re-displayed in the browser. The GridView in the content page, however, lists the same prices as before the "Double Product Prices" button was clicked. This is because the data initially loaded in the GridView had its state stored in view state, so it's not reloaded on postbacks unless instructed otherwise. If you visit a different page and then return to the `~/Admin/Products.aspx` page you'll see the updated prices.

Step 3: Raising an Event When the Prices are Doubled

Because the GridView in the `~/Admin/Products.aspx` page does not immediately reflect the price doubling, a user may understandably think that they did not click the "Double Product Prices" button, or that it didn't work. They may try clicking the button a few more times, doubling the prices again and again. To fix this we need to have the

grid in the content page display the new prices immediately after they are doubled.

As discussed earlier in this tutorial, we need to raise an event in the master page whenever the user clicks the `DoublePrice` Button. An event is a way for one class (an event publisher) to notify another a set of other classes (the event subscribers) that something interesting has occurred. In this example, the master page is the event publisher; those content pages that care about when the `DoublePrice` Button is clicked are the subscribers.

A class subscribes to an event by creating an *event handler*, which is a method that is executed in response to the event being raised. The publisher defines the events he raises by defining an *event delegate*. The event delegate specifies what input parameters the event handler must accept. In the .NET Framework, event delegates do not return any value and accept two input parameters:

- An `Object`, which identifies the event source, and
- A class derived from `System.EventArgs`

The second parameter passed to an event handler can include additional information about the event. While the base `EventArgs` class does not pass along any information, the .NET Framework includes a number of classes that extend `EventArgs` and encompass additional properties. For example, a `CommandEventArgs` instance is passed to event handlers that respond to the `Command` event, and includes two informational properties: `CommandArgument` and `CommandName`.

> **NOTE**
> For more information on creating, raising, and handling events, see Events and Delegates and Event Delegates in Simple English.

To define an event use the following syntax:

```
Public Event eventName As eventDelegate
```

Because we only need to alert the content page when the user has clicked the `DoublePrice` Button and do not need to pass along any other additional information, we can use the event delegate `EventHandler`, which defines an event handler that accepts as its second parameter an object of type `System.EventArgs`. To create the event in the master page, add the following line of code to the master page's code-behind class:

```
Partial Class Site
 Inherits System.Web.UI.MasterPage

 Public Event PricesDoubled As EventHandler
 ...
End Class
```

The above code adds a public event to the master page named `PricesDoubled`. We now need to raise this event after the prices have been doubled. To raise an event use the following syntax:

```
RaiseEvent eventName(sender, eventArgs)
```

Where *sender* and *eventArgs* are the values you want to pass to the subscriber's event handler.

Update the `DoublePrice` `Click` event handler with the following code:

```
Protected Sub DoublePrice_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
DoublePrice.Click
 ' Double the prices
 DoublePricesDataSource.Update()

 ' Refresh RecentProducts
 RecentProducts.DataBind()

 ' Raise the PricesDoubled event
 RaiseEvent PricesDoubled(Me, EventArgs.Empty)
End Sub
```

As before, the `Click` event handler starts by calling the `DoublePricesDataSource` SqlDataSource control's `Update` method to double the prices of all products. Following that there are two additions to the event handler. First, the `RecentProducts` GridView's data is refreshed. This GridView was added to the master page in the preceding tutorial and displays the five most recently-added products. We need to refresh this grid so that it shows the just-doubled prices for these five products. Following that, the `PricesDoubled` event is raised. A reference to the master page itself (`Me`) is sent to the event handler as the event source and an empty `EventArgs` object is sent as the event arguments.

Step 4: Handling the Event in the Content Page

At this point the master page raises its `PricesDoubled` event whenever the `DoublePrice` Button control is clicked. However, this is only half the battle - we still need to handle the event in the subscriber. This involves two steps: creating the event handler and adding event wiring code so that when the event is raised the event handler is executed.

Start by creating an event handler named `Master_PricesDoubled`. Because of how we defined the `PricesDoubled` event in the master page the event handler's two input parameters must be of types `Object` and `EventArgs`, respectively. In the event handler call the `ProductsGrid` GridView's `DataBind` method to rebind the data to the grid.

```
Private Sub Master_PricesDoubled(ByVal sender As Object, ByVal e As EventArgs)
 ' Rebind data to ProductsGrid
 ProductsGrid.DataBind()
End Sub
```

The code for the event handler is complete but we've yet to wire the master page's `PricesDoubled` event to this event handler. The subscriber wires an event to an event handler via the following syntax:

```
AddHandler publisher.eventName, AddressOf methodName
```

publisher is a reference to the object that offers the event *eventName*, and *methodName* is the name of the event handler defined in the subscriber.

This event wiring code must be executed on the first page visit and subsequent postbacks and should occur at a point in the page lifecycle that precedes when the event may be raised. A good time to add event wiring code is in the PreInit stage, which occurs very early in the page lifecycle.

Open `~/Admin/Products.aspx` and create a `Page_PreInit` event handler:

```
Protected Sub Page_PreInit(ByVal sender As Object, ByVal e As EventArgs) Handles Me.PreInit
 ' TODO: Put event wiring logic here
End Sub
```

In order to complete this wiring code we need a programmatic reference to the master page from the content page. As noted in the previous tutorial, there are two ways to do this:

- By casting the loosely-typed `Page.Master` property to the appropriate master page type, or
- By adding a `@MasterType` directive in the `.aspx` page and then using the strongly-typed `Master` property.

Let's use the latter approach. Add the following `@MasterType` directive to the top of the page's declarative markup:

```
<%@ MasterType VirtualPath="~/Site.master" %>
```

Then add the following event wiring code in the `Page_PreInit` event handler:

```
Protected Sub Page_PreInit(ByVal sender As Object, ByVal e As EventArgs) Handles Me.PreInit
 AddHandler Master.PricesDoubled, AddressOf Master_PricesDoubled
End Sub
```

With this code in place, the GridView in the content page is refreshed whenever the `DoublePrice` Button is clicked.

Figures 8 and 9 illustrate this behavior. Figure 8 shows the page when first visited. Note that price values in both the `RecentProducts` GridView (in the left column of the master page) and the `ProductsGrid` GridView (in the content page). Figure 9 shows the same screen immediately after the `DoublePrice` Button has been clicked. As you can see, the new prices are instantaneously reflected in both GridViews.

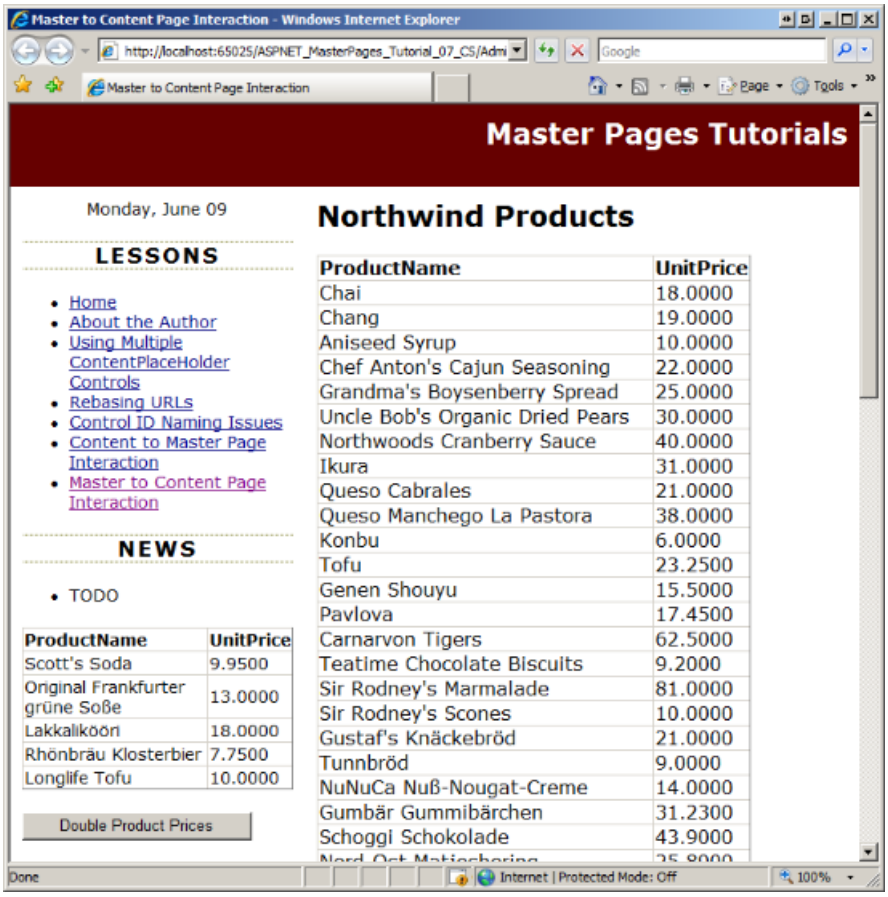


Figure 08: The Initial Price Values (Click to view full-size image)

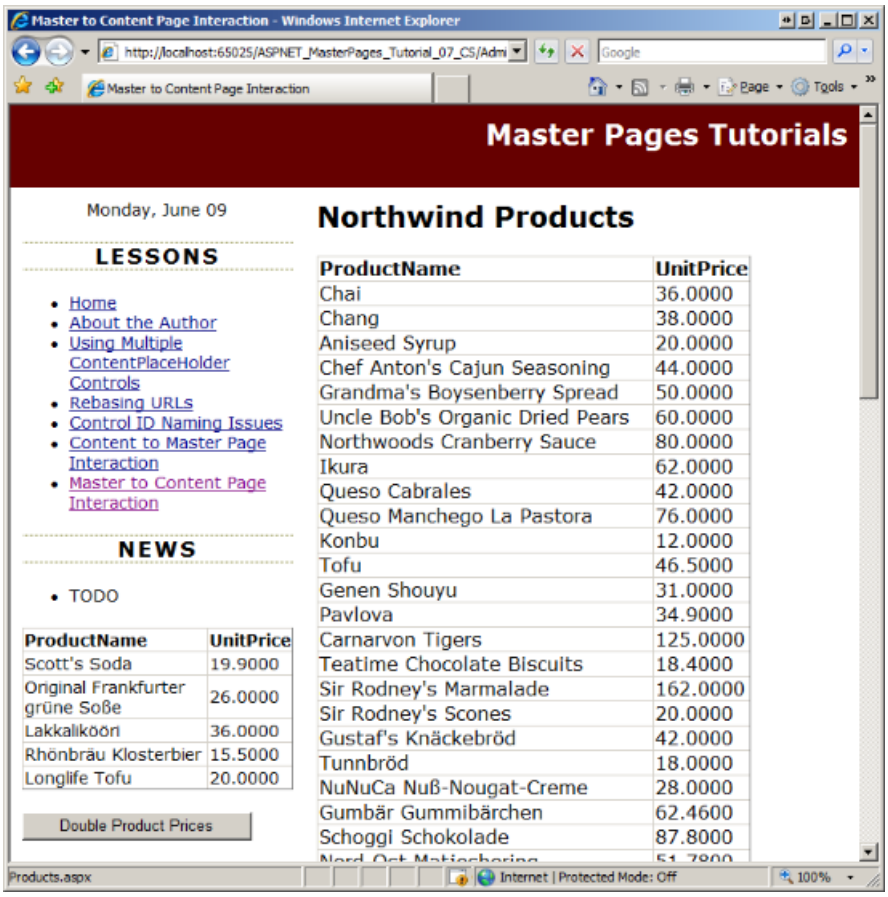


Figure 09: The Just-Doubled Prices are Displayed in the GridViews (Click to view full-size image)

Summary

Ideally, a master page and its content pages are completely separate from one another and require no level of interaction. However, if you have a master page or content page that displays data that can be modified from the master page or content page, then you may need to have the master page alert the content page (or vice-a-versa) when data is modified so that the display can be updated. In the preceding tutorial we saw how to have a content page programmatically interact with its master page; in this tutorial we looked at how to have a master page initiate the interaction.

While programmatic interaction between a content and master page can originate from either the content or master page, the interaction pattern used depends on the origination. The differences are due to the fact that a content page has a single master page, but a master page can have many different content pages. Rather than having a master page directly interact with a content page, a better approach is to have the master page raise an event to signal that some action has taken place. Those content pages that care about the action can create event handlers.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Accessing and Updating Data in ASP.NET
- Events and Delegates
- Passing Information Between Content and Master Pages
- Working with Data in ASP.NET Tutorials

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His

latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Suchi Banerjee. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Master Pages and ASP.NET AJAX (VB)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Discusses options for using ASP.NET AJAX and master pages. Looks at using the ScriptManagerProxy class; discusses how the various JS files are loaded depending on whether the ScriptManager is used in the Master page or Content page.

Introduction

Over the past several years, more and more developers have been building AJAX-enabled web applications. An AJAX-enabled website uses a number of related web technologies to offer a more responsive user experience. Creating AJAX-enabled ASP.NET applications is amazingly easy thanks to Microsoft's ASP.NET AJAX framework. ASP.NET AJAX is built into ASP.NET 3.5 and Visual Studio 2008; it is also available as a separate download for ASP.NET 2.0 applications.

When building AJAX-enabled web pages with the ASP.NET AJAX framework, you must add precisely one ScriptManager control to each and every page that uses the framework. As its name implies, the ScriptManager manages the client-side script used in AJAX-enabled web pages. At a minimum, the ScriptManager emits HTML that instructs the browser to download the JavaScript files that makeup the ASP.NET AJAX Client Library. It can also be used to register custom JavaScript files, script-enabled web services, and custom application service functionality.

If your site uses master pages (as it should), you do not necessarily need to add a ScriptManager control to every single content page; rather, you can add a ScriptManager control to the master page. This tutorial shows how to add the ScriptManager control to the master page. It also looks at how to use the ScriptManagerProxy control to register custom scripts and script services in a specific content page.

> **NOTE**
> This tutorial does not explore designing or building AJAX-enabled web applications with the ASP.NET AJAX framework. For more information on using AJAX consult the ASP.NET AJAX videos and tutorials, as well as those resources listed in the Further Reading section at the end of this tutorial.

Examining the Markup Emitted by the ScriptManager Control

The ScriptManager control emits markup that instructs the browser to download the JavaScript files that makeup the ASP.NET AJAX Client Library. It also adds a bit of inline JavaScript to the page that initializes this library. The following markup shows the content that is added to the rendered output of a page that includes a ScriptManager control:

```
<script src="/ASPNET_MasterPages_Tutorial_08_CS/WebResource.axd?d=T8EVk6SsA8mgPKu7_sBX5w2
t=633363040378379010" type="text/javascript"></script>

<script src="/ASPNET_MasterPages_Tutorial_08_CS/ScriptResource.axd?
d=SCE1TCh8B24VkQIU5a8iJFizuPBIqs6Lka7GEkxo-6ROKNw5LVPCpF_pmLFR-R-
p_Uf42Ahmr_SKd8lwgZUWb2uPJmfX0X_H6oLA50bniyQ1 t=633465688673751480" type="text/javascript"></script>

<script type="text/javascript">
//<![CDATA[
if (typeof(Sys) === 'undefined') throw new Error('ASP.NET Ajax client-side framework failed to load.');
//]]>
</script>

<script src="/ASPNET_MasterPages_Tutorial_08_CS/ScriptResource.axd?
d=SCE1TCh8B24VkQIU5a8iJFizuPBIqs6Lka7GEkxo-6ROKNw5LVPCpF_pmLFR-R-
phT96yZPngppiP_VXlN4Vz2RuVtvwDiQzF9xt42dUCiYjL0UylAJoyYzStwvObx0U0 t=633465688673751480"
type="text/javascript"></script>

<script type="text/javascript">
//<![CDATA[
Sys.WebForms.PageRequestManager._initialize('ScriptManager1', document.getElementById('form1'));
Sys.WebForms.PageRequestManager.getInstance()._updateControls([], [], [], 90);
//]]>
</script>

<script type="text/javascript">
//<![CDATA[
Sys.Application.initialize();
//]]>
</script>
```

The `<script src="url"></script>` tags instruct the browser to download and execute the JavaScript file at *url*. The ScriptManager emits three such tags; one references the file `WebResource.axd`, while the other two reference the file `ScriptResource.axd`. These files do not actually exist as files in your website. Instead, when a request for either one of these files arrives at the web server, the ASP.NET engine examines the querystring and returns the appropriate JavaScript content. The script provided by these three external JavaScript files constitute the ASP.NET AJAX framework's Client Library. The other `<script>` tags emitted by the ScriptManager include inline script that initializes this library.

The external script references and inline script emitted by the ScriptManager are essential for a page that uses the ASP.NET AJAX framework, but is not needed for pages that do not use the framework. Therefore, you might reason that it is ideal to only add a ScriptManager to those pages that use the ASP.NET AJAX framework. And this is sufficient, but if you have many pages that use the framework you'll end up adding the ScriptManager control to all pages - a repetitive task, to say the least. Alternatively, you can add a ScriptManager to the master page, which then injects this necessary script into all content pages. With this approach, you do not need to remember to add a ScriptManager to a new page that uses the ASP.NET AJAX framework because it is already included by the master page. Step 1 walks through adding a ScriptManager to the master page.

> **NOTE**
> If you plan on including AJAX functionality within the user interface of your master page, then you have no choice in the matter - you must include the ScriptManager in the master page.

One downside of adding the ScriptManager to the master page is that the above script is emitted in *every* page, regardless of whether its needed. This clearly leads to wasted bandwidth for those pages that have the ScriptManager included (via the master page) yet don't use any features of the ASP.NET AJAX framework. But just how much bandwidth is wasted?

- The actual content emitted by the ScriptManager (shown above) totals a little over 1KB.

- The three external script files referenced by the `<script>` element, however, comprise roughly 450KB of data uncompressed; in a website that uses gzip compression, this total bandwidth can be reduced near 100KB. However, these script files are cached by the browser for one year, meaning that they only need to be downloaded once and then can be reused in other pages on the site.

In the best case, then, when the script files are cached, the total cost is 1KB, which is negligible. In the worst case, however - which is when the script files have not yet been downloaded and the web server is not using any form of compression - the bandwidth hit is around 450KB, which can add anywhere from a second or two over a broadband connection to up to a minute for users over dial-up modems. The good news is that because the external script files are cached by the browser, this worst case scenario occurs infrequently.

> **NOTE**
> If you still feel uncomfortable placing the ScriptManager control in the master page, consider the Web Form (the `<form runat="server">` markup in the master page). Every ASP.NET page that uses the postback model must include precisely one Web Form. Adding a Web Form adds additional content: a number of hidden form fields, the `<form>` tag itself, and, if necessary, a JavaScript function for initiating a postback from script. This markup is unnecessary for pages that don't postback. This extraneous markup could be eliminated by removing the Web Form from the master page and manually adding it to each content page that needs one. However, the benefits of having the Web Form in the master page outweigh the disadvantages from having it added unnecessarily to certain content pages.

Step 1: Adding a ScriptManager Control to the Master Page

Every web page that uses the ASP.NET AJAX framework must contain precisely one ScriptManager control. Because of this requirement, it usually makes sense to place a single ScriptManager control on the master page so that all content pages have the ScriptManager control automatically included. Furthermore, the ScriptManager must come before any of the ASP.NET AJAX server controls, such as the UpdatePanel and UpdateProgress controls. Therefore, it's best to put the ScriptManager before any ContentPlaceHolder controls within the Web Form.

Open the `Site.master` master page and add a ScriptManager control to the page within the Web Form, but before the `<div id="topContent">` element (see Figure 1). If you are using Visual Web Developer 2008 or Visual Studio 2008, the ScriptManager control is located in the Toolbox in the AJAX Extensions tab. If you are using Visual Studio 2005, you will need to first install the ASP.NET AJAX framework and add the controls to the Toolbox. Visit the ASP.NET AJAX download page to get the framework for ASP.NET 2.0.

After adding the ScriptManager to the page, change its `ID` from `ScriptManager1` to `MyManager`.

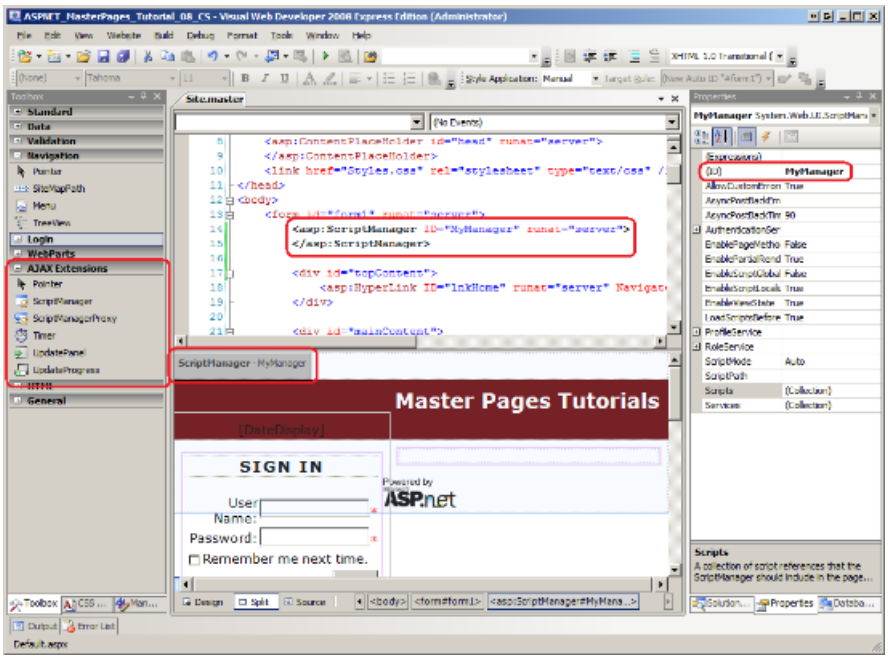


Figure 01: Add the ScriptManager to the Master Page (Click to view full-size image)

Step 2: Using the ASP.NET AJAX Framework from a Content Page

With the ScriptManager control added to the master page we can now add ASP.NET AJAX framework functionality to any content page. Let's create a new ASP.NET page that displays a randomly selected product from the Northwind database. We'll use the ASP.NET AJAX framework's Timer control to update this display every 15 seconds, showing a new product.

Start by creating a new page in the root directory named `ShowRandomProduct.aspx`. Don't forget to bind this new page to the `Site.master` master page.

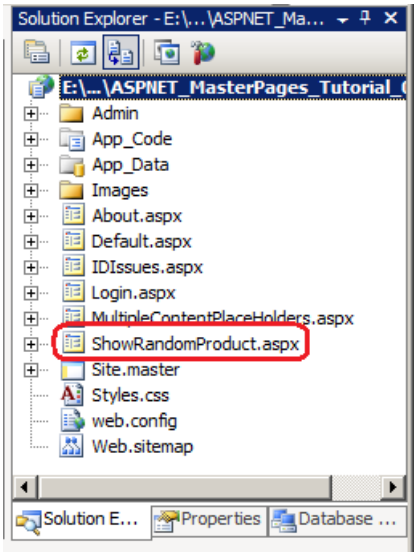


Figure 02: Add a New ASP.NET Page to the Website (Click to view full-size image)

Recall that in the Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page[SKM1] tutorial we created a custom base page class named `BasePage` that generated the page's title if it was not explicitly set. Go to the `ShowRandomProduct.aspx` page's code-behind class and have it derive from `BasePage` (instead of from `System.Web.UI.Page`).

Finally, update the `Web.sitemap` file to include an entry for this lesson. Add the following markup beneath the `<siteMapNode>` for the Master to Content Page Interaction lesson:

```
<siteMapNode url="~/ShowRandomProduct.aspx" title="Master Pages and ASP.NET AJAX" />
```

The addition of this `<siteMapNode>` element is reflected in the Lessons list (see Figure 5).

Displaying a Randomly Selected Product

Return to `ShowRandomProduct.aspx`. From the Designer, drag an UpdatePanel control from the Toolbox into the `MainContent` Content control and set its `ID` property to `ProductPanel`. The UpdatePanel represents a region on the screen that can be asynchronously updated through a partial page postback.

Our first task is to display information about a randomly selected product within the UpdatePanel. Start by dragging a DetailsView control into the UpdatePanel. Set the DetailsView control's `ID` property to `ProductInfo` and clear out its `Height` and `Width` properties. Expand the DetailsView's smart tag and, from the Choose Data Source drop-down list, choose to bind the DetailsView to a new SqlDataSource control named `RandomProductDataSource`.

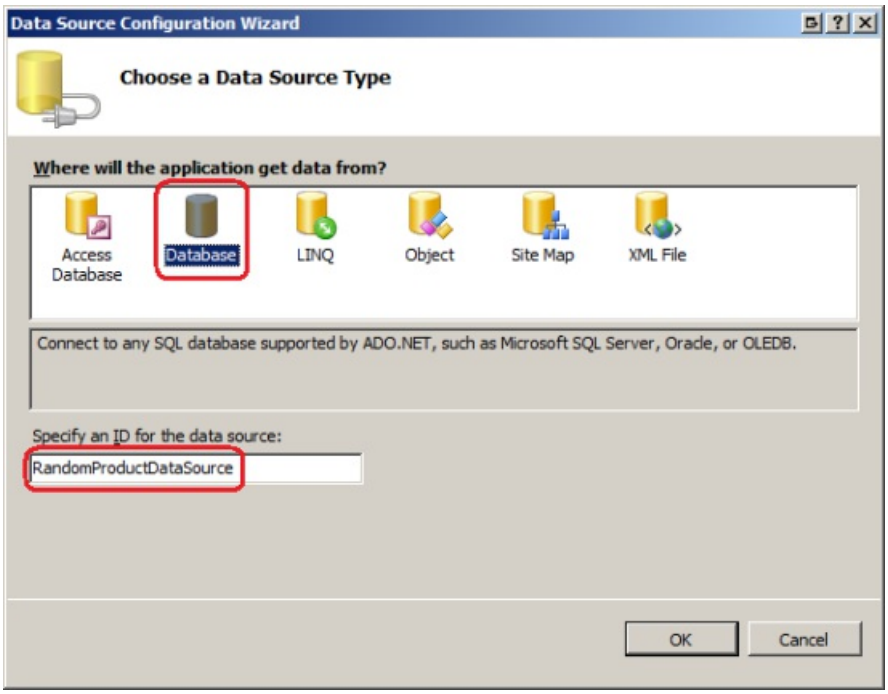


Figure 03: Bind the DetailsView to a New SqlDataSource Control (Click to view full-size image)

Configure the SqlDataSource control to connect to the Northwind database via the `NorthwindConnectionString` (which we created in the Interacting with the Master Page from the Content Page[SKM2] tutorial). When configuring the select statement choose to specify a custom SQL statement and then enter the following query:

```
SELECT TOP 1 ProductName, UnitPrice
FROM Products
ORDER BY NEWID()
```

The `TOP 1` keyword in the `SELECT` clause returns only the first record returned by the query. The `NEWID()` function generates a new globally unique identifier value (GUID) and can be used in an `ORDER BY` clause to return the table's records in a random order.

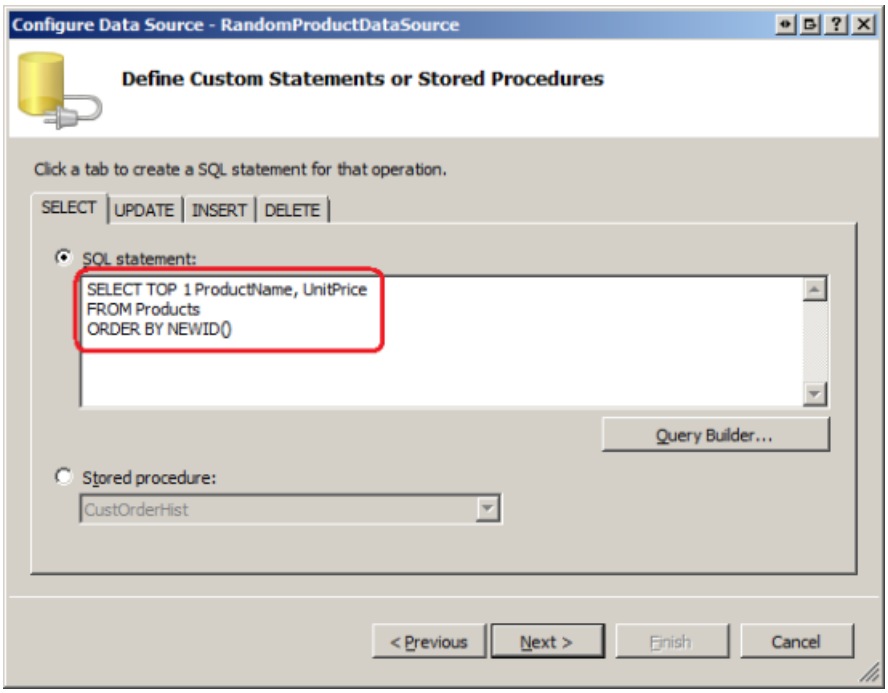


Figure 04: Configure the SqlDataSource to Return a Single, Randomly Selected Record (Click to view full-size image)

After completing the wizard, Visual Studio creates a BoundField for the two columns returned by the above query. At this point your page's declarative markup should look similar to the following:

```
<asp:UpdatePanel ID="ProductPanel" runat="server">
 <ContentTemplate>
 <asp:DetailsView ID="ProductInfo" runat="server" AutoGenerateRows="False"
 DataSourceID="RandomProductDataSource">
 <Fields>
 <asp:BoundField DataField="ProductName" HeaderText="ProductName"
 SortExpression="ProductName" />
 <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
 SortExpression="UnitPrice" />
 </Fields>
 </asp:DetailsView>
 <asp:SqlDataSource ID="RandomProductDataSource" runat="server"
 ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>" SelectCommand="SELECT TOP 1
ProductName, UnitPrice FROM Products ORDER BY NEWID()"></asp:SqlDataSource>
 </ContentTemplate>
</asp:UpdatePanel>
```

Figure 5 shows the `ShowRandomProduct.aspx` page when viewed through a browser. Click your browser's Refresh button to reload the page; you should see the `ProductName` and `UnitPrice` values for a new randomly selected record.

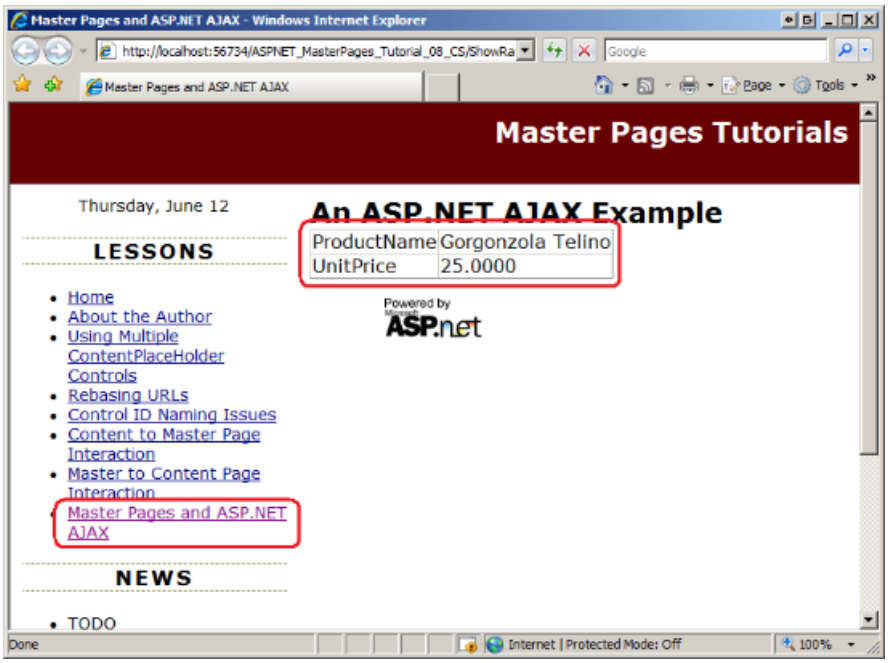


Figure 05: A Random Product's Name and Price is Displayed (Click to view full-size image)

Automatically Displaying a New Product Every 15 Seconds

The ASP.NET AJAX framework includes a Timer control that performs a postback at a specified time; on postback the Timer's `Tick` event is raised. If the Timer control is placed within an UpdatePanel it triggers a partial page postback, during which we can rebind the data to the DetailsView to display a new randomly selected product.

To accomplish this, drag a Timer from the Toolbox and drop it into the UpdatePanel. Change the Timer's `ID` from `Timer1` to `ProductTimer` and its `Interval` property from 60000 to 15000. The `Interval` property indicates the number of milliseconds between postbacks; setting it to 15000 causes the Timer to trigger a partial page postback every 15 seconds. At this point your Timer's declarative markup should look similar to the following:

```
<asp:UpdatePanel ID="ProductPanel" runat="server" onload="ProductPanel_Load">
 <ContentTemplate>
 ...

 <asp:Timer ID="ProductTimer" runat="server" Interval="15000">
 </asp:Timer>
 </ContentTemplate>
</asp:UpdatePanel>
```

Create an event handler for the Timer's `Tick` event. In this event handler we need to rebind the data to the DetailsView by calling the DetailsView's `DataBind` method. Doing so instructs the DetailsView to re-retrieve the data from its data source control, which will select and display a new randomly selected record (just like when reloading the page by clicking the browser's Refresh button).

```
Protected Sub ProductTimer_Tick(ByVal sender As Object, ByVal e As System.EventArgs) Handles
ProductTimer.Tick
 ProductInfo.DataBind()
End Sub
```

That's all there is to it! Revisit the page through a browser. Initially, a random product's information is displayed. If you patiently watch the screen you'll notice that, after 15 seconds, information about a new product magically replaces the existing display.

To better see what's happening here, let's add a Label control to the UpdatePanel that displays the time the display was last updated. Add a Label Web control within the UpdatePanel, set its `ID` to `LastUpdateTime`, and clear its

`Text` property. Next, create an event handler for the UpdatePanel's `Load` event and display the current time in the Label. (The UpdatePanel's `Load` event is fired on every full or partial page postback.)

```
Protected Sub ProductPanel_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles
ProductPanel.Load
 LastUpdateTime.Text = "Last updated at " & DateTime.Now.ToLongTimeString()
End Sub
```

With this change complete, the page includes the time the currently displayed product was loaded. Figure 6 shows the page when first visited. Figure 7 shows the page 15 seconds later after the Timer control has "ticked" and the UpdatePanel has been refreshed to display information about a new product.

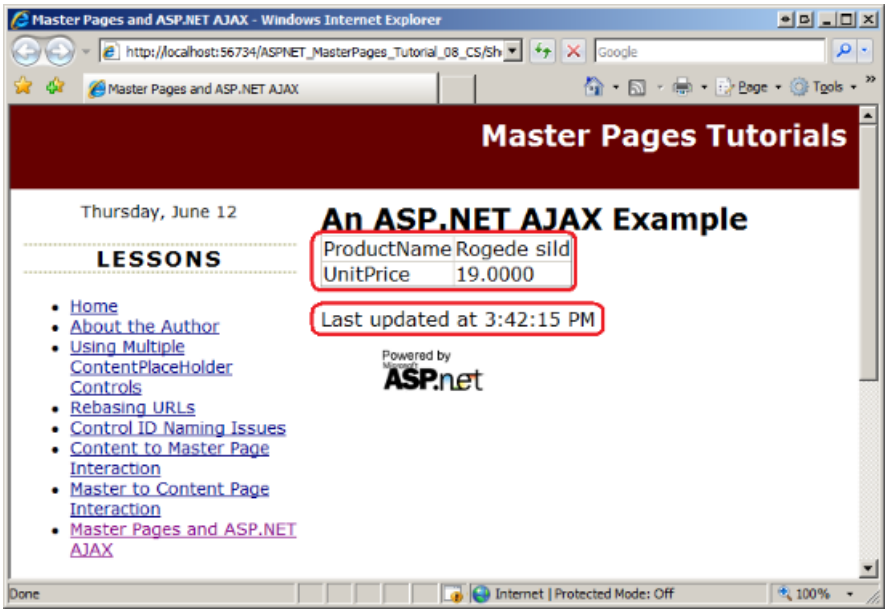


Figure 06: A Randomly Selected Product is Displayed on Page Load (Click to view full-size image)

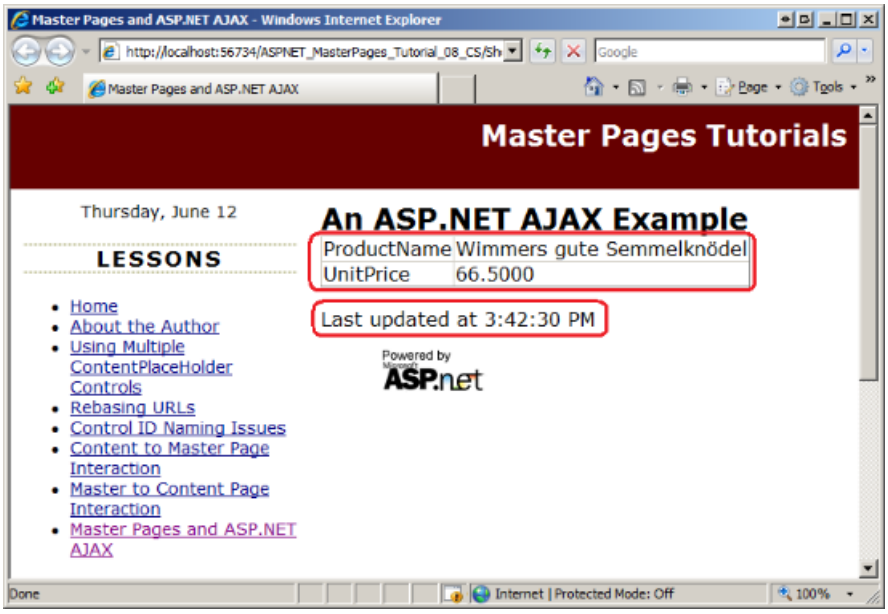


Figure 07: Every 15 Seconds a New Randomly Selected Product is Displayed (Click to view full-size image)

Step 3: Using the ScriptManagerProxy Control

Along with including the necessary script for the ASP.NET AJAX framework Client Library, the ScriptManager can also register custom JavaScript files, references to script-enabled Web Services, and custom authentication, authorization, and profile services. Usually such customizations are specific to a certain page. However, if the custom script files, Web Service references, or authentication, authorization, or profile services are referenced in

the ScriptManager in the master page then they are included in all pages in the website.

To add ScriptManager-related customizations on a page-by-page basis use the ScriptManagerProxy control. You can add a ScriptManagerProxy to a content page and then register the custom JavaScript file, Web Service reference, or authentication, authorization, or profile service from the ScriptManagerProxy; this has the effect of registering these services for the particular content page.

> **NOTE**
>
> An ASP.NET page can only have no more than one ScriptManager control present. Therefore, you cannot add a ScriptManager control to a content page if the ScriptManager control is already defined in the master page. The sole purpose of the ScriptManagerProxy is to provide a way for developers to define the ScriptManager in the master page, but still have the ability to add ScriptManager customizations on a page-by-page basis.

To see the ScriptManagerProxy control in action, let's augment the UpdatePanel in `ShowRandomProduct.aspx` to include a button that uses client-side script to pause or resume the Timer control. The Timer control has three client-side methods that we can use to achieve this desired functionality:

- `_startTimer()` - starts the Timer control
- `_raiseTick()` - causes the Timer control to "tick," thereby posting back and raising its Tick event on the server
- `_stopTimer()` - stops the Timer control

Let's create a JavaScript file with a variable named `timerEnabled` and a function named `ToggleTimer`. The `timerEnabled` variable indicates whether the Timer control is currently enabled or disabled; it defaults to true. The `ToggleTimer` function accepts two input parameters: a reference to the Pause/Resume button and the client-side `id` value of the Timer control. This function toggles the value of `timerEnabled`, gets a reference to the Timer control, starts or stops the Timer (depending on the value of `timerEnabled`), and updates the button's display text to "Pause" or "Resume". This function will be called whenever the Pause/Resume button is clicked.

Start by creating a new folder in the website named `Scripts`. Next, add a new file to the Scripts folder named `TimerScript.js` of type JScript File.

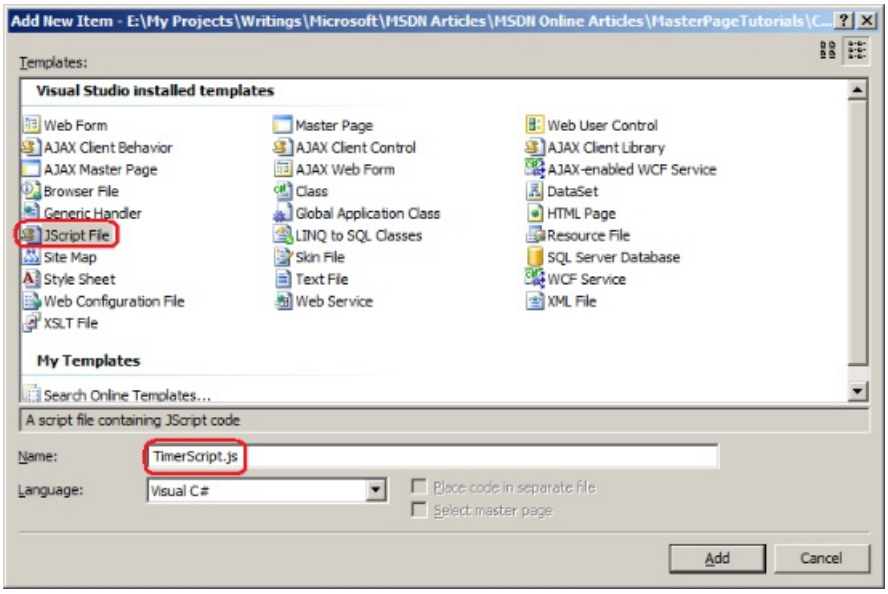


Figure 08: Add a New JavaScript File to the `Scripts` Folder (Click to view full-size image)

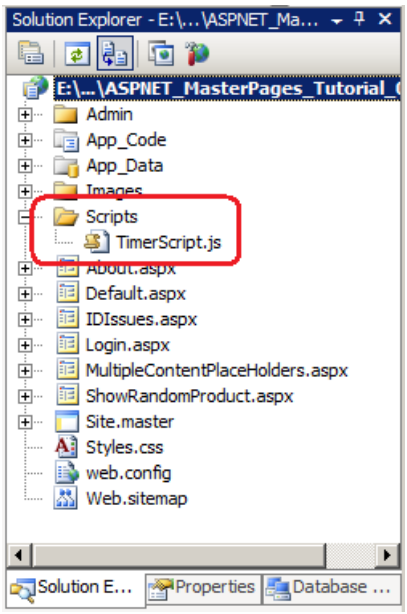


Figure 09: A New JavaScript File has been Added to the Website (Click to view full-size image)

Next, add the following scrip to the `TimerScript.js` file:

```
var timerEnabled = true;
function ToggleTimer(btn, timerID)
{
    // Toggle the timer enabled state
    timerEnabled = !timerEnabled;

    // Get a reference to the Timer
    var timer = $find(timerID);

    if (timerEnabled)
    {
        // Start timer
        timer._startTimer();

        // Immediately raise a tick
        timer._raiseTick();

        btn.value = 'Pause';
    }
    else
    {
        // Stop timer
        timer._stopTimer();

        btn.value = 'Resume';
    }
}
```

We now need to register this custom JavaScript file in `ShowRandomProduct.aspx`. Return to `ShowRandomProduct.aspx` and add a ScriptManagerProxy control to the page; set its `ID` to `MyManagerProxy`. To register a custom JavaScript file select the ScriptManagerProxy control in the Designer and then go to the Properties window. One of the properties is titled Scripts. Selecting this property displays the ScriptReference Collection Editor shown in Figure 10. Click the Add button to include a new script reference and then enter the path to the script file in the Path property: `~/Scripts/TimerScript.js`.

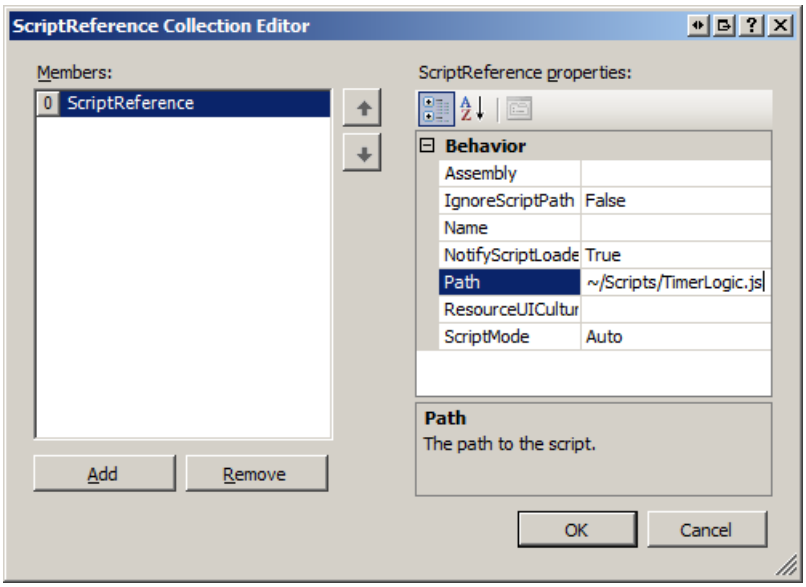


Figure 10: Add a Script Reference to the ScriptManagerProxy Control (Click to view full-size image)

After adding the script reference the ScriptManagerProxy control's declarative markup is updated to include a `<Scripts>` collection with a single `ScriptReference` entry, as the following snippet of markup illustrates:

```
<asp:ScriptManagerProxy ID="MyManagerProxy" runat="server">
 <Scripts>
 <asp:ScriptReference Path="~/Scripts/TimerScript.js" />
 </Scripts>
</asp:ScriptManagerProxy>
```

The `ScriptReference` entry instructs the ScriptManagerProxy to include a reference to the JavaScript file in its rendered markup. That is, by registering the custom script in the ScriptManagerProxy the `ShowRandomProduct.aspx` page's rendered output now includes another `<script src="url"></script>` tag: `<script src="Scripts/TimerScript.js" type="text/javascript"></script>`.

We can now call the `ToggleTimer` function defined in `TimerScript.js` from the client script in the `ShowRandomProduct.aspx` page. Add the following HTML within the UpdatePanel:

```
<input type="button" id="PauseResumeButton"
    value="Pause"
    onclick="ToggleTimer(this, '<%=ProductTimer.ClientID %>');" />
```

This displays a button with the text "Pause". Whenever it is clicked, the JavaScript function `ToggleTimer` is called, passing in a reference to the button and the `id` value of the Timer control (`ProductTimer`). Note the syntax for obtaining the `id` value of the Timer control. `<%=ProductTimer.ClientID%>` emits the value of the `ProductTimer` Timer control's `ClientID` property. In the Control ID Naming in Content Pages[SKM3] tutorial we discussed the differences between the server-side `ID` value and the resulting client-side `id` value, and how `ClientID` returns the client-side `id`.

Figure 11 shows this page when first visited through a browser. The Timer is currently running and updates the displayed product information every 15 seconds. Figure 12 shows the screen after the Pause button has been clicked. Clicking the Pause button stops the Timer and updates the button's text to "Resume". The product information will refresh (and continue to refresh every 15 seconds) once the user clicks Resume.

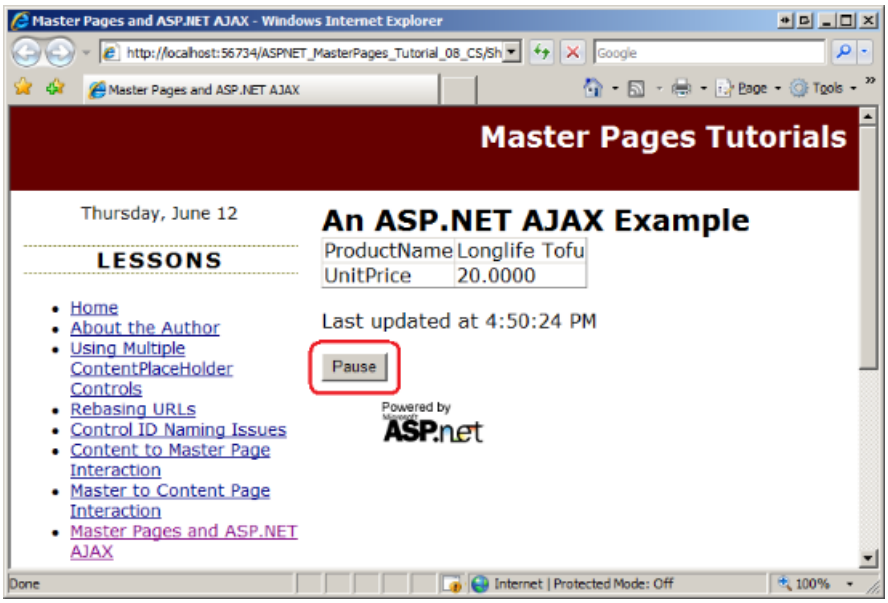


Figure 11: Click the Pause Button to Stop the Timer Control (Click to view full-size image)

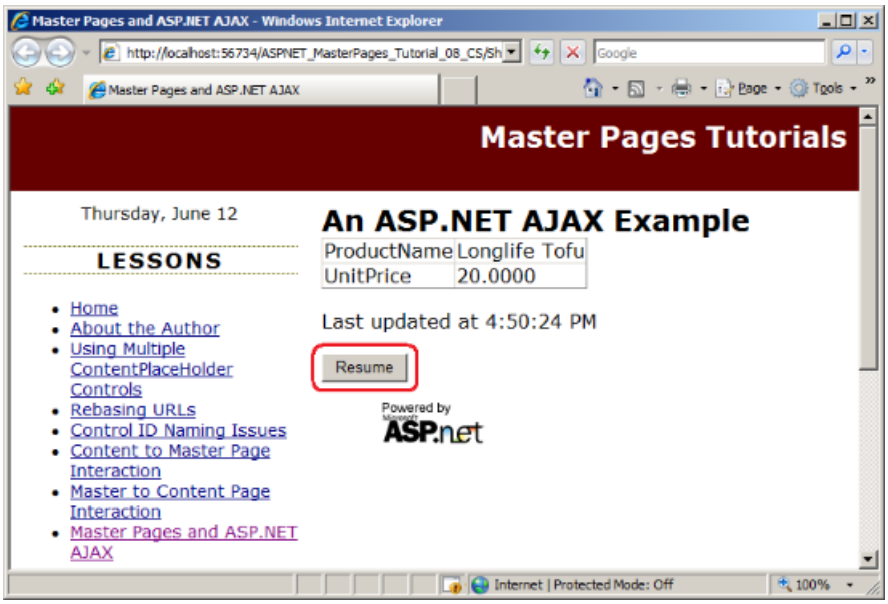


Figure 12: Click the Resume Button to Restart the Timer (Click to view full-size image)

Summary

When building AJAX-enabled web applications using the ASP.NET AJAX framework it is imperative that every AJAX-enabled web page include a ScriptManager control. To facilitate this process, we can add a ScriptManager to the master page rather than having to remember to add a ScriptManager to each and every content page. Step 1 showed how to add the ScriptManager to the master page while Step 2 looked at implementing AJAX functionality in a content page.

If you need to add custom scripts, references to script-enabled Web Services, or customized authentication, authorization, or profile services to a particular content page, add a ScriptManagerProxy control to the content page and then configure the customizations there. Step 3 examined how to use the ScriptManagerProxy to register a custom JavaScript file in a specific content page.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET AJAX Framework

- ASP.NET AJAX Tutorials
- ASP.NET AJAX Videos
- Building Interactive User Interface with Microsoft ASP.NET AJAX
- Using NEWID to Randomly Sort Records
- Using the Timer Control

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Specifying the Master Page Programmatically (VB)

2/20/2020 • 19 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> Looks at setting the content page's master page programmatically via the PreInit event handler.

Introduction

Since the inaugural example in *Creating a Site-Wide Layout Using Master Pages*, all content pages have referenced their master page declaratively via the `MasterPageFile` attribute in the `@Page` directive. For example, the following `@Page` directive links the content page to the master page `Site.master`:

```
<%@ Page Language="C#" MasterPageFile="~/Site.master"... %>
```

The `Page` class in the `System.Web.UI` namespace includes a `MasterPageFile` property that returns the path to the content page's master page; it is this property that is set by the `@Page` directive. This property can also be used to programmatically specify the content page's master page. This approach is useful if you want to dynamically assign the master page based on external factors, such as the user visiting the page.

In this tutorial we add a second master page to our website and dynamically decide which master page to use at runtime.

Step 1: A Look at the Page Lifecycle

Whenever a request arrives at the web server for an ASP.NET page that is a content page, the ASP.NET engine must fuse the page's Content controls into the master page's corresponding ContentPlaceHolder controls. This fusion creates a single control hierarchy that can then proceed through the typical page lifecycle.

Figure 1 illustrates this fusion. Step 1 in Figure 1 shows the initial content and master page control hierarchies. At the tail end of the PreInit stage the Content controls in the page are added to the corresponding ContentPlaceHolders in the master page (Step 2). After this fusion, the master page serves as the root of the fused control hierarchy. This fused control hierarchy is then added to the page to produce the finalized control hierarchy (Step 3). The net result is that the page's control hierarchy includes the fused control hierarchy.

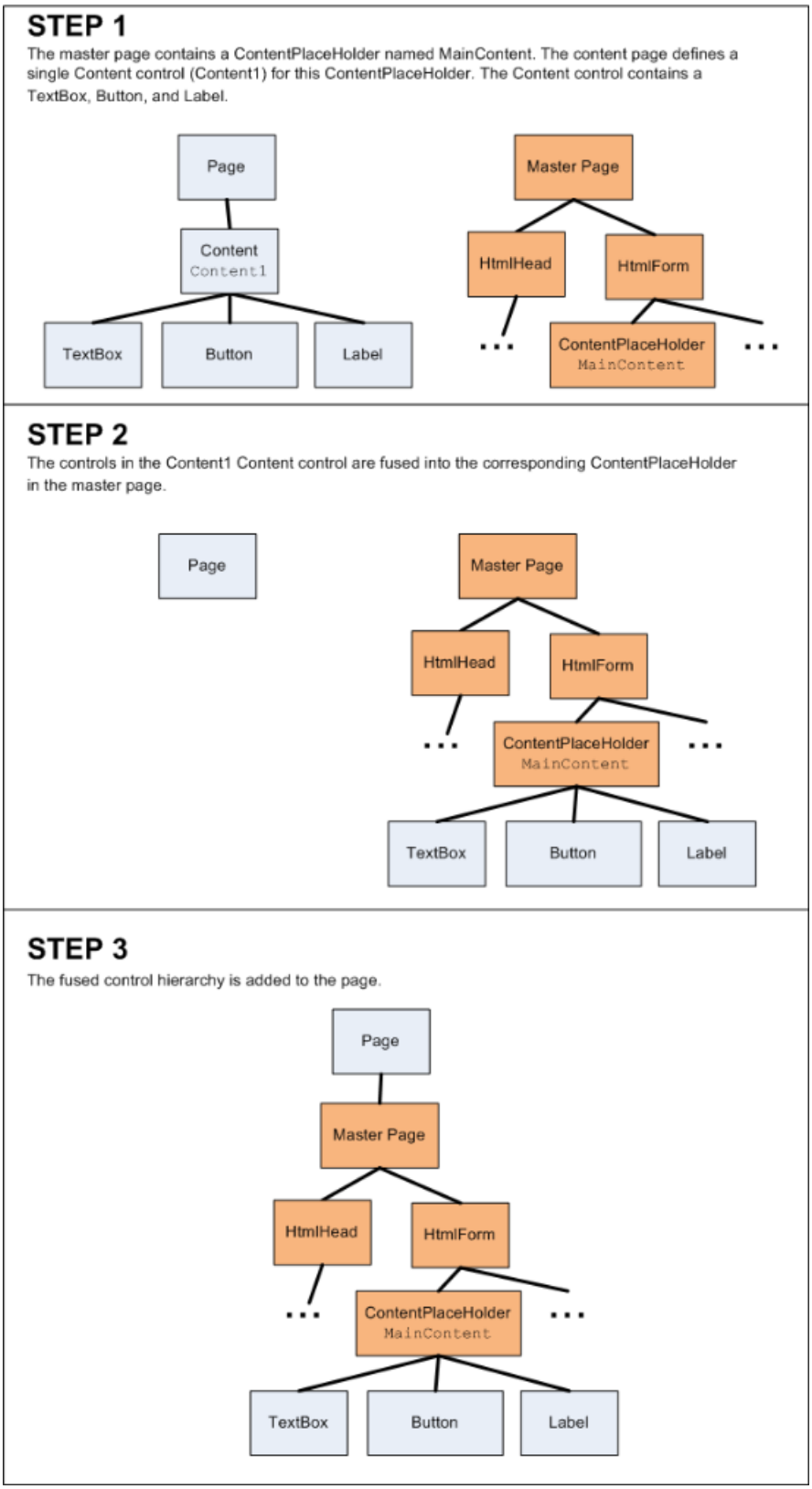


Figure 01: The Master Page and Content Page's Control Hierarchies are Fused Together during the PreInit Stage (Click to view full-size image)

Step 2: Setting the `MasterPageFile` Property from Code

What master page partakes in this fusion depends on the value of the `Page` object's `MasterPageFile` property. Setting the `MasterPageFile` attribute in the `@Page` directive has the net effect of assigning the `Page`'s `MasterPageFile` property during the Initialization stage, which is the very first stage of the page's lifecycle. We can alternatively set this property programmatically. However, it is imperative that this property be set before the fusion in Figure 1 takes place.

At the start of the PreInit stage the `Page` object raises its `PreInit` event and calls its `OnPreInit` method. To set the master page programmatically, then, we can either create an event handler for the `PreInit` event or override the `OnPreInit` method. Let's look at both approaches.

Start by opening `Default.aspx.vb`, the code-behind class file for our site's homepage. Add an event handler for the page's `PreInit` event by typing in the following code:

```
Protected Sub Page_PreInit(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.PreInit
End Sub
```

From here we can set the `MasterPageFile` property. Update the code so that it assigns the value "~/Site.master" to the `MasterPageFile` property.

```
Protected Sub Page_PreInit(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.PreInit
 Me.MasterPageFile = "~/Site.master"
End Sub
```

If you set a breakpoint and start with debugging you'll see that whenever the `Default.aspx` page is visited or whenever there's a postback to this page, the `Page_PreInit` event handler executes and the `MasterPageFile` property is assigned to "~/Site.master".

Alternatively, you can override the `Page` class's `OnPreInit` method and set the `MasterPageFile` property there. For this example, let's not set the master page in a particular page, but rather from `BasePage`. Recall that we created a custom base page class (`BasePage`) back in the *Specifying the Title, Meta Tags, and Other HTML Headers in the Master Page* tutorial. Currently `BasePage` overrides the `Page` class's `OnLoadComplete` method, where it sets the page's `Title` property based on the site map data. Let's update `BasePage` to also override the `OnPreInit` method to programmatically specify the master page.

```
Protected Overrides Sub OnPreInit(ByVal e As System.EventArgs)
 Me.MasterPageFile = "~/Site.master"
 MyBase.OnPreInit(e)
End Sub
```

Because all our content pages derive from `BasePage`, all of them now have their master page programmatically assigned. At this point the `PreInit` event handler in `Default.aspx.vb` is superfluous; feel free to remove it.

What About the `@Page` Directive?

What may be a little confusing is that the content pages' `MasterPageFile` properties are now being specified in two places: programmatically in the `BasePage` class's `OnPreInit` method as well as through the `MasterPageFile` attribute in each content page's `@Page` directive.

The first stage in the page lifecycle is the Initialization stage. During this stage the `Page` object's `MasterPageFile` property is assigned the value of the `MasterPageFile` attribute in the `@Page` directive (if it is provided). The PreInit stage follows the Initialization stage, and it is here where we programmatically set the `Page` object's `MasterPageFile` property, thereby overwriting the value assigned from the `@Page` directive. Because we are setting the `Page` object's `MasterPageFile` property programmatically, we could remove the `MasterPageFile` attribute from the `@Page` directive without affecting the end user's experience. To convince yourself of this, go ahead and remove the `MasterPageFile` attribute from the `@Page` directive in `Default.aspx` and then visit the page through a browser. As you would expect, the output is the same as before the attribute was removed.

Whether the `MasterPageFile` property is set via the `@Page` directive or programmatically is inconsequential to the end user's experience. However, the `MasterPageFile` attribute in the `@Page` directive is used by Visual Studio during design-time to produce the WYSIWYG view in the Designer. If you return to `Default.aspx` in Visual Studio

and navigate to the Designer you'll see the message, "Master Page error: The page has controls that require a Master Page reference, but none is specified" (see Figure 2).

In short, you need to leave the `MasterPageFile` attribute in the `@Page` directive to enjoy a rich design-time experience in Visual Studio.

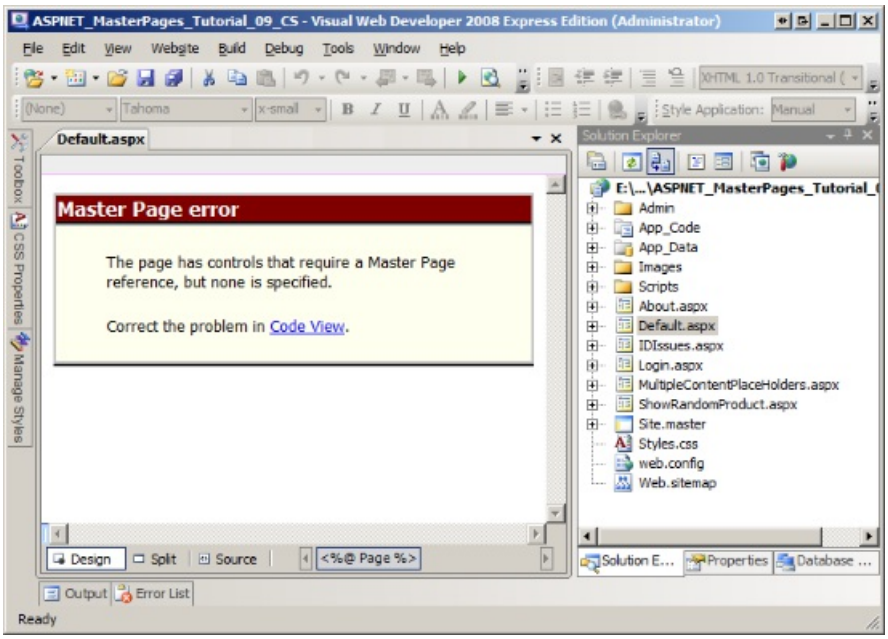


Figure 02: Visual Studio Uses the `@Page` Directive's `MasterPageFile` Attribute to Render the Design View (Click to view full-size image)

Step 3: Creating an Alternative Master Page

Because a content page's master page can be set programmatically at runtime it's possible to dynamically load a particular master page based on some external criteria. This functionality can be useful in situations where the site's layout needs to vary based on the user. For instance, a blog engine web application may allow its users to choose a layout for their blog, where each layout is associated with a different master page. At runtime, when a visitor is viewing a user's blog, the web application would need to determine the blog's layout and dynamically associate the corresponding master page with the content page.

Let's examine how to dynamically load a master page at runtime based on some external criteria. Our website currently contains just one master page (`Site.master`). We need another master page to illustrate choosing a master page at runtime. This step focuses on creating and configuring the new master page. Step 4 looks at determining what master page to use at runtime.

Create a new master page in the root folder named `Alternate.master`. Also add a new style sheet to the website named `AlternateStyles.css`.

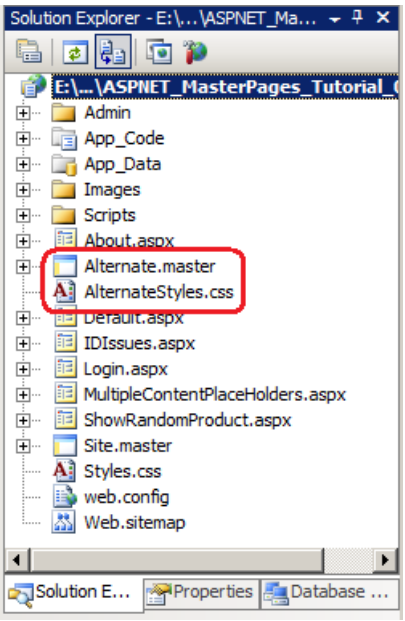


Figure 03: Add Another Master Page and CSS File to the Website (Click to view full-size image)

I've designed the `Alternate.master` master page to have the title displayed at the top of the page, centered and on a navy background. I've dispensed of the left column and moved that content beneath the `MainContent` ContentPlaceHolder control, which now spans the entire width of the page. Furthermore, I nixed the unordered Lessons list and replaced it with a horizontal list above `MainContent`. I also updated the fonts and colors used by the master page (and, by extension, its content pages). Figure 4 shows `Default.aspx` when using the `Alternate.master` master page.

> **NOTE**
>
> ASP.NET includes the ability to define *Themes*. A Theme is a collection of images, CSS files, and style-related Web control property settings that can be applied to a page at runtime. Themes are the way to go if your site's layouts differ only in the images displayed and by their CSS rules. If the layouts differ more substantially, such as using different Web controls or having a radically different layout, then you will need to use separate master pages. Consult the Further Reading section at the end of this tutorial for more information on Themes.

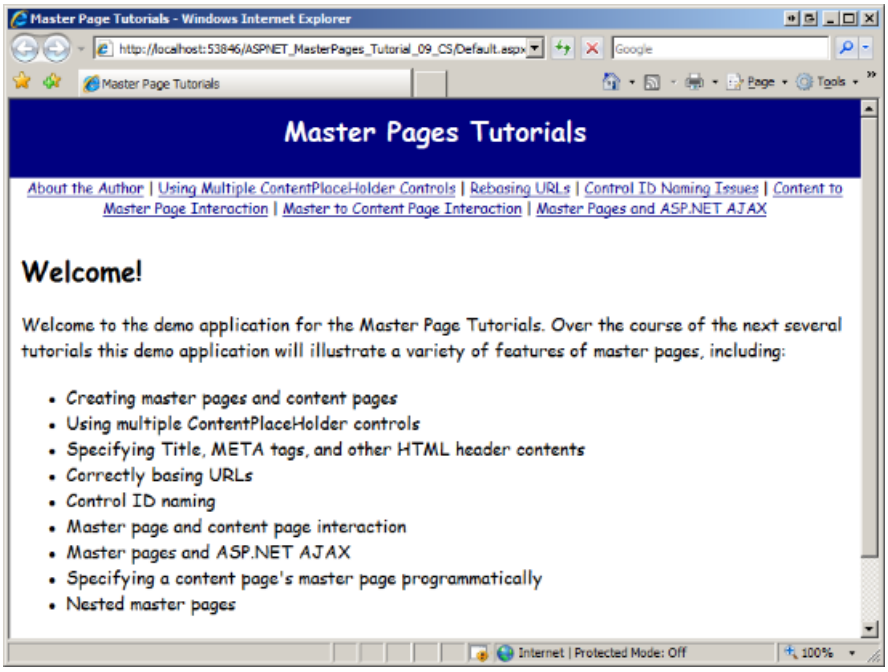


Figure 04: Our Content Pages Can Now Use a New Look and Feel (Click to view full-size image)

When the master and content pages' markup are fused, the `MasterPage` class checks to ensure that every Content control in the content page references a ContentPlaceHolder in the master page. An exception is thrown if a Content control that references a non-existent ContentPlaceHolder is found. In other words, it is imperative that the master page being assigned to the content page have a ContentPlaceHolder for each Content control in the content page.

The `Site.master` master page includes four ContentPlaceHolder controls:

- `head`
- `MainContent`
- `QuickLoginUI`
- `LeftColumnContent`

Some of the content pages in our website include just one or two Content controls; others include a Content control for each of the available ContentPlaceHolders. If our new master page (`Alternate.master`) may ever be assigned to those content pages that have Content controls for all of the ContentPlaceHolders in `Site.master` then it is essential that `Alternate.master` also include the same ContentPlaceHolder controls as `Site.master`.

To get your `Alternate.master` master page to look similar to mine (see Figure 4), start by defining the master page's styles in the `AlternateStyles.css` style sheet. Add the following rules into `AlternateStyles.css`:

```
body
{
 font-family: Comic Sans MS, Arial;
 font-size: medium;
 margin: 0px;
}
#topContent
{
 text-align: center;
 background-color: Navy;
 color: White;
 font-size: x-large;
 text-decoration: none;
 font-weight: bold;
 padding: 10px;
 height: 50px;
}
#topContent a
{
 text-decoration: none;
 color: White;
}
#navContent
{
 font-size: small;
 text-align: center;
}
#footerContent
{
 padding: 10px;
 font-size: 90%;
 text-align: center;
 border-top: solid 1px black;
}
#mainContent
{
 text-align: left;
 padding: 10px;
}
```

Next, add the following declarative markup to `Alternate.master`. As you can see, `Alternate.master` contains four ContentPlaceHolder controls with the same `ID` values as the ContentPlaceHolder controls in `Site.master`. Moreover, it includes a ScriptManager control, which is necessary for those pages in our website that use the ASP.NET AJAX framework.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
 <link href="AlternateStyles.css" rel="stylesheet" type="text/css"/>
</head>
<body>
 <form id="form1" runat="server">
 <asp:ScriptManager ID="MyManager" runat="server">
 </asp:ScriptManager>
 <div id="topContent">
 <asp:HyperLink ID="lnkHome" runat="server" NavigateUrl="~/Default.aspx"
 Text="Master Pages Tutorials" />
 </div>
 <div id="navContent">
 <asp:ListView ID="LessonsList" runat="server"
 DataSourceID="LessonsDataSource">
 <LayoutTemplate>
 <asp:PlaceHolder runat="server" ID="itemPlaceholder" />
 </LayoutTemplate>
 <ItemTemplate>
 <asp:HyperLink runat="server" ID="lnkLesson"
 NavigateUrl='<%# Eval("Url") %>'
 Text='<%# Eval("Title") %>' />
 </ItemTemplate>
 <ItemSeparatorTemplate> | </ItemSeparatorTemplate>
 </asp:ListView>
 <asp:SiteMapDataSource ID="LessonsDataSource" runat="server"
 ShowStartingNode="false" />
 </div>
 <div id="mainContent">
 <asp:ContentPlaceHolder id="MainContent" runat="server">
 </asp:ContentPlaceHolder>
 </div>
 <div id="footerContent">
 <p>
 <asp:Label ID="DateDisplay" runat="server"></asp:Label>
 </p>
 <asp:ContentPlaceHolder ID="QuickLoginUI" runat="server">
 </asp:ContentPlaceHolder>
 <asp:ContentPlaceHolder ID="LeftColumnContent" runat="server">
 </asp:ContentPlaceHolder>
 </div>
 </form>
</body>
</html>
```

Testing the New Master Page

To test this new master page update the `BasePage` class's `OnPreInit` method so that the `MasterPageFile` property is assigned the value `"~/Alternate.maser"` and then visit the website. Every page should function without error except for two: `~/Admin/AddProduct.aspx` and `~/Admin/Products.aspx`. Adding a product to the DetailsView in `~/Admin/AddProduct.aspx` results in a `NullReferenceException` from the line of code that attempts to set the master page's `GridMessageText` property. When visiting `~/Admin/Products.aspx` an `InvalidCastException` is thrown on page load with the message: "Unable to cast object of type 'ASP.alternate_master' to type 'ASP.site_master'."

These errors occur because the `Site.master` code-behind class includes public events, properties, and methods that are not defined in `Alternate.master`. The markup portion of these two pages have a `@MasterType` directive that references the `Site.master` master page.

```
<%@ MasterType VirtualPath="~/Site.master" %>
```

Also, the DetailsView's `ItemInserted` event handler in `~/Admin/AddProduct.aspx` includes code that casts the loosely-typed `Page.Master` property to an object of type `Site`. The `@MasterType` directive (used this way) and the cast in the `ItemInserted` event handler tightly couples the `~/Admin/AddProduct.aspx` and `~/Admin/Products.aspx` pages to the `Site.master` master page.

To break this tight coupling we can have `Site.master` and `Alternate.master` derive from a common base class that contains definitions for the public members. Following that, we can update the `@MasterType` directive to reference this common base type.

Creating a Custom Base Master Page Class

Add a new class file to the `App_Code` folder named `BaseMasterPage.vb` and have it derive from `System.Web.UI.MasterPage`. We need to define the `RefreshRecentProductsGrid` method and the `GridMessageText` property in `BaseMasterPage`, but we can't simply move them there from `Site.master` because these members work with Web controls that are specific to the `Site.master` master page (the `RecentProducts` GridView and `GridMessage` Label).

What we need to do is configure `BaseMasterPage` in such a way that these members are defined there, but are actually implemented by `BaseMasterPage`'s derived classes (`Site.master` and `Alternate.master`). This type of inheritance is possible by marking the class as `MustInherit` and its members as `MustOverride`. In short, adding these keywords to the class and its two members announces that `BaseMasterPage` hasn't implemented `RefreshRecentProductsGrid` and `GridMessageText`, but that its derived classes will.

We also need to define the `PricesDoubled` event in `BaseMasterPage` and provide a means by the derived classes to raise the event. The pattern used in the .NET Framework to facilitate this behavior is to create a public event in the base class and add a protected, overridable method named `OnEventName`. Derived classes can then call this method to raise the event or can override it to execute code immediately before or after the event is raised.

Update your `BaseMasterPage` class so that it contains the following code:

```
Public MustInherit Class BaseMasterPage
 Inherits System.Web.UI.MasterPage
 Public Event PricesDoubled As EventHandler
 Protected Overridable Sub OnPricesDoubled(ByVal e As EventArgs)
 RaiseEvent PricesDoubled(Me, e)
 End Sub
 Public MustOverride Sub RefreshRecentProductsGrid()
 Public MustOverride Property GridMessageText() As String
End Class
```

Next, go to the `Site.master` code-behind class and have it derive from `BaseMasterPage`. Because `BaseMasterPage` contains members marked `MustOverride` we need to override those members here in `Site.master`. Add the `Overrides` keyword to the method and property definitions. Also update the code that raises the `PricesDoubled` event in the `DoublePrice` Button's `Click` event handler with a call to the base class's `OnPricesDoubled` method.

After these modifications the `Site.master` code-behind class should contain the following code:

```
Partial Class Site
 Inherits BaseMasterPage
 Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
 DateDisplay.Text = DateTime.Now.ToString("dddd, MMMM dd")
 End Sub
 Public Overrides Sub RefreshRecentProductsGrid()
 RecentProducts.DataBind()
 End Sub
 Public Overrides Property GridMessageText() As String
 Get
 Return GridMessage.Text
 End Get
 Set(ByVal Value As String)
 GridMessage.Text = Value
 End Set
 End Property
 Protected Sub DoublePrice_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
DoublePrice.Click
 ' Double the prices
 DoublePricesDataSource.Update()
 ' Refresh RecentProducts
 RecentProducts.DataBind()
 ' Raise the PricesDoubled event
 MyBase.OnPricesDoubled(EventArgs.Empty)
 End Sub
End Class
```

We also need to update `Alternate.master`'s code-behind class to derive from `BaseMasterPage` and override the two `MustOverride` members. But because `Alternate.master` does not contain a GridView that lists the most recent products nor a Label that displays a message after a new product is added to the database, these methods do not need to do anything.

```
Partial Class Alternate
 Inherits BaseMasterPage
 Public Overrides Property GridMessageText() As String
 Get
 Return String.Empty
 End Get
 Set(ByVal value As String)
 ' Do nothing
 End Set
 End Property
 Public Overrides Sub RefreshRecentProductsGrid()
 ' Do nothing
 End Sub
End Class
```

Referencing the Base Master Page Class

Now that we have completed the `BaseMasterPage` class and have our two master pages extending it, our final step is to update the `~/Admin/AddProduct.aspx` and `~/Admin/Products.aspx` pages to refer to this common type. Start by changing the `@MasterType` directive in both pages from:

```
<%@ MasterType VirtualPath="~/Site.master" %>
```

To:

```
<%@ MasterType TypeName="BaseMasterPage" %>
```

Rather than referencing a file path, the `@MasterType` property now references the base type (`BaseMasterPage`). Consequently, the strongly-typed `Master` property used in both pages' code-behind classes is now of type `BaseMasterPage` (instead of type `Site`). With this change in place revisit `~/Admin/Products.aspx`. Previously, this resulted in a casting error because the page is configured to use the `Alternate.master` master page, but the `@MasterType` directive referenced the `Site.master` file. But now the page renders without error. This is because the `Alternate.master` master page can be cast to an object of type `BaseMasterPage` (since it extends it).

There's one small change that needs to be made in `~/Admin/AddProduct.aspx`. The DetailsView control's `ItemInserted` event handler uses both the strongly-typed `Master` property and the loosely-typed `Page.Master` property. We fixed the strongly-typed reference when we updated the `@MasterType` directive, but we still need to update the loosely-typed reference. Replace the following line of code:

```
Dim myMasterPage As Site = CType(Page.Master, Site)
```

With the following, which casts `Page.Master` to the base type:

```
Dim myMasterPage As BaseMasterPage = CType(Page.Master, BaseMasterPage)
```

Step 4: Determining What Master Page to Bind to the Content Pages

Our `BasePage` class currently sets all content pages' `MasterPageFile` properties to a hard-coded value in the PreInit stage of the page lifecycle. We can update this code to base the master page on some external factor. Perhaps the master page to load depends on the preferences of the currently logged on user. In that case, we'd need to write code in the `OnPreInit` method in `BasePage` that looks up the currently visiting user's master page preferences.

Let's create a web page that allows the user to choose which master page to use - `Site.master` or `Alternate.master` - and save this choice in a Session variable. Start by creating a new web page in the root directory named `ChooseMasterPage.aspx`. When creating this page (or any other content pages henceforth) you don't need to bind it to a master page because the master page is set programmatically in `BasePage`. However, if you do not bind the new page to a master page then the new page's default declarative markup contains a Web Form and other content supplied by the master page. You'll need to manually replace this markup with the appropriate Content controls. For that reason, I find it easier to bind the new ASP.NET page to a master page.

> **NOTE**
> Because `Site.master` and `Alternate.master` have the same set of ContentPlaceHolder controls it doesn't matter what master page you choose when creating the new content page. For consistency, I'd suggest using `Site.master`.

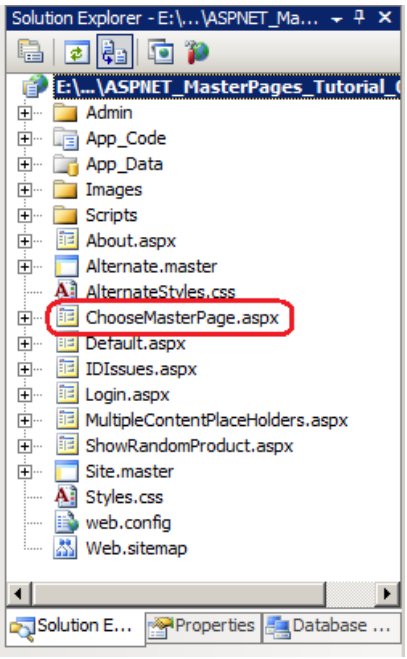


Figure 05: Add a New Content Page to the Website (Click to view full-size image)

Update the `Web.sitemap` file to include an entry for this lesson. Add the following markup beneath the `<siteMapNode>` for the Master Pages and ASP.NET AJAX lesson:

```
<siteMapNode url="~/ChooseMasterPage.aspx" title="Choose a Master Page" />
```

Before adding any content to the `ChooseMasterPage.aspx` page take a moment to update the page's code-behind class so that it derives from `BasePage` (rather than `System.Web.UI.Page`). Next, add a DropDownList control to the page, set its `ID` property to `MasterPageChoice`, and add two ListItems with the `Text` values of "~/Site.master" and "~/Alternate.master".

Add a Button Web control to the page and set its `ID` and `Text` properties to `SaveLayout` and "Save Layout Choice", respectively. At this point your page's declarative markup should look similar to the following:

```
<p>
 Your layout choice:
 <asp:DropDownList ID="MasterPageChoice" runat="server">
 <asp:ListItem>~/Site.master</asp:ListItem>
 <asp:ListItem>~/Alternate.master</asp:ListItem>
 </asp:DropDownList>
</p>
<p>
 <asp:Button ID="SaveLayout" runat="server" Text="Save Layout Choice" />
</p>
```

When the page is first visited we need to display the user's currently selected master page choice. Create a `Page_Load` event handler and add the following code:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
 If Not Page.IsPostBack Then
 If Session("MyMasterPage") IsNot Nothing Then
 Dim li As ListItem = MasterPageChoice.Items.FindByText(Session("MyMasterPage").ToString())
 If li IsNot Nothing Then
 li.Selected = True
 End If
 End If
 End If
End Sub
```

The above code executes only on the first page visit (and not on subsequent postbacks). It first checks to see if the Session variable `MyMasterPage` exists. If it does, it attempts to find the matching ListItem in the `MasterPageChoice` DropDownList. If a matching ListItem is found, its `Selected` property is set to `True`.

We also need code that saves the user's choice into the `MyMasterPage` Session variable. Create an event handler for the `SaveLayout` Button's `Click` event and add the following code:

```
Protected Sub SaveLayout_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles SaveLayout.Click
 Session("MyMasterPage") = MasterPageChoice.SelectedValue
 Response.Redirect("ChooseMasterPage.aspx")
End Sub
```

NOTE

By the time the `Click` event handler executes on postback, the master page has already been selected. Therefore, the user's drop-down list selection won't be in effect until the next page visit. The `Response.Redirect` forces the browser to re-request `ChooseMasterPage.aspx`.

With the `ChooseMasterPage.aspx` page complete, our final task is to have `BasePage` assign the `MasterPageFile` property based on the value of the `MyMasterPage` Session variable. If the Session variable is not set have `BasePage` default to `Site.master`.

```
Protected Overrides Sub OnPreInit(ByVal e As System.EventArgs)
 SetMasterPageFile()
 MyBase.OnPreInit(e)
End Sub
Protected Overridable Sub SetMasterPageFile()
 Me.MasterPageFile = GetMasterPageFileFromSession()
End Sub
Protected Function GetMasterPageFileFromSession() As String
 If Session("MyMasterPage") Is Nothing Then
 Return "~/Site.master"
 Else
 Return Session("MyMasterPage").ToString()
 End If
End Function
```

NOTE

I moved the code that assigns the `Page` object's `MasterPageFile` property out of the `OnPreInit` event handler and into two separate methods. This first method, `SetMasterPageFile`, assigns the `MasterPageFile` property to the value returned by the second method, `GetMasterPageFileFromSession`. I marked the `SetMasterPageFile` method `Overridable` so that future classes that extend `BasePage` can optionally override it to implement custom logic, if needed. We'll see an example of overriding `BasePage`'s `SetMasterPageFile` property in the next tutorial.

With this code in place, visit the `ChooseMasterPage.aspx` page. Initially, the `Site.master` master page is selected (see Figure 6), but the user can pick a different master page from the drop-down list.

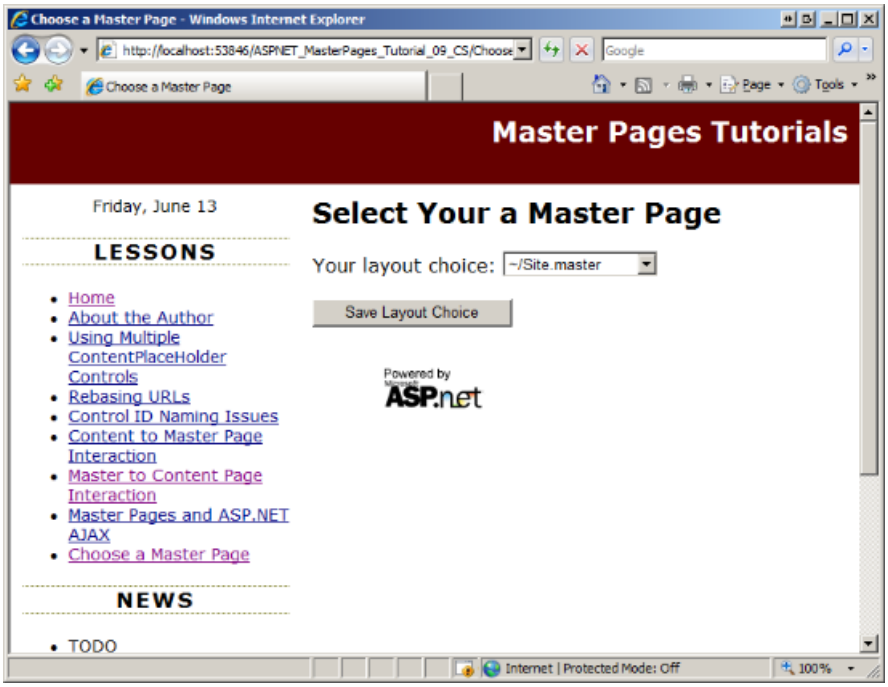


Figure 06: Content Pages are Displayed Using the `Site.master` Master Page (Click to view full-size image)

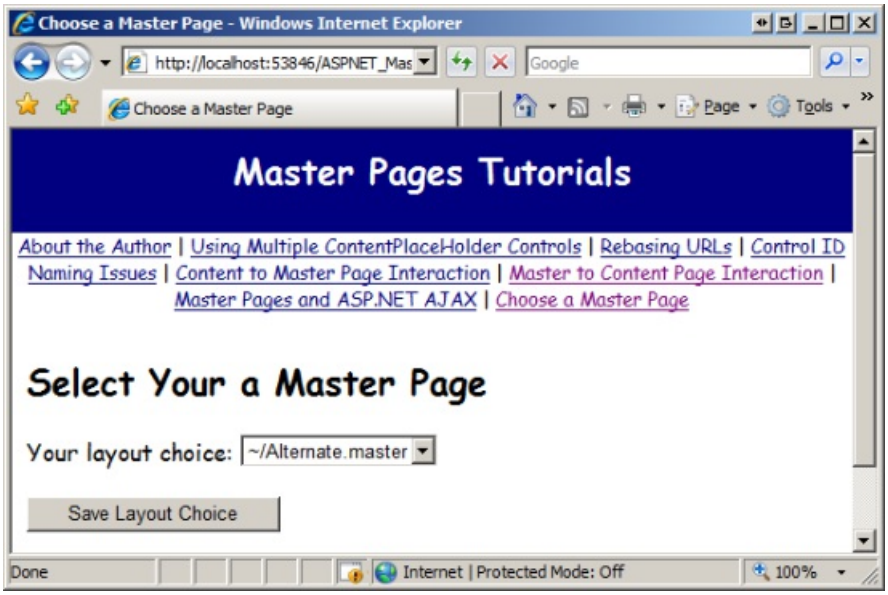


Figure 07: Content Pages are Now Displayed Using the `Alternate.master` Master Page (Click to view full-size image)

Summary

When a content page is visited, its Content controls are fused with its master page's ContentPlaceHolder controls. The content page's master page is denoted by the `Page` class's `MasterPageFile` property, which is assigned to the `@Page` directive's `MasterPageFile` attribute during the Initialization stage. As this tutorial showed, we can assign a value to the `MasterPageFile` property as long as we do so before the end of the PreInit stage. Being able to programmatically specify the master page opens the door for more advanced scenarios, such as dynamically binding a content page to a master page based on external factors.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Page Lifecycle Diagram
- ASP.NET Page Lifecycle Overview
- ASP.NET Themes and Skins Overview
- Master Pages: Tips, Tricks, and Traps
- Themes in ASP.NET

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Suchi Banerjee. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Nested Master Pages (VB)

2/20/2020 • 26 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Shows how to nest one master page within another.

Introduction

Over the course of the past nine tutorials we have seen how to implement a site-wide layout with master pages. In a nutshell, master pages allow us, the page developer, to define common markup in the master page along with specific regions that can be customized on a content page-by-content page basis. The ContentPlaceHolder controls in a master page indicate the customizable regions; the customized markup for the ContentPlaceHolder controls are defined in the content page via Content controls.

The master page techniques we've explored thus far are great if you have a single layout used across the entire site. However, many large websites have a site layout that is customized across various sections. For example, consider a health care application used by hospital staff to manage patient information, activities, and billing. There may be three types of web pages in this application:

- Staff member-specific pages where staff members can update availability, view schedules, or request vacation time.
- Patient-specific pages where staff members view or edit information for a specific patient.
- Billing-specific pages where accountants review current claim statuses and financial reports.

Every page might share a common layout, such as a menu across the top and a series of frequently used links along the bottom. But the staff-, patient-, and billing-specific pages may need to customize this generic layout. For example, perhaps all staff-specific pages should include a calendar and task list showing the currently logged on user's availability and daily schedule. Perhaps all patient-specific pages need to show the name, address, and insurance information for the patient whose information is being edited.

It's possible to create such customized layouts by using *nested master pages*. To implement the above scenario, we would start by creating a master page that defined the site-wide layout, the menu and footer content, with ContentPlaceHolders defining the customizable regions. We would then create three nested master pages, one for each type of web page. Each nested master page would define the content among the type of content pages that use the master page. In other words, the nested master page for patient-specific content pages would include markup and programmatic logic for displaying information about the patient being edited. When creating a new patient-specific page we would bind it to this nested master page.

This tutorial starts by highlighting the benefits of nested master pages. It then shows how to create and use nested master pages.

> **NOTE**
>
> Nested master pages have been possible since version 2.0 of the .NET Framework. However, Visual Studio 2005 did not include design-time support for nested master pages. The good news is that Visual Studio 2008 offers a rich design-time experience for nested master pages. If you are interested in using nested master pages but are still using Visual Studio 2005, check out Scott Guthrie's blog entry, Tips for Nested Master Pages in VS 2005 Design-Time.

The Benefits of Nested Master Pages

Many websites have an overarching site design as well as more customized designs specific to certain types of pages. For instance, in our demo web application we have created a rudimentary Administration section (the pages in the `~/Admin` folder). Currently the web pages in the `~/Admin` folder use the same master page as those pages not in the administration section (namely, `Site.master` or `Alternate.master`, depending on the user's selection).

> **NOTE**
> For now, pretend that our site has just one master page, `Site.master`. We'll address using nested master pages with two (or more) master pages starting with "Using a Nested Master Page for the Administration Section" later in this tutorial.

Imagine that we were asked to customize the layout of the Administration pages to include additional information or links that would not otherwise be present in other pages in the site. There are four techniques to implement this requirement:

1. Manually add the Administration-specific information and links to every content page in the `~/Admin` folder.
2. Update the `Site.master` master page to include the Administration section-specific information and links, and then add code to the master page to show or hide these sections based on whether one of the Administration pages is being visited.
3. Create a new master page specifically for the Administration section, copy over the markup from `Site.master`, add the Administration section-specific information and links, and then update the content pages in the `~/Admin` folder to use this new master page.
4. Create a nested master page that binds to `Site.master` and have the content pages in the `~/Admin` folder use this new nested master page. This nested master page would include just the additional information and links specific to the Administration pages and would not need to repeat the markup already defined in `Site.master`.

The first option is the least palatable. The whole point of using master pages is to move away from having to manually copy and paste common markup to new ASP.NET pages. The second option is acceptable, but makes the application less maintainable as it bulks up the master pages with markup that is only occasionally displayed and requires developers editing the master page to work around this markup and to have to remember when, exactly, certain markup is displayed versus when it is hidden. This approach would be less tenable as customizations from more and more types of web pages needed to be accommodated by this single master page.

The third option removes the clutter and complexity issues the surfaced with the second option. However, option three's main drawback is that it requires us to copy and paste the common layout from `Site.master` to the new Administration section-specific master page. If we later decide to change the site-wide layout we have to remember to change it in two places.

The fourth option, nested master pages, give us the best of the second and third options. The site-wide layout information is maintained in one file - the top-level master page - while the content specific to particular regions is separated out into different files.

This tutorial starts with a look at creating and using a simple nested master page. We create a brand new top-level master page, two nested master pages, and two content pages. Starting with "Using a Nested Master Page for the Administration Section," we look at updating our existing master page architecture to include the use of nested master pages. Specifically, we create a nested master page and use it to include additional custom content for the content pages in the `~/Admin` folder.

Step 1: Creating a Simple Top-Level Master Page

Creating a nested master based on one of the existing master pages and then updating an existing content page to use this new nested master page instead of the top-level master page entails some complexity because the

existing content pages already expect certain ContentPlaceHolder controls defined in the top-level master page. Therefore, the nested master page must also include the same ContentPlaceHolder controls with the same names. Furthermore, our particular demo application has two master pages (`Site.master` and `Alternate.master`) that are dynamically assigned to a content page based on a user's preferences, which further adds to this complexity. We will look at updating the existing application to use nested master pages later in this tutorial, but let's first focus on a simple nested master pages example.

Create a new folder named `NestedMasterPages` and then add a new master page file to that folder named `Simple.master`. (See Figure 1 for a screen shot of the Solution Explorer after this folder and file have been added.) Drag the `AlternateStyles.css` style sheet file from the Solution Explorer onto the Designer. This adds a `<link>` element to the style sheet file in the `<head>` element, after which the master page's `<head>` element's markup should look like:

```
<head runat="server">
 <title>Untitled Page</title>
 <asp:ContentPlaceHolder id="head" runat="server">
 </asp:ContentPlaceHolder>
 <link href="../AlternateStyles.css" rel="stylesheet" type="text/css" />
</head>
```

Next, add the following markup within the Web Form of `Simple.master`:

```
<div id="topContent">
 <asp:HyperLink ID="lnkHome" runat="server"
 NavigateUrl="~/NestedMasterPages/Default.aspx"
 Text="Nested Master Pages Tutorial (Simple)" />
</div>
<div id="mainContent">
 <asp:ContentPlaceHolder id="MainContent" runat="server">
 </asp:ContentPlaceHolder>
</div>
```

This markup displays a link titled "Nested Master Pages (Simple)" at the top of the page in a large white font on a navy background. Beneath that is the `MainContent` ContentPlaceHolder. Figure 1 shows the `Simple.master` master page when loaded in the Visual Studio Designer.

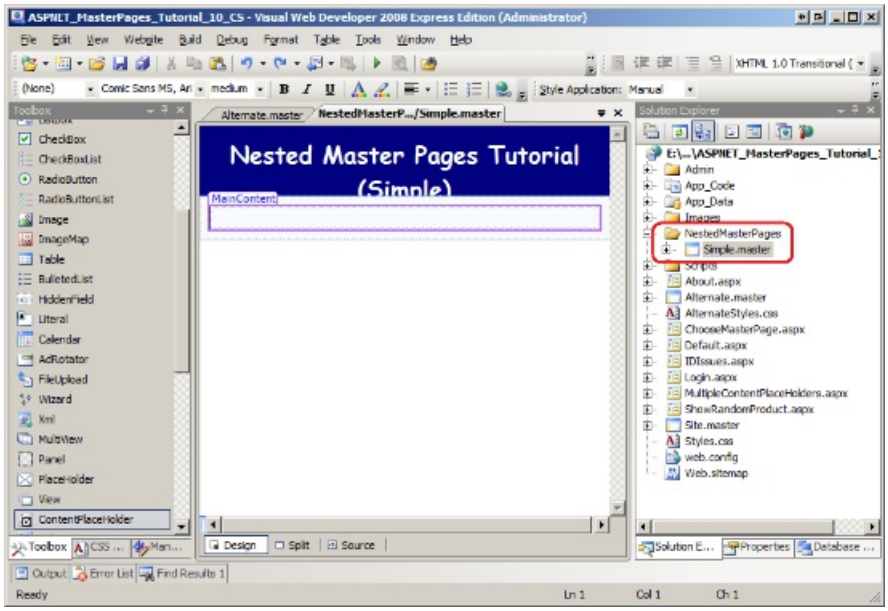


Figure 01: The Nested Master Page Defines Content Specific to the Pages in the Administration Section (Click to view full-size image)

Step 2: Creating a Simple Nested Master Page

`Simple.master` contains two ContentPlaceHolder controls: the `MainContent` ContentPlaceHolder we added within the Web Form along with the `head` ContentPlaceHolder in the `<head>` element. If we were to create a content page and bind it to `Simple.master` the content page would have two Content controls referencing the two ContentPlaceHolders. Similarly, if we create a nested master page and bind it to `Simple.master` then the nested master page will have two Content controls.

Let's add a new nested master page to the `NestedMasterPages` folder named `SimpleNested.master`. Right-click on the `NestedMasterPages` folder and choose Add New Item. This brings up the Add New Item dialog box shown in Figure 2. Select the Master Page template type and type in the name of the new master page. To indicate that the new master page should be a nested master page, check the "Select master page" checkbox.

Next, click the Add button. This will display the same Select a Master Page dialog box you see when binding a content page to a master page (see Figure 3). Choose the `Simple.master` master page in the `NestedMasterPages` folder and click OK.

> **NOTE**
>
> If you created your ASP.NET website using the Web Application Project model instead of the Web Site Project model you will not see the "Select master page" checkbox in the Add New Item dialog box shown in Figure 2. To create a nested master page when using the Web Application Project model you must choose the Nested Master Page template (instead of the Master Page template). After selecting the Nested Master Page template and clicking Add, the same Select a Master Page dialog box shown in Figure 3 will appear.

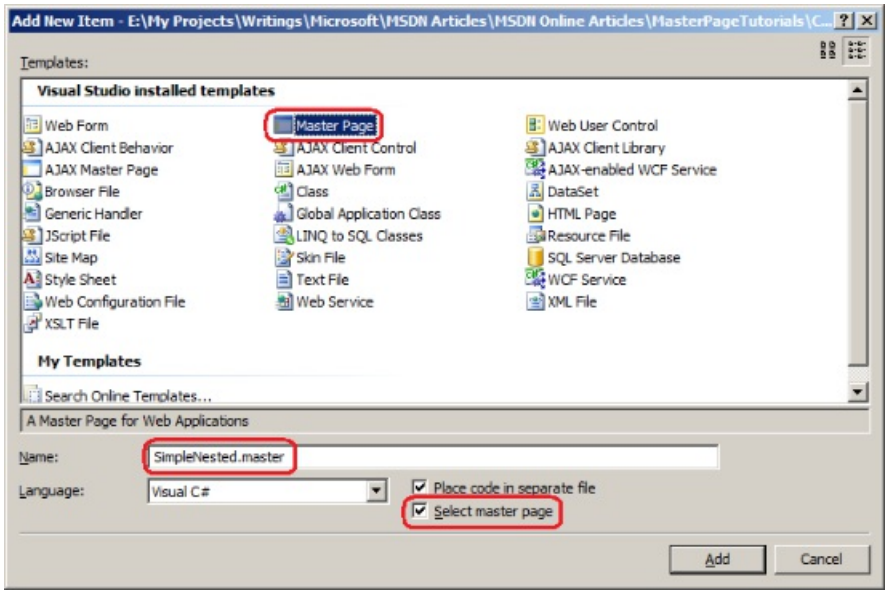


Figure 02: Check the "Select master page" Checkbox to Add a Nested Master Page (Click to view full-size image)

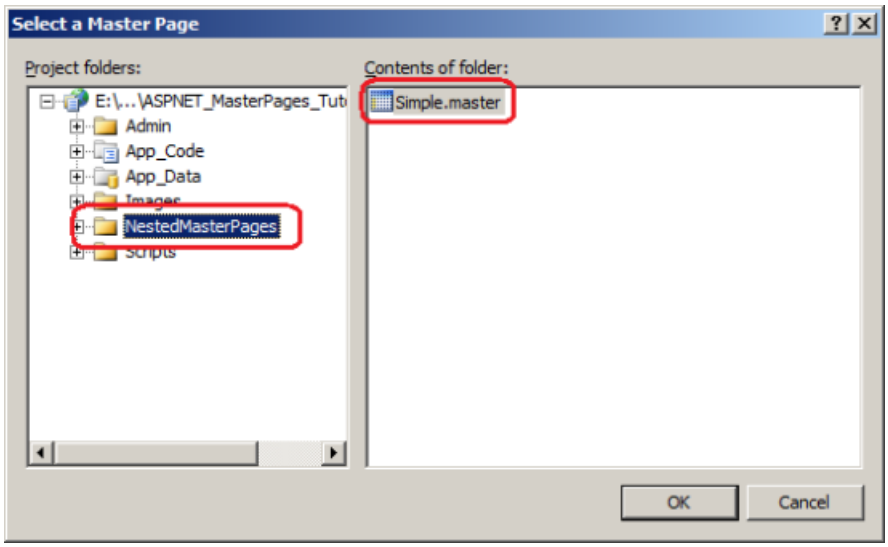


Figure 03: Bind the Nested Master Page to the `Simple.master` Master Page (Click to view full-size image)

The nested master page's declarative markup, shown below, contains two Content controls referencing the top-level master page's two ContentPlaceHolder controls.

```
<%@ Master Language="VB" MasterPageFile="~/NestedMasterPages/Simple.master" AutoEventWireup="false"
CodeFile="SimpleNested.master.vb" Inherits="NestedMasterPages_SimpleNested" %>
 <asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
 </asp:Content>
 <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
 </asp:Content>
```

Except for the `<%@ Master %>` directive, the nested master page's initial declarative markup is identical to the markup that is initially generated when binding a content page to the same top-level master page. Like a content page's `<%@ Page %>` directive, the `<%@ Master %>` directive here includes a `MasterPageFile` attribute that specifies the nested master page's parent master page. The main difference between the nested master page and a content page bound to the same top-level master page is that the nested master page can include ContentPlaceHolder controls. The nested master page's ContentPlaceHolder controls define the regions where the content pages can customize the markup.

Update this nested master page so that it displays the text "Hello, from SimpleNested!" in the Content control that corresponds to the `MainContent` ContentPlaceHolder control.

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
 <p>Hello, from SimpleNested!</p>
</asp:Content>
```

After making this addition, save the nested master page and then add a new content page to the `NestedMasterPages` folder named `Default.aspx`, and bind it to the `SimpleNested.master` master page. Upon adding this page you may be surprised to see that it contains no Content controls (see Figure 4)! A content page can only access its *parent* master page's ContentPlaceHolders. `SimpleNested.master` does not contain any ContentPlaceHolder controls; therefore, any content page bound to this master page cannot contain any Content controls.

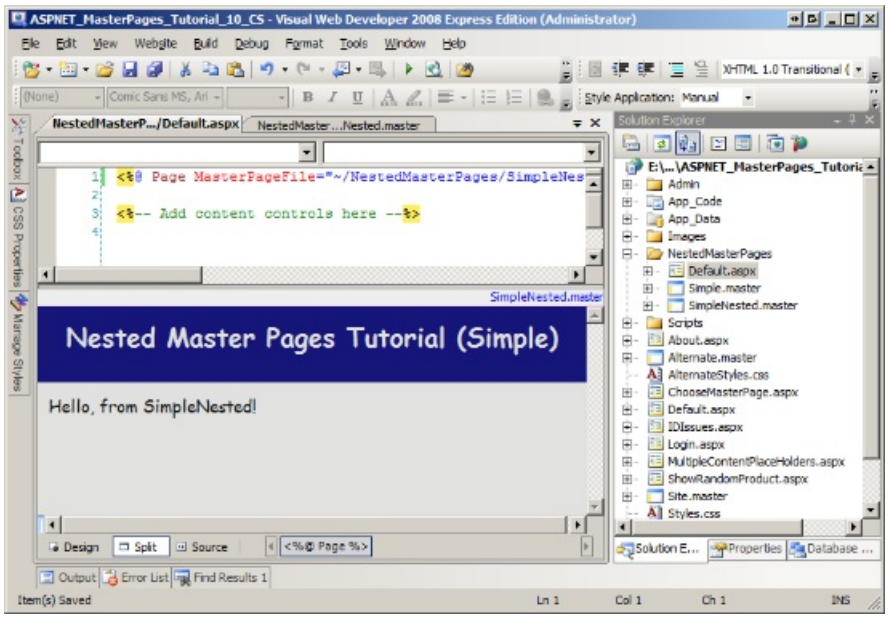


Figure 04: The New Content Page Contains No Content Controls (Click to view full-size image)

What we need to do is update the nested master page (`SimpleNested.master`) to include ContentPlaceHolder controls. Typically you'll want your nested master pages to include a ContentPlaceHolder for each ContentPlaceHolder defined by its parent master page, thereby allowing its child master page or content page to work with any of the top-level master page's ContentPlaceHolder controls.

Update the `SimpleNested.master` master page to include a ContentPlaceHolder in its two Content controls. Give the ContentPlaceHolder controls the same name as the ContentPlaceHolder control their Content control refers to. That is, add a ContentPlaceHolder control named `MainContent` to the Content control in `SimpleNested.master` that references the `MainContent` ContentPlaceHolder in `Simple.master`. Do the same thing in the Content control that references the `head` ContentPlaceHolder.

> **NOTE**
>
> While I recommend naming the ContentPlaceHolder controls in the nested master page the same as the ContentPlaceHolders in the top-level master page, this naming symmetry is not required. You can give the ContentPlaceHolder controls in your nested master page any name you like. However, I find it easier to remember what ContentPlaceHolders correspond with what regions of the page if my top-level master page and nested master pages use the same names.

After making these additions your `SimpleNested.master` master page's declarative markup should look similar to the following:

```
<%@ Master Language="VB" MasterPageFile="~/NestedMasterPages/Simple.master" AutoEventWireup="false"
CodeFile="SimpleNested.master.vb" Inherits="NestedMasterPages_SimpleNested" %>
 <asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
 <asp:ContentPlaceHolder ID="head" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
 <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
 <p>Hello, from SimpleNested!</p>
 <asp:ContentPlaceHolder ID="MainContent" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
```

Delete the `Default.aspx` content page we just created and then re-add it, binding it to the `SimpleNested.master` master page. This time Visual Studio adds two Content controls to the `Default.aspx`, referencing the ContentPlaceHolders now defined in `SimpleNested.master` (see Figure 6). Add the text, "Hello, from Default.aspx!"

in the Content control that referenced `MainContent`.

Figure 5 shows the three entities involved here - `Simple.master`, `SimpleNested.master`, and `Default.aspx` - and how they relate to one another. As the diagram shows, the nested master page implements Content controls for its parent's ContentPlaceHolder. If these regions need to be accessible to the content page, the nested master page must add its own ContentPlaceHolders to the Content controls.

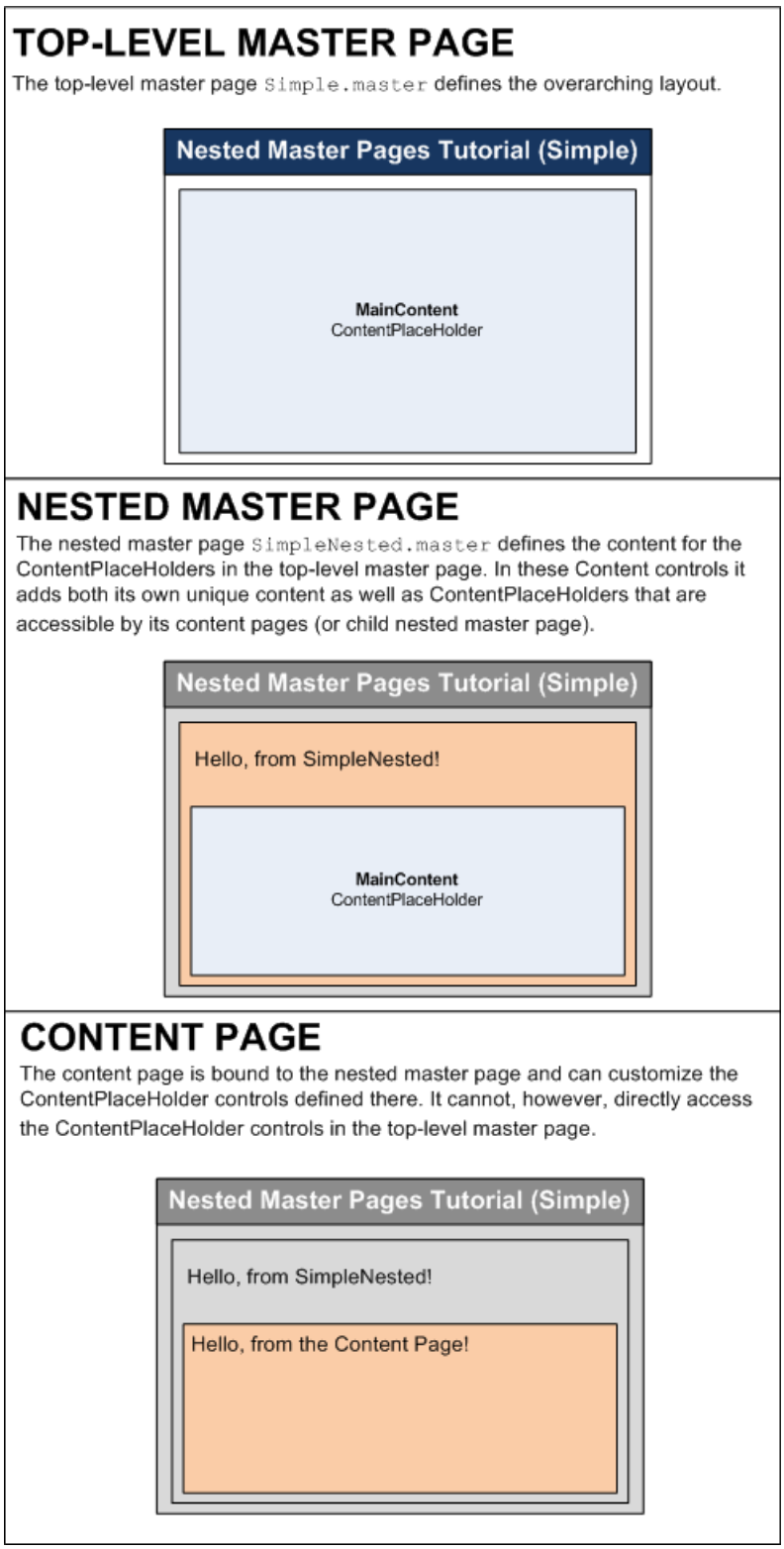


Figure 05: The Top-Level and Nested Master Pages Dictate the Content Page's Layout (Click to view full-size image)

This behavior illustrates how a content page or master page is only cognizant of its parent master page. This behavior is also indicated by the Visual Studio Designer. Figure 6 shows the Designer for `Default.aspx`. While the Designer clearly shows what regions are editable from the content page and what portions aren't, it doesn't disambiguate what non-editable regions are from the nested master page and what regions are from the top-

level master page.

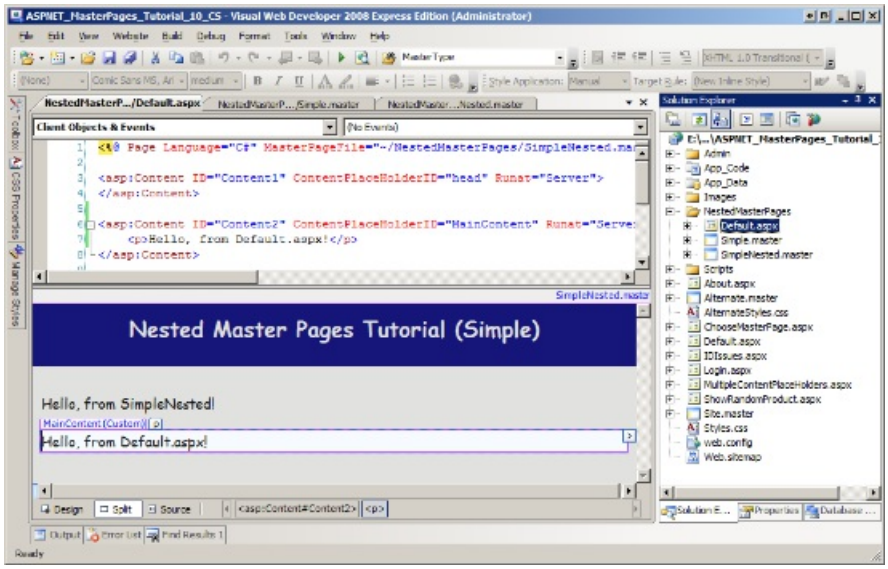


Figure 06: The Content Page Now Includes Content Controls for the Nested Master Page's ContentPlaceHolders (Click to view full-size image)

Step 3: Adding a Second Simple Nested Master Page

The benefit of nested master pages is more evident when there are multiple nested master pages. To illustrate this benefit, create another nested master page in the `NestedMasterPages` folder; name this new nested master page `SimpleNestedAlternate.master` and bind it to the `Simple.master` master page. Add ContentPlaceHolder controls in the nested master page's two Content controls like we did in Step 2. Also add the text, "Hello, from SimpleNestedAlternate!" in the Content control that corresponds to the top-level master page's `MainContent` ContentPlaceHolder. After making these changes your new nested master page's declarative markup should look similar to the following:

```
<%@ Master Language="VB" MasterPageFile="~/NestedMasterPages/Simple.master" AutoEventWireup="false"
CodeFile="SimpleNestedAlternate.master.vb" Inherits="NestedMasterPages_SimpleNestedAlternate" %>
 <asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
 <asp:ContentPlaceHolder ID="head" runat="server">
 </asp:ContentPlaceHolder> </asp:Content>
 <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
 <p>Hello, from SimpleNestedAlternate!</p>
 <asp:ContentPlaceHolder ID="MainContent" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
```

Create a content page named `Alternate.aspx` in the `NestedMasterPages` folder and bind it to the `SimpleNestedAlternate.master` nested master page. Add the text, "Hello, from Alternate!" in the Content control that corresponds to `MainContent`. Figure 7 shows `Alternate.aspx` when viewed through the Visual Studio Designer.

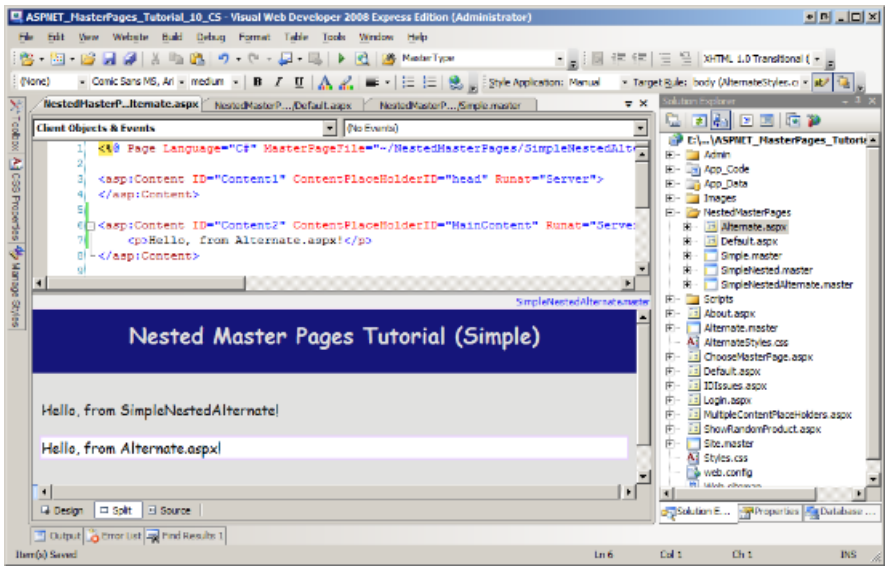


Figure 07: `Alternate.aspx` is Bound to the `SimpleNestedAlternate.master` Master Page (Click to view full-size image)

Compare the Designer in Figure 7 to the Designer in Figure 6. Both content pages share the same layout defined in the top-level master page (`Simple.master`), namely the "Nested Master Pages Tutorial (Simple)" title. Yet both have distinct content defined in their parent master pages - the text "Hello, from SimpleNested!" in Figure 6 and "Hello, from SimpleNestedAlternate!" in Figure 7. Granted, these differences here are trivial, but you could extend this example to include more meaningful differences. For instance, the `SimpleNested.master` page might include a menu with options specific to its content pages, whereas `SimpleNestedAlternate.master` might have information pertinent to the content pages that bind to it.

Now, imagine that we needed to make a change to the overarching site layout. For example, imagine that we wanted to add a list of common links to all content pages. To accomplish this we update the top-level master page, `Simple.master`. Any changes there are immediately reflected in its nested master pages and, by extension, their content pages.

To demonstrate the ease with which we can change the overarching site layout, open the `Simple.master` master page and add the following markup between the `topContent` and `mainContent` `<div>` elements:

```
<div id="navContent">
 <asp:HyperLink ID="lnkDefault" runat="server"
 NavigateUrl="~/NestedMasterPages/Default.aspx"
 Text="Nested Master Page Example 1" />
 |
 <asp:HyperLink ID="lnkAlternate" runat="server"
 NavigateUrl="~/NestedMasterPages/Alternate.aspx"
 Text="Nested Master Page Example 2" />
</div>
```

This adds two links to the top of every page that binds to `Simple.master`, `SimpleNested.master`, or `SimpleNestedAlternate.master`; these changes apply to all nested master pages and their content pages immediately. Figure 8 shows `Alternate.aspx` when viewed through a browser. Note the addition of the links at the top of the page (compared to Figure 7).

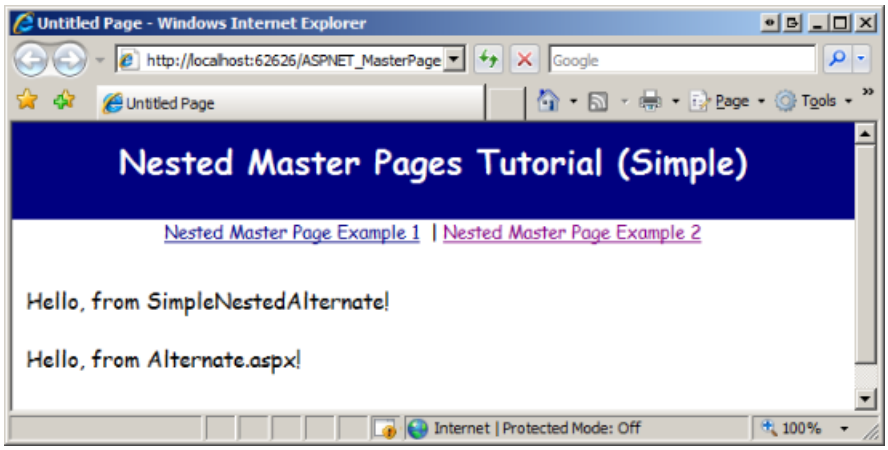


Figure 08: Changed to the Top-Level Master Page are Immediately Reflected in its Nested Master Pages and Their Content Pages (Click to view full-size image)

Using a Nested Master Page for the Administration Section

At this point we have looked at the advantages of nested master pages and have seen how to create and use them in an ASP.NET application. The examples in Steps 1, 2, and 3, however, involved creating a new top-level master page, new nested master pages, and new content pages. What about adding a new nested master page to a website with an existing top-level master page and content pages?

Integrating a nested master page into an existing website and associating it with existing content pages requires a bit more effort than starting from scratch. Steps 4, 5, 6, and 7 explore these challenges as we augment our demo application to include a new nested master page named `AdminNested.master` that contains instructions for the administrator and is used by the ASP.NET pages in the `~/Admin` folder.

Integrating a nested master page into our demo application introduces the following hurdles:

- The existing content pages in the `~/Admin` folder have certain expectations from their master page. For starters, they expect certain ContentPlaceHolder controls to be present. Furthermore, the `~/Admin/AddProduct.aspx` and `~/Admin/Products.aspx` pages call the master page's public `RefreshRecentProductsGrid` method, set its `GridMessageText` property, or have an event handler for its `PricesDoubled` event. Consequently, our nested master page must provide the same ContentPlaceHolders and public members.
- In the preceding tutorial we enhanced the `BasePage` class to dynamically set the `Page` object's `MasterPageFile` property based on a Session variable. How to we support dynamic master pages when using nested master pages?

These two challenges will surface as we build the nested master page and use it from our existing content pages. We'll investigate and surmount these issues as they arise.

Step 4: Creating the Nested Master Page

Our first task is to create the nested master page to be used by the pages in the Administration section. As we saw in Step 2, when adding a new nested master page we need to specify the nested master page's parent master page. But we have two top-level master pages: `Site.master` and `Alternate.master`. Recall that we created `Alternate.master` in the preceding tutorial and wrote code in the `BasePage` class that set the `Page` object's `MasterPageFile` property at runtime to either `Site.master` or `Alternate.master` depending on the value of the `MyMasterPage` Session variable.

How do we configure our nested master page so that it uses the appropriate top-level master page? We have two options:

- Create two nested master pages, `AdminNestedSite.master` and `AdminNestedAlternate.master`, and bind them to the top-level master pages `Site.master` and `Alternate.master`, respectively. In `BasePage`, then, we'd set the `Page` object's `MasterPageFile` to the appropriate nested master page.
- Create a single nested master page and have the content pages use this particular master page. Then, at runtime, we'd need to set the nested master page's `MasterPageFile` property to the appropriate top-level master page at runtime. (As you might have figured out by now, master pages also have a `MasterPageFile` property.)

Let's use the second option. Create a single nested master page file in the `~/Admin` folder named `AdminNested.master`. Because both `Site.master` and `Alternate.master` have the same set of ContentPlaceHolder controls, it doesn't matter what master page you bind it to, although I encourage you to bind it to `Site.master` for consistency's sake.

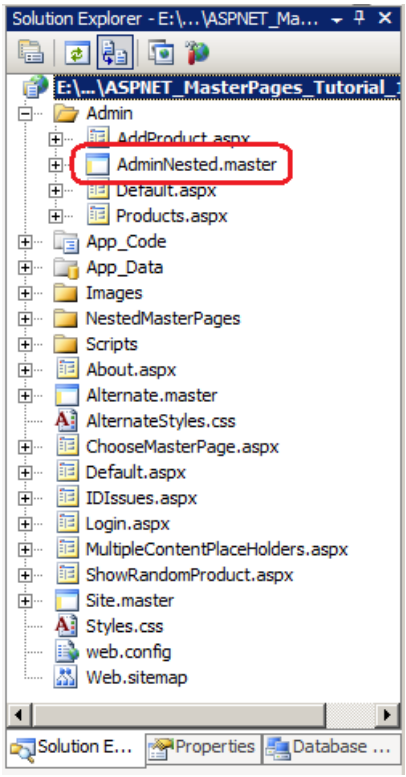


Figure 09: Add a Nested Master Page to the `~/Admin` Folder. (Click to view full-size image)

Because the nested master page is bound to a master page with four ContentPlaceHolder controls, Visual Studio adds four Content controls to the new nested master page file's initial markup. Like we did in Steps 2 and 3, add a ContentPlaceHolder control in each Content control, giving it the same name as the top-level master page's ContentPlaceHolder control. Also add the following markup to the Content control that corresponds to the `MainContent` ContentPlaceHolder:

```
<div class="instructions">
 <b>Administration Instructions:</b>
 <br />
 The pages in the Administration section allow you, the Administrator, to
 add new products and view existing products.
</div>
```

Next, define the `instructions` CSS class in the `Styles.css` and `AlternateStyles.css` CSS files. The following CSS rules cause HTML elements styled with the `instructions` class to be displayed with a light yellow background color and a black, solid border:

```
.instructions
{
 padding: 6px;
 border: dashed 1px black;
 background-color: #ffb;
 margin-bottom: 10px;
}
```

Because this markup has been added to the nested master page, it will only appear in those pages that use this nested master page (namely, the pages in the Administration section).

After making these additions to your nested master page, its declarative markup should look similar to the following:

```
<%@ Master Language="VB" MasterPageFile="~/Site.master" AutoEventWireup="false"
CodeFile="AdminNested.master.vb" Inherits="Admin_AdminNested" %>
 <asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
 <asp:ContentPlaceHolder ID="head" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
 <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" Runat="Server">
 <div class="instructions">
 <b>Administration Instructions:</b>
 <br />
 The pages in the Administration section allow you, the Administrator, to
 add new products and view existing products.
 </div>
 <asp:ContentPlaceHolder ID="MainContent" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
 <asp:Content ID="Content3" ContentPlaceHolderID="QuickLoginUI" Runat="Server">
 <asp:ContentPlaceHolder ID="QuickLoginUI" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
 <asp:Content ID="Content4" ContentPlaceHolderID="LeftColumnContent" Runat="Server">
 <asp:ContentPlaceHolder ID="LeftColumnContent" runat="server">
 </asp:ContentPlaceHolder>
 </asp:Content>
```

Note that each Content control has a ContentPlaceHolder control and that the ContentPlaceHolder controls' `ID` properties are assigned the same values as the corresponding ContentPlaceHolder controls in the top-level master page. Moreover, the Administration section-specific markup appears in the `MainContent` ContentPlaceHolder.

Figure 10 shows the `AdminNested.master` nested master page when viewed through Visual Studio's Designer. You can see the instructions in the yellow box at the top of the `MainContent` Content control.

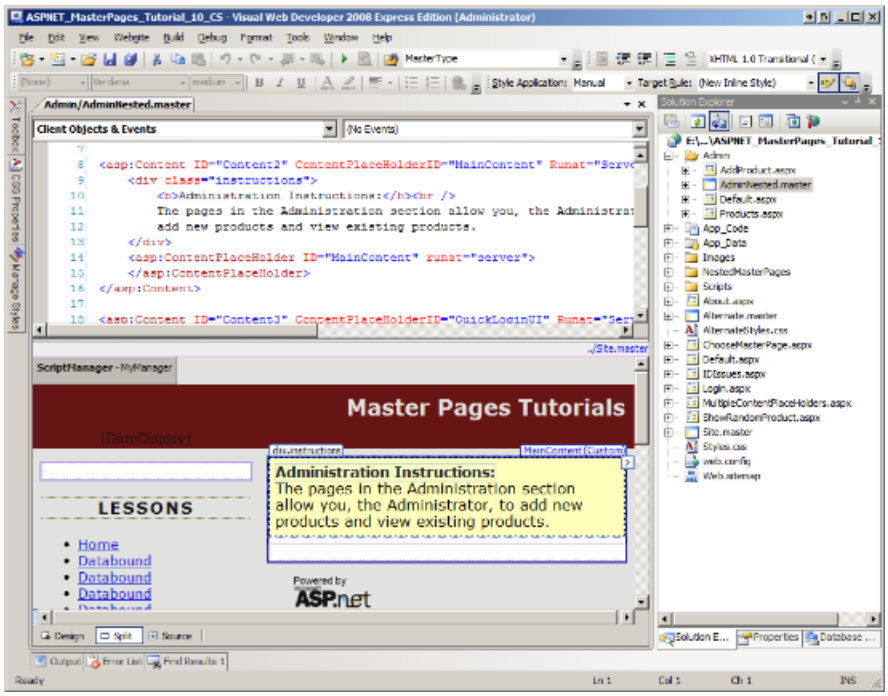


Figure 10: The Nested Master Page Extends the Top-Level Master Page to Include Instructions for the Administrator. (Click to view full-size image)

Step 5: Updating the Existing Content Pages to Use the New Nested Master Page

Anytime we add a new content page to the Administration section we need to bind it to the `AdminNested.master` master page we just created. But what about the existing content pages? Currently, all content pages in the site derive from the `BasePage` class, which programmatically sets the content page's master page at runtime. This is not the behavior we want for the content pages in the Administration section. Instead, we want these content pages to always use the `AdminNested.master` page. It will be the responsibility of the nested master page to choose the right top-level content page at runtime.

To best way to achieve this desired behavior is to create a new custom base page class named `AdminBasePage` that extends the `BasePage` class. `AdminBasePage` can then override the `SetMasterPageFile` and set the `Page` object's `MasterPageFile` to the hard-coded value "~/Admin/AdminNested.master". In this way, any page that derives from `AdminBasePage` will use `AdminNested.master`, whereas any page that derives from `BasePage` will have its `MasterPageFile` property set dynamically to either "~/Site.master" or "~/Alternate.master" based on the value of the `MyMasterPage` Session variable.

Start by adding a new class file to the `App_Code` folder named `AdminBasePage.vb`. Have `AdminBasePage` extend `BasePage` and then override the `SetMasterPageFile` method. In that method assign the `MasterPageFile` the value "~/Admin/AdminNested.master". After making these changes your class file should look similar to the following:

```
Public Class AdminBasePage
 Inherits BasePage
 Protected Overrides Sub SetMasterPageFile()
 Me.MasterPageFile = "~/Admin/AdminNested.master"
 End Sub
End Class
```

We now need to have the existing content pages in the Administration section derive from `AdminBasePage` instead of `BasePage`. Go to the code-behind class file for each content page in the `~/Admin` folder and make this change. For example, in `~/Admin/Default.aspx` you'd change the code-behind class declaration from:

```
Partial Class Admin_Default
 Inherits BasePage
```

To:

```
Partial Class Admin_Default
 Inherits AdminBasePage
```

Figure 11 depicts how the top-level master page (`Site.master` or `Alternate.master`), the nested master page (`AdminNested.master`), and the Administration section content pages relate to one another.

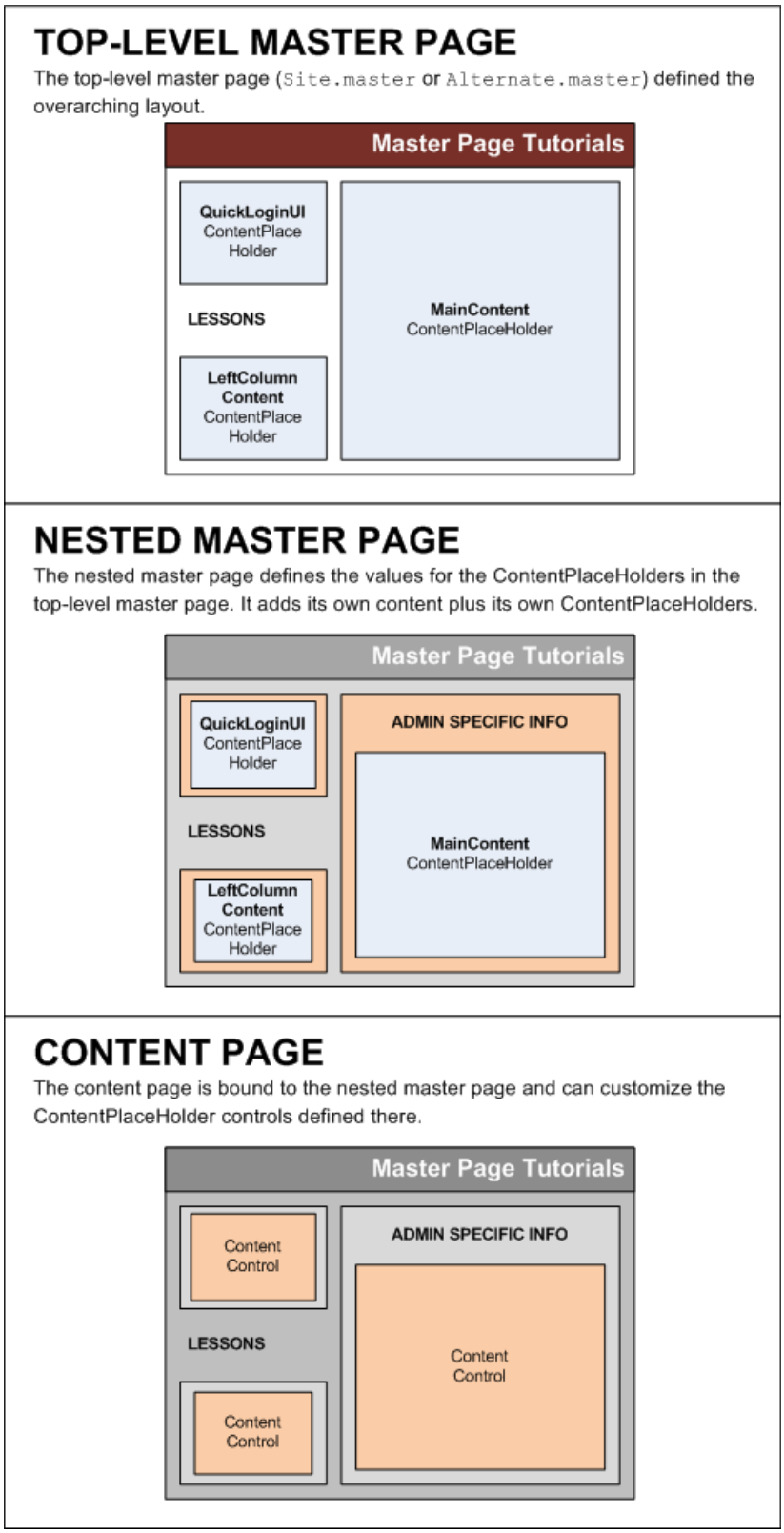


Figure 11: The Nested Master Page Defines Content Specific to the Pages in the Administration Section (Click to view full-size image)

Step 6: Mirroring the Master Page's Public Methods and Properties

Recall that the `~/Admin/AddProduct.aspx` and `~/Admin/Products.aspx` pages interact programmatically with the master page: `~/Admin/AddProduct.aspx` calls the master page's public `RefreshRecentProductsGrid` method and sets its `GridMessageText` property; `~/Admin/Products.aspx` has an event handler for the `PricesDoubled` event. In the preceding tutorial we created a `MustInherit` `BaseMasterPage` class that defined these public members.

The `~/Admin/AddProduct.aspx` and `~/Admin/Products.aspx` pages assume that their master page derives from the `BaseMasterPage` class. The `AdminNested.master` page, however, currently extends the `System.Web.UI.MasterPage` class. As a result, when visiting `~/Admin/Products.aspx` an `InvalidCastException` is thrown with the message: "Unable to cast object of type 'ASP.admin_adminnested_master' to type 'BaseMasterPage'."

To fix this we need to have the `AdminNested.master` code-behind class extend `BaseMasterPage`. Update the nested master page's code-behind class declaration from:

```
Partial Class Admin_AdminNested
 Inherits System.Web.UI.MasterPage
```

To:

```
Partial Class Admin_AdminNested
 Inherits BaseMasterPage
```

We're not done yet. We need to override the members marked as `MustOverride`, namely `RefreshRecentProductsGrid` and `GridMessageText`. These members are used by the top-level master pages to update their user interfaces. (Actually, only the `Site.master` master page uses these methods, although both top-level master pages implement these methods, since both extend `BaseMasterPage`.)

While we need to implement these members in `AdminNested.master`, all these implementations need to do is simply call the same member in the top-level master page used by the nested master page. For instance, when a content page in the Administration section calls the nested master page's `RefreshRecentProductsGrid` method, all the nested master page needs to do is, in turn, call `Site.master` or `Alternate.master`'s `RefreshRecentProductsGrid` method.

To achieve this, start by adding the following `@MasterType` directive to the top of `AdminNested.master`:

```
<%@ MasterType TypeName="BaseMasterPage" %>
```

Recall that the `@MasterType` directive adds a strongly-typed property to the code-behind class named `Master`. Then override the `RefreshRecentProductsGrid` and `GridMessageText` members and simply delegate the call to the `Master`'s corresponding method:

```
Partial Class Admin_AdminNested
 Inherits BaseMasterPage
 Public Overrides Property GridMessageText() As String
 Get
 Return Master.GridMessageText
 End Get
 Set(ByVal value As String)
 Master.GridMessageText = value
 End Set
 End Property
 Public Overrides Sub RefreshRecentProductsGrid()
 Master.RefreshRecentProductsGrid()
 End Sub
End Class
```

With this code in place, you should be able to visit and use the content pages in the Administration section. Figure 12 shows the `~/Admin/Products.aspx` page when viewed through a browser. As you can see, the page includes the Administration Instructions box, which is defined in the nested master page.

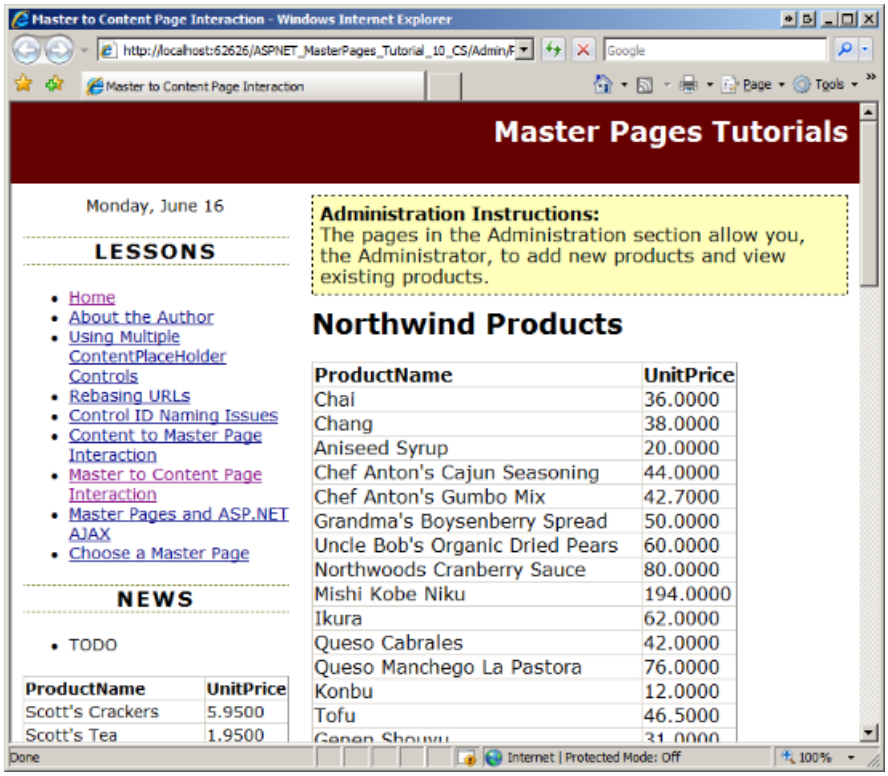


Figure 12: The Content Pages in the Administration Section Include Instructions at the Top of Each Page (Click to view full-size image)

Step 7: Using the Appropriate Top-Level Master Page at Runtime

While all the content pages in the Administration section are fully functional, they all use the same top-level master page and ignore the master page selected by the user at `ChooseMasterPage.aspx`. This behavior is due to the fact that the nested master page has its `MasterPageFile` property statically set to `Site.master` in its `<%@ Master %>` directive.

To use the top-level master page selected by the end user we need to set the `AdminNested.master`'s `MasterPageFile` property to the value in the `MyMasterPage` Session variable. Because we set the content pages' `MasterPageFile` properties in `BasePage`, you may think that we would set the nested master page's `MasterPageFile` property in `BaseMasterPage` or in the `AdminNested.master`'s code-behind class. This won't work, however, because we need to have set the `MasterPageFile` property by the end of the PreInit stage. The earliest time that we can programmatically tap into the page lifecycle from a master page is the Init stage (which occurs

after the PreInit stage).

Therefore, we need to set the nested master page's `MasterPageFile` property from the content pages. The only content pages that use the `AdminNested.master` master page derive from `AdminBasePage`. Therefore, we can put this logic there. In Step 5 we overrode the `SetMasterPageFile` method, setting the Page object's `MasterPageFile` property to "~/Admin/AdminNested.master". Update `SetMasterPageFile` to also set the master page's `MasterPageFile` property to the result stored in Session:

```
Public Class AdminBasePage
 Inherits BasePage
 Protected Overrides Sub SetMasterPageFile()
 Me.MasterPageFile = "~/Admin/AdminNested.master"
 Page.Master.MasterPageFile = MyBase.GetMasterPageFileFromSession()
 End Sub
End Class
```

The `GetMasterPageFileFromSession` method, which we added to the `BasePage` class in the preceding tutorial, returns the appropriate master page file path based on the Session variable value.

With this change in place, the user's master page selection carries over to the Administration section. Figure 13 shows the same page as Figure 12, but after the user has changed their master page selection to `Alternate.master`.

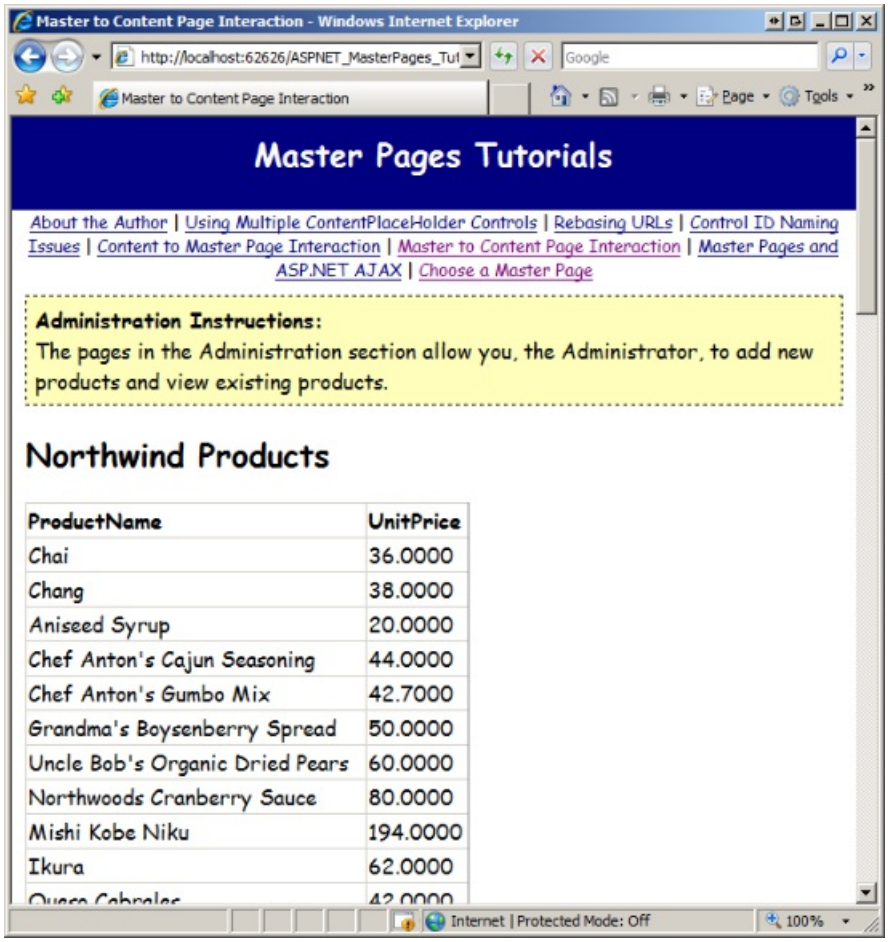


Figure 13: The Nested Administration Page Uses the Top-Level Master Page Selected by the User (Click to view full-size image)

Summary

Much like how content pages can bind to a master page, it is possible to create nested master pages by having a child master page bind to a parent master page. The child master page may define Content controls for each of its

parent's ContentPlaceHolders; it can then add its own ContentPlaceHolder controls (as well as other markup) to these Content controls. Nested master pages are quite useful in large web applications where all pages share an overarching look and feel, yet certain sections of the site require unique customizations.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Nested ASP.NET Master Pages
- Tips for Nested Master Pages and VS 2005 Design-Time
- VS 2008 Nested Master Page Support

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 3.5 in 24 Hours*. Scott can be reached at mitchell@4GuysFromRolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS

Deploying Web Site Projects in Visual Studio 2008 and earlier

2/20/2020 • 2 minutes to read • Edit Online

ASP.NET web applications are typically designed, created, and tested in a local development environment and need to be deployed to a production environment once it is ready for release. This tutorial series details the deployment process.

- ASP.NET Hosting Options (C#)
- Determining What Files Need to Be Deployed (C#)
- Deploying Your Site Using an FTP Client (C#)
- Deploying Your Site Using Visual Studio (C#)
- Common Configuration Differences Between Development and Production (C#)
- Core Differences Between IIS and the ASP.NET Development Server (C#)
- Deploying a Database (C#)
- Configuring the Production Web Application to Use the Production Database (C#)
- Configuring a Website that Uses Application Services (C#)
- Strategies for Database Development and Deployment (C#)
- Displaying a Custom Error Page (C#)
- Processing Unhandled Exceptions (C#)
- Logging Error Details with ASP.NET Health Monitoring (C#)
- Logging Error Details with ELMAH (C#)
- Precompiling Your Website (C#)
- Users and Roles On Production Website (C#)
- ASP.NET Hosting Options (VB)
- Determining What Files Need to Be Deployed (VB)
- Deploying Your Site Using an FTP Client (VB)
- Deploying Your Site Using Visual Studio (VB)
- Common Configuration Differences Between Development and Production (VB)
- Core Differences Between IIS and the ASP.NET Development Server (VB)
- Deploying a Database (VB)
- Configuring the Production Web Application to Use the Production Database (VB)
- Configuring a Website that Uses Application Services (VB)
- Strategies for Database Development and Deployment (VB)
- Displaying a Custom Error Page (VB)
- Processing Unhandled Exceptions (VB)
- Logging Error Details with ASP.NET Health Monitoring (VB)
- Logging Error Details with ELMAH (VB)
- Precompiling Your Website (VB)
- Users and Roles On Production Website (VB)

ASP.NET Hosting Options (C#)

2/20/2020 • 8 minutes to read • Edit Online

by Scott Mitchell

Download PDF

ASP.NET web applications are typically designed, created, and tested in a local development environment and need to be deployed to a production environment once it is ready for release. This tutorial provides a high-level overview of the deployment process and serves as an introduction to this tutorial series.

Introduction

Web applications are typically designed, created, and tested in a development environment that is accessible only to the programmers working on the site. Once the application is ready to be released, it is moved to a production environment where the site can be accessed by anyone on the Internet. This deployment process introduces a number of challenges:

- A production environment must exist and be properly setup before an ASP.NET application can be deployed; moreover, the production environment must be kept up to date with the latest security patches.
- The correct set of markup files, code files, and support files must be copied from the development environment to the production environment. For data-driven applications, this might require copying the database schema and/or data, as well.
- There may be configuration differences between the two environments. The database connection string or email server used in the development environment will likely be different than the production environment. What's more, the behavior of the application may depend on the environment. For example, when an error occurs in development the error's details can be displayed on screen, but when an error occurs in production, a user-friendly error page should be displayed instead, and the error details emailed to the developers.

To obviate the first challenge - setting up and maintaining a production environment - many individuals and businesses outsource their production environments to *web hosting providers*. A web hosting provider is a company that manages the production environment on your behalf. There are countless web host providers, each with varying prices and service levels; see the "Finding a Web Host Provider" section for tips on locating such a service provider.

This is the first in a series of tutorials that look at the steps involved in deploying an ASP.NET web application to a production environment managed by a web host provider. Over the course of these tutorials we will examine:

- What files need to be deployed to the web host provider.
- Tools for streamlining the deployment process.
- How to deploy a database.
- Tips for deploying a database that uses the SQL-based Membership and Roles provider, along with ways to mimic the Website Administration Tool in a production environment.
- Strategies for smoothly updating the database in production with changes made during development.
- Techniques for logging errors that occur on production, and ways to notify developers when an error occurs.

These tutorials are geared to be concise and to provide step-by-step instructions with plenty of screen shots to walk you through the process visually. This inaugural tutorial provides an overview of the ASP.NET deployment process and advice on finding a web hosting provider. Let's get started!

An Overview of the ASP.NET Deployment Process

In a nutshell, deploying an ASP.NET application involves the following three steps:

1. Configure the web application, web server, and database in the production environment.
2. Synchronize the ASP.NET pages, code files, the assemblies in the `Bin` folder, and HTML-related support files like CSS and JavaScript files.
3. Synchronize the database schema and/or data.

The configuration information for a web application is typically located in the `Web.config` file, and includes database connection strings, error handling criteria, URL rewriting rules, and email server information. Oftentimes this information is different for an application in development versus the same application in production. For instance, when developing an application it's best to use a development database so that you're not testing against the production database. As a result, the database connection strings typically differ between development and production applications. Due to these differences, part of deployment involves making changes to the web application's configuration information.

In addition to web application configuration changes, Step 1 also may entail configuration for the web server and database. For example, if an ASP.NET page creates or deletes files from a directory on the web server then the web server needs to be configured to permit these file system modifications. Similarly, there may be permission or authentication settings that need to be made to the database.

Step 2 involves synchronizing the set of essential ASP.NET pages and support files between the development and production environments. The particular set of ASP.NET-related files that need to be synchronized between the two environments depends on the type of project you created in Visual Studio, and is the discussion in the next tutorial, *Determining What Files Need to Be Deployed*. The third and fourth tutorials - *Deploying Your Site Using FTP* and *Deploying Your Site Using Visual Studio* - examine different tools and techniques for syncing these files.

When building data-driven applications there are typically two databases being used: one for development and one on production. During development, the development database's schema may be modified to include new tables, columns, stored procedures, and triggers, or may be modified to remove or rename existing database objects. Between the time that these changes are made and the time the application is deployed to production, the development and production databases are out of sync. This asynchrony needs to be fixed during the deployment process. These challenges will be examined in future tutorials.

Finding a Web Host Provider

ASP.NET applications can be deployed to any web server that has the .NET Framework and Internet Information Services (IIS) installed. You could host a site from your personal computer, assuming you had a broadband connection to the Internet and the know how to configure your router to allow incoming web requests. You could also host a site from a computer in an intranet, as many companies do. The focus of these tutorials, however, is hosting your website with a web host provider.

> **NOTE**
>
> IIS is Microsoft's enterprise-grade web server. It ships with the non-Home editions of Windows, such as Windows Server 2008 and certain editions of Windows Vista. You do not need to install IIS to serve ASP.NET applications in a development environment, as Visual Studio includes the ASP.NET Development Web Server. However, the ASP.NET Development Web Server only accepts local connections and therefore cannot be used in a production environment.

Before you can deploy your site to a web host provider you must first decide what company to do business with. There are countless web hosting companies in the marketplace; a search for "web hosting company" returns more than five million results. How do you find the one that's right for you? Your favorite search engine is a good starting place, as are websites like TopHosts and HostCritique, which compare and contrast various hosting

services. I also advise asking your colleagues and coworkers for any recommendations; you can also ask for recommendations at the Hosting Open Forum here at the ASP.NET Forums.

Web hosting companies typically offer shared hosting plans and dedicated hosting plans. With shared hosting a single web server hosts dozens if not hundreds of different websites. With dedicated hosting you lease a computer from the company that serves your site and your site alone. A shared hosting plan might include support for ASP.NET pages, the ability to work with Microsoft Access databases, 5 GB of disk space, and 100 GB of monthly bandwidth traffic for $9.95 per month. Another shared hosting plan might include support for ASP.NET pages, access to the Microsoft SQL Server 2008 database server, 10 GB of disk space and 250 GB of monthly bandwidth traffic for $19.95 per month. Dedicated hosting plans are usually much more expensive, costing several hundred dollars per month, but offer better performance and more control than shared hosting options. What plan you choose depends on your budget, how much traffic your website receives, and the features you anticipate you'll need.

Two important considerations when choosing a web host provider are customer service and quality of service. If you have a question or a configuration problem, how long does it take from submitting your problem to the web host's helpdesk until you get a response? How reliable are the company's services? Do they frequently have database outages? How often does their email server go offline? You can always ask a company to provide details about their uptime and inquire about their customer service policy, but a more surefire way is to solicit the feedback of current and past customers, which you can do through online forums, newsgroups, and email listservs.

NOTE

Some web hosting companies focus their business on a particular technology stack, such as .NET or LAMP (**L** inux, **A** pache, **M** ySQL, and **P** HP), so make sure that the company you select hosts ASP.NET applications. Also check to ensure that they support the version of ASP.NET you are using to build your application. And if you are building a data-driven application, make sure that the web host offers the same database server and version that you are using.

Summary

ASP.NET web applications are typically designed, created, and tested in a local development environment. Once a version is ready for release, it is moved to a production environment. While it is possible to host ASP.NET websites on your personal computer or on servers within your company, many businesses and individuals choose to outsource their hosting to a web host provider.

This tutorial series examines the steps for deploying an ASP.NET application to a web host provider, exploring common challenges. This tutorial offered a high-level overview of the ASP.NET deployment process and gave tips for finding a suitable web host provider. The next tutorial looks at what ASP.NET-related files need to be copied to the production environment when deploying your website.

Happy Programming!

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Determining What Files Need to Be Deployed (C#)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> What files need to be deployed from the development environment to the production environment depends in part on whether the ASP.NET application was built using the Web Site Model or Web Application Model. Learn more about these two project models and how the project model affects deployment.

Introduction

Deploying an ASP.NET web application entails copying the ASP.NET-related files from the development environment to the production environment. The ASP.NET-related files include ASP.NET web page markup and code and client- and server-side support files. Client-side support files are those files referenced by your web pages and sent directly to the browser - images, CSS files and JavaScript files, for example. Server-side support files include those that are used to process a request on the server-side. This includes configuration files, web services, class files, Typed DataSets, and LINQ to SQL files, among others.

In general, all client-side support files should be copied from the development environment to the production environment, but what server-side support files get copied depends on whether you are explicitly compiling the server-side code into an assembly (a `.dll` file) or if you are having these assemblies auto-generated. This tutorial highlights what files need to be deployed when explicitly compiling the code into an assembly versus having this compilation step occur automatically.

Explicit Compilation Versus Automatic Compilation

ASP.NET web pages are divided into declarative markup and source code. The declarative markup portion includes HTML, Web controls, and databinding syntax; the code portion contains event handlers written in Visual Basic or C# code. The markup and code portions are typically separated into different files: `WebPage.aspx` contains the declarative markup while `WebPage.aspx.cs` houses the code.

Consider an ASP.NET page named Clock.aspx that contains a Label control whose Text property is set to the current date and time when the page loads. The declarative markup portion (in `Clock.aspx`) would contain the markup for a Label Web control - `<asp:Label runat="server" id="TimeLabel" />` - while the code portion (in `Clock.aspx.cs`) would have a `Page_Load` event handler with the following code:

```
protected void Page_Load(object sender, EventArgs e)
{
    TimeLabel.Text =  "The time at the beep is: " + DateTime.Now.ToString();
}
```

In order for the ASP.NET engine to service a request for this page, the page's code portion (the `WebPage.aspx.cs` file) must first be compiled. This compilation can happen explicitly or automatically.

If the compilation happens explicitly then the entire application's source code is compiled into one or more assemblies (`.dll` files) located in the application's `Bin` directory. If the compilation happens automatically then the resulting auto-generated assembly is, by default, placed in the `Temporary ASP.NET` Files folder, which can be found at `%WINDOWS%\Microsoft.NET\Framework\` *<version>*, although this location is configurable via the

`<compilation>` element in `Web.config`. With explicit compilation you must take some action to compile the ASP.NET application's code into an assembly, and this step occurs prior to deployment. With automatic compilation the compilation process occurs on the web server when the resource is first accessed.

Regardless of what compilation model you use, the markup portion of all ASP.NET pages (the `WebPage.aspx` files) need to be copied to the production environment. With explicit compilation you need to copy up the assemblies in the `Bin` folder, but you do not need to copy up the ASP.NET pages' code portions (the `WebPage.aspx.cs` files). With automatic compilation you need to copy up the code portion files so that the code is present and can be compiled automatically when the page is visited. The markup portion of each ASP.NET web page includes a `@Page` directive with attributes that indicate whether the page's associated code was already explicitly compiled or whether it needs to be automatically compiled. As a result, the production environment can work with either compilation model seamlessly and you do not need to apply any special configuration settings to indicate that explicit or automatic compilation is used.

Table 1 summarizes the different files to deploy when using explicit compilation versus automatic compilation. Note that regardless of the compilation model used you should always deploy the assemblies in the `Bin` folder, if that folder exists. The `Bin` folder contains the assemblies specific to the web application, which include the compiled source code when using the explicit compilation model. The `Bin` directory also contains assemblies from other projects and any open-source or third-party assemblies you may be using, and these need to be on the production server. Therefore, as a general rule of thumb, copy the `Bin` folder up to production when deploying. (If you are using the automatic compilation model and are not using any external assemblies then you won't have a `Bin` directory - that's OK!)

COMPILATION MODEL	DEPLOY MARKUP PORTION FILE?	DEPLOY SOURCE CODE FILE?	DEPLOY ASSEMBLIES IN `BIN` DIRECTORY?
Explicit Compilation	Yes	No	Yes
Automatic Compilation	Yes	Yes	Yes (if it exists)

Table 1: What files you deploy depends on the compilation model used.

Taking a Trip Down Memory Lane

What compilation approach is used depends, in part, on how the ASP.NET application is managed in Visual Studio. Since .NET's inception in the year 2000 there have been four different versions of Visual Studio - Visual Studio .NET 2002, Visual Studio .NET 2003, Visual Studio 2005, and Visual Studio 2008. Visual Studio .NET 2002 and 2003 managed ASP.NET applications using the *Web Application Project model*. The key features of the Web Application Project model are:

- The files that makeup the project are defined in a single project file. Any files not defined in the project file are not considered part of the web application by Visual Studio.
- Uses explicit compilation. Building the project compiles the code files within the project into a single assembly that is placed in the `Bin` folder.

When Microsoft released Visual Studio 2005 they dropped support for the Web Application Project model and replaced it with the Web Site Project model. The Web Site Project model differentiated itself from the Web Application Project model in the following ways:

- Rather than having a single project file that spells out the project's files, the file system is used instead. In short, any files within the web application folder (or subfolders) are considered part of the project.
- Building a project in Visual Studio does not create an assembly in the `Bin` directory. Instead, building a Web Site Project reports any compile-time errors.
- Support for automatic compilation. Web Site Projects are typically deployed by copying the markup and

source code to the production environment, although the code can be precompiled (explicit compilation).

Microsoft revived the Web Application Project model when it released Visual Studio 2005 Service Pack 1. However, Visual Web Developer continued to only support the Web Site Project model. The good news is that this limitation was dropped with Visual Web Developer 2008 Service Pack 1. Today you can create ASP.NET applications in Visual Studio (and Visual Web Developer) using either the Web Application Project model or the Web Site Project model. Both models have their pros and cons. Refer to Introduction to Web Application Projects: Comparing Web Site Projects and Web Application Projects for a comparison of the two models and to help decide what project model works best for your situation.

Exploring the Sample Web Application

The download for this tutorial includes an ASP.NET application called Book Reviews. The website mimics a hobby website someone might create to share their book reviews with the online community. This ASP.NET web application is very simple and consists of the following resources:

- `Web.config`, the application's configuration file.
- A master page (`Site.master`).
- Seven different ASP.NET pages:
  - ~`/Default.aspx` - the site's homepage.
  - ~`/About.aspx` - an "About the Site" page.
  - ~`/Fiction/Default.aspx` - a page listing the fiction books that have been reviewed.
    - ~`/Fiction/Blaze.aspx` - a review of the Richard Bachman novel *Blaze*.
  - ~/`Tech/Default.aspx` - a page listing the technology books that have been reviewed.
    - ~/`Tech/CYOW.aspx` - a review of *Create Your Own Website*.
    - ~/`Tech/TYASP35.aspx` - a review of *Teach Yourself ASP.NET 3.5 in 24 Hours*.
- Three different CSS files in the Styles folder.
- Four image files - a Powered by ASP.NET logo and images of the covers of the three reviewed books - all located in the `Images` folder.
- A `Web.sitemap` file, which defines the site map and is used to display menus in the `Default.aspx` pages in the root directory and `Fiction` and `Tech` folders.
- A class file named `BasePage.cs` that defines a base `Page` class. This class extends the functionality of the `Page` class by automatically setting the `Title` property based on the page's position in the site map. In a nutshell, any ASP.NET code-behind class that extends `BasePage` (instead of `System.Web.UI.Page`) will have its title set to a value depending on its position in the site map. For instance, when viewing the ~/`Tech/CYOW.aspx` page, the title is set to "Home : Technology : Create Your Own Website".

Figure 1 shows a screen shot of the Book Reviews website when viewed through a browser. Here you see the page ~/`Tech/TYASP35.aspx`, which reviews the book *Teach Yourself ASP.NET 3.5 in 24 Hours*. The breadcrumb that spans the top of the page and the menu in the left column are based on the site map structure defined in `Web.sitemap`. The image in the right upper corner is one of the book cover images located in the `Images` folder. The website's look and feel are defined via cascading style sheet rules spelled out by the CSS files in the Styles folder, while the overarching page layout is defined in the master page, `Site.master`.

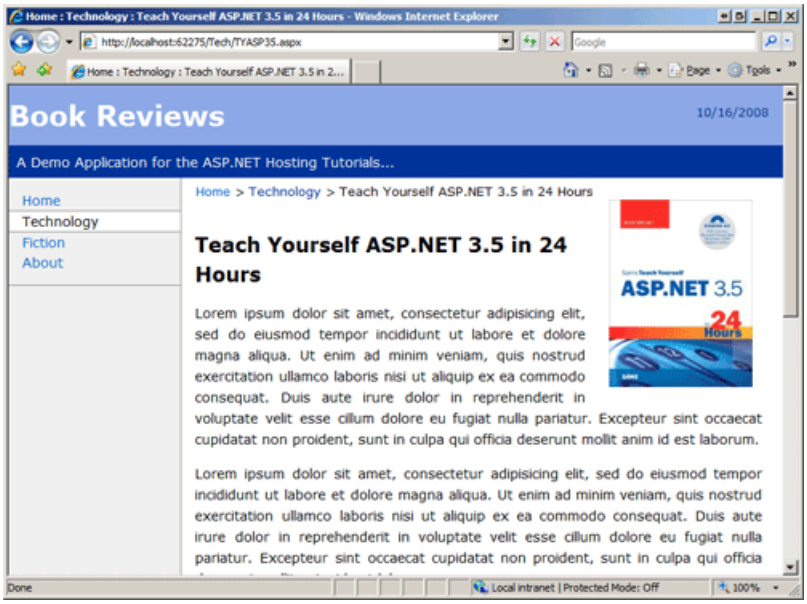


Figure 1: The Book Reviews website offers reviews on an assortment of titles (Click to view full-size image)

This application does not use a database; each review is implemented as a separate web page in the application. This tutorial (and the next several tutorials) walk through deploying a web application that does not have a database. However, in a future tutorial we will enhance this application to store reviews, reader comments, and other information within a database, and will explore what steps need to be performed to correctly deploy a data-driven web application.

> **NOTE**
>
> These tutorials focus on hosting ASP.NET applications with a web host provider and do not explore ancillary topics like ASP.NET's site map system or using a base `Page` class. For more information on these technologies, and for more background on other topics covered throughout the tutorial, refer to the Further Reading section at the end of each tutorial.

This tutorial's download has two copies of the web application, each implemented as a different Visual Studio project type: BookReviewsWAP, a Web Application Project, and BookReviewsWSP, a Web Site Project. Both projects were created with Visual Web Developer 2008 SP1 and use ASP.NET 3.5 SP1. To work with these projects start by unzipping the contents to your Desktop. To open the Web Application Project (BookReviewsWAP), navigate to the BookReviewsWAP folder and double-click the Solution file, `BookReviewsWAP.sln`. To open the Web Site Project (BookReviewsWSP), launch Visual Studio and then, from the File menu, choose the Open Web Site option, browse to the `BookReviewsWSP` folder on your Desktop, and click OK.

The remaining two sections in this tutorial look at what files you will need to copy to the production environment when deploying the application. The next two tutorials - *Deploying Your Site Using FTP* and *Deploying Your Site Using Visual Studio* - show different ways to copy these files to a web host provider.

Determining the Files to Deploy for the Web Application Project

The Web Application Project model uses explicit compilation - the project's source code is compiled into a single assembly each time you build the application. This compilation includes the ASP.NET pages' code-behind files (~/`Default.aspx.cs`, ~/`About.aspx.cs`, and so on), as well as the `BasePage.cs` class. The resulting assembly is named BookReviewsWAP.dll and is located in the application's `Bin` directory.

Figure 2 shows the files that make up the Book Reviews Web Application Project.

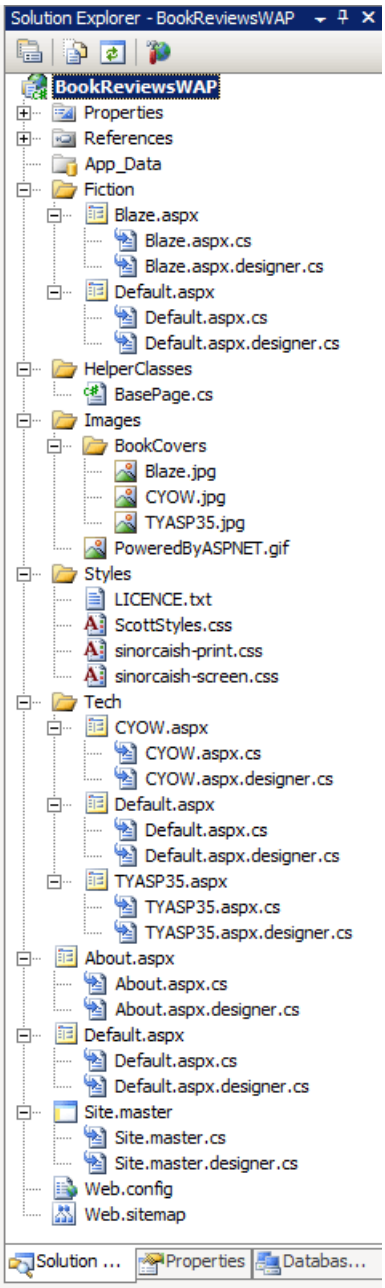


Figure 2: The Solution Explorer lists the files that comprise the Web Application Project

To deploy an ASP.NET application developed using the Web Application Project model start by building the application so as to explicitly compile the most recent source code into an assembly. Next, copy the following files to the production environment:

- The files that contain the declarative markup for every ASP.NET page, such as ~/`Default.aspx`, ~/`About.aspx`, and so on. Also, copy up the declarative markup for any master pages and User Controls.
- The assemblies (`.dll` files) in the `Bin` folder. You do not need to copy the program database files (`.pdb`) or any XML files you may find in the `Bin` directory.

You do not need to copy the ASP.NET pages' source code files to the production environment, nor do you need to copy the `BasePage.cs` class file.

NOTE

As Figure 2 shows, the `BasePage` class is implemented as a class file in the project, placed in folder named `HelperClasses`. When the project is compiled the code in the `BasePage.cs` file is compiled along with the ASP.NET pages' code-behind classes into the single assembly, `BookReviewsWAP.dll.` ASP.NET has a special folder named `App_Code` that is designed to hold class files for Web Site Projects. The code in the `App_Code` folder is automatically compiled and therefore should not be used with Web Application Projects. Instead, you should place your application's class files in a normal folder named `HelperClasses`, or `Classes`, or something similar. Alternatively, you can place class files in a separate Class Library project.

In addition to copying the ASP.NET-related markup files and the assembly in the `Bin` folder, you also need to copy the client-side support files - the images and CSS files - as well as the other server-side support files, `Web.config` and `Web.sitemap`. These client- and server-side support files need to be copied to the production environment regardless of whether you use explicit or automatic compilation.

Determining the Files to Deploy for the Web Site Project Files

The Web Site Project model supports automatic compilation, a feature not available when using the Web Application Project model. With explicit compilation you must compile your project's source code into an assembly and copy that assembly to the production environment. On the other hand, with automatic compilation you simply copy the source code to the production environment and it is compiled by the runtime on demand as needed.

The Build menu option in Visual Studio is present in both Web Application Projects and Web Site Projects. Building a Web Application Projects compiles the project's source code into a single assembly located in the `Bin` directory; building a Web Site Project checks for any compile-time errors, but does not create any assemblies. To deploy an ASP.NET application developed using the Web Site Project model all you need to do is copy the appropriate files to the production environment, but I would encourage you to first build the project to ensure that there are no compile-time errors.

Figure 3 shows the files that make up the Book Reviews Web Site Project.

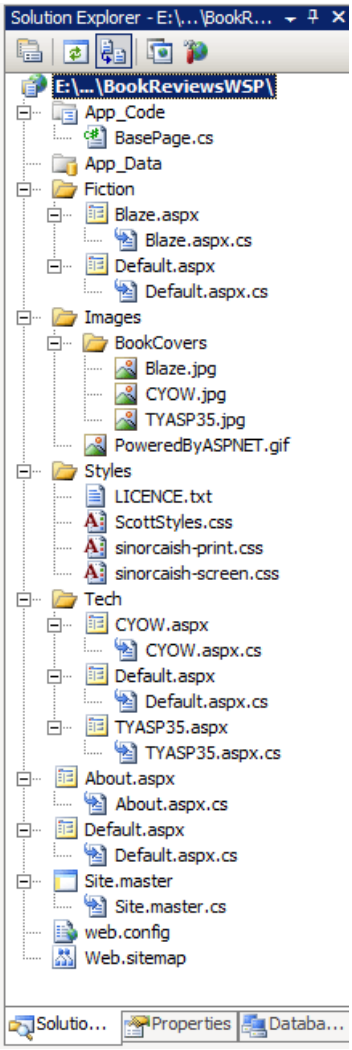


Figure 3: The Solution Explorer lists the files that comprise the Web Site Project

Deploying a Web Site Project involves copying all of the ASP.NET-related files to the production environment - that includes the markup pages for ASP.NET pages, master pages, and User Controls, along with their code files. You also need to copy up any class files, such as BasePage.cs. Note that the `BasePage.cs` file is located in the `App_Code` folder, which is a special ASP.NET folder used in Web Site Projects for class files. The special folder needs to be created on production, as well, as the class files in the `App_Code` folder on the development environment must be copied to the `App_Code` folder on production.

In addition to copying the ASP.NET markup and source code files, you also need to copy the client-side support files - the images and CSS files - as well as the other server-side support files, `Web.config` and `Web.sitemap`.

> **NOTE**
> Web Site Projects can also use explicit compilation. A future tutorial will examine how to explicitly compile a Web Site Project.

Summary

Deploying an ASP.NET application entails copying the necessary files from the development environment to the production environment. The precise set of files that need to be synced depends on whether the ASP.NET application's code is explicitly or automatically compiled. The compilation strategy employed is influenced by whether Visual Studio is configured to manage the ASP.NET application using the Web Application Project model or the Web Site Project model.

The Web Application Project model uses explicit compilation and compiles the project's code into a single

assembly in the `Bin` folder. When deploying the application, the markup portion of the ASP.NET pages and the contents of the `Bin` folder must be pushed up to the production environment; the source code in the application - the code files and code-behind classes, for example - do not need to be copied to the production environment.

The Web Site Project model uses automatic compilation by default, although it is possible to explicitly compile a Web Site Project, as we will see in future tutorials. Deploying an ASP.NET application that uses automatic compilation requires that the markup portion *and* source code must be copied to the production environment. The code is automatically compiled on the production environment when it is requested for the first time.

Now that we have examined what files need to be synced between the development and production environments we are ready to deploy the Book Reviews application to a web host provider.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Compilation Overview
- ASP.NET User Controls
- Examining ASP.NET's Site Navigation
- Introduction to Web Application Projects
- Master Page Tutorials
- Sharing Code Between Pages
- Using a Custom Base Class For Your ASP.NET Pages' Code-Behind Classes
- Visual Studio 2005's Web Site Project System: What Is It and Why Did We Do It?
- Walkthrough: Converting a Web Site Project to a Web Application Project in Visual Studio

PREVIOUS NEXT

Deploying Your Site Using an FTP Client (C#)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> The simplest way to deploy an ASP.NET application is to manually copy the necessary files from the development environment to the production environment. This tutorial shows how to use an FTP client to get the files from your desktop to the web host provider.

Introduction

The previous tutorial introduced a simple Book Review ASP.NET web application, which is comprised of a handful of ASP.NET pages, a master page, a custom base `Page` class, a number of images, and three CSS style sheets. We are now ready to deploy this application to a web host provider, at which point the application will be accessible to anyone with a connection to the Internet!

From our discussions in the *Determining What Files Need to Be Deployed* tutorial, we know what files need to be copied to the web host provider. (Recall that what files are copied depends on whether your application is explicitly or automatically compiled.) But how do we get the files from the development environment (our desktop) up to the production environment (the web server managed by the web host provider)? The **F**ile **T**ransfer **P**rotocol (FTP) is a commonly used protocol for copying files from one machine to another over a network. Another option is FrontPage Server Extensions (FPSE). This tutorial focuses on using stand-alone FTP client software to deploy the necessary files from the development environment to the production environment.

> **NOTE**
>
> Visual Studio includes tools for publishing websites via FTP; these tools, as well as a look at tools that use FPSE, are covered in the next tutorial.

To copy the files using FTP we need an *FTP client* on the development environment. An FTP client is an application that is designed to copy files from the computer it's installed to a computer that is running an *FTP server*. (If your web host provider supports file transfers via FTP, as most do, then there is an FTP server running on their web servers.) There are a number of FTP client applications available. Your web browser can even double as an FTP client. My favorite FTP client, and the one I will be using for this tutorial, is FileZilla, a free, open-source FTP client that is available for Windows, Linux, and Macs. Any FTP client will work, though, so feel free to use whatever client you are most comfortable with.

If you are following along you will need to create an account with a web host provider before you can complete this tutorial or subsequent ones. As noted in the previous tutorial, there are a gaggle of web host provider companies with a wide spectrum of prices, features, and quality of service. For this tutorial series I will be using Discount ASP.NET as my web host provider, but you can follow along with any web host provider as long as they support the ASP.NET version your site is developed in. (These tutorials were created using ASP.NET 3.5.) Also, because we will be copying the files to the web host provider using FTP in this tutorial and in future ones it is imperative that your web host provider supports FTP access to their web servers. Virtually all web host providers offer this feature, but you should double-check before signing up.

Deploying the Book Review Web Application Project

Recall that there are two versions of the Book Review web application: one implemented using the Web Application Project model (BookReviewsWAP) and the other using the Web Site Project model (BookReviewsWSP). The project type influences whether the site is compiled automatically or explicitly, and that compilation model dictates what files need to be deployed. Consequently, we will examine deploying the BookReviewsWAP and BookReviewsWSP projects separately, starting with the BookReviewsWAP. Take a moment to download these two ASP.NET applications if you have not done so already.

Launch the BookReviewsWAP project by navigating to the `BookReviewsWAP` folder and double-clicking the `BookReviewsWAP.sln` file. Before deploying the project it is important to build it to ensure that any changes to the source code are included in the compiled assembly. To build the project go to the Build menu and choose the Build BookReviewsWAP menu option. This compiles the source code in the project into a single assembly, `BookReviewsWAP.dll`, which is placed in the `Bin` folder.

We are now ready to deploy the necessary files! Launch your FTP client and connect to the web server at your web host provider. (When you sign up with a web hosting company they will email you information on how to connect to the FTP server; this includes the address for the FTP server as well as a username and password.)

Copy the following files from your desktop to the root website folder at your web host provider. When you FTP into the web server at the web host provider you are likely at the root website directory. However, some web host providers have a subfolder named `www` or `wwwroot` that serves as the root folder for your website files. Finally, when FTPing the files you may need to create the corresponding folder structure on the production environment - the `Bin` folder, the `Fiction` folder, the `Images` folder, and so on.

- `~/Default.aspx`
- `~/About.aspx`
- `~/Site.master`
- `~/Web.config`
- `~/Web.sitemap`
- The complete contents of the `Styles` folder
- The complete contents of the `Images` folder (and its subfolder, `BookCovers`)
- `~/Fiction/Default.aspx`
- `~/Fiction/Blaze.aspx`
- `~/Tech/Default.aspx`
- `~/Tech/CYOW.aspx`
- `~/Tech/TYASP35.aspx`
- `~/Bin/BookReviewsWAP.dll`

Figure 1 shows FileZilla after the necessary files have been copied. FileZilla displays the files on the local computer on the left and the files on the remote computer on the right. As Figure 1 shows, the ASP.NET source code files, such as `About.aspx.cs`, are on the local computer (the development environment) but were not copied to the web host provider (the production environment) because code files do not need to be deployed when using explicit compilation.

> **NOTE**
>
> There is no harm in having the source code files on the production server, as they are ignored. ASP.NET forbids HTTP requests to source code files by default so that even if the source code files are present on the production server they are inaccessible to visitors to your website. (That is, if a user tries to visit `http://www.yoursite.com/Default.aspx.cs` they will get an error page that explains that these types of files - `.cs` files - are forbidden.)

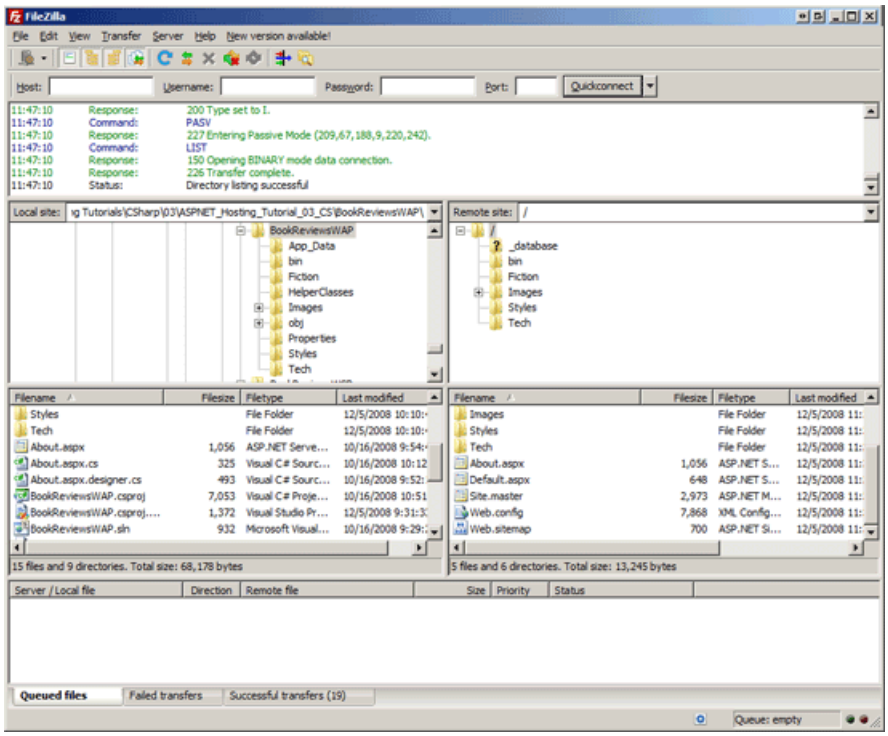


Figure 1: Use an FTP Client to Copy the Necessary Files from Your Desktop to the Web Server at the Web Host Provider (Click to view full-size image)

After deploying your site take a moment to test the site. If you have purchased a domain name and configured the DNS settings properly, you can visit your site by entering your domain name. Alternatively, your web host provider should have supplied you with a URL to your site, which will look something like *accountname.webhostprovider*.com or *webhostprovider*.com/*accountname*. For example, the URL for my account on Discount ASP.NET is: `http://httpruntime.web703.discountasp.net`.

Figure 2 shows the deployed Book Reviews site. Note that I am viewing it on Discount ASP.NET's servers, at `http://httpruntime.web703.discountasp.net`. At this point in time anyone with a connection to the Internet could view my website! As we'd expect, the site looks and behaves just as it did when testing it in the development environment.

> **NOTE**
>
> If you get an error when viewing your application take a moment to ensure that you deployed the correct set of files. Next, check the error message to see if it reveals any clues as to the problem. Following that, you can turn to your web host company's helpdesk or post your question to the appropriate forum at the ASP.NET Forums.

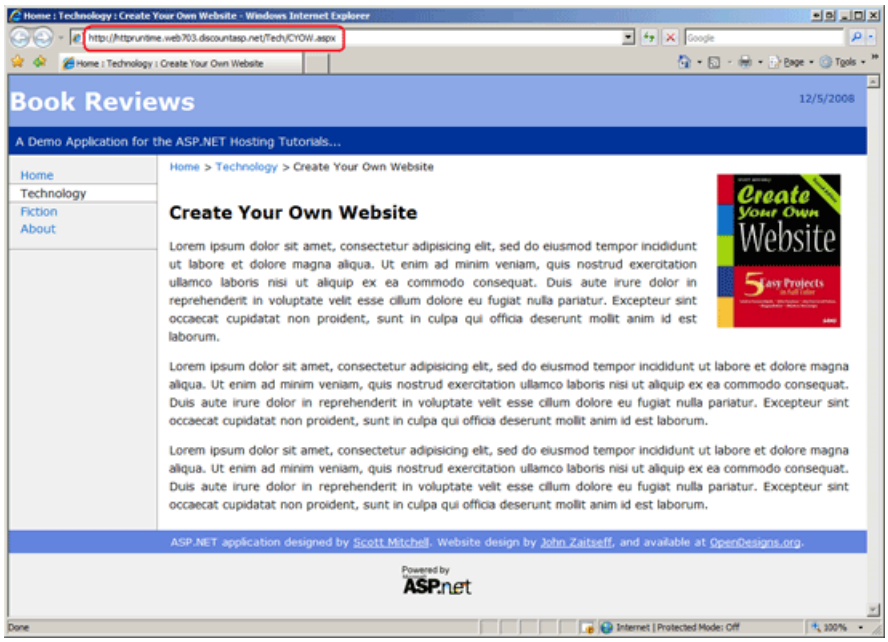


Figure 2: The Book Reviews Site is Now Accessible to Anyone with an Internet Connection (Click to view full-size image)

Deploying the Book Review Web Site Project

When deploying an ASP.NET application that uses automatic compilation, such as the BookReviewsWSP Web Site Project, there is no compiled assembly in the `Bin` folder. As a result, the web application's source code files must be deployed to the production environment. Let's walk through this process.

As with the Web Application Project it's wise to first Build the application before deploying it. While building a Web Site Project does not create an assembly, it does check for any compile-time errors in the page. Better to find these errors now rather than having a visitor to your site discover them for you!

Once you've successfully built the project, use your FTP client to copy the following files to the root website folder at your web host provider. You may need to create the corresponding folder structure on the production environment.

> **NOTE**
>
> If you already deployed the BookReviewsWAP project but still want to try deploying the BookReviewsWSP project, first delete all of the files on the web server that were uploaded when deploying BookReviewsWAP and then deploy the files for BookReviewsWSP.

- `~/Default.aspx`
- `~/Default.aspx.cs`
- `~/About.aspx`
- `~/About.aspx.cs`
- `~/Site.master`
- `~/Site.master.cs`
- `~/Web.config`
- `~/Web.sitemap`
- The complete contents of the `Styles` folder
- The complete contents of the `Images` folder (and its subfolder, `BookCovers`)
- `~/App_Code/BasePage.cs`

- `~/Fiction/Default.aspx`
- `~/Fiction/Default.aspx.cs`
- `~/Fiction/Blaze.aspx`
- `~/Fiction/Blaze.aspx.cs`
- `~/Tech/Default.aspx`
- `~/Tech/Default.aspx.cs`
- `~/Tech/CYOW.aspx`
- `~/Tech/CYOW.aspx.cs`
- `~/Tech/TYASP35.aspx`
- `~/Tech/TYASP35.aspx.cs`

Figure 3 shows FileZilla after copying up the necessary files. As you can see, the ASP.NET source code files, such as `About.aspx.cs`, are present on both the local computer (the development environment) and the web host provider (the production environment) because code files need to be deployed when using automatic compilation.

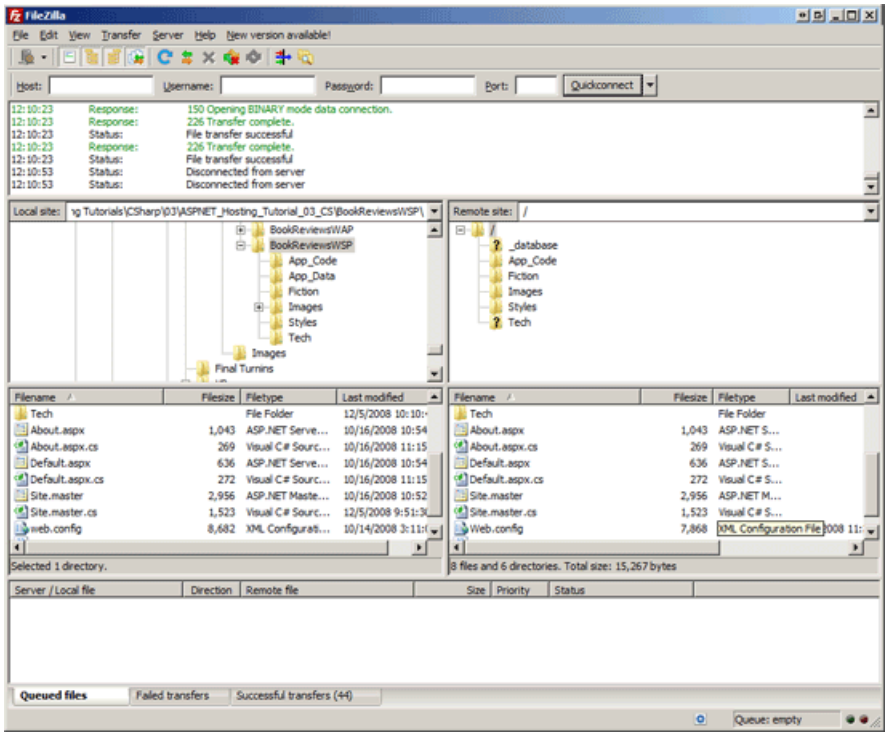


Figure 3: Use an FTP Client to Copy the Necessary Files from Your Desktop to the Web Server at the Web Host Provider (Click to view full-size image)

The user experience is not affected by the application's compilation model. The same ASP.NET pages are accessible and they look and behave the same whether the website was created using the Web Application Project model or the Web Site Project model.

Updating a Web Application on Production

Web application development and deployment are not a one-time process. For example, when creating the Book Review website I built the various pages and wrote the accompanying code on my personal computer (the development environment). After reaching a certain stable state, I deployed my application so that others could visit the site and read my reviews. But deployment does not mark the end of my development on this site. I may add more book reviews or implement new features, such as allowing my visitors to rate books or leave their own comments. Such enhancements would be developed on the development environment and, when completed, would need to be deployed. Development and deployment, therefore, are cyclical. You develop an application and then deploy it. While the site is live and in production, new features are added and bugs are fixed over time, which necessitates re-deploying the application. And so on and so on.

As you might expect, when re-deploying a web application you only need to copy new and changed files. There's no need to re-deploy unchanged pages or server- or client-side support files (although there's no harm in doing so).

> **NOTE**
>
> One thing to keep in mind when using explicit compilation is that anytime you add a new ASP.NET page to the project or make any code-related changes, you need to rebuild your project, which updates the assembly in the `Bin` folder. Consequently, you'll need to copy this updated assembly to production when updating a web application on production (along with the other new and updated content).

Also understand that any changes to the `Web.config` or the files in the `Bin` directory stops and restarts the website's Application Pool. If your session state is stored using the `InProc` mode (the default) then your site's visitors will lose their session state whenever these key files are modified. To avoid this pitfall, consider storing session using the `StateServer` or `SQLServer` modes. For more information on this topic read Session-State Modes.

Finally, keep in mind that re-deploying an application can take anywhere from a few seconds to several minutes, depending on the number and size of files that need to be copied to the production environment. During this time users visiting your site may experience errors or odd behavior. You can "turn off" your entire application by adding a page named `App_Offline.htm` to your application's root directory that explains to your users that the site is down for maintenance (or whatever) and will be back up shortly. When the `App_Offline.htm` file is present, the ASP.NET runtime redirects all incoming requests to that page.

Summary

Deploying a web application entails copying the necessary files from the development environment to the production environment. The most common means by which files are transferred over a network is the File Transfer Protocol (FTP), and most web host providers support FTP access to their web servers. In this tutorial we saw how to use an FTP client to deploy the needed files to the web server. Once deployed, the website can be visited by anyone with a connection to the Internet!

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- App_Offline.htm and Working Around the "IE Friendly Errors" Feature
- Session-State Modes

PREVIOUS NEXT

Deploying Your Site Using Visual Studio (C#)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Visual Studio includes tools for deploying a website. Learn more about these tools in this tutorial.

Introduction

The preceding tutorial looked at how to deploy a simple ASP.NET web application to a web host provider. Specifically, the tutorial showed how to use an FTP client like FileZilla to transfer the necessary files from the development environment to the production environment. Visual Studio also offers built-in tools to facilitate deployment to a web host provider. This tutorial examines two of these tools: the Copy Web Site tool, where you can move files to and from a remote web server using FTP or FrontPage Server Extensions; and the Publish tool, which copies the entire website to a specified location.

> **NOTE**
>
> Other deployment-related tools offered by Visual Studio include Web Setup Projects and Web Deployment Projects Add-In. Web Setup Projects package a website's contents and configuration information into a single MSI file. This option is most useful for websites that are deployed within an intranet or for companies that sell a pre-packaged web application that customers install on their own web servers. The Web Deployment Projects Add-In is a Visual Studio Add-In that facilitates specifying configuration differences between builds for development environments and production environments. Web Setup Projects are not discussed in this tutorial series; Web Deployment Projects are summarized in the *Common Configuration Differences Between Development and Production* tutorial.

Deploying Your Site Using the Copy Web Site Tool

Visual Studio's Copy Web Site tool is similar in functionality to a stand-alone FTP client. In a nutshell, the Copy Web Site tool allows you to connect to a remote web site through FTP or FrontPage Server Extensions. Similar to FileZilla's user interface, the Copy Web Site user interface consists of two panes: the left pane lists the local files while the right pane lists those files on the destination server.

> **NOTE**
>
> The Copy Web Site tool is only available for Web Site Projects. Visual Studio does offer this tool when you are working with a Web Application Project.

Let's take a look at using the Copy Web Site tool to publish the Book Review application to production. Because the Copy Web Site tool only works with projects that use the Web Site Project model we can only examine using this tool with the BookReviewsWSP project. Open that project.

Launch the Copy Web Site tool project by clicking the Copy Web Site icon in the Solution Explorer (this icon is circled in Figure 1); alternatively, you can select the Copy Web Site option from the Website menu. Either approach launches the Copy Web Site user interface shown in Figure 1; only the left pane in Figure 1 is populated because we have yet to connect to a remote server.

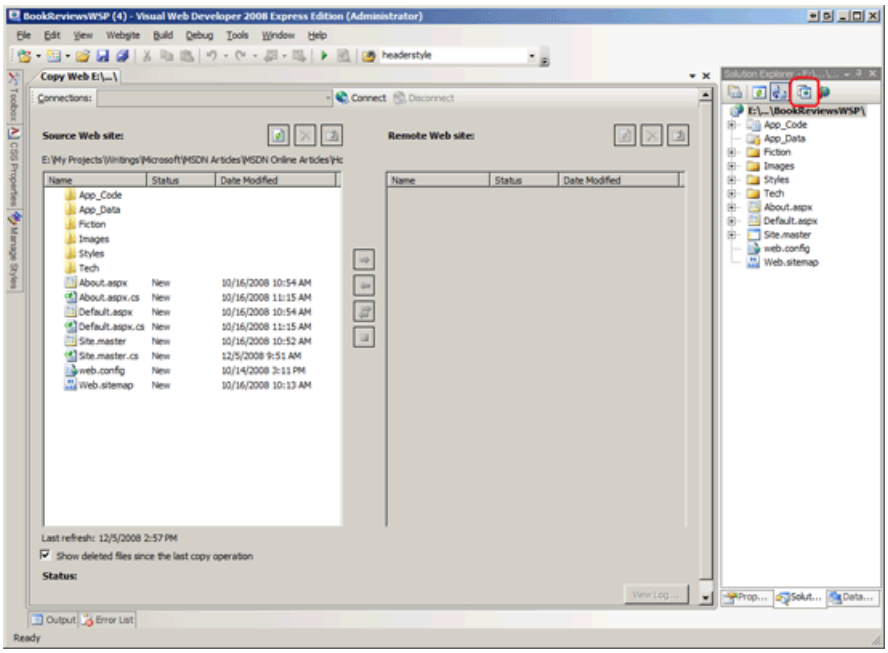


Figure 1: The Copy Web Site Tool's User Interface is Divided Into Two Panes (Click to view full-size image)

In order to deploy our site we need to first connect to the web host provider. Click the Connect button at the top of the Copy Web Site user interface. This displays the Open Web Site dialog box shown in Figure 2.

You can connect to the destination website by selecting one of the four options from the left:

- **File System** - select this to deploy your site to a folder or network share accessible from your computer.
- **Local IIS** - use this option to deploy the site to the IIS web server installed on your computer.
- **FTP Site** - connect to a remote web site using FTP.
- **Remote Site** - connect to a remote website using FrontPage Server Extensions.

Most web host providers support FTP, but fewer offer FrontPage Server Extension support. For that reason, I've selected the FTP Site option and then entered the connection information as shown in Figure 2.

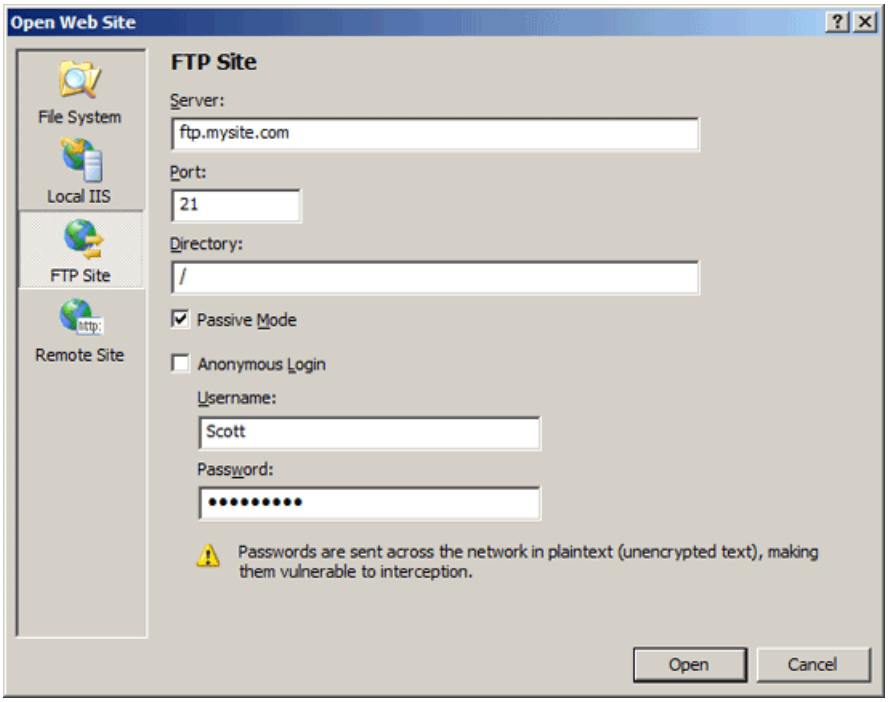


Figure 2: Specify the Destination Website (Click to view full-size image)

After you connect, the Copy Web Site tool loads the files at the remote site in the right pane and indicates the status of each file: New, Deleted, Changed, or Unchanged. You can copy a file from the local site to the remote

site, or vice-a-versa.

Let's add a new page to the BookReviewsWSP project and then deploy it so that we can see the Copy Web Site tool in action. Create a new ASP.NET page in Visual Studio in the root directory named `Privacy.aspx`. Have the page use the master page `Site.master` and add your site's privacy policy to this page. Figure 3 shows Visual Studio after this page has been created.

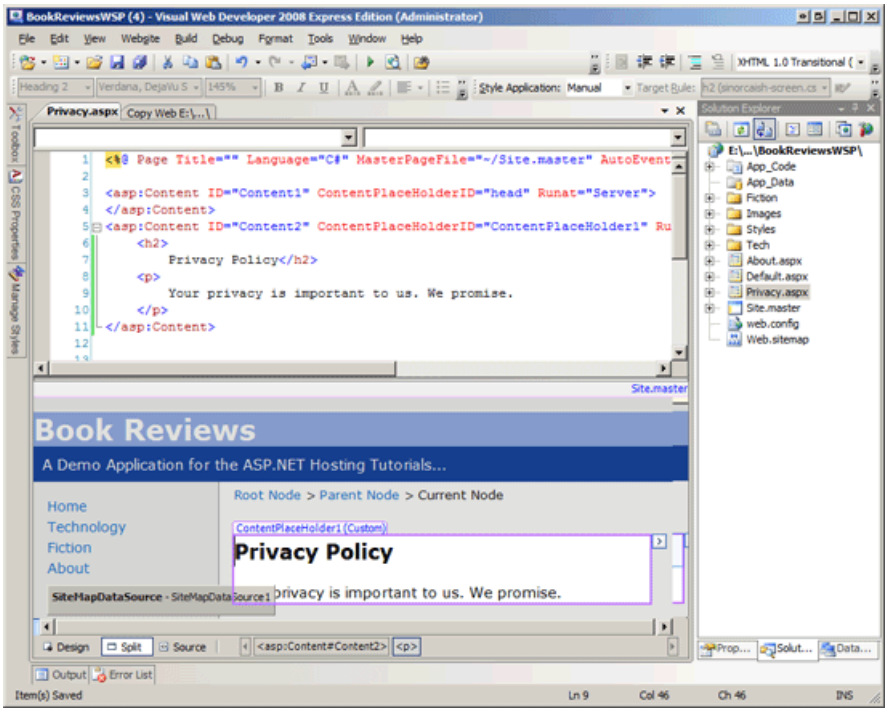


Figure 3: Add a New Page Named `Privacy.aspx` to the Website's Root Folder (Click to view full-size image)

Next, return to the Copy Web Site user interface. As Figure 4 shows, the left pane now includes the new files - `Policy.aspx` and `Policy.aspx.cs`. What's more, these files are marked with an arrow icon and a Status of New, indicating that they exist on the local site but not on the remote site.

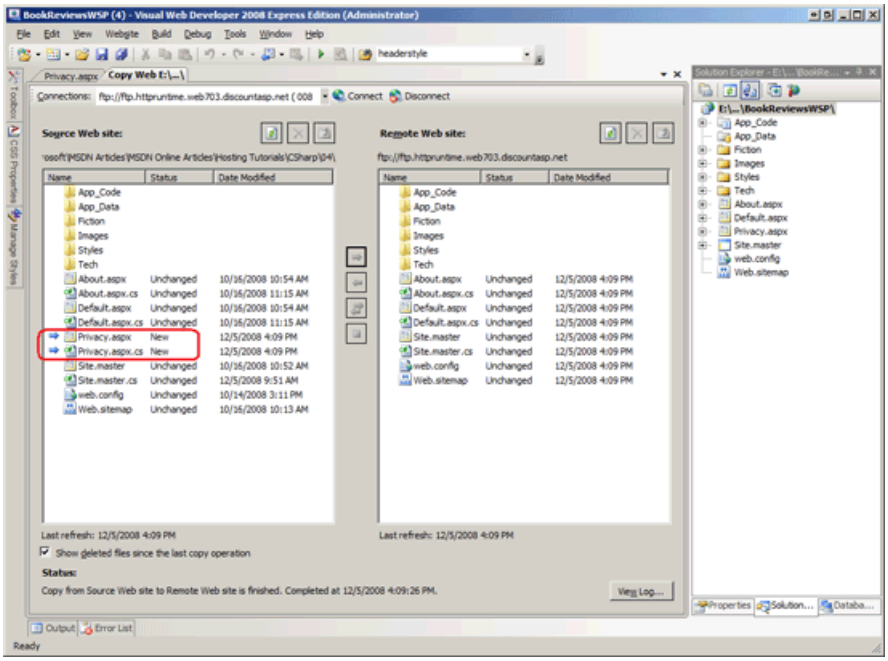


Figure 4: The Copy Web Site Tool Includes the New `Privacy.aspx` Page in its Left Pane (Click to view full-size image)

To deploy the new files select them and then click the arrow icon to transfer them to the remote site. After the transfer completes the `Policy.aspx` and `Policy.aspx.cs` files exist on both the local and remote sites with the status Unchanged.

Along with listing new files, the Copy Web Site tool highlights any files that differ between the local and remote sites. To see this in action, return to the `Privacy.aspx` page and add a few more words to the privacy policy. Save the page and then return to the Copy Web Site tool. As Figure 5 shows, the `Privacy.aspx` page in the left pane has a status of Changed indicating that it is out of sync with the remote site.

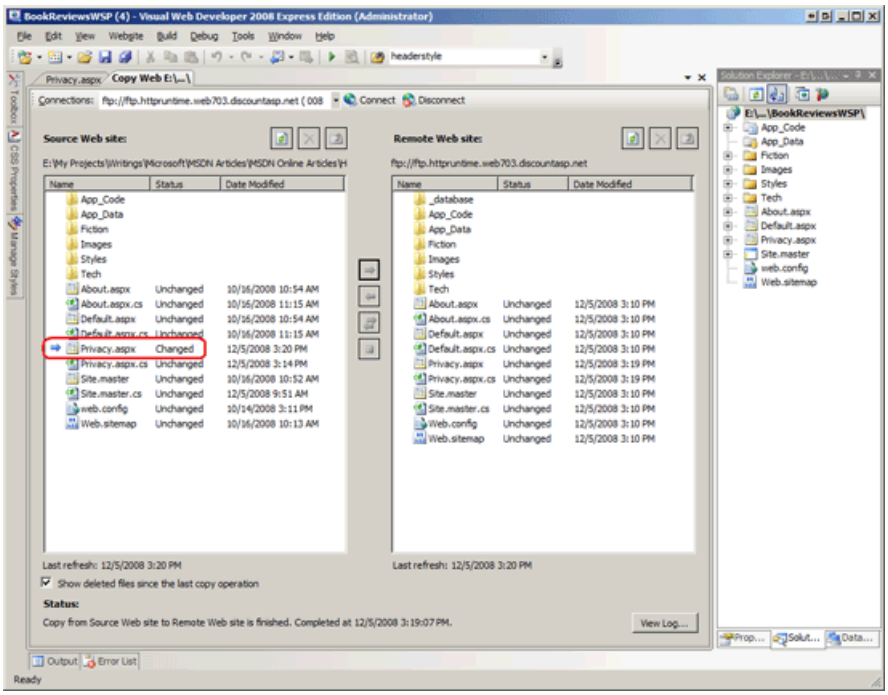


Figure 5: The Copy Web Site Tool Indicates that the `Privacy.aspx` Page has been Changed (Click to view full-size image)

The Copy Web Site tool also indicates if a file has been deleted since the last copy operation. Delete the `Privacy.aspx` from the local project and refresh the Copy Web Site tool. The `Privacy.aspx` and `Privacy.aspx.cs` files remain listed in the left pane, but have a Deleted status indicating that they have been removed since the last copy operation.

Publishing a Web Application

Another way to deploy your web application from within Visual Studio is to use the Publish option, which is accessible via the Build menu. The Publish option explicitly compiles the application and then copies all of the necessary files up to the specified remote site. As we'll see shortly, the Publish option is more blunt than the Copy Web Site tool. Whereas the Copy Web Site tool lets you examine the files on the local and remote sites and permits you to upload or download individual files as needed, the Publish option deploys the entire web application.

In addition to copying all of the needed files to the specified remote site, the Publish option also explicitly compiles the application. Given that Web Application Projects need to be explicitly compiled it should come as no surprise that the Publish option is available for Web Application Projects. What may be a bit surprising is that the Publish option is also available for Web Site Projects. As noted in the *Determining What Files Need to Be Deployed* tutorial, Web Site Projects can be explicitly compiled through a process referred to as *pre-compilation*. This tutorial focuses on using the Publish option with Web Application Projects; a future tutorial will explore pre-compilation, at which point we'll return to look at using the Publish option with Web Site Projects.

> **NOTE**
>
> While the Publish option is available in Visual Studio for both Web Site Projects and Web Application Projects, Visual Web Developer only offers the Publish option for Web Application Projects.

Let's look at deploying the Book Reviews application using the Publish option. Start by opening BookReviewsWAP (the Web Application Project) in Visual Studio. From the Publish menu choose the Build BookReviewsWAP project. This brings up a dialog box that prompts for the target location, among other configuration options (see Figure 6). Much like with the Copy Web Site tool you can enter a location that points to a local folder, a local website on IIS, a remote website that supports FrontPage Server Extensions, or an FTP server address. You can choose whether to replace the files on the remote web server with the deployed files or to delete all of the content on the remote site before publishing. You can also specify whether to copy:

- Only the files in the project needed to run the application, which omits the unneeded source code and project-related files.
- All project files, which includes the source code files and Visual Studio project files like the Solution file.
- All files in the source project folder, which copies all files in the source project folder regardless of whether they're included in the project.

There's also an option to upload the contents of the `App_Data` folder.

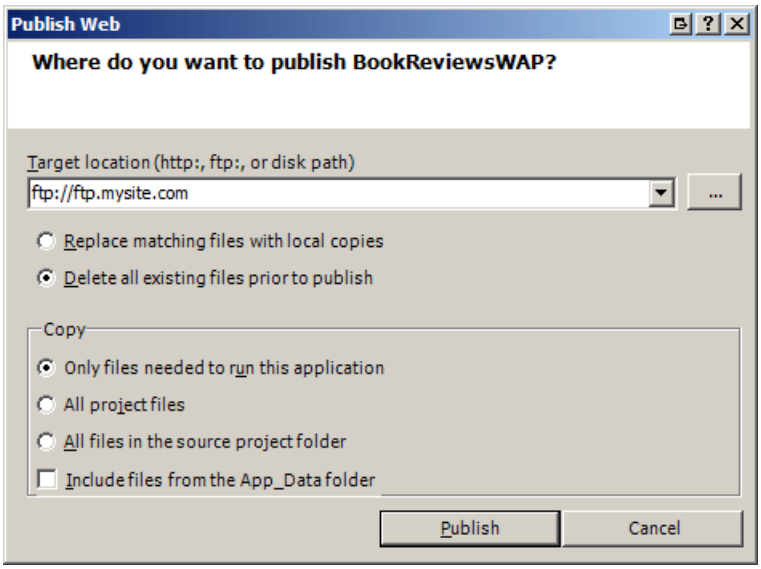


Figure 6: Specify the Destination Website (Click to view full-size image)

For the Book Review application the remote site contains the files deployed when copying the BookReviewsWSP project via the Copy Web Site tool. Therefore, let's have the Publish option start by deleting all existing content. Also, let's just copy the necessary files rather than cluttering the production environment with unneeded source code and project files. After specifying these options, click the Publish button. Over the next several seconds Visual Studio will deploy the necessary files to the destination site, displaying its progress in the Output window.

Figure 7 shows the files on the FTP site after the Publish operation has completed. Note that only the markup pages and the necessary sever- and client-side support files have been uploaded.

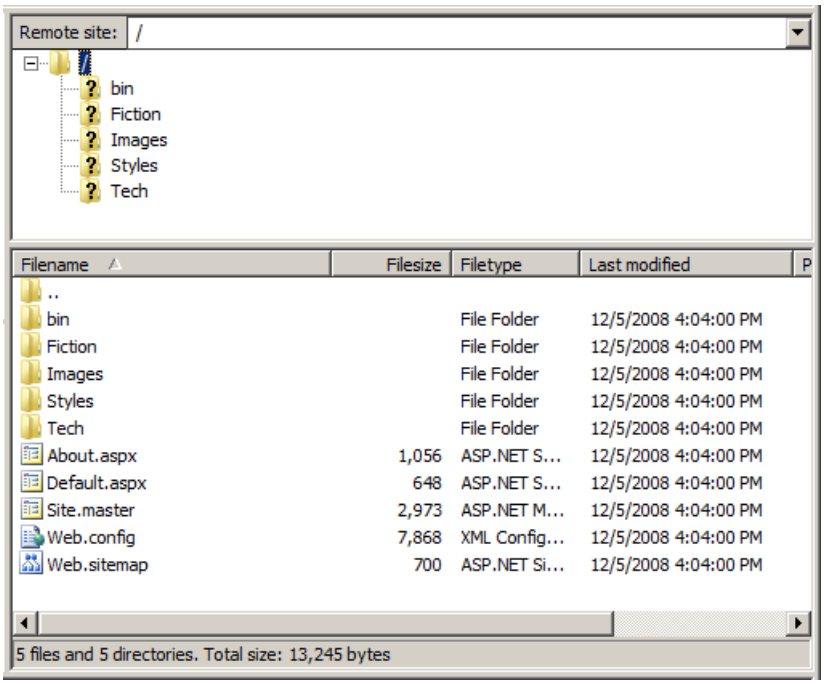


Figure 7: Only the Needed Files Were Published to the Production Environment (Click to view full-size image)

The Publish option is a less nuanced tool than the Copy Web Site tool. Whereas the Copy Web Site tool allows you to inspect the files on the local and remote sites and see how they differ, the Publish option provides no such interface. Moreover, the Copy Web Site tool enables you to make one-off changes, uploading or deleting individual files. The Publish option does not allow such fine-grained control; instead, it publishes the *entire* application. This behavior has its pros and cons. On the plus side, you know when using the Publish option you won't be forgetting to upload an important file. But consider what happens if you have made a small change to a very large website - with the Publish option you cannot update that page or two that has been modified, but instead you must wait while Visual Studio deploys the entire site.

It's not uncommon for there to be certain files whose content differs between the production and development environments. A key example is the application's configuration file, `Web.config`. Because the Publish option blindly copies the web application files it overwrites the production environment's customized configuration files with the version in the development environment. The subsequent tutorial explores this topic further and offers tips for deploying a web application when such differences exist.

Summary

Deploying a website involves copying the necessary files from the development environment to the production environment. The previous tutorial showed how to transfer files using an FTP client like FileZilla. This tutorial examined two deployment tools in Visual Studio: the Copy Web Site tool and the Publish option. The Copy Web Site tool is similar to an FTP client in that it has a two-paned interface listing the files on the local computer and a specified remote computer that makes it easy to upload or download files between the two computers. The Publish option is a more blunt tool that explicitly compiles the project and then deploys the entire application to the specified destination.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Copying Web Site with the Copy Web Site Tool
- How Do I: Deploy a Web Site Using the Copy Web Site Tool (Video)
- How To: Publish Web Application Projects
- How To: Publish Web Sites

- Setup and Deployment Projects in Visual Studio

PREVIOUS NEXT

Common Configuration Differences Between Development and Production (C#)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download PDF

In earlier tutorials we deployed our website by copying all of the pertinent files from the development environment to the production environment. However, it is not uncommon for there to be configuration differences between environments, which necessitates that each environment have a unique Web.config file. This tutorial examines typical configuration differences and looks at strategies for maintaining separate configuration information.

Introduction

The last two tutorials walked through deploying a simple web application. The *Deploying Your Site Using an FTP Client* tutorial showed how to use a stand-alone FTP client to copy the necessary files from the development environment up to production. The preceding tutorial, *Deploying Your Site Using Visual Studio*, looked at deployment using Visual Studio's Copy Web Site tool and Publish option. In both tutorials every file in the production environment was a copy of a file on the development environment. However, it is not uncommon for configuration files in the production environment to differ from those in the development environment. A web application's configuration is stored in the `Web.config` file and typically includes information about external resources, such as database, web, and email servers. It also spells out the behavior of the application in certain situations, such as the course of action to take when an unhandled exception occurs.

When deploying a web application it is important that the correct configuration information end up in the production environment. In most cases the `Web.config` file in the development environment cannot be copied to the production environment as-is. Instead, a customized version of `Web.config` needs to be uploaded to production. This tutorial briefly reviews some of the more common configuration differences; it also summarizes some techniques for maintaining different configuration information between the environments.

Typical Configuration Differences Between the Development and Production Environments

The `Web.config` file includes an assortment of configuration information for an ASP.NET application. Some of this configuration information is the same regardless of the environment. For instance, the authentication settings and URL authorization rules spelled out in the `Web.config` file's `<authentication>` and `<authorization>` elements are usually the same regardless of the environment. But other configuration information - such as information about external resources - typically differs depending on the environment.

Database connections strings are a prime example of configuration information that differs based on the environment. When a web application communicates with a database server it must first establish a connection, and that is achieved through a connection string. While it is possible to hard-code the database connection string directly in the web pages or the code that connects to the database, it is best to place it `Web.config`'s `<connectionStrings>` element so that the connection string information is in a single, centralized location. Oftentimes a different database is used during development than is used in production; consequently, the connection string information must be unique for each environment.

> **NOTE**
> Future tutorials explore deploying data-driven applications, at which point we'll dive into the specifics of how database connection strings are stored in the configuration file.

The intended behavior of the development and production environments differs substantially. A web application in the development environment is being created, tested, and debugged by a small group of developers. In the production environment that same application is being visited by many different simultaneous users. ASP.NET includes a number of features that help developers in testing and debugging an application, but these features should be disabled for performance and security reasons when in the production environment. Let's look at a few such configuration settings.

Configuration Settings That Impact Performance

When an ASP.NET page is visited for the first time (or the first time after it has changed), its declarative markup must be converted into a class and this class must be compiled. If the web application uses automatic compilation then the page's code-behind class needs to be compiled, too. You can configure an assortment of compilation options via the `Web.config` file's `<compilation>` element.

The debug attribute is one of the most important attributes in the `<compilation>` element. If the `debug` attribute is set to "true" then the compiled assemblies include debug symbols, which are needed when debugging an application in Visual Studio. But debug symbols increase the size of the assembly and impose additional memory requirements when running the code. Furthermore, when the `debug` attribute is set to "true" any content returned by `WebResource.axd` is not cached, meaning that each time a user visits a page they will need to re-download the static content returned by `WebResource.axd`.

> **NOTE**
> `WebResource.axd` is a built-in HTTP Handler introduced in ASP.NET 2.0 that server controls use to retrieve embedded resources, such as script files, images, CSS files, and other content. For more information on how `WebResource.axd` works and how you can use it to access embedded resources from your custom server controls, see Accessing Embedded Resources Through a URL Using `WebResource.axd`.

The `<compilation>` element's `debug` attribute is usually set to "true" in the development environment. In fact, this attribute must be set to "true" in order to debug a web application; if you try to debug an ASP.NET application from Visual Studio and the `debug` attribute is set to "false", Visual Studio will display a message explaining that the application cannot be debugged until the `debug` attribute is set to "true" and will offer to make this change for you.

You should **never** have the `debug` attribute set to "true" in a production environment because of its impact on performance. For a more thorough discussion on this topic, refer to Scott Guthrie's blog post, Don't Run Production ASP.NET Applications With `debug="true"` Enabled.

Custom Errors and Tracing

When an unhandled exception occurs in an ASP.NET application it bubbles up to the runtime at which point one of three things happens:

- A generic runtime error message is displayed. This page informs the user that there was a runtime error, but does not provide any details about the error.
- An exception details message is displayed, which includes information on the exception that was just thrown.
- A custom error page is displayed, which is an ASP.NET page that you create that displays any message you desire.

What happens in the face of an unhandled exception depends on the `Web.config` file's `<customErrors>` section.

When developing and testing an application it helps to see details of any exception in the browser. However, showing exception details in an application on production is a potential security risk. Moreover, it's unflattering and makes your website look unprofessional. Ideally, in the event of an unhandled exception a web application in the development environment will show the exception's details while the same application in production will show a custom error page.

> **NOTE**
>
> The default `<customErrors>` section setting shows the exception details message only when the page is being visited through localhost, and shows the generic runtime error page otherwise. This isn't ideal, but it's assuring to know that the default behavior doesn't reveal exception details to non-local visitors. A future tutorial examines the `<customErrors>` section in more detail and shows how to have a custom error page shown when an error occurs in production.

Another ASP.NET feature that is useful during development is tracing. Tracing, if enabled, records information about each incoming request and provides a special web page, `Trace.axd`, for viewing recent request details. You can turn on and configure tracing via the `<trace>` element in `Web.config`.

If you enable tracing make sure that it is disabled in production. Because the trace information includes cookies, session data, and other potentially sensitive information, it is important to disable tracing in production. The good news is that, by default, tracing is disabled and the `Trace.axd` file is only accessible through localhost. If you change these default settings in development make sure that they are turned back off in production.

Techniques for Maintaining Separate Configuration Information

Having different configuration settings in the development and production environments complicates the deployment process. In the previous two tutorials the deployment process involved copying all of the necessary files from development to production, but that approach only works if the configuration information is the same in both environments. There are a variety of techniques for deploying an application with varying configuration information. Let's catalog some of these options for hosted web applications.

Manually Deploying the Production Environment Configuration File

The simplest approach is to maintain two versions of the `Web.config` file: one for the development environment and one for the production environment. Deploying a site to production entails copying all of the files to the production server in the development environment *except* for the `Web.config` file. Instead, the production environment-specific `Web.config` file would be copied to production.

This approach is not very sophisticated, but it is easy to implement because configuration information changes infrequently. It works best for applications with a small development team that are hosted on a single web server and whose configuration information is infrequently changed. It's most tenable when manually deploying the application files using a stand-alone FTP client. When using Visual Studio's Copy Web Site tool or Publish option you'll need to first swap out the deployment-specific `Web.config` file with the production-specific one before deploying, and then swap them back after deployment completes.

Change the Configuration During the Build or Deployment Process

The discussions thus far have assumed an ad-hoc build and deployment process. Many larger software projects have more formalized processes that make use of open-source, home-grown, or third-party tools. For such projects you can likely customize the build or deployment process to appropriately modify the configuration information before it is pushed to production. If you build your web application using MSBuild, NAnt, or some other build tool, you can likely add a build step to modify the `Web.config` file to include the production-specific settings. Or your deployment workflow could programmatically connect to the source control server and retrieve the appropriate `Web.config` file.

The actual approach for getting the appropriate configuration information to production varies widely based on your tools and workflow. As such, we will not delve into this topic further. If you are using a popular build tool like

MSBuild or NAnt you can find deployment articles and tutorials specific to these tools through a web search.

Managing Configuration Differences via the Web Deployment Project Add-In

In 2006 Microsoft released the Web Development Project Add-In for Visual Studio 2005. An Add-In for Visual Studio 2008 was released in 2008. This Add-In allows for ASP.NET developers to create a separate Web Deployment Project alongside their web application project that, when built, explicitly compiles the web application and copies the files to deploy to a local output directory. The Web Application Project uses MSBuild behind the scenes.

By default, the development environment's `Web.config` file is copied to the output directory, but you can setup the Web Deployment Project to customize the

configuration information that gets copied to this directory in the following ways:

- Via `Web.config` file section replacement, in which you specify the section to replace and an XML file that contains the replacement text.
- By providing a path to an external configuration source file. With this option selected, the Web Deployment Project copies a particular `Web.config` file to the output directory (rather than the `Web.config` file used in the development environment).
- By adding custom rules to the MSBuild file used by the Web Deployment Project.

To deploy the web application build the Web Deployment Project and then copy the files from the project's output folder to the production environment.

To learn more about using the Web Deployment Project check out this Web Deployment Projects article from the April 2007 issue of MSDN Magazine, or consult the links in the Further Reading section at the end of this tutorial.

> **NOTE**
> You cannot use the Web Deployment Project with Visual Web Developer because the Web Deployment Project is implemented as a Visual Studio Add-In and the Visual Studio Express Editions (including Visual Web Developer) do not support Add-Ins.

Summary

The external resources and behavior of a web application in development are typically different than when the same application is in production. For instance, database connection strings, compilation options, and the behavior when an unhandled exception occurs commonly differ between environments. The deployment process must accommodate these differences. As we discussed in this tutorial, the simplest approach is to manually copy an alternate configuration file to the production environment. More elegant solutions are possible when using the Web Deployment Project Add-In or with a more formalized build or deployment process that can accommodate such customizations.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Connection Strings Explained
- Database Connection Strings @ ConnectionStrings.com
- Don't Run Production ASP.NET Applications with `debug="true"` Enabled
- Gracefully Responding to Unhandled Exceptions - Displaying User-Friendly Error Pages
- How Do I: Use a Visual Studio 2008 Web Deployment Project?
- Key Configuration Settings When Deploying a Database

- Visual Studio 2008 Web Deployment Projects Download | Visual Studio 2005 Web Deployment Projects Download
- VS 2008 Web Deployment Projects | VS 2008 Web Deployment Project Support Released
- Web Deployment Projects

PREVIOUS NEXT

Core Differences Between IIS and the ASP.NET Development Server (C#)

2/20/2020 • 12 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

When testing an ASP.NET application locally, chances are you are using the ASP.NET Development Web Server. However, the production website is most likely powered IIS. There are some differences between how these web servers handle requests, and these differences can have important consequences. This tutorial explores some of the more germane differences.

Introduction

Whenever a user visits an ASP.NET application his browser sends a request to the website. That request is picked up by the web server software, which coordinates with the ASP.NET runtime to generate and return the content for the requested resource. The Internet Information Services (IIS) are a suite of services that provide common Internet-based functionality for Windows servers. IIS is the most commonly used web server for ASP.NET applications in production environments; it's most likely the web server software being used by your web host provider to serve your ASP.NET application. IIS can also be used as the web server software in the development environment, although this entails installing IIS and properly configuring it.

The ASP.NET Development Server is an alternative web server option for the development environment; it ships with and is integrated into Visual Studio. Unless the web application has been configured to use IIS, the ASP.NET Development Server is automatically started and used as the web server the first time you visit a web page from within Visual Studio. The demo web applications we created back in the *Determining What Files Need to Be Deployed* tutorial were both file system-based web applications that were not configured to use IIS. Therefore, when visiting either of these websites from within Visual Studio the ASP.NET Development Server is used.

In a perfect world the development and production environments would be identical. However, as we discussed in the preceding tutorial it is not uncommon for the environments to have differing configuration settings. Using different web server software in the environments adds another variable that must be taken into consideration when deploying an application. This tutorial covers the key differences between IIS and the ASP.NET Development Server. Because of these differences there are scenarios where code that runs flawlessly in the development environment throws an exception or behaves differently when executed in production.

Security Context Differences

Whenever the web server software handles an incoming request it associates that request with a particular security context. This security context information is used by the operating system to determine what actions are permissible by the request. For example, an ASP.NET page might include code that logs some message to a file on disk. In order for this ASP.NET page to execute without error, the security context must have the appropriate file system-level permissions, namely write permissions on that file.

The ASP.NET Development Server associates incoming requests with the security context of the currently logged on user. If you are logged on to your desktop as an administrator, then the ASP.NET pages served by the ASP.NET Development Server will have the same access rights as an administrator. However, ASP.NET requests handled by IIS are associated with a specific machine account. By default, the Network Service machine account is used by IIS versions 6 and 7, although your web host provider may have configured a unique account for each customer.

What's more, your web host provider has likely given limited permissions to this machine account. The net result is that you may have web pages that execute without error in the development environment, but generate authorization-related exceptions when hosted in the production environment.

To show this type of error in action I've created a page in the Book Reviews website that creates a file on disk that stores the most recent date and time someone viewed the *Teach Yourself ASP.NET 3.5 in 24 Hours* review. To follow along, open the `~/Tech/TYASP35.aspx` page and add the following code to the `Page_Load` event handler:

```
protected void  Page_Load(object sender, EventArgs e)

{

    string filePath =  Server.MapPath("~/LastTYASP35Access.txt");

    string contents =  string.Format("Last accessed on {0} by {1}",

        DateTime.Now.ToString(), Request.UserHostAddress);

    System.IO.File.WriteAllText(filePath,  contents);

}
```

NOTE

The `File.WriteAllText` method creates a new file if it does not exist and then writes the specified contents to it. If the file already exists, it's existing content is overwritten.

Next, visit the *Teach Yourself ASP.NET 3.5 in 24 Hours* book review page in the development environment using the ASP.NET Development Server. Assuming that you are logged on to your computer with an account that has adequate permissions to create and modify a text file in the web application's root directory the book review appears the same as before, but each time the page is visited the date and time and user's IP address is stored in the `LastTYASP35Access.txt` file. Point your browser to this file; you should see a message similar to the one shown in Figure 1.

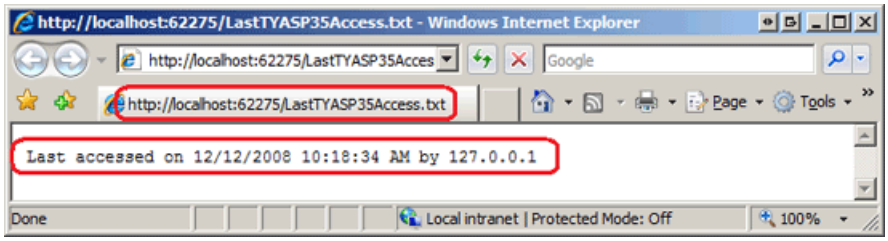


Figure 1: The Text File Contains the Last Date and Time the Book Review was Visited (Click to view full-size image)

Deploy the web application to production and then visit the hosted *Teach Yourself ASP.NET 3.5 in 24 Hours* book review page. At this point you should either see the book review page as normal or the error message shown in Figure 2. Some web host providers grant write permissions to the anonymous ASP.NET machine account, in which case the page will work without error. If, however, your web host provider prohibits write access for the anonymous account then an `UnauthorizedAccessException` exception is raised when the `TYASP35.aspx` page attempts to write the current date and time to the `LastTYASP35Access.txt` file.

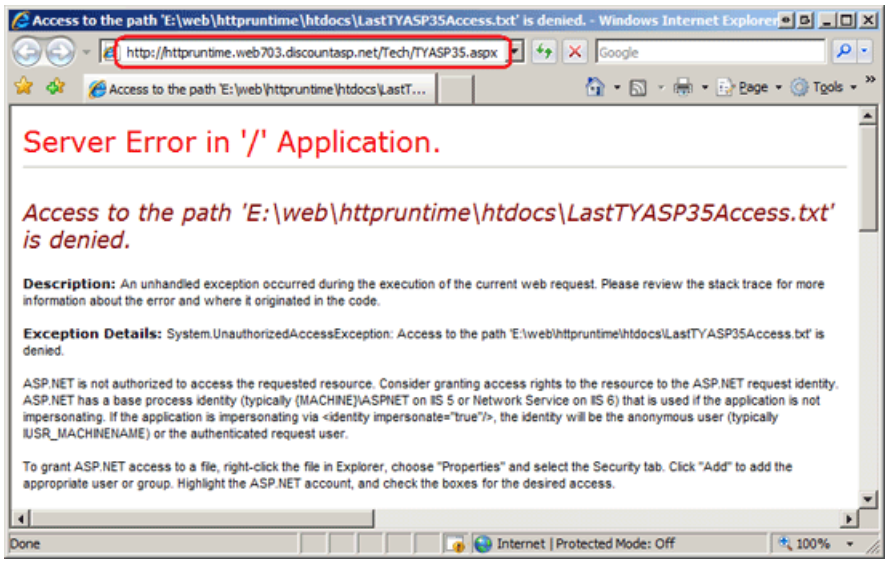


Figure 2: The Default Machine Account Used by IIS Does Not Have Permissions to Write to the File System (Click to view full-size image)

The good news is that most web host providers have some sort of permissions tool that allows you to specify file system permissions in your website. Grant the anonymous ASP.NET account write access to the root directory and then revisit the book review page. (If needed, contact your web host provider for assistance on how to grant write permissions to the default ASP.NET account.) This time the page should load without error and the `LastTYASP35Access.txt` file should be created successfully.

The take away here is that because the ASP.NET Development Server operates under a different security context than IIS, it is possible that your ASP.NET pages that read or write to the file system, read from or write to the Windows Event Log, or read or write to the Windows registry will work as expected on development but generate exceptions when on production. When building a web application that will be deployed to a shared web hosting environment, do not read or write to the Event Log or the Windows registry. Also take note of any ASP.NET pages that read from or write to the file system as you may need to grant read and write privileges on the appropriate folders in the production environment.

Differences On Serving Static Content

Another core difference between IIS and the ASP.NET Development Server is how they handle requests for static content. Every request that comes into the ASP.NET Development Server, whether for an ASP.NET page, an image, or a JavaScript file, is processed by the ASP.NET runtime. By default, IIS only invokes the ASP.NET runtime when a request comes in for an ASP.NET resource, such as an ASP.NET web page, a Web Service, and so forth. Requests for static content - images, CSS files, JavaScript files, PDF files, ZIP files, and the like - are retrieved by IIS without the involvement of the ASP.NET runtime. (It is possible to instruct IIS to work with the ASP.NET runtime when serving static content; consult the "Performing Forms-Based Authentication and URL Authentication on Static Files with IIS 7" section in this tutorial for more information.)

The ASP.NET runtime performs a number of steps to generate the requested content, including authentication (identifying the requestor) and authorization (determining if the requestor has permission to view the requested content). A popular form of authentication is *forms-based authentication*, in which a user is identified by entering their credentials - usually a username and password - into textboxes on a web page. Upon validating their credentials, the website stores an *authentication ticket* cookie on the user's browser, which is sent with every subsequent request to the website and is what is used to authenticate the user. Moreover, it is possible to specify *URL authorization* rules that dictate what users can or cannot access a particular folder. Many ASP.NET websites use forms-based authentication and URL authorization to support user accounts and to define portions of the site that are only accessible to authenticated users or users that belong to a certain role.

NOTE

For a thorough examination of ASP.NET's forms-based authentication, URL authorization, and other user account-related features, be sure to check out my Website Security Tutorials.

Consider a website that supports user accounts using forms-based authorization and has a folder that, using URL authorization, is configured to only allow authenticated users. Imagine that this folder contains ASP.NET pages and PDF files and that the intent is that only authenticated users can view these PDF files.

What happens if a visitor attempts to view one of these PDF files by entering the URL directly in his browser's Address bar? To find out, let's create a new folder in the Book Reviews site, add some PDF files, and configure the site to use URL authorization to prohibit anonymous users from visiting this folder. If you download the demo application you'll see that I created a folder called `PrivateDocs` and added a PDF from my Website Security Tutorials (how fitting!). The `PrivateDocs` folder also contains a `Web.config` file that specifies the URL authorization rules to deny anonymous users:

```
<?xml version="1.0"?>
<?xml version="1.0"?>

<configuration>

    <system.web>

         <authorization>

            <deny  users="?" />

         </authorization>

     </system.web>

</configuration>
```

Finally, I configured the web application to use forms-based authentication by updating the `Web.config` file in the root directory, replacing:

```
<authentication mode="Windows" />
```

With:

```
<authentication mode="Forms" />
```

Using the ASP.NET Development Server, visit the site and enter the direct URL to one of the PDF files in your browser's Address bar. If you downloaded the website associated with this tutorial the URL should look something like: `http://localhost:portNumber/PrivateDocs/aspnet_tutorial01_Basics_vb.pdf`

Entering this URL into the Address bar causes the browser to send a request to the ASP.NET Development Server for the file. The ASP.NET Development Server hands off the request to the ASP.NET runtime for processing. Because we have not yet logged in, and because the `Web.config` in the `PrivateDocs` folder is configured to deny anonymous access, the ASP.NET runtime automatically redirects us to the login page, `Login.aspx` (see Figure 3). When redirecting the user to the log in page, ASP.NET includes a `ReturnUrl` querystring parameter that indicates the page the user was attempting to view. After successfully logging in the user can be returned to this page.

Figure 3: Unauthorized Users are Automatically Redirected to the Login Page (Click to view full-size image)

Now let's see how this behaves on production. Deploy your application and enter the direct URL to one of the PDFs in the `PrivateDocs` folder in production. This prompts your browser to send a request IIS for the file. Because a static file is requested, IIS retrieves and returns the file without invoking the ASP.NET runtime. As a result, there was no URL authorization check performed; the contents of the supposedly private PDF are accessible to anyone who knows the direct URL to the file.

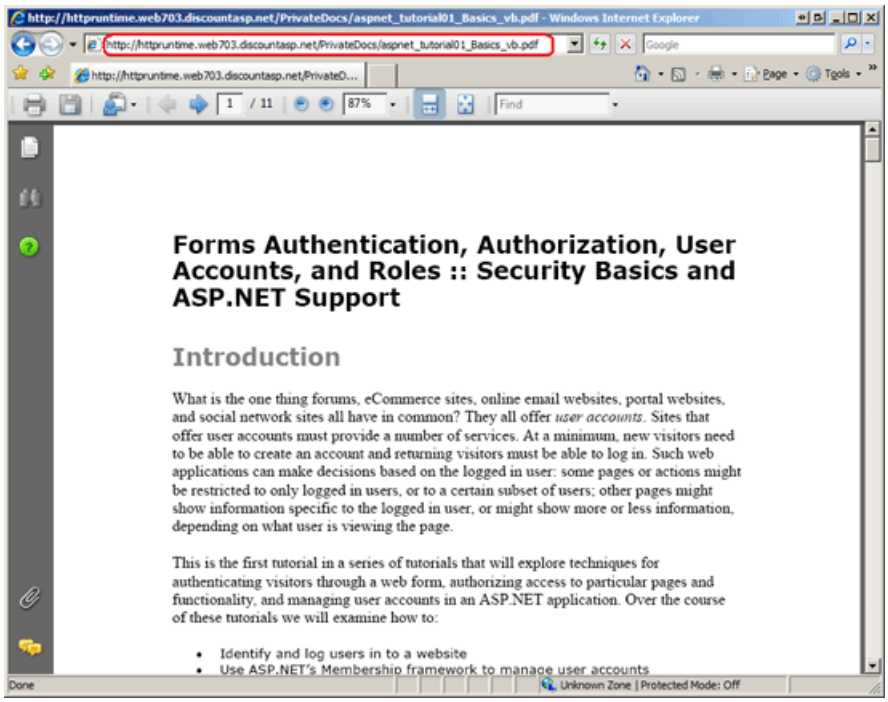


Figure 4: Anonymous Users Can Download the Private PDF Files By Entering the Direct URL to the File (Click to view full-size image)

Performing Forms-Based Authentication and URL Authentication on Static Files with IIS 7

There are a couple of techniques you can use to protect static content from unauthorized users. IIS 7 introduced the *integrated pipeline*, which marries IIS's workflow with the ASP.NET runtime's workflow. In a nutshell, you can instruct IIS to invoke the ASP.NET runtime's authentication and authorization modules all incoming requests (including static content like PDF files). Contact your web host provider to find out how to configure your website to use the integrated pipeline.

Once IIS has been configured to use the integrated pipeline add the following markup to the `Web.config` file in the root directory:

```
<system.webServer>

      <modules>

          <add  name="FormsAuthenticationModule"  type="System.Web.Security.FormsAuthenticationModule" />

          <remove  name="UrlAuthorization" />

          <add  name="UrlAuthorization" type="System.Web.Security.UrlAuthorizationModule"  />

          <remove  name="DefaultAuthentication" />

          <add  name="DefaultAuthentication"  type="System.Web.Security.DefaultAuthenticationModule" />

      </modules>

</system.webServer>
```

This markup instructs IIS 7 to use the ASP.NET-based authentication and authorization modules. Re-deploy your application and then re-visit the PDF file. This time when IIS handles the request it gives the ASP.NET runtime's authentication and authorization logic an opportunity to inspect the request. Because only authenticated users are authorized to view the contents in the `PrivateDocs` folder, the anonymous visitor is automatically redirected to the login page (refer back to Figure 3).

NOTE

If your web host provider is still using IIS 6 then you cannot use the integrated pipeline feature. One workaround is to put your private documents in a folder that prohibits HTTP access (such as `App_Data`) and then create a page to serve these documents. This page might be called `GetPDF.aspx`, and is passed the name of the PDF through a querystring parameter. The `GetPDF.aspx` page would first verify that the user has permission to view the file and, if so, would use the `Response.WriteFile(filePath)` method to send the contents of the requested PDF file back to the requesting client. This technique would also work for IIS 7 if you did not wish to enable the integrated pipeline.

Summary

Web applications in a production environment are hosted using Microsoft's IIS web server software. In the development environment, however, the application may be hosted using IIS or the ASP.NET Development Server. Ideally, the same web server software should be used in both environments because using different software adds another variable in the mix. However, the ease of use of the ASP.NET Development Server makes it an attractive choice in the development environment. The good news is that there are only a few fundamental differences between IIS and the ASP.NET Development Server, and if you are aware of these differences you can take steps to help ensure that the application works and functions the same way regardless of the environment.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Integration with IIS 7.0
- Using ASP.NET Forums Authentication With All Types of Content on IIS 7 (Video)
- Web Servers in Visual Web Developer

Deploying a Database (C#)

2/20/2020 • 14 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Deploying an ASP.NET web application entails getting the necessary files and resources from the development environment to the production environment. For data-driven web applications this includes the database schema and data. This tutorial is the first in a series that explores the steps needed to successfully deploy the database from the development environment to production.

Introduction

Deploying an ASP.NET web application entails getting the necessary files and resources from the development environment to the production environment. Over the course of the past six tutorials we looked at deploying a simple Book Reviews web application. This demo site was comprised of a number of server-side resources - ASP.NET pages, configuration files, a `Web.sitemap` file, and so forth - along with client-side resources, like images and CSS files. But what about data-driven web applications? What extra steps must be taken to deploy a web application that uses a database?

Over the next several tutorials we will address the steps needed to deploy a data-driven web application. This tutorial starts by examining how to get a database s schema and contents from the development environment to the production environment, while the subsequent tutorial looks at the needed configuration changes. Following that we'll explore challenges of deploying a database that uses the Application Services (Membership, Roles, Profile, and so on).

Examining the Updated Book Reviews Web Application

In order to demonstrate deploying a data-driven web application, I ve updated the Book Reviews web application from a simple, static website to a data-driven one. As before, there are two versions of the application in this tutorial s download: one that uses the Web Application Project model and one that uses the Web Site Project model.

The updated Book Reviews web application uses a SQL Server 2008 Express Edition database, which is stored in the site s `App_Data` folder (`~/App_Data/Reviews.mdf`). If you have SQL Server 2008 installed on your computer then the demo should run without error. If you have an older version of SQL Server you can either install the free SQL Server 2008 Express Edition or you can use the database scripts available in this tutorial s download to create the database yourself.

The `Reviews.mdf` database contains four tables:

- `Genres` - includes a record for each genre, such as Technology, Fiction, and Business.
- `Books` - includes a record for each review, with columns like `Title` , `GenreId` , `ReviewDate` , and `Review` , among others.
- `Authors` - includes information about each author who has contributed to a reviewed book.
- `BooksAuthors` - a many-to-many join table that specifies what authors have written what books.

Figure 1 shows an ER diagram of these four tables.

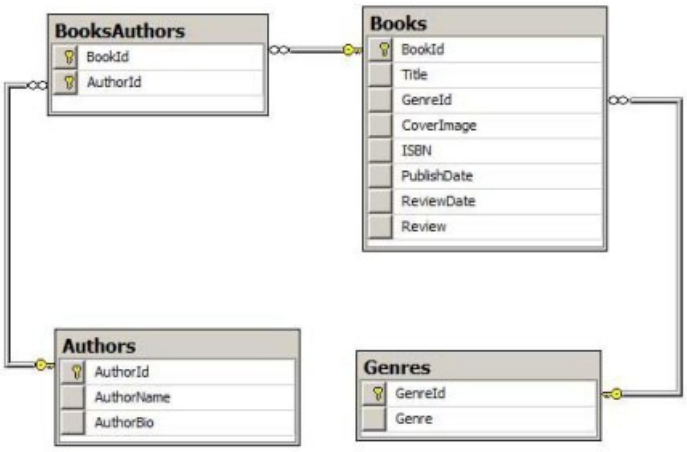


Figure 1: The Book Reviews Web Application s Database is Comprised of Four Tables (Click to view full-size image)

The previous version of the Book Reviews website had a separate ASP.NET page for each book. For example, there was a page named `~/Tech/TYASP35.aspx` that contained the review for *Teach Yourself ASP.NET 3.5 in 24 Hours*. This new data-driven version of the website has the reviews stored in the database and a single ASP.NET page, Review.aspx?ID=*bookId*, which displays the review for the specified book. Likewise, there is a Genre.aspx?ID=*genreId* page that lists the reviewed books in the specified genre.

Figures 2 and 3 show the `Genre.aspx` and `Review.aspx` pages in action. Note the URL in the Address bar for each page. In Figure 2 it s Genre.aspx?ID=85d164ba-1123-4c47-82a0-c8ec75de7e0e. Because 85d164ba-1123-4c47-82a0-c8ec75de7e0e is the `GenreId` value for the Technology genre, the page s heading reads "Technology Reviews" and the bulleted list enumerates those reviews on the site that fall under this genre.

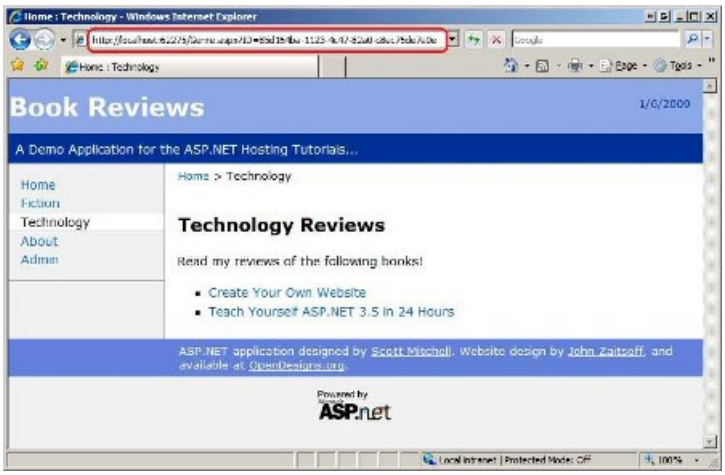


Figure 2: The Technology Genre Page (Click to view full-size image)

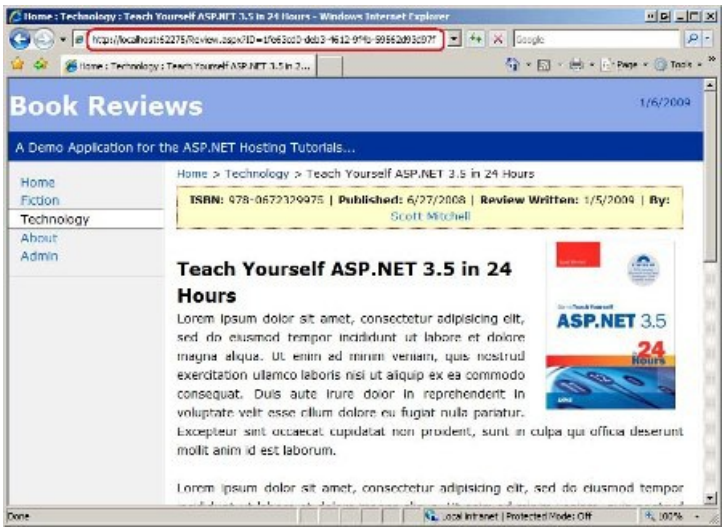


Figure 3: The Review for *Teach Yourself ASP.NET 3.5 in 24 Hours* (Click to view full-size image)

The Book Reviews web application also includes an administration section where administrators can add, edit, and delete genres, reviews, and author information. Currently, any visitor can access the administration section. In a future tutorial we'll add support for user accounts and only permit authorized users into the administration pages.

If you download the Book Reviews application please keep in mind that its purpose is to demonstrate deploying a data-driven application. It does not exhibit best practices as far as application design. For example, there is no separate Data Access Layer (DAL); the ASP.NET pages communicate directly with the database through the SqlDataSource control or ADO.NET code in their code-behind classes. For a more in-depth look at creating data-driven applications using a tiered architecture, refer to my *Working with Data* tutorials.

Databases on Development Versus Production

When you start development on a data-driven web application you must specify a database connection string, which provides the application details on how to connect to the database. This connection string specifies, among other things, the database server, the database name, and security information. Most often, the database used by the application during development is different than the database used when it s in production. There are many benefits of using different databases for development versus production. Having a different database in development means you don t have to worry about accidentally modifying or deleting live data. It also lets you put in dummy test data or make breaking changes to the data model without having to worry about the effects on the application in production. The downside of having a different database in the development and production environments is that when the application is deployed the database and any pertinent changes to the database s schema or data must also be deployed.

Prior to the first deployment, there is only one instance of the database, and that instance is in the development environment. When deploying the application to production for the first time we must not only copy up the necessary server-side and client-side files, but also copy the database from the development environment to the production environment. This is where we stand right now with the Book Reviews web application - the database resides in the `App_Data` folder in our development environment but has not yet been pushed up to the production environment.

Once the application has been deployed there are two copies of the database. As the application matures, new features may be added, necessitating a change to the data model (such as adding new columns to existing tables, making changes to existing columns, adding new tables, and so on). When the web application is next deployed, the changes applied to the database in the development environment since the last deployment must be applied to the production database. Some strategies for managing this process are discussed in a future tutorial. This tutorial focuses on deploying the entire database from the development environment to production.

Deploying the Database to the Production Environment

The remainder of this tutorial looks at how to deploy the database from the development environment to the production environment. If you are following along you need to make sure that your account with your web host provider includes Microsoft SQL Server database support. You'll also need to have some information at hand, namely the database server name, the database name, and the username and password used to connect to the database.

As noted earlier in this tutorial, the Book Reviews website s database is a SQL Server 2008 Express Edition database stored in the `App_Data` folder. It would stand to reason that deploying such a database would be as simple as copying the `App_Data` folder from the development environment to the production environment. However, most web host providers do not support hosting databases in the `App_Data` folder because of security reasons. Instead, web hosts provide an account on a SQL Server database server within their environment. Deploying the database from your development environment to the production environment requires getting your database registered on your web host s database server.

So how do you get your database from the development environment to the production environment? There are a couple ways to accomplish this depending on what services your web host offers. With some hosts, such as DiscountASP.NET, you can FTP a backup of the database or the actual `.mdf` file to your website and then, from the Control Panel, restore the backup file or attach the `.mdf` file to the SQL Server database server. With such tools deploying the database is as simple as copying the `App_Data` folder to the production environment and then attaching it via the Control Panel. This is perhaps the easiest and quickest way to publish your database for the first time.

Another approach is to use the Database Publishing Wizard. The Database Publishing Wizard is a Windows desktop application that will generate the SQL commands to create your database s schema - the tables, stored procedures, views, user-defined functions, and so forth - and, optionally, the data in its tables. You can then connect to your web host provider s database server through SQL Server Management Studio and then execute this script to duplicate the database on production. Even better, if your web host provider supports Microsoft s Database Publishing Services you can have the script generated by the Database Publishing Wizard automatically executed on the database server on your behalf. Because the Database Publishing Wizard generates a script that creates the database s schema and data, it will work regardless of whether your web host provider offers features like attaching an uploaded `.mdf` file.

Generating the SQL Commands to Create the Database Schema and Data Using the Database Publishing Wizard

Let s walk through using the Database Publishing Wizard to deploy the Book Reviews database to production. If you are using Visual Studio 2008 or beyond, the Database Publishing Wizard is already installed. If you are using Visual Studio 2005 then you will need to first download and install the wizard.

Open Visual Studio and navigate to the `Reviews.mdf` database. If you are using Visual Web Developer, go to the Database Explorer; if you are using Visual Studio, use the Server Explorer. Figure 4 shows the `Reviews.mdf` database in the Database Explorer in Visual Web Developer. As Figure 4 shows, the `Reviews.mdf` database is composed of four tables, three stored procedures, and a user-defined function.

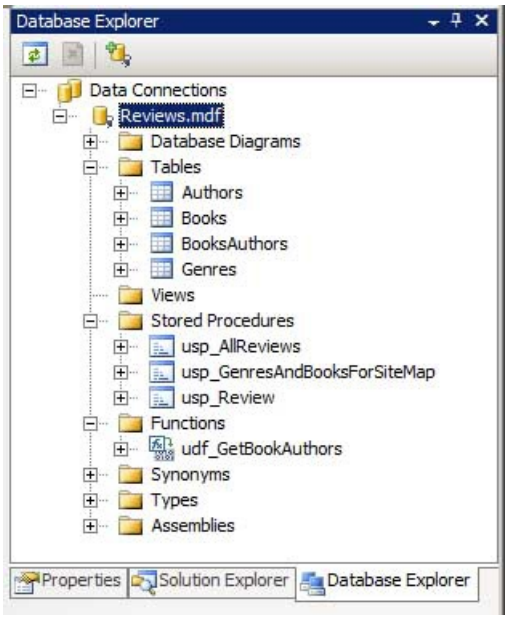


Figure 4: Locate the Database in the Database Explorer or Server Explorer (Click to view full-size image)

Right-click on the database name and choose the "Publish to provider" option from the context menu. This launches the Database Publishing Wizard (see Figure 5). Click Next to advance past the splash screen.

Figure 5: The Database Publishing Wizard Splash Screen (Click to view full-size image)

The second screen in the wizard lists the databases accessible to the Database Publishing Wizard and lets you choose whether to script all objects in the selected database or to pick which objects to script. Select the appropriate database and leave the "Script all objects in the selected database" option checked.

> **NOTE**
>
> If you get the error "There are no objects in database *databaseName* of the types scriptable by this wizard" when clicking Next in the screen shown in Figure 6, make sure that the path to your database file is not overly long. It has been discovered that this error can arise if the path to the database file is too long.

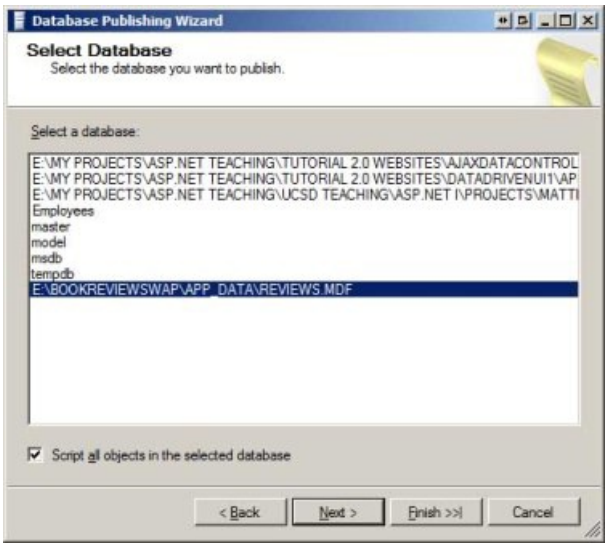


Figure 6: The Database Publishing Wizard Splash Screen (Click to view full-size image)

From the next screen you can generate a script file or, if your web host supports it, publish the database directly to your web host provider s database server. As Figure 7 shows, I am having the script written to the file `C:\REVIEWS.MDF.sql`.

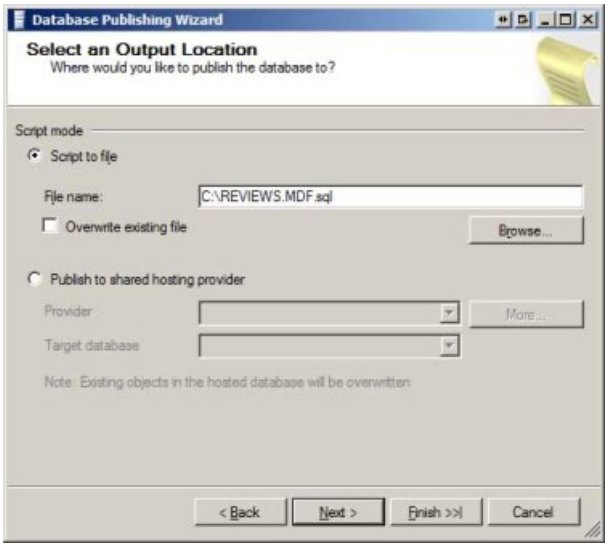


Figure 7: Script the Database to a File or Publish it Directly to Your Web Host Provider (Click to view full-size image)

The subsequent screen prompts you for a variety of scripting options. You can specify whether the script should include drop statements to remove these existing objects. This defaults to True, which is fine when deploying a database for the first time. You can also specify whether the target database is SQL Server 2000, SQL Server 2005, or SQL Server 2008. Finally, you can indicate whether to script the schema and data, just the data, or just the schema. The schema is the collection of database objects, the tables, stored procedures, views, and so on. The data is the information residing in the tables.

As Figure 8 illustrates, I ve got the wizard configured to drop existing database objects, to generate script for a SQL Server 2008 database, and to publish both the schema and data.

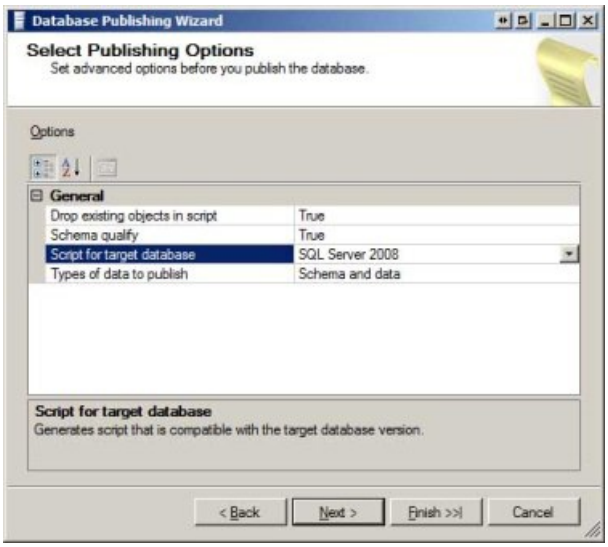


Figure 8: Specify the Publishing Options (Click to view full-size image)

The final two screens summarize the actions that are about to be taken and then display the status of the scripting. The net result of running the wizard is that we have a script file that contains the SQL commands needed to create the database on production and populate it with the same data as on development.

Executing the SQL Commands On the Production Environment Database

Now that we have the script that contains the SQL commands to create the database and its data all that remains is to execute the script on the production database. Some web host providers offer a textbox in their Control Panel where you can enter SQL commands to execute on your database. If you have a very large script file then this option might not work (the `REVIEWS.MDF.sql` script file is over 425 KB in size, for instance).

A better approach is to connect directly to the production database server using SQL Server Management Studio (SSMS). If you have a non-Express Edition of SQL Server installed on your computer then you likely already have SSMS installed. Otherwise, you can download and install a free copy of SQL Server Management Studio Express Edition.

Launch SSMS and connect to your web host s database server using the information provided by your web host provider.

Figure 9: Connect to Your Web Host Provider s Database Server (Click to view full-size image)

Expand the Databases tab and locate your database. Click the New Query button in the upper left corner of the Toolbar, paste in the SQL commands from the script file created by the Database Publishing Wizard, and click the Execute button to run these commands on the production database server. If your script file is especially large it may take several minutes to execute the commands.

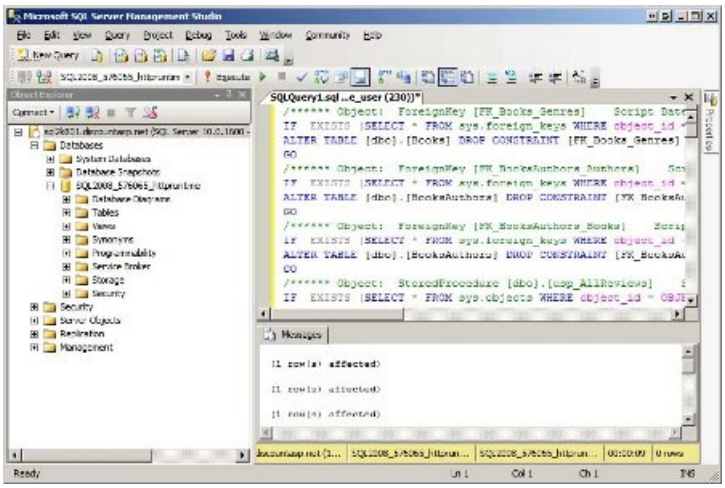


Figure 10: Connect to Your Web Host Provider s Database Server (Click to view full-size image)

That s all there is to it! At this point the development database has been duplicated to production. If you Refresh the database in SSMS you should see the new database objects. Figure 11 shows the production database s tables, stored procedures, and user-defined functions, which mirror those on the development database. And because we instructed the Database Publishing Wizard to publish the data, the production database s tables have the same data as the development database s tables at the time the wizard was executed. Figure 12 shows the data in the `Books` table on the production database.

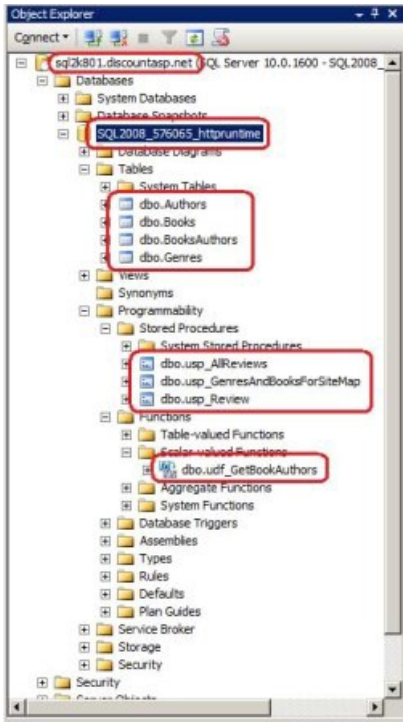


Figure 11: The Database Objects Have Been Duplicated on the Production Database (Click to view full-size image)

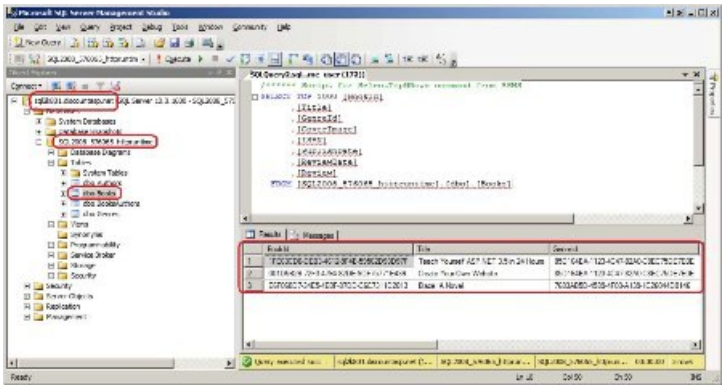


Figure 12: The Production Database Contains the Same Data as on the Development Database (Click to view full-size image)

At this point we have only deployed the development database to production. We have not yet looked at deploying the web application itself or examined what configuration changes are needed to have the application on production use the production database. We'll cover these issues in the next tutorial!

Summary

Deploying a data-driven web application requires copying the database used during development to the production environment. Many web host providers offer tools to simplify the process of deploying a database. For example, with DiscountASP.NET you can FTP your database `.mdf` file (or a backup) and then attach the database to the database server from the Control Panel. Another option that works regardless of what features your web host provider offers is Microsoft s Database Publishing Wizard tool, which generates a script of SQL commands to create the development database s schema and data. Once this script has been generated you can execute it on the production database.

Now that the Book Reviews web application s database is on production we can deploy the application. However, the web application s configuration information specifies the connection string to the database, and that connection string references the development database. We need to update this connection string information when deploying the site to production. The next tutorial looks at these configuration differences and walks through the steps needed to publish the data-driven Book Reviews site to production.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Download the Microsoft SQL Server Database Publishing Wizard 1.1
- Download the Microsoft SQL Server Management Studio Express Edition

PREVIOUS NEXT

Configuring the Production Web Application to Use the Production Database (C#)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

As discussed in earlier tutorials, it is not uncommon for configuration information to differ between the development and production environments. This is especially true for data-driven web applications, as the database connection strings differ between the development and production environments. This tutorial explores ways to configure the production environment to include the appropriate connection string in more detail.

Introduction

Data-driven web applications usually use a different database when in development than when in production. For applications hosted by a web host provider and developed locally, the development database typically resides on the developer s computer while the production database is hosted on a database server at the web hosting company s facility. Deploying a data-driven web application entails copying the development database to the production database server. In the previous tutorial we looked at ways to accomplish this step.

The web application uses the information in a *connection string* to establish a connection with the database. The connection string, which is typically stored in `Web.config`, specifies the database server name, the name of the database, the security context, and other information. Because the database used by the web application depends on whether the web application is running in the development or production environments, the connection strings must differ between the two environments.

It is not uncommon for configuration information to differ between the development and production environments. The *Common Configuration Differences Between Development and Production* tutorial discussed techniques for maintaining separate configuration information between these two environments, as well as a brief discussion on database connection strings. This tutorial explores ways to configure the production environment to include the appropriate connection string in more detail.

Examining the Connection String Information

The connection string used by the Book Reviews web application is stored in the application s configuration file, `Web.config`. `Web.config` includes a special section for storing connection strings, aptly named <connectionStrings>. The `Web.config` file for the Book Reviews website has one connection string defined in this section named `ReviewsConnectionString`:

```
<configuration>
    ...
    <connectionStrings>

        <add name="ReviewsConnectionString"
            connectionString="Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\Reviews.mdf;Integrated
            Security=True;User Instance=True"
            providerName="System.Data.SqlClient" />

    </connectionStrings>
    ...
</configuration>
```

The connection string - Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\Reviews.mdf;Integrated Security=True;User Instance=True - is composed of a number of options and values, with option/value pairs delimited by a semicolon and each option and value delimited by an equals sign. The four options used in this connection string are:

- `Data Source` - specifies the location of the database server and the database server instance name (if any). The value, `.\SQLEXPRESS`, is an example where there is a database server and an instance name. The period specifies that the database server is on the same computer as the application; the instance name is `SQLEXPRESS` .
- `AttachDbFilename` - specifies the location of the database file. The value contains the placeholder `|DataDirectory|`, which is resolved to the full path of the application s `App_Data` folder at runtime.
- `Integrated Security` - a Boolean value that indicates whether to use a specified username/password when connecting to the database (false) or the current Windows account credentials (true).
- `User Instance` - a configuration option specific to the SQL Server Express Editions that indicates whether to allow non-Administrative users on the local computer attach and connect to a SQL Server Express Edition database. See SQL Server Express User Instances for more information on this setting.

The allowable connection string options depend on the database you are connecting to and the ADO.NET database provider being used. For example, the connection string for connecting to a Microsoft SQL Server database differs from that used to connect to an Oracle database. Likewise, connecting to a Microsoft SQL Server database using the SqlClient provider uses a different connection string than when using the OLE-DB provider.

You can build the database connection string by hand using a site like ConnectionStrings.com as a resource for what options are available. However, an easier approach is to add the database to the Server Explorer in Visual Studio and then to grab the connection string from the Properties window. Let s use this latter technique for constructing the connection string to the production database server.

Open Visual Studio and then navigate to the Server Explorer window (in Visual Web Developer, this window is called Database Explorer). Right-click on the Data Connections option and choose the Add Connection option from the context menu. This brings up the wizard shown in Figure 1. Choose the appropriate data source and click Continue.

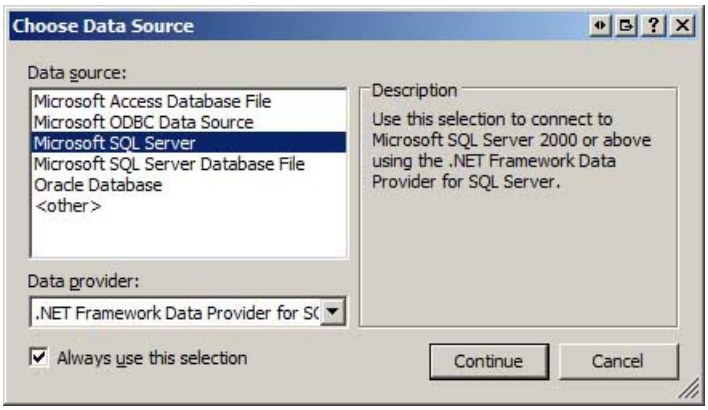


Figure 1: Choose to Add a New Database to the Server Explorer (Click to view full-size image)

Next, specify the various database connection information (see Figure 2). When you signed up with your web hosting company they should have provided information on how to connect to the database - the database server name, the database name, the username and password to use to connect to the database, and so on. After entering this information click OK to complete this wizard and to add the database to the Server Explorer.

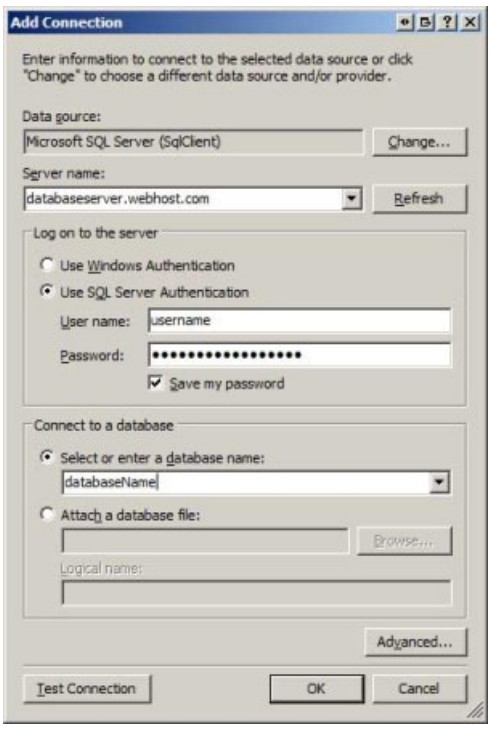


Figure 2: Specify the Database Connection Information (Click to view full-size image)

The production environment database should now be listed in the Server Explorer. Select the database from the Server Explorer and go to the Properties window. There you will find a property named Connection String with the database s connection string. Assuming you are using a Microsoft SQL Server database on production and the SqlClient provider your connection string should look similar to the following:

Data Source=*serverName*; Initial Catalog=*databaseName*; Persist Security Info=True; User ID=*username*; Password=*password*

Where *serverName*, *databaseName*, *username*, and *password* are with the values for the database server name, the database name, and the username and password supplied to you by your web host company.

Deploying the Book Reviews Web Application

The preceding tutorial stepped through copying the development database to the production environment, but did not explore deploying the data-driven application. At this point the production environment contains the database but is using the version of the Book Reviews application with static reviews. We need to deploy the new,

data-driven application to the production server along with the updated configuration information.

Take a moment to deploy the data-driven application from the development environment to production. This process was covered in detail in previous tutorials. If you need a refresher, see the *Deploying Your Website Using an FTP Client* or the *Deploying Your Website Using Visual Studio* tutorials. You will need to ensure that the production database connection string is the one used in the production environment, which means that an alternate `Web.config` file must be deployed. Specifically, this modified `Web.config` file s `<connectionStrings>` element needs to contain the production database connection string and should look similar to the following:

```
<configuration>

    ...
    <connectionStrings>

        <add name="ReviewsConnectionString"
            connectionString="Data Source=serverName; Initial Catalog=databaseName;
            Persist Security Info=True; User ID=username; Password=password"
            providerName="System.Data.SqlClient" />

    </connectionStrings>
    ...
</configuration>
```

Note that the connection string in the `<connectionStrings>` element is named the same (`ReviewsConnectionString`), but now contains the production database connection string instead of the development database connection string.

Unless you have a more formalized deployment workflow, either manually modify the `Web.config` file to use the production database connection string before deploying (remembering to revert it back to using the development database connection string afterwards) or maintain a separate `Web.config` file with the production environment configuration information that gets uploaded to the production environment as part of the deployment process.

> **NOTE**
> If you accidentally deploy a `Web.config` file that contains the development database connection string then there will be an error when the application on production attempts to connect to the database. This error manifests as a `SqlException` with a message reporting that the server was not found or was not accessible.

Once the site has been deployed to production, visit the production site through your browser. You should see and enjoy the same user experience as when running the data-driven application locally. Of course when you visit the website on production the site is powered by the production database server, whereas visiting the website in the development environment uses the database in development. Figure 3 shows the *Teach Yourself ASP.NET 3.5 in 24 Hours* review page from the website in the production environment (note the URL in the browser s Address bar).

Figure 3: The Data-Driven Application is Now Available On Production! (Click to view full-size image)

Storing Connection Strings in a Separate Configuration File

A common technique for maintaining separate configuration information on the development and production environments is to have two versions of the `Web.config`: one for the development environment and one for production. At deploy-time the appropriate `Web.config` version can be copied to the production environment. Ideally, this process would be automated as part of the deployment workflow.

Instead of maintaining two separate `Web.config` files you can, optionally, provide more granular differences. The elements that make up the `Web.config` file can be defined in an external configuration files that are then referenced in the `Web.config` file. In a nutshell you can have one `Web.config` file for both environments that references a databaseConnectionStrings.config file, which would contain the connection strings used by the application and would be unique for each environment. I find that separating the differing configuration information into separate files provides a tidier and simpler `Web.config` file and more clearly outlines the configuration differences between the development and production environments.

To use this technique, start by creating a new folder in the web application named `ConfigSections`. Next, add two files to this new folder named databaseConnectionStrings.dev.config and databaseConnectionStrings.production.config. Next, copy the `<connectionStrings>` element from `Web.config` into the databaseConnectionStrings.dev.config and databaseConnectionStrings.production.config files, and then modify the connection string in the databaseConnectionStrings.production.config file so that it specifies the production database connection string. For example, the databaseConnectionStrings.dev.config file should contain just the `<connectionStrings>` element with a connection string that references the development database:

```
<?xml version="1.0"?>
<connectionStrings>
    <add name="ReviewsConnectionString"
        connectionString="Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\Reviews.mdf;Integrated
        Security=True;User Instance=True"
        providerName="System.Data.SqlClient" />

</connectionStrings>
```

Similarly, the databaseConnectionStrings.production.config file should contain just a `<connectionStrings>` element, but one that has the production database connection string.

Make a copy of the databaseConnectionStrings.dev.config file and name it databaseConnectionStrings.config.

NOTE

You can name the configuration file something other than databaseConnectionStrings.config, if you d like, such as `connectionStrings.config` or `dbInfo.config`. However, be sure to name the file with a `.config` extension as `.config` files are, by default, not served by the ASP.NET engine. If you name the file something else, like `connectionStrings.txt`, a user could point their browser to www.yoursite.com/ConfigSettings/connectionStrings.txt and view the contents of the file!

At this point the `ConfigSections` folder should contain three files (see Figure 4). The databaseConnectionStrings.dev.config and databaseConnectionStrings.production.config files contain the connection strings for the development and production environments, respectively. The databaseConnectionStrings.config file contains the connection string information that will be used by the web application at runtime. Consequently, the databaseConnectionStrings.config file should be identical to the databaseConnectionStrings.dev.config file in the development environment, whereas on production the databaseConnectionStrings.config file should be identical to databaseConnectionStrings.production.config.

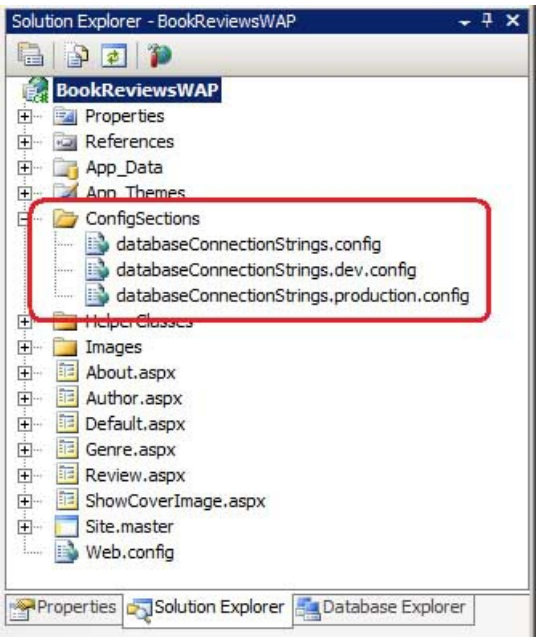


Figure 4: ConfigSections (Click to view full-size image)

We now need to instruct `Web.config` to use the databaseConnectionStrings.config file for its connection string store. Open `Web.config` and replace the existing `<connectionStrings>` element with the following:

```
<configuration>
    ...
    <connectionStrings
        configSource="ConfigSections\databaseConnectionStrings.config"
    />
    ...
</configuration>
```

The `configSource` attribute specifies a physical path relative to the `Web.config` file. If the external `.config` file is in the same directory as `Web.config` then set this attribute to the file name of the `.config` file. If it s in a subdirectory, as is the case with databaseConnectionStrings.config, specify the subfolder using a backslash to delimit the folder and file names, like ConfigSections\databaseConnectionStrings.config.

With this modification, the development and production environments contain the same `Web.config` file. Now the only difference is the databaseConnectionStrings.config file. Copy the databaseConnectionStrings.production.config file to production and rename it to

databaseConnectionStrings.config. If, in the future, there are changes to the production database connection string you will need to make them to the databaseConnectionStrings.production.config file and then upload that file to production, renaming it databaseConnectionStrings.config.

> **NOTE**
> You may specify the information for any `Web.config` element in a separate file and use the `configSource` attribute to reference that file from within `Web.config`.

Summary

Data-driven applications usually use different databases in the development and production environments. Consequently, the database connection strings stored in the web application s configuration must be unique per environment. In this tutorial we looked at how to determine the production database connection string and ways to maintain unique connection string information in the two environments.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Connection Strings and Configuration Files
- Database Configuration Strings Information @ ConnectionStrings.com
- Move Settings Out of the Web.config File
- Technical Documentation for the <connectionStrings> Element

PREVIOUS NEXT

Configuring a Website that Uses Application Services (C#)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

ASP.NET version 2.0 introduced a series of application services, which are part of the .NET Framework and serve as a suite of building block services that you can use to add rich functionality to your web application. This tutorial explores how to configure a website in the production environment to use application services and addresses common issues with managing user accounts and roles on the production environment.

Introduction

ASP.NET version 2.0 introduced a series of *application services*, which are part of the .NET Framework and serve as a suite of building block services that you can use to add rich functionality to your web application. The application services include:

- **Membership** - an API for creating and managing user accounts.
- **Roles** - an API for categorizing users into groups.
- **Profile** - an API for storing custom, user-specific content.
- **Site Map** - an API for defining a site s logical structure in the form of a hierarchy, which can then be displayed via navigation controls, such as menus and breadcrumbs.
- **Personalization** - an API for maintaining customization preferences, most often used with *WebParts*.
- **Health Monitoring** - an API for monitoring performance, security, errors, and other system health metrics for a running web application.

The application services APIs are not tied to a specific implementation. Instead, you instruct the application services to use a particular *provider*, and that provider implements the service using a particular technology. The most commonly used providers for Internet-based web applications hosted at a web hosting company are those providers that use a SQL Server database implementation. For example, the `SqlMembershipProvider` is a provider for the Membership API that stores user account information in a Microsoft SQL Server database.

Using the application services and SQL Server providers adds some challenges when deploying the application. For starters, the application services database objects must be properly created on both the development and production databases and appropriately initialized. There are also important configuration settings that need to be made.

> **NOTE**
>
> The application services APIs were designed using the *provider model*, a design pattern that allows for an API s implementation details to be provided at runtime. The .NET Framework ships with a number of application service providers that can be used, such as the `SqlMembershipProvider` and `SqlRoleProvider`, which are providers for the Membership and Roles APIs that use a SQL Server database implementation. You can also create and plug-in a custom provider. In fact, the Book Reviews web application already contains a custom provider for the Site Map API (`ReviewSiteMapProvider`), which constructs the site map from the data in the `Genres` and `Books` tables in the database.

This tutorial starts with a look at how I extended the Book Reviews web application to use the Membership and Roles APIs. It then walks through deploying a web application that uses application services with a SQL Server database implementation, and concludes by addressing common issues with managing user accounts and roles on the production environment.

Updates to the Book Reviews Application

Over the past couple tutorials the Book Reviews web application was updated from a static website to a dynamic, data-driven web application complete with a set of administration pages for managing genres and reviews. However, this administration section is currently not protected - any user who knows (or guesses) the administration page URL can waltz in and create, edit, or delete reviews on our site. A common way to protect certain portions of a website is to implement user accounts and then use URL authorization rules to restrict access to certain users or roles. The Book Reviews web application available for download with this tutorial supports user accounts and roles. It has a single role defined named Admin and only users in this role can access the administration pages.

> **NOTE**
> I ve created three user accounts in the Book Reviews web application: Scott, Jisun, and Alice. All three users have the same password: **password!** Scott and Jisun are in the Admin role, Alice is not. The site s non-administration pages are still accessible to anonymous users. That is, you do not need to sign in to visit the site, unless you want to administer it, in which case you must sign in as a user in the Admin role.

The Book Reviews application s master page has been updated to include a different user interface for authenticated and anonymous users. If an anonymous user visits the site she sees a Login link in the upper right corner. An authenticated user sees the message, "Welcome back, *username*!" and a link to log out. There s also a login page (`~/Login.aspx`), which contains a Login Web control that provides the user interface and logic for authenticating a visitor. Only administrators can create new accounts. (There are pages for creating and managing user accounts in the `~/Admin` folder.)

Configuring the Membership and Roles APIs

The Book Reviews web application uses the Membership and Roles APIs to support user accounts and to group those users into roles (namely, the Admin role). The `SqlMembershipProvider` and `SqlRoleProvider` provider classes are used because we want to store account and role information in a SQL Server database.

> **NOTE**
> This tutorial is not intended to be a detailed examination at configuring a web application to support the Membership and Roles APIs. For a thorough look at these APIs and the steps you need to take to configure a website to use them, please read my *Website Security Tutorials*.

To use the application services with a SQL Server database you must first add the database objects used by these providers to the database where you want the user account and role information stored. These requisite database objects include a variety of tables, views, and stored procedures. Unless specified otherwise, the `SqlMembershipProvider` and `SqlRoleProvider` provider classes use a SQL Server Express Edition database named `ASPNETDB` located in the application s `App_Data` folder; if such a database does not exist, it is automatically created with the necessary database objects by these providers at runtime.

It is possible, and usually ideal, to create the application services database objects in the same database where the website s application-specific data is stored. The .NET Framework ships with a tool named `aspnet_regsql.exe` that installs the database objects on a specified database. I have gone ahead and used this tool to add these objects to the `Reviews.mdf` database in the `App_Data` folder (the development database). We'll see how to use this tool later in this tutorial when we add these objects to the production database.

If you add the application services database objects to a database other than `ASPNETDB` you will need to customize the `SqlMembershipProvider` and `SqlRoleProvider` provider classes configurations so that they use the appropriate database. To customize the Membership provider add a *<membership> element* within the `<system.web>` section in `Web.config`; use the *<roleManager> element* to configure the Roles provider. The following snippet is taken from the Book Reviews application s `Web.config` and shows the configure settings for the Membership and Roles APIs. Note that both register a new provider - `ReviewMembership` and `ReviewRole` - that use the `SqlMembershipProvider` and `SqlRoleProvider` providers, respectively.

```
<configuration>
    <system.web>
        ...

        <membership defaultProvider="ReviewMembership">
            <providers>
                <clear />

                <add type="System.Web.Security.SqlMembershipProvider"
                     name="ReviewMembership"
                     connectionStringName="ReviewsConnectionString"
                     applicationName="BookReviews" />
            </providers>
        </membership>

        <roleManager enabled="true" defaultProvider="ReviewRole">
            <providers>
                <clear />

                <add type="System.Web.Security.SqlRoleProvider"
                     name="ReviewRole"
                     connectionStringName="ReviewsConnectionString"
                     applicationName="BookReviews" />
            </providers>
        </roleManager>

        ...
    </system.web>
</configuration>
```

The `Web.config` file s `<authentication>` element has also been configured to support forms-based authentication.

```
<configuration>
    <system.web>
        ...

        <authentication mode="Forms" />

        ...
    </system.web>
</configuration>
```

Limiting Access to the Administration Pages

ASP.NET makes it easy to grant or deny access to a particular file or folder by user or by role via its *URL authorization* feature. (We briefly discussed URL authorization in the *Core Differences Between IIS and the ASP.NET Development Server* tutorial and showed how IIS and the ASP.NET Development Server apply URL authorization rules differently for static versus dynamic content.) Because we want to prohibit access to the `~/Admin` folder except for those users in the Admin role, we need to add URL authorization rules to this folder. Specifically, the URL authorization rules need to allow users in the Admin role and deny all other users. This is accomplished by adding a `Web.config` file to the `~/Admin` folder with the following contents:

```
<?xml version="1.0"?>
<configuration>
    <system.web>
        <authorization>
            <allow roles="Admin" />
            <deny users="*" />
        </authorization>
    </system.web>
</configuration>
```

For more information on ASP.NET s URL authorization feature and how to use it to spell out authorization rules for users and for roles, be sure to read the *User-Based Authorization* and *Role-Based Authorization* tutorials from my *Website Security Tutorials*.

Deploying a Web Application That Uses Application Services

When deploying a website that uses application services and a provider that stores the application services information in a database, it is imperative that the database objects needed by the application services be created on the production database. Initially the production database does not contain these objects, so when the application is first deployed (or when it is deployed for the first time after application services have been added), you must take extra steps to get these requisite database objects on the production database.

Another challenge can arise when deploying a website that uses application services if you intend to replicate the user accounts created in the development environment to the production environment. Depending on the Membership and Roles configuration, it is possible that even if you successfully copy the user accounts that were created in the development environment to the production database, these users cannot sign into the web application in production. We'll look at the cause of this issue and discuss how to prevent it from happening.

ASP.NET ships with a nice *Web Site Administration Tool (WSAT)* that can be launched from Visual Studio and allows the user account, roles, and authorization rules to be managed through a web-based interface. Unfortunately, the WSAT only works for local websites, meaning that it cannot be used to remotely manage user accounts, roles, and authorization rules for the web application in the production environment. We'll look at different ways to implement WSAT-like behavior from your production website.

Adding the Database Objects Using aspnet_regsql.exe

The *Deploying a Database* tutorial showed how to copy the tables and data from the development database to the production database, and these techniques can certainly be used to copy the application services database objects to the production database. Another option is the `aspnet_regsql.exe` tool, which adds or removes the application services database objects from a database.

> **NOTE**
>
> The `aspnet_regsql.exe` tool creates the database objects on a specified database. It does not migrate data in those database objects from the development database to the production database. If you mean to copy the user account and role information in the development database to the production database use the techniques covered in the *Deploying a Database* tutorial.

Let s look at how to add the database objects to the production database using the `aspnet_regsql.exe` tool. Start by opening Windows Explorer and navigating to the .NET Framework version 2.0 directory on your computer, %WINDIR%\ Microsoft.NET\Framework\v2.0.50727. There you should find the `aspnet_regsql.exe` tool. This tool can be used from the command-line, but it also includes a graphical user interface; double-click the `aspnet_regsql.exe` file to launch its graphical component.

The tool starts by displaying a splash screen explaining its purpose. Click Next to advance to the "Select a Setup Option" screen, which is shown in Figure 1. From here you can choose to add the application services database

objects or remove them from a database. Because we want to add these objects to the production database, select the "Configure SQL Server for application services" option and click Next.

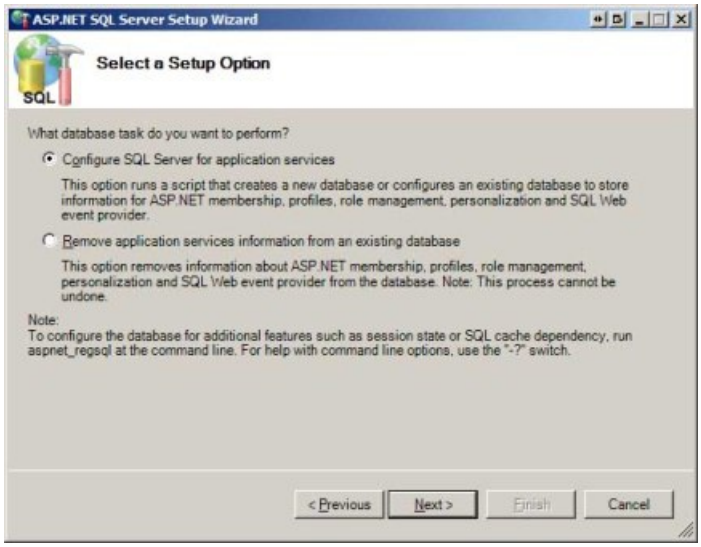


Figure 1: Choose to Configure SQL Server for Application Services (Click to view full-size image)

In "Select the Server and Database" screen prompts for information to connect to the database. Enter the database server, the security credentials, and the database name supplied to you by your web hosting company and click Next.

> **NOTE**
>
> After entering your database server and credentials you may get an error when expanding the database drop-down list. The `aspnet_regsql.exe` tool queries the `sysdatabases` system table to retrieve a list of databases on the server, but some web hosting companies lock down their database servers so that this information is not publicly available. If you get this error you can type the database name directly into the drop-down list.

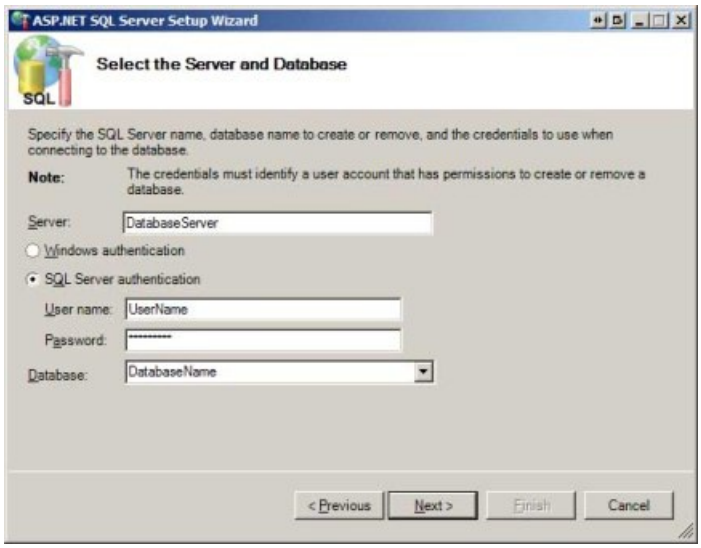


Figure 2: Supply the Tool With Your Database s Connection Information (Click to view full-size image)

The subsequent screen summarizes the actions that are about to take place, namely that the application services database objects are going to be added to the specified database. Click Next to complete this action. After a few moments, the final screen is displayed, noting that the database objects have been added (see Figure 3).

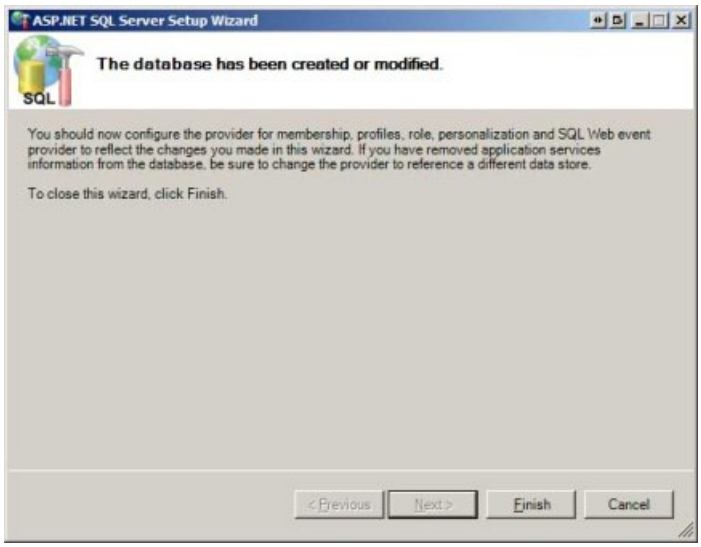


Figure 3: Success! The Application Services Database Objects Were Added to the Production Database (Click to view full-size image)

To verify that the application services database objects were successfully added to the production database, open SQL Server Management Studio and connect to your production database. As Figure 4 shows, you should now see the application services database tables in your database, `aspnet_Applications`, `aspnet_Membership`, `aspnet_Users`, and so forth.

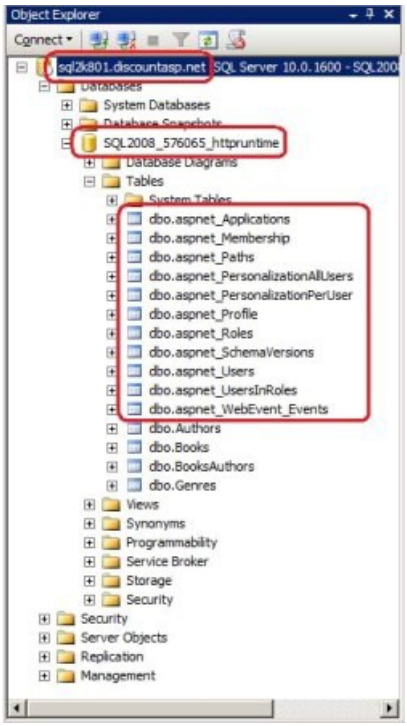


Figure 4: Confirm That the Database Objects Were Added to the Production Database (Click to view full-size image)

You will only need to use the `aspnet_regsql.exe` tool when deploying your web application for the first time or for the first time after you have started using the application services. Once these database objects are on the production database they won't need to be re-added or modified.

Copying User Accounts from Development to Production

When using the `SqlMembershipProvider` and `SqlRoleProvider` provider classes to store the application services information in a SQL Server database, the user account and role information is stored in a variety of database tables, including `aspnet_Users`, `aspnet_Membership`, `aspnet_Roles`, and `aspnet_UsersInRoles`, among others. If during development you create user accounts in the development environment you can replicate those user

accounts in production by copying the corresponding records from the applicable database tables. If you used the Database Publishing Wizard to deploy the application services database objects you may have also elected to copy the records, which would result in the user accounts created in development to also be on production. But, depending on your configuration settings, you may find that those users whose accounts were created in development and copied to production are unable to login from the production website. What gives?

The `SqlMembershipProvider` and `SqlRoleProvider` provider classes were designed such that a single database could serve as a user store for multiple applications, where each application could, in theory, have users with overlapping usernames and roles with the same name. To allow for this flexibility, the database maintains a list of applications in the `aspnet_Applications` table, and each user is associated with one of these applications. Specifically, the `aspnet_Users` table has an `ApplicationId` column that ties each user to a record in the `aspnet_Applications` table.

In addition to the `ApplicationId` column, the `aspnet_Applications` table also includes an `ApplicationName` column, which provides a more human-friendly name for the application. When a website attempts to work with a user account, such as validating a user s credentials from the login page, it must tell the `SqlMembershipProvider` class what application to work with. It usually does this by supplying the application name, and the this value comes from the provider s configuration in `Web.config` - specifically via the `applicationName` attribute.

But what happens if the `applicationName` attribute is not specified in `Web.config`? In such a case the Membership system uses the application root path as the `applicationName` value. If the `applicationName` attribute is not explicitly set in `Web.config`, then, there is the possibility that the development environment and production environment use a different application root and therefore will be associated with different application names in the application services. If such a mismatch occurs then those users created in the development environment will have an `ApplicationId` value that does not match with the `ApplicationId` value for the production environment. The net result is that those users won't be able to login.

> **NOTE**
> If you find yourself in this situation - with user accounts copied to production with a mismatched `ApplicationId` value - you could write a query to update these incorrect `ApplicationId` values to the `ApplicationId` used on production. Once updated, the users whose accounts were created on the development environment would now be able to sign into the web application on production.

The good news is that there is a simple step you can take to ensure that the two environments use the same `ApplicationId` - explicitly set the `applicationName` attribute in `Web.config` for all of your application services providers. I explicitly set the `applicationName` attribute to "BookReviews" in the `<membership>` and `<roleManager>` elements as this snippet from `Web.config` shows.

```
<membership defaultProvider="ReviewMembership">
    <providers>
        <clear />

        <add type="System.Web.Security.SqlMembershipProvider"
             name="ReviewMembership"
             connectionStringName="ReviewsConnectionString"
             applicationName="BookReviews" />
    </providers>
</membership>
```

For more discussion on setting the `applicationName` attribute and the rationale behind it, refer to *Scott Guthrie* s blog post, *Always set the applicationName property when configuring ASP.NET Membership and other Providers*.

Managing User Accounts in the Production Environment

The ASP.NET Web Site Administration Tool (WSAT) makes it easy to create and manage user accounts, define and

apply roles, and spell out user- and role-based authorization rules. You can launch the WSAT from Visual Studio by going to the Solution Explorer and clicking the ASP.NET Configuration icon or by going to the Website or Project menus and selecting the ASP.NET Configuration menu item. Unfortunately, the WSAT can only work with local websites. Therefore, you cannot use the WSAT from your workstation to manage the website in the production environment.

The good news is that all of the functionality exposed provided by the WSAT is available programmatically through the Membership and Roles APIs; furthermore, many of the WSAT screens use the standard ASP.NET Login-related controls. In short, you can add ASP.NET pages to your website that offer the necessary management capabilities.

Recall that an earlier tutorial updated the Book Reviews web application to include an `~/Admin` folder, and this folder has been configured to only allow users in the Admin role. I added a page to that folder named `CreateAccount.aspx` from which an administrator can create a new user account. This page uses the CreateUserWizard control to display the user interface and backend logic for creating a new user account. What s more, I customized the control to include a CheckBox that prompts whether the new user should also be added to the Admin role (see Figure 5). With a little bit of work you can build a custom set of pages that implements the user and role management-related tasks that would otherwise be provided by the WSAT.

> **NOTE**
>
> For more information on using the Membership and Roles APIs along with the Login-related ASP.NET Web controls, be sure to read my *Website Security Tutorials*. For more on customizing the CreateUserWizard control refer to the *Creating User Accounts* and *Storing Additional User Information* tutorials, or check out *Erich Peterson* s article, *Customizing the CreateUserWizard Control*.

Figure 5: Administrators Can Create New User Accounts (Click to view full-size image)

If you need the full functionality of the WSAT check out *Rolling Your Own Web Site Administration Tool*, in which author Dan Clem walks through the process of building a custom WSAT-like tool. Dan shares his application s source code (in C#) and provides step-by-step instructions for adding it to your hosted website.

Summary

When deploying a web application that uses the application services database implementation you must first ensure that the production database has the requisite database objects. These objects can be added using the techniques discussed in the *Deploying a Database* tutorial; alternatively, you can use the `aspnet_regsql.exe` tool, as we saw in this tutorial. Other challenges we touched on center around synchronizing the application name used in the development and production environments (which is important if you want users and roles created

in the development environment to be valid on production) and techniques for managing the users and roles in the production environment.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- *ASP.NET SQL Server Registration Tool (aspnet_regsql.exe)*
- *Creating the Application Services Database for SQL Server*
- *Creating the Membership Schema in SQL Server*
- *Examining ASP.NET s Membership, Roles, and Profile*
- *Rolling Your Own Web Site Administration Tool*
- *Website Security Tutorials*
- *Web Site Administration Tool Overview*

PREVIOUS NEXT

Strategies for Database Development and Deployment (C#)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download PDF

When deploying a data-driven application for the first time you can blindly copy the database in the development environment to the production environment. But performing a blind copy in subsequent deployments will overwrite any data entered into the production database. Instead, deploying a database involves applying the changes made to the development database since the last deployment onto the production database. This tutorial examines these challenges and offers various strategies to assist with chronicling and applying the changes made to the database since the last deployment.

Introduction

As discussed in previous tutorials, deploying an ASP.NET application entails copying the pertinent content from the development environment to the production environment. Deployment is not a one-time event, but rather something that happens every time a new version of the software is released or when bugs or security concerns have been identified and addressed. When copying ASP.NET pages, images, JavaScript files, and other such files to the production environment you do not need to concern yourself with how these file have been changed since the last deployment. You can blindly copy the file to production, overwriting the existing content. Unfortunately, this simplicity does not extend to deploying the database.

When deploying a data-driven application for the first time you can blindly copy the database in the development environment to the production environment. But performing a blind copy in subsequent deployments will overwrite any data entered into the production database. Instead, deploying a database involves applying the *changes* made to the development database since the last deployment onto the production database. This tutorial examines these challenges and offers various strategies to assist with chronicling and applying the changes made to the database since the last deployment.

The Challenges of Deploying a Database

Before a data-driven application has been deployed for the first time, there is only one database, namely the database in the development environment, which is why when deploying a data-driven application for the first time you can blindly copy the database in the development environment to the production environment. But once the application has been deployed there are two copies of the database: one in development and one in production.

Between deployments the development and production databases can become out of sync. While the production database s schema remains unchanged, the development database s schema may change as new features are added. You might add or remove columns, tables, views, or stored procedures. There may also be important data that gets added to the development database. Many data-driven applications include lookup tables populated with hard-coded, application-specific data that are not user-editable. For example, an auction website might have a drop-down list with choices that describe the condition of the item being auctioned: New, Like New, Good, and Fair. Rather than hard-coding these options directly in the drop-down list it is usually better to place them in a database table. If, during development, a new condition named Poor is added to the table then when deploying the application this same record needs to be added to the lookup table in the production database.

Ideally, deploying the database would involve copying the database from development to production. But keep in mind that after you have deployed the application and resumed development, the production database is being populated with real data from real users. Therefore, if you were to simply copy the database from development to production at the next deployment you would overwrite the production database and lose its existing data. The net result is that deploying the database boils down to applying the changes made to the development database since the last deployment.

Because deploying a database involves applying the changes in the schema and, possibly, the data since the last deployment, a history of changes must be maintained (or determined at deploy-time) so that those changes can be applied on production. There are a variety of techniques for managing and applying changes to the data model.

Defining the Baseline

To maintain the changes to your application s database you need to have some starting state, a baseline to which the changes are applied to. At one extreme the starting state could be an empty database with no tables, views, or stored procedures. Such a baseline results in a large change log because it must include the creation of all of the database s tables, views, and stored procedures along with any changes made after the initial deployment. At the other end of the spectrum you could set the baseline as the version of the database that is initially deployed to the production environment. This choice results in a much smaller change log because it only includes the changes made to the database following the first deployment. This is the approach I prefer. And of course you can choose a more middle of the road approach, defining the baseline as some point between the initial creation of the database and when the database is first deployed.

Once you have chosen a baseline consider generating a SQL script that can be executed to recreate the baseline version. Such a script makes it possible to quickly recreate the baseline version of the database. This functionality is especially useful in larger projects, where there may be multiple developers working on the project or additional environments, such as testing or staging, that each need their own copy of the database.

There are a variety of tools at your disposal to generate a SQL script of the baseline version. From SQL Server Management Studio (SSMS) you can right-click on the database, go to the Tasks submenu, and choose the Generate Scripts option. This launches the Script Wizard, which you can instruct to generate a file that contains the SQL commands to create your database s objects. Another option is the Database Publishing Wizard, which can generate the SQL commands to not only create the database schema, but also the data in the database tables. The Database Publishing Wizard was examined in detail back in the *Deploying a Database* tutorial. Regardless of what tool you use, in the end you should have a script file that you can use to recreate the baseline version of your database, should the need arise.

Documenting the Database Changes in Prose

The simplest way to maintain a log of changes to the data model during the development phase is to record the changes in prose. For example, if during development of an already-deployed application you add a new column to the `Employees` table, remove a column from the `Orders` table, and add a new table (`ProductCategories`), you would maintain a text file or Microsoft Word document with the following history:

CHANGE DATE	CHANGE DETAILS
2009-02-03:	Added column `DepartmentID` (`int`, NOT NULL) to the `Employees` table. Added a foreign key constraint from `Departments.DepartmentID` to `Employees.DepartmentID`.
2009-02-05:	Removed column `TotalWeight` from the `Orders` table. Data already captured in associated `OrderDetails` records.

CHANGE DATE	CHANGE DETAILS
2009-02-12:	Created the `ProductCategories` table. There are three columns: `ProductCategoryID` (`int` , `IDENTITY` , `NOT NULL`), `CategoryName` (`nvarchar(50)` , `NOT NULL`), and `Active` (`bit` , `NOT NULL`). Added a primary key constraint to `ProductCategoryID` , and a default value of 1 to `Active` .

There are a number of drawbacks to this approach. For starters, there is no hope for automation. Anytime these changes need to be applied to a database - such as when the application is deployed - a developer must manually implement each change, one at a time. Moreover, if you need to reconstruct a particular version of the database from the baseline using the change log, doing so will take more and more time as the size of the log grows. Another drawback to this method is that the clarity and level of detail of each change log entry is left to the person recording the change. In a team with multiple developers some may make more detailed, more readable, or more precise entries than others. Also, typos and other human-related data entry errors are possible.

The primary benefit of documenting the database changes in prose is simplicity. You don t need familiarity with the SQL syntax for creating and altering database objects. Instead, you can record the changes in prose and implement them through SQL Server Management Studio s graphical user interface.

Maintaining your change log in prose is, admittedly, not very sophisticated and won't work well with certain projects, such as ones that are large in scope, have frequent changes to the data model, or involve multiple developers. But I have seen this approach work quite well in small, one-man projects that have only occasional changes to the data model and where the solo developer does not have a strong background in the SQL syntax for creating and altering database objects.

> **NOTE**
> While the information in the change log is, technically, only needed until deploy-time, I recommend keeping a history of changes. But rather than maintaining a single, ever growing change log file, consider having a different change log file for each database version. Typically you will want to version the database each time it is deployed. By maintaining a log of change logs you can, starting from the baseline, recreate any database version by executing the change log scripts starting from version 1 and continuing until you reach the version you need to recreate.

Recording the SQL Change Statements

The primary drawback of maintaining the change log in prose is the lack of automation. Ideally, implementing the database changes to the production database at deploy-time would be as easy as clicking a button to execute a script rather than having to manually perform a list of instructions. Such automation is possible by maintaining a change log that contains those SQL commands used to alter the data model.

The SQL syntax includes a number of statements for creating and modifying various database objects. For example, the *CREATE TABLE statement*, when executed, creates a new table with the specified columns and constraints. The *ALTER TABLE statement* modifies an existing table, adding, removing, or modifying its columns or constraints. There are also statements to create, modify, and drop indexes, views, user-defined functions, stored procedures, triggers, and other database objects.

Returning to our earlier example, image that during development of an already-deployed application you add a new column to the `Employees` table, remove a column from the `Orders` table, and add a new table (`ProductCategories`). Such actions would result in a change log file with the following SQL commands:

```
-- Add the DepartmentID column

ALTER TABLE [Employees] ADD [DepartmentID]
int NOT NULL

-- Add a foreign key constraint between Departments.DepartmentID and Employees.DepartmentID
ALTER TABLE [Employees] ADD
CONSTRAINT [FK_Departments_DepartmentID]
      FOREIGN
KEY ([DepartmentID])
      REFERENCES
[Departments] ([DepartmentID])

-- Remove TotalWeight column from Orders
ALTER TABLE [Orders] DROP COLUMN
[TotalWeight]

-- Create the ProductCategories table

CREATE TABLE [ProductCategories]
(
      [ProductCategoryID]
int IDENTITY(1,1) NOT NULL,
      [CategoryName]
nvarchar(50) NOT NULL,
      [Active]
bit NOT NULL CONSTRAINT [DF_ProductCategories_Active]  DEFAULT
((1)),
      CONSTRAINT
[PK_ProductCategories] PRIMARY KEY CLUSTERED ( [ProductCategoryID])
)
```

Pushing these changes to the production database at deploy-time is a one-click operation: open SQL Server Management Studio, connect to your production database, open a New Query window, paste the contents of the change log, and click Execute to run the script.

Using a Comparison Tool to Synchronize the Data Models

Documenting database changes in prose is easy, but implementing the changes requires a developer to make each change on the production database one at a time; documenting the change SQL commands makes implementing those changes on the production database as easy and quick as clicking a button, but requires learning and mastering the SQL statements and syntax for creating and altering database objects. Database comparison tools take the best from both approaches and discard the worst.

A database comparison tool compares the schema or data of two databases and displays a summary report showing you how the databases differ. Then, with the click of a button, you can generate the SQL commands for synchronizing one or more database objects. In a nutshell, you can use a database comparison tool to compare the development and production databases at deploy-time, generating a file that contains the SQL commands that, when executed, will apply the changes to the production database s schema so that it mirrors the development database s schema.

There are a variety of third-party database comparison tools offered by many different vendors. One such example is *SQL Compare*, by *Red Gate Software*. Let s walk through the process of using SQL Compare to compare and synchronize the development and production databases schemas.

> **NOTE**
>
> At the time of this writing the current version of SQL Compare was version 7.1, with the Standard Edition costing $395. You can follow along by downloading a free 14-day trial.

When SQL Compare starts the Comparison Projects dialog box opens, showing the saved SQL Compare projects. Create a new project. This launches the Project Configuration wizard, which prompts for information about the databases to compare (see Figure 1). Enter the information for the development and production environment databases.

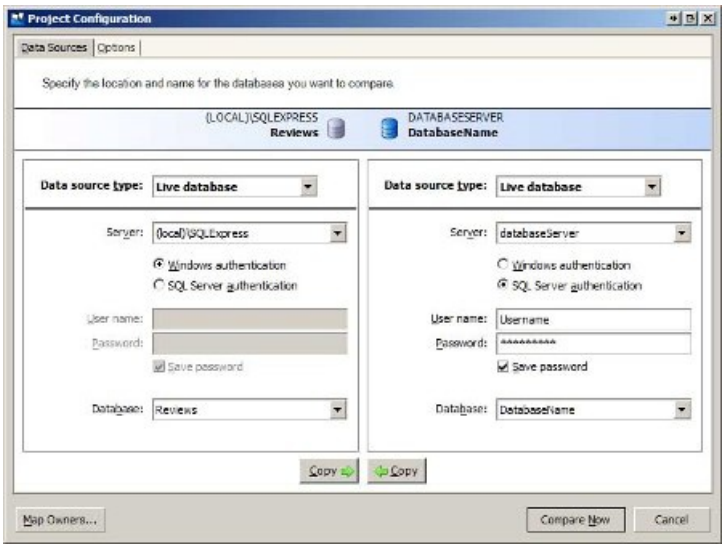


Figure 1: Compare the Development and Production Databases (Click to view full-size image)

> **NOTE**
>
> If your development environment database is a SQL Express Edition database file in the `App_Data` folder of your website you will need to register the database in the SQL Server Express database server in order to select it from the dialog box shown in Figure 1. The easiest way to accomplish this is to open SQL Server Management Studio (SSMS), connect to the SQL Server Express database server, and attach the database. If you do not have SSMS installed on your computer you can download and install the free *SQL Server 2008 Management Studio Basic version*.

In addition to selecting the databases to compare, you can also specify a variety of comparison settings from the Options tab. One option you may want to turn on is the "Ignore constraint and index names." Recall that in the preceding tutorial we added the application services database objects to the development and production databases. If you used the `aspnet_regsql.exe` tool to create these objects on the production database then you will find that the primary key and unique constraint names differ between the development and production databases. Consequently, SQL Compare will flag all of the application services tables as differing. You can either leave the "Ignore constraint and index names" unchecked and synchronize the constraint names, or instruct SQL Compare to ignore these differences.

After selecting the databases to compare (and reviewing the comparison options), click the Compare Now button to begin the comparison. Over the next several seconds, SQL Compare examines the schemas of the two databases and generates a report of how they differ. I ve purposefully made some modifications to the development database to show how such discrepancies are noted in the SQL Compare interface. As Figure 2 shows, I ve added a `BirthDate` column to the `Authors` table, removed the `ISBN` column from the `Books` table, and added a new table, `Ratings`, which is meant to let users visiting the site rate the reviewed books.

> **NOTE**
>
> The data model changes made in this tutorial were done to illustrate using a database comparison tool. You will not find these changes in the database in future tutorials.

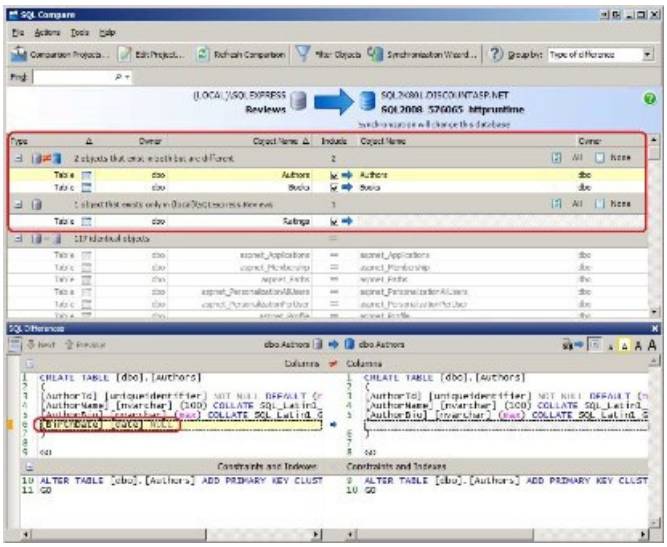


Figure 2: SQL Compare Lists the Differences Between the Development and Production Databases (Click to view full-size image)

SQL Compare breaks down the database objects into groups, quickly showing you what objects exist in both databases but are different, which objects exist in one database but not the other, and which objects are identical. As you can see, there are two objects that exist in both databases but are different: the `Authors` table, which had a column added, and the `Books` table, which had one removed. There is one object that exists only in the development database, namely the newly created `Ratings` table. And there are 117 objects that are identical in both databases.

Selecting a database object displays the SQL Differences window, which shows how these objects differ. The SQL Differences window, displayed at the bottom in Figure 2, highlights that the `Authors` table in the development database has the `BirthDate` column, which is not found in the `Authors` table on the production database.

After reviewing the differences and selecting which objects you want to synchronize, the next step is to generate the SQL commands needed to update the production database s schema to match the development database. This is accomplished through the Synchronization Wizard. The Synchronization Wizard confirms what objects to synchronize and summarizes the action plan (see Figure 3). You can synchronize the databases immediately or generate a script with the SQL commands that can be run at your leisure.

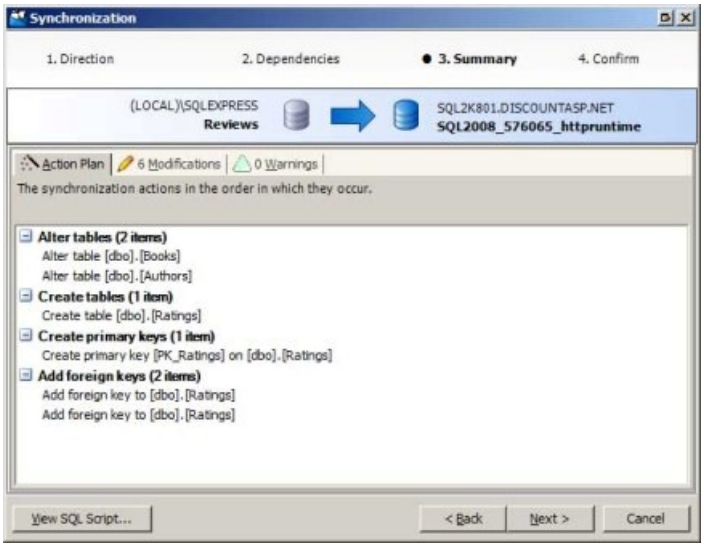


Figure 3: Use the Synchronization Wizard to Synchronize Your Databases Schemas (Click to view full-size image)

Database comparison tools like Red Gate Software s SQL Compare make applying the changes to the development database schema to the production database as easy as point and click.

> **NOTE**
> SQL Compare compares and synchronizes two databases *schemas*. Unfortunately, it does not compare and synchronize the data within two databases tables. Red Gate Software does offer a product named *SQL Data Compare* that compares and synchronizes the data between two databases, but it is a separate product from SQL Compare and costs another $395.

Taking the Application Offline During Deployment

As we ve seen throughout these tutorials, deployment is a process that involves multiple steps: copying the ASP.NET pages, master pages, CSS files, JavaScript files, images, and other necessary content from the development environment to the production environment; copying up the production environment-specific configuration information, if needed; and applying the changes to the data model since the last deployment. Depending on the number of files and the complexity of your database changes, these steps can take anywhere from a few seconds to several minutes to complete. During this window the web application is in flux and users visiting the site may experience errors or unexpected behavior.

When deploying a website it is best to take the web application "offline" until deployment has completed. Taking the application offline (and bringing it back up once the deployment process has finished) is as easy as uploading a file and then deleting it. Starting with ASP.NET 2.0, the mere presence of a file named `app_offline.htm` in the application s root directory takes the entire website "offline." Any request to an ASP.NET page on that site is automatically responded with the contents of the `app_offline.htm` file. Once that file is removed, the application comes back online.

Taking an application offline during deployment, then, is as simple as uploading an `app_offline.htm` file to the production environment s root directory prior to beginning the deployment process and then deleting it (or renaming it to something else) once deployment is complete. For more information on this technique refer to John Peterson s article, Taking an *ASP.NET Application Offline*.

Summary

The main challenge in deploying a data-driven application centers around deploying the database. Because there are two versions of the database - one in the development environment and one in the production environment - these two databases schemas can become out of sync as new features are added in development. What s more, because the production database as being populated with real data from real users, you cannot overwrite the production database with the modified development database like you can when deploying the files that make up the application (the ASP.NET pages, image files, and so forth). Instead, deploying a database entails implementing the precise set of changes made to the development database on the production database since the last deployment.

This tutorial looked at three techniques for maintaining and applying a log of database changes. The simplest approach is to record the changes in prose. While this tactic makes implementing these changes on the production database a manual process, it does not require knowledge of the SQL commands for creating and altering database objects. A more sophisticated approach, and one that is much more palatable in larger projects or projects with multiple developers, is to record the changes as a series of SQL commands. This greatly hastens rolling out these changes to the target database. The best of both approaches can be achieved by using a database comparison tool, such as Red Gate Software s SQL Compare.

This tutorial concludes our focus on deploying a data-driven application. The next set of tutorials looks at how to respond to errors in the production environment. We'll look at how to display a friendly error page rather instead of the Yellow Screen of Death. And we'll see how to log the error s details and how to alert you when such errors occur.

Happy Programming!

Displaying a Custom Error Page (C#)

9/16/2020 • 13 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

What does the user see when a runtime error occurs in an ASP.NET web application? The answer depends on how the website's <customErrors> configuration. By default, users are shown an unsightly yellow screen proclaiming that a runtime error has occurred. This tutorial shows how to customize these settings to display an aesthetically-pleasing custom error page that matches your site's look and feel.

Introduction

In a perfect world there would be no run-time errors. Programmers would write code with nary a bug and with robust user input validation, and external resources like database servers and email servers would never go offline. Of course, in reality errors are inevitable. The classes in the .NET Framework signal an error by throwing an exception. For example, calling a SqlConnection object's Open method establishes a connection to the database specified by a connection string. However, if the database is down or if the credentials in the connection string are invalid then the Open method throws a `SqlException`. Exceptions can be handled by the use of `try/catch/finally` blocks. If code within a `try` block throws an exception, control is transferred to the appropriate catch block where the developer can attempt to recover from the error. If there is no matching catch block, or if the code that threw the exception is not in a try block, the exception percolates up the call stack in search of `try/catch/finally` blocks.

If the exception bubbles all the way up to the ASP.NET runtime without being handled, the `HttpApplication` class's `Error` event is raised and the configured *error page* is displayed. By default, ASP.NET displays an error page that is affectionately referred to as the Yellow Screen of Death (YSOD). There are two versions of the YSOD: one shows the exception details, a stack trace, and other information helpful to developers debugging the application (see **Figure 1**); the other simply states that there was a run-time error (see **Figure 2**).

The exception details YSOD is quite helpful for developers debugging the application, but showing a YSOD to end users is tacky and unprofessional. Instead, end users should be taken to an error page that maintains the site's look and feel with more user-friendly prose describing the situation. The good news is that creating such a custom error page is quite easy. This tutorial starts with a look at ASP.NET's different error pages. It then shows how to configure the web application to show users a custom error page in the face of an error.

Examining the Three Types of Error Pages

When an unhandled exception arises in an ASP.NET application one of three types of error pages is displayed:

- The Exception Details Yellow Screen of Death error page,
- The Runtime Error Yellow Screen of Death error page, or
- A custom error page

The error page developers are most familiar with is the Exception Details YSOD. By default, this page is displayed to users who are visiting locally and therefore is the page that you see when an error occurs when testing the site in the development environment. As its name implies, the Exception Details YSOD provides details about the exception - the type, the message, and the stack trace. What's more, if the exception was raised by code in your ASP.NET page's code-behind class and if the application is configured for debugging then the Exception Details YSOD will also show this line of code (and a few lines of code above and below it).

Figure 1 shows the Exception Details YSOD page. Note the URL in the browser's address window: `http://localhost:62275/Genre.aspx?ID=foo`. Recall that the `Genre.aspx` page lists the book reviews in a particular genre. It requires that `GenreId` value (a `uniqueidentifier`) be passed through the querystring; for example, the appropriate URL to view the fiction reviews is `Genre.aspx?ID=7683ab5d-4589-4f03-a139-1c26044d0146`. If a non-`uniqueidentifier` value is passed in through the querystring (such as "foo") an exception is thrown.

> **NOTE**
> To reproduce this error in the demo web application available for download you can either visit `Genre.aspx?ID=foo` directly or click the "Generate a Runtime Error" link in `Default.aspx`.

Note the exception information presented in **Figure 1**. The exception message, "Conversion failed when converting from a character string to uniqueidentifier" is present at the top of the page. The type of the exception, `System.Data.SqlClient.SqlException`, is listed, as well. There's also the stack trace.

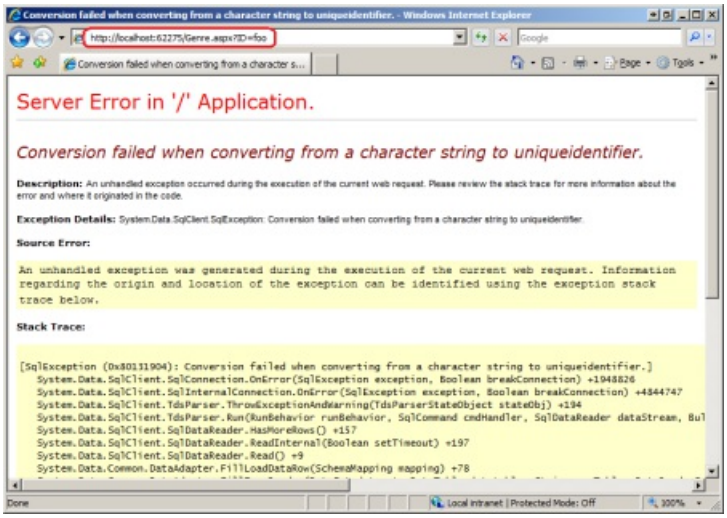


Figure 1: The Exception Details YSOD Includes Information About the Exception
(Click to view full-size image)

The other type of YSOD is the Runtime Error YSOD, and is shown in **Figure 2**. The Runtime Error YSOD informs the visitor that a run-time error has occurred, but it does not include any information about the exception that was thrown. (It does, however, provide instructions on how to make the error details viewable by modifying the `Web.config` file, which is part of what makes such a YSOD look unprofessional.)

By default, the Runtime Error YSOD is shown to users visiting remotely (through http://www.yoursite.com), as evidenced by the URL in the browser's Address bar in **Figure 2**: `http://httpruntime.web703.discountasp.net/Genre.aspx?ID=foo`. The two different YSOD screens exist because developers are interested in knowing the error details, but such information should not be shown on a live site as it may reveal potential security vulnerabilities or other sensitive information to anyone who visits your site.

> **NOTE**
> If you are following along and are using DiscountASP.NET as your web host, you may notice that the Runtime Error YSOD does not display when visiting the live site. This is because DiscountASP.NET has their servers configured to show the Exception Details YSOD by default. The good news is that you can override this default behavior by adding a `<customErrors>` section to your `Web.config` file. The "Configuring Which Error Page is Displayed" section examines the `<customErrors>` section in detail.

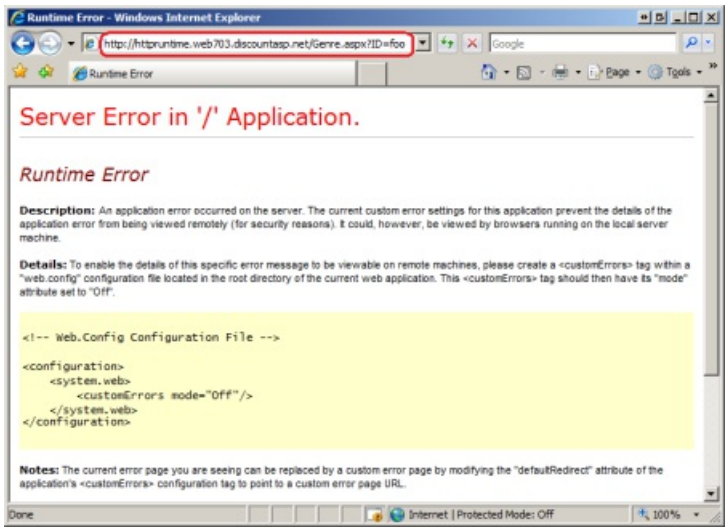


Figure 2: The Runtime Error YSOD Does Not Include Any Error Details
(Click to view full-size image)

The third type of error page is the custom error page, which is a web page that you create. The benefit of a custom error page is that you have complete control over the information that is displayed to the user along with the page's look and feel; the custom error page can use the same master page and styles as your other pages. The "Using a Custom Error Page" section walks through creating a custom error page and configuring it to display in the event of an unhandled exception. **Figure 3** offers a sneak peak of this custom error page. As you can see, the look and feel of the error page is much more professional-looking than either of the Yellow Screens of Death shown in Figures 1 and 2.

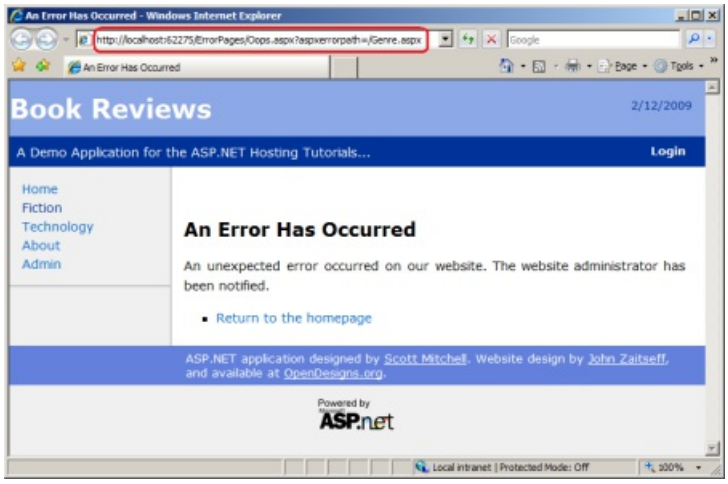


Figure 3: A Custom Error Page Offers a More Tailored Look and Feel
(Click to view full-size image)

Take a moment to inspect the browser's Address bar in **Figure 3**. Note that the Address bar shows the URL of the custom error page (`/ErrorPages/Oops.aspx`). In Figures 1 and 2 the Yellow Screens of Death are shown in the same page that the error originated from (`Genre.aspx`). The custom error page is passed the URL of the page where the error occurred via the `aspxerrorpath` querystring parameter.

Configuring Which Error Page is Displayed

Which of the three possible error pages is displayed is based on two variables:

- The configuration information in the `<customErrors>` section, and
- Whether the user is visiting the site locally or remotely.

The `<customErrors>` section in `Web.config` has two attributes that affect what error page is shown:

`defaultRedirect` and `mode`. The `defaultRedirect` attribute is optional. If provided, it specifies the URL of the custom error page and indicates that the custom error page should be shown instead of the Runtime Error YSOD. The `mode` attribute is required and accepts one of three values: `On`, `Off`, or `RemoteOnly`. These values have the following behavior:

- `On` - indicates that the custom error page or the Runtime Error YSOD is shown to all visitors, regardless of whether they are local or remote.
- `Off` - specifies that the Exception Details YSOD is displayed to all visitors, regardless of whether they are local or remote.
- `RemoteOnly` - indicates that the custom error page or the Runtime Error YSOD is shown to remote visitors, while the Exception Details YSOD is shown to local visitors.

Unless you specify otherwise, ASP.NET acts as if you had set the mode attribute to `RemoteOnly` and had not specified a `defaultRedirect` value. In other words, the default behavior is that the Exception Details YSOD is displayed to local visitors while the Runtime Error YSOD is shown to remote visitors. You can override this default behavior by adding a `<customErrors>` section to your web application's `Web.config file.`

Using a Custom Error Page

Every web application should have a custom error page. It provides a more professional-looking alternative to the Runtime Error YSOD, it is easy to create, and configuring the application to use the custom error page takes only a few moments. The first step is creating the custom error page. I've added a new folder to the Book Reviews application named `ErrorPages` and added to that a new ASP.NET page named `Oops.aspx`. Have the page use the same master page as the rest of the pages on your site so that it automatically inherits the same look and feel.

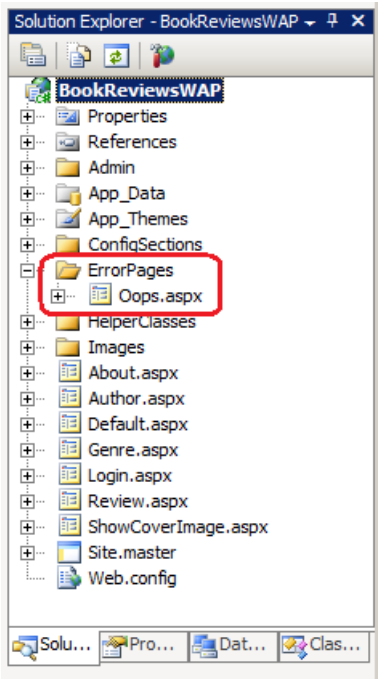


Figure 4: Create a Custom Error Page

Next, spend a few minutes creating the content for the error page. I've created a rather simple custom error page with a message indicating that there was an unexpected error and a link back to the site's homepage.

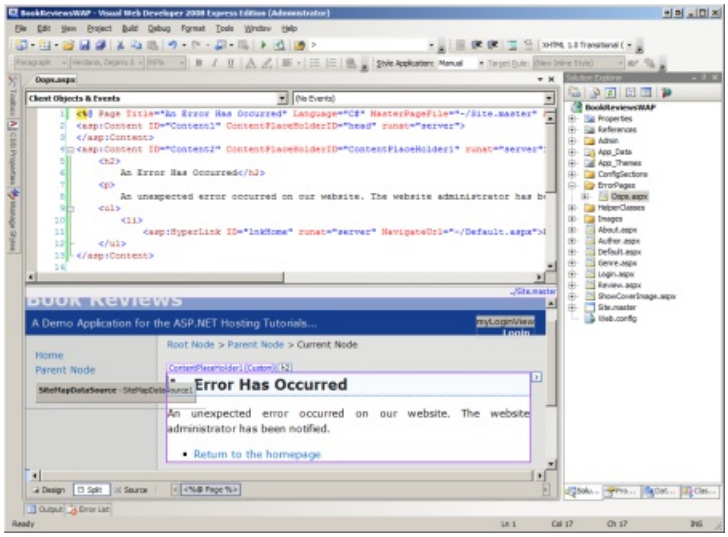


Figure 5: Design Your Custom Error Page
(Click to view full-size image)

With the error page completed, configure the web application to use the custom error page in lieu of the Runtime Error YSOD. This is accomplished by specifying the URL of the error page in the `<customErrors>` section's `defaultRedirect` attribute. Add the following markup to your application's `Web.config` file:

```
<configuration>
    ...

    <system.web>
        <customErrors mode="RemoteOnly"
                      defaultRedirect="~/ErrorPages/Oops.aspx" />

        ...
    </system.web>
</configuration>
```

The above markup configures the application to show the Exception Details YSOD to users visiting locally, while using the custom error page Oops.aspx for those users visiting remotely. To see this in action, deploy your website to the production environment and then visit the Genre.aspx page on the live site with an invalid querystring value. You should see the custom error page (refer back to **Figure 3**).

To verify that the custom error page is only shown to remote users, visit the `Genre.aspx` page with an invalid querystring from the development environment. You should still see the Exception Details YSOD (refer back to **Figure 1**). The `RemoteOnly` setting ensures that users visiting the site on the production environment see the custom error page while developers working locally continue to see the details of the exception.

Notifying Developers and Logging Error Details

Errors that occur in the development environment were caused by the developer sitting at her computer. She is shown the exception's information in the Exception Details YSOD, and she knows what steps she was performing when the error occurred. But when an error occurs on production, the developer has no knowledge that an error occurred unless the end user visiting the site takes the time to report the error. And even if the user goes out of his way to alert the development team that an error occurred, without knowing the exception type, message, and stack trace it can be difficult to diagnose the cause of the error, let alone fix it.

For these reasons it is paramount that any error in the production environment is logged to some persistent store (such as a database) and that the developers are alerted of this error. The custom error page may seem like a good place to do this logging and notification. Unfortunately, the custom error page does not have access to the error details and therefore cannot be used to log this information. The good news is that there are a number of

ways to intercept the error details and to log them, and the next three tutorials explore this topic in more detail.

Using Different Custom Error Pages for Different HTTP Error Statuses

When an exception is thrown by an ASP.NET page and is not handled, the exception percolates up to the ASP.NET runtime, which displays the configured error page. If a request comes into the ASP.NET engine but cannot be processed for some reason - perhaps the requested file is not found or Read permissions have been disabled for the file - then the ASP.NET engine raises an `HttpException`. This exception, like exceptions raised from ASP.NET pages, bubbles up to the runtime, causing the appropriate error page to be displayed.

What this means for the web application in production is that if a user requests a page that is not found then they will see the custom error page. **Figure 6** shows such an example. Because the request is for a non-existent page (`NoSuchPage.aspx`), an `HttpException` is thrown and the custom error page is displayed (note the reference to `NoSuchPage.aspx` in the `aspxerrorpath` querystring parameter).

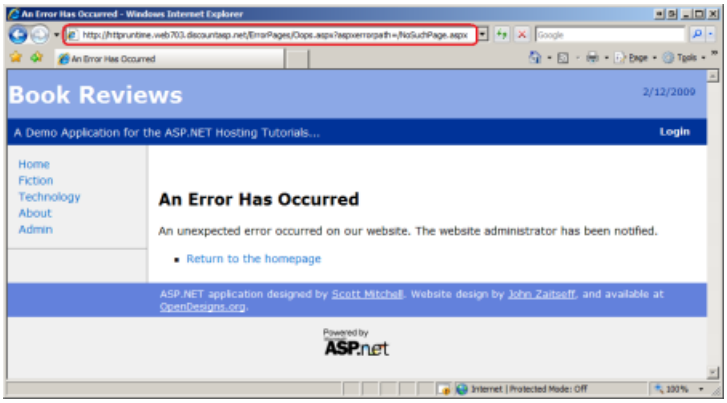


Figure 6: The ASP.NET Runtime Displays the Configured Error Page In Response to an Invalid Request (Click to view full-size image)

By default, all types of errors cause the same custom error page to be displayed. However, you can specify a different custom error page for a specific HTTP status code using `<error>` children elements within the `<customErrors>` section. For example, to have a different error page displayed in the event of a page not found error, which has an HTTP status code of 404, update the `<customErrors>` section to include the following markup:

```
<customErrors mode="RemoteOnly" defaultRedirect="~/ErrorPages/Oops.aspx">
    <error statusCode="404" redirect="~/ErrorPages/404.aspx" />
</customErrors>
```

With this change in place, whenever a user visiting remotely requests an ASP.NET resource that does not exist, they will be redirected to the `404.aspx` custom error page instead of `Oops.aspx`. As **Figure 7** illustrates, the `404.aspx` page can include a more specific message than the general custom error page.

> **NOTE**
>
> Check out 404 Error Pages, One More Time for guidance on creating effective 404 error pages.

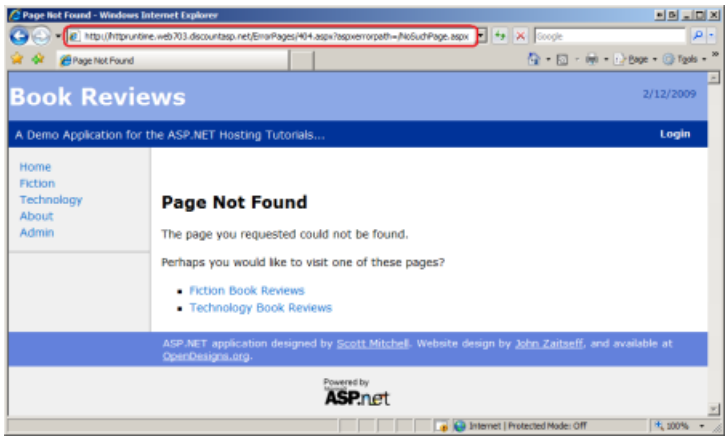


Figure 7: The Custom 404 Error Page Displays a More Targeted Message Than `Oops.aspx` (Click to view full-size image)

Because you know that the `404.aspx` page is only reached when the user makes a request for a page that was not found, you can enhance this custom error page to include functionality to help the user address this specific type of error. For example, you could build a database table that maps known bad URLs to good URLs, and then have the `404.aspx` custom error page run a query against that table and suggest pages the user may be trying to reach.

> **NOTE**
>
> The custom error page is only displayed when a request is made to a resource handled by the ASP.NET engine. As we discussed in the Core Differences Between IIS and the ASP.NET Development Server tutorial, the web server may handle certain requests itself. By default, the IIS web server processes requests for static content like images and HTML files without invoking the ASP.NET engine. Consequently, if the user requests a non-existent image file they will get back IIS's default 404 error message rather than ASP.NET's configured error page.

Summary

When an unhandled exception occurs in an ASP.NET application, the user is shown one of three error pages: the Exception Details Yellow Screen of Death; the Runtime Error Yellow Screen of Death; or a custom error page. Which error page is displayed depends on the application's `<customErrors>` configuration and whether the user is visiting locally or remotely. The default behavior is to show the Exception Details YSOD to local visitors and the Runtime Error YSOD to remote visitors.

While the Runtime Error YSOD hides potentially sensitive error information from the user visiting the site, it breaks from your site's look and feel and makes your application look buggy. A better approach is to use a custom error page, which entails creating and designing the custom error page and specifying its URL in the `<customErrors>` section's `defaultRedirect` attribute. You can even have multiple custom error pages for different HTTP error statuses.

The custom error page is the first step in a comprehensive error handling strategy for a website in production. Alerting the developer of the error and logging its details are also important steps. The next three tutorials explore techniques for error notification and logging.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Error Pages, One More Time
- Design Guidelines for Exceptions

- User-Friendly Error Pages
- Handling and Throwing Exceptions
- Properly Using Custom Error Pages in ASP.NET
- ASP.NET Tracing Overview

PREVIOUS NEXT

Processing Unhandled Exceptions (C#)

2/21/2020 • 16 minutes to read • Edit Online

by Scott Mitchell

View or download sample code (how to download)

When a runtime error occurs on a web application in production it is important to notify a developer and to log the error so that it may be diagnosed at a later point in time. This tutorial provides an overview of how ASP.NET processes runtime errors and looks at one way to have custom code execute whenever an unhandled exception bubbles up to the ASP.NET runtime.

Introduction

When an unhandled exception occurs in an ASP.NET application, it bubbles up to the ASP.NET runtime, which raises the `Error` event and displays the appropriate error page. There are three different types of error pages: the Runtime Error Yellow Screen of Death (YSOD); the Exception Details YSOD; and custom error pages. In the preceding tutorial we configured the application to use a custom error page for remote users and the Exception Details YSOD for users visiting locally.

Using a human-friendly custom error page that matches the look and feel of the site is preferred to the default Runtime Error YSOD, but displaying a custom error page is only one part of a comprehensive error handling solution. When an error occurs in an application in production, it is important that the developers are notified of the error so that they can unearth the cause of the exception and address it. Furthermore, the error's details should be logged so that the error can be examined and diagnosed at a later point in time.

This tutorial shows how to access the details of an unhandled exception so that they can be logged and a developer notified. The two tutorials following this one explore error logging libraries that, after a bit of configuration, will automatically notify developers of runtime errors and log their details.

> **NOTE**
>
> The information examined in this tutorial is most useful if you need to process unhandled exceptions in some unique or customized manner. In cases where you only need to log the exception and notify a developer, using an error logging library is the way to go. The next two tutorials provide an overview of two such libraries.

Executing Code When The `Error` Event Is Raised

Events provide an object a mechanism for signaling that something interesting has occurred, and for another object to execute code in response. As an ASP.NET developer you are accustomed to thinking in terms of events. If you want to run some code when the visitor clicks a particular Button, you create an event handler for that Button's `Click` event and put your code there. Given that the ASP.NET runtime raises its `Error` event whenever an unhandled exception occurs, it follows that the code for logging the error's details would go in an event handler. But how do you create an event handler for the `Error` event?

The `Error` event is one of many events in the `HttpApplication` class that are raised at certain stages in the HTTP pipeline during the lifetime of a request. For example, the `HttpApplication` class's `BeginRequest` event is raised at the start of every request; its `AuthenticateRequest` event is raised when a security module has identified the requestor. These `HttpApplication` events give the page developer a means to execute custom logic at the various points in the lifetime of a request.

Event handlers for the `HttpApplication` events can be placed in a special file named `Global.asax`. To create this file in your website, add a new item to the root of your website using the Global Application Class template with the name `Global.asax`.

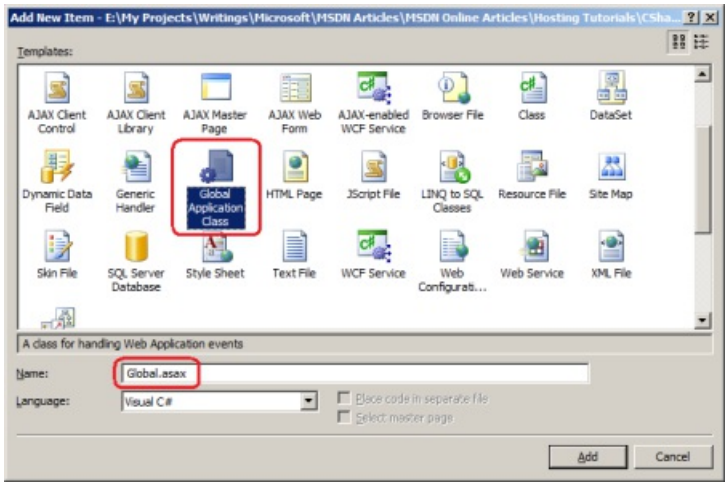


Figure 1: Add `Global.asax` To Your Web Application
(Click to view full-size image)

The contents and structure of the `Global.asax` file created by Visual Studio differ slightly based on whether you are using a Web Application Project (WAP) or Web Site Project (WSP). With a WAP, the `Global.asax` is implemented as two separate files - `Global.asax` and `Global.asax.cs`. The `Global.asax` file contains nothing but an `@Application` directive that references the `.cs` file; the event handlers of interest are defined in the `Global.asax.cs` file. For WSPs, only a single file is created, `Global.asax`, and the event handlers are defined in a `<script runat="server">` block.

The `Global.asax` file created in a WAP by Visual Studio's Global Application Class template includes event handlers named `Application_BeginRequest`, `Application_AuthenticateRequest`, and `Application_Error`, which are event handlers for the `HttpApplication` events `BeginRequest`, `AuthenticateRequest`, and `Error`, respectively. There are also event handlers named `Application_Start`, `Session_Start`, `Application_End`, and `Session_End`, which are event handlers that fire when the web application starts, when a new session starts, when the application ends, and when a session ends, respectively. The `Global.asax` file created in a WSP by Visual Studio contains just the `Application_Error`, `Application_Start`, `Session_Start`, `Application_End`, and `Session_End` event handlers.

> **NOTE**
> When deploying the ASP.NET application you need to copy the `Global.asax` file to the production environment. The `Global.asax.cs` file, which is created in the WAP, does not need to be copied to production because this code is compiled into the project's assembly.

The event handlers created by Visual Studio's Global Application Class template are not exhaustive. You can add an event handler for any `HttpApplication` event by naming the event handler `Application_EventName`. For example, you could add the following code to the `Global.asax` file to create an event handler for the `AuthorizeRequest` event:

```
protected void Application_AuthorizeRequest(object sender, EventArgs e)
{
    // Event handler code
}
```

Likewise, you can remove any event handlers created by the Global Application Class template that are not

needed. For this tutorial we only require an event handler for the `Error` event; feel free to remove the other event handlers from the `Global.asax` file.

> **NOTE**
> *HTTP Modules* offer another way to define event handlers for `HttpApplication` events. HTTP Modules are created as a class file that can be placed directly within the web application project or separated out into a separate class library. Because they can be separated out into a class library, HTTP Modules offer a more flexible and reusable model for creating `HttpApplication` event handlers. Whereas the `Global.asax` file is specific to the web application where it resides, HTTP Modules can be compiled into assemblies, at which point adding the HTTP Module to a website is as simple as dropping the assembly in the `Bin` folder and registering the Module in `Web.config`. This tutorial does not look at creating and using HTTP Modules, but the two error logging libraries used in the following two tutorials are implemented as HTTP Modules. For more background on the benefits of HTTP Modules refer to Using HTTP Modules and Handlers to Create Pluggable ASP.NET Components.

Retrieving Information About the Unhandled Exception

At this point we have a Global.asax file with an `Application_Error` event handler. When this event handler executes we need to notify a developer of the error and log its details. To accomplish these tasks we first need to determine the details of the exception that was raised. Use the Server object's `GetLastError` method to retrieve details of the unhandled exception that caused the `Error` event to fire.

```
protected void Application_Error(object sender, EventArgs e)
{
    // Get the error details
    HttpException lastErrorWrapper =
        Server.GetLastError() as HttpException;
}
```

The `GetLastError` method returns an object of type `Exception`, which is the base type for all exceptions in the .NET Framework. However, in the code above I am casting the Exception object returned by `GetLastError` into an `HttpException` object. If the `Error` event is being fired because an exception was thrown during the processing of an ASP.NET resource then the exception that was thrown is wrapped within an `HttpException`. To get the actual exception that precipitated the Error event use the `InnerException` property. If the `Error` event was raised because of an HTTP-based exception, such as a request for a non-existent page, an `HttpException` is thrown, but it does not have an inner exception.

The following code uses the `GetLastErrormessage` to retrieve information about the exception that triggered the `Error` event, storing the `HttpException` in a variable named `lastErrorWrapper`. It then stores the type, message, and stack trace of the originating exception in three string variables, checking to see if the `lastErrorWrapper` is the actual exception that triggered the `Error` event (in the case of HTTP-based exceptions) or if it's merely a wrapper for an exception that was thrown while processing the request.

```
protected void Application_Error(object sender, EventArgs e)
{
    // Get the error details
    HttpException lastErrorWrapper =
        Server.GetLastError() as HttpException;

    Exception lastError = lastErrorWrapper;
    if (lastErrorWrapper.InnerException != null)
        lastError = lastErrorWrapper.InnerException;

    string lastErrorTypeName = lastError.GetType().ToString();
    string lastErrorMessage = lastError.Message;
    string lastErrorStackTrace = lastError.StackTrace;
}
```

At this point you have all the information you need to write code that will log the exception's details to a database table. You could create a database table with columns for each of the error details of interest - the type, the message, the stack trace, and so on - along with other useful pieces of information, such as the URL of the requested page and the name of the currently logged on user. In the `Application_Error` event handler you would then connect to the database and insert a record into the table. Likewise, you could add code to alert a developer of the error via email.

The error logging libraries examined in the next two tutorials provide such functionality out of the box, so there's no need to build this error logging and notification yourself. However, to illustrate that the `Error` event is being raised and that the `Application_Error` event handler can be used to log error details and notify a developer, let's add code that notifies a developer when an error occurs.

Notifying a Developer When an Unhandled Exception Occurs

When an unhandled exception occurs in the production environment it is important to alert the development team so that they can assess the error and determine what actions need to be taken. For example, if there is an error in connecting to the database then you'll need to double check your connection string and, perhaps, open a support ticket with your web hosting company. If the exception occurred because of a programming error, additional code or validation logic may need to be added to prevent such errors in the future.

The .NET Framework classes in the `System.Net.Mail` namespace make it easy to send an email. The `MailMessage` class represents an email message and has properties like `To`, `From`, `Subject`, `Body`, and `Attachments`. The `SmtpClass` is used to send a `MailMessage` object using a specified SMTP server; the SMTP server settings can be specified programmatically or declaratively in the `<system.net>` element in the `Web.config file`. For more information on sending email messages in an ASP.NET application check out my article, Sending Email in ASP.NET, and the System.Net.Mail FAQ.

> **NOTE**
> The `<system.net>` element contains the SMTP server settings used by the `SmtpClient` class when sending an email. Your web hosting company likely has an SMTP server that you can use to send email from your application. Consult your web host's support section for information on the SMTP server settings you should use in your web application.

Add the following code to the `Application_Error` event handler to send a developer an email when an error occurs:

```
void Application_Error(object sender, EventArgs e)
{
    // Get the error details
    HttpException lastErrorWrapper =
        Server.GetLastError() as HttpException;
```

```
    Exception lastError = lastErrorWrapper;
    if (lastErrorWrapper.InnerException != null)
        lastError = lastErrorWrapper.InnerException;

    string lastErrorTypeName = lastError.GetType().ToString();
    string lastErrorMessage = lastError.Message;
    string lastErrorStackTrace = lastError.StackTrace;

    const string ToAddress = "support@example.com";
    const string FromAddress = "support@example.com";
    const string Subject = "An Error Has Occurred!";

    // Create the MailMessage object
    MailMessage mm = new MailMessage(FromAddress, ToAddress);
    mm.Subject = Subject;
    mm.IsBodyHtml = true;
    mm.Priority = MailPriority.High;
    mm.Body = string.Format(@"
<html>
<body>
  <h1>An Error Has Occurred!</h1>
  <table cellpadding=""5"" cellspacing=""0"" border=""1"">
  <tr>
  <tdtext-align: right;font-weight: bold"">URL:</td>
  <td>{0}</td>
  </tr>
  <tr>
  <tdtext-align: right;font-weight: bold"">User:</td>
  <td>{1}</td>
  </tr>
  <tr>
  <tdtext-align: right;font-weight: bold"">Exception Type:</td>
  <td>{2}</td>
  </tr>
  <tr>
  <tdtext-align: right;font-weight: bold"">Message:</td>
  <td>{3}</td>
  </tr>
  <tr>
  <tdtext-align: right;font-weight: bold"">Stack Trace:</td>
  <td>{4}</td>
  </tr>
  </table>
</body>
</html>",
        Request.RawUrl,
        User.Identity.Name,
        lastErrorTypeName,
        lastErrorMessage,
        lastErrorStackTrace.Replace(Environment.NewLine, "<br />"));

    // Attach the Yellow Screen of Death for this error
    string YSODmarkup = lastErrorWrapper.GetHtmlErrorMessage();
    if (!string.IsNullOrEmpty(YSODmarkup))
    {
        Attachment YSOD =
            Attachment.CreateAttachmentFromString(YSODmarkup, "YSOD.htm");
        mm.Attachments.Add(YSOD);
    }

    // Send the email
    SmtpClient smtp = new SmtpClient();
    smtp.Send(mm);
}
```

While the above code is quite lengthy, the bulk of it creates the HTML that appears in the email sent to the

developer. The code starts by referencing the `HttpException` returned by the `GetLastError` method (`lastErrorWrapper`). The actual exception that was raised by the request is retrieved via `lastErrorWrapper.InnerException` and is assigned to the variable `lastError`. The type, message, and stack trace information is retrieved from `lastError` and stored in three string variables.

Next, a `MailMessage` object named `mm` is created. The email body is HTML formatted and displays the URL of the requested page, the name of the currently logged on user, and information about the exception (the type, message, and stack trace). One of the cool things about the `HttpException` class is that you can generate the HTML used to create the Exception Details Yellow Screen of Death (YSOD) by calling the GetHtmlErrorMessage method. This method is used here to retrieve the Exception Details YSOD markup and add it to the email as an attachment. One word of caution: if the exception that triggered the `Error` event was an HTTP-based exception (such as a request for a non-existent page) then the `GetHtmlErrorMessage` method will return `null`.

The final step is to send the `MailMessage`. This is done by creating a new `SmtpClient` method and calling its `Send` method.

> **NOTE**
> Before using this code in your web application you'll want to change the values in the `ToAddress` and `FromAddress` constants from support@example.com to whatever email address the error notification email should be sent to and originate from. You'll also need to specify SMTP server settings in the `<system.net>` section in `Web.config`. Consult your web host provider to determine the SMTP server settings to use.

With this code in place anytime there's an error the developer is sent an email message that summarizes the error and includes the YSOD. In the preceding tutorial we demonstrated a runtime error by visiting Genre.aspx and passing in an invalid `ID` value through the querystring, like `Genre.aspx?ID=foo`. Visiting the page with the `Global.asax` file in place produces the same user experience as in the preceding tutorial - in the development environment you'll continue to see the Exception Details Yellow Screen of Death, while in the production environment you'll see the custom error page. In addition to this existing behavior, the developer is sent an email.

Figure 2 shows the email received when visiting `Genre.aspx?ID=foo`. The email body summarizes the exception information, while the `YSOD.htm` attachment displays the content that is shown in the Exception Details YSOD (see **Figure 3**).

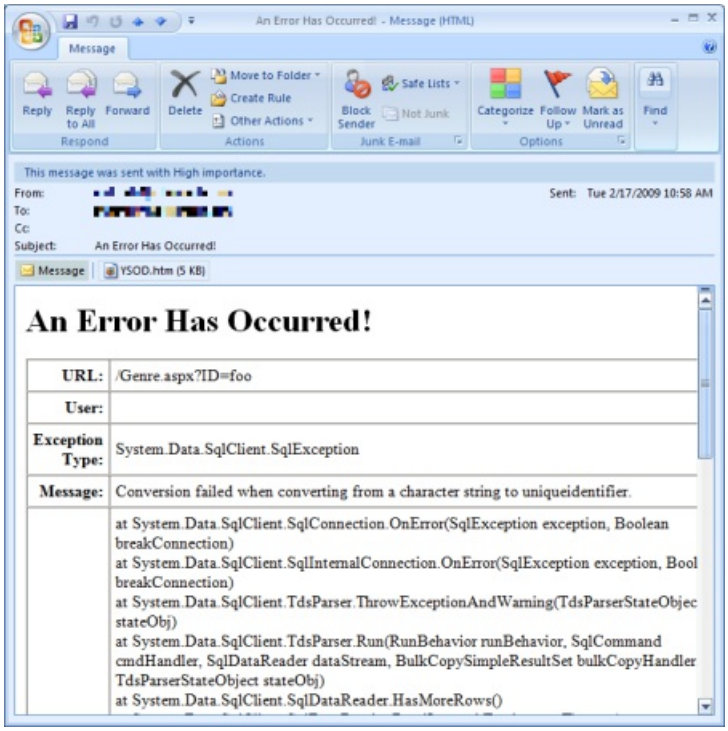


Figure 2: The Developer Is Sent An Email Notification Whenever There's An Unhandled Exception (Click to view full-size image)

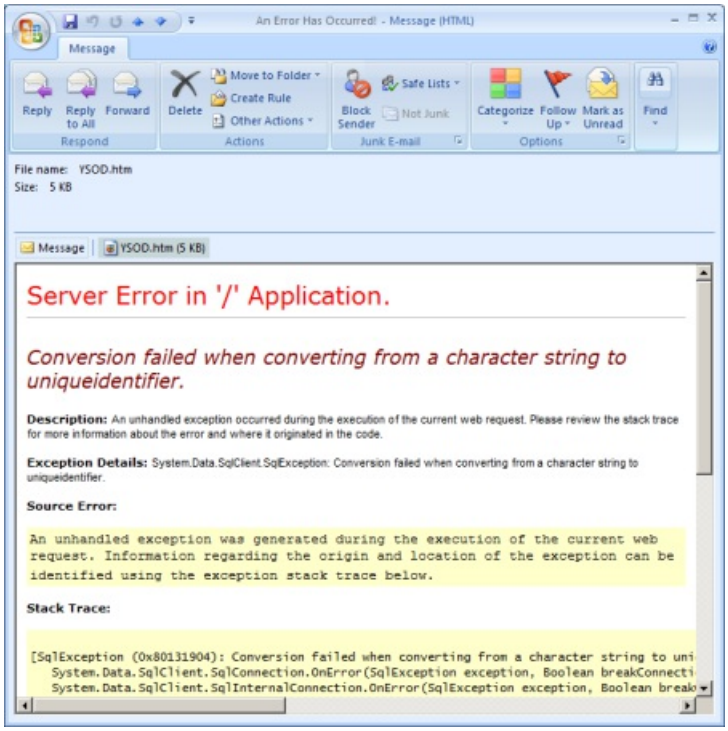


Figure 3: The Email Notification Includes the Exception Details YSOD As An Attachment (Click to view full-size image)

What About Using the Custom Error Page?

This tutorial showed how to use `Global.asax` and the `Application_Error` event handler to execute code when an unhandled exception occurs. Specifically, we used this event handler to notify a developer of an error; we could extend it to also log the error details in a database. The presence of the `Application_Error` event handler does not affect the end user's experience. They still see the configured error page, be it the Error Details YSOD, the Runtime Error YSOD, or the custom error page.

It's natural to wonder whether the `Global.asax` file and `Application_Error` event is necessary when using a custom error page. When an error occurs the user is shown the custom error page so why can't we put the code to notify the developer and log the error details into the code-behind class of the custom error page? While you can certainly add code to the custom error page's code-behind class you do not have access to the details of the exception that triggered the `Error` event when using the technique we explored in the preceding tutorial. Calling the `GetLastError` method from the custom error page returns `Nothing`.

The reason for this behavior is because the custom error page is reached via a redirect. When an unhandled exception reaches the ASP.NET runtime the ASP.NET engine raises its `Error` event (which executes the `Application_Error` event handler) and then *redirects* the user to the custom error page by issuing a `Response.Redirect(customErrorPageUrl)`. The `Response.Redirect` method sends a response to the client with an HTTP 302 status code, instructing the browser to request a new URL, namely the custom error page. The browser then automatically requests this new page. You can tell that the custom error page was requested separately from the page where the error originated because the browser's Address bar changes to the custom error page URL (see **Figure 4**).

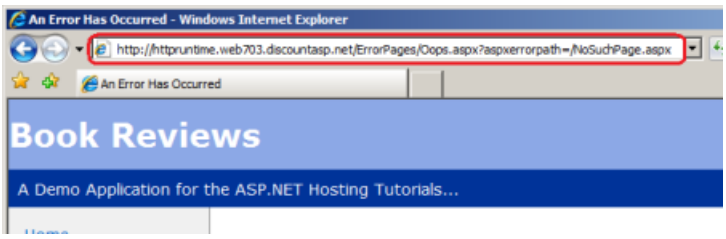


Figure 4: When an Error Occurs the Browser Gets Redirected to the Custom Error Page URL (Click to view full-size image)

The net effect is that the request where the unhandled exception occurred ends when the server responds with the HTTP 302 redirect. The subsequent request to the custom error page is a brand new request; by this point the ASP.NET engine has discarded the error information and, moreover, has no way to associate the unhandled exception in the previous request with the new request for the custom error page. This is why `GetLastError` returns `null` when called from the custom error page.

However, it is possible to have the custom error page executed during the same request that caused the error. The `Server.Transfer(url)` method transfers execution to the specified URL and processes it within the same request. You could move the code in the `Application_Error` event handler to the custom error page's code-behind class, replacing it in `Global.asax` with the following code:

```
protected void Application_Error(object sender, EventArgs e)
{
    // Transfer the user to the appropriate custom error page
    HttpException lastErrorWrapper =
        Server.GetLastError() as HttpException;

    if (lastErrorWrapper.GetHttpCode() == 404)
    {
        Server.Transfer("~/ErrorPages/404.aspx");
    }
    else
    {
        Server.Transfer("~/ErrorPages/Oops.aspx");
    }
}
```

Now when an unhandled exception occurs the `Application_Error` event handler transfers control to the appropriate custom error page based on the HTTP status code. Because control was transferred, the custom error page has access to the unhandled exception information via `Server.GetLastError` and can notify a developer of the error and log its details. The `Server.Transfer` call stops the ASP.NET engine from redirecting the user to the custom error page. Instead, the custom error page's content is returned as the response to the page that generated the error.

Summary

When an unhandled exception occurs in an ASP.NET web application the ASP.NET runtime raises the `Error` event and displays the configured error page. We can notify the developer of the error, log its details, or process it in some other fashion, by creating an event handler for the Error event. There are two ways to create an event handler for `HttpApplication` events like `Error`: in the `Global.asax` file or from an HTTP Module. This tutorial showed how to create an `Error` event handler in the `Global.asax` file that notifies developers of an error by means of an email message.

Creating an `Error` event handler is useful if you need to process unhandled exceptions in some unique or customized manner. However, creating your own `Error` event handler to log the exception or to notify a developer is not the most efficient use of your time as there already exist free and easy to use error logging

libraries that can be setup in a matter of minutes. The next two tutorials examine two such libraries.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET HTTP Modules and HTTP Handlers Overview
- Gracefully Responding to Unhandled Exceptions - Processing Unhandled Exceptions
- `HttpApplication` Class and the ASP.NET Application Object
- HTTP Handlers and HTTP Modules in ASP.NET
- Sending Email in ASP.NET
- Understanding the `Global.asax` File
- Using HTTP Modules and Handlers to Create Pluggable ASP.NET Components
- Working with the ASP.NET `Global.asax` File
- Working with `HttpApplication` Instances

PREVIOUS NEXT

Logging Error Details with ASP.NET Health Monitoring (C#)

2/20/2020 • 11 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Microsoft's health monitoring system provides an easy and customizable way to log various web events, including unhandled exceptions. This tutorial walks through setting up the health monitoring system to log unhandled exceptions to a database and to notify developers via an email message.

Introduction

Logging is a useful tool for monitoring the health of a deployed application and for diagnosing any problems that may arise. It is especially important to log errors that occur in a deployed application so that they can be remedied. The `Error` event is raised whenever an unhandled exception occurs in an ASP.NET application; the preceding tutorial showed how to notify a developer of an error and log its details by creating an event handler for the `Error` event. However, creating an `Error` event handler to log the error's details and notify a developer is unnecessary, as this task can be performed by ASP.NET's *health monitoring system*.

The health monitoring system was introduced in ASP.NET 2.0 and is designed to monitor the health of a deployed ASP.NET application by logging events that occur during the application or request's lifetime. The events logged by the health monitoring system are referred to as *health monitoring events* or *Web events*, and include:

- Application lifetime events, such as when an application starts or stops
- Security events, including failed login attempts and failed URL authorization requests
- Application errors, including unhandled exceptions, view state parsing exceptions, request validation exceptions, and compilation errors, among other types of errors.

When a health monitoring event is raised it can be logged to any number of specified *log sources*. The health monitoring system ships with log sources that log Web events to a Microsoft SQL Server database, to the Windows Event Log, or via an email message, among others. You can also create your own log sources.

The events the health monitoring system logs, along with the log sources used, are defined in `Web.config`. With a few lines of configuration markup you can use health monitoring to log all unhandled exceptions to a database and to notify you of the exception via email.

Exploring the Health Monitoring System's Configuration

The health monitoring system's behavior is defined by its configuration information, which is located in the `<healthMonitoring>` element in `Web.config`. This configuration section defines, among other things, the following three important pieces of information:

1. The health monitoring events that, when raised, should be logged,
2. The log sources, and
3. How each health monitoring event defined in (1) is mapped to the log sources defined in (2).

This information is specified through three children configuration elements: `<eventMappings>`, `<providers>`, and `<rules>`, respectively.

The default health monitoring system configuration information can be found in the `Web.config` file in `%WINDIR%\Microsoft.NET\Framework\version\CONFIG` folder. This default configuration information, with some markup removed for brevity, is shown below:

```
<configuration>
  <system.web>
  <healthMonitoring>
  <eventMappings>
  <add name="All Errors"
type="System.Web.Management.WebBaseErrorEvent,System.Web,Version=2.0.0.0,Culture=neutral,PublicKeyToken=b03f5
f7f11d50a3a" startEventCode="0" endEventCode="2147483647"/>

  <add name="Failure Audits"
type="System.Web.Management.WebFailureAuditEvent,System.Web,Version=2.0.0.0,Culture=neutral,PublicKeyToken=b0
3f5f7f11d50a3a" startEventCode="0" endEventCode="2147483647"/>
  </eventMappings>

  <providers>
  <add name="EventLogProvider"
type="System.Web.Management.EventLogWebEventProvider,System.Web,Version=2.0.0.0,Culture=neutral,PublicKeyToke
n=b03f5f7f11d50a3a"/>

  <add connectionStringName="LocalSqlServer" maxEventDetailsLength="1073741823" buffer="false"
bufferMode="Notification" name="SqlWebEventProvider"
type="System.Web.Management.SqlWebEventProvider,System.Web,Version=2.0.0.0,Culture=neutral,PublicKeyToken=b03
f5f7f11d50a3a"/>
  </providers>

  <rules>
  <add name="All Errors Default" eventName="All Errors" provider="EventLogProvider" profile="Default"
minInstances="1" maxLimit="Infinite" minInterval="00:01:00" custom=""/>

  <add name="Failure Audits Default" eventName="Failure Audits" provider="EventLogProvider" profile="Default"
minInstances="1" maxLimit="Infinite" minInterval="00:01:00" custom=""/>
  </rules>
  </healthMonitoring>
  </system.web>
</configuration>
```

The health monitoring events of interest are defined in the `<eventMappings>` element, which gives a human-friendly name to a class of health monitoring events. In the markup above, the `<eventMappings>` element assigns the human-friendly name "All Errors" to the health monitoring events of type `WebBaseErrorEvent` and the name "Failure Audits" to health monitoring events of type `WebFailureAuditEvent`.

The `<providers>` element defines the log sources, giving them a human-friendly name and specifying any log source-specific configuration information. The first `<add>` element defines the "EventLogProvider" provider, which logs the specified health monitoring events using the `EventLogWebEventProvider` class. The `EventLogWebEventProvider` class logs the event to the Windows Event Log. The second `<add>` element defines the "SqlWebEventProvider" provider, which logs events to a Microsoft SQL Server database via the `SqlWebEventProvider` class. The "SqlWebEventProvider" configuration specifies the database's connection string (`connectionStringName`) among other configuration options.

The `<rules>` element maps the events specified in the `<eventMappings>` element to log sources in the `<providers>` element. By default, ASP.NET web applications log all unhandled exceptions and audit failures to the Windows Event Log.

Logging Events to a Database

The health monitoring system's default configuration can be customized on a web application-by-web application basis by adding a `<healthMonitoring>` section to the application's `Web.config` file. You can include additional

elements in the `<eventMappings>`, `<providers>`, and `<rules>` sections by using the `<add>` element. To remove a setting from the default configuration use the `<remove>` element, or use `<clear />` to remove all default values from one of these sections. Let's configure the Book Reviews web application to log all unhandled exceptions to a Microsoft SQL Server database using the `SqlWebEventProvider` class.

The `SqlWebEventProvider` class is part of the health monitoring system and logs a health monitoring event to a specified SQL Server database. The `SqlWebEventProvider` class expects that the specified database includes a stored procedure named `aspnet_WebEvent_LogEvent`. This stored procedure is passed the details of the event and is tasked with storing the event details. The good news is that you do not need to create this stored procedure nor the table to store the event details. You can add these objects to your database using the `aspnet_regsql.exe` tool.

> **NOTE**
> The `aspnet_regsql.exe` tool was discussed back in the *Configuring a Website That Uses Application Services* tutorial when we added support for ASP.NET's application services. Consequently, the Book Reviews website's database already contains the `aspnet_WebEvent_LogEvent` stored procedure, which stores the event information into a table named `aspnet_WebEvent_Events`.

Once you have the necessary stored procedure and table added to your database, all that remains is to instruct health monitoring to log all unhandled exceptions to the database. Accomplish this by adding the following markup to your website's `Web.config` file:

```
<configuration>
  ...
  <system.web>
  ...
  <healthMonitoring enabled="true">
  <eventMappings>
  <clear />

  <add name="All Errors" type="System.Web.Management.WebBaseErrorEvent"
  startEventCode="0" endEventCode="2147483647" />
  </eventMappings>

  <providers>
  <clear />

  <add connectionStringName="ReviewsConnectionString" maxEventDetailsLength="1073741823"
  buffer="false" name="SqlWebEventProvider"
  type="System.Web.Management.SqlWebEventProvider" />
  </providers>

  <rules>
  <clear />

  <add name="All Errors Default" eventName="All Errors" provider="SqlWebEventProvider"
  profile="Default" minInstances="1" maxLimit="Infinite" minInterval="00:00:00" />
  </rules>
  </healthMonitoring>
  </system.web>
</configuration>
```

The health monitoring configuration markup above uses `<clear />` elements to wipe the pre-defined health monitoring configuration information from the `<eventMappings>`, `<providers>`, and `<rules>` sections. It then adds a single entry to each of these sections.

- The `<eventMappings>` element defines a single health monitoring event of interest named "All Errors", which is raised whenever an unhandled exception occurs.

- The `<providers>` element defines a single log source named "SqlWebEventProvider" that uses the `SqlWebEventProvider` class. The `connectionStringName` attribute has been set to "ReviewsConnectionString", which is the name of our connection string defined in the `<connectionStrings>` section.
- Finally, the <rules> element indicates that when an "All Errors" event transpires that it should be logged using the "SqlWebEventProvider" provider.

This configuration information instructs the health monitoring system to log all unhandled exceptions to the Book Reviews database.

> **NOTE**
>
> The `WebBaseErrorEvent` event is only raised for server errors; it is not raised for HTTP errors, such as a request for an ASP.NET resource that is not found. This differs from the behavior of the `HttpApplication` class's `Error` event, which is raised for both server and HTTP errors.

To see the health monitoring system in action, visit the website and generate a runtime error by visiting `Genre.aspx?ID=foo`. You should see the appropriate error page - either the Exception Details Yellow Screen of Death (when visiting locally) or the custom error page (when visiting the site in production). Behind the scenes, the health monitoring system logged the error information to the database. There should be one record in the `aspnet_WebEvent_Events` table (see **Figure 1**); this record contains information about the runtime error that just occurred.

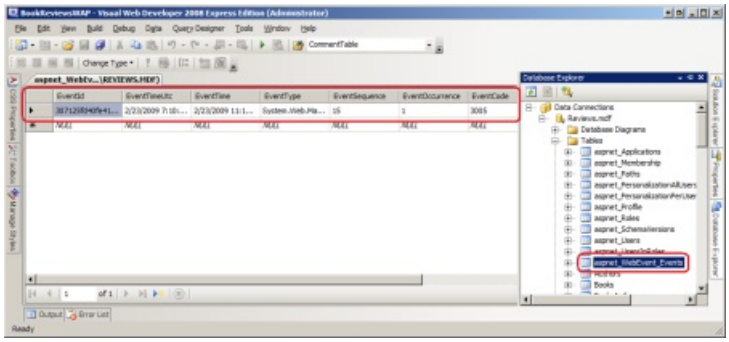


Figure 1: The Error Details Were Logged to the `aspnet_WebEvent_Events` Table
(Click to view full-size image)

Displaying the Error Log In a Web Page

With the website's current configuration, the health monitoring system logs all unhandled exceptions to the database. However, health monitoring does not provide any mechanism to view the error log through a web page. However, you could build an ASP.NET page that displays this information from the database. (As we'll see momentarily, you can opt to have the error details sent to you in an email message.)

If you create such a page, make sure you take steps to allow only authorized users to view the error details. If your site already employs user accounts then you can use URL authorization rules to restrict access to the page to certain users or roles. For more information on how to grant or restrict access to web pages based on the logged in user, refer to my Website Security Tutorials.

> **NOTE**
>
> The subsequent tutorial explores an alternative error logging and notification system named ELMAH. ELMAH includes a built-in mechanism to view the error log from both a web page and as an RSS feed.

Logging Events to Email

The health monitoring system includes a log source provider that "logs" an event to an email message. The log

source includes the same information that is logged to the database in the email message body. You can use this log source to notify a developer when a certain health monitoring event occurs.

Let's update the Book Reviews website's configuration so that we receive an email whenever an exception occurs. To accomplish this we need to perform three tasks:

1. Configure the ASP.NET web application to send email. This is accomplished by specifying how email messages are sent via the `<system.net>` configuration element. For more information on sending email messages in an ASP.NET application refer to Sending Email in ASP.NET and the System.Net.Mail FAQ.
2. Register the email log source provider in the `<providers>` element, and
3. Add an entry to the `<rules>` element that maps the "All Errors" event to the log source provider added in step (2).

The health monitoring system includes two email log source provider classes: `SimpleMailWebEventProvider` and `TemplatedMailWebEventProvider`. The `SimpleMailWebEventProvider` class sends a plain-text email message that includes the event details and provides little customization of the email body. With the `TemplatedMailWebEventProvider` class you specify an ASP.NET page whose rendered markup is used as the body for the email message. The `TemplatedMailWebEventProvider` class gives you much greater control over the contents and format of the email message, but requires a bit more upfront work as you have to create the ASP.NET page that generates the email message's body. This tutorial focuses on using the `SimpleMailWebEventProvider` class.

Update the health monitoring system's `<providers>` element in the `Web.config` file to include a log source for the `SimpleMailWebEventProvider` class:

```
<providers>
  <clear />

  <add type="System.Web.Management.SimpleMailWebEventProvider"
  name="EmailWebEventProvider" buffer="false"
  from="support@example.com" to="support@example.com"
  subjectPrefix="Book Reviews Runtime Error: " />

  <add connectionStringName="ReviewsConnectionString" maxEventDetailsLength="1073741823"
  buffer="false" name="SqlWebEventProvider"
  type="System.Web.Management.SqlWebEventProvider" />
</providers>
```

The above markup uses the `SimpleMailWebEventProvider` class as the log source provider, and assigns it the friendly name "EmailWebEventProvider". Moreover, the `<add>` attribute includes additional configuration options, such as the To and From addresses of the email message.

With the email log source defined, all that remains is to instruct the health monitoring system to use this source to "log" unhandled exceptions. This is accomplished by adding a new rule in the `<rules>` section:

```
<rules>
  <clear />

  <add name="All Errors To Email" eventName="All Errors" provider="EmailWebEventProvider"
  profile="Default" minInstances="1" maxLimit="Infinite" minInterval="00:00:00" />

  <add name="All Errors To Database" eventName="All Errors" provider="SqlWebEventProvider"
  profile="Default" minInstances="1" maxLimit="Infinite" minInterval="00:00:00" />
</rules>
```

The `<rules>` section now includes two rules. The first one, named "All Errors To Email", sends all unhandled exceptions to the "EmailWebEventProvider" log source. This rule has the effect of sending details about errors on

the website to the specified To address. The "All Errors To Database" rule logs the error details to the site's database. Consequently, whenever an unhandled exception occurs on the site its details are both logged to the database and sent to specified email address.

Figure 2 shows the email generated by the `SimpleMailWebEventProvider` class when visiting `Genre.aspx?ID=foo`.

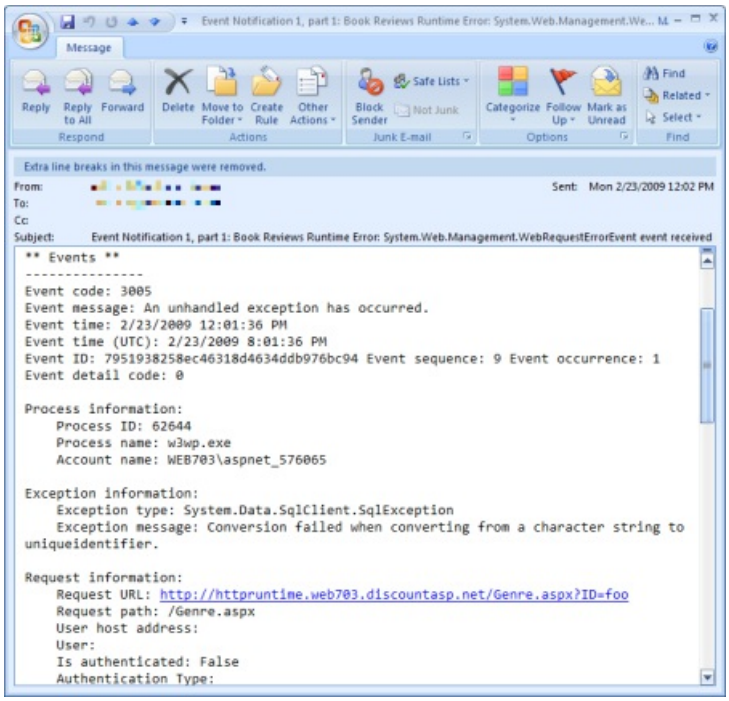


Figure 2: The Error Details are Sent in an Email Message
(Click to view full-size image)

Summary

The ASP.NET health monitoring system is designed to allow administrators to monitor the health of a deployed web application. Health monitoring events are raised when certain actions unfold, such as when the application stops, when a user successfully logs onto the site, or when an unhandled exception occurs. These events can be logged to any number of log sources. This tutorial showed how to log the details of unhandled exceptions to a database and through an email message.

This tutorial focused on using health monitoring to log unhandled exceptions, but keep in mind that health monitoring is designed to measure the overall health of a deployed ASP.NET application and includes a wealth of health monitoring events and log sources not explored here. What's more, you can create your own health monitoring events and log sources, should the need arise. If you are interested in learning more about health monitoring, a good first step is to read through Erik Reitan's health monitoring FAQ. Following that, consult How To: Use Health Monitoring in ASP.NET 2.0.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Health Monitoring Overview
- Configuring and Customizing the Health Monitoring System of ASP.NET
- FAQ - Health Monitoring in ASP.NET 2.0
- How To: Send E-Mail for Health Monitoring Notifications
- How To: Use Health Monitoring in ASP.NET
- Health Monitoring in ASP.NET

Logging Error Details with ELMAH (C#)

2/20/2020 • 18 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Error Logging Modules And Handlers (ELMAH) offers another approach to logging runtime errors in a production environment. ELMAH is a free, open source error logging library that includes features like error filtering and the ability to view the error log from a web page, as an RSS feed, or to download it as a comma-delimited file. This tutorial walks through downloading and configuring ELMAH.

Introduction

The preceding tutorial examined ASP.NET's health monitoring system, which offers an out of the box library for recording a wide array of Web events. Many developers use health monitoring to log and email the details of unhandled exceptions. However, there are a few pain points with this system. First and foremost is the lack any sort of user interface for viewing information about the logged events. If you want to see a summary of the 10 last errors, or view the details of an error that happened last week, you must either poke through the database, scan through your email Inbox, or build a web page that displays information from the `aspnet_WebEvent_Events` table.

Another pain point centers around health monitoring's complexity. Because health monitoring can be used to record a plethora of different events, and because there are a variety of options for instructing how and when events are logged, correctly configuring the health monitoring system can be an onerous task. Finally, there are compatibility issues. Because health monitoring was first added to the .NET Framework in version 2.0, it is not available for older web applications built using ASP.NET version 1.x. Moreover, the `SqlWebEventProvider` class, which we used in the preceding tutorial to logs error details to a database, only works with Microsoft SQL Server databases. You will need to create a custom log provider class should you need to log errors to an alternate data store, such as an XML file or Oracle database.

An alternative to the health monitoring system is Error Logging Modules And Handlers (ELMAH), a free, open-source error logging system created by Atif Aziz. The most notable difference between the two systems is ELAMH's ability to display a list of errors and the details of a specific error from a web page and as an RSS feed. ELMAH is easier to configure than health monitoring because it only logs errors. Furthermore, ELMAH includes support for ASP.NET 1.x, ASP.NET 2.0, and ASP.NET 3.5 applications, and ships with a variety of log source providers.

This tutorial walks through the steps involved in adding ELMAH to an ASP.NET application. Let's get started!

> **NOTE**
>
> The health monitoring system and ELMAH both have their own sets of pros and cons. I encourage you to try both systems and decide what one best suits your needs.

Adding ELMAH to an ASP.NET Web Application

Integrating ELMAH into a new or existing ASP.NET application is an easy and straightforward process that takes under five minutes. In a nutshell, it involves four simple steps:

1. Download ELMAH and add the `Elmah.dll` assembly to your web application,
2. Register ELMAH's HTTP Modules and Handler in `Web.config`,
3. Specify ELMAH's configuration options, and
4. Create the error log source infrastructure, if needed.

Let's walk through each of these four steps, one at a time.

Step 1: Downloading the ELMAH Project Files and Adding `Elmah.dll` To Your Web Application

ELMAH 1.0 BETA 3 (Build 10617), the most recent version at the time of writing, is included in the download available with this tutorial. Alternatively, you may visit the ELMAH website to get the most recent version or to download the source code. Extract the ELMAH download to a folder on your desktop and locate the ELMAH assembly file (`Elmah.dll`).

> **NOTE**
> The `Elmah.dll` file is located in the download's `Bin` folder, which has subfolders for different .NET Framework versions and for Release and Debug builds. Use the Release build for the appropriate framework version. For instance, if you are building an ASP.NET 3.5 web application, copy the `Elmah.dll` file from the `Bin\net-3.5\Release` folder.

Next, open Visual Studio and add the assembly to your project by right-clicking on the website name in the Solution Explorer and choosing Add Reference from the context menu. This brings up the Add Reference dialog box. Navigate to the Browse tab and choose the `Elmah.dll` file. This action adds the `Elmah.dll` file to the web application's `Bin` folder.

> **NOTE**
> The Web Application Project (WAP) type does not show the `Bin` folder in the Solution Explorer. Instead, it lists these items under the References folder.

The `Elmah.dll` assembly includes the classes used by the ELMAH system. These classes fall into one of three categories:

- **HTTP Modules** - an HTTP Module is a class that defines event handlers for `HttpApplication` events, such as the `Error` event. ELMAH includes multiple HTTP Modules, the three most germane ones being:
  - `ErrorLogModule` - logs unhandled exceptions to a log source.
  - `ErrorMailModule` - sends the details of an unhandled exception in an email message.
  - `ErrorFilterModule` - applies developer-specified filters to determine what exceptions are logged and what ones are ignored.
- **HTTP Handlers** - an HTTP Handler is a class that is responsible for generating the markup for a particular type of request. ELMAH includes HTTP Handlers that render error details as a web page, as an RSS feed, or as a comma-delimited file (CSV).
- **Error Log Sources** - out of the box ELMAH can log errors to memory, to a Microsoft SQL Server database, to a Microsoft Access database, to an Oracle database, to an XML file, to a SQLite database, or to a Vista DB database. Like the health monitoring system, ELMAH's architecture was built using the provider model, meaning that you can create and seamlessly integrate your own custom log source providers, if needed.

Step 2: Registering ELMAH's HTTP Module and Handler

While the `Elmah.dll` file contains the HTTP Modules and Handler needed to automatically log unhandled exceptions and to display error details from a web page, these must be explicitly registered in the web

application's configuration. The `ErrorLogModule` HTTP Module, once registered, subscribes to the `HttpApplication` 's `Error` event. Whenever this event is raised the `ErrorLogModule` logs the details of the exception to a specified log source. We'll see how to define the log source provider in the next section, "Configuring ELMAH." The `ErrorLogPageFactory` HTTP Handler factory is responsible for generating the markup when viewing the error log from a web page.

The specific syntax for registering HTTP Modules and Handlers depends upon the web server that is powering the site. For the ASP.NET Development Server and Microsoft's IIS version 6.0 and earlier, HTTP Modules and Handlers are registered in the `<httpModules>` and `<httpHandlers>` sections, which appear within the `<system.web>` element. If you are using IIS 7.0 then they need to be registered in the `<system.webServer>` element's `<modules>` and `<handlers>` sections. Fortunately, you can define the HTTP Modules and Handlers in *both* places regardless of the web server being used. This option is the most portable one as it allows the same configuration to be used in the development and production environments regardless of the web server being used.

Start by registering the `ErrorLogModule` HTTP Module and the `ErrorLogPageFactory` HTTP Handler in the `<httpModules>` and `<httpHandlers>` section in `<system.web>`. If your configuration already defines these two elements then simply include the `<add>` element for ELMAH's HTTP Module and Handler.

```
<?xml version="1.0"?>
<configuration>
  ...

  <system.web>
  ...

  <httpHandlers>
  ...

  <add verb="POST,GET,HEAD" path="elmah.axd" type="Elmah.ErrorLogPageFactory, Elmah" />
  </httpHandlers>

  <httpModules>
  ...

  <add name="ErrorLog" type="Elmah.ErrorLogModule, Elmah"/>
  </httpModules>

  ...
  </system.web>

  ...
</configuration>
```

Next, register ELMAH's HTTP Module and Handler in the `<system.webServer>` element. As before, if this element is not already present in your configuration then add it.

```
<?xml version="1.0"?>
<configuration>
  ...

  <system.webServer>
  <validation validateIntegratedModeConfiguration="false"/>
  <modules>
  ...

  <add name="Elmah.ErrorLog" type="Elmah.ErrorLogModule, Elmah" preCondition="managedHandler" />
  </modules>
  <handlers>
  ...

  <add name="Elmah" path="elmah.axd" verb="POST,GET,HEAD" type="Elmah.ErrorLogPageFactory, Elmah"
preCondition="integratedMode" />
  </handlers>
  </system.webServer>
</configuration>
```

By default, IIS 7 complains if HTTP Modules and Handlers are registered in the `<system.web>` section. The `validateIntegratedModeConfiguration` attribute in the `<validation>` element instructs IIS 7 to suppress such error messages.

Note that the syntax for registering the `ErrorLogPageFactory` HTTP Handler includes a `path` attribute, which is set to `elmah.axd`. This attribute informs the web application that if a request arrives for a page named `elmah.axd` then the request should be processed by the `ErrorLogPageFactory` HTTP Handler. We'll see the `ErrorLogPageFactory` HTTP Handler in action later in this tutorial.

Step 3: Configuring ELMAH

ELMAH looks for its configuration options in the website's `Web.config` file in a custom configuration section named `<elmah>`. In order to use a custom section in `Web.config` it must first be defined in the `<configSections>` element. Open the `Web.config` file and add the following markup to the `<configSections>`:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
  ...

  <sectionGroup name="elmah">
  <section name="security" requirePermission="false" type="Elmah.SecuritySectionHandler, Elmah"/>
  <section name="errorLog" requirePermission="false" type="Elmah.ErrorLogSectionHandler, Elmah" />
  <section name="errorMail" requirePermission="false" type="Elmah.ErrorMailSectionHandler, Elmah" />
  <section name="errorFilter" requirePermission="false" type="Elmah.ErrorFilterSectionHandler, Elmah"/>
  </sectionGroup>
  </configSections>

  ...
</configuration>
```

NOTE

If you are configuring ELMAH for an ASP.NET 1.x application then remove the `requirePermission="false"` attribute from the `<section>` elements above.

The above syntax registers the custom `<elmah>` section and its subsections: `<security>`, `<errorLog>`, `<errorMail>`, and `<errorFilter>`.

Next, add the `<elmah>` section to `Web.config`. This section should appear at the same level as the `<system.web>` element. Inside the `<elmah>` section add the `<security>` and `<errorLog>` sections like so:

```
<?xml version="1.0"?>
<configuration>
  ...

  <elmah>
  <security allowRemoteAccess="0" />

  <errorLog type="Elmah.SqlErrorLog, Elmah" connectionStringName="ReviewsConnectionString" />
  </elmah>

  ...
</configuration>
```

The `<security>` section's `allowRemoteAccess` attribute indicates whether remote access is allowed. If this value is set to 0, then the error log web page can only be viewed locally. If this attribute is set to 1 then the error log web page is enabled for both remote and local visitors. For now, let's disable the error log web page for remote visitors. We'll allow remote access later after we have an opportunity to discuss the security concerns of doing so.

The `<errorLog>` section defines the error log source, which dictates where the error details are recorded; it is similar to the `<providers>` section in the health monitoring system. The above syntax specifies the `SqlErrorLog` class as the error log source, which logs the errors to a Microsoft SQL Server database specified by the `connectionStringName` attribute value.

> **NOTE**
> ELMAH ships with additional error log providers that can be used to log errors to an XML file, a Microsoft Access database, an Oracle database, and other data stores. Refer to the sample `Web.config` file that is included with the ELMAH download for information on how to use these alternate error log providers.

Step 4: Creating the Error Log Source Infrastructure

ELMAH's `SqlErrorLog` provider logs error details to a specified Microsoft SQL Server database. The `SqlErrorLog` provider expects this database to have a table named `ELMAH_Error` and three stored procedures: `ELMAH_GetErrorsXml`, `ELMAH_GetErrorXml`, and `ELMAH_LogError`. The ELMAH download includes a file named `SQLServer.sql` in the `db` folder that contains the T-SQL for creating this table and these stored procedures. You'll need to run these statements on your database to use the `SqlErrorLog` provider.

Figures 1 and **2** show the Database Explorer in Visual Studio after the database objects needed by the `SqlErrorLog` provider have been added.

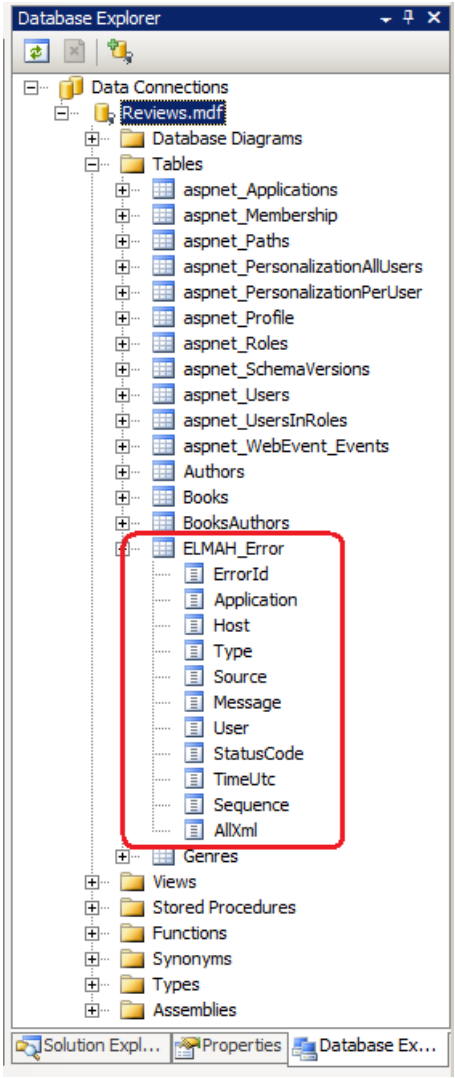


Figure 1: The `SqlErrorLog` Provider Logs Errors to the `ELMAH_Error` Table

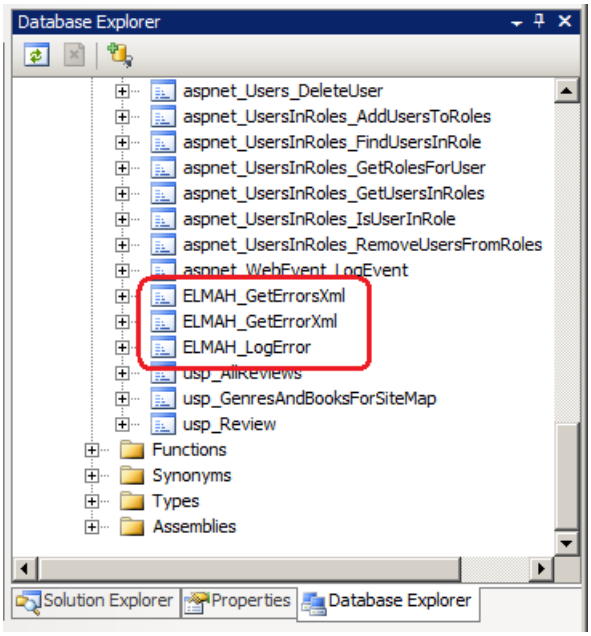


Figure 2: The `SqlErrorLog` Provider Uses Three Stored Procedures

ELMAH In Action

At this point we have added ELMAH to the web application, registered the `ErrorLogModule` HTTP Module and the `ErrorLogPageFactory` HTTP Handler, specified ELMAH's configuration options in `Web.config`, and added the

needed database objects for the `SqlErrorLog` error log provider. We are now ready to see ELMAH in action! Visit the Book Reviews website and visit a page that generates a runtime error, such as `Genre.aspx?ID=foo`, or a non-existent page, such as `NoSuchPage.aspx`. What you see when visiting these pages depends on the `<customErrors>` configuration and whether you're visiting locally or remotely. (Refer back to the *Displaying a Custom Error Page* tutorial for a refresher on this topic.)

ELMAH doesn't affect what content is shown to the user when an unhandled exception occurs; it just logs its details. This error log is accessible from the web page `elmah.axd` from the root of your website, such as `http://localhost/BookReviews/elmah.axd`. (This file does not physically exist in your project, but when a request comes in for `elmah.axd` the runtime dispatches it to the `ErrorLogPageFactory` HTTP Handler, which generates the markup sent back to the browser.)

> **NOTE**
>
> You can also use the `elmah.axd` page to instruct ELMAH to generate a test error. Visiting `elmah.axd/test` (as in, `http://localhost/BookReviews/elmah.axd/test`) causes ELMAH to throw an exception of type `Elmah.TestException`, which has the error message: " This is a test exception that can be safely ignored."

Figure 3 shows the error log when visiting `elmah.axd` from the development environment.

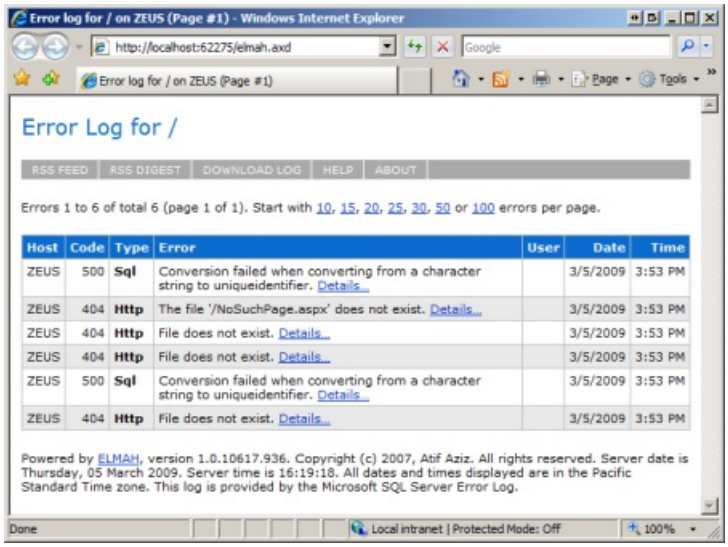


Figure 3: `Elmah.axd` Displays the Error Log from a Web Page
(Click to view full-size image)

The error log in **Figure 3** contains six error entries. Each entry includes the HTTP status code (404 or 500, for these errors), the type, the description, the name of the logged on user when the error occurred, and the date and time. Clicking the Details link displays a page that includes the same error message shown in the Error Details Yellow Screen of Death (see **Figure 4**) along with the values of the server variables at the time of the error (see **Figure 5**). You can also view the raw XML in which the error details are saved, which includes additional information such as the values in the HTTP POST header.

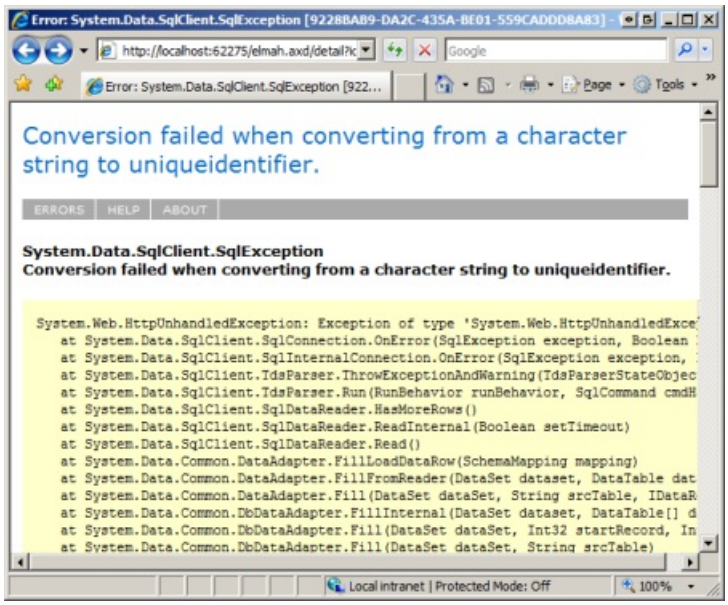


Figure 4: View the Error Details YSOD
(Click to view full-size image)

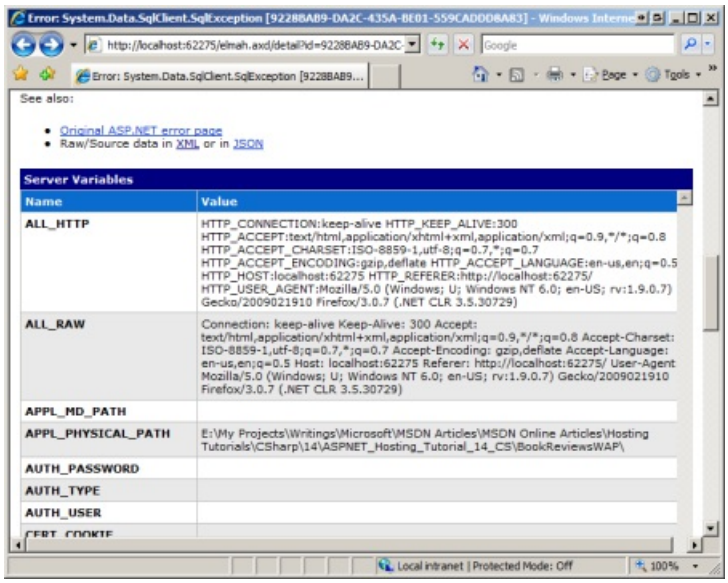


Figure 5: Explore the Values of the Server Variables Collection at the Time of the Error
(Click to view full-size image)

Deploying ELMAH to the production website entails:

- Copying the `Elmah.dll` file to the `Bin` folder on production,
- Copying the ELMAH-specific configuration settings to the `Web.config` file used on production, and
- Adding the error log source infrastructure to the production database.

We've explored techniques for copying files from development to production in previous tutorials. Perhaps the easiest way to get the error log source infrastructure on the production database is to use SQL Server Management Studio to connect to the production database and then execute the `SqlServer.sql` script file, which will create the needed table and stored procedures.

Viewing the Error Details Page on Production

After deploying your site to production, visit the production website and generate an unhandled exception. As in the development environment, ELMAH has no affect on the error page displayed when an unhandled exception occurs; instead, it merely logs the error. If you attempt to visit the error log page (`elmah.axd`) from the production environment, you will be greeted with the Forbidden page shown in **Figure 6**.

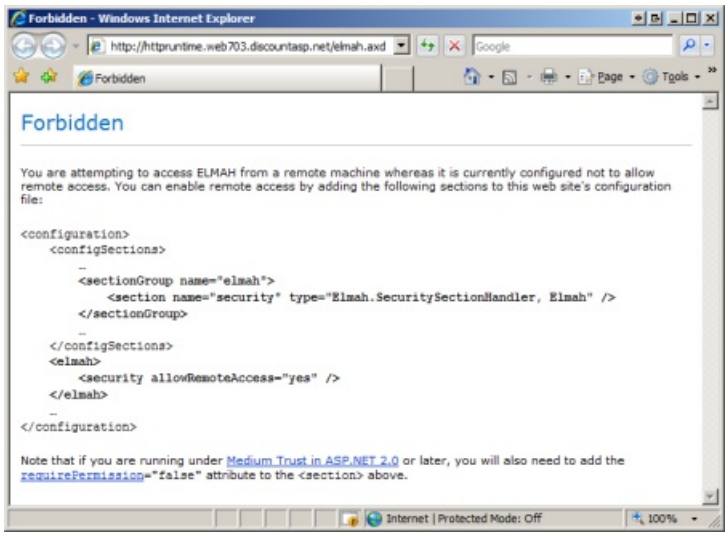


Figure 6: By Default, Remote Visitors Cannot View the Error Log Web Page
(Click to view full-size image)

Recall that in the ELMAH configuration's `<security>` section we set the `allowRemoteAccess` attribute to 0, which prohibits remote users from viewing the error log. It's important to prohibit anonymous visitors from viewing the error log, as the error details might reveal security vulnerabilities or other sensitive information. If you decide to set this attribute to 1 and enable remote access to the error log then it is important to lock down the `elmah.axd` path so that only authorized visitors can access it. This can be achieved by adding a `<location>` element to the `Web.config` file.

The following configuration permits only users in the Admin role to access the error log web page:

```
<?xml version="1.0"?>
<configuration>
  ...

  <elmah>
  <security allowRemoteAccess="1" />

  ...
  </elmah>

  ...

  <location path="elmah.axd">
  <system.web>
  <authorization>
  <allow roles="Admin" />
  <deny users="*" />
  </authorization>
  </system.web>
  </location>
</configuration>
```

NOTE

The Admin role and the three users in the system - Scott, Jisun, and Alice - were added in the *Configuring a Website That Uses Application Services* tutorial. Users Scott and Jisun are members of the Admin role. For more information on authentication and authorization, refer to my Website Security Tutorials.

The error log on the production environment can now be viewed by remote users; refer back to **Figures 3**, **4**, and **5** for screen shots of the error log web page. However, if an anonymous or non-Admin user attempts to view the error log page they are automatically redirected to the login page (`Login.aspx`), as **Figure 7** shows.

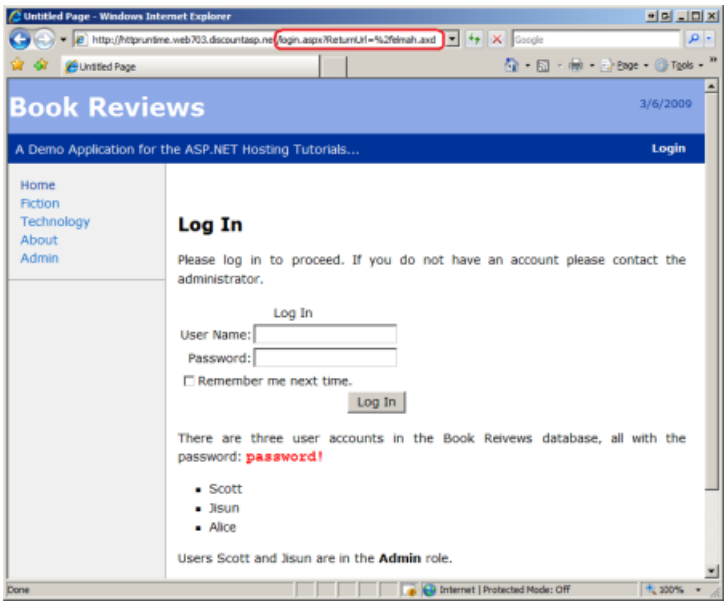


Figure 7: Unauthorized Users are Automatically Redirected to the Login Page
(Click to view full-size image)

Programmatically Logging Errors

ELMAH's `ErrorLogModule` HTTP Module automatically logs unhandled exceptions to the specified log source. Alternatively, you can log an error without having to raise an unhandled exception by using the `ErrorSignal` class and its `Raise` method. The `Raise` method is passed an `Exception` object and logs it as if that exception had been thrown and had reached the ASP.NET runtime without being handled. The difference, however, is that the request continues executing normally after the `Raise` method has been called, whereas a thrown, unhandled exception interrupts the request's normal execution and causes the ASP.NET runtime to display the configured error page.

The `ErrorSignal` class is useful in situations where there is some action that may fail, but its failure is not catastrophic to the overall operation being performed. For instance, a website may contain a form that takes the user's input, stores it in a database, and then sends the user an email informing them that they information was processed. What should happen if the information is saved to the database successfully, but there is an error when sending the email message? One option would be to throw an exception and send the user to the error page. However, this might confuse the user into thinking that the information they entered was not saved. Another approach would be to log the email-related error, but not alter the user's experience in any way. This is where the `ErrorSignal` class is useful.

```
// ... Save user's information to the database ...
...

// Attempt to send the user a confirmation email
try
{
    // ... Send an email ...
}
catch (Exception e)
{
    // Error in sending email. Log it!
    ErrorSignal.FromCurrentContext().Raise(e);
}
```

Error Notification Via Email

Along with logging errors to a database, ELMAH can also be configured to email error details to a specified recipient. This functionality is provided by the `ErrorMailModule` HTTP Module; therefore, you must register this HTTP Module in `Web.config` in order to send error details via email.

```
<?xml version="1.0"?>
<configuration>
  ...

  <system.web>
  ...

  <httpModules>
  ...

  <add name="ErrorLog" type="Elmah.ErrorLogModule, Elmah"/>
  <add name="ErrorMail" type="Elmah.ErrorMailModule, Elmah"/>
  </httpModules>
  </system.web>

  <system.webServer>
  <validation validateIntegratedModeConfiguration="false"/>
  <modules>
  ...

  <add name="Elmah.ErrorLog" type="Elmah.ErrorLogModule, Elmah" preCondition="managedHandler" />
  <add name="Elmah.ErrorMail" type="Elmah.ErrorMailModule" preCondition="managedHandler" />
  </modules>

  ...
  </system.webServer>
</configuration>
```

Next, specify information about the error email in the `<elmah>` element's `<errorMail>` section, indicating the email's sender and recipient, the subject, and whether the email is sent asynchronously.

```
<?xml version="1.0"?>
<configuration>
  ...

  <elmah>
  ...

  <errorMail from="support@example.com"
  to="support@example.com"
  subject="Book Reviews Runtime Error"
  async="true" />
  </elmah>

  ...
</configuration>
```

With the above settings in place, whenever a runtime error occurs ELMAH sends an email to support@example.com with the error details. ELMAH's error email includes the same information shown in the error details web page, namely the error message, the stack trace, and the server variables (refer back to **Figures 4** and **5**). The error email also includes the Exception Details Yellow Screen of Death content as an attachment (`YSOD.html`).

Figure 8 shows ELMAH's error email generated by visiting `Genre.aspx?ID=foo`. While **Figure 8** shows only the error message and stack trace, the server variables are included further down in the email's body.

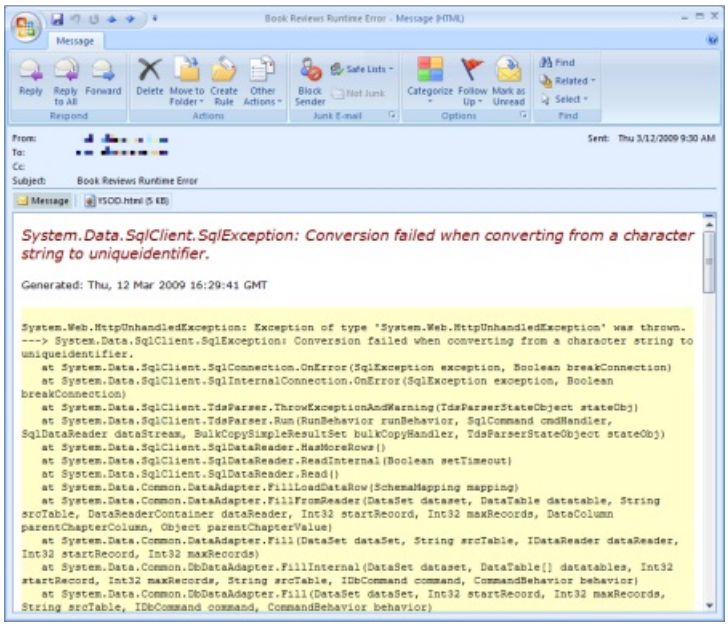


Figure 8: You Can Configure ELMAH To Send Error Details Via Email
(Click to view full-size image)

Only Logging Errors Of Interest

By default, ELMAH logs the details of every unhandled exception, including 404 and other HTTP errors. You can instruct ELMAH to ignore these or other types of errors using error filtering. The filtering logic is performed by ELMAH's `ErrorFilterModule` HTTP Module, which you'll need to register in `Web.config` in order to use the filtering logic. The rules for filtering are specified in the `<errorFilter>` section.

The following markup instructs ELMAH to not log 404 errors.

```
<?xml version="1.0"?>
<configuration>
  ...

  <elmah>
  ...

  <errorFilter>
  <test>
  <equal binding="HttpStatusCode" value="404" type="Int32" />
  </test>
  </errorFilter>
  </elmah>

  ...
</configuration>
```

> **NOTE**
>
> Don't forget, in order to use error filtering you must register the `ErrorFilterModule` HTTP Module.

The `<equal>` element inside the `<test>` section is referred to as an assertion. If the assertion evaluates to true then the error is filtered from ELMAH's log. There are other assertions available, including: `<greater>`, `<greater-or-equal>`, `<not-equal>`, `<lesser>`, `<lesser-or-equal>`, and so on. You can also combine assertions using the `<and>` and `<or>` Boolean operators. What's more, you can even include a simple JavaScript expression as an assertion, or write your own assertions in C# or Visual Basic.

For more information on ELMAH's error filtering capabilities, refer to the Error Filtering section in the ELMAH wiki.

Summary

ELMAH provides a simple, yet powerful mechanism for logging errors in an ASP.NET web application. Like Microsoft's health monitoring system, ELMAH can log errors to a database and can send the error details to a developer via email. Unlike the health monitoring system, ELMAH includes out of the box support for a wider range of error log data stores, including: Microsoft SQL Server, Microsoft Access, Oracle, XML files, and several others. Moreover, ELMAH offers a built-in mechanism for viewing the error log and details about a specific error from a web page, `elmah.axd`. The `elmah.axd` page can also render error information as an RSS feed or as a comma-separated value file (CSV), which you can read using Microsoft Excel. You can also instruct ELMAH to filter errors from the log using declarative or programmatic assertions. And ELMAH can be used with ASP.NET version 1.x applications.

Every deployed application should have some mechanism for automatically logging unhandled exceptions and sending notification to the development team. Whether this is accomplished using health monitoring or ELMAH is secondary. In other words, it doesn't really matter much whether you use health monitoring or ELMAH; evaluate both systems and then choose the one that best fits your needs. What is fundamentally important is that some mechanism be put in place to log unhandled exceptions in the production environment.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ELMAH - Error Logging Modules and Handlers
- ELMAH Project Page (source code, samples, wiki)
- Plugging ELMAH Into a Web Application to Catch Unhandled Exceptions (video)
- Security Error Log Pages
- Using HTTP Modules and Handlers to Create Pluggable ASP.NET Components
- Website Security Tutorials

PREVIOUS NEXT

Precompiling Your Website (C#)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Visual Studio offers ASP.NET developers two types of projects: Web Application Projects (WAPs) and Web Site Projects (WSPs). One of the key differences between the two project types is that WAPs must have the code explicitly compiled prior to deployment whereas the code in a WSP can be automatically compiled on the web server. However, it is possible to precompile a WSP prior to deployment. This tutorial explores the benefits of precompilation and shows how to precompile a website from within Visual Studio and from the command line.

Introduction

Visual Studio offers ASP.NET developers two different project types: Web Application Projects (WAP) and Web Site Projects (WSP). One of the key differences between these project types is that WAPs require *explicit compilation* whereas WSPs use *automatic compilation*, by default. With WAPs, you compile the web application's code into a single assembly, which is created in the website's `Bin` folder. Deployment entails copying the markup content (the `.aspx.ascx`, and `.master` files) in the project, along with the assembly in the `Bin` folder; the code-behind class files themselves do not need to be deployed. On the other hand, you deploy WSPs by copying both the markup pages and their corresponding code-behind classes to the production environment. The code-behind classes are compiled on demand on the web server.

> **NOTE**
>
> Refer back to the "Explicit Compilation Versus Automatic Compilation" section in the *Determining What Files Need to Be Deployed* tutorial for more background on the differences between the project models, explicit and automatic compilation, and how the compilation model affects deployment.

The automatic compilation option is simple to use. There's no explicit compilation step and only the files that have been modified need to be deployed, whereas explicit compilation necessitates deploying the changed markup pages and the just-compiled assembly. However, automatic deployment has two potential drawbacks:

- Because the pages must be automatically compiled when they are first visited, there can be a short but noticeable delay when an ASP.NET page is requested for the first time after being deployed.
- Automatic compilation requires that both the declarative markup and source code be present on the web server. This can be an unattractive option if you plan on selling the web application to customers who will install it on their web servers.

If either of the two above shortcomings are deal breakers, you can either switch to the WAP model or *precompile* the WSP prior to deployment. This tutorial examines the precompilation options best suited for a hosted website and walks through the precompilation process and deployment of a precompiled website.

An Overview of ASP.NET Code Generation and Compilation

Before we look at the available precompilation options, let's first talk about the code generation and compilation that occurs when an ASP.NET page is requested for the first time since it's been created or last updated. As you know, ASP.NET pages are composed of two portions: declarative markup in the `.aspx` file; and a source code

portion, typically in a separate code-behind class file (`.aspx.cs`). The steps performed by the runtime when an ASP.NET page is requested depends on the application's compilation model.

With WAPs, the pages' source code must be explicitly compiled into a single assembly before being deployed. During deployment, this assembly and the various markup pages are copied to the production environment. When a request arrives to the web server for an ASP.NET page, the runtime creates an instance of the page's code-behind class and invokes its `ProcessRequest` method, which starts the page lifecycle and, ultimately, generates the page's content, which is returned to the requestor. The runtime can work with the ASP.NET page's code-behind class because the code-behind class was already compiled into an assembly prior to deployment.

With WSPs and automatic compilation, there is no explicit compilation step prior to deployment. Instead, deployment involves copying both the declarative and the source code content to the production environment. When a request arrives to the web server for an ASP.NET page for the first time since the page has been created or last updated, the runtime must first compile the code-behind class into an assembly. This compiled assembly is saved in the folder `%WINDIR%\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files`, although the location of this folder can be customized via the `<compilation tempDirectory="" />` element of `<system.web>`, usually in `Web.config`. Because the assembly is saved to disk, it does not need to be recompiled on subsequent requests to the same page.

> **NOTE**
> As you would expect, there is a slight delay when requesting a page for the first time (or for the first time since it's been changed) in a site that uses automatic compilation as it takes a moment for the server to compile the page's code and save the resulting assembly to disk.

In short, with explicit compilation you are required to compile the website's source code before deployment, saving the runtime from having to perform that step. With automatic compilation the runtime handles the compilation of the pages' source code, but with a slight initialization cost for the first visit to the page since it was created or last updated.

But what about the declarative part of ASP.NET pages (the `.aspx` file)? It's obvious that there's a relationship between the `.aspx` files and the code in their code-behind classes, as the Web controls defined in the declarative markup are accessible in the code. It's also obvious that the content in the `.aspx` files greatly influences the rendered markup generated by the page. So how does the runtime work with the text, HTML, and Web control syntax defined in the `.aspx` file to generate the requested page's rendered content?

I don't want to get too sidetracked on the low-level implementation details, which vary between WAPs and WSPs, but in a nutshell the runtime automatically generates a class file that contains the various Web controls as protected members and methods. This generated file is implemented as a *partial class* to the corresponding code-behind class. (Partial classes allow for the contents of a single class to be spread across multiple files.) Therefore, the code-behind class is defined in two places: in the `.aspx.cs` file that you create, and in this auto-generated class created by the runtime. This auto-generated class is stored in the `%WINDIR%\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files` folder.

The important take away here is that for an ASP.NET page to be rendered by the runtime both its declarative and source code portions must be compiled into an assembly. With WAPs, the source code is explicitly compiled into an assembly prior to deployment, but the declarative markup must still be converted into code and compiled by the runtime on the web server. With WSPs using automatic compilation, both the source code and the declarative markup need to be compiled by the web server.

It is possible to use explicit compilation with the WSP model. You can explicitly compile the source code portion, like with the WAP model. What's more, you can also compile the declarative markup.

Precompilation Options

The .NET Framework ships with an ASP.NET compilation tool (`aspnet_compiler.exe`) that enables you to compile the source code (and even the content) of an ASP.NET application built using the WSP model. This tool was released with the .NET Framework version 2.0 and is located in the `%WINDIR%\Microsoft.NET\Framework\v2.0.50727` folder; it can be used from the command line or launched from within Visual Studio via the Build menu's Publish Web Site option.

The compilation tool provides two general forms of compilation: in-place precompilation and precompilation for deployment. With in-place precompilation you run the `aspnet_compiler.exe` tool from the command-line and specify the path to the virtual directory or physical path of a website that resides on your computer. The compilation tool then compiles each ASP.NET page in the project, storing the compiled version in the `%WINDIR%\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files` folder just like if the pages had each been visited for the first time from a browser. In-place precompilation can speed up the first request made to newly deployed ASP.NET pages on your site because it alleviates the runtime from needing to perform this step. However, in-place precompilation is not useful for the majority of hosted websites because it requires that you are able to run programs from the web server's command-line. In shared hosting environments this level of access is not permitted.

> **NOTE**
>
> For more information on in-place precompilation, check out How To: Precompile ASP.NET Web Sites and Precompilation in ASP.NET 2.0.

Instead of compiling the pages in the website to the `Temporary ASP.NET Files` folder, precompilation for deployment compiles the pages to a directory of your choosing and in a format that can be deployed to the production environment.

There are two flavors of precompilation for deployment that we explore in this tutorial: precompilation with an updatable user interface, and precompilation with a non-updatable user interface. Precompilation with an updatable user interface leaves the declarative markup in the `.aspx` , `.ascx` , and `.master` files, thereby allowing a developer to view and, if desired, modify the declarative markup on the production server. Precompilation with a non-updatable user interface generates `.aspx` pages that are void of any content and removes `.ascx` and `.master` files, thereby hiding the declarative markup and prohibiting a developer from changing it from the production environment.

Precompiling for Deployment With an Updatable User Interface

The best way to understand precompilation for deployment is to see an example in action. Let's precompile the Book Reviews WSP for deployment using an updatable user interface. The ASP.NET compilation tool can be invoked from Visual Studio's Build menu or from the command-line. This section examines using the tool from within Visual Studio; the "Precompiling from the Command Line" section looks at running the compiler tool from the command line.

Open the Book Review WSP in Visual Studio, go to the Build menu, and select the Publish Web Site menu option. This launches the Publish Web Site dialog box (see **Figure 1**), where you can specify the target location, whether or not the precompiled site's user interface is updatable, and other compiler tool options. The target location can be a remote web server or FTP server, but for now choose a folder on your computer's hard drive. Because we want to precompile the site with an updatable user interface, leave the "Allow this precompiled site to be updatable" checkbox checked and click OK.

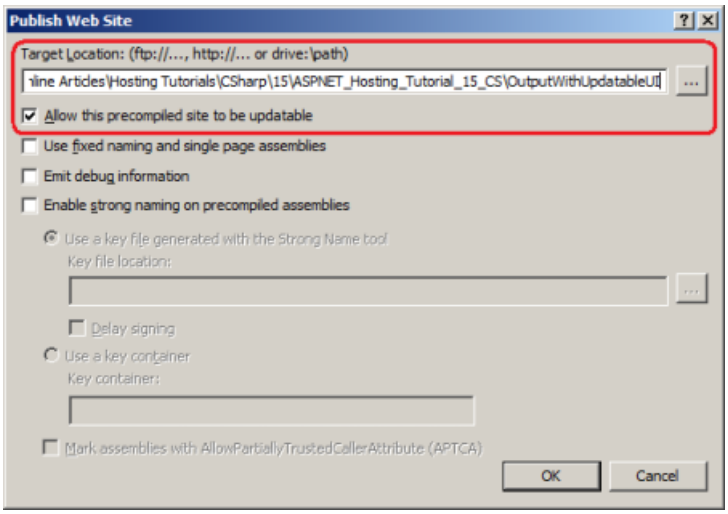


Figure 1: The ASP.NET Compilation Tool Will Precompile Your Website to the Specified Target Location (Click to view full-size image)

> **NOTE**
>
> The Publish Web Site option in the Build menu is not available in Visual Web Developer. If you are using Visual Web Developer you will need to use the command-line version of the ASP.NET compilation tool, which is covered in the "Precompiling from the Command Line" section.

After precompiling the website, navigate to the target location you entered in the Publish Web Site dialog box. Take a moment to compare the contents of this folder with the contents of your website. **Figure 2** shows the Book Reviews website folder. Note that it contains both `.aspx` and `.aspx.cs` files. Also, note that the `Bin` directory includes only one file, `Elmah.dll`, which we added in the preceding tutorial

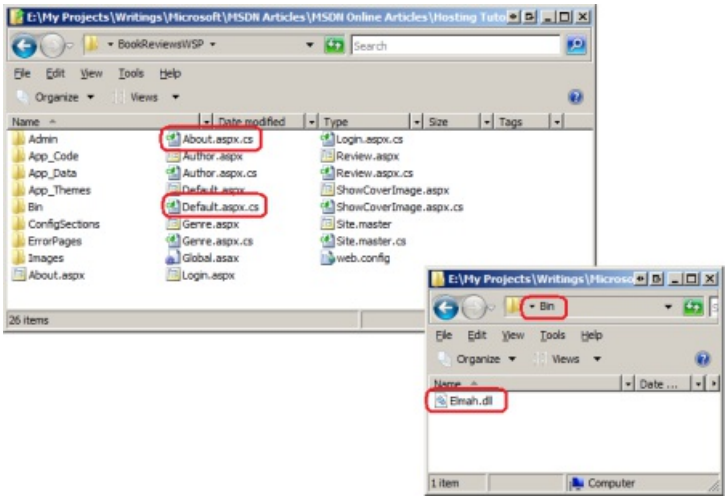


Figure 2: The Project Directory Contains `.aspx` and `.aspx.cs` Files; the `Bin` Folder Includes Just `Elmah.dll` (Click to view full-size image)

Figure 3 shows the target location folder whose contents were created by the ASP.NET compilation tool. This folder does not contain any code-behind files. Moreover, this folder's `Bin` directory includes several assemblies and two `.compiled` files in addition to the `Elmah.dll` assembly.

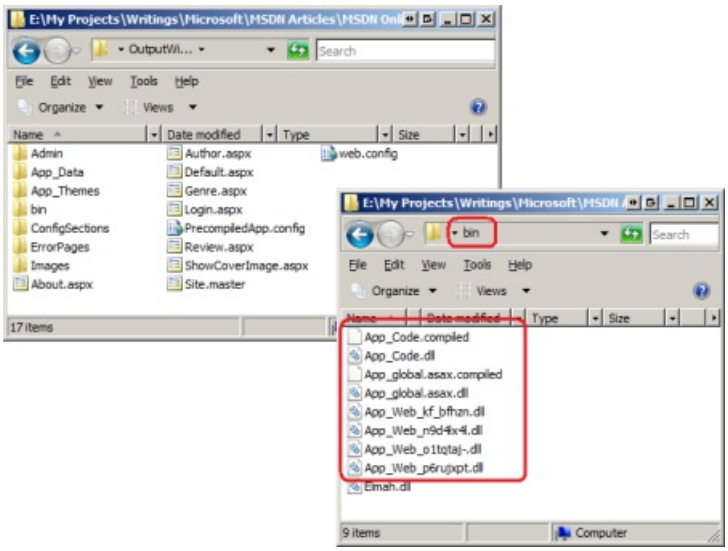


Figure 3: The Target Location Folder Includes the Files for Deployment
(Click to view full-size image)

Unlike explicit compilation in WAPs, the precompilation for deployment process does not create one assembly for the entire site. Instead, it batches together several pages into each assembly. It also compiles the `Global.asax` file (if present) into its own assembly, as well as any classes in the `App_Code` folder. The files that hold the declarative markup for ASP.NET web pages, User Controls, and master pages (`.aspx` , `.ascx` , and `.master` files, respectively) are copied as-is to the target location directory. Likewise, the `Web.config` file is copied straight over, along with any static files, such as images, CSS classes, and PDF files. For a more formal description of how the compilation tool handles various file types, refer to File Handling During ASP.NET Precompilation.

> **NOTE**
>
> You can instruct the compilation tool to create one assembly per ASP.NET page, User Control, or master page by checking the "Used fixed naming and single page assemblies" checkbox from the Publish Web Site dialog box. Having each ASP.NET page compiled into its own assembly allows for more fine-grained control over deployment. For example, if you updated a single ASP.NET web page and needed to deploy that change, you need only deploy that page's `.aspx` file and associated assembly to the production environment. Consult How To: Generate Fixed Names with the ASP.NET Compilation Tool for more information.

The target location directory also contains a file that was not part of the precompiled web project, namely `PrecompiledApp.config`. This file informs the ASP.NET runtime that the application was precompiled and whether it was precompiled with an updatable or noon-updatable UI.

Finally, take a moment to open one of the `.aspx` files in the target location using Visual Studio or your text editor of choice. When precompiling for deployment with an updatable user interface, the ASP.NET pages in the target location directory contain the exact same markup as the corresponding files in the website.

Precompiling for Deployment With a Non-Updatable User Interface

The ASP.NET compiler tool can also be used to precompile a site for deployment with a non-updatable UI. Precompiling the site with a non-updatable UI works much like precompiling with an updatable UI, the key difference being that the ASP.NET pages, User Controls, and master pages in the target directory are stripped of their markup. To precompile a website for deployment with a non-updatable UI, choose the Publish Web Site option from the Build menu, but uncheck the "Allow this precompiled site to be updatable" option (see **Figure 4**).

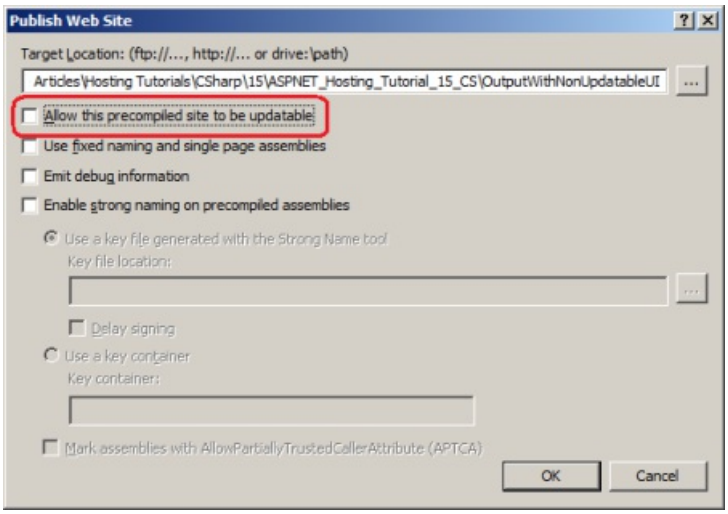


Figure 4: Uncheck the "Allow this precompiled site to be updatable" Option To Precompile With a Non-Updatable UI
(Click to view full-size image)

Figure 5 shows the target location folder after precompiling with a non-updatable user interface.

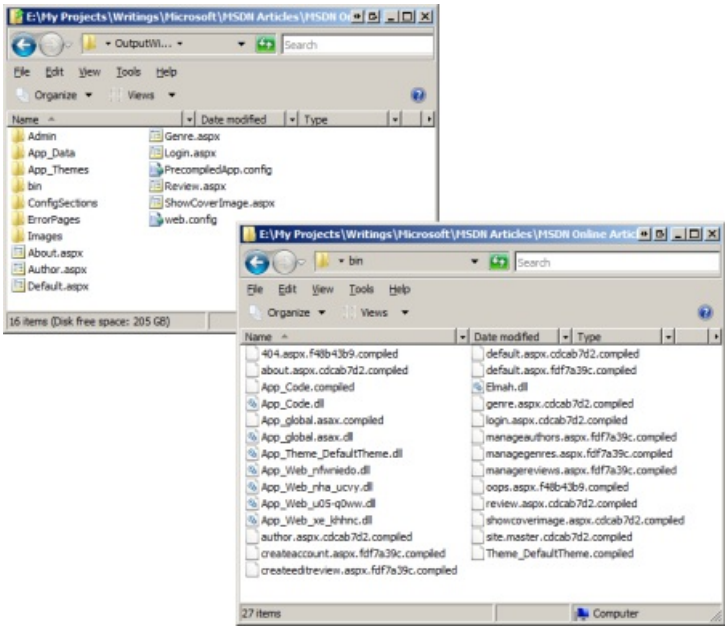


Figure 5: The Target Location Folder for Deployment With a Non-Updatable UI
(Click to view full-size image)

Compare **Figure 3** to **Figure 5**. While the two folders may look identical, note that the non-updatable UI folder lacks the master page, `Site.master`. And while **Figure 5** includes the various ASP.NET pages, if you view the contents of these files you'll see that they've been stripped of their declarative markup and replaced with the placeholder text: "This is a marker file generated by the precompilation tool, and should not be deleted!"

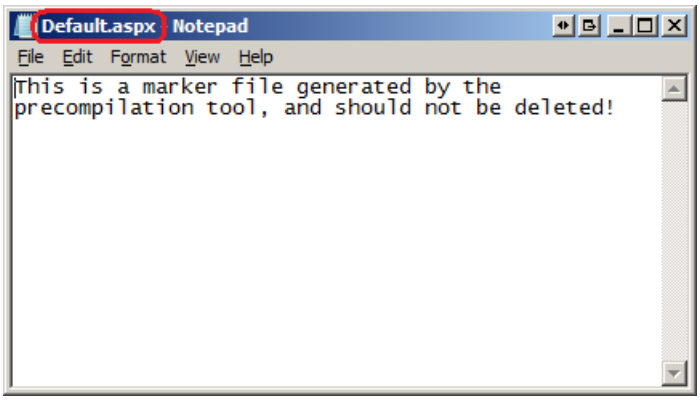


Figure 5: The Declarative Markup Has Been Removed from the ASP.NET Pages

The `Bin` folders in **Figures 3** and **5** differ more substantially. In addition to the assemblies, the `Bin` folder in **Figure 5** includes a `.compiled` file for each ASP.NET page, User Control, and master page.

Precompiling a site with a non-updatable UI is useful in situations where you do not want the ASP.NET pages' contents to be modified by the person or company who installs or manages the website in the production environment. If you build an ASP.NET web application that you sell to customers to install on their own web servers, you may want to ensure that they do not modify the look and feel of your site by directly editing the `.aspx` pages you ship them. By precompiling your website with a non-updatable UI, you ship the placeholder `.aspx` pages as part of the installation, thereby preventing your customers from examining or modifying their content.

Precompiling from the Command Line

Behind the scenes, Visual Studio's Publish Web Site dialog box invokes the ASP.NET compilation tool (`aspnet_compiler.exe`) to precompile the website. Alternatively, you can invoke this tool from the command line. In fact, if you use Visual Web Developer then you will need to run the compiler tool from the command line, as Visual Web Developer's Build menu does not include the Publish Web Site option.

To use the compiler tool from the command line, start by dropping to the command line and navigating to the framework directory, `%WINDIR%\Microsoft.NET\Framework\v2.0.50727`. Next, enter the following statement into the command line:

```
aspnet_compiler -p "physical_path_to_app" -v / -f -u "target_location_folder"
```

The above command launches the ASP.NET compiler tool (`aspnet_compiler.exe`) and, via the `-p` switch, instructs it to precompile the website rooted at *physical_path_to_app*; this value will be something like `C:\MySites\BookReviews`, and should be delimited by quotation marks.

The `-v` switch specifies the virtual directory of the site. If your site is registered as the default website in the IIS metabase then you can omit the `-p` switch and just specify the virtual directory of the application. If you use the `-p` switch, the value proceeding the `-v` switch indicates the root of the website, and is used to resolve application-root references. For instance, if you specify a value of `-v /MySite` then references in the application to `~/path/file` will be resolved as `~/MySite/path/file`. Because the Book Reviews site is located at the root directory at my web hosting company I have used the switch `-v /`.

The `-f` switch, if present, instructs the compilation tool to overwrite the *target_location_folder* directory if it already exists. If you omit the `-f` switch and the target location folder already exists, the compilation tool will quit with the error: "error ASPRUNTIME: The target directory is not empty. Please delete it manually or choose a different target."

The `-u` switch, if present, informs the tool to create an updatable user interface. Omit this switch to precompile the site with a non-updatable user interface.

Lastly, the *target_location_folder* is the physical path to the target location directory; this value will be something like `C:\MySites\Output\BookReviews`, and should be delimited by quotation marks.

Deploying the Precompiled Website

At this point we have seen how to use the ASP.NET compilation tool to precompile a website using both the updatable and non-updatable user interface options. However, our examples thus far have precompiled the website to a local folder, and not to the production environment. The good news is that deploying the precompiled website is a breeze and can be done through Visual Studio or through some other file copy mechanism, such as from a stand-alone FTP client.

The Publish Web Site dialog box (first shown in **Figure 1**) has a target location option, which indicates where the

precompiled website files are copied to. This location can be a remote web server or FTP server. Entering a remote server into this textbox precompiles and deploys the website to the specified server in one step. Alternatively, you can precompile the website to a local folder and then manually copy the contents of that folder to the production environment via FTP or some other approach.

Having the precompiled website automatically deployed via Visual Studio's Publish Web Site dialog box is helpful for simple sites where there are no configuration differences between the development and production environments. However, as noted in the *Common Configuration Differences Between Development and Production* tutorial it is not uncommon for such differences to exist. For example, the Book Reviews web application uses a different database in the production environment than in the development environment. When Visual Studio publishes the website to a remote server it blindly copies up the configuration file information in the development environment.

For sites with configuration differences between the development and production environments it may be best to precompile the site to a local directory, copy over the production-specific configuration files, and then copy the contents of the precompiled output to production.

For a refresher on copying files from the development environment to the production environment refer to the *Deploying Your Website Using an FTP Client* and *Deploying Your Website Using Visual Studio* tutorials.

Summary

ASP.NET supports two modes of compilation: automatic and explicit. As discussed in previous tutorials, Web Application Projects (WAPs) use explicit compilation whereas Web Site Projects (WSPs) use automatic compilation, by default. However, it is possible to explicitly compile a WSP prior to deployment by using the ASP.NET compilation tool.

This tutorial focused on the compilation tool's Precompilation for Deployment support. When precompiling for deployment, the compilation tool creates a target location folder, compiles the specified web application's source code, and copies these compiled assemblies and the content files into the target location folder. The compilation tool can be configured to create an updatable or non-updatable user interface. When precompiling with a non-updatable user interface option, the declarative markup in the content files is removed. In a nutshell, precompilation permits you to deploy your Web Site Project-based application without including any source code files and with the declarative markup removed, if desired.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Web Site Precompilation
- Codebehind and Compilation in ASP.NET 2.0
- Precompilation in ASP.NET
- Precompiled Site Options in ASP.NET

PREVIOUS NEXT

Users and Roles On The Production Website (C#)

2/20/2020 • 9 minutes to read • Edit Online

by Scott Mitchell

Download PDF

The ASP.NET Website Administration Tool (WSAT) provides a web-based user interface for configuring Membership and Roles settings and for creating, editing, and deleting users and roles. Unfortunately, the WSAT only works when visited from localhost, meaning that you cannot reach the production website's Administration Tool through your browser. The good news is that there are workarounds that make it possible to manage users and roles on production. This tutorial looks at these workarounds and others.

Introduction

ASP.NET 2.0 introduced a number of *application services*, which are a suite of building block services that you can add to your web application. We added the Membership and Roles services to the Book Reviews website back in the *Configuring a Website That Uses Application Services* tutorial. The Membership service facilitates creating and managing user accounts; the Roles service offers an API for categorizing users into groups. The Book Reviews site has three user accounts - Scott, Jisun, and Alice - and a single role, Admin, with Scott and Jisun in the Admin role.

ASP.NET's application services are not tied to a specific implementation. Instead, you instruct the application services to use a particular *provider*, and that provider implements the service using a particular technology. We configured the Book Reviews web application to use the `SqlMembershipProvider` and `SqlRoleProvider` providers for the Membership and Roles services. These two providers store user account and role information in a SQL Server database and are the most commonly used providers for Internet-based web applications hosted at a web hosting company.

A common challenge for developers using the Membership and Roles services is managing the users and roles on the production environment. How do you delete a user account from the production website, add a new role, or add an existing user to an existing role? This tutorial explores different techniques for managing users and roles on the production website.

Using the ASP.NET Web Site Administration Tool

ASP.NET includes a Web Site Administration Tool (WSAT) that makes it easy to create and manage user accounts and roles and to specify user- and role-based authorization rules. To use the WSAT, click the ASP.NET Configuration icon in the Solution Explorer, or go to the Website or Project menu and choose the ASP.NET Configuration option. Either approach launches a web browser and points it to the WSAT at an address like:

```
http://localhost:portNumber/asp.netwebadminfiles/default.aspx?applicationPhysicalPath=pathToApplication
```

The WSAT is divided into three sections:

- **Security** - manage users, roles, and authorization rules.
- **ApplicationConfiguration** - manage the <appSettings> and SMTP settings from here. You can also take the application offline and manage debugging and tracing settings from here, as well as specify the default custom error page.
- **ProviderConfiguration** - configure the providers used by the application services.

The Security section (shown in **Figure 1**) includes links for creating new users, managing users, creating and managing roles, and creating and managing access rules. From here you can add a new role to the system, delete

an existing user, or add or remove roles from a particular user account.

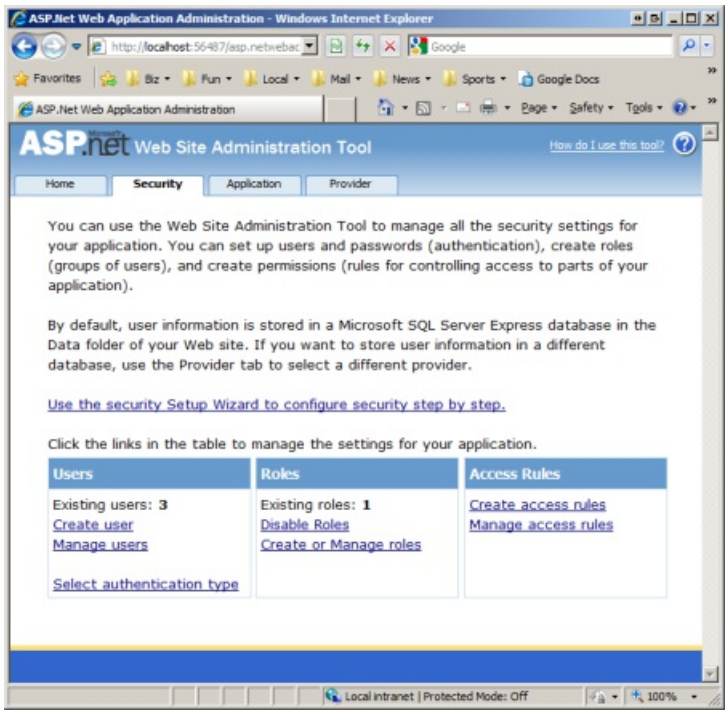


Figure 1: The WSAT Security Section Includes Options for Managing Users and Roles
(Click to view full-size image)

Unfortunately, the WSAT is only accessible locally. You cannot visit the WSAT on your remote production website; if you visit `www.yoursite.com/asp.netwebadminfiles/default.aspx` you get a 404 Not Found response. The code that powers the WSAT uses the `Membership` and `Roles` classes in the .NET Framework to create, edit, and delete users and roles. These classes consult the web application's configuration information to determine what provider to use; back in the *Configuring a Website That Uses Application Services* tutorial we setup the Book Reviews website to use the `SqlMembershipProvider` and `SqlRoleProvider` providers. This entailed adding `<membership>` and `<roleManager>` sections to `Web.config`.

```
<?xml version="1.0"?>
<configuration>
  ...

  <connectionStrings configSource="ConfigSections\databaseConnectionStrings.config" />

  <system.web>
  ...

  <membership defaultProvider="ReviewMembership">
  <providers>
  <clear />

  <add type="System.Web.Security.SqlMembershipProvider"
  name="ReviewMembership"
  connectionStringName="ReviewsConnectionString"
  applicationName="BookReviews" />
  </providers>
  </membership>

  <roleManager enabled="true" defaultProvider="ReviewRole">
  <providers>
  <clear />

  <add type="System.Web.Security.SqlRoleProvider"
  name="ReviewRole"
  connectionStringName="ReviewsConnectionString"
  applicationName="BookReviews" />
  </providers>
  </roleManager>
  </system.web>

  ...
</configuration>
```

Note that the `<membership>` and `<roleManager>` sections reference the `SqlMembershipProvider` and `SqlRoleProvider` providers in their `type` attribute, respectively. These providers store the user and role information in a specified SQL Server database. The database used by these providers is specified by the `connectionStringName` attribute, `ReviewsConnectionString`, which is defined in the `~/ConfigSections/databaseConnectionStrings.config` file. Recall that the `databaseConnectionStrings.config` file in the development environment contains the connection string to the development database whereas the `databaseConnectionStrings.config` file on production contains the connection string to the production database.

In a nutshell, the WSAT must be accessed locally through the development environment, and it works with the user and role information in the database specified in the `databaseConnectionStrings.config` file. Consequently, if we change the connection string information in the `databaseConnectionStrings.config` file on the development environment we can use the WSAT locally to manage users and roles in the production environment.

To illustrate this functionality, open the `databaseConnectionStrings.config` file in Visual Studio on the development environment and replace the development database connection string with the production database connection string. Then launch the WSAT, go the Security tab, and add a new user named Sam with password "password!" (less the quotation marks). **Figure 2** shows the WSAT screen when creating this account.

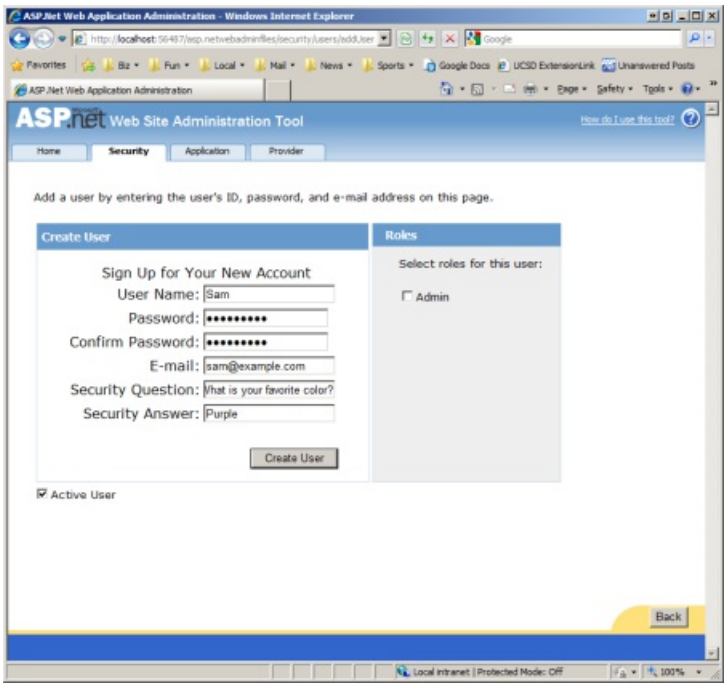


Figure 2: Create a New User Named Sam In the Production Environment
(Click to view full-size image)

Because we changed the connection string in `databaseConnectionStrings.config` to point to the production database server, Sam was added as a user in the production environment. To verify this, change the connection string in the `databaseConnectionStrings.config` file back to the development database and then visit the `Login.aspx` page in the development environment. Try to sign in as Sam (see **Figure 3**).

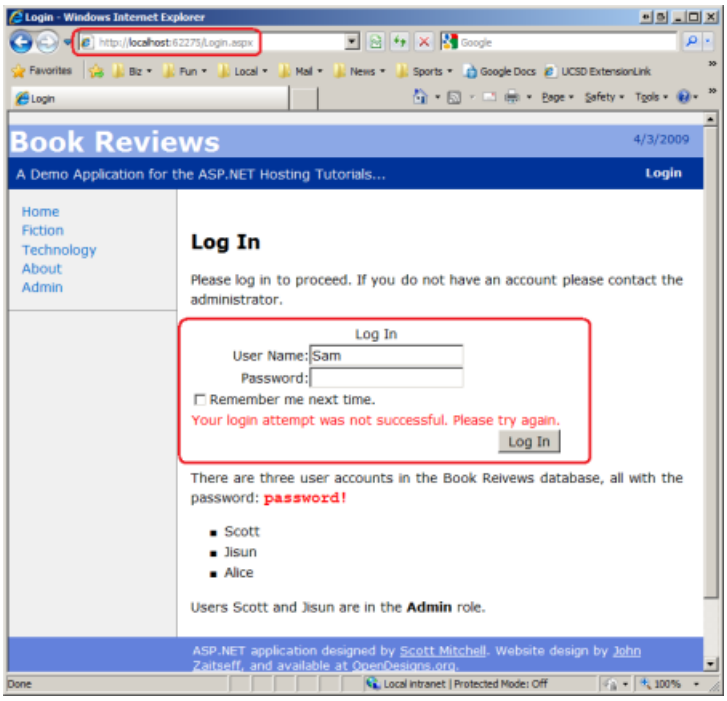


Figure 3: You Cannot Sign In As Sam in the Development Environment
(Click to view full-size image)

You cannot sign in as Sam in the development environment because the user account information does not exist in the local database. Rather, is was added to the production database. To verify this, view the contents of the `aspnet_Users` table in both the development and production databases. In the development environment there should be only three records for users Scott, Jisun, and Alice. However, the `aspnet_Users` table in the production database has four records: Scott, Jisun, Alice, and Sam. Consequently, Sam can sign in through the website in production, but not through the development environment.

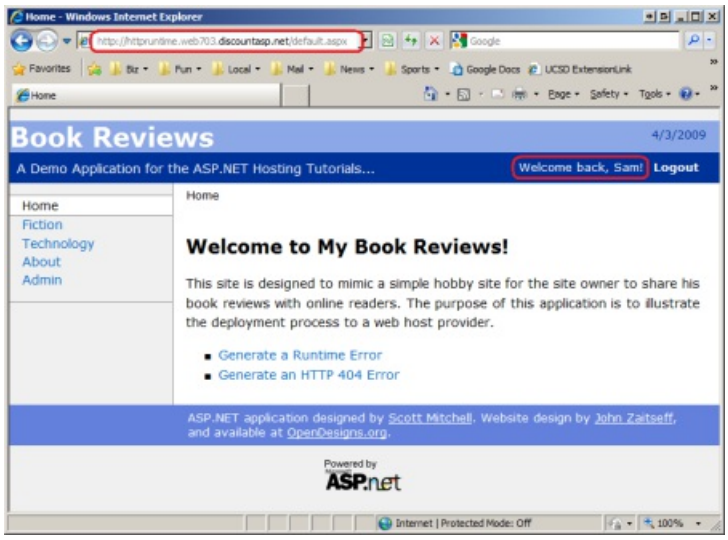


Figure 4: Sam Can Sign In On the Production Website
(Click to view full-size image)

> **NOTE**
>
> Don't forget to change the connection string in the `databaseConnectionStrings.config` file back to the development database's connect string when you're done working with the WSAT otherwise you will be working with production data when testing the site through the development environment. Also keep in mind that while the technique we just discussed allows us to use the WSAT to remotely manage users and roles, changes to any of the other WSAT configuration options (access rules, SMTP settings, debugging and tracing settings, and so on) modify the `Web.config` file. Consequently, any changes made to the settings apply to the development environment and not to the production environment.

Creating Custom User and Role Management Web Pages

The WSAT provides an out of the box system for managing users and roles, but can only be launched locally and requires making changes to the connection string information in order to manage the users and roles on production. Most websites that support user accounts also include a number of user and role administration web pages that enable administrators to manage users and roles from pages within the site. Such web-based administration pages make it much easier to manage users and roles and are essential for sites where there may be many administrators or administrators that do not have access to or the technical background to use Visual Studio to launch the WSAT.

ASP.NET includes a number of built-in Login-related Web controls that make implementing many of these administrative web pages as easy as drag and drop. For example, you can create a page for administrators to create a new user account by dragging the CreateUserWizard control onto the page and setting a few properties. In fact, the page for creating users in the WSAT shown in **Figure 2** uses the same CreateUserWizard control that you can add to your pages. Furthermore, the Membership and Roles services' functionalities are available programmatically through the `Membership` and `Roles` classes in the .NET Framework. With these classes you can write code to create, edit, and delete users and roles, as well as to add or remove users to roles, to determine what users are in what roles, and to perform other user- and role-related tasks.

In the *Configuring a Website That Uses Application Services* tutorial I added a page to the `Admin` folder named `CreateAccount.aspx`. This page allows an administrator to add a new user account to the site and to specify whether or not the newly created user is in the Admin role (see **Figure 5**).

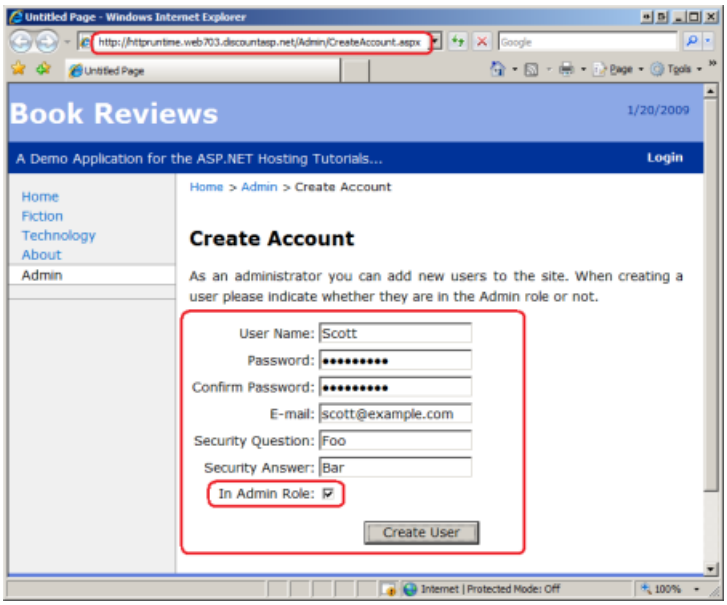


Figure 5: Administrators Can Create New User Accounts
(Click to view full-size image)

For a more detailed look at building user and role administration pages, along with step-by-step instructions on using the `Membership` and `Roles` classes and the Login-related ASP.NET Web controls, be sure to read my Website Security Tutorials. There you'll find guidance on how to build web pages for creating new accounts, creating and managing roles, assigning users to roles, and other common administrative tasks.

To implement WSAT-like functionality on the production website you can always build your own series of web pages that implement the WSAT's features. To help get started, check out the WSAT source code, which is located in the folder `%WINDIR%\Microsoft.NET\Framework\v2.0.50727\ASP.NETWebAdminFiles`. Another option is to use Dan Clem's WSAT alternative, which he shares in his article, Rolling Your Own Web Site Administration Tool. Dan walks readers through the process of building a custom WSAT-like tool, includes his application's source code for download (in C#), and gives step-by-step instructions for adding his custom WSAT to a hosted website.

Summary

The ASP.NET Web Site Administration Tool (WSAT) can be used in tandem with the Membership and Roles application services to manage user and role information for your website. Unfortunately, the WSAT is only accessible locally and cannot be visited from your production website. However, by changing the connection string in the development environment to point to the production database you can use the WSAT to manage the users and roles on the production website.

While the WSAT approach affords a quick and easy way to manage users and roles, it necessitates launching the WSAT from Visual Studio as well as temporary changes to the connection string information. The WSAT offers a quick way to manage users and roles on production, but is cumbersome and does not work well for websites with multiple administrators or with administrators who do not have or are not familiar with Visual Studio and the WSAT. For these reasons, most websites that support user accounts include a set of administrative web pages. Such a set of web pages eliminates the need for the WSAT and used by various administrative users from any computer.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Examining ASP.NET's Membership, Roles, and Profile
- Rolling Your Own Web Site Administration Tool
- Web Site Administration Tool Overview

- Website Security Tutorials

PREVIOUS NEXT

ASP.NET Hosting Options (VB)

2/20/2020 • 8 minutes to read • Edit Online

by Scott Mitchell

Download PDF

ASP.NET web applications are typically designed, created, and tested in a local development environment and need to be deployed to a production environment once it is ready for release. This tutorial provides a high-level overview of the deployment process and serves as an introduction to this tutorial series.

Introduction

Web applications are typically designed, created, and tested in a development environment that is accessible only to the programmers working on the site. Once the application is ready to be released, it is moved to a production environment where the site can be accessed by anyone on the Internet. This deployment process introduces a number of challenges:

- A production environment must exist and be properly setup before an ASP.NET application can be deployed; moreover, the production environment must be kept up to date with the latest security patches.
- The correct set of markup files, code files, and support files must be copied from the development environment to the production environment. For data-driven applications, this might require copying the database schema and/or data, as well.
- There may be configuration differences between the two environments. The database connection string or email server used in the development environment will likely be different than the production environment. What's more, the behavior of the application may depend on the environment. For example, when an error occurs in development the error's details can be displayed on screen, but when an error occurs in production, a user-friendly error page should be displayed instead, and the error details emailed to the developers.

To obviate the first challenge - setting up and maintaining a production environment - many individuals and businesses outsource their production environments to *web hosting providers*. A web hosting provider is a company that manages the production environment on your behalf. There are countless web host providers, each with varying prices and service levels; see the "Finding a Web Host Provider" section for tips on locating such a service provider.

This is the first in a series of tutorials that look at the steps involved in deploying an ASP.NET web application to a production environment managed by a web host provider. Over the course of these tutorials we will examine:

- What files need to be deployed to the web host provider.
- Tools for streamlining the deployment process.
- How to deploy a database.
- Tips for deploying a database that uses the SQL-based Membership and Roles provider, along with ways to mimic the Website Administration Tool in a production environment.
- Strategies for smoothly updating the database in production with changes made during development.
- Techniques for logging errors that occur on production, and ways to notify developers when an error occurs.

These tutorials are geared to be concise and to provide step-by-step instructions with plenty of screen shots to walk you through the process visually. This inaugural tutorial provides an overview of the ASP.NET deployment process and advice on finding a web hosting provider. Let's get started!

An Overview of the ASP.NET Deployment Process

In a nutshell, deploying an ASP.NET application involves the following three steps:

1. Configure the web application, web server, and database in the production environment.
2. Synchronize the ASP.NET pages, code files, the assemblies in the `Bin` folder, and HTML-related support files like CSS and JavaScript files.
3. Synchronize the database schema and/or data.

The configuration information for a web application is typically located in the `Web.config` file, and includes database connection strings, error handling criteria, URL rewriting rules, and email server information. Oftentimes this information is different for an application in development versus the same application in production. For instance, when developing an application it's best to use a development database so that you're not testing against the production database. As a result, the database connection strings typically differ between development and production applications. Due to these differences, part of deployment involves making changes to the web application's configuration information.

In addition to web application configuration changes, Step 1 also may entail configuration for the web server and database. For example, if an ASP.NET page creates or deletes files from a directory on the web server then the web server needs to be configured to permit these file system modifications. Similarly, there may be permission or authentication settings that need to be made to the database.

Step 2 involves synchronizing the set of essential ASP.NET pages and support files between the development and production environments. The particular set of ASP.NET-related files that need to be synchronized between the two environments depends on the type of project you created in Visual Studio, and is the discussion in the next tutorial, *Determining What Files Need to Be Deployed*. The third and fourth tutorials - *Deploying Your Site Using FTP*and *Deploying Your Site Using Visual Studio* - examine different tools and techniques for syncing these files.

When building data-driven applications there are typically two databases being used: one for development and one on production. During development, the development database's schema may be modified to include new tables, columns, stored procedures, and triggers, or may be modified to remove or rename existing database objects. Between the time that these changes are made and the time the application is deployed to production, the development and production databases are out of sync. This asynchrony needs to be fixed during the deployment process. These challenges will be examined in future tutorials.

Finding a Web Host Provider

ASP.NET applications can be deployed to any web server that has the .NET Framework and Internet Information Services (IIS) installed. You could host a site from your personal computer, assuming you had a broadband connection to the Internet and the know how to configure your router to allow incoming web requests. You could also host a site from a computer in an intranet, as many companies do. The focus of these tutorials, however, is hosting your website with a web host provider.

> **NOTE**
>
> IIS is Microsoft's enterprise-grade web server. It ships with the non-Home editions of Windows, such as Windows Server 2008 and certain editions of Windows Vista. You do not need to install IIS to serve ASP.NET applications in a development environment, as Visual Studio includes the ASP.NET Development Web Server. However, the ASP.NET Development Web Server only accepts local connections and therefore cannot be used in a production environment.

Before you can deploy your site to a web host provider you must first decide what company to do business with. There are countless web hosting companies in the marketplace; a search for "web hosting company" returns more than five million results. How do you find the one that's right for you? Your favorite search engine is a good starting place, as are websites like TopHosts and HostCritique, which compare and contrast various hosting

services. I also advise asking your colleagues and coworkers for any recommendations; you can also ask for recommendations at the Hosting Open Forum here at the ASP.NET Forums.

Web hosting companies typically offer shared hosting plans and dedicated hosting plans. With shared hosting a single web server hosts dozens if not hundreds of different websites. With dedicated hosting you lease a computer from the company that serves your site and your site alone. A shared hosting plan might include support for ASP.NET pages, the ability to work with Microsoft Access databases, 5 GB of disk space, and 100 GB of monthly bandwidth traffic for $9.95 per month. Another shared hosting plan might include support for ASP.NET pages, access to the Microsoft SQL Server 2008 database server, 10 GB of disk space and 250 GB of monthly bandwidth traffic for $19.95 per month. Dedicated hosting plans are usually much more expensive, costing several hundred dollars per month, but offer better performance and more control than shared hosting options. What plan you choose depends on your budget, how much traffic your website receives, and the features you anticipate you'll need.

Two important considerations when choosing a web host provider are customer service and quality of service. If you have a question or a configuration problem, how long does it take from submitting your problem to the web host's helpdesk until you get a response? How reliable are the company's services? Do they frequently have database outages? How often does their email server go offline? You can always ask a company to provide details about their uptime and inquire about their customer service policy, but a more surefire way is to solicit the feedback of current and past customers, which you can do through online forums, newsgroups, and email listservs.

> **NOTE**
>
> Some web hosting companies focus their business on a particular technology stack, such as .NET or LAMP (**L** inux, **A** pache, **M** ySQL, and **P** HP), so make sure that the company you select hosts ASP.NET applications. Also check to ensure that they support the version of ASP.NET you are using to build your application. And if you are building a data-driven application, make sure that the web host offers the same database server and version that you are using.

Summary

ASP.NET web applications are typically designed, created, and tested in a local development environment. Once a version is ready for release, it is moved to a production environment. While it is possible to host ASP.NET websites on your personal computer or on servers within your company, many businesses and individuals choose to outsource their hosting to a web host provider.

This tutorial series examines the steps for deploying an ASP.NET application to a web host provider, exploring common challenges. This tutorial offered a high-level overview of the ASP.NET deployment process and gave tips for finding a suitable web host provider. The next tutorial looks at what ASP.NET-related files need to be copied to the production environment when deploying your website.

Happy Programming!

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Determining What Files Need to Be Deployed (VB)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> What files need to be deployed from the development environment to the production environment depends in part on whether the ASP.NET application was built using the Web Site Model or Web Application Model. Learn more about these two project models and how the project model affects deployment.

Introduction

Deploying an ASP.NET web application entails copying the ASP.NET-related files from the development environment to the production environment. The ASP.NET-related files include ASP.NET web page markup and code and client- and server-side support files. Client-side support files are those files referenced by your web pages and sent directly to the browser - images, CSS files and JavaScript files, for example. Server-side support files include those that are used to process a request on the server-side. This includes configuration files, web services, class files, Typed DataSets, and LINQ to SQL files, among others.

In general, all client-side support files should be copied from the development environment to the production environment, but what server-side support files get copied depends on whether you are explicitly compiling the server-side code into an assembly (a `.dll` file) or if you are having these assemblies auto-generated. This tutorial highlights what files need to be deployed when explicitly compiling the code into an assembly versus having this compilation step occur automatically.

Explicit Compilation Versus Automatic Compilation

ASP.NET web pages are divided into declarative markup and source code. The declarative markup portion includes HTML, Web controls, and databinding syntax; the code portion contains event handlers written in Visual Basic or C# code. The markup and code portions are typically separated into different files: `WebPage.aspx` contains the declarative markup while `WebPage.aspx.vb` houses the code.

Consider an ASP.NET page named `Clock.aspx` that contains a Label control whose Text property is set to the current date and time when the page loads. The declarative markup portion (in `Clock.aspx`) would contain the markup for a Label Web control - `<asp:Label runat="server" id="TimeLabel" />` - while the code portion (in `Clock.aspx.vb`) would have a `Page_Load` event handler with the following code:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As  System.EventArgs) Handles Me.Load
    TimeLabel.Text =  "The time at the beep is: " & DateTime.Now.ToString()
End Sub
```

In order for the ASP.NET engine to service a request for this page, the page's code portion (the *`WebPage`*`.aspx.vb` file) must first be compiled. This compilation can happen explicitly or automatically.

If the compilation happens explicitly then the entire application's source code is compiled into one or more assemblies (`.dll` files) located in the application's `Bin` directory. If the compilation happens automatically then the resulting auto-generated assembly is, by default, placed in the `Temporary ASP.NET Files` folder, which can be found at `%WINDOWS%\Microsoft.NET\Framework\<version>`, although this location is configurable via the `<compilation>` element in `Web.config`. With explicit compilation you must take some action to compile the

ASP.NET application's code into an assembly, and this step occurs prior to deployment. With automatic compilation the compilation process occurs on the web server when the resource is first accessed.

Regardless of what compilation model you use, the markup portion of all ASP.NET pages (the `WebPage.aspx` files) need to be copied to the production environment. With explicit compilation you need to copy up the assemblies in the `Bin` folder, but you do not need to copy up the ASP.NET pages' code portions (the `WebPage.aspx.vb` files). With automatic compilation you need to copy up the code portion files so that the code is present and can be compiled automatically when the page is visited. The markup portion of each ASP.NET web page includes a `@Page` directive with attributes that indicate whether the page's associated code was already explicitly compiled or whether it needs to be automatically compiled. As a result, the production environment can work with either compilation model seamlessly and you do not need to apply any special configuration settings to indicate that explicit or automatic compilation is used.

Table 1 summarizes the different files to deploy when using explicit compilation versus automatic compilation. Note that regardless of the compilation model used you should always deploy the assemblies in the `Bin` folder, if that folder exists. The `Bin` folder contains the assemblies specific to the web application, which include the compiled source code when using the explicit compilation model. The `Bin` directory also contains assemblies from other projects and any open-source or third-party assemblies you may be using, and these need to be on the production server. Therefore, as a general rule of thumb, copy the `Bin` folder up to production when deploying. (If you are using the automatic compilation model and are not using any external assemblies then you won't have a `Bin` directory - that's OK!)

COMPILATION MODEL	DEPLOY MARKUP PORTION FILE?	DEPLOY SOURCE CODE FILE?	DEPLOY ASSEMBLIES IN `BIN` DIRECTORY?
Explicit Compilation	Yes	No	Yes
Automatic Compilation	Yes	Yes	Yes (if it exists)

Table 1: What files you deploy depends on the compilation model used.

Taking a Trip Down Memory Lane

What compilation approach is used depends, in part, on how the ASP.NET application is managed in Visual Studio. Since .NET's inception in the year 2000 there have been four different versions of Visual Studio - Visual Studio .NET 2002, Visual Studio .NET 2003, Visual Studio 2005, and Visual Studio 2008. Visual Studio .NET 2002 and 2003 managed ASP.NET applications using the *Web Application Project* model. The key features of the Web Application Project model are:

- The files that makeup the project are defined in a single project file. Any files not defined in the project file are not considered part of the web application by Visual Studio.
- Uses explicit compilation. Building the project compiles the code files within the project into a single assembly that is placed in the `Bin` folder.

When Microsoft released Visual Studio 2005 they dropped support for the Web Application Project model and replaced it with the Web Site Project model. The Web Site Project model differentiated itself from the *Web Application Project* model in the following ways:

- Rather than having a single project file that spells out the project's files, the file system is used instead. In short, any files within the web application folder (or subfolders) are considered part of the project.
- Building a project in Visual Studio does not create an assembly in the `Bin` directory. Instead, building a Web Site Project reports any compile-time errors.
- Support for automatic compilation. Web Site Projects are typically deployed by copying the markup and source code to the production environment, although the code can be precompiled (explicit compilation).

Microsoft revived the Web Application Project model when it released Visual Studio 2005 Service Pack 1. However, Visual Web Developer continued to only support the Web Site Project model. The good news is that this limitation was dropped with Visual Web Developer 2008 Service Pack 1. Today you can create ASP.NET applications in Visual Studio (and Visual Web Developer) using either the Web Application Project model or the Web Site Project model. Both models have their pros and cons. Refer to Introduction to Web Application Projects: Comparing Web Site Projects and Web Application Projects for a comparison of the two models and to help decide what project model works best for your situation.

Exploring the Sample Web Application

The download for this tutorial includes an ASP.NET application called Book Reviews. The website mimics a hobby website someone might create to share their book reviews with the online community. This ASP.NET web application is very simple and consists of the following resources:

- `Web.config`, the application's configuration file.
- A master page (`Site.master`).
- Seven different ASP.NET pages:
  - ~/`Default.aspx` - the site's homepage.
  - ~/`About.aspx` - an "About the Site" page.
  - ~/`Fiction/Default.aspx` - a page listing the fiction books that have been reviewed.
    - ~/`Fiction/Blaze.aspx` - a review of the Richard Bachman novel *Blaze*.
  - ~/`Tech/Default.aspx` - a page listing the technology books that have been reviewed.
    - ~/`Tech/CYOW.aspx` - a review of *Create Your Own Website*.
    - ~/`Tech/TYASP35.aspx` - a review of *Teach Yourself ASP.NET 3.5 in 24 Hours*.
- Three different CSS files in the `Styles` folder.
- Four image files - a Powered by ASP.NET logo and images of the covers of the three reviewed books - all located in the `Images` folder.
- A `Web.sitemap` file, which defines the site map and is used to display menus in the `Default.aspx` pages in the root directory and `Fiction` and `Tech` folders.
- A class file named `BasePage.vb` that defines a base `Page` class. This class extends the functionality of the `Page` class by automatically setting the `Title` property based on the page's position in the site map. In a nutshell, any ASP.NET code-behind class that extends `BasePage` (instead of `System.Web.UI.Page`) will have its title set to a value depending on its position in the site map. For instance, when viewing the ~/`Tech/CYOW.aspx` page, the title is set to "Home : Technology : Create Your Own Website".

Figure 1 shows a screen shot of the Book Reviews website when viewed through a browser. Here you see the page ~/Tech/TYASP35.aspx, which reviews the book *Teach Yourself ASP.NET 3.5 in 24 Hours*. The breadcrumb that spans the top of the page and the menu in the left column are based on the site map structure defined in `Web.sitemap`. The image in the right upper corner is one of the book cover images located in the `Images` folder. The website's look and feel are defined via cascading style sheet rules spelled out by the CSS files in the `Styles` folder, while the overarching page layout is defined in the master page, `Site.master`.

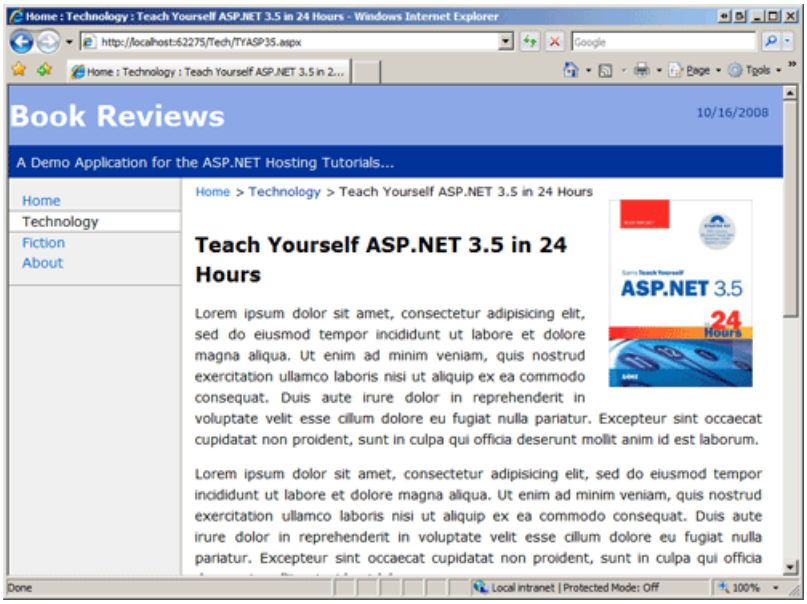


Figure 1: The Book Reviews website offers reviews on an assortment of titles (Click to view full-size image)

This application does not use a database; each review is implemented as a separate web page in the application. This tutorial (and the next several tutorials) walk through deploying a web application that does not have a database. However, in a future tutorial we will enhance this application to store reviews, reader comments, and other information within a database, and will explore what steps need to be performed to correctly deploy a data-driven web application.

> **NOTE**
>
> These tutorials focus on hosting ASP.NET applications with a web host provider and do not explore ancillary topics like ASP.NET's site map system or using a base Page class. For more information on these technologies, and for more background on other topics covered throughout the tutorial, refer to the Further Reading section at the end of each tutorial.

This tutorial's download has two copies of the web application, each implemented as a different Visual Studio project type: BookReviewsWAP, a Web Application Project, and BookReviewsWSP, a Web Site Project. Both projects were created with Visual Web Developer 2008 SP1 and use ASP.NET 3.5 SP1. To work with these projects start by unzipping the contents to your Desktop. To open the Web Application Project (BookReviewsWAP), navigate to the `BookReviewsWAP` folder and double-click the Solution file, `BookReviewsWAP.sln`. To open the Web Site Project (BookReviewsWSP), launch Visual Studio and then, from the File menu, choose the Open Web Site option, browse to the `BookReviewsWSP` folder on your Desktop, and click OK.

The remaining two sections in this tutorial look at what files you will need to copy to the production environment when deploying the application. The next two tutorials - *Deploying Your Site Using FTP* and *Deploying Your Site Using Visual Studio* - show different ways to copy these files to a web host provider.

Determining the Files to Deploy for the Web Application Project

The Web Application Project model uses explicit compilation - the project's source code is compiled into a single assembly each time you build the application. This compilation includes the ASP.NET pages' code-behind files (~/`Default.aspx.vb`, ~/`About.aspx.vb`, and so on), as well as the `BasePage.vb` class. The resulting assembly is named `BookReviewsWAP.dll` and is located in the application's `Bin` directory.

Figure 2 shows the files that make up the Book Reviews Web Application Project.

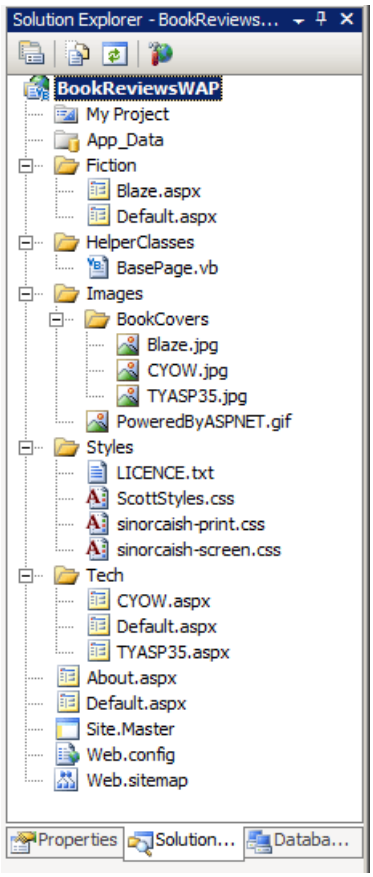


Figure 2: The Solution Explorer lists the files that comprise the Web Application Project

> **NOTE**
>
> As Figure 2 shows, the ASP.NET pages' code-behind files are not displayed in the Solution Explorer for a Visual Basic Web Application Project. To view the code-behind class for a page, right-click on the page in Solution Explorer and choose View Code.

To deploy an ASP.NET application developed using the Web Application Project model start by building the application so as to explicitly compile the most recent source code into an assembly. Next, copy the following files to the production environment:

- The files that contain the declarative markup for every ASP.NET page, such as ~/`Default.aspx`, ~/`About.aspx`, and so on. Also, copy up the declarative markup for any master pages and User Controls.
- The assemblies (`.dll` files) in the `Bin` folder. You do not need to copy the program database files (`.pdb`) or any XML files you may find in the `Bin` directory.

You do not need to copy the ASP.NET pages' source code files to the production environment, nor do you need to copy the `BasePage.vb` class file.

> **NOTE**
>
> As Figure 2 shows, the `BasePage` class is implemented as a class file in the project, placed in folder named `HelperClasses`. When the project is compiled the code in the `BasePage.vb` file is compiled along with the ASP.NET pages' code-behind classes into the single assembly, `BookReviewsWAP.dll`. ASP.NET has a special folder named `App_Code` that is designed to hold class files for Web Site Projects. The code in the `App_Code` folder is automatically compiled and therefore should not be used with Web Application Projects. Instead, you should place your application's class files in a normal folder named `HelperClasses`, or `Classes`, or something similar. Alternatively, you can place class files in a separate Class Library project.

In addition to copying the ASP.NET-related markup files and the assembly in the `Bin` folder, you also need to copy the client-side support files - the images and CSS files - as well as the other server-side support files, `Web.config` and `Web.sitemap`. These client- and server-side support files need to be copied to the production environment regardless of whether you use explicit or automatic compilation.

Determining the Files to Deploy for the Web Site Project Files

The Web Site Project model supports automatic compilation, a feature not available when using the Web Application Project model. With explicit compilation you must compile your project's source code into an assembly and copy that assembly to the production environment. On the other hand, with automatic compilation you simply copy the source code to the production environment and it is compiled by the runtime on demand as needed.

The Build menu option in Visual Studio is present in both Web Application Projects and Web Site Projects. Building a Web Application Projects compiles the project's source code into a single assembly located in the `Bin` directory; building a Web Site Project checks for any compile-time errors, but does not create any assemblies. To deploy an ASP.NET application developed using the Web Site Project model all you need to do is copy the appropriate files to the production environment, but I would encourage you to first build the project to ensure that there are no compile-time errors.

Figure 3 shows the files that make up the Book Reviews Web Site Project.

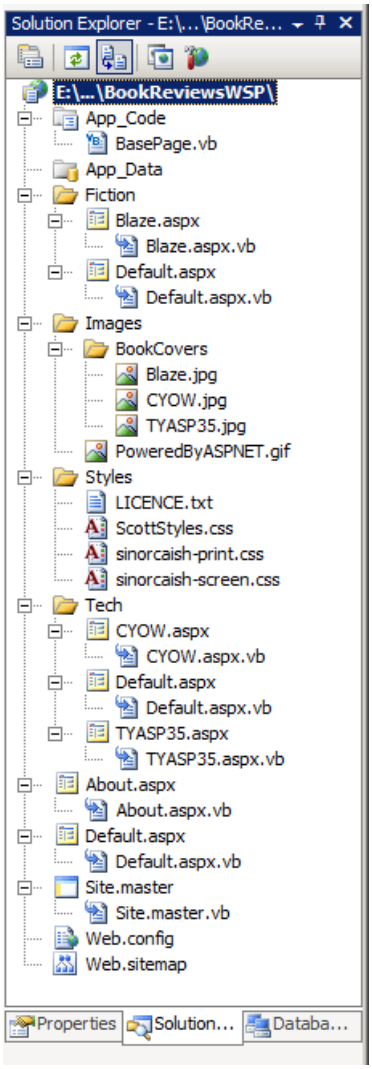


Figure 3: The Solution Explorer lists the files that comprise the Web Site Project

Deploying a Web Site Project involves copying all of the ASP.NET-related files to the production environment - that includes the markup pages for ASP.NET pages, master pages, and User Controls, along with their code files.

You also need to copy up any class files, such as `BasePage.vb`. Note that the `BasePage.vb` file is located in the `App_Code` folder, which is a special ASP.NET folder used in Web Site Projects for class files. The special folder needs to be created on production, as well, as the class files in the `App_Code` folder on the development environment must be copied to the `App_Code` folder on production.

In addition to copying the ASP.NET markup and source code files, you also need to copy the client-side support files - the images and CSS files - as well as the other server-side support files, `Web.config` and `Web.sitemap`.

> **NOTE**
> Web Site Projects can also use explicit compilation. A future tutorial will examine how to explicitly compile a Web Site Project.

Summary

Deploying an ASP.NET application entails copying the necessary files from the development environment to the production environment. The precise set of files that need to be synced depends on whether the ASP.NET application's code is explicitly or automatically compiled. The compilation strategy employed is influenced by whether Visual Studio is configured to manage the ASP.NET application using the Web Application Project model or the Web Site Project model.

The Web Application Project model uses explicit compilation and compiles the project's code into a single assembly in the `Bin` folder. When deploying the application, the markup portion of the ASP.NET pages and the contents of the `Bin` folder must be pushed up to the production environment; the source code in the application - the code files and code-behind classes, for example - do not need to be copied to the production environment.

The Web Site Project model uses automatic compilation by default, although it is possible to explicitly compile a Web Site Project, as we will see in future tutorials. Deploying an ASP.NET application that uses automatic compilation requires that the markup portion *and* source code must be copied to the production environment. The code is automatically compiled on the production environment when it is requested for the first time.

Now that we have examined what files need to be synced between the development and production environments we are ready to deploy the Book Reviews application to a web host provider.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Compilation Overview
- ASP.NET User Controls
- Examining ASP.NET's Site Navigation
- Introduction to Web Application Projects
- Master Page Tutorials
- Sharing Code Between Pages
- Using a Custom Base Class For Your ASP.NET Pages' Code-Behind Classes
- Visual Studio 2005's Web Site Project System: What Is It and Why Did We Do It?
- Walkthrough: Converting a Web Site Project to a Web Application Project in Visual Studio

Deploying Your Site Using an FTP Client (VB)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> The simplest way to deploy an ASP.NET application is to manually copy the necessary files from the development environment to the production environment. This tutorial shows how to use an FTP client to get the files from your desktop to the web host provider.

Introduction

The previous tutorial introduced a simple Book Review ASP.NET web application, which is comprised of a handful of ASP.NET pages, a master page, a custom base `Page` class, a number of images, and three CSS style sheets. We are now ready to deploy this application to a web host provider, at which point the application will be accessible to anyone with a connection to the Internet!

From our discussions in the *Determining What Files Need to Be Deployed* tutorial, we know what files need to be copied to the web host provider. (Recall that what files are copied depends on whether your application is explicitly or automatically compiled.) But how do we get the files from the development environment (our desktop) up to the production environment (the web server managed by the web host provider)? The **F**ile **T**ransfer **P**rotocol (FTP) is a commonly used protocol for copying files from one machine to another over a network. Another option is FrontPage Server Extensions (FPSE). This tutorial focuses on using stand-alone FTP client software to deploy the necessary files from the development environment to the production environment.

> **NOTE**
>
> Visual Studio includes tools for publishing websites via FTP; these tools, as well as a look at tools that use FPSE, are covered in the next tutorial.

To copy the files using FTP we need an *FTP client* on the development environment. An FTP client is an application that is designed to copy files from the computer it's installed to a computer that is running an *FTP server*. (If your web host provider supports file transfers via FTP, as most do, then there is an FTP server running on their web servers.) There are a number of FTP client applications available. Your web browser can even double as an FTP client. My favorite FTP client, and the one I will be using for this tutorial, is FileZilla, a free, open-source FTP client that is available for Windows, Linux, and Macs. Any FTP client will work, though, so feel free to use whatever client you are most comfortable with.

If you are following along you will need to create an account with a web host provider before you can complete this tutorial or subsequent ones. As noted in the previous tutorial, there are a gaggle of web host provider companies with a wide spectrum of prices, features, and quality of service. For this tutorial series I will be using Discount ASP.NET as my web host provider, but you can follow along with any web host provider as long as they support the ASP.NET version your site is developed in. (These tutorials were created using ASP.NET 3.5.) Also, because we will be copying the files to the web host provider using FTP in this tutorial and in future ones it is imperative that your web host provider supports FTP access to their web servers. Virtually all web host providers offer this feature, but you should double-check before signing up.

Deploying the Book Review Web Application Project

Recall that there are two versions of the Book Review web application: one implemented using the Web Application Project model (BookReviewsWAP) and the other using the Web Site Project model (BookReviewsWSP). The project type influences whether the site is compiled automatically or explicitly, and that compilation model dictates what files need to be deployed. Consequently, we will examine deploying the BookReviewsWAP and BookReviewsWSP projects separately, starting with the BookReviewsWAP. Take a moment to download these two ASP.NET applications if you have not done so already.

Launch the BookReviewsWAP project by navigating to the `BookReviewsWAP` folder and double-clicking the `BookReviewsWAP.sln` file. Before deploying the project it is important to build it to ensure that any changes to the source code are included in the compiled assembly. To build the project go to the Build menu and choose the Build BookReviewsWAP menu option. This compiles the source code in the project into a single assembly, `BookReviewsWAP.dll`, which is placed in the `Bin` folder.

We are now ready to deploy the necessary files! Launch your FTP client and connect to the web server at your web host provider. (When you sign up with a web hosting company they will email you information on how to connect to the FTP server; this includes the address for the FTP server as well as a username and password.)

Copy the following files from your desktop to the root website folder at your web host provider. When you FTP into the web server at the web host provider you are likely at the root website directory. However, some web host providers have a subfolder named `www` or `wwwroot` that serves as the root folder for your website files. Finally, when FTPing the files you may need to create the corresponding folder structure on the production environment - the `Bin` folder, the `Fiction` folder, the `Images` folder, and so on.

- `~/Default.aspx`
- `~/About.aspx`
- `~/Site.master`
- `~/Web.config`
- `~/Web.sitemap`
- The complete contents of the `Styles` folder
- The complete contents of the `Images` folder (and its subfolder, `BookCovers`)
- `~/Fiction/Default.aspx`
- `~/Fiction/Blaze.aspx`
- `~/Tech/Default.aspx`
- `~/Tech/CYOW.aspx`
- `~/Tech/TYASP35.aspx`
- `~/Bin/BookReviewsWAP.dll`

Figure 1 shows FileZilla after the necessary files have been copied. FileZilla displays the files on the local computer on the left and the files on the remote computer on the right. As Figure 1 shows, the ASP.NET source code files, such as `About.aspx.vb`, are on the local computer (the development environment) but were not copied to the web host provider (the production environment) because code files do not need to be deployed when using explicit compilation.

> **NOTE**
>
> There is no harm in having the source code files on the production server, as they are ignored. ASP.NET forbids HTTP requests to source code files by default so that even if the source code files are present on the production server they are inaccessible to visitors to your website. (That is, if a user tries to visit `http://www.yoursite.com/Default.aspx.vb` they will get an error page that explains that these types of files - `.vb` files - are forbidden.)

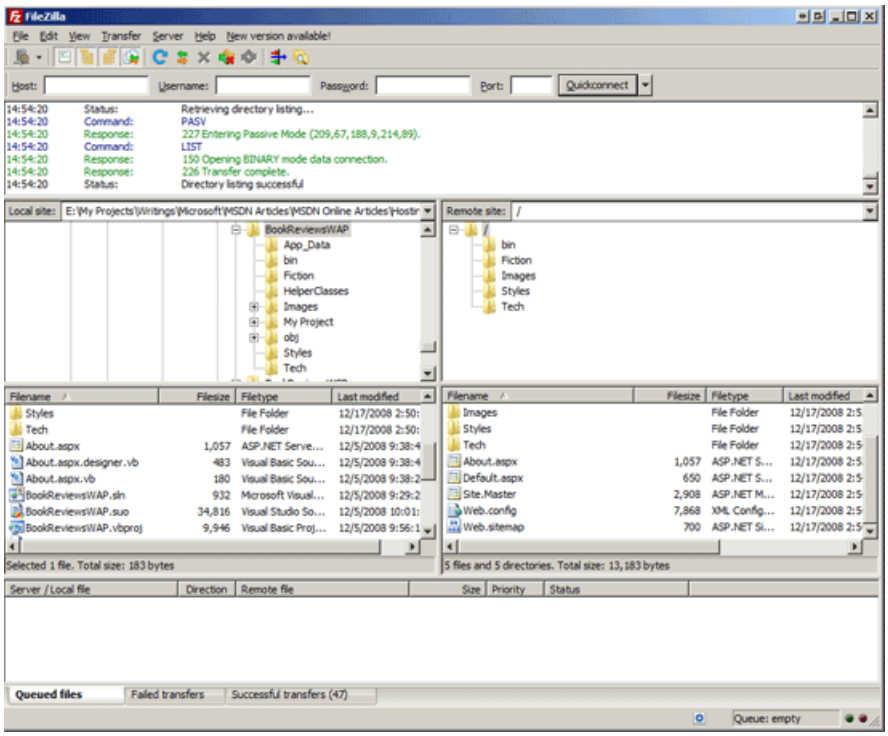


Figure 1: Use an FTP Client to Copy the Necessary Files from Your Desktop to the Web Server at the Web Host Provider (Click to view full-size image)

After deploying your site take a moment to test the site. If you have purchased a domain name and configured the DNS settings properly, you can visit your site by entering your domain name. Alternatively, your web host provider should have supplied you with a URL to your site, which will look something like *accountname.webhostprovider*.com or *webhostprovider*.com/*accountname*. For example, the URL for my account on Discount ASP.NET is: `http://httpruntime.web703.discountasp.net`.

Figure 2 shows the deployed Book Reviews site. Note that I am viewing it on Discount ASP.NET's servers, at `http://httpruntime.web703.discountasp.net`. At this point in time anyone with a connection to the Internet could view my website! As we'd expect, the site looks and behaves just as it did when testing it in the development environment.

> **NOTE**
>
> If you get an error when viewing your application take a moment to ensure that you deployed the correct set of files. Next, check the error message to see if it reveals any clues as to the problem. Following that, you can turn to your web host company's helpdesk or post your question to the appropriate forum at the ASP.NET Forums.

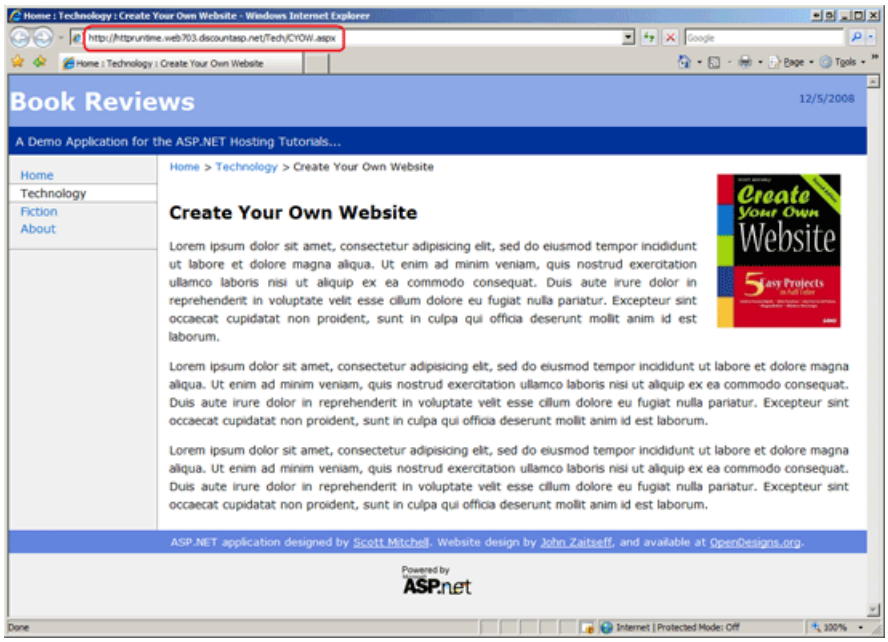


Figure 2: The Book Reviews Site is Now Accessible to Anyone with an Internet Connection (Click to view full-size image)

Deploying the Book Review Web Site Project

When deploying an ASP.NET application that uses automatic compilation, such as the BookReviewsWSP Web Site Project, there is no compiled assembly in the `Bin` folder. As a result, the web application's source code files must be deployed to the production environment. Let's walk through this process.

As with the Web Application Project it's wise to first Build the application before deploying it. While building a Web Site Project does not create an assembly, it does check for any compile-time errors in the page. Better to find these errors now rather than having a visitor to your site discover them for you!

Once you've successfully built the project, use your FTP client to copy the following files to the root website folder at your web host provider. You may need to create the corresponding folder structure on the production environment.

> **NOTE**
>
> If you already deployed the BookReviewsWAP project but still want to try deploying the BookReviewsWSP project, first delete all of the files on the web server that were uploaded when deploying BookReviewsWAP and then deploy the files for BookReviewsWSP.

- `~/Default.aspx`
- `~/Default.aspx.vb`
- `~/About.aspx`
- `~/About.aspx.vb`
- `~/Site.master`
- `~/Site.master.vb`
- `~/Web.config`
- `~/Web.sitemap`
- The complete contents of the `Styles` folder
- The complete contents of the `Images` folder (and its subfolder, `BookCovers`)
- `~/App_Code/BasePage.vb`

- `~/Fiction/Default.aspx`
- `~/Fiction/Default.aspx.vb`
- `~/Fiction/Blaze.aspx`
- `~/Fiction/Blaze.aspx.vb`
- `~/Tech/Default.aspx`
- `~/Tech/Default.aspx.vb`
- `~/Tech/CYOW.aspx`
- `~/Tech/CYOW.aspx.vb`
- `~/Tech/TYASP35.aspx`
- `~/Tech/TYASP35.aspx.vb`

Figure 3 shows FileZilla after copying up the necessary files. As you can see, the ASP.NET source code files, such as `About.aspx.vb`, are present on both the local computer (the development environment) and the web host provider (the production environment) because code files need to be deployed when using automatic compilation.

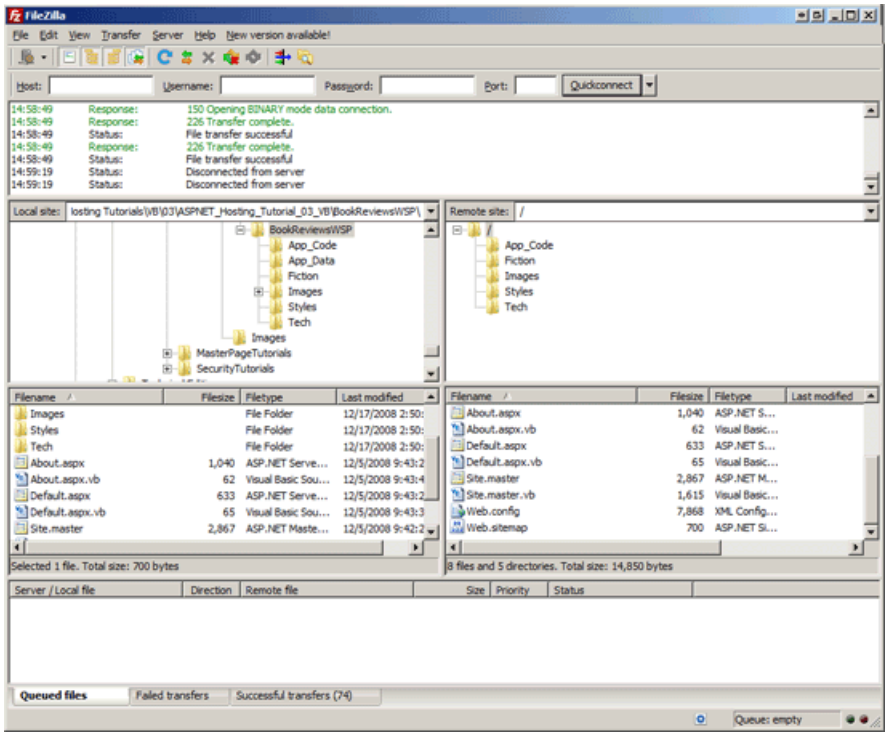


Figure 3: Use an FTP Client to Copy the Necessary Files from Your Desktop to the Web Server at the Web Host Provider (Click to view full-size image)

The user experience is not affected by the application's compilation model. The same ASP.NET pages are accessible and they look and behave the same whether the website was created using the Web Application Project model or the Web Site Project model.

Updating a Web Application on Production

Web application development and deployment are not a one-time process. For example, when creating the Book Review website I built the various pages and wrote the accompanying code on my personal computer (the development environment). After reaching a certain stable state, I deployed my application so that others could visit the site and read my reviews. But deployment does not mark the end of my development on this site. I may add more book reviews or implement new features, such as allowing my visitors to rate books or leave their own comments. Such enhancements would be developed on the development environment and, when completed, would need to be deployed. Development and deployment, therefore, are cyclical. You develop an application and then deploy it. While the site is live and in production, new features are added and bugs are fixed over time,

which necessitates re-deploying the application. And so on and so on.

As you might expect, when re-deploying a web application you only need to copy new and changed files. There's no need to re-deploy unchanged pages or server- or client-side support files (although there's no harm in doing so).

> **NOTE**
> One thing to keep in mind when using explicit compilation is that anytime you add a new ASP.NET page to the project or make any code-related changes, you need to rebuild your project, which updates the assembly in the `Bin` folder. Consequently, you'll need to copy this updated assembly to production when updating a web application on production (along with the other new and updated content).

Also understand that any changes to the `Web.config` or the files in the `Bin` directory stops and restarts the website's Application Pool. If your session state is stored using the `InProc` mode (the default) then your site's visitors will lose their session state whenever these key files are modified. To avoid this pitfall, consider storing session using the `StateServer` or `SQLServer` modes. For more information on this topic read Session-State Modes.

Finally, keep in mind that re-deploying an application can take anywhere from a few seconds to several minutes, depending on the number and size of files that need to be copied to the production environment. During this time users visiting your site may experience errors or odd behavior. You can "turn off" your entire application by adding a page named `App_Offline.htm` to your application's root directory that explains to your users that the site is down for maintenance (or whatever) and will be back up shortly. When the `App_Offline.htm` file is present, the ASP.NET runtime redirects all incoming requests to that page.

Summary

Deploying a web application entails copying the necessary files from the development environment to the production environment. The most common means by which files are transferred over a network is the File Transfer Protocol (FTP), and most web host providers support FTP access to their web servers. In this tutorial we saw how to use an FTP client to deploy the needed files to the web server. Once deployed, the website can be visited by anyone with a connection to the Internet!

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- App_Offline.htm and Working Around the "IE Friendly Errors" Feature
- Session-State Modes

PREVIOUS NEXT

Deploying Your Site Using Visual Studio (VB)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Visual Studio includes tools for deploying a website. Learn more about these tools in this tutorial.

Introduction

The preceding tutorial looked at how to deploy a simple ASP.NET web application to a web host provider. Specifically, the tutorial showed how to use an FTP client like FileZilla to transfer the necessary files from the development environment to the production environment. Visual Studio also offers built-in tools to facilitate deployment to a web host provider. This tutorial examines two of these tools: the Copy Web Site tool, where you can move files to and from a remote web server using FTP or FrontPage Server Extensions; and the Publish tool, which copies the entire website to a specified location.

> **NOTE**
>
> Other deployment-related tools offered by Visual Studio include Web Setup Projects and Web Deployment Projects Add-In. Web Setup Projects package a website's contents and configuration information into a single MSI file. This option is most useful for websites that are deployed within an intranet or for companies that sell a pre-packaged web application that customers install on their own web servers. The Web Deployment Projects Add-In is a Visual Studio Add-In that facilitates specifying configuration differences between builds for development environments and production environments. Web Setup Projects are not discussed in this tutorial series; Web Deployment Projects are summarized in the *Common Configuration Differences Between Development and Production* tutorial.

Deploying Your Site Using the Copy Web Site Tool

Visual Studio's Copy Web Site tool is similar in functionality to a stand-alone FTP client. In a nutshell, the Copy Web Site tool allows you to connect to a remote web site through FTP or FrontPage Server Extensions. Similar to FileZilla's user interface, the Copy Web Site user interface consists of two panes: the left pane lists the local files while the right pane lists those files on the destination server.

> **NOTE**
>
> The Copy Web Site tool is only available for Web Site Projects. Visual Studio does offer this tool when you are working with a Web Application Project.

Let's take a look at using the Copy Web Site tool to publish the Book Review application to production. Because the Copy Web Site tool only works with projects that use the Web Site Project model we can only examine using this tool with the BookReviewsWSP project. Open that project.

Launch the Copy Web Site tool project by clicking the Copy Web Site icon in the Solution Explorer (this icon is circled in Figure 1); alternatively, you can select the Copy Web Site option from the Website menu. Either approach launches the Copy Web Site user interface shown in Figure 1; only the left pane in Figure 1 is populated because we have yet to connect to a remote server.

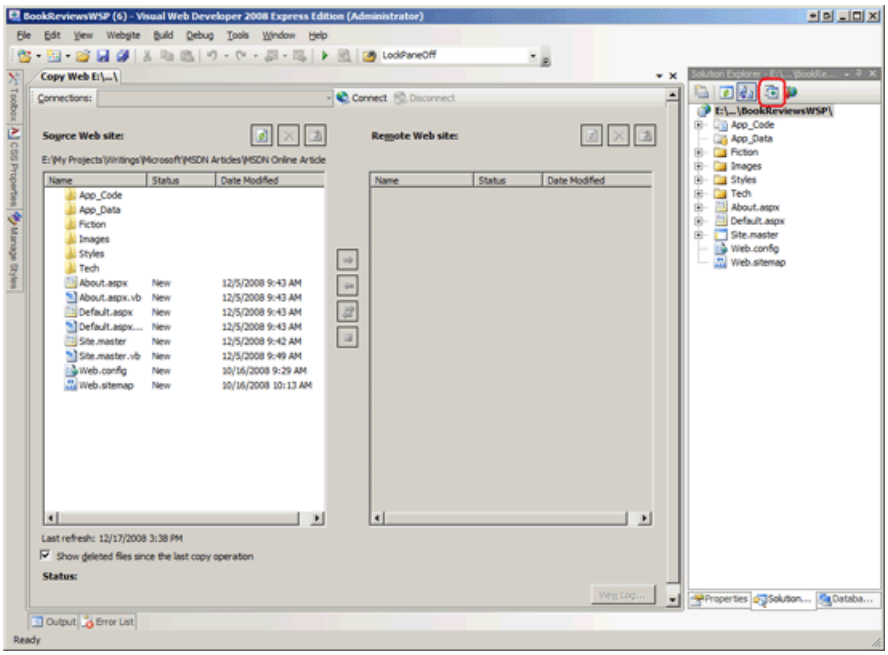


Figure 1: The Copy Web Site Tool's User Interface is Divided Into Two Panes (Click to view full-size image)

In order to deploy our site we need to first connect to the web host provider. Click the Connect button at the top of the Copy Web Site user interface. This displays the Open Web Site dialog box shown in Figure 2.

You can connect to the destination website by selecting one of the four options from the left:

- **File System** - select this to deploy your site to a folder or network share accessible from your computer.
- **Local IIS** - use this option to deploy the site to the IIS web server installed on your computer.
- **FTP Site** - connect to a remote web site using FTP.
- **Remote Site** - connect to a remote website using FrontPage Server Extensions.

Most web host providers support FTP, but fewer offer FrontPage Server Extension support. For that reason, I've selected the FTP Site option and then entered the connection information as shown in Figure 2.

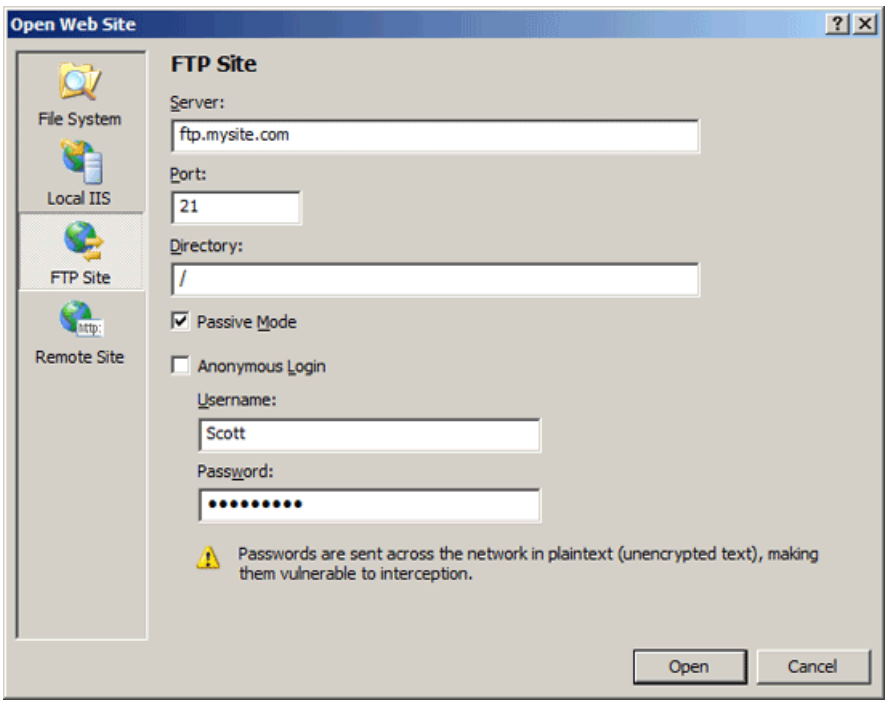


Figure 2: Specify the Destination Website (Click to view full-size image)

After you connect, the Copy Web Site tool loads the files at the remote site in the right pane and indicates the status of each file: New, Deleted, Changed, or Unchanged. You can copy a file from the local site to the remote

site, or vice-a-versa.

Let's add a new page to the BookReviewsWSP project and then deploy it so that we can see the Copy Web Site tool in action. Create a new ASP.NET page in Visual Studio in the root directory named `Privacy.aspx`. Have the page use the master page `Site.master` and add your site's privacy policy to this page. Figure 3 shows Visual Studio after this page has been created.

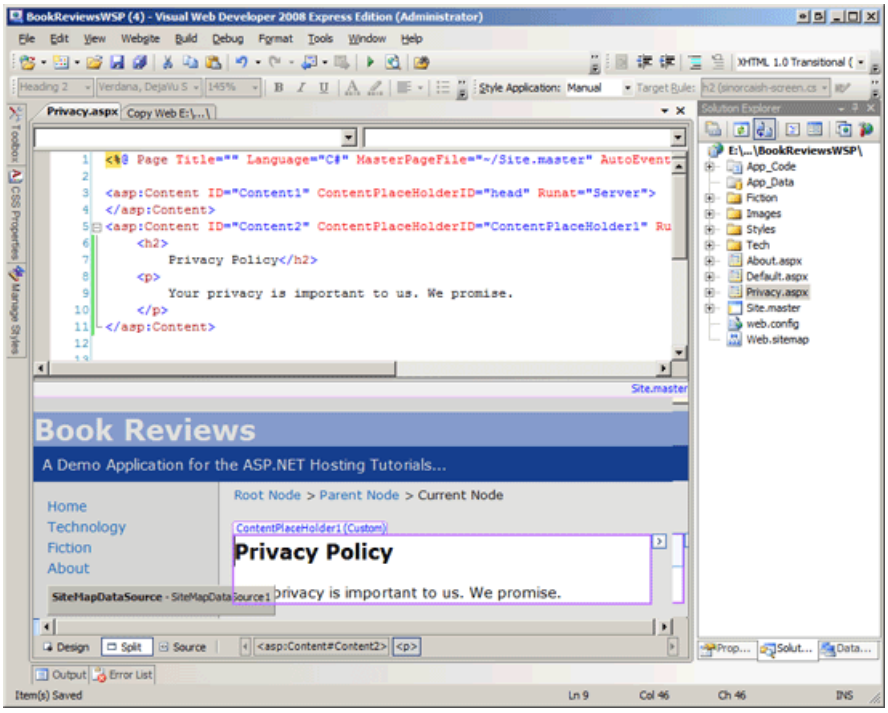


Figure 3: Add a New Page Named `Privacy.aspx` to the Website's Root Folder (Click to view full-size image)

Next, return to the Copy Web Site user interface. As Figure 4 shows, the left pane now includes the new files - `Policy.aspx` and `Policy.aspx.vb`. What's more, these files are marked with an arrow icon and a Status of New, indicating that they exist on the local site but not on the remote site.

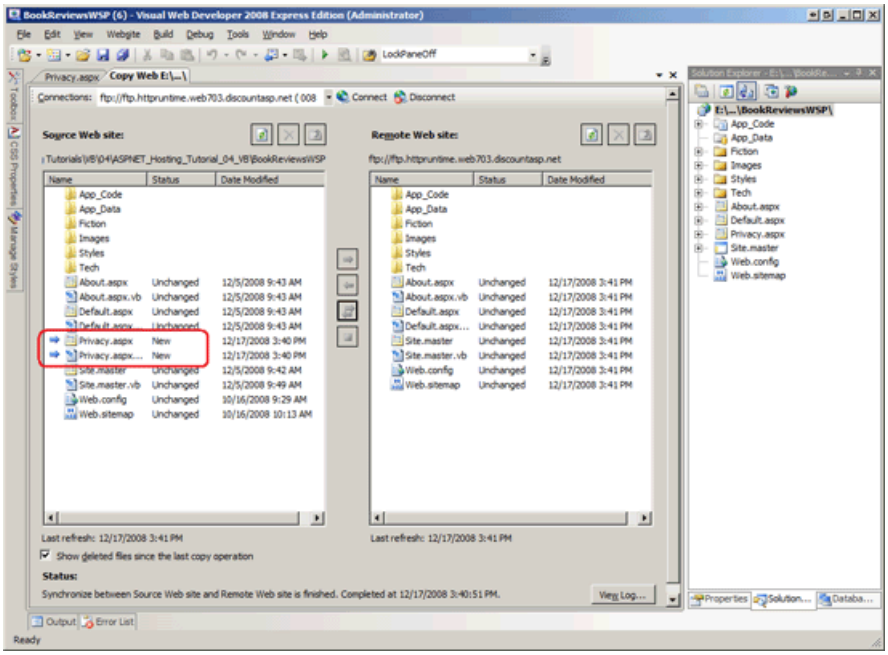


Figure 4: The Copy Web Site Tool Includes the New `Privacy.aspx` Page in its Left Pane (Click to view full-size image)

To deploy the new files select them and then click the arrow icon to transfer them to the remote site. After the transfer completes the `Policy.aspx` and `Policy.aspx.vb` files exist on both the local and remote sites with the status Unchanged.

Along with listing new files, the Copy Web Site tool highlights any files that differ between the local and remote sites. To see this in action, return to the `Privacy.aspx` page and add a few more words to the privacy policy. Save the page and then return to the Copy Web Site tool. As Figure 5 shows, the `Privacy.aspx` page in the left pane has a status of Changed indicating that it is out of sync with the remote site.

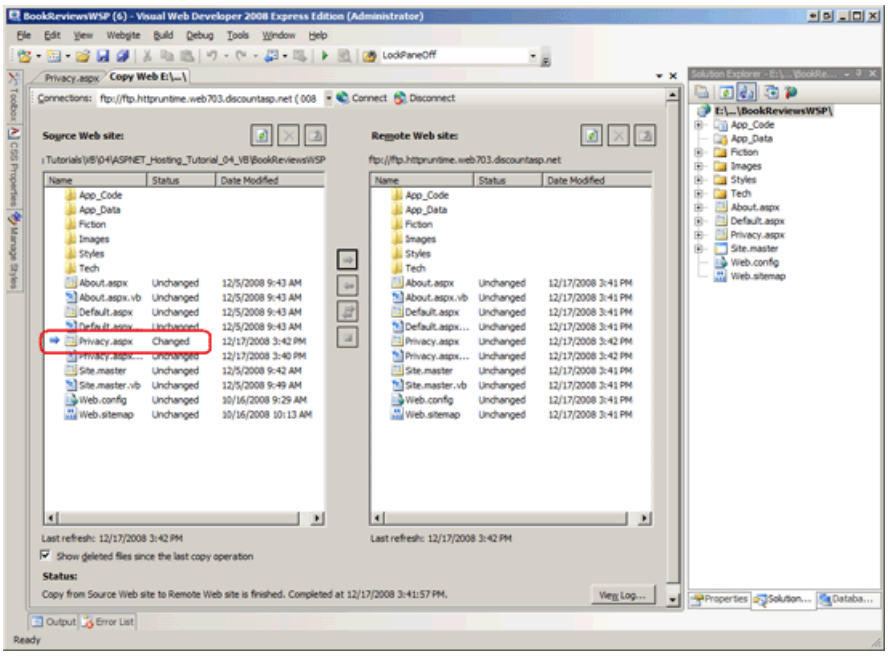


Figure 5: The Copy Web Site Tool Indicates that the `Privacy.aspx` Page has been Changed (Click to view full-size image)

The Copy Web Site tool also indicates if a file has been deleted since the last copy operation. Delete the `Privacy.aspx` from the local project and refresh the Copy Web Site tool. The `Privacy.aspx` and `Privacy.aspx.vb` files remain listed in the left pane, but have a Deleted status indicating that they have been removed since the last copy operation.

Publishing a Web Application

Another way to deploy your web application from within Visual Studio is to use the Publish option, which is accessible via the Build menu. The Publish option explicitly compiles the application and then copies all of the necessary files up to the specified remote site. As we'll see shortly, the Publish option is more blunt than the Copy Web Site tool. Whereas the Copy Web Site tool lets you examine the files on the local and remote sites and permits you to upload or download individual files as needed, the Publish option deploys the entire web application.

In addition to copying all of the needed files to the specified remote site, the Publish option also explicitly compiles the application. Given that Web Application Projects need to be explicitly compiled it should come as no surprise that the Publish option is available for Web Application Projects. What may be a bit surprising is that the Publish option is also available for Web Site Projects. As noted in the *Determining What Files Need to Be Deployed* tutorial, Web Site Projects can be explicitly compiled through a process referred to as *pre-compilation*. This tutorial focuses on using the Publish option with Web Application Projects; a future tutorial will explore pre-compilation, at which point we'll return to look at using the Publish option with Web Site Projects.

> **NOTE**
>
> While the Publish option is available in Visual Studio for both Web Site Projects and Web Application Projects, Visual Web Developer only offers the Publish option for Web Application Projects.

Let's look at deploying the Book Reviews application using the Publish option. Start by opening BookReviewsWAP

(the Web Application Project) in Visual Studio. From the Publish menu choose the Build BookReviewsWAP project. This brings up a dialog box that prompts for the target location, among other configuration options (see Figure 6). Much like with the Copy Web Site tool you can enter a location that points to a local folder, a local website on IIS, a remote website that supports FrontPage Server Extensions, or an FTP server address. You can choose whether to replace the files on the remote web server with the deployed files or to delete all of the content on the remote site before publishing. You can also specify whether to copy:

- Only the files in the project needed to run the application, which omits the unneeded source code and project-related files.
- All project files, which includes the source code files and Visual Studio project files like the Solution file.
- All files in the source project folder, which copies all files in the source project folder regardless of whether they're included in the project.

There's also an option to upload the contents of the `App_Data` folder.

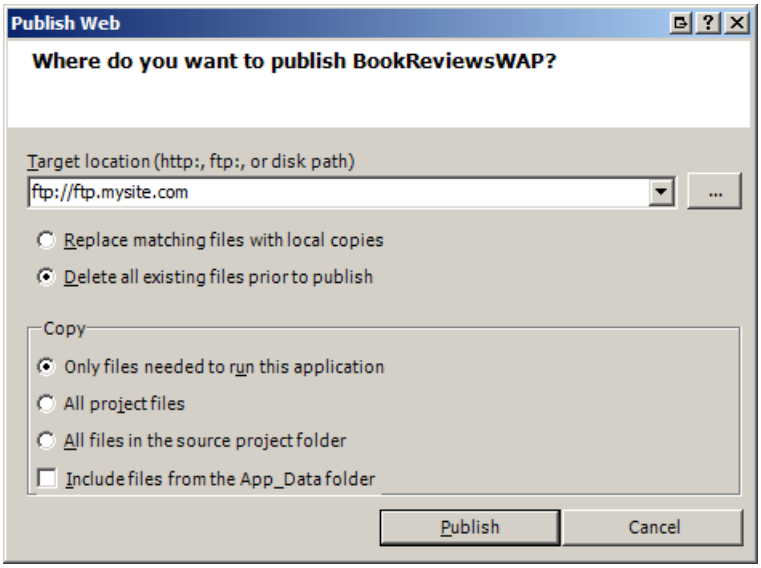


Figure 6: Specify the Destination Website (Click to view full-size image)

For the Book Review application the remote site contains the files deployed when copying the BookReviewsWSP project via the Copy Web Site tool. Therefore, let's have the Publish option start by deleting all existing content. Also, let's just copy the necessary files rather than cluttering the production environment with unneeded source code and project files. After specifying these options, click the Publish button. Over the next several seconds Visual Studio will deploy the necessary files to the destination site, displaying its progress in the Output window.

Figure 7 shows the files on the FTP site after the Publish operation has completed. Note that only the markup pages and the necessary sever- and client-side support files have been uploaded.

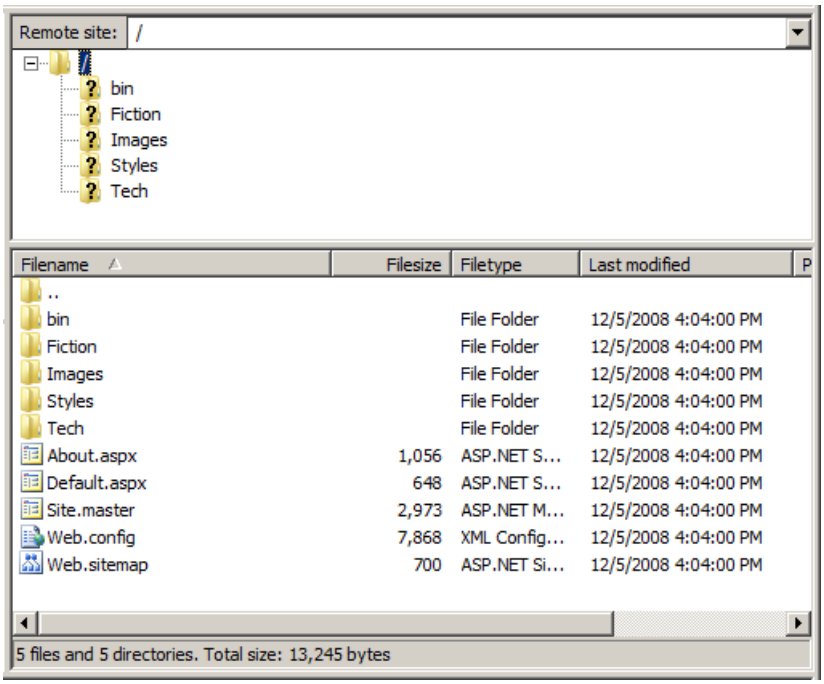


Figure 7: Only the Needed Files Were Published to the Production Environment (Click to view full-size image)

The Publish option is a less nuanced tool than the Copy Web Site tool. Whereas the Copy Web Site tool allows you to inspect the files on the local and remote sites and see how they differ, the Publish option provides no such interface. Moreover, the Copy Web Site tool enables you to make one-off changes, uploading or deleting individual files. The Publish option does not allow such fine-grained control; instead, it publishes the *entire* application. This behavior has its pros and cons. On the plus side, you know when using the Publish option you won't be forgetting to upload an important file. But consider what happens if you have made a small change to a very large website - with the Publish option you cannot update that page or two that has been modified, but instead you must wait while Visual Studio deploys the entire site.

It's not uncommon for there to be certain files whose content differs between the production and development environments. A key example is the application's configuration file, `Web.config`. Because the Publish option blindly copies the web application files it overwrites the production environment's customized configuration files with the version in the development environment. The subsequent tutorial explores this topic further and offers tips for deploying a web application when such differences exist.

Summary

Deploying a website involves copying the necessary files from the development environment to the production environment. The previous tutorial showed how to transfer files using an FTP client like FileZilla. This tutorial examined two deployment tools in Visual Studio: the Copy Web Site tool and the Publish option. The Copy Web Site tool is similar to an FTP client in that it has a two-paned interface listing the files on the local computer and a specified remote computer that makes it easy to upload or download files between the two computers. The Publish option is a more blunt tool that explicitly compiles the project and then deploys the entire application to the specified destination.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Copying Web Site with the Copy Web Site Tool
- How Do I: Deploy a Web Site Using the Copy Web Site Tool (Video)
- How To: Publish Web Application Projects
- How To: Publish Web Sites

- Setup and Deployment Projects in Visual Studio

PREVIOUS NEXT

Common Configuration Differences Between Development and Production (VB)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download PDF

> In earlier tutorials we deployed our website by copying all of the pertinent files from the development environment to the production environment. However, it is not uncommon for there to be configuration differences between environments, which necessitates that each environment have a unique Web.config file. This tutorial examines typical configuration differences and looks at strategies for maintaining separate configuration information.

Introduction

The last two tutorials walked through deploying a simple web application. The *Deploying Your Site Using an FTP Client* tutorial showed how to use a stand-alone FTP client to copy the necessary files from the development environment up to production. The preceding tutorial, *Deploying Your Site Using Visual Studio*, looked at deployment using Visual Studio's Copy Web Site tool and Publish option. In both tutorials every file in the production environment was a copy of a file on the development environment. However, it is not uncommon for configuration files in the production environment to differ from those in the development environment. A web application's configuration is stored in the `Web.config` file and typically includes information about external resources, such as database, web, and email servers. It also spells out the behavior of the application in certain situations, such as the course of action to take when an unhandled exception occurs.

When deploying a web application it is important that the correct configuration information end up in the production environment. In most cases the `Web.config` file in the development environment cannot be copied to the production environment as-is. Instead, a customized version of `Web.config` needs to be uploaded to production. This tutorial briefly reviews some of the more common configuration differences; it also summarizes some techniques for maintaining different configuration information between the environments.

Typical Configuration Differences Between the Development and Production Environments

The `Web.config` file includes an assortment of configuration information for an ASP.NET application. Some of this configuration information is the same regardless of the environment. For instance, the authentication settings and URL authorization rules spelled out in the `Web.config` file's `<authentication>` and `<authorization>` elements are usually the same regardless of the environment. But other configuration information - such as information about external resources - typically differs depending on the environment.

Database connections strings are a prime example of configuration information that differs based on the environment. When a web application communicates with a database server it must first establish a connection, and that is achieved through a connection string. While it is possible to hard-code the database connection string directly in the web pages or the code that connects to the database, it is best to place it `Web.config`'s `<connectionStrings>` element so that the connection string information is in a single, centralized location. Oftentimes a different database is used during development than is used in production; consequently, the connection string information must be unique for each environment.

> **NOTE**
> Future tutorials explore deploying data-driven applications, at which point we'll dive into the specifics of how database connection strings are stored in the configuration file.

The intended behavior of the development and production environments differs substantially. A web application in the development environment is being created, tested, and debugged by a small group of developers. In the production environment that same application is being visited by many different simultaneous users. ASP.NET includes a number of features that help developers in testing and debugging an application, but these features should be disabled for performance and security reasons when in the production environment. Let's look at a few such configuration settings.

Configuration Settings That Impact Performance

When an ASP.NET page is visited for the first time (or the first time after it has changed), its declarative markup must be converted into a class and this class must be compiled. If the web application uses automatic compilation then the page's code-behind class needs to be compiled, too. You can configure an assortment of compilation options via the `Web.config` file's `<compilation>` element.

The debug attribute is one of the most important attributes in the `<compilation>` element. If the `debug` attribute is set to "true" then the compiled assemblies include debug symbols, which are needed when debugging an application in Visual Studio. But debug symbols increase the size of the assembly and impose additional memory requirements when running the code. Furthermore, when the `debug` attribute is set to "true" any content returned by `WebResource.axd` is not cached, meaning that each time a user visits a page they will need to re-download the static content returned by `WebResource.axd`.

> **NOTE**
> `WebResource.axd` is a built-in HTTP Handler introduced in ASP.NET 2.0 that server controls use to retrieve embedded resources, such as script files, images, CSS files, and other content. For more information on how `WebResource.axd` works and how you can use it to access embedded resources from your custom server controls, see Accessing Embedded Resources Through a URL Using `WebResource.axd`.

The `<compilation>` element's `debug` attribute is usually set to "true" in the development environment. In fact, this attribute must be set to "true" in order to debug a web application; if you try to debug an ASP.NET application from Visual Studio and the `debug` attribute is set to "false", Visual Studio will display a message explaining that the application cannot be debugged until the `debug` attribute is set to "true" and will offer to make this change for you.

You should **never** have the `debug` attribute set to "true" in a production environment because of its impact on performance. For a more thorough discussion on this topic, refer to Scott Guthrie's blog post, Don't Run Production ASP.NET Applications With `debug="true"` Enabled.

Custom Errors and Tracing

When an unhandled exception occurs in an ASP.NET application it bubbles up to the runtime at which point one of three things happens:

- A generic runtime error message is displayed. This page informs the user that there was a runtime error, but does not provide any details about the error.
- An exception details message is displayed, which includes information on the exception that was just thrown.
- A custom error page is displayed, which is an ASP.NET page that you create that displays any message you desire.

What happens in the face of an unhandled exception depends on the `Web.config` file's `<customErrors>` section.

When developing and testing an application it helps to see details of any exception in the browser. However, showing exception details in an application on production is a potential security risk. Moreover, it's unflattering and makes your website look unprofessional. Ideally, in the event of an unhandled exception a web application in the development environment will show the exception's details while the same application in production will show a custom error page.

> **NOTE**
>
> The default `<customErrors>` section setting shows the exception details message only when the page is being visited through localhost, and shows the generic runtime error page otherwise. This isn't ideal, but it's assuring to know that the default behavior doesn't reveal exception details to non-local visitors. A future tutorial examines the `<customErrors>` section in more detail and shows how to have a custom error page shown when an error occurs in production.

Another ASP.NET feature that is useful during development is tracing. Tracing, if enabled, records information about each incoming request and provides a special web page, `Trace.axd`, for viewing recent request details. You can turn on and configure tracing via the `<trace>` element in `Web.config`.

If you enable tracing make sure that it is disabled in production. Because the trace information includes cookies, session data, and other potentially sensitive information, it is important to disable tracing in production. The good news is that, by default, tracing is disabled and the `Trace.axd` file is only accessible through localhost. If you change these default settings in development make sure that they are turned back off in production.

Techniques for Maintaining Separate Configuration Information

Having different configuration settings in the development and production environments complicates the deployment process. In the previous two tutorials the deployment process involved copying all of the necessary files from development to production, but that approach only works if the configuration information is the same in both environments. There are a variety of techniques for deploying an application with varying configuration information. Let's catalog some of these options for hosted web applications.

Manually Deploying the Production Environment Configuration File

The simplest approach is to maintain two versions of the `Web.config` file: one for the development environment and one for the production environment. Deploying a site to production entails copying all of the files to the production server in the development environment *except* for the `Web.config` file. Instead, the production environment-specific `Web.config` file would be copied to production.

This approach is not very sophisticated, but it is easy to implement because configuration information changes infrequently. It works best for applications with a small development team that are hosted on a single web server and whose configuration information is infrequently changed. It's most tenable when manually deploying the application files using a stand-alone FTP client. When using Visual Studio's Copy Web Site tool or Publish option you'll need to first swap out the deployment-specific `Web.config` file with the production-specific one before deploying, and then swap them back after deployment completes.

Change the Configuration During the Build or Deployment Process

The discussions thus far have assumed an ad-hoc build and deployment process. Many larger software projects have more formalized processes that make use of open-source, home-grown, or third-party tools. For such projects you can likely customize the build or deployment process to appropriately modify the configuration information before it is pushed to production. If you build your web application using MSBuild, NAnt, or some other build tool, you can likely add a build step to modify the `Web.config` file to include the production-specific settings. Or your deployment workflow could programmatically connect to the source control server and retrieve the appropriate `Web.config` file.

The actual approach for getting the appropriate configuration information to production varies widely based on your tools and workflow. As such, we will not delve into this topic further. If you are using a popular build tool like

MSBuild or NAnt you can find deployment articles and tutorials specific to these tools through a web search.

Managing Configuration Differences via the Web Deployment Project Add-In

In 2006 Microsoft released the Web Development Project Add-In for Visual Studio 2005. An Add-In for Visual Studio 2008 was released in 2008. This Add-In allows for ASP.NET developers to create a separate Web Deployment Project alongside their web application project that, when built, explicitly compiles the web application and copies the files to deploy to a local output directory. The Web Application Project uses MSBuild behind the scenes.

By default, the development environment's `Web.config` file is copied to the output directory, but you can setup the Web Deployment Project to customize the

configuration information that gets copied to this directory in the following ways:

- Via `Web.config` file section replacement, in which you specify the section to replace and an XML file that contains the replacement text.
- By providing a path to an external configuration source file. With this option selected, the Web Deployment Project copies a particular `Web.config` file to the output directory (rather than the `Web.config` file used in the development environment).
- By adding custom rules to the MSBuild file used by the Web Deployment Project.

To deploy the web application build the Web Deployment Project and then copy the files from the project's output folder to the production environment.

To learn more about using the Web Deployment Project check out this Web Deployment Projects article from the April 2007 issue of MSDN Magazine, or consult the links in the Further Reading section at the end of this tutorial.

> **NOTE**
> You cannot use the Web Deployment Project with Visual Web Developer because the Web Deployment Project is implemented as a Visual Studio Add-In and the Visual Studio Express Editions (including Visual Web Developer) do not support Add-Ins.

Summary

The external resources and behavior of a web application in development are typically different than when the same application is in production. For instance, database connection strings, compilation options, and the behavior when an unhandled exception occurs commonly differ between environments. The deployment process must accommodate these differences. As we discussed in this tutorial, the simplest approach is to manually copy an alternate configuration file to the production environment. More elegant solutions are possible when using the Web Deployment Project Add-In or with a more formalized build or deployment process that can accommodate such customizations.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Connection Strings Explained
- Database Connection Strings @ ConnectionStrings.com
- Don't Run Production ASP.NET Applications with `debug="true"` Enabled
- Gracefully Responding to Unhandled Exceptions - Displaying User-Friendly Error Pages
- How Do I: Use a Visual Studio 2008 Web Deployment Project?
- Key Configuration Settings When Deploying a Database

- Visual Studio 2008 Web Deployment Projects Download | Visual Studio 2005 Web Deployment Projects Download
- VS 2008 Web Deployment Projects | VS 2008 Web Deployment Project Support Released
- Web Deployment Projects

PREVIOUS NEXT

Core Differences Between IIS and the ASP.NET Development Server (VB)

2/20/2020 • 12 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

When testing an ASP.NET application locally, chances are you are using the ASP.NET Development Web Server. However, the production website is most likely powered IIS. There are some differences between how these web servers handle requests, and these differences can have important consequences. This tutorial explores some of the more germane differences.

Introduction

Whenever a user visits an ASP.NET application his browser sends a request to the website. That request is picked up by the web server software, which coordinates with the ASP.NET runtime to generate and return the content for the requested resource. The Internet Information Services (IIS) are a suite of services that provide common Internet-based functionality for Windows servers. IIS is the most commonly used web server for ASP.NET applications in production environments; it's most likely the web server software being used by your web host provider to serve your ASP.NET application. IIS can also be used as the web server software in the development environment, although this entails installing IIS and properly configuring it.

The ASP.NET Development Server is an alternative web server option for the development environment; it ships with and is integrated into Visual Studio. Unless the web application has been configured to use IIS, the ASP.NET Development Server is automatically started and used as the web server the first time you visit a web page from within Visual Studio. The demo web applications we created back in the *Determining What Files Need to Be Deployed* tutorial were both file system-based web applications that were not configured to use IIS. Therefore, when visiting either of these websites from within Visual Studio the ASP.NET Development Server is used.

In a perfect world the development and production environments would be identical. However, as we discussed in the preceding tutorial it is not uncommon for the environments to have differing configuration settings. Using different web server software in the environments adds another variable that must be taken into consideration when deploying an application. This tutorial covers the key differences between IIS and the ASP.NET Development Server. Because of these differences there are scenarios where code that runs flawlessly in the development environment throws an exception or behaves differently when executed in production.

Security Context Differences

Whenever the web server software handles an incoming request it associates that request with a particular security context. This security context information is used by the operating system to determine what actions are permissible by the request. For example, an ASP.NET page might include code that logs some message to a file on disk. In order for this ASP.NET page to execute without error, the security context must have the appropriate file system-level permissions, namely write permissions on that file.

The ASP.NET Development Server associates incoming requests with the security context of the currently logged on user. If you are logged on to your desktop as an administrator, then the ASP.NET pages served by the ASP.NET Development Server will have the same access rights as an administrator. However, ASP.NET requests handled by IIS are associated with a specific machine account. By default, the Network Service machine account is used by IIS versions 6 and 7, although your web host provider may have configured a unique account for each customer.

What's more, your web host provider has likely given limited permissions to this machine account. The net result is that you may have web pages that execute without error in the development environment, but generate authorization-related exceptions when hosted in the production environment.

To show this type of error in action I've created a page in the Book Reviews website that creates a file on disk that stores the most recent date and time someone viewed the *Teach Yourself ASP.NET 3.5 in 24 Hours* review. To follow along, open the `~/Tech/TYASP35.aspx` page and add the following code to the `Page_Load` event handler:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As  System.EventArgs) Handles Me.Load

    Dim filePath As  String = Server.MapPath("~/LastTYASP35Access.txt")

    Dim contents As  String = String.Format("Last accessed on {0} by {1}", _

                                DateTime.Now.ToString(), Request.UserHostAddress)

     System.IO.File.WriteAllText(filePath, contents)

End Sub
```

NOTE

The `File.WriteAllText` method creates a new file if it does not exist and then writes the specified contents to it. If the file already exists, it's existing content is overwritten.

Next, visit the *Teach Yourself ASP.NET 3.5 in 24 Hours* book review page in the development environment using the ASP.NET Development Server. Assuming that you are logged on to your computer with an account that has adequate permissions to create and modify a text file in the web application's root directory the book review appears the same as before, but each time the page is visited the date and time and user's IP address is stored in the `LastTYASP35Access.txt` file. Point your browser to this file; you should see a message similar to the one shown in Figure 1.

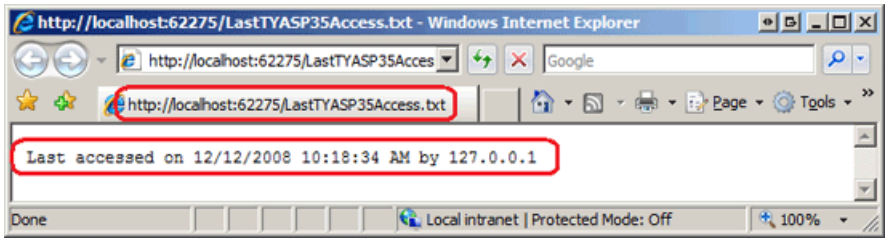


Figure 1: The Text File Contains the Last Date and Time the Book Review was Visited(Click to view full-size image)

Deploy the web application to production and then visit the hosted *Teach Yourself ASP.NET 3.5 in 24 Hours* book review page. At this point you should either see the book review page as normal or the error message shown in Figure 2. Some web host providers grant write permissions to the anonymous ASP.NET machine account, in which case the page will work without error. If, however, your web host provider prohibits write access for the anonymous account then an `UnauthorizedAccessException` exception is raised when the `TYASP35.aspx` page attempts to write the current date and time to the `LastTYASP35Access.txt` file.

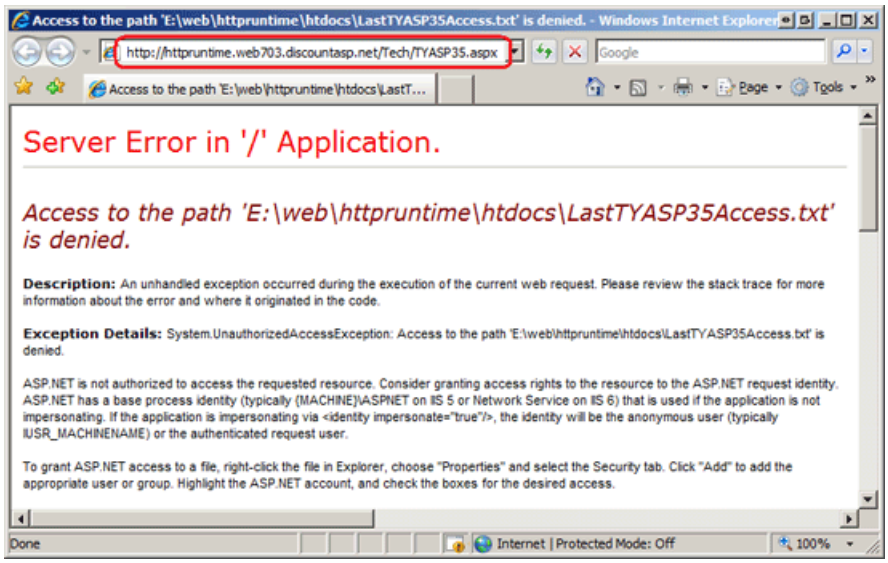


Figure 2: The Default Machine Account Used by IIS Does Not Have Permissions to Write to the File System (Click to view full-size image)

The good news is that most web host providers have some sort of permissions tool that allows you to specify file system permissions in your website. Grant the anonymous ASP.NET account write access to the root directory and then revisit the book review page. (If needed, contact your web host provider for assistance on how to grant write permissions to the default ASP.NET account.) This time the page should load without error and the `LastTYASP35Access.txt` file should be created successfully.

The take away here is that because the ASP.NET Development Server operates under a different security context than IIS, it is possible that your ASP.NET pages that read or write to the file system, read from or write to the Windows Event Log, or read or write to the Windows registry will work as expected on development but generate exceptions when on production. When building a web application that will be deployed to a shared web hosting environment, do not read or write to the Event Log or the Windows registry. Also take note of any ASP.NET pages that read from or write to the file system as you may need to grant read and write privileges on the appropriate folders in the production environment.

Differences On Serving Static Content

Another core difference between IIS and the ASP.NET Development Server is how they handle requests for static content. Every request that comes into the ASP.NET Development Server, whether for an ASP.NET page, an image, or a JavaScript file, is processed by the ASP.NET runtime. By default, IIS only invokes the ASP.NET runtime when a request comes in for an ASP.NET resource, such as an ASP.NET web page, a Web Service, and so forth. Requests for static content - images, CSS files, JavaScript files, PDF files, ZIP files, and the like - are retrieved by IIS without the involvement of the ASP.NET runtime. (It is possible to instruct IIS to work with the ASP.NET runtime when serving static content; consult the "Performing Forms-Based Authentication and URL Authentication on Static Files with IIS 7" section in this tutorial for more information.)

The ASP.NET runtime performs a number of steps to generate the requested content, including authentication (identifying the requestor) and authorization (determining if the requestor has permission to view the requested content). A popular form of authentication is *forms-based authentication*, in which a user is identified by entering their credentials - usually a username and password - into textboxes on a web page. Upon validating their credentials, the website stores an *authentication ticket* cookie on the user's browser, which is sent with every subsequent request to the website and is what is used to authenticate the user. Moreover, it is possible to specify *URL authorization* rules that dictate what users can or cannot access a particular folder. Many ASP.NET websites use forms-based authentication and URL authorization to support user accounts and to define portions of the site that are only accessible to authenticated users or users that belong to a certain role.

> **NOTE**
> For a thorough examination of ASP.NET's forms-based authentication, URL authorization, and other user account-related features, be sure to check out my Website Security Tutorials.

Consider a website that supports user accounts using forms-based authorization and has a folder that, using URL authorization, is configured to only allow authenticated users. Imagine that this folder contains ASP.NET pages and PDF files and that the intent is that only authenticated users can view these PDF files.

What happens if a visitor attempts to view one of these PDF files by entering the URL directly in his browser's Address bar? To find out, let's create a new folder in the Book Reviews site, add some PDF files, and configure the site to use URL authorization to prohibit anonymous users from visiting this folder. If you download the demo application you'll see that I created a folder called `PrivateDocs` and added a PDF from my Website Security Tutorials (how fitting!). The `PrivateDocs` folder also contains a `Web.config` file that specifies the URL authorization rules to deny anonymous users:

```
<?xml version="1.0"?>

<configuration>

    <system.web>

         <authorization>

            <deny  users="?" />

         </authorization>

     </system.web>

</configuration>
```

Finally, I configured the web application to use forms-based authentication by updating the Web.config file in the root directory, replacing:

```
<authentication mode="Windows" />
```

With:

```
<authentication mode="Forms" />
```

Using the ASP.NET Development Server, visit the site and enter the direct URL to one of the PDF files in your browser's Address bar. If you downloaded the website associated with this tutorial the URL should look something like: `http://localhost:portNumber/PrivateDocs/aspnet_tutorial01_Basics_vb.pdf`

Entering this URL into the Address bar causes the browser to send a request to the ASP.NET Development Server for the file. The ASP.NET Development Server hands off the request to the ASP.NET runtime for processing. Because we have not yet logged in, and because the `Web.config` in the `PrivateDocs` folder is configured to deny anonymous access, the ASP.NET runtime automatically redirects us to the login page, `Login.aspx` (see Figure 3). When redirecting the user to the log in page, ASP.NET includes a `ReturnUrl` querystring parameter that indicates the page the user was attempting to view. After successfully logging in the user can be returned to this page.

Figure 3: Unauthorized Users are Automatically Redirected to the Login Page (Click to view full-size image)

Now let's see how this behaves on production. Deploy your application and enter the direct URL to one of the PDFs in the `PrivateDocs` folder in production. This prompts your browser to send a request IIS for the file. Because a static file is requested, IIS retrieves and returns the file without invoking the ASP.NET runtime. As a result, there was no URL authorization check performed; the contents of the supposedly private PDF are accessible to anyone who knows the direct URL to the file.

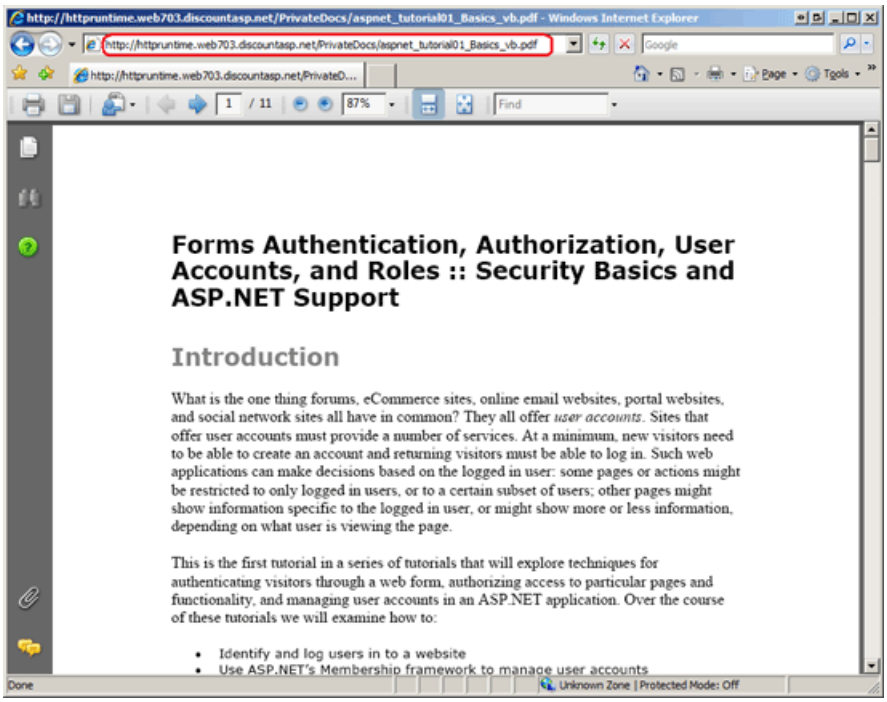


Figure 4: Anonymous Users Can Download the Private PDF Files By Entering the Direct URL to the File (Click to view full-size image)

Performing Forms-Based Authentication and URL Authentication on Static Files with IIS 7

There are a couple of techniques you can use to protect static content from unauthorized users. IIS 7 introduced the *integrated pipeline*, which marries IIS's workflow with the ASP.NET runtime's workflow. In a nutshell, you can instruct IIS to invoke the ASP.NET runtime's authentication and authorization modules all incoming requests (including static content like PDF files). Contact your web host provider to find out how to configure your website to use the integrated pipeline.

Once IIS has been configured to use the integrated pipeline add the following markup to the `Web.config` file in the root directory:

```
<system.webServer>

      <modules>

          <add  name="FormsAuthenticationModule"  type="System.Web.Security.FormsAuthenticationModule" />

          <remove  name="UrlAuthorization" />

          <add  name="UrlAuthorization"  type="System.Web.Security.UrlAuthorizationModule" />

          <remove  name="DefaultAuthentication" />

          <add  name="DefaultAuthentication"  type="System.Web.Security.DefaultAuthenticationModule" />

      </modules>

</system.webServer>
```

This markup instructs IIS 7 to use the ASP.NET-based authentication and authorization modules. Re-deploy your application and then re-visit the PDF file. This time when IIS handles the request it gives the ASP.NET runtime's authentication and authorization logic an opportunity to inspect the request. Because only authenticated users are authorized to view the contents in the `PrivateDocs` folder, the anonymous visitor is automatically redirected to the login page (refer back to Figure 3).

NOTE

If your web host provider is still using IIS 6 then you cannot use the integrated pipeline feature. One workaround is to put your private documents in a folder that prohibits HTTP access (such as `App_Data`) and then create a page to serve these documents. This page might be called `GetPDF.aspx`, and is passed the name of the PDF through a querystring parameter. The `GetPDF.aspx` page would first verify that the user has permission to view the file and, if so, would use the `Response.WriteFile(filePath)` method to send the contents of the requested PDF file back to the requesting client. This technique would also work for IIS 7 if you did not wish to enable the integrated pipeline.

Summary

Web applications in a production environment are hosted using Microsoft's IIS web server software. In the development environment, however, the application may be hosted using IIS or the ASP.NET Development Server. Ideally, the same web server software should be used in both environments because using different software adds another variable in the mix. However, the ease of use of the ASP.NET Development Server makes it an attractive choice in the development environment. The good news is that there are only a few fundamental differences between IIS and the ASP.NET Development Server, and if you are aware of these differences you can take steps to help ensure that the application works and functions the same way regardless of the environment.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Integration with IIS 7.0
- Using ASP.NET Forums Authentication With All Types of Content on IIS 7 (Video)
- Web Servers in Visual Web Developer

PREVIOUS NEXT

Deploying a Database (VB)

2/20/2020 • 14 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Deploying an ASP.NET web application entails getting the necessary files and resources from the development environment to the production environment. For data-driven web applications this includes the database schema and data. This tutorial is the first in a series that explores the steps needed to successfully deploy the database from the development environment to production.

Introduction

Deploying an ASP.NET web application entails getting the necessary files and resources from the development environment to the production environment. Over the course of the past six tutorials we looked at deploying a simple Book Reviews web application. This demo site was comprised of a number of server-side resources - ASP.NET pages, configuration files, a `Web.sitemap` file, and so forth - along with client-side resources, like images and CSS files. But what about data-driven web applications? What extra steps must be taken to deploy a web application that uses a database?

Over the next several tutorials we will address the steps needed to deploy a data-driven web application. This tutorial starts by examining how to get a database s schema and contents from the development environment to the production environment, while the subsequent tutorial looks at the needed configuration changes. Following that we'll explore challenges of deploying a database that uses the Application Services (Membership, Roles, Profile, and so on).

Examining the Updated Book Reviews Web Application

In order to demonstrate deploying a data-driven web application, I ve updated the Book Reviews web application from a simple, static website to a data-driven one. As before, there are two versions of the application in this tutorial s download: one that uses the Web Application Project model and one that uses the Web Site Project model.

The updated Book Reviews web application uses a SQL Server 2008 Express Edition database, which is stored in the site s `App_Data` folder (`~/App_Data/Reviews.mdf`). If you have SQL Server 2008 installed on your computer then the demo should run without error. If you have an older version of SQL Server you can either install the free SQL Server 2008 Express Edition or you can use the database scripts available in this tutorial s download to create the database yourself.

The `Reviews.mdf` database contains four tables:

- `Genres` - includes a record for each genre, such as Technology, Fiction, and Business.
- `Books` - includes a record for each review, with columns like `Title` , `GenreId` , `ReviewDate` , and `Review` , among others.
- `Authors` - includes information about each author who has contributed to a reviewed book.
- `BooksAuthors` - a many-to-many join table that specifies what authors have written what books.

Figure 1 shows an ER diagram of these four tables.

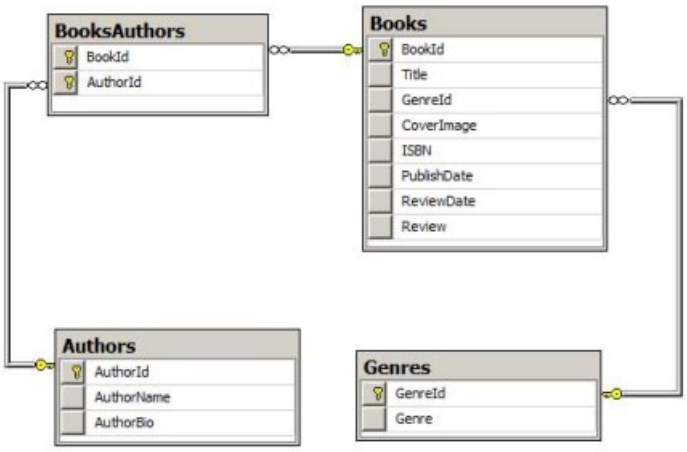


Figure 1: The Book Reviews Web Application s Database is Comprised of Four Tables (Click to view full-size image)

The previous version of the Book Reviews website had a separate ASP.NET page for each book. For example, there was a page named `~/Tech/TYASP35.aspx` that contained the review for *Teach Yourself ASP.NET 3.5 in 24 Hours*. This new data-driven version of the website has the reviews stored in the database and a single ASP.NET page, Review.aspx?ID=*bookId*, which displays the review for the specified book. Likewise, there is a Genre.aspx?ID=*genreId* page that lists the reviewed books in the specified genre.

Figures 2 and 3 show the `Genre.aspx` and `Review.aspx` pages in action. Note the URL in the Address bar for each page. In Figure 2 it s Genre.aspx?ID=85d164ba-1123-4c47-82a0-c8ec75de7e0e. Because 85d164ba-1123-4c47-82a0-c8ec75de7e0e is the `GenreId` value for the Technology genre, the page s heading reads "Technology Reviews" and the bulleted list enumerates those reviews on the site that fall under this genre.

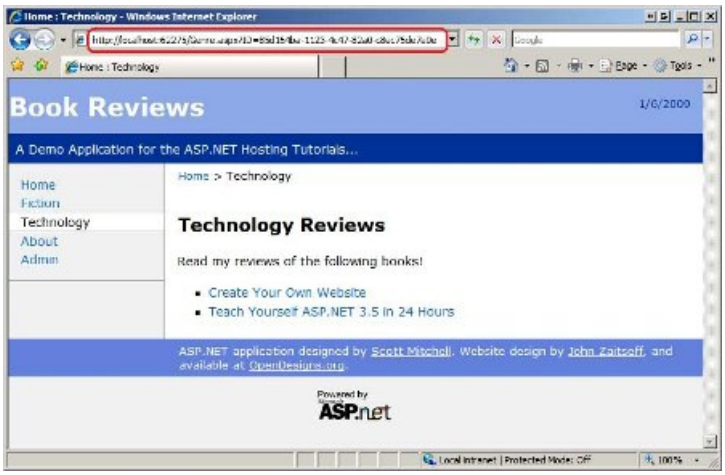


Figure 2: The Technology Genre Page (Click to view full-size image)

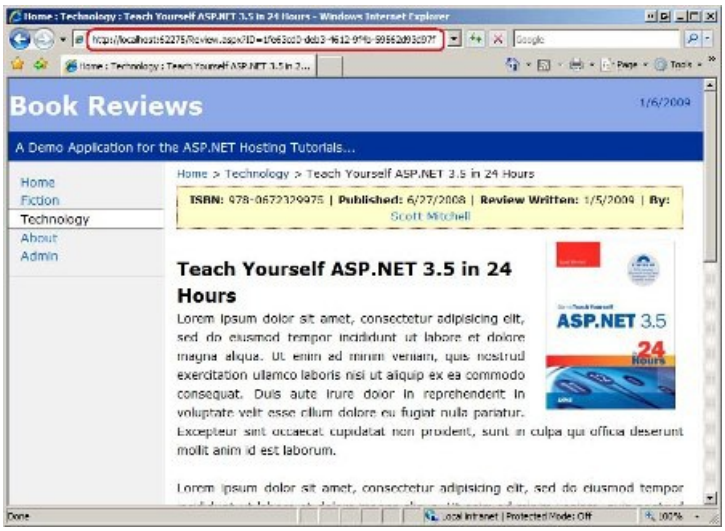


Figure 3: The Review for *Teach Yourself ASP.NET 3.5 in 24 Hours* (Click to view full-size image)

The Book Reviews web application also includes an administration section where administrators can add, edit, and delete genres, reviews, and author information. Currently, any visitor can access the administration section. In a future tutorial we'll add support for user accounts and only permit authorized users into the administration pages.

If you download the Book Reviews application please keep in mind that its purpose is to demonstrate deploying a data-driven application. It does not exhibit best practices as far as application design. For example, there is no separate Data Access Layer (DAL); the ASP.NET pages communicate directly with the database through the SqlDataSource control or ADO.NET code in their code-behind classes. For a more in-depth look at creating data-driven applications using a tiered architecture, refer to my *Working with Data* tutorials.

Databases on Development Versus Production

When you start development on a data-driven web application you must specify a database connection string, which provides the application details on how to connect to the database. This connection string specifies, among other things, the database server, the database name, and security information. Most often, the database used by the application during development is different than the database used when it s in production. There are many benefits of using different databases for development versus production. Having a different database in development means you don t have to worry about accidentally modifying or deleting live data. It also lets you put in dummy test data or make breaking changes to the data model without having to worry about the effects on the application in production. The downside of having a different database in the development and production environments is that when the application is deployed the database and any pertinent changes to the database s schema or data must also be deployed.

Prior to the first deployment, there is only one instance of the database, and that instance is in the development environment. When deploying the application to production for the first time we must not only copy up the necessary server-side and client-side files, but also copy the database from the development environment to the production environment. This is where we stand right now with the Book Reviews web application - the database resides in the `App_Data` folder in our development environment but has not yet been pushed up to the production environment.

Once the application has been deployed there are two copies of the database. As the application matures, new features may be added, necessitating a change to the data model (such as adding new columns to existing tables, making changes to existing columns, adding new tables, and so on). When the web application is next deployed, the changes applied to the database in the development environment since the last deployment must be applied to the production database. Some strategies for managing this process are discussed in a future tutorial. This tutorial focuses on deploying the entire database from the development environment to production.

Deploying the Database to the Production Environment

The remainder of this tutorial looks at how to deploy the database from the development environment to the production environment. If you are following along you need to make sure that your account with your web host provider includes Microsoft SQL Server database support. You'll also need to have some information at hand, namely the database server name, the database name, and the username and password used to connect to the database.

As noted earlier in this tutorial, the Book Reviews website s database is a SQL Server 2008 Express Edition database stored in the `App_Data` folder. It would stand to reason that deploying such a database would be as simple as copying the `App_Data` folder from the development environment to the production environment. However, most web host providers do not support hosting databases in the `App_Data` folder because of security reasons. Instead, web hosts provide an account on a SQL Server database server within their environment. Deploying the database from your development environment to the production environment requires getting your database registered on your web host s database server.

So how do you get your database from the development environment to the production environment? There are a couple ways to accomplish this depending on what services your web host offers. With some hosts, such as DiscountASP.NET, you can FTP a backup of the database or the actual `.mdf` file to your website and then, from the Control Panel, restore the backup file or attach the `.mdf` file to the SQL Server database server. With such tools deploying the database is as simple as copying the `App_Data` folder to the production environment and then attaching it via the Control Panel. This is perhaps the easiest and quickest way to publish your database for the first time.

Another approach is to use the Database Publishing Wizard. The Database Publishing Wizard is a Windows desktop application that will generate the SQL commands to create your database s schema - the tables, stored procedures, views, user-defined functions, and so forth - and, optionally, the data in its tables. You can then connect to your web host provider s database server through SQL Server Management Studio and then execute this script to duplicate the database on production. Even better, if your web host provider supports Microsoft s Database Publishing Services you can have the script generated by the Database Publishing Wizard automatically executed on the database server on your behalf. Because the Database Publishing Wizard generates a script that creates the database s schema and data, it will work regardless of whether your web host provider offers features like attaching an uploaded `.mdf` file.

Generating the SQL Commands to Create the Database Schema and Data Using the Database Publishing Wizard

Let s walk through using the Database Publishing Wizard to deploy the Book Reviews database to production. If you are using Visual Studio 2008 or beyond, the Database Publishing Wizard is already installed. If you are using Visual Studio 2005 then you will need to first download and install the wizard.

Open Visual Studio and navigate to the `Reviews.mdf` database. If you are using Visual Web Developer, go to the Database Explorer; if you are using Visual Studio, use the Server Explorer. Figure 4 shows the `Reviews.mdf` database in the Database Explorer in Visual Web Developer. As Figure 4 shows, the `Reviews.mdf` database is composed of four tables, three stored procedures, and a user-defined function.

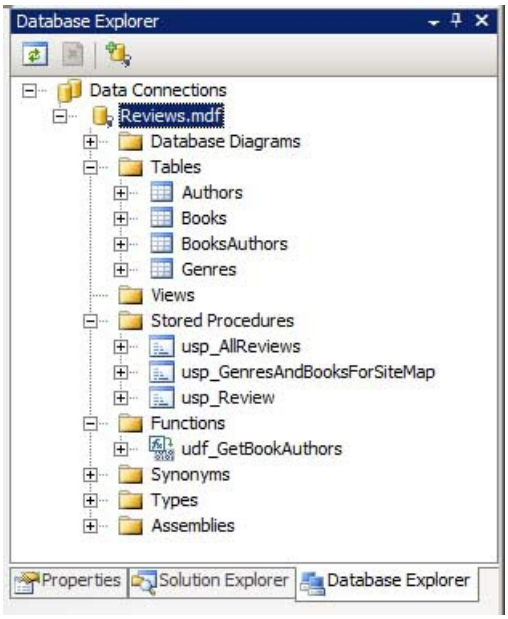


Figure 4: Locate the Database in the Database Explorer or Server Explorer (Click to view full-size image)

Right-click on the database name and choose the "Publish to provider" option from the context menu. This launches the Database Publishing Wizard (see Figure 5). Click Next to advance past the splash screen.

Figure 5: The Database Publishing Wizard Splash Screen (Click to view full-size image)

The second screen in the wizard lists the databases accessible to the Database Publishing Wizard and lets you choose whether to script all objects in the selected database or to pick which objects to script. Select the appropriate database and leave the "Script all objects in the selected database" option checked.

> **NOTE**
>
> If you get the error "There are no objects in database *databaseName* of the types scriptable by this wizard" when clicking Next in the screen shown in Figure 6, make sure that the path to your database file is not overly long. As noted in this discussion item on the Database Publishing Wizard project page, this error can arise if the path to the database file is too long.

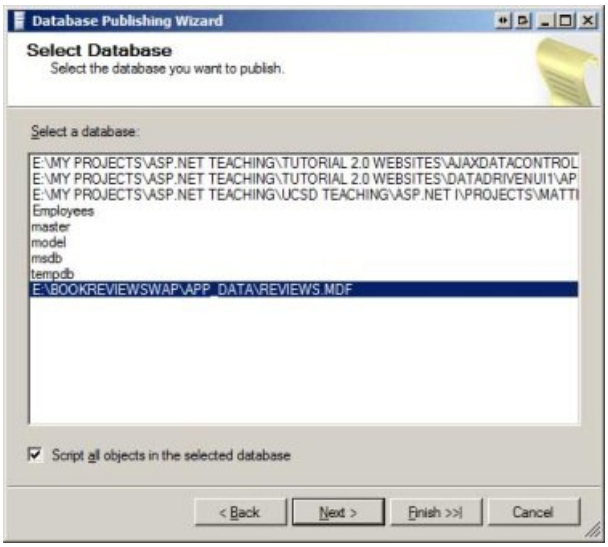


Figure 6: The Database Publishing Wizard Splash Screen (Click to view full-size image)

From the next screen you can generate a script file or, if your web host supports it, publish the database directly to your web host provider s database server. As Figure 7 shows, I am having the script written to the file `C:\REVIEWS.MDF.sql`.

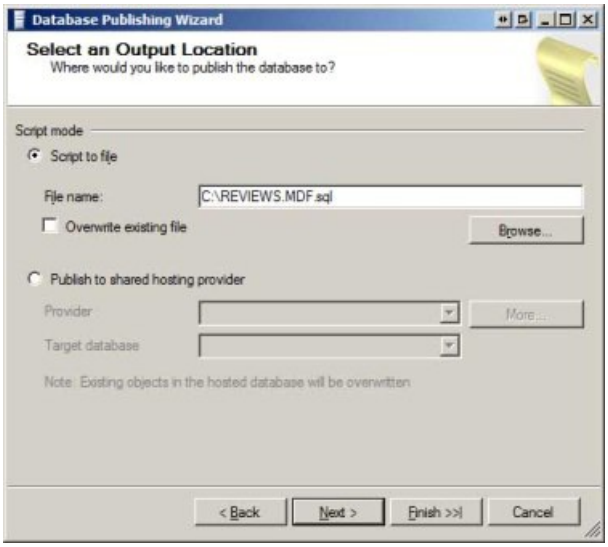


Figure 7: Script the Database to a File or Publish it Directly to Your Web Host Provider (Click to view full-size image)

The subsequent screen prompts you for a variety of scripting options. You can specify whether the script should include drop statements to remove these existing objects. This defaults to True, which is fine when deploying a database for the first time. You can also specify whether the target database is SQL Server 2000, SQL Server 2005, or SQL Server 2008. Finally, you can indicate whether to script the schema and data, just the data, or just the schema. The schema is the collection of database objects, the tables, stored procedures, views, and so on. The data is the information residing in the tables.

As Figure 8 illustrates, I ve got the wizard configured to drop existing database objects, to generate script for a SQL Server 2008 database, and to publish both the schema and data.

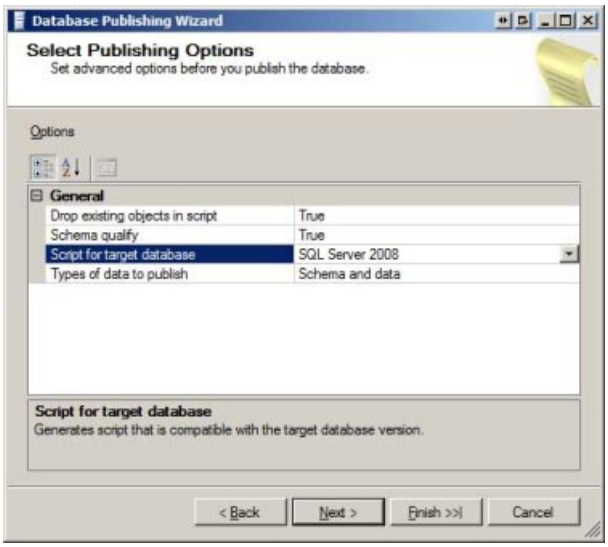


Figure 8: Specify the Publishing Options (Click to view full-size image)

The final two screens summarize the actions that are about to be taken and then display the status of the scripting. The net result of running the wizard is that we have a script file that contains the SQL commands needed to create the database on production and populate it with the same data as on development.

Executing the SQL Commands On the Production Environment Database

Now that we have the script that contains the SQL commands to create the database and its data all that remains is to execute the script on the production database. Some web host providers offer a textbox in their Control Panel where you can enter SQL commands to execute on your database. If you have a very large script file then this option might not work (the `REVIEWS.MDF.sql` script file is over 425 KB in size, for instance).

A better approach is to connect directly to the production database server using SQL Server Management Studio (SSMS). If you have a non-Express Edition of SQL Server installed on your computer then you likely already have SSMS installed. Otherwise, you can download and install a free copy of SQL Server Management Studio Express Edition.

Launch SSMS and connect to your web host s database server using the information provided by your web host provider.

Figure 9: Connect to Your Web Host Provider s Database Server (Click to view full-size image)

Expand the Databases tab and locate your database. Click the New Query button in the upper left corner of the Toolbar, paste in the SQL commands from the script file created by the Database Publishing Wizard, and click the Execute button to run these commands on the production database server. If your script file is especially large it may take several minutes to execute the commands.

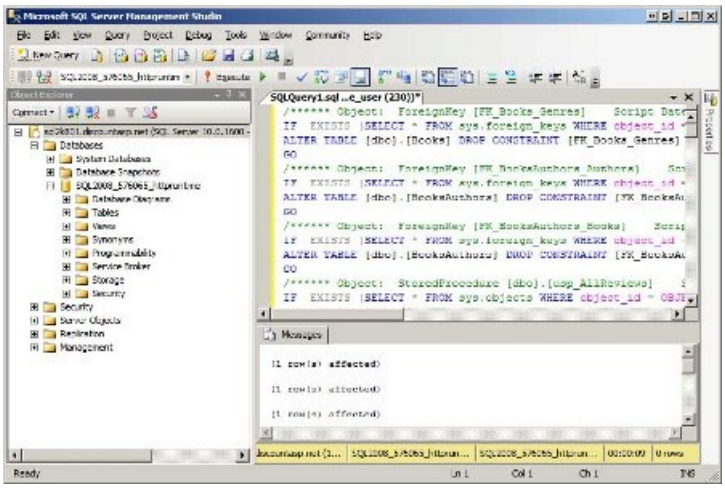


Figure 10: Connect to Your Web Host Provider s Database Server (Click to view full-size image)

That s all there is to it! At this point the development database has been duplicated to production. If you Refresh the database in SSMS you should see the new database objects. Figure 11 shows the production database s tables, stored procedures, and user-defined functions, which mirror those on the development database. And because we instructed the Database Publishing Wizard to publish the data, the production database s tables have the same data as the development database s tables at the time the wizard was executed. Figure 12 shows the data in the `Books` table on the production database.

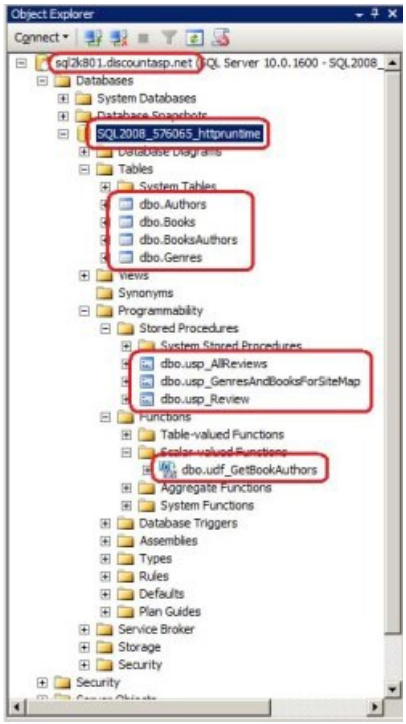


Figure 11: The Database Objects Have Been Duplicated on the Production Database (Click to view full-size image)

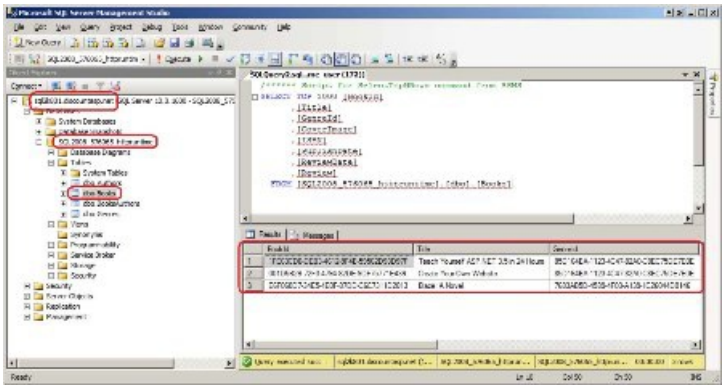


Figure 12: The Production Database Contains the Same Data as on the Development Database (Click to view full-size image)

At this point we have only deployed the development database to production. We have not yet looked at deploying the web application itself or examined what configuration changes are needed to have the application on production use the production database. We'll cover these issues in the next tutorial!

Summary

Deploying a data-driven web application requires copying the database used during development to the production environment. Many web host providers offer tools to simplify the process of deploying a database. For example, with DiscountASP.NET you can FTP your database `.mdf` file (or a backup) and then attach the database to the database server from the Control Panel. Another option that works regardless of what features your web host provider offers is Microsoft s Database Publishing Wizard tool, which generates a script of SQL commands to create the development database s schema and data. Once this script has been generated you can execute it on the production database.

Now that the Book Reviews web application s database is on production we can deploy the application. However, the web application s configuration information specifies the connection string to the database, and that connection string references the development database. We need to update this connection string information when deploying the site to production. The next tutorial looks at these configuration differences and walks through the steps needed to publish the data-driven Book Reviews site to production.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Download the Microsoft SQL Server Database Publishing Wizard 1.1
- Download the Microsoft SQL Server Management Studio Express Edition

PREVIOUS NEXT

Configuring the Production Web Application to Use the Production Database (VB)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> As discussed in earlier tutorials, it is not uncommon for configuration information to differ between the development and production environments. This is especially true for data-driven web applications, as the database connection strings differ between the development and production environments. This tutorial explores ways to configure the production environment to include the appropriate connection string in more detail.

Introduction

Data-driven web applications usually use a different database when in development than when in production. For applications hosted by a web host provider and developed locally, the development database typically resides on the developer s computer while the production database is hosted on a database server at the web hosting company s facility. Deploying a data-driven web application entails copying the development database to the production database server. In the previous tutorial we looked at ways to accomplish this step.

The web application uses the information in a *connection string* to establish a connection with the database. The connection string, which is typically stored in `Web.config`, specifies the database server name, the name of the database, the security context, and other information. Because the database used by the web application depends on whether the web application is running in the development or production environments, the connection strings must differ between the two environments.

It is not uncommon for configuration information to differ between the development and production environments. The *Common Configuration Differences Between Development and Production* tutorial discussed techniques for maintaining separate configuration information between these two environments, as well as a brief discussion on database connection strings. This tutorial explores ways to configure the production environment to include the appropriate connection string in more detail.

Examining the Connection String Information

The connection string used by the Book Reviews web application is stored in the application s configuration file, `Web.config`. `Web.config` includes a special section for storing connection strings, aptly named <connectionStrings>. The `Web.config` file for the Book Reviews website has one connection string defined in this section named `ReviewsConnectionString`:

```
<configuration>
    ...
    <connectionStrings>

        <add name="ReviewsConnectionString"
            connectionString="Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\Reviews.mdf;Integrated
            Security=True;User Instance=True"
            providerName="System.Data.SqlClient" />

    </connectionStrings>
    ...
</configuration>
```

The connection string - Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\Reviews.mdf;Integrated Security=True;User Instance=True - is composed of a number of options and values, with option/value pairs delimited by a semicolon and each option and value delimited by an equals sign. The four options used in this connection string are:

- `Data Source` - specifies the location of the database server and the database server instance name (if any). The value, `.\SQLEXPRESS`, is an example where there is a database server and an instance name. The period specifies that the database server is on the same computer as the application; the instance name is `SQLEXPRESS` .
- `AttachDbFilename` - specifies the location of the database file. The value contains the placeholder `|DataDirectory|`, which is resolved to the full path of the application s `App_Data` folder at runtime.
- `Integrated Security` - a Boolean value that indicates whether to use a specified username/password when connecting to the database (false) or the current Windows account credentials (true).
- `User Instance` - a configuration option specific to the SQL Server Express Editions that indicates whether to allow non-Administrative users on the local computer attach and connect to a SQL Server Express Edition database. See SQL Server Express User Instances for more information on this setting.

The allowable connection string options depend on the database you are connecting to and the ADO.NET database provider being used. For example, the connection string for connecting to a Microsoft SQL Server database differs from that used to connect to an Oracle database. Likewise, connecting to a Microsoft SQL Server database using the SqlClient provider uses a different connection string than when using the OLE-DB provider.

You can build the database connection string by hand using a site like ConnectionStrings.com as a resource for what options are available. However, an easier approach is to add the database to the Server Explorer in Visual Studio and then to grab the connection string from the Properties window. Let s use this latter technique for constructing the connection string to the production database server.

Open Visual Studio and then navigate to the Server Explorer window (in Visual Web Developer, this window is called Database Explorer). Right-click on the Data Connections option and choose the Add Connection option from the context menu. This brings up the wizard shown in Figure 1. Choose the appropriate data source and click Continue.

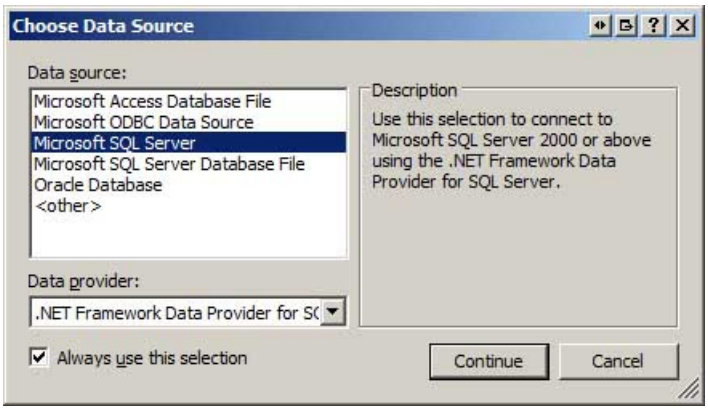


Figure 1: Choose to Add a New Database to the Server Explorer (Click to view full-size image)

Next, specify the various database connection information (see Figure 2). When you signed up with your web hosting company they should have provided information on how to connect to the database - the database server name, the database name, the username and password to use to connect to the database, and so on. After entering this information click OK to complete this wizard and to add the database to the Server Explorer.

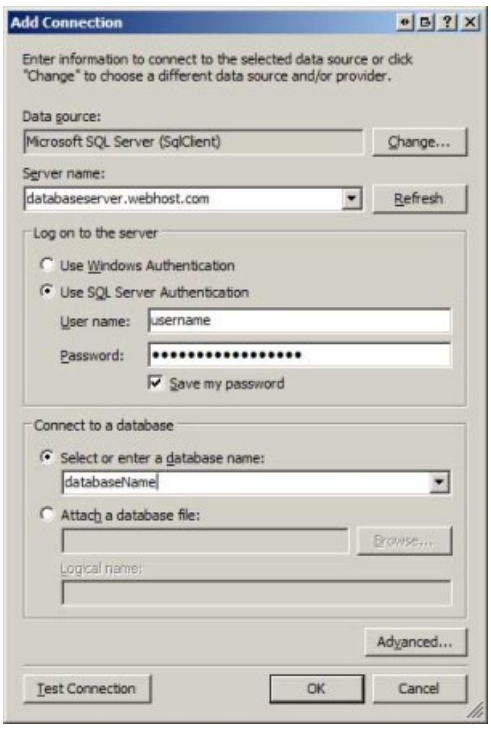


Figure 2: Specify the Database Connection Information (Click to view full-size image)

The production environment database should now be listed in the Server Explorer. Select the database from the Server Explorer and go to the Properties window. There you will find a property named Connection String with the database s connection string. Assuming you are using a Microsoft SQL Server database on production and the SqlClient provider your connection string should look similar to the following:

Data Source=*serverName*; Initial Catalog=*databaseName*; Persist Security Info=True; User ID=*username*; Password=*password*

Where *serverName*, *databaseName*, *username*, and *password* are with the values for the database server name, the database name, and the username and password supplied to you by your web host company.

Deploying the Book Reviews Web Application

The preceding tutorial stepped through copying the development database to the production environment, but did not explore deploying the data-driven application. At this point the production environment contains the database but is using the version of the Book Reviews application with static reviews. We need to deploy the new,

data-driven application to the production server along with the updated configuration information.

Take a moment to deploy the data-driven application from the development environment to production. This process was covered in detail in previous tutorials. If you need a refresher, see the *Deploying Your Website Using an FTP Client* or the *Deploying Your Website Using Visual Studio* tutorials. You will need to ensure that the production database connection string is the one used in the production environment, which means that an alternate `Web.config` file must be deployed. Specifically, this modified `Web.config` file s `<connectionStrings>` element needs to contain the production database connection string and should look similar to the following:

```
<configuration>

    ...
    <connectionStrings>

        <add name="ReviewsConnectionString"
            connectionString="Data Source=serverName; Initial Catalog=databaseName;
            Persist Security Info=True; User ID=username; Password=password"
            providerName="System.Data.SqlClient" />

    </connectionStrings>
    ...
</configuration>
```

Note that the connection string in the `<connectionStrings>` element is named the same (`ReviewsConnectionString`), but now contains the production database connection string instead of the development database connection string.

Unless you have a more formalized deployment workflow, either manually modify the `Web.config` file to use the production database connection string before deploying (remembering to revert it back to using the development database connection string afterwards) or maintain a separate `Web.config` file with the production environment configuration information that gets uploaded to the production environment as part of the deployment process.

> **NOTE**
> If you accidentally deploy a `Web.config` file that contains the development database connection string then there will be an error when the application on production attempts to connect to the database. This error manifests as a `SqlException` with a message reporting that the server was not found or was not accessible.

Once the site has been deployed to production, visit the production site through your browser. You should see and enjoy the same user experience as when running the data-driven application locally. Of course when you visit the website on production the site is powered by the production database server, whereas visiting the website in the development environment uses the database in development. Figure 3 shows the *Teach Yourself ASP.NET 3.5 in 24 Hours* review page from the website in the production environment (note the URL in the browser s Address bar).

Figure 3: The Data-Driven Application is Now Available On Production! (Click to view full-size image)

Storing Connection Strings in a Separate Configuration File

A common technique for maintaining separate configuration information on the development and production environments is to have two versions of the `Web.config`: one for the development environment and one for production. At deploy-time the appropriate `Web.config` version can be copied to the production environment. Ideally, this process would be automated as part of the deployment workflow.

Instead of maintaining two separate `Web.config` files you can, optionally, provide more granular differences. The elements that make up the `Web.config` file can be defined in an external configuration files that are then referenced in the `Web.config` file. In a nutshell you can have one `Web.config` file for both environments that references a databaseConnectionStrings.config file, which would contain the connection strings used by the application and would be unique for each environment. I find that separating the differing configuration information into separate files provides a tidier and simpler `Web.config` file and more clearly outlines the configuration differences between the development and production environments.

To use this technique, start by creating a new folder in the web application named `ConfigSections`. Next, add two files to this new folder named databaseConnectionStrings.dev.config and databaseConnectionStrings.production.config. Next, copy the `<connectionStrings>` element from `Web.config` into the databaseConnectionStrings.dev.config and databaseConnectionStrings.production.config files, and then modify the connection string in the databaseConnectionStrings.production.config file so that it specifies the production database connection string. For example, the databaseConnectionStrings.dev.config file should contain just the `<connectionStrings>` element with a connection string that references the development database:

```
<?xml version="1.0"?>
<connectionStrings>
    <add name="ReviewsConnectionString"
        connectionString="Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\Reviews.mdf;Integrated
        Security=True;User Instance=True"
        providerName="System.Data.SqlClient" />

</connectionStrings>
```

Similarly, the databaseConnectionStrings.production.config file should contain just a `<connectionStrings>` element, but one that has the production database connection string.

Make a copy of the databaseConnectionStrings.dev.config file and name it databaseConnectionStrings.config.

NOTE

You can name the configuration file something other than databaseConnectionStrings.config, if you d like, such as `connectionStrings.config` or `dbInfo.config`. However, be sure to name the file with a `.config` extension as `.config` files are, by default, not served by the ASP.NET engine. If you name the file something else, like `connectionStrings.txt`, a user could point their browser to www.yoursite.com/ConfigSettings/connectionStrings.txt and view the contents of the file!

At this point the `ConfigSections` folder should contain three files (see Figure 4). The databaseConnectionStrings.dev.config and databaseConnectionStrings.production.config files contain the connection strings for the development and production environments, respectively. The databaseConnectionStrings.config file contains the connection string information that will be used by the web application at runtime. Consequently, the databaseConnectionStrings.config file should be identical to the databaseConnectionStrings.dev.config file in the development environment, whereas on production the databaseConnectionStrings.config file should be identical to databaseConnectionStrings.production.config.

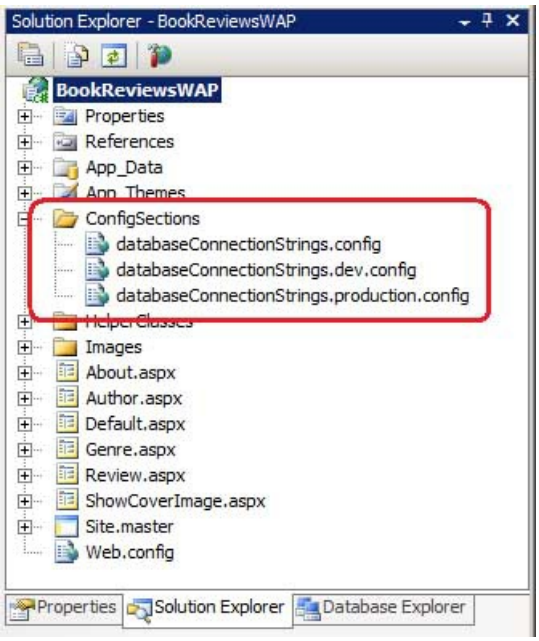


Figure 4: ConfigSections (Click to view full-size image)

We now need to instruct `Web.config` to use the databaseConnectionStrings.config file for its connection string store. Open `Web.config` and replace the existing `<connectionStrings>` element with the following:

```
<configuration>
    ...
    <connectionStrings
        configSource="ConfigSections\databaseConnectionStrings.config"
    />
    ...
</configuration>
```

The `configSource` attribute specifies a physical path relative to the `Web.config` file. If the external `.config` file is in the same directory as `Web.config` then set this attribute to the file name of the `.config` file. If it s in a subdirectory, as is the case with databaseConnectionStrings.config, specify the subfolder using a backslash to delimit the folder and file names, like ConfigSections\databaseConnectionStrings.config.

With this modification, the development and production environments contain the same `Web.config` file. Now the only difference is the databaseConnectionStrings.config file. Copy the databaseConnectionStrings.production.config file to production and rename it to

databaseConnectionStrings.config. If, in the future, there are changes to the production database connection string you will need to make them to the databaseConnectionStrings.production.config file and then upload that file to production, renaming it databaseConnectionStrings.config.

> **NOTE**
> You may specify the information for any `Web.config` element in a separate file and use the `configSource` attribute to reference that file from within `Web.config`.

Summary

Data-driven applications usually use different databases in the development and production environments. Consequently, the database connection strings stored in the web application s configuration must be unique per environment. In this tutorial we looked at how to determine the production database connection string and ways to maintain unique connection string information in the two environments.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Connection Strings and Configuration Files
- Database Configuration Strings Information @ ConnectionStrings.com
- Move Settings Out of the Web.config File
- Technical Documentation for the <connectionStrings> Element

PREVIOUS NEXT

Configuring a Website that Uses Application Services (VB)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

ASP.NET version 2.0 introduced a series of application services, which are part of the .NET Framework and serve as a suite of building block services that you can use to add rich functionality to your web application. This tutorial explores how to configure a website in the production environment to use application services and addresses common issues with managing user accounts and roles on the production environment.

Introduction

ASP.NET version 2.0 introduced a series of *application services*, which are part of the .NET Framework and serve as a suite of building block services that you can use to add rich functionality to your web application. The application services include:

- **Membership** - an API for creating and managing user accounts.
- **Roles** - an API for categorizing users into groups.
- **Profile** - an API for storing custom, user-specific content.
- **Site Map** - an API for defining a site s logical structure in the form of a hierarchy, which can then be displayed via navigation controls, such as menus and breadcrumbs.
- **Personalization** - an API for maintaining customization preferences, most often used with *WebParts*.
- **Health Monitoring** - an API for monitoring performance, security, errors, and other system health metrics for a running web application.

The application services APIs are not tied to a specific implementation. Instead, you instruct the application services to use a particular *provider*, and that provider implements the service using a particular technology. The most commonly used providers for Internet-based web applications hosted at a web hosting company are those providers that use a SQL Server database implementation. For example, the `SqlMembershipProvider` is a provider for the Membership API that stores user account information in a Microsoft SQL Server database.

Using the application services and SQL Server providers adds some challenges when deploying the application. For starters, the application services database objects must be properly created on both the development and production databases and appropriately initialized. There are also important configuration settings that need to be made.

> **NOTE**
>
> The application services APIs were designed using the *provider model*, a design pattern that allows for an API s implementation details to be provided at runtime. The .NET Framework ships with a number of application service providers that can be used, such as the `SqlMembershipProvider` and `SqlRoleProvider`, which are providers for the Membership and Roles APIs that use a SQL Server database implementation. You can also create and plug-in a custom provider. In fact, the Book Reviews web application already contains a custom provider for the Site Map API (`ReviewSiteMapProvider`), which constructs the site map from the data in the `Genres` and `Books` tables in the database.

This tutorial starts with a look at how I extended the Book Reviews web application to use the Membership and Roles APIs. It then walks through deploying a web application that uses application services with a SQL Server database implementation, and concludes by addressing common issues with managing user accounts and roles on the production environment.

Updates to the Book Reviews Application

Over the past couple tutorials the Book Reviews web application was updated from a static website to a dynamic, data-driven web application complete with a set of administration pages for managing genres and reviews. However, this administration section is currently not protected - any user who knows (or guesses) the administration page URL can waltz in and create, edit, or delete reviews on our site. A common way to protect certain portions of a website is to implement user accounts and then use URL authorization rules to restrict access to certain users or roles. The Book Reviews web application available for download with this tutorial supports user accounts and roles. It has a single role defined named Admin and only users in this role can access the administration pages.

> **NOTE**
> I ve created three user accounts in the Book Reviews web application: Scott, Jisun, and Alice. All three users have the same password: **password!** Scott and Jisun are in the Admin role, Alice is not. The site s non-administration pages are still accessible to anonymous users. That is, you do not need to sign in to visit the site, unless you want to administer it, in which case you must sign in as a user in the Admin role.

The Book Reviews application s master page has been updated to include a different user interface for authenticated and anonymous users. If an anonymous user visits the site she sees a Login link in the upper right corner. An authenticated user sees the message, "Welcome back, *username*!" and a link to log out. There s also a login page (`~/Login.aspx`), which contains a Login Web control that provides the user interface and logic for authenticating a visitor. Only administrators can create new accounts. (There are pages for creating and managing user accounts in the `~/Admin` folder.)

Configuring the Membership and Roles APIs

The Book Reviews web application uses the Membership and Roles APIs to support user accounts and to group those users into roles (namely, the Admin role). The `SqlMembershipProvider` and `SqlRoleProvider` provider classes are used because we want to store account and role information in a SQL Server database.

> **NOTE**
> This tutorial is not intended to be a detailed examination at configuring a web application to support the Membership and Roles APIs. For a thorough look at these APIs and the steps you need to take to configure a website to use them, please read my *Website Security Tutorials*.

To use the application services with a SQL Server database you must first add the database objects used by these providers to the database where you want the user account and role information stored. These requisite database objects include a variety of tables, views, and stored procedures. Unless specified otherwise, the `SqlMembershipProvider` and `SqlRoleProvider` provider classes use a SQL Server Express Edition database named `ASPNETDB` located in the application s `App_Data` folder; if such a database does not exist, it is automatically created with the necessary database objects by these providers at runtime.

It is possible, and usually ideal, to create the application services database objects in the same database where the website s application-specific data is stored. The .NET Framework ships with a tool named `aspnet_regsql.exe` that installs the database objects on a specified database. I have gone ahead and used this tool to add these objects to the `Reviews.mdf` database in the `App_Data` folder (the development database). We'll see how to use this tool later in this tutorial when we add these objects to the production database.

If you add the application services database objects to a database other than `ASPNETDB` you will need to customize the `SqlMembershipProvider` and `SqlRoleProvider` provider classes configurations so that they use the appropriate database. To customize the Membership provider add a *<membership> element* within the `<system.web>` section in `Web.config`; use the *<roleManager> element* to configure the Roles provider. The following snippet is taken from the Book Reviews application s `Web.config` and shows the configure settings for the Membership and Roles APIs. Note that both register a new provider - `ReviewMembership` and `ReviewRole` - that use the `SqlMembershipProvider` and `SqlRoleProvider` providers, respectively.

```
<configuration>
    <system.web>
        ...

        <membership defaultProvider="ReviewMembership">
            <providers>
                <clear />

                <add type="System.Web.Security.SqlMembershipProvider"
                     name="ReviewMembership"
                     connectionStringName="ReviewsConnectionString"
                     applicationName="BookReviews" />
            </providers>
        </membership>

        <roleManager enabled="true" defaultProvider="ReviewRole">
            <providers>
                <clear />

                <add type="System.Web.Security.SqlRoleProvider"
                     name="ReviewRole"
                     connectionStringName="ReviewsConnectionString"
                     applicationName="BookReviews" />
            </providers>
        </roleManager>

        ...
    </system.web>
</configuration>
```

The `Web.config` file s `<authentication>` element has also been configured to support forms-based authentication.

```
<configuration>
    <system.web>
        ...

        <authentication mode="Forms" />

        ...
    </system.web>
</configuration>
```

Limiting Access to the Administration Pages

ASP.NET makes it easy to grant or deny access to a particular file or folder by user or by role via its *URL authorization* feature. (We briefly discussed URL authorization in the *Core Differences Between IIS and the ASP.NET Development Server* tutorial and showed how IIS and the ASP.NET Development Server apply URL authorization rules differently for static versus dynamic content.) Because we want to prohibit access to the `~/Admin` folder except for those users in the Admin role, we need to add URL authorization rules to this folder. Specifically, the URL authorization rules need to allow users in the Admin role and deny all other users. This is accomplished by adding a `Web.config` file to the `~/Admin` folder with the following contents:

```
<?xml version="1.0"?>
<configuration>
    <system.web>
        <authorization>
            <allow roles="Admin" />
            <deny users="*" />
        </authorization>
    </system.web>
</configuration>
```

For more information on ASP.NET s URL authorization feature and how to use it to spell out authorization rules for users and for roles, be sure to read the *User-Based Authorization* and *Role-Based Authorization* tutorials from my *Website Security Tutorials*.

Deploying a Web Application That Uses Application Services

When deploying a website that uses application services and a provider that stores the application services information in a database, it is imperative that the database objects needed by the application services be created on the production database. Initially the production database does not contain these objects, so when the application is first deployed (or when it is deployed for the first time after application services have been added), you must take extra steps to get these requisite database objects on the production database.

Another challenge can arise when deploying a website that uses application services if you intend to replicate the user accounts created in the development environment to the production environment. Depending on the Membership and Roles configuration, it is possible that even if you successfully copy the user accounts that were created in the development environment to the production database, these users cannot sign into the web application in production. We'll look at the cause of this issue and discuss how to prevent it from happening.

ASP.NET ships with a nice *Web Site Administration Tool (WSAT)* that can be launched from Visual Studio and allows the user account, roles, and authorization rules to be managed through a web-based interface. Unfortunately, the WSAT only works for local websites, meaning that it cannot be used to remotely manage user accounts, roles, and authorization rules for the web application in the production environment. We'll look at different ways to implement WSAT-like behavior from your production website.

Adding the Database Objects Using aspnet_regsql.exe

The *Deploying a Database* tutorial showed how to copy the tables and data from the development database to the production database, and these techniques can certainly be used to copy the application services database objects to the production database. Another option is the `aspnet_regsql.exe` tool, which adds or removes the application services database objects from a database.

> **NOTE**
>
> The `aspnet_regsql.exe` tool creates the database objects on a specified database. It does not migrate data in those database objects from the development database to the production database. If you mean to copy the user account and role information in the development database to the production database use the techniques covered in the *Deploying a Database* tutorial.

Let s look at how to add the database objects to the production database using the `aspnet_regsql.exe` tool. Start by opening Windows Explorer and navigating to the .NET Framework version 2.0 directory on your computer, %WINDIR%\ Microsoft.NET\Framework\v2.0.50727. There you should find the `aspnet_regsql.exe` tool. This tool can be used from the command-line, but it also includes a graphical user interface; double-click the `aspnet_regsql.exe` file to launch its graphical component.

The tool starts by displaying a splash screen explaining its purpose. Click Next to advance to the "Select a Setup Option" screen, which is shown in Figure 1. From here you can choose to add the application services database

objects or remove them from a database. Because we want to add these objects to the production database, select the "Configure SQL Server for application services" option and click Next.

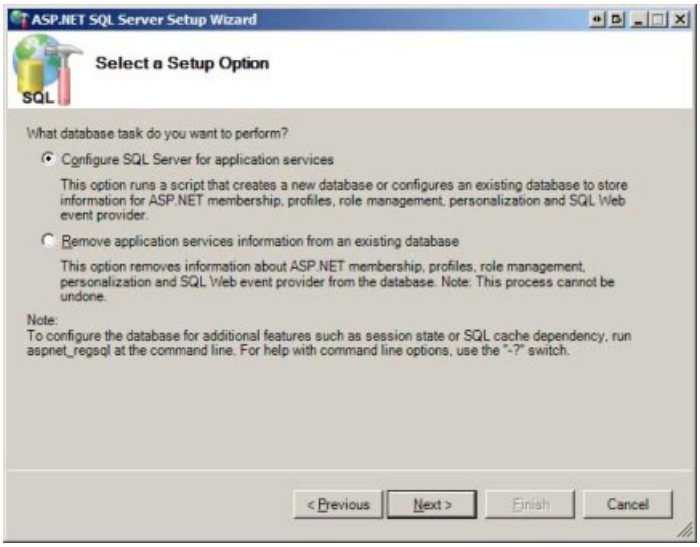


Figure 1: Choose to Configure SQL Server for Application Services (Click to view full-size image)

In "Select the Server and Database" screen prompts for information to connect to the database. Enter the database server, the security credentials, and the database name supplied to you by your web hosting company and click Next.

> **NOTE**
>
> After entering your database server and credentials you may get an error when expanding the database drop-down list. The `aspnet_regsql.exe` tool queries the `sysdatabases` system table to retrieve a list of databases on the server, but some web hosting companies lock down their database servers so that this information is not publicly available. If you get this error you can type the database name directly into the drop-down list.

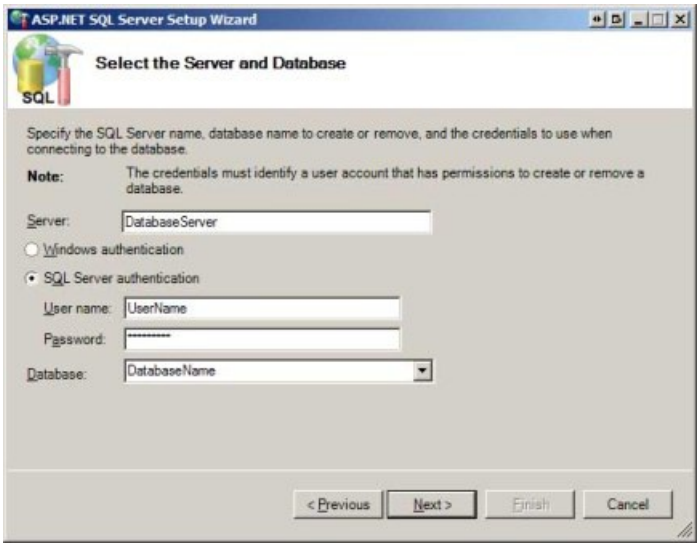


Figure 2: Supply the Tool With Your Database s Connection Information (Click to view full-size image)

The subsequent screen summarizes the actions that are about to take place, namely that the application services database objects are going to be added to the specified database. Click Next to complete this action. After a few moments, the final screen is displayed, noting that the database objects have been added (see Figure 3).

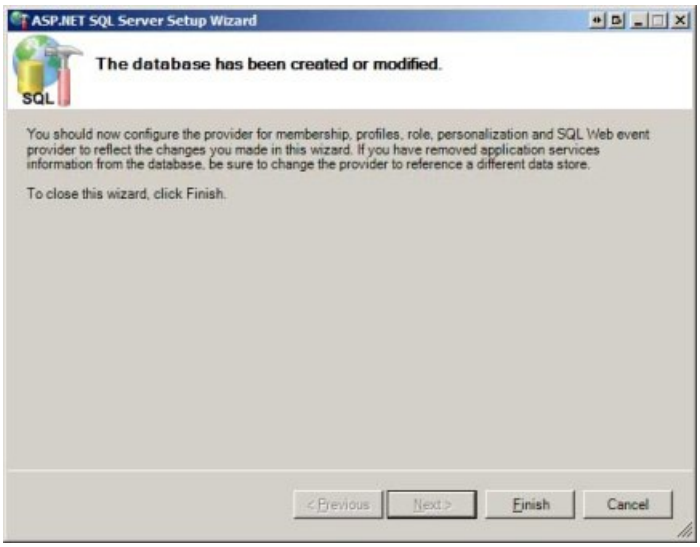


Figure 3: Success! The Application Services Database Objects Were Added to the Production Database (Click to view full-size image)

To verify that the application services database objects were successfully added to the production database, open SQL Server Management Studio and connect to your production database. As Figure 4 shows, you should now see the application services database tables in your database, `aspnet_Applications`, `aspnet_Membership`, `aspnet_Users`, and so forth.

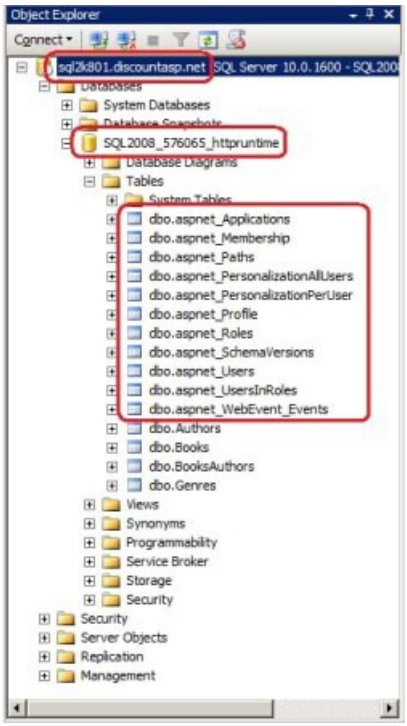


Figure 4: Confirm That the Database Objects Were Added to the Production Database (Click to view full-size image)

You will only need to use the `aspnet_regsql.exe` tool when deploying your web application for the first time or for the first time after you have started using the application services. Once these database objects are on the production database they won't need to be re-added or modified.

Copying User Accounts from Development to Production

When using the `SqlMembershipProvider` and `SqlRoleProvider` provider classes to store the application services information in a SQL Server database, the user account and role information is stored in a variety of database tables, including `aspnet_Users`, `aspnet_Membership`, `aspnet_Roles`, and `aspnet_UsersInRoles`, among others. If during development you create user accounts in the development environment you can replicate those user

accounts in production by copying the corresponding records from the applicable database tables. If you used the Database Publishing Wizard to deploy the application services database objects you may have also elected to copy the records, which would result in the user accounts created in development to also be on production. But, depending on your configuration settings, you may find that those users whose accounts were created in development and copied to production are unable to login from the production website. What gives?

The `SqlMembershipProvider` and `SqlRoleProvider` provider classes were designed such that a single database could serve as a user store for multiple applications, where each application could, in theory, have users with overlapping usernames and roles with the same name. To allow for this flexibility, the database maintains a list of applications in the `aspnet_Applications` table, and each user is associated with one of these applications. Specifically, the `aspnet_Users` table has an `ApplicationId` column that ties each user to a record in the `aspnet_Applications` table.

In addition to the `ApplicationId` column, the `aspnet_Applications` table also includes an `ApplicationName` column, which provides a more human-friendly name for the application. When a website attempts to work with a user account, such as validating a user s credentials from the login page, it must tell the `SqlMembershipProvider` class what application to work with. It usually does this by supplying the application name, and the this value comes from the provider s configuration in `Web.config` - specifically via the `applicationName` attribute.

But what happens if the `applicationName` attribute is not specified in `Web.config`? In such a case the Membership system uses the application root path as the `applicationName` value. If the `applicationName` attribute is not explicitly set in `Web.config`, then, there is the possibility that the development environment and production environment use a different application root and therefore will be associated with different application names in the application services. If such a mismatch occurs then those users created in the development environment will have an `ApplicationId` value that does not match with the `ApplicationId` value for the production environment. The net result is that those users won't be able to login.

> **NOTE**
> If you find yourself in this situation - with user accounts copied to production with a mismatched `ApplicationId` value - you could write a query to update these incorrect `ApplicationId` values to the `ApplicationId` used on production. Once updated, the users whose accounts were created on the development environment would now be able to sign into the web application on production.

The good news is that there is a simple step you can take to ensure that the two environments use the same `ApplicationId` - explicitly set the `applicationName` attribute in `Web.config` for all of your application services providers. I explicitly set the `applicationName` attribute to "BookReviews" in the `<membership>` and `<roleManager>` elements as this snippet from `Web.config` shows.

```
<membership defaultProvider="ReviewMembership">
    <providers>
        <clear />

        <add type="System.Web.Security.SqlMembershipProvider"
             name="ReviewMembership"
             connectionStringName="ReviewsConnectionString"
             applicationName="BookReviews" />
    </providers>
</membership>
```

For more discussion on setting the `applicationName` attribute and the rationale behind it, refer to *Scott Guthrie* s blog post, *Always set the applicationName property when configuring ASP.NET Membership and other Providers*.

Managing User Accounts in the Production Environment

The ASP.NET Web Site Administration Tool (WSAT) makes it easy to create and manage user accounts, define and

apply roles, and spell out user- and role-based authorization rules. You can launch the WSAT from Visual Studio by going to the Solution Explorer and clicking the ASP.NET Configuration icon or by going to the Website or Project menus and selecting the ASP.NET Configuration menu item. Unfortunately, the WSAT can only work with local websites. Therefore, you cannot use the WSAT from your workstation to manage the website in the production environment.

The good news is that all of the functionality exposed provided by the WSAT is available programmatically through the Membership and Roles APIs; furthermore, many of the WSAT screens use the standard ASP.NET Login-related controls. In short, you can add ASP.NET pages to your website that offer the necessary management capabilities.

Recall that an earlier tutorial updated the Book Reviews web application to include an `~/Admin` folder, and this folder has been configured to only allow users in the Admin role. I added a page to that folder named `CreateAccount.aspx` from which an administrator can create a new user account. This page uses the CreateUserWizard control to display the user interface and backend logic for creating a new user account. What s more, I customized the control to include a CheckBox that prompts whether the new user should also be added to the Admin role (see Figure 5). With a little bit of work you can build a custom set of pages that implements the user and role management-related tasks that would otherwise be provided by the WSAT.

> **NOTE**
>
> For more information on using the Membership and Roles APIs along with the Login-related ASP.NET Web controls, be sure to read my *Website Security Tutorials*. For more on customizing the CreateUserWizard control refer to the *Creating User Accounts* and *Storing Additional User Information* tutorials, or check out *Erich Peterson* s article, *Customizing the CreateUserWizard Control*.

Figure 5: Administrators Can Create New User Accounts (Click to view full-size image)

If you need the full functionality of the WSAT check out *Rolling Your Own Web Site Administration Tool*, in which author Dan Clem walks through the process of building a custom WSAT-like tool. Dan shares his application s source code (in C#) and provides step-by-step instructions for adding it to your hosted website.

Summary

When deploying a web application that uses the application services database implementation you must first ensure that the production database has the requisite database objects. These objects can be added using the techniques discussed in the *Deploying a Database* tutorial; alternatively, you can use the `aspnet_regsql.exe` tool, as we saw in this tutorial. Other challenges we touched on center around synchronizing the application name used in the development and production environments (which is important if you want users and roles created in

the development environment to be valid on production) and techniques for managing the users and roles in the production environment.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- *ASP.NET SQL Server Registration Tool (aspnet_regsql.exe)*
- *Creating the Application Services Database for SQL Server*
- *Creating the Membership Schema in SQL Server*
- *Examining ASP.NET s Membership, Roles, and Profile*
- *Rolling Your Own Web Site Administration Tool*
- *Website Security Tutorials*
- *Web Site Administration Tool Overview*

PREVIOUS NEXT

Strategies for Database Development and Deployment (VB)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download PDF

When deploying a data-driven application for the first time you can blindly copy the database in the development environment to the production environment. But performing a blind copy in subsequent deployments will overwrite any data entered into the production database. Instead, deploying a database involves applying the changes made to the development database since the last deployment onto the production database. This tutorial examines these challenges and offers various strategies to assist with chronicling and applying the changes made to the database since the last deployment.

Introduction

As discussed in previous tutorials, deploying an ASP.NET application entails copying the pertinent content from the development environment to the production environment. Deployment is not a one-time event, but rather something that happens every time a new version of the software is released or when bugs or security concerns have been identified and addressed. When copying ASP.NET pages, images, JavaScript files, and other such files to the production environment you do not need to concern yourself with how these file have been changed since the last deployment. You can blindly copy the file to production, overwriting the existing content. Unfortunately, this simplicity does not extend to deploying the database.

When deploying a data-driven application for the first time you can blindly copy the database in the development environment to the production environment. But performing a blind copy in subsequent deployments will overwrite any data entered into the production database. Instead, deploying a database involves applying the *changes* made to the development database since the last deployment onto the production database. This tutorial examines these challenges and offers various strategies to assist with chronicling and applying the changes made to the database since the last deployment.

The Challenges of Deploying a Database

Before a data-driven application has been deployed for the first time, there is only one database, namely the database in the development environment, which is why when deploying a data-driven application for the first time you can blindly copy the database in the development environment to the production environment. But once the application has been deployed there are two copies of the database: one in development and one in production.

Between deployments the development and production databases can become out of sync. While the production database s schema remains unchanged, the development database s schema may change as new features are added. You might add or remove columns, tables, views, or stored procedures. There may also be important data that gets added to the development database. Many data-driven applications include lookup tables populated with hard-coded, application-specific data that are not user-editable. For example, an auction website might have a drop-down list with choices that describe the condition of the item being auctioned: New, Like New, Good, and Fair. Rather than hard-coding these options directly in the drop-down list it is usually better to place them in a database table. If, during development, a new condition named Poor is added to the table then when deploying the application this same record needs to be added to the lookup table in the production database.

Ideally, deploying the database would involve copying the database from development to production. But keep in mind that after you have deployed the application and resumed development, the production database is being populated with real data from real users. Therefore, if you were to simply copy the database from development to production at the next deployment you would overwrite the production database and lose its existing data. The net result is that deploying the database boils down to applying the changes made to the development database since the last deployment.

Because deploying a database involves applying the changes in the schema and, possibly, the data since the last deployment, a history of changes must be maintained (or determined at deploy-time) so that those changes can be applied on production. There are a variety of techniques for managing and applying changes to the data model.

Defining the Baseline

To maintain the changes to your application s database you need to have some starting state, a baseline to which the changes are applied to. At one extreme the starting state could be an empty database with no tables, views, or stored procedures. Such a baseline results in a large change log because it must include the creation of all of the database s tables, views, and stored procedures along with any changes made after the initial deployment. At the other end of the spectrum you could set the baseline as the version of the database that is initially deployed to the production environment. This choice results in a much smaller change log because it only includes the changes made to the database following the first deployment. This is the approach I prefer. And of course you can choose a more middle of the road approach, defining the baseline as some point between the initial creation of the database and when the database is first deployed.

Once you have chosen a baseline consider generating a SQL script that can be executed to recreate the baseline version. Such a script makes it possible to quickly recreate the baseline version of the database. This functionality is especially useful in larger projects, where there may be multiple developers working on the project or additional environments, such as testing or staging, that each need their own copy of the database.

There are a variety of tools at your disposal to generate a SQL script of the baseline version. From SQL Server Management Studio (SSMS) you can right-click on the database, go to the Tasks submenu, and choose the Generate Scripts option. This launches the Script Wizard, which you can instruct to generate a file that contains the SQL commands to create your database s objects. Another option is the Database Publishing Wizard, which can generate the SQL commands to not only create the database schema, but also the data in the database tables. The Database Publishing Wizard was examined in detail back in the *Deploying a Database* tutorial. Regardless of what tool you use, in the end you should have a script file that you can use to recreate the baseline version of your database, should the need arise.

Documenting the Database Changes in Prose

The simplest way to maintain a log of changes to the data model during the development phase is to record the changes in prose. For example, if during development of an already-deployed application you add a new column to the `Employees` table, remove a column from the `Orders` table, and add a new table (`ProductCategories`), you would maintain a text file or Microsoft Word document with the following history:

CHANGE DATE	CHANGE DETAILS
2009-02-03:	Added column `DepartmentID` (`int`, NOT NULL) to the `Employees` table. Added a foreign key constraint from `Departments.DepartmentID` to `Employees.DepartmentID`.
2009-02-05:	Removed column `TotalWeight` from the `Orders` table. Data already captured in associated `OrderDetails` records.

CHANGE DATE	CHANGE DETAILS
2009-02-12:	Created the `ProductCategories` table. There are three columns: `ProductCategoryID` (`int` , `IDENTITY` , `NOT NULL`), `CategoryName` (`nvarchar(50)` , `NOT NULL`), and `Active` (`bit` , `NOT NULL`). Added a primary key constraint to `ProductCategoryID` , and a default value of 1 to `Active` .

There are a number of drawbacks to this approach. For starters, there is no hope for automation. Anytime these changes need to be applied to a database - such as when the application is deployed - a developer must manually implement each change, one at a time. Moreover, if you need to reconstruct a particular version of the database from the baseline using the change log, doing so will take more and more time as the size of the log grows. Another drawback to this method is that the clarity and level of detail of each change log entry is left to the person recording the change. In a team with multiple developers some may make more detailed, more readable, or more precise entries than others. Also, typos and other human-related data entry errors are possible.

The primary benefit of documenting the database changes in prose is simplicity. You don t need familiarity with the SQL syntax for creating and altering database objects. Instead, you can record the changes in prose and implement them through SQL Server Management Studio s graphical user interface.

Maintaining your change log in prose is, admittedly, not very sophisticated and won't work well with certain projects, such as ones that are large in scope, have frequent changes to the data model, or involve multiple developers. But I have seen this approach work quite well in small, one-man projects that have only occasional changes to the data model and where the solo developer does not have a strong background in the SQL syntax for creating and altering database objects.

> **NOTE**
> While the information in the change log is, technically, only needed until deploy-time, I recommend keeping a history of changes. But rather than maintaining a single, ever growing change log file, consider having a different change log file for each database version. Typically you will want to version the database each time it is deployed. By maintaining a log of change logs you can, starting from the baseline, recreate any database version by executing the change log scripts starting from version 1 and continuing until you reach the version you need to recreate.

Recording the SQL Change Statements

The primary drawback of maintaining the change log in prose is the lack of automation. Ideally, implementing the database changes to the production database at deploy-time would be as easy as clicking a button to execute a script rather than having to manually perform a list of instructions. Such automation is possible by maintaining a change log that contains those SQL commands used to alter the data model.

The SQL syntax includes a number of statements for creating and modifying various database objects. For example, the *CREATE TABLE statement*, when executed, creates a new table with the specified columns and constraints. The *ALTER TABLE statement* modifies an existing table, adding, removing, or modifying its columns or constraints. There are also statements to create, modify, and drop indexes, views, user-defined functions, stored procedures, triggers, and other database objects.

Returning to our earlier example, image that during development of an already-deployed application you add a new column to the `Employees` table, remove a column from the `Orders` table, and add a new table (`ProductCategories`). Such actions would result in a change log file with the following SQL commands:

```
-- Add the DepartmentID column

ALTER TABLE [Employees] ADD [DepartmentID]
int NOT NULL

-- Add a foreign key constraint between Departments.DepartmentID and Employees.DepartmentID
ALTER TABLE [Employees] ADD
CONSTRAINT [FK_Departments_DepartmentID]
      FOREIGN
KEY ([DepartmentID])
      REFERENCES
[Departments] ([DepartmentID])

-- Remove TotalWeight column from Orders
ALTER TABLE [Orders] DROP COLUMN
[TotalWeight]

-- Create the ProductCategories table

CREATE TABLE [ProductCategories]
(
      [ProductCategoryID]
int IDENTITY(1,1) NOT NULL,
      [CategoryName]
nvarchar(50) NOT NULL,
      [Active]
bit NOT NULL CONSTRAINT [DF_ProductCategories_Active]  DEFAULT
((1)),
      CONSTRAINT
[PK_ProductCategories] PRIMARY KEY CLUSTERED ( [ProductCategoryID])
)
```

Pushing these changes to the production database at deploy-time is a one-click operation: open SQL Server Management Studio, connect to your production database, open a New Query window, paste the contents of the change log, and click Execute to run the script.

Using a Comparison Tool to Synchronize the Data Models

Documenting database changes in prose is easy, but implementing the changes requires a developer to make each change on the production database one at a time; documenting the change SQL commands makes implementing those changes on the production database as easy and quick as clicking a button, but requires learning and mastering the SQL statements and syntax for creating and altering database objects. Database comparison tools take the best from both approaches and discard the worst.

A database comparison tool compares the schema or data of two databases and displays a summary report showing you how the databases differ. Then, with the click of a button, you can generate the SQL commands for synchronizing one or more database objects. In a nutshell, you can use a database comparison tool to compare the development and production databases at deploy-time, generating a file that contains the SQL commands that, when executed, will apply the changes to the production database s schema so that it mirrors the development database s schema.

There are a variety of third-party database comparison tools offered by many different vendors. One such example is *SQL Compare*, by *Red Gate Software*. Let s walk through the process of using SQL Compare to compare and synchronize the development and production databases schemas.

> **NOTE**
>
> At the time of this writing the current version of SQL Compare was version 7.1, with the Standard Edition costing $395. You can follow along by downloading a free 14-day trial.

When SQL Compare starts the Comparison Projects dialog box opens, showing the saved SQL Compare projects. Create a new project. This launches the Project Configuration wizard, which prompts for information about the databases to compare (see Figure 1). Enter the information for the development and production environment databases.

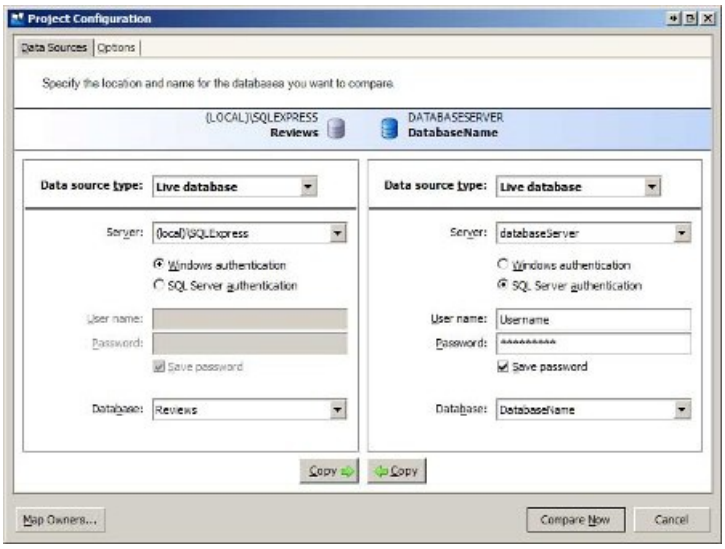


Figure 1: Compare the Development and Production Databases (Click to view full-size image)

> **NOTE**
>
> If your development environment database is a SQL Express Edition database file in the `App_Data` folder of your website you will need to register the database in the SQL Server Express database server in order to select it from the dialog box shown in Figure 1. The easiest way to accomplish this is to open SQL Server Management Studio (SSMS), connect to the SQL Server Express database server, and attach the database. If you do not have SSMS installed on your computer you can download and install the free *SQL Server 2008 Management Studio Basic version*.

In addition to selecting the databases to compare, you can also specify a variety of comparison settings from the Options tab. One option you may want to turn on is the "Ignore constraint and index names." Recall that in the preceding tutorial we added the application services database objects to the development and production databases. If you used the `aspnet_regsql.exe` tool to create these objects on the production database then you will find that the primary key and unique constraint names differ between the development and production databases. Consequently, SQL Compare will flag all of the application services tables as differing. You can either leave the "Ignore constraint and index names" unchecked and synchronize the constraint names, or instruct SQL Compare to ignore these differences.

After selecting the databases to compare (and reviewing the comparison options), click the Compare Now button to begin the comparison. Over the next several seconds, SQL Compare examines the schemas of the two databases and generates a report of how they differ. I ve purposefully made some modifications to the development database to show how such discrepancies are noted in the SQL Compare interface. As Figure 2 shows, I ve added a `BirthDate` column to the `Authors` table, removed the `ISBN` column from the `Books` table, and added a new table, `Ratings`, which is meant to let users visiting the site rate the reviewed books.

> **NOTE**
>
> The data model changes made in this tutorial were done to illustrate using a database comparison tool. You will not find these changes in the database in future tutorials.

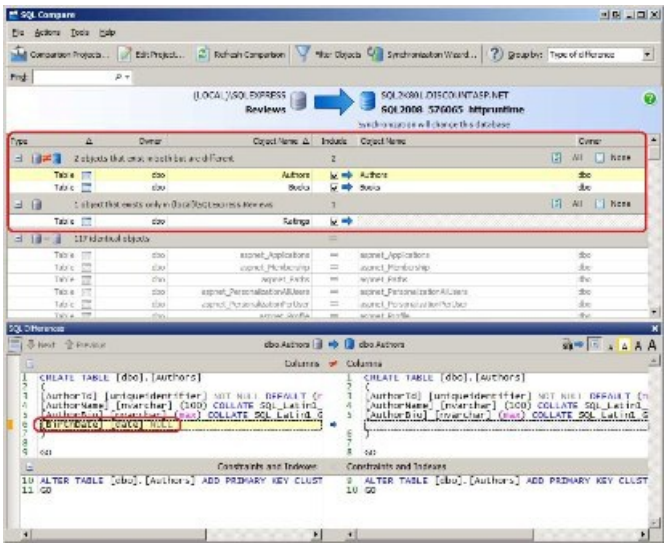


Figure 2: SQL Compare Lists the Differences Between the Development and Production Databases (Click to view full-size image)

SQL Compare breaks down the database objects into groups, quickly showing you what objects exist in both databases but are different, which objects exist in one database but not the other, and which objects are identical. As you can see, there are two objects that exist in both databases but are different: the `Authors` table, which had a column added, and the `Books` table, which had one removed. There is one object that exists only in the development database, namely the newly created `Ratings` table. And there are 117 objects that are identical in both databases.

Selecting a database object displays the SQL Differences window, which shows how these objects differ. The SQL Differences window, displayed at the bottom in Figure 2, highlights that the `Authors` table in the development database has the `BirthDate` column, which is not found in the `Authors` table on the production database.

After reviewing the differences and selecting which objects you want to synchronize, the next step is to generate the SQL commands needed to update the production database s schema to match the development database. This is accomplished through the Synchronization Wizard. The Synchronization Wizard confirms what objects to synchronize and summarizes the action plan (see Figure 3). You can synchronize the databases immediately or generate a script with the SQL commands that can be run at your leisure.

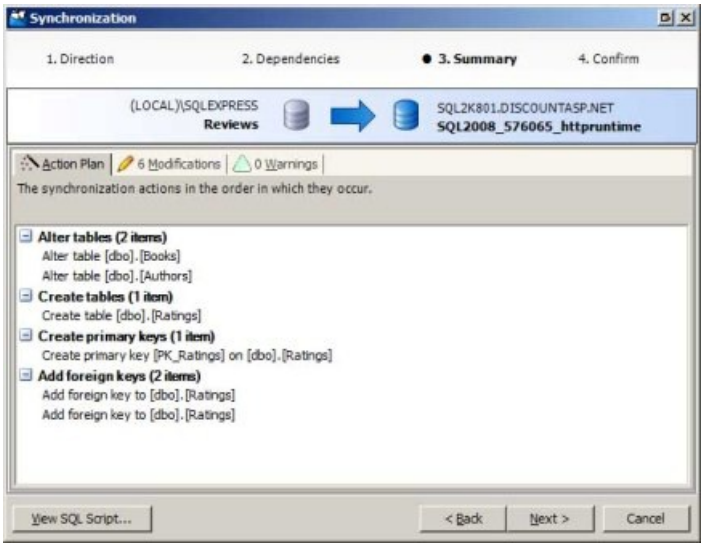


Figure 3: Use the Synchronization Wizard to Synchronize Your Databases Schemas (Click to view full-size image)

Database comparison tools like Red Gate Software s SQL Compare make applying the changes to the development database schema to the production database as easy as point and click.

> **NOTE**
> SQL Compare compares and synchronizes two databases *schemas*. Unfortunately, it does not compare and synchronize the data within two databases tables. Red Gate Software does offer a product named *SQL Data Compare* that compares and synchronizes the data between two databases, but it is a separate product from SQL Compare and costs another $395.

Taking the Application Offline During Deployment

As we ve seen throughout these tutorials, deployment is a process that involves multiple steps: copying the ASP.NET pages, master pages, CSS files, JavaScript files, images, and other necessary content from the development environment to the production environment; copying up the production environment-specific configuration information, if needed; and applying the changes to the data model since the last deployment. Depending on the number of files and the complexity of your database changes, these steps can take anywhere from a few seconds to several minutes to complete. During this window the web application is in flux and users visiting the site may experience errors or unexpected behavior.

When deploying a website it is best to take the web application "offline" until deployment has completed. Taking the application offline (and bringing it back up once the deployment process has finished) is as easy as uploading a file and then deleting it. Starting with ASP.NET 2.0, the mere presence of a file named `app_offline.htm` in the application s root directory takes the entire website "offline." Any request to an ASP.NET page on that site is automatically responded with the contents of the `app_offline.htm` file. Once that file is removed, the application comes back online.

Taking an application offline during deployment, then, is as simple as uploading an `app_offline.htm` file to the production environment s root directory prior to beginning the deployment process and then deleting it (or renaming it to something else) once deployment is complete. For more information on this technique refer to John Peterson s article, Taking an *ASP.NET Application Offline*.

Summary

The main challenge in deploying a data-driven application centers around deploying the database. Because there are two versions of the database - one in the development environment and one in the production environment - these two databases schemas can become out of sync as new features are added in development. What s more, because the production database as being populated with real data from real users, you cannot overwrite the production database with the modified development database like you can when deploying the files that make up the application (the ASP.NET pages, image files, and so forth). Instead, deploying a database entails implementing the precise set of changes made to the development database on the production database since the last deployment.

This tutorial looked at three techniques for maintaining and applying a log of database changes. The simplest approach is to record the changes in prose. While this tactic makes implementing these changes on the production database a manual process, it does not require knowledge of the SQL commands for creating and altering database objects. A more sophisticated approach, and one that is much more palatable in larger projects or projects with multiple developers, is to record the changes as a series of SQL commands. This greatly hastens rolling out these changes to the target database. The best of both approaches can be achieved by using a database comparison tool, such as Red Gate Software s SQL Compare.

This tutorial concludes our focus on deploying a data-driven application. The next set of tutorials looks at how to respond to errors in the production environment. We'll look at how to display a friendly error page rather instead of the Yellow Screen of Death. And we'll see how to log the error s details and how to alert you when such errors occur.

Happy Programming!

Displaying a Custom Error Page (VB)

2/20/2020 • 13 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

What does the user see when a runtime error occurs in an ASP.NET web application? The answer depends on how the website's <customErrors> configuration. By default, users are shown an unsightly yellow screen proclaiming that a runtime error has occurred. This tutorial shows how to customize these settings to display an aesthetically-pleasing custom error page that matches your site's look and feel.

Introduction

In a perfect world there would be no run-time errors. Programmers would write code with nary a bug and with robust user input validation, and external resources like database servers and email servers would never go offline. Of course, in reality errors are inevitable. The classes in the .NET Framework signal an error by throwing an exception. For example, calling a SqlConnection object's Open method establishes a connection to the database specified by a connection string. However, if the database is down or if the credentials in the connection string are invalid then the Open method throws a `SqlException`. Exceptions can be handled by the use of `Try/Catch/Finally` blocks. If code within a `Try` block throws an exception, control is transferred to the appropriate catch block where the developer can attempt to recover from the error. If there is no matching catch block, or if the code that threw the exception is not in a try block, the exception percolates up the call stack in search of `Try/Catch/Finally` blocks.

If the exception bubbles all the way up to the ASP.NET runtime without being handled, the `HttpApplication` class's `Error` event is raised and the configured *error page* is displayed. By default, ASP.NET displays an error page that is affectionately referred to as the Yellow Screen of Death (YSOD). There are two versions of the YSOD: one shows the exception details, a stack trace, and other information helpful to developers debugging the application (see **Figure 1**); the other simply states that there was a run-time error (see **Figure 2**).

The exception details YSOD is quite helpful for developers debugging the application, but showing a YSOD to end users is tacky and unprofessional. Instead, end users should be taken to an error page that maintains the site's look and feel with more user-friendly prose describing the situation. The good news is that creating such a custom error page is quite easy. This tutorial starts with a look at ASP.NET's different error pages. It then shows how to configure the web application to show users a custom error page in the face of an error.

Examining the Three Types of Error Pages

When an unhandled exception arises in an ASP.NET application one of three types of error pages is displayed:

- The Exception Details Yellow Screen of Death error page,
- The Runtime Error Yellow Screen of Death error page, or
- A custom error page

The error page developers are most familiar with is the Exception Details YSOD. By default, this page is displayed to users who are visiting locally and therefore is the page that you see when an error occurs when testing the site in the development environment. As its name implies, the Exception Details YSOD provides details about the exception - the type, the message, and the stack trace. What's more, if the exception was raised by code in your ASP.NET page's code-behind class and if the application is configured for debugging then the Exception Details YSOD will also show this line of code (and a few lines of code above and below it).

Figure 1 shows the Exception Details YSOD page. Note the URL in the browser's address window: `http://localhost:62275/Genre.aspx?ID=foo`. Recall that the `Genre.aspx` page lists the book reviews in a particular genre. It requires that `GenreId` value (a `uniqueidentifier`) be passed through the querystring; for example, the appropriate URL to view the fiction reviews is `Genre.aspx?ID=7683ab5d-4589-4f03-a139-1c26044d0146`. If a non-`uniqueidentifier` value is passed in through the querystring (such as "foo") an exception is thrown.

> **NOTE**
> To reproduce this error in the demo web application available for download you can either visit `Genre.aspx?ID=foo` directly or click the "Generate a Runtime Error" link in `Default.aspx`.

Note the exception information presented in **Figure 1**. The exception message, "Conversion failed when converting from a character string to uniqueidentifier" is present at the top of the page. The type of the exception, `System.Data.SqlClient.SqlException`, is listed, as well. There's also the stack trace.

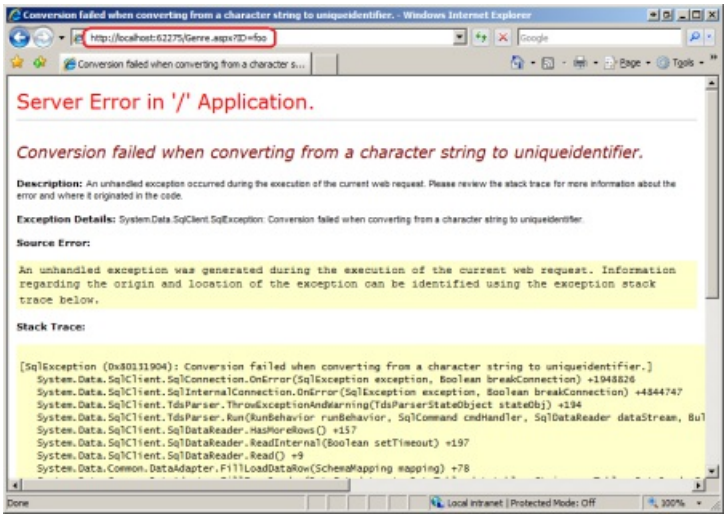


Figure 1: The Exception Details YSOD Includes Information About the Exception
(Click to view full-size image)

The other type of YSOD is the Runtime Error YSOD, and is shown in **Figure 2**. The Runtime Error YSOD informs the visitor that a run-time error has occurred, but it does not include any information about the exception that was thrown. (It does, however, provide instructions on how to make the error details viewable by modifying the `Web.config` file, which is part of what makes such a YSOD look unprofessional.)

By default, the Runtime Error YSOD is shown to users visiting remotely (through http://www.yoursite.com), as evidenced by the URL in the browser's Address bar in **Figure 2**: `http://httpruntime.web703.discountasp.net/Genre.aspx?ID=foo`. The two different YSOD screens exist because developers are interested in knowing the error details, but such information should not be shown on a live site as it may reveal potential security vulnerabilities or other sensitive information to anyone who visits your site.

> **NOTE**
> If you are following along and are using DiscountASP.NET as your web host, you may notice that the Runtime Error YSOD does not display when visiting the live site. This is because DiscountASP.NET has their servers configured to show the Exception Details YSOD by default. The good news is that you can override this default behavior by adding a `<customErrors>` section to your `Web.config` file. The "Configuring Which Error Page is Displayed" section examines the `<customErrors>` section in detail.

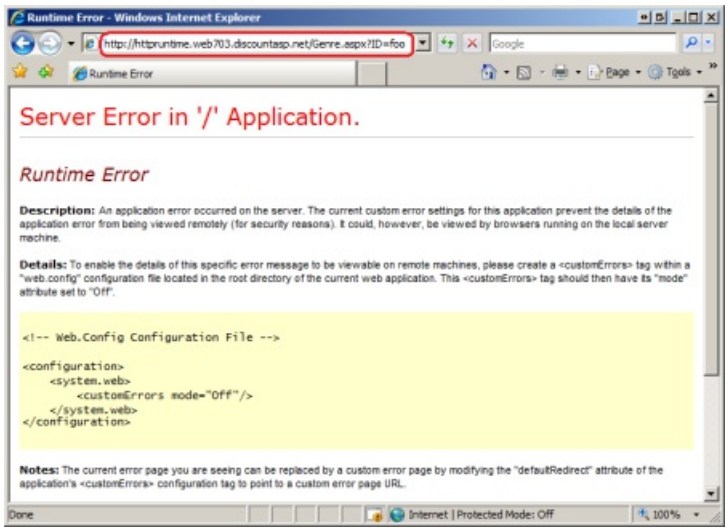


Figure 2: The Runtime Error YSOD Does Not Include Any Error Details
(Click to view full-size image)

The third type of error page is the custom error page, which is a web page that you create. The benefit of a custom error page is that you have complete control over the information that is displayed to the user along with the page's look and feel; the custom error page can use the same master page and styles as your other pages. The "Using a Custom Error Page" section walks through creating a custom error page and configuring it to display in the event of an unhandled exception. **Figure 3** offers a sneak peak of this custom error page. As you can see, the look and feel of the error page is much more professional-looking than either of the Yellow Screens of Death shown in Figures 1 and 2.

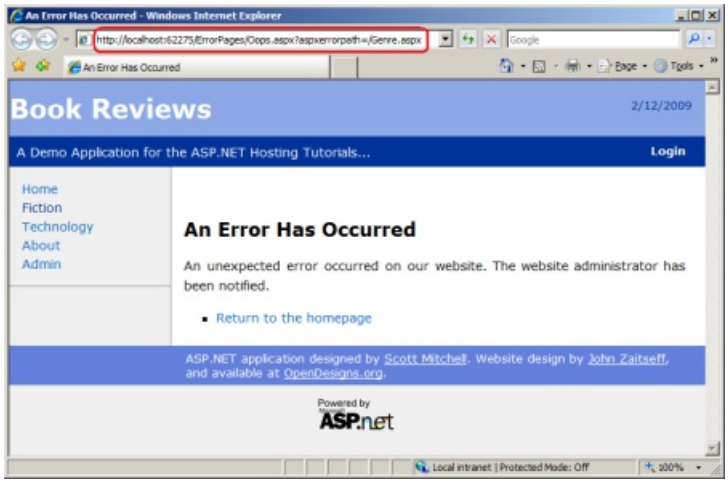


Figure 3: A Custom Error Page Offers a More Tailored Look and Feel
(Click to view full-size image)

Take a moment to inspect the browser's Address bar in **Figure 3**. Note that the Address bar shows the URL of the custom error page (`/ErrorPages/Oops.aspx`). In Figures 1 and 2 the Yellow Screens of Death are shown in the same page that the error originated from (`Genre.aspx`). The custom error page is passed the URL of the page where the error occurred via the `aspxerrorpath` querystring parameter.

Configuring Which Error Page is Displayed

Which of the three possible error pages is displayed is based on two variables:

- The configuration information in the `<customErrors>` section, and
- Whether the user is visiting the site locally or remotely.

The `<customErrors>` section in `Web.config` has two attributes that affect what error page is shown:

`defaultRedirect` and `mode`. The `defaultRedirect` attribute is optional. If provided, it specifies the URL of the custom error page and indicates that the custom error page should be shown instead of the Runtime Error YSOD. The `mode` attribute is required and accepts one of three values: `On`, `Off`, or `RemoteOnly`. These values have the following behavior:

- `On` - indicates that the custom error page or the Runtime Error YSOD is shown to all visitors, regardless of whether they are local or remote.
- `Off` - specifies that the Exception Details YSOD is displayed to all visitors, regardless of whether they are local or remote.
- `RemoteOnly` - indicates that the custom error page or the Runtime Error YSOD is shown to remote visitors, while the Exception Details YSOD is shown to local visitors.

Unless you specify otherwise, ASP.NET acts as if you had set the mode attribute to `RemoteOnly` and had not specified a `defaultRedirect` value. In other words, the default behavior is that the Exception Details YSOD is displayed to local visitors while the Runtime Error YSOD is shown to remote visitors. You can override this default behavior by adding a `<customErrors>` section to your web application's `Web.config file.`

Using a Custom Error Page

Every web application should have a custom error page. It provides a more professional-looking alternative to the Runtime Error YSOD, it is easy to create, and configuring the application to use the custom error page takes only a few moments. The first step is creating the custom error page. I've added a new folder to the Book Reviews application named `ErrorPages` and added to that a new ASP.NET page named `Oops.aspx`. Have the page use the same master page as the rest of the pages on your site so that it automatically inherits the same look and feel.

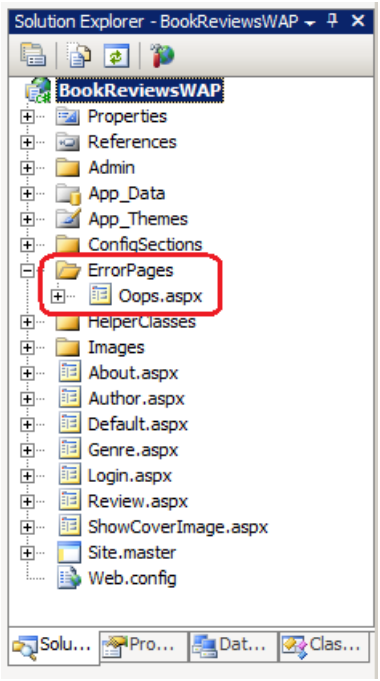


Figure 4: Create a Custom Error Page

Next, spend a few minutes creating the content for the error page. I've created a rather simple custom error page with a message indicating that there was an unexpected error and a link back to the site's homepage.

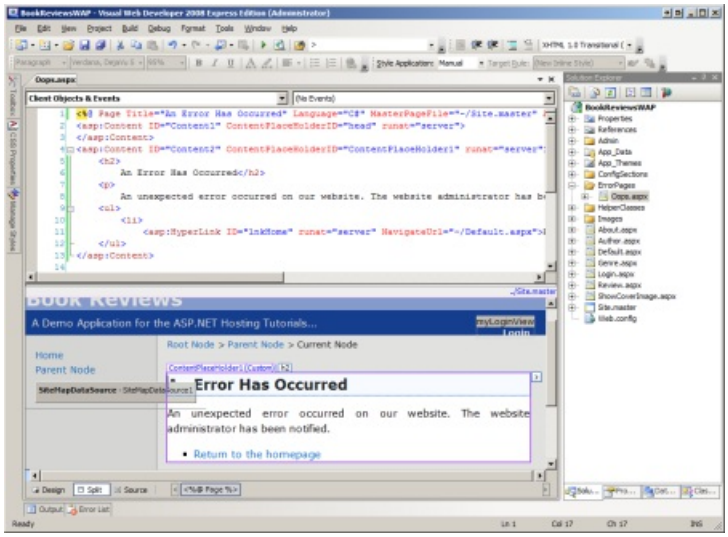


Figure 5: Design Your Custom Error Page
(Click to view full-size image)

With the error page completed, configure the web application to use the custom error page in lieu of the Runtime Error YSOD. This is accomplished by specifying the URL of the error page in the `<customErrors>` section's `defaultRedirect` attribute. Add the following markup to your application's `Web.config` file:

```
<configuration>
    ...

    <system.web>
        <customErrors mode="RemoteOnly"
                      defaultRedirect="~/ErrorPages/Oops.aspx" />

        ...
    </system.web>
</configuration>
```

The above markup configures the application to show the Exception Details YSOD to users visiting locally, while using the custom error page Oops.aspx for those users visiting remotely. To see this in action, deploy your website to the production environment and then visit the Genre.aspx page on the live site with an invalid querystring value. You should see the custom error page (refer back to **Figure 3**).

To verify that the custom error page is only shown to remote users, visit the `Genre.aspx` page with an invalid querystring from the development environment. You should still see the Exception Details YSOD (refer back to **Figure 1**). The `RemoteOnly` setting ensures that users visiting the site on the production environment see the custom error page while developers working locally continue to see the details of the exception.

Notifying Developers and Logging Error Details

Errors that occur in the development environment were caused by the developer sitting at her computer. She is shown the exception's information in the Exception Details YSOD, and she knows what steps she was performing when the error occurred. But when an error occurs on production, the developer has no knowledge that an error occurred unless the end user visiting the site takes the time to report the error. And even if the user goes out of his way to alert the development team that an error occurred, without knowing the exception type, message, and stack trace it can be difficult to diagnose the cause of the error, let alone fix it.

For these reasons it is paramount that any error in the production environment is logged to some persistent store (such as a database) and that the developers are alerted of this error. The custom error page may seem like a good place to do this logging and notification. Unfortunately, the custom error page does not have access to the error details and therefore cannot be used to log this information. The good news is that there are a number of

ways to intercept the error details and to log them, and the next three tutorials explore this topic in more detail.

Using Different Custom Error Pages for Different HTTP Error Statuses

When an exception is thrown by an ASP.NET page and is not handled, the exception percolates up to the ASP.NET runtime, which displays the configured error page. If a request comes into the ASP.NET engine but cannot be processed for some reason - perhaps the requested file is not found or Read permissions have been disabled for the file - then the ASP.NET engine raises an `HttpException`. This exception, like exceptions raised from ASP.NET pages, bubbles up to the runtime, causing the appropriate error page to be displayed.

What this means for the web application in production is that if a user requests a page that is not found then they will see the custom error page. **Figure 6** shows such an example. Because the request is for a non-existent page (`NoSuchPage.aspx`), an `HttpException` is thrown and the custom error page is displayed (note the reference to `NoSuchPage.aspx` in the `aspxerrorpath` querystring parameter).

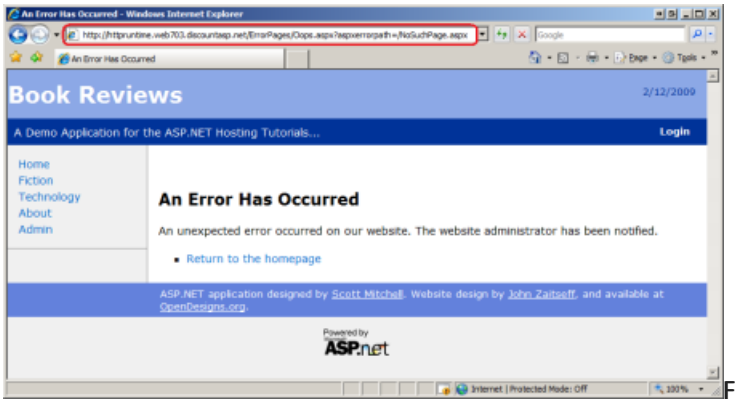


Figure 6: The ASP.NET Runtime Displays the Configured Error Page In Response to an Invalid Request
(Click to view full-size image)

By default, all types of errors cause the same custom error page to be displayed. However, you can specify a different custom error page for a specific HTTP status code using `<error>` children elements within the `<customErrors>` section. For example, to have a different error page displayed in the event of a page not found error, which has an HTTP status code of 404, update the `<customErrors>` section to include the following markup:

```
<customErrors mode="RemoteOnly" defaultRedirect="~/ErrorPages/Oops.aspx">
    <error statusCode="404" redirect="~/ErrorPages/404.aspx" />
</customErrors>
```

With this change in place, whenever a user visiting remotely requests an ASP.NET resource that does not exist, they will be redirected to the `404.aspx` custom error page instead of `Oops.aspx`. As **Figure 7** illustrates, the `404.aspx` page can include a more specific message than the general custom error page.

> **NOTE**
>
> Check out 404 Error Pages, One More Time for guidance on creating effective 404 error pages.

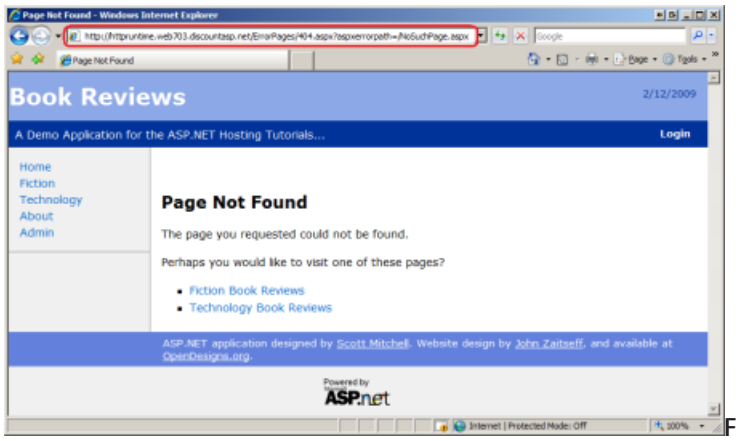


Figure 7: The Custom 404 Error Page Displays a More Targeted Message Than `Oops.aspx`
(Click to view full-size image)

Because you know that the `404.aspx` page is only reached when the user makes a request for a page that was not found, you can enhance this custom error page to include functionality to help the user address this specific type of error. For example, you could build a database table that maps known bad URLs to good URLs, and then have the `404.aspx` custom error page run a query against that table and suggest pages the user may be trying to reach.

> **NOTE**
> The custom error page is only displayed when a request is made to a resource handled by the ASP.NET engine. As we discussed in the Core Differences Between IIS and the ASP.NET Development Server tutorial, the web server may handle certain requests itself. By default, the IIS web server processes requests for static content like images and HTML files without invoking the ASP.NET engine. Consequently, if the user requests a non-existent image file they will get back IIS's default 404 error message rather than ASP.NET's configured error page.

Summary

When an unhandled exception occurs in an ASP.NET application, the user is shown one of three error pages: the Exception Details Yellow Screen of Death; the Runtime Error Yellow Screen of Death; or a custom error page. Which error page is displayed depends on the application's `<customErrors>` configuration and whether the user is visiting locally or remotely. The default behavior is to show the Exception Details YSOD to local visitors and the Runtime Error YSOD to remote visitors.

While the Runtime Error YSOD hides potentially sensitive error information from the user visiting the site, it breaks from your site's look and feel and makes your application look buggy. A better approach is to use a custom error page, which entails creating and designing the custom error page and specifying its URL in the `<customErrors>` section's `defaultRedirect` attribute. You can even have multiple custom error pages for different HTTP error statuses.

The custom error page is the first step in a comprehensive error handling strategy for a website in production. Alerting the developer of the error and logging its details are also important steps. The next three tutorials explore techniques for error notification and logging.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Error Pages, One More Time
- Design Guidelines for Exceptions

- User-Friendly Error Pages
- Handling and Throwing Exceptions
- Properly Using Custom Error Pages in ASP.NET

PREVIOUS NEXT

Processing Unhandled Exceptions (VB)

2/21/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

View or download sample code (how to download)

When a runtime error occurs on a web application in production it is important to notify a developer and to log the error so that it may be diagnosed at a later point in time. This tutorial provides an overview of how ASP.NET processes runtime errors and looks at one way to have custom code execute whenever an unhandled exception bubbles up to the ASP.NET runtime.

Introduction

When an unhandled exception occurs in an ASP.NET application, it bubbles up to the ASP.NET runtime, which raises the `Error` event and displays the appropriate error page. There are three different types of error pages: the Runtime Error Yellow Screen of Death (YSOD); the Exception Details YSOD; and custom error pages. In the preceding tutorial we configured the application to use a custom error page for remote users and the Exception Details YSOD for users visiting locally.

Using a human-friendly custom error page that matches the look and feel of the site is preferred to the default Runtime Error YSOD, but displaying a custom error page is only one part of a comprehensive error handling solution. When an error occurs in an application in production, it is important that the developers are notified of the error so that they can unearth the cause of the exception and address it. Furthermore, the error's details should be logged so that the error can be examined and diagnosed at a later point in time.

This tutorial shows how to access the details of an unhandled exception so that they can be logged and a developer notified. The two tutorials following this one explore error logging libraries that, after a bit of configuration, will automatically notify developers of runtime errors and log their details.

> **NOTE**
>
> The information examined in this tutorial is most useful if you need to process unhandled exceptions in some unique or customized manner. In cases where you only need to log the exception and notify a developer, using an error logging library is the way to go. The next two tutorials provide an overview of two such libraries.

Executing Code When The `Error` Event Is Raised

Events provide an object a mechanism for signaling that something interesting has occurred, and for another object to execute code in response. As an ASP.NET developer you are accustomed to thinking in terms of events. If you want to run some code when the visitor clicks a particular Button, you create an event handler for that Button's `Click` event and put your code there. Given that the ASP.NET runtime raises its `Error` event whenever an unhandled exception occurs, it follows that the code for logging the error's details would go in an event handler. But how do you create an event handler for the `Error` event?

The `Error` event is one of many events in the `HttpApplication` class that are raised at certain stages in the HTTP pipeline during the lifetime of a request. For example, the `HttpApplication` class's `BeginRequest` event is raised at the start of every request; its `AuthenticateRequest` event is raised when a security module has identified the requestor. These `HttpApplication` events give the page developer a means to execute custom logic at the various points in the lifetime of a request.

Event handlers for the `HttpApplication` events can be placed in a special file named `Global.asax`. To create this file in your website, add a new item to the root of your website using the Global Application Class template with the name `Global.asax`.

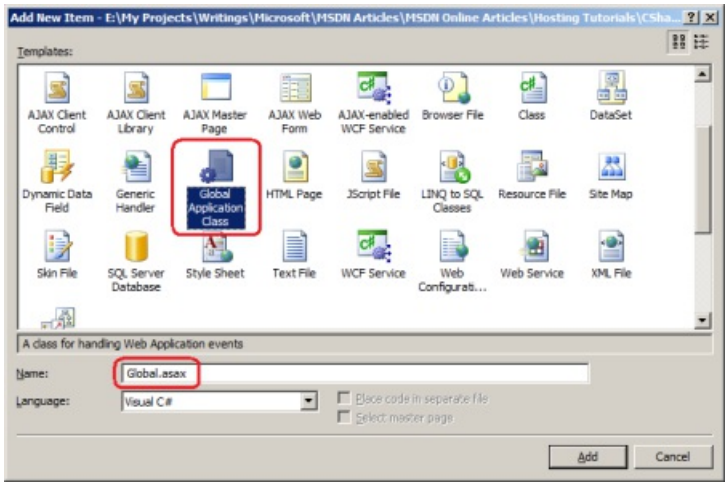


Figure 1: Add `Global.asax` To Your Web Application
(Click to view full-size image)

The contents and structure of the `Global.asax` file created by Visual Studio differ slightly based on whether you are using a Web Application Project (WAP) or Web Site Project (WSP). With a WAP, the `Global.asax` is implemented as two separate files - `Global.asax` and `Global.asax.vb`. The `Global.asax` file contains nothing but an `@Application` directive that references the `.vb` file; the event handlers of interest are defined in the `Global.asax.vb` file. For WSPs, only a single file is created, `Global.asax`, and the event handlers are defined in a `<script runat="server">` block.

The `Global.asax` file created in a WAP by Visual Studio's Global Application Class template includes event handlers named `Application_BeginRequest`, `Application_AuthenticateRequest`, and `Application_Error`, which are event handlers for the `HttpApplication` events `BeginRequest`, `AuthenticateRequest`, and `Error`, respectively. There are also event handlers named `Application_Start`, `Session_Start`, `Application_End`, and `Session_End`, which are event handlers that fire when the web application starts, when a new session starts, when the application ends, and when a session ends, respectively. The `Global.asax` file created in a WSP by Visual Studio contains just the `Application_Error`, `Application_Start`, `Session_Start`, `Application_End`, and `Session_End` event handlers.

> **NOTE**
> When deploying the ASP.NET application you need to copy the `Global.asax` file to the production environment. The `Global.asax.vb` file, which is created in the WAP, does not need to be copied to production because this code is compiled into the project's assembly.

The event handlers created by Visual Studio's Global Application Class template are not exhaustive. You can add an event handler for any `HttpApplication` event by naming the event handler `Application_EventName`. For example, you could add the following code to the `Global.asax` file to create an event handler for the `AuthorizeRequest` event:

```
Sub Application_AuthorizeRequest(ByVal sender As Object, ByVal e As EventArgs)
    ' Event handler code
End Sub
```

Likewise, you can remove any event handlers created by the Global Application Class template that are not needed. For this tutorial we only require an event handler for the `Error` event; feel free to remove the other event

handlers from the `Global.asax` file.

> **NOTE**
> *HTTP Modules* offer another way to define event handlers for `HttpApplication` events. HTTP Modules are created as a class file that can be placed directly within the web application project or separated out into a separate class library. Because they can be separated out into a class library, HTTP Modules offer a more flexible and reusable model for creating `HttpApplication` event handlers. Whereas the `Global.asax` file is specific to the web application where it resides, HTTP Modules can be compiled into assemblies, at which point adding the HTTP Module to a website is as simple as dropping the assembly in the `Bin` folder and registering the Module in `Web.config`. This tutorial does not look at creating and using HTTP Modules, but the two error logging libraries used in the following two tutorials are implemented as HTTP Modules. For more background on the benefits of HTTP Modules refer to Using HTTP Modules and Handlers to Create Pluggable ASP.NET Components.

Retrieving Information About the Unhandled Exception

At this point we have a Global.asax file with an `Application_Error` event handler. When this event handler executes we need to notify a developer of the error and log its details. To accomplish these tasks we first need to determine the details of the exception that was raised. Use the Server object's `GetLastError` method to retrieve details of the unhandled exception that caused the `Error` event to fire.

```
Sub Application_Error(ByVal sender As Object, ByVal e As EventArgs)
    ' Get the error details
    Dim lastErrorWrapper As HttpException = _
        CType(Server.GetLastError(), HttpException)
End Sub
```

The `GetLastError` method returns an object of type `Exception`, which is the base type for all exceptions in the .NET Framework. However, in the code above I am casting the Exception object returned by `GetLastError` into an `HttpException` object. If the `Error` event is being fired because an exception was thrown during the processing of an ASP.NET resource then the exception that was thrown is wrapped within an `HttpException`. To get the actual exception that precipitated the Error event use the `InnerException` property. If the `Error` event was raised because of an HTTP-based exception, such as a request for a non-existent page, an `HttpException` is thrown, but it does not have an inner exception.

The following code uses the `GetLastErrormessage` to retrieve information about the exception that triggered the `Error` event, storing the `HttpException` in a variable named `lastErrorWrapper`. It then stores the type, message, and stack trace of the originating exception in three string variables, checking to see if the `lastErrorWrapper` is the actual exception that triggered the `Error` event (in the case of HTTP-based exceptions) or if it's merely a wrapper for an exception that was thrown while processing the request.

```
Sub Application_Error(ByVal sender As Object, ByVal e As EventArgs)
    ' Get the error details
    Dim lastErrorWrapper As HttpException = _
        CType(Server.GetLastError(), HttpException)

    Dim lastError As Exception = lastErrorWrapper
    If lastErrorWrapper.InnerException IsNot Nothing Then
        lastError = lastErrorWrapper.InnerException
    End If

    Dim lastErrorTypeName As String = lastError.GetType().ToString()
    Dim lastErrorMessage As String = lastError.Message
    Dim lastErrorStackTrace As String = lastError.StackTrace
End Sub
```

At this point you have all the information you need to write code that will log the exception's details to a database table. You could create a database table with columns for each of the error details of interest - the type, the message, the stack trace, and so on - along with other useful pieces of information, such as the URL of the requested page and the name of the currently logged on user. In the `Application_Error` event handler you would then connect to the database and insert a record into the table. Likewise, you could add code to alert a developer of the error via email.

The error logging libraries examined in the next two tutorials provide such functionality out of the box, so there's no need to build this error logging and notification yourself. However, to illustrate that the `Error` event is being raised and that the `Application_Error` event handler can be used to log error details and notify a developer, let's add code that notifies a developer when an error occurs.

Notifying a Developer When an Unhandled Exception Occurs

When an unhandled exception occurs in the production environment it is important to alert the development team so that they can assess the error and determine what actions need to be taken. For example, if there is an error in connecting to the database then you'll need to double check your connection string and, perhaps, open a support ticket with your web hosting company. If the exception occurred because of a programming error, additional code or validation logic may need to be added to prevent such errors in the future.

The .NET Framework classes in the `System.Net.Mail` namespace make it easy to send an email. The `MailMessage` class represents an email message and has properties like `To`, `From`, `Subject`, `Body`, and `Attachments`. The `SmtpClass` is used to send a `MailMessage` object using a specified SMTP server; the SMTP server settings can be specified programmatically or declaratively in the `<system.net>` element in the `Web.config file`. For more information on sending email messages in an ASP.NET application check out my article, Sending Email in ASP.NET, and the System.Net.Mail FAQ.

> **NOTE**
> The `<system.net>` element contains the SMTP server settings used by the `SmtpClient` class when sending an email. Your web hosting company likely has an SMTP server that you can use to send email from your application. Consult your web host's support section for information on the SMTP server settings you should use in your web application.

Add the following code to the `Application_Error` event handler to send a developer an email when an error occurs:

```
Sub Application_Error(ByVal sender As Object, ByVal e As EventArgs)
    ' Get the error details
    Dim lastErrorWrapper As HttpException = _
        CType(Server.GetLastError(), HttpException)

    Dim lastError As Exception = lastErrorWrapper
    If lastErrorWrapper.InnerException IsNot Nothing Then
        lastError = lastErrorWrapper.InnerException
    End If

    Dim lastErrorTypeName As String = lastError.GetType().ToString()
    Dim lastErrorMessage As String = lastError.Message
    Dim lastErrorStackTrace As String = lastError.StackTrace

    Const ToAddress As String = "support@example.com"
    Const FromAddress As String = "support@example.com"
    Const Subject As String = "An Error Has Occurred!"

    ' Create the MailMessage object
    Dim mm As New MailMessage(FromAddress, ToAddress)
    mm.Subject = Subject
    mm.IsBodyHtml = True
```

```
      mm.Priority = MailPriority.High
    mm.Body = string.Format( _
  "<html>" & vbCrLf & _
  "  <body>" & vbCrLf & _
  "  <h1>An Error Has Occurred!</h1>" & vbCrLf & _
  "  <table cellpadding=""5"" cellspacing=""0"" border=""1""">" & vbCrLf & _
  "  <tr>" & vbCrLf & _
  "  <tdtext-align: right;font-weight: bold"">URL:</td>" & vbCrLf & _
  "  <td>{0}</td>" & vbCrLf & _
  "  </tr>" & vbCrLf & _
  "  <tr>" & vbCrLf & _
  "  <tdtext-align: right;font-weight: bold"">User:</td>" & vbCrLf & _
  "  <td>{1}</td>" & vbCrLf & _
  "  </tr>" & vbCrLf & _
  "  <tr>" & vbCrLf & _
  "  <tdtext-align: right;font-weight: bold"">Exception Type:</td>" & vbCrLf & _
  "  <td>{2}</td>" & vbCrLf & _
  "  </tr>" & vbCrLf & _
  "  <tr>" & vbCrLf & _
  "  <tdtext-align: right;font-weight: bold"">Message:</td>" & vbCrLf & _
  "  <td>{3}</td>" & vbCrLf & _
  "  </tr>" & vbCrLf & _
  "  <tr>" & vbCrLf & _
  "  <tdtext-align: right;font-weight: bold"">Stack Trace:</td>" & vbCrLf & _
  "  <td>{4}</td>" & vbCrLf & _
  "  </tr> " & vbCrLf & _
  "  </table>" & vbCrLf & _
  "  </body>" & vbCrLf & _
  "</html>", _
    Request.RawUrl, _
    User.Identity.Name, _
    lastErrorTypeName, _
    lastErrorMessage, _
    lastErrorStackTrace.Replace(Environment.NewLine, "<br />"))

      'Attach the Yellow Screen of Death for this error
      Dim YSODmarkup As String = lastErrorWrapper.GetHtmlErrorMessage()
      If Not String.IsNullOrEmpty(YSODmarkup) Then
          Dim YSOD As Attachment = _
              Attachment.CreateAttachmentFromString(YSODmarkup, "YSOD.htm")
          mm.Attachments.Add(YSOD)
      End If

      ' Send the email
      Dim smtp As New SmtpClient()
      smtp.Send(mm)
  End Sub
```

While the above code is quite lengthy, the bulk of it creates the HTML that appears in the email sent to the developer. The code starts by referencing the `HttpException` returned by the `GetLastError` method (`lastErrorWrapper`). The actual exception that was raised by the request is retrieved via `lastErrorWrapper.InnerException` and is assigned to the variable `lastError`. The type, message, and stack trace information is retrieved from `lastError` and stored in three string variables.

Next, a `MailMessage` object named `mm` is created. The email body is HTML formatted and displays the URL of the requested page, the name of the currently logged on user, and information about the exception (the type, message, and stack trace). One of the cool things about the `HttpException` class is that you can generate the HTML used to create the Exception Details Yellow Screen of Death (YSOD) by calling the GetHtmlErrorMessage method. This method is used here to retrieve the Exception Details YSOD markup and add it to the email as an attachment. One word of caution: if the exception that triggered the `Error` event was an HTTP-based exception (such as a request for a non-existent page) then the `GetHtmlErrorMessage` method will return `null`.

The final step is to send the `MailMessage`. This is done by creating a new `SmtpClient` method and calling its `Send`

method.

> **NOTE**
> Before using this code in your web application you'll want to change the values in the `ToAddress` and `FromAddress` constants from support@example.com to whatever email address the error notification email should be sent to and originate from. You'll also need to specify SMTP server settings in the `<system.net>` section in `Web.config`. Consult your web host provider to determine the SMTP server settings to use.

With this code in place anytime there's an error the developer is sent an email message that summarizes the error and includes the YSOD. In the preceding tutorial we demonstrated a runtime error by visiting Genre.aspx and passing in an invalid `ID` value through the querystring, like `Genre.aspx?ID=foo`. Visiting the page with the `Global.asax` file in place produces the same user experience as in the preceding tutorial - in the development environment you'll continue to see the Exception Details Yellow Screen of Death, while in the production environment you'll see the custom error page. In addition to this existing behavior, the developer is sent an email.

Figure 2 shows the email received when visiting `Genre.aspx?ID=foo`. The email body summarizes the exception information, while the `YSOD.htm` attachment displays the content that is shown in the Exception Details YSOD (see **Figure 3**).

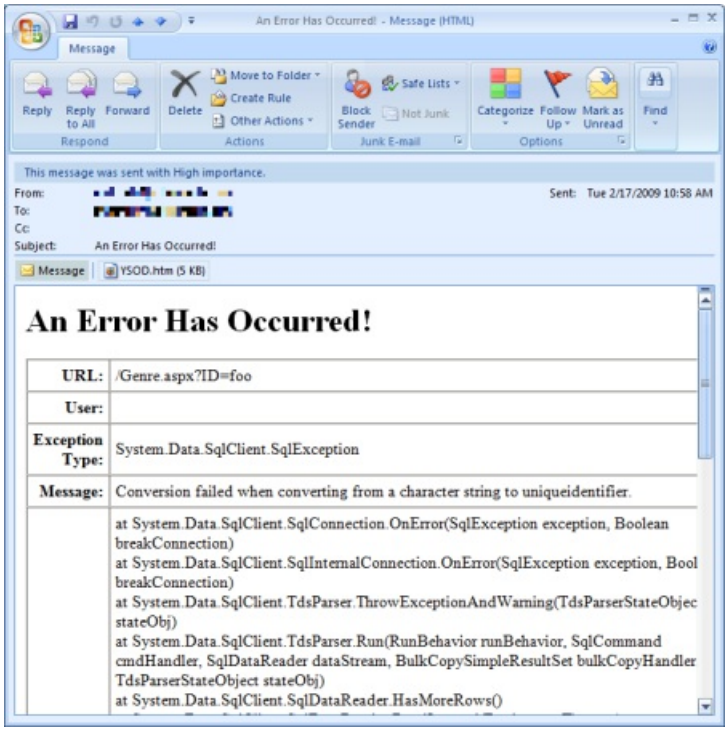


Figure 2: The Developer Is Sent An Email Notification Whenever There's An Unhandled Exception
(Click to view full-size image)

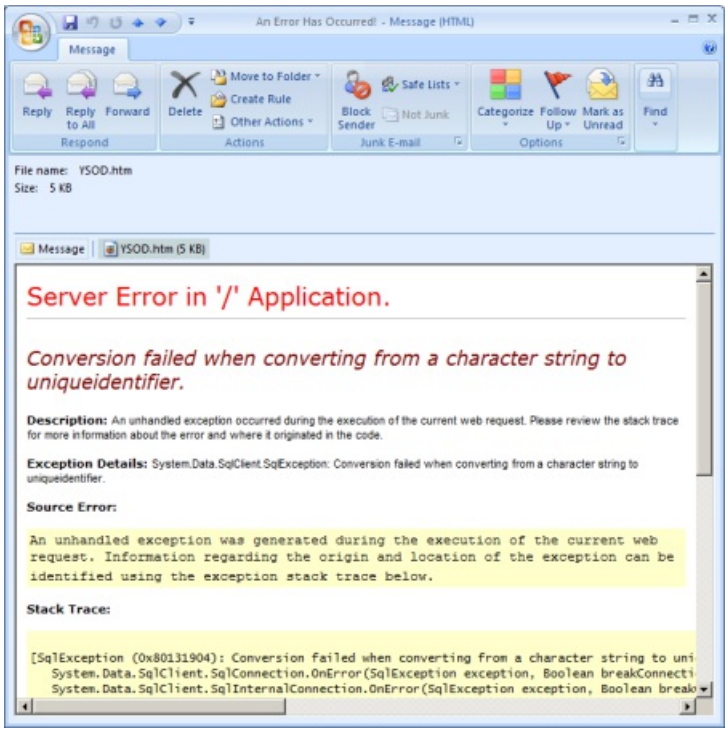


Figure 3: The Email Notification Includes the Exception Details YSOD As An Attachment
(Click to view full-size image)

What About Using the Custom Error Page?

This tutorial showed how to use `Global.asax` and the `Application_Error` event handler to execute code when an unhandled exception occurs. Specifically, we used this event handler to notify a developer of an error; we could extend it to also log the error details in a database. The presence of the `Application_Error` event handler does not affect the end user's experience. They still see the configured error page, be it the Error Details YSOD, the Runtime Error YSOD, or the custom error page.

It's natural to wonder whether the `Global.asax` file and `Application_Error` event is necessary when using a custom error page. When an error occurs the user is shown the custom error page so why can't we put the code to notify the developer and log the error details into the code-behind class of the custom error page? While you can certainly add code to the custom error page's code-behind class you do not have access to the details of the exception that triggered the `Error` event when using the technique we explored in the preceding tutorial. Calling the `GetLastError` method from the custom error page returns `Nothing`.

The reason for this behavior is because the custom error page is reached via a redirect. When an unhandled exception reaches the ASP.NET runtime the ASP.NET engine raises its `Error` event (which executes the `Application_Error` event handler) and then *redirects* the user to the custom error page by issuing a `Response.Redirect(customErrorPageUrl)`. The `Response.Redirect` method sends a response to the client with an HTTP 302 status code, instructing the browser to request a new URL, namely the custom error page. The browser then automatically requests this new page. You can tell that the custom error page was requested separately from the page where the error originated because the browser's Address bar changes to the custom error page URL (see **Figure 4**).

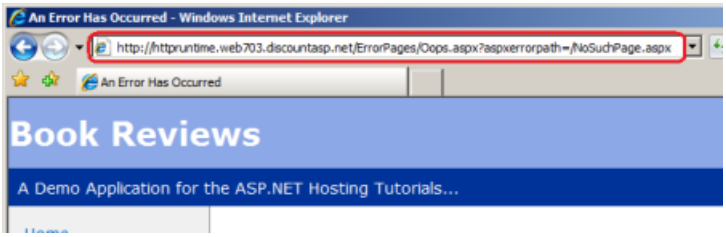


Figure 4: When an Error Occurs the Browser Gets Redirected to the Custom Error Page URL

(Click to view full-size image)

The net effect is that the request where the unhandled exception occurred ends when the server responds with the HTTP 302 redirect. The subsequent request to the custom error page is a brand new request; by this point the ASP.NET engine has discarded the error information and, moreover, has no way to associate the unhandled exception in the previous request with the new request for the custom error page. This is why `GetLastError` returns `null` when called from the custom error page.

However, it is possible to have the custom error page executed during the same request that caused the error. The `Server.Transfer(url)` method transfers execution to the specified URL and processes it within the same request. You could move the code in the `Application_Error` event handler to the custom error page's code-behind class, replacing it in `Global.asax` with the following code:

```
Sub Application_Error(ByVal sender As Object, ByVal e As EventArgs)
    ' Get the error details
    Dim lastErrorWrapper As HttpException = _
        CType(Server.GetLastError(), HttpException)

    If lastErrorWrapper.GetHttpCode() = 404 Then
        Server.Transfer("~/ErrorPages/404.aspx")
    Else
        Server.Transfer("~/ErrorPages/Oops.aspx")
    End If
End Sub
```

Now when an unhandled exception occurs the `Application_Error` event handler transfers control to the appropriate custom error page based on the HTTP status code. Because control was transferred, the custom error page has access to the unhandled exception information via `Server.GetLastError` and can notify a developer of the error and log its details. The `Server.Transfer` call stops the ASP.NET engine from redirecting the user to the custom error page. Instead, the custom error page's content is returned as the response to the page that generated the error.

Summary

When an unhandled exception occurs in an ASP.NET web application the ASP.NET runtime raises the `Error` event and displays the configured error page. We can notify the developer of the error, log its details, or process it in some other fashion, by creating an event handler for the Error event. There are two ways to create an event handler for `HttpApplication` events like `Error`: in the `Global.asax` file or from an HTTP Module. This tutorial showed how to create an `Error` event handler in the `Global.asax` file that notifies developers of an error by means of an email message.

Creating an `Error` event handler is useful if you need to process unhandled exceptions in some unique or customized manner. However, creating your own `Error` event handler to log the exception or to notify a developer is not the most efficient use of your time as there already exist free and easy to use error logging libraries that can be setup in a matter of minutes. The next two tutorials examine two such libraries.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET HTTP Modules and HTTP Handlers Overview
- Gracefully Responding to Unhandled Exceptions - Processing Unhandled Exceptions
- `HttpApplication` Class and the ASP.NET Application Object
- HTTP Handlers and HTTP Modules in ASP.NET
- Sending Email in ASP.NET

- Understanding the `Global.asax` File
- Using HTTP Modules and Handlers to Create Pluggable ASP.NET Components
- Working with the ASP.NET `Global.asax` File
- Working with `HttpApplication` Instances

PREVIOUS NEXT

Logging Error Details with ASP.NET Health Monitoring (VB)

2/20/2020 • 11 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> Microsoft's health monitoring system provides an easy and customizable way to log various web events, including unhandled exceptions. This tutorial walks through setting up the health monitoring system to log unhandled exceptions to a database and to notify developers via an email message.

Introduction

Logging is a useful tool for monitoring the health of a deployed application and for diagnosing any problems that may arise. It is especially important to log errors that occur in a deployed application so that they can be remedied. The `Error` event is raised whenever an unhandled exception occurs in an ASP.NET application; the preceding tutorial showed how to notify a developer of an error and log its details by creating an event handler for the `Error` event. However, creating an `Error` event handler to log the error's details and notify a developer is unnecessary, as this task can be performed by ASP.NET's *health monitoring system*.

The health monitoring system was introduced in ASP.NET 2.0 and is designed to monitor the health of a deployed ASP.NET application by logging events that occur during the application or request's lifetime. The events logged by the health monitoring system are referred to as *health monitoring events* or *Web events*, and include:

- Application lifetime events, such as when an application starts or stops
- Security events, including failed login attempts and failed URL authorization requests
- Application errors, including unhandled exceptions, view state parsing exceptions, request validation exceptions, and compilation errors, among other types of errors.

When a health monitoring event is raised it can be logged to any number of specified *log sources*. The health monitoring system ships with log sources that log Web events to a Microsoft SQL Server database, to the Windows Event Log, or via an email message, among others. You can also create your own log sources.

The events the health monitoring system logs, along with the log sources used, are defined in `Web.config`. With a few lines of configuration markup you can use health monitoring to log all unhandled exceptions to a database and to notify you of the exception via email.

Exploring the Health Monitoring System's Configuration

The health monitoring system's behavior is defined by its configuration information, which is located in the `<healthMonitoring>` element in `Web.config`. This configuration section defines, among other things, the following three important pieces of information:

1. The health monitoring events that, when raised, should be logged,
2. The log sources, and
3. How each health monitoring event defined in (1) is mapped to the log sources defined in (2).

This information is specified through three children configuration elements: `<eventMappings>`, `<providers>`, and `<rules>`, respectively.

The default health monitoring system configuration information can be found in the `Web.config` file in `%WINDIR%\Microsoft.NET\Framework\version\CONFIG` folder. This default configuration information, with some markup removed for brevity, is shown below:

```
<configuration>
  <system.web>
  <healthMonitoring>
  <eventMappings>
  <add name="All Errors"
type="System.Web.Management.WebBaseErrorEvent,System.Web,Version=2.0.0.0,Culture=neutral,PublicKeyToken=b03f5
f7f11d50a3a" startEventCode="0" endEventCode="2147483647"/>

  <add name="Failure Audits"
type="System.Web.Management.WebFailureAuditEvent,System.Web,Version=2.0.0.0,Culture=neutral,PublicKeyToken=b0
3f5f7f11d50a3a" startEventCode="0" endEventCode="2147483647"/>
  </eventMappings>

  <providers>
  <add name="EventLogProvider"
type="System.Web.Management.EventLogWebEventProvider,System.Web,Version=2.0.0.0,Culture=neutral,PublicKeyToke
n=b03f5f7f11d50a3a"/>

  <add connectionStringName="LocalSqlServer" maxEventDetailsLength="1073741823" buffer="false"
bufferMode="Notification" name="SqlWebEventProvider"
type="System.Web.Management.SqlWebEventProvider,System.Web,Version=2.0.0.0,Culture=neutral,PublicKeyToken=b03
f5f7f11d50a3a"/>
  </providers>

  <rules>
  <add name="All Errors Default" eventName="All Errors" provider="EventLogProvider" profile="Default"
minInstances="1" maxLimit="Infinite" minInterval="00:01:00" custom=""/>

  <add name="Failure Audits Default" eventName="Failure Audits" provider="EventLogProvider" profile="Default"
minInstances="1" maxLimit="Infinite" minInterval="00:01:00" custom=""/>
  </rules>
  </healthMonitoring>
  </system.web>
</configuration>
```

The health monitoring events of interest are defined in the `<eventMappings>` element, which gives a human-friendly name to a class of health monitoring events. In the markup above, the `<eventMappings>` element assigns the human-friendly name "All Errors" to the health monitoring events of type `WebBaseErrorEvent` and the name "Failure Audits" to health monitoring events of type `WebFailureAuditEvent`.

The `<providers>` element defines the log sources, giving them a human-friendly name and specifying any log source-specific configuration information. The first `<add>` element defines the "EventLogProvider" provider, which logs the specified health monitoring events using the `EventLogWebEventProvider` class. The `EventLogWebEventProvider` class logs the event to the Windows Event Log. The second `<add>` element defines the "SqlWebEventProvider" provider, which logs events to a Microsoft SQL Server database via the `SqlWebEventProvider` class. The "SqlWebEventProvider" configuration specifies the database's connection string (`connectionStringName`) among other configuration options.

The `<rules>` element maps the events specified in the `<eventMappings>` element to log sources in the `<providers>` element. By default, ASP.NET web applications log all unhandled exceptions and audit failures to the Windows Event Log.

Logging Events to a Database

The health monitoring system's default configuration can be customized on a web application-by-web application basis by adding a `<healthMonitoring>` section to the application's `Web.config` file. You can include additional

elements in the `<eventMappings>`, `<providers>`, and `<rules>` sections by using the `<add>` element. To remove a setting from the default configuration use the `<remove>` element, or use `<clear />` to remove all default values from one of these sections. Let's configure the Book Reviews web application to log all unhandled exceptions to a Microsoft SQL Server database using the `SqlWebEventProvider` class.

The `SqlWebEventProvider` class is part of the health monitoring system and logs a health monitoring event to a specified SQL Server database. The `SqlWebEventProvider` class expects that the specified database includes a stored procedure named `aspnet_WebEvent_LogEvent`. This stored procedure is passed the details of the event and is tasked with storing the event details. The good news is that you do not need to create this stored procedure nor the table to store the event details. You can add these objects to your database using the `aspnet_regsql.exe` tool.

> **NOTE**
> The `aspnet_regsql.exe` tool was discussed back in the *Configuring a Website That Uses Application Services* tutorial when we added support for ASP.NET's application services. Consequently, the Book Reviews website's database already contains the `aspnet_WebEvent_LogEvent` stored procedure, which stores the event information into a table named `aspnet_WebEvent_Events`.

Once you have the necessary stored procedure and table added to your database, all that remains is to instruct health monitoring to log all unhandled exceptions to the database. Accomplish this by adding the following markup to your website's `Web.config` file:

```
<configuration>
  ...
  <system.web>
  ...
  <healthMonitoring enabled="true">
  <eventMappings>
  <clear />

  <add name="All Errors" type="System.Web.Management.WebBaseErrorEvent"
  startEventCode="0" endEventCode="2147483647" />
  </eventMappings>

  <providers>
  <clear />

  <add connectionStringName="ReviewsConnectionString" maxEventDetailsLength="1073741823"
  buffer="false" name="SqlWebEventProvider"
  type="System.Web.Management.SqlWebEventProvider" />
  </providers>

  <rules>
  <clear />

  <add name="All Errors Default" eventName="All Errors" provider="SqlWebEventProvider"
  profile="Default" minInstances="1" maxLimit="Infinite" minInterval="00:00:00" />
  </rules>
  </healthMonitoring>
  </system.web>
</configuration>
```

The health monitoring configuration markup above uses `<clear />` elements to wipe the pre-defined health monitoring configuration information from the `<eventMappings>`, `<providers>`, and `<rules>` sections. It then adds a single entry to each of these sections.

- The `<eventMappings>` element defines a single health monitoring event of interest named "All Errors", which is raised whenever an unhandled exception occurs.

- The `<providers>` element defines a single log source named "SqlWebEventProvider" that uses the `SqlWebEventProvider` class. The `connectionStringName` attribute has been set to "ReviewsConnectionString", which is the name of our connection string defined in the `<connectionStrings>` section.
- Finally, the <rules> element indicates that when an "All Errors" event transpires that it should be logged using the "SqlWebEventProvider" provider.

This configuration information instructs the health monitoring system to log all unhandled exceptions to the Book Reviews database.

> **NOTE**
>
> The `WebBaseErrorEvent` event is only raised for server errors; it is not raised for HTTP errors, such as a request for an ASP.NET resource that is not found. This differs from the behavior of the `HttpApplication` class's `Error` event, which is raised for both server and HTTP errors.

To see the health monitoring system in action, visit the website and generate a runtime error by visiting `Genre.aspx?ID=foo`. You should see the appropriate error page - either the Exception Details Yellow Screen of Death (when visiting locally) or the custom error page (when visiting the site in production). Behind the scenes, the health monitoring system logged the error information to the database. There should be one record in the `aspnet_WebEvent_Events` table (see **Figure 1**); this record contains information about the runtime error that just occurred.

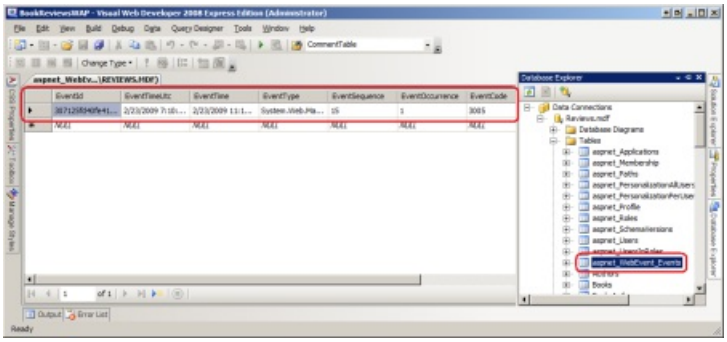


Figure 1: The Error Details Were Logged to the `aspnet_WebEvent_Events` Table
(Click to view full-size image)

Displaying the Error Log In a Web Page

With the website's current configuration, the health monitoring system logs all unhandled exceptions to the database. However, health monitoring does not provide any mechanism to view the error log through a web page. However, you could build an ASP.NET page that displays this information from the database. (As we'll see momentarily, you can opt to have the error details sent to you in an email message.)

If you create such a page, make sure you take steps to allow only authorized users to view the error details. If your site already employs user accounts then you can use URL authorization rules to restrict access to the page to certain users or roles. For more information on how to grant or restrict access to web pages based on the logged in user, refer to my Website Security Tutorials.

> **NOTE**
>
> The subsequent tutorial explores an alternative error logging and notification system named ELMAH. ELMAH includes a built-in mechanism to view the error log from both a web page and as an RSS feed.

Logging Events to Email

The health monitoring system includes a log source provider that "logs" an event to an email message. The log

source includes the same information that is logged to the database in the email message body. You can use this log source to notify a developer when a certain health monitoring event occurs.

Let's update the Book Reviews website's configuration so that we receive an email whenever an exception occurs. To accomplish this we need to perform three tasks:

1. Configure the ASP.NET web application to send email. This is accomplished by specifying how email messages are sent via the `<system.net>` configuration element. For more information on sending email messages in an ASP.NET application refer to Sending Email in ASP.NET and the System.Net.Mail FAQ.
2. Register the email log source provider in the `<providers>` element, and
3. Add an entry to the `<rules>` element that maps the "All Errors" event to the log source provider added in step (2).

The health monitoring system includes two email log source provider classes: `SimpleMailWebEventProvider` and `TemplatedMailWebEventProvider`. The `SimpleMailWebEventProvider` class sends a plain-text email message that includes the event details and provides little customization of the email body. With the `TemplatedMailWebEventProvider` class you specify an ASP.NET page whose rendered markup is used as the body for the email message. The `TemplatedMailWebEventProvider` class gives you much greater control over the contents and format of the email message, but requires a bit more upfront work as you have to create the ASP.NET page that generates the email message's body. This tutorial focuses on using the `SimpleMailWebEventProvider` class.

Update the health monitoring system's `<providers>` element in the `Web.config` file to include a log source for the `SimpleMailWebEventProvider` class:

```
<providers>
  <clear />

  <add type="System.Web.Management.SimpleMailWebEventProvider"
  name="EmailWebEventProvider" buffer="false"
  from="support@example.com" to="support@example.com"
  subjectPrefix="Book Reviews Runtime Error: " />

  <add connectionStringName="ReviewsConnectionString" maxEventDetailsLength="1073741823"
  buffer="false" name="SqlWebEventProvider"
  type="System.Web.Management.SqlWebEventProvider" />
</providers>
```

The above markup uses the `SimpleMailWebEventProvider` class as the log source provider, and assigns it the friendly name "EmailWebEventProvider". Moreover, the `<add>` attribute includes additional configuration options, such as the To and From addresses of the email message.

With the email log source defined, all that remains is to instruct the health monitoring system to use this source to "log" unhandled exceptions. This is accomplished by adding a new rule in the `<rules>` section:

```
<rules>
  <clear />

  <add name="All Errors To Email" eventName="All Errors" provider="EmailWebEventProvider"
  profile="Default" minInstances="1" maxLimit="Infinite" minInterval="00:00:00" />

  <add name="All Errors To Database" eventName="All Errors" provider="SqlWebEventProvider"
  profile="Default" minInstances="1" maxLimit="Infinite" minInterval="00:00:00" />
</rules>
```

The `<rules>` section now includes two rules. The first one, named "All Errors To Email", sends all unhandled exceptions to the "EmailWebEventProvider" log source. This rule has the effect of sending details about errors on the website to the specified To address. The "All Errors To Database" rule logs the error details to the site's

database. Consequently, whenever an unhandled exception occurs on the site its details are both logged to the database and sent to specified email address.

Figure 2 shows the email generated by the `SimpleMailWebEventProvider` class when visiting `Genre.aspx?ID=foo`.

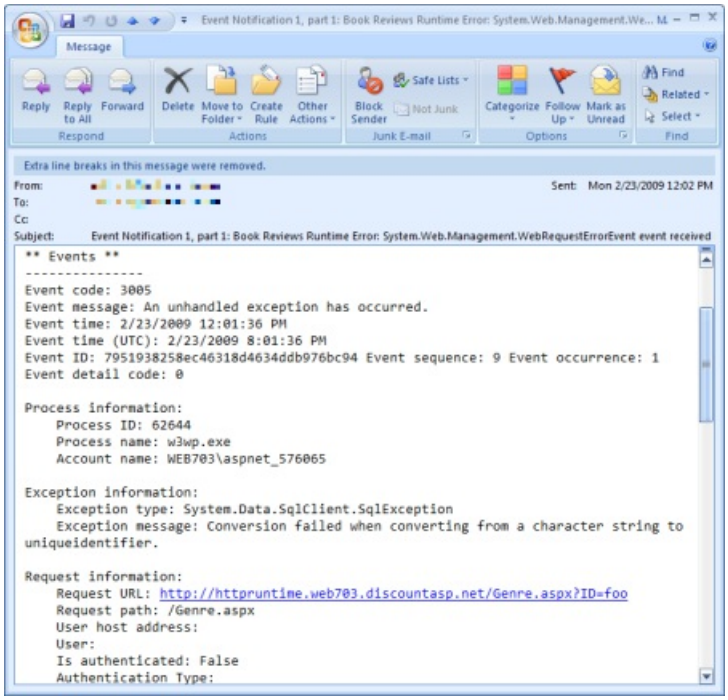


Figure 2: The Error Details are Sent in an Email Message
(Click to view full-size image)

Summary

The ASP.NET health monitoring system is designed to allow administrators to monitor the health of a deployed web application. Health monitoring events are raised when certain actions unfold, such as when the application stops, when a user successfully logs onto the site, or when an unhandled exception occurs. These events can be logged to any number of log sources. This tutorial showed how to log the details of unhandled exceptions to a database and through an email message.

This tutorial focused on using health monitoring to log unhandled exceptions, but keep in mind that health monitoring is designed to measure the overall health of a deployed ASP.NET application and includes a wealth of health monitoring events and log sources not explored here. What's more, you can create your own health monitoring events and log sources, should the need arise. If you are interested in learning more about health monitoring, a good first step is to read through Erik Reitan's health monitoring FAQ. Following that, consult How To: Use Health Monitoring in ASP.NET 2.0.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Health Monitoring Overview
- Configuring and Customizing the Health Monitoring System of ASP.NET
- FAQ - Health Monitoring in ASP.NET 2.0
- How To: Send E-Mail for Health Monitoring Notifications
- How To: Use Health Monitoring in ASP.NET
- Health Monitoring in ASP.NET

Logging Error Details with ELMAH (VB)

2/20/2020 • 18 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Error Logging Modules And Handlers (ELMAH) offers another approach to logging runtime errors in a production environment. ELMAH is a free, open source error logging library that includes features like error filtering and the ability to view the error log from a web page, as an RSS feed, or to download it as a comma-delimited file. This tutorial walks through downloading and configuring ELMAH.

Introduction

The preceding tutorial examined ASP.NET's health monitoring system, which offers an out of the box library for recording a wide array of Web events. Many developers use health monitoring to log and email the details of unhandled exceptions. However, there are a few pain points with this system. First and foremost is the lack any sort of user interface for viewing information about the logged events. If you want to see a summary of the 10 last errors, or view the details of an error that happened last week, you must either poke through the database, scan through your email Inbox, or build a web page that displays information from the `aspnet_WebEvent_Events` table.

Another pain point centers around health monitoring's complexity. Because health monitoring can be used to record a plethora of different events, and because there are a variety of options for instructing how and when events are logged, correctly configuring the health monitoring system can be an onerous task. Finally, there are compatibility issues. Because health monitoring was first added to the .NET Framework in version 2.0, it is not available for older web applications built using ASP.NET version 1.x. Moreover, the `SqlWebEventProvider` class, which we used in the preceding tutorial to logs error details to a database, only works with Microsoft SQL Server databases. You will need to create a custom log provider class should you need to log errors to an alternate data store, such as an XML file or Oracle database.

An alternative to the health monitoring system is Error Logging Modules And Handlers (ELMAH), a free, open-source error logging system created by Atif Aziz. The most notable difference between the two systems is ELAMH's ability to display a list of errors and the details of a specific error from a web page and as an RSS feed. ELMAH is easier to configure than health monitoring because it only logs errors. Furthermore, ELMAH includes support for ASP.NET 1.x, ASP.NET 2.0, and ASP.NET 3.5 applications, and ships with a variety of log source providers.

This tutorial walks through the steps involved in adding ELMAH to an ASP.NET application. Let's get started!

> **NOTE**
>
> The health monitoring system and ELMAH both have their own sets of pros and cons. I encourage you to try both systems and decide what one best suits your needs.

Adding ELMAH to an ASP.NET Web Application

Integrating ELMAH into a new or existing ASP.NET application is an easy and straightforward process that takes under five minutes. In a nutshell, it involves four simple steps:

1. Download ELMAH and add the `Elmah.dll` assembly to your web application,
2. Register ELMAH's HTTP Modules and Handler in `Web.config`,
3. Specify ELMAH's configuration options, and
4. Create the error log source infrastructure, if needed.

Let's walk through each of these four steps, one at a time.

Step 1: Downloading the ELMAH Project Files and Adding `Elmah.dll` To Your Web Application

ELMAH 1.0 BETA 3 (Build 10617), the most recent version at the time of writing, is included in the download available with this tutorial. Alternatively, you may visit the ELMAH website to get the most recent version or to download the source code. Extract the ELMAH download to a folder on your desktop and locate the ELMAH assembly file (`Elmah.dll`).

> **NOTE**
> The `Elmah.dll` file is located in the download's `Bin` folder, which has subfolders for different .NET Framework versions and for Release and Debug builds. Use the Release build for the appropriate framework version. For instance, if you are building an ASP.NET 3.5 web application, copy the `Elmah.dll` file from the `Bin\net-3.5\Release` folder.

Next, open Visual Studio and add the assembly to your project by right-clicking on the website name in the Solution Explorer and choosing Add Reference from the context menu. This brings up the Add Reference dialog box. Navigate to the Browse tab and choose the `Elmah.dll` file. This action adds the `Elmah.dll` file to the web application's `Bin` folder.

> **NOTE**
> The Web Application Project (WAP) type does not show the `Bin` folder in the Solution Explorer. Instead, it lists these items under the References folder.

The `Elmah.dll` assembly includes the classes used by the ELMAH system. These classes fall into one of three categories:

- **HTTP Modules** - an HTTP Module is a class that defines event handlers for `HttpApplication` events, such as the `Error` event. ELMAH includes multiple HTTP Modules, the three most germane ones being:
  - `ErrorLogModule` - logs unhandled exceptions to a log source.
  - `ErrorMailModule` - sends the details of an unhandled exception in an email message.
  - `ErrorFilterModule` - applies developer-specified filters to determine what exceptions are logged and what ones are ignored.
- **HTTP Handlers** - an HTTP Handler is a class that is responsible for generating the markup for a particular type of request. ELMAH includes HTTP Handlers that render error details as a web page, as an RSS feed, or as a comma-delimited file (CSV).
- **Error Log Sources** - out of the box ELMAH can log errors to memory, to a Microsoft SQL Server database, to a Microsoft Access database, to an Oracle database, to an XML file, to a SQLite database, or to a Vista DB database. Like the health monitoring system, ELMAH's architecture was built using the provider model, meaning that you can create and seamlessly integrate your own custom log source providers, if needed.

Step 2: Registering ELMAH's HTTP Module and Handler

While the `Elmah.dll` file contains the HTTP Modules and Handler needed to automatically log unhandled exceptions and to display error details from a web page, these must be explicitly registered in the web

application's configuration. The `ErrorLogModule` HTTP Module, once registered, subscribes to the `HttpApplication` 's `Error` event. Whenever this event is raised the `ErrorLogModule` logs the details of the exception to a specified log source. We'll see how to define the log source provider in the next section, "Configuring ELMAH." The `ErrorLogPageFactory` HTTP Handler factory is responsible for generating the markup when viewing the error log from a web page.

The specific syntax for registering HTTP Modules and Handlers depends upon the web server that is powering the site. For the ASP.NET Development Server and Microsoft's IIS version 6.0 and earlier, HTTP Modules and Handlers are registered in the `<httpModules>` and `<httpHandlers>` sections, which appear within the `<system.web>` element. If you are using IIS 7.0 then they need to be registered in the `<system.webServer>` element's `<modules>` and `<handlers>` sections. Fortunately, you can define the HTTP Modules and Handlers in *both* places regardless of the web server being used. This option is the most portable one as it allows the same configuration to be used in the development and production environments regardless of the web server being used.

Start by registering the `ErrorLogModule` HTTP Module and the `ErrorLogPageFactory` HTTP Handler in the `<httpModules>` and `<httpHandlers>` section in `<system.web>`. If your configuration already defines these two elements then simply include the `<add>` element for ELMAH's HTTP Module and Handler.

```
<?xml version="1.0"?>
<configuration>
  ...

  <system.web>
  ...

  <httpHandlers>
  ...

  <add verb="POST,GET,HEAD" path="elmah.axd" type="Elmah.ErrorLogPageFactory, Elmah" />
  </httpHandlers>

  <httpModules>
  ...

  <add name="ErrorLog" type="Elmah.ErrorLogModule, Elmah"/>
  </httpModules>

  ...
  </system.web>

  ...
</configuration>
```

Next, register ELMAH's HTTP Module and Handler in the `<system.webServer>` element. As before, if this element is not already present in your configuration then add it.

```
<?xml version="1.0"?>
<configuration>
  ...

  <system.webServer>
  <validation validateIntegratedModeConfiguration="false"/>
  <modules>
  ...

  <add name="Elmah.ErrorLog" type="Elmah.ErrorLogModule, Elmah" preCondition="managedHandler" />
  </modules>
  <handlers>
  ...

  <add name="Elmah" path="elmah.axd" verb="POST,GET,HEAD" type="Elmah.ErrorLogPageFactory, Elmah"
preCondition="integratedMode" />
  </handlers>
  </system.webServer>
</configuration>
```

By default, IIS 7 complains if HTTP Modules and Handlers are registered in the `<system.web>` section. The `validateIntegratedModeConfiguration` attribute in the `<validation>` element instructs IIS 7 to suppress such error messages.

Note that the syntax for registering the `ErrorLogPageFactory` HTTP Handler includes a `path` attribute, which is set to `elmah.axd`. This attribute informs the web application that if a request arrives for a page named `elmah.axd` then the request should be processed by the `ErrorLogPageFactory` HTTP Handler. We'll see the `ErrorLogPageFactory` HTTP Handler in action later in this tutorial.

Step 3: Configuring ELMAH

ELMAH looks for its configuration options in the website's `Web.config` file in a custom configuration section named `<elmah>`. In order to use a custom section in `Web.config` it must first be defined in the `<configSections>` element. Open the `Web.config` file and add the following markup to the `<configSections>`:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
  ...

  <sectionGroup name="elmah">
  <section name="security" requirePermission="false" type="Elmah.SecuritySectionHandler, Elmah"/>
  <section name="errorLog" requirePermission="false" type="Elmah.ErrorLogSectionHandler, Elmah" />
  <section name="errorMail" requirePermission="false" type="Elmah.ErrorMailSectionHandler, Elmah" />
  <section name="errorFilter" requirePermission="false" type="Elmah.ErrorFilterSectionHandler, Elmah"/>
  </sectionGroup>
  </configSections>

  ...
</configuration>
```

NOTE

If you are configuring ELMAH for an ASP.NET 1.x application then remove the `requirePermission="false"` attribute from the `<section>` elements above.

The above syntax registers the custom `<elmah>` section and its subsections: `<security>`, `<errorLog>`, `<errorMail>`, and `<errorFilter>`.

Next, add the `<elmah>` section to `Web.config`. This section should appear at the same level as the `<system.web>` element. Inside the `<elmah>` section add the `<security>` and `<errorLog>` sections like so:

```
<?xml version="1.0"?>
<configuration>
  ...

  <elmah>
  <security allowRemoteAccess="0" />

  <errorLog type="Elmah.SqlErrorLog, Elmah" connectionStringName="ReviewsConnectionString" />
  </elmah>

  ...
</configuration>
```

The `<security>` section's `allowRemoteAccess` attribute indicates whether remote access is allowed. If this value is set to 0, then the error log web page can only be viewed locally. If this attribute is set to 1 then the error log web page is enabled for both remote and local visitors. For now, let's disable the error log web page for remote visitors. We'll allow remote access later after we have an opportunity to discuss the security concerns of doing so.

The `<errorLog>` section defines the error log source, which dictates where the error details are recorded; it is similar to the `<providers>` section in the health monitoring system. The above syntax specifies the `SqlErrorLog` class as the error log source, which logs the errors to a Microsoft SQL Server database specified by the `connectionStringName` attribute value.

> **NOTE**
> ELMAH ships with additional error log providers that can be used to log errors to an XML file, a Microsoft Access database, an Oracle database, and other data stores. Refer to the sample `Web.config` file that is included with the ELMAH download for information on how to use these alternate error log providers.

Step 4: Creating the Error Log Source Infrastructure

ELMAH's `SqlErrorLog` provider logs error details to a specified Microsoft SQL Server database. The `SqlErrorLog` provider expects this database to have a table named `ELMAH_Error` and three stored procedures: `ELMAH_GetErrorsXml`, `ELMAH_GetErrorXml`, and `ELMAH_LogError`. The ELMAH download includes a file named `SQLServer.sql` in the `db` folder that contains the T-SQL for creating this table and these stored procedures. You'll need to run these statements on your database to use the `SqlErrorLog` provider.

Figures 1 and **2** show the Database Explorer in Visual Studio after the database objects needed by the `SqlErrorLog` provider have been added.

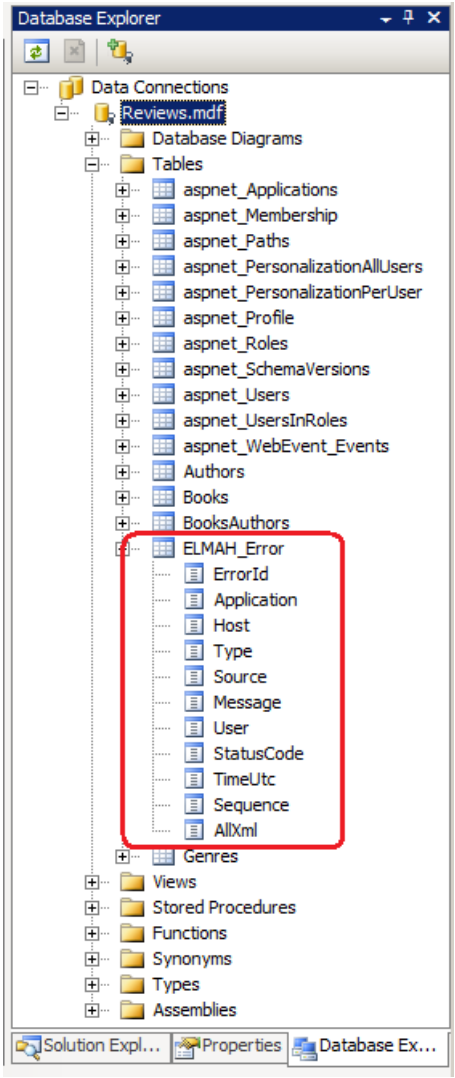


Figure 1: The `SqlErrorLog` Provider Logs Errors to the `ELMAH_Error` Table

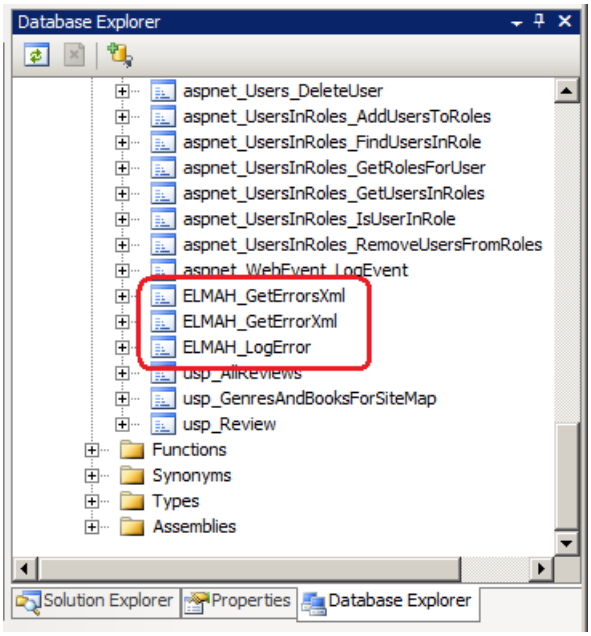


Figure 2: The `SqlErrorLog` Provider Uses Three Stored Procedures

ELMAH In Action

At this point we have added ELMAH to the web application, registered the `ErrorLogModule` HTTP Module and the `ErrorLogPageFactory` HTTP Handler, specified ELMAH's configuration options in `Web.config`, and added the

needed database objects for the `SqlErrorLog` error log provider. We are now ready to see ELMAH in action! Visit the Book Reviews website and visit a page that generates a runtime error, such as `Genre.aspx?ID=foo`, or a non-existent page, such as `NoSuchPage.aspx`. What you see when visiting these pages depends on the `<customErrors>` configuration and whether you're visiting locally or remotely. (Refer back to the *Displaying a Custom Error Page* tutorial for a refresher on this topic.)

ELMAH doesn't affect what content is shown to the user when an unhandled exception occurs; it just logs its details. This error log is accessible from the web page `elmah.axd` from the root of your website, such as `http://localhost/BookReviews/elmah.axd`. (This file does not physically exist in your project, but when a request comes in for `elmah.axd` the runtime dispatches it to the `ErrorLogPageFactory` HTTP Handler, which generates the markup sent back to the browser.)

> **NOTE**
>
> You can also use the `elmah.axd` page to instruct ELMAH to generate a test error. Visiting `elmah.axd/test` (as in, `http://localhost/BookReviews/elmah.axd/test`) causes ELMAH to throw an exception of type `Elmah.TestException`, which has the error message: " This is a test exception that can be safely ignored."

Figure 3 shows the error log when visiting `elmah.axd` from the development environment.

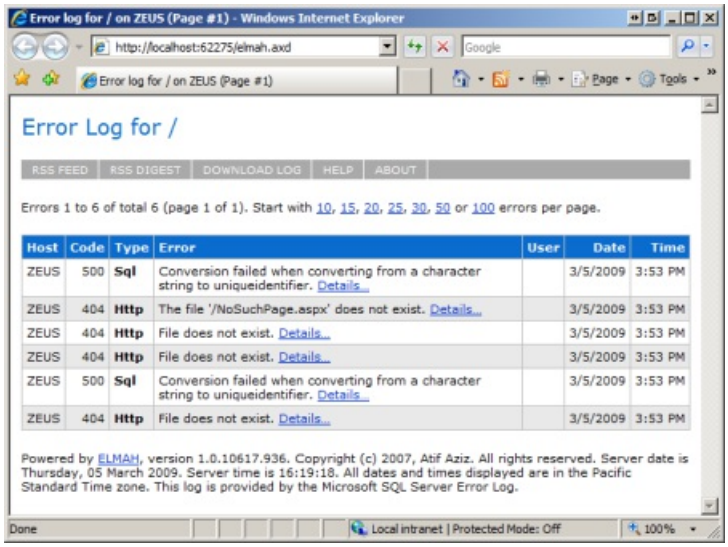


Figure 3: `Elmah.axd` Displays the Error Log from a Web Page
(Click to view full-size image)

The error log in **Figure 3** contains six error entries. Each entry includes the HTTP status code (404 or 500, for these errors), the type, the description, the name of the logged on user when the error occurred, and the date and time. Clicking the Details link displays a page that includes the same error message shown in the Error Details Yellow Screen of Death (see **Figure 4**) along with the values of the server variables at the time of the error (see **Figure 5**). You can also view the raw XML in which the error details are saved, which includes additional information such as the values in the HTTP POST header.

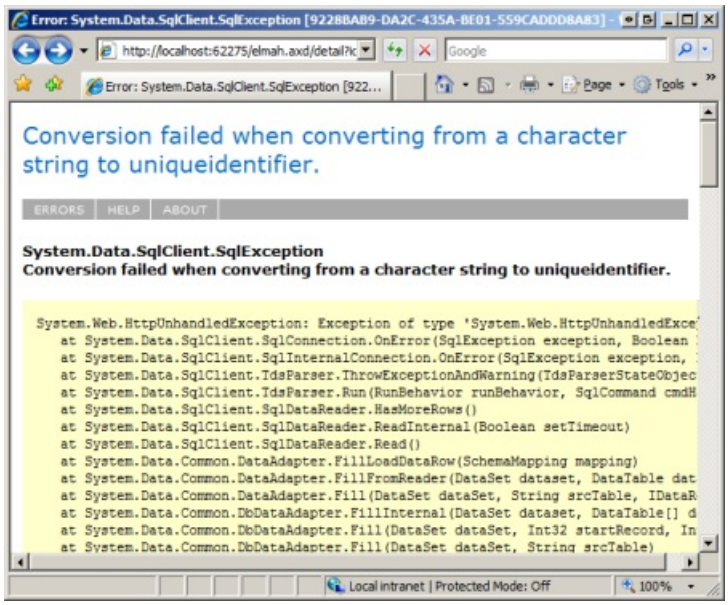


Figure 4: View the Error Details YSOD
(Click to view full-size image)

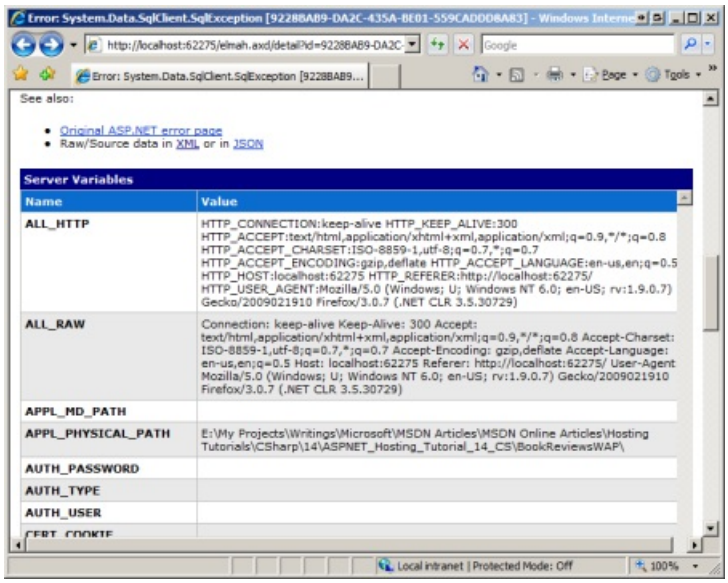


Figure 5: Explore the Values of the Server Variables Collection at the Time of the Error
(Click to view full-size image)

Deploying ELMAH to the production website entails:

- Copying the `Elmah.dll` file to the `Bin` folder on production,
- Copying the ELMAH-specific configuration settings to the `Web.config` file used on production, and
- Adding the error log source infrastructure to the production database.

We've explored techniques for copying files from development to production in previous tutorials. Perhaps the easiest way to get the error log source infrastructure on the production database is to use SQL Server Management Studio to connect to the production database and then execute the `SqlServer.sql` script file, which will create the needed table and stored procedures.

Viewing the Error Details Page on Production

After deploying your site to production, visit the production website and generate an unhandled exception. As in the development environment, ELMAH has no affect on the error page displayed when an unhandled exception occurs; instead, it merely logs the error. If you attempt to visit the error log page (`elmah.axd`) from the production environment, you will be greeted with the Forbidden page shown in **Figure 6**.

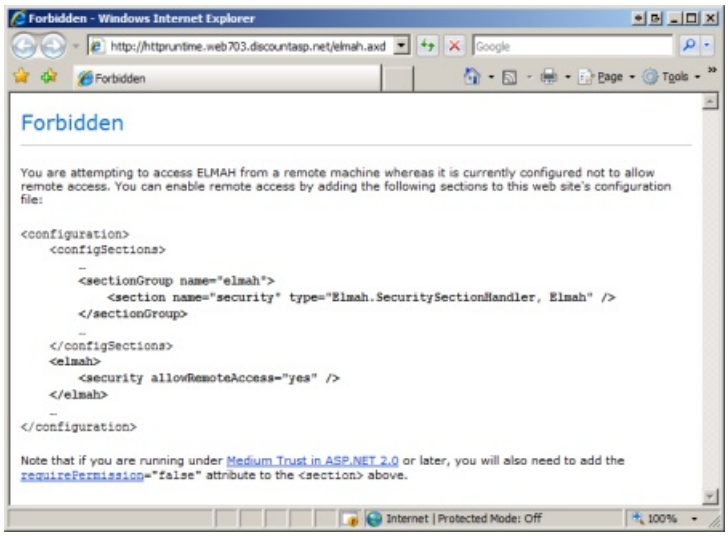


Figure 6: By Default, Remote Visitors Cannot View the Error Log Web Page
(Click to view full-size image)

Recall that in the ELMAH configuration's `<security>` section we set the `allowRemoteAccess` attribute to 0, which prohibits remote users from viewing the error log. It's important to prohibit anonymous visitors from viewing the error log, as the error details might reveal security vulnerabilities or other sensitive information. If you decide to set this attribute to 1 and enable remote access to the error log then it is important to lock down the `elmah.axd` path so that only authorized visitors can access it. This can be achieved by adding a `<location>` element to the `Web.config` file.

The following configuration permits only users in the Admin role to access the error log web page:

```
<?xml version="1.0"?>
<configuration>
  ...

  <elmah>
  <security allowRemoteAccess="1" />

  ...
  </elmah>

  ...

  <location path="elmah.axd">
  <system.web>
  <authorization>
  <allow roles="Admin" />
  <deny users="*" />
  </authorization>
  </system.web>
  </location>
</configuration>
```

NOTE

The Admin role and the three users in the system - Scott, Jisun, and Alice - were added in the *Configuring a Website That Uses Application Services* tutorial. Users Scott and Jisun are members of the Admin role. For more information on authentication and authorization, refer to my Website Security Tutorials.

The error log on the production environment can now be viewed by remote users; refer back to **Figures 3**, **4**, and **5** for screen shots of the error log web page. However, if an anonymous or non-Admin user attempts to view the error log page they are automatically redirected to the login page (`Login.aspx`), as **Figure 7** shows.

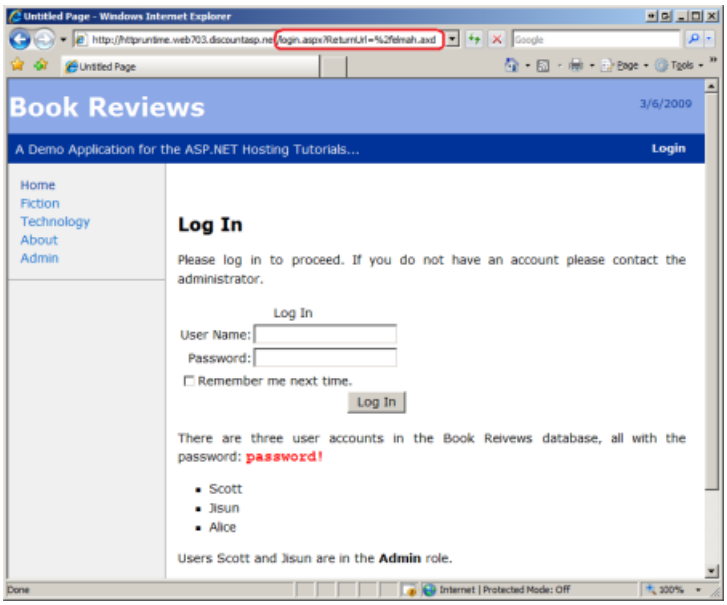


Figure 7: Unauthorized Users are Automatically Redirected to the Login Page
(Click to view full-size image)

Programmatically Logging Errors

ELMAH's `ErrorLogModule` HTTP Module automatically logs unhandled exceptions to the specified log source. Alternatively, you can log an error without having to raise an unhandled exception by using the `ErrorSignal` class and its `Raise` method. The `Raise` method is passed an `Exception` object and logs it as if that exception had been thrown and had reached the ASP.NET runtime without being handled. The difference, however, is that the request continues executing normally after the `Raise` method has been called, whereas a thrown, unhandled exception interrupts the request's normal execution and causes the ASP.NET runtime to display the configured error page.

The `ErrorSignal` class is useful in situations where there is some action that may fail, but its failure is not catastrophic to the overall operation being performed. For instance, a website may contain a form that takes the user's input, stores it in a database, and then sends the user an email informing them that they information was processed. What should happen if the information is saved to the database successfully, but there is an error when sending the email message? One option would be to throw an exception and send the user to the error page. However, this might confuse the user into thinking that the information they entered was not saved. Another approach would be to log the email-related error, but not alter the user's experience in any way. This is where the `ErrorSignal` class is useful.

```
' ... Save user's information to the database ...
...
' Attempt to send the user a confirmation email
    ' ... Send an email ...
Try
Catch e As Exception
' Error in sending email. Log it!
ErrorSignal.FromCurrentContext().Raise(e)
End Try
```

Error Notification Via Email

Along with logging errors to a database, ELMAH can also be configured to email error details to a specified recipient. This functionality is provided by the `ErrorMailModule` HTTP Module; therefore, you must register this HTTP Module in `Web.config` in order to send error details via email.

```
<?xml version="1.0"?>
<configuration>
  ...

  <system.web>
  ...

  <httpModules>
  ...

  <add name="ErrorLog" type="Elmah.ErrorLogModule, Elmah"/>
  <add name="ErrorMail" type="Elmah.ErrorMailModule, Elmah"/>
  </httpModules>
  </system.web>

  <system.webServer>
  <validation validateIntegratedModeConfiguration="false"/>
  <modules>
  ...

  <add name="Elmah.ErrorLog" type="Elmah.ErrorLogModule, Elmah" preCondition="managedHandler" />
  <add name="Elmah.ErrorMail" type="Elmah.ErrorMailModule" preCondition="managedHandler" />
  </modules>

  ...
  </system.webServer>
</configuration>
```

Next, specify information about the error email in the `<elmah>` element's `<errorMail>` section, indicating the email's sender and recipient, the subject, and whether the email is sent asynchronously.

```
<?xml version="1.0"?>
<configuration>
  ...

  <elmah>
  ...

  <errorMail from="support@example.com"
  to="support@example.com"
  subject="Book Reviews Runtime Error"
  async="true" />
  </elmah>

  ...
</configuration>
```

With the above settings in place, whenever a runtime error occurs ELMAH sends an email to support@example.com with the error details. ELMAH's error email includes the same information shown in the error details web page, namely the error message, the stack trace, and the server variables (refer back to **Figures 4** and **5**). The error email also includes the Exception Details Yellow Screen of Death content as an attachment (`YSOD.html`).

Figure 8 shows ELMAH's error email generated by visiting `Genre.aspx?ID=foo`. While **Figure 8** shows only the error message and stack trace, the server variables are included further down in the email's body.

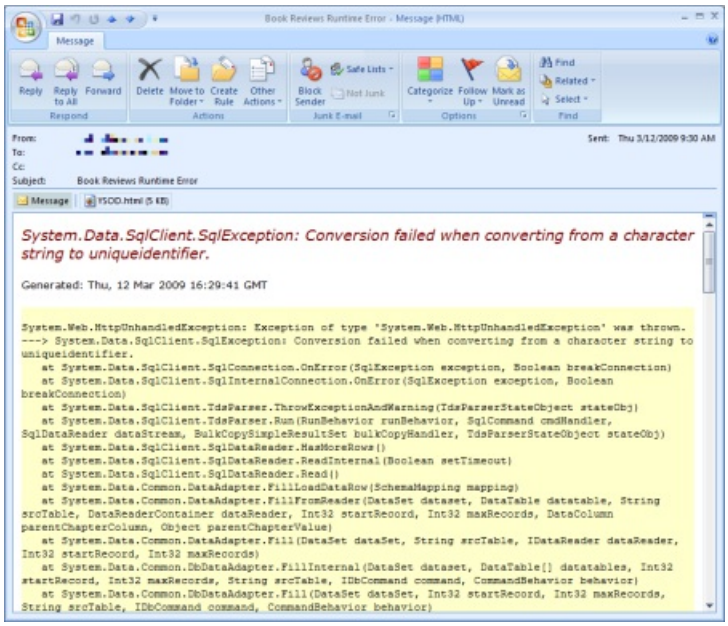


Figure 8: You Can Configure ELMAH To Send Error Details Via Email
(Click to view full-size image)

Only Logging Errors Of Interest

By default, ELMAH logs the details of every unhandled exception, including 404 and other HTTP errors. You can instruct ELMAH to ignore these or other types of errors using error filtering. The filtering logic is performed by ELMAH's `ErrorFilterModule` HTTP Module, which you'll need to register in `Web.config` in order to use the filtering logic. The rules for filtering are specified in the `<errorFilter>` section.

The following markup instructs ELMAH to not log 404 errors.

```
<?xml version="1.0"?>
<configuration>
  ...

  <elmah>
  ...

  <errorFilter>
  <test>
  <equal binding="HttpStatusCode" value="404" type="Int32" />
  </test>
  </errorFilter>
  </elmah>

  ...
</configuration>
```

> **NOTE**
>
> Don't forget, in order to use error filtering you must register the `ErrorFilterModule` HTTP Module.

The `<equal>` element inside the `<test>` section is referred to as an assertion. If the assertion evaluates to true then the error is filtered from ELMAH's log. There are other assertions available, including: `<greater>`, `<greater-or-equal>`, `<not-equal>`, `<lesser>`, `<lesser-or-equal>`, and so on. You can also combine assertions using the `<and>` and `<or>` Boolean operators. What's more, you can even include a simple JavaScript expression as an assertion, or write your own assertions in C# or Visual Basic.

For more information on ELMAH's error filtering capabilities, refer to the Error Filtering section in the ELMAH wiki.

Summary

ELMAH provides a simple, yet powerful mechanism for logging errors in an ASP.NET web application. Like Microsoft's health monitoring system, ELMAH can log errors to a database and can send the error details to a developer via email. Unlike the health monitoring system, ELMAH includes out of the box support for a wider range of error log data stores, including: Microsoft SQL Server, Microsoft Access, Oracle, XML files, and several others. Moreover, ELMAH offers a built-in mechanism for viewing the error log and details about a specific error from a web page, `elmah.axd`. The `elmah.axd` page can also render error information as an RSS feed or as a comma-separated value file (CSV), which you can read using Microsoft Excel. You can also instruct ELMAH to filter errors from the log using declarative or programmatic assertions. And ELMAH can be used with ASP.NET version 1.x applications.

Every deployed application should have some mechanism for automatically logging unhandled exceptions and sending notification to the development team. Whether this is accomplished using health monitoring or ELMAH is secondary. In other words, it doesn't really matter much whether you use health monitoring or ELMAH; evaluate both systems and then choose the one that best fits your needs. What is fundamentally important is that some mechanism be put in place to log unhandled exceptions in the production environment.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ELMAH - Error Logging Modules and Handlers
- ELMAH Project Page (source code, samples, wiki)
- Plugging ELMAH Into a Web Application to Catch Unhandled Exceptions (video)
- Security Error Log Pages
- Using HTTP Modules and Handlers to Create Pluggable ASP.NET Components
- Website Security Tutorials

PREVIOUS NEXT

Precompiling Your Website (VB)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Visual Studio offers ASP.NET developers two types of projects: Web Application Projects (WAPs) and Web Site Projects (WSPs). One of the key differences between the two project types is that WAPs must have the code explicitly compiled prior to deployment whereas the code in a WSP can be automatically compiled on the web server. However, it is possible to precompile a WSP prior to deployment. This tutorial explores the benefits of precompilation and shows how to precompile a website from within Visual Studio and from the command line.

Introduction

Visual Studio offers ASP.NET developers two different project types: Web Application Projects (WAP) and Web Site Projects (WSP). One of the key differences between these project types is that WAPs require *explicit compilation* whereas WSPs use *automatic compilation*, by default. With WAPs, you compile the web application's code into a single assembly, which is created in the website's `Bin` folder. Deployment entails copying the markup content (the `.aspx.ascx`, and `.master` files) in the project, along with the assembly in the `Bin` folder; the code-behind class files themselves do not need to be deployed. On the other hand, you deploy WSPs by copying both the markup pages and their corresponding code-behind classes to the production environment. The code-behind classes are compiled on demand on the web server.

> **NOTE**
>
> Refer back to the "Explicit Compilation Versus Automatic Compilation" section in the *Determining What Files Need to Be Deployed* tutorial for more background on the differences between the project models, explicit and automatic compilation, and how the compilation model affects deployment.

The automatic compilation option is simple to use. There's no explicit compilation step and only the files that have been modified need to be deployed, whereas explicit compilation necessitates deploying the changed markup pages and the just-compiled assembly. However, automatic deployment has two potential drawbacks:

- Because the pages must be automatically compiled when they are first visited, there can be a short but noticeable delay when an ASP.NET page is requested for the first time after being deployed.
- Automatic compilation requires that both the declarative markup and source code be present on the web server. This can be an unattractive option if you plan on selling the web application to customers who will install it on their web servers.

If either of the two above shortcomings are deal breakers, you can either switch to the WAP model or *precompile* the WSP prior to deployment. This tutorial examines the precompilation options best suited for a hosted website and walks through the precompilation process and deployment of a precompiled website.

An Overview of ASP.NET Code Generation and Compilation

Before we look at the available precompilation options, let's first talk about the code generation and compilation that occurs when an ASP.NET page is requested for the first time since it's been created or last updated. As you know, ASP.NET pages are composed of two portions: declarative markup in the `.aspx` file; and a source code

portion, typically in a separate code-behind class file (`.aspx.vb`). The steps performed by the runtime when an ASP.NET page is requested depends on the application's compilation model.

With WAPs, the pages' source code must be explicitly compiled into a single assembly before being deployed. During deployment, this assembly and the various markup pages are copied to the production environment. When a request arrives to the web server for an ASP.NET page, the runtime creates an instance of the page's code-behind class and invokes its `ProcessRequest` method, which starts the page lifecycle and, ultimately, generates the page's content, which is returned to the requestor. The runtime can work with the ASP.NET page's code-behind class because the code-behind class was already compiled into an assembly prior to deployment.

With WSPs and automatic compilation, there is no explicit compilation step prior to deployment. Instead, deployment involves copying both the declarative and the source code content to the production environment. When a request arrives to the web server for an ASP.NET page for the first time since the page has been created or last updated, the runtime must first compile the code-behind class into an assembly. This compiled assembly is saved in the folder `%WINDIR%\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files`, although the location of this folder can be customized via the `<pages>` element in `Web.config`. Because the assembly is saved to disk, it does not need to be recompiled on subsequent requests to the same page.

> **NOTE**
> As you would expect, there is a slight delay when requesting a page for the first time (or for the first time since it's been changed) in a site that uses automatic compilation as it takes a moment for the server to compile the page's code and save the resulting assembly to disk.

In short, with explicit compilation you are required to compile the website's source code before deployment, saving the runtime from having to perform that step. With automatic compilation the runtime handles the compilation of the pages' source code, but with a slight initialization cost for the first visit to the page since it was created or last updated.

But what about the declarative part of ASP.NET pages (the `.aspx` file)? It's obvious that there's a relationship between the `.aspx` files and the code in their code-behind classes, as the Web controls defined in the declarative markup are accessible in the code. It's also obvious that the content in the `.aspx` files greatly influences the rendered markup generated by the page. So how does the runtime work with the text, HTML, and Web control syntax defined in the `.aspx` file to generate the requested page's rendered content?

I don't want to get too sidetracked on the low-level implementation details, which vary between WAPs and WSPs, but in a nutshell the runtime automatically generates a class file that contains the various Web controls as protected members and methods. This generated file is implemented as a *partial class* to the corresponding code-behind class. (Partial classes allow for the contents of a single class to be spread across multiple files.) Therefore, the code-behind class is defined in two places: in the `.aspx.vb` file that you create, and in this auto-generated class created by the runtime. This auto-generated class is stored in the `%WINDIR%\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files` folder.

The important take away here is that for an ASP.NET page to be rendered by the runtime both its declarative and source code portions must be compiled into an assembly. With WAPs, the source code is explicitly compiled into an assembly prior to deployment, but the declarative markup must still be converted into code and compiled by the runtime on the web server. With WSPs using automatic compilation, both the source code and the declarative markup need to be compiled by the web server.

It is possible to use explicit compilation with the WSP model. You can explicitly compile the source code portion, like with the WAP model. What's more, you can also compile the declarative markup.

Precompilation Options

The .NET Framework ships with an ASP.NET compilation tool (`aspnet_compiler.exe`) that enables you to compile the source code (and even the content) of an ASP.NET application built using the WSP model. This tool was released with the .NET Framework version 2.0 and is located in the `%WINDIR%\Microsoft.NET\Framework\v2.0.50727` folder; it can be used from the command line or launched from within Visual Studio via the Build menu's Publish Web Site option.

The compilation tool provides two general forms of compilation: in-place precompilation and precompilation for deployment. With in-place precompilation you run the `aspnet_compiler.exe` tool from the command-line and specify the path to the virtual directory or physical path of a website that resides on your computer. The compilation tool then compiles each ASP.NET page in the project, storing the compiled version in the `%WINDIR%\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files` folder just like if the pages had each been visited for the first time from a browser. In-place precompilation can speed up the first request made to newly deployed ASP.NET pages on your site because it alleviates the runtime from needing to perform this step. However, in-place precompilation is not useful for the majority of hosted websites because it requires that you are able to run programs from the web server's command-line. In shared hosting environments this level of access is not permitted.

> **NOTE**
>
> For more information on in-place precompilation, check out How To: Precompile ASP.NET Web Sites and Precompilation in ASP.NET 2.0.

Instead of compiling the pages in the website to the `Temporary ASP.NET Files` folder, precompilation for deployment compiles the pages to a directory of your choosing and in a format that can be deployed to the production environment.

There are two flavors of precompilation for deployment that we explore in this tutorial: precompilation with an updatable user interface, and precompilation with a non-updatable user interface. Precompilation with an updatable user interface leaves the declarative markup in the `.aspx` , `.ascx` , and `.master` files, thereby allowing a developer to view and, if desired, modify the declarative markup on the production server. Precompilation with a non-updatable user interface generates `.aspx` pages that are void of any content and removes `.ascx` and `.master` files, thereby hiding the declarative markup and prohibiting a developer from changing it from the production environment.

Precompiling for Deployment With an Updatable User Interface

The best way to understand precompilation for deployment is to see an example in action. Let's precompile the Book Reviews WSP for deployment using an updatable user interface. The ASP.NET compilation tool can be invoked from Visual Studio's Build menu or from the command-line. This section examines using the tool from within Visual Studio; the "Precompiling from the Command Line" section looks at running the compiler tool from the command line.

Open the Book Review WSP in Visual Studio, go to the Build menu, and select the Publish Web Site menu option. This launches the Publish Web Site dialog box (see **Figure 1**), where you can specify the target location, whether or not the precompiled site's user interface is updatable, and other compiler tool options. The target location can be a remote web server or FTP server, but for now choose a folder on your computer's hard drive. Because we want to precompile the site with an updatable user interface, leave the "Allow this precompiled site to be updatable" checkbox checked and click OK.

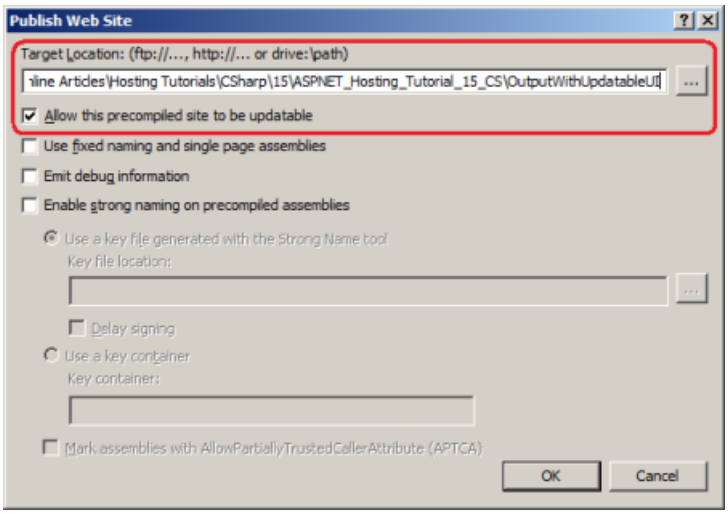


Figure 1: The ASP.NET Compilation Tool Will Precompile Your Website to the Specified Target Location (Click to view full-size image)

> **NOTE**
>
> The Publish Web Site option in the Build menu is not available in Visual Web Developer. If you are using Visual Web Developer you will need to use the command-line version of the ASP.NET compilation tool, which is covered in the "Precompiling from the Command Line" section.

After precompiling the website, navigate to the target location you entered in the Publish Web Site dialog box. Take a moment to compare the contents of this folder with the contents of your website. **Figure 2** shows the Book Reviews website folder. Note that it contains both `.aspx` and `.aspx.cs` files. Also, note that the `Bin` directory includes only one file, `Elmah.dll`, which we added in the preceding tutorial

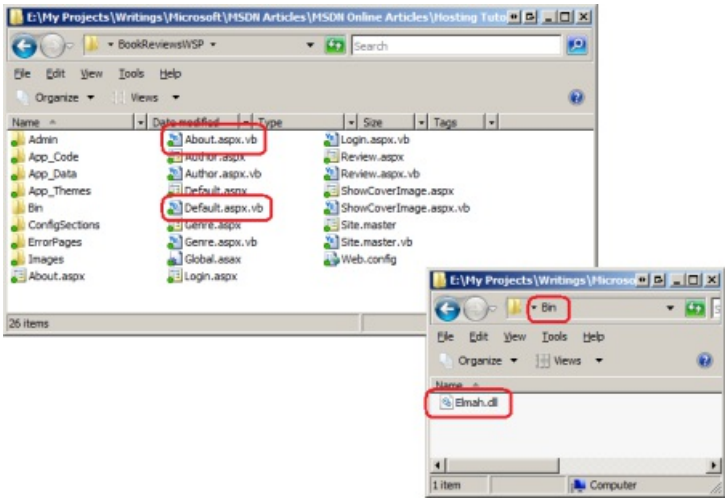


Figure 2: The Project Directory Contains `.aspx` and `.aspx.cs` Files; the `Bin` Folder Includes Just `Elmah.dll` (Click to view full-size image)

Figure 3 shows the target location folder whose contents were created by the ASP.NET compilation tool. This folder does not contain any code-behind files. Moreover, this folder's `Bin` directory includes several assemblies and two `.compiled` files in addition to the `Elmah.dll` assembly.

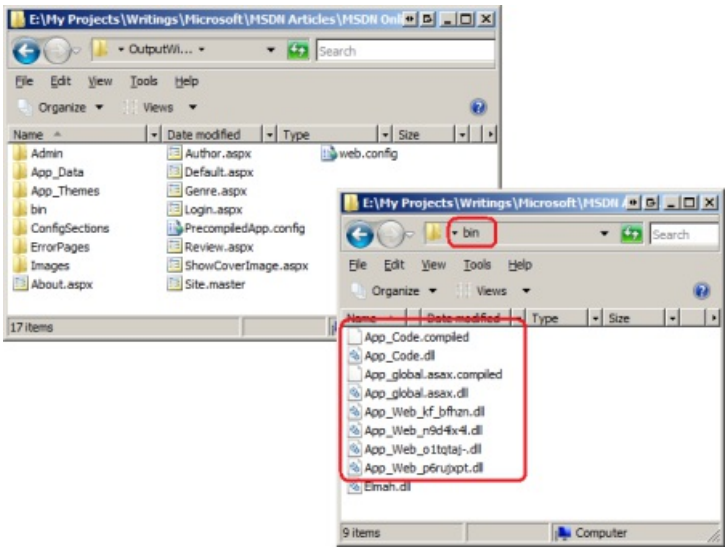


Figure 3: The Target Location Folder Includes the Files for Deployment
(Click to view full-size image)

Unlike explicit compilation in WAPs, the precompilation for deployment process does not create one assembly for the entire site. Instead, it batches together several pages into each assembly. It also compiles the `Global.asax` file (if present) into its own assembly, as well as any classes in the `App_Code` folder. The files that hold the declarative markup for ASP.NET web pages, User Controls, and master pages (`.aspx` , `.ascx` , and `.master` files, respectively) are copied as-is to the target location directory. Likewise, the `Web.config` file is copied straight over, along with any static files, such as images, CSS classes, and PDF files. For a more formal description of how the compilation tool handles various file types, refer to File Handling During ASP.NET Precompilation.

> **NOTE**
>
> You can instruct the compilation tool to create one assembly per ASP.NET page, User Control, or master page by checking the "Used fixed naming and single page assemblies" checkbox from the Publish Web Site dialog box. Having each ASP.NET page compiled into its own assembly allows for more fine-grained control over deployment. For example, if you updated a single ASP.NET web page and needed to deploy that change, you need only deploy that page's `.aspx` file and associated assembly to the production environment. Consult How To: Generate Fixed Names with the ASP.NET Compilation Tool for more information.

The target location directory also contains a file that was not part of the precompiled web project, namely `PrecompiledApp.config`. This file informs the ASP.NET runtime that the application was precompiled and whether it was precompiled with an updatable or noon-updatable UI.

Finally, take a moment to open one of the `.aspx` files in the target location using Visual Studio or your text editor of choice. When precompiling for deployment with an updatable user interface, the ASP.NET pages in the target location directory contain the exact same markup as the corresponding files in the website.

Precompiling for Deployment With a Non-Updatable User Interface

The ASP.NET compiler tool can also be used to precompile a site for deployment with a non-updatable UI. Precompiling the site with a non-updatable UI works much like precompiling with an updatable UI, the key difference being that the ASP.NET pages, User Controls, and master pages in the target directory are stripped of their markup. To precompile a website for deployment with a non-updatable UI, choose the Publish Web Site option from the Build menu, but uncheck the "Allow this precompiled site to be updatable" option (see **Figure 4**).

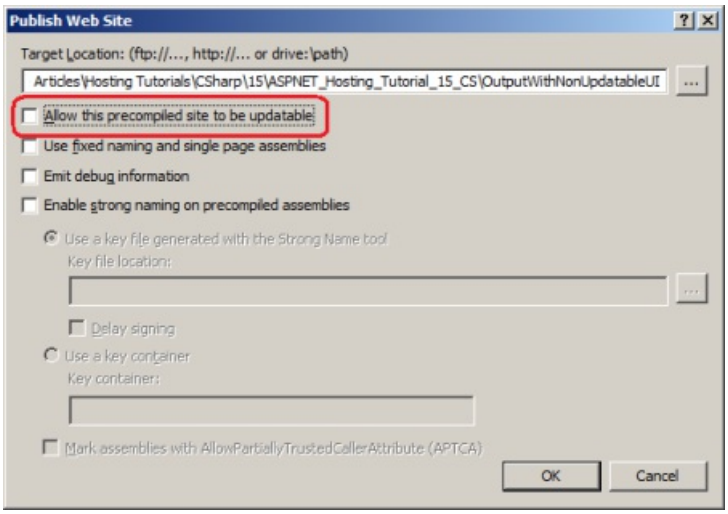


Figure 4: Uncheck the "Allow this precompiled site to be updatable" Option To Precompile With a Non-Updatable UI
(Click to view full-size image)

Figure 5 shows the target location folder after precompiling with a non-updatable user interface.

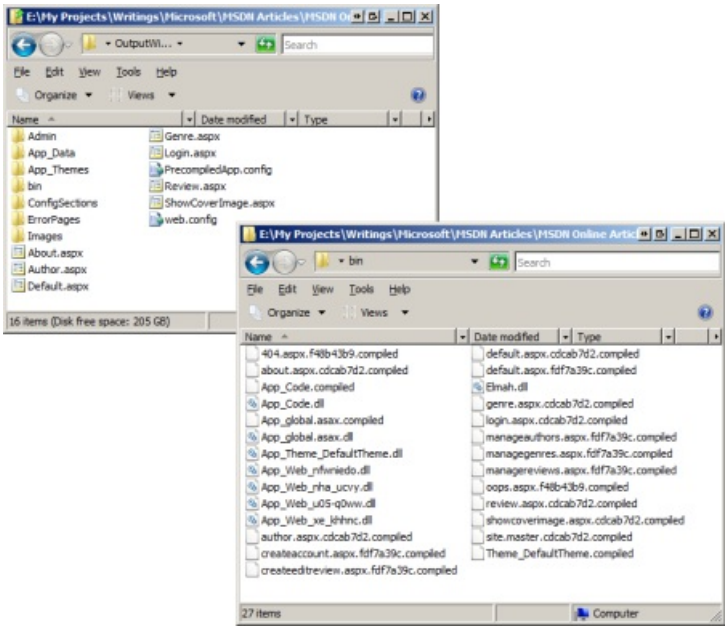


Figure 5: The Target Location Folder for Deployment With a Non-Updatable UI
(Click to view full-size image)

Compare **Figure 3** to **Figure 5**. While the two folders may look identical, note that the non-updatable UI folder lacks the master page, `Site.master`. And while **Figure 5** includes the various ASP.NET pages, if you view the contents of these files you'll see that they've been stripped of their declarative markup and replaced with the placeholder text: "This is a marker file generated by the precompilation tool, and should not be deleted!"

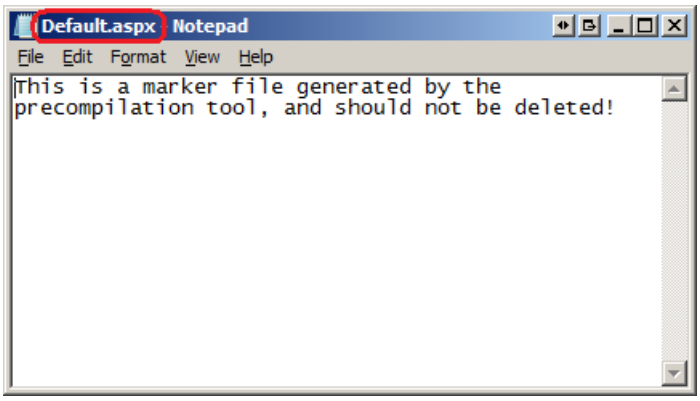


Figure 5: The Declarative Markup Has Been Removed from the ASP.NET Pages

The `Bin` folders in **Figures 3** and **5** differ more substantially. In addition to the assemblies, the `Bin` folder in **Figure 5** includes a `.compiled` file for each ASP.NET page, User Control, and master page.

Precompiling a site with a non-updatable UI is useful in situations where you do not want the ASP.NET pages' contents to be modified by the person or company who installs or manages the website in the production environment. If you build an ASP.NET web application that you sell to customers to install on their own web servers, you may want to ensure that they do not modify the look and feel of your site by directly editing the `.aspx` pages you ship them. By precompiling your website with a non-updatable UI, you ship the placeholder `.aspx` pages as part of the installation, thereby preventing your customers from examining or modifying their content.

Precompiling from the Command Line

Behind the scenes, Visual Studio's Publish Web Site dialog box invokes the ASP.NET compilation tool (`aspnet_compiler.exe`) to precompile the website. Alternatively, you can invoke this tool from the command line. In fact, if you use Visual Web Developer then you will need to run the compiler tool from the command line, as Visual Web Developer's Build menu does not include the Publish Web Site option.

To use the compiler tool from the command line, start by dropping to the command line and navigating to the framework directory, `%WINDIR%\Microsoft.NET\Framework\v2.0.50727`. Next, enter the following statement into the command line:

```
aspnet_compiler -p "physical_path_to_app" -v / -f -u "target_location_folder"
```

The above command launches the ASP.NET compiler tool (`aspnet_compiler.exe`) and, via the `-p` switch, instructs it to precompile the website rooted at *physical_path_to_app*; this value will be something like `C:\MySites\BookReviews`, and should be delimited by quotation marks.

The `-v` switch specifies the virtual directory of the site. If your site is registered as the default website in the IIS metabase then you can omit the `-p` switch and just specify the virtual directory of the application. If you use the `-p` switch, the value proceeding the `-v` switch indicates the root of the website, and is used to resolve application-root references. For instance, if you specify a value of `-v /MySite` then references in the application to `~/path/file` will be resolved as `~/MySite/path/file`. Because the Book Reviews site is located at the root directory at my web hosting company I have used the switch `-v /`.

The `-f` switch, if present, instructs the compilation tool to overwrite the *target_location_folder* directory if it already exists. If you omit the `-f` switch and the target location folder already exists, the compilation tool will quit with the error: "error ASPRUNTIME: The target directory is not empty. Please delete it manually or choose a different target."

The `-u` switch, if present, informs the tool to create an updatable user interface. Omit this switch to precompile the site with a non-updatable user interface.

Lastly, the *target_location_folder* is the physical path to the target location directory; this value will be something like `C:\MySites\Output\BookReviews`, and should be delimited by quotation marks.

Deploying the Precompiled Website

At this point we have seen how to use the ASP.NET compilation tool to precompile a website using both the updatable and non-updatable user interface options. However, our examples thus far have precompiled the website to a local folder, and not to the production environment. The good news is that deploying the precompiled website is a breeze and can be done through Visual Studio or through some other file copy mechanism, such as from a stand-alone FTP client.

The Publish Web Site dialog box (first shown in **Figure 1**) has a target location option, which indicates where the

precompiled website files are copied to. This location can be a remote web server or FTP server. Entering a remote server into this textbox precompiles and deploys the website to the specified server in one step. Alternatively, you can precompile the website to a local folder and then manually copy the contents of that folder to the production environment via FTP or some other approach.

Having the precompiled website automatically deployed via Visual Studio's Publish Web Site dialog box is helpful for simple sites where there are no configuration differences between the development and production environments. However, as noted in the *Common Configuration Differences Between Development and Production* tutorial it is not uncommon for such differences to exist. For example, the Book Reviews web application uses a different database in the production environment than in the development environment. When Visual Studio publishes the website to a remote server it blindly copies up the configuration file information in the development environment.

For sites with configuration differences between the development and production environments it may be best to precompile the site to a local directory, copy over the production-specific configuration files, and then copy the contents of the precompiled output to production.

For a refresher on copying files from the development environment to the production environment refer to the *Deploying Your Website Using an FTP Client* and *Deploying Your Website Using Visual Studio* tutorials.

Summary

ASP.NET supports two modes of compilation: automatic and explicit. As discussed in previous tutorials, Web Application Projects (WAPs) use explicit compilation whereas Web Site Projects (WSPs) use automatic compilation, by default. However, it is possible to explicitly compile a WSP prior to deployment by using the ASP.NET compilation tool.

This tutorial focused on the compilation tool's Precompilation for Deployment support. When precompiling for deployment, the compilation tool creates a target location folder, compiles the specified web application's source code, and copies these compiled assemblies and the content files into the target location folder. The compilation tool can be configured to create an updatable or non-updatable user interface. When precompiling with a non-updatable user interface option, the declarative markup in the content files is removed. In a nutshell, precompilation permits you to deploy your Web Site Project-based application without including any source code files and with the declarative markup removed, if desired.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Web Site Precompilation
- Codebehind and Compilation in ASP.NET 2.0
- Precompilation in ASP.NET
- Precompiled Site Options in ASP.NET

PREVIOUS NEXT

Users and Roles On The Production Website (VB)

2/20/2020 • 9 minutes to read • Edit Online

by Scott Mitchell

Download PDF

The ASP.NET Website Administration Tool (WSAT) provides a web-based user interface for configuring Membership and Roles settings and for creating, editing, and deleting users and roles. Unfortunately, the WSAT only works when visited from localhost, meaning that you cannot reach the production website's Administration Tool through your browser. The good news is that there are workarounds that make it possible to manage users and roles on production. This tutorial looks at these workarounds and others.

Introduction

ASP.NET 2.0 introduced a number of *application services*, which are a suite of building block services that you can add to your web application. We added the Membership and Roles services to the Book Reviews website back in the *Configuring a Website That Uses Application Services* tutorial. The Membership service facilitates creating and managing user accounts; the Roles service offers an API for categorizing users into groups. The Book Reviews site has three user accounts - Scott, Jisun, and Alice - and a single role, Admin, with Scott and Jisun in the Admin role.

ASP.NET's application services are not tied to a specific implementation. Instead, you instruct the application services to use a particular *provider*, and that provider implements the service using a particular technology. We configured the Book Reviews web application to use the `SqlMembershipProvider` and `SqlRoleProvider` providers for the Membership and Roles services. These two providers store user account and role information in a SQL Server database and are the most commonly used providers for Internet-based web applications hosted at a web hosting company.

A common challenge for developers using the Membership and Roles services is managing the users and roles on the production environment. How do you delete a user account from the production website, add a new role, or add an existing user to an existing role? This tutorial explores different techniques for managing users and roles on the production website.

Using the ASP.NET Web Site Administration Tool

ASP.NET includes a Web Site Administration Tool (WSAT) that makes it easy to create and manage user accounts and roles and to specify user- and role-based authorization rules. To use the WSAT, click the ASP.NET Configuration icon in the Solution Explorer, or go to the Website or Project menu and choose the ASP.NET Configuration option. Either approach launches a web browser and points it to the WSAT at an address like:

```
http://localhost:portNumber/asp.netwebadminfiles/default.aspx?applicationPhysicalPath=pathToApplication
```

The WSAT is divided into three sections:

- **Security** - manage users, roles, and authorization rules.
- **ApplicationConfiguration** - manage the <appSettings> and SMTP settings from here. You can also take the application offline and manage debugging and tracing settings from here, as well as specify the default custom error page.
- **ProviderConfiguration** - configure the providers used by the application services.

The Security section (shown in **Figure 1**) includes links for creating new users, managing users, creating and managing roles, and creating and managing access rules. From here you can add a new role to the system, delete

an existing user, or add or remove roles from a particular user account.

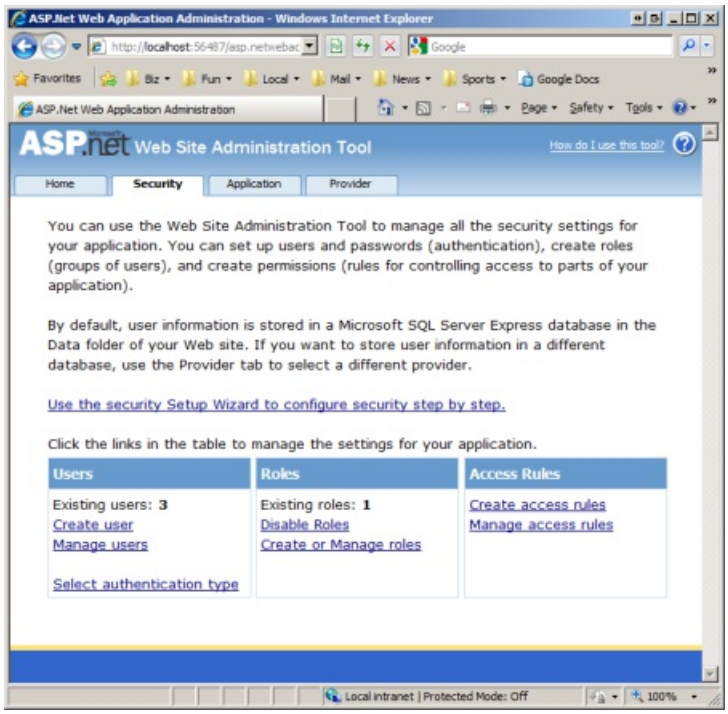


Figure 1: The WSAT Security Section Includes Options for Managing Users and Roles
(Click to view full-size image)

Unfortunately, the WSAT is only accessible locally. You cannot visit the WSAT on your remote production website; if you visit `www.yoursite.com/asp.netwebadminfiles/default.aspx` you get a 404 Not Found response. The code that powers the WSAT uses the `Membership` and `Roles` classes in the .NET Framework to create, edit, and delete users and roles. These classes consult the web application's configuration information to determine what provider to use; back in the *Configuring a Website That Uses Application Services* tutorial we setup the Book Reviews website to use the `SqlMembershipProvider` and `SqlRoleProvider` providers. This entailed adding `<membership>` and `<roleManager>` sections to `Web.config`.

```
<?xml version="1.0"?>
<configuration>
  ...

  <connectionStrings configSource="ConfigSections\databaseConnectionStrings.config" />

  <system.web>
  ...

  <membership defaultProvider="ReviewMembership">
  <providers>
  <clear />

  <add type="System.Web.Security.SqlMembershipProvider"
  name="ReviewMembership"
  connectionStringName="ReviewsConnectionString"
  applicationName="BookReviews" />
  </providers>
  </membership>

  <roleManager enabled="true" defaultProvider="ReviewRole">
  <providers>
  <clear />

  <add type="System.Web.Security.SqlRoleProvider"
  name="ReviewRole"
  connectionStringName="ReviewsConnectionString"
  applicationName="BookReviews" />
  </providers>
  </roleManager>
  </system.web>

  ...
</configuration>
```

Note that the `<membership>` and `<roleManager>` sections reference the `SqlMembershipProvider` and `SqlRoleProvider` providers in their `type` attribute, respectively. These providers store the user and role information in a specified SQL Server database. The database used by these providers is specified by the `connectionStringName` attribute, `ReviewsConnectionString`, which is defined in the `~/ConfigSections/databaseConnectionStrings.config` file. Recall that the `databaseConnectionStrings.config` file in the development environment contains the connection string to the development database whereas the `databaseConnectionStrings.config` file on production contains the connection string to the production database.

In a nutshell, the WSAT must be accessed locally through the development environment, and it works with the user and role information in the database specified in the `databaseConnectionStrings.config` file. Consequently, if we change the connection string information in the `databaseConnectionStrings.config` file on the development environment we can use the WSAT locally to manage users and roles in the production environment.

To illustrate this functionality, open the `databaseConnectionStrings.config` file in Visual Studio on the development environment and replace the development database connection string with the production database connection string. Then launch the WSAT, go the Security tab, and add a new user named Sam with password "password!" (less the quotation marks). **Figure 2** shows the WSAT screen when creating this account.

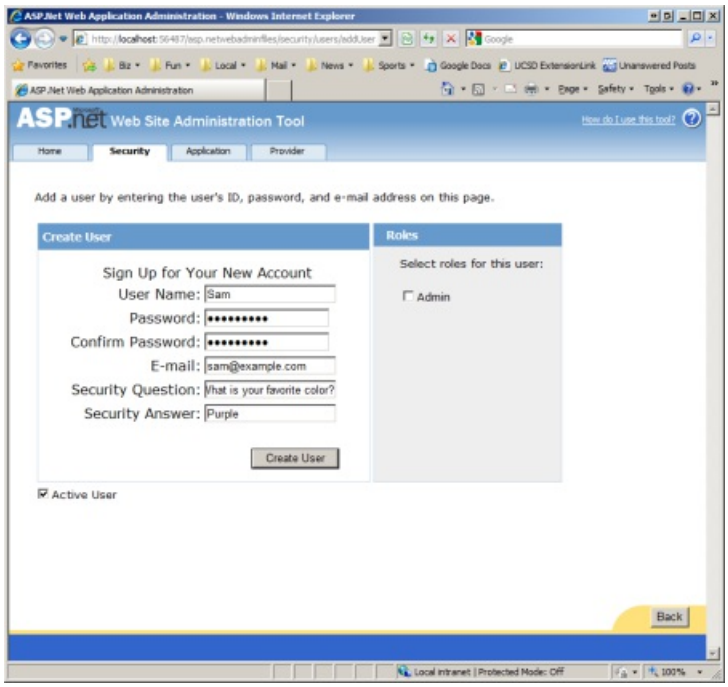


Figure 2: Create a New User Named Sam In the Production Environment
(Click to view full-size image)

Because we changed the connection string in `databaseConnectionStrings.config` to point to the production database server, Sam was added as a user in the production environment. To verify this, change the connection string in the `databaseConnectionStrings.config` file back to the development database and then visit the `Login.aspx` page in the development environment. Try to sign in as Sam (see **Figure 3**).

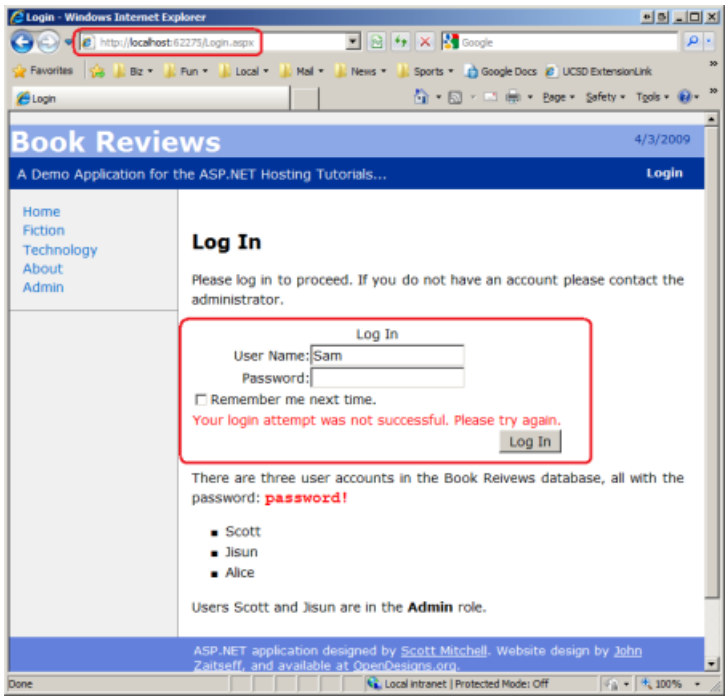


Figure 3: You Cannot Sign In As Sam in the Development Environment
(Click to view full-size image)

You cannot sign in as Sam in the development environment because the user account information does not exist in the local database. Rather, is was added to the production database. To verify this, view the contents of the `aspnet_Users` table in both the development and production databases. In the development environment there should be only three records for users Scott, Jisun, and Alice. However, the `aspnet_Users` table in the production database has four records: Scott, Jisun, Alice, and Sam. Consequently, Sam can sign in through the website in production, but not through the development environment.

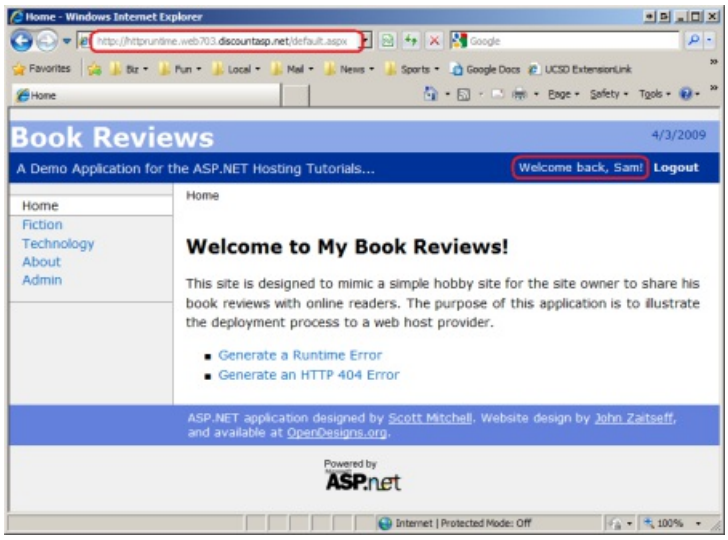


Figure 4: Sam Can Sign In On the Production Website
(Click to view full-size image)

> **NOTE**
>
> Don't forget to change the connection string in the `databaseConnectionStrings.config` file back to the development database's connect string when you're done working with the WSAT otherwise you will be working with production data when testing the site through the development environment. Also keep in mind that while the technique we just discussed allows us to use the WSAT to remotely manage users and roles, changes to any of the other WSAT configuration options (access rules, SMTP settings, debugging and tracing settings, and so on) modify the `Web.config` file. Consequently, any changes made to the settings apply to the development environment and not to the production environment.

Creating Custom User and Role Management Web Pages

The WSAT provides an out of the box system for managing users and roles, but can only be launched locally and requires making changes to the connection string information in order to manage the users and roles on production. Most websites that support user accounts also include a number of user and role administration web pages that enable administrators to manage users and roles from pages within the site. Such web-based administration pages make it much easier to manage users and roles and are essential for sites where there may be many administrators or administrators that do not have access to or the technical background to use Visual Studio to launch the WSAT.

ASP.NET includes a number of built-in Login-related Web controls that make implementing many of these administrative web pages as easy as drag and drop. For example, you can create a page for administrators to create a new user account by dragging the CreateUserWizard control onto the page and setting a few properties. In fact, the page for creating users in the WSAT shown in **Figure 2** uses the same CreateUserWizard control that you can add to your pages. Furthermore, the Membership and Roles services' functionalities are available programmatically through the `Membership` and `Roles` classes in the .NET Framework. With these classes you can write code to create, edit, and delete users and roles, as well as to add or remove users to roles, to determine what users are in what roles, and to perform other user- and role-related tasks.

In the *Configuring a Website That Uses Application Services* tutorial I added a page to the `Admin` folder named `CreateAccount.aspx`. This page allows an administrator to add a new user account to the site and to specify whether or not the newly created user is in the Admin role (see **Figure 5**).

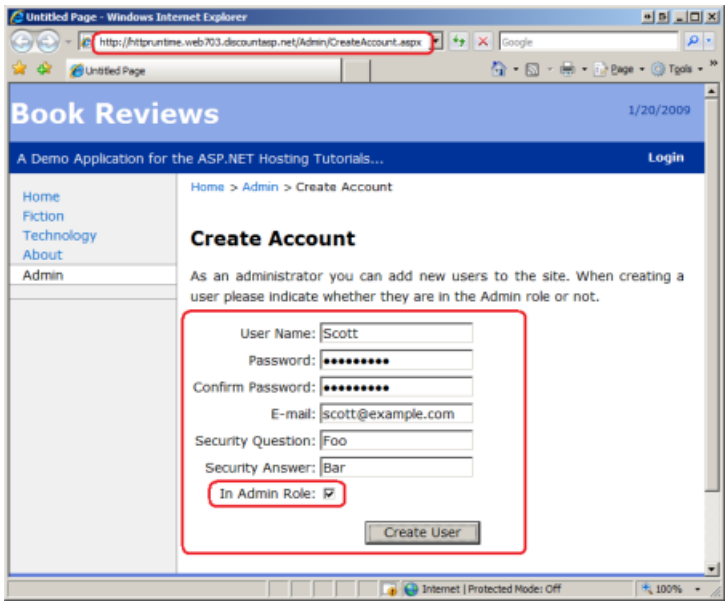


Figure 5: Administrators Can Create New User Accounts
(Click to view full-size image)

For a more detailed look at building user and role administration pages, along with step-by-step instructions on using the `Membership` and `Roles` classes and the Login-related ASP.NET Web controls, be sure to read my Website Security Tutorials. There you'll find guidance on how to build web pages for creating new accounts, creating and managing roles, assigning users to roles, and other common administrative tasks.

To implement WSAT-like functionality on the production website you can always build your own series of web pages that implement the WSAT's features. To help get started, check out the WSAT source code, which is located in the folder `%WINDIR%\Microsoft.NET\Framework\v2.0.50727\ASP.NETWebAdminFiles`. Another option is to use Dan Clem's WSAT alternative, which he shares in his article, Rolling Your Own Web Site Administration Tool. Dan walks readers through the process of building a custom WSAT-like tool, includes his application's source code for download (in C#), and gives step-by-step instructions for adding his custom WSAT to a hosted website.

Summary

The ASP.NET Web Site Administration Tool (WSAT) can be used in tandem with the Membership and Roles application services to manage user and role information for your website. Unfortunately, the WSAT is only accessible locally and cannot be visited from your production website. However, by changing the connection string in the development environment to point to the production database you can use the WSAT to manage the users and roles on the production website.

While the WSAT approach affords a quick and easy way to manage users and roles, it necessitates launching the WSAT from Visual Studio as well as temporary changes to the connection string information. The WSAT offers a quick way to manage users and roles on production, but is cumbersome and does not work well for websites with multiple administrators or with administrators who do not have or are not familiar with Visual Studio and the WSAT. For these reasons, most websites that support user accounts include a set of administrative web pages. Such a set of web pages eliminates the need for the WSAT and used by various administrative users from any computer.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Examining ASP.NET's Membership, Roles, and Profile
- Rolling Your Own Web Site Administration Tool
- Web Site Administration Tool Overview

- Website Security Tutorials

PREVIOUS

Older Versions - Ajax Control Toolkit

2/20/2020 • 3 minutes to read • Edit Online

How to get started with the Ajax Control Toolkit, and how to use some of the most popular controls. DevExpress maintains the free and open sourced Ajax Control Toolkit.

DevExpress, developers of award-winning ASP.NET WebForms, MVC and HTML5 UI Controls and Frameworks, took over the maintenance of the ASP.NET AJAX Control Toolkit in order to bring this highly-demanded open-source library back to a dependable state. Learn more.

- Getting Started
  - Get Started with the AJAX Control Toolkit (C#)
  - Using AJAX Control Toolkit Controls and Control Extenders (C#)
  - Creating a Custom AJAX Control Toolkit Control Extender (C#)
  - Get Started with the AJAX Control Toolkit (VB)
  - Using AJAX Control Toolkit Controls and Control Extenders (VB)
  - Creating a Custom AJAX Control Toolkit Control Extender (VB)
- Accordion
  - Databinding to an Accordion (C#)
  - Dynamically Adding Accordion Pane (C#)
  - Databinding to an Accordion (VB)
  - Dynamically Adding Accordion Pane (VB)
- Animation
  - Adding Animation to a Control (C#)
  - Executing Several Animations at Same Time (C#)
  - Executing Several Animations after Each Other (C#)
  - Animation Depending On a Condition (C#)
  - Picking One Animation Out Of a List (C#)
  - Animating in Response To User Interaction (C#)
  - Disabling Actions during Animation (C#)
  - Triggering an Animation in another Control (C#)
  - Modifying Animations From Server Side (C#)
  - Executing Animations Using Client-Side Code (C#)
  - Changing an Animation Using Client-Side Code (C#)
  - Animating an UpdatePanel Control (C#)
  - Dynamically Controlling UpdatePanel Animations (C#)
  - Adding Animation to a Control (VB)
  - Executing Several Animations at Same Time (VB)
  - Executing Several Animations after Each Other (VB)
  - Animation Depending On a Condition (VB)
  - Picking One Animation Out Of a List (VB)

  - Animating in Response To User Interaction (VB)
  - Disabling Actions during Animation (VB)
  - Triggering an Animation in another Control (VB)
  - Modifying Animations From Server Side (VB)
  - Executing Animations Using Client-Side Code (VB)
  - Changing an Animation Using Client-Side Code (VB)
  - Animating an UpdatePanel Control (VB)
  - Dynamically Controlling UpdatePanel Animations (VB)
- CascadingDropdown
  - Filling a List Using CascadingDropDown (C#)
  - Using CascadingDropDown with a Database (C#)
  - Presetting List Entries with CascadingDropDown (C#)
  - Using Auto-Postback with CascadingDropDown (C#)
  - Filling a List Using CascadingDropDown (VB)
  - Using CascadingDropDown with a Database (VB)
  - Presetting List Entries with CascadingDropDown (VB)
  - Using Auto-Postback with CascadingDropDown (VB)
- CollapsiblePanel
  - Collapsing and Expanding a Panel from JavaScript (C#)
  - Collapsing and Expanding a Panel from JavaScript (VB)
- ColorPicker
  - ColorPicker Control Extender (C#)
  - ColorPicker Control Extender (VB)
- ComboBox
  - ComboBox Control (C#)
  - ComboBox Control (VB)
- ConfirmButton
  - Using a ConfirmButton In a Repeater (C#)
  - Using a ConfirmButton In a Repeater (VB)
- DropShadow
  - Adjusting the Z-Index of a DropShadow (C#)
  - Manipulating DropShadow Properties from Client Code (C#)
  - Adjusting the Z-Index of a DropShadow (VB)
  - Manipulating DropShadow Properties from Client Code (VB)
- DynamicPopulate
  - Dynamically Populating a Control (C#)
  - Dynamically Populating a Control Using JavaScript Code (C#)
  - Using DynamicPopulate with a User Control And JavaScript (C#)
  - Dynamically Populating a Control (VB)
  - Dynamically Populating a Control Using JavaScript Code (VB)
  - Using DynamicPopulate with a User Control And JavaScript (VB)
- FilteredTextBox
  - Allowing Only Certain Characters in a Text Box (C#)

  - Allowing Only Certain Characters in a Text Box (VB)
- HoverMenu
  - Using HoverMenu with a Repeater Control (C#)
  - Using HoverMenu with a Repeater Control (VB)
- HTMLEditor
  - HTML Editor Control (C#)
  - HTML Editor Control (VB)
- ModalPopup
  - Launching a Modal Popup Window from Server Code (C#)
  - Using ModalPopup with a Repeater Control (C#)
  - Handling Postbacks from a ModalPopup (C#)
  - Positioning a ModalPopup (C#)
  - Launching a Modal Popup Window from Server Code (VB)
  - Using ModalPopup with a Repeater Control (VB)
  - Handling Postbacks from a ModalPopup (VB)
  - Positioning a ModalPopup (VB)
- MutuallyExclusiveCheckbox
  - Creating Mutually Exclusive Checkboxes (C#)
  - Creating Mutually Exclusive Checkboxes (VB)
- NoBot
  - Fighting Bots (C#)
  - Fighting Bots (VB)
- NumericUpDown
  - Creating a Numeric Up/Down Control with a Web Service Backend (C#)
  - Creating a Numeric Up/Down Control with a Web Service Backend (VB)
- PasswordStrength
  - Testing the Strength of a Password (C#)
  - Testing the Strength of a Password (VB)
- Popup
  - Using Multiple Popup Controls (C#)
  - Handling Postbacks from A Popup Control With an UpdatePanel (C#)
  - Handling Postbacks from A Popup Control Without an UpdatePanel (C#)
  - Using Multiple Popup Controls (VB)
  - Handling Postbacks from A Popup Control With an UpdatePanel (VB)
  - Handling Postbacks from A Popup Control Without an UpdatePanel (VB)
- Rating
  - Creating a Rating Control (C#)
  - Creating a Rating Control (VB)
- ReorderList
  - Using Postbacks with ReorderList (C#)
  - Drag and Drop via ReorderList (C#)
  - Using Postbacks with ReorderList (VB)

- Drag and Drop via ReorderList (VB)
- Slider
  - Slider Control With Auto-Postback (C#)
  - Databinding the Slider Control (C#)
  - Slider Control With Auto-Postback (VB)
  - Databinding the Slider Control (VB)
- TextBoxWatermark
  - Using TextBoxWatermark in a FormView (C#)
  - Using TextBoxWatermark With Validation Controls (C#)
  - Using TextBoxWatermark in a FormView (VB)
  - Using TextBoxWatermark With Validation Controls (VB)

Getting Started

2/20/2020 • 2 minutes to read • Edit Online

This page describes how to download the Ajax Control Toolkit and how to add its controls to the Visual Studio or Visual Web Developer toolbox.

- Get Started with the AJAX Control Toolkit (C#)
- Using AJAX Control Toolkit Controls and Control Extenders (C#)
- Creating a Custom AJAX Control Toolkit Control Extender (C#)
- Get Started with the AJAX Control Toolkit (VB)
- Using AJAX Control Toolkit Controls and Control Extenders (VB)
- Creating a Custom AJAX Control Toolkit Control Extender (VB)

Get Started with the AJAX Control Toolkit (C#)

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Learn all you need to know to get started using the AJAX Control Toolkit.

The AJAX Control Toolkit contains more than 30 free controls that you can use in your ASP.NET applications. In this tutorial, you learn how to download the AJAX Control Toolkit and add the toolkit controls to your Visual Studio/Visual Web Developer Express toolbox.

Downloading the AJAX Control Toolkit

The AJAX Control Toolkit is an open source project developed by the members of the ASP.NET community and the ASP.NET team.

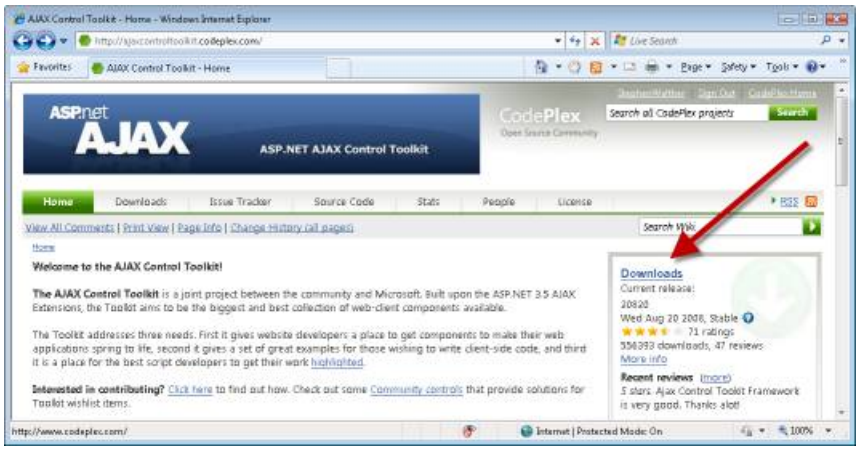


Figure 01: Downloading the AJAX Control Toolkit(Click to view full-size image)

After you download the file, you need to unblock the file. Right-click the file, select Properties, and click the **Unblock** button (see Figure 2).

Figure 02: Unblocking the AJAX Control Toolkit ZIP file(Click to view full-size image)

After you unblock the file, you can unzip the file: Right-click the file and select the **Extract All** menu option. Now, we are ready to add the toolkit to the Visual Studio/Visual Web Developer toolbox.

Adding the AJAX Control Toolkit to the Toolbox

The easiest way to use the AJAX Control Toolkit is to add the toolkit to your Visual Studio/Visual Web Developer toolbox (see Figure 3). That way, you can simply drag a toolkit control onto a page when you want to use it.

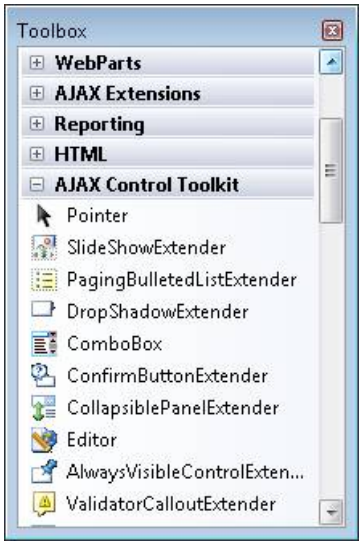


Figure 03: AJAX Control Toolkit appears in toolbox(Click to view full-size image)

First, you need to add an AJAX Control Toolkit tab to the toolbox. Follow these steps.

1. Create a new ASP.NET Website by selecting the menu option File, New Website. Double-click the Default.aspx in the Solution Explorer window to open the file in the editor.
2. Right-click the Toolbox beneath the General Tab and select the menu option **Add Tab** (see Figure 4).
3. Enter a new tab named AJAX Control Toolkit.

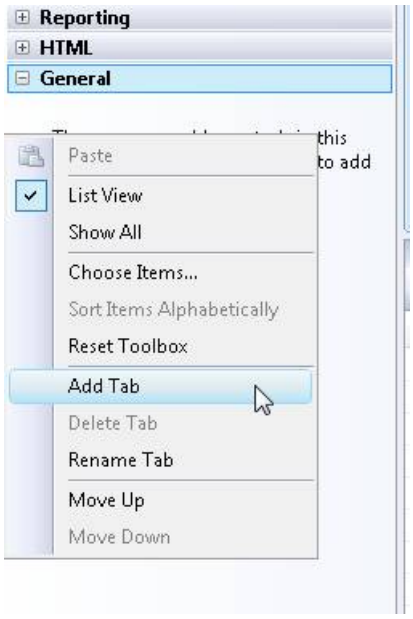


Figure 04: Adding a new tab(Click to view full-size image)

Next, you need to add the AJAX Control Toolkit controls to the new tab. Follow these steps:

- Right-click beneath the AJAX Control Toolkit tab and select the menu option **Choose Items (see Figure 5)**.
- Browse to the location where you unzipped the AJAX Control Toolkit and select the AjaxControlToolkit.dll assembly.

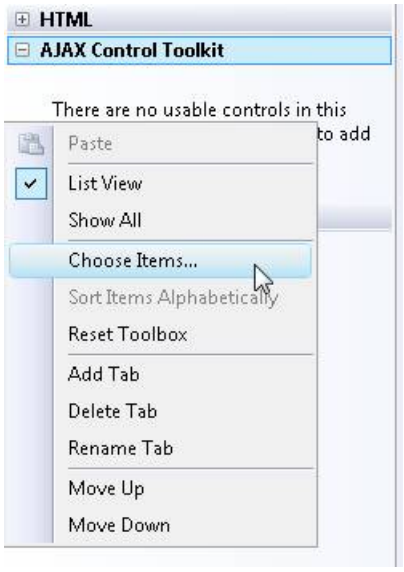


Figure 05: Choose items to add to the toolbox(Click to view full-size image)

After you complete these steps, all of the toolkit controls will appear in your toolbox.

Upgrading to a New Version of the Toolkit

If you were using an older release of the Toolkit and now need to move to a later version here are the recommended steps:

- Binaries - Delete the old version of the AjaxControlToolkit.dll assembly from your website Bin folder.
- Toolbox Items - Delete the AJAX Control Toolkit tab and follow the steps above to re-create the tab with the new version of the AjaxControlToolkit.dll assembly.

NEXT

Using AJAX Control Toolkit Controls and Control Extenders (C#)

9/16/2020 • 3 minutes to read • Edit Online

by Microsoft

> Learn how to add AJAX Control Toolkit controls and extenders to your ASP.NET pages.

The AJAX Control Toolkit contains a set of controls and control extenders. In this brief tutorial, you learn how to add both controls and control extenders to an ASP.NET page.

> **NOTE**
>
> For instructions on installing the AJAX Control Toolkit and adding the AJAX Control Toolkit to the Visual Studio/Visual Web Developer toolbox, see the tutorial Get Started with the AJAX Control Toolkit.

Using AJAX Control Toolkit Controls

An AJAX Control Toolkit control works just like a normal ASP.NET control. You can drag the control from the toolbox onto an ASP.NET page. You can add the control to the page in either Design view or Source view.

There is one special requirement when using the controls from the AJAX Control Toolkit. The page must contain a ScriptManager control. The ScriptManager control is responsible for including all of the necessary JavaScript required by the AJAX Control Toolkit controls.

For example, the AJAX Control Toolkit tab includes a control named the Editor control. This control displays a rich HTML editor. Follow these steps to add the Editor control to a page:

1. Create a new ASP.NET page named ShowEditor.aspx
2. Select the ScriptManager control from beneath the AJAX Extensions tab in the toolbox and drag the control onto the page.
3. Select the Editor control from beneath the AJAX Control Toolkit tab in the toolbox and drag the control onto the page (see Figure 1). The Designer should look like Figure 2.
4. Run the web site by selecting the menu option **Debug, Start Debugging** or hitting the F5 key.
5. You should see the page in Figure 3.

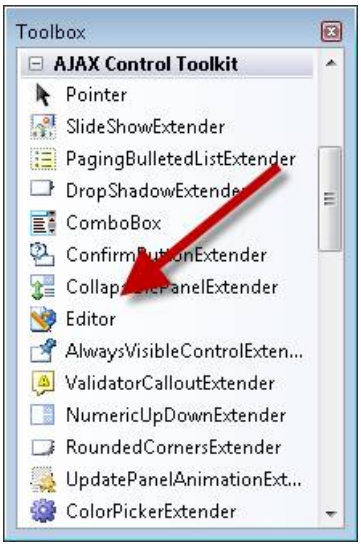


Figure 01: Selecting the HTML Editor control(Click to view full-size image)

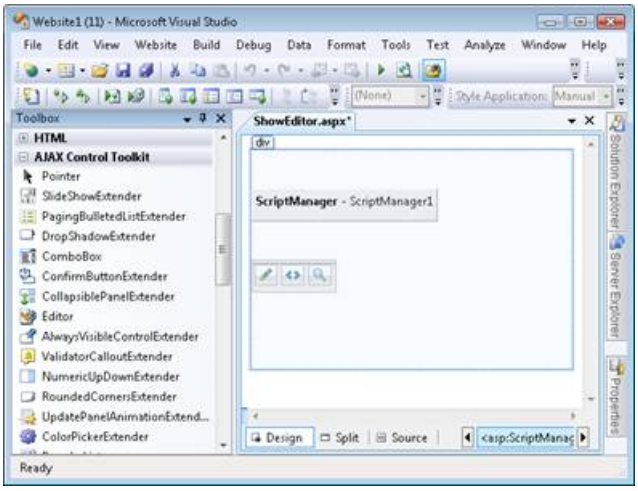


Figure 02: Visual Studio Designer with ScriptManager and Edit control(Click to view full-size image)

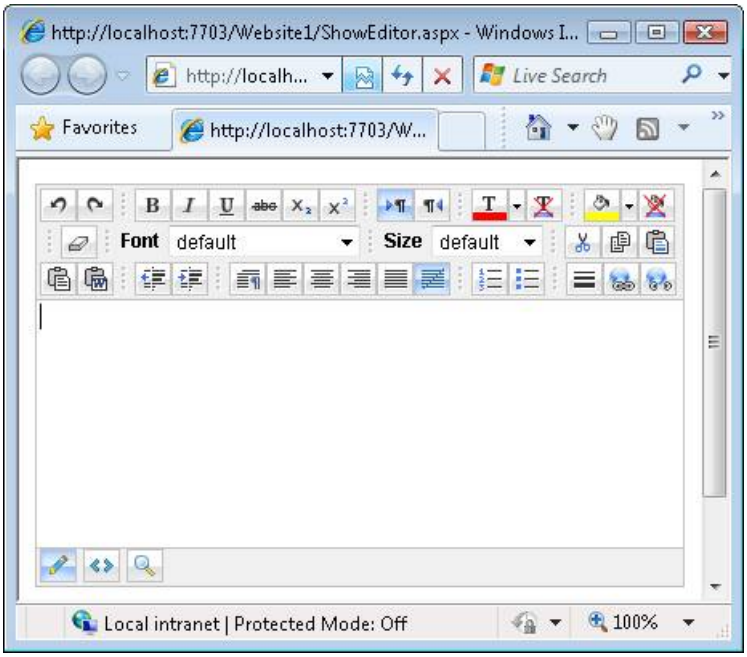


Figure 03: The DisplayEditor.aspx page(Click to view full-size image)

Using AJAX Control Toolkit Control Extenders

The AJAX Control Toolkit also contains control extenders. As its name suggests, a control extender extends the

functionality of an existing control. For example, the ConfirmButton control extender extends the standard ASP.NET Button control. The extender changes the Button control's behavior so that the Button displays a confirmation dialog when you click it.

A control extender, just like an AJAX Control Toolkit control, requires a ScriptManager control. You must add a ScriptManager control to a page before you start using control extenders in the page.

Follow these steps to use the ConfirmButton control extender:

1. Create a new ASP.NET page named ShowConfirmButton.aspx
2. Add a ScriptManager control to the page by dragging the control onto the page from beneath the AJAX Extensions tab.
3. Add a standard Button control to the page by dragging the Button from beneath the Standard tab in the toolbox onto the Designer surface.
4. Click the **Add Extender** task option (see Figure 4).
5. In the Choose Extender dialog, select ConfirmButtonExtender (see Figure 5) and click the OK button.
6. Select the Button control in the Designer and expand the Extenders, Button1_ConfirmButtonExtender node in the Properties window (see Figure 6). Assign the value *'Really?'* to the ConfirmText property.
7. Run the page by selecting the menu option **Debug, Start Debugging** or hit the F5 key.

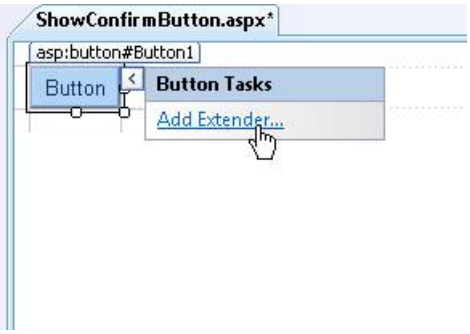


Figure 04: The Add Extender task option(Click to view full-size image)

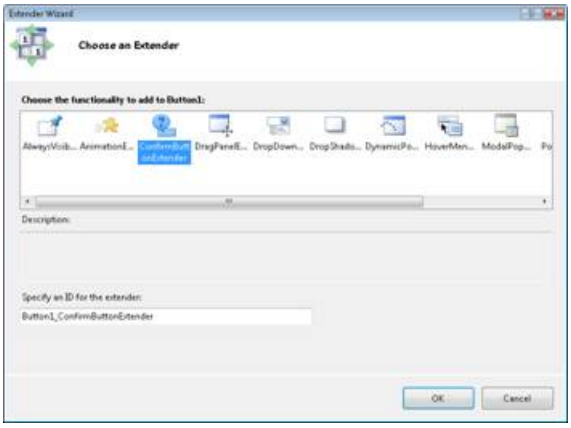


Figure 05: Selecting the ConfirmButton control extender(Click to view full-size image)

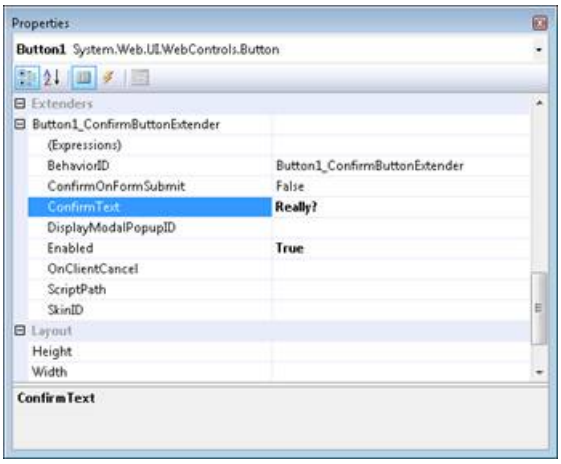


Figure 06: Setting a ConfirmButton property(Click to view full-size image)

When the page opens, you should see a button. When you click the button, you get the confirmation dialog in Figure 7.

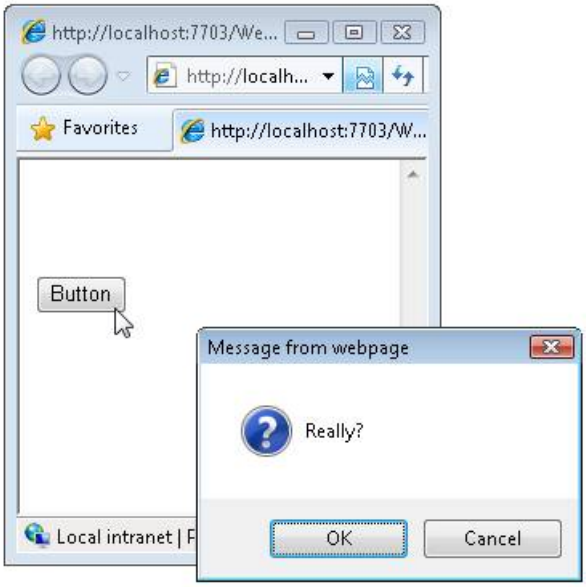


Figure 07: Displaying the confirmation dialog(Click to view full-size image)

Notice that you normally do not drag a control extender onto a page. Instead, you use the **Add Extender** task option to add an extender to a control that you have already added to a page. Notice, furthermore, that you set control extender properties by opening the property sheet for the control being extended.

A single ASP.NET control can be extended by multiple control extenders. The property sheet for the control being extended will list all of the control extenders associated with the control.

PREVIOUS NEXT

Creating a Custom AJAX Control Toolkit Control Extender (C#)

9/16/2020 • 9 minutes to read • Edit Online

by Microsoft

> Custom Extenders enable you to customize and extend the capabilities of ASP.NET controls without having to create new classes.

In this tutorial, you learn how to create a custom AJAX Control Toolkit control extender. We create a simple, but useful, new extender that changes the state of a Button from disabled to enabled when you type text into a TextBox. After reading this tutorial, you will be able to extend the ASP.NET AJAX Toolkit with your own control extenders.

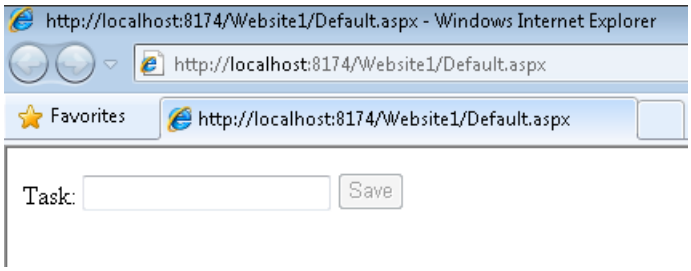
You can create custom control extenders using either Visual Studio or Visual Web Developer (make sure that you have the latest version of Visual Web Developer).

Overview of the DisabledButton Extender

Our new control extender is named the DisabledButton extender. This extender will have three properties:

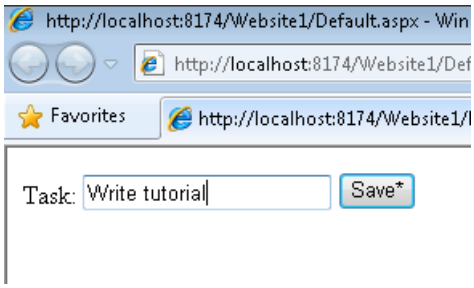
- TargetControlID - The TextBox that the control extends.
- TargetButtonID - The Button that is disabled or enabled.
- DisabledText - The text that is initially displayed in the Button. When you start typing, the Button displays the value of the Button Text property.

You hook the DisabledButton extender to a TextBox and Button control. Before you type any text, the Button is disabled and the TextBox and Button look like this:



(Click to view full-size image)

After you start typing text, the Button is enabled and the TextBox and Button look like this:



(Click to view full-size image)

To create our control extender, we need to create the following three files:

- DisabledButtonExtender.cs - This file is the server-side control class that will manage creating your extender and allow you to set the properties at design-time. It also defines the properties that can be set on your extender. These properties are accessible via code and at design time and match properties defined in the DisableButtonBehavior.js file.
- DisabledButtonBehavior.js -- This file is where you will add all of your client script logic.
- DisabledButtonDesigner.cs - This class enables design-time functionality. You need this class if you want the control extender to work correctly with the Visual Studio/Visual Web Developer Designer.

So a control extender consists of a server-side control, a client-side behavior, and a server-side designer class. You learn how to create all three of these files in the following sections.

Creating the Custom Extender Website and Project

The first step is to create a class library project and website in Visual Studio/Visual Web Developer. We'll create the custom extender in the class library project and test the custom extender in the website.

Let's start with the website. Follow these steps to create the website:

1. Select the menu option **File, New Web Site**.
2. Select the **ASP.NET Web Site** template.
3. Name the new website *Website1*.
4. Click the **OK** button.

Next, we need to create the class library project that will contain the code for the control extender:

1. Select the menu option **File, Add, New Project**.
2. Select the **Class Library** template.
3. Name the new class library with the name **CustomExtenders**.
4. Click the **OK** button.

After you complete these steps, your Solution Explorer window should look like Figure 1.

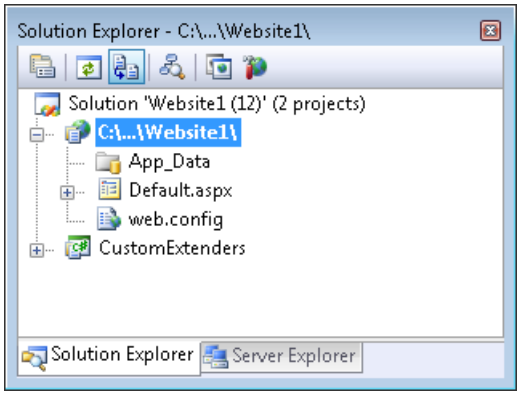


Figure 01: Solution with website and class library project(Click to view full-size image)

Next, you need to add all of the necessary assembly references to the class library project:

1. Right-click the CustomExtenders project and select the menu option **Add Reference**.
2. Select the .NET tab.
3. Add references to the following assemblies:
   a. System.Web.dll
   b. System.Web.Extensions.dll

c. System.Design.dll
d. System.Web.Extensions.Design.dll

4. Select the Browse tab.

5. Add a reference to the AjaxControlToolkit.dll assembly. This assembly is located in the folder where you downloaded the AJAX Control Toolkit.

After you complete these steps, your class library project References folder should look like Figure 2.

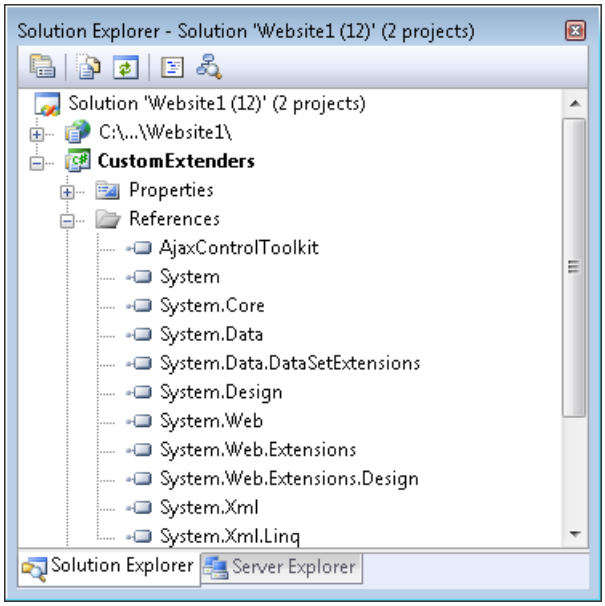


Figure 02: References folder with required references(Click to view full-size image)

Creating the Custom Control Extender

Now that we have our class library, we can start building our extender control. Let's start with the bare bones of a custom extender control class (see Listing 1).

Listing 1 - MyCustomExtender.cs

```
using System.ComponentModel;
using System.Web.UI;
using System.Web.UI.WebControls;
using AjaxControlToolkit;

[assembly: System.Web.UI.WebResource("CustomExtenders.MyControlBehavior.js", "text/javascript")]

namespace CustomExtenders
{
    [ClientScriptResource("CustomExtenders.MyControlBehavior", "CustomExtenders.MyControlBehavior.js")]
    [TargetControlType(typeof(TextBox))]
    public class MyControlExtender : ExtenderControlBase
    {

        [ExtenderControlProperty]
        [DefaultValue("")]
        public string MyProperty
        {
            get
            {
                return GetPropertyValue("MyProperty", "");
            }
            set
            {
                SetPropertyValue("MyProperty", value);
            }
        }
    }
}
```

There are several things that you notice about the control extender class in Listing 1. First, notice that the class inherits from the base ExtenderControlBase class. All AJAX Control Toolkit extender controls derive from this base class. For example, the base class includes the TargetID property that is a required property of every control extender.

Next, notice that the class includes the following two attributes related to client script:

- WebResource - Causes a file to be included as an embedded resource in an assembly.
- ClientScriptResource - Causes a script resource to be retrieved from an assembly.

The WebResource attribute is used to embed the MyControlBehavior.js JavaScript file into the assembly when the custom extender is compiled. The ClientScriptResource attribute is used to retrieve the MyControlBehavior.js script from the assembly when the custom extender is used in a web page.

In order for the WebResource and ClientScriptResource attributes to work, you must compile the JavaScript file as an embedded resource. Select the file in the Solution Explorer window, open the property sheet, and assign the value *Embedded Resource* to the **Build Action** property.

Notice that the control extender also includes a TargetControlType attribute. This attribute is used to specify the type of control that is extended by the control extender. In the case of Listing 1, the control extender is used to extend a TextBox.

Finally, notice that the custom extender includes a property named MyProperty. The property is marked with the ExtenderControlProperty attribute. The GetPropertyValue() and SetPropertyValue() methods are used to pass the property value from the server-side control extender to the client-side behavior.

Let's go ahead and implement the code for our DisabledButton extender. The code for this extender can be found in Listing 2.

Listing 2 - DisabledButtonExtender.cs

```
using System.ComponentModel;
using System.Web.UI;
using System.Web.UI.WebControls;
using AjaxControlToolkit;

[assembly: System.Web.UI.WebResource("CustomExtenders.DisabledButtonBehavior.js", "text/javascript")]

namespace CustomExtenders
{
    [ClientScriptResource("CustomExtenders.DisabledButtonBehavior",
"CustomExtenders.DisabledButtonBehavior.js")]
    [TargetControlType(typeof(TextBox))]
    public class DisabledButtonExtender : ExtenderControlBase
    {
        [ExtenderControlProperty]
        [DefaultValue("")]
        [IDReferenceProperty(typeof(Button))]
        public string TargetButtonID
        {
            get
            {
                return GetPropertyValue("TargetButtonID", "");
            }
            set
            {
                SetPropertyValue("TargetButtonID", value);
            }
        }

        [ExtenderControlProperty]
        [DefaultValue("")]
        public string DisabledText
        {
            get
            {
                return GetPropertyValue("DisabledText", "");
            }
            set
            {
                SetPropertyValue("DisabledText", value);
            }
        }

    }
}
```

The DisabledButton extender in Listing 2 has two properties named TargetButtonID and DisabledText. The IDReferenceProperty applied to the TargetButtonID property prevents you from assigning anything other than the ID of a Button control to this property.

The WebResource and ClientScriptResource attributes associate a client-side behavior located in a file named DisabledButtonBehavior.js with this extender. We discuss this JavaScript file in the next section.

Creating the Custom Extender Behavior

The client-side component of a control extender is called a behavior. The actual logic for disabling and enabling the Button is contained in the DisabledButton behavior. The JavaScript code for the behavior is included in Listing 3.

Listing 3 - DisabledButton.js

```
Type.registerNamespace('CustomExtenders');
```

```
CustomExtenders.DisabledButtonBehavior = function(element) {

    CustomExtenders.DisabledButtonBehavior.initializeBase(this, [element]);

    this._targetButtonIDValue = null;
    this._disabledTextValue = null;

}

CustomExtenders.DisabledButtonBehavior.prototype = {

    initialize : function() {
        CustomExtenders.DisabledButtonBehavior.callBaseMethod(this, 'initialize');

        // Initalization code
        $addHandler(this.get_element(), 'keyup',
        Function.createDelegate(this, this._onkeyup));
        this._onkeyup();
    },

    dispose : function() {
        // Cleanup code

        CustomExtenders.DisabledButtonBehavior.callBaseMethod(this, 'dispose');
    },

    // Property accessors
    //
    get_TargetButtonID : function() {
        return this._targetButtonIDValue;
    },

    set_TargetButtonID : function(value) {
        this._targetButtonIDValue = value;
    },

    get_DisabledText : function() {
        return this._disabledTextValue;
    },

    set_DisabledText : function(value) {
        this._disabledTextValue = value;
    },

  _onkeyup : function() {

    var e = $get(this._targetButtonIDValue);
    if (e) {
      var disabled = ("" == this.get_element().value);
      e.disabled = disabled;
      if ( this._disabledTextValue) {
        if (disabled) {
          this._oldValue = e.value;
          e.value = this._disabledTextValue;
        }
        else
        {
          if(this._oldValue){
            e.value = this._oldValue;
          }
        }
      }
    }
  }

}

CustomExtenders.DisabledButtonBehavior.registerClass('CustomExtenders.DisabledButtonBehavior',
AjaxControlToolkit.BehaviorBase);
```

The JavaScript file in Listing 3 contains a client-side class named DisabledButtonBehavior. This class, like its server-side twin, includes two properties named TargetButtonID and DisabledText which you can access using get_TargetButtonID/set_TargetButtonID and get_DisabledText/set_DisabledText.

The initialize() method associates a keyup event handler with the target element for the behavior. Each time you type a letter into the TextBox associated with this behavior, the keyup handler executes. The keyup handler either enables or disables the Button depending on whether the TextBox associated with the behavior contains any text.

Remember that you must compile the JavaScript file in Listing 3 as an embedded resource. Select the file in the Solution Explorer window, open the property sheet, and assign the value *Embedded Resource* to the **Build Action** property (see Figure 3). This option is available in both Visual Studio and Visual Web Developer.

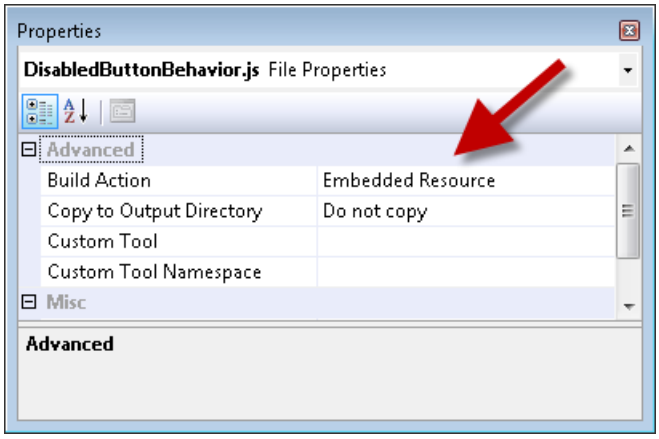


Figure 03: Adding a JavaScript file as an embedded resource(Click to view full-size image)

Creating the Custom Extender Designer

There is one last class that we need to create to complete our extender. We need to create the designer class in Listing 4. This class is required to make the extender behave correctly with the Visual Studio/Visual Web Developer Designer.

Listing 4 - DisabledButtonDesigner.cs

```
using System.Web.UI.WebControls;
using System.Web.UI;

namespace CustomExtenders
{
    class DisabledButtonDesigner : AjaxControlToolkit.Design.ExtenderControlBaseDesigner
    {
    }
}
```

You associate the designer in Listing 4 with the DisabledButton extender with the Designer attribute.You need to apply the Designer attribute to the DisabledButtonExtender class like this:

```
[Designer(typeof(DisabledButtonDesigner))]
[ClientScriptResource("CustomExtenders.DisabledButtonBehavior", "CustomExtenders.DisabledButtonBehavior.js")]
[TargetControlType(typeof(TextBox))]
public class DisabledButtonExtender : ExtenderControlBase
{
```

Using the Custom Extender

Now that we have finished creating the DisabledButton control extender, it is time to use it in our ASP.NET website. First, we need to add the custom extender to the toolbox. Follow these steps:

1. Open an ASP.NET page by double-clicking the page in the Solution Explorer window.
2. Right-click the toolbox and select the menu option **Choose Items**.
3. In the Choose Toolbox Items dialog, browse to the CustomExtenders.dll assembly.
4. Click the **OK** button to close the dialog.

After you complete these steps, the DisabledButton control extender should appear in the toolbox (see Figure 4).

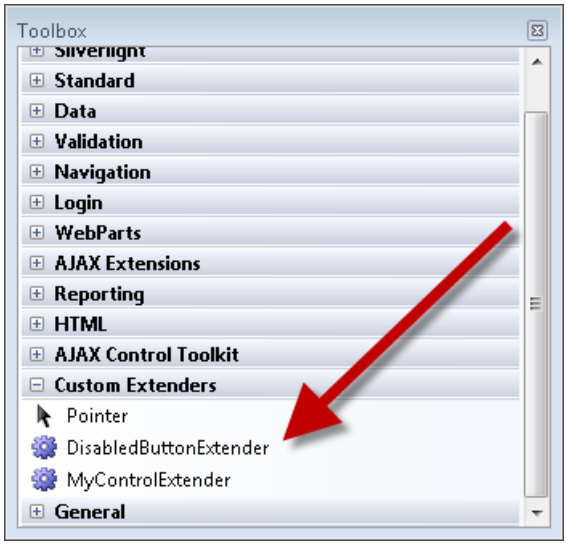


Figure 04: DisabledButton in the toolbox(Click to view full-size image)

Next, we need to create a new ASP.NET page. Follow these steps:

1. Create a new ASP.NET page named ShowDisabledButton.aspx.
2. Drag a ScriptManager onto the page.
3. Drag a TextBox control onto the page.
4. Drag a Button control onto the page.
5. In the Properties window, change the Button ID property to the value *btnSave* and the Text property to the value *Save**.

We created a page with a standard ASP.NET TextBox and Button control.

Next, we need to extend the TextBox control with the DisabledButton extender:

1. Select the **Add Extender** task option to open the Extender Wizard dialog (see Figure 5). Notice that the dialog includes our custom DisabledButton extender.
2. Select the DisabledButton extender and click the **OK** button.

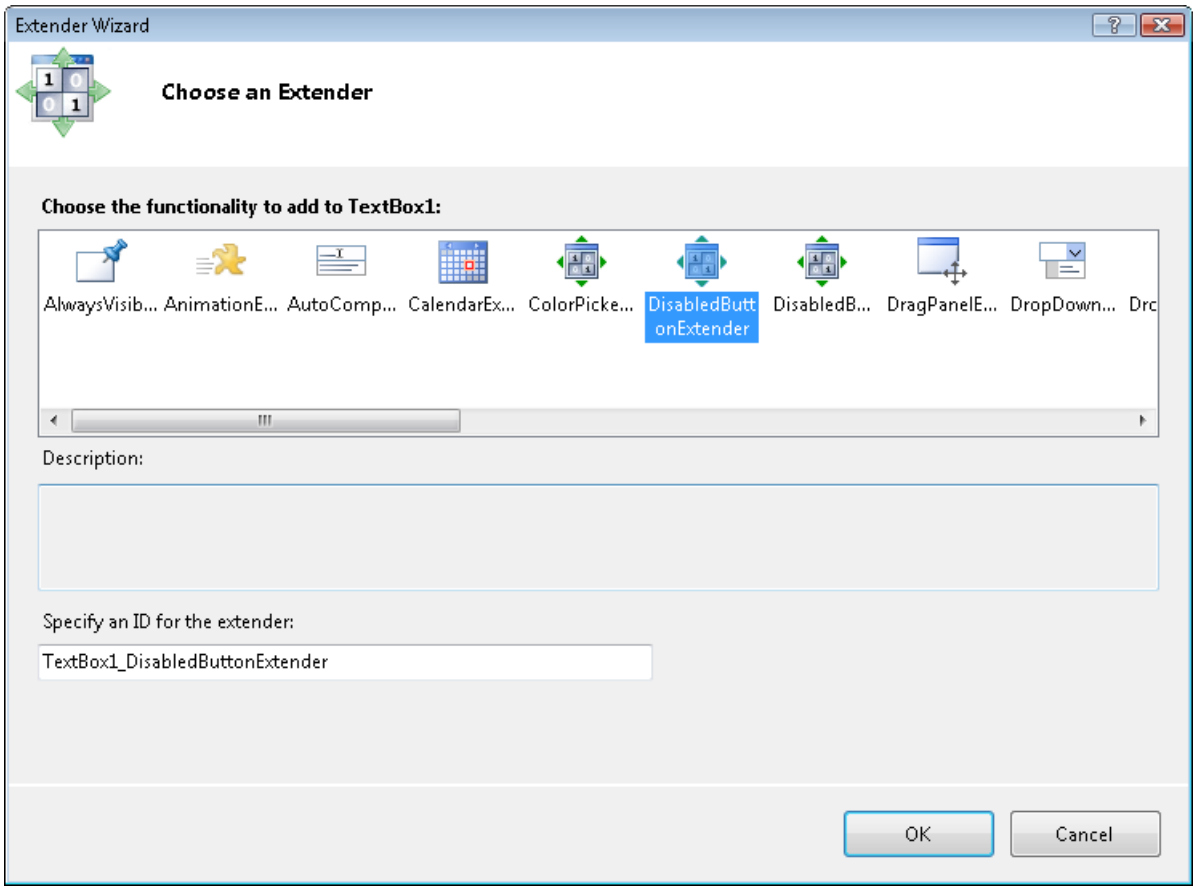


Figure 05: The Extender Wizard dialog(Click to view full-size image)

Finally, we can set the properties of the DisabledButton extender. You can modify the properties of the DisabledButton extender by modifying the properties of the TextBox control:

1. Select the TextBox in the Designer.
2. In the Properties window, expand the Extenders node (see Figure 6).
3. Assign the value *Save* to the DisabledText property and the value *btnSave* to the TargetButtonID property.

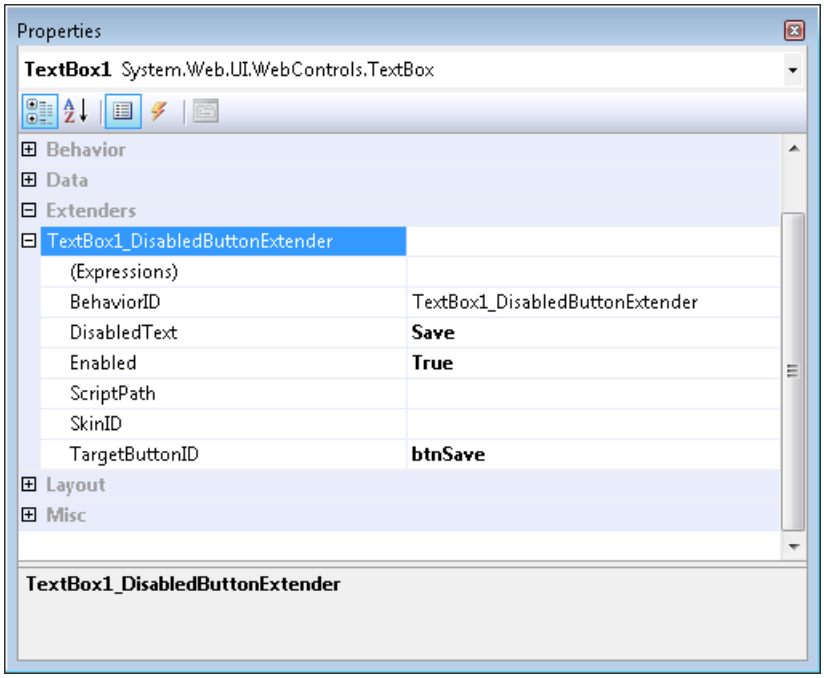


Figure 06: Setting extender properties(Click to view full-size image)

When you run the page (by hitting F5), the Button control is initially disabled. As soon as you start entering text into the TextBox, the Button control is enabled (see Figure 7).

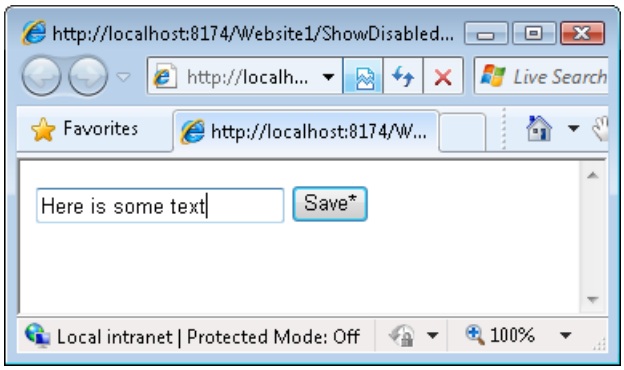


Figure 07: The DisabledButton extender in action(Click to view full-size image)

Summary

The goal of this tutorial was to explain how you can extend the AJAX Control Toolkit with custom extender controls. In this tutorial, we created a simple DisabledButton control extender. We implemented this extender by creating a DisabledButtonExtender class, a DisabledButtonBehavior JavaScript behavior, and a DisabledButtonDesigner class. You follow a similar set of steps whenever you create a custom control extender.

PREVIOUS NEXT

Get Started with the AJAX Control Toolkit (VB)

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Learn all you need to know to get started using the AJAX Control Toolkit.

The AJAX Control Toolkit contains more than 30 free controls that you can use in your ASP.NET applications. In this tutorial, you learn how to download the AJAX Control Toolkit and add the toolkit controls to your Visual Studio/Visual Web Developer Express toolbox.

Downloading the AJAX Control Toolkit

The AJAX Control Toolkit is an open source project developed by the members of the ASP.NET community and the ASP.NET team.

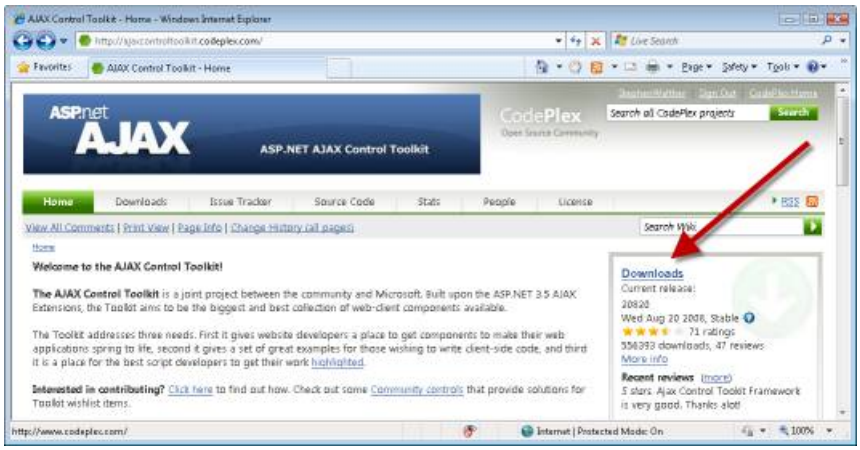


Figure 01: Downloading the AJAX Control Toolkit(Click to view full-size image)

After you download the file, you need to unblock the file. Right-click the file, select Properties, and click the **Unblock** button (see Figure 2).

Figure 02: Unblocking the AJAX Control Toolkit ZIP file(Click to view full-size image)

After you unblock the file, you can unzip the file: Right-click the file and select the **Extract All** menu option. Now, we are ready to add the toolkit to the Visual Studio/Visual Web Developer toolbox.

Adding the AJAX Control Toolkit to the Toolbox

The easiest way to use the AJAX Control Toolkit is to add the toolkit to your Visual Studio/Visual Web Developer toolbox (see Figure 3). That way, you can simply drag a toolkit control onto a page when you want to use it.

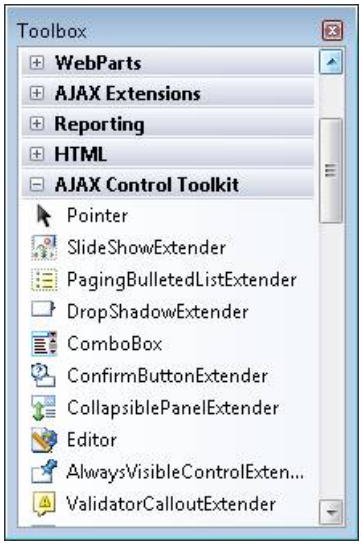


Figure 03: AJAX Control Toolkit appears in toolbox(Click to view full-size image)

First, you need to add an AJAX Control Toolkit tab to the toolbox. Follow these steps.

1. Create a new ASP.NET Website by selecting the menu option File, New Website. Double-click the Default.aspx in the Solution Explorer window to open the file in the editor.
2. Right-click the Toolbox beneath the General Tab and select the menu option **Add Tab** (see Figure 4).
3. Enter a new tab named AJAX Control Toolkit.

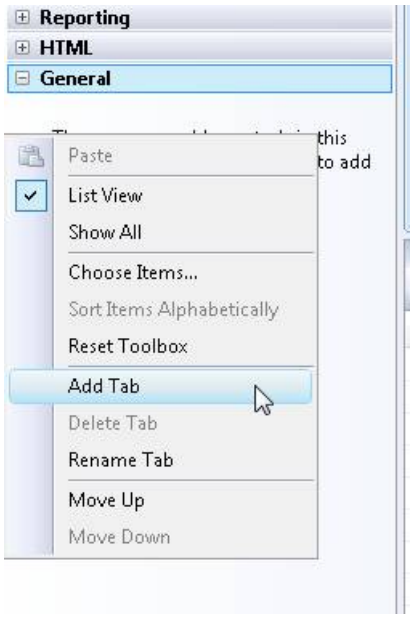


Figure 04: Adding a new tab(Click to view full-size image)

Next, you need to add the AJAX Control Toolkit controls to the new tab. Follow these steps:

- Right-click beneath the AJAX Control Toolkit tab and select the menu option **Choose Items (see Figure 5)**.
- Browse to the location where you unzipped the AJAX Control Toolkit and select the AjaxControlToolkit.dll assembly.

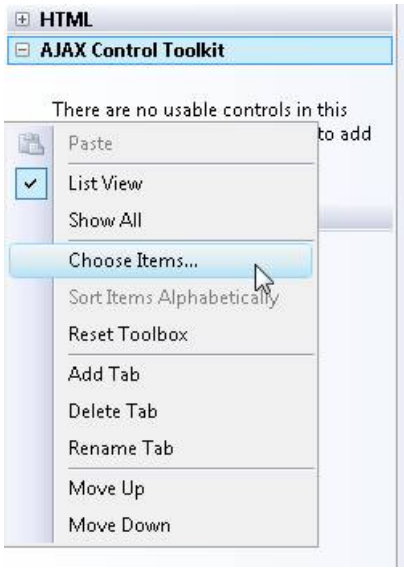


Figure 05: Choose items to add to the toolbox(Click to view full-size image)

After you complete these steps, all of the toolkit controls will appear in your toolbox.

Upgrading to a New Version of the Toolkit

If you were using an older release of the Toolkit and now need to move to a later version here are the recommended steps:

- Binaries - Delete the old version of the AjaxControlToolkit.dll assembly from your website Bin folder.
- Toolbox Items - Delete the AJAX Control Toolkit tab and follow the steps above to re-create the tab with the new version of the AjaxControlToolkit.dll assembly.

Using AJAX Control Toolkit Controls and Control Extenders (VB)

9/16/2020 • 3 minutes to read • Edit Online

by Microsoft

> Learn how to add AJAX Control Toolkit controls and extenders to your ASP.NET pages.

The AJAX Control Toolkit contains a set of controls and control extenders. In this brief tutorial, you learn how to add both controls and control extenders to an ASP.NET page.

> **NOTE**
>
> For instructions on installing the AJAX Control Toolkit and adding the AJAX Control Toolkit to the Visual Studio/Visual Web Developer toolbox, see the tutorial Get Started with the AJAX Control Toolkit.

Using AJAX Control Toolkit Controls

An AJAX Control Toolkit control works just like a normal ASP.NET control. You can drag the control from the toolbox onto an ASP.NET page. You can add the control to the page in either Design view or Source view.

There is one special requirement when using the controls from the AJAX Control Toolkit. The page must contain a ScriptManager control. The ScriptManager control is responsible for including all of the necessary JavaScript required by the AJAX Control Toolkit controls.

For example, the AJAX Control Toolkit tab includes a control named the Editor control. This control displays a rich HTML editor. Follow these steps to add the Editor control to a page:

1. Create a new ASP.NET page named ShowEditor.aspx
2. Select the ScriptManager control from beneath the AJAX Extensions tab in the toolbox and drag the control onto the page.
3. Select the Editor control from beneath the AJAX Control Toolkit tab in the toolbox and drag the control onto the page (see Figure 1). The Designer should look like Figure 2.
4. Run the web site by selecting the menu option **Debug, Start Debugging** or hitting the F5 key.
5. You should see the page in Figure 3.

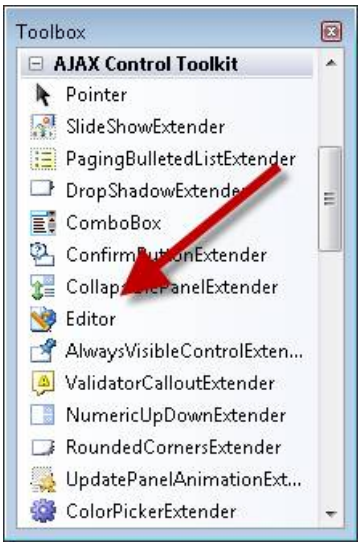


Figure 01: Selecting the HTML Editor control(Click to view full-size image)

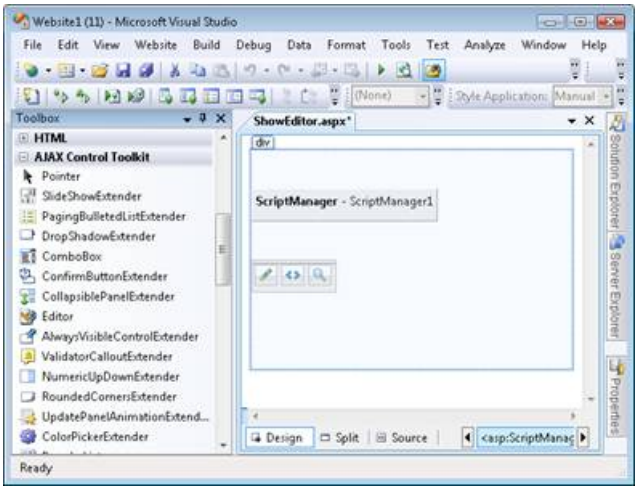


Figure 02: Visual Studio Designer with ScriptManager and Edit control(Click to view full-size image)

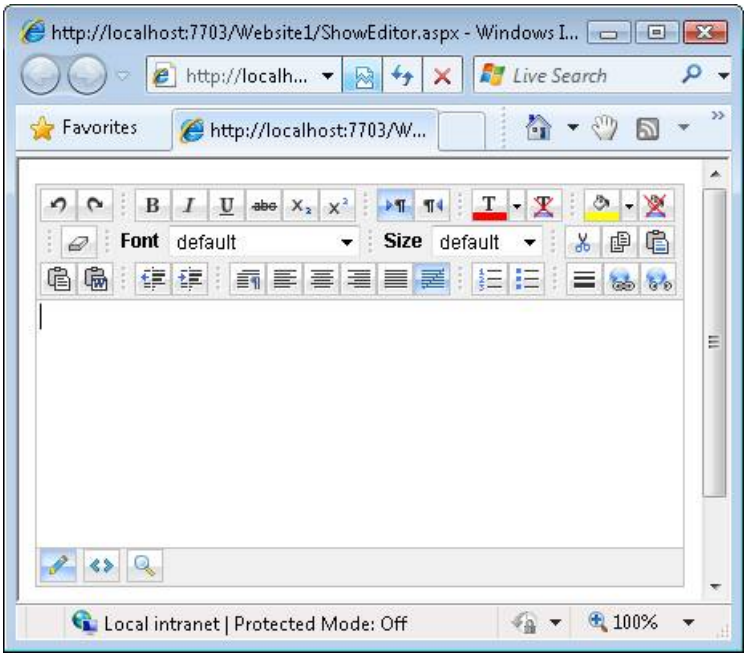


Figure 03: The DisplayEditor.aspx page(Click to view full-size image)

Using AJAX Control Toolkit Control Extenders

The AJAX Control Toolkit also contains control extenders. As its name suggests, a control extender extends the

functionality of an existing control. For example, the ConfirmButton control extender extends the standard ASP.NET Button control. The extender changes the Button control's behavior so that the Button displays a confirmation dialog when you click it.

A control extender, just like an AJAX Control Toolkit control, requires a ScriptManager control. You must add a ScriptManager control to a page before you start using control extenders in the page.

Follow these steps to use the ConfirmButton control extender:

1. Create a new ASP.NET page named ShowConfirmButton.aspx
2. Add a ScriptManager control to the page by dragging the control onto the page from beneath the AJAX Extensions tab.
3. Add a standard Button control to the page by dragging the Button from beneath the Standard tab in the toolbox onto the Designer surface.
4. Click the **Add Extender** task option (see Figure 4).
5. In the Choose Extender dialog, select ConfirmButtonExtender (see Figure 5) and click the OK button.
6. Select the Button control in the Designer and expand the Extenders, Button1_ConfirmButtonExtender node in the Properties window (see Figure 6). Assign the value *'Really?'* to the ConfirmText property.
7. Run the page by selecting the menu option **Debug, Start Debugging** or hit the F5 key.

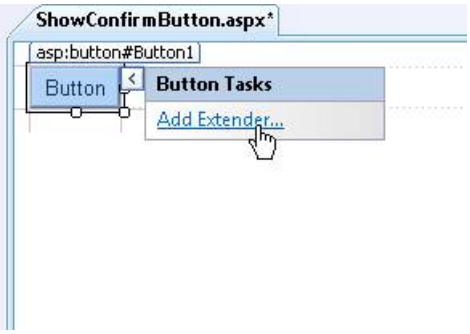


Figure 04: The Add Extender task option(Click to view full-size image)

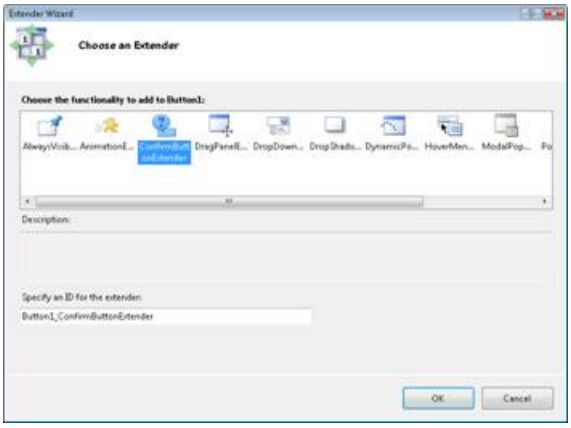


Figure 05: Selecting the ConfirmButton control extender(Click to view full-size image)

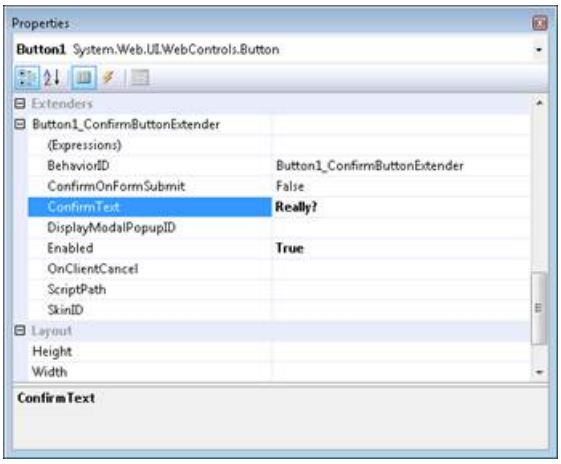


Figure 06: Setting a ConfirmButton property(Click to view full-size image)

When the page opens, you should see a button. When you click the button, you get the confirmation dialog in Figure 7.

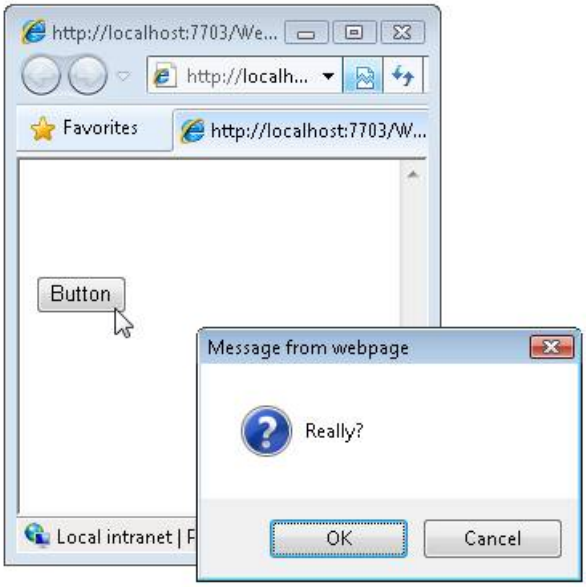


Figure 07: Displaying the confirmation dialog(Click to view full-size image)

Notice that you normally do not drag a control extender onto a page. Instead, you use the **Add Extender** task option to add an extender to a control that you have already added to a page. Notice, furthermore, that you set control extender properties by opening the property sheet for the control being extended.

A single ASP.NET control can be extended by multiple control extenders. The property sheet for the control being extended will list all of the control extenders associated with the control.

PREVIOUS NEXT

Creating a Custom AJAX Control Toolkit Control Extender (VB)

9/16/2020 • 9 minutes to read • Edit Online

by Microsoft

> Custom Extenders enable you to customize and extend the capabilities of ASP.NET controls without having to create new classes.

In this tutorial, you learn how to create a custom AJAX Control Toolkit control extender. We create a simple, but useful, new extender that changes the state of a Button from disabled to enabled when you type text into a TextBox. After reading this tutorial, you will be able to extend the ASP.NET AJAX Toolkit with your own control extenders.

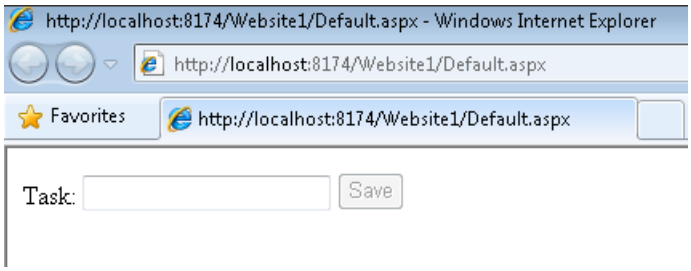
You can create custom control extenders using either Visual Studio or Visual Web Developer (make sure that you have the latest version of Visual Web Developer).

Overview of the DisabledButton Extender

Our new control extender is named the DisabledButton extender. This extender will have three properties:

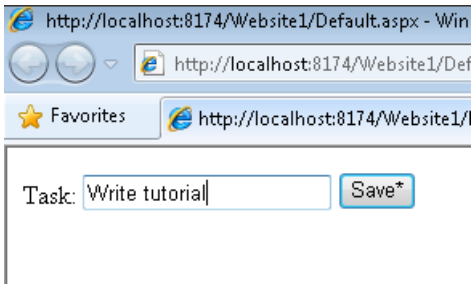
- TargetControlID - The TextBox that the control extends.
- TargetButtonID - The Button that is disabled or enabled.
- DisabledText - The text that is initially displayed in the Button. When you start typing, the Button displays the value of the Button Text property.

You hook the DisabledButton extender to a TextBox and Button control. Before you type any text, the Button is disabled and the TextBox and Button look like this:



(Click to view full-size image)

After you start typing text, the Button is enabled and the TextBox and Button look like this:



(Click to view full-size image)

To create our control extender, we need to create the following three files:

- DisabledButtonExtender.vb - This file is the server-side control class that will manage creating your extender and allow you to set the properties at design-time. It also defines the properties that can be set on your extender. These properties are accessible via code and at design time and match properties defined in the DisableButtonBehavior.js file.
- DisabledButtonBehavior.js -- This file is where you will add all of your client script logic.
- DisabledButtonDesigner.vb - This class enables design-time functionality. You need this class if you want the control extender to work correctly with the Visual Studio/Visual Web Developer Designer.

So a control extender consists of a server-side control, a client-side behavior, and a server-side designer class. You learn how to create all three of these files in the following sections.

Creating the Custom Extender Website and Project

The first step is to create a class library project and website in Visual Studio/Visual Web Developer. We'll create the custom extender in the class library project and test the custom extender in the website.

Let's start with the website. Follow these steps to create the website:

1. Select the menu option **File, New Web Site**.
2. Select the **ASP.NET Web Site** template.
3. Name the new website *Website1*.
4. Click the **OK** button.

Next, we need to create the class library project that will contain the code for the control extender:

1. Select the menu option **File, Add, New Project**.
2. Select the **Class Library** template.
3. Name the new class library with the name **CustomExtenders**.
4. Click the **OK** button.

After you complete these steps, your Solution Explorer window should look like Figure 1.

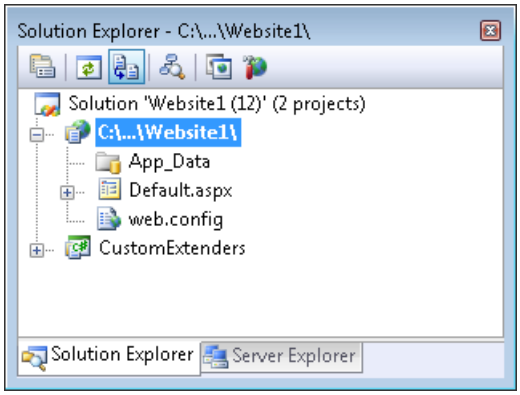


Figure 01: Solution with website and class library project(Click to view full-size image)

Next, you need to add all of the necessary assembly references to the class library project:

1. Right-click the CustomExtenders project and select the menu option **Add Reference**.
2. Select the .NET tab.
3. Add references to the following assemblies:
   a. System.Web.dll
   b. System.Web.Extensions.dll

c. System.Design.dll
d. System.Web.Extensions.Design.dll

4. Select the Browse tab.

5. Add a reference to the AjaxControlToolkit.dll assembly. This assembly is located in the folder where you downloaded the AJAX Control Toolkit.

You can verify that you have added all of the right references by right-clicking your project, selecting Properties, and clicking the References tab (see Figure 2).

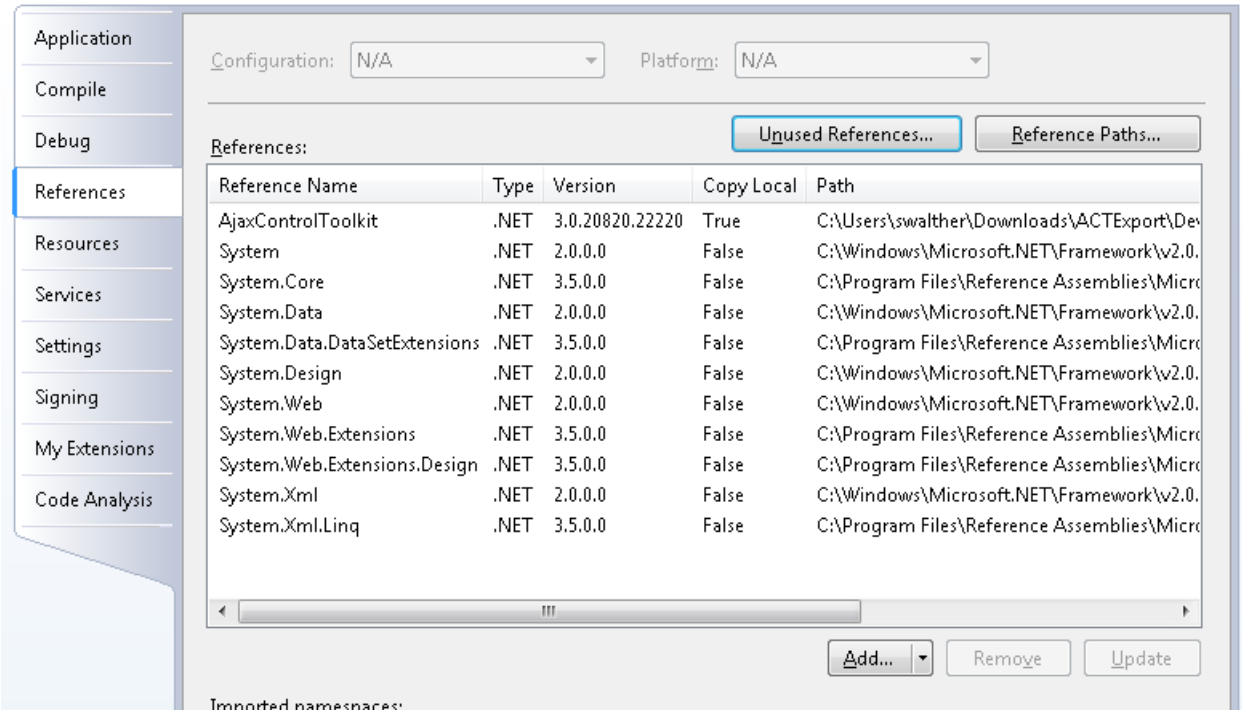


Figure 02: References folder with required references(Click to view full-size image)

Creating the Custom Control Extender

Now that we have our class library, we can start building our extender control. Let's start with the bare bones of a custom extender control class (see Listing 1).

Listing 1 - MyCustomExtender.vb

```
Imports AjaxControlToolkit
Imports System.ComponentModel
Imports System.Web.UI
Imports System.Web.UI.WebControls

<Assembly: System.Web.UI.WebResource("CustomExtenders.MyControlBehavior.js", "text/javascript")>

<ClientScriptResource("CustomExtenders.MyControlBehavior", "CustomExtenders.MyControlBehavior.js")> _
<TargetControlType(GetType(TextBox))> _
Public Class MyControlExtender
    Inherits ExtenderControlBase

    <ExtenderControlProperty()> _
    <DefaultValue("")> _
    Public Property MyProperty() As String
        Get
            Return GetPropertyValue("MyProperty", "")
        End Get
        Set(ByVal value As String)
            SetPropertyValue("MyProperty", value)
        End Set
    End Property
End Class
```

There are several things that you notice about the control extender class in Listing 1. First, notice that the class inherits from the base ExtenderControlBase class. All AJAX Control Toolkit extender controls derive from this base class. For example, the base class includes the TargetID property that is a required property of every control extender.

Next, notice that the class includes the following two attributes related to client script:

- WebResource - Causes a file to be included as an embedded resource in an assembly.
- ClientScriptResource - Causes a script resource to be retrieved from an assembly.

The WebResource attribute is used to embed the MyControlBehavior.js JavaScript file into the assembly when the custom extender is compiled. The ClientScriptResource attribute is used to retrieve the MyControlBehavior.js script from the assembly when the custom extender is used in a web page.

In order for the WebResource and ClientScriptResource attributes to work, you must compile the JavaScript file as an embedded resource. Select the file in the Solution Explorer window, open the property sheet, and assign the value *Embedded Resource* to the **Build Action** property.

Notice that the control extender also includes a TargetControlType attribute. This attribute is used to specify the type of control that is extended by the control extender. In the case of Listing 1, the control extender is used to extend a TextBox.

Finally, notice that the custom extender includes a property named MyProperty. The property is marked with the ExtenderControlProperty attribute. The GetPropertyValue() and SetPropertyValue() methods are used to pass the property value from the server-side control extender to the client-side behavior.

Let's go ahead and implement the code for our DisabledButton extender. The code for this extender can be found in Listing 2.

Listing 2 - DisabledButtonExtender.vb

```
Imports System.ComponentModel
Imports System.Web.UI
Imports System.Web.UI.WebControls
Imports AjaxControlToolkit

<Assembly: System.Web.UI.WebResource("CustomExtenders.DisabledButtonBehavior.js", "text/javascript")>

<Designer(GetType(DisabledButtonExtender))> _
<ClientScriptResource("CustomExtenders.DisabledButtonBehavior", "CustomExtenders.DisabledButtonBehavior.js")>
_
<TargetControlType(GetType(TextBox))> _
Public Class DisabledButtonExtender
    Inherits ExtenderControlBase

    <ExtenderControlProperty()> _
    <DefaultValue("")> _
    <IDReferenceProperty(GetType(Button))> _
    Public Property TargetButtonID() As String
        Get
            Return GetPropertyValue("TargetButtonID", "")
        End Get
        Set(ByVal value As String)
            SetPropertyValue("TargetButtonID", value)
        End Set
    End Property

    <ExtenderControlProperty(), DefaultValue("")> _
    Public Property DisabledText() As String
        Get
            Return GetPropertyValue("DisabledText", "")
        End Get
        Set(ByVal value As String)
            SetPropertyValue("DisabledText", value)
        End Set
    End Property

End Class
```

The DisabledButton extender in Listing 2 has two properties named TargetButtonID and DisabledText. The IDReferenceProperty applied to the TargetButtonID property prevents you from assigning anything other than the ID of a Button control to this property.

The WebResource and ClientScriptResource attributes associate a client-side behavior located in a file named DisabledButtonBehavior.js with this extender. We discuss this JavaScript file in the next section.

Creating the Custom Extender Behavior

The client-side component of a control extender is called a behavior. The actual logic for disabling and enabling the Button is contained in the DisabledButton behavior. The JavaScript code for the behavior is included in Listing 3.

Listing 3 - DisabledButton.js

```
Type.registerNamespace('CustomExtenders');

CustomExtenders.DisabledButtonBehavior = function(element) {

    CustomExtenders.DisabledButtonBehavior.initializeBase(this, [element]);

    this._targetButtonIDValue = null;
    this._disabledTextValue = null;

}

CustomExtenders.DisabledButtonBehavior.prototype = {
```

```
CustomExtenders.DisabledButtonBehavior.prototype = {

    initialize : function() {
        CustomExtenders.DisabledButtonBehavior.callBaseMethod(this, 'initialize');

        // Initalization code
        $addHandler(this.get_element(), 'keyup',
        Function.createDelegate(this, this._onkeyup));
        this._onkeyup();
    },

    dispose : function() {
        // Cleanup code

        CustomExtenders.DisabledButtonBehavior.callBaseMethod(this, 'dispose');
    },

    // Property accessors
    //
    get_TargetButtonID : function() {
        return this._targetButtonIDValue;
    },

    set_TargetButtonID : function(value) {
        this._targetButtonIDValue = value;
    },

    get_DisabledText : function() {
        return this._disabledTextValue;
    },

    set_DisabledText : function(value) {
        this._disabledTextValue = value;
    },

  _onkeyup : function() {

    var e = $get(this._targetButtonIDValue);
    if (e) {
      var disabled = ("" == this.get_element().value);
      e.disabled = disabled;
      if ( this._disabledTextValue) {
        if (disabled) {
          this._oldValue = e.value;
          e.value = this._disabledTextValue;
        }
        else
        {
          if(this._oldValue){
            e.value = this._oldValue;
          }
        }
      }
    }
  }

}

CustomExtenders.DisabledButtonBehavior.registerClass('CustomExtenders.DisabledButtonBehavior',
AjaxControlToolkit.BehaviorBase);
```

The JavaScript file in Listing 3 contains a client-side class named DisabledButtonBehavior. This class, like its server-side twin, includes two properties named TargetButtonID and DisabledText which you can access using get_TargetButtonID/set_TargetButtonID and get_DisabledText/set_DisabledText.

The initialize() method associates a keyup event handler with the target element for the behavior. Each time you type a letter into the TextBox associated with this behavior, the keyup handler executes. The keyup handler either

enables or disables the Button depending on whether the TextBox associated with the behavior contains any text.

Remember that you must compile the JavaScript file in Listing 3 as an embedded resource. Select the file in the Solution Explorer window, open the property sheet, and assign the value *Embedded Resource* to the **Build Action** property (see Figure 3). This option is available in both Visual Studio and Visual Web Developer.

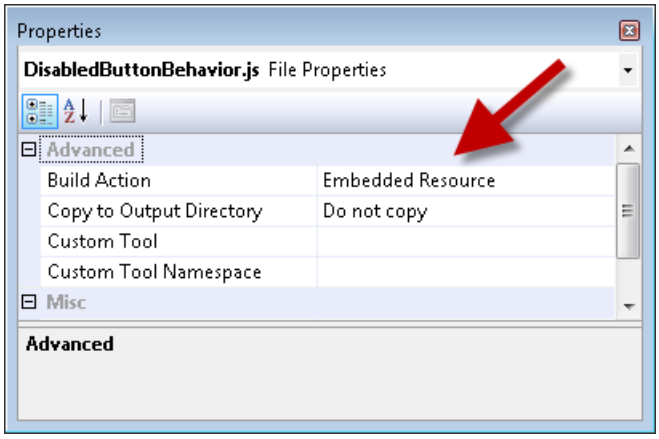


Figure 03: Adding a JavaScript file as an embedded resource(Click to view full-size image)

Creating the Custom Extender Designer

There is one last class that we need to create to complete our extender. We need to create the designer class in Listing 4. This class is required to make the extender behave correctly with the Visual Studio/Visual Web Developer Designer.

Listing 4 - DisabledButtonDesigner.vb

```
Imports AjaxControlToolkit.Design

Public Class DisabledButtonDesigner
    Inherits ExtenderControlBaseDesigner(Of DisabledButtonExtender)

End Class
```

You associate the designer in Listing 4 with the DisabledButton extender with the Designer attribute.You need to apply the Designer attribute to the DisabledButtonExtender class like this:

```
<Designer(GetType(DisabledButtonDesigner))> _
<ClientScriptResource("CustomExtenders.DisabledButtonBehavior", "CustomExtenders.DisabledButtonBehavior.js")>
_
<TargetControlType(GetType(TextBox))> _
Public Class DisabledButtonExtender
   Inherits ExtenderControlBase
```

Using the Custom Extender

Now that we have finished creating the DisabledButton control extender, it is time to use it in our ASP.NET website. First, we need to add the custom extender to the toolbox. Follow these steps:

1. Open an ASP.NET page by double-clicking the page in the Solution Explorer window.
2. Right-click the toolbox and select the menu option **Choose Items**.
3. In the Choose Toolbox Items dialog, browse to the CustomExtenders.dll assembly.
4. Click the **OK** button to close the dialog.

After you complete these steps, the DisabledButton control extender should appear in the toolbox (see Figure 4).

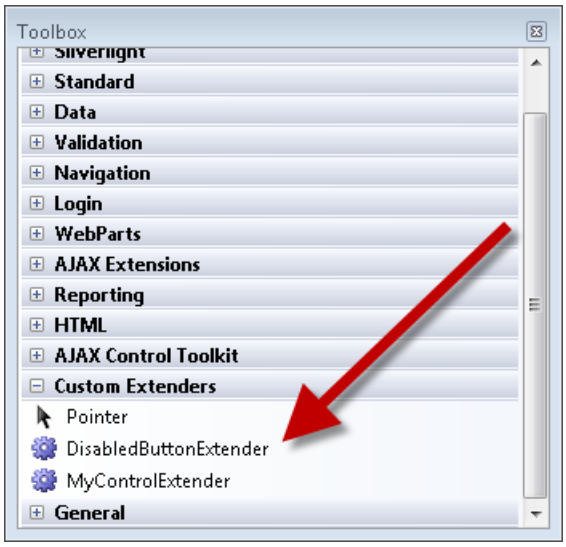


Figure 04: DisabledButton in the toolbox(Click to view full-size image)

Next, we need to create a new ASP.NET page. Follow these steps:

1. Create a new ASP.NET page named ShowDisabledButton.aspx.
2. Drag a ScriptManager onto the page.
3. Drag a TextBox control onto the page.
4. Drag a Button control onto the page.
5. In the Properties window, change the Button ID property to the value *btnSave* and the Text property to the value *Save**.

We created a page with a standard ASP.NET TextBox and Button control.

Next, we need to extend the TextBox control with the DisabledButton extender:

1. Select the **Add Extender** task option to open the Extender Wizard dialog (see Figure 5). Notice that the dialog includes our custom DisabledButton extender.
2. Select the DisabledButton extender and click the **OK** button.

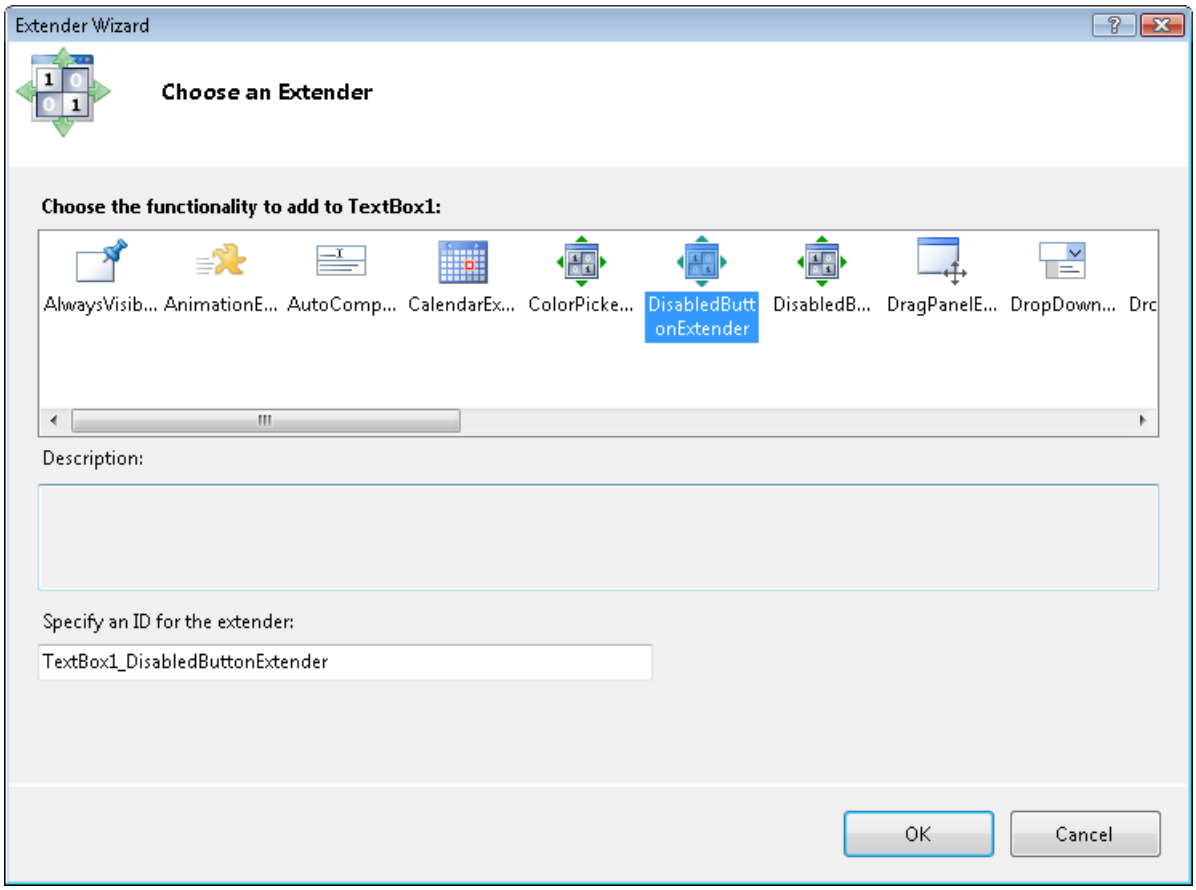


Figure 05: The Extender Wizard dialog(Click to view full-size image)

Finally, we can set the properties of the DisabledButton extender. You can modify the properties of the DisabledButton extender by modifying the properties of the TextBox control:

1. Select the TextBox in the Designer.
2. In the Properties window, expand the Extenders node (see Figure 6).
3. Assign the value *Save* to the DisabledText property and the value *btnSave* to the TargetButtonID property.

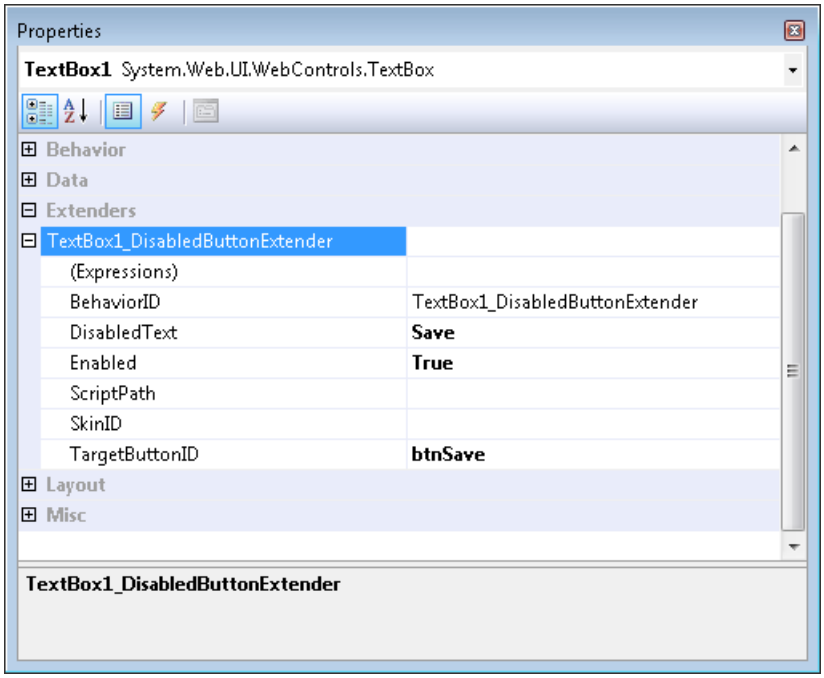


Figure 06: Setting extender properties(Click to view full-size image)

When you run the page (by hitting F5), the Button control is initially disabled. As soon as you start entering text into the TextBox, the Button control is enabled (see Figure 7).

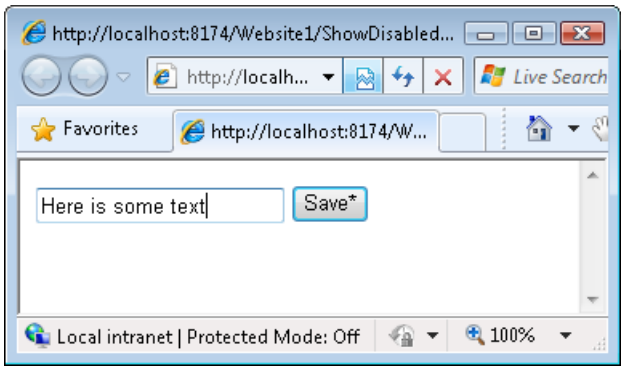


Figure 07: The DisabledButton extender in action(Click to view full-size image)

Summary

The goal of this tutorial was to explain how you can extend the AJAX Control Toolkit with custom extender controls. In this tutorial, we created a simple DisabledButton control extender. We implemented this extender by creating a DisabledButtonExtender class, a DisabledButtonBehavior JavaScript behavior, and a DisabledButtonDesigner class. You follow a similar set of steps whenever you create a custom control extender.

PREVIOUS

Accordion

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the Accordion control, which contains multiple panes and lets users display one of them at a time.

- Databinding to an Accordion (C#)
- Dynamically Adding Accordion Pane (C#)
- Databinding to an Accordion (VB)
- Dynamically Adding Accordion Pane (VB)

Databinding to an Accordion (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Accordion control in the AJAX Control Toolkit provides multiple panes and allows the user to display one of them at a time. Panels are usually declared within the page itself, but binding to a data source offers more flexibility.

Overview

The Accordion control in the AJAX Control Toolkit provides multiple panes and allows the user to display one of them at a time. Panels are usually declared within the page itself, but binding to a data source offers more flexibility.

Steps

First of all, a data source is required. This sample uses the AdventureWorks database and the Microsoft SQL Server 2005 Express Edition. The database is an optional part of a Visual Studio installation (including express edition) and is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. The AdventureWorks database is part of the SQL Server 2005 Samples and Sample Databases (download at https://www.microsoft.com/downloads/details.aspx?FamilyID=e719ecf7-9f46-4312-af89-6ad8702e4e6e&DisplayLang=en). The easiest way to set the database up is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en) and attach the `AdventureWorks.mdf` database file.

For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database.

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server"/>
```

Then, add a data source to the page. In order to use a limited amount of data, we only select the first five entries in the Vendor table of the AdventureWorks database. If you are using the Visual Studio assistant to create the data source, mind that a bug in the current version does not prefix the table name (`Vendor`) with `Purchasing`. The following markup shows the correct syntax:

```
<asp:SqlDataSource ID="sds1" runat="server" ConnectionString="Data
 Source=(local)\SQLEXPRESS;Initial Catalog=AdventureWorks;Integrated Security=True"
 ProviderName="System.Data.SqlClient" SelectCommand="SELECT TOP 5
 [VendorID], [Name] FROM [Purchasing].[Vendor]" />
```

Remember the name (ID) of the data source. This very identification must then be used in the `DataSourceID` property of the Accordion control:

```
<ajaxToolkit:Accordion ID="acc1" runat="server"
 HeaderCssClass="header" ContentCssClass="content"Width="300px"
 DataSourceID="sds1" FadeTransitions="true">
```

Within the Accordion control, you can provide templates for various parts of the control, including the header (`<HeaderTemplate>`) and the content (`<ContentTemplate>`). Within these elements, just output the data from the data source, using the `DataBinder.Eval()` method:

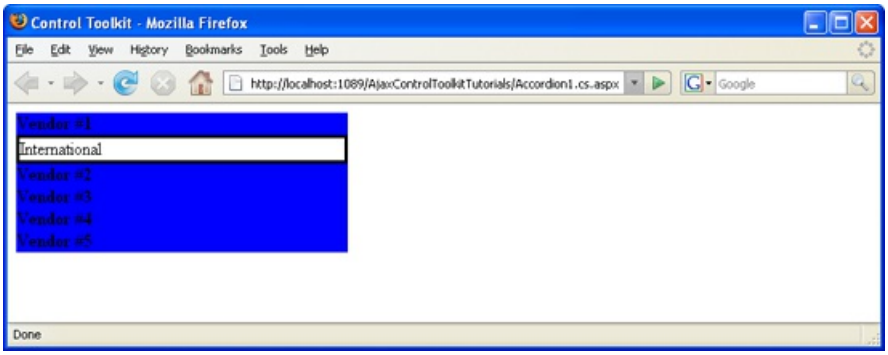
```
<HeaderTemplate><b>Vendor #<%#DataBinder.Eval(Container.DataItem, "VendorID")%></b>
 </HeaderTemplate>
 <ContentTemplate><%#DataBinder.Eval(Container.DataItem, "Name")%></ContentTemplate>
</ajaxToolkit:Accordion>
```

When the page is loaded, the data source must be bound to the accordion with this server-side code:

```
<script runat="server">
 void Page_Load()
 {
 acc1.DataBind();
 }
</script>
```

To conclude this sample, you need to define the two CSS classes that are referenced in the Accordion control (in its properties `HeaderCssClass` and `ContentCssClass`). Put the following markup in the `<head>` section of the page:

```
<style type="text/css">
 .header {background-color: blue;}
 .content {border: solid;}
</style>
```



The data in the accordion comes directly from the data source (Click to view full-size image)

NEXT

Dynamically Adding An Accordion Pane (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Accordion control in the AJAX Control Toolkit provides multiple panes and allows the user to display one of them at a time. Panels are usually declared within the page itself, but server-side code can be used to achieve the same result.

Overview

The Accordion control in the AJAX Control Toolkit provides multiple panes and allows the user to display one of them at a time. Panels are usually declared within the page itself, but server-side code can be used to achieve the same result.

Steps

The Accordion control exposes all important properties to server-side code. Among other things, the `Panes` property grants access to the collection of panes that make up the Accordion. Every pane there is of type `AccordionPane`. It is therefore trivial to create such a pane:

```
AccordionPane ap1 = new AccordionPane();
```

The `HeaderContainer` property of `AccordionPane` provides access to the ASP.NET controls within the header section of the pane; the `ContentContainer` property of `AccordionPane` does the same for the content section of the pane. This allows ASP.NET code to add content to the panes:

```
ap1.HeaderContainer.Controls.Add(new LiteralControl("Using Code"));
ap1.ContentContainer.Controls.Add(new
    LiteralControl("Adding panes using code is really flexible."));
```

Finally, the pane(s) must be added to the `Panes` collection of the Accordion:

```
acc1.Panes.Add(ap1);
```

Here is a complete server-side code that adds two panes to an Accordion control:

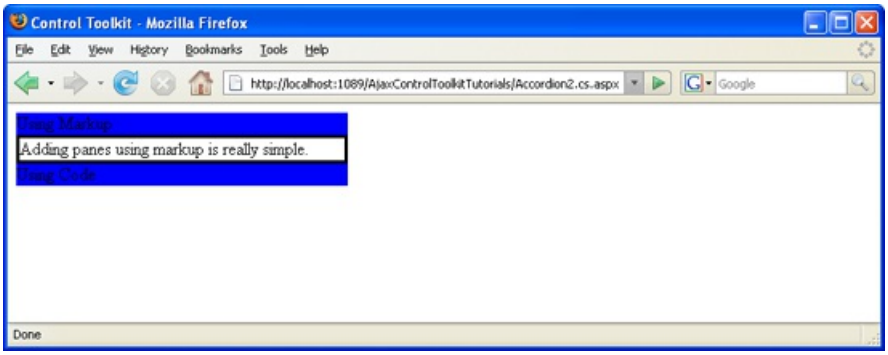
```
<script runat="server">
void Page_Load()
{
 if (!Page.IsPostBack)
 {
 AccordionPane ap1 = new AccordionPane();
 ap1.HeaderContainer.Controls.Add(new LiteralControl("Using Markup"));
 ap1.ContentContainer.Controls.Add(new
 LiteralControl("Adding panes using markup is really simple."));
 AccordionPane ap2 = new AccordionPane();
 ap2.HeaderContainer.Controls.Add(new LiteralControl("Using Code"));
 ap2.ContentContainer.Controls.Add(new
 LiteralControl("Adding panes using code is really flexible."));
 acc1.Panes.Add(ap1);
 acc1.Panes.Add(ap2);
 }
}
</script>
```

The only missing element is the Accordion itself, which depends on the presence of the ASP.NET `ScriptManager` control:

```
<form id="form1" runat="server">
 <asp:ScriptManager ID="asm" runat="server" />
 <div>
 <ajaxToolkit:Accordion ID="acc1" runat="server"
 HeaderCssClass="header" ContentCssClass="content"
 Width="300px" FadeTransitions="true">
 </ajaxToolkit:Accordion>
 </div>
</form>
```

To finish the example, the two CSS classes referenced in the Accordion control provide style information for the browser:

```
<style type="text/css">
 .header {background-color: blue;}
 .content {border: solid;}
</style>
```



The data in the accordion was dynamically added by server-side code (Click to view full-size image)

PREVIOUS NEXT

Databinding to an Accordion (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Accordion control in the AJAX Control Toolkit provides multiple panes and allows the user to display one of them at a time. Panels are usually declared within the page itself, but binding to a data source offers more flexibility.

Overview

The Accordion control in the AJAX Control Toolkit provides multiple panes and allows the user to display one of them at a time. Panels are usually declared within the page itself, but binding to a data source offers more flexibility.

Steps

First of all, a data source is required. This sample uses the AdventureWorks database and the Microsoft SQL Server 2005 Express Edition. The database is an optional part of a Visual Studio installation (including express edition) and is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. The AdventureWorks database is part of the SQL Server 2005 Samples and Sample Databases (download at https://www.microsoft.com/downloads/details.aspx?FamilyID=e719ecf7-9f46-4312-af89-6ad8702e4e6e&DisplayLang=en). The easiest way to set the database up is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en) and attach the `AdventureWorks.mdf` database file.

For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database.

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server"/>
```

Then, add a data source to the page. In order to use a limited amount of data, we only select the first five entries in the Vendor table of the AdventureWorks database. If you are using the Visual Studio assistant to create the data source, mind that a bug in the current version does not prefix the table name (`Vendor`) with `Purchasing`. The following markup shows the correct syntax:

```
<asp:SqlDataSource ID="sds1" runat="server" ConnectionString="Data
 Source=(local)\SQLEXPRESS;Initial Catalog=AdventureWorks;Integrated Security=True"
 ProviderName="System.Data.SqlClient" SelectCommand="SELECT TOP 5
 [VendorID], [Name] FROM [Purchasing].[Vendor]" />
```

Remember the name (ID) of the data source. This very identification must then be used in the `DataSourceID` property of the Accordion control:

```
<ajaxToolkit:Accordion ID="acc1" runat="server"
 HeaderCssClass="header" ContentCssClass="content"Width="300px"
 DataSourceID="sds1" FadeTransitions="true">
```

Within the Accordion control, you can provide templates for various parts of the control, including the header (`<HeaderTemplate>`) and the content (`<ContentTemplate>`). Within these elements, just output the data from the data source, using the `DataBinder.Eval()` method:

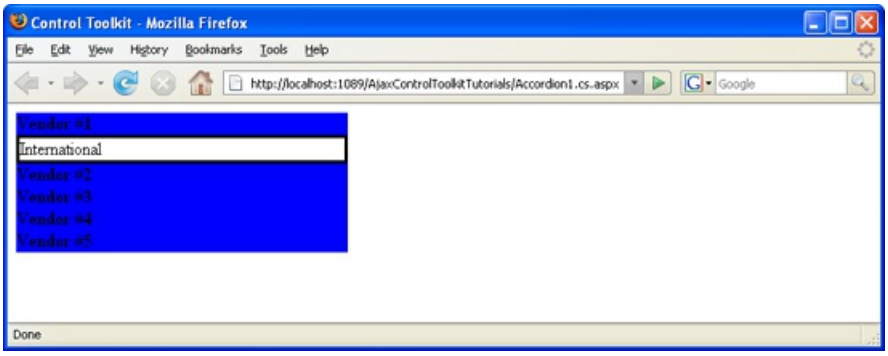
```
<HeaderTemplate><b>Vendor #<%#DataBinder.Eval(Container.DataItem, "VendorID")%></b>
 </HeaderTemplate>
 <ContentTemplate><%#DataBinder.Eval(Container.DataItem, "Name")%></ContentTemplate>
</ajaxToolkit:Accordion>
```

When the page is loaded, the data source must be bound to the accordion with this server-side code:

```
<script runat="server">
 Sub Page_Load()
 acc1.DataBind()
 End Sub
</script>
```

To conclude this sample, you need to define the two CSS classes that are referenced in the Accordion control (in its properties `HeaderCssClass` and `ContentCssClass`). Put the following markup in the `<head>` section of the page:

```
<style type="text/css">
 .header {background-color: blue;}
 .content {border: solid;}
</style>
```



The data in the accordion comes directly from the data source (Click to view full-size image)

PREVIOUS NEXT

Dynamically Adding An Accordion Pane (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Accordion control in the AJAX Control Toolkit provides multiple panes and allows the user to display one of them at a time. Panels are usually declared within the page itself, but server-side code can be used to achieve the same result.

Overview

The Accordion control in the AJAX Control Toolkit provides multiple panes and allows the user to display one of them at a time. Panels are usually declared within the page itself, but server-side code can be used to achieve the same result.

Steps

The Accordion control exposes all important properties to server-side code. Among other things, the `Panes` property grants access to the collection of panes that make up the Accordion. Every pane there is of type `AccordionPane`. It is therefore trivial to create such a pane:

```
Dim ap1 As New AccordionPane()
```

The `HeaderContainer` property of `AccordionPane` provides access to the ASP.NET controls within the header section of the pane; the `ContentContainer` property of `AccordionPane` does the same for the content section of the pane. This allows ASP.NET code to add content to the panes:

```
ap1.HeaderContainer.Controls.Add(New LiteralControl("Using Code"))
ap1.ContentContainer.Controls.Add(New LiteralControl("Adding panes using code is really flexible."))
```

Finally, the pane(s) must be added to the `Panes` collection of the Accordion:

```
acc1.Panes.Add(ap1)
```

Here is a complete server-side code that adds two panes to an Accordion control:

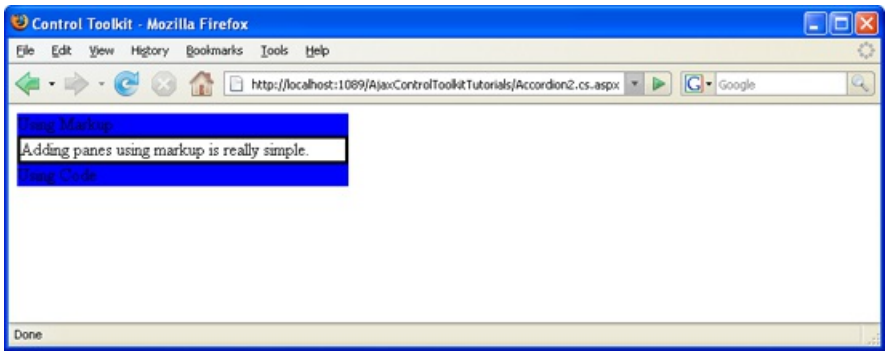
```
<script runat="server">
Sub Page_Load()
 If Not Page.IsPostBack Then
 Dim ap1 As New AccordionPane()
 ap1.HeaderContainer.Controls.Add(New LiteralControl("Using Markup"))
 ap1.ContentContainer.Controls.Add(New LiteralControl("Adding panes using markup is really simple."))
 Dim ap2 As New AccordionPane()
 ap2.HeaderContainer.Controls.Add(New LiteralControl("Using Code"))
 ap2.ContentContainer.Controls.Add(New LiteralControl("Adding panes using code is really flexible."))
 acc1.Panes.Add(ap1)
 acc1.Panes.Add(ap2)
 End If
End Sub
</script>
```

The only missing element is the Accordion itself, which depends on the presence of the ASP.NET `ScriptManager` control:

```
<form id="form1" runat="server">
 <asp:ScriptManager ID="asm" runat="server" />
 <div>
 <ajaxToolkit:Accordion ID="acc1" runat="server"
 HeaderCssClass="header" ContentCssClass="content" Width="300px" FadeTransitions="true">
 </ajaxToolkit:Accordion>
 </div>
</form>
```

To finish the example, the two CSS classes referenced in the Accordion control provide style information for the browser:

```
<style type="text/css">
 .header {background-color: blue;}
 .content {border: solid;}
</style>
```



The data in the accordion was dynamically added by server-side code (Click to view full-size image)

PREVIOUS

Animation

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to create animations using the Animation control and animation framework.

- Adding Animation to a Control (C#)
- Executing Several Animations at Same Time (C#)
- Executing Several Animations after Each Other (C#)
- Animation Depending On a Condition (C#)
- Picking One Animation Out Of a List (C#)
- Animating in Response To User Interaction (C#)
- Disabling Actions during Animation (C#)
- Triggering an Animation in another Control (C#)
- Modifying Animations From Server Side (C#)
- Executing Animations Using Client-Side Code (C#)
- Changing an Animation Using Client-Side Code (C#)
- Animating an UpdatePanel Control (C#)
- Dynamically Controlling UpdatePanel Animations (C#)
- Adding Animation to a Control (VB)
- Executing Several Animations at Same Time (VB)
- Executing Several Animations after Each Other (VB)
- Animation Depending On a Condition (VB)
- Picking One Animation Out Of a List (VB)
- Animating in Response To User Interaction (VB)
- Disabling Actions during Animation (VB)
- Triggering an Animation in another Control (VB)
- Modifying Animations From Server Side (VB)
- Executing Animations Using Client-Side Code (VB)
- Changing an Animation Using Client-Side Code (VB)
- Animating an UpdatePanel Control (VB)
- Dynamically Controlling UpdatePanel Animations (VB)

Adding Animation to a Control (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. This tutorial shows how to set up such an animation.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. This tutorial shows how to set up such an animation.

Steps

The first step is as usual to include the `ScriptManager` in the page so that the ASP.NET AJAX library is loaded and the Control Toolkit can be used:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation in this scenario will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more efficient,
 more interactive and highly-personalized Web experiences that work across all the
 most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more efficient,
 more interactive and highly-personalized Web experiences that work across all the
 most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more efficient,
 more interactive and highly-personalized Web experiences that work across all the
 most popular browsers.<br />
</asp:Panel>
```

The associated CSS class for the panel defines a background color and a width:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

Next up, we need the `AnimationExtender`. After providing an `ID` and the usual `runat="server"`, the `TargetControlID` attribute must be set to the control to animate in our case, the panel:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

The whole animation is applied declaratively, using an XML syntax, unfortunately currently not fully supported by Visual Studio's IntelliSense. The root node is `<Animations>;` within this node, several events are allowed which determine when the animation(s) take(s) place:

- `OnClick` (mouse click)
- `OnHoverOut` (when the mouse leaves a control)
- `OnHoverOver` (when the mouse hovers over a control, stopping the `OnHoverOut` animation)
- `OnLoad` (when the page has been loaded)
- `OnMouseOut` (when the mouse leaves a control)
- `OnMouseOver` (when the mouse hovers over a control, not stopping the `OnMouseOut` animation)

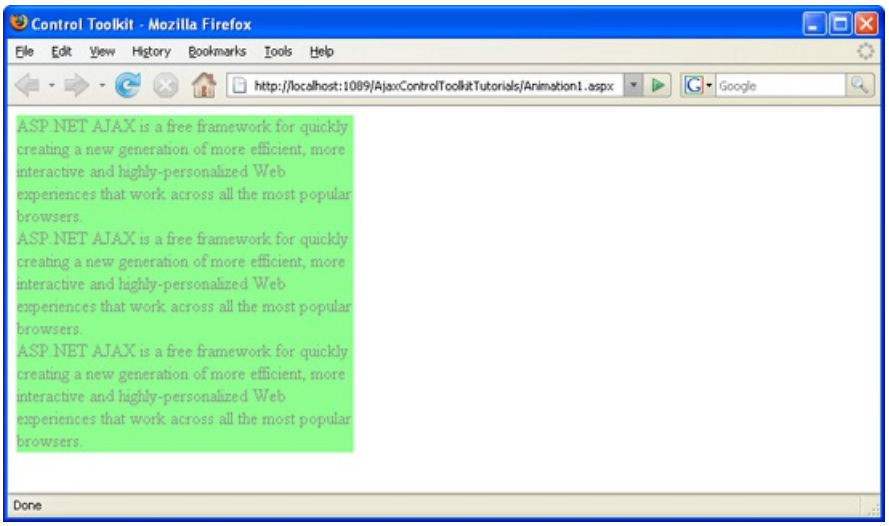
The framework comes with a set of animations, each one represented by its own XML element. Here is a selection:

- `<Color>` (changing a color)
- `<FadeIn>` (fading in)
- `<FadeOut>` (fading out)
- `<Property>` (changing a control's property)
- `<Pulse>` (pulsating)
- `<Resize>` (changing the size)
- `<Scale>` (proportionally changing the size)

In this example, the panel shall fade out. The animation shall take 1.5 seconds (`Duration` attribute), displaying 24 frames (animation steps) per second (`Fps` attribute). Here is the complete markup for the `AnimationExtender` control:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
 <Animations>
 <OnLoad>
 <FadeOut Duration="1.5" Fps="24" />
 </OnLoad>
 </Animations>
</ajaxToolkit:AnimationExtender>
```

When you run this script, the panel is displayed and fades out in one and a half seconds.



The panel is fading out (Click to view full-size image)

NEXT

Executing Several Animations at The Same Time (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. It allows to run several animations in a parallel fashion.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. It allows to run several animations in a parallel fashion.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more efficient,
 more interactive and highly-personalized Web experiences that work across all the
 most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more efficient,
 more interactive and highly-personalized Web experiences that work across all the
 most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more efficient,
 more interactive and highly-personalized Web experiences that work across all the
 most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server"`:

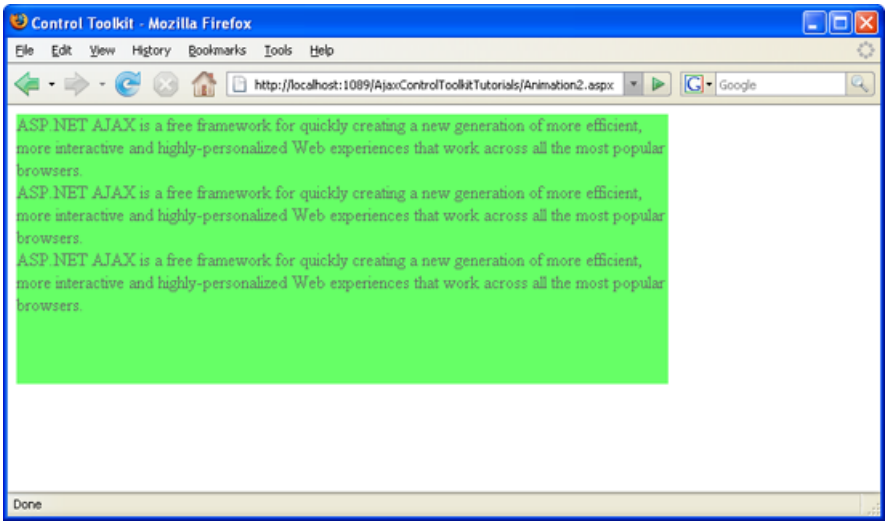
```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

Within the `<Animations>` node, use `<OnLoad>` to run the animations once the page has been fully loaded. Generally, `<OnLoad>` only accepts one animation. The Animation framework allows you to join several animations into one using the `<Parallel>` element. All animations within `<Parallel>` are executed at the same time.

Here is the a possible markup for the `AnimationExtender` control, fading out and resizing the panel at the same time:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
 <Animations>
 <OnLoad>
 <Parallel>
 <FadeOut Duration="1.5" Fps="24" />
 <Resize Width="1000" Height="150" Unit="px" />
 </Parallel>
 </OnLoad>
 </Animations>
</ajaxToolkit:AnimationExtender>
```

And indeed: when you run this script, the panel is displayed, then resizes (more than tripling its width and halving its height) and fades out at the same time.



The panel is fading out and resizing (including its content, thanks to the browser's rendering engine) (Click to view full-size image)

PREVIOUS NEXT

Executing Several Animations after Each Other (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. It allows to run several animations one after the other.

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. It allows to run several animations one after the other.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

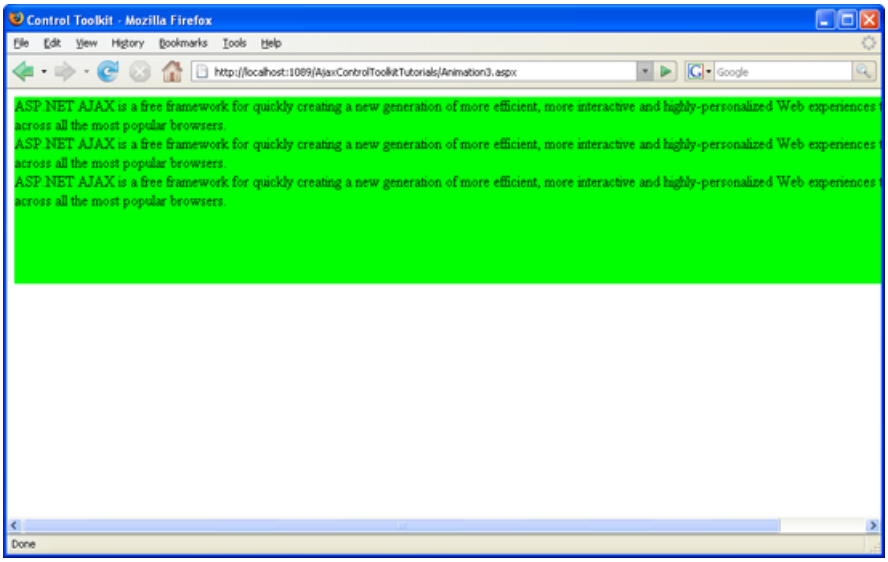
Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server":`

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

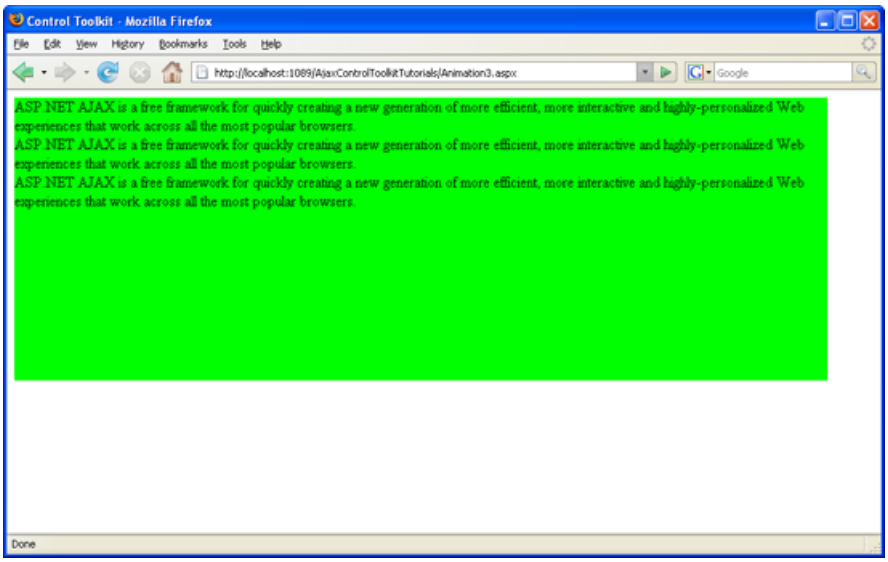
Within the `<Animations>` node, use `<OnLoad>` to run the animations once the page has been fully loaded. Generally, `<OnLoad>` only accepts one animation. The Animation framework allows you to join several animations into one using the `<Sequence>` element. All animations within `<Sequence>` are executed one after the other. Here is the a possible markup for the `AnimationExtender` control, first making the panel wider and then decreasing its height:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
 <Animations>
 <OnLoad>
 <Sequence>
 <Resize Width="1000" Unit="px" />
 <Resize Height="150" Unit="px" />
 </Sequence>
 </OnLoad>
 </Animations>
</ajaxToolkit:AnimationExtender>
```

When you run this script, the panel first gets wider and then smaller.



First the width is increased (Click to view full-size image)



Then the height is decreased (Click to view full-size image)

PREVIOUS NEXT

Animation Depending On a Condition (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. Whether an animation is run or not can also depend on a condition in form of some JavaScript code.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. Whether an animation is run or not can also depend on a condition in form of some JavaScript code.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server":`

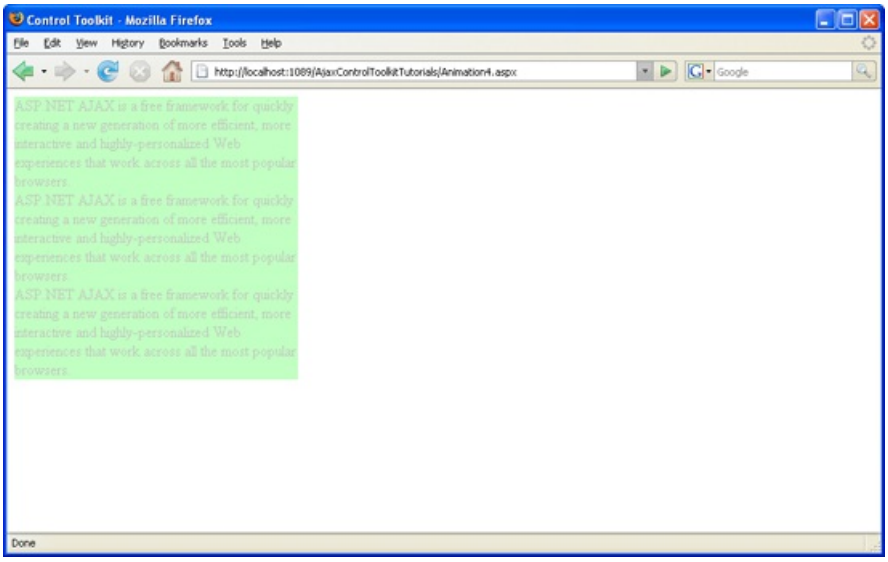
```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

Within the `<Animations>` node, use `<OnLoad>` to run the animations once the page has been fully loaded. Instead

of one of the regular animations, the `<Condition>` element comes into play. The JavaScript code provided as the value of the `ConditionScript` attribute is executed at runtime. If it evaluates to true, the animation is executed, otherwise not. The following markup provides two animations, each of them being executed in 50% of cases upon random. Since there may only be one animation within `<OnLoad>`, the two `<Condition>` animations are joined together using the `<Sequence>` element:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server"
 TargetControlID="Panel1">
  <Animations>
    <OnLoad>
      <Sequence>
        <Condition ConditionScript="Math.random() &lt; 0.5">
          <Resize Width="1000" Height="150" Unit="px" />
        </Condition>
        <Condition ConditionScript="Math.random() &lt; 0.5">
          <FadeOut Duration="1.5" Fps="24" />
        </Condition>
      </Sequence>
    </OnLoad>
  </Animations>
</ajaxToolkit:AnimationExtender>
```

Note that the less than sign (`<`) in the `ConditionScript` attribute must be escaped (). When you run this script, either no animation runs, or one of the two does, or both do.



The panel is fading out without resizing, so the second animation runs, the first one didn't (Click to view full-size image)

PREVIOUS NEXT

Picking One Animation Out Of a List (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The framework also allows the programmer to pick one animation out of a list of animations, depending on the evaluation of some JavaScript code.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The framework also allows the programmer to pick one animation out of a list of animations, depending on the evaluation of some JavaScript code.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server":`

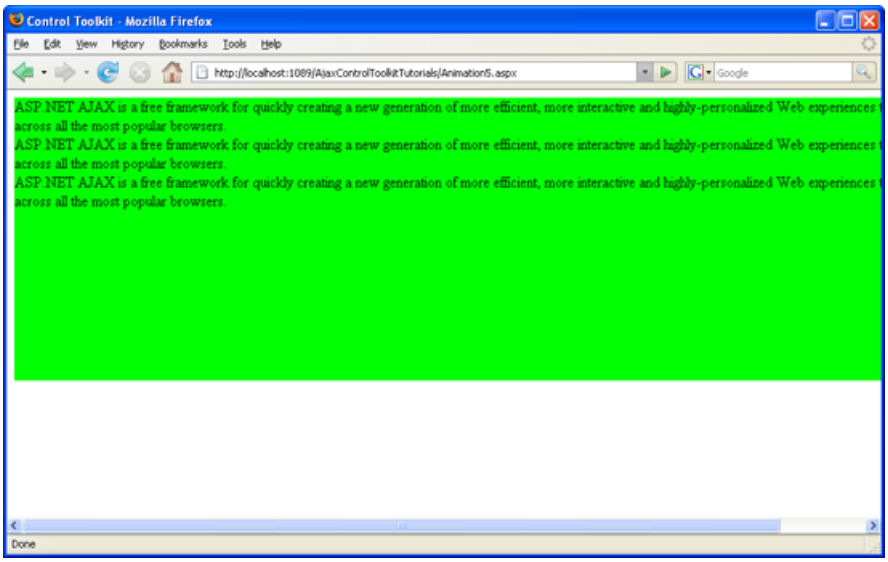
```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

Within the `<Animations>` node, use `<OnLoad>` to run the animations once the page has been fully loaded. Instead

of one of the regular animations, the `<Case>` element comes into play. The value of its SelectScript attribute is evaluated; the return value must be numerical. Depending on this number, one of the subanimations within <Case> is executed. For instance, if SelectScript evaluates to 2, the Control Toolkit runs the third animation within <Case> (counting starts at 0).

The following markup defines three subanimations: Resizing the width, resizing the height, and fading out. The JavaScript code (`Math.floor(3 * Math.random())`) then picks a number between 0 and 2, so that one of the three animations is run:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server"
 TargetControlID="Panel1">
 <Animations>
 <OnLoad>
 <Case SelectScript="Math.floor(3 * Math.random())">
 <Resize Width="1000" Unit="px" />
 <Resize Height="150" Unit="px" />
 <FadeOut Duration="1.5" Fps="24" />
 </Case>
 </OnLoad>
 </Animations>
</ajaxToolkit:AnimationExtender>
```



One of the possible three animations: The panel gets wider (Click to view full-size image)

PREVIOUS NEXT

Animating in Response To User Interaction (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animations can start automatically or may be triggered by user interaction, e.g. by clicking with the mouse.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animations can start automatically or may be triggered by user interaction, e.g. by clicking with the mouse.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server"`:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

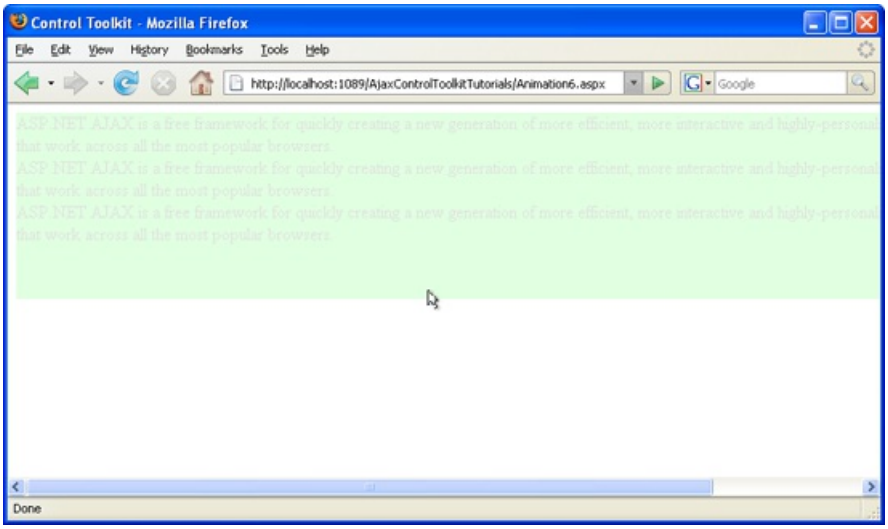
Within the `<Animations>` node, there are five ways to start the animation via user interaction (the missing element

is `<OnLoad>` which is executed once the whole page has been fully loaded):

- `<OnClick>` (mouse click on the control)
- `<OnHoverOut>` (mouse leaves the control)
- `<OnHoverOver>` (mouse hovers over a control, stopping the `<OnHoverOut>` animation)
- `<OnMouseOut>` (mouse leaves a control)
- `<OnMouseOver>` (mouse hovers over a control, not stopping the `<OnMouseOut>` animation)

In this scenario, `<OnClick>` is used. When the user clicks on the panel, it is resized and fades out at the same time.

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
 <Animations>
 <OnClick>
 <Parallel>
 <FadeOut Duration="1.5" Fps="24" />
 <Resize Width="1000" Height="150" Unit="px" />
 </Parallel>
 </OnClick>
 </Animations>
</ajaxToolkit:AnimationExtender>
```



A mouse click starts the animation (Click to view full-size image)

PREVIOUS NEXT

Disabling Actions during Animation (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. It also supports actions, like mouse clicks. However when a mouse click starts an animation, it is desirable to disable mouse clicks during the animation.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. It also supports actions, like mouse clicks. However when a mouse click starts an animation, it is desirable to disable mouse clicks during the animation.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to an HTML button like this:

```
<input type="button" ID="Button1" runat="server" Value="Launch Animation" />
```

Note that an HTML Control is used instead of a Web Control since we do not want the button to create a postback; it shall just launch the client-side animation for us.

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server"`:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Button1">
```

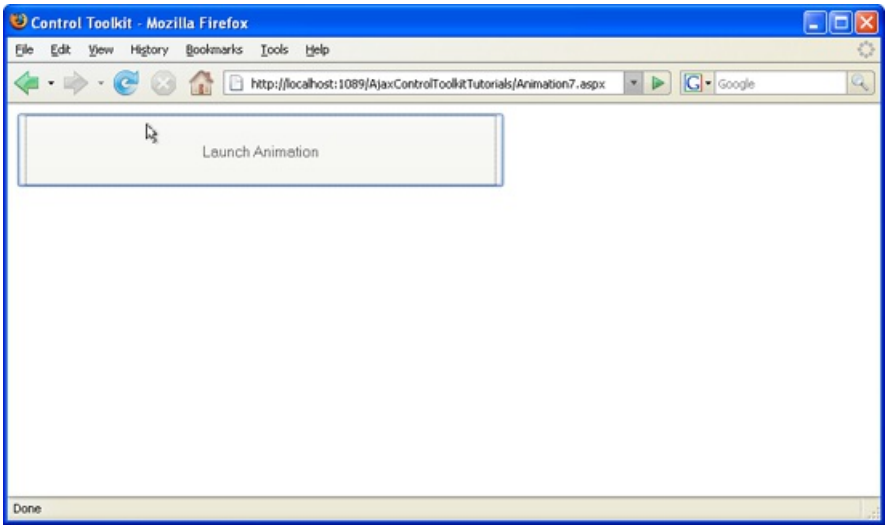
Within the `<Animations>` node, `<OnClick>` is the right element to handle the mouse click. However, the button could be clicked during the animation, as well. The `<EnableAction>` element can take care of that. Setting `Enabled="false"` disables the button as part of the animation. Since we are using several individual animations (disabling the button and the actual animations), the `<Parallel>` element is required to glue the single animations together into one. Here is the complete markup for `AnimationExtender`:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Button1">
 <Animations>
 <OnClick>
 <Parallel>
 <EnableAction Enabled="false" />
 <FadeOut Duration="1.5" Fps="24" />
 <Resize Width="1000" Height="150" Unit="px" />
 </Parallel>
 </OnClick>
 </Animations>
</ajaxToolkit:AnimationExtender>
```

It would also be possible to re-enable to button after the animation, using the following XML element at the end of the list:

```
<EnableAction Enabled="true" />
```

However in the given scenario this would be useless since the button fades out and is not visible at the end of the animation.



The button is disabled as soon as the animation runs (Click to view full-size image)

PREVIOUS NEXT

Triggering an Animation in another Control (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. Generally, launching an animation is triggered by user interaction with the same control. It is however also possible to interact with one control and then animation another control.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. Generally, launching an animation is triggered by user interaction with the same control. It is however also possible to interact with one control and then animation another control.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

In order to start animating the panel, an HTML button is used. Note that `<input type="button" />` is favoured over `<asp:Button />` since we do not want a postback when the user clicks on that button.

```
<input type="button" id="Button1" runat="server" Value="Launch Animation" />
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory

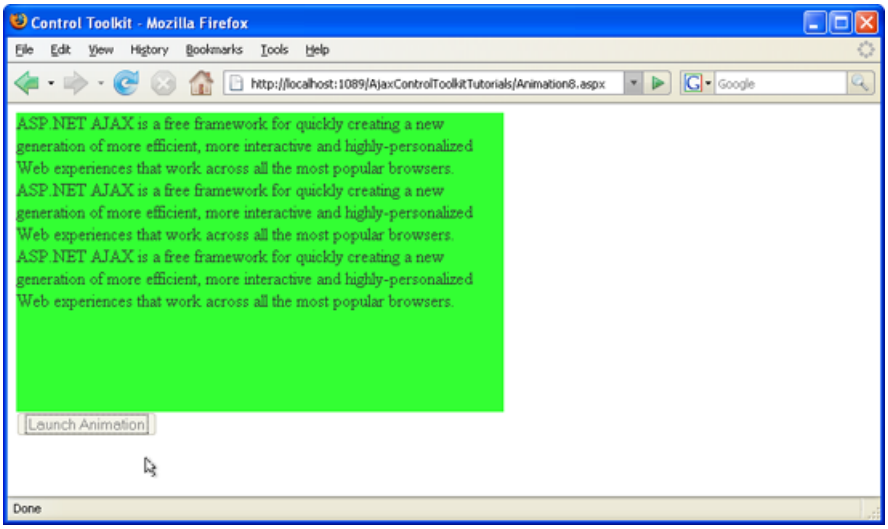
`runat="server"`. It is important to set `TargetControlID` to the ID of the button (the element triggering the animation), not to the ID of the panel (the element being animated)

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Button1">
```

Within the `<Animations>` node, place animations as usual. In order to make them change the panel, not the button, set the `AnimationTarget` attribute for every animation element within `AnimationExtender`. The value for `AnimationTarget` is the ID of the panel, of course. That way, the animations happen with the panel, not with the triggering button. Here is the `AnimationExtender` markup for this scenario:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
 <Animations>
 <OnClick>
 <Sequence>
 <EnableAction Enabled="false" />
 <Parallel>
 <FadeOut Duration="1.5" Fps="24" AnimationTarget="Panel1" />
 <Resize Width="1000" Height="150" Unit="px"
 AnimationTarget="Panel1" />
 </Parallel>
 </Sequence>
 </OnClick>
 </Animations>
</ajaxToolkit:AnimationExtender>
```

Note the special order in which the individual animations appear. First of all, the button gets deactivated once the animation runs. Since there is no `AnimationTarget` attribute in the `<EnableAction>` element, this animation is applied to the originating control: the button. The next two animation steps shall be carried out in parallel (`<Parallel>` element). Both have their `AnimationTarget` attributes set to `"Panel1"`, thus animating the panel, not the button.



A mouse click on the button starts the panel animation (Click to view full-size image)

PREVIOUS NEXT

Modifying Animations From The Server Side (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animations may also be changed on the server-side

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animations may also be changed on the server-side

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

The rest of the code runs on the server-side and does not use markup; instead, it uses code to create the `AnimationExtender` control:

```
<script runat="server">
void Page_Load()
{
 AjaxControlToolkit.AnimationExtender ae = new AjaxControlToolkit.AnimationExtender();
 ae.TargetControlID = "Panel1";
```

However, the Control Toolkit currently does not provide an API access to create the individual animations. It is however possible to set the `AnimationExtender`'s Animations property to a string containing the XML markup used when assigning the animations declaratively. In order to create the XML which must not contain the `<Animations>` element you could use the .NET Framework's XML support or, as in the following code, just provide the string:

```
ae.Animations = "<OnLoad><Parallel><FadeOut Duration=\"1.5\"
 Fps=\"24\" /><Resize Width=\"1000\" Height=\"150\"
 Unit=\"px\" /></Parallel></OnLoad>";
```

Finally, add the `AnimationExtender` control to the current page, within the `<form runat="server">` element, making sure that the animation is included and runs:

```
form1.Controls.Add(ae);
}
</script>
```

The animation is created using server-side C#/VB code (Click to view full-size image)

PREVIOUS NEXT

Executing Animations Using Client-Side Code (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animation execution may also be triggered using custom client-side JavaScript code.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animation execution may also be triggered using custom client-side JavaScript code.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server"`:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

Within the `<Animations>` node, use `<OnClick>` to run the animations once the user clicks on the panel. Add two animations to be executed in parallel:

```
<Animations>
 <OnClick>
 <Parallel>
 <FadeOut Duration="1.5" Fps="24" />
 <Resize Width="1000" Height="150" Unit="px" />
 </Parallel>
 </OnClick>
 </Animations>
</ajaxToolkit:AnimationExtender>
```

For the sake of demonstration, this animation (and any other animation created using the Control Toolkit) is executed using JavaScript code, once the page runs. First of all we need access to the `AnimationExtender` control. The ASP.NET AJAX library provides the `$find()` function for this task:

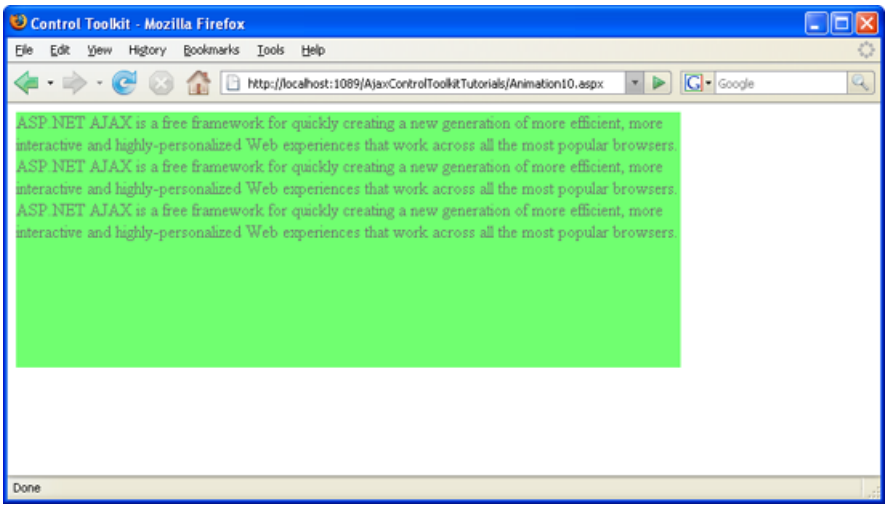
```
var ae = $find("ae");
```

The `AnimationExtender` control exposes a rich API, including methods with names identical to the event handlers used in the XML markup: `OnClick()`, `OnLoad()`, and so on. For instance, a call of the `OnClick()` method executes the animation within the `<OnClick>` element of the `AnimationExtender` control:

```
ae.OnClick();
```

Here is the complete client-side JavaScript code that emulates the click on the panel once the page has been fully loaded note that the `pageLoad()` function name is used which is called by ASP.NET AJAX once the page and all included JavaScript libraries have been loaded.

```
<script type="text/javascript">
 function pageLoad() {
 var ae = $find("ae");
 ae.OnClick();
 }
</script>
```



The animation runs immediately, without a mouse click (Click to view full-size image)

PREVIOUS NEXT

Changing an Animation Using Client-Side Code (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animation can also be changed using custom client-side JavaScript code.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animation can also be changed using custom client-side JavaScript code.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server"/>
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of
 more efficient, more interactive and highly-personalized Web experiences
 that work across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of
 more efficient, more interactive and highly-personalized Web experiences
 that work across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of
 more efficient, more interactive and highly-personalized Web experiences
 that work across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

The actual animation is launched by an HTML button:

```
<input type="button" id="Button1" runat="server" value="Launch Animation" />
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server"`:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Button1" />
```

Note that there is no `<Animations>` node within the `AnimationExtender` control. Custom JavaScript code is used to provide the animations to be used with the control.

As with the server API of `AnimationExtender`, there is no easy way to assign an animation to the extender yet. However the extender does expose several methods to read and write animations registered with the various events (`OnClick`, `OnLoad`, and so on). Here are some examples:

- `get_OnClick()`
- `set_OnClick()`
- `get_OnLoad()`
- `set_OnLoad()`
- `...`

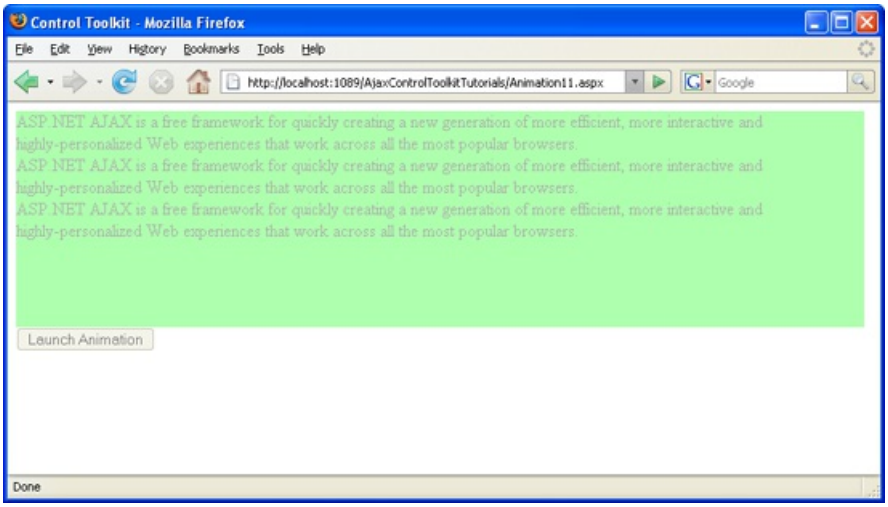
The format of the return value of the `get_*()` functions and the format of the argument for the `set_*()` functions is a JSON string, providing an object representation of what the XML markup would be. Currently, there is no way to pass an object in, but it is possible to read an object from a given animation (`get_OnXXXBehavior()` methods).

Here is a JSON string (without the delimiting quotes and formatted nicely) representing an animation triggered by the button, but animating the panel by resizing it and fading it out at the same time:

```
{
 "AnimationName":"Sequence",
 "AnimationChildren":[
 {
 "AnimationName":"EnableAction",
 "Enabled":"false",
 "AnimationChildren":[]
 },
 {
 "AnimationName":"Parallel",
 "AnimationChildren":[
 {
 "AnimationName":"FadeOut",
 "Duration":"1.5",
 "Fps":"24",
 "AnimationTarget":"Panel1",
 "AnimationChildren":[]
 },
 {
 "AnimationName":"Resize",
 "Width":"1000",
 "Height":"150",
 "Unit":"px",
 "AnimationTarget":"Panel1",
 "AnimationChildren":[]
 }]
 }]
}
```

The following JavaScript code assigns this JSON descripting to the `OnClick` animation of the current extender and runs it:

```
<script type="text/javascript">
 function pageLoad()
 {
 var ae = $find("ae");
 var animation = '{"AnimationName":"Sequence","AnimationChildren":
[{"AnimationName":"EnableAction","Enabled":"false","AnimationChildren":[]},
{"AnimationName":"Parallel","AnimationChildren":
[{"AnimationName":"FadeOut","Duration":"1.5","Fps":"24","AnimationTarget":"Panel1","AnimationChildren":[]},
{"AnimationName":"Resize","Width":"1000","Height":"150","Unit":"px","AnimationTarget":"Panel1","AnimationChil
dren":[]}]}]}';
 ae.set_OnClick(animation);
 ae.OnClick();
 }
</script>
```



The animation runs immediately, without a mouse click (and with very little markup) (Click to view full-size image)

PREVIOUS NEXT

Animating an UpdatePanel Control (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. For the contents of an UpdatePanel, a special extender exists that relies heavily on the animation framework: UpdatePanelAnimation. This tutorial shows how to set up such an animation for an UpdatePanel.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. For the contents of an `UpdatePanel`, a special extender exists that relies heavily on the animation framework: `UpdatePanelAnimation`. This tutorial shows how to set up such an animation for an `UpdatePanel`.

Steps

The first step is as usual to include the `ScriptManager` in the page so that the ASP.NET AJAX library is loaded and the Control Toolkit can be used:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation in this scenario will be applied to an ASP.NET `Wizard` web control residing in an `UpdatePanel`. Three (arbitrary) steps provide enough options to trigger postbacks:

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
 <ContentTemplate>
 <asp:Wizard ID="Wizard1" runat="server">
 <WizardSteps>
 <asp:WizardStep runat="server" StepType="Start" Title="Ready!">
 </asp:WizardStep>
 <asp:WizardStep runat="server" StepType="Step" Title="Set!">
 </asp:WizardStep>
 <asp:WizardStep runat="server" StepType="Finish" Title="Go!">
 </asp:WizardStep>
 </WizardSteps>
 </asp:Wizard>
 </ContentTemplate>
</asp:UpdatePanel>
```

The markup necessary for the `UpdatePanelAnimationExtender` control is quite similar to the markup used for the `AnimationExtender`. In the `TargetControlID` attribute we provide the `ID` of the `UpdatePanel` to animate; within the `UpdatePanelAnimationExtender` control, the `<Animations>` element holds XML markup for the animation(s). However there is one difference: The amount of events and event handlers is limited in comparison to `AnimationExtender`. For `UpdatePanels`, only two of them exist:

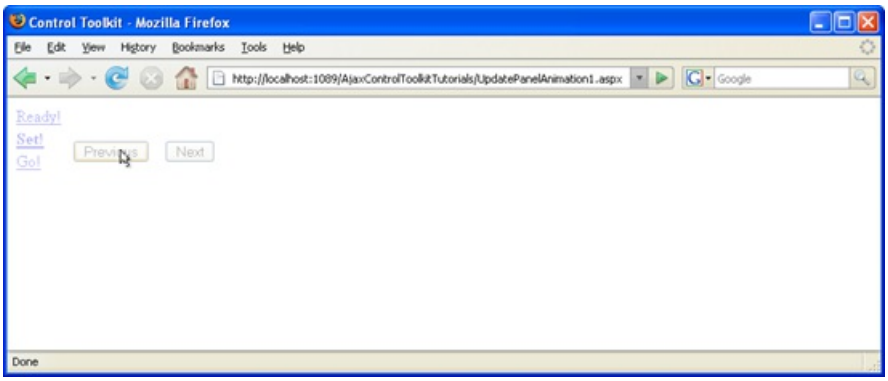
- `<OnUpdated>` when the UpdatePanel has been updated

- `<OnUpdating>` when the UpdatePanel starts updating

In this scenario, the new content of the `UpdatePanel` (after the postback) shall fade in. This is the necessary markup for that:

```
<ajaxToolkit:UpdatePanelAnimationExtender ID="upae" runat="server"
 TargetControlID="UpdatePanel1">
 <Animations>
 <OnUpdated>
 <FadeIn Duration="1.0" Fps="24" />
 </OnUpdated>
 </Animations>
</ajaxToolkit:UpdatePanelAnimationExtender>
```

Now whenever a postback occurs within the UpdatePanel, the new contents of the panel fade in smoothly.



The next wizard step is fading in (Click to view full-size image)

PREVIOUS NEXT

Dynamically Controlling UpdatePanel Animations (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. For the contents of an UpdatePanel, a special extender exists that relies heavily on the animation framework: UpdatePanelAnimation. It can also work together with UpdatePanel triggers.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. For the contents of an `UpdatePanel`, a special extender exists that relies heavily on the animation framework: `UpdatePanelAnimation`. It can also work together with `UpdatePanel` triggers.

Steps

The first step is as usual to include the `ScriptManager` in the page so that the ASP.NET AJAX library is loaded and the Control Toolkit can be used:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation in this scenario will be applied to a display of the current time. This information can be written into a label using the `Page_Load()` method, or (for the sake of simplicity) the following inline code is used:

```
<%= DateTime.Now.ToLongTimeString() %>
```

Also, a button to trigger updating the time is created:

```
<asp:Button ID="Button1" runat="server" Text="Update" />
```

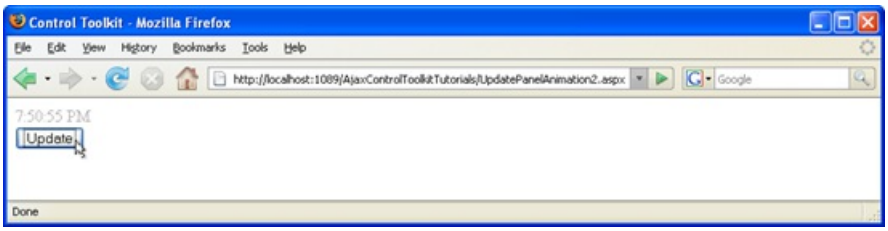
This code is then put into the `<ContentTemplate>` section of an `UpdatePanel` element. The panel's `UpdateMode` attribute must be set to `"Conditional"`, since only triggers may update the panel's contents. In the `<Triggers>` section of the `UpdatePanel`, an asynchronous postback trigger is created and tied to the `Click` event of the button. Thus, if the user clicks on the button, the `UpdatePanel` is refreshed. Here is the markup for the `UpdatePanel` control:

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server" UpdateMode="Conditional">
 <ContentTemplate>
 <%= DateTime.Now.ToLongTimeString() %>
 </ContentTemplate>
 <Triggers>
 <asp:AsyncPostBackTrigger ControlID="Button1" EventName="Click" />
 </Triggers>
</asp:UpdatePanel>
```

Finally, the `UpdatePanelAnimationExtender` must be configured: Set the `TargetControlID` attribute to the ID of the panel, and define an animation within the extender. Fading in makes sense, which creates a nice visual emphasis on the updated time. Your extender markup may then look like this:

```
<ajaxToolkit:UpdatePanelAnimationExtender ID="upae" runat="server" TargetControlID="UpdatePanel1">
 <Animations>
 <OnUpdated>
 <FadeIn Duration="1.0" Fps="24" />
 </OnUpdated>
 </Animations>
</ajaxToolkit:UpdatePanelAnimationExtender>
```

Run the file in the browser. Whenever you click on the button, the current time is shown in the panel, always fading in for the duration of one second.



The current time is fading in (Click to view full-size image)

PREVIOUS NEXT

Adding Animation to a Control (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. This tutorial shows how to set up such an animation.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. This tutorial shows how to set up such an animation.

Steps

The first step is as usual to include the `ScriptManager` in the page so that the ASP.NET AJAX library is loaded and the Control Toolkit can be used:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation in this scenario will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more efficient,
 more interactive and highly-personalized Web experiences that work across all the
 most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more efficient,
 more interactive and highly-personalized Web experiences that work across all the
 most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more efficient,
 more interactive and highly-personalized Web experiences that work across all the
 most popular browsers.<br />
</asp:Panel>
```

The associated CSS class for the panel defines a background color and a width:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

Next up, we need the `AnimationExtender`. After providing an `ID` and the usual `runat="server"`, the `TargetControlID` attribute must be set to the control to animate in our case, the panel:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

The whole animation is applied declaratively, using an XML syntax, unfortunately currently not fully supported by Visual Studio's IntelliSense. The root node is `<Animations>;` within this node, several events are allowed which determine when the animation(s) take(s) place:

- `OnClick` (mouse click)
- `OnHoverOut` (when the mouse leaves a control)
- `OnHoverOver` (when the mouse hovers over a control, stopping the `OnHoverOut` animation)
- `OnLoad` (when the page has been loaded)
- `OnMouseOut` (when the mouse leaves a control)
- `OnMouseOver` (when the mouse hovers over a control, not stopping the `OnMouseOut` animation)

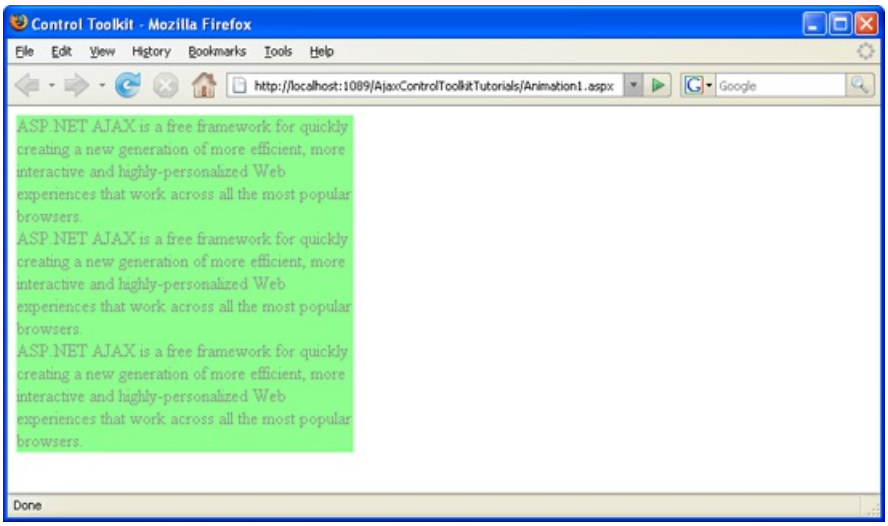
The framework comes with a set of animations, each one represented by its own XML element. Here is a selection:

- `<Color>` (changing a color)
- `<FadeIn>` (fading in)
- `<FadeOut>` (fading out)
- `<Property>` (changing a control's property)
- `<Pulse>` (pulsating)
- `<Resize>` (changing the size)
- `<Scale>` (proportionally changing the size)

In this example, the panel shall fade out. The animation shall take 1.5 seconds (`Duration` attribute), displaying 24 frames (animation steps) per second (`Fps` attribute). Here is the complete markup for the `AnimationExtender` control:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
 <Animations>
 <OnLoad>
 <FadeOut Duration="1.5" Fps="24" />
 </OnLoad>
 </Animations>
</ajaxToolkit:AnimationExtender>
```

When you run this script, the panel is displayed and fades out in one and a half seconds.



The panel is fading out (Click to view full-size image)

PREVIOUS NEXT

Executing Several Animations at The Same Time (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. It allows to run several animations in a parallel fashion.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. It allows to run several animations in a parallel fashion.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more efficient,
 more interactive and highly-personalized Web experiences that work across all the
 most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more efficient,
 more interactive and highly-personalized Web experiences that work across all the
 most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more efficient,
 more interactive and highly-personalized Web experiences that work across all the
 most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server"`:

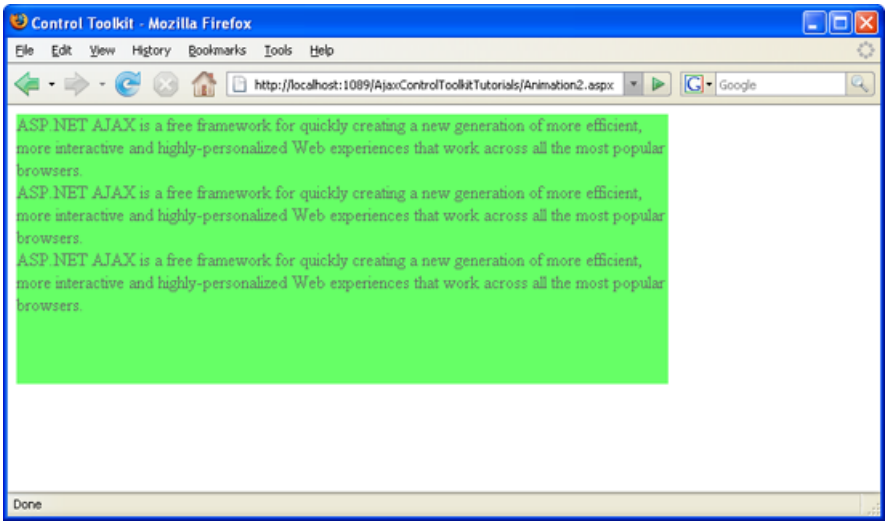
```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

Within the `<Animations>` node, use `<OnLoad>` to run the animations once the page has been fully loaded. Generally, `<OnLoad>` only accepts one animation. The Animation framework allows you to join several animations into one using the `<Parallel>` element. All animations within `<Parallel>` are executed at the same time.

Here is the a possible markup for the `AnimationExtender` control, fading out and resizing the panel at the same time:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
 <Animations>
 <OnLoad>
 <Parallel>
 <FadeOut Duration="1.5" Fps="24" />
 <Resize Width="1000" Height="150" Unit="px" />
 </Parallel>
 </OnLoad>
 </Animations>
</ajaxToolkit:AnimationExtender>
```

And indeed: when you run this script, the panel is displayed, then resizes (more than tripling its width and halving its height) and fades out at the same time.



The panel is fading out and resizing (including its content, thanks to the browser's rendering engine) (Click to view full-size image)

PREVIOUS NEXT

Executing Several Animations after Each Other (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. It allows to run several animations one after the other.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. It allows to run several animations one after the other.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server":`

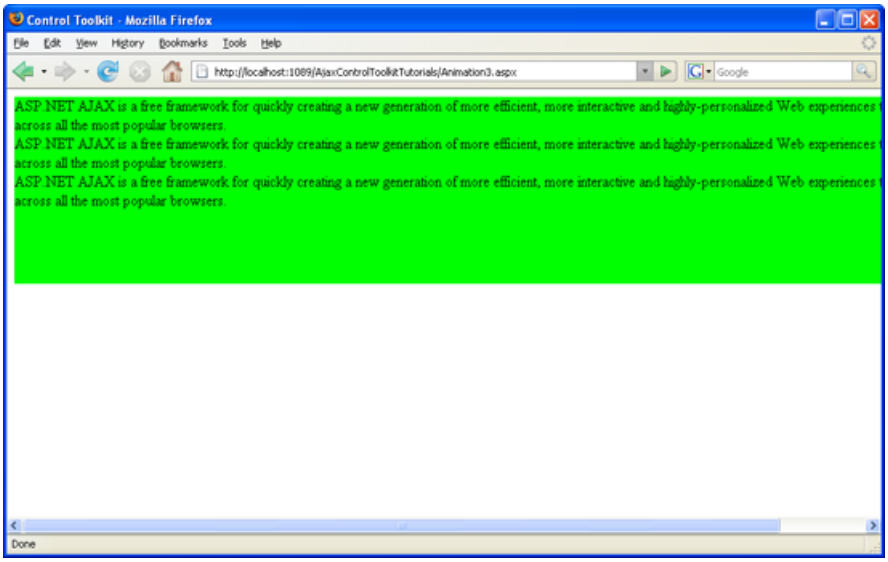
```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

Within the `<Animations>` node, use `<OnLoad>` to run the animations once the page has been fully loaded. Generally, `<OnLoad>` only accepts one animation. The Animation framework allows you to join several animations into one using the `<Sequence>` element. All animations within `<Sequence>` are executed one after the other. Here is

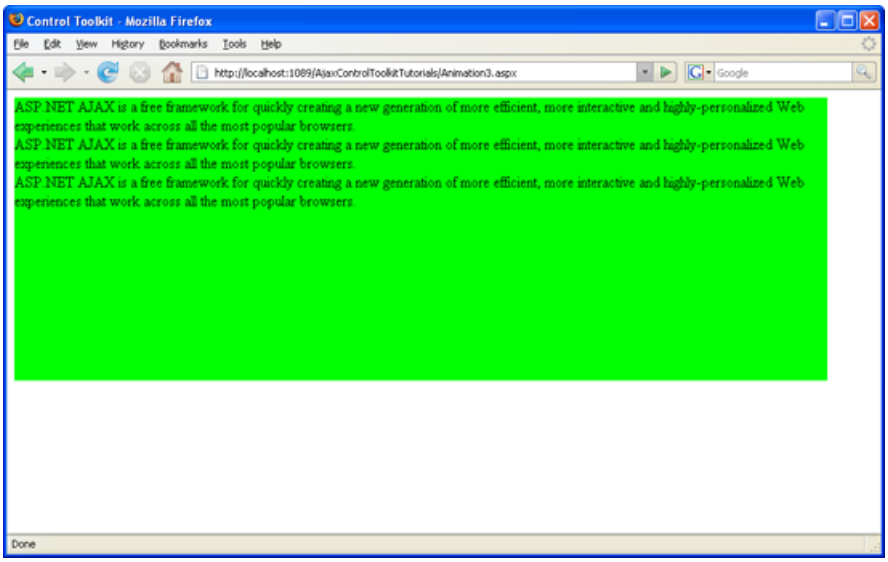
the a possible markup for the `AnimationExtender` control, first making the panel wider and then decreasing its height:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
 <Animations>
 <OnLoad>
 <Sequence>
 <Resize Width="1000" Unit="px" />
 <Resize Height="150" Unit="px" />
 </Sequence>
 </OnLoad>
 </Animations>
</ajaxToolkit:AnimationExtender>
```

When you run this script, the panel first gets wider and then smaller.



First the width is increased (Click to view full-size image)



Then the height is decreased (Click to view full-size image)

PREVIOUS NEXT

Animation Depending On a Condition (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. Whether an animation is run or not can also depend on a condition in form of some JavaScript code.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. Whether an animation is run or not can also depend on a condition in form of some JavaScript code.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server":`

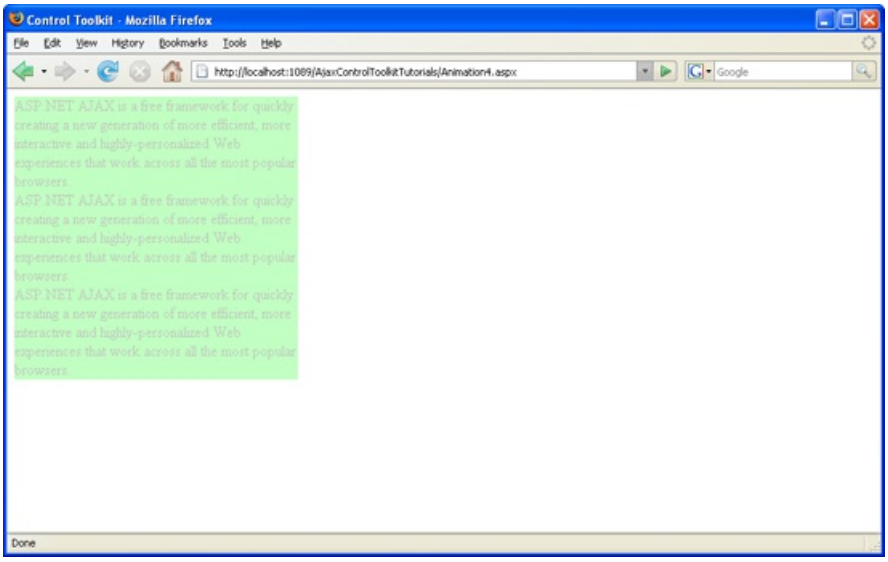
```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

Within the `<Animations>` node, use `<OnLoad>` to run the animations once the page has been fully loaded. Instead

of one of the regular animations, the `<Condition>` element comes into play. The JavaScript code provided as the value of the `ConditionScript` attribute is executed at runtime. If it evaluates to true, the animation is executed, otherwise not. The following markup provides two animations, each of them being executed in 50% of cases upon random. Since there may only be one animation within `<OnLoad>`, the two `<Condition>` animations are joined together using the `<Sequence>` element:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server"
 TargetControlID="Panel1">
 <Animations>
 <OnLoad>
 <Sequence>
 <Condition ConditionScript="Math.random()   0.5">
 <Resize Width="1000" Height="150" Unit="px" />
 </Condition>
 <Condition ConditionScript="Math.random()   0.5">
 <FadeOut Duration="1.5" Fps="24" />
 </Condition>
 </Sequence>
 </OnLoad>
 </Animations>
</ajaxToolkit:AnimationExtender>
```

Note that the less than sign (`<`) in the `ConditionScript` attribute must be escaped (). When you run this script, either no animation runs, or one of the two does, or both do.



The panel is fading out without resizing, so the second animation runs, the first one didn't (Click to view full-size image)

PREVIOUS NEXT

Picking One Animation Out Of a List (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The framework also allows the programmer to pick one animation out of a list of animations, depending on the evaluation of some JavaScript code.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The framework also allows the programmer to pick one animation out of a list of animations, depending on the evaluation of some JavaScript code.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server":`

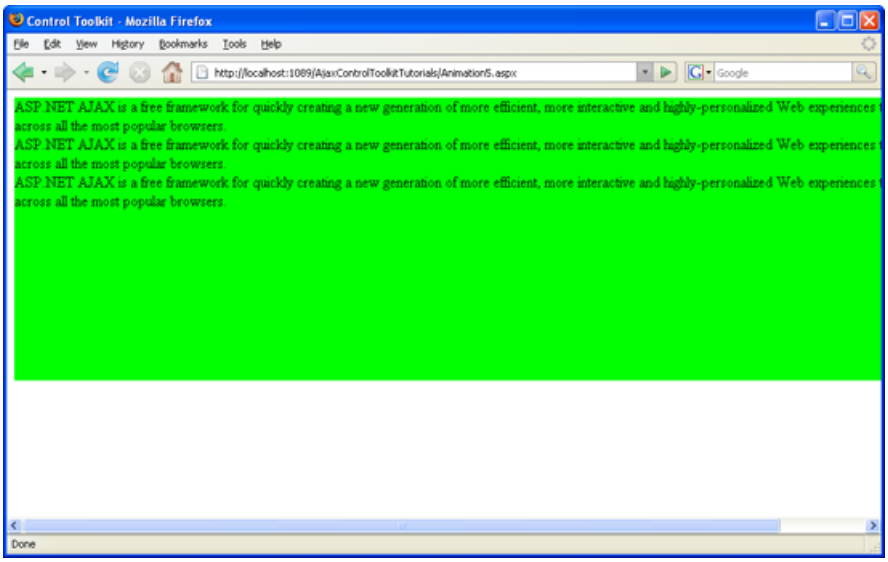
```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

Within the `<Animations>` node, use `<OnLoad>` to run the animations once the page has been fully loaded. Instead

of one of the regular animations, the `<Case>` element comes into play. The value of its SelectScript attribute is evaluated; the return value must be numerical. Depending on this number, one of the subanimations within <Case> is executed. For instance, if SelectScript evaluates to 2, the Control Toolkit runs the third animation within <Case> (counting starts at 0).

The following markup defines three subanimations: Resizing the width, resizing the height, and fading out. The JavaScript code (`Math.floor(3 * Math.random())`) then picks a number between 0 and 2, so that one of the three animations is run:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server"
 TargetControlID="Panel1">
 <Animations>
 <OnLoad>
 <Case SelectScript="Math.floor(3 * Math.random())">
 <Resize Width="1000" Unit="px" />
 <Resize Height="150" Unit="px" />
 <FadeOut Duration="1.5" Fps="24" />
 </Case>
 </OnLoad>
 </Animations>
</ajaxToolkit:AnimationExtender>
```



One of the possible three animations: The panel gets wider (Click to view full-size image)

PREVIOUS NEXT

Animating in Response To User Interaction (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animations can start automatically or may be triggered by user interaction, e.g. by clicking with the mouse.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animations can start automatically or may be triggered by user interaction, e.g. by clicking with the mouse.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server"`:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

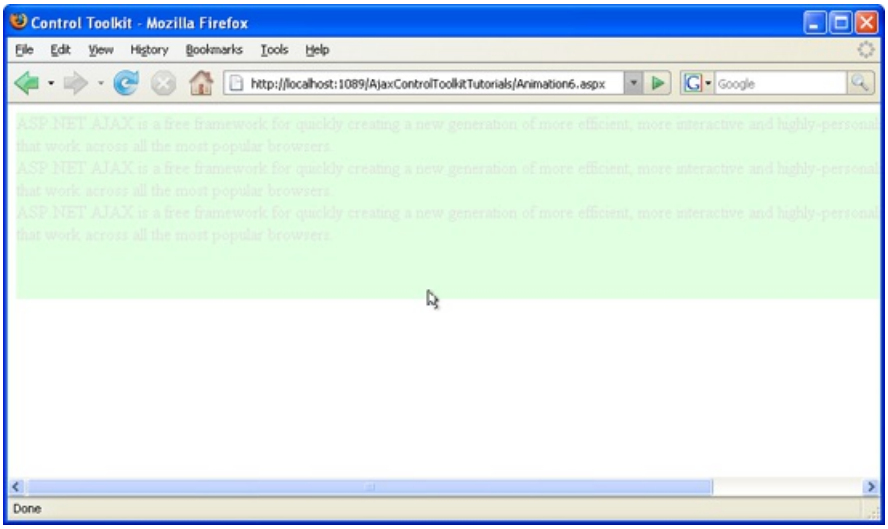
Within the `<Animations>` node, there are five ways to start the animation via user interaction (the missing element

is `<OnLoad>` which is executed once the whole page has been fully loaded):

- `<OnClick>` (mouse click on the control)
- `<OnHoverOut>` (mouse leaves the control)
- `<OnHoverOver>` (mouse hovers over a control, stopping the `<OnHoverOut>` animation)
- `<OnMouseOut>` (mouse leaves a control)
- `<OnMouseOver>` (mouse hovers over a control, not stopping the `<OnMouseOut>` animation)

In this scenario, `<OnClick>` is used. When the user clicks on the panel, it is resized and fades out at the same time.

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
 <Animations>
 <OnClick>
 <Parallel>
 <FadeOut Duration="1.5" Fps="24" />
 <Resize Width="1000" Height="150" Unit="px" />
 </Parallel>
 </OnClick>
 </Animations>
</ajaxToolkit:AnimationExtender>
```



A mouse click starts the animation (Click to view full-size image)

PREVIOUS NEXT

Disabling Actions during Animation (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. It also supports actions, like mouse clicks. However when a mouse click starts an animation, it is desirable to disable mouse clicks during the animation.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. It also supports actions, like mouse clicks. However when a mouse click starts an animation, it is desirable to disable mouse clicks during the animation.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to an HTML button like this:

```
<input type="button" ID="Button1" runat="server" Value="Launch Animation" />
```

Note that an HTML Control is used instead of a Web Control since we do not want the button to create a postback; it shall just launch the client-side animation for us.

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server"`:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Button1">
```

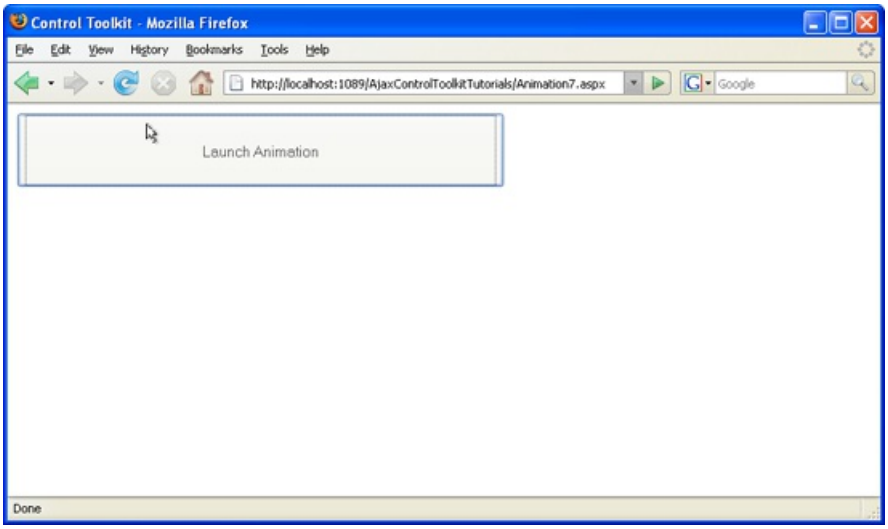
Within the `<Animations>` node, `<OnClick>` is the right element to handle the mouse click. However, the button could be clicked during the animation, as well. The `<EnableAction>` element can take care of that. Setting `Enabled="false"` disables the button as part of the animation. Since we are using several individual animations (disabling the button and the actual animations), the `<Parallel>` element is required to glue the single animations together into one. Here is the complete markup for `AnimationExtender`:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Button1">
 <Animations>
 <OnClick>
 <Parallel>
 <EnableAction Enabled="false" />
 <FadeOut Duration="1.5" Fps="24" />
 <Resize Width="1000" Height="150" Unit="px" />
 </Parallel>
 </OnClick>
 </Animations>
</ajaxToolkit:AnimationExtender>
```

It would also be possible to re-enable to button after the animation, using the following XML element at the end of the list:

```
<EnableAction Enabled="true" />
```

However in the given scenario this would be useless since the button fades out and is not visible at the end of the animation.



The button is disabled as soon as the animation runs (Click to view full-size image)

PREVIOUS NEXT

Triggering an Animation in another Control (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. Generally, launching an animation is triggered by user interaction with the same control. It is however also possible to interact with one control and then animation another control.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. Generally, launching an animation is triggered by user interaction with the same control. It is however also possible to interact with one control and then animation another control.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

In order to start animating the panel, an HTML button is used. Note that `<input type="button" />` is favoured over `<asp:Button />` since we do not want a postback when the user clicks on that button.

```
<input type="button" id="Button1" runat="server" Value="Launch Animation" />
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory

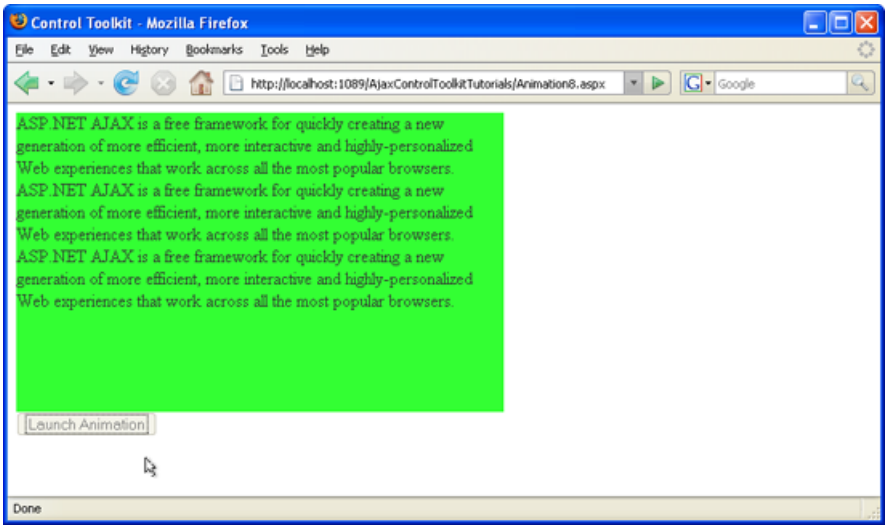
`runat="server"`. It is important to set `TargetControlID` to the ID of the button (the element triggering the animation), not to the ID of the panel (the element being animated)

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Button1">
```

Within the `<Animations>` node, place animations as usual. In order to make them change the panel, not the button, set the `AnimationTarget` attribute for every animation element within `AnimationExtender`. The value for `AnimationTarget` is the ID of the panel, of course. That way, the animations happen with the panel, not with the triggering button. Here is the `AnimationExtender` markup for this scenario:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
 <Animations>
 <OnClick>
 <Sequence>
 <EnableAction Enabled="false" />
 <Parallel>
 <FadeOut Duration="1.5" Fps="24" AnimationTarget="Panel1" />
 <Resize Width="1000" Height="150" Unit="px"
 AnimationTarget="Panel1" />
 </Parallel>
 </Sequence>
 </OnClick>
 </Animations>
</ajaxToolkit:AnimationExtender>
```

Note the special order in which the individual animations appear. First of all, the button gets deactivated once the animation runs. Since there is no `AnimationTarget` attribute in the `<EnableAction>` element, this animation is applied to the originating control: the button. The next two animation steps shall be carried out in parallel (`<Parallel>` element). Both have their `AnimationTarget` attributes set to `"Panel1"`, thus animating the panel, not the button.



A mouse click on the button starts the panel animation (Click to view full-size image)

PREVIOUS NEXT

Modifying Animations From The Server Side (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animations may also be changed on the server-side

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animations may also be changed on the server-side

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

The rest of the code runs on the server-side and does not use markup; instead, it uses code to create the `AnimationExtender` control:

```
<script runat="server">
Sub Page_Load()
 Dim ae As New AjaxControlToolkit.AnimationExtender()
 ae.TargetControlID = "Panel1"
```

However, the Control Toolkit currently does not provide an API access to create the individual animations. It is

however possible to set the `AnimationExtender`'s Animations property to a string containing the XML markup used when assigning the animations declaratively. In order to create the XML which must not contain the `<Animations>` element you could use the .NET Framework's XML support or, as in the following code, just provide the string:

```
ae.Animations = "<OnLoad><Parallel><FadeOut Duration=""1.5""
 Fps=""24"" /><Resize Width=""1000""
 Height=""150"" Unit=""px"" /></Parallel></OnLoad>"
```

Finally, add the `AnimationExtender` control to the current page, within the `<form runat="server">` element, making sure that the animation is included and runs:

```
form1.Controls.Add(ae)
End Sub
</script>
```

The animation is created using server-side C#/VB code (Click to view full-size image)

PREVIOUS NEXT

Executing Animations Using Client-Side Code (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animation execution may also be triggered using custom client-side JavaScript code.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animation execution may also be triggered using custom client-side JavaScript code.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server"`:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Panel1">
```

Within the `<Animations>` node, use `<OnClick>` to run the animations once the user clicks on the panel. Add two animations to be executed in parallel:

```
<Animations>
 <OnClick>
 <Parallel>
 <FadeOut Duration="1.5" Fps="24" />
 <Resize Width="1000" Height="150" Unit="px" />
 </Parallel>
 </OnClick>
 </Animations>
</ajaxToolkit:AnimationExtender>
```

For the sake of demonstration, this animation (and any other animation created using the Control Toolkit) is executed using JavaScript code, once the page runs. First of all we need access to the `AnimationExtender` control. The ASP.NET AJAX library provides the `$find()` function for this task:

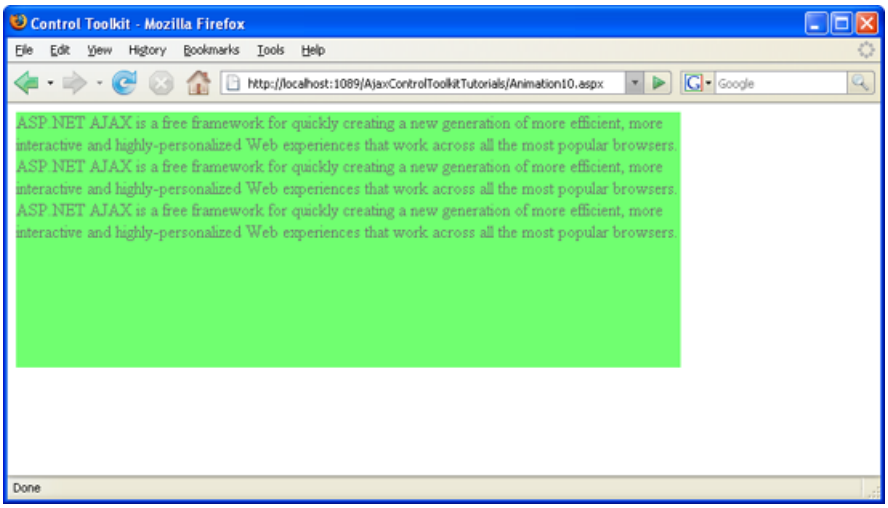
```
var ae = $find("ae");
```

The `AnimationExtender` control exposes a rich API, including methods with names identical to the event handlers used in the XML markup: `OnClick()`, `OnLoad()`, and so on. For instance, a call of the `OnClick()` method executes the animation within the `<OnClick>` element of the `AnimationExtender` control:

```
ae.OnClick();
```

Here is the complete client-side JavaScript code that emulates the click on the panel once the page has been fully loaded note that the `pageLoad()` function name is used which is called by ASP.NET AJAX once the page and all included JavaScript libraries have been loaded.

```
<script type="text/javascript">
 function pageLoad() {
 var ae = $find("ae");
 ae.OnClick();
 }
</script>
```



The animation runs immediately, without a mouse click (Click to view full-size image)

PREVIOUS NEXT

Changing an Animation Using Client-Side Code (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animation can also be changed using custom client-side JavaScript code.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. The animation can also be changed using custom client-side JavaScript code.

Steps

First of all, include the `ScriptManager` in the page; then, the ASP.NET AJAX library is loaded, making it possible to use the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server"/>
```

The animation will be applied to a panel of text which looks like this:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panelClass">
 ASP.NET AJAX is a free framework for quickly creating a new generation of
 more efficient, more interactive and highly-personalized Web experiences
 that work across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of
 more efficient, more interactive and highly-personalized Web experiences
 that work across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of
 more efficient, more interactive and highly-personalized Web experiences
 that work across all the most popular browsers.<br />
</asp:Panel>
```

In the associated CSS class for the panel, define a nice background color and also set a fixed width for the panel:

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

The actual animation is launched by an HTML button:

```
<input type="button" id="Button1" runat="server" value="Launch Animation" />
```

Then, add the `AnimationExtender` to the page, providing an `ID`, the `TargetControlID` attribute and the obligatory `runat="server"`:

```
<ajaxToolkit:AnimationExtender ID="ae" runat="server" TargetControlID="Button1" />
```

Note that there is no `<Animations>` node within the `AnimationExtender` control. Custom JavaScript code is used to provide the animations to be used with the control.

As with the server API of `AnimationExtender`, there is no easy way to assign an animation to the extender yet. However the extender does expose several methods to read and write animations registered with the various events (`OnClick`, `OnLoad`, and so on). Here are some examples:

- `get_OnClick()`
- `set_OnClick()`
- `get_OnLoad()`
- `set_OnLoad()`
- `...`

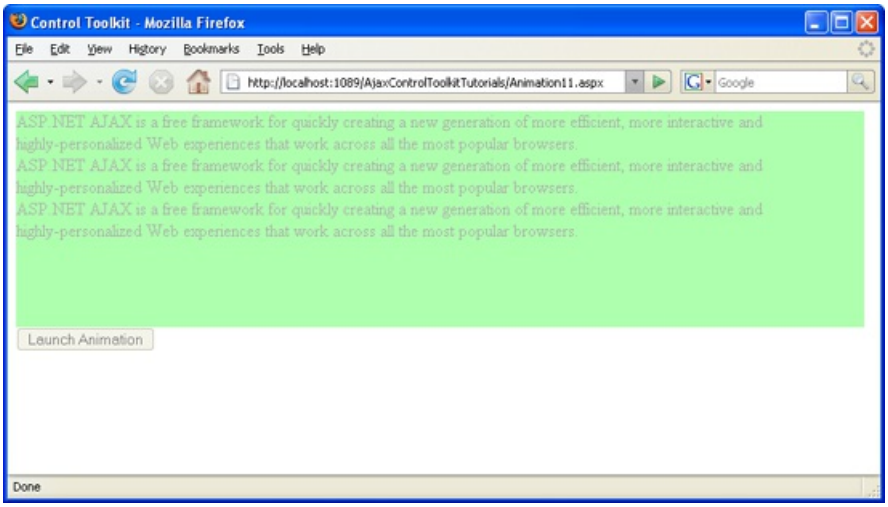
The format of the return value of the `get_*()` functions and the format of the argument for the `set_*()` functions is a JSON string, providing an object representation of what the XML markup would be. Currently, there is no way to pass an object in, but it is possible to read an object from a given animation (`get_OnXXXBehavior()` methods).

Here is a JSON string (without the delimiting quotes and formatted nicely) representing an animation triggered by the button, but animating the panel by resizing it and fading it out at the same time:

```
{
 "AnimationName":"Sequence",
 "AnimationChildren":[
 {
 "AnimationName":"EnableAction",
 "Enabled":"false",
 "AnimationChildren":[]
 },
 {
 "AnimationName":"Parallel",
 "AnimationChildren":[
 {
 "AnimationName":"FadeOut",
 "Duration":"1.5",
 "Fps":"24",
 "AnimationTarget":"Panel1",
 "AnimationChildren":[]
 },
 {
 "AnimationName":"Resize",
 "Width":"1000",
 "Height":"150",
 "Unit":"px",
 "AnimationTarget":"Panel1",
 "AnimationChildren":[]
 }]
 }]
}
```

The following JavaScript code assigns this JSON descripting to the `OnClick` animation of the current extender and runs it:

```
<script type="text/javascript">
 function pageLoad()
 {
 var ae = $find("ae");
 var animation = '{"AnimationName":"Sequence","AnimationChildren":
[{"AnimationName":"EnableAction","Enabled":"false","AnimationChildren":[]},
{"AnimationName":"Parallel","AnimationChildren":
[{"AnimationName":"FadeOut","Duration":"1.5","Fps":"24","AnimationTarget":"Panel1","AnimationChildren":[]},
{"AnimationName":"Resize","Width":"1000","Height":"150","Unit":"px","AnimationTarget":"Panel1","AnimationChil
dren":[]}]}]}';
 ae.set_OnClick(animation);
 ae.OnClick();
 }
</script>
```



The animation runs immediately, without a mouse click (and with very little markup) (Click to view full-size image)

PREVIOUS NEXT

Animating an UpdatePanel Control (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. For the contents of an UpdatePanel, a special extender exists that relies heavily on the animation framework: UpdatePanelAnimation. This tutorial shows how to set up such an animation for an UpdatePanel.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. For the contents of an `UpdatePanel`, a special extender exists that relies heavily on the animation framework: `UpdatePanelAnimation`. This tutorial shows how to set up such an animation for an `UpdatePanel`.

Steps

The first step is as usual to include the `ScriptManager` in the page so that the ASP.NET AJAX library is loaded and the Control Toolkit can be used:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation in this scenario will be applied to an ASP.NET `Wizard` web control residing in an `UpdatePanel`. Three (arbitrary) steps provide enough options to trigger postbacks:

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
 <ContentTemplate>
 <asp:Wizard ID="Wizard1" runat="server">
 <WizardSteps>
 <asp:WizardStep runat="server" StepType="Start" Title="Ready!">
 </asp:WizardStep>
 <asp:WizardStep runat="server" StepType="Step" Title="Set!">
 </asp:WizardStep>
 <asp:WizardStep runat="server" StepType="Finish" Title="Go!">
 </asp:WizardStep>
 </WizardSteps>
 </asp:Wizard>
 </ContentTemplate>
</asp:UpdatePanel>
```

The markup necessary for the `UpdatePanelAnimationExtender` control is quite similar to the markup used for the `AnimationExtender`. In the `TargetControlID` attribute we provide the `ID` of the `UpdatePanel` to animate; within the `UpdatePanelAnimationExtender` control, the `<Animations>` element holds XML markup for the animation(s). However there is one difference: The amount of events and event handlers is limited in comparison to `AnimationExtender`. For `UpdatePanels`, only two of them exist:

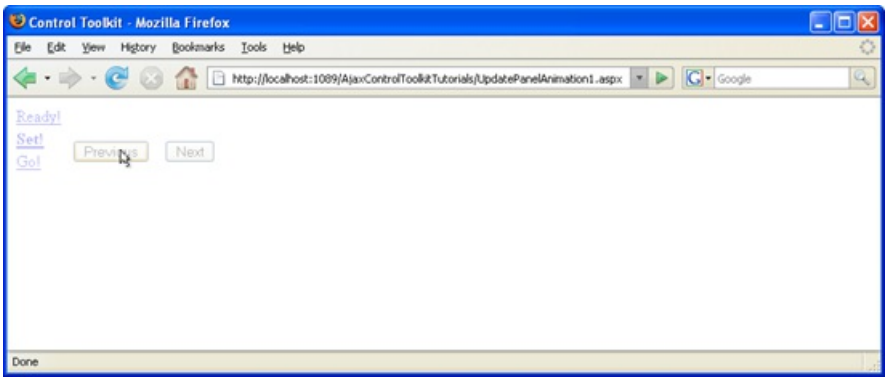
- `<OnUpdated>` when the UpdatePanel has been updated

- `<OnUpdating>` when the UpdatePanel starts updating

In this scenario, the new content of the `UpdatePanel` (after the postback) shall fade in. This is the necessary markup for that:

```
<ajaxToolkit:UpdatePanelAnimationExtender ID="upae" runat="server"
 TargetControlID="UpdatePanel1">
 <Animations>
 <OnUpdated>
 <FadeIn Duration="1.0" Fps="24" />
 </OnUpdated>
 </Animations>
</ajaxToolkit:UpdatePanelAnimationExtender>
```

Now whenever a postback occurs within the UpdatePanel, the new contents of the panel fade in smoothly.



The next wizard step is fading in (Click to view full-size image)

PREVIOUS NEXT

Dynamically Controlling UpdatePanel Animations (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. For the contents of an UpdatePanel, a special extender exists that relies heavily on the animation framework: UpdatePanelAnimation. It can also work together with UpdatePanel triggers.

Overview

The Animation control in the ASP.NET AJAX Control Toolkit is not just a control but a whole framework to add animations to a control. For the contents of an `UpdatePanel`, a special extender exists that relies heavily on the animation framework: `UpdatePanelAnimation`. It can also work together with `UpdatePanel` triggers.

Steps

The first step is as usual to include the `ScriptManager` in the page so that the ASP.NET AJAX library is loaded and the Control Toolkit can be used:

```
<asp:ScriptManager ID="asm" runat="server" />
```

The animation in this scenario will be applied to a display of the current time. This information can be written into a label using the `Page_Load()` method, or (for the sake of simplicity) the following inline code is used:

```
<%= DateTime.Now.ToLongTimeString() %>
```

Also, a button to trigger updating the time is created:

```
<asp:Button ID="Button1" runat="server" Text="Update" />
```

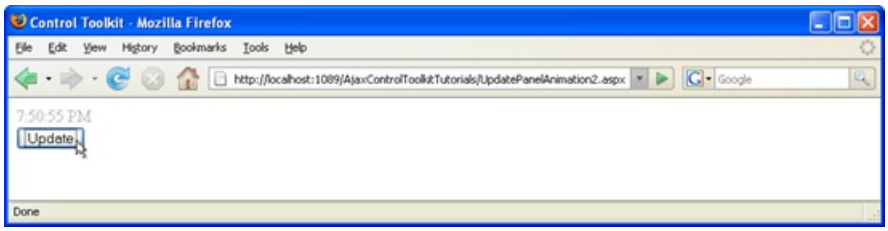
This code is then put into the `<ContentTemplate>` section of an `UpdatePanel` element. The panel's `UpdateMode` attribute must be set to `"Conditional"`, since only triggers may update the panel's contents. In the `<Triggers>` section of the `UpdatePanel`, an asynchronous postback trigger is created and tied to the `Click` event of the button. Thus, if the user clicks on the button, the `UpdatePanel` is refreshed. Here is the markup for the `UpdatePanel` control:

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server" UpdateMode="Conditional">
 <ContentTemplate>
 <%= DateTime.Now.ToLongTimeString() %>
 </ContentTemplate>
 <Triggers>
 <asp:AsyncPostBackTrigger ControlID="Button1" EventName="Click" />
 </Triggers>
</asp:UpdatePanel>
```

Finally, the `UpdatePanelAnimationExtender` must be configured: Set the `TargetControlID` attribute to the ID of the panel, and define an animation within the extender. Fading in makes sense, which creates a nice visual emphasis on the updated time. Your extender markup may then look like this:

```
<ajaxToolkit:UpdatePanelAnimationExtender ID="upae" runat="server" TargetControlID="UpdatePanel1">
 <Animations>
 <OnUpdated>
 <FadeIn Duration="1.0" Fps="24" />
 </OnUpdated>
 </Animations>
</ajaxToolkit:UpdatePanelAnimationExtender>
```

Run the file in the browser. Whenever you click on the button, the current time is shown in the panel, always fading in for the duration of one second.



The current time is fading in (Click to view full-size image)

PREVIOUS

CascadingDropdown

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the CascadingDropDown control to create related drop-down lists that update when users make a selection.

- Filling a List Using CascadingDropDown (C#)
- Using CascadingDropDown with a Database (C#)
- Presetting List Entries with CascadingDropDown (C#)
- Using Auto-Postback with CascadingDropDown (C#)
- Filling a List Using CascadingDropDown (VB)
- Using CascadingDropDown with a Database (VB)
- Presetting List Entries with CascadingDropDown (VB)
- Using Auto-Postback with CascadingDropDown (VB)

Filling a List Using CascadingDropDown (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. (For instance, one list provides a list of US states, and the next list is then filled with major cities in that state.) The first challenge to solve is to actually fill a dropdown list using this control.

Overview

The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. (For instance, one list provides a list of US states, and the next list is then filled with major cities in that state.) The first challenge to solve is to actually fill a dropdown list using this control.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, a DropDownList control is required:

```
<div>
 Vendor: <asp:DropDownList ID="VendorsList" runat="server" />
</div>
```

For this list, a CascadingDropDown extender is added. It will send an asynchronous request to a web service which will then return a list of entries to be displayed in the list. For this to work, the following CascadingDropDown attributes need to be set:

- `ServicePath`: URL of a web service delivering the list entries
- `ServiceMethod`: Web method delivering the list entries
- `TargetControlID`: ID of the dropdown list
- `Category`: Category information that is submitted to the web method when called
- `PromptText`: Text displayed when asynchronously loading list data from the server

Here is the markup for the `CascadingDropDown` element. The only difference between C# and VB is the name of the associated web service:

```
<ajaxToolkit:CascadingDropDown ID="ccd1" runat="server"
 ServicePath="CascadingDropdown0.cs.asmx" ServiceMethod="GetVendors"
 TargetControlID="VendorsList" Category="Vendor" />
```

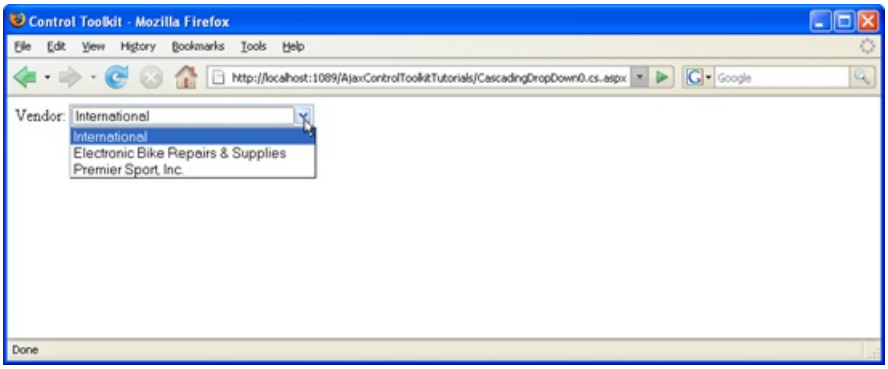
The JavaScript code coming from the `CascadingDropDown` extender calls a web service method with the following signature:

```
public CascadingDropDownNameValue[] MethodNameHere(string knownCategoryValues,
    string category)
```

So the important aspect is that the method needs to return an array of type `CascadingDropDownNameValue` (defined by the ASP.NET AJAX Control Toolkit). In the `CascadingDropDownNameValue` constructor, first the list entry's text and then its value must be provided, just as `<option value="VALUE">NAME</option>` would do in HTML. Here is some sample data:

```
<%@ WebService Language="C#" Class="CascadingDropdown0" %>
using System.Web.Script.Services;
using AjaxControlToolkit;
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Collections.Generic;
[ScriptService]
public class CascadingDropdown0 : System.Web.Services.WebService
{
 [WebMethod]
 public CascadingDropDownNameValue[] GetVendors(string knownCategoryValues,
 string category)
 {
 List<CascadingDropDownNameValue> l = new List<CascadingDropDownNameValue>();
 l.Add(new CascadingDropDownNameValue("International", "1"));
 l.Add(new CascadingDropDownNameValue("Electronic Bike Repairs & Supplies", "2"));
 l.Add(new CascadingDropDownNameValue("Premier Sport, Inc.", "3"));
 return l.ToArray();
 }
}
```

Loading the page in the browser will trigger the list to be filled with three vendors.



The list is filled automatically (Click to view full-size image)

NEXT

Using CascadingDropDown with a Database (C#)

2/20/2020 • 4 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. In order for this to work, a special web service must be created.

Overview

The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. (For instance, one list provides a list of US states, and the next list is then filled with major cities in that state.) In order for this to work, a special web service must be created.

Steps

First of all, a data source is required. This sample uses the AdventureWorks database and the Microsoft SQL Server 2005 Express Edition. The database is an optional part of a Visual Studio installation (including express edition) and is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. The AdventureWorks database is part of the SQL Server 2005 Samples and Sample Databases (download at https://www.microsoft.com/downloads/details.aspx?FamilyID=e719ecf7-9f46-4312-af89-6ad8702e4e6e&DisplayLang=en). The easiest way to set the database up is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en) and attach the `AdventureWorks.mdf` database file.

For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database.

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the < `form` > element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

In the next step, two DropDownList controls are required. In this sample, we use the vendor and contact information from AdventureWorks, thus we create one list for the available vendors and one for the available contacts:

```
<div>
 Vendor: <asp:DropDownList ID="VendorsList" runat="server"/><br />
 Contacts: <asp:DropDownList ID="ContactsList" runat="server"/><br />
</div>
```

Then, two CascadingDropDown extenders must be added to the page. One fills the first (vendors) list, and the other one fills the second (contacts) list. The following attributes must be set:

- `ServicePath`: URL of a web service delivering the list entries
- `ServiceMethod`: Web method delivering the list entries
- `TargetControlID`: ID of the dropdown list
- `Category`: Category information that is submitted to the web method when called
- `PromptText`: Text displayed when asynchronously loading list data from the server
- `ParentControlID`: (optional) parent dropdown list that triggers loading of the current list

Depending on the programming language used, the name of the web service in question changes, but all other attribute values are the same. Here is the CascadingDropDown element for the first dropdown list:

```
<ajaxToolkit:CascadingDropDown ID="ccd1" runat="server"
 ServicePath="CascadingDropdown1.cs.asmx" ServiceMethod="GetVendors"
 TargetControlID="VendorsList" Category="Vendor"
 PromptText="Select Vendor" />
```

The control extenders for the second list need to set the `ParentControlID` attribute so that selecting an entry in the vendors list triggers loading associated elements in the contacts list.

```
<ajaxToolkit:CascadingDropDown ID="ccd2" runat="server"
 ServicePath="CascadingDropdown1.cs.asmx" ServiceMethod="GetContactsForVendor"
 TargetControlID="ContactsList" ParentControlID="VendorsList"
 Category="Contact"
 PromptText="Select Contact" />
```

The actual work is then done in the web service, which is set up as follows. Note that the `[ScriptService]` attribute is used, otherwise ASP.NET AJAX cannot create the JavaScript proxy to access the web methods from client-side script code.

```
<%@ WebService Language="C#" Class="CascadingDropdown1" %>
using System.Web.Script.Services;
using AjaxControlToolkit;
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Data.SqlClient;
[ScriptService]
public class CascadingDropdown1 : System.Web.Services.WebService
{
 // ...
}
```

The signature of the web methods called by CascadingDropDown is as follows:

```
public CascadingDropDownNameValue[] MethodNameHere(string knownCategoryValues,
    string category)
```

So the return value must be an array of type `CascadingDropDownNameValue` which is defined by the Control Toolkit. The `GetVendors()` method is quite easy to implement: The code connects to the AdventureWorks database and queries the first 25 vendors. The first parameter in the `CascadingDropDownNameValue` constructor is the caption of the list entry, the second one its value (value attribute in HTML's < `option` > element). Here is the code:

```
[WebMethod]
public CascadingDropDownNameValue[] GetVendors(string knownCategoryValues, string category)
{
    SqlConnection conn = new SqlConnection("server=(local)\\SQLEXPRESS;
    Integrated Security=true; Initial Catalog=AdventureWorks");
    conn.Open();
    SqlCommand comm = new SqlCommand("SELECT TOP 25 VendorID, Name
    FROM Purchasing.Vendor",conn);
    SqlDataReader dr = comm.ExecuteReader();
    List<CascadingDropDownNameValue> l = new List<CascadingDropDownNameValue>();
    while (dr.Read())
    {
        l.Add(new CascadingDropDownNameValue(dr["Name"].ToString(),
        dr["VendorID"].ToString()));
    }
    conn.Close();
    return l.ToArray();
}
```

Getting the associated contacts for a vendor (method name: `GetContactsForVendor()`) is a bit trickier. First of all, the vendor which has been selected in the first dropdown list must be determined. The Control Toolkit defines a helper method for that task: The `ParseKnownCategoryValuesString()` method returns a `StringDictionary` element with the dropdown data:

```
[WebMethod]
public CascadingDropDownNameValue[] GetContactsForVendor(string knownCategoryValues,
    string category)
{
    int VendorID;
    CascadingDropDown.ParseKnownCategoryValuesString(knownCategoryValues);
```

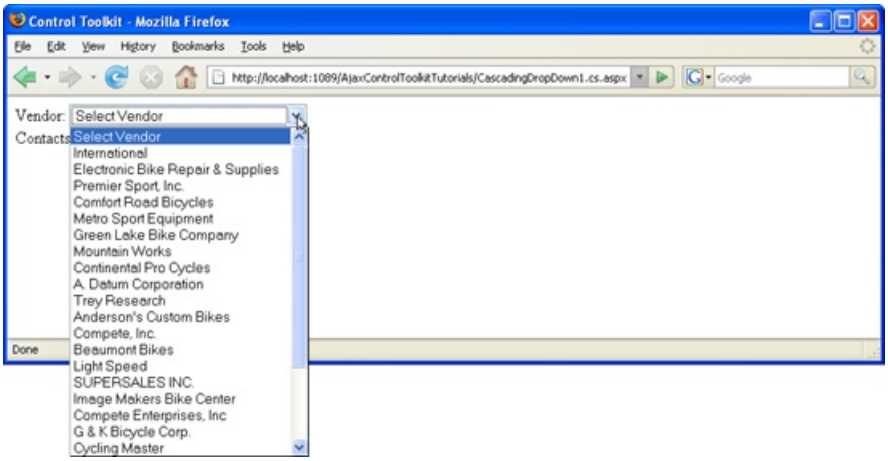
For security reasons, this data must be validated first. So if there is a Vendor entry (because the `Category` property of the first CascadingDropDown element is set to `"Vendor"`), the ID of the selected vendor may be retrieved:

```
if (!kv.ContainsKey("Vendor") || !Int32.TryParse(kv["Vendor"],out VendorID))
{
    throw new ArgumentException("Couldn't find vendor.");
};
```

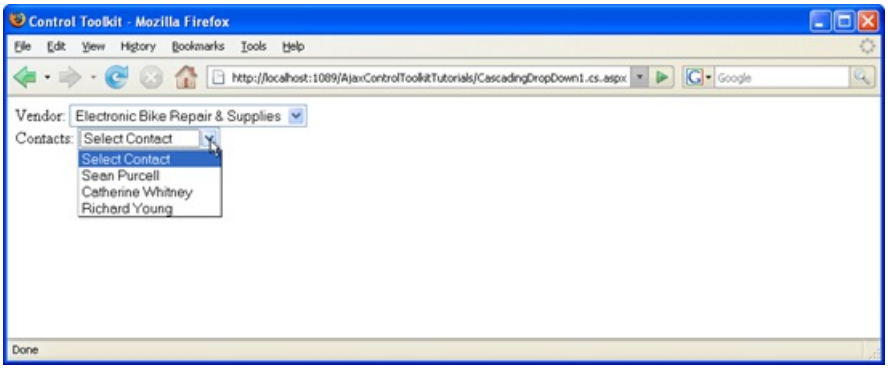
The rest of the method is fairly straight-forward, then. The vendor's ID is used as a parameter for an SQL query that retrieves all associated contacts for that vendor. Once again, the method returns an array of type `CascadingDropDownNameValue`.

```
SqlConnection conn = new SqlConnection("server=(local)\\SQLEXPRESS;
 Integrated Security=true; Initial Catalog=AdventureWorks");
 conn.Open();
 SqlCommand comm = new SqlCommand("SELECT Person.Contact.ContactID, FirstName, LastName
 FROM Person.Contact,Purchasing.VendorContact
 WHERE VendorID=@VendorID
 AND Person.Contact.ContactID=Purchasing.VendorContact.ContactID",conn);
 comm.Parameters.AddWithValue("@VendorID", VendorID);
 SqlDataReader dr = comm.ExecuteReader();
 List<CascadingDropDownNameValue> l = new List<CascadingDropDownNameValue>();
 while (dr.Read())
 {
 l.Add(new CascadingDropDownNameValue(
 dr["FirstName"].ToString() + " " + dr["LastName"].ToString(),
 dr["ContactID"].ToString()));
 }
 conn.Close();
 return l.ToArray();
}
```

Load the ASP.NET page, and after a short while the vendor list is filled with 25 entries. Pick one entry and notice how the second dropdown list is filled with data.



The first list is filled automatically (Click to view full-size image)



The second list is filled according to the selection in the first list (Click to view full-size image)

PREVIOUS NEXT

Presetting List Entries with CascadingDropDown (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. With a little bit of code it is possible that a list element is preselected once the data has been dynamically loaded.

Overview

The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. (For instance, one list provides a list of US states, and the next list is then filled with major cities in that state.) With a little bit of code it is possible that a list element is preselected once the data has been dynamically loaded.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, a DropDownList control is required:

```
<div>
 Vendor: <asp:DropDownList ID="VendorsList" runat="server"/>
</div>
```

For this list, a CascadingDropDown extender is added, providing web service URL and method information:

```
<ajaxToolkit:CascadingDropDown ID="ccd1" runat="server"
 ServicePath="CascadingDropdown2.cs.asmx" ServiceMethod="GetVendors"
 TargetControlID="VendorsList" Category="Vendor" />
```

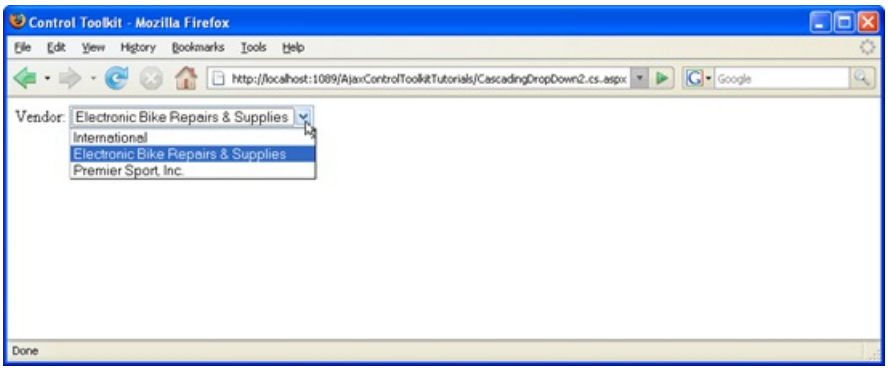
The CascadingDropDown extender then asynchronously calls a web service with the following method signature:

```
public CascadingDropDownNameValue[] MethodNameHere(string knownCategoryValues, string category)
```

The method returns an array of type CascadingDropDown value. The type's constructor expects first the list entry's caption and then the value (HTML `value` attribute). If the third argument is set to true, the list element is automatically selected in the browser.

```
<%@ WebService Language="C#" Class="CascadingDropdown2" %>
using System.Web.Script.Services;
using AjaxControlToolkit;
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Collections.Generic;
[ScriptService]
public class CascadingDropdown2 : System.Web.Services.WebService
{
 [WebMethod]
 public CascadingDropDownNameValue[] GetVendors(string knownCategoryValues, string
 category)
 {
 List<CascadingDropDownNameValue> l = new List<CascadingDropDownNameValue>();
 l.Add(new CascadingDropDownNameValue(
 "International", "1"));
 l.Add(new CascadingDropDownNameValue(
 "Electronic Bike Repairs & Supplies", "2", true));
 l.Add(new CascadingDropDownNameValue(
 "Premier Sport, Inc.", "3"));
 return l.ToArray();
 }
}
```

Loading the page in the browser will fill the dropdown list with three vendors, the second one being preselected.



The list is filled and preselected automatically (Click to view full-size image)

PREVIOUS NEXT

Using Auto-Postback with CascadingDropDown (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. However when using the CascadingDropDown control, ASP.NET's DropDownList control's AutoPostBack feature does not work, since asynchronously loading data into the list generates an (unnecessary) postback itself. With some JavaScript code, this effect can be avoided.

Overview

The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. (For instance, one list provides a list of US states, and the next list is then filled with major cities in that state.) However when using the CascadingDropDown control, ASP.NET's DropDownList control's AutoPostBack feature does not work, since asynchronously loading data into the list generates an (unnecessary) postback itself. With some JavaScript code, this effect can be avoided.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the < `form` > element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, a DropDownList control is required:

```
<div>
 Vendor: <asp:DropDownList ID="VendorsList" runat="server"/>
</div>
```

For this list, a CascadingDropDown extender is added, providing web service URL and method information:

```
<ajaxToolkit:CascadingDropDown ID="ccd1" runat="server"
 ServicePath="CascadingDropdown3.cs.asmx" ServiceMethod="GetVendors"
 TargetControlID="VendorsList" Category="Vendor" />
```

The CascadingDropDown extender then asynchronously calls a web service with the following method signature:

```
public CascadingDropDownNameValue[] MethodNameHere(string knownCategoryValues, string category)
```

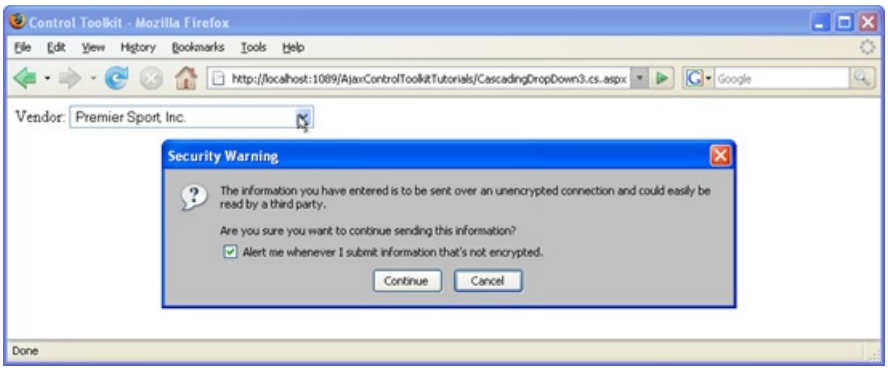
The method returns an array of type CascadingDropDown value. The type's constructor expects first the list entry's caption and then the value (HTML `value` attribute).

```
<%@ WebService Language="C#" Class="CascadingDropdown3" %>
using System.Web.Script.Services;
using AjaxControlToolkit;
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Collections.Generic;
[ScriptService]
public class CascadingDropdown3 : System.Web.Services.WebService
{
 [WebMethod]
 public CascadingDropDownNameValue[] GetVendors(string knownCategoryValues,
 string category)
 {
 List<CascadingDropDownNameValue> l = new List<CascadingDropDownNameValue>();
 l.Add(new CascadingDropDownNameValue("International", "1"));
 l.Add(new CascadingDropDownNameValue("Electronic Bike Repairs & Supplies", "2"));
 l.Add(new CascadingDropDownNameValue("Premier Sport, Inc.", "3"));
 return l.ToArray();
 }
}
```

Loading the page in the browser will fill the dropdown list with three vendors, the second one being preselected. Also, ASP.NET defines the `__doPostBack()` JavaScript method. Once the page has been loaded, this JavaScript call is added to the dropdown list, but only if there are elements in it. If there are no elements in the list, the Control Toolkit is currently loading them, so the JavaScript code uses a timeout and tries again in a half second.

```
<script type="text/javascript">
 function pageLoad()
 {
 addAutoPostBack();
 }
 function addAutoPostBack()
 {
 if ($get("VendorsList").options.length > 0)
 {
 $get("VendorsList").setAttribute("onchange","javascript:setTimeout('__doPostBack(\\'VendorsList\\',\\'\\')',
0)");
 }
 else
 {
 setTimeout("addAutoPostBack()", 500);
 }
 }
</script>
```

This way, a postback is only executed when there are actually elements in the list and the user selects an entry.



Selecting a list element causes a postback (Click to view full-size image)

Filling a List Using CascadingDropDown (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. (For instance, one list provides a list of US states, and the next list is then filled with major cities in that state.) The first challenge to solve is to actually fill a dropdown list using this control.

Overview

The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. (For instance, one list provides a list of US states, and the next list is then filled with major cities in that state.) The first challenge to solve is to actually fill a dropdown list using this control.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, a DropDownList control is required:

```
<div>
 Vendor: <asp:DropDownList ID="VendorsList" runat="server" />
</div>
```

For this list, a CascadingDropDown extender is added. It will send an asynchronous request to a web service which will then return a list of entries to be displayed in the list. For this to work, the following CascadingDropDown attributes need to be set:

- `ServicePath`: URL of a web service delivering the list entries
- `ServiceMethod`: Web method delivering the list entries
- `TargetControlID`: ID of the dropdown list
- `Category`: Category information that is submitted to the web method when called
- `PromptText`: Text displayed when asynchronously loading list data from the server

Here is the markup for the `CascadingDropDown` element. The only difference between C# and VB is the name of the associated web service:

```
<ajaxToolkit:CascadingDropDown ID="ccd1" runat="server"
 ServicePath="CascadingDropdown0.vb.asmx" ServiceMethod="GetVendors"
 TargetControlID="VendorsList" Category="Vendor" />
```

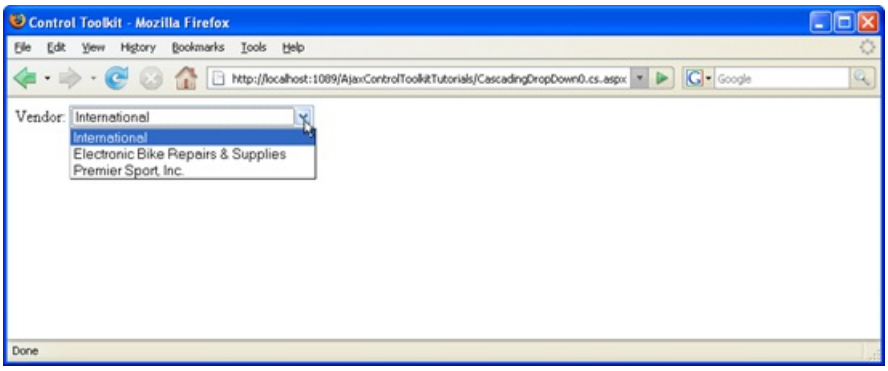
The JavaScript code coming from the `CascadingDropDown` extender calls a web service method with the following signature:

```
Public Function MethodNameHere(ByVal knownCategoryValues As String, ByVal category As String) As
CascadingDropDownNameValue()
```

So the important aspect is that the method needs to return an array of type `CascadingDropDownNameValue` (defined by the ASP.NET AJAX Control Toolkit). In the `CascadingDropDownNameValue` constructor, first the list entry's text and then its value must be provided, just as `<option value="VALUE">NAME</option>` would do in HTML. Here is some sample data:

```
<%@ WebService Language="VB" Class="CascadingDropDown0" %>
Imports System.Web.Script.Services
Imports AjaxControlToolkit
Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols
Imports System.Collections.Generic
<ScriptService()> _
Public Class CascadingDropDown0
 Inherits System.Web.Services.WebService
 <WebMethod()> _
 Public Function GetVendors(ByVal knownCategoryValues As String, ByVal category As String) As
CascadingDropDownNameValue()
 Dim l As New List(Of CascadingDropDownNameValue)
 l.Add(New CascadingDropDownNameValue("International", "1"))
 l.Add(New CascadingDropDownNameValue("Electronic Bike Repairs & Supplies","2"))
 l.Add(New CascadingDropDownNameValue("Premier Sport, Inc.", "3"))
 Return l.ToArray()
 End Function
End Class
```

Loading the page in the browser will trigger the list to be filled with three vendors.



The list is filled automatically (Click to view full-size image)

PREVIOUS NEXT

Using CascadingDropDown with a Database (VB)

2/20/2020 • 5 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. In order for this to work, a special web service must be created.

Overview

The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. (For instance, one list provides a list of US states, and the next list is then filled with major cities in that state.) In order for this to work, a special web service must be created.

Steps

First of all, a data source is required. This sample uses the AdventureWorks database and the Microsoft SQL Server 2005 Express Edition. The database is an optional part of a Visual Studio installation (including express edition) and is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. The AdventureWorks database is part of the SQL Server 2005 Samples and Sample Databases (download at https://www.microsoft.com/downloads/details.aspx?FamilyID=e719ecf7-9f46-4312-af89-6ad8702e4e6e&DisplayLang=en). The easiest way to set the database up is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en) and attach the `AdventureWorks.mdf` database file.

For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database.

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the < `form` > element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

In the next step, two DropDownList controls are required. In this sample, we use the vendor and contact information from AdventureWorks, thus we create one list for the available vendors and one for the available contacts:

```
<div>
 Vendor: <asp:DropDownList ID="VendorsList" runat="server"/><br />
 Contacts: <asp:DropDownList ID="ContactsList" runat="server"/><br />
</div>
```

Then, two CascadingDropDown extenders must be added to the page. One fills the first (vendors) list, and the other one fills the second (contacts) list. The following attributes must be set:

- `ServicePath`: URL of a web service delivering the list entries
- `ServiceMethod`: Web method delivering the list entries
- `TargetControlID`: ID of the dropdown list
- `Category`: Category information that is submitted to the web method when called
- `PromptText`: Text displayed when asynchronously loading list data from the server
- `ParentControlID`: (optional) parent dropdown list that triggers loading of the current list

Depending on the programming language used, the name of the web service in question changes, but all other attribute values are the same. Here is the CascadingDropDown element for the first dropdown list:

```
<ajaxToolkit:CascadingDropDown ID="ccd1" runat="server"
 ServicePath="CascadingDropdown1.vb.asmx" ServiceMethod="GetVendors"
 TargetControlID="VendorsList" Category="Vendor"
 PromptText="Select Vendor" />
```

The control extenders for the second list need to set the `ParentControlID` attribute so that selecting an entry in the vendors list triggers loading associated elements in the contacts list.

```
<ajaxToolkit:CascadingDropDown ID="ccd2" runat="server"
 ServicePath="CascadingDropdown1.vb.asmx" ServiceMethod="GetContactsForVendor"
 TargetControlID="ContactsList" ParentControlID="VendorsList"
 Category="Contact"
 PromptText="Select Contact" />
```

The actual work is then done in the web service, which is set up as follows. Note that the `[ScriptService]` attribute is used, otherwise ASP.NET AJAX cannot create the JavaScript proxy to access the web methods from client-side script code.

```
<%@ WebService Language="VB" Class="CascadingDropdown1" %>
Imports System.Web.Script.Services
Imports AjaxControlToolkit
Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols
Imports System.Collections.Generic
Imports System.Collections.Specialized
Imports System.Data.SqlClient
<ScriptService()> _
Public Class CascadingDropdown1
 Inherits System.Web.Services.WebService
 ' ...
End Class
```

The signature of the web methods called by CascadingDropDown is as follows:

```
Public Function MethodNameHere(ByVal knownCategoryValues As String, ByVal category As String) As
CascadingDropDownNameValue()
```

So the return value must be an array of type `CascadingDropDownNameValue` which is defined by the Control Toolkit. The `GetVendors()` method is quite easy to implement: The code connects to the AdventureWorks database and queries the first 25 vendors. The first parameter in the `CascadingDropDownNameValue` constructor is the caption of the list entry, the second one its value (value attribute in HTML's < `option` > element). Here is the code:

```
<WebMethod()> _
Public Function GetVendors(ByVal knownCategoryValues As String, ByVal category As String) As
CascadingDropDownNameValue()
 Dim conn As New SqlConnection("server=(local)\SQLEXPRESS; Integrated Security=true; Initial
Catalog=AdventureWorks")
 conn.Open()
 Dim comm As New SqlCommand( _
 "SELECT TOP 25 VendorID, Name FROM Purchasing.Vendor", conn)
 Dim dr As SqlDataReader = comm.ExecuteReader()
 Dim l As New List(Of CascadingDropDownNameValue)
 While (dr.Read())
 l.Add(New CascadingDropDownNameValue(dr("Name").ToString(),dr("VendorID").ToString()))
 End While
 conn.Close()
 Return l.ToArray()
End Function
```

Getting the associated contacts for a vendor (method name: `GetContactsForVendor()`) is a bit trickier. First of all, the vendor which has been selected in the first dropdown list must be determined. The Control Toolkit defines a helper method for that task: The `ParseKnownCategoryValuesString()` method returns a `StringDictionary` element with the dropdown data:

```
<WebMethod()> _
Public Function GetContactsForVendor(ByVal knownCategoryValues As String, ByVal category As String) As
CascadingDropDownNameValue()
 Dim VendorID As Integer
 CascadingDropDown.ParseKnownCategoryValuesString(knownCategoryValues)
```

For security reasons, this data must be validated first. So if there is a Vendor entry (because the `Category` property of the first CascadingDropDown element is set to `"Vendor"`), the ID of the selected vendor may be retrieved:

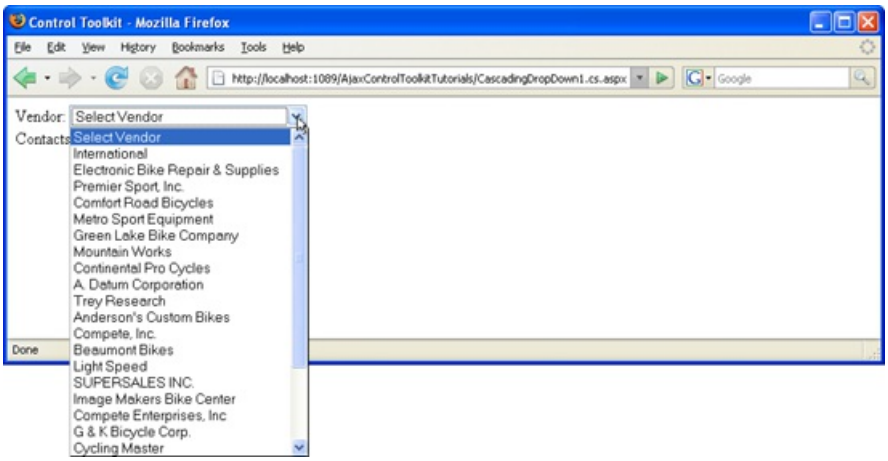
```
If Not kv.ContainsKey("Vendor") Or Not Int32.TryParse(kv("Vendor"),VendorID) Then
 Throw New ArgumentException("Couldn't find vendor.")
 End If
```

The rest of the method is fairly straight-forward, then. The vendor's ID is used as a parameter for an SQL query that retrieves all associated contacts for that vendor. Once again, the method returns an array of type `CascadingDropDownNameValue`.

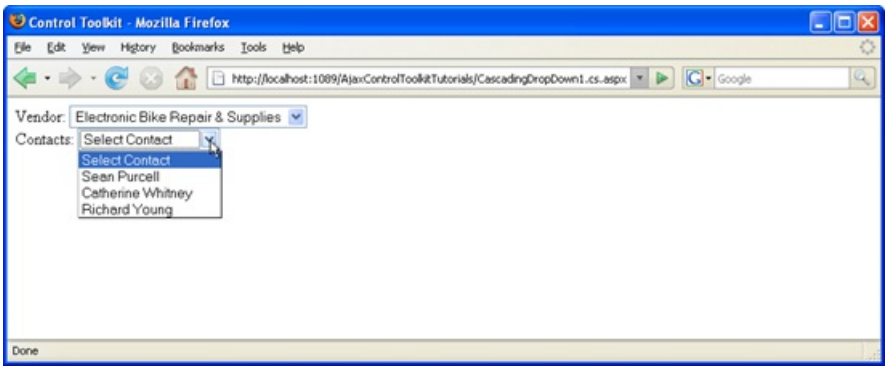
```
Dim conn As New SqlConnection("server=(local)\SQLEXPRESS; Integrated Security=true; Initial
Catalog=AdventureWorks")
 conn.Open()
 Dim comm As New SqlCommand("SELECT Person.Contact.ContactID, FirstName, LastName FROM
Person.Contact,Purchasing.VendorContact WHERE VendorID=@VendorID AND
Person.Contact.ContactID=Purchasing.VendorContact.ContactID",conn)
 comm.Parameters.AddWithValue("@VendorID", VendorID)
 Dim dr As SqlDataReader = comm.ExecuteReader()
 Dim l As New List(Of CascadingDropDownNameValue)
 While (dr.Read())
 l.Add(New CascadingDropDownNameValue(dr("FirstName").ToString() & " " &
dr("LastName").ToString(),dr("ContactID").ToString()))
 End While
 conn.Close()
 Return l.ToArray()
End Function
```

Load the ASP.NET page, and after a short while the vendor list is filled with 25 entries. Pick one entry and notice

how the second dropdown list is filled with data.



The first list is filled automatically (Click to view full-size image)



The second list is filled according to the selection in the first list (Click to view full-size image)

PREVIOUS NEXT

Presetting List Entries with CascadingDropDown (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. With a little bit of code it is possible that a list element is preselected once the data has been dynamically loaded.

Overview

The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. (For instance, one list provides a list of US states, and the next list is then filled with major cities in that state.) With a little bit of code it is possible that a list element is preselected once the data has been dynamically loaded.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, a DropDownList control is required:

```
<div>
 Vendor: <asp:DropDownList ID="VendorsList" runat="server"/>
</div>
```

For this list, a CascadingDropDown extender is added, providing web service URL and method information:

```
<ajaxToolkit:CascadingDropDown ID="ccd1" runat="server"
 ServicePath="CascadingDropdown2.vb.asmx" ServiceMethod="GetVendors"
 TargetControlID="VendorsList" Category="Vendor" />
```

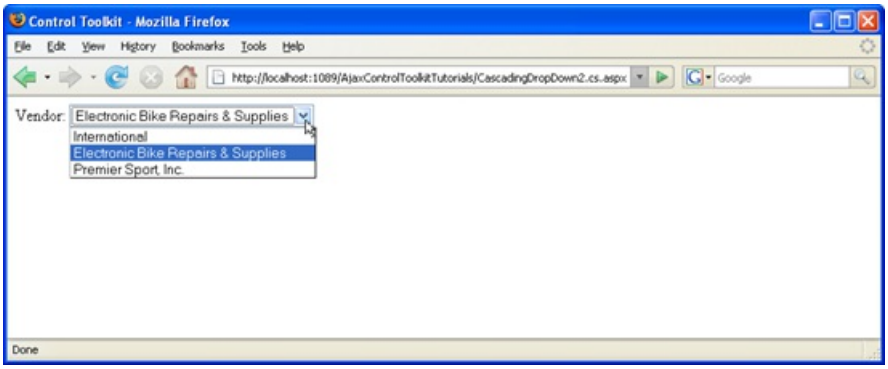
The CascadingDropDown extender then asynchronously calls a web service with the following method signature:

```
Public Function MethodNameHere(ByVal knownCategoryValues As String, ByVal category As String) As
CascadingDropDownNameValue()
```

The method returns an array of type CascadingDropDown value. The type's constructor expects first the list entry's caption and then the value (HTML `value` attribute). If the third argument is set to true, the list element is automatically selected in the browser.

```
<%@ WebService Language="VB" Class="CascadingDropDown2" %>
Imports System.Web.Script.Services
Imports AjaxControlToolkit
Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols
Imports System.Collections.Generic
<ScriptService()> _
Public Class CascadingDropDown2
 Inherits System.Web.Services.WebService
 <WebMethod()> _
 Public Function GetVendors(ByVal knownCategoryValues As String, ByVal category As
 String) As CascadingDropDownNameValue()
 Dim l As New List(Of CascadingDropDownNameValue)
 l.Add(New CascadingDropDownNameValue("International", "1"))
 l.Add(New CascadingDropDownNameValue("Electronic Bike Repairs & Supplies","2", True))
 l.Add(New CascadingDropDownNameValue("Premier Sport, Inc.", "3"))
 Return l.ToArray()
 End Function
End Class
```

Loading the page in the browser will fill the dropdown list with three vendors, the second one being preselected.



The list is filled and preselected automatically (Click to view full-size image)

PREVIOUS NEXT

Using Auto-Postback with CascadingDropDown (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. However when using the CascadingDropDown control, ASP.NET's DropDownList control's AutoPostBack feature does not work, since asynchronously loading data into the list generates an (unnecessary) postback itself. With some JavaScript code, this effect can be avoided.

Overview

The CascadingDropDown control in the AJAX Control Toolkit extends a DropDownList control so that changes in one DropDownList loads associated values in another DropDownList. (For instance, one list provides a list of US states, and the next list is then filled with major cities in that state.) However when using the CascadingDropDown control, ASP.NET's DropDownList control's AutoPostBack feature does not work, since asynchronously loading data into the list generates an (unnecessary) postback itself. With some JavaScript code, this effect can be avoided.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the < `form` > element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, a DropDownList control is required:

```
<div>
 Vendor: <asp:DropDownList ID="VendorsList" runat="server"/>
</div>
```

For this list, a CascadingDropDown extender is added, providing web service URL and method information:

```
<ajaxToolkit:CascadingDropDown ID="ccd1" runat="server"
 ServicePath="CascadingDropdown3.vb.asmx" ServiceMethod="GetVendors"
 TargetControlID="VendorsList" Category="Vendor" />
```

The CascadingDropDown extender then asynchronously calls a web service with the following method signature:

```
Public Function MethodNameHere(ByVal knownCategoryValues As String, ByVal category As String) As
CascadingDropDownNameValue()
```

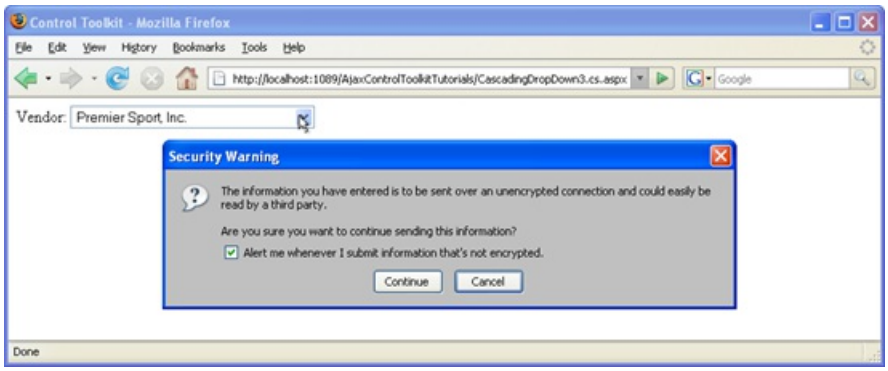
The method returns an array of type CascadingDropDown value. The type's constructor expects first the list entry's caption and then the value (HTML `value` attribute).

```
<%@ WebService Language="VB" Class="CascadingDropDown3" %>
Imports System.Web.Script.Services
Imports AjaxControlToolkit
Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols
Imports System.Collections.Generic
<ScriptService()> _
Public Class CascadingDropDown3
 Inherits System.Web.Services.WebService
 <WebMethod()> _
 Public Function GetVendors(ByVal knownCategoryValues As String, ByVal category As String) As
CascadingDropDownNameValue()
 Dim l As New List(Of CascadingDropDownNameValue)
 l.Add(New CascadingDropDownNameValue("International", "1"))
 l.Add(New CascadingDropDownNameValue("Electronic Bike Repairs & Supplies","2"))
 l.Add(New CascadingDropDownNameValue("Premier Sport, Inc.", "3"))
 Return l.ToArray()
 End Function
End Class
```

Loading the page in the browser will fill the dropdown list with three vendors, the second one being preselected. Also, ASP.NET defines the `__doPostBack()` JavaScript method. Once the page has been loaded, this JavaScript call is added to the dropdown list, but only if there are elements in it. If there are no elements in the list, the Control Toolkit is currently loading them, so the JavaScript code uses a timeout and tries again in a half second.

```
<script type="text/javascript">
 function pageLoad()
 {
 addAutoPostBack();
 }
 function addAutoPostBack()
 {
 if ($get("VendorsList").options.length > 0)
 {
 $get("VendorsList").setAttribute("onchange","javascript:setTimeout('__doPostBack(\\'VendorsList\\',\\'\\')',
0)");
 }
 else
 {
 setTimeout("addAutoPostBack()", 500);
 }
 }
</script>
```

This way, a postback is only executed when there are actually elements in the list and the user selects an entry.



Selecting a list element causes a postback (Click to view full-size image)

PREVIOUS

CollapsiblePanel

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the CollapsiblePanel control to let users expand and collapse areas on the page.

- Collapsing and Expanding a Panel from JavaScript (C#)
- Collapsing and Expanding a Panel from JavaScript (VB)

Collapsing and Expanding a Panel from JavaScript (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The CollapsiblePanel control in the ASP.NET AJAX Control Toolkit extends a panel and provides it with the capability to collapse its contents and expand it again. These two actions can also be triggered from custom JavaScript code.

Overview

The CollapsiblePanel control in the ASP.NET AJAX Control Toolkit extends a panel and provides it with the capability to collapse its contents and expand it again. These two actions can also be triggered from custom JavaScript code.

Steps

First of all, create a new ASP.NET page and include the `ScriptManager` within the one `<form>` element. This loads the ASP.NET AJAX library which is required by the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, create a panel with some text so that the collapse/expand effect can be seen:

```
<asp:Panel ID="Panel1" CssClass="panelClass" runat="server">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

As you can see, the panel references a CSS class which is shown here (and basically defines a background color and the panel's width):

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

The `CollapsiblePanelExtender` control requires the `TargetControlID` attribute so that the toolkit knows which panel to collapse or expand upon request:

```
<ajaxToolkit:CollapsiblePanelExtender ID="cpe" runat="server"
 TargetControlID="Panel1" />
```

Unfortunately, the extender currently does not expose a specific API for collapsing or expanding the panel, but some undocumented methods will do. First of all, add three HTML buttons to the page which will then trigger the client-side JavaScript to collapse or expand the panel's contents:

```
<input type="button" id="Button1" runat="server" value="Open" onclick="doOpen();" />
<input type="button" id="Button2" runat="server" value="Close" onclick="doClose();" />
<input type="button" id="Button3" runat="server" value="Toggle" onclick="doToggle();" />
```

In the client-side JavaScript code (started with `<script type="text/javascript">`), the `$find()` method needs to be used to access the `CollapsiblePanelExtender`. `$find("cpe")` will return a reference to it. From there on, specific methods will solve the task at hand.

The method for opening (expanding) the panel is called `_doOpen()`; the following code implements the `doOpen()` function called when the first button is clicked:

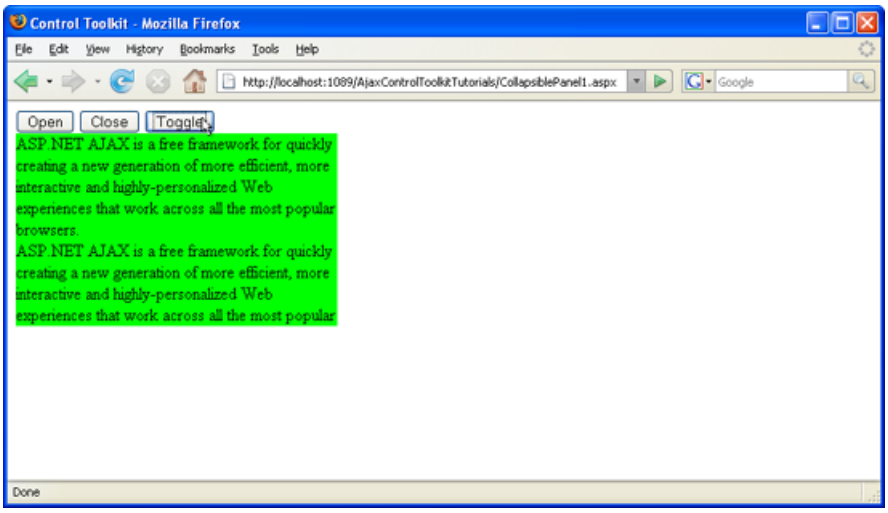
```
function doOpen()
{
 $find("cpe")._doOpen();
}
```

For closing, or collapsing the panel, the `_doClose()` method needs to be executed. So when the user clicks on the second button, the following JavaScript code is called:

```
function doClose()
{
 $find("cpe")._doClose();
}
```

The third button toggles the state of the panel: from collapsed to expanded, and vice versa. The `CollapsiblePanelExtender` exposes the `toggle()` method which does exactly that: reverses the state of the panel. However there is also another approach (which is internally used by the `toggle()` method): The `get_Collapsed()` method of the `CollapsiblePanelExtender()` tells us whether the panel is collapsed or not. Depending on the return value of this function, the panel is then either expanded (`_doOpen()` method) or collapsed (`_doClose()`) method:

```
function doToggle()
{
 var cpe = $find("cpe");
 //cpe._toggle();
 if (cpe.get_Collapsed()) {
 cpe._doOpen();
 } else {
 cpe._doClose();
 }
}
```



The third button changes the state of the panel: from collapsed to expanded and back (Click to view full-size image)

NEXT

Collapsing and Expanding a Panel from JavaScript (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The CollapsiblePanel control in the ASP.NET AJAX Control Toolkit extends a panel and provides it with the capability to collapse its contents and expand it again. These two actions can also be triggered from custom JavaScript code.

Overview

The CollapsiblePanel control in the ASP.NET AJAX Control Toolkit extends a panel and provides it with the capability to collapse its contents and expand it again. These two actions can also be triggered from custom JavaScript code.

Steps

First of all, create a new ASP.NET page and include the `ScriptManager` within the one `<form>` element. This loads the ASP.NET AJAX library which is required by the Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, create a panel with some text so that the collapse/expand effect can be seen:

```
<asp:Panel ID="Panel1" CssClass="panelClass" runat="server">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

As you can see, the panel references a CSS class which is shown here (and basically defines a background color and the panel's width):

```
<style type="text/css">
 .panelClass {background-color: lime; width: 300px;}
</style>
```

The `CollapsiblePanelExtender` control requires the `TargetControlID` attribute so that the toolkit knows which panel to collapse or expand upon request:

```
<ajaxToolkit:CollapsiblePanelExtender ID="cpe" runat="server"
 TargetControlID="Panel1" />
```

Unfortunately, the extender currently does not expose a specific API for collapsing or expanding the panel, but some undocumented methods will do. First of all, add three HTML buttons to the page which will then trigger the client-side JavaScript to collapse or expand the panel's contents:

```
<input type="button" id="Button1" runat="server" value="Open" onclick="doOpen();" />
<input type="button" id="Button2" runat="server" value="Close" onclick="doClose();" />
<input type="button" id="Button3" runat="server" value="Toggle" onclick="doToggle();" />
```

In the client-side JavaScript code (started with `<script type="text/javascript">`), the `$find()` method needs to be used to access the `CollapsiblePanelExtender`. `$find("cpe")` will return a reference to it. From there on, specific methods will solve the task at hand.

The method for opening (expanding) the panel is called `_doOpen()`; the following code implements the `doOpen()` function called when the first button is clicked:

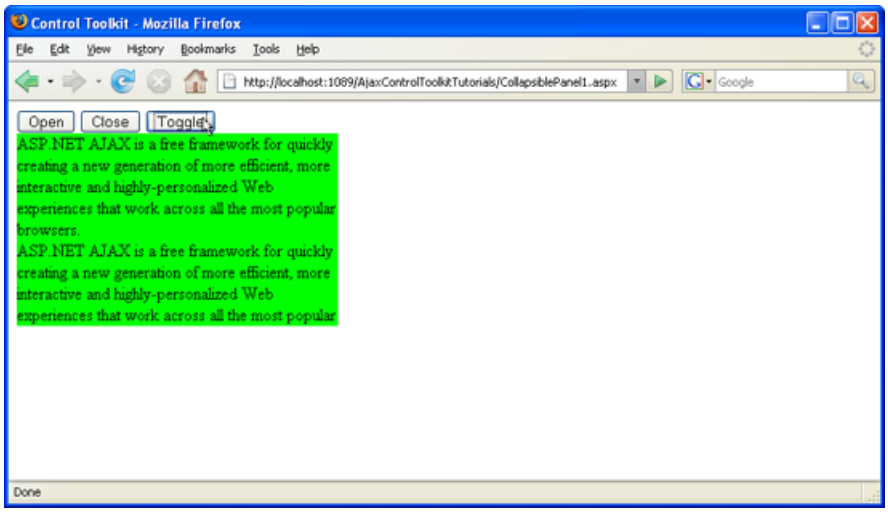
```
function doOpen()
{
 $find("cpe")._doOpen();
}
```

For closing, or collapsing the panel, the `_doClose()` method needs to be executed. So when the user clicks on the second button, the following JavaScript code is called:

```
function doClose()
{
 $find("cpe")._doClose();
}
```

The third button toggles the state of the panel: from collapsed to expanded, and vice versa. The `CollapsiblePanelExtender` exposes the `toggle()` method which does exactly that: reverses the state of the panel. However there is also another approach (which is internally used by the `toggle()` method): The `get_Collapsed()` method of the `CollapsiblePanelExtender()` tells us whether the panel is collapsed or not. Depending on the return value of this function, the panel is then either expanded (`_doOpen()` method) or collapsed (`_doClose()`) method:

```
function doToggle()
{
 var cpe = $find("cpe");
 //cpe._toggle();
 if (cpe.get_Collapsed()) {
 cpe._doOpen();
 } else {
 cpe._doClose();
 }
}
```



The third button changes the state of the panel: from collapsed to expanded and back (Click to view full-size image)

PREVIOUS

ColorPicker

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the ColorPicker control, which lets users select a color from a popup window.

- ColorPicker Control Extender (C#)
- ColorPicker Control Extender (VB)

Using the ColorPicker Control Extender (C#)

4/10/2020 • 6 minutes to read • Edit Online

by Microsoft

> ColorPicker is an ASP.NET AJAX extender that provides client-side color-picking functionality with UI in a popup control. It can be attached to any ASP.NET TextBox control. It.

The goal of this tutorial is to explain how you can use the AJAX Control Toolkit ColorPicker control extender. The ColorPicker control extender displays a popup dialog that enables you to select a color. The ColorPicker is useful whenever you want to provide an intuitive user interface for a user to pick a color.

Extending a TextBox Control with the ColorPicker Control Extender

Imagine, for example, that you want to create a website that enables visitors to create customized business cards. Visitors can enter the text for a business card and pick the color. The ASP.NET page in Listing 1 contains two TextBox controls named txtCardText and txtCardColor. When you submit the form, the selected values are displayed (see Figure 1).

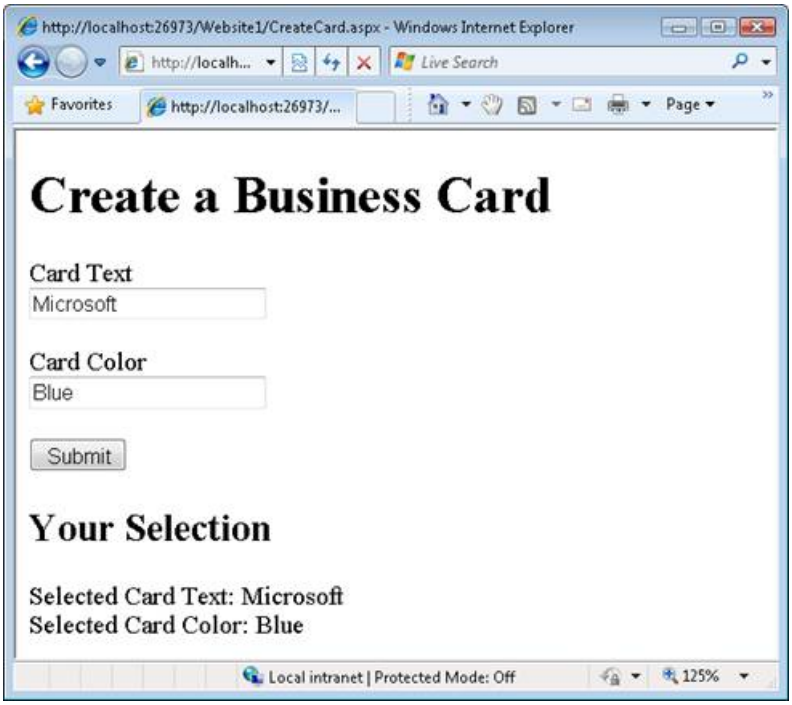


Figure 01: Simple form for creating a business card (Click to view full-size image)

Listing 1 - CreateCard.aspx

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

    protected void btnSubmit_Click(object sender, EventArgs e)
    {
        // Show the panel
        pnlResults.Visible = true;

        // Show the selected values
```

```
        lblSelectedText.Text = txtCardText.Text;
        lblSelectedColor.Text = txtCardColor.Text;
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        <h1>Create a Business Card</h1>

        <asp:Label
            ID="lblCardText"
            Text="Card Text"
            AssociatedControlID="txtCardText"
            runat="server" />
        <br />
        <asp:TextBox
            ID="txtCardText"
            Runat="server" />

        <br /><br />

        <asp:Label
            ID="lblCardColor"
            Text="Card Color"
            AssociatedControlID="txtCardColor"
            runat="server" />
        <br />
        <asp:TextBox
            ID="txtCardColor"
            Runat="server" />

        <br /><br />

        <asp:Button
            ID="btnSubmit"
            Text="Submit"
            Runat="server" OnClick="btnSubmit_Click" />

        <asp:Panel ID="pnlResults" Visible="false" runat="server">

            <h2>Your Selection</h2>

            Selected Card Text:
            <asp:Label
                ID="lblSelectedText"
                Runat="server" />
            <br />
            Selected Card Color:
            <asp:Label
                ID="lblSelectedColor"
                Runat="server" />

        </asp:Panel>

    </div>
    </form>
</body>
</html>
```

The form in Listing 1 works, but it does not provide a great user experience. The user has to type a color into the textbox. If the user wants a specialized color - for example, just the right shade of pea green - then the user must

figure out the HTML color code without any help.

You can use the ColorPicker control extender to create a better user experience. The ColorPicker displays a color dialog when you move focus to a TextBox control (see Figure 2).

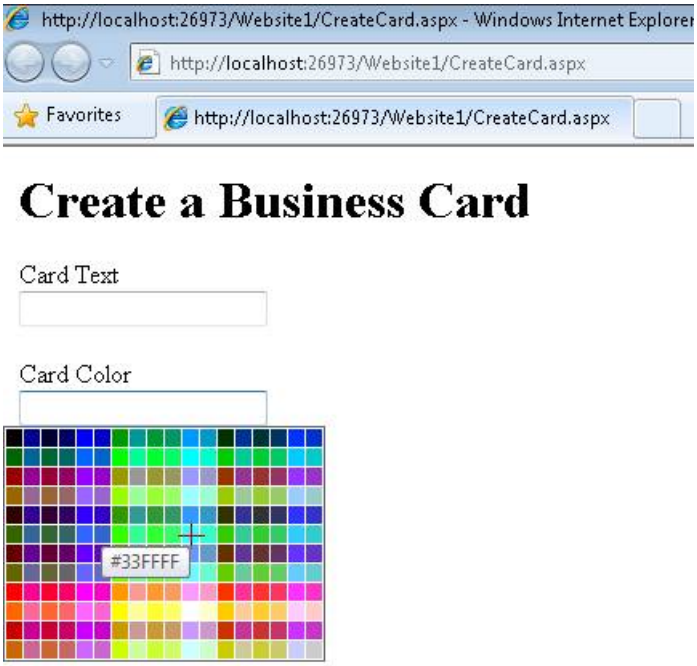


Figure 02: The ColorPicker Control Extender (Click to view full-size image)

You need to complete two steps to use the ColorPicker control extender with the form in Listing 1:

1. Add a ScriptManager control to the page
2. Add the ColorPicker control extender to the page

Before you can use the ColorPicker, you must add a ScriptManager to your page. A good place to add the ScriptManager is right below the opening server-side <form> tag. You can drag the ScriptManager onto the page from the toolbox (the ScriptManager is located under the AJAX Extensions tab). Alternatively, you can type the following tag into Source View beneath the opening server-side form tag:

```
<asp:ScriptManager ID="ScriptManager1" runat="server" />
```

The easiest way to add the ColorPicker control extender to the page is in Design View. If you hover your mouse over the txtCardColor TextBox, a smart task option appears the enables you to add an extender (see Figure 3). If you pick this option, the Extender Wizard appears (see Figure 4).

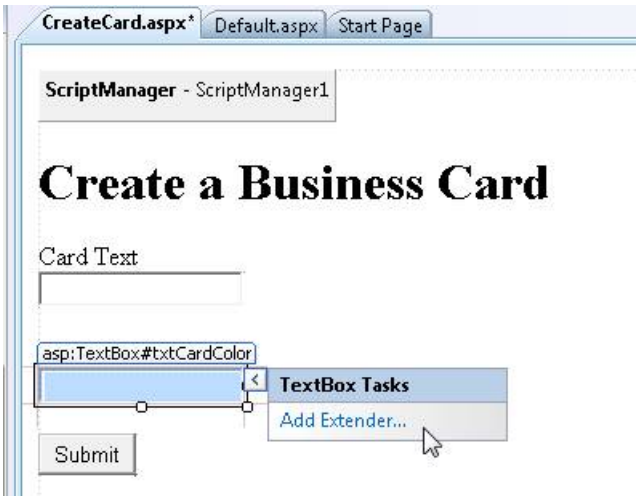


Figure 03: Adding an extender (Click to view full-size image)

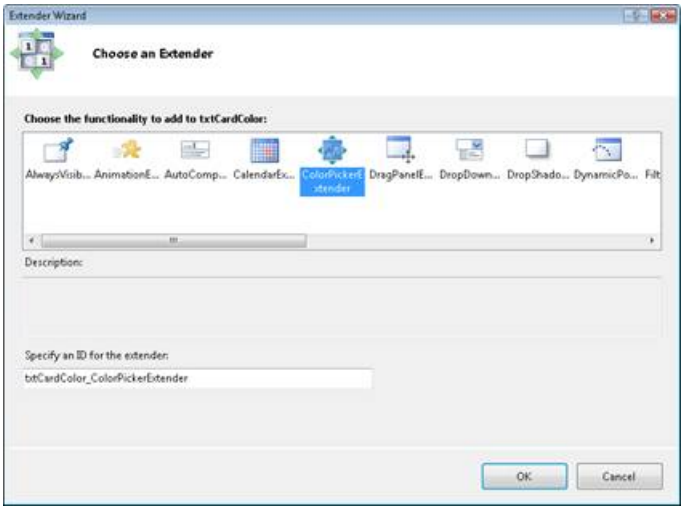


Figure 04: Selecting a control extender with the Extender Wizard (Click to view full-size image)

You can pick the ColorPicker extender to extend the txtCardColor TextBox with the ColorPicker extender. Click OK to close the dialog.

After you make these changes, the source for the page looks like Listing 2.

Listing 2 - CreateCard.aspx (with ColorPicker)

```
<%@ Page Language="C#" %>

<%@ Register assembly="AjaxControlToolkit" namespace="AjaxControlToolkit" tagprefix="cc1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<script runat="server">

    protected void btnSubmit_Click(object sender, EventArgs e)
    {
        // Show the panel
        pnlResults.Visible = true;

        // Show the selected values
        lblSelectedText.Text = txtCardText.Text;
        lblSelectedColor.Text = txtCardColor.Text;
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:ScriptManager ID="ScriptManager1" runat="server" />

        <h1>Create a Business Card</h1>

        <asp:Label
            ID="lblCardText"
            Text="Card Text"
            AssociatedControlID="txtCardText"
            runat="server" />
        <br />
        <asp:TextBox
            ID="txtCardText"
```

```
            ID="txtCardText"
            Runat="server" />

        <br /><br />

        <asp:Label
            ID="lblCardColor"
            Text="Card Color"
            AssociatedControlID="txtCardColor"
            runat="server" />
        <br />
        <asp:TextBox
            ID="txtCardColor"
            AutoCompleteType="None"
            Runat="server" />

        <cc1:ColorPickerExtender
            ID="txtCardColor_ColorPickerExtender"
            TargetControlID="txtCardColor"
            Enabled="True"
            runat="server">
        </cc1:ColorPickerExtender>

        <br /><br />

        <asp:Button
            ID="btnSubmit"
            Text="Submit"
            Runat="server" OnClick="btnSubmit_Click" />

        <asp:Panel ID="pnlResults" Visible="false" runat="server">

            <h2>Your Selection</h2>

            Selected Card Text:
            <asp:Label
                ID="lblSelectedText"
                Runat="server" />
            <br />
            Selected Card Color:
            <asp:Label
                ID="lblSelectedColor"
                Runat="server" />

        </asp:Panel>

    </div>
    </form>
</body>
</html>
```

Notice that the page now contains a ColorPickerExtender control that appears directly below the txtCardColor TextBox control. The ColorPickerExtender control extends the txtCardColor control so that it displays a color picker dialog.

Using a Button to Launch the Color Picker Dialog

The ColorPicker extender supports the following properties:

- PopupButtonId - The ID of a button on the page that causes the color picker dialog to appear.
- PopupPosition - The position, relative to the target control, of the color picker dialog. Possible values are Absolute, Center, BottomLeft, BottomRight, TopLeft, TopRight, Right, and Left (the default is BottomLeft).
- SampleControlId - The ID of a control that displays the selected color.
- SelectedColor - The initial color selected by the ColorPicker.

You can use these properties to customize how the color picker dialog is displayed and how the selected color is displayed. The page in Listing 3 illustrates how you can use several of these properties.

Listing 3 - CreateCardButton.aspx

```
<%@ Page Language="C#" %>

<%@ Register assembly="AjaxControlToolkit" namespace="AjaxControlToolkit" tagprefix="cc1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<script runat="server">

    protected void btnSubmit_Click(object sender, EventArgs e)
    {
        // Show the panel
        pnlResults.Visible = true;

        // Show the selected values
        lblSelectedText.Text = txtCardText.Text;
        lblSelectedColor.Text = txtCardColor.Text;
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:ScriptManager ID="ScriptManager1" runat="server" />

        <h1>Create a Business Card</h1>

        <asp:Label
            ID="lblCardText"
            Text="Card Text"
            AssociatedControlID="txtCardText"
            runat="server" />
        <br />
        <asp:TextBox
            ID="txtCardText"
            Runat="server" />

        <br /><br />

        <asp:Label
            ID="lblCardColor"
            Text="Card Color"
            AssociatedControlID="txtCardColor"
            runat="server" />
        <br />
        <asp:TextBox
            ID="txtCardColor"
            AutoCompleteType="None"
            Runat="server" />
        <asp:Button
            ID="btnPickColor"
            Text="Pick Color"
            Runat="server" />
        <asp:Label
            ID="lblSample"
            Runat="Server"> Sample </asp:Label>
        <cc1:ColorPickerExtender
            ID="txtCardColor_ColorPickerExtender"
```

```
            TargetControlID="txtCardColor"
            PopupButtonID="btnPickColor"
            PopupPosition="TopRight"
            SampleControlID="lblSample"
            Enabled="True"
            runat="server">
        </cc1:ColorPickerExtender>

        <br /><br />

        <asp:Button
            ID="btnSubmit"
            Text="Submit"
            Runat="server" OnClick="btnSubmit_Click" />

        <asp:Panel ID="pnlResults" Visible="false" runat="server">

            <h2>Your Selection</h2>

            Selected Card Text:
            <asp:Label
                ID="lblSelectedText"
                Runat="server" />
            <br />
            Selected Card Color:
            <asp:Label
                ID="lblSelectedColor"
                Runat="server" />

        </asp:Panel>

    </div>
    </form>
</body>
</html>
```

The page in Listing 3 includes a Pick Color button (see Figure 5). When you click this button, the color picker dialog appears above the TextBox. If you select a color from the dialog then the selected color appears as the background color of the lblSample Label control.

The ColorPicker PopupButtonID property is used to associate the Pick Color button with the ColorPicker extender. When you supply a value for the PopupButtonID property, the color picker dialog no longer appears when the target control has focus. You must click the button to display the dialog.

The SampleControlID property is used to associate a control that displays the selected color with the ColorPicker. The ColorPicker changes the background color of this control to the currently selected color.

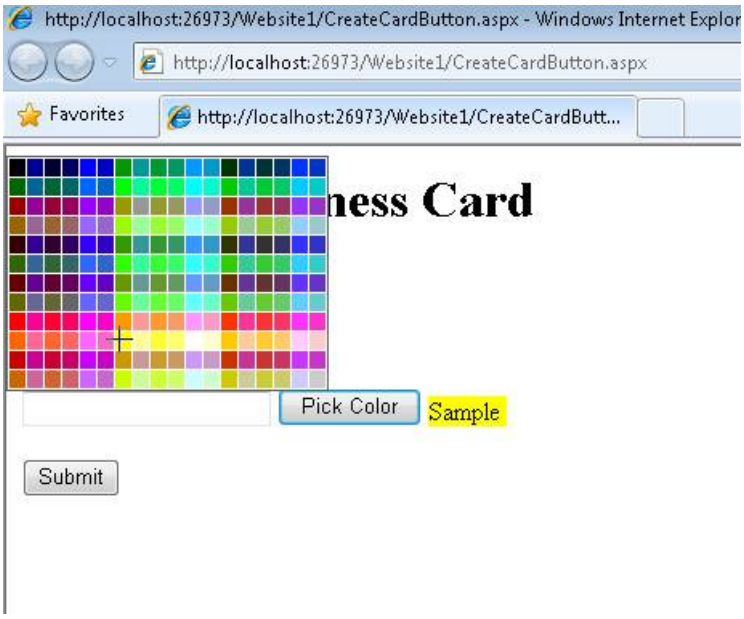


Figure 05: Displaying the color picker dialog with a button (Click to view full-size image)

Summary

In this tutorial, you learned how to use the ColorPicker control extender to display a popup color picker dialog. First, we examined how you can display the dialog when focus is moved to a TextBox control. Next, you learned how to create a button that displays the color picker dialog when the button is clicked.

NEXT

Using the ColorPicker Control Extender (VB)

4/10/2020 • 6 minutes to read • Edit Online

by Microsoft

> ColorPicker is an ASP.NET AJAX extender that provides client-side color-picking functionality with UI in a popup control. It can be attached to any ASP.NET TextBox control. It.

The goal of this tutorial is to explain how you can use the AJAX Control Toolkit ColorPicker control extender. The ColorPicker control extender displays a popup dialog that enables you to select a color. The ColorPicker is useful whenever you want to provide an intuitive user interface for a user to pick a color.

Extending a TextBox Control with the ColorPicker Control Extender

Imagine, for example, that you want to create a website that enables visitors to create customized business cards. Visitors can enter the text for a business card and pick the color. The ASP.NET page in Listing 1 contains two TextBox controls named txtCardText and txtCardColor. When you submit the form, the selected values are displayed (see Figure 1).

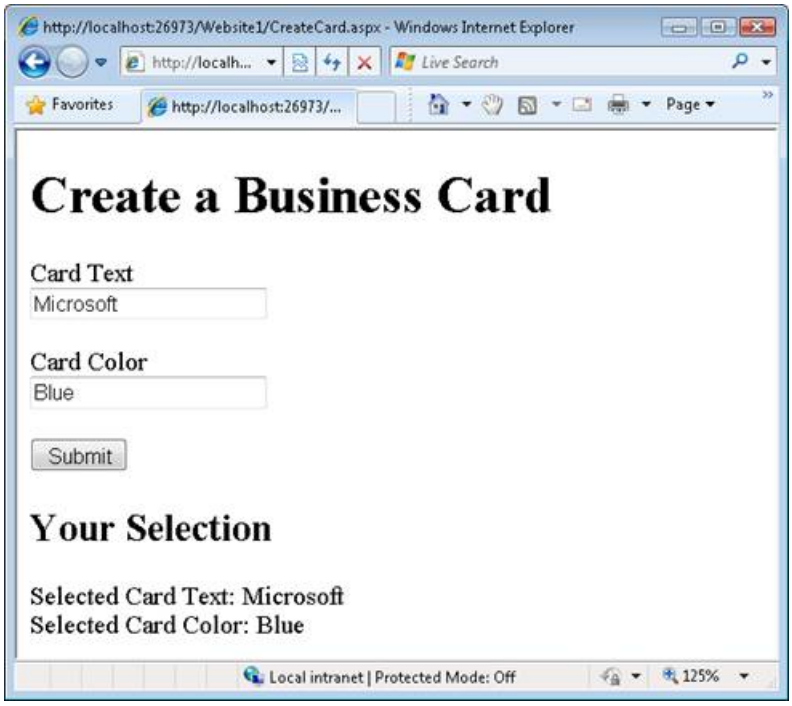


Figure 01: Simple form for creating a business card (Click to view full-size image)

Listing 1 - CreateCard.aspx

```
<%@ Page Language="VB" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

    Protected Sub btnSubmit_Click(ByVal sender As Object, ByVal e As System.EventArgs)
        ' Show the panel
        pnlResults.Visible = true

        ' Show the selected values
```

```
        lblSelectedText.Text = txtCardText.Text
        lblSelectedColor.Text = txtCardColor.Text
    End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        <h1>Create a Business Card</h1>

        <asp:Label
            ID="lblCardText"
            Text="Card Text"
            AssociatedControlID="txtCardText"
            runat="server" />
        <br />
        <asp:TextBox
            ID="txtCardText"
            Runat="server" />

        <br /><br />

        <asp:Label
            ID="lblCardColor"
            Text="Card Color"
            AssociatedControlID="txtCardColor"
            runat="server" />
        <br />
        <asp:TextBox
            ID="txtCardColor"
            Runat="server" />

        <br /><br />

        <asp:Button
            ID="btnSubmit"
            Text="Submit"
            Runat="server" OnClick="btnSubmit_Click" />

        <asp:Panel ID="pnlResults" Visible="false" runat="server">

            <h2>Your Selection</h2>

            Selected Card Text:
            <asp:Label
                ID="lblSelectedText"
                Runat="server" />
            <br />
            Selected Card Color:
            <asp:Label
                ID="lblSelectedColor"
                Runat="server" />

        </asp:Panel>

    </div>
    </form>
</body>
</html>
```

The form in Listing 1 works, but it does not provide a great user experience. The user has to type a color into the textbox. If the user wants a specialized color - for example, just the right shade of pea green - then the user must

figure out the HTML color code without any help.

You can use the ColorPicker control extender to create a better user experience. The ColorPicker displays a color dialog when you move focus to a TextBox control (see Figure 2).

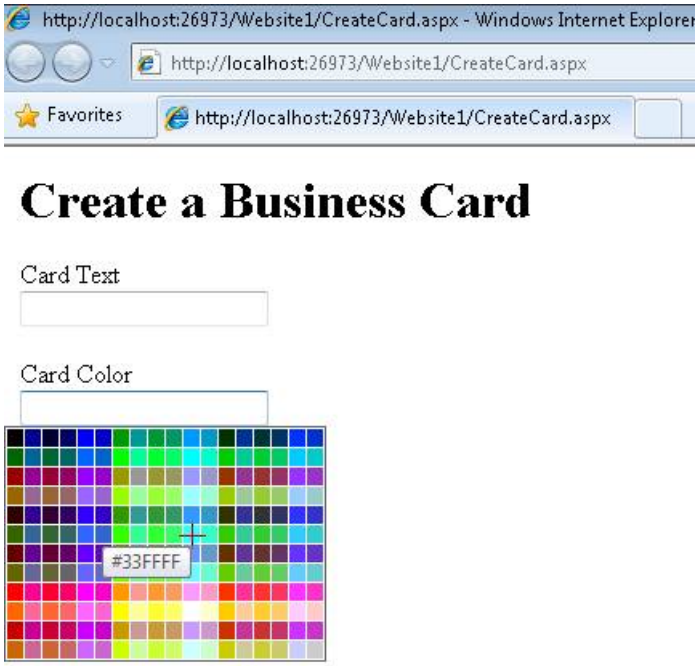


Figure 02: The ColorPicker Control Extender (Click to view full-size image)

You need to complete two steps to use the ColorPicker control extender with the form in Listing 1:

1. Add a ScriptManager control to the page
2. Add the ColorPicker control extender to the page

Before you can use the ColorPicker, you must add a ScriptManager to your page. A good place to add the ScriptManager is right below the opening server-side <form> tag. You can drag the ScriptManager onto the page from the toolbox (the ScriptManager is located under the AJAX Extensions tab). Alternatively, you can type the following tag into Source View beneath the opening server-side form tag:

```
<asp:ScriptManager ID="ScriptManager1" runat="server" />
```

The easiest way to add the ColorPicker control extender to the page is in Design View. If you hover your mouse over the txtCardColor TextBox, a smart task option appears the enables you to add an extender (see Figure 3). If you pick this option, the Extender Wizard appears (see Figure 4).

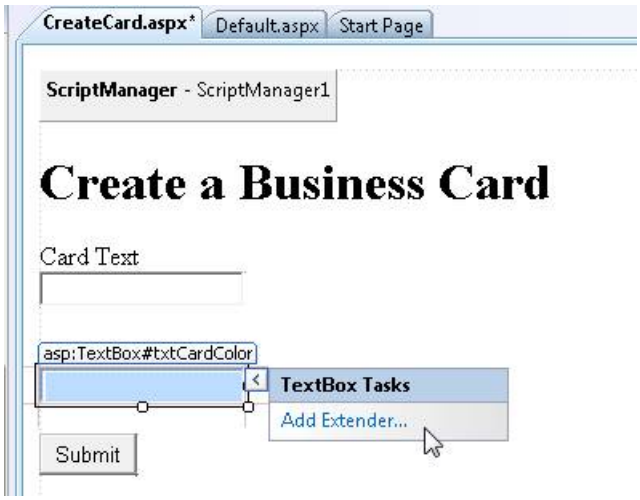


Figure 03: Adding an extender (Click to view full-size image)

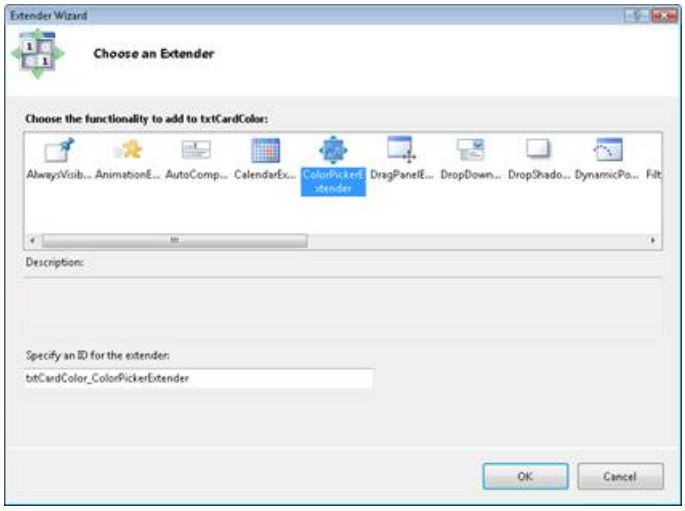


Figure 04: Selecting a control extender with the Extender Wizard (Click to view full-size image)

You can pick the ColorPicker extender to extend the txtCardColor TextBox with the ColorPicker extender. Click OK to close the dialog.

After you make these changes, the source for the page looks like Listing 2.

Listing 2 - CreateCard.aspx (with ColorPicker)

```
<%@ Page Language="VB" %>

<%@ Register assembly="AjaxControlToolkit" namespace="AjaxControlToolkit" tagprefix="cc1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<script runat="server">

    Protected Sub btnSubmit_Click(ByVal sender As Object, ByVal e As System.EventArgs)
        ' Show the panel
        pnlResults.Visible = true

        ' Show the selected values
        lblSelectedText.Text = txtCardText.Text
        lblSelectedColor.Text = txtCardColor.Text
    End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:ScriptManager ID="ScriptManager1" runat="server" />

        <h1>Create a Business Card</h1>

        <asp:Label
            ID="lblCardText"
            Text="Card Text"
            AssociatedControlID="txtCardText"
            runat="server" />
        <br />
        <asp:TextBox
            ID="txtCardText"
            Runat="server" />
```

```
            runat="server" />

        <br /><br />

        <asp:Label
            ID="lblCardColor"
            Text="Card Color"
            AssociatedControlID="txtCardColor"
            runat="server" />
        <br />
        <asp:TextBox
            ID="txtCardColor"
            AutoCompleteType="None"
            Runat="server" />

        <cc1:ColorPickerExtender
            ID="txtCardColor_ColorPickerExtender"
            TargetControlID="txtCardColor"
            Enabled="True"
            runat="server">
        </cc1:ColorPickerExtender>

        <br /><br />

        <asp:Button
            ID="btnSubmit"
            Text="Submit"
            Runat="server" OnClick="btnSubmit_Click" />

        <asp:Panel ID="pnlResults" Visible="false" runat="server">

            <h2>Your Selection</h2>

            Selected Card Text:
            <asp:Label
                ID="lblSelectedText"
                Runat="server" />
            <br />
            Selected Card Color:
            <asp:Label
                ID="lblSelectedColor"
                Runat="server" />

        </asp:Panel>

    </div>
    </form>
</body>
</html>
```

Notice that the page now contains a ColorPickerExtender control that appears directly below the txtCardColor TextBox control. The ColorPickerExtender control extends the txtCardColor control so that it displays a color picker dialog.

Using a Button to Launch the Color Picker Dialog

The ColorPicker extender supports the following properties:

- PopupButtonId - The ID of a button on the page that causes the color picker dialog to appear.
- PopupPosition - The position, relative to the target control, of the color picker dialog. Possible values are Absolute, Center, BottomLeft, BottomRight, TopLeft, TopRight, Right, and Left (the default is BottomLeft).
- SampleControlId - The ID of a control that displays the selected color.
- SelectedColor - The initial color selected by the ColorPicker.

You can use these properties to customize how the color picker dialog is displayed and how the selected color is

displayed. The page in Listing 3 illustrates how you can use several of these properties.

Listing 3 - CreateCardButton.aspx

```
<%@ Page Language="VB" %>

<%@ Register assembly="AjaxControlToolkit" namespace="AjaxControlToolkit" tagprefix="cc1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<script runat="server">

    Protected Sub btnSubmit_Click(ByVal sender As Object, ByVal e As System.EventArgs)
        ' Show the panel
        pnlResults.Visible = true

        ' Show the selected values
        lblSelectedText.Text = txtCardText.Text
        lblSelectedColor.Text = txtCardColor.Text
    End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:ScriptManager ID="ScriptManager1" runat="server" />

        <h1>Create a Business Card</h1>

        <asp:Label
            ID="lblCardText"
            Text="Card Text"
            AssociatedControlID="txtCardText"
            runat="server" />
        <br />
        <asp:TextBox
            ID="txtCardText"
            Runat="server" />

        <br /><br />

        <asp:Label
            ID="lblCardColor"
            Text="Card Color"
            AssociatedControlID="txtCardColor"
            runat="server" />
        <br />
        <asp:TextBox
            ID="txtCardColor"
            AutoCompleteType="None"
            Runat="server" />
        <asp:Button
            ID="btnPickColor"
            Text="Pick Color"
            Runat="server" />
        <asp:Label
            ID="lblSample"
            Runat="Server"> Sample </asp:Label>
        <cc1:ColorPickerExtender
            ID="txtCardColor_ColorPickerExtender"
            TargetControlID="txtCardColor"
            PopupButtonID="btnPickColor"
            PopupPosition="TopRight"
```

```
            PopupPosition="TopRight"
            SampleControlID="lblSample"
            Enabled="True"
            runat="server">
        </cc1:ColorPickerExtender>

        <br /><br />

        <asp:Button
            ID="btnSubmit"
            Text="Submit"
            Runat="server" OnClick="btnSubmit_Click" />

        <asp:Panel ID="pnlResults" Visible="false" runat="server">

            <h2>Your Selection</h2>

            Selected Card Text:
            <asp:Label
                ID="lblSelectedText"
                Runat="server" />
            <br />
            Selected Card Color:
            <asp:Label
                ID="lblSelectedColor"
                Runat="server" />

        </asp:Panel>

    </div>
    </form>
</body>
</html>
```

The page in Listing 3 includes a Pick Color button (see Figure 5). When you click this button, the color picker dialog appears above the TextBox. If you select a color from the dialog then the selected color appears as the background color of the lblSample Label control.

The ColorPicker PopupButtonID property is used to associate the Pick Color button with the ColorPicker extender. When you supply a value for the PopupButtonID property, the color picker dialog no longer appears when the target control has focus. You must click the button to display the dialog.

The SampleControlID property is used to associate a control that displays the selected color with the ColorPicker. The ColorPicker changes the background color of this control to the currently selected color.

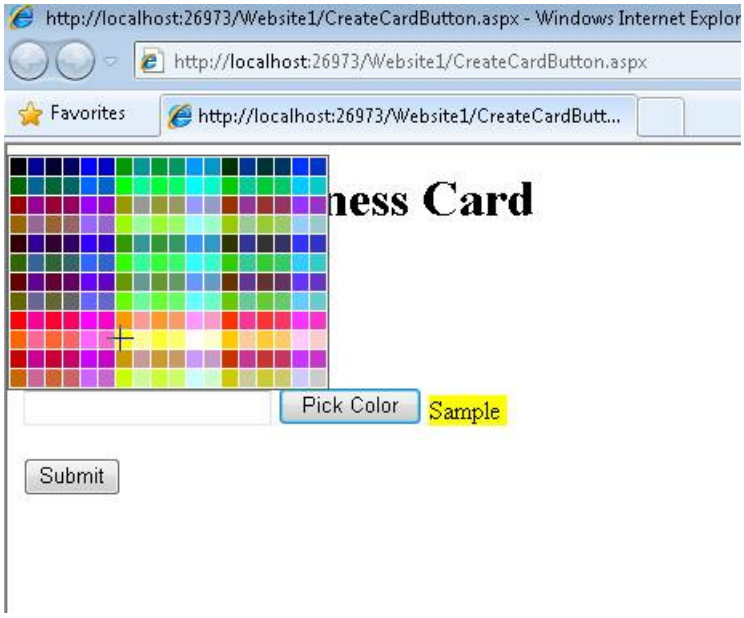


Figure 05: Displaying the color picker dialog with a button (Click to view full-size image)

Summary

In this tutorial, you learned how to use the ColorPicker control extender to display a popup color picker dialog. First, we examined how you can display the dialog when focus is moved to a TextBox control. Next, you learned how to create a button that displays the color picker dialog when the button is clicked.

PREVIOUS

ComboBox

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the ComboBox control, which combines the flexibility of a text box with a list of options that users can choose from.

- ComboBox Control (C#)
- ComboBox Control (VB)

How do I use the ComboBox Control? (C#)

9/16/2020 • 8 minutes to read • Edit Online

by Microsoft

> ComboBox is an ASP.NET AJAX control that combines the flexibility of a TextBox with a list of options from which users can choose.

The goal of this tutorial is to explain the AJAX Control Toolkit ComboBox control. The ComboBox works like a combination between a standard ASP.NET DropDownList control and a TextBox control. You can either select from a pre-existing list of items or enter a new item.

The ComboBox is similar to the AutoComplete control extender, but the controls are used in different scenarios. The AutoComplete extender queries a web service to get matching entries. The ComboBox control, in contrast, is initialized with a set of items. Using the AutoComplete extender makes sense when you are working with a large set of data (millions of car parts) while using the ComboBox control makes sense when working with a small set of data (dozens of car parts).

Selecting from a Static List of Items

Let's start with a simple sample of using the ComboBox control. Imagine that you want to display a static list of items in a dropdown list. However, you want to leave open the possibility that the list is not complete. You want to allow a user to enter a custom value into the list.

We'll create a new ASP.NET Web Forms page and use the ComboBox control in the page. Add the new ASP.NET page to your project and switch to Design view.

If you want to use the ComboBox control in the page then you must add a ScriptManager control to the page. Drag the ScriptManager control from beneath the AJAX Extensions tab onto the Designer surface. You should add the ScriptManager control at the top of the page; you can add it immediately below the opening server-side <form> tag.

Next, drag the ComboBox control onto the page. You can find the ComboBox control in the Toolbox with the other AJAX Control Toolkit controls and control extenders (see figure1).

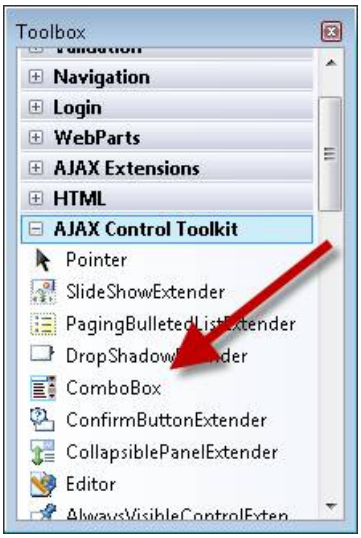


Figure 01: Selecting the ComboBox control from the toolbox (Click to view full-size image)

We'll use the ComboBox control to display a static list of choices. The user can select a particular level of spiciness for their food from a list of three choices: Mild, Medium, and Hot (see Figure 2).

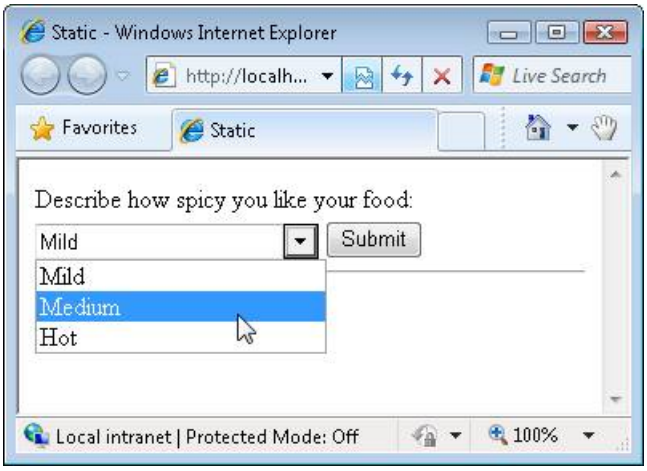


Figure 02: Selecting from a static list of items(Click to view full-size image)

There are two ways that you can add these choices to the ComboBox control. First, you select the Edit Options task option when hovering your mouse over the control in Design view and open the Item Editor (see Figure 3).

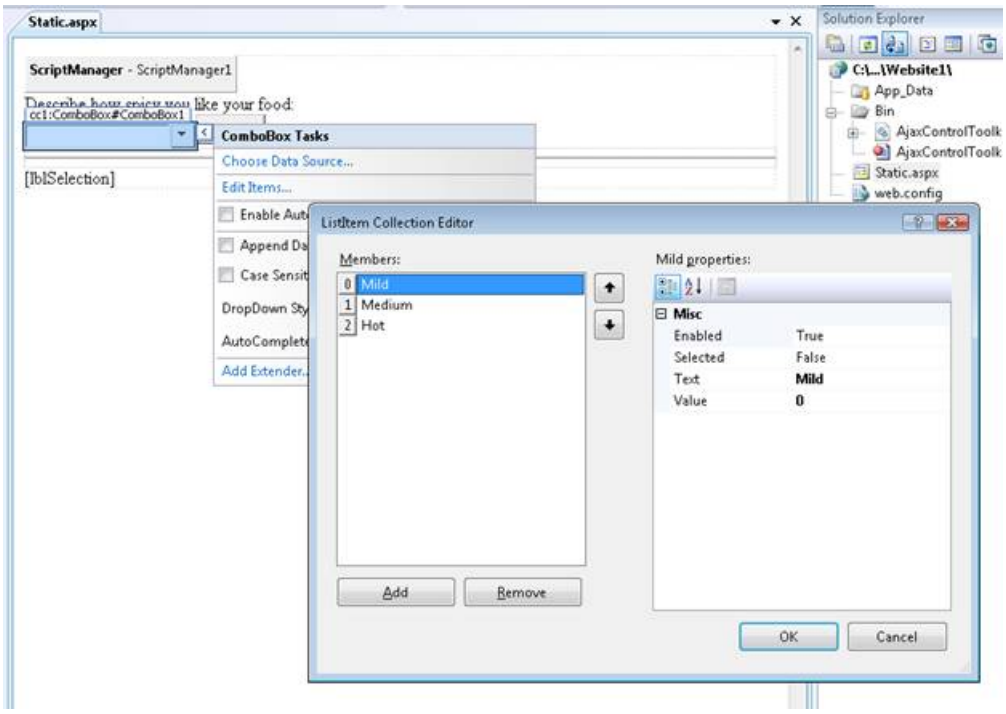


Figure 03: Editing ComboBox items(Click to view full-size image)

The second option is to add the list of items in between the opening and closing <asp:ComboBox> tags in Source view. The page in Listing 1 contains the updated ComboBox that has the list of items.

Listing 1 - Static.aspx

```
<%@ Page Language="C#" %>
<%@ Register Assembly="AjaxControlToolkit" Namespace="AjaxControlToolkit" TagPrefix="cc1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<script runat="server">

    protected void btnSubmit_Click(object sender, EventArgs e)
    {
        lblSelection.Text = "You picked " + ComboBox1.SelectedItem.Text;
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Static</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:ScriptManager ID="ScriptManager1" runat="server" />

        Describe how spicy you like your food:
        <br />
        <cc1:ComboBox ID="ComboBox1" runat="server">
            <asp:ListItem Text="Mild" Value="0" />
            <asp:ListItem Text="Medium" Value="1" />
            <asp:ListItem Text="Hot" Value="2" />
        </cc1:ComboBox>

        <asp:Button
            ID="btnSubmit"
            Text="Submit"
            Runat="server" OnClick="btnSubmit_Click" />

        <hr />
        <asp:Label
            ID="lblSelection"
            Runat="server" />

    </div>
    </form>
</body>
</html>
```

When you open the page in Listing 1, you can select one of the pre-existing options from the ComboBox. In other words, the ComboBox works just like a DropDownList control.

However, you also have the option of entering a new choice (for example, Super Spicy) that is not in the existing list. So, the ComboBox also works like a TextBox control.

Regardless of whether you pick a pre-existing item or you enter a custom item, when you submit the form, your choice appears in the label control. When you submit the form, the btnSubmit_Click handler executes and updates the label (see Figure 4).

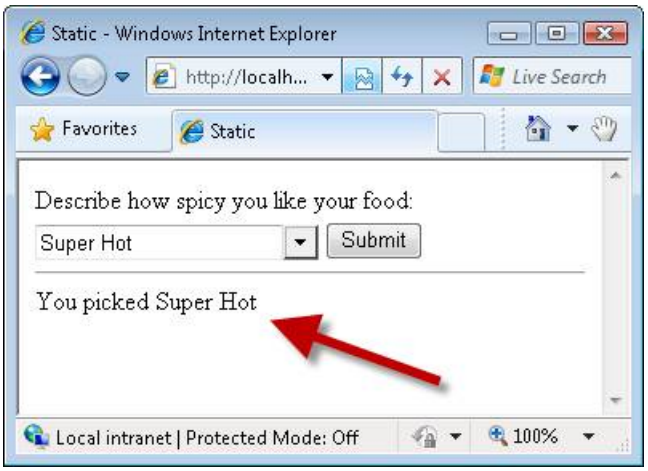


Figure 04: Displaying the selected item(Click to view full-size image)

The ComboBox supports the same properties as the DropDownList control for retrieving the selected item after a form is submitted:

- SelectedItem.Text - Displays the value of the Text property of the selected item.
- SelectedItem.Value - Displays the value of the Value property of the selected item or displays the text typed into the ComboBox.
- SelectedValue - Same as SelectedItem.Value except that this property enables you to specify the default (initial) selected item.

If you type a custom choice into the ComboBox then the custom choice is assigned to both the SelectedItem.Text and SelectedItem.Value properties.

Selecting the List of Items from the Database

You can retrieve the list of items that the ComboBox displays from a database. For example, you can bind the ComboBox to a SqlDataSource control, an ObjectDataSource control, a LinqDataSource, or an EntityDataSource.

Imagine that you want to display a list of movies in a ComboBox. You want to retrieve the list of movies from the Movies database table. Follow these steps:

1. Create a page named Movies.aspx
2. Add a ScriptManager control to the page by dragging the ScriptManager from under the AJAX Extensions tab in the Toolbox onto the page.
3. Add a ComboBox control to the page by dragging the ComboBox onto the page.
4. In Design view, hover your mouse over the ComboBox control and select the **Choose Data Source** task option (see Figure 5). The Data Source Configuration Wizard is launched.
5. In the **Choose a Data Source** step, select the <New data source> option.
6. In the **Choose a Data Source Type** step, select Database.
7. In the **Choose Your Data Connection** step, select your database (for example, MoviesDB.mdf).
8. In the **Save the Connection String to the Application Configuration File** step, select the option to save your connection string.
9. In the **Configure the Select Statement** step, select the Movies database table and select all of the columns.
10. In the **Test Query** step, click the Finish button.
11. Back in the **Choose Data Source** step, select the Title column for the field to display and the Id column for the data field (see Figure).
12. Click the OK button to close the wizard.

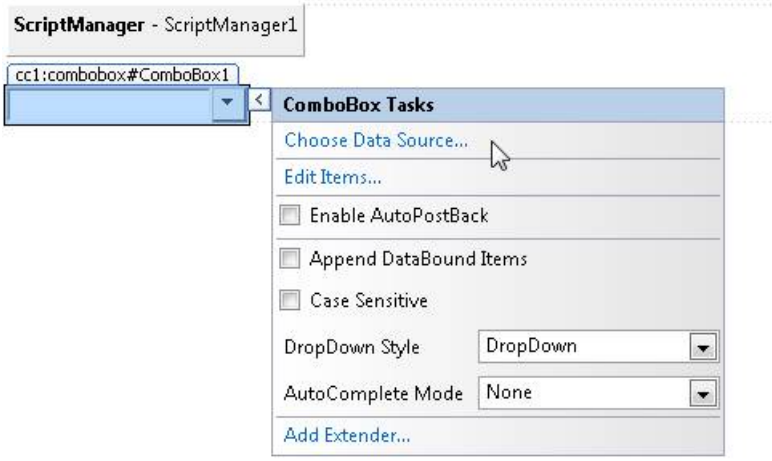


Figure 05: Choosing a data source(Click to view full-size image)

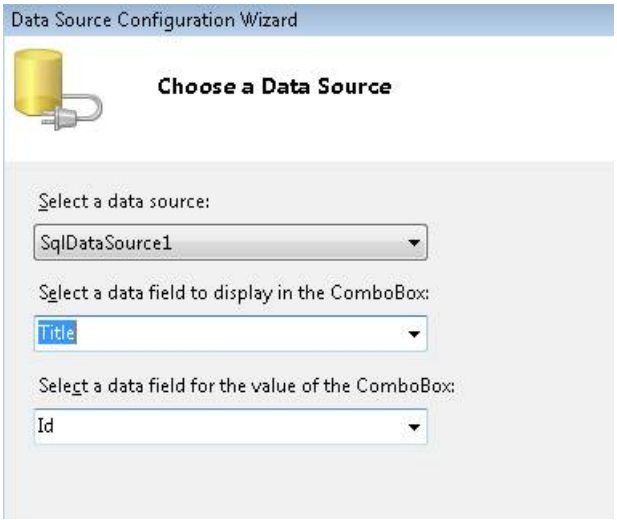


Figure 06: Choosing the data text and value fields(Click to view full-size image)

After you complete the steps above, the ComboBox is bound to a SqlDataSource control that represents the movies from the Movies database table. The source for the page looks like Listing 2 (I cleaned up the formatting a little bit).

Listing 2 - Movies.aspx

```
<%@ Page Language="C#" %>
<%@ Register assembly="AjaxControlToolkit" namespace="AjaxControlToolkit" tagprefix="cc1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        <asp:ScriptManager ID="ScriptManager1" runat="server" />
        <br />

        Select your favorite movie:
        <br />
        <cc1:ComboBox
            ID="ComboBox1"
            DataSourceID="SqlDataSource1"
            DataTextField="Title"
            DataValueField="Id"
            MaxLength="0"
            runat="server" >
        </cc1:ComboBox>

        <asp:SqlDataSource
            ID="SqlDataSource1"
            ConnectionString="<%$ ConnectionStrings:ConnectionString %>"
            SelectCommand="SELECT * FROM [Movies]"
            runat="server">
        </asp:SqlDataSource>

    </div>
    </form>
</body>
</html>
```

Notice that the ComboBox control has a DataSourceID property that points to the SqlDataSource control. When you open the page in a browser, the list of movies from the database is displayed (see Figure 7). You can either a pick a movie from the list or enter a new movie by typing the movie into the ComboBox.

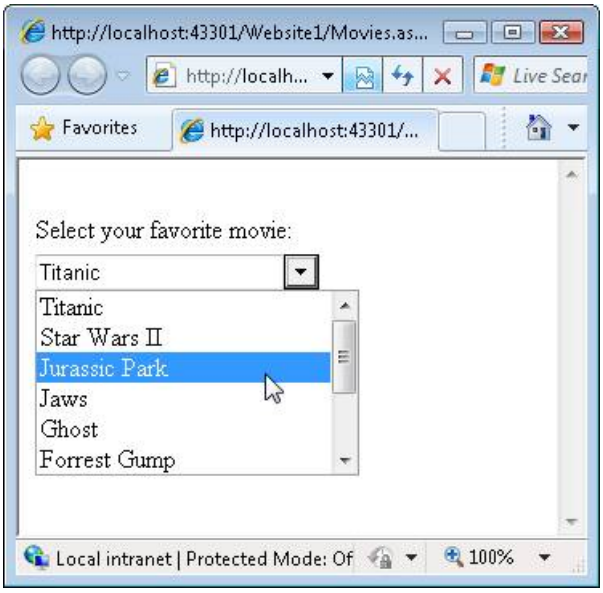


Figure 07: Displaying a list of movies(Click to view full-size image)

Setting the DropDownStyle

You can use the ComboBox DropDownStyle property to change the behavior of the ComboBox. This property accepts there possible values:

- DropDown - (default value) The ComboBox displays a dropdown list when you click the arrow and you can enter a custom value.
- Simple - The ComboBox displays a dropdown list automatically and you can enter a custom value.
- DropDownList - The ComboBox works just like a DropDownList control.

The different between DropDown and Simple is when the list of items is displayed. In the case of Simple, the list is displayed immediately when you move focus to the ComboBox. In the case of DropDown, you must click the arrow to see the list of items.

The DropDownList value causes the ComboBox control to work just like a standard DropDownList control. However, there is an important difference here. Older versions of Internet Explorer display a DropDownList control with an infinite z-index so the control will appear in front of any control placed in front of it. Because the ComboBox renders an HTML tag instead of an HTML <select> tag, the ComboBox correctly respects z-ordering.

Setting the AutoCompleteMode

You use the ComboBox AutoCompleteMode property to specify what happens when someone types text into the ComboBox. This property accepts the following possible values:

- None - (default value) The ComboBox does not provide any auto-complete behavior.
- Suggest - The ComboBox displays the list and it highlights the matching item in the list (see Figure 8).
- Append - The ComboBox does not display the list and it appends the matching item from the list onto what you have typed (see Figure 9).
- SuggestAppend - The ComboBox both displays the list and appends the matching item from the list onto what you have typed (see Figure 10).

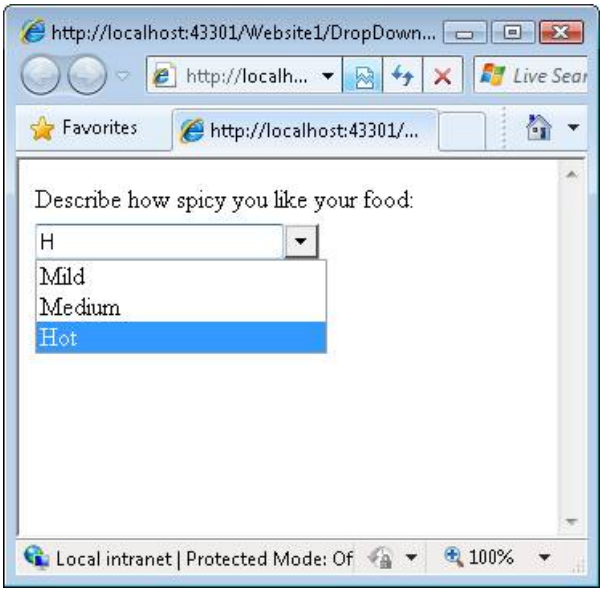


Figure 08: The ComboBox makes a suggestion(Click to view full-size image)

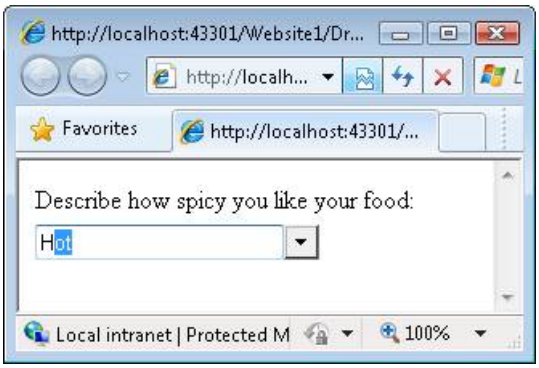


Figure 09: ComboBox appends matching text(Click to view full-size image)

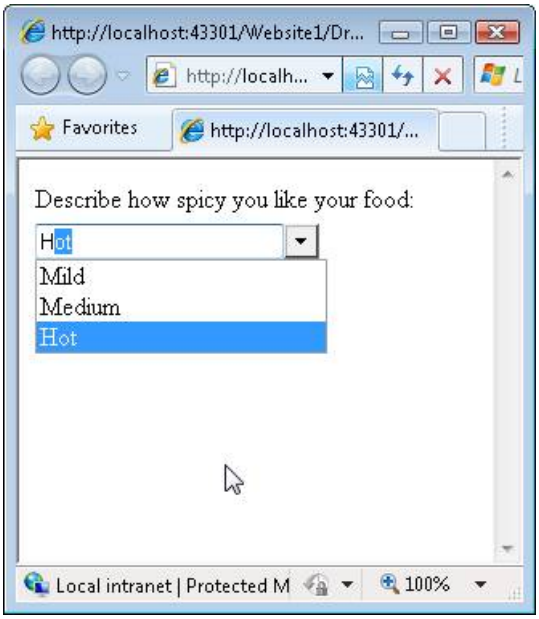


Figure 10: The ComboBox suggests and appends(Click to view full-size image)

Summary

In this tutorial, you learned how to use the ComboBox control to display a fixed set of items. We bound the ComboBox control both to a static set of items and to a database table. Finally, you learned how to modify the behavior of the ComboBox by setting its DropDownStyle and AutoCompleteMode properties.

NEXT

How do I use the ComboBox Control? (VB)

9/16/2020 • 8 minutes to read • Edit Online

by Microsoft

> ComboBox is an ASP.NET AJAX control that combines the flexibility of a TextBox with a list of options from which users can choose.

The goal of this tutorial is to explain the AJAX Control Toolkit ComboBox control. The ComboBox works like a combination between a standard ASP.NET DropDownList control and a TextBox control. You can either select from a pre-existing list of items or enter a new item.

The ComboBox is similar to the AutoComplete control extender, but the controls are used in different scenarios. The AutoComplete extender queries a web service to get matching entries. The ComboBox control, in contrast, is initialized with a set of items. Using the AutoComplete extender makes sense when you are working with a large set of data (millions of car parts) while using the ComboBox control makes sense when working with a small set of data (dozens of car parts).

Selecting from a Static List of Items

Let's start with a simple sample of using the ComboBox control. Imagine that you want to display a static list of items in a dropdown list. However, you want to leave open the possibility that the list is not complete. You want to allow a user to enter a custom value into the list.

We'll create a new ASP.NET Web Forms page and use the ComboBox control in the page. Add the new ASP.NET page to your project and switch to Design view.

If you want to use the ComboBox control in the page then you must add a ScriptManager control to the page. Drag the ScriptManager control from beneath the AJAX Extensions tab onto the Designer surface. You should add the ScriptManager control at the top of the page; you can add it immediately below the opening server-side <form> tag.

Next, drag the ComboBox control onto the page. You can find the ComboBox control in the Toolbox with the other AJAX Control Toolkit controls and control extenders (see figure1).

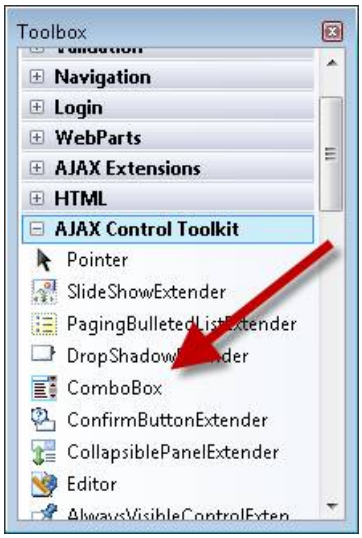


Figure 01: Selecting the ComboBox control from the toolbox (Click to view full-size image)

We'll use the ComboBox control to display a static list of choices. The user can select a particular level of spiciness for their food from a list of three choices: Mild, Medium, and Hot (see Figure 2).

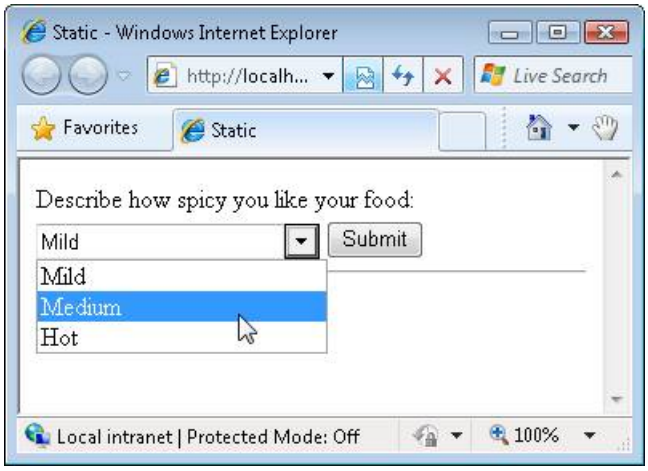


Figure 02: Selecting from a static list of items(Click to view full-size image)

There are two ways that you can add these choices to the ComboBox control. First, you select the Edit Options task option when hovering your mouse over the control in Design view and open the Item Editor (see Figure 3).

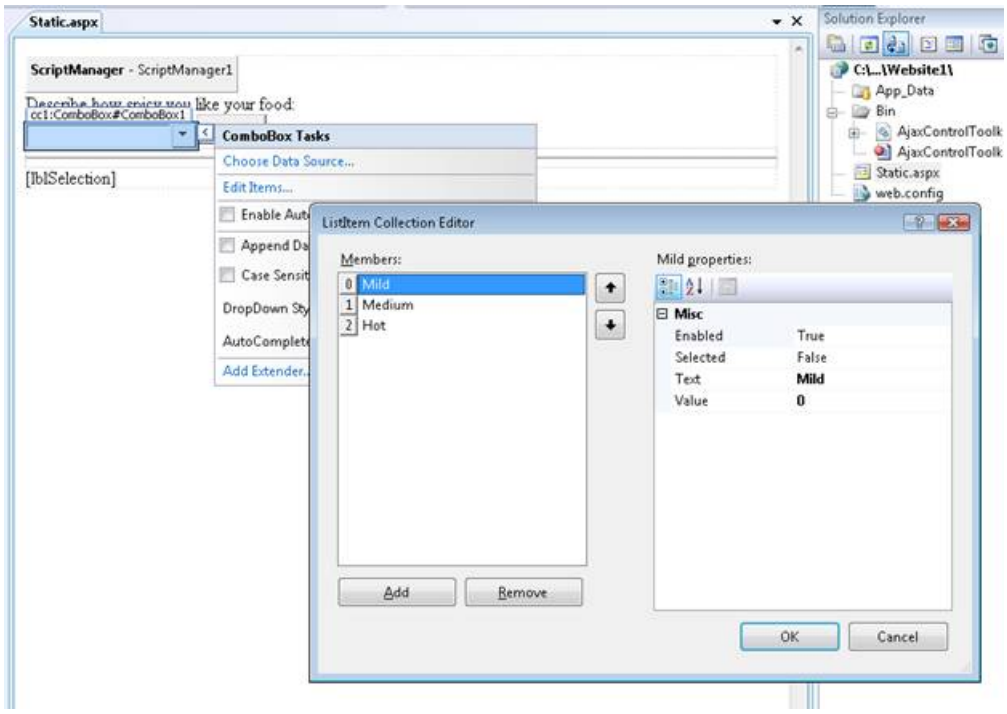


Figure 03: Editing ComboBox items(Click to view full-size image)

The second option is to add the list of items in between the opening and closing <asp:ComboBox> tags in Source view. The page in Listing 1 contains the updated ComboBox that has the list of items.

Listing 1 - Static.aspx

```
<%@ Page Language="VB" %>
<%@ Register Assembly="AjaxControlToolkit" Namespace="AjaxControlToolkit" TagPrefix="cc1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<script runat="server">

    Protected Sub btnSubmit_Click(ByVal sender As Object, ByVal e As System.EventArgs)
        lblSelection.Text = "You picked " & ComboBox1.SelectedItem.Text
    End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Static</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:ScriptManager ID="ScriptManager1" runat="server" />

        Describe how spicy you like your food:
        <br />
        <cc1:ComboBox ID="ComboBox1" runat="server">
            <asp:ListItem Text="Mild" Value="0" />
            <asp:ListItem Text="Medium" Value="1" />
            <asp:ListItem Text="Hot" Value="2" />
        </cc1:ComboBox>

        <asp:Button
            ID="btnSubmit"
            Text="Submit"
            Runat="server" OnClick="btnSubmit_Click" />

        <hr />
        <asp:Label
            ID="lblSelection"
            Runat="server" />

    </div>
    </form>
</body>
</html>
```

When you open the page in Listing 1, you can select one of the pre-existing options from the ComboBox. In other words, the ComboBox works just like a DropDownList control.

However, you also have the option of entering a new choice (for example, Super Spicy) that is not in the existing list. So, the ComboBox also works like a TextBox control.

Regardless of whether you pick a pre-existing item or you enter a custom item, when you submit the form, your choice appears in the label control. When you submit the form, the btnSubmit_Click handler executes and updates the label (see Figure 4).

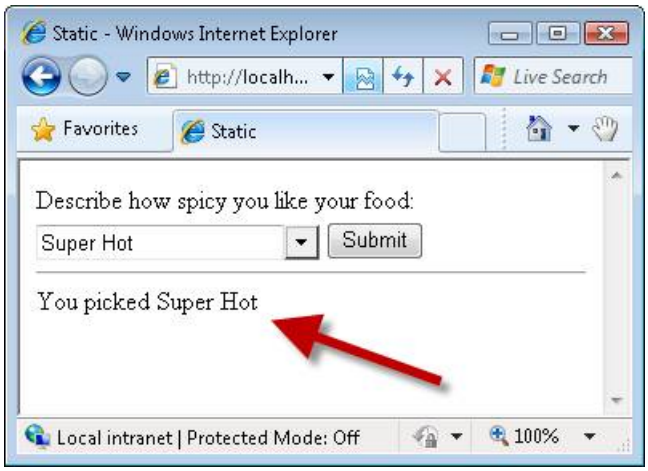


Figure 04: Displaying the selected item(Click to view full-size image)

The ComboBox supports the same properties as the DropDownList control for retrieving the selected item after a form is submitted:

- SelectedItem.Text - Displays the value of the Text property of the selected item.
- SelectedItem.Value - Displays the value of the Value property of the selected item or displays the text typed into the ComboBox.
- SelectedValue - Same as SelectedItem.Value except that this property enables you to specify the default (initial) selected item.

If you type a custom choice into the ComboBox then the custom choice is assigned to both the SelectedItem.Text and SelectedItem.Value properties.

Selecting the List of Items from the Database

You can retrieve the list of items that the ComboBox displays from a database. For example, you can bind the ComboBox to a SqlDataSource control, an ObjectDataSource control, a LinqDataSource, or an EntityDataSource.

Imagine that you want to display a list of movies in a ComboBox. You want to retrieve the list of movies from the Movies database table. Follow these steps:

1. Create a page named Movies.aspx
2. Add a ScriptManager control to the page by dragging the ScriptManager from under the AJAX Extensions tab in the Toolbox onto the page.
3. Add a ComboBox control to the page by dragging the ComboBox onto the page.
4. In Design view, hover your mouse over the ComboBox control and select the **Choose Data Source** task option (see Figure 5). The Data Source Configuration Wizard is launched.
5. In the **Choose a Data Source** step, select the <New data source> option.
6. In the **Choose a Data Source Type** step, select Database.
7. In the **Choose Your Data Connection** step, select your database (for example, MoviesDB.mdf).
8. In the **Save the Connection String to the Application Configuration File** step, select the option to save your connection string.
9. In the **Configure the Select Statement** step, select the Movies database table and select all of the columns.
10. In the **Test Query** step, click the Finish button.
11. Back in the **Choose Data Source** step, select the Title column for the field to display and the Id column for the data field (see Figure).
12. Click the OK button to close the wizard.

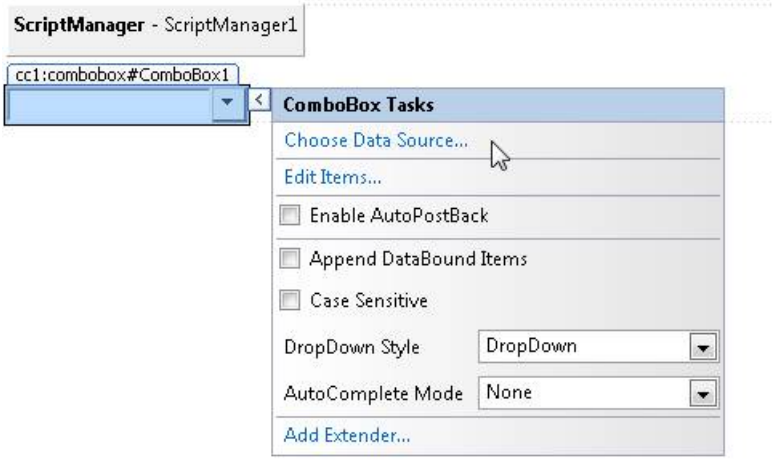


Figure 05: Choosing a data source(Click to view full-size image)

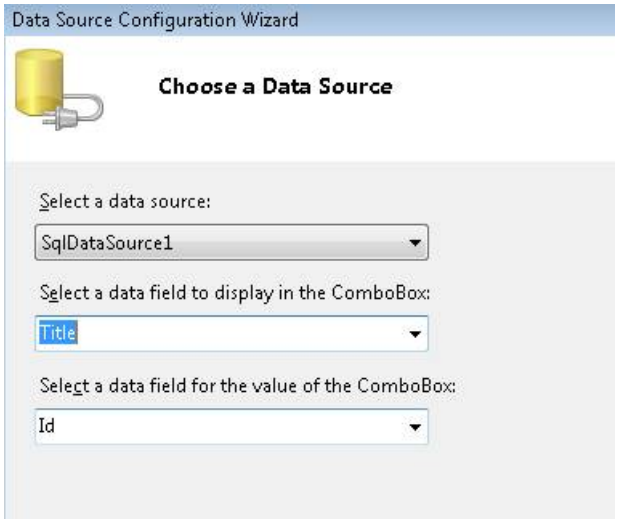


Figure 06: Choosing the data text and value fields(Click to view full-size image)

After you complete the steps above, the ComboBox is bound to a SqlDataSource control that represents the movies from the Movies database table. The source for the page looks like Listing 2 (I cleaned up the formatting a little bit).

Listing 2 - Movies.aspx

```
<%@ Page Language="VB" %>
<%@ Register assembly="AjaxControlToolkit" namespace="AjaxControlToolkit" tagprefix="cc1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        <asp:ScriptManager ID="ScriptManager1" runat="server" />
        <br />

        Select your favorite movie:
        <br />
        <cc1:ComboBox
            ID="ComboBox1"
            DataSourceID="SqlDataSource1"
            DataTextField="Title"
            DataValueField="Id"
            MaxLength="0"
            runat="server" >
        </cc1:ComboBox>

        <asp:SqlDataSource
            ID="SqlDataSource1"
            ConnectionString="<%$ ConnectionStrings:ConnectionString %>"
            SelectCommand="SELECT * FROM [Movies]"
            runat="server">
        </asp:SqlDataSource>

    </div>
    </form>
</body>
</html>
```

Notice that the ComboBox control has a DataSourceID property that points to the SqlDataSource control. When you open the page in a browser, the list of movies from the database is displayed (see Figure 7). You can either a pick a movie from the list or enter a new movie by typing the movie into the ComboBox.

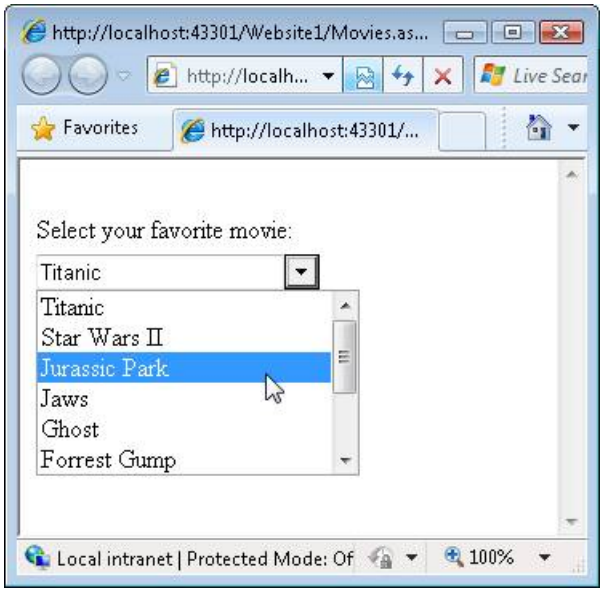


Figure 07: Displaying a list of movies(Click to view full-size image)

Setting the DropDownStyle

You can use the ComboBox DropDownStyle property to change the behavior of the ComboBox. This property accepts there possible values:

- DropDown - (default value) The ComboBox displays a dropdown list when you click the arrow and you can enter a custom value.
- Simple - The ComboBox displays a dropdown list automatically and you can enter a custom value.
- DropDownList - The ComboBox works just like a DropDownList control.

The different between DropDown and Simple is when the list of items is displayed. In the case of Simple, the list is displayed immediately when you move focus to the ComboBox. In the case of DropDown, you must click the arrow to see the list of items.

The DropDownList value causes the ComboBox control to work just like a standard DropDownList control. However, there is an important difference here. Older versions of Internet Explorer display a DropDownList control with an infinite z-index so the control will appear in front of any control placed in front of it. Because the ComboBox renders an HTML tag instead of an HTML <select> tag, the ComboBox correctly respects z-ordering.

Setting the AutoCompleteMode

You use the ComboBox AutoCompleteMode property to specify what happens when someone types text into the ComboBox. This property accepts the following possible values:

- None - (default value) The ComboBox does not provide any auto-complete behavior.
- Suggest - The ComboBox displays the list and it highlights the matching item in the list (see Figure 8).
- Append - The ComboBox does not display the list and it appends the matching item from the list onto what you have typed (see Figure 9).
- SuggestAppend - The ComboBox both displays the list and appends the matching item from the list onto what you have typed (see Figure 10).

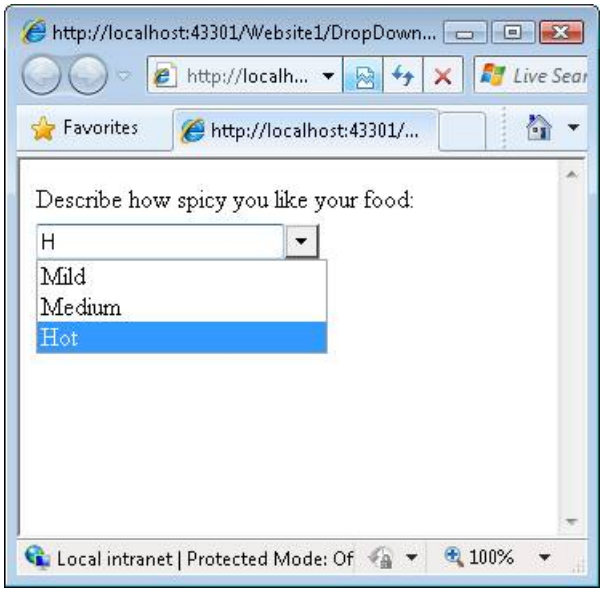


Figure 08: The ComboBox makes a suggestion(Click to view full-size image)

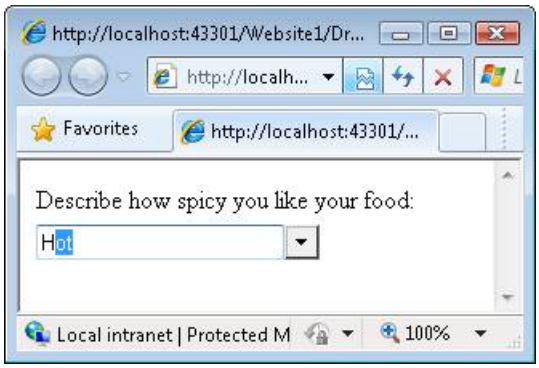


Figure 09: ComboBox appends matching text(Click to view full-size image)

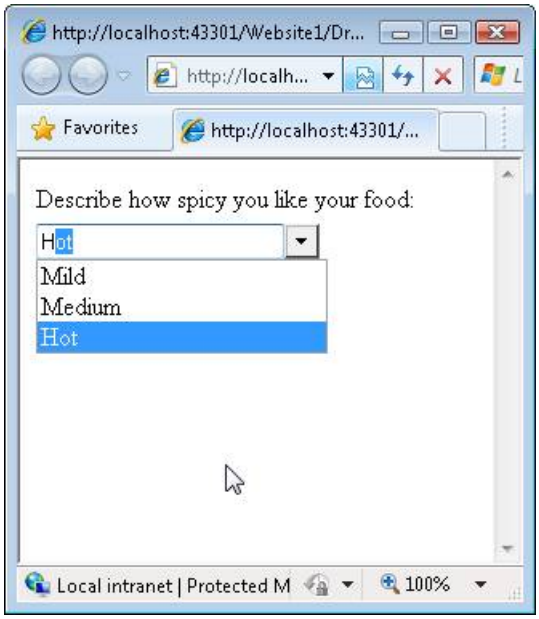


Figure 10: The ComboBox suggests and appends(Click to view full-size image)

Summary

In this tutorial, you learned how to use the ComboBox control to display a fixed set of items. We bound the ComboBox control both to a static set of items and to a database table. Finally, you learned how to modify the behavior of the ComboBox by setting its DropDownStyle and AutoCompleteMode properties.

PREVIOUS

ConfirmButton

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the ConfirmButton control, which displays a Yes/No popup when the user clicks a button.

- Using a ConfirmButton In a Repeater (C#)
- Using a ConfirmButton In a Repeater (VB)

Using a ConfirmButton In a Repeater (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ConfirmButton extender in the AJAX Control Toolkit creates a Yes/No popup when the user clicks on a button (including LinkButton control). Only if Yes is clicked, the button's action is executed, otherwise cancelled. This is also possible in a repeater.

Overview

The ConfirmButton extender in the AJAX Control Toolkit creates a Yes/No popup when the user clicks on a button (including LinkButton control). Only if Yes is clicked, the button's action is executed, otherwise cancelled. This is also possible in a repeater.

Steps

First of all, a data source is required. This sample uses the AdventureWorks database and the Microsoft SQL Server 2005 Express Edition. The database is an optional part of a Visual Studio installation (including express edition) and is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. The AdventureWorks database is part of the SQL Server 2005 Samples and Sample Databases (download at https://www.microsoft.com/downloads/details.aspx?FamilyID=e719ecf7-9f46-4312-af89-6ad8702e4e6e&DisplayLang=en). The easiest way to set the database up is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en) and attach the `AdventureWorks.mdf` database file.

For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database.

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

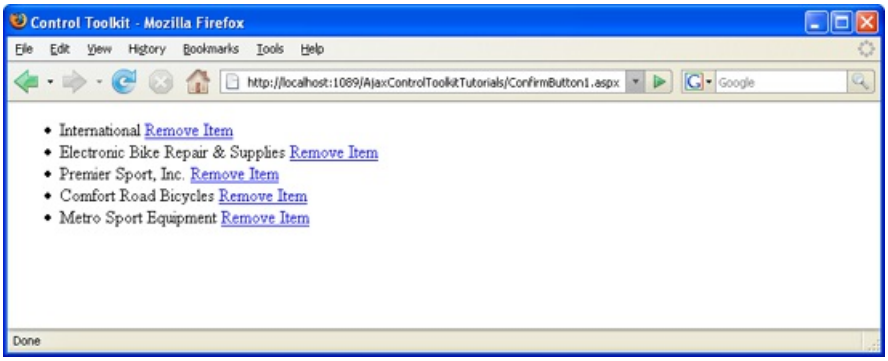
```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, a data source is required. For the sake of simplicity, only the first five entries in AdventureWorks' Vendors table are retrieved. Note that when using the Visual Studio wizard to create the data source, the table name (`Vendors`) is currently not correctly prefixed with `Purchasing`. The following markup is the correct one:

```
<asp:SqlDataSource ID="sds1" runat="server" ConnectionString="
 Data Source=(local)\SQLEXPRESS;Initial Catalog=AdventureWorks;Integrated Security=True"
 ProviderName="System.Data.SqlClient" SelectCommand="SELECT TOP 5
 [VendorID], [Name] FROM [Purchasing].[Vendor]" />
```

This data source can then be used within a repeater. As usual, the `DataBinder.Eval()` method retrieves data from the data source. The `ConfirmButtonExtender` control must then be placed within the `<ItemTemplate>` section of the repeater so that it appears for every entry in the data source.

```
<div>
 <ul>
 <asp:Repeater ID="rep1" DataSourceID="sds1" runat="server">
 <ItemTemplate>
 <li>
 <%#DataBinder.Eval(Container.DataItem, "Name")%>
 <asp:LinkButton ID="btn1" Text="Remove Item" runat="server" />
 <ajaxToolkit:ConfirmButtonExtender ID="cfe1" runat="server" TargetControlID="btn1" ConfirmText="Are you
sure?!" />
 </li>
 </ItemTemplate>
 </asp:Repeater>
 </ul>
</div>
```



The confirm button appears next to each entry from the data source (Click to view full-size image)

NEXT

Using a ConfirmButton In a Repeater (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ConfirmButton extender in the AJAX Control Toolkit creates a Yes/No popup when the user clicks on a button (including LinkButton control). Only if Yes is clicked, the button's action is executed, otherwise cancelled. This is also possible in a repeater.

Overview

The ConfirmButton extender in the AJAX Control Toolkit creates a Yes/No popup when the user clicks on a button (including LinkButton control). Only if Yes is clicked, the button's action is executed, otherwise cancelled. This is also possible in a repeater.

Steps

First of all, a data source is required. This sample uses the AdventureWorks database and the Microsoft SQL Server 2005 Express Edition. The database is an optional part of a Visual Studio installation (including express edition) and is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. The AdventureWorks database is part of the SQL Server 2005 Samples and Sample Databases (download at https://www.microsoft.com/downloads/details.aspx?FamilyID=e719ecf7-9f46-4312-af89-6ad8702e4e6e&DisplayLang=en). The easiest way to set the database up is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en) and attach the `AdventureWorks.mdf` database file.

For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database.

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

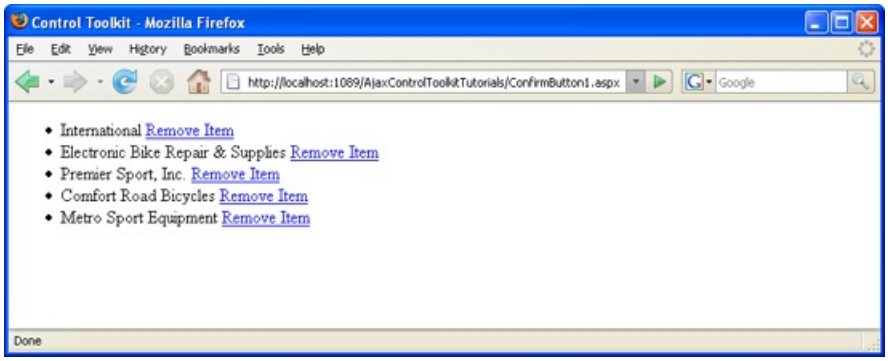
```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, a data source is required. For the sake of simplicity, only the first five entries in AdventureWorks' Vendors table are retrieved. Note that when using the Visual Studio wizard to create the data source, the table name (`Vendors`) is currently not correctly prefixed with `Purchasing`. The following markup is the correct one:

```
<asp:SqlDataSource ID="sds1" runat="server" ConnectionString="
 Data Source=(local)\SQLEXPRESS;Initial Catalog=AdventureWorks;Integrated Security=True"
 ProviderName="System.Data.SqlClient" SelectCommand="SELECT TOP 5
 [VendorID], [Name] FROM [Purchasing].[Vendor]" />
```

This data source can then be used within a repeater. As usual, the `DataBinder.Eval()` method retrieves data from the data source. The `ConfirmButtonExtender` control must then be placed within the `<ItemTemplate>` section of the repeater so that it appears for every entry in the data source.

```
<div>
 <ul>
 <asp:Repeater ID="rep1" DataSourceID="sds1" runat="server">
 <ItemTemplate>
 <li>
 <%#DataBinder.Eval(Container.DataItem, "Name")%>
 <asp:LinkButton ID="btn1" Text="Remove Item" runat="server" />
 <ajaxToolkit:ConfirmButtonExtender ID="cfe1" runat="server" TargetControlID="btn1" ConfirmText="Are you
sure?!" />
 </li>
 </ItemTemplate>
 </asp:Repeater>
 </ul>
</div>
```



The confirm button appears next to each entry from the data source (Click to view full-size image)

PREVIOUS

DropShadow

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the DropShadow control, which draws a drop shadow behind a panel.

- Adjusting the Z-Index of a DropShadow (C#)
- Manipulating DropShadow Properties from Client Code (C#)
- Adjusting the Z-Index of a DropShadow (VB)
- Manipulating DropShadow Properties from Client Code (VB)

Adjusting the Z-Index of a DropShadow (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The DropShadow control in the AJAX Control Toolkit extends a panel with a drop shadow. However this shadow sometimes conflicts with other controls, for instance the ASP.NET Menu control. When a menu entry pops up, it appears behind the drop shadow.

Overview

The DropShadow control in the AJAX Control Toolkit extends a panel with a drop shadow. However this shadow sometimes conflicts with other controls, for instance the ASP.NET Menu control. When a menu entry pops up, it appears behind the drop shadow.

Steps

The code commences with the Panel itself, containing enough text so that the panel contains enough text for the effect to be visible:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panel" Width="300px">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

Another panel is placed directly before the `panelShadow` panel. It contains a menu with horizontal orientation so that menu entries appear over (or rather: under) the `dropShadow` panel):

```
<asp:Panel ID="panelMenu" runat="server" CssClass="ForegroundStyle">
 <asp:Menu ID="menu1" runat="server" ForeColor="Red" Orientation="Horizontal">
 <Items>
 <asp:MenuItem Text="One">
 <asp:MenuItem Text="1.1" />
 <asp:MenuItem Text="1.2" />
 </asp:MenuItem>
 <asp:MenuItem Text="Two" />
 <asp:MenuItem Text="Three" />
 </Items>
 </asp:Menu><br />
</asp:Panel>
```

Then, the `DropShadowExtender` is added to extend the `panelShadow` panel with a drop shadow effect:

```
<ajaxToolkit:DropShadowExtender ID="dse1" runat="server"
 TargetControlID="panelShadow"
 Opacity="0.5" Rounded="true" />
```

Finally, the ASP.NET AJAX `ScriptManager` control enables the Control Toolkit to work:

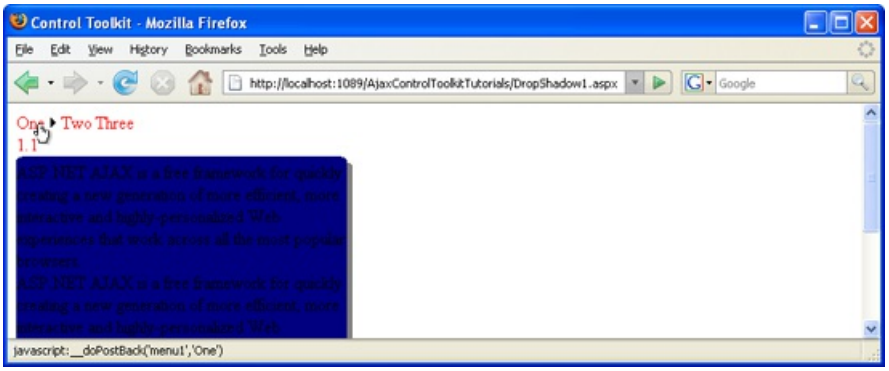
```
<asp:ScriptManager ID="asm" runat="server" />
```

When you run this script, the menu entries appear underneath the panel. However the menu uses the CSS class `panel` where you just have to define two things to make elements appear in front of the other panel:

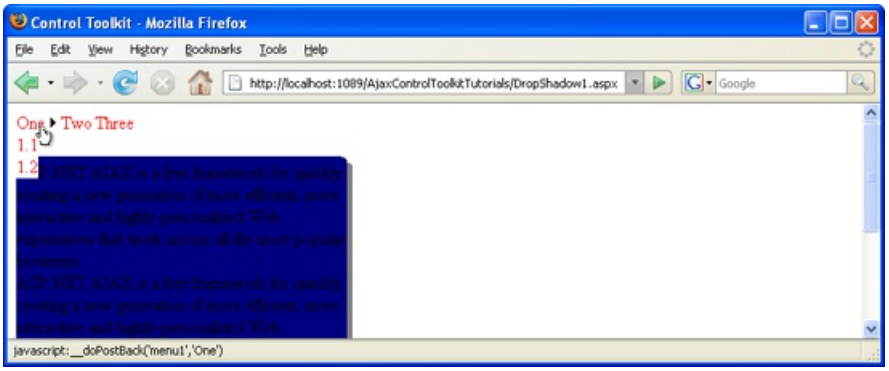
- Relative positioning
- A positive z-index

```
<style type="text/css">
 .ForegroundStyle {z-index: 123; position: relative;}
 .panel {background-color: navy;}
</style>
```

Then, the `DropShadowExtender` control does not conflict any longer with the Menu control.



Before: The menu entry is not visible (Click to view full-size image)



After: The menu entry appears (Click to view full-size image)

NEXT

Manipulating DropShadow Properties from Client Code (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The DropShadow control in the AJAX Control Toolkit extends a panel with a drop shadow. Properties of this extender can also be changed using client JavaScript code.

Overview

The DropShadow control in the AJAX Control Toolkit extends a panel with a drop shadow. Properties of this extender can also be changed using client JavaScript code.

Steps

The code starts with a panel containing some lines of text:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panel" Width="300px">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

The associated CSS class gives the panel a nice background color:

```
<style type="text/css">
 .panel {background-color: navy;}
</style>
```

The `DropShadowExtender` is added to extend the panel with a drop shadow effect, opacity set to 50%:

```
<ajaxToolkit:DropShadowExtender ID="dse1" runat="server"
 TargetControlID="panelShadow"
 Opacity="0.5" Rounded="true" />
```

Then, the ASP.NET AJAX `ScriptManager` control enables the Control Toolkit to work:

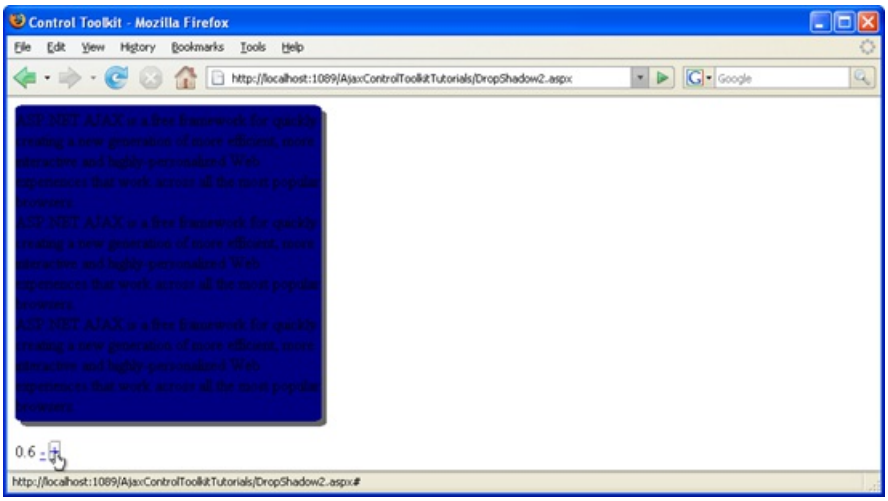
```
<asp:ScriptManager ID="asm" runat="server" />
```

Another panel contains two JavaScript links for setting the opacity of the drop shadow: the minus link decreases the shadow's opacity, the plus link increases it.

```
<asp:Panel ID="panelControl" runat="server">
 <br />
 <label id="txtOpacity" runat="server">0.5</label>
 <a href="#" onclick="changeOpacity(-0.1); return false;">-</a>
 <a href="#" onclick="changeOpacity(+0.1); return false;">+</a>
</asp:Panel>
```

The JavaScript function `changeOpacity()` must then first find the `DropShadowExtender` control on the page. ASP.NET AJAX defines the `$find()` method for exactly that task. Then, the `get_Opacity()` method retrieves the current opacity, the `set_Opacity()` method sets it. The JavaScript code then puts the current opacity value in the `<label>` element:

```
<script type="text/javascript">
 function changeOpacity(delta)
 {
 var dse = $find("dse1");
 var o = dse.get_Opacity();
 o += delta;
 o = Math.round(10 * o) / 10;
 if (o <= 1.0 && o >= 0.0)
 {
 dse.set_Opacity(o);
 $get("txtOpacity").firstChild.nodeValue = o;
 }
 }
</script>
```



The opacity is changed on the client side (Click to view full-size image)

PREVIOUS NEXT

Adjusting the Z-Index of a DropShadow (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The DropShadow control in the AJAX Control Toolkit extends a panel with a drop shadow. However this shadow sometimes conflicts with other controls, for instance the ASP.NET Menu control. When a menu entry pops up, it appears behind the drop shadow.

Overview

The DropShadow control in the AJAX Control Toolkit extends a panel with a drop shadow. However this shadow sometimes conflicts with other controls, for instance the ASP.NET Menu control. When a menu entry pops up, it appears behind the drop shadow.

Steps

The code commences with the Panel itself, containing enough text so that the panel contains enough text for the effect to be visible:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panel" Width="300px">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

Another panel is placed directly before the `panelShadow` panel. It contains a menu with horizontal orientation so that menu entries appear over (or rather: under) the `dropShadow` panel):

```
<asp:Panel ID="panelMenu" runat="server" CssClass="ForegroundStyle">
 <asp:Menu ID="menu1" runat="server" ForeColor="Red" Orientation="Horizontal">
 <Items>
 <asp:MenuItem Text="One">
 <asp:MenuItem Text="1.1" />
 <asp:MenuItem Text="1.2" />
 </asp:MenuItem>
 <asp:MenuItem Text="Two" />
 <asp:MenuItem Text="Three" />
 </Items>
 </asp:Menu><br />
</asp:Panel>
```

Then, the `DropShadowExtender` is added to extend the `panelShadow` panel with a drop shadow effect:

```
<ajaxToolkit:DropShadowExtender ID="dse1" runat="server"
 TargetControlID="panelShadow"
 Opacity="0.5" Rounded="true" />
```

Finally, the ASP.NET AJAX `ScriptManager` control enables the Control Toolkit to work:

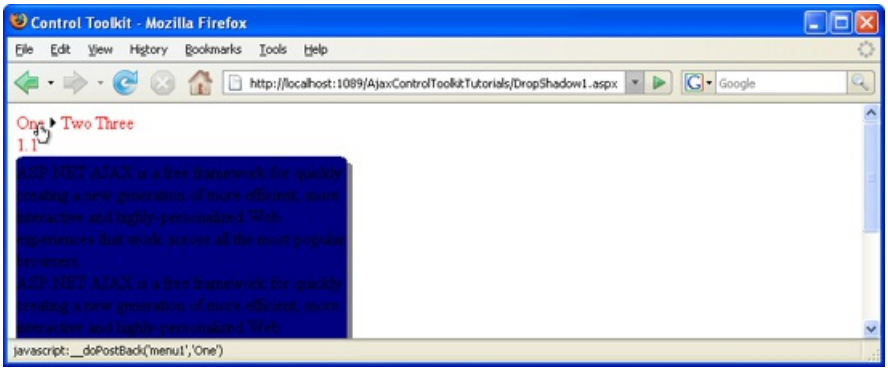
```
<asp:ScriptManager ID="asm" runat="server" />
```

When you run this script, the menu entries appear underneath the panel. However the menu uses the CSS class `panel` where you just have to define two things to make elements appear in front of the other panel:

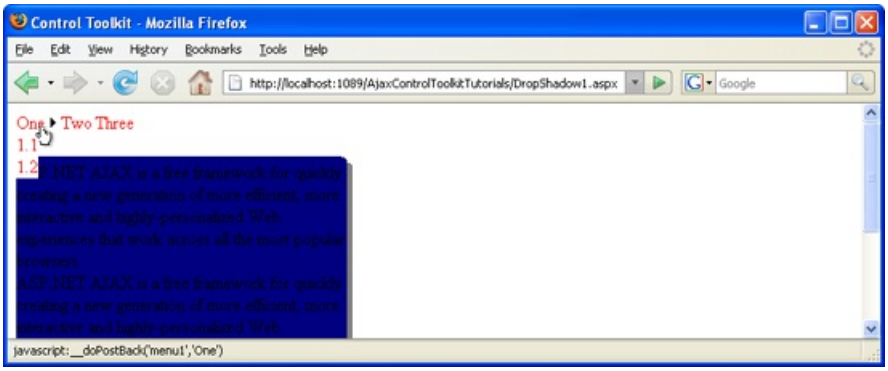
- Relative positioning
- A positive z-index

```
<style type="text/css">
 .ForegroundStyle {z-index: 123; position: relative;}
 .panel {background-color: navy;}
</style>
```

Then, the `DropShadowExtender` control does not conflict any longer with the Menu control.



Before: The menu entry is not visible (Click to view full-size image)



After: The menu entry appears (Click to view full-size image)

PREVIOUS NEXT

Manipulating DropShadow Properties from Client Code (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The DropShadow control in the AJAX Control Toolkit extends a panel with a drop shadow. Properties of this extender can also be changed using client JavaScript code.

Overview

The DropShadow control in the AJAX Control Toolkit extends a panel with a drop shadow. Properties of this extender can also be changed using client JavaScript code.

Steps

The code starts with a panel containing some lines of text:

```
<asp:Panel ID="panelShadow" runat="server" CssClass="panel" Width="300px">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
</asp:Panel>
```

The associated CSS class gives the panel a nice background color:

```
<style type="text/css">
 .panel {background-color: navy;}
</style>
```

The `DropShadowExtender` is added to extend the panel with a drop shadow effect, opacity set to 50%:

```
<ajaxToolkit:DropShadowExtender ID="dse1" runat="server"
 TargetControlID="panelShadow"
 Opacity="0.5" Rounded="true" />
```

Then, the ASP.NET AJAX `ScriptManager` control enables the Control Toolkit to work:

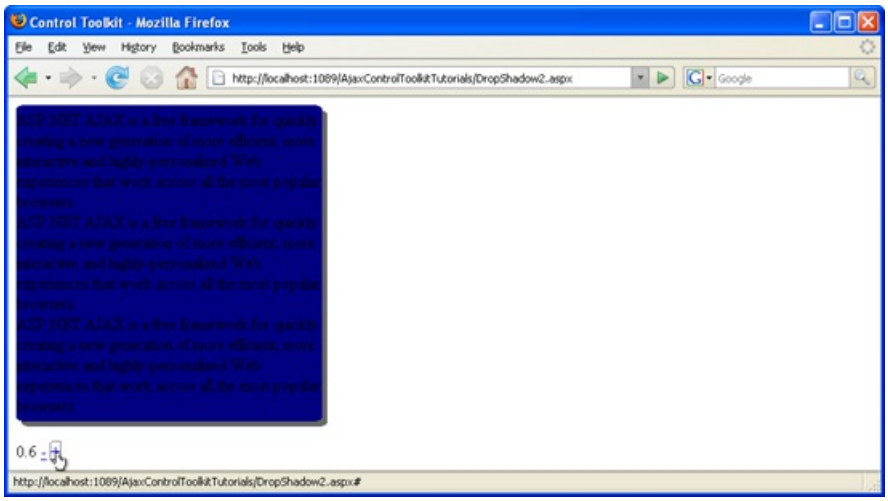
```
<asp:ScriptManager ID="asm" runat="server" />
```

Another panel contains two JavaScript links for setting the opacity of the drop shadow: the minus link decreases the shadow's opacity, the plus link increases it.

```
<asp:Panel ID="panelControl" runat="server">
 <br />
 <label id="txtOpacity" runat="server">0.5</label>
 <a href="#" onclick="changeOpacity(-0.1); return false;">-</a>
 <a href="#" onclick="changeOpacity(+0.1); return false;">+</a>
</asp:Panel>
```

The JavaScript function `changeOpacity()` must then first find the `DropShadowExtender` control on the page. ASP.NET AJAX defines the `$find()` method for exactly that task. Then, the `get_Opacity()` method retrieves the current opacity, the `set_Opacity()` method sets it. The JavaScript code then puts the current opacity value in the `<label>` element:

```
<script type="text/javascript">
 function changeOpacity(delta)
 {
 var dse = $find("dse1");
 var o = dse.get_Opacity();
 o += delta;
 o = Math.round(10 * o) / 10;
 if (o <= 1.0 && o >= 0.0)
 {
 dse.set_Opacity(o);
 $get("txtOpacity").firstChild.nodeValue = o;
 }
 }
</script>
```



The opacity is changed on the client side (Click to view full-size image)

PREVIOUS

DynamicPopulate

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the DynamicPopulate control, which calls a web service and displays the resulting value in a control on the page.

- Dynamically Populating a Control (C#)
- Dynamically Populating a Control Using JavaScript Code (C#)
- Using DynamicPopulate with a User Control And JavaScript (C#)
- Dynamically Populating a Control (VB)
- Dynamically Populating a Control Using JavaScript Code (VB)
- Using DynamicPopulate with a User Control And JavaScript (VB)

Dynamically Populating a Control (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The DynamicPopulate control in the ASP.NET AJAX Control Toolkit calls a web service (or page method) and fills the resulting value into a target control on the page, without a page refresh.

Overview

The `DynamicPopulate` control in the ASP.NET AJAX Control Toolkit calls a web service (or page method) and fills the resulting value into a target control on the page, without a page refresh. This tutorial shows how to set this up.

Steps

First of all, you need an ASP.NET Web Service which implements the method to be called by `DynamicPopulate`. The web service class requires the `ScriptService` attribute which is defined within `Microsoft.Web.Script.Services`; otherwise ASP.NET AJAX cannot create the client-side JavaScript proxy for the web service which in turn is required by `DynamicPopulate`.

The web method must expect one argument of type string, called `contextKey`, since the `DynamicPopulate` control sends one piece of context information with each web service call. The following web service returns the current date in a format represented by the `contextKey` argument:

```
<%@ WebService Language="C#" Class="DynamicPopulate" %>
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Web.Script.Services;
[ScriptService]
public class DynamicPopulate : System.Web.Services.WebService
{
 [WebMethod]
 public string getDate(string contextKey)
 {
 string myDate = "";
 switch (contextKey)
 {
 case "format1":
 myDate = String.Format("{0:MM}-{0:dd}-{0:yyyy}", DateTime.Now);
 break;
 case "format2":
 myDate = String.Format("{0:dd}.{0:MM}.{0:yyyy}", DateTime.Now);
 break;
 case "format3":
 myDate = String.Format("{0:yyyy}/{0:MM}/{0:dd}", DateTime.Now);
 break;
 }
 return myDate;
 }
}
```

The web service is then saved as `DynamicPopulate.cs.asmx`. Alternatively, you could implement the `getDate()` method as a page method within the actual ASP.NET page with the `DynamicPopulate` control.

In the next step, create a new ASP.NET file. As always, the first step is to include the `ScriptManager` in the current page to load the ASP.NET AJAX library and to make the Control Toolkit work:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, add a label control (for instance using the HTML control of the same name, or the < `asp:Label` /> web control) which will later show the result of the web service call.

```
<label id="myDate" runat="server" />
```

An HTML button (as an HTML control, since we do not require a postback to the server) will then be used to trigger the dynamic population:

```
<input type="button" id="Button1" runat="server" value="Load date (m-d-y)" />
```

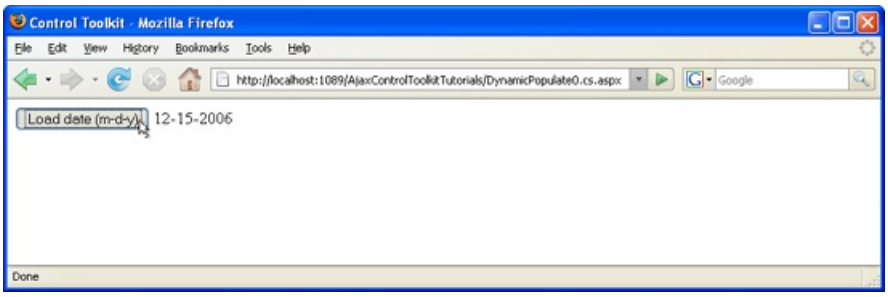
Finally, we need the `DynamicPopulateExtender` control to wire things up. The following attributes will be set (apart from the obvious ones, `ID` and `runat` = `"server"`):

- `TargetControlID` where to put the result from the web service call
- `ServicePath` path to the web service (omit if you want to use a page method)
- `ServiceMethod` name of the web method or page method
- `ContextKey` context information to be sent to the web service
- `PopulateTriggerControlID` element which triggers the web service call
- `ClearContentsDuringUpdate` whether to empty the target element during the web service call

As you can see, the control requires some information but putting everything into place is quite straight-forward. Here is the markup for the `DynamicPopulateExtender` control in the current scenario:

```
<ajaxToolkit:DynamicPopulateExtender ID="dpe1" runat="server"
 ClearContentsDuringUpdate="true"
 TargetControlID="myDate" ServicePath="DynamicPopulate.cs.asmx"
 ServiceMethod="getDate"
 ContextKey="format1" PopulateTriggerControlID="Button1" />
```

Run the ASP.NET page in the browser and click on the button; you will receive the current date in month-day-year format.



A click on the button retrieves the date from the server (Click to view full-size image)

NEXT

Dynamically Populating a Control Using JavaScript Code (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The DynamicPopulate control in the ASP.NET AJAX Control Toolkit calls a web service (or page method) and fills the resulting value into a target control on the page, without a page refresh. It is also possible to trigger the population using custom client-side JavaScript code.

Overview

The `DynamicPopulate` control in the ASP.NET AJAX Control Toolkit calls a web service (or page method) and fills the resulting value into a target control on the page, without a page refresh. It is also possible to trigger the population using custom client-side JavaScript code.

Steps

First of all, you need an ASP.NET Web Service which implements the method to be called by the `DynamicPopulateExtender` control. The web service implements the method `getDate()` that expects one argument of type string, called `contextKey`, since the `DynamicPopulate` control sends one piece of context information with each web service call. Here is the code (file `DynamicPopulate.cs.asmx`) which retrieves the current date in one of three formats:

```
<%@ WebService Language="C#" Class="DynamicPopulate" %>
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Web.Script.Services;
[ScriptService]
public class DynamicPopulate : System.Web.Services.WebService
{
 [WebMethod]
 public string getDate(string contextKey)
 {
 string myDate = "";
 switch (contextKey)
 {
 case "format1":
 myDate = String.Format("{0:MM}-{0:dd}-{0:yyyy}", DateTime.Now);
 break;
 case "format2":
 myDate = String.Format("{0:dd}.{0:MM}.{0:yyyy}", DateTime.Now);
 break;
 case "format3":
 myDate = String.Format("{0:yyyy}/{0:MM}/{0:dd}", DateTime.Now);
 break;
 }
 return myDate;
 }
}
```

In the next step, create a new ASP.NET site and start with the ASP.NET AJAX ScriptManager control:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, add a label control (for instance using the HTML control of the same name, or the `<asp:Label />` web control) which will later show the result of the web service call.

```
<label id="myDate" runat="server" />
```

Next, include a `DynamicPopulateExtender` control and provide web service information, target control, but not the name of the control which triggers the population this will be done later on, using custom JavaScript!

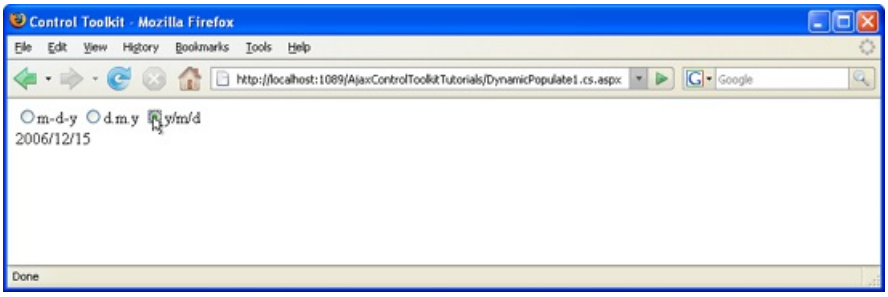
```
<ajaxToolkit:DynamicPopulateExtender ID="dpe1" runat="server"
 ClearContentsDuringUpdate="true"
 TargetControlID="myDate" ServicePath="DynamicPopulate.cs.asmx"
 ServiceMethod="getDate" />
```

Now to the JavaScript part. The `$find()` function, defined by the ASP.NET AJAX library, returns a reference to server-side objects of the ASP.NET AJAX Control Toolkit such as `DynamicPopulateExtender`. In the current file, `$find("dpe")` returns a reference to the one `DynamicPopulateExtender` control in the page. It exposes a method called `populate()` which triggers the dynamic population process. The `populate()` method requires one argument: the context key which will serve as argument to the `getDate()` web method. So for instance, `$find("dpe").populate("format1")` would populate the label with the current date in month-day-year format.

In order to make the sample a bit more flexible, the user may now choose between several date formats. For each one of them, a radio button is displayed. Once the user clicks on a radio button, JavaScript code dynamically populates the label with the selected date format. Here are those radio buttons:

```
<asp:Panel ID="panel1" runat="server">
 <input type="radio" id="rb1" name="format" value="format1" runat="server"
 onclick="$find('dpe1').populate(this.value);" />m-d-y
 <input type="radio" id="rb2" name="format" value="format2" runat="server"
 onclick="$find('dpe1').populate(this.value);" />d.m.y
 <input type="radio" id="rb3" name="format" value="format3" runat="server"
 onclick="$find('dpe1').populate(this.value);" />y/m/d
</asp:Panel>
```

Note that within the context of a radio button, the JavaScript expression `this.value` refers to the value of the current button, which happens to be exactly the same information the `getDate()` method can work with.



A click on the button retrieves the date from the server, in the format specified (Click to view full-size image)

Using DynamicPopulate with a User Control And JavaScript (C#)

2/20/2020 • 3 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The DynamicPopulate control in the ASP.NET AJAX Control Toolkit calls a web service (or page method) and fills the resulting value into a target control on the page, without a page refresh. It is also possible to trigger the population using custom client-side JavaScript code. However special care has to be taken when the extender resides in a user control.

Overview

The `DynamicPopulate` control in the ASP.NET AJAX Control Toolkit calls a web service (or page method) and fills the resulting value into a target control on the page, without a page refresh. It is also possible to trigger the population using custom client-side JavaScript code. However special care has to be taken when the extender resides in a user control.

Steps

First of all, you need an ASP.NET Web Service which implements the method to be called by the `DynamicPopulateExtender` control. The web service implements the method `getDate()` that expects one argument of type string, called `contextKey`, since the `DynamicPopulate` control sends one piece of context information with each web service call. Here is the code (file `DynamicPopulate.cs.asmx`) which retrieves the current date in one of three formats:

```
<%@ WebService Language="C#" Class="DynamicPopulate" %>
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Web.Script.Services;
[ScriptService]
public class DynamicPopulate : System.Web.Services.WebService
{
 [WebMethod]
 public string getDate(string contextKey)
 {
 string myDate = "";
 switch (contextKey)
 {
 case "format1":
 myDate = String.Format("{0:MM}-{0:dd}-{0:yyyy}", DateTime.Now);
 break;
 case "format2":
 myDate = String.Format("{0:dd}.{0:MM}.{0:yyyy}", DateTime.Now);
 break;
 case "format3":
 myDate = String.Format("{0:yyyy}/{0:MM}/{0:dd}", DateTime.Now);
 break;
 }
 return myDate;
 }
}
```

In the next step, create a new user control (`.ascx` file), denoted by the following declaration in its first line:

```
<%@ Control Language="C#" ClassName="DynamicPopulate2" %>
```

A < `label` > element will be used to display the data coming from the server.

```
<label id="myDate" runat="server" />
```

Also in the user control file, we will use three radio buttons, each one representing one of the three possible date formats supported by the web service. When the user clicks on one of the radio buttons, the browser will execute JavaScript code which looks like this:

```
$find("mcd1_dpe1").populate(this.value)
```

This code accesses the `DynamicPopulateExtender` (do not worry about the strange ID yet, this will be covered later on) and triggers the dynamic population with data. In the context of the current radio button, `this.value` refers to its value which is `format1`, `format2` or `format3` exactly what the web method expects.

The only thing missing in the user control yet is the `DynamicPopulateExtender` control which links the radio buttons to the web service.

```
<ajaxToolkit:DynamicPopulateExtender ID="dpe1" runat="server"
 ClearContentsDuringUpdate="true"
 TargetControlID="mcd1$myDate"
 ServicePath="DynamicPopulate.cs.asmx" ServiceMethod="getDate"/>
```

Again you may note the strange ID used in the control: `mcd1$myDate` instead of `myDate`. Previously, the JavaScript code used `mcd1_dpe1` to access the `DynamicPopulateExtender` instead of `dpe1`.This naming strategy is a special

requirement when using `DynamicPopulateExtender` within a user control. Furthermore, you have to embed the user control in a specific way to make it all work. Create a new ASP.NET page and register a tag prefix for the user control you have just implemented:

```
<%@ Register TagPrefix="uc1" TagName="myCustomDate" Src="~/DynamicPopulate2.cs.ascx"%>
```

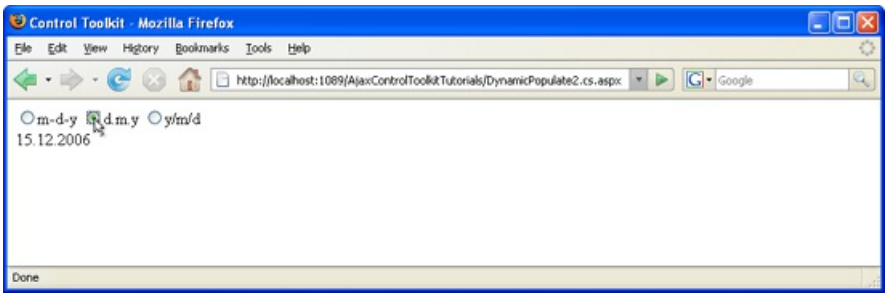
Then, include the ASP.NET AJAX `ScriptManager` control on the new page:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Finally, add the user control to the page. You only have to set its `ID` attribute (and `runat="server"`, of course), but you also have to set it to a specific name: `mcd1` since this is the prefix used within the user control to access it using JavaScript.

```
<div>
 <uc1:myCustomDate ID="mcd1" runat="server" />
</div>
```

And that's it! The page behaves as expected: A user clicks on one of the radio buttons, the control in the Toolkit calls the web service and displays the current date in the desired format.



The radio buttons reside in a user control (Click to view full-size image)

PREVIOUS NEXT

Dynamically Populating a Control (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The DynamicPopulate control in the ASP.NET AJAX Control Toolkit calls a web service (or page method) and fills the resulting value into a target control on the page, without a page refresh.

Overview

The `DynamicPopulate` control in the ASP.NET AJAX Control Toolkit calls a web service (or page method) and fills the resulting value into a target control on the page, without a page refresh. This tutorial shows how to set this up.

Steps

First of all, you need an ASP.NET Web Service which implements the method to be called by `DynamicPopulate`. The web service class requires the `ScriptService` attribute which is defined within `Microsoft.Web.Script.Services`; otherwise ASP.NET AJAX cannot create the client-side JavaScript proxy for the web service which in turn is required by `DynamicPopulate`.

The web method must expect one argument of type string, called `contextKey`, since the `DynamicPopulate` control sends one piece of context information with each web service call. The following web service returns the current date in a format represented by the `contextKey` argument:

```
<%@ WebService Language="VB" Class="DynamicPopulate" %>
Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols
Imports System.Web.Script.Services
<ScriptService()> _
Public Class DynamicPopulate
 Inherits System.Web.Services.WebService
 <WebMethod()> _
 Public Function getDate(ByVal contextKey As String) As String
 Dim myDate As String = ""
 Select Case contextKey
 Case "format1"
 myDate = String.Format("{0:MM}-{0:dd}-{0:yyyy}", DateTime.Now)
 Case "format2"
 myDate = String.Format("{0:dd}.{0:MM}.{0:yyyy}", DateTime.Now)
 Case "format3"
 myDate = String.Format("{0:yyyy}/{0:MM}/{0:dd}", DateTime.Now)
 End Select
 Return myDate
 End Function
End Class
```

The web service is then saved as `DynamicPopulate.vb.asmx`. Alternatively, you could implement the `getDate()` method as a page method within the actual ASP.NET page with the `DynamicPopulate` control.

In the next step, create a new ASP.NET file. As always, the first step is to include the `ScriptManager` in the current page to load the ASP.NET AJAX library and to make the Control Toolkit work:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, add a label control (for instance using the HTML control of the same name, or the < `asp:Label` /> web control) which will later show the result of the web service call.

```
<label id="myDate" runat="server" />
```

An HTML button (as an HTML control, since we do not require a postback to the server) will then be used to trigger the dynamic population:

```
<input type="button" id="Button1" runat="server" value="Load date (m-d-y)" />
```

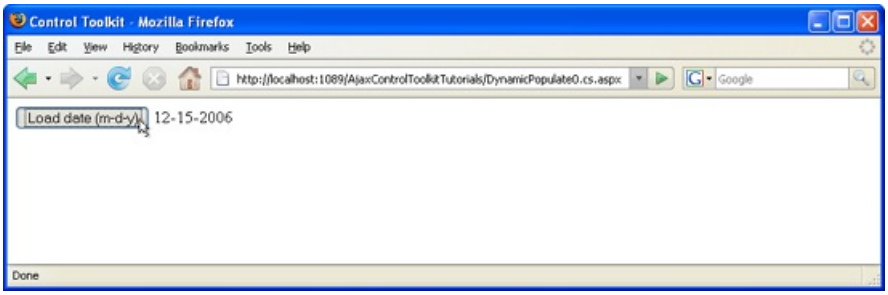
Finally, we need the `DynamicPopulateExtender` control to wire things up. The following attributes will be set (apart from the obvious ones, `ID` and `runat` = `"server"`):

- `TargetControlID` where to put the result from the web service call
- `ServicePath` path to the web service (omit if you want to use a page method)
- `ServiceMethod` name of the web method or page method
- `ContextKey` context information to be sent to the web service
- `PopulateTriggerControlID` element which triggers the web service call
- `ClearContentsDuringUpdate` whether to empty the target element during the web service call

As you can see, the control requires some information but putting everything into place is quite straight-forward. Here is the markup for the `DynamicPopulateExtender` control in the current scenario:

```
<ajaxToolkit:DynamicPopulateExtender ID="dpe1" runat="server"
 ClearContentsDuringUpdate="true"
 TargetControlID="myDate" ServicePath="DynamicPopulate.vb.asmx"
 ServiceMethod="getDate"
 ContextKey="format1" PopulateTriggerControlID="Button1" />
```

Run the ASP.NET page in the browser and click on the button; you will receive the current date in month-day-year format.



A click on the button retrieves the date from the server (Click to view full-size image)

PREVIOUS NEXT

Dynamically Populating a Control Using JavaScript Code (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The DynamicPopulate control in the ASP.NET AJAX Control Toolkit calls a web service (or page method) and fills the resulting value into a target control on the page, without a page refresh. It is also possible to trigger the population using custom client-side JavaScript code.

Overview

The `DynamicPopulate` control in the ASP.NET AJAX Control Toolkit calls a web service (or page method) and fills the resulting value into a target control on the page, without a page refresh. It is also possible to trigger the population using custom client-side JavaScript code.

Steps

First of all, you need an ASP.NET Web Service which implements the method to be called by the `DynamicPopulateExtender` control. The web service implements the method `getDate()` that expects one argument of type string, called `contextKey`, since the `DynamicPopulate` control sends one piece of context information with each web service call. Here is the code (file `DynamicPopulate.vb.asmx`) which retrieves the current date in one of three formats:

```
<%@ WebService Language="VB" Class="DynamicPopulate" %>
Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols
Imports System.Web.Script.Services
<ScriptService()> _
Public Class DynamicPopulate
 Inherits System.Web.Services.WebService
 <WebMethod()> _
 Public Function getDate(ByVal contextKey As String) As String
 Dim myDate As String = ""
 Select Case contextKey
 Case "format1"
 myDate = String.Format("{0:MM}-{0:dd}-{0:yyyy}", DateTime.Now)
 Case "format2"
 myDate = String.Format("{0:dd}.{0:MM}.{0:yyyy}", DateTime.Now)
 Case "format3"
 myDate = String.Format("{0:yyyy}/{0:MM}/{0:dd}", DateTime.Now)
 End Select
 Return myDate
 End Function
End Class
```

In the next step, create a new ASP.NET site and start with the ASP.NET AJAX ScriptManager control:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, add a label control (for instance using the HTML control of the same name, or the `<asp:Label />` web control) which will later show the result of the web service call.

```
<label id="myDate" runat="server" />
```

Next, include a `DynamicPopulateExtender` control and provide web service information, target control, but not the name of the control which triggers the population this will be done later on, using custom JavaScript!

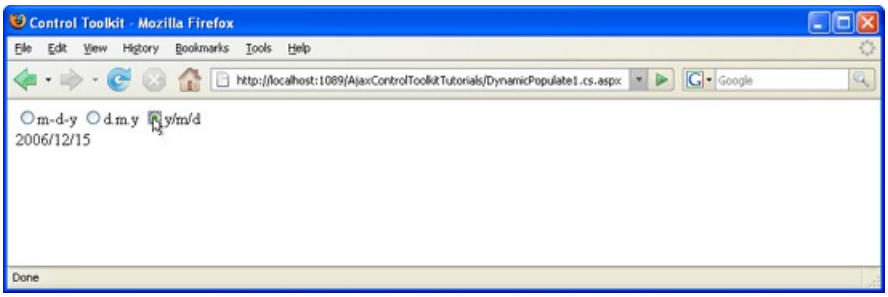
```
<ajaxToolkit:DynamicPopulateExtender ID="dpe1" runat="server"
 ClearContentsDuringUpdate="true"
 TargetControlID="myDate" ServicePath="DynamicPopulate.vb.asmx"
 ServiceMethod="getDate" />
```

Now to the JavaScript part. The `$find()` function, defined by the ASP.NET AJAX library, returns a reference to server-side objects of the ASP.NET AJAX Control Toolkit such as `DynamicPopulateExtender`. In the current file, `$find("dpe")` returns a reference to the one `DynamicPopulateExtender` control in the page. It exposes a method called `populate()` which triggers the dynamic population process. The `populate()` method requires one argument: the context key which will serve as argument to the `getDate()` web method. So for instance, `$find("dpe").populate("format1")` would populate the label with the current date in month-day-year format.

In order to make the sample a bit more flexible, the user may now choose between several date formats. For each one of them, a radio button is displayed. Once the user clicks on a radio button, JavaScript code dynamically populates the label with the selected date format. Here are those radio buttons:

```
<asp:Panel ID="panel1" runat="server">
 <input type="radio" id="rb1" name="format" value="format1" runat="server"
 onclick="$find('dpe1').populate(this.value);" />m-d-y
 <input type="radio" id="rb2" name="format" value="format2" runat="server"
 onclick="$find('dpe1').populate(this.value);" />d.m.y
 <input type="radio" id="rb3" name="format" value="format3" runat="server"
 onclick="$find('dpe1').populate(this.value);" />y/m/d
</asp:Panel>
```

Note that within the context of a radio button, the JavaScript expression `this.value` refers to the value of the current button, which happens to be exactly the same information the `getDate()` method can work with.



A click on the button retrieves the date from the server, in the format specified (Click to view full-size image)

PREVIOUS NEXT

Using DynamicPopulate with a User Control And JavaScript (VB)

2/20/2020 • 3 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The DynamicPopulate control in the ASP.NET AJAX Control Toolkit calls a web service (or page method) and fills the resulting value into a target control on the page, without a page refresh. It is also possible to trigger the population using custom client-side JavaScript code. However special care has to be taken when the extender resides in a user control.

Overview

The `DynamicPopulate` control in the ASP.NET AJAX Control Toolkit calls a web service (or page method) and fills the resulting value into a target control on the page, without a page refresh. It is also possible to trigger the population using custom client-side JavaScript code. However special care has to be taken when the extender resides in a user control.

Steps

First of all, you need an ASP.NET Web Service which implements the method to be called by the `DynamicPopulateExtender` control. The web service implements the method `getDate()` that expects one argument of type string, called `contextKey`, since the `DynamicPopulate` control sends one piece of context information with each web service call. Here is the code (files `DynamicPopulate.vb.asmx`) which retrieves the current date in one of three formats:

```
<%@ WebService Language="VB" Class="DynamicPopulate" %>
Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols
Imports System.Web.Script.Services
<ScriptService()> _
Public Class DynamicPopulate
 Inherits System.Web.Services.WebService
 <WebMethod()> _
 Public Function getDate(ByVal contextKey As String) As String
 Dim myDate As String = ""
 Select Case contextKey
 Case "format1"
 myDate = String.Format("{0:MM}-{0:dd}-{0:yyyy}", DateTime.Now)
 Case "format2"
 myDate = String.Format("{0:dd}.{0:MM}.{0:yyyy}", DateTime.Now)
 Case "format3"
 myDate = String.Format("{0:yyyy}/{0:MM}/{0:dd}", DateTime.Now)
 End Select
 Return myDate
 End Function
End Class
```

In the next step, create a new user control (`.ascx` file), denoted by the following declaration in its first line:

```
<%@ Control Language="C#" ClassName="DynamicPopulate2" %>
```

A < `label` > element will be used to display the data coming from the server.

```
<label id="myDate" runat="server" />
```

Also in the user control file, we will use three radio buttons, each one representing one of the three possible date formats supported by the web service. When the user clicks on one of the radio buttons, the browser will execute JavaScript code which looks like this:

```
$find("mcd1_dpe1").populate(this.value)
```

This code accesses the `DynamicPopulateExtender` (do not worry about the strange ID yet, this will be covered later on) and triggers the dynamic population with data. In the context of the current radio button, `this.value` refers to its value which is `format1`, `format2` or `format3` exactly what the web method expects.

The only thing missing in the user control yet is the `DynamicPopulateExtender` control which links the radio buttons to the web service.

```
<ajaxToolkit:DynamicPopulateExtender ID="dpe1" runat="server"
 ClearContentsDuringUpdate="true"
 TargetControlID="mcd1$myDate"
 ServicePath="DynamicPopulate.vb.asmx" ServiceMethod="getDate"/>
```

Again you may note the strange ID used in the control: `mcd1$myDate` instead of `myDate`. Previously, the JavaScript code used `mcd1_dpe1` to access the `DynamicPopulateExtender` instead of `dpe1`.This naming strategy is a special requirement when using `DynamicPopulateExtender` within a user control. Furthermore, you have to embed the user control in a specific way to make it all work. Create a new ASP.NET page and register a tag prefix for the user control you have just implemented:

```
<%@ Register TagPrefix="uc1" TagName="myCustomDate" Src="~/DynamicPopulate2.vb.ascx"%>
```

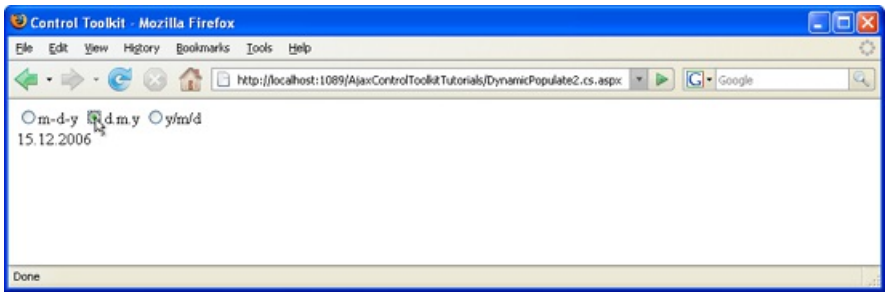
Then, include the ASP.NET AJAX `ScriptManager` control on the new page:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Finally, add the user control to the page. You only have to set its `ID` attribute (and `runat="server"`, of course), but you also have to set it to a specific name: `mcd1` since this is the prefix used within the user control to access it using JavaScript.

```
<div>
 <uc1:myCustomDate ID="mcd1" runat="server" />
</div>
```

And that's it! The page behaves as expected: A user clicks on one of the radio buttons, the control in the Toolkit calls the web service and displays the current date in the desired format.



The radio buttons reside in a user control (Click to view full-size image)

PREVIOUS

FilteredTextBox

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the FilteredTextBox control, which lets you specify which characters are allowed in user input.

- Allowing Only Certain Characters in a Text Box (C#)
- Allowing Only Certain Characters in a Text Box (VB)

Allowing Only Certain Characters in a Text Box (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> ASP.NET validation controls can ensure that only certain characters are allowed in user input. However this still does not prevent users from typing invalid characters and trying to submit the form.

Overview

ASP.NET validation controls can ensure that only certain characters are allowed in user input. However this still does not prevent users from typing invalid characters and trying to submit the form.

Steps

The ASP.NET AJAX Control Toolkit contains the `FilteredTextBox` control which extends a text box. Once activated, only a certain set of characters may be entered into the field.

For this to work, we first need as usual the ASP.NET AJAX `ScriptManager` which loads the JavaScript libraries which are also used by the ASP.NET AJAX Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, we need a text box:

```
Numbers only: <asp:TextBox ID="TextBox1" runat="server" />
```

Finally, the `FilteredTextBoxExtender` control takes care of restricting the characters the user is allowed to type. First, set the `TargetControlID` attribute to the `ID` of the `TextBox` control. Then, choose one of the available `FilterType` values:

- `Custom` default; you have to provide a list of valid chars
- `LowercaseLetters` lowercase letters only
- `Numbers` digits only
- `UppercaseLetters` uppercase letters only

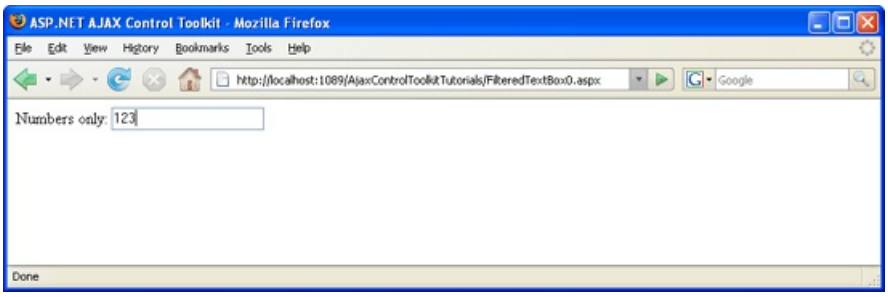
If the `Custom FilterType` is used, the `ValidChars` property must be set and provide a list of characters that may be typed. By the way: if you try to paste text into the text box, all invalid chars are removed.

Here is the markup for the `FilteredTextBoxExtender` control that only allows digits (something that would also have been possible with `FilterType="Numbers"`):

```
<ajaxToolkit:FilteredTextBoxExtender ID="ftbe" runat="server"
 TargetControlID="TextBox1" ValidChars="1234567890" />
```

Run the page and try to enter a letter if JavaScript is enabled, it will not work; digits however appear on the page. However note that the protection `FilteredTextBox` provides is not bullet-proof: If JavaScript is enabled, any data

may be entered in the text box, so you have to use additional validation means, i.e. ASP.NET's validation controls.



Only digits may be entered (Click to view full-size image)

NEXT

Allowing Only Certain Characters in a Text Box (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> ASP.NET validation controls can ensure that only certain characters are allowed in user input. However this still does not prevent users from typing invalid characters and trying to submit the form.

Overview

ASP.NET validation controls can ensure that only certain characters are allowed in user input. However this still does not prevent users from typing invalid characters and trying to submit the form.

Steps

The ASP.NET AJAX Control Toolkit contains the `FilteredTextBox` control which extends a text box. Once activated, only a certain set of characters may be entered into the field.

For this to work, we first need as usual the ASP.NET AJAX `ScriptManager` which loads the JavaScript libraries which are also used by the ASP.NET AJAX Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, we need a text box:

```
Numbers only: <asp:TextBox ID="TextBox1" runat="server" />
```

Finally, the `FilteredTextBoxExtender` control takes care of restricting the characters the user is allowed to type. First, set the `TargetControlID` attribute to the `ID` of the `TextBox` control. Then, choose one of the available `FilterType` values:

- `Custom` default; you have to provide a list of valid chars
- `LowercaseLetters` lowercase letters only
- `Numbers` digits only
- `UppercaseLetters` uppercase letters only

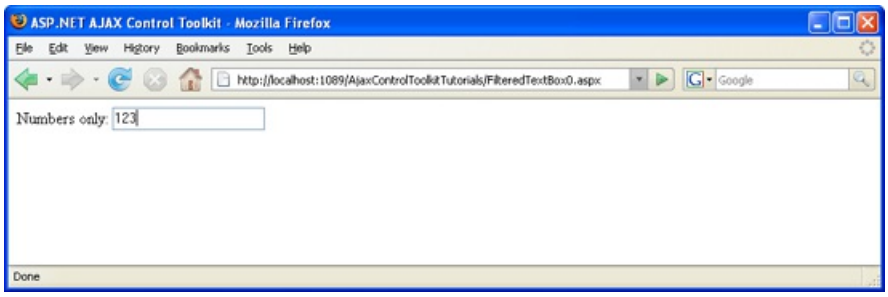
If the `Custom FilterType` is used, the `ValidChars` property must be set and provide a list of characters that may be typed. By the way: if you try to paste text into the text box, all invalid chars are removed.

Here is the markup for the `FilteredTextBoxExtender` control that only allows digits (something that would also have been possible with `FilterType="Numbers"`):

```
<ajaxToolkit:FilteredTextBoxExtender ID="ftbe" runat="server"
 TargetControlID="TextBox1" ValidChars="1234567890" />
```

Run the page and try to enter a letter if JavaScript is enabled, it will not work; digits however appear on the page. However note that the protection `FilteredTextBox` provides is not bullet-proof: If JavaScript is enabled, any data

may be entered in the text box, so you have to use additional validation means, i.e. ASP.NET's validation controls.



Only digits may be entered (Click to view full-size image)

PREVIOUS

HoverMenu

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the HoverMenu control, which displays a popup when users hold the mouse pointer over an element.

- Using HoverMenu with a Repeater Control (C#)
- Using HoverMenu with a Repeater Control (VB)

Using HoverMenu with a Repeater Control (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The HoverMenu control in the AJAX Control Toolkit provides a simple popup effect: When the mouse pointer hovers over an element, a popup appears at a specified position. It is also possible to use this control within a repeater.

Overview

The `HoverMenu` control in the AJAX Control Toolkit provides a simple popup effect: When the mouse pointer hovers over an element, a popup appears at a specified position. It is also possible to use this control within a repeater.

Steps

First of all, a data source is required. This sample uses the AdventureWorks database and the Microsoft SQL Server 2005 Express Edition. The database is an optional part of a Visual Studio installation (including express edition) and is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. The AdventureWorks database is part of the SQL Server 2005 Samples and Sample Databases (download at https://www.microsoft.com/downloads/details.aspx?FamilyID=e719ecf7-9f46-4312-af89-6ad8702e4e6e&DisplayLang=en). The easiest way to set the database up is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en) and attach the `AdventureWorks.mdf` database file.

For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database.

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, add a data source to the page. In order to use a limited amount of data, we only select the first five entries in the Vendor table of the AdventureWorks database. If you are using the Visual Studio assistant to create the data source, mind that a bug in the current version does not prefix the table name (`Vendor`) with `Purchasing`. The following markup shows the correct syntax:

```
<asp:SqlDataSource ID="sds1" runat="server" ConnectionString="
 Data Source=(local)\SQLEXPRESS;Initial Catalog=AdventureWorks;Integrated Security=True"
 ProviderName="System.Data.SqlClient" SelectCommand="SELECT TOP 5
 [VendorID], [Name] FROM [Purchasing].[Vendor]" />
```

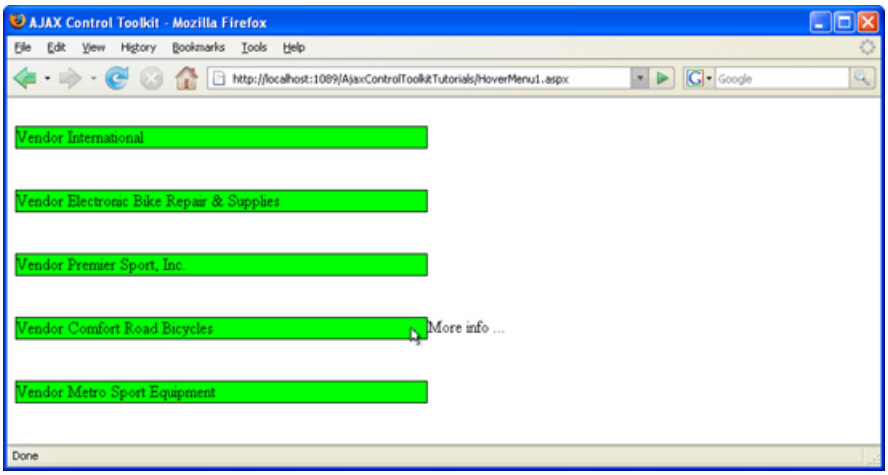
Next, add a panel which serves as the modal popup:

```
<asp:Panel ID="HoverPanel" runat="server">
 More info ...
</asp:Panel>
```

Now, the `HoverMenuExtender` comes into play. So that every element in the data source gets its own popup, the extender must be put within the repeater's `<ItemTemplate>` section. Here is the markup:

```
<asp:Repeater ID="rep1" DataSourceID="sds1" runat="server">
 <ItemTemplate>
 <br />
 <asp:Panel ID="myPanel" runat="server" Width="400px" BackColor="Lime" BorderWidth="1px">
 <div>
 Vendor
 <%#DataBinder.Eval(Container.DataItem, "Name")%>
 </div>
 </asp:Panel>
 <br />
 <ajaxToolkit:HoverMenuExtender ID="hme" runat="server" TargetControlID="myPanel"
 PopupControlID="HoverPanel" PopupPosition="Right" PopDelay="50" />
 </ItemTemplate>
</asp:Repeater>
```

Now every item in the data source displays a popup to the right (`PopupPosition` attribute) after a delay of 50 milliseconds (`PopDelay` attribute).



The hover menu appears next to each item in the repeater (Click to view full-size image)

NEXT

Using HoverMenu with a Repeater Control (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The HoverMenu control in the AJAX Control Toolkit provides a simple popup effect: When the mouse pointer hovers over an element, a popup appears at a specified position. It is also possible to use this control within a repeater.

Overview

The `HoverMenu` control in the AJAX Control Toolkit provides a simple popup effect: When the mouse pointer hovers over an element, a popup appears at a specified position. It is also possible to use this control within a repeater.

Steps

First of all, a data source is required. This sample uses the AdventureWorks database and the Microsoft SQL Server 2005 Express Edition. The database is an optional part of a Visual Studio installation (including express edition) and is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. The AdventureWorks database is part of the SQL Server 2005 Samples and Sample Databases (download at https://www.microsoft.com/downloads/details.aspx?FamilyID=e719ecf7-9f46-4312-af89-6ad8702e4e6e&DisplayLang=en). The easiest way to set the database up is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en) and attach the `AdventureWorks.mdf` database file.

For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database.

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, add a data source to the page. In order to use a limited amount of data, we only select the first five entries in the Vendor table of the AdventureWorks database. If you are using the Visual Studio assistant to create the data source, mind that a bug in the current version does not prefix the table name (`Vendor`) with `Purchasing`. The following markup shows the correct syntax:

```
<asp:SqlDataSource ID="sds1" runat="server" ConnectionString="
 Data Source=(local)\SQLEXPRESS;Initial Catalog=AdventureWorks;Integrated Security=True"
 ProviderName="System.Data.SqlClient" SelectCommand="SELECT TOP 5
 [VendorID], [Name] FROM [Purchasing].[Vendor]" />
```

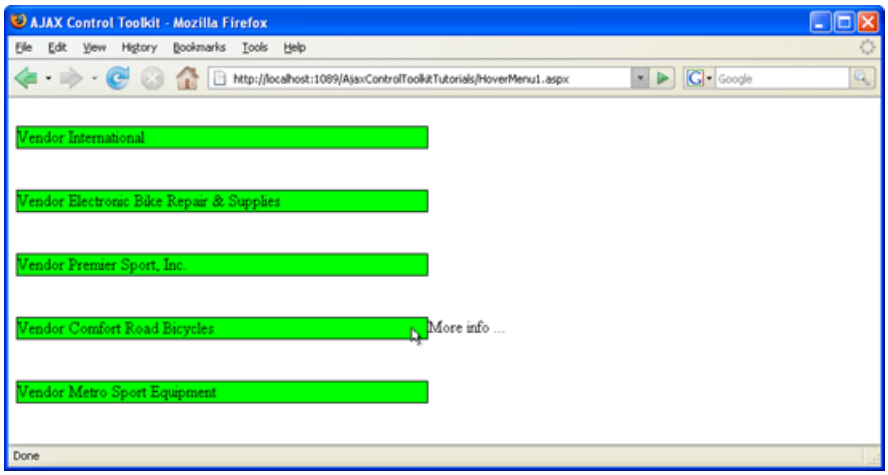
Next, add a panel which serves as the modal popup:

```
<asp:Panel ID="HoverPanel" runat="server">
 More info ...
</asp:Panel>
```

Now, the `HoverMenuExtender` comes into play. So that every element in the data source gets its own popup, the extender must be put within the repeater's `<ItemTemplate>` section. Here is the markup:

```
<asp:Repeater ID="rep1" DataSourceID="sds1" runat="server">
 <ItemTemplate>
 <br />
 <asp:Panel ID="myPanel" runat="server" Width="400px" BackColor="Lime" BorderWidth="1px">
 <div>
 Vendor
 <%#DataBinder.Eval(Container.DataItem, "Name")%>
 </div>
 </asp:Panel>
 <br />
 <ajaxToolkit:HoverMenuExtender ID="hme" runat="server" TargetControlID="myPanel"
 PopupControlID="HoverPanel" PopupPosition="Right" PopDelay="50" />
 </ItemTemplate>
</asp:Repeater>
```

Now every item in the data source displays a popup to the right (`PopupPosition` attribute) after a delay of 50 milliseconds (`PopDelay` attribute).



The hover menu appears next to each item in the repeater (Click to view full-size image)

PREVIOUS

HTMLEditor

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the HTMLEditor control, which provides a rich editor and WYSIWYG or source editing of HTML content.

- HTML Editor Control (C#)
- HTML Editor Control (VB)

How do I use the HTML Editor Control? (C#)

4/10/2020 • 5 minutes to read • Edit Online

by Microsoft

> HTMLEditor is an ASP.NET AJAX Control that allows you to easily create and edit HTML content via buttons in a toolbar.

The goal of this tutorial is to provide you with an overview of the HTML Editor control included with the AJAX Control Toolkit. The HTML Editor includes options for changing font size, selecting a font, changing background color, modifying the foreground color, adding links, adding images, changing text alignment, and performing cut, copy, and paste operations (see Figure 1).

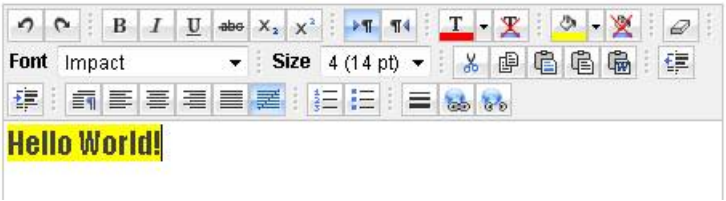


Figure 01: The HTML Editor(Click to view full-size image)

The HTML editor enables you to enter content using a design mode or you can enter HTML directly. You also are provided with the option to preview your HTML content (see Figure 2).

Figure 02: Design, HTML, and Preview buttons(Click to view full-size image)

In this tutorial, you learn how to display the HTML Editor, how to customize the toolbar buttons that appear in the HTML Editor, and how to avoid Cross-Site Scripting Attacks.

Displaying the HTML Editor

Before you can use the HTML Editor in an ASP.NET page, you must first add a ScriptManager control to the page. The ScriptManager control is located beneath the AJAX Extensions tab in the Visual Studio/Visual Web Developer Express toolbox.

You should place the ScriptManager control at the top of the page before any other controls on the page. For example, you can place it immediately below the opening server-side <form> tag.

The HTML Editor control is located in the toolbox with the rest of the AJAX Control Toolkit controls. It is named the Editor control (see Figure 3).

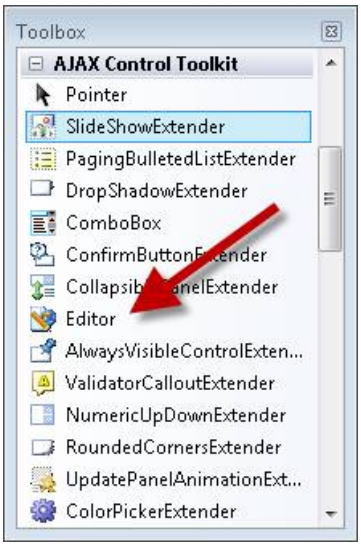


Figure 03: The HTML Editor control(Click to view full-size image)

After you drag the HTML Editor onto a page, you can set its properties in the property sheet. For example, you normally want to set the Width and Height properties. Listing 1 contains the source for an ASP.NET page that contains an HTML editor.

Listing 1 - SimpleEditor.aspx

```
<%@ Page Language="C#" %>
<%@ Register assembly="AjaxControlToolkit" namespace="AjaxControlToolkit.HTMLEditor" tagprefix="cc1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<script runat="server">

    protected void btnSubmit_Click(object sender, EventArgs e)
    {
        ltlResult.Text = Editor1.Content;
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        <asp:ScriptManager ID="ScriptManager1" runat="server" />

        <cc1:Editor
            ID="Editor1"
            Width="450px"
            Height="200px"
            runat="server"/>
        <br />
        <asp:Button
            id="btnSubmit"
            Text="Submit"
            Runat="server" onclick="btnSubmit_Click" />

        <hr />
        <h1>You Entered:</h1>

        <asp:Literal
            id="ltlResult"
            Runat="server" />

    </div>
    </form>
</body>
</html>
```

The page in Listing 1 contains an HTML Editor control, a Button control, and a Literal control. When you click the button, the contents of the HTML Editor appear in the Literal control (see Figure 4).

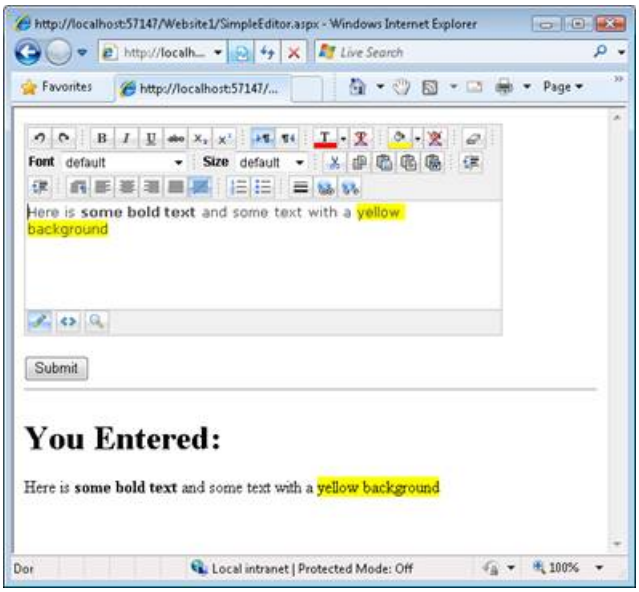


Figure 04: Submitting a form with an HTML Editor(Click to view full-size image)

The HTML Editor Content property is used to retrieve the HTML content entered into the HTML Editor. Be aware that this HTML content can contain JavaScript. In the next section, we discuss how you can prevent JavaScript Injection Attacks.

Customizing the HTML Editor Toolbar

You can customize exactly which buttons appear in the editor. For example, you might want to remove the HTML tab to prevent users from switching the HTML Editor into HTML mode. Or, you might want to remove the font size dropdown list to prevent users from creating overly large text in a forum message post (see Figure 5).

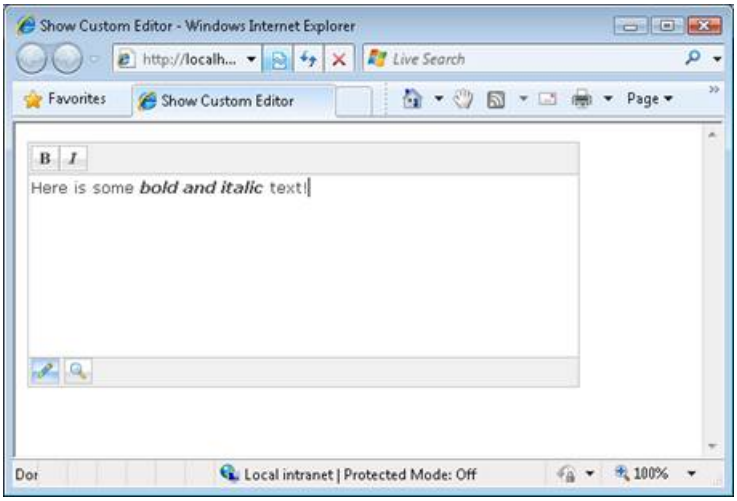


Figure 05: A customized HTML Editor(Click to view full-size image)

You customize the toolbar buttons by deriving a new HTML Editor from the base Editor class. For example, the custom editor in Listing 2 only contains toolbar buttons for bold and italic. All other toolbar buttons have been removed. Furthermore, the HTML tab has been removed from the bottom of the editor (but the Design and Preview tabs are still there).

Listing 2 - App_Code\CustomEditor.cs

```
using AjaxControlToolkit.HTMLEditor;

namespace MyControls
{
    public class CustomEditor : Editor
    {
        protected override void FillTopToolbar()
        {
            TopToolbar.Buttons.Add(new AjaxControlToolkit.HTMLEditor.ToolbarButton.Bold());
            TopToolbar.Buttons.Add(new AjaxControlToolkit.HTMLEditor.ToolbarButton.Italic());
        }

        protected override void FillBottomToolbar()
        {
            BottomToolbar.Buttons.Add(new AjaxControlToolkit.HTMLEditor.ToolbarButton.DesignMode());
            BottomToolbar.Buttons.Add(new AjaxControlToolkit.HTMLEditor.ToolbarButton.PreviewMode());
        }
    }
}
```

You must add the class in Listing 2 to your App_Code folder so that the class will be compiled automatically. If the App_Code folder does not exist in your website then you can simply add the folder.

After you create a custom editor, you can add it to an ASP.NET page in the same way as you add the normal HTML Editor (see Listing 3).

Listing 3 - ShowCustomEditor.aspx

```
<%@ Page Language="C#" %>
<%@ Register namespace="MyControls" tagprefix="custom" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>Show Custom Editor</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    <asp:ScriptManager ID="ScriptManager1" runat="server" />

    <custom:CustomEditor
        Width="450px"
        Height="200px"
        runat="server" />

    </div>
    </form>
</body>
</html>
```

Avoiding Cross-Site Scripting (XSS) Attacks

Whenever you accept input from a user, and redisplay that input on your website, you potentially open your website to Cross-Site Scripting (XSS) Attacks. In theory, a malicious hacker could submit JavaScript code that gets executed when the input is redisplayed. The JavaScript could be used to steal user passwords or other sensitive information.

Normally, you can defeat XSS attacks by HTML encoding whatever input you retrieve from a user before displaying it in a web page. However, HTML encoding the output of the HTML Editor would not only encode

<script> tags, it would also encode all HTML tags. In other words, you would lose all of the formatting such as the font type, font size, and background color.

If you are collecting sensitive information from your users -- such as passwords, credit-card numbers, and social security numbers - then you should not display un-encoded content that you retrieve from a user with the HTML Editor. You should use the HTML Editor only in situations in which you are not redisplaying the HTML content, or the HTML content is being submitted to your website by a trusted party.

Imagine, for example, that you are creating a blog application. In this situation, it makes sense to use the HTML Editor when composing blog posts. You are the only one who submits a blog post and, presumably, you can trust yourself not to submit malicious JavaScript. However, it does not make sense to use the HTML Editor when allowing anonymous users to post comments. You should be especially careful in situations in which users submit sensitive information such as passwords. Potentially, a malicious user could post a comment that contains the right JavaScript for stealing a password.

Summary

In this tutorial, you were provided with a brief overview of the HTML Editor control included in the AJAX Control Toolkit. You learned how to use the HTML Editor to accept rich content from a user and submit the content to the server. We also discussed how you can customize the toolbar buttons that are displayed by the HTML Editor. Finally, you learned how to avoid Cross-Site Scripting Attacks when using the HTML Editor to accept potentially malicious input.

NEXT

How do I use the HTML Editor Control? (VB)

4/10/2020 • 5 minutes to read • Edit Online

by Microsoft

> HTMLEditor is an ASP.NET AJAX Control that allows you to easily create and edit HTML content via buttons in a toolbar.

The goal of this tutorial is to provide you with an overview of the HTML Editor control included with the AJAX Control Toolkit. The HTML Editor includes options for changing font size, selecting a font, changing background color, modifying the foreground color, adding links, adding images, changing text alignment, and performing cut, copy, and paste operations (see Figure 1).

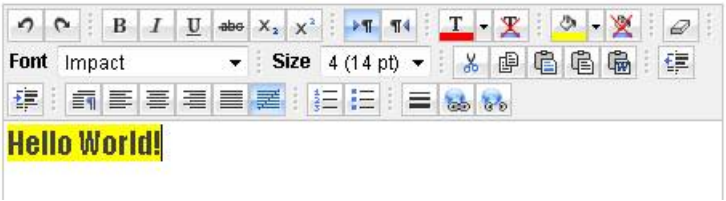


Figure 01: The HTML Editor(Click to view full-size image)

The HTML editor enables you to enter content using a design mode or you can enter HTML directly. You also are provided with the option to preview your HTML content (see Figure 2).

Figure 02: Design, HTML, and Preview buttons(Click to view full-size image)

In this tutorial, you learn how to display the HTML Editor, how to customize the toolbar buttons that appear in the HTML Editor, and how to avoid Cross-Site Scripting Attacks.

Displaying the HTML Editor

Before you can use the HTML Editor in an ASP.NET page, you must first add a ScriptManager control to the page. The ScriptManager control is located beneath the AJAX Extensions tab in the Visual Studio/Visual Web Developer Express toolbox.

You should place the ScriptManager control at the top of the page before any other controls on the page. For example, you can place it immediately below the opening server-side <form> tag.

The HTML Editor control is located in the toolbox with the rest of the AJAX Control Toolkit controls. It is named the Editor control (see Figure 3).

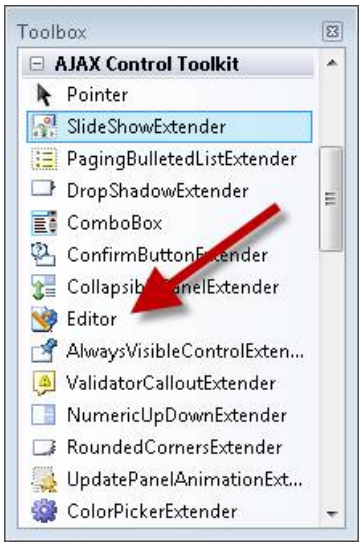


Figure 03: The HTML Editor control(Click to view full-size image)

After you drag the HTML Editor onto a page, you can set its properties in the property sheet. For example, you normally want to set the Width and Height properties. Listing 1 contains the source for an ASP.NET page that contains an HTML editor.

Listing 1 - SimpleEditor.aspx

```
<%@ Page Language="VB" %>

<%@ Register Assembly="AjaxControlToolkit" Namespace="AjaxControlToolkit" TagPrefix="cc2" %>
<%@ Register assembly="AjaxControlToolkit" namespace="AjaxControlToolkit.HTMLEditor" tagprefix="cc1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<script runat="server">

    Protected Sub btnSubmit_Click(ByVal sender As Object, ByVal e As System.EventArgs)
        ltlResult.Text = Editor1.Content
    End Sub
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title></title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        <asp:ScriptManager ID="ScriptManager1" runat="server" />

        <cc1:Editor
            ID="Editor1"
            Width="450px"
            Height="200px"
            runat="server"/>
        <br />
        <asp:Button
            id="btnSubmit"
            Text="Submit"
            Runat="server" onclick="btnSubmit_Click" />

        <hr />
        <h1>You Entered:</h1>

        <asp:Literal
            id="ltlResult"
            Runat="server" />

    </div>
    </form>
</body>
</html>
```

The page in Listing 1 contains an HTML Editor control, a Button control, and a Literal control. When you click the button, the contents of the HTML Editor appear in the Literal control (see Figure 4).

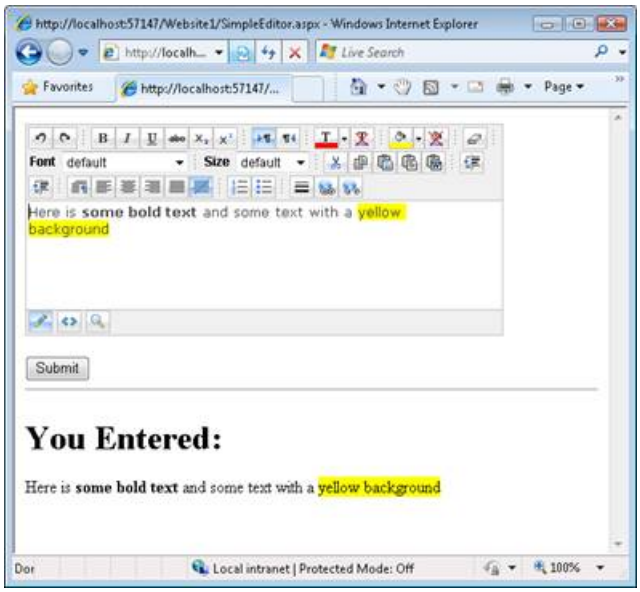


Figure 04: Submitting a form with an HTML Editor(Click to view full-size image)

The HTML Editor Content property is used to retrieve the HTML content entered into the HTML Editor. Be aware that this HTML content can contain JavaScript. In the next section, we discuss how you can prevent JavaScript Injection Attacks.

Customizing the HTML Editor Toolbar

You can customize exactly which buttons appear in the editor. For example, you might want to remove the HTML tab to prevent users from switching the HTML Editor into HTML mode. Or, you might want to remove the font size dropdown list to prevent users from creating overly large text in a forum message post (see Figure 5).

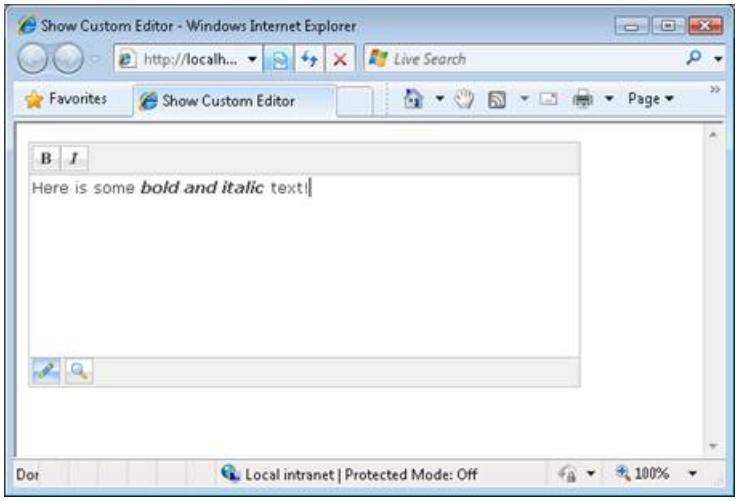


Figure 05: A customized HTML Editor(Click to view full-size image)

You customize the toolbar buttons by deriving a new HTML Editor from the base Editor class. For example, the custom editor in Listing 2 only contains toolbar buttons for bold and italic. All other toolbar buttons have been removed. Furthermore, the HTML tab has been removed from the bottom of the editor (but the Design and Preview tabs are still there).

Listing 2 - App_Code\CustomEditor.vb

```
Imports AjaxControlToolkit.HTMLEditor

Namespace MyControls

    Public Class CustomEditor
        Inherits Editor
        Protected Overrides Sub FillTopToolbar()
            TopToolbar.Buttons.Add(New AjaxControlToolkit.HTMLEditor.ToolbarButton.Bold())
            TopToolbar.Buttons.Add(New AjaxControlToolkit.HTMLEditor.ToolbarButton.Italic())
        End Sub

        Protected Overrides Sub FillBottomToolbar()
            BottomToolbar.Buttons.Add(New AjaxControlToolkit.HTMLEditor.ToolbarButton.DesignMode())
            BottomToolbar.Buttons.Add(New AjaxControlToolkit.HTMLEditor.ToolbarButton.PreviewMode())
        End Sub
    End Class

End Namespace
```

You must add the class in Listing 2 to your App_Code folder so that the class will be compiled automatically. If the App_Code folder does not exist in your website then you can simply add the folder.

After you create a custom editor, you can add it to an ASP.NET page in the same way as you add the normal HTML Editor (see Listing 3).

Listing 3 - ShowCustomEditor.aspx

```
<%@ Page Language="VB" %>
<%@ Register namespace="MyControls" tagprefix="custom" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>Show Custom Editor</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    <asp:ScriptManager ID="ScriptManager1" runat="server" />

    <custom:CustomEditor ID="CustomEditor1"
        Width="450px"
        Height="200px"
        runat="server" />

    </div>
    </form>
</body>
</html>
```

Avoiding Cross-Site Scripting (XSS) Attacks

Whenever you accept input from a user, and redisplay that input on your website, you potentially open your website to Cross-Site Scripting (XSS) Attacks. In theory, a malicious hacker could submit JavaScript code that gets executed when the input is redisplayed. The JavaScript could be used to steal user passwords or other sensitive information.

Normally, you can defeat XSS attacks by HTML encoding whatever input you retrieve from a user before displaying it in a web page. However, HTML encoding the output of the HTML Editor would not only encode <script> tags, it would also encode all HTML tags. In other words, you would lose all of the formatting such as the

font type, font size, and background color.

If you are collecting sensitive information from your users -- such as passwords, credit-card numbers, and social security numbers - then you should not display un-encoded content that you retrieve from a user with the HTML Editor. You should use the HTML Editor only in situations in which you are not redisplaying the HTML content, or the HTML content is being submitted to your website by a trusted party.

Imagine, for example, that you are creating a blog application. In this situation, it makes sense to use the HTML Editor when composing blog posts. You are the only one who submits a blog post and, presumably, you can trust yourself not to submit malicious JavaScript. However, it does not make sense to use the HTML Editor when allowing anonymous users to post comments. You should be especially careful in situations in which users submit sensitive information such as passwords. Potentially, a malicious user could post a comment that contains the right JavaScript for stealing a password.

Summary

In this tutorial, you were provided with a brief overview of the HTML Editor control included in the AJAX Control Toolkit. You learned how to use the HTML Editor to accept rich content from a user and submit the content to the server. We also discussed how you can customize the toolbar buttons that are displayed by the HTML Editor. Finally, you learned how to avoid Cross-Site Scripting Attacks when using the HTML Editor to accept potentially malicious input.

PREVIOUS

ModalPopup

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the ModalPopup control, which lets you create the equivalent of a modal dialog box.

- Launching a Modal Popup Window from Server Code (C#)
- Using ModalPopup with a Repeater Control (C#)
- Handling Postbacks from a ModalPopup (C#)
- Positioning a ModalPopup (C#)
- Launching a Modal Popup Window from Server Code (VB)
- Using ModalPopup with a Repeater Control (VB)
- Handling Postbacks from a ModalPopup (VB)
- Positioning a ModalPopup (VB)

Launching a Modal Popup Window from Server Code (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. However some scenarios require that the opening of the modal popup is triggered on the server-side.

Overview

The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. However some scenarios require that the opening of the modal popup is triggered on the server-side.

Steps

First of all, an ASP.NET Button web control is required to demonstrate how the ModalPopup control works. Add such a button within the <form> element on a new page:

```
<asp:Button ID="ClientButton" runat="server" Text="Launch Modal Popup (Client)" />
```

Then, you need the markup for the popup you want to create. Define it as an `<asp:Panel>` control and make sure that it includes a Button control. The ModalPopup control offers the functionality to make such a button close the popup; otherwise there is no easy way to let it vanish.

```
<asp:Panel ID="ModalPanel" runat="server" Width="500px">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 <asp:Button ID="OKButton" runat="server" Text="Close" />
</asp:Panel>
```

Next add the ModalPopup control from the ASP.NET AJAX Toolkit to the page. Set properties for the button which loads the control, the button which makes it disappear, and the ID of the actual popup.

```
<ajaxToolkit:ModalPopupExtender ID="mpe" runat="server" TargetControlId="ClientButton"
 PopupControlID="ModalPanel" OkControlID="OKButton" />
```

As with all web pages based on ASP.NET AJAX; the Script Manager is required to load the necessary JavaScript libraries for the different target browsers:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Run the example in the browser. When you click on the button, the modal popup appears. In order to achieve the same effect using server-side code, a new button is required:

```
<asp:Button ID="ServerButton" runat="server" Text="Launch Modal Popup (Server)"
 OnClick="ServerButton_Click" />
```

As you can see, a click on the button generates a postback and executes the `ServerButton_Click()` method on the server. In this method, a JavaScript function called `launchModal()` is executed to be exact, the JavaScript function will be executed once the page has been loaded:

```
<script runat="server">
 protected void ServerButton_Click(object sender, EventArgs e)
 {
 ClientScript.RegisterStartupScript(this.GetType(), "key", "launchModal();", true);
 }
</script>
```

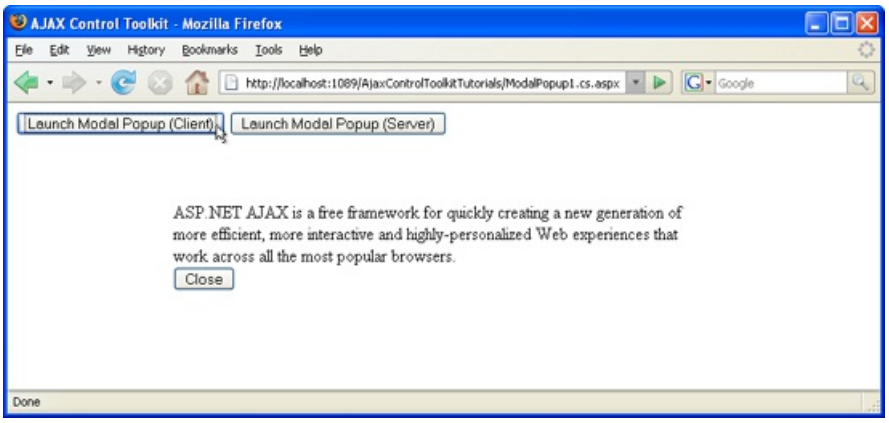
The job of `launchModal()` is to display the ModalPopup. The `launchModal()` function is executed once the complete HTML page has been loaded. At that moment, however, the ASP.NET AJAX framework has not been fully loaded yet. Therefore, the `launchModal()` function just sets a variable that the ModalPopup control must be shown later on:

```
<script type="text/javascript">
 var launch = false;
 function launchModal()
 {
 launch = true;
 }
```

The `pageLoad()` JavaScript function is a special function that gets executed once ASP.NET AJAX has been fully loaded. Therefore we add code to this function to show the ModalPopup control, but only if `launchModal()` has been called before:

```
function pageLoad()
 {
 if (launch)
 {
 $find("mpe").show();
 }
 }
</script>
```

The `$find()` function is looking for a named element on the page and expects the server-side ID as a parameter. Therefore, `$find("mpe")` returns the client representation of the ModalPopup control; its `show()` method lets the popup appear.



The modal popup appears when either of the buttons is clicked (Click to view full-size image)

Using ModalPopup with a Repeater Control (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. It is also possible to use this control within a repeater.

Overview

The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. It is also possible to use this control within a repeater.

Steps

First of all, a data source is required. This sample uses the AdventureWorks database and the Microsoft SQL Server 2005 Express Edition. The database is an optional part of a Visual Studio installation (including express edition) and is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. The AdventureWorks database is part of the SQL Server 2005 Samples and Sample Databases (download at https://www.microsoft.com/downloads/details.aspx?FamilyID=e719ecf7-9f46-4312-af89-6ad8702e4e6e&DisplayLang=en). The easiest way to set the database up is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en) and attach the `AdventureWorks.mdf` database file. For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database. In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, add a data source to the page. In order to use a limited amount of data, we only select the first five entries in the Vendor table of the AdventureWorks database. If you are using the Visual Studio assistant to create the data source, mind that a bug in the current version does not prefix the table name (`Vendor`) with `Purchasing`. The following markup shows the correct syntax:

```
<asp:SqlDataSource ID="sds1" runat="server" ConnectionString="Data
 Source=(local)\SQLEXPRESS;Initial Catalog=AdventureWorks;Integrated Security=True"
 ProviderName="System.Data.SqlClient" SelectCommand="SELECT TOP 5
 [VendorID], [Name] FROM [Purchasing].[Vendor]" />
```

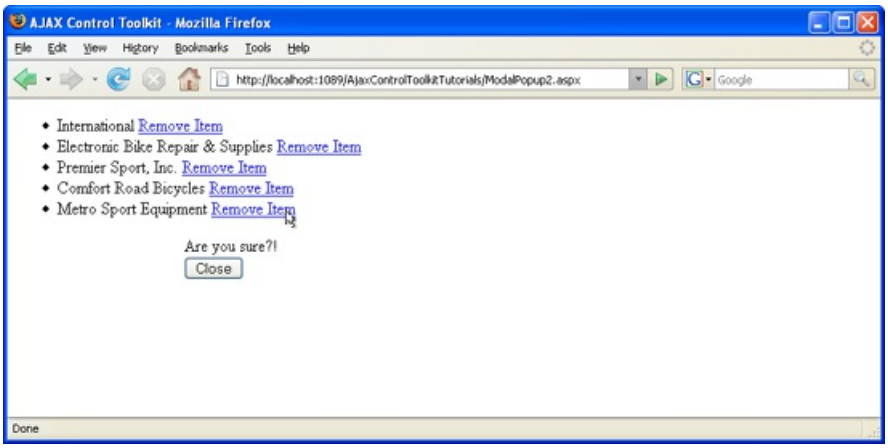
Next, add a panel which serves as the modal popup. It contains a `Button` control to close the popup again:

```
<asp:Panel ID="ModalPanel" runat="server" Width="500px">
 Are you sure?!<br />
 <asp:Button ID="OKButton" runat="server" Text="Close" />
</asp:Panel>
```

In order to make the popup work within the repeater, the `ModalPopupExtender` control must be put within the `<ItemTemplate>` section of the repeater. So the panel is outside the repeater, but the extender is inside. Here is the markup for the repeater:

```
<div>
 <ul>
 <asp:Repeater ID="rep1" DataSourceID="sds1" runat="server">
 <ItemTemplate>
 <li>
 <%#DataBinder.Eval(Container.DataItem, "Name")%>
 <asp:LinkButton ID="btn1" Text="Remove Item" runat="server" />
 <ajaxToolkit:ModalPopupExtender ID="mpe" runat="server"
 TargetControlId="btn1" PopupControlID="ModalPanel" OkControlID="OKButton" />
 </li>
 </ItemTemplate>
 </asp:Repeater>
 </ul>
</div>
```

Then, every item in the data source is displayed with a button next to it that triggers the modal popup.



The modal popup can be triggered for every data source entry (Click to view full-size image)

PREVIOUS NEXT

Handling Postbacks from a ModalPopup (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. Special care must be taken when a postback is created from within the popup.

Overview

The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. Special care must be taken when a postback is created from within the popup.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Next, add a panel which serves as the modal popup. There, the user can enter a name and an email address. A button is used to close the popup and save the information. Note that the `OnClick` attribute is set so that a postback occurs when this button is clicked:

```
<asp:Panel ID="ModalPanel" runat="server" Width="500px">
 Name: <asp:TextBox ID="tbName" runat="server" /><br/>
 Email: <asp:TextBox ID="tbEmail" runat="server" /><br/>
 <asp:Button ID="OKButton" runat="server" Text="Save"OnClick="SaveData" />
</asp:Panel>
```

The page itself consists of two labels for exactly the same information: name and email address. A button is used to trigger the modal popup:

```
<div>
 Contact Information:
 <asp:Label ID="lblName" runat="server" Text="AJAX Fanatic"/><br />
 <asp:Label ID="lblEmail" runat="server" Text="ajax@fanatic"/><br />
 <asp:Button ID="btn1" runat="server" Text="Edit"/>
</div>
```

In order to make the popup appear, add the `ModalPopupExtender` control. Set the `PopupControlID` attribute to the panel's ID and `TargetControlID` to the button's ID:

```
<ajaxToolkit:ModalPopupExtender ID="mpe" runat="server"
 TargetControlID="btn1" PopupControlID="ModalPanel" />
```

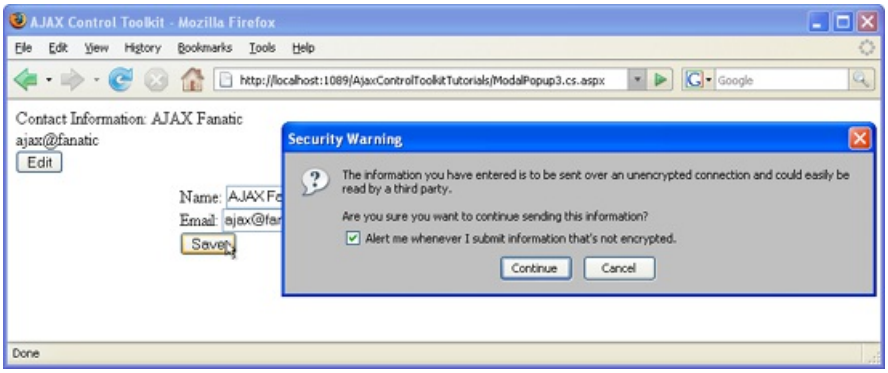
Now whenever the `Save` button within the modal popup is clicked, the server-side `SaveData()` method is executed. There, you could save the entered data in a data store. For the sake of simplicity, the new data is just

output in the label:

```
protected void SaveData(object sender, EventArgs e)
{
    lblName.Text = HttpUtility.HtmlEncode(tbName.Text);
    lblEmail.Text = HttpUtility.HtmlEncode(tbEmail.Text);
}
```

Also, the textbox controls within the modal popup should be filled with the current name and email. However this is only necessary when no postback occurs. If there is a postback, the ASP.NET viewstate feature will automatically fill the textboxes with the appropriate values.

```
void Page_Load()
{
    if (!Page.IsPostBack)
    {
        tbName.Text = lblName.Text;
        tbEmail.Text = lblEmail.Text;
    }
}
```



The modal popup causes a postback (Click to view full-size image)

PREVIOUS NEXT

Positioning a ModalPopup (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. However the control does not offer a built-in functionality to position the popup.

Overview

The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. However the control does not offer a built-in functionality to position the popup.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager`. control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Next, add a panel which serves as the modal popup. A button is used to close the popup:

```
<asp:Panel ID="ModalPanel" runat="server" Width="500px">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 <asp:Button ID="OKButton" runat="server" Text="Close" />
</asp:Panel>
```

Whenever the popup is shown, it shall be positioned at a certain place in the page. For this task, a client-side JavaScript function is created. It first tries to access the panel. If it succeeds, the panel's position is set using CSS and JavaScript (change the position of the popup at will). However the `ModalPopupExtender` control also tries to position the popup. Therefore, the JavaScript code repeatedly positions the popup, every tenth of a second.

```
<script type="text/javascript">
 var id = null;
 function movePanel()
 {
 var pnl = $get("ModalPanel");
 if (pnl != null)
 {
 pnl.style.left = "50px";
 pnl.style.top = "50px";
 id = setTimeout("movePanel();", 100);
 }
 }
```

As you can see, the return value of the `setTimeout()` JavaScript method is saved in a global variable. This allows to stop the repeated positioning of the popup on demand, using the `clearTimeout()` method:

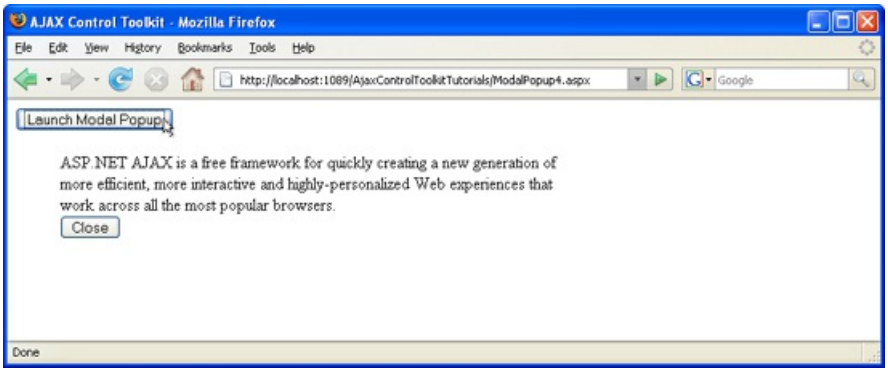
```
function stopMoving()
 {
 clearTimeout(id);
 }
</script>
```

Now all that is left to do is to make the browser call these functions whenever appropriate. The `movePanel()` JavaScript function must be called when the button is clicked that triggers the panel:

```
<div>
 <asp:Button ID="btn1" runat="server" Text="Launch Modal Popup"
 OnClientClick="movePanel();" />
</div>
```

And the `stopMoving()` function comes into play when the popup is closed this can be triggered in the `ModalPopupExtender` control:

```
<ajaxToolkit:ModalPopupExtender ID="mpe" runat="server"
 TargetControlId="btn1" PopupControlID="ModalPanel" OkControlID="OKButton"
 OnOkScript="stopMoving();" />
```



The modal popup appears at the designated position (Click to view full-size image)

PREVIOUS NEXT

Launching a Modal Popup Window from Server Code (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. However some scenarios require that the opening of the modal popup is triggered on the server-side.

Overview

The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. However some scenarios require that the opening of the modal popup is triggered on the server-side.

Steps

First of all, an ASP.NET Button web control is required to demonstrate how the ModalPopup control works. Add such a button within the <form> element on a new page:

```
<asp:Button ID="ClientButton" runat="server" Text="Launch
 Modal Popup (Client)" />
```

Then, you need the markup for the popup you want to create. Define it as an `<asp:Panel>` control and make sure that it includes a Button control. The ModalPopup control offers the functionality to make such a button close the popup; otherwise there is no easy way to let it vanish.

```
<asp:Panel ID="ModalPanel" runat="server" Width="500px">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more efficient,
 more interactive and highly-personalized Web experiences that work across all the
 most popular browsers.<br />
 <asp:Button ID="OKButton" runat="server" Text="Close" />
</asp:Panel>
```

Next add the ModalPopup control from the ASP.NET AJAX Toolkit to the page. Set properties for the button which loads the control, the button which makes it disappear, and the ID of the actual popup.

```
<ajaxToolkit:ModalPopupExtender ID="mpe" runat="server"
 TargetControlId="ClientButton" PopupControlID="ModalPanel"
 OkControlID="OKButton" />
```

As with all web pages based on ASP.NET AJAX; the Script Manager is required to load the necessary JavaScript libraries for the different target browsers:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Run the example in the browser. When you click on the button, the modal popup appears. In order to achieve the

same effect using server-side code, a new button is required:

```
<asp:Button ID="ServerButton" runat="server" Text="Launch Modal Popup (Server)"
 OnClick="ServerButton_Click" />
```

As you can see, a click on the button generates a postback and executes the `ServerButton_Click()` method on the server. In this method, a JavaScript function called `launchModal()` is executed to be exact, the JavaScript function will be executed once the page has been loaded:

```
<script runat="server">
 Sub ServerButton_Click(ByVal sender As Object, ByVal e As EventArgs)
 ClientScript.RegisterStartupScript(Me.GetType(), "key", "launchModal();", True)
 End Sub
</script>
```

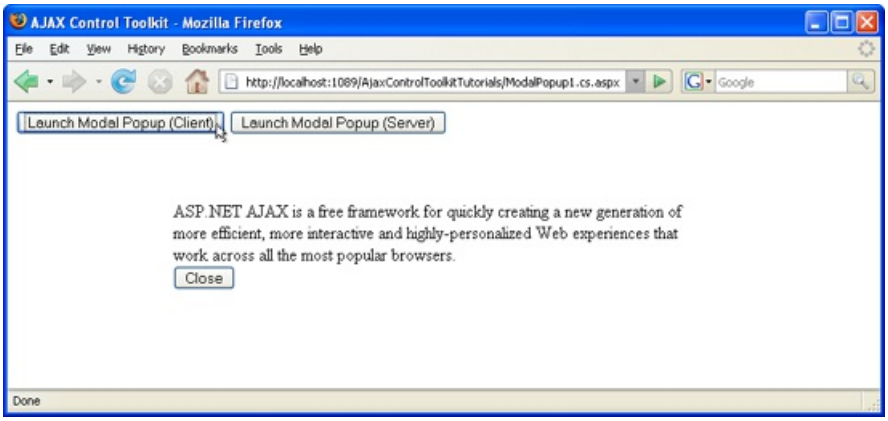
The job of `launchModal()` is to display the ModalPopup. The `launchModal()` function is executed once the complete HTML page has been loaded. At that moment, however, the ASP.NET AJAX framework has not been fully loaded yet. Therefore, the `launchModal()` function just sets a variable that the ModalPopup control must be shown later on:

```
<script type="text/javascript">
 var launch = false;
 function launchModal()
 {
 launch = true;
 }
```

The `pageLoad()` JavaScript function is a special function that gets executed once ASP.NET AJAX has been fully loaded. Therefore we add code to this function to show the ModalPopup control, but only if `launchModal()` has been called before:

```
function pageLoad()
 {
 if (launch)
 {
 $find("mpe").show();
 }
 }
</script>
```

The `$find()` function is looking for a named element on the page and expects the server-side ID as a parameter. Therefore, `$find("mpe")` returns the client representation of the ModalPopup control; its `show()` method lets the popup appear.



The modal popup appears when either of the buttons is clicked (Click to view full-size image)

Using ModalPopup with a Repeater Control (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. It is also possible to use this control within a repeater.

Overview

The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. It is also possible to use this control within a repeater.

Steps

First of all, a data source is required. This sample uses the AdventureWorks database and the Microsoft SQL Server 2005 Express Edition. The database is an optional part of a Visual Studio installation (including express edition) and is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. The AdventureWorks database is part of the SQL Server 2005 Samples and Sample Databases (download at https://www.microsoft.com/downloads/details.aspx?FamilyID=e719ecf7-9f46-4312-af89-6ad8702e4e6e&DisplayLang=en). The easiest way to set the database up is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en) and attach the `AdventureWorks.mdf` database file. For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database. In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, add a data source to the page. In order to use a limited amount of data, we only select the first five entries in the Vendor table of the AdventureWorks database. If you are using the Visual Studio assistant to create the data source, mind that a bug in the current version does not prefix the table name (`Vendor`) with `Purchasing`. The following markup shows the correct syntax:

```
<asp:SqlDataSource ID="sds1" runat="server" ConnectionString="Data
 Source=(local)\SQLEXPRESS;Initial Catalog=AdventureWorks;Integrated Security=True"
 ProviderName="System.Data.SqlClient" SelectCommand="SELECT TOP 5
 [VendorID], [Name] FROM [Purchasing].[Vendor]" />
```

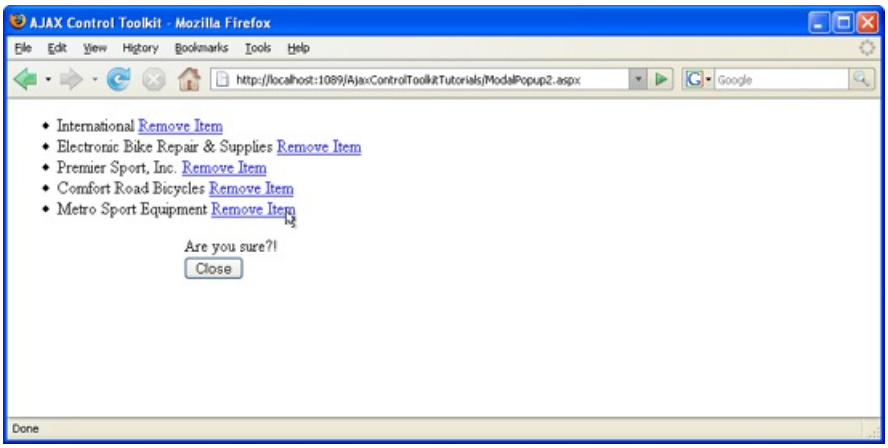
Next, add a panel which serves as the modal popup. It contains a `Button` control to close the popup again:

```
<asp:Panel ID="ModalPanel" runat="server" Width="500px">
 Are you sure?!<br />
 <asp:Button ID="OKButton" runat="server" Text="Close" />
</asp:Panel>
```

In order to make the popup work within the repeater, the `ModalPopupExtender` control must be put within the `<ItemTemplate>` section of the repeater. So the panel is outside the repeater, but the extender is inside. Here is the markup for the repeater:

```
<div>
 <ul>
 <asp:Repeater ID="rep1" DataSourceID="sds1" runat="server">
 <ItemTemplate>
 <li>
 <%#DataBinder.Eval(Container.DataItem, "Name")%>
 <asp:LinkButton ID="btn1" Text="Remove Item" runat="server" />
 <ajaxToolkit:ModalPopupExtender ID="mpe" runat="server"
 TargetControlId="btn1" PopupControlID="ModalPanel" OkControlID="OKButton" />
 </li>
 </ItemTemplate>
 </asp:Repeater>
 </ul>
</div>
```

Then, every item in the data source is displayed with a button next to it that triggers the modal popup.



The modal popup can be triggered for every data source entry (Click to view full-size image)

PREVIOUS NEXT

Handling Postbacks from a ModalPopup (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. Special care must be taken when a postback is created from within the popup.

Overview

The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. Special care must be taken when a postback is created from within the popup.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Next, add a panel which serves as the modal popup. There, the user can enter a name and an email address. A button is used to close the popup and save the information. Note that the `OnClick` attribute is set so that a postback occurs when this button is clicked:

```
<asp:Panel ID="ModalPanel" runat="server" Width="500px">
 Name: <asp:TextBox ID="tbName" runat="server" /><br/>
 Email: <asp:TextBox ID="tbEmail" runat="server" /><br/>
 <asp:Button ID="OKButton" runat="server" Text="Save"OnClick="SaveData" />
</asp:Panel>
```

The page itself consists of two labels for exactly the same information: name and email address. A button is used to trigger the modal popup:

```
<div>
 Contact Information:
 <asp:Label ID="lblName" runat="server" Text="AJAX Fanatic"/><br />
 <asp:Label ID="lblEmail" runat="server" Text="ajax@fanatic"/><br />
 <asp:Button ID="btn1" runat="server" Text="Edit"/>
</div>
```

In order to make the popup appear, add the `ModalPopupExtender` control. Set the `PopupControlID` attribute to the panel's ID and `TargetControlID` to the button's ID:

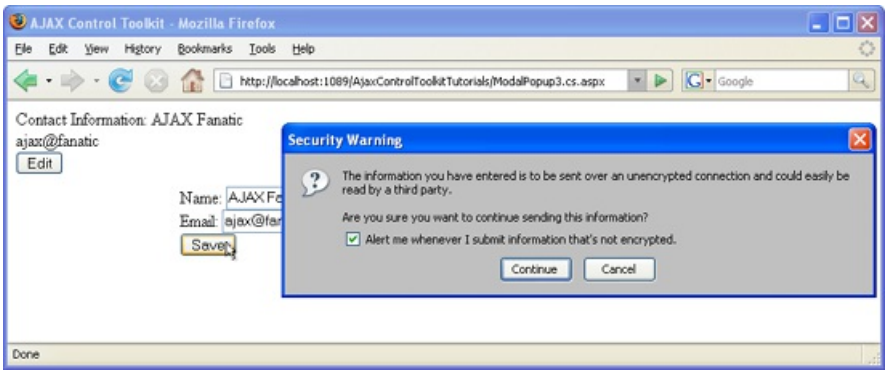
```
<ajaxToolkit:ModalPopupExtender ID="mpe" runat="server" TargetControlID="btn1" PopupControlID="ModalPanel" />
```

Now whenever the `Save` button within the modal popup is clicked, the server-side `SaveData()` method is executed. There, you could save the entered data in a data store. For the sake of simplicity, the new data is just output in the label:

```
Protected Sub SaveData(ByVal sender As Object, ByVal e As EventArgs)
 lblName.Text = HttpUtility.HtmlEncode(tbName.Text)
 lblEmail.Text = HttpUtility.HtmlEncode(tbEmail.Text)
End Sub
```

Also, the textbox controls within the modal popup should be filled with the current name and email. However this is only necessary when no postback occurs. If there is a postback, the ASP.NET viewstate feature will automatically fill the textboxes with the appropriate values.

```
Sub Page_Load()
 If Not Page.IsPostBack Then
 tbName.Text = lblName.Text
 tbEmail.Text = lblEmail.Text
 End If
End Sub
```



The modal popup causes a postback (Click to view full-size image)

PREVIOUS NEXT

Positioning a ModalPopup (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. However the control does not offer a built-in functionality to position the popup.

Overview

The ModalPopup control in the AJAX Control Toolkit offers a simple way to create a modal popup using client-side means. However the control does not offer a built-in functionality to position the popup.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager`. control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Next, add a panel which serves as the modal popup. A button is used to close the popup:

```
<asp:Panel ID="ModalPanel" runat="server" Width="500px">
 ASP.NET AJAX is a free framework for quickly creating a new generation of more
 efficient, more interactive and highly-personalized Web experiences that work
 across all the most popular browsers.<br />
 <asp:Button ID="OKButton" runat="server" Text="Close" />
</asp:Panel>
```

Whenever the popup is shown, it shall be positioned at a certain place in the page. For this task, a client-side JavaScript function is created. It first tries to access the panel. If it succeeds, the panel's position is set using CSS and JavaScript (change the position of the popup at will). However the `ModalPopupExtender` control also tries to position the popup. Therefore, the JavaScript code repeatedly positions the popup, every tenth of a second.

```
<script type="text/javascript">
 var id = null;
 function movePanel()
 {
 var pnl = $get("ModalPanel");
 if (pnl != null)
 {
 pnl.style.left = "50px";
 pnl.style.top = "50px";
 id = setTimeout("movePanel();", 100);
 }
 }
```

As you can see, the return value of the `setTimeout()` JavaScript method is saved in a global variable. This allows to stop the repeated positioning of the popup on demand, using the `clearTimeout()` method:

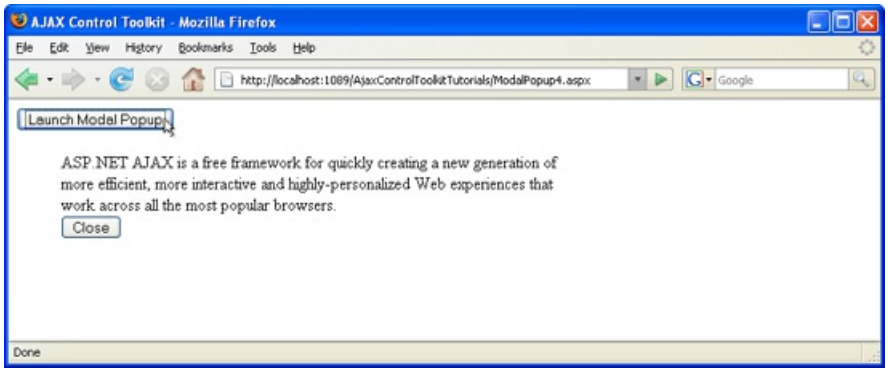
```
function stopMoving()
 {
 clearTimeout(id);
 }
</script>
```

Now all that is left to do is to make the browser call these functions whenever appropriate. The `movePanel()` JavaScript function must be called when the button is clicked that triggers the panel:

```
<div>
 <asp:Button ID="btn1" runat="server" Text="Launch Modal Popup"
 OnClientClick="movePanel();" />
</div>
```

And the `stopMoving()` function comes into play when the popup is closed this can be triggered in the `ModalPopupExtender` control:

```
<ajaxToolkit:ModalPopupExtender ID="mpe" runat="server"
 TargetControlId="btn1" PopupControlID="ModalPanel" OkControlID="OKButton"
 OnOkScript="stopMoving();" />
```



The modal popup appears at the designated position (Click to view full-size image)

PREVIOUS

MutuallyExclusiveCheckbox

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the MutuallyExclusiveCheckbox control to create check boxes that work like radio buttons (mutually exclusive) but that can be completely cleared.

- Creating Mutually Exclusive Checkboxes (C#)
- Creating Mutually Exclusive Checkboxes (VB)

Creating Mutually Exclusive Checkboxes (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> When only one of a set of options may be selected, radio buttons are usually used. There is a drawback, though: Once one radio button in a group is selected, it is not possible to uncheck all radio buttons. Check boxes can be unchecked at any time, however are not mutually exclusive. This tutorial provides the best of both approaches: check boxes that are mutually exclusive.

Overview

When only one of a set of options may be selected, radio buttons are usually used. There is a drawback, though: Once one radio button in a group is selected, it is not possible to uncheck all radio buttons. Check boxes can be unchecked at any time, however are not mutually exclusive. This tutorial provides the best of both approaches: check boxes that are mutually exclusive.

Steps

The ASP.NET AJAX Control Toolkit contains the MutuallyExclusiveCheckBox extender. This enables programmers to assign any checkbox to a group name (`Key` attribute). From all check boxes within the same group, only one may be selected at one time.

Let's start with putting two check boxes on a new ASP.NET page. There can be more, but two of them suffice to demonstrate the principle:

```
<asp:CheckBox ID="cbYes" runat="server" />Yes
<asp:CheckBox ID="cbNo" runat="server" />No
```

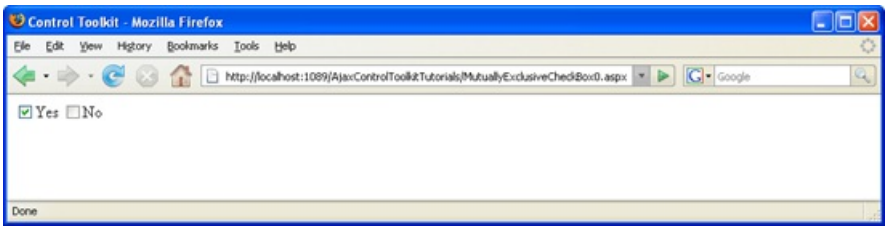
For both checkboxes, a MutuallyExclusiveCheckBoxExtender control must be put on the page. Both Key attributes need to have the same value, just as the value attributes of HTML radio button elements must be identical to denote the group they belong to. The TargetControlID property of the extender points to the ID of the check box.

```
<ajaxToolkit:MutuallyExclusiveCheckBoxExtender ID="mecbe1" runat="server"
 TargetControlID="cbYes" Key="YesNo" />
<ajaxToolkit:MutuallyExclusiveCheckBoxExtender ID="mecbe2" runat="server"
 TargetControlID="cbNo" Key="YesNo" />
```

Finally, include the ASP.NET AJAX `ScriptManager` which is required by all elements of the ASP.NET AJAX Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Save and run the page: You can check and uncheck both check boxes, however at no time can both check boxes be checked.



Only one checkbox can be checked at a time (Click to view full-size image)

NEXT

Creating Mutually Exclusive Checkboxes (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

When only one of a set of options may be selected, radio buttons are usually used. There is a drawback, though: Once one radio button in a group is selected, it is not possible to uncheck all radio buttons. Check boxes can be unchecked at any time, however are not mutually exclusive. This tutorial provides the best of both approaches: check boxes that are mutually exclusive.

Overview

When only one of a set of options may be selected, radio buttons are usually used. There is a drawback, though: Once one radio button in a group is selected, it is not possible to uncheck all radio buttons. Check boxes can be unchecked at any time, however are not mutually exclusive. This tutorial provides the best of both approaches: check boxes that are mutually exclusive.

Steps

The ASP.NET AJAX Control Toolkit contains the MutuallyExclusiveCheckBox extender. This enables programmers to assign any checkbox to a group name (`Key` attribute). From all check boxes within the same group, only one may be selected at one time.

Let's start with putting two check boxes on a new ASP.NET page. There can be more, but two of them suffice to demonstrate the principle:

```
<asp:CheckBox ID="cbYes" runat="server" />Yes
<asp:CheckBox ID="cbNo" runat="server" />No
```

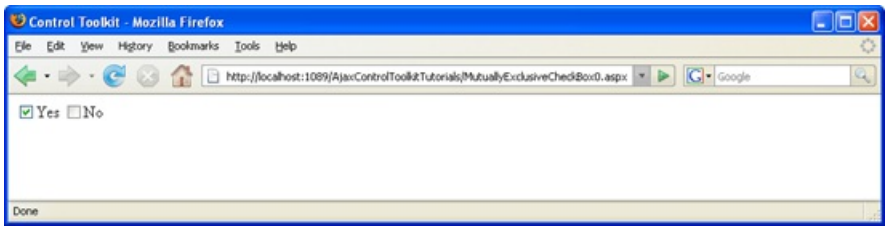
For both checkboxes, a MutuallyExclusiveCheckBoxExtender control must be put on the page. Both Key attributes need to have the same value, just as the value attributes of HTML radio button elements must be identical to denote the group they belong to. The TargetControlID property of the extender points to the ID of the check box.

```
<ajaxToolkit:MutuallyExclusiveCheckBoxExtender ID="mecbe1" runat="server"
 TargetControlID="cbYes" Key="YesNo" />
<ajaxToolkit:MutuallyExclusiveCheckBoxExtender ID="mecbe2" runat="server"
 TargetControlID="cbNo" Key="YesNo" />
```

Finally, include the ASP.NET AJAX `ScriptManager` which is required by all elements of the ASP.NET AJAX Control Toolkit:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Save and run the page: You can check and uncheck both check boxes, however at no time can both check boxes be checked.



Only one checkbox can be checked at a time (Click to view full-size image)

PREVIOUS

NoBot

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the NoBot control, which provides an automated way to prevent bots from posting to a site.

- Fighting Bots (C#)
- Fighting Bots (VB)

Fighting Bots (C#)

2/20/2020 • 3 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> Automated bots plaster weblogs and other websites with spam, submitting comment forms without any user interaction. The NoBot control in the ASP.NET AJAX Control Toolkit can help fight those bots.

Overview

Automated bots plaster weblogs and other websites with spam, submitting comment forms without any user interaction. The NoBot control in the ASP.NET AJAX Control Toolkit can help fight those bots.

Steps

One common approach to defeat bots is to use CAPTCHAs Completely Automated Public Turing test to tell Computers and Humans Apart. A Turing test was originally a test where someone needed to decide whether a communication partner is a human or a machine. In the web, a CAPTCHA usually consists of an image with some distorted letters on it. The idea is that only a human can read the letters on the image, whereas OCR algorithms will fail.

There are several advantages and disadvantages to this approach, but a discussion of this is beyond the scope of this tutorial. There is however a control in the ASP.NET AJAX Control Toolkit which provides a similar approach: `NoBot`. It is easier to overcome than a CAPTCHA, but is very easy to use and fares extremely well on websites like blogs where it is considered a success if most spam attempts are defeated, which the `NoBot` control can do.

`NoBot` intercepts the postback of the current ASP.NET web form if at least one of these conditions is met:

- The browser fails to solve a JavaScript puzzle (for instance when JavaScript is deactivated)
- The user submitted the form to fast
- The client IP address submitted the form too often in a certain period of time.

In order to check for these conditions, the `NoBot` control requires these attributes (all of them optional):

- `ResponseMinimumDelaySeconds` minimum amount of seconds between postbacks
- `CutoffWindowSeconds` length of time interval in which postbacks from one IP are measures
- `CutoffMaximumInstances` maximum amount of seconds per time interval

The following markup demands that at least two seconds elapse between postbacks and that there are only five postbacks or less within a 30 seconds interval:

```
<ajaxToolkit:NoBot ID="nb" runat="server" CutoffMaximumInstances="5"
 CutoffWindowSeconds="30" ResponseMinimumDelaySeconds="2" />
```

Then as usual make sure to include the `ScriptManager` in the page so that the ASP.NET AJAX library is loaded and the Control Toolkit can be used:

```
<asp:ScriptManager ID="asm" runat="server" />
```

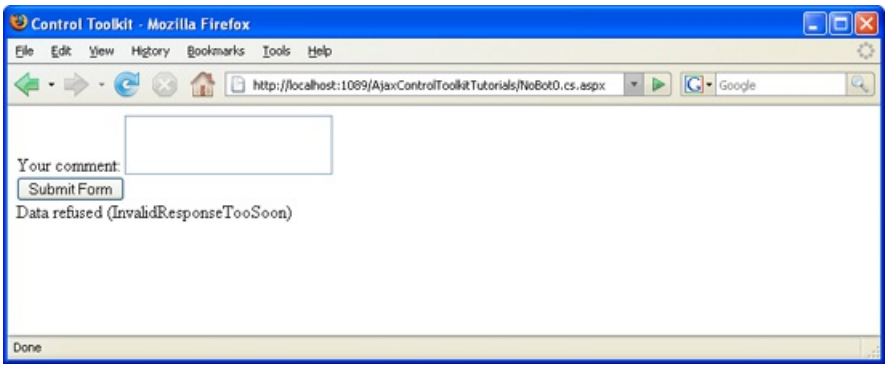
Since most of the checks `NoBot` is doing occur on the server side, you need to check the result of these validations. This can be done by calling `NoBot`'s `IsValid()` method. It has one argument (as an `out` parameter/ `ByRef` parameter) which is of type `NoBotState`. Its string representation contains the reason when the check fails and `Valid` otherwise. The following code outputs a message according to `NoBot`'s result:

```
<script runat="server">
 void Page_Load()
 {
 if (Page.IsPostBack)
 {
 NoBotState state;
 if (!nb.IsValid(out state))
 {
 Label1.Text = "Data refused ("
 + HttpUtility.HtmlEncode(state.ToString()) + ")";
 }
 else
 {
 Label1.Text = "Data entered.";
 }
 }
 }
</script>
```

Finally, you need a form to submit and a label element to output the message, and you are done!

```
Your comment:
<asp:TextBox ID="TextBox1" runat="server" TextMode="MultiLine"/><br />
<input type="submit" id="Submit1" runat="server" value="Submit Form" /><br />
<asp:Label ID="Label1" runat="server" />
```

When you run this script and deactivate JavaScript or submit the form within the first two seconds or submit the form seven times within thirty seconds, you will get an error message. However use this control wisely, since only about 90-95% of users have JavaScript activated, therefore 5-10% of users will fail `NoBot`'s test.



This error message could have been caused by a bot (Click to view full-size image)

NEXT

Fighting Bots (VB)

2/20/2020 • 3 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> Automated bots plaster weblogs and other websites with spam, submitting comment forms without any user interaction. The NoBot control in the ASP.NET AJAX Control Toolkit can help fight those bots.

Overview

Automated bots plaster weblogs and other websites with spam, submitting comment forms without any user interaction. The NoBot control in the ASP.NET AJAX Control Toolkit can help fight those bots.

Steps

One common approach to defeat bots is to use CAPTCHAs Completely Automated Public Turing test to tell Computers and Humans Apart. A Turing test was originally a test where someone needed to decide whether a communication partner is a human or a machine. In the web, a CAPTCHA usually consists of an image with some distorted letters on it. The idea is that only a human can read the letters on the image, whereas OCR algorithms will fail.

There are several advantages and disadvantages to this approach, but a discussion of this is beyond the scope of this tutorial. There is however a control in the ASP.NET AJAX Control Toolkit which provides a similar approach: `NoBot`. It is easier to overcome than a CAPTCHA, but is very easy to use and fares extremely well on websites like blogs where it is considered a success if most spam attempts are defeated, which the `NoBot` control can do.

`NoBot` intercepts the postback of the current ASP.NET web form if at least one of these conditions is met:

- The browser fails to solve a JavaScript puzzle (for instance when JavaScript is deactivated)
- The user submitted the form to fast
- The client IP address submitted the form too often in a certain period of time.

In order to check for these conditions, the `NoBot` control requires these attributes (all of them optional):

- `ResponseMinimumDelaySeconds` minimum amount of seconds between postbacks
- `CutoffWindowSeconds` length of time interval in which postbacks from one IP are measures
- `CutoffMaximumInstances` maximum amount of seconds per time interval

The following markup demands that at least two seconds elapse between postbacks and that there are only five postbacks or less within a 30 seconds interval:

```
<ajaxToolkit:NoBot ID="nb" runat="server" CutoffMaximumInstances="5"
 CutoffWindowSeconds="30" ResponseMinimumDelaySeconds="2" />
```

Then as usual make sure to include the `ScriptManager` in the page so that the ASP.NET AJAX library is loaded and the Control Toolkit can be used:

```
<asp:ScriptManager ID="asm" runat="server" />
```

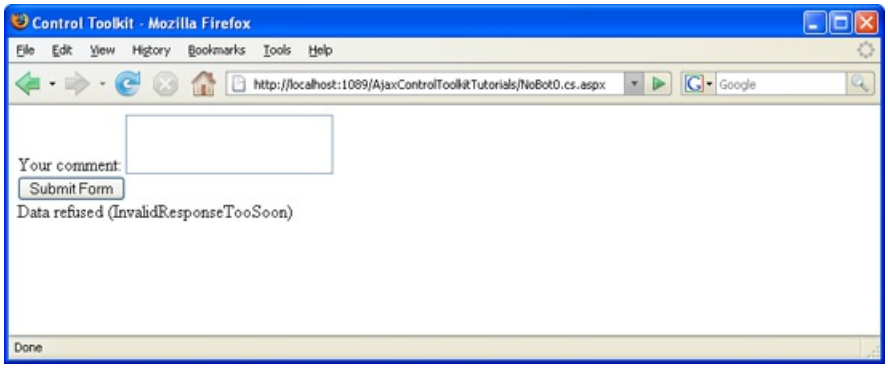
Since most of the checks `NoBot` is doing occur on the server side, you need to check the result of these validations. This can be done by calling `NoBot`'s `IsValid()` method. It has one argument (as an `out` parameter/ `ByRef` parameter) which is of type `NoBotState`. Its string representation contains the reason when the check fails and `Valid` otherwise. The following code outputs a message according to `NoBot`'s result:

```
<script runat="server">
 Sub Page_Load()
 If Page.IsPostBack Then
 Dim state As NoBotState
 If Not nb.IsValid(state) Then
 Label1.Text = "Data refused (" + HttpUtility.HtmlEncode(state.ToString()) + ")"
 Else
 Label1.Text = "Data entered."
 End If
 End If
 End Sub
</script>
```

Finally, you need a form to submit and a label element to output the message, and you are done!

```
Your comment:
<asp:TextBox ID="TextBox1" runat="server" TextMode="MultiLine"/><br />
<input type="submit" id="Submit1" runat="server" value="Submit Form" /><br />
<asp:Label ID="Label1" runat="server" />
```

When you run this script and deactivate JavaScript or submit the form within the first two seconds or submit the form seven times within thirty seconds, you will get an error message. However use this control wisely, since only about 90-95% of users have JavaScript activated, therefore 5-10% of users will fail `NoBot`'s test.



This error message could have been caused by a bot (Click to view full-size image)

PREVIOUS

NumericUpDown

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the NumericUpDown control, which lets users select a value using up and down arrows.

- Creating a Numeric Up/Down Control with a Web Service Backend (C#)
- Creating a Numeric Up/Down Control with a Web Service Backend (VB)

Creating a Numeric Up/Down Control with a Web Service Backend (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> Instead of letting a user type a value into a check box, a numeric up/down control (that exists on Windows and other operating systems) could prove as more comfortable. By default, the NumericUpDown control always increases or decreases a value by 1, but a web service proves more flexibility.

Overview

Instead of letting a user type a value into a check box, a numeric up/down control (that exists on Windows and other operating systems) could prove as more comfortable. By default, the `NumericUpDown` control always increases or decreases a value by 1, but a web service proves more flexibility.

Steps

The ASP.NET AJAX Control Toolkit contains the `NumericUpDown` extender which automatically adds two buttons to a text box: One for increasing its value, one for decreasing it. However the control also supports a web service call (or page method call). Whenever the up or down button is clicked, the JavaScript code connects to the web server and executes a method there. The method signature is the following one:

```
public int MethodName(int current, string tag) {}
```

The `current` argument is the current value in the text box; the `tag` attribute is additional context data that can be set as a property of the `NumericUpDown` extender (but is not required).

For this sample, the numeric up/down control shall only allow values that are powers of two: 1, 2, 4, 8, 16, 32, 64, and so on. Therefore, the method executed when the user wants to increase the value must double the old value; the other method must divide value by two. So here is the complete web service:

```
<%@ WebService Language="C#" Class="NumericUpDown1" %>
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
[System.Web.Script.Services.ScriptService]
public class NumericUpDown1 : System.Web.Services.WebService
{
 [WebMethod]
 public int Up(int current, string tag)
 {
 if (current <= 536870912)
 {
 return current * 2;
 }
 else
 {
 return current;
 }
 }
 [WebMethod]
 public int Down(int current, string tag)
 {
 if (current >= 2)
 {
 return (int)(current / 2);
 }
 else
 {
 return current;
 };
 }
}
```

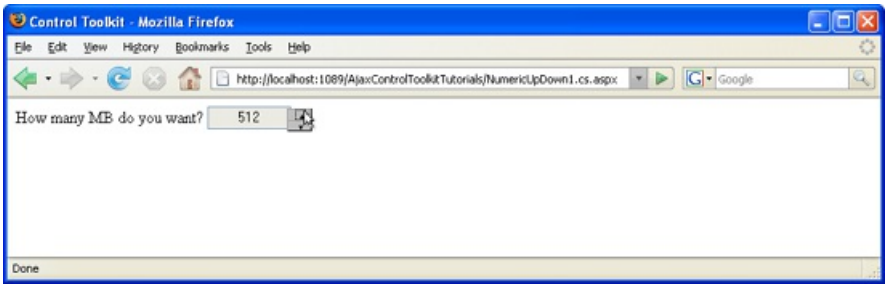
Finally, create a new ASP.NET page. As usual, you need a `ScriptManager` control, a `TextBox` control and a `NumericUpDownExtender` control. For the latter, you have to provide the web service information:

- `ServiceDownMethod` name of the down web method or page method
- `ServiceDownPath` path to the web service with the down service method; omit if you are using a page method
- `ServiceUpMethod` name of the up web method or page method
- `ServiceUpPath` path to the web service with the up service method; omit if you are using a page method

Here is the complete markup for the page:

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
 <title>Control Toolkit</title>
</head>
<body>
 <form id="form1" runat="server">
 <asp:ScriptManager ID="asm" runat="server" />
 <div>
 How many MB do you want? <asp:TextBox ID="TextBox1" Text="32" runat="server" />
 <ajaxToolkit:NumericUpDownExtender ID="nud" runat="server"
 TargetControlID="TextBox1" Width="100"
 ServiceUpPath="NumericUpDown1.cs.asmx"
 ServiceDownPath="NumericUpDown1.cs.asmx"
 ServiceUpMethod="Up" ServiceDownMethod="Down" />
 </div>
 </form>
</body>
</html>
```

If you run the page, notice how the value in the text box always doubles when you click on the upper button, and is halved when you click on the lower button.



Only numbers that are a power of 2 appear (Click to view full-size image)

NEXT

Creating a Numeric Up/Down Control with a Web Service Backend (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> Instead of letting a user type a value into a check box, a numeric up/down control (that exists on Windows and other operating systems) could prove as more comfortable. By default, the NumericUpDown control always increases or decreases a value by 1, but a web service proves more flexibility.

Overview

Instead of letting a user type a value into a check box, a numeric up/down control (that exists on Windows and other operating systems) could prove as more comfortable. By default, the `NumericUpDown` control always increases or decreases a value by 1, but a web service proves more flexibility.

Steps

The ASP.NET AJAX Control Toolkit contains the `NumericUpDown` extender which automatically adds two buttons to a text box: One for increasing its value, one for decreasing it. However the control also supports a web service call (or page method call). Whenever the up or down button is clicked, the JavaScript code connects to the web server and executes a method there. The method signature is the following one:

```
Function MethodName(ByVal current As Integer, ByVal tag As String) As Integer
```

The `current` argument is the current value in the text box; the `tag` attribute is additional context data that can be set as a property of the `NumericUpDown` extender (but is not required).

For this sample, the numeric up/down control shall only allow values that are powers of two: 1, 2, 4, 8, 16, 32, 64, and so on. Therefore, the method executed when the user wants to increase the value must double the old value; the other method must divide value by two. So here is the complete web service:

```
<%@ WebService Language="VB" Class="NumericUpDown1" %>
Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols
<System.Web.Script.Services.ScriptService()> _
Public Class NumericUpDown1
 Inherits System.Web.Services.WebService
 <WebMethod()> _
 Function Up(ByVal current As Integer, ByVal tag As String) As Integer
 If current <= 536870912 Then
 Return current * 2
 Else
 Return current
 End If
 End Function
 <WebMethod()> _
 Function Down(ByVal current As Integer, ByVal tag As String) As Integer
 If current >= 2 Then
 Return CInt(current / 2)
 Else
 Return current
 End If
 End Function
End Class
```

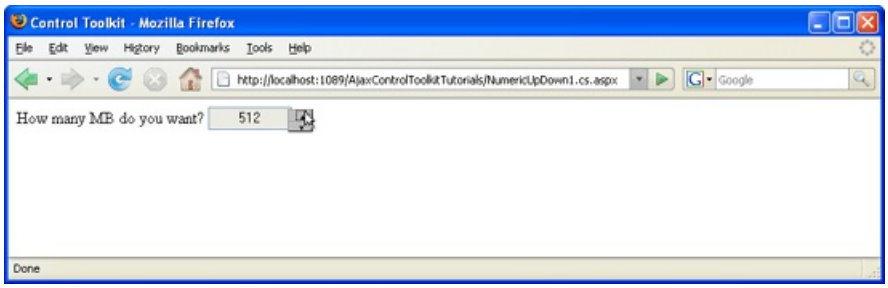
Finally, create a new ASP.NET page. As usual, you need a `ScriptManager` control, a `TextBox` control and a `NumericUpDownExtender` control. For the latter, you have to provide the web service information:

- `ServiceDownMethod` name of the down web method or page method
- `ServiceDownPath` path to the web service with the down service method; omit if you are using a page method
- `ServiceUpMethod` name of the up web method or page method
- `ServiceUpPath` path to the web service with the up service method; omit if you are using a page method

Here is the complete markup for the page:

```
<%@ Page Language="VB" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
 <title>Control Toolkit</title>
</head>
<body>
 <form id="form1" runat="server">
 <asp:ScriptManager ID="asm" runat="server" />
 <div>
 How many MB do you want? <asp:TextBox ID="TextBox1" Text="32" runat="server" />
 <ajaxToolkit:NumericUpDownExtender ID="nud" runat="server"
 TargetControlID="TextBox1" Width="100"
 ServiceUpPath="NumericUpDown1.vb.asmx" ServiceDownPath="NumericUpDown1.vb.asmx"
 ServiceUpMethod="Up" ServiceDownMethod="Down" />
 </div>
 </form>
</body>
</html>
```

If you run the page, notice how the value in the text box always doubles when you click on the upper button, and is halved when you click on the lower button.



Only numbers that are a power of 2 appear (Click to view full-size image)

PREVIOUS

PasswordStrength

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to PasswordStrength control to check how strong a password is as a user creates it.

- Testing the Strength of a Password (C#)
- Testing the Strength of a Password (VB)

Testing the Strength of a Password (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> Passwords are required almost anywhere, so that lazy users tend to choose simple passwords which are easy to break. The PasswordStrength control in the ASP.NET AJAX Control Toolkit can check how good a password is.

Overview

Passwords are required almost anywhere, so that lazy users tend to choose simple passwords which are easy to break. The `PasswordStrength` control in the ASP.NET AJAX Control Toolkit can check how good a password is.

Steps

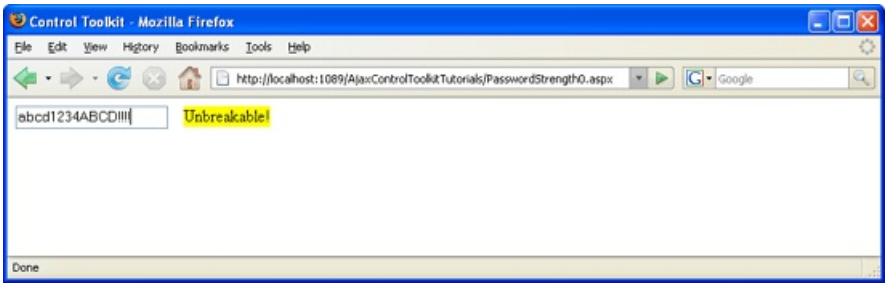
The `PasswordStrength` control extends a text box and checks whether the password in it is good enough. It offers a wealth of options via attributes; here are just some of them:

- `MinimumNumericCharacters` minimum number of numeric characters required in the password
- `MinimumSymbolCharacters` minimum number of symbol characters (not letters and digits) required in the password
- `PreferredPasswordLength` minimum length of the password
- `RequiresUpperAndLowerCaseCharacters` whether the password needs to use both uppercase and lowercase characters

The `StrengthIndicatorType` provides the information how to present the strength of the password, as text (value `"Text"`) or as a kind of progress bar (value `"BarIndicator"`). In the `DisplayPosition` attribute, you configure where the information appears. Here is a complete example, including the ASP.NET AJAX `ScriptManager` control, the `PasswordStrength` control and of course a text box where the user may enter a password. For the sake of demonstration, the latter form field is a regular text field and not a password field so that you can see during development what you are typing.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
 <title>Control Toolkit</title>
</head>
<body>
 <form id="form1" runat="server">
 <asp:ScriptManager ID="asm" runat="server" />
 <div>
 <asp:TextBox ID="Password" runat="server" />
 <ajaxToolkit:PasswordStrength ID="ps1" runat="server"
 TargetControlID="Password" RequiresUpperAndLowerCaseCharacters="true"
 MinimumNumericCharacters="1" MinimumSymbolCharacters="1"
 PreferredPasswordLength="8" DisplayPosition="RightSide"
 StrengthIndicatorType="Text" />
 </div>
 </form>
</body>
</html>
```

Run the page and type away: Only after you have entered lowercase letters, uppercase letters, digits and symbols, the password is deemed as unbreakable .



Now the password is (quite) good (Click to view full-size image)

NEXT

Testing the Strength of a Password (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> Passwords are required almost anywhere, so that lazy users tend to choose simple passwords which are easy to break. The PasswordStrength control in the ASP.NET AJAX Control Toolkit can check how good a password is.

Overview

Passwords are required almost anywhere, so that lazy users tend to choose simple passwords which are easy to break. The `PasswordStrength` control in the ASP.NET AJAX Control Toolkit can check how good a password is.

Steps

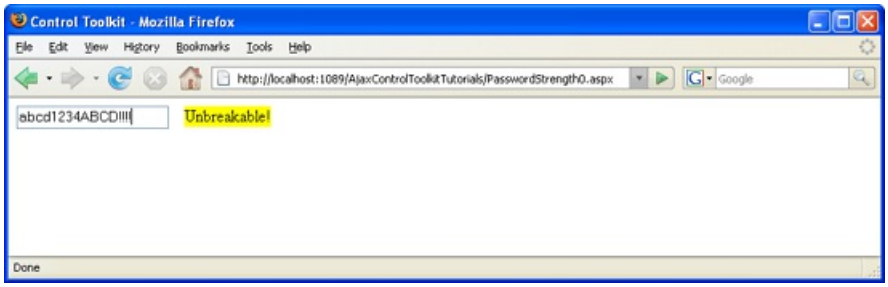
The `PasswordStrength` control extends a text box and checks whether the password in it is good enough. It offers a wealth of options via attributes; here are just some of them:

- `MinimumNumericCharacters` minimum number of numeric characters required in the password
- `MinimumSymbolCharacters` minimum number of symbol characters (not letters and digits) required in the password
- `PreferredPasswordLength` minimum length of the password
- `RequiresUpperAndLowerCaseCharacters` whether the password needs to use both uppercase and lowercase characters

The `StrengthIndicatorType` provides the information how to present the strength of the password, as text (value `"Text"`) or as a kind of progress bar (value `"BarIndicator"`). In the `DisplayPosition` attribute, you configure where the information appears. Here is a complete example, including the ASP.NET AJAX `ScriptManager` control, the `PasswordStrength` control and of course a text box where the user may enter a password. For the sake of demonstration, the latter form field is a regular text field and not a password field so that you can see during development what you are typing.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
 <title>Control Toolkit</title>
</head>
<body>
 <form id="form1" runat="server">
 <asp:ScriptManager ID="asm" runat="server" />
 <div>
 <asp:TextBox ID="Password" runat="server" />
 <ajaxToolkit:PasswordStrength ID="ps1" runat="server"
 TargetControlID="Password" RequiresUpperAndLowerCaseCharacters="true"
 MinimumNumericCharacters="1" MinimumSymbolCharacters="1"
 PreferredPasswordLength="8" DisplayPosition="RightSide"
 StrengthIndicatorType="Text" />
 </div>
 </form>
</body>
</html>
```

Run the page and type away: Only after you have entered lowercase letters, uppercase letters, digits and symbols, the password is deemed as unbreakable .



Now the password is (quite) good (Click to view full-size image)

PREVIOUS

Popup

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the Popup control to display a popup when any other control is activated.

- Using Multiple Popup Controls (C#)
- Handling Postbacks from A Popup Control With an UpdatePanel (C#)
- Handling Postbacks from A Popup Control Without an UpdatePanel (C#)
- Using Multiple Popup Controls (VB)
- Handling Postbacks from A Popup Control With an UpdatePanel (VB)
- Handling Postbacks from A Popup Control Without an UpdatePanel (VB)

Using Multiple Popup Controls (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The PopupControl extender in the AJAX Control Toolkit offers an easy way to trigger a popup when any other control is activated. It is also possible to use more than one popup control on one page.

Overview

The PopupControl extender in the AJAX Control Toolkit offers an easy way to trigger a popup when any other control is activated. It is also possible to use more than one popup control on one page.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Next, add a panel which serves as the popup. In the current scenario, the panel contains a `Calendar` control. In order to avoid the page refreshes caused by the Calendar's postbacks, the panel is put within an `UpdatePanel` control:

```
<asp:Panel ID="pnlCalendar" runat="server">
 <asp:UpdatePanel ID="up1" runat="server">
 <ContentTemplate>
 <asp:Calendar ID="c1" runat="server" />
 </ContentTemplate>
 </asp:UpdatePanel>
</asp:Panel>
```

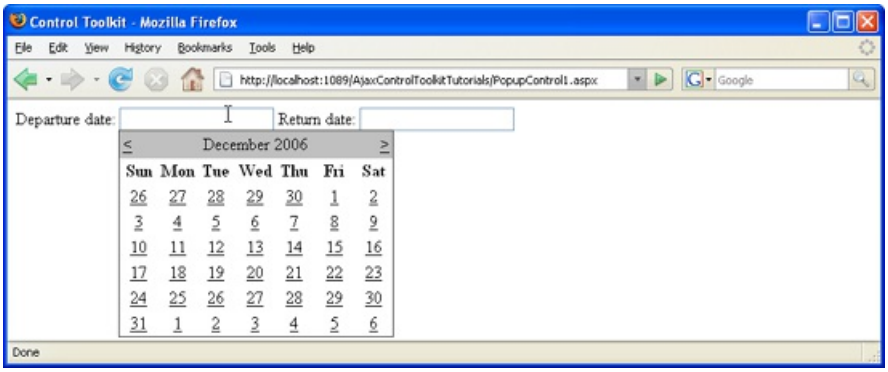
The page also contains two text boxes. For each text box, the calendar popup shall appear once the text box is activated.

```
<div>
 Departure date: <asp:TextBox ID="tbDeparture" runat="server" />
 Return date: <asp:TextBox ID="tbReturn" runat="server" />
</div>
```

Now extend each of the two text boxes with a `PopupControlExtender`. The `TargetControlID` attribute provides the ID of the control tied to the extender. The `PopupControlID` attribute contains the ID of the popup panel. In this case, both extenders show the same panel, but different panels are possible, as well.

```
<ajaxToolkit:PopupControlExtender ID="pce1" runat="server"
 TargetControlID="tbDeparture" PopupControlID="pnlCalendar" Position="Bottom" />
<ajaxToolkit:PopupControlExtender ID="pce2" runat="server"
 TargetControlID="tbReturn" PopupControlID="pnlCalendar" Position="Bottom" />
```

Now whenever you click within a text field, a calendar appears below the field, allowing you to select a date. (Getting the selected date back into the text boxes will be covered in a different tutorial.)



The Calendar appears when the user clicks into the textbox (Click to view full-size image)

NEXT

Handling Postbacks from A Popup Control With an UpdatePanel (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The PopupControl extender in the AJAX Control Toolkit offers an easy way to trigger a popup when any other control is activated. Special care has to be taken when a postback occurs within such a popup.

Overview

The PopupControl extender in the AJAX Control Toolkit offers an easy way to trigger a popup when any other control is activated. Special care has to be taken when a postback occurs within such a popup.

Steps

When using a `PopupControl` with a postback, an `UpdatePanel` can prevent the page refresh caused by the postback. The following markup defines a couple of important elements:

- A `ScriptManager` control so that the ASP.NET AJAX Control Toolkit works
- Two `TextBox` controls which will both trigger a popup
- A `Panel` control that will serve as the popup
- Within the panel, a `Calendar` control is embedded within an `UpdatePanel` control
- Two `PopupControlExtender` controls that assign the panel to the text boxes

```
<form id="form1" runat="server">
 <asp:ScriptManager ID="asm" runat="server" />
 <div>
 Departure date: <asp:TextBox ID="tbDeparture" runat="server" />
 Return date: <asp:TextBox ID="tbReturn" runat="server" />
 </div>
 <asp:Panel ID="pnlCalendar" runat="server">
 <asp:UpdatePanel ID="up1" runat="server">
 <ContentTemplate>
 <asp:Calendar ID="c1" runat="server"
 OnSelectionChanged="c1_SelectionChanged" />
 </ContentTemplate>
 </asp:UpdatePanel>
 </asp:Panel>
 <ajaxToolkit:PopupControlExtender ID="pce1" runat="server"
 TargetControlID="tbDeparture" PopupControlID="pnlCalendar" Position="Bottom" />
 <ajaxToolkit:PopupControlExtender ID="pce2" runat="server"
 TargetControlID="tbReturn" PopupControlID="pnlCalendar" Position="Bottom" />
</form>
```

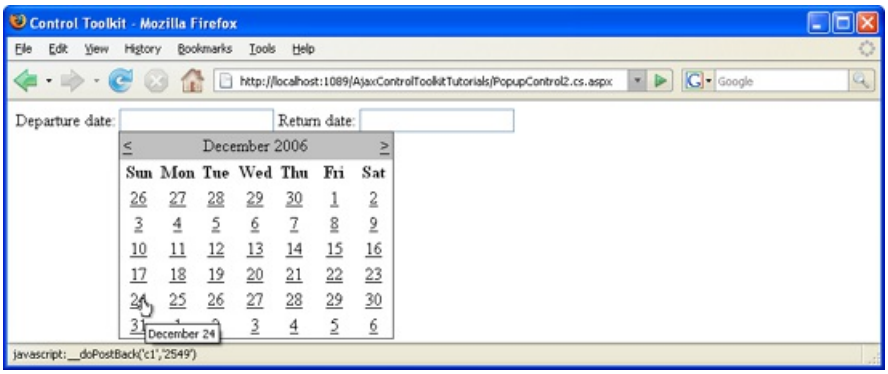
Note that the `OnSelectionChanged` attribute of the `Calendar` control is set. So when the user selects a date within the calendar, a postback occurs and the server-side method `c1_SelectionChanged()` is executed. Within that method, the current date must be retrieved and written back to the textbox.

The syntax for that is as follows: First of all, a proxy object for the `PopupControlExtender` on the page must be

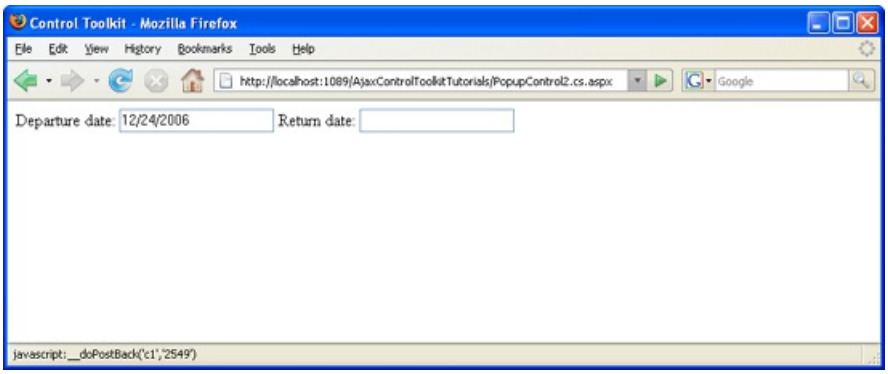
generated. The ASP.NET AJAX Control Toolkit offers the `GetProxyForCurrentPopup()` method. The object this method returns supports the `Commit()` method which sends a value back to the control that triggered the popup (not the control that triggered the method call!). The following code provides the selected date as the argument for the `Commit()` method, causing the code to write the selected date back to the text box:

```
<script runat="server">
 protected void c1_SelectionChanged(object sender, EventArgs e)
 {
 PopupControlExtender pce = AjaxControlToolkit.PopupControlExtender.GetProxyForCurrentPopup(Page);
 pce.Commit((sender as Calendar).SelectedDate.ToShortDateString());
 }
</script>
```

Now whenever you click on a calendar date, the selected date appears in the associated text box, creating a date picker control that can currently be found on many websites.



The Calendar appears when the user clicks into the textbox (Click to view full-size image)



Clicking on a date puts it in the textbox (Click to view full-size image)

PREVIOUS NEXT

Handling Postbacks from A Popup Control Without an UpdatePanel (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The PopupControl extender in the AJAX Control Toolkit offers an easy way to trigger a popup when any other control is activated. When a postback occurs in such a panel and there are several panels on the page it is hard to determine which panel has been clicked.

Overview

The PopupControl extender in the AJAX Control Toolkit offers an easy way to trigger a popup when any other control is activated. When a postback occurs in such a panel and there are several panels on the page it is hard to determine which panel has been clicked.

Steps

When using a `PopupControl` with a postback, but without having an `UpdatePanel` on the page, the Control Toolkit does not offer a way to determine which client element has triggered the popup which in turn caused the postback. However a small trick provides a workaround for this scenario.

First of all, here is the basic setup: two text boxes which both trigger the same popup, a calendar. Two `PopupControlExtenders` bring text boxes and popup together.

```
<form id="form1" runat="server">
 <asp:ScriptManager ID="asm" runat="server" />
 <div>
 Departure date: <asp:TextBox ID="tbDeparture" runat="server" />
 Return date: <asp:TextBox ID="tbReturn" runat="server" />
 </div>
 <asp:Panel ID="pnlCalendar" runat="server">
 <asp:Calendar ID="c1" runat="server" OnSelectionChanged="c1_SelectionChanged" />
 </asp:Panel>
 <ajaxToolkit:PopupControlExtender ID="pce1" runat="server"
 TargetControlID="tbDeparture" PopupControlID="pnlCalendar" Position="Bottom" />
 <ajaxToolkit:PopupControlExtender ID="pce2" runat="server"
 TargetControlID="tbReturn" PopupControlID="pnlCalendar" Position="Bottom" />
</form>
```

The basic idea is to add a hidden form field in the < `form` > element that holds the text box which launched the popup:

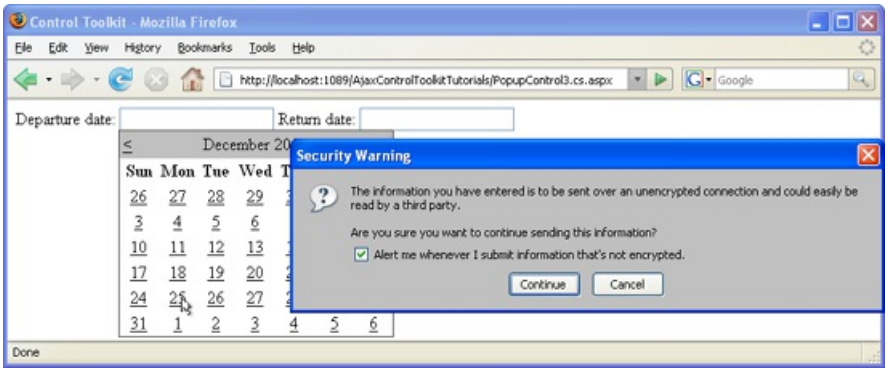
```
<input type="hidden" id="tbHidden" runat="server" />
```

When the page is loaded, JavaScript code adds an event handler to both text boxes: Whenever the user clicks on a text box, its name is written into the hidden form field:

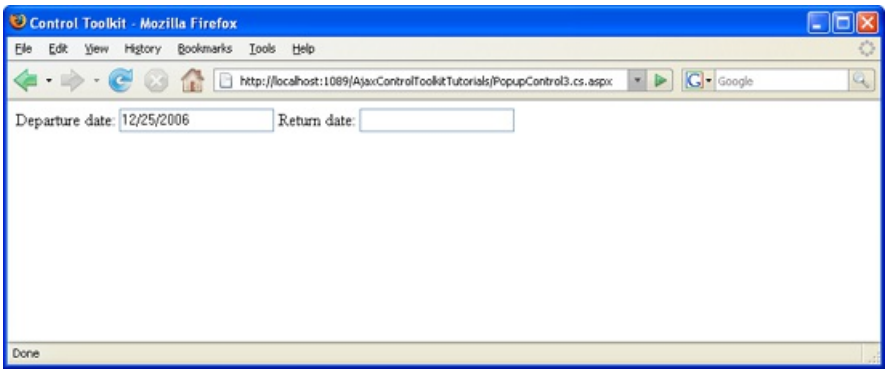
```
<script type="text/javascript">
 function pageLoad()
 {
 $get("tbDeparture").onclick = saveTextBox;
 $get("tbReturn").onclick = saveTextBox;
 }
 function saveTextBox()
 {
 $get("tbHidden").value = this.id;
 }
</script>
```

In the server-side code, the value of the hidden field must be read. Since hidden form fields are trivial to manipulate, a whitelist approach to validate the hidden value is required. Once the correct text box has been identified, the date from the calendar is written into it.

```
<script runat="server">
 protected void c1_SelectionChanged(object sender, EventArgs e)
 {
 string id = tbHidden.Value;
 if (id == "tbDeparture" || id == "tbReturn")
 {
 TextBox tb = (TextBox)FindControl(id);
 tb.Text = (sender as Calendar).SelectedDate.ToShortDateString();
 }
 }
</script>
```



The Calendar appears when the user clicks into the textbox (Click to view full-size image)



Clicking on a date puts it in the textbox (Click to view full-size image)

PREVIOUS NEXT

Using Multiple Popup Controls (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The PopupControl extender in the AJAX Control Toolkit offers an easy way to trigger a popup when any other control is activated. It is also possible to use more than one popup control on one page.

Overview

The PopupControl extender in the AJAX Control Toolkit offers an easy way to trigger a popup when any other control is activated. It is also possible to use more than one popup control on one page.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

Next, add a panel which serves as the popup. In the current scenario, the panel contains a `Calendar` control. In order to avoid the page refreshes caused by the Calendar's postbacks, the panel is put within an `UpdatePanel` control:

```
<asp:Panel ID="pnlCalendar" runat="server">
 <asp:UpdatePanel ID="up1" runat="server">
 <ContentTemplate>
 <asp:Calendar ID="c1" runat="server" />
 </ContentTemplate>
 </asp:UpdatePanel>
</asp:Panel>
```

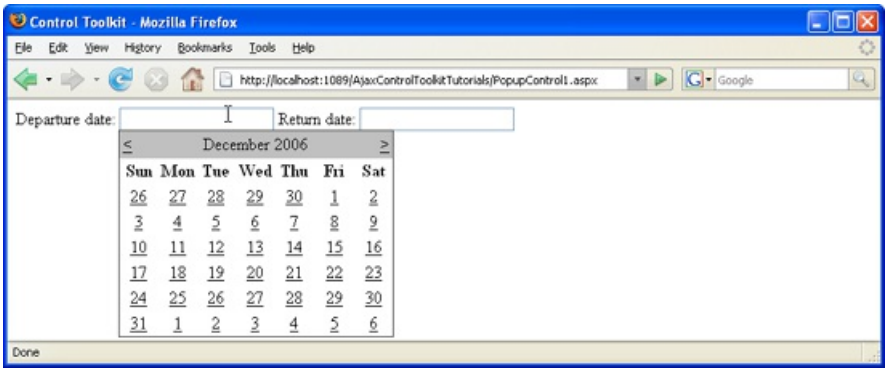
The page also contains two text boxes. For each text box, the calendar popup shall appear once the text box is activated.

```
<div>
 Departure date: <asp:TextBox ID="tbDeparture" runat="server" />
 Return date: <asp:TextBox ID="tbReturn" runat="server" />
</div>
```

Now extend each of the two text boxes with a `PopupControlExtender`. The `TargetControlID` attribute provides the ID of the control tied to the extender. The `PopupControlID` attribute contains the ID of the popup panel. In this case, both extenders show the same panel, but different panels are possible, as well.

```
<ajaxToolkit:PopupControlExtender ID="pce1" runat="server"
 TargetControlID="tbDeparture" PopupControlID="pnlCalendar" Position="Bottom" />
<ajaxToolkit:PopupControlExtender ID="pce2" runat="server"
 TargetControlID="tbReturn" PopupControlID="pnlCalendar" Position="Bottom" />
```

Now whenever you click within a text field, a calendar appears below the field, allowing you to select a date. (Getting the selected date back into the text boxes will be covered in a different tutorial.)



The Calendar appears when the user clicks into the textbox (Click to view full-size image)

PREVIOUS NEXT

Handling Postbacks from A Popup Control With an UpdatePanel (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The PopupControl extender in the AJAX Control Toolkit offers an easy way to trigger a popup when any other control is activated. Special care has to be taken when a postback occurs within such a popup.

Overview

The PopupControl extender in the AJAX Control Toolkit offers an easy way to trigger a popup when any other control is activated. Special care has to be taken when a postback occurs within such a popup.

Steps

When using a `PopupControl` with a postback, an `UpdatePanel` can prevent the page refresh caused by the postback. The following markup defines a couple of important elements:

- A `ScriptManager` control so that the ASP.NET AJAX Control Toolkit works
- Two `TextBox` controls which will both trigger a popup
- A `Panel` control that will serve as the popup
- Within the panel, a `Calendar` control is embedded within an `UpdatePanel` control
- Two `PopupControlExtender` controls that assign the panel to the text boxes

```
<form id="form1" runat="server">
 <asp:ScriptManager ID="asm" runat="server" />
 <div>
 Departure date: <asp:TextBox ID="tbDeparture" runat="server" />
 Return date: <asp:TextBox ID="tbReturn" runat="server" />
 </div>
 <asp:Panel ID="pnlCalendar" runat="server">
 <asp:UpdatePanel ID="up1" runat="server">
 <ContentTemplate>
 <asp:Calendar ID="c1" runat="server" OnSelectionChanged="c1_SelectionChanged" />
 </ContentTemplate>
 </asp:UpdatePanel>
 </asp:Panel>
 <ajaxToolkit:PopupControlExtender ID="pce1" runat="server"
 TargetControlID="tbDeparture" PopupControlID="pnlCalendar" Position="Bottom" />
 <ajaxToolkit:PopupControlExtender ID="pce2" runat="server"
 TargetControlID="tbReturn" PopupControlID="pnlCalendar" Position="Bottom" />
</form>
```

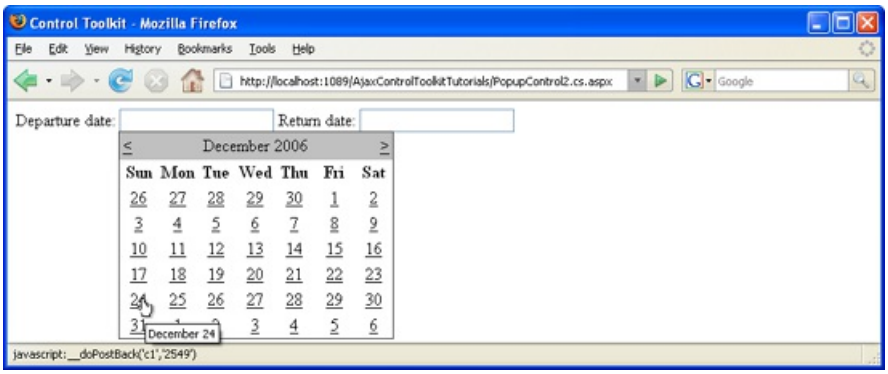
Note that the `OnSelectionChanged` attribute of the `Calendar` control is set. So when the user selects a date within the calendar, a postback occurs and the server-side method `c1_SelectionChanged()` is executed. Within that method, the current date must be retrieved and written back to the textbox.

The syntax for that is as follows: First of all, a proxy object for the `PopupControlExtender` on the page must be generated. The ASP.NET AJAX Control Toolkit offers the `GetProxyForCurrentPopup()` method. The object this

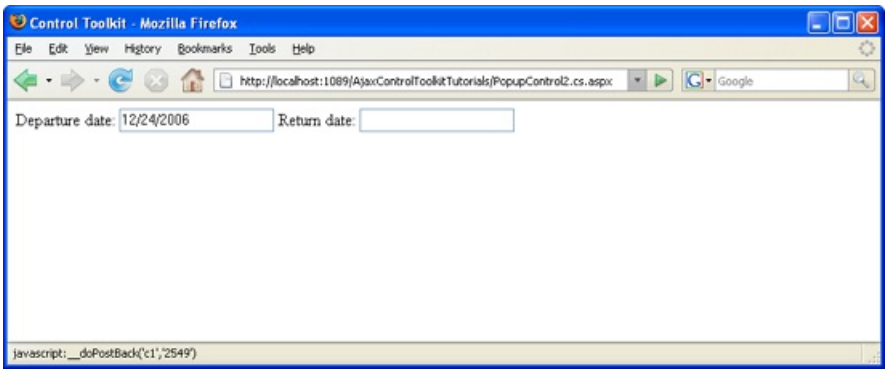
method returns supports the `Commit()` method which sends a value back to the control that triggered the popup (not the control that triggered the method call!). The following code provides the selected date as the argument for the `Commit()` method, causing the code to write the selected date back to the text box:

```
<script runat="server">
 Protected Sub c1_SelectionChanged(sender As object, e As EventArgs)
 Dim pce As PopupControlExtender = AjaxControlToolkit.PopupControlExtender.GetProxyForCurrentPopup(Page)
 pce.Commit(CType(sender, Calendar).SelectedDate.ToShortDateString())
 End Sub
</script>
```

Now whenever you click on a calendar date, the selected date appears in the associated text box, creating a date picker control that can currently be found on many websites.



The Calendar appears when the user clicks into the textbox (Click to view full-size image)



Clicking on a date puts it in the textbox (Click to view full-size image)

PREVIOUS NEXT

Handling Postbacks from A Popup Control Without an UpdatePanel (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The PopupControl extender in the AJAX Control Toolkit offers an easy way to trigger a popup when any other control is activated. When a postback occurs in such a panel and there are several panels on the page it is hard to determine which panel has been clicked.

Overview

The PopupControl extender in the AJAX Control Toolkit offers an easy way to trigger a popup when any other control is activated. When a postback occurs in such a panel and there are several panels on the page it is hard to determine which panel has been clicked.

Steps

When using a `PopupControl` with a postback, but without having an `UpdatePanel` on the page, the Control Toolkit does not offer a way to determine which client element has triggered the popup which in turn caused the postback. However a small trick provides a workaround for this scenario.

First of all, here is the basic setup: two text boxes which both trigger the same popup, a calendar. Two `PopupControlExtenders` bring text boxes and popup together.

```
<form id="form1" runat="server">
 <asp:ScriptManager ID="asm" runat="server" />
 <div>
 Departure date: <asp:TextBox ID="tbDeparture" runat="server" />
 Return date: <asp:TextBox ID="tbReturn" runat="server" />
 </div>
 <asp:Panel ID="pnlCalendar" runat="server">
 <asp:Calendar ID="c1" runat="server" OnSelectionChanged="c1_SelectionChanged" />
 </asp:Panel>
 <ajaxToolkit:PopupControlExtender ID="pce1" runat="server"
 TargetControlID="tbDeparture" PopupControlID="pnlCalendar" Position="Bottom" />
 <ajaxToolkit:PopupControlExtender ID="pce2" runat="server"
 TargetControlID="tbReturn" PopupControlID="pnlCalendar" Position="Bottom" />
</form>
```

The basic idea is to add a hidden form field in the < `form` > element that holds the text box which launched the popup:

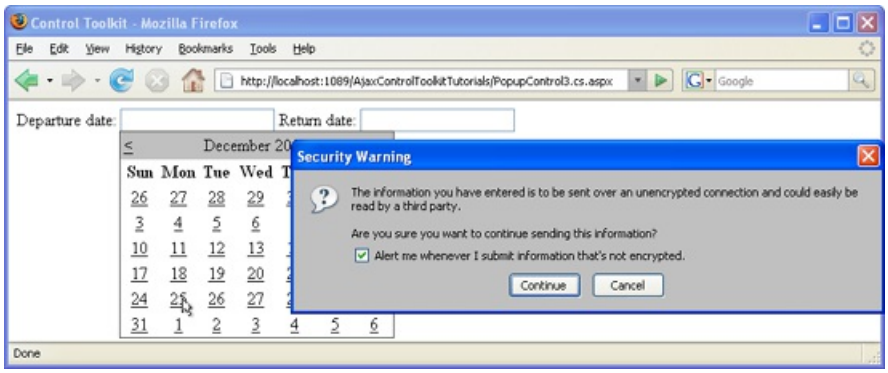
```
<input type="hidden" id="tbHidden" runat="server" />
```

When the page is loaded, JavaScript code adds an event handler to both text boxes: Whenever the user clicks on a text box, its name is written into the hidden form field:

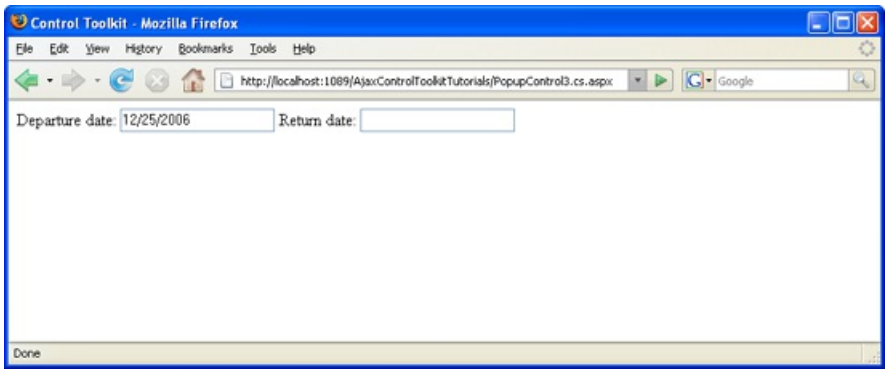
```
<script type="text/javascript">
 function pageLoad()
 {
 $get("tbDeparture").onclick = saveTextBox;
 $get("tbReturn").onclick = saveTextBox;
 }
 function saveTextBox()
 {
 $get("tbHidden").value = this.id;
 }
</script>
```

In the server-side code, the value of the hidden field must be read. Since hidden form fields are trivial to manipulate, a whitelist approach to validate the hidden value is required. Once the correct text box has been identified, the date from the calendar is written into it.

```
<script runat="server">
 Protected Sub c1_SelectionChanged(sender As object, e As EventArgs)
 Dim id As String = tbHidden.Value
 If (id = "tbDeparture" Or id = "tbReturn")
 Dim tb As TextBox = CType(FindControl(id), TextBox)
 tb.Text = CType(sender, Calendar).SelectedDate.ToShortDateString()
 End If
 End Sub
</script>
```



The Calendar appears when the user clicks into the textbox (Click to view full-size image)



Clicking on a date puts it in the textbox (Click to view full-size image)

PREVIOUS

Rating

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the Rating control to let users enter ratings for articles or items.

- Creating a Rating Control (C#)
- Creating a Rating Control (VB)

Creating a Rating Control (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> Many websites, from e-commerce to community sites, offer their users to rate articles or items. This usually requires some coding effort, but we do have the Control Toolkit to our disposal.

Overview

Many websites, from e-commerce to community sites, offer their users to rate articles or items. This usually requires some coding effort, but we do have the Control Toolkit to our disposal.

Steps

First of all, you need (at least) two kinds of images: one for a filled out rating item, and one for an empty rating item. A rating item is usually a star or a smiley. For this scenario, you find three files, smiley.png and empty.png and smiley-done.png as part of the source code downloads for this tutorial.

Then, create a new ASP.NET file and start with adding a `ScriptManager` control to it:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, add the `Rating` control from the ASP.NET AJAX Control Toolkit. The following attributes need to be set for this example:

- `CurrentRating` the initial rating to be used
- `MaxRating` the maximum rating
- `EmptyStarCssClass` the CSS class to use when a rating item (star) is empty
- `FilledStarCssClass` the CSS class to use when a rating item (star) is filled out
- `StarCssClass` the CSS class to use for a visible stat
- `WaitingStarCssClass` the CSS class to use while a star rating is sent back to the server

And here is the markup which creates a rating control with five items (smileys) of which none is filled out initially:

```
<ajaxToolkit:Rating ID="r1" runat="server"
 CurrentRating="0" MaxRating="5"
 EmptyStarCssClass="emptypng" FilledStarCssClass="smileypng"
 StarCssClass="smileypng" WaitingStarCssClass="donesmileypng"/>
```

The three referenced CSS classes now need to show the appropriate image files, which is easy to do using CSS:

```
<style type="text/css">
 .emptypng { background-image: url(empty.png); width: 32px; height: 32px; }
 .smileypng { background-image: url(smiley.png); width: 32px; height: 32px; }
 .donesmileypng { background-image: url(smiley-done.png); width: 32px; height: 32px; }
</style>
```

Make sure that you provide the width and height of the three images, otherwise the display may look a bit messed up.

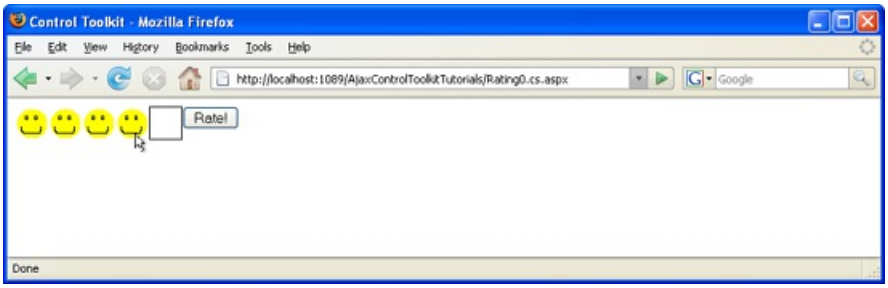
Finally, the result of the rating should be displayed to the user (or, at least saved in a database). So add a label for the output of a text message and a submit button to post back the rating form to the server:

```
<asp:Label ID="Label1" runat="server" />
<input type="submit" id="Submit1" runat="server" value="Rate!" />
```

In the server-side code, access the Rating control via its `ID` and then access its `CurrentRating` property which is the number of the selected rating items, in our example a value between 0 and 5.

```
<script runat="server">
 void Page_Load()
 {
 if (Page.IsPostBack)
 {
 Label1.Text = "Your rating: " + r1.CurrentRating;
 }
 }
</script>
```

Save the page and load it into your browser. When you hover over the (initially empty) rating items, a JavaScript effect occurs: The rating changes. When you click on the set of stars, the current rating is retained. Finally, when you submit the form, the server-side code outputs the selected rating.



Creating a rating system with minimal code (Click to view full-size image)

NEXT

Creating a Rating Control (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> Many websites, from e-commerce to community sites, offer their users to rate articles or items. This usually requires some coding effort, but we do have the Control Toolkit to our disposal.

Overview

Many websites, from e-commerce to community sites, offer their users to rate articles or items. This usually requires some coding effort, but we do have the Control Toolkit to our disposal.

Steps

First of all, you need (at least) two kinds of images: one for a filled out rating item, and one for an empty rating item. A rating item is usually a star or a smiley. For this scenario, you find three files, smiley.png and empty.png and smiley-done.png as part of the source code downloads for this tutorial.

Then, create a new ASP.NET file and start with adding a `ScriptManager` control to it:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Then, add the `Rating` control from the ASP.NET AJAX Control Toolkit. The following attributes need to be set for this example:

- `CurrentRating` the initial rating to be used
- `MaxRating` the maximum rating
- `EmptyStarCssClass` the CSS class to use when a rating item (star) is empty
- `FilledStarCssClass` the CSS class to use when a rating item (star) is filled out
- `StarCssClass` the CSS class to use for a visible stat
- `WaitingStarCssClass` the CSS class to use while a star rating is sent back to the server

And here is the markup which creates a rating control with five items (smileys) of which none is filled out initially:

```
<ajaxToolkit:Rating ID="r1" runat="server"
 CurrentRating="0" MaxRating="5"
 EmptyStarCssClass="emptypng" FilledStarCssClass="smileypng"
 StarCssClass="smileypng" WaitingStarCssClass="donesmileypng"/>
```

The three referenced CSS classes now need to show the appropriate image files, which is easy to do using CSS:

```
<style type="text/css">
 .emptypng { background-image: url(empty.png); width: 32px; height: 32px; }
 .smileypng { background-image: url(smiley.png); width: 32px; height: 32px; }
 .donesmileypng { background-image: url(smiley-done.png); width: 32px; height: 32px; }
</style>
```

Make sure that you provide the width and height of the three images, otherwise the display may look a bit messed up.

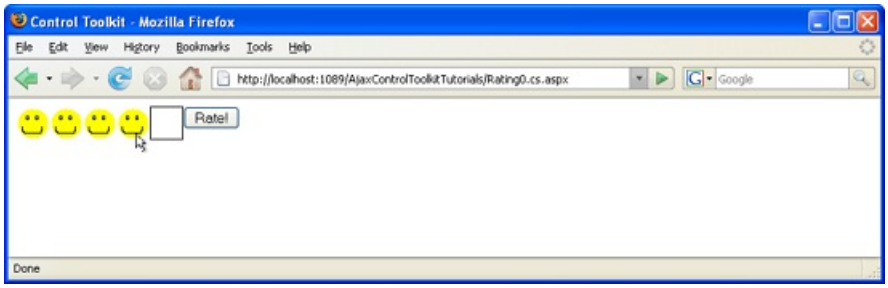
Finally, the result of the rating should be displayed to the user (or, at least saved in a database). So add a label for the output of a text message and a submit button to post back the rating form to the server:

```
<asp:Label ID="Label1" runat="server" />
<input type="submit" id="Submit1" runat="server" value="Rate!" />
```

In the server-side code, access the Rating control via its `ID` and then access its `CurrentRating` property which is the number of the selected rating items, in our example a value between 0 and 5.

```
<script runat="server">
 Sub Page_Load()
 If Page.IsPostBack Then
 Label1.Text = "Your rating: " & r1.CurrentRating
 End If
 End Sub
</script>
```

Save the page and load it into your browser. When you hover over the (initially empty) rating items, a JavaScript effect occurs: The rating changes. When you click on the set of stars, the current rating is retained. Finally, when you submit the form, the server-side code outputs the selected rating.



Creating a rating system with minimal code (Click to view full-size image)

PREVIOUS

ReorderList

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the ReorderList control to display a list that users can reorder by dragging items.

- Using Postbacks with ReorderList (C#)
- Drag and Drop via ReorderList (C#)
- Using Postbacks with ReorderList (VB)
- Drag and Drop via ReorderList (VB)

Using Postbacks with ReorderList (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ReorderList control in the AJAX Control Toolkit provides a list that can be reordered by the user via drag and drop. Whenever the list is reordered, a postback shall inform the server of the change.

Overview

The `ReorderList` control in the AJAX Control Toolkit provides a list that can be reordered by the user via drag and drop. Whenever the list is reordered, a postback shall inform the server of the change.

Steps

There are several possible data sources for the `ReorderList` control. One is to use an `XmlDataSource` control:

```
<asp:XmlDataSource ID="XmlDataSource1" runat="server" XPath="acronym/letter">
 <Data>
 <acronym>
 <letter char="A" description="Asynchronous" />
 <letter char="J" description="JavaScript" />
 <letter char="A" description="And" />
 <letter char="X" description="XML" />
 </acronym>
 </Data>
</asp:XmlDataSource>
```

In order to bind this XML to a `ReorderList` control and enable postbacks, the following attributes must be set:

- `DataSourceID`: The ID of the data source
- `SortOrderField`: The property to sort by
- `AllowReorder`: Whether to allow the user to reorder the list elements
- `PostBackOnReorder`: Whether to create a postback whenever the list is rearranged

Here is the appropriate markup for the control:

```
<ajaxToolkit:ReorderList ID="rl1" runat="server" SortOrderField="char"
 AllowReorder="true"
 DataSourceID="XmlDataSource1" PostBackOnReorder="true">
```

Within the `ReorderList` control, specific data from the data source may be bound using the `Eval()` method:

```
<DragHandleTemplate>
 <div class="DragHandleClass">
 </div>
 </DragHandleTemplate>
 <ItemTemplate>
 <div>
 <asp:Label ID="ItemLabel" Text='<%# Eval("description") %>' runat="server" />
 </div>
 </ItemTemplate>
</ajaxToolkit:ReorderList>
```

At an arbitrary position on the page, a label will hold the information when the last reordering occurred:

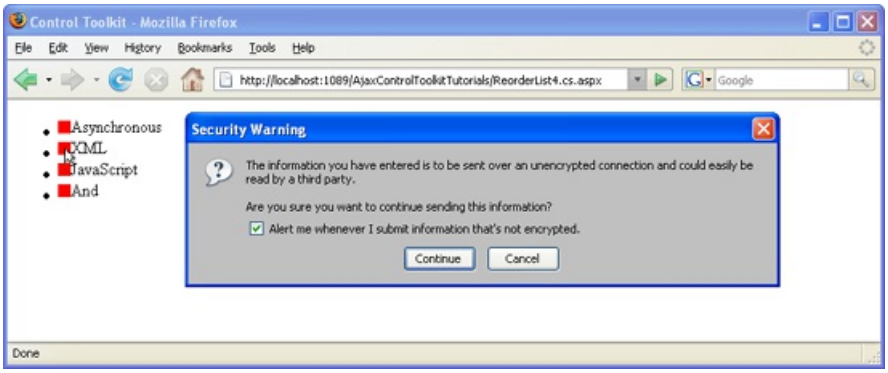
```
<div>
 <asp:Label ID="lastUpdate" runat="server" />
</div>
```

This label is filled with text in the server-side code, handling the postback:

```
<script runat="server">
 void Page_Load()
 {
 if (Page.IsPostBack)
 {
 lastUpdate.Text = "List last reordered at " +
 DateTime.Now.ToLongTimeString();
 }
 }
</script>
```

Finally, in order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put on the page:

```
<asp:ScriptManager ID="asm" runat="server" />
```



Each reordering triggers a postback (Click to view full-size image)

NEXT

Drag and Drop via ReorderList (C#)

2/20/2020 • 3 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ReorderList control in the AJAX Control Toolkit provides a list that can be reordered by the user via drag and drop. The current order of the list shall be persisted on the server.

Overview

The `ReorderList` control in the AJAX Control Toolkit provides a list that can be reordered by the user via drag and drop. The current order of the list shall be persisted on the server.

Steps

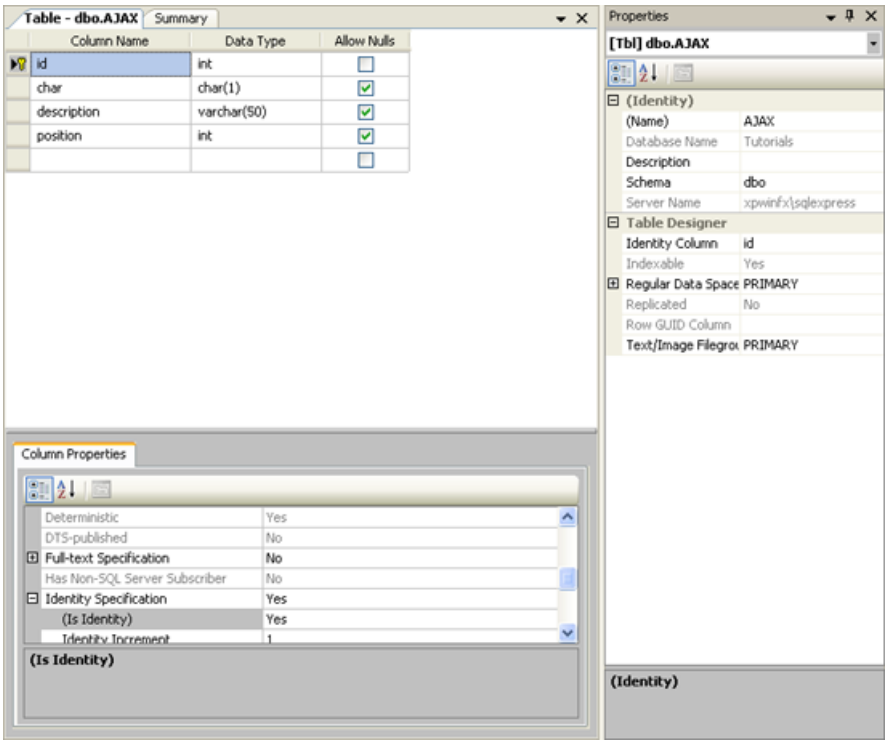
The `ReorderList` control supports binding data from a database to the list. Best of all, it also supports writing changes to the order of the list element back to the data store.

This sample uses Microsoft SQL Server 2005 Express Edition as the data store. The database is an optional (and free) part of a Visual Studio installation, including express edition. It is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database.

The easiest way to set up the database is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en). Connect to the server, double-click on `Databases` and create a new database (right-click and choose `New Database`) called `Tutorials` .

In this database, create a new table called `AJAX` with the following four columns:

- `id` (primary key, integer, identity, not NULL)
- `char` (char(1), NULL)
- `description` (varchar(50), NULL)
- `position` (int, NULL)



The layout of the AJAX table (Click to view full-size image)

Next, fill the table with a couple of values. Note that the `position` column holds the sort order of the elements.

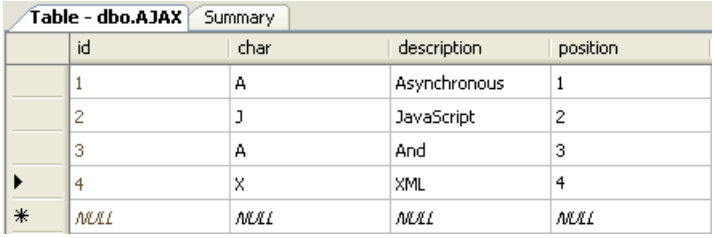


Table - dbo.AJAX | Summary

id	char	description	position
1	A	Asynchronous	1
2	J	JavaScript	2
3	A	And	3
4	X	XML	4
NULL	NULL	NULL	NULL

The initial data in the AJAX table (Click to view full-size image)

The next step requires to generate an `SqlDataSource` control to communicate with the new database and its table. The data source must support the `SELECT` and `UPDATE` SQL commands. When the order of the list elements is later changed, the `ReorderList` control automatically submits two values to the data source's `Update` command: the new position and the ID of the element. Therefore, the data source needs an `<UpdateParameters>` section for these two values:

```
<asp:SqlDataSource ID="sds" runat="server" ConnectionString="Data
 Source=(local)\SQLEXPRESS;Initial Catalog=Tutorials;Integrated Security=True"
 ProviderName="System.Data.SqlClient" OldValuesParameterFormatString="original_{0}"
 SelectCommand="SELECT [id], [char], [description], [position] FROM [AJAX] ORDER BY [position]"
 UpdateCommand="UPDATE [AJAX] SET position=@position WHERE [id]=@original_id">
 <UpdateParameters>
 <asp:Parameter Name="position" Type="Int32" />
 <asp:Parameter Name="original_id" Type="Int32" />
 </UpdateParameters>
</asp:SqlDataSource>
```

The `ReorderList` control needs to set the following attributes:

- `AllowReorder`: Whether the list items may be rearranged
- `DataSourceID`: The ID of the data source
- `DataKeyField`: The name of the primary key column in the data source
- `SortOrderField`: The data source column that provides the sort order for the list items

In the `<DragHandleTemplate>` and `<ItemTemplate>` sections, the layout of the list can be fine-tuned. Also, databinding is possible using the `Eval()` method, as seen here:

```
<ajaxToolkit:ReorderList ID="rl1" runat="server" SortOrderField="position"
 AllowReorder="true" DataSourceID="sds" DataKeyField="id">
 <DragHandleTemplate>
 <div class="DragHandleClass">
 </div>
 </DragHandleTemplate>
 <ItemTemplate>
 <asp:Label ID="ItemLabel" runat="server" Text='<%#Eval("description") %>' />
 </ItemTemplate>
</ajaxToolkit:ReorderList>
```

The following CSS style information (referenced in the `<DragHandleTemplate>` section of the `ReorderList` control) makes sure that the mouse pointer changes appropriately when it hovers over the drag handle:

```
<style type="text/css">
 .DragHandleClass
 {
 width: 12px;
 height: 12px;
 background-color: red;
 cursor:move;
 }
</style>
```

Finally, a `ScriptManager` control initializes ASP.NET AJAX for the page:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Run this example in the browser and rearrange the list items a bit. Then, reload the page and/or have a look at the database. The altered positions have been maintained and are also reflected by the values in the `position` column in the database and that all without any code, just by using markup.

Table - dbo.AJAX | Summary

id	char	description	position
1	A	Asynchronous	3
2	J	JavaScript	4
3	A	And	2
4	X	XML	1
NULL	NULL	NULL	NULL

The data in the database changes according to the new list item order (Click to view full-size image)

PREVIOUS NEXT

Using Postbacks with ReorderList (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ReorderList control in the AJAX Control Toolkit provides a list that can be reordered by the user via drag and drop. Whenever the list is reordered, a postback shall inform the server of the change.

Overview

The `ReorderList` control in the AJAX Control Toolkit provides a list that can be reordered by the user via drag and drop. Whenever the list is reordered, a postback shall inform the server of the change.

Steps

There are several possible data sources for the `ReorderList` control. One is to use an `XmlDataSource` control:

```
<asp:XmlDataSource ID="XmlDataSource1" runat="server" XPath="acronym/letter">
 <Data>
 <acronym>
 <letter char="A" description="Asynchronous" />
 <letter char="J" description="JavaScript" />
 <letter char="A" description="And" />
 <letter char="X" description="XML" />
 </acronym>
 </Data>
</asp:XmlDataSource>
```

In order to bind this XML to a `ReorderList` control and enable postbacks, the following attributes must be set:

- `DataSourceID`: The ID of the data source
- `SortOrderField`: The property to sort by
- `AllowReorder`: Whether to allow the user to reorder the list elements
- `PostBackOnReorder`: Whether to create a postback whenever the list is rearranged

Here is the appropriate markup for the control:

```
<ajaxToolkit:ReorderList ID="rl1" runat="server" SortOrderField="char"
 AllowReorder="true"
 DataSourceID="XmlDataSource1" PostBackOnReorder="true">
```

Within the `ReorderList` control, specific data from the data source may be bound using the `Eval()` method:

```
<DragHandleTemplate>
 <div class="DragHandleClass">
 </div>
 </DragHandleTemplate>
 <ItemTemplate>
 <div>
 <asp:Label ID="ItemLabel" Text='<%# Eval("description") %>' runat="server" />
 </div>
 </ItemTemplate>
</ajaxToolkit:ReorderList>
```

At an arbitrary position on the page, a label will hold the information when the last reordering occurred:

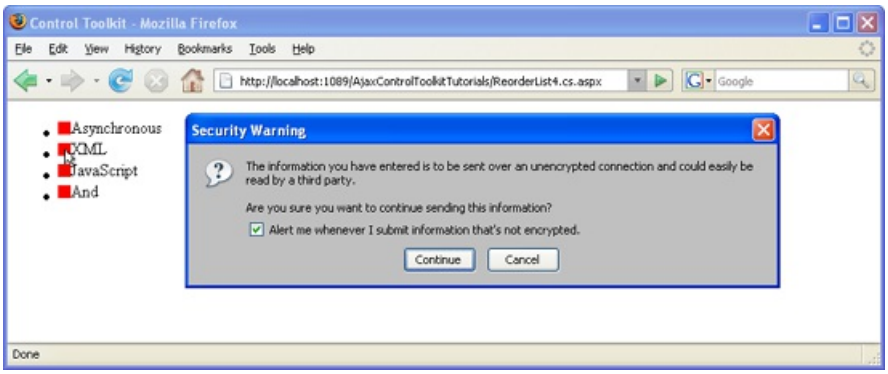
```
<div>
 <asp:Label ID="lastUpdate" runat="server" />
</div>
```

This label is filled with text in the server-side code, handling the postback:

```
<script runat="server">
 Sub Page_Load()
 If Page.IsPostBack Then
 lastUpdate.Text = "List last reordered at " & DateTime.Now.ToLongTimeString()
 End If
 End Sub
</script>
```

Finally, in order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put on the page:

```
<asp:ScriptManager ID="asm" runat="server" />
```



Each reordering triggers a postback (Click to view full-size image)

PREVIOUS NEXT

Drag and Drop via ReorderList (VB)

2/20/2020 • 3 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The ReorderList control in the AJAX Control Toolkit provides a list that can be reordered by the user via drag and drop. The current order of the list shall be persisted on the server.

Overview

The `ReorderList` control in the AJAX Control Toolkit provides a list that can be reordered by the user via drag and drop. The current order of the list shall be persisted on the server.

Steps

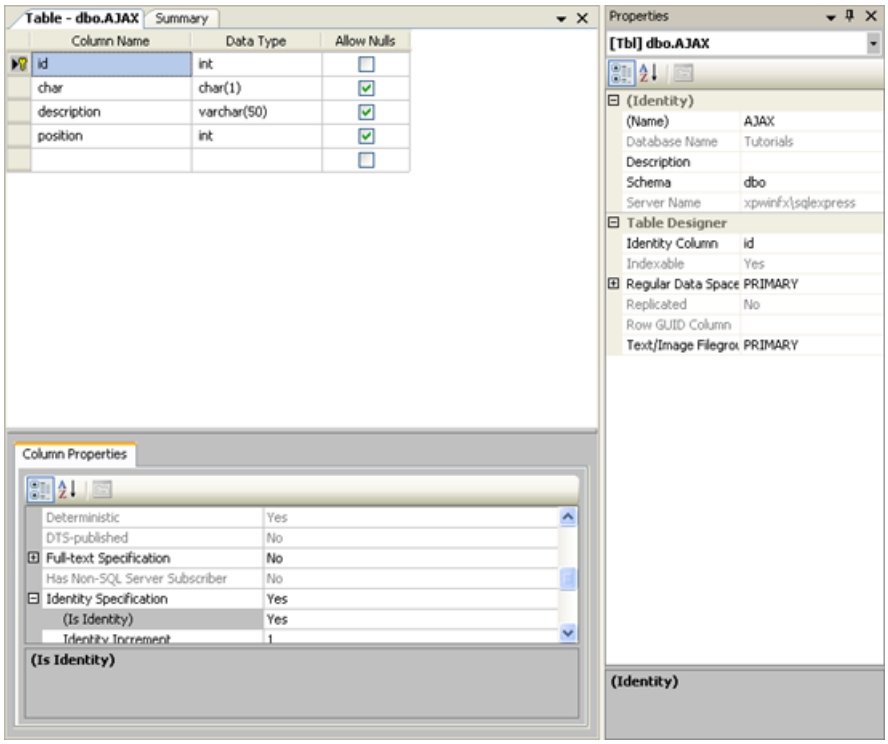
The `ReorderList` control supports binding data from a database to the list. Best of all, it also supports writing changes to the order of the list element back to the data store.

This sample uses Microsoft SQL Server 2005 Express Edition as the data store. The database is an optional (and free) part of a Visual Studio installation, including express edition. It is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database.

The easiest way to set up the database is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en). Connect to the server, double-click on `Databases` and create a new database (right-click and choose `New Database`) called `Tutorials`.

In this database, create a new table called `AJAX` with the following four columns:

- `id` (primary key, integer, identity, not NULL)
- `char` (char(1), NULL)
- `description` (varchar(50), NULL)
- `position` (int, NULL)



The layout of the AJAX table (Click to view full-size image)

Next, fill the table with a couple of values. Note that the `position` column holds the sort order of the elements.

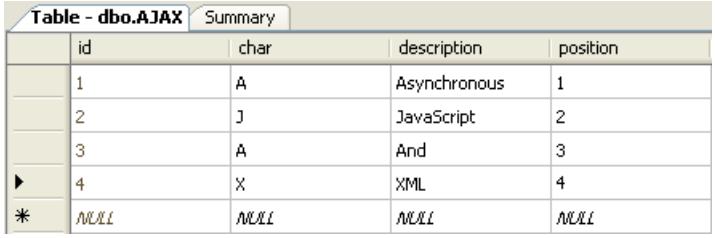


Table - dbo.AJAX | Summary

id	char	description	position
1	A	Asynchronous	1
2	J	JavaScript	2
3	A	And	3
4	X	XML	4
NULL	NULL	NULL	NULL

The initial data in the AJAX table (Click to view full-size image)

The next step requires to generate an `SqlDataSource` control to communicate with the new database and its table. The data source must support the `SELECT` and `UPDATE` SQL commands. When the order of the list elements is later changed, the `ReorderList` control automatically submits two values to the data source's `Update` command: the new position and the ID of the element. Therefore, the data source needs an `<UpdateParameters>` section for these two values:

```
<asp:SqlDataSource ID="sds" runat="server" ConnectionString="Data
 Source=(local)\SQLEXPRESS;Initial Catalog=Tutorials;Integrated Security=True"
 ProviderName="System.Data.SqlClient" OldValuesParameterFormatString="original_{0}"
 SelectCommand="SELECT [id], [char], [description], [position] FROM [AJAX] ORDER BY [position]"
 UpdateCommand="UPDATE [AJAX] SET position=@position WHERE [id]=@original_id">
 <UpdateParameters>
 <asp:Parameter Name="position" Type="Int32" />
 <asp:Parameter Name="original_id" Type="Int32" />
 </UpdateParameters>
</asp:SqlDataSource>
```

The `ReorderList` control needs to set the following attributes:

- `AllowReorder`: Whether the list items may be rearranged
- `DataSourceID`: The ID of the data source
- `DataKeyField`: The name of the primary key column in the data source
- `SortOrderField`: The data source column that provides the sort order for the list items

In the `<DragHandleTemplate>` and `<ItemTemplate>` sections, the layout of the list can be fine-tuned. Also, databinding is possible using the `Eval()` method, as seen here:

```
<ajaxToolkit:ReorderList ID="rl1" runat="server" SortOrderField="position"
 AllowReorder="true" DataSourceID="sds" DataKeyField="id">
 <DragHandleTemplate>
 <div class="DragHandleClass">
 </div>
 </DragHandleTemplate>
 <ItemTemplate>
 <asp:Label ID="ItemLabel" runat="server" Text='<%#Eval("description") %>' />
 </ItemTemplate>
</ajaxToolkit:ReorderList>
```

The following CSS style information (referenced in the `<DragHandleTemplate>` section of the `ReorderList` control) makes sure that the mouse pointer changes appropriately when it hovers over the drag handle:

```
<style type="text/css">
 .DragHandleClass
 {
 width: 12px;
 height: 12px;
 background-color: red;
 cursor:move;
 }
</style>
```

Finally, a `ScriptManager` control initializes ASP.NET AJAX for the page:

```
<asp:ScriptManager ID="asm" runat="server" />
```

Run this example in the browser and rearrange the list items a bit. Then, reload the page and/or have a look at the database. The altered positions have been maintained and are also reflected by the values in the `position` column in the database and that all without any code, just by using markup.

Table - dbo.AJAX | Summary

id	char	description	position
1	A	Asynchronous	3
2	J	JavaScript	4
3	A	And	2
4	X	XML	1
NULL	NULL	NULL	NULL

The data in the database changes according to the new list item order (Click to view full-size image)

PREVIOUS

Slider

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the Slider control to display a graphical slider that users can control using the mouse.

- Slider Control With Auto-Postback (C#)
- Databinding the Slider Control (C#)
- Slider Control With Auto-Postback (VB)
- Databinding the Slider Control (VB)

Using the Slider Control With Auto-Postback (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Slider control in the AJAX Control Toolkit provides a graphical slider that can be controlled using the mouse. It is possible to make the slider autopostback once its value changes.

Overview

The Slider control in the AJAX Control Toolkit provides a graphical slider that can be controlled using the mouse. It is possible to make the slider autopostback once its value changes.

Steps

In order to make the slider automatically postback upon a change, both text boxes need the attribute `AutoPostBack="true"`: The text box that will become the slider itself, and the text box that holds the slider's position. Here is the required markup for that:

```
<asp:TextBox ID="Slider1" runat="server" AutoPostBack="true" />
<asp:TextBox ID="SliderValue" runat="server" AutoPostBack="true" />
```

The `SliderExtender` control from the ASP.NET AJAX Control Toolkit assigns the slider functionality to the two text boxes:

```
<ajaxToolkit:SliderExtender ID="se1" runat="server"
 TargetControlId="Slider1" BoundControlID="SliderValue" />
```

An additional label element will later be used to inform the user of a postback:

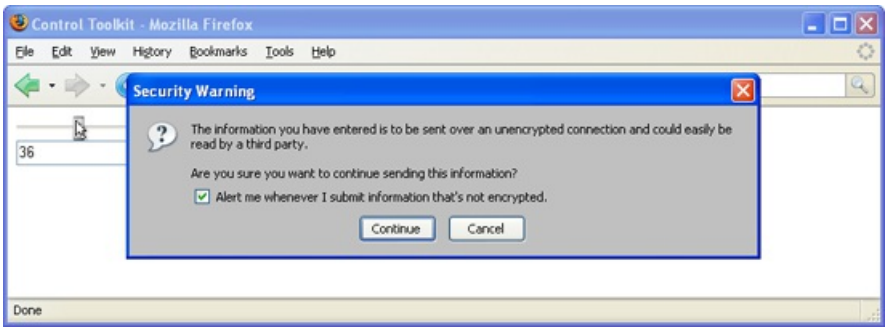
```
<asp:Label ID="LastUpdate" runat="server" />
```

Finally, the `ScriptManager` control of ASP.NET AJAX loads the required JavaScript for the Control Toolkit to work:

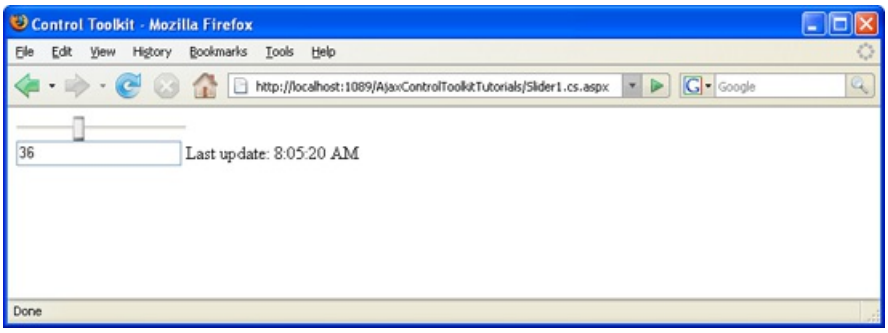
```
<asp:ScriptManager ID="asm" runat="server" />
```

Now the slider is posting back; on the server-side, this event may be caught and acted upon:

```
<script runat="server">
 void Page_Load()
 {
 if (Page.IsPostBack)
 {
 LastUpdate.Text = "Last update: " + DateTime.Now.ToLongTimeString();
 }
 }
</script>
```



Moving the slider triggers a postback (Click to view full-size image)



Afterwards, the date of this change is written in the label (Click to view full-size image)

NEXT

Databinding the Slider Control (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Slider control in the AJAX Control Toolkit provides a graphical slider that can be controlled using the mouse. It is possible to bind the current position of the slider to another ASP.NET control.

Overview

The Slider control in the AJAX Control Toolkit provides a graphical slider that can be controlled using the mouse. It is possible to bind the current position of the slider to another ASP.NET control.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

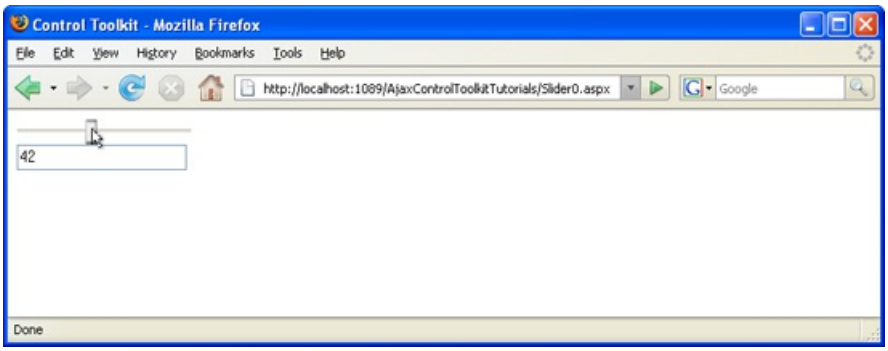
Next, add two `TextBox` controls to the page. One will be transformed into a graphical slider, and the other one will hold the position of the slider.

```
<asp:TextBox ID="Slider1" runat="server" />
<asp:TextBox ID="SliderValue" runat="server" />
```

The next step is already the final step. The `SliderExtender` control from the ASP.NET AJAX Control Toolkit makes a slider out of the first text box and automatically updates the second text box when the slider position changes. In order for that to work, The `SliderExtender`'s `TargetControlID` attribute must be set to the ID of the first text box; the `BoundControlID` attribute must be set to the ID of the second text box.

```
<ajaxToolkit:SliderExtender ID="se1" runat="server" TargetControlID="Slider1"
 BoundControlID="SliderValue" />
```

As you can see in the browser, the data binding works in both directions: entering a new value in the text box updates the slider's position. If you make the second text box read only, you may add a weak protection to the text field so that it is harder for the user to manually update the value in there.



Slider and text box are in sync (Click to view full-size image)

PREVIOUS NEXT

Using the Slider Control With Auto-Postback (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Slider control in the AJAX Control Toolkit provides a graphical slider that can be controlled using the mouse. It is possible to make the slider autopostback once its value changes.

Overview

The Slider control in the AJAX Control Toolkit provides a graphical slider that can be controlled using the mouse. It is possible to make the slider autopostback once its value changes.

Steps

In order to make the slider automatically postback upon a change, both text boxes need the attribute `AutoPostBack="true"`: The text box that will become the slider itself, and the text box that holds the slider's position. Here is the required markup for that:

```
<asp:TextBox ID="Slider1" runat="server" AutoPostBack="true" />
<asp:TextBox ID="SliderValue" runat="server" AutoPostBack="true" />
```

The `SliderExtender` control from the ASP.NET AJAX Control Toolkit assigns the slider functionality to the two text boxes:

```
<ajaxToolkit:SliderExtender ID="se1" runat="server"
 TargetControlId="Slider1" BoundControlID="SliderValue" />
```

An additional label element will later be used to inform the user of a postback:

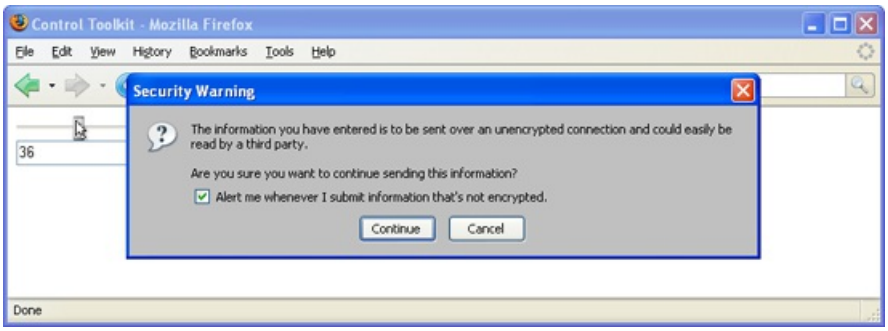
```
<asp:Label ID="LastUpdate" runat="server" />
```

Finally, the `ScriptManager` control of ASP.NET AJAX loads the required JavaScript for the Control Toolkit to work:

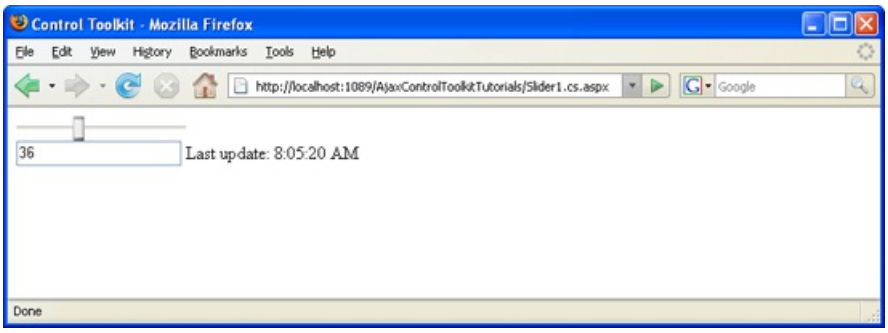
```
<asp:ScriptManager ID="asm" runat="server" />
```

Now the slider is posting back; on the server-side, this event may be caught and acted upon:

```
<script runat="server">
 Sub Page_Load()
 If Page.IsPostBack Then
 LastUpdate.Text = "Last update: " & DateTime.Now.ToLongTimeString()
 End If
 End Sub
</script>
```



Moving the slider triggers a postback (Click to view full-size image)



Afterwards, the date of this change is written in the label (Click to view full-size image)

PREVIOUS NEXT

Databinding the Slider Control (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The Slider control in the AJAX Control Toolkit provides a graphical slider that can be controlled using the mouse. It is possible to bind the current position of the slider to another ASP.NET control.

Overview

The Slider control in the AJAX Control Toolkit provides a graphical slider that can be controlled using the mouse. It is possible to bind the current position of the slider to another ASP.NET control.

Steps

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

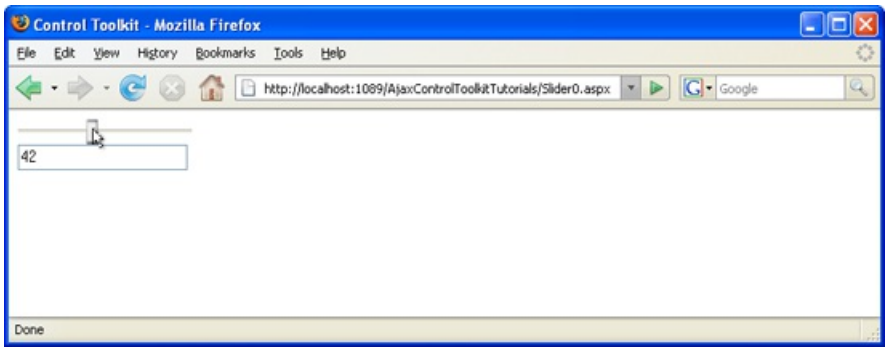
Next, add two `TextBox` controls to the page. One will be transformed into a graphical slider, and the other one will hold the position of the slider.

```
<asp:TextBox ID="Slider1" runat="server" />
<asp:TextBox ID="SliderValue" runat="server" />
```

The next step is already the final step. The `SliderExtender` control from the ASP.NET AJAX Control Toolkit makes a slider out of the first text box and automatically updates the second text box when the slider position changes. In order for that to work, The `SliderExtender`'s `TargetControlID` attribute must be set to the ID of the first text box; the `BoundControlID` attribute must be set to the ID of the second text box.

```
<ajaxToolkit:SliderExtender ID="se1" runat="server" TargetControlID="Slider1"
 BoundControlID="SliderValue" />
```

As you can see in the browser, the data binding works in both directions: entering a new value in the text box updates the slider's position. If you make the second text box read only, you may add a weak protection to the text field so that it is harder for the user to manually update the value in there.



Slider and text box are in sync (Click to view full-size image)

PREVIOUS

TextBoxWatermark

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to use the TextBoxWatermark control to display a text box with a message that disappears when users start typing in it.

- Using TextBoxWatermark in a FormView (C#)
- Using TextBoxWatermark With Validation Controls (C#)
- Using TextBoxWatermark in a FormView (VB)
- Using TextBoxWatermark With Validation Controls (VB)

Using TextBoxWatermark in a FormView (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The TextBoxWatermark control in the AJAX Control Toolkit extends a text box so that a text is displayed within the box. When a user clicks into the box, it is emptied. If the user leaves the box without entering text, the prefilled text reappears. This is also possible within a FormView control.

Overview

The `TextBoxWatermark` control in the AJAX Control Toolkit extends a text box so that a text is displayed within the box. When a user clicks into the box, it is emptied. If the user leaves the box without entering text, the prefilled text reappears. This is also possible within a `FormView` control.

Steps

First of all, a data source is required. This sample uses the AdventureWorks database and the Microsoft SQL Server 2005 Express Edition. The database is an optional part of a Visual Studio installation (including express edition) and is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. The AdventureWorks database is part of the SQL Server 2005 Samples and Sample Databases (download at https://www.microsoft.com/downloads/details.aspx?FamilyID=e719ecf7-9f46-4312-af89-6ad8702e4e6e&DisplayLang=en). The easiest way to set the database up is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en) and attach the `AdventureWorks.mdf` database file.

For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database.

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

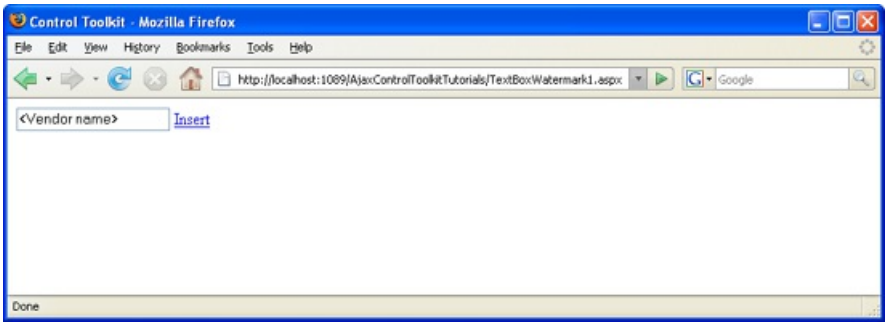
Then, add a data source to the page which supports the `DELETE`, `INSERT` and `UPDATE` SQL statements. If you are using the Visual Studio assistant to create the data source, mind that a bug in the current version does not prefix the table name (`Vendor`) with `Purchasing`. The following markup shows the correct syntax:

```
<asp:SqlDataSource ID="sds" runat="server" ConnectionString="Data
 Source=(local)\SQLEXPRESS;Initial Catalog=AdventureWorks;Integrated Security=True"
 DeleteCommand="DELETE FROM [Purchasing].[Vendor] WHERE [VendorID] = @VendorID"
 InsertCommand="INSERT INTO [Purchasing].[Vendor] ([Name]) VALUES (@Name)"
 ProviderName="System.Data.SqlClient"
 SelectCommand="SELECT [VendorID], [Name] FROM [Purchasing].[Vendor]"
 UpdateCommand="UPDATE [Purchasing].[Vendor] SET [Name] = @Name WHERE [VendorID] = @VendorID">
 <DeleteParameters>
 <asp:Parameter Name="VendorID" Type="Int32" />
 </DeleteParameters>
 <UpdateParameters>
 <asp:Parameter Name="Name" Type="String" />
 <asp:Parameter Name="VendorID" Type="Int32" />
 </UpdateParameters>
 <InsertParameters>
 <asp:Parameter Name="Name" Type="String" />
 </InsertParameters>
</asp:SqlDataSource>
```

Remember the name (`ID`) of the data source, since it will be used in the `DataSourceID` property of the `FormView` control. The `<InsertItemTemplate>` section of the `FormView` contains a textbox which is extended by the `TextBoxWatermarkExtender` control. Make sure that the extender resides within the template and not outside of it.

```
<div>
 <asp:FormView ID="FormView1" runat="server" DataSourceID="sds" AllowPaging="True">
 <ItemTemplate>
 <%# Eval("Name") %>
 <asp:LinkButton ID="btnNew" runat="server" CommandName="New" Text="Insert" />
 <asp:LinkButton ID="btnEdit" runat="server" CommandName="Edit" Text="Edit" />
 <asp:LinkButton ID="btnDelete" runat="server" CommandName="Delete" Text="Delete" />
 </ItemTemplate>
 <EditItemTemplate>
 <asp:TextBox ID="tbEdit" runat="server" Text='<%# Bind("Name") %>' />
 <asp:LinkButton ID="btnUpdate" runat="server" CommandName="Update" Text="Update" />
 </EditItemTemplate>
 <InsertItemTemplate>
 <asp:TextBox ID="tbNew" runat="server" Text='<%# Bind("Name") %>' />
 <asp:LinkButton ID="btnInsert" runat="server" CommandName="Insert" Text="Insert" />
 <ajaxToolkit:TextBoxWatermarkExtender ID="tbwe" runat="server"
 TargetControlID="tbNew" WatermarkText=" Vendor name " />
 </InsertItemTemplate>
 </asp:FormView>
</div>
```

Now when the user changes into the insert mode of the `FormView` control, the text field for the new vendor is prefilled thanks to the `TextBoxWatermarkExtender` control. A click inside the textbox lets the filler text disappear.



The watermark in the field comes from the extender (Click to view full-size image)

NEXT

Using TextBoxWatermark With Validation Controls (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The TextBoxWatermark control in the AJAX Control Toolkit extends a text box so that a text is displayed within the box. When a user clicks into the box, it is emptied. If the user leaves the box without entering text, the prefilled text reappears. This may collide with ASP.NET Validation Controls on the same page, but these issues may be overcome.

Overview

The `TextBoxWatermark` control in the AJAX Control Toolkit extends a text box so that a text is displayed within the box. When a user clicks into the box, it is emptied. If the user leaves the box without entering text, the prefilled text reappears. This may collide with ASP.NET Validation Controls on the same page, but these issues may be overcome.

Steps

The basic setup of the sample is the following: a `TextBox` control is watermarked using a `TextBoxWatermarkExtender` control. A button triggers a postback and will later be used to trigger the validation controls on the page. Also, a `ScriptManager` control is required to initialize ASP.NET AJAX:

```
<form id="form1" runat="server">
 <asp:ScriptManager ID="asm" runat="server" />
 <div>
 <asp:TextBox ID="Name" runat="server" /> <br />
 <asp:Button ID="btn" runat="server" Text="Submit" OnClick="btn_Click" /><br />
 <asp:Label ID="lbl" runat="server" />
 </div>
 <ajaxToolkit:TextBoxWatermarkExtender ID="tbwe" runat="server"
 TargetControlID="Name" WatermarkText=" Your Name " />
</form>
```

Now add a `RequiredFieldValidator` control that checks whether there is text in the field when the form is submitted. The `InitialValue` property of the validator must be set to the same value that is used in the `TextBoxWatermarkExtender` control: When the form is submitted, the value of an unchanged textbox is the watermark value within it:

```
<asp:RequiredFieldValidator ID="rfv1" ControlToValidate="Name"
 Text="*" InitialValue=" Your Name " Display="Dynamic" runat="server" />
```

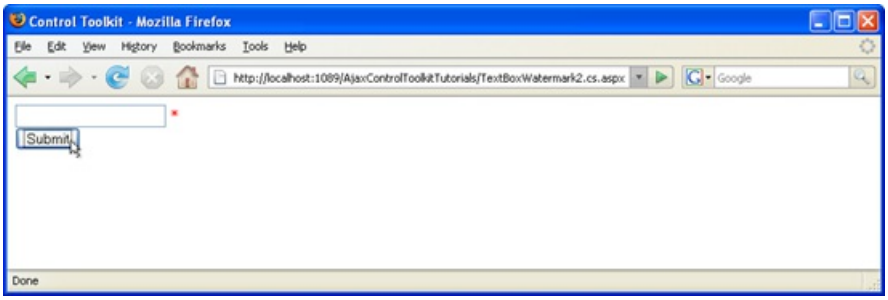
However there is one problem with this approach: If the client disables JavaScript, the text field is not prefilled with the watermark text, therefore the `RequiredFieldValidator` does not trigger an error message. Therefore, a second `RequiredFieldValidator` control is required which checks for an empty text box (omitting the `InitialValue` attribute).

```
<asp:RequiredFieldValidator ID="rfv2" ControlToValidate="Name"
 Text="*" Display="Dynamic" runat="server" />
```

Since both validators use `Display` = `"Dynamic"`, the end user cannot distinguish from the visual appearance which of the two validators was fired; instead, it looks like there was only one of them.

Finally, add some server-side code to output the text in the field if no validator issued an error message:

```
<script runat="server">
 protected void btn_Click(object sender, EventArgs e)
 {
 lbl.Text = HttpUtility.HtmlEncode(Name.Text);
 }
</script>
```



The validator complains that there is no text in the field (Click to view full-size image)

PREVIOUS NEXT

Using TextBoxWatermark in a FormView (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The TextBoxWatermark control in the AJAX Control Toolkit extends a text box so that a text is displayed within the box. When a user clicks into the box, it is emptied. If the user leaves the box without entering text, the prefilled text reappears. This is also possible within a FormView control.

Overview

The `TextBoxWatermark` control in the AJAX Control Toolkit extends a text box so that a text is displayed within the box. When a user clicks into the box, it is emptied. If the user leaves the box without entering text, the prefilled text reappears. This is also possible within a `FormView` control.

Steps

First of all, a data source is required. This sample uses the AdventureWorks database and the Microsoft SQL Server 2005 Express Edition. The database is an optional part of a Visual Studio installation (including express edition) and is also available as a separate download under https://go.microsoft.com/fwlink/?LinkId=64064. The AdventureWorks database is part of the SQL Server 2005 Samples and Sample Databases (download at https://www.microsoft.com/downloads/details.aspx?FamilyID=e719ecf7-9f46-4312-af89-6ad8702e4e6e&DisplayLang=en). The easiest way to set the database up is to use the Microsoft SQL Server Management Studio Express (https://www.microsoft.com/downloads/details.aspx?FamilyID=c243a5ae-4bd1-4e3d-94b8-5a0f62bf7796&DisplayLang=en) and attach the `AdventureWorks.mdf` database file.

For this sample, we assume that the instance of the SQL Server 2005 Express Edition is called `SQLEXPRESS` and resides on the same machine as the web server; this is also the default setup. If your setup differs, you have to adapt the connection information for the database.

In order to activate the functionality of ASP.NET AJAX and the Control Toolkit, the `ScriptManager` control must be put anywhere on the page (but within the `<form>` element):

```
<asp:ScriptManager ID="asm" runat="server" />
```

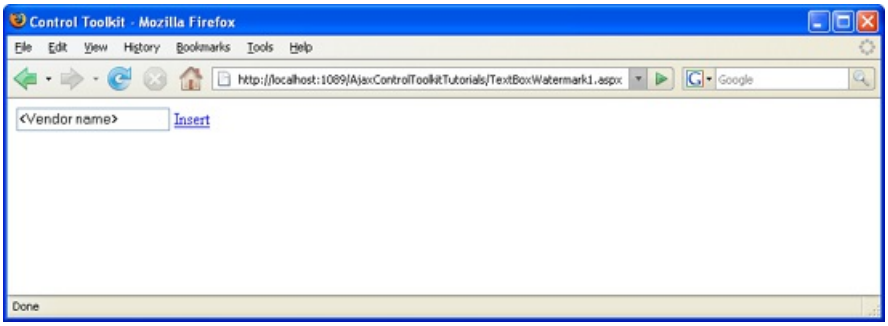
Then, add a data source to the page which supports the `DELETE`, `INSERT` and `UPDATE` SQL statements. If you are using the Visual Studio assistant to create the data source, mind that a bug in the current version does not prefix the table name (`Vendor`) with `Purchasing`. The following markup shows the correct syntax:

```
<asp:SqlDataSource ID="sds" runat="server" ConnectionString="Data
 Source=(local)\SQLEXPRESS;Initial Catalog=AdventureWorks;Integrated Security=True"
 DeleteCommand="DELETE FROM [Purchasing].[Vendor] WHERE [VendorID] = @VendorID"
 InsertCommand="INSERT INTO [Purchasing].[Vendor] ([Name]) VALUES (@Name)"
 ProviderName="System.Data.SqlClient"
 SelectCommand="SELECT [VendorID], [Name] FROM [Purchasing].[Vendor]"
 UpdateCommand="UPDATE [Purchasing].[Vendor] SET [Name] = @Name WHERE [VendorID] = @VendorID">
 <DeleteParameters>
 <asp:Parameter Name="VendorID" Type="Int32" />
 </DeleteParameters>
 <UpdateParameters>
 <asp:Parameter Name="Name" Type="String" />
 <asp:Parameter Name="VendorID" Type="Int32" />
 </UpdateParameters>
 <InsertParameters>
 <asp:Parameter Name="Name" Type="String" />
 </InsertParameters>
</asp:SqlDataSource>
```

Remember the name (`ID`) of the data source, since it will be used in the `DataSourceID` property of the `FormView` control. The `<InsertItemTemplate>` section of the `FormView` contains a textbox which is extended by the `TextBoxWatermarkExtender` control. Make sure that the extender resides within the template and not outside of it.

```
<div>
 <asp:FormView ID="FormView1" runat="server" DataSourceID="sds" AllowPaging="True">
 <ItemTemplate>
 <%# Eval("Name") %>
 <asp:LinkButton ID="btnNew" runat="server" CommandName="New" Text="Insert" />
 <asp:LinkButton ID="btnEdit" runat="server" CommandName="Edit" Text="Edit" />
 <asp:LinkButton ID="btnDelete" runat="server" CommandName="Delete" Text="Delete" />
 </ItemTemplate>
 <EditItemTemplate>
 <asp:TextBox ID="tbEdit" runat="server" Text='<%# Bind("Name") %>' />
 <asp:LinkButton ID="btnUpdate" runat="server" CommandName="Update" Text="Update" />
 </EditItemTemplate>
 <InsertItemTemplate>
 <asp:TextBox ID="tbNew" runat="server" Text='<%# Bind("Name") %>' />
 <asp:LinkButton ID="btnInsert" runat="server" CommandName="Insert" Text="Insert" />
 <ajaxToolkit:TextBoxWatermarkExtender ID="tbwe" runat="server"
 TargetControlID="tbNew" WatermarkText=" Vendor name " />
 </InsertItemTemplate>
 </asp:FormView>
</div>
```

Now when the user changes into the insert mode of the `FormView` control, the text field for the new vendor is prefilled thanks to the `TextBoxWatermarkExtender` control. A click inside the textbox lets the filler text disappear.



The watermark in the field comes from the extender (Click to view full-size image)

Using TextBoxWatermark With Validation Controls (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Christian Wenz

Download Code or Download PDF

> The TextBoxWatermark control in the AJAX Control Toolkit extends a text box so that a text is displayed within the box. When a user clicks into the box, it is emptied. If the user leaves the box without entering text, the prefilled text reappears. This may collide with ASP.NET Validation Controls on the same page, but these issues may be overcome.

Overview

The `TextBoxWatermark` control in the AJAX Control Toolkit extends a text box so that a text is displayed within the box. When a user clicks into the box, it is emptied. If the user leaves the box without entering text, the prefilled text reappears. This may collide with ASP.NET Validation Controls on the same page, but these issues may be overcome.

Steps

The basic setup of the sample is the following: a `TextBox` control is watermarked using a `TextBoxWatermarkExtender` control. A button triggers a postback and will later be used to trigger the validation controls on the page. Also, a `ScriptManager` control is required to initialize ASP.NET AJAX:

```
<form id="form1" runat="server">
 <asp:ScriptManager ID="asm" runat="server" />
 <div>
 <asp:TextBox ID="Name" runat="server" /> <br />
 <asp:Button ID="btn" runat="server" Text="Submit" OnClick="btn_Click" /><br />
 <asp:Label ID="lbl" runat="server" />
 </div>
 <ajaxToolkit:TextBoxWatermarkExtender ID="tbwe" runat="server"
 TargetControlID="Name" WatermarkText=" Your Name " />
</form>
```

Now add a `RequiredFieldValidator` control that checks whether there is text in the field when the form is submitted. The `InitialValue` property of the validator must be set to the same value that is used in the `TextBoxWatermarkExtender` control: When the form is submitted, the value of an unchanged textbox is the watermark value within it:

```
<asp:RequiredFieldValidator ID="rfv1" ControlToValidate="Name"
 Text="*" InitialValue=" Your Name " Display="Dynamic" runat="server" />
```

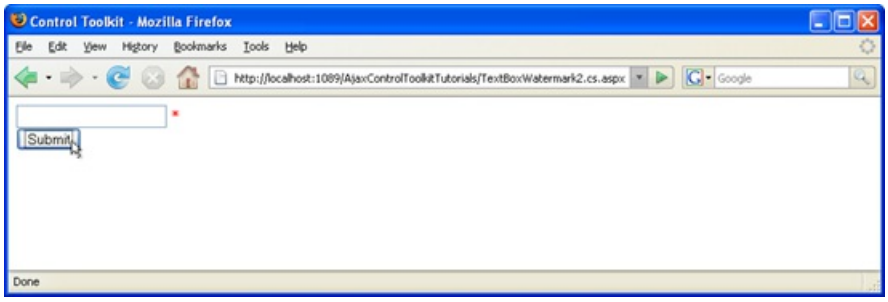
However there is one problem with this approach: If the client disables JavaScript, the text field is not prefilled with the watermark text, therefore the `RequiredFieldValidator` does not trigger an error message. Therefore, a second `RequiredFieldValidator` control is required which checks for an empty text box (omitting the `InitialValue` attribute).

```
<asp:RequiredFieldValidator ID="rfv2" ControlToValidate="Name"
 Text="*" Display="Dynamic" runat="server" />
```

Since both validators use `Display` = `"Dynamic"`, the end user cannot distinguish from the visual appearance which of the two validators was fired; instead, it looks like there was only one of them.

Finally, add some server-side code to output the text in the field if no validator issued an error message:

```
<script runat="server">
 Protected Sub btn_Click(ByVal sender As Object, ByVal e As System.EventArgs)
 lbl.Text = HttpUtility.HtmlEncode(Name.Text)
 End Sub
</script>
```



The validator complains that there is no text in the field (Click to view full-size image)

PREVIOUS

Older Versions - Data Access

2/20/2020 • 5 minutes to read • Edit Online

How to perform a variety of data-related tasks using ASP.NET Web Forms data controls.

- Introduction
  - Creating a Data Access Layer (C#)
  - Creating a Business Logic Layer (C#)
  - Master Pages and Site Navigation (C#)
  - Creating a Data Access Layer (VB)
  - Creating a Business Logic Layer (VB)
  - Master Pages and Site Navigation (VB)
- Basic Reporting
  - Displaying Data With the ObjectDataSource (C#)
  - Declarative Parameters (C#)
  - Programmatically Setting the ObjectDataSource's Parameter Values (C#)
  - Displaying Data With the ObjectDataSource (VB)
  - Declarative Parameters (VB)
  - Programmatically Setting the ObjectDataSource's Parameter Values (VB)
- Master/Detail
  - Master/Detail Filtering With a DropDownList (C#)
  - Master/Detail Filtering With Two DropDownLists (C#)
  - Master/Detail Filtering Across Two Pages (C#)
  - Master/Detail Using a Selectable Master GridView with a Details DetailView (C#)
  - Master/Detail Filtering With a DropDownList (VB)
  - Master/Detail Filtering With Two DropDownLists (VB)
  - Master/Detail Filtering Across Two Pages (VB)
  - Master/Detail Using a Selectable Master GridView with a Details DetailView (VB)
- Custom Formatting
  - Custom Formatting Based Upon Data (C#)
  - Using TemplateFields in the GridView Control (C#)
  - Using TemplateFields in the DetailsView Control (C#)
  - FormView's Templates (C#)
  - Displaying Summary Information in the GridView's Footer (C#)
  - Custom Formatting Based Upon Data (VB)
  - Using TemplateFields in the GridView Control (VB)
  - Using TemplateFields in the DetailsView Control (VB)
  - FormView's Templates (VB)
  - Displaying Summary Information in the GridView's Footer (VB)
- Editing, Inserting, and Deleting Data
  - Overview of Inserting, Updating, and Deleting Data (C#)

  - Examining the Events Associated with Inserting, Updating, and Deleting (C#)
  - Handling BLL- and DAL-Level Exceptions in an ASP.NET Page (C#)
  - Adding Validation Controls to the Editing and Inserting Interfaces (C#)
  - Customizing the Data Modification Interface (C#)
  - Implementing Optimistic Concurrency (C#)
  - Adding Client-Side Confirmation When Deleting (C#)
  - Limiting Data Modification Functionality Based on the User (C#)
  - Overview of Inserting, Updating, and Deleting Data (VB)
  - Examining the Events Associated with Inserting, Updating, and Deleting (VB)
  - Handling BLL- and DAL-Level Exceptions in an ASP.NET Page (VB)
  - Adding Validation Controls to the Editing and Inserting Interfaces (VB)
  - Customizing the Data Modification Interface (VB)
  - Implementing Optimistic Concurrency (VB)
  - Adding Client-Side Confirmation When Deleting (VB)
  - Limiting Data Modification Functionality Based on the User (VB)
- Paging and Sorting
  - Paging and Sorting Report Data (C#)
  - Efficiently Paging Through Large Amounts of Data (C#)
  - Sorting Custom Paged Data (C#)
  - Creating a Customized Sorting User Interface (C#)
  - Paging and Sorting Report Data (VB)
  - Efficiently Paging Through Large Amounts of Data (VB)
  - Sorting Custom Paged Data (VB)
  - Creating a Customized Sorting User Interface (VB)
- Custom Button Actions
  - Adding and Responding to Buttons to a GridView (C#)
  - Adding and Responding to Buttons to a GridView (VB)
- Displaying Data with the DataList and Repeater
  - Displaying Data with the DataList and Repeater Controls (C#)
  - Formatting the DataList and Repeater Based Upon Data (C#)
  - Showing Multiple Records per Row with the DataList Control (C#)
  - Nested Data Web Controls (C#)
  - Displaying Data with the DataList and Repeater Controls (VB)
  - Formatting the DataList and Repeater Based Upon Data (VB)
  - Showing Multiple Records per Row with the DataList Control (VB)
  - Nested Data Web Controls (VB)
- Filtering Scenarios with the DataList and Repeater
  - Master/Detail Filtering With a DropDownList (C#)
  - Master/Detail Filtering Across Two Pages (C#)
  - Master/Detail Using a Bulleted List of Master Records with a Details DataList (C#)
  - Master/Detail Filtering With a DropDownList (VB)
  - Master/Detail Filtering Across Two Pages (VB)
  - Master/Detail Using a Bulleted List of Master Records with a Details DataList (VB)
- Editing and Deleting Data Through the DataList

  - Overview of Editing and Deleting Data in the DataList (C#)
  - Performing Batch Updates (C#)
  - Handling BLL- and DAL-Level Exceptions (C#)
  - Adding Validation Controls to the DataList's Editing Interface (C#)
  - Customizing the DataList's Editing Interface (C#)
  - Overview of Editing and Deleting Data in the DataList (VB)
  - Performing Batch Updates (VB)
  - Handling BLL- and DAL-Level Exceptions (VB)
  - Adding Validation Controls to the DataList's Editing Interface (VB)
  - Customizing the DataList's Editing Interface (VB)
- Paging and Sorting with the DataList and Repeater
  - Paging Report Data in a DataList or Repeater Control (C#)
  - Sorting Data in a DataList or Repeater Control (C#)
  - Paging Report Data in a DataList or Repeater Control (VB)
  - Sorting Data in a DataList or Repeater Control (VB)
- Custom Button Actions with the DataList and Repeater
  - Custom Buttons in the DataList and Repeater (C#)
  - Custom Buttons in the DataList and Repeater (VB)
- Accessing the Database Directly from an ASP.NET Page
  - Querying Data with the SqlDataSource Control (C#)
  - Using Parameterized Queries with the SqlDataSource (C#)
  - Inserting, Updating, and Deleting Data with the SqlDataSource (C#)
  - Implementing Optimistic Concurrency with the SqlDataSource (C#)
  - Querying Data with the SqlDataSource Control (VB)
  - Using Parameterized Queries with the SqlDataSource (VB)
  - Inserting, Updating, and Deleting Data with the SqlDataSource (VB)
  - Implementing Optimistic Concurrency with the SqlDataSource (VB)
- Enhancing the GridView
  - Adding a GridView Column of Radio Buttons (C#)
  - Adding a GridView Column of Checkboxes (C#)
  - Inserting a New Record from the GridView's Footer (C#)
  - Adding a GridView Column of Radio Buttons (VB)
  - Adding a GridView Column of Checkboxes (VB)
  - Inserting a New Record from the GridView's Footer (VB)
- Working with Binary Files
  - Uploading Files (C#)
  - Displaying Binary Data in the Data Web Controls (C#)
  - Including a File Upload Option When Adding a New Record (C#)
  - Updating and Deleting Existing Binary Data (C#)
  - Uploading Files (VB)
  - Displaying Binary Data in the Data Web Controls (VB)
  - Including a File Upload Option When Adding a New Record (VB)
  - Updating and Deleting Existing Binary Data (VB)
- Caching Data

  - Caching Data with the ObjectDataSource (C#)
  - Caching Data in the Architecture (C#)
  - Caching Data at Application Startup (C#)
  - Using SQL Cache Dependencies (C#)
  - Caching Data with the ObjectDataSource (VB)
  - Caching Data in the Architecture (VB)
  - Caching Data at Application Startup (VB)
  - Using SQL Cache Dependencies (VB)
- Database-Driven Site Maps
  - Building a Custom Database-Driven Site Map Provider (C#)
  - Building a Custom Database-Driven Site Map Provider (VB)
- Working with Batched Data
  - Wrapping Database Modifications within a Transaction (C#)
  - Batch Updating (C#)
  - Batch Deleting (C#)
  - Batch Inserting (C#)
  - Wrapping Database Modifications within a Transaction (VB)
  - Batch Updating (VB)
  - Batch Deleting (VB)
  - Batch Inserting (VB)
- Advanced Data Access Scenarios
  - Creating New Stored Procedures for the Typed DataSet's TableAdapters (C#)
  - Using Existing Stored Procedures for the Typed DataSet's TableAdapters (C#)
  - Updating the TableAdapter to Use JOINs (C#)
  - Adding Additional DataTable Columns (C#)
  - Working with Computed Columns (C#)
  - Configuring the Data Access Layer's Connection- and Command-Level Settings (C#)
  - Protecting Connection Strings and Other Configuration Information (C#)
  - Debugging Stored Procedures (C#)
  - Creating Stored Procedures and User-Defined Functions with Managed Code (C#)
  - Creating New Stored Procedures for the Typed DataSet's TableAdapters (VB)
  - Using Existing Stored Procedures for the Typed DataSet's TableAdapters (VB)
  - Updating the TableAdapter to Use JOINs (VB)
  - Adding Additional DataTable Columns (VB)
  - Working with Computed Columns (VB)
  - Configuring the Data Access Layer's Connection- and Command-Level Settings (VB)
  - Protecting Connection Strings and Other Configuration Information (VB)
  - Debugging Stored Procedures (VB)
  - Creating Stored Procedures and User-Defined Functions with Managed Code (VB)

Introduction

2/20/2020 • 2 minutes to read • Edit Online

This introductory set of tutorials explains how to create a Data Access Layer (DAL), a Business Logic Layer (BLL), and a consistent look across your site.

- Creating a Data Access Layer (C#)
- Creating a Business Logic Layer (C#)
- Master Pages and Site Navigation (C#)
- Creating a Data Access Layer (VB)
- Creating a Business Logic Layer (VB)
- Master Pages and Site Navigation (VB)

Creating a Data Access Layer (C#)

2/20/2020 • 33 minutes to read • Edit Online

by Scott Mitchell

Download PDF

> In this tutorial we'll start from the very beginning and create the Data Access Layer (DAL), using typed DataSets, to access the information in a database.

Introduction

As web developers, our lives revolve around working with data. We create databases to store the data, code to retrieve and modify it, and web pages to collect and summarize it. This is the first tutorial in a lengthy series that will explore techniques for implementing these common patterns in ASP.NET 2.0. We'll start with creating a software architecture composed of a Data Access Layer (DAL) using Typed DataSets, a Business Logic Layer (BLL) that enforces custom business rules, and a presentation layer composed of ASP.NET pages that share a common page layout. Once this backend groundwork has been laid, we'll move into reporting, showing how to display, summarize, collect, and validate data from a web application. These tutorials are geared to be concise and provide step-by-step instructions with plenty of screen shots to walk you through the process visually. Each tutorial is available in C# and Visual Basic versions and includes a download of the complete code used. (This first tutorial is quite lengthy, but the rest are presented in much more digestible chunks.)

For these tutorials we'll be using a Microsoft SQL Server 2005 Express Edition version of the Northwind database placed in the **App_Data** directory. In addition to the database file, the **App_Data** folder also contains the SQL scripts for creating the database, in case you want to use a different database version. These scripts can be also be downloaded directly from Microsoft, if you'd prefer. If you use a different SQL Server version of the Northwind database, you will need to update the **NORTHWNDConnectionString** setting in the application's **Web.config** file. The web application was built using Visual Studio 2005 Professional Edition as a file system-based Web site project. However, all of the tutorials will work equally well with the free version of Visual Studio 2005, Visual Web Developer.

In this tutorial we'll start from the very beginning and create the Data Access Layer (DAL), followed by creating the Business Logic Layer (BLL) in the second tutorial, and working on page layout and navigation in the third. The tutorials after the third one will build upon the foundation laid in the first three. We've got a lot to cover in this first tutorial, so fire up Visual Studio and let's get started!

Step 1: Creating a Web Project and Connecting to the Database

Before we can create our Data Access Layer (DAL), we first need to create a web site and setup our database. Start by creating a new file system-based ASP.NET web site. To accomplish this, go to the File menu and choose New Web Site, displaying the New Web Site dialog box. Choose the ASP.NET Web Site template, set the Location drop-down list to File System, choose a folder to place the web site, and set the language to C#.

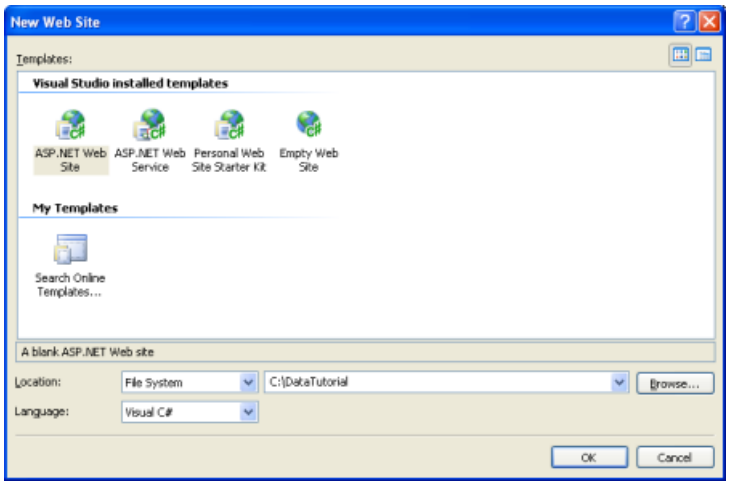


Figure 1: Create a New File System-Based Web Site (Click to view full-size image)

This will create a new web site with a **Default.aspx** ASP.NET page and an **App_Data** folder.

With the web site created, the next step is to add a reference to the database in Visual Studio's Server Explorer. By adding a database to the Server Explorer you can add tables, stored procedures, views, and so on all from within Visual Studio. You can also view table data or create your own queries either by hand or graphically via the Query Builder. Furthermore, when we build the Typed DataSets for the DAL we'll need to point Visual Studio to the database from which the Typed DataSets should be constructed. While we can provide this connection information at that point in time, Visual Studio automatically populates a drop-down list of the databases already registered in the Server Explorer.

The steps for adding the Northwind database to the Server Explorer depend on whether you want to use the SQL Server 2005 Express Edition database in the **App_Data** folder or if you have a Microsoft SQL Server 2000 or 2005 database server setup that you want to use instead.

Using a Database in the App_Data Folder

If you do not have a SQL Server 2000 or 2005 database server to connect to, or you simply want to avoid having to add the database to a database server, you can use the SQL Server 2005 Express Edition version of the Northwind database that is located in the downloaded website's **App_Data** folder (**NORTHWND.MDF**).

A database placed in the **App_Data** folder is automatically added to the Server Explorer. Assuming you have SQL Server 2005 Express Edition installed on your machine you should see a node named NORTHWND.MDF in the Server Explorer, which you can expand and explore its tables, views, stored procedure, and so on (see Figure 2).

The **App_Data** folder can also hold Microsoft Access **.mdb** files, which, like their SQL Server counterparts, are automatically added to the Server Explorer. If you don't want to use any of the SQL Server options, you can always download a Microsoft Access version of the Northwind database file and drop into the **App_Data** directory. Keep in mind, however, that Access databases aren't as feature-rich as SQL Server, and aren't designed to be used in web site scenarios. Furthermore, a couple of the 35+ tutorials will utilize certain database-level features that aren't supported by Access.

Connecting to the Database in a Microsoft SQL Server 2000 or 2005 Database Server

Alternatively, you may connect to a Northwind database installed on a database server. If the database server does not already have the Northwind database installed, you first must add it to database server by running the installation script included in this tutorial's download or by downloading the SQL Server 2000 version of Northwind and installation script directly from Microsoft's web site.

Once you have the database installed, go to the Server Explorer in Visual Studio, right-click on the Data Connections node, and choose Add Connection. If you don't see the Server Explorer go to the View / Server Explorer, or hit Ctrl+Alt+S. This will bring up the Add Connection dialog box, where you can specify the server to connect to, the authentication information, and the database name. Once you have successfully configured the database connection information and clicked the OK button, the database will be added as a node underneath the Data Connections node. You can expand the database node to explore its tables, views, stored procedures, and so on.

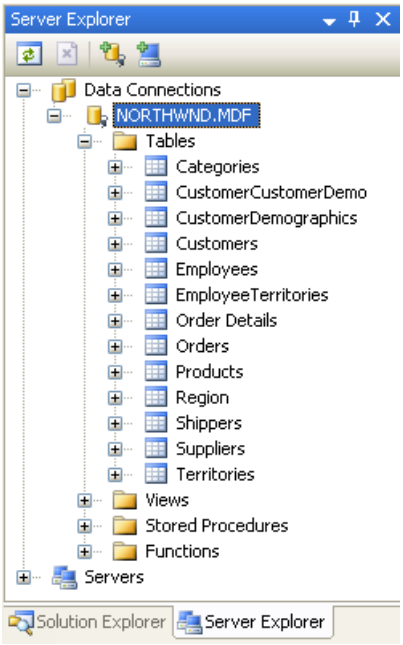


Figure 2: Add a Connection to Your Database Server's Northwind Database

Step 2: Creating the Data Access Layer

When working with data one option is to embed the data-specific logic directly into the presentation layer (in a web application, the ASP.NET pages make up the presentation layer). This may take the form of writing ADO.NET code in the ASP.NET page's code portion or using the SqlDataSource control from the markup portion. In either case, this approach tightly couples the data access logic with the presentation layer. The recommended approach, however, is to separate the data access logic from the presentation layer. This separate layer is referred to as the Data Access Layer, DAL for short, and is typically implemented as a separate Class Library project. The benefits of this layered architecture are well documented (see the "Further Readings" section at the end of this tutorial for information on these advantages) and is the approach we will take in this series.

All code that is specific to the underlying data source such as creating a connection to the database, issuing **SELECT**, **INSERT**, **UPDATE**, and **DELETE** commands, and so on should be located in the DAL. The presentation layer should not contain any references to such data access code, but should instead make calls into the DAL for any and all data requests. Data Access Layers typically contain methods for accessing the underlying database data. The Northwind database, for example, has **Products** and **Categories** tables that record the products for sale and the categories to which they belong. In our DAL we will have methods like:

- **GetCategories(),** which will return information about all of the categories
- **GetProducts()**, which will return information about all of the products
- **GetProductsByCategoryID(*categoryID*)**, which will return all products that belong to a specified category
- **GetProductByProductID(*productID*)**, which will return information about a particular product

These methods, when invoked, will connect to the database, issue the appropriate query, and return the results. How we return these results is important. These methods could simply return a DataSet or DataReader

populated by the database query, but ideally these results should be returned using *strongly-typed objects*. A strongly-typed object is one whose schema is rigidly defined at compile time, whereas the opposite, a loosely-typed object, is one whose schema is not known until runtime.

For example, the DataReader and the DataSet (by default) are loosely-typed objects since their schema is defined by the columns returned by the database query used to populate them. To access a particular column from a loosely-typed DataTable we need to use syntax like: *DataTable*.Rows[*index*]["*columnName*"]. The DataTable's loose typing in this example is exhibited by the fact that we need to access the column name using a string or ordinal index. A strongly-typed DataTable, on the other hand, will have each of its columns implemented as properties, resulting in code that looks like: *DataTable*.Rows[*index*].*columnName*.

To return strongly-typed objects, developers can either create their own custom business objects or use Typed DataSets. A business object is implemented by the developer as a class whose properties typically reflect the columns of the underlying database table the business object represents. A Typed DataSet is a class generated for you by Visual Studio based on a database schema and whose members are strongly-typed according to this schema. The Typed DataSet itself consists of classes that extend the ADO.NET DataSet, DataTable, and DataRow classes. In addition to strongly-typed DataTables, Typed DataSets now also include TableAdapters, which are classes with methods for populating the DataSet's DataTables and propagating modifications within the DataTables back to the database.

> **NOTE**
> For more information on the advantages and disadvantages of using Typed DataSets versus custom business objects, refer to Designing Data Tier Components and Passing Data Through Tiers.

We'll use strongly-typed DataSets for these tutorials' architecture. Figure 3 illustrates the workflow between the different layers of an application that uses Typed DataSets.

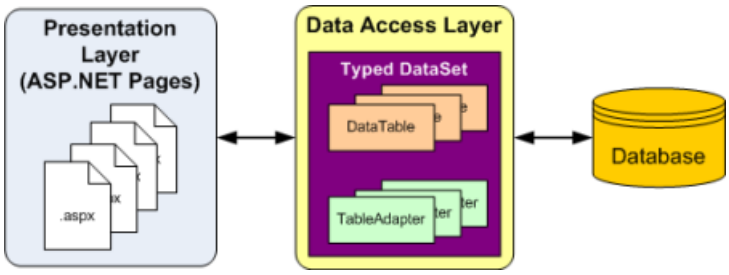


Figure 3: All Data Access Code is Relegated to the DAL (Click to view full-size image)

Creating a Typed DataSet and Table Adapter

To begin creating our DAL, we start by adding a Typed DataSet to our project. To accomplish this, right-click on the project node in the Solution Explorer and choose Add a New Item. Select the DataSet option from the list of templates and name it **Northwind.xsd**.

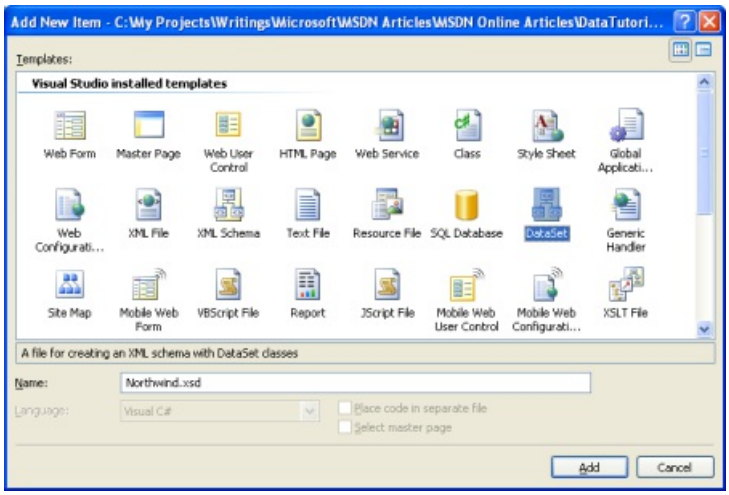


Figure 4: Choose to Add a New DataSet to Your Project (Click to view full-size image)

After clicking Add, when prompted to add the DataSet to the **App_Code** folder, choose Yes. The Designer for the Typed DataSet will then be displayed, and the TableAdapter Configuration Wizard will start, allowing you to add your first TableAdapter to the Typed DataSet.

A Typed DataSet serves as a strongly-typed collection of data; it is composed of strongly-typed DataTable instances, each of which is in turn composed of strongly-typed DataRow instances. We will create a strongly-typed DataTable for each of the underlying database tables that we need to work with in this tutorials series. Let's start with creating a DataTable for the **Products** table.

Keep in mind that strongly-typed DataTables do not include any information on how to access data from their underlying database table. In order to retrieve the data to populate the DataTable, we use a TableAdapter class, which functions as our Data Access Layer. For our **Products** DataTable, the TableAdapter will contain the methods **GetProducts()**, **GetProductByCategoryID(*categoryID*)**, and so on that we'll invoke from the presentation layer. The DataTable's role is to serve as the strongly-typed objects used to pass data between the layers.

The TableAdapter Configuration Wizard begins by prompting you to select which database to work with. The drop-down list shows those databases in the Server Explorer. If you did not add the Northwind database to the Server Explorer, you can click the New Connection button at this time to do so.

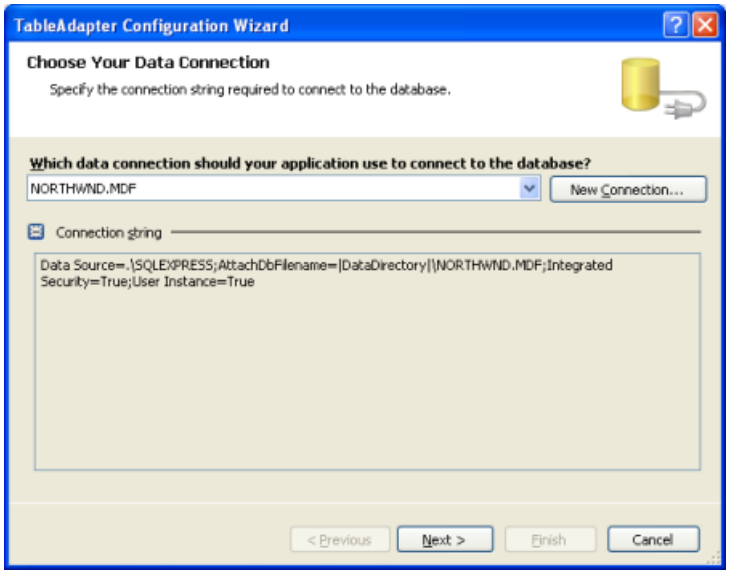


Figure 5: Choose the Northwind Database from the Drop-Down List (Click to view full-size image)

After selecting the database and clicking Next, you'll be asked if you want to save the connection string in the **Web.config** file. By saving the connection string you'll avoid having it hard coded in the TableAdapter classes, which simplifies things if the connection string information changes in the future. If you opt to save the

connection string in the configuration file it's placed in the **<connectionStrings>** section, which can be optionally encrypted for improved security or modified later through the new ASP.NET 2.0 Property Page within the IIS GUI Admin Tool, which is more ideal for administrators.

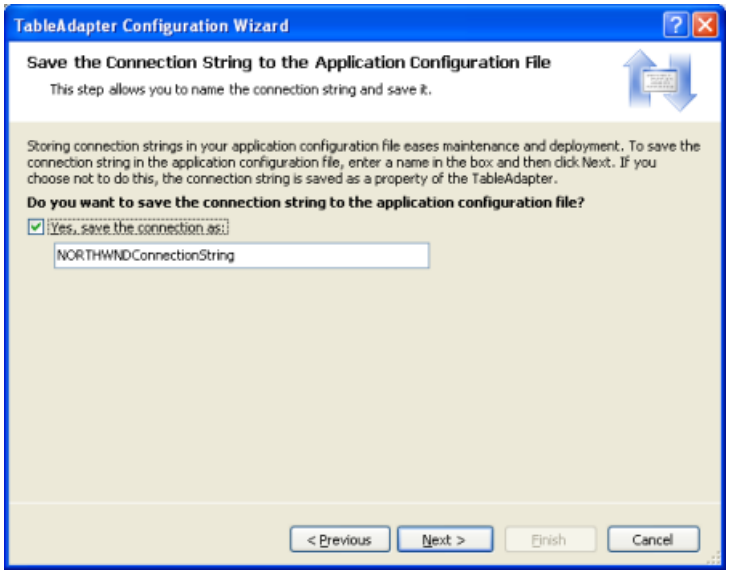


Figure 6: Save the Connection String to **Web.config** (Click to view full-size image)

Next, we need to define the schema for the first strongly-typed DataTable and provide the first method for our TableAdapter to use when populating the strongly-typed DataSet. These two steps are accomplished simultaneously by creating a query that returns the columns from the table that we want reflected in our DataTable. At the end of the wizard we'll give a method name to this query. Once that's been accomplished, this method can be invoked from our presentation layer. The method will execute the defined query and populate a strongly-typed DataTable.

To get started defining the SQL query we must first indicate how we want the TableAdapter to issue the query. We can use an ad-hoc SQL statement, create a new stored procedure, or use an existing stored procedure. For these tutorials we'll use ad-hoc SQL statements. Refer to Brian Noyes's article, Build a Data Access Layer with the Visual Studio 2005 DataSet Designer for an example of using stored procedures.

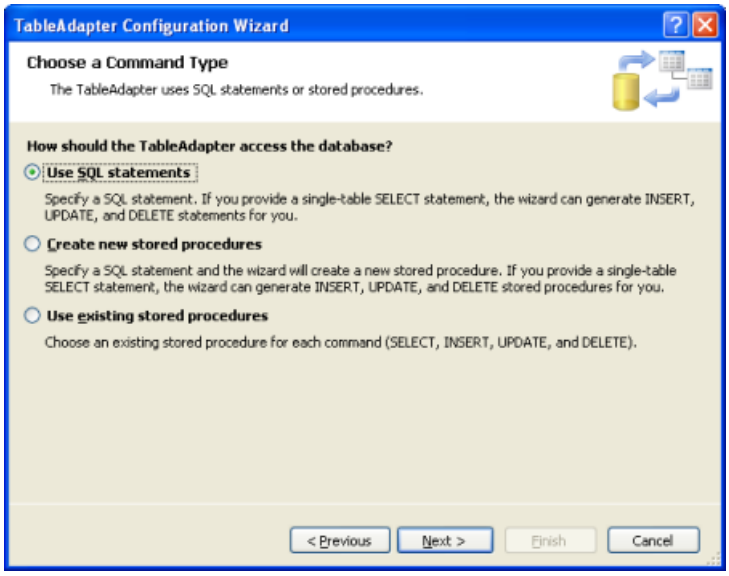


Figure 7: Query the Data Using an Ad-Hoc SQL Statement (Click to view full-size image)

At this point we can type in the SQL query by hand. When creating the first method in the TableAdapter you typically want to have the query return those columns that need to be expressed in the corresponding DataTable. We can accomplish this by creating a query that returns all columns and all rows from the **Products** table:

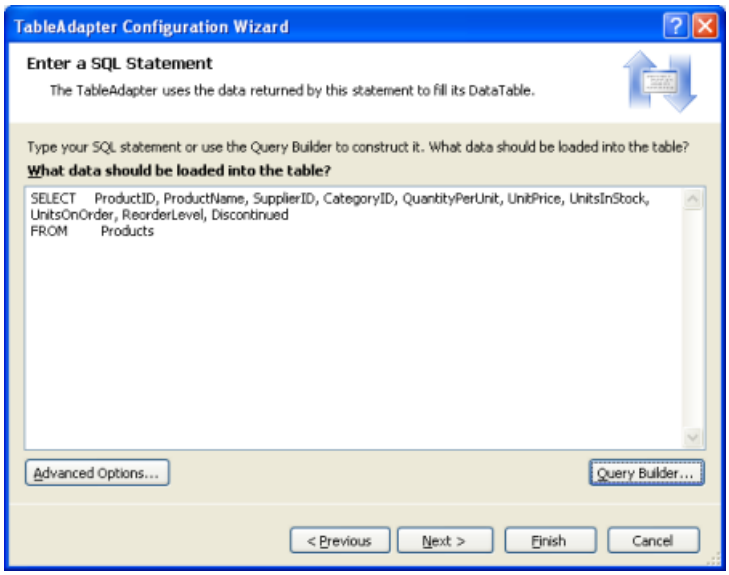


Figure 8: Enter the SQL Query Into the Textbox (Click to view full-size image)

Alternatively, use the Query Builder and graphically construct the query, as shown in Figure 9.

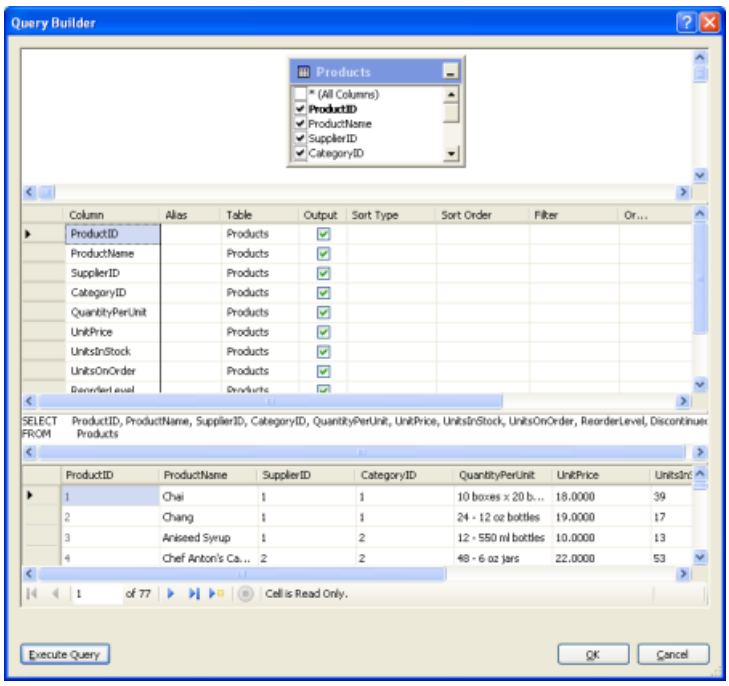


Figure 9: Create the Query Graphically, through the Query Editor (Click to view full-size image)

After creating the query, but before moving onto the next screen, click the Advanced Options button. In Web Site Projects, "Generate Insert, Update, and Delete statements" is the only advanced option selected by default; if you run this wizard from a Class Library or a Windows Project the "Use optimistic concurrency" option will also be selected. Leave the "Use optimistic concurrency" option unchecked for now. We'll examine optimistic concurrency in future tutorials.

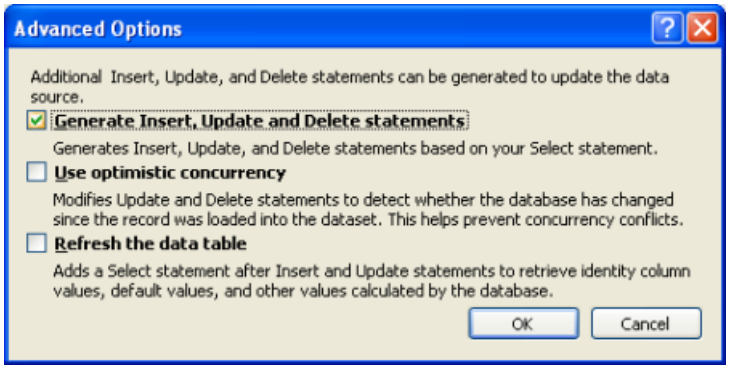


Figure 10: Select Only the Generate Insert, Update, and Delete statements Option (Click to view full-size image)

After verifying the advanced options, click Next to proceed to the final screen. Here we are asked to select which methods to add to the TableAdapter. There are two patterns for populating data:

- **Fill a DataTable** with this approach a method is created that takes in a DataTable as a parameter and populates it based on the results of the query. The ADO.NET DataAdapter class, for example, implements this pattern with its **Fill()** method.
- **Return a DataTable** with this approach the method creates and fills the DataTable for you and returns it as the methods return value.

You can have the TableAdapter implement one or both of these patterns. You can also rename the methods provided here. Let's leave both checkboxes checked, even though we'll only be using the latter pattern throughout these tutorials. Also, let's rename the rather generic **GetData** method to **GetProducts**.

If checked, the final checkbox, "GenerateDBDirectMethods," creates **Insert()**, **Update()**, and **Delete()** methods for the TableAdapter. If you leave this option unchecked, all updates will need to be done through the TableAdapter's sole **Update()** method, which takes in the Typed DataSet, a DataTable, a single DataRow, or an array of DataRows. (If you've unchecked the "Generate Insert, Update, and Delete statements" option from the advanced properties in Figure 9 this checkbox's setting will have no effect.) Let's leave this checkbox selected.

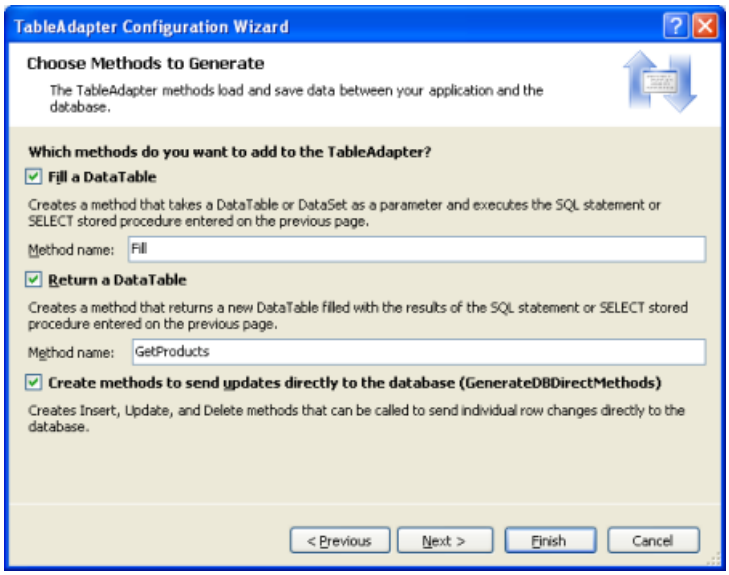


Figure 11: Change the Method Name from **GetData** to **GetProducts** (Click to view full-size image)

Complete the wizard by clicking Finish. After the wizard closes we are returned to the DataSet Designer which shows the DataTable we just created. You can see the list of columns in the **Products** DataTable (**ProductID**, **ProductName**, and so on), as well as the methods of the **ProductsTableAdapter** (**Fill()** and **GetProducts()**).

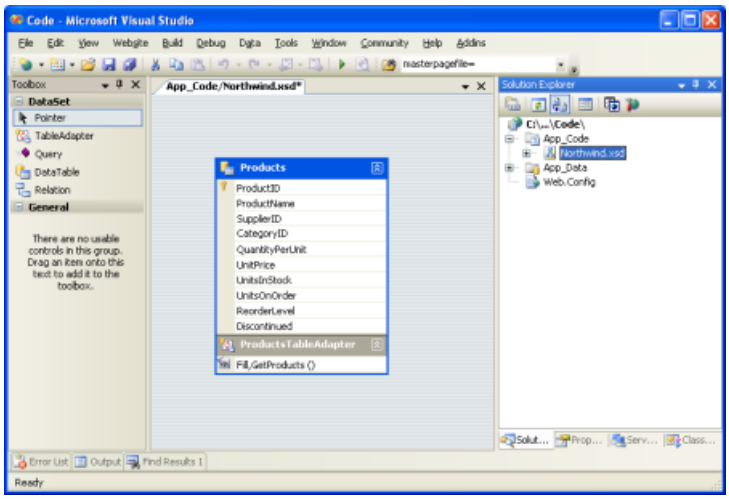


Figure 12: The **Products** DataTable and **ProductsTableAdapter** have been Added to the Typed DataSet (Click to view full-size image)

At this point we have a Typed DataSet with a single DataTable (**Northwind.Products**) and a strongly-typed DataAdapter class (**NorthwindTableAdapters.ProductsTableAdapter**) with a **GetProducts()** method. These objects can be used to access a list of all products from code like:

```
NorthwindTableAdapters.ProductsTableAdapter productsAdapter =
    new NorthwindTableAdapters.ProductsTableAdapter();
Northwind.ProductsDataTable products;
products = productsAdapter.GetProducts();
foreach (Northwind.ProductsRow productRow in products)
    Response.Write("Product: " + productRow.ProductName + "<br />");
```

This code did not require us to write one bit of data access-specific code. We did not have to instantiate any ADO.NET classes, we didn't have to refer to any connection strings, SQL queries, or stored procedures. Instead, the TableAdapter provides the low-level data access code for us.

Each object used in this example is also strongly-typed, allowing Visual Studio to provide IntelliSense and compile-time type checking. And best of all the DataTables returned by the TableAdapter can be bound to ASP.NET data Web controls, such as the GridView, DetailsView, DropDownList, CheckBoxList, and several others. The following example illustrates binding the DataTable returned by the **GetProducts()** method to a GridView in just a scant three lines of code within the **Page_Load** event handler.

AllProducts.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="AllProducts.aspx.cs"
    Inherits="AllProducts" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>View All Products in a GridView</title>
    <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <h2>
            All Products</h2>
        <p>
            <asp:GridView ID="GridView1" runat="server"
             CssClass="DataWebControlStyle">
               <HeaderStyle CssClass="HeaderStyle" />
               <AlternatingRowStyle CssClass="AlternatingRowStyle" />
            </asp:GridView>
             </p>
    </div>
    </form>
</body>
</html>
```

AllProducts.aspx.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using NorthwindTableAdapters;
public partial class AllProducts : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        ProductsTableAdapter productsAdapter = new
         ProductsTableAdapter();
        GridView1.DataSource = productsAdapter.GetProducts();
        GridView1.DataBind();
    }
}
```

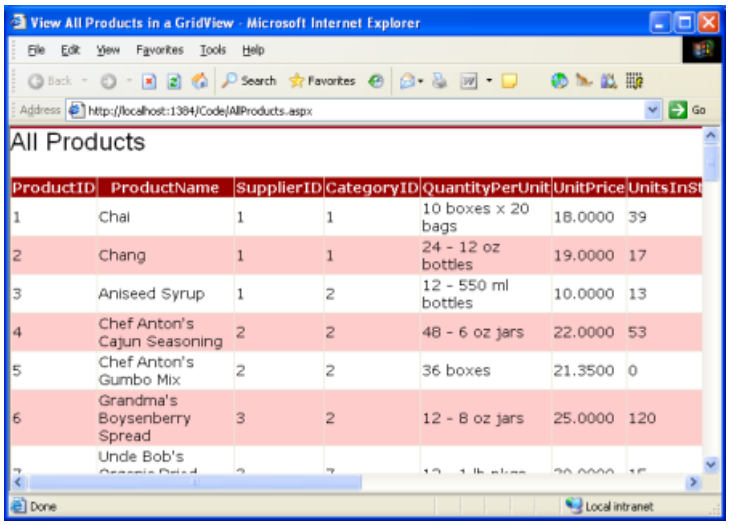


Figure 13: The List of Products is Displayed in a GridView (Click to view full-size image)

While this example required that we write three lines of code in our ASP.NET page's **Page_Load** event handler, in future tutorials we'll examine how to use the ObjectDataSource to declaratively retrieve the data from the DAL. With the ObjectDataSource we'll not have to write any code and will get paging and sorting support as well!

Step 3: Adding Parameterized Methods to the Data Access Layer

At this point our **ProductsTableAdapter** class has but one method, **GetProducts()**, which returns all of the products in the database. While being able to work with all products is definitely useful, there are times when we'll want to retrieve information about a specific product, or all products that belong to a particular category. To add such functionality to our Data Access Layer we can add parameterized methods to the TableAdapter.

Let's add the **GetProductsByCategoryID(*categoryID*)** method. To add a new method to the DAL, return to the DataSet Designer, right-click in the **ProductsTableAdapter** section, and choose Add Query.

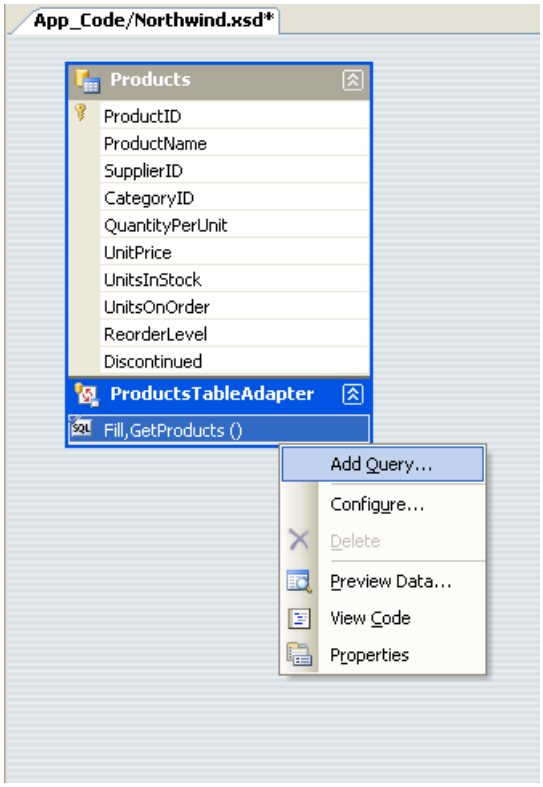


Figure 14: Right-Click on the TableAdapter and Choose Add Query

We are first prompted about whether we want to access the database using an ad-hoc SQL statement or a new

or existing stored procedure. Let's choose to use an ad-hoc SQL statement again. Next, we are asked what type of SQL query we'd like to use. Since we want to return all products that belong to a specified category, we want to write a **SELECT** statement which returns rows.

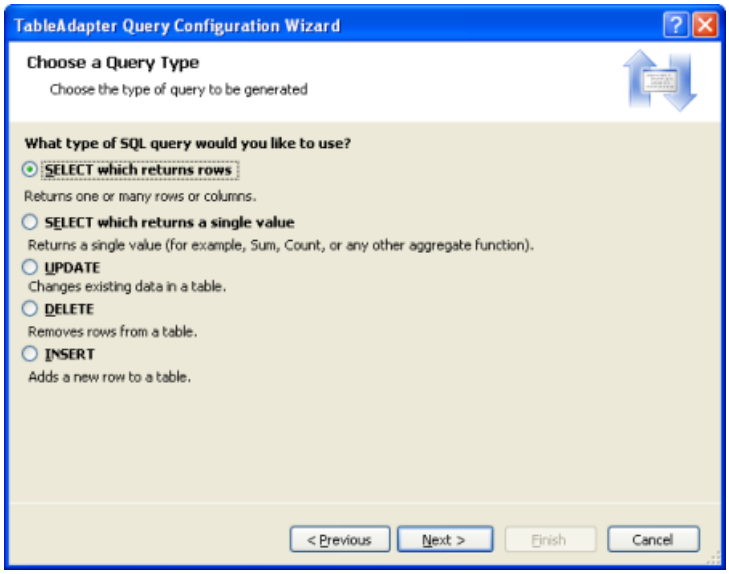


Figure 15: Choose to Create a **SELECT** Statement Which Returns Rows (Click to view full-size image)

The next step is to define the SQL query used to access the data. Since we want to return only those products that belong to a particular category, I use the same **SELECT** statement from **GetProducts()**, but add the following **WHERE** clause: **WHERE CategoryID = @CategoryID**. The **@CategoryID** parameter indicates to the TableAdapter wizard that the method we're creating will require an input parameter of the corresponding type (namely, a nullable integer).

Figure 16: Enter a Query to Only Return Products in a Specified Category (Click to view full-size image)

In the final step we can choose which data access patterns to use, as well as customize the names of the methods generated. For the Fill pattern, let's change the name to **FillByCategoryID** and for the return a DataTable return pattern (the **Get*X*** methods), let's use **GetProductsByCategoryID**.

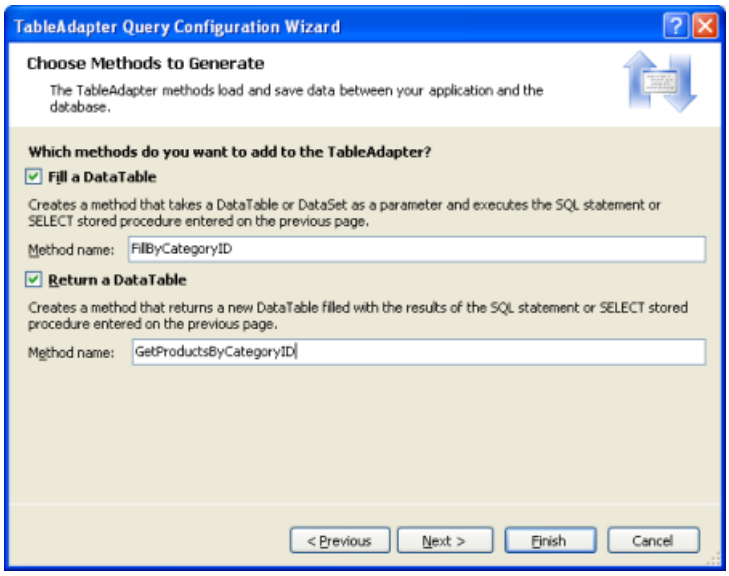


Figure 17: Choose the Names for the TableAdapter Methods (Click to view full-size image)

After completing the wizard, the DataSet Designer includes the new TableAdapter methods.

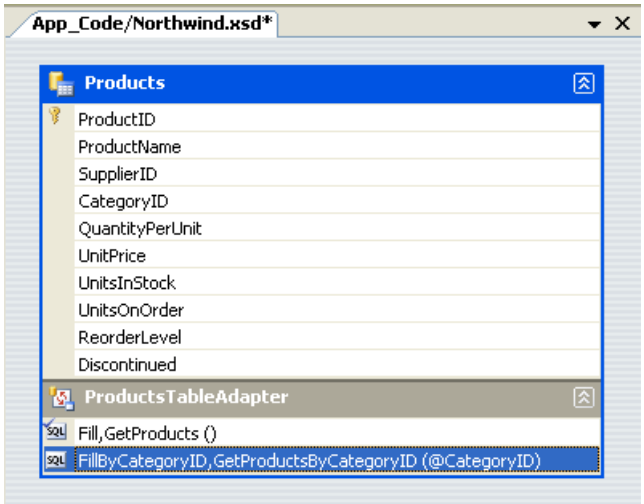


Figure 18: The Products Can Now be Queried by Category

Take a moment to add a **GetProductByProductID(*productID*)** method using the same technique.

These parameterized queries can be tested directly from the DataSet Designer. Right-click on the method in the TableAdapter and choose Preview Data. Next, enter the values to use for the parameters and click Preview.

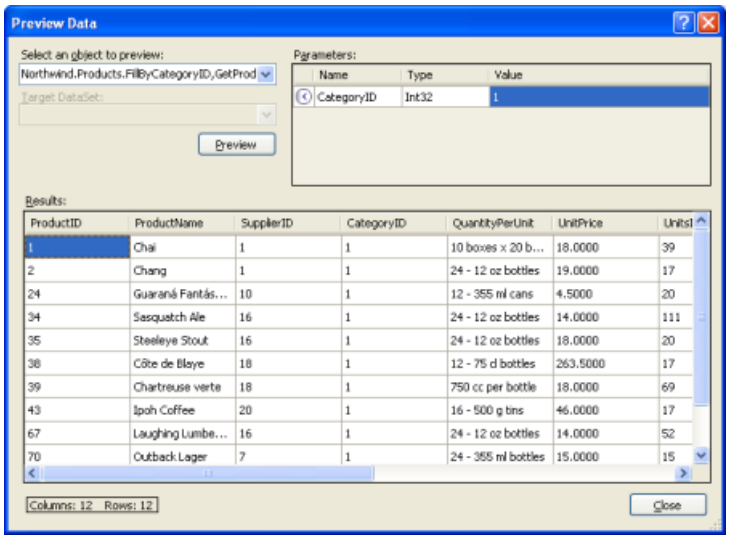


Figure 19: Those Products Belonging to the Beverages Category are Shown (Click to view full-size image)

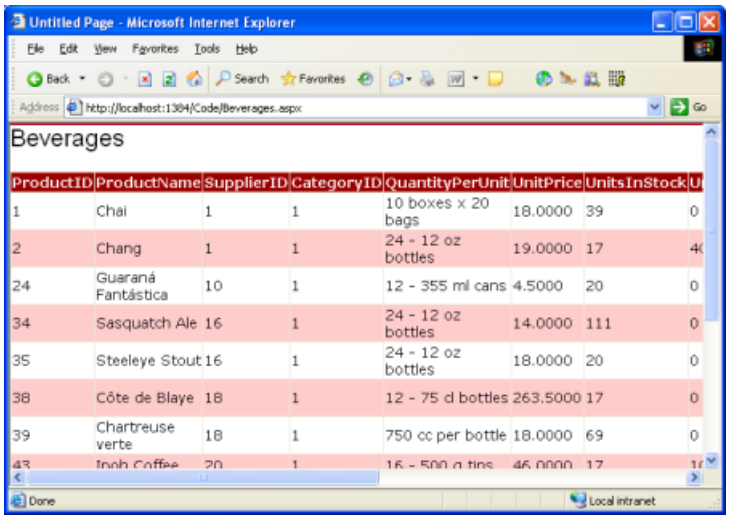
With the **GetProductsByCategoryID(*categoryID*)** method in our DAL, we can now create an ASP.NET page that displays only those products in a specified category. The following example shows all products that are in the Beverages category, which have a **CategoryID** of 1.

Beverages.asp

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Beverages.aspx.cs"
    Inherits="Beverages" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
    <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <h2>Beverages</h2>
        <p>
            <asp:GridView ID="GridView1" runat="server"
             CssClass="DataWebControlStyle">
               <HeaderStyle CssClass="HeaderStyle" />
               <AlternatingRowStyle CssClass="AlternatingRowStyle" />
            </asp:GridView>
             </p>
    </div>
    </form>
</body>
</html>
```

Beverages.aspx.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using NorthwindTableAdapters;
public partial class Beverages : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        ProductsTableAdapter productsAdapter = new
         ProductsTableAdapter();
        GridView1.DataSource =
          productsAdapter.GetProductsByCategoryID(1);
        GridView1.DataBind();
    }
}
```



ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	U
1	Chai	1	1	10 boxes x 20 bags	18.0000	39	0
2	Chang	1	1	24 - 12 oz bottles	19.0000	17	4
24	Guaraná Fantástica	10	1	12 - 355 ml cans	4.5000	20	0
34	Sasquatch Ale	16	1	24 - 12 oz bottles	14.0000	111	0
35	Steeleye Stout	16	1	24 - 12 oz bottles	18.0000	20	0
38	Côte de Blaye	18	1	12 - 75 cl bottles	263.5000	17	0
39	Chartreuse verte	18	1	750 cc per bottle	18.0000	69	0
43	Ipoh Coffee	20	1	16 - 500 g tins	46.0000	17	1

Figure 20: Those Products in the Beverages Category are Displayed (Click to view full-size image)

Step 4: Inserting, Updating, and Deleting Data

There are two patterns commonly used for inserting, updating, and deleting data. The first pattern, which I'll call the database direct pattern, involves creating methods that, when invoked, issue an **INSERT**, **UPDATE**, or **DELETE** command to the database that operates on a single database record. Such methods are typically passed in a series of scalar values (integers, strings, Booleans, DateTimes, and so on) that correspond to the values to insert, update, or delete. For example, with this pattern for the **Products** table the delete method would take in an integer parameter, indicating the **ProductID** of the record to delete, while the insert method would take in a string for the **ProductName**, a decimal for the **UnitPrice**, an integer for the **UnitsOnStock**, and so on.

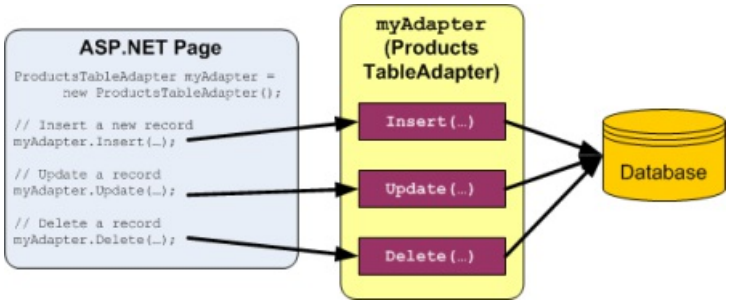


Figure 21: Each Insert, Update, and Delete Request is Sent to the Database Immediately (Click to view full-size image)

The other pattern, which I'll refer to as the batch update pattern, is to update an entire DataSet, DataTable, or collection of DataRows in one method call. With this pattern a developer deletes, inserts, and modifies the DataRows in a DataTable and then passes those DataRows or DataTable into an update method. This method then enumerates the DataRows passed in, determines whether or not they've been modified, added, or deleted (via the DataRow's RowState property value), and issues the appropriate database request for each record.

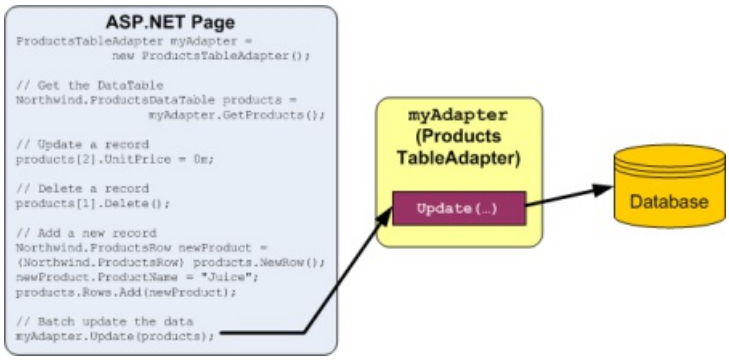


Figure 22: All Changes are Synchronized with the Database When the Update Method is Invoked (Click to view full-size image)

The TableAdapter uses the batch update pattern by default, but also supports the DB direct pattern. Since we selected the "Generate Insert, Update, and Delete statements" option from the Advanced Properties when creating our TableAdapter, the **ProductsTableAdapter** contains an **Update()** method, which implements the batch update pattern. Specifically, the TableAdapter contains an **Update()** method that can be passed the Typed DataSet, a strongly-typed DataTable, or one or more DataRows. If you left the "GenerateDBDirectMethods" checkbox checked when first creating the TableAdapter the DB direct pattern will also be implemented via **Insert()**, **Update()**, and **Delete()** methods.

Both data modification patterns use the TableAdapter's **InsertCommand**, **UpdateCommand**, and **DeleteCommand** properties to issue their **INSERT**, **UPDATE**, and **DELETE** commands to the database. You can inspect and modify the **InsertCommand**, **UpdateCommand**, and **DeleteCommand** properties by clicking on the TableAdapter in the DataSet Designer and then going to the Properties window. (Make sure you have selected the TableAdapter, and that the **ProductsTableAdapter** object is the one selected in the drop-down list in the Properties window.)

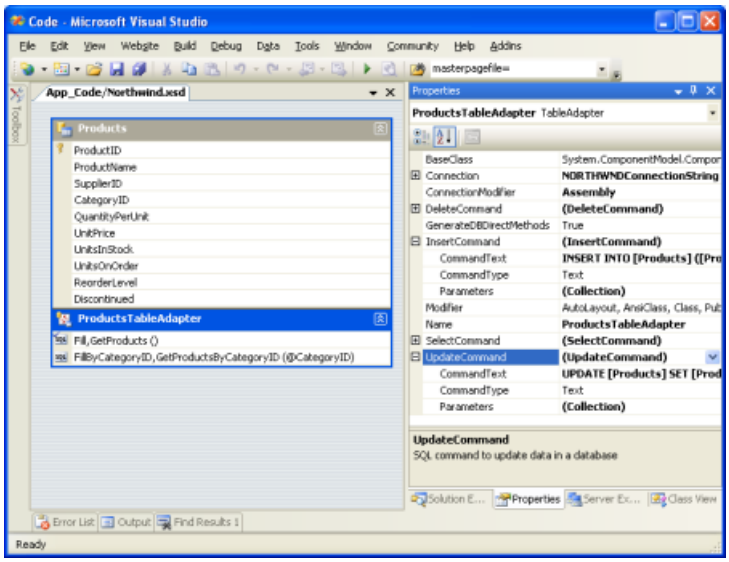


Figure 23: The TableAdapter has **InsertCommand**, **UpdateCommand**, and **DeleteCommand** Properties (Click to view full-size image)

To examine or modify any of these database command properties, click on the **CommandText** subproperty, which will bring up the Query Builder.

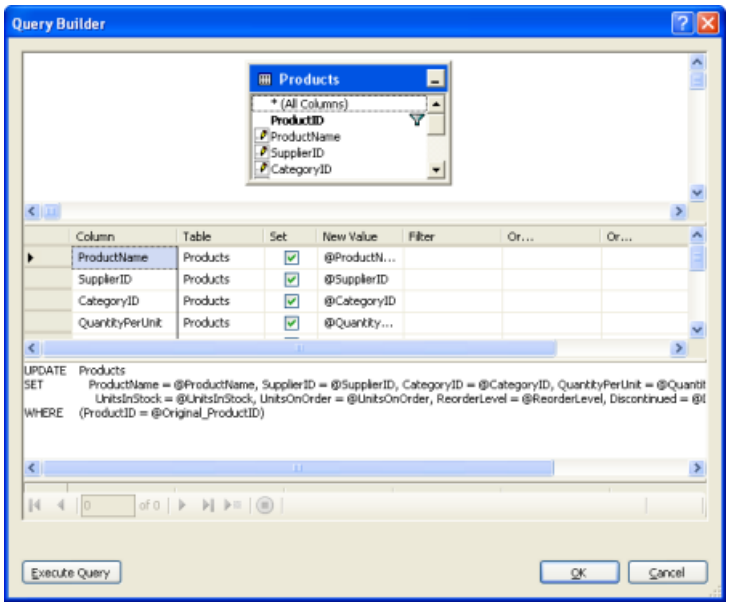


Figure 24: Configure the **INSERT**, **UPDATE**, and **DELETE** Statements in the Query Builder (Click to view full-size image)

The following code example shows how to use the batch update pattern to double the price of all products that are not discontinued and that have 25 units in stock or less:

```
NorthwindTableAdapters.ProductsTableAdapter productsAdapter =
  new NorthwindTableAdapters.ProductsTableAdapter();
// For each product, double its price if it is not discontinued and
// there are 25 items in stock or less
Northwind.ProductsDataTable products = productsAdapter.GetProducts();
foreach (Northwind.ProductsRow product in products)
   if (!product.Discontinued && product.UnitsInStock <= 25)
      product.UnitPrice *= 2;
// Update the products
productsAdapter.Update(products);
```

The code below illustrates how to use the DB direct pattern to programmatically delete a particular product, then update one, and then add a new one:

```
NorthwindTableAdapters.ProductsTableAdapter productsAdapter =
    new NorthwindTableAdapters.ProductsTableAdapter();
// Delete the product with ProductID 3
productsAdapter.Delete(3);
// Update Chai (ProductID of 1), setting the UnitsOnOrder to 15
productsAdapter.Update("Chai", 1, 1, "10 boxes x 20 bags",
  18.0m, 39, 15, 10, false, 1);
// Add a new product
productsAdapter.Insert("New Product", 1, 1,
  "12 tins per carton", 14.95m, 15, 0, 10, false);
```

Creating Custom Insert, Update, and Delete Methods

The **Insert()**, **Update()**, and **Delete()** methods created by the DB direct method can be a bit cumbersome, especially for tables with many columns. Looking at the previous code example, without IntelliSense's help it's not particularly clear what **Products** table column maps to each input parameter to the **Update()** and **Insert()** methods. There may be times when we only want to update a single column or two, or want a customized **Insert()** method that will, perhaps, return the value of the newly inserted record's **IDENTITY** (auto-increment) field.

To create such a custom method, return to the DataSet Designer. Right-click on the TableAdapter and choose Add Query, returning to the TableAdapter wizard. On the second screen we can indicate the type of query to create. Let's create a method that adds a new product and then returns the value of the newly added record's **ProductID**. Therefore, opt to create an **INSERT** query.

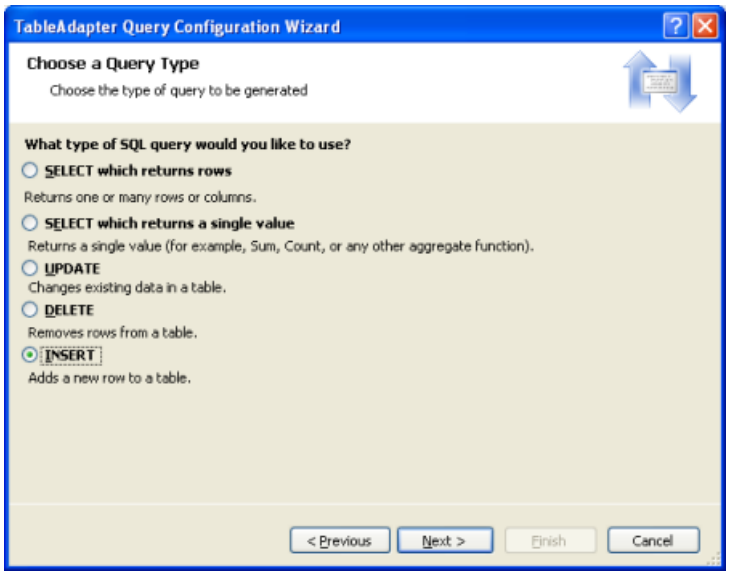


Figure 25: Create a Method to Add a New Row to the **Products** Table (Click to view full-size image)

On the next screen the **InsertCommand**'s **CommandText** appears. Augment this query by adding **SELECT SCOPE_IDENTITY()** at the end of the query, which will return the last identity value inserted into an **IDENTITY** column in the same scope. (See the technical documentation for more information about **SCOPE_IDENTITY()** and why you probably want to use SCOPE_IDENTITY() in lieu of @@IDENTITY.) Make sure that you end the **INSERT** statement with a semi-colon before adding the **SELECT** statement.

Figure 26: Augment the Query to Return the **SCOPE_IDENTITY()** Value (Click to view full-size image)

Finally, name the new method **InsertProduct**.

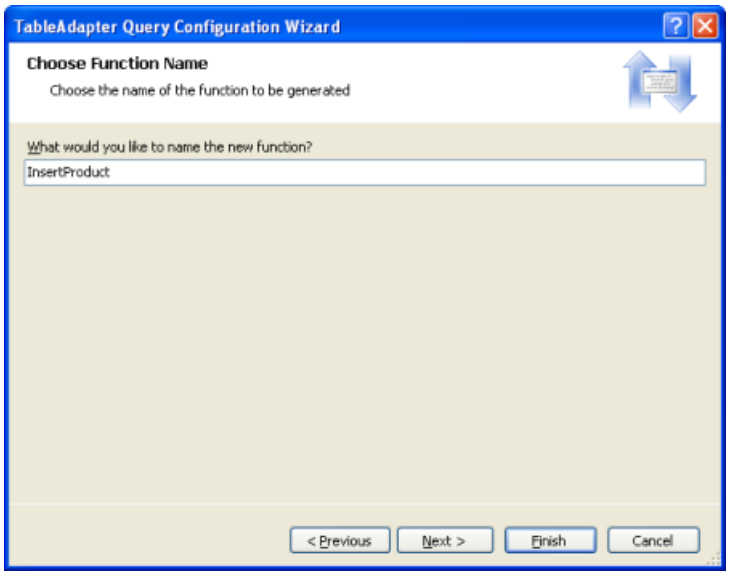


Figure 27: Set the New Method Name to **InsertProduct** (Click to view full-size image)

When you return to the DataSet Designer you'll see that the **ProductsTableAdapter** contains a new method, **InsertProduct**. If this new method doesn't have a parameter for each column in the **Products** table, chances are you forgot to terminate the **INSERT** statement with a semi-colon. Configure the **InsertProduct** method and ensure you have a semi-colon delimiting the **INSERT** and **SELECT** statements.

By default, insert methods issue non-query methods, meaning that they return the number of affected rows. However, we want the **InsertProduct** method to return the value returned by the query, not the number of rows affected. To accomplish this, adjust the **InsertProduct** method's **ExecuteMode** property to **Scalar**.

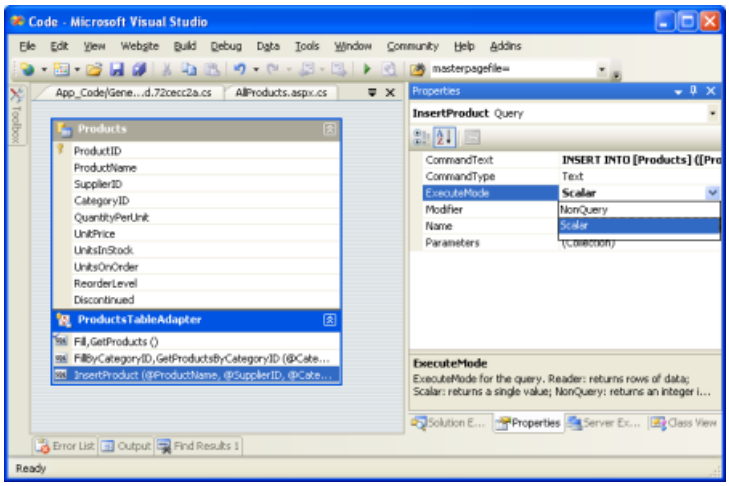


Figure 28: Change the **ExecuteMode** Property to **Scalar** (Click to view full-size image)

The following code shows this new **InsertProduct** method in action:

```
NorthwindTableAdapters.ProductsTableAdapter productsAdapter =
    new NorthwindTableAdapters.ProductsTableAdapter();
// Add a new product
int new_productID = Convert.ToInt32(productsAdapter.InsertProduct
    ("New Product", 1, 1, "12 tins per carton", 14.95m, 10, 0, 10, false));
// On second thought, delete the product
productsAdapter.Delete(new_productID);
```

Step 5: Completing the Data Access Layer

Note that the **ProductsTableAdapters** class returns the **CategoryID** and **SupplierID** values from the **Products** table, but doesn't include the **CategoryName** column from the **Categories** table or the

CompanyName column from the **Suppliers** table, although these are likely the columns we want to display when showing product information. We can augment the TableAdapter's initial method, **GetProducts()**, to include both the **CategoryName** and **CompanyName** column values, which will update the strongly-typed DataTable to include these new columns as well.

This can present a problem, however, as the TableAdapter's methods for inserting, updating, and deleting data are based off of this initial method. Fortunately, the auto-generated methods for inserting, updating, and deleting are not affected by subqueries in the **SELECT** clause. By taking care to add our queries to **Categories** and **Suppliers** as subqueries, rather than **JOIN** s, we'll avoid having to rework those methods for modifying data. Right-click on the **GetProducts()** method in the **ProductsTableAdapter** and choose Configure. Then, adjust the **SELECT** clause so that it looks like:

```
SELECT     ProductID, ProductName, SupplierID, CategoryID,
QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued,
(SELECT CategoryName FROM Categories
WHERE Categories.CategoryID = Products.CategoryID) as CategoryName,
(SELECT CompanyName FROM Suppliers
WHERE Suppliers.SupplierID = Products.SupplierID) as SupplierName
FROM         Products
```

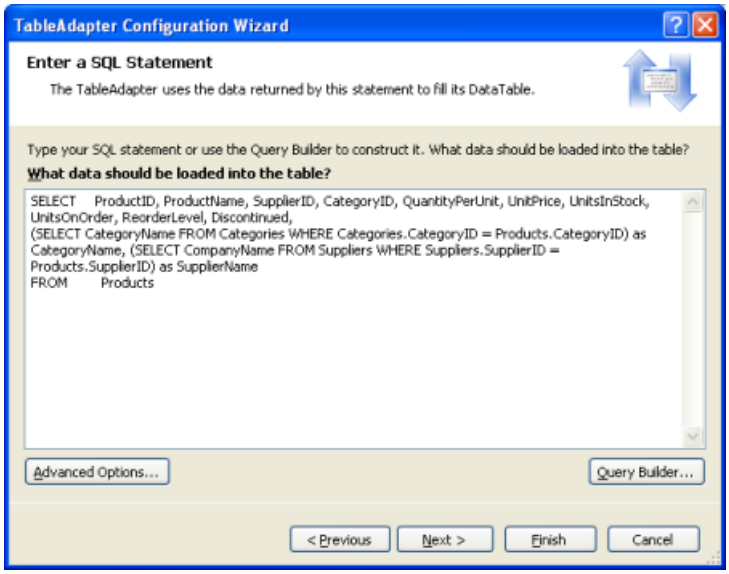


Figure 29: Update the **SELECT** Statement for the **GetProducts()** Method (Click to view full-size image)

After updating the **GetProducts()** method to use this new query the DataTable will include two new columns: **CategoryName** and **SupplierName**.

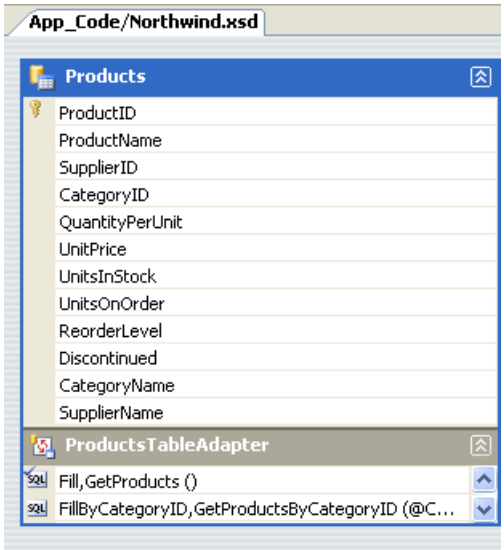


Figure 30: The **Products** DataTable has Two New Columns

Take a moment to update the **SELECT** clause in the **GetProductsByCategoryID(*categoryID*)** method as well.

If you update the **GetProducts() SELECT** using **JOIN** syntax the DataSet Designer won't be able to auto-generate the methods for inserting, updating, and deleting database data using the DB direct pattern. Instead, you'll have to manually create them much like we did with the **InsertProduct** method earlier in this tutorial. Furthermore, you'll manually have to provide the **InsertCommand**, **UpdateCommand**, and **DeleteCommand** property values if you want to use the batch updating pattern.

Adding the Remaining TableAdapters

Up until now, we've only looked at working with a single TableAdapter for a single database table. However, the Northwind database contains several related tables that we'll need to work with in our web application. A Typed DataSet can contain multiple, related DataTables. Therefore, to complete our DAL we need to add DataTables for the other tables we'll be using in these tutorials. To add a new TableAdapter to a Typed DataSet, open the DataSet Designer, right-click in the Designer, and choose Add / TableAdapter. This will create a new DataTable and TableAdapter and walk you through the wizard we examined earlier in this tutorial.

Take a few minutes to create the following TableAdapters and methods using the following queries. Note that the queries in the **ProductsTableAdapter** include the subqueries to grab each product's category and supplier names. Additionally, if you've been following along, you've already added the **ProductsTableAdapter** class's **GetProducts()** and **GetProductsByCategoryID(*categoryID*)** methods.

- **ProductsTableAdapter**
  - **GetProducts**:

```
SELECT     ProductID, ProductName, SupplierID,
CategoryID, QuantityPerUnit, UnitPrice, UnitsInStock,
UnitsOnOrder, ReorderLevel, Discontinued,
(SELECT CategoryName FROM Categories WHERE
Categories.CategoryID = Products.CategoryID) as
CategoryName, (SELECT CompanyName FROM Suppliers
WHERE Suppliers.SupplierID = Products.SupplierID)
as SupplierName
FROM         Products
```

  - **GetProductsByCategoryID**:

```
SELECT     ProductID, ProductName, SupplierID, CategoryID,
QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
ReorderLevel, Discontinued, (SELECT CategoryName
FROM Categories WHERE Categories.CategoryID =
Products.CategoryID) as CategoryName,
(SELECT CompanyName FROM Suppliers WHERE
Suppliers.SupplierID = Products.SupplierID)
as SupplierName
FROM         Products
WHERE      CategoryID = @CategoryID
```

  - **GetProductsBySupplierID**:

```
SELECT     ProductID, ProductName, SupplierID, CategoryID,
QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
ReorderLevel, Discontinued, (SELECT CategoryName
FROM Categories WHERE Categories.CategoryID =
Products.CategoryID) as CategoryName,
(SELECT CompanyName FROM Suppliers WHERE
Suppliers.SupplierID = Products.SupplierID) as SupplierName
FROM         Products
WHERE SupplierID = @SupplierID
```

- GetProductByProductID:

```
SELECT     ProductID, ProductName, SupplierID, CategoryID,
QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
ReorderLevel, Discontinued, (SELECT CategoryName
FROM Categories WHERE Categories.CategoryID =
Products.CategoryID) as CategoryName,
(SELECT CompanyName FROM Suppliers WHERE Suppliers.SupplierID = Products.SupplierID)
as SupplierName
FROM         Products
WHERE ProductID = @ProductID
```

- CategoriesTableAdapter
  - GetCategories:

```
SELECT     CategoryID, CategoryName, Description
FROM         Categories
```

  - GetCategoryByCategoryID:

```
SELECT     CategoryID, CategoryName, Description
FROM         Categories
WHERE CategoryID = @CategoryID
```

- SuppliersTableAdapter
  - GetSuppliers:

```
SELECT     SupplierID, CompanyName, Address,
City, Country, Phone
FROM         Suppliers
```

  - GetSuppliersByCountry:

```
SELECT     SupplierID, CompanyName, Address,
City, Country, Phone
FROM         Suppliers
WHERE Country = @Country
```

  - GetSupplierBySupplierID:

```
SELECT     SupplierID, CompanyName, Address,
City, Country, Phone
FROM         Suppliers
WHERE SupplierID = @SupplierID
```

- **EmployeesTableAdapter**
  - **GetEmployees**:

```
SELECT     EmployeeID, LastName, FirstName, Title,
HireDate, ReportsTo, Country
FROM         Employees
```

  - **GetEmployeesByManager**:

```
SELECT     EmployeeID, LastName, FirstName, Title,
HireDate, ReportsTo, Country
FROM         Employees
WHERE ReportsTo = @ManagerID
```

  - **GetEmployeeByEmployeeID**:

```
SELECT     EmployeeID, LastName, FirstName, Title,
HireDate, ReportsTo, Country
FROM         Employees
WHERE EmployeeID = @EmployeeID
```

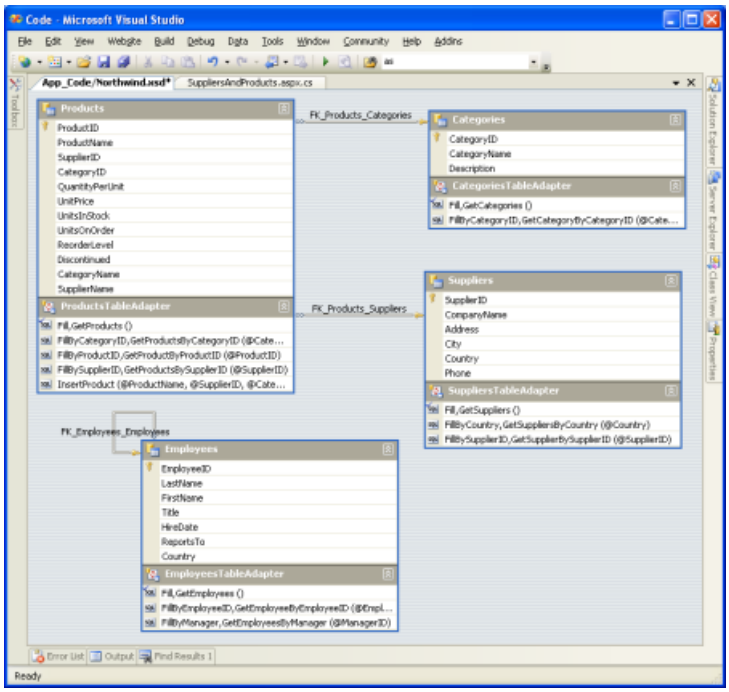


Figure 31: The DataSet Designer After the Four TableAdapters Have Been Added (Click to view full-size image)

Adding Custom Code to the DAL

The TableAdapters and DataTables added to the Typed DataSet are expressed as an XML Schema Definition file (**Northwind.xsd**). You can view this schema information by right-clicking on the **Northwind.xsd** file in the Solution Explorer and choosing View Code.

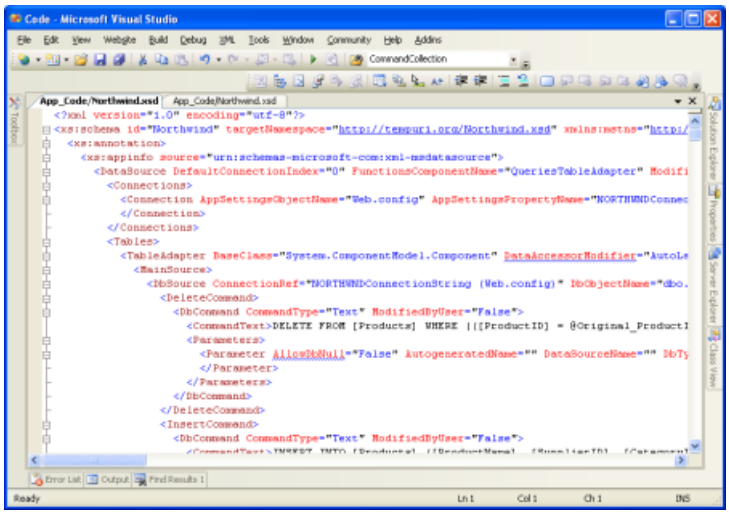


Figure 32: The XML Schema Definition (XSD) File for the Northwinds Typed DataSet (Click to view full-size image)

This schema information is translated into C# or Visual Basic code at design time when compiled or at runtime (if needed), at which point you can step through it with the debugger. To view this auto-generated code go to the Class View and drill down to the TableAdapter or Typed DataSet classes. If you don't see the Class View on your screen, go to the View menu and select it from there, or hit Ctrl+Shift+C. From the Class View you can see the properties, methods, and events of the Typed DataSet and TableAdapter classes. To view the code for a particular method, double-click the method name in the Class View or right-click on it and choose Go To Definition.

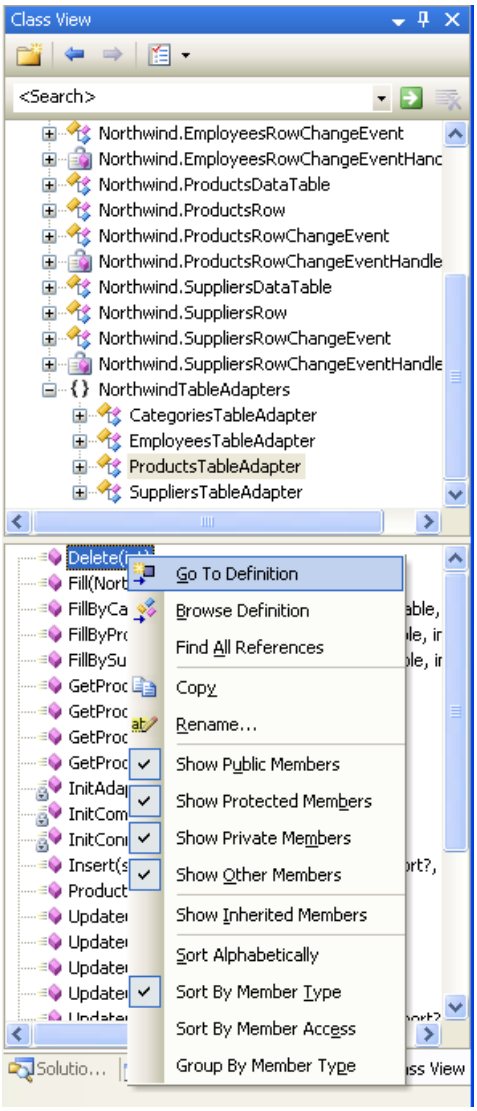


Figure 33: Inspect the Auto-Generated Code by Selecting Go To Definition from the Class View

While auto-generated code can be a great time saver, the code is often very generic and needs to be customized to meet the unique needs of an application. The risk of extending auto-generated code, though, is that the tool that generated the code might decide it's time to "regenerate" and overwrite your customizations. With .NET 2.0's new partial class concept, it's easy to split a class across multiple files. This enables us to add our own methods, properties, and events to the auto-generated classes without having to worry about Visual Studio overwriting our customizations.

To demonstrate how to customize the DAL, let's add a **GetProducts()** method to the **SuppliersRow** class. The **SuppliersRow** class represents a single record in the **Suppliers** table; each supplier can provider zero to many products, so **GetProducts()** will return those products of the specified supplier. To accomplish this create a new class file in the **App_Code** folder named **SuppliersRow.cs** and add the following code:

```
using System;
using System.Data;
using NorthwindTableAdapters;
public partial class Northwind
{
    public partial class SuppliersRow
    {
        public Northwind.ProductsDataTable GetProducts()
        {
            ProductsTableAdapter productsAdapter =
             new ProductsTableAdapter();
            return
              productsAdapter.GetProductsBySupplierID(this.SupplierID);
        }
    }
}
```

This partial class instructs the compiler that when building the **Northwind.SuppliersRow** class to include the **GetProducts()** method we just defined. If you build your project and then return to the Class View you'll see **GetProducts()** now listed as a method of **Northwind.SuppliersRow**.

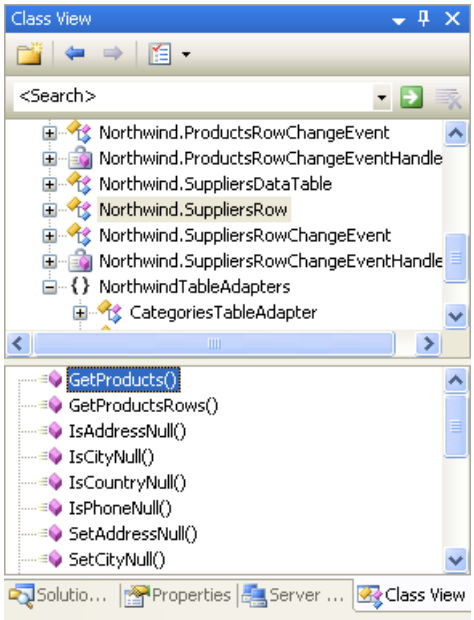


Figure 34: The **GetProducts()** Method is Now Part of the **Northwind.SuppliersRow** Class

The **GetProducts()** method can now be used to enumerate the set of products for a particular supplier, as the following code shows:

```
NorthwindTableAdapters.SuppliersTableAdapter suppliersAdapter =
    new NorthwindTableAdapters.SuppliersTableAdapter();
// Get all of the suppliers
Northwind.SuppliersDataTable suppliers =
  suppliersAdapter.GetSuppliers();
// Enumerate the suppliers
foreach (Northwind.SuppliersRow supplier in suppliers)
{
    Response.Write("Supplier: " + supplier.CompanyName);
    Response.Write("<ul>");
    // List the products for this supplier
    Northwind.ProductsDataTable products = supplier.GetProducts();
    foreach (Northwind.ProductsRow product in products)
        Response.Write("<li>" + product.ProductName + "</li>");
    Response.Write("</ul><p> </p>");
}
```

This data can also be displayed in any of ASP.NET's data Web controls. The following page uses a GridView control with two fields:

- A BoundField that displays the name of each supplier, and
- A TemplateField that contains a BulletedList control that is bound to the results returned by the **GetProducts()** method for each supplier.

We'll examine how to display such master-detail reports in future tutorials. For now, this example is designed to illustrate using the custom method added to the **Northwind.SuppliersRow** class.

SuppliersAndProducts.aspx

```
<%@ Page Language="C#" CodeFile="SuppliersAndProducts.aspx.cs"
    AutoEventWireup="true" Inherits="SuppliersAndProducts" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
    <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <h2>
            Suppliers and Their Products</h2>
        <p>
            <asp:GridView ID="GridView1" runat="server"
             AutoGenerateColumns="False"
             CssClass="DataWebControlStyle">
                <HeaderStyle CssClass="HeaderStyle" />
                <AlternatingRowStyle CssClass="AlternatingRowStyle" />
                <Columns>
                    <asp:BoundField DataField="CompanyName"
                      HeaderText="Supplier" />
                    <asp:TemplateField HeaderText="Products">
                        <ItemTemplate>
                            <asp:BulletedList ID="BulletedList1"
                             runat="server" DataSource="<%# ((Northwind.SuppliersRow)
((System.Data.DataRowView) Container.DataItem).Row).GetProducts() %>"
                                 DataTextField="ProductName">
                            </asp:BulletedList>
                        </ItemTemplate>
                    </asp:TemplateField>
                </Columns>
            </asp:GridView>
             </p>
    </div>
    </form>
</body>
</html>
```

SuppliersAndProducts.aspx.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using NorthwindTableAdapters;
public partial class SuppliersAndProducts : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        SuppliersTableAdapter suppliersAdapter = new
          SuppliersTableAdapter();
        GridView1.DataSource = suppliersAdapter.GetSuppliers();
        GridView1.DataBind();
    }
}
```

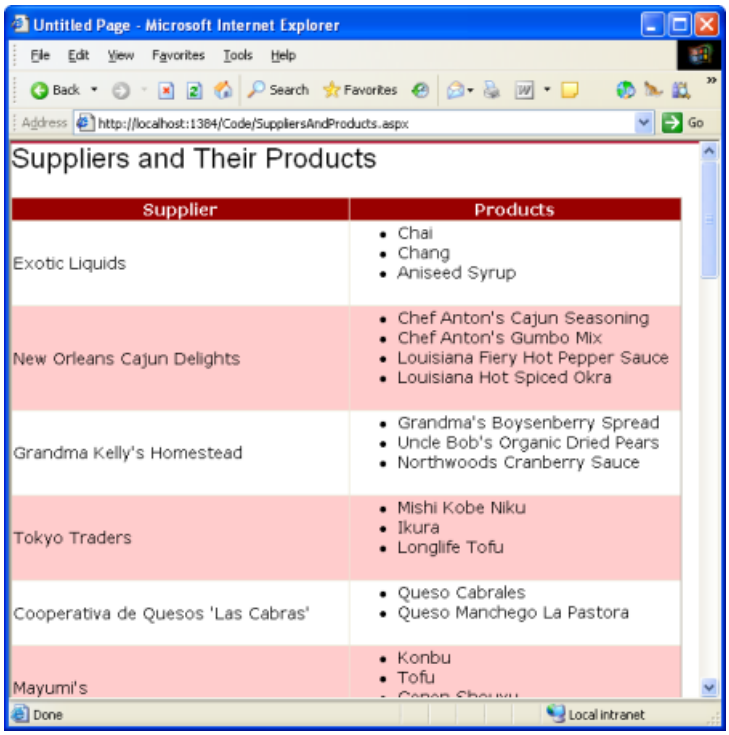


Figure 35: The Supplier's Company Name is Listed in the Left Column, Their Products in the Right (Click to view full-size image)

Summary

When building a web application creating the DAL should be one of your first steps, occurring before you start creating your presentation layer. With Visual Studio, creating a DAL based on Typed DataSets is a task that can be accomplished in 10-15 minutes without writing a line of code. The tutorials moving forward will build upon this DAL. In the next tutorial we'll define a number of business rules and see how to implement them in a separate Business Logic Layer.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Building a DAL using Strongly Typed TableAdapters and DataTables in VS 2005 and ASP.NET 2.0
- Designing Data Tier Components and Passing Data Through Tiers
- Build a Data Access Layer with the Visual Studio 2005 DataSet Designer
- Encrypting Configuration Information in ASP.NET 2.0 Applications
- TableAdapter Overview
- Working with a Typed DataSet
- Using Strongly-Typed Data Access in Visual Studio 2005 and ASP.NET 2.0
- How to Extend TableAdapter Methods
- Retrieving Scalar Data from a Stored Procedure

Video Training on Topics Contained in this Tutorial

- Data Access Layers in ASP.NET Applications
- How to Manually Bind a Dataset to a Datagrid
- How to Work with Datasets and Filters from an ASP Application

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Ron Green, Hilton Giesenow, Dennis Patterson, Liz Shulok, Abel Gomez, and Carlos Santos. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Creating a Business Logic Layer (C#)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll see how to centralize your business rules into a Business Logic Layer (BLL) that serves as an intermediary for data exchange between the presentation layer and the DAL.

Introduction

The Data Access Layer (DAL) created in the first tutorial cleanly separates the data access logic from the presentation logic. However, while the DAL cleanly separates the data access details from the presentation layer, it does not enforce any business rules that may apply. For example, for our application we may want to disallow the `CategoryID` or `SupplierID` fields of the `Products` table to be modified when the `Discontinued` field is set to 1, or we might want to enforce seniority rules, prohibiting situations in which an employee is managed by someone who was hired after them. Another common scenario is authorization perhaps only users in a particular role can delete products or can change the `UnitPrice` value.

In this tutorial we'll see how to centralize these business rules into a Business Logic Layer (BLL) that serves as an intermediary for data exchange between the presentation layer and the DAL. In a real-world application, the BLL should be implemented as a separate Class Library project; however, for these tutorials we'll implement the BLL as a series of classes in our `App_Code` folder in order to simplify the project structure. Figure 1 illustrates the architectural relationships among the presentation layer, BLL, and DAL.

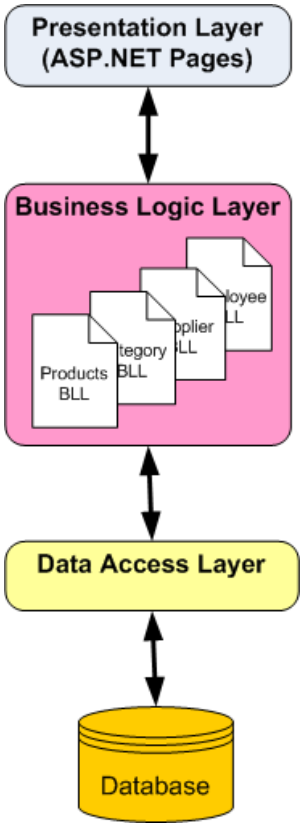


Figure 1: The BLL Separates the Presentation Layer from the Data Access Layer and Imposes Business Rules

Step 1: Creating the BLL Classes

Our BLL will be composed of four classes, one for each TableAdapter in the DAL; each of these BLL classes will have methods for retrieving, inserting, updating, and deleting from the respective TableAdapter in the DAL, applying the appropriate business rules.

To more cleanly separate the DAL- and BLL-related classes, let's create two subfolders in the `App_Code` folder, `DAL` and `BLL`. Simply right-click on the `App_Code` folder in the Solution Explorer and choose New Folder. After creating these two folders, move the Typed DataSet created in the first tutorial into the `DAL` subfolder.

Next, create the four BLL class files in the `BLL` subfolder. To accomplish this, right-click on the `BLL` subfolder, choose Add a New Item, and choose the Class template. Name the four classes `ProductsBLL`, `CategoriesBLL`, `SuppliersBLL`, and `EmployeesBLL`.

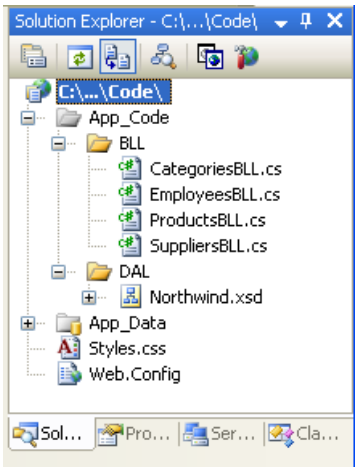


Figure 2: Add Four New Classes to the `App_Code` Folder

Next, let's add methods to each of the classes to simply wrap the methods defined for the TableAdapters from the first tutorial. For now, these methods will just call directly into the DAL; we'll return later to add any needed business logic.

> **NOTE**
>
> If you are using Visual Studio Standard Edition or above (that is, you're *not* using Visual Web Developer), you can optionally design your classes visually using the Class Designer. Refer to the Class Designer Blog for more information on this new feature in Visual Studio.

For the `ProductsBLL` class we need to add a total of seven methods:

- `GetProducts()` returns all products
- `GetProductByProductID(productID)` returns the product with the specified product ID
- `GetProductsByCategoryID(categoryID)` returns all products from the specified category
- `GetProductsBySupplier(supplierID)` returns all products from the specified supplier
- `AddProduct(productName, supplierID, categoryID, quantityPerUnit, unitPrice, unitsInStock, unitsOnOrder, reorderLevel, discontinued)` inserts a new product into the database using the values passed-in; returns the `ProductID` value of the newly inserted record
- `UpdateProduct(productName, supplierID, categoryID, quantityPerUnit, unitPrice, unitsInStock, unitsOnOrder, reorderLevel, discontinued, productID)` updates an existing product in the database using the passed-in values; returns `true` if precisely one row was updated, `false` otherwise
- `DeleteProduct(productID)` deletes the specified product from the database

ProductsBLL.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using NorthwindTableAdapters;

[System.ComponentModel.DataObject]
public class ProductsBLL
{
    private ProductsTableAdapter _productsAdapter = null;
    protected ProductsTableAdapter Adapter
    {
        get {
            if (_productsAdapter == null)
                _productsAdapter = new ProductsTableAdapter();

            return _productsAdapter;
        }
    }

    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Select, true)]
    public Northwind.ProductsDataTable GetProducts()
    {
        return Adapter.GetProducts();
    }

    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Select, false)]
    public Northwind.ProductsDataTable GetProductByProductID(int productID)
    {
        return Adapter.GetProductByProductID(productID);
    }

    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Select, false)]
    public Northwind.ProductsDataTable GetProductsByCategoryID(int categoryID)
    {
        return Adapter.GetProductsByCategoryID(categoryID);
    }

    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Select, false)]
    public Northwind.ProductsDataTable GetProductsBySupplierID(int supplierID)
    {
        return Adapter.GetProductsBySupplierID(supplierID);
    }
    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Insert, true)]
    public bool AddProduct(string productName, int? supplierID, int? categoryID,
        string quantityPerUnit, decimal? unitPrice,  short? unitsInStock,
        short? unitsOnOrder, short? reorderLevel, bool discontinued)
    {
        // Create a new ProductRow instance
        Northwind.ProductsDataTable products = new Northwind.ProductsDataTable();
        Northwind.ProductsRow product = products.NewProductsRow();

        product.ProductName = productName;
        if (supplierID == null) product.SetSupplierIDNull();
          else product.SupplierID = supplierID.Value;
        if (categoryID == null) product.SetCategoryIDNull();
```

```
      else product.CategoryID = categoryID.Value;
    if (quantityPerUnit == null) product.SetQuantityPerUnitNull();
      else product.QuantityPerUnit = quantityPerUnit;
    if (unitPrice == null) product.SetUnitPriceNull();
      else product.UnitPrice = unitPrice.Value;
    if (unitsInStock == null) product.SetUnitsInStockNull();
      else product.UnitsInStock = unitsInStock.Value;
    if (unitsOnOrder == null) product.SetUnitsOnOrderNull();
      else product.UnitsOnOrder = unitsOnOrder.Value;
    if (reorderLevel == null) product.SetReorderLevelNull();
      else product.ReorderLevel = reorderLevel.Value;
    product.Discontinued = discontinued;

    // Add the new product
    products.AddProductsRow(product);
    int rowsAffected = Adapter.Update(products);

    // Return true if precisely one row was inserted,
    // otherwise false
    return rowsAffected == 1;
}

[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Update, true)]
public bool UpdateProduct(string productName, int? supplierID, int? categoryID,
    string quantityPerUnit, decimal? unitPrice, short? unitsInStock,
    short? unitsOnOrder, short? reorderLevel, bool discontinued, int productID)
{
    Northwind.ProductsDataTable products = Adapter.GetProductByProductID(productID);
    if (products.Count == 0)
        // no matching record found, return false
        return false;

    Northwind.ProductsRow product = products[0];

    product.ProductName = productName;
    if (supplierID == null) product.SetSupplierIDNull();
      else product.SupplierID = supplierID.Value;
    if (categoryID == null) product.SetCategoryIDNull();
      else product.CategoryID = categoryID.Value;
    if (quantityPerUnit == null) product.SetQuantityPerUnitNull();
      else product.QuantityPerUnit = quantityPerUnit;
    if (unitPrice == null) product.SetUnitPriceNull();
      else product.UnitPrice = unitPrice.Value;
    if (unitsInStock == null) product.SetUnitsInStockNull();
      else product.UnitsInStock = unitsInStock.Value;
    if (unitsOnOrder == null) product.SetUnitsOnOrderNull();
      else product.UnitsOnOrder = unitsOnOrder.Value;
    if (reorderLevel == null) product.SetReorderLevelNull();
      else product.ReorderLevel = reorderLevel.Value;
    product.Discontinued = discontinued;

    // Update the product record
    int rowsAffected = Adapter.Update(product);

    // Return true if precisely one row was updated,
    // otherwise false
    return rowsAffected == 1;
}

[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Delete, true)]
public bool DeleteProduct(int productID)
{
    int rowsAffected = Adapter.Delete(productID);

    // Return true if precisely one row was deleted,
    // otherwise false
    return rowsAffected == 1;
```

```
        return rowsAffected == 1;
    }
}
```

The methods that simply return data `GetProducts`, `GetProductByProductID`, `GetProductsByCategoryID`, and `GetProductBySuppliersID` are fairly straightforward as they simply call down into the DAL. While in some scenarios there may be business rules that need to be implemented at this level (such as authorization rules based on the currently logged on user or the role to which the user belongs), we'll simply leave these methods as-is. For these methods, then, the BLL serves merely as a proxy through which the presentation layer accesses the underlying data from the Data Access Layer.

The `AddProduct` and `UpdateProduct` methods both take in as parameters the values for the various product fields and add a new product or update an existing one, respectively. Since many of the `Product` table's columns can accept `NULL` values (`CategoryID`, `SupplierID`, and `UnitPrice`, to name a few), those input parameters for `AddProduct` and `UpdateProduct` that map to such columns use nullable types. Nullable types are new to .NET 2.0 and provide a technique for indicating whether a value type should, instead, be `null`. In C# you can flag a value type as a nullable type by adding `?` after the type (like `int? x;`). Refer to the Nullable Types section in the C# Programming Guide for more information.

All three methods return a Boolean value indicating whether a row was inserted, updated, or deleted since the operation may not result in an affected row. For example, if the page developer calls `DeleteProduct` passing in a `ProductID` for a non-existent product, the `DELETE` statement issued to the database will have no affect and therefore the `DeleteProduct` method will return `false`.

Note that when adding a new product or updating an existing one we take in the new or modified product's field values as a list of scalars as opposed to accepting a `ProductsRow` instance. This approach was chosen because the `ProductsRow` class derives from the ADO.NET `DataRow` class, which doesn't have a default parameterless constructor. In order to create a new `ProductsRow` instance, we must first create a `ProductsDataTable` instance and then invoke its `NewProductRow()` method (which we do in `AddProduct`). This shortcoming rears its head when we turn to inserting and updating products using the ObjectDataSource. In short, the ObjectDataSource will try to create an instance of the input parameters. If the BLL method expects a `ProductsRow` instance, the ObjectDataSource will try to create one, but fail due to the lack of a default parameterless constructor. For more information on this problem, refer to the following two ASP.NET Forums posts: Updating ObjectDataSources with Strongly-Typed DataSets, and Problem With ObjectDataSource and Strongly-Typed DataSet.

Next, in both `AddProduct` and `UpdateProduct`, the code creates a `ProductsRow` instance and populates it with the values just passed in. When assigning values to DataColumns of a DataRow various field-level validation checks can occur. Therefore, manually putting the passed in values back into a DataRow helps ensure the validity of the data being passed to the BLL method. Unfortunately the strongly-typed DataRow classes generated by Visual Studio do not use nullable types. Rather, to indicate that a particular DataColumn in a DataRow should correspond to a `NULL` database value we must use the `SetColumnNameNull()` method.

In `UpdateProduct` we first load in the product to update using `GetProductByProductID(productID)`. While this may seem like an unnecessary trip to the database, this extra trip will prove worthwhile in future tutorials that explore optimistic concurrency. Optimistic concurrency is a technique to ensure that two users who are simultaneously working on the same data don't accidentally overwrite one another's changes. Grabbing the entire record also makes it easier to create update methods in the BLL that only modify a subset of the DataRow's columns. When we explore the `SuppliersBLL` class we'll see such an example.

Finally, note that the `ProductsBLL` class has the DataObject attribute applied to it (the `[System.ComponentModel.DataObject]` syntax right before the class statement near the top of the file) and the methods have DataObjectMethodAttribute attributes. The `DataObject` attribute marks the class as being an object suitable for binding to an ObjectDataSource control, whereas the `DataObjectMethodAttribute` indicates the purpose of the method. As we'll see in future tutorials, ASP.NET 2.0's ObjectDataSource makes it easy to

declaratively access data from a class. To help filter the list of possible classes to bind to in the ObjectDataSource's wizard, by default only those classes marked as `DataObjects` are shown in the wizard's drop-down list. The `ProductsBLL` class will work just as well without these attributes, but adding them makes it easier to work with in the ObjectDataSource's wizard.

Adding the Other Classes

With the `ProductsBLL` class complete, we still need to add the classes for working with categories, suppliers, and employees. Take a moment to create the following classes and methods using the concepts from the example above:

- **CategoriesBLL.cs**
  - `GetCategories()`
  - `GetCategoryByCategoryID(categoryID)`
- **SuppliersBLL.cs**
  - `GetSuppliers()`
  - `GetSupplierBySupplierID(supplierID)`
  - `GetSuppliersByCountry(country)`
  - `UpdateSupplierAddress(supplierID, address, city, country)`
- **EmployeesBLL.cs**
  - `GetEmployees()`
  - `GetEmployeeByEmployeeID(employeeID)`
  - `GetEmployeesByManager(managerID)`

The one method worth noting is the `SuppliersBLL` class's `UpdateSupplierAddress` method. This method provides an interface for updating just the supplier's address information. Internally, this method reads in the `SupplierDataRow` object for the specified `supplierID` (using `GetSupplierBySupplierID`), sets its address-related properties, and then calls down into the `SupplierDataTable`'s `Update` method. The `UpdateSupplierAddress` method follows:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Update, true)]
public bool UpdateSupplierAddress
    (int supplierID, string address, string city, string country)
{
    Northwind.SuppliersDataTable suppliers =
        Adapter.GetSupplierBySupplierID(supplierID);
    if (suppliers.Count == 0)
        // no matching record found, return false
        return false;
    else
    {
        Northwind.SuppliersRow supplier = suppliers[0];

        if (address == null) supplier.SetAddressNull();
          else supplier.Address = address;
        if (city == null) supplier.SetCityNull();
          else supplier.City = city;
        if (country == null) supplier.SetCountryNull();
          else supplier.Country = country;

        // Update the supplier Address-related information
        int rowsAffected = Adapter.Update(supplier);

        // Return true if precisely one row was updated,
        // otherwise false
        return rowsAffected == 1;
    }
}
```

Refer to this article's download for my complete implementation of the BLL classes.

Step 2: Accessing the Typed DataSets Through the BLL Classes

In the first tutorial we saw examples of working directly with the Typed DataSet programmatically, but with the addition of our BLL classes, the presentation tier should work against the BLL instead. In the `AllProducts.aspx` example from the first tutorial, the `ProductsTableAdapter` was used to bind the list of products to a GridView, as shown in the following code:

```
ProductsTableAdapter productsAdapter = new ProductsTableAdapter();
GridView1.DataSource = productsAdapter.GetProducts();
GridView1.DataBind();
```

To use the new BLL classes, all that needs to be changed is the first line of code simply replace the `ProductsTableAdapter` object with a `ProductBLL` object:

```
ProductsBLL productLogic = new ProductsBLL();
GridView1.DataSource = productLogic.GetProducts();
GridView1.DataBind();
```

The BLL classes can also be accessed declaratively (as can the Typed DataSet) by using the ObjectDataSource. We'll be discussing the ObjectDataSource in greater detail in the following tutorials.

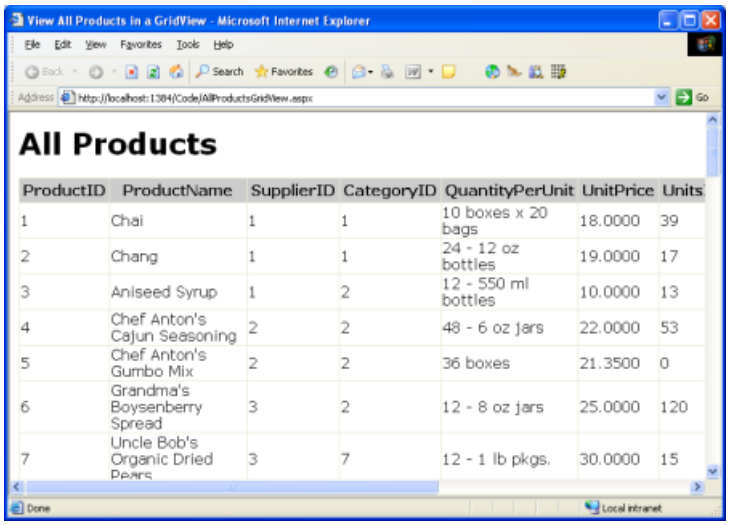


Figure 3: The List of Products is Displayed in a GridView (Click to view full-size image)

Step 3: Adding Field-Level Validation to the DataRow Classes

Field-level validation are checks that pertains to the property values of the business objects when inserting or updating. Some field-level validation rules for products include:

- The `ProductName` field must be 40 characters or less in length
- The `QuantityPerUnit` field must be 20 characters or less in length
- The `ProductID`, `ProductName`, and `Discontinued` fields are required, but all other fields are optional
- The `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, and `ReorderLevel` fields must be greater than or equal to zero

These rules can and should be expressed at the database level. The character limit on the `ProductName` and `QuantityPerUnit` fields are captured by the data types of those columns in the `Products` table (`nvarchar(40)` and `nvarchar(20)`, respectively). Whether fields are required and optional are expressed by if the database table column allows `NULL` s. Four check constraints exist that ensure that only values greater than or equal to zero can make it into the `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, or `ReorderLevel` columns.

In addition to enforcing these rules at the database they should also be enforced at the DataSet level. In fact, the field length and whether a value is required or optional are already captured for each DataTable's set of DataColumns. To see the existing field-level validation automatically provided, go to the DataSet Designer, select a field from one of the DataTables and then go to the Properties window. As Figure 4 shows, the `QuantityPerUnit` DataColumn in the `ProductsDataTable` has a maximum length of 20 characters and does allow `NULL` values. If we attempt to set the `ProductsDataRow`'s `QuantityPerUnit` property to a string value longer than 20 characters an `ArgumentException` will be thrown.

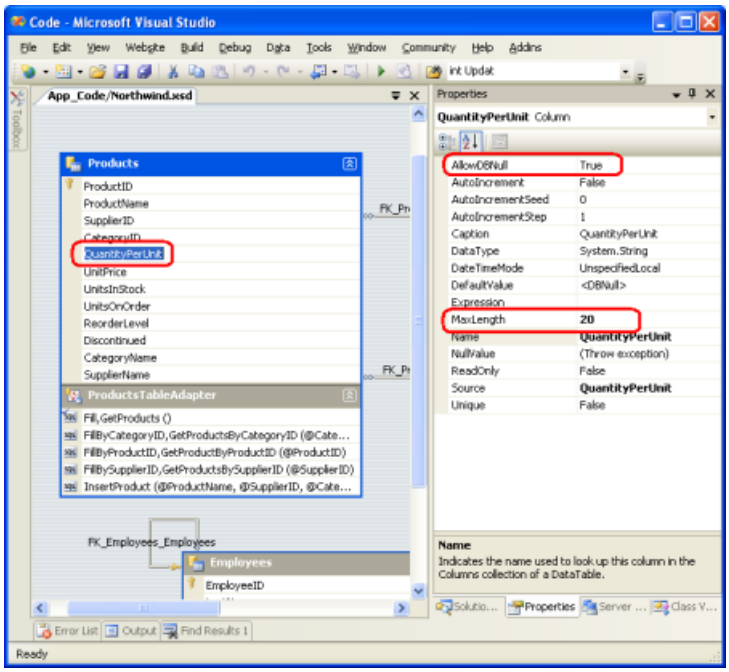


Figure 4: The DataColumn Provides Basic Field-Level Validation (Click to view full-size image)

Unfortunately, we can't specify bounds checks, such as the `UnitPrice` value must be greater than or equal to zero, through the Properties window. In order to provide this type of field-level validation we need to create an event handler for the DataTable's ColumnChanging event. As mentioned in the preceding tutorial, the DataSet, DataTables, and DataRow objects created by the Typed DataSet can be extended through the use of partial classes. Using this technique we can create a `ColumnChanging` event handler for the `ProductsDataTable` class. Start by creating a class in the `App_Code` folder named `ProductsDataTable.ColumnChanging.cs`.

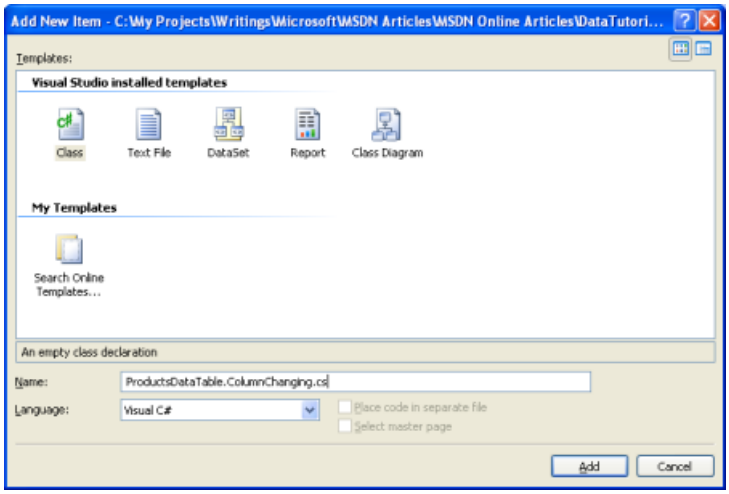


Figure 5: Add a New Class to the `App_Code` Folder (Click to view full-size image)

Next, create an event handler for the `ColumnChanging` event that ensures that the `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, and `ReorderLevel` column values (if not `NULL`) are greater than or equal to zero. If any such column is out of range, throw an `ArgumentException`.

ProductsDataTable.ColumnChanging.cs

```
public partial class Northwind
{
    public partial class ProductsDataTable
    {
        public override void BeginInit()
         {
            this.ColumnChanging += ValidateColumn;
         }

         void ValidateColumn(object sender,
           DataColumnChangeEventArgs e)
         {
            if(e.Column.Equals(this.UnitPriceColumn))
            {
               if(!Convert.IsDBNull(e.ProposedValue) &&
                  (decimal)e.ProposedValue < 0)
               {
                  throw new ArgumentException(
                      "UnitPrice cannot be less than zero", "UnitPrice");
               }
            }
            else if (e.Column.Equals(this.UnitsInStockColumn) ||
                     e.Column.Equals(this.UnitsOnOrderColumn) ||
                     e.Column.Equals(this.ReorderLevelColumn))
            {
                if (!Convert.IsDBNull(e.ProposedValue) &&
                    (short)e.ProposedValue < 0)
                {
                    throw new ArgumentException(string.Format(
                        "{0} cannot be less than zero", e.Column.ColumnName),
                        e.Column.ColumnName);
                }
            }
         }
    }
}
```

Step 4: Adding Custom Business Rules to the BLL's Classes

In addition to field-level validation, there may be high-level custom business rules that involve different entities or concepts not expressible at the single column level, such as:

- If a product is discontinued, its `UnitPrice` cannot be updated
- An employee's country of residence must be the same as their manager's country of residence
- A product cannot be discontinued if it is the only product provided by the supplier

The BLL classes should contain checks to ensure adherence to the application's business rules. These checks can be added directly to the methods to which they apply.

Imagine that our business rules dictate that a product could not be marked discontinued if it was the only product from a given supplier. That is, if product *X* was the only product we purchased from supplier *Y*, we could not mark *X* as discontinued; if, however, supplier *Y* supplied us with three products, *A*, *B*, and *C*, then we could mark any and all of these as discontinued. An odd business rule, but business rules and common sense aren't always aligned!

To enforce this business rule in the `UpdateProducts` method we'd start by checking if `Discontinued` was set to `true` and, if so, we'd call `GetProductsBySupplierID` to determine how many products we purchased from this product's supplier. If only one product is purchased from this supplier, we throw an `ApplicationException`.

```
public bool UpdateProduct(string productName, int? supplierID, int? categoryID,
    string quantityPerUnit, decimal? unitPrice, short? unitsInStock,
    short? unitsOnOrder, short? reorderLevel, bool discontinued, int productID)
{
    Northwind.ProductsDataTable products = Adapter.GetProductByProductID(productID);
    if (products.Count == 0)
        // no matching record found, return false
        return false;

    Northwind.ProductsRow product = products[0];

    // Business rule check - cannot discontinue
    // a product that is supplied by only
    // one supplier
    if (discontinued)
    {
        // Get the products we buy from this supplier
        Northwind.ProductsDataTable productsBySupplier =
            Adapter.GetProductsBySupplierID(product.SupplierID);

        if (productsBySupplier.Count == 1)
            // this is the only product we buy from this supplier
            throw new ApplicationException(
                "You cannot mark a product as discontinued if it is the only
                  product purchased from a supplier");
    }

    product.ProductName = productName;
    if (supplierID == null) product.SetSupplierIDNull();
      else product.SupplierID = supplierID.Value;
    if (categoryID == null) product.SetCategoryIDNull();
      else product.CategoryID = categoryID.Value;
    if (quantityPerUnit == null) product.SetQuantityPerUnitNull();
      else product.QuantityPerUnit = quantityPerUnit;
    if (unitPrice == null) product.SetUnitPriceNull();
      else product.UnitPrice = unitPrice.Value;
    if (unitsInStock == null) product.SetUnitsInStockNull();
      else product.UnitsInStock = unitsInStock.Value;
    if (unitsOnOrder == null) product.SetUnitsOnOrderNull();
      else product.UnitsOnOrder = unitsOnOrder.Value;
    if (reorderLevel == null) product.SetReorderLevelNull();
      else product.ReorderLevel = reorderLevel.Value;
    product.Discontinued = discontinued;

    // Update the product record
    int rowsAffected = Adapter.Update(product);

    // Return true if precisely one row was updated,
    // otherwise false
    return rowsAffected == 1;
}
```

Responding to Validation Errors in the Presentation Tier

When calling the BLL from the presentation tier we can decide whether to attempt to handle any exceptions that might be raised or let them bubble up to ASP.NET (which will raise the `HttpApplication`'s `Error` event). To handle an exception when working with the BLL programmatically, we can use a try...catch block, as the following example shows:

```
ProductsBLL productLogic = new ProductsBLL();

// Update information for ProductID 1
try
{
    // This will fail since we are attempting to use a
    // UnitPrice value less than 0.
    productLogic.UpdateProduct(
        "Scott s Tea", 1, 1, null, -14m, 10, null, null, false, 1);
}
catch (ArgumentException ae)
{
    Response.Write("There was a problem: " + ae.Message);
}
```

As we'll see in future tutorials, handling exceptions that bubble up from the BLL when using a data Web control for inserting, updating, or deleting data can be handled directly in an event handler as opposed to having to wrap code in `try...catch` blocks.

Summary

A well architected application is crafted into distinct layers, each of which encapsulates a particular role. In the first tutorial of this article series we created a Data Access Layer using Typed DataSets; in this tutorial we built a Business Logic Layer as a series of classes in our application's `App_Code` folder that call down into our DAL. The BLL implements the field-level and business-level logic for our application. In addition to creating a separate BLL, as we did in this tutorial, another option is to extend the TableAdapters' methods through the use of partial classes. However, using this technique does not allow us to override existing methods nor does it separate our DAL and our BLL as cleanly as the approach we've taken in this article.

With the DAL and BLL complete, we're ready to start on our presentation layer. In the next tutorial we'll take a brief detour from data access topics and define a consistent page layout for use throughout the tutorials.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Liz Shulok, Dennis Patterson, Carlos Santos, and Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Master Pages and Site Navigation (C#)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

One common characteristic of user-friendly websites is that they have a consistent, site-wide page layout and navigation scheme. This tutorial looks at how you can create a consistent look and feel across all pages that can easily be updated.

Introduction

One common characteristic of user-friendly websites is that they have a consistent, site-wide page layout and navigation scheme. ASP.NET 2.0 introduces two new features that greatly simplify implementing both a site-wide page layout and navigation scheme: master pages and site navigation. Master pages allow for developers to create a site-wide template with designated editable regions. This template can then be applied to ASP.NET pages in the site. Such ASP.NET pages need only provide content for the master page's specified editable regions all other markup in the master page is identical across all ASP.NET pages that use the master page. This model allows developers to define and centralize a site-wide page layout, thereby making it easier to create a consistent look and feel across all pages that can easily be updated.

The site navigation system provides both a mechanism for page developers to define a site map and an API for that site map to be programmatically queried. The new navigation Web controls the Menu, TreeView, and SiteMapPath make it easy to render all or part of the site map in a common navigation user interface element. We'll be using the default site navigation provider, meaning that our site map will be defined in an XML-formatted file.

To illustrate these concepts and make our tutorials website more usable, let's spend this lesson defining a site-wide page layout, implementing a site map, and adding the navigation UI. By the end of this tutorial we'll have a polished website design for building our tutorial web pages.

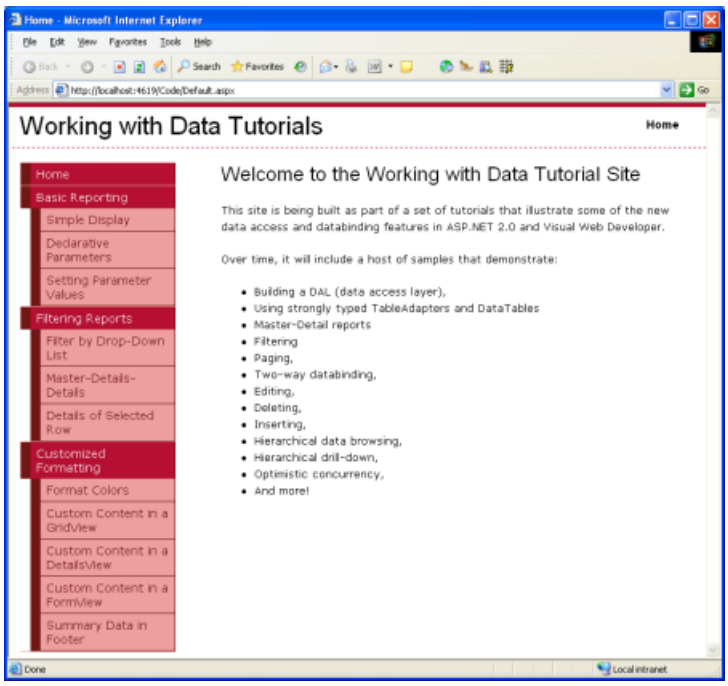


Figure 1: The End Result of This Tutorial (Click to view full-size image)

Step 1: Creating the Master Page

The first step is to create the master page for the site. Right now our website consists of only the Typed DataSet (`Northwind.xsd` , in the `App_Code` folder), the BLL classes (`ProductsBLL.cs` , `CategoriesBLL.cs` , and so on, all in the `App_Code` folder), the database (`NORTHWND.MDF` , in the `App_Data` folder), the configuration file (`Web.config`), and a CSS stylesheet file (`Styles.css`). I cleaned out those pages and files demonstrating using the DAL and BLL from the first two tutorials since we will be reexamining those examples in greater detail in future tutorials.

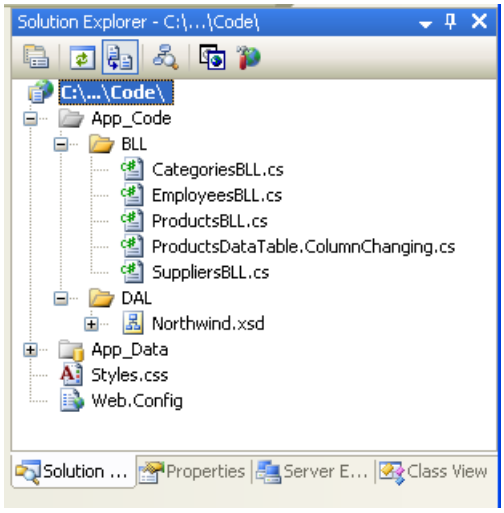


Figure 2: The Files in Our Project

To create a master page, right-click on the project name in the Solution Explorer and choose Add New Item. Then select the Master Page type from the list of templates and name it `Site.master` .

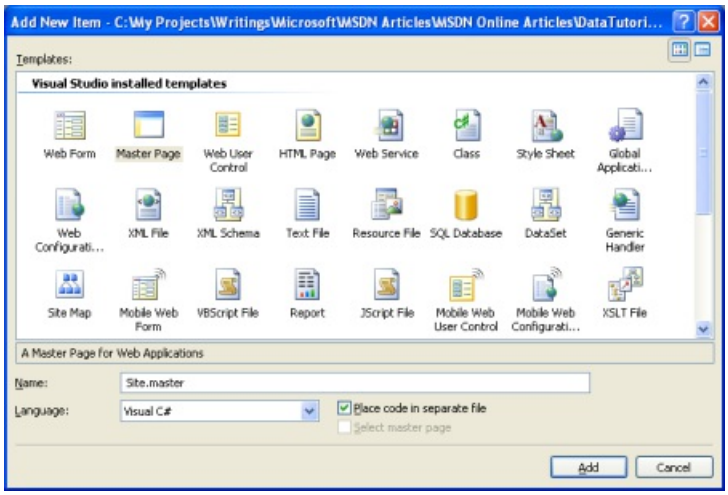


Figure 3: Add a New Master Page to the Website (Click to view full-size image)

Define the site-wide page layout here in the master page. You can use the Design view and add whatever Layout or Web controls you need, or you can manually add the markup by hand in the Source view. In my master page I use cascading style sheets for positioning and styles with the CSS settings defined in the external file `Style.css` . While you cannot tell from the markup shown below, the CSS rules are defined such that the navigation `<div>` 's content is absolutely positioned so that it appears on the left and has a fixed width of 200 pixels.

Site.master

```
<%@ Master Language="C#" AutoEventWireup="true"
    CodeFile="Site.master.cs" Inherits="Site" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Working with Data Tutorials</title>
    <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <div id="wrapper">

        <form id="form1" runat="server">

            <div id="header">
                <span class="title">Working with Data Tutorials</span>
                <span class="breadcrumb">
                 TODO: Breadcrumb will go here...</span>
            </div>

            <div id="content">
                <asp:contentplaceholder id="MainContent"
                 runat="server">
                  <!-- Page-specific content will go here... -->
                </asp:contentplaceholder>
            </div>

            <div id="navigation">
                TODO: Menu will go here...
            </div>
        </form>
    </div>
</body>
</html>
```

A master page defines both the static page layout and the regions that can be edited by the ASP.NET pages that use the master page. These content editable regions are indicated by the ContentPlaceHolder control, which can be seen within the content `<div>`. Our master page has a single ContentPlaceHolder (`MainContent`), but master page's may have multiple ContentPlaceHolders.

With the markup entered above, switching to the Design view shows the master page's layout. Any ASP.NET pages that use this master page will have this uniform layout, with the ability to specify the markup for the `MainContent` region.

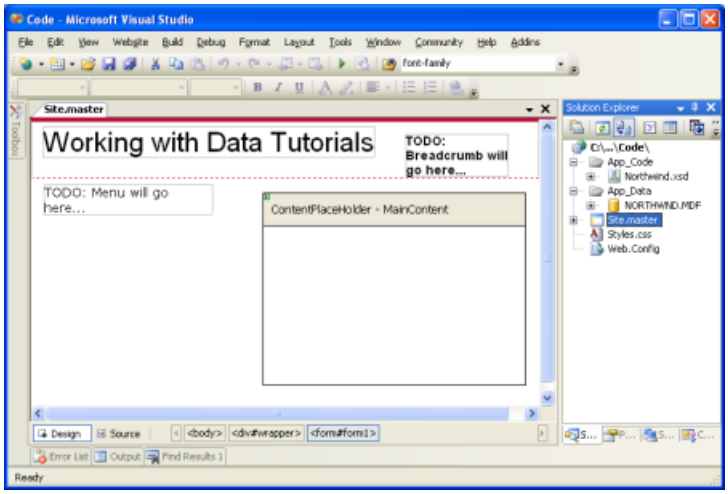


Figure 4: The Master Page, When Viewed Through the Design View (Click to view full-size image)

Step 2: Adding a Homepage to the Website

With the master page defined, we're ready to add the ASP.NET pages for the website. Let's start by adding `Default.aspx`, our website's homepage. Right-click on the project name in the Solution Explorer and choose Add New Item. Pick the Web Form option from the template list and name the file `Default.aspx`. Also, check the "Select master page" checkbox.

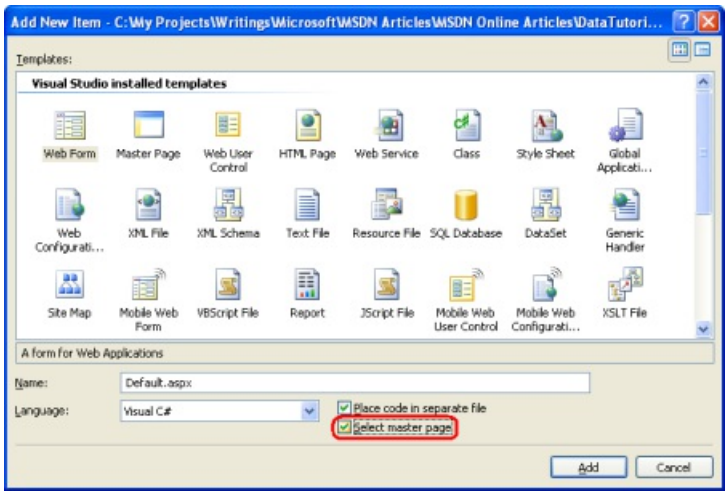


Figure 5: Add a New Web Form, Checking the Select master page Checkbox (Click to view full-size image)

After clicking the OK button, we're asked to choose what master page this new ASP.NET page should use. While you can have multiple master pages in your project, we have only one.

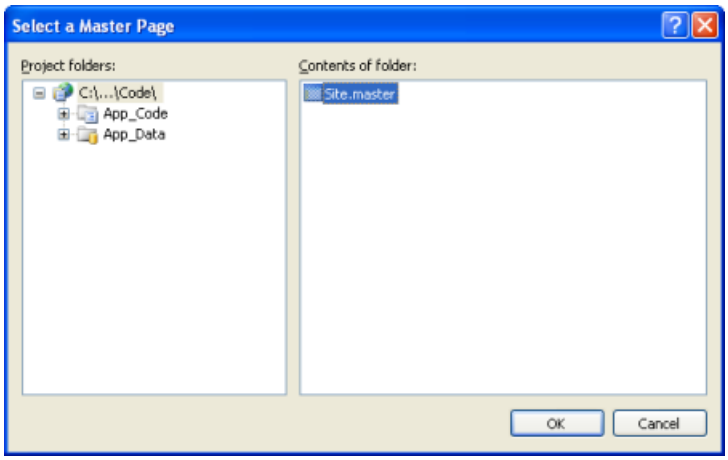


Figure 6: Choose the Master Page this ASP.NET Page Should Use (Click to view full-size image)

After picking the master page, the new ASP.NET pages will contain the following markup:

Default.aspx

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent"
  Runat="Server">
</asp:Content>
```

In the `@Page` directive there's a reference to the master page file used (`MasterPageFile="~/Site.master"`), and the ASP.NET page's markup contains a Content control for each of the ContentPlaceHolder controls defined in the master page, with the control's `ContentPlaceHolderID` mapping the Content control to a specific ContentPlaceHolder. The Content control is where you place the markup you want to appear in the corresponding ContentPlaceHolder. Set the `@Page` directive's `Title` attribute to Home and add some welcoming content to the Content control:

Default.aspx

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" Title="Home" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent"
    Runat="Server">
    <h1>Welcome to the Working with Data Tutorial Site</h1>

    <p>This site is being built as part of a set of tutorials that
illustrate some of the new data access and databinding features in
ASP.NET 2.0 and Visual Web Developer.</p>

    <p>Over time, it will include a host of samples that
demonstrate:</p>

    <ul>
        <li>Building a DAL (data access layer),</li>
        <li>Using strongly typed TableAdapters and DataTables</li>
        <li>Master-Detail reports</li>
        <li>Filtering</li>
        <li>Paging,</li>
        <li>Two-way databinding,</li>
        <li>Editing,</li>
        <li>Deleting,</li>
        <li>Inserting,</li>
        <li>Hierarchical data browsing,</li>
        <li>Hierarchical drill-down,</li>
        <li>Optimistic concurrency,</li>
        <li>And more!</li>
    </ul>
</asp:Content>
```

The `Title` attribute in the `@Page` directive allows us to set the page's title from the ASP.NET page, even though the `<title>` element is defined in the master page. We can also set the title programmatically, using `Page.Title`. Also note that the master page's references to stylesheets (such as `Style.css`) are automatically updated so that they work in any ASP.NET page, regardless of what directory the ASP.NET page is in relative to the master page.

Switching to the Design view we can see how our page will look in a browser. Note that in the Design view for the ASP.NET page that only the content editable regions are editable the non-ContentPlaceHolder markup defined in the master page is grayed out.

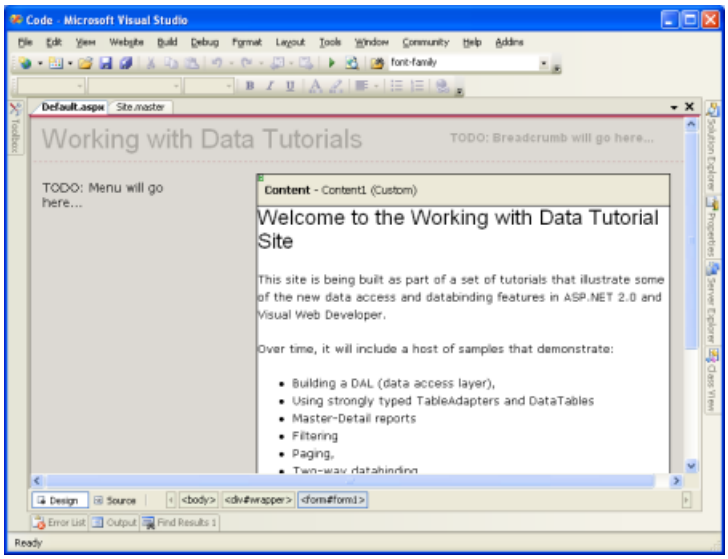


Figure 7: The Design View for the ASP.NET Page Shows Both the Editable and Non-Editable Regions (Click to view full-size image)

When the `Default.aspx` page is visited by a browser, the ASP.NET engine automatically merges the page's master page content and the ASP.NET's content, and renders the merged content into the final HTML that is sent down to the requesting browser. When the master page's content is updated, all ASP.NET pages that use this master page will have their content remerged with the new master page content the next time they are requested. In short, the master page model allows for a single page layout template to be defined (the master page) whose changes are immediately reflected across the entire site.

Adding Additional ASP.NET Pages to the Website

Let's take a moment to add additional ASP.NET page stubs to the site that will eventually hold the various reporting demos. There will be more than 35 demos in total, so rather than creating all of the stub pages let's just create the first few. Since there will also be many categories of demos, to better manage the demos add a folder for the categories. Add the following three folders for now:

- `BasicReporting`
- `Filtering`
- `CustomFormatting`

Finally, add new files as shown in the Solution Explorer in Figure 8. When adding each file, remember to check the "Select master page" checkbox.

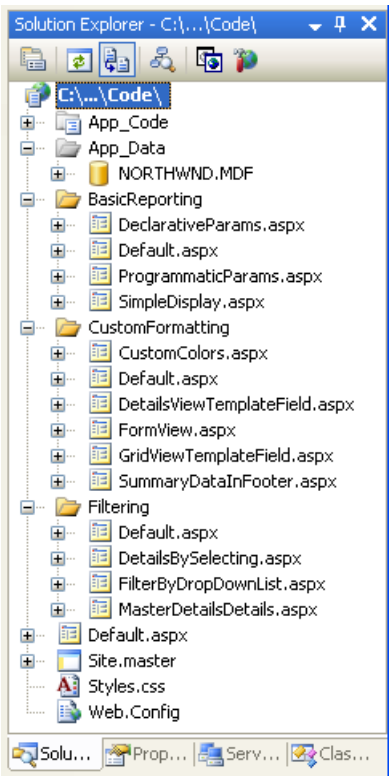


Figure 8: Add the Following Files

Step 2: Creating a Site Map

One of the challenges of managing a website composed of more than a handful of pages is providing a straightforward way for visitors to navigate through the site. To begin with, the site's navigational structure must be defined. Next, this structure must be translated into navigable user interface elements, such as menus or breadcrumbs. Finally, this whole process needs to be maintained and updated as new pages are added to the site and existing ones removed. Prior to ASP.NET 2.0, developers were on their own for creating the site's navigational structure, maintaining it, and translating it into navigable user interface elements. With ASP.NET 2.0, however, developers can utilize the very flexible built in site navigation system.

The ASP.NET 2.0 site navigation system provides a means for a developer to define a site map and to then access

this information through a programmatic API. ASP.NET ships with a site map provider that expects site map data to be stored in an XML file formatted in a particular way. But, since the site navigation system is built on the provider model it can be extended to support alternative ways for serializing the site map information. Jeff Prosise's article, The SQL Site Map Provider You've Been Waiting For shows how to create a site map provider that stores the site map in a SQL Server database; another option is to create a site map provider based on the file system structure.

For this tutorial, however, let's use the default site map provider that ships with ASP.NET 2.0. To create the site map, simply right-click on the project name in the Solution Explorer, choose Add New Item, and choose the Site Map option. Leave the name as `Web.sitemap` and click the Add button.

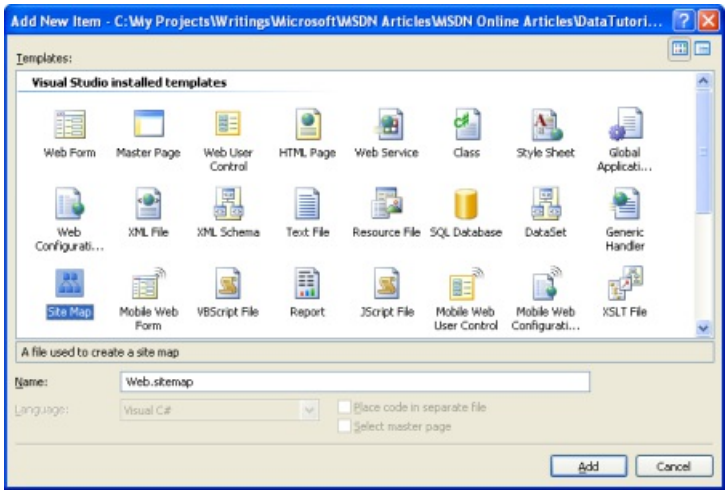


Figure 9: Add a Site Map to Your Project (Click to view full-size image)

The site map file is an XML file. Note that Visual Studio provides IntelliSense for the site map structure. The site map file must have the `<siteMap>` node as its root node, which must contain precisely one `<siteMapNode>` child element. That first `<siteMapNode>` element can then contain an arbitrary number of descendent `<siteMapNode>` elements.

Define the site map to mimic the file system structure. That is, add a `<siteMapNode>` element for each of the three folders, and child `<siteMapNode>` elements for each of the ASP.NET pages in those folders, like so:

Web.sitemap

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >

  <siteMapNode url="~/Default.aspx" title="Home" description="Home">
      <siteMapNode title="Basic Reporting"
        url="~/BasicReporting/Default.aspx"
        description="Basic Reporting Samples">
        <siteMapNode url="~/BasicReporting/SimpleDisplay.aspx"
         title="Simple Display"
         description="Displays the complete contents
          of a database table." />
        <siteMapNode url="~/BasicReporting/DeclarativeParams.aspx"
          title="Declarative Parameters"
          description="Displays a subset of the contents
            of a database table using parameters." />
        <siteMapNode url="~/BasicReporting/ProgrammaticParams.aspx"
         title="Setting Parameter Values"
         description="Shows how to set parameter values
          programmatically." />
      </siteMapNode>

      <siteMapNode title="Filtering Reports"
       url="~/Filtering/Default.aspx"
       description="Samples of Reports that Support Filtering">
        <siteMapNode url="~/Filtering/FilterByDropDownList.aspx"
          title="Filter by Drop-Down List"
          description="Filter results using a drop-down list." />
        <siteMapNode url="~/Filtering/MasterDetailsDetails.aspx"
         title="Master-Details-Details"
         description="Filter results two levels down." />
        <siteMapNode url="~/Filtering/DetailsBySelecting.aspx"
          title="Details of Selected Row"
          description="Show detail results for a selected item in a GridView." />
      </siteMapNode>

      <siteMapNode title="Customized Formatting"
         url="~/CustomFormatting/Default.aspx"
         description="Samples of Reports Whose Formats are Customized">
        <siteMapNode url="~/CustomFormatting/CustomColors.aspx"
         title="Format Colors"
         description="Format the grid s colors based
           on the underlying data." />
        <siteMapNode
          url="~/CustomFormatting/GridViewTemplateField.aspx"
          title="Custom Content in a GridView"
          description="Shows using the TemplateField to
          customize the contents of a field in a GridView." />
        <siteMapNode
          url="~/CustomFormatting/DetailsViewTemplateField.aspx"
          title="Custom Content in a DetailsView"
          description="Shows using the TemplateField to customize
           the contents of a field in a DetailsView." />
        <siteMapNode url="~/CustomFormatting/FormView.aspx"
          title="Custom Content in a FormView"
          description="Illustrates using a FormView for a
           highly customized view." />
        <siteMapNode url="~/CustomFormatting/SummaryDataInFooter.aspx"
          title="Summary Data in Footer"
          description="Display summary data in the grids footer." />
      </siteMapNode>

  </siteMapNode>

</siteMap>
```

The site map defines the website's navigational structure, which is a hierarchy that describes the various sections

of the site. Each `<siteMapNode>` element in `Web.sitemap` represents a section in the site's navigational structure.

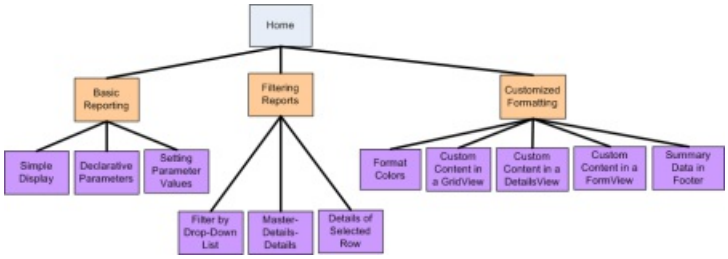


Figure 10: The Site Map Represents a Hierarchical Navigational Structure (Click to view full-size image)

ASP.NET exposes the site map's structure through the .NET Framework's SiteMap class. This class has a `CurrentNode` property, which returns information about the section the user is currently visiting; the `RootNode` property returns the root of the site map (Home, in our site map). Both the `CurrentNode` and `RootNode` properties return SiteMapNode instances, which have properties like `ParentNode`, `ChildNodes`, `NextSibling`, `PreviousSibling`, and so on, that allow for the site map hierarchy to be walked.

Step 3: Displaying a Menu Based on the Site Map

Accessing data in ASP.NET 2.0 can be accomplished programmatically, like in ASP.NET 1.x, or declaratively, through the new data source controls. There are several built-in data source controls such as the SqlDataSource control, for accessing relational database data, the ObjectDataSource control, for accessing data from classes, and others. You can even create your own custom data source controls.

The data source controls serve as a proxy between your ASP.NET page and the underlying data. In order to display a data source control's retrieved data, we'll typically add another Web control to the page and bind it to the data source control. To bind a Web control to a data source control, simply set the Web control's `DataSourceID` property to the value of the data source control's `ID` property.

To aid in working with the site map's data, ASP.NET includes the SiteMapDataSource control, which allows us to bind a Web control against our website's site map. Two Web controls the TreeView and Menu are commonly used to provide a navigation user interface. To bind the site map data to one of these two controls, simply add a SiteMapDataSource to the page along with a TreeView or Menu control whose `DataSourceID` property is set accordingly. For example, we could add a Menu control to the master page using the following markup:

```
<div id="navigation">
    <asp:Menu ID="Menu1" runat="server"
      DataSourceID="SiteMapDataSource1">
    </asp:Menu>

    <asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
</div>
```

For a finer degree of control over the emitted HTML, we can bind the SiteMapDataSource control to the Repeater control, like so:

```
<div id="navigation">
    <ul>
        <li><asp:HyperLink runat="server" ID="lnkHome"
         NavigateUrl="~/Default.aspx">Home</asp:HyperLink></li>

        <asp:Repeater runat="server" ID="menu"
          DataSourceID="SiteMapDataSource1">
            <ItemTemplate>
                <li>
                    <asp:HyperLink runat="server"
                    NavigateUrl='<%# Eval("Url") %>'>
                    <%# Eval("Title") %></asp:HyperLink>
                </li>
            </ItemTemplate>
        </asp:Repeater>
    </ul>

    <asp:SiteMapDataSource ID="SiteMapDataSource1"
      runat="server" ShowStartingNode="false" />
</div>
```

The SiteMapDataSource control returns the site map hierarchy one level at a time, starting with the root site map node (Home, in our site map), then the next level (Basic Reporting, Filtering Reports, and Customized Formatting), and so on. When binding the SiteMapDataSource to a Repeater, it enumerates the first level returned and instantiates the `ItemTemplate` for each `SiteMapNode` instance in that first level. To access a particular property of the `SiteMapNode`, we can use `Eval(propertyName)`, which is how we get each `SiteMapNode` 's `Url` and `Title` properties for the HyperLink control.

The Repeater example above will render the following markup:

```
<li>
    <a href="/Code/BasicReporting/Default.aspx">Basic Reporting</a>
</li>

<li>
    <a href="/Code/Filtering/Default.aspx">Filtering Reports</a>
</li>

<li>
    <a href="/Code/CustomFormatting/Default.aspx">
     Customized Formatting</a>
</li>
```

These site map nodes (Basic Reporting, Filtering Reports, and Customized Formatting) comprise the *second* level of the site map being rendered, not the first. This is because the SiteMapDataSource's `ShowStartingNode` property is set to False, causing the SiteMapDataSource to bypass the root site map node and instead begin by returning the second level in the site map hierarchy.

To display the children for the Basic Reporting, Filtering Reports, and Customized Formatting `SiteMapNode` s, we can add another Repeater to the initial Repeater's `ItemTemplate`. This second Repeater will be bound to the `SiteMapNode` instance's `ChildNodes` property, like so:

```
<asp:Repeater runat="server" ID="menu" DataSourceID="SiteMapDataSource1">
    <ItemTemplate>
        <li>
            <asp:HyperLink runat="server"
             NavigateUrl='<%# Eval("Url") %>'>
             <%# Eval("Title") %></asp:HyperLink>

            <asp:Repeater runat="server"
                DataSource='<%# ((SiteMapNode) Container.DataItem).ChildNodes %>'>
                <HeaderTemplate>
                    <ul>
                </HeaderTemplate>

                <ItemTemplate>
                    <li>
                        <asp:HyperLink runat="server"
                         NavigateUrl='<%# Eval("Url") %>'>
                         <%# Eval("Title") %></asp:HyperLink>
                    </li>
                </ItemTemplate>

                <FooterTemplate>
                    </ul>
                </FooterTemplate>
            </asp:Repeater>
        </li>
    </ItemTemplate>
</asp:Repeater>
```

These two Repeaters result in the following markup (some markup has been removed for brevity):

```
<li>
    <a href="/Code/BasicReporting/Default.aspx">Basic Reporting</a>
    <ul>
       <li>
          <a href="/Code/BasicReporting/SimpleDisplay.aspx">
            Simple Display</a>
       </li>
       <li>
          <a href="/Code/BasicReporting/DeclarativeParams.aspx">
            Declarative Parameters</a>
       </li>
       <li>
          <a href="/Code/BasicReporting/ProgrammaticParams.aspx">
            Setting Parameter Values</a>
       </li>
    </ul>
</li>

<li>
    <a href="/Code/Filtering/Default.aspx">Filtering Reports</a>
    ...
</li>

<li>
    <a href="/Code/CustomFormatting/Default.aspx">
      Customized Formatting</a>
    ...
</li>
```

Using CSS styles chosen from Rachel Andrew's book The CSS Anthology: 101 Essential Tips, Tricks, & Hacks, the `<ul>` and `<li>` elements are styled such that the markup produces the following visual output:

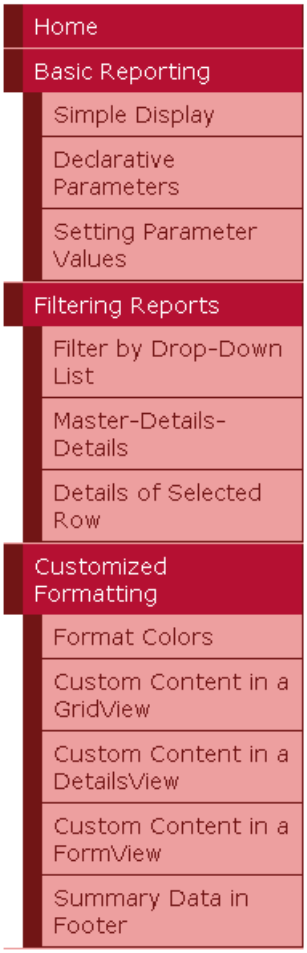


Figure 11: A Menu Composed from Two Repeaters and Some CSS

This menu is in the master page and bound to the site map defined in `Web.sitemap`, meaning that any change to the site map will be immediately reflected on all pages that use the `Site.master` master page.

Disabling ViewState

All ASP.NET controls can optionally persist their state to the view state, which is serialized as a hidden form field in the rendered HTML. View state is used by controls to remember their programmatically-changed state across postbacks, such as the data bound to a data Web control. While view state permits information to be remembered across postbacks, it increases the size of the markup that must be sent to the client and can lead to severe page bloat if not closely monitored. Data Web controls especially the GridView are particularly notorious for adding dozens of extra kilobytes of markup to a page. While such an increase may be negligible for broadband or intranet users, view state can add several seconds to the round trip for dial-up users.

To see the impact of view state, visit a page in a browser and then view the source sent by the web page (in Internet Explorer, go to the View menu and choose the Source option). You can also turn on page tracing to see the view state allocation used by each of the controls on the page. The view state information is serialized in a hidden form field named `__VIEWSTATE`, located in a `<div>` element immediately after the opening `<form>` tag. View state is only persisted when there is a Web Form being used; if your ASP.NET page does not include a `<form runat="server">` in its declarative syntax there won't be a `__VIEWSTATE` hidden form field in the rendered markup.

The `__VIEWSTATE` form field generated by the master page adds roughly 1,800 bytes to the page's generated markup. This extra bloat is due primarily to the Repeater control, as the contents of the SiteMapDataSource control are persisted to view state. While an extra 1,800 bytes may not seem like much to get excited about, when using a GridView with many fields and records, the view state can easily swell by a factor of 10 or more.

View state can be disabled at the page or control level by setting the `EnableViewState` property to `false`,

thereby reducing the size of the rendered markup. Since the view state for a data Web control persists the data bound to the data Web control across postbacks, when disabling the view state for a data Web control the data must be bound on each and every postback. In ASP.NET version 1.x this responsibility fell on the shoulders of the page developer; with ASP.NET 2.0, however, the data Web controls will rebind to their data source control on each postback if needed.

To reduce the page's view state let's set the Repeater control's `EnableViewState` property to `false`. This can be done through the Properties window in the Designer or declaratively in the Source view. After making this change the Repeater's declarative markup should look like:

```
<asp:Repeater runat="server" ID="menu" DataSourceID="SiteMapDataSource1"
    EnableViewState="False">
    <ItemTemplate>
        ... <i>ItemTemplate contents omitted for brevity</i> ...
    </ItemTemplate>
</asp:Repeater>
```

After this change, the page's rendered view state size has shrunk to a mere 52 bytes, a 97% savings in view state size! In the tutorials throughout this series we'll disable the view state of the data Web controls by default in order to reduce the size of the rendered markup. In the majority of the examples the `EnableViewState` property will be set to `false` and done so without mention. The only time view state will be discussed is in scenarios where it must be enabled in order for the data Web control to provide its expected functionality.

Step 4: Adding Breadcrumb Navigation

To complete the master page, let's add a breadcrumb navigation UI element to each page. The breadcrumb quickly shows users their current position within the site hierarchy. Adding a breadcrumb in ASP.NET 2.0 is easy just add a SiteMapPath control to the page; no code is needed.

For our site, add this control to the header `<div>`:

```
<span class="breadcrumb">
    <asp:SiteMapPath ID="SiteMapPath1" runat="server">
    </asp:SiteMapPath>
</span>
```

The breadcrumb shows the current page the user is visiting in the site map hierarchy as well as that site map node's "ancestors," all the way up to the root (Home, in our site map).

Figure 12: The Breadcrumb Displays the Current Page and its Ancestors in the Site Map Hierarchy

Step 5: Adding the Default Page for Each Section

The tutorials in our site are broken down into different categories Basic Reporting, Filtering, Custom Formatting, and so on with a folder for each category and the corresponding tutorials as ASP.NET pages within that folder. Additionally, each folder contains a `Default.aspx` page. For this default page, let's display all of the tutorials for the current section. That is, for the `Default.aspx` in the `BasicReporting` folder we'd have links to `SimpleDisplay.aspx`, `DeclarativeParams.aspx`, and `ProgrammaticParams.aspx`. Here, again, we can use the `SiteMap` class and a data Web control to display this information based upon the site map defined in `Web.sitemap`.

Let's display an unordered list using a Repeater again, but this time we'll display the title and description of the

tutorials. Since the markup and code to accomplish this will need to be repeated for each `Default.aspx` page, we can encapsulate this UI logic in a User Control. Create a folder in the website called `UserControls` and add to that a new item of type Web User Control named `SectionLevelTutorialListing.ascx`, and add the following markup:

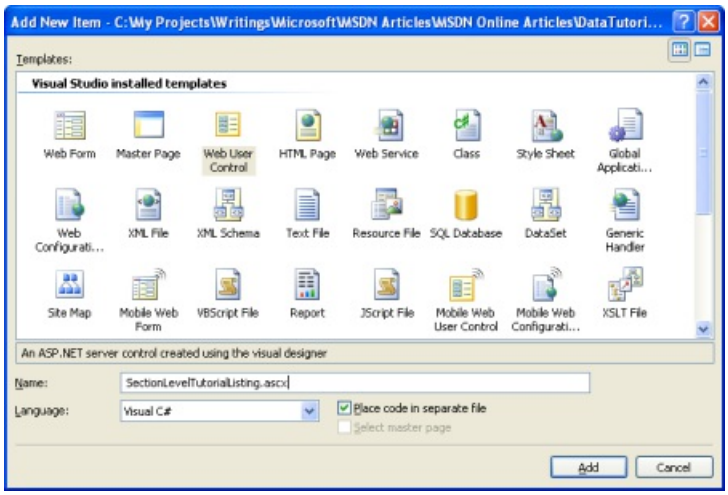


Figure 13: Add a New Web User Control to the `UserControls` Folder (Click to view full-size image)

SectionLevelTutorialListing.ascx

```
<%@ Control Language="CS" AutoEventWireup="true"
    CodeFile="SectionLevelTutorialListing.ascx.cs"
    Inherits="UserControls_SectionLevelTutorialListing" %>
<asp:Repeater ID="TutorialList" runat="server" EnableViewState="False">
    <HeaderTemplate><ul></HeaderTemplate>
    <ItemTemplate>
        <li><asp:HyperLink runat="server"
         NavigateUrl='<%# Eval("Url") %>'
         Text='<%# Eval("Title") %>'></asp:HyperLink>
                - <%# Eval("Description") %></li>
    </ItemTemplate>
    <FooterTemplate></ul></FooterTemplate>
</asp:Repeater>
```

SectionLevelTutorialListing.ascx.cs

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class UserControls_SectionLevelTutorialListing : UserControl
{
    protected void Page_Load(object sender, EventArgs e)
    {
        // If SiteMap.CurrentNode is not null,
        // bind CurrentNode ChildNodes to the GridView
        if (SiteMap.CurrentNode != null)
        {
            TutorialList.DataSource = SiteMap.CurrentNode.ChildNodes;
            TutorialList.DataBind();
        }
    }
}
```

In the previous Repeater example we bound the `SiteMap` data to the Repeater declaratively; the `SectionLevelTutorialListing` User Control, however, does so programmatically. In the `Page_Load` event handler, a check is made to ensure that this page s URL maps to a node in the site map. If this User Control is used in a page that does not have a corresponding `<siteMapNode>` entry, `SiteMap.CurrentNode` will return `null` and no data will be bound to the Repeater. Assuming we have a `CurrentNode`, we bind its `ChildNodes` collection to the Repeater. Since our site map is set up such that the `Default.aspx` page in each section is the parent node of all of the tutorials within that section, this code will display links to and descriptions of all of the section's tutorials, as shown in the screen shot below.

Once this Repeater has been created, open the `Default.aspx` pages in each of the folders, go to the Design view, and simply drag the User Control from the Solution Explorer onto the Design surface where you want the tutorial list to appear.

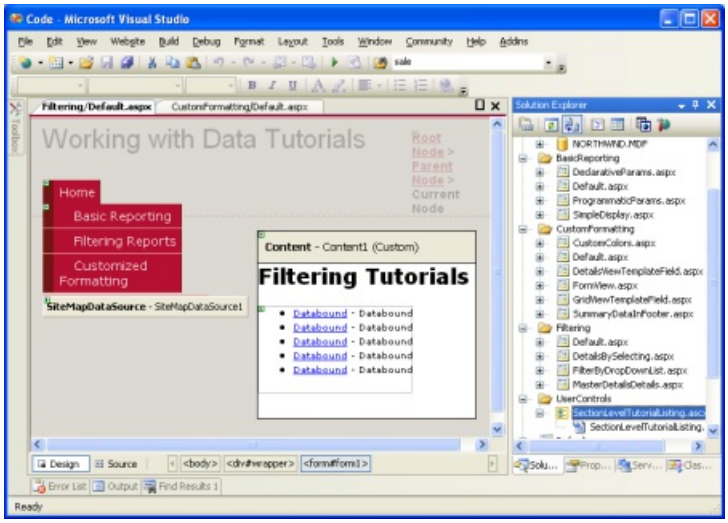


Figure 14: The User Control has Been Added to `Default.aspx` (Click to view full-size image)

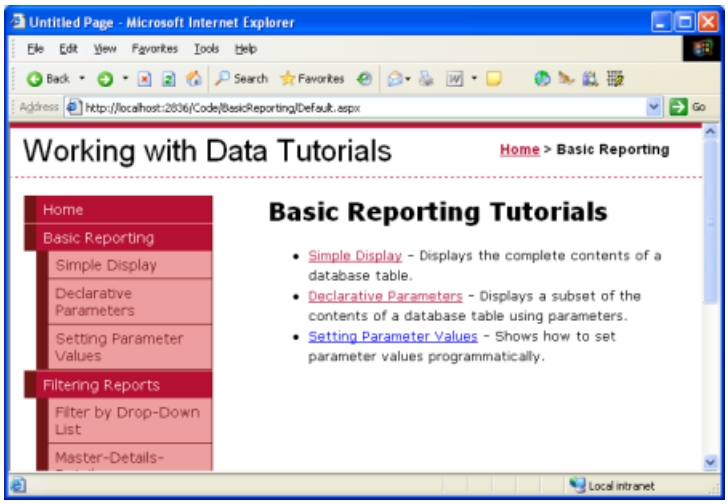


Figure 15: The Basic Reporting Tutorials are Listed (Click to view full-size image)

Summary

With the site map defined and the master page complete, we now have a consistent page layout and navigation scheme for our data-related tutorials. Regardless of how many pages we add to our site, updating the site-wide page layout or site navigation information is a quick and simple process due to this information being centralized. Specifically, the page layout information is defined in the master page `Site.master` and the site map in `Web.sitemap`. We didn't need to write *any* code to achieve this site-wide page layout and navigation mechanism, and we retain full WYSIWYG designer support in Visual Studio.

Having completed the Data Access Layer and Business Logic Layer and having a consistent page layout and site navigation defined, we're ready to begin exploring common reporting patterns. In the next three tutorials we'll look at basic reporting tasks displaying data retrieved from the BLL in the GridView, DetailsView, and FormView controls.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Master Pages Overview
- Master Pages in ASP.NET 2.0
- ASP.NET 2.0 Design Templates
- ASP.NET Site Navigation Overview
- Examining ASP.NET 2.0's Site Navigation
- ASP.NET 2.0 Site Navigation Features
- Understanding ASP.NET View State
- How to: Enable Tracing for an ASP.NET Page
- ASP.NET User Controls

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Liz Shulok, Dennis Patterson, and Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Creating a Data Access Layer (VB)

2/20/2020 • 33 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll start from the very beginning and create the Data Access Layer (DAL), using typed DataSets, to access the information in a database.

Introduction

As web developers, our lives revolve around working with data. We create databases to store the data, code to retrieve and modify it, and web pages to collect and summarize it. This is the first tutorial in a lengthy series that will explore techniques for implementing these common patterns in ASP.NET 2.0. We'll start with creating a software architecture composed of a Data Access Layer (DAL) using Typed DataSets, a Business Logic Layer (BLL) that enforces custom business rules, and a presentation layer composed of ASP.NET pages that share a common page layout. Once this backend groundwork has been laid, we'll move into reporting, showing how to display, summarize, collect, and validate data from a web application. These tutorials are geared to be concise and provide step-by-step instructions with plenty of screen shots to walk you through the process visually. Each tutorial is available in C# and Visual Basic versions and includes a download of the complete code used. (This first tutorial is quite lengthy, but the rest are presented in much more digestible chunks.)

For these tutorials we'll be using a Microsoft SQL Server 2005 Express Edition version of the Northwind database placed in the `App_Data` directory. In addition to the database file, the `App_Data` folder also contains the SQL scripts for creating the database, in case you want to use a different database version. These scripts can be also be downloaded directly from Microsoft, if you'd prefer. If you use a different SQL Server version of the Northwind database, you will need to update the `NORTHWNDConnectionString` setting in the application's `Web.config` file. The web application was built using Visual Studio 2005 Professional Edition as a file system-based Web site project. However, all of the tutorials will work equally well with the free version of Visual Studio 2005, Visual Web Developer.

In this tutorial we'll start from the very beginning and create the Data Access Layer (DAL), followed by creating the Business Logic Layer (BLL) in the second tutorial, and working on page layout and navigation in the third. The tutorials after the third one will build upon the foundation laid in the first three. We've got a lot to cover in this first tutorial, so fire up Visual Studio and let's get started!

Step 1: Creating a Web Project and Connecting to the Database

Before we can create our Data Access Layer (DAL), we first need to create a web site and setup our database. Start by creating a new file system-based ASP.NET web site. To accomplish this, go to the File menu and choose New Web Site, displaying the New Web Site dialog box. Choose the ASP.NET Web Site template, set the Location drop-down list to File System, choose a folder to place the web site, and set the language to Visual Basic.

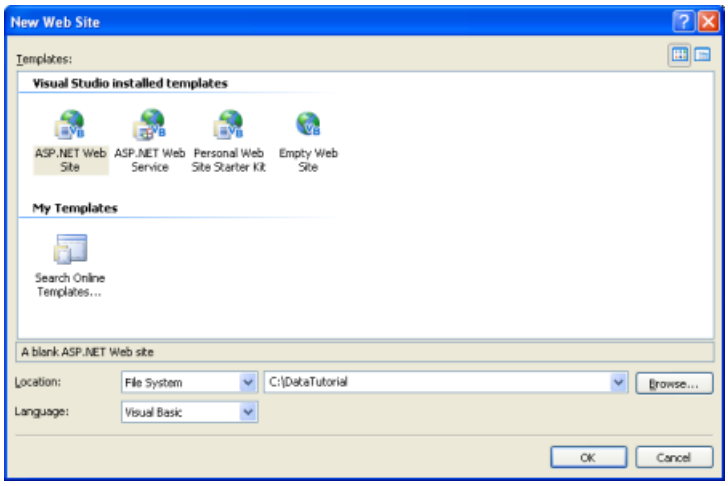


Figure 1: Create a New File System-Based Web Site (Click to view full-size image)

This will create a new web site with a `Default.aspx` ASP.NET page, an `App_Data` folder, and a `Web.config` file.

With the web site created, the next step is to add a reference to the database in Visual Studio's Server Explorer. By adding a database to the Server Explorer you can add tables, stored procedures, views, and so on all from within Visual Studio. You can also view table data or create your own queries either by hand or graphically via the Query Builder. Furthermore, when we build the Typed DataSets for the DAL we'll need to point Visual Studio to the database from which the Typed DataSets should be constructed. While we can provide this connection information at that point in time, Visual Studio automatically populates a drop-down list of the databases already registered in the Server Explorer.

The steps for adding the Northwind database to the Server Explorer depend on whether you want to use the SQL Server 2005 Express Edition database in the `App_Data` folder or if you have a Microsoft SQL Server 2000 or 2005 database server setup that you want to use instead.

Using a Database in the `App_Data` Folder

If you do not have a SQL Server 2000 or 2005 database server to connect to, or you simply want to avoid having to add the database to a database server, you can use the SQL Server 2005 Express Edition version of the Northwind database that is located in the downloaded website's `App_Data` folder (`NORTHWND.MDF`).

A database placed in the `App_Data` folder is automatically added to the Server Explorer. Assuming you have SQL Server 2005 Express Edition installed on your machine you should see a node named NORTHWND.MDF in the Server Explorer, which you can expand and explore its tables, views, stored procedure, and so on (see Figure 2).

The `App_Data` folder can also hold Microsoft Access `.mdb` files, which, like their SQL Server counterparts, are automatically added to the Server Explorer. If you don't want to use any of the SQL Server options, you can always download a Microsoft Access version of the Northwind database file and drop into the `App_Data` directory. Keep in mind, however, that Access databases aren't as feature-rich as SQL Server, and aren't designed to be used in web site scenarios. Furthermore, a couple of the 35+ tutorials will utilize certain database-level features that aren't supported by Access.

Connecting to the Database in a Microsoft SQL Server 2000 or 2005 Database Server

Alternatively, you may connect to a Northwind database installed on a database server. If the database server does not already have the Northwind database installed, you first must add it to database server by running the installation script included in this tutorial's download or by downloading the SQL Server 2000 version of Northwind and installation script directly from Microsoft's web site.

Once you have the database installed, go to the Server Explorer in Visual Studio, right-click on the Data Connections node, and choose Add Connection. If you don't see the Server Explorer go to the View / Server Explorer, or hit Ctrl+Alt+S. This will bring up the Add Connection dialog box, where you can specify the server to connect to, the authentication information, and the database name. Once you have successfully configured the database connection information and clicked the OK button, the database will be added as a node underneath the Data Connections node. You can expand the database node to explore its tables, views, stored procedures, and so on.

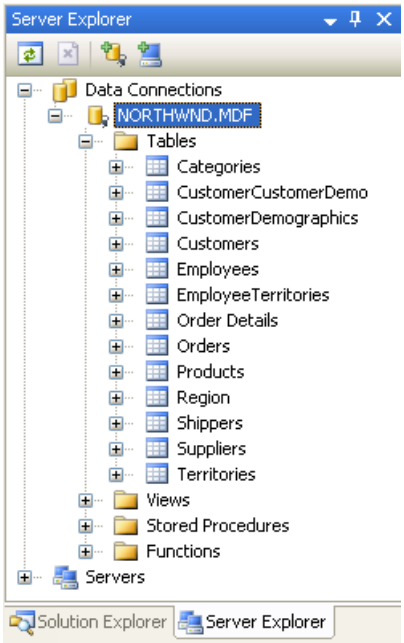


Figure 2: Add a Connection to Your Database Server's Northwind Database

Step 2: Creating the Data Access Layer

When working with data one option is to embed the data-specific logic directly into the presentation layer (in a web application, the ASP.NET pages make up the presentation layer). This may take the form of writing ADO.NET code in the ASP.NET page's code portion or using the SqlDataSource control from the markup portion. In either case, this approach tightly couples the data access logic with the presentation layer. The recommended approach, however, is to separate the data access logic from the presentation layer. This separate layer is referred to as the Data Access Layer, DAL for short, and is typically implemented as a separate Class Library project. The benefits of this layered architecture are well documented (see the "Further Readings" section at the end of this tutorial for information on these advantages) and is the approach we will take in this series.

All code that is specific to the underlying data source such as creating a connection to the database, issuing `SELECT`, `INSERT`, `UPDATE`, and `DELETE` commands, and so on should be located in the DAL. The presentation layer should not contain any references to such data access code, but should instead make calls into the DAL for any and all data requests. Data Access Layers typically contain methods for accessing the underlying database data. The Northwind database, for example, has `Products` and `Categories` tables that record the products for sale and the categories to which they belong. In our DAL we will have methods like:

- `GetCategories(),` which will return information about all of the categories
- `GetProducts()`, which will return information about all of the products
- `GetProductsByCategoryID(categoryID)`, which will return all products that belong to a specified category
- `GetProductByProductID(productID)`, which will return information about a particular product

These methods, when invoked, will connect to the database, issue the appropriate query, and return the results. How we return these results is important. These methods could simply return a DataSet or DataReader

populated by the database query, but ideally these results should be returned using *strongly-typed objects*. A strongly-typed object is one whose schema is rigidly defined at compile time, whereas the opposite, a loosely-typed object, is one whose schema is not known until runtime.

For example, the DataReader and the DataSet (by default) are loosely-typed objects since their schema is defined by the columns returned by the database query used to populate them. To access a particular column from a loosely-typed DataTable we need to use syntax like: `DataTable.Rows(index)("columnName")`. The DataTable's loose typing in this example is exhibited by the fact that we need to access the column name using a string or ordinal index. A strongly-typed DataTable, on the other hand, will have each of its columns implemented as properties, resulting in code that looks like: `DataTable.Rows(index).columnName`.

To return strongly-typed objects, developers can either create their own custom business objects or use Typed DataSets. A business object is implemented by the developer as a class whose properties typically reflect the columns of the underlying database table the business object represents. A Typed DataSet is a class generated for you by Visual Studio based on a database schema and whose members are strongly-typed according to this schema. The Typed DataSet itself consists of classes that extend the ADO.NET DataSet, DataTable, and DataRow classes. In addition to strongly-typed DataTables, Typed DataSets now also include TableAdapters, which are classes with methods for populating the DataSet's DataTables and propagating modifications within the DataTables back to the database.

> **NOTE**
>
> For more information on the advantages and disadvantages of using Typed DataSets versus custom business objects, refer to Designing Data Tier Components and Passing Data Through Tiers.

We'll use strongly-typed DataSets for these tutorials' architecture. Figure 3 illustrates the workflow between the different layers of an application that uses Typed DataSets.

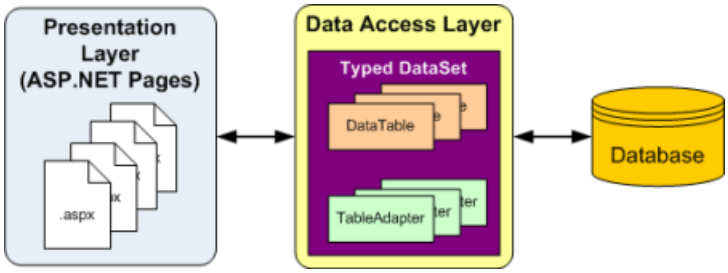


Figure 3: All Data Access Code is Relegated to the DAL (Click to view full-size image)

Creating a Typed DataSet and Table Adapter

To begin creating our DAL, we start by adding a Typed DataSet to our project. To accomplish this, right-click on the project node in the Solution Explorer and choose Add a New Item. Select the DataSet option from the list of templates and name it `Northwind.xsd`.

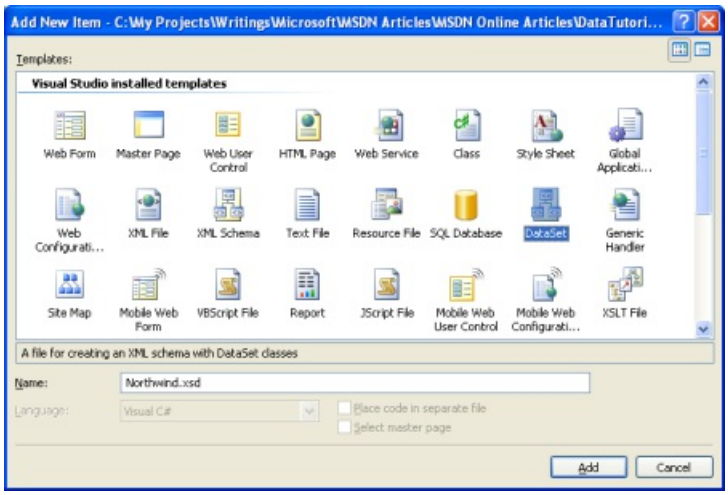


Figure 4: Choose to Add a New DataSet to Your Project (Click to view full-size image)

After clicking Add, when prompted to add the DataSet to the `App_Code` folder, choose Yes. The Designer for the Typed DataSet will then be displayed, and the TableAdapter Configuration Wizard will start, allowing you to add your first TableAdapter to the Typed DataSet.

A Typed DataSet serves as a strongly-typed collection of data; it is composed of strongly-typed DataTable instances, each of which is in turn composed of strongly-typed DataRow instances. We will create a strongly-typed DataTable for each of the underlying database tables that we need to work with in this tutorials series. Let's start with creating a DataTable for the `Products` table.

Keep in mind that strongly-typed DataTables do not include any information on how to access data from their underlying database table. In order to retrieve the data to populate the DataTable, we use a TableAdapter class, which functions as our Data Access Layer. For our `Products` DataTable, the TableAdapter will contain the methods `GetProducts()`, `GetProductByCategoryID(categoryID)`, and so on that we'll invoke from the presentation layer. The DataTable's role is to serve as the strongly-typed objects used to pass data between the layers.

The TableAdapter Configuration Wizard begins by prompting you to select which database to work with. The drop-down list shows those databases in the Server Explorer. If you did not add the Northwind database to the Server Explorer, you can click the New Connection button at this time to do so.

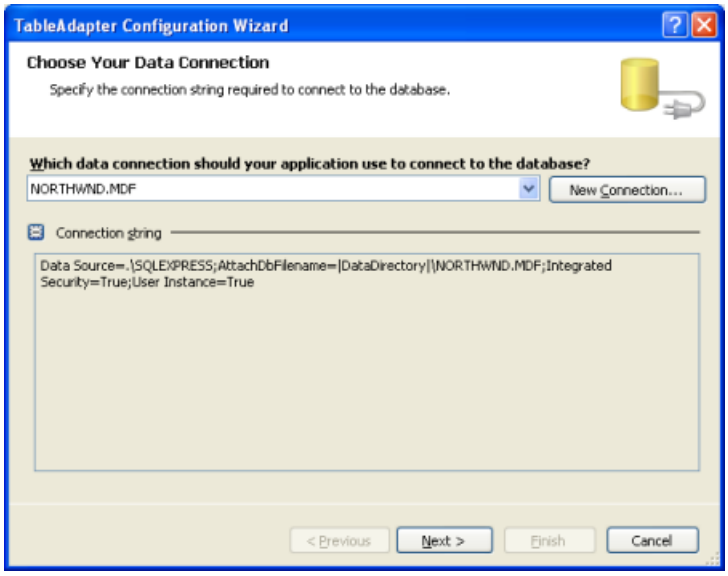


Figure 5: Choose the Northwind Database from the Drop-Down List (Click to view full-size image)

After selecting the database and clicking Next, you'll be asked if you want to save the connection string in the `Web.config` file. By saving the connection string you'll avoid having it hard coded in the TableAdapter classes, which simplifies things if the connection string information changes in the future. If you opt to save the

connection string in the configuration file it's placed in the `<connectionStrings>` section, which can be optionally encrypted for improved security or modified later through the new ASP.NET 2.0 Property Page within the IIS GUI Admin Tool, which is more ideal for administrators.

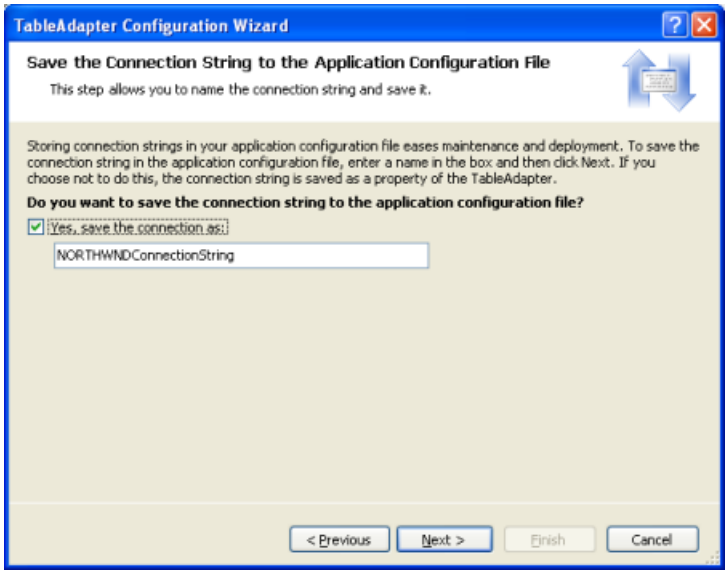


Figure 6: Save the Connection String to `Web.config` (Click to view full-size image)

Next, we need to define the schema for the first strongly-typed DataTable and provide the first method for our TableAdapter to use when populating the strongly-typed DataSet. These two steps are accomplished simultaneously by creating a query that returns the columns from the table that we want reflected in our DataTable. At the end of the wizard we'll give a method name to this query. Once that's been accomplished, this method can be invoked from our presentation layer. The method will execute the defined query and populate a strongly-typed DataTable.

To get started defining the SQL query we must first indicate how we want the TableAdapter to issue the query. We can use an ad-hoc SQL statement, create a new stored procedure, or use an existing stored procedure. For these tutorials we'll use ad-hoc SQL statements. Refer to Brian Noyes's article, Build a Data Access Layer with the Visual Studio 2005 DataSet Designer for an example of using stored procedures.

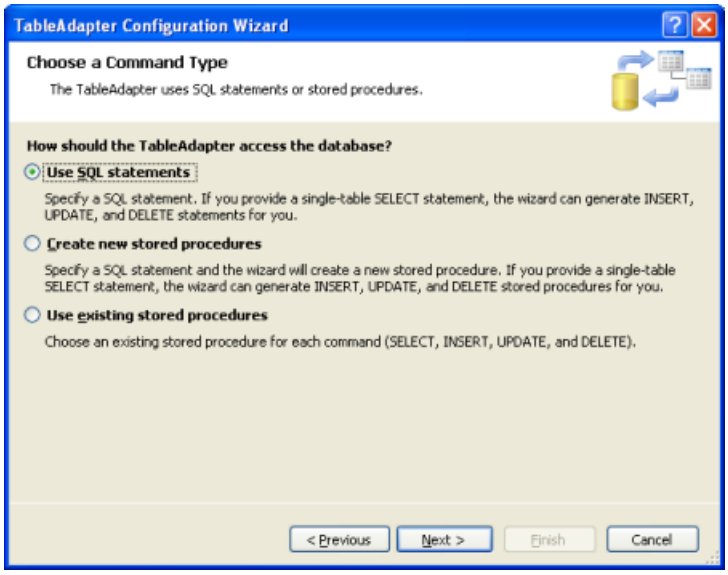


Figure 7: Query the Data Using an Ad-Hoc SQL Statement (Click to view full-size image)

At this point we can type in the SQL query by hand. When creating the first method in the TableAdapter you typically want to have the query return those columns that need to be expressed in the corresponding DataTable. We can accomplish this by creating a query that returns all columns and all rows from the `Products` table:

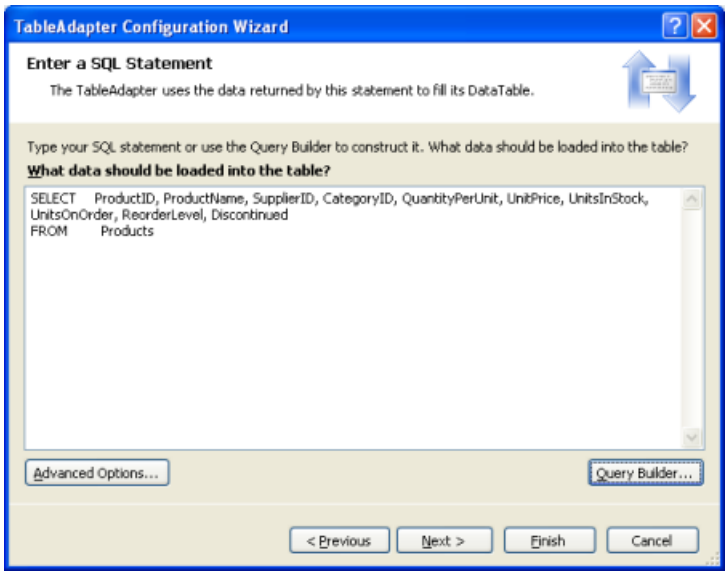


Figure 8: Enter the SQL Query Into the Textbox (Click to view full-size image)

Alternatively, use the Query Builder and graphically construct the query, as shown in Figure 9.

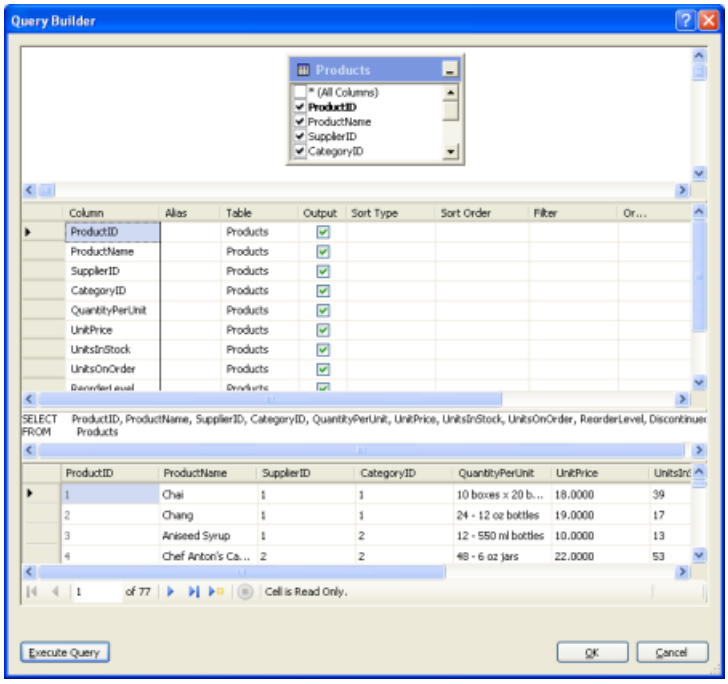


Figure 9: Create the Query Graphically, through the Query Editor (Click to view full-size image)

After creating the query, but before moving onto the next screen, click the Advanced Options button. In Web Site Projects, "Generate Insert, Update, and Delete statements" is the only advanced option selected by default; if you run this wizard from a Class Library or a Windows Project the "Use optimistic concurrency" option will also be selected. Leave the "Use optimistic concurrency" option unchecked for now. We'll examine optimistic concurrency in future tutorials.

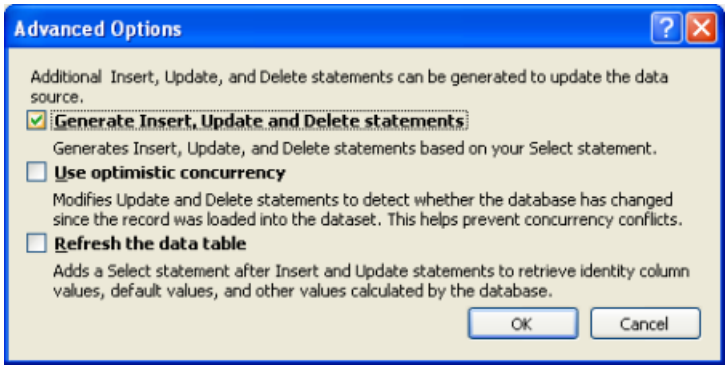


Figure 10: Select Only the Generate Insert, Update, and Delete statements Option (Click to view full-size image)

After verifying the advanced options, click Next to proceed to the final screen. Here we are asked to select which methods to add to the TableAdapter. There are two patterns for populating data:

- **Fill a DataTable** with this approach a method is created that takes in a DataTable as a parameter and populates it based on the results of the query. The ADO.NET DataAdapter class, for example, implements this pattern with its `Fill()` method.
- **Return a DataTable** with this approach the method creates and fills the DataTable for you and returns it as the methods return value.

You can have the TableAdapter implement one or both of these patterns. You can also rename the methods provided here. Let's leave both checkboxes checked, even though we'll only be using the latter pattern throughout these tutorials. Also, let's rename the rather generic `GetData` method to `GetProducts`.

If checked, the final checkbox, "GenerateDBDirectMethods," creates `Insert()`, `Update()`, and `Delete()` methods for the TableAdapter. If you leave this option unchecked, all updates will need to be done through the TableAdapter's sole `Update()` method, which takes in the Typed DataSet, a DataTable, a single DataRow, or an array of DataRows. (If you've unchecked the "Generate Insert, Update, and Delete statements" option from the advanced properties in Figure 9 this checkbox's setting will have no effect.) Let's leave this checkbox selected.

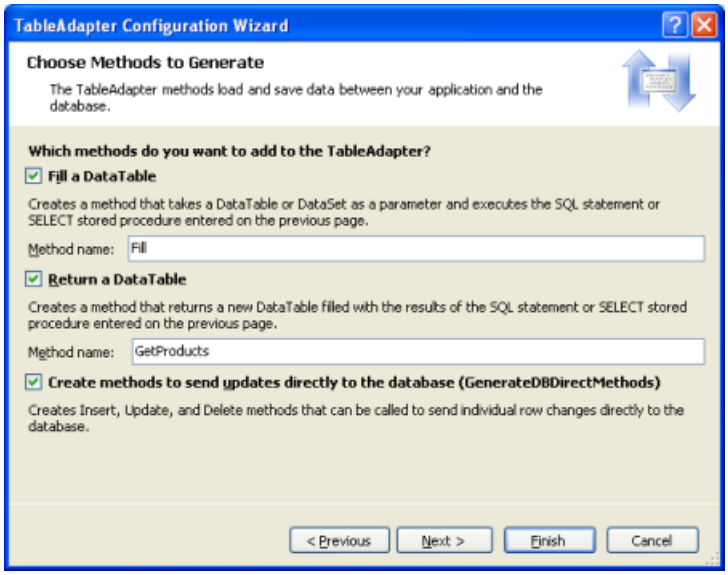


Figure 11: Change the Method Name from `GetData` to `GetProducts` (Click to view full-size image)

Complete the wizard by clicking Finish. After the wizard closes we are returned to the DataSet Designer which shows the DataTable we just created. You can see the list of columns in the `Products` DataTable (`ProductID`, `ProductName`, and so on), as well as the methods of the `ProductsTableAdapter` (`Fill()` and `GetProducts()`).

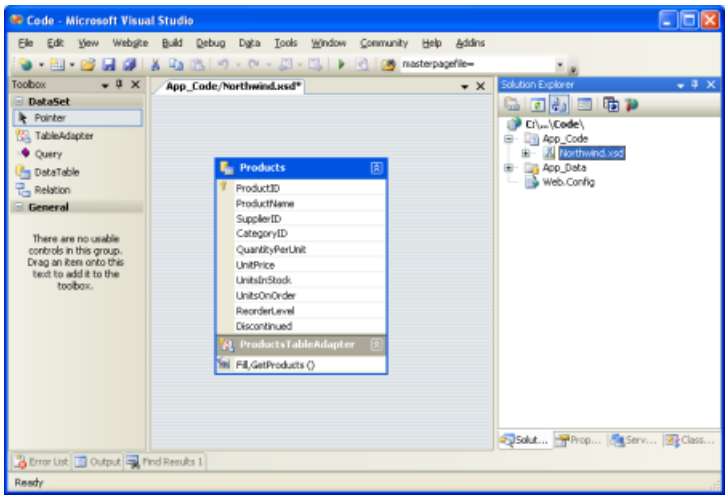


Figure 12: The `Products` DataTable and `ProductsTableAdapter` have been Added to the Typed DataSet (Click to view full-size image)

At this point we have a Typed DataSet with a single DataTable (`Northwind.Products`) and a strongly-typed DataAdapter class (`NorthwindTableAdapters.ProductsTableAdapter`) with a `GetProducts()` method. These objects can be used to access a list of all products from code like:

```
Dim productsAdapter As New NorthwindTableAdapters.ProductsTableAdapter()
Dim products as Northwind.ProductsDataTable
products = productsAdapter.GetProducts()
For Each productRow As Northwind.ProductsRow In products
    Response.Write("Product: " & productRow.ProductName & "<br />")
Next
```

This code did not require us to write one bit of data access-specific code. We did not have to instantiate any ADO.NET classes, we didn't have to refer to any connection strings, SQL queries, or stored procedures. Instead, the TableAdapter provides the low-level data access code for us.

Each object used in this example is also strongly-typed, allowing Visual Studio to provide IntelliSense and compile-time type checking. And best of all the DataTables returned by the TableAdapter can be bound to ASP.NET data Web controls, such as the GridView, DetailsView, DropDownList, CheckBoxList, and several others. The following example illustrates binding the DataTable returned by the `GetProducts()` method to a GridView in just a scant three lines of code within the `Page_Load` event handler.

AllProducts.aspx

```
<%@ Page Language="VB" AutoEventWireup="true" CodeFile="AllProducts.aspx.vb"
    Inherits="AllProducts" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>View All Products in a GridView</title>
    <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <h1>
            All Products</h1>
        <p>
            <asp:GridView ID="GridView1" runat="server"
             CssClass="DataWebControlStyle">
               <HeaderStyle CssClass="HeaderStyle" />
               <AlternatingRowStyle CssClass="AlternatingRowStyle" />
            </asp:GridView>
             </p>
    </div>
    </form>
</body>
</html>
```

AllProducts.aspx.vb

```
Imports NorthwindTableAdapters
Partial Class AllProducts
    Inherits System.Web.UI.Page
    Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
        Handles Me.Load
        Dim productsAdapter As New ProductsTableAdapter
        GridView1.DataSource = productsAdapter.GetProducts()
        GridView1.DataBind()
    End Sub
End Class
```

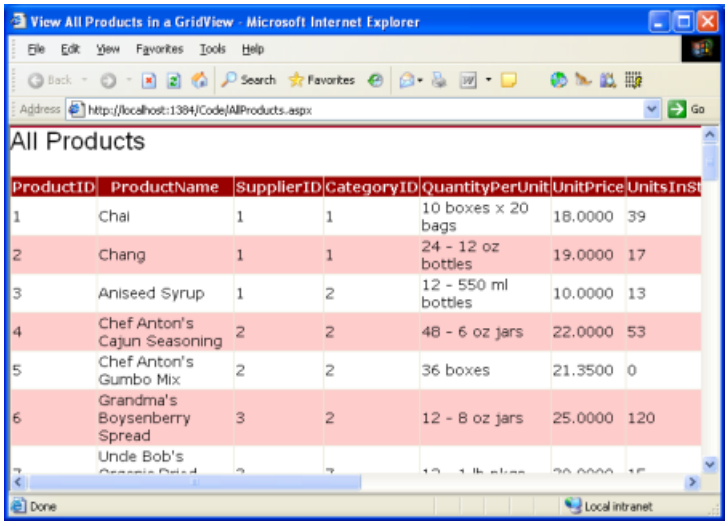


Figure 13: The List of Products is Displayed in a GridView (Click to view full-size image)

While this example required that we write three lines of code in our ASP.NET page's `Page_Load` event handler, in future tutorials we'll examine how to use the ObjectDataSource to declaratively retrieve the data from the DAL. With the ObjectDataSource we'll not have to write any code and will get paging and sorting support as well!

Step 3: Adding Parameterized Methods to the Data Access Layer

At this point our `ProductsTableAdapter` class has but one method, `GetProducts()`, which returns all of the products in the database. While being able to work with all products is definitely useful, there are times when we'll want to retrieve information about a specific product, or all products that belong to a particular category. To add such functionality to our Data Access Layer we can add parameterized methods to the TableAdapter.

Let's add the `GetProductsByCategoryID(categoryID)` method. To add a new method to the DAL, return to the DataSet Designer, right-click in the `ProductsTableAdapter` section, and choose Add Query.

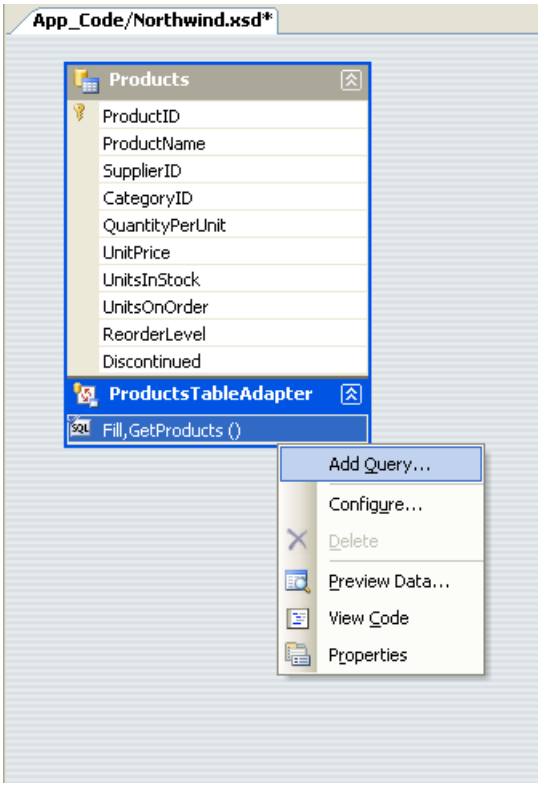


Figure 14: Right-Click on the TableAdapter and Choose Add Query

We are first prompted about whether we want to access the database using an ad-hoc SQL statement or a new or existing stored procedure. Let's choose to use an ad-hoc SQL statement again. Next, we are asked what type of SQL query we'd like to use. Since we want to return all products that belong to a specified category, we want to write a `SELECT` statement which returns rows.

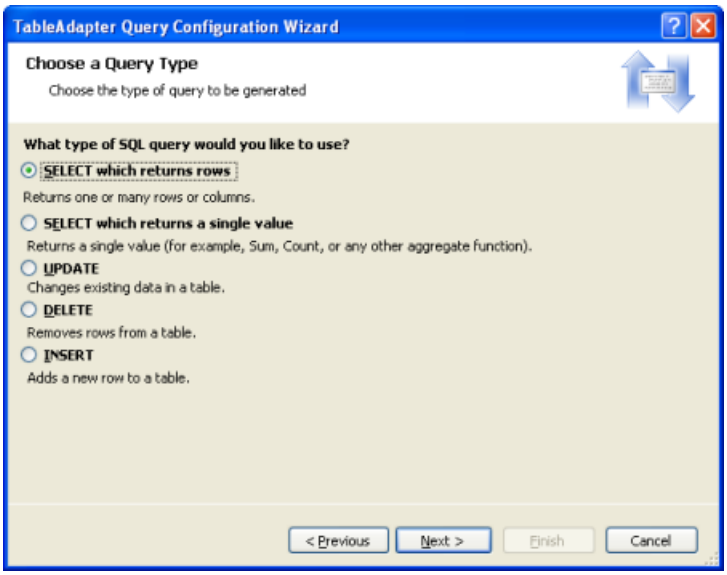


Figure 15: Choose to Create a `SELECT` Statement Which Returns Rows (Click to view full-size image)

The next step is to define the SQL query used to access the data. Since we want to return only those products that belong to a particular category, I use the same `SELECT` statement from `GetProducts()`, but add the following `WHERE` clause: `WHERE CategoryID = @CategoryID`. The `@CategoryID` parameter indicates to the TableAdapter wizard that the method we're creating will require an input parameter of the corresponding type (namely, a nullable integer).

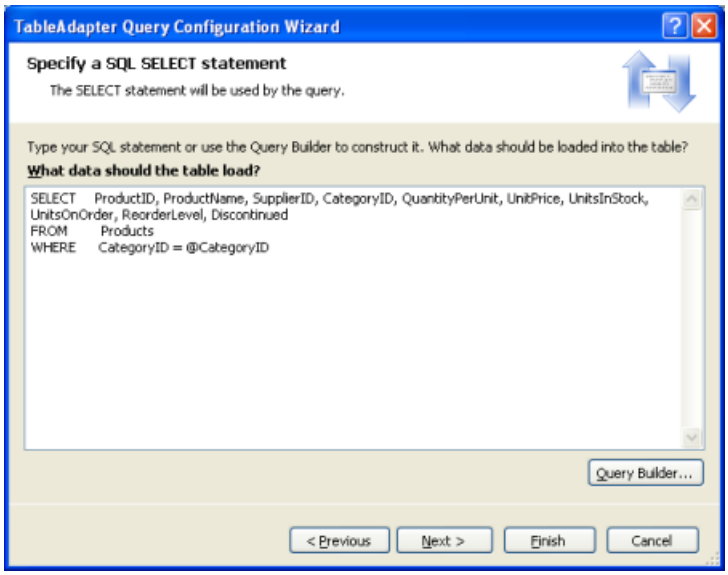


Figure 16: Enter a Query to Only Return Products in a Specified Category (Click to view full-size image)

In the final step we can choose which data access patterns to use, as well as customize the names of the methods generated. For the Fill pattern, let's change the name to `FillByCategoryID` and for the return a DataTable return pattern (the `GetX` methods), let's use `GetProductsByCategoryID`.

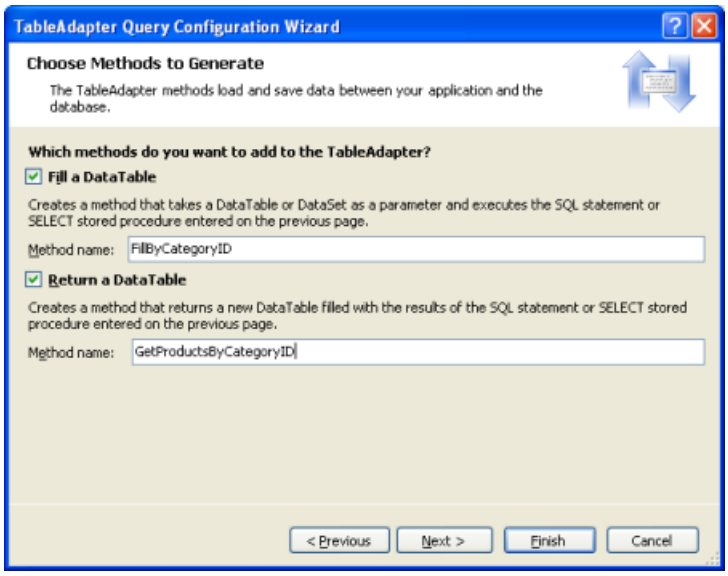


Figure 17: Choose the Names for the TableAdapter Methods (Click to view full-size image)

After completing the wizard, the DataSet Designer includes the new TableAdapter methods.

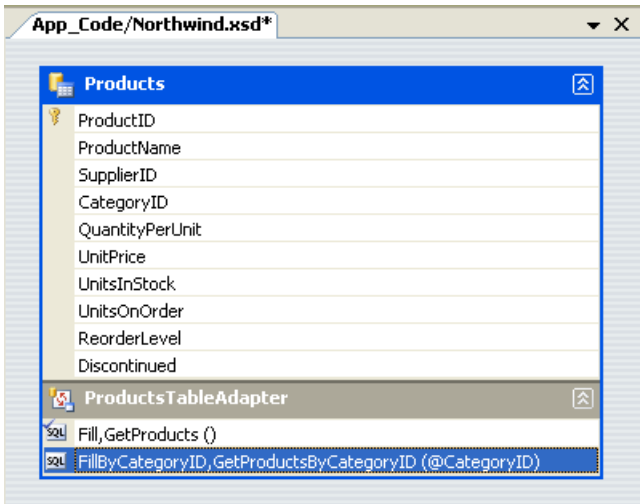


Figure 18: The Products Can Now be Queried by Category

Take a moment to add a `GetProductByProductID(productID)` method using the same technique.

These parameterized queries can be tested directly from the DataSet Designer. Right-click on the method in the TableAdapter and choose Preview Data. Next, enter the values to use for the parameters and click Preview.

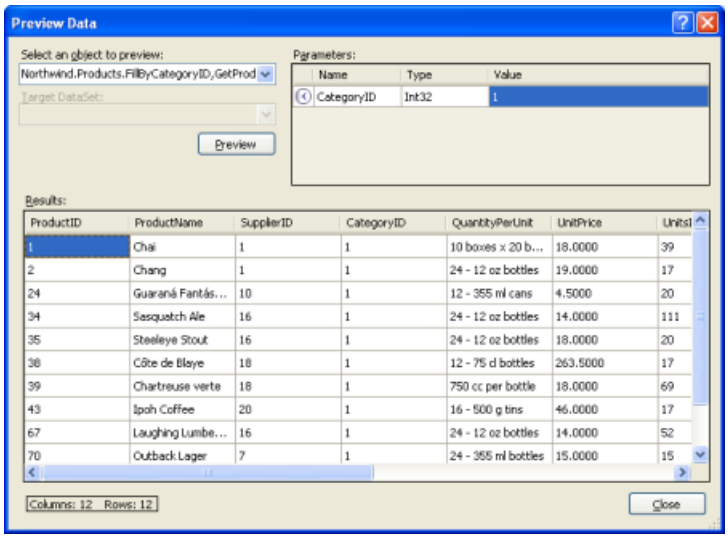


Figure 19: Those Products Belonging to the Beverages Category are Shown (Click to view full-size image)

With the `GetProductsByCategoryID(categoryID)` method in our DAL, we can now create an ASP.NET page that displays only those products in a specified category. The following example shows all products that are in the Beverages category, which have a `CategoryID` of 1.

Beverages.aspx

```
<%@ Page Language="VB" AutoEventWireup="true" CodeFile="Beverages.aspx.vb"
    Inherits="Beverages" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
    <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <h1>Beverages</h1>
        <p>
            <asp:GridView ID="GridView1" runat="server"
             CssClass="DataWebControlStyle">
               <HeaderStyle CssClass="HeaderStyle" />
               <AlternatingRowStyle CssClass="AlternatingRowStyle" />
            </asp:GridView>
             </p>
    </div>
    </form>
</body>
</html>
```

Beverages.aspx.vb

```
Imports NorthwindTableAdapters
Partial Class Beverages
    Inherits System.Web.UI.Page
    Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
        Handles Me.Load
        Dim productsAdapter As New ProductsTableAdapter
        GridView1.DataSource =
         productsAdapter.GetProductsByCategoryID(1)
        GridView1.DataBind()
    End Sub
End Class
```

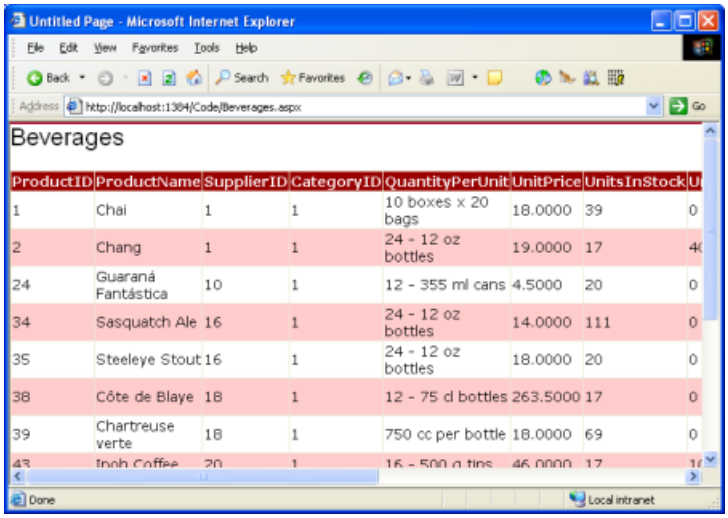


Figure 20: Those Products in the Beverages Category are Displayed (Click to view full-size image)

Step 4: Inserting, Updating, and Deleting Data

There are two patterns commonly used for inserting, updating, and deleting data. The first pattern, which I'll call the database direct pattern, involves creating methods that, when invoked, issue an `INSERT`, `UPDATE`, or

`DELETE` command to the database that operates on a single database record. Such methods are typically passed in a series of scalar values (integers, strings, Booleans, DateTimes, and so on) that correspond to the values to insert, update, or delete. For example, with this pattern for the `Products` table the delete method would take in an integer parameter, indicating the `ProductID` of the record to delete, while the insert method would take in a string for the `ProductName`, a decimal for the `UnitPrice`, an integer for the `UnitsOnStock`, and so on.

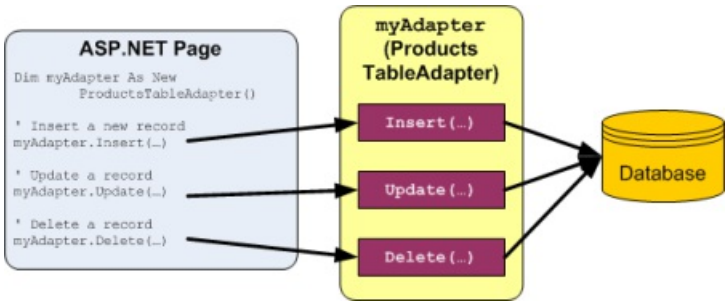


Figure 21: Each Insert, Update, and Delete Request is Sent to the Database Immediately (Click to view full-size image)

The other pattern, which I'll refer to as the batch update pattern, is to update an entire DataSet, DataTable, or collection of DataRows in one method call. With this pattern a developer deletes, inserts, and modifies the DataRows in a DataTable and then passes those DataRows or DataTable into an update method. This method then enumerates the DataRows passed in, determines whether or not they've been modified, added, or deleted (via the DataRow's RowState property value), and issues the appropriate database request for each record.

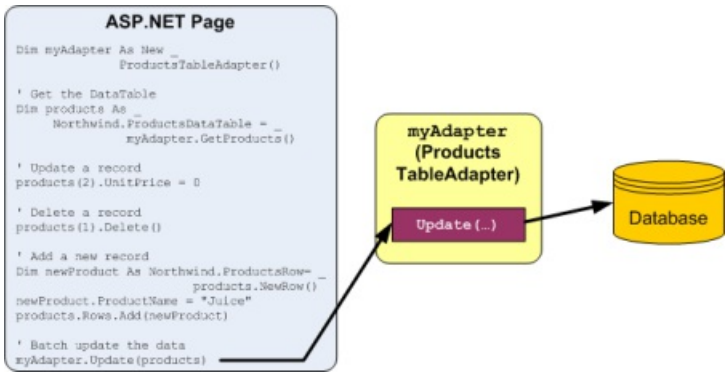


Figure 22: All Changes are Synchronized with the Database When the Update Method is Invoked (Click to view full-size image)

The TableAdapter uses the batch update pattern by default, but also supports the DB direct pattern. Since we selected the "Generate Insert, Update, and Delete statements" option from the Advanced Properties when creating our TableAdapter, the `ProductsTableAdapter` contains an `Update()` method, which implements the batch update pattern. Specifically, the TableAdapter contains an `Update()` method that can be passed the Typed DataSet, a strongly-typed DataTable, or one or more DataRows. If you left the "GenerateDBDirectMethods" checkbox checked when first creating the TableAdapter the DB direct pattern will also be implemented via `Insert()`, `Update()`, and `Delete()` methods.

Both data modification patterns use the TableAdapter's `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties to issue their `INSERT`, `UPDATE`, and `DELETE` commands to the database. You can inspect and modify the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties by clicking on the TableAdapter in the DataSet Designer and then going to the Properties window. (Make sure you have selected the TableAdapter, and that the `ProductsTableAdapter` object is the one selected in the drop-down list in the Properties window.)

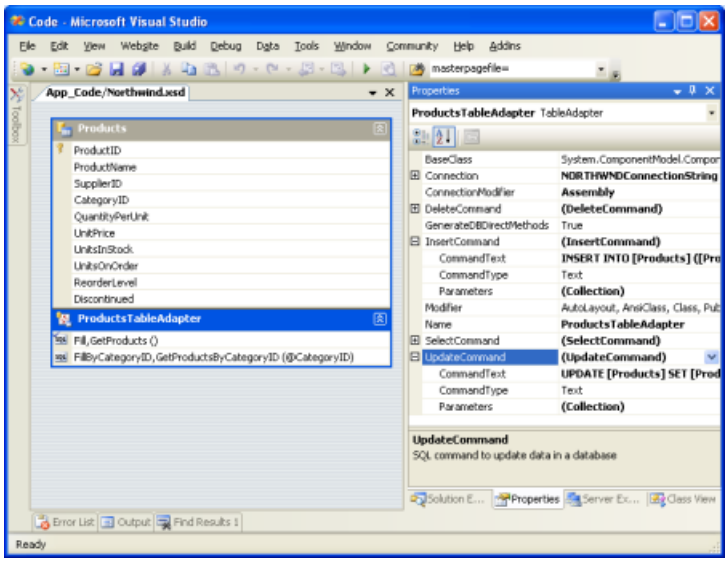


Figure 23: The TableAdapter has `InsertCommand`, `UpdateCommand`, and `DeleteCommand` Properties (Click to view full-size image)

To examine or modify any of these database command properties, click on the `CommandText` subproperty, which will bring up the Query Builder.

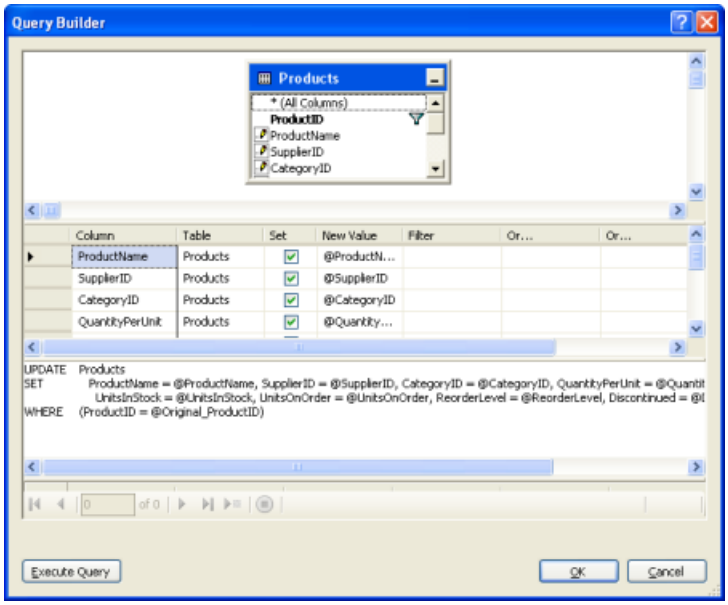


Figure 24: Configure the `INSERT`, `UPDATE`, and `DELETE` Statements in the Query Builder (Click to view full-size image)

The following code example shows how to use the batch update pattern to double the price of all products that are not discontinued and that have 25 units in stock or less:

```
Dim productsAdapter As New NorthwindTableAdapters.ProductsTableAdapter()
Dim products As Northwind.ProductsDataTable = productsAdapter.GetProducts()
For Each product As Northwind.ProductsRow In products
   If Not product.Discontinued AndAlso product.UnitsInStock <= 25 Then
      product.UnitPrice *= 2
   End if
Next
productsAdapter.Update(products)
```

The code below illustrates how to use the DB direct pattern to programmatically delete a particular product, then update one, and then add a new one:

```
Dim productsAdapter As New NorthwindTableAdapters.ProductsTableAdapter()
productsAdapter.Delete(3)
productsAdapter.Update( _
    "Chai", 1, 1, "10 boxes x 20 bags", 18.0, 39, 15, 10, false, 1)
productsAdapter.Insert( _
    "New Product", 1, 1, "12 tins per carton", 14.95, 15, 0, 10, false)
```

Creating Custom Insert, Update, and Delete Methods

The `Insert()`, `Update()`, and `Delete()` methods created by the DB direct method can be a bit cumbersome, especially for tables with many columns. Looking at the previous code example, without IntelliSense's help it's not particularly clear what `Products` table column maps to each input parameter to the `Update()` and `Insert()` methods. There may be times when we only want to update a single column or two, or want a customized `Insert()` method that will, perhaps, return the value of the newly inserted record's `IDENTITY` (auto-increment) field.

To create such a custom method, return to the DataSet Designer. Right-click on the TableAdapter and choose Add Query, returning to the TableAdapter wizard. On the second screen we can indicate the type of query to create. Let's create a method that adds a new product and then returns the value of the newly added record's `ProductID`. Therefore, opt to create an `INSERT` query.

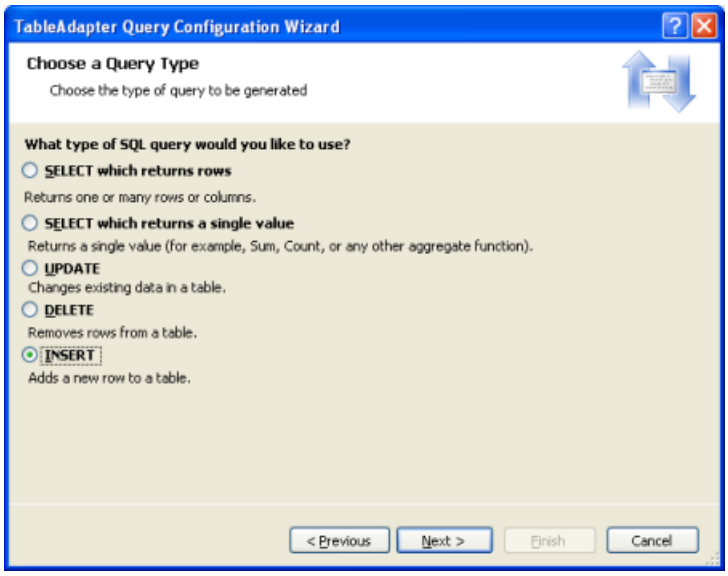


Figure 25: Create a Method to Add a New Row to the `Products` Table (Click to view full-size image)

On the next screen the `InsertCommand`'s `CommandText` appears. Augment this query by adding `SELECT SCOPE_IDENTITY()` at the end of the query, which will return the last identity value inserted into an `IDENTITY` column in the same scope. (See the technical documentation for more information about `SCOPE_IDENTITY()` and why you probably want to use SCOPE_IDENTITY() in lieu of @@IDENTITY.) Make sure that you end the `INSERT` statement with a semi-colon before adding the `SELECT` statement.

Figure 26: Augment the Query to Return the `SCOPE_IDENTITY()` Value (Click to view full-size image)

Finally, name the new method `InsertProduct`.

Figure 27: Set the New Method Name to `InsertProduct` (Click to view full-size image)

When you return to the DataSet Designer you'll see that the `ProductsTableAdapter` contains a new method, `InsertProduct`. If this new method doesn't have a parameter for each column in the `Products` table, chances are you forgot to terminate the `INSERT` statement with a semi-colon. Configure the `InsertProduct` method and ensure you have a semi-colon delimiting the `INSERT` and `SELECT` statements.

By default, insert methods issue non-query methods, meaning that they return the number of affected rows. However, we want the `InsertProduct` method to return the value returned by the query, not the number of rows affected. To accomplish this, adjust the `InsertProduct` method's `ExecuteMode` property to `Scalar`.

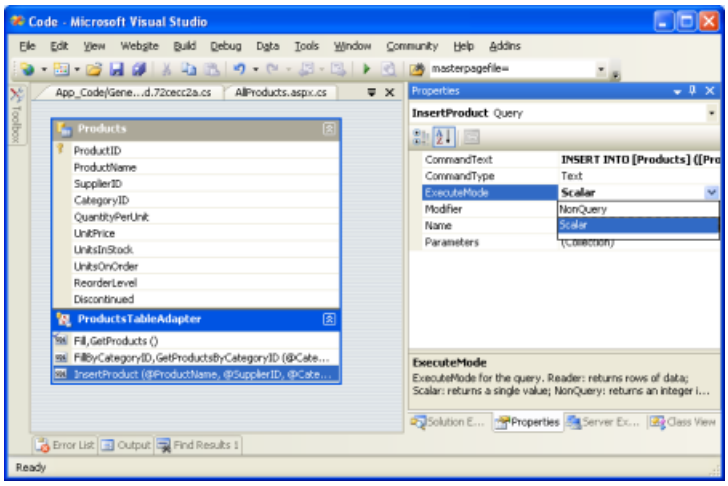


Figure 28: Change the `ExecuteMode` Property to `Scalar` (Click to view full-size image)

The following code shows this new `InsertProduct` method in action:

```
Dim productsAdapter As New NorthwindTableAdapters.ProductsTableAdapter()
Dim new_productID As Integer = Convert.ToInt32(productsAdapter.InsertProduct( _
    "New Product", 1, 1, "12 tins per carton", 14.95, 10, 0, 10, false))
productsAdapter.Delete(new_productID)
```

Step 5: Completing the Data Access Layer

Note that the `ProductsTableAdapters` class returns the `CategoryID` and `SupplierID` values from the `Products` table, but doesn't include the `CategoryName` column from the `Categories` table or the `CompanyName` column from the `Suppliers` table, although these are likely the columns we want to display when showing product information. We can augment the TableAdapter's initial method, `GetProducts()`, to include both the `CategoryName` and `CompanyName` column values, which will update the strongly-typed DataTable to include these new columns as well.

This can present a problem, however, as the TableAdapter's methods for inserting, updating, and deleting data are based off of this initial method. Fortunately, the auto-generated methods for inserting, updating, and deleting are not affected by subqueries in the `SELECT` clause. By taking care to add our queries to `Categories` and `Suppliers` as subqueries, rather than `JOIN` s, we'll avoid having to rework those methods for modifying data. Right-click on the `GetProducts()` method in the `ProductsTableAdapter` and choose Configure. Then, adjust the `SELECT` clause so that it looks like:

```
SELECT     ProductID, ProductName, SupplierID, CategoryID,
QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued,
(SELECT CategoryName FROM Categories
WHERE Categories.CategoryID = Products.CategoryID) as CategoryName,
(SELECT CompanyName FROM Suppliers
WHERE Suppliers.SupplierID = Products.SupplierID) as SupplierName
FROM         Products
```

Figure 29: Update the `SELECT` Statement for the `GetProducts()` Method (Click to view full-size image)

After updating the `GetProducts()` method to use this new query the DataTable will include two new columns: `CategoryName` and `SupplierName`.

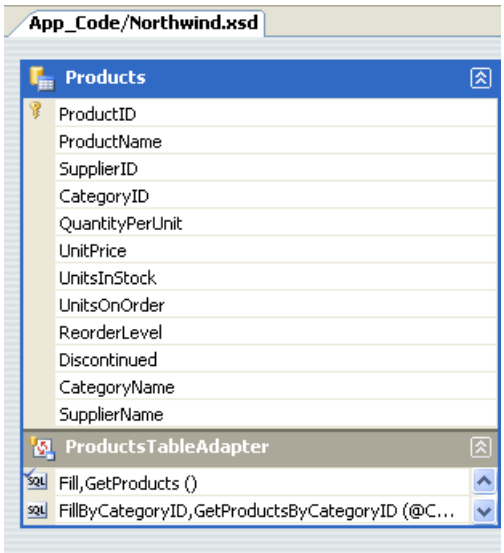


Figure 30: The `Products` DataTable has Two New Columns

Take a moment to update the `SELECT` clause in the `GetProductsByCategoryID(categoryID)` method as well.

If you update the `GetProducts()` `SELECT` using `JOIN` syntax the DataSet Designer won't be able to auto-generate the methods for inserting, updating, and deleting database data using the DB direct pattern. Instead, you'll have to manually create them much like we did with the `InsertProduct` method earlier in this tutorial. Furthermore, you'll manually have to provide the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` property values if you want to use the batch updating pattern.

Adding the Remaining TableAdapters

Up until now, we've only looked at working with a single TableAdapter for a single database table. However, the Northwind database contains several related tables that we'll need to work with in our web application. A Typed DataSet can contain multiple, related DataTables. Therefore, to complete our DAL we need to add DataTables for the other tables we'll be using in these tutorials. To add a new TableAdapter to a Typed DataSet, open the DataSet Designer, right-click in the Designer, and choose Add / TableAdapter. This will create a new DataTable and TableAdapter and walk you through the wizard we examined earlier in this tutorial.

Take a few minutes to create the following TableAdapters and methods using the following queries. Note that

the queries in the `ProductsTableAdapter` include the subqueries to grab each product's category and supplier names. Additionally, if you've been following along, you've already added the `ProductsTableAdapter` class's `GetProducts()` and `GetProductsByCategoryID(categoryID)` methods.

- ProductsTableAdapter
  - GetProducts:

```
SELECT     ProductID, ProductName, SupplierID,
CategoryID, QuantityPerUnit, UnitPrice, UnitsInStock,
UnitsOnOrder, ReorderLevel, Discontinued,
(SELECT CategoryName FROM Categories WHERE
Categories.CategoryID = Products.CategoryID) as
CategoryName, (SELECT CompanyName FROM Suppliers
WHERE Suppliers.SupplierID = Products.SupplierID)
as SupplierName
FROM         Products
```

  - GetProductsByCategoryID:

```
SELECT     ProductID, ProductName, SupplierID, CategoryID,
QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
ReorderLevel, Discontinued, (SELECT CategoryName
FROM Categories WHERE Categories.CategoryID =
Products.CategoryID) as CategoryName,
(SELECT CompanyName FROM Suppliers WHERE
Suppliers.SupplierID = Products.SupplierID)
as SupplierName
FROM         Products
WHERE      CategoryID = @CategoryID
```

  - GetProductsBySupplierID:

```
SELECT     ProductID, ProductName, SupplierID, CategoryID,
QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
ReorderLevel, Discontinued, (SELECT CategoryName
FROM Categories WHERE Categories.CategoryID =
Products.CategoryID) as CategoryName,
(SELECT CompanyName FROM Suppliers WHERE
Suppliers.SupplierID = Products.SupplierID) as SupplierName
FROM         Products
WHERE SupplierID = @SupplierID
```

  - GetProductByProductID:

```
SELECT     ProductID, ProductName, SupplierID, CategoryID,
QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
ReorderLevel, Discontinued, (SELECT CategoryName
FROM Categories WHERE Categories.CategoryID =
Products.CategoryID) as CategoryName,
(SELECT CompanyName FROM Suppliers WHERE Suppliers.SupplierID = Products.SupplierID)
as SupplierName
FROM         Products
WHERE ProductID = @ProductID
```

- CategoriesTableAdapter
  - GetCategories:

```
SELECT     CategoryID, CategoryName, Description
FROM         Categories
```

- GetCategoryByCategoryID:

```
SELECT     CategoryID, CategoryName, Description
FROM         Categories
WHERE CategoryID = @CategoryID
```

- SuppliersTableAdapter
  - GetSuppliers:

```
SELECT     SupplierID, CompanyName, Address,
City, Country, Phone
FROM         Suppliers
```

  - GetSuppliersByCountry:

```
SELECT     SupplierID, CompanyName, Address,
City, Country, Phone
FROM         Suppliers
WHERE Country = @Country
```

  - GetSupplierBySupplierID:

```
SELECT     SupplierID, CompanyName, Address,
City, Country, Phone
FROM         Suppliers
WHERE SupplierID = @SupplierID
```

- EmployeesTableAdapter
  - GetEmployees:

```
SELECT     EmployeeID, LastName, FirstName, Title,
HireDate, ReportsTo, Country
FROM         Employees
```

  - GetEmployeesByManager:

```
SELECT     EmployeeID, LastName, FirstName, Title,
HireDate, ReportsTo, Country
FROM         Employees
WHERE ReportsTo = @ManagerID
```

  - GetEmployeeByEmployeeID:

```
SELECT     EmployeeID, LastName, FirstName, Title,
HireDate, ReportsTo, Country
FROM         Employees
WHERE EmployeeID = @EmployeeID
```

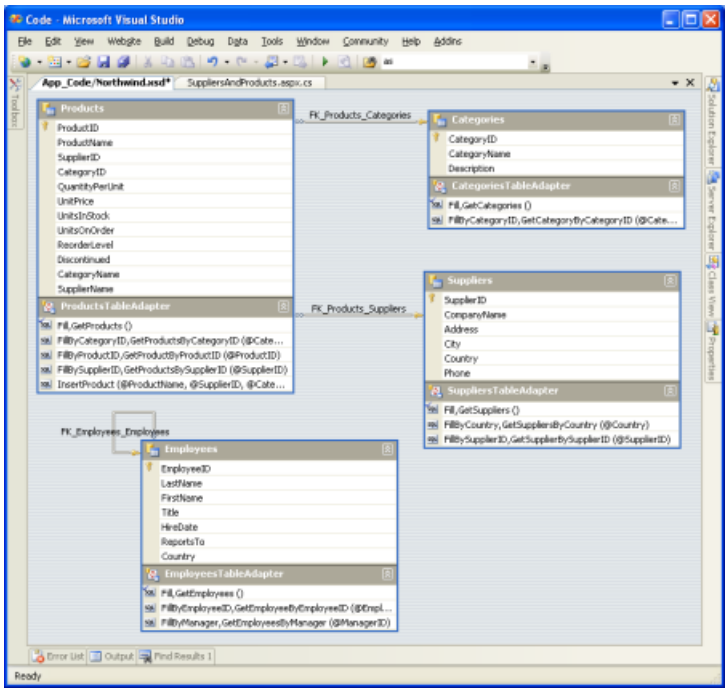


Figure 31: The DataSet Designer After the Four TableAdapters Have Been Added (Click to view full-size image)

Adding Custom Code to the DAL

The TableAdapters and DataTables added to the Typed DataSet are expressed as an XML Schema Definition file (`Northwind.xsd`). You can view this schema information by right-clicking on the `Northwind.xsd` file in the Solution Explorer and choosing View Code.

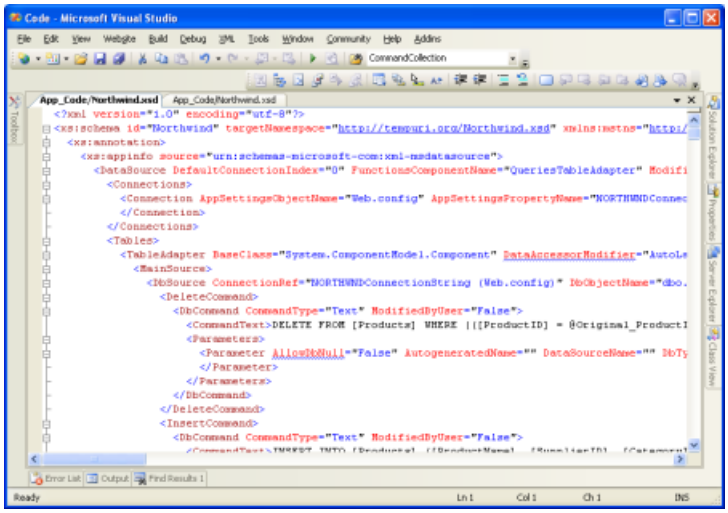


Figure 32: The XML Schema Definition (XSD) File for the Northwinds Typed DataSet (Click to view full-size image)

This schema information is translated into C# or Visual Basic code at design time when compiled or at runtime (if needed), at which point you can step through it with the debugger. To view this auto-generated code go to the Class View and drill down to the TableAdapter or Typed DataSet classes. If you don't see the Class View on your screen, go to the View menu and select it from there, or hit Ctrl+Shift+C. From the Class View you can see the properties, methods, and events of the Typed DataSet and TableAdapter classes. To view the code for a particular method, double-click the method name in the Class View or right-click on it and choose Go To Definition.

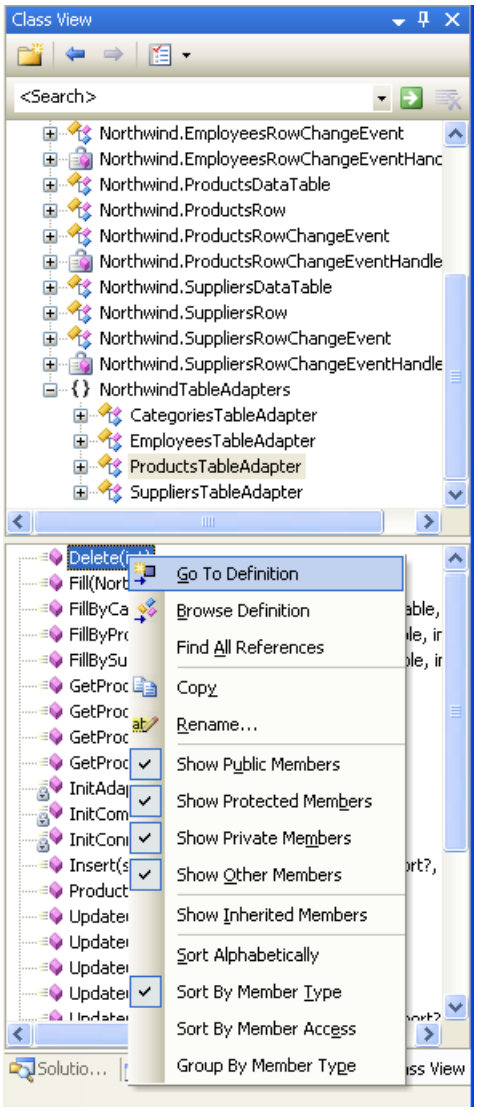


Figure 33: Inspect the Auto-Generated Code by Selecting Go To Definition from the Class View

While auto-generated code can be a great time saver, the code is often very generic and needs to be customized to meet the unique needs of an application. The risk of extending auto-generated code, though, is that the tool that generated the code might decide it's time to "regenerate" and overwrite your customizations. With .NET 2.0's new partial class concept, it's easy to split a class across multiple files. This enables us to add our own methods, properties, and events to the auto-generated classes without having to worry about Visual Studio overwriting our customizations.

To demonstrate how to customize the DAL, let's add a `GetProducts()` method to the `SuppliersRow` class. The `SuppliersRow` class represents a single record in the `Suppliers` table; each supplier can provider zero to many products, so `GetProducts()` will return those products of the specified supplier. To accomplish this create a new class file in the `App_Code` folder named `SuppliersRow.vb` and add the following code:

```
Imports NorthwindTableAdapters
Partial Public Class Northwind
    Partial Public Class SuppliersRow
        Public Function GetProducts() As Northwind.ProductsDataTable
            Dim productsAdapter As New ProductsTableAdapter
            Return productsAdapter.GetProductsBySupplierID(Me.SupplierID)
        End Function
    End Class
End Class
```

This partial class instructs the compiler that when building the `Northwind.SuppliersRow` class to include the `GetProducts()` method we just defined. If you build your project and then return to the Class View you'll see

`GetProducts()` now listed as a method of `Northwind.SuppliersRow`.

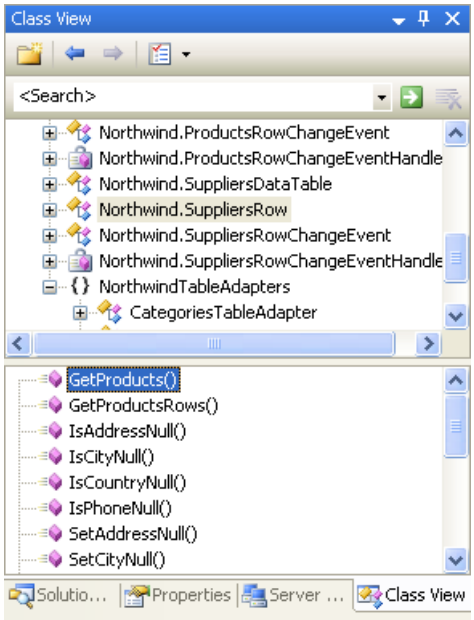


Figure 34: The `GetProducts()` Method is Now Part of the `Northwind.SuppliersRow` Class

The `GetProducts()` method can now be used to enumerate the set of products for a particular supplier, as the following code shows:

```
Dim suppliersAdapter As New NorthwindTableAdapters.SuppliersTableAdapter()
Dim suppliers As Northwind.SuppliersDataTable = suppliersAdapter.GetSuppliers()
For Each supplier As Northwind.SuppliersRow In suppliers
    Response.Write("Supplier: " & supplier.CompanyName)
    Response.Write("<ul>")
    Dim products As Northwind.ProductsDataTable = supplier.GetProducts()
    For Each product As Northwind.ProductsRow In products
        Response.Write("<li>" & product.ProductName & "</li>")
    Next
    Response.Write("</ul><p> </p>")
Next
```

This data can also be displayed in any of ASP.NET's data Web controls. The following page uses a GridView control with two fields:

- A BoundField that displays the name of each supplier, and
- A TemplateField that contains a BulletedList control that is bound to the results returned by the `GetProducts()` method for each supplier.

We'll examine how to display such master-detail reports in future tutorials. For now, this example is designed to illustrate using the custom method added to the `Northwind.SuppliersRow` class.

SuppliersAndProducts.aspx

```
<%@ Page Language="VB" CodeFile="SuppliersAndProducts.aspx.vb"
    AutoEventWireup="true" Inherits="SuppliersAndProducts" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
    <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <h1>
            Suppliers and Their Products</h1>
        <p>
            <asp:GridView ID="GridView1" runat="server"
             AutoGenerateColumns="False"
             CssClass="DataWebControlStyle">
                <HeaderStyle CssClass="HeaderStyle" />
                <AlternatingRowStyle CssClass="AlternatingRowStyle" />
                <Columns>
                    <asp:BoundField DataField="CompanyName"
                      HeaderText="Supplier" />
                    <asp:TemplateField HeaderText="Products">
                        <ItemTemplate>
                            <asp:BulletedList ID="BulletedList1"
                             runat="server" DataSource="<%# CType(CType(Container.DataItem,
System.Data.DataRowView).Row, Northwind.SuppliersRow).GetProducts() %>"
                                 DataTextField="ProductName">
                            </asp:BulletedList>
                        </ItemTemplate>
                    </asp:TemplateField>
                </Columns>
            </asp:GridView>
             </p>
    </div>
    </form>
</body>
</html>
```

SuppliersAndProducts.aspx.vb

```
Imports NorthwindTableAdapters
Partial Class SuppliersAndProducts
    Inherits System.Web.UI.Page
    Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
        Handles Me.Load
        Dim suppliersAdapter As New SuppliersTableAdapter
        GridView1.DataSource = suppliersAdapter.GetSuppliers()
        GridView1.DataBind()
    End Sub
End Class
```

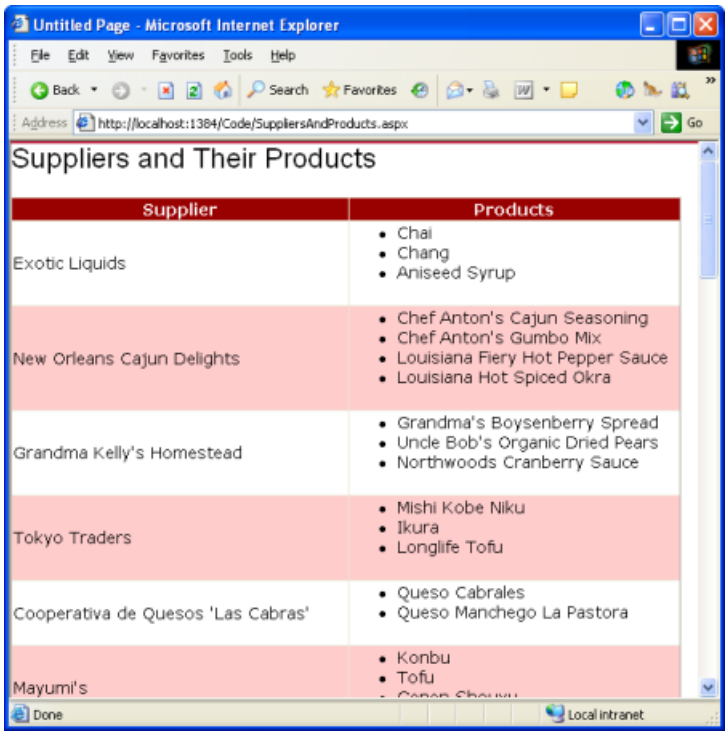


Figure 35: The Supplier's Company Name is Listed in the Left Column, Their Products in the Right (Click to view full-size image)

Summary

When building a web application creating the DAL should be one of your first steps, occurring before you start creating your presentation layer. With Visual Studio, creating a DAL based on Typed DataSets is a task that can be accomplished in 10-15 minutes without writing a line of code. The tutorials moving forward will build upon this DAL. In the next tutorial we'll define a number of business rules and see how to implement them in a separate Business Logic Layer.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Building a DAL using Strongly Typed TableAdapters and DataTables in VS 2005 and ASP.NET 2.0
- Designing Data Tier Components and Passing Data Through Tiers
- Build a Data Access Layer with the Visual Studio 2005 DataSet Designer
- Encrypting Configuration Information in ASP.NET 2.0 Applications
- TableAdapter Overview
- Working with a Typed DataSet
- Using Strongly-Typed Data Access in Visual Studio 2005 and ASP.NET 2.0
- How to Extend TableAdapter Methods
- Retrieving Scalar Data from a Stored Procedure

Video Training on Topics Contained in this Tutorial

- Data Access Layers in ASP.NET Applications
- How to Manually Bind a Dataset to a Datagrid
- How to Work with Datasets and Filters from an ASP Application

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Ron Green, Hilton Giesenow, Dennis Patterson, Liz Shulok, Abel Gomez, and Carlos Santos. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Creating a Business Logic Layer (VB)

2/20/2020 • 18 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll see how to centralize your business rules into a Business Logic Layer (BLL) that serves as an intermediary for data exchange between the presentation layer and the DAL.

Introduction

The Data Access Layer (DAL) created in the first tutorial cleanly separates the data access logic from the presentation logic. However, while the DAL cleanly separates the data access details from the presentation layer, it does not enforce any business rules that may apply. For example, for our application we may want to disallow the `CategoryID` or `SupplierID` fields of the `Products` table to be modified when the `Discontinued` field is set to 1, or we might want to enforce seniority rules, prohibiting situations in which an employee is managed by someone who was hired after them. Another common scenario is authorization perhaps only users in a particular role can delete products or can change the `UnitPrice` value.

In this tutorial we'll see how to centralize these business rules into a Business Logic Layer (BLL) that serves as an intermediary for data exchange between the presentation layer and the DAL. In a real-world application, the BLL should be implemented as a separate Class Library project; however, for these tutorials we'll implement the BLL as a series of classes in our `App_Code` folder in order to simplify the project structure. Figure 1 illustrates the architectural relationships among the presentation layer, BLL, and DAL.

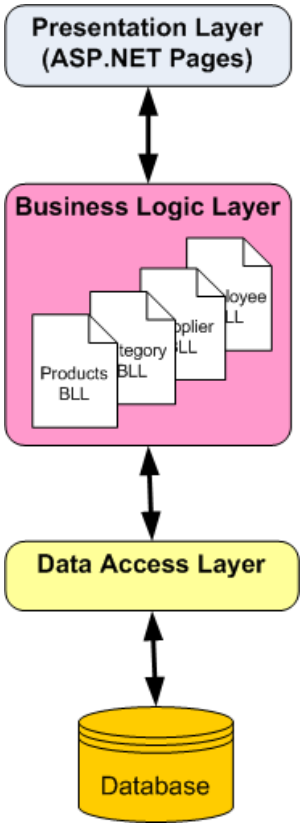


Figure 1: The BLL Separates the Presentation Layer from the Data Access Layer and Imposes Business Rules

Rather than creating separate classes to implement our business logic, we could alternatively place this logic

directly in the Typed DataSet with partial classes. For an example of creating and extending a Typed DataSet, refer back to the first tutorial.

Step 1: Creating the BLL Classes

Our BLL will be composed of four classes, one for each TableAdapter in the DAL; each of these BLL classes will have methods for retrieving, inserting, updating, and deleting from the respective TableAdapter in the DAL, applying the appropriate business rules.

To more cleanly separate the DAL- and BLL-related classes, let's create two subfolders in the `App_Code` folder, `DAL` and `BLL`. Simply right-click on the `App_Code` folder in the Solution Explorer and choose New Folder. After creating these two folders, move the Typed DataSet created in the first tutorial into the `DAL` subfolder.

Next, create the four BLL class files in the `BLL` subfolder. To accomplish this, right-click on the `BLL` subfolder, choose Add a New Item, and choose the Class template. Name the four classes `ProductsBLL`, `CategoriesBLL`, `SuppliersBLL`, and `EmployeesBLL`.

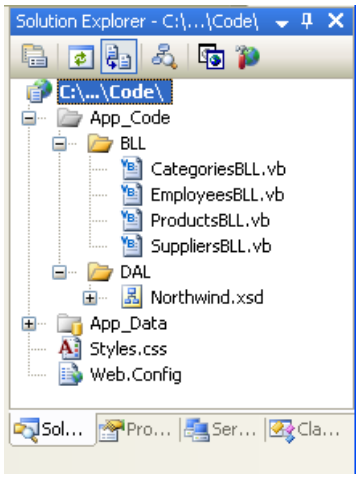


Figure 2: Add Four New Classes to the `App_Code` Folder

Next, let's add methods to each of the classes to simply wrap the methods defined for the TableAdapters from the first tutorial. For now, these methods will just call directly into the DAL; we'll return later to add any needed business logic.

> **NOTE**
>
> If you are using Visual Studio Standard Edition or above (that is, you're *not* using Visual Web Developer), you can optionally design your classes visually using the Class Designer. Refer to the Class Designer Blog for more information on this new feature in Visual Studio.

For the `ProductsBLL` class we need to add a total of seven methods:

- `GetProducts()` returns all products
- `GetProductByProductID(productID)` returns the product with the specified product ID
- `GetProductsByCategoryID(categoryID)` returns all products from the specified category
- `GetProductsBySupplier(supplierID)` returns all products from the specified supplier
- `AddProduct(productName, supplierID, categoryID, quantityPerUnit, unitPrice, unitsInStock, unitsOnOrder, reorderLevel, discontinued)` inserts a new product into the database using the values passed-in; returns the `ProductID` value of the newly inserted record
- `UpdateProduct(productName, supplierID, categoryID, quantityPerUnit, unitPrice, unitsInStock, unitsOnOrder, reorderLevel, discontinued, productID)` updates an existing product in the database using the passed-in values; returns `True` if precisely one row

was updated, `False` otherwise

- `DeleteProduct(productID)` deletes the specified product from the database

ProductsBLL.vb

```
Imports NorthwindTableAdapters

<System.ComponentModel.DataObject()> _
Public Class ProductsBLL

    Private _productsAdapter As ProductsTableAdapter = Nothing
    Protected ReadOnly Property Adapter() As ProductsTableAdapter
        Get
            If _productsAdapter Is Nothing Then
                _productsAdapter = New ProductsTableAdapter()
            End If

            Return _productsAdapter
        End Get
    End Property

    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Select, True)> _
    Public Function GetProducts() As Northwind.ProductsDataTable
        Return Adapter.GetProducts()
    End Function

    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Select, False)> _
    Public Function GetProductByProductID(ByVal productID As Integer) _
        As Northwind.ProductsDataTable
        Return Adapter.GetProductByProductID(productID)
    End Function

    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Select, False)> _
    Public Function GetProductsByCategoryID(ByVal categoryID As Integer) _
        As Northwind.ProductsDataTable
        Return Adapter.GetProductsByCategoryID(categoryID)
    End Function

    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Select, False)> _
    Public Function GetProductsBySupplierID(ByVal supplierID As Integer) _
        As Northwind.ProductsDataTable
        Return Adapter.GetProductsBySupplierID(supplierID)
    End Function

    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Insert, True)> _
    Public Function AddProduct( _
        productName As String, supplierID As Nullable(Of Integer), _
        categoryID As Nullable(Of Integer), quantityPerUnit As String, _
        unitPrice As Nullable(Of Decimal), unitsInStock As Nullable(Of Short), _
        unitsOnOrder As Nullable(Of Short), reorderLevel As Nullable(Of Short), _
        discontinued As Boolean) _
        As Boolean

        Dim products As New Northwind.ProductsDataTable()
        Dim product As Northwind.ProductsRow = products.NewProductsRow()

        product.ProductName = productName
        If Not supplierID.HasValue Then
            product.SetSupplierIDNull()
        Else
            product.SupplierID = supplierID.Value
        End If
```

```
        If Not categoryID.HasValue Then
            product.SetCategoryIDNull()
        Else
            product.CategoryID = categoryID.Value
        End If

        If quantityPerUnit Is Nothing Then
            product.SetQuantityPerUnitNull()
        Else
            product.QuantityPerUnit = quantityPerUnit
        End If

        If Not unitPrice.HasValue Then
            product.SetUnitPriceNull()
        Else
            product.UnitPrice = unitPrice.Value
        End If

        If Not unitsInStock.HasValue Then
            product.SetUnitsInStockNull()
        Else
            product.UnitsInStock = unitsInStock.Value
        End If

        If Not unitsOnOrder.HasValue Then
            product.SetUnitsOnOrderNull()
        Else
            product.UnitsOnOrder = unitsOnOrder.Value
        End If

        If Not reorderLevel.HasValue Then
            product.SetReorderLevelNull()
        Else
            product.ReorderLevel = reorderLevel.Value
        End If

        product.Discontinued = discontinued

        products.AddProductsRow(product)
        Dim rowsAffected As Integer = Adapter.Update(products)

        Return rowsAffected = 1
    End Function

    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Update, True)> _
    Public Function UpdateProduct(_
        productName As String, supplierID As Nullable(Of Integer), _
        categoryID As Nullable(Of Integer), quantityPerUnit As String, _
        unitPrice As Nullable(Of Decimal), unitsInStock As Nullable(Of Short), _
        unitsOnOrder As Nullable(Of Short), reorderLevel As Nullable(Of Short), _
        discontinued As Boolean, productID As Integer) _
        As Boolean

        Dim products As Northwind.ProductsDataTable = _
            Adapter.GetProductByProductID(productID)

        If products.Count = 0 Then
            Return False
        End If

        Dim product as Northwind.ProductsRow = products(0)

        product.ProductName = productName
        If Not supplierID.HasValue Then
            product.SetSupplierIDNull()
        Else
            product.SupplierID = supplierID.Value
```

```
        End If

        If Not categoryID.HasValue Then
            product.SetCategoryIDNull()
        Else
            product.CategoryID = categoryID.Value
        End If

        If quantityPerUnit Is Nothing Then
            product.SetQuantityPerUnitNull()
        Else
            product.QuantityPerUnit = quantityPerUnit
        End If

        If Not unitPrice.HasValue Then
            product.SetUnitPriceNull()
        Else
            product.UnitPrice = unitPrice.Value
        End If

        If Not unitsInStock.HasValue Then
            product.SetUnitsInStockNull()
        Else
            product.UnitsInStock = unitsInStock.Value
        End If

        If Not unitsOnOrder.HasValue Then
            product.SetUnitsOnOrderNull()
        Else
            product.UnitsOnOrder = unitsOnOrder.Value
        End If

        If Not reorderLevel.HasValue Then
            product.SetReorderLevelNull()
        Else
            product.ReorderLevel = reorderLevel.Value
        End If

        product.Discontinued = discontinued

        Dim rowsAffected As Integer = Adapter.Update(product)

        Return rowsAffected = 1
    End Function

    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Delete, True)> _
    Public Function DeleteProduct(ByVal productID As Integer) As Boolean
        Dim rowsAffected As Integer = Adapter.Delete(productID)

        Return rowsAffected = 1
    End Function
End Class
```

The methods that simply return data `GetProducts`, `GetProductByProductID`, `GetProductsByCategoryID`, and `GetProductBySuppliersID` are fairly straightforward as they simply call down into the DAL. While in some scenarios there may be business rules that need to be implemented at this level (such as authorization rules based on the currently logged on user or the role to which the user belongs), we'll simply leave these methods as-is. For these methods, then, the BLL serves merely as a proxy through which the presentation layer accesses the underlying data from the Data Access Layer.

The `AddProduct` and `UpdateProduct` methods both take in as parameters the values for the various product fields and add a new product or update an existing one, respectively. Since many of the `Product` table's columns can accept `NULL` values (`CategoryID`, `SupplierID`, and `UnitPrice`, to name a few), those input parameters for

`AddProduct` and `UpdateProduct` that map to such columns use nullable types. Nullable types are new to .NET 2.0 and provide a technique for indicating whether a value type should, instead, be `Nothing`. Refer to the Paul Vick's blog entry The Truth About Nullable Types and VB and the technical documentation for the Nullable structure for more information.

All three methods return a Boolean value indicating whether a row was inserted, updated, or deleted since the operation may not result in an affected row. For example, if the page developer calls `DeleteProduct` passing in a `ProductID` for a non-existent product, the `DELETE` statement issued to the database will have no affect and therefore the `DeleteProduct` method will return `False`.

Note that when adding a new product or updating an existing one we take in the new or modified product's field values as a list of scalars as opposed to accepting a `ProductsRow` instance. This approach was chosen because the `ProductsRow` class derives from the ADO.NET `DataRow` class, which doesn't have a default parameterless constructor. In order to create a new `ProductsRow` instance, we must first create a `ProductsDataTable` instance and then invoke its `NewProductRow()` method (which we do in `AddProduct`). This shortcoming rears its head when we turn to inserting and updating products using the ObjectDataSource. In short, the ObjectDataSource will try to create an instance of the input parameters. If the BLL method expects a `ProductsRow` instance, the ObjectDataSource will try to create one, but fail due to the lack of a default parameterless constructor. For more information on this problem, refer to the following two ASP.NET Forums posts: Updating ObjectDataSources with Strongly-Typed DataSets, and Problem With ObjectDataSource and Strongly-Typed DataSet.

Next, in both `AddProduct` and `UpdateProduct`, the code creates a `ProductsRow` instance and populates it with the values just passed in. When assigning values to DataColumns of a DataRow various field-level validation checks can occur. Therefore, manually putting the passed in values back into a DataRow helps ensure the validity of the data being passed to the BLL method. Unfortunately the strongly-typed DataRow classes generated by Visual Studio do not use nullable types. Rather, to indicate that a particular DataColumn in a DataRow should correspond to a `NULL` database value we must use the `SetColumnNameNull()` method.

In `UpdateProduct` we first load in the product to update using `GetProductByProductID(productID)`. While this may seem like an unnecessary trip to the database, this extra trip will prove worthwhile in future tutorials that explore optimistic concurrency. Optimistic concurrency is a technique to ensure that two users who are simultaneously working on the same data don't accidentally overwrite one another's changes. Grabbing the entire record also makes it easier to create update methods in the BLL that only modify a subset of the DataRow's columns. When we explore the `SuppliersBLL` class we'll see such an example.

Finally, note that the `ProductsBLL` class has the DataObject attribute applied to it (the `[System.ComponentModel.DataObject]` syntax right before the class statement near the top of the file) and the methods have DataObjectMethodAttribute attributes. The `DataObject` attribute marks the class as being an object suitable for binding to an ObjectDataSource control, whereas the `DataObjectMethodAttribute` indicates the purpose of the method. As we'll see in future tutorials, ASP.NET 2.0's ObjectDataSource makes it easy to declaratively access data from a class. To help filter the list of possible classes to bind to in the ObjectDataSource's wizard, by default only those classes marked as `DataObjects` are shown in the wizard's drop-down list. The `ProductsBLL` class will work just as well without these attributes, but adding them makes it easier to work with in the ObjectDataSource's wizard.

Adding the Other Classes

With the `ProductsBLL` class complete, we still need to add the classes for working with categories, suppliers, and employees. Take a moment to create the following classes and methods using the concepts from the example above:

- **CategoriesBLL.cs**
  - `GetCategories()`

- `GetCategoryByCategoryID(categoryID)`

- SuppliersBLL.cs
  - `GetSuppliers()`
  - `GetSupplierBySupplierID(supplierID)`
  - `GetSuppliersByCountry(country)`
  - `UpdateSupplierAddress(supplierID, address, city, country)`
- EmployeesBLL.cs
  - `GetEmployees()`
  - `GetEmployeeByEmployeeID(employeeID)`
  - `GetEmployeesByManager(managerID)`

The one method worth noting is the `SuppliersBLL` class's `UpdateSupplierAddress` method. This method provides an interface for updating just the supplier's address information. Internally, this method reads in the `SupplierDataRow` object for the specified `supplierID` (using `GetSupplierBySupplierID`), sets its address-related properties, and then calls down into the `SupplierDataTable`'s `Update` method. The `UpdateSupplierAddress` method follows:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Update, True)> _
Public Function UpdateSupplierAddress(ByVal supplierID As Integer, _
    ByVal address As String, ByVal city As String, ByVal country As String) _
    As Boolean

    Dim suppliers As Northwind.SuppliersDataTable = _
        Adapter.GetSupplierBySupplierID(supplierID)

    If suppliers.Count = 0 Then
        Return False
    Else
        Dim supplier As Northwind.SuppliersRow = suppliers(0)

        If address Is Nothing Then
            supplier.SetAddressNull()
        Else
            supplier.Address = address
        End If

        If city Is Nothing Then
            supplier.SetCityNull()
        Else
            supplier.City = city
        End If

        If country Is Nothing Then
            supplier.SetCountryNull()
        Else
            supplier.Country = country
        End If

        Dim rowsAffected As Integer = Adapter.Update(supplier)

        Return rowsAffected = 1
    End If
End Function
```

Refer to this article's download for my complete implementation of the BLL classes.

Step 2: Accessing the Typed DataSets Through the BLL Classes

In the first tutorial we saw examples of working directly with the Typed DataSet programmatically, but with the addition of our BLL classes, the presentation tier should work against the BLL instead. In the `AllProducts.aspx` example from the first tutorial, the `ProductsTableAdapter` was used to bind the list of products to a GridView, as shown in the following code:

```
Dim productsAdapter As New ProductsTableAdapter()
GridView1.DataSource = productsAdapter.GetProducts()
GridView1.DataBind()
```

To use the new BLL classes, all that needs to be changed is the first line of code simply replace the `ProductsTableAdapter` object with a `ProductBLL` object:

```
Dim productLogic As New ProductsBLL()
GridView1.DataSource = productLogic.GetProducts()
GridView1.DataBind()
```

The BLL classes can also be accessed declaratively (as can the Typed DataSet) by using the ObjectDataSource. We'll be discussing the ObjectDataSource in greater detail in the following tutorials.

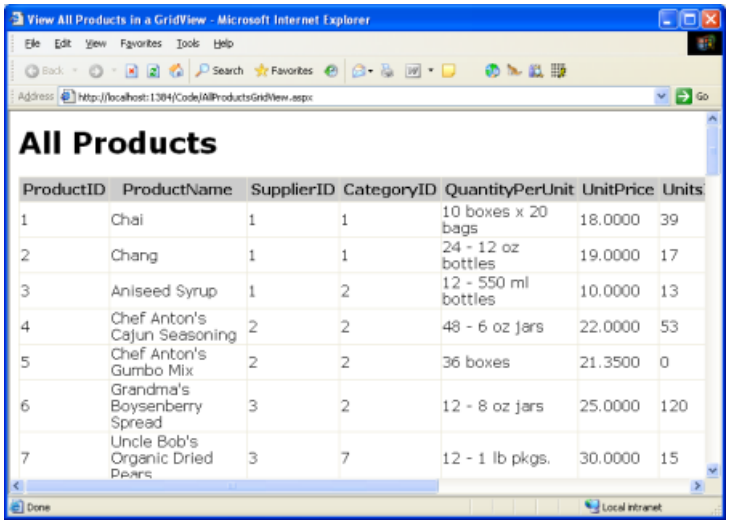


Figure 3: The List of Products is Displayed in a GridView (Click to view full-size image)

Step 3: Adding Field-Level Validation to the DataRow Classes

Field-level validation are checks that pertains to the property values of the business objects when inserting or updating. Some field-level validation rules for products include:

- The `ProductName` field must be 40 characters or less in length
- The `QuantityPerUnit` field must be 20 characters or less in length
- The `ProductID`, `ProductName`, and `Discontinued` fields are required, but all other fields are optional
- The `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, and `ReorderLevel` fields must be greater than or equal to zero

These rules can and should be expressed at the database level. The character limit on the `ProductName` and `QuantityPerUnit` fields are captured by the data types of those columns in the `Products` table (`nvarchar(40)` and `nvarchar(20)`, respectively). Whether fields are required and optional are expressed by if the database table column allows `NULL` s. Four check constraints exist that ensure that only values greater than or equal to zero can make it into the `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, or `ReorderLevel` columns.

In addition to enforcing these rules at the database they should also be enforced at the DataSet level. In fact, the field length and whether a value is required or optional are already captured for each DataTable's set of

DataColumns. To see the existing field-level validation automatically provided, go to the DataSet Designer, select a field from one of the DataTables and then go to the Properties window. As Figure 4 shows, the `QuantityPerUnit` DataColumn in the `ProductsDataTable` has a maximum length of 20 characters and does allow `NULL` values. If we attempt to set the `ProductsDataRow`'s `QuantityPerUnit` property to a string value longer than 20 characters an `ArgumentException` will be thrown.

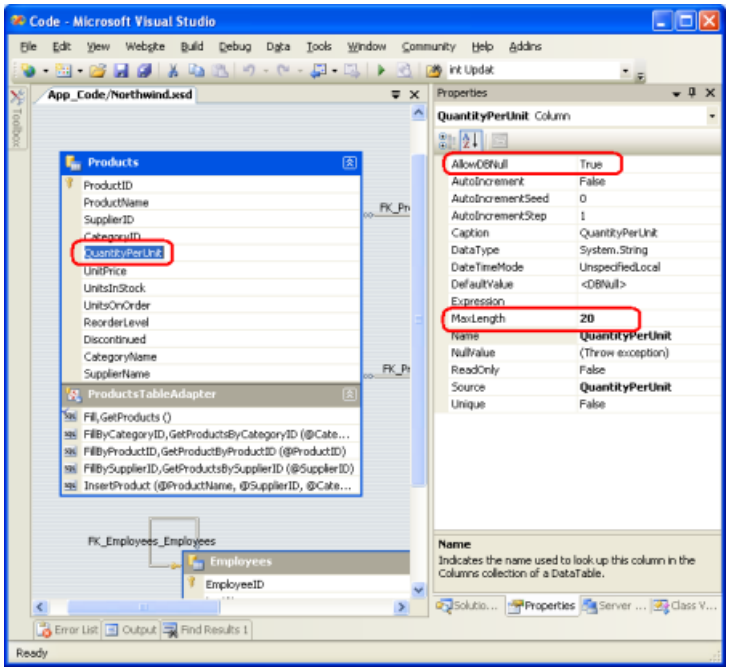


Figure 4: The DataColumn Provides Basic Field-Level Validation (Click to view full-size image)

Unfortunately, we can't specify bounds checks, such as the `UnitPrice` value must be greater than or equal to zero, through the Properties window. In order to provide this type of field-level validation we need to create an event handler for the DataTable's ColumnChanging event. As mentioned in the preceding tutorial, the DataSet, DataTables, and DataRow objects created by the Typed DataSet can be extended through the use of partial classes. Using this technique we can create a `ColumnChanging` event handler for the `ProductsDataTable` class. Start by creating a class in the `App_Code` folder named `ProductsDataTable.ColumnChanging.vb`.

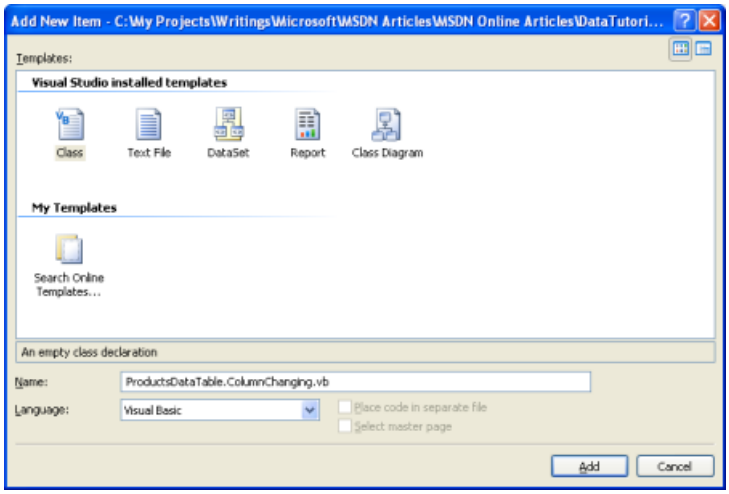


Figure 5: Add a New Class to the `App_Code` Folder (Click to view full-size image)

Next, create an event handler for the `ColumnChanging` event that ensures that the `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, and `ReorderLevel` column values (if not `NULL`) are greater than or equal to zero. If any such column is out of range, throw an `ArgumentException`.

ProductsDataTable.ColumnChanging.vb

```
Imports System.data

Partial Public Class Northwind
    Partial Public Class ProductsDataTable
        Public Overrides Sub BeginInit()
            AddHandler Me.ColumnChanging, AddressOf ValidateColumn
        End Sub

        Sub ValidateColumn(sender As Object, e As DataColumnChangeEventArgs)
            If e.Column.Equals(Me.UnitPriceColumn) Then
                If Not Convert.IsDBNull(e.ProposedValue) AndAlso _
                    CType(e.ProposedValue, Decimal) < 0 Then
                    Throw New ArgumentException( _
                        "UnitPrice cannot be less than zero", "UnitPrice")
                End If
            ElseIf e.Column.Equals(Me.UnitsInStockColumn) OrElse _
                e.Column.Equals(Me.UnitsOnOrderColumn) OrElse _
                e.Column.Equals(Me.ReorderLevelColumn) Then
                If Not Convert.IsDBNull(e.ProposedValue) AndAlso _
                    CType(e.ProposedValue, Short) < 0 Then
                    Throw New ArgumentException(String.Format( _
                        "{0} cannot be less than zero", e.Column.ColumnName), _
                        e.Column.ColumnName)
                End If
            End If
        End Sub
    End Class
End Class
```

Step 4: Adding Custom Business Rules to the BLL's Classes

In addition to field-level validation, there may be high-level custom business rules that involve different entities or concepts not expressible at the single column level, such as:

- If a product is discontinued, its `UnitPrice` cannot be updated
- An employee's country of residence must be the same as their manager's country of residence
- A product cannot be discontinued if it is the only product provided by the supplier

The BLL classes should contain checks to ensure adherence to the application's business rules. These checks can be added directly to the methods to which they apply.

Imagine that our business rules dictate that a product could not be marked discontinued if it was the only product from a given supplier. That is, if product *X* was the only product we purchased from supplier *Y*, we could not mark *X* as discontinued; if, however, supplier *Y* supplied us with three products, *A*, *B*, and *C*, then we could mark any and all of these as discontinued. An odd business rule, but business rules and common sense aren't always aligned!

To enforce this business rule in the `UpdateProducts` method we'd start by checking if `Discontinued` was set to `True` and, if so, we'd call `GetProductsBySupplierID` to determine how many products we purchased from this product's supplier. If only one product is purchased from this supplier, we throw an `ApplicationException`.

```
<System.ComponentModel.DataObjectMethodAttribute_
    (System.ComponentModel.DataObjectMethodType.Update, True)> _
Public Function UpdateProduct( _
    productName As String, supplierID As Nullable(Of Integer), _
    categoryID As Nullable(Of Integer), quantityPerUnit As String, _
    unitPrice As Nullable(Of Decimal), unitsInStock As Nullable(Of Short), _
    unitsOnOrder As Nullable(Of Short), reorderLevel As Nullable(Of Short), _
    discontinued As Boolean, productID As Integer) _
    As Boolean
```

```
    Dim products As Northwind.ProductsDataTable = _
        Adapter.GetProductByProductID(productID)

    If products.Count = 0 Then
        Return False
    End If

    Dim product As Northwind.ProductsRow = products(0)

    If discontinued Then
        Dim productsBySupplier As Northwind.ProductsDataTable = _
            Adapter.GetProductsBySupplierID(product.SupplierID)

        If productsBySupplier.Count = 1 Then
            Throw New ApplicationException( _
                "You cannot mark a product as discontinued if it is " & _
                "the only product purchased from a supplier")
        End If
    End If

    product.ProductName = productName

    If Not supplierID.HasValue Then
        product.SetSupplierIDNull()
    Else
        product.SupplierID = supplierID.Value
    End If

    If Not categoryID.HasValue Then
        product.SetCategoryIDNull()
    Else
        product.CategoryID = categoryID.Value
    End If

    If quantityPerUnit Is Nothing Then
        product.SetQuantityPerUnitNull()
    Else
        product.QuantityPerUnit = quantityPerUnit
    End If

    If Not unitPrice.HasValue Then
        product.SetUnitPriceNull()
    Else
        product.UnitPrice = unitPrice.Value
    End If

    If Not unitsInStock.HasValue Then
        product.SetUnitsInStockNull()
    Else
        product.UnitsInStock = unitsInStock.Value
    End If

    If Not unitsOnOrder.HasValue Then
        product.SetUnitsOnOrderNull()
    Else
        product.UnitsOnOrder = unitsOnOrder.Value
    End If

    If Not reorderLevel.HasValue Then
        product.SetReorderLevelNull()
    Else
        product.ReorderLevel = reorderLevel.Value
    End If

    product.Discontinued = discontinued

    Dim rowsAffected As Integer = Adapter.Update(product)

    Return rowsAffected = 1
```

```
End Function
```

Responding to Validation Errors in the Presentation Tier

When calling the BLL from the presentation tier we can decide whether to attempt to handle any exceptions that might be raised or let them bubble up to ASP.NET (which will raise the `HttpApplication`'s `Error` event). To handle an exception when working with the BLL programmatically, we can use a Try...Catch block, as the following example shows:

```
Dim productLogic As New ProductsBLL()

Try
    productLogic.UpdateProduct("Scotts Tea", 1, 1, Nothing, _
      -14, 10, Nothing, Nothing, False, 1)
Catch ae As ArgumentException
    Response.Write("There was a problem: " & ae.Message)
End Try
```

As we'll see in future tutorials, handling exceptions that bubble up from the BLL when using a data Web control for inserting, updating, or deleting data can be handled directly in an event handler as opposed to having to wrap code in `Try...Catch` blocks.

Summary

A well architected application is crafted into distinct layers, each of which encapsulates a particular role. In the first tutorial of this article series we created a Data Access Layer using Typed DataSets; in this tutorial we built a Business Logic Layer as a series of classes in our application's `App_Code` folder that call down into our DAL. The BLL implements the field-level and business-level logic for our application. In addition to creating a separate BLL, as we did in this tutorial, another option is to extend the TableAdapters' methods through the use of partial classes. However, using this technique does not allow us to override existing methods nor does it separate our DAL and our BLL as cleanly as the approach we've taken in this article.

With the DAL and BLL complete, we're ready to start on our presentation layer. In the next tutorial we'll take a brief detour from data access topics and define a consistent page layout for use throughout the tutorials.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Liz Shulok, Dennis Patterson, Carlos Santos, and Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Master Pages and Site Navigation (VB)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

One common characteristic of user-friendly websites is that they have a consistent, site-wide page layout and navigation scheme. This tutorial looks at how you can create a consistent look and feel across all pages that can easily be updated.

Introduction

One common characteristic of user-friendly websites is that they have a consistent, site-wide page layout and navigation scheme. ASP.NET 2.0 introduces two new features that greatly simplify implementing both a site-wide page layout and navigation scheme: master pages and site navigation. Master pages allow for developers to create a site-wide template with designated editable regions. This template can then be applied to ASP.NET pages in the site. Such ASP.NET pages need only provide content for the master page's specified editable regions all other markup in the master page is identical across all ASP.NET pages that use the master page. This model allows developers to define and centralize a site-wide page layout, thereby making it easier to create a consistent look and feel across all pages that can easily be updated.

The site navigation system provides both a mechanism for page developers to define a site map and an API for that site map to be programmatically queried. The new navigation Web controls the Menu, TreeView, and SiteMapPath make it easy to render all or part of the site map in a common navigation user interface element. We'll be using the default site navigation provider, meaning that our site map will be defined in an XML-formatted file.

To illustrate these concepts and make our tutorials website more usable, let's spend this lesson defining a site-wide page layout, implementing a site map, and adding the navigation UI. By the end of this tutorial we'll have a polished website design for building our tutorial web pages.

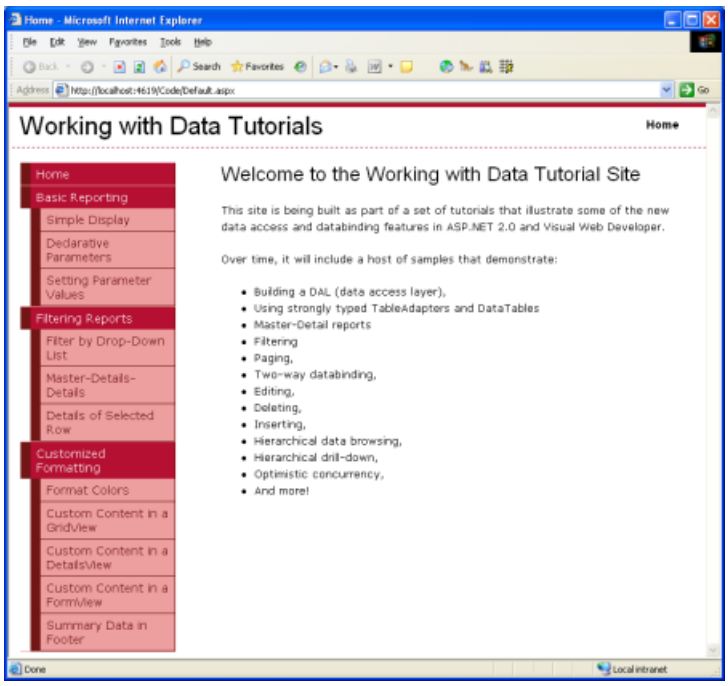


Figure 1: The End Result of This Tutorial (Click to view full-size image)

Step 1: Creating the Master Page

The first step is to create the master page for the site. Right now our website consists of only the Typed DataSet (`Northwind.xsd` , in the `App_Code` folder), the BLL classes (`ProductsBLL.vb` , `CategoriesBLL.vb` , and so on, all in the `App_Code` folder), the database (`NORTHWND.MDF` , in the `App_Data` folder), the configuration file (`Web.config`), and a CSS stylesheet file (`Styles.css`). I cleaned out those pages and files demonstrating using the DAL and BLL from the first two tutorials since we will be reexamining those examples in greater detail in future tutorials.

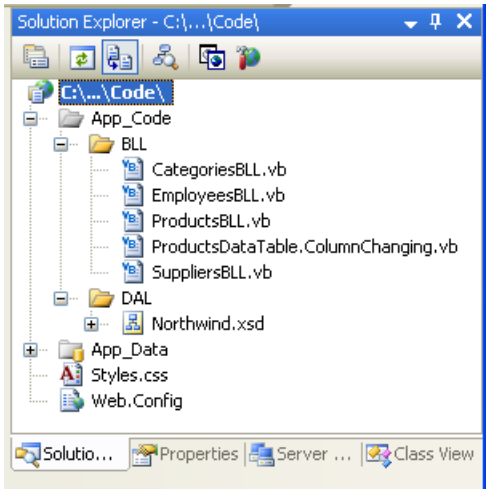


Figure 2: The Files in Our Project

To create a master page, right-click on the project name in the Solution Explorer and choose Add New Item. Then select the Master Page type from the list of templates and name it `Site.master` .

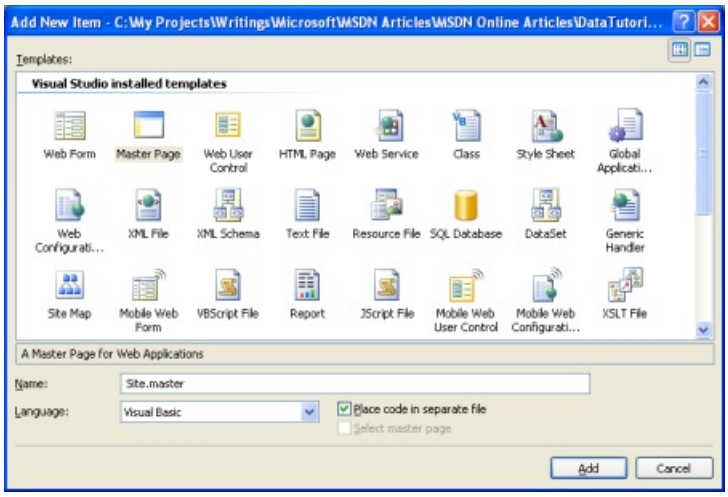


Figure 3: Add a New Master Page to the Website (Click to view full-size image)

Define the site-wide page layout here in the master page. You can use the Design view and add whatever Layout or Web controls you need, or you can manually add the markup by hand in the Source view. In my master page I use cascading style sheets for positioning and styles with the CSS settings defined in the external file `Style.css` . While you cannot tell from the markup shown below, the CSS rules are defined such that the navigation `<div>` 's content is absolutely positioned so that it appears on the left and has a fixed width of 200 pixels.

Site.master

```
<%@ Master Language="VB" AutoEventWireup="true"
    CodeFile="Site.master.vb" Inherits="Site" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Working with Data Tutorials</title>
    <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <div id="wrapper">

        <form id="form1" runat="server">

            <div id="header">
                <span class="title">Working with Data Tutorials</span>
                <span class="breadcrumb">
                 TODO: Breadcrumb will go here...</span>
            </div>

            <div id="content">
                <asp:contentplaceholder id="MainContent"
                  runat="server">
                  <!-- Page-specific content will go here... -->
                </asp:contentplaceholder>
            </div>

            <div id="navigation">
                TODO: Menu will go here...
            </div>
        </form>
    </div>
</body>
</html>
```

A master page defines both the static page layout and the regions that can be edited by the ASP.NET pages that use the master page. These content editable regions are indicated by the ContentPlaceHolder control, which can be seen within the content `<div>`. Our master page has a single ContentPlaceHolder (`MainContent`), but master page's may have multiple ContentPlaceHolders.

With the markup entered above, switching to the Design view shows the master page's layout. Any ASP.NET pages that use this master page will have this uniform layout, with the ability to specify the markup for the `MainContent` region.

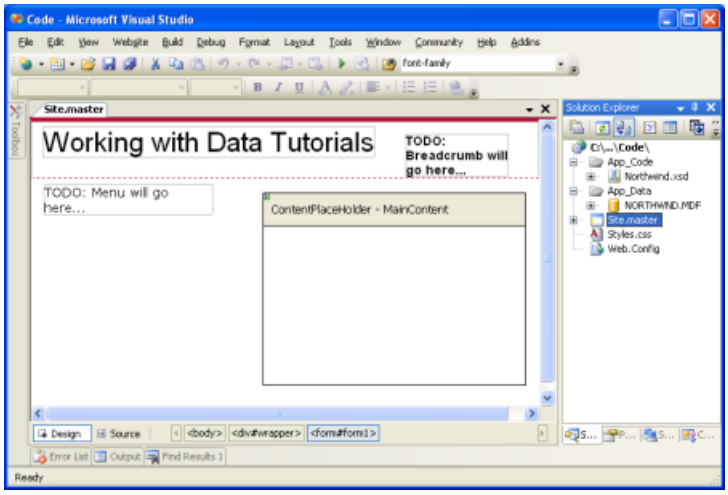


Figure 4: The Master Page, When Viewed Through the Design View (Click to view full-size image)

Step 2: Adding a Homepage to the Website

With the master page defined, we're ready to add the ASP.NET pages for the website. Let's start by adding `Default.aspx`, our website's homepage. Right-click on the project name in the Solution Explorer and choose Add New Item. Pick the Web Form option from the template list and name the file `Default.aspx`. Also, check the "Select master page" checkbox.

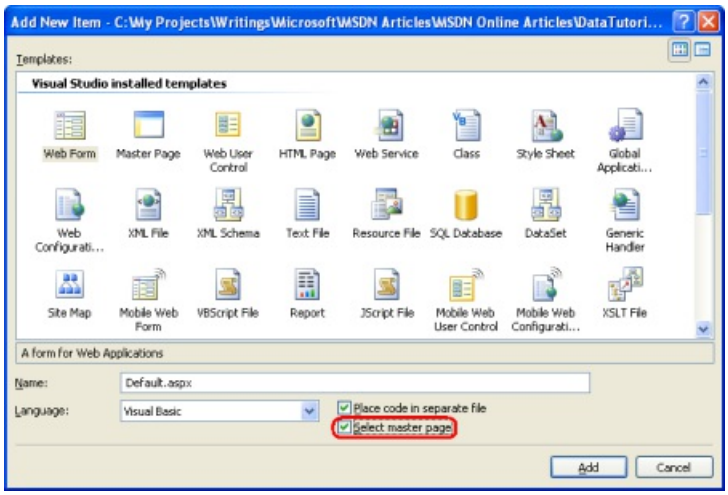


Figure 5: Add a New Web Form, Checking the Select master page Checkbox (Click to view full-size image)

After clicking the OK button, we're asked to choose what master page this new ASP.NET page should use. While you can have multiple master pages in your project, we have only one.

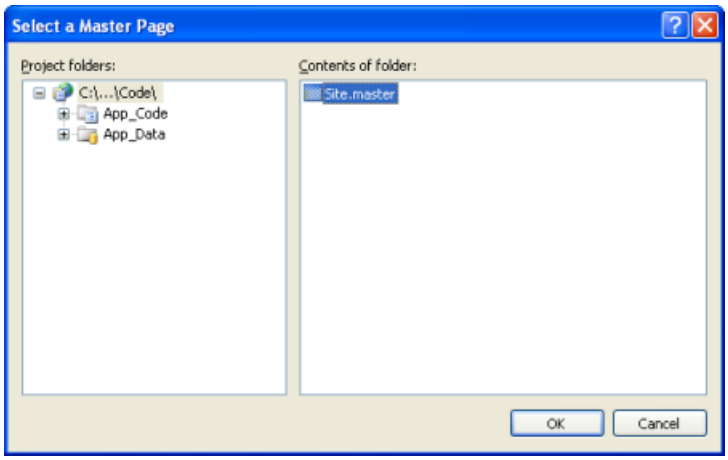


Figure 6: Choose the Master Page this ASP.NET Page Should Use (Click to view full-size image)

After picking the master page, the new ASP.NET pages will contain the following markup:

Default.aspx

```
<%@ Page Language="VB" MasterPageFile="~/Site.master"
    AutoEventWireup="true" CodeFile="Default.aspx.vb"
    Inherits="_Default" Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent"
  Runat="Server">
</asp:Content>
```

In the `@Page` directive there's a reference to the master page file used (`MasterPageFile="~/Site.master"`), and the ASP.NET page's markup contains a Content control for each of the ContentPlaceHolder controls defined in the master page, with the control's `ContentPlaceHolderID` mapping the Content control to a specific ContentPlaceHolder. The Content control is where you place the markup you want to appear in the corresponding ContentPlaceHolder. Set the `@Page` directive's `Title` attribute to Home and add some

welcoming content to the Content control:

Default.aspx

```
<%@ Page Language="VB" MasterPageFile="~/Site.master"
    AutoEventWireup="true" CodeFile="Default.aspx.vb"
    Inherits="_Default" Title="Home" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent"
    Runat="Server">
    <h1>Welcome to the Working with Data Tutorial Site</h1>

    <p>This site is being built as part of a set of tutorials
      that illustrate some of the new data access and databinding
      features in ASP.NET 2.0 and Visual Web Developer.</p>

    <p>Over time, it will include a host of samples that
     demonstrate:</p>

    <ul>
        <li>Building a DAL (data access layer),</li>
        <li>Using strongly typed TableAdapters and DataTables</li>
        <li>Master-Detail reports</li>
        <li>Filtering</li>
        <li>Paging,</li>
        <li>Two-way databinding,</li>
        <li>Editing,</li>
        <li>Deleting,</li>
        <li>Inserting,</li>
        <li>Hierarchical data browsing,</li>
        <li>Hierarchical drill-down,</li>
        <li>Optimistic concurrency,</li>
        <li>And more!</li>
    </ul>
</asp:Content>
```

The `Title` attribute in the `@Page` directive allows us to set the page's title from the ASP.NET page, even though the `<title>` element is defined in the master page. We can also set the title programmatically, using `Page.Title`. Also note that the master page's references to stylesheets (such as `Style.css`) are automatically updated so that they work in any ASP.NET page, regardless of what directory the ASP.NET page is in relative to the master page.

Switching to the Design view we can see how our page will look in a browser. Note that in the Design view for the ASP.NET page that only the content editable regions are editable the non-ContentPlaceHolder markup defined in the master page is grayed out.

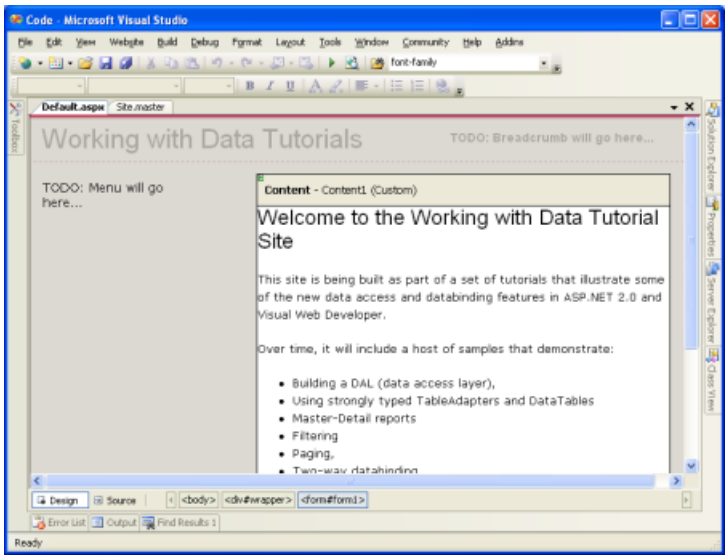


Figure 7: The Design View for the ASP.NET Page Shows Both the Editable and Non-Editable Regions (Click to view full-size image)

When the `Default.aspx` page is visited by a browser, the ASP.NET engine automatically merges the page's master page content and the ASP.NET's content, and renders the merged content into the final HTML that is sent down to the requesting browser. When the master page's content is updated, all ASP.NET pages that use this master page will have their content remerged with the new master page content the next time they are requested. In short, the master page model allows for a single page layout template to be defined (the master page) whose changes are immediately reflected across the entire site.

Adding Additional ASP.NET Pages to the Website

Let's take a moment to add additional ASP.NET page stubs to the site that will eventually hold the various reporting demos. There will be more than 35 demos in total, so rather than creating all of the stub pages let's just create the first few. Since there will also be many categories of demos, to better manage the demos add a folder for the categories. Add the following three folders for now:

- `BasicReporting`
- `Filtering`
- `CustomFormatting`

Finally, add new files as shown in the Solution Explorer in Figure 8. When adding each file, remember to check the "Select master page" checkbox.

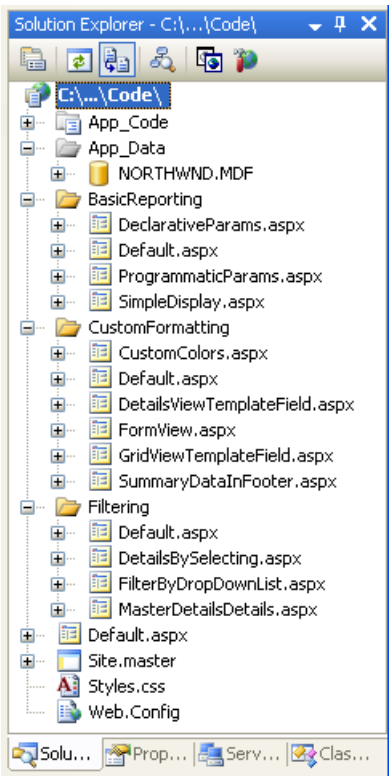


Figure 8: Add the Following Files

Step 2: Creating a Site Map

One of the challenges of managing a website composed of more than a handful of pages is providing a straightforward way for visitors to navigate through the site. To begin with, the site's navigational structure must be defined. Next, this structure must be translated into navigable user interface elements, such as menus or breadcrumbs. Finally, this whole process needs to be maintained and updated as new pages are added to the site and existing ones removed. Prior to ASP.NET 2.0, developers were on their own for creating the site's navigational structure, maintaining it, and translating it into navigable user interface elements. With ASP.NET 2.0,

however, developers can utilize the very flexible built in site navigation system.

The ASP.NET 2.0 site navigation system provides a means for a developer to define a site map and to then access this information through a programmatic API. ASP.NET ships with a site map provider that expects site map data to be stored in an XML file formatted in a particular way. But, since the site navigation system is built on the provider model it can be extended to support alternative ways for serializing the site map information. Jeff Prosise's article, The SQL Site Map Provider You've Been Waiting For shows how to create a site map provider that stores the site map in a SQL Server database; another option is to create a site map provider based on the file system structure.

For this tutorial, however, let's use the default site map provider that ships with ASP.NET 2.0. To create the site map, simply right-click on the project name in the Solution Explorer, choose Add New Item, and choose the Site Map option. Leave the name as `Web.sitemap` and click the Add button.

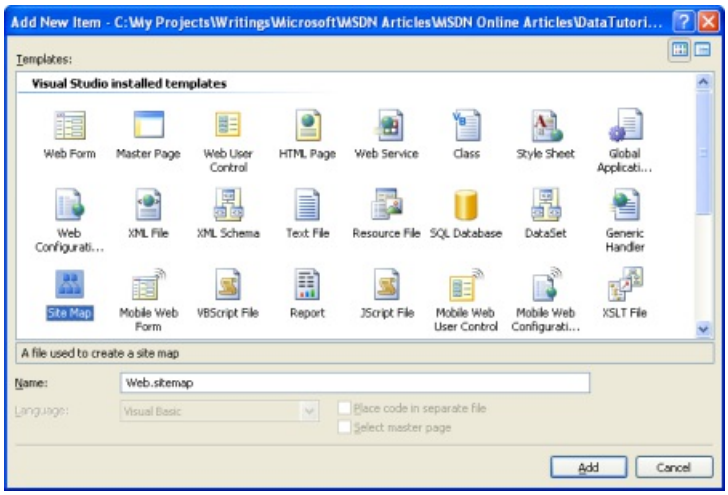


Figure 9: Add a Site Map to Your Project (Click to view full-size image)

The site map file is an XML file. Note that Visual Studio provides IntelliSense for the site map structure. The site map file must have the `<siteMap>` node as its root node, which must contain precisely one `<siteMapNode>` child element. That first `<siteMapNode>` element can then contain an arbitrary number of descendent `<siteMapNode>` elements.

Define the site map to mimic the file system structure. That is, add a `<siteMapNode>` element for each of the three folders, and child `<siteMapNode>` elements for each of the ASP.NET pages in those folders, like so:

Web.sitemap

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >

  <siteMapNode url="~/Default.aspx" title="Home" description="Home">
      <siteMapNode title="Basic Reporting"
        url="~/BasicReporting/Default.aspx"
        description="Basic Reporting Samples">
        <siteMapNode url="~/BasicReporting/SimpleDisplay.aspx"
         title="Simple Display"
         description="Displays the complete contents
          of a database table." />
        <siteMapNode url="~/BasicReporting/DeclarativeParams.aspx"
          title="Declarative Parameters"
          description="Displays a subset of the contents
            of a database table using parameters." />
        <siteMapNode url="~/BasicReporting/ProgrammaticParams.aspx"
         title="Setting Parameter Values"
         description="Shows how to set parameter values
          programmatically." />
      </siteMapNode>

      <siteMapNode title="Filtering Reports"
       url="~/Filtering/Default.aspx"
       description="Samples of Reports that Support Filtering">
        <siteMapNode url="~/Filtering/FilterByDropDownList.aspx"
          title="Filter by Drop-Down List"
          description="Filter results using a drop-down list." />
        <siteMapNode url="~/Filtering/MasterDetailsDetails.aspx"
         title="Master-Details-Details"
         description="Filter results two levels down." />
        <siteMapNode url="~/Filtering/DetailsBySelecting.aspx"
          title="Details of Selected Row"
          description="Show detail results for a selected item in a GridView." />
      </siteMapNode>

      <siteMapNode title="Customized Formatting"
         url="~/CustomFormatting/Default.aspx"
         description="Samples of Reports Whose Formats are Customized">
        <siteMapNode url="~/CustomFormatting/CustomColors.aspx"
         title="Format Colors"
         description="Format the grid s colors based
           on the underlying data." />
        <siteMapNode
          url="~/CustomFormatting/GridViewTemplateField.aspx"
          title="Custom Content in a GridView"
          description="Shows using the TemplateField to
          customize the contents of a field in a GridView." />
        <siteMapNode
          url="~/CustomFormatting/DetailsViewTemplateField.aspx"
          title="Custom Content in a DetailsView"
          description="Shows using the TemplateField to customize
           the contents of a field in a DetailsView." />
        <siteMapNode url="~/CustomFormatting/FormView.aspx"
          title="Custom Content in a FormView"
          description="Illustrates using a FormView for a
           highly customized view." />
        <siteMapNode url="~/CustomFormatting/SummaryDataInFooter.aspx"
          title="Summary Data in Footer"
          description="Display summary data in the grids footer." />
      </siteMapNode>

  </siteMapNode>

</siteMap>
```

The site map defines the website's navigational structure, which is a hierarchy that describes the various sections

of the site. Each `<siteMapNode>` element in `Web.sitemap` represents a section in the site's navigational structure.

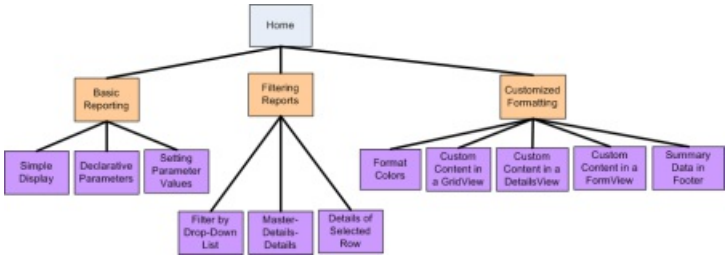


Figure 10: The Site Map Represents a Hierarchical Navigational Structure (Click to view full-size image)

ASP.NET exposes the site map's structure through the .NET Framework's SiteMap class. This class has a `CurrentNode` property, which returns information about the section the user is currently visiting; the `RootNode` property returns the root of the site map (Home, in our site map). Both the `CurrentNode` and `RootNode` properties return SiteMapNode instances, which have properties like `ParentNode`, `ChildNodes`, `NextSibling`, `PreviousSibling`, and so on, that allow for the site map hierarchy to be walked.

Step 3: Displaying a Menu Based on the Site Map

Accessing data in ASP.NET 2.0 can be accomplished programmatically, like in ASP.NET 1.x, or declaratively, through the new data source controls. There are several built-in data source controls such as the SqlDataSource control, for accessing relational database data, the ObjectDataSource control, for accessing data from classes, and others. You can even create your own custom data source controls.

The data source controls serve as a proxy between your ASP.NET page and the underlying data. In order to display a data source control's retrieved data, we'll typically add another Web control to the page and bind it to the data source control. To bind a Web control to a data source control, simply set the Web control's `DataSourceID` property to the value of the data source control's `ID` property.

To aid in working with the site map's data, ASP.NET includes the SiteMapDataSource control, which allows us to bind a Web control against our website's site map. Two Web controls the TreeView and Menu are commonly used to provide a navigation user interface. To bind the site map data to one of these two controls, simply add a SiteMapDataSource to the page along with a TreeView or Menu control whose `DataSourceID` property is set accordingly. For example, we could add a Menu control to the master page using the following markup:

```
<div id="navigation">
    <asp:Menu ID="Menu1" runat="server"
      DataSourceID="SiteMapDataSource1">
    </asp:Menu>

    <asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" />
</div>
```

For a finer degree of control over the emitted HTML, we can bind the SiteMapDataSource control to the Repeater control, like so:

```
<div id="navigation">
    <ul>
        <li><asp:HyperLink runat="server" ID="lnkHome"
         NavigateUrl="~/Default.aspx">Home</asp:HyperLink></li>

        <asp:Repeater runat="server" ID="menu"
          DataSourceID="SiteMapDataSource1">
            <ItemTemplate>
                <li>
                    <asp:HyperLink runat="server"
                    NavigateUrl='<%# Eval("Url") %>'>
                    <%# Eval("Title") %></asp:HyperLink>
                </li>
            </ItemTemplate>
        </asp:Repeater>
    </ul>

    <asp:SiteMapDataSource ID="SiteMapDataSource1"
      runat="server" ShowStartingNode="false" />
</div>
```

The SiteMapDataSource control returns the site map hierarchy one level at a time, starting with the root site map node (Home, in our site map), then the next level (Basic Reporting, Filtering Reports, and Customized Formatting), and so on. When binding the SiteMapDataSource to a Repeater, it enumerates the first level returned and instantiates the `ItemTemplate` for each `SiteMapNode` instance in that first level. To access a particular property of the `SiteMapNode`, we can use `Eval(propertyName)`, which is how we get each `SiteMapNode` 's `Url` and `Title` properties for the HyperLink control.

The Repeater example above will render the following markup:

```
<li>
    <a href="/Code/BasicReporting/Default.aspx">Basic Reporting</a>
</li>

<li>
    <a href="/Code/Filtering/Default.aspx">Filtering Reports</a>
</li>

<li>
    <a href="/Code/CustomFormatting/Default.aspx">
      Customized Formatting</a>
</li>
```

These site map nodes (Basic Reporting, Filtering Reports, and Customized Formatting) comprise the *second* level of the site map being rendered, not the first. This is because the SiteMapDataSource's `ShowStartingNode` property is set to False, causing the SiteMapDataSource to bypass the root site map node and instead begin by returning the second level in the site map hierarchy.

To display the children for the Basic Reporting, Filtering Reports, and Customized Formatting `SiteMapNode` s, we can add another Repeater to the initial Repeater's `ItemTemplate`. This second Repeater will be bound to the `SiteMapNode` instance's `ChildNodes` property, like so:

```
<asp:Repeater runat="server" ID="menu" DataSourceID="SiteMapDataSource1">
    <ItemTemplate>
        <li>
            <asp:HyperLink runat="server"
             NavigateUrl='<%# Eval("Url") %>'>
             <%# Eval("Title") %></asp:HyperLink>

            <asp:Repeater runat="server"
             DataSource="<%# CType(Container.DataItem,
             SiteMapNode).ChildNodes %>">
                <HeaderTemplate>
                    <ul>
                </HeaderTemplate>

                <ItemTemplate>
                    <li>
                        <asp:HyperLink runat="server"
                         NavigateUrl='<%# Eval("Url") %>'>
                         <%# Eval("Title") %></asp:HyperLink>
                    </li>
                </ItemTemplate>

                <FooterTemplate>
                    </ul>
                </FooterTemplate>
            </asp:Repeater>
        </li>
    </ItemTemplate>
</asp:Repeater>
```

These two Repeaters result in the following markup (some markup has been removed for brevity):

```
<li>
    <a href="/Code/BasicReporting/Default.aspx">Basic Reporting</a>
    <ul>
       <li>
          <a href="/Code/BasicReporting/SimpleDisplay.aspx">
           Simple Display</a>
       </li>
       <li>
          <a href="/Code/BasicReporting/DeclarativeParams.aspx">
           Declarative Parameters</a>
       </li>
       <li>
          <a href="/Code/BasicReporting/ProgrammaticParams.aspx">
           Setting Parameter Values</a>
       </li>
    </ul>
</li>

<li>
    <a href="/Code/Filtering/Default.aspx">Filtering Reports</a>
    ...
</li>

<li>
    <a href="/Code/CustomFormatting/Default.aspx">
        Customized Formatting</a>
    ...
</li>
```

Using CSS styles chosen from Rachel Andrew's book The CSS Anthology: 101 Essential Tips, Tricks, & Hacks, the `<ul>` and `<li>` elements are styled such that the markup produces the following visual output:

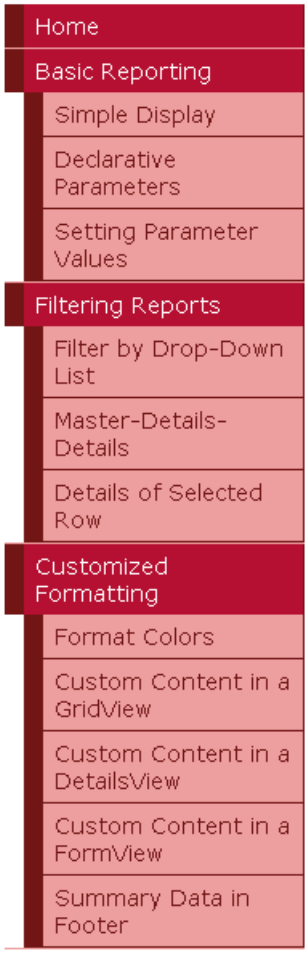


Figure 11: A Menu Composed from Two Repeaters and Some CSS

This menu is in the master page and bound to the site map defined in `Web.sitemap`, meaning that any change to the site map will be immediately reflected on all pages that use the `Site.master` master page.

Disabling ViewState

All ASP.NET controls can optionally persist their state to the view state, which is serialized as a hidden form field in the rendered HTML. View state is used by controls to remember their programmatically-changed state across postbacks, such as the data bound to a data Web control. While view state permits information to be remembered across postbacks, it increases the size of the markup that must be sent to the client and can lead to severe page bloat if not closely monitored. Data Web controls especially the GridView are particularly notorious for adding dozens of extra kilobytes of markup to a page. While such an increase may be negligible for broadband or intranet users, view state can add several seconds to the round trip for dial-up users.

To see the impact of view state, visit a page in a browser and then view the source sent by the web page (in Internet Explorer, go to the View menu and choose the Source option). You can also turn on page tracing to see the view state allocation used by each of the controls on the page. The view state information is serialized in a hidden form field named `__VIEWSTATE`, located in a `<div>` element immediately after the opening `<form>` tag. View state is only persisted when there is a Web Form being used; if your ASP.NET page does not include a `<form runat="server">` in its declarative syntax there won't be a `__VIEWSTATE` hidden form field in the rendered markup.

The `__VIEWSTATE` form field generated by the master page adds roughly 1,800 bytes to the page's generated markup. This extra bloat is due primarily to the Repeater control, as the contents of the SiteMapDataSource control are persisted to view state. While an extra 1,800 bytes may not seem like much to get excited about, when using a GridView with many fields and records, the view state can easily swell by a factor of 10 or more.

View state can be disabled at the page or control level by setting the `EnableViewState` property to `False`,

thereby reducing the size of the rendered markup. Since the view state for a data Web control persists the data bound to the data Web control across postbacks, when disabling the view state for a data Web control the data must be bound on each and every postback. In ASP.NET version 1.x this responsibility fell on the shoulders of the page developer; with ASP.NET 2.0, however, the data Web controls will rebind to their data source control on each postback if needed.

To reduce the page's view state let's set the Repeater control's `EnableViewState` property to `False`. This can be done through the Properties window in the Designer or declaratively in the Source view. After making this change the Repeater's declarative markup should look like:

```
<asp:Repeater runat="server" ID="menu" DataSourceID="SiteMapDataSource1"
    EnableViewState="False">
    <ItemTemplate>
        ... <i>ItemTemplate contents omitted for brevity</i> ...
    </ItemTemplate>
</asp:Repeater>
```

After this change, the page's rendered view state size has shrunk to a mere 52 bytes, a 97% savings in view state size! In the tutorials throughout this series we'll disable the view state of the data Web controls by default in order to reduce the size of the rendered markup. In the majority of the examples the `EnableViewState` property will be set to `False` and done so without mention. The only time view state will be discussed is in scenarios where it must be enabled in order for the data Web control to provide its expected functionality.

Step 4: Adding Breadcrumb Navigation

To complete the master page, let's add a breadcrumb navigation UI element to each page. The breadcrumb quickly shows users their current position within the site hierarchy. Adding a breadcrumb in ASP.NET 2.0 is easy just add a SiteMapPath control to the page; no code is needed.

For our site, add this control to the header `<div>`:

```
<span class="breadcrumb">
    <asp:SiteMapPath ID="SiteMapPath1" runat="server">
    </asp:SiteMapPath>
</span>
```

The breadcrumb shows the current page the user is visiting in the site map hierarchy as well as that site map node's "ancestors," all the way up to the root (Home, in our site map).

Figure 12: The Breadcrumb Displays the Current Page and its Ancestors in the Site Map Hierarchy

Step 5: Adding the Default Page for Each Section

The tutorials in our site are broken down into different categories Basic Reporting, Filtering, Custom Formatting, and so on with a folder for each category and the corresponding tutorials as ASP.NET pages within that folder. Additionally, each folder contains a `Default.aspx` page. For this default page, let's display all of the tutorials for the current section. That is, for the `Default.aspx` in the `BasicReporting` folder we'd have links to `SimpleDisplay.aspx`, `DeclarativeParams.aspx`, and `ProgrammaticParams.aspx`. Here, again, we can use the `SiteMap` class and a data Web control to display this information based upon the site map defined in `Web.sitemap`.

Let's display an unordered list using a Repeater again, but this time we'll display the title and description of the

tutorials. Since the markup and code to accomplish this will need to be repeated for each `Default.aspx` page, we can encapsulate this UI logic in a User Control. Create a folder in the website called `UserControls` and add to that a new item of type Web User Control named `SectionLevelTutorialListing.ascx`, and add the following markup:

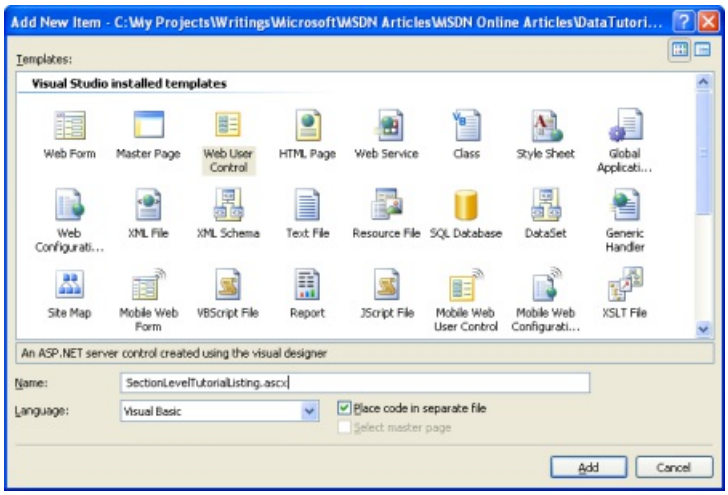


Figure 13: Add a New Web User Control to the `UserControls` Folder (Click to view full-size image)

SectionLevelTutorialListing.ascx

```
<%@ Control Language="VB" AutoEventWireup="true"
    CodeFile="SectionLevelTutorialListing.ascx.vb"
    Inherits="UserControls_SectionLevelTutorialListing" %>
<asp:Repeater ID="TutorialList" runat="server" EnableViewState="False">
    <HeaderTemplate><ul></HeaderTemplate>
    <ItemTemplate>
        <li><asp:HyperLink runat="server"
         NavigateUrl='<%# Eval("Url") %>'
         Text='<%# Eval("Title") %>'></asp:HyperLink>
                - <%# Eval("Description") %></li>
    </ItemTemplate>
    <FooterTemplate></ul></FooterTemplate>
</asp:Repeater>
```

SectionLevelTutorialListing.ascx.vb

```
Partial Class UserControls_SectionLevelTutorialListing
    Inherits UserControl

    Protected Sub Page_Load(sender As Object, e As EventArgs) Handles Me.Load
        If SiteMap.CurrentNode IsNot Nothing Then
            TutorialList.DataSource = SiteMap.CurrentNode.ChildNodes
            TutorialList.DataBind()
        End If
    End Sub
End Class
```

In the previous Repeater example we bound the `SiteMap` data to the Repeater declaratively; the `SectionLevelTutorialListing` User Control, however, does so programmatically. In the `Page_Load` event handler, a check is made to ensure that this page s URL maps to a node in the site map. If this User Control is used in a page that does not have a corresponding `<siteMapNode>` entry, `SiteMap.CurrentNode` will return `Nothing` and no data will be bound to the Repeater. Assuming we have a `CurrentNode`, we bind its `ChildNodes` collection to the Repeater. Since our site map is set up such that the `Default.aspx` page in each section is the parent node of all of the tutorials within that section, this code will display links to and descriptions of all of the section's tutorials, as shown in the screen shot below.

Once this Repeater has been created, open the `Default.aspx` pages in each of the folders, go to the Design view, and simply drag the User Control from the Solution Explorer onto the Design surface where you want the tutorial list to appear.

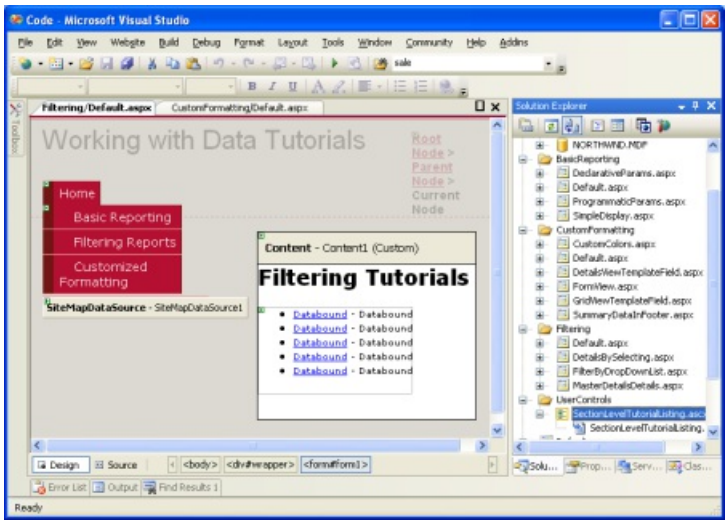


Figure 14: The User Control has Been Added to `Default.aspx` (Click to view full-size image)

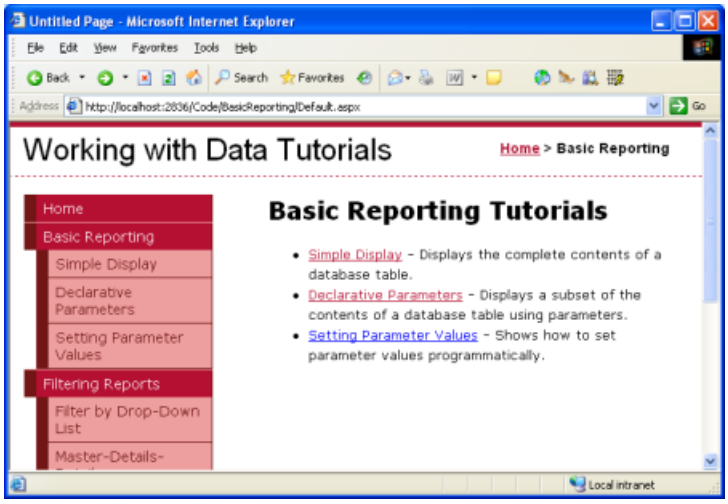


Figure 15: The Basic Reporting Tutorials are Listed (Click to view full-size image)

Summary

With the site map defined and the master page complete, we now have a consistent page layout and navigation scheme for our data-related tutorials. Regardless of how many pages we add to our site, updating the site-wide page layout or site navigation information is a quick and simple process due to this information being centralized. Specifically, the page layout information is defined in the master page `Site.master` and the site map in `Web.sitemap`. We didn't need to write *any* code to achieve this site-wide page layout and navigation mechanism, and we retain full WYSIWYG designer support in Visual Studio.

Having completed the Data Access Layer and Business Logic Layer and having a consistent page layout and site navigation defined, we're ready to begin exploring common reporting patterns. In the next three tutorials we'll look at basic reporting tasks displaying data retrieved from the BLL in the GridView, DetailsView, and FormView controls.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Master Pages Overview
- Master Pages in ASP.NET 2.0
- ASP.NET 2.0 Design Templates
- ASP.NET Site Navigation Overview
- Examining ASP.NET 2.0's Site Navigation
- ASP.NET 2.0 Site Navigation Features
- Understanding ASP.NET View State
- How to: Enable Tracing for an ASP.NET Page
- ASP.NET User Controls

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Liz Shulok, Dennis Patterson, and Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Basic Reporting

2/20/2020 • 2 minutes to read • Edit Online

This set of tutorials looks at how to use the ObjectDataSource control to work with data retrieved from the BLL, all without code.

- Displaying Data With the ObjectDataSource (C#)
- Declarative Parameters (C#)
- Programmatically Setting the ObjectDataSource's Parameter Values (C#)
- Displaying Data With the ObjectDataSource (VB)
- Declarative Parameters (VB)
- Programmatically Setting the ObjectDataSource's Parameter Values (VB)

Displaying Data With the ObjectDataSource (C#)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> This tutorial looks at the ObjectDataSource control Using this control you can bind data retrieved from the BLL created in the previous tutorial without having to write a line of code!

Introduction

With our application architecture and website page layout complete, we're ready to start exploring how to accomplish a variety of common data- and reporting-related tasks. In the previous tutorials we've seen how to programmatically bind data from the DAL and BLL to a data Web control in an ASP.NET page. This syntax assigning the data Web control's `DataSource` property to the data to display and then calling the control's `DataBind()` method was the pattern used in ASP.NET 1.x applications, and can continue to be used in your 2.0 applications. However, ASP.NET 2.0's new data source controls offer a declarative way to work with data. Using these controls you can bind data retrieved from the BLL created in the previous tutorial without having to write a line of code!

ASP.NET 2.0 ships with five built-in data source controls SqlDataSource, AccessDataSource, ObjectDataSource, XmlDataSource, and SiteMapDataSource although you can build your own custom data source controls, if needed. Since we have developed an architecture for our tutorial application, we'll be using the ObjectDataSource against our BLL classes.

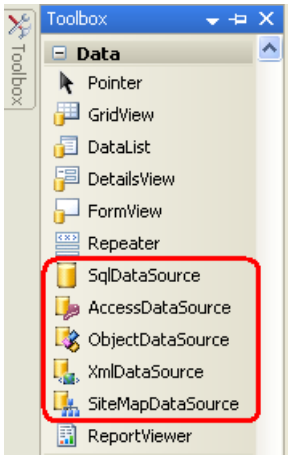


Figure 1: ASP.NET 2.0 Includes Five Built-In Data Source Controls

The ObjectDataSource serves as a proxy for working with some other object. To configure the ObjectDataSource we specify this underlying object and how its methods map to the ObjectDataSource's `Select`, `Insert`, `Update`, and `Delete` methods. Once this underlying object has been specified and its methods mapped to the ObjectDataSource's, we can then bind the ObjectDataSource to a data Web control. ASP.NET ships with many data Web controls, including the GridView, DetailsView, RadioButtonList, and DropDownList, among others. During the page lifecycle, the data Web control may need to access the data it's bound to, which it will accomplish by invoking its ObjectDataSource's `Select` method; if the data Web control supports inserting, updating, or deleting, calls may be made to its ObjectDataSource's `Insert`, `Update`, or `Delete` methods. These calls are then routed by the ObjectDataSource to the appropriate underlying object's methods as the following diagram illustrates.

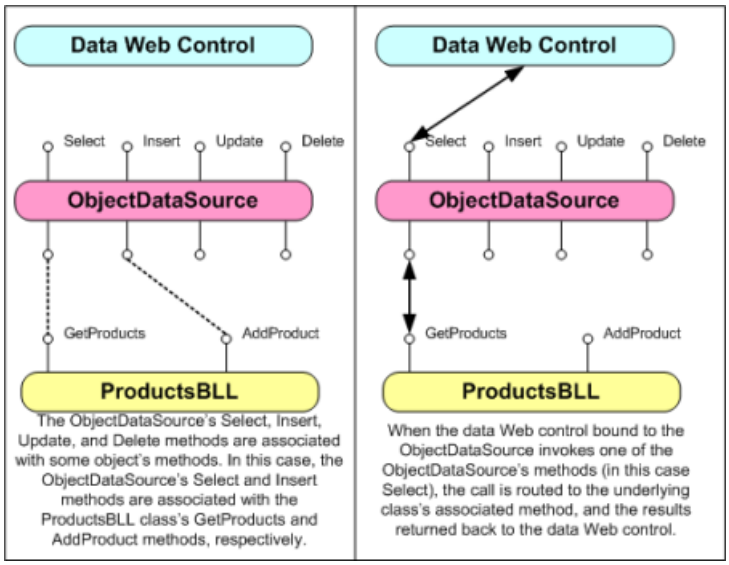


Figure 2: The ObjectDataSource Serves as a Proxy (Click to view full-size image)

While the ObjectDataSource can be used to invoke methods for inserting, updating, or deleting data, let's just focus on returning data; future tutorials will explore using the ObjectDataSource and data Web controls that modify data.

Step 1: Adding and Configuring the ObjectDataSource Control

Start by opening the `SimpleDisplay.aspx` page in the `BasicReporting` folder, switch to Design view, and then drag an ObjectDataSource control from the Toolbox onto the page's design surface. The ObjectDataSource appears as a gray box on the design surface because it does not produce any markup; it simply accesses data by invoking a method from a specified object. The data returned by an ObjectDataSource can be displayed by a data Web control, such as the GridView, DetailsView, FormView, and so on.

> **NOTE**
>
> Alternatively, you may first add the data Web control to the page and then, from its smart tag, choose the <New data source> option from the drop-down list.

To specify the ObjectDataSource's underlying object and how that object's methods map to the ObjectDataSource's, click on the Configure Data Source link from the ObjectDataSource's smart tag.

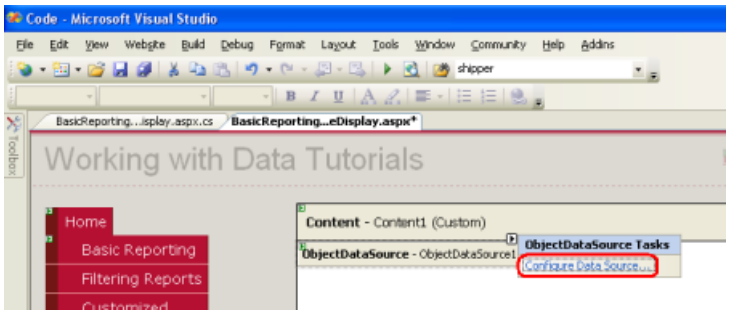


Figure 3: Click the Configure Data Source Link from the Smart Tag (Click to view full-size image)

This brings up the Configure Data Source wizard. First, we must specify the object the ObjectDataSource is to work with. If the "Show only data components" checkbox is checked, the drop-down list on this screen lists only those objects that have been decorated with the `DataObject` attribute. Currently our list includes the TableAdapters in the Typed DataSet and the BLL classes we created in the previous tutorial. If you forgot to add the `DataObject` attribute to the Business Logic Layer classes you won't see them in this list. In that case, uncheck the "Show only data components" checkbox to view all objects, which should include the BLL classes (along with

the other classes in the Typed DataSet the DataTables, DataRows, and so on).

From this first screen choose the `ProductsBLL` class from the drop-down list and click Next.

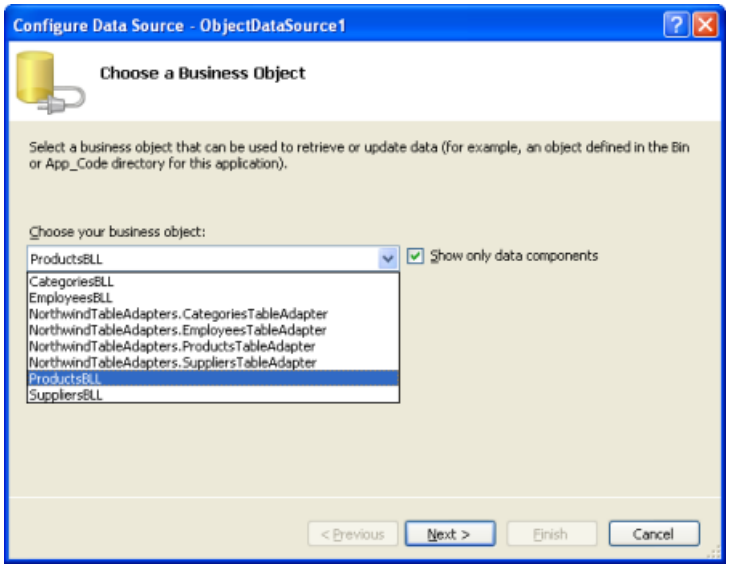


Figure 4: Specify the Object to Use with the ObjectDataSource Control (Click to view full-size image)

The next screen in the wizard prompts you to select what method the ObjectDataSource should invoke. The drop-down lists those methods that return data in the object selected from the previous screen. Here we see `GetProductByProductID`, `GetProducts`, `GetProductsByCategoryID`, and `GetProductsBySupplierID`. Select the `GetProducts` method from the drop-down list and click Finish (if you added the `DataObjectMethodAttribute` to the `ProductBLL`'s methods as shown in the previous tutorial, this option will be selected by default).

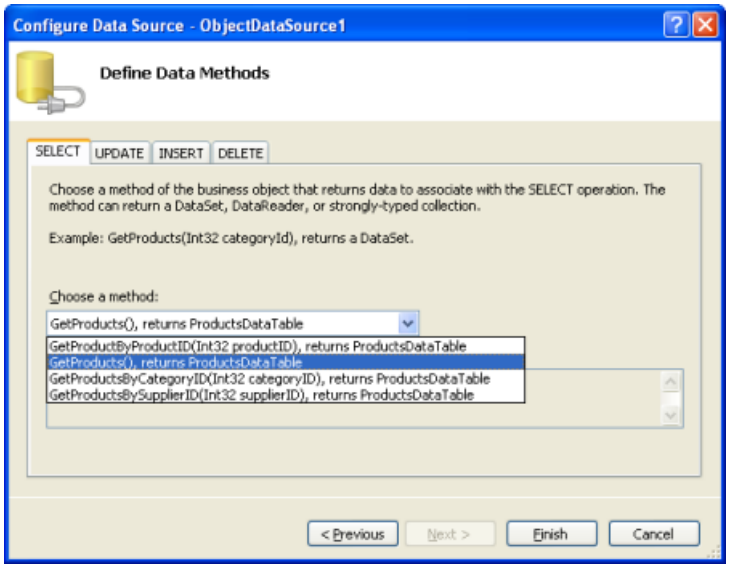


Figure 5: Choose the Method for Returning Data from the SELECT Tab (Click to view full-size image)

Configure the ObjectDataSource Manually

The ObjectDataSource's Configure Data Source wizard offers a quick way to specify the object it uses and to associate what methods of the object are invoked. You can, however, configure the ObjectDataSource through its properties, either through the Properties window or directly in the declarative markup. Simply set the `TypeName` property to the type of the underlying object to be used, and the `SelectMethod` to the method to invoke when retrieving data.

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
</asp:ObjectDataSource>
```

Even if you prefer the Configure Data Source wizard there may be times when you need to manually configure the ObjectDataSource, as the wizard only lists developer-created classes. If you want to bind the ObjectDataSource to a class in the .NET Framework such as the Membership class, to access user account information, or the Directory class to work with file system information you'll need to manually set the ObjectDataSource's properties.

Step 2: Adding a Data Web Control and Binding it to the ObjectDataSource

Once the ObjectDataSource has been added to the page and configured, we're ready to add data Web controls to the page to display the data returned by the ObjectDataSource's `Select` method. Any data Web control can be bound to an ObjectDataSource; let's look at displaying the ObjectDataSource's data in a GridView, DetailsView, and FormView.

Binding a GridView to the ObjectDataSource

Add a GridView control from the Toolbox to `SimpleDisplay.aspx`'s design surface. From the GridView's smart tag, choose the ObjectDataSource control we added in Step 1. This will automatically create a BoundField in the GridView for each property returned by the data from the ObjectDataSource's `Select` method (namely, the properties defined by the Products DataTable).

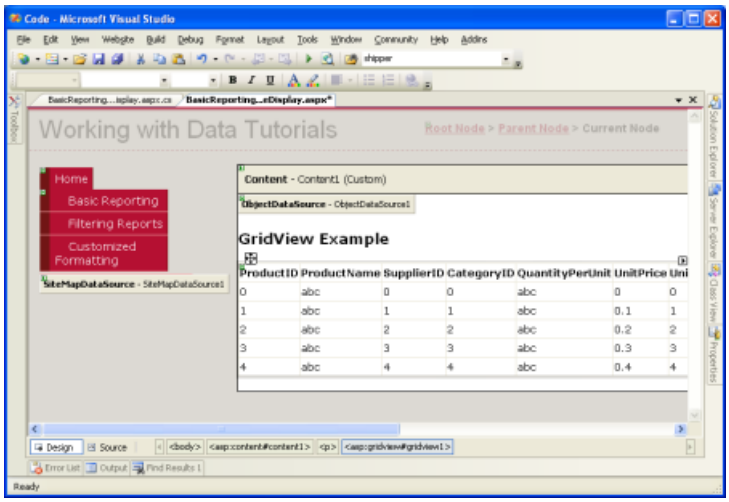


Figure 6: A GridView Has Been Added to the Page and Bound to the ObjectDataSource (Click to view full-size image)

You can then customize, rearrange, or remove the GridView's BoundFields by clicking the Edit Columns option from the smart tag.

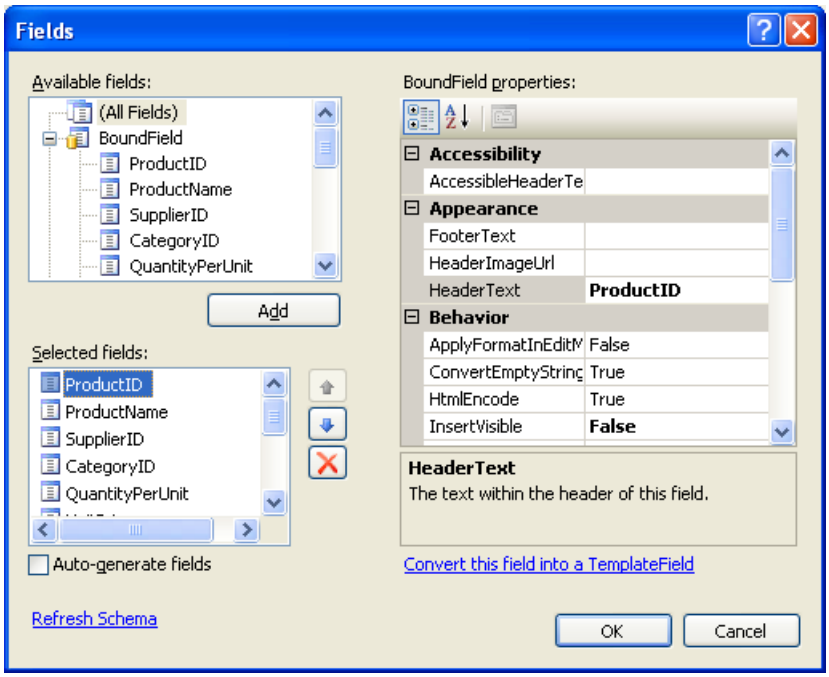


Figure 7: Manage the GridView's BoundFields Through the Edit Columns Dialog Box (Click to view full-size image)

Take a moment to modify the GridView's BoundFields, removing the `ProductID`, `SupplierID`, `CategoryID`, `QuantityPerUnit`, `UnitsInStock`, `UnitsOnOrder`, and `ReorderLevel` BoundFields. Simply select the BoundField from the list in the bottom left and click the delete button (the red X) to remove them. Next, Rearrange the BoundFields so that the `CategoryName` and `SupplierName` BoundFields precede the `UnitPrice` BoundField by selecting these BoundFields and clicking the up arrow. Set the `HeaderText` properties of the remaining BoundFields to `Products`, `Category`, `Supplier`, and `Price`, respectively. Next, have the `Price` BoundField formatted as a currency by setting the BoundField's `HtmlEncode` property to False and its `DataFormatString` property to `{0:c}`. Finally, horizontally align the `Price` to the right and the `Discontinued` checkbox in the center via the `ItemStyle` / `HorizontalAlign` property.

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductName"
         HeaderText="Product" SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName"
          HeaderText="Category" ReadOnly="True"
          SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName"
          HeaderText="Supplier" ReadOnly="True"
          SortExpression="SupplierName" />
        <asp:BoundField DataField="UnitPrice"
          DataFormatString="{0:c}" HeaderText="Price"
            HtmlEncode="False" SortExpression="UnitPrice">
            <ItemStyle HorizontalAlign="Right" />
        </asp:BoundField>
        <asp:CheckBoxField DataField="Discontinued"
          HeaderText="Discontinued" SortExpression="Discontinued">
            <ItemStyle HorizontalAlign="Center" />
        </asp:CheckBoxField>
    </Columns>
</asp:GridView>
```

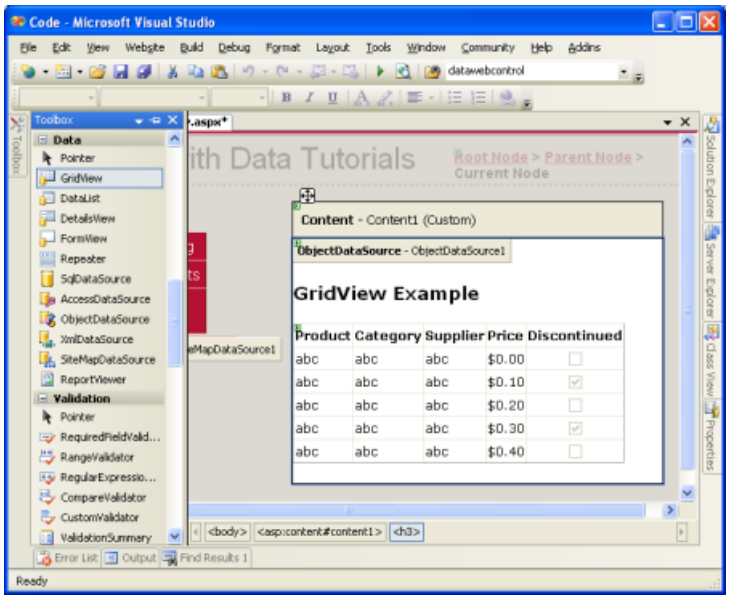


Figure 8: The GridView's BoundFields Have Been Customized (Click to view full-size image)

Using Themes for a Consistent Look

These tutorials strive to remove any control-level style settings, instead using cascading style sheets defined in an external file whenever possible. The `Styles.css` file contains `DataWebControlStyle`, `HeaderStyle`, `RowStyle`, and `AlternatingRowStyle` CSS classes that should be used to dictate the appearance of the data Web controls used in these tutorials. To accomplish this, we could set the GridView's `CssClass` property to `DataWebControlStyle`, and its `HeaderStyle`, `RowStyle`, and `AlternatingRowStyle` properties' `CssClass` properties accordingly.

If we set these `CssClass` properties at the Web control we'd need to remember to explicitly set these property values for each and every data Web control added to our tutorials. A more manageable approach is to define the default CSS-related properties for the GridView, DetailsView, and FormView controls using a Theme. A Theme is a collection of control-level property settings, images, and CSS classes that can be applied to pages across a site to enforce a common look and feel.

Our Theme won't include any images or CSS files (we'll leave the stylesheet `Styles.css` as-is, defined in the root folder of the web application), but will include two Skins. A Skin is a file that defines the default properties for a Web control. Specifically, we'll have a Skin file for the GridView and DetailsView controls, indicating the default `CssClass`-related properties.

Start by adding a new Skin File to your project named `GridView.skin` by right-clicking on the project name in the Solution Explorer and choosing Add New Item.

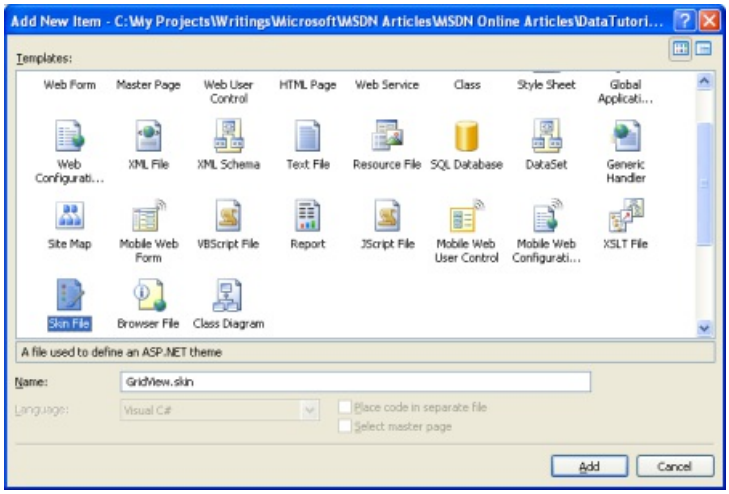


Figure 9: Add a Skin File Named `GridView.skin` (Click to view full-size image)

Skin files need to be placed in a Theme, which are located in the `App_Themes` folder. Since we don't yet have such a folder, Visual Studio will kindly offer to create one for us when adding our first Skin. Click Yes to create the `App_Theme` folder and place the new `GridView.skin` file there.

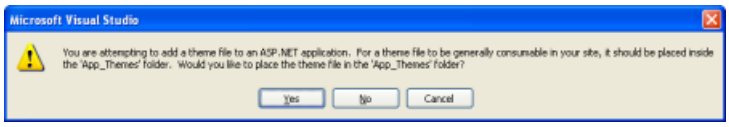


Figure 10: Let Visual Studio Create the `App_Theme` Folder (Click to view full-size image)

This will create a new Theme in the `App_Themes` folder named GridView with the Skin file `GridView.skin`.

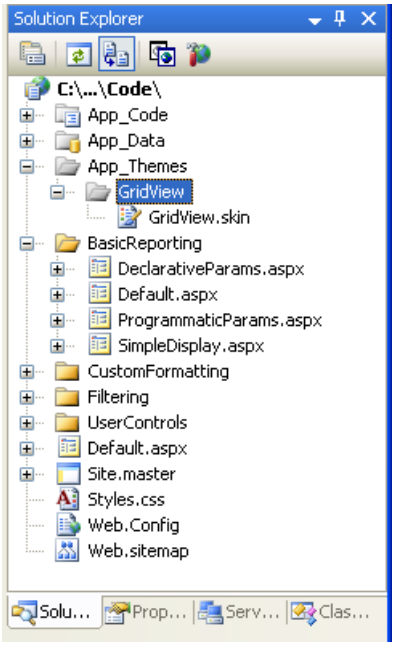


Figure 11: The GridView Theme has Been Added to the `App_Theme` Folder

Rename the GridView Theme to DataWebControls (right-click on the GridView folder in the `App_Theme` folder and choose Rename). Next, enter the following markup into the `GridView.skin` file:

```
<asp:GridView runat="server" CssClass="DataWebControlStyle">
   <AlternatingRowStyle CssClass="AlternatingRowStyle" />
   <RowStyle CssClass="RowStyle" />
   <HeaderStyle CssClass="HeaderStyle" />
</asp:GridView>
```

This defines the default properties for the `CssClass`-related properties for any GridView in any page that uses the DataWebControls Theme. Let's add another Skin for the DetailsView, a data Web control that we'll be using shortly. Add a new Skin to the DataWebControls Theme named `DetailsView.skin` and add the following markup:

```
<asp:DetailsView runat="server" CssClass="DataWebControlStyle">
   <AlternatingRowStyle CssClass="AlternatingRowStyle" />
   <RowStyle CssClass="RowStyle" />
   <FieldHeaderStyle CssClass="HeaderStyle" />
</asp:DetailsView>
```

With our Theme defined, the last step is to apply the Theme to our ASP.NET page. A Theme can be applied on a page-by-page basis or for all pages in a website. Let's use this theme for all pages in the website. To accomplish

this, add the following markup to `Web.config`'s `<system.web>` section:

```
<pages styleSheetTheme="DataWebControls" />
```

That's all there is to it! The `styleSheetTheme` setting indicates that the properties specified in the Theme should *not* override the properties specified at the control level. To specify that Theme settings should trump control settings, use the `theme` attribute in place of `styleSheetTheme`; unfortunately, Theme settings specified via the `theme` attribute do not appear in the Visual Studio Design view. Refer to ASP.NET Themes and Skins Overview and Server-Side Styles Using Themes for more information on Themes and Skins; see How To: Apply ASP.NET Themes for more on configuring a page to use a theme.

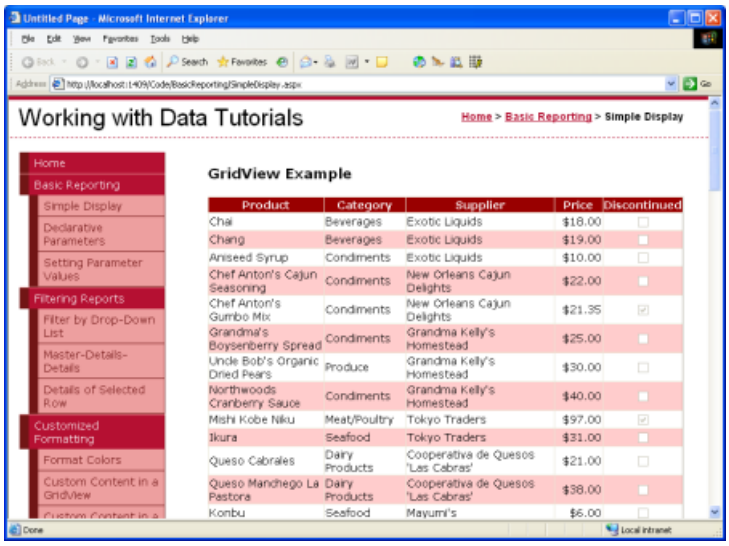


Figure 12: The GridView Displays the Product's Name, Category, Supplier, Price, and Discontinued Information (Click to view full-size image)

Displaying One Record at a Time in the DetailsView

The GridView displays one row for each record returned by the data source control to which it's bound. There are times, however, when we may want to display a sole record or just one record at a time. The DetailsView control offers this functionality, rendering as an HTML `<table>` with two columns and one row for each column or property bound to the control. You can think of the DetailsView as a GridView with a single record rotated 90 degrees.

Start by adding a DetailsView control *above* the GridView in `SimpleDisplay.aspx`. Next, bind it to the same ObjectDataSource control as the GridView. Like with the GridView, a BoundField will be added to the DetailsView for each property in the object returned by the ObjectDataSource's `Select` method. The only difference is that the DetailsView's BoundFields are laid out horizontally rather than vertically.

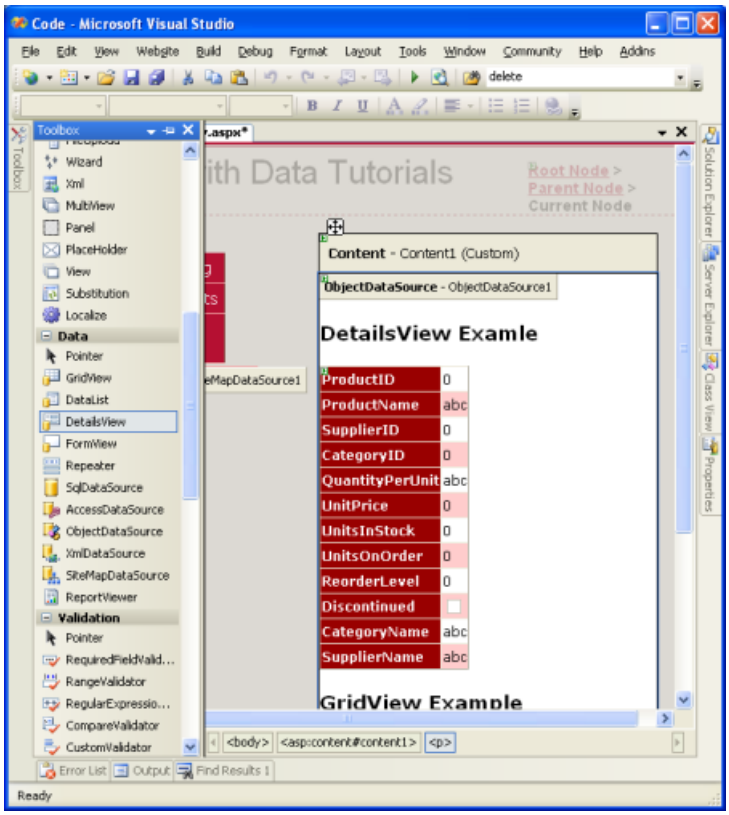


Figure 13: Add a DetailsView to the Page and Bind it to the ObjectDataSource (Click to view full-size image)

Like the GridView, the DetailsView's BoundFields can be tweaked to provide a more customized display of the data returned by the ObjectDataSource. Figure 14 shows the DetailsView after its BoundFields and `CssClass` properties have been configured to make its appearance similar to the GridView example.

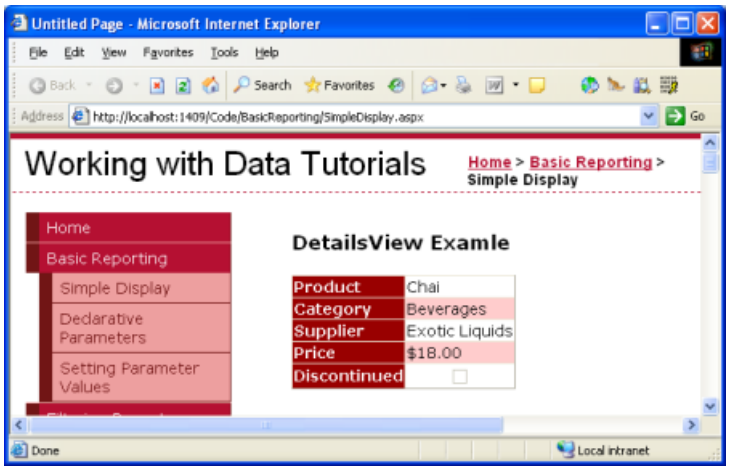


Figure 14: The DetailsView Shows a Single Record (Click to view full-size image)

Note that the DetailsView only displays the first record returned by its data source. To allow the user to step through all of the records, one at a time, we must enable paging for the DetailsView. To do so, return to Visual Studio and check the Enable Paging checkbox in the DetailsView's smart tag.

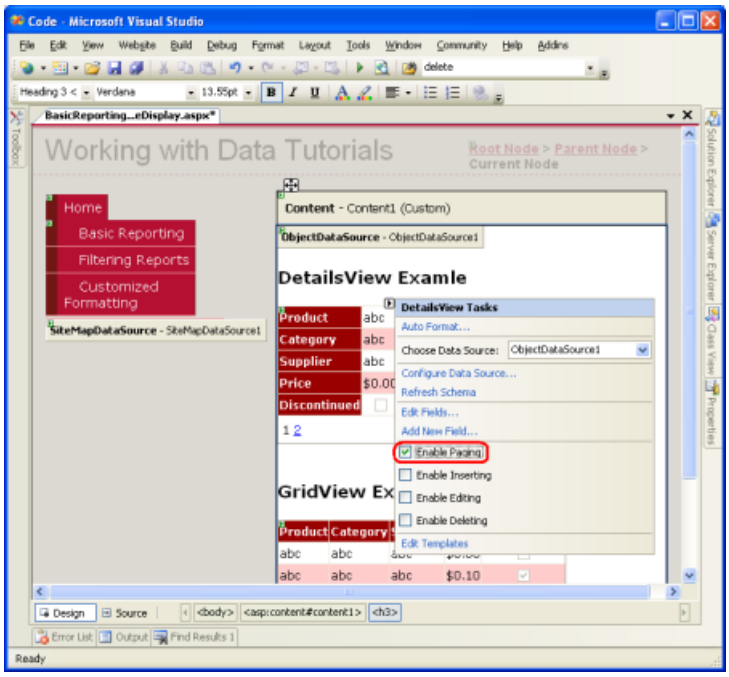


Figure 15: Enable Paging in the DetailsView Control (Click to view full-size image)

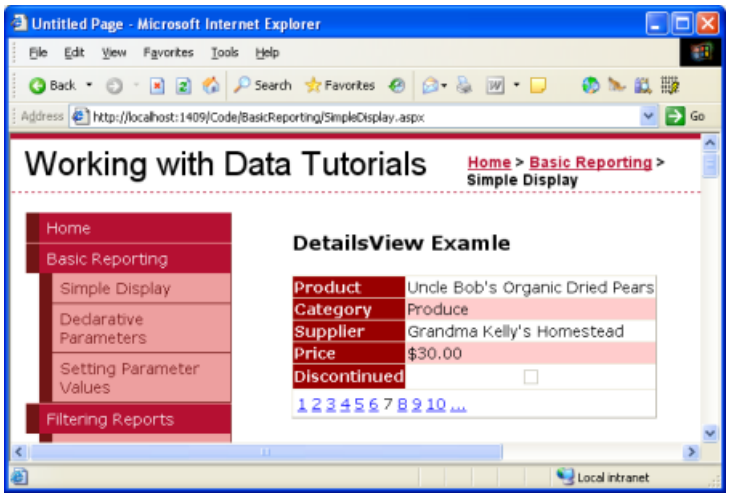


Figure 16: With Paging Enabled, the DetailsView Allows the User to View Any of the Products (Click to view full-size image)

We'll talk more about paging in future tutorials.

A More Flexible Layout for Showing One Record at a Time

The DetailsView is pretty rigid in how it displays each record returned from the ObjectDataSource. We may want a more flexible view of the data. For example, rather than showing the product's name, category, supplier, price, and discontinued information each on a separate row, we may want to show the product name and price in an `<h4>` heading, with the category and supplier information appearing below the name and price in a smaller font size. And we may not care to show the property names (Product, Category, and so on) next to the values.

The FormView control provides this level of customization. Rather than using fields (like the GridView and DetailsView do), the FormView uses templates, which allow for a mix of Web controls, static HTML, and databinding syntax. If you are familiar with the Repeater control from ASP.NET 1.x, you can think of the FormView as the Repeater for showing a single record.

Add a FormView control to the `SimpleDisplay.aspx` page's design surface. Initially the FormView displays as a gray block, informing us that we need to provide, at minimum, the control's `ItemTemplate`.

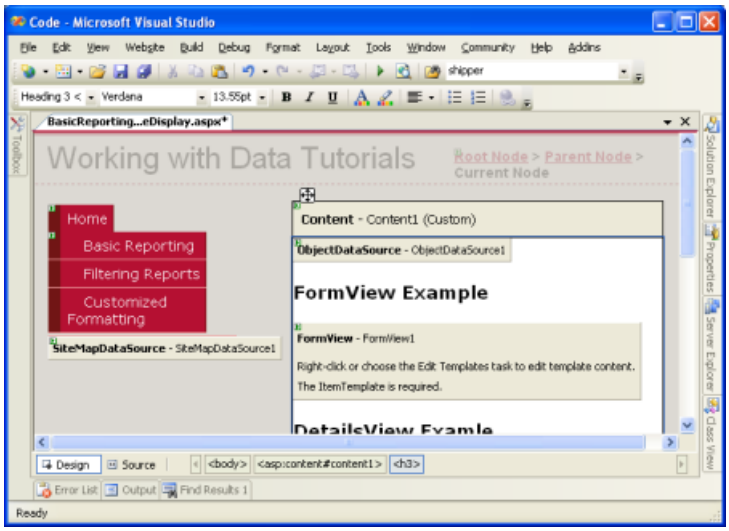


Figure 17: The FormView Must Include an `ItemTemplate` (Click to view full-size image)

You can bind the FormView directly to a data source control through the FormView's smart tag, which will create a default `ItemTemplate` automatically (along with an `EditItemTemplate` and `InsertItemTemplate`, if the ObjectDataSource control's `InsertMethod` and `UpdateMethod` properties are set). However, for this example let's bind the data to the FormView and specify its `ItemTemplate` manually. Start by setting the FormView's `DataSourceID` property to the `ID` of the ObjectDataSource control, `ObjectDataSource1`. Next, create the `ItemTemplate` so that it displays the product's name and price in an `<h4>` element and the category and shipper names beneath that in a smaller font size.

```
<asp:FormView ID="FormView1" runat="server"
  DataSourceID="ObjectDataSource1" EnableViewState="False">
    <ItemTemplate>
        <h4><%# Eval("ProductName") %>
          (<%# Eval("UnitPrice", "{0:c}") %>)</h4>
        Category: <%# Eval("CategoryName") %>;
        Supplier: <%# Eval("SupplierName") %>
    </ItemTemplate>
</asp:FormView>
```

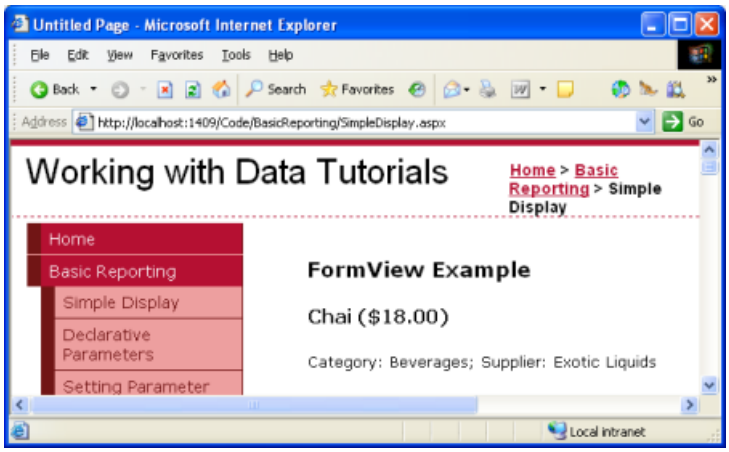


Figure 18: The First Product (Chai) is Displayed in a Custom Format (Click to view full-size image)

The `<%# Eval(propertyName) %>` is the databinding syntax. The `Eval` method returns the value of the specified property for the current object being bound to the FormView control. Check out Alex Homer's article Simplified and Extended Data Binding Syntax in ASP.NET 2.0 for more information on the ins and outs of databinding.

Like the DetailsView, the FormView only shows the first record returned from the ObjectDataSource. You can enable paging in the FormView to allow visitors to step through the products one at a time.

Summary

Accessing and displaying data from a Business Logic Layer can be accomplished without writing a line of code thanks to ASP.NET 2.0's ObjectDataSource control. The ObjectDataSource invokes a specified method of a class and returns the results. These results can be displayed in a data Web control that's bound to the ObjectDataSource. In this tutorial we looked at binding the GridView, DetailsView, and FormView controls to the ObjectDataSource.

So far we've only seen how to use the ObjectDataSource to invoke a parameter-less method, but what if we want to invoke a method that expects input parameters, such as the `ProductBLL` class's `GetProductsByCategoryID(categoryID)`? In order to call a method that expects one or more parameters we must configure the ObjectDataSource to specify the values for these parameters. We'll see how to accomplish this in our next tutorial.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Create Your Own Data Source Controls
- GridView Examples for ASP.NET 2.0
- Simplified and Extended Data Binding Syntax in ASP.NET 2.0
- Themes in ASP.NET 2.0
- Server-Side Styles Using Themes
- How To: Apply ASP.NET Themes Programmatically

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Declarative Parameters (C#)

2/20/2020 • 8 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll illustrate how to use a parameter set to a hard-coded value to select the data to display in a DetailsView control.

Introduction

In the last tutorial we looked at displaying data with the GridView, DetailsView, and FormView controls bound to an ObjectDataSource control that invoked the `GetProducts()` method from the `ProductsBLL` class. The `GetProducts()` method returns a strongly-typed DataTable populated with all of the records from the Northwind database's `Products` table. The `ProductsBLL` class contains additional methods for returning just subsets of the products - `GetProductByProductID(productID)`, `GetProductsByCategoryID(categoryID)`, and `GetProductsBySupplierID(supplierID)`. These three methods expect an input parameter indicating how to filter the returned product information.

The ObjectDataSource can be used to invoke methods that expect input parameters, but in order to do so we must specify where the values for these parameters come from. The parameter values can be hard-coded or can come from a variety of dynamic sources, including: querystring values, Session variables, the property value of a Web control on the page, or others.

For this tutorial let's start by illustrating how to use a parameter set to a hard-coded value. Specifically, we'll look at adding a DetailsView to the page that displays information about a specific product, namely Chef Anton's Gumbo Mix, which has a `ProductID` of 5. Next, we'll see how to set the parameter value based on a Web control. In particular, we'll use a TextBox to let the user type in a country, after which they can click a Button to see the list of suppliers that reside in that country.

Using a Hard-Coded Parameter Value

For the first example, start by adding a DetailsView control to the `DeclarativeParams.aspx` page in the `BasicReporting` folder. From the DetailsView's smart tag, select <New data source> from the drop-down list and choose to add an ObjectDataSource.

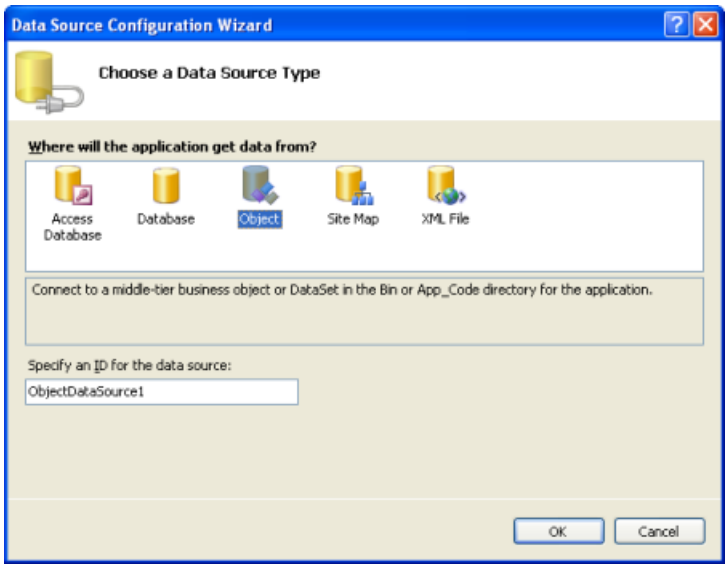


Figure 1: Add an ObjectDataSource to the Page (Click to view full-size image)

This will automatically start the ObjectDataSource control's Choose Data Source wizard. Select the `ProductsBLL` class from the first screen of the wizard.

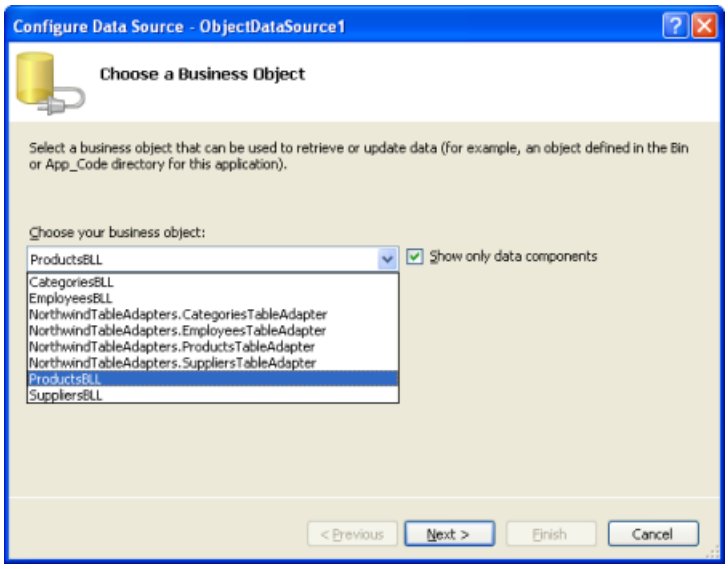


Figure 2: Select the `ProductsBLL` Class (Click to view full-size image)

Since we want to display information about a particular product we want to use the `GetProductByProductID(productID)` method.

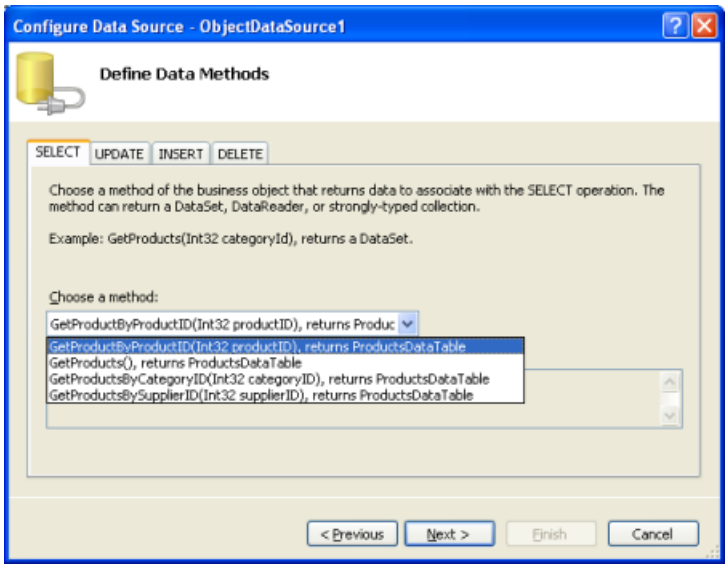


Figure 3: Choose the `GetProductByProductID(productID)` method (Click to view full-size image)

Since the method we selected includes a parameter, there's one more screen for the wizard, where we're asked to define the value to be used for the parameter. The list on the left shows all of the parameters for the selected method. For `GetProductByProductID(productID)` there's only one `productID`. On the right we can specify the value for the selected parameter. The parameter source drop-down list enumerates the various possible sources for the parameter value. Since we want to specify a hard-coded value of 5 for the `productID` parameter, leave the Parameter source as None and enter 5 into the DefaultValue textbox.

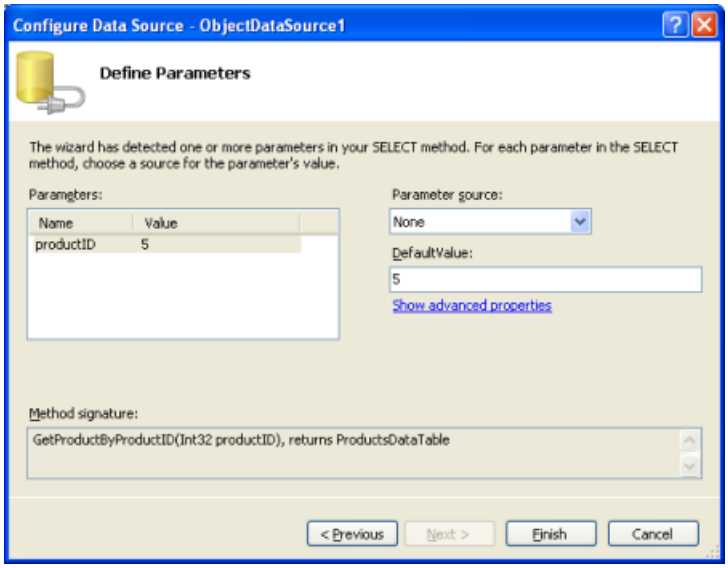


Figure 4: A Hard-Coded Parameter Value of 5 Will Be Used for the `productID` Parameter (Click to view full-size image)

After completing the Configure Data Source wizard, the ObjectDataSource control's declarative markup includes a `Parameter` object in the `SelectParameters` collection for each of the input parameters expected by the method defined in the `SelectMethod` property. Since the method we're using in this example expects just a single input parameter, `parameterID`, there's only one entry here. The `SelectParameters` collection can contain any class that derives from the `Parameter` class in the `System.Web.UI.WebControls` namespace. For hard-coded parameter values the base `Parameter` class is used, but for the other parameter source options a derived `Parameter` class is used; you can also create your own custom parameter types, if needed.

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    SelectMethod="GetProductByProductID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:Parameter DefaultValue="5" Name="productID"
         Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

NOTE

If you're following along on your own computer the declarative markup you see at this point may include values for the `InsertMethod`, `UpdateMethod`, and `DeleteMethod` properties, as well as `DeleteParameters`. The ObjectDataSource's Choose Data Source wizard automatically specifies the methods from the `ProductBLL` to use for inserting, updating, and deleting, so unless you explicitly cleared those out, they'll be included in the markup above.

When visiting this page, the data Web control will invoke the ObjectDataSource's `Select` method, which will call the `ProductsBLL` class's `GetProductByProductID(productID)` method using the hard-coded value of 5 for the `productID` input parameter. The method will return a strongly-typed `ProductDataTable` object that contains a

single row with information about Chef Anton's Gumbo Mix (the product with `ProductID` 5).

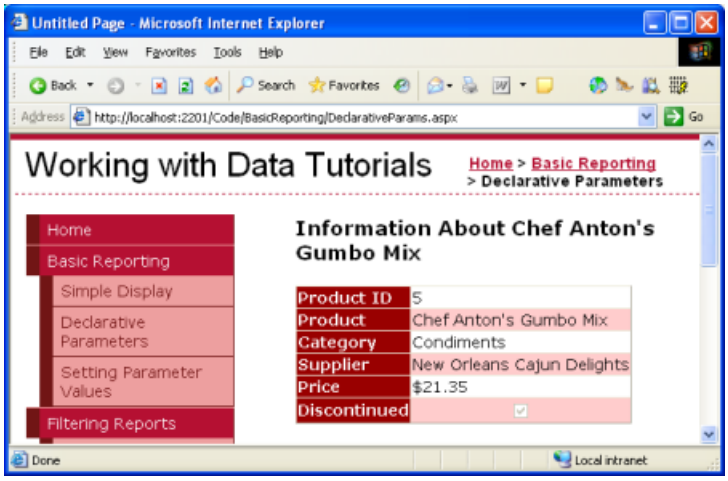


Figure 5: Information About Chef Anton's Gumbo Mix are Displayed (Click to view full-size image)

Setting the Parameter Value to the Property Value of a Web Control

The ObjectDataSource's parameter values can also be set based on the value of a Web control on the page. To illustrate this, let's have a GridView that lists all of the suppliers that are located in a country specified by the user. To accomplish this start by adding a TextBox to the page into which the user can enter a country name. Set this TextBox control's `ID` property to `CountryName`. Also add a Button Web control.

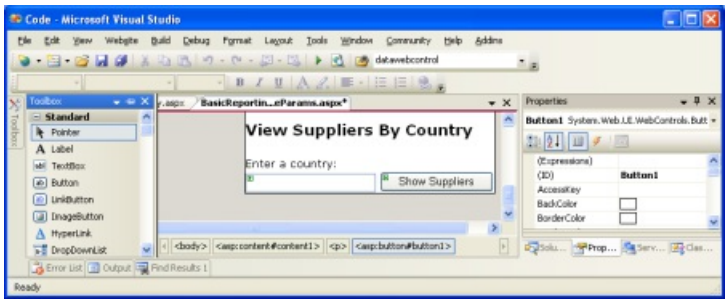


Figure 6: Add a TextBox to the Page with `ID` `CountryName` (Click to view full-size image)

Next, add a GridView to the page and, from the smart tag, choose to add a new ObjectDataSource. Since we want to display supplier information select the `SuppliersBLL` class from the wizard's first screen. From the second screen, pick the `GetSuppliersByCountry(country)` method.

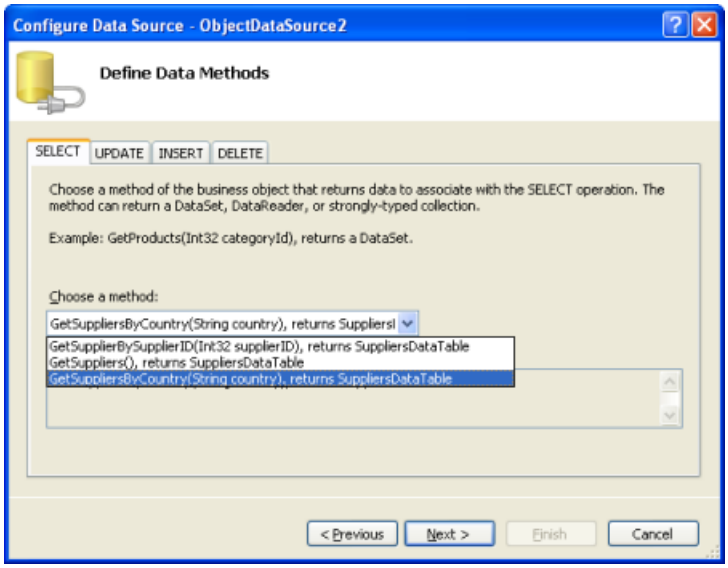


Figure 7: Choose the `GetSuppliersByCountry(country)` Method (Click to view full-size image)

Since the `GetSuppliersByCountry(country)` method has an input parameter, the wizard once again includes a final screen for choosing the parameter value. This time, set the Parameter source to Control. This will populate the ControlID drop-down list with the names of the controls on the page; select the `CountryName` control from the list. When the page is first visited the `CountryName` TextBox will be blank, so no results are returned and nothing is displayed. If you want to display some results by default, set the DefaultValue textbox accordingly.

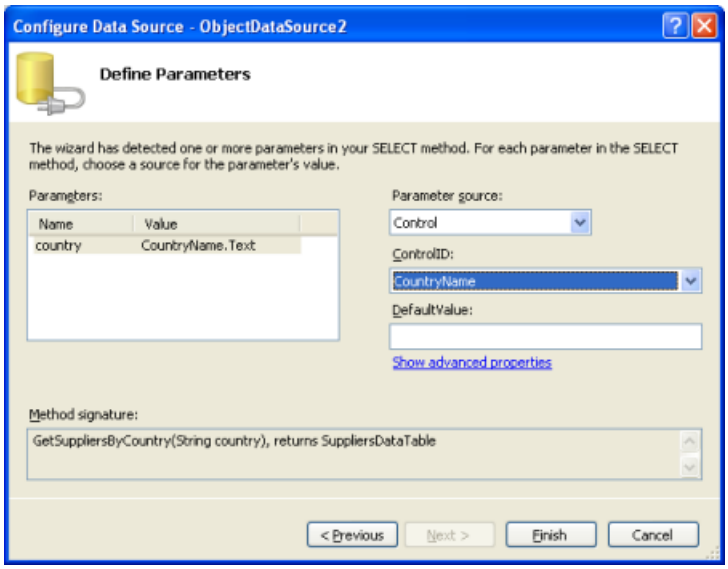


Figure 8: Set the Parameter Value to the `CountryName` Control Value (Click to view full-size image)

The ObjectDataSource's declarative markup differs slightly from our first example, using a ControlParameter instead of the standard `Parameter` object. A `ControlParameter` has additional properties to specify the `ID` of the Web control and the property value to use for the parameter (`PropertyName`). The Configure Data Source wizard was smart enough to determine that, for a TextBox, we'll likely want to use the `Text` property for the parameter value. If, however, you want to use a different property value from the Web control you can change the `PropertyName` value here or by clicking the "Show advanced properties" link in the wizard.

```
<asp:ObjectDataSource ID="ObjectDataSource2" runat="server"
    SelectMethod="GetSuppliersByCountry" TypeName="SuppliersBLL">
    <SelectParameters>
        <asp:ControlParameter ControlID="CountryName"
          Name="country" PropertyName="Text"
            Type="String" />
    </SelectParameters>
</asp:ObjectDataSource>
```

When visiting the page for the first time the `CountryName` TextBox is empty. The ObjectDataSource's `Select` method is still invoked by the GridView, but a value of `null` is passed into the `GetSuppliersByCountry(country)` method. The TableAdapter converts the `null` into a database `NULL` value (`DBNull.Value`), but the query used by the `GetSuppliersByCountry(country)` method is written such that it doesn't return any values when a `NULL` value is specified for the `@CategoryID` parameter. In short, no suppliers are returned.

Once the visitor enters in a country, however, and clicks the Show Suppliers button to cause a postback, the ObjectDataSource's `Select` method is requeried, passing in the TextBox control's `Text` value as the `country` parameter.

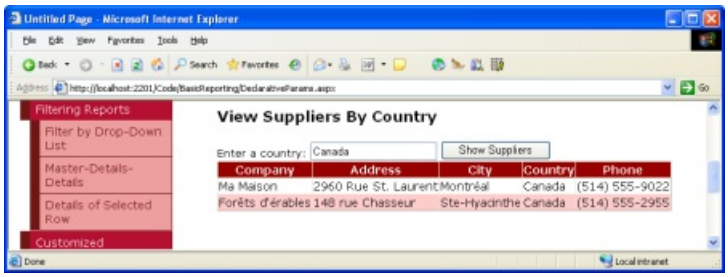


Figure 9: Those Suppliers from Canada are Shown (Click to view full-size image)

Showing All Suppliers By Default

Rather than show none of the suppliers when first viewing the page we may want to show *all* suppliers at first, allowing the user to pare down the list by entering a country name in the TextBox. When the TextBox is empty, the `SuppliersBLL` class's `GetSuppliersByCountry(country)` method is passed in a `null` value for its *`country`* input parameter. This `null` value is then passed down into the DAL's `GetSupplierByCountry(country)` method, where it's translated to a database `NULL` value for the `@Country` parameter in the following query:

```
SELECT     SupplierID, CompanyName, Address, City, Country, Phone
FROM         Suppliers
WHERE Country = @Country
```

The expression `Country = NULL` always returns False, even for records whose `Country` column has a `NULL` value; therefore, no records are returned.

To return *all* suppliers when the country TextBox is empty, we can augment the `GetSuppliersByCountry(country)` method in the BLL to invoke the `GetSuppliers()` method when its country parameter is `null` and to call the DAL's `GetSuppliersByCountry(country)` method otherwise. This will have the effect of returning all suppliers when no country is specified and the appropriate subset of suppliers when the country parameter is included.

Change the `GetSuppliersByCountry(country)` method in the `SuppliersBLL` class to the following:

```
public Northwind.SuppliersDataTable GetSuppliersByCountry(string country)
{
    if (string.IsNullOrEmpty(country))
        return GetSuppliers();
    else
        return Adapter.GetSuppliersByCountry(country);
}
```

With this change the `DeclarativeParams.aspx` page shows all of the suppliers when first visited (or whenever the `CountryName` TextBox is empty).

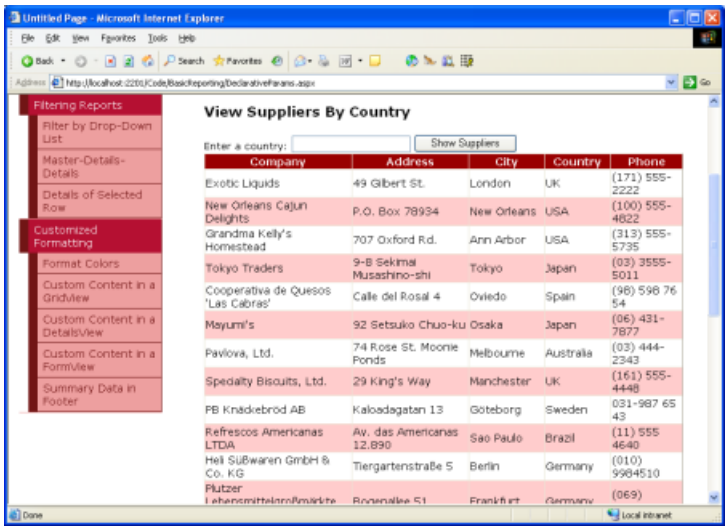


Figure 10: All Suppliers are Now Shown by Default (Click to view full-size image)

Summary

In order to use methods with input parameters, we need to specify the values for the parameters in the ObjectDataSource's `SelectParameters` collection. Different types of parameters allow for the parameter value to be obtained from different sources. The default parameter type uses a hard-coded value, but just as easily (and without a line of code) parameter values can be obtained from the querystring, Session variables, cookies, and even user-entered values from Web controls on the page.

The examples we looked at in this tutorial illustrated how to use declarative parameter values. However, there may be times when we need to use a parameter source that's not available, such as the current date and time, or, if our site was using Membership, the User ID of the visitor. For such scenarios we can set the parameter values programmatically prior to the ObjectDataSource invoking its underlying object's method. We'll see how to accomplish this in the next tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Programmatically Setting the ObjectDataSource's Parameter Values (C#)

2/20/2020 • 6 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll look at adding a method to our DAL and BLL that accepts a single input parameter and returns data. The example will set this parameter programmatically.

Introduction

As we saw in the previous tutorial, a number of options are available for declaratively passing parameter values to the ObjectDataSource's methods. If the parameter value is hard-coded, comes from a Web control on the page, or is in any other source that is readable by a data source `Parameter` object, for example, that value can be bound to the input parameter without writing a line of code.

There may be times, however, when the parameter value comes from some source not already accounted for by one of the built-in data source `Parameter` objects. If our site supported user accounts we might want to set the parameter based on the currently logged in visitor's User ID. Or we may need to customize the parameter value before sending it along to the ObjectDataSource's underlying object's method.

Whenever the ObjectDataSource's `Select` method is invoked the ObjectDataSource first raises its Selecting event. The ObjectDataSource's underlying object's method is then invoked. Once that completes the ObjectDataSource's Selected event fires (Figure 1 illustrates this sequence of events). The parameter values passed into the ObjectDataSource's underlying object's method can be set or customized in an event handler for the `Selecting` event.

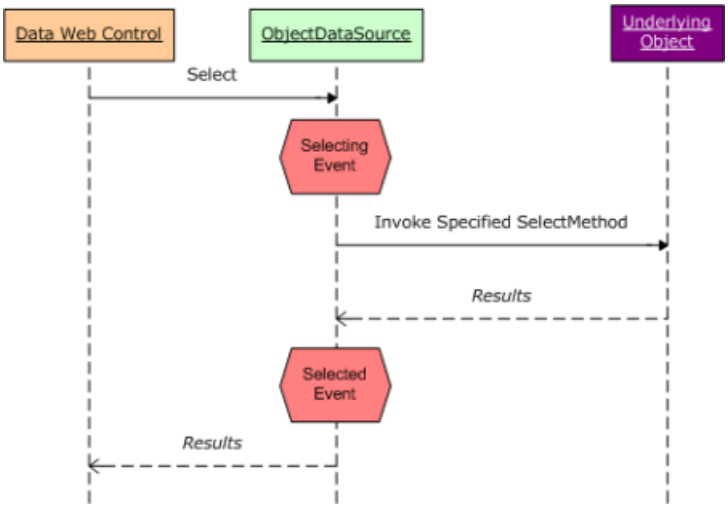


Figure 1: The ObjectDataSource's `Selected` and `Selecting` Events Fire Before and After Its Underlying Object's Method is Invoked (Click to view full-size image)

In this tutorial we'll look at adding a method to our DAL and BLL that accepts a single input parameter `Month`, of type `int` and returns an `EmployeesDataTable` object populated with those employees that have their hiring anniversary in the specified `Month`. Our example will set this parameter programmatically based on the current month, showing a list of "Employee Anniversaries This Month."

Let's get started!

Step 1: Adding a Method to `EmployeesTableAdapter`

For our first example we need to add a means to retrieve those employees whose `HireDate` occurred in a specified month. To provide this functionality in accordance with our architecture we need to first create a method in `EmployeesTableAdapter` that maps to the proper SQL statement. To accomplish this, start by opening the Northwind Typed DataSet. Right-click on the `EmployeesTableAdapter` label and choose Add Query.

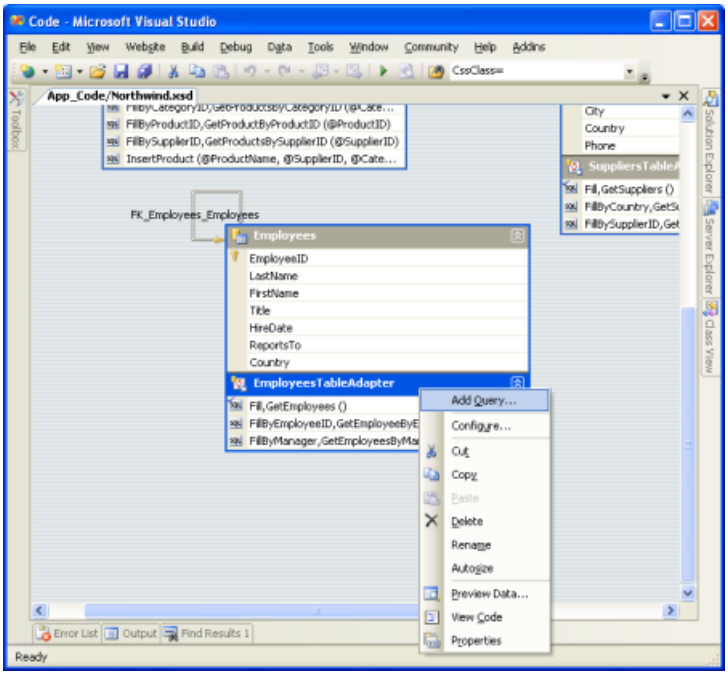


Figure 2: Add a New Query to the `EmployeesTableAdapter` (Click to view full-size image)

Choose to add a SQL statement that returns rows. When you reach the Specify a `SELECT` Statement screen the default `SELECT` statement for the `EmployeesTableAdapter` will already be loaded. Simply add in the `WHERE` clause: `WHERE DATEPART(m, HireDate) = @Month`. DATEPART is a T-SQL function that returns a particular date portion of a `datetime` type; in this case we're using `DATEPART` to return the month of the `HireDate` column.

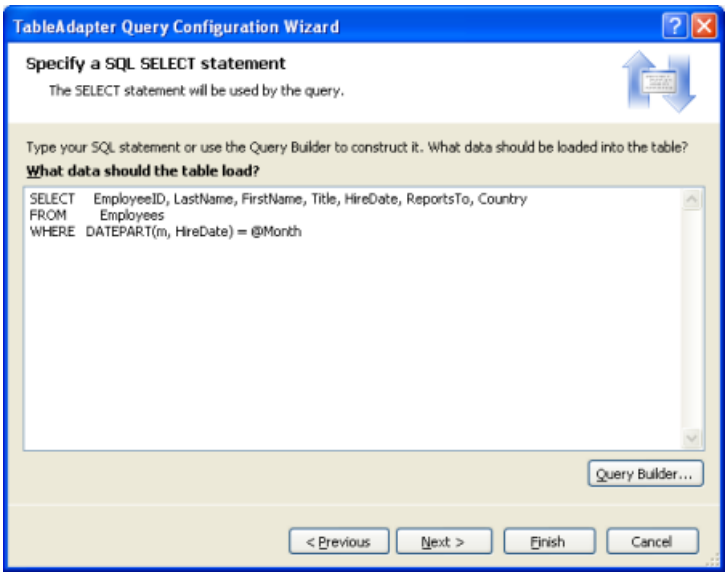


Figure 3: Return Only Those Rows Where the `HireDate` Column is Less Than or Equal to the `@HiredBeforeDate` Parameter (Click to view full-size image)

Finally, change the `FillBy` and `GetDataBy` method names to `FillByHiredDateMonth` and `GetEmployeesByHiredDateMonth`, respectively.

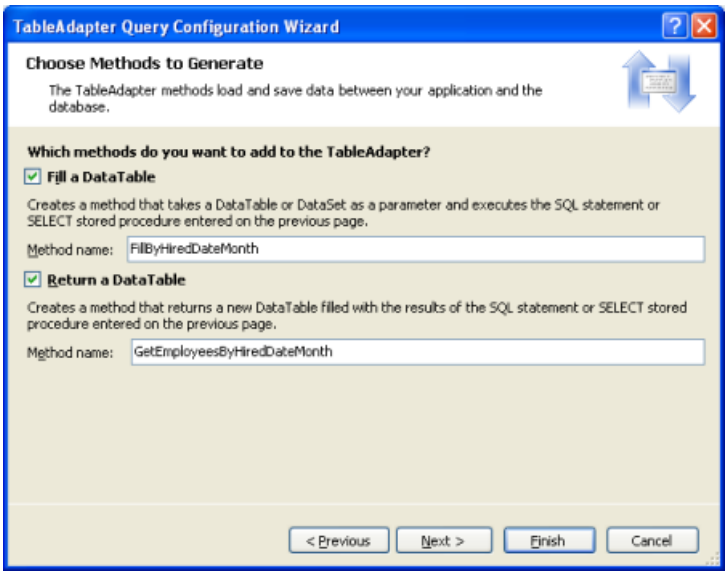


Figure 4: Choose More Appropriate Method Names Than `FillBy` and `GetDataBy` (Click to view full-size image)

Click Finish to complete the wizard and return to the DataSet's design surface. The `EmployeesTableAdapter` should now include a new set of methods for accessing employees hired in a specified month.

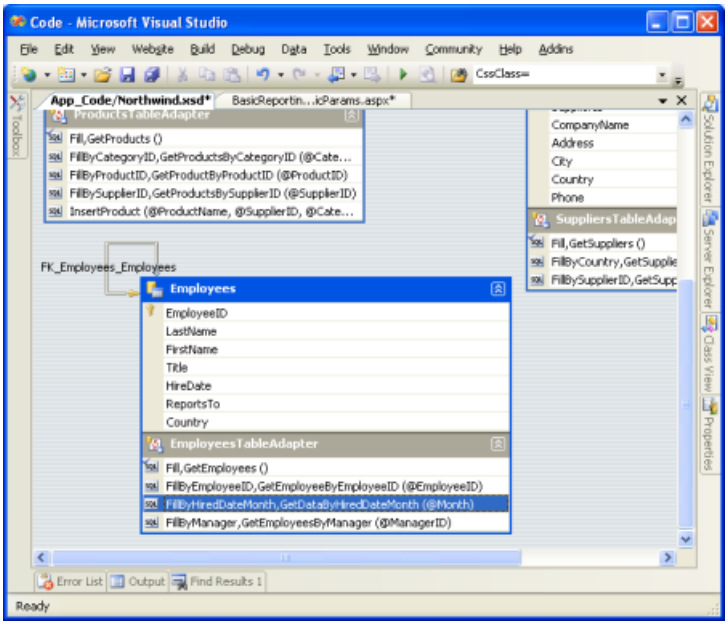


Figure 5: The New Methods Appear in the DataSet's Design Surface (Click to view full-size image)

Step 2: Adding the `GetEmployeesByHiredDateMonth(month)` Method to the Business Logic Layer

Since our application architecture uses a separate layer for the business logic and data access logic, we need to add a method to our BLL that calls down to the DAL to retrieve employees hired before a specified date. Open the `EmployeesBLL.cs` file and add the following method:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Select, false)]
public Northwind.EmployeesDataTable GetEmployeesByHiredDateMonth(int month)
{
    return Adapter.GetEmployeesByHiredDateMonth(month);
}
```

As with our other methods in this class, `GetEmployeesByHiredDateMonth(month)` simply calls down into the DAL and

returns the results.

Step 3: Displaying Employees Whose Hiring Anniversary Is This Month

Our final step for this example is to display those employees whose hiring anniversary is this month. Start by adding a GridView to the `ProgrammaticParams.aspx` page in the `BasicReporting` folder and add a new ObjectDataSource as its data source. Configure the ObjectDataSource to use the `EmployeesBLL` class with the `SelectMethod` set to `GetEmployeesByHiredDateMonth(month)`.

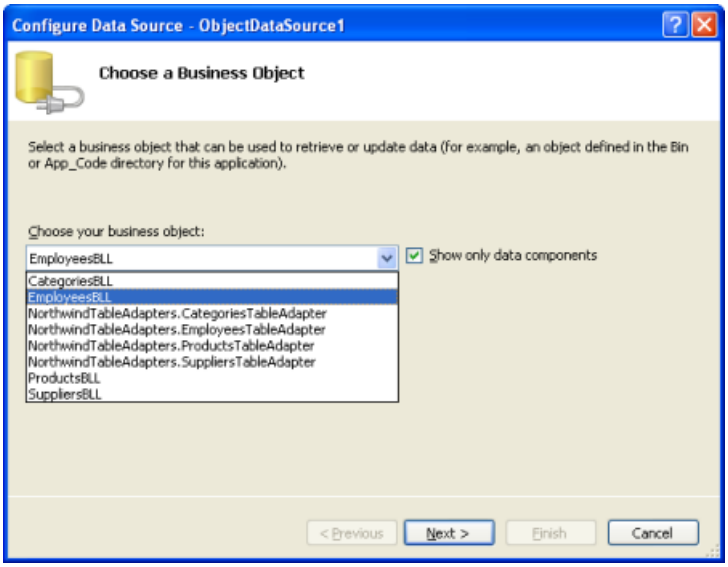


Figure 6: Use the `EmployeesBLL` Class (Click to view full-size image)

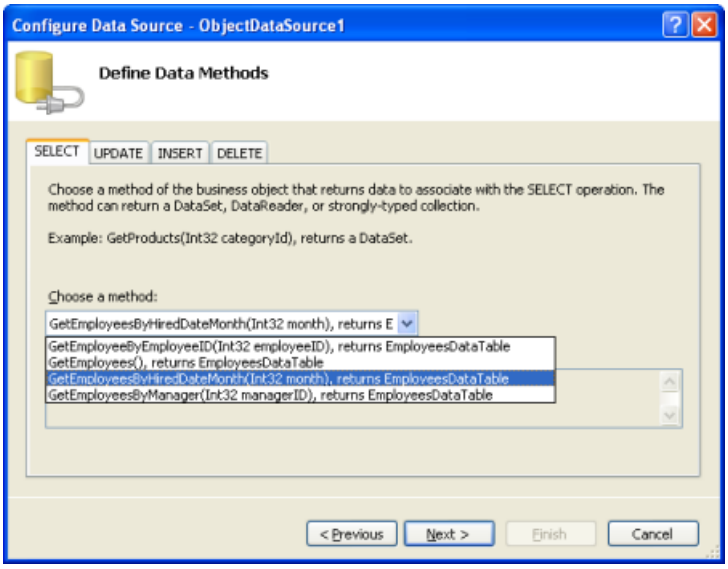


Figure 7: Select From the `GetEmployeesByHiredDateMonth(month)` method (Click to view full-size image)

The final screen asks us to provide the `month` parameter value's source. Since we'll set this value programmatically, leave the Parameter source set to the default None option and click Finish.

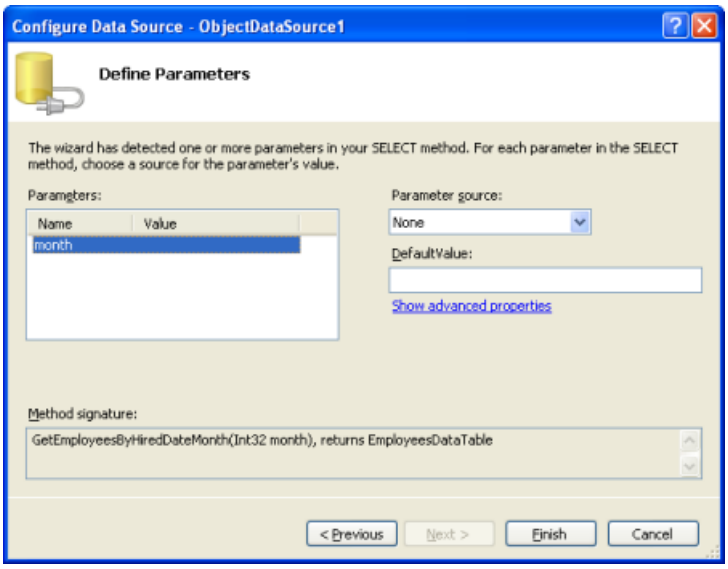


Figure 8: Leave the Parameter Source Set to None (Click to view full-size image)

This will create a `Parameter` object in the ObjectDataSource's `SelectParameters` collection that does not have a value specified.

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetEmployeesByHiredDateMonth" TypeName="EmployeesBLL">
    <SelectParameters>
        <asp:Parameter Name="month" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

To set this value programmatically, we need to create an event handler for the ObjectDataSource's `Selecting` event. To accomplish this, go to the Design view and double-click the ObjectDataSource. Alternatively, select the ObjectDataSource, go to the Properties window, and click the lightning bolt icon. Next, either double-click in the textbox next to the `Selecting` event or type in the name of the event handler you want to use.

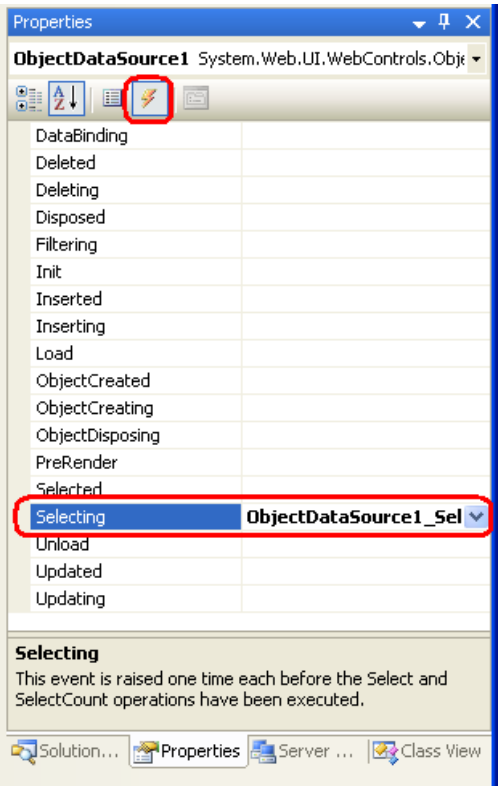


Figure 9: Click on the Lightning Bolt Icon in the Properties Window to List a Web Control's Events

Both approaches add a new event handler for the ObjectDataSource's `Selecting` event to the page's code-behind class. In this event handler we can read and write to the parameter values using `e.InputParameters[parameterName]`, where *`parameterName`* is the value of the `Name` attribute in the `<asp:Parameter>` tag (the `InputParameters` collection can also be indexed ordinally, as in `e.InputParameters[index]`). To set the `month` parameter to the current month, add the following to the `Selecting` event handler:

```
protected void ObjectDataSource1_Selecting
    (object sender, ObjectDataSourceSelectingEventArgs e)
{
    e.InputParameters["month"] = DateTime.Now.Month;
}
```

When visiting this page through a browser we can see that only one employee was hired this month (March) Laura Callahan, who's been with the company since 1994.

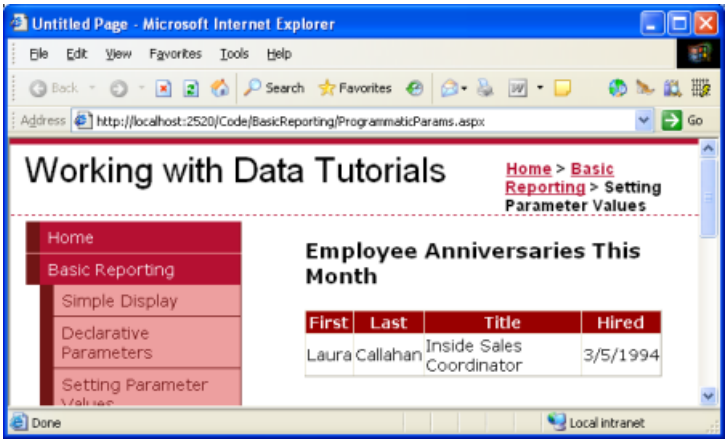


Figure 10: Those Employees Whose Anniversaries This Month Are Shown (Click to view full-size image)

Summary

While the ObjectDataSource's parameters' values can typically be set declaratively, without requiring a line of code, it's easy to set the parameter values programmatically. All we need to do is create an event handler for the ObjectDataSource's `Selecting` event, which fires before the underlying object's method is invoked, and manually set the values for one or more parameters via the `InputParameters` collection.

This tutorial concludes the Basic Reporting section. The next tutorial kicks off the Filtering and Master-Details Scenarios section, in which we'll look at techniques for allowing the visitor to filter data and drill down from a master report into a details report.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Displaying Data With the ObjectDataSource (VB)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> This tutorial looks at the ObjectDataSource control Using this control you can bind data retrieved from the BLL created in the previous tutorial without having to write a line of code!

Introduction

With our application architecture and website page layout complete, we're ready to start exploring how to accomplish a variety of common data- and reporting-related tasks. In the previous tutorials we've seen how to programmatically bind data from the DAL and BLL to a data Web control in an ASP.NET page. This syntax assigning the data Web control's `DataSource` property to the data to display and then calling the control's `DataBind()` method was the pattern used in ASP.NET 1.x applications, and can continue to be used in your 2.0 applications. However, ASP.NET 2.0's new data source controls offer a declarative way to work with data. Using these controls you can bind data retrieved from the BLL created in the previous tutorial without having to write a line of code!

ASP.NET 2.0 ships with five built-in data source controls SqlDataSource, AccessDataSource, ObjectDataSource, XmlDataSource, and SiteMapDataSource although you can build your own custom data source controls, if needed. Since we have developed an architecture for our tutorial application, we'll be using the ObjectDataSource against our BLL classes.

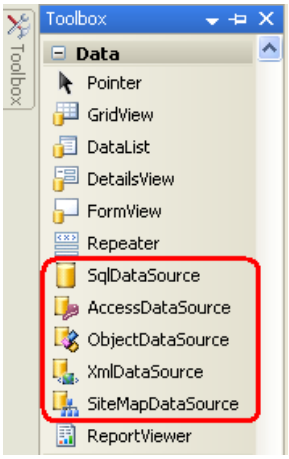


Figure 1: ASP.NET 2.0 Includes Five Built-In Data Source Controls

The ObjectDataSource serves as a proxy for working with some other object. To configure the ObjectDataSource we specify this underlying object and how its methods map to the ObjectDataSource's `Select`, `Insert`, `Update`, and `Delete` methods. Once this underlying object has been specified and its methods mapped to the ObjectDataSource's, we can then bind the ObjectDataSource to a data Web control. ASP.NET ships with many data Web controls, including the GridView, DetailsView, RadioButtonList, and DropDownList, among others. During the page lifecycle, the data Web control may need to access the data it's bound to, which it will accomplish by invoking its ObjectDataSource's `Select` method; if the data Web control supports inserting, updating, or deleting, calls may be made to its ObjectDataSource's `Insert`, `Update`, or `Delete` methods. These calls are then routed by the ObjectDataSource to the appropriate underlying object's methods as the following diagram illustrates.

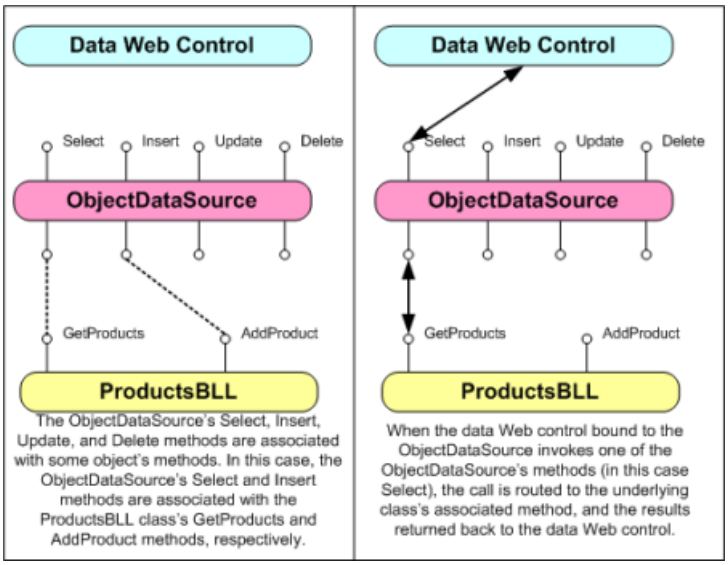


Figure 2: The ObjectDataSource Serves as a Proxy (Click to view full-size image)

While the ObjectDataSource can be used to invoke methods for inserting, updating, or deleting data, let's just focus on returning data; future tutorials will explore using the ObjectDataSource and data Web controls that modify data.

Step 1: Adding and Configuring the ObjectDataSource Control

Start by opening the `SimpleDisplay.aspx` page in the `BasicReporting` folder, switch to Design view, and then drag an ObjectDataSource control from the Toolbox onto the page's design surface. The ObjectDataSource appears as a gray box on the design surface because it does not produce any markup; it simply accesses data by invoking a method from a specified object. The data returned by an ObjectDataSource can be displayed by a data Web control, such as the GridView, DetailsView, FormView, and so on.

> **NOTE**
>
> Alternatively, you may first add the data Web control to the page and then, from its smart tag, choose the <New data source> option from the drop-down list.

To specify the ObjectDataSource's underlying object and how that object's methods map to the ObjectDataSource's, click on the Configure Data Source link from the ObjectDataSource's smart tag.

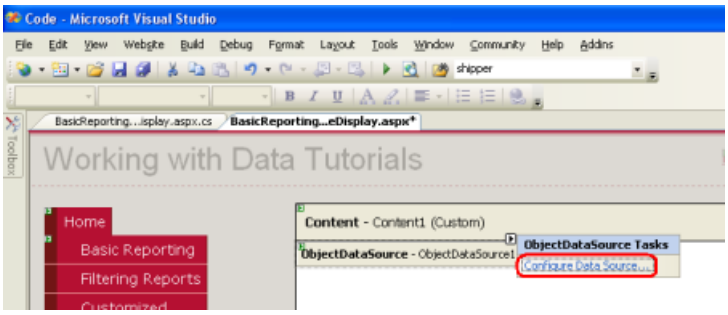


Figure 3: Click the Configure Data Source Link from the Smart Tag (Click to view full-size image)

This brings up the Configure Data Source wizard. First, we must specify the object the ObjectDataSource is to work with. If the "Show only data components" checkbox is checked, the drop-down list on this screen lists only those objects that have been decorated with the `DataObject` attribute. Currently our list includes the TableAdapters in the Typed DataSet and the BLL classes we created in the previous tutorial. If you forgot to add the `DataObject` attribute to the Business Logic Layer classes you won't see them in this list. In that case, uncheck the "Show only data components" checkbox to view all objects, which should include the BLL classes (along with

the other classes in the Typed DataSet the DataTables, DataRows, and so on).

From this first screen choose the `ProductsBLL` class from the drop-down list and click Next.

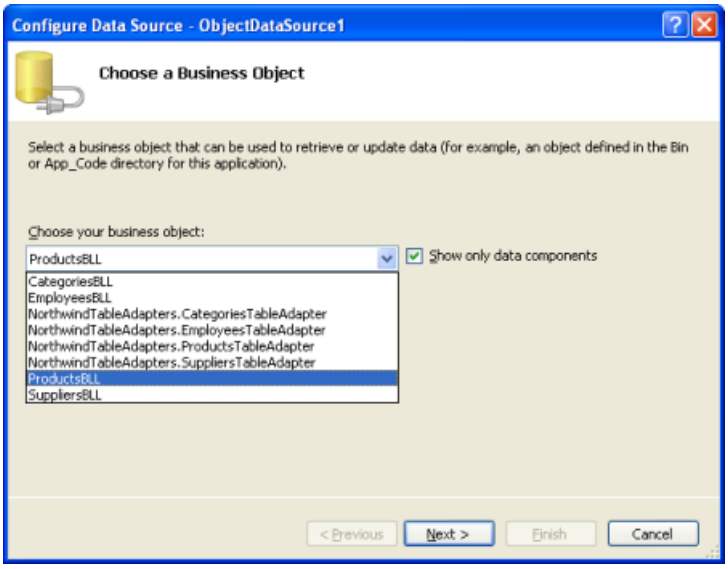


Figure 4: Specify the Object to Use with the ObjectDataSource Control (Click to view full-size image)

The next screen in the wizard prompts you to select what method the ObjectDataSource should invoke. The drop-down lists those methods that return data in the object selected from the previous screen. Here we see `GetProductByProductID`, `GetProducts`, `GetProductsByCategoryID`, and `GetProductsBySupplierID`. Select the `GetProducts` method from the drop-down list and click Finish (if you added the `DataObjectMethodAttribute` to the `ProductBLL`'s methods as shown in the previous tutorial, this option will be selected by default).

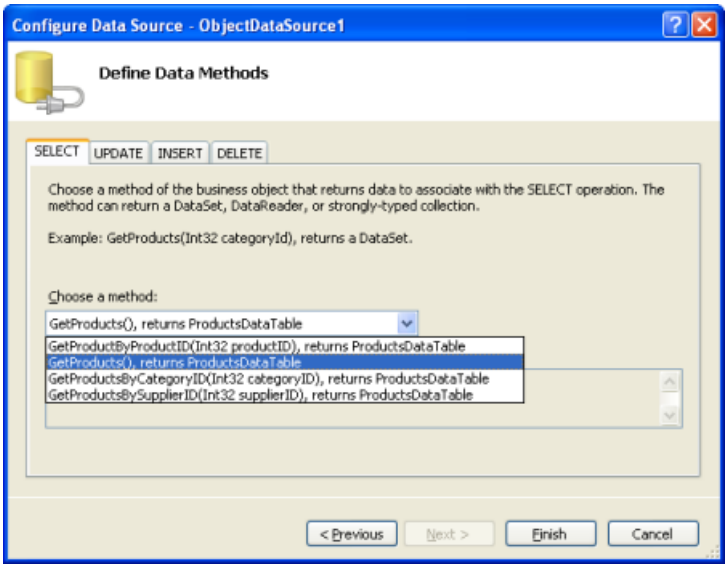


Figure 5: Choose the Method for Returning Data from the SELECT Tab (Click to view full-size image)

Configure the ObjectDataSource Manually

The ObjectDataSource's Configure Data Source wizard offers a quick way to specify the object it uses and to associate what methods of the object are invoked. You can, however, configure the ObjectDataSource through its properties, either through the Properties window or directly in the declarative markup. Simply set the `TypeName` property to the type of the underlying object to be used, and the `SelectMethod` to the method to invoke when retrieving data.

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
</asp:ObjectDataSource>
```

Even if you prefer the Configure Data Source wizard there may be times when you need to manually configure the ObjectDataSource, as the wizard only lists developer-created classes. If you want to bind the ObjectDataSource to a class in the .NET Framework such as the Membership class, to access user account information, or the Directory class to work with file system information you'll need to manually set the ObjectDataSource's properties.

Step 2: Adding a Data Web Control and Binding it to the ObjectDataSource

Once the ObjectDataSource has been added to the page and configured, we're ready to add data Web controls to the page to display the data returned by the ObjectDataSource's `Select` method. Any data Web control can be bound to an ObjectDataSource; let's look at displaying the ObjectDataSource's data in a GridView, DetailsView, and FormView.

Binding a GridView to the ObjectDataSource

Add a GridView control from the Toolbox to `SimpleDisplay.aspx`'s design surface. From the GridView's smart tag, choose the ObjectDataSource control we added in Step 1. This will automatically create a BoundField in the GridView for each property returned by the data from the ObjectDataSource's `Select` method (namely, the properties defined by the Products DataTable).

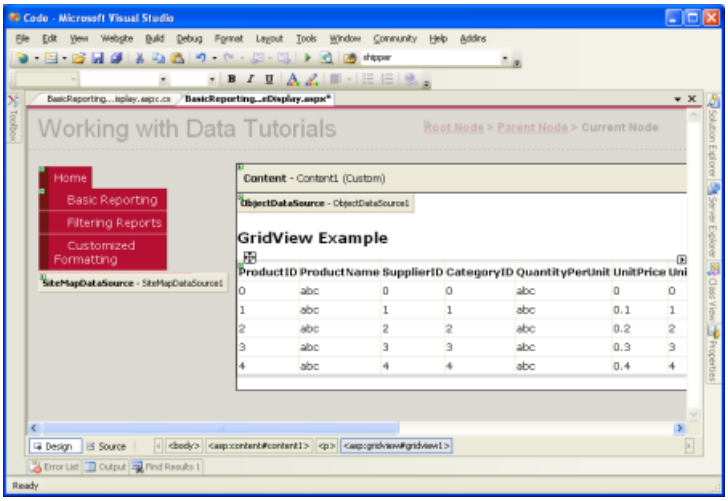


Figure 6: A GridView Has Been Added to the Page and Bound to the ObjectDataSource (Click to view full-size image)

You can then customize, rearrange, or remove the GridView's BoundFields by clicking the Edit Columns option from the smart tag.

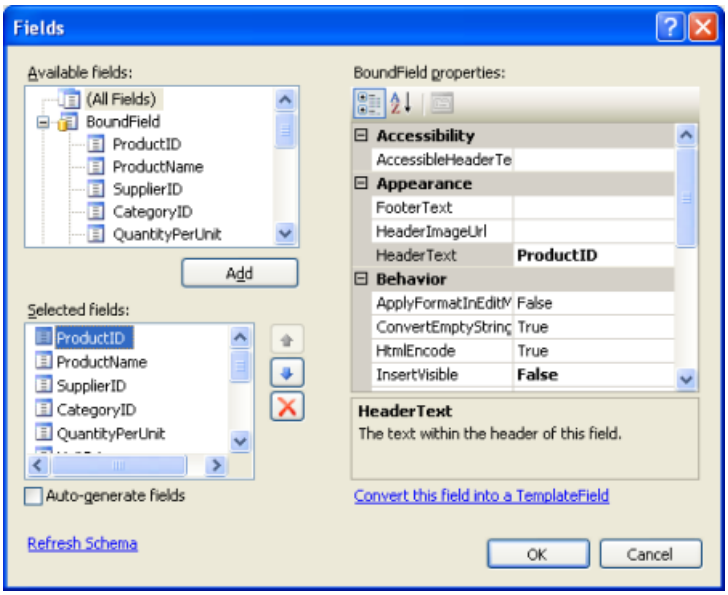


Figure 7: Manage the GridView's BoundFields Through the Edit Columns Dialog Box (Click to view full-size image)

Take a moment to modify the GridView's BoundFields, removing the `ProductID`, `SupplierID`, `CategoryID`, `QuantityPerUnit`, `UnitsInStock`, `UnitsOnOrder`, and `ReorderLevel` BoundFields. Simply select the BoundField from the list in the bottom left and click the delete button (the red X) to remove them. Next, Rearrange the BoundFields so that the `CategoryName` and `SupplierName` BoundFields precede the `UnitPrice` BoundField by selecting these BoundFields and clicking the up arrow. Set the `HeaderText` properties of the remaining BoundFields to `Products`, `Category`, `Supplier`, and `Price`, respectively. Next, have the `Price` BoundField formatted as a currency by setting the BoundField's `HtmlEncode` property to False and its `DataFormatString` property to `{0:c}`. Finally, horizontally align the `Price` to the right and the `Discontinued` checkbox in the center via the `ItemStyle` / `HorizontalAlign` property.

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductName"
          HeaderText="Product" SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName"
          HeaderText="Category" ReadOnly="True"
          SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName"
          HeaderText="Supplier" ReadOnly="True"
          SortExpression="SupplierName" />
        <asp:BoundField DataField="UnitPrice"
          DataFormatString="{0:c}" HeaderText="Price"
            HtmlEncode="False" SortExpression="UnitPrice">
            <ItemStyle HorizontalAlign="Right" />
        </asp:BoundField>
        <asp:CheckBoxField DataField="Discontinued"
          HeaderText="Discontinued" SortExpression="Discontinued">
            <ItemStyle HorizontalAlign="Center" />
        </asp:CheckBoxField>
    </Columns>
</asp:GridView>
```

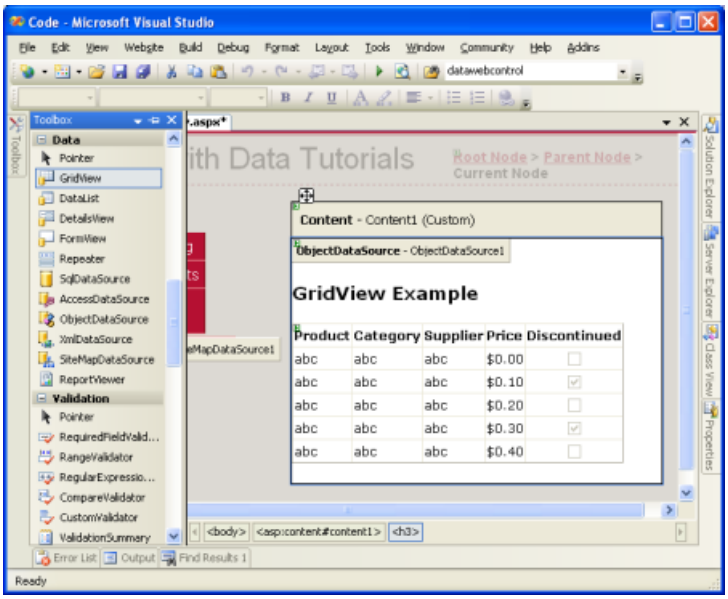


Figure 8: The GridView's BoundFields Have Been Customized (Click to view full-size image)

Using Themes for a Consistent Look

These tutorials strive to remove any control-level style settings, instead using cascading style sheets defined in an external file whenever possible. The `Styles.css` file contains `DataWebControlStyle`, `HeaderStyle`, `RowStyle`, and `AlternatingRowStyle` CSS classes that should be used to dictate the appearance of the data Web controls used in these tutorials. To accomplish this, we could set the GridView's `CssClass` property to `DataWebControlStyle`, and its `HeaderStyle`, `RowStyle`, and `AlternatingRowStyle` properties' `CssClass` properties accordingly.

If we set these `CssClass` properties at the Web control we'd need to remember to explicitly set these property values for each and every data Web control added to our tutorials. A more manageable approach is to define the default CSS-related properties for the GridView, DetailsView, and FormView controls using a Theme. A Theme is a collection of control-level property settings, images, and CSS classes that can be applied to pages across a site to enforce a common look and feel.

Our Theme won't include any images or CSS files (we'll leave the stylesheet `Styles.css` as-is, defined in the root folder of the web application), but will include two Skins. A Skin is a file that defines the default properties for a Web control. Specifically, we'll have a Skin file for the GridView and DetailsView controls, indicating the default `CssClass`-related properties.

Start by adding a new Skin File to your project named `GridView.skin` by right-clicking on the project name in the Solution Explorer and choosing Add New Item.

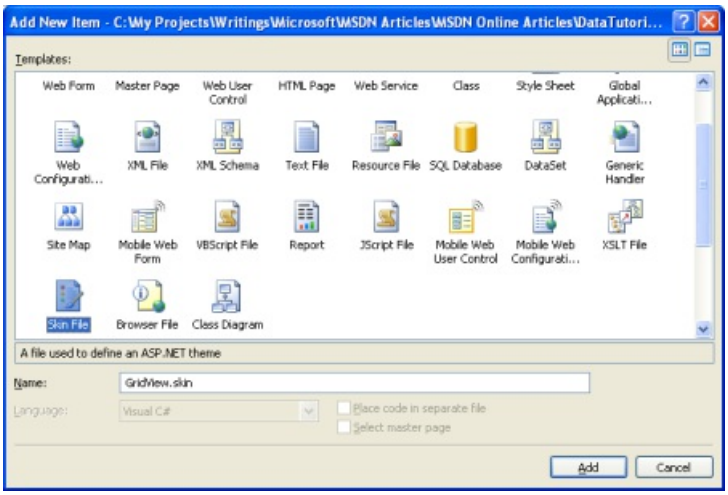


Figure 9: Add a Skin File Named `GridView.skin` (Click to view full-size image)

Skin files need to be placed in a Theme, which are located in the `App_Themes` folder. Since we don't yet have such a folder, Visual Studio will kindly offer to create one for us when adding our first Skin. Click Yes to create the `App_Theme` folder and place the new `GridView.skin` file there.

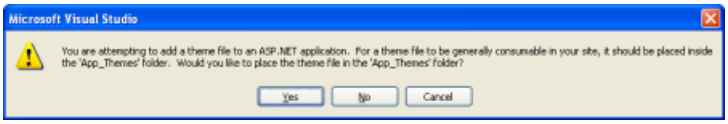


Figure 10: Let Visual Studio Create the `App_Theme` Folder (Click to view full-size image)

This will create a new Theme in the `App_Themes` folder named GridView with the Skin file `GridView.skin`.

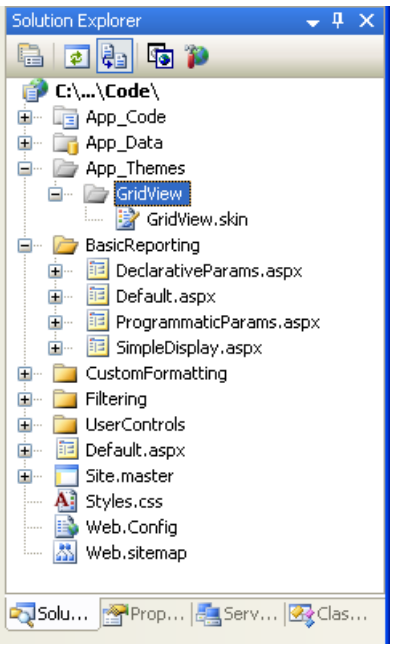


Figure 11: The GridView Theme has Been Added to the `App_Theme` Folder

Rename the GridView Theme to DataWebControls (right-click on the GridView folder in the `App_Theme` folder and choose Rename). Next, enter the following markup into the `GridView.skin` file:

```
<asp:GridView runat="server" CssClass="DataWebControlStyle">
   <AlternatingRowStyle CssClass="AlternatingRowStyle" />
   <RowStyle CssClass="RowStyle" />
   <HeaderStyle CssClass="HeaderStyle" />
</asp:GridView>
```

This defines the default properties for the `CssClass`-related properties for any GridView in any page that uses the DataWebControls Theme. Let's add another Skin for the DetailsView, a data Web control that we'll be using shortly. Add a new Skin to the DataWebControls Theme named `DetailsView.skin` and add the following markup:

```
<asp:DetailsView runat="server" CssClass="DataWebControlStyle">
   <AlternatingRowStyle CssClass="AlternatingRowStyle" />
   <RowStyle CssClass="RowStyle" />
   <FieldHeaderStyle CssClass="HeaderStyle" />
</asp:DetailsView>
```

With our Theme defined, the last step is to apply the Theme to our ASP.NET page. A Theme can be applied on a page-by-page basis or for all pages in a website. Let's use this theme for all pages in the website. To accomplish this, add the following markup to `Web.config`'s `<system.web>` section:

```
<pages styleSheetTheme="DataWebControls" />
```

That's all there is to it! The `styleSheetTheme` setting indicates that the properties specified in the Theme should *not* override the properties specified at the control level. To specify that Theme settings should trump control settings, use the `theme` attribute in place of `styleSheetTheme`; unfortunately, Theme settings do not appear in the Visual Studio Design view. Refer to ASP.NET Themes and Skins Overview and Server-Side Styles Using Themes for more information on Themes and Skins; see How To: Apply ASP.NET Themes for more on configuring a page to use a theme.

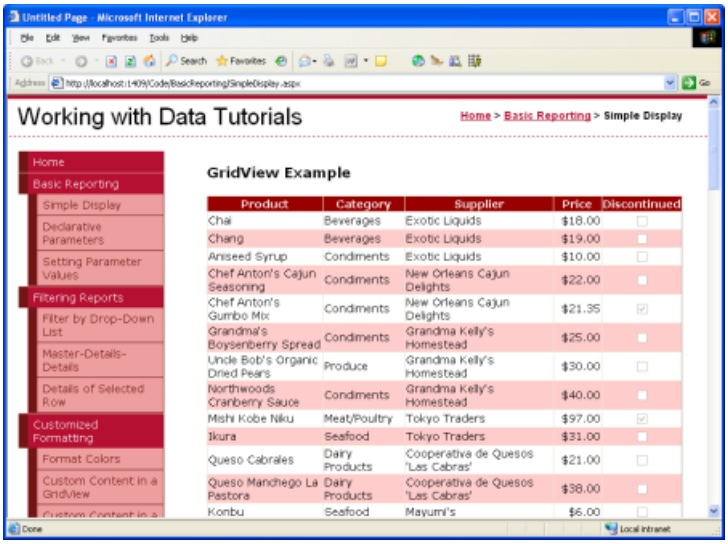


Figure 12: The GridView Displays the Product's Name, Category, Supplier, Price, and Discontinued Information (Click to view full-size image)

Displaying One Record at a Time in the DetailsView

The GridView displays one row for each record returned by the data source control to which it's bound. There are times, however, when we may want to display a sole record or just one record at a time. The DetailsView control offers this functionality, rendering as an HTML `<table>` with two columns and one row for each column or property bound to the control. You can think of the DetailsView as a GridView with a single record rotated 90 degrees.

Start by adding a DetailsView control *above* the GridView in `SimpleDisplay.aspx`. Next, bind it to the same ObjectDataSource control as the GridView. Like with the GridView, a BoundField will be added to the DetailsView for each property in the object returned by the ObjectDataSource's `Select` method. The only difference is that the DetailsView's BoundFields are laid out horizontally rather than vertically.

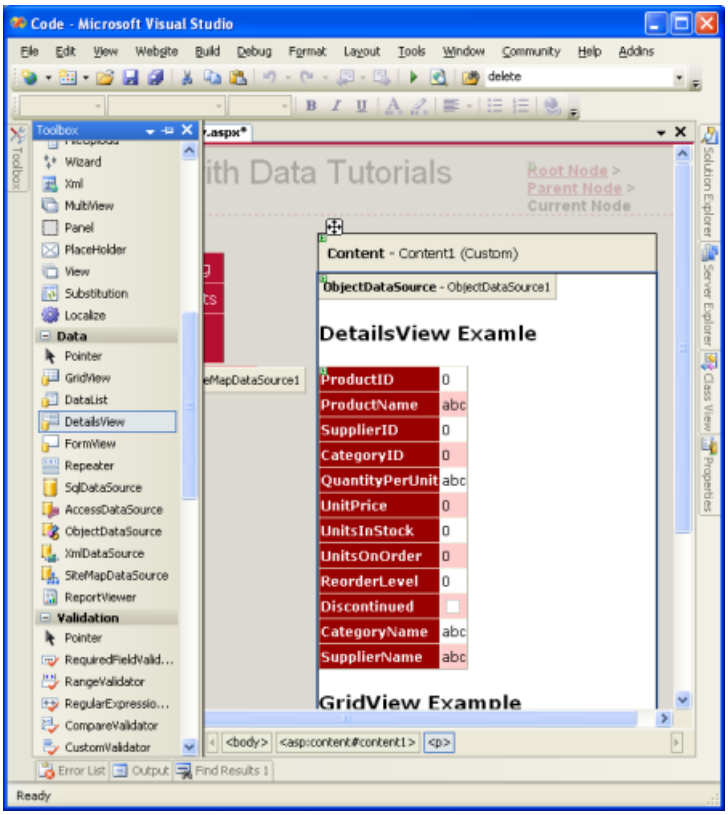


Figure 13: Add a DetailsView to the Page and Bind it to the ObjectDataSource (Click to view full-size image)

Like the GridView, the DetailsView's BoundFields can be tweaked to provide a more customized display of the data returned by the ObjectDataSource. Figure 14 shows the DetailsView after its BoundFields and `CssClass` properties have been configured to make its appearance similar to the GridView example.

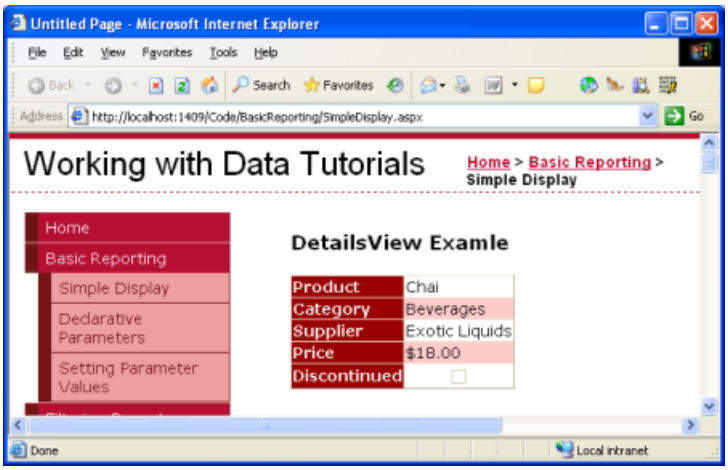


Figure 14: The DetailsView Shows a Single Record (Click to view full-size image)

Note that the DetailsView only displays the first record returned by its data source. To allow the user to step through all of the records, one at a time, we must enable paging for the DetailsView. To do so, return to Visual Studio and check the Enable Paging checkbox in the DetailsView's smart tag.

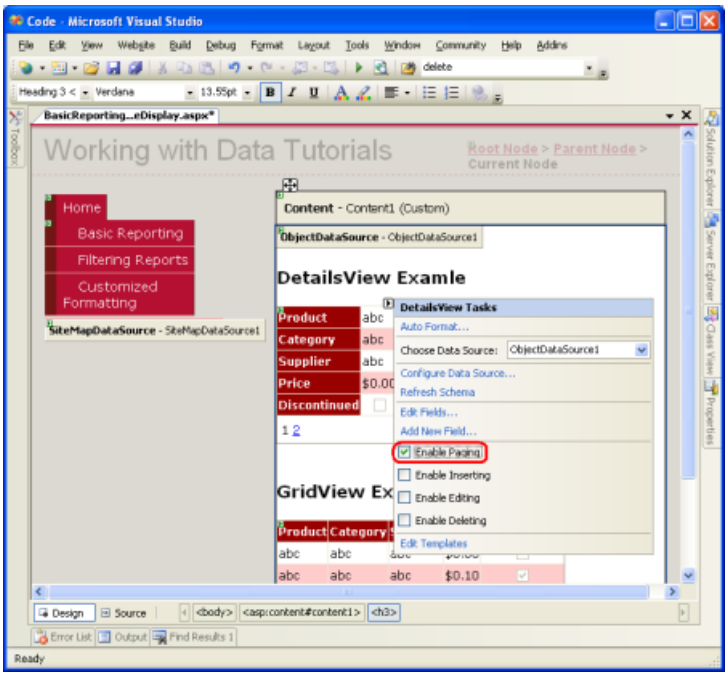


Figure 15: Enable Paging in the DetailsView Control (Click to view full-size image)

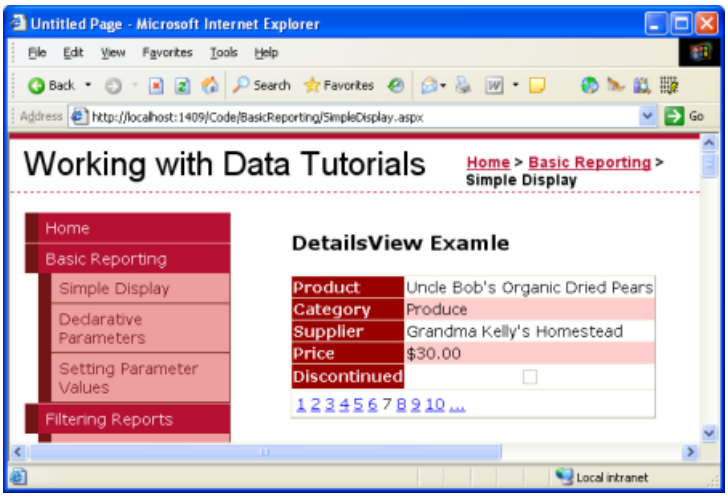


Figure 16: With Paging Enabled, the DetailsView Allows the User to View Any of the Products (Click to view full-size image)

We'll talk more about paging in future tutorials.

A More Flexible Layout for Showing One Record at a Time

The DetailsView is pretty rigid in how it displays each record returned from the ObjectDataSource. We may want a more flexible view of the data. For example, rather than showing the product's name, category, supplier, price, and discontinued information each on a separate row, we may want to show the product name and price in an `<h4>` heading, with the category and supplier information appearing below the name and price in a smaller font size. And we may not care to show the property names (Product, Category, and so on) next to the values.

The FormView control provides this level of customization. Rather than using fields (like the GridView and DetailsView do), the FormView uses templates, which allow for a mix of Web controls, static HTML, and databinding syntax. If you are familiar with the Repeater control from ASP.NET 1.x, you can think of the FormView as the Repeater for showing a single record.

Add a FormView control to the `SimpleDisplay.aspx` page's design surface. Initially the FormView displays as a gray block, informing us that we need to provide, at minimum, the control's `ItemTemplate`.

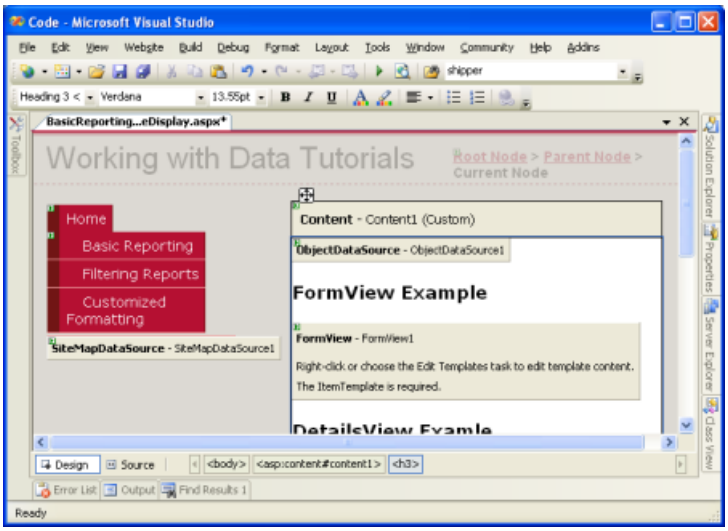


Figure 17: The FormView Must Include an `ItemTemplate` (Click to view full-size image)

You can bind the FormView directly to a data source control through the FormView's smart tag, which will create a default `ItemTemplate` automatically (along with an `EditItemTemplate` and `InsertItemTemplate`, if the ObjectDataSource control's `InsertMethod` and `UpdateMethod` properties are set). However, for this example let's bind the data to the FormView and specify its `ItemTemplate` manually. Start by setting the FormView's `DataSourceID` property to the `ID` of the ObjectDataSource control, `ObjectDataSource1`. Next, create the `ItemTemplate` so that it displays the product's name and price in an `<h4>` element and the category and shipper names beneath that in a smaller font size.

```
<asp:FormView ID="FormView1" runat="server"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <ItemTemplate>
        <h4><%# Eval("ProductName") %>
          (<%# Eval("UnitPrice", "{0:c}") %>)</h4>
        Category: <%# Eval("CategoryName") %>;
        Supplier: <%# Eval("SupplierName") %>
    </ItemTemplate>
</asp:FormView>
```

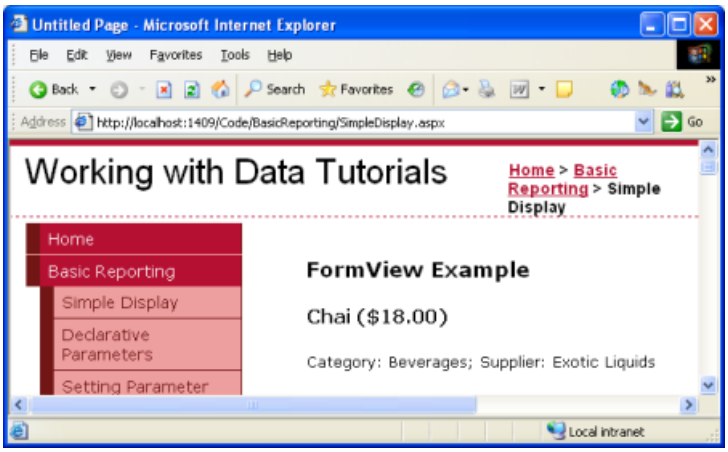


Figure 18: The First Product (Chai) is Displayed in a Custom Format (Click to view full-size image)

The `<%# Eval(propertyName) %>` is the databinding syntax. The `Eval` method returns the value of the specified property for the current object being bound to the FormView control. Check out Alex Homer's article Simplified and Extended Data Binding Syntax in ASP.NET 2.0 for more information on the ins and outs of databinding.

Like the DetailsView, the FormView only shows the first record returned from the ObjectDataSource. You can enable paging in the FormView to allow visitors to step through the products one at a time.

Summary

Accessing and displaying data from a Business Logic Layer can be accomplished without writing a line of code thanks to ASP.NET 2.0's ObjectDataSource control. The ObjectDataSource invokes a specified method of a class and returns the results. These results can be displayed in a data Web control that's bound to the ObjectDataSource. In this tutorial we looked at binding the GridView, DetailsView, and FormView controls to the ObjectDataSource.

So far we've only seen how to use the ObjectDataSource to invoke a parameter-less method, but what if we want to invoke a method that expects input parameters, such as the `ProductBLL` class's `GetProductsByCategoryID(categoryID)`? In order to call a method that expects one or more parameters we must configure the ObjectDataSource to specify the values for these parameters. We'll see how to accomplish this in our next tutorial.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Create Your Own Data Source Controls
- GridView Examples for ASP.NET 2.0
- Simplified and Extended Data Binding Syntax in ASP.NET 2.0
- Themes in ASP.NET 2.0
- Server-Side Styles Using Themes
- How To: Apply ASP.NET Themes Programmatically

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Declarative Parameters (VB)

2/20/2020 • 8 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll illustrate how to use a parameter set to a hard-coded value to select the data to display in a DetailsView control.

Introduction

In the last tutorial we looked at displaying data with the GridView, DetailsView, and FormView controls bound to an ObjectDataSource control that invoked the `GetProducts()` method from the `ProductsBLL` class. The `GetProducts()` method returns a strongly-typed DataTable populated with all of the records from the Northwind database's `Products` table. The `ProductsBLL` class contains additional methods for returning just subsets of the products - `GetProductByProductID(productID)`, `GetProductsByCategoryID(categoryID)`, and `GetProductsBySupplierID(supplierID)`. These three methods expect an input parameter indicating how to filter the returned product information.

The ObjectDataSource can be used to invoke methods that expect input parameters, but in order to do so we must specify where the values for these parameters come from. The parameter values can be hard-coded or can come from a variety of dynamic sources, including: querystring values, Session variables, the property value of a Web control on the page, or others.

For this tutorial let's start by illustrating how to use a parameter set to a hard-coded value. Specifically, we'll look at adding a DetailsView to the page that displays information about a specific product, namely Chef Anton's Gumbo Mix, which has a `ProductID` of 5. Next, we'll see how to set the parameter value based on a Web control. In particular, we'll use a TextBox to let the user type in a country, after which they can click a Button to see the list of suppliers that reside in that country.

Using a Hard-Coded Parameter Value

For the first example, start by adding a DetailsView control to the `DeclarativeParams.aspx` page in the `BasicReporting` folder. From the DetailsView's smart tag, select <New data source> from the drop-down list and choose to add an ObjectDataSource.

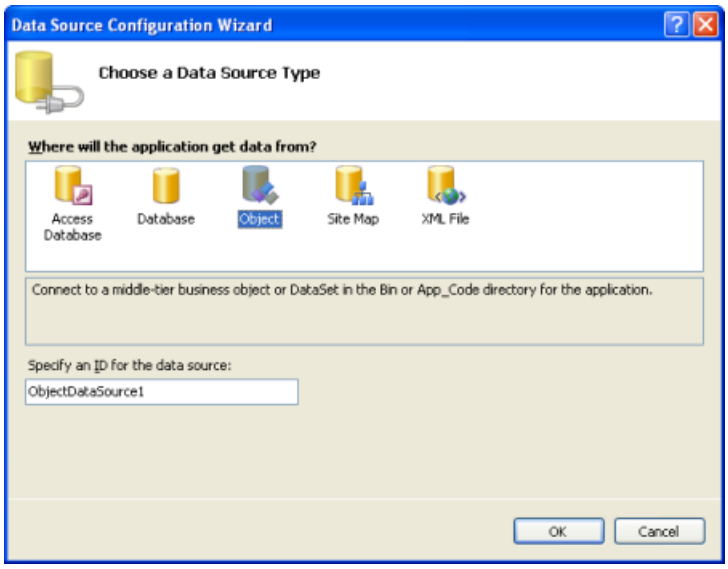


Figure 1: Add an ObjectDataSource to the Page (Click to view full-size image)

This will automatically start the ObjectDataSource control's Choose Data Source wizard. Select the `ProductsBLL` class from the first screen of the wizard.

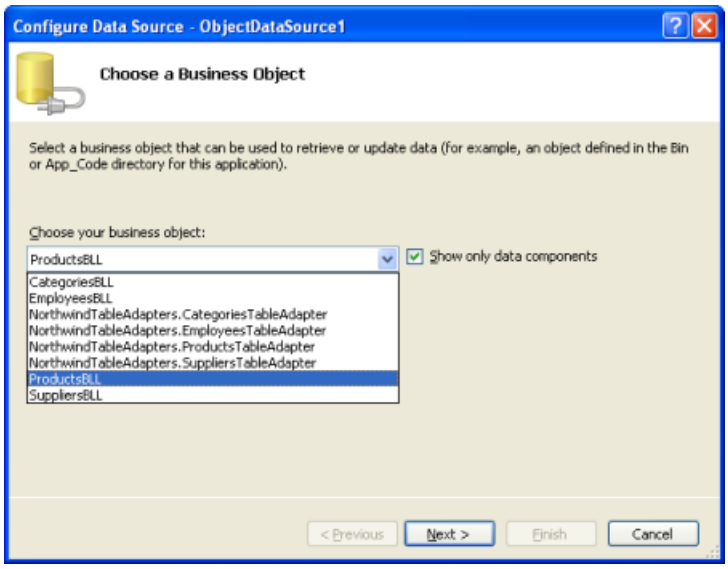


Figure 2: Select the `ProductsBLL` Class (Click to view full-size image)

Since we want to display information about a particular product we want to use the `GetProductByProductID(productID)` method.

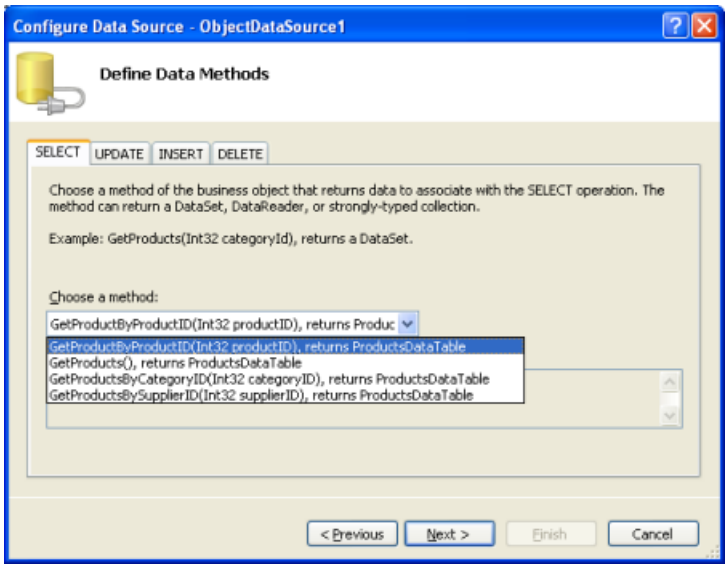


Figure 3: Choose the `GetProductByProductID(productID)` method (Click to view full-size image)

Since the method we selected includes a parameter, there's one more screen for the wizard, where we're asked to define the value to be used for the parameter. The list on the left shows all of the parameters for the selected method. For `GetProductByProductID(productID)` there's only one `productID`. On the right we can specify the value for the selected parameter. The parameter source drop-down list enumerates the various possible sources for the parameter value. Since we want to specify a hard-coded value of 5 for the `productID` parameter, leave the Parameter source as None and enter 5 into the DefaultValue textbox.

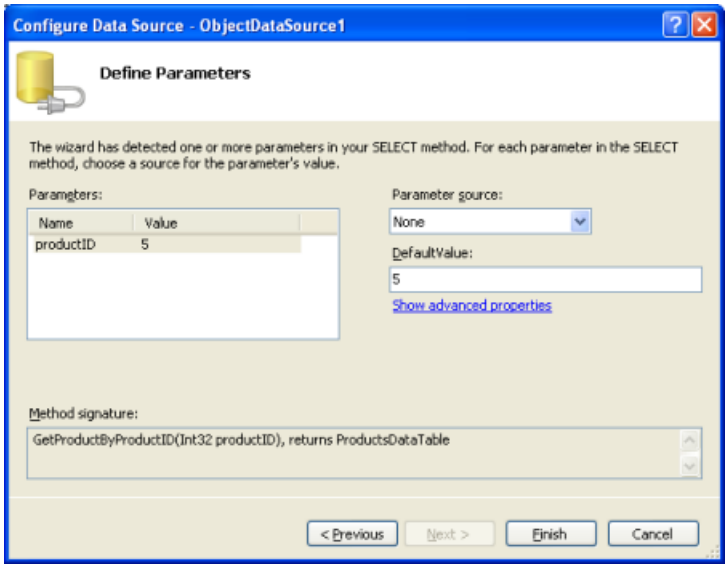


Figure 4: A Hard-Coded Parameter Value of 5 Will Be Used for the `productID` Parameter (Click to view full-size image)

After completing the Configure Data Source wizard, the ObjectDataSource control's declarative markup includes a `Parameter` object in the `SelectParameters` collection for each of the input parameters expected by the method defined in the `SelectMethod` property. Since the method we're using in this example expects just a single input parameter, `parameterID`, there's only one entry here. The `SelectParameters` collection can contain any class that derives from the `Parameter` class in the `System.Web.UI.WebControls` namespace. For hard-coded parameter values the base `Parameter` class is used, but for the other parameter source options a derived `Parameter` class is used; you can also create your own custom parameter types, if needed.

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    SelectMethod="GetProductByProductID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:Parameter DefaultValue="5"
           Name="productID" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

NOTE

If you're following along on your own computer the declarative markup you see at this point may include values for the `InsertMethod`, `UpdateMethod`, and `DeleteMethod` properties, as well as `DeleteParameters`. The ObjectDataSource's Choose Data Source wizard automatically specifies the methods from the `ProductBLL` to use for inserting, updating, and deleting, so unless you explicitly cleared those out, they'll be included in the markup above.

When visiting this page, the data Web control will invoke the ObjectDataSource's `Select` method, which will call the `ProductsBLL` class's `GetProductByProductID(productID)` method using the hard-coded value of 5 for the `productID` input parameter. The method will return a strongly-typed `ProductDataTable` object that contains a

single row with information about Chef Anton's Gumbo Mix (the product with `ProductID` 5).

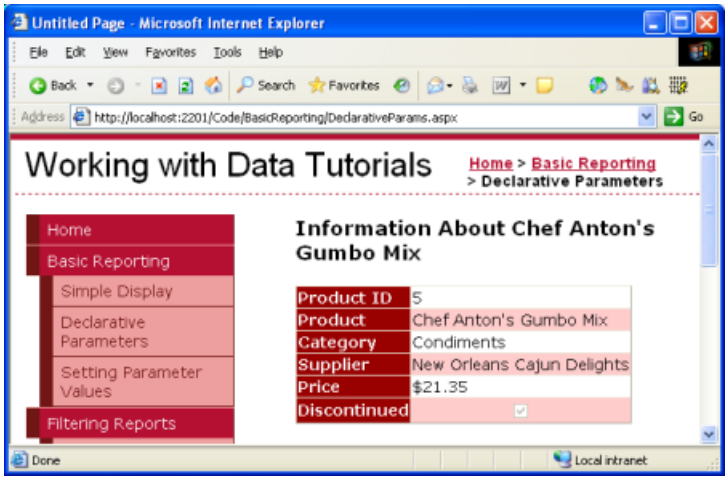


Figure 5: Information About Chef Anton's Gumbo Mix are Displayed (Click to view full-size image)

Setting the Parameter Value to the Property Value of a Web Control

The ObjectDataSource's parameter values can also be set based on the value of a Web control on the page. To illustrate this, let's have a GridView that lists all of the suppliers that are located in a country specified by the user. To accomplish this start by adding a TextBox to the page into which the user can enter a country name. Set this TextBox control's `ID` property to `CountryName`. Also add a Button Web control.

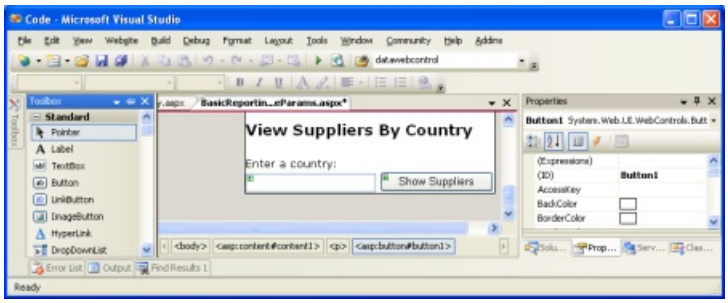


Figure 6: Add a TextBox to the Page with `ID` `CountryName` (Click to view full-size image)

Next, add a GridView to the page and, from the smart tag, choose to add a new ObjectDataSource. Since we want to display supplier information select the `SuppliersBLL` class from the wizard's first screen. From the second screen, pick the `GetSuppliersByCountry(country)` method.

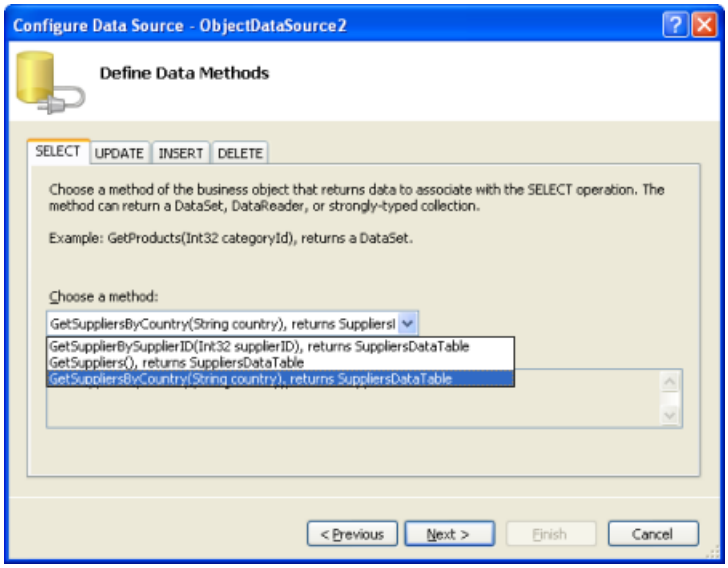


Figure 7: Choose the `GetSuppliersByCountry(country)` Method (Click to view full-size image)

Since the `GetSuppliersByCountry(country)` method has an input parameter, the wizard once again includes a final screen for choosing the parameter value. This time, set the Parameter source to Control. This will populate the ControlID drop-down list with the names of the controls on the page; select the `CountryName` control from the list. When the page is first visited the `CountryName` TextBox will be blank, so no results are returned and nothing is displayed. If you want to display some results by default, set the DefaultValue textbox accordingly.

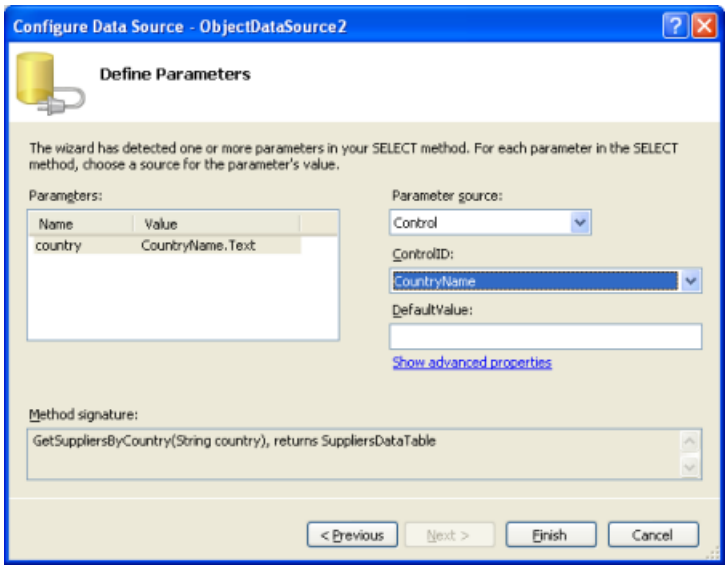


Figure 8: Set the Parameter Value to the `CountryName` Control Value (Click to view full-size image)

The ObjectDataSource's declarative markup differs slightly from our first example, using a ControlParameter instead of the standard `Parameter` object. A `ControlParameter` has additional properties to specify the `ID` of the Web control and the property value to use for the parameter (`PropertyName`). The Configure Data Source wizard was smart enough to determine that, for a TextBox, we'll likely want to use the `Text` property for the parameter value. If, however, you want to use a different property value from the Web control you can change the `PropertyName` value here or by clicking the "Show advanced properties" link in the wizard.

```
<asp:ObjectDataSource ID="ObjectDataSource2" runat="server"
    SelectMethod="GetSuppliersByCountry" TypeName="SuppliersBLL">
    <SelectParameters>
        <asp:ControlParameter ControlID="CountryName"
            Name="country" PropertyName="Text"
            Type="String" />
    </SelectParameters>
</asp:ObjectDataSource>
```

When visiting the page for the first time the `CountryName` TextBox is empty. The ObjectDataSource's `Select` method is still invoked by the GridView, but a value of `Nothing` is passed into the `GetSuppliersByCountry(country)` method. The TableAdapter converts the `Nothing` into a database `NULL` value (`DBNull.Value`), but the query used by the `GetSuppliersByCountry(country)` method is written such that it doesn't return any values when a `NULL` value is specified for the `@CategoryID` parameter. In short, no suppliers are returned.

Once the visitor enters in a country, however, and clicks the Show Suppliers button to cause a postback, the ObjectDataSource's `Select` method is requeried, passing in the TextBox control's `Text` value as the `country` parameter.

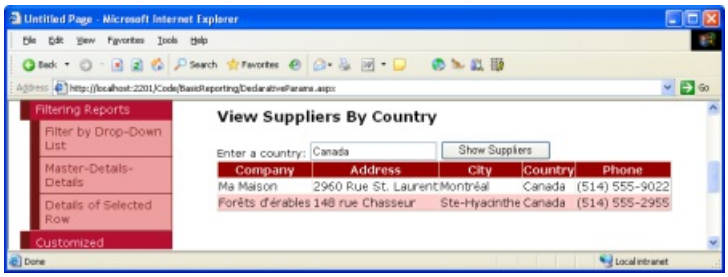


Figure 9: Those Suppliers from Canada are Shown (Click to view full-size image)

Showing All Suppliers By Default

Rather than show none of the suppliers when first viewing the page we may want to show *all* suppliers at first, allowing the user to pare down the list by entering a country name in the TextBox. When the TextBox is empty, the `SuppliersBLL` class's `GetSuppliersByCountry(country)` method is passed in `Nothing` for its *`country`* input parameter. This `Nothing` value is then passed down into the DAL's `GetSupplierByCountry(country)` method, where it's translated to a database `NULL` value for the `@Country` parameter in the following query:

```
SELECT     SupplierID, CompanyName, Address, City, Country, Phone
FROM         Suppliers
WHERE Country = @Country
```

The expression `Country = NULL` always returns False, even for records whose `Country` column has a `NULL` value; therefore, no records are returned.

To return *all* suppliers when the country TextBox is empty, we can augment the `GetSuppliersByCountry(country)` method in the BLL to invoke the `GetSuppliers()` method when its country parameter is `Nothing` and to call the DAL's `GetSuppliersByCountry(country)` method otherwise. This will have the effect of returning all suppliers when no country is specified and the appropriate subset of suppliers when the country parameter is included.

Change the `GetSuppliersByCountry(country)` method in the `SuppliersBLL` class to the following:

```
Public Function GetSuppliersByCountry(country As String) _
    As Northwind.SuppliersDataTable
    If String.IsNullOrEmpty(country) Then
        Return GetSuppliers()
    Else
        Return Adapter.GetSuppliersByCountry(country)
    End If
End Function
```

With this change the `DeclarativeParams.aspx` page shows all of the suppliers when first visited (or whenever the `CountryName` TextBox is empty).

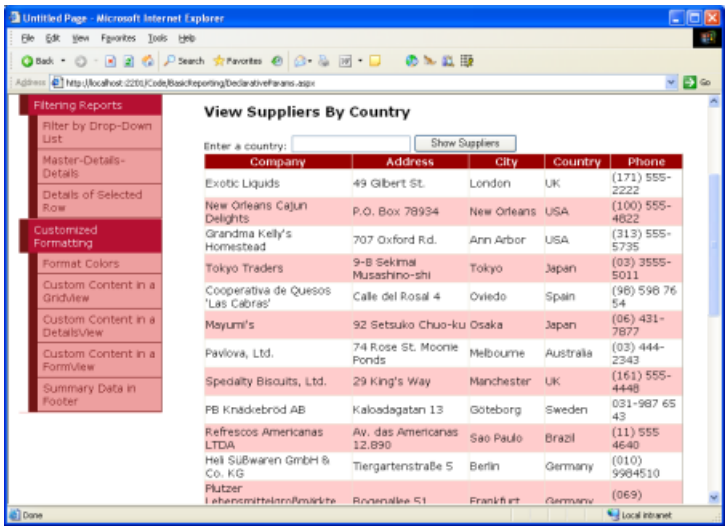


Figure 10: All Suppliers are Now Shown by Default (Click to view full-size image)

Summary

In order to use methods with input parameters, we need to specify the values for the parameters in the ObjectDataSource's `SelectParameters` collection. Different types of parameters allow for the parameter value to be obtained from different sources. The default parameter type uses a hard-coded value, but just as easily (and without a line of code) parameter values can be obtained from the querystring, Session variables, cookies, and even user-entered values from Web controls on the page.

The examples we looked at in this tutorial illustrated how to use declarative parameter values. However, there may be times when we need to use a parameter source that's not available, such as the current date and time, or, if our site was using Membership, the User ID of the visitor. For such scenarios we can set the parameter values programmatically prior to the ObjectDataSource invoking its underlying object's method. We'll see how to accomplish this in the next tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Programmatically Setting the ObjectDataSource's Parameter Values (VB)

2/20/2020 • 6 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll look at adding a method to our DAL and BLL that accepts a single input parameter and returns data. The example will set this parameter programmatically.

Introduction

As we saw in the previous tutorial, a number of options are available for declaratively passing parameter values to the ObjectDataSource's methods. If the parameter value is hard-coded, comes from a Web control on the page, or is in any other source that is readable by a data source `Parameter` object, for example, that value can be bound to the input parameter without writing a line of code.

There may be times, however, when the parameter value comes from some source not already accounted for by one of the built-in data source `Parameter` objects. If our site supported user accounts we might want to set the parameter based on the currently logged in visitor's User ID. Or we may need to customize the parameter value before sending it along to the ObjectDataSource's underlying object's method.

Whenever the ObjectDataSource's `Select` method is invoked the ObjectDataSource first raises its Selecting event. The ObjectDataSource's underlying object's method is then invoked. Once that completes the ObjectDataSource's Selected event fires (Figure 1 illustrates this sequence of events). The parameter values passed into the ObjectDataSource's underlying object's method can be set or customized in an event handler for the `Selecting` event.

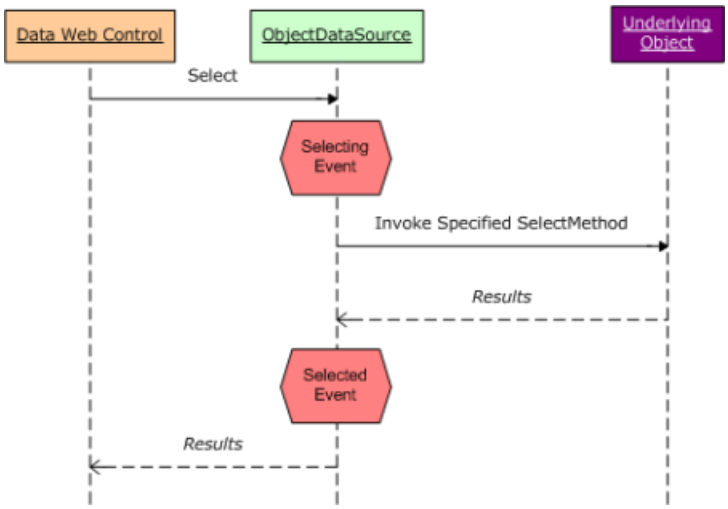


Figure 1: The ObjectDataSource's `Selected` and `Selecting` Events Fire Before and After Its Underlying Object's Method is Invoked (Click to view full-size image)

In this tutorial we'll look at adding a method to our DAL and BLL that accepts a single input parameter `Month`, of type `Integer` and returns an `EmployeesDataTable` object populated with those employees that have their hiring anniversary in the specified `Month`. Our example will set this parameter programmatically based on the current month, showing a list of "Employee Anniversaries This Month."

Let's get started!

Step 1: Adding a Method to `EmployeesTableAdapter`

For our first example we need to add a means to retrieve those employees whose `HireDate` occurred in a specified month. To provide this functionality in accordance with our architecture we need to first create a method in `EmployeesTableAdapter` that maps to the proper SQL statement. To accomplish this, start by opening the Northwind Typed DataSet. Right-click on the `EmployeesTableAdapter` label and choose Add Query.

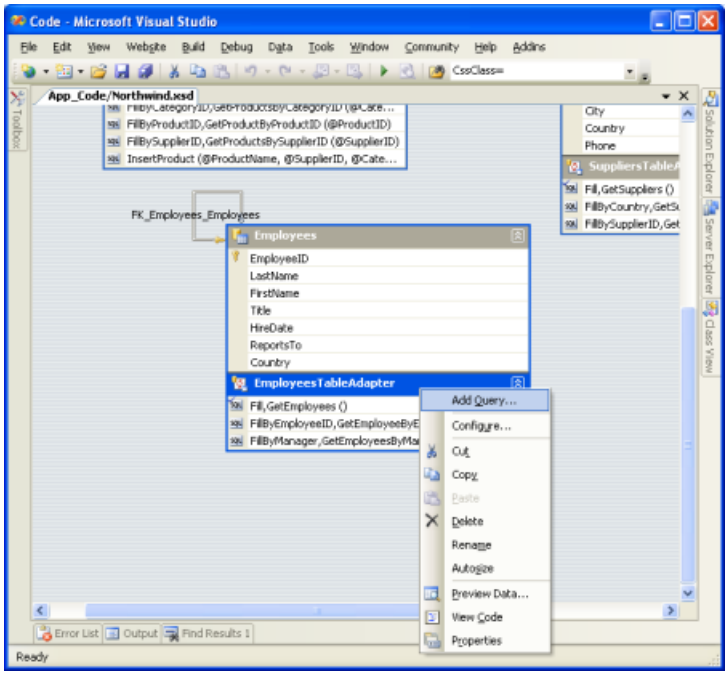


Figure 2: Add a New Query to the `EmployeesTableAdapter` (Click to view full-size image)

Choose to add a SQL statement that returns rows. When you reach the Specify a `SELECT` Statement screen the default `SELECT` statement for the `EmployeesTableAdapter` will already be loaded. Simply add in the `WHERE` clause: `WHERE DATEPART(m, HireDate) = @Month`. DATEPART is a T-SQL function that returns a particular date portion of a `datetime` type; in this case we're using `DATEPART` to return the month of the `HireDate` column.

Figure 3: Return Only Those Rows Where the `HireDate` Column is Less Than or Equal to the `@HiredBeforeDate` Parameter (Click to view full-size image)

Finally, change the `FillBy` and `GetDataBy` method names to `FillByHiredDateMonth` and `GetEmployeesByHiredDateMonth`, respectively.

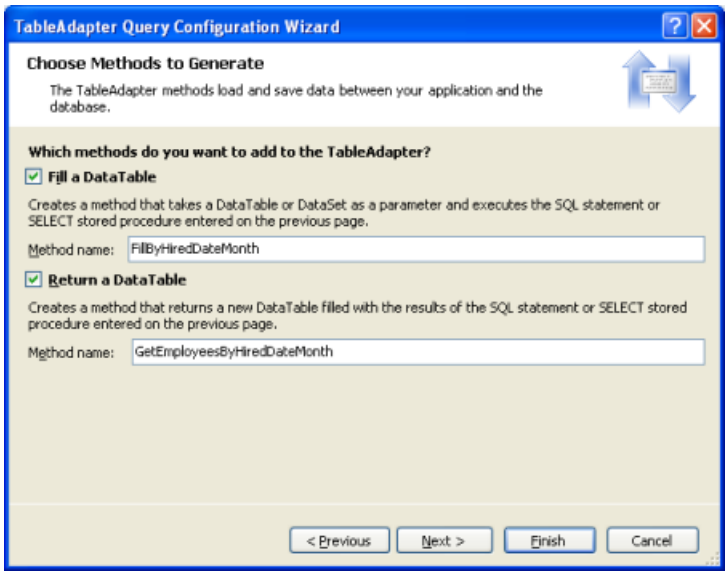


Figure 4: Choose More Appropriate Method Names Than `FillBy` and `GetDataBy` (Click to view full-size image)

Click Finish to complete the wizard and return to the DataSet's design surface. The `EmployeesTableAdapter` should now include a new set of methods for accessing employees hired in a specified month.

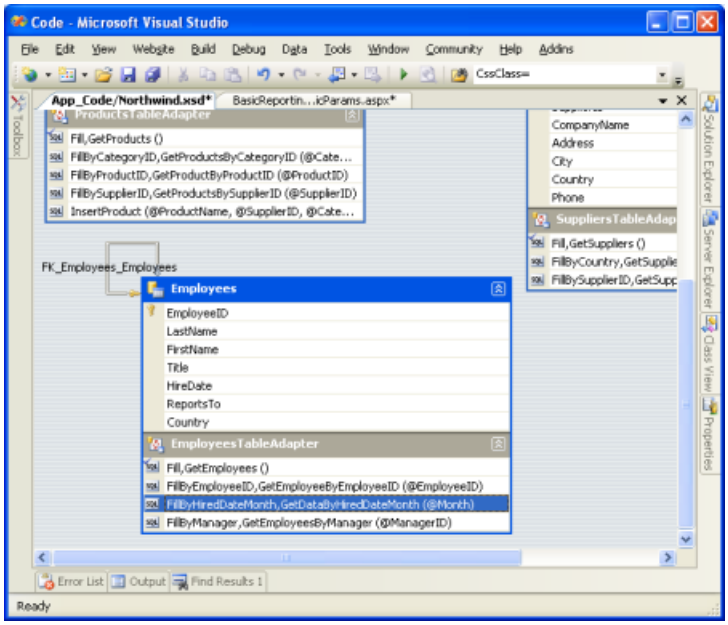


Figure 5: The New Methods Appear in the DataSet's Design Surface (Click to view full-size image)

Step 2: Adding the `GetEmployeesByHiredDateMonth(month)` Method to the Business Logic Layer

Since our application architecture uses a separate layer for the business logic and data access logic, we need to add a method to our BLL that calls down to the DAL to retrieve employees hired before a specified date. Open the `EmployeesBLL.vb` file and add the following method:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetEmployeesByHiredDateMonth(ByVal month As Integer) _
    As Northwind.EmployeesDataTable
    Return Adapter.GetEmployeesByHiredDateMonth(month)
End Function
```

As with our other methods in this class, `GetEmployeesByHiredDateMonth(month)` simply calls down into the DAL and

returns the results.

Step 3: Displaying Employees Whose Hiring Anniversary Is This Month

Our final step for this example is to display those employees whose hiring anniversary is this month. Start by adding a GridView to the `ProgrammaticParams.aspx` page in the `BasicReporting` folder and add a new ObjectDataSource as its data source. Configure the ObjectDataSource to use the `EmployeesBLL` class with the `SelectMethod` set to `GetEmployeesByHiredDateMonth(month)`.

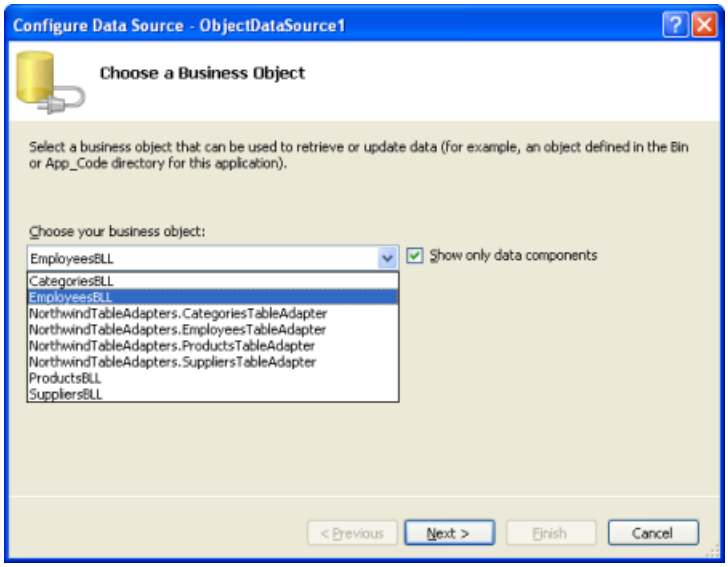


Figure 6: Use the `EmployeesBLL` Class (Click to view full-size image)

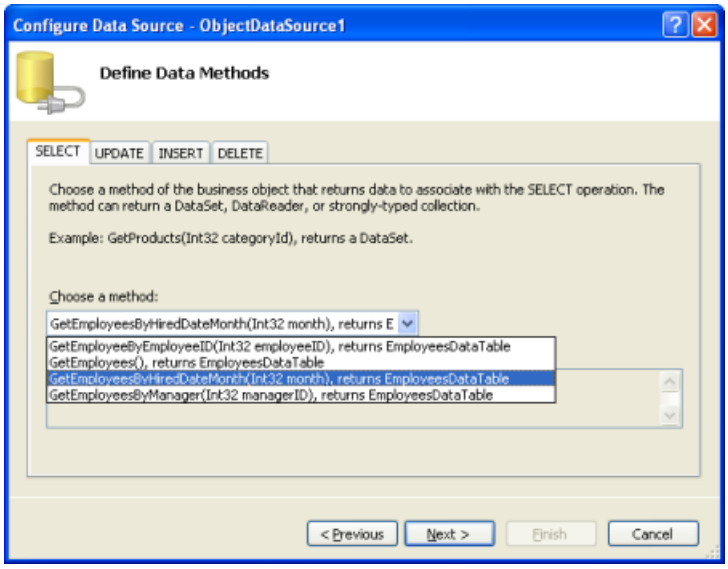


Figure 7: Select From the `GetEmployeesByHiredDateMonth(month)` method (Click to view full-size image)

The final screen asks us to provide the `month` parameter value's source. Since we'll set this value programmatically, leave the Parameter source set to the default None option and click Finish.

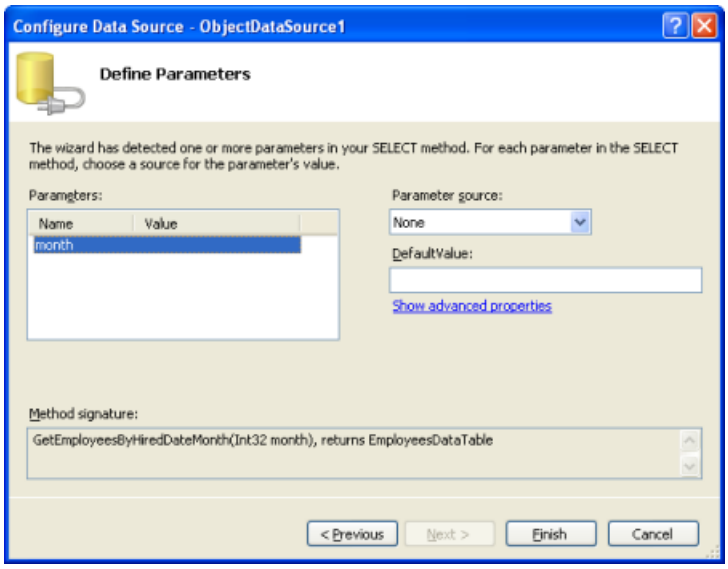


Figure 8: Leave the Parameter Source Set to None (Click to view full-size image)

This will create a `Parameter` object in the ObjectDataSource's `SelectParameters` collection that does not have a value specified.

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetEmployeesByHiredDateMonth" TypeName="EmployeesBLL">
    <SelectParameters>
        <asp:Parameter Name="month" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

To set this value programmatically, we need to create an event handler for the ObjectDataSource's `Selecting` event. To accomplish this, go to the Design view and double-click the ObjectDataSource. Alternatively, select the ObjectDataSource, go to the Properties window, and click the lightning bolt icon. Next, either double-click in the textbox next to the `Selecting` event or type in the name of the event handler you want to use. As a third option, you can create the event handler by selecting the ObjectDataSource and its `Selecting` event from the two drop-down lists at the top of the page's code-behind class.

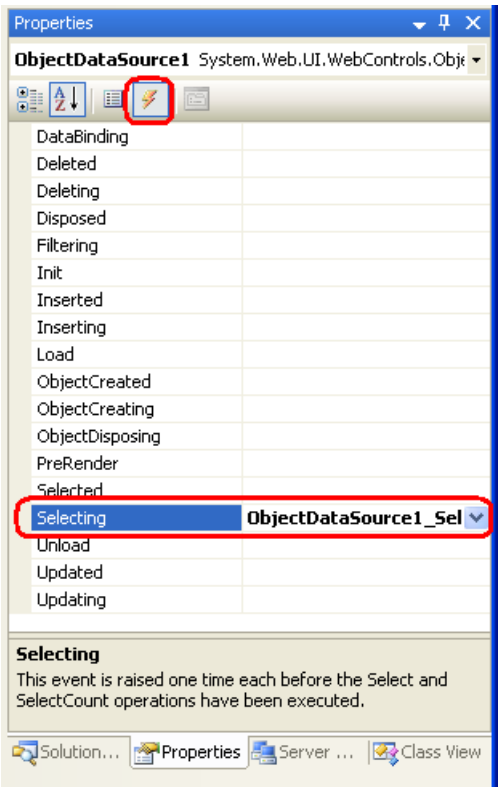


Figure 9: Click on the Lightning Bolt Icon in the Properties Window to List a Web Control's Events

All three approaches add a new event handler for the ObjectDataSource's `Selecting` event to the page's code-behind class. In this event handler we can read and write to the parameter values using `e.InputParameters(parameterName)`, where *`parameterName`* is the value of the `Name` attribute in the `<asp:Parameter>` tag (the `InputParameters` collection can also be indexed ordinally, as in `e.InputParameters(index)`). To set the `month` parameter to the current month, add the following to the `Selecting` event handler:

```
Protected Sub ObjectDataSource1_Selecting _
    (sender As Object, e As ObjectDataSourceSelectingEventArgs) _
        Handles ObjectDataSource1.Selecting
    e.InputParameters("month") = DateTime.Now.Month
End Sub
```

When visiting this page through a browser we can see that only one employee was hired this month (March) Laura Callahan, who's been with the company since 1994.

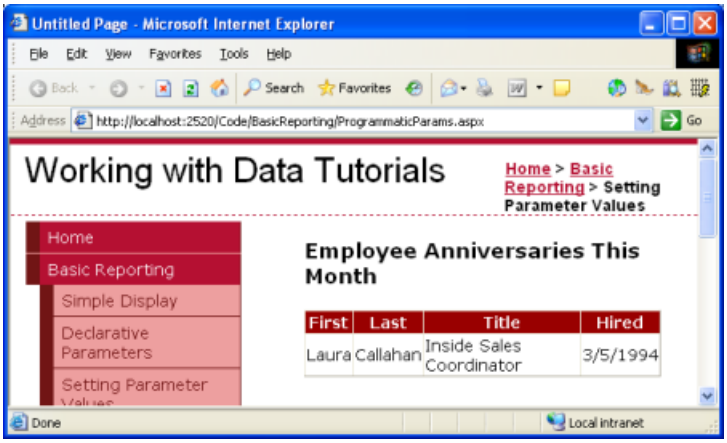


Figure 10: Those Employees Whose Anniversaries This Month Are Shown (Click to view full-size image)

Summary

While the ObjectDataSource's parameters' values can typically be set declaratively, without requiring a line of code, it's easy to set the parameter values programmatically. All we need to do is create an event handler for the ObjectDataSource's `Selecting` event, which fires before the underlying object's method is invoked, and manually set the values for one or more parameters via the `InputParameters` collection.

This tutorial concludes the Basic Reporting section. The next tutorial kicks off the Filtering and Master-Details Scenarios section, in which we'll look at techniques for allowing the visitor to filter data and drill down from a master report into a details report.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Master/Detail

2/20/2020 • 2 minutes to read • Edit Online

These tutorials show how to display master records in a drop-down list and details in a GridView control.

- Master/Detail Filtering With a DropDownList (C#)
- Master/Detail Filtering With Two DropDownLists (C#)
- Master/Detail Filtering Across Two Pages (C#)
- Master/Detail Using a Selectable Master GridView with a Details DetailView (C#)
- Master/Detail Filtering With a DropDownList (VB)
- Master/Detail Filtering With Two DropDownLists (VB)
- Master/Detail Filtering Across Two Pages (VB)
- Master/Detail Using a Selectable Master GridView with a Details DetailView (VB)

Master/Detail Filtering With a DropDownList (C#)

2/20/2020 • 7 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll see how to display the master records in a DropDownList control and the details of the selected list item in a GridView.

Introduction

A common type of report is the *master/detail report*, in which the report begins by showing some set of "master" records. The user can then drill down into one of the master records, thereby viewing that master record's "details." Master/detail reports are an ideal choice for visualizing one-to-many relationships, such as a report showing all of the categories and then allowing a user to select a particular category and display its associated products. Additionally, master/detail reports are useful for displaying detailed information from particularly "wide" tables (ones that have a lot of columns). For example, the "master" level of a master/detail report might show just the product name and unit price of the products in the database, and drilling down into a particular product would show the additional product fields (category, supplier, quantity per unit, and so on).

There are many ways with which a master/detail report can be implemented. Over this and the next three tutorials we'll look at a variety of master/detail reports. In this tutorial we'll see how to display the master records in a DropDownList control and the details of the selected list item in a GridView. In particular, this tutorial's master/detail report will list category and product information.

Step 1: Displaying the Categories in a DropDownList

Our master/detail report will list the categories in a DropDownList, with the selected list item's products displayed further down in the page in a GridView. The first task ahead of us, then, is to have the categories displayed in a DropDownList. Open the `FilterByDropDownList.aspx` page in the `Filtering` folder, drag on a DropDownList from the Toolbox onto the page's designer, and set its `ID` property to `Categories`. Next, click on the Choose Data Source link from the DropDownList's smart tag. This will display the Data Source Configuration wizard.

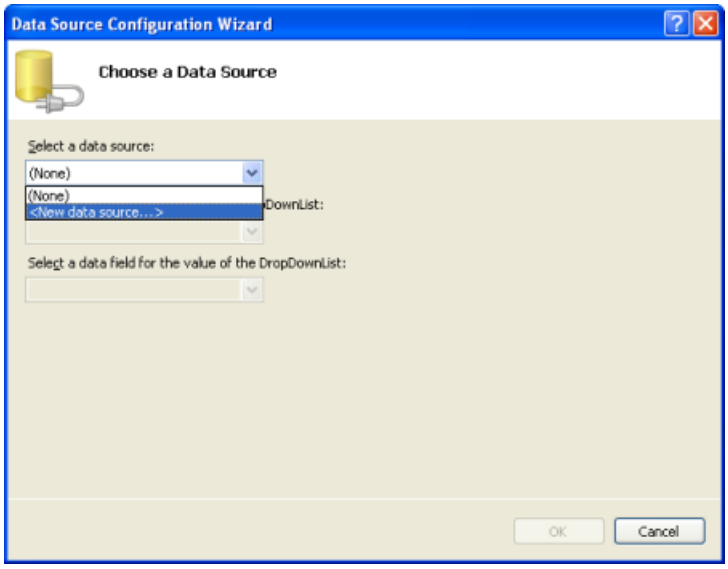


Figure 1: Specify the DropDownList's Data Source (Click to view full-size image)

Choose to add a new ObjectDataSource named `CategoriesDataSource` that invokes the `CategoriesBLL` class's `GetCategories()` method.

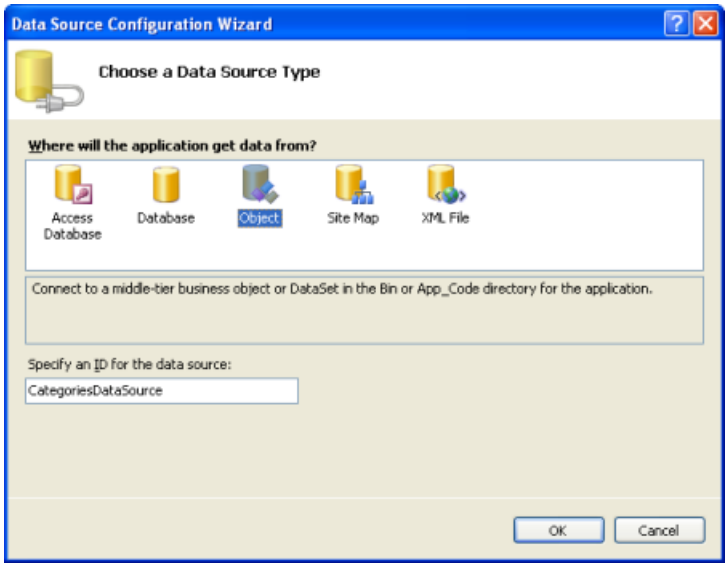


Figure 2: Add a New ObjectDataSource Named `CategoriesDataSource` (Click to view full-size image)

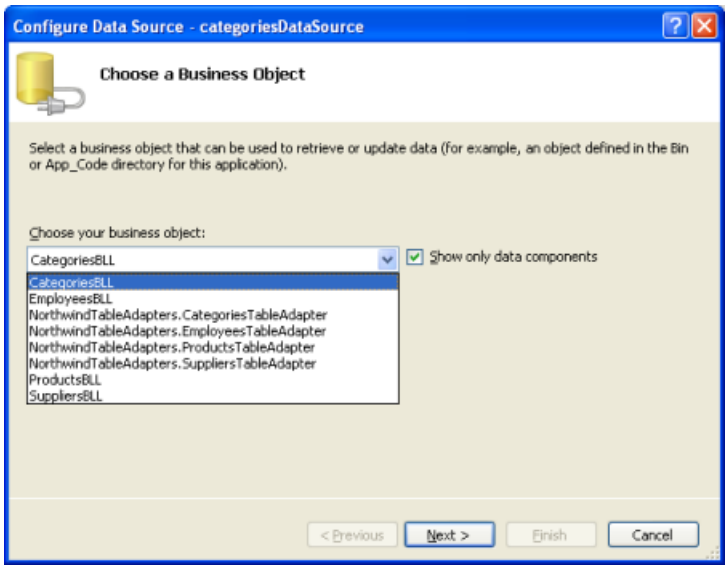


Figure 3: Choose to Use the `CategoriesBLL` Class (Click to view full-size image)

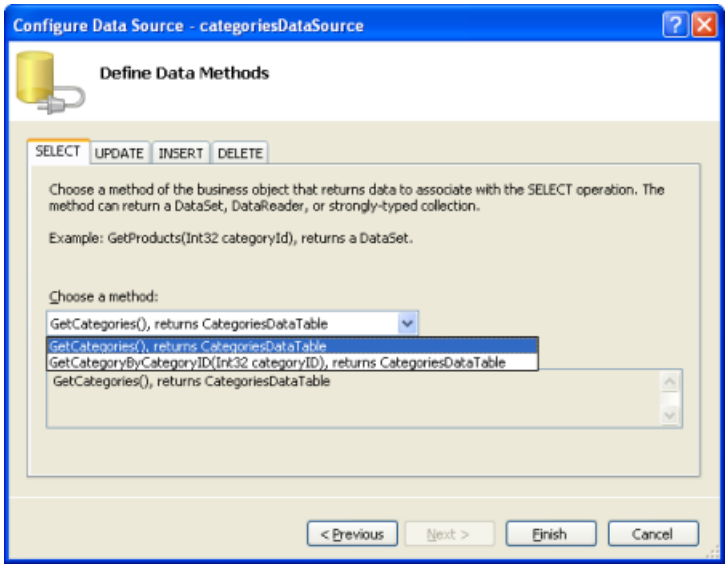


Figure 4: Configure the ObjectDataSource to Use the `GetCategories()` Method (Click to view full-size image)

After configuring the ObjectDataSource we still need to specify what data source field should be displayed in DropDownList and which one should be associated as the value for the list item. Have the `CategoryName` field as the display and `CategoryID` as the value for each list item.

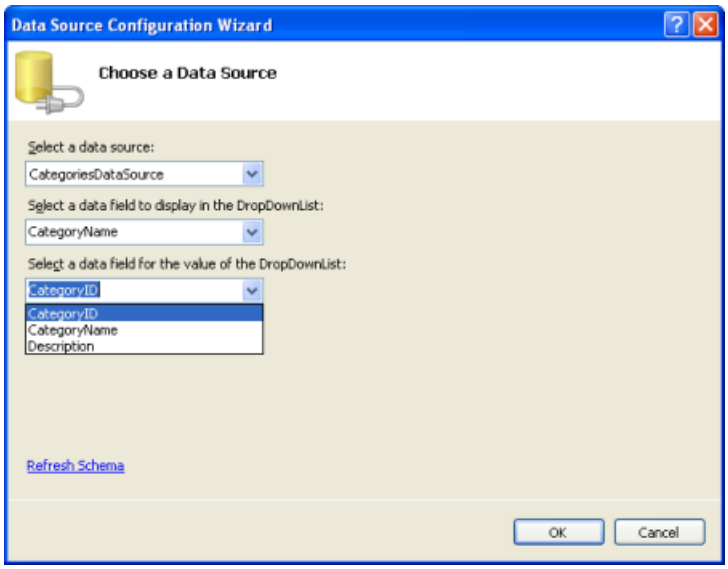


Figure 5: Have the DropDownList Display the `CategoryName` Field and Use `CategoryID` as the Value (Click to view full-size image)

At this point we have a DropDownList control that's populated with the records from the `Categories` table (all accomplished in about six seconds). Figure 6 shows our progress thus far when viewed through a browser.

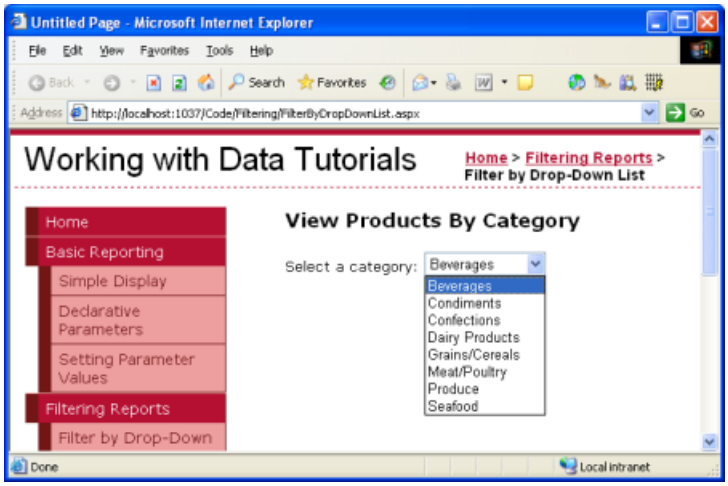


Figure 6: A Drop-Down Lists the Current Categories (Click to view full-size image)

Step 2: Adding the Products GridView

That last step in our master/detail report is to list the products associated with the selected category. To accomplish this, add a GridView to the page and create a new ObjectDataSource named `productsDataSource`. Have the `productsDataSource` control cull its data from the `ProductsBLL` class's `GetProductsByCategoryID(categoryID)` method.

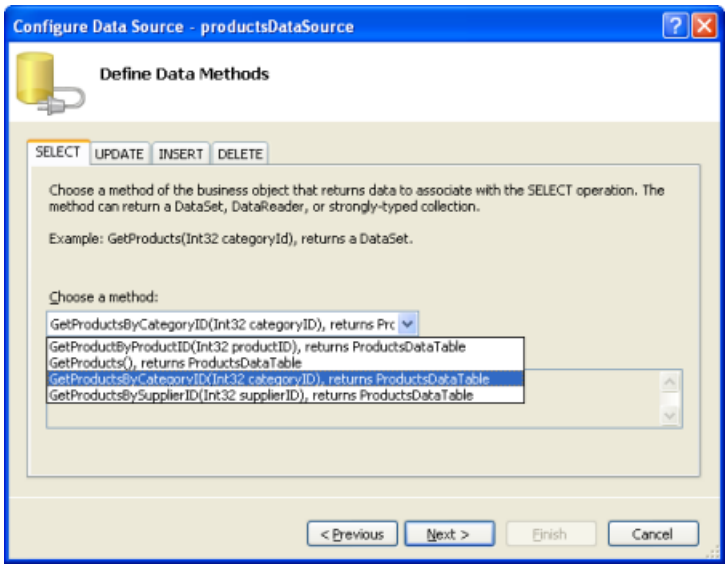


Figure 7: Select the `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

After choosing this method, the ObjectDataSource wizard prompts us for the value for the method's *`categoryID`* parameter. To use the value of the selected `categories` DropDownList item set the Parameter source to Control and the ControlID to `Categories`.

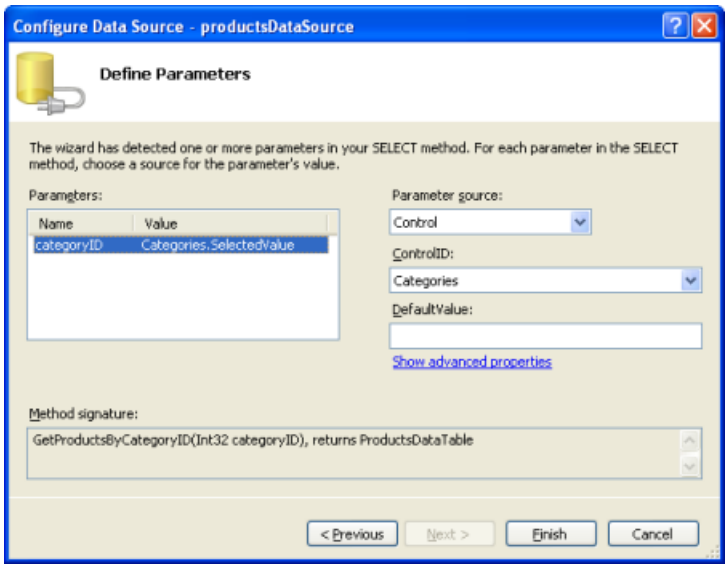


Figure 8: Set the *`categoryID`* Parameter to the Value of the `Categories` DropDownList (Click to view full-size image)

Take a moment to check out our progress in a browser. When first visiting the page, those products belong to the selected category (Beverages) are displayed (as shown in Figure 9), but changing the DropDownList doesn't update the data. This is because a postback must occur for the GridView to update. To accomplish this we have two options (neither of which requires writing any code):

- **Set the categories DropDownList's** AutoPostBack property **to True.** (You can accomplish this by checking the Enable AutoPostBack option in the DropDownList's smart tag.) This will trigger a postback whenever the DropDownList's selected item is changed by the user. Therefore, when the user selects a new category from the DropDownList a postback will ensue and the GridView will be updated with the products for the newly selected category. (This is the approach I've used in this tutorial.)
- **Add a Button Web control next to the DropDownList.** Set its `Text` property to Refresh or something similar. With this approach, the user will need to select a new category and then click the Button. Clicking the Button will cause a postback and update the GridView to list those products of the selected category.

Figures 9 and 10 illustrate the master/detail report in action.

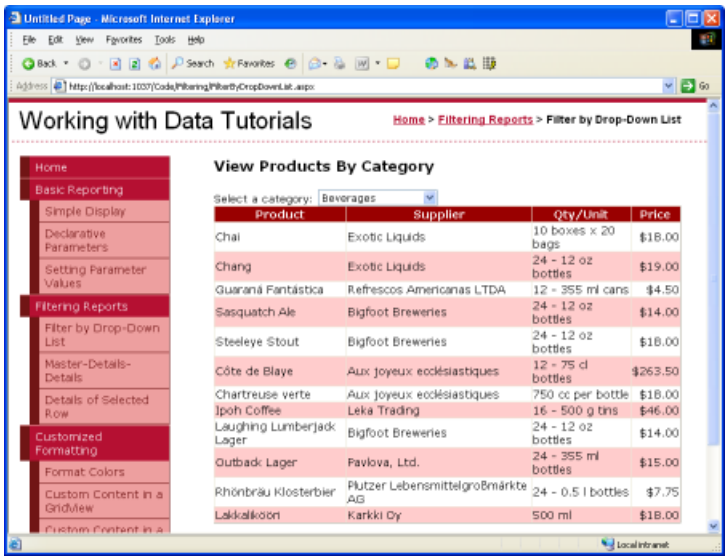


Figure 9: When First Visiting the Page, the Beverage Products are Displayed (Click to view full-size image)

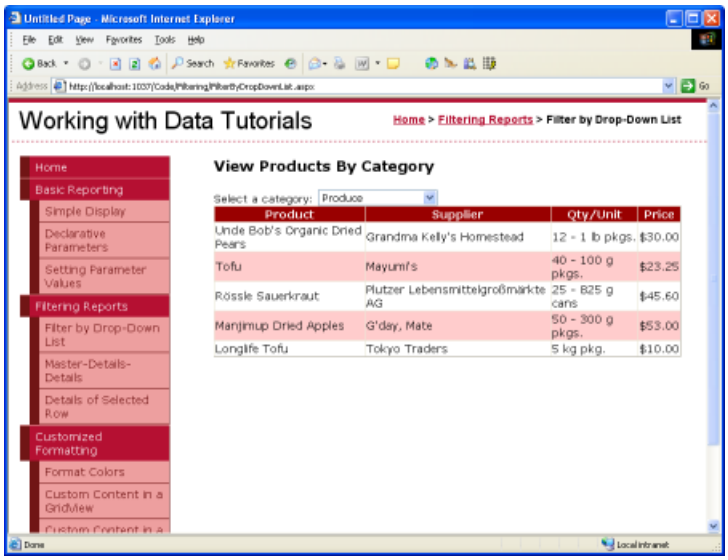


Figure 10: Selecting a New Product (Produce) Automatically Causes a PostBack, Updating the GridView (Click to view full-size image)

Adding a "-- Choose a Category --" List Item

When first visiting the `FilterByDropDownList.aspx` page the categories DropDownList's first list item (Beverages) is selected by default, showing the beverage products in the GridView. Rather than showing the first category's products, we may want to instead have a DropDownList item selected that says something like, "-- Choose a Category --".

To add a new list item to the DropDownList, go to the Properties window and click on the ellipses in the `Items` property. Add a new list item with the `Text` "-- Choose a Category --" and the `Value` `-1`.

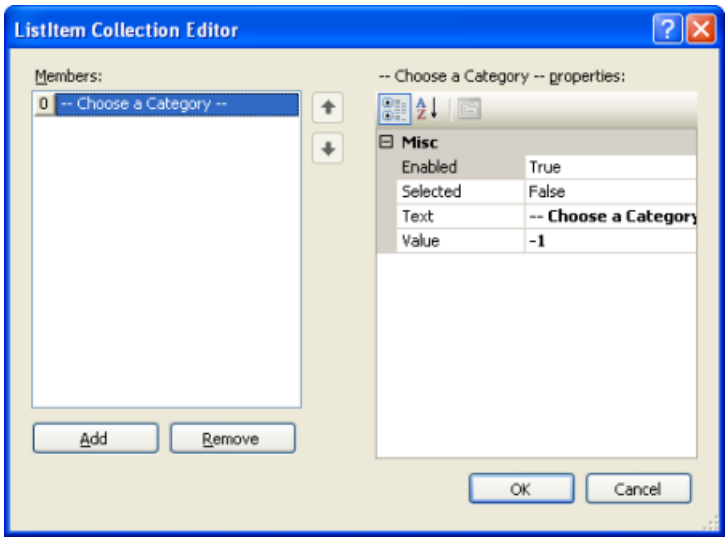


Figure 11: Add a -- Choose a Category -- List Item (Click to view full-size image)

Alternatively, you can add the list item by adding the following markup to the DropDownList:

```
<asp:DropDownList ID="categories" runat="server"
    AutoPostBack="True" DataSourceID="categoriesDataSource"
    DataTextField="CategoryName" DataValueField="CategoryID"
    EnableViewState="False">
    <asp:ListItem Value="-1">
       -- Choose a Category --
    </asp:ListItem>
</asp:DropDownList>
```

Additionally, we need to set the DropDownList control's `AppendDataBoundItems` to True because when the categories are bound to the DropDownList from the ObjectDataSource they'll overwrite any manually-added list items if `AppendDataBoundItems` isn't True.

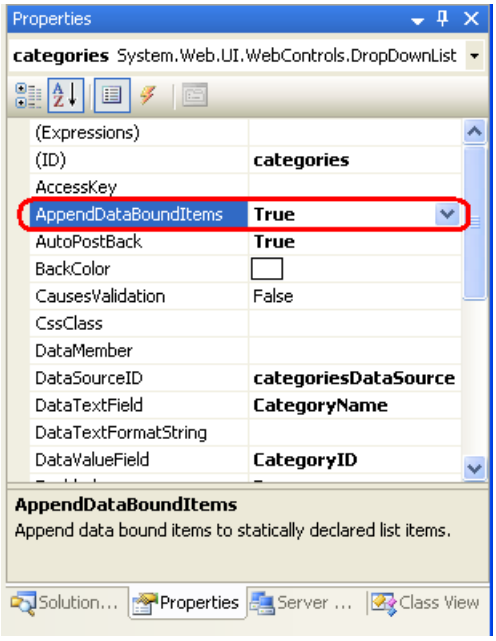


Figure 12: Set the `AppendDataBoundItems` Property to True

After these changes, when first visiting the page the "-- Choose a Category --" option is selected and no products are displayed.

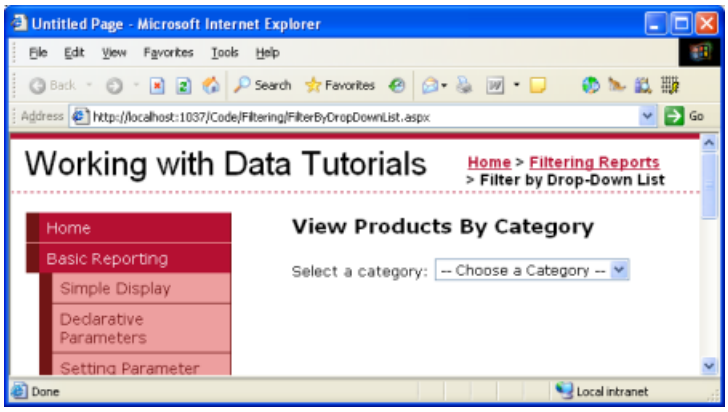


Figure 13: On the Initial Page Load No Products are Displayed (Click to view full-size image)

The reason no products are displayed when because the "-- Choose a Category --" list item is selected is because its value is `-1` and there are no products in the database with a `CategoryID` of `-1`. If this is the behavior you want then you're done at this point! If, however, you want to display *all* of the categories when the "-- Choose a Category --" list item is selected, return to the `ProductsBLL` class and customize the `GetProductsByCategoryID(categoryID)` method so that it invokes the `GetProducts()` method if the passed in *`categoryID`* parameter is less than zero:

```
public Northwind.ProductsDataTable GetProductsByCategoryID(int categoryID)
{
    if (categoryID < 0)
        return GetProducts();
    else
        return Adapter.GetProductsByCategoryID(categoryID);
}
```

The technique used here is similar to the approach we used to display all suppliers back in the Declarative Parameters tutorial, although for this example we're using a value of `-1` to indicate that all records should be retrieved as opposed to `null`. This is because the *`categoryID`* parameter of the `GetProductsByCategoryID(categoryID)` method expects as integer value passed in, whereas in the Declarative Parameters tutorial we were passing in a string input parameter.

Figure 14 shows a screen shot of `FilterByDropDownList.aspx` when the "-- Choose a Category --" option is selected. Here, all of the products are displayed by default, and the user can narrow the display by choosing a specific category.

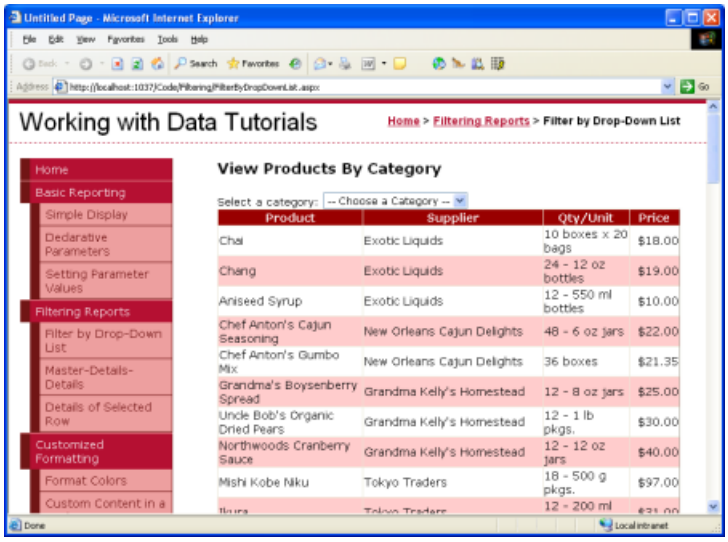


Figure 14: All of the Products are Now Listed By Default (Click to view full-size image)

Summary

When displaying hierarchically-related data, it often helps to present the data using master/detail reports, from which the user can start perusing the data from the top of the hierarchy and drill down into details. In this tutorial we examined building a simple master/detail report showing a selected category's products. This was accomplished by using a DropDownList for the list of categories and a GridView for the products belonging to the selected category.

In the next tutorial we'll take the DropDownList interface one step further, using two DropDownLists.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

NEXT

Master/Detail Filtering With Two DropDownLists (C#)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> This tutorial expands the master/detail relationship to add a third layer, using two DropDownList controls to select the desired parent and grandparent records.

Introduction

In the previous tutorial we examined how to display a simple master/details report using a single DropDownList populated with the categories and a GridView showing those products that belong to the selected category. This report pattern works well when displaying records that have a one-to-many relationship and can easily be extended to work for scenarios that include multiple one-to-many relationships. For example, an order entry system would have tables that correspond to customers, orders, and order line items. A given customer may have multiple orders with each order consisting of multiple items. Such data can be presented to the user with two DropDownLists and a GridView. The first DropDownList would have a list item for each customer in the database with the second one's contents being the orders placed by the selected customer. A GridView would list the line items from the selected order.

While the Northwind database include the canonical customer/order/order details information in its `Customers`, `Orders`, and `Order Details` tables, these tables aren't captured in our architecture. Nonetheless, we can still illustrate using two dependent DropDownLists. The first DropDownList will list the categories and the second the products belonging to the selected category. A DetailsView will then list the details of the selected product.

Step 1: Creating and Populating the Categories DropDownList

Our first goal is to add the DropDownList that lists the categories. These steps were examined in detail in the preceding tutorial, but are summarized here for completeness.

Open the `MasterDetailsDetails.aspx` page in the `Filtering` folder, add a DropDownList to the page, set its `ID` property to `Categories`, and then click the Configure Data Source link in its smart tag. From the Data Source Configuration Wizard choose to add a new data source.

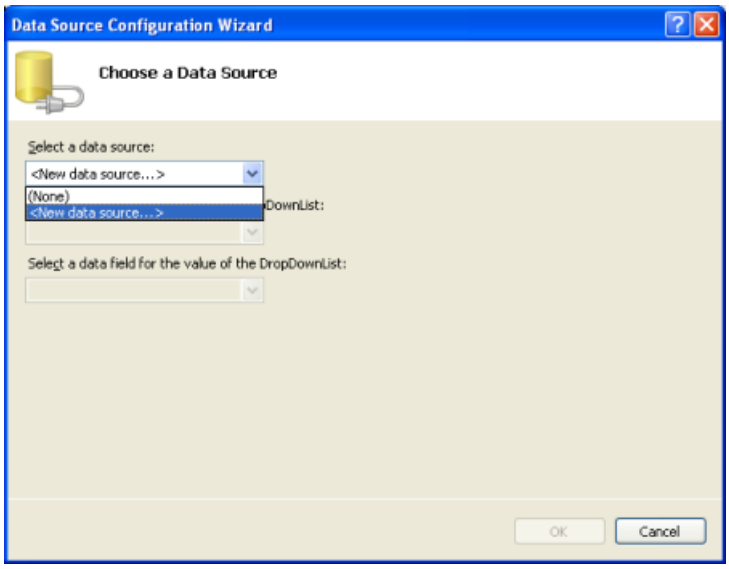


Figure 1: Add a New Data Source for the DropDownList (Click to view full-size image)

The new data source should, naturally, be an ObjectDataSource. Name this new ObjectDataSource `CategoriesDataSource` and have it invoke the `CategoriesBLL` object's `GetCategories()` method.

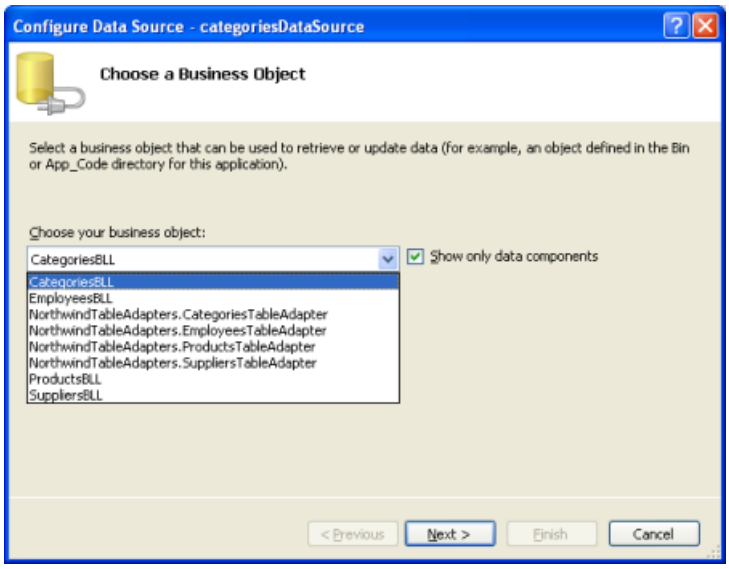


Figure 2: Choose to Use the `CategoriesBLL` Class (Click to view full-size image)

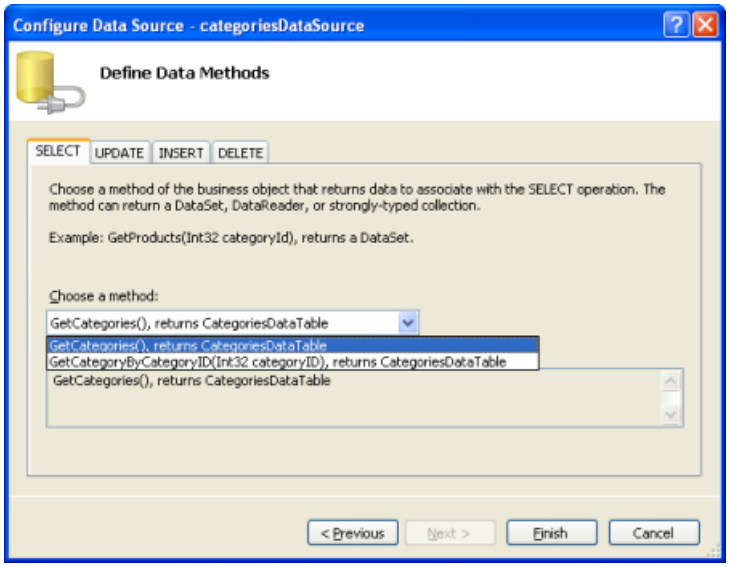


Figure 3: Configure the ObjectDataSource to Use the `GetCategories()` Method (Click to view full-size image)

After configuring the ObjectDataSource we still need to specify which data source field should be displayed in the `Categories` DropDownList and which one should be configured as the value for the list item. Set the `CategoryName` field as the display and `CategoryID` as the value for each list item.

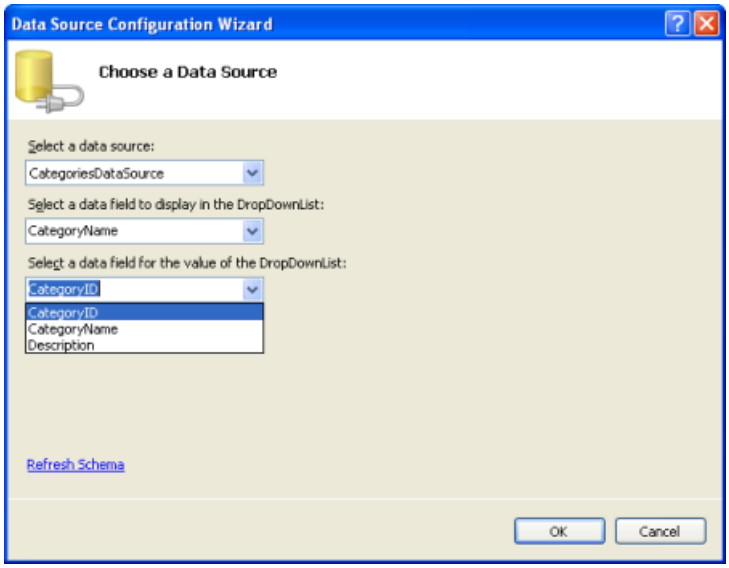


Figure 4: Have the DropDownList Display the `CategoryName` Field and Use `CategoryID` as the Value (Click to view full-size image)

At this point we have a DropDownList control (`Categories`) that's populated with the records from the `Categories` table. When the user chooses a new category from the DropDownList we'll want a postback to occur in order to refresh the product DropDownList that we're going to create in Step 2. Therefore, check the Enable AutoPostBack option from the `categories` DropDownList's smart tag.

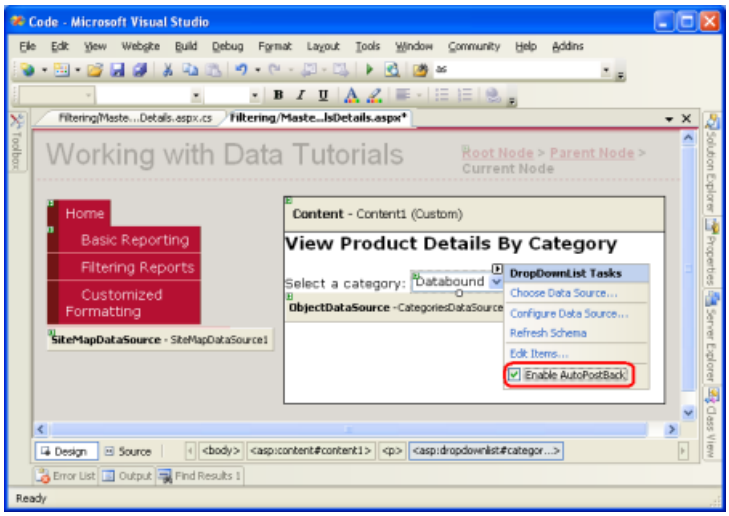


Figure 5: Enable AutoPostBack for the `Categories` DropDownList (Click to view full-size image)

Step 2: Displaying the Selected Category's Products in a Second DropDownList

With the `Categories` DropDownList completed, our next step is to display a DropDownList of products belonging to the selected category. To accomplish this, add another DropDownList to the page named `ProductsByCategory`. As with the `Categories` DropDownList, create a new ObjectDataSource for the `ProductsByCategory` DropDownList named `ProductsByCategoryDataSource`.

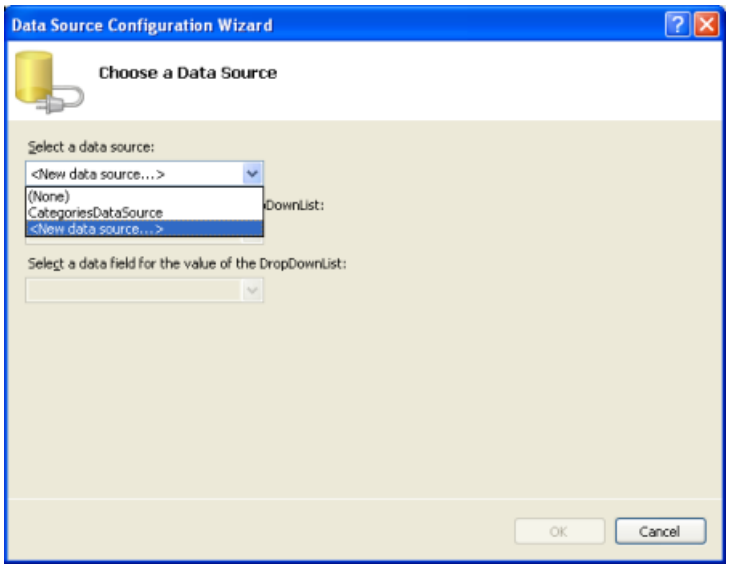


Figure 6: Add a New Data Source for the `ProductsByCategory` DropDownList (Click to view full-size image)

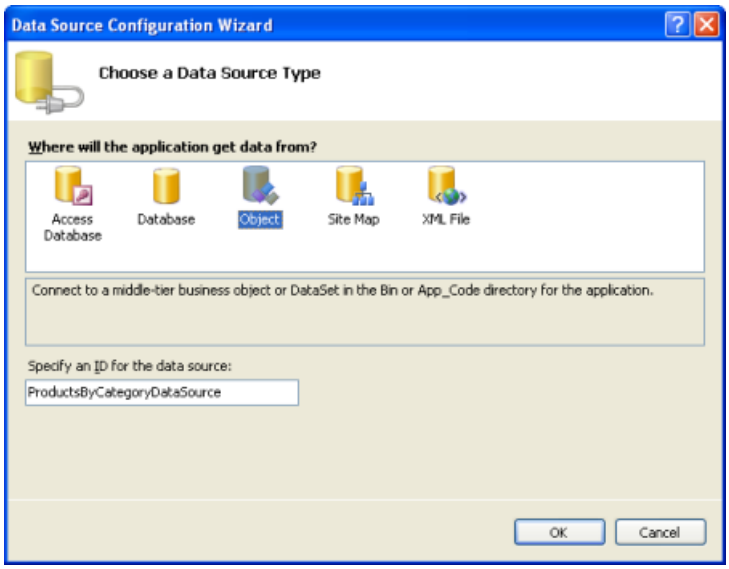


Figure 7: Create a New ObjectDataSource Named `ProductsByCategoryDataSource` (Click to view full-size image)

Since the `ProductsByCategory` DropDownList needs to display just those products belonging to the selected category, have the ObjectDataSource invoke the `GetProductsByCategoryID(categoryID)` method from the `ProductsBLL` object.

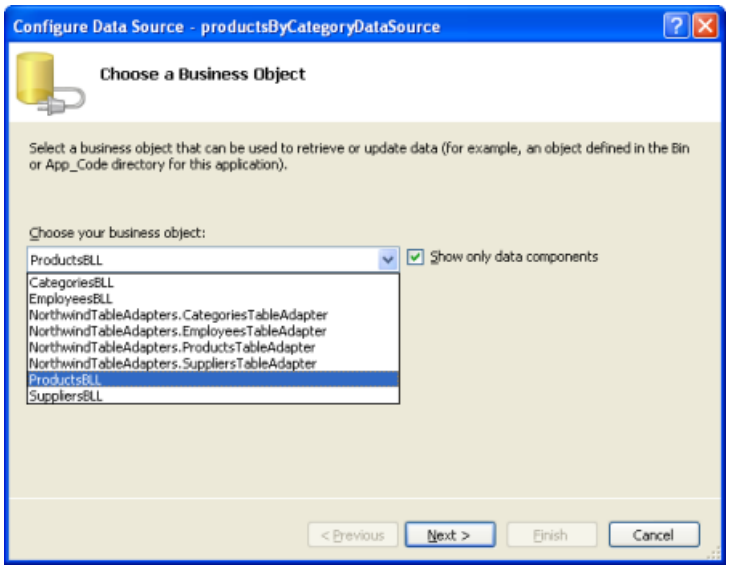


Figure 8: Choose to Use the `ProductsBLL` Class (Click to view full-size image)

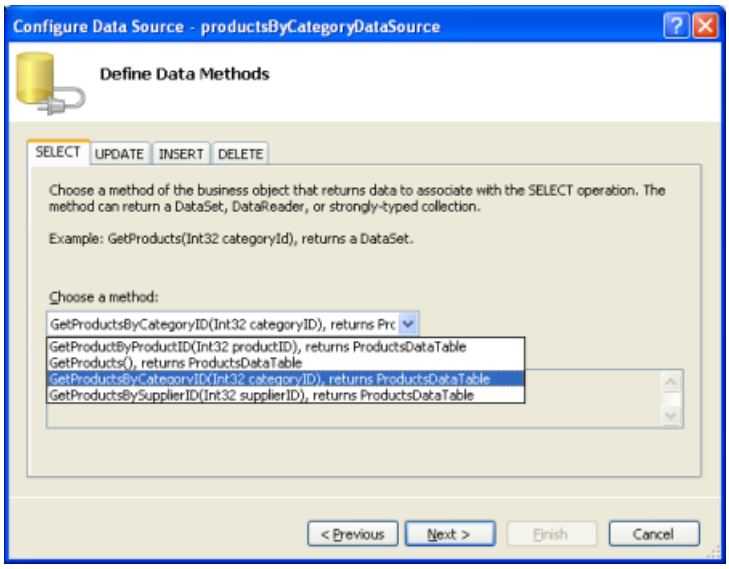


Figure 9: Configure the ObjectDataSource to Use the `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

In the final step of the wizard we need to specify the value of the *`categoryID`* parameter. Assign this parameter to the selected item from the `Categories` DropDownList.

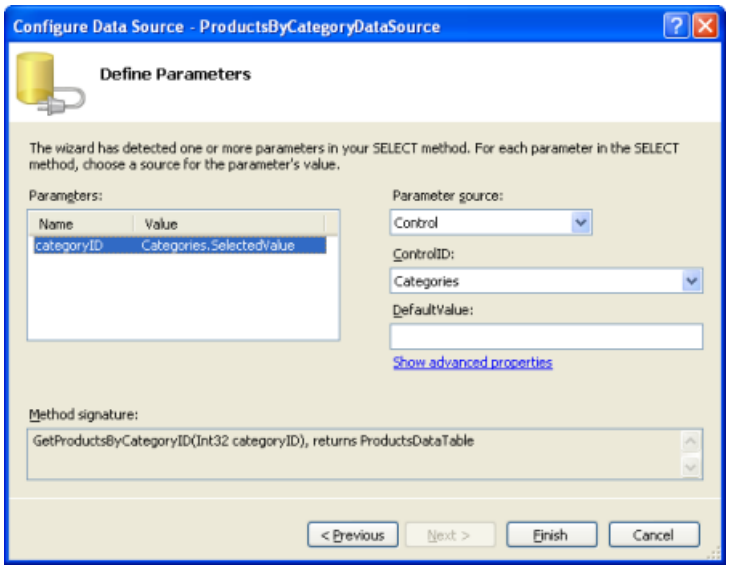


Figure 10: Pull the *`categoryID`* Parameter Value from the `Categories` DropDownList (Click to view full-size image)

With the ObjectDataSource configured, all that remains is to specify what data source fields are used for the display and value of the DropDownList's items. Display the `ProductName` field and use the `ProductID` field as the value.

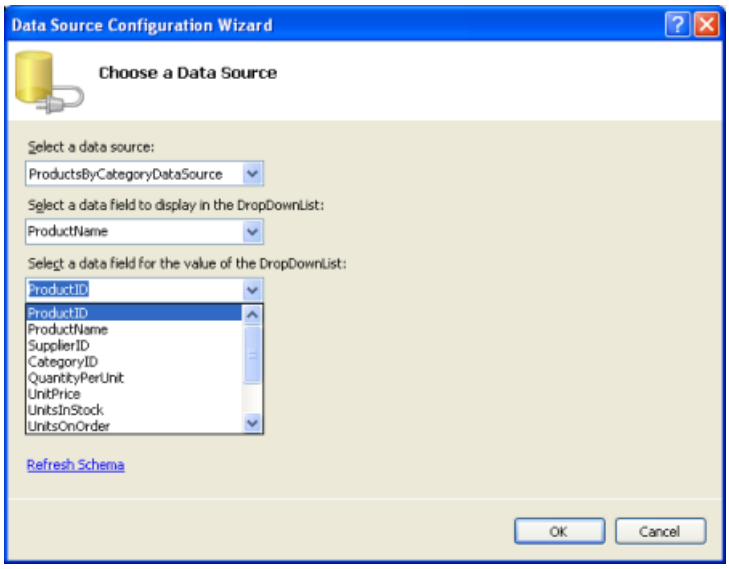


Figure 11: Specify the Data Source Fields Used for the DropDownList's `ListItem` s' `Text` and `Value` Properties (Click to view full-size image)

With the ObjectDataSource and `ProductsByCategory` DropDownList configured our page will display two DropDownLists: the first will list all of the categories while the second will list those products belonging to the selected category. When the user selects a new category from the first DropDownList, a postback will ensue and the second DropDownList will be rebound, showing those products that belong to the newly selected category. Figures 12 and 13 show `MasterDetailsDetails.aspx` in action when viewed through a browser.

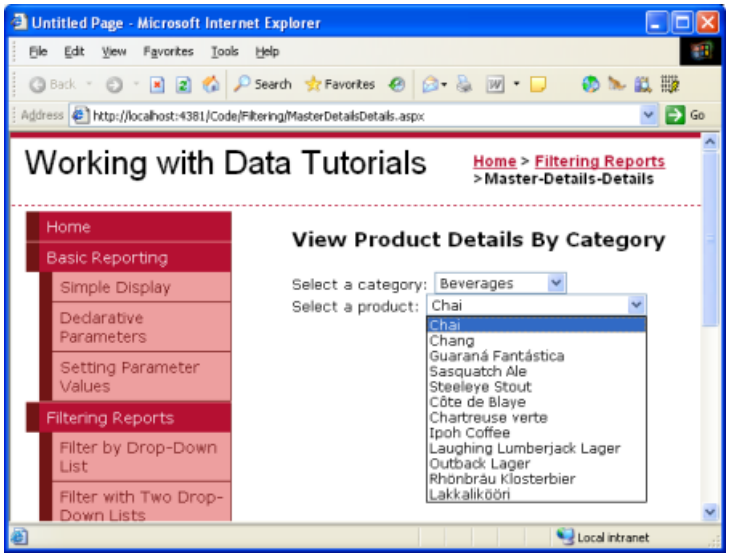


Figure 12: When First Visiting the Page, the Beverages Category is Selected (Click to view full-size image)

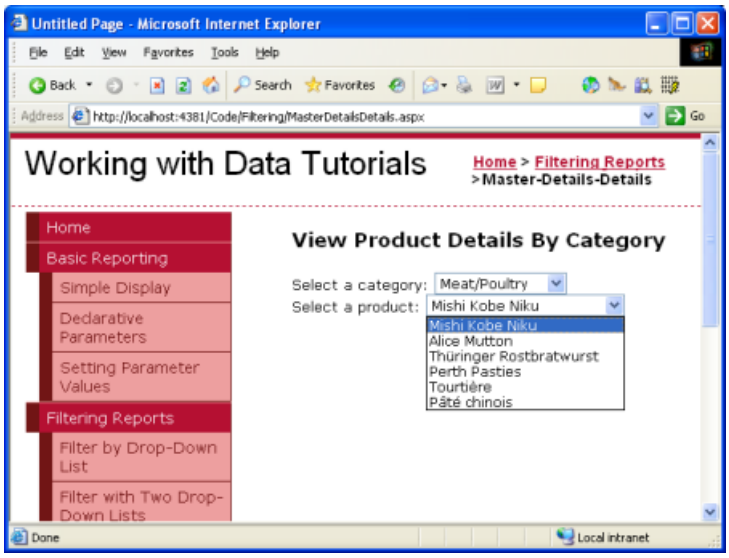


Figure 13: Choosing a Different Category Displays the New Category's Products (Click to view full-size image)

Currently the `productsByCategory` DropDownList, when changed, does *not* cause a postback. However, we will want a postback to occur once we add a DetailsView to display the selected product's details (Step 3). Therefore, check the Enable AutoPostBack checkbox from the `productsByCategory` DropDownList's smart tag.

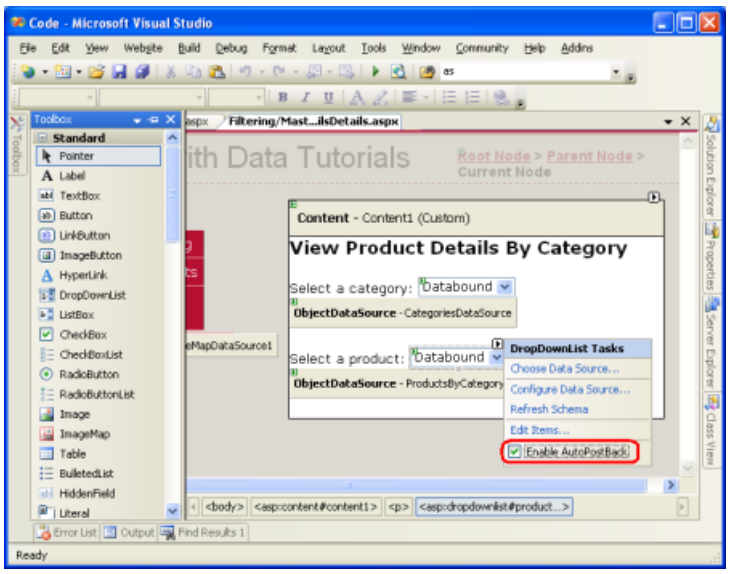


Figure 14: Enable the AutoPostBack Feature for the `productsByCategory` DropDownList (Click to view full-size image)

Step 3: Using a DetailsView to Display Details for the Selected Product

The final step is to display the details for the selected product in a DetailsView. To accomplish this, add a DetailsView to the page, set its `ID` property to `ProductDetails`, and create a new ObjectDataSource for it. Configure this ObjectDataSource to pull its data from the `ProductsBLL` class's `GetProductByProductID(productID)` method using the selected value of the `ProductsByCategory` DropDownList for the value of the *`productID`* parameter.

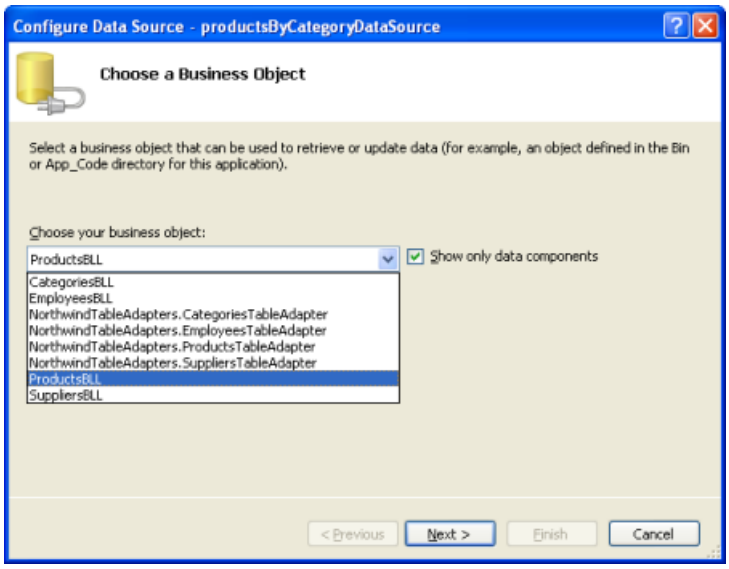


Figure 15: Choose to Use the `ProductsBLL` Class (Click to view full-size image)

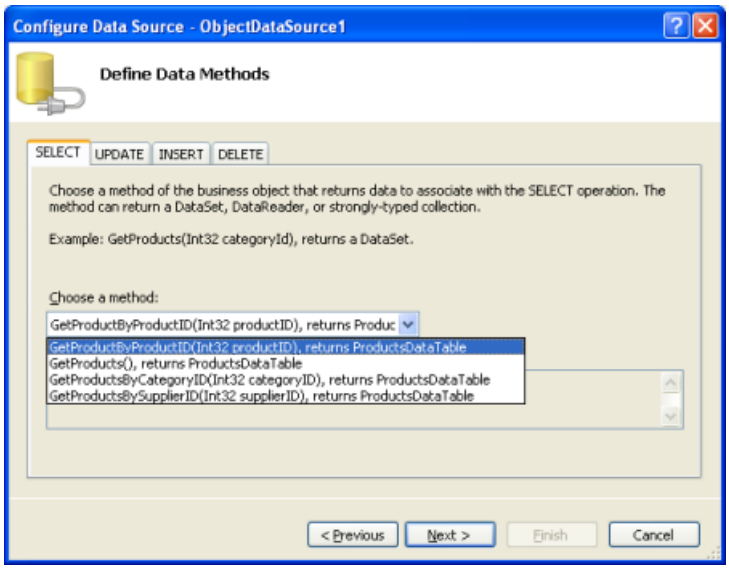


Figure 16: Configure the ObjectDataSource to Use the `GetProductByProductID(productID)` Method (Click to view full-size image)

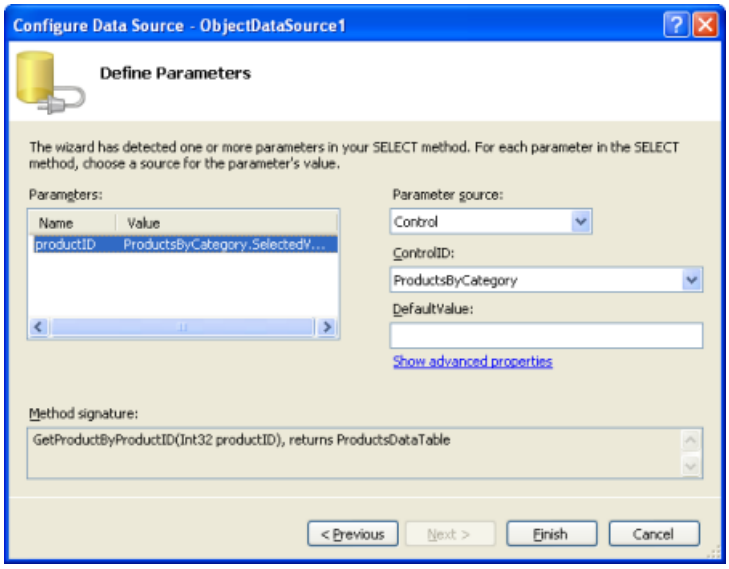


Figure 17: Pull the `productID` Parameter Value from the `ProductsByCategory` DropDownList (Click to view full-size image)

You can choose to display any of the available fields in the DetailsView. I've opted to remove the `ProductID`,

`SupplierID`, and `CategoryID` fields and reordered and formatted the remaining fields. In addition, I cleared out the DetailsView's `Height` and `Width` properties, allowing the DetailsView to expand to the width needed to best display its data rather than having it constrained to a specified size. The full markup appears below:

```
<asp:DetailsView ID="ProductDetails" runat="server"
    AutoGenerateRows="False" DataKeyNames="ProductID"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <Fields>
        <asp:BoundField DataField="ProductName"
          HeaderText="Product" SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName"
          HeaderText="Category" ReadOnly="True"
          SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName"
          HeaderText="Supplier" ReadOnly="True"
          SortExpression="SupplierName" />
        <asp:BoundField DataField="QuantityPerUnit"
          HeaderText="Qty/Unit" SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice"
          DataFormatString="{0:c}" HeaderText="Price"
          HtmlEncode="False" SortExpression="UnitPrice" />
        <asp:BoundField DataField="UnitsInStock"
          HeaderText="UnitsInStock" SortExpression="Units In Stock" />
        <asp:BoundField DataField="UnitsOnOrder"
          HeaderText="UnitsOnOrder" SortExpression="Units On Order" />
        <asp:BoundField DataField="ReorderLevel"
          HeaderText="ReorderLevel" SortExpression="Reorder Level" />
        <asp:CheckBoxField DataField="Discontinued"
          HeaderText="Discontinued" SortExpression="Discontinued" />
    </Fields>
</asp:DetailsView>
```

Take a moment to try out the `MasterDetailsDetails.aspx` page in a browser. At first glance it may appear that everything is working as desired, but there's a subtle problem. When you choose a new category the `ProductsByCategory` DropDownList is updated to include those products for the selected category, but the `ProductDetails` DetailsView continued to show the previous product information. The DetailsView is updated when choosing a different product for the selected category. Furthermore, if you test thoroughly enough, you'll find that if you continually choose new categories (such as choosing Beverages from the `Categories` DropDownList, then Condiments, then Confections) every other category selection causes the `ProductDetails` DetailsView to be refreshed.

To help concretize this problem, let's look at a specific example. When you first visit the page the Beverages category is selected and the related products are loaded in the `ProductsByCategory` DropDownList. Chai is the selected product and its details are displayed in the `ProductDetails` DetailsView, as shown in Figure 18.

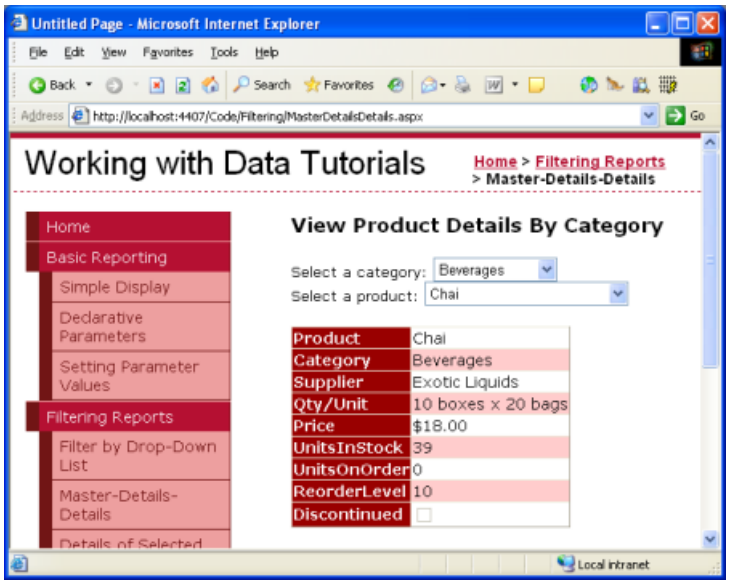


Figure 18: The Selected Product's Details are Displayed in a DetailsView (Click to view full-size image)

If you change the category selection from Beverages to Condiments, a postback occurs and the `ProductsByCategory` DropDownList is updated accordingly, but the DetailsView still displays details for Chai.

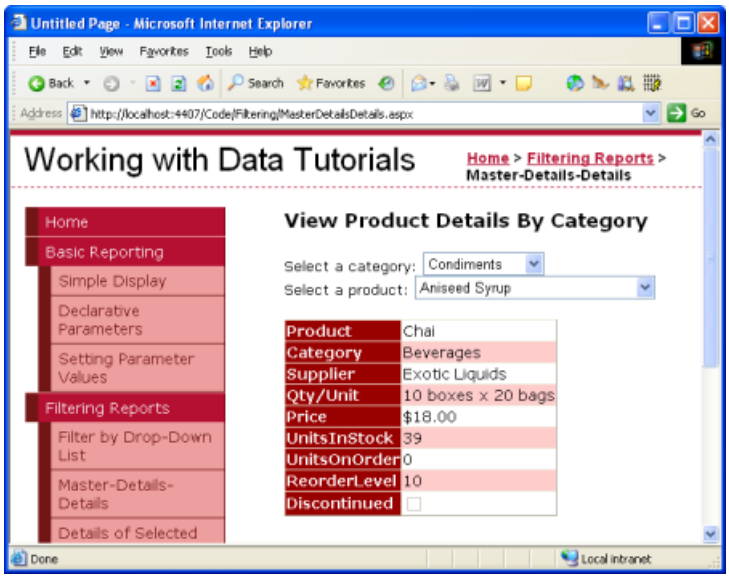


Figure 19: The Previously Selected Product's Details are Still Displayed (Click to view full-size image)

Picking a new product from the list refreshes the DetailsView as expected. If you pick a new category after changing the product, the DetailsView again won't refresh. However, if instead of choosing a new product you selected a new category, the DetailsView would refresh. What in the world is going on here?

The problem is a timing issue in the page's lifecycle. Whenever a page is requested it proceeds through a number of steps as its rendering. In one of these steps the ObjectDataSource controls check to see if any of their `SelectParameters` values have changed. If so, the data Web control bound to the ObjectDataSource knows that it needs to refresh its display. For example, when a new category is selected, the `ProductsByCategoryDataSource` ObjectDataSource detects that its parameter values have changed and the `ProductsByCategory` DropDownList rebinds itself, getting the products for the selected category.

The problem that arises in this situation is that the point in the page lifecycle that the ObjectDataSources check for changed parameters occurs *before* the rebinding of the associated data Web controls. Therefore, when selecting a new category the `ProductsByCategoryDataSource` ObjectDataSource detects a change in its parameter's value. The ObjectDataSource used by the `ProductDetails` DetailsView, however, doesn't note any such changes because the `ProductsByCategory` DropDownList has yet to be rebound. Later in the lifecycle the `ProductsByCategory` DropDownList rebinds to its ObjectDataSource, grabbing the products for the newly selected

category. While the `ProductsByCategory` DropDownList's value has changed, the `ProductDetails` DetailsView's ObjectDataSource has already done its parameter value check; therefore, the DetailsView displays its previous results. This interaction is depicted in Figure 20.

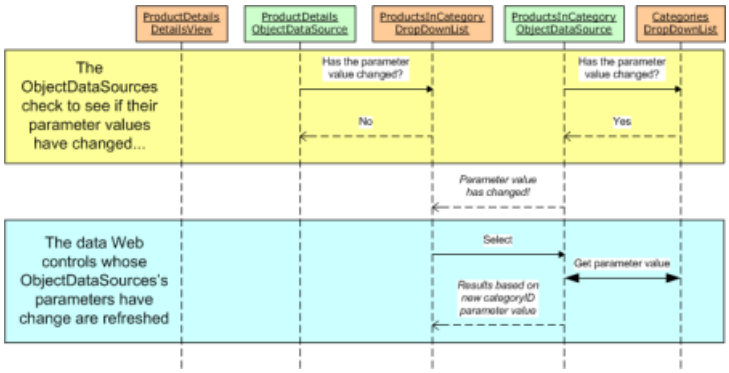


Figure 20: The `ProductsByCategory` DropDownList Value Changes After the `ProductDetails` DetailsView's ObjectDataSource Checks for Changes (Click to view full-size image)

To remedy this we need to explicitly rebind the `ProductDetails` DetailsView after the `ProductsByCategory` DropDownList has been bound. We can accomplish this by calling the `ProductDetails` DetailsView's `DataBind()` method when the `ProductsByCategory` DropDownList's `DataBound` event fires. Add the following event handler code to the `MasterDetailsDetails.aspx` page's code-behind class (refer to the "Programmatically Setting the ObjectDataSource's Parameter Values" for a discussion on how to add an event handler):

```
protected void ProductsByCategory_DataBound(object sender, EventArgs e)
{
    ProductDetails.DataBind();
}
```

After this explicit call to the `ProductDetails` DetailsView's `DataBind()` method has been added, the tutorial works as expected. Figure 21 highlights how this changed remedied our earlier problem.

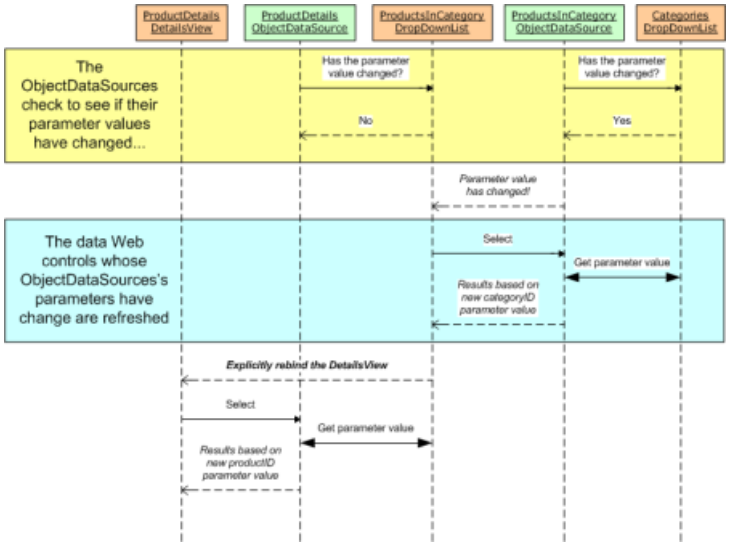


Figure 21: The `ProductDetails` DetailsView is Explicitly Refreshed When the `ProductsByCategory` DropDownList's `DataBound` Event Fires (Click to view full-size image)

Summary

The DropDownList serves as an ideal user interface element for master/detail reports where there is a one-to-many relationship between the master and detail records. In the preceding tutorial we saw how to use a single DropDownList to filter the products displayed by the selected category. In this tutorial we replaced the GridView

of products with a DropDownList, and used a DetailsView to display the details of the selected product. The concepts discussed in this tutorial can easily be extended to data models involving multiple one-to-many relationships, such as customers, orders, and order items. In general, you can always add a DropDownList for each of the "one" entities in the one-to-many relationships.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Master/Detail Filtering Across Two Pages (C#)

9/16/2020 • 11 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll implement this pattern by using a GridView to list the suppliers in the database. Each supplier row in the GridView will contain a View Products link that, when clicked, will take the user to a separate page that lists those products for the selected supplier.

Introduction

In the previous two tutorials we saw how to display master/detail reports in a single web page using DropDownLists to display the "master" records and a GridView or DetailsView control to display the "details." Another common pattern used for master/details reports is to have the master records on one web page and the details shown on another. A forum website, like the ASP.NET Forums, is a great example of this pattern in practice. The ASP.NET Forums are composed of a variety of forums Getting Started, Web Forms, Data Presentation Controls, and so on. Each forum is composed of many threads and each thread is composed of a number of posts. On the ASP.NET Forums homepage, the forums are listed. Clicking on a forum whisks you to a `ShowForum.aspx` page, which lists the threads for that forum. Likewise, clicking on a thread takes you to `ShowPost.aspx`, which displays the posts for the thread that was clicked.

In this tutorial we'll implement this pattern by using a GridView to list the suppliers in the database. Each supplier row in the GridView will contain a View Products link that, when clicked, will take the user to a separate page that lists those products for the selected supplier.

Step 1: Adding `SupplierListMaster.aspx` and `ProductsForSupplierDetails.aspx` Pages to the `Filtering` Folder

When defining the page layout in the third tutorial we added a number of "starter" pages in the `BasicReporting`, `Filtering`, and `CustomFormatting` folders. However, we did not add a starter page for this tutorial at that time, so take a moment to add two new pages to the `Filtering` folder: `SupplierListMaster.aspx` and `ProductsForSupplierDetails.aspx`. `SupplierListMaster.aspx` will list the "master" records (the suppliers) while `ProductsForSupplierDetails.aspx` will display the products for the selected supplier.

When creating these two new pages be certain to associate them with the `Site.master` master page.

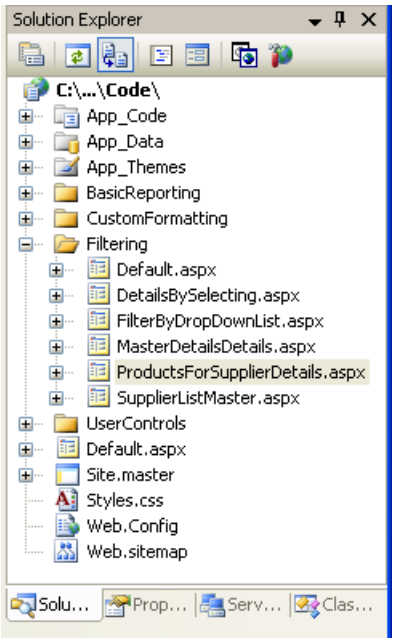


Figure 1: Add the `SupplierListMaster.aspx` and `ProductsForSupplierDetails.aspx` Pages to the `Filtering` Folder

Also, when adding new pages to the project, be sure to update the site map file, `Web.sitemap`, accordingly. For this tutorial simply add the `SupplierListMaster.aspx` page to the site map using the following XML content as a child of the Filtering Reports `<siteMapNode>` element:

```
<siteMapNode url="~/Filtering/SupplierListMaster.aspx"
  title="Master/Detail Across Two Pages"
  description="Master records on one page, detail records on another."
/>
```

NOTE

You can help automate the process of updating the site map file when adding new ASP.NET pages using K. Scott Allen's free Visual Studio Site Map Macro.

Step 2: Displaying the Supplier List in `SupplierListMaster.aspx`

With the `SupplierListMaster.aspx` and `ProductsForSupplierDetails.aspx` pages created, our next step is to create the GridView of suppliers in `SupplierListMaster.aspx`. Add a GridView to the page and bind it to a new ObjectDataSource. This ObjectDataSource should use the `SuppliersBLL` class's `GetSuppliers()` method to return all suppliers.

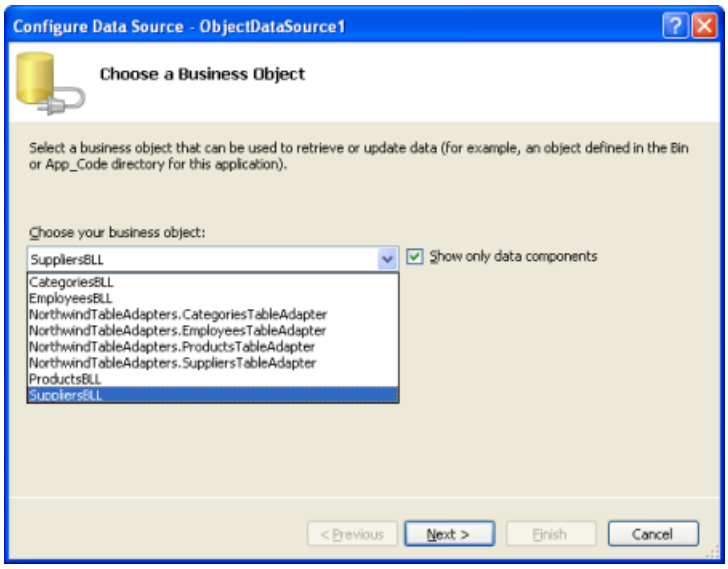


Figure 2: Select the `SuppliersBLL` Class (Click to view full-size image)

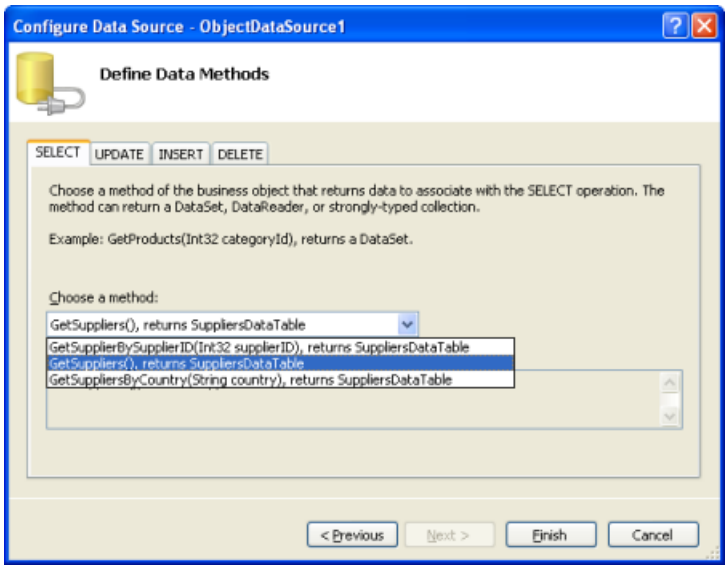


Figure 3: Configure the ObjectDataSource to Use the `GetSuppliers()` Method (Click to view full-size image)

We need to include a link titled View Products in each GridView row that, when clicked, takes the user to `ProductsForSupplierDetails.aspx` passing in the selected row's `SupplierID` value through the querystring. For example, if user clicks on the View Products link for the Tokyo Traders supplier (which has a `SupplierID` value of 4), they should be sent to `ProductsForSupplierDetails.aspx?SupplierID=4`.

To accomplish this, add a HyperLinkField to the GridView, which adds a hyperlink to each GridView row. Start by clicking the Edit Columns link from the GridView's smart tag. Next, select the HyperLinkField from the list in the upper left and click Add to include the HyperLinkField in the GridView's field list.

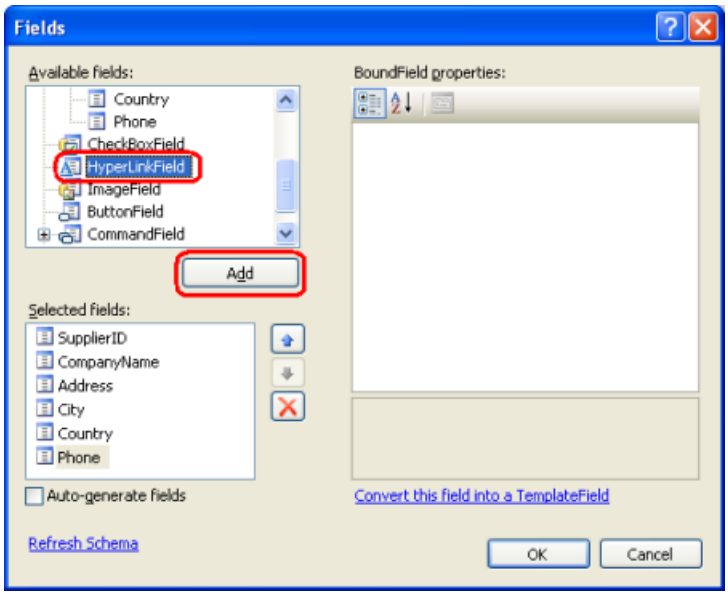


Figure 4: Add a HyperLinkField to the GridView (Click to view full-size image)

The HyperLinkField can be configured to use the same text or URL values the link in each GridView row, or can base these values on the data values bound to each particular row. To specify a static value across all rows use the HyperLinkField's `Text` or `NavigateUrl` properties. Since we want the link text to be the same for all rows, set the HyperLinkField's `Text` property to View Products.

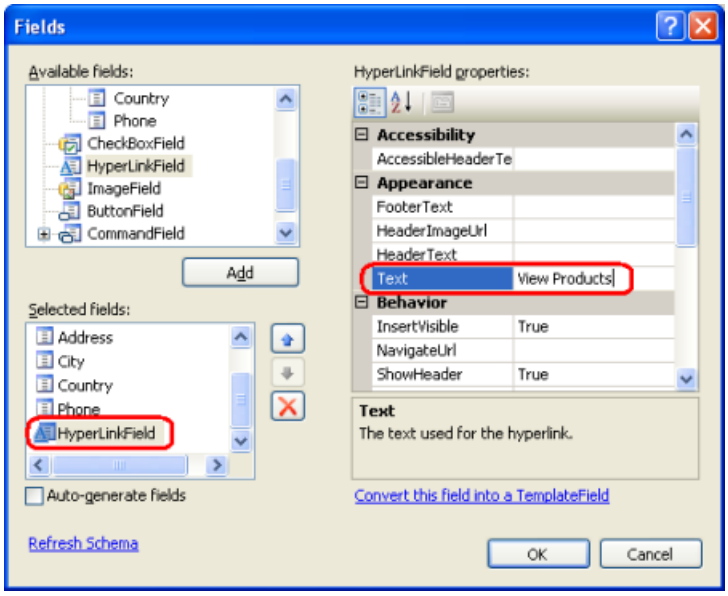


Figure 5: Set the HyperLinkField's `Text` Property to View Products (Click to view full-size image)

To set the text or URL values to be based on the underlying data bound to the GridView row, specify the data fields the text or URL values should be pulled from in the `DataTextField` or `DataNavigateUrlFields` properties. `DataTextField` can only be set to a single data field; `DataNavigateUrlFields`, however, can be set to a comma-delimited list of data fields. We frequently need to base the text or URL on a combination of the current row's data field value and some static markup. In this tutorial, for example, we want the URL of the HyperLinkField's links to be `ProductsForSupplierDetails.aspx?SupplierID=supplierID`, where *`supplierID`* is each GridView's row's `SupplierID` value. Notice that we need both static and data-driven values here: the `ProductsForSupplierDetails.aspx?SupplierID=` portion of the link's URL is static, whereas the *`supplierID`* portion is data-driven as its value is each row's own `SupplierID` value.

To indicate a combination of static and data-driven values, use the `DataTextFormatString` and `DataNavigateUrlFormatString` properties. In these properties enter the static markup as needed and then use the marker `{0}` where you want the value of the field specified in the `DataTextField` or `DataNavigateUrlFields`

properties to appear. If the `DataNavigateUrlFields` property has multiple fields specified use `{0}` where you want the first field value inserted, `{1}` for the second field value, and so on.

Applying this to our tutorial, we need to set the `DataNavigateUrlFields` property to `SupplierID`, since that's the data field whose value we need to customize on a per-row basis, and the `DataNavigateUrlFormatString` property to `ProductsForSupplierDetails.aspx?SupplierID={0}`.

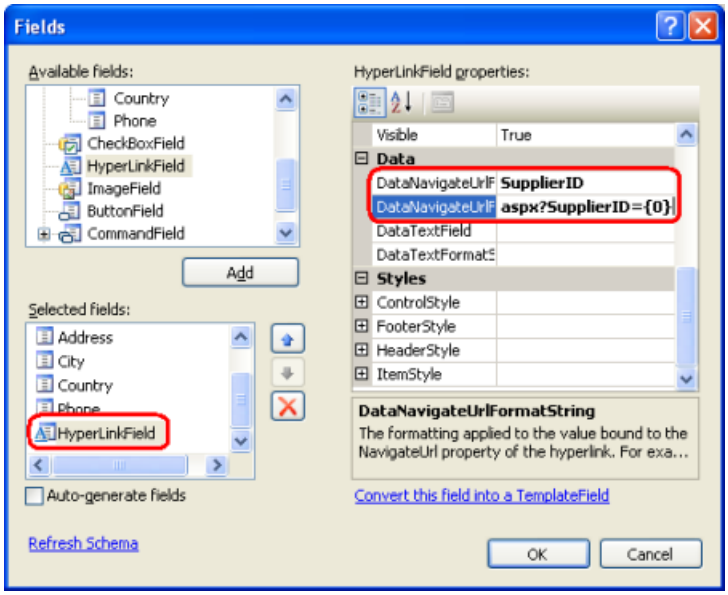


Figure 6: Configure the HyperLinkField to Include the Proper Link URL Based Upon the `SupplierID` (Click to view full-size image)

After adding the HyperLinkField, feel free to customize and reorder the GridView's fields. The following markup shows the GridView after I've made some minor field-level customizations.

```
<asp:GridView ID="GridView1" runat="server"
    AutoGenerateColumns="False" DataKeyNames="SupplierID"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <Columns>
        <asp:HyperLinkField DataNavigateUrlFields="SupplierID"
            DataNavigateUrlFormatString=
            "ProductsForSupplierDetails.aspx?SupplierID={0}"
            Text="View Products" />
        <asp:BoundField DataField="CompanyName"
          HeaderText="Company" SortExpression="CompanyName" />
        <asp:BoundField DataField="City" HeaderText="City"
          SortExpression="City" />
        <asp:BoundField DataField="Country" HeaderText="Country"
          SortExpression="Country" />
    </Columns>
</asp:GridView>
```

Take a moment to view the `SupplierListMaster.aspx` page through a browser. As Figure 7 shows, the page currently lists all of the suppliers including a View Products link. Clicking on the View Products link will take you to `ProductsForSupplierDetails.aspx`, passing along the supplier's `SupplierID` in the querystring.

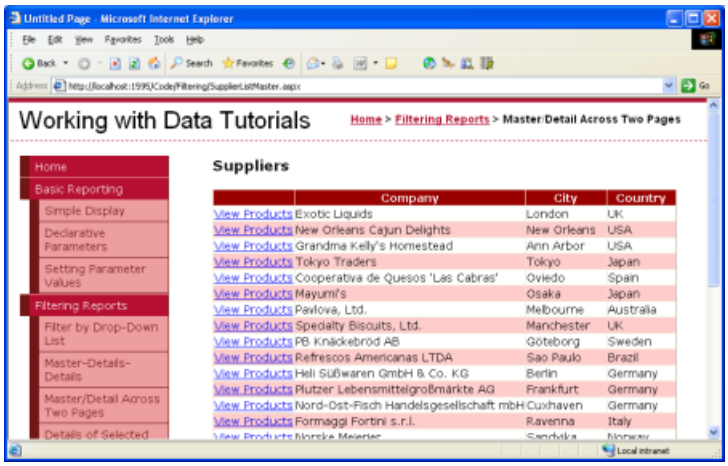


Figure 7: Each Supplier Row Contains a View Products Link (Click to view full-size image)

Step 3: Listing the Supplier's Products in `ProductsForSupplierDetails.aspx`

At this point the `SupplierListMaster.aspx` page is sending users to `ProductsForSupplierDetails.aspx`, passing the selected supplier's `SupplierID` in the querystring. The tutorial's final step is to display the products in a GridView in `ProductsForSupplierDetails.aspx` whose `SupplierID` equals the `SupplierID` passed in through the querystring. To accomplish this start by adding a GridView to the `ProductsForSupplierDetails.aspx` page, using a new ObjectDataSource control named `ProductsBySupplierDataSource` that invokes the `GetProductsBySupplierID(supplierID)` method from the `ProductsBLL` class.

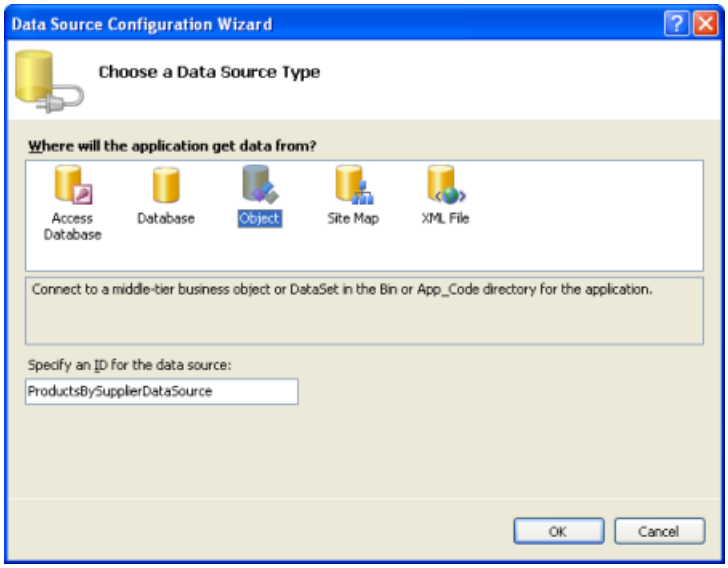


Figure 8: Add a New ObjectDataSource Named `ProductsBySupplierDataSource` (Click to view full-size image)

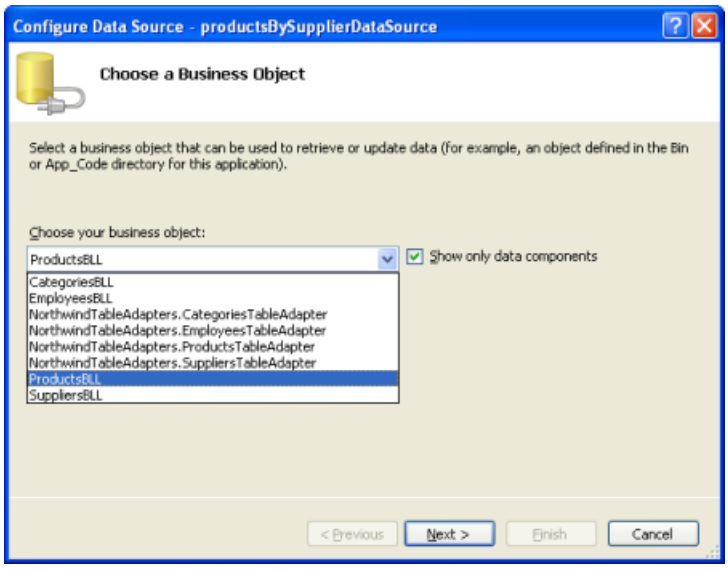


Figure 9: Select the `ProductsBLL` Class (Click to view full-size image)

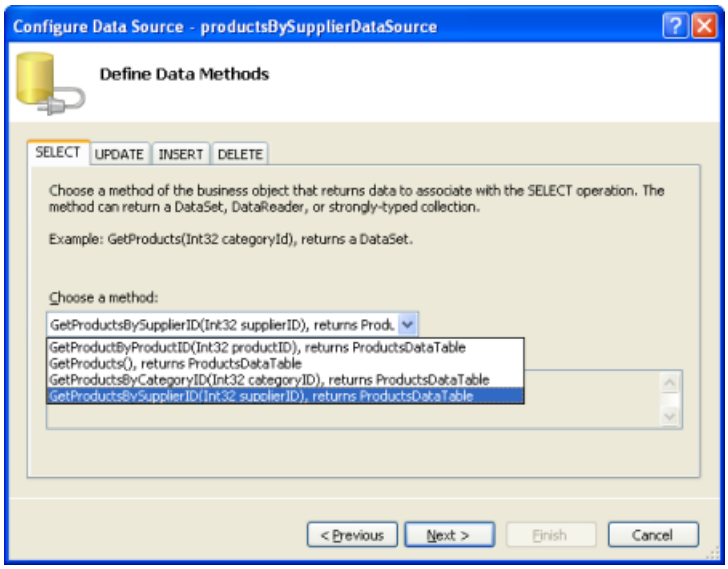


Figure 10: Have the ObjectDataSource Invoke the `GetProductsBySupplierID(supplierID)` Method (Click to view full-size image)

The final step of the Configure Data Source wizard asks us to provide the source of the `GetProductsBySupplierID(supplierID)` method's *`supplierID`* parameter. To use the querystring value, set the Parameter source to QueryString and enter the name of the querystring value to use in the QueryStringField textbox (`SupplierID`).

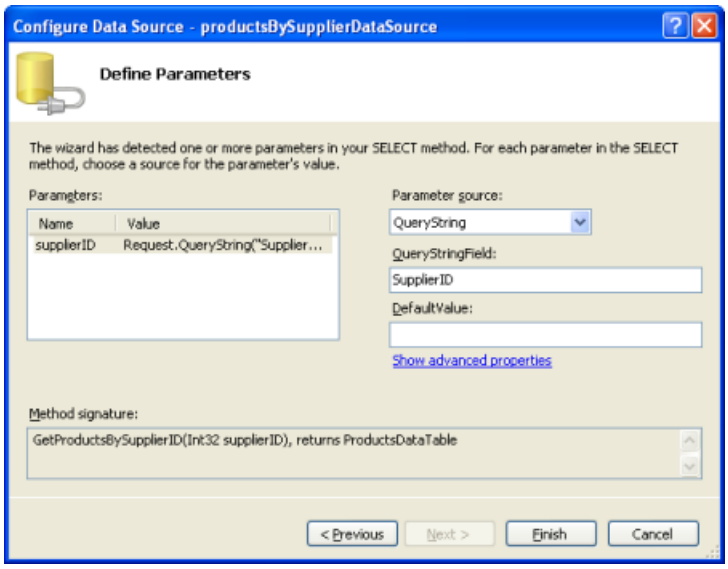


Figure 11: Populate the `supplierID` Parameter Value from the `SupplierID` Querystring Value (Click to view full-size image)

That's all there is to it! Figure 12 shows the `ProductsForSupplierDetails.aspx` page when visited by clicking the Tokyo Traders link from `SupplierListMaster.aspx`.

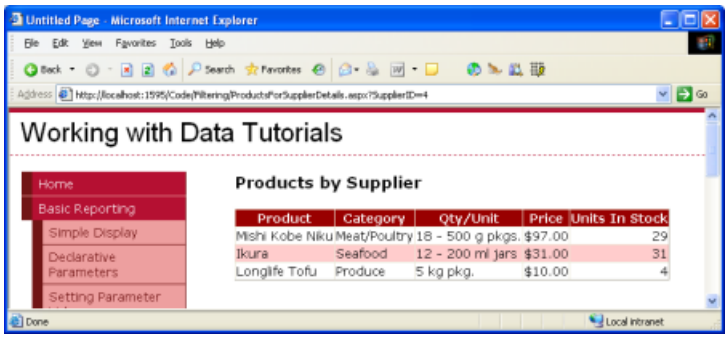


Figure 12: The Products Supplied by Tokyo Traders are Shown (Click to view full-size image)

Displaying Supplier Information in `ProductsForSupplierDetails.aspx`

As Figure 12 shows, the `ProductsForSupplierDetails.aspx` page simply lists the products that are supplied by the `SupplierID` specified in the querystring. Someone sent directly to this page, however, would not know that Figure 12 is showing Tokyo Traders' products. To remedy this we can display supplier information in this page as well.

Start by adding a FormView above the products GridView. Create a new ObjectDataSource control named `SuppliersDataSource` that invokes the `SuppliersBLL` class's `GetSupplierBySupplierID(supplierID)` method.

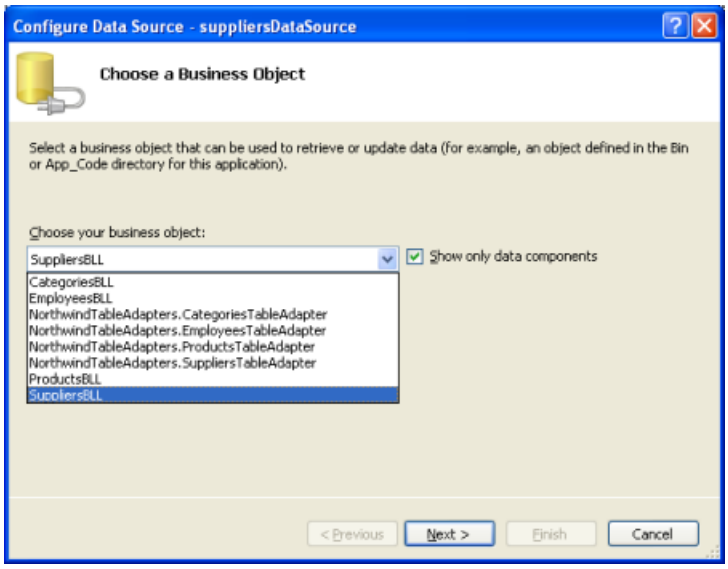


Figure 13: Select the `SuppliersBLL` Class (Click to view full-size image)

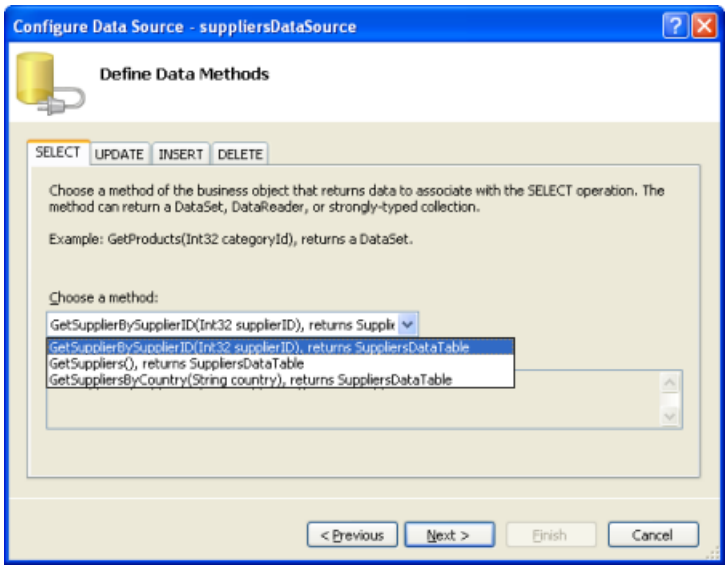


Figure 14: Have the ObjectDataSource Invoke the `GetSupplierBySupplierID(supplierID)` Method (Click to view full-size image)

As with the `ProductsBySupplierDataSource`, have the *`supplierID`* parameter assigned the value of the `SupplierID` querystring value.

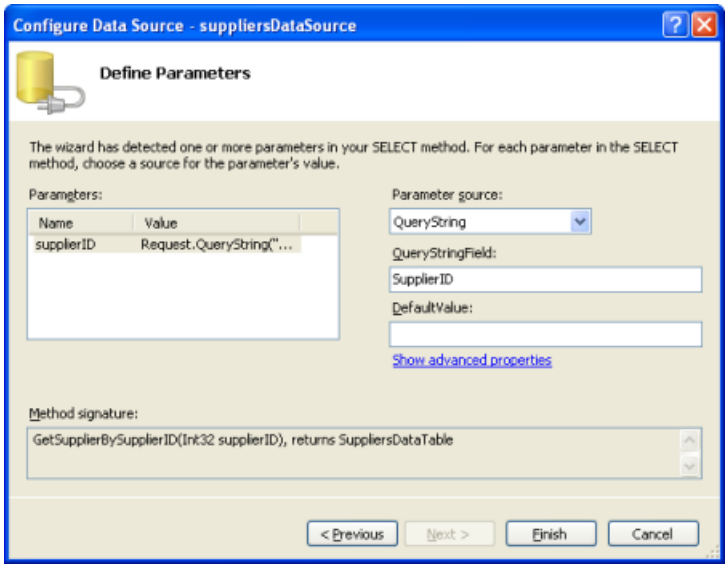


Figure 15: Populate the *`supplierID`* Parameter Value from the `SupplierID` Querystring Value (Click to view full-

size image)

When binding the FormView to the ObjectDataSource in the Design view, Visual Studio will automatically create the FormView's `ItemTemplate`, `InsertItemTemplate`, and `EditItemTemplate` with Label and TextBox Web controls for each of the data fields returned by the ObjectDataSource. Since we just want to display supplier information feel free to remove the `InsertItemTemplate` and `EditItemTemplate`. Next, edit the ItemTemplate so that it displays the supplier's company name in an `<h3>` element and the address, city, country, and phone number beneath the company name. Alternatively, you can manually set the FormView's `DataSourceID` and create the `ItemTemplate` markup, as we did back in the "Displaying Data With the ObjectDataSource" tutorial.

After these edits the FormView's declarative markup should look similar to the following:

```
<asp:FormView ID="FormView1" runat="server" DataKeyNames="SupplierID"
    DataSourceID="suppliersDataSource" EnableViewState="False">
    <ItemTemplate>
        <h3><%# Eval("CompanyName") %></h3>
        <p>
            <asp:Label ID="AddressLabel" runat="server"
                Text='<%# Bind("Address") %>'></asp:Label><br />
            <asp:Label ID="CityLabel" runat="server"
                Text='<%# Bind("City") %>'></asp:Label>,
            <asp:Label ID="CountryLabel" runat="server"
                Text='<%# Bind("Country") %>'></asp:Label><br />
            Phone:
            <asp:Label ID="PhoneLabel" runat="server"
                Text='<%# Bind("Phone") %>'></asp:Label>
        </p>
    </ItemTemplate>
</asp:FormView>
```

Figure 16 shows a screen shot of the `ProductsForSupplierDetails.aspx` page after the supplier information detailed above has been included.

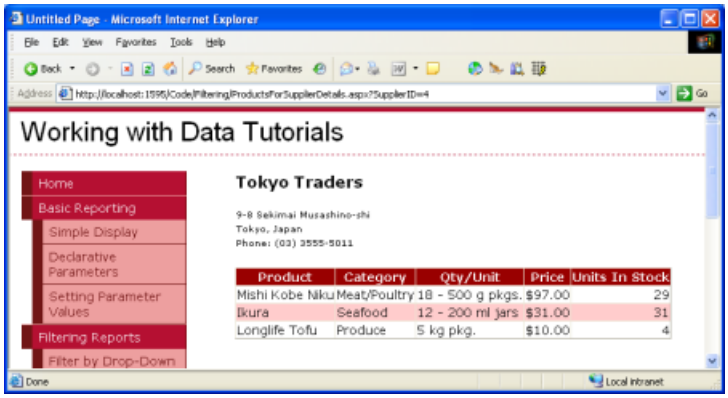


Figure 16: The List of Products Includes a Summary About the Supplier (Click to view full-size image)

Applying the Final Touches for the `ProductsForSupplierDetails.aspx` UI

To improve the user experience for this report there are a couple of additions we ought to make to the `ProductsForSupplierDetails.aspx` page. Currently the only way a user can go from the `ProductsForSupplierDetails.aspx` page back to the list of suppliers is to click their browser's back button. Let's add a HyperLink control to the `ProductsForSupplierDetails.aspx` page that links back to `SupplierListMaster.aspx`, providing another way for the user to return to the master list.

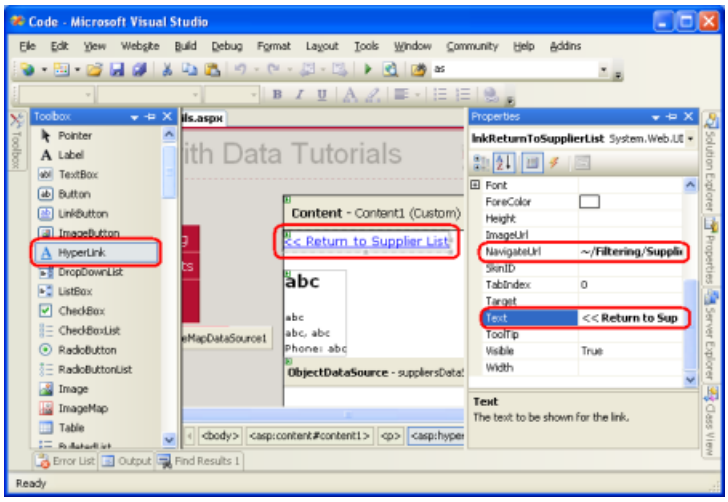


Figure 17: Add a HyperLink Control to Take the User Back to `SupplierListMaster.aspx` (Click to view full-size image)

If the user clicks on the View Products link for a supplier that doesn't have any products, the `ProductsBySupplierDataSource` ObjectDataSource in `ProductsForSupplierDetails.aspx` won't return any results. The GridView bound to the ObjectDataSource won't render any markup resulting in a blank region on the page in the user's browser. To more clearly communicate to the user that there are no products associated with the selected supplier we can set the GridView's `EmptyDataText` property to the message we want displayed when such a situation arises. I've set this property to "There are no products provided by this supplier'"

By default, all suppliers in the Northwinds database provide at least one product. However, for this tutorial I've manually modified the `Products` table so that the supplier Escargots Nouveaux is no longer associated with any products. Figure 18 shows the details page for Escargots Nouveaux after this change has been made.

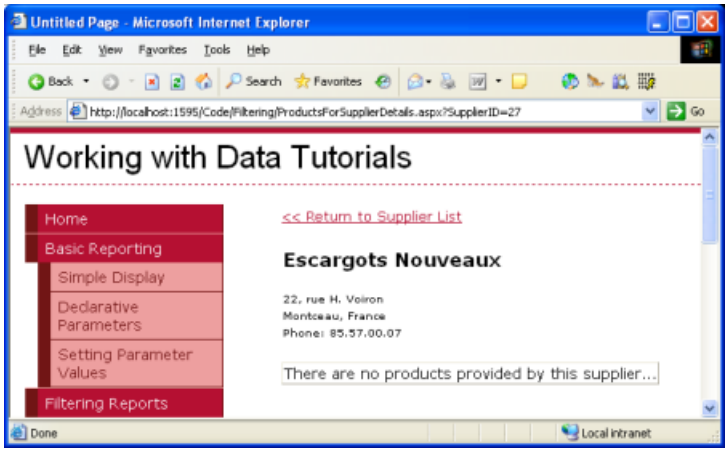


Figure 18: Users are Informed that the Supplier Doesn't Provide Any Products (Click to view full-size image)

Summary

While master/detail reports can display both the master and detail records on a single page, in many websites they are separated out across two web pages. In this tutorial we looked at how to implement such a master/detail report by having the suppliers listed in a GridView in the "master" web page and the associated products listed in the "details" page. Each supplier row in the master web page contained a link to the details page that passed along the row's `SupplierID` value. Such row-specific links can be easily added using the GridView's HyperLinkField.

In the details page retrieving those products for the specified supplier was accomplished by invoking the `ProductsBLL` class's `GetProductsBySupplierID(supplierID)` method. The *`supplierID`* parameter value was specified declaratively using the querystring as the parameter source. We also looked at how to display the supplier details in the details page using a FormView.

Our next tutorial is the final one on master/detail reports. We'll look at how to display a list of products in a GridView where each row has a Select button. Clicking on the Select button will display that product's details in a DetailsView control on the same page.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Master/Detail Using a Selectable Master GridView with a Details DetailView (C#)

2/20/2020 • 7 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> This tutorial will have a GridView whose rows include the name and price of each product along with a Select button. Clicking the Select button for a particular product will cause its full details to be displayed in a DetailsView control on the same page.

Introduction

In the previous tutorial we saw how to create a master/detail report using two web pages: a "master" web page, from which we displayed the list of suppliers; and a "details" web page that listed those products provided by the selected supplier. This two page report format can be condensed into one page. This tutorial will have a GridView whose rows include the name and price of each product along with a Select button. Clicking the Select button for a particular product will cause its full details to be displayed in a DetailsView control on the same page.

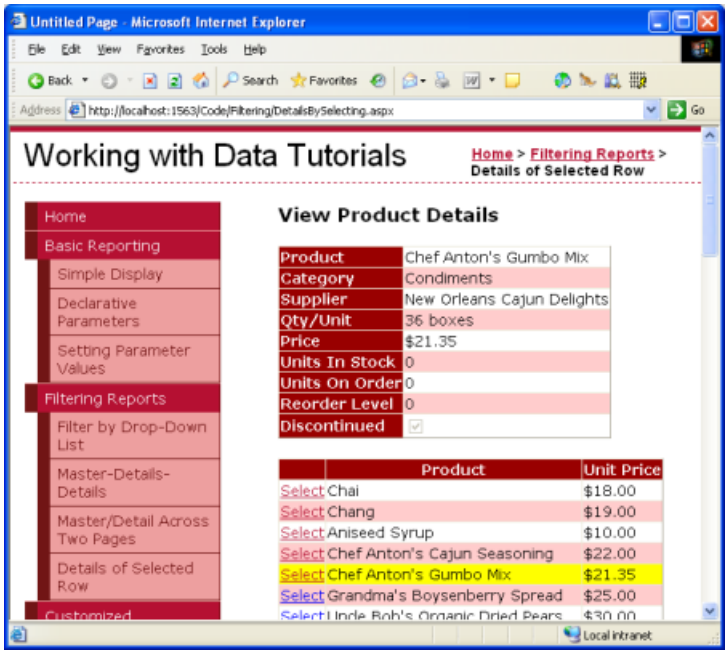


Figure 1: Clicking the Select Button Displays the Product's Details (Click to view full-size image)

Step 1: Creating a Selectable GridView

Recall that in the two-page master/detail report that each master record included a hyperlink that, when clicked, sent the user to the details page passing the clicked row's `SupplierID` value in the querystring. Such a hyperlink was added to each GridView row using a HyperLinkField. For the single page master/details report, we will need a Button for each GridView row that, when clicked, shows the details. The GridView control can be configured to include a Select button for each row that causes a postback and marks that row as the GridView's SelectedRow.

Start by adding a GridView control to the `DetailsBySelecting.aspx` page in the `Filtering` folder, setting its `ID` property to `ProductsGrid`. Next, add a new ObjectDataSource named `AllProductsDataSource` that invokes the

`ProductsBLL` class's `GetProducts()` method.

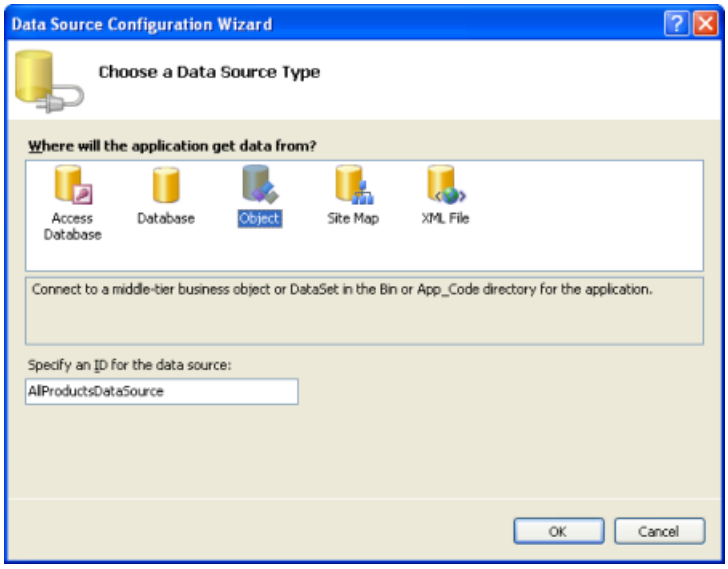


Figure 2: Create an ObjectDataSource Named `AllProductsDataSource` (Click to view full-size image)

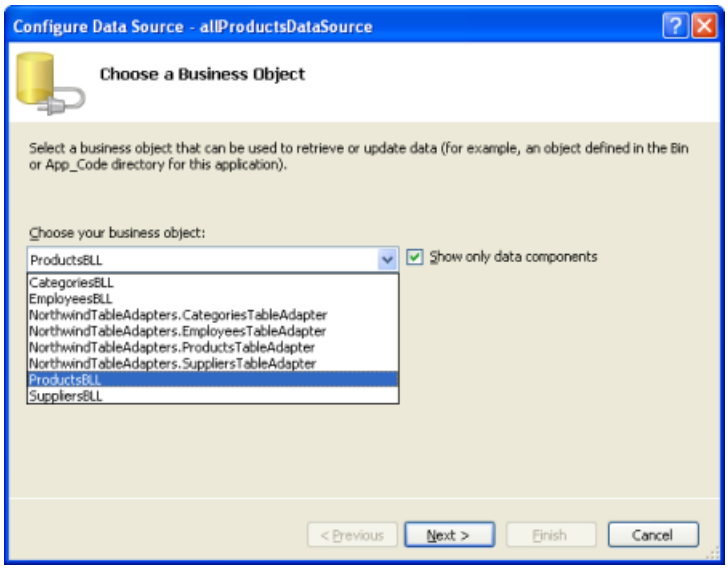


Figure 3: Use the `ProductsBLL` Class (Click to view full-size image)

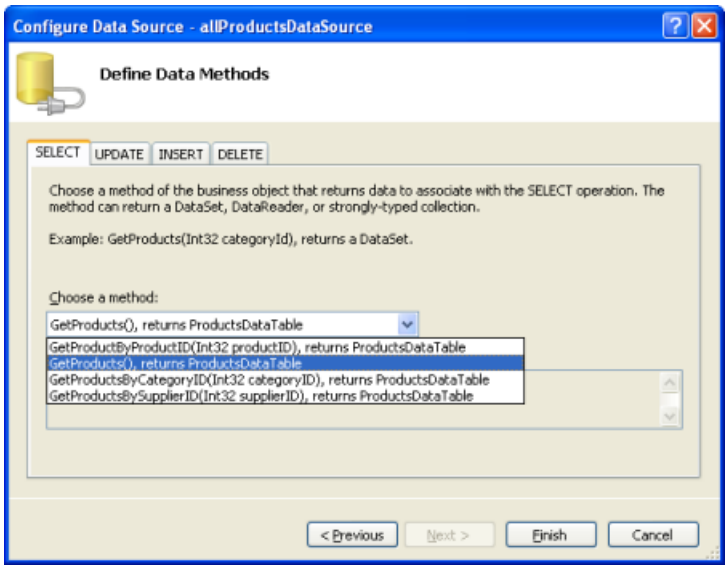


Figure 4: Configure the ObjectDataSource to Invoke the `GetProducts()` Method (Click to view full-size image)

Edit the GridView's fields removing all but the `ProductName` and `UnitPrice` BoundFields. Also, feel free to

customize these BoundFields as needed, such as formatting the `UnitPrice` BoundField as a currency and changing the `HeaderText` properties of the BoundFields. These steps can be accomplished graphically, by clicking the Edit Columns link from the GridView's smart tag, or by manually configuring the declarative syntax.

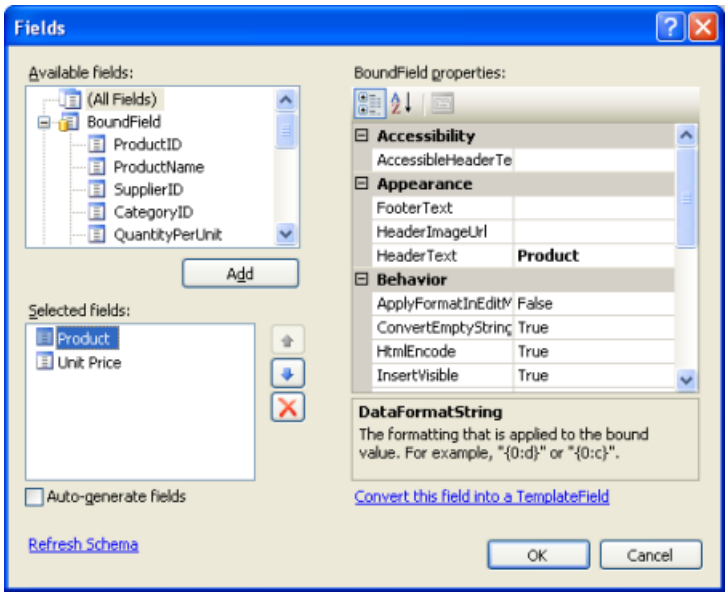


Figure 5: Remove All But the `ProductName` and `UnitPrice` BoundFields (Click to view full-size image)

The final markup for the GridView is:

```
<asp:GridView ID="ProductsGrid" runat="server"
  AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="AllProductsDataSource" EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductName"
          HeaderText="Product" SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice"
          DataFormatString="{0:c}" HeaderText="Unit Price"
            HtmlEncode="False" SortExpression="UnitPrice" />
    </Columns>
</asp:GridView>
```

Next, we need to mark the GridView as selectable, which will add a Select button to each row. To accomplish this, simply check the Enable Selection checkbox in the GridView's smart tag.

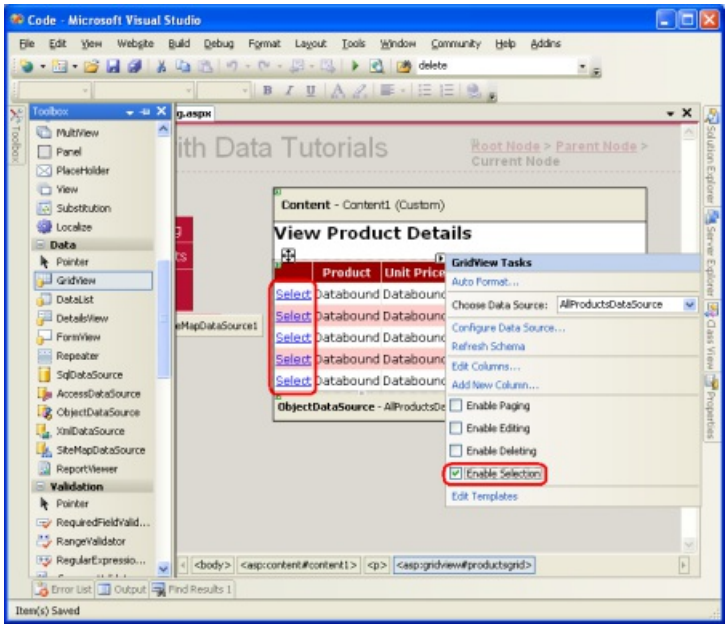


Figure 6: Make the GridView's Rows Selectable (Click to view full-size image)

Checking the Enabling Selection option adds a CommandField to the `ProductsGrid` GridView with its `ShowSelectButton` property set to True. This results in a Select button for each row of the GridView, as Figure 6 illustrates. By default, the Select buttons are rendered as LinkButtons, but you can use Buttons or ImageButtons instead through the CommandField's `ButtonType` property.

```
<asp:GridView ID="ProductsGrid" runat="server"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="AllProductsDataSource" EnableViewState="False">
    <Columns>
        <asp:CommandField ShowSelectButton="True" />
        <asp:BoundField DataField="ProductName"
         HeaderText="Product" SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice"
          DataFormatString="{0:c}" HeaderText="Unit Price"
          HtmlEncode="False" SortExpression="UnitPrice" />
    </Columns>
</asp:GridView>
```

When a GridView row's Select button is clicked a postback ensues and the GridView's `SelectedRow` property is updated. In addition to the `SelectedRow` property, the GridView provides the SelectedIndex, SelectedValue, and SelectedDataKey properties. The `SelectedIndex` property returns the index of the selected row, whereas the `SelectedValue` and `SelectedDataKey` properties return values based upon the GridView's DataKeyNames property.

The `DataKeyNames` property is used to associate one or more data field values with each row and is commonly used to attribute uniquely identifying information from the underlying data with each GridView row. The `SelectedValue` property returns the value of the first `DataKeyNames` data field for the selected row where as the `SelectedDataKey` property returns the selected row's `DataKey` object, which contains all of the values for the specified data key fields for that row.

The `DataKeyNames` property is automatically set to the uniquely-identifying data field(s) when you bind a data source to a GridView, DetailsView, or FormView through the Designer. While this property has been set for us automatically in the preceding tutorials, the examples would have worked without the `DataKeyNames` property specified. However, for the selectable GridView in this tutorial, as well as for future tutorials in which we'll be examining inserting, updating, and deleting, the `DataKeyNames` property must be set properly. Take a moment to ensure that your GridView's `DataKeyNames` property is set to `ProductID`.

Let's view our progress thus far through a browser. Note that the GridView lists the name and price for all of the products along with a Select LinkButton. Clicking the Select button causes a postback. In Step 2 we'll see how to have a DetailsView respond to this postback by displaying the details for the selected product.

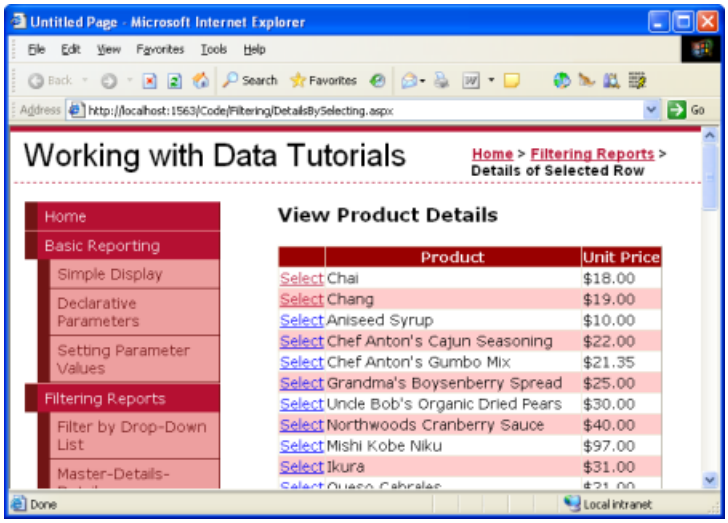


Figure 7: Each Product Row Contains a Select LinkButton (Click to view full-size image)

Highlighting the Selected Row

The `ProductsGrid` GridView has a `SelectedRowStyle` property that can be used to dictate the visual style for the selected row. Used properly, this can improve the user's experience by more clearly showing which row of the GridView is currently selected. For this tutorial, let's have the selected row be highlighted with a yellow background.

As with our earlier tutorials, let's strive to keep the aesthetic-related settings defined as CSS classes. Therefore, create a new CSS class in `Styles.css` named `SelectedRowStyle`.

```
.SelectedRowStyle
{
    background-color: Yellow;
}
```

To apply this CSS class to the `SelectedRowStyle` property of *all* GridViews in our tutorial series, edit the `GridView.skin` Skin in the `DataWebControls` Theme to include the `SelectedRowStyle` settings as shown below:

```
<asp:GridView runat="server" CssClass="DataWebControlStyle">
   <AlternatingRowStyle CssClass="AlternatingRowStyle" />
   <RowStyle CssClass="RowStyle" />
   <HeaderStyle CssClass="HeaderStyle" />
   <SelectedRowStyle CssClass="SelectedRowStyle" />
</asp:GridView>
```

With this addition, the selected GridView row is now highlighted with a yellow background color.

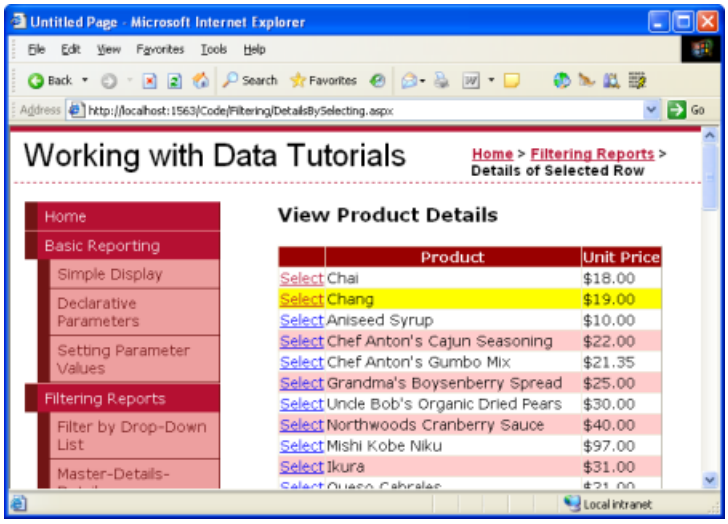


Figure 8: Customize the Selected Row's Appearance Using the GridView's `SelectedRowStyle` Property (Click to view full-size image)

Step 2: Displaying the Selected Product's Details in a DetailsView

With the `ProductsGrid` GridView complete, all that remains is to add a DetailsView that displays information about the particular product selected. Add a DetailsView control above the GridView and create a new ObjectDataSource named `ProductDetailsDataSource`. Since we want this DetailsView to display particular information about the selected product, configure the `ProductDetailsDataSource` to use the `ProductsBLL` class's `GetProductByProductID(productID)` method.

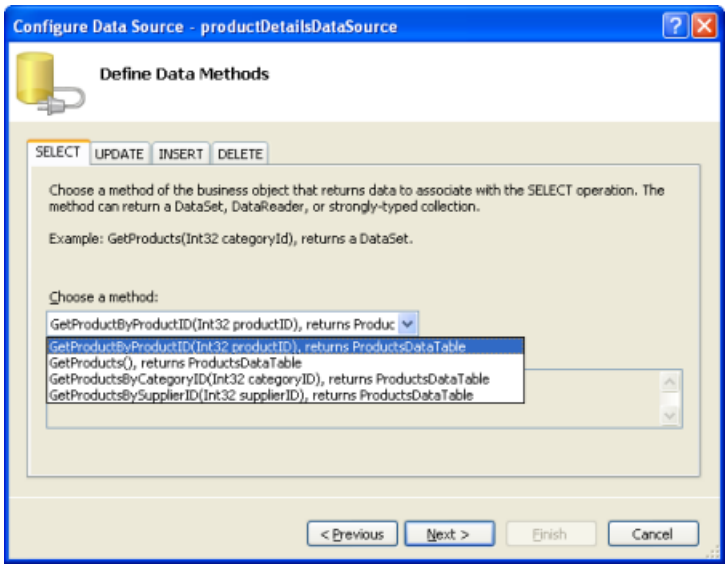


Figure 9: Invoke the `ProductsBLL` Class's `GetProductByProductID(productID)` Method (Click to view full-size image)

Have the *`productID`* parameter's value obtained from the GridView control's `SelectedValue` property. As we discussed earlier, the GridView's `SelectedValue` property returns the first data key value for the selected row. Therefore, it's imperative that the GridView's `DataKeyNames` property is set to `ProductID`, so that the selected row's `ProductID` value is returned by `SelectedValue`.

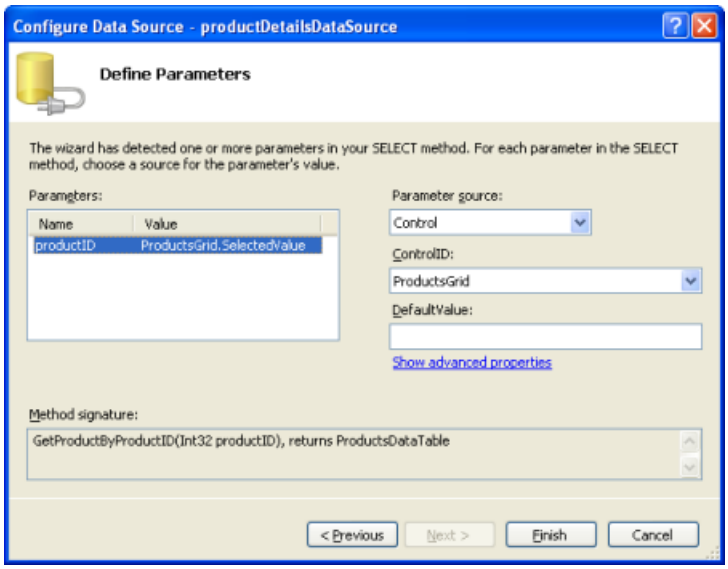


Figure 10: Set the *`productID`* Parameter to the GridView's `SelectedValue` Property (Click to view full-size image)

Once the `productDetailsDataSource` ObjectDataSource has been configured correctly and bound to the DetailsView, this tutorial is complete! When the page is first visited no row is selected, so the GridView's `SelectedValue` property returns `null`. Since there are no products with a `NULL` `ProductID` value, no records are returned by the `GetProductByProductID(productID)` method, meaning that the DetailsView isn't displayed (see Figure 11). Upon clicking a GridView row's Select button, a postback ensues and the DetailsView is refreshed. This time the GridView's `SelectedValue` property returns the `ProductID` of the selected row, the `GetProductByProductID(productID)` method returns a `ProductsDataTable` with information about that particular product, and the DetailsView shows these details (see Figure 12).

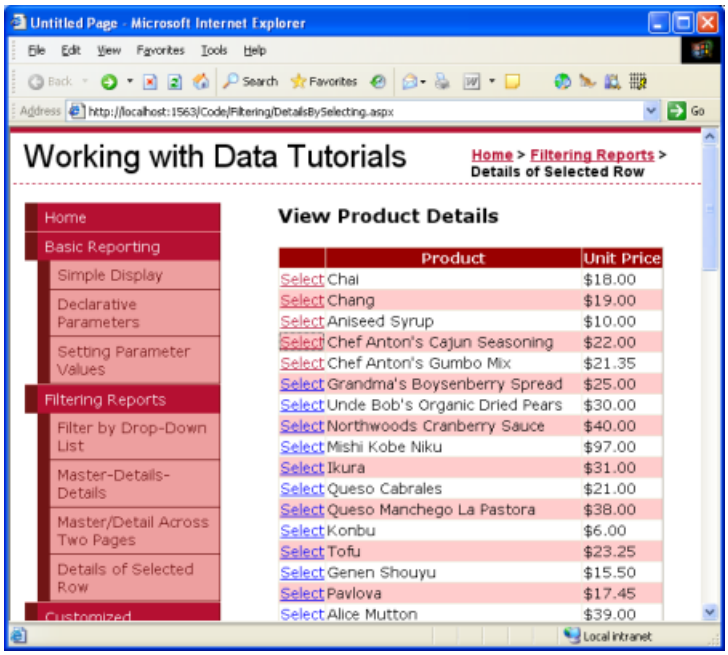


Figure 11: When First Visited, Only the GridView is Displayed (Click to view full-size image)

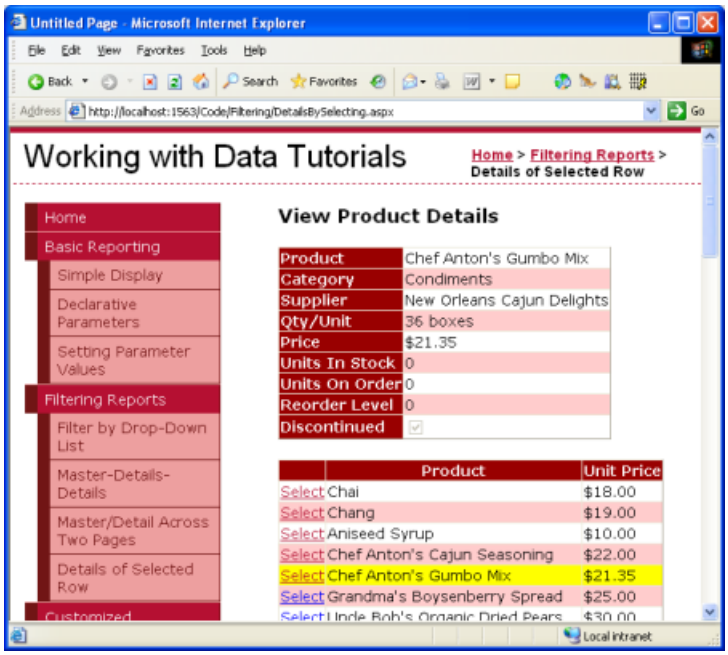


Figure 12: Upon Selecting a Row, the Product's Details are Displayed (Click to view full-size image)

Summary

In this and the preceding three tutorials we've seen a number of techniques for displaying master/detail reports. In this tutorial we examined using a selectable GridView to house the master records and a DetailsView to display details about the selected master record on the same page. In the earlier tutorials we looked at how to display master/details reports using DropDownLists and displaying master records on one web page and detail records on another.

This tutorial concludes our examination of master/detail reports. Starting with the next tutorial we'll begin our exploration of customized formatting with the GridView, DetailsView, and FormView. We'll see how to customize the appearance of these controls based on the data bound to them, how to summarize data in the GridView's footer, and how to use templates to obtain a greater degree of control over the layout.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Master/Detail Filtering With a DropDownList (VB)

2/20/2020 • 7 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll see how to display the master records in a DropDownList control and the details of the selected list item in a GridView.

Introduction

A common type of report is the *master/detail report*, in which the report begins by showing some set of "master" records. The user can then drill down into one of the master records, thereby viewing that master record's "details." Master/detail reports are an ideal choice for visualizing one-to-many relationships, such as a report showing all of the categories and then allowing a user to select a particular category and display its associated products. Additionally, master/detail reports are useful for displaying detailed information from particularly "wide" tables (ones that have a lot of columns). For example, the "master" level of a master/detail report might show just the product name and unit price of the products in the database, and drilling down into a particular product would show the additional product fields (category, supplier, quantity per unit, and so on).

There are many ways with which a master/detail report can be implemented. Over this and the next three tutorials we'll look at a variety of master/detail reports. In this tutorial we'll see how to display the master records in a DropDownList control and the details of the selected list item in a GridView. In particular, this tutorial's master/detail report will list category and product information.

Step 1: Displaying the Categories in a DropDownList

Our master/detail report will list the categories in a DropDownList, with the selected list item's products displayed further down in the page in a GridView. The first task ahead of us, then, is to have the categories displayed in a DropDownList. Open the `FilterByDropDownList.aspx` page in the `Filtering` folder, drag on a DropDownList from the Toolbox onto the page's designer, and set its `ID` property to `Categories`. Next, click on the Choose Data Source link from the DropDownList's smart tag. This will display the Data Source Configuration wizard.

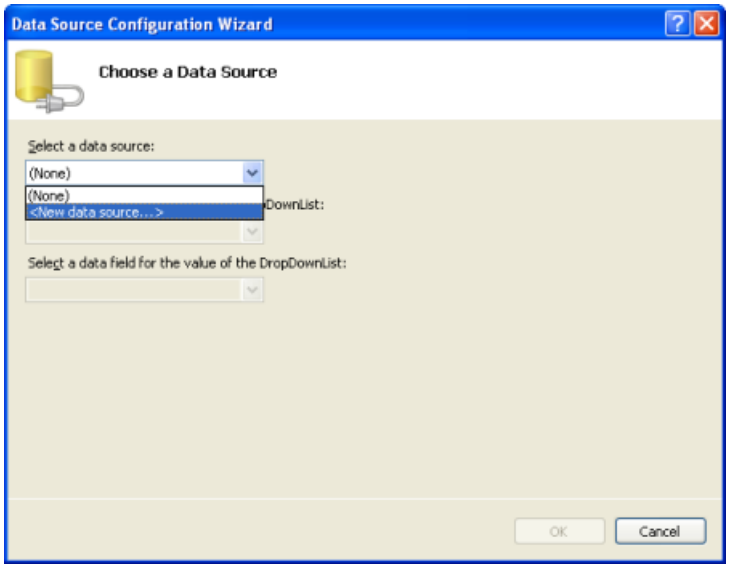


Figure 1: Specify the DropDownList's Data Source (Click to view full-size image)

Choose to add a new ObjectDataSource named `CategoriesDataSource` that invokes the `CategoriesBLL` class's `GetCategories()` method.

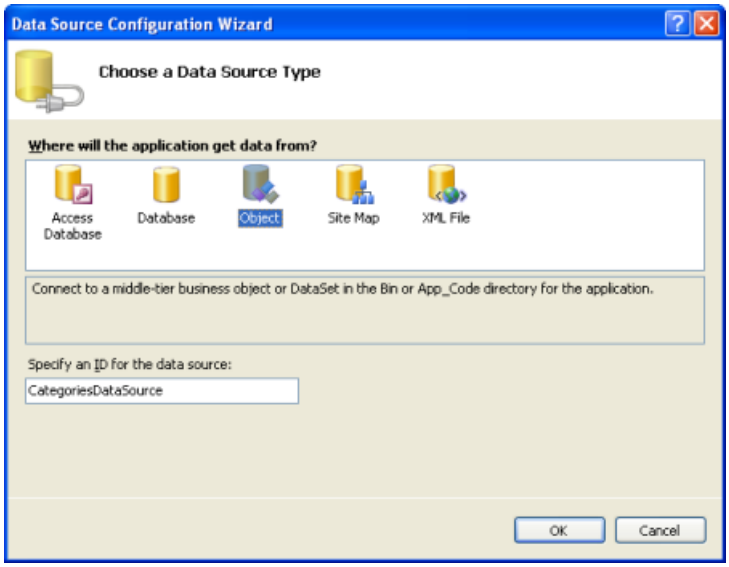


Figure 2: Add a New ObjectDataSource Named `CategoriesDataSource` (Click to view full-size image)

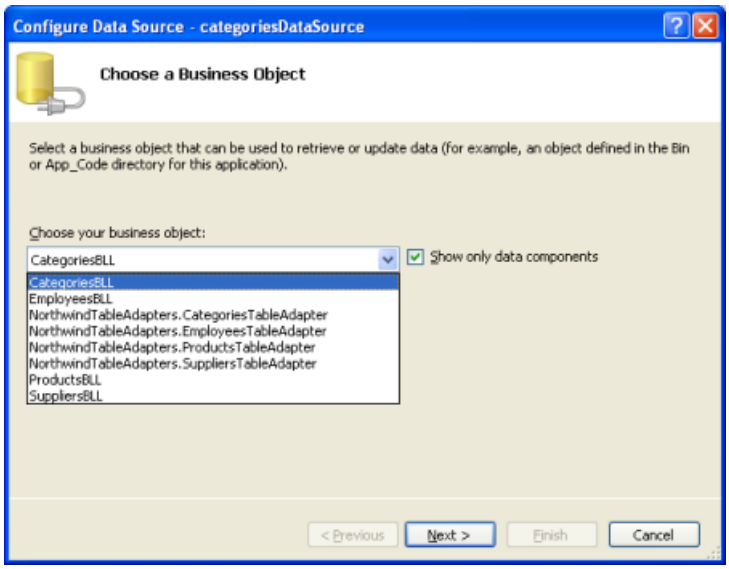


Figure 3: Choose to Use the `CategoriesBLL` Class (Click to view full-size image)

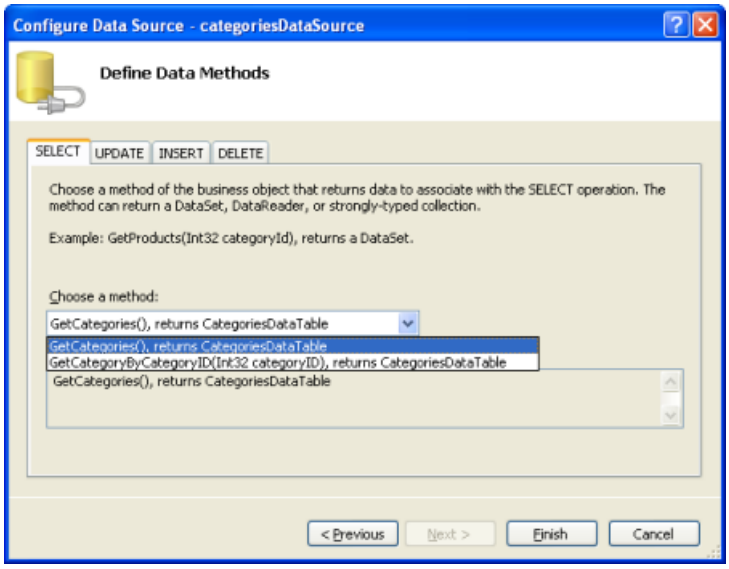


Figure 4: Configure the ObjectDataSource to Use the `GetCategories()` Method (Click to view full-size image)

After configuring the ObjectDataSource we still need to specify what data source field should be displayed in DropDownList and which one should be associated as the value for the list item. Have the `CategoryName` field as the display and `CategoryID` as the value for each list item.

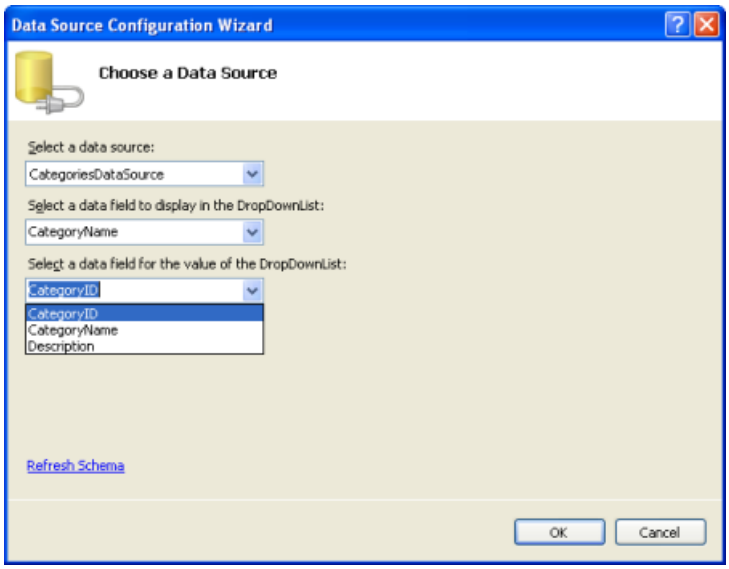


Figure 5: Have the DropDownList Display the `CategoryName` Field and Use `CategoryID` as the Value (Click to view full-size image)

At this point we have a DropDownList control that's populated with the records from the `Categories` table (all accomplished in about six seconds). Figure 6 shows our progress thus far when viewed through a browser.

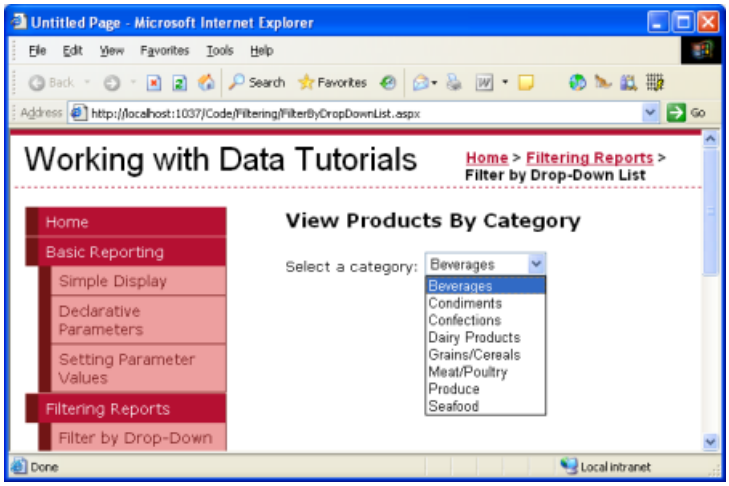


Figure 6: A Drop-Down Lists the Current Categories (Click to view full-size image)

Step 2: Adding the Products GridView

That last step in our master/detail report is to list the products associated with the selected category. To accomplish this, add a GridView to the page and create a new ObjectDataSource named `productsDataSource`. Have the `productsDataSource` control cull its data from the `ProductsBLL` class's `GetProductsByCategoryID(categoryID)` method.

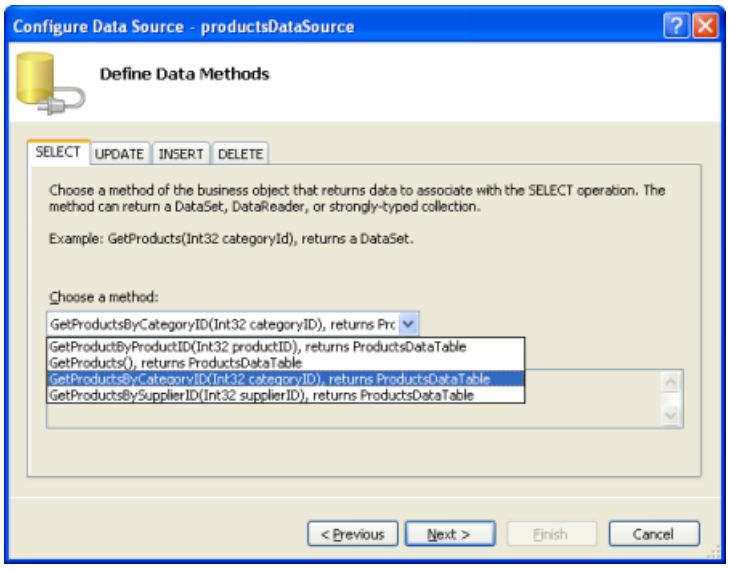


Figure 7: Select the `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

After choosing this method, the ObjectDataSource wizard prompts us for the value for the method's *`categoryID`* parameter. To use the value of the selected `categories` DropDownList item set the Parameter source to Control and the ControlID to `Categories`.

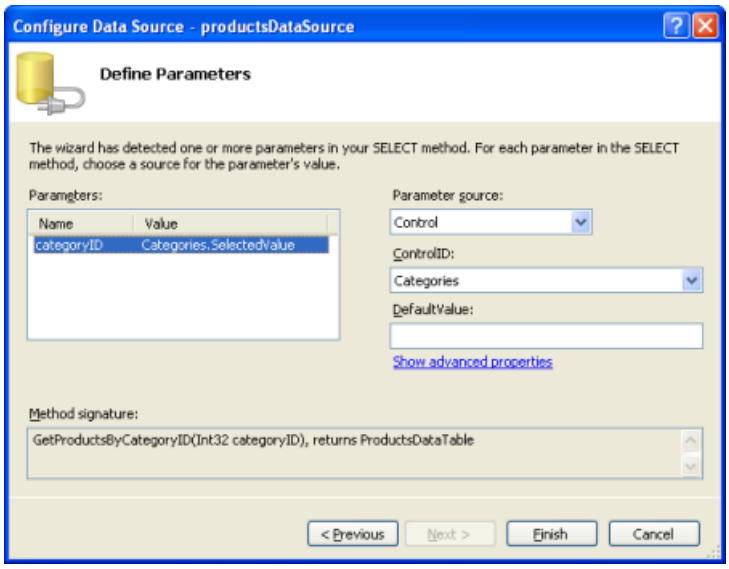


Figure 8: Set the *`categoryID`* Parameter to the Value of the `Categories` DropDownList (Click to view full-size image)

Take a moment to check out our progress in a browser. When first visiting the page, those products belong to the selected category (Beverages) are displayed (as shown in Figure 9), but changing the DropDownList doesn't update the data. This is because a postback must occur for the GridView to update. To accomplish this we have two options (neither of which requires writing any code):

- **Set the categories DropDownList's** AutoPostBack property **to True.** (You can accomplish this by checking the Enable AutoPostBack option in the DropDownList's smart tag.) This will trigger a postback whenever the DropDownList's selected item is changed by the user. Therefore, when the user selects a new category from the DropDownList a postback will ensue and the GridView will be updated with the products for the newly selected category. (This is the approach I've used in this tutorial.)
- **Add a Button Web control next to the DropDownList.** Set its `Text` property to Refresh or something similar. With this approach, the user will need to select a new category and then click the Button. Clicking the Button will cause a postback and update the GridView to list those products of the selected category.

Figures 9 and 10 illustrate the master/detail report in action.

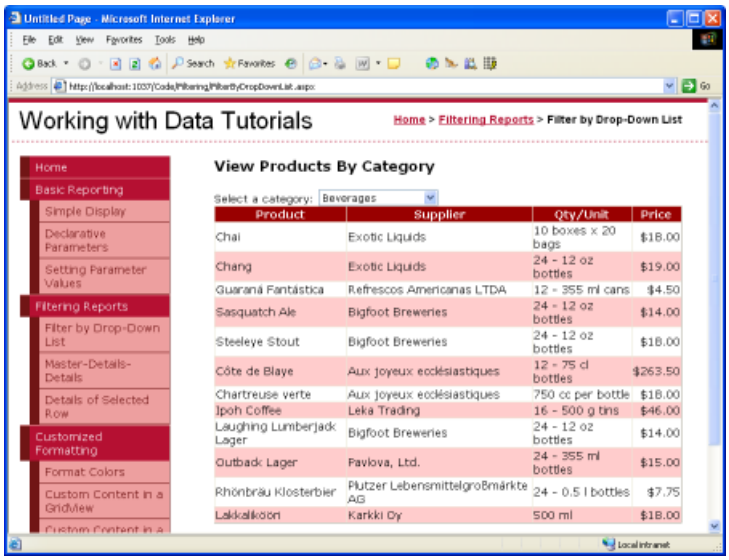


Figure 9: When First Visiting the Page, the Beverage Products are Displayed (Click to view full-size image)

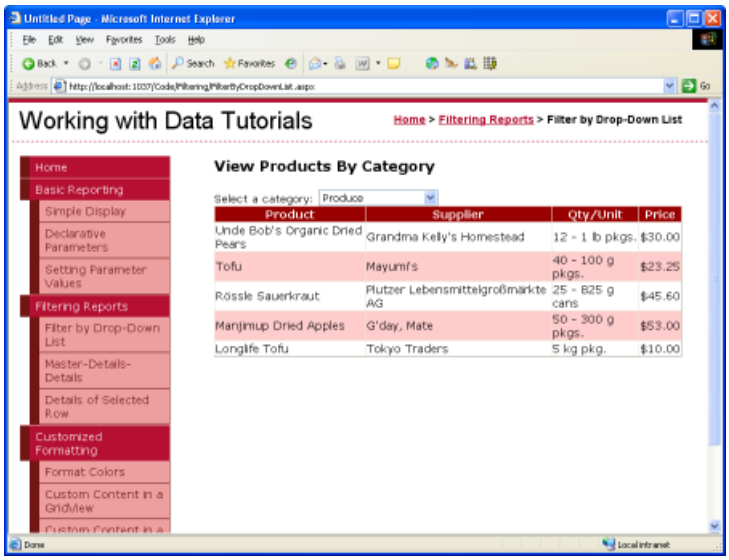


Figure 10: Selecting a New Product (Produce) Automatically Causes a PostBack, Updating the GridView (Click to view full-size image)

Adding a "-- Choose a Category --" List Item

When first visiting the `FilterByDropDownList.aspx` page the categories DropDownList's first list item (Beverages) is selected by default, showing the beverage products in the GridView. Rather than showing the first category's products, we may want to instead have a DropDownList item selected that says something like, "-- Choose a Category --".

To add a new list item to the DropDownList, go to the Properties window and click on the ellipses in the `Items` property. Add a new list item with the `Text` "-- Choose a Category --" and the `Value` `-1`.

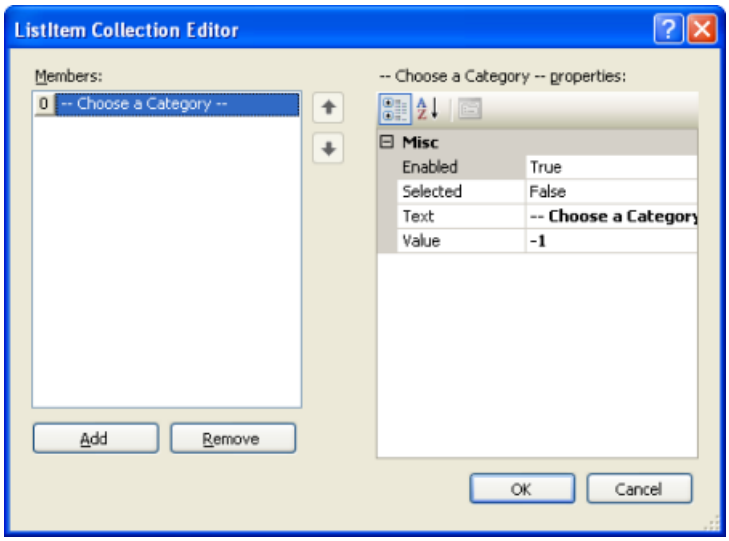


Figure 11: Add a -- Choose a Category -- List Item (Click to view full-size image)

Alternatively, you can add the list item by adding the following markup to the DropDownList:

```
<asp:DropDownList ID="categories" runat="server"
    AutoPostBack="True" DataSourceID="categoriesDataSource"
    DataTextField="CategoryName" DataValueField="CategoryID"
    EnableViewState="False">
    <asp:ListItem Value="-1">
       -- Choose a Category --
    </asp:ListItem>
</asp:DropDownList>
```

Additionally, we need to set the DropDownList control's `AppendDataBoundItems` to True because when the categories are bound to the DropDownList from the ObjectDataSource they'll overwrite any manually-added list items if `AppendDataBoundItems` isn't True.

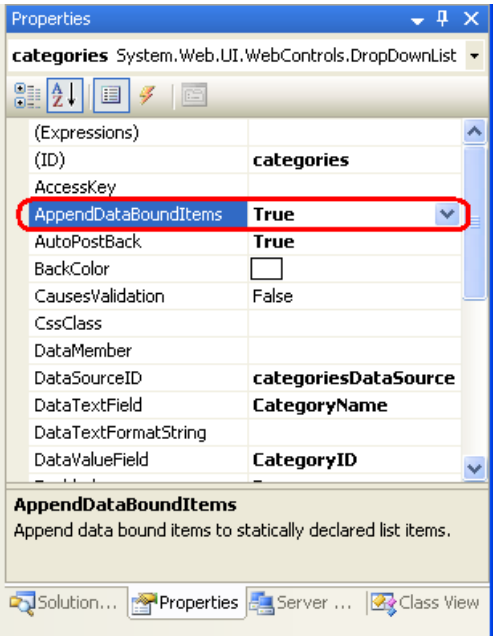


Figure 12: Set the `AppendDataBoundItems` Property to True

After these changes, when first visiting the page the "-- Choose a Category --" option is selected and no products are displayed.

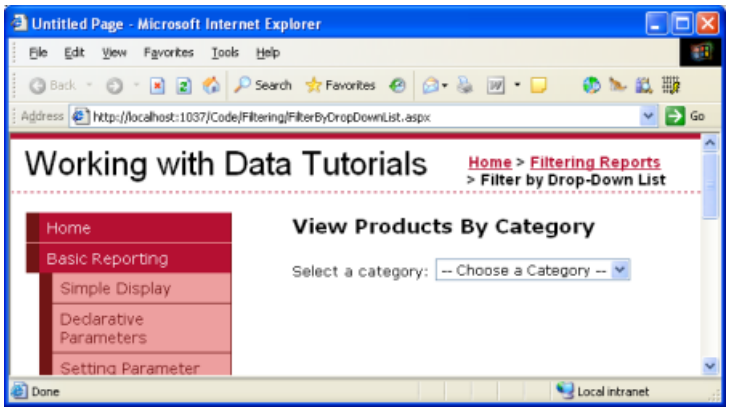


Figure 13: On the Initial Page Load No Products are Displayed (Click to view full-size image)

The reason no products are displayed when because the "-- Choose a Category --" list item is selected is because its value is `-1` and there are no products in the database with a `CategoryID` of `-1`. If this is the behavior you want then you're done at this point! If, however, you want to display *all* of the categories when the "-- Choose a Category --" list item is selected, return to the `ProductsBLL` class and customize the `GetProductsByCategoryID(categoryID)` method so that it invokes the `GetProducts()` method if the passed in *`categoryID`* parameter is less than zero:

```
Public Function GetProductsByCategoryID(categoryID As Integer) _
    As Northwind.ProductsDataTable
    If categoryID < 0 Then
        Return GetProducts()
    Else
        Return Adapter.GetProductsByCategoryID(categoryID)
    End If
End Function
```

The technique used here is similar to the approach we used to display all suppliers back in the Declarative Parameters tutorial, although for this example we're using a value of `-1` to indicate that all records should be retrieved as opposed to `Nothing`. This is because the *`categoryID`* parameter of the `GetProductsByCategoryID(categoryID)` method expects as integer value passed in, whereas in the Declarative Parameters tutorial we were passing in a string input parameter.

Figure 14 shows a screen shot of `FilterByDropDownList.aspx` when the "-- Choose a Category --" option is selected. Here, all of the products are displayed by default, and the user can narrow the display by choosing a specific category.

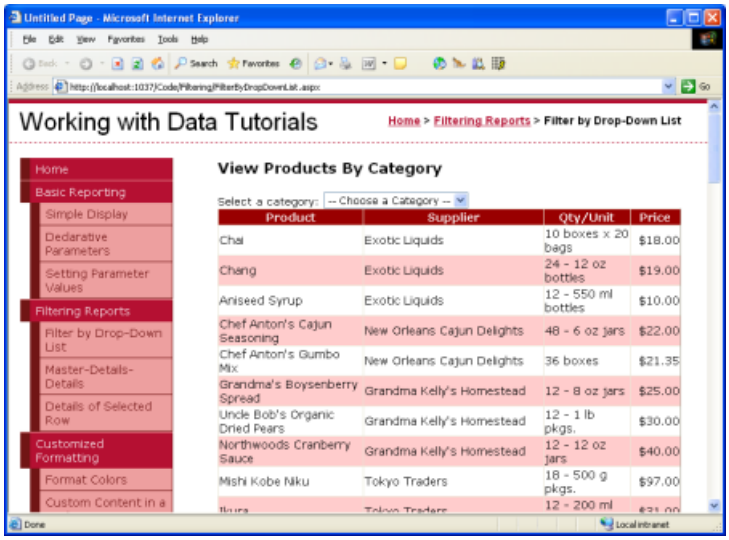


Figure 14: All of the Products are Now Listed By Default (Click to view full-size image)

Summary

When displaying hierarchically-related data, it often helps to present the data using master/detail reports, from which the user can start perusing the data from the top of the hierarchy and drill down into details. In this tutorial we examined building a simple master/detail report showing a selected category's products. This was accomplished by using a DropDownList for the list of categories and a GridView for the products belonging to the selected category.

In the next tutorial we'll take the DropDownList interface one step further, using two DropDownLists.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Master/Detail Filtering With Two DropDownLists (VB)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> This tutorial expands the master/detail relationship to add a third layer, using two DropDownList controls to select the desired parent and grandparent records.

Introduction

In the previous tutorial we examined how to display a simple master/details report using a single DropDownList populated with the categories and a GridView showing those products that belong to the selected category. This report pattern works well when displaying records that have a one-to-many relationship and can easily be extended to work for scenarios that include multiple one-to-many relationships. For example, an order entry system would have tables that correspond to customers, orders, and order line items. A given customer may have multiple orders with each order consisting of multiple items. Such data can be presented to the user with two DropDownLists and a GridView. The first DropDownList would have a list item for each customer in the database with the second one's contents being the orders placed by the selected customer. A GridView would list the line items from the selected order.

While the Northwind database include the canonical customer/order/order details information in its `Customers`, `Orders`, and `Order Details` tables, these tables aren't captured in our architecture. Nonetheless, we can still illustrate using two dependent DropDownLists. The first DropDownList will list the categories and the second the products belonging to the selected category. A DetailsView will then list the details of the selected product.

Step 1: Creating and Populating the Categories DropDownList

Our first goal is to add the DropDownList that lists the categories. These steps were examined in detail in the preceding tutorial, but are summarized here for completeness.

Open the `MasterDetailsDetails.aspx` page in the `Filtering` folder, add a DropDownList to the page, set its `ID` property to `Categories`, and then click the Configure Data Source link in its smart tag. From the Data Source Configuration Wizard choose to add a new data source.

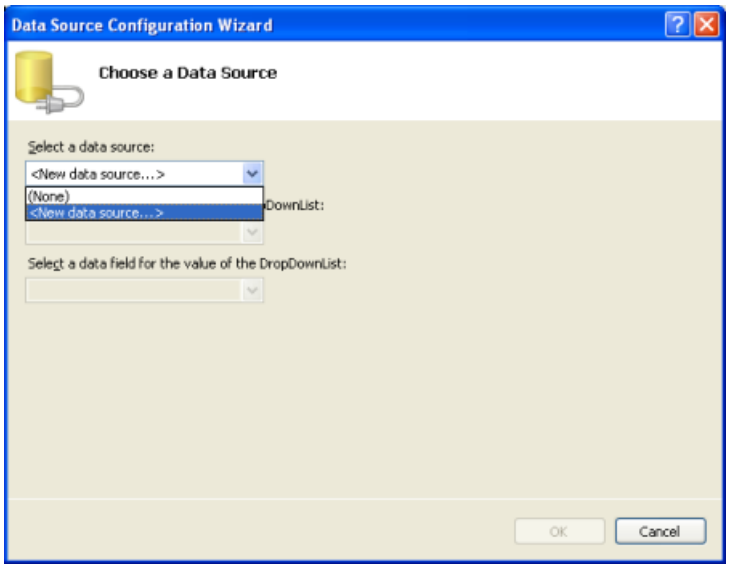


Figure 1: Add a New Data Source for the DropDownList (Click to view full-size image)

The new data source should, naturally, be an ObjectDataSource. Name this new ObjectDataSource `CategoriesDataSource` and have it invoke the `CategoriesBLL` object's `GetCategories()` method.

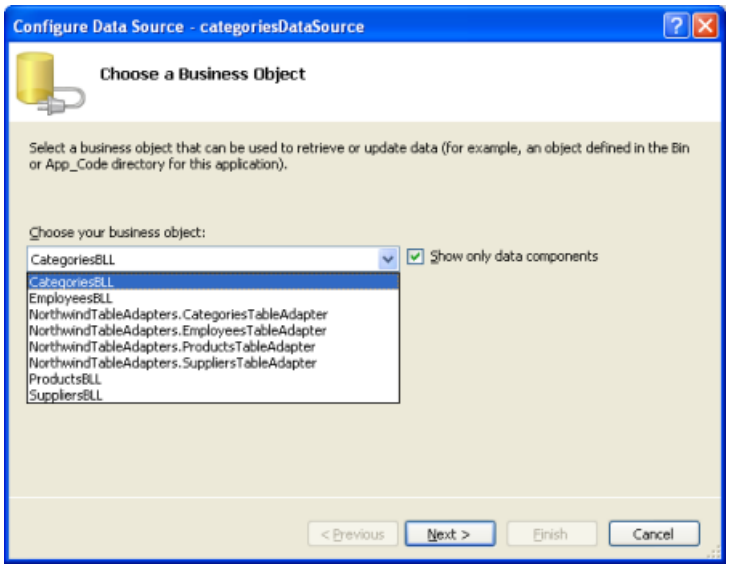


Figure 2: Choose to Use the `CategoriesBLL` Class (Click to view full-size image)

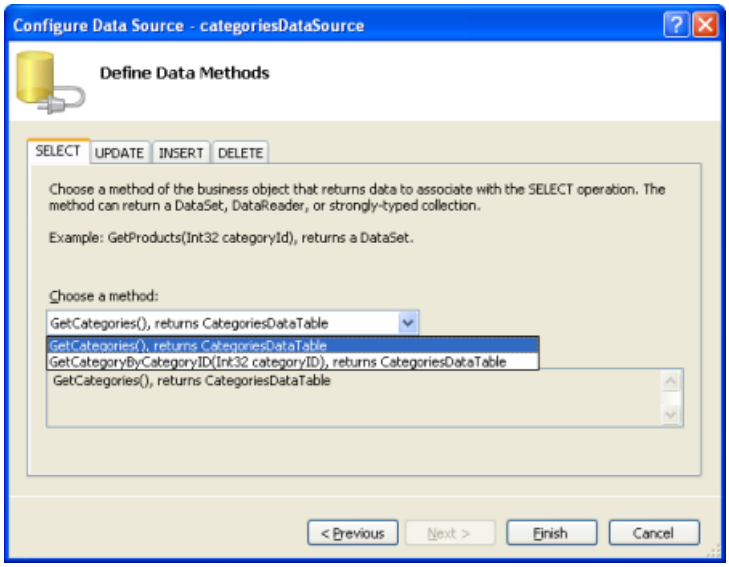


Figure 3: Configure the ObjectDataSource to Use the `GetCategories()` Method (Click to view full-size image)

After configuring the ObjectDataSource we still need to specify which data source field should be displayed in the `Categories` DropDownList and which one should be configured as the value for the list item. Set the `CategoryName` field as the display and `CategoryID` as the value for each list item.

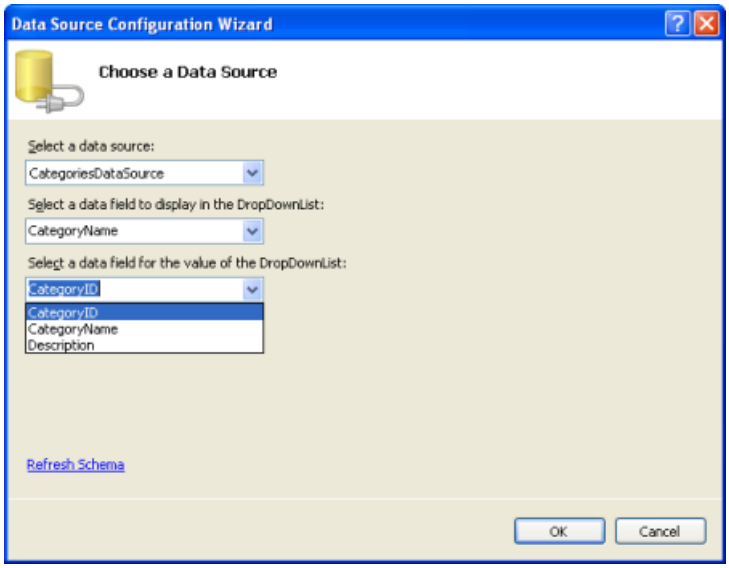


Figure 4: Have the DropDownList Display the `CategoryName` Field and Use `CategoryID` as the Value (Click to view full-size image)

At this point we have a DropDownList control (`Categories`) that's populated with the records from the `Categories` table. When the user chooses a new category from the DropDownList we'll want a postback to occur in order to refresh the product DropDownList that we're going to create in Step 2. Therefore, check the Enable AutoPostBack option from the `categories` DropDownList's smart tag.

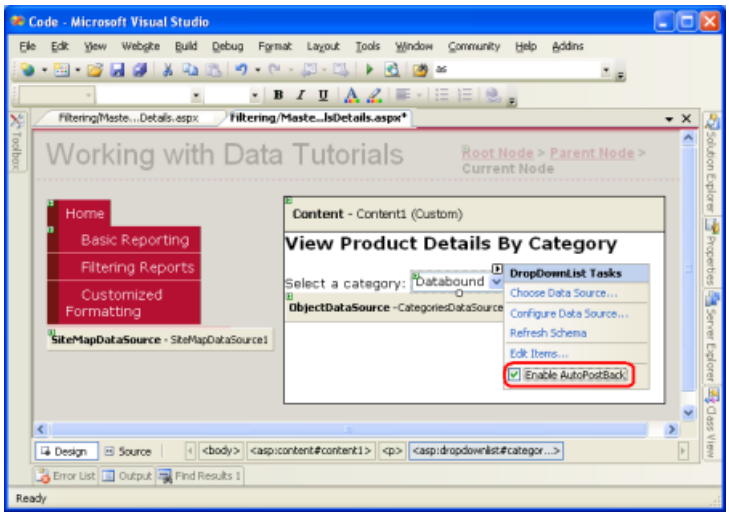


Figure 5: Enable AutoPostBack for the `Categories` DropDownList (Click to view full-size image)

Step 2: Displaying the Selected Category's Products in a Second DropDownList

With the `Categories` DropDownList completed, our next step is to display a DropDownList of products belonging to the selected category. To accomplish this, add another DropDownList to the page named `ProductsByCategory`. As with the `Categories` DropDownList, create a new ObjectDataSource for the `ProductsByCategory` DropDownList named `ProductsByCategoryDataSource`.

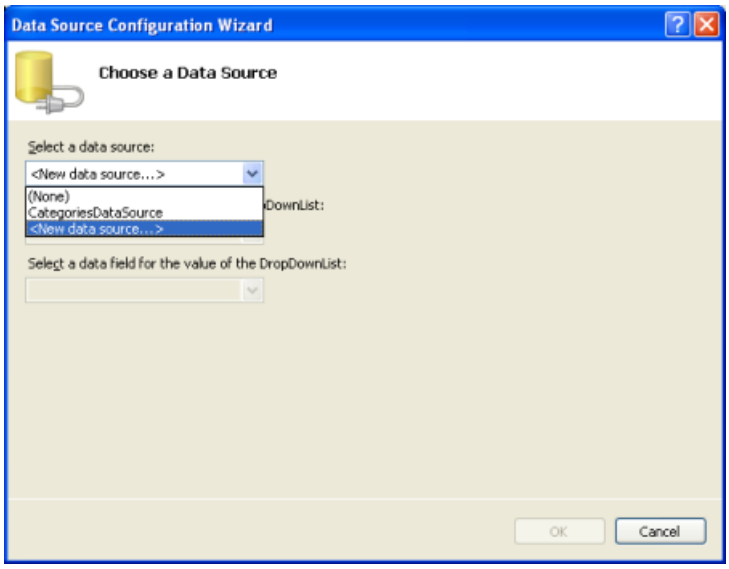


Figure 6: Add a New Data Source for the `ProductsByCategory` DropDownList (Click to view full-size image)

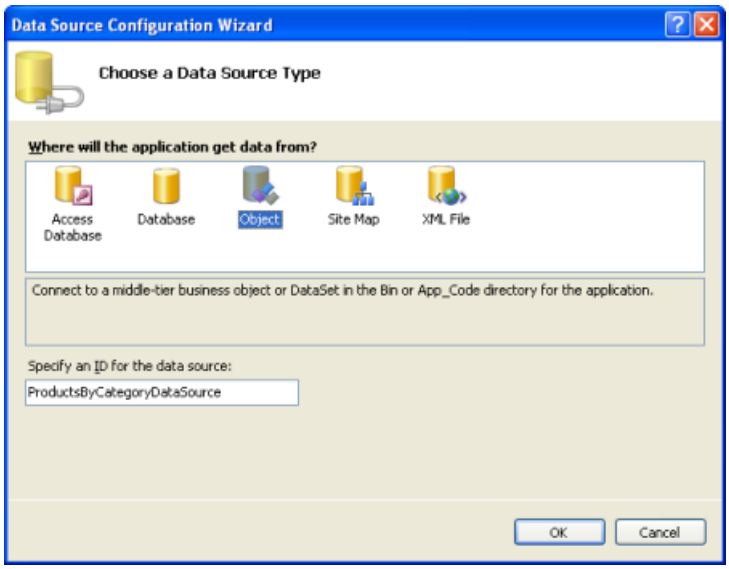


Figure 7: Create a New ObjectDataSource Named `ProductsByCategoryDataSource` (Click to view full-size image)

Since the `ProductsByCategory` DropDownList needs to display just those products belonging to the selected category, have the ObjectDataSource invoke the `GetProductsByCategoryID(categoryID)` method from the `ProductsBLL` object.

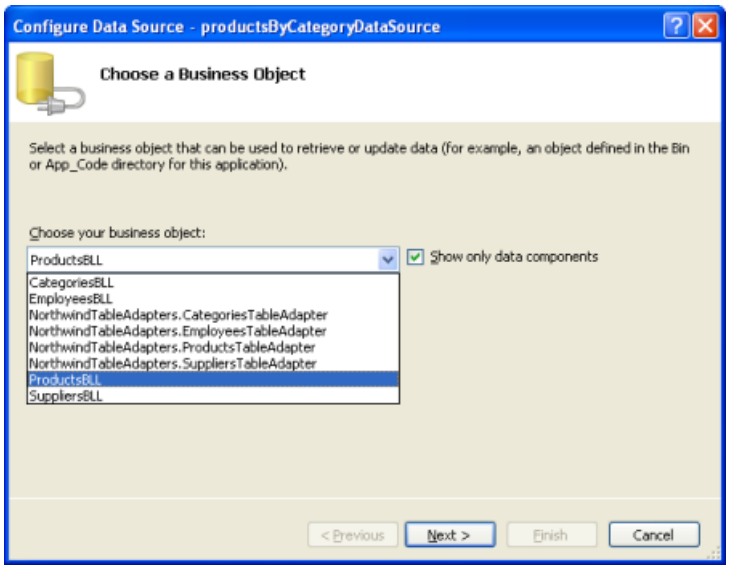


Figure 8: Choose to Use the `ProductsBLL` Class (Click to view full-size image)

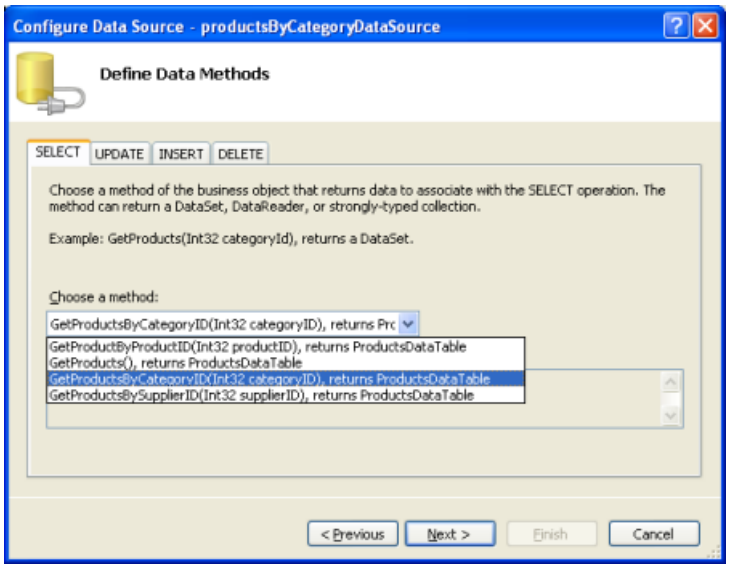


Figure 9: Configure the ObjectDataSource to Use the `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

In the final step of the wizard we need to specify the value of the *`categoryID`* parameter. Assign this parameter to the selected item from the `Categories` DropDownList.

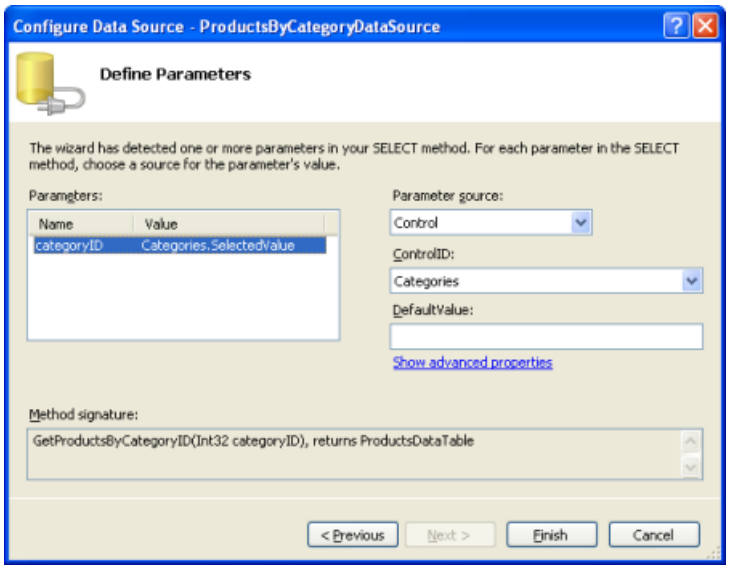


Figure 10: Pull the *`categoryID`* Parameter Value from the `Categories` DropDownList (Click to view full-size image)

With the ObjectDataSource configured, all that remains is to specify what data source fields are used for the display and value of the DropDownList's items. Display the `ProductName` field and use the `ProductID` field as the value.

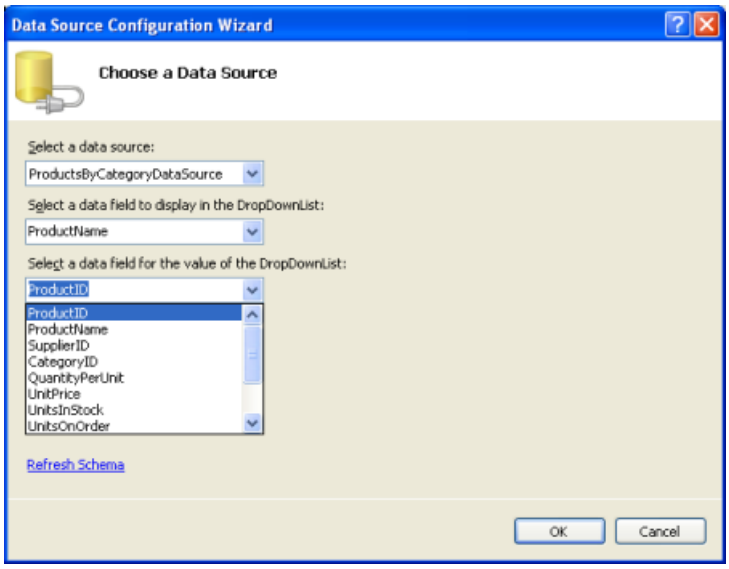


Figure 11: Specify the Data Source Fields Used for the DropDownList's `ListItem` s' `Text` and `Value` Properties (Click to view full-size image)

With the ObjectDataSource and `ProductsByCategory` DropDownList configured our page will display two DropDownLists: the first will list all of the categories while the second will list those products belonging to the selected category. When the user selects a new category from the first DropDownList, a postback will ensue and the second DropDownList will be rebound, showing those products that belong to the newly selected category. Figures 12 and 13 show `MasterDetailsDetails.aspx` in action when viewed through a browser.

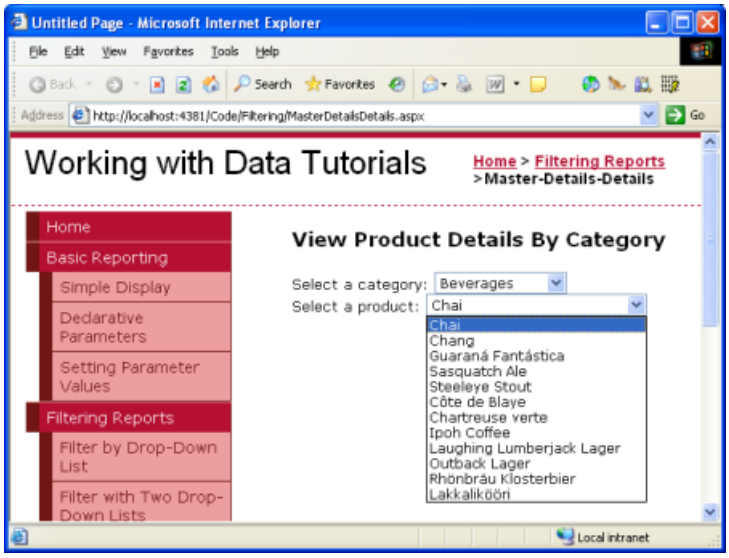


Figure 12: When First Visiting the Page, the Beverages Category is Selected (Click to view full-size image)

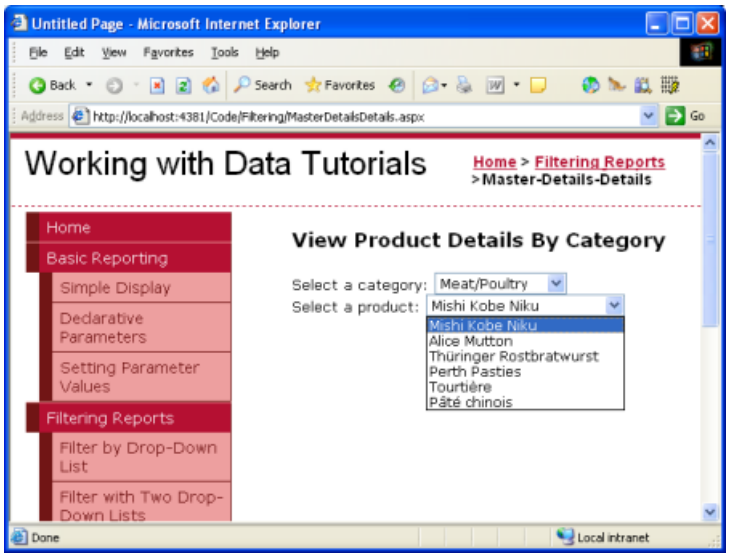


Figure 13: Choosing a Different Category Displays the New Category's Products (Click to view full-size image)

Currently the `productsByCategory` DropDownList, when changed, does *not* cause a postback. However, we will want a postback to occur once we add a DetailsView to display the selected product's details (Step 3). Therefore, check the Enable AutoPostBack checkbox from the `productsByCategory` DropDownList's smart tag.

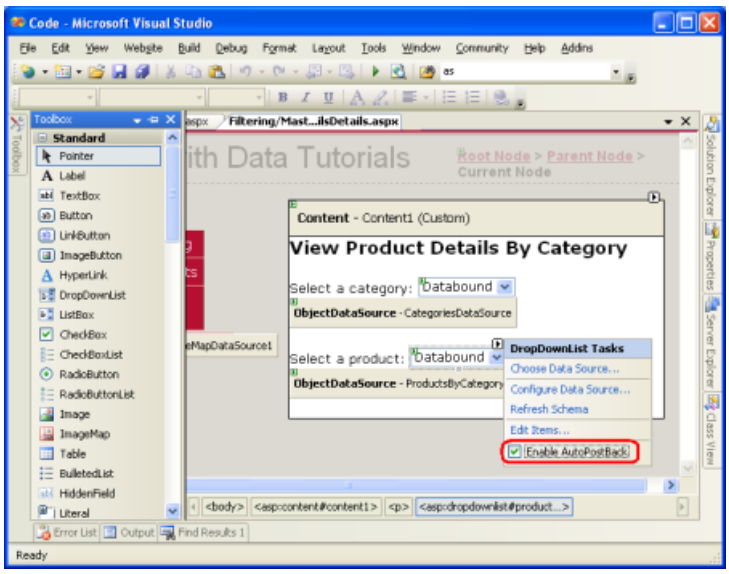


Figure 14: Enable the AutoPostBack Feature for the `productsByCategory` DropDownList (Click to view full-size image)

Step 3: Using a DetailsView to Display Details for the Selected Product

The final step is to display the details for the selected product in a DetailsView. To accomplish this, add a DetailsView to the page, set its `ID` property to `ProductDetails`, and create a new ObjectDataSource for it. Configure this ObjectDataSource to pull its data from the `ProductsBLL` class's `GetProductByProductID(productID)` method using the selected value of the `ProductsByCategory` DropDownList for the value of the *`productID`* parameter.

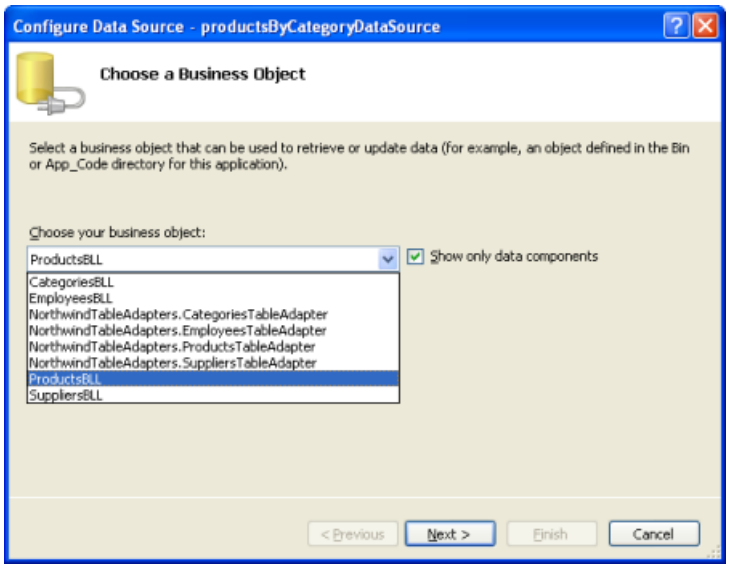


Figure 15: Choose to Use the `ProductsBLL` Class (Click to view full-size image)

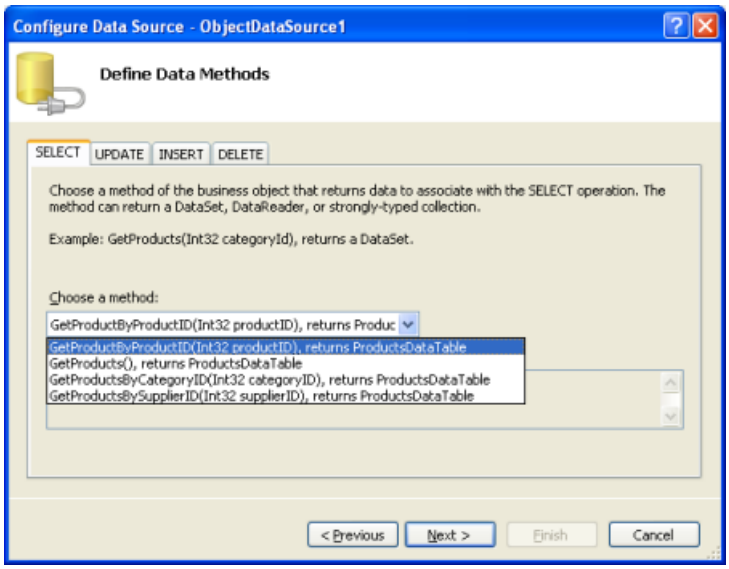


Figure 16: Configure the ObjectDataSource to Use the `GetProductByProductID(productID)` Method (Click to view full-size image)

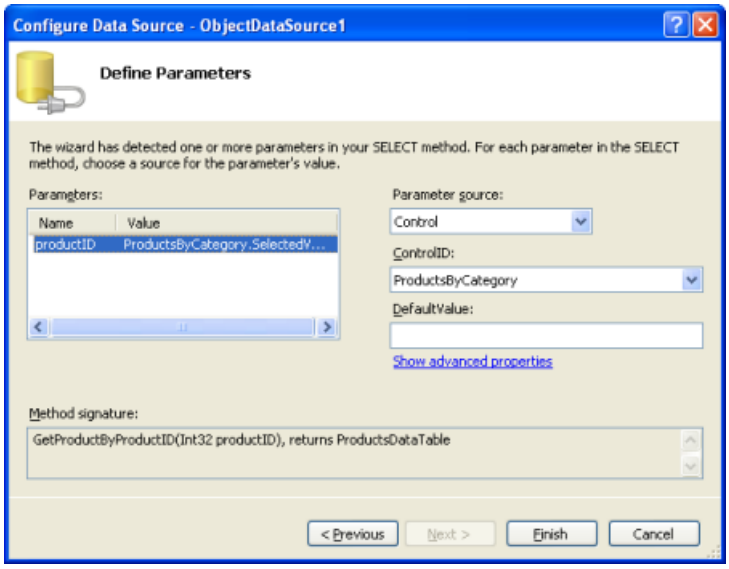


Figure 17: Pull the *`productID`* Parameter Value from the `ProductsByCategory` DropDownList (Click to view full-size image)

You can choose to display any of the available fields in the `ProductDetails` DetailsView. I've opted to remove the

`ProductID`, `SupplierID`, and `CategoryID` fields and reordered and formatted the remaining fields. In addition, I cleared out the DetailsView's `Height` and `Width` properties, allowing the DetailsView to expand to the width needed to best display its data rather than having it constrained to a specified size. The full markup appears below:

```
<asp:DetailsView ID="ProductDetails" runat="server"
    AutoGenerateRows="False" DataKeyNames="ProductID"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <Fields>
        <asp:BoundField DataField="ProductName"
          HeaderText="Product" SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName"
          HeaderText="Category" ReadOnly="True"
          SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName"
          HeaderText="Supplier" ReadOnly="True"
          SortExpression="SupplierName" />
        <asp:BoundField DataField="QuantityPerUnit"
           HeaderText="Qty/Unit" SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice"
           DataFormatString="{0:c}" HeaderText="Price"
            HtmlEncode="False" SortExpression="UnitPrice" />
        <asp:BoundField DataField="UnitsInStock"
            HeaderText="UnitsInStock"
            SortExpression="Units In Stock" />
        <asp:BoundField DataField="UnitsOnOrder"
            HeaderText="UnitsOnOrder"
            SortExpression="Units On Order" />
        <asp:BoundField DataField="ReorderLevel"
            HeaderText="ReorderLevel" SortExpression="Reorder Level" />
        <asp:CheckBoxField DataField="Discontinued"
            HeaderText="Discontinued" SortExpression="Discontinued" />
    </Fields>
</asp:DetailsView>
```

Take a moment to try out the `MasterDetailsDetails.aspx` page in a browser. At first glance it may appear that everything is working as desired, but there's a subtle problem. When you choose a new category the `ProductsByCategory` DropDownList is updated to include those products for the selected category, but the `ProductDetails` DetailsView continued to show the previous product information. The DetailsView is updated when choosing a different product for the selected category. Furthermore, if you test thoroughly enough, you'll find that if you continually choose new categories (such as choosing Beverages from the `Categories` DropDownList, then Condiments, then Confections) every other category selection causes the `ProductDetails` DetailsView to be refreshed.

To help concretize this problem, let's look at a specific example. When you first visit the page the Beverages category is selected and the related products are loaded in the `ProductsByCategory` DropDownList. Chai is the selected product and its details are displayed in the `ProductDetails` DetailsView, as shown in Figure 18.

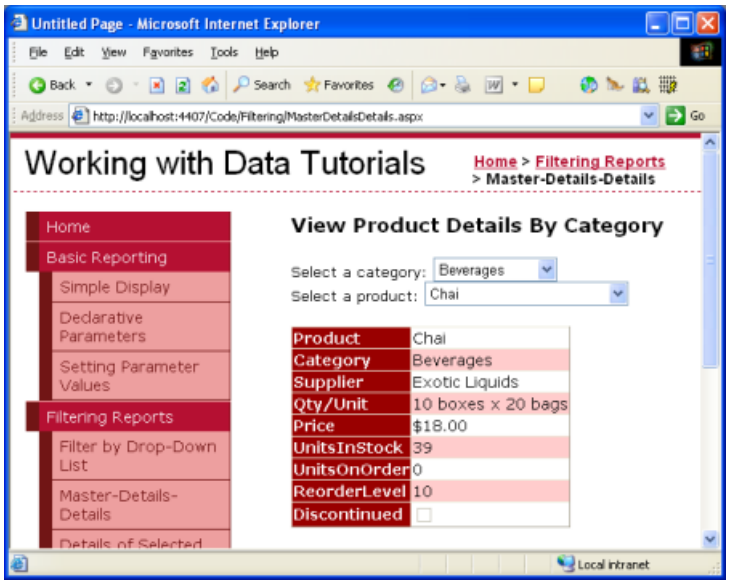


Figure 18: The Selected Product's Details are Displayed in a DetailsView (Click to view full-size image)

If you change the category selection from Beverages to Condiments, a postback occurs and the `ProductsByCategory` DropDownList is updated accordingly, but the DetailsView still displays details for Chai.

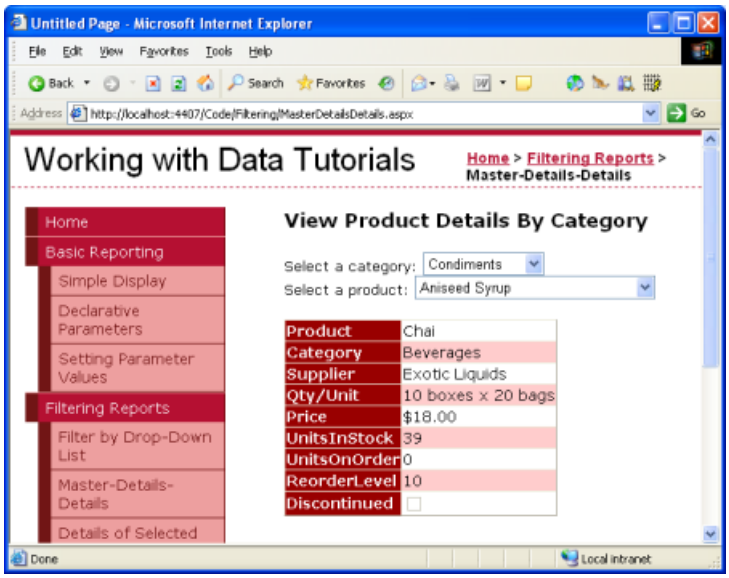


Figure 19: The Previously Selected Product's Details are Still Displayed (Click to view full-size image)

Picking a new product from the list refreshes the DetailsView as expected. If you pick a new category after changing the product, the DetailsView again won't refresh. However, if instead of choosing a new product you selected a new category, the DetailsView would refresh. What in the world is going on here?

The problem is a timing issue in the page's lifecycle. Whenever a page is requested it proceeds through a number of steps as its rendering. In one of these steps the ObjectDataSource controls check to see if any of their `SelectParameters` values have changed. If so, the data Web control bound to the ObjectDataSource knows that it needs to refresh its display. For example, when a new category is selected, the `ProductsByCategoryDataSource` ObjectDataSource detects that its parameter values have changed and the `ProductsByCategory` DropDownList rebinds itself, getting the products for the selected category.

The problem that arises in this situation is that the point in the page lifecycle that the ObjectDataSources check for changed parameters occurs *before* the rebinding of the associated data Web controls. Therefore, when selecting a new category the `ProductsByCategoryDataSource` ObjectDataSource detects a change in its parameter's value. The ObjectDataSource used by the `ProductDetails` DetailsView, however, doesn't note any such changes because the `ProductsByCategory` DropDownList has yet to be rebound. Later in the lifecycle the `ProductsByCategory` DropDownList rebinds to its ObjectDataSource, grabbing the products for the newly selected

category. While the `ProductsByCategory` DropDownList's value has changed, the `ProductDetails` DetailsView's ObjectDataSource has already done its parameter value check; therefore, the DetailsView displays its previous results. This interaction is depicted in Figure 20.

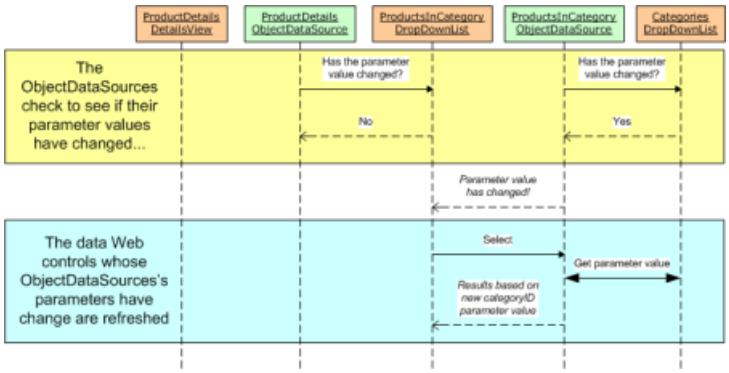


Figure 20: The `ProductsByCategory` DropDownList Value Changes After the `ProductDetails` DetailsView's ObjectDataSource Checks for Changes (Click to view full-size image)

To remedy this we need to explicitly rebind the `ProductDetails` DetailsView after the `ProductsByCategory` DropDownList has been bound. We can accomplish this by calling the `ProductDetails` DetailsView's `DataBind()` method when the `ProductsByCategory` DropDownList's `DataBound` event fires. Add the following event handler code to the `MasterDetailsDetails.aspx` page's code-behind class (refer to the "Programmatically Setting the ObjectDataSource's Parameter Values" for a discussion on how to add an event handler):

```
Protected Sub ProductsByCategory_DataBound(sender As Object, e As EventArgs) _
    Handles ProductsByCategory.DataBound
        ProductDetails.DataBind()
End Sub
```

After this explicit call to the `ProductDetails` DetailsView's `DataBind()` method has been added, the tutorial works as expected. Figure 21 highlights how this changed remedied our earlier problem.

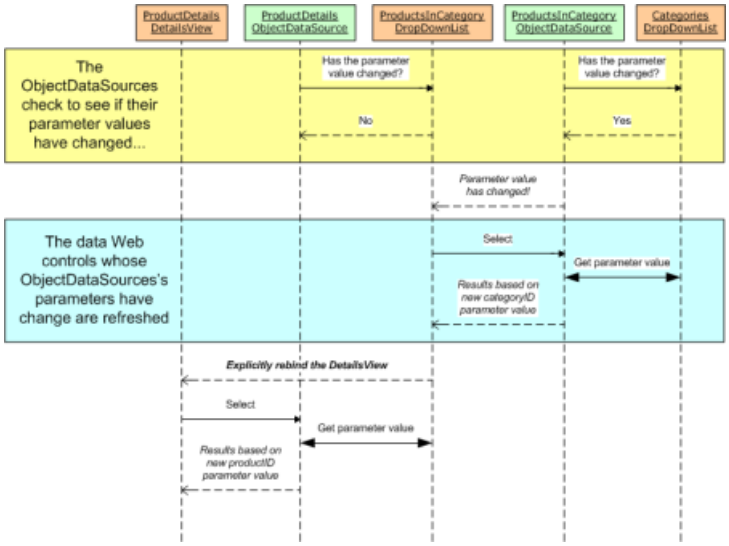


Figure 21: The `ProductDetails` DetailsView is Explicitly Refreshed When the `ProductsByCategory` DropDownList's `DataBound` Event Fires (Click to view full-size image)

Summary

The DropDownList serves as an ideal user interface element for master/detail reports where there is a one-to-many relationship between the master and detail records. In the preceding tutorial we saw how to use a single DropDownList to filter the products displayed by the selected category. In this tutorial we replaced the GridView

of products with a DropDownList, and used a DetailsView to display the details of the selected product. The concepts discussed in this tutorial can easily be extended to data models involving multiple one-to-many relationships, such as customers, orders, and order items. In general, you can always add a DropDownList for each of the "one" entities in the one-to-many relationships.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Master/Detail Filtering Across Two Pages (VB)

9/16/2020 • 11 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll implement this pattern by using a GridView to list the suppliers in the database. Each supplier row in the GridView will contain a View Products link that, when clicked, will take the user to a separate page that lists those products for the selected supplier.

Introduction

In the previous two tutorials we saw how to display master/detail reports in a single web page using DropDownLists to display the "master" records and a GridView or DetailsView control to display the "details." Another common pattern used for master/details reports is to have the master records on one web page and the details shown on another. A forum website, like the ASP.NET Forums, is a great example of this pattern in practice. The ASP.NET Forums are composed of a variety of forums Getting Started, Web Forms, Data Presentation Controls, and so on. Each forum is composed of many threads and each thread is composed of a number of posts. On the ASP.NET Forums homepage, the forums are listed. Clicking on a forum whisks you to a `ShowForum.aspx` page, which lists the threads for that forum. Likewise, clicking on a thread takes you to `ShowPost.aspx`, which displays the posts for the thread that was clicked.

In this tutorial we'll implement this pattern by using a GridView to list the suppliers in the database. Each supplier row in the GridView will contain a View Products link that, when clicked, will take the user to a separate page that lists those products for the selected supplier.

Step 1: Adding `SupplierListMaster.aspx` and `ProductsForSupplierDetails.aspx` Pages to the `Filtering` Folder

When defining the page layout in the third tutorial we added a number of "starter" pages in the `BasicReporting`, `Filtering`, and `CustomFormatting` folders. However, we did not add a starter page for this tutorial at that time, so take a moment to add two new pages to the `Filtering` folder: `SupplierListMaster.aspx` and `ProductsForSupplierDetails.aspx`. `SupplierListMaster.aspx` will list the "master" records (the suppliers) while `ProductsForSupplierDetails.aspx` will display the products for the selected supplier.

When creating these two new pages be certain to associate them with the `Site.master` master page.

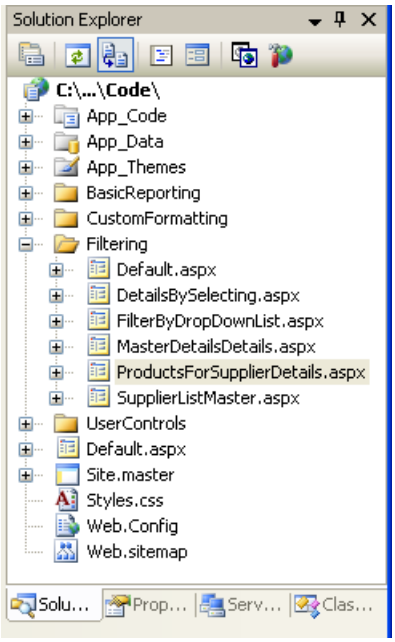


Figure 1: Add the `SupplierListMaster.aspx` and `ProductsForSupplierDetails.aspx` Pages to the `Filtering` Folder

Also, when adding new pages to the project, be sure to update the site map file, `Web.sitemap`, accordingly. For this tutorial simply add the `SupplierListMaster.aspx` page to the site map using the following XML content as a child of the Filtering Reports `<siteMapNode>` element:

```
<siteMapNode url="~/Filtering/SupplierListMaster.aspx"
  title="Master/Detail Across Two Pages"
  description="Master records on one page, detail records on another." />
```

> **NOTE**
>
> You can help automate the process of updating the site map file when adding new ASP.NET pages using K. Scott Allen's free Visual Studio Site Map Macro.

Step 2: Displaying the Supplier List in `SupplierListMaster.aspx`

With the `SupplierListMaster.aspx` and `ProductsForSupplierDetails.aspx` pages created, our next step is to create the GridView of suppliers in `SupplierListMaster.aspx`. Add a GridView to the page and bind it to a new ObjectDataSource. This ObjectDataSource should use the `SuppliersBLL` class's `GetSuppliers()` method to return all suppliers.

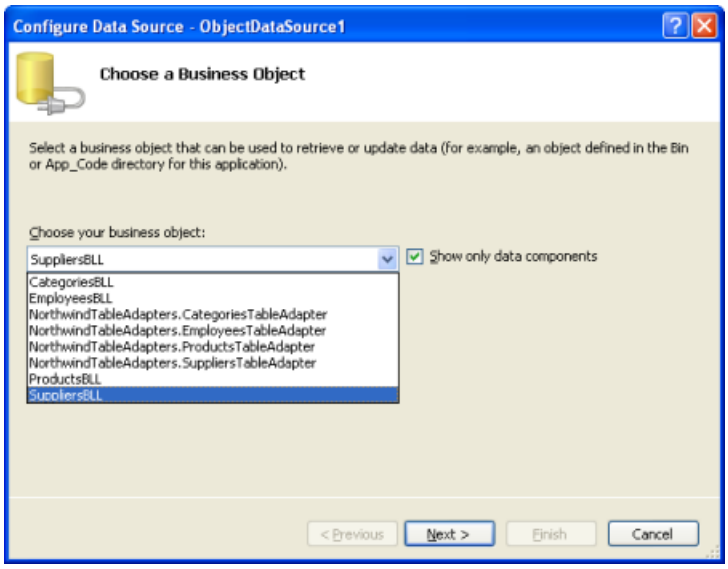


Figure 2: Select the `SuppliersBLL` Class (Click to view full-size image)

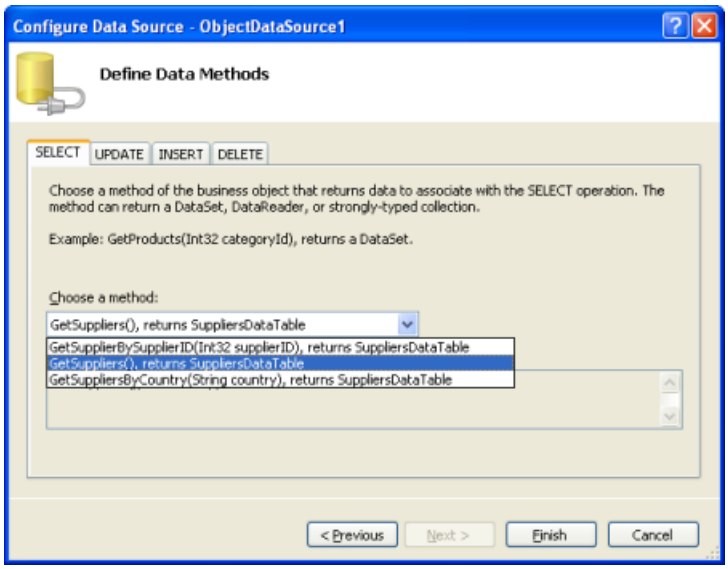


Figure 3: Configure the ObjectDataSource to Use the `GetSuppliers()` Method (Click to view full-size image)

We need to include a link titled View Products in each GridView row that, when clicked, takes the user to `ProductsForSupplierDetails.aspx` passing in the selected row's `SupplierID` value through the querystring. For example, if user clicks on the View Products link for the Tokyo Traders supplier (which has a `SupplierID` value of 4), they should be sent to `ProductsForSupplierDetails.aspx?SupplierID=4`.

To accomplish this, add a HyperLinkField to the GridView, which adds a hyperlink to each GridView row. Start by clicking the Edit Columns link from the GridView's smart tag. Next, select the HyperLinkField from the list in the upper left and click Add to include the HyperLinkField in the GridView's field list.

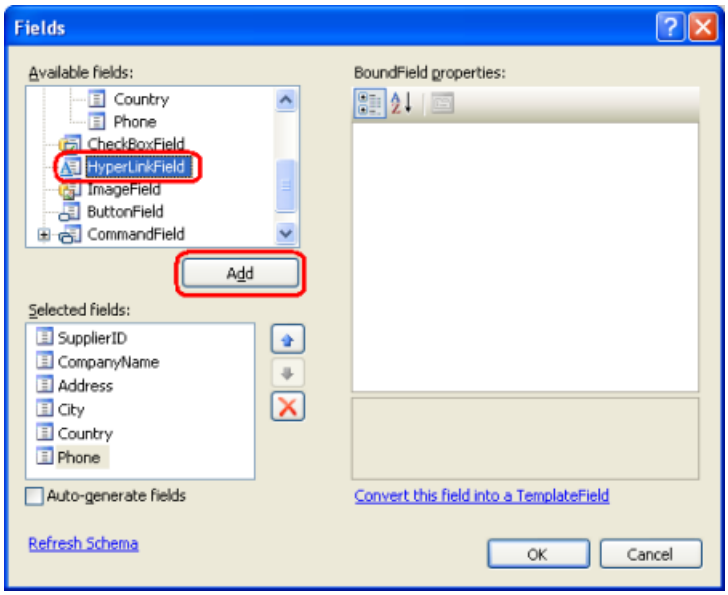


Figure 4: Add a HyperLinkField to the GridView (Click to view full-size image)

The HyperLinkField can be configured to use the same text or URL values the link in each GridView row, or can base these values on the data values bound to each particular row. To specify a static value across all rows use the HyperLinkField's `Text` or `NavigateUrl` properties. Since we want the link text to be the same for all rows, set the HyperLinkField's `Text` property to View Products.

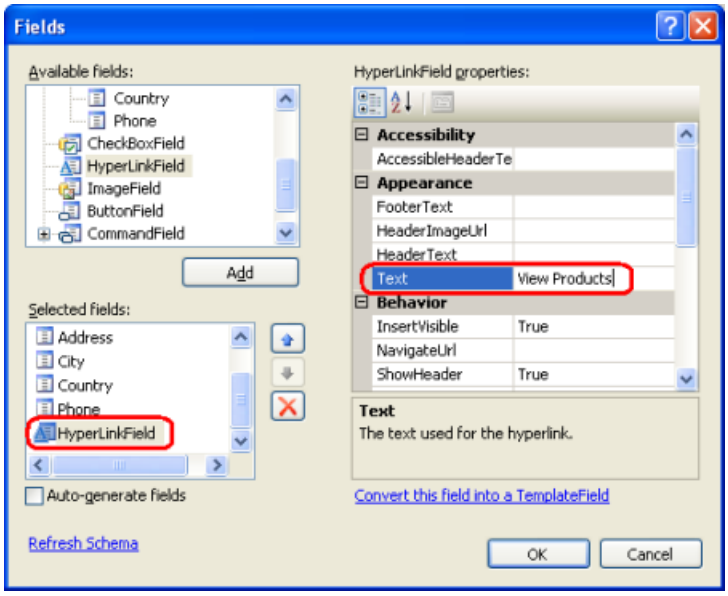


Figure 5: Set the HyperLinkField's `Text` Property to View Products (Click to view full-size image)

To set the text or URL values to be based on the underlying data bound to the GridView row, specify the data fields the text or URL values should be pulled from in the `DataTextField` or `DataNavigateUrlFields` properties. `DataTextField` can only be set to a single data field; `DataNavigateUrlFields`, however, can be set to a comma-delimited list of data fields. We frequently need to base the text or URL on a combination of the current row's data field value and some static markup. In this tutorial, for example, we want the URL of the HyperLinkField's links to be `ProductsForSupplierDetails.aspx?SupplierID=supplierID`, where *`supplierID`* is each GridView's row's `SupplierID` value. Notice that we need both static and data-driven values here: the `ProductsForSupplierDetails.aspx?SupplierID=` portion of the link's URL is static, whereas the *`supplierID`* portion is data-driven as its value is each row's own `SupplierID` value.

To indicate a combination of static and data-driven values, use the `DataTextFormatString` and `DataNavigateUrlFormatString` properties. In these properties enter the static markup as needed and then use the marker `{0}` where you want the value of the field specified in the `DataTextField` or `DataNavigateUrlFields`

properties to appear. If the `DataNavigateUrlFields` property has multiple fields specified use `{0}` where you want the first field value inserted, `{1}` for the second field value, and so on.

Applying this to our tutorial, we need to set the `DataNavigateUrlFields` property to `SupplierID`, since that's the data field whose value we need to customize on a per-row basis, and the `DataNavigateUrlFormatString` property to `ProductsForSupplierDetails.aspx?SupplierID={0}`.

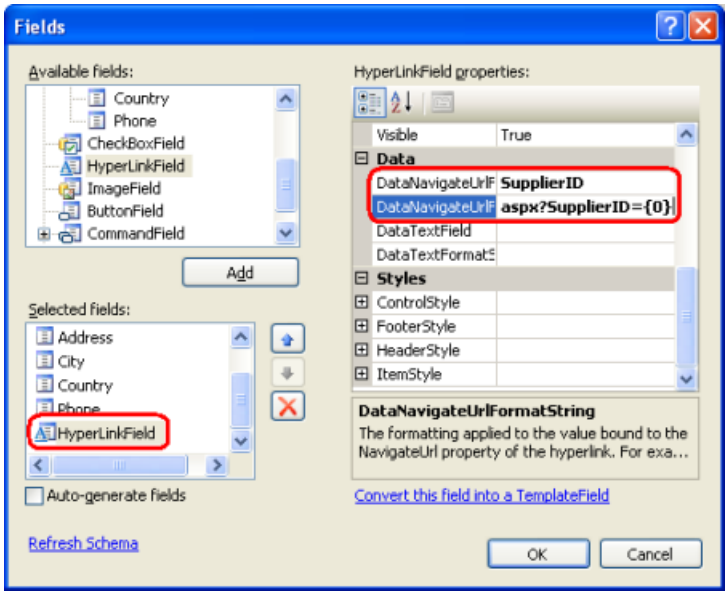


Figure 6: Configure the HyperLinkField to Include the Proper Link URL Based Upon the `SupplierID` (Click to view full-size image)

After adding the HyperLinkField, feel free to customize and reorder the GridView's fields. The following markup shows the GridView after I've made some minor field-level customizations.

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="SupplierID" DataSourceID="ObjectDataSource1"
    EnableViewState="False">
    <Columns>
        <asp:HyperLinkField DataNavigateUrlFields="SupplierID"
          DataNavigateUrlFormatString=
          "ProductsForSupplierDetails.aspx?SupplierID={0}"
            Text="View Products" />
        <asp:BoundField DataField="CompanyName"
            HeaderText="Company" SortExpression="CompanyName" />
        <asp:BoundField DataField="City" HeaderText="City"
            SortExpression="City" />
        <asp:BoundField DataField="Country"
            HeaderText="Country" SortExpression="Country" />
    </Columns>
</asp:GridView>
```

Take a moment to view the `SupplierListMaster.aspx` page through a browser. As Figure 7 shows, the page currently lists all of the suppliers including a View Products link. Clicking on the View Products link will take you to `ProductsForSupplierDetails.aspx`, passing along the supplier's `SupplierID` in the querystring.

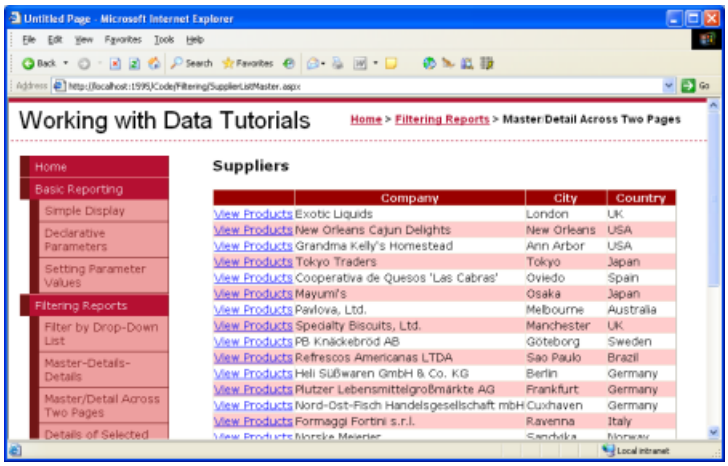


Figure 7: Each Supplier Row Contains a View Products Link (Click to view full-size image)

Step 3: Listing the Supplier's Products in `ProductsForSupplierDetails.aspx`

At this point the `SupplierListMaster.aspx` page is sending users to `ProductsForSupplierDetails.aspx`, passing the selected supplier's `SupplierID` in the querystring. The tutorial's final step is to display the products in a GridView in `ProductsForSupplierDetails.aspx` whose `SupplierID` equals the `SupplierID` passed in through the querystring. To accomplish this start by adding a GridView to the `ProductsForSupplierDetails.aspx` page, using a new ObjectDataSource control named `ProductsBySupplierDataSource` that invokes the `GetProductsBySupplierID(supplierID)` method from the `ProductsBLL` class.

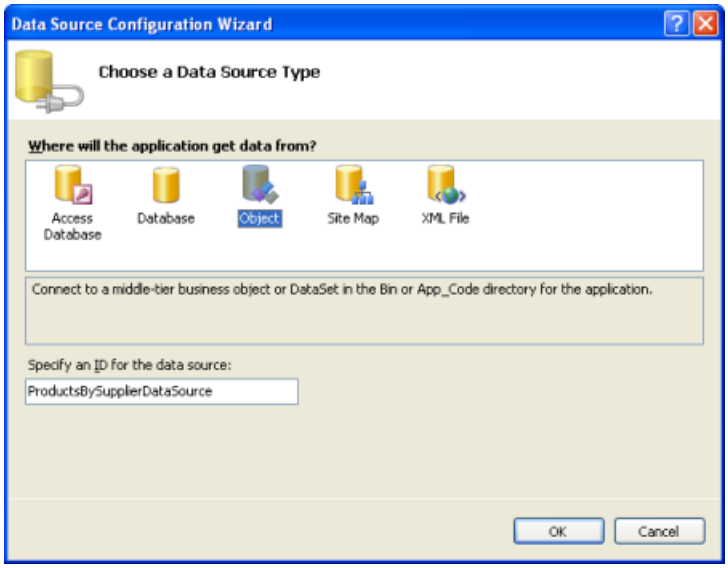


Figure 8: Add a New ObjectDataSource Named `ProductsBySupplierDataSource` (Click to view full-size image)

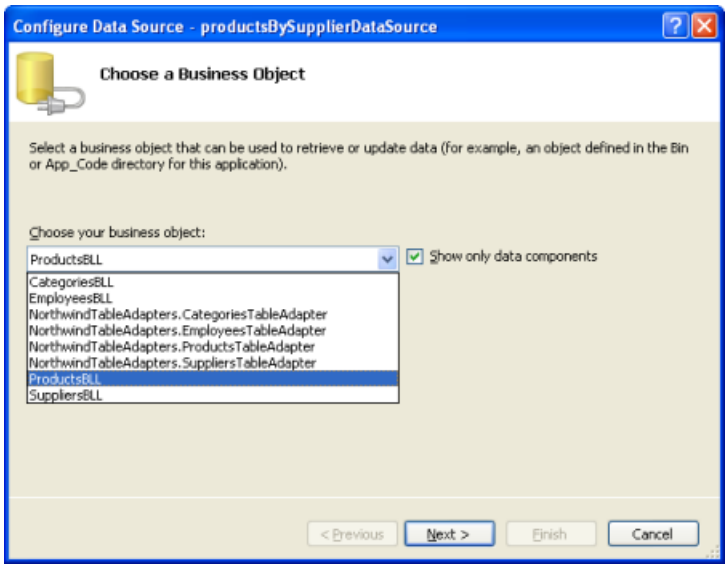


Figure 9: Select the `ProductsBLL` Class (Click to view full-size image)

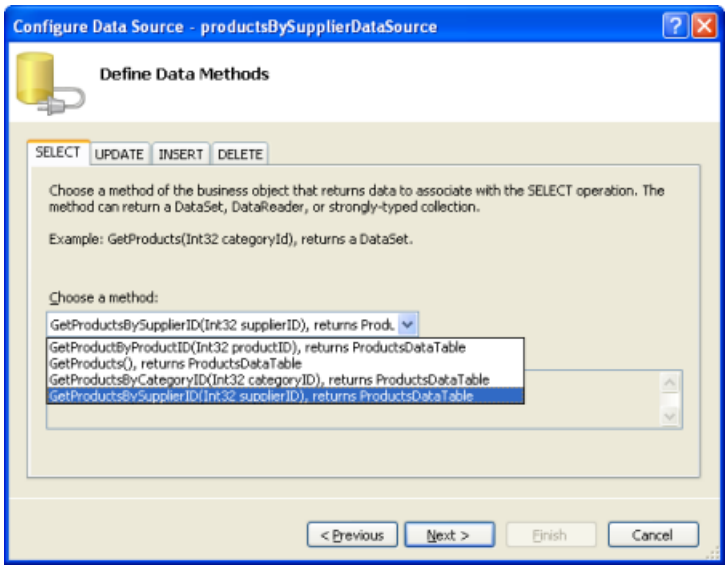


Figure 10: Have the ObjectDataSource Invoke the `GetProductsBySupplierID(supplierID)` Method (Click to view full-size image)

The final step of the Configure Data Source wizard asks us to provide the source of the `GetProductsBySupplierID(supplierID)` method's *`supplierID`* parameter. To use the querystring value, set the Parameter source to QueryString and enter the name of the querystring value to use in the QueryStringField textbox (`SupplierID`).

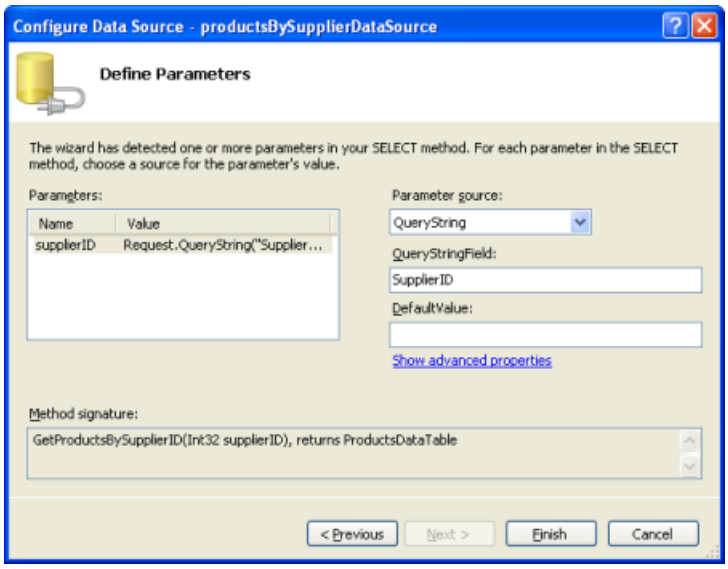


Figure 11: Populate the `supplierID` Parameter Value from the `SupplierID` Querystring Value (Click to view full-size image)

That's all there is to it! Figure 12 shows the `ProductsForSupplierDetails.aspx` page when visited by clicking the Tokyo Traders link from `SupplierListMaster.aspx`.

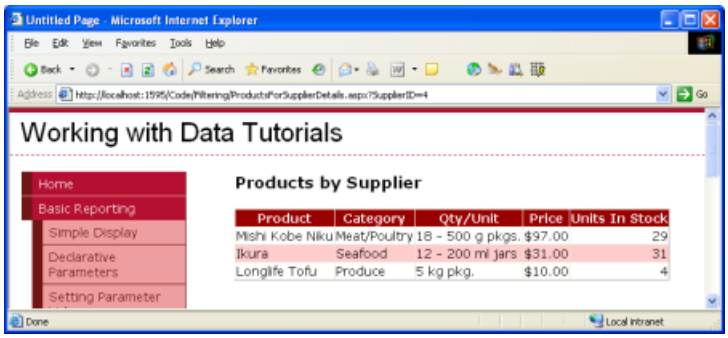


Figure 12: The Products Supplied by Tokyo Traders are Shown (Click to view full-size image)

Displaying Supplier Information in `ProductsForSupplierDetails.aspx`

As Figure 12 shows, the `ProductsForSupplierDetails.aspx` page simply lists the products that are supplied by the `SupplierID` specified in the querystring. Someone sent directly to this page, however, would not know that Figure 12 is showing Tokyo Traders' products. To remedy this we can display supplier information in this page as well.

Start by adding a FormView above the products GridView. Create a new ObjectDataSource control named `SuppliersDataSource` that invokes the `SuppliersBLL` class's `GetSupplierBySupplierID(supplierID)` method.

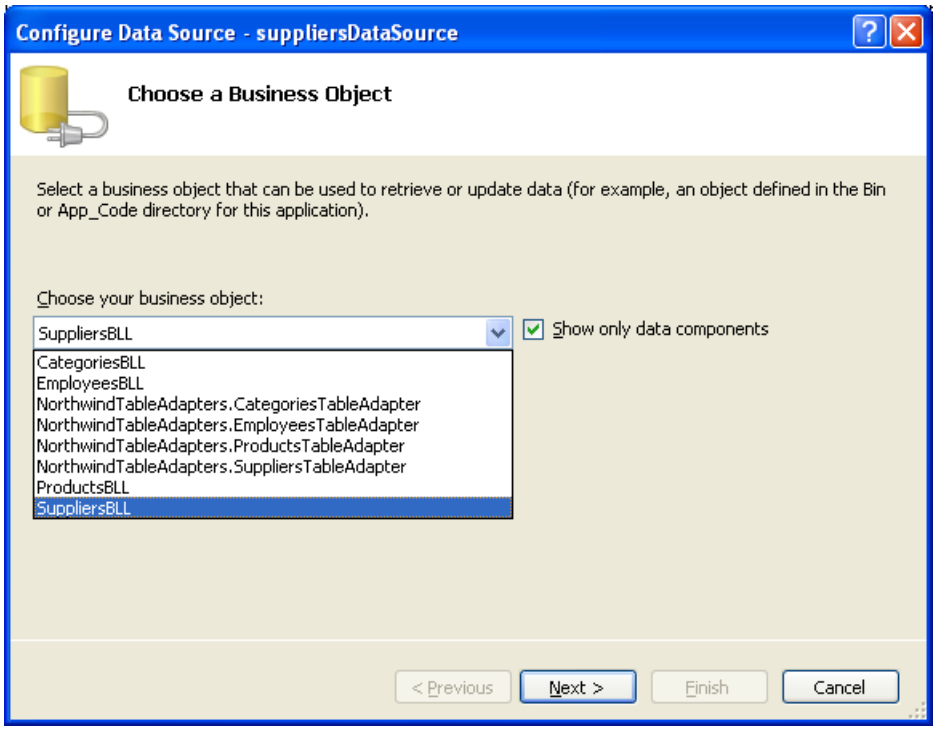


Figure 13: Select the `SuppliersBLL` Class (Click to view full-size image)

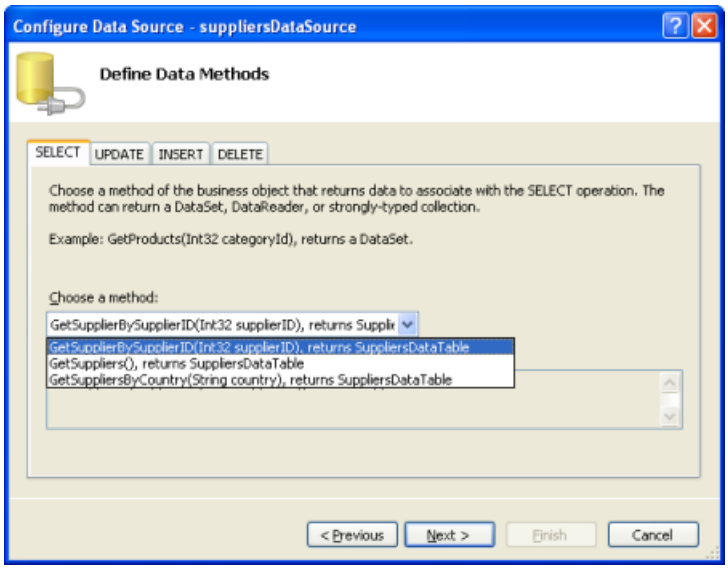


Figure 14: Have the ObjectDataSource Invoke the `GetSupplierBySupplierID(supplierID)` Method (Click to view full-size image)

As with the `ProductsBySupplierDataSource`, have the *`supplierID`* parameter assigned the value of the `SupplierID` querystring value.

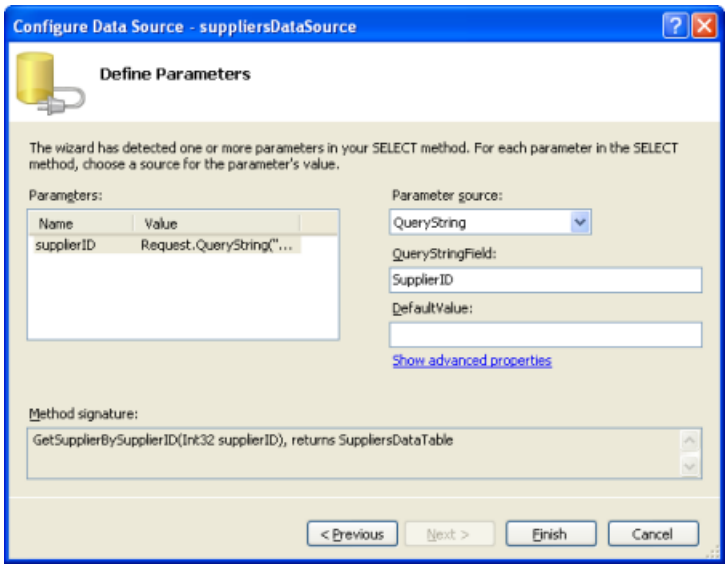


Figure 15: Populate the `supplierID` Parameter Value from the `SupplierID` Querystring Value (Click to view full-size image)

When binding the FormView to the ObjectDataSource in the Design view, Visual Studio will automatically create the FormView's `ItemTemplate`, `InsertItemTemplate`, and `EditItemTemplate` with Label and TextBox Web controls for each of the data fields returned by the ObjectDataSource. Since we just want to display supplier information feel free to remove the `InsertItemTemplate` and `EditItemTemplate`. Next, edit the ItemTemplate so that it displays the supplier's company name in an `<h3>` element and the address, city, country, and phone number beneath the company name. Alternatively, you can manually set the FormView's `DataSourceID` and create the `ItemTemplate` markup, as we did back in the "Displaying Data With the ObjectDataSource" tutorial.

After these edits the FormView's declarative markup should look similar to the following:

```
<asp:FormView ID="FormView1" runat="server" DataKeyNames="SupplierID"
    DataSourceID="suppliersDataSource" EnableViewState="False">
    <ItemTemplate>
        <h3><%# Eval("CompanyName") %></h3>
        <p>
            <asp:Label ID="AddressLabel" runat="server"
                Text='<%# Bind("Address") %>'></asp:Label><br />
            <asp:Label ID="CityLabel" runat="server"
                Text='<%# Bind("City") %>'></asp:Label>,
            <asp:Label ID="CountryLabel" runat="server"
                Text='<%# Bind("Country") %>'></asp:Label><br />
            Phone:
            <asp:Label ID="PhoneLabel" runat="server"
                Text='<%# Bind("Phone") %>'></asp:Label>
        </p>
    </ItemTemplate>
</asp:FormView>
```

Figure 16 shows a screen shot of the `ProductsForSupplierDetails.aspx` page after the supplier information detailed above has been included.

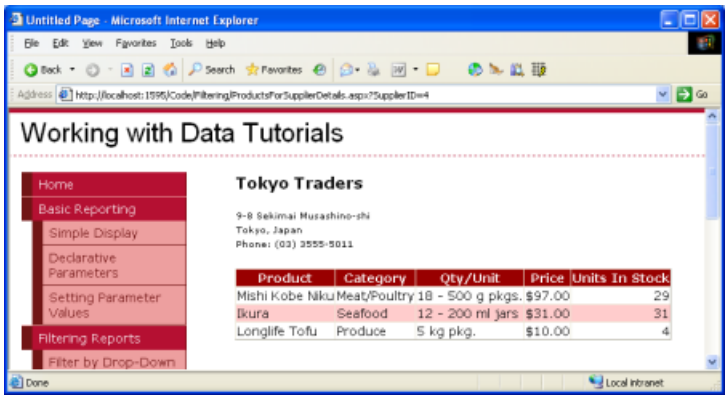


Figure 16: The List of Products Includes a Summary About the Supplier (Click to view full-size image)

Applying the Final Touches for the `ProductsForSupplierDetails.aspx` UI

To improve the user experience for this report there are a couple of additions we ought to make to the `ProductsForSupplierDetails.aspx` page. Currently the only way a user can go from the `ProductsForSupplierDetails.aspx` page back to the list of suppliers is to click their browser's back button. Let's add a HyperLink control to the `ProductsForSupplierDetails.aspx` page that links back to `SupplierListMaster.aspx`, providing another way for the user to return to the master list.

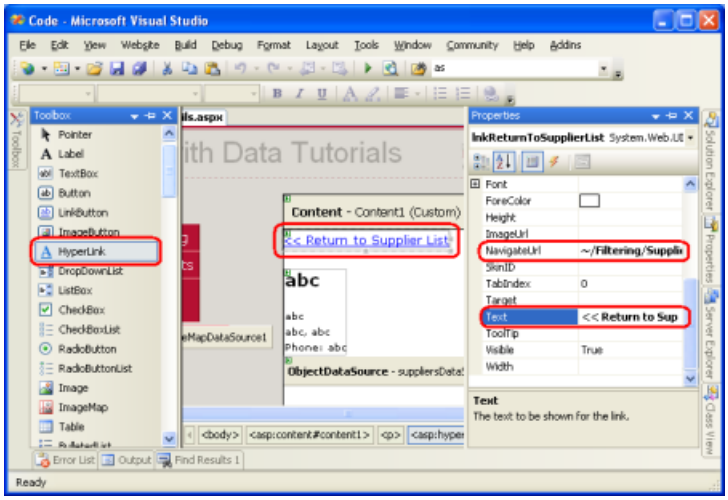


Figure 17: Add a HyperLink Control to Take the User Back to `SupplierListMaster.aspx` (Click to view full-size image)

If the user clicks on the View Products link for a supplier that doesn't have any products, the `ProductsBySupplierDataSource` ObjectDataSource in `ProductsForSupplierDetails.aspx` won't return any results. The GridView bound to the ObjectDataSource won't render any markup resulting in a blank region on the page in the user's browser. To more clearly communicate to the user that there are no products associated with the selected supplier we can set the GridView's `EmptyDataText` property to the message we want displayed when such a situation arises. I've set this property to "There are no products provided by this supplier'"

By default, all suppliers in the Northwinds database provide at least one product. However, for this tutorial I've manually modified the `Products` table so that the supplier Escargots Nouveaux is no longer associated with any products. Figure 18 shows the details page for Escargots Nouveaux after this change has been made.

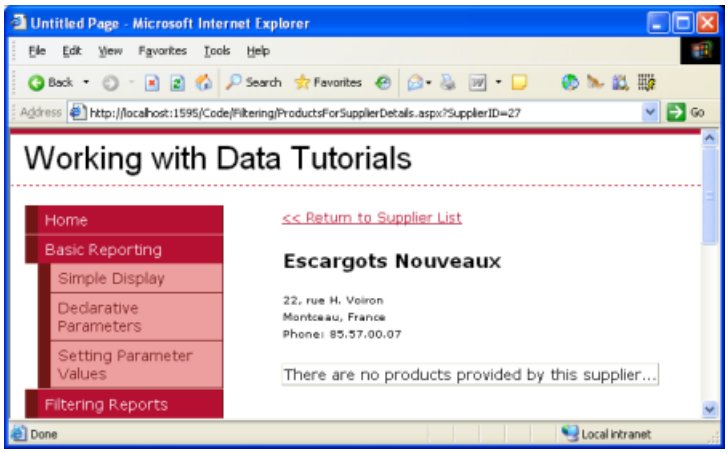


Figure 18: Users are Informed that the Supplier Doesn't Provide Any Products (Click to view full-size image)

Summary

While master/detail reports can display both the master and detail records on a single page, in many websites they are separated out across two web pages. In this tutorial we looked at how to implement such a master/detail report by having the suppliers listed in a GridView in the "master" web page and the associated products listed in the "details" page. Each supplier row in the master web page contained a link to the details page that passed along the row's `SupplierID` value. Such row-specific links can be easily added using the GridView's HyperLinkField.

In the details page retrieving those products for the specified supplier was accomplished by invoking the `ProductsBLL` class's `GetProductsBySupplierID(supplierID)` method. The *`supplierID`* parameter value was specified declaratively using the querystring as the parameter source. We also looked at how to display the supplier details in the details page using a FormView.

Our next tutorial is the final one on master/detail reports. We'll look at how to display a list of products in a GridView where each row has a Select button. Clicking on the Select button will display that product's details in a DetailsView control on the same page.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Master/Detail Using a Selectable Master GridView with a Details DetailView (VB)

2/20/2020 • 7 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> This tutorial will have a GridView whose rows include the name and price of each product along with a Select button. Clicking the Select button for a particular product will cause its full details to be displayed in a DetailsView control on the same page.

Introduction

In the previous tutorial we saw how to create a master/detail report using two web pages: a "master" web page, from which we displayed the list of suppliers; and a "details" web page that listed those products provided by the selected supplier. This two page report format can be condensed into one page. This tutorial will have a GridView whose rows include the name and price of each product along with a Select button. Clicking the Select button for a particular product will cause its full details to be displayed in a DetailsView control on the same page.

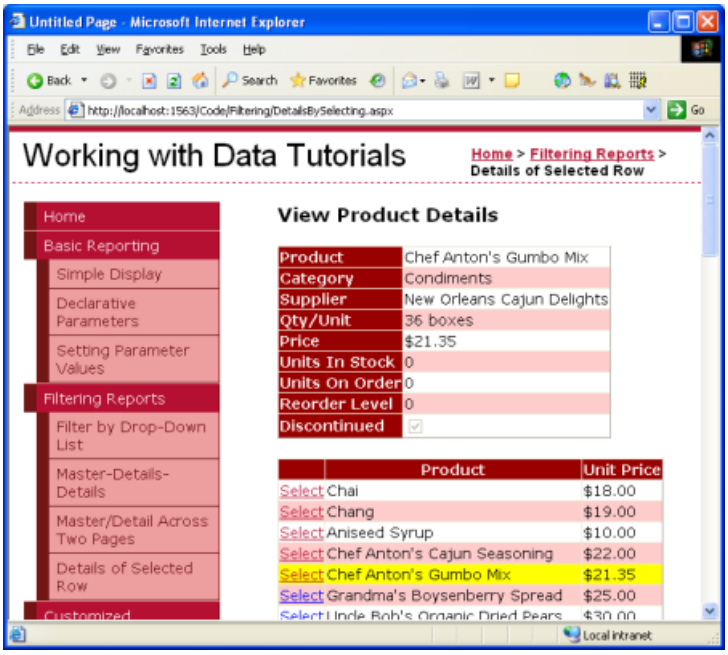


Figure 1: Clicking the Select Button Displays the Product's Details (Click to view full-size image)

Step 1: Creating a Selectable GridView

Recall that in the two-page master/detail report that each master record included a hyperlink that, when clicked, sent the user to the details page passing the clicked row's `SupplierID` value in the querystring. Such a hyperlink was added to each GridView row using a HyperLinkField. For the single page master/details report, we will need a Button for each GridView row that, when clicked, shows the details. The GridView control can be configured to include a Select button for each row that causes a postback and marks that row as the GridView's SelectedRow.

Start by adding a GridView control to the `DetailsBySelecting.aspx` page in the `Filtering` folder, setting its `ID` property to `ProductsGrid`. Next, add a new ObjectDataSource named `AllProductsDataSource` that invokes the

`ProductsBLL` class's `GetProducts()` method.

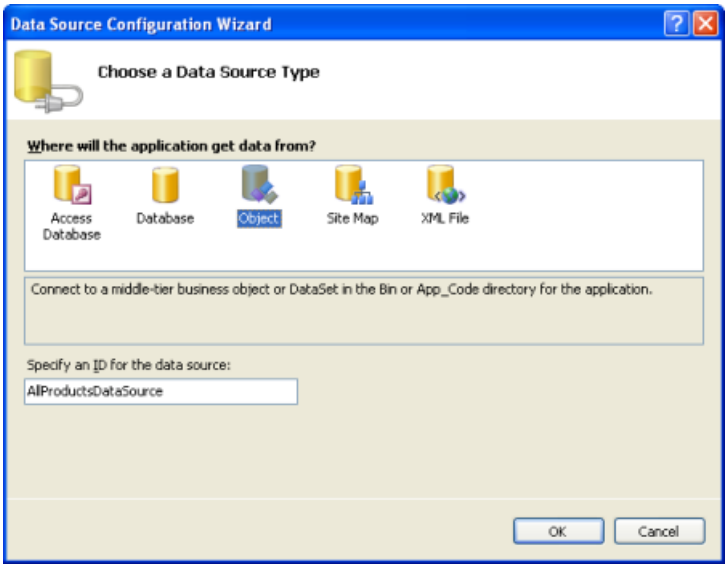


Figure 2: Create an ObjectDataSource Named `AllProductsDataSource` (Click to view full-size image)

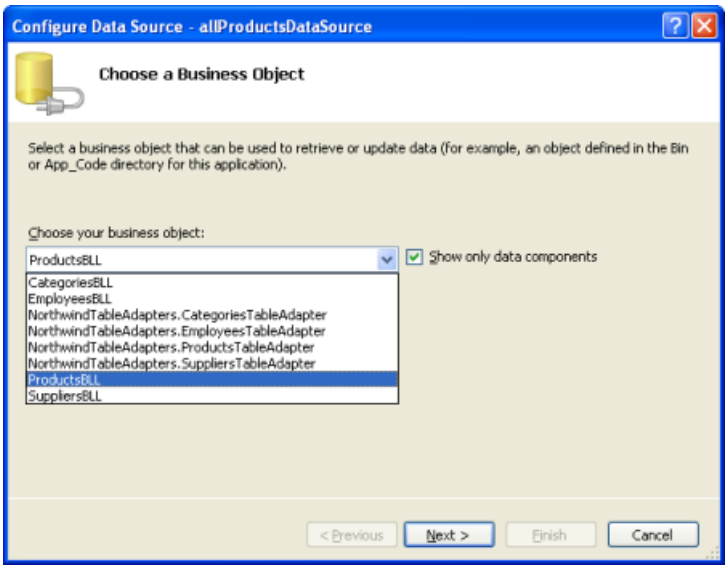


Figure 3: Use the `ProductsBLL` Class (Click to view full-size image)

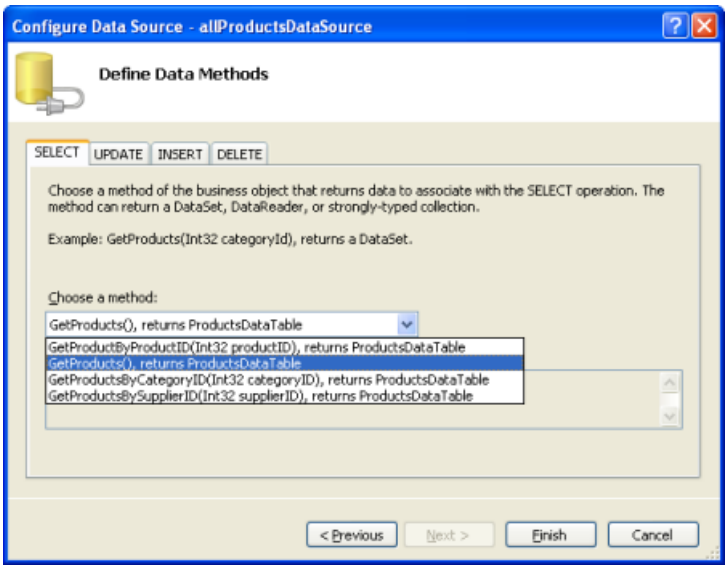


Figure 4: Configure the ObjectDataSource to Invoke the `GetProducts()` Method (Click to view full-size image)

Edit the GridView's fields removing all but the `ProductName` and `UnitPrice` BoundFields. Also, feel free to

customize these BoundFields as needed, such as formatting the `UnitPrice` BoundField as a currency and changing the `HeaderText` properties of the BoundFields. These steps can be accomplished graphically, by clicking the Edit Columns link from the GridView's smart tag, or by manually configuring the declarative syntax.

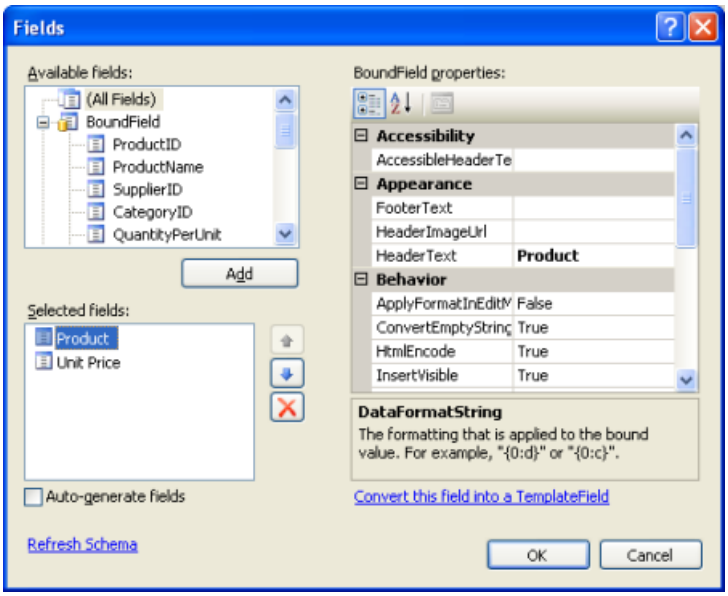


Figure 5: Remove All But the `ProductName` and `UnitPrice` BoundFields (Click to view full-size image)

The final markup for the GridView is:

```
<asp:GridView ID="ProductsGrid" runat="server"
  AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="AllProductsDataSource" EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductName"
         HeaderText="Product" SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice"
            DataFormatString="{0:c}" HeaderText="Unit Price"
            HtmlEncode="False" SortExpression="UnitPrice" />
    </Columns>
</asp:GridView>
```

Next, we need to mark the GridView as selectable, which will add a Select button to each row. To accomplish this, simply check the Enable Selection checkbox in the GridView's smart tag.

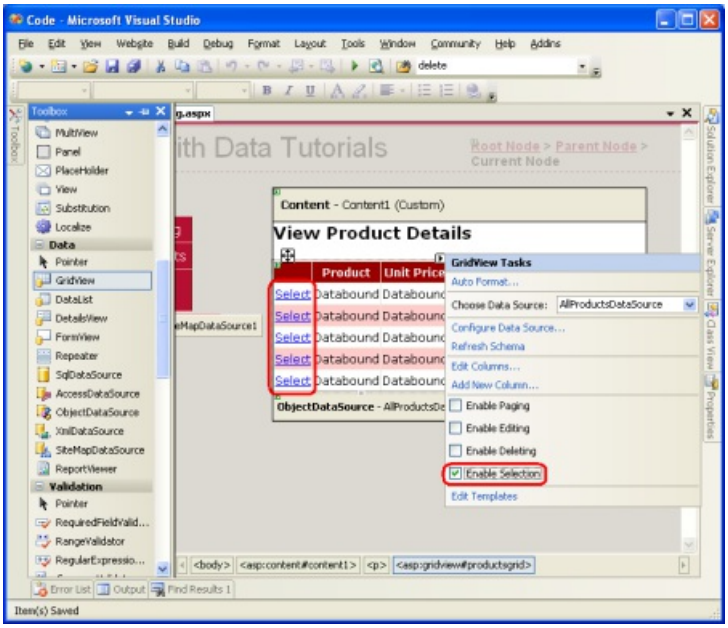


Figure 6: Make the GridView's Rows Selectable (Click to view full-size image)

Checking the Enabling Selection option adds a CommandField to the `ProductsGrid` GridView with its `ShowSelectButton` property set to True. This results in a Select button for each row of the GridView, as Figure 6 illustrates. By default, the Select buttons are rendered as LinkButtons, but you can use Buttons or ImageButtons instead through the CommandField's `ButtonType` property.

```
<asp:GridView ID="ProductsGrid" runat="server"
  AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="AllProductsDataSource" EnableViewState="False">
    <Columns>
        <asp:CommandField ShowSelectButton="True" />
        <asp:BoundField DataField="ProductName"
          HeaderText="Product" SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice"
          DataFormatString="{0:c}" HeaderText="Unit Price"
            HtmlEncode="False" SortExpression="UnitPrice" />
    </Columns>
</asp:GridView>
```

When a GridView row's Select button is clicked a postback ensues and the GridView's `SelectedRow` property is updated. In addition to the `SelectedRow` property, the GridView provides the SelectedIndex, SelectedValue, and SelectedDataKey properties. The `SelectedIndex` property returns the index of the selected row, whereas the `SelectedValue` and `SelectedDataKey` properties return values based upon the GridView's DataKeyNames property.

The `DataKeyNames` property is used to associate one or more data field values with each row and is commonly used to attribute uniquely identifying information from the underlying data with each GridView row. The `SelectedValue` property returns the value of the first `DataKeyNames` data field for the selected row where as the `SelectedDataKey` property returns the selected row's `DataKey` object, which contains all of the values for the specified data key fields for that row.

The `DataKeyNames` property is automatically set to the uniquely-identifying data field(s) when you bind a data source to a GridView, DetailsView, or FormView through the Designer. While this property has been set for us automatically in the preceding tutorials, the examples would have worked without the `DataKeyNames` property specified. However, for the selectable GridView in this tutorial, as well as for future tutorials in which we'll be examining inserting, updating, and deleting, the `DataKeyNames` property must be set properly. Take a moment to ensure that your GridView's `DataKeyNames` property is set to `ProductID`.

Let's view our progress thus far through a browser. Note that the GridView lists the name and price for all of the products along with a Select LinkButton. Clicking the Select button causes a postback. In Step 2 we'll see how to have a DetailsView respond to this postback by displaying the details for the selected product.

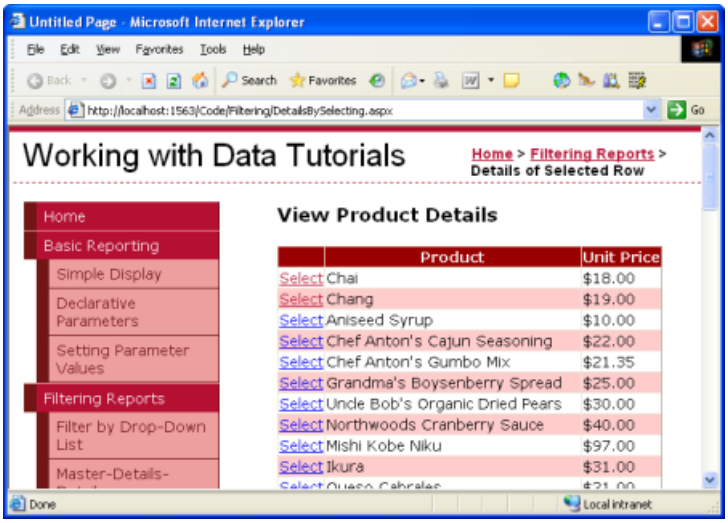


Figure 7: Each Product Row Contains a Select LinkButton (Click to view full-size image)

Highlighting the Selected Row

The `ProductsGrid` GridView has a `SelectedRowStyle` property that can be used to dictate the visual style for the selected row. Used properly, this can improve the user's experience by more clearly showing which row of the GridView is currently selected. For this tutorial, let's have the selected row be highlighted with a yellow background.

As with our earlier tutorials, let's strive to keep the aesthetic-related settings defined as CSS classes. Therefore, create a new CSS class in `Styles.css` named `SelectedRowStyle`.

```
.SelectedRowStyle
{
    background-color: Yellow;
}
```

To apply this CSS class to the `SelectedRowStyle` property of *all* GridViews in our tutorial series, edit the `GridView.skin` Skin in the `DataWebControls` Theme to include the `SelectedRowStyle` settings as shown below:

```
<asp:GridView runat="server" CssClass="DataWebControlStyle">
   <AlternatingRowStyle CssClass="AlternatingRowStyle" />
   <RowStyle CssClass="RowStyle" />
   <HeaderStyle CssClass="HeaderStyle" />
   <SelectedRowStyle CssClass="SelectedRowStyle" />
</asp:GridView>
```

With this addition, the selected GridView row is now highlighted with a yellow background color.

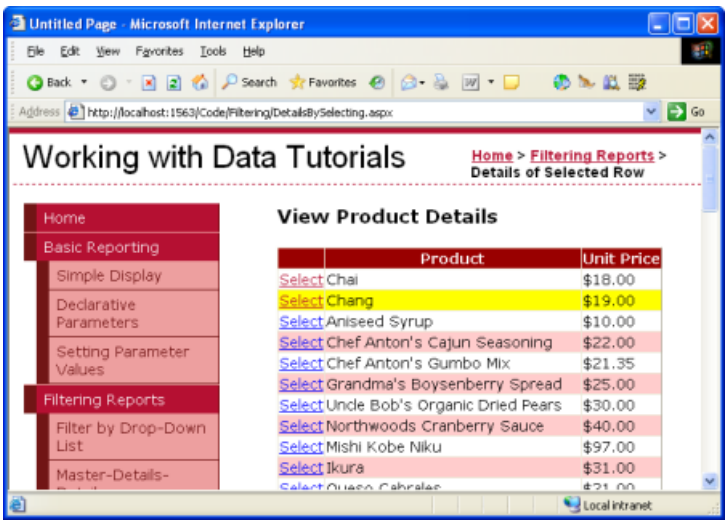


Figure 8: Customize the Selected Row's Appearance Using the GridView's `SelectedRowStyle` Property (Click to view full-size image)

Step 2: Displaying the Selected Product's Details in a DetailsView

With the `ProductsGrid` GridView complete, all that remains is to add a DetailsView that displays information about the particular product selected. Add a DetailsView control above the GridView and create a new ObjectDataSource named `ProductDetailsDataSource`. Since we want this DetailsView to display particular information about the selected product, configure the `ProductDetailsDataSource` to use the `ProductsBLL` class's `GetProductByProductID(productID)` method.

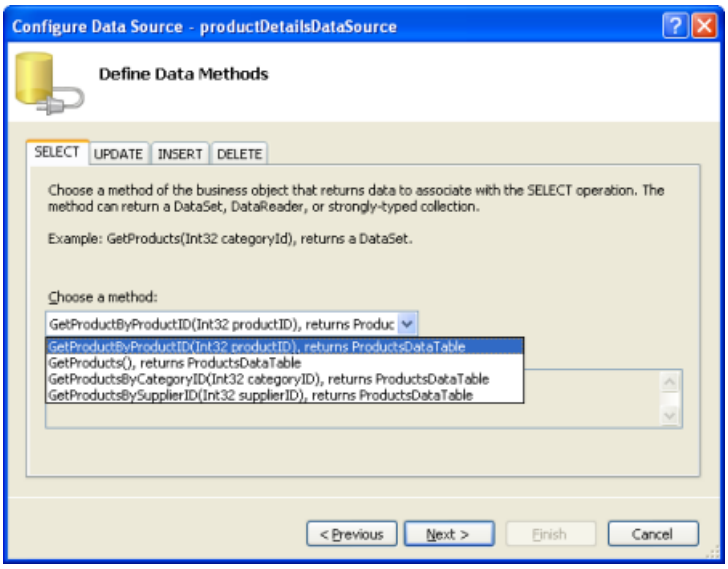


Figure 9: Invoke the `ProductsBLL` Class's `GetProductByProductID(productID)` Method (Click to view full-size image)

Have the *`productID`* parameter's value obtained from the GridView control's `SelectedValue` property. As we discussed earlier, the GridView's `SelectedValue` property returns the first data key value for the selected row. Therefore, it's imperative that the GridView's `DataKeyNames` property is set to `ProductID`, so that the selected row's `ProductID` value is returned by `SelectedValue`.

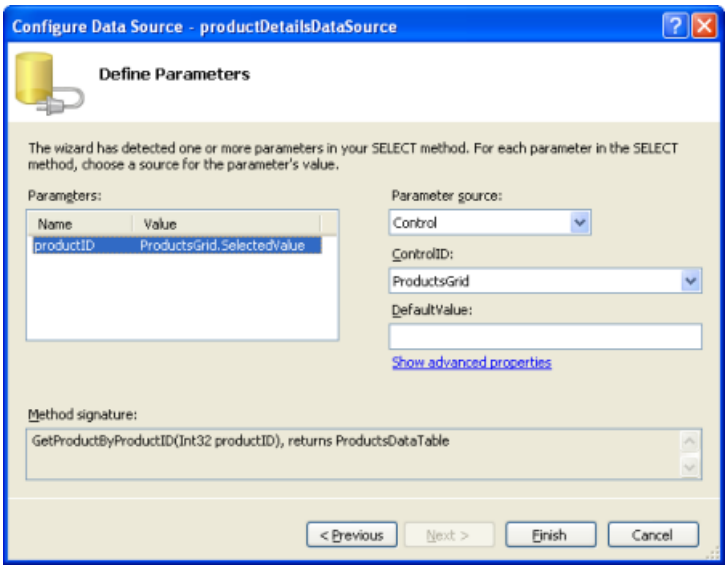


Figure 10: Set the *`productID`* Parameter to the GridView's `SelectedValue` Property (Click to view full-size image)

Once the `productDetailsDataSource` ObjectDataSource has been configured correctly and bound to the DetailsView, this tutorial is complete! When the page is first visited no row is selected, so the GridView's `SelectedValue` property returns `Nothing`. Since there are no products with a `NULL` `ProductID` value, no records are returned by the `GetProductByProductID(productID)` method, meaning that the DetailsView isn't displayed (see Figure 11). Upon clicking a GridView row's Select button, a postback ensues and the DetailsView is refreshed. This time the GridView's `SelectedValue` property returns the `ProductID` of the selected row, the `GetProductByProductID(productID)` method returns a `ProductsDataTable` with information about that particular product, and the DetailsView shows these details (see Figure 12).

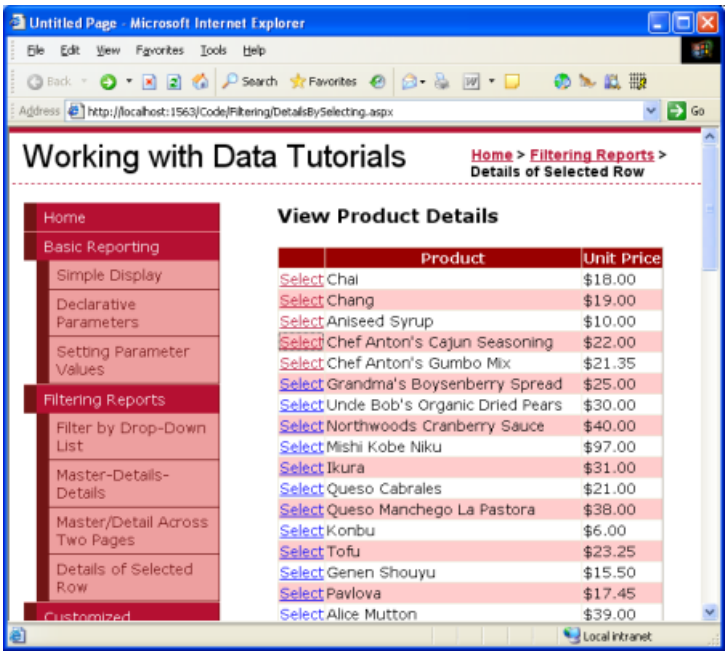


Figure 11: When First Visited, Only the GridView is Displayed (Click to view full-size image)

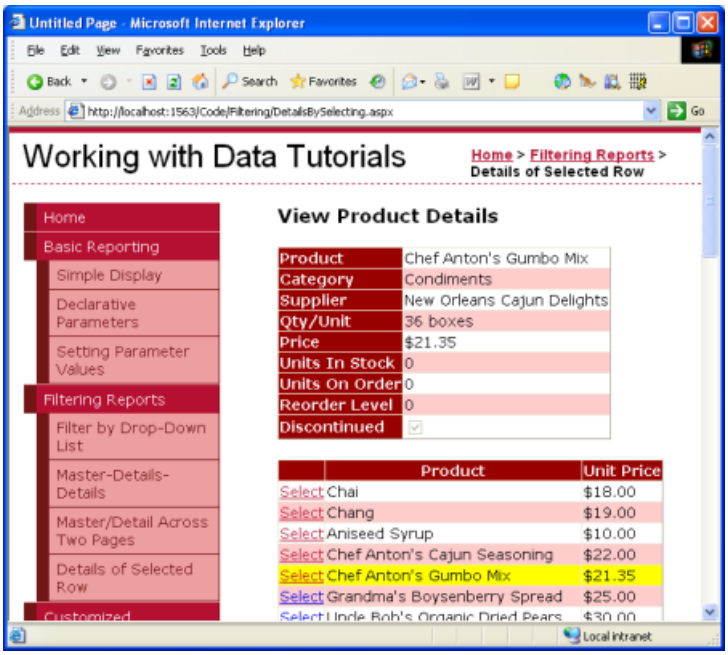


Figure 12: Upon Selecting a Row, the Product's Details are Displayed (Click to view full-size image)

Summary

In this and the preceding three tutorials we've seen a number of techniques for displaying master/detail reports. In this tutorial we examined using a selectable GridView to house the master records and a DetailsView to display details about the selected master record on the same page. In the earlier tutorials we looked at how to display master/details reports using DropDownLists and displaying master records on one web page and detail records on another.

This tutorial concludes our examination of master/detail reports. Starting with the next tutorial we'll begin our exploration of customized formatting with the GridView, DetailsView, and FormView. We'll see how to customize the appearance of these controls based on the data bound to them, how to summarize data in the GridView's footer, and how to use templates to obtain a greater degree of control over the layout.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Custom Formatting

2/20/2020 • 2 minutes to read • Edit Online

These tutorials look at how to customize data display during data binding by using event handlers in the GridView, DetailsView, and FormView controls.

- Custom Formatting Based Upon Data (C#)
- Using TemplateFields in the GridView Control (C#)
- Using TemplateFields in the DetailsView Control (C#)
- FormView's Templates (C#)
- Displaying Summary Information in the GridView's Footer (C#)
- Custom Formatting Based Upon Data (VB)
- Using TemplateFields in the GridView Control (VB)
- Using TemplateFields in the DetailsView Control (VB)
- FormView's Templates (VB)
- Displaying Summary Information in the GridView's Footer (VB)

Custom Formatting Based Upon Data (C#)

2/20/2020 • 18 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> Adjusting the format of the GridView, DetailsView, or FormView based upon the data bound to it can be accomplished in multiple ways. In this tutorial we'll look at how to accomplish data bound formatting through the use of the DataBound and RowDataBound event handlers.

Introduction

The appearance of the GridView, DetailsView, and FormView controls can be customized through a myriad of style-related properties. Properties like `CssClass`, `Font`, `BorderWidth`, `BorderStyle`, `BorderColor`, `Width`, and `Height`, among others, dictate the general appearance of the rendered control. Properties including `HeaderStyle`, `RowStyle`, `AlternatingRowStyle`, and others allow these same style settings to be applied to particular sections. Likewise, these style settings can be applied at the field level.

In many scenarios though, the formatting requirements depend upon the value of the displayed data. For example, to draw attention to out of stock products, a report listing product information might set the background color to yellow for those products whose `UnitsInStock` and `UnitsOnOrder` fields are both equal to 0. To highlight the more expensive products, we may want to display the prices of those products costing more than $75.00 in a bold font.

Adjusting the format of the GridView, DetailsView, or FormView based upon the data bound to it can be accomplished in multiple ways. In this tutorial we'll look at how to accomplish data bound formatting through the use of the `DataBound` and `RowDataBound` event handlers. In the next tutorial we'll explore an alternative approach.

Using the DetailsView Control's `DataBound` Event Handler

When data is bound to a DetailsView, either from a data source control or through programmatically assigning data to the control's `DataSource` property and calling its `DataBind()` method, the following sequence of steps occur:

1. The data Web control's `DataBinding` event fires.
2. The data is bound to the data Web control.
3. The data Web control's `DataBound` event fires.

Custom logic can be injected immediately after steps 1 and 3 through an event handler. By creating an event handler for the `DataBound` event we can programmatically determine the data that has been bound to the data Web control and adjust the formatting as needed. To illustrate this let's create a DetailsView that will list general information about a product, but will display the `UnitPrice` value in a ***bold, italic font*** if it exceeds $75.00.

Step 1: Displaying the Product Information in a DetailsView

Open the `CustomColors.aspx` page in the `CustomFormatting` folder, drag a DetailsView control from the Toolbox onto the Designer, set its `ID` property value to `ExpensiveProductsPriceInBoldItalic`, and bind it to a new ObjectDataSource control that invokes the `ProductsBLL` class's `GetProducts()` method. The detailed steps for

accomplishing this are omitted here for brevity since we examined them in detail in previous tutorials.

Once you've bound the ObjectDataSource to the DetailsView, take a moment to modify the field list. I've opted to remove the `ProductID`, `SupplierID`, `CategoryID`, `UnitsInStock`, `UnitsOnOrder`, `ReorderLevel`, and `Discontinued` BoundFields and renamed and reformatted the remaining BoundFields. I also cleared out the `Width` and `Height` settings. Since the DetailsView displays only a single record, we need to enable paging in order to allow the end user to view all of the products. Do so by checking the Enable Paging checkbox in the DetailsView's smart tag.

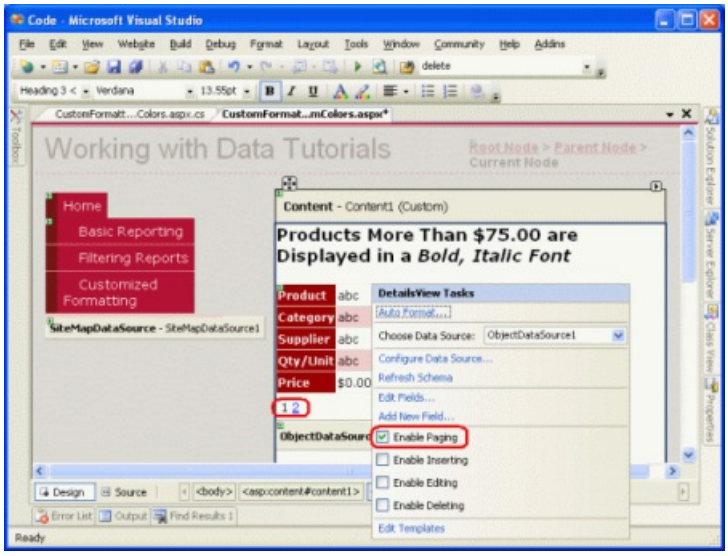


Figure 1: Check the Enable Paging Checkbox in the DetailsView's Smart Tag (Click to view full-size image)

After these changes, the DetailsView markup will be:

```
<asp:DetailsView ID="DetailsView1" runat="server" AllowPaging="True"
    AutoGenerateRows="False" DataKeyNames="ProductID"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <Fields>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
          SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
          ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
          ReadOnly="True" SortExpression="SupplierName" />
        <asp:BoundField DataField="QuantityPerUnit"
          HeaderText="Qty/Unit" SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
          HeaderText="Price"
            HtmlEncode="False" SortExpression="UnitPrice" />
    </Fields>
</asp:DetailsView>
```

Take a moment to test out this page in your browser.

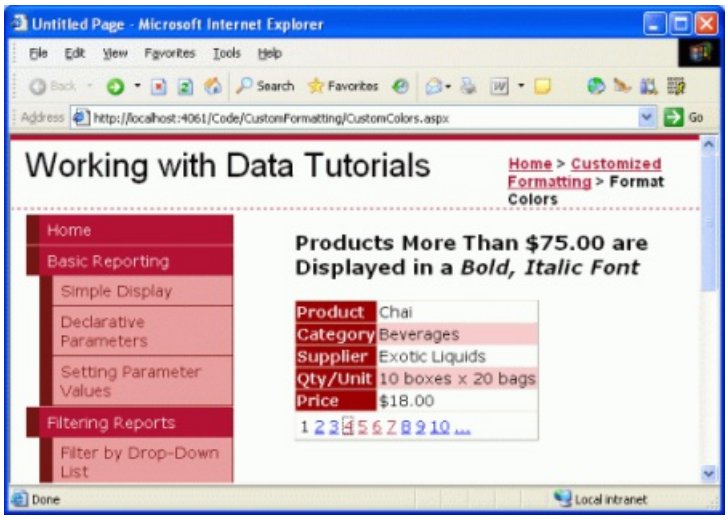


Figure 2: The DetailsView Control Displays One Product at a Time (Click to view full-size image)

Step 2: Programmatically Determining the Value of the Data in the DataBound Event Handler

In order to display the price in a bold, italic font for those products whose `UnitPrice` value exceeds $75.00, we need to first be able to programmatically determine the `UnitPrice` value. For the DetailsView, this can be accomplished in the `DataBound` event handler. To create the event handler click on the DetailsView in the Designer then navigate to the Properties window. Press F4 to bring it up, if it's not visible, or go to the View menu and select the Properties Window menu option. From the Properties window, click on the lightning bolt icon to list the DetailsView's events. Next, either double-click the `DataBound` event or type in the name of the event handler you want to create.

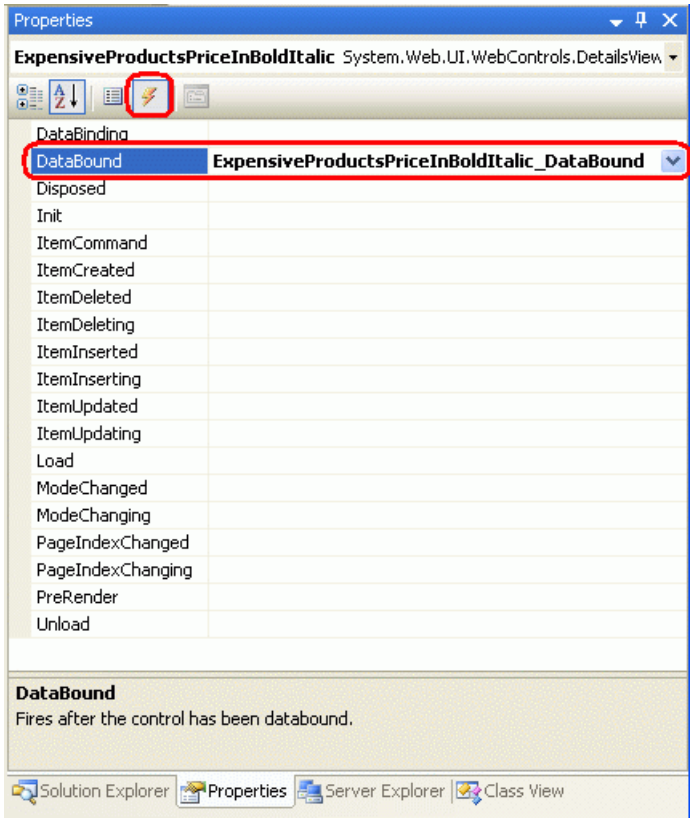


Figure 3: Create an Event Handler for the `DataBound` Event

Doing so will automatically create the event handler and take you to the code portion where it has been added. At this point you will see:

```
protected void ExpensiveProductsPriceInBoldItalic_DataBound(object sender, EventArgs e)
{

}
```

The data bound to the DetailsView can be accessed via the `DataItem` property. Recall that we are binding our controls to a strongly-typed DataTable, which is composed of a collection of strongly-typed DataRow instances. When the DataTable is bound to the DetailsView, the first DataRow in the DataTable is assigned to the DetailsView's `DataItem` property. Specifically, the `DataItem` property is assigned a `DataRowView` object. We can use the `DataRowView`'s `Row` property to get access to the underlying DataRow object, which is actually a `ProductsRow` instance. Once we have this `ProductsRow` instance we can make our decision by simply inspecting the object's property values.

The following code illustrates how to determine whether the `UnitPrice` value bound to the DetailsView control is greater than $75.00:

```
protected void ExpensiveProductsPriceInBoldItalic_DataBound(object sender, EventArgs e)
{
    // Get the ProductsRow object from the DataItem property...
    Northwind.ProductsRow product = (Northwind.ProductsRow)
        ((DataRowView)ExpensiveProductsPriceInBoldItalic.DataItem).Row;
    if (!product.IsUnitPriceNull() && product.UnitPrice > 75m)
    {
        // TODO: Make the UnitPrice text bold and italic
    }
}
```

NOTE

Since `UnitPrice` can have a `NULL` value in the database, we first check to make sure that we're not dealing with a `NULL` value before accessing the `ProductsRow`'s `UnitPrice` property. This check is important because if we attempt to access the `UnitPrice` property when it has a `NULL` value the `ProductsRow` object will throw a StrongTypingException exception.

Step 3: Formatting the UnitPrice Value in the DetailsView

At this point we can determine whether the `UnitPrice` value bound to the DetailsView has a value that exceeds $75.00, but we've yet to see how to programmatically adjust the DetailsView's formatting accordingly. To modify the formatting of an entire row in the DetailsView, programmatically access the row using `DetailsViewID.Rows[index]`; to modify a particular cell, access use `DetailsViewID.Rows[index].Cells[index]`. Once we have a reference to the row or cell we can then adjust its appearance by setting its style-related properties.

Accessing a row programmatically requires that you know the row's index, which starts at 0. The `UnitPrice` row is the fifth row in the DetailsView, giving it an index of 4 and making it programmatically accessible using `ExpensiveProductsPriceInBoldItalic.Rows[4]`. At this point we could have the entire row's content displayed in a bold, italic font by using the following code:

```
ExpensiveProductsPriceInBoldItalic.Rows[4].Font.Bold = true;
ExpensiveProductsPriceInBoldItalic.Rows[4].Font.Italic = true;
```

However, this will make *both* the label (Price) and the value bold and italic. If we want to make just the value bold and italic we need to apply this formatting to the second cell in the row, which can be accomplished using the following:

```
ExpensiveProductsPriceInBoldItalic.Rows[4].Cells[1].Font.Bold = true;
ExpensiveProductsPriceInBoldItalic.Rows[4].Cells[1].Font.Italic = true;
```

Since our tutorials thus far have used stylesheets to maintain a clean separation between the rendered markup and style-related information, rather than setting the specific style properties as shown above let's instead use a CSS class. Open the `Styles.css` stylesheet and add a new CSS class named `ExpensivePriceEmphasis` with the following definition:

```
.ExpensivePriceEmphasis
{
    font-weight: bold;
    font-style: italic;
}
```

Then, in the `DataBound` event handler, set the cell's `CssClass` property to `ExpensivePriceEmphasis`. The following code shows the `DataBound` event handler in its entirety:

```
protected void ExpensiveProductsPriceInBoldItalic_DataBound(object sender, EventArgs e)
{
    // Get the ProductsRow object from the DataItem property...
    Northwind.ProductsRow product = (Northwind.ProductsRow)
        ((DataRowView)ExpensiveProductsPriceInBoldItalic.DataItem).Row;
    if (!product.IsUnitPriceNull() && product.UnitPrice > 75m)
    {
        ExpensiveProductsPriceInBoldItalic.Rows[4].Cells[1].CssClass =
            "ExpensivePriceEmphasis";
    }
}
```

When viewing Chai, which costs less than $75.00, the price is displayed in a normal font (see Figure 4). However, when viewing Mishi Kobe Niku, which has a price of $97.00, the price is displayed in a bold, italic font (see Figure 5).

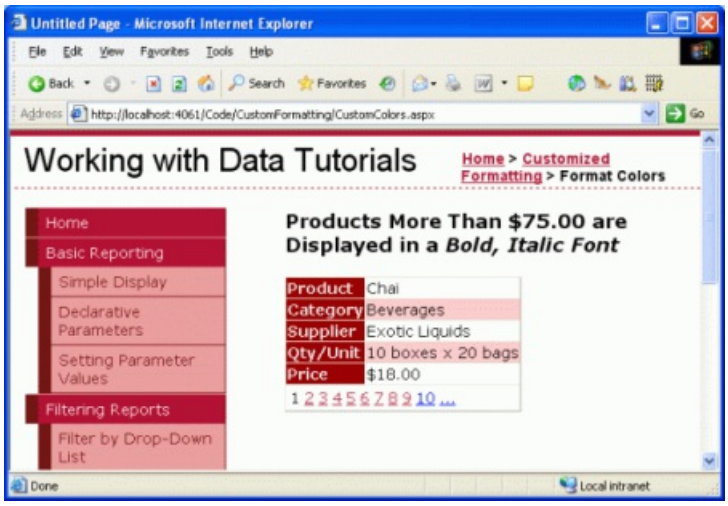


Figure 4: Prices Less than $75.00 are Displayed in a Normal Font (Click to view full-size image)

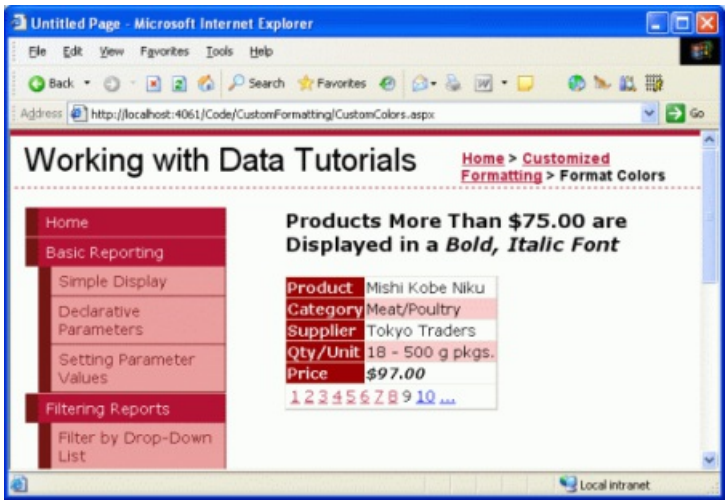


Figure 5: Expensive Products' Prices are Displayed in a Bold, Italic Font (Click to view full-size image)

Using the FormView Control's `DataBound` Event Handler

The steps for determining the underlying data bound to a FormView are identical to those for a DetailsView create a `DataBound` event handler, cast the `DataItem` property to the appropriate object type bound to the control, and determine how to proceed. The FormView and DetailsView differ, however, in how their user interface's appearance is updated.

The FormView does not contain any BoundFields and therefore lacks the `Rows` collection. Instead, a FormView is composed of templates, which can contain a mix of static HTML, Web controls, and databinding syntax. Adjusting the style of a FormView typically involves adjusting the style of one or more of the Web controls within the FormView's templates.

To illustrate this, let's use a FormView to list products like in the previous example, but this time let's display just the product name and units in stock with the units in stock displayed in a red font if it is less than or equal to 10.

Step 4: Displaying the Product Information in a FormView

Add a FormView to the `CustomColors.aspx` page beneath the DetailsView and set its `ID` property to `LowStockedProductsInRed`. Bind the FormView to the ObjectDataSource control created from the previous step. This will create an `ItemTemplate`, `EditItemTemplate`, and `InsertItemTemplate` for the FormView. Remove the `EditItemTemplate` and `InsertItemTemplate` and simplify the `ItemTemplate` to include just the `ProductName` and `UnitsInStock` values, each in their own appropriately-named Label controls. As with the DetailsView from the earlier example, also check the Enable Paging checkbox in the FormView's smart tag.

After these edits your FormView's markup should look similar to the following:

```
<asp:FormView ID="LowStockedProductsInRed" DataKeyNames="ProductID"
    DataSourceID="ObjectDataSource1" AllowPaging="True"
    EnableViewState="False" runat="server">
    <ItemTemplate>
        <b>Product:</b>
        <asp:Label ID="ProductNameLabel" runat="server"
         Text='<%# Bind("ProductName") %>'>
        </asp:Label><br />
        <b>Units In Stock:</b>
        <asp:Label ID="UnitsInStockLabel" runat="server"
          Text='<%# Bind("UnitsInStock") %>'>
        </asp:Label>
    </ItemTemplate>
</asp:FormView>
```

Note that the `ItemTemplate` contains:

- **Static HTML** the text "Product:" and "Units In Stock:" along with the `<br />` and `<b>` elements.
- **Web controls** the two Label controls, `ProductNameLabel` and `UnitsInStockLabel`.
- **Databinding syntax** the `<%# Bind("ProductName") %>` and `<%# Bind("UnitsInStock") %>` syntax, which assigns the values from these fields to the Label controls' `Text` properties.

Step 5: Programmatically Determining the Value of the Data in the DataBound Event Handler

With the FormView's markup complete, the next step is to programmatically determine if the `UnitsInStock` value is less than or equal to 10. This is accomplished in the exact same manner with the FormView as it was with the DetailsView. Start by creating an event handler for the FormView's `DataBound` event.

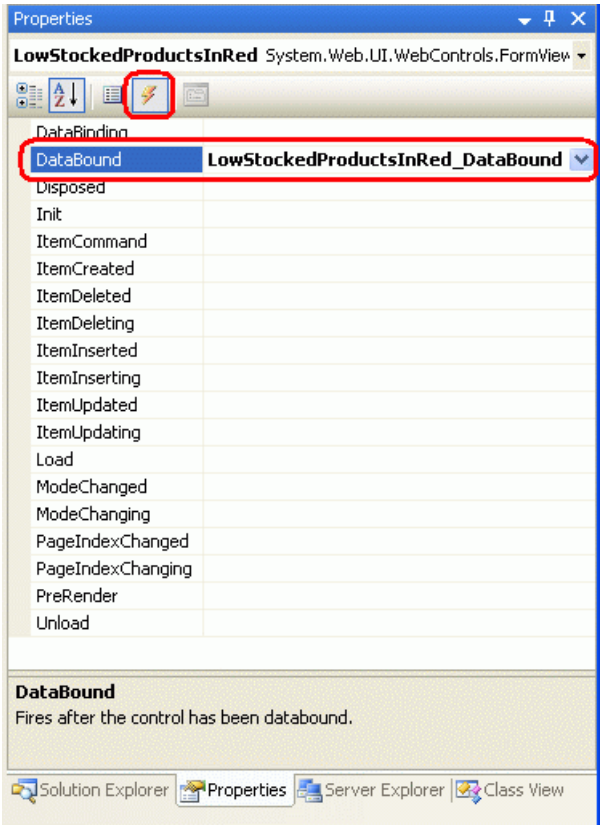


Figure 6: Create the `DataBound` Event Handler

In the event handler cast the FormView's `DataItem` property to a `ProductsRow` instance and determine whether the `UnitsInPrice` value is such that we need to display it in a red font.

```
protected void LowStockedProductsInRed_DataBound(object sender, EventArgs e)
{
    // Get the ProductsRow object from the DataItem property...
    Northwind.ProductsRow product = (Northwind.ProductsRow)
        ((DataRowView)LowStockedProductsInRed.DataItem).Row;
    if (!product.IsUnitsInStockNull() && product.UnitsInStock <= 10)
    {
        // TODO: Make the UnitsInStockLabel text red
    }
}
```

Step 6: Formatting the UnitsInStockLabel Label Control in the FormView's ItemTemplate

The final step is to format the displayed `UnitsInStock` value in a red font if the value is 10 or less. To accomplish this we need to programmatically access the `UnitsInStockLabel` control in the `ItemTemplate` and set its style properties so that its text is displayed in red. To access a Web control in a template, use the `FindControl("controlID")` method like this:

```
WebControlType someName = (WebControlType)FormViewID.FindControl("controlID");
```

For our example we want to access a Label control whose `ID` value is `UnitsInStockLabel`, so we'd use:

```
Label unitsInStock =
    (Label)LowStockedProductsInRed.FindControl("UnitsInStockLabel");
```

Once we have a programmatic reference to the Web control, we can modify its style-related properties as needed. As with the earlier example, I've created a CSS class in `Styles.css` named `LowUnitsInStockEmphasis`. To apply this style to the Label Web control, set its `CssClass` property accordingly.

```
protected void LowStockedProductsInRed_DataBound(object sender, EventArgs e)
{
    // Get the ProductsRow object from the DataItem property...
    Northwind.ProductsRow product = (Northwind.ProductsRow)
        ((DataRowView)LowStockedProductsInRed.DataItem).Row;
    if (!product.IsUnitsInStockNull() && product.UnitsInStock <= 10)
    {
        Label unitsInStock =
            (Label)LowStockedProductsInRed.FindControl("UnitsInStockLabel");

        if (unitsInStock != null)
        {
          unitsInStock.CssClass = "LowUnitsInStockEmphasis";
        }
    }
}
```

NOTE

The syntax for formatting a template programmatically accessing the Web control using `FindControl("controlID")` and then setting its style-related properties can also be used when using TemplateFields in the DetailsView or GridView controls. We'll examine TemplateFields in our next tutorial.

Figures 7 shows the FormView when viewing a product whose `UnitsInStock` value is greater than 10, while the product in Figure 8 has its value less than 10.

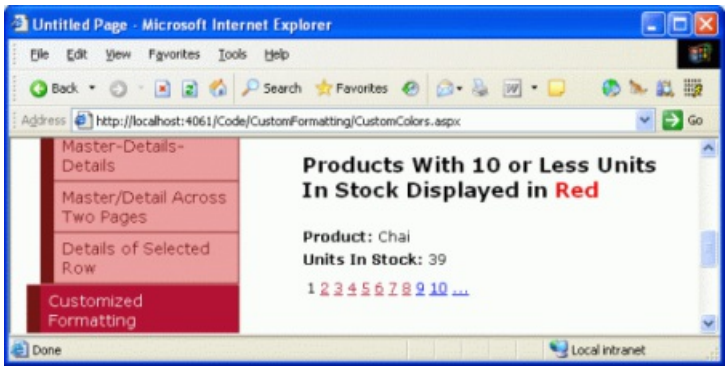


Figure 7: For Products With a Sufficiently Large Units In Stock, No Custom Formatting is Applied (Click to view full-size image)

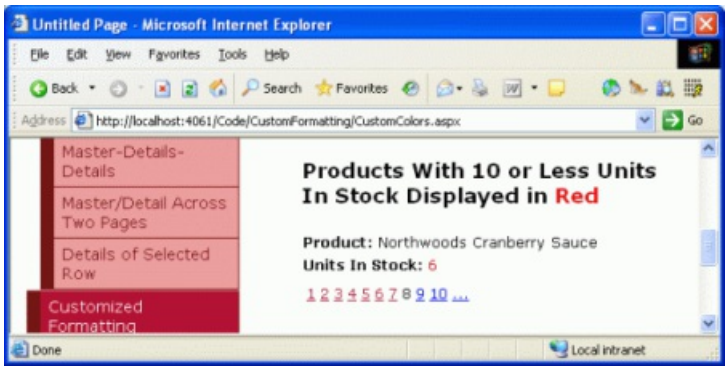


Figure 8: The Units in Stock Number is Shown in Red for Those Products With Values of 10 or Less (Click to view full-size image)

Formatting with the GridView's `RowDataBound` Event

Earlier we examined the sequence of steps the DetailsView and FormView controls progress through during databinding. Let's look over these steps once again as a refresher.

1. The data Web control's `DataBinding` event fires.
2. The data is bound to the data Web control.
3. The data Web control's `DataBound` event fires.

These three simple steps are sufficient for the DetailsView and FormView because they display only a single record. For the GridView, which displays *all* records bound to it (not just the first), step 2 is a bit more involved.

In step 2 the GridView enumerates the data source and, for each record, creates a `GridViewRow` instance and binds the current record to it. For each `GridViewRow` added to the GridView, two events are raised:

- `RowCreated` fires after the `GridViewRow` has been created
- `RowDataBound` fires after the current record has been bound to the `GridViewRow`.

For the GridView, then, data binding is more accurately described by the following sequence of steps:

1. The GridView's `DataBinding` event fires.
2. The data is bound to the GridView.

   For each record in the data source

   a. Create a `GridViewRow` object
   b. Fire the `RowCreated` event
   c. Bind the record to the `GridViewRow`
   d. Fire the `RowDataBound` event
   e. Add the `GridViewRow` to the `Rows` collection
3. The GridView's `DataBound` event fires.

To customize the format of the GridView's individual records, then, we need to create an event handler for the `RowDataBound` event. To illustrate this, let's add a GridView to the `CustomColors.aspx` page that lists the name, category, and price for each product, highlighting those products whose price is less than $10.00 with a yellow background color.

Step 7: Displaying Product Information in a GridView

Add a GridView beneath the FormView from the previous example and set its `ID` property to `HighlightCheapProducts`. Since we already have an ObjectDataSource that returns all products on the page, bind

the GridView to that. Finally, edit the GridView's BoundFields to include just the products' names, categories, and prices. After these edits the GridView's markup should look like:

```
<asp:GridView ID="HighlightCheapProducts" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1"
    EnableViewState="False" runat="server">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
          SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
          ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
          HeaderText="Price"
            HtmlEncode="False" SortExpression="UnitPrice" />
    </Columns>
</asp:GridView>
```

Figure 9 shows our progress to this point when viewed through a browser.

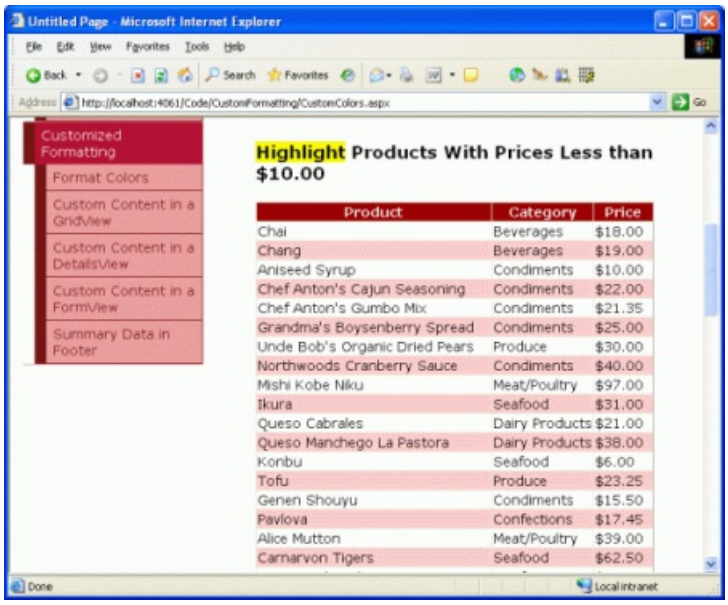


Figure 9: The GridView Lists the Name, Category, and Price For Each Product (Click to view full-size image)

Step 8: Programmatically Determining the Value of the Data in the RowDataBound Event Handler

When the `ProductsDataTable` is bound to the GridView its `ProductsRow` instances are enumerated and for each `ProductsRow` a `GridViewRow` is created. The `GridViewRow`'s `DataItem` property is assigned to the particular `ProductRow`, after which the GridView's `RowDataBound` event handler is raised. To determine the `UnitPrice` value for each product bound to the GridView, then, we need to create an event handler for the GridView's `RowDataBound` event. In this event handler we can inspect the `UnitPrice` value for the current `GridViewRow` and make a formatting decision for that row.

This event handler can be created using the same series of steps as with the FormView and DetailsView.

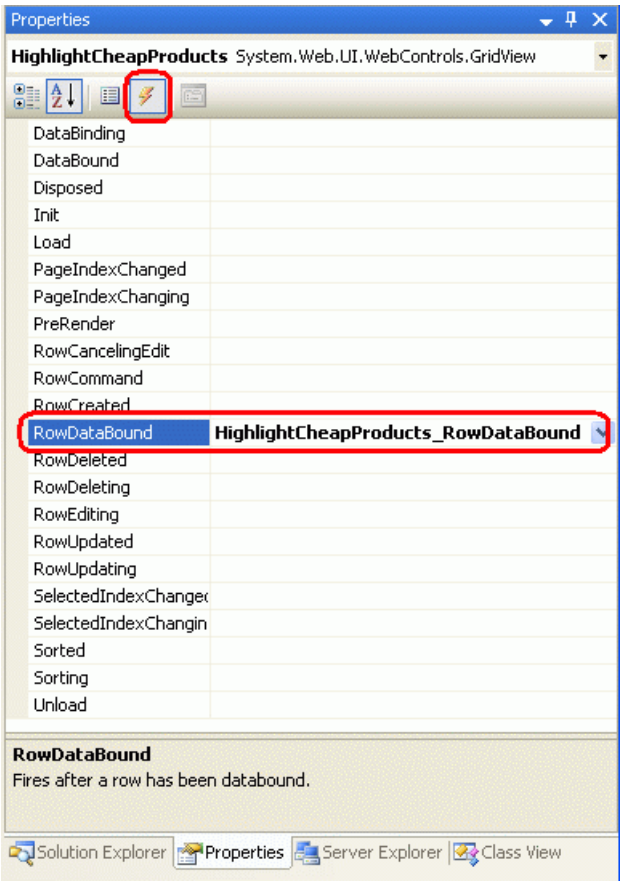


Figure 10: Create an Event Handler for the GridView's `RowDataBound` Event

Creating the event handler in this manner will cause the following code to be automatically added to the ASP.NET page's code portion:

```
protected void HighlightCheapProducts_RowDataBound(object sender, GridViewRowEventArgs e)
{

}
```

When the `RowDataBound` event fires, the event handler is passed as its second parameter an object of type `GridViewRowEventArgs`, which has a property named `Row`. This property returns a reference to the `GridViewRow` that was just data bound. To access the `ProductsRow` instance bound to the `GridViewRow` we use the `DataItem` property like so:

```
protected void HighlightCheapProducts_RowDataBound(object sender, GridViewRowEventArgs e)
{
    // Get the ProductsRow object from the DataItem property...
    Northwind.ProductsRow product = (Northwind.ProductsRow)
        ((System.Data.DataRowView)e.Row.DataItem).Row;
    if (!product.IsUnitPriceNull() && product.UnitPrice < 10m)
    {
        // TODO: Highlight the row yellow...
    }
}
```

When working with the `RowDataBound` event handler it is important to keep in mind that the GridView is composed of different types of rows and that this event is fired for *all* row types. A `GridViewRow`'s type can be determined by its `RowType` property, and can have one of the possible values:

- `DataRow` a row that is bound to a record from the GridView's `DataSource`
- `EmptyDataRow` the row displayed if the GridView's `DataSource` is empty

- `Footer` the footer row; shown if the GridView's `ShowFooter` property is set to `true`
- `Header` the header row; shown if the GridView's ShowHeader property is set to `true` (the default)
- `Pager` for GridView's that implement paging, the row that displays the paging interface
- `Separator` not used for the GridView, but used by the `RowType` properties for the DataList and Repeater controls, two data Web controls we'll discuss in future tutorials

Since the `EmptyDataRow`, `Header`, `Footer`, and `Pager` rows aren't associated with a `DataSource` record, they will always have a `null` value for their `DataItem` property. For this reason, before attempting to work with the current `GridViewRow`'s `DataItem` property, we first must make sure that we're dealing with a `DataRow`. This can be accomplished by checking the `GridViewRow`'s `RowType` property like so:

```
protected void HighlightCheapProducts_RowDataBound(object sender, GridViewRowEventArgs e)
{
    // Make sure we are working with a DataRow
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        // Get the ProductsRow object from the DataItem property...
        Northwind.ProductsRow product = (Northwind.ProductsRow)
            ((System.Data.DataRowView)e.Row.DataItem).Row;
        if (!product.IsUnitPriceNull() && product.UnitPrice < 10m)
        {
          // TODO: Highlight row yellow...
        }
    }
}
```

Step 9: Highlighting the Row Yellow When the UnitPrice Value is Less than $10.00

The last step is to programmatically highlight the entire `GridViewRow` if the `UnitPrice` value for that row is less than $10.00. The syntax for accessing a GridView's rows or cells is the same as with the DetailsView `GridViewID.Rows[index]` to access the entire row, `GridViewID.Rows[index].Cells[index]` to access a particular cell. However, when the `RowDataBound` event handler fires the data bound `GridViewRow` has yet to be added to the GridView's `Rows` collection. Therefore you cannot access the current `GridViewRow` instance from the `RowDataBound` event handler using the Rows collection.

Instead of `GridViewID.Rows[index]`, we can reference the current `GridViewRow` instance in the `RowDataBound` event handler using `e.Row`. That is, in order to highlight the current `GridViewRow` instance from the `RowDataBound` event handler we would use:

```
e.Row.BackColor = System.Drawing.Color.Yellow;
```

Rather than set the `GridViewRow`'s `BackColor` property directly, let's stick with using CSS classes. I've created a CSS class named `AffordablePriceEmphasis` that sets the background color to yellow. The completed `RowDataBound` event handler follows:

```
protected void HighlightCheapProducts_RowDataBound(object sender, GridViewRowEventArgs e)
{
    // Make sure we are working with a DataRow
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        // Get the ProductsRow object from the DataItem property...
        Northwind.ProductsRow product = (Northwind.ProductsRow)
            ((System.Data.DataRowView)e.Row.DataItem).Row;
        if (!product.IsUnitPriceNull() && product.UnitPrice < 10m)
        {
            e.Row.CssClass = "AffordablePriceEmphasis";
        }
    }
}
```

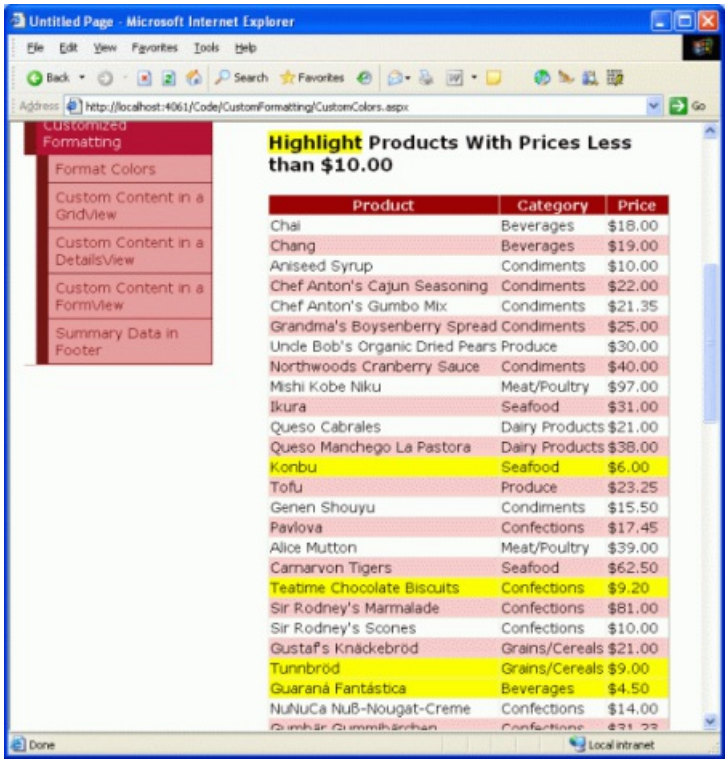


Figure 11: The Most Affordable Products are Highlighted Yellow (Click to view full-size image)

Summary

In this tutorial we saw how to format the GridView, DetailsView, and FormView based on the data bound to the control. To accomplish this we created an event handler for the `DataBound` or `RowDataBound` events, where the underlying data was examined along with a formatting change, if needed. To access the data bound to a DetailsView or FormView, we use the `DataItem` property in the `DataBound` event handler; for a GridView, each `GridViewRow` instance's `DataItem` property contains the data bound to that row, which is available in the `RowDataBound` event handler.

The syntax for programmatically adjusting the data Web control's formatting depends upon the Web control and how the data to be formatted is displayed. For DetailsView and GridView controls, the rows and cells can be accessed by an ordinal index. For the FormView, which uses templates, the `FindControl("controlID")` method is commonly used to locate a Web control from within the template.

In the next tutorial we'll look at how to use templates with the GridView and DetailsView. Additionally, we'll see another technique for customizing the formatting based on the underlying data.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were E.R. Gilmore, Dennis Patterson, and Dan Jagers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Using TemplateFields in the GridView Control (C#)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> To provide flexibility, the GridView offers the TemplateField, which renders using a template. A template can include a mix of static HTML, Web controls, and databinding syntax. In this tutorial we'll examine how to use the TemplateField to achieve a greater degree of customization with the GridView control.

Introduction

The GridView is composed of a set of fields that indicate what properties from the `DataSource` are to be included in the rendered output along with how the data will be displayed. The simplest field type is the BoundField, which displays a data value as text. Other field types display the data using alternate HTML elements. The CheckBoxField, for example, renders as a checkbox whose checked state depends on the value of a specified data field; the ImageField renders an image whose image source is based upon a specified data field. Hyperlinks and buttons whose state depends on an underlying data field value can be rendered using the HyperLinkField and ButtonField field types.

While the CheckBoxField, ImageField, HyperLinkField, and ButtonField field types allow for an alternate view of the data, they still are fairly limited with respect to formatting. A CheckBoxField can only display a single checkbox, whereas an ImageField can only display a single image. What if a particular field needs to display some text, a checkbox, *and* an image, all based upon different data field values? Or what if we wanted to display the data using a Web control other than the CheckBox, Image, HyperLink, or Button? Furthermore, the BoundField limits its display to a single data field. What if we wanted to show two or more data field values in a single GridView column?

To accommodate this level of flexibility the GridView offers the TemplateField, which renders using a *template*. A template can include a mix of static HTML, Web controls, and databinding syntax. Furthermore, the TemplateField has a variety of templates that can be used to customize the rendering for different situations. For example, the `ItemTemplate` is used by default to render the cell for each row, but the `EditItemTemplate` template can be used to customize the interface when editing data.

In this tutorial we'll examine how to use the TemplateField to achieve a greater degree of customization with the GridView control. In the preceding tutorial we saw how to customize the formatting based on the underlying data using the `DataBound` and `RowDataBound` event handlers. Another way to customize the formatting based on the underlying data is by calling formatting methods from within a template. We'll look at this technique in this tutorial as well.

For this tutorial we will use TemplateFields to customize the appearance of a list of employees. Specifically, we'll list all of the employees, but will display the employee's first and last names in one column, their hire date in a Calendar control, and a status column that indicates how many days they've been employed at the company.

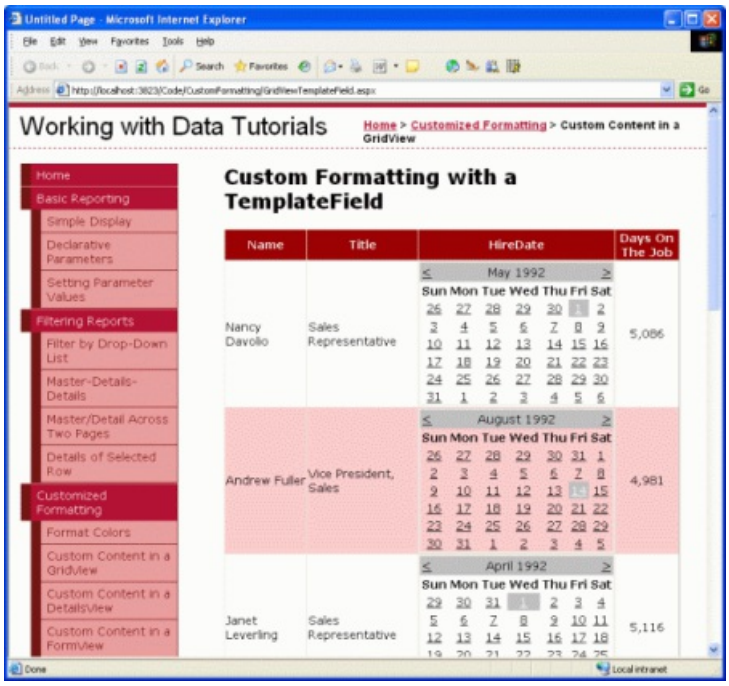


Figure 1: Three TemplateFields are Used to Customize the Display (Click to view full-size image)

Step 1: Binding the Data to the GridView

In reporting scenarios where you need to use TemplateFields to customize the appearance, I find it easiest to start by creating a GridView control that contains just BoundFields first and then to add new TemplateFields or convert the existing BoundFields to TemplateFields as needed. Therefore, let's start this tutorial by adding a GridView to the page through the Designer and binding it to an ObjectDataSource that returns the list of employees. These steps will create a GridView with BoundFields for each of the employee fields.

Open the `GridViewTemplateField.aspx` page and drag a GridView from the Toolbox onto the Designer. From the GridView's smart tag choose to add a new ObjectDataSource control that invokes the `EmployeesBLL` class's `GetEmployees()` method.

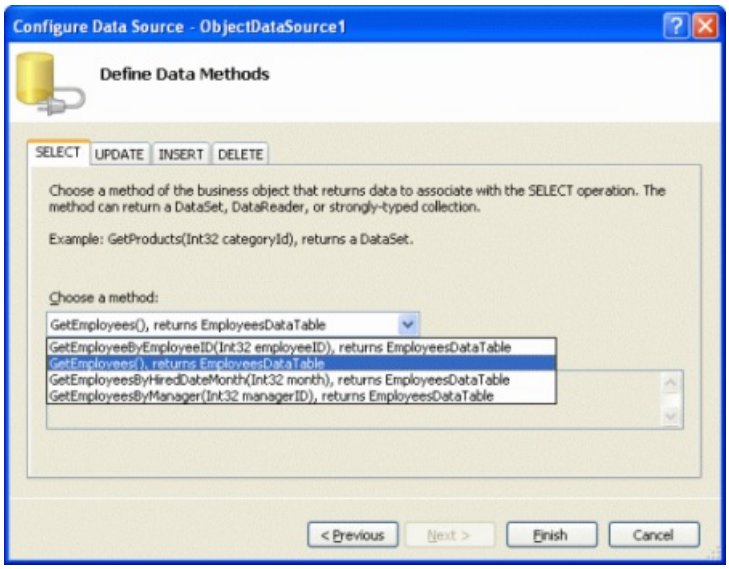


Figure 2: Add a New ObjectDataSource Control that Invokes the `GetEmployees()` Method (Click to view full-size image)

Binding the GridView in this manner will automatically add a BoundField for each of the employee properties: `EmployeeID`, `LastName`, `FirstName`, `Title`, `HireDate`, `ReportsTo`, and `Country`. For this report let's not bother with displaying the `EmployeeID`, `ReportsTo`, or `Country` properties. To remove these BoundFields you can:

- Use the Fields dialog box click on the Edit Columns link from the GridView's smart tag to bring up this dialog box. Next, select the BoundFields from the lower left list and click the red X button to remove the BoundField.
- Edit the GridView's declarative syntax by hand from the Source view, delete the `<asp:BoundField>` element for the BoundField you wish to remove.

After you have removed the `EmployeeID`, `ReportsTo`, and `Country` BoundFields, your GridView's markup should look like:

```
<asp:GridView ID="GridView1" runat="server"
    AutoGenerateColumns="False" DataKeyNames="EmployeeID"
    DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:BoundField DataField="LastName" HeaderText="LastName"
            SortExpression="LastName" />
        <asp:BoundField DataField="FirstName" HeaderText="FirstName"
            SortExpression="FirstName" />
        <asp:BoundField DataField="Title" HeaderText="Title"
            SortExpression="Title" />
        <asp:BoundField DataField="HireDate" HeaderText="HireDate"
            SortExpression="HireDate" />
    </Columns>
</asp:GridView>
```

Take a moment to view our progress in a browser. At this point you should see a table with a record for each employee and four columns: one for the employee's last name, one for their first name, one for their title, and one for their hire date.

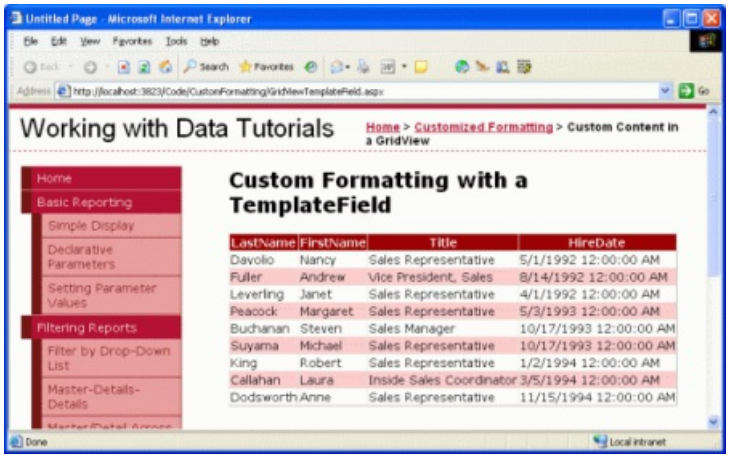


Figure 3: The `LastName`, `FirstName`, `Title`, and `HireDate` Fields are Displayed for Each Employee (Click to view full-size image)

Step 2: Displaying the First and Last Names in a Single Column

Currently, each employee's first and last names are displayed in a separate column. It might be nice to combine them into a single column instead. To accomplish this we need to use a TemplateField. We can either add a new TemplateField, add to it the needed markup and databinding syntax, and then delete the `FirstName` and `LastName` BoundFields, or we can convert the `FirstName` BoundField into a TemplateField, edit the TemplateField to include the `LastName` value, and then remove the `LastName` BoundField.

Both approaches net the same result, but personally I like converting BoundFields to TemplateFields when possible because the conversion automatically adds an `ItemTemplate` and `EditItemTemplate` with Web controls and databinding syntax to mimic the appearance and functionality of the BoundField. The benefit is that we'll need to do less work with the TemplateField as the conversion process will have performed some of the work for us.

To convert an existing BoundField into a TemplateField, click on the Edit Columns link from the GridView's smart tag, bringing up the Fields dialog box. Select the BoundField to convert from the list in the lower left corner and then click the "Convert this field into a TemplateField" link in the bottom right corner.

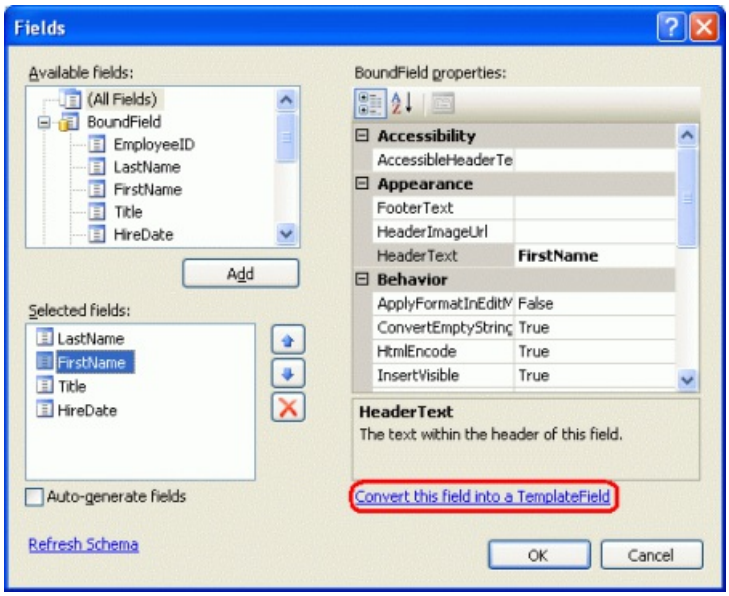


Figure 4: Convert a BoundField Into a TemplateField from the Fields Dialog Box (Click to view full-size image)

Go ahead and convert the `FirstName` BoundField into a TemplateField. After this change there's no perceptive difference in the Designer. This is because converting the BoundField into a TemplateField creates a TemplateField that maintains the look and feel of the BoundField. Despite there being no visual difference at this point in the Designer, this conversion process has replaced the BoundField's declarative syntax - `<asp:BoundField DataField="FirstName" HeaderText="FirstName" SortExpression="FirstName" />` - with the following TemplateField syntax:

```
<asp:TemplateField HeaderText="FirstName" SortExpression="FirstName">
    <EditItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
            Text='<%# Bind("FirstName") %>'></asp:TextBox>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
            Text='<%# Bind("FirstName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

As you can see, the TemplateField consists of two templates an `ItemTemplate` that has a Label whose `Text` property is set to the value of the `FirstName` data field, and an `EditItemTemplate` with a TextBox control whose `Text` property is also set to the `FirstName` data field. The databinding syntax - `<%# Bind("fieldName") %>` - indicates that the data field *`fieldName`* is bound to the specified Web control property.

To add the `LastName` data field value to this TemplateField we need to add another Label Web control in the `ItemTemplate` and bind its `Text` property to `LastName`. This can be accomplished either by hand or through the Designer. To do it by hand, simply add the appropriate declarative syntax to the `ItemTemplate`:

```
<asp:TemplateField HeaderText="FirstName" SortExpression="FirstName">
    <EditItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
            Text='<%# Bind("FirstName") %>'></asp:TextBox>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
            Text='<%# Bind("FirstName") %>'></asp:Label>
        <asp:Label ID="Label2" runat="server"
            Text='<%# Bind("LastName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

To add it through the Designer, click on the Edit Templates link from the GridView's smart tag. This will display the GridView's template editing interface. In this interface's smart tag is a list of the templates in the GridView. Since we only have one TemplateField at this point, the only templates listed in the drop-down list are those templates for the `FirstName` TemplateField along with the `EmptyDataTemplate` and `PagerTemplate`. The `EmptyDataTemplate` template, if specified, is used to render the GridView's output if there are no results in the data bound to the GridView; the `PagerTemplate`, if specified, is used to render the paging interface for a GridView that supports paging.

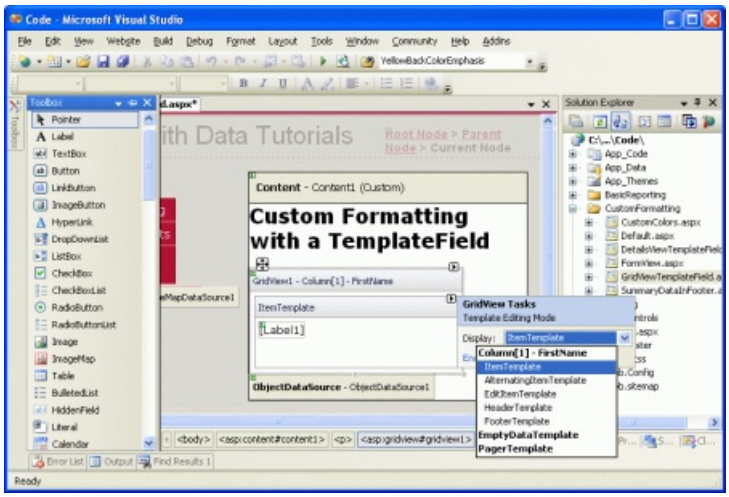


Figure 5: The GridView's Templates Can Be Edited Through the Designer (Click to view full-size image)

To also display the `LastName` in the `FirstName` TemplateField drag the Label control from the Toolbox into the `FirstName` TemplateField's `ItemTemplate` in the GridView's template editing interface.

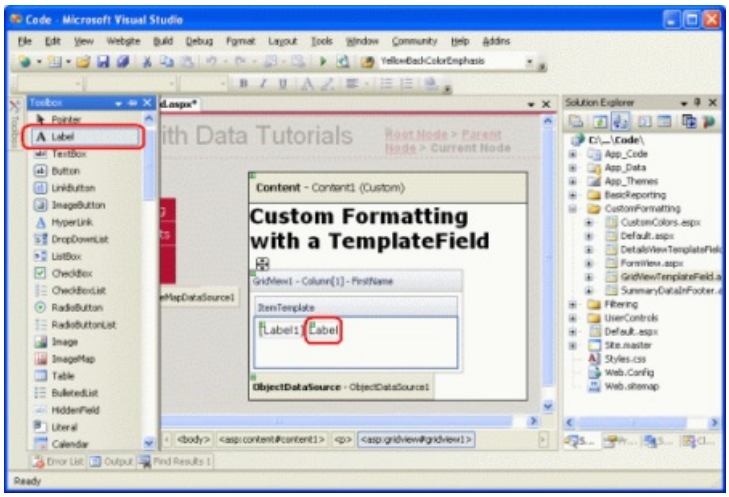


Figure 6: Add a Label Web Control to the `FirstName` TemplateField's ItemTemplate (Click to view full-size image)

At this point the Label Web control added to the TemplateField has its `Text` property set to "Label". We need to change this so that this property is bound to the value of the `LastName` data field instead. To accomplish this click on the Label control's smart tag and choose the Edit DataBindings option.

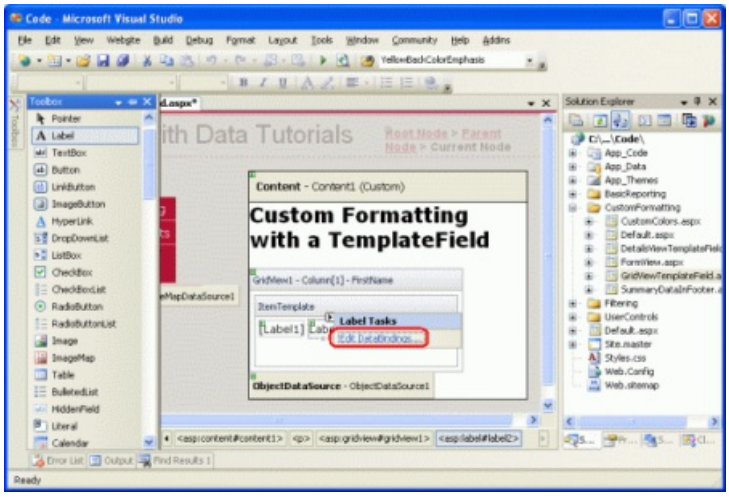


Figure 7: Choose the Edit DataBindings Option from the Label's Smart Tag (Click to view full-size image)

This will bring up the DataBindings dialog box. From here you can select the property to participate in databinding from the list on the left and choose the field to bind the data to from the drop-down list on the right. Choose the `Text` property from the left and the `LastName` field from the right and click OK.

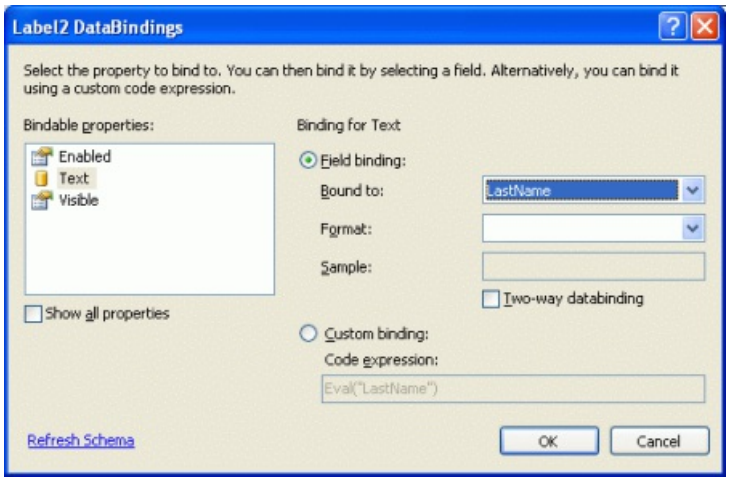


Figure 8: Bind the `Text` Property to the `LastName` Data Field (Click to view full-size image)

> **NOTE**
>
> The DataBindings dialog box allows you to indicate whether to perform two-way databinding. If you leave this unchecked, the databinding syntax `<%# Eval("LastName")%>` will be used instead of `<%# Bind("LastName")%>`. Either approach is fine for this tutorial. Two-way databinding becomes important when inserting and editing data. For simply displaying data, however, either approach will work equally well. We'll discuss two-way databinding in detail in future tutorials.

Take a moment to view this page through a browser. As you can see, the GridView still includes four columns; however, the `FirstName` column now lists *both* the `FirstName` and `LastName` data field values.

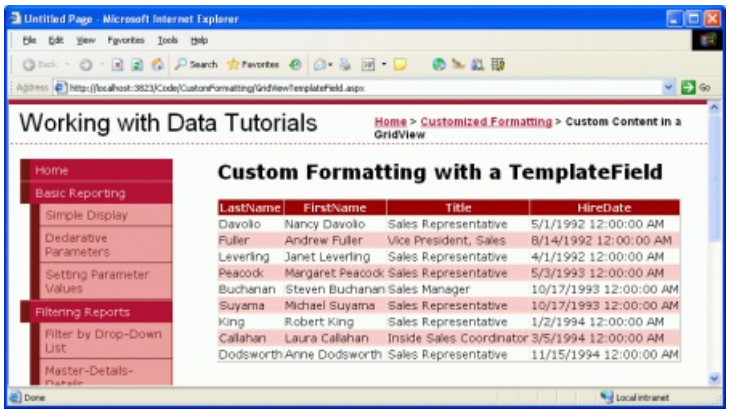


Figure 9: Both the `FirstName` and `LastName` Values are Shown in a Single Column (Click to view full-size image)

To complete this first step, remove the `LastName` BoundField and rename the `FirstName` TemplateField's `HeaderText` property to "Name". After these changes the GridView's declarative markup should look like the following:

```
<asp:GridView ID="GridView1" runat="server"
    AutoGenerateColumns="False" DataKeyNames="EmployeeID"
    DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:TemplateField HeaderText="Name" SortExpression="FirstName">
            <EditItemTemplate>
                <asp:TextBox ID="TextBox1" runat="server"
                    Text='<%# Bind("FirstName") %>'></asp:TextBox>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("FirstName") %>'></asp:Label>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Eval("LastName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="Title" HeaderText="Title"
            SortExpression="Title" />
        <asp:BoundField DataField="HireDate" HeaderText="HireDate"
            SortExpression="HireDate" />
    </Columns>
</asp:GridView>
```

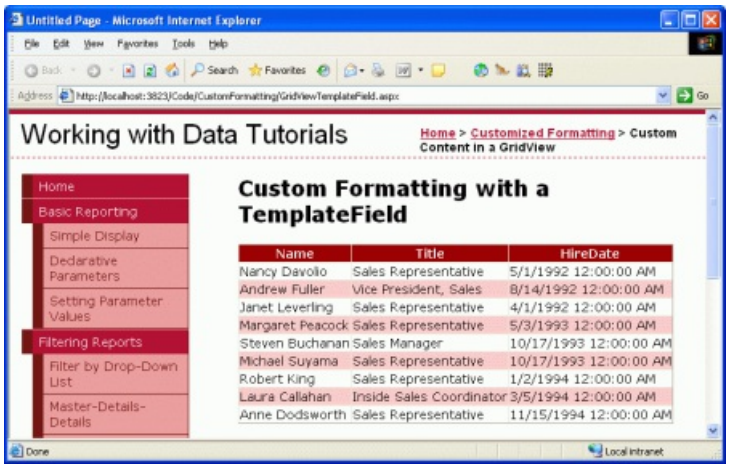


Figure 10: Each Employee's First and Last Names are Displayed in One Column (Click to view full-size image)

Step 3: Using the Calendar Control to Display the `HiredDate` Field

Displaying a data field value as text in a GridView is as simple as using a BoundField. For certain scenarios,

however, the data is best expressed using a particular Web control instead of just text. Such customization of the display of data is possible with TemplateFields. For example, rather than display the employee's hire date as text, we could show a calendar (using the Calendar control) with their hire date highlighted.

To accomplish this, start by converting the `HiredDate` BoundField into a TemplateField. Simply go to the GridView's smart tag and click the Edit Columns link, bringing up the Fields dialog box. Select the `HiredDate` BoundField and click "Convert this field into a TemplateField."

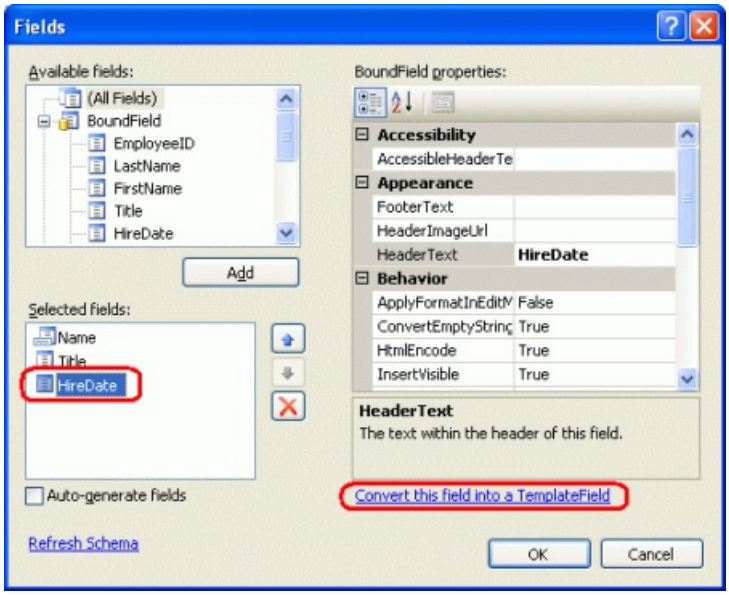


Figure 11: Convert the `HiredDate` BoundField Into a TemplateField (Click to view full-size image)

As we saw in Step 2, this will replace the BoundField with a TemplateField that contains an `ItemTemplate` and `EditItemTemplate` with a Label and TextBox whose `Text` properties are bound to the `HiredDate` value using the databinding syntax `<%# Bind("HiredDate")%>`.

To replace the text with a Calendar control, edit the template by removing the Label and adding a Calendar control. From the Designer, select Edit Templates from the GridView's smart tag and choose the `HireDate` TemplateField's `ItemTemplate` from the drop-down list. Next, delete the Label control and drag a Calendar control from the Toolbox into the template editing interface.

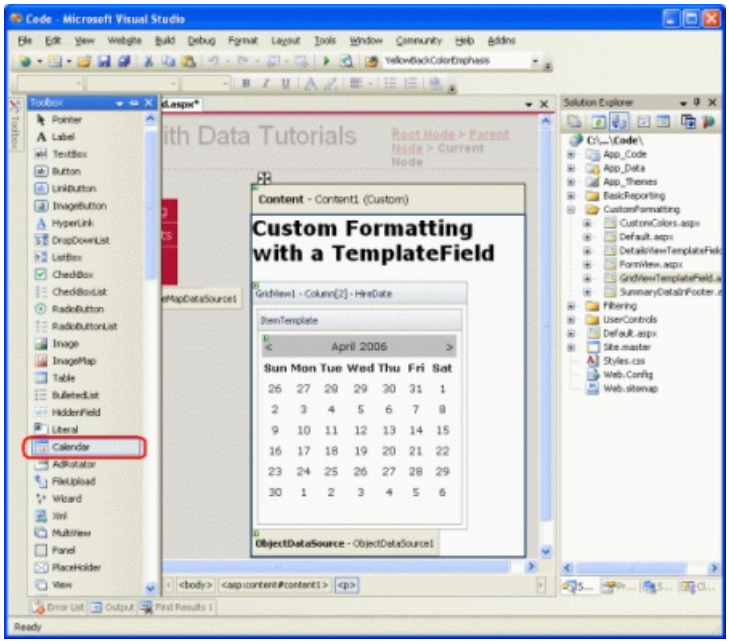


Figure 12: Add a Calendar Control to the `HireDate` TemplateField's `ItemTemplate` (Click to view full-size image)

At this point each row in the GridView will contain a Calendar control in its `HiredDate` TemplateField. However,

the employee's actual `HiredDate` value is not set anywhere in the Calendar control, causing each Calendar control to default to showing the current month and date. To remedy this, we need to assign each employee's `HiredDate` to the Calendar control's SelectedDate and VisibleDate properties.

From the Calendar control's smart tag, choose Edit DataBindings. Next, bind both `SelectedDate` and `VisibleDate` properties to the `HiredDate` data field.

Figure 13: Bind the `SelectedDate` and `VisibleDate` Properties to the `HiredDate` Data Field (Click to view full-size image)

> **NOTE**
>
> The Calendar control's selected date need not necessarily be visible. For example, a Calendar may have August 1st, 1999 as the selected date, but be showing the current month and year. The selected date and visible date are specified by the Calendar control's `SelectedDate` and `VisibleDate` properties. Since we want to both select the employee's `HiredDate` and make sure that it's shown we need to bind both of these properties to the `HireDate` data field.

When viewing the page in a browser, the calendar now shows the month of the employee's hired date and selects that particular date.

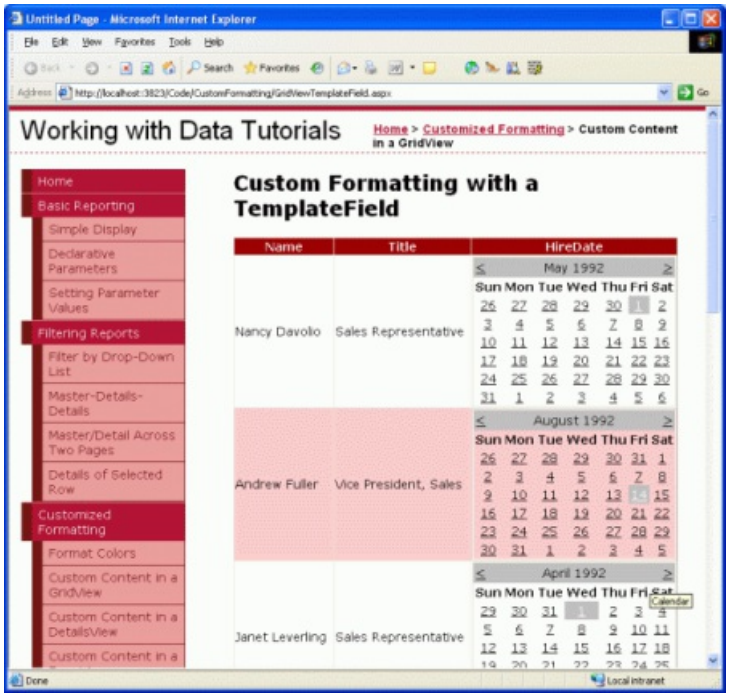


Figure 14: The Employee's `HiredDate` is Shown in the Calendar Control (Click to view full-size image)

NOTE

Contrary to all the examples we've seen thus far, for this tutorial we did *not* set `EnableViewState` property to `false` for this GridView. The reason for this decision is because clicking the dates of the Calendar control causes a postback, setting the Calendar's selected date to the date just clicked. If the GridView's view state is disabled, however, on each postback the GridView's data is rebound to its underlying data source, which causes the Calendar's selected date to be set *back* to the employee's `HireDate`, overwriting the date chosen by the user.

For this tutorial this is a moot discussion since the user is not able to update the employee's `HireDate`. It would probably be best to configure the Calendar control so that its dates are not selectable. Regardless, this tutorial shows that in some circumstances view state must be enabled in order to provide certain functionality.

Step 4: Showing the Number of Days the Employee Has Worked for the Company

So far we have seen two applications of TemplateFields:

- Combining two or more data field values into one column, and
- Expressing a data field value using a Web control instead of text

A third use of TemplateFields is in displaying metadata about the GridView's underlying data. In addition to showing the employees' hire dates, for example, we might also want to have a column that displays how many total days they've been on the job.

Yet another use of TemplateFields arises in scenarios when the underlying data needs to be displayed differently in the web page report than in the format it's stored in the database. Imagine that the `Employees` table had a `Gender` field that stored the character `M` or `F` to indicate the sex of the employee. When displaying this information in a web page we might want to show the gender as either "Male" or "Female", as opposed to just "M" or "F".

Both of these scenarios can be handled by creating a *formatting method* in the ASP.NET page's code-behind class (or in a separate class library, implemented as a `static` method) that is invoked from the template. Such a formatting method is invoked from the template using the same databinding syntax seen earlier. The formatting method can take in any number of parameters, but must return a string. This returned string is the HTML that is injected into the template.

To illustrate this concept, let's augment our tutorial to show a column that lists the total number of days an employee has been on the job. This formatting method will take in a `Northwind.EmployeesRow` object and return the number of days the employee has been employed as a string. This method can be added to the ASP.NET page's code-behind class, but *must* be marked as `protected` or `public` in order to be accessible from the template.

```
protected string DisplayDaysOnJob(Northwind.EmployeesRow employee)
{
    // Make sure HiredDate is not null... if so, return "Unknown"
    if (employee.IsHireDateNull())
        return "Unknown";
    else
    {
        // Returns the number of days between the current
        // date/time and HireDate
        TimeSpan ts = DateTime.Now.Subtract(employee.HireDate);
        return ts.Days.ToString("#,##0");
    }
}
```

Since the `HiredDate` field can contain `NULL` database values we must first ensure that the value is not `NULL` before proceeding with the calculation. If the `HiredDate` value is `NULL`, we simply return the string "Unknown"; if it is not `NULL`, we compute the difference between the current time and the `HiredDate` value and return the number of days.

To utilize this method we need to invoke it from a TemplateField in the GridView using the databinding syntax. Start by adding a new TemplateField to the GridView by clicking on the Edit Columns link in the GridView's smart tag and adding a new TemplateField.

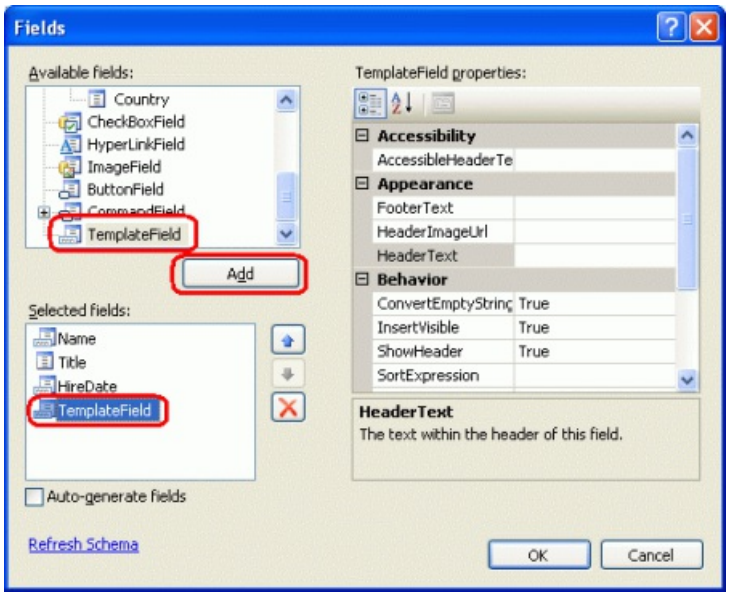


Figure 15: Add a New TemplateField to the GridView (Click to view full-size image)

Set this new TemplateField's `HeaderText` property to "Days on the Job" and its `ItemStyle`'s `HorizontalAlign` property to `Center`. To call the `DisplayDaysOnJob` method from the template, add an `ItemTemplate` and use the following databinding syntax:

```
<%# DisplayDaysOnJob((Northwind.EmployeesRow)
     ((System.Data.DataRowView) Container.DataItem).Row) %>
```

`Container.DataItem` returns a `DataRowView` object corresponding to the `DataSource` record bound to the `GridViewRow`. Its `Row` property returns the strongly-typed `Northwind.EmployeesRow`, which is passed to the `DisplayDaysOnJob` method. This databinding syntax can appear directly in the `ItemTemplate` (as shown in the declarative syntax below) or can be assigned to the `Text` property of a Label Web control.

> **NOTE**
>
> Alternatively, instead of passing in an `EmployeesRow` instance, we could just pass in the `HireDate` value using `<%# DisplayDaysOnJob(Eval("HireDate")) %>`. However, the `Eval` method returns an `object`, so we'd have to change our `DisplayDaysOnJob` method signature to accept an input parameter of type `object`, instead. We can't blindly cast the `Eval("HireDate")` call to a `DateTime` because the `HireDate` column in the `Employees` table can contain `NULL` values. Therefore, we'd need to accept an `object` as the input parameter for the `DisplayDaysOnJob` method, check to see if it had a database `NULL` value (which can be accomplished using `Convert.IsDBNull(objectToCheck)`), and then proceed accordingly.

Due to these subtleties, I've opted to pass in the entire `EmployeesRow` instance. In the next tutorial we'll see a more fitting example for using the `Eval("columnName")` syntax for passing an input parameter into a formatting method.

The following shows the declarative syntax for our GridView after the TemplateField has been added and the

`DisplayDaysOnJob` method called from the `ItemTemplate`:

```
<asp:GridView ID="GridView1" runat="server"
    AutoGenerateColumns="False" DataKeyNames="EmployeeID"
    DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:TemplateField HeaderText="Name" SortExpression="FirstName">
            <EditItemTemplate>
                <asp:TextBox ID="TextBox1" runat="server"
                    Text='<%# Bind("FirstName") %>'></asp:TextBox>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("FirstName") %>'></asp:Label>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Eval("LastName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="Title" HeaderText="Title"
            SortExpression="Title" />
        <asp:TemplateField HeaderText="HireDate"
            SortExpression="HireDate">
            <EditItemTemplate>
                <asp:TextBox ID="TextBox2" runat="server"
                    Text='<%# Bind("HireDate") %>'></asp:TextBox>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Calendar ID="Calendar1" runat="server"
                    SelectedDate='<%# Bind("HireDate") %>'
                    VisibleDate='<%# Eval("HireDate") %>'>
            </asp:Calendar>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Days On The Job">
            <ItemTemplate>
                <%# DisplayDaysOnJob((Northwind.EmployeesRow)
                    ((System.Data.DataRowView) Container.DataItem).Row) %>
            </ItemTemplate>
            <ItemStyle HorizontalAlign="Center" />
        </asp:TemplateField>
    </Columns>
</asp:GridView>
```

Figure 16 shows the completed tutorial, when viewed through a browser.

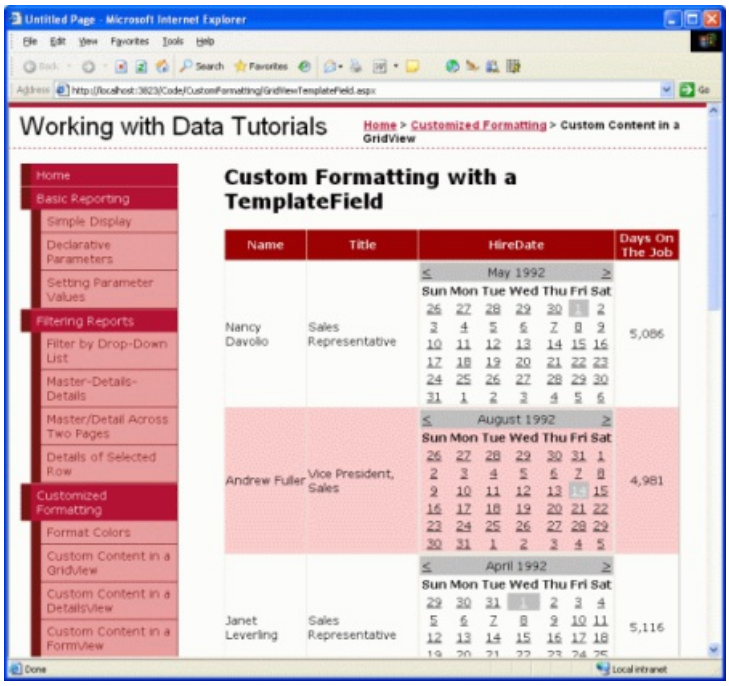


Figure 16: The Number of Days the Employee Has Been on the Job is Displayed (Click to view full-size image)

Summary

The TemplateField in the GridView control allows for a higher degree of flexibility in displaying data than is available with the other field controls. TemplateFields are ideal for situations where:

- Multiple data fields need to be displayed in one GridView column
- The data is best expressed using a Web control rather than plain text
- The output depends on the underlying data, such as displaying metadata or in reformatting the data

In addition to customizing the display of data, TemplateFields are also used for customizing the user interfaces used for editing and inserting data, as we'll see in future tutorials.

The next two tutorials continue exploring templates, starting with a look at using TemplateFields in a DetailsView. Following that, we'll turn to the FormView, which uses templates in lieu of fields to provide greater flexibility in the layout and structure of the data.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Dan Jagers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Using TemplateFields in the DetailsView Control (C#)

2/20/2020 • 11 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> The same TemplateFields capabilities available with the GridView are also available with the DetailsView control. In this tutorial we'll display one product at a time using a DetailsView containing TemplateFields.

Introduction

The TemplateField offers a higher degree of flexibility in rendering data than the BoundField, CheckBoxField, HyperLinkField, and other data field controls. In the previous tutorial we looked at using the TemplateField in a GridView to:

- Display multiple data field values in one column. Specifically, both the `FirstName` and `LastName` fields were combined into one GridView column.
- Use an alternate Web control to express a data field value. We saw how to show the `HiredDate` value using a Calendar control.
- Show status information based on the underlying data. While the `Employees` table does not contain a column that returns the number of days an employee has been on the job, we were able to display such information in the GridView example in the previous tutorial with the use of a TemplateField and formatting method.

The same TemplateFields capabilities available with the GridView are also available with the DetailsView control. In this tutorial we'll display one product at a time using a DetailsView that contains two TemplateFields. The first TemplateField will combine the `UnitPrice`, `UnitsInStock`, and `UnitsOnOrder` data fields into one DetailsView row. The second TemplateField will display the value of the `Discontinued` field, but will use a formatting method to display "YES" if `Discontinued` is `true`, and "NO" otherwise.

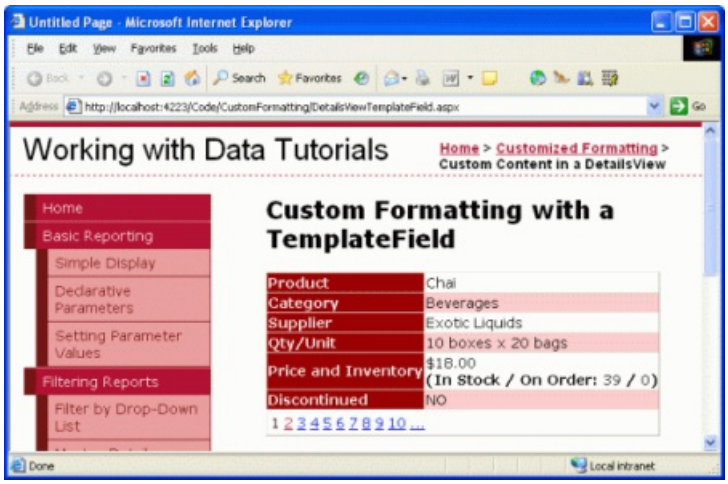


Figure 1: Two TemplateFields are Used to Customize the Display (Click to view full-size image)

Let's get started!

Step 1: Binding the Data to the DetailsView

As discussed in the previous tutorial, when working with TemplateFields it's often easiest to start by creating the DetailsView control that contains just BoundFields and then add new TemplateFields or convert the existing

BoundFields to TemplateFields as needed. Therefore, start this tutorial by adding a DetailsView to the page through the Designer and binding it to an ObjectDataSource that returns the list of products. These steps will create a DetailsView with BoundFields for each of the product's non-Boolean value fields and a CheckBoxField for the one Boolean value field (Discontinued).

Open the `DetailsViewTemplateField.aspx` page and drag a DetailsView from the Toolbox onto the Designer. From the DetailsView's smart tag choose to add a new ObjectDataSource control that invokes the `ProductsBLL` class's `GetProducts()` method.

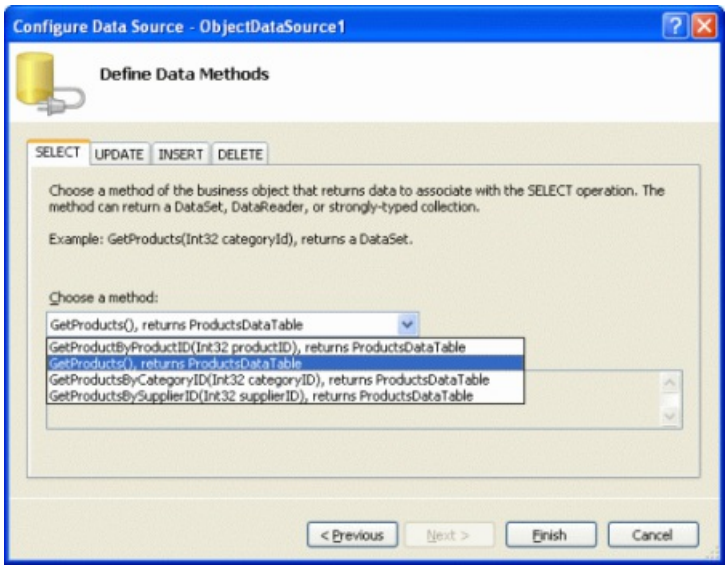


Figure 2: Add a New ObjectDataSource Control that Invokes the `GetProducts()` Method (Click to view full-size image)

For this report remove the `ProductID`, `SupplierID`, `CategoryID`, and `ReorderLevel` BoundFields. Next, reorder the BoundFields so that the `CategoryName` and `SupplierName` BoundFields appear immediately after the `ProductName` BoundField. Feel free to adjust the `HeaderText` properties and formatting properties for the BoundFields as you see fit. Like with the GridView, these BoundField-level edits can be performed through the Fields dialog box (accessible by clicking on the Edit Fields link in the DetailsView's smart tag) or through the declarative syntax. Lastly, clear out the DetailsView's `Height` and `Width` property values in order to allow the DetailsView control to expand based on the data displayed and check the Enable Paging checkbox in the smart tag.

After making these changes, your DetailsView control's declarative markup should look similar to the following:

```
<asp:DetailsView ID="DetailsView1" runat="server" AutoGenerateRows="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1" AllowPaging="True"
    EnableViewState="False">
    <Fields>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
          SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
          ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
          ReadOnly="True" SortExpression="SupplierName" />
        <asp:BoundField DataField="QuantityPerUnit"
          HeaderText="Qty/Unit" SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
          SortExpression="UnitPrice" />
        <asp:BoundField DataField="UnitsInStock"
          HeaderText="Units In Stock" SortExpression="UnitsInStock" />
        <asp:BoundField DataField="UnitsOnOrder"
          HeaderText="Units On Order" SortExpression="UnitsOnOrder" />
        <asp:CheckBoxField DataField="Discontinued"
          HeaderText="Discontinued" SortExpression="Discontinued" />
    </Fields>
</asp:DetailsView>
```

Take a moment to view the page through a browser. At this point you should see a single product listed (Chai) with rows showing the product's name, category, supplier, price, units in stock, units on order, and its discontinued status.

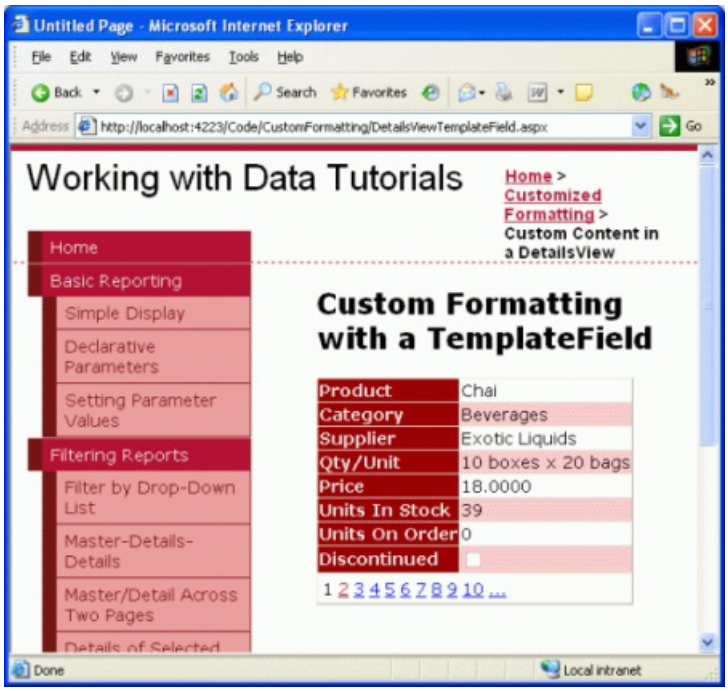


Figure 3: The Product's Details Are Shown Using a Series of BoundFields (Click to view full-size image)

Step 2: Combining the Price, Units In Stock, and Units On Order Into One Row

The DetailsView has a row for the `UnitPrice`, `UnitsInStock`, and `UnitsOnOrder` fields. We can combine these data fields into a single row with a TemplateField either by adding a new TemplateField or by converting one of the existing `UnitPrice`, `UnitsInStock`, and `UnitsOnOrder` BoundFields into a TemplateField. While I personally prefer converting existing BoundFields, let's practice by adding a new TemplateField.

Start by clicking on the Edit Fields link in the DetailsView's smart tag to bring up the Fields dialog box. Next, add a new TemplateField and set its `HeaderText` property to "Price and Inventory" and move the new TemplateField so

that it is positioned above the `UnitPrice` BoundField.

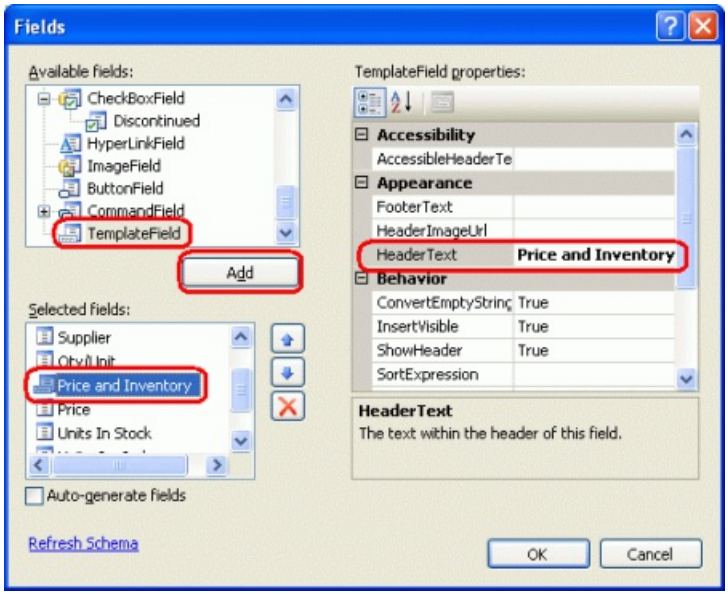


Figure 4: Add a New TemplateField to the DetailsView Control (Click to view full-size image)

Since this new TemplateField will contain the values currently displayed in the `UnitPrice`, `UnitsInStock`, and `UnitsOnOrder` BoundFields, let's remove them.

The last task for this step is to define the `ItemTemplate` markup for the Price and Inventory TemplateField, which can be accomplished either through the DetailsView's template editing interface in the Designer or by hand through the control's declarative syntax. As with the GridView, the DetailsView's template editing interface can be accessed by clicking on the Edit Templates link in the smart tag. From here you can select the template to edit from the drop-down list and then add any Web controls from the Toolbox.

For this tutorial, start by adding a Label control to the Price and Inventory TemplateField's `ItemTemplate`. Next, click on the Edit DataBindings link from the Label Web control's smart tag and bind the `Text` property to the `UnitPrice` field.

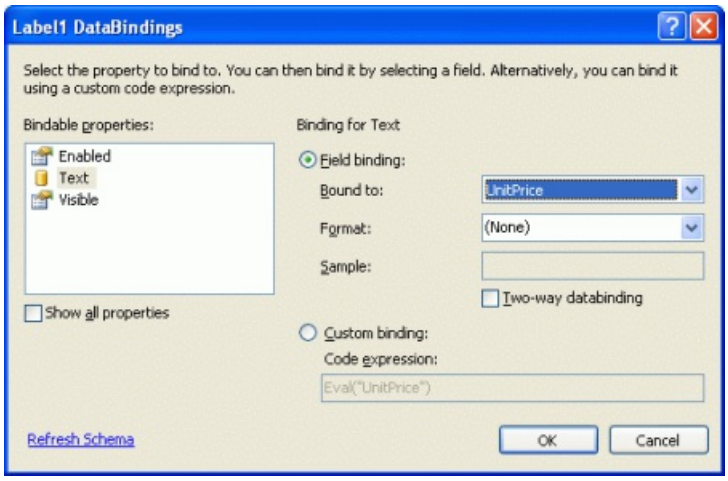


Figure 5: Bind the Label's `Text` Property to the `UnitPrice` Data Field (Click to view full-size image)

Formatting the Price as a Currency

With this addition, the Label Web control Price and Inventory TemplateField will now display just the price for the selected product. Figure 6 shows a screen shot of our progress thus far when viewed through a browser.

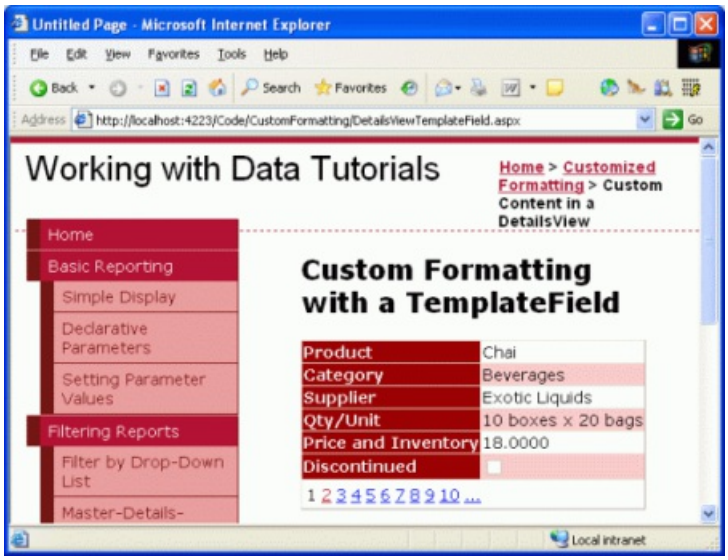


Figure 6: The Price and Inventory TemplateField Shows the Price (Click to view full-size image)

Note that the product's price is not formatted as a currency. With a BoundField, formatting is possible by setting the `HtmlEncode` property to `false` and the `DataFormatString` property to `{0:formatSpecifier}`. For a TemplateField, however, any formatting instructions must be specified in the databinding syntax or through the use of a formatting method defined somewhere within the application's code (such as in the ASP.NET page's code-behind class).

To specify the formatting for the databinding syntax used in the Label Web control, return to the DataBindings dialog box by clicking on the Edit DataBindings link from the Label's smart tag. You can type the formatting instructions directly in the Format drop-down list or select one of the defined format strings. Like with the BoundField's `DataFormatString` property, the formatting is specified using `{0:formatSpecifier}`.

For the `UnitPrice` field use the currency formatting specified either by selecting the appropriate drop-down list value or by typing in `{0:C}` by hand.

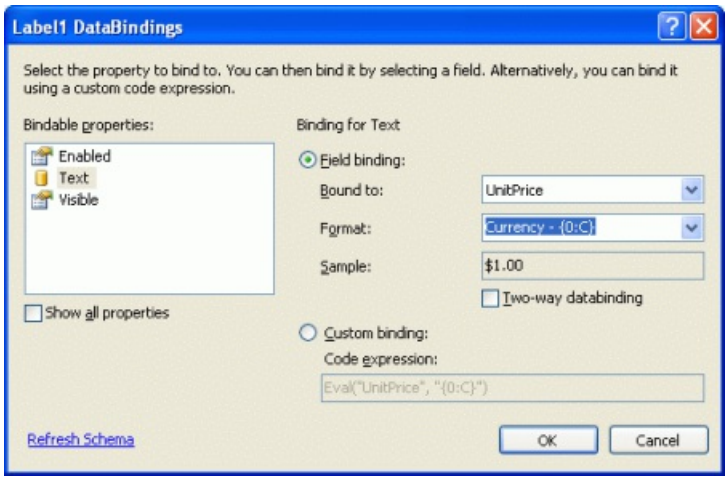


Figure 7: Format the Price as a Currency (Click to view full-size image)

Declaratively, the formatting specification is indicated as a second parameter into the `Bind` or `Eval` methods. The settings just made through the Designer result in the following databinding expression in the declarative markup:

```
<asp:Label ID="Label1" runat="server" Text='<%# Eval("UnitPrice", "{0:C}") %>'/>
```

Adding the Remaining Data Fields to the TemplateField

At this point we've displayed and formatted the `UnitPrice` data field in the Price and Inventory TemplateField, but still need to display the `UnitsInStock` and `UnitsOnOrder` fields. Let's display these on a line below the price and in parentheses. From the template editing interface in the Designer, such markup can be added by positioning your cursor within the template and simply typing in the text to be displayed. Alternatively, this markup can be entered directly in the declarative syntax.

Add the static markup, Label Web controls, and databinding syntax so that the Price and Inventory TemplateField displays the price and inventory information like so:

UnitPrice
(**In Stock / On Order:** *UnitsInStock* / *UnitsOnOrder*)

After performing this task your DetailsView's declarative markup should look similar to the following:

```
<asp:DetailsView ID="DetailsView1" runat="server" AutoGenerateRows="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1" AllowPaging="True"
    EnableViewState="False">
    <Fields>
        <asp:BoundField DataField="ProductName"
          HeaderText="Product" SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
          ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName"
          HeaderText="Supplier" ReadOnly="True"
          SortExpression="SupplierName" />
        <asp:BoundField DataField="QuantityPerUnit"
          HeaderText="Qty/Unit" SortExpression="QuantityPerUnit" />
        <asp:TemplateField HeaderText="Price and Inventory">
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                  Text='<%# Eval("UnitPrice", "{0:C}") %>'></asp:Label>
                <br />
                <strong>
                (In Stock / On Order: </strong>
                <asp:Label ID="Label2" runat="server"
                  Text='<%# Eval("UnitsInStock") %>'></asp:Label>
                <strong>/</strong>
                <asp:Label ID="Label3" runat="server"
                  Text='<%# Eval("UnitsOnOrder") %>'>
                </asp:Label><strong>)</strong>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:CheckBoxField DataField="Discontinued"
           HeaderText="Discontinued" SortExpression="Discontinued" />
    </Fields>
</asp:DetailsView>
```

With these changes we've consolidated the price and inventory information into a single DetailsView row.

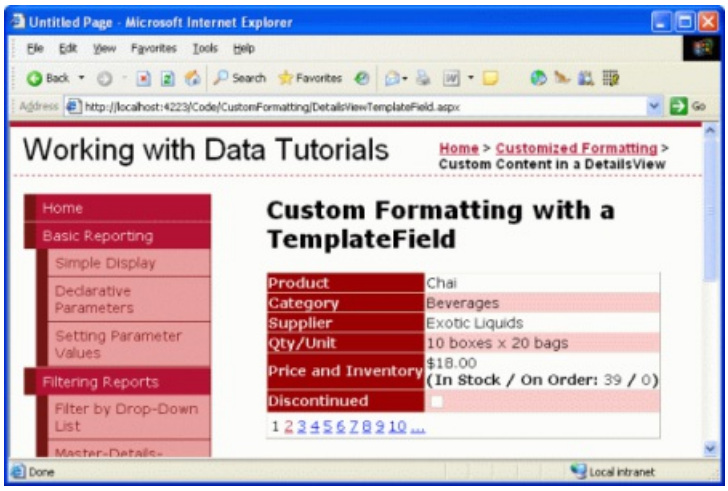


Figure 8: The Price and Inventory Information is Displayed in a Single Row (Click to view full-size image)

Step 3: Customizing the Discontinued Field Information

The `Products` table's `Discontinued` column is a bit value that indicates whether the product has been discontinued. When binding a DetailsView (or GridView) to a data source control, the Boolean value fields, like `Discontinued`, are implemented as CheckBoxFields whereas non-Boolean value fields, like `ProductID`, `ProductName`, and so on, are implemented as BoundFields. The CheckBoxField renders as a disabled checkbox that is checked if the data field's value is True and unchecked otherwise.

Rather than display the CheckBoxField we may want to instead display text indicating whether or not the product is discontinued. To accomplish this we could remove the CheckBoxField from the DetailsView and then add a BoundField whose `DataField` property was set to `Discontinued`. Take a moment to do this. After this change the DetailsView shows the text "True" for discontinued products and "False" for products that are still active.

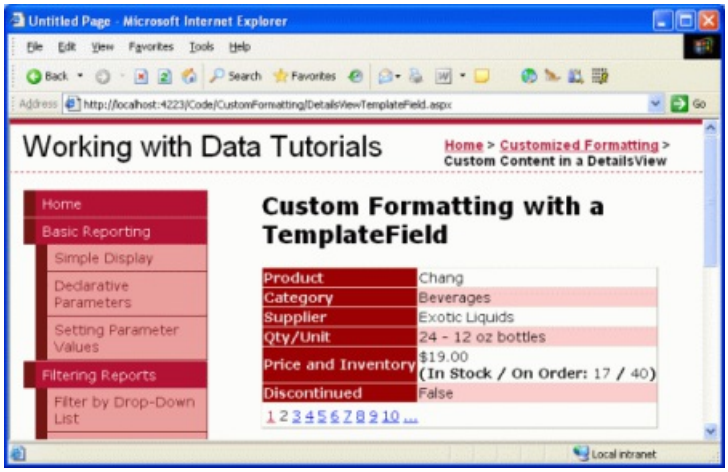


Figure 9: The Strings True and False Are Used to Display the Discontinued State (Click to view full-size image)

Imagine that we didn't want the strings "True" or "False" to be used, but "YES" and "NO", instead. Such customization can be performed with the aid of a TemplateField and a formatting method. A formatting method can take in any number of input parameters, but must return the HTML (as a string) to inject into the template.

Add a formatting method to the `DetailsViewTemplateField.aspx` page's code-behind class named `DisplayDiscontinuedAsYESorNO` that accepts a Boolean as an input parameter and returns a string. As discussed in the previous tutorial, this method *must* be marked as `protected` or `public` in order to be accessible from the template.

```
protected string DisplayDiscontinuedAsYESorNO(bool discontinued)
{
    if (discontinued)
        return "YES";
    else
        return "NO";
}
```

This method checks the input parameter (`discontinued`) and returns "YES" if it is `true`, "NO" otherwise.

> **NOTE**
> In the formatting method examined in the previous tutorial recall that we were passing in a data field that might contain `NULL` s and therefore needed to check if the employee's `HiredDate` property value had a database `NULL` value before accessing the `EmployeesRow` 's `HiredDate` property. Such a check is not needed here since the `Discontinued` column can never have database `NULL` values assigned. Moreover, this is why the method can accept a Boolean input parameter rather than having to accept a `ProductsRow` instance or a parameter of type `object`.

With this formatting method complete, all that remains is to call it from the TemplateField's `ItemTemplate`. To create the TemplateField either remove the `Discontinued` BoundField and add a new TemplateField or convert the `Discontinued` BoundField into a TemplateField. Then, from the declarative markup view, edit the TemplateField so that it contains just an ItemTemplate that invokes the `DisplayDiscontinuedAsYESorNO` method, passing in the value of the current `ProductRow` instance's `Discontinued` property. This can be accessed via the `Eval` method. Specifically, the TemplateField's markup should look like:

```
<asp:TemplateField HeaderText="Discontinued" SortExpression="Discontinued">
    <ItemTemplate>
        <%# DisplayDiscontinuedAsYESorNO((bool)
          Eval("Discontinued")) %>
    </ItemTemplate>
</asp:TemplateField>
```

This will cause the `DisplayDiscontinuedAsYESorNO` method to be invoked when rendering the DetailsView, passing in the `ProductRow` instance's `Discontinued` value. Since the `Eval` method returns a value of type `object`, but the `DisplayDiscontinuedAsYESorNO` method expects an input parameter of type `bool`, we cast the `Eval` methods return value to `bool`. The `DisplayDiscontinuedAsYESorNO` method will then return "YES" or "NO" depending on the value it receives. The returned value is what is displayed in this DetailsView row (see Figure 10).

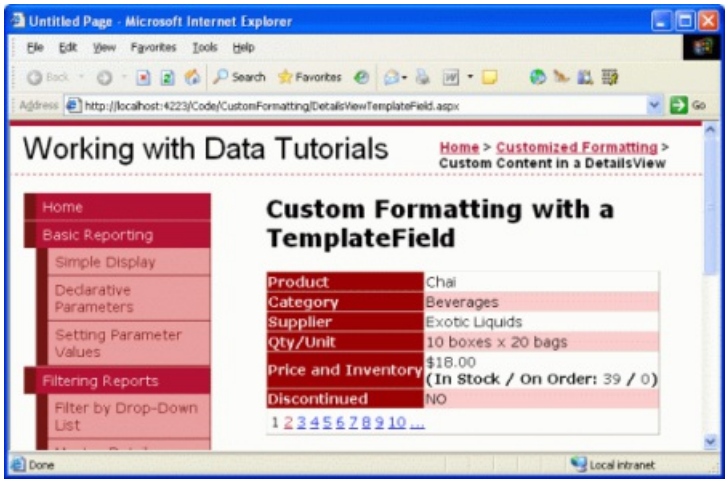


Figure 10: YES or NO Values are Now Shown in the Discontinued Row (Click to view full-size image)

Summary

The TemplateField in the DetailsView control allows for a higher degree of flexibility in displaying data than is available with the other field controls and are ideal for situations where:

- Multiple data fields need to be displayed in one GridView column
- The data is best expressed using a Web control rather than plain text
- The output depends on the underlying data, such as displaying metadata or in reformatting the data

While TemplateFields allow for a greater degree of flexibility in the rendering of the DetailsView's underlying data, the DetailsView output still feels a bit boxy as each field is rendered as a row in an HTML `<table>`.

The FormView control offers a greater degree of flexibility in configuring the rendered output. The FormView does not contain fields but rather just a series of templates (`ItemTemplate`, `EditItemTemplate`, `HeaderTemplate`, and so on). We'll see how to use the FormView to achieve even more control of the rendered layout in our next tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Dan Jagers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Using the FormView's Templates (C#)

2/20/2020 • 6 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> Unlike the DetailsView, the FormView is not composed of fields. Instead, the FormView is rendered using templates. In this tutorial we'll examine using the FormView control to present a less rigid display of data.

Introduction

In the last two tutorials we saw how to customize the GridView and DetailsView controls' outputs using TemplateFields. TemplateFields allow for the contents for a specific field to be highly customized, but in the end both the GridView and DetailsView have a rather boxy, grid-like appearance. For many scenarios such a grid-like layout is ideal, but at times a more fluid, less rigid display is needed. When displaying a single record, such a fluid layout is possible using the FormView control.

Unlike the DetailsView, the FormView is not composed of fields. You can't add a BoundField or TemplateField to a FormView. Instead, the FormView is rendered using templates. Think of the FormView as a DetailsView control that contains a single TemplateField. The FormView supports the following templates:

- `ItemTemplate` used to render the particular record displayed in the FormView
- `HeaderTemplate` used to specify an optional header row
- `FooterTemplate` used to specify an optional footer row
- `EmptyDataTemplate` when the FormView's `DataSource` lacks any records, the `EmptyDataTemplate` is used in place of the `ItemTemplate` for rendering the control's markup
- `PagerTemplate` can be used to customize the paging interface for FormViews that have paging enabled
- `EditItemTemplate` / `InsertItemTemplate` used to customize the editing interface or inserting interface for FormViews that support such functionality

In this tutorial we'll examine using the FormView control to present a less rigid display of products. Rather than having fields for the name, category, supplier, and so on, the FormView's `ItemTemplate` will show these values using a combination of a header element and a `<table>` (see Figure 1).

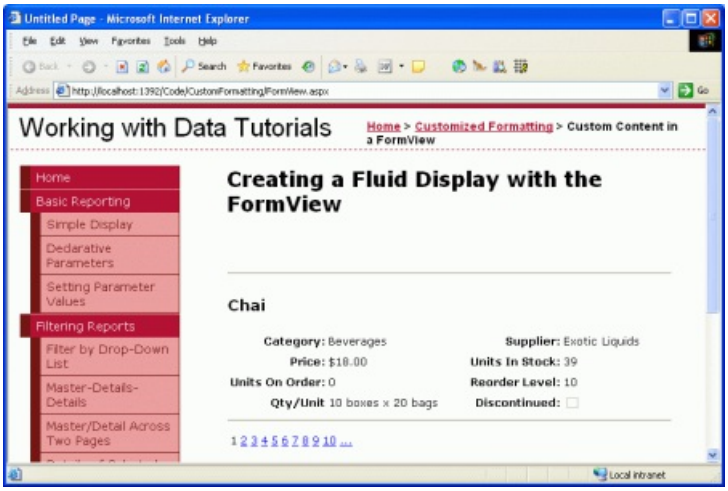


Figure 1: The FormView Breaks Out of the Grid-Like Layout Seen in the DetailsView (Click to view full-size image)

Step 1: Binding the Data to the FormView

Open the `FormView.aspx` page and drag a FormView from the Toolbox onto the Designer. When first adding the FormView it appears as a gray box, instructing us that an `ItemTemplate` is needed.

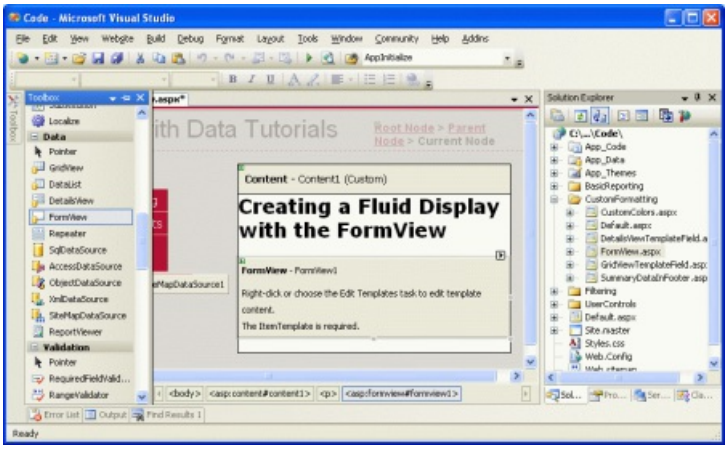


Figure 2: The FormView Cannot be Rendered in the Designer Until an `ItemTemplate` is Provided (Click to view full-size image)

The `ItemTemplate` can be created by hand (through the declarative syntax) or can be auto-created by binding the FormView to a data source control through the Designer. This auto-created `ItemTemplate` contains HTML that lists the name of each field and a Label control whose `Text` property is bound to the field's value. This approach also auto-creates an `InsertItemTemplate` and `EditItemTemplate`, both of which are populated with input controls for each of the data fields returned by the data source control.

If you want to auto-create the template, from the FormView's smart tag add a new ObjectDataSource control that invokes the `ProductsBLL` class's `GetProducts()` method. This will create a FormView with an `ItemTemplate`, `InsertItemTemplate`, and `EditItemTemplate`. From the Source view, remove the `InsertItemTemplate` and `EditItemTemplate` since we're not interested in creating a FormView that supports editing or inserting yet. Next, clear out the markup within the `ItemTemplate` so that we have a clean slate to work from.

If you'd rather build up the `ItemTemplate` manually, you can add and configure the ObjectDataSource by dragging it from the Toolbox onto the Designer. However, don't set the FormView's data source from the Designer. Instead, go to the Source view and manually set the FormView's `DataSourceID` property to the `ID` value of the ObjectDataSource. Next, manually add the `ItemTemplate`.

Regardless of what approach you decided to take, at this point your FormView's declarative markup should look like:

```
<asp:FormView ID="FormView1" runat="server" DataSourceID="ObjectDataSource1">
    <ItemTemplate>

    </ItemTemplate>
</asp:FormView>
```

Take a moment to check the Enable Paging checkbox in the FormView's smart tag; this will add the `AllowPaging="True"` attribute to the FormView's declarative syntax. Also, set the `EnableViewState` property to False.

Step 2: Defining the `ItemTemplate`'s Markup

With the FormView bound to the ObjectDataSource control and configured to support paging we're ready to specify the content for the `ItemTemplate`. For this tutorial, let's have the product's name displayed in an `<h3>`

heading. Following that, let's use an HTML `<table>` to display the remaining product properties in a four-column table where the first and third columns list the property names and the second and fourth list their values.

This markup can be entered in through the FormView's template editing interface in the Designer or entered manually through the declarative syntax. When working with templates I typically find it quicker to work directly with the declarative syntax, but feel free to use whatever technique you're most comfortable with.

The following markup shows the FormView declarative markup after the `ItemTemplate`'s structure has been completed:

```
<asp:FormView ID="FormView1" runat="server" DataSourceID="ObjectDataSource1"
    AllowPaging="True" EnableViewState="False">
    <ItemTemplate>
        <hr />
        <h3><%# Eval("ProductName") %></h3>
        <table border="0">
            <tr>
                <td class="ProductPropertyLabel">Category:</td>
                <td class="ProductPropertyValue">
                  <%# Eval("CategoryName") %></td>
                <td class="ProductPropertyLabel">Supplier:</td>
                <td class="ProductPropertyValue">
                  <%# Eval("SupplierName")%></td>
            </tr>
            <tr>
                <td class="ProductPropertyLabel">Price:</td>
                <td class="ProductPropertyValue"><%# Eval("UnitPrice",
                  "{0:C}") %></td>
                <td class="ProductPropertyLabel">Units In Stock:</td>
                <td class="ProductPropertyValue">
                  <%# Eval("UnitsInStock")%></td>
            </tr>
            <tr>
                <td class="ProductPropertyLabel">Units On Order:</td>
                <td class="ProductPropertyValue">
                  <%# Eval("UnitsOnOrder") %></td>
                <td class="ProductPropertyLabel">Reorder Level:</td>
                <td class="ProductPropertyValue">
                  <%# Eval("ReorderLevel")%></td>
            </tr>
            <tr>
                <td class="ProductPropertyLabel">Qty/Unit</td>
                <td class="ProductPropertyValue">
                  <%# Eval("QuantityPerUnit") %></td>
                <td class="ProductPropertyLabel">Discontinued:</td>
                <td class="ProductPropertyValue">
                    <asp:CheckBox runat="server" Enabled="false"
                      Checked='<%# Eval("Discontinued") %>' />
                </td>
            </tr>
        </table>
        <hr />
    </ItemTemplate>
</asp:FormView>
```

Notice that the databinding syntax - `<%# Eval("ProductName") %>`, for example can be injected directly into the template's output. That is, it need not be assigned to a Label control's `Text` property. For example, we have the `ProductName` value displayed in an `<h3>` element using `<h3><%# Eval("ProductName") %></h3>`, which for the product Chai will render as `<h3>Chai</h3>`.

The `ProductPropertyLabel` and `ProductPropertyValue` CSS classes are used for specifying the style of the product property names and values in the `<table>`. These CSS classes are defined in `Styles.css` and cause the property names to be bold and right-aligned and add a right padding to the property values.

Since there are no CheckBoxFields available with the FormView, in order to show the `Discontinued` value as a checkbox we must add our own CheckBox control. The `Enabled` property is set to False, making it read-only, and the CheckBox's `Checked` property is bound to the value of the `Discontinued` data field.

With the `ItemTemplate` complete, the product information is displayed in a much more fluid manner. Compare the DetailsView output from the last tutorial (Figure 3) with the output generated by the FormView in this tutorial (Figure 4).

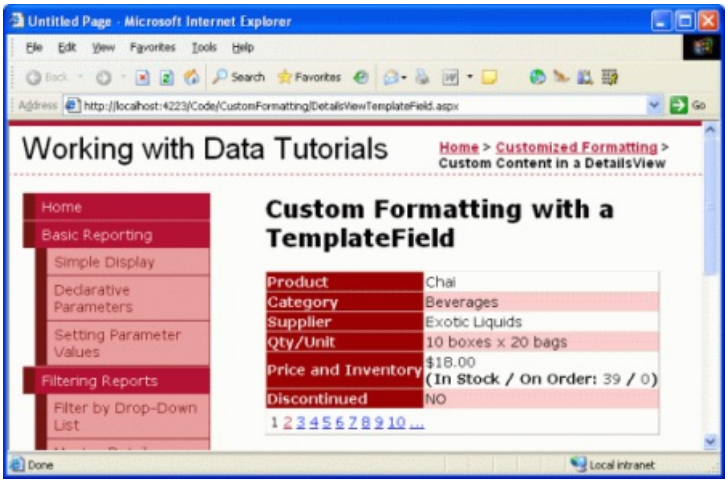


Figure 3: The Rigid DetailsView Output (Click to view full-size image)

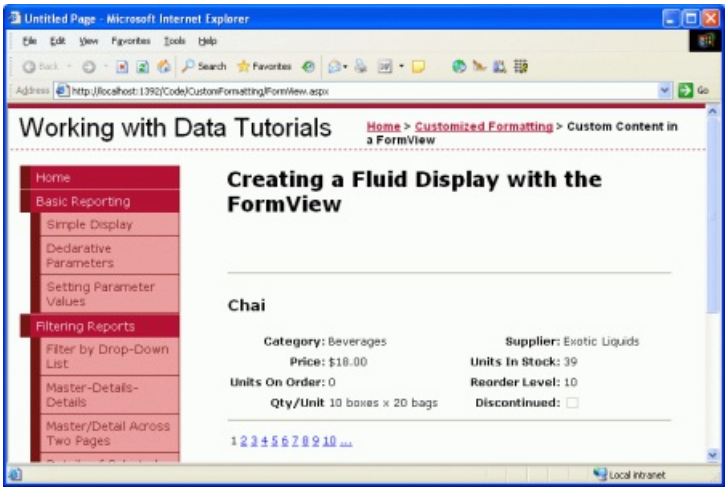


Figure 4: The Fluid FormView Output (Click to view full-size image)

Summary

While the GridView and DetailsView controls can have their output customized using TemplateFields, both still present their data in a grid-like, boxy format. For those times when a single record needs to be shown using a less rigid layout, the FormView is an ideal choice. Like the DetailsView, the FormView renders a single record from its `DataSource`, but unlike the DetailsView it is composed just of templates and does not support fields.

As we saw in this tutorial, the FormView allows for a more flexible layout when displaying a single record. In future tutorials we'll examine the DataList and Repeater controls, which provide the same level of flexibility as the FormsView, but are able to display multiple records (like the GridView).

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest

book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was E.R. Gilmore. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Displaying Summary Information in the GridView's Footer (C#)

2/20/2020 • 11 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> Summary information is often displayed at the bottom of the report in a summary row. The GridView control can include a footer row into whose cells we can programmatically inject aggregate data. In this tutorial we'll see how to display aggregate data in this footer row.

Introduction

In addition to seeing each of the products' prices, units in stock, units on order, and reorder levels, a user might also be interested in aggregate information, such as the average price, the total number of units in stock, and so on. Such summary information is often displayed at the bottom of the report in a summary row. The GridView control can include a footer row into whose cells we can programmatically inject aggregate data.

This task presents us with three challenges:

1. Configuring the GridView to display its footer row
2. Determining the summary data; that is, how do we compute the average price or the total of the units in stock?
3. Injecting the summary data into the appropriate cells of the footer row

In this tutorial we'll see how to overcome these challenges. Specifically, we'll create a page that lists the categories in a drop-down list with the selected category's products displayed in a GridView. The GridView will include a footer row that shows the average price and total number of units in stock and on order for products in that category.

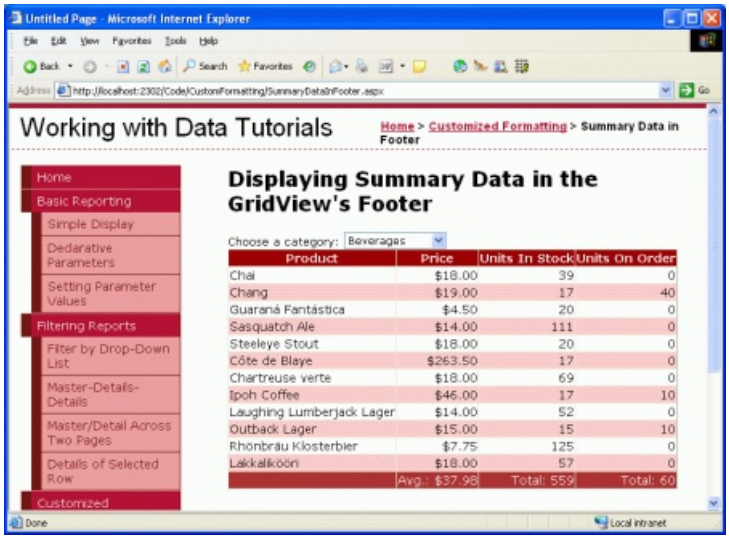


Figure 1: Summary Information is Displayed in the GridView's Footer Row (Click to view full-size image)

This tutorial, with its category to products master/detail interface, builds upon the concepts covered in the earlier Master/Detail Filtering With a DropDownList tutorial. If you've not yet worked through the earlier tutorial, please do so before continuing on with this one.

Step 1: Adding the Categories DropDownList and Products GridView

Before concerning ourselves with adding summary information to the GridView's footer, let's first simply build the master/detail report. Once we've completed this first step, we'll look at how to include summary data.

Start by opening the `SummaryDataInFooter.aspx` page in the `CustomFormatting` folder. Add a DropDownList control and set its `ID` to `Categories`. Next, click on the Choose Data Source link from the DropDownList's smart tag and opt to add a new ObjectDataSource named `CategoriesDataSource` that invokes the `CategoriesBLL` class's `GetCategories()` method.

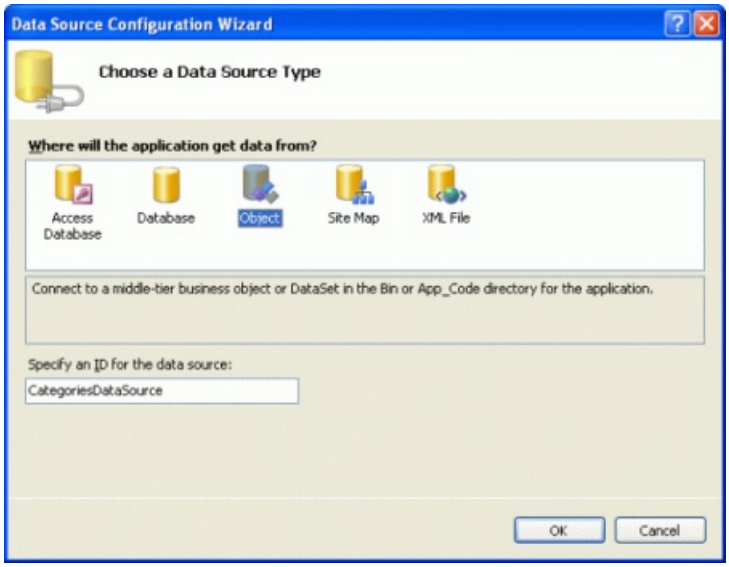


Figure 2: Add a New ObjectDataSource Named `CategoriesDataSource` (Click to view full-size image)

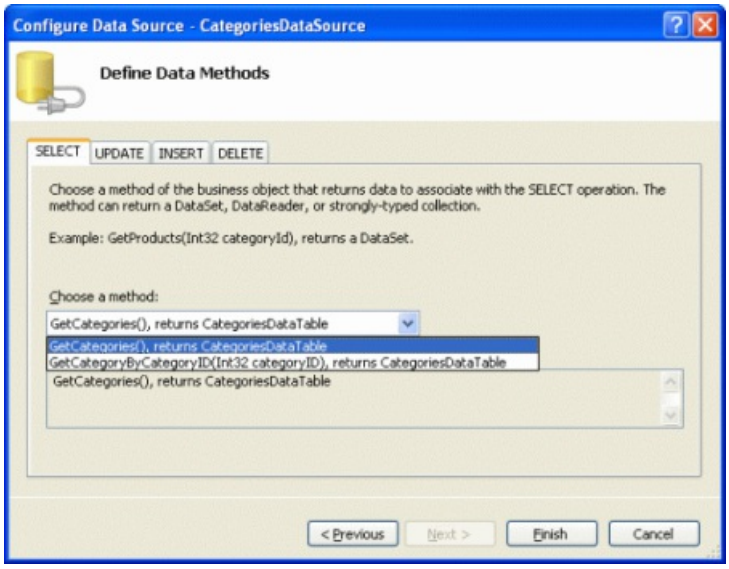


Figure 3: Have the ObjectDataSource Invoke the `CategoriesBLL` Class's `GetCategories()` Method (Click to view full-size image)

After configuring the ObjectDataSource, the wizard returns us to the DropDownList's Data Source Configuration wizard from which we need to specify what data field value should be displayed and which one should correspond to the value of the DropDownList's `ListItem` s. Have the `CategoryName` field displayed and use the `CategoryID` as the value.

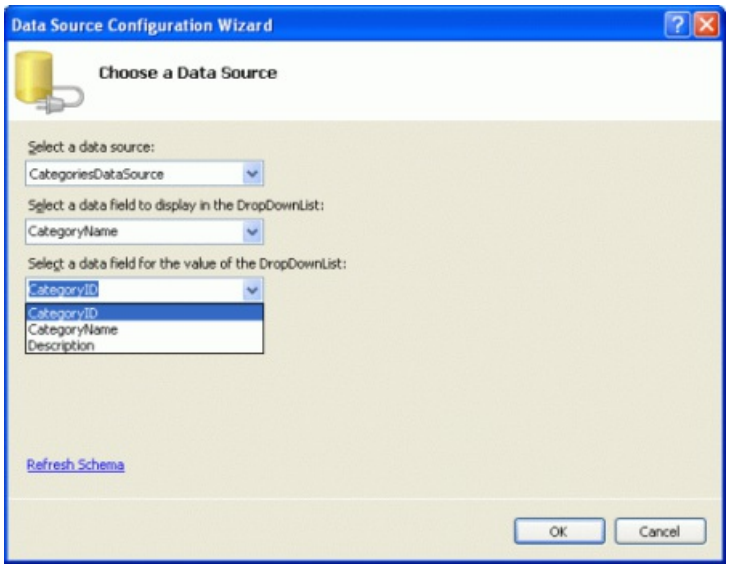


Figure 4: Use the `CategoryName` and `CategoryID` Fields as the `Text` and `Value` for the `ListItem` s, Respectively (Click to view full-size image)

At this point we have a DropDownList (`Categories`) that lists the categories in the system. We now need to add a GridView that lists those products that belong to the selected category. Before we do, though, take a moment to check the Enable AutoPostBack checkbox in the DropDownList's smart tag. As discussed in the *Master/Detail Filtering With a DropDownList* tutorial, by setting the DropDownList's `AutoPostBack` property to `true` the page will be posted back each time the DropDownList value is changed. This will cause the GridView to be refreshed, showing those products for the newly selected category. If the `AutoPostBack` property is set to `false` (the default), changing the category won't cause a postback and therefore won't update the listed products.

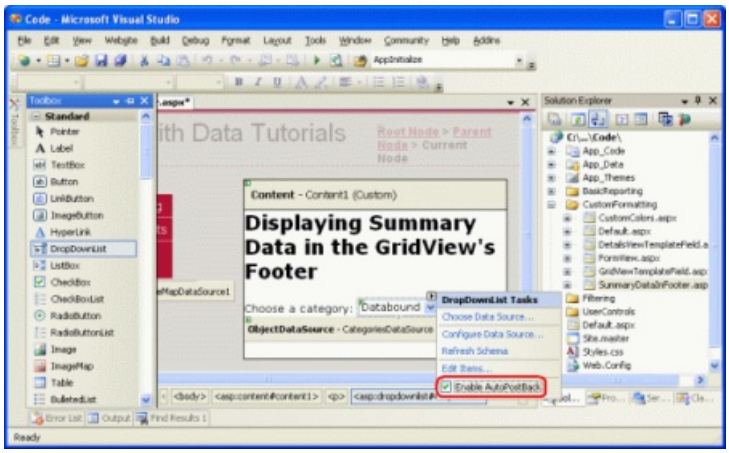


Figure 5: Check the Enable AutoPostBack Checkbox in the DropDownList's Smart Tag (Click to view full-size image)

Add a GridView control to the page in order to display the products for the selected category. Set the GridView's `ID` to `ProductsInCategory` and bind it to a new ObjectDataSource named `ProductsInCategoryDataSource`.

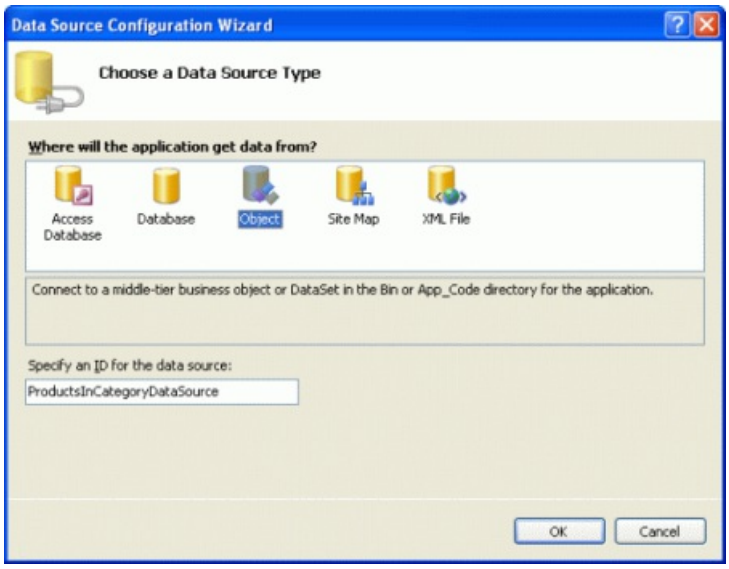


Figure 6: Add a New ObjectDataSource Named `ProductsInCategoryDataSource` (Click to view full-size image)

Configure the ObjectDataSource so that it invokes the `ProductsBLL` class's `GetProductsByCategoryID(categoryID)` method.

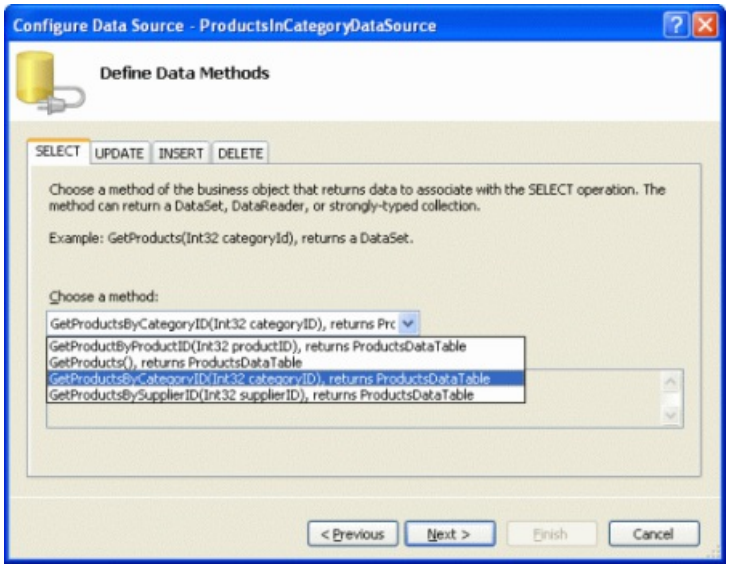


Figure 7: Have the ObjectDataSource Invoke the `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

Since the `GetProductsByCategoryID(categoryID)` method takes in an input parameter, in the final step of the wizard we can specify the source of the parameter value. In order to display those products from the selected category, have the parameter pulled from the `Categories` DropDownList.

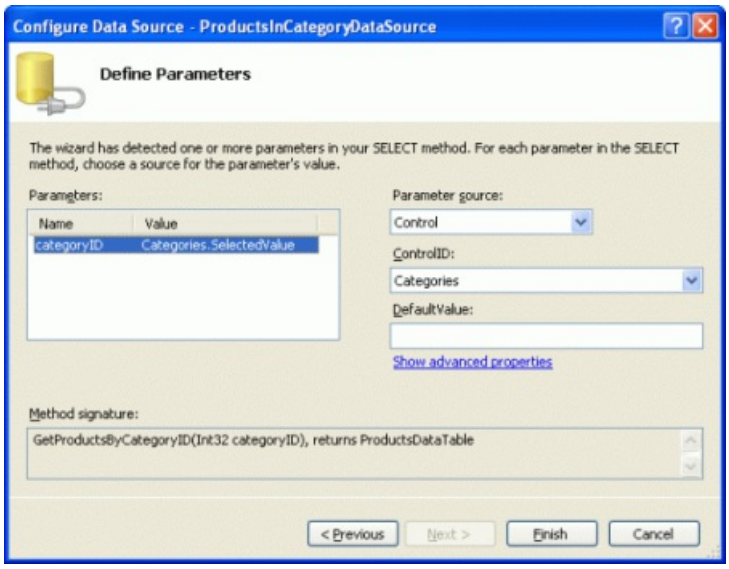


Figure 8: Get the `categoryID` Parameter Value from the Selected Categories DropDownList (Click to view full-size image)

After completing the wizard the GridView will have a BoundField for each of the product properties. Let's clean up these BoundFields so that only the `ProductName`, `UnitPrice`, `UnitsInStock`, and `UnitsOnOrder` BoundFields are displayed. Feel free to add any field-level settings to the remaining BoundFields (such as formatting the `UnitPrice` as a currency). After making these changes, the GridView's declarative markup should look similar to the following:

```
<asp:GridView ID="ProductsInCategory" runat="server"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ProductsInCategoryDataSource" EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
          SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
            HeaderText="Price"
            HtmlEncode="False" SortExpression="UnitPrice">
            <ItemStyle HorizontalAlign="Right" />
        </asp:BoundField>
        <asp:BoundField DataField="UnitsInStock"
         HeaderText="Units In Stock" SortExpression="UnitsInStock">
            <ItemStyle HorizontalAlign="Right" />
        </asp:BoundField>
        <asp:BoundField DataField="UnitsOnOrder"
           HeaderText="Units On Order" SortExpression="UnitsOnOrder">
            <ItemStyle HorizontalAlign="Right" />
        </asp:BoundField>
    </Columns>
</asp:GridView>
```

At this point we have a fully functioning master/detail report that shows the name, unit price, units in stock, and units on order for those products that belong to the selected category.

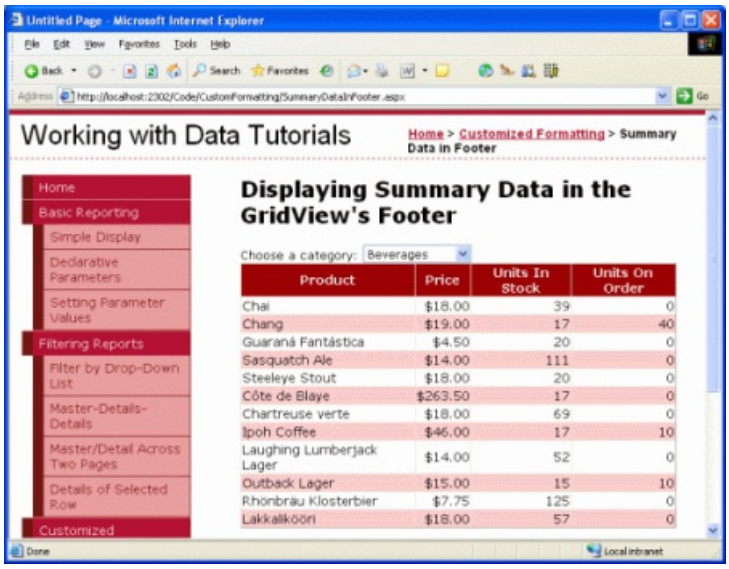


Figure 9: Get the `categoryID` Parameter Value from the Selected Categories DropDownList (Click to view full-size image)

Step 2: Displaying a Footer in the GridView

The GridView control can display both a header and footer row. These rows are displayed depending on the values of the `ShowHeader` and `ShowFooter` properties, respectively, with `ShowHeader` defaulting to `true` and `ShowFooter` to `false`. To include a footer in the GridView simply set its `ShowFooter` property to `true`.

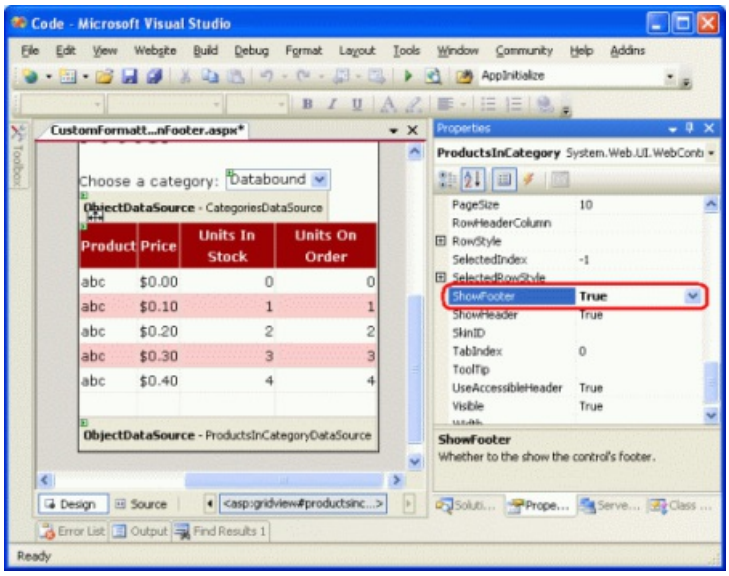


Figure 10: Set the GridView's `ShowFooter` Property to `true` (Click to view full-size image)

The footer row has a cell for each of the fields defined in the GridView; however, these cells are empty by default. Take a moment to view our progress in a browser. With the `ShowFooter` property now set to `true`, the GridView includes an empty footer row.

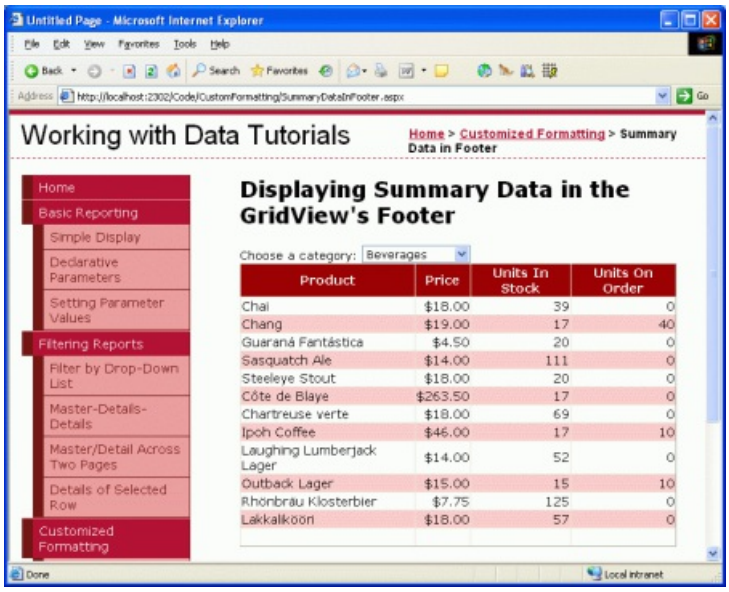


Figure 11: The GridView Now Includes a Footer Row (Click to view full-size image)

The footer row in Figure 11 doesn't stand out, as it has a white background. Let's create a `FooterStyle` CSS class in `Styles.css` that specifies a dark red background and then configure the `GridView.skin` Skin file in the `DataWebControls` Theme to assign this CSS class to the GridView's `FooterStyle`'s `CssClass` property. If you need to brush up on Skins and Themes, refer back to the Displaying Data With the ObjectDataSource tutorial.

Start by adding the following CSS class to `Styles.css`:

```
.FooterStyle
{
    background-color: #a33;
    color: White;
    text-align: right;
}
```

The `FooterStyle` CSS class is similar in style to the `HeaderStyle` class, although the `HeaderStyle`'s background color is subtlety darker and its text is displayed in a bold font. Furthermore, the text in the footer is right-aligned whereas the header's text is centered.

Next, to associate this CSS class with every GridView's footer, open the `GridView.skin` file in the `DataWebControls` Theme and set the `FooterStyle`'s `CssClass` property. After this addition the file's markup should look like:

```
<asp:GridView runat="server" CssClass="DataWebControlStyle">
   <AlternatingRowStyle CssClass="AlternatingRowStyle" />
   <RowStyle CssClass="RowStyle" />
   <HeaderStyle CssClass="HeaderStyle" />
   <FooterStyle CssClass="FooterStyle" />
   <SelectedRowStyle CssClass="SelectedRowStyle" />
</asp:GridView>
```

As the screen shot below shows, this change makes the footer stand out more clearly.

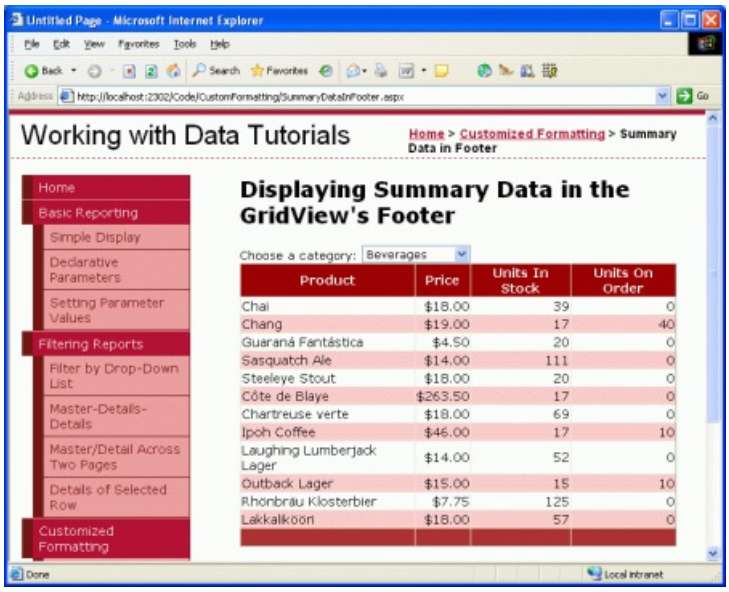


Figure 12: The GridView's Footer Row Now Has a Reddish Background Color (Click to view full-size image)

Step 3: Computing the Summary Data

With the GridView's footer displayed, the next challenge facing us is how to compute the summary data. There are two ways to compute this aggregate information:

1. Through a SQL query we could issue an additional query to the database to compute the summary data for a particular category. SQL includes a number of aggregate functions along with a `GROUP BY` clause to specify the data over which the data should be summarized. The following SQL query would bring back the needed information:

   ```
   SELECT CategoryID, AVG(UnitPrice), SUM(UnitsInStock),
   SUM(UnitsOnOrder)
   FROM Products
   WHERE CategoryID = categoryID
   GROUP BY CategoryID
   ```

   Of course you wouldn't want to issue this query directly from the `SummaryDataInFooter.aspx` page, but rather by creating a method in the `ProductsTableAdapter` and the `ProductsBLL`.

2. Compute this information as it's being added to the GridView as discussed in Custom Formatting Based Upon Data tutorial, the GridView's `RowDataBound` event handler fires once for each row being added to the GridView after its been databound. By creating an event handler for this event we can keep a running total of the values we want to aggregate. After the last data row has been bound to the GridView we have the totals and the information needed to compute the average.

I typically employ the second approach as it saves a trip to the database and the effort needed to implement the summary functionality in the Data Access Layer and Business Logic Layer, but either approach would suffice. For this tutorial let's use the second option and keep track of the running total using the `RowDataBound` event handler.

Create a `RowDataBound` event handler for the GridView by selecting the GridView in the Designer, clicking the lightning bolt icon from the Properties window, and double-clicking the `RowDataBound` event. This will create a new event handler named `ProductsInCategory_RowDataBound` in the `SummaryDataInFooter.aspx` page's code-behind class.

```
protected void ProductsInCategory_RowDataBound
    (object sender, GridViewRowEventArgs e)
{
}
```

In order to maintain a running total we need to define variables outside of the scope of the event handler. Create the following four page-level variables:

- `_totalUnitPrice`, of type `decimal`
- `_totalNonNullUnitPriceCount`, of type `int`
- `_totalUnitsInStock`, of type `int`
- `_totalUnitsOnOrder`, of type `int`

Next, write the code to increment these three variables for each data row encountered in the `RowDataBound` event handler.

```
// Class-scope, running total variables...
decimal _totalUnitPrice = 0m;
int _totalNonNullUnitPriceCount = 0;
int _totalUnitsInStock = 0;
int _totalUnitsOnOrder = 0;
protected void ProductsInCategory_RowDataBound(object sender,
  GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        // Reference the ProductsRow via the e.Row.DataItem property
        Northwind.ProductsRow product =
          (Northwind.ProductsRow)
          ((System.Data.DataRowView)e.Row.DataItem).Row;
        // Increment the running totals (if they are not NULL!)
        if (!product.IsUnitPriceNull())
        {
            _totalUnitPrice += product.UnitPrice;
            _totalNonNullUnitPriceCount++;
        }
        if (!product.IsUnitsInStockNull())
            _totalUnitsInStock += product.UnitsInStock;
        if (!product.IsUnitsOnOrderNull())
            _totalUnitsOnOrder += product.UnitsOnOrder;
    }
}
```

The `RowDataBound` event handler starts by ensuring that we're dealing with a DataRow. Once that's been established, the `Northwind.ProductsRow` instance that was just bound to the `GridViewRow` object in `e.Row` is stored in the variable `product`. Next, running total variables are incremented by the current product's corresponding values (assuming that they don't contain a database `NULL` value). We keep track of both the running `UnitPrice` total and the number of non-`NULL` `UnitPrice` records because the average price is the quotient of these two numbers.

Step 4: Displaying the Summary Data in the Footer

With the summary data totaled, the last step is to display it in the GridView's footer row. This task, too, can be accomplished programmatically through the `RowDataBound` event handler. Recall that the `RowDataBound` event handler fires for *every* row that's bound to the GridView, including the footer row. Therefore, we can augment our event handler to display the data in the footer row using the following code:

```
protected void ProductsInCategory_RowDataBound
    (object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
      ... Increment the running totals ...
    }
    else if (e.Row.RowType == DataControlRowType.Footer)
    {
      ... Display the summary data in the footer ...
    }
}
```

Since the footer row is added to the GridView after all of the data rows have been added, we can be confident that by the time we're ready to display the summary data in the footer the running total calculations will have completed. The last step, then, is to set these values in the footer's cells.

To display text in a particular footer cell, use `e.Row.Cells[index].Text = value`, where the `Cells` indexing starts at 0. The following code computes the average price (the total price divided by the number of products) and displays it along with the total number of units in stock and units on order in the appropriate footer cells of the GridView.

```
protected void ProductsInCategory_RowDataBound
    (object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
      ... <i>Increment the running totals</i> ...
    }
    else if (e.Row.RowType == DataControlRowType.Footer)
    {
      // Determine the average UnitPrice
      decimal avgUnitPrice = _totalUnitPrice / (decimal) _totalNonNullUnitPriceCount;
      // Display the summary data in the appropriate cells
      e.Row.Cells[1].Text = "Avg.: " + avgUnitPrice.ToString("c");
      e.Row.Cells[2].Text = "Total: " + _totalUnitsInStock.ToString();
      e.Row.Cells[3].Text = "Total: " + _totalUnitsOnOrder.ToString();
    }
}
```

Figure 13 shows the report after this code has been added. Note how the `ToString("c")` causes the average price summary information to be formatted like a currency.

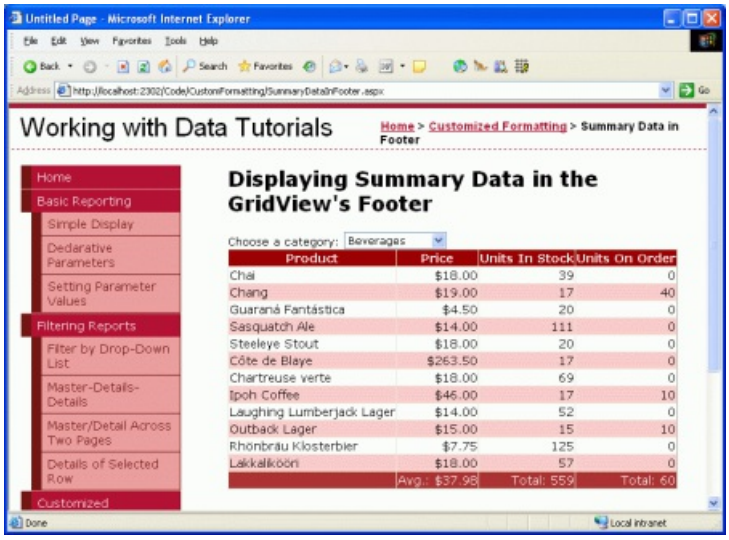


Figure 13: The GridView's Footer Row Now Has a Reddish Background Color (Click to view full-size image)

Summary

Displaying summary data is a common report requirement, and the GridView control makes it easy to include such information in its footer row. The footer row is displayed when the GridView's `ShowFooter` property is set to `true` and can have the text in its cells set programmatically through the `RowDataBound` event handler. Computing the summary data can either be done by re-querying the database or by using code in the ASP.NET page's code-behind class to programmatically compute the summary data.

This tutorial concludes our examination of custom formatting with the GridView, DetailsView, and FormView controls. Our next tutorial kicks off our exploration of inserting, updating, and deleting data using these same controls.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Custom Formatting Based Upon Data (VB)

2/20/2020 • 18 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> Adjusting the format of the GridView, DetailsView, or FormView based upon the data bound to it can be accomplished in multiple ways. In this tutorial we'll look at how to accomplish data bound formatting through the use of the DataBound and RowDataBound event handlers.

Introduction

The appearance of the GridView, DetailsView, and FormView controls can be customized through a myriad of style-related properties. Properties like `CssClass`, `Font`, `BorderWidth`, `BorderStyle`, `BorderColor`, `Width`, and `Height`, among others, dictate the general appearance of the rendered control. Properties including `HeaderStyle`, `RowStyle`, `AlternatingRowStyle`, and others allow these same style settings to be applied to particular sections. Likewise, these style settings can be applied at the field level.

In many scenarios though, the formatting requirements depend upon the value of the displayed data. For example, to draw attention to out of stock products, a report listing product information might set the background color to yellow for those products whose `UnitsInStock` and `UnitsOnOrder` fields are both equal to 0. To highlight the more expensive products, we may want to display the prices of those products costing more than $75.00 in a bold font.

Adjusting the format of the GridView, DetailsView, or FormView based upon the data bound to it can be accomplished in multiple ways. In this tutorial we'll look at how to accomplish data bound formatting through the use of the `DataBound` and `RowDataBound` event handlers. In the next tutorial we'll explore an alternative approach.

Using the DetailsView Control's `DataBound` Event Handler

When data is bound to a DetailsView, either from a data source control or through programmatically assigning data to the control's `DataSource` property and calling its `DataBind()` method, the following sequence of steps occur:

1. The data Web control's `DataBinding` event fires.
2. The data is bound to the data Web control.
3. The data Web control's `DataBound` event fires.

Custom logic can be injected immediately after steps 1 and 3 through an event handler. By creating an event handler for the `DataBound` event we can programmatically determine the data that has been bound to the data Web control and adjust the formatting as needed. To illustrate this let's create a DetailsView that will list general information about a product, but will display the `UnitPrice` value in a ***bold, italic font*** if it exceeds $75.00.

Step 1: Displaying the Product Information in a DetailsView

Open the `CustomColors.aspx` page in the `CustomFormatting` folder, drag a DetailsView control from the Toolbox onto the Designer, set its `ID` property value to `ExpensiveProductsPriceInBoldItalic`, and bind it to a new ObjectDataSource control that invokes the `ProductsBLL` class's `GetProducts()` method. The detailed steps for

accomplishing this are omitted here for brevity since we examined them in detail in previous tutorials.

Once you've bound the ObjectDataSource to the DetailsView, take a moment to modify the field list. I've opted to remove the `ProductID`, `SupplierID`, `CategoryID`, `UnitsInStock`, `UnitsOnOrder`, `ReorderLevel`, and `Discontinued` BoundFields and renamed and reformatted the remaining BoundFields. I also cleared out the `Width` and `Height` settings. Since the DetailsView displays only a single record, we need to enable paging in order to allow the end user to view all of the products. Do so by checking the Enable Paging checkbox in the DetailsView's smart tag.

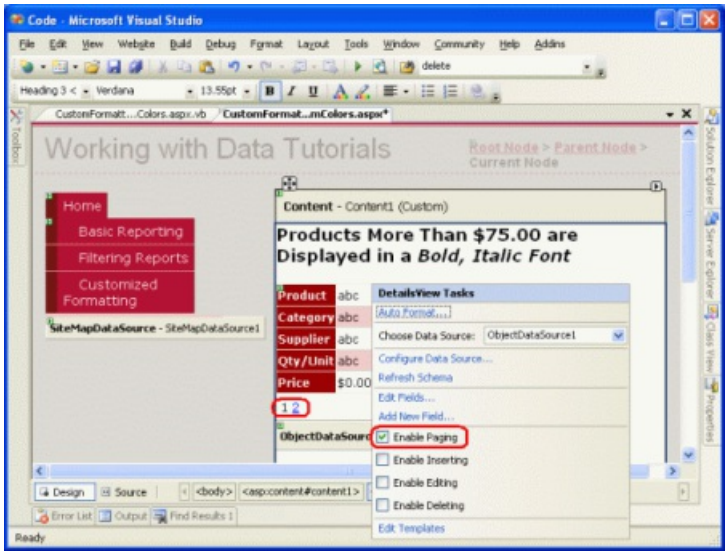


Figure 1: Figure 1: Check the Enable Paging Checkbox in the DetailsView's Smart Tag (Click to view full-size image)

After these changes, the DetailsView markup will be:

```
<asp:DetailsView ID="DetailsView1" runat="server" AllowPaging="True"
    AutoGenerateRows="False" DataKeyNames="ProductID"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <Fields>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
          SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
          ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
          ReadOnly="True" SortExpression="SupplierName" />
        <asp:BoundField DataField="QuantityPerUnit"
          HeaderText="Qty/Unit" SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
          HeaderText="Price"
            HtmlEncode="False" SortExpression="UnitPrice" />
    </Fields>
</asp:DetailsView>
```

Take a moment to test out this page in your browser.

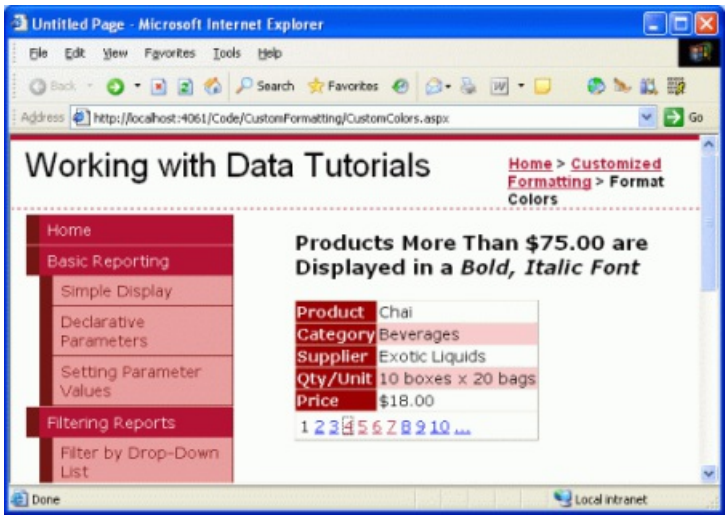


Figure 2: The DetailsView Control Displays One Product at a Time (Click to view full-size image)

Step 2: Programmatically Determining the Value of the Data in the DataBound Event Handler

In order to display the price in a bold, italic font for those products whose `UnitPrice` value exceeds $75.00, we need to first be able to programmatically determine the `UnitPrice` value. For the DetailsView, this can be accomplished in the `DataBound` event handler. To create the event handler click on the DetailsView in the Designer then navigate to the Properties window. Press F4 to bring it up, if it's not visible, or go to the View menu and select the Properties Window menu option. From the Properties window, click on the lightning bolt icon to list the DetailsView's events. Next, either double-click the `DataBound` event or type in the name of the event handler you want to create.

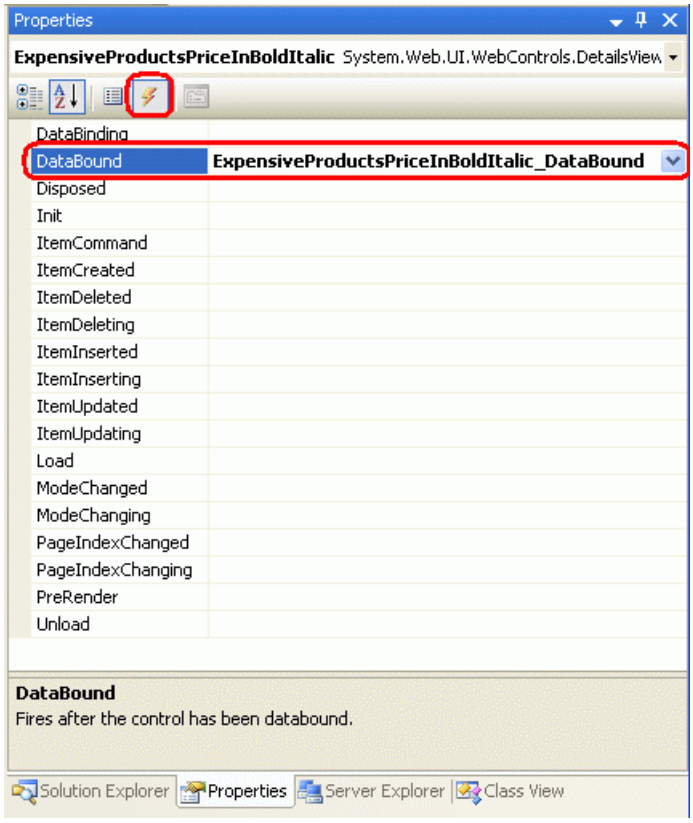


Figure 3: Create an Event Handler for the `DataBound` Event

> **NOTE**
> You can also create an event handler from the ASP.NET page's code portion. There you'll find two drop-down lists at the top of the page. Select the object from the left drop-down list and the event you want to create a handler for from the right drop-down list and Visual Studio will automatically create the appropriate event handler.

Doing so will automatically create the event handler and take you to the code portion where it has been added. At this point you will see:

```
Protected Sub ExpensiveProductsPriceInBoldItalic_DataBound _
    (sender As Object, e As System.EventArgs) _
    Handles ExpensiveProductsPriceInBoldItalic.DataBound
End Sub
```

The data bound to the DetailsView can be accessed via the `DataItem` property. Recall that we are binding our controls to a strongly-typed DataTable, which is composed of a collection of strongly-typed DataRow instances. When the DataTable is bound to the DetailsView, the first DataRow in the DataTable is assigned to the DetailsView's `DataItem` property. Specifically, the `DataItem` property is assigned a `DataRowView` object. We can use the `DataRowView`'s `Row` property to get access to the underlying DataRow object, which is actually a `ProductsRow` instance. Once we have this `ProductsRow` instance we can make our decision by simply inspecting the object's property values.

The following code illustrates how to determine whether the `UnitPrice` value bound to the DetailsView control is greater than $75.00:

```
Protected Sub ExpensiveProductsPriceInBoldItalic_DataBound _
    (sender As Object, e As System.EventArgs) _
    Handles ExpensiveProductsPriceInBoldItalic.DataBound

    Dim product As Northwind.ProductsRow = _
        CType(CType(ExpensiveProductsPriceInBoldItalic.DataItem, _
            System.Data.DataRowView).Row, Northwind.ProductsRow)
    If Not product.IsUnitPriceNull() AndAlso product.UnitPrice > 75 Then
    End If
End Sub
```

> **NOTE**
> Since `UnitPrice` can have a `NULL` value in the database, we first check to make sure that we're not dealing with a `NULL` value before accessing the `ProductsRow`'s `UnitPrice` property. This check is important because if we attempt to access the `UnitPrice` property when it has a `NULL` value the `ProductsRow` object will throw a StrongTypingException exception.

Step 3: Formatting the UnitPrice Value in the DetailsView

At this point we can determine whether the `UnitPrice` value bound to the DetailsView has a value that exceeds $75.00, but we've yet to see how to programmatically adjust the DetailsView's formatting accordingly. To modify the formatting of an entire row in the DetailsView, programmatically access the row using `DetailsViewID.Rows(index)`; to modify a particular cell, access use `DetailsViewID.Rows(index).Cells(index)`. Once we have a reference to the row or cell we can then adjust its appearance by setting its style-related properties.

Accessing a row programmatically requires that you know the row's index, which starts at 0. The `UnitPrice` row is the fifth row in the DetailsView, giving it an index of 4 and making it programmatically accessible using

`ExpensiveProductsPriceInBoldItalic.Rows(4)`. At this point we could have the entire row's content displayed in a bold, italic font by using the following code:

```
ExpensiveProductsPriceInBoldItalic.Rows(4).Font.Bold = True
ExpensiveProductsPriceInBoldItalic.Rows(4).Font.Italic = True
```

However, this will make *both* the label (Price) and the value bold and italic. If we want to make just the value bold and italic we need to apply this formatting to the second cell in the row, which can be accomplished using the following:

```
ExpensiveProductsPriceInBoldItalic.Rows(4).Cells(1).Font.Bold = True
ExpensiveProductsPriceInBoldItalic.Rows(4).Cells(1).Font.Italic = True
```

Since our tutorials thus far have used stylesheets to maintain a clean separation between the rendered markup and style-related information, rather than setting the specific style properties as shown above let's instead use a CSS class. Open the `Styles.css` stylesheet and add a new CSS class named `ExpensivePriceEmphasis` with the following definition:

```
.ExpensivePriceEmphasis
{
    font-weight: bold;
    font-style: italic;
}
```

Then, in the `DataBound` event handler, set the cell's `CssClass` property to `ExpensivePriceEmphasis`. The following code shows the `DataBound` event handler in its entirety:

```
Protected Sub ExpensiveProductsPriceInBoldItalic_DataBound _
    (sender As Object, e As System.EventArgs) _
    Handles ExpensiveProductsPriceInBoldItalic.DataBound

    Dim product As Northwind.ProductsRow = _
        CType(CType(ExpensiveProductsPriceInBoldItalic.DataItem, _
            System.Data.DataRowView).Row, Northwind.ProductsRow)
    If Not product.IsUnitPriceNull() AndAlso product.UnitPrice > 75 Then
       ExpensiveProductsPriceInBoldItalic.Rows(4).Cells(1).CssClass = _
            "ExpensivePriceEmphasis"
    End If
End Sub
```

When viewing Chai, which costs less than $75.00, the price is displayed in a normal font (see Figure 4). However, when viewing Mishi Kobe Niku, which has a price of $97.00, the price is displayed in a bold, italic font (see Figure 5).

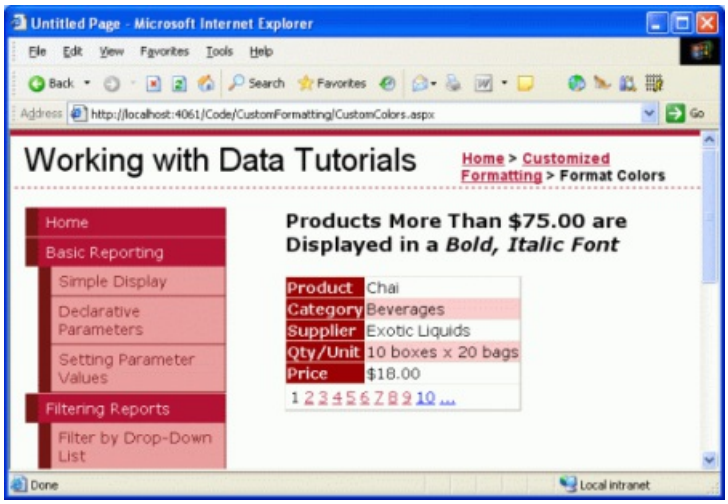


Figure 4: Prices Less than $75.00 are Displayed in a Normal Font (Click to view full-size image)

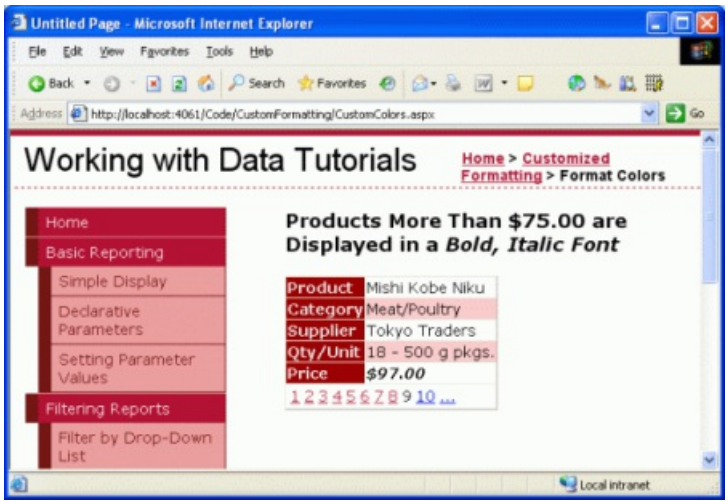


Figure 5: Expensive Products' Prices are Displayed in a Bold, Italic Font (Click to view full-size image)

Using the FormView Control's `DataBound` Event Handler

The steps for determining the underlying data bound to a FormView are identical to those for a DetailsView create a `DataBound` event handler, cast the `DataItem` property to the appropriate object type bound to the control, and determine how to proceed. The FormView and DetailsView differ, however, in how their user interface's appearance is updated.

The FormView does not contain any BoundFields and therefore lacks the `Rows` collection. Instead, a FormView is composed of templates, which can contain a mix of static HTML, Web controls, and databinding syntax. Adjusting the style of a FormView typically involves adjusting the style of one or more of the Web controls within the FormView's templates.

To illustrate this, let's use a FormView to list products like in the previous example, but this time let's display just the product name and units in stock with the units in stock displayed in a red font if it is less than or equal to 10.

Step 4: Displaying the Product Information in a FormView

Add a FormView to the `CustomColors.aspx` page beneath the DetailsView and set its `ID` property to `LowStockedProductsInRed`. Bind the FormView to the ObjectDataSource control created from the previous step. This will create an `ItemTemplate`, `EditItemTemplate`, and `InsertItemTemplate` for the FormView. Remove the `EditItemTemplate` and `InsertItemTemplate` and simplify the `ItemTemplate` to include just the `ProductName` and `UnitsInStock` values, each in their own appropriately-named Label controls. As with the DetailsView from the earlier example, also check the Enable Paging checkbox in the FormView's smart tag.

After these edits your FormView's markup should look similar to the following:

```
<asp:FormView ID="LowStockedProductsInRed" runat="server"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1"
    AllowPaging="True" EnableViewState="False">
    <ItemTemplate>
        <b>Product:</b>
        <asp:Label ID="ProductNameLabel" runat="server"
         Text='<%# Bind("ProductName") %>'>
        </asp:Label><br />
        <b>Units In Stock:</b>
        <asp:Label ID="UnitsInStockLabel" runat="server"
          Text='<%# Bind("UnitsInStock") %>'>
        </asp:Label>
    </ItemTemplate>
</asp:FormView>
```

Note that the `ItemTemplate` contains:

- **Static HTML** the text "Product:" and "Units In Stock:" along with the `<br />` and `<b>` elements.
- **Web controls** the two Label controls, `ProductNameLabel` and `UnitsInStockLabel`.
- **Databinding syntax** the `<%# Bind("ProductName") %>` and `<%# Bind("UnitsInStock") %>` syntax, which assigns the values from these fields to the Label controls' `Text` properties.

Step 5: Programmatically Determining the Value of the Data in the DataBound Event Handler

With the FormView's markup complete, the next step is to programmatically determine if the `UnitsInStock` value is less than or equal to 10. This is accomplished in the exact same manner with the FormView as it was with the DetailsView. Start by creating an event handler for the FormView's `DataBound` event.

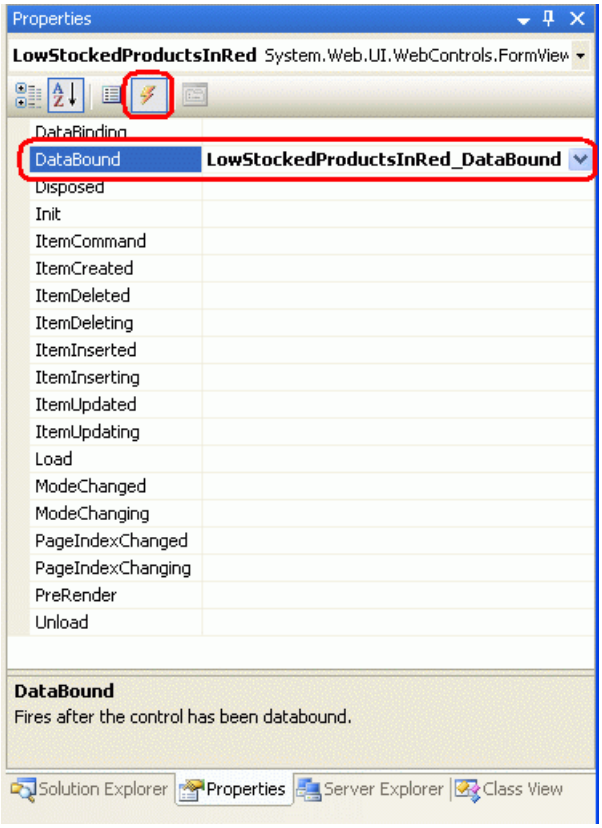


Figure 6: Create the `DataBound` Event Handler

In the event handler cast the FormView's `DataItem` property to a `ProductsRow` instance and determine whether

the `UnitsInPrice` value is such that we need to display it in a red font.

```
Protected Sub LowStockedProductsInRed_DataBound _
    (sender As Object, e As System.EventArgs) _
    Handles LowStockedProductsInRed.DataBound

    Dim product As Northwind.ProductsRow = _
        CType(CType(LowStockedProductsInRed.DataItem, System.Data.DataRowView).Row, _
            Northwind.ProductsRow)
    If Not product.IsUnitsInStockNull() AndAlso product.UnitsInStock <= 10 Then
        Dim unitsInStock As Label = _
            CType(LowStockedProductsInRed.FindControl("UnitsInStockLabel"), Label)

        If unitsInStock IsNot Nothing Then
        End If
    End If
End Sub
```

Step 6: Formatting the UnitsInStockLabel Label Control in the FormView's ItemTemplate

The final step is to format the displayed `UnitsInStock` value in a red font if the value is 10 or less. To accomplish this we need to programmatically access the `UnitsInStockLabel` control in the `ItemTemplate` and set its style properties so that its text is displayed in red. To access a Web control in a template, use the `FindControl("controlID")` method like this:

```
Dim someName As WebControlType = _
    CType(FormViewID.FindControl("controlID"), WebControlType)
```

For our example we want to access a Label control whose `ID` value is `UnitsInStockLabel`, so we'd use:

```
Dim unitsInStock As Label = _
    CType(LowStockedProductsInRed.FindControl("UnitsInStockLabel"), Label)
```

Once we have a programmatic reference to the Web control, we can modify its style-related properties as needed. As with the earlier example, I've created a CSS class in `Styles.css` named `LowUnitsInStockEmphasis`. To apply this style to the Label Web control, set its `CssClass` property accordingly.

```
Protected Sub LowStockedProductsInRed_DataBound _
    (sender As Object, e As System.EventArgs) _
    Handles LowStockedProductsInRed.DataBound

    Dim product As Northwind.ProductsRow = _
        CType(CType(LowStockedProductsInRed.DataItem, System.Data.DataRowView).Row, _
            Northwind.ProductsRow)
    If Not product.IsUnitsInStockNull() AndAlso product.UnitsInStock <= 10 Then
        Dim unitsInStock As Label = _
            CType(LowStockedProductsInRed.FindControl("UnitsInStockLabel"), Label)

        If unitsInStock IsNot Nothing Then
            unitsInStock.CssClass = "LowUnitsInStockEmphasis"
        End If
    End If
End Sub
```

> **NOTE**
> The syntax for formatting a template programmatically accessing the Web control using `FindControl("controlID")` and then setting its style-related properties can also be used when using TemplateFields in the DetailsView or GridView controls. We'll examine TemplateFields in our next tutorial.

Figures 7 shows the FormView when viewing a product whose `UnitsInStock` value is greater than 10, while the product in Figure 8 has its value less than 10.

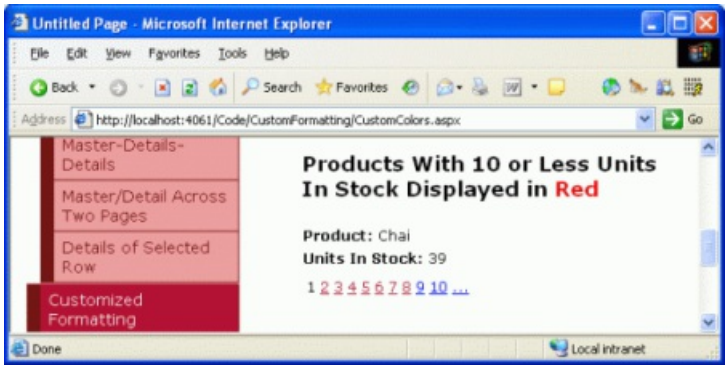


Figure 7: For Products With a Sufficiently Large Units In Stock, No Custom Formatting is Applied (Click to view full-size image)

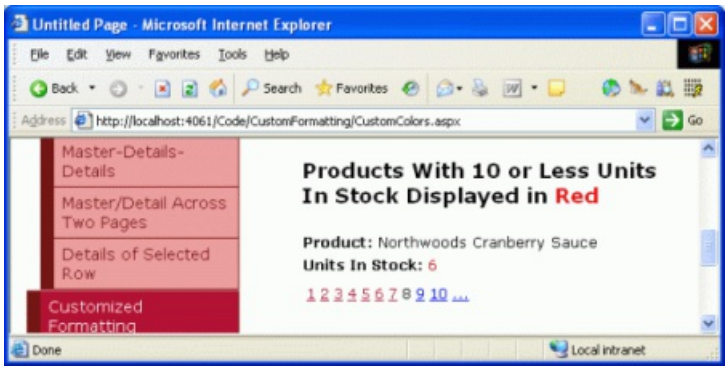


Figure 8: The Units in Stock Number is Shown in Red for Those Products With Values of 10 or Less (Click to view full-size image)

Formatting with the GridView's `RowDataBound` Event

Earlier we examined the sequence of steps the DetailsView and FormView controls progress through during databinding. Let's look over these steps once again as a refresher.

1. The data Web control's `DataBinding` event fires.
2. The data is bound to the data Web control.
3. The data Web control's `DataBound` event fires.

These three simple steps are sufficient for the DetailsView and FormView because they display only a single record. For the GridView, which displays *all* records bound to it (not just the first), step 2 is a bit more involved.

In step 2 the GridView enumerates the data source and, for each record, creates a `GridViewRow` instance and binds the current record to it. For each `GridViewRow` added to the GridView, two events are raised:

- `RowCreated` fires after the `GridViewRow` has been created
- `RowDataBound` fires after the current record has been bound to the `GridViewRow`.

For the GridView, then, data binding is more accurately described by the following sequence of steps:

1. The GridView's `DataBinding` event fires.
2. The data is bound to the GridView.

   For each record in the data source

   a. Create a `GridViewRow` object
   b. Fire the `RowCreated` event
   c. Bind the record to the `GridViewRow`
   d. Fire the `RowDataBound` event
   e. Add the `GridViewRow` to the `Rows` collection
3. The GridView's `DataBound` event fires.

To customize the format of the GridView's individual records, then, we need to create an event handler for the `RowDataBound` event. To illustrate this, let's add a GridView to the `CustomColors.aspx` page that lists the name, category, and price for each product, highlighting those products whose price is less than $10.00 with a yellow background color.

Step 7: Displaying Product Information in a GridView

Add a GridView beneath the FormView from the previous example and set its `ID` property to `HighlightCheapProducts`. Since we already have an ObjectDataSource that returns all products on the page, bind the GridView to that. Finally, edit the GridView's BoundFields to include just the products' names, categories, and prices. After these edits the GridView's markup should look like:

```
<asp:GridView ID="HighlightCheapProducts" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1"
    EnableViewState="False" runat="server">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
          SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
          ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
          HeaderText="Price"
            HtmlEncode="False" SortExpression="UnitPrice" />
    </Columns>
</asp:GridView>
```

Figure 9 shows our progress to this point when viewed through a browser.

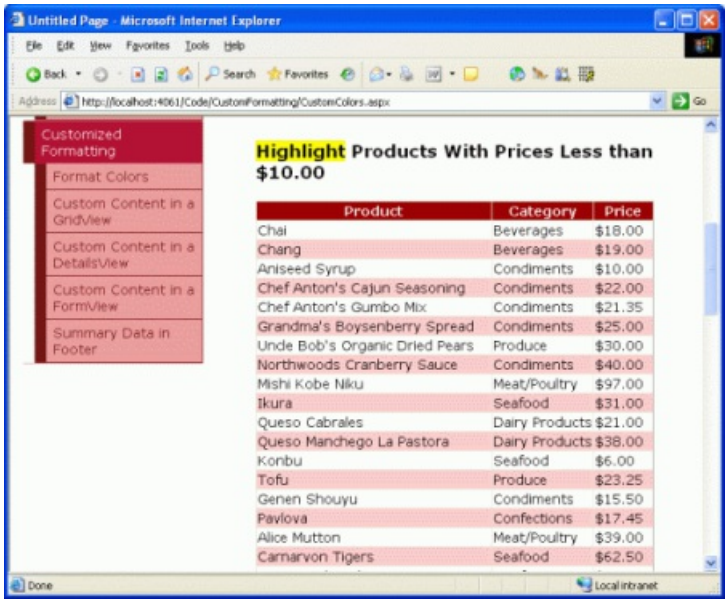


Figure 9: The GridView Lists the Name, Category, and Price For Each Product (Click to view full-size image)

Step 8: Programmatically Determining the Value of the Data in the RowDataBound Event Handler

When the `ProductsDataTable` is bound to the GridView its `ProductsRow` instances are enumerated and for each `ProductsRow` a `GridViewRow` is created. The `GridViewRow`'s `DataItem` property is assigned to the particular `ProductRow`, after which the GridView's `RowDataBound` event handler is raised. To determine the `UnitPrice` value for each product bound to the GridView, then, we need to create an event handler for the GridView's `RowDataBound` event. In this event handler we can inspect the `UnitPrice` value for the current `GridViewRow` and make a formatting decision for that row.

This event handler can be created using the same series of steps as with the FormView and DetailsView.

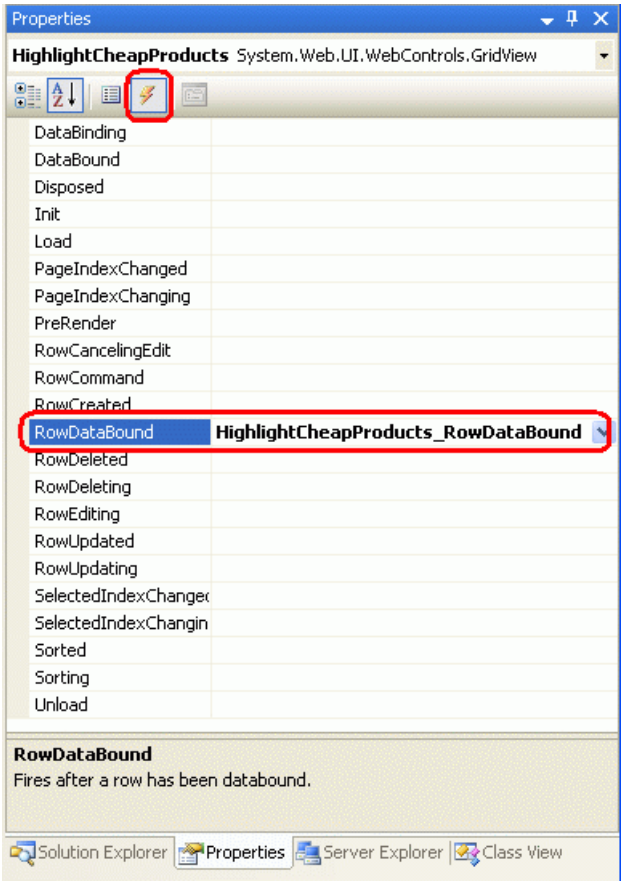


Figure 10: Create an Event Handler for the GridView's `RowDataBound` Event

Creating the event handler in this manner will cause the following code to be automatically added to the ASP.NET page's code portion:

```
Protected Sub HighlightCheapProducts_RowDataBound _
    (sender As Object, e As System.Web.UI.WebControls.GridViewRowEventArgs) _
    Handles HighlightCheapProducts.RowDataBound

End Sub
```

When the `RowDataBound` event fires, the event handler is passed as its second parameter an object of type `GridViewRowEventArgs`, which has a property named `Row`. This property returns a reference to the `GridViewRow` that was just data bound. To access the `ProductsRow` instance bound to the `GridViewRow` we use the `DataItem` property like so:

```
Protected Sub HighlightCheapProducts_RowDataBound _
    (sender As Object, e As System.Web.UI.WebControls.GridViewRowEventArgs) _
    Handles HighlightCheapProducts.RowDataBound

    Dim product As Northwind.ProductsRow = _
        CType(CType(e.Row.DataItem, System.Data.DataRowView).Row, Northwind.ProductsRow)
    If Not product.IsUnitPriceNull() AndAlso product.UnitPrice < 10 Then
    End If
End Sub
```

When working with the `RowDataBound` event handler it is important to keep in mind that the GridView is composed of different types of rows and that this event is fired for *all* row types. A `GridViewRow`'s type can be determined by its `RowType` property, and can have one of the possible values:

- `DataRow` a row that is bound to a record from the GridView's `DataSource`
- `EmptyDataRow` the row displayed if the GridView's `DataSource` is empty
- `Footer` the footer row; shown if the GridView's `ShowFooter` property is set to `True`
- `Header` the header row; shown if the GridView's ShowHeader property is set to `True` (the default)
- `Pager` for GridView's that implement paging, the row that displays the paging interface
- `Separator` not used for the GridView, but used by the `RowType` properties for the DataList and Repeater controls, two data Web controls we'll discuss in future tutorials

Since the `EmptyDataRow`, `Header`, `Footer`, and `Pager` rows aren't associated with a `DataSource` record, they will always have a value of `Nothing` for their `DataItem` property. For this reason, before attempting to work with the current `GridViewRow`'s `DataItem` property, we first must make sure that we're dealing with a `DataRow`. This can be accomplished by checking the `GridViewRow`'s `RowType` property like so:

```
Protected Sub HighlightCheapProducts_RowDataBound _
    (sender As Object, e As System.Web.UI.WebControls.GridViewRowEventArgs) _
    Handles HighlightCheapProducts.RowDataBound

    If e.Row.RowType = DataControlRowType.DataRow Then
        Dim product As Northwind.ProductsRow = _
            CType(CType(e.Row.DataItem, System.Data.DataRowView).Row, Northwind.ProductsRow)
        If Not product.IsUnitPriceNull() AndAlso product.UnitPrice < 10 Then
        End If
    End If
End Sub
```

Step 9: Highlighting the Row Yellow When the UnitPrice Value is Less than $10.00

The last step is to programmatically highlight the entire `GridViewRow` if the `UnitPrice` value for that row is less than $10.00. The syntax for accessing a GridView's rows or cells is the same as with the DetailsView `GridViewID.Rows(index)` to access the entire row, `GridViewID.Rows(index).Cells(index)` to access a particular cell. However, when the `RowDataBound` event handler fires the data bound `GridViewRow` has yet to be added to the GridView's `Rows` collection. Therefore you cannot access the current `GridViewRow` instance from the `RowDataBound` event handler using the Rows collection.

Instead of `GridViewID.Rows(index)`, we can reference the current `GridViewRow` instance in the `RowDataBound` event handler using `e.Row`. That is, in order to highlight the current `GridViewRow` instance from the `RowDataBound` event handler we would use:

```
e.Row.BackColor = System.Drawing.Color.Yellow
```

Rather than set the `GridViewRow`'s `BackColor` property directly, let's stick with using CSS classes. I've created a CSS class named `AffordablePriceEmphasis` that sets the background color to yellow. The completed `RowDataBound` event handler follows:

```
Protected Sub HighlightCheapProducts_RowDataBound _
    (sender As Object, e As System.Web.UI.WebControls.GridViewRowEventArgs) _
    Handles HighlightCheapProducts.RowDataBound

    If e.Row.RowType = DataControlRowType.DataRow Then
        Dim product As Northwind.ProductsRow = _
            CType(CType(e.Row.DataItem, System.Data.DataRowView).Row, Northwind.ProductsRow)
        If Not product.IsUnitPriceNull() AndAlso product.UnitPrice < 10 Then
            e.Row.CssClass = "AffordablePriceEmphasis"
        End If
    End If
End Sub
```

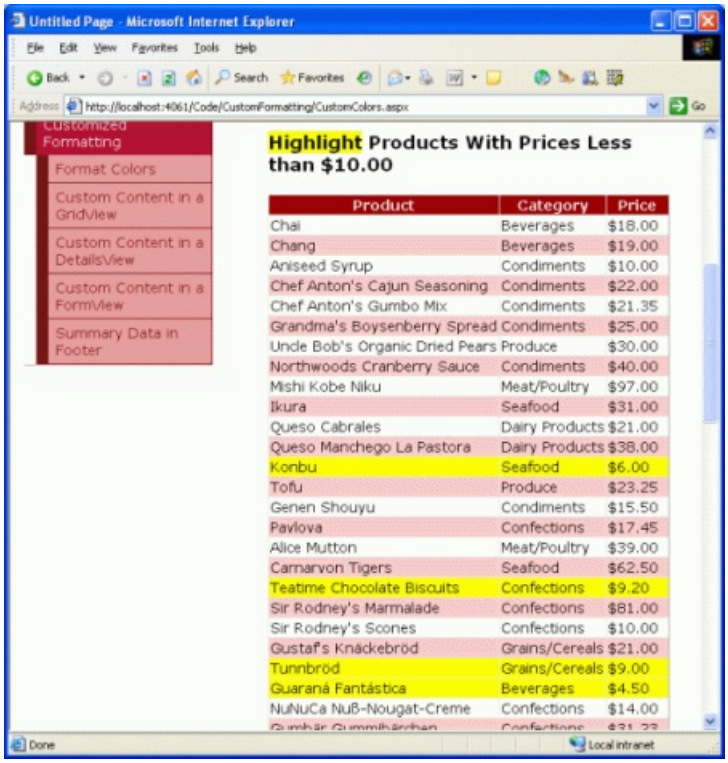


Figure 11: The Most Affordable Products are Highlighted Yellow (Click to view full-size image)

Summary

In this tutorial we saw how to format the GridView, DetailsView, and FormView based on the data bound to the control. To accomplish this we created an event handler for the `DataBound` or `RowDataBound` events, where the underlying data was examined along with a formatting change, if needed. To access the data bound to a DetailsView or FormView, we use the `DataItem` property in the `DataBound` event handler; for a GridView, each `GridViewRow` instance's `DataItem` property contains the data bound to that row, which is available in the `RowDataBound` event handler.

The syntax for programmatically adjusting the data Web control's formatting depends upon the Web control and how the data to be formatted is displayed. For DetailsView and GridView controls, the rows and cells can be accessed by an ordinal index. For the FormView, which uses templates, the `FindControl("controlID")` method is commonly used to locate a Web control from within the template.

In the next tutorial we'll look at how to use templates with the GridView and DetailsView. Additionally, we'll see another technique for customizing the formatting based on the underlying data.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were E.R. Gilmore, Dennis Patterson, and Dan Jagers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Using TemplateFields in the GridView Control (VB)

2/20/2020 • 18 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> To provide flexibility, the GridView offers the TemplateField, which renders using a template. A template can include a mix of static HTML, Web controls, and databinding syntax. In this tutorial we'll examine how to use the TemplateField to achieve a greater degree of customization with the GridView control.

Introduction

The GridView is composed of a set of fields that indicate what properties from the `DataSource` are to be included in the rendered output along with how the data will be displayed. The simplest field type is the BoundField, which displays a data value as text. Other field types display the data using alternate HTML elements. The CheckBoxField, for example, renders as a checkbox whose checked state depends on the value of a specified data field; the ImageField renders an image whose image source is based upon a specified data field. Hyperlinks and buttons whose state depends on an underlying data field value can be rendered using the HyperLinkField and ButtonField field types.

While the CheckBoxField, ImageField, HyperLinkField, and ButtonField field types allow for an alternate view of the data, they still are fairly limited with respect to formatting. A CheckBoxField can only display a single checkbox, whereas an ImageField can only display a single image. What if a particular field needs to display some text, a checkbox, *and* an image, all based upon different data field values? Or what if we wanted to display the data using a Web control other than the CheckBox, Image, HyperLink, or Button? Furthermore, the BoundField limits its display to a single data field. What if we wanted to show two or more data field values in a single GridView column?

To accommodate this level of flexibility the GridView offers the TemplateField, which renders using a *template*. A template can include a mix of static HTML, Web controls, and databinding syntax. Furthermore, the TemplateField has a variety of templates that can be used to customize the rendering for different situations. For example, the `ItemTemplate` is used by default to render the cell for each row, but the `EditItemTemplate` template can be used to customize the interface when editing data.

In this tutorial we'll examine how to use the TemplateField to achieve a greater degree of customization with the GridView control. In the preceding tutorial we saw how to customize the formatting based on the underlying data using the `DataBound` and `RowDataBound` event handlers. Another way to customize the formatting based on the underlying data is by calling formatting methods from within a template. We'll look at this technique in this tutorial as well.

For this tutorial we will use TemplateFields to customize the appearance of a list of employees. Specifically, we'll list all of the employees, but will display the employee's first and last names in one column, their hire date in a Calendar control, and a status column that indicates how many days they've been employed at the company.

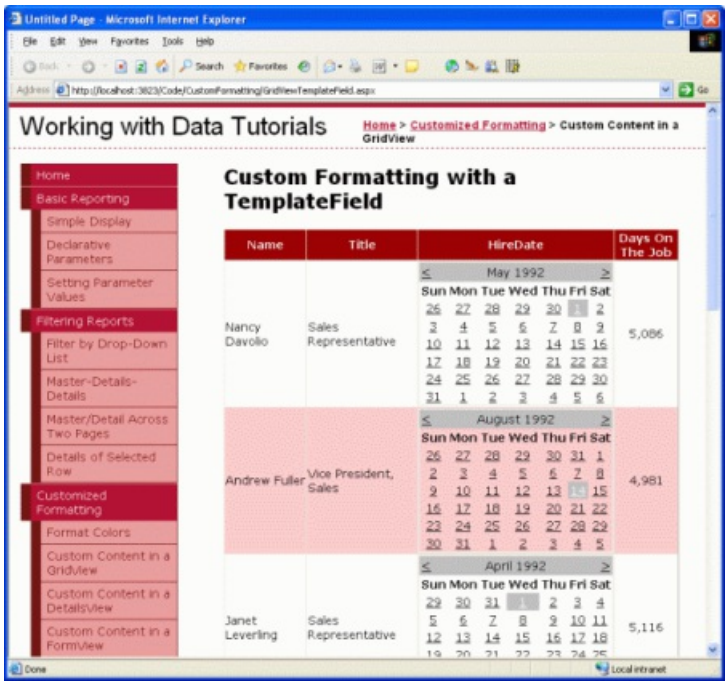


Figure 1: Three TemplateFields are Used to Customize the Display (Click to view full-size image)

Step 1: Binding the Data to the GridView

In reporting scenarios where you need to use TemplateFields to customize the appearance, I find it easiest to start by creating a GridView control that contains just BoundFields first and then to add new TemplateFields or convert the existing BoundFields to TemplateFields as needed. Therefore, let's start this tutorial by adding a GridView to the page through the Designer and binding it to an ObjectDataSource that returns the list of employees. These steps will create a GridView with BoundFields for each of the employee fields.

Open the `GridViewTemplateField.aspx` page and drag a GridView from the Toolbox onto the Designer. From the GridView's smart tag choose to add a new ObjectDataSource control that invokes the `EmployeesBLL` class's `GetEmployees()` method.

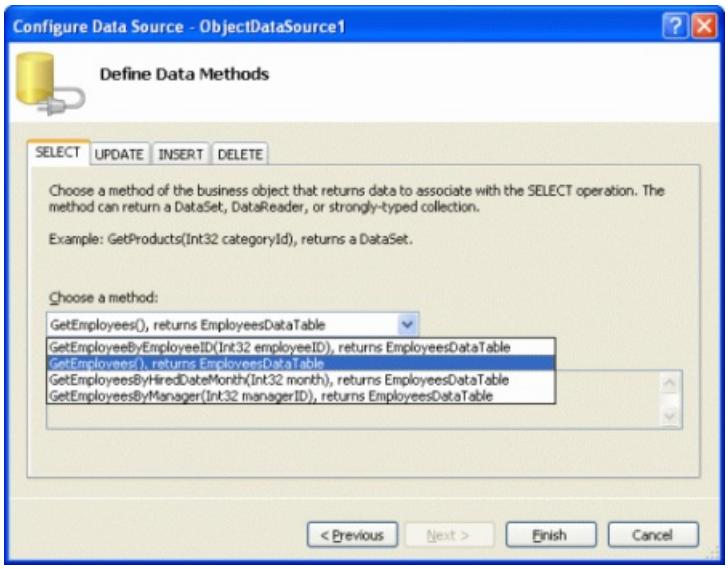


Figure 2: Add a New ObjectDataSource Control that Invokes the `GetEmployees()` Method (Click to view full-size image)

Binding the GridView in this manner will automatically add a BoundField for each of the employee properties: `EmployeeID`, `LastName`, `FirstName`, `Title`, `HireDate`, `ReportsTo`, and `Country`. For this report let's not bother with displaying the `EmployeeID`, `ReportsTo`, or `Country` properties. To remove these BoundFields you can:

- Use the Fields dialog box click on the Edit Columns link from the GridView's smart tag to bring up this dialog box. Next, select the BoundFields from the lower left list and click the red X button to remove the BoundField.
- Edit the GridView's declarative syntax by hand from the Source view, delete the `<asp:BoundField>` element for the BoundField you wish to remove.

After you have removed the `EmployeeID`, `ReportsTo`, and `Country` BoundFields, your GridView's markup should look like:

```
<asp:GridView ID="GridView1" runat="server"
    AutoGenerateColumns="False" DataKeyNames="EmployeeID"
    DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:BoundField DataField="LastName" HeaderText="LastName"
          SortExpression="LastName" />
        <asp:BoundField DataField="FirstName" HeaderText="FirstName"
          SortExpression="FirstName" />
        <asp:BoundField DataField="Title" HeaderText="Title"
          SortExpression="Title" />
        <asp:BoundField DataField="HireDate" HeaderText="HireDate"
          SortExpression="HireDate" />
    </Columns>
</asp:GridView>
```

Take a moment to view our progress in a browser. At this point you should see a table with a record for each employee and four columns: one for the employee's last name, one for their first name, one for their title, and one for their hire date.

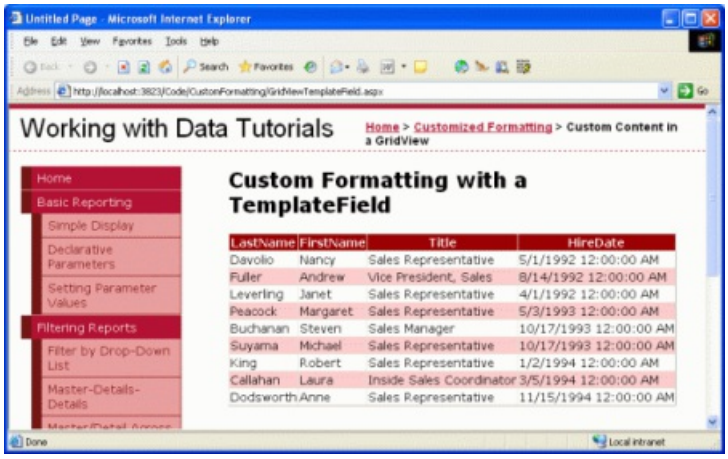


Figure 3: The `LastName`, `FirstName`, `Title`, and `HireDate` Fields are Displayed for Each Employee (Click to view full-size image)

Step 2: Displaying the First and Last Names in a Single Column

Currently, each employee's first and last names are displayed in a separate column. It might be nice to combine them into a single column instead. To accomplish this we need to use a TemplateField. We can either add a new TemplateField, add to it the needed markup and databinding syntax, and then delete the `FirstName` and `LastName` BoundFields, or we can convert the `FirstName` BoundField into a TemplateField, edit the TemplateField to include the `LastName` value, and then remove the `LastName` BoundField.

Both approaches net the same result, but personally I like converting BoundFields to TemplateFields when possible because the conversion automatically adds an `ItemTemplate` and `EditItemTemplate` with Web controls and databinding syntax to mimic the appearance and functionality of the BoundField. The benefit is that we'll need to do less work with the TemplateField as the conversion process will have performed some of the work for us.

To convert an existing BoundField into a TemplateField, click on the Edit Columns link from the GridView's smart tag, bringing up the Fields dialog box. Select the BoundField to convert from the list in the lower left corner and then click the "Convert this field into a TemplateField" link in the bottom right corner.

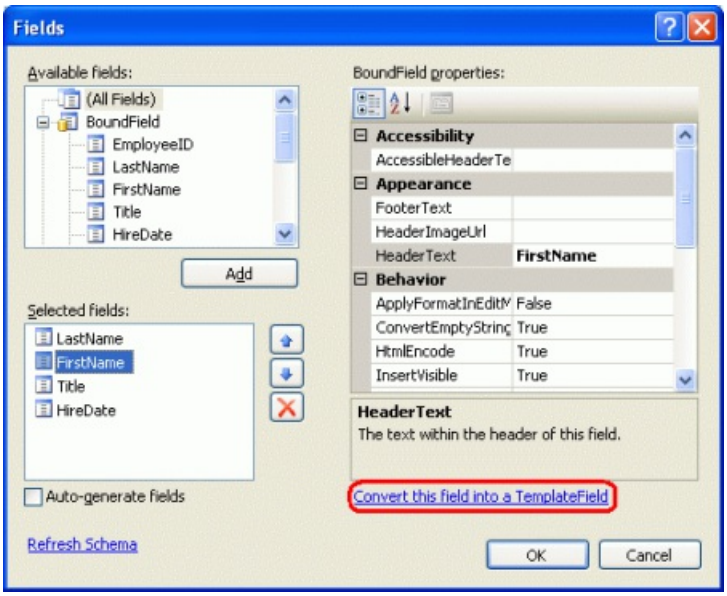


Figure 4: Convert a BoundField Into a TemplateField from the Fields Dialog Box (Click to view full-size image)

Go ahead and convert the `FirstName` BoundField into a TemplateField. After this change there's no perceptive difference in the Designer. This is because converting the BoundField into a TemplateField creates a TemplateField that maintains the look and feel of the BoundField. Despite there being no visual difference at this point in the Designer, this conversion process has replaced the BoundField's declarative syntax - `<asp:BoundField DataField="FirstName" HeaderText="FirstName" SortExpression="FirstName" />` - with the following TemplateField syntax:

```
<asp:TemplateField HeaderText="FirstName" SortExpression="FirstName">
    <EditItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
            Text='<%# Bind("FirstName") %>'></asp:TextBox>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
            Text='<%# Bind("FirstName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

As you can see, the TemplateField consists of two templates an `ItemTemplate` that has a Label whose `Text` property is set to the value of the `FirstName` data field, and an `EditItemTemplate` with a TextBox control whose `Text` property is also set to the `FirstName` data field. The databinding syntax - `<%# Bind("fieldName") %>` - indicates that the data field `fieldName` is bound to the specified Web control property.

To add the `LastName` data field value to this TemplateField we need to add another Label Web control in the `ItemTemplate` and bind its `Text` property to `LastName`. This can be accomplished either by hand or through the Designer. To do it by hand, simply add the appropriate declarative syntax to the `ItemTemplate`:

```
<asp:TemplateField HeaderText="FirstName" SortExpression="FirstName">
    <EditItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
            Text='<%# Bind("FirstName") %>'></asp:TextBox>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
            Text='<%# Bind("FirstName") %>'></asp:Label>
        <asp:Label ID="Label2" runat="server"
            Text='<%# Bind("LastName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

To add it through the Designer, click on the Edit Templates link from the GridView's smart tag. This will display the GridView's template editing interface. In this interface's smart tag is a list of the templates in the GridView. Since we only have one TemplateField at this point, the only templates listed in the drop-down list are those templates for the `FirstName` TemplateField along with the `EmptyDataTemplate` and `PagerTemplate`. The `EmptyDataTemplate` template, if specified, is used to render the GridView's output if there are no results in the data bound to the GridView; the `PagerTemplate`, if specified, is used to render the paging interface for a GridView that supports paging.

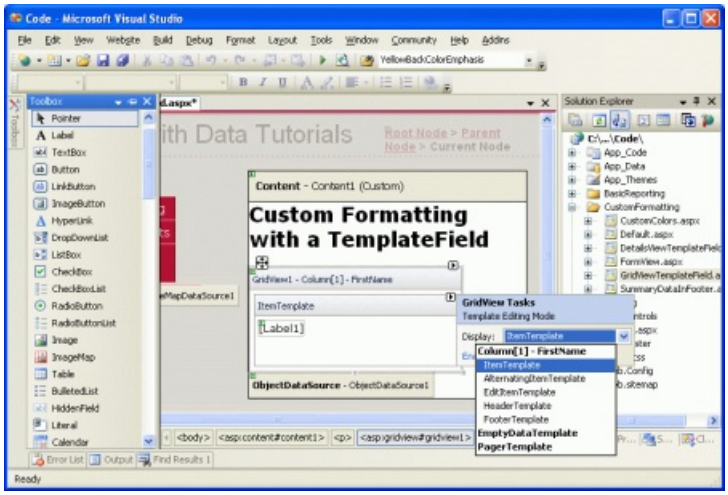


Figure 5: The GridView's Templates Can Be Edited Through the Designer (Click to view full-size image)

To also display the `LastName` in the `FirstName` TemplateField drag the Label control from the Toolbox into the `FirstName` TemplateField's `ItemTemplate` in the GridView's template editing interface.

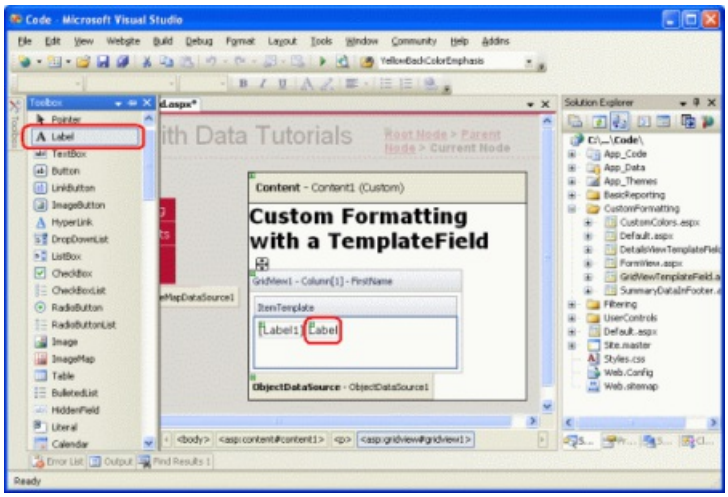


Figure 6: Add a Label Web Control to the `FirstName` TemplateField's ItemTemplate (Click to view full-size image)

At this point the Label Web control added to the TemplateField has its `Text` property set to "Label". We need to

change this so that this property is bound to the value of the `LastName` data field instead. To accomplish this click on the Label control's smart tag and choose the Edit DataBindings option.

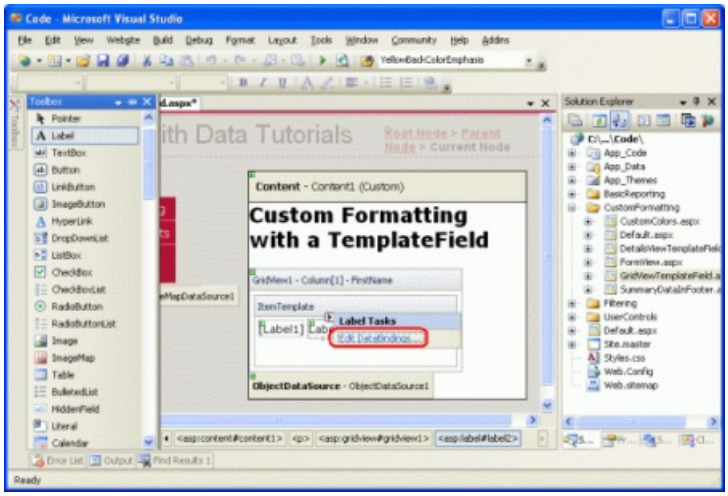


Figure 7: Choose the Edit DataBindings Option from the Label's Smart Tag (Click to view full-size image)

This will bring up the DataBindings dialog box. From here you can select the property to participate in databinding from the list on the left and choose the field to bind the data to from the drop-down list on the right. Choose the `Text` property from the left and the `LastName` field from the right and click OK.

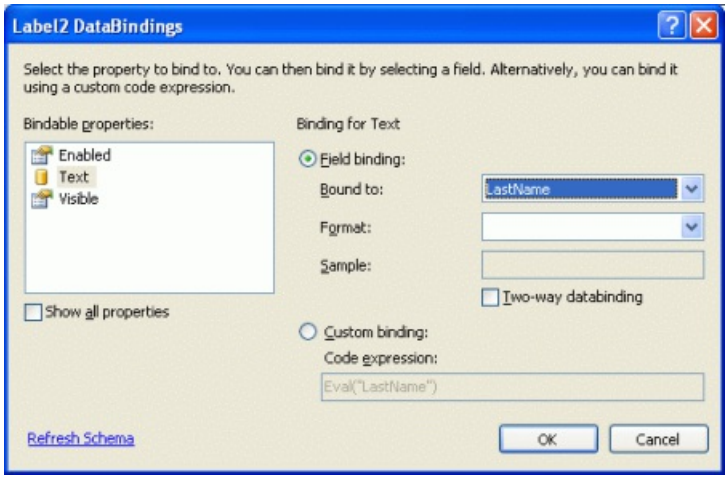


Figure 8: Bind the `Text` Property to the `LastName` Data Field (Click to view full-size image)

> **NOTE**
>
> The DataBindings dialog box allows you to indicate whether to perform two-way databinding. If you leave this unchecked, the databinding syntax `<%# Eval("LastName")%>` will be used instead of `<%# Bind("LastName")%>`. Either approach is fine for this tutorial. Two-way databinding becomes important when inserting and editing data. For simply displaying data, however, either approach will work equally well. We'll discuss two-way databinding in detail in future tutorials.

Take a moment to view this page through a browser. As you can see, the GridView still includes four columns; however, the `FirstName` column now lists *both* the `FirstName` and `LastName` data field values.

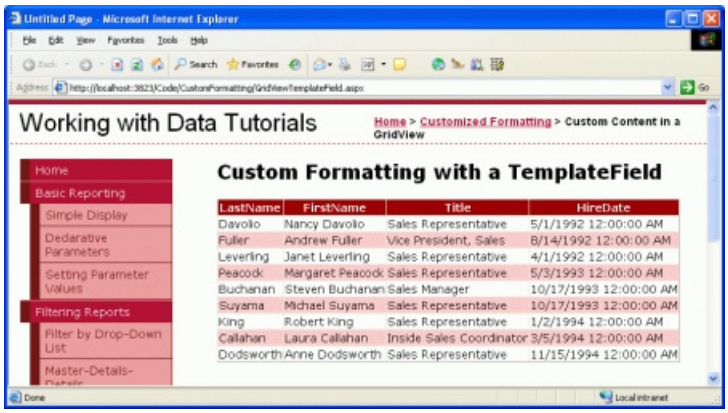


Figure 9: Both the `FirstName` and `LastName` Values are Shown in a Single Column (Click to view full-size image)

To complete this first step, remove the `LastName` BoundField and rename the `FirstName` TemplateField's `HeaderText` property to "Name". After these changes the GridView's declarative markup should look like the following:

```
<asp:GridView ID="GridView1" runat="server"
    AutoGenerateColumns="False" DataKeyNames="EmployeeID"
    DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:TemplateField HeaderText="Name" SortExpression="FirstName">
            <EditItemTemplate>
                <asp:TextBox ID="TextBox1" runat="server"
                    Text='<%# Bind("FirstName") %>'></asp:TextBox>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("FirstName") %>'></asp:Label>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Eval("LastName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="Title" HeaderText="Title"
            SortExpression="" Title" />
        <asp:BoundField DataField="HireDate" HeaderText="
            HireDate" SortExpression="" HireDate" />
    </Columns>
</asp:GridView>
```

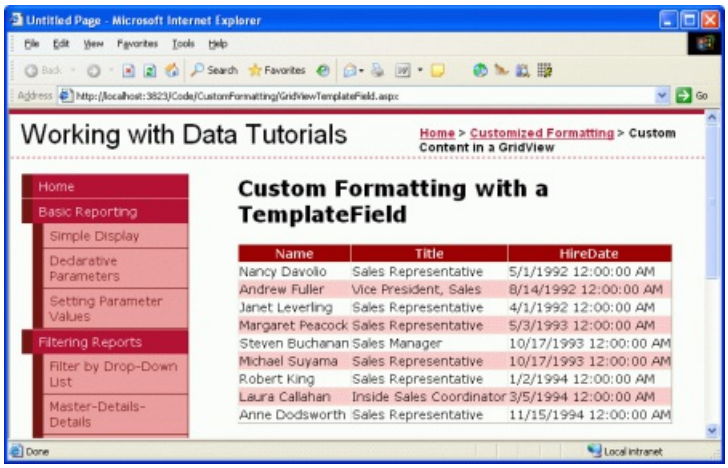


Figure 10: Each Employee's First and Last Names are Displayed in One Column (Click to view full-size image)

Step 3: Using the Calendar Control to Display the `HiredDate` Field

Displaying a data field value as text in a GridView is as simple as using a BoundField. For certain scenarios,

however, the data is best expressed using a particular Web control instead of just text. Such customization of the display of data is possible with TemplateFields. For example, rather than display the employee's hire date as text, we could show a calendar (using the Calendar control) with their hire date highlighted.

To accomplish this, start by converting the `HiredDate` BoundField into a TemplateField. Simply go to the GridView's smart tag and click the Edit Columns link, bringing up the Fields dialog box. Select the `HiredDate` BoundField and click "Convert this field into a TemplateField."

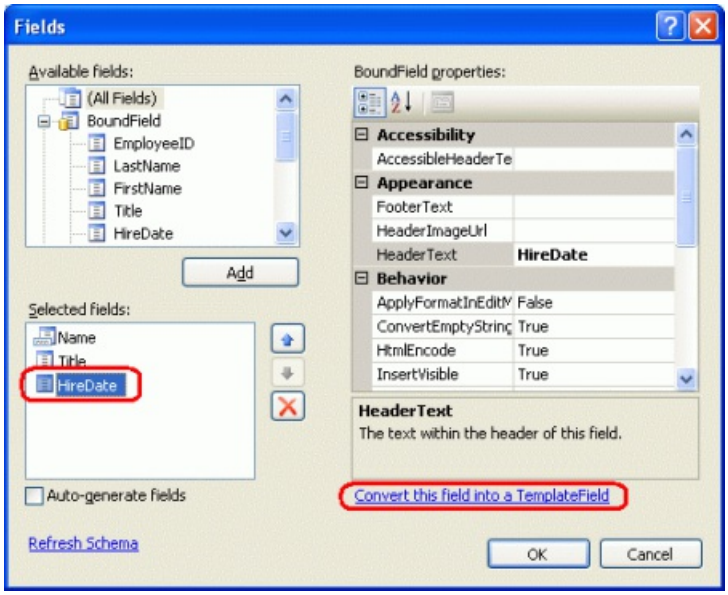


Figure 11: Convert the `HiredDate` BoundField Into a TemplateField (Click to view full-size image)

As we saw in Step 2, this will replace the BoundField with a TemplateField that contains an `ItemTemplate` and `EditItemTemplate` with a Label and TextBox whose `Text` properties are bound to the `HiredDate` value using the databinding syntax `<%# Bind("HiredDate")%>`.

To replace the text with a Calendar control, edit the template by removing the Label and adding a Calendar control. From the Designer, select Edit Templates from the GridView's smart tag and choose the `HireDate` TemplateField's `ItemTemplate` from the drop-down list. Next, delete the Label control and drag a Calendar control from the Toolbox into the template editing interface.

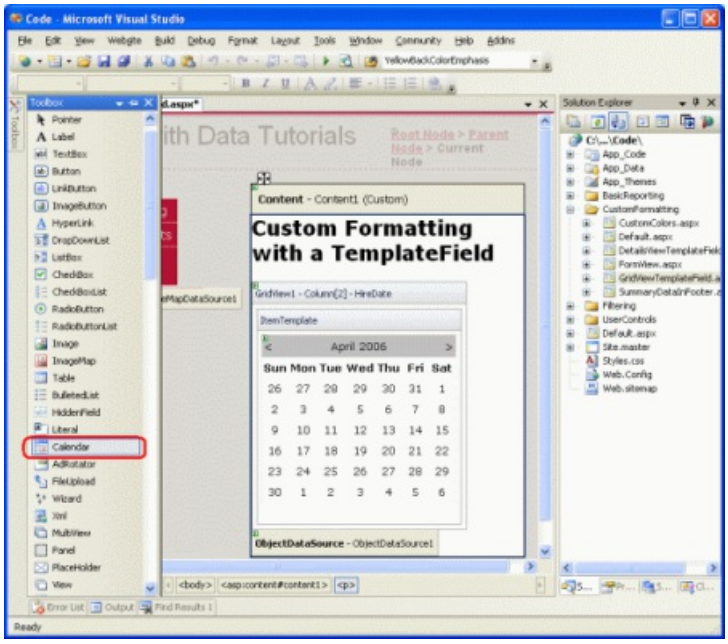


Figure 12: Add a Calendar Control to the `HireDate` TemplateField's `ItemTemplate` (Click to view full-size image)

At this point each row in the GridView will contain a Calendar control in its `HiredDate` TemplateField. However,

the employee's actual `HiredDate` value is not set anywhere in the Calendar control, causing each Calendar control to default to showing the current month and date. To remedy this, we need to assign each employee's `HiredDate` to the Calendar control's SelectedDate and VisibleDate properties.

From the Calendar control's smart tag, choose Edit DataBindings. Next, bind both `SelectedDate` and `VisibleDate` properties to the `HiredDate` data field.

Figure 13: Bind the `SelectedDate` and `VisibleDate` Properties to the `HiredDate` Data Field (Click to view full-size image)

> **NOTE**
>
> The Calendar control's selected date need not necessarily be visible. For example, a Calendar may have August 1st, 1999 as the selected date, but be showing the current month and year. The selected date and visible date are specified by the Calendar control's `SelectedDate` and `VisibleDate` properties. Since we want to both select the employee's `HiredDate` and make sure that it's shown we need to bind both of these properties to the `HireDate` data field.

When viewing the page in a browser, the calendar now shows the month of the employee's hired date and selects that particular date.

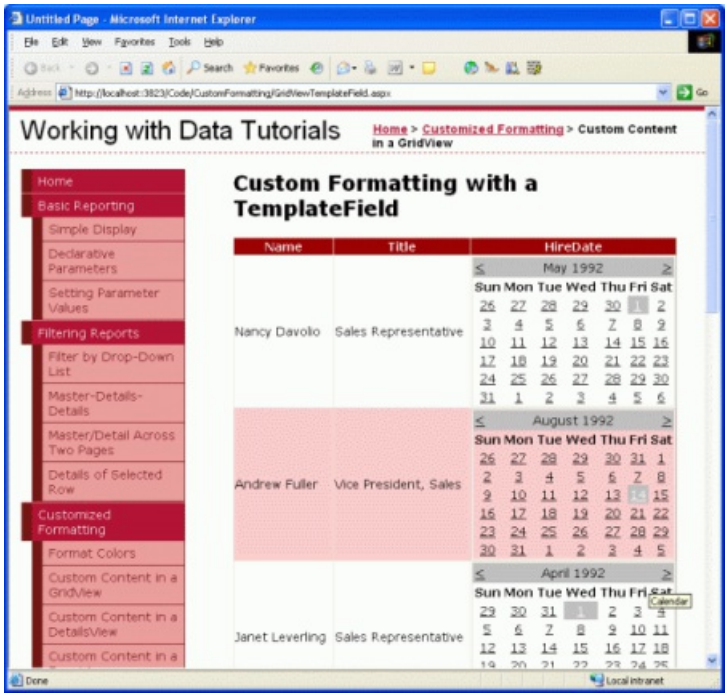


Figure 14: The Employee's `HiredDate` is Shown in the Calendar Control (Click to view full-size image)

NOTE

Contrary to all the examples we've seen thus far, for this tutorial we did *not* set `EnableViewState` property to `False` for this GridView. The reason for this decision is because clicking the dates of the Calendar control causes a postback, setting the Calendar's selected date to the date just clicked. If the GridView's view state is disabled, however, on each postback the GridView's data is rebound to its underlying data source, which causes the Calendar's selected date to be set *back* to the employee's `HireDate`, overwriting the date chosen by the user.

For this tutorial this is a moot discussion since the user is not able to update the employee's `HireDate`. It would probably be best to configure the Calendar control so that its dates are not selectable. Regardless, this tutorial shows that in some circumstances view state must be enabled in order to provide certain functionality.

Step 4: Showing the Number of Days the Employee Has Worked for the Company

So far we have seen two applications of TemplateFields:

- Combining two or more data field values into one column, and
- Expressing a data field value using a Web control instead of text

A third use of TemplateFields is in displaying metadata about the GridView's underlying data. In addition to showing the employees' hire dates, for example, we might also want to have a column that displays how many total days they've been on the job.

Yet another use of TemplateFields arises in scenarios when the underlying data needs to be displayed differently in the web page report than in the format it's stored in the database. Imagine that the `Employees` table had a `Gender` field that stored the character `M` or `F` to indicate the sex of the employee. When displaying this information in a web page we might want to show the gender as either "Male" or "Female", as opposed to just "M" or "F".

Both of these scenarios can be handled by creating a *formatting method* in the ASP.NET page's code-behind class (or in a separate class library, implemented as a `Shared` method) that is invoked from the template. Such a formatting method is invoked from the template using the same databinding syntax seen earlier. The formatting method can take in any number of parameters, but must return a string. This returned string is the HTML that is injected into the template.

To illustrate this concept, let's augment our tutorial to show a column that lists the total number of days an employee has been on the job. This formatting method will take in a `Northwind.EmployeesRow` object and return the number of days the employee has been employed as a string. This method can be added to the ASP.NET page's code-behind class, but *must* be marked as `Protected` or `Public` in order to be accessible from the template.

```
Protected Function DisplayDaysOnJob(employee As Northwind.EmployeesRow) As String
    ' Make sure HiredDate is not NULL... if so, return "Unknown"
    If employee.IsHireDateNull() Then
        Return "Unknown"
    Else
        ' Returns the number of days between the current
        ' date/time and HireDate
        Dim ts As TimeSpan = DateTime.Now.Subtract(employee.HireDate)
        Return ts.Days.ToString("#,##0")
    End If
End Function
```

Since the `HiredDate` field can contain `NULL` database values we must first ensure that the value is not `NULL`

before proceeding with the calculation. If the `HiredDate` value is `NULL`, we simply return the string "Unknown"; if it is not `NULL`, we compute the difference between the current time and the `HiredDate` value and return the number of days.

To utilize this method we need to invoke it from a TemplateField in the GridView using the databinding syntax. Start by adding a new TemplateField to the GridView by clicking on the Edit Columns link in the GridView's smart tag and adding a new TemplateField.

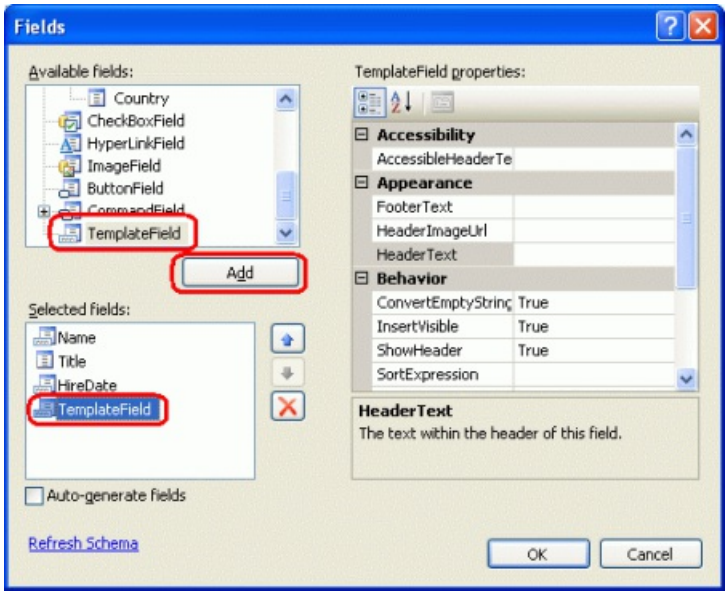


Figure 15: Add a New TemplateField to the GridView (Click to view full-size image)

Set this new TemplateField's `HeaderText` property to "Days on the Job" and its `ItemStyle`'s `HorizontalAlign` property to `Center`. To call the `DisplayDaysOnJob` method from the template, add an `ItemTemplate` and use the following databinding syntax:

```
<%# DisplayDaysOnJob(CType(CType(Container.DataItem, DataRowView).Row,
    Northwind.EmployeesRow)) %>
```

`Container.DataItem` returns a `DataRowView` object corresponding to the `DataSource` record bound to the `GridViewRow`. Its `Row` property returns the strongly-typed `Northwind.EmployeesRow`, which is passed to the `DisplayDaysOnJob` method. This databinding syntax can appear directly in the `ItemTemplate` (as shown in the declarative syntax below) or can be assigned to the `Text` property of a Label Web control.

> **NOTE**
> Alternatively, instead of passing in an `EmployeesRow` instance, we could just pass in the `HireDate` value using `<%# DisplayDaysOnJob(Eval("HireDate")) %>`. However, the `Eval` method returns an `Object`, so we'd have to change our `DisplayDaysOnJob` method signature to accept an input parameter of type `Object`, instead. We can't blindly cast the `Eval("HireDate")` call to a `DateTime` because the `HireDate` column in the `Employees` table can contain `NULL` values. Therefore, we'd need to accept an `Object` as the input parameter for the `DisplayDaysOnJob` method, check to see if it had a database `NULL` value (which can be accomplished using `Convert.IsDBNull(objectToCheck)`), and then proceed accordingly.

Due to these subtleties I've opted to pass in the entire `EmployeesRow` instance. In the next tutorial we'll see a more fitting example for using the `Eval("columnName")` syntax for passing an input parameter into a formatting method.

The following shows the declarative syntax for our GridView after the TemplateField has been added and the `DisplayDaysOnJob` method called from the `ItemTemplate`:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="EmployeeID" DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:TemplateField HeaderText="Name"
          SortExpression="FirstName">
            <EditItemTemplate>
                <asp:TextBox ID="TextBox1" runat="server"
                    Text='<%# Bind("FirstName") %>'></asp:TextBox>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("FirstName") %>'></asp:Label>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Eval("LastName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="Title" HeaderText="" Title"
            SortExpression="Title" />
        <asp:TemplateField HeaderText="HireDate" SortExpression="
            HireDate">
            <EditItemTemplate>
                <asp:TextBox ID="TextBox2" runat="server"
                    Text='<%# Bind("HireDate") %>'></asp:TextBox>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Calendar ID="Calendar1" runat="server"
                    SelectedDate='<%# Bind("HireDate") %>'
                    VisibleDate='<%# Eval("HireDate") %>'>
                </asp:Calendar>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Days On The Job">
            <ItemTemplate>
                <%# DisplayDaysOnJob(CType(CType(Container.DataItem, DataRowView).Row,
                    Northwind.EmployeesRow)) %>
            </ItemTemplate>
            <ItemStyle HorizontalAlign="Center" />
        </asp:TemplateField>
    </Columns>
</asp:GridView>
```

Figure 16 shows the completed tutorial, when viewed through a browser.

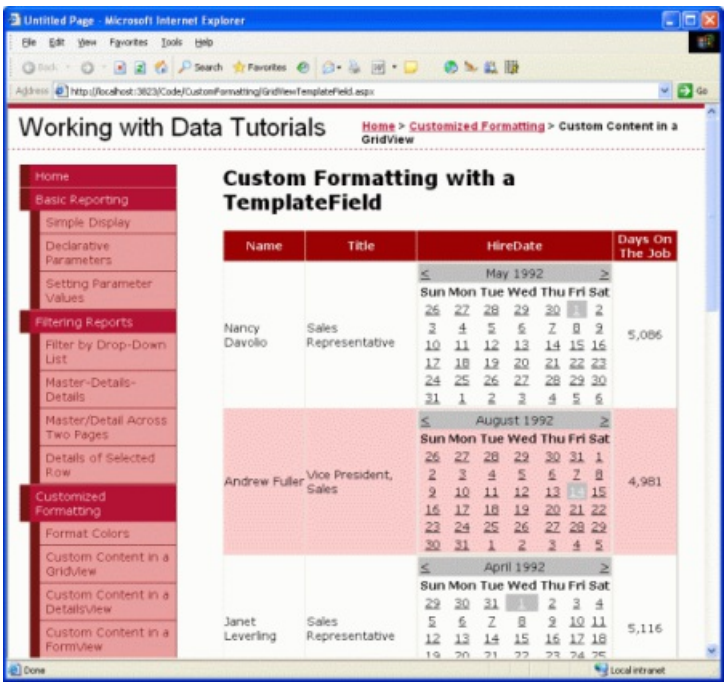


Figure 16: The Number of Days the Employee Has Been on the Job is Displayed (Click to view full-size image)

Summary

The TemplateField in the GridView control allows for a higher degree of flexibility in displaying data than is available with the other field controls. TemplateFields are ideal for situations where:

- Multiple data fields need to be displayed in one GridView column
- The data is best expressed using a Web control rather than plain text
- The output depends on the underlying data, such as displaying metadata or in reformatting the data

In addition to customizing the display of data, TemplateFields are also used for customizing the user interfaces used for editing and inserting data, as we'll see in future tutorials.

The next two tutorials continue exploring templates, starting with a look at using TemplateFields in a DetailsView. Following that, we'll turn to the FormView, which uses templates in lieu of fields to provide greater flexibility in the layout and structure of the data.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Dan Jagers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Using TemplateFields in the DetailsView Control (VB)

2/20/2020 • 11 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> The same TemplateFields capabilities available with the GridView are also available with the DetailsView control. In this tutorial we'll display one product at a time using a DetailsView containing TemplateFields.

Introduction

The TemplateField offers a higher degree of flexibility in rendering data than the BoundField, CheckBoxField, HyperLinkField, and other data field controls. In the previous tutorial we looked at using the TemplateField in a GridView to:

- Display multiple data field values in one column. Specifically, both the `FirstName` and `LastName` fields were combined into one GridView column.
- Use an alternate Web control to express a data field value. We saw how to show the `HiredDate` value using a Calendar control.
- Show status information based on the underlying data. While the `Employees` table does not contain a column that returns the number of days an employee has been on the job, we were able to display such information in the GridView example in the previous tutorial with the use of a TemplateField and formatting method.

The same TemplateFields capabilities available with the GridView are also available with the DetailsView control. In this tutorial we'll display one product at a time using a DetailsView that contains two TemplateFields. The first TemplateField will combine the `UnitPrice`, `UnitsInStock`, and `UnitsOnOrder` data fields into one DetailsView row. The second TemplateField will display the value of the `Discontinued` field, but will use a formatting method to display "YES" if `Discontinued` is `True`, and "NO" otherwise.

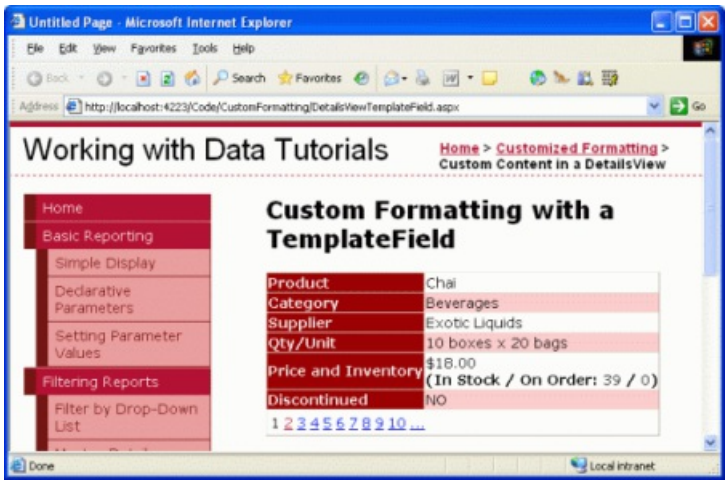


Figure 1: Two TemplateFields are Used to Customize the Display (Click to view full-size image)

Let's get started!

Step 1: Binding the Data to the DetailsView

As discussed in the previous tutorial, when working with TemplateFields it's often easiest to start by creating the DetailsView control that contains just BoundFields and then add new TemplateFields or convert the existing

BoundFields to TemplateFields as needed. Therefore, start this tutorial by adding a DetailsView to the page through the Designer and binding it to an ObjectDataSource that returns the list of products. These steps will create a DetailsView with BoundFields for each of the product's non-Boolean value fields and a CheckBoxField for the one Boolean value field (Discontinued).

Open the `DetailsViewTemplateField.aspx` page and drag a DetailsView from the Toolbox onto the Designer. From the DetailsView's smart tag choose to add a new ObjectDataSource control that invokes the `ProductsBLL` class's `GetProducts()` method.

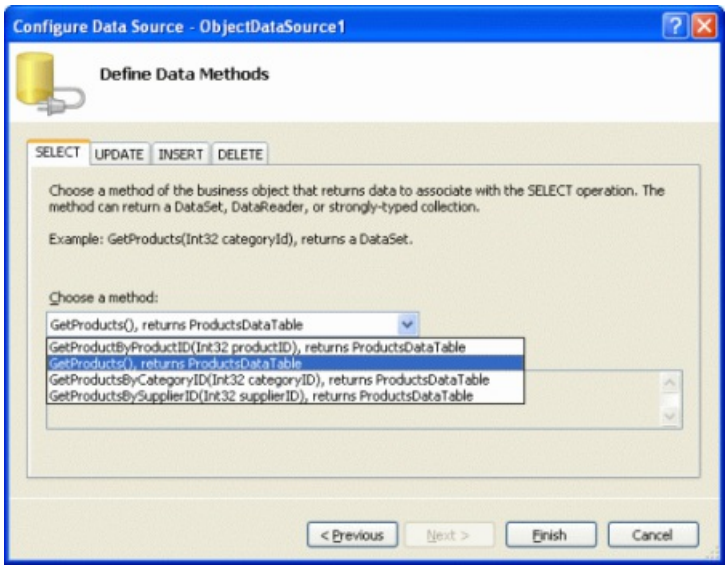


Figure 2: Add a New ObjectDataSource Control that Invokes the `GetProducts()` Method (Click to view full-size image)

For this report remove the `ProductID`, `SupplierID`, `CategoryID`, and `ReorderLevel` BoundFields. Next, reorder the BoundFields so that the `CategoryName` and `SupplierName` BoundFields appear immediately after the `ProductName` BoundField. Feel free to adjust the `HeaderText` properties and formatting properties for the BoundFields as you see fit. Like with the GridView, these BoundField-level edits can be performed through the Fields dialog box (accessible by clicking on the Edit Fields link in the DetailsView's smart tag) or through the declarative syntax. Lastly, clear out the DetailsView's `Height` and `Width` property values in order to allow the DetailsView control to expand based on the data displayed and check the Enable Paging checkbox in the smart tag.

After making these changes, your DetailsView control's declarative markup should look similar to the following:

```
<asp:DetailsView ID="DetailsView1" runat="server" AutoGenerateRows="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1" AllowPaging="True"
    EnableViewState="False">
    <Fields>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
          SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
          ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
          ReadOnly="True" SortExpression="SupplierName" />
        <asp:BoundField DataField="QuantityPerUnit"
          HeaderText="Qty/Unit" SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
          SortExpression="UnitPrice" />
        <asp:BoundField DataField="UnitsInStock"
          HeaderText="Units In Stock" SortExpression="UnitsInStock" />
        <asp:BoundField DataField="UnitsOnOrder"
          HeaderText="Units On Order" SortExpression="UnitsOnOrder" />
        <asp:CheckBoxField DataField="Discontinued"
          HeaderText="Discontinued" SortExpression="Discontinued" />
    </Fields>
</asp:DetailsView>
```

Take a moment to view the page through a browser. At this point you should see a single product listed (Chai) with rows showing the product's name, category, supplier, price, units in stock, units on order, and its discontinued status.

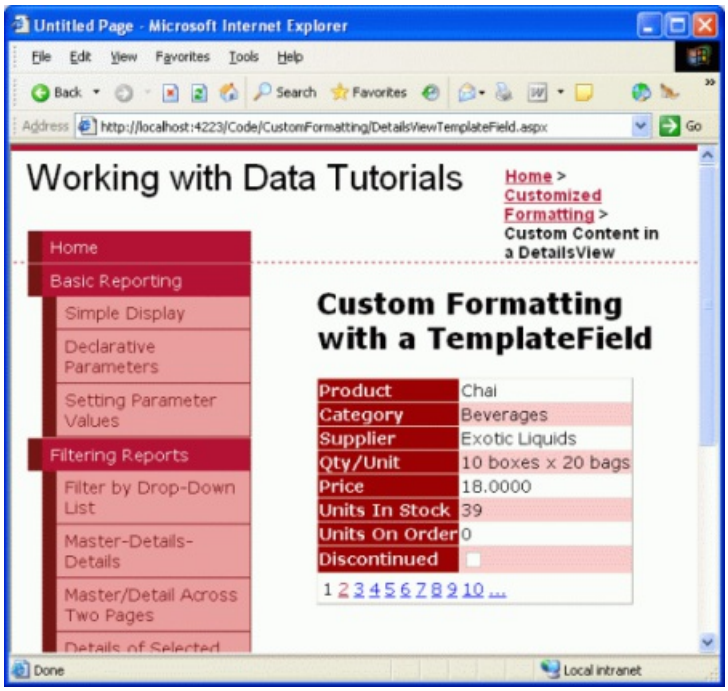


Figure 3: The Product's Details Are Shown Using a Series of BoundFields (Click to view full-size image)

Step 2: Combining the Price, Units In Stock, and Units On Order Into One Row

The DetailsView has a row for the `UnitPrice`, `UnitsInStock`, and `UnitsOnOrder` fields. We can combine these data fields into a single row with a TemplateField either by adding a new TemplateField or by converting one of the existing `UnitPrice`, `UnitsInStock`, and `UnitsOnOrder` BoundFields into a TemplateField. While I personally prefer converting existing BoundFields, let's practice by adding a new TemplateField.

Start by clicking on the Edit Fields link in the DetailsView's smart tag to bring up the Fields dialog box. Next, add a new TemplateField and set its `HeaderText` property to "Price and Inventory" and move the new TemplateField so

that it is positioned above the `UnitPrice` BoundField.

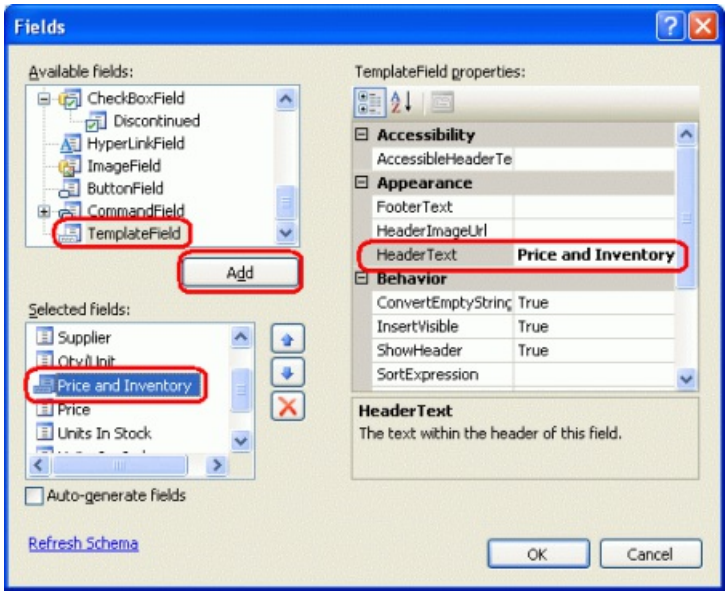


Figure 4: Add a New TemplateField to the DetailsView Control (Click to view full-size image)

Since this new TemplateField will contain the values currently displayed in the `UnitPrice`, `UnitsInStock`, and `UnitsOnOrder` BoundFields, let's remove them.

The last task for this step is to define the `ItemTemplate` markup for the Price and Inventory TemplateField, which can be accomplished either through the DetailsView's template editing interface in the Designer or by hand through the control's declarative syntax. As with the GridView, the DetailsView's template editing interface can be accessed by clicking on the Edit Templates link in the smart tag. From here you can select the template to edit from the drop-down list and then add any Web controls from the Toolbox.

For this tutorial, start by adding a Label control to the Price and Inventory TemplateField's `ItemTemplate`. Next, click on the Edit DataBindings link from the Label Web control's smart tag and bind the `Text` property to the `UnitPrice` field.

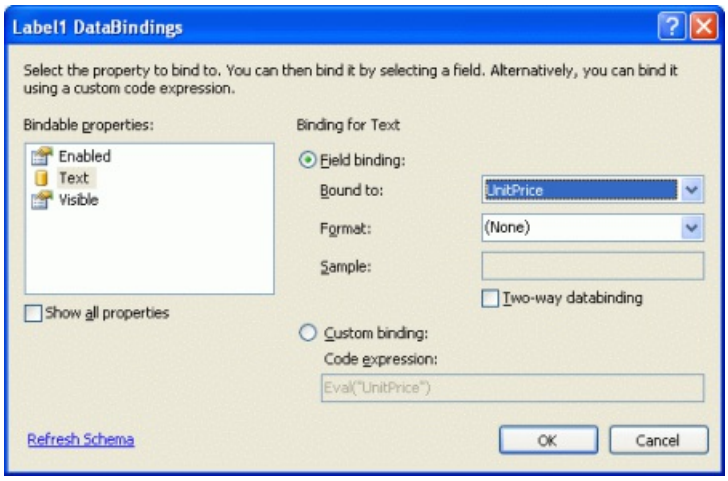


Figure 5: Bind the Label's `Text` Property to the `UnitPrice` Data Field (Click to view full-size image)

Formatting the Price as a Currency

With this addition, the Label Web control Price and Inventory TemplateField will now display just the price for the selected product. Figure 6 shows a screen shot of our progress thus far when viewed through a browser.

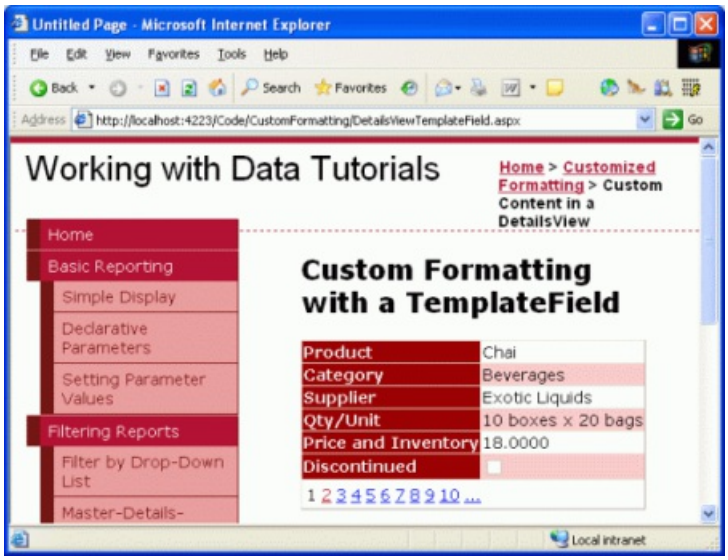


Figure 6: The Price and Inventory TemplateField Shows the Price (Click to view full-size image)

Note that the product's price is not formatted as a currency. With a BoundField, formatting is possible by setting the `HtmlEncode` property to `False` and the `DataFormatString` property to `{0:formatSpecifier}`. For a TemplateField, however, any formatting instructions must be specified in the databinding syntax or through the use of a formatting method defined somewhere within the application's code (such as in the ASP.NET page's code-behind class).

To specify the formatting for the databinding syntax used in the Label Web control, return to the DataBindings dialog box by clicking on the Edit DataBindings link from the Label's smart tag. You can type the formatting instructions directly in the Format drop-down list or select one of the defined format strings. Like with the BoundField's `DataFormatString` property, the formatting is specified using `{0:formatSpecifier}`.

For the `UnitPrice` field use the currency formatting specified either by selecting the appropriate drop-down list value or by typing in `{0:C}` by hand.

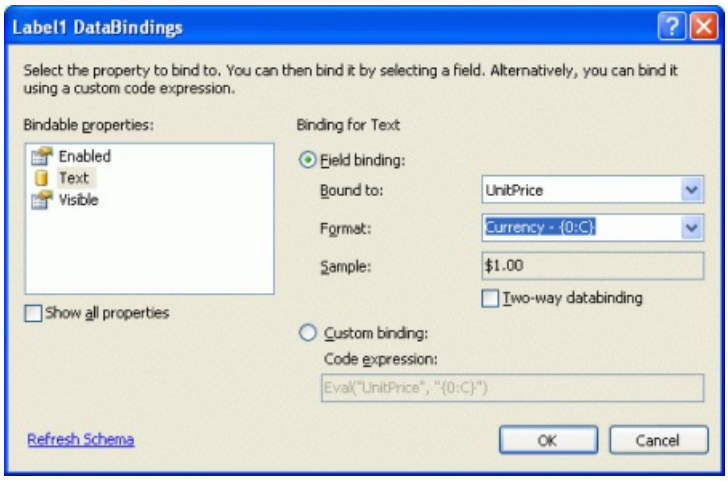


Figure 7: Format the Price as a Currency (Click to view full-size image)

Declaratively, the formatting specification is indicated as a second parameter into the `Bind` or `Eval` methods. The settings just made through the Designer result in the following databinding expression in the declarative markup:

```
<asp:Label ID="Label1" runat="server" Text='<%# Eval("UnitPrice", "{0:C}") %>'/>
```

Adding the Remaining Data Fields to the TemplateField

At this point we've displayed and formatted the `UnitPrice` data field in the Price and Inventory TemplateField, but still need to display the `UnitsInStock` and `UnitsOnOrder` fields. Let's display these on a line below the price and in parentheses. From the template editing interface in the Designer, such markup can be added by positioning your cursor within the template and simply typing in the text to be displayed. Alternatively, this markup can be entered directly in the declarative syntax.

Add the static markup, Label Web controls, and databinding syntax so that the Price and Inventory TemplateField displays the price and inventory information like so:

UnitPrice
(**In Stock / On Order:** *UnitsInStock* / *UnitsOnOrder*)

After performing this task your DetailsView's declarative markup should look similar to the following:

```
<asp:DetailsView ID="DetailsView1" runat="server" AutoGenerateRows="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1" AllowPaging="True"
    EnableViewState="False">
    <Fields>
        <asp:BoundField DataField="ProductName"
          HeaderText="Product" SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
          ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName"
          HeaderText="Supplier" ReadOnly="True"
          SortExpression="SupplierName" />
        <asp:BoundField DataField="QuantityPerUnit"
          HeaderText="Qty/Unit" SortExpression="QuantityPerUnit" />
        <asp:TemplateField HeaderText="Price and Inventory">
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                  Text='<%# Eval("UnitPrice", "{0:C}") %>'></asp:Label>
                <br />
                <strong>
                (In Stock / On Order: </strong>
                <asp:Label ID="Label2" runat="server"
                  Text='<%# Eval("UnitsInStock") %>'></asp:Label>
                <strong>/</strong>
                <asp:Label ID="Label3" runat="server"
                  Text='<%# Eval("UnitsOnOrder") %>'>
                </asp:Label><strong>)</strong>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:CheckBoxField DataField="Discontinued"
           HeaderText="Discontinued" SortExpression="Discontinued" />
    </Fields>
</asp:DetailsView>
```

With these changes we've consolidated the price and inventory information into a single DetailsView row.

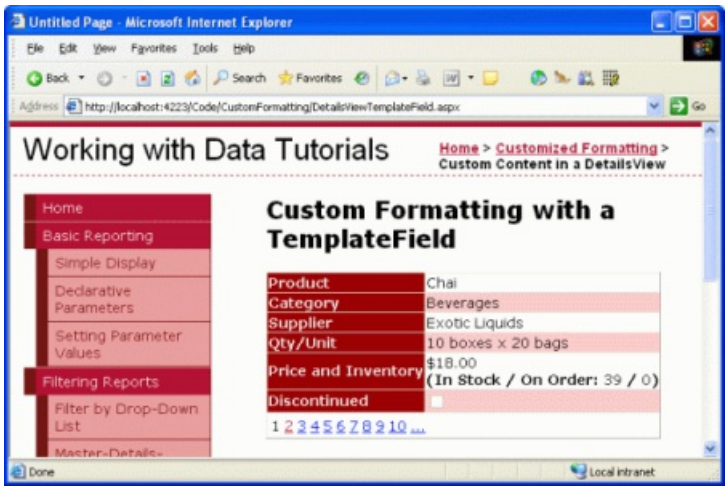


Figure 8: The Price and Inventory Information is Displayed in a Single Row (Click to view full-size image)

Step 3: Customizing the Discontinued Field Information

The `Products` table's `Discontinued` column is a bit value that indicates whether the product has been discontinued. When binding a DetailsView (or GridView) to a data source control, the Boolean value fields, like `Discontinued`, are implemented as CheckBoxFields whereas non-Boolean value fields, like `ProductID`, `ProductName`, and so on, are implemented as BoundFields. The CheckBoxField renders as a disabled checkbox that is checked if the data field's value is True and unchecked otherwise.

Rather than display the CheckBoxField we may want to instead display text indicating whether or not the product is discontinued. To accomplish this we could remove the CheckBoxField from the DetailsView and then add a BoundField whose `DataField` property was set to `Discontinued`. Take a moment to do this. After this change the DetailsView shows the text "True" for discontinued products and "False" for products that are still active.

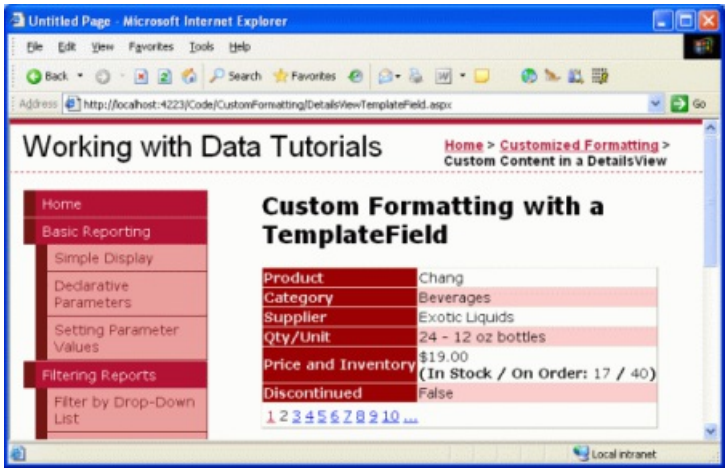


Figure 9: The Strings True and False Are Used to Display the Discontinued State (Click to view full-size image)

Imagine that we didn't want the strings "True" or "False" to be used, but "YES" and "NO", instead. Such customization can be performed with the aid of a TemplateField and a formatting method. A formatting method can take in any number of input parameters, but must return the HTML (as a string) to inject into the template.

Add a formatting method to the `DetailsViewTemplateField.aspx` page's code-behind class named `DisplayDiscontinuedAsYESorNO` that accepts a `Northwind.ProductsRow` object as an input parameter and returns a string. As discussed in the previous tutorial, this method *must* be marked as `Protected` or `Public` in order to be accessible from the template.

```
Protected Function DisplayDiscontinuedAsYESorNO(discontinued As Boolean) As String
    If discontinued Then
        Return "YES"
    Else
        Return "NO"
    End If
End Function
```

This method checks the input parameter (`discontinued`) and returns "YES" if it is `True`, "NO" otherwise.

> **NOTE**
> In the formatting method examined in the previous tutorial recall that we were passing in a data field that might contain `NULL` s and therefore needed to check if the employee's `HiredDate` property value had a database `NULL` value before accessing the `EmployeesRow`'s `HiredDate` property. Such a check is not needed here since the `Discontinued` column can never have database `NULL` values assigned. Moreover, this is why the method can accept a Boolean input parameter rather than having to accept a `ProductsRow` instance or a parameter of type `Object`.

With this formatting method complete, all that remains is to call it from the TemplateField's `ItemTemplate`. To create the TemplateField either remove the `Discontinued` BoundField and add a new TemplateField or convert the `Discontinued` BoundField into a TemplateField. Then, from the declarative markup view, edit the TemplateField so that it contains just an ItemTemplate that invokes the `DisplayDiscontinuedAsYESorNO` method, passing in the value of the current `ProductRow` instance's `Discontinued` property. This can be accessed via the `Eval` method. Specifically, the TemplateField's markup should look like:

```
<asp:TemplateField HeaderText="Discontinued" SortExpression="Discontinued">
    <ItemTemplate>
        <%#DisplayDiscontinuedAsYESorNO(Convert.ToBoolean(Eval("Discontinued")))%>
    </ItemTemplate>
</asp:TemplateField>
```

This will cause the `DisplayDiscontinuedAsYESorNO` method to be invoked when rendering the DetailsView, passing in the `ProductRow` instance's `Discontinued` value. Since the `Eval` method returns a value of type `Object`, but the `DisplayDiscontinuedAsYESorNO` method expects an input parameter of type `Boolean`, we cast the `Eval` methods return value to `Boolean`. The `DisplayDiscontinuedAsYESorNO` method will then return "YES" or "NO" depending on the value it receives. The returned value is what is displayed in this DetailsView row (see Figure 10).

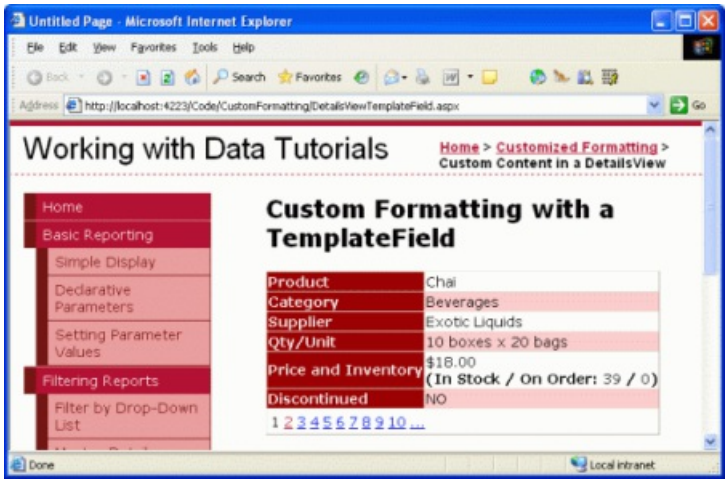


Figure 10: YES or NO Values are Now Shown in the Discontinued Row (Click to view full-size image)

Summary

The TemplateField in the DetailsView control allows for a higher degree of flexibility in displaying data than is available with the other field controls and are ideal for situations where:

- Multiple data fields need to be displayed in one GridView column
- The data is best expressed using a Web control rather than plain text
- The output depends on the underlying data, such as displaying metadata or in reformatting the data

While TemplateFields allow for a greater degree of flexibility in the rendering of the DetailsView's underlying data, the DetailsView output still feels a bit boxy as each field is rendered as a row in an HTML `<table>`.

The FormView control offers a greater degree of flexibility in configuring the rendered output. The FormView does not contain fields but rather just a series of templates (`ItemTemplate`, `EditItemTemplate`, `HeaderTemplate`, and so on). We'll see how to use the FormView to achieve even more control of the rendered layout in our next tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Dan Jagers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Using the FormView's Templates (VB)

2/20/2020 • 6 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> Unlike the DetailsView, the FormView is not composed of fields. Instead, the FormView is rendered using templates. In this tutorial we'll examine using the FormView control to present a less rigid display of data.

Introduction

In the last two tutorials we saw how to customize the GridView and DetailsView controls' outputs using TemplateFields. TemplateFields allow for the contents for a specific field to be highly customized, but in the end both the GridView and DetailsView have a rather boxy, grid-like appearance. For many scenarios such a grid-like layout is ideal, but at times a more fluid, less rigid display is needed. When displaying a single record, such a fluid layout is possible using the FormView control.

Unlike the DetailsView, the FormView is not composed of fields. You can't add a BoundField or TemplateField to a FormView. Instead, the FormView is rendered using templates. Think of the FormView as a DetailsView control that contains a single TemplateField. The FormView supports the following templates:

- `ItemTemplate` used to render the particular record displayed in the FormView
- `HeaderTemplate` used to specify an optional header row
- `FooterTemplate` used to specify an optional footer row
- `EmptyDataTemplate` when the FormView's `DataSource` lacks any records, the `EmptyDataTemplate` is used in place of the `ItemTemplate` for rendering the control's markup
- `PagerTemplate` can be used to customize the paging interface for FormViews that have paging enabled
- `EditItemTemplate` / `InsertItemTemplate` used to customize the editing interface or inserting interface for FormViews that support such functionality

In this tutorial we'll examine using the FormView control to present a less rigid display of products. Rather than having fields for the name, category, supplier, and so on, the FormView's `ItemTemplate` will show these values using a combination of a header element and a `<table>` (see Figure 1).

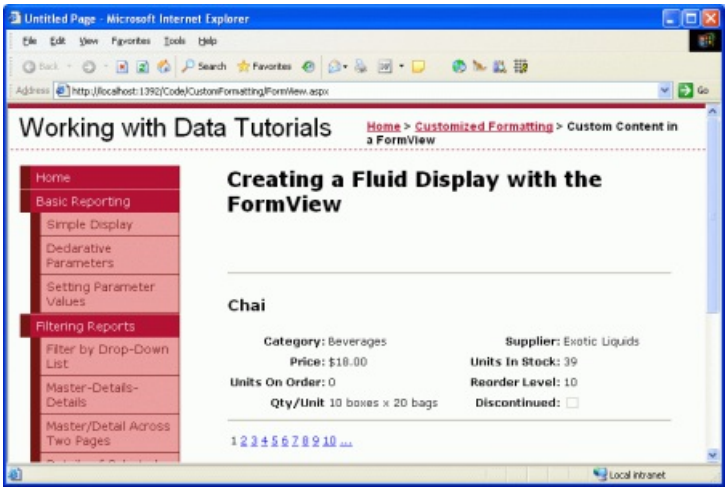


Figure 1: The FormView Breaks Out of the Grid-Like Layout Seen in the DetailsView (Click to view full-size image)

Step 1: Binding the Data to the FormView

Open the `FormView.aspx` page and drag a FormView from the Toolbox onto the Designer. When first adding the FormView it appears as a gray box, instructing us that an `ItemTemplate` is needed.

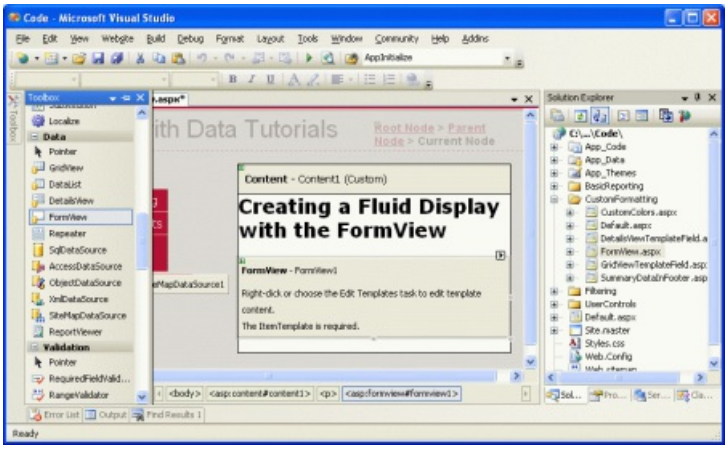


Figure 2: The FormView Cannot be Rendered in the Designer Until an `ItemTemplate` is Provided (Click to view full-size image)

The `ItemTemplate` can be created by hand (through the declarative syntax) or can be auto-created by binding the FormView to a data source control through the Designer. This auto-created `ItemTemplate` contains HTML that lists the name of each field and a Label control whose `Text` property is bound to the field's value. This approach also auto-creates an `InsertItemTemplate` and `EditItemTemplate`, both of which are populated with input controls for each of the data fields returned by the data source control.

If you want to auto-create the template, from the FormView's smart tag add a new ObjectDataSource control that invokes the `ProductsBLL` class's `GetProducts()` method. This will create a FormView with an `ItemTemplate`, `InsertItemTemplate`, and `EditItemTemplate`. From the Source view, remove the `InsertItemTemplate` and `EditItemTemplate` since we're not interested in creating a FormView that supports editing or inserting yet. Next, clear out the markup within the `ItemTemplate` so that we have a clean slate to work from.

If you'd rather build up the `ItemTemplate` manually, you can add and configure the ObjectDataSource by dragging it from the Toolbox onto the Designer. However, don't set the FormView's data source from the Designer. Instead, go to the Source view and manually set the FormView's `DataSourceID` property to the `ID` value of the ObjectDataSource. Next, manually add the `ItemTemplate`.

Regardless of what approach you decided to take, at this point your FormView's declarative markup should look like:

```
<asp:FormView ID="FormView1" runat="server" DataSourceID="ObjectDataSource1">
    <ItemTemplate>

    </ItemTemplate>
</asp:FormView>
```

Take a moment to check the Enable Paging checkbox in the FormView's smart tag; this will add the `AllowPaging="True"` attribute to the FormView's declarative syntax. Also, set the `EnableViewState` property to False.

Step 2: Defining the `ItemTemplate`'s Markup

With the FormView bound to the ObjectDataSource control and configured to support paging we're ready to specify the content for the `ItemTemplate`. For this tutorial, let's have the product's name displayed in an `<h3>`

heading. Following that, let's use an HTML `<table>` to display the remaining product properties in a four-column table where the first and third columns list the property names and the second and fourth list their values.

This markup can be entered in through the FormView's template editing interface in the Designer or entered manually through the declarative syntax. When working with templates I typically find it quicker to work directly with the declarative syntax, but feel free to use whatever technique you're most comfortable with.

The following markup shows the FormView declarative markup after the `ItemTemplate`'s structure has been completed:

```
<asp:FormView ID="FormView1" runat="server" DataSourceID="ObjectDataSource1"
    AllowPaging="True" EnableViewState="False">
    <ItemTemplate>
        <hr />
        <h3><%# Eval("ProductName") %></h3>
        <table border="0">
            <tr>
                <td class="ProductPropertyLabel">Category:</td>
                <td class="ProductPropertyValue">
                  <%# Eval("CategoryName") %></td>
                <td class="ProductPropertyLabel">Supplier:</td>
                <td class="ProductPropertyValue">
                  <%# Eval("SupplierName")%></td>
            </tr>
            <tr>
                <td class="ProductPropertyLabel">Price:</td>
                <td class="ProductPropertyValue"><%# Eval("UnitPrice",
                  "{0:C}") %></td>
                <td class="ProductPropertyLabel">Units In Stock:</td>
                <td class="ProductPropertyValue">
                  <%# Eval("UnitsInStock")%></td>
            </tr>
            <tr>
                <td class="ProductPropertyLabel">Units On Order:</td>
                <td class="ProductPropertyValue">
                  <%# Eval("UnitsOnOrder") %></td>
                <td class="ProductPropertyLabel">Reorder Level:</td>
                <td class="ProductPropertyValue">
                  <%# Eval("ReorderLevel")%></td>
            </tr>
            <tr>
                <td class="ProductPropertyLabel">Qty/Unit</td>
                <td class="ProductPropertyValue">
                  <%# Eval("QuantityPerUnit") %></td>
                <td class="ProductPropertyLabel">Discontinued:</td>
                <td class="ProductPropertyValue">
                    <asp:CheckBox runat="server" Enabled="false"
                      Checked='<%# Eval("Discontinued") %>' />
                </td>
            </tr>
        </table>
        <hr />
    </ItemTemplate>
</asp:FormView>
```

Notice that the databinding syntax - `<%# Eval("ProductName") %>`, for example can be injected directly into the template's output. That is, it need not be assigned to a Label control's `Text` property. For example, we have the `ProductName` value displayed in an `<h3>` element using `<h3><%# Eval("ProductName") %></h3>`, which for the product Chai will render as `<h3>Chai</h3>`.

The `ProductPropertyLabel` and `ProductPropertyValue` CSS classes are used for specifying the style of the product property names and values in the `<table>`. These CSS classes are defined in `Styles.css` and cause the property names to be bold and right-aligned and add a right padding to the property values.

Since there are no CheckBoxFields available with the FormView, in order to show the `Discontinued` value as a checkbox we must add our own CheckBox control. The `Enabled` property is set to False, making it read-only, and the CheckBox's `Checked` property is bound to the value of the `Discontinued` data field.

With the `ItemTemplate` complete, the product information is displayed in a much more fluid manner. Compare the DetailsView output from the last tutorial (Figure 3) with the output generated by the FormView in this tutorial (Figure 4).

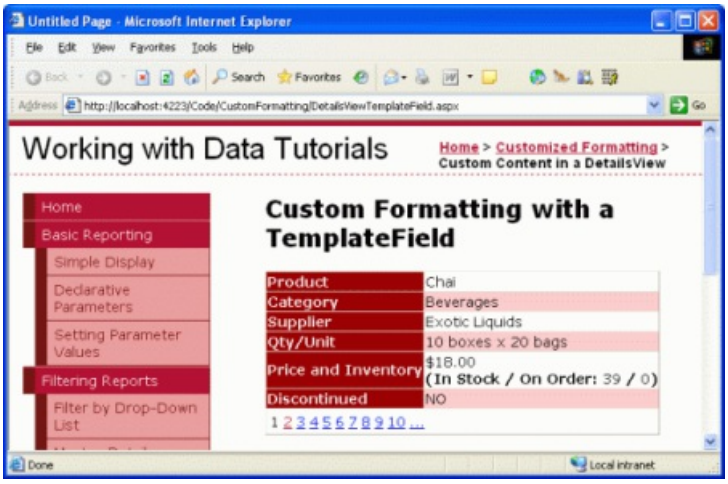


Figure 3: The Rigid DetailsView Output (Click to view full-size image)

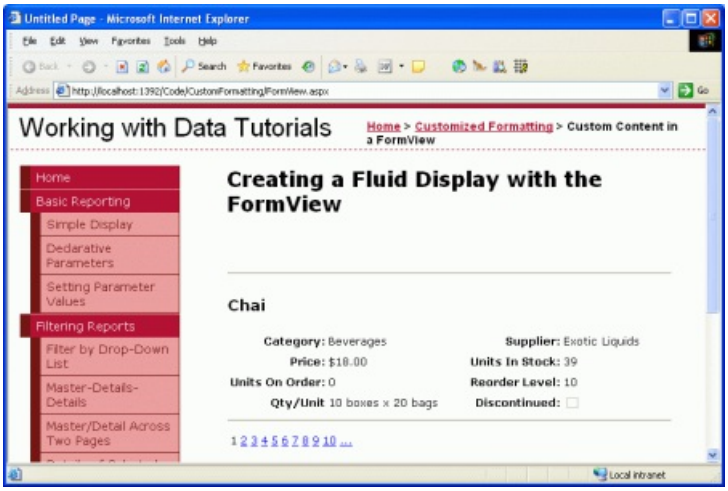


Figure 4: The Fluid FormView Output (Click to view full-size image)

Summary

While the GridView and DetailsView controls can have their output customized using TemplateFields, both still present their data in a grid-like, boxy format. For those times when a single record needs to be shown using a less rigid layout, the FormView is an ideal choice. Like the DetailsView, the FormView renders a single record from its `DataSource`, but unlike the DetailsView it is composed just of templates and does not support fields.

As we saw in this tutorial, the FormView allows for a more flexible layout when displaying a single record. In future tutorials we'll examine the DataList and Repeater controls, which provide the same level of flexibility as the FormsView, but are able to display multiple records (like the GridView).

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest

book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was E.R. Gilmore. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Displaying Summary Information in the GridView's Footer (VB)

2/20/2020 • 11 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

Summary information is often displayed at the bottom of the report in a summary row. The GridView control can include a footer row into whose cells we can programmatically inject aggregate data. In this tutorial we'll see how to display aggregate data in this footer row.

Introduction

In addition to seeing each of the products' prices, units in stock, units on order, and reorder levels, a user might also be interested in aggregate information, such as the average price, the total number of units in stock, and so on. Such summary information is often displayed at the bottom of the report in a summary row. The GridView control can include a footer row into whose cells we can programmatically inject aggregate data.

This task presents us with three challenges:

1. Configuring the GridView to display its footer row
2. Determining the summary data; that is, how do we compute the average price or the total of the units in stock?
3. Injecting the summary data into the appropriate cells of the footer row

In this tutorial we'll see how to overcome these challenges. Specifically, we'll create a page that lists the categories in a drop-down list with the selected category's products displayed in a GridView. The GridView will include a footer row that shows the average price and total number of units in stock and on order for products in that category.

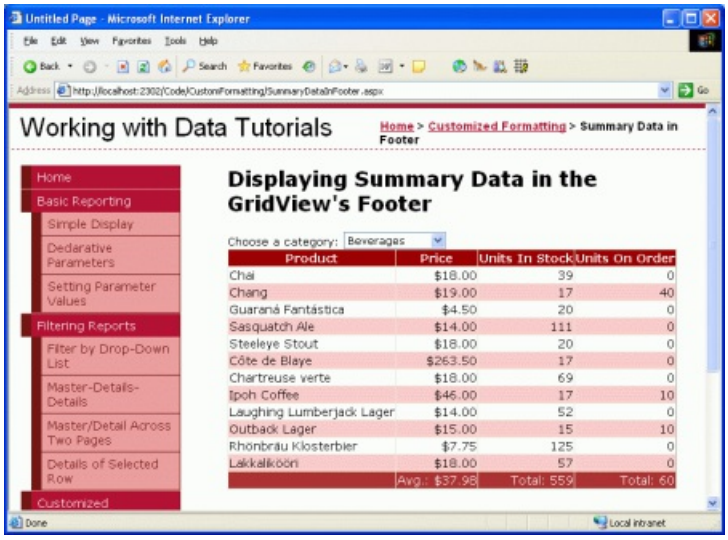


Figure 1: Summary Information is Displayed in the GridView's Footer Row (Click to view full-size image)

This tutorial, with its category to products master/detail interface, builds upon the concepts covered in the earlier Master/Detail Filtering With a DropDownList tutorial. If you've not yet worked through the earlier tutorial, please do so before continuing on with this one.

Step 1: Adding the Categories DropDownList and Products GridView

Before concerning ourselves with adding summary information to the GridView's footer, let's first simply build the master/detail report. Once we've completed this first step, we'll look at how to include summary data.

Start by opening the `SummaryDataInFooter.aspx` page in the `CustomFormatting` folder. Add a DropDownList control and set its `ID` to `Categories`. Next, click on the Choose Data Source link from the DropDownList's smart tag and opt to add a new ObjectDataSource named `CategoriesDataSource` that invokes the `CategoriesBLL` class's `GetCategories()` method.

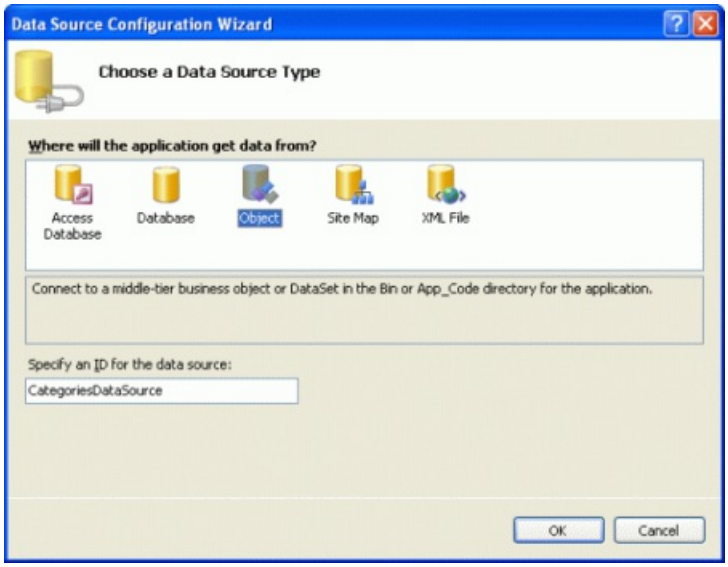


Figure 2: Add a New ObjectDataSource Named `CategoriesDataSource` (Click to view full-size image)

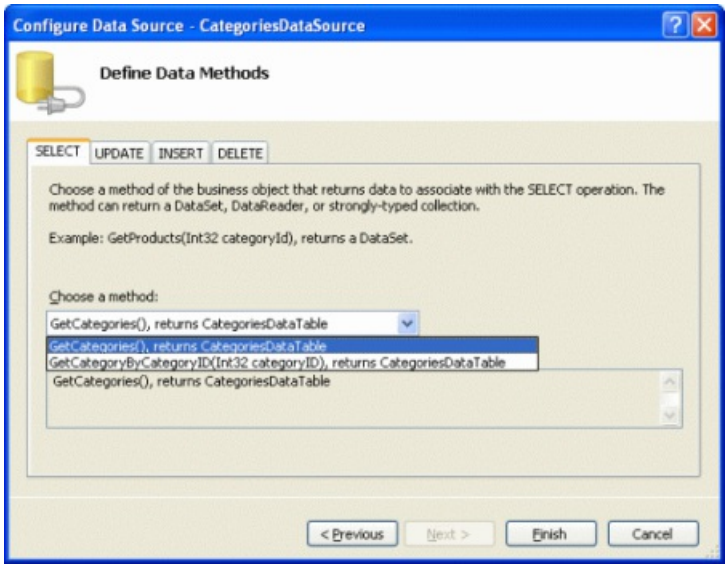


Figure 3: Have the ObjectDataSource Invoke the `CategoriesBLL` Class's `GetCategories()` Method (Click to view full-size image)

After configuring the ObjectDataSource, the wizard returns us to the DropDownList's Data Source Configuration wizard from which we need to specify what data field value should be displayed and which one should correspond to the value of the DropDownList's `ListItem` s. Have the `CategoryName` field displayed and use the `CategoryID` as the value.

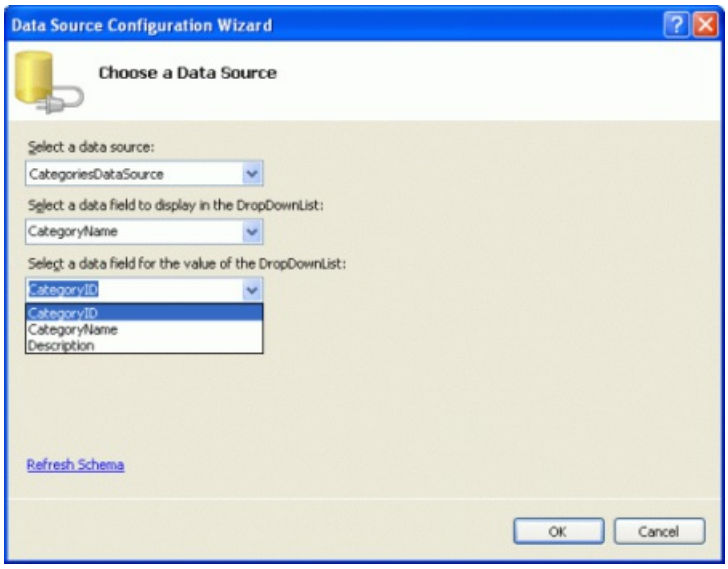


Figure 4: Use the `CategoryName` and `CategoryID` Fields as the `Text` and `Value` for the `ListItem` s, Respectively (Click to view full-size image)

At this point we have a DropDownList (`Categories`) that lists the categories in the system. We now need to add a GridView that lists those products that belong to the selected category. Before we do, though, take a moment to check the Enable AutoPostBack checkbox in the DropDownList's smart tag. As discussed in the *Master/Detail Filtering With a DropDownList* tutorial, by setting the DropDownList's `AutoPostBack` property to `True` the page will be posted back each time the DropDownList value is changed. This will cause the GridView to be refreshed, showing those products for the newly selected category. If the `AutoPostBack` property is set to `False` (the default), changing the category won't cause a postback and therefore won't update the listed products.

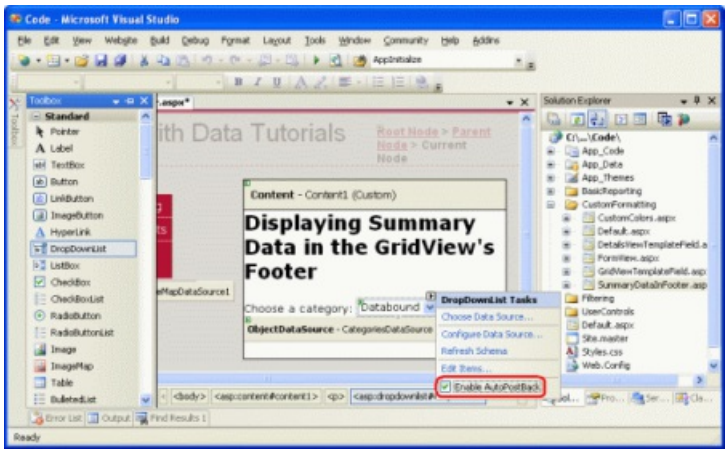


Figure 5: Check the Enable AutoPostBack Checkbox in the DropDownList's Smart Tag (Click to view full-size image)

Add a GridView control to the page in order to display the products for the selected category. Set the GridView's `ID` to `ProductsInCategory` and bind it to a new ObjectDataSource named `ProductsInCategoryDataSource` .

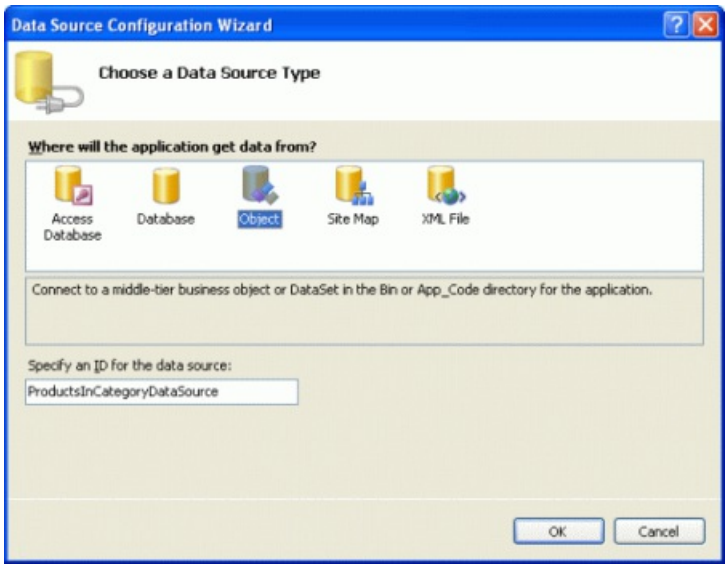


Figure 6: Add a New ObjectDataSource Named `ProductsInCategoryDataSource` (Click to view full-size image)

Configure the ObjectDataSource so that it invokes the `ProductsBLL` class's `GetProductsByCategoryID(categoryID)` method.

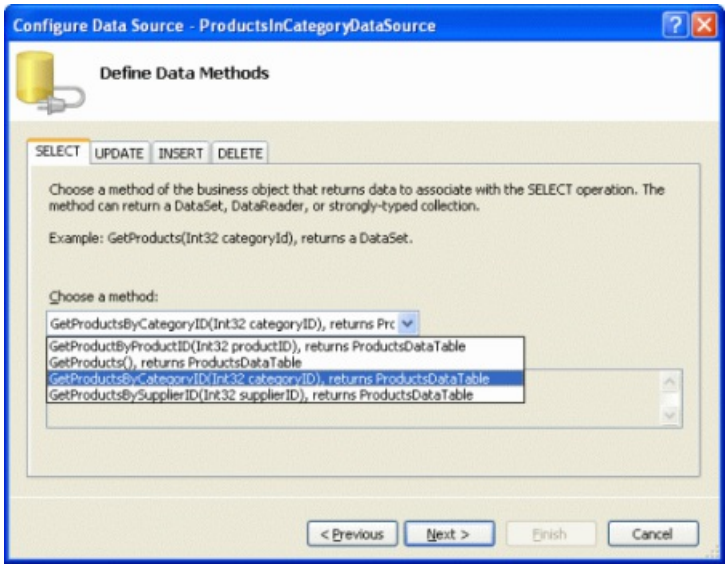


Figure 7: Have the ObjectDataSource Invoke the `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

Since the `GetProductsByCategoryID(categoryID)` method takes in an input parameter, in the final step of the wizard we can specify the source of the parameter value. In order to display those products from the selected category, have the parameter pulled from the `Categories` DropDownList.

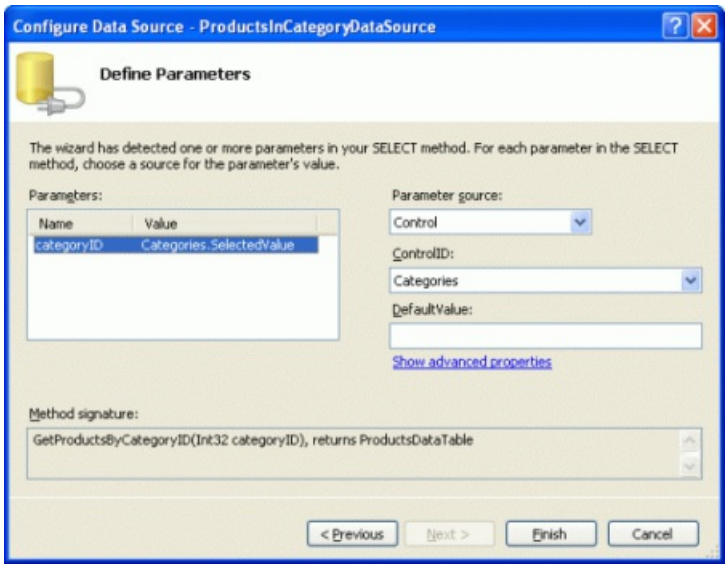


Figure 8: Get the `categoryID` Parameter Value from the Selected Categories DropDownList (Click to view full-size image)

After completing the wizard the GridView will have a BoundField for each of the product properties. Let's clean up these BoundFields so that only the `ProductName`, `UnitPrice`, `UnitsInStock`, and `UnitsOnOrder` BoundFields are displayed. Feel free to add any field-level settings to the remaining BoundFields (such as formatting the `UnitPrice` as a currency). After making these changes, the GridView's declarative markup should look similar to the following:

```
<asp:GridView ID="ProductsInCategory" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsInCategoryDataSource"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
          SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
          HeaderText="Price"
            HtmlEncode="False" SortExpression="UnitPrice">
            <ItemStyle HorizontalAlign="Right" />
        </asp:BoundField>
        <asp:BoundField DataField="UnitsInStock"
          HeaderText="Units In Stock" SortExpression="UnitsInStock">
            <ItemStyle HorizontalAlign="Right" />
        </asp:BoundField>
        <asp:BoundField DataField="UnitsOnOrder"
          HeaderText="Units On Order" SortExpression="UnitsOnOrder">
            <ItemStyle HorizontalAlign="Right" />
        </asp:BoundField>
    </Columns>
</asp:GridView>
```

At this point we have a fully functioning master/detail report that shows the name, unit price, units in stock, and units on order for those products that belong to the selected category.

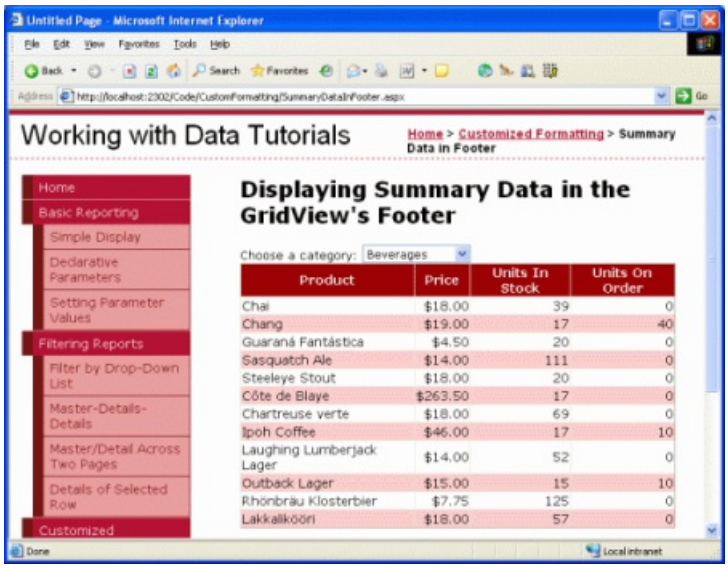


Figure 9: Get the `categoryID` Parameter Value from the Selected Categories DropDownList (Click to view full-size image)

Step 2: Displaying a Footer in the GridView

The GridView control can display both a header and footer row. These rows are displayed depending on the values of the `ShowHeader` and `ShowFooter` properties, respectively, with `ShowHeader` defaulting to `True` and `ShowFooter` to `False`. To include a footer in the GridView simply set its `ShowFooter` property to `True`.

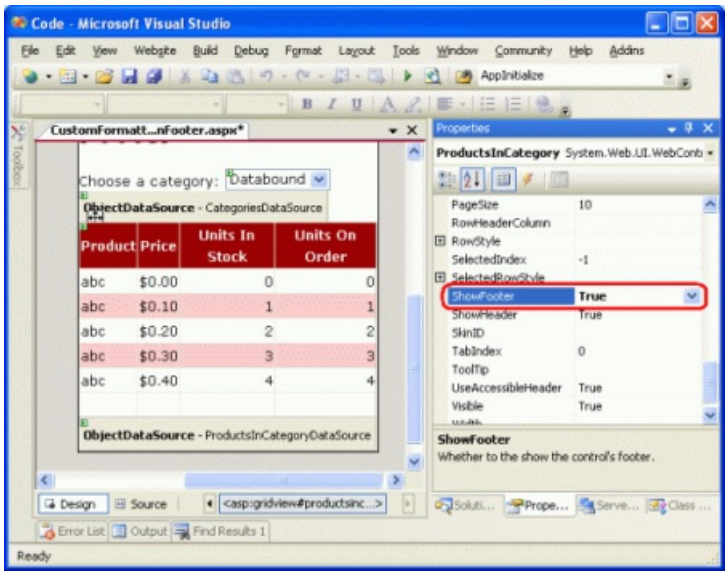


Figure 10: Set the GridView's `ShowFooter` Property to `True` (Click to view full-size image)

The footer row has a cell for each of the fields defined in the GridView; however, these cells are empty by default. Take a moment to view our progress in a browser. With the `ShowFooter` property now set to `True`, the GridView includes an empty footer row.

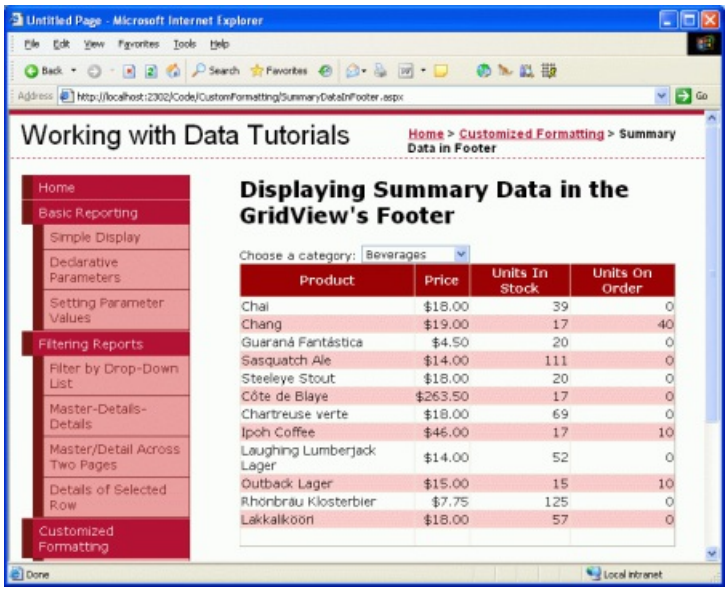


Figure 11: The GridView Now Includes a Footer Row (Click to view full-size image)

The footer row in Figure 11 doesn't stand out, as it has a white background. Let's create a `FooterStyle` CSS class in `Styles.css` that specifies a dark red background and then configure the `GridView.skin` Skin file in the `DataWebControls` Theme to assign this CSS class to the GridView's `FooterStyle`'s `CssClass` property. If you need to brush up on Skins and Themes, refer back to the Displaying Data With the ObjectDataSource tutorial.

Start by adding the following CSS class to `Styles.css`:

```
.FooterStyle
{
    background-color: #a33;
    color: White;
    text-align: right;
}
```

The `FooterStyle` CSS class is similar in style to the `HeaderStyle` class, although the `HeaderStyle`'s background color is subtlety darker and its text is displayed in a bold font. Furthermore, the text in the footer is right-aligned whereas the header's text is centered.

Next, to associate this CSS class with every GridView's footer, open the `GridView.skin` file in the `DataWebControls` Theme and set the `FooterStyle`'s `CssClass` property. After this addition the file's markup should look like:

```
<asp:GridView runat="server" CssClass="DataWebControlStyle">
   <AlternatingRowStyle CssClass="AlternatingRowStyle" />
   <RowStyle CssClass="RowStyle" />
   <HeaderStyle CssClass="HeaderStyle" />
   <FooterStyle CssClass="FooterStyle" />
   <SelectedRowStyle CssClass="SelectedRowStyle" />
</asp:GridView>
```

As the screen shot below shows, this change makes the footer stand out more clearly.

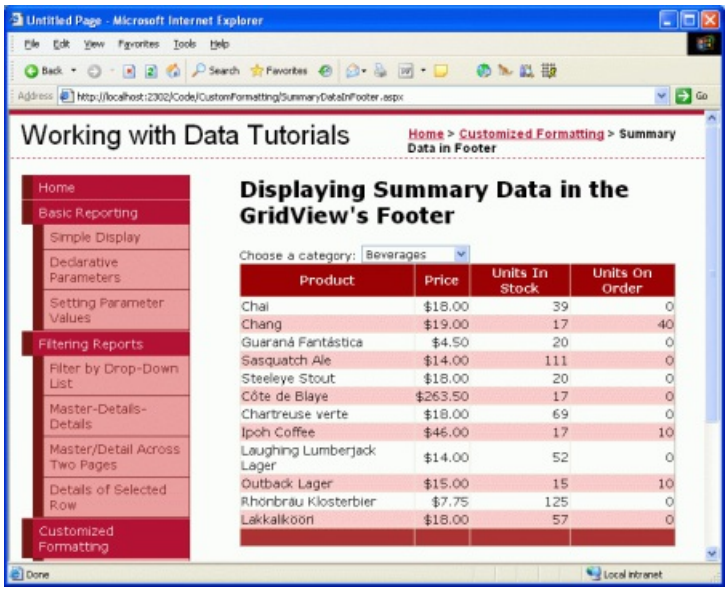


Figure 12: The GridView's Footer Row Now Has a Reddish Background Color (Click to view full-size image)

Step 3: Computing the Summary Data

With the GridView's footer displayed, the next challenge facing us is how to compute the summary data. There are two ways to compute this aggregate information:

1. Through a SQL query we could issue an additional query to the database to compute the summary data for a particular category. SQL includes a number of aggregate functions along with a `GROUP BY` clause to specify the data over which the data should be summarized. The following SQL query would bring back the needed information:

   ```
   SELECT CategoryID, AVG(UnitPrice), SUM(UnitsInStock),
   SUM(UnitsOnOrder)
   FROM Products
   WHERE CategoryID = categoryID
   GROUP BY CategoryID
   ```

   Of course you wouldn't want to issue this query directly from the `SummaryDataInFooter.aspx` page, but rather by creating a method in the `ProductsTableAdapter` and the `ProductsBLL`.

2. Compute this information as it's being added to the GridView as discussed in Custom Formatting Based Upon Data tutorial, the GridView's `RowDataBound` event handler fires once for each row being added to the GridView after its been databound. By creating an event handler for this event we can keep a running total of the values we want to aggregate. After the last data row has been bound to the GridView we have the totals and the information needed to compute the average.

I typically employ the second approach as it saves a trip to the database and the effort needed to implement the summary functionality in the Data Access Layer and Business Logic Layer, but either approach would suffice. For this tutorial let's use the second option and keep track of the running total using the `RowDataBound` event handler.

Create a `RowDataBound` event handler for the GridView by selecting the GridView in the Designer, clicking the lightning bolt icon from the Properties window, and double-clicking the `RowDataBound` event. Alternatively, you can select the GridView and its RowDataBound event from the drop-down lists at the top of the ASP.NET code-behind class file. This will create a new event handler named `ProductsInCategory_RowDataBound` in the `SummaryDataInFooter.aspx` page's code-behind class.

```
Protected Sub ProductsInCategory_RowDataBound _
    (sender As Object, e As GridViewRowEventArgs) _
        Handles ProductsInCategory.RowDataBound
End Sub
```

In order to maintain a running total we need to define variables outside of the scope of the event handler. Create the following four page-level variables:

- `_totalUnitPrice`, of type `Decimal`
- `_totalNonNullUnitPriceCount`, of type `Integer`
- `_totalUnitsInStock`, of type `Integer`
- `_totalUnitsOnOrder`, of type `Integer`

Next, write the code to increment these three variables for each data row encountered in the `RowDataBound` event handler.

```
Dim _totalUnitPrice As Decimal = 0
Dim _totalNonNullUnitPriceCount As Integer = 0
Dim _totalUnitsInStock As Integer = 0
Dim _totalUnitsOnOrder As Integer = 0
Protected Sub ProductsInCategory_RowDataBound _
    (sender As Object, e As GridViewRowEventArgs) _
        Handles ProductsInCategory.RowDataBound
    If e.Row.RowType = DataControlRowType.DataRow Then
        Dim product As Northwind.ProductsRow = _
            CType(CType(e.Row.DataItem, DataRowView).Row, Northwind.ProductsRow)
        If Not product.IsUnitPriceNull() Then
            _totalUnitPrice += product.UnitPrice
            _totalNonNullUnitPriceCount += 1
        End If
        If Not product.IsUnitsInStockNull() Then
            _totalUnitsInStock += product.UnitsInStock
        End If
        If Not product.IsUnitsOnOrderNull() Then
            _totalUnitsOnOrder += product.UnitsOnOrder
        End If
    ElseIf e.Row.RowType = DataControlRowType.Footer Then
        Dim avgUnitPrice As Decimal = _
            _totalUnitPrice / CType(_totalNonNullUnitPriceCount, Decimal)
        e.Row.Cells(1).Text = "Avg.: " & avgUnitPrice.ToString("c")
        e.Row.Cells(2).Text = "Total: " & _totalUnitsInStock.ToString()
        e.Row.Cells(3).Text = "Total: " & _totalUnitsOnOrder.ToString()
    End If
End Sub
```

The `RowDataBound` event handler starts by ensuring that we're dealing with a DataRow. Once that's been established, the `Northwind.ProductsRow` instance that was just bound to the `GridViewRow` object in `e.Row` is stored in the variable `product`. Next, running total variables are incremented by the current product's corresponding values (assuming that they don't contain a database `NULL` value). We keep track of both the running `UnitPrice` total and the number of non-`NULL` `UnitPrice` records because the average price is the quotient of these two numbers.

Step 4: Displaying the Summary Data in the Footer

With the summary data totaled, the last step is to display it in the GridView's footer row. This task, too, can be accomplished programmatically through the `RowDataBound` event handler. Recall that the `RowDataBound` event handler fires for *every* row that's bound to the GridView, including the footer row. Therefore, we can augment our event handler to display the data in the footer row using the following code:

```
Protected Sub ProductsInCategory_RowDataBound _
    (sender As Object, e As GridViewRowEventArgs) _
        Handles ProductsInCategory.RowDataBound
    If e.Row.RowType = DataControlRowType.DataRow Then
      ... Increment the running totals ...
    ElseIf e.Row.RowType = DataControlRowType.Footer
      ... Display the summary data in the footer ...
    End If
End Sub
```

Since the footer row is added to the GridView after all of the data rows have been added, we can be confident that by the time we're ready to display the summary data in the footer the running total calculations will have completed. The last step, then, is to set these values in the footer's cells.

To display text in a particular footer cell, use `e.Row.Cells(index).Text = value`, where the `Cells` indexing starts at 0. The following code computes the average price (the total price divided by the number of products) and displays it along with the total number of units in stock and units on order in the appropriate footer cells of the GridView.

```
Protected Sub ProductsInCategory_RowDataBound _
    (sender As Object, e As GridViewRowEventArgs) _
        Handles ProductsInCategory.RowDataBound
    If e.Row.RowType = DataControlRowType.DataRow Then
      ... <i>Increment the running totals</i> ...
    ElseIf e.Row.RowType = DataControlRowType.Footer
      Dim avgUnitPrice As Decimal = _
        _totalUnitPrice / CType(_totalNonNullUnitPriceCount, Decimal)
      e.Row.Cells(1).Text = "Avg.: " & avgUnitPrice.ToString("c")
      e.Row.Cells(2).Text = "Total: " & _totalUnitsInStock.ToString()
      e.Row.Cells(3).Text = "Total: " & _totalUnitsOnOrder.ToString()
    End If
End Sub
```

Figure 13 shows the report after this code has been added. Note how the `ToString("c")` causes the average price summary information to be formatted like a currency.

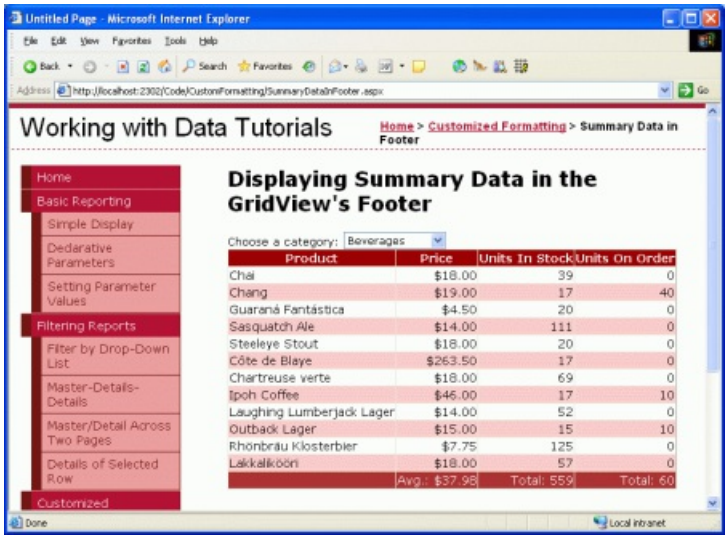


Figure 13: The GridView's Footer Row Now Has a Reddish Background Color (Click to view full-size image)

Summary

Displaying summary data is a common report requirement, and the GridView control makes it easy to include such information in its footer row. The footer row is displayed when the GridView's `ShowFooter` property is set to `True` and can have the text in its cells set programmatically through the `RowDataBound` event handler. Computing

the summary data can either be done by re-querying the database or by using code in the ASP.NET page's code-behind class to programmatically compute the summary data.

This tutorial concludes our examination of custom formatting with the GridView, DetailsView, and FormView controls. Our next tutorial kicks off our exploration of inserting, updating, and deleting data using these same controls.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS

Editing, Inserting, and Deleting Data

2/20/2020 • 2 minutes to read • Edit Online

In these tutorials you see how to map methods of the ObjectDataSource control to BLL methods, and how to configure the GridView, DetailsView, and FormView controls to let users modify data.

- Overview of Inserting, Updating, and Deleting Data (C#)
- Examining the Events Associated with Inserting, Updating, and Deleting (C#)
- Handling BLL- and DAL-Level Exceptions in an ASP.NET Page (C#)
- Adding Validation Controls to the Editing and Inserting Interfaces (C#)
- Customizing the Data Modification Interface (C#)
- Implementing Optimistic Concurrency (C#)
- Adding Client-Side Confirmation When Deleting (C#)
- Limiting Data Modification Functionality Based on the User (C#)
- Overview of Inserting, Updating, and Deleting Data (VB)
- Examining the Events Associated with Inserting, Updating, and Deleting (VB)
- Handling BLL- and DAL-Level Exceptions in an ASP.NET Page (VB)
- Adding Validation Controls to the Editing and Inserting Interfaces (VB)
- Customizing the Data Modification Interface (VB)
- Implementing Optimistic Concurrency (VB)
- Adding Client-Side Confirmation When Deleting (VB)
- Limiting Data Modification Functionality Based on the User (VB)

An Overview of Inserting, Updating, and Deleting Data (C#)

2/20/2020 • 32 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll see how to map an ObjectDataSource's Insert(), Update(), and Delete() methods to the methods of BLL classes, as well as how to configure the GridView, DetailsView, and FormView controls to provide data modification capabilities.

Introduction

Over the past several tutorials we've examined how to display data in an ASP.NET page using the GridView, DetailsView, and FormView controls. These controls simply work with data supplied to them. Commonly, these controls access data through the use of a data source control, such as the ObjectDataSource. We've seen how the ObjectDataSource acts as a proxy between the ASP.NET page and the underlying data. When a GridView needs to display data, it invokes its ObjectDataSource's `Select()` method, which in turn invokes a method from our Business Logic Layer (BLL), which calls a method in the appropriate Data Access Layer's (DAL) TableAdapter, which in turn sends a `SELECT` query to the Northwind database.

Recall that when we created the TableAdapters in the DAL in our first tutorial, Visual Studio automatically added methods for inserting, updating, and deleting data from the underlying database table. Moreover, in Creating a Business Logic Layer we designed methods in the BLL that called down into these data modification DAL methods.

In addition to its `Select()` method, the ObjectDataSource also has `Insert()`, `Update()`, and `Delete()` methods. Like the `Select()` method, these three methods can be mapped to methods in an underlying object. When configured to insert, update, or delete data, the GridView, DetailsView, and FormView controls provide a user interface for modifying the underlying data. This user interface calls the `Insert()`, `Update()`, and `Delete()` methods of the ObjectDataSource, which then invoke the underlying object's associated methods (see Figure 1).

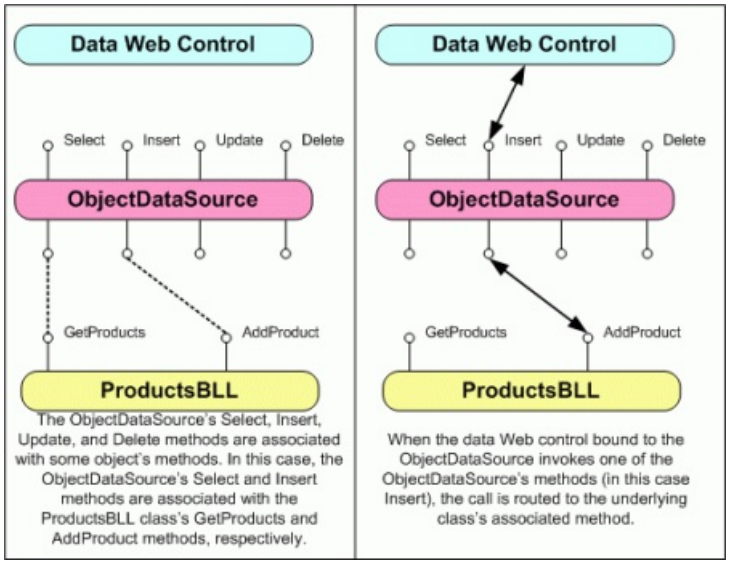


Figure 1: The ObjectDataSource's `Insert()`, `Update()`, and `Delete()` Methods Serve as a Proxy into the BLL (Click to view full-size image)

In this tutorial we'll see how to map the ObjectDataSource's `Insert()`, `Update()`, and `Delete()` methods to methods of classes in the BLL, as well as how to configure the GridView, DetailsView, and FormView controls to provide data modification capabilities.

Step 1: Creating the Insert, Update, and Delete Tutorials Web Pages

Before we start exploring how to insert, update, and delete data, let's first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial and the next several ones. Start by adding a new folder named `EditInsertDelete`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `Basics.aspx`
- `DataModificationEvents.aspx`
- `ErrorHandling.aspx`
- `UIValidation.aspx`
- `CustomizedUI.aspx`
- `OptimisticConcurrency.aspx`
- `ConfirmationOnDelete.aspx`
- `UserLevelAccess.aspx`

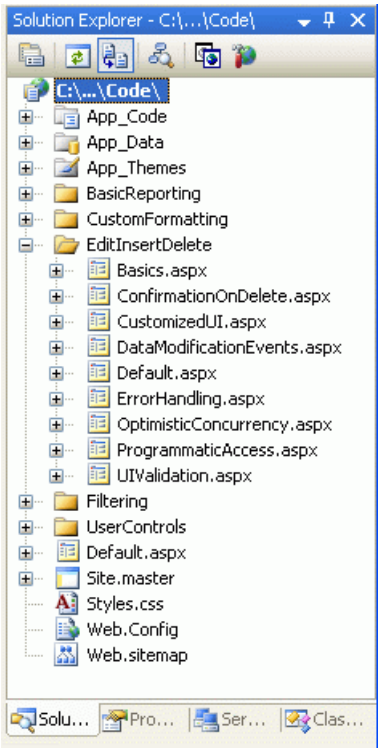


Figure 2: Add the ASP.NET Pages for the Data Modification-Related Tutorials

Like in the other folders, `Default.aspx` in the `EditInsertDelete` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page's Design view.

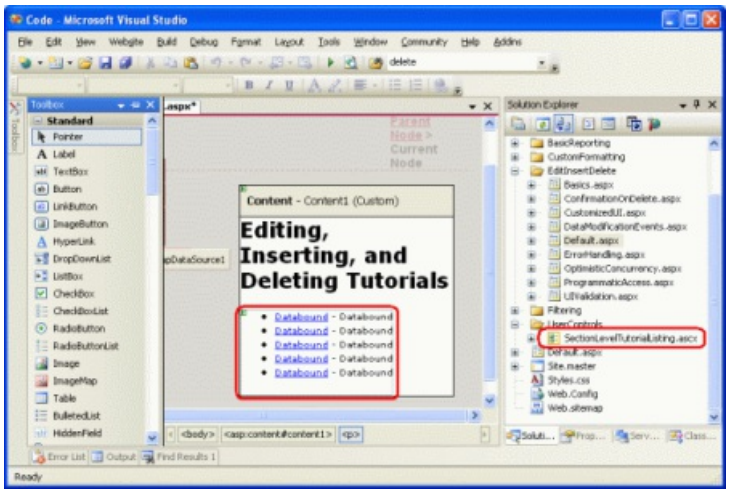


Figure 3: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add the pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Customized Formatting `<siteMapNode>`:

```
<siteMapNode title="Editing, Inserting, and Deleting"
    url="~/EditInsertDelete/Default.aspx"
    description="Samples of Reports that Provide Editing, Inserting,
                 and Deleting Capabilities">
    <siteMapNode url="~/EditInsertDelete/Basics.aspx"
        title="Basics"
        description="Examines the basics of data modification with the
                     GridView, DetailsView, and FormView controls." />
    <siteMapNode url="~/EditInsertDelete/DataModificationEvents.aspx"
        title="Data Modification Events"
        description="Explores the events raised by the ObjectDataSource
                     pertinent to data modification." />
    <siteMapNode url="~/EditInsertDelete/ErrorHandling.aspx"
        title="Error Handling"
        description="Learn how to gracefully handle exceptions raised
                     during the data modification workflow." />
    <siteMapNode url="~/EditInsertDelete/UIValidation.aspx"
        title="Adding Data Entry Validation"
        description="Help prevent data entry errors by providing validation." />
    <siteMapNode url="~/EditInsertDelete/CustomizedUI.aspx"
        title="Customize the User Interface"
        description="Customize the editing and inserting user interfaces." />
    <siteMapNode url="~/EditInsertDelete/OptimisticConcurrency.aspx"
        title="Optimistic Concurrency"
        description="Learn how to help prevent simultaneous users from
                     overwritting one another s changes." />
    <siteMapNode url="~/EditInsertDelete/ConfirmationOnDelete.aspx"
        title="Confirm On Delete"
        description="Prompt a user for confirmation when deleting a record." />
    <siteMapNode url="~/EditInsertDelete/UserLevelAccess.aspx"
        title="Limit Capabilities Based on User"
        description="Learn how to limit the data modification functionality
                     based on the user role or permissions." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the editing, inserting, and deleting tutorials.

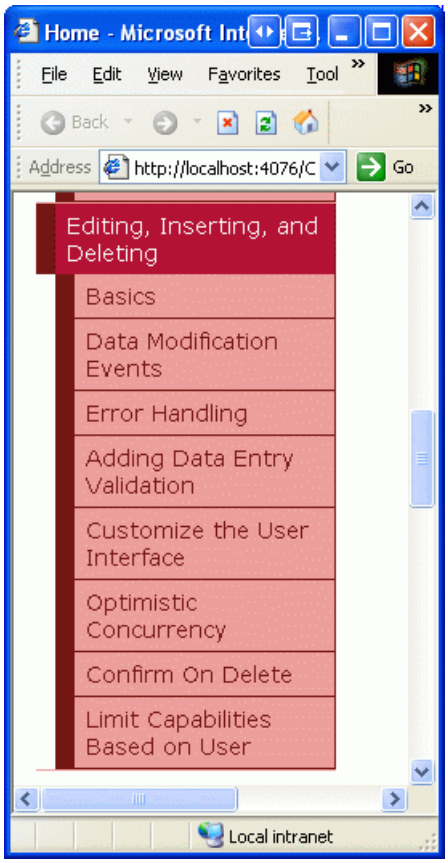


Figure 4: The Site Map Now Includes Entries for the Editing, Inserting, and Deleting Tutorials

Step 2: Adding and Configuring the ObjectDataSource Control

Since the GridView, DetailsView, and FormView each differ in their data modification capabilities and layout, let's examine each one individually. Rather than have each control using its own ObjectDataSource, however, let's just create a single ObjectDataSource that all three control examples can share.

Open the `Basics.aspx` page, drag an ObjectDataSource from the Toolbox onto the Designer, and click the Configure Data Source link from its smart tag. Since the `ProductsBLL` is the only BLL class that provides editing, inserting, and deleting methods, configure the ObjectDataSource to use this class.

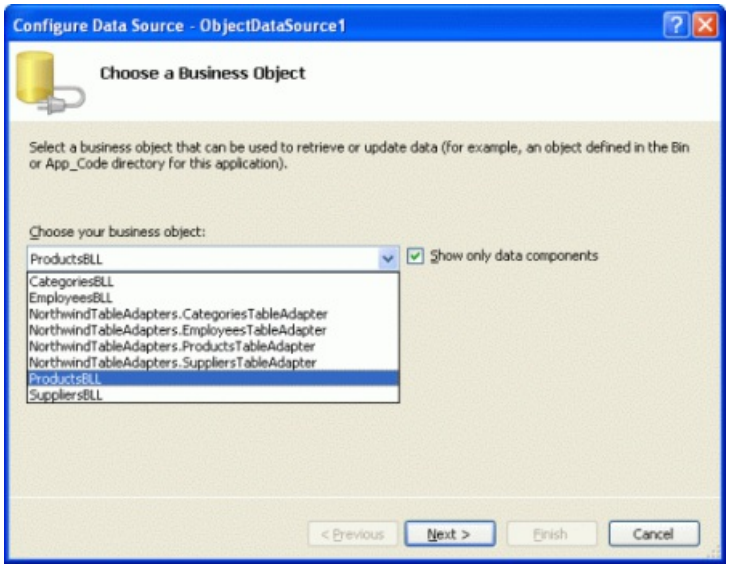


Figure 5: Configure the ObjectDataSource to Use the `ProductsBLL` Class (Click to view full-size image)

In the next screen we can specify what methods of the `ProductsBLL` class are mapped to the ObjectDataSource's `Select()`, `Insert()`, `Update()`, and `Delete()` by selecting the appropriate tab and

choosing the method from the drop-down list. Figure 6, which should look familiar by now, maps the ObjectDataSource's `Select()` method to the `ProductsBLL` class's `GetProducts()` method. The `Insert()`, `Update()`, and `Delete()` methods can be configured by selecting the appropriate tab from the list along the top.

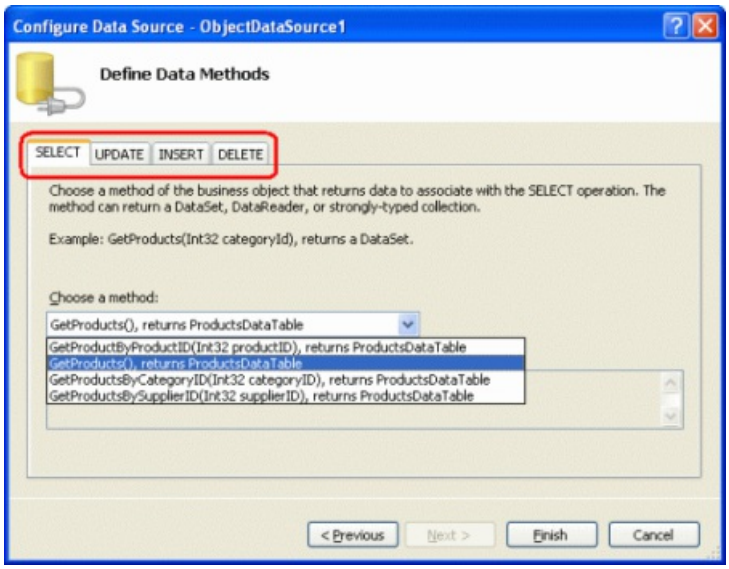


Figure 6: Have the ObjectDataSource Return All of the Products (Click to view full-size image)

Figures 7, 8, and 9 show the ObjectDataSource's UPDATE, INSERT, and DELETE tabs. Configure these tabs so that the `Insert()`, `Update()`, and `Delete()` methods invoke the `ProductsBLL` class's `UpdateProduct`, `AddProduct`, and `DeleteProduct` methods, respectively.

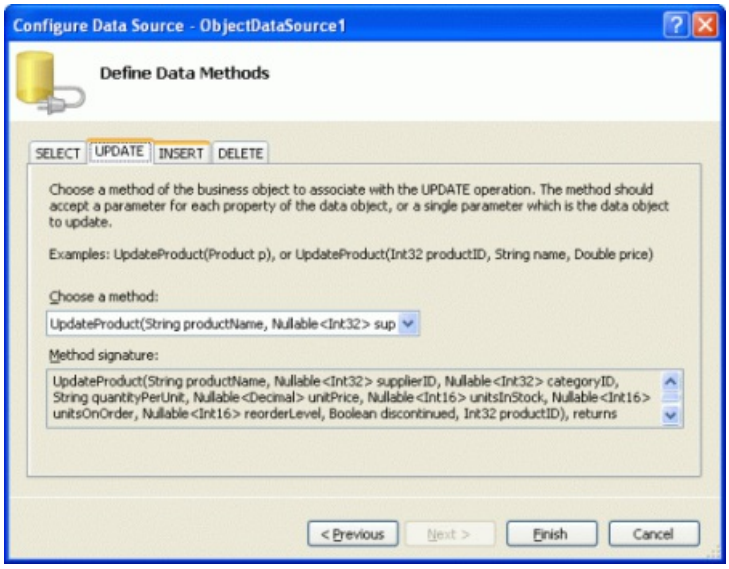


Figure 7: Map the ObjectDataSource's `Update()` Method to the `ProductBLL` Class's `UpdateProduct` Method (Click to view full-size image)

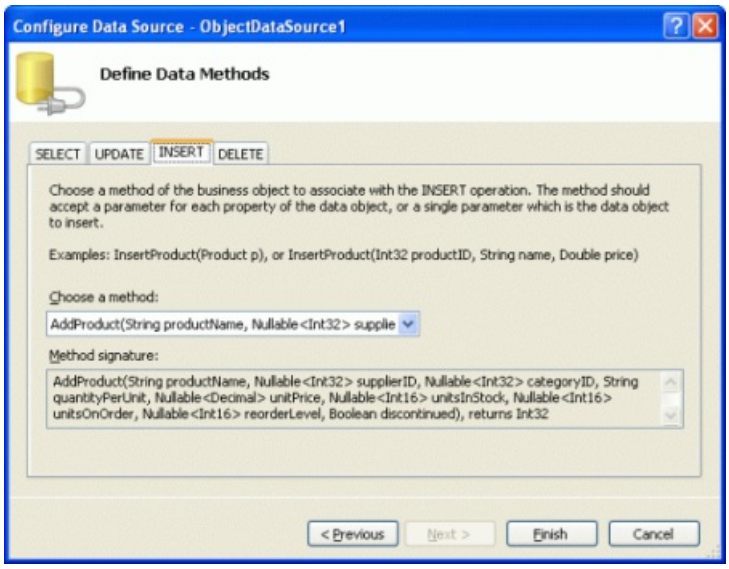


Figure 8: Map the ObjectDataSource's `Insert()` Method to the `ProductBLL` Class's Add `Product` Method (Click to view full-size image)

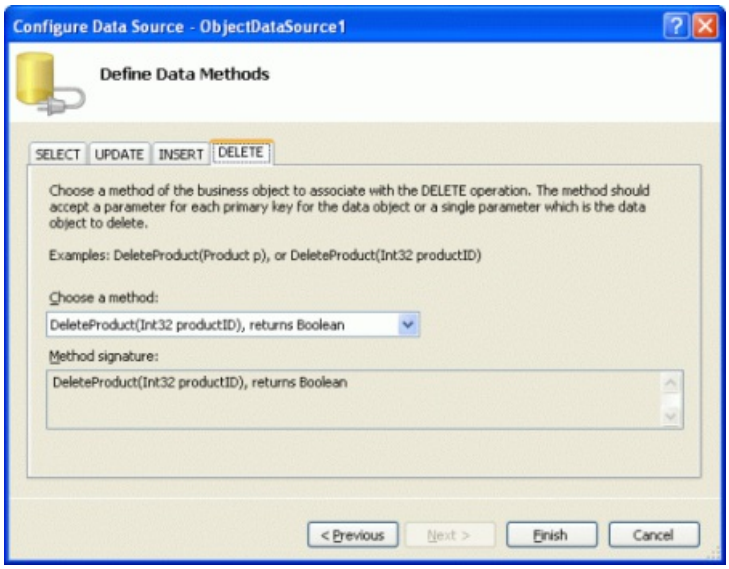


Figure 9: Map the ObjectDataSource's `Delete()` Method to the `ProductBLL` Class's `DeleteProduct` Method (Click to view full-size image)

You may have noticed that the drop-down lists in the UPDATE, INSERT, and DELETE tabs already had these methods selected. This is thanks to our use of the `DataObjectMethodAttribute` that decorates the methods of `ProductsBLL`. For example, the DeleteProduct method has the following signature:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Delete, true)]
public bool DeleteProduct(int productID)
{
    ...
}
```

The `DataObjectMethodAttribute` attribute indicates the purpose of each method whether it is for selecting, inserting, updating, or deleting and whether or not it's the default value. If you omitted these attributes when creating your BLL classes, you'll need to manually select the methods from the UPDATE, INSERT, and DELETE tabs.

After ensuring that the appropriate `ProductsBLL` methods are mapped to the ObjectDataSource's `Insert()`, `Update()`, and `Delete()` methods, click Finish to complete the wizard.

Examining the ObjectDataSource's Markup

After configuring the ObjectDataSource through its wizard, go to the Source view to examine the generated declarative markup. The `<asp:ObjectDataSource>` tag specifies the underlying object and the methods to invoke. Additionally, there are `DeleteParameters`, `UpdateParameters`, and `InsertParameters` that map to the input parameters for the `ProductsBLL` class's `AddProduct`, `UpdateProduct`, and `DeleteProduct` methods:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    DeleteMethod="DeleteProduct" InsertMethod="AddProduct"
    OldValuesParameterFormatString="original_{0}" SelectMethod="GetProducts"
    TypeName="ProductsBLL" UpdateMethod="UpdateProduct">
    <DeleteParameters>
        <asp:Parameter Name="productID" Type="Int32" />
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="unitsInStock" Type="Int16" />
        <asp:Parameter Name="unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="reorderLevel" Type="Int16" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
    <InsertParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="unitsInStock" Type="Int16" />
        <asp:Parameter Name="unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="reorderLevel" Type="Int16" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
    </InsertParameters>
</asp:ObjectDataSource>
```

The ObjectDataSource includes a parameter for each of the input parameters for its associated methods, just like a list of `SelectParameter` s is present when the ObjectDataSource is configured to call a select method that expects an input parameter (such as `GetProductsByCategoryID(categoryID)`). As we'll see shortly, values for these `DeleteParameters`, `UpdateParameters`, and `InsertParameters` are set automatically by the GridView, DetailsView, and FormView prior to invoking the ObjectDataSource's `Insert()`, `Update()`, or `Delete()` method. These values can also be set programmatically as needed, as we'll discuss in a future tutorial.

One side effect of using the wizard to configure to ObjectDataSource is that Visual Studio sets the OldValuesParameterFormatString property to `original_{0}`. This property value is used to include the original values of the data being edited and is useful in two scenarios:

- If, when editing a record, users are able to change the primary key value. In this case, both the new primary key value and the original primary key value must be provided so that the record with the original primary key value can be found and have its value updated accordingly.
- When using optimistic concurrency. Optimistic concurrency is a technique to ensure that two simultaneous users don't overwrite one another's changes, and is the topic for a future tutorial.

The `OldValuesParameterFormatString` property indicates the name of the input parameters in the underlying object's update and delete methods for the original values. We'll discuss this property and its purpose in greater detail when we explore optimistic concurrency. I bring it up now, however, because our BLL's methods do not expect the original values and therefore it's important that we remove this property. Leaving the

`OldValuesParameterFormatString` property set to anything other than the default (`{0}`) will cause an error when a data Web control attempts to invoke the ObjectDataSource's `Update()` or `Delete()` methods because the ObjectDataSource will attempt to pass in both the `UpdateParameters` or `DeleteParameters` specified as well as original value parameters.

If this isn't terribly clear at this juncture, don't worry, we'll examine this property and its utility in a future tutorial. For now, just be certain to either remove this property declaration entirely from the declarative syntax or set the value to the default value ({0}).

> **NOTE**
> If you simply clear out the `OldValuesParameterFormatString` property value from the Properties window in the Design view, the property will still exist in the declarative syntax, but be set to an empty string. This, unfortunately, will still result in the same problem discussed above. Therefore, either remove the property altogether from the declarative syntax or, from the Properties window, set the value to the default, `{0}`.

Step 3: Adding a Data Web Control and Configuring It for Data Modification

Once the ObjectDataSource has been added to the page and configured, we're ready to add data Web controls to the page to both display the data and provide a means for the end user to modify it. We'll look at the GridView, DetailsView, and FormView separately, as these data Web controls differ in their data modification capabilities and configuration.

As we'll see in the remainder of this article, adding very basic editing, inserting, and deleting support through the GridView, DetailsView, and FormView controls is really as simple as checking a couple of checkboxes. There are many subtleties and edge cases in the real-world that make providing such functionality more involved than just point and click. This tutorial, however, focuses solely on proving simplistic data modification capabilities. Future tutorials will examine concerns that will undoubtedly arise in a real-world setting.

Deleting Data from the GridView

Start by dragging a GridView from the Toolbox onto the Designer. Next, bind the ObjectDataSource to the GridView by selecting it from the drop-down list in the GridView's smart tag. At this point the GridView's declarative markup will be:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="ProductID"
            InsertVisible="False"
            ReadOnly="True" SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName" HeaderText="ProductName"
            SortExpression="ProductName" />
        <asp:BoundField DataField="SupplierID" HeaderText="SupplierID"
           SortExpression="SupplierID" />
        <asp:BoundField DataField="CategoryID" HeaderText="CategoryID"
           SortExpression="CategoryID" />
        <asp:BoundField DataField="QuantityPerUnit"
           HeaderText="QuantityPerUnit"
           SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
           SortExpression="UnitPrice" />
        <asp:BoundField DataField="UnitsInStock"
           HeaderText="UnitsInStock" SortExpression="UnitsInStock" />
        <asp:BoundField DataField="UnitsOnOrder"
           HeaderText="UnitsOnOrder" SortExpression="UnitsOnOrder" />
        <asp:BoundField DataField="ReorderLevel"
           HeaderText="ReorderLevel" SortExpression="ReorderLevel" />
        <asp:CheckBoxField DataField="Discontinued"
           HeaderText="Discontinued" SortExpression="Discontinued" />
        <asp:BoundField DataField="CategoryName"
           HeaderText="CategoryName" ReadOnly="True"
            SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName"
            HeaderText="SupplierName" ReadOnly="True"
            SortExpression="SupplierName" />
    </Columns>
</asp:GridView>
```

Binding the GridView to the ObjectDataSource through its smart tag has two benefits:

- BoundFields and CheckBoxFields are automatically created for each of the fields returned by the ObjectDataSource. Moreover, the BoundField and CheckBoxField's properties are set based on the underlying field's metadata. For example, the `ProductID`, `CategoryName`, and `SupplierName` fields are marked as read-only in the `ProductsDataTable` and therefore shouldn't be updatable when editing. To accommodate this, these BoundFields' ReadOnly properties are set to `true`.
- The DataKeyNames property is assigned to the primary key field(s) of the underlying object. This is essential when using the GridView for editing or deleting data, as this property indicates the field (or set of fields) that unique identifies each record. For more information on the `DataKeyNames` property, refer back to the Master/Detail Using a Selectable Master GridView with a Details DetailView tutorial.

While the GridView can be bound to the ObjectDataSource through the Properties window or declarative syntax, doing so requires you to manually add the appropriate BoundField and `DataKeyNames` markup.

The GridView control provides built-in support for row-level editing and deleting. Configuring a GridView to support deleting adds a column of Delete buttons. When the end user clicks the Delete button for a particular row, a postback ensues and the GridView performs the following steps:

1. The ObjectDataSource's `DeleteParameters` value(s) are assigned
2. The ObjectDataSource's `Delete()` method is invoked, deleting the specified record
3. The GridView rebinds itself to the ObjectDataSource by invoking its `Select()` method

The values assigned to the `DeleteParameters` are the values of the `DataKeyNames` field(s) for the row whose Delete button was clicked. Therefore it's vital that a GridView's `DataKeyNames` property be correctly set. If it's missing, the `DeleteParameters` will be assigned a `null` value in Step 1, which in turn will not result in any

deleted records in Step 2.

> **NOTE**
>
> The `DataKeys` collection is stored in the GridView s control state, meaning that the `DataKeys` values will be remembered across postback even if the GridView s view state has been disabled. However, it is very important that view state remains enabled for GridViews that support editing or deleting (the default behavior). If you set the GridView s `EnableViewState` property to `false`, the editing and deleting behavior will work fine for a single user, but if there are concurrent users deleting data, there exists the possibility that these concurrent users may accidentally delete or edit records that they didn't intend. See my blog entry, WARNING: Concurrency Issue with ASP.NET 2.0 GridViews/DetailsView/FormViews that Support Editing and/or Deleting and Whose View State is Disabled, for more information.

This same warning also applies to DetailsViews and FormViews.

To add deleting capabilities to a GridView, simply go to its smart tag and check the Enable Deleting checkbox.

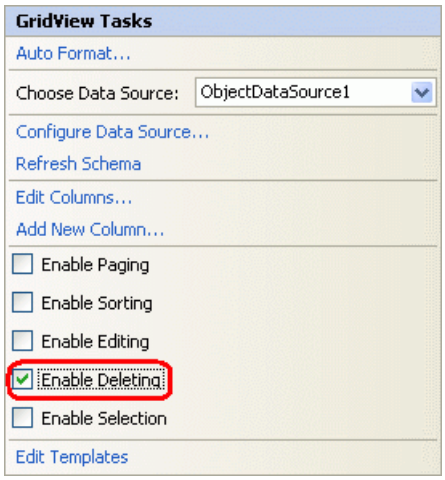


Figure 10: Check the Enable Deleting Checkbox

Checking the Enable Deleting checkbox from the smart tag adds a CommandField to the GridView. The CommandField renders a column in the GridView with buttons for performing one or more of the following tasks: selecting a record, editing a record, and deleting a record. We previously saw the CommandField in action with selecting records in the Master/Detail Using a Selectable Master GridView with a Details DetailView tutorial.

The CommandField contains a number of `ShowXButton` properties that indicate what series of buttons are displayed in the CommandField. By checking the Enable Deleting checkbox a CommandField whose `ShowDeleteButton` property is `true` has been added to the GridView's Columns collection.

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:CommandField ShowDeleteButton="True" />
        ... BoundFields removed for brevity ...
    </Columns>
</asp:GridView>
```

At this point, believe it or not, we're done with adding deleting support to the GridView! As Figure 11 shows, when visiting this page through a browser a column of Delete buttons is present.

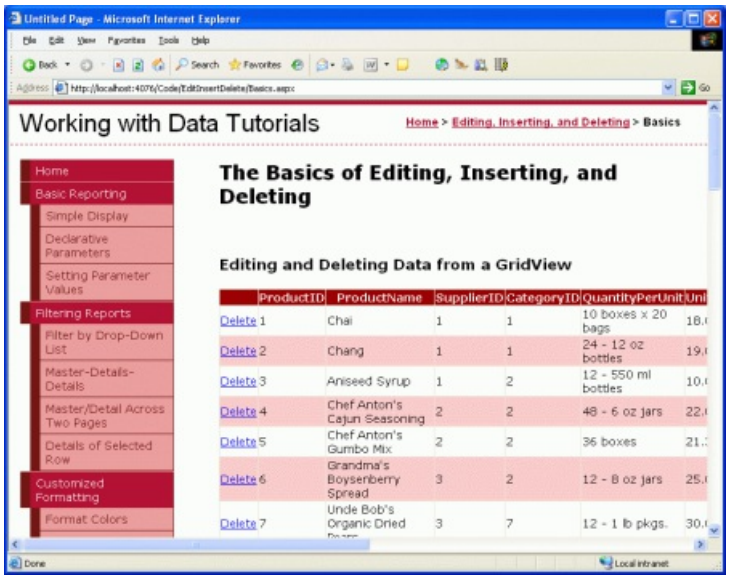


Figure 11: The CommandField Adds a Column of Delete Buttons (Click to view full-size image)

If you've been building this tutorial from the ground up on your own, when testing this page clicking the Delete button will raise an exception. Continue reading to learn as to why these exceptions were raised and how to fix them.

> **NOTE**
>
> If you're following along using the download accompanying this tutorial, these problems have already been accounted for. However, I encourage you to read through the details listed below to help identify problems that may arise and suitable workarounds.

If, when attempting to delete a product, you get an exception whose message is similar to "*ObjectDataSource 'ObjectDataSource1' could not find a non-generic method 'DeleteProduct' that has parameters: productID, original_ProductID*," you likely forgot to remove the `OldValuesParameterFormatString` property from the ObjectDataSource. With the `OldValuesParameterFormatString` property specified, the ObjectDataSource attempts to pass in both `productID` and `original_ProductID` input parameters to the `DeleteProduct` method. `DeleteProduct`, however, only accepts a single input parameter, hence the exception. Removing the `OldValuesParameterFormatString` property (or setting it to `{0}`) instructs the ObjectDataSource to not attempt to pass in the original input parameter.

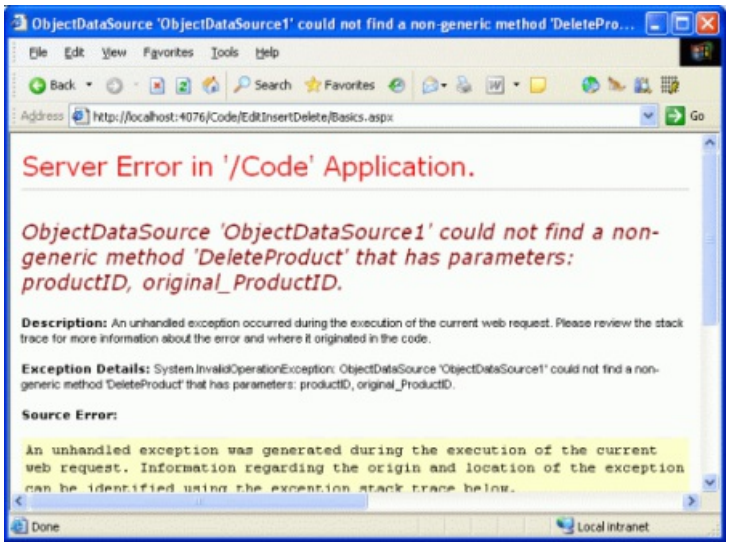


Figure 12: Ensure That the `OldValuesParameterFormatString` Property Has Been Cleared Out (Click to view full-size image)

Even if you had removed the `OldValuesParameterFormatString` property, you still will get an exception when trying to delete a product with the message: "*The DELETE statement conflicted with the REFERENCE constraint 'FK_Order_Details_Products'.*" The Northwind database contains a foreign key constraint between the `Order Details` and `Products` table, meaning that a product cannot be deleted from the system if there are one or more records for it in the `Order Details` table. Since every product in the Northwind database has at least one record in `Order Details`, we cannot delete any products until we first delete the product's associated order details records.

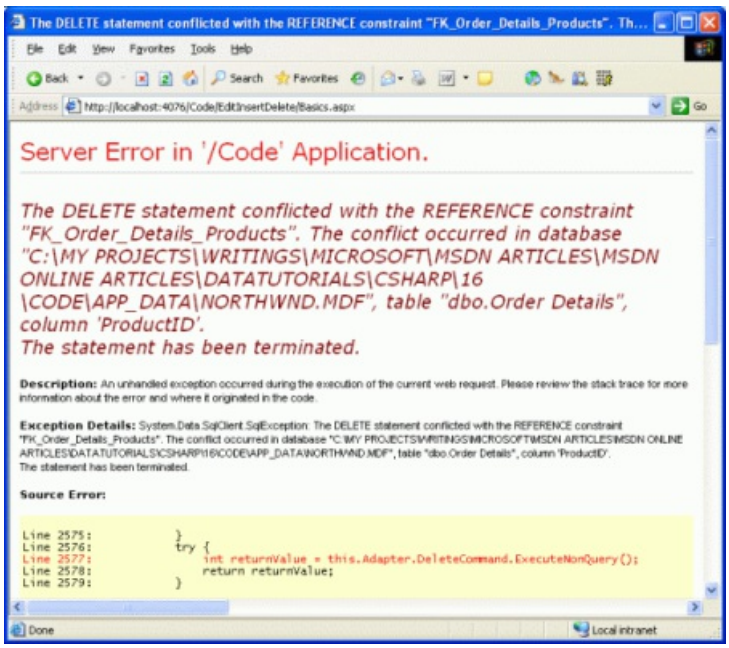


Figure 13: A Foreign Key Constraint Prohibits the Deletion of Products (Click to view full-size image)

For our tutorial, let's just delete all of the records from the `Order Details` table. In a real-world application we'd need to either:

- Have another screen to manage order details information
- Augment the `DeleteProduct` method to include logic to delete the specified product's order details
- Modify the SQL query used by the TableAdapter to include deletion of the specified product's order details

Let's just delete all of the records from the `Order Details` table to circumvent the foreign key constraint. Go to the Server Explorer in Visual Studio, right-click on the `NORTHWND.MDF` node, and choose New Query. Then, in the query window, run the following SQL statement: `DELETE FROM [Order Details]`

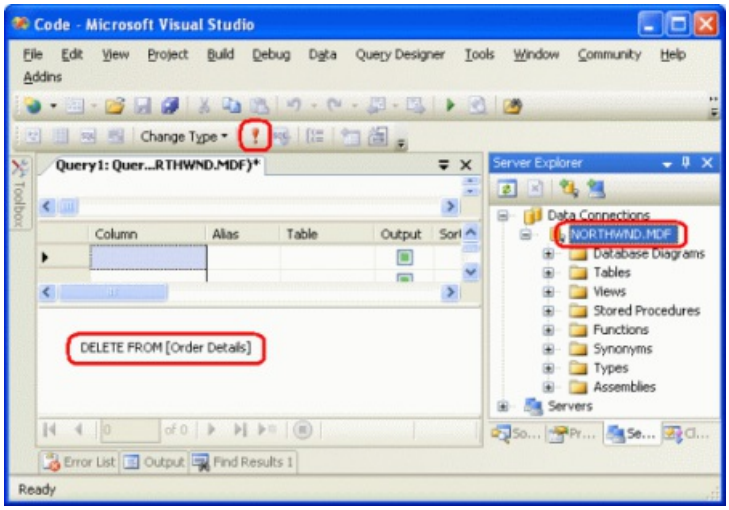


Figure 14: Delete All Records from the `Order Details` Table (Click to view full-size image)

After clearing out the `Order Details` table clicking on the Delete button will delete the product without error. If

clicking on the Delete button does not delete the product, check to ensure that the GridView's `DataKeyNames` property is set to the primary key field (`ProductID`).

> **NOTE**
>
> When clicking on the Delete button a postback ensues and the record is deleted. This can be dangerous since it is easy to accidentally click on the wrong row's Delete button. In a future tutorial we'll see how to add a client-side confirmation when deleting a record.

Editing Data with the GridView

Along with deleting, the GridView control also provides built-in row-level editing support. Configuring a GridView to support editing adds a column of Edit buttons. From the end user's perspective, clicking a row's Edit button causes that row to become editable, turning the cells into textboxes containing the existing values and replacing the Edit button with Update and Cancel buttons. After making their desired changes, the end user can click the Update button to commit the changes or the Cancel button to discard them. In either case, after clicking Update or Cancel the GridView returns to its pre-editing state.

From our perspective as the page developer, when the end user clicks the Edit button for a particular row, a postback ensues and the GridView performs the following steps:

1. The GridView's `EditItemIndex` property is assigned to the index of the row whose Edit button was clicked
2. The GridView rebinds itself to the ObjectDataSource by invoking its `Select()` method
3. The row index that matches the `EditItemIndex` is rendered in "edit mode." In this mode, the Edit button is replaced by Update and Cancel buttons and BoundFields whose `ReadOnly` properties are False (the default) are rendered as TextBox Web controls whose `Text` properties are assigned to the data fields' values.

At this point the markup is returned to the browser, allowing the end user to make any changes to the row's data. When the user clicks the Update button, a postback occurs and the GridView performs the following steps:

1. The ObjectDataSource's `UpdateParameters` value(s) are assigned the values entered by the end user into the GridView's editing interface
2. The ObjectDataSource's `Update()` method is invoked, updating the specified record
3. The GridView rebinds itself to the ObjectDataSource by invoking its `Select()` method

The primary key values assigned to the `UpdateParameters` in Step 1 come from the values specified in the `DataKeyNames` property, whereas the non-primary key values come from the text in the TextBox Web controls for the edited row. As with deleting, it is vital that a GridView's `DataKeyNames` property be correctly set. If it's missing, the `UpdateParameters` primary key value will be assigned a `null` value in Step 1, which in turn will not result in any updated records in Step 2.

Editing functionality can be activated by simply checking the Enable Editing checkbox in the GridView's smart tag.

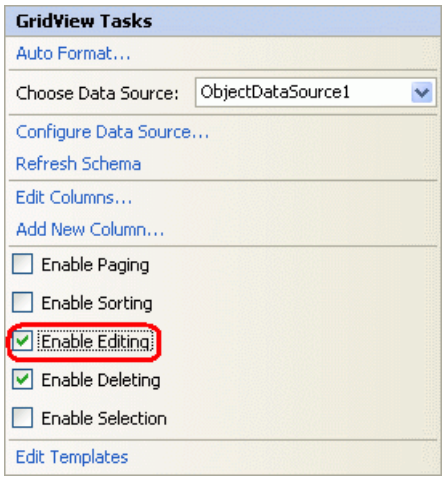


Figure 15: Check the Enable Editing Checkbox

Checking the Enable Editing checkbox will add a CommandField (if needed) and set its `ShowEditButton` property to `true`. As we saw earlier, the CommandField contains a number of `ShowXButton` properties that indicate what series of buttons are displayed in the CommandField. Checking the Enable Editing checkbox adds the `ShowEditButton` property to the existing CommandField:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:CommandField ShowDeleteButton="True"
            ShowEditButton="True" />
        ... BoundFields removed for brevity ...
    </Columns>
</asp:GridView>
```

That's all there is to adding rudimentary editing support. As Figure16 shows, the editing interface is rather crude each BoundField whose `ReadOnly` property is set to `false` (the default) is rendered as a TextBox. This includes fields like `CategoryID` and `SupplierID`, which are keys to other tables.

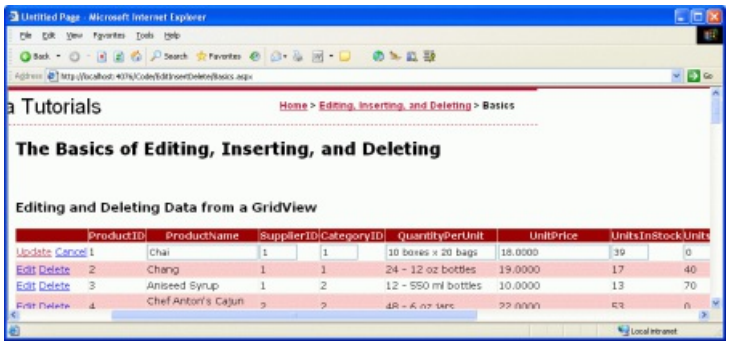


Figure 16: Clicking Chai s Edit Button Displays the Row in Edit Mode (Click to view full-size image)

In addition to asking users to edit foreign key values directly, the editing interface's interface is lacking in the following ways:

- If the user enters a `CategoryID` or `SupplierID` that does not exist in the database, the `UPDATE` will violate a foreign key constraint, causing an exception to be raised.
- The editing interface doesn't include any validation. If you don't provide a required value (such as `ProductName`), or enter a string value where a numeric value is expected (such as entering "Too much!" into the `UnitPrice` textbox), an exception will be thrown. A future tutorial will examine how to add validation controls to the editing user interface.
- Currently, *all* product fields that are not read-only must be included in the GridView. If we were to remove a field from the GridView, say `UnitPrice`, when updating the data the GridView would not set the `UnitPrice`

`UpdateParameters` value, which would change the database record's `UnitPrice` to a `NULL` value. Similarly, if a required field, such as `ProductName`, is removed from the GridView, the update will fail with the same "*Column 'ProductName' does not allow nulls*" exception mentioned above.

- The editing interface formatting leaves a lot to be desired. The `UnitPrice` is shown with four decimal points. Ideally the `CategoryID` and `SupplierID` values would contain DropDownLists that list the categories and suppliers in the system.

These are all shortcomings that we'll have to live with for now, but will be addressed in future tutorials.

Inserting, Editing, and Deleting Data with the DetailsView

As we've seen in earlier tutorials, the DetailsView control displays one record at a time and, like the GridView, allows for editing and deleting of the currently displayed record. Both the end user's experience with editing and deleting items from a DetailsView and the workflow from the ASP.NET side is identical to that of the GridView. Where the DetailsView differs from the GridView is that it also provides built-in inserting support.

To demonstrate the data modification capabilities of the GridView, start by adding a DetailsView to the `Basics.aspx` page above the existing GridView and bind it to the existing ObjectDataSource through the DetailsView's smart tag. Next, clear out the DetailsView's `Height` and `Width` properties, and check the Enable Paging option from the smart tag. To enable editing, inserting, and deleting support, simply check the Enable Editing, Enable Inserting, and Enable Deleting checkboxes in the smart tag.

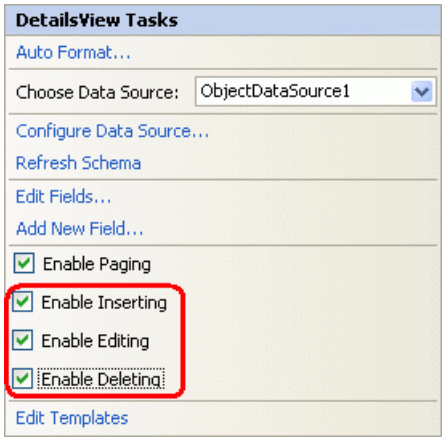


Figure 17: Configure the DetailsView to Support Editing, Inserting, and Deleting

As with the GridView, adding editing, inserting, or deleting support adds a CommandField to the DetailsView, as the following declarative syntax shows:

```
<asp:DetailsView ID="DetailsView1" runat="server" AutoGenerateRows="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1" AllowPaging="True">
    <Fields>
        <asp:BoundField DataField="ProductID"
            HeaderText="ProductID" InsertVisible="False"
            ReadOnly="True" SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName"
            HeaderText="ProductName" SortExpression="ProductName" />
        <asp:BoundField DataField="SupplierID" HeaderText="SupplierID"
            SortExpression="SupplierID" />
        <asp:BoundField DataField="CategoryID" HeaderText="CategoryID"
            SortExpression="CategoryID" />
        <asp:BoundField DataField="QuantityPerUnit"
            HeaderText="QuantityPerUnit"
            SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice"
            HeaderText="UnitPrice" SortExpression="UnitPrice" />
        <asp:BoundField DataField="UnitsInStock"
            HeaderText="UnitsInStock" SortExpression="UnitsInStock" />
        <asp:BoundField DataField="UnitsOnOrder"
            HeaderText="UnitsOnOrder" SortExpression="UnitsOnOrder" />
        <asp:BoundField DataField="ReorderLevel"
            HeaderText="ReorderLevel" SortExpression="ReorderLevel" />
        <asp:CheckBoxField DataField="Discontinued"
            HeaderText="Discontinued" SortExpression="Discontinued" />
        <asp:BoundField DataField="CategoryName"
            HeaderText="CategoryName" ReadOnly="True"
            SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName"
            HeaderText="SupplierName" ReadOnly="True"
            SortExpression="SupplierName" />
        <asp:CommandField ShowDeleteButton="True"
            ShowEditButton="True" ShowInsertButton="True" />
    </Fields>
</asp:DetailsView>
```

Note that for the DetailsView the CommandField appears at the end of the Columns collection by default. Since the DetailsView's fields are rendered as rows, the CommandField appears as a row with Insert, Edit, and Delete buttons at the bottom of the DetailsView.

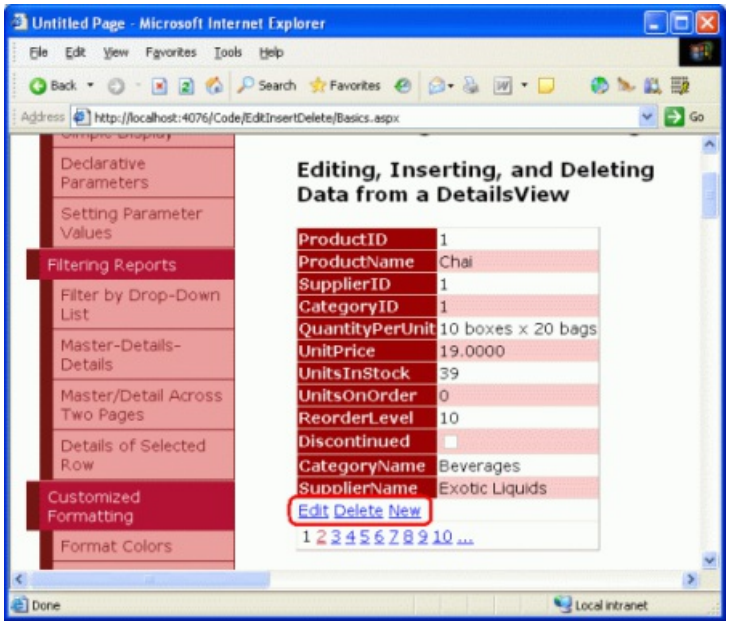


Figure 18: Configure the DetailsView to Support Editing, Inserting, and Deleting (Click to view full-size image)

Clicking on the Delete button starts the same sequence of events as with the GridView: a postback; followed by the DetailsView populating its ObjectDataSource's `DeleteParameters` based on the `DataKeyNames` values; and

completed with a call its ObjectDataSource's `Delete()` method, which actually removes the product from the database. Editing in the DetailsView also works in a fashion identical to that of the GridView.

For inserting, the end user is presented with a New button that, when clicked, renders the DetailsView in "insert mode." With "insert mode" the New button is replaced by Insert and Cancel buttons and only those BoundFields whose `InsertVisible` property is set to `true` (the default) are displayed. Those data fields identified as auto-increment fields, such as `ProductID`, have their InsertVisible property set to `false` when binding the DetailsView to the data source through the smart tag.

When binding a data source to a DetailsView through the smart tag, Visual Studio sets the `InsertVisible` property to `false` only for auto-increment fields. Read-only fields, like `CategoryName` and `SupplierName`, will be displayed in the "insert mode" user interface unless their `InsertVisible` property is explicitly set to `false`. Take a moment to set these two fields' `InsertVisible` properties to `false`, either through the DetailsView's declarative syntax or through the Edit Fields link in the smart tag. Figure 19 shows setting the `InsertVisible` properties to `false` by clicking on the Edit Fields link.

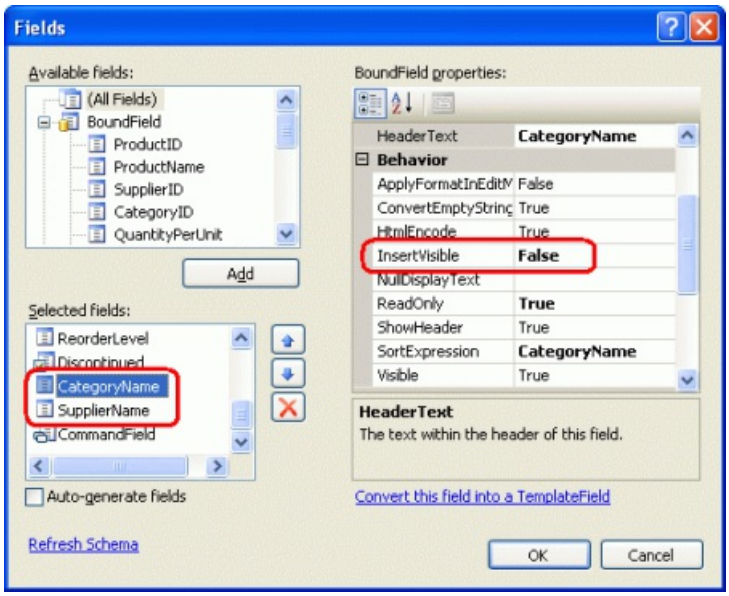


Figure 19: Northwind Traders Now Offers Acme Tea (Click to view full-size image)

After setting the `InsertVisible` properties, view the `Basics.aspx` page in a browser and click the New button. Figure 20 shows the DetailsView when adding a new beverage, Acme Tea, to our product line.

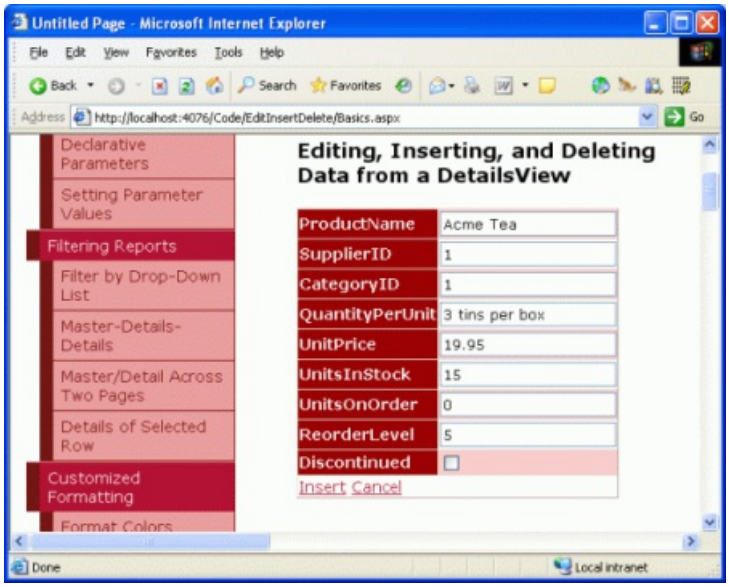


Figure 20: Northwind Traders Now Offers Acme Tea (Click to view full-size image)

After entering the details for Acme Tea and clicking the Insert button, a postback ensues and the new record is

added to the `Products` database table. Since this DetailsView lists the products in order with which they exist in the database table, we must page to the last product in order to see the new product.

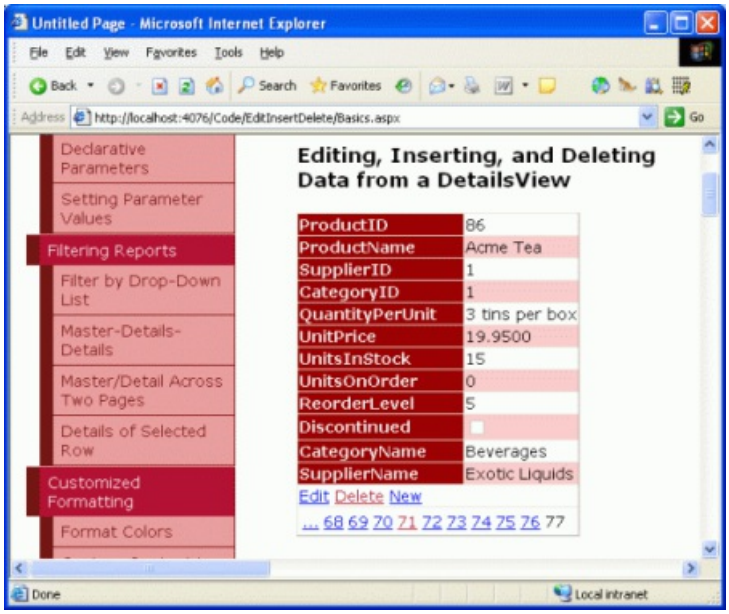


Figure 21: Details for Acme Tea (Click to view full-size image)

> **NOTE**
>
> The DetailsView's CurrentMode property indicates the interface being displayed and can be one of the following values: `Edit`, `Insert`, or `ReadOnly`. The DefaultMode property indicates the mode the DetailsView returns to after an edit or insert has been completed and is useful for displaying a DetailsView that is permanently in edit or insert mode.

The point and click inserting and editing capabilities of the DetailsView suffer from the same limitations as the GridView: the user must enter existing `CategoryID` and `SupplierID` values through a textbox; the interface lacks any validation logic; all product fields that do not allow `NULL` values or don't have a default value specified at the database level must be included in the inserting interface, and so on.

The techniques we will examine for extending and enhancing the GridView's editing interface in future articles can be applied to the DetailsView control's editing and inserting interfaces as well.

Using the FormView for a More Flexible Data Modification User Interface

The FormView offers built-in support for inserting, editing, and deleting data, but because it uses templates instead of fields there's no place to add the BoundFields or the CommandField used by the GridView and DetailsView controls to provide the data modification interface. Instead, this interface the Web controls for collecting user input when adding a new item or editing an existing one along with the New, Edit, Delete, Insert, Update, and Cancel buttons must be added manually to the appropriate templates. Fortunately, Visual Studio will automatically create the needed interface when binding the FormView to a data source through the drop-down list in its smart tag.

To illustrate these techniques, start by adding a FormView to the `Basics.aspx` page and, from the FormView's smart tag, bind it to the ObjectDataSource already created. This will generate an `EditItemTemplate`, `InsertItemTemplate`, and `ItemTemplate` for the FormView with TextBox Web controls for collecting the user's input and Button Web controls for the New, Edit, Delete, Insert, Update, and Cancel buttons. Additionally, the FormView's `DataKeyNames` property is set to the primary key field (`ProductID`) of the object returned by the ObjectDataSource. Lastly, check the Enable Paging option in the FormView's smart tag.

The following shows the declarative markup for the FormView's `ItemTemplate` after the FormView has been bound to the ObjectDataSource. By default, each non-Boolean value product field is bound to the `Text` property of a Label Web control while each Boolean value field (`Discontinued`) is bound to the `Checked` property of a disabled CheckBox Web control. In order for the New, Edit, and Delete buttons to trigger certain FormView behavior when clicked, it is imperative that their `CommandName` values be set to `New`, `Edit`, and `Delete`, respectively.

```
<asp:FormView ID="FormView1" runat="server" DataKeyNames="ProductID"
    DataSourceID="ObjectDataSource1" AllowPaging="True">
    <EditItemTemplate>
        ...
    </EditItemTemplate>
    <InsertItemTemplate>
        ...
    </InsertItemTemplate>
    <ItemTemplate>
        ProductID:
        <asp:Label ID="ProductIDLabel" runat="server"
            Text='<%# Eval("ProductID") %>'></asp:Label><br />
        ProductName:
        <asp:Label ID="ProductNameLabel" runat="server"
            Text='<%# Bind("ProductName") %>'>
        </asp:Label><br />
        SupplierID:
        <asp:Label ID="SupplierIDLabel" runat="server"
            Text='<%# Bind("SupplierID") %>'>
        </asp:Label><br />
        CategoryID:
        <asp:Label ID="CategoryIDLabel" runat="server"
            Text='<%# Bind("CategoryID") %>'>
        </asp:Label><br />
        QuantityPerUnit:
        <asp:Label ID="QuantityPerUnitLabel" runat="server"
            Text='<%# Bind("QuantityPerUnit") %>'>
        </asp:Label><br />
        UnitPrice:
        <asp:Label ID="UnitPriceLabel" runat="server"
            Text='<%# Bind("UnitPrice") %>'></asp:Label><br />
        UnitsInStock:
        <asp:Label ID="UnitsInStockLabel" runat="server"
            Text='<%# Bind("UnitsInStock") %>'>
        </asp:Label><br />
        UnitsOnOrder:
        <asp:Label ID="UnitsOnOrderLabel" runat="server"
            Text='<%# Bind("UnitsOnOrder") %>'>
        </asp:Label><br />
        ReorderLevel:
        <asp:Label ID="ReorderLevelLabel" runat="server"
            Text='<%# Bind("ReorderLevel") %>'>
        </asp:Label><br />
        Discontinued:
        <asp:CheckBox ID="DiscontinuedCheckBox" runat="server"
            Checked='<%# Bind("Discontinued") %>'
            Enabled="false" /><br />
        CategoryName:
        <asp:Label ID="CategoryNameLabel" runat="server"
            Text='<%# Bind("CategoryName") %>'>
        </asp:Label><br />
        SupplierName:
        <asp:Label ID="SupplierNameLabel" runat="server"
            Text='<%# Bind("SupplierName") %>'>
        </asp:Label><br />
        <asp:LinkButton ID="EditButton" runat="server"
            CausesValidation="False" CommandName="Edit"
            Text="Edit">
        </asp:LinkButton>
```

```
        <asp:LinkButton ID="DeleteButton" runat="server"
            CausesValidation="False" CommandName="Delete"
            Text="Delete">
        </asp:LinkButton>
        <asp:LinkButton ID="NewButton" runat="server"
            CausesValidation="False" CommandName="New"
            Text="New">
        </asp:LinkButton>
    </ItemTemplate>
</asp:FormView>
```

Figure 22 shows the FormView's `ItemTemplate` when viewed through a browser. Each product field is listed with the New, Edit, and Delete buttons at the bottom.

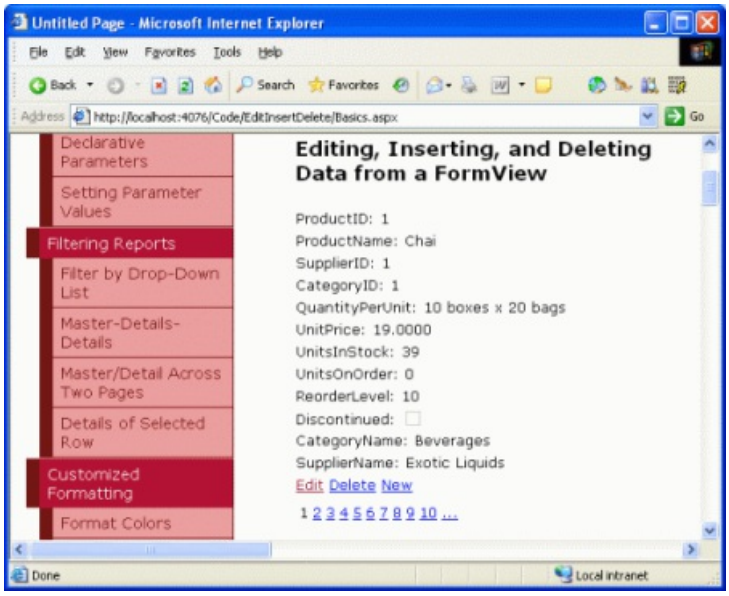


Figure 22: The Defaut FormView `ItemTemplate` Lists Each Product Field Along with New, Edit, and Delete Buttons (Click to view full-size image)

Like with the GridView and DetailsView, clicking the Delete button or any Button, LinkButton, or ImageButton whose `CommandName` property is set to Delete causes a postback, populates the ObjectDataSource's `DeleteParameters` based on the FormView's `DataKeyNames` value, and invokes the ObjectDataSource's `Delete()` method.

When the Edit button is clicked a postback ensues and the data is rebound to the `EditItemTemplate`, which is responsible for rendering the editing interface. This interface includes the Web controls for editing data along with the Update and Cancel buttons. The default `EditItemTemplate` generated by Visual Studio contains a Label for any auto-increment fields (`ProductID`), a TextBox for each non-Boolean value field, and a CheckBox for each Boolean value field. This behavior is very similar to the auto-generated BoundFields in the GridView and DetailsView controls.

> **NOTE**
> One small issue with the FormView's auto-generation of the `EditItemTemplate` is that it renders TextBox Web controls for those fields that are read-only, such as `CategoryName` and `SupplierName`. We'll see how to account for this shortly.

The TextBox controls in the `EditItemTemplate` have their `Text` property bound to the value of their corresponding data field using *two-way databinding*. Two-way databinding, denoted by `<%# Bind("dataField") %>`, performs databinding both when binding data to the template and when populating the ObjectDataSource's parameters for inserting or editing records. That is, when the user clicks the Edit button from the `ItemTemplate`, the `Bind()` method returns the specified data field value. After the user makes their

changes and clicks Update, the values posted back that correspond to the data fields specified using `Bind()` are applied to the ObjectDataSource's `UpdateParameters`. Alternatively, one-way databinding, denoted by `<%# Eval("dataField") %>`, only retrieves the data field values when binding data to the template and does *not* return the user-entered values to the data source's parameters on postback.

The following declarative markup shows the FormView's `EditItemTemplate`. Note that the `Bind()` method is used in the databinding syntax here and that the Update and Cancel Button Web controls have their `CommandName` properties set accordingly.

```
<asp:FormView ID="FormView1" runat="server" DataKeyNames="ProductID"
    DataSourceID="ObjectDataSource1" AllowPaging="True">
    <EditItemTemplate>
        ProductID:
        <asp:Label ID="ProductIDLabel1" runat="server"
          Text="<%# Eval("ProductID") %>"></asp:Label><br />
        ProductName:
        <asp:TextBox ID="ProductNameTextBox" runat="server"
          Text="<%# Bind("ProductName") %>">
        </asp:TextBox><br />
        SupplierID:
        <asp:TextBox ID="SupplierIDTextBox" runat="server"
          Text="<%# Bind("SupplierID") %>">
        </asp:TextBox><br />
        CategoryID:
        <asp:TextBox ID="CategoryIDTextBox" runat="server"
          Text="<%# Bind("CategoryID") %>">
        </asp:TextBox><br />
        QuantityPerUnit:
        <asp:TextBox ID="QuantityPerUnitTextBox" runat="server"
           Text="<%# Bind("QuantityPerUnit") %>">
        </asp:TextBox><br />
        UnitPrice:
        <asp:TextBox ID="UnitPriceTextBox" runat="server"
           Text="<%# Bind("UnitPrice") %>">
        </asp:TextBox><br />
        UnitsInStock:
        <asp:TextBox ID="UnitsInStockTextBox" runat="server"
           Text="<%# Bind("UnitsInStock") %>">
        </asp:TextBox><br />
        UnitsOnOrder:
        <asp:TextBox ID="UnitsOnOrderTextBox" runat="server"
           Text="<%# Bind("UnitsOnOrder") %>">
        </asp:TextBox><br />
        ReorderLevel:
        <asp:TextBox ID="ReorderLevelTextBox" runat="server"
           Text="<%# Bind("ReorderLevel") %>">
        </asp:TextBox><br />
        Discontinued:
        <asp:CheckBox ID="DiscontinuedCheckBox" runat="server"
            Checked="<%# Bind("Discontinued") %>" /><br />
        CategoryName:
        <asp:TextBox ID="CategoryNameTextBox" runat="server"
             Text="<%# Bind("CategoryName") %>">
        </asp:TextBox><br />
        SupplierName:
        <asp:TextBox ID="SupplierNameTextBox" runat="server"
             Text="<%# Bind("SupplierName") %>">
        </asp:TextBox><br />
        <asp:LinkButton ID="UpdateButton" runat="server"
            CausesValidation="True" CommandName="Update"
            Text="Update">
        </asp:LinkButton>
        <asp:LinkButton ID="UpdateCancelButton" runat="server"
            CausesValidation="False" CommandName="Cancel"
            Text="Cancel">
        </asp:LinkButton>
    </EditItemTemplate>
    <InsertItemTemplate>
        ...
    </InsertItemTemplate>
    <ItemTemplate>
        ...
    </ItemTemplate>
</asp:FormView>
```

Our `EditItemTemplate`, at this point, will cause an exception to be thrown if we attempt to use it. The problem is that the `CategoryName` and `SupplierName` fields are rendered as TextBox Web controls in the `EditItemTemplate`. We either need to change these TextBoxes to Labels or remove them altogether. Let's simply delete them entirely from the `EditItemTemplate`.

Figure 23 shows the FormView in a browser after the Edit button has been clicked for Chai. Note that the `SupplierName` and `CategoryName` fields shown in the `ItemTemplate` are no longer present, as we just removed them from the `EditItemTemplate`. When the Update button is clicked the FormView proceeds through the same sequence of steps as the GridView and DetailsView controls.

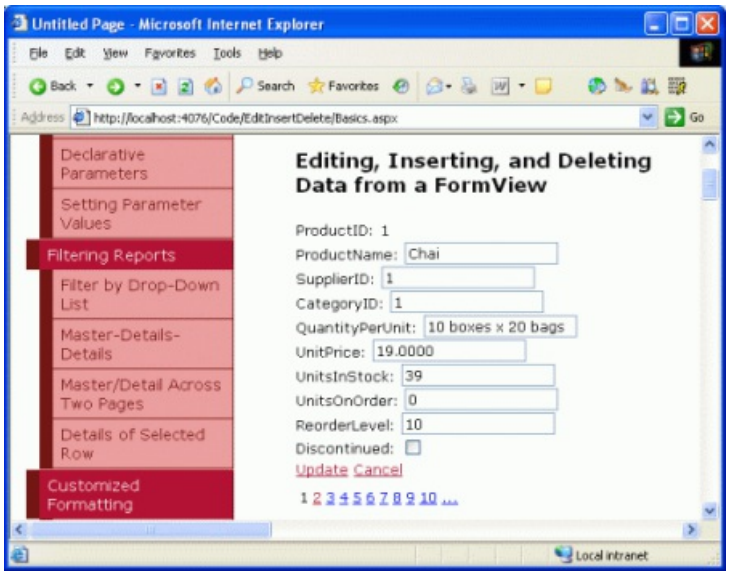


Figure 23: By Default the `EditItemTemplate` Shows Each Editable Product Field as a TextBox or CheckBox (Click to view full-size image)

When the Insert button is clicked the FormView's `ItemTemplate` a postback ensues. However, no data is bound to the FormView because a new record is being added. The `InsertItemTemplate` interface includes the Web controls for adding a new record along with the Insert and Cancel buttons. The default `InsertItemTemplate` generated by Visual Studio contains a TextBox for each non-Boolean value field and a CheckBox for each Boolean value field, similar to the auto-generated `EditItemTemplate`'s interface. The TextBox controls have their `Text` property bound to the value of their corresponding data field using two-way databinding.

The following declarative markup shows the FormView's `InsertItemTemplate`. Note that the `Bind()` method is used in the databinding syntax here and that the Insert and Cancel Button Web controls have their `CommandName` properties set accordingly.

```
<asp:FormView ID="FormView1" runat="server" DataKeyNames="ProductID"
    DataSourceID="ObjectDataSource1" AllowPaging="True">
    <EditItemTemplate>
        ...
    </EditItemTemplate>
    <InsertItemTemplate>
        ProductName:
        <asp:TextBox ID="ProductNameTextBox" runat="server"
           Text="<%# Bind("ProductName") %>">
        </asp:TextBox><br />
        SupplierID:
        <asp:TextBox ID="SupplierIDTextBox" runat="server"
           Text="<%# Bind("SupplierID") %>">
        </asp:TextBox><br />
        CategoryID:
        <asp:TextBox ID="CategoryIDTextBox" runat="server"
           Text="<%# Bind("CategoryID") %>">
        </asp:TextBox><br />
        QuantityPerUnit:
        <asp:TextBox ID="QuantityPerUnitTextBox" runat="server"
           Text="<%# Bind("QuantityPerUnit") %>">
        </asp:TextBox><br />
        UnitPrice:
        <asp:TextBox ID="UnitPriceTextBox" runat="server"
           Text="<%# Bind("UnitPrice") %>">
        </asp:TextBox><br />
        UnitsInStock:
        <asp:TextBox ID="UnitsInStockTextBox" runat="server"
           Text="<%# Bind("UnitsInStock") %>">
        </asp:TextBox><br />
        UnitsOnOrder:
        <asp:TextBox ID="UnitsOnOrderTextBox" runat="server"
           Text="<%# Bind("UnitsOnOrder") %>">
        </asp:TextBox><br />
        ReorderLevel:
        <asp:TextBox ID="ReorderLevelTextBox" runat="server"
           Text="<%# Bind("ReorderLevel") %>">
        </asp:TextBox><br />
        Discontinued:
        <asp:CheckBox ID="DiscontinuedCheckBox" runat="server"
           Checked="<%# Bind("Discontinued") %>" /><br />
        CategoryName:
        <asp:TextBox ID="CategoryNameTextBox" runat="server"
            Text="<%# Bind("CategoryName") %>">
        </asp:TextBox><br />
        SupplierName:
        <asp:TextBox ID="SupplierNameTextBox" runat="server"
           Text="<%# Bind("SupplierName") %>">
        </asp:TextBox><br />
        <asp:LinkButton ID="InsertButton" runat="server"
            CausesValidation="True" CommandName="Insert"
            Text="Insert">
        </asp:LinkButton>
        <asp:LinkButton ID="InsertCancelButton" runat="server"
            CausesValidation="False" CommandName="Cancel"
            Text="Cancel">
        </asp:LinkButton>
    </InsertItemTemplate>
    <ItemTemplate>
        ...
    </ItemTemplate>
</asp:FormView>
```

There's a subtlety with the FormView's auto-generation of the `InsertItemTemplate`. Specifically, the TextBox Web controls are created even for those fields that are read-only, such as `CategoryName` and `SupplierName`. Like with the `EditItemTemplate`, we need to remove these TextBoxes from the `InsertItemTemplate`.

Figure 24 shows the FormView in a browser when adding a new product, Acme Coffee. Note that the `SupplierName` and `CategoryName` fields shown in the `ItemTemplate` are no longer present, as we just removed them. When the Insert button is clicked the FormView proceeds through the same sequence of steps as the DetailsView control, adding a new record to the `Products` table. Figure 25 shows Acme Coffee product's details in the FormView after it has been inserted.

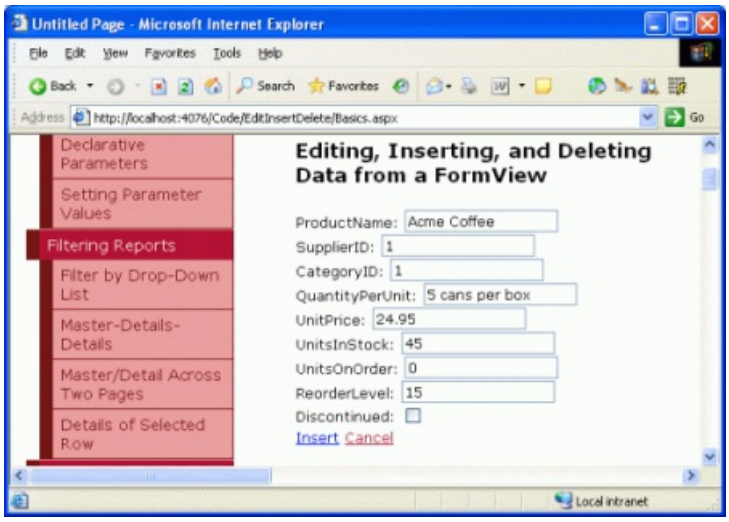


Figure 24: The `InsertItemTemplate` Dictates the FormView's Inserting Interface (Click to view full-size image)

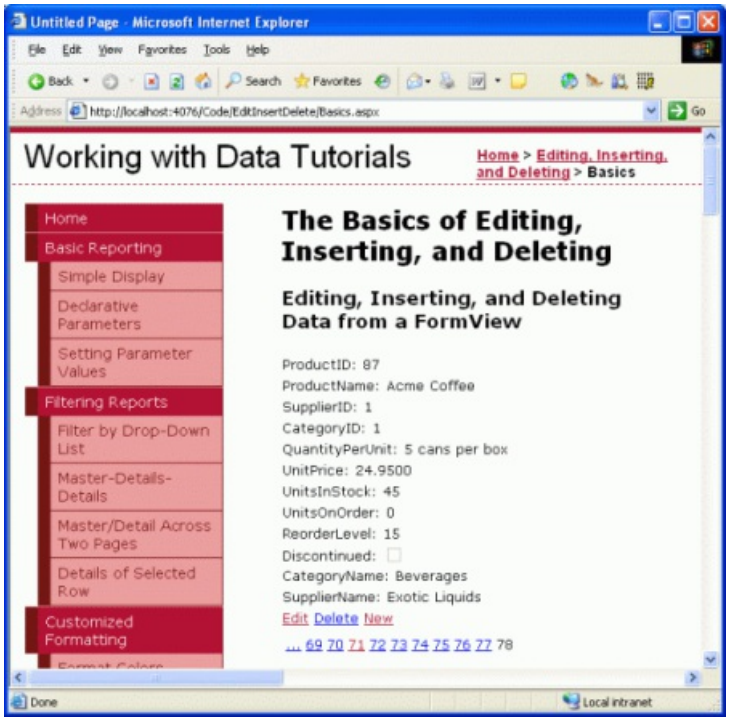


Figure 25: The Details for New Product, Acme Coffee, are Displayed in the FormView (Click to view full-size image)

By separating out the read-only, editing, and inserting interfaces into three separate templates, the FormView allows for a finer degree of control over these interfaces than the DetailsView and GridView.

> **NOTE**
>
> Like the DetailsView, the FormView's `CurrentMode` property indicates the interface being displayed and its `DefaultMode` property indicates the mode the FormView returns to after an edit or insert has been completed.

Summary

In this tutorial we examined the basics of inserting, editing, and deleting data using the GridView, DetailsView, and FormView. All three of these controls provide some level of built-in data modification capabilities that can be utilized without writing a single line of code in the ASP.NET page thanks to the data Web controls and the ObjectDataSource. However, the simple point and click techniques render a fairly frail and naïve data modification user interface. To provide validation, inject programmatic values, gracefully handle exceptions, customize the user interface, and so on, we'll need to rely on a bevy of techniques that will be discussed over the next several tutorials.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

NEXT

Examining the Events Associated with Inserting, Updating, and Deleting (C#)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll examine using the events that occur before, during, and after an insert, update, or delete operation of an ASP.NET data Web control. We'll also see how to customize the editing interface to only update a subset of the product fields.

Introduction

When using the built-in inserting, editing, or deleting features of the GridView, DetailsView, or FormView controls, a variety of steps transpire when the end user completes the process of adding a new record or updating or deleting an existing record. As we discussed in the previous tutorial, when a row is edited in the GridView the Edit button is replaced by Update and Cancel buttons and the BoundFields turn into TextBoxes. After the end user updates the data and clicks Update, the following steps are performed on postback:

1. The GridView populates its ObjectDataSource's `UpdateParameters` with the edited record's unique identifying field(s) (via the `DataKeyNames` property) along with the values entered by the user
2. The GridView invokes its ObjectDataSource's `Update()` method, which in turn invokes the appropriate method in the underlying object (`ProductsDAL.UpdateProduct`, in our previous tutorial)
3. The underlying data, which now includes the updated changes, is rebound to the GridView

During this sequence of steps, a number of events fire, enabling us to create event handlers to add custom logic where needed. For example, prior to Step 1, the GridView's `RowUpdating` event fires. We can, at this point, cancel the update request if there is some validation error. When the `Update()` method is invoked, the ObjectDataSource's `Updating` event fires, providing an opportunity to add or customize the values of any of the `UpdateParameters`. After the ObjectDataSource's underlying object's method has completed executing, the ObjectDataSource's `Updated` event is raised. An event handler for the `Updated` event can inspect the details about the update operation, such as how many rows were affected and whether or not an exception occurred. Finally, after Step 2, the GridView's `RowUpdated` event fires; an event handler for this event can examine additional information about the update operation just performed.

Figure 1 depicts this series of events and steps when updating a GridView. The event pattern in Figure 1 is not unique to updating with a GridView. Inserting, updating, or deleting data from the GridView, DetailsView, or FormView precipitates the same sequence of pre- and post-level events for both the data Web control and the ObjectDataSource.

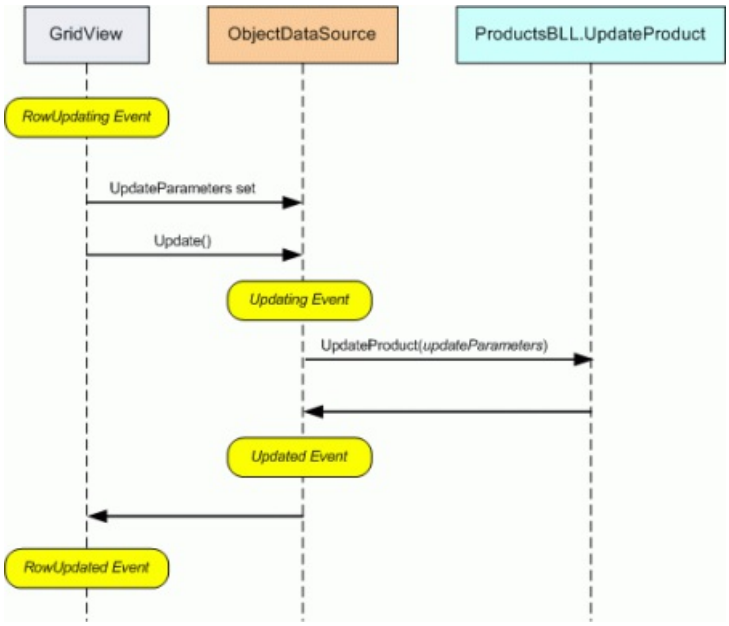


Figure 1: A Series of Pre- and Post-Events Fire When Updating Data in a GridView (Click to view full-size image)

In this tutorial we'll examine using these events to extend the built-in inserting, updating, and deleting capabilities of the ASP.NET data Web controls. We'll also see how to customize the editing interface to only update a subset of the product fields.

Step 1: Updating a Product's `ProductName` and `UnitPrice` Fields

In the editing interfaces from the previous tutorial *all* product fields that were not read-only had to be included. If we were to remove a field from the GridView - say `QuantityPerUnit` - when updating the data the data Web control would not set the ObjectDataSource's `QuantityPerUnit` `UpdateParameters` value. The ObjectDataSource would then pass in a `null` value into the `UpdateProduct` Business Logic Layer (BLL) method, which would change the edited database record's `QuantityPerUnit` column to a `NULL` value. Similarly, if a required field, such as `ProductName`, is removed from the editing interface, the update will fail with a "*Column 'ProductName' does not allow nulls*" exception. The reason for this behavior was because the ObjectDataSource was configured to call the `ProductsBLL` class's `UpdateProduct` method, which expected an input parameter for each of the product fields. Therefore, the ObjectDataSource's `UpdateParameters` collection contained a parameter for each of the method's input parameters.

If we want to provide a data Web control that allows the end user to only update a subset of fields, then we need to either programmatically set the missing `UpdateParameters` values in the ObjectDataSource's `Updating` event handler or create and call a BLL method that expects only a subset of the fields. Let's explore this latter approach.

Specifically, let's create a page that displays just the `ProductName` and `UnitPrice` fields in an editable GridView. This GridView's editing interface will only allow the user to update the two displayed fields, `ProductName` and `UnitPrice`. Since this editing interface only provides a subset of a product's fields, we either need to create an ObjectDataSource that uses the existing BLL's `UpdateProduct` method and has the missing product field values set programmatically in its `Updating` event handler, or we need to create a new BLL method that expects only the subset of fields defined in the GridView. For this tutorial, let's use the latter option and create an overload of the `UpdateProduct` method, one that takes in just three input parameters: `productName`, `unitPrice`, and `productID`:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Update, false)]
public bool UpdateProduct(string productName, decimal? unitPrice, int productID)
{
    Northwind.ProductsDataTable products = Adapter.GetProductByProductID(productID);
    if (products.Count == 0)
        // no matching record found, return false
        return false;

    Northwind.ProductsRow product = products[0];

    product.ProductName = productName;
    if (unitPrice == null) product.SetUnitPriceNull();
      else product.UnitPrice = unitPrice.Value;

    // Update the product record
    int rowsAffected = Adapter.Update(product);

    // Return true if precisely one row was updated, otherwise false
    return rowsAffected == 1;
}
```

Like the original `UpdateProduct` method, this overload starts by checking to see if there is a product in the database with the specified `ProductID`. If not, it returns `false`, indicating that the request to update the product information failed. Otherwise it updates the existing product record's `ProductName` and `UnitPrice` fields accordingly and commits the update by calling the TableAdapter's `Update()` method, passing in the `ProductsRow` instance.

With this addition to our `ProductsBLL` class, we're ready to create the simplified GridView interface. Open the `DataModificationEvents.aspx` in the `EditInsertDelete` folder and add a GridView to the page. Create a new ObjectDataSource and configure it to use the `ProductsBLL` class with its `Select()` method mapping to `GetProducts` and its `Update()` method mapping to the `UpdateProduct` overload that takes in only the `productName`, `unitPrice`, and `productID` input parameters. Figure 2 shows the Create Data Source wizard when mapping the ObjectDataSource's `Update()` method to the `ProductsBLL` class's new `UpdateProduct` method overload.

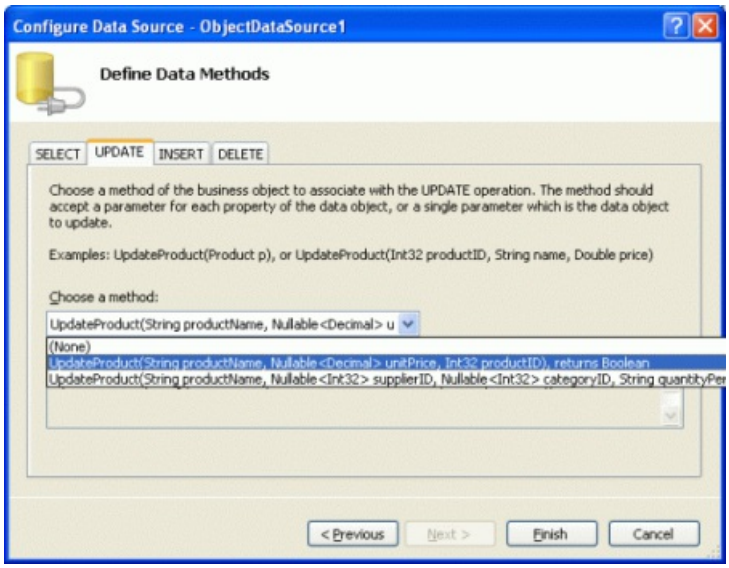


Figure 2: Map the ObjectDataSource's `Update()` Method to the New `UpdateProduct` Overload (Click to view full-size image)

Since our example will initially just need the ability to edit data, but not to insert or delete records, take a moment to explicitly indicate that the ObjectDataSource's `Insert()` and `Delete()` methods shouldn't be mapped to any of the `ProductsBLL` class's methods by going to the INSERT and DELETE tabs and choosing

(None) from the drop-down list.

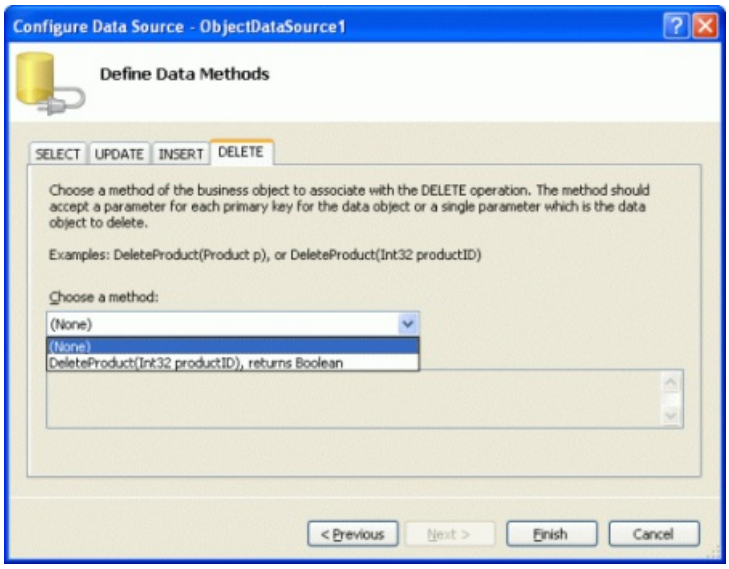


Figure 3: Choose (None) From the Drop-Down List for the INSERT and DELETE Tabs (Click to view full-size image)

After completing this wizard check the Enable Editing checkbox from the GridView's smart tag.

With the completion of the Create Data Source wizard and binding that to the GridView, Visual Studio has created the declarative syntax for both controls. Go to the Source view to inspect the ObjectDataSource's declarative markup, which is shown below:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}" SelectMethod="GetProducts"
    TypeName="ProductsBLL" UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

Since there are no mappings for the ObjectDataSource's `Insert()` and `Delete()` methods, there are no `InsertParameters` or `DeleteParameters` sections. Furthermore, since the `Update()` method is mapped to the `UpdateProduct` method overload that only accepts three input parameters, the `UpdateParameters` section has just three `Parameter` instances.

Note that the ObjectDataSource's `OldValuesParameterFormatString` property is set to `original_{0}`. This property is set automatically by Visual Studio when using the Configure Data Source wizard. However, since our BLL methods don't expect the original `ProductID` value to be passed in, remove this property assignment altogether from the ObjectDataSource's declarative syntax.

> **NOTE**
>
> If you simply clear out the `OldValuesParameterFormatString` property value from the Properties window in the Design view, the property will still exist in the declarative syntax, but will be set to an empty string. Either remove the property altogether from the declarative syntax or, from the Properties window, set the value to the default, `{0}`.

While the ObjectDataSource only has `UpdateParameters` for the product's name, price, and ID, Visual Studio has added a BoundField or CheckBoxField in the GridView for each of the product's fields.

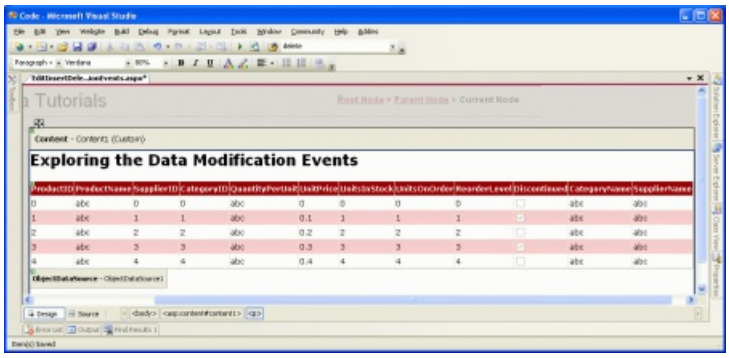


Figure 4: The GridView Contains a BoundField or CheckBoxField for Each of the Product's Fields (Click to view full-size image)

When the end user edits a product and clicks its Update button, the GridView enumerates those fields that were not read-only. It then sets the value of the corresponding parameter in the ObjectDataSource's `UpdateParameters` collection to the value entered by the user. If there is not a corresponding parameter, the GridView adds one to the collection. Therefore, if our GridView contains BoundFields and CheckBoxFields for all of the product's fields, the ObjectDataSource will end up invoking the `UpdateProduct` overload that takes in all of these parameters, despite the fact that the ObjectDataSource's declarative markup specifies only three input parameters (see Figure 5). Similarly, if there is some combination of non-read-only product fields in the GridView that doesn't correspond to the input parameters for a `UpdateProduct` overload, an exception will be raised when attempting to update.

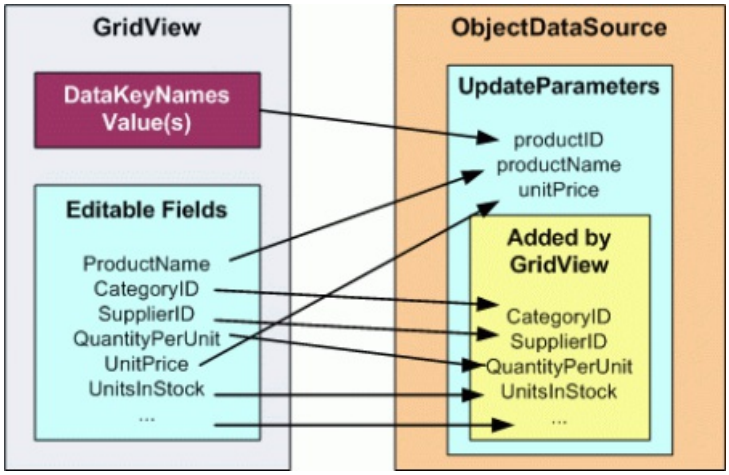


Figure 5: The GridView Will Add Parameters to the ObjectDataSource's `UpdateParameters` Collection (Click to view full-size image)

To ensure that the ObjectDataSource invokes the `UpdateProduct` overload that takes in just the product's name, price, and ID, we need to restrict the GridView to having editable fields for just the `ProductName` and `UnitPrice`. This can be accomplished by removing the other BoundFields and CheckBoxFields, by setting those other fields' `ReadOnly` property to `true`, or by some combination of the two. For this tutorial let's simply remove all GridView fields except the `ProductName` and `UnitPrice` BoundFields, after which the GridView's declarative markup will look like:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:BoundField DataField="ProductName"
          HeaderText="ProductName" SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
          SortExpression="UnitPrice" />
    </Columns>
</asp:GridView>
```

Even though the `UpdateProduct` overload expects three input parameters, we only have two BoundFields in our GridView. This is because the `productID` input parameter is a primary key value and passed in through the value of the `DataKeyNames` property for the edited row.

Our GridView, along with the `UpdateProduct` overload, allows a user to edit just the name and price of a product without losing any of the other product fields.

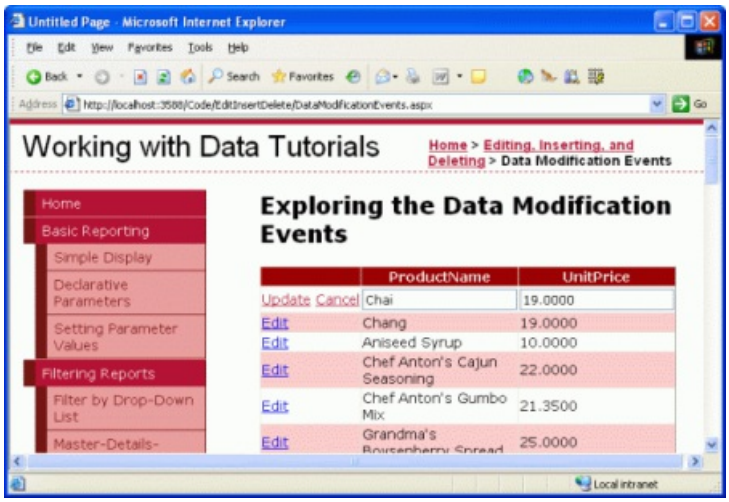


Figure 6: The Interface Allows Editing Just the Product's Name and Price (Click to view full-size image)

> **NOTE**
>
> As discussed in the previous tutorial, it is vitally important that the GridView s view state be enabled (the default behavior). If you set the GridView s `EnableViewState` property to `false`, you run the risk of having concurrent users unintentionally deleting or editing records. See WARNING: Concurrency Issue with ASP.NET 2.0 GridViews/DetailsView/FormViews that Support Editing and/or Deleting and Whose View State is Disabled for more information.

Improving the `UnitPrice` Formatting

While the GridView example shown in Figure 6 works, the `UnitPrice` field is not formatted at all, resulting in a price display that lacks any currency symbols and has four decimal places. To apply a currency formatting for the non-editable rows, simply set the `UnitPrice` BoundField's `DataFormatString` property to `{0:c}` and its `HtmlEncode` property to `false`.

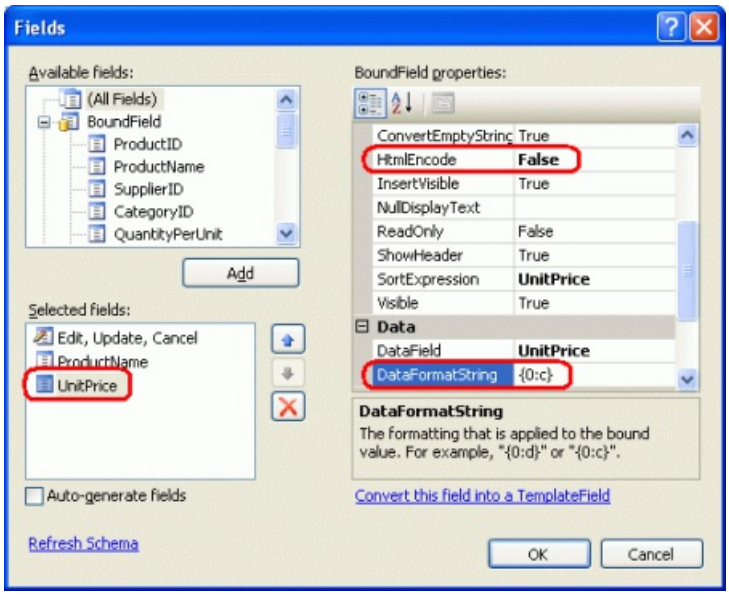


Figure 7: Set the `UnitPrice`'s `DataFormatString` and `HtmlEncode` Properties Accordingly (Click to view full-size image)

With this change, the non-editable rows format the price as a currency; the edited row, however, still displays the value without the currency symbol and with four decimal places.

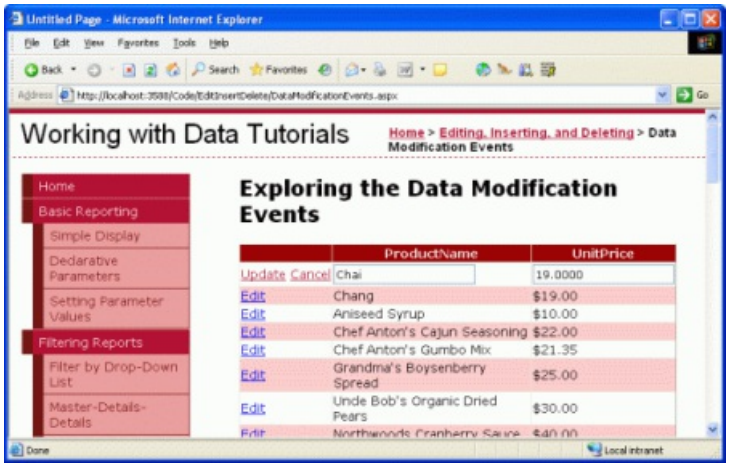


Figure 8: Non-Editable Rows are Now Formatted as Currency Values (Click to view full-size image)

The formatting instructions specified in the `DataFormatString` property can be applied to the editing interface by setting the BoundField's `ApplyFormatInEditMode` property to `true` (the default is `false`). Take a moment to set this property to `true`.

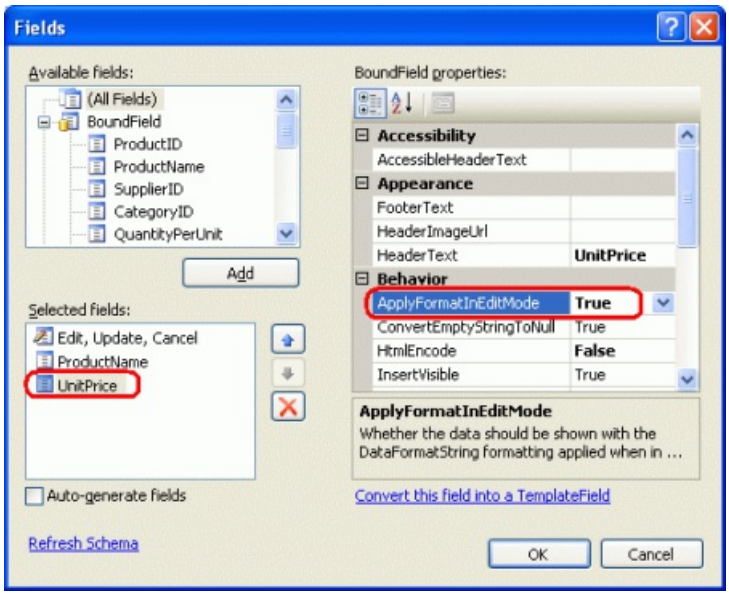


Figure 9: Set the `UnitPrice` BoundField's `ApplyFormatInEditMode` property to `true` (Click to view full-size image)

With this change, the value of the `UnitPrice` displayed in the edited row is also formatted as a currency.

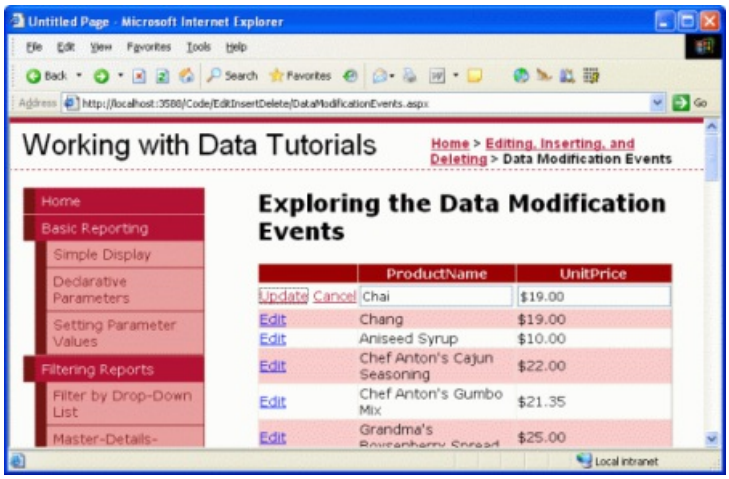


Figure 10: The Edited Row's `UnitPrice` Value is Now Formatted as a Currency (Click to view full-size image)

However, updating a product with the currency symbol in the textbox such as $19.00 throws a `FormatException`. When the GridView attempts to assign the user-supplied values to the ObjectDataSource's `UpdateParameters` collection it is unable to convert the `UnitPrice` string "$19.00" into the `decimal` required by the parameter (see Figure 11). To remedy this we can create an event handler for the GridView's `RowUpdating` event and have it parse the user-supplied `UnitPrice` as a currency-formatted `decimal`.

The GridView's `RowUpdating` event accepts as its second parameter an object of type GridViewUpdateEventArgs, which includes a `NewValues` dictionary as one of its properties that holds the user-supplied values ready to be assigned to the ObjectDataSource's `UpdateParameters` collection. We can overwrite the existing `UnitPrice` value in the `NewValues` collection with a decimal value parsed using the currency format with the following lines of code in the `RowUpdating` event handler:

```
protected void GridView1_RowUpdating(object sender, GridViewUpdateEventArgs e)
{
  if (e.NewValues["UnitPrice"] != null)
    e.NewValues["UnitPrice"] =
        decimal.Parse(e.NewValues["UnitPrice"].ToString(),
            System.Globalization.NumberStyles.Currency);
}
```

If the user has supplied a `UnitPrice` value (such as "$19.00"), this value is overwritten with the decimal value computed by Decimal.Parse, parsing the value as a currency. This will correctly parse the decimal in the event of any currency symbols, commas, decimal points, and so on, and uses the NumberStyles enumeration in the System.Globalization namespace.

Figure 11 shows both the problem caused by currency symbols in the user-supplied `UnitPrice`, along with how the GridView's `RowUpdating` event handler can be utilized to correctly parse such input.

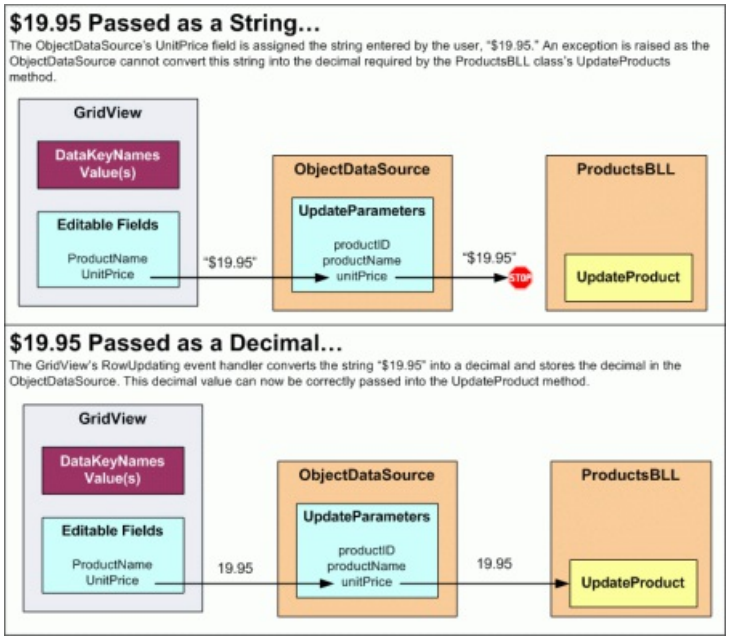


Figure 11: The Edited Row's `UnitPrice` Value is Now Formatted as a Currency (Click to view full-size image)

Step 2: Prohibiting `NULL UnitPrices`

While the database is configured to allow `NULL` values in the `Products` table's `UnitPrice` column, we may want to prevent users visiting this particular page from specifying a `NULL` `UnitPrice` value. That is, if a user fails to enter a `UnitPrice` value when editing a product row, rather than save the results to the database we want to display a message informing the user that, through this page, any edited products must have a price specified.

The `GridViewUpdateEventArgs` object passed into the GridView's `RowUpdating` event handler contains a `Cancel` property that, if set to `true`, terminates the updating process. Let's extend the `RowUpdating` event handler to set `e.Cancel` to `true` and display a message explaining why if the `UnitPrice` value in the `NewValues` collection is `null`.

Start by adding a Label Web control to the page named `MustProvideUnitPriceMessage`. This Label control will be displayed if the user fails to specify a `UnitPrice` value when updating a product. Set the Label's `Text` property to "You must provide a price for the product." I've also created a new CSS class in `Styles.css` named `Warning` with the following definition:

```
.Warning
{
    color: Red;
    font-style: italic;
    font-weight: bold;
    font-size: x-large;
}
```

Finally, set the Label's `CssClass` property to `Warning`. At this point the Designer should show the warning message in a red, bold, italic, extra large font size above the GridView, as shown in Figure 12.

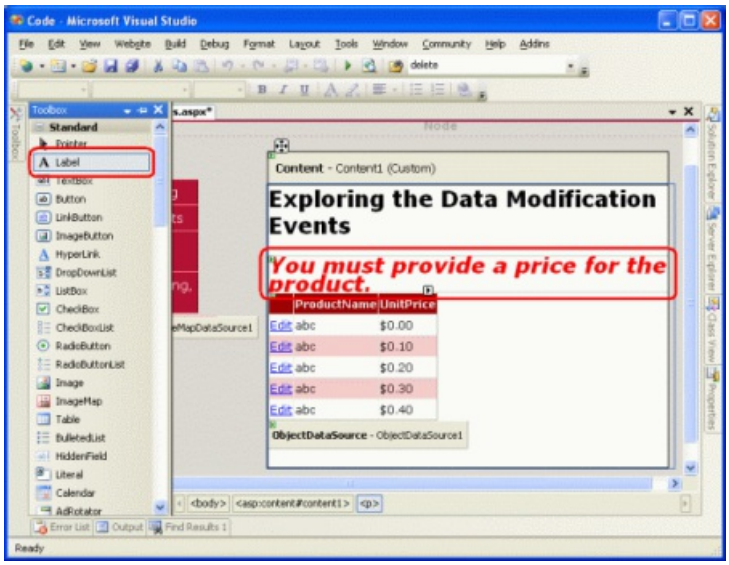


Figure 12: A Label Has Been Added Above the GridView (Click to view full-size image)

By default, this Label should be hidden, so set its `Visible` property to `false` in the `Page_Load` event handler:

```
protected void Page_Load(object sender, EventArgs e)
{
    MustProvideUnitPriceMessage.Visible = false;
}
```

If the user attempts to update a product without specifying the `UnitPrice`, we want to cancel the update and display the warning label. Augment the GridView's `RowUpdating` event handler as follows:

```
protected void GridView1_RowUpdating(object sender, GridViewUpdateEventArgs e)
{
    if (e.NewValues["UnitPrice"] != null)
    {
        e.NewValues["UnitPrice"] =
            decimal.Parse(e.NewValues["UnitPrice"].ToString(),
                System.Globalization.NumberStyles.Currency);
    }
    else
    {
        // Show the Label
        MustProvideUnitPriceMessage.Visible = true;

        // Cancel the update
        e.Cancel = true;
    }
}
```

If a user attempts to save a product without specifying a price, the update is cancelled and a helpful message is displayed. While the database (and business logic) allows for `NULL` `UnitPrice` s, this particular ASP.NET page does not.

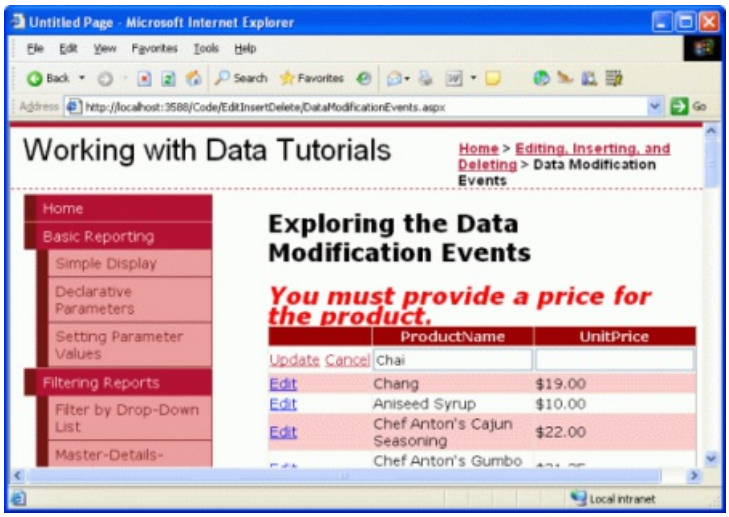


Figure 13: A User Cannot Leave `UnitPrice` Blank (Click to view full-size image)

So far we have seen how to use the GridView's `RowUpdating` event to programmatically alter the parameter values assigned to the ObjectDataSource's `UpdateParameters` collection as well as how to cancel the updating process altogether. These concepts carry over to the DetailsView and FormView controls and also apply to inserting and deleting.

These tasks can also be done at the ObjectDataSource level through event handlers for its `Inserting`, `Updating`, and `Deleting` events. These events fire before the associated method of the underlying object is invoked and provide a last-chance opportunity to modify the input parameters collection or cancel the operation outright. The event handlers for these three events are passed an object of type ObjectDataSourceMethodEventArgs that has two properties of interest:

- Cancel, which, if set to `true`, cancels the operation being performed
- InputParameters, which is the collection of `InsertParameters`, `UpdateParameters`, or `DeleteParameters`, depending on whether the event handler is for the `Inserting`, `Updating`, or `Deleting` event

To illustrate working with the parameter values at the ObjectDataSource level, let's include a DetailsView in our page that allows the users to add a new product. This DetailsView will be used to provide an interface for quickly adding a new product to the database. To keep a consistent user interface when adding a new product let's allow the user to only enter values for the `ProductName` and `UnitPrice` fields. By default, those values that aren't supplied in the DetailsView's inserting interface will be set to a `NULL` database value. However, we can use the ObjectDataSource's `Inserting` event to inject different default values, as we'll see shortly.

Step 3: Providing an Interface to Add New Products

Drag a DetailsView from the Toolbox onto the Designer above the GridView, clear out its `Height` and `Width` properties, and bind it to the ObjectDataSource already present on the page. This will add a BoundField or CheckBoxField for each of the product's fields. Since we want to use this DetailsView to add new products, we need to check the Enable Inserting option from the smart tag; however, there's no such option because the ObjectDataSource's `Insert()` method is not mapped to a method in the `ProductsBLL` class (recall that we set this mapping to (None) when configuring the data source see Figure 3).

To configure the ObjectDataSource, select the Configure Data Source link from its smart tag, launching the wizard. The first screen allows you to change the underlying object the ObjectDataSource is bound to; leave it set to `ProductsBLL`. The next screen lists the mappings from the ObjectDataSource's methods to the underlying object's. Even though we explicitly indicated that the `Insert()` and `Delete()` methods should not be mapped to any methods, if you go to the INSERT and DELETE tabs you'll see that a mapping is there. This is because the `ProductsBLL`'s `AddProduct` and `DeleteProduct` methods use the `DataObjectMethodAttribute` attribute to indicate that they are the default methods for `Insert()` and `Delete()`, respectively. Hence, the ObjectDataSource wizard

selects these each time you run the wizard unless there's some other value explicitly specified.

Leave the `Insert()` method pointing to the `AddProduct` method, but again set the DELETE tab's drop-down list to (None).

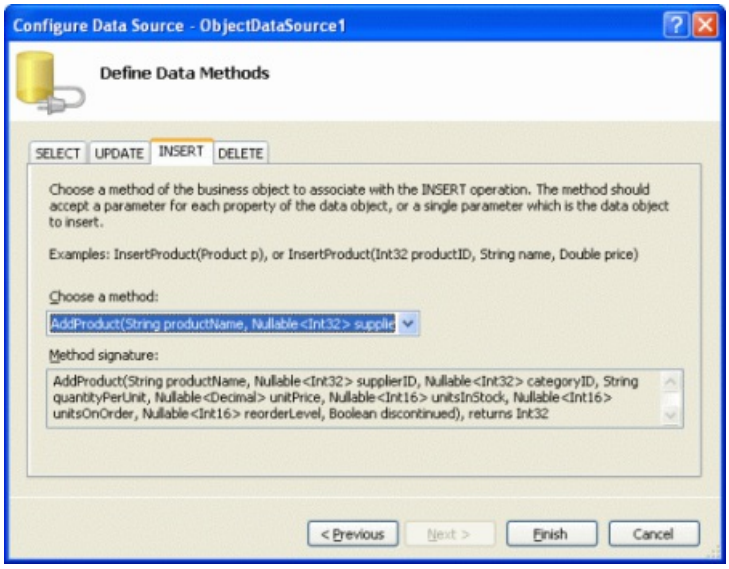


Figure 14: Set the INSERT Tab's Drop-Down List to the `AddProduct` Method (Click to view full-size image)

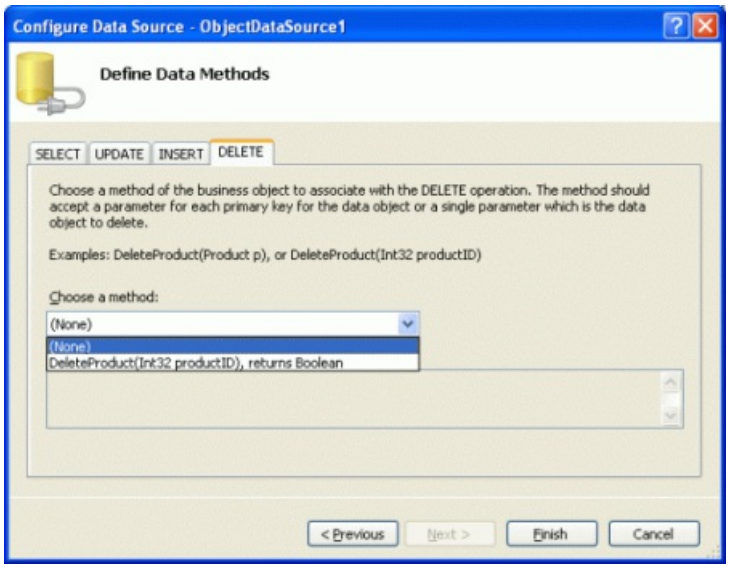


Figure 15: Set the DELETE Tab's Drop-Down List to (None) (Click to view full-size image)

After making these changes, the ObjectDataSource's declarative syntax will be expanded to include an `InsertParameters` collection, as shown below:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    SelectMethod="GetProducts" TypeName="ProductsBLL"
    UpdateMethod="UpdateProduct" OnUpdating="ObjectDataSource1_Updating"
    InsertMethod="AddProduct" OldValuesParameterFormatString="original_{0}">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
    <InsertParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="unitsInStock" Type="Int16" />
        <asp:Parameter Name="unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="reorderLevel" Type="Int16" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
    </InsertParameters>
</asp:ObjectDataSource>
```

Rerunning the wizard added back the `OldValuesParameterFormatString` property. Take a moment to clear this property by setting it to the default value (`{0}`) or removing it altogether from the declarative syntax.

With the ObjectDataSource providing inserting capabilities, the DetailsView's smart tag will now include the Enable Inserting checkbox; return to the Designer and check this option. Next, pare down the DetailsView so that it only has two BoundFields - `ProductName` and `UnitPrice` - and the CommandField. At this point the DetailsView's declarative syntax should look like:

```
<asp:DetailsView ID="DetailsView1" runat="server" AutoGenerateRows="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1">
    <Fields>
        <asp:BoundField DataField="ProductName"
          HeaderText="ProductName" SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
          SortExpression="UnitPrice" />
        <asp:CommandField ShowInsertButton="True" />
    </Fields>
</asp:DetailsView>
```

Figure 16 shows this page when viewed through a browser at this point. As you can see, the DetailsView lists the name and price of the first product (Chai). What we want, however, is an inserting interface that provides a means for the user to quickly add a new product to the database.

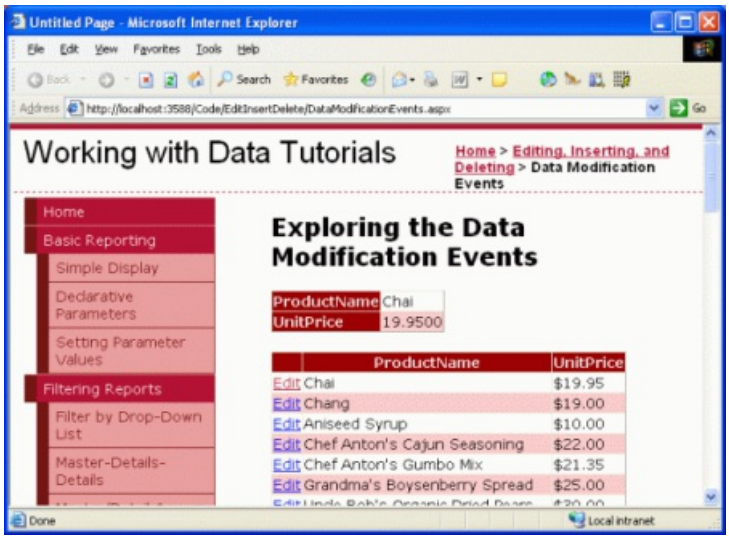


Figure 16: The DetailsView is Currently Rendered in Read-Only Mode (Click to view full-size image)

In order to show the DetailsView in its inserting mode we need to set the `DefaultMode` property to `Inserting`. This renders the DetailsView in insert mode when first visited and keeps it there after inserting a new record. As Figure 17 shows, such a DetailsView provides a quick interface for adding a new record.

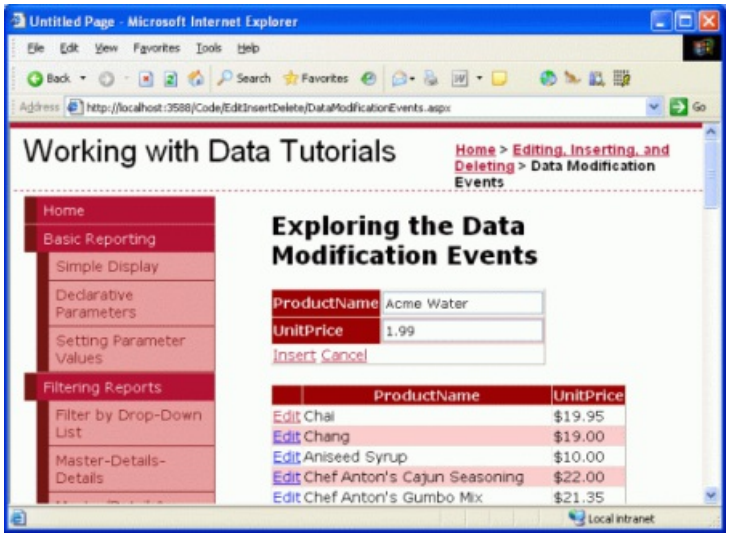


Figure 17: The DetailsView Provides an Interface for Quickly Adding a New Product (Click to view full-size image)

When the user enters a product name and price (such as "Acme Water" and 1.99, as in Figure 17) and clicks Insert, a postback ensues and the inserting workflow commences, culminating in a new product record being added to the database. The DetailsView maintains its inserting interface and the GridView is automatically rebound to its data source in order to include the new product, as shown in Figure 18.

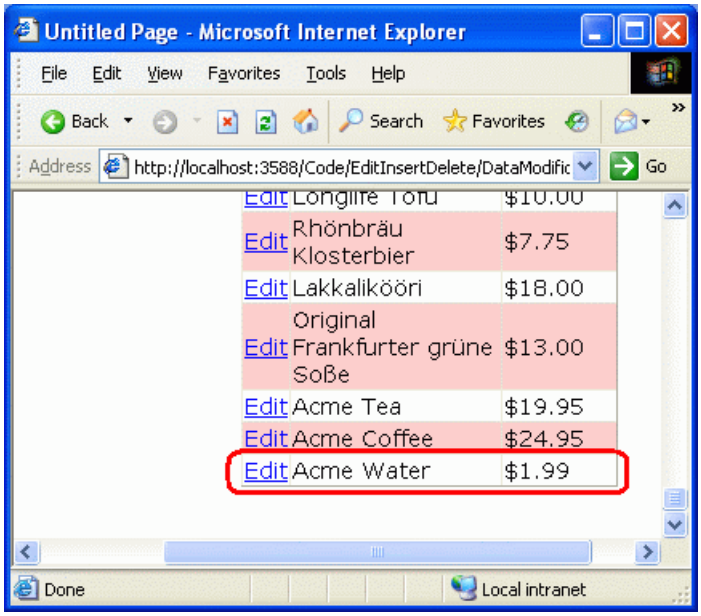


Figure 18: The Product "Acme Water" Has Been Added to the Database

While the GridView in Figure 18 doesn't show it, the product fields lacking from the DetailsView interface `CategoryID`, `SupplierID`, `QuantityPerUnit`, and so on are assigned `NULL` database values. You can see this by performing the following steps:

1. Go to the Server Explorer in Visual Studio
2. Expanding the `NORTHWND.MDF` database node
3. Right-click on the `Products` database table node
4. Select Show Table Data

This will list all of the records in the `Products` table. As Figure 19 shows, all of our new product's columns other than `ProductID`, `ProductName`, and `UnitPrice` have `NULL` values.

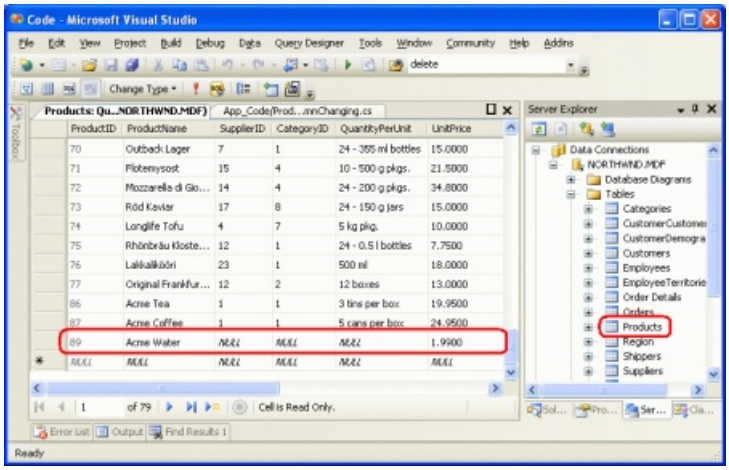


Figure 19: The Product Fields Not Provided in the DetailsView are Assigned `NULL` Values (Click to view full-size image)

We may want to provide a default value other than `NULL` for one or more of these column values, either because `NULL` isn't the best default option or because the database column itself doesn't allow `NULL` s. To accomplish this we can programmatically set the values of the parameters of the DetailsView's `InputParameters` collection. This assignment can be done either in the event handler for the DetailsView's `ItemInserting` event or the ObjectDataSource's `Inserting` event. Since we've already looked at using the pre- and post-level events at the data Web control level, let's explore using the ObjectDataSource's events this time.

Step 4: Assigning Values to the `CategoryID` and `SupplierID` Parameters

For this tutorial let's imagine that for our application when adding a new product through this interface it should be assigned a `CategoryID` and `SupplierID` value of 1. As mentioned earlier, the ObjectDataSource has a pair of pre- and post-level events that fire during the data modification process. When its `Insert()` method is invoked, the ObjectDataSource first raises its `Inserting` event, then calls the method that its `Insert()` method has been mapped to, and finally raises the `Inserted` event. The `Inserting` event handler affords us one last opportunity to tweak the input parameters or cancel the operation outright.

> **NOTE**
>
> In a real-world application you would likely want to either let the user specify the category and supplier or would pick this value for them based on some criteria or business logic (rather than blindly selecting an ID of 1). Regardless, the example illustrates how to programmatically set the value of an input parameter from the ObjectDataSource's pre-level event.

Take a moment to create an event handler for the ObjectDataSource's `Inserting` event. Notice that the event handler's second input parameter is an object of type `ObjectDataSourceMethodEventArgs`, which has a property to access the parameters collection (`InputParameters`) and a property to cancel the operation (`Cancel`).

```
protected void ObjectDataSource1_Inserting
    (object sender, ObjectDataSourceMethodEventArgs e)
{

}
```

At this point, the `InputParameters` property contains the ObjectDataSource's `InsertParameters` collection with the values assigned from the DetailsView. To change the value of one of these parameters, simply use:

`e.InputParameters["paramName"] = value`. Therefore, to set the `CategoryID` and `SupplierID` to values of 1, adjust the `Inserting` event handler to look like the following:

```
protected void ObjectDataSource1_Inserting
    (object sender, ObjectDataSourceMethodEventArgs e)
{
    e.InputParameters["CategoryID"] = 1;
    e.InputParameters["SupplierID"] = 1;
}
```

This time when adding a new product (such as Acme Soda), the `CategoryID` and `SupplierID` columns of the new product are set to 1 (see Figure 20).

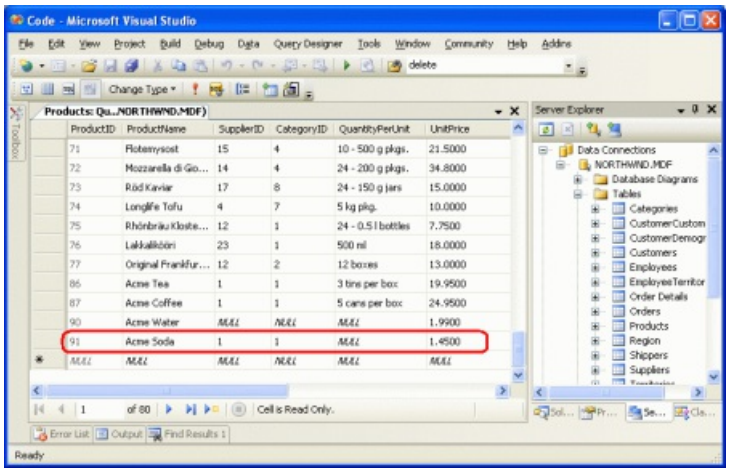


Figure 20: New Products Now Have Their `CategoryID` and `SupplierID` Values Set to 1 (Click to view full-size image)

Summary

During the editing, inserting, and deleting process, both the data Web control and the ObjectDataSource proceed through a number of pre- and post-level events. In this tutorial we examined the pre-level events and saw how to use these to customize the input parameters or cancel the data modification operation altogether both from the data Web control and ObjectDataSource's events. In the next tutorial we'll look at creating and using event handlers for the post-level events.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Jackie Goor and Liz Shulok. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Handling BLL- and DAL-Level Exceptions in an ASP.NET Page (C#)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we will see how to display a friendly, informative error message should an exception occur during an insert, update, or delete operation of an ASP.NET data Web control.

Introduction

Working with data from an ASP.NET web application using a tiered application architecture involves the following three general steps:

1. Determine what method of the Business Logic Layer needs to be invoked and what parameter values to pass it. The parameter values can be hard coded, programmatically-assigned, or inputs entered by the user.
2. Invoke the method.
3. Process the results. When calling a BLL method that returns data, this may involve binding the data to a data Web control. For BLL methods that modify data, this may include performing some action based on a return value or gracefully handling any exception that arose in Step 2.

As we saw in the previous tutorial, both the ObjectDataSource and the data Web controls provide extensibility points for Steps 1 and 3. The GridView, for example, fires its `RowUpdating` event prior to assigning its field values to its ObjectDataSource's `UpdateParameters` collection; its `RowUpdated` event is raised after the ObjectDataSource has completed the operation.

We've already examined the events that fire during Step 1 and have seen how they can be used to customize the input parameters or cancel the operation. In this tutorial we'll turn our attention to the events that fire after the operation has completed. With these post-level event handlers we can, among other things, determine if an exception occurred during the operation and handle it gracefully, displaying a friendly, informative error message on the screen rather than defaulting to the standard ASP.NET exception page.

To illustrate working with these post-level events, let's create a page that lists the products in an editable GridView. When updating a product, if an exception is raised our ASP.NET page will display a short message above the GridView explaining that a problem has occurred. Let's get started!

Step 1: Creating an Editable GridView of Products

In the previous tutorial we created an editable GridView with just two fields, `ProductName` and `UnitPrice`. This required creating an additional overload for the `ProductsBLL` class's `UpdateProduct` method, one that only accepted three input parameters (the product's name, unit price, and ID) as opposed a parameter for each product field. For this tutorial, let's practice this technique again, creating an editable GridView that displays the product's name, quantity per unit, unit price, and units in stock, but only allows the name, unit price, and units in stock to be edited.

To accommodate this scenario we'll need another overload of the `UpdateProduct` method, one that accepts four parameters: the product's name, unit price, units in stock, and ID. Add the following method to the `ProductsBLL` class:

```
[System.ComponentModel.DataObjectMethodAttribute(
    System.ComponentModel.DataObjectMethodType.Update, false)]
public bool UpdateProduct(string productName, decimal? unitPrice, short? unitsInStock,
    int productID)
{
    Northwind.ProductsDataTable products = Adapter.GetProductByProductID(productID);
    if (products.Count == 0)
        // no matching record found, return false
        return false;
    Northwind.ProductsRow product = products[0];
    product.ProductName = productName;
    if (unitPrice == null) product.SetUnitPriceNull();
      else product.UnitPrice = unitPrice.Value;
    if (unitsInStock == null) product.SetUnitsInStockNull();
      else product.UnitsInStock = unitsInStock.Value;
    // Update the product record
    int rowsAffected = Adapter.Update(product);
    // Return true if precisely one row was updated, otherwise false
    return rowsAffected == 1;
}
```

With this method complete, we're ready to create the ASP.NET page that allows for editing these four particular product fields. Open the `ErrorHandling.aspx` page in the `EditInsertDelete` folder and add a GridView to the page through the Designer. Bind the GridView to a new ObjectDataSource, mapping the `Select()` method to the `ProductsBLL` class's `GetProducts()` method and the `Update()` method to the `UpdateProduct` overload just created.

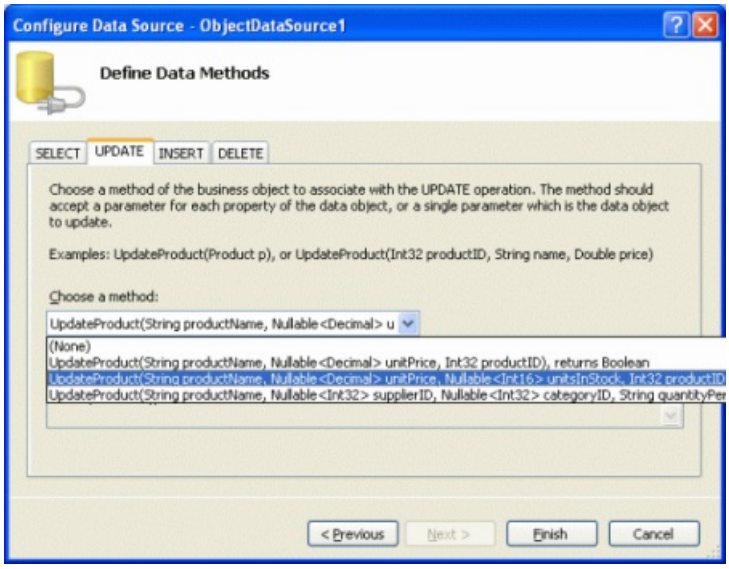


Figure 1: Use the `UpdateProduct` Method Overload That Accepts Four Input Parameters (Click to view full-size image)

This will create an ObjectDataSource with an `UpdateParameters` collection with four parameters and a GridView with a field for each of the product fields. The ObjectDataSource's declarative markup assigns the `OldValuesParameterFormatString` property the value `original_{0}`, which will cause an exception since our BLL class's don't expect an input parameter named `original_productID` to be passed in. Don't forget to remove this setting altogether from the declarative syntax (or set it to the default value, `{0}`).

Next, pare down the GridView to include only the `ProductName`, `QuantityPerUnit`, `UnitPrice`, and `UnitsInStock` BoundFields. Also feel free to apply any field-level formatting you deem necessary (such as changing the `HeaderText` properties).

In the previous tutorial we looked at how to format the `UnitPrice` BoundField as a currency both in read-only mode and edit mode. Let's do the same here. Recall that this required setting the BoundField's `DataFormatString`

property to `{0:c}`, its `HtmlEncode` property to `false`, and its `ApplyFormatInEditMode` to `true`, as shown in Figure 2.

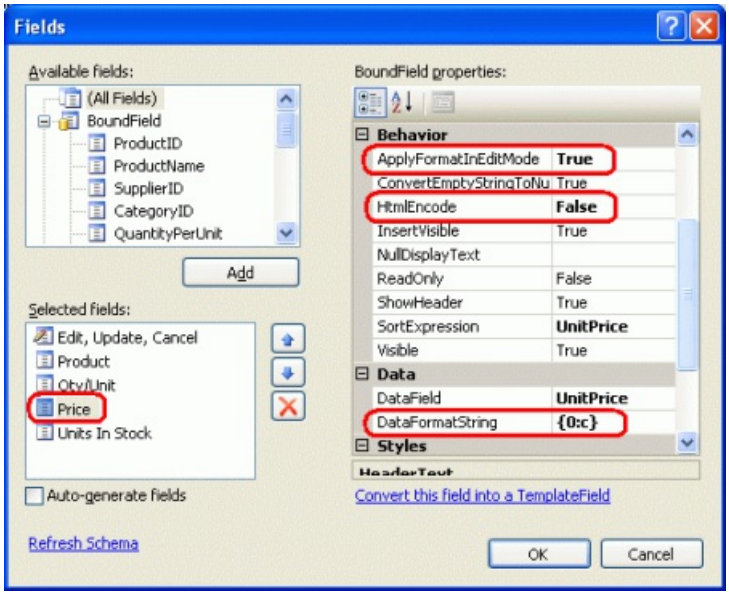


Figure 2: Configure the `UnitPrice` BoundField to Display as a Currency (Click to view full-size image)

Formatting the `UnitPrice` as a currency in the editing interface requires creating an event handler for the GridView's `RowUpdating` event that parses the currency-formatted string into a `decimal` value. Recall that the `RowUpdating` event handler from the last tutorial also checked to ensure that the user provided a `UnitPrice` value. However, for this tutorial let's allow the user to omit the price.

```
protected void GridView1_RowUpdating(object sender, GridViewUpdateEventArgs e)
{
    if (e.NewValues["UnitPrice"] != null)
        e.NewValues["UnitPrice"] =decimal.Parse(e.NewValues["UnitPrice"].ToString(),
            System.Globalization.NumberStyles.Currency);
}
```

Our GridView includes a `QuantityPerUnit` BoundField, but this BoundField should be only for display purposes and should not be editable by the user. To arrange this, simply set the BoundFields' `ReadOnly` property to `true`.

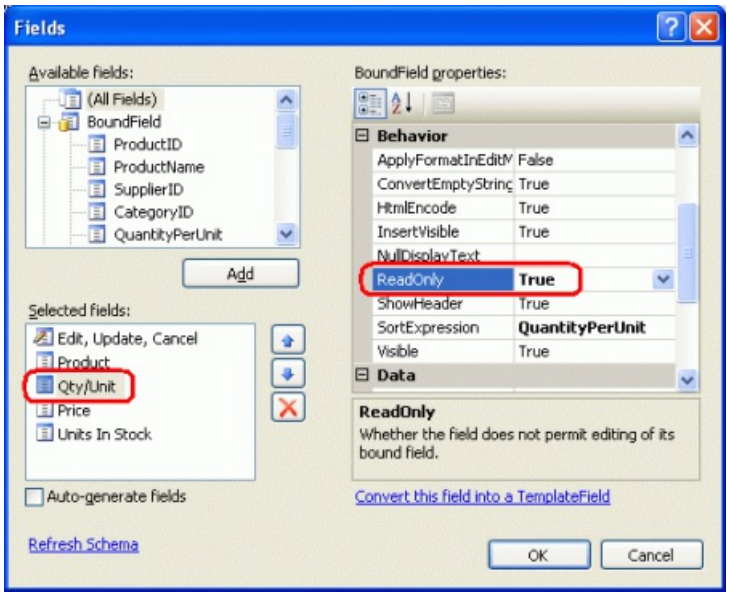


Figure 3: Make the `QuantityPerUnit` BoundField Read-Only (Click to view full-size image)

Finally, check the Enable Editing checkbox from the GridView's smart tag. After completing these steps the

`ErrorHandling.aspx` page's Designer should look similar to Figure 4.

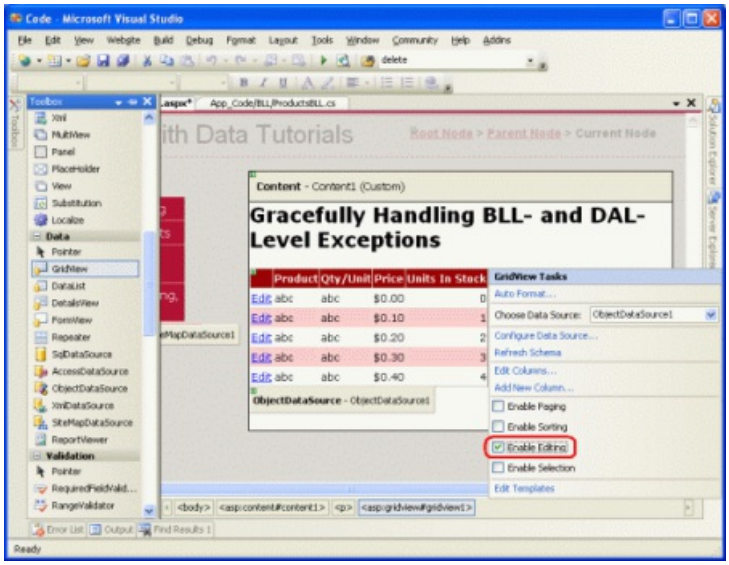


Figure 4: Remove All But the Needed BoundFields and Check the Enable Editing Checkbox (Click to view full-size image)

At this point we have a list of all of the products' `ProductName`, `QuantityPerUnit`, `UnitPrice`, and `UnitsInStock` fields; however, only the `ProductName`, `UnitPrice`, and `UnitsInStock` fields can be edited.

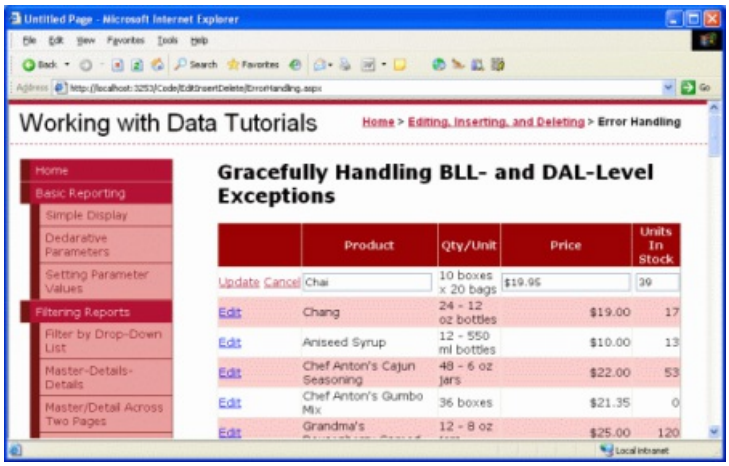


Figure 5: Users Can Now Easily Edit Products' Names, Prices, and Units In Stock Fields (Click to view full-size image)

Step 2: Gracefully Handling DAL-Level Exceptions

While our editable GridView works wonderfully when users enter legal values for the edited product's name, price, and units in stock, entering illegal values results in an exception. For example, omitting the `ProductName` value causes a NoNullAllowedException to be thrown since the `ProductName` property in the `ProductsRow` class has its `AllowDBNull` property set to `false`; if the database is down, a `SqlException` will be thrown by the TableAdapter when attempting to connect to the database. Without taking any action, these exceptions bubble up from the Data Access Layer to the Business Logic Layer, then to the ASP.NET page, and finally to the ASP.NET runtime.

Depending on how your web application is configured and whether or not you're visiting the application from `localhost`, an unhandled exception can result in either a generic server-error page, a detailed error report, or a user-friendly web page. See Web Application Error Handling in ASP.NET and the customErrors Element for more information on how the ASP.NET runtime responds to an uncaught exception.

Figure 6 shows the screen encountered when attempting to update a product without specifying the

`ProductName` value. This is the default detailed error report displayed when coming through `localhost`.

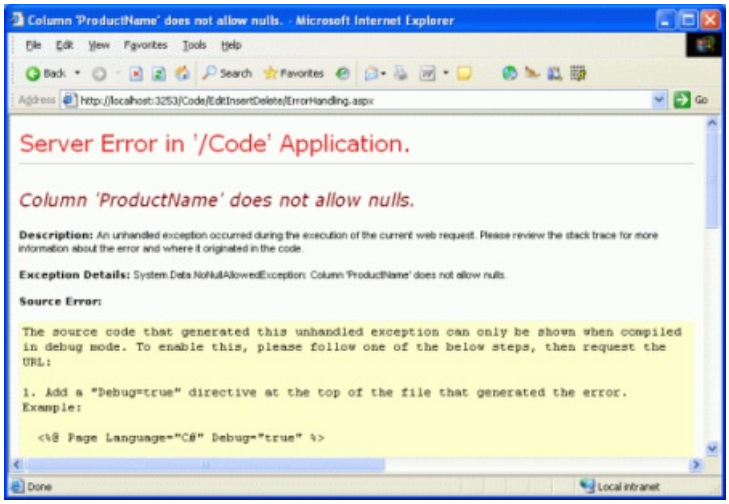


Figure 6: Omitting the Product's Name Will Display Exception Details (Click to view full-size image)

While such exception details are helpful when testing an application, presenting an end user with such a screen in the face of an exception is less than ideal. An end user likely doesn't know what a `NoNullAllowedException` is or why it was caused. A better approach is to present the user with a more user-friendly message explaining that there were problems attempting to update the product.

If an exception occurs when performing the operation, the post-level events in both the ObjectDataSource and the data Web control provide a means to detect it and cancel the exception from bubbling up to the ASP.NET runtime. For our example, let's create an event handler for the GridView's `RowUpdated` event that determines if an exception has fired and, if so, displays the exception details in a Label Web control.

Start by adding a Label to the ASP.NET page, setting its `ID` property to `ExceptionDetails` and clearing out its `Text` property. In order to draw the user's eye to this message, set its `CssClass` property to `Warning`, which is a CSS class we added to the `Styles.css` file in the previous tutorial. Recall that this CSS class causes the Label's text to be displayed in a red, italic, bold, extra large font.

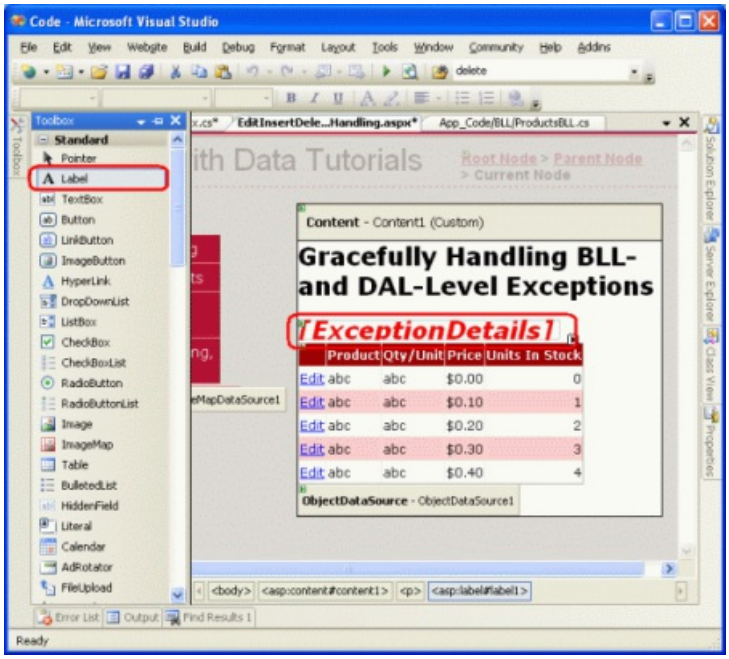


Figure 7: Add a Label Web Control to the Page (Click to view full-size image)

Since we want this Label Web control to be visible only immediately after an exception has occurred, set its `Visible` property to false in the `Page_Load` event handler:

```
protected void Page_Load(object sender, EventArgs e)
{
    ExceptionDetails.Visible = false;
}
```

With this code, on the first page visit and subsequent postbacks the `ExceptionDetails` control will have its `Visible` property set to `false`. In the face of a DAL- or BLL-level exception, which we can detect in the GridView's `RowUpdated` event handler, we will set the `ExceptionDetails` control's `Visible` property to true. Since Web control event handlers occur after the `Page_Load` event handler in the page lifecycle, the Label will be shown. However, on the next postback, the `Page_Load` event handler will revert the `Visible` property back to `false`, hiding it from view again.

> **NOTE**
>
> Alternatively, we could remove the necessity for setting the `ExceptionDetails` control's `Visible` property in `Page_Load` by assigning its `Visible` property `false` in the declarative syntax and disabling its view state (setting its `EnableViewState` property to `false`). We'll use this alternative approach in a future tutorial.

With the Label control added, our next step is to create the event handler for the GridView's `RowUpdated` event. Select the GridView in the Designer, go to the Properties window, and click the lightning bolt icon, listing the GridView's events. There should already be an entry there for the GridView's `RowUpdating` event, as we created an event handler for this event earlier in this tutorial. Create an event handler for the `RowUpdated` event as well.

Figure 8: Create an Event Handler for the GridView's `RowUpdated` Event

> **NOTE**
> You can also create the event handler through the drop-down lists at the top of the code-behind class file. Select the GridView from the drop-down list on the left and the `RowUpdated` event from the one on the right.

Creating this event handler will add the following code to the ASP.NET page's code-behind class:

```
protected void GridView1_RowUpdated(object sender, GridViewUpdatedEventArgs e)
{
}
```

This event handler's second input parameter is an object of type GridViewUpdatedEventArgs, which has three properties of interest for handling exceptions:

- `Exception` a reference to the thrown exception; if no exception has been thrown, this property will have a value of `null`
- `ExceptionHandled` a Boolean value that indicates whether or not the exception was handled in the `RowUpdated` event handler; if `false` (the default), the exception is re-thrown, percolating up to the ASP.NET runtime
- `KeepInEditMode` if set to `true` the edited GridView row remains in edit mode; if `false` (the default), the GridView row reverts back to its read-only mode

Our code, then, should check to see if `Exception` is not `null`, meaning that an exception was raised while performing the operation. If this is the case, we want to:

- Display a user-friendly message in the `ExceptionDetails` Label
- Indicate that the exception was handled
- Keep the GridView row in edit mode

This following code accomplishes these objectives:

```
protected void GridView1_RowUpdated(object sender, GridViewUpdatedEventArgs e)
{
    if (e.Exception != null)
    {
        // Display a user-friendly message
        ExceptionDetails.Visible = true;
        ExceptionDetails.Text = "There was a problem updating the product. ";
        if (e.Exception.InnerException != null)
        {
            Exception inner = e.Exception.InnerException;
            if (inner is System.Data.Common.DbException)
                ExceptionDetails.Text +=
                    "Our database is currently experiencing problems." +
                    "Please try again later.";
            else if (inner is NoNullAllowedException)
                ExceptionDetails.Text +=
                    "There are one or more required fields that are missing.";
            else if (inner is ArgumentException)
            {
                string paramName = ((ArgumentException)inner).ParamName;
                ExceptionDetails.Text +=
                    string.Concat("The ", paramName, " value is illegal.");
            }
            else if (inner is ApplicationException)
                ExceptionDetails.Text += inner.Message;
        }
        // Indicate that the exception has been handled
        e.ExceptionHandled = true;
        // Keep the row in edit mode
        e.KeepInEditMode = true;
    }
}
```

This event handler begins by checking to see if `e.Exception` is `null`. If it's not, the `ExceptionDetails` Label's `Visible` property is set to `true` and its `Text` property to "There was a problem updating the product." The details of the actual exception that was thrown reside in the `e.Exception` object's `InnerException` property. This inner exception is examined and, if it is of a particular type, an additional, helpful message is appended to the `ExceptionDetails` Label's `Text` property. Lastly, the `ExceptionHandled` and `KeepInEditMode` properties are both set to `true`.

Figure 9 shows a screen shot of this page when omitting the name of the product; Figure 10 shows the results when entering an illegal `UnitPrice` value (-50).

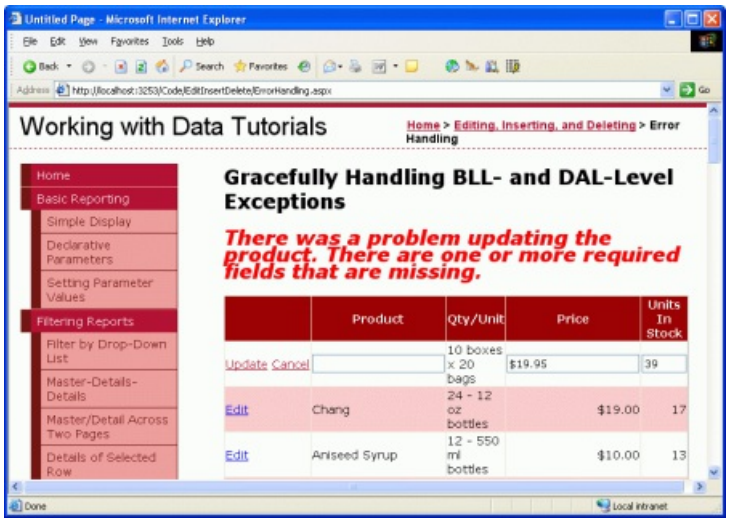


Figure 9: The `ProductName` BoundField Must Contain a Value (Click to view full-size image)

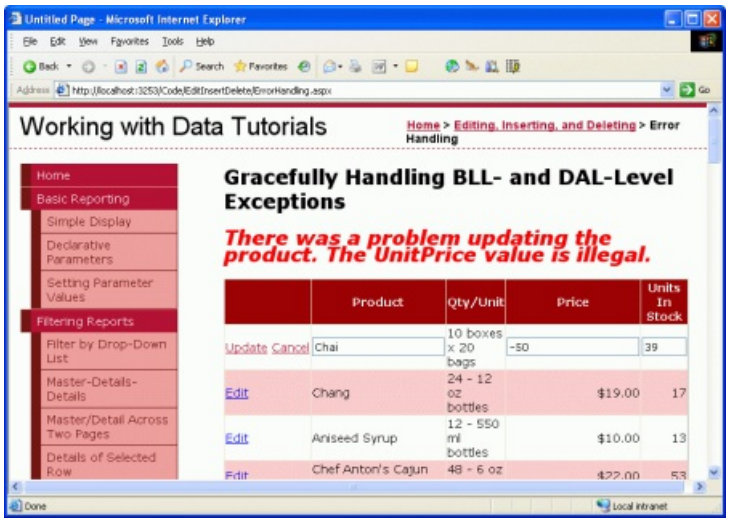


Figure 10: Negative `UnitPrice` Values are Not Allowed (Click to view full-size image)

By setting the `e.ExceptionHandled` property to `true`, the `RowUpdated` event handler has indicated that it has handled the exception. Therefore, the exception won't propagate up to the ASP.NET runtime.

> **NOTE**
>
> Figures 9 and 10 show a graceful way to handle exceptions raised due to invalid user input. Ideally, though, such invalid input will never reach the Business Logic Layer in the first place, as the ASP.NET page should ensure that the user's inputs are valid before invoking the `ProductsBLL` class's `UpdateProduct` method. In our next tutorial we'll see how to add validation controls to the editing and inserting interfaces to ensure that the data submitted to the Business Logic Layer conforms to the business rules. The validation controls not only prevent the invocation of the `UpdateProduct` method until the user-supplied data is valid, but also provide a more informative user experience for identifying data entry problems.

Step 3: Gracefully Handling BLL-Level Exceptions

When inserting, updating, or deleting data, the Data Access Layer may throw an exception in the face of a data-related error. The database may be offline, a required database table column might not have had a value specified, or a table-level constraint may have been violated. In addition to strictly data-related exceptions, the Business Logic Layer can use exceptions to indicate when business rules have been violated. In the Creating a Business Logic Layer tutorial, for example, we added a business rule check to the original `UpdateProduct` overload. Specifically, if the user was marking a product as discontinued, we required that the product not be the only one provided by its supplier. If this condition was violated, an `ApplicationException` was thrown.

For the `UpdateProduct` overload created in this tutorial, let's add a business rule that prohibits the `UnitPrice` field from being set to a new value that's more than twice the original `UnitPrice` value. To accomplish this, adjust the `UpdateProduct` overload so that it performs this check and throws an `ApplicationException` if the rule is violated. The updated method follows:

```
public bool UpdateProduct(string productName, decimal? unitPrice, short? unitsInStock,
    int productID)
{
    Northwind.ProductsDataTable products = Adapter.GetProductByProductID(productID);
    if (products.Count == 0)
        // no matching record found, return false
        return false;
    Northwind.ProductsRow product = products[0];
    // Make sure the price has not more than doubled
    if (unitPrice != null && !product.IsUnitPriceNull())
        if (unitPrice > product.UnitPrice * 2)
          throw new ApplicationException(
            "When updating a product price," +
            " the new price cannot exceed twice the original price.");
    product.ProductName = productName;
    if (unitPrice == null) product.SetUnitPriceNull();
      else product.UnitPrice = unitPrice.Value;
    if (unitsInStock == null) product.SetUnitsInStockNull();
      else product.UnitsInStock = unitsInStock.Value;
    // Update the product record
    int rowsAffected = Adapter.Update(product);
    // Return true if precisely one row was updated, otherwise false
    return rowsAffected == 1;
}
```

With this change, any price update that is more than twice the existing price will cause an `ApplicationException` to be thrown. Just like the exception raised from the DAL, this BLL-raised `ApplicationException` can be detected and handled in the GridView's `RowUpdated` event handler. In fact, the `RowUpdated` event handler's code, as written, will correctly detect this exception and display the `ApplicationException`'s `Message` property value. Figure 11 shows a screen shot when a user attempts to update the price of Chai to $50.00, which is more than double its current price of $19.95.

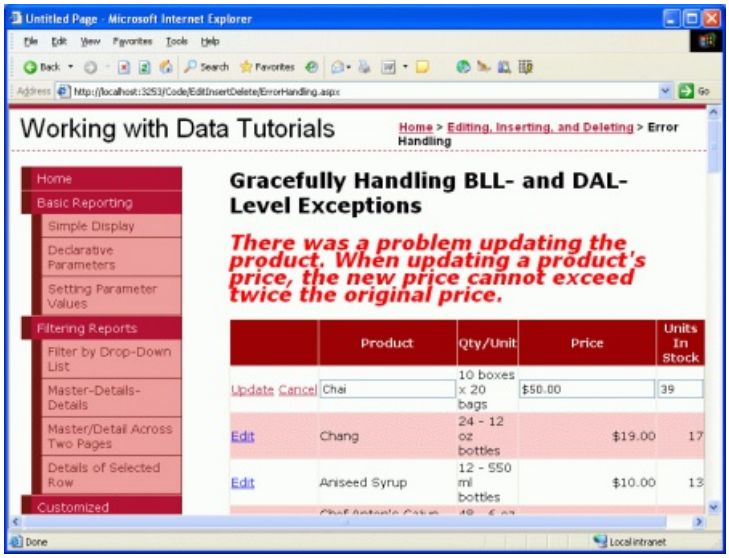


Figure 11: The Business Rules Disallow Price Increases That More Than Double a Product's Price (Click to view full-size image)

> **NOTE**
>
> Ideally our business logic rules would be refactored out of the `UpdateProduct` method overloads and into a common method. This is left as an exercise for the reader.

Summary

During inserting, updating, and deleting operations, both the data Web control and the ObjectDataSource involved fire pre- and post-level events that bookend the actual operation. As we saw in this tutorial and the preceding one, when working with an editable GridView the GridView's `RowUpdating` event fires, followed by the ObjectDataSource's `Updating` event, at which point the update command is made to the ObjectDataSource's underlying object. After the operation has completed, the ObjectDataSource's `Updated` event fires, followed by the GridView's `RowUpdated` event.

We can create event handlers for the pre-level events in order to customize the input parameters or for the post-level events in order to inspect and respond to the operation's results. Post-level event handlers are most commonly used to detect whether an exception occurred during the operation. In the face of an exception, these post-level event handlers can optionally handle the exception on their own. In this tutorial we saw how to handle such an exception by displaying a friendly error message.

In the next tutorial we'll see how to lessen the likelihood of exceptions arising from data formatting issues (such as entering a negative `UnitPrice`). Specifically, we'll look at how to add validation controls to the editing and inserting interfaces.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Liz Shulok. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Adding Validation Controls to the Editing and Inserting Interfaces (C#)

2/20/2020 • 19 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll see how easy it is to add validation controls to the EditItemTemplate and InsertItemTemplate of a data Web control, to provide a more foolproof user interface.

Introduction

The GridView and DetailsView controls in the examples we've explored over the past three tutorials have all been composed of BoundFields and CheckBoxFields (the field types automatically added by Visual Studio when binding a GridView or DetailsView to a data source control through the smart tag). When editing a row in a GridView or DetailsView, those BoundFields that are not read-only are converted into textboxes, from which the end user can modify the existing data. Similarly, when inserting a new record into a DetailsView control, those BoundFields whose `InsertVisible` property is set to `true` (the default) are rendered as empty textboxes, into which the user can provide the new record's field values. Likewise, CheckBoxFields, which are disabled in the standard, read-only interface, are converted into enabled checkboxes in the editing and inserting interfaces.

While the default editing and inserting interfaces for the BoundField and CheckBoxField can be helpful, the interface lacks any sort of validation. If a user makes a data entry mistake - such as omitting the `ProductName` field or entering an invalid value for `UnitsInStock` (such as -50) an exception will be raised from within the depths of the application architecture. While this exception can be gracefully handled as demonstrated in the previous tutorial, ideally the editing or inserting user interface would include validation controls to prevent a user from entering such invalid data in the first place.

In order to provide a customized editing or inserting interface, we need to replace the BoundField or CheckBoxField with a TemplateField. TemplateFields, which were the topic of discussion in the Using TemplateFields in the GridView Control and Using TemplateFields in the DetailsView Control tutorials, can consist of multiple templates defining separate interfaces for different row states. The TemplateField's `ItemTemplate` is used to when rendering read-only fields or rows in the DetailsView or GridView controls, whereas the `EditItemTemplate` and `InsertItemTemplate` indicate the interfaces to use for the editing and inserting modes, respectively.

In this tutorial we'll see how easy it is to add validation controls to the TemplateField's `EditItemTemplate` and `InsertItemTemplate` to provide a more foolproof user interface. Specifically, this tutorial takes the example created in the Examining the Events Associated with Inserting, Updating, and Deleting tutorial and augments the editing and inserting interfaces to include appropriate validation.

Step 1: Replicating the Example fromExamining the Events Associated with Inserting, Updating, and Deleting

In the Examining the Events Associated with Inserting, Updating, and Deleting tutorial we created a page that listed the names and prices of the products in an editable GridView. Additionally, the page included a DetailsView whose `DefaultMode` property was set to `Insert`, thereby always rendering in insert mode. From this DetailsView, the user could enter the name and price for a new product, click Insert, and have it added to the

system (see Figure 1).

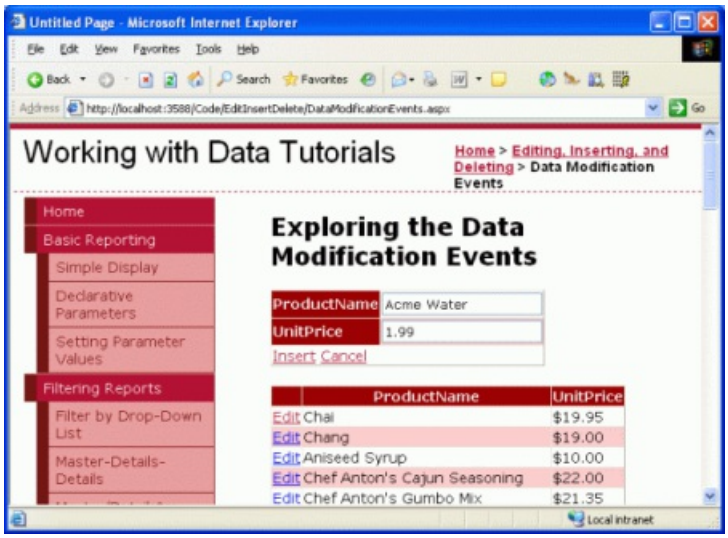


Figure 1: The Previous Example Allows Users to Add New Products and Edit Existing Ones (Click to view full-size image)

Our goal for this tutorial is to augment the DetailsView and GridView to provide validation controls. In particular, our validation logic will:

- Require that the name be provided when inserting or editing a product
- Require that the price be provided when inserting a record; when editing a record, we will still require a price, but will use the programmatic logic in the GridView's `RowUpdating` event handler already present from the earlier tutorial
- Ensure that the value entered for the price is a valid currency format

Before we can look at augmenting the previous example to include validation, we first need to replicate the example from the `DataModificationEvents.aspx` page to the page for this tutorial, `UIValidation.aspx`. To accomplish this we need to copy over both the `DataModificationEvents.aspx` page's declarative markup and its source code. First copy over the declarative markup by performing the following steps:

1. Open the `DataModificationEvents.aspx` page in Visual Studio
2. Go to the page's declarative markup (click on the Source button at the bottom of the page)
3. Copy the text within the `<asp:Content>` and `</asp:Content>` tags (lines 3 through 44), as shown in Figure 2.

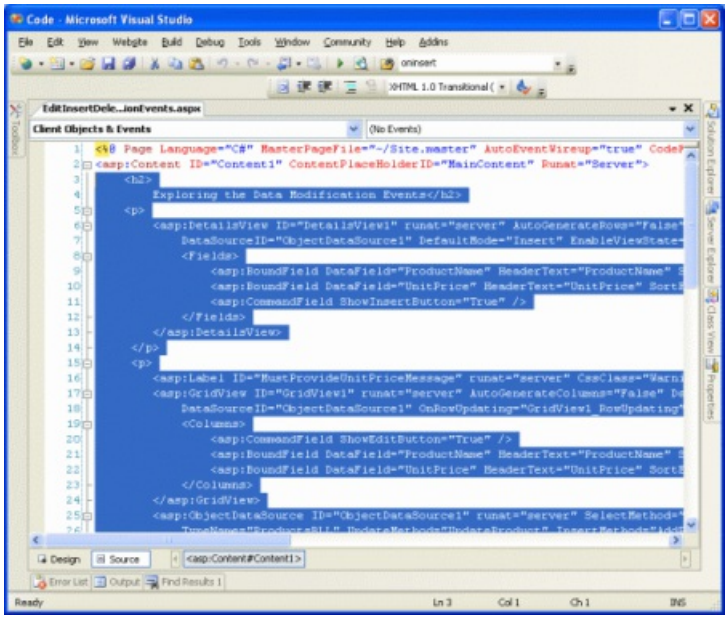


Figure 2: Copy the Text Within the `<asp:Content>` Control (Click to view full-size image)

1. Open the `UIValidation.aspx` page
2. Go to the page's declarative markup
3. Paste the text within the `<asp:Content>` control.

To copy over the source code, open the `DataModificationEvents.aspx.cs` page and copy just the text *within* the `EditInsertDelete_DataModificationEvents` class. Copy the three event handlers (`Page_Load` , `GridView1_RowUpdating` , and `ObjectDataSource1_Inserting`), but do **not** copy the class declaration or `using` statements. Paste the copied text *within* the `EditInsertDelete_UIValidation` class in `UIValidation.aspx.cs` .

After moving over the content and code from `DataModificationEvents.aspx` to `UIValidation.aspx` , take a moment to test out your progress in a browser. You should see the same output and experience the same functionality in each of these two pages (refer back to Figure 1 for a screen shot of `DataModificationEvents.aspx` in action).

Step 2: Converting the BoundFields Into TemplateFields

To add validation controls to the editing and inserting interfaces, the BoundFields used by the DetailsView and GridView controls need to be converted into TemplateFields. To achieve this, click on the Edit Columns and Edit Fields links in the GridView and DetailsView's smart tags, respectively. There, select each of the BoundFields and click the "Convert this field into a TemplateField" link.

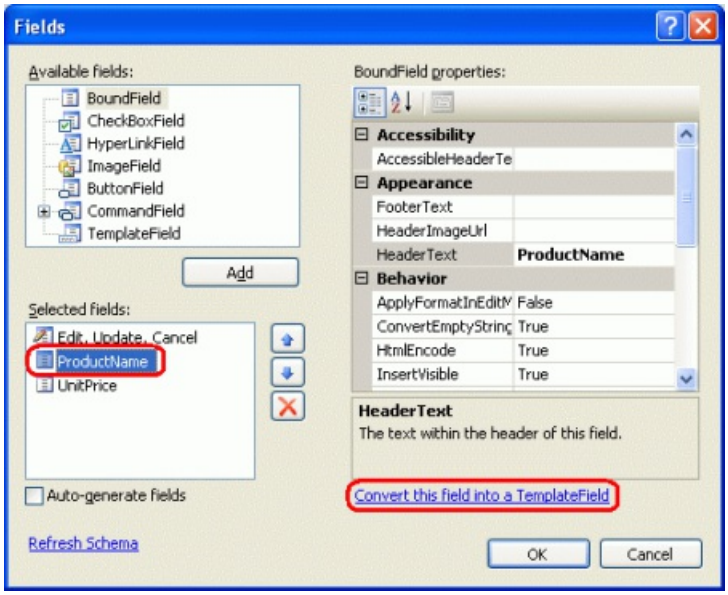


Figure 3: Convert Each of the DetailsView's and GridView's BoundFields Into TemplateFields (Click to view full-size image)

Converting a BoundField into a TemplateField through the Fields dialog box generates a TemplateField that exhibits the same read-only, editing, and inserting interfaces as the BoundField itself. The following markup shows the declarative syntax for the `ProductName` field in the DetailsView after it has been converted into a TemplateField:

```
<asp:TemplateField HeaderText="ProductName" SortExpression="ProductName">
    <EditItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
         Text='<%# Bind("ProductName") %>'></asp:TextBox>
    </EditItemTemplate>
    <InsertItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
         Text='<%# Bind("ProductName") %>'></asp:TextBox>
    </InsertItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
         Text='<%# Bind("ProductName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

Note that this TemplateField had three templates automatically created `ItemTemplate`, `EditItemTemplate`, and `InsertItemTemplate`. The `ItemTemplate` displays a single data field value (`ProductName`) using a Label Web control, while the `EditItemTemplate` and `InsertItemTemplate` present the data field value in a TextBox Web control that associates the data field with the TextBox's `Text` property using two-way databinding. Since we are only using the DetailsView in this page for inserting, you may remove the `ItemTemplate` and `EditItemTemplate` from the two TemplateFields, although there's no harm in leaving them.

Since the GridView does not support the built-in inserting features of the DetailsView, converting the GridView's `ProductName` field into a TemplateField results in only an `ItemTemplate` and `EditItemTemplate`:

```
<asp:TemplateField HeaderText="ProductName" SortExpression="ProductName">
    <EditItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
         Text='<%# Bind("ProductName") %>'></asp:TextBox>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
          Text='<%# Bind("ProductName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

By clicking the "Convert this field into a TemplateField," Visual Studio has created a TemplateField whose templates mimic the user interface of the converted BoundField. You can verify this by visiting this page through a browser. You'll find that the appearance and behavior of the TemplateFields is identical to the experience when BoundFields were used instead.

> **NOTE**
>
> Feel free to customize the editing interfaces in the templates as needed. For example, we may want to have the TextBox in the `UnitPrice` TemplateFields rendered as a smaller textbox than the `ProductName` textbox. To accomplish this you can set the TextBox's `Columns` property to an appropriate value or provide an absolute width via the `Width` property. In the next tutorial we'll see how to completely customize the editing interface by replacing the TextBox with an alternate data entry Web control.

Step 3: Adding the Validation Controls to the GridView's `EditItemTemplate` S

When constructing data entry forms, it is important that users enter any required fields and that all provided inputs are legal, properly-formatted values. To help ensure that a user's inputs are valid, ASP.NET provides five built-in validation controls that are designed to be used to validate the value of a single input control:

- RequiredFieldValidator ensures that a value has been provided
- CompareValidator validates a value against another Web control value or a constant value, or ensures that the value's format is legal for a specified data type
- RangeValidator ensures that a value is within a range of values
- RegularExpressionValidator validates a value against a regular expression
- CustomValidator validates a value against a custom, user-defined method

For more information on these five controls, check out the Validation Controls section of the ASP.NET Quickstart Tutorials.

For our tutorial we'll need to use a RequiredFieldValidator in both the DetailsView and GridView's `ProductName` TemplateFields and a RequiredFieldValidator in the DetailsView's `UnitPrice` TemplateField. Furthermore, we'll need to add a CompareValidator to both controls' `UnitPrice` TemplateFields that ensures that the entered price has a value greater than or equal to 0 and is presented in a valid currency format.

> **NOTE**
> While ASP.NET 1.x had these same five validation controls, ASP.NET 2.0 has added a number of improvements, the main two being client-side script support for browsers other than Internet Explorer and the ability to partition validation controls on a page into validation groups. For more information on the new validation control features in 2.0, refer to Dissecting the Validation Controls in ASP.NET 2.0.

Let's start by adding the necessary validation controls to the `EditItemTemplate` s in the GridView's TemplateFields. To accomplish this, click on the Edit Templates link from the GridView's smart tag to bring up the template editing interface. From here, you can select which template to edit from the drop-down list. Since we want to augment the editing interface, we need to add validation controls to the `ProductName` and `UnitPrice`'s `EditItemTemplate` s.

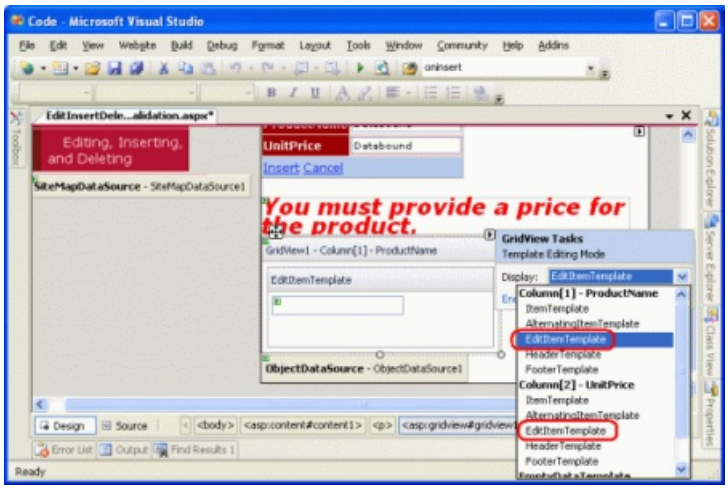


Figure 4: We Need to Extend the `ProductName` and `UnitPrice`'s `EditItemTemplate` s (Click to view full-size image)

In the `ProductName` `EditItemTemplate`, add a RequiredFieldValidator by dragging it from the Toolbox into the template editing interface, placing after the TextBox.

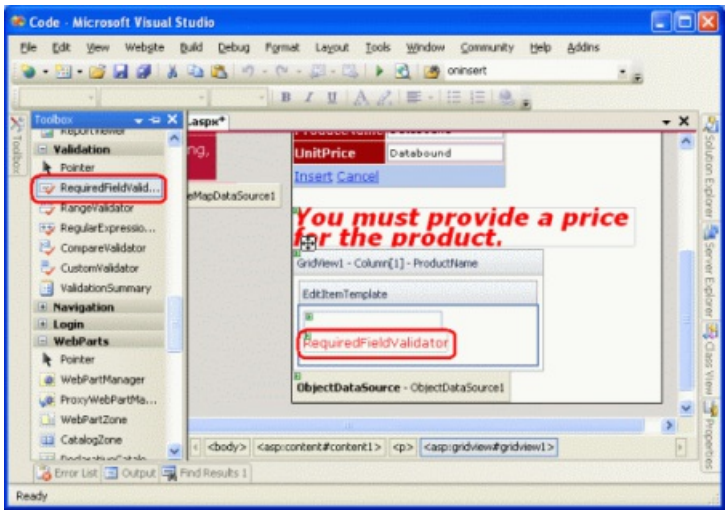


Figure 5: Add a RequiredFieldValidator to the `ProductName` `EditItemTemplate` (Click to view full-size image)

All validation controls work by validating the input of a single ASP.NET Web control. Therefore, we need to indicate that the RequiredFieldValidator we just added should validate against the TextBox in the `EditItemTemplate`; this is accomplished by setting the validation control's ControlToValidate property to the `ID` of the appropriate Web control. The TextBox currently has the rather nondescript `ID` of `TextBox1`, but let's change it to something more appropriate. Click on the TextBox in the template and then, from the Properties window, change the `ID` from `TextBox1` to `EditProductName`.

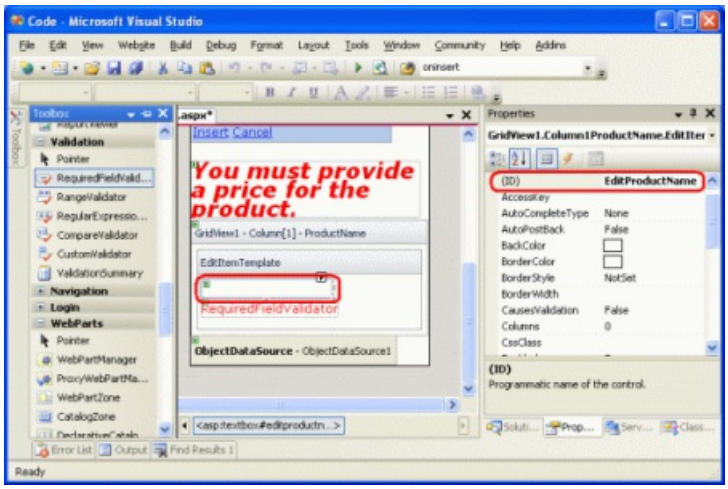


Figure 6: Change the TextBox's `ID` to `EditProductName` (Click to view full-size image)

Next, set the RequiredFieldValidator's `ControlToValidate` property to `EditProductName`. Finally, set the ErrorMessage property to "You must provide the product's name" and the Text property to "*". The `Text` property value, if provided, is the text that is displayed by the validation control if the validation fails. The `ErrorMessage` property value, which is required, is used by the ValidationSummary control; if the `Text` property value is omitted, the `ErrorMessage` property value is also the text displayed by the validation control on invalid input.

After setting these three properties of the RequiredFieldValidator, your screen should look similar to Figure 7.

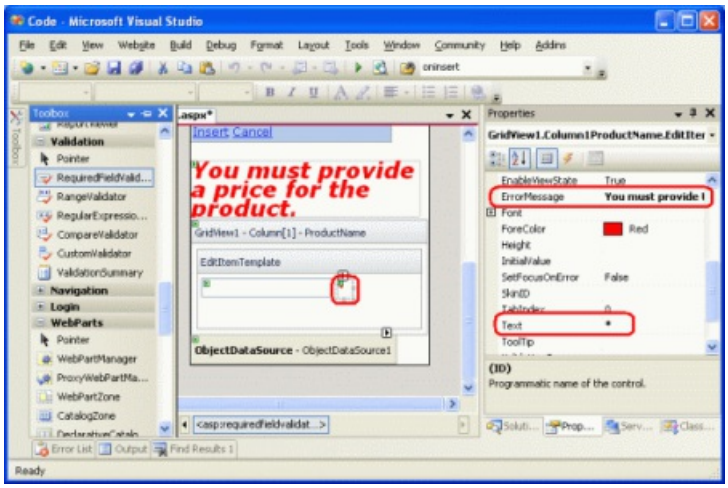


Figure 7: Set the RequiredFieldValidator's `ControlToValidate`, `ErrorMessage`, and `Text` Properties (Click to view full-size image)

With the RequiredFieldValidator added to the `ProductName` `EditItemTemplate`, all that remains is to add the necessary validation to the `UnitPrice` `EditItemTemplate`. Since we've decided that, for this page, the `UnitPrice` is optional when editing a record, we don't need to add a RequiredFieldValidator. We do, however, need to add a CompareValidator to ensure that the `UnitPrice`, if supplied, is properly formatted as a currency and is greater than or equal to 0.

Before we add the CompareValidator to the `UnitPrice` `EditItemTemplate`, let's first change the TextBox Web control's ID from `TextBox2` to `EditUnitPrice`. After making this change, add the CompareValidator, setting its `ControlToValidate` property to `EditUnitPrice`, its `ErrorMessage` property to "The price must be greater than or equal to zero and cannot include the currency symbol", and its `Text` property to "*".

To indicate that the `UnitPrice` value must be greater than or equal to 0, set the CompareValidator's Operator property to `GreaterThanEqual`, its ValueToCompare property to "0", and its Type property to `Currency`. The following declarative syntax shows the `UnitPrice` TemplateField's `EditItemTemplate` after these changes have been made:

```
<EditItemTemplate>
    <asp:TextBox ID="EditUnitPrice" runat="server"
      Text='<%# Bind("UnitPrice", "{0:c}") %>'
      Columns="6"></asp:TextBox>
    <asp:CompareValidator ID="CompareValidator1" runat="server"
        ControlToValidate="EditUnitPrice"
        ErrorMessage="The price must be greater than or equal to zero and
                      cannot include the currency symbol"
        Operator="GreaterThanEqual" Type="Currency"
        ValueToCompare="0">*</asp:CompareValidator>
</EditItemTemplate>
```

After making these changes, open the page in a browser. If you attempt to omit the name or enter an invalid price value when editing a product, an asterisk appears next to the textbox. As Figure 8 shows, a price value that includes the currency symbol such as $19.95 is considered invalid. The CompareValidator's `Currency` `Type` allows for digit separators (such as commas or periods, depending on the culture settings) and a leading plus or minus sign, but does *not* permit a currency symbol. This behavior may perplex users as the editing interface currently renders the `UnitPrice` using the currency format.

> **NOTE**
>
> Recall that in the *Events Associated with Inserting, Updating, and Deleting* tutorial we set the BoundField's `DataFormatString` property to `{0:c}` in order to format it as a currency. Furthermore, we set the `ApplyFormatInEditMode` property to true, causing the GridView's editing interface to format the `UnitPrice` as a currency. When converting the BoundField into a TemplateField, Visual Studio noted these settings and formatted the TextBox's `Text` property as a currency using the databinding syntax `<%# Bind("UnitPrice", "{0:c}") %>`.

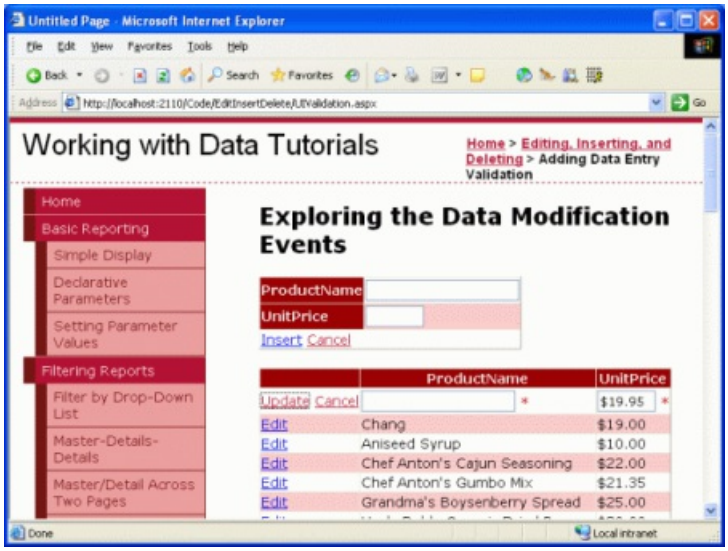


Figure 8: An Asterisk Appears Next to the Textboxes with Invalid Input (Click to view full-size image)

While the validation works as-is, the user has to manually remove the currency symbol when editing a record, which is not acceptable. To remedy this, we have three options:

1. Configure the `EditItemTemplate` so that the `UnitPrice` value is not formatted as a currency.
2. Allow the user to enter a currency symbol by removing the CompareValidator and replacing it with a RegularExpressionValidator that properly checks for a properly formatted currency value. The problem here is that the regular expression to validate a currency value is not pretty and would require writing code if we wanted to incorporate culture settings.
3. Remove the validation control altogether and rely on server-side validation logic in the GridView's `RowUpdating` event handler.

Let's go with option #1 for this exercise. Currently the `UnitPrice` is formatted as a currency due to the databinding expression for the TextBox in the `EditItemTemplate`: `<%# Bind("UnitPrice", "{0:c}") %>`. Change the Bind statement to `Bind("UnitPrice", "{0:n2}")`, which formats the result as a number with two digits of precision. This can be done directly through the declarative syntax or by clicking on the Edit DataBindings link from the `EditUnitPrice` TextBox in the `UnitPrice` TemplateField's `EditItemTemplate` (see Figures 9 and 10).

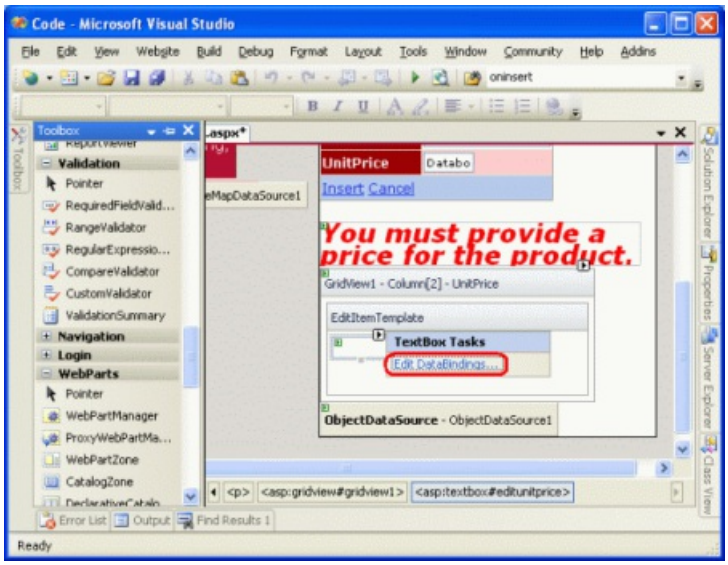


Figure 9: Click on the TextBox's Edit DataBindings link (Click to view full-size image)

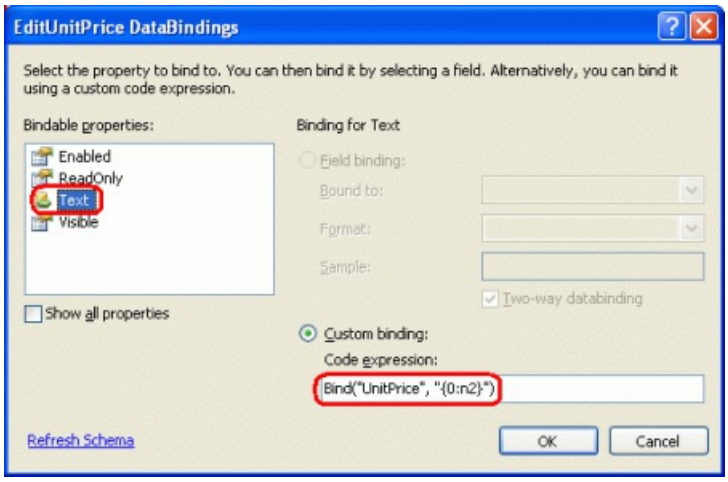


Figure 10: Specify the Format Specifier in the `Bind` Statement (Click to view full-size image)

With this change, the formatted price in the editing interface includes commas as the group separator and a period as the decimal separator, but leaves off the currency symbol.

> **NOTE**
>
> The `UnitPrice` `EditItemTemplate` doesn't include a RequiredFieldValidator, allowing the postback to ensue and the updating logic to commence. However, the `RowUpdating` event handler copied over from the *Examining the Events Associated with Inserting, Updating, and Deleting* tutorial includes a programmatic check that ensures that the `UnitPrice` is provided. Feel free to remove this logic, leave it in as-is, or add a RequiredFieldValidator to the `UnitPrice` `EditItemTemplate`.

Step 4: Summarizing Data Entry Problems

In addition to the five validation controls, ASP.NET includes the ValidationSummary control, which displays the `ErrorMessage` s of those validation controls that detected invalid data. This summary data can be displayed as text on the web page or through a modal, client-side messagebox. Let's enhance this tutorial to include a client-side messagebox summarizing any validation problems.

To accomplish this, drag a ValidationSummary control from the Toolbox onto the Designer. The location of the Validation control doesn't really matter, since we're going to configure it to only display the summary as a messagebox. After adding the control, set its ShowSummary property to `false` and its ShowMessageBox property to `true`. With this addition, any validation errors are summarized in a client-side messagebox.

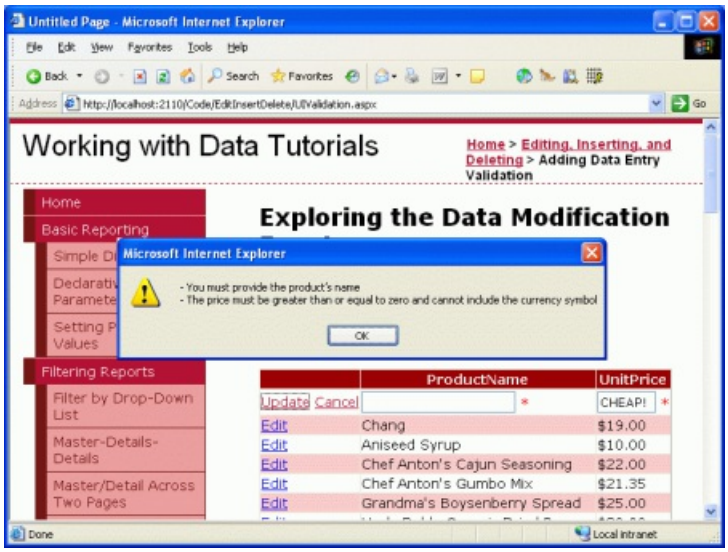


Figure 11: The Validation Errors are Summarized in a Client-Side Messagebox (Click to view full-size image)

Step 5: Adding the Validation Controls to the DetailsView's `InsertItemTemplate`

All that remains for this tutorial is to add the validation controls to the DetailsView's inserting interface. The process of adding validation controls to the DetailsView's templates is identical to that examined in Step 3; therefore, we'll breeze through the task in this step. As we did with the GridView's `EditItemTemplate` s, I encourage you to rename the `ID` s of the TextBoxes from the nondescript `TextBox1` and `TextBox2` to `InsertProductName` and `InsertUnitPrice`.

Add a RequiredFieldValidator to the `ProductName` `InsertItemTemplate`. Set the `ControlToValidate` to the `ID` of the TextBox in the template, its `Text` property to "*" and its `ErrorMessage` property to "You must provide the product's name".

Since the `UnitPrice` is required for this page when adding a new record, add a RequiredFieldValidator to the `UnitPrice` `InsertItemTemplate`, setting its `ControlToValidate`, `Text`, and `ErrorMessage` properties appropriately. Finally, add a CompareValidator to the `UnitPrice` `InsertItemTemplate` as well, configuring its `ControlToValidate`, `Text`, `ErrorMessage`, `Type`, `Operator`, and `ValueToCompare` properties just like we did with the `UnitPrice`'s CompareValidator in the GridView's `EditItemTemplate`.

After adding these validation controls, a new product cannot be added to the system if its name is not supplied or if its price is a negative number or illegally formatted.

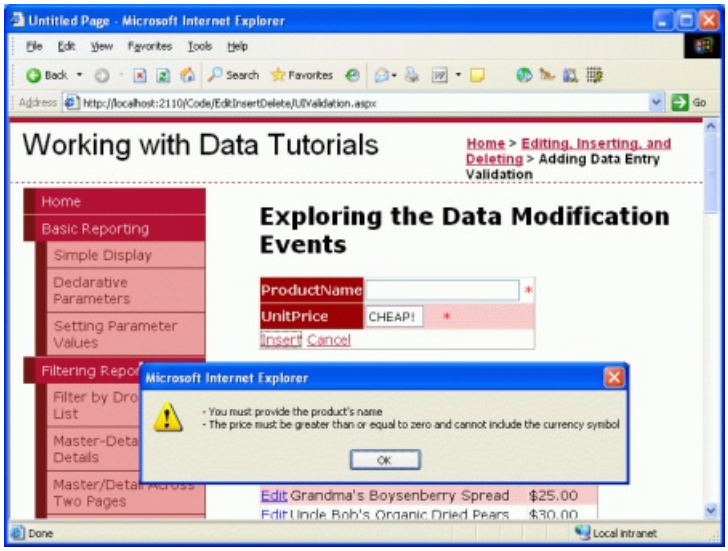


Figure 12: Validation Logic has been Added to the DetailsView's Inserting Interface (Click to view full-size image)

Step 6: Partitioning the Validation Controls Into Validation Groups

Our page consists of two logically disparate sets of validation controls: those that correspond to the GridView's editing interface and those that correspond to the DetailsView's inserting interface. By default, when a postback occurs *all* validation controls on the page are checked. However, when editing a record we don't want the DetailsView's inserting interface's validation controls to validate. Figure 13 illustrates our current dilemma when a user is editing a product with perfectly legal values, clicking Update causes a validation error because the name and price values in the inserting interface are blank.

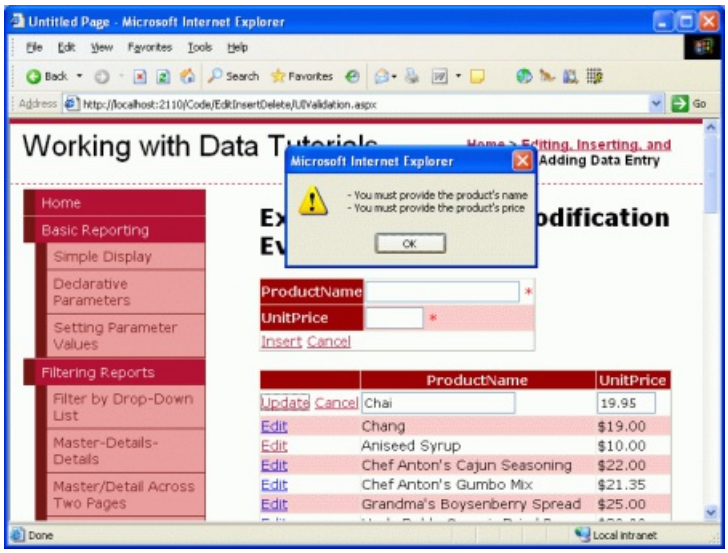


Figure 13: Updating a Product Causes the Inserting Interface's Validation Controls to Fire (Click to view full-size image)

The validation controls in ASP.NET 2.0 can be partitioned into validation groups through their `ValidationGroup` property. To associate a set of validation controls in a group, simply set their `ValidationGroup` property to the same value. For our tutorial, set the `ValidationGroup` properties of the validation controls in the GridView's TemplateFields to `EditValidationControls` and the `ValidationGroup` properties of the DetailsView's TemplateFields to `InsertValidationControls`. These changes can be done directly in the declarative markup or through the Properties window when using the Designer's edit template interface.

In addition to the validation controls, the Button and Button-related controls in ASP.NET 2.0 also include a `ValidationGroup` property. A validation group's validators are checked for validity only when a postback is induced by a Button that has the same `ValidationGroup` property setting. For example, in order for the DetailsView's Insert button to trigger the `InsertValidationControls` validation group we need to set the CommandField's `ValidationGroup` property to `InsertValidationControls` (see Figure 14). Additionally, set the GridView's CommandField's `ValidationGroup` property to `EditValidationControls`.

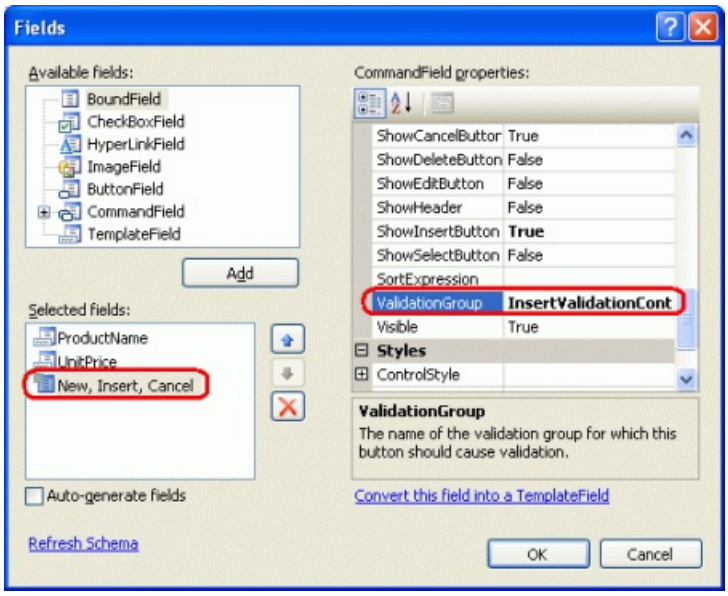


Figure 14: Set the DetailsView's CommandField's `ValidationGroup` property to `InsertValidationControls` (Click to view full-size image)

After these changes, the DetailsView and GridView's TemplateFields and CommandFields should look similar to the following:

The DetailsView's TemplateFields and CommandField

```
<asp:TemplateField HeaderText="ProductName"
  SortExpression="ProductName">
    <InsertItemTemplate>
        <asp:TextBox ID="InsertProductName" runat="server"
         Text='<%# Bind("ProductName") %>'></asp:TextBox>
        <asp:RequiredFieldValidator ID="RequiredFieldValidator2"
          runat="server" ControlToValidate="InsertProductName"
            ErrorMessage="You must provide the product name"
            ValidationGroup="InsertValidationControls">*
        </asp:RequiredFieldValidator>
    </InsertItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="UnitPrice" SortExpression="UnitPrice">
    <InsertItemTemplate>
         <asp:TextBox ID="InsertUnitPrice" runat="server"
           Text='<%# Bind("UnitPrice") %>' Columns="6">
         </asp:TextBox>
         <asp:RequiredFieldValidator ID="RequiredFieldValidator3"
           runat="server" ControlToValidate="InsertUnitPrice"
            ErrorMessage="You must provide the product price"
            ValidationGroup="InsertValidationControls">*
         </asp:RequiredFieldValidator>
        <asp:CompareValidator ID="CompareValidator2" runat="server"
           ControlToValidate="InsertUnitPrice"
           ErrorMessage="The price must be greater than or equal to zero and
                         cannot include the currency symbol"
           Operator="GreaterThanEqual" Type="Currency" ValueToCompare="0"
           ValidationGroup="InsertValidationControls">*
        </asp:CompareValidator>
     </InsertItemTemplate>
 </asp:TemplateField>
<asp:CommandField ShowInsertButton="True"
  ValidationGroup="InsertValidationControls" />
```

The GridView's CommandField and TemplateFields

```
<asp:CommandField ShowEditButton="True" ValidationGroup="EditValidationControls" />
<asp:TemplateField HeaderText="ProductName"
  SortExpression="ProductName">
    <EditItemTemplate>
        <asp:TextBox ID="EditProductName" runat="server"
          Text='<%# Bind("ProductName") %>'>
        </asp:TextBox>
        <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
            runat="server" ControlToValidate="EditProductName"
            ErrorMessage="You must provide the product name"
            ValidationGroup="EditValidationControls">*
        </asp:RequiredFieldValidator>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
          Text='<%# Bind("ProductName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="UnitPrice" SortExpression="UnitPrice">
    <EditItemTemplate>
        <asp:TextBox ID="EditUnitPrice" runat="server"
          Text='<%# Bind("UnitPrice", "{0:n2}") %>' Columns="6"></asp:TextBox>
        <asp:CompareValidator ID="CompareValidator1" runat="server"
            ControlToValidate="EditUnitPrice"
            ErrorMessage="The price must be greater than or equal to zero and
                           cannot include the currency symbol"
            Operator="GreaterThanEqual" Type="Currency"
            ValueToCompare="0"
            ValidationGroup="EditValidationControls">*
        </asp:CompareValidator>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label2" runat="server"
            Text='<%# Bind("UnitPrice", "{0:c}") %>'>
        </asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

At this point the edit-specific validation controls fire only when the GridView's Update button is clicked and the insert-specific validation controls fire only when the DetailsView's Insert button is clicked, resolving the problem highlighted by Figure 13. However, with this change our ValidationSummary control no longer displays when entering invalid data. The ValidationSummary control also contains a `ValidationGroup` property and only shows summary information for those validation controls in its validation group. Therefore, we need to have two validation controls in this page, one for the `InsertValidationControls` validation group and one for `EditValidationControls`.

```
<asp:ValidationSummary ID="ValidationSummary1" runat="server"
    ShowMessageBox="True" ShowSummary="False"
    ValidationGroup="EditValidationControls" />
<asp:ValidationSummary ID="ValidationSummary2" runat="server"
    ShowMessageBox="True" ShowSummary="False"
    ValidationGroup="InsertValidationControls" />
```

With this addition our tutorial is complete!

Summary

While BoundFields can provide both an inserting and editing interface, the interface is not customizable. Commonly, we want to add validation controls to the editing and inserting interface to ensure that the user enters required inputs in a legal format. To accomplish this we must convert the BoundFields into TemplateFields and add the validation controls to the appropriate template(s). In this tutorial we extended the example from the

Examining the Events Associated with Inserting, Updating, and Deleting tutorial, adding validation controls to both the DetailsView's inserting interface and the GridView's editing interface. Moreover, we saw how to display summary validation information using the ValidationSummary control and how to partition the validation controls on the page into distinct validation groups.

As we saw in this tutorial, TemplateFields allow the editing and inserting interfaces to be augmented to include validation controls. TemplateFields can also be extended to include additional input Web controls, enabling the TextBox to be replaced by a more suitable Web control. In our next tutorial we'll see how to replace the TextBox control with a data-bound DropDownList control, which is ideal when editing a foreign key (such as `CategoryID` or `SupplierID` in the `Products` table).

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Liz Shulok and Zack Jones. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Customizing the Data Modification Interface (C#)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll look at how to customize the interface of an editable GridView, by replacing the standard TextBox and CheckBox controls with alternative input Web controls.

Introduction

The BoundFields and CheckBoxFields used by the GridView and DetailsView controls simplify the process of modifying data due to their ability to render read-only, editable, and insertable interfaces. These interfaces can be rendered without the need for adding any additional declarative markup or code. However, the BoundField and CheckBoxField's interfaces lack the customizability often needed in real-world scenarios. In order to customize the editable or insertable interface in a GridView or DetailsView we need to instead use a TemplateField.

In the preceding tutorial we saw how to customize the data modification interfaces by adding validation Web controls. In this tutorial we'll look at how to customize the actual data collection Web controls, replacing the BoundField and CheckBoxField's standard TextBox and CheckBox controls with alternative input Web controls. In particular, we'll build an editable GridView that allows a product's name, category, supplier, and discontinued status to be updated. When editing a particular row, the category and supplier fields will render as DropDownLists, containing the set of available categories and suppliers to choose from. Furthermore, we'll replace the CheckBoxField's default CheckBox with a RadioButtonList control that offers two options: "Active" and "Discontinued".

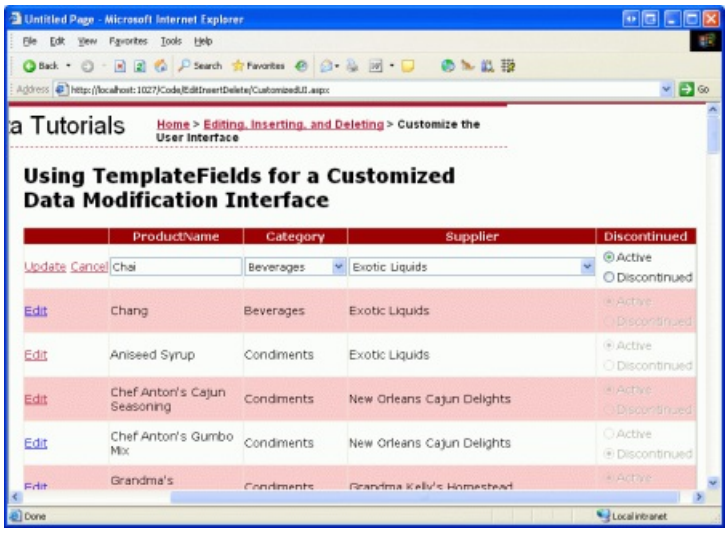


Figure 1: The GridView's Editing Interface Includes DropDownLists and RadioButtons (Click to view full-size image)

Step 1: Creating the Appropriate `UpdateProduct` Overload

In this tutorial we will build an editable GridView that permits editing of a product's name, category, supplier, and discontinued status. Therefore, we need an `UpdateProduct` overload that accepts five input parameters these four product values plus the `ProductID`. Like in our previous overloads, this one will:

1. Retrieve the product information from the database for the specified `ProductID`,
2. Update the `ProductName`, `CategoryID`, `SupplierID`, and `Discontinued` fields, and
3. Send the update request to the DAL through the TableAdapter's `Update()` method.

For brevity, for this particular overload I've omitted the business rule check that ensures a product being marked as discontinued isn't the only product offered by its supplier. Feel free to add it in if you prefer, or, ideally, refactor out the logic to a separate method.

The following code shows the new `UpdateProduct` overload in the `ProductsBLL` class:

```
[System.ComponentModel.DataObjectMethodAttribute(System.ComponentModel.DataObjectMethodType.Update, false)]
public bool UpdateProduct(string productName, int? categoryID,
    int? supplierID, bool discontinued, int productID)
{
    Northwind.ProductsDataTable products = Adapter.GetProductByProductID(productID);
    if (products.Count == 0)
        // no matching record found, return false
        return false;
    Northwind.ProductsRow product = products[0];
    product.ProductName = productName;
    if (supplierID == null) product.SetSupplierIDNull();
      else product.SupplierID = supplierID.Value;
    if (categoryID == null) product.SetCategoryIDNull();
      else product.CategoryID = categoryID.Value;
    product.Discontinued = discontinued;
    // Update the product record
    int rowsAffected = Adapter.Update(product);
    // Return true if precisely one row was updated, otherwise false
    return rowsAffected == 1;
}
```

Step 2: Crafting the Editable GridView

With the `UpdateProduct` overload added, we're ready to create our editable GridView. Open the `CustomizedUI.aspx` page in the `EditInsertDelete` folder and add a GridView control to the Designer. Next, create a new ObjectDataSource from the GridView's smart tag. Configure the ObjectDataSource to retrieve product information via the `ProductBLL` class's `GetProducts()` method and to update product data using the `UpdateProduct` overload we just created. From the INSERT and DELETE tabs, select (None) from the drop-down lists.

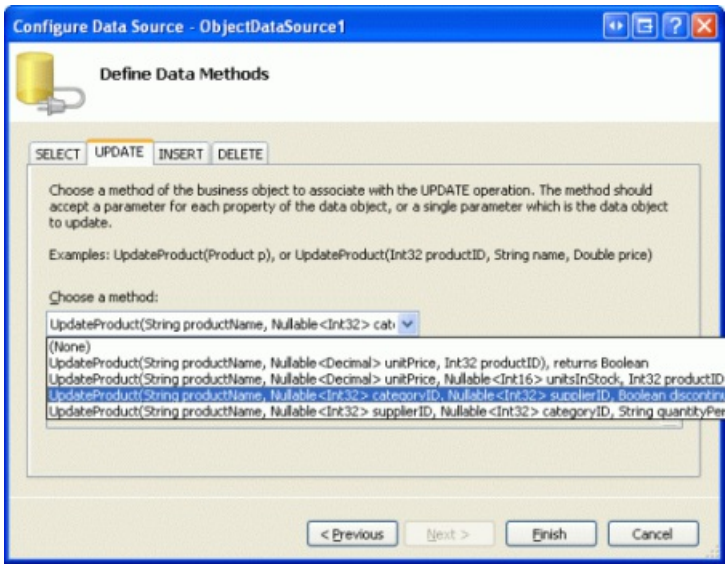


Figure 2: Configure the ObjectDataSource to Use the `UpdateProduct` Overload Just Created (Click to view full-size image)

As we've seen throughout the data modification tutorials, the declarative syntax for the ObjectDataSource created by Visual Studio assigns the `OldValuesParameterFormatString` property to `original_{0}`. This, of course, won't work with our Business Logic Layer since our methods don't expect the original `ProductID` value to be passed in. Therefore, as we've done in previous tutorials, take a moment to remove this property assignment from the declarative syntax or, instead, set this property's value to `{0}`.

After this change, the ObjectDataSource's declarative markup should look like the following:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    SelectMethod="GetProducts" TypeName="ProductsBLL"
    UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

Note that the `OldValuesParameterFormatString` property has been removed and that there is a `Parameter` in the `UpdateParameters` collection for each of the input parameters expected by our `UpdateProduct` overload.

While the ObjectDataSource is configured to update only a subset of product values, the GridView currently shows *all* of the product fields. Take a moment to edit the GridView so that:

- It only includes the `ProductName`, `SupplierName`, `CategoryName` BoundFields and the `Discontinued` CheckBoxField
- The `CategoryName` and `SupplierName` fields to appear before (to the left of) the `Discontinued` CheckBoxField
- The `CategoryName` and `SupplierName` BoundFields' `HeaderText` property is set to "Category" and "Supplier", respectively
- Editing support is enabled (check the Enable Editing checkbox in the GridView's smart tag)

After these changes, the Designer will look similar to Figure 3, with the GridView's declarative syntax shown below.

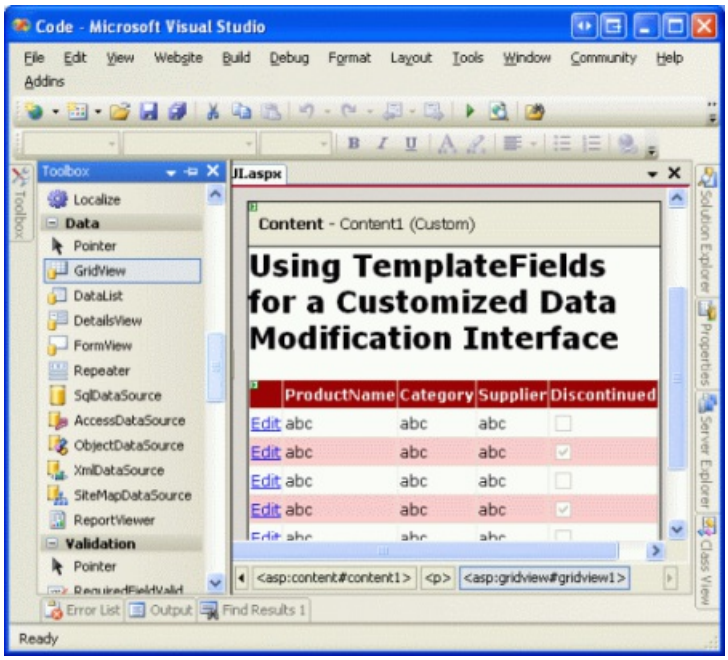


Figure 3: Remove the Unneeded Fields from the GridView (Click to view full-size image)

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:BoundField DataField="ProductName"
           HeaderText="ProductName" SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
           ReadOnly="True"
           SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
           ReadOnly="True"
           SortExpression="SupplierName" />
        <asp:CheckBoxField DataField="Discontinued"
           HeaderText="Discontinued" SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
```

At this point the GridView's read-only behavior is complete. When viewing the data, each product is rendered as a row in the GridView, showing the product's name, category, supplier, and discontinued status.

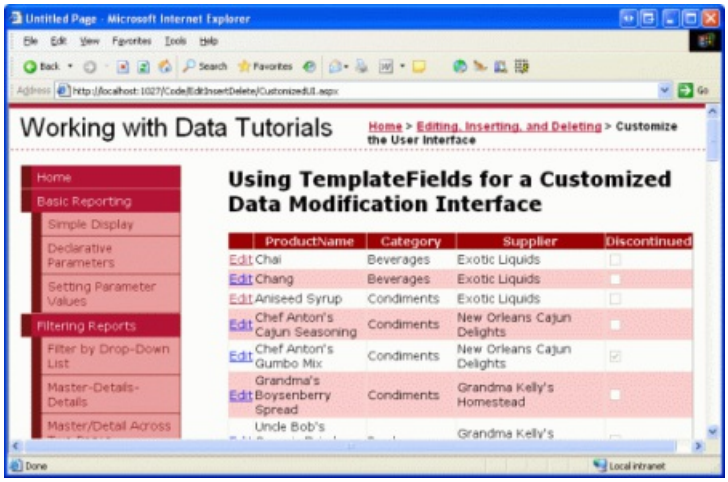


Figure 4: The GridView's Read-Only Interface is Complete (Click to view full-size image)

> **NOTE**
>
> As discussed in An Overview of Inserting, Updating, and Deleting Data tutorial, it is vitally important that the GridView s view state be enabled (the default behavior). If you set the GridView s `EnableViewState` property to `false`, you run the risk of having concurrent users unintentionally deleting or editing records. See WARNING: Concurrency Issue with ASP.NET 2.0 GridViews/DetailsView/FormViews that Support Editing and/or Deleting and Whose View State is Disabled for more information.

Step 3: Using a DropDownList for the Category and Supplier Editing Interfaces

Recall that the `ProductsRow` object contains `CategoryID`, `CategoryName`, `SupplierID`, and `SupplierName` properties, which provide the actual foreign-key ID values in the `Products` database table and the corresponding `Name` values in the `Categories` and `Suppliers` tables. The `ProductRow`'s `CategoryID` and `SupplierID` can both be read from and written to, while the `CategoryName` and `SupplierName` properties are marked read-only.

Due to the read-only status of the `CategoryName` and `SupplierName` properties, the corresponding BoundFields have had their `ReadOnly` property set to `true`, preventing these values from being modified when a row is edited. While we can set the `ReadOnly` property to `false`, rendering the `CategoryName` and `SupplierName` BoundFields as TextBoxes during editing, such an approach will result in an exception when the user attempts to

update the product since there is no `UpdateProduct` overload that takes in `CategoryName` and `SupplierName` inputs. In fact, we don't want to create such an overload for two reasons:

- The `Products` table doesn't have `SupplierName` or `CategoryName` fields, but `SupplierID` and `CategoryID`. Therefore, we want our method to be passed these particular ID values, not their lookup tables' values.
- Requiring the user to type in the name of the supplier or category is less than ideal, as it requires the user to know the available categories and suppliers and their correct spellings.

The supplier and category fields should display the category and suppliers' names when in read-only mode (as it does now) and a drop-down list of applicable options when being edited. Using a drop-down list, the end user can quickly see what categories and suppliers are available to choose among and can more easily make their selection.

To provide this behavior, we need to convert the `SupplierName` and `CategoryName` BoundFields into TemplateFields whose `ItemTemplate` emits the `SupplierName` and `CategoryName` values and whose `EditItemTemplate` uses a DropDownList control to list the available categories and suppliers.

Adding the `Categories` and `Suppliers` DropDownLists

Start by converting the `SupplierName` and `CategoryName` BoundFields into TemplateFields by: clicking on the Edit Columns link from the GridView's smart tag; selecting the BoundField from the list in the lower left; and clicking the "Convert this field into a TemplateField" link. The conversion process will create a TemplateField with both an `ItemTemplate` and an `EditItemTemplate`, as shown in the declarative syntax below:

```
<asp:TemplateField HeaderText="Category" SortExpression="CategoryName">
    <EditItemTemplate>
        <asp:Label ID="Label1" runat="server"
          Text='<%# Eval("CategoryName") %>'></asp:Label>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
          Text='<%# Bind("CategoryName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

Since the BoundField was marked as read-only, both the `ItemTemplate` and `EditItemTemplate` contain a Label Web control whose `Text` property is bound to the applicable data field (`CategoryName`, in the syntax above). We need to modify the `EditItemTemplate`, replacing the Label Web control with a DropDownList control.

As we've seen in previous tutorials, the template can be edited through the Designer or directly from the declarative syntax. To edit it through the Designer, click on the Edit Templates link from the GridView's smart tag and choose to work with the Category field's `EditItemTemplate`. Remove the Label Web control and replace it with a DropDownList control, setting the DropDownList's ID property to `Categories`.

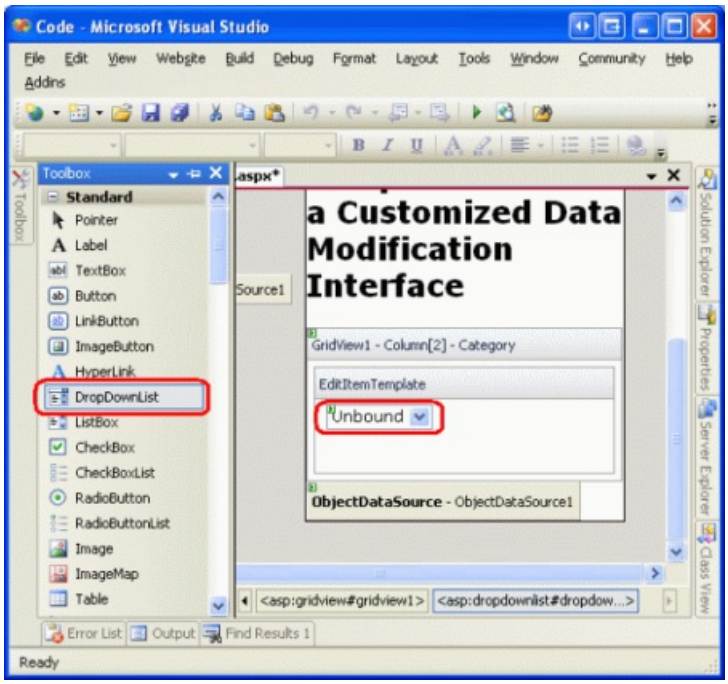


Figure 5: Remove the TexBox and Add a DropDownList to the `EditItemTemplate` (Click to view full-size image)

We next need to populate the DropDownList with the available categories. Click on the Choose Data Source link from the DropDownList's smart tag and opt to create a new ObjectDataSource named `CategoriesDataSource`.

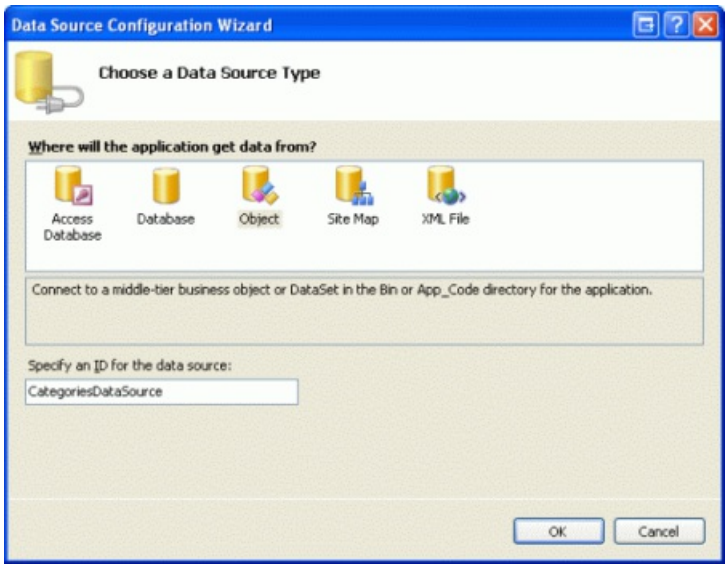


Figure 6: Create a New ObjectDataSource Control Named `CategoriesDataSource` (Click to view full-size image)

To have this ObjectDataSource return all of the categories, bind it to the `CategoriesBLL` class's `GetCategories()` method.

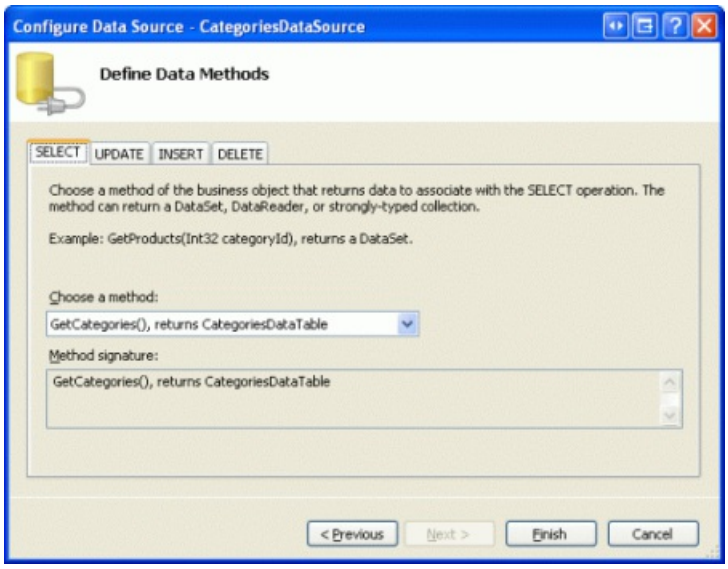


Figure 7: Bind the ObjectDataSource to the `CategoriesBLL`'s `GetCategories()` Method (Click to view full-size image)

Finally, configure the DropDownList's settings such that the `CategoryName` field is displayed in each DropDownList `ListItem` with the `CategoryID` field used as the value.

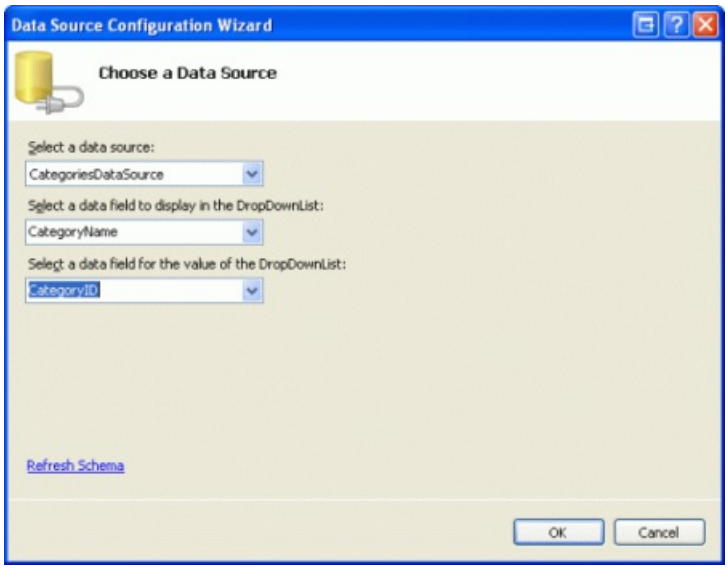


Figure 8: Have the `CategoryName` Field Displayed and the `CategoryID` Used as the Value (Click to view full-size image)

After making these changes the declarative markup for the `EditItemTemplate` in the `CategoryName` TemplateField will include both a DropDownList and an ObjectDataSource:

```
<asp:TemplateField HeaderText="Category" SortExpression="CategoryName">
    <EditItemTemplate>
        <asp:DropDownList ID="Categories" runat="server"
          DataSourceID="CategoriesDataSource"
          DataTextField="CategoryName" DataValueField="CategoryID">
        </asp:DropDownList>
        <asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
            OldValuesParameterFormatString="original_{0}"
            SelectMethod="GetCategories" TypeName="CategoriesBLL">
        </asp:ObjectDataSource>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
          Text='<%# Bind("CategoryName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

NOTE

The DropDownList in the `EditItemTemplate` must have its view state enabled. We will soon add databinding syntax to the DropDownList's declarative syntax and databinding commands like `Eval()` and `Bind()` can only appear in controls whose view state is enabled.

Repeat these steps to add a DropDownList named `Suppliers` to the `SupplierName` TemplateField's `EditItemTemplate`. This will involve adding a DropDownList to the `EditItemTemplate` and creating another ObjectDataSource. The `Suppliers` DropDownList's ObjectDataSource, however, should be configured to invoke the `SuppliersBLL` class's `GetSuppliers()` method. Additionally, configure the `Suppliers` DropDownList to display the `CompanyName` field and use the `SupplierID` field as the value for its `ListItem` s.

After adding the DropDownLists to the two `EditItemTemplate` s, load the page in a browser and click the Edit button for the Chef Anton's Cajun Seasoning product. As Figure 9 shows, the product's category and supplier columns are rendered as drop-down lists containing the available categories and suppliers to choose from. However, note that the *first* items in both drop-down lists are selected by default (Beverages for the category and Exotic Liquids as the supplier), even though Chef Anton's Cajun Seasoning is a Condiment supplied by New Orleans Cajun Delights.

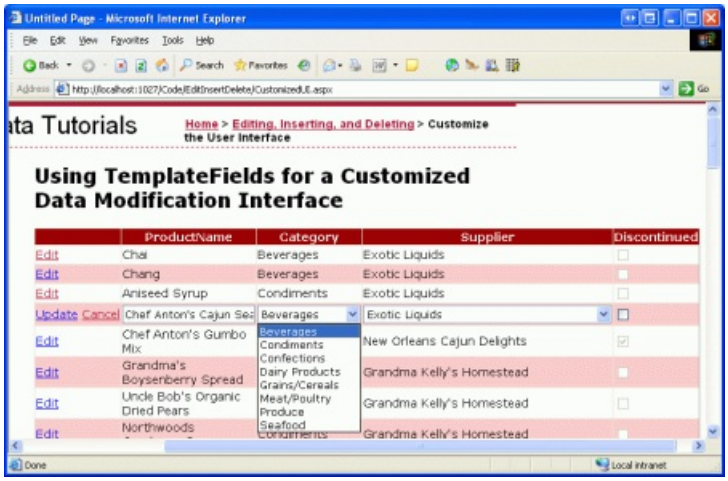


Figure 9: The First Item in the Drop-Down Lists is Selected by Default (Click to view full-size image)

Furthermore, if you click Update, you'll find that the product's `CategoryID` and `SupplierID` values are set to `NULL`. Both of these undesired behaviors are caused because the DropDownLists in the `EditItemTemplate` s are not bound to any data fields from the underlying product data.

Binding the DropDownLists to the `CategoryID` and `SupplierID` Data Fields

In order to have the edited product's category and supplier drop-down lists set to the appropriate values and to have these values sent back to the BLL's `UpdateProduct` method upon clicking Update, we need to bind the DropDownLists' `SelectedValue` properties to the `CategoryID` and `SupplierID` data fields using two-way databinding. To accomplish this with the `Categories` DropDownList, you can add `SelectedValue='<%# Bind("CategoryID") %>'` directly to the declarative syntax.

Alternatively, you can set the DropDownList's databindings by editing the template through the Designer and clicking the Edit DataBindings link from the DropDownList's smart tag. Next, indicate that the `SelectedValue` property should be bound to the `CategoryID` field using two-way databinding (see Figure 10). Repeat either the declarative or Designer process to bind the `SupplierID` data field to the `Suppliers` DropDownList.

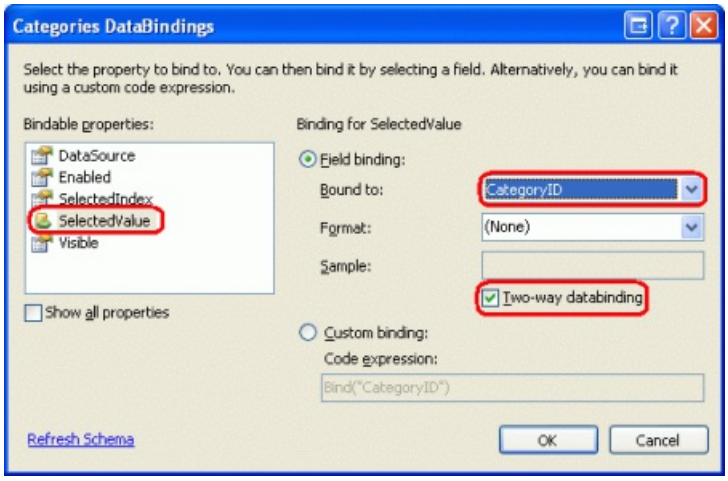


Figure 10: Bind the `CategoryID` to the DropDownList's `SelectedValue` Property Using Two-Way Databinding (Click to view full-size image)

Once the bindings have been applied to the `SelectedValue` properties of the two DropDownLists, the edited product's category and supplier columns will default to the current product's values. Upon clicking Update, the `CategoryID` and `SupplierID` values of the selected drop-down list item will be passed to the `UpdateProduct` method. Figure 11 shows the tutorial after the databinding statements have been added; note how the selected drop-down list items for Chef Anton's Cajun Seasoning are correctly Condiment and New Orleans Cajun Delights.

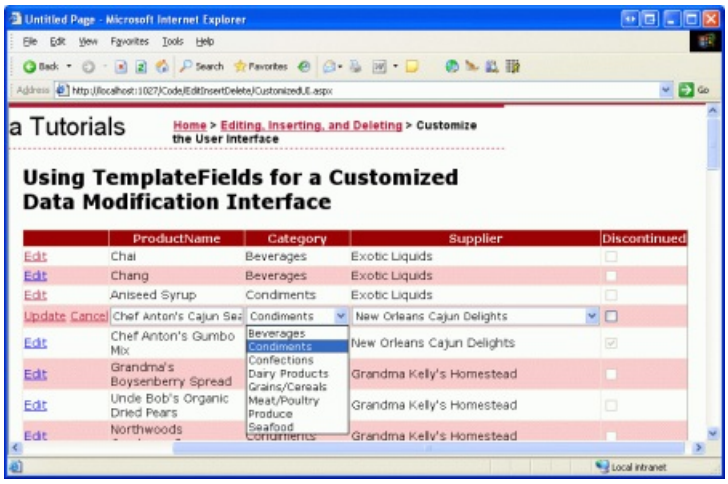


Figure 11: The Edited Product's Current Category and Supplier Values are Selected by Default (Click to view full-size image)

Handling `NULL` Values

The `CategoryID` and `SupplierID` columns in the `Products` table can be `NULL`, yet the DropDownLists in the `EditItemTemplate` s don't include a list item to represent a `NULL` value. This has two consequences:

- User cannot use our interface to change a product's category or supplier from a non-`NULL` value to a `NULL` one
- If a product has a `NULL` `CategoryID` or `SupplierID`, clicking the Edit button will result in an exception. This is because the `NULL` value returned by `CategoryID` (or `SupplierID`) in the `Bind()` statement does not map to a value in the DropDownList (the DropDownList throws an exception when its `SelectedValue` property is set to a value *not* in its collection of list items).

In order to support `NULL` `CategoryID` and `SupplierID` values, we need to add another `ListItem` to each DropDownList to represent the `NULL` value. In the Master/Detail Filtering With a DropDownList tutorial, we saw how to add an additional `ListItem` to a databound DropDownList, which involved setting the DropDownList's `AppendDataBoundItems` property to `true` and manually adding the additional `ListItem`. In that previous tutorial, however, we added a `ListItem` with a `Value` of `-1`. The databinding logic in ASP.NET, however, will automatically convert a blank string to a `NULL` value and vice-a-versa. Therefore, for this tutorial we want the `ListItem`'s `Value` to be an empty string.

Start by setting both DropDownLists' `AppendDataBoundItems` property to `true`. Next, add the `NULL` `ListItem` by adding the following `<asp:ListItem>` element to each DropDownList so that the declarative markup looks like:

```
<asp:DropDownList ID="Categories" runat="server"
    DataSourceID="CategoriesDataSource" DataTextField="CategoryName"
    DataValueField="CategoryID" SelectedValue='<%# Bind("CategoryID") %>'
    AppendDataBoundItems="True">
    <asp:ListItem Value="">(None)</asp:ListItem>
</asp:DropDownList>
```

I've chosen to use "(None)" as the Text value for this `ListItem`, but you can change it to also be a blank string if you'd like.

> **NOTE**
> As we saw in the *Master/Detail Filtering With a DropDownList* tutorial, `ListItem` s can be added to a DropDownList through the Designer by clicking on the DropDownList's `Items` property in the Properties window (which will display the `ListItem` Collection Editor). However, be sure to add the `NULL` `ListItem` for this tutorial through the declarative syntax. If you use the `ListItem` Collection Editor, the generated declarative syntax will omit the `Value` setting altogether when assigned a blank string, creating declarative markup like: `<asp:ListItem>(None)</asp:ListItem>`. While this may look harmless, the missing Value causes the DropDownList to use the `Text` property value in its place. That means that if this `NULL` `ListItem` is selected, the value "(None)" will be attempted to be assigned to the `CategoryID`, which will result in an exception. By explicitly setting `Value=""`, a `NULL` value will be assigned to `CategoryID` when the `NULL` `ListItem` is selected.

Repeat these steps for the Suppliers DropDownList.

With this additional `ListItem`, the editing interface can now assign `NULL` values to a Product's `CategoryID` and `SupplierID` fields, as shown in Figure 12.

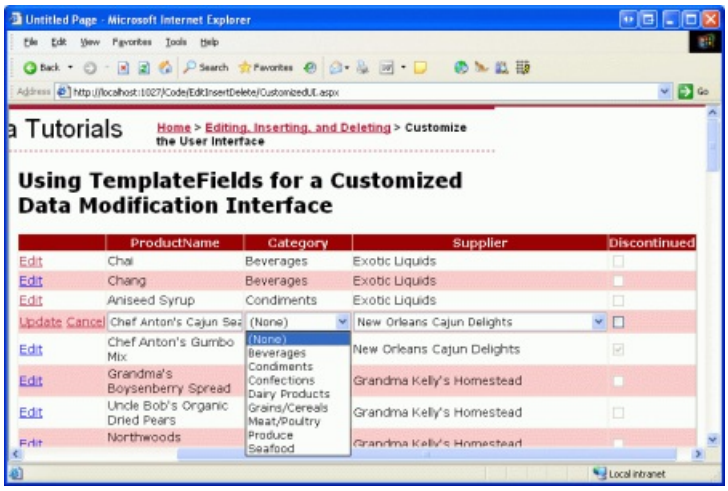


Figure 12: Choose (None) to Assign a `NULL` Value for a Product's Category or Supplier (Click to view full-size image)

Step 4: Using RadioButtons for the Discontinued Status

Currently the products' `Discontinued` data field is expressed using a CheckBoxField, which renders a disabled checkbox for the read-only rows and an enabled checkbox for the row being edited. While this user interface is often suitable, we can customize it if needed using a TemplateField. For this tutorial, let's change the CheckBoxField into a TemplateField that uses a RadioButtonList control with two options "Active" and "Discontinued" from which the user can specify the product's `Discontinued` value.

Start by converting the `Discontinued` CheckBoxField into a TemplateField, which will create a TemplateField with an `ItemTemplate` and `EditItemTemplate`. Both templates include a CheckBox with its `Checked` property bound to the `Discontinued` data field, the only difference between the two being that the `ItemTemplate`'s CheckBox's `Enabled` property is set to `false`.

Replace the CheckBox in both the `ItemTemplate` and `EditItemTemplate` with a RadioButtonList control, setting both RadioButtonLists' `ID` properties to `DiscontinuedChoice`. Next, indicate that the RadioButtonLists should each contain two radio buttons, one labeled "Active" with a value of "False" and one labeled "Discontinued" with a value of "True". To accomplish this you can either enter the `<asp:ListItem>` elements in directly through the declarative syntax or use the `ListItem` Collection Editor from the Designer. Figure 13 shows the `ListItem` Collection Editor after the two radio button options have been specified.

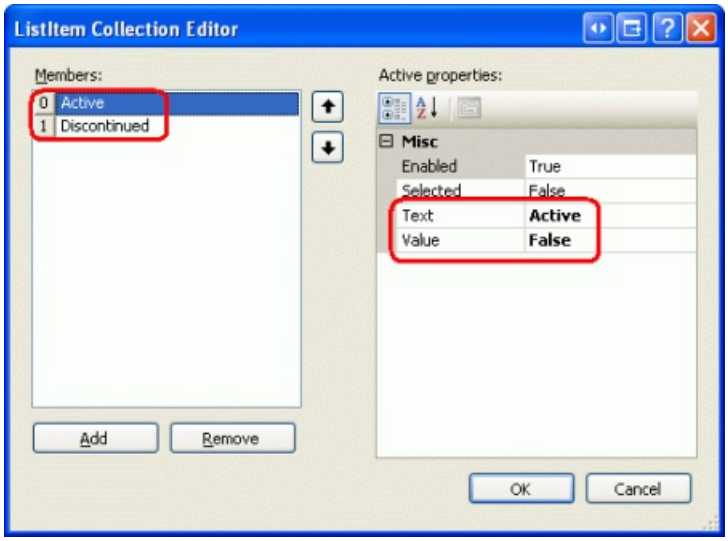


Figure 13: Add "Active" and "Discontinued" Options to the RadioButtonList (Click to view full-size image)

Since the RadioButtonList in the `ItemTemplate` shouldn't be editable, set its `Enabled` property to `false`, leaving

the `Enabled` property to `true` (the default) for the RadioButtonList in the `EditItemTemplate`. This will make the radio buttons in the non-edited row as read-only, but will allow the user to change the RadioButton values for the edited row.

We still need to assign the RadioButtonList controls' `SelectedValue` properties so that the appropriate radio button is selected based upon the product's `Discontinued` data field. As with the DropDownLists examined earlier in this tutorial, this databinding syntax can either be added directly into the declarative markup or through the Edit DataBindings link in the RadioButtonLists' smart tags.

After adding the two RadioButtonLists and configuring them, the `Discontinued` TemplateField's declarative markup should look like:

```
<asp:TemplateField HeaderText="Discontinued" SortExpression="Discontinued">
    <ItemTemplate>
        <asp:RadioButtonList ID="DiscontinuedChoice" runat="server"
          Enabled="False" SelectedValue='<%# Bind("Discontinued") %>'>
            <asp:ListItem Value="False">Active</asp:ListItem>
            <asp:ListItem Value="True">Discontinued</asp:ListItem>
        </asp:RadioButtonList>
    </ItemTemplate>
    <EditItemTemplate>
        <asp:RadioButtonList ID="DiscontinuedChoice" runat="server"
            SelectedValue='<%# Bind("Discontinued") %>'>
            <asp:ListItem Value="False">Active</asp:ListItem>
            <asp:ListItem Value="True">Discontinued</asp:ListItem>
        </asp:RadioButtonList>
    </EditItemTemplate>
</asp:TemplateField>
```

With these changes, the `Discontinued` column has been transformed from a list of checkboxes to a list of radio button pairs (see Figure 14). When editing a product, the appropriate radio button is selected and the product's discontinued status can be updated by selecting the other radio button and clicking Update.

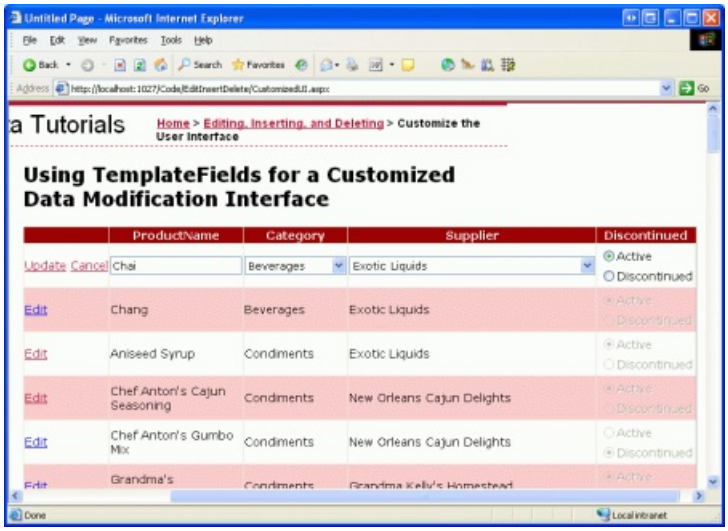


Figure 14: The Discontinued CheckBoxes Have Been Replaced by Radio Button Pairs (Click to view full-size image)

> **NOTE**
>
> Since the `Discontinued` column in the `Products` database cannot have `NULL` values, we do not need to worry about capturing `NULL` information in the interface. If, however, `Discontinued` column could contain `NULL` values we'd want to add a third radio button to the list with its `Value` set to an empty string (`Value=""`), just like with the category and supplier DropDownLists.

Summary

While the BoundField and CheckBoxField automatically render read-only, editing, and inserting interfaces, they lack the ability for customization. Often, though, we'll need to customize the editing or inserting interface, perhaps adding validation controls (as we saw in the preceding tutorial) or by customizing the data collection user interface (as we saw in this tutorial). Customizing the interface with a TemplateField can be summed up in the following steps:

1. Add a TemplateField or convert an existing BoundField or CheckBoxField into a TemplateField
2. Augment the interface as needed
3. Bind the appropriate data fields to the newly added Web controls using two-way databinding

In addition to using the built-in ASP.NET Web controls, you can also customize the templates of a TemplateField with custom, compiled server controls and User Controls.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Implementing Optimistic Concurrency (C#)

2/20/2020 • 38 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> For a web application that allows multiple users to edit data, there is the risk that two users may be editing the same data at the same time. In this tutorial we'll implement optimistic concurrency control to handle this risk.

Introduction

For web applications that only allow users to view data, or for those that include only a single user who can modify data, there's no threat of two concurrent users accidentally overwriting one another's changes. For web applications that allow multiple users to update or delete data, however, there's the potential for one user's modifications to clash with another concurrent user's. Without any concurrency policy in place, when two users are simultaneously editing a single record, the user who commits her changes last will override the changes made by the first.

For example, imagine that two users, Jisun and Sam, were both visiting a page in our application that allowed visitors to update and delete the products through a GridView control. Both click the Edit button in the GridView around the same time. Jisun changes the product name to "Chai Tea" and clicks the Update button. The net result is an `UPDATE` statement that is sent to the database, which sets *all* of the product's updateable fields (even though Jisun only updated one field, `ProductName`). At this point in time, the database has the values "Chai Tea," the category Beverages, the supplier Exotic Liquids, and so on for this particular product. However, the GridView on Sam's screen still shows the product name in the editable GridView row as "Chai". A few seconds after Jisun's changes have been committed, Sam updates the category to Condiments and clicks Update. This results in an `UPDATE` statement sent to the database that sets the product name to "Chai," the `CategoryID` to the corresponding Beverages category ID, and so on. Jisun's changes to the product name have been overwritten. Figure 1 graphically depicts this series of events.

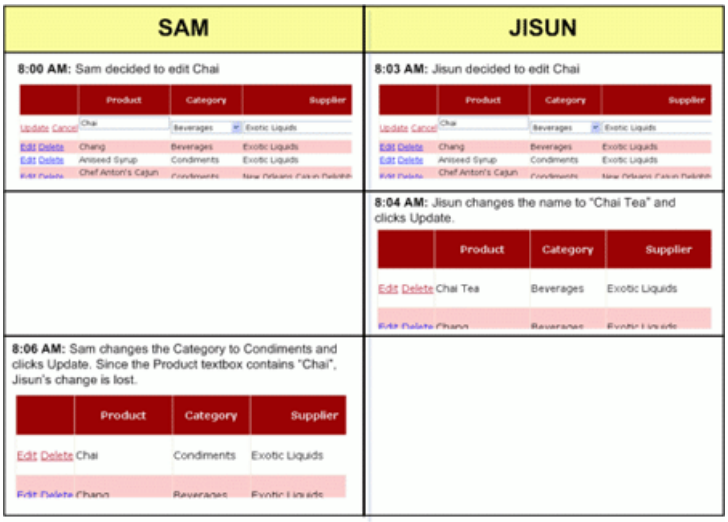


Figure 1: When Two Users Simultaneously Update a Record There s Potential for One User s Changes to Overwrite the Other s (Click to view full-size image)

Similarly, when two users are visiting a page, one user might be in the midst of updating a record when it is

deleted by another user. Or, between when a user loads a page and when they click the Delete button, another user may have modified the contents of that record.

There are three concurrency control strategies available:

- **Do Nothing** -if concurrent users are modifying the same record, let the last commit win (the default behavior)
- **Optimistic Concurrency** - assume that while there may be concurrency conflicts every now and then, the vast majority of the time such conflicts won't arise; therefore, if a conflict does arise, simply inform the user that their changes can't be saved because another user has modified the same data
- **Pessimistic Concurrency** - assume that concurrency conflicts are commonplace and that users won't tolerate being told their changes weren't saved due to another user's concurrent activity; therefore, when one user starts updating a record, lock it, thereby preventing any other users from editing or deleting that record until the user commits their modifications

All of our tutorials thus far have used the default concurrency resolution strategy - namely, we've let the last write win. In this tutorial we'll examine how to implement optimistic concurrency control.

> **NOTE**
>
> We won't look at pessimistic concurrency examples in this tutorial series. Pessimistic concurrency is rarely used because such locks, if not properly relinquished, can prevent other users from updating data. For example, if a user locks a record for editing and then leaves for the day before unlocking it, no other user will be able to update that record until the original user returns and completes his update. Therefore, in situations where pessimistic concurrency is used, there's typically a timeout that, if reached, cancels the lock. Ticket sales websites, which lock a particular seating location for short period while the user completes the order process, is an example of pessimistic concurrency control.

Step 1: Looking at How Optimistic Concurrency is Implemented

Optimistic concurrency control works by ensuring that the record being updated or deleted has the same values as it did when the updating or deleting process started. For example, when clicking the Edit button in an editable GridView, the record's values are read from the database and displayed in TextBoxes and other Web controls. These original values are saved by the GridView. Later, after the user makes her changes and clicks the Update button, the original values plus the new values are sent to the Business Logic Layer, and then down to the Data Access Layer. The Data Access Layer must issue a SQL statement that will only update the record if the original values that the user started editing are identical to the values still in the database. Figure 2 depicts this sequence of events.

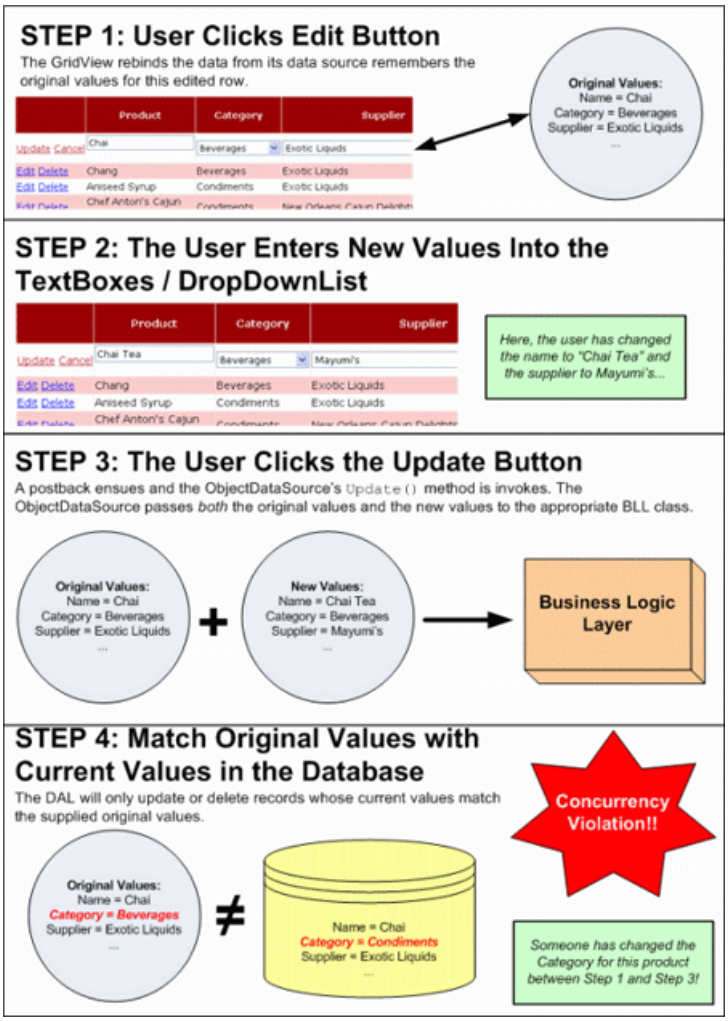


Figure 2: For the Update or Delete to Succeed, the Original Values Must Be Equal to the Current Database Values (Click to view full-size image)

There are various approaches to implementing optimistic concurrency (see Peter A. Bromberg's Optimistic Concurrency Updating Logic for a brief look at a number of options). The ADO.NET Typed DataSet provides one implementation that can be configured with just the tick of a checkbox. Enabling optimistic concurrency for a TableAdapter in the Typed DataSet augments the TableAdapter's `UPDATE` and `DELETE` statements to include a comparison of all of the original values in the `WHERE` clause. The following `UPDATE` statement, for example, updates the name and price of a product only if the current database values are equal to the values that were originally retrieved when updating the record in the GridView. The `@ProductName` and `@UnitPrice` parameters contain the new values entered by the user, whereas `@original_ProductName` and `@original_UnitPrice` contain the values that were originally loaded into the GridView when the Edit button was clicked:

```
UPDATE Products SET
    ProductName = @ProductName,
    UnitPrice = @UnitPrice
WHERE
    ProductID = @original_ProductID AND
    ProductName = @original_ProductName AND
    UnitPrice = @original_UnitPrice
```

NOTE

This `UPDATE` statement has been simplified for readability. In practice, the `UnitPrice` check in the `WHERE` clause would be more involved since `UnitPrice` can contain `NULL` s and checking if `NULL = NULL` always returns False (instead you must use `IS NULL`).

In addition to using a different underlying `UPDATE` statement, configuring a TableAdapter to use optimistic concurrency also modifies the signature of its DB direct methods. Recall from our first tutorial, *Creating a Data Access Layer*, that DB direct methods were those that accepts a list of scalar values as input parameters (rather than a strongly-typed DataRow or DataTable instance). When using optimistic concurrency, the DB direct `Update()` and `Delete()` methods include input parameters for the original values as well. Moreover, the code in the BLL for using the batch update pattern (the `Update()` method overloads that accept DataRows and DataTables rather than scalar values) must be changed as well.

Rather than extend our existing DAL's TableAdapters to use optimistic concurrency (which would necessitate changing the BLL to accommodate), let's instead create a new Typed DataSet named `NorthwindOptimisticConcurrency`, to which we'll add a `Products` TableAdapter that uses optimistic concurrency. Following that, we'll create a `ProductsOptimisticConcurrencyBLL` Business Logic Layer class that has the appropriate modifications to support the optimistic concurrency DAL. Once this groundwork has been laid, we'll be ready to create the ASP.NET page.

Step 2: Creating a Data Access Layer That Supports Optimistic Concurrency

To create a new Typed DataSet, right-click on the `DAL` folder within the `App_Code` folder and add a new DataSet named `NorthwindOptimisticConcurrency`. As we saw in the first tutorial, doing so will add a new TableAdapter to the Typed DataSet, automatically launching the TableAdapter Configuration Wizard. In the first screen, we're prompted to specify the database to connect to - connect to the same Northwind database using the `NORTHWNDConnectionString` setting from `Web.config`.

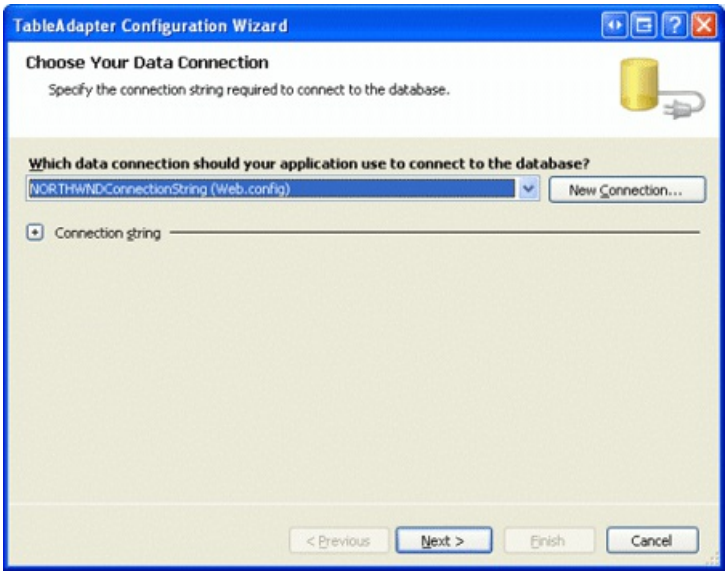


Figure 3: Connect to the Same Northwind Database (Click to view full-size image)

Next, we are prompted as to how to query the data: through an ad-hoc SQL statement, a new stored procedure, or an existing stored procedure. Since we used ad-hoc SQL queries in our original DAL, use this option here as well.

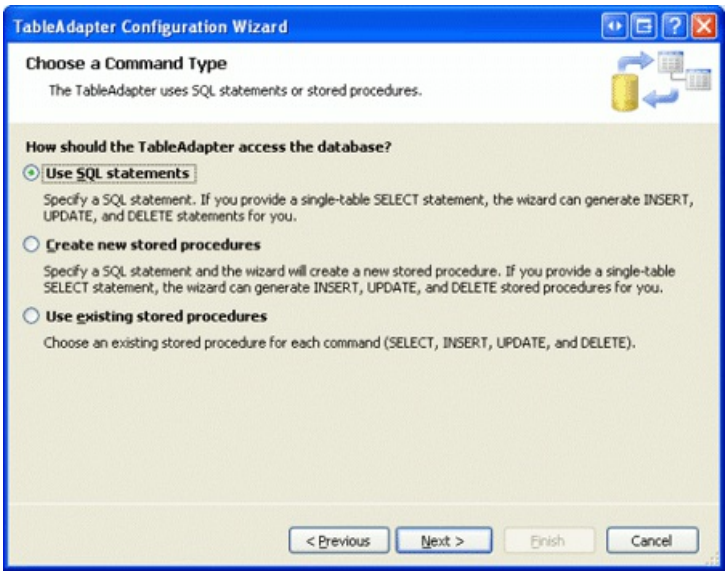


Figure 4: Specify the Data to Retrieve Using an Ad-Hoc SQL Statement (Click to view full-size image)

On the following screen, enter the SQL query to use to retrieve the product information. Let's use the exact same SQL query used for the `Products` TableAdapter from our original DAL, which returns all of the `Product` columns along with the product's supplier and category names:

```
SELECT   ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit,
           UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued,
           (SELECT CategoryName FROM Categories
              WHERE Categories.CategoryID = Products.CategoryID)
              as CategoryName,
           (SELECT CompanyName FROM Suppliers
              WHERE Suppliers.SupplierID = Products.SupplierID)
              as SupplierName
FROM     Products
```

Figure 5: Use the Same SQL Query from the `Products` TableAdapter in the Original DAL (Click to view full-size image)

Before moving onto the next screen, click the Advanced Options button. To have this TableAdapter employ optimistic concurrency control, simply check the "Use optimistic concurrency" checkbox.

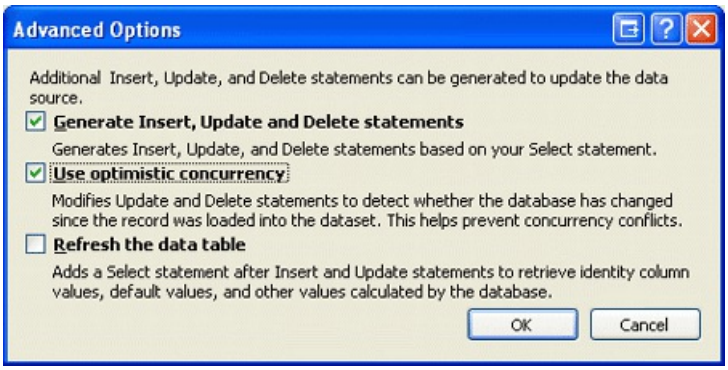


Figure 6: Enable Optimistic Concurrency Control by Checking the "Use optimistic concurrency" CheckBox (Click to view full-size image)

Lastly, indicate that the TableAdapter should use the data access patterns that both fill a DataTable and return a DataTable; also indicate that the DB direct methods should be created. Change the method name for the Return a DataTable pattern from GetData to GetProducts, so as to mirror the naming conventions we used in our original DAL.

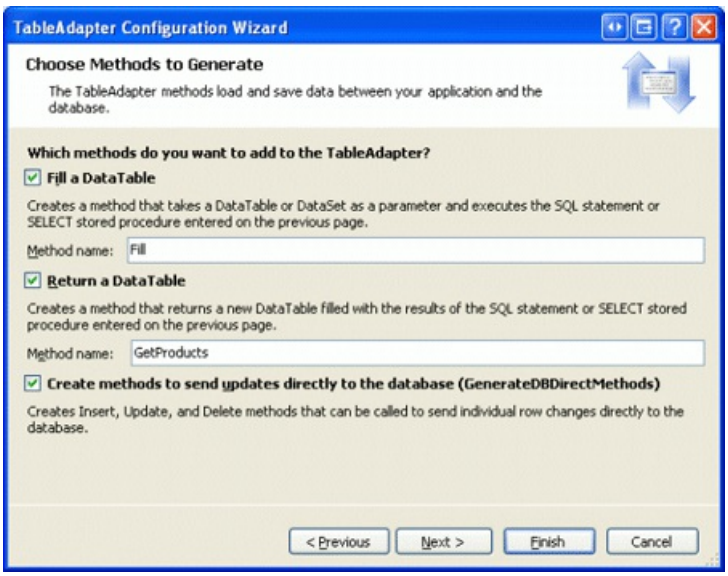


Figure 7: Have the TableAdapter Utilize All Data Access Patterns (Click to view full-size image)

After completing the wizard, the DataSet Designer will include a strongly-typed `Products` DataTable and TableAdapter. Take a moment to rename the DataTable from `Products` to `ProductsOptimisticConcurrency`, which you can do by right-clicking on the DataTable's title bar and choosing Rename from the context menu.

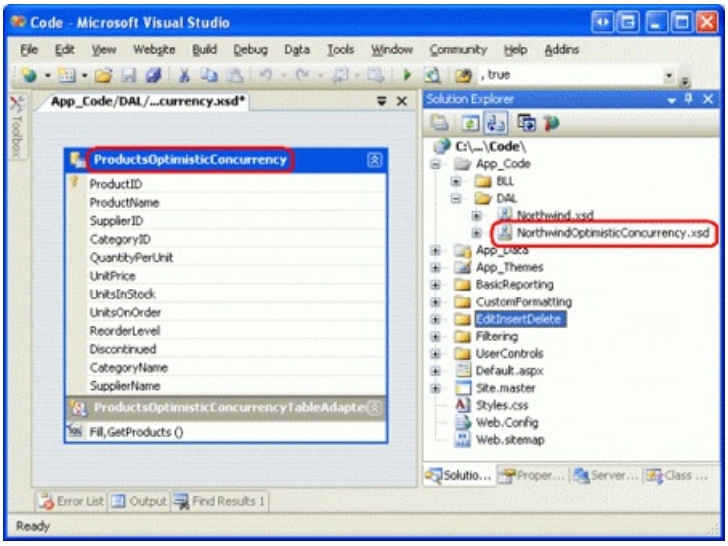


Figure 8: A DataTable and TableAdapter Have Been Added to the Typed DataSet (Click to view full-size image)

To see the differences between the `UPDATE` and `DELETE` queries between the `ProductsOptimisticConcurrency` TableAdapter (which uses optimistic concurrency) and the Products TableAdapter (which doesn't), click on the TableAdapter and go to the Properties window. In the `DeleteCommand` and `UpdateCommand` properties' `CommandText` subproperties you can see the actual SQL syntax that is sent to the database when the DAL's update or delete-related methods are invoked. For the `ProductsOptimisticConcurrency` TableAdapter the `DELETE` statement used is:

```
DELETE FROM [Products]
    WHERE (([ProductID] = @Original_ProductID)
    AND ([ProductName] = @Original_ProductName)
    AND ((@IsNull_SupplierID = 1 AND [SupplierID] IS NULL)
       OR ([SupplierID] = @Original_SupplierID))
    AND ((@IsNull_CategoryID = 1 AND [CategoryID] IS NULL)
       OR ([CategoryID] = @Original_CategoryID))
    AND ((@IsNull_QuantityPerUnit = 1 AND [QuantityPerUnit] IS NULL)
       OR ([QuantityPerUnit] = @Original_QuantityPerUnit))
    AND ((@IsNull_UnitPrice = 1 AND [UnitPrice] IS NULL)
       OR ([UnitPrice] = @Original_UnitPrice))
    AND ((@IsNull_UnitsInStock = 1 AND [UnitsInStock] IS NULL)
       OR ([UnitsInStock] = @Original_UnitsInStock))
    AND ((@IsNull_UnitsOnOrder = 1 AND [UnitsOnOrder] IS NULL)
       OR ([UnitsOnOrder] = @Original_UnitsOnOrder))
    AND ((@IsNull_ReorderLevel = 1 AND [ReorderLevel] IS NULL)
       OR ([ReorderLevel] = @Original_ReorderLevel))
    AND ([Discontinued] = @Original_Discontinued))
```

Whereas the `DELETE` statement for the Product TableAdapter in our original DAL is the much simpler:

```
DELETE FROM [Products] WHERE (([ProductID] = @Original_ProductID))
```

As you can see, the `WHERE` clause in the `DELETE` statement for the TableAdapter that uses optimistic concurrency includes a comparison between each of the `Product` table's existing column values and the original values at the time the GridView (or DetailsView or FormView) was last populated. Since all fields other than `ProductID`, `ProductName`, and `Discontinued` can have `NULL` values, additional parameters and checks are included to correctly compare `NULL` values in the `WHERE` clause.

We won't be adding any additional DataTables to the optimistic concurrency-enabled DataSet for this tutorial, as our ASP.NET page will only provide updating and deleting product information. However, we do still need to add the `GetProductByProductID(productID)` method to the `ProductsOptimisticConcurrency` TableAdapter.

To accomplish this, right-click on the TableAdapter's title bar (the area right above the `Fill` and `GetProducts` method names) and choose Add Query from the context menu. This will launch the TableAdapter Query Configuration Wizard. As with our TableAdapter's initial configuration, opt to create the `GetProductByProductID(productID)` method using an ad-hoc SQL statement (see Figure 4). Since the `GetProductByProductID(productID)` method returns information about a particular product, indicate that this query is a `SELECT` query type that returns rows.

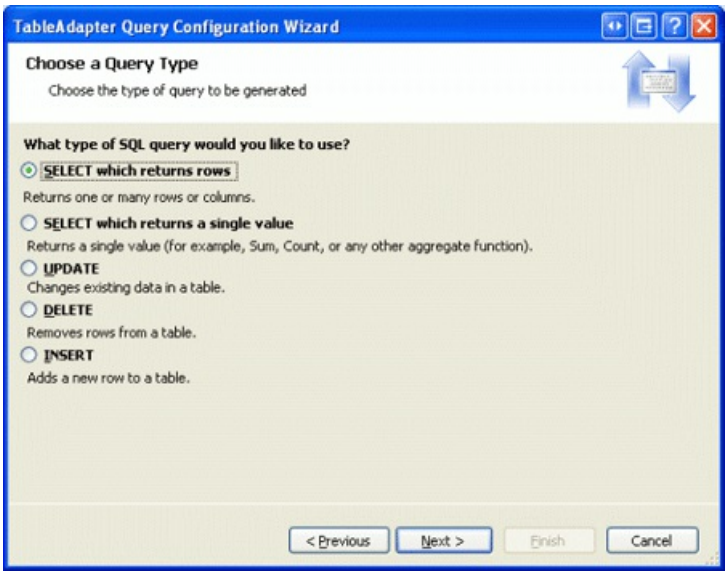


Figure 9: Mark the Query Type as a "`SELECT` which returns rows" (Click to view full-size image)

On the next screen we're prompted for the SQL query to use, with the TableAdapter's default query pre-loaded. Augment the existing query to include the clause `WHERE ProductID = @ProductID`, as shown in Figure 10.

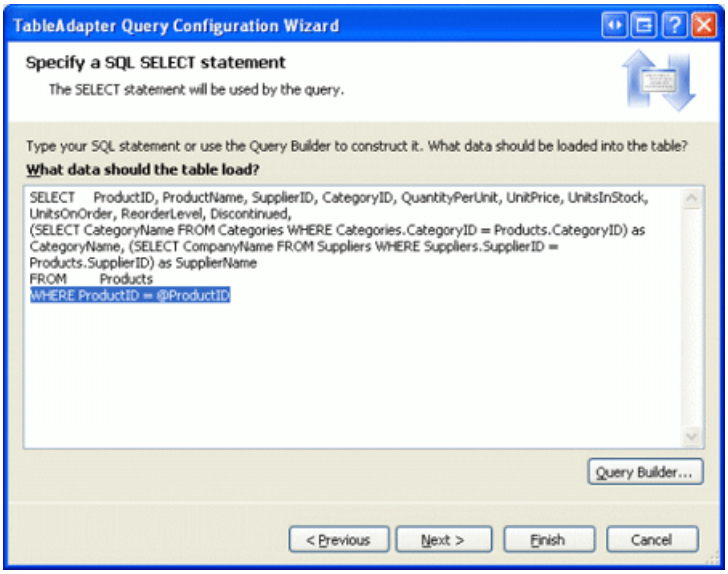


Figure 10: Add a `WHERE` Clause to the Pre-Loaded Query to Return a Specific Product Record (Click to view full-size image)

Finally, change the generated method names to `FillByProductID` and `GetProductByProductID`.

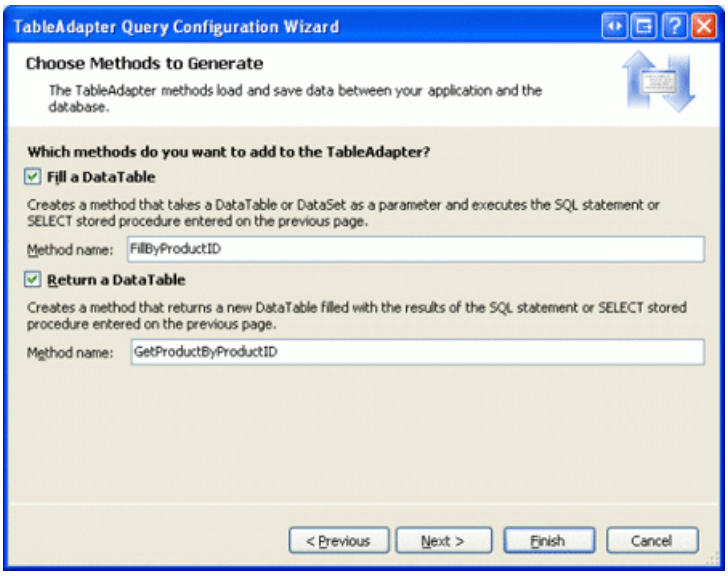


Figure 11: Rename the Methods to `FillByProductID` and `GetProductByProductID` (Click to view full-size image)

With this wizard complete, the TableAdapter now contains two methods for retrieving data: `GetProducts()`, which returns *all* products; and `GetProductByProductID(productID)`, which returns the specified product.

Step 3: Creating a Business Logic Layer for the Optimistic Concurrency-Enabled DAL

Our existing `ProductsBLL` class has examples of using both the batch update and DB direct patterns. The `AddProduct` method and `UpdateProduct` overloads both use the batch update pattern, passing in a `ProductRow` instance to the TableAdapter's Update method. The `DeleteProduct` method, on the other hand, uses the DB direct pattern, calling the TableAdapter's `Delete(productID)` method.

With the new `ProductsOptimisticConcurrency` TableAdapter, the DB direct methods now require that the original values also be passed in. For example, the `Delete` method now expects ten input parameters: the original `ProductID`, `ProductName`, `SupplierID`, `CategoryID`, `QuantityPerUnit`, `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, `ReorderLevel`, and `Discontinued`. It uses these additional input parameters' values in `WHERE` clause of the `DELETE` statement sent to the database, only deleting the specified record if the database's current values map up to the original ones.

While the method signature for the TableAdapter's `Update` method used in the batch update pattern hasn't changed, the code needed to record the original and new values has. Therefore, rather than attempt to use the optimistic concurrency-enabled DAL with our existing `ProductsBLL` class, let's create a new Business Logic Layer class for working with our new DAL.

Add a class named `ProductsOptimisticConcurrencyBLL` to the `BLL` folder within the `App_Code` folder.

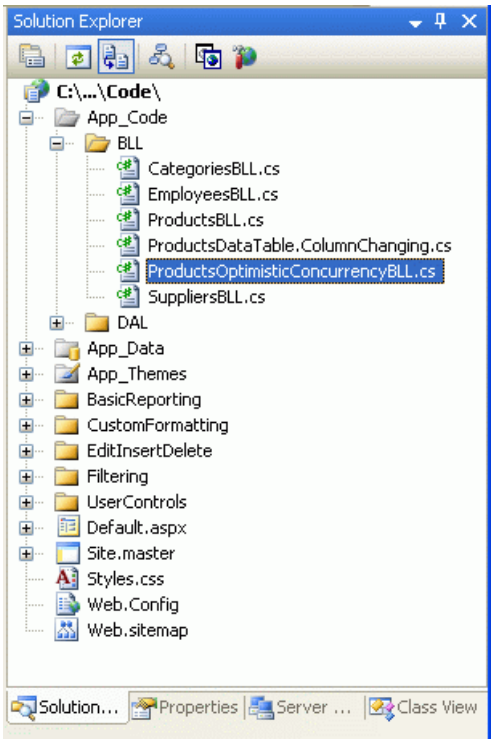


Figure 12: Add the `ProductsOptimisticConcurrencyBLL` Class to the BLL Folder

Next, add the following code to the `ProductsOptimisticConcurrencyBLL` class:

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using NorthwindOptimisticConcurrencyTableAdapters;
[System.ComponentModel.DataObject]
public class ProductsOptimisticConcurrencyBLL
{
    private ProductsOptimisticConcurrencyTableAdapter _productsAdapter = null;
    protected ProductsOptimisticConcurrencyTableAdapter Adapter
    {
        get
        {
            if (_productsAdapter == null)
                _productsAdapter = new ProductsOptimisticConcurrencyTableAdapter();
            return _productsAdapter;
        }
    }
    [System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Select, true)]
    public NorthwindOptimisticConcurrency.ProductsOptimisticConcurrencyDataTable GetProducts()
    {
        return Adapter.GetProducts();
    }
}
```

Note the using `NorthwindOptimisticConcurrencyTableAdapters` statement above the start of the class declaration. The `NorthwindOptimisticConcurrencyTableAdapters` namespace contains the `ProductsOptimisticConcurrencyTableAdapter` class, which provides the DAL's methods. Also before the class declaration you'll find the `System.ComponentModel.DataObject` attribute, which instructs Visual Studio to include this class in the ObjectDataSource wizard's drop-down list.

The `ProductsOptimisticConcurrencyBLL`'s `Adapter` property provides quick access to an instance of the `ProductsOptimisticConcurrencyTableAdapter` class, and follows the pattern used in our original BLL classes (`ProductsBLL`, `CategoriesBLL`, and so on). Finally, the `GetProducts()` method simply calls down into the DAL's `GetProducts()` method and returns a `ProductsOptimisticConcurrencyDataTable` object populated with a `ProductsOptimisticConcurrencyRow` instance for each product record in the database.

Deleting a Product Using the DB Direct Pattern with Optimistic Concurrency

When using the DB direct pattern against a DAL that uses optimistic concurrency, the methods must be passed the new and original values. For deleting, there are no new values, so only the original values need be passed in. In our BLL, then, we must accept all of the original parameters as input parameters. Let's have the `DeleteProduct` method in the `ProductsOptimisticConcurrencyBLL` class use the DB direct method. This means that this method needs to take in all ten product data fields as input parameters, and pass these to the DAL, as shown in the following code:

```
[System.ComponentModel.DataObjectMethodAttribute
(System.ComponentModel.DataObjectMethodType.Delete, true)]
public bool DeleteProduct
    (int original_productID, string original_productName,
    int? original_supplierID, int? original_categoryID,
    string original_quantityPerUnit, decimal? original_unitPrice,
    short? original_unitsInStock, short? original_unitsOnOrder,
    short? original_reorderLevel, bool original_discontinued)
{
    int rowsAffected = Adapter.Delete(original_productID,
                                      original_productName,
                                      original_supplierID,
                                      original_categoryID,
                                      original_quantityPerUnit,
                                      original_unitPrice,
                                      original_unitsInStock,
                                      original_unitsOnOrder,
                                      original_reorderLevel,
                                      original_discontinued);
    // Return true if precisely one row was deleted, otherwise false
    return rowsAffected == 1;
}
```

If the original values - those values that were last loaded into the GridView (or DetailsView or FormView) - differ from the values in the database when the user clicks the Delete button the `WHERE` clause won't match up with any database record and no records will be affected. Hence, the TableAdapter's `Delete` method will return `0` and the BLL's `DeleteProduct` method will return `false`.

Updating a Product Using the Batch Update Pattern with Optimistic Concurrency

As noted earlier, the TableAdapter's `Update` method for the batch update pattern has the same method signature regardless of whether or not optimistic concurrency is employed. Namely, the `Update` method expects a DataRow, an array of DataRows, a DataTable, or a Typed DataSet. There are no additional input parameters for specifying the original values. This is possible because the DataTable keeps track of the original and modified values for its DataRow(s). When the DAL issues its `UPDATE` statement, the `@original_ColumnName` parameters are populated with the DataRow's original values, whereas the `@ColumnName` parameters are populated with the DataRow's modified values.

In the `ProductsBLL` class (which uses our original, non-optimistic concurrency DAL), when using the batch update pattern to update product information our code performs the following sequence of events:

1. Read the current database product information into a `ProductRow` instance using the TableAdapter's `GetProductByProductID(productID)` method
2. Assign the new values to the `ProductRow` instance from Step 1
3. Call the TableAdapter's `Update` method, passing in the `ProductRow` instance

This sequence of steps, however, won't correctly support optimistic concurrency because the `ProductRow` populated in Step 1 is populated directly from the database, meaning that the original values used by the DataRow are those that currently exist in the database, and not those that were bound to the GridView at the start of the editing process. Instead, when using an optimistic concurrency-enabled DAL, we need to alter the `UpdateProduct` method overloads to use the following steps:

1. Read the current database product information into a `ProductsOptimisticConcurrencyRow` instance using the TableAdapter's `GetProductByProductID(productID)` method
2. Assign the *original* values to the `ProductsOptimisticConcurrencyRow` instance from Step 1
3. Call the `ProductsOptimisticConcurrencyRow` instance's `AcceptChanges()` method, which instructs the DataRow that its current values are the "original" ones
4. Assign the *new* values to the `ProductsOptimisticConcurrencyRow` instance
5. Call the TableAdapter's `Update` method, passing in the `ProductsOptimisticConcurrencyRow` instance

Step 1 reads in all of the current database values for the specified product record. This step is superfluous in the `UpdateProduct` overload that updates *all* of the product columns (as these values are overwritten in Step 2), but is essential for those overloads where only a subset of the column values are passed in as input parameters. Once the original values have been assigned to the `ProductsOptimisticConcurrencyRow` instance, the `AcceptChanges()` method is called, which marks the current DataRow values as the original values to be used in the `@original_ColumnName` parameters in the `UPDATE` statement. Next, the new parameter values are assigned to the `ProductsOptimisticConcurrencyRow` and, finally, the `Update` method is invoked, passing in the DataRow.

The following code shows the `UpdateProduct` overload that accepts all product data fields as input parameters. While not shown here, the `ProductsOptimisticConcurrencyBLL` class included in the download for this tutorial also contains an `UpdateProduct` overload that accepts just the product's name and price as input parameters.

```
protected void AssignAllProductValues
    (NorthwindOptimisticConcurrency.ProductsOptimisticConcurrencyRow product,
    string productName, int? supplierID, int? categoryID, string quantityPerUnit,
    decimal? unitPrice, short? unitsInStock, short? unitsOnOrder,
    short? reorderLevel, bool discontinued)
{
    product.ProductName = productName;
    if (supplierID == null)
        product.SetSupplierIDNull();
    else
        product.SupplierID = supplierID.Value;
    if (categoryID == null)
        product.SetCategoryIDNull();
    else
        product.CategoryID = categoryID.Value;
    if (quantityPerUnit == null)
        product.SetQuantityPerUnitNull();
    else
        product.QuantityPerUnit = quantityPerUnit;
    if (unitPrice == null)
        product.SetUnitPriceNull();
    else
        product.UnitPrice = unitPrice.Value;
    if (unitsInStock == null)
```

```
        product.SetUnitsInStockNull();
    else
        product.UnitsInStock = unitsInStock.Value;
    if (unitsOnOrder == null)
        product.SetUnitsOnOrderNull();
    else
        product.UnitsOnOrder = unitsOnOrder.Value;
    if (reorderLevel == null)
        product.SetReorderLevelNull();
    else
        product.ReorderLevel = reorderLevel.Value;
    product.Discontinued = discontinued;
}
[System.ComponentModel.DataObjectMethodAttribute
(System.ComponentModel.DataObjectMethodType.Update, true)]
public bool UpdateProduct(
    // new parameter values
    string productName, int? supplierID, int? categoryID, string quantityPerUnit,
    decimal? unitPrice, short? unitsInStock, short? unitsOnOrder,
    short? reorderLevel, bool discontinued, int productID,
    // original parameter values
    string original_productName, int? original_supplierID, int? original_categoryID,
    string original_quantityPerUnit, decimal? original_unitPrice,
    short? original_unitsInStock, short? original_unitsOnOrder,
    short? original_reorderLevel, bool original_discontinued,
    int original_productID)
{
    // STEP 1: Read in the current database product information
    NorthwindOptimisticConcurrency.ProductsOptimisticConcurrencyDataTable products =
        Adapter.GetProductByProductID(original_productID);
    if (products.Count == 0)
        // no matching record found, return false
        return false;
    NorthwindOptimisticConcurrency.ProductsOptimisticConcurrencyRow product = products[0];
    // STEP 2: Assign the original values to the product instance
    AssignAllProductValues(product, original_productName, original_supplierID,
        original_categoryID, original_quantityPerUnit, original_unitPrice,
        original_unitsInStock, original_unitsOnOrder, original_reorderLevel,
        original_discontinued);
    // STEP 3: Accept the changes
    product.AcceptChanges();
    // STEP 4: Assign the new values to the product instance
    AssignAllProductValues(product, productName, supplierID, categoryID,
        quantityPerUnit, unitPrice, unitsInStock, unitsOnOrder, reorderLevel,
        discontinued);
    // STEP 5: Update the product record
    int rowsAffected = Adapter.Update(product);
    // Return true if precisely one row was updated, otherwise false
    return rowsAffected == 1;
}
```

Step 4: Passing the Original and New Values From the ASP.NET Page to the BLL Methods

With the DAL and BLL complete, all that remains is to create an ASP.NET page that can utilize the optimistic concurrency logic built in to the system. Specifically, the data Web control (the GridView, DetailsView, or FormView) must remember its original values and the ObjectDataSource must pass both sets of values to the Business Logic Layer. Furthermore, the ASP.NET page must be configured to gracefully handle concurrency violations.

Start by opening the `OptimisticConcurrency.aspx` page in the `EditInsertDelete` folder and adding a GridView to the Designer, setting its `ID` property to `ProductsGrid`. From the GridView's smart tag, opt to create a new ObjectDataSource named `ProductsOptimisticConcurrencyDataSource`. Since we want this ObjectDataSource to use

the DAL that supports optimistic concurrency, configure it to use the `ProductsOptimisticConcurrencyBLL` object.

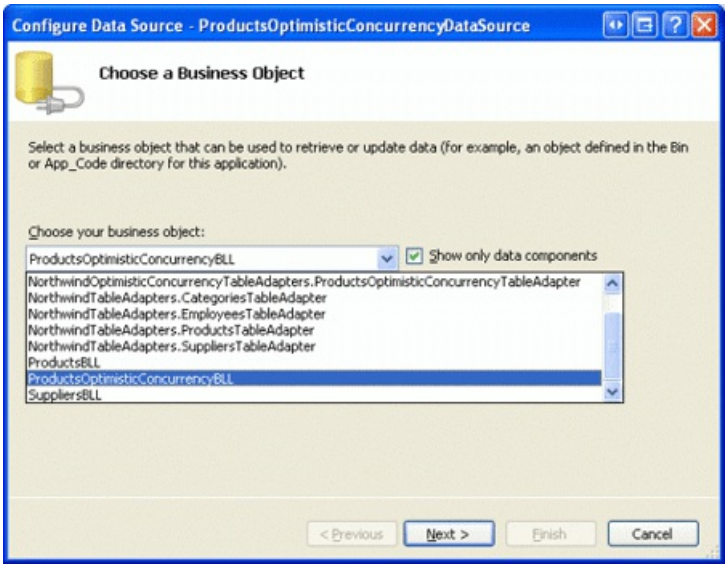


Figure 13: Have the ObjectDataSource Use the `ProductsOptimisticConcurrencyBLL` Object (Click to view full-size image)

Choose the `GetProducts`, `UpdateProduct`, and `DeleteProduct` methods from drop-down lists in the wizard. For the UpdateProduct method, use the overload that accepts all of the product's data fields.

Configuring the ObjectDataSource Control's Properties

After completing the wizard, the ObjectDataSource's declarative markup should look like the following:

```
<asp:ObjectDataSource ID="ProductsOptimisticConcurrencyDataSource" runat="server"
    DeleteMethod="DeleteProduct" OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsOptimisticConcurrencyBLL"
    UpdateMethod="UpdateProduct">
    <DeleteParameters>
        <asp:Parameter Name="original_productID" Type="Int32" />
        <asp:Parameter Name="original_productName" Type="String" />
        <asp:Parameter Name="original_supplierID" Type="Int32" />
        <asp:Parameter Name="original_categoryID" Type="Int32" />
        <asp:Parameter Name="original_quantityPerUnit" Type="String" />
        <asp:Parameter Name="original_unitPrice" Type="Decimal" />
        <asp:Parameter Name="original_unitsInStock" Type="Int16" />
        <asp:Parameter Name="original_unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="original_reorderLevel" Type="Int16" />
        <asp:Parameter Name="original_discontinued" Type="Boolean" />
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="unitsInStock" Type="Int16" />
        <asp:Parameter Name="unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="reorderLevel" Type="Int16" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
        <asp:Parameter Name="productID" Type="Int32" />
        <asp:Parameter Name="original_productName" Type="String" />
        <asp:Parameter Name="original_supplierID" Type="Int32" />
        <asp:Parameter Name="original_categoryID" Type="Int32" />
        <asp:Parameter Name="original_quantityPerUnit" Type="String" />
        <asp:Parameter Name="original_unitPrice" Type="Decimal" />
        <asp:Parameter Name="original_unitsInStock" Type="Int16" />
        <asp:Parameter Name="original_unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="original_reorderLevel" Type="Int16" />
        <asp:Parameter Name="original_discontinued" Type="Boolean" />
        <asp:Parameter Name="original_productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

As you can see, the `DeleteParameters` collection contains a `Parameter` instance for each of the ten input parameters in the `ProductsOptimisticConcurrencyBLL` class's `DeleteProduct` method. Likewise, the `UpdateParameters` collection contains a `Parameter` instance for each of the input parameters in `UpdateProduct`.

For those previous tutorials that involved data modification, we'd remove the ObjectDataSource's `OldValuesParameterFormatString` property at this point, since this property indicates that the BLL method expects the old (or original) values to be passed in as well as the new values. Furthermore, this property value indicates the input parameter names for the original values. Since we are passing in the original values into the BLL, do *not* remove this property.

> **NOTE**
> The value of the `OldValuesParameterFormatString` property must map to the input parameter names in the BLL that expect the original values. Since we named these parameters `original_productName`, `original_supplierID`, and so on, you can leave the `OldValuesParameterFormatString` property value as `original_{0}`. If, however, the BLL methods' input parameters had names like `old_productName`, `old_supplierID`, and so on, you'd need to update the `OldValuesParameterFormatString` property to `old_{0}`.

There's one final property setting that needs to be made in order for the ObjectDataSource to correctly pass the original values to the BLL methods. The ObjectDataSource has a ConflictDetection property that can be assigned

to one of two values:

- `OverwriteChanges` - the default value; does not send the original values to the BLL methods' original input parameters
- `CompareAllValues` - does send the original values to the BLL methods; choose this option when using optimistic concurrency

Take a moment to set the `ConflictDetection` property to `CompareAllValues`.

Configuring the GridView's Properties and Fields

With the ObjectDataSource's properties properly configured, let's turn our attention to setting up the GridView. First, since we want the GridView to support editing and deleting, click the Enable Editing and Enable Deleting checkboxes from the GridView's smart tag. This will add a CommandField whose `ShowEditButton` and `ShowDeleteButton` are both set to `true`.

When bound to the `ProductsOptimisticConcurrencyDataSource` ObjectDataSource, the GridView contains a field for each of the product's data fields. While such a GridView can be edited, the user experience is anything but acceptable. The `CategoryID` and `SupplierID` BoundFields will render as TextBoxes, requiring the user to enter the appropriate category and supplier as ID numbers. There will be no formatting for the numeric fields and no validation controls to ensure that the product's name has been supplied and that the unit price, units in stock, units on order, and reorder level values are both proper numeric values and are greater than or equal to zero.

As we discussed in the *Adding Validation Controls to the Editing and Inserting Interfaces* and *Customizing the Data Modification Interface* tutorials, the user interface can be customized by replacing the BoundFields with TemplateFields. I've modified this GridView and its editing interface in the following ways:

- Removed the `ProductID`, `SupplierName`, and `CategoryName` BoundFields
- Converted the `ProductName` BoundField to a TemplateField and added a RequiredFieldValidation control.
- Converted the `CategoryID` and `SupplierID` BoundFields to TemplateFields, and adjusted the editing interface to use DropDownLists rather than TextBoxes. In these TemplateFields' `ItemTemplates`, the `CategoryName` and `SupplierName` data fields are displayed.
- Converted the `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, and `ReorderLevel` BoundFields to TemplateFields and added CompareValidator controls.

Since we've already examined how to accomplish these tasks in previous tutorials, I'll just list the final declarative syntax here and leave the implementation as practice.

```
<asp:GridView ID="ProductsGrid" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsOptimisticConcurrencyDataSource"
    OnRowUpdated="ProductsGrid_RowUpdated">
    <Columns>
        <asp:CommandField ShowDeleteButton="True" ShowEditButton="True" />
        <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
            <EditItemTemplate>
                <asp:TextBox ID="EditProductName" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:TextBox>
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                    ControlToValidate="EditProductName"
                    ErrorMessage="You must enter a product name."
                    runat="server">*</asp:RequiredFieldValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Category" SortExpression="CategoryName">
            <EditItemTemplate>
```

```
            <asp:DropDownList ID="EditCategoryID" runat="server"
                DataSourceID="CategoriesDataSource" AppendDataBoundItems="true"
                DataTextField="CategoryName" DataValueField="CategoryID"
                SelectedValue='<%# Bind("CategoryID") %>'>
                <asp:ListItem Value=">(None)</asp:ListItem>
            </asp:DropDownList><asp:ObjectDataSource ID="CategoriesDataSource"
                runat="server" OldValuesParameterFormatString="original_{0}"
                SelectMethod="GetCategories" TypeName="CategoriesBLL">
            </asp:ObjectDataSource>
        </EditItemTemplate>
        <ItemTemplate>
            <asp:Label ID="Label2" runat="server"
                Text='<%# Bind("CategoryName") %>'></asp:Label>
        </ItemTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Supplier" SortExpression="SupplierName">
        <EditItemTemplate>
            <asp:DropDownList ID="EditSuppliersID" runat="server"
                DataSourceID="SuppliersDataSource" AppendDataBoundItems="true"
                DataTextField="CompanyName" DataValueField="SupplierID"
                SelectedValue='<%# Bind("SupplierID") %>'>
                <asp:ListItem Value=">(None)</asp:ListItem>
            </asp:DropDownList><asp:ObjectDataSource ID="SuppliersDataSource"
                runat="server" OldValuesParameterFormatString="original_{0}"
                SelectMethod="GetSuppliers" TypeName="SuppliersBLL">
            </asp:ObjectDataSource>
        </EditItemTemplate>
        <ItemTemplate>
            <asp:Label ID="Label3" runat="server"
                Text='<%# Bind("SupplierName") %>'></asp:Label>
        </ItemTemplate>
    </asp:TemplateField>
    <asp:BoundField DataField="QuantityPerUnit" HeaderText="Qty/Unit"
        SortExpression="QuantityPerUnit" />
    <asp:TemplateField HeaderText="Price" SortExpression="UnitPrice">
        <EditItemTemplate>
            <asp:TextBox ID="EditUnitPrice" runat="server"
                Text='<%# Bind("UnitPrice", "{0:N2}") %>' Columns="8" />
            <asp:CompareValidator ID="CompareValidator1" runat="server"
                ControlToValidate="EditUnitPrice"
                ErrorMessage="Unit price must be a valid currency value without the
                currency symbol and must have a value greater than or equal to zero."
                Operator="GreaterThanEqual" Type="Currency"
                ValueToCompare="0">*</asp:CompareValidator>
        </EditItemTemplate>
        <ItemTemplate>
            <asp:Label ID="Label4" runat="server"
                Text='<%# Bind("UnitPrice", "{0:C}") %>'></asp:Label>
        </ItemTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Units In Stock" SortExpression="UnitsInStock">
        <EditItemTemplate>
            <asp:TextBox ID="EditUnitsInStock" runat="server"
                Text='<%# Bind("UnitsInStock") %>' Columns="6"></asp:TextBox>
            <asp:CompareValidator ID="CompareValidator2" runat="server"
                ControlToValidate="EditUnitsInStock"
                ErrorMessage="Units in stock must be a valid number
                    greater than or equal to zero."
                Operator="GreaterThanEqual" Type="Integer"
                ValueToCompare="0">*</asp:CompareValidator>
        </EditItemTemplate>
        <ItemTemplate>
            <asp:Label ID="Label5" runat="server"
                Text='<%# Bind("UnitsInStock", "{0:N0}") %>'></asp:Label>
        </ItemTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Units On Order" SortExpression="UnitsOnOrder">
        <EditItemTemplate>
            <asp:TextBox ID="EditUnitsOnOrder" runat="server"
```

```
                <asp:TextBox ID="EditUnitsOnOrder" runat="server"
                    Text='<%# Bind("UnitsOnOrder") %>' Columns="6"></asp:TextBox>
                <asp:CompareValidator ID="CompareValidator3" runat="server"
                    ControlToValidate="EditUnitsOnOrder"
                    ErrorMessage="Units on order must be a valid numeric value
                        greater than or equal to zero."
                    Operator="GreaterThanEqual" Type="Integer"
                    ValueToCompare="0">*</asp:CompareValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label6" runat="server"
                    Text='<%# Bind("UnitsOnOrder", "{0:N0}") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Reorder Level" SortExpression="ReorderLevel">
            <EditItemTemplate>
                <asp:TextBox ID="EditReorderLevel" runat="server"
                    Text='<%# Bind("ReorderLevel") %>' Columns="6"></asp:TextBox>
                <asp:CompareValidator ID="CompareValidator4" runat="server"
                    ControlToValidate="EditReorderLevel"
                    ErrorMessage="Reorder level must be a valid numeric value
                        greater than or equal to zero."
                    Operator="GreaterThanEqual" Type="Integer"
                    ValueToCompare="0">*</asp:CompareValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label7" runat="server"
                    Text='<%# Bind("ReorderLevel", "{0:N0}") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
```

We're very close to having a fully-working example. However, there are a few subtleties that will creep up and cause us problems. Additionally, we still need some interface that alerts the user when a concurrency violation has occurred.

> **NOTE**
>
> In order for a data Web control to correctly pass the original values to the ObjectDataSource (which are then passed to the BLL), it's vital that the GridView's `EnableViewState` property is set to `true` (the default). If you disable view state, the original values are lost on postback.

Passing the Correct Original Values to the ObjectDataSource

There are a couple of problems with the way the GridView has been configured. If the ObjectDataSource's `ConflictDetection` property is set to `CompareAllValues` (as is ours), when the ObjectDataSource's `Update()` or `Delete()` methods are invoked by the GridView (or DetailsView or FormView), the ObjectDataSource attempts to copy the GridView's original values into its appropriate `Parameter` instances. Refer back to Figure 2 for a graphical representation of this process.

Specifically, the GridView's original values are assigned the values in the two-way databinding statements each time the data is bound to the GridView. Therefore, it's essential that the required original values all are captured via two-way databinding and that they are provided in a convertible format.

To see why this is important, take a moment to visit our page in a browser. As expected, the GridView lists each product with an Edit and Delete button in the leftmost column.

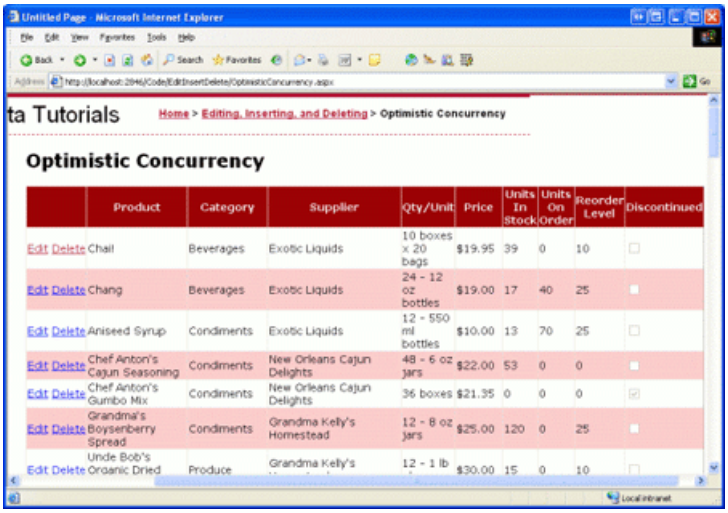


Figure 14: The Products are Listed in a GridView (Click to view full-size image)

If you click the Delete button for any product, a `FormatException` is thrown.

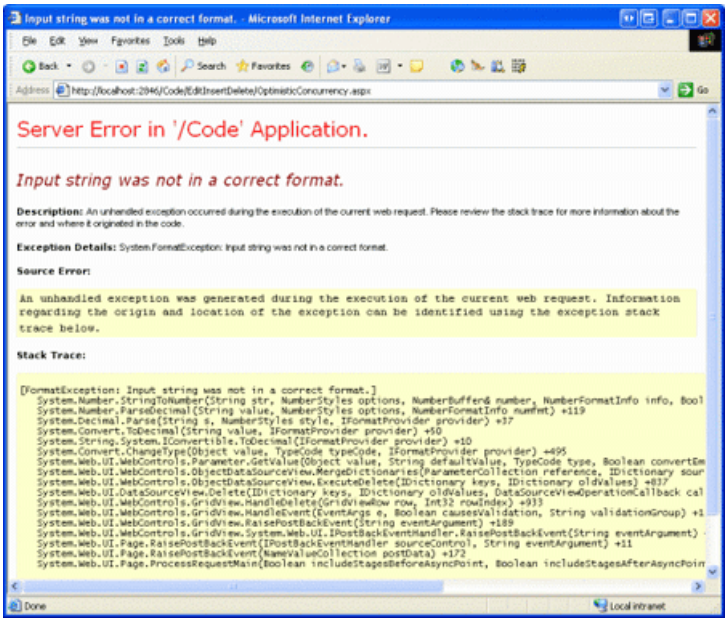


Figure 15: Attempting to Delete Any Product Results in a `FormatException` (Click to view full-size image)

The `FormatException` is raised when the ObjectDataSource attempts to read in the original `UnitPrice` value. Since the `ItemTemplate` has the `UnitPrice` formatted as a currency (`<%# Bind("UnitPrice", "{0:C}") %>`), it includes a currency symbol, like $19.95. The `FormatException` occurs as the ObjectDataSource attempts to convert this string into a `decimal`. To circumvent this problem, we have a number of options:

- Remove the currency formatting from the `ItemTemplate`. That is, instead of using `<%# Bind("UnitPrice", "{0:C}") %>`, simply use `<%# Bind("UnitPrice") %>`. The downside of this is that the price is no longer formatted.
- Display the `UnitPrice` formatted as a currency in the `ItemTemplate`, but use the `Eval` keyword to accomplish this. Recall that `Eval` performs one-way databinding. We still need to provide the `UnitPrice` value for the original values, so we'll still need a two-way databinding statement in the `ItemTemplate`, but this can be placed in a Label Web control whose `Visible` property is set to `false`. We could use the following markup in the ItemTemplate:

```
<ItemTemplate>
    <asp:Label ID="DummyUnitPrice" runat="server"
        Text='<%# Bind("UnitPrice") %>' Visible="false"></asp:Label>
    <asp:Label ID="Label4" runat="server"
        Text='<%# Eval("UnitPrice", "{0:C}") %>'></asp:Label>
</ItemTemplate>
```

- Remove the currency formatting from the `ItemTemplate`, using `<%# Bind("UnitPrice") %>`. In the GridView's `RowDataBound` event handler, programmatically access the Label Web control within which the `UnitPrice` value is displayed and set its `Text` property to the formatted version.
- Leave the `UnitPrice` formatted as a currency. In the GridView's `RowDeleting` event handler, replace the existing original `UnitPrice` value ($19.95) with an actual decimal value using `Decimal.Parse`. We saw how to accomplish something similar in the `RowUpdating` event handler in the *Handling BLL- and DAL-Level Exceptions in an ASP.NET Page* tutorial.

For my example I chose to go with the second approach, adding a hidden Label Web control whose `Text` property is two-way data bound to the unformatted `UnitPrice` value.

After solving this problem, try clicking the Delete button for any product again. This time you'll get an `InvalidOperationException` when the ObjectDataSource attempts to invoke the BLL's `UpdateProduct` method.

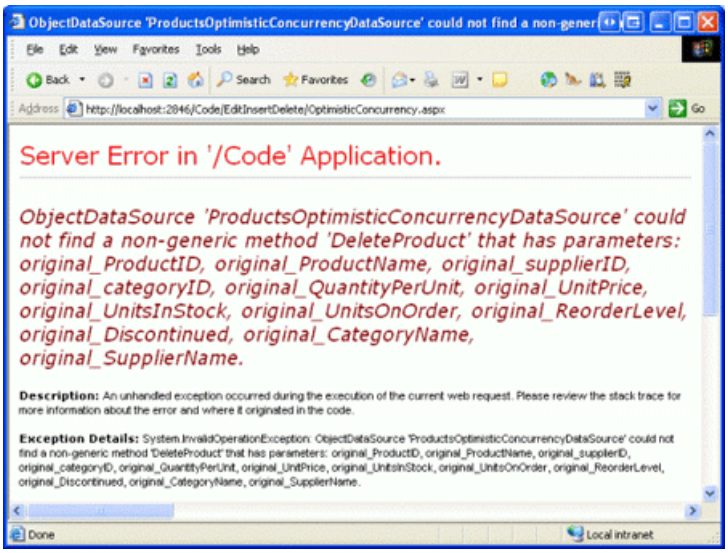


Figure 16: The ObjectDataSource Cannot Find a Method with the Input Parameters it Wants to Send (Click to view full-size image)

Looking at the exception's message, it's clear that the ObjectDataSource wants to invoke a BLL `DeleteProduct` method that includes `original_CategoryName` and `original_SupplierName` input parameters. This is because the `ItemTemplate` s for the `CategoryID` and `SupplierID` TemplateFields currently contain two-way Bind statements with the `CategoryName` and `SupplierName` data fields. Instead, we need to include `Bind` statements with the `CategoryID` and `SupplierID` data fields. To accomplish this, replace the existing Bind statements with `Eval` statements, and then add hidden Label controls whose `Text` properties are bound to the `CategoryID` and `SupplierID` data fields using two-way databinding, as shown below:

```
<asp:TemplateField HeaderText="Category" SortExpression="CategoryName">
    <EditItemTemplate>
        ...
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="DummyCategoryID" runat="server"
            Text='<%# Bind("CategoryID") %>' Visible="False"></asp:Label>
        <asp:Label ID="Label2" runat="server"
            Text='<%# Eval("CategoryName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="Supplier" SortExpression="SupplierName">
    <EditItemTemplate>
        ...
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="DummySupplierID" runat="server"
            Text='<%# Bind("SupplierID") %>' Visible="False"></asp:Label>
        <asp:Label ID="Label3" runat="server"
            Text='<%# Eval("SupplierName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

With these changes, we are now able to successfully delete and edit product information! In Step 5 we'll look at how to verify that concurrency violations are being detected. But for now, take a few minutes to try updating and deleting a few records to ensure that updating and deleting for a single user works as expected.

Step 5: Testing the Optimistic Concurrency Support

In order to verify that concurrency violations are being detected (rather than resulting in data being blindly overwritten), we need to open two browser windows to this page. In both browser instances, click on the Edit button for Chai. Then, in just one of the browsers, change the name to "Chai Tea" and click Update. The update should succeed and return the GridView to its pre-editing state, with "Chai Tea" as the new product name.

In the other browser window instance, however, the product name TextBox still shows "Chai". In this second browser window, update the `UnitPrice` to `25.00`. Without optimistic concurrency support, clicking update in the second browser instance would change the product name back to "Chai", thereby overwriting the changes made by the first browser instance. With optimistic concurrency employed, however, clicking the Update button in the second browser instance results in a DBConcurrencyException.

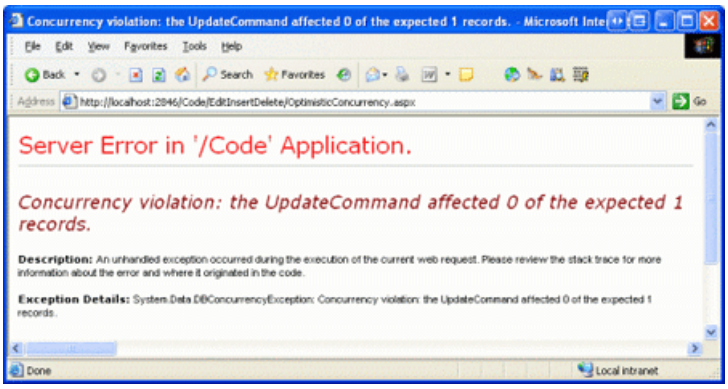


Figure 17: When a Concurrency Violation is Detected, a `DBConcurrencyException` is Thrown (Click to view full-size image)

The `DBConcurrencyException` is only thrown when the DAL's batch update pattern is utilized. The DB direct pattern does not raise an exception, it merely indicates that no rows were affected. To illustrate this, return both browser instances' GridView to their pre-editing state. Next, in the first browser instance, click the Edit button and change the product name from "Chai Tea" back to "Chai" and click Update. In the second browser window, click the Delete

button for Chai.

Upon clicking Delete, the page posts back, the GridView invokes the ObjectDataSource's `Delete()` method, and the ObjectDataSource calls down into the `ProductsOptimisticConcurrencyBLL` class's `DeleteProduct` method, passing along the original values. The original `ProductName` value for the second browser instance is "Chai Tea", which doesn't match up with the current `ProductName` value in the database. Therefore the `DELETE` statement issued to the database affects zero rows since there's no record in the database that the `WHERE` clause satisfies. The `DeleteProduct` method returns `false` and the ObjectDataSource's data is rebound to the GridView.

From the end user's perspective, clicking on the Delete button for Chai Tea in the second browser window caused the screen to flash and, upon coming back, the product is still there, although now it's listed as "Chai" (the product name change made by the first browser instance). If the user clicks the Delete button again, the Delete will succeed, as the GridView's original `ProductName` value ("Chai") now matches up with the value in the database.

In both of these cases, the user experience is far from ideal. We clearly don't want to show the user the nitty-gritty details of the `DBConcurrencyException` exception when using the batch update pattern. And the behavior when using the DB direct pattern is somewhat confusing as the users command failed, but there was no precise indication of why.

To remedy these two issues, we can create Label Web controls on the page that provide an explanation to why an update or delete failed. For the batch update pattern, we can determine whether or not a `DBConcurrencyException` exception occurred in the GridView's post-level event handler, displaying the warning label as needed. For the DB direct method, we can examine the return value of the BLL method (which is `true` if one row was affected, `false` otherwise) and display an informational message as needed.

Step 6: Adding Informational Messages and Displaying Them in the Face of a Concurrency Violation

When a concurrency violation occurs, the behavior exhibited depends on whether the DAL's batch update or DB direct pattern was used. Our tutorial uses both patterns, with the batch update pattern being used for updating and the DB direct pattern used for deleting. To get started, let's add two Label Web controls to our page that explain that a concurrency violation occurred when attempting to delete or update data. Set the Label control's `Visible` and `EnableViewState` properties to `false`; this will cause them to be hidden on each page visit except for those particular page visits where their `Visible` property is programmatically set to `true`.

```
<asp:Label ID="DeleteConflictMessage" runat="server" Visible="False"
    EnableViewState="False" CssClass="Warning"
    Text="The record you attempted to delete has been modified by another user
           since you last visited this page. Your delete was cancelled to allow
           you to review the other user's changes and determine if you want to
           continue deleting this record." />
<asp:Label ID="UpdateConflictMessage" runat="server" Visible="False"
    EnableViewState="False" CssClass="Warning"
    Text="The record you attempted to update has been modified by another user
           since you started the update process. Your changes have been replaced
           with the current values. Please review the existing values and make
           any needed changes." />
```

In addition to setting their `Visible`, `EnabledViewState`, and `Text` properties, I've also set the `CssClass` property to `Warning`, which causes the Label's to be displayed in a large, red, italic, bold font. This CSS `Warning` class was defined and added to Styles.css back in the *Examining the Events Associated with Inserting, Updating, and Deleting* tutorial.

After adding these Labels, the Designer in Visual Studio should look similar to Figure 18.

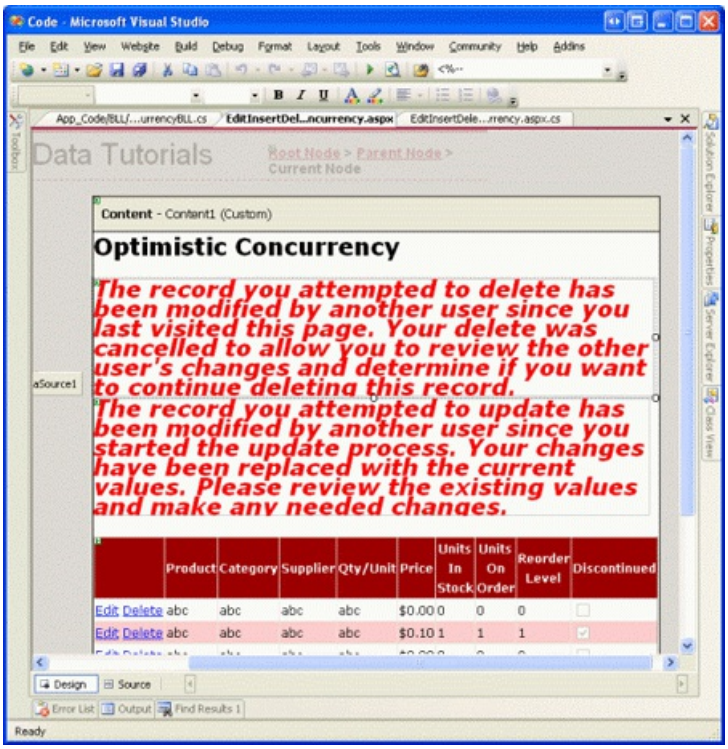


Figure 18: Two Label Controls Have Been Added to the Page (Click to view full-size image)

With these Label Web controls in place, we're ready to examine how to determine when a concurrency violation has occurred, at which point the appropriate Label's `Visible` property can be set to `true`, displaying the informational message.

Handling Concurrency Violations When Updating

Let's first look at how to handle concurrency violations when using the batch update pattern. Since such violations with the batch update pattern cause a `DBConcurrencyException` exception to be thrown, we need to add code to our ASP.NET page to determine whether a `DBConcurrencyException` exception occurred during the update process. If so, we should display a message to the user explaining that their changes were not saved because another user had modified the same data between when they started editing the record and when they clicked the Update button.

As we saw in the *Handling BLL- and DAL-Level Exceptions in an ASP.NET Page* tutorial, such exceptions can be detected and suppressed in the data Web control's post-level event handlers. Therefore, we need to create an event handler for the GridView's `RowUpdated` event that checks if a `DBConcurrencyException` exception has been thrown. This event handler is passed a reference to any exception that was raised during the updating process, as shown in the event handler code below:

```
protected void ProductsGrid_RowUpdated(object sender, GridViewUpdatedEventArgs e)
{
    if (e.Exception != null && e.Exception.InnerException != null)
    {
        if (e.Exception.InnerException is System.Data.DBConcurrencyException)
        {
            // Display the warning message and note that the
            // exception has been handled...
            UpdateConflictMessage.Visible = true;
            e.ExceptionHandled = true;
        }
    }
}
```

In the face of a `DBConcurrencyException` exception, this event handler displays the `UpdateConflictMessage` Label

control and indicates that the exception has been handled. With this code in place, when a concurrency violation occurs when updating a record, the user's changes are lost, since they would have overwritten another user's modifications at the same time. In particular, the GridView is returned to its pre-editing state and bound to the current database data. This will update the GridView row with the other user's changes, which were previously not visible. Additionally, the `UpdateConflictMessage` Label control will explain to the user what just happened. This sequence of events is detailed in Figure 19.

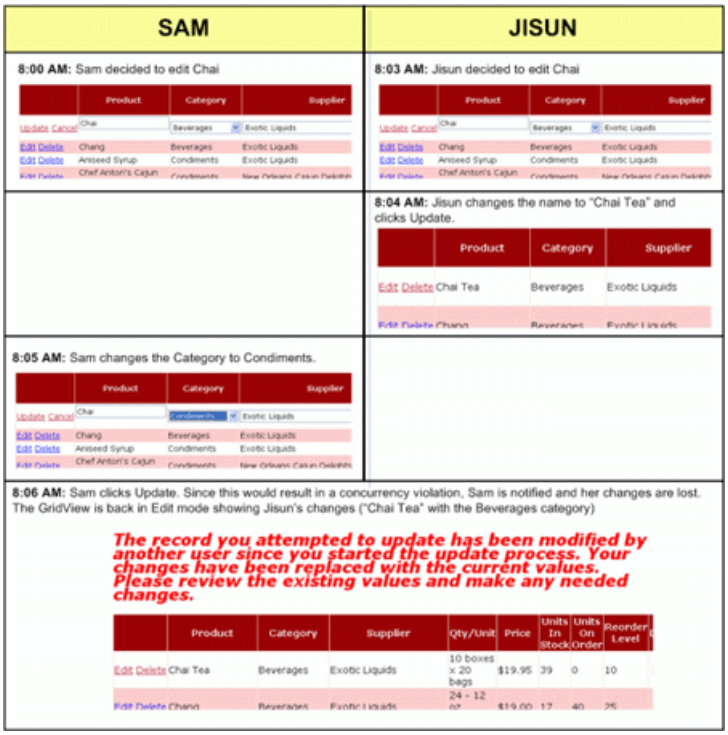


Figure 19: A User s Updates are Lost in the Face of a Concurrency Violation (Click to view full-size image)

> **NOTE**
>
> Alternatively, rather than returning the GridView to the pre-editing state, we could leave the GridView in its editing state by setting the `KeepInEditMode` property of the passed-in `GridViewUpdatedEventArgs` object to true. If you take this approach, however, be certain to rebind the data to the GridView (by invoking its `DataBind()` method) so that the other user's values are loaded into the editing interface. The code available for download with this tutorial has these two lines of code in the `RowUpdated` event handler commented out; simply uncomment these lines of code to have the GridView remain in edit mode after a concurrency violation.

Responding to Concurrency Violations When Deleting

With the DB direct pattern, there is no exception raised in the face of a concurrency violation. Instead, the database statement simply affects no records, as the WHERE clause does not match with any record. All of the data modification methods created in the BLL have been designed such that they return a Boolean value indicating whether or not they affected precisely one record. Therefore, to determine if a concurrency violation occurred when deleting a record, we can examine the return value of the BLL's `DeleteProduct` method.

The return value for a BLL method can be examined in the ObjectDataSource's post-level event handlers through the `ReturnValue` property of the `ObjectDataSourceStatusEventArgs` object passed into the event handler. Since we are interested in determining the return value from the `DeleteProduct` method, we need to create an event handler for the ObjectDataSource's `Deleted` event. The `ReturnValue` property is of type `object` and can be `null` if an exception was raised and the method was interrupted before it could return a value. Therefore, we should first ensure that the `ReturnValue` property is not `null` and is a Boolean value. Assuming this check passes, we show the `DeleteConflictMessage` Label control if the `ReturnValue` is `false`. This can be accomplished by using the following code:

```
protected void ProductsOptimisticConcurrencyDataSource_Deleted(
    object sender, ObjectDataSourceStatusEventArgs e)
{
    if (e.ReturnValue != null && e.ReturnValue is bool)
    {
        bool deleteReturnValue = (bool)e.ReturnValue;
        if (deleteReturnValue == false)
        {
            // No row was deleted, display the warning message
            DeleteConflictMessage.Visible = true;
        }
    }
}
```

In the face of a concurrency violation, the user's delete request is canceled. The GridView is refreshed, showing the changes that occurred for that record between the time the user loaded the page and when he clicked the Delete button. When such a violation transpires, the `DeleteConflictMessage` Label is shown, explaining what just happened (see Figure 20).

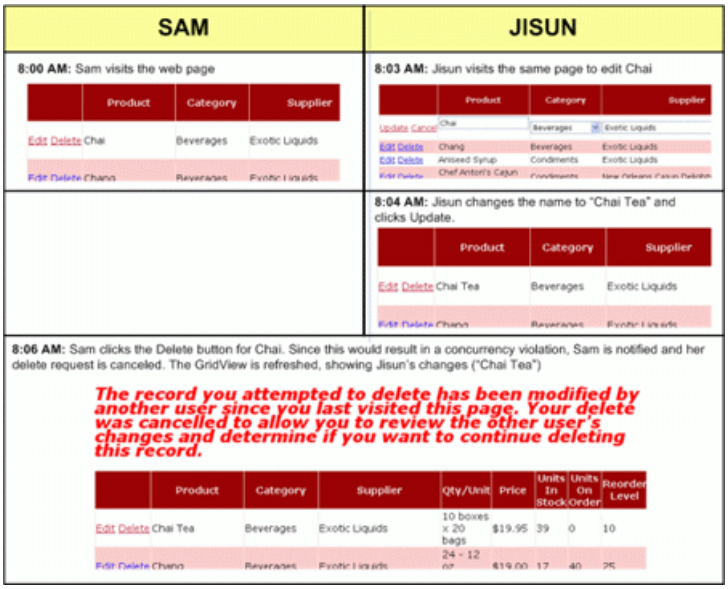


Figure 20: A User s Delete is Canceled in the Face of a Concurrency Violation (Click to view full-size image)

Summary

Opportunities for concurrency violations exist in every application that allows multiple, concurrent users to update or delete data. If such violations are not accounted for, when two users simultaneously update the same data whoever gets in the last write "wins," overwriting the other user's changes changes. Alternatively, developers can implement either optimistic or pessimistic concurrency control. Optimistic concurrency control assumes that concurrency violations are infrequent and simply disallows an update or delete command that would constitute a concurrency violation. Pessimistic concurrency control assumes that concurrency violations are frequent and simply rejecting one user's update or delete command is not acceptable. With pessimistic concurrency control, updating a record involves locking it, thereby preventing any other users from modifying or deleting the record while it is locked.

The Typed DataSet in .NET provides functionality for supporting optimistic concurrency control. In particular, the `UPDATE` and `DELETE` statements issued to the database include all of the table's columns, thereby ensuring that the update or delete will only occur if the record's current data matches with the original data the user had when performing their update or delete. Once the DAL has been configured to support optimistic concurrency, the BLL methods need to be updated. Additionally, the ASP.NET page that calls down into the BLL must be configured such that the ObjectDataSource retrieves the original values from its data Web control and passes them down into the BLL.

As we saw in this tutorial, implementing optimistic concurrency control in an ASP.NET web application involves updating the DAL and BLL and adding support in the ASP.NET page. Whether or not this added work is a wise investment of your time and effort depends on your application. If you infrequently have concurrent users updating data, or the data they are updating is different from one another, then concurrency control is not a key issue. If, however, you routinely have multiple users on your site working with the same data, concurrency control can help prevent one user's updates or deletes from unwittingly overwriting another's.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Adding Client-Side Confirmation When Deleting (C#)

2/20/2020 • 11 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In the interfaces we've created so far, a user can accidentally delete data by clicking the Delete button when they meant to click the Edit button. In this tutorial we'll add a client-side confirmation dialog box that appears when the Delete button is clicked.

Introduction

Over the past several tutorials we ve seen how to use our application architecture, ObjectDataSource, and the data Web controls in concert to provide inserting, editing, and deleting capabilities. The deleting interfaces we ve examined thus far have been composed of a Delete button that, when clicked, causes a postback and invokes the ObjectDataSource s `Delete()` method. The `Delete()` method then invokes the configured method from the Business Logic Layer, which propagates the call down to the Data Access Layer, issuing the actual `DELETE` statement to the database.

While this user interface enables visitors to delete records through the GridView, DetailsView, or FormView controls, it lacks any sort of confirmation when the user clicks the Delete button. If a user accidentally clicks the Delete button when they meant to click Edit, the record they meant to update will instead be deleted. To help prevent this, in this tutorial we'll add a client-side confirmation dialog box that appears when the Delete button is clicked.

The JavaScript `confirm(string)` function displays its string input parameter as the text inside a modal dialog box that comes equipped with two buttons - OK and Cancel (see Figure 1). The `confirm(string)` function returns a Boolean value depending on what button is clicked (`true` , if the user clicks OK, and `false` if they click Cancel).

Figure 1: The JavaScript `confirm(string)` Method Displays a Modal, Client-Side Messagebox

During a form submission, if a value of `false` is returned from a client-side event handler then the form submission is cancelled. Using this feature, we can have the Delete button s client-side `onclick` event handler return the value of a call to `confirm("Are you sure you want to delete this product?")`. If the user clicks Cancel, `confirm(string)` will return false, thereby causing the form submission to cancel. With no postback, the product whose Delete button was clicked won't be deleted. If, however, the user clicks OK in the confirmation dialog box, the postback will continue unabated and the product will be deleted. Consult Using JavaScript s `confirm()` Method to Control Form Submission for more information on this technique.

Adding the necessary client-side script differs slightly if using templates than when using a CommandField. Therefore, in this tutorial we will look at both a FormView and GridView example.

NOTE

Using client-side confirmation techniques, like the ones discussed in this tutorial, assumes that your users are visiting with browsers that support JavaScript and that they have JavaScript enabled. If either of these assumptions are not true for a particular user, clicking the Delete button will immediately cause a postback (not displaying a confirm messagebox).

Step 1: Creating a FormView That Supports Deletion

Start by adding a FormView to the `ConfirmationOnDelete.aspx` page in the `EditInsertDelete` folder, binding it to a new ObjectDataSource that pulls back the product information via the `ProductsBLL` class s `GetProducts()` method. Also configure the ObjectDataSource so that the `ProductsBLL` class s `DeleteProduct(productID)` method is mapped to the ObjectDataSource s `Delete()` method; ensure that the INSERT and UPDATE tabs drop-down lists are set to (None). Finally, check the Enable Paging checkbox in the FormView s smart tag.

After these steps, the new ObjectDataSource s declarative markup will look like the following:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    DeleteMethod="DeleteProduct" OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
    <DeleteParameters>
        <asp:Parameter Name="productID" Type="Int32" />
    </DeleteParameters>
</asp:ObjectDataSource>
```

As in our past examples that did not use optimistic concurrency, take a moment to clear out the ObjectDataSource s `OldValuesParameterFormatString` property.

Since it has been bound to an ObjectDataSource control that only supports deleting, the FormView s `ItemTemplate` offers only the Delete button, lacking the New and Update buttons. The FormView s declarative markup, however, includes a superfluous `EditItemTemplate` and `InsertItemTemplate`, which can be removed. Take a moment to customize the `ItemTemplate` so that is shows only a subset of the product data fields. I ve configured mine to show the product s name in an `<h3>` heading above its supplier and category names (along with the Delete button).

```
<asp:FormView ID="FormView1" AllowPaging="True" DataKeyNames="ProductID"
    DataSourceID="ObjectDataSource1" runat="server">
    <ItemTemplate>
        <h3><i><%# Eval("ProductName") %></i></h3>
        <b>Category:</b>
        <asp:Label ID="CategoryNameLabel" runat="server"
            Text='<%# Eval("CategoryName") %>'>
        </asp:Label><br />
        <b>Supplier:</b>
        <asp:Label ID="SupplierNameLabel" runat="server"
            Text='<%# Eval("SupplierName") %>'>
        </asp:Label><br />
        <asp:LinkButton ID="DeleteButton" runat="server" CausesValidation="False"
            CommandName="Delete" Text="Delete">
        </asp:LinkButton>
    </ItemTemplate>
</asp:FormView>
```

With these changes, we have a fully functional web page that allows a user to toggle through the products one at a time, with the ability to delete a product by simply clicking the Delete button. Figure 2 shows a screen shot of our progress thus far when viewed through a browser.

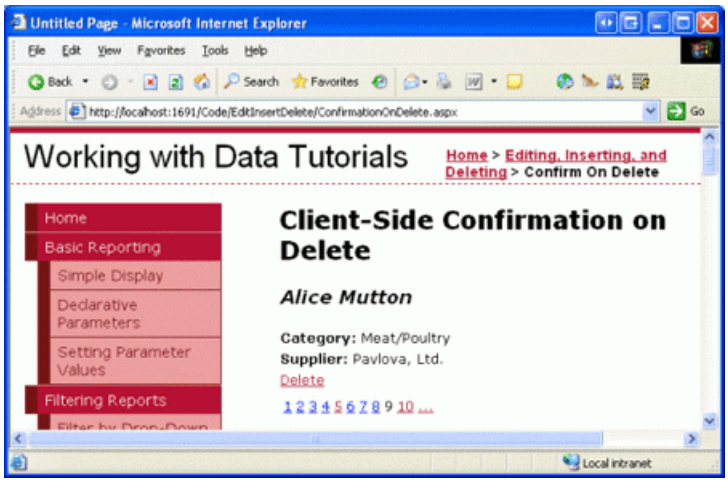


Figure 2: The FormView Shows Information About a Single Product (Click to view full-size image)

Step 2: Calling the confirm(string) Function from the Delete Buttons Client-Side onclick Event

With the FormView created, the final step is to configure the Delete button such that when it s clicked by the visitor, the JavaScript `confirm(string)` function is invoked. Adding client-side script to a Button, LinkButton, or ImageButton s client-side `onclick` event can be accomplished through the use of the `OnClientClick property`, which is new to ASP.NET 2.0. Since we want to have the value of the `confirm(string)` function returned, simply set this property to: `return confirm('Are you certain that you want to delete this product?');`

After this change the Delete LinkButton s declarative syntax should look something like:

```
<asp:LinkButton ID="DeleteButton" runat="server" CausesValidation="False"
    CommandName="Delete" Text="Delete"
    OnClientClick="return confirm('Are you certain you want to delete this product?');">
</asp:LinkButton>
```

That s all there is to it! Figure 3 shows a screen shot of this confirmation in action. Clicking the Delete button brings up the confirm dialog box. If the user clicks Cancel, the postback is cancelled and the product is not deleted. If, however, the user clicks OK, the postback continues and the ObjectDataSource s `Delete()` method is invoked, culminating in the database record being deleted.

> **NOTE**
>
> The string passed into the `confirm(string)` JavaScript function is delimited with apostrophes (rather than quotation marks). In JavaScript, strings can be delimited using either character. We use apostrophes here so that the delimiters for the string passed into `confirm(string)` do not introduce an ambiguity with the delimiters used for the `OnClientClick` property value.

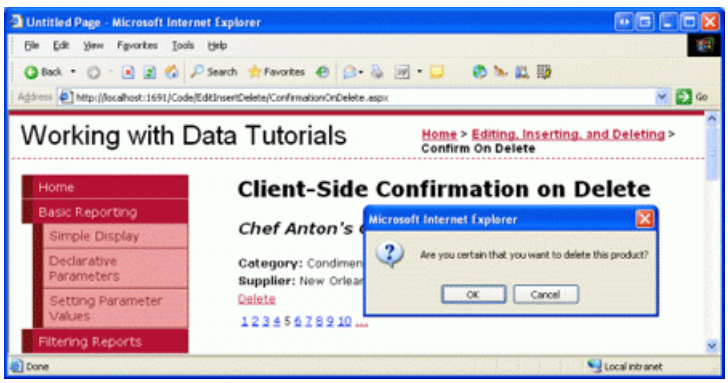


Figure 3: A Confirmation is Now Displayed When Clicking the Delete Button (Click to view full-size image)

Step 3: Configuring the OnClientClick Property for the Delete Button in a CommandField

When working with a Button, LinkButton, or ImageButton directly in a template, a confirmation dialog box can be associated with it by simply configuring its `OnClientClick` property to return the results of the JavaScript `confirm(string)` function. However, the CommandField - which adds a field of Delete buttons to a GridView or DetailsView - does not have an `OnClientClick` property that can be set declaratively. Instead, we must programmatically reference the Delete button in the GridView or DetailsView s appropriate `DataBound` event handler, and then set its `OnClientClick` property there.

> **NOTE**
>
> When setting the Delete button s `OnClientClick` property in the appropriate `DataBound` event handler, we have access to the data was bound to the current record. This means we can extend the confirmation message to include details about the particular record, such as, "Are you sure you want to delete the Chai product?" Such customization is also possible in templates using databinding syntax.

To practice setting the `OnClientClick` property for the Delete button(s) in a CommandField, let s add a GridView to the page. Configure this GridView to use the same ObjectDataSource control that the FormView uses. Also limit the GridView s BoundFields to only include the product s name, category, and supplier. Lastly, check the Enable Deleting checkbox from the GridView s smart tag. This will add a CommandField to the GridView s `Columns` collection with its `ShowDeleteButton` property set to `true`.

After making these changes, your GridView s declarative markup should look like the following:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:CommandField ShowDeleteButton="True" />
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category" ReadOnly="True"
            SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier" ReadOnly="True"
            SortExpression="SupplierName" />
    </Columns>
</asp:GridView>
```

The CommandField contains a single Delete LinkButton instance that can be accessed programmatically from the GridView s `RowDataBound` event handler. Once referenced, we can set its `OnClientClick` property accordingly. Create an event handler for the `RowDataBound` event using the following code:

```
protected void GridView1_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        // reference the Delete LinkButton
        LinkButton db = (LinkButton)e.Row.Cells[0].Controls[0];

        // Get information about the product bound to the row
        Northwind.ProductsRow product =
            (Northwind.ProductsRow) ((System.Data.DataRowView) e.Row.DataItem).Row;

        db.OnClientClick = string.Format(
            "return confirm('Are you certain you want to delete the {0} product?');",
            product.ProductName.Replace("'", @"\'"));
    }
}
```

This event handler works with data rows (those that will have the Delete button) and begins by programmatically referencing the Delete button. In general use the following pattern:

```
ButtonType obj = (ButtonType) e.Row.Cells[commandFieldIndex].Controls[controlIndex];
```

ButtonType is the type of button being used by the CommandField - Button, LinkButton, or ImageButton. By default, the CommandField uses LinkButtons, but this can be customized via the CommandField s `ButtonType property`. The *commandFieldIndex* is the ordinal index of the CommandField within the GridView s `Columns` collection, whereas the *controlIndex* is the index of the Delete button within the CommandField s `Controls` collection. The *controlIndex* value depends on the button s position relative to other buttons in the CommandField. For example, if the only button displayed in the CommandField is the Delete button, use an index of 0. If, however, there s an Edit button that precedes the Delete button, use an index of 2. The reason an index of 2 is used is because two controls are added by the CommandField before the Delete button: the Edit button and a LiteralControl that s used to add some space between the Edit and Delete buttons.

For our particular example, the CommandField uses LinkButtons and, being the left-most field, has a *commandFieldIndex* of 0. Since there are no other buttons but the Delete button in the CommandField, we use a *controlIndex* of 0.

After referencing the Delete button in the CommandField, we next grab information about the product bound to the current GridView row. Finally, we set the Delete button s `OnClientClick` property to the appropriate JavaScript, which includes the product s name. Since the JavaScript string passed into the `confirm(string)` function is delimited using apostrophes we must escape any apostrophes that appear within the product s name. In particular, any apostrophes in the product s name are escaped with "`\'`".

With these changes complete, clicking on a Delete button in the GridView displays a customized confirmation dialog box (see Figure 4). As with the confirmation messagebox from the FormView, if the user clicks Cancel the postback is cancelled, thereby preventing the deletion from occurring.

NOTE

This technique can also be used to programmatically access the Delete button in the CommandField in a DetailsView. For the DetailsView, however, you d create an event handler for the `DataBound` event, since the DetailsView does not have a `RowDataBound` event.

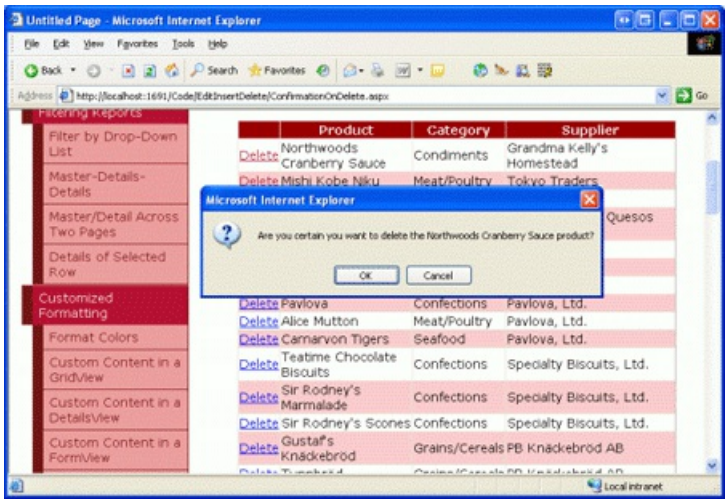


Figure 4: Clicking the GridView s Delete Button Displays a Customized Confirmation Dialog Box (Click to view full-size image)

Using TemplateFields

One of the disadvantages of the CommandField is that its buttons must be accessed through indexing and that the resulting object must be cast to the appropriate button type (Button, LinkButton, or ImageButton). Using "magic numbers" and hard-coded types invites problems that cannot be discovered until runtime. For example, if you, or another developer, adds new buttons to the CommandField at some point in the future (such as an Edit button) or changes the `ButtonType` property, the existing code will still compile without error, but visiting the page may cause an exception or unexpected behavior, depending on how your code was written and what changes were made.

An alternative approach is to convert the GridView and DetailsView s CommandFields into TemplateFields. This will generate a TemplateField with an `ItemTemplate` that has a LinkButton (or Button or ImageButton) for each button in the CommandField. These buttons `OnClientClick` properties can be assigned declaratively, as we saw with the FormView, or can be programmatically accessed in the appropriate `DataBound` event handler using the following pattern:

```
ButtonType obj = (ButtonType) e.Row.FindControl("controlID");
```

Where *controlID* is the value of the button s `ID` property. While this pattern still requires a hard-coded type for the cast, it removes the need for indexing, allowing for the layout to change without resulting in a runtime error.

Summary

The JavaScript `confirm(string)` function is a commonly used technique for controlling form submission workflow. When executed, the function displays a modal, client-side dialog box that includes two buttons, OK and Cancel. If the user clicks OK, the `confirm(string)` function returns `true`; clicking Cancel returns `false`. This functionality, coupled with a browser s behavior to cancel a form submission if an event handler during the submission process returns `false`, can be used to display a confirmation messagebox when deleting a record.

The `confirm(string)` function can be associated with a Button Web control s client-side `onclick` event handler through the control s `OnClientClick` property. When working with a Delete button in a template - either in one of the FormView s templates or in a TemplateField in the DetailsView or GridView - this property can be set either declaratively or programmatically, as we saw in this tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Limiting Data Modification Functionality Based on the User (C#)

2/20/2020 • 18 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

In a web application that allows users to edit data, different user accounts may have different data-editing privileges. In this tutorial we'll examine how to dynamically adjust the data modification capabilities based on the visiting user.

Introduction

A number of web applications support user accounts and provide different options, reports, and functionality based on the logged in user. For example, with our tutorials we might want to allow users from the supplier companies to log on to the site and update general information about their products - their name and quantity per unit, perhaps - along with supplier information, such as their company name, address, the contact person s information, and so on. Furthermore, we might want to include some user accounts for people from our company so that they can log on and update product information like units on stock, reorder level, and so forth. Our web application might also allow anonymous users to visit (people who have not logged on), but would limit them to just viewing data. With such a user account system in place, we would want the data Web controls in our ASP.NET pages to offer the inserting, editing, and deleting capabilities appropriate for the currently logged on user.

In this tutorial we'll examine how to dynamically adjust the data modification capabilities based on the visiting user. In particular, we'll create a page that displays the suppliers information in an editable DetailsView along with a GridView that lists the products provided by the supplier. If the user visiting the page is from our company, they can: view any supplier s information; edit their address; and edit the information for any product provided by the supplier. If, however, the user is from a particular company, they can only view and edit their own address information and can only edit their products that have not been marked as discontinued.

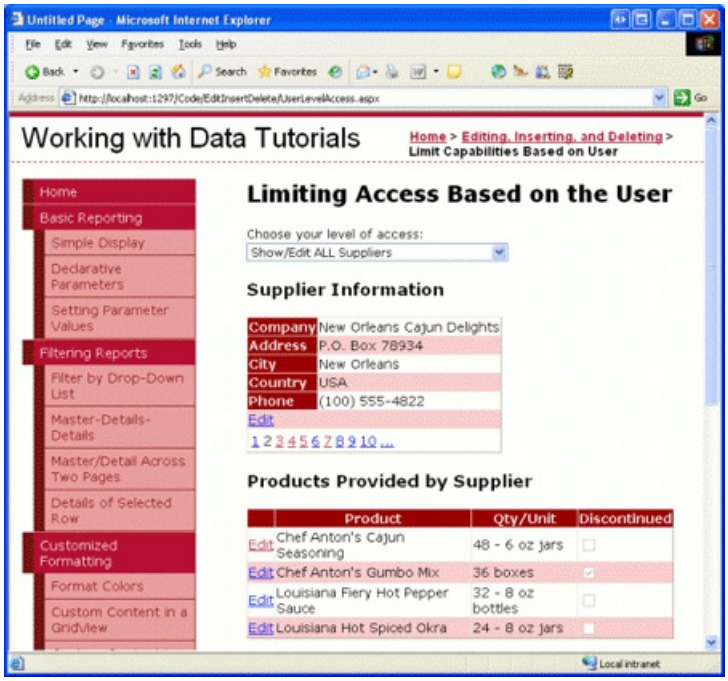


Figure 1: A User from Our Company Can Edit Any Supplier s Information (Click to view full-size image)

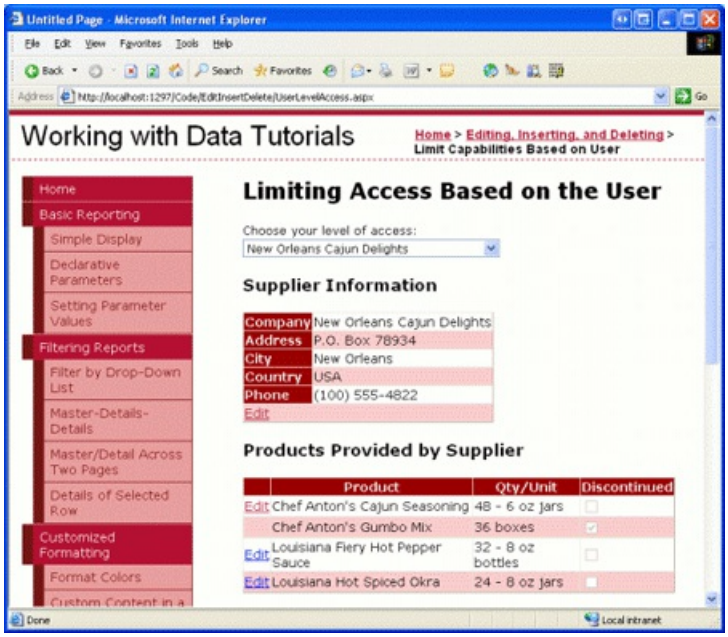


Figure 2: A User from a Particular Supplier Can Only View and Edit their Information (Click to view full-size image)

Let s get started!

> **NOTE**
>
> ASP.NET 2.0 s membership system provides a standardized, extensible platform for creating, managing, and validating user accounts. Since an examination of the membership system is beyond the scope of these tutorials, this tutorial instead "fakes" membership by allowing anonymous visitors to choose whether they are from a particular supplier or from our company. For more on membership, refer to my Examining ASP.NET 2.0 s Membership, Roles, and Profile article series.

Step 1: Allowing the User to Specify their Access Rights

In a real-world web application, a user s account information would include whether they worked for our company or for a particular supplier, and this information would be programmatically accessible from our ASP.NET pages once the user has logged on to the site. This information could be captured through ASP.NET 2.0 s roles system, as user-level account information through the profile system, or through some custom means.

Since the aim of this tutorial is to demonstrate adjusting the data modification capabilities based on the logged on user, and is not meant to showcase ASP.NET 2.0 s membership, roles, and profile systems, we'll use a very simply mechanism to determine the capabilities for the user visiting the page - a DropDownList from which the user can indicate if they should be able to view and edit any of the suppliers information or, alternatively, what particular supplier s information they can view and edit. If the user indicates that she can view and edit all supplier information (the default), she can page through all suppliers, edit any supplier s address information, and edit the name and quantity per unit for any product provided by the selected supplier. If the user indicates that she can only view and edit a particular supplier, however, then she can only view the details and products for that one supplier and can only update the name and quantity per unit information for those products that are *not* discontinued.

Our first step in this tutorial, then, is to create this DropDownList and populate it with the suppliers in the system. Open the `UserLevelAccess.aspx` page in the `EditInsertDelete` folder, add a DropDownList whose `ID` property is set to `Suppliers`, and bind this DropDownList to a new ObjectDataSource named `AllSuppliersDataSource`.

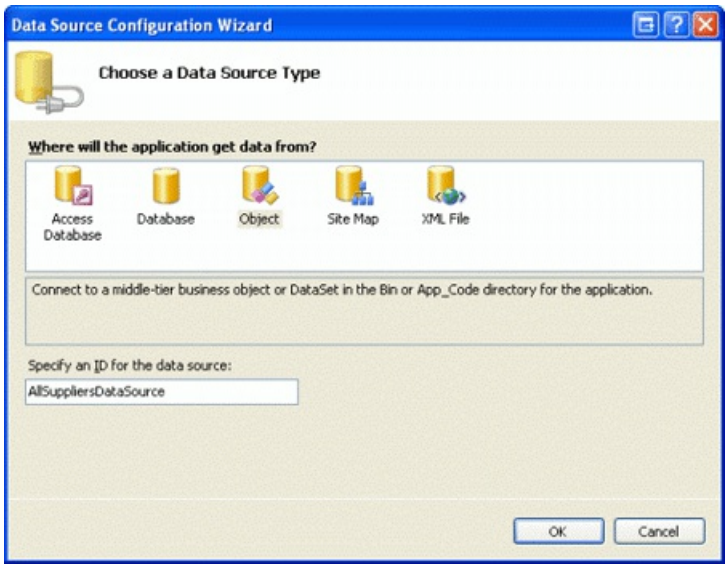


Figure 3: Create a New ObjectDataSource Named `AllSuppliersDataSource` (Click to view full-size image)

Since we want this DropDownList to include all suppliers, configure the ObjectDataSource to invoke the `SuppliersBLL` class s `GetSuppliers()` method. Also ensure that the ObjectDataSource s `Update()` method is mapped to the `SuppliersBLL` class s `UpdateSupplierAddress` method, as this ObjectDataSource will also be used by the DetailsView we'll be adding in Step 2.

After completing the ObjectDataSource wizard, complete the steps by configuring the `Suppliers` DropDownList such that it shows the `CompanyName` data field and uses the `SupplierID` data field as the value for each `ListItem`.

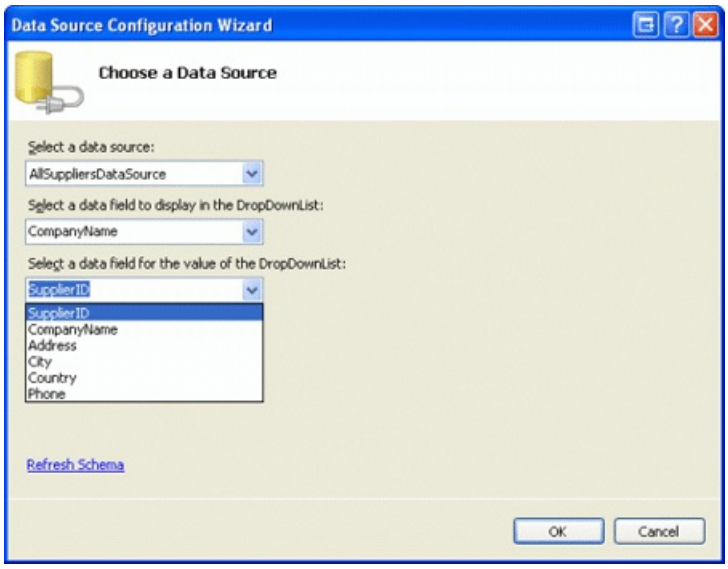


Figure 4: Configure the `Suppliers` DropDownList to Use the `CompanyName` and `SupplierID` Data Fields (Click to view full-size image)

At this point, the DropDownList lists the company names of the suppliers in the database. However, we also need to include a "Show/Edit ALL Suppliers" option to the DropDownList. To accomplish this, set the `Suppliers` DropDownList s `AppendDataBoundItems` property to `true` and then add a `ListItem` whose `Text` property is "Show/Edit ALL Suppliers" and whose value is `-1`. This can be added directly through the declarative markup or through the Designer by going to the Properties window and clicking on the ellipses in the DropDownList s `Items` property.

> **NOTE**
> Refer back to the *Master/Detail Filtering With a DropDownList* tutorial for a more detailed discussion on adding a Select All item to a databound DropDownList.

After the `AppendDataBoundItems` property has been set and the `ListItem` added, the DropDownList s declarative markup should look like:

```
<asp:DropDownList ID="Suppliers" runat="server" AppendDataBoundItems="True"
    DataSourceID="AllSuppliersDataSource" DataTextField="CompanyName"
    DataValueField="SupplierID">
    <asp:ListItem Value="-1">Show/Edit ALL Suppliers</asp:ListItem>
</asp:DropDownList>
```

Figure 5 shows a screen shot of our current progress, when viewed through a browser.

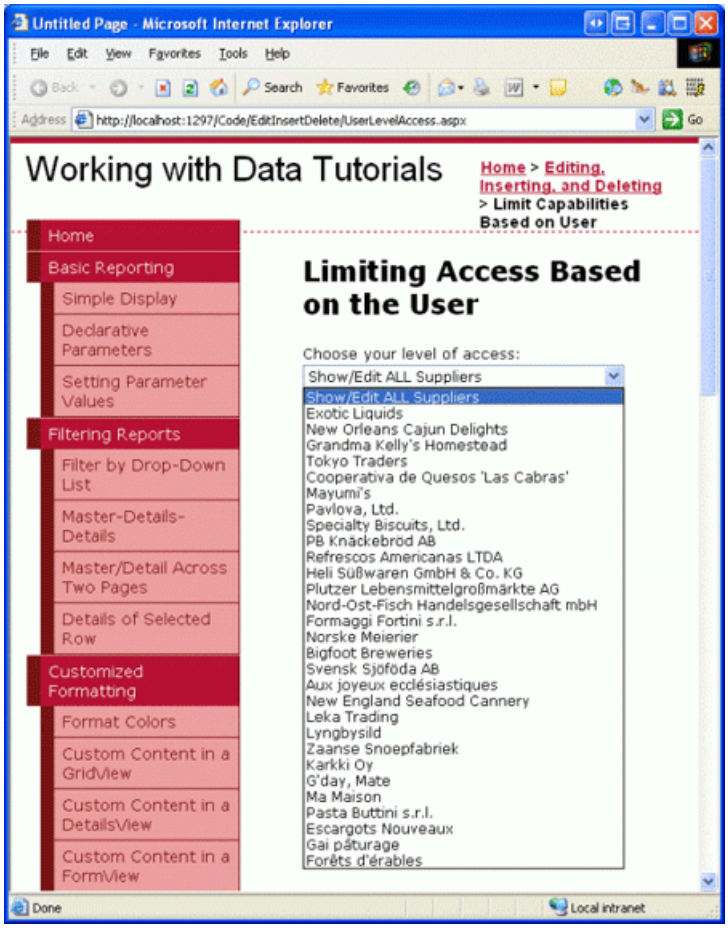


Figure 5: The `Suppliers` DropDownList Contains a Show ALL `ListItem`, Plus One for Each Supplier (Click to view full-size image)

Since we want to update the user interface immediately after the user has changed their selection, set the `Suppliers` DropDownList s `AutoPostBack` property to `true`. In Step 2 we'll create a DetailsView control that will show the information for the supplier(s) based on the DropDownList selection. Then, in Step 3, we'll create an event handler for this DropDownList s `SelectedIndexChanged` event, in which we'll add code that binds the appropriate supplier information to the DetailsView based upon the selected supplier.

Step 2: Adding a DetailsView Control

Let s use a DetailsView to show supplier information. For the user who can view and edit all suppliers, the DetailsView will support paging, allowing the user to step through the supplier information one record at a time.

If the user works for a particular supplier, however, the DetailsView will show only that particular supplier s information and will not include a paging interface. In either case, the DetailsView needs to allow the user to edit the supplier s address, city, and country fields.

Add a DetailsView to the page beneath the `Suppliers` DropDownList, set its `ID` property to `SupplierDetails`, and bind it to the `AllSuppliersDataSource` ObjectDataSource created in the previous step. Next, check the Enable Paging and Enable Editing checkboxes from the DetailsView s smart tag.

> **NOTE**
> If you don t see an Enable Editing option in the DetailsView s smart tag it s because you did not map the ObjectDataSource s `Update()` method to the `SuppliersBLL` class s `UpdateSupplierAddress` method. Take a moment to go back and make this configuration change, after which the Enable Editing option should appear in the DetailsView s smart tag.

Since the `SuppliersBLL` class s `UpdateSupplierAddress` method only accepts four parameters - `supplierID`, `address`, `city`, and `country` - modify the DetailsView s BoundFields so that the `CompanyName` and `Phone` BoundFields are read-only. Additionally, remove the `SupplierID` BoundField altogether. Finally, the `AllSuppliersDataSource` ObjectDataSource currently has its `OldValuesParameterFormatString` property set to `original_{0}`. Take a moment to remove this property setting from the declarative syntax altogether, or set it to the default value, `{0}`.

After configuring the `SupplierDetails` DetailsView and `AllSuppliersDataSource` ObjectDataSource, we will have the following declarative markup:

```
<asp:ObjectDataSource ID="AllSuppliersDataSource" runat="server"
    SelectMethod="GetSuppliers" TypeName="SuppliersBLL"
    UpdateMethod="UpdateSupplierAddress">
    <UpdateParameters>
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="address" Type="String" />
        <asp:Parameter Name="city" Type="String" />
        <asp:Parameter Name="country" Type="String" />
    </UpdateParameters>
</asp:ObjectDataSource>
<asp:DetailsView ID="SupplierDetails" runat="server" AllowPaging="True"
    AutoGenerateRows="False" DataKeyNames="SupplierID"
    DataSourceID="AllSuppliersDataSource">
    <Fields>
        <asp:BoundField DataField="CompanyName" HeaderText="Company"
            ReadOnly="True" SortExpression="CompanyName" />
        <asp:BoundField DataField="Address" HeaderText="Address"
            SortExpression="Address" />
        <asp:BoundField DataField="City" HeaderText="City"
            SortExpression="City" />
        <asp:BoundField DataField="Country" HeaderText="Country"
            SortExpression="Country" />
        <asp:BoundField DataField="Phone" HeaderText="Phone" ReadOnly="True"
            SortExpression="Phone" />
        <asp:CommandField ShowEditButton="True" />
    </Fields>
</asp:DetailsView>
```

At this point the DetailsView can be paged through and the selected supplier s address information can be updated, regardless of the selection made in the `Suppliers` DropDownList (see Figure 6).

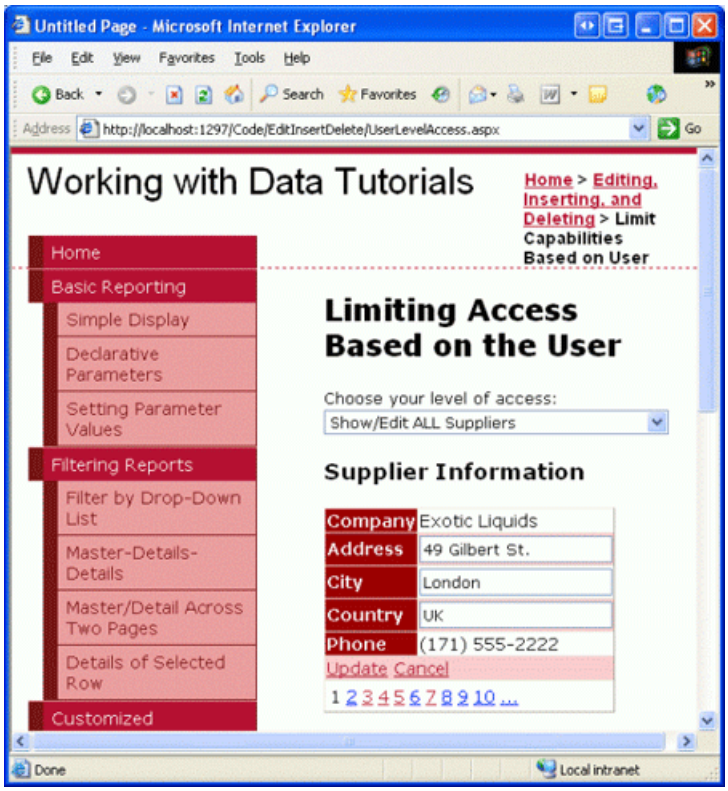


Figure 6: Any Suppliers Information Can Be Viewed, and Its Address Updated (Click to view full-size image)

Step 3: Displaying Only the Selected Supplier s Information

Our page currently displays the information for all suppliers regardless of whether a particular supplier has been selected from the `Suppliers` DropDownList. In order to display just the supplier information for the selected supplier we need to add another ObjectDataSource to our page, one that retrieves information about a particular supplier.

Add a new ObjectDataSource to the page, naming it `SingleSupplierDataSource`. From its smart tag, click the Configure Data Source link and have it use the `SuppliersBLL` class s `GetSupplierBySupplierID(supplierID)` method. As with the `AllSuppliersDataSource` ObjectDataSource, have the `SingleSupplierDataSource` ObjectDataSource s `Update()` method mapped to the `SuppliersBLL` class s `UpdateSupplierAddress` method.

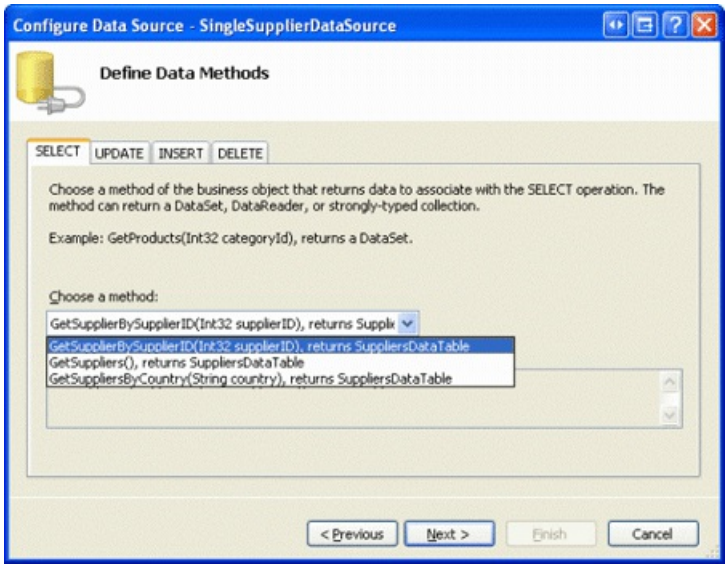


Figure 7: Configure the `SingleSupplierDataSource` ObjectDataSource to Use the `GetSupplierBySupplierID(supplierID)` Method (Click to view full-size image)

Next, we re prompted to specify the parameter source for the `GetSupplierBySupplierID(supplierID)` method s

`supplierID` input parameter. Since we want to show the information for the supplier selected from the DropDownList, use the `Suppliers` DropDownList s `SelectedValue` property as the parameter source.

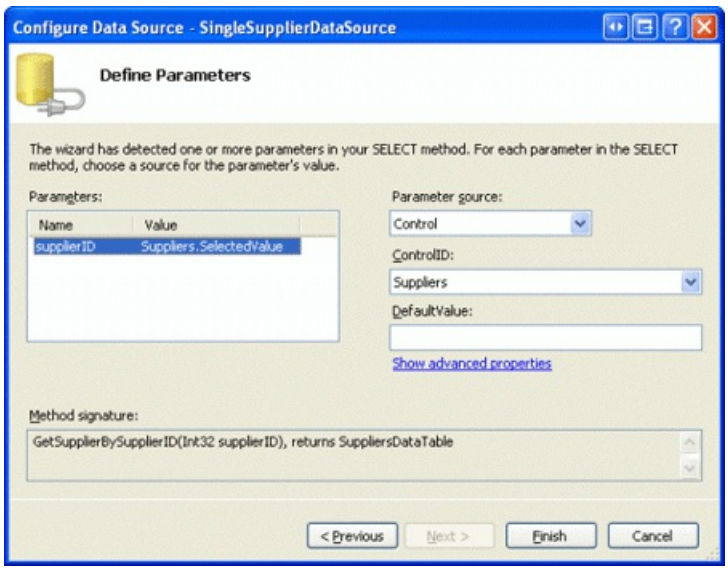


Figure 8: Use the `Suppliers` DropDownList as the `supplierID` Parameter Source (Click to view full-size image)

Even with this second ObjectDataSource added, the DetailsView control is currently configured to always use the `AllSuppliersDataSource` ObjectDataSource. We need to add logic to adjust the data source used by the DetailsView depending on the `Suppliers` DropDownList item selected. To accomplish this, create a `SelectedIndexChanged` event handler for the Suppliers DropDownList. This can most easily be created by double-clicking the DropDownList in the Designer. This event handler needs to determine what data source to use and must rebind the data to the DetailsView. This is accomplished with the following code:

```
protected void Suppliers_SelectedIndexChanged(object sender, EventArgs e)
{
    if (Suppliers.SelectedValue == "-1")
    {
        // The "Show/Edit ALL" option has been selected
        SupplierDetails.DataSourceID = "AllSuppliersDataSource";
        // Reset the page index to show the first record
        SupplierDetails.PageIndex = 0;
    }
    else
        // The user picked a particular supplier
        SupplierDetails.DataSourceID = "SingleSupplierDataSource";
    // Ensure that the DetailsView is in read-only mode
    SupplierDetails.ChangeMode(DetailsViewMode.ReadOnly);
    // Need to "refresh" the DetailsView
    SupplierDetails.DataBind();
}
```

The event handler begins by determining whether the "Show/Edit ALL Suppliers" option was selected. If it was, it sets the `SupplierDetails` DetailsView s `DataSourceID` to `AllSuppliersDataSource` and returns the user to the first record in the set of suppliers by setting the `PageIndex` property to 0. If, however, the user has selected a particular supplier from the DropDownList, the DetailsView s `DataSourceID` is assigned to `SingleSuppliersDataSource`. Regardless of what data source is used, the `SuppliersDetails` mode is reverted back to the read-only mode and the data is rebound to the DetailsView by a call to the `SuppliersDetails` control s `DataBind()` method.

With this event handler in place, the DetailsView control now shows the selected supplier, unless the "Show/Edit ALL Suppliers" option was selected, in which case all of the suppliers can be viewed through the paging interface. Figure 9 shows the page with the "Show/Edit ALL Suppliers" option selected; note that the paging interface is present, allowing the user to visit and update any supplier. Figure 10 shows the page with the Ma Maison supplier selected. Only Ma Maison s information is viewable and editable in this case.

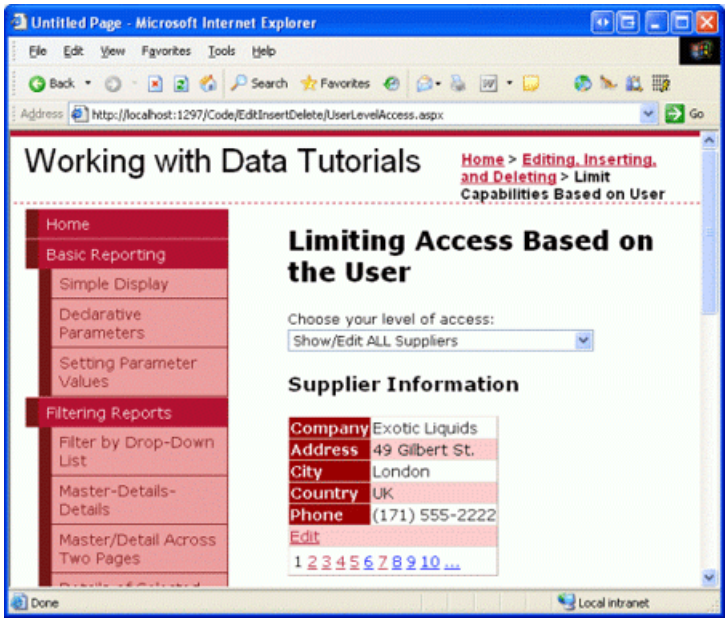


Figure 9: All of the Suppliers Information Can Be Viewed and Edited (Click to view full-size image)

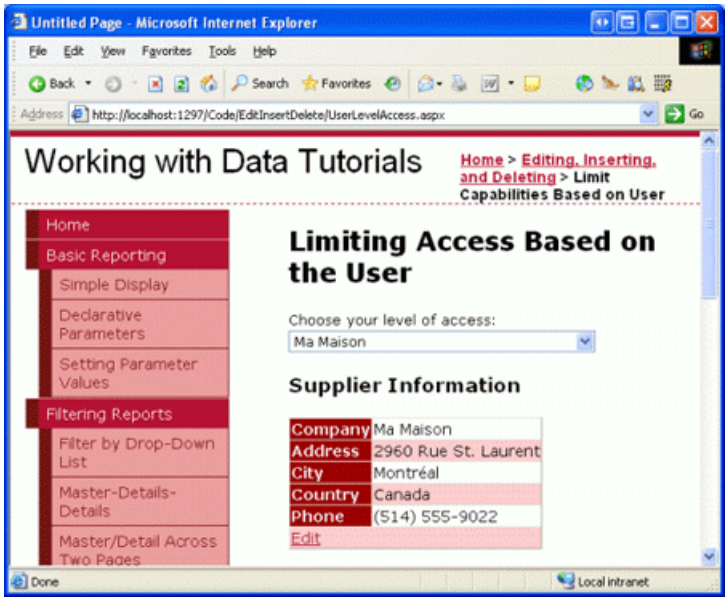


Figure 10: Only the Selected Supplier s Information Can Be Viewed and Edited (Click to view full-size image)

> **NOTE**
>
> For this tutorial, both the DropDownList and DetailsView control s `EnableViewState` must be set to `true` (the default) because the DropDownList s `SelectedIndex` and the DetailsView s `DataSourceID` property s changes must be remembered across postbacks.

Step 4: Listing the Suppliers Products in an Editable GridView

With the DetailsView complete, our next step is to include an editable GridView that lists those products provided by the selected supplier. This GridView should allow edits to only the `ProductName` and `QuantityPerUnit` fields. Moreover, if the user visiting the page is from a particular supplier, it should only allow updates to those products that are *not* discontinued. To accomplish this we'll need to first add an overload of the `ProductsBLL` class s `UpdateProducts` method that takes in just the `ProductID`, `ProductName`, and `QuantityPerUnit` fields as inputs. We ve stepped through this process beforehand in numerous tutorials, so let s just look at the code here, which should be added to `ProductsBLL`:

```
[System.ComponentModel.DataObjectMethodAttribute(
System.ComponentModel.DataObjectMethodType.Update, false)]
public bool UpdateProduct(string productName, string quantityPerUnit, int productID)
{
    Northwind.ProductsDataTable products = Adapter.GetProductByProductID(productID);
    if (products.Count == 0)
        // no matching record found, return false
        return false;
    Northwind.ProductsRow product = products[0];
    product.ProductName = productName;
    if (quantityPerUnit == null)
        product.SetQuantityPerUnitNull();
    else
        product.QuantityPerUnit = quantityPerUnit;
    // Update the product record
    int rowsAffected = Adapter.Update(product);
    // Return true if precisely one row was updated, otherwise false
    return rowsAffected == 1;
}
```

With this overload created, we're ready to add the GridView control and its associated ObjectDataSource. Add a new GridView to the page, set its `ID` property to `ProductsBySupplier`, and configure it to use a new ObjectDataSource named `ProductsBySupplierDataSource`. Since we want this GridView to list those products by the selected supplier, use the `ProductsBLL` class's `GetProductsBySupplierID(supplierID)` method. Also map the `Update()` method to the new `UpdateProduct` overload we just created.

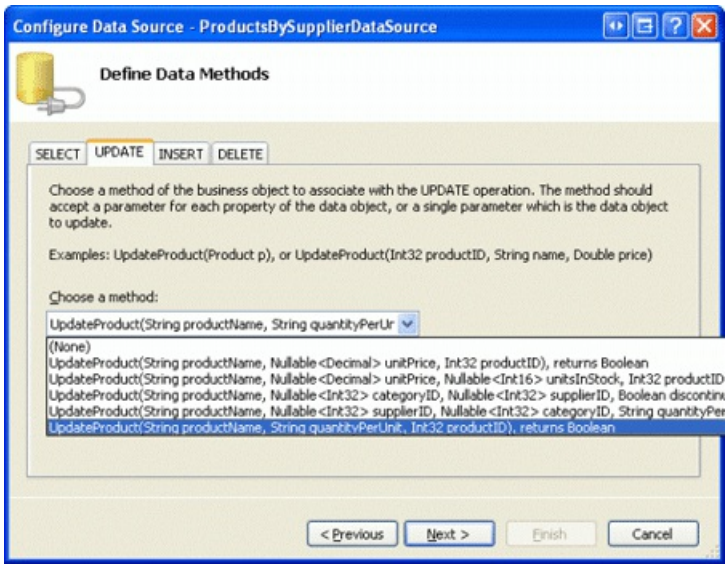


Figure 11: Configure the ObjectDataSource to Use the `UpdateProduct` Overload Just Created (Click to view full-size image)

We're prompted to select the parameter source for the `GetProductsBySupplierID(supplierID)` method's `supplierID` input parameter. Since we want to show the products for the supplier selected in the DetailsView, use the `SuppliersDetails` DetailsView control's `SelectedValue` property as the parameter source.

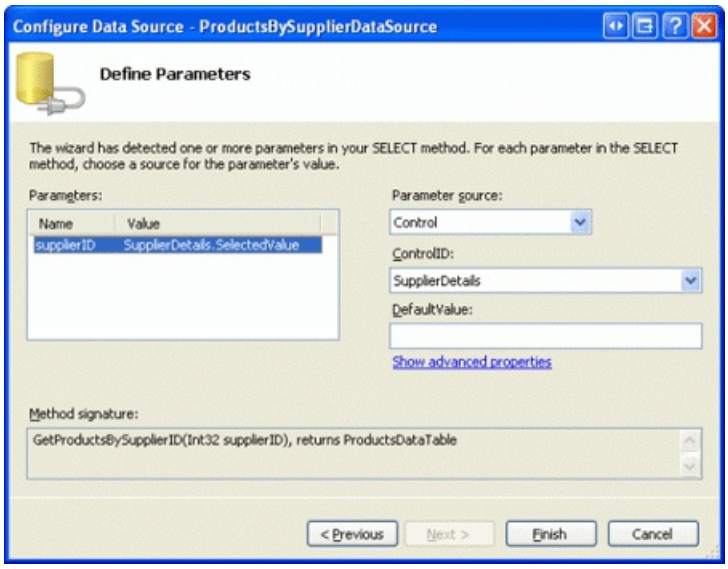


Figure 12: Use the `SuppliersDetails` DetailsView s `SelectedValue` Property as the Parameter Source (Click to view full-size image)

Returning to the GridView, remove all of the GridView fields except for `ProductName`, `QuantityPerUnit`, and `Discontinued`, marking the `Discontinued` CheckBoxField as read-only. Also, check the Enable Editing option from the GridView s smart tag. After these changes have been made, the declarative markup for the GridView and ObjectDataSource should look similar to the following:

```
<asp:GridView ID="ProductsBySupplier" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsBySupplierDataSource">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="QuantityPerUnit" HeaderText="Qty/Unit"
            SortExpression="QuantityPerUnit" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            ReadOnly="True" SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsBySupplierDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}" TypeName="ProductsBLL"
    SelectMethod="GetProductsBySupplierID" UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
    <SelectParameters>
        <asp:ControlParameter ControlID="SupplierDetails" Name="supplierID"
            PropertyName="SelectedValue" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

As with our previous ObjectDataSources, this one s `OldValuesParameterFormatString` property is set to `original_{0}`, which will cause problems when attempting to update a product s name or quantity per unit. Remove this property from the declarative syntax altogether or set it to its default, `{0}`.

With this configuration complete, our page now lists the products provided by the supplier selected in the GridView (see Figure 13). Currently *any* product s name or quantity per unit can be updated. However, we need to update our page logic so that such functionality is prohibited for discontinued products for users associated with a particular supplier. We'll tackle this final piece in Step 5.

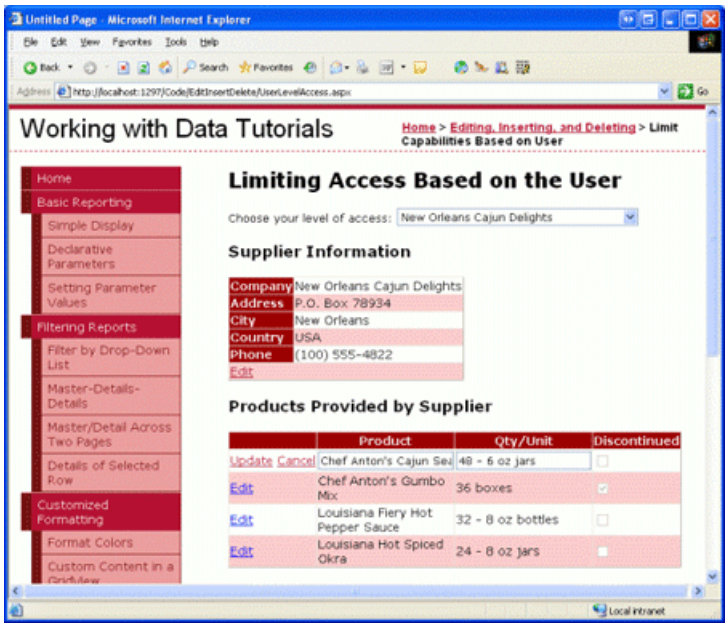


Figure 13: The Products Provided by the Selected Supplier are Displayed (Click to view full-size image)

> **NOTE**
>
> With the addition of this editable GridView the `Suppliers` DropDownList s `SelectedIndexChanged` event handler should be updated to return the GridView to a read-only state. Otherwise, if a different supplier is selected while in the middle of editing product information, the corresponding index in the GridView for the new supplier will also be editable. To prevent this, simply set the GridView s `EditIndex` property to `-1` in the `SelectedIndexChanged` event handler.

Also, recall that it is important that the GridView s view state be enabled (the default behavior). If you set the GridView s `EnableViewState` property to `false`, you run the risk of having concurrent users unintentionally deleting or editing records. See WARNING: Concurrency Issue with ASP.NET 2.0 GridViews/DetailsView/FormViews that Support Editing and/or Deleting and Whose View State is Disabled for more information.

Step 5: Disallow Editing for Discontinued Products When Show/Edit ALL Suppliers is Not Selected

While the `ProductsBySupplier` GridView is fully functional, it currently grants too much access to those users who are from a particular supplier. Per our business rules, such users should not be able to update discontinued products. To enforce this, we can hide (or disable) the Edit button in those GridView rows with discontinued products when the page is being visited by a user from a supplier.

Create an event handler for the GridView s `RowDataBound` event. In this event handler we need to determine whether or not the user is associated with a particular supplier, which, for this tutorial, can be determined by checking the Suppliers DropDownList s `SelectedValue` property - if it s something other than -1, then the user is associated with a particular supplier. For such users, we then need to determine whether or not the product is discontinued. We can grab a reference to the actual `ProductRow` instance bound to the GridView row via the `e.Row.DataItem` property, as discussed in the *Displaying Summary Information in the GridView s Footer* tutorial. If the product is discontinued, we can grab a programmatic reference to the Edit button in the GridView s CommandField using the techniques discussed in the previous tutorial, *Adding Client-Side Confirmation When Deleting*. Once we have a reference we can then hide or disable the button.

```
protected void ProductsBySupplier_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        // Is this a supplier-specific user?
        if (Suppliers.SelectedValue != "-1")
        {
            // Get a reference to the ProductRow
            Northwind.ProductsRow product =
                (Northwind.ProductsRow)((System.Data.DataRowView)e.Row.DataItem).Row;
            // Is this product discontinued?
            if (product.Discontinued)
            {
                // Get a reference to the Edit LinkButton
                LinkButton editButton = (LinkButton)e.Row.Cells[0].Controls[0];
                // Hide the Edit button
                editButton.Visible = false;
            }
        }
    }
}
```

With this event handler in place, when visiting this page as a user from a specific supplier those products that are discontinued are not editable, as the Edit button is hidden for these products. For example, Chef Anton s Gumbo Mix is a discontinued product for the New Orleans Cajun Delights supplier. When visiting the page for this particular supplier, the Edit button for this product is hidden from sight (see Figure 14). However, when visiting using the "Show/Edit ALL Suppliers," the Edit button is available (see Figure 15).

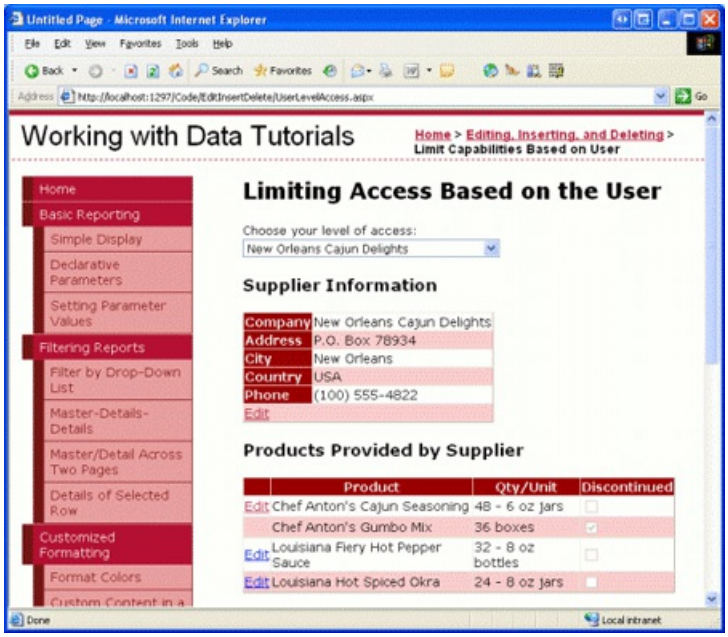


Figure 14: For Supplier-Specific Users the Edit Button for Chef Anton s Gumbo Mix is Hidden (Click to view full-size image)

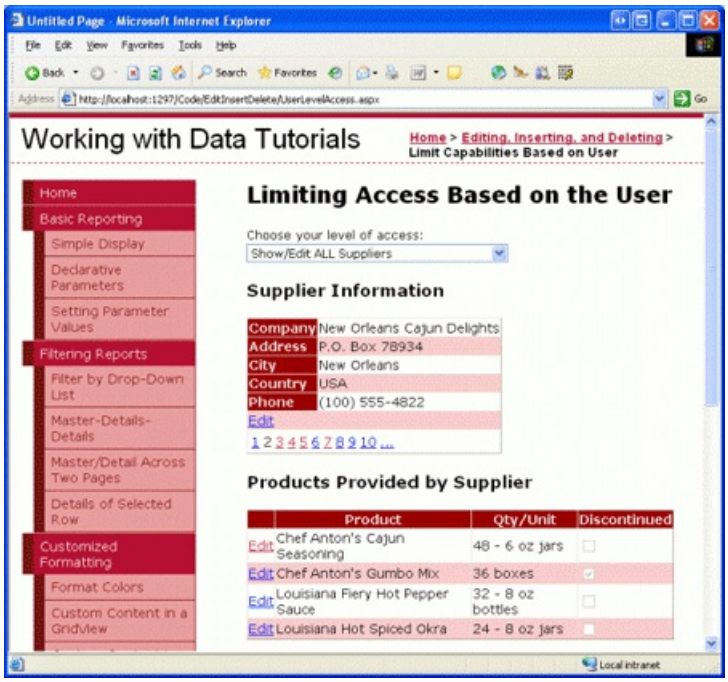


Figure 15: For Show/Edit ALL Suppliers Users, the Edit Button for Chef Anton s Gumbo Mix is Displayed (Click to view full-size image)

Checking for Access Rights in the Business Logic Layer

In this tutorial the ASP.NET page handles all logic with regards to what information the user can see and what products he can update. Ideally, this logic would also be present at the Business Logic Layer. For example, the `SuppliersBLL` class s `GetSuppliers()` method (which returns all suppliers) might include a check to ensure that the currently logged on user is *not* associated with a specific supplier. Likewise, the `UpdateSupplierAddress` method could include a check to ensure that the currently logged on user either worked for our company (and therefore can update all suppliers address information) or is associated with the supplier whose data is being updated.

I did not include such BLL-layer checks here because in our tutorial the user s rights are determined by a DropDownList on the page, which the BLL classes cannot access. When using the membership system or one of the out-of-the box authentication schemes provided by ASP.NET (such as Windows authentication), the currently logged on user s information and roles information can be accessed from the BLL, thereby making such access rights checks possible at both the presentation and BLL layers.

Summary

Most sites that provide user accounts need to customize the data modification interface based upon the logged in user. Administrative users may be able to delete and edit any record, whereas non-administrative users may be limited to only updating or deleting records they created themselves. Whatever the scenario may be, the data Web controls, ObjectDataSource, and Business Logic Layer classes can be extended to add or deny certain functionality based on the logged on user. In this tutorial we saw how to limit the viewable and editable data depending on whether the user was associated with a particular supplier or if they worked for our company.

This tutorial concludes our examination of inserting, updating, and deleting data using the GridView, DetailsView, and FormView controls. Starting with the next tutorial, we'll turn our attention to adding paging and sorting support.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

An Overview of Inserting, Updating, and Deleting Data (VB)

2/20/2020 • 32 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll see how to map an ObjectDataSource's Insert(), Update(), and Delete() methods to the methods of BLL classes, as well as how to configure the GridView, DetailsView, and FormView controls to provide data modification capabilities.

Introduction

Over the past several tutorials we've examined how to display data in an ASP.NET page using the GridView, DetailsView, and FormView controls. These controls simply work with data supplied to them. Commonly, these controls access data through the use of a data source control, such as the ObjectDataSource. We've seen how the ObjectDataSource acts as a proxy between the ASP.NET page and the underlying data. When a GridView needs to display data, it invokes its ObjectDataSource's `Select()` method, which in turn invokes a method from our Business Logic Layer (BLL), which calls a method in the appropriate Data Access Layer's (DAL) TableAdapter, which in turn sends a `SELECT` query to the Northwind database.

Recall that when we created the TableAdapters in the DAL in our first tutorial, Visual Studio automatically added methods for inserting, updating, and deleting data from the underlying database table. Moreover, in Creating a Business Logic Layer we designed methods in the BLL that called down into these data modification DAL methods.

In addition to its `Select()` method, the ObjectDataSource also has `Insert()`, `Update()`, and `Delete()` methods. Like the `Select()` method, these three methods can be mapped to methods in an underlying object. When configured to insert, update, or delete data, the GridView, DetailsView, and FormView controls provide a user interface for modifying the underlying data. This user interface calls the `Insert()`, `Update()`, and `Delete()` methods of the ObjectDataSource, which then invoke the underlying object's associated methods (see Figure 1).

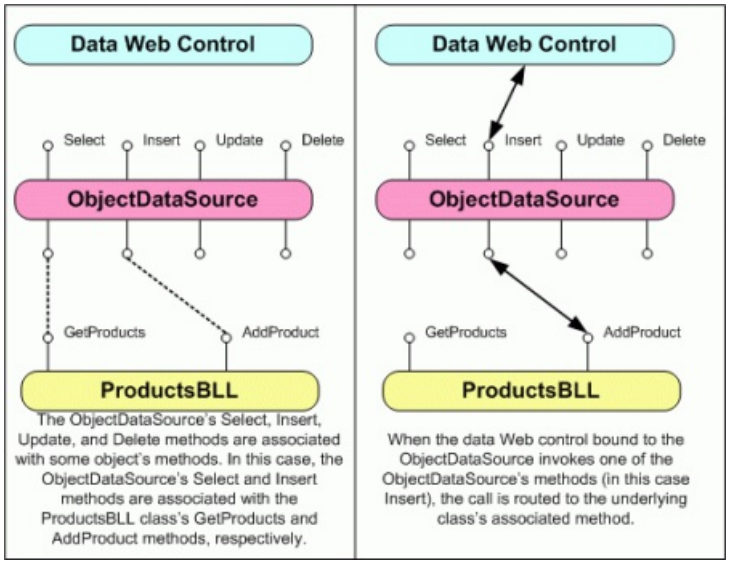


Figure 1: The ObjectDataSource's `Insert()`, `Update()`, and `Delete()` Methods Serve as a Proxy into the BLL (Click to view full-size image)

In this tutorial we'll see how to map the ObjectDataSource's `Insert()`, `Update()`, and `Delete()` methods to methods of classes in the BLL, as well as how to configure the GridView, DetailsView, and FormView controls to provide data modification capabilities.

Step 1: Creating the Insert, Update, and Delete Tutorials Web Pages

Before we start exploring how to insert, update, and delete data, let's first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial and the next several ones. Start by adding a new folder named `EditInsertDelete`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `Basics.aspx`
- `DataModificationEvents.aspx`
- `ErrorHandling.aspx`
- `UIValidation.aspx`
- `CustomizedUI.aspx`
- `OptimisticConcurrency.aspx`
- `ConfirmationOnDelete.aspx`
- `UserLevelAccess.aspx`

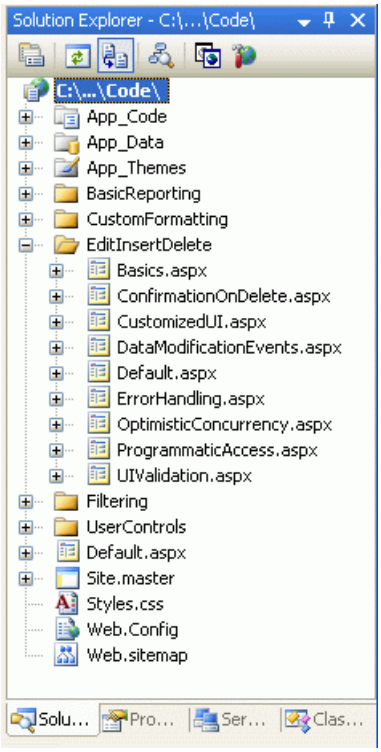


Figure 2: Add the ASP.NET Pages for the Data Modification-Related Tutorials

Like in the other folders, `Default.aspx` in the `EditInsertDelete` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page's Design view.

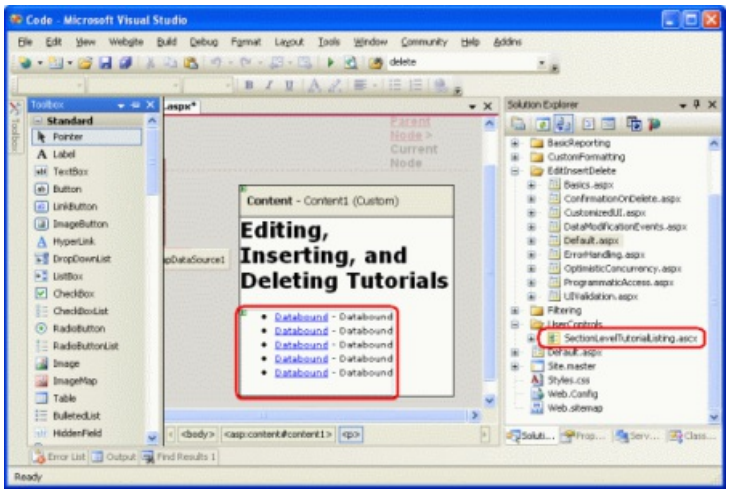


Figure 3: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add the pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Customized Formatting `<siteMapNode>`:

```
<siteMapNode title="Editing, Inserting, and Deleting" url="~/EditInsertDelete/Default.aspx"
description="Samples of Reports that Provide Editing, Inserting, and Deleting Capabilities"> <siteMapNode
url="~/EditInsertDelete/Basics.aspx" title="Basics" description="Examines the basics of data modification
with the GridView, DetailsView, and FormView controls." /> <siteMapNode
url="~/EditInsertDelete/DataModificationEvents.aspx" title="Data Modification Events" description="Explores
the events raised by the ObjectDataSource pertinent to data modification." /> <siteMapNode
url="~/EditInsertDelete/ErrorHandling.aspx" title="Error Handling" description="Learn how to gracefully
handle exceptions raised during the data modification workflow." /> <siteMapNode
url="~/EditInsertDelete/UIValidation.aspx" title="Adding Data Entry Validation" description="Help prevent
data entry errors by providing validation." /> <siteMapNode url="~/EditInsertDelete/CustomizedUI.aspx"
title="Customize the User Interface" description="Customize the editing and inserting user interfaces." />
<siteMapNode url="~/EditInsertDelete/OptimisticConcurrency.aspx" title="Optimistic Concurrency"
description="Learn how to help prevent simultaneous users from overwritting one another s changes." />
<siteMapNode url="~/EditInsertDelete/ConfirmationOnDelete.aspx" title="Confirm On Delete"
description="Prompt a user for confirmation when deleting a record." /> <siteMapNode
url="~/EditInsertDelete/UserLevelAccess.aspx" title="Limit Capabilities Based on User" description="Learn
how to limit the data modification functionality based on the user role or permissions." /> </siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the editing, inserting, and deleting tutorials.

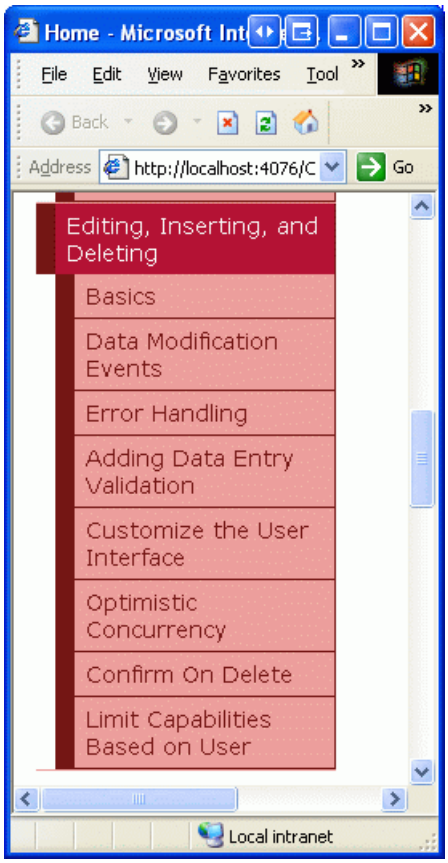


Figure 4: The Site Map Now Includes Entries for the Editing, Inserting, and Deleting Tutorials

Step 2: Adding and Configuring the ObjectDataSource Control

Since the GridView, DetailsView, and FormView each differ in their data modification capabilities and layout, let's examine each one individually. Rather than have each control using its own ObjectDataSource, however, let's just create a single ObjectDataSource that all three control examples can share.

Open the `Basics.aspx` page, drag an ObjectDataSource from the Toolbox onto the Designer, and click the Configure Data Source link from its smart tag. Since the `ProductsBLL` is the only BLL class that provides editing, inserting, and deleting methods, configure the ObjectDataSource to use this class.

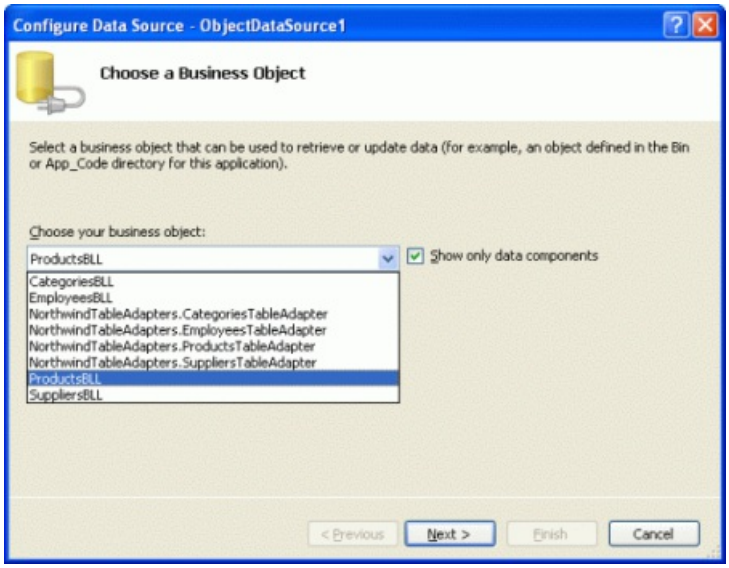


Figure 5: Configure the ObjectDataSource to Use the `ProductsBLL` Class (Click to view full-size image)

In the next screen we can specify what methods of the `ProductsBLL` class are mapped to the ObjectDataSource's `Select()`, `Insert()`, `Update()`, and `Delete()` by selecting the appropriate tab and choosing the method from

the drop-down list. Figure 6, which should look familiar by now, maps the ObjectDataSource's `Select()` method to the `ProductsBLL` class's `GetProducts()` method. The `Insert()`, `Update()`, and `Delete()` methods can be configured by selecting the appropriate tab from the list along the top.

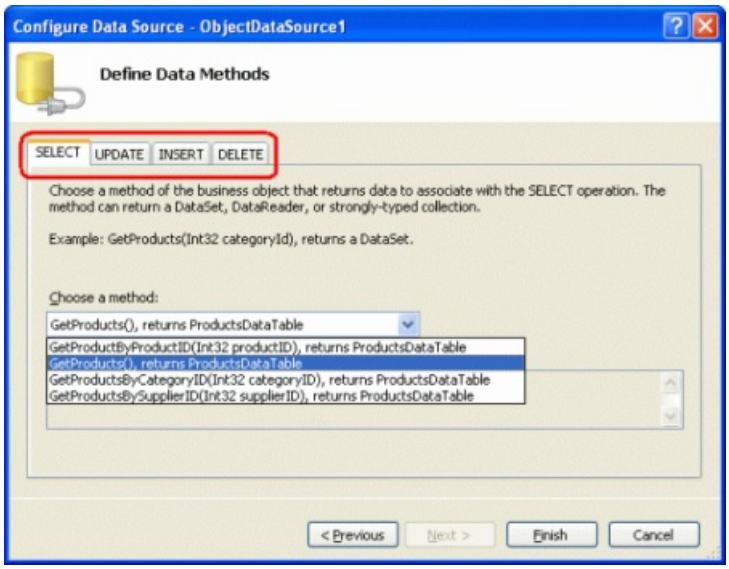


Figure 6: Have the ObjectDataSource Return All of the Products (Click to view full-size image)

Figures 7, 8, and 9 show the ObjectDataSource's UPDATE, INSERT, and DELETE tabs. Configure these tabs so that the `Insert()`, `Update()`, and `Delete()` methods invoke the `ProductsBLL` class's `UpdateProduct`, `AddProduct`, and `DeleteProduct` methods, respectively.

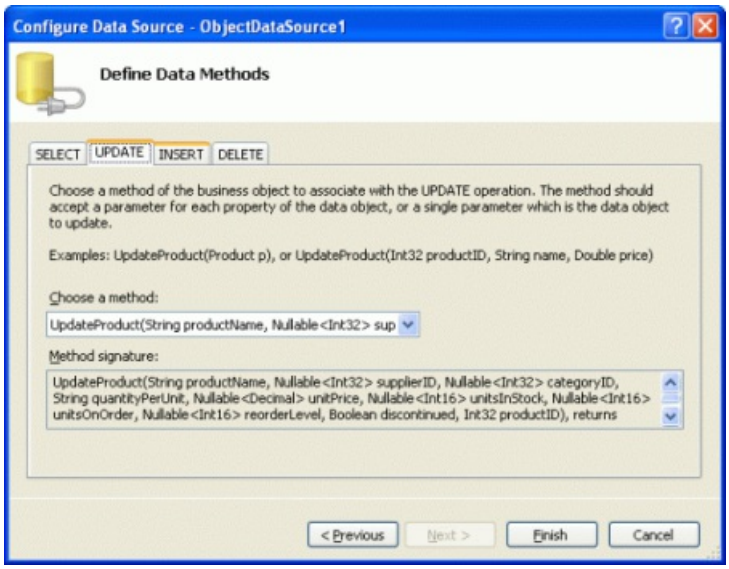


Figure 7: Map the ObjectDataSource's `Update()` Method to the `ProductBLL` Class's `UpdateProduct` Method (Click to view full-size image)

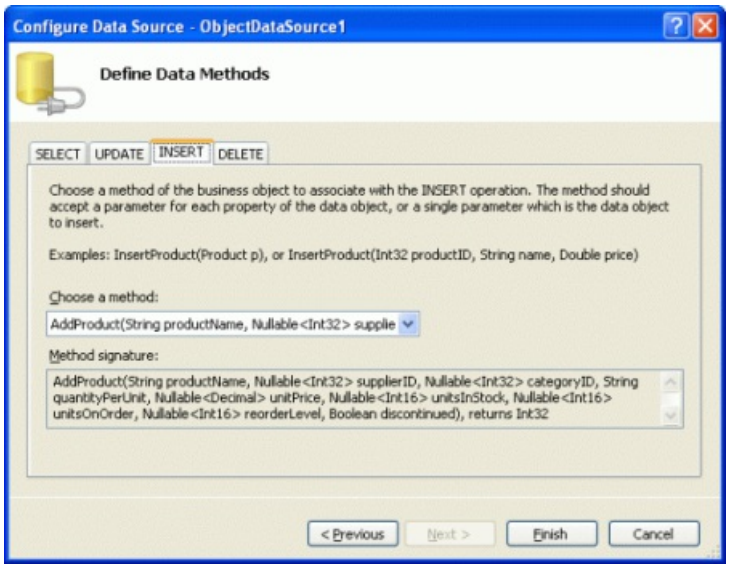


Figure 8: Map the ObjectDataSource's `Insert()` Method to the `ProductBLL` Class's Add `Product` Method (Click to view full-size image)

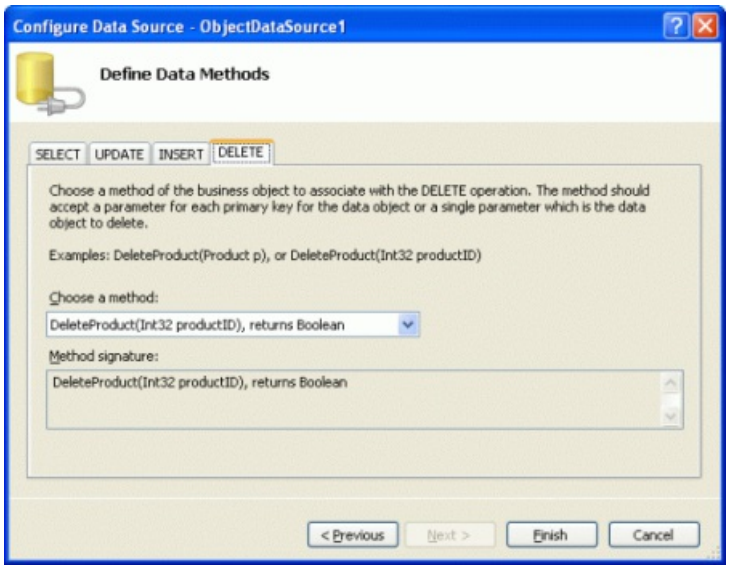


Figure 9: Map the ObjectDataSource's `Delete()` Method to the `ProductBLL` Class's `DeleteProduct` Method (Click to view full-size image)

You may have noticed that the drop-down lists in the UPDATE, INSERT, and DELETE tabs already had these methods selected. This is thanks to our use of the `DataObjectMethodAttribute` that decorates the methods of `ProductsBLL`. For example, the DeleteProduct method has the following signature:

```
<System.ComponentModel.DataObjectMethodAttribute _ (System.ComponentModel.DataObjectMethodType.Delete,
True)> _ Public Function DeleteProduct(ByVal productID As Integer) As Boolean End Function
```

The `DataObjectMethodAttribute` attribute indicates the purpose of each method whether it is for selecting, inserting, updating, or deleting and whether or not it's the default value. If you omitted these attributes when creating your BLL classes, you'll need to manually select the methods from the UPDATE, INSERT, and DELETE tabs.

After ensuring that the appropriate `ProductsBLL` methods are mapped to the ObjectDataSource's `Insert()`, `Update()`, and `Delete()` methods, click Finish to complete the wizard.

Examining the ObjectDataSource's Markup

After configuring the ObjectDataSource through its wizard, go to the Source view to examine the generated declarative markup. The `<asp:ObjectDataSource>` tag specifies the underlying object and the methods to invoke. Additionally, there are `DeleteParameters`, `UpdateParameters`, and `InsertParameters` that map to the input parameters for the `ProductsBLL` class's `AddProduct`, `UpdateProduct`, and `DeleteProduct` methods:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server" DeleteMethod="DeleteProduct"
InsertMethod="AddProduct" OldValuesParameterFormatString="original_{0}" SelectMethod="GetProducts"
TypeName="ProductsBLL" UpdateMethod="UpdateProduct"> <DeleteParameters> <asp:Parameter Name="productID"
Type="Int32" /> </DeleteParameters> <UpdateParameters> <asp:Parameter Name="productName" Type="String" />
<asp:Parameter Name="supplierID" Type="Int32" /> <asp:Parameter Name="categoryID" Type="Int32" />
<asp:Parameter Name="quantityPerUnit" Type="String" /> <asp:Parameter Name="unitPrice" Type="Decimal" />
<asp:Parameter Name="unitsInStock" Type="Int16" /> <asp:Parameter Name="unitsOnOrder" Type="Int16" />
<asp:Parameter Name="reorderLevel" Type="Int16" /> <asp:Parameter Name="discontinued" Type="Boolean" />
<asp:Parameter Name="productID" Type="Int32" /> </UpdateParameters> <InsertParameters> <asp:Parameter
Name="productName" Type="String" /> <asp:Parameter Name="supplierID" Type="Int32" /> <asp:Parameter
Name="categoryID" Type="Int32" /> <asp:Parameter Name="quantityPerUnit" Type="String" /> <asp:Parameter
Name="unitPrice" Type="Decimal" /> <asp:Parameter Name="unitsInStock" Type="Int16" /> <asp:Parameter
Name="unitsOnOrder" Type="Int16" /> <asp:Parameter Name="reorderLevel" Type="Int16" /> <asp:Parameter
Name="discontinued" Type="Boolean" /> </InsertParameters> </asp:ObjectDataSource>
```

The ObjectDataSource includes a parameter for each of the input parameters for its associated methods, just like a list of `SelectParameter` s is present when the ObjectDataSource is configured to call a select method that expects an input parameter (such as `GetProductsByCategoryID(categoryID)`). As we'll see shortly, values for these `DeleteParameters`, `UpdateParameters`, and `InsertParameters` are set automatically by the GridView, DetailsView, and FormView prior to invoking the ObjectDataSource's `Insert()`, `Update()`, or `Delete()` method. These values can also be set programmatically as needed, as we'll discuss in a future tutorial.

One side effect of using the wizard to configure to ObjectDataSource is that Visual Studio sets the OldValuesParameterFormatString property to `original_{0}`. This property value is used to include the original values of the data being edited and is useful in two scenarios:

- If, when editing a record, users are able to change the primary key value. In this case, both the new primary key value and the original primary key value must be provided so that the record with the original primary key value can be found and have its value updated accordingly.
- When using optimistic concurrency. Optimistic concurrency is a technique to ensure that two simultaneous users don't overwrite one another's changes, and is the topic for a future tutorial.

The `OldValuesParameterFormatString` property indicates the name of the input parameters in the underlying object's update and delete methods for the original values. We'll discuss this property and its purpose in greater detail when we explore optimistic concurrency. I bring it up now, however, because our BLL's methods do not expect the original values and therefore it's important that we remove this property. Leaving the `OldValuesParameterFormatString` property set to anything other than the default (`{0}`) will cause an error when a data Web control attempts to invoke the ObjectDataSource's `Update()` or `Delete()` methods because the ObjectDataSource will attempt to pass in both the `UpdateParameters` or `DeleteParameters` specified as well as original value parameters.

If this isn't terribly clear at this juncture, don't worry, we'll examine this property and its utility in a future tutorial. For now, just be certain to either remove this property declaration entirely from the declarative syntax or set the value to the default value ({0}).

> **NOTE**
> If you simply clear out the `OldValuesParameterFormatString` property value from the Properties window in the Design view, the property will still exist in the declarative syntax, but be set to an empty string. This, unfortunately, will still result in the same problem discussed above. Therefore, either remove the property altogether from the declarative syntax or, from the Properties window, set the value to the default, `{0}`.

Step 3: Adding a Data Web Control and Configuring It for Data Modification

Once the ObjectDataSource has been added to the page and configured, we're ready to add data Web controls to the page to both display the data and provide a means for the end user to modify it. We'll look at the GridView, DetailsView, and FormView separately, as these data Web controls differ in their data modification capabilities and configuration.

As we'll see in the remainder of this article, adding very basic editing, inserting, and deleting support through the GridView, DetailsView, and FormView controls is really as simple as checking a couple of checkboxes. There are many subtleties and edge cases in the real-world that make providing such functionality more involved than just point and click. This tutorial, however, focuses solely on proving simplistic data modification capabilities. Future tutorials will examine concerns that will undoubtedly arise in a real-world setting.

Deleting Data from the GridView

Start by dragging a GridView from the Toolbox onto the Designer. Next, bind the ObjectDataSource to the GridView by selecting it from the drop-down list in the GridView's smart tag. At this point the GridView's declarative markup will be:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False" DataKeyNames="ProductID"
DataSourceID="ObjectDataSource1"> <Columns> <asp:BoundField DataField="ProductID" HeaderText="ProductID"
InsertVisible="False" ReadOnly="True" SortExpression="ProductID" /> <asp:BoundField DataField="ProductName"
HeaderText="ProductName" SortExpression="ProductName" /> <asp:BoundField DataField="SupplierID"
HeaderText="SupplierID" SortExpression="SupplierID" /> <asp:BoundField DataField="CategoryID"
HeaderText="CategoryID" SortExpression="CategoryID" /> <asp:BoundField DataField="QuantityPerUnit"
HeaderText="QuantityPerUnit" SortExpression="QuantityPerUnit" /> <asp:BoundField DataField="UnitPrice"
HeaderText="UnitPrice" SortExpression="UnitPrice" /> <asp:BoundField DataField="UnitsInStock"
HeaderText="UnitsInStock" SortExpression="UnitsInStock" /> <asp:BoundField DataField="UnitsOnOrder"
HeaderText="UnitsOnOrder" SortExpression="UnitsOnOrder" /> <asp:BoundField DataField="ReorderLevel"
HeaderText="ReorderLevel" SortExpression="ReorderLevel" /> <asp:CheckBoxField DataField="Discontinued"
HeaderText="Discontinued" SortExpression="Discontinued" /> <asp:BoundField DataField="CategoryName"
HeaderText="CategoryName" ReadOnly="True" SortExpression="CategoryName" /> <asp:BoundField
DataField="SupplierName" HeaderText="SupplierName" ReadOnly="True" SortExpression="SupplierName" />
</Columns> </asp:GridView>
```

Binding the GridView to the ObjectDataSource through its smart tag has two benefits:

- BoundFields and CheckBoxFields are automatically created for each of the fields returned by the ObjectDataSource. Moreover, the BoundField and CheckBoxField's properties are set based on the underlying field's metadata. For example, the `ProductID`, `CategoryName`, and `SupplierName` fields are marked as read-only in the `ProductsDataTable` and therefore shouldn't be updatable when editing. To accommodate this, these BoundFields' ReadOnly properties are set to `True`.
- The DataKeyNames property is assigned to the primary key field(s) of the underlying object. This is essential when using the GridView for editing or deleting data, as this property indicates the field (or set of fields) that unique identifies each record. For more information on the `DataKeyNames` property, refer back to the Master/Detail Using a Selectable Master GridView with a Details DetailView tutorial.

While the GridView can be bound to the ObjectDataSource through the Properties window or declarative syntax, doing so requires you to manually add the appropriate BoundField and `DataKeyNames` markup.

The GridView control provides built-in support for row-level editing and deleting. Configuring a GridView to support deleting adds a column of Delete buttons. When the end user clicks the Delete button for a particular row, a postback ensues and the GridView performs the following steps:

1. The ObjectDataSource's `DeleteParameters` value(s) are assigned
2. The ObjectDataSource's `Delete()` method is invoked, deleting the specified record
3. The GridView rebinds itself to the ObjectDataSource by invoking its `Select()` method

The values assigned to the `DeleteParameters` are the values of the `DataKeyNames` field(s) for the row whose Delete button was clicked. Therefore it's vital that a GridView's `DataKeyNames` property be correctly set. If it's missing, the `DeleteParameters` will be assigned a value of `Nothing` in Step 1, which in turn will not result in any deleted records in Step 2.

> **NOTE**
>
> The `DataKeys` collection is stored in the GridView s control state, meaning that the `DataKeys` values will be remembered across postback even if the GridView s view state has been disabled. However, it is very important that view state remains enabled for GridViews that support editing or deleting (the default behavior). If you set the GridView s `EnableViewState` property to `false`, the editing and deleting behavior will work fine for a single user, but if there are concurrent users deleting data, there exists the possibility that these concurrent users may accidentally delete or edit records that they didn't intend. See my blog entry, WARNING: Concurrency Issue with ASP.NET 2.0 GridViews/DetailsView/FormViews that Support Editing and/or Deleting and Whose View State is Disabled, for more information.

This same warning also applies to DetailsViews and FormViews.

To add deleting capabilities to a GridView, simply go to its smart tag and check the Enable Deleting checkbox.

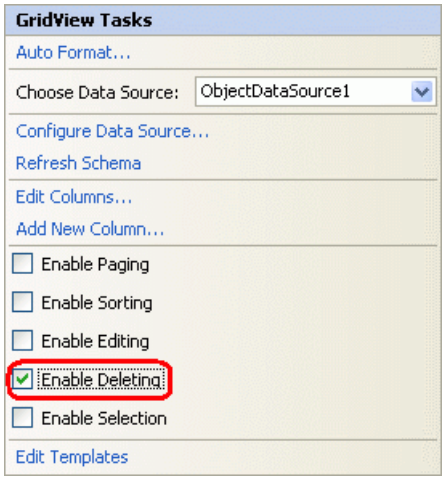


Figure 10: Check the Enable Deleting Checkbox

Checking the Enable Deleting checkbox from the smart tag adds a CommandField to the GridView. The CommandField renders a column in the GridView with buttons for performing one or more of the following tasks: selecting a record, editing a record, and deleting a record. We previously saw the CommandField in action with selecting records in the Master/Detail Using a Selectable Master GridView with a Details DetailView tutorial.

The CommandField contains a number of `ShowXButton` properties that indicate what series of buttons are displayed in the CommandField. By checking the Enable Deleting checkbox a CommandField whose `ShowDeleteButton` property is `True` has been added to the GridView's Columns collection.

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False" DataKeyNames="ProductID"
DataSourceID="ObjectDataSource1"> <Columns> <asp:CommandField ShowDeleteButton="True" /> ... BoundFields
removed for brevity ... </Columns> </asp:GridView>
```

At this point, believe it or not, we're done with adding deleting support to the GridView! As Figure 11 shows, when visiting this page through a browser a column of Delete buttons is present.

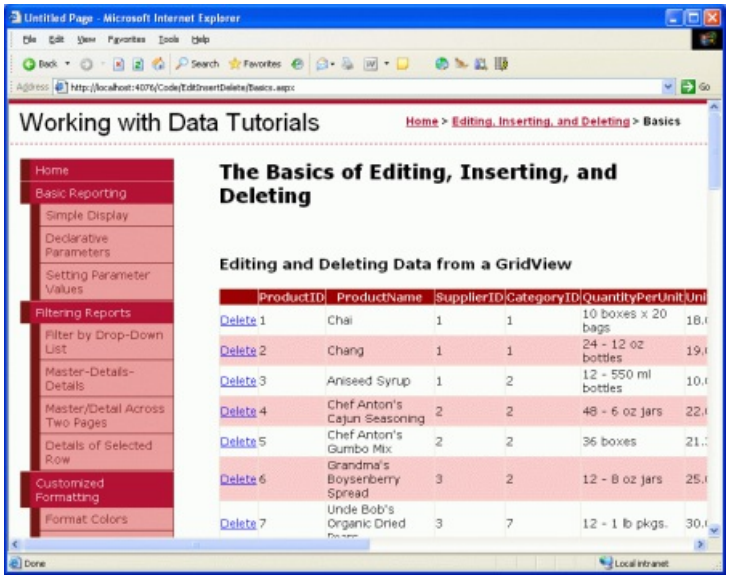


Figure 11: The CommandField Adds a Column of Delete Buttons (Click to view full-size image)

If you've been building this tutorial from the ground up on your own, when testing this page clicking the Delete button will raise an exception. Continue reading to learn as to why these exceptions were raised and how to fix them.

> **NOTE**
>
> If you're following along using the download accompanying this tutorial, these problems have already been accounted for. However, I encourage you to read through the details listed below to help identify problems that may arise and suitable workarounds.

If, when attempting to delete a product, you get an exception whose message is similar to "*ObjectDataSource 'ObjectDataSource1' could not find a non-generic method 'DeleteProduct' that has parameters: productID, original_ProductID*," you likely forgot to remove the `OldValuesParameterFormatString` property from the ObjectDataSource. With the `OldValuesParameterFormatString` property specified, the ObjectDataSource attempts to pass in both `productID` and `original_ProductID` input parameters to the `DeleteProduct` method. `DeleteProduct`, however, only accepts a single input parameter, hence the exception. Removing the `OldValuesParameterFormatString` property (or setting it to `{0}`) instructs the ObjectDataSource to not attempt to pass in the original input parameter.

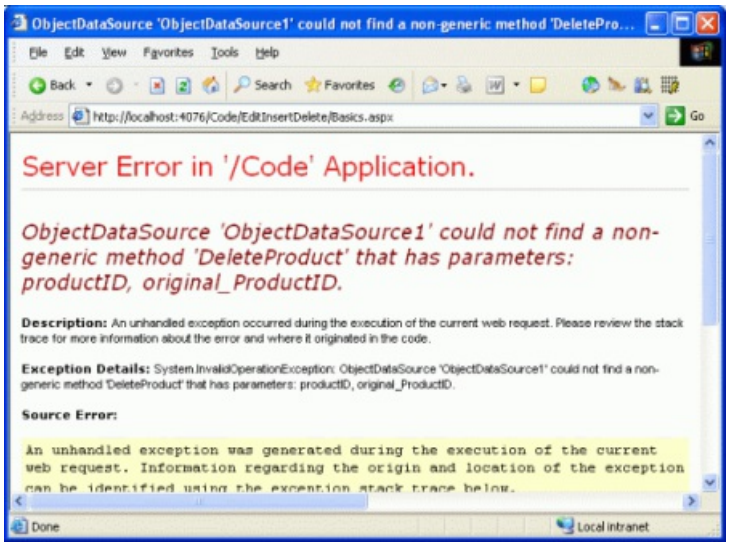


Figure 12: Ensure That the `OldValuesParameterFormatString` Property Has Been Cleared Out (Click to view full-size image)

Even if you had removed the `OldValuesParameterFormatString` property, you still will get an exception when trying to delete a product with the message: "*The DELETE statement conflicted with the REFERENCE constraint 'FK_Order_Details_Products'*." The Northwind database contains a foreign key constraint between the `Order Details` and `Products` table, meaning that a product cannot be deleted from the system if there are one or more records for it in the `Order Details` table. Since every product in the Northwind database has at least one record in `Order Details`, we cannot delete any products until we first delete the product's associated order details records.

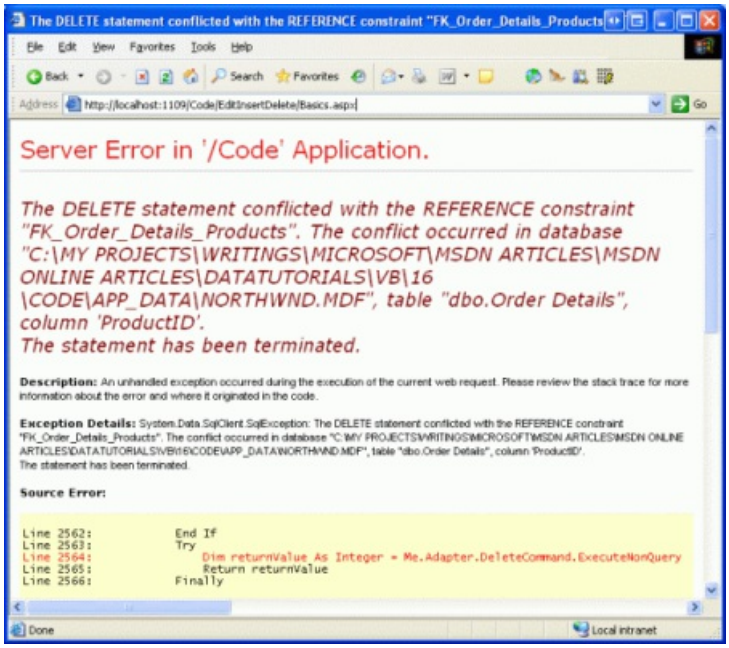


Figure 13: A Foreign Key Constraint Prohibits the Deletion of Products (Click to view full-size image)

For our tutorial, let's just delete all of the records from the `Order Details` table. In a real-world application we'd need to either:

- Have another screen to manage order details information
- Augment the `DeleteProduct` method to include logic to delete the specified product's order details
- Modify the SQL query used by the TableAdapter to include deletion of the specified product's order details

Let's just delete all of the records from the `Order Details` table to circumvent the foreign key constraint. Go to the Server Explorer in Visual Studio, right-click on the `NORTHWND.MDF` node, and choose New Query. Then, in the

query window, run the following SQL statement: `DELETE FROM [Order Details]`

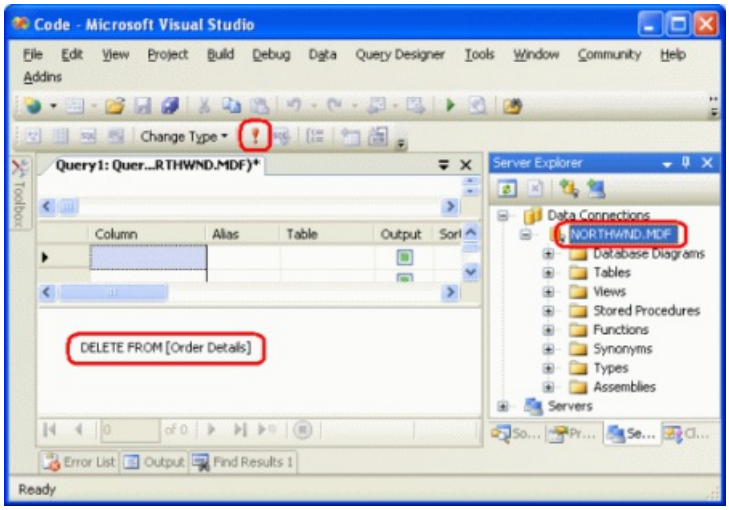


Figure 14: Delete All Records from the `Order Details` Table (Click to view full-size image)

After clearing out the `Order Details` table clicking on the Delete button will delete the product without error. If clicking on the Delete button does not delete the product, check to ensure that the GridView's `DataKeyNames` property is set to the primary key field (`ProductID`).

> **NOTE**
>
> When clicking on the Delete button a postback ensues and the record is deleted. This can be dangerous since it is easy to accidentally click on the wrong row's Delete button. In a future tutorial we'll see how to add a client-side confirmation when deleting a record.

Editing Data with the GridView

Along with deleting, the GridView control also provides built-in row-level editing support. Configuring a GridView to support editing adds a column of Edit buttons. From the end user's perspective, clicking a row's Edit button causes that row to become editable, turning the cells into textboxes containing the existing values and replacing the Edit button with Update and Cancel buttons. After making their desired changes, the end user can click the Update button to commit the changes or the Cancel button to discard them. In either case, after clicking Update or Cancel the GridView returns to its pre-editing state.

From our perspective as the page developer, when the end user clicks the Edit button for a particular row, a postback ensues and the GridView performs the following steps:

1. The GridView's `EditItemIndex` property is assigned to the index of the row whose Edit button was clicked
2. The GridView rebinds itself to the ObjectDataSource by invoking its `Select()` method
3. The row index that matches the `EditItemIndex` is rendered in "edit mode." In this mode, the Edit button is replaced by Update and Cancel buttons and BoundFields whose `ReadOnly` properties are False (the default) are rendered as TextBox Web controls whose `Text` properties are assigned to the data fields' values.

At this point the markup is returned to the browser, allowing the end user to make any changes to the row's data. When the user clicks the Update button, a postback occurs and the GridView performs the following steps:

1. The ObjectDataSource's `UpdateParameters` value(s) are assigned the values entered by the end user into the GridView's editing interface
2. The ObjectDataSource's `Update()` method is invoked, updating the specified record
3. The GridView rebinds itself to the ObjectDataSource by invoking its `Select()` method

The primary key values assigned to the `UpdateParameters` in Step 1 come from the values specified in the `DataKeyNames` property, whereas the non-primary key values come from the text in the TextBox Web controls for the edited row. As with deleting, it is vital that a GridView's `DataKeyNames` property be correctly set. If it's missing, the `UpdateParameters` primary key value will be assigned a value of `Nothing` in Step 1, which in turn will not result in any updated records in Step 2.

Editing functionality can be activated by simply checking the Enable Editing checkbox in the GridView's smart tag.

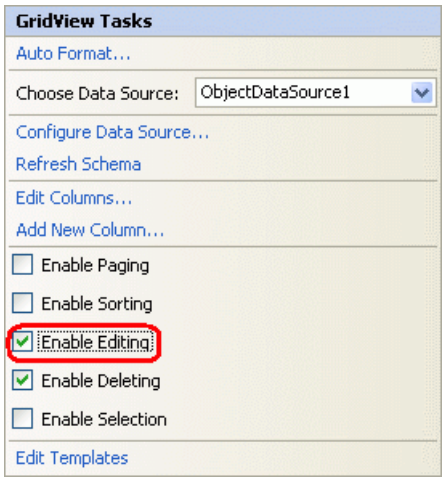


Figure 15: Check the Enable Editing Checkbox

Checking the Enable Editing checkbox will add a CommandField (if needed) and set its `ShowEditButton` property to `True`. As we saw earlier, the CommandField contains a number of `ShowXButton` properties that indicate what series of buttons are displayed in the CommandField. Checking the Enable Editing checkbox adds the `ShowEditButton` property to the existing CommandField:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False" DataKeyNames="ProductID"
DataSourceID="ObjectDataSource1"> <Columns> <asp:CommandField ShowDeleteButton="True" ShowEditButton="True"
/> ... BoundFields removed for brevity ... </Columns> </asp:GridView>
```

That's all there is to adding rudimentary editing support. As Figure16 shows, the editing interface is rather crude each BoundField whose `ReadOnly` property is set to `False` (the default) is rendered as a TextBox. This includes fields like `CategoryID` and `SupplierID`, which are keys to other tables.

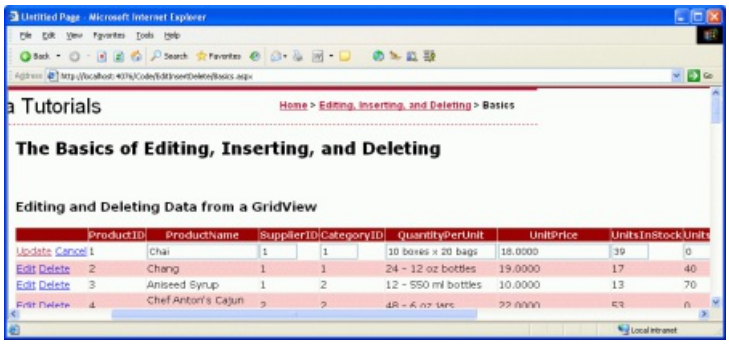


Figure 16: Clicking Chai s Edit Button Displays the Row in Edit Mode (Click to view full-size image)

In addition to asking users to edit foreign key values directly, the editing interface's interface is lacking in the following ways:

- If the user enters a `CategoryID` or `SupplierID` that does not exist in the database, the `UPDATE` will violate a foreign key constraint, causing an exception to be raised.
- The editing interface doesn't include any validation. If you don't provide a required value (such as

`ProductName`), or enter a string value where a numeric value is expected (such as entering "Too much!" into the `UnitPrice` textbox), an exception will be thrown. A future tutorial will examine how to add validation controls to the editing user interface.

- Currently, *all* product fields that are not read-only must be included in the GridView. If we were to remove a field from the GridView, say `UnitPrice`, when updating the data the GridView would not set the `UnitPrice` `UpdateParameters` value, which would change the database record's `UnitPrice` to a `NULL` value. Similarly, if a required field, such as `ProductName`, is removed from the GridView, the update will fail with the same "*Column 'ProductName' does not allow nulls*" exception mentioned above.
- The editing interface formatting leaves a lot to be desired. The `UnitPrice` is shown with four decimal points. Ideally the `CategoryID` and `SupplierID` values would contain DropDownLists that list the categories and suppliers in the system.

These are all shortcomings that we'll have to live with for now, but will be addressed in future tutorials.

Inserting, Editing, and Deleting Data with the DetailsView

As we've seen in earlier tutorials, the DetailsView control displays one record at a time and, like the GridView, allows for editing and deleting of the currently displayed record. Both the end user's experience with editing and deleting items from a DetailsView and the workflow from the ASP.NET side is identical to that of the GridView. Where the DetailsView differs from the GridView is that it also provides built-in inserting support.

To demonstrate the data modification capabilities of the GridView, start by adding a DetailsView to the `Basics.aspx` page above the existing GridView and bind it to the existing ObjectDataSource through the DetailsView's smart tag. Next, clear out the DetailsView's `Height` and `Width` properties, and check the Enable Paging option from the smart tag. To enable editing, inserting, and deleting support, simply check the Enable Editing, Enable Inserting, and Enable Deleting checkboxes in the smart tag.

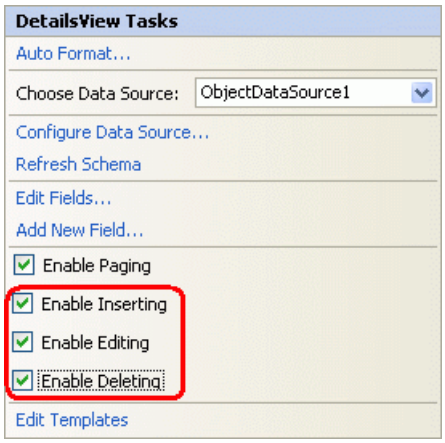


Figure 17: Configure the DetailsView to Support Editing, Inserting, and Deleting

As with the GridView, adding editing, inserting, or deleting support adds a CommandField to the DetailsView, as the following declarative syntax shows:

```
<asp:DetailsView ID="DetailsView1" runat="server" AutoGenerateRows="False" DataKeyNames="ProductID"
DataSourceID="ObjectDataSource1" AllowPaging="True"> <Fields> <asp:BoundField DataField="ProductID"
HeaderText="ProductID" InsertVisible="False" ReadOnly="True" SortExpression="ProductID" /> <asp:BoundField
DataField="ProductName" HeaderText="ProductName" SortExpression="ProductName" /> <asp:BoundField
DataField="SupplierID" HeaderText="SupplierID" SortExpression="SupplierID" /> <asp:BoundField
DataField="CategoryID" HeaderText="CategoryID" SortExpression="CategoryID" /> <asp:BoundField
DataField="QuantityPerUnit" HeaderText="QuantityPerUnit" SortExpression="QuantityPerUnit" />
<asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice" SortExpression="UnitPrice" /> <asp:BoundField
DataField="UnitsInStock" HeaderText="UnitsInStock" SortExpression="UnitsInStock" /> <asp:BoundField
DataField="UnitsOnOrder" HeaderText="UnitsOnOrder" SortExpression="UnitsOnOrder" /> <asp:BoundField
DataField="ReorderLevel" HeaderText="ReorderLevel" SortExpression="ReorderLevel" /> <asp:CheckBoxField
DataField="Discontinued" HeaderText="Discontinued" SortExpression="Discontinued" /> <asp:BoundField
DataField="CategoryName" HeaderText="CategoryName" ReadOnly="True" SortExpression="CategoryName" />
<asp:BoundField DataField="SupplierName" HeaderText="SupplierName" ReadOnly="True"
SortExpression="SupplierName" /> <asp:CommandField ShowDeleteButton="True" ShowEditButton="True"
ShowInsertButton="True" /> </Fields> </asp:DetailsView>
```

Note that for the DetailsView the CommandField appears at the end of the Columns collection by default. Since the DetailsView's fields are rendered as rows, the CommandField appears as a row with Insert, Edit, and Delete buttons at the bottom of the DetailsView.

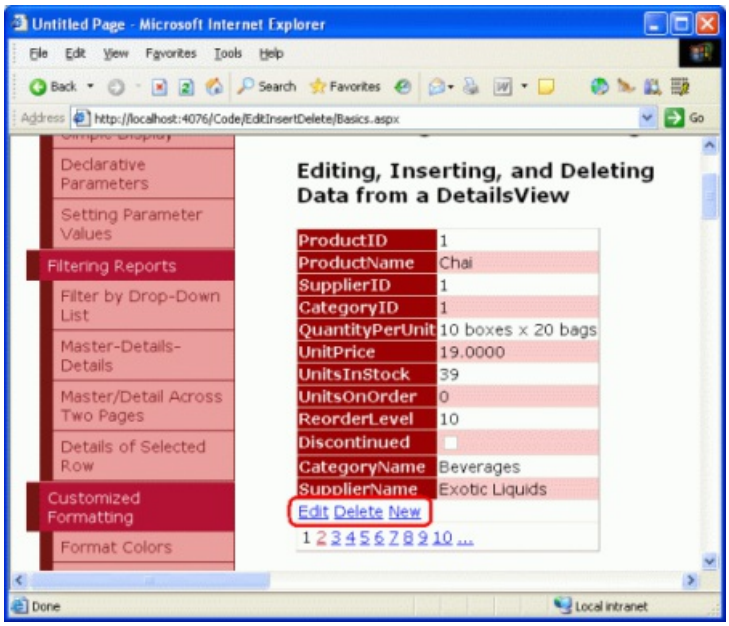


Figure 18: Configure the DetailsView to Support Editing, Inserting, and Deleting (Click to view full-size image)

Clicking on the Delete button starts the same sequence of events as with the GridView: a postback; followed by the DetailsView populating its ObjectDataSource's `DeleteParameters` based on the `DataKeyNames` values; and completed with a call its ObjectDataSource's `Delete()` method, which actually removes the product from the database. Editing in the DetailsView also works in a fashion identical to that of the GridView.

For inserting, the end user is presented with a New button that, when clicked, renders the DetailsView in "insert mode." With "insert mode" the New button is replaced by Insert and Cancel buttons and only those BoundFields whose `InsertVisible` property is set to `True` (the default) are displayed. Those data fields identified as auto-increment fields, such as `ProductID`, have their InsertVisible property set to `False` when binding the DetailsView to the data source through the smart tag.

When binding a data source to a DetailsView through the smart tag, Visual Studio sets the `InsertVisible` property to `False` only for auto-increment fields. Read-only fields, like `CategoryName` and `SupplierName`, will be displayed in the "insert mode" user interface unless their `InsertVisible` property is explicitly set to `False`. Take a moment to set these two fields' `InsertVisible` properties to `False`, either through the DetailsView's declarative syntax or through the Edit Fields link in the smart tag. Figure 19 shows setting the `InsertVisible` properties to `False` by clicking on the Edit Fields link.

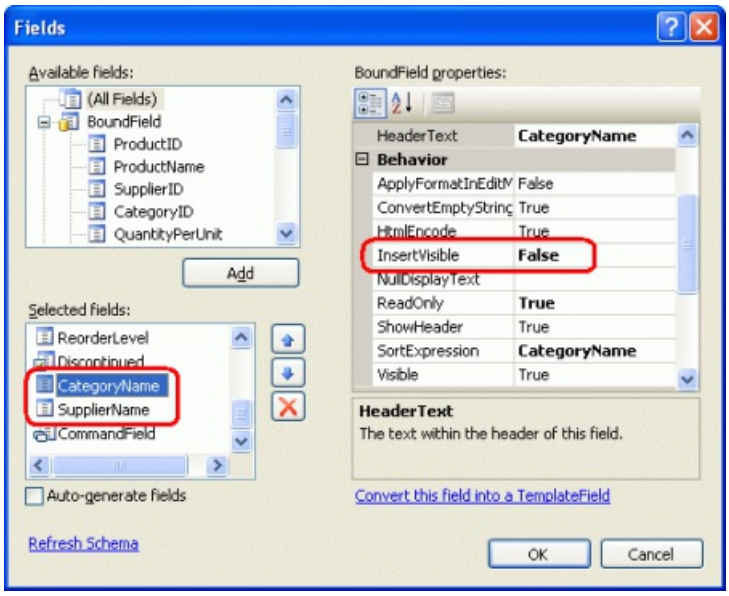


Figure 19: Northwind Traders Now Offers Acme Tea (Click to view full-size image)

After setting the `InsertVisible` properties, view the `Basics.aspx` page in a browser and click the New button. Figure 20 shows the DetailsView when adding a new beverage, Acme Tea, to our product line.

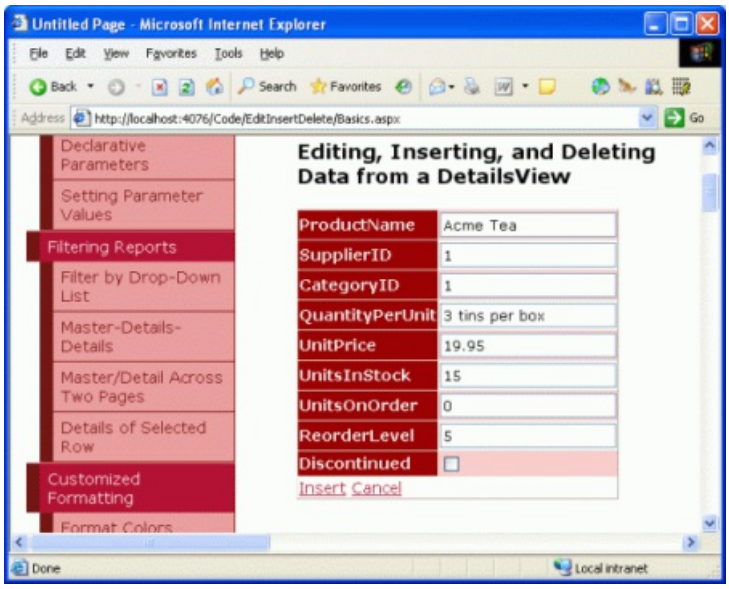


Figure 20: Northwind Traders Now Offers Acme Tea (Click to view full-size image)

After entering the details for Acme Tea and clicking the Insert button, a postback ensues and the new record is added to the `Products` database table. Since this DetailsView lists the products in order with which they exist in the database table, we must page to the last product in order to see the new product.

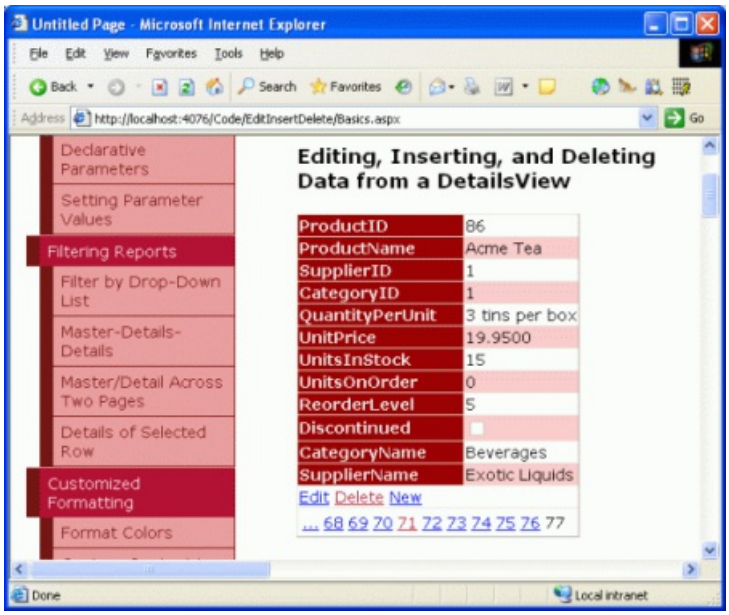


Figure 21: Details for Acme Tea (Click to view full-size image)

> **NOTE**
>
> The DetailsView's CurrentMode property indicates the interface being displayed and can be one of the following values: `Edit`, `Insert`, or `ReadOnly`. The DefaultMode property indicates the mode the DetailsView returns to after an edit or insert has been completed and is useful for displaying a DetailsView that is permanently in edit or insert mode.

The point and click inserting and editing capabilities of the DetailsView suffer from the same limitations as the GridView: the user must enter existing `CategoryID` and `SupplierID` values through a textbox; the interface lacks any validation logic; all product fields that do not allow `NULL` values or don't have a default value specified at the database level must be included in the inserting interface, and so on.

The techniques we will examine for extending and enhancing the GridView's editing interface in future articles can be applied to the DetailsView control's editing and inserting interfaces as well.

Using the FormView for a More Flexible Data Modification User Interface

The FormView offers built-in support for inserting, editing, and deleting data, but because it uses templates instead of fields there's no place to add the BoundFields or the CommandField used by the GridView and DetailsView controls to provide the data modification interface. Instead, this interface the Web controls for collecting user input when adding a new item or editing an existing one along with the New, Edit, Delete, Insert, Update, and Cancel buttons must be added manually to the appropriate templates. Fortunately, Visual Studio will automatically create the needed interface when binding the FormView to a data source through the drop-down list in its smart tag.

To illustrate these techniques, start by adding a FormView to the `Basics.aspx` page and, from the FormView's smart tag, bind it to the ObjectDataSource already created. This will generate an `EditItemTemplate`, `InsertItemTemplate`, and `ItemTemplate` for the FormView with TextBox Web controls for collecting the user's input and Button Web controls for the New, Edit, Delete, Insert, Update, and Cancel buttons. Additionally, the FormView's `DataKeyNames` property is set to the primary key field (`ProductID`) of the object returned by the ObjectDataSource. Lastly, check the Enable Paging option in the FormView's smart tag.

The following shows the declarative markup for the FormView's `ItemTemplate` after the FormView has been bound to the ObjectDataSource. By default, each non-Boolean value product field is bound to the `Text` property of a Label Web control while each Boolean value field (`Discontinued`) is bound to the `Checked`

property of a disabled CheckBox Web control. In order for the New, Edit, and Delete buttons to trigger certain FormView behavior when clicked, it is imperative that their `CommandName` values be set to `New`, `Edit`, and `Delete`, respectively.

```
<asp:FormView ID="FormView1" runat="server" DataKeyNames="ProductID" DataSourceID="ObjectDataSource1"
AllowPaging="True"> <EditItemTemplate> ... </EditItemTemplate> <InsertItemTemplate> ...
</InsertItemTemplate> <ItemTemplate> ProductID: <asp:Label ID="ProductIDLabel" runat="server" Text='<%#
Eval("ProductID") %>'></asp:Label><br /> ProductName: <asp:Label ID="ProductNameLabel" runat="server"
Text='<%# Bind("ProductName") %>'> </asp:Label><br /> SupplierID: <asp:Label ID="SupplierIDLabel"
runat="server" Text='<%# Bind("SupplierID") %>'> </asp:Label><br /> CategoryID: <asp:Label
ID="CategoryIDLabel" runat="server" Text='<%# Bind("CategoryID") %>'> </asp:Label><br /> QuantityPerUnit:
<asp:Label ID="QuantityPerUnitLabel" runat="server" Text='<%# Bind("QuantityPerUnit") %>'> </asp:Label><br
/> UnitPrice: <asp:Label ID="UnitPriceLabel" runat="server" Text='<%# Bind("UnitPrice") %>'></asp:Label><br
/> UnitsInStock: <asp:Label ID="UnitsInStockLabel" runat="server" Text='<%# Bind("UnitsInStock") %>'>
</asp:Label><br /> UnitsOnOrder: <asp:Label ID="UnitsOnOrderLabel" runat="server" Text='<%#
Bind("UnitsOnOrder") %>'> </asp:Label><br /> ReorderLevel: <asp:Label ID="ReorderLevelLabel" runat="server"
Text='<%# Bind("ReorderLevel") %>'> </asp:Label><br /> Discontinued: <asp:CheckBox
ID="DiscontinuedCheckBox" runat="server" Checked='<%# Bind("Discontinued") %>' Enabled="false" /><br />
CategoryName: <asp:Label ID="CategoryNameLabel" runat="server" Text='<%# Bind("CategoryName") %>'>
</asp:Label><br /> SupplierName: <asp:Label ID="SupplierNameLabel" runat="server" Text='<%#
Bind("SupplierName") %>'> </asp:Label><br /> <asp:LinkButton ID="EditButton" runat="server"
CausesValidation="False" CommandName="Edit" Text="Edit"> </asp:LinkButton> <asp:LinkButton
ID="DeleteButton" runat="server" CausesValidation="False" CommandName="Delete" Text="Delete">
</asp:LinkButton> <asp:LinkButton ID="NewButton" runat="server" CausesValidation="False" CommandName="New"
Text="New"> </asp:LinkButton> </ItemTemplate> </asp:FormView>
```

Figure 22 shows the FormView's `ItemTemplate` when viewed through a browser. Each product field is listed with the New, Edit, and Delete buttons at the bottom.

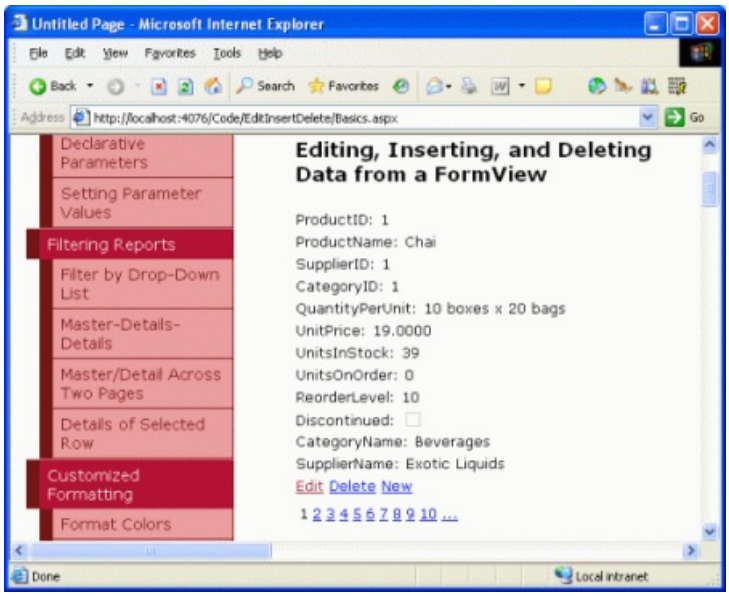


Figure 22: The Defaut FormView `ItemTemplate` Lists Each Product Field Along with New, Edit, and Delete Buttons (Click to view full-size image)

Like with the GridView and DetailsView, clicking the Delete button or any Button, LinkButton, or ImageButton whose `CommandName` property is set to Delete causes a postback, populates the ObjectDataSource's `DeleteParameters` based on the FormView's `DataKeyNames` value, and invokes the ObjectDataSource's `Delete()` method.

When the Edit button is clicked a postback ensues and the data is rebound to the `EditItemTemplate`, which is responsible for rendering the editing interface. This interface includes the Web controls for editing data along with the Update and Cancel buttons. The default `EditItemTemplate` generated by Visual Studio contains a Label for any auto-increment fields (`ProductID`), a TextBox for each non-Boolean value field, and a CheckBox for each Boolean value field. This behavior is very similar to the auto-generated BoundFields in the GridView and

DetailsView controls.

> **NOTE**
> One small issue with the FormView's auto-generation of the `EditItemTemplate` is that it renders TextBox Web controls for those fields that are read-only, such as `CategoryName` and `SupplierName`. We'll see how to account for this shortly.

The TextBox controls in the `EditItemTemplate` have their `Text` property bound to the value of their corresponding data field using *two-way databinding*. Two-way databinding, denoted by `<%# Bind("dataField") %>`, performs databinding both when binding data to the template and when populating the ObjectDataSource's parameters for inserting or editing records. That is, when the user clicks the Edit button from the `ItemTemplate`, the `Bind()` method returns the specified data field value. After the user makes their changes and clicks Update, the values posted back that correspond to the data fields specified using `Bind()` are applied to the ObjectDataSource's `UpdateParameters`. Alternatively, one-way databinding, denoted by `<%# Eval("dataField") %>`, only retrieves the data field values when binding data to the template and does *not* return the user-entered values to the data source's parameters on postback.

The following declarative markup shows the FormView's `EditItemTemplate`. Note that the `Bind()` method is used in the databinding syntax here and that the Update and Cancel Button Web controls have their `CommandName` properties set accordingly.

```
<asp:FormView ID="FormView1" runat="server" DataKeyNames="ProductID" DataSourceID="ObjectDataSource1"
AllowPaging="True"> <EditItemTemplate> ProductID: <asp:Label ID="ProductIDLabel1" runat="server" Text="<%#
Eval("ProductID") %>"></asp:Label><br /> ProductName: <asp:TextBox ID="ProductNameTextBox" runat="server"
Text="<%# Bind("ProductName") %>"> </asp:TextBox><br /> SupplierID: <asp:TextBox ID="SupplierIDTextBox"
runat="server" Text="<%# Bind("SupplierID") %>"> </asp:TextBox><br /> CategoryID: <asp:TextBox
ID="CategoryIDTextBox" runat="server" Text="<%# Bind("CategoryID") %>"> </asp:TextBox><br />
QuantityPerUnit: <asp:TextBox ID="QuantityPerUnitTextBox" runat="server" Text="<%# Bind("QuantityPerUnit")
%>"> </asp:TextBox><br /> UnitPrice: <asp:TextBox ID="UnitPriceTextBox" runat="server" Text="<%#
Bind("UnitPrice") %>"> </asp:TextBox><br /> UnitsInStock: <asp:TextBox ID="UnitsInStockTextBox"
runat="server" Text="<%# Bind("UnitsInStock") %>"> </asp:TextBox><br /> UnitsOnOrder: <asp:TextBox
ID="UnitsOnOrderTextBox" runat="server" Text="<%# Bind("UnitsOnOrder") %>"> </asp:TextBox><br />
ReorderLevel: <asp:TextBox ID="ReorderLevelTextBox" runat="server" Text="<%# Bind("ReorderLevel") %>">
</asp:TextBox><br /> Discontinued: <asp:CheckBox ID="DiscontinuedCheckBox" runat="server" Checked="<%#
Bind("Discontinued") %>" /><br /> CategoryName: <asp:TextBox ID="CategoryNameTextBox" runat="server" Text="
<%# Bind("CategoryName") %>"> </asp:TextBox><br /> SupplierName: <asp:TextBox ID="SupplierNameTextBox"
runat="server" Text="<%# Bind("SupplierName") %>"> </asp:TextBox><br /> <asp:LinkButton ID="UpdateButton"
runat="server" CausesValidation="True" CommandName="Update" Text="Update"> </asp:LinkButton>
<asp:LinkButton ID="UpdateCancelButton" runat="server" CausesValidation="False" CommandName="Cancel"
Text="Cancel"> </asp:LinkButton> </EditItemTemplate> <InsertItemTemplate> ... </InsertItemTemplate>
<ItemTemplate> ... </ItemTemplate> </asp:FormView>
```

Our `EditItemTemplate`, at this point, will cause an exception to be thrown if we attempt to use it. The problem is that the `CategoryName` and `SupplierName` fields are rendered as TextBox Web controls in the `EditItemTemplate`. We either need to change these TextBoxes to Labels or remove them altogether. Let's simply delete them entirely from the `EditItemTemplate`.

Figure 23 shows the FormView in a browser after the Edit button has been clicked for Chai. Note that the `SupplierName` and `CategoryName` fields shown in the `ItemTemplate` are no longer present, as we just removed them from the `EditItemTemplate`. When the Update button is clicked the FormView proceeds through the same sequence of steps as the GridView and DetailsView controls.

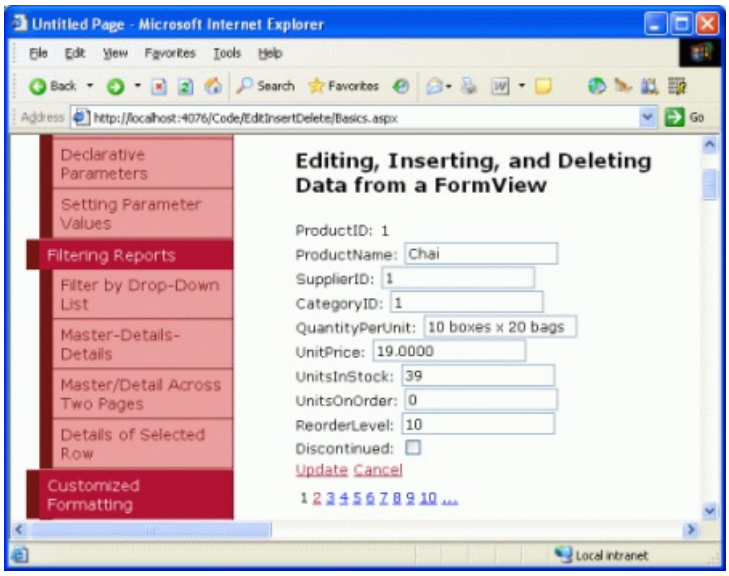


Figure 23: By Default the `EditItemTemplate` Shows Each Editable Product Field as a TextBox or CheckBox (Click to view full-size image)

When the Insert button is clicked the FormView's `ItemTemplate` a postback ensues. However, no data is bound to the FormView because a new record is being added. The `InsertItemTemplate` interface includes the Web controls for adding a new record along with the Insert and Cancel buttons. The default `InsertItemTemplate` generated by Visual Studio contains a TextBox for each non-Boolean value field and a CheckBox for each Boolean value field, similar to the auto-generated `EditItemTemplate`'s interface. The TextBox controls have their `Text` property bound to the value of their corresponding data field using two-way databinding.

The following declarative markup shows the FormView's `InsertItemTemplate`. Note that the `Bind()` method is used in the databinding syntax here and that the Insert and Cancel Button Web controls have their `CommandName` properties set accordingly.

```
<asp:FormView ID="FormView1" runat="server" DataKeyNames="ProductID" DataSourceID="ObjectDataSource1"
AllowPaging="True"> <EditItemTemplate> ... </EditItemTemplate> <InsertItemTemplate> ProductName:
<asp:TextBox ID="ProductNameTextBox" runat="server" Text="<%# Bind("ProductName") %>"> </asp:TextBox><br />
SupplierID: <asp:TextBox ID="SupplierIDTextBox" runat="server" Text="<%# Bind("SupplierID") %>">
</asp:TextBox><br /> CategoryID: <asp:TextBox ID="CategoryIDTextBox" runat="server" Text="<%#
Bind("CategoryID") %>"> </asp:TextBox><br /> QuantityPerUnit: <asp:TextBox ID="QuantityPerUnitTextBox"
runat="server" Text="<%# Bind("QuantityPerUnit") %>"> </asp:TextBox><br /> UnitPrice: <asp:TextBox
ID="UnitPriceTextBox" runat="server" Text="<%# Bind("UnitPrice") %>"> </asp:TextBox><br /> UnitsInStock:
<asp:TextBox ID="UnitsInStockTextBox" runat="server" Text="<%# Bind("UnitsInStock") %>"> </asp:TextBox><br
/> UnitsOnOrder: <asp:TextBox ID="UnitsOnOrderTextBox" runat="server" Text="<%# Bind("UnitsOnOrder") %>">
</asp:TextBox><br /> ReorderLevel: <asp:TextBox ID="ReorderLevelTextBox" runat="server" Text="<%#
Bind("ReorderLevel") %>"> </asp:TextBox><br /> Discontinued: <asp:CheckBox ID="DiscontinuedCheckBox"
runat="server" Checked="<%# Bind("Discontinued") %>" /><br /> CategoryName: <asp:TextBox
ID="CategoryNameTextBox" runat="server" Text="<%# Bind("CategoryName") %>"> </asp:TextBox><br />
SupplierName: <asp:TextBox ID="SupplierNameTextBox" runat="server" Text="<%# Bind("SupplierName") %>">
</asp:TextBox><br /> <asp:LinkButton ID="InsertButton" runat="server" CausesValidation="True"
CommandName="Insert" Text="Insert"> </asp:LinkButton> <asp:LinkButton ID="InsertCancelButton"
runat="server" CausesValidation="False" CommandName="Cancel" Text="Cancel"> </asp:LinkButton>
</InsertItemTemplate> <ItemTemplate> ... </ItemTemplate> </asp:FormView>
```

There's a subtlety with the FormView's auto-generation of the `InsertItemTemplate`. Specifically, the TextBox Web controls are created even for those fields that are read-only, such as `CategoryName` and `SupplierName`. Like with the `EditItemTemplate`, we need to remove these TextBoxes from the `InsertItemTemplate`.

Figure 24 shows the FormView in a browser when adding a new product, Acme Coffee. Note that the `SupplierName` and `CategoryName` fields shown in the `ItemTemplate` are no longer present, as we just removed them. When the Insert button is clicked the FormView proceeds through the same sequence of steps as the DetailsView control, adding a new record to the `Products` table. Figure 25 shows Acme Coffee product's details

in the FormView after it has been inserted.

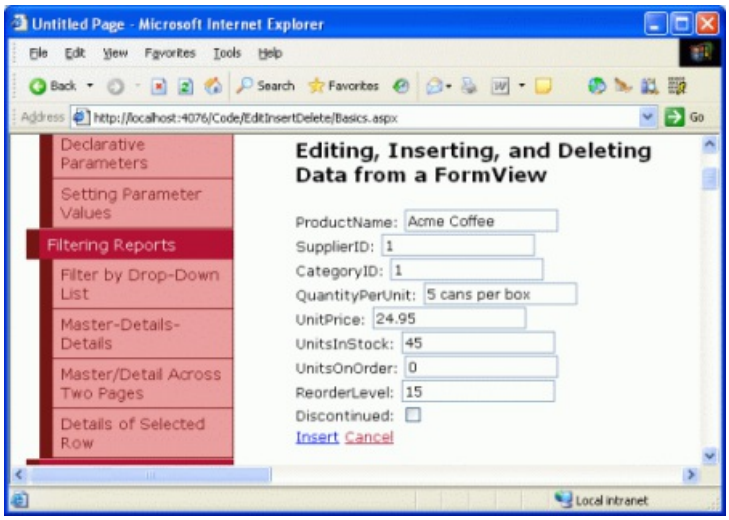


Figure 24: The `InsertItemTemplate` Dictates the FormView's Inserting Interface (Click to view full-size image)

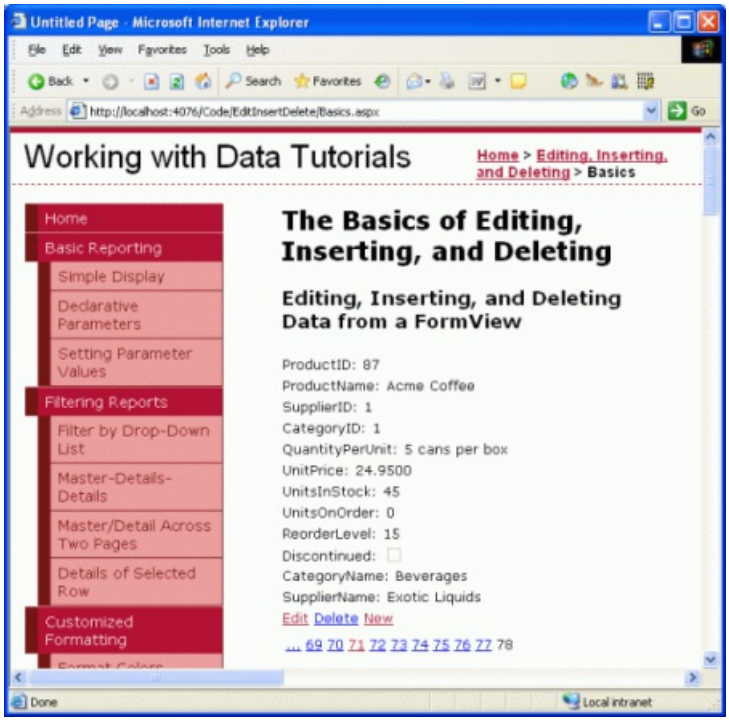


Figure 25: The Details for New Product, Acme Coffee, are Displayed in the FormView (Click to view full-size image)

By separating out the read-only, editing, and inserting interfaces into three separate templates, the FormView allows for a finer degree of control over these interfaces than the DetailsView and GridView.

> **NOTE**
>
> Like the DetailsView, the FormView's `CurrentMode` property indicates the interface being displayed and its `DefaultMode` property indicates the mode the FormView returns to after an edit or insert has been completed.

Summary

In this tutorial we examined the basics of inserting, editing, and deleting data using the GridView, DetailsView, and FormView. All three of these controls provide some level of built-in data modification capabilities that can be utilized without writing a single line of code in the ASP.NET page thanks to the data Web controls and the ObjectDataSource. However, the simple point and click techniques render a fairly frail and naïve data

modification user interface. To provide validation, inject programmatic values, gracefully handle exceptions, customize the user interface, and so on, we'll need to rely on a bevy of techniques that will be discussed over the next several tutorials.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Examining the Events Associated with Inserting, Updating, and Deleting (VB)

2/20/2020 • 22 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

In this tutorial we'll examine using the events that occur before, during, and after an insert, update, or delete operation of an ASP.NET data Web control. We'll also see how to customize the editing interface to only update a subset of the product fields.

Introduction

When using the built-in inserting, editing, or deleting features of the GridView, DetailsView, or FormView controls, a variety of steps transpire when the end user completes the process of adding a new record or updating or deleting an existing record. As we discussed in the previous tutorial, when a row is edited in the GridView the Edit button is replaced by Update and Cancel buttons and the BoundFields turn into TextBoxes. After the end user updates the data and clicks Update, the following steps are performed on postback:

1. The GridView populates its ObjectDataSource's `UpdateParameters` with the edited record's unique identifying field(s) (via the `DataKeyNames` property) along with the values entered by the user
2. The GridView invokes its ObjectDataSource's `Update()` method, which in turn invokes the appropriate method in the underlying object (`ProductsDAL.UpdateProduct`, in our previous tutorial)
3. The underlying data, which now includes the updated changes, is rebound to the GridView

During this sequence of steps, a number of events fire, enabling us to create event handlers to add custom logic where needed. For example, prior to Step 1, the GridView's `RowUpdating` event fires. We can, at this point, cancel the update request if there is some validation error. When the `Update()` method is invoked, the ObjectDataSource's `Updating` event fires, providing an opportunity to add or customize the values of any of the `UpdateParameters`. After the ObjectDataSource's underlying object's method has completed executing, the ObjectDataSource's `Updated` event is raised. An event handler for the `Updated` event can inspect the details about the update operation, such as how many rows were affected and whether or not an exception occurred. Finally, after Step 2, the GridView's `RowUpdated` event fires; an event handler for this event can examine additional information about the update operation just performed.

Figure 1 depicts this series of events and steps when updating a GridView. The event pattern in Figure 1 is not unique to updating with a GridView. Inserting, updating, or deleting data from the GridView, DetailsView, or FormView precipitates the same sequence of pre- and post-level events for both the data Web control and the ObjectDataSource.

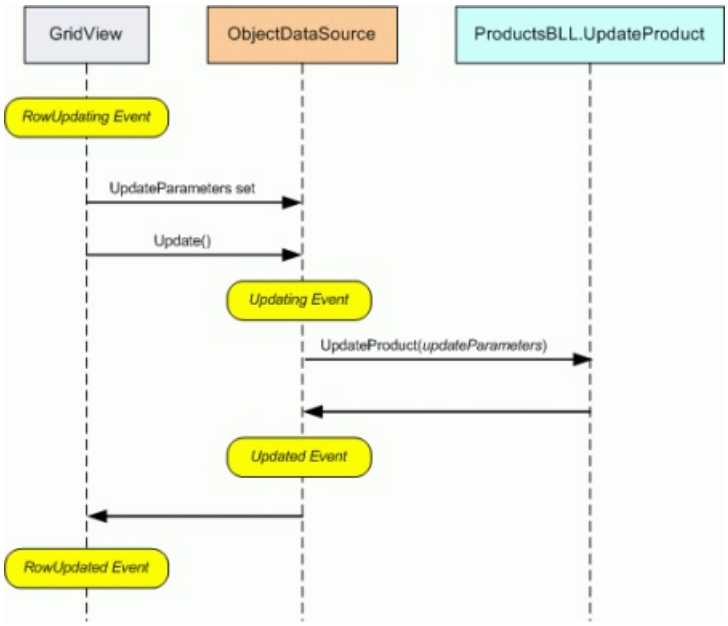


Figure 1: A Series of Pre- and Post-Events Fire When Updating Data in a GridView (Click to view full-size image)

In this tutorial we'll examine using these events to extend the built-in inserting, updating, and deleting capabilities of the ASP.NET data Web controls. We'll also see how to customize the editing interface to only update a subset of the product fields.

Step 1: Updating a Product's `ProductName` and `UnitPrice` Fields

In the editing interfaces from the previous tutorial *all* product fields that were not read-only had to be included. If we were to remove a field from the GridView - say `QuantityPerUnit` - when updating the data the data Web control would not set the ObjectDataSource's `QuantityPerUnit` `UpdateParameters` value. The ObjectDataSource would then pass in a value of `Nothing` into the `UpdateProduct` Business Logic Layer (BLL) method, which would change the edited database record's `QuantityPerUnit` column to a `NULL` value. Similarly, if a required field, such as `ProductName`, is removed from the editing interface, the update will fail with a "*Column 'ProductName' does not allow nulls*" exception. The reason for this behavior was because the ObjectDataSource was configured to call the `ProductsBLL` class's `UpdateProduct` method, which expected an input parameter for each of the product fields. Therefore, the ObjectDataSource's `UpdateParameters` collection contained a parameter for each of the method's input parameters.

If we want to provide a data Web control that allows the end user to only update a subset of fields, then we need to either programmatically set the missing `UpdateParameters` values in the ObjectDataSource's `Updating` event handler or create and call a BLL method that expects only a subset of the fields. Let's explore this latter approach.

Specifically, let's create a page that displays just the `ProductName` and `UnitPrice` fields in an editable GridView. This GridView's editing interface will only allow the user to update the two displayed fields, `ProductName` and `UnitPrice`. Since this editing interface only provides a subset of a product's fields, we either need to create an ObjectDataSource that uses the existing BLL's `UpdateProduct` method and has the missing product field values set programmatically in its `Updating` event handler, or we need to create a new BLL method that expects only the subset of fields defined in the GridView. For this tutorial, let's use the latter option and create an overload of the `UpdateProduct` method, one that takes in just three input parameters: `productName`, `unitPrice`, and `productID`:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Update, False)> _
    Public Function UpdateProduct(productName As String, _
        unitPrice As Nullable(Of Decimal), productID As Integer) _
        As Boolean

    Dim products As Northwind.ProductsDataTable = _
        Adapter.GetProductByProductID(productID)

    If products.Count = 0 Then
        Return False
    End If

    Dim product As Northwind.ProductsRow = products(0)

    product.ProductName = productName
    If Not unitPrice.HasValue Then
        product.SetUnitPriceNull()
    Else
        product.UnitPrice = unitPrice.Value
    End If

    Dim rowsAffected As Integer = Adapter.Update(product)

    Return rowsAffected = 1
End Function
```

Like the original `UpdateProduct` method, this overload starts by checking to see if there is a product in the database with the specified `ProductID`. If not, it returns `False`, indicating that the request to update the product information failed. Otherwise it updates the existing product record's `ProductName` and `UnitPrice` fields accordingly and commits the update by calling the TableAdapter's `Update()` method, passing in the `ProductsRow` instance.

With this addition to our `ProductsBLL` class, we're ready to create the simplified GridView interface. Open the `DataModificationEvents.aspx` in the `EditInsertDelete` folder and add a GridView to the page. Create a new ObjectDataSource and configure it to use the `ProductsBLL` class with its `Select()` method mapping to `GetProducts` and its `Update()` method mapping to the `UpdateProduct` overload that takes in only the `productName`, `unitPrice`, and `productID` input parameters. Figure 2 shows the Create Data Source wizard when mapping the ObjectDataSource's `Update()` method to the `ProductsBLL` class's new `UpdateProduct` method overload.

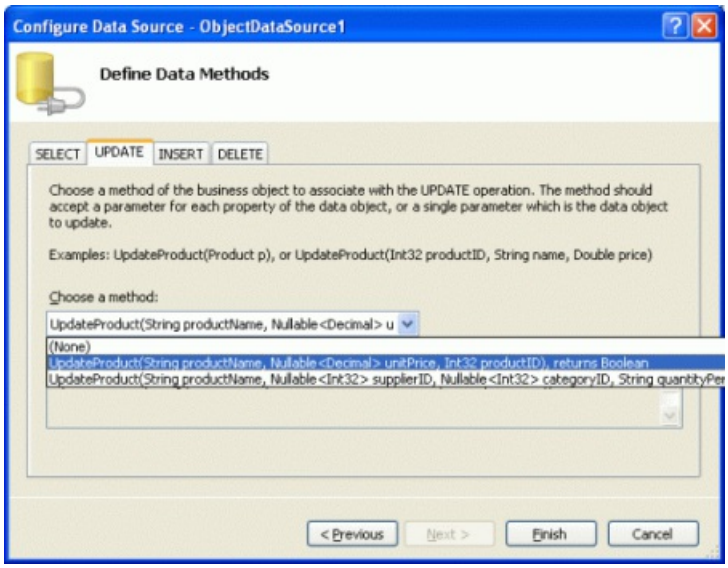


Figure 2: Map the ObjectDataSource's `Update()` Method to the New `UpdateProduct` Overload (Click to view full-size image)

Since our example will initially just need the ability to edit data, but not to insert or delete records, take a moment to explicitly indicate that the ObjectDataSource's `Insert()` and `Delete()` methods shouldn't be mapped to any of the `ProductsBLL` class's methods by going to the INSERT and DELETE tabs and choosing (None) from the drop-down list.

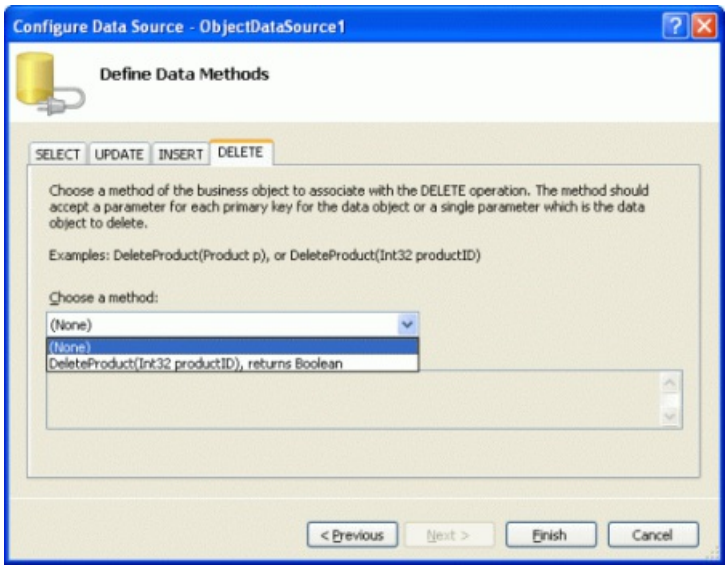


Figure 3: Choose (None) From the Drop-Down List for the INSERT and DELETE Tabs (Click to view full-size image)

After completing this wizard check the Enable Editing checkbox from the GridView's smart tag.

With the completion of the Create Data Source wizard and binding that to the GridView, Visual Studio has created the declarative syntax for both controls. Go to the Source view to inspect the ObjectDataSource's declarative markup, which is shown below:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}" SelectMethod="GetProducts"
    TypeName="ProductsBLL" UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

Since there are no mappings for the ObjectDataSource's `Insert()` and `Delete()` methods, there are no `InsertParameters` or `DeleteParameters` sections. Furthermore, since the `Update()` method is mapped to the `UpdateProduct` method overload that only accepts three input parameters, the `UpdateParameters` section has just three `Parameter` instances.

Note that the ObjectDataSource's `OldValuesParameterFormatString` property is set to `original_{0}`. This property is set automatically by Visual Studio when using the Configure Data Source wizard. However, since our BLL methods don't expect the original `ProductID` value to be passed in, remove this property assignment altogether from the ObjectDataSource's declarative syntax.

> **NOTE**
>
> If you simply clear out the `OldValuesParameterFormatString` property value from the Properties window in the Design view, the property will still exist in the declarative syntax, but will be set to an empty string. Either remove the property altogether from the declarative syntax or, from the Properties window, set the value to the default, `{0}`.

While the ObjectDataSource only has `UpdateParameters` for the product's name, price, and ID, Visual Studio has added a BoundField or CheckBoxField in the GridView for each of the product's fields.

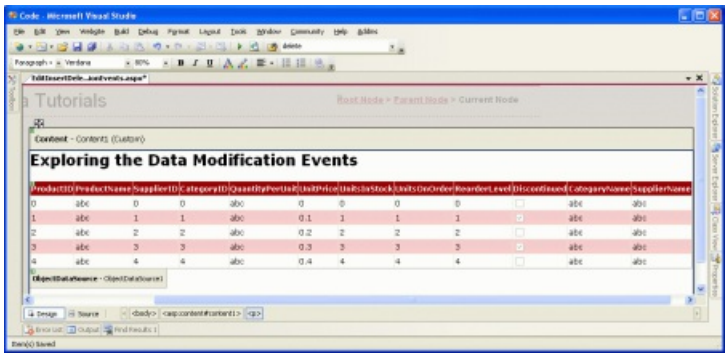


Figure 4: The GridView Contains a BoundField or CheckBoxField for Each of the Product's Fields (Click to view full-size image)

When the end user edits a product and clicks its Update button, the GridView enumerates those fields that were not read-only. It then sets the value of the corresponding parameter in the ObjectDataSource's `UpdateParameters` collection to the value entered by the user. If there is not a corresponding parameter, the GridView adds one to the collection. Therefore, if our GridView contains BoundFields and CheckBoxFields for all of the product's fields, the ObjectDataSource will end up invoking the `UpdateProduct` overload that takes in all of these parameters, despite the fact that the ObjectDataSource's declarative markup specifies only three input parameters (see Figure 5). Similarly, if there is some combination of non-read-only product fields in the GridView that doesn't correspond to the input parameters for a `UpdateProduct` overload, an exception will be raised when attempting to update.

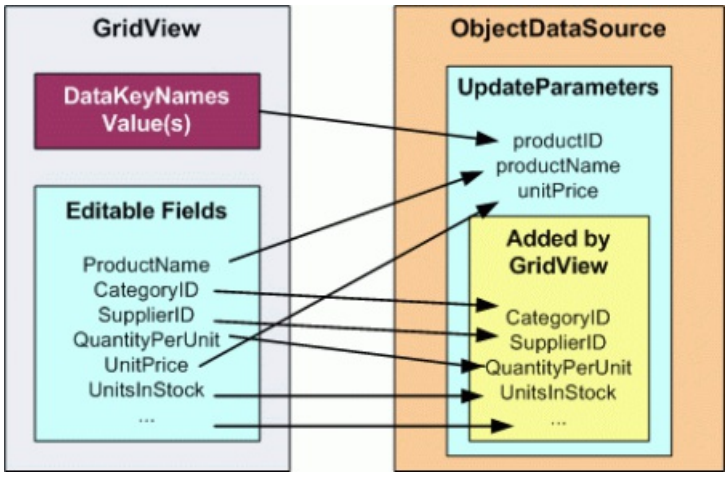


Figure 5: The GridView Will Add Parameters to the ObjectDataSource's `UpdateParameters` Collection (Click to view full-size image)

To ensure that the ObjectDataSource invokes the `UpdateProduct` overload that takes in just the product's name, price, and ID, we need to restrict the GridView to having editable fields for just the `ProductName` and `UnitPrice`. This can be accomplished by removing the other BoundFields and CheckBoxFields, by setting those other fields' `ReadOnly` property to `True`, or by some combination of the two. For this tutorial let's simply remove all GridView fields except the `ProductName` and `UnitPrice` BoundFields, after which the GridView's declarative markup will look like:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:BoundField DataField="ProductName"
          HeaderText="ProductName" SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
          SortExpression="UnitPrice" />
    </Columns>
</asp:GridView>
```

Even though the `UpdateProduct` overload expects three input parameters, we only have two BoundFields in our GridView. This is because the `productID` input parameter is a primary key value and passed in through the value of the `DataKeyNames` property for the edited row.

Our GridView, along with the `UpdateProduct` overload, allows a user to edit just the name and price of a product without losing any of the other product fields.

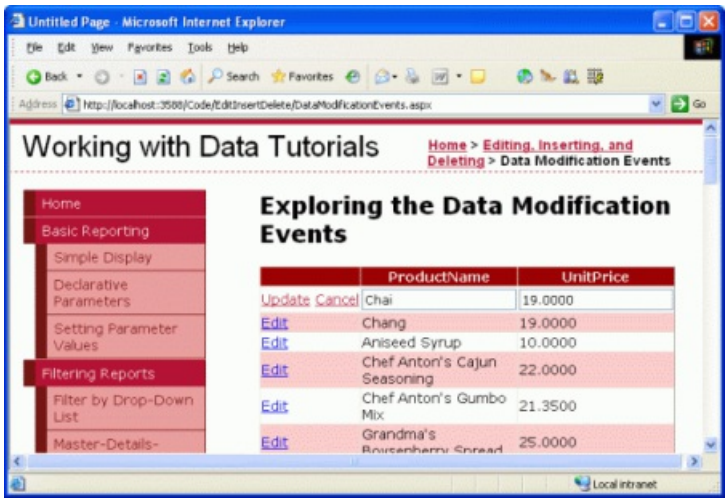


Figure 6: The Interface Allows Editing Just the Product's Name and Price (Click to view full-size image)

> **NOTE**
>
> As discussed in the previous tutorial, it is vitally important that the GridView s view state be enabled (the default behavior). If you set the GridView s `EnableViewState` property to `false`, you run the risk of having concurrent users unintentionally deleting or editing records. See WARNING: Concurrency Issue with ASP.NET 2.0 GridViews/DetailsView/FormViews that Support Editing and/or Deleting and Whose View State is Disabled for more information.

Improving the `UnitPrice` Formatting

While the GridView example shown in Figure 6 works, the `UnitPrice` field is not formatted at all, resulting in a price display that lacks any currency symbols and has four decimal places. To apply a currency formatting for the non-editable rows, simply set the `UnitPrice` BoundField's `DataFormatString` property to `{0:c}` and its `HtmlEncode` property to `False`.

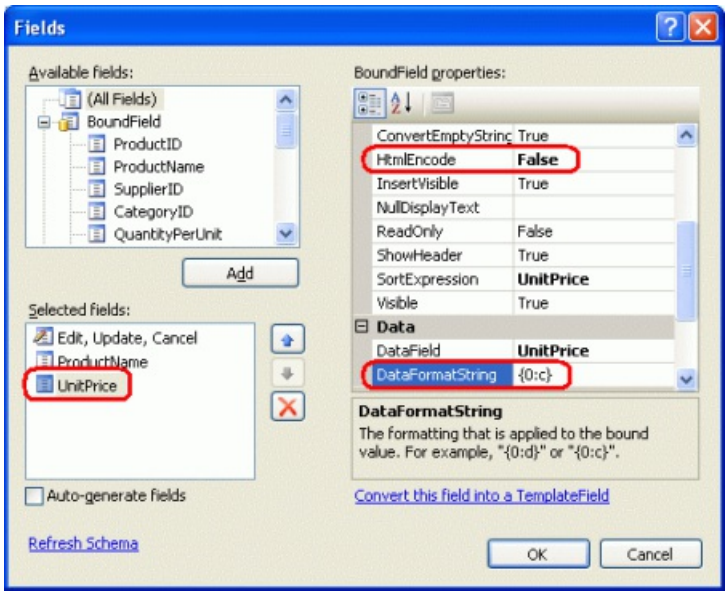


Figure 7: Set the `UnitPrice`'s `DataFormatString` and `HtmlEncode` Properties Accordingly (Click to view full-size image)

With this change, the non-editable rows format the price as a currency; the edited row, however, still displays the value without the currency symbol and with four decimal places.

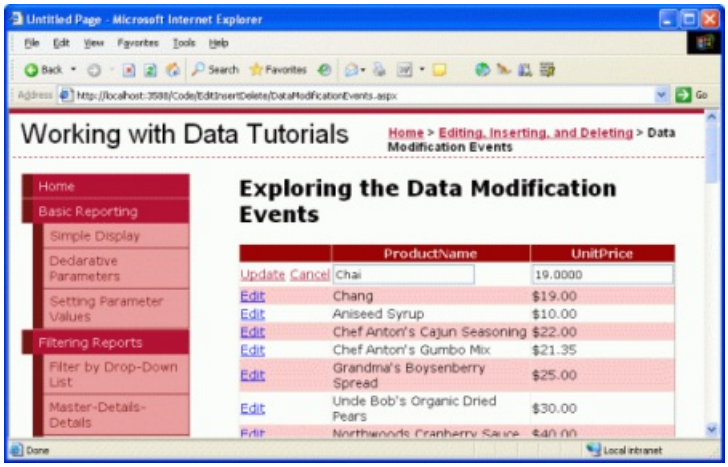


Figure 8: The Non-Editable Rows are Now Formatted as Currency Values (Click to view full-size image)

The formatting instructions specified in the `DataFormatString` property can be applied to the editing interface by setting the BoundField's `ApplyFormatInEditMode` property to `True` (the default is `False`). Take a moment to set this property to `True`.

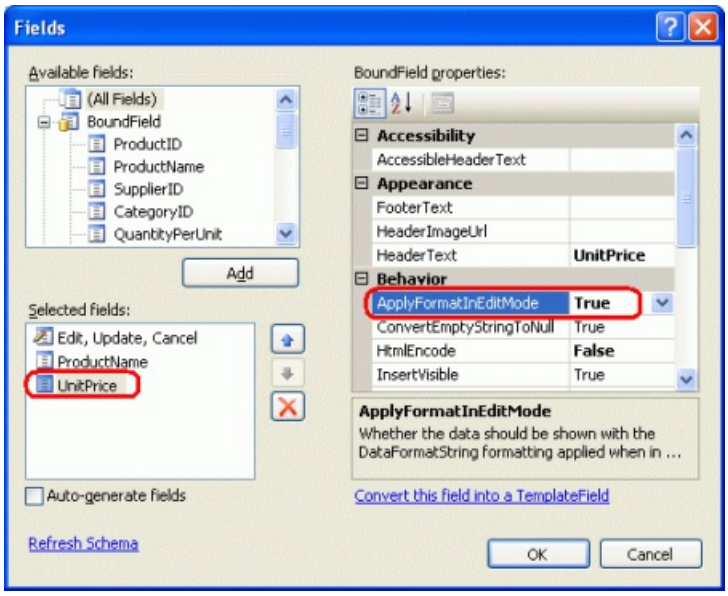


Figure 9: Set the `UnitPrice` BoundField's `ApplyFormatInEditMode` property to `True` (Click to view full-size image)

With this change, the value of the `UnitPrice` displayed in the edited row is also formatted as a currency.

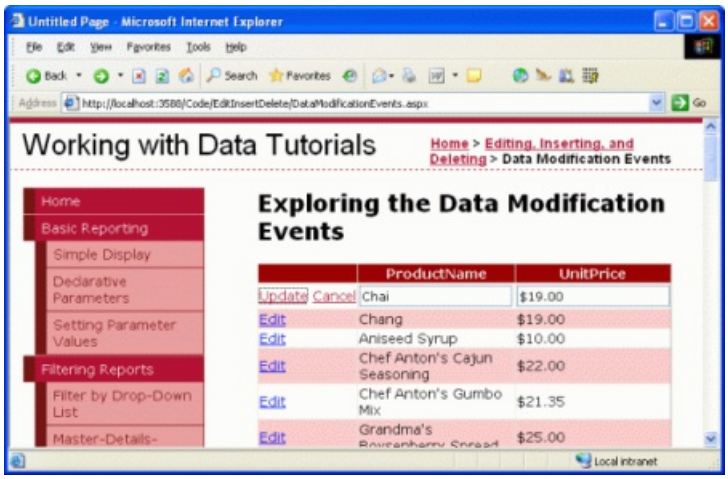


Figure 10: The Edited Row's `UnitPrice` Value is Now Formatted as a Currency (Click to view full-size image)

However, updating a product with the currency symbol in the textbox such as $19.00 throws a `FormatException`. When the GridView attempts to assign the user-supplied values to the ObjectDataSource's `UpdateParameters` collection it is unable to convert the `UnitPrice` string "$19.00" into the `Decimal` required by the parameter (see Figure 11). To remedy this we can create an event handler for the GridView's `RowUpdating` event and have it parse the user-supplied `UnitPrice` as a currency-formatted `Decimal`.

The GridView's `RowUpdating` event accepts as its second parameter an object of type GridViewUpdateEventArgs, which includes a `NewValues` dictionary as one of its properties that holds the user-supplied values ready to be assigned to the ObjectDataSource's `UpdateParameters` collection. We can overwrite the existing `UnitPrice` value in the `NewValues` collection with a decimal value parsed using the currency format with the following lines of code in the `RowUpdating` event handler:

```
Protected Sub GridView1_RowUpdating(sender As Object, e As GridViewUpdateEventArgs) _
    Handles GridView1.RowUpdating
    If e.NewValues("UnitPrice") IsNot Nothing Then
        e.NewValues("UnitPrice") = _
            Decimal.Parse(e.NewValues("UnitPrice").ToString(), _
                System.Globalization.NumberStyles.Currency)
    End If
End Sub
```

If the user has supplied a `UnitPrice` value (such as "$19.00"), this value is overwritten with the decimal value computed by Decimal.Parse, parsing the value as a currency. This will correctly parse the decimal in the event of any currency symbols, commas, decimal points, and so on, and uses the NumberStyles enumeration in the System.Globalization namespace.

Figure 11 shows both the problem caused by currency symbols in the user-supplied `UnitPrice`, along with how the GridView's `RowUpdating` event handler can be utilized to correctly parse such input.

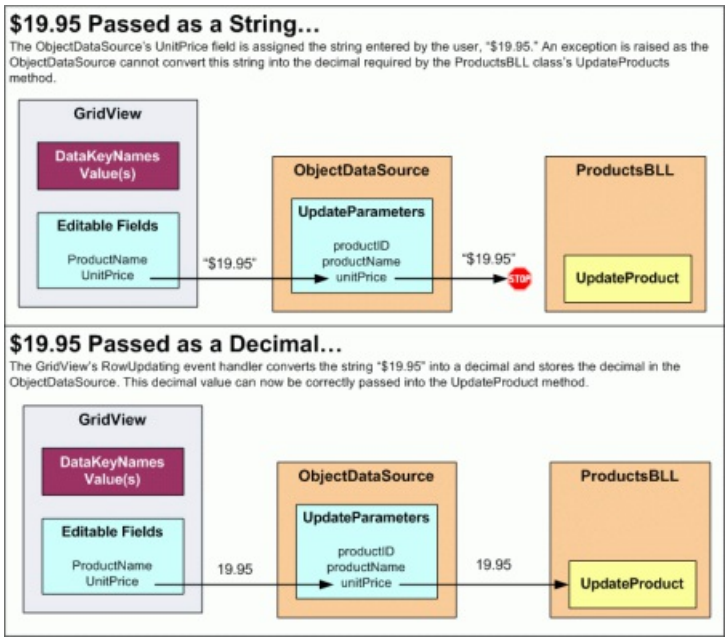


Figure 11: The Edited Row's `UnitPrice` Value is Now Formatted as a Currency (Click to view full-size image)

Step 2: Prohibiting `NULL UnitPrices`

While the database is configured to allow `NULL` values in the `Products` table's `UnitPrice` column, we may want to prevent users visiting this particular page from specifying a `NULL` `UnitPrice` value. That is, if a user fails to enter a `UnitPrice` value when editing a product row, rather than save the results to the database we want to display a message informing the user that, through this page, any edited products must have a price specified.

The `GridViewUpdateEventArgs` object passed into the GridView's `RowUpdating` event handler contains a `Cancel` property that, if set to `True`, terminates the updating process. Let's extend the `RowUpdating` event handler to set `e.Cancel` to `True` and display a message explaining why if the `UnitPrice` value in the `NewValues` collection has a value of `Nothing`.

Start by adding a Label Web control to the page named `MustProvideUnitPriceMessage`. This Label control will be displayed if the user fails to specify a `UnitPrice` value when updating a product. Set the Label's `Text` property to "You must provide a price for the product." I've also created a new CSS class in `Styles.css` named `Warning` with the following definition:

```
.Warning
{
    color: Red;
    font-style: italic;
    font-weight: bold;
    font-size: x-large;
}
```

Finally, set the Label's `CssClass` property to `Warning`. At this point the Designer should show the warning message in a red, bold, italic, extra large font size above the GridView, as shown in Figure 12.

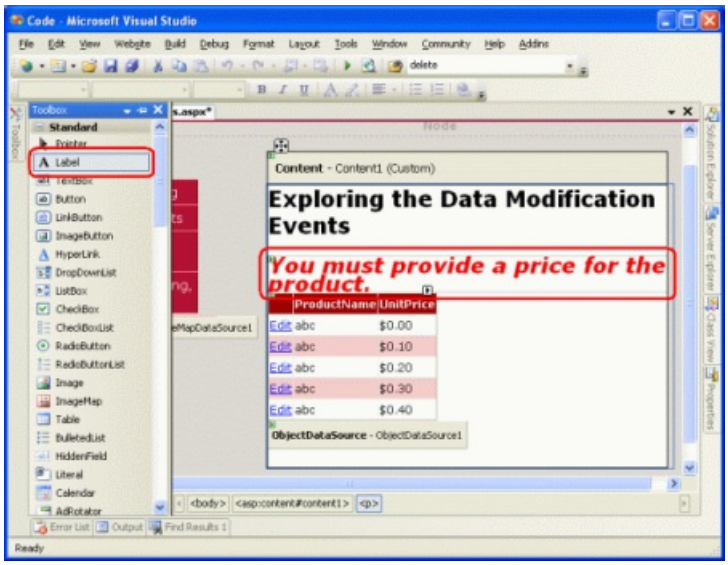


Figure 12: A Label Has Been Added Above the GridView (Click to view full-size image)

By default, this Label should be hidden, so set its `Visible` property to `False` in the `Page_Load` event handler:

```
Protected Sub Page_Load(sender As Object, e As System.EventArgs) Handles Me.Load
    MustProvideUnitPriceMessage.Visible = False
End Sub
```

If the user attempts to update a product without specifying the `UnitPrice`, we want to cancel the update and display the warning label. Augment the GridView's `RowUpdating` event handler as follows:

```
Protected Sub GridView1_RowUpdating(sender As Object, e As GridViewUpdateEventArgs) _
    Handles GridView1.RowUpdating

    If e.NewValues("UnitPrice") IsNot Nothing Then
        e.NewValues("UnitPrice") = _
            Decimal.Parse(e.NewValues("UnitPrice").ToString(), _
                System.Globalization.NumberStyles.Currency)
    Else
        MustProvideUnitPriceMessage.Visible = True

        e.Cancel = True
    End If
End Sub
```

If a user attempts to save a product without specifying a price, the update is cancelled and a helpful message is displayed. While the database (and business logic) allows for `NULL` `UnitPrice` s, this particular ASP.NET page does not.

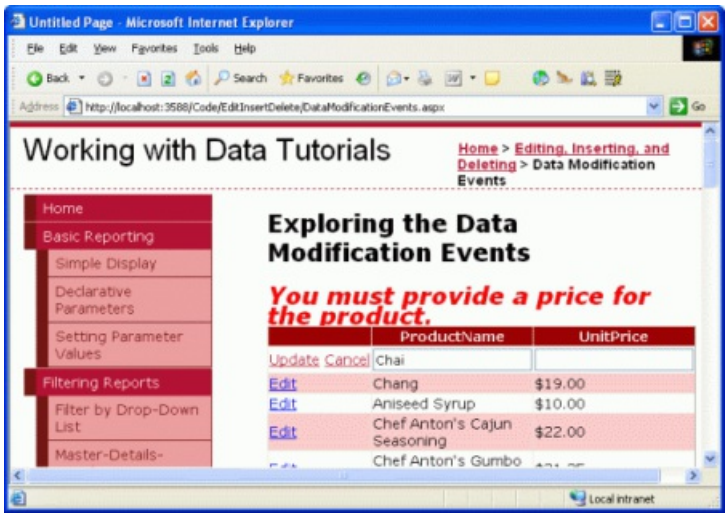


Figure 13: A User Cannot Leave `UnitPrice` Blank (Click to view full-size image)

So far we have seen how to use the GridView's `RowUpdating` event to programmatically alter the parameter values assigned to the ObjectDataSource's `UpdateParameters` collection as well as how to cancel the updating process altogether. These concepts carry over to the DetailsView and FormView controls and also apply to inserting and deleting.

These tasks can also be done at the ObjectDataSource level through event handlers for its `Inserting`, `Updating`, and `Deleting` events. These events fire before the associated method of the underlying object is invoked and provide a last-chance opportunity to modify the input parameters collection or cancel the operation outright. The event handlers for these three events are passed an object of type ObjectDataSourceMethodEventArgs that has two properties of interest:

- Cancel, which, if set to `True`, cancels the operation being performed
- InputParameters, which is the collection of `InsertParameters`, `UpdateParameters`, or `DeleteParameters`, depending on whether the event handler is for the `Inserting`, `Updating`, or `Deleting` event

To illustrate working with the parameter values at the ObjectDataSource level, let's include a DetailsView in our page that allows the users to add a new product. This DetailsView will be used to provide an interface for quickly adding a new product to the database. To keep a consistent user interface when adding a new product let's allow the user to only enter values for the `ProductName` and `UnitPrice` fields. By default, those values that aren't supplied in the DetailsView's inserting interface will be set to a `NULL` database value. However, we can use the ObjectDataSource's `Inserting` event to inject different default values, as we'll see shortly.

Step 3: Providing an Interface to Add New Products

Drag a DetailsView from the Toolbox onto the Designer above the GridView, clear out its `Height` and `Width` properties, and bind it to the ObjectDataSource already present on the page. This will add a BoundField or CheckBoxField for each of the product's fields. Since we want to use this DetailsView to add new products, we need to check the Enable Inserting option from the smart tag; however, there's no such option because the ObjectDataSource's `Insert()` method is not mapped to a method in the `ProductsBLL` class (recall that we set this mapping to (None) when configuring the data source see Figure 3).

To configure the ObjectDataSource, select the Configure Data Source link from its smart tag, launching the wizard. The first screen allows you to change the underlying object the ObjectDataSource is bound to; leave it set to `ProductsBLL`. The next screen lists the mappings from the ObjectDataSource's methods to the underlying object's. Even though we explicitly indicated that the `Insert()` and `Delete()` methods should not be mapped to any methods, if you go to the INSERT and DELETE tabs you'll see that a mapping is there. This is because the `ProductsBLL`'s `AddProduct` and `DeleteProduct` methods use the `DataObjectMethodAttribute` attribute to indicate that they are the default methods for `Insert()` and `Delete()`, respectively. Hence, the ObjectDataSource wizard

selects these each time you run the wizard unless there's some other value explicitly specified.

Leave the `Insert()` method pointing to the `AddProduct` method, but again set the DELETE tab's drop-down list to (None).

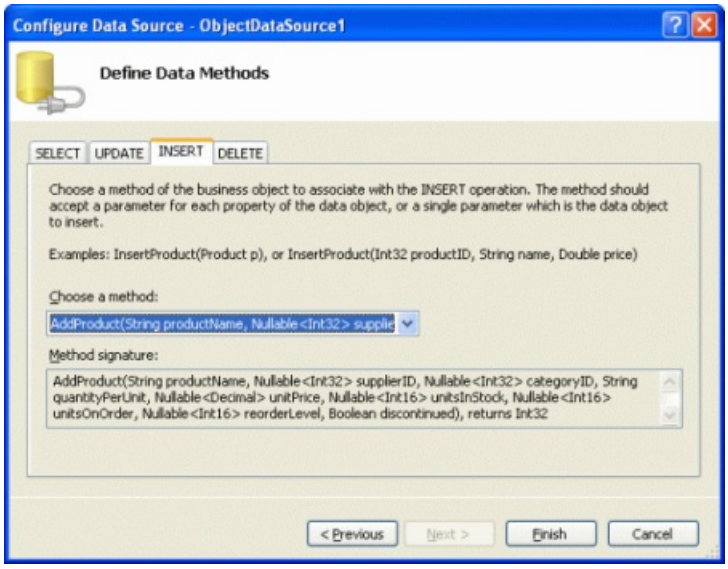


Figure 14: Set the INSERT Tab's Drop-Down List to the `AddProduct` Method (Click to view full-size image)

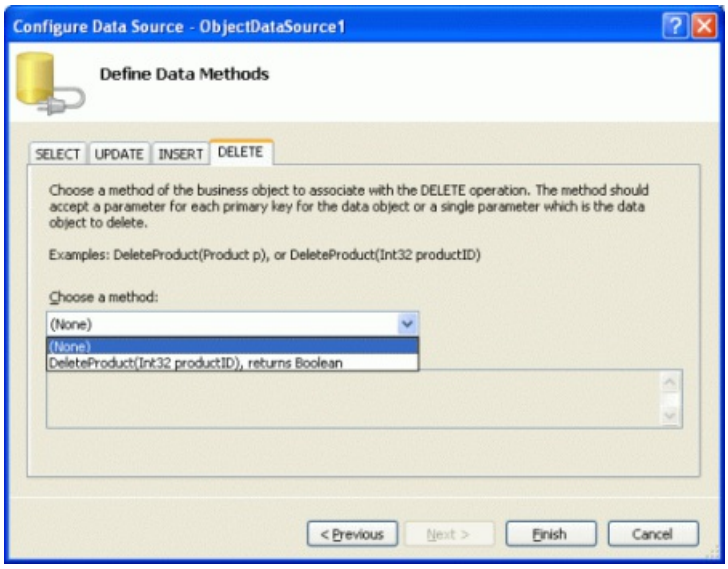


Figure 15: Set the DELETE Tab's Drop-Down List to (None) (Click to view full-size image)

After making these changes, the ObjectDataSource's declarative syntax will be expanded to include an `InsertParameters` collection, as shown below:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    SelectMethod="GetProducts" TypeName="ProductsBLL"
    UpdateMethod="UpdateProduct" OnUpdating="ObjectDataSource1_Updating"
    InsertMethod="AddProduct" OldValuesParameterFormatString="original_{0}">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
    <InsertParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="unitsInStock" Type="Int16" />
        <asp:Parameter Name="unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="reorderLevel" Type="Int16" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
    </InsertParameters>
</asp:ObjectDataSource>
```

Rerunning the wizard added back the `OldValuesParameterFormatString` property. Take a moment to clear this property by setting it to the default value (`{0}`) or removing it altogether from the declarative syntax.

With the ObjectDataSource providing inserting capabilities, the DetailsView's smart tag will now include the Enable Inserting checkbox; return to the Designer and check this option. Next, pare down the DetailsView so that it only has two BoundFields - `ProductName` and `UnitPrice` - and the CommandField. At this point the DetailsView's declarative syntax should look like:

```
<asp:DetailsView ID="DetailsView1" runat="server" AutoGenerateRows="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1">
    <Fields>
        <asp:BoundField DataField="ProductName"
          HeaderText="ProductName" SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
          SortExpression="UnitPrice" />
        <asp:CommandField ShowInsertButton="True" />
    </Fields>
</asp:DetailsView>
```

Figure 16 shows this page when viewed through a browser at this point. As you can see, the DetailsView lists the name and price of the first product (Chai). What we want, however, is an inserting interface that provides a means for the user to quickly add a new product to the database.

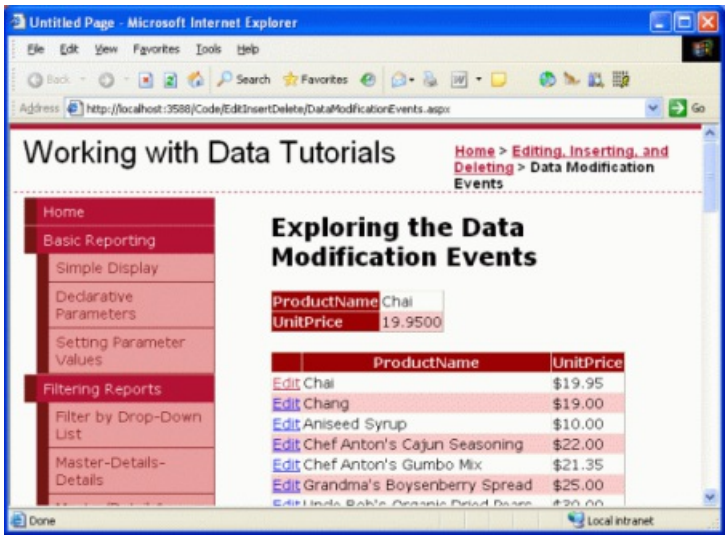


Figure 16: The DetailsView is Currently Rendered in Read-Only Mode (Click to view full-size image)

In order to show the DetailsView in its inserting mode we need to set the `DefaultMode` property to `Inserting`. This renders the DetailsView in insert mode when first visited and keeps it there after inserting a new record. As Figure 17 shows, such a DetailsView provides a quick interface for adding a new record.

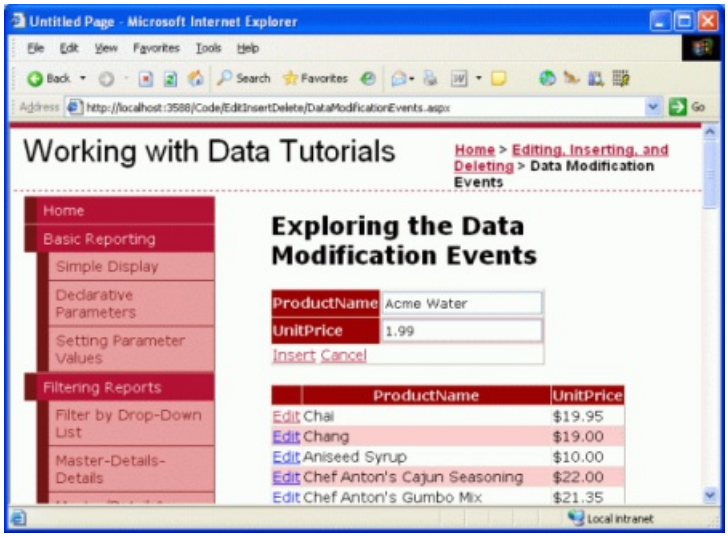


Figure 17: The DetailsView Provides an Interface for Quickly Adding a New Product (Click to view full-size image)

When the user enters a product name and price (such as "Acme Water" and 1.99, as in Figure 17) and clicks Insert, a postback ensues and the inserting workflow commences, culminating in a new product record being added to the database. The DetailsView maintains its inserting interface and the GridView is automatically rebound to its data source in order to include the new product, as shown in Figure 18.

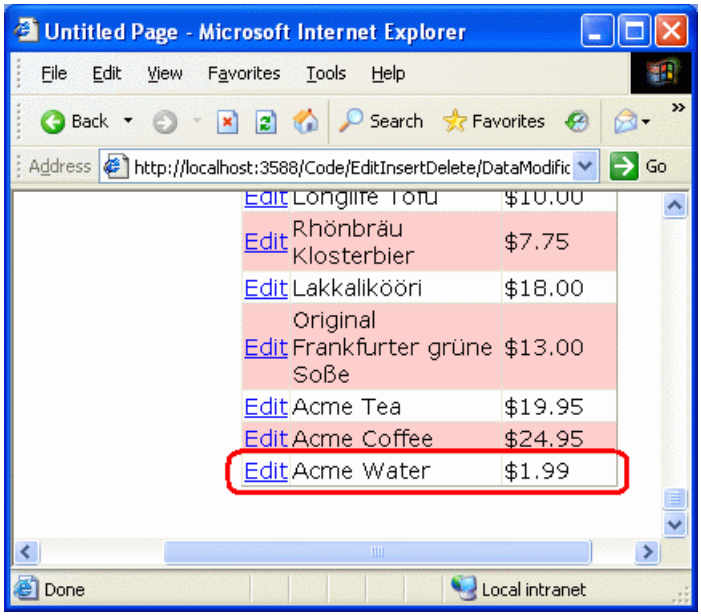


Figure 18: The Product "Acme Water" Has Been Added to the Database

While the GridView in Figure 18 doesn't show it, the product fields lacking from the DetailsView interface `CategoryID`, `SupplierID`, `QuantityPerUnit`, and so on are assigned `NULL` database values. You can see this by performing the following steps:

1. Go to the Server Explorer in Visual Studio
2. Expanding the `NORTHWND.MDF` database node
3. Right-click on the `Products` database table node
4. Select Show Table Data

This will list all of the records in the `Products` table. As Figure 19 shows, all of our new product's columns other than `ProductID`, `ProductName`, and `UnitPrice` have `NULL` values.

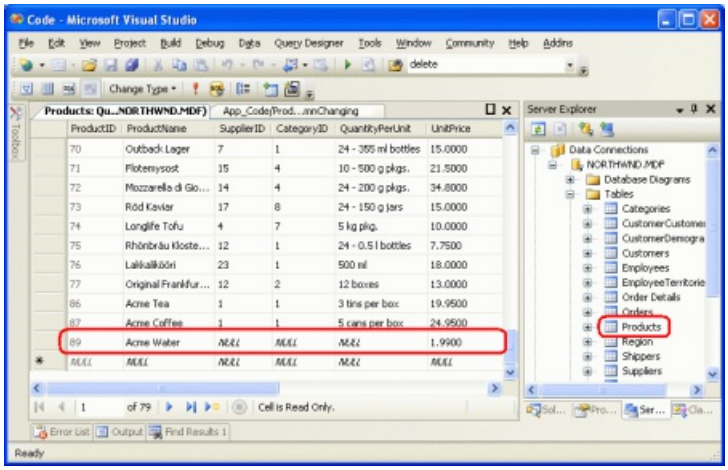


Figure 19: The Product Fields Not Provided in the DetailsView are Assigned `NULL` Values (Click to view full-size image)

We may want to provide a default value other than `NULL` for one or more of these column values, either because `NULL` isn't the best default option or because the database column itself doesn't allow `NULL` s. To accomplish this we can programmatically set the values of the parameters of the DetailsView's `InputParameters` collection. This assignment can be done either in the event handler for the DetailsView's `ItemInserting` event or the ObjectDataSource's `Inserting` event. Since we've already looked at using the pre- and post-level events at the data Web control level, let's explore using the ObjectDataSource's events this time.

Step 4: Assigning Values to the `CategoryID` and `SupplierID` Parameters

For this tutorial let's imagine that for our application when adding a new product through this interface it should be assigned a `CategoryID` and `SupplierID` value of 1. As mentioned earlier, the ObjectDataSource has a pair of pre- and post-level events that fire during the data modification process. When its `Insert()` method is invoked, the ObjectDataSource first raises its `Inserting` event, then calls the method that its `Insert()` method has been mapped to, and finally raises the `Inserted` event. The `Inserting` event handler affords us one last opportunity to tweak the input parameters or cancel the operation outright.

> **NOTE**
>
> In a real-world application you would likely want to either let the user specify the category and supplier or would pick this value for them based on some criteria or business logic (rather than blindly selecting an ID of 1). Regardless, the example illustrates how to programmatically set the value of an input parameter from the ObjectDataSource's pre-level event.

Take a moment to create an event handler for the ObjectDataSource's `Inserting` event. Notice that the event handler's second input parameter is an object of type `ObjectDataSourceMethodEventArgs`, which has a property to access the parameters collection (`InputParameters`) and a property to cancel the operation (`Cancel`).

```
Protected Sub ObjectDataSource1_Inserting _
    (sender As Object, e As ObjectDataSourceMethodEventArgs) _
    Handles ObjectDataSource1.Inserting

End Sub
```

At this point, the `InputParameters` property contains the ObjectDataSource's `InsertParameters` collection with the values assigned from the DetailsView. To change the value of one of these parameters, simply use:

`e.InputParameters("paramName") = value`. Therefore, to set the `CategoryID` and `SupplierID` to values of 1, adjust the `Inserting` event handler to look like the following:

```
Protected Sub ObjectDataSource1_Inserting _
    (sender As Object, e As ObjectDataSourceMethodEventArgs) _
    Handles ObjectDataSource1.Inserting

    e.InputParameters("CategoryID") = 1
    e.InputParameters("SupplierID") = 1
End Sub
```

This time when adding a new product (such as Acme Soda), the `CategoryID` and `SupplierID` columns of the new product are set to 1 (see Figure 20).

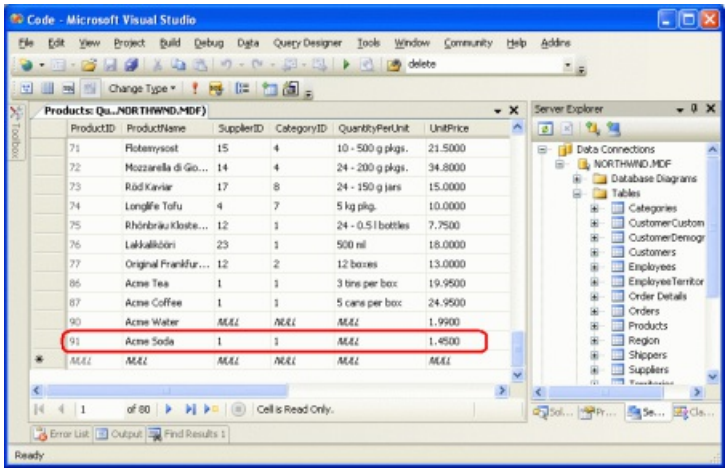


Figure 20: New Products Now Have Their `CategoryID` and `SupplierID` Values Set to 1 (Click to view full-size image)

Summary

During the editing, inserting, and deleting process, both the data Web control and the ObjectDataSource proceed through a number of pre- and post-level events. In this tutorial we examined the pre-level events and saw how to use these to customize the input parameters or cancel the data modification operation altogether both from the data Web control and ObjectDataSource's events. In the next tutorial we'll look at creating and using event handlers for the post-level events.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Jackie Goor and Liz Shulok. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Handling BLL- and DAL-Level Exceptions in an ASP.NET Page (VB)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we will see how to display a friendly, informative error message should an exception occur during an insert, update, or delete operation of an ASP.NET data Web control.

Introduction

Working with data from an ASP.NET web application using a tiered application architecture involves the following three general steps:

1. Determine what method of the Business Logic Layer needs to be invoked and what parameter values to pass it. The parameter values can be hard coded, programmatically-assigned, or inputs entered by the user.
2. Invoke the method.
3. Process the results. When calling a BLL method that returns data, this may involve binding the data to a data Web control. For BLL methods that modify data, this may include performing some action based on a return value or gracefully handling any exception that arose in Step 2.

As we saw in the previous tutorial, both the ObjectDataSource and the data Web controls provide extensibility points for Steps 1 and 3. The GridView, for example, fires its `RowUpdating` event prior to assigning its field values to its ObjectDataSource's `UpdateParameters` collection; its `RowUpdated` event is raised after the ObjectDataSource has completed the operation.

We've already examined the events that fire during Step 1 and have seen how they can be used to customize the input parameters or cancel the operation. In this tutorial we'll turn our attention to the events that fire after the operation has completed. With these post-level event handlers we can, among other things, determine if an exception occurred during the operation and handle it gracefully, displaying a friendly, informative error message on the screen rather than defaulting to the standard ASP.NET exception page.

To illustrate working with these post-level events, let's create a page that lists the products in an editable GridView. When updating a product, if an exception is raised our ASP.NET page will display a short message above the GridView explaining that a problem has occurred. Let's get started!

Step 1: Creating an Editable GridView of Products

In the previous tutorial we created an editable GridView with just two fields, `ProductName` and `UnitPrice`. This required creating an additional overload for the `ProductsBLL` class's `UpdateProduct` method, one that only accepted three input parameters (the product's name, unit price, and ID) as opposed a parameter for each product field. For this tutorial, let's practice this technique again, creating an editable GridView that displays the product's name, quantity per unit, unit price, and units in stock, but only allows the name, unit price, and units in stock to be edited.

To accommodate this scenario we'll need another overload of the `UpdateProduct` method, one that accepts four parameters: the product's name, unit price, units in stock, and ID. Add the following method to the `ProductsBLL` class:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Update, True)> _
Public Function UpdateProduct _
    (ByVal productName As String, ByVal unitPrice As Nullable(Of Decimal), _
ByVal unitsInStock As Nullable(Of Short), ByVal productID As Integer) As Boolean
    Dim products As Northwind.ProductsDataTable = _
        Adapter.GetProductByProductID(productID)
    If products.Count = 0 Then
        Return False
    End If
    Dim product As Northwind.ProductsRow = products(0)
    product.ProductName = productName
    If Not unitPrice.HasValue Then
        product.SetUnitPriceNull()
    Else
        product.UnitPrice = unitPrice.Value
    End If
    If Not unitsInStock.HasValue Then
        product.SetUnitsInStockNull()
    Else
        product.UnitsInStock = unitsInStock.Value
    End If
    Dim rowsAffected As Integer = Adapter.Update(product)
    Return rowsAffected = 1
End Function
```

With this method complete, we're ready to create the ASP.NET page that allows for editing these four particular product fields. Open the `ErrorHandling.aspx` page in the `EditInsertDelete` folder and add a GridView to the page through the Designer. Bind the GridView to a new ObjectDataSource, mapping the `Select()` method to the `ProductsBLL` class's `GetProducts()` method and the `Update()` method to the `UpdateProduct` overload just created.

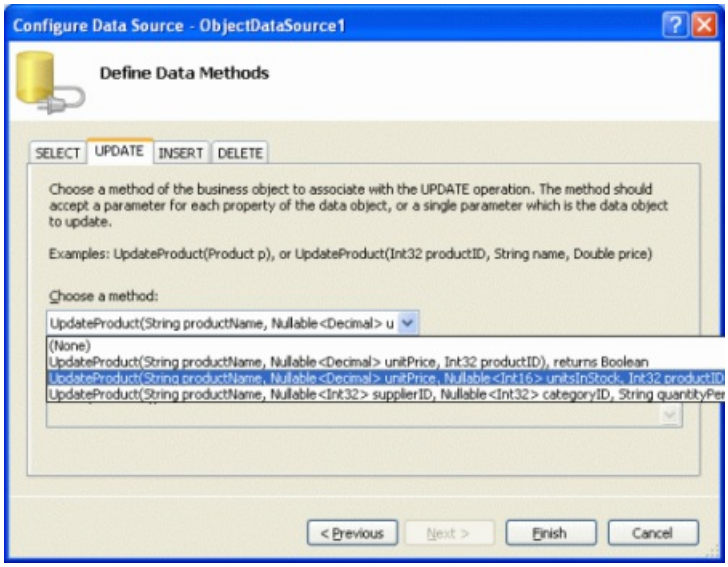


Figure 1: Use the `UpdateProduct` Method Overload That Accepts Four Input Parameters (Click to view full-size image)

This will create an ObjectDataSource with an `UpdateParameters` collection with four parameters and a GridView with a field for each of the product fields. The ObjectDataSource's declarative markup assigns the `OldValuesParameterFormatString` property the value `original_{0}`, which will cause an exception since our BLL class's don't expect an input parameter named `original_productID` to be passed in. Don't forget to remove this setting altogether from the declarative syntax (or set it to the default value, `{0}`).

Next, pare down the GridView to include only the `ProductName`, `QuantityPerUnit`, `UnitPrice`, and `UnitsInStock` BoundFields. Also feel free to apply any field-level formatting you deem necessary (such as changing the

`HeaderText` properties).

In the previous tutorial we looked at how to format the `UnitPrice` BoundField as a currency both in read-only mode and edit mode. Let's do the same here. Recall that this required setting the BoundField's `DataFormatString` property to `{0:c}`, its `HtmlEncode` property to `false`, and its `ApplyFormatInEditMode` to `true`, as shown in Figure 2.

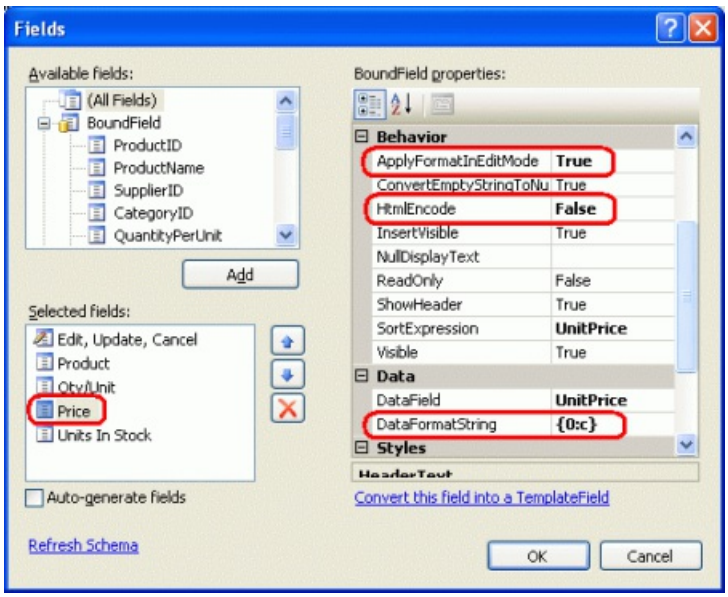


Figure 2: Configure the `UnitPrice` BoundField to Display as a Currency (Click to view full-size image)

Formatting the `UnitPrice` as a currency in the editing interface requires creating an event handler for the GridView's `RowUpdating` event that parses the currency-formatted string into a `decimal` value. Recall that the `RowUpdating` event handler from the last tutorial also checked to ensure that the user provided a `UnitPrice` value. However, for this tutorial let's allow the user to omit the price.

```
Protected Sub GridView1_RowUpdating(ByVal sender As Object, _
    ByVal e As System.Web.UI.WebControls.GridViewUpdateEventArgs) _
    Handles GridView1.RowUpdating
    If e.NewValues("UnitPrice") IsNot Nothing Then
        e.NewValues("UnitPrice") = _
            Decimal.Parse(e.NewValues("UnitPrice").ToString(), _
            System.Globalization.NumberStyles.Currency)
    End If
```

Our GridView includes a `QuantityPerUnit` BoundField, but this BoundField should be only for display purposes and should not be editable by the user. To arrange this, simply set the BoundFields' `ReadOnly` property to `true`.

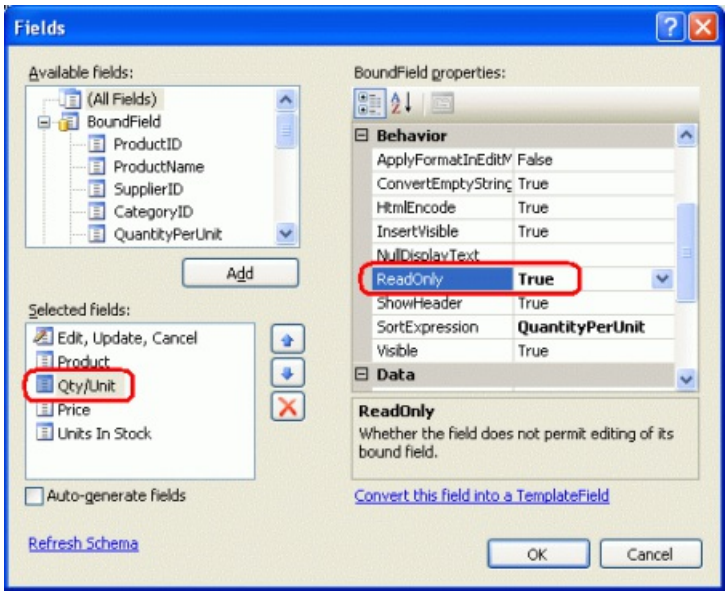


Figure 3: Make the `QuantityPerUnit` BoundField Read-Only (Click to view full-size image)

Finally, check the Enable Editing checkbox from the GridView's smart tag. After completing these steps the `ErrorHandling.aspx` page's Designer should look similar to Figure 4.

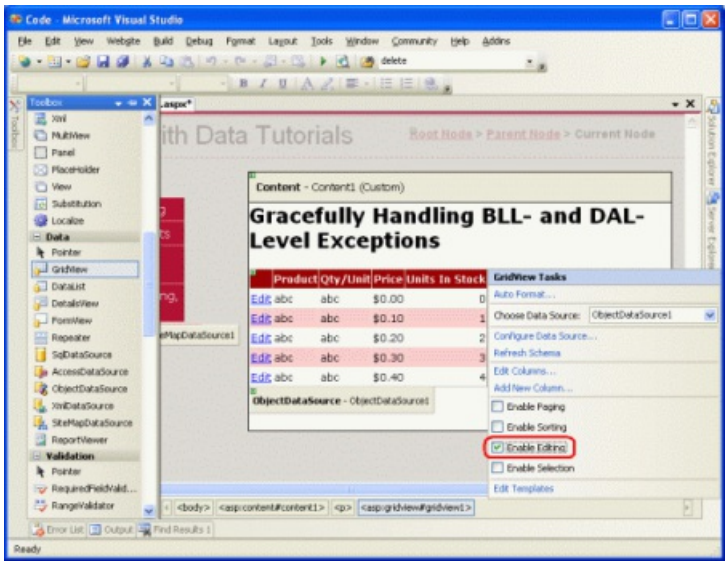


Figure 4: Remove All But the Needed BoundFields and Check the Enable Editing Checkbox (Click to view full-size image)

At this point we have a list of all of the products' `ProductName`, `QuantityPerUnit`, `UnitPrice`, and `UnitsInStock` fields; however, only the `ProductName`, `UnitPrice`, and `UnitsInStock` fields can be edited.

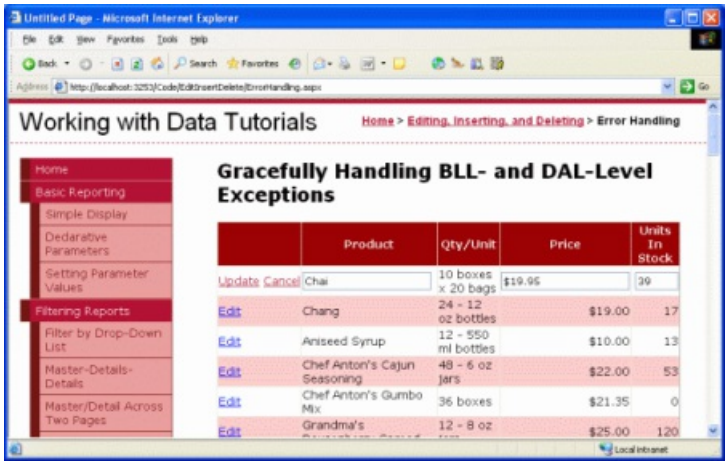


Figure 5: Users Can Now Easily Edit Products' Names, Prices, and Units In Stock Fields (Click to view full-size image)

Step 2: Gracefully Handling DAL-Level Exceptions

While our editable GridView works wonderfully when users enter legal values for the edited product's name, price, and units in stock, entering illegal values results in an exception. For example, omitting the `ProductName` value causes a NoNullAllowedException to be thrown since the `ProductName` property in the `ProductsRow` class has its `AllowDBNull` property set to `false`; if the database is down, a `SqlException` will be thrown by the TableAdapter when attempting to connect to the database. Without taking any action, these exceptions bubble up from the Data Access Layer to the Business Logic Layer, then to the ASP.NET page, and finally to the ASP.NET runtime.

Depending on how your web application is configured and whether or not you're visiting the application from `localhost`, an unhandled exception can result in either a generic server-error page, a detailed error report, or a user-friendly web page. See Web Application Error Handling in ASP.NET and the customErrors Element for more information on how the ASP.NET runtime responds to an uncaught exception.

Figure 6 shows the screen encountered when attempting to update a product without specifying the `ProductName` value. This is the default detailed error report displayed when coming through `localhost`.

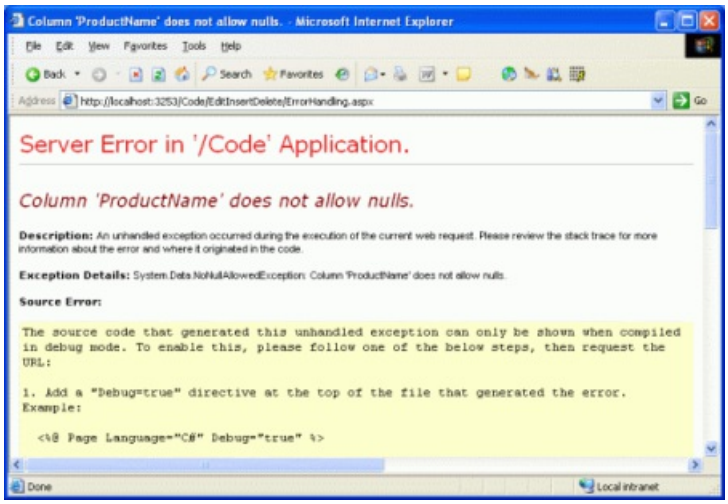


Figure 6: Omitting the Product's Name Will Display Exception Details (Click to view full-size image)

While such exception details are helpful when testing an application, presenting an end user with such a screen in the face of an exception is less than ideal. An end user likely doesn't know what a `NoNullAllowedException` is or why it was caused. A better approach is to present the user with a more user-friendly message explaining that there were problems attempting to update the product.

If an exception occurs when performing the operation, the post-level events in both the ObjectDataSource and the data Web control provide a means to detect it and cancel the exception from bubbling up to the ASP.NET runtime. For our example, let's create an event handler for the GridView's `RowUpdated` event that determines if an exception has fired and, if so, displays the exception details in a Label Web control.

Start by adding a Label to the ASP.NET page, setting its `ID` property to `ExceptionDetails` and clearing out its `Text` property. In order to draw the user's eye to this message, set its `CssClass` property to `Warning`, which is a CSS class we added to the `Styles.css` file in the previous tutorial. Recall that this CSS class causes the Label's text to be displayed in a red, italic, bold, extra large font.

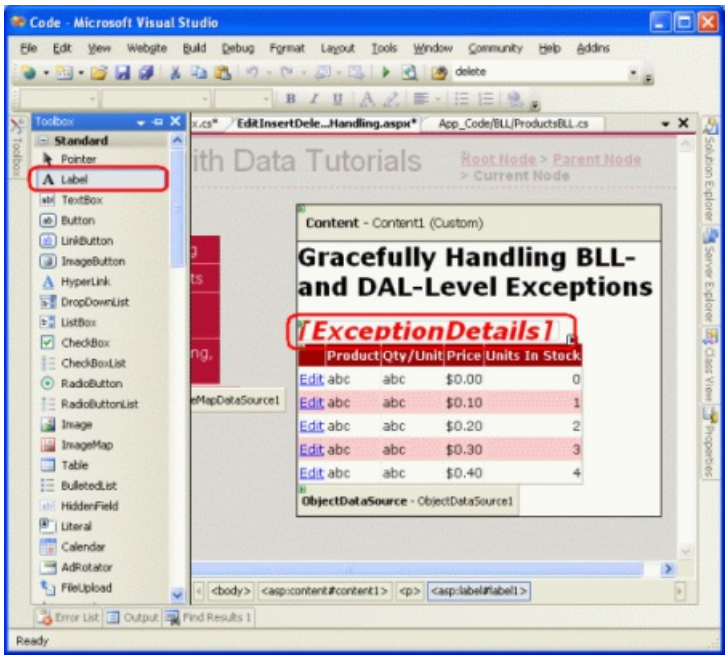


Figure 7: Add a Label Web Control to the Page (Click to view full-size image)

Since we want this Label Web control to be visible only immediately after an exception has occurred, set its `Visible` property to false in the `Page_Load` event handler:

```
Protected Sub Page_Load(sender As Object, e As System.EventArgs) Handles Me.Load
    ExceptionDetails.Visible = False
End Sub
```

With this code, on the first page visit and subsequent postbacks the `ExceptionDetails` control will have its `Visible` property set to `false`. In the face of a DAL- or BLL-level exception, which we can detect in the GridView's `RowUpdated` event handler, we will set the `ExceptionDetails` control's `Visible` property to true. Since Web control event handlers occur after the `Page_Load` event handler in the page lifecycle, the Label will be shown. However, on the next postback, the `Page_Load` event handler will revert the `Visible` property back to `false`, hiding it from view again.

> **NOTE**
>
> Alternatively, we could remove the necessity for setting the `ExceptionDetails` control's `Visible` property in `Page_Load` by assigning its `Visible` property `false` in the declarative syntax and disabling its view state (setting its `EnableViewState` property to `false`). We'll use this alternative approach in a future tutorial.

With the Label control added, our next step is to create the event handler for the GridView's `RowUpdated` event. Select the GridView in the Designer, go to the Properties window, and click the lightning bolt icon, listing the GridView's events. There should already be an entry there for the GridView's `RowUpdating` event, as we created an event handler for this event earlier in this tutorial. Create an event handler for the `RowUpdated` event as well.

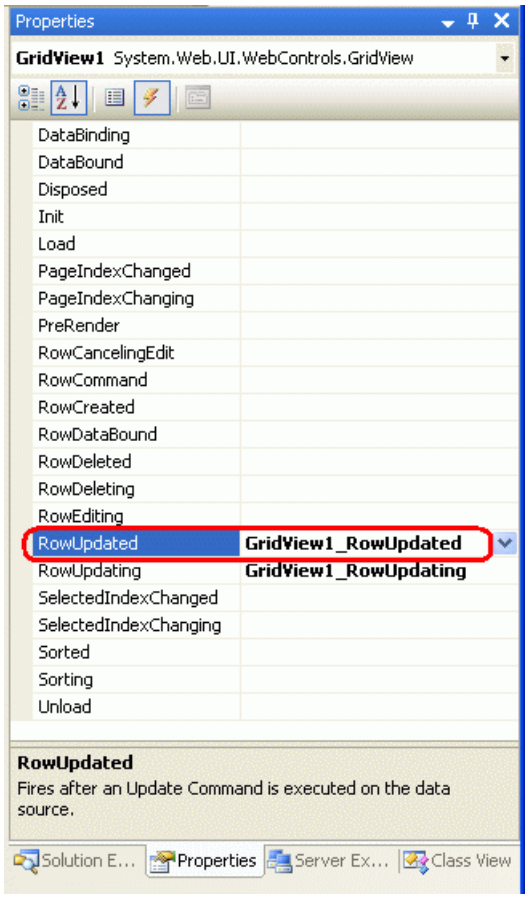


Figure 8: Create an Event Handler for the GridView's `RowUpdated` Event

> **NOTE**
>
> You can also create the event handler through the drop-down lists at the top of the code-behind class file. Select the GridView from the drop-down list on the left and the `RowUpdated` event from the one on the right.

Creating this event handler will add the following code to the ASP.NET page's code-behind class:

```
Protected Sub GridView1_RowUpdated(ByVal sender As Object, _
    ByVal e As System.Web.UI.WebControls.GridViewUpdatedEventArgs) _
    Handles GridView1.RowUpdated
End Sub
```

This event handler's second input parameter is an object of type GridViewUpdatedEventArgs, which has three properties of interest for handling exceptions:

- `Exception` a reference to the thrown exception; if no exception has been thrown, this property will have a value of `null`
- `ExceptionHandled` a Boolean value that indicates whether or not the exception was handled in the `RowUpdated` event handler; if `false` (the default), the exception is re-thrown, percolating up to the ASP.NET runtime
- `KeepInEditMode` if set to `true` the edited GridView row remains in edit mode; if `false` (the default), the GridView row reverts back to its read-only mode

Our code, then, should check to see if `Exception` is not `null`, meaning that an exception was raised while performing the operation. If this is the case, we want to:

- Display a user-friendly message in the `ExceptionDetails` Label
- Indicate that the exception was handled
- Keep the GridView row in edit mode

This following code accomplishes these objectives:

```
Protected Sub GridView1_RowUpdated(ByVal sender As Object, _
    ByVal e As System.Web.UI.WebControls.GridViewUpdatedEventArgs) _
    Handles GridView1.RowUpdated
    If e.Exception IsNot Nothing Then
        ExceptionDetails.Visible = True
        ExceptionDetails.Text = "There was a problem updating the product. "
        If e.Exception.InnerException IsNot Nothing Then
            Dim inner As Exception = e.Exception.InnerException
            If TypeOf inner Is System.Data.Common.DbException Then
                ExceptionDetails.Text &= _
                "Our database is currently experiencing problems." & _
                "Please try again later."
            ElseIf TypeOf inner _
             Is System.Data.NoNullAllowedException Then
                ExceptionDetails.Text += _
                    "There are one or more required fields that are missing."
            ElseIf TypeOf inner Is ArgumentException Then
                Dim paramName As String = CType(inner, ArgumentException).ParamName
                ExceptionDetails.Text &= _
                    String.Concat("The ", paramName, " value is illegal.")
            ElseIf TypeOf inner Is ApplicationException Then
                ExceptionDetails.Text += inner.Message
            End If
        End If
        e.ExceptionHandled = True
        e.KeepInEditMode = True
    End If
End Sub
```

This event handler begins by checking to see if `e.Exception` is `null`. If it's not, the `ExceptionDetails` Label's `Visible` property is set to `true` and its `Text` property to "There was a problem updating the product." The details of the actual exception that was thrown reside in the `e.Exception` object's `InnerException` property. This inner exception is examined and, if it is of a particular type, an additional, helpful message is appended to the `ExceptionDetails` Label's `Text` property. Lastly, the `ExceptionHandled` and `KeepInEditMode` properties are both set to `true`.

Figure 9 shows a screen shot of this page when omitting the name of the product; Figure 10 shows the results when entering an illegal `UnitPrice` value (-50).

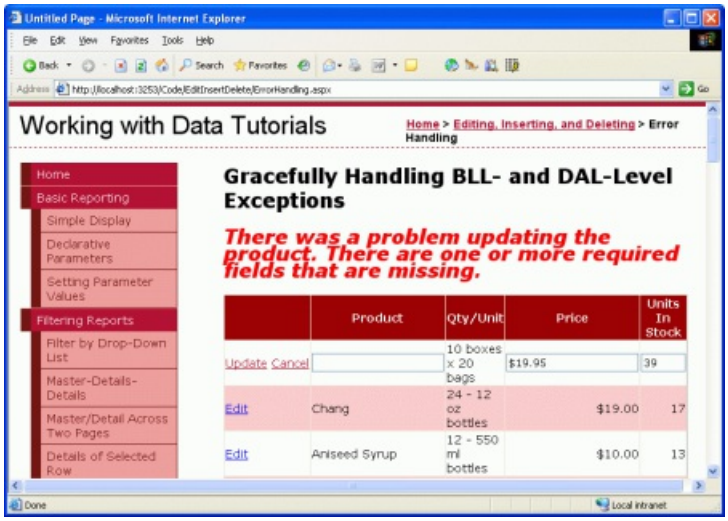


Figure 9: The `ProductName` BoundField Must Contain a Value (Click to view full-size image)

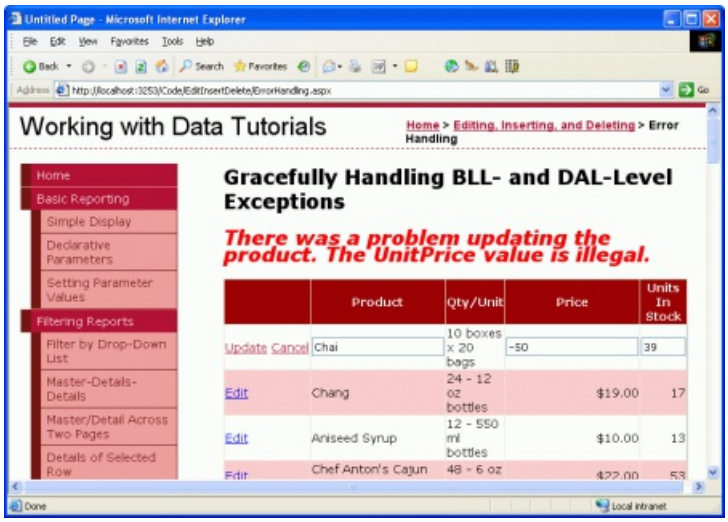


Figure 10: Negative `UnitPrice` Values are Not Allowed (Click to view full-size image)

By setting the `e.ExceptionHandled` property to `true`, the `RowUpdated` event handler has indicated that it has handled the exception. Therefore, the exception won't propagate up to the ASP.NET runtime.

> **NOTE**
>
> Figures 9 and 10 show a graceful way to handle exceptions raised due to invalid user input. Ideally, though, such invalid input will never reach the Business Logic Layer in the first place, as the ASP.NET page should ensure that the user's inputs are valid before invoking the `ProductsBLL` class's `UpdateProduct` method. In our next tutorial we'll see how to add validation controls to the editing and inserting interfaces to ensure that the data submitted to the Business Logic Layer conforms to the business rules. The validation controls not only prevent the invocation of the `UpdateProduct` method until the user-supplied data is valid, but also provide a more informative user experience for identifying data entry problems.

Step 3: Gracefully Handling BLL-Level Exceptions

When inserting, updating, or deleting data, the Data Access Layer may throw an exception in the face of a data-related error. The database may be offline, a required database table column might not have had a value specified, or a table-level constraint may have been violated. In addition to strictly data-related exceptions, the Business Logic Layer can use exceptions to indicate when business rules have been violated. In the Creating a Business Logic Layer tutorial, for example, we added a business rule check to the original `UpdateProduct` overload. Specifically, if the user was marking a product as discontinued, we required that the product not be the only one provided by its supplier. If this condition was violated, an `ApplicationException` was thrown.

For the `UpdateProduct` overload created in this tutorial, let's add a business rule that prohibits the `UnitPrice` field from being set to a new value that's more than twice the original `UnitPrice` value. To accomplish this, adjust the `UpdateProduct` overload so that it performs this check and throws an `ApplicationException` if the rule is violated. The updated method follows:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Update, True)> _
    Public Function UpdateProduct(ByVal productName As String, _
    ByVal unitPrice As Nullable(Of Decimal), ByVal unitsInStock As Nullable(Of Short), _
    ByVal productID As Integer) As Boolean
    Dim products As Northwind.ProductsDataTable = Adapter.GetProductByProductID(productID)
    If products.Count = 0 Then
        Return False
    End If
    Dim product As Northwind.ProductsRow = products(0)
    If unitPrice.HasValue AndAlso Not product.IsUnitPriceNull() Then
        If unitPrice > product.UnitPrice * 2 Then
            Throw New ApplicationException( _
                "When updating a product price," & _
                " the new price cannot exceed twice the original price.")
        End If
    End If
    product.ProductName = productName
    If Not unitPrice.HasValue Then
        product.SetUnitPriceNull()
    Else
        product.UnitPrice = unitPrice.Value
    End If
    If Not unitsInStock.HasValue Then
        product.SetUnitsInStockNull()
    Else
        product.UnitsInStock = unitsInStock.Value
    End If
    Dim rowsAffected As Integer = Adapter.Update(product)
    Return rowsAffected = 1
End Function
```

With this change, any price update that is more than twice the existing price will cause an `ApplicationException` to be thrown. Just like the exception raised from the DAL, this BLL-raised `ApplicationException` can be detected and handled in the GridView's `RowUpdated` event handler. In fact, the `RowUpdated` event handler's code, as written, will correctly detect this exception and display the `ApplicationException`'s `Message` property value. Figure 11 shows a screen shot when a user attempts to update the price of Chai to $50.00, which is more than double its current price of $19.95.

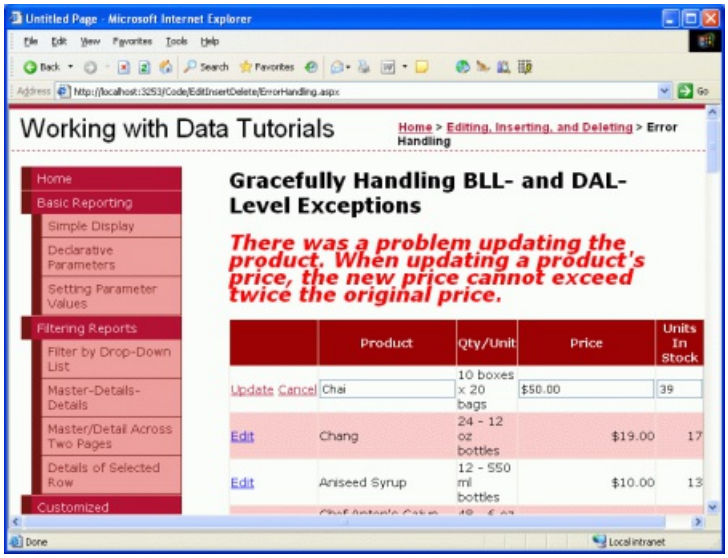


Figure 11: The Business Rules Disallow Price Increases That More Than Double a Product's Price (Click to view full-size image)

NOTE

Ideally our business logic rules would be refactored out of the `UpdateProduct` method overloads and into a common method. This is left as an exercise for the reader.

Summary

During inserting, updating, and deleting operations, both the data Web control and the ObjectDataSource involved fire pre- and post-level events that bookend the actual operation. As we saw in this tutorial and the preceding one, when working with an editable GridView the GridView's `RowUpdating` event fires, followed by the ObjectDataSource's `Updating` event, at which point the update command is made to the ObjectDataSource's underlying object. After the operation has completed, the ObjectDataSource's `Updated` event fires, followed by the GridView's `RowUpdated` event.

We can create event handlers for the pre-level events in order to customize the input parameters or for the post-level events in order to inspect and respond to the operation's results. Post-level event handlers are most commonly used to detect whether an exception occurred during the operation. In the face of an exception, these post-level event handlers can optionally handle the exception on their own. In this tutorial we saw how to handle such an exception by displaying a friendly error message.

In the next tutorial we'll see how to lessen the likelihood of exceptions arising from data formatting issues (such as entering a negative `UnitPrice`). Specifically, we'll look at how to add validation controls to the editing and inserting interfaces.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Liz Shulok. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Adding Validation Controls to the Editing and Inserting Interfaces (VB)

2/20/2020 • 19 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll see how easy it is to add validation controls to the EditItemTemplate and InsertItemTemplate of a data Web control, to provide a more foolproof user interface.

Introduction

The GridView and DetailsView controls in the examples we've explored over the past three tutorials have all been composed of BoundFields and CheckBoxFields (the field types automatically added by Visual Studio when binding a GridView or DetailsView to a data source control through the smart tag). When editing a row in a GridView or DetailsView, those BoundFields that are not read-only are converted into textboxes, from which the end user can modify the existing data. Similarly, when inserting a new record into a DetailsView control, those BoundFields whose `InsertVisible` property is set to `True` (the default) are rendered as empty textboxes, into which the user can provide the new record's field values. Likewise, CheckBoxFields, which are disabled in the standard, read-only interface, are converted into enabled checkboxes in the editing and inserting interfaces.

While the default editing and inserting interfaces for the BoundField and CheckBoxField can be helpful, the interface lacks any sort of validation. If a user makes a data entry mistake - such as omitting the `ProductName` field or entering an invalid value for `UnitsInStock` (such as -50) an exception will be raised from within the depths of the application architecture. While this exception can be gracefully handled as demonstrated in the previous tutorial, ideally the editing or inserting user interface would include validation controls to prevent a user from entering such invalid data in the first place.

In order to provide a customized editing or inserting interface, we need to replace the BoundField or CheckBoxField with a TemplateField. TemplateFields, which were the topic of discussion in the Using TemplateFields in the GridView Control and Using TemplateFields in the DetailsView Control tutorials, can consist of multiple templates defining separate interfaces for different row states. The TemplateField's `ItemTemplate` is used to when rendering read-only fields or rows in the DetailsView or GridView controls, whereas the `EditItemTemplate` and `InsertItemTemplate` indicate the interfaces to use for the editing and inserting modes, respectively.

In this tutorial we'll see how easy it is to add validation controls to the TemplateField's `EditItemTemplate` and `InsertItemTemplate` to provide a more foolproof user interface. Specifically, this tutorial takes the example created in the Examining the Events Associated with Inserting, Updating, and Deleting tutorial and augments the editing and inserting interfaces to include appropriate validation.

Step 1: Replicating the Example fromExamining the Events Associated with Inserting, Updating, and Deleting

In the Examining the Events Associated with Inserting, Updating, and Deleting tutorial we created a page that listed the names and prices of the products in an editable GridView. Additionally, the page included a DetailsView whose `DefaultMode` property was set to `Insert`, thereby always rendering in insert mode. From this DetailsView, the user could enter the name and price for a new product, click Insert, and have it added to the

system (see Figure 1).

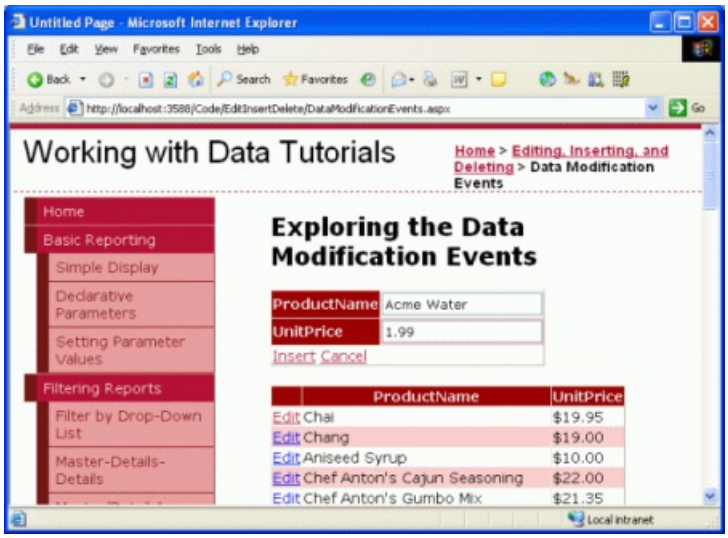


Figure 1: The Previous Example Allows Users to Add New Products and Edit Existing Ones (Click to view full-size image)

Our goal for this tutorial is to augment the DetailsView and GridView to provide validation controls. In particular, our validation logic will:

- Require that the name be provided when inserting or editing a product
- Require that the price be provided when inserting a record; when editing a record, we will still require a price, but will use the programmatic logic in the GridView's `RowUpdating` event handler already present from the earlier tutorial
- Ensure that the value entered for the price is a valid currency format

Before we can look at augmenting the previous example to include validation, we first need to replicate the example from the `DataModificationEvents.aspx` page to the page for this tutorial, `UIValidation.aspx`. To accomplish this we need to copy over both the `DataModificationEvents.aspx` page's declarative markup and its source code. First copy over the declarative markup by performing the following steps:

1. Open the `DataModificationEvents.aspx` page in Visual Studio
2. Go to the page's declarative markup (click on the Source button at the bottom of the page)
3. Copy the text within the `<asp:Content>` and `</asp:Content>` tags (lines 3 through 44), as shown in Figure 2.

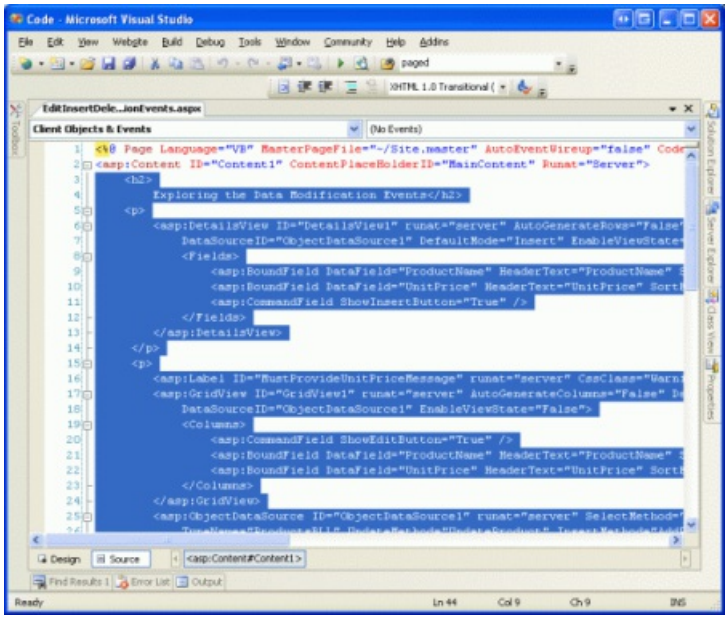


Figure 2: Copy the Text Within the `<asp:Content>` Control (Click to view full-size image)

1. Open the `UIValidation.aspx` page
2. Go to the page's declarative markup
3. Paste the text within the `<asp:Content>` control.

To copy over the source code, open the `DataModificationEvents.aspx.vb` page and copy just the text *within* the `EditInsertDelete_DataModificationEvents` class. Copy the three event handlers (`Page_Load` , `GridView1_RowUpdating` , and `ObjectDataSource1_Inserting`), but do **not** copy the class declaration. Paste the copied text *within* the `EditInsertDelete_UIValidation` class in `UIValidation.aspx.vb` .

After moving over the content and code from `DataModificationEvents.aspx` to `UIValidation.aspx` , take a moment to test out your progress in a browser. You should see the same output and experience the same functionality in each of these two pages (refer back to Figure 1 for a screen shot of `DataModificationEvents.aspx` in action).

Step 2: Converting the BoundFields Into TemplateFields

To add validation controls to the editing and inserting interfaces, the BoundFields used by the DetailsView and GridView controls need to be converted into TemplateFields. To achieve this, click on the Edit Columns and Edit Fields links in the GridView and DetailsView's smart tags, respectively. There, select each of the BoundFields and click the "Convert this field into a TemplateField" link.

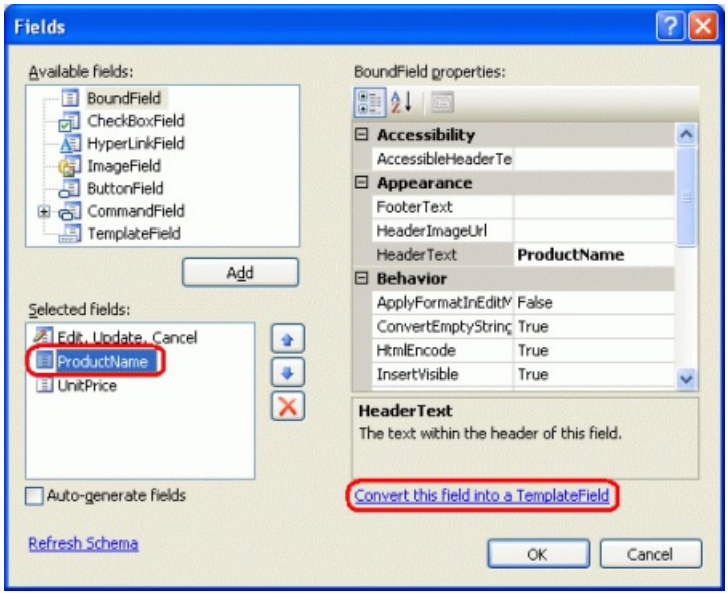


Figure 3: Convert Each of the DetailsView's and GridView's BoundFields Into TemplateFields (Click to view full-size image)

Converting a BoundField into a TemplateField through the Fields dialog box generates a TemplateField that exhibits the same read-only, editing, and inserting interfaces as the BoundField itself. The following markup shows the declarative syntax for the `ProductName` field in the DetailsView after it has been converted into a TemplateField:

```
<asp:TemplateField HeaderText="ProductName" SortExpression="ProductName">
    <EditItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
         Text='<%# Bind("ProductName") %>'></asp:TextBox>
    </EditItemTemplate>
    <InsertItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
         Text='<%# Bind("ProductName") %>'></asp:TextBox>
    </InsertItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
         Text='<%# Bind("ProductName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

Note that this TemplateField had three templates automatically created `ItemTemplate`, `EditItemTemplate`, and `InsertItemTemplate`. The `ItemTemplate` displays a single data field value (`ProductName`) using a Label Web control, while the `EditItemTemplate` and `InsertItemTemplate` present the data field value in a TextBox Web control that associates the data field with the TextBox's `Text` property using two-way databinding. Since we are only using the DetailsView in this page for inserting, you may remove the `ItemTemplate` and `EditItemTemplate` from the two TemplateFields, although there's no harm in leaving them.

Since the GridView does not support the built-in inserting features of the DetailsView, converting the GridView's `ProductName` field into a TemplateField results in only an `ItemTemplate` and `EditItemTemplate`:

```
<asp:TemplateField HeaderText="ProductName" SortExpression="ProductName">
    <EditItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
         Text='<%# Bind("ProductName") %>'></asp:TextBox>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
          Text='<%# Bind("ProductName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

By clicking the "Convert this field into a TemplateField," Visual Studio has created a TemplateField whose templates mimic the user interface of the converted BoundField. You can verify this by visiting this page through a browser. You'll find that the appearance and behavior of the TemplateFields is identical to the experience when BoundFields were used instead.

> **NOTE**
>
> Feel free to customize the editing interfaces in the templates as needed. For example, we may want to have the TextBox in the `UnitPrice` TemplateFields rendered as a smaller textbox than the `ProductName` textbox. To accomplish this you can set the TextBox's `Columns` property to an appropriate value or provide an absolute width via the `Width` property. In the next tutorial we'll see how to completely customize the editing interface by replacing the TextBox with an alternate data entry Web control.

Step 3: Adding the Validation Controls to the GridView's `EditItemTemplate` S

When constructing data entry forms, it is important that users enter any required fields and that all provided inputs are legal, properly-formatted values. To help ensure that a user's inputs are valid, ASP.NET provides five built-in validation controls that are designed to be used to validate the value of a single input control:

- RequiredFieldValidator ensures that a value has been provided
- CompareValidator validates a value against another Web control value or a constant value, or ensures that the value's format is legal for a specified data type
- RangeValidator ensures that a value is within a range of values
- RegularExpressionValidator validates a value against a regular expression
- CustomValidator validates a value against a custom, user-defined method

For more information on these five controls, check out the Validation Controls section of the ASP.NET Quickstart Tutorials.

For our tutorial we'll need to use a RequiredFieldValidator in both the DetailsView and GridView's `ProductName` TemplateFields and a RequiredFieldValidator in the DetailsView's `UnitPrice` TemplateField. Furthermore, we'll need to add a CompareValidator to both controls' `UnitPrice` TemplateFields that ensures that the entered price has a value greater than or equal to 0 and is presented in a valid currency format.

> **NOTE**
> While ASP.NET 1.x had these same five validation controls, ASP.NET 2.0 has added a number of improvements, the main two being client-side script support for browsers other than Internet Explorer and the ability to partition validation controls on a page into validation groups. For more information on the new validation control features in 2.0, refer to Dissecting the Validation Controls in ASP.NET 2.0.

Let's start by adding the necessary validation controls to the `EditItemTemplate` s in the GridView's TemplateFields. To accomplish this, click on the Edit Templates link from the GridView's smart tag to bring up the template editing interface. From here, you can select which template to edit from the drop-down list. Since we want to augment the editing interface, we need to add validation controls to the `ProductName` and `UnitPrice`'s `EditItemTemplate` s.

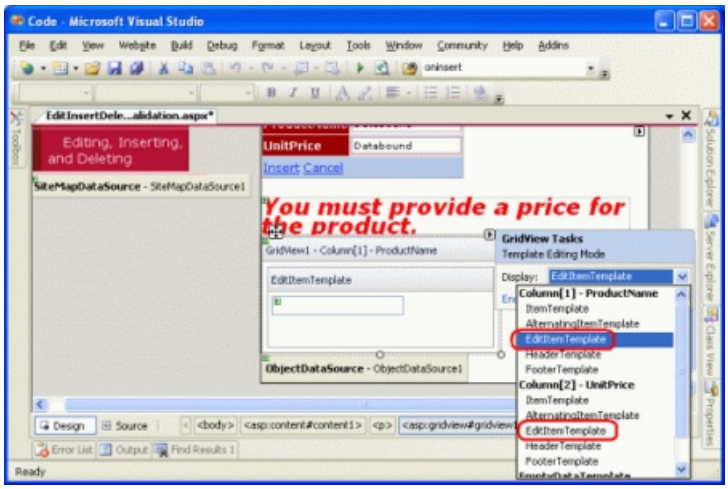


Figure 4: We Need to Extend the `ProductName` and `UnitPrice`'s `EditItemTemplate` s (Click to view full-size image)

In the `ProductName` `EditItemTemplate`, add a RequiredFieldValidator by dragging it from the Toolbox into the template editing interface, placing after the TextBox.

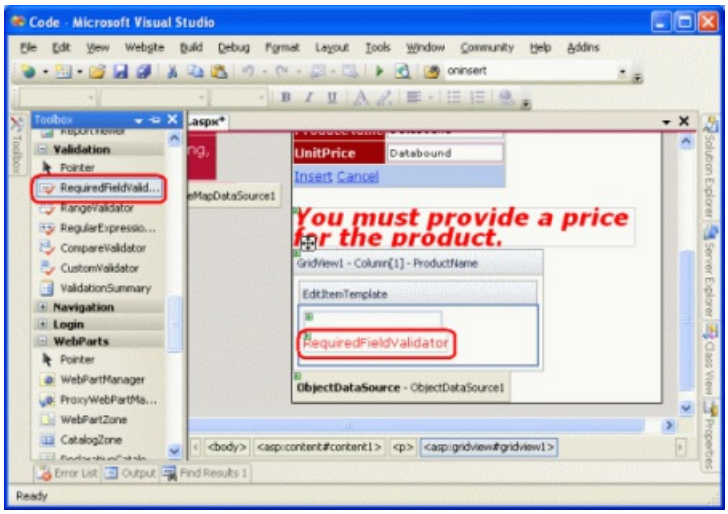


Figure 5: Add a RequiredFieldValidator to the `ProductName` `EditItemTemplate` (Click to view full-size image)

All validation controls work by validating the input of a single ASP.NET Web control. Therefore, we need to indicate that the RequiredFieldValidator we just added should validate against the TextBox in the `EditItemTemplate`; this is accomplished by setting the validation control's ControlToValidate property to the `ID` of the appropriate Web control. The TextBox currently has the rather nondescript `ID` of `TextBox1`, but let's change it to something more appropriate. Click on the TextBox in the template and then, from the Properties window, change the `ID` from `TextBox1` to `EditProductName`.

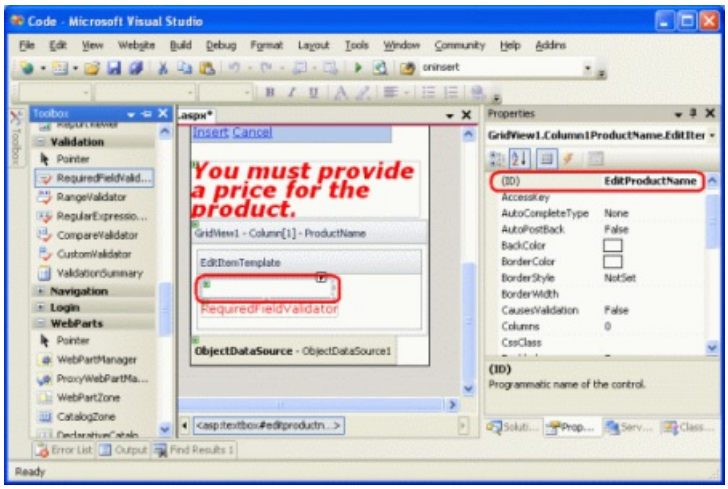


Figure 6: Change the TextBox's `ID` to `EditProductName` (Click to view full-size image)

Next, set the RequiredFieldValidator's `ControlToValidate` property to `EditProductName`. Finally, set the ErrorMessage property to "You must provide the product's name" and the Text property to "*". The `Text` property value, if provided, is the text that is displayed by the validation control if the validation fails. The `ErrorMessage` property value, which is required, is used by the ValidationSummary control; if the `Text` property value is omitted, the `ErrorMessage` property value is also the text displayed by the validation control on invalid input.

After setting these three properties of the RequiredFieldValidator, your screen should look similar to Figure 7.

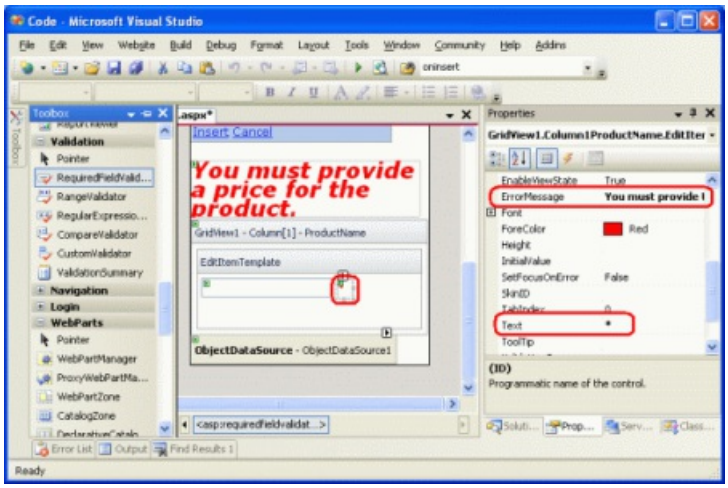


Figure 7: Set the RequiredFieldValidator's `ControlToValidate`, `ErrorMessage`, and `Text` Properties (Click to view full-size image)

With the RequiredFieldValidator added to the `ProductName` `EditItemTemplate`, all that remains is to add the necessary validation to the `UnitPrice` `EditItemTemplate`. Since we've decided that, for this page, the `UnitPrice` is optional when editing a record, we don't need to add a RequiredFieldValidator. We do, however, need to add a CompareValidator to ensure that the `UnitPrice`, if supplied, is properly formatted as a currency and is greater than or equal to 0.

Before we add the CompareValidator to the `UnitPrice` `EditItemTemplate`, let's first change the TextBox Web control's ID from `TextBox2` to `EditUnitPrice`. After making this change, add the CompareValidator, setting its `ControlToValidate` property to `EditUnitPrice`, its `ErrorMessage` property to "The price must be greater than or equal to zero and cannot include the currency symbol", and its `Text` property to "*".

To indicate that the `UnitPrice` value must be greater than or equal to 0, set the CompareValidator's Operator property to `GreaterThanEqual`, its ValueToCompare property to "0", and its Type property to `Currency`. The following declarative syntax shows the `UnitPrice` TemplateField's `EditItemTemplate` after these changes have been made:

```
<EditItemTemplate>
    <asp:TextBox ID="EditUnitPrice" runat="server"
      Text='<%# Bind("UnitPrice", "{0:c}") %>'
      Columns="6"></asp:TextBox>
    <asp:CompareValidator ID="CompareValidator1" runat="server"
        ControlToValidate="EditUnitPrice"
        ErrorMessage="The price must be greater than or equal to zero and
                      cannot include the currency symbol"
        Operator="GreaterThanEqual" Type="Currency"
        ValueToCompare="0">*</asp:CompareValidator>
</EditItemTemplate>
```

After making these changes, open the page in a browser. If you attempt to omit the name or enter an invalid price value when editing a product, an asterisk appears next to the textbox. As Figure 8 shows, a price value that includes the currency symbol such as $19.95 is considered invalid. The CompareValidator's `Currency` `Type` allows for digit separators (such as commas or periods, depending on the culture settings) and a leading plus or minus sign, but does *not* permit a currency symbol. This behavior may perplex users as the editing interface currently renders the `UnitPrice` using the currency format.

> **NOTE**
>
> Recall that in the *Events Associated with Inserting, Updating, and Deleting* tutorial we set the BoundField's `DataFormatString` property to `{0:c}` in order to format it as a currency. Furthermore, we set the `ApplyFormatInEditMode` property to true, causing the GridView's editing interface to format the `UnitPrice` as a currency. When converting the BoundField into a TemplateField, Visual Studio noted these settings and formatted the TextBox's `Text` property as a currency using the databinding syntax `<%# Bind("UnitPrice", "{0:c}") %>`.

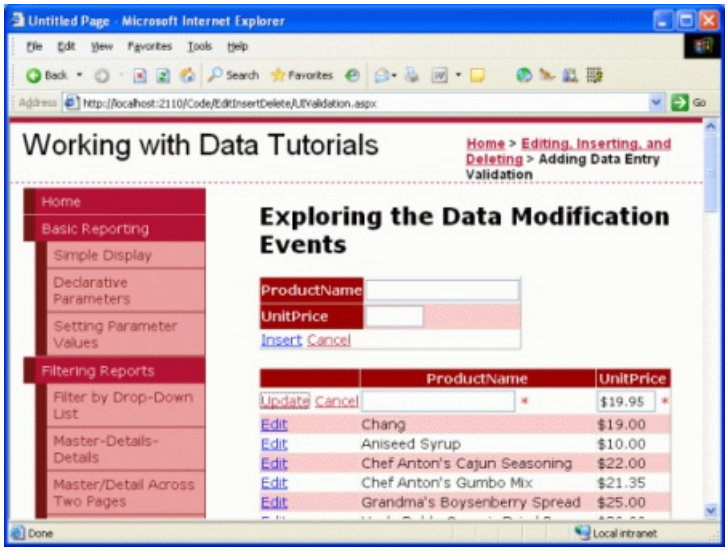


Figure 8: An Asterisk Appears Next to the Textboxes with Invalid Input (Click to view full-size image)

While the validation works as-is, the user has to manually remove the currency symbol when editing a record, which is not acceptable. To remedy this, we have three options:

1. Configure the `EditItemTemplate` so that the `UnitPrice` value is not formatted as a currency.
2. Allow the user to enter a currency symbol by removing the CompareValidator and replacing it with a RegularExpressionValidator that properly checks for a properly formatted currency value. The problem here is that the regular expression to validate a currency value is not pretty and would require writing code if we wanted to incorporate culture settings.
3. Remove the validation control altogether and rely on server-side validation logic in the GridView's `RowUpdating` event handler.

Let's go with option #1 for this exercise. Currently the `UnitPrice` is formatted as a currency due to the databinding expression for the TextBox in the `EditItemTemplate`: `<%# Bind("UnitPrice", "{0:c}") %>`. Change the Bind statement to `Bind("UnitPrice", "{0:n2}")`, which formats the result as a number with two digits of precision. This can be done directly through the declarative syntax or by clicking on the Edit DataBindings link from the `EditUnitPrice` TextBox in the `UnitPrice` TemplateField's `EditItemTemplate` (see Figures 9 and 10).

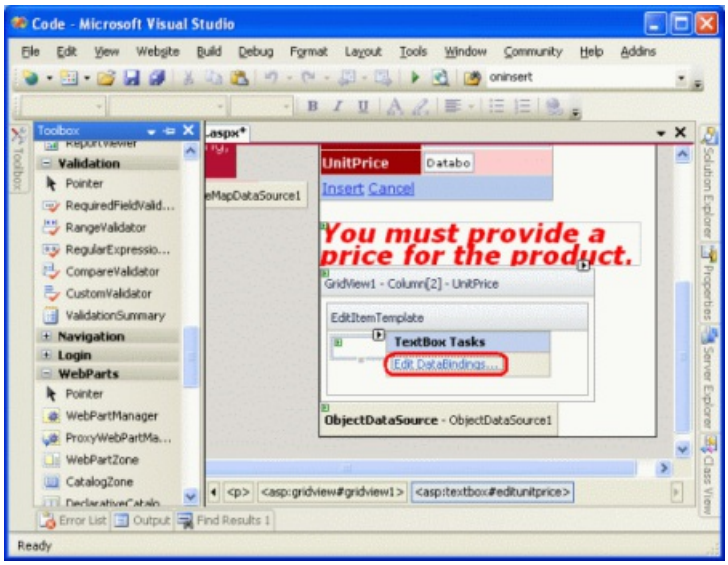


Figure 9: Click on the TextBox's Edit DataBindings link (Click to view full-size image)

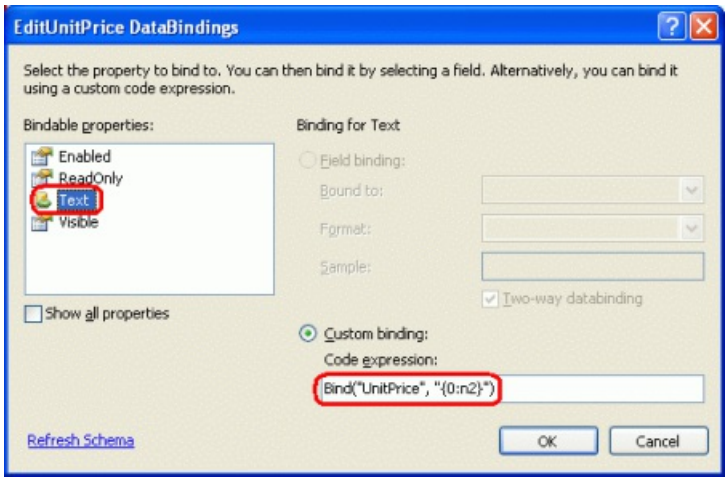


Figure 10: Specify the Format Specifier in the `Bind` Statement (Click to view full-size image)

With this change, the formatted price in the editing interface includes commas as the group separator and a period as the decimal separator, but leaves off the currency symbol.

> **NOTE**
>
> The `UnitPrice` `EditItemTemplate` doesn't include a RequiredFieldValidator, allowing the postback to ensue and the updating logic to commence. However, the `RowUpdating` event handler copied over from the *Examining the Events Associated with Inserting, Updating, and Deleting* tutorial includes a programmatic check that ensures that the `UnitPrice` is provided. Feel free to remove this logic, leave it in as-is, or add a RequiredFieldValidator to the `UnitPrice` `EditItemTemplate`.

Step 4: Summarizing Data Entry Problems

In addition to the five validation controls, ASP.NET includes the ValidationSummary control, which displays the `ErrorMessage` s of those validation controls that detected invalid data. This summary data can be displayed as text on the web page or through a modal, client-side messagebox. Let's enhance this tutorial to include a client-side messagebox summarizing any validation problems.

To accomplish this, drag a ValidationSummary control from the Toolbox onto the Designer. The location of the Validation control doesn't really matter, since we're going to configure it to only display the summary as a messagebox. After adding the control, set its ShowSummary property to `False` and its ShowMessageBox property to `True`. With this addition, any validation errors are summarized in a client-side messagebox.

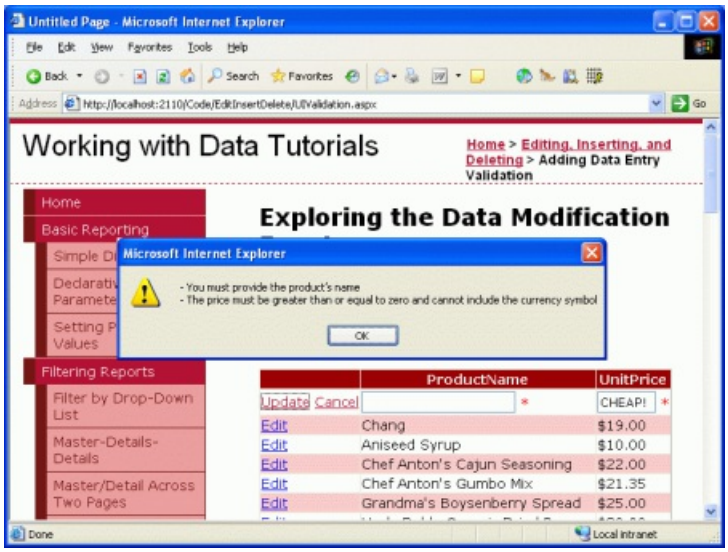


Figure 11: The Validation Errors are Summarized in a Client-Side Messagebox (Click to view full-size image)

Step 5: Adding the Validation Controls to the DetailsView's `InsertItemTemplate`

All that remains for this tutorial is to add the validation controls to the DetailsView's inserting interface. The process of adding validation controls to the DetailsView's templates is identical to that examined in Step 3; therefore, we'll breeze through the task in this step. As we did with the GridView's `EditItemTemplate` s, I encourage you to rename the `ID` s of the TextBoxes from the nondescript `TextBox1` and `TextBox2` to `InsertProductName` and `InsertUnitPrice`.

Add a RequiredFieldValidator to the `ProductName` `InsertItemTemplate`. Set the `ControlToValidate` to the `ID` of the TextBox in the template, its `Text` property to "*" and its `ErrorMessage` property to "You must provide the product's name".

Since the `UnitPrice` is required for this page when adding a new record, add a RequiredFieldValidator to the `UnitPrice` `InsertItemTemplate`, setting its `ControlToValidate`, `Text`, and `ErrorMessage` properties appropriately. Finally, add a CompareValidator to the `UnitPrice` `InsertItemTemplate` as well, configuring its `ControlToValidate`, `Text`, `ErrorMessage`, `Type`, `Operator`, and `ValueToCompare` properties just like we did with the `UnitPrice`'s CompareValidator in the GridView's `EditItemTemplate`.

After adding these validation controls, a new product cannot be added to the system if its name is not supplied or if its price is a negative number or illegally formatted.

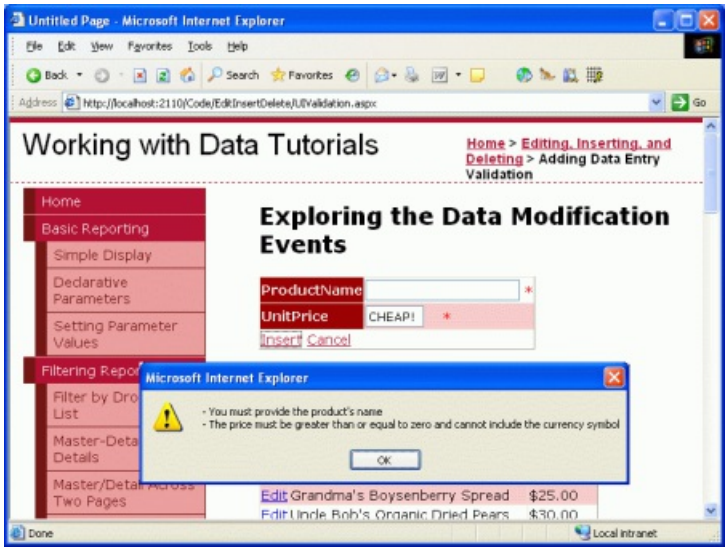


Figure 12: Validation Logic has been Added to the DetailsView's Inserting Interface (Click to view full-size image)

Step 6: Partitioning the Validation Controls Into Validation Groups

Our page consists of two logically disparate sets of validation controls: those that correspond to the GridView's editing interface and those that correspond to the DetailsView's inserting interface. By default, when a postback occurs *all* validation controls on the page are checked. However, when editing a record we don't want the DetailsView's inserting interface's validation controls to validate. Figure 13 illustrates our current dilemma when a user is editing a product with perfectly legal values, clicking Update causes a validation error because the name and price values in the inserting interface are blank.

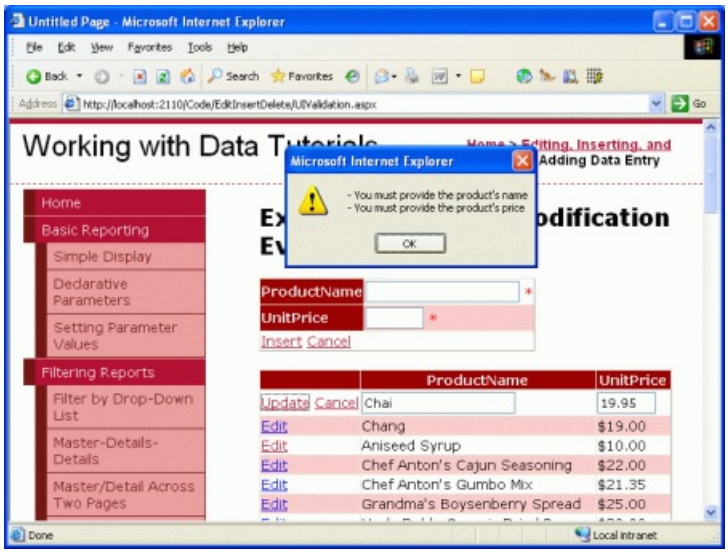


Figure 13: Updating a Product Causes the Inserting Interface's Validation Controls to Fire (Click to view full-size image)

The validation controls in ASP.NET 2.0 can be partitioned into validation groups through their `ValidationGroup` property. To associate a set of validation controls in a group, simply set their `ValidationGroup` property to the same value. For our tutorial, set the `ValidationGroup` properties of the validation controls in the GridView's TemplateFields to `EditValidationControls` and the `ValidationGroup` properties of the DetailsView's TemplateFields to `InsertValidationControls`. These changes can be done directly in the declarative markup or through the Properties window when using the Designer's edit template interface.

In addition to the validation controls, the Button and Button-related controls in ASP.NET 2.0 also include a `ValidationGroup` property. A validation group's validators are checked for validity only when a postback is induced by a Button that has the same `ValidationGroup` property setting. For example, in order for the DetailsView's Insert button to trigger the `InsertValidationControls` validation group we need to set the CommandField's `ValidationGroup` property to `InsertValidationControls` (see Figure 14). Additionally, set the GridView's CommandField's `ValidationGroup` property to `EditValidationControls`.

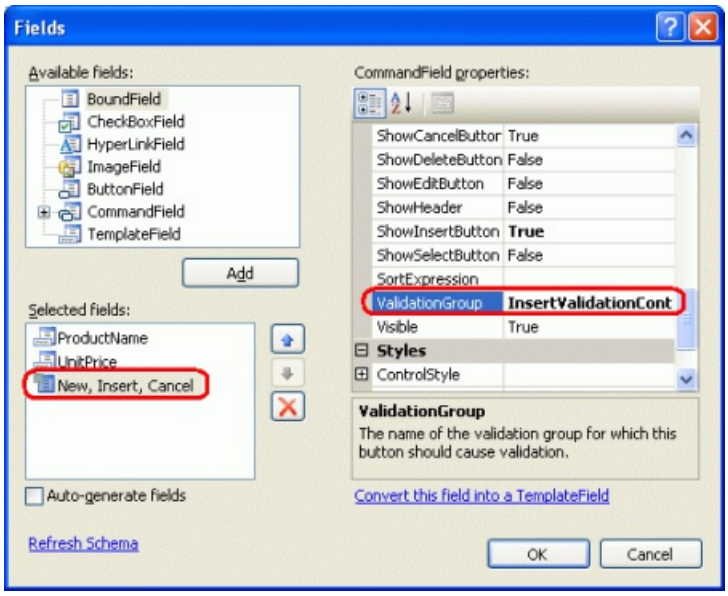


Figure 14: Set the DetailsView's CommandField's `ValidationGroup` property to `InsertValidationControls` (Click to view full-size image)

After these changes, the DetailsView and GridView's TemplateFields and CommandFields should look similar to the following:

The DetailsView's TemplateFields and CommandField

```
<asp:TemplateField HeaderText="ProductName"
  SortExpression="ProductName">
    <InsertItemTemplate>
        <asp:TextBox ID="InsertProductName" runat="server"
         Text='<%# Bind("ProductName") %>'></asp:TextBox>
        <asp:RequiredFieldValidator ID="RequiredFieldValidator2"
          runat="server" ControlToValidate="InsertProductName"
            ErrorMessage="You must provide the product name"
            ValidationGroup="InsertValidationControls">*
        </asp:RequiredFieldValidator>
    </InsertItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="UnitPrice" SortExpression="UnitPrice">
    <InsertItemTemplate>
         <asp:TextBox ID="InsertUnitPrice" runat="server"
           Text='<%# Bind("UnitPrice") %>' Columns="6">
         </asp:TextBox>
         <asp:RequiredFieldValidator ID="RequiredFieldValidator3"
           runat="server" ControlToValidate="InsertUnitPrice"
            ErrorMessage="You must provide the product price"
            ValidationGroup="InsertValidationControls">*
         </asp:RequiredFieldValidator>
        <asp:CompareValidator ID="CompareValidator2" runat="server"
           ControlToValidate="InsertUnitPrice"
           ErrorMessage="The price must be greater than or equal to zero and
                         cannot include the currency symbol"
           Operator="GreaterThanEqual" Type="Currency" ValueToCompare="0"
           ValidationGroup="InsertValidationControls">*
        </asp:CompareValidator>
     </InsertItemTemplate>
 </asp:TemplateField>
<asp:CommandField ShowInsertButton="True"
  ValidationGroup="InsertValidationControls" />
```

The GridView's CommandField and TemplateFields

```
<asp:CommandField ShowEditButton="True" ValidationGroup="EditValidationControls" />
<asp:TemplateField HeaderText="ProductName"
  SortExpression="ProductName">
    <EditItemTemplate>
        <asp:TextBox ID="EditProductName" runat="server"
          Text='<%# Bind("ProductName") %>'>
        </asp:TextBox>
        <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
            runat="server" ControlToValidate="EditProductName"
            ErrorMessage="You must provide the product name"
            ValidationGroup="EditValidationControls">*
        </asp:RequiredFieldValidator>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
          Text='<%# Bind("ProductName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="UnitPrice" SortExpression="UnitPrice">
    <EditItemTemplate>
        <asp:TextBox ID="EditUnitPrice" runat="server"
          Text='<%# Bind("UnitPrice", "{0:n2}") %>' Columns="6"></asp:TextBox>
        <asp:CompareValidator ID="CompareValidator1" runat="server"
            ControlToValidate="EditUnitPrice"
            ErrorMessage="The price must be greater than or equal to zero and
                           cannot include the currency symbol"
            Operator="GreaterThanEqual" Type="Currency"
            ValueToCompare="0"
            ValidationGroup="EditValidationControls">*
        </asp:CompareValidator>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label2" runat="server"
            Text='<%# Bind("UnitPrice", "{0:c}") %>'>
        </asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

At this point the edit-specific validation controls fire only when the GridView's Update button is clicked and the insert-specific validation controls fire only when the DetailsView's Insert button is clicked, resolving the problem highlighted by Figure 13. However, with this change our ValidationSummary control no longer displays when entering invalid data. The ValidationSummary control also contains a `ValidationGroup` property and only shows summary information for those validation controls in its validation group. Therefore, we need to have two validation controls in this page, one for the `InsertValidationControls` validation group and one for `EditValidationControls`.

```
<asp:ValidationSummary ID="ValidationSummary1" runat="server"
    ShowMessageBox="True" ShowSummary="False"
    ValidationGroup="EditValidationControls" />
<asp:ValidationSummary ID="ValidationSummary2" runat="server"
    ShowMessageBox="True" ShowSummary="False"
    ValidationGroup="InsertValidationControls" />
```

With this addition our tutorial is complete!

Summary

While BoundFields can provide both an inserting and editing interface, the interface is not customizable. Commonly, we want to add validation controls to the editing and inserting interface to ensure that the user enters required inputs in a legal format. To accomplish this we must convert the BoundFields into TemplateFields and add the validation controls to the appropriate template(s). In this tutorial we extended the example from the

Examining the Events Associated with Inserting, Updating, and Deleting tutorial, adding validation controls to both the DetailsView's inserting interface and the GridView's editing interface. Moreover, we saw how to display summary validation information using the ValidationSummary control and how to partition the validation controls on the page into distinct validation groups.

As we saw in this tutorial, TemplateFields allow the editing and inserting interfaces to be augmented to include validation controls. TemplateFields can also be extended to include additional input Web controls, enabling the TextBox to be replaced by a more suitable Web control. In our next tutorial we'll see how to replace the TextBox control with a data-bound DropDownList control, which is ideal when editing a foreign key (such as `CategoryID` or `SupplierID` in the `Products` table).

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Liz Shulok and Zack Jones. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Customizing the Data Modification Interface (VB)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll look at how to customize the interface of an editable GridView, by replacing the standard TextBox and CheckBox controls with alternative input Web controls.

Introduction

The BoundFields and CheckBoxFields used by the GridView and DetailsView controls simplify the process of modifying data due to their ability to render read-only, editable, and insertable interfaces. These interfaces can be rendered without the need for adding any additional declarative markup or code. However, the BoundField and CheckBoxField's interfaces lack the customizability often needed in real-world scenarios. In order to customize the editable or insertable interface in a GridView or DetailsView we need to instead use a TemplateField.

In the preceding tutorial we saw how to customize the data modification interfaces by adding validation Web controls. In this tutorial we'll look at how to customize the actual data collection Web controls, replacing the BoundField and CheckBoxField's standard TextBox and CheckBox controls with alternative input Web controls. In particular, we'll build an editable GridView that allows a product's name, category, supplier, and discontinued status to be updated. When editing a particular row, the category and supplier fields will render as DropDownLists, containing the set of available categories and suppliers to choose from. Furthermore, we'll replace the CheckBoxField's default CheckBox with a RadioButtonList control that offers two options: "Active" and "Discontinued".

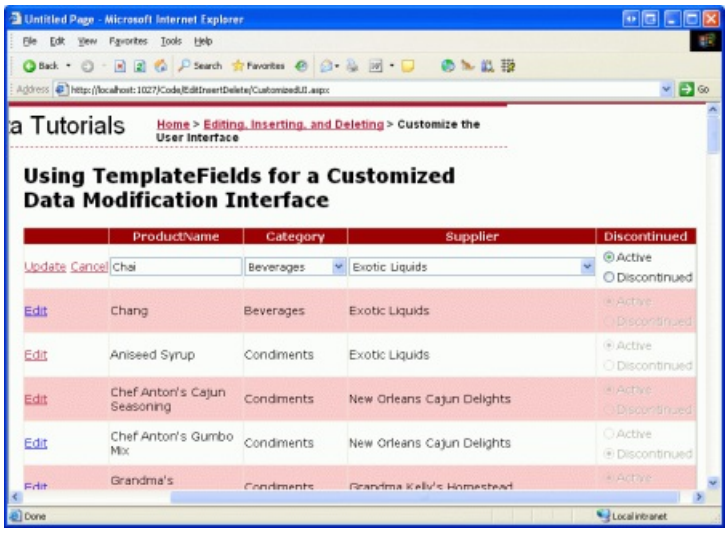


Figure 1: The GridView's Editing Interface Includes DropDownLists and RadioButtons (Click to view full-size image)

Step 1: Creating the Appropriate `UpdateProduct` Overload

In this tutorial we will build an editable GridView that permits editing of a product's name, category, supplier, and discontinued status. Therefore, we need an `UpdateProduct` overload that accepts five input parameters these four product values plus the `ProductID`. Like in our previous overloads, this one will:

1. Retrieve the product information from the database for the specified `ProductID`,
2. Update the `ProductName`, `CategoryID`, `SupplierID`, and `Discontinued` fields, and
3. Send the update request to the DAL through the TableAdapter's `Update()` method.

For brevity, for this particular overload I've omitted the business rule check that ensures a product being marked as discontinued isn't the only product offered by its supplier. Feel free to add it in if you prefer, or, ideally, refactor out the logic to a separate method.

The following code shows the new `UpdateProduct` overload in the `ProductsBLL` class:

```
<System.ComponentModel.DataObjectMethodAttribute(System.ComponentModel.DataObjectMethodType.Update, False)>
Public Function UpdateProduct(
    ByVal productName As String, ByVal categoryID As Nullable(Of Integer),
    ByVal supplierID As Nullable(Of Integer), ByVal discontinued As Boolean,
    ByVal productID As Integer)
    As Boolean
    Dim products As Northwind.ProductsDataTable = Adapter.GetProductByProductID(productID)
    If products.Count = 0 Then
        Return False
    End If
    Dim product As Northwind.ProductsRow = products(0)
    product.ProductName = productName
    If Not supplierID.HasValue Then
        product.SetSupplierIDNull()
    Else
        product.SupplierID = supplierID.Value
    End If
    If Not categoryID.HasValue Then
        product.SetCategoryIDNull()
    Else
        product.CategoryID = categoryID.Value
    End If
    product.Discontinued = discontinued
    Dim rowsAffected As Integer = Adapter.Update(product)
    Return rowsAffected = 1
End Function
```

Step 2: Crafting the Editable GridView

With the `UpdateProduct` overload added, we're ready to create our editable GridView. Open the `CustomizedUI.aspx` page in the `EditInsertDelete` folder and add a GridView control to the Designer. Next, create a new ObjectDataSource from the GridView's smart tag. Configure the ObjectDataSource to retrieve product information via the `ProductBLL` class's `GetProducts()` method and to update product data using the `UpdateProduct` overload we just created. From the INSERT and DELETE tabs, select (None) from the drop-down lists.

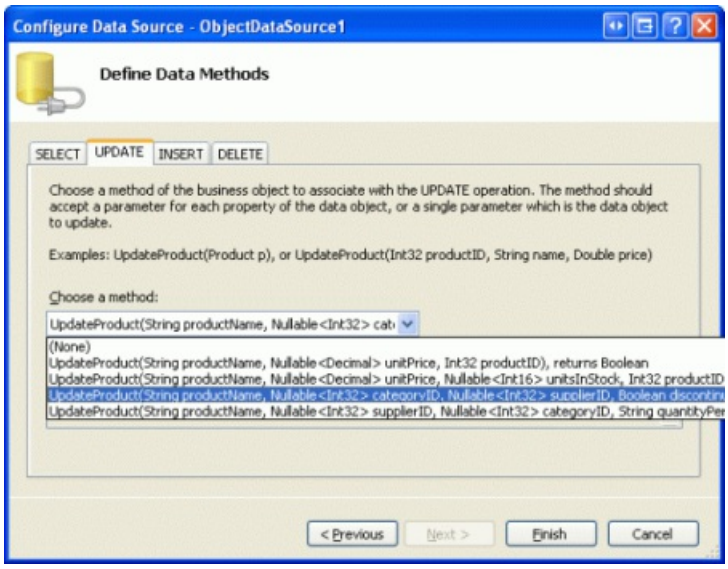


Figure 2: Configure the ObjectDataSource to Use the `UpdateProduct` Overload Just Created (Click to view full-size image)

As we've seen throughout the data modification tutorials, the declarative syntax for the ObjectDataSource created by Visual Studio assigns the `OldValuesParameterFormatString` property to `original_{0}`. This, of course, won't work with our Business Logic Layer since our methods don't expect the original `ProductID` value to be passed in. Therefore, as we've done in previous tutorials, take a moment to remove this property assignment from the declarative syntax or, instead, set this property's value to `{0}`.

After this change, the ObjectDataSource's declarative markup should look like the following:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    SelectMethod="GetProducts" TypeName="ProductsBLL"
    UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

Note that the `OldValuesParameterFormatString` property has been removed and that there is a `Parameter` in the `UpdateParameters` collection for each of the input parameters expected by our `UpdateProduct` overload.

While the ObjectDataSource is configured to update only a subset of product values, the GridView currently shows *all* of the product fields. Take a moment to edit the GridView so that:

- It only includes the `ProductName`, `SupplierName`, `CategoryName` BoundFields and the `Discontinued` CheckBoxField
- The `CategoryName` and `SupplierName` fields to appear before (to the left of) the `Discontinued` CheckBoxField
- The `CategoryName` and `SupplierName` BoundFields' `HeaderText` property is set to "Category" and "Supplier", respectively
- Editing support is enabled (check the Enable Editing checkbox in the GridView's smart tag)

After these changes, the Designer will look similar to Figure 3, with the GridView's declarative syntax shown below.

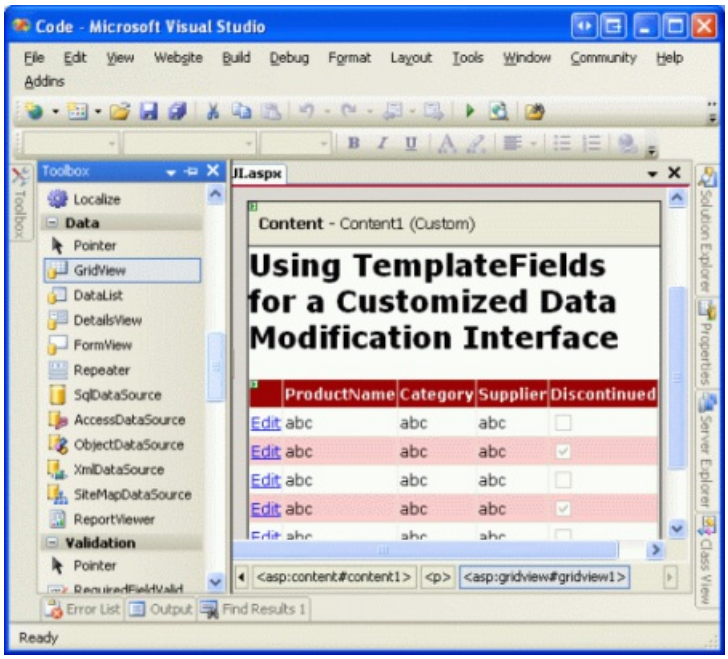


Figure 3: Remove the Unneeded Fields from the GridView (Click to view full-size image)

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:BoundField DataField="ProductName"
           HeaderText="ProductName" SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
           ReadOnly="True"
           SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
           ReadOnly="True"
           SortExpression="SupplierName" />
        <asp:CheckBoxField DataField="Discontinued"
           HeaderText="Discontinued" SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
```

At this point the GridView's read-only behavior is complete. When viewing the data, each product is rendered as a row in the GridView, showing the product's name, category, supplier, and discontinued status.

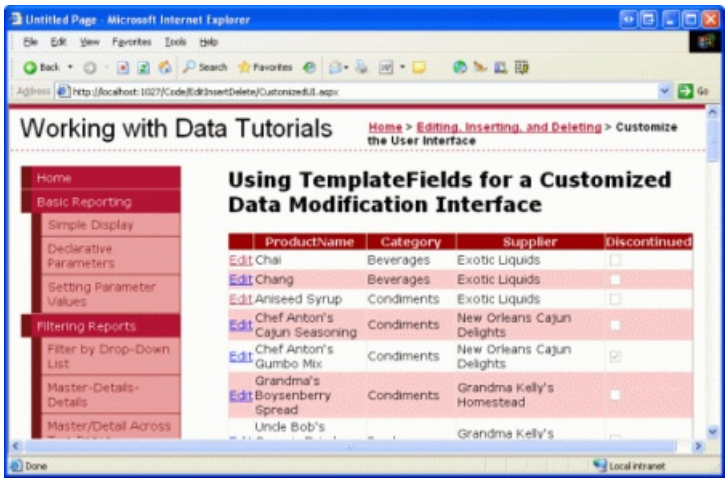


Figure 4: The GridView's Read-Only Interface is Complete (Click to view full-size image)

> **NOTE**
> As discussed in An Overview of Inserting, Updating, and Deleting Data tutorial, it is vitally important that the GridView s view state be enabled (the default behavior). If you set the GridView s `EnableViewState` property to `false`, you run the risk of having concurrent users unintentionally deleting or editing records. See WARNING: Concurrency Issue with ASP.NET 2.0 GridViews/DetailsView/FormViews that Support Editing and/or Deleting and Whose View State is Disabled for more information.

Step 3: Using a DropDownList for the Category and Supplier Editing Interfaces

Recall that the `ProductsRow` object contains `CategoryID`, `CategoryName`, `SupplierID`, and `SupplierName` properties, which provide the actual foreign-key ID values in the `Products` database table and the corresponding `Name` values in the `Categories` and `Suppliers` tables. The `ProductRow`'s `CategoryID` and `SupplierID` can both be read from and written to, while the `CategoryName` and `SupplierName` properties are marked read-only.

Due to the read-only status of the `CategoryName` and `SupplierName` properties, the corresponding BoundFields have had their `ReadOnly` property set to `True`, preventing these values from being modified when a row is edited. While we can set the `ReadOnly` property to `False`, rendering the `CategoryName` and `SupplierName` BoundFields as TextBoxes during editing, such an approach will result in an exception when the user attempts to update the product since there is no `UpdateProduct` overload that takes in `CategoryName` and `SupplierName` inputs. In fact, we don't want to create such an overload for two reasons:

- The `Products` table doesn't have `SupplierName` or `CategoryName` fields, but `SupplierID` and `CategoryID`. Therefore, we want our method to be passed these particular ID values, not their lookup tables' values.
- Requiring the user to type in the name of the supplier or category is less than ideal, as it requires the user to know the available categories and suppliers and their correct spellings.

The supplier and category fields should display the category and suppliers' names when in read-only mode (as it does now) and a drop-down list of applicable options when being edited. Using a drop-down list, the end user can quickly see what categories and suppliers are available to choose among and can more easily make their selection.

To provide this behavior, we need to convert the `SupplierName` and `CategoryName` BoundFields into TemplateFields whose `ItemTemplate` emits the `SupplierName` and `CategoryName` values and whose `EditItemTemplate` uses a DropDownList control to list the available categories and suppliers.

Adding the `Categories` and `Suppliers` DropDownLists

Start by converting the `SupplierName` and `CategoryName` BoundFields into TemplateFields by: clicking on the Edit Columns link from the GridView's smart tag; selecting the BoundField from the list in the lower left; and clicking the "Convert this field into a TemplateField" link. The conversion process will create a TemplateField with both an `ItemTemplate` and an `EditItemTemplate`, as shown in the declarative syntax below:

```
<asp:TemplateField HeaderText="Category" SortExpression="CategoryName">
    <EditItemTemplate>
        <asp:Label ID="Label1" runat="server"
          Text='<%# Eval("CategoryName") %>'></asp:Label>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
          Text='<%# Bind("CategoryName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

Since the BoundField was marked as read-only, both the `ItemTemplate` and `EditItemTemplate` contain a Label Web control whose `Text` property is bound to the applicable data field (`CategoryName`, in the syntax above). We need to modify the `EditItemTemplate`, replacing the Label Web control with a DropDownList control.

As we've seen in previous tutorials, the template can be edited through the Designer or directly from the declarative syntax. To edit it through the Designer, click on the Edit Templates link from the GridView's smart tag and choose to work with the Category field's `EditItemTemplate`. Remove the Label Web control and replace it with a DropDownList control, setting the DropDownList's ID property to `Categories`.

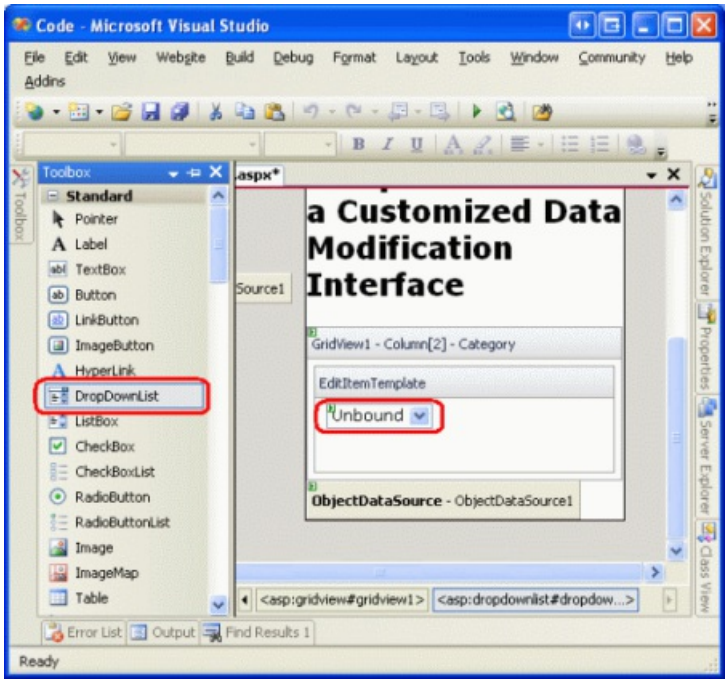


Figure 5: Remove the TexBox and Add a DropDownList to the `EditItemTemplate` (Click to view full-size image)

We next need to populate the DropDownList with the available categories. Click on the Choose Data Source link from the DropDownList's smart tag and opt to create a new ObjectDataSource named `CategoriesDataSource`.

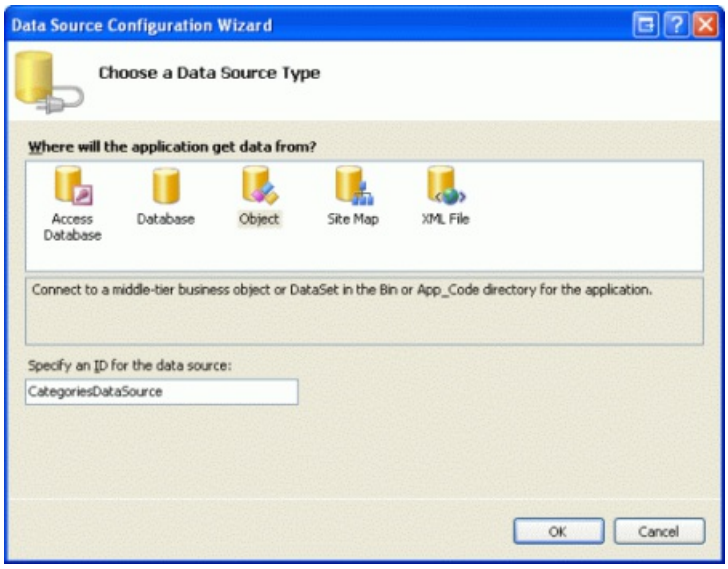


Figure 6: Create a New ObjectDataSource Control Named `CategoriesDataSource` (Click to view full-size image)

To have this ObjectDataSource return all of the categories, bind it to the `CategoriesBLL` class's `GetCategories()` method.

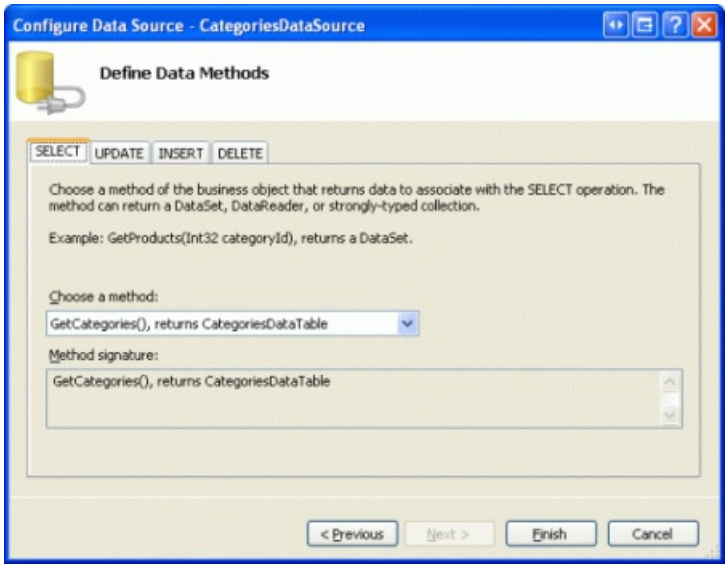


Figure 7: Bind the ObjectDataSource to the `CategoriesBLL`'s `GetCategories()` Method (Click to view full-size image)

Finally, configure the DropDownList's settings such that the `CategoryName` field is displayed in each DropDownList `ListItem` with the `CategoryID` field used as the value.

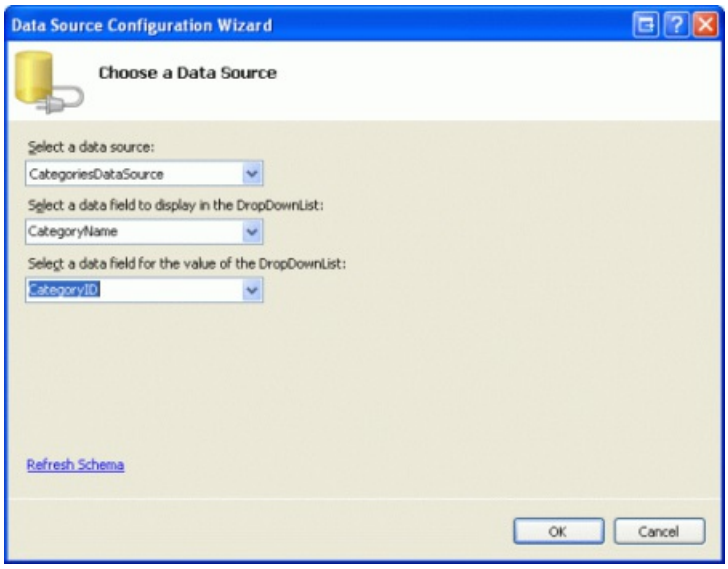


Figure 8: Have the `CategoryName` Field Displayed and the `CategoryID` Used as the Value (Click to view full-size image)

After making these changes the declarative markup for the `EditItemTemplate` in the `CategoryName` TemplateField will include both a DropDownList and an ObjectDataSource:

```
<asp:TemplateField HeaderText="Category" SortExpression="CategoryName">
    <EditItemTemplate>
        <asp:DropDownList ID="Categories" runat="server"
          DataSourceID="CategoriesDataSource"
          DataTextField="CategoryName" DataValueField="CategoryID">
        </asp:DropDownList>
        <asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
            OldValuesParameterFormatString="original_{0}"
            SelectMethod="GetCategories" TypeName="CategoriesBLL">
        </asp:ObjectDataSource>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label1" runat="server"
          Text='<%# Bind("CategoryName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

NOTE

The DropDownList in the `EditItemTemplate` must have its view state enabled. We will soon add databinding syntax to the DropDownList's declarative syntax and databinding commands like `Eval()` and `Bind()` can only appear in controls whose view state is enabled.

Repeat these steps to add a DropDownList named `Suppliers` to the `SupplierName` TemplateField's `EditItemTemplate`. This will involve adding a DropDownList to the `EditItemTemplate` and creating another ObjectDataSource. The `Suppliers` DropDownList's ObjectDataSource, however, should be configured to invoke the `SuppliersBLL` class's `GetSuppliers()` method. Additionally, configure the `Suppliers` DropDownList to display the `CompanyName` field and use the `SupplierID` field as the value for its `ListItem` s.

After adding the DropDownLists to the two `EditItemTemplate` s, load the page in a browser and click the Edit button for the Chef Anton's Cajun Seasoning product. As Figure 9 shows, the product's category and supplier columns are rendered as drop-down lists containing the available categories and suppliers to choose from. However, note that the *first* items in both drop-down lists are selected by default (Beverages for the category and Exotic Liquids as the supplier), even though Chef Anton's Cajun Seasoning is a Condiment supplied by New

Orleans Cajun Delights.

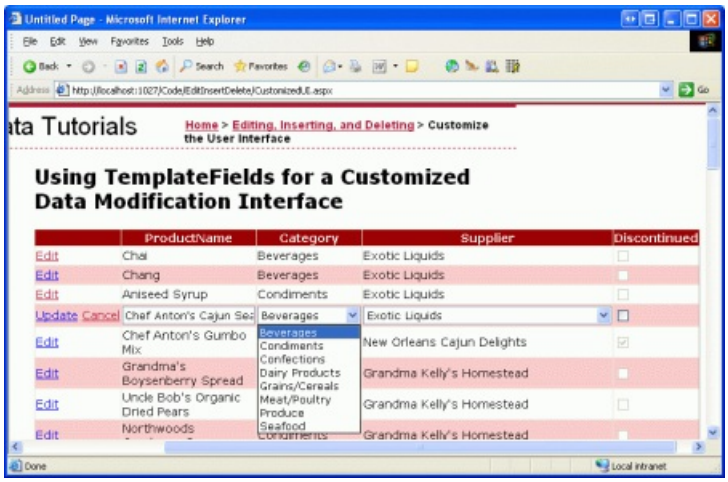


Figure 9: The First Item in the Drop-Down Lists is Selected by Default (Click to view full-size image)

Furthermore, if you click Update, you'll find that the product's `CategoryID` and `SupplierID` values are set to `NULL`. Both of these undesired behaviors are caused because the DropDownLists in the `EditItemTemplate` s are not bound to any data fields from the underlying product data.

Binding the DropDownLists to the `CategoryID` and `SupplierID` Data Fields

In order to have the edited product's category and supplier drop-down lists set to the appropriate values and to have these values sent back to the BLL's `UpdateProduct` method upon clicking Update, we need to bind the DropDownLists' `SelectedValue` properties to the `CategoryID` and `SupplierID` data fields using two-way databinding. To accomplish this with the `Categories` DropDownList, you can add `SelectedValue='<%# Bind("CategoryID") %>'` directly to the declarative syntax.

Alternatively, you can set the DropDownList's databindings by editing the template through the Designer and clicking the Edit DataBindings link from the DropDownList's smart tag. Next, indicate that the `SelectedValue` property should be bound to the `CategoryID` field using two-way databinding (see Figure 10). Repeat either the declarative or Designer process to bind the `SupplierID` data field to the `Suppliers` DropDownList.

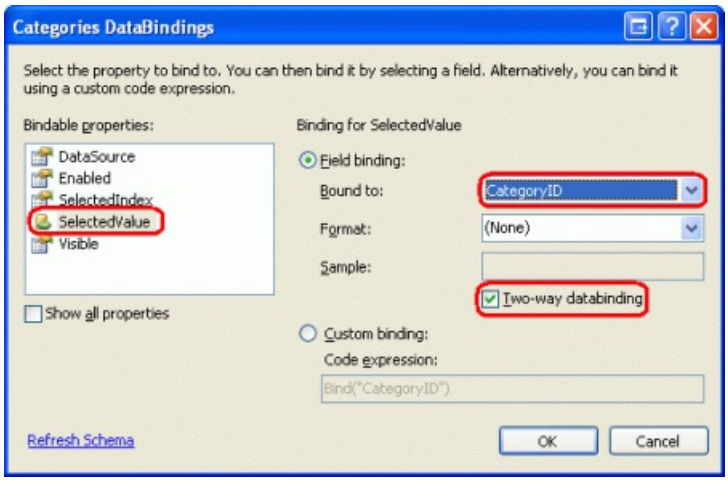


Figure 10: Bind the `CategoryID` to the DropDownList's `SelectedValue` Property Using Two-Way Databinding (Click to view full-size image)

Once the bindings have been applied to the `SelectedValue` properties of the two DropDownLists, the edited product's category and supplier columns will default to the current product's values. Upon clicking Update, the `CategoryID` and `SupplierID` values of the selected drop-down list item will be passed to the `UpdateProduct`

method. Figure 11 shows the tutorial after the databinding statements have been added; note how the selected drop-down list items for Chef Anton's Cajun Seasoning are correctly Condiment and New Orleans Cajun Delights.

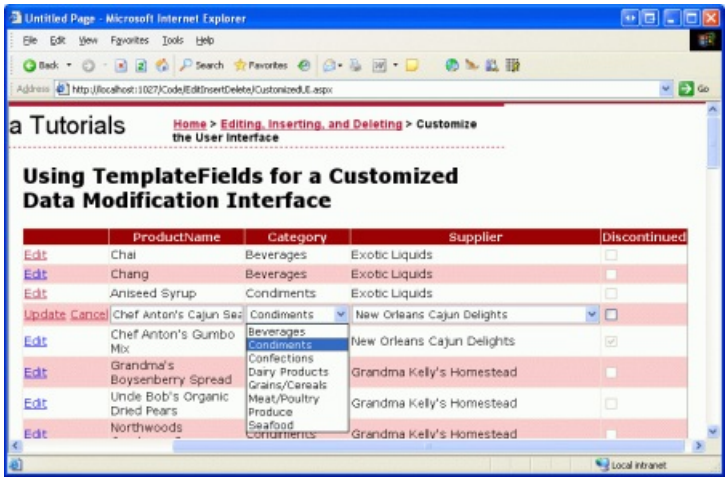


Figure 11: The Edited Product's Current Category and Supplier Values are Selected by Default (Click to view full-size image)

Handling `NULL` Values

The `CategoryID` and `SupplierID` columns in the `Products` table can be `NULL`, yet the DropDownLists in the `EditItemTemplate` s don't include a list item to represent a `NULL` value. This has two consequences:

- User cannot use our interface to change a product's category or supplier from a non-`NULL` value to a `NULL` one
- If a product has a `NULL` `CategoryID` or `SupplierID`, clicking the Edit button will result in an exception. This is because the `NULL` value returned by `CategoryID` (or `SupplierID`) in the `Bind()` statement does not map to a value in the DropDownList (the DropDownList throws an exception when its `SelectedValue` property is set to a value *not* in its collection of list items).

In order to support `NULL` `CategoryID` and `SupplierID` values, we need to add another `ListItem` to each DropDownList to represent the `NULL` value. In the Master/Detail Filtering With a DropDownList tutorial, we saw how to add an additional `ListItem` to a databound DropDownList, which involved setting the DropDownList's `AppendDataBoundItems` property to `True` and manually adding the additional `ListItem`. In that previous tutorial, however, we added a `ListItem` with a `Value` of `-1`. The databinding logic in ASP.NET, however, will automatically convert a blank string to a `NULL` value and vice-a-versa. Therefore, for this tutorial we want the `ListItem`'s `Value` to be an empty string.

Start by setting both DropDownLists' `AppendDataBoundItems` property to `True`. Next, add the `NULL` `ListItem` by adding the following `<asp:ListItem>` element to each DropDownList so that the declarative markup looks like:

```
<asp:DropDownList ID="Categories" runat="server"
    DataSourceID="CategoriesDataSource" DataTextField="CategoryName"
    DataValueField="CategoryID" SelectedValue='<%# Bind("CategoryID") %>'
    AppendDataBoundItems="True">
    <asp:ListItem Value="">(None)</asp:ListItem>
</asp:DropDownList>
```

I've chosen to use "(None)" as the Text value for this `ListItem`, but you can change it to also be a blank string if you'd like.

NOTE

As we saw in the *Master/Detail Filtering With a DropDownList* tutorial, `ListItem` s can be added to a DropDownList through the Designer by clicking on the DropDownList's `Items` property in the Properties window (which will display the `ListItem` Collection Editor). However, be sure to add the `NULL` `ListItem` for this tutorial through the declarative syntax. If you use the `ListItem` Collection Editor, the generated declarative syntax will omit the `Value` setting altogether when assigned a blank string, creating declarative markup like: `<asp:ListItem>(None)</asp:ListItem>`. While this may look harmless, the missing Value causes the DropDownList to use the `Text` property value in its place. That means that if this `NULL` `ListItem` is selected, the value "(None)" will be attempted to be assigned to the `CategoryID`, which will result in an exception. By explicitly setting `Value=""`, a `NULL` value will be assigned to `CategoryID` when the `NULL` `ListItem` is selected.

Repeat these steps for the Suppliers DropDownList.

With this additional `ListItem`, the editing interface can now assign `NULL` values to a Product's `CategoryID` and `SupplierID` fields, as shown in Figure 12.

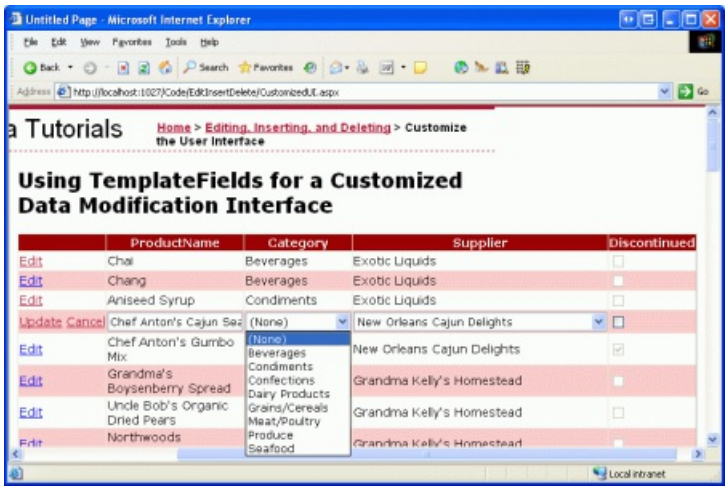


Figure 12: Choose (None) to Assign a `NULL` Value for a Product's Category or Supplier (Click to view full-size image)

Step 4: Using RadioButtons for the Discontinued Status

Currently the products' `Discontinued` data field is expressed using a CheckBoxField, which renders a disabled checkbox for the read-only rows and an enabled checkbox for the row being edited. While this user interface is often suitable, we can customize it if needed using a TemplateField. For this tutorial, let's change the CheckBoxField into a TemplateField that uses a RadioButtonList control with two options "Active" and "Discontinued" from which the user can specify the product's `Discontinued` value.

Start by converting the `Discontinued` CheckBoxField into a TemplateField, which will create a TemplateField with an `ItemTemplate` and `EditItemTemplate`. Both templates include a CheckBox with its `Checked` property bound to the `Discontinued` data field, the only difference between the two being that the `ItemTemplate`'s CheckBox's `Enabled` property is set to `False`.

Replace the CheckBox in both the `ItemTemplate` and `EditItemTemplate` with a RadioButtonList control, setting both RadioButtonLists' `ID` properties to `DiscontinuedChoice`. Next, indicate that the RadioButtonLists should each contain two radio buttons, one labeled "Active" with a value of "False" and one labeled "Discontinued" with a value of "True". To accomplish this you can either enter the `<asp:ListItem>` elements in directly through the declarative syntax or use the `ListItem` Collection Editor from the Designer. Figure 13 shows the `ListItem` Collection Editor after the two radio button options have been specified.

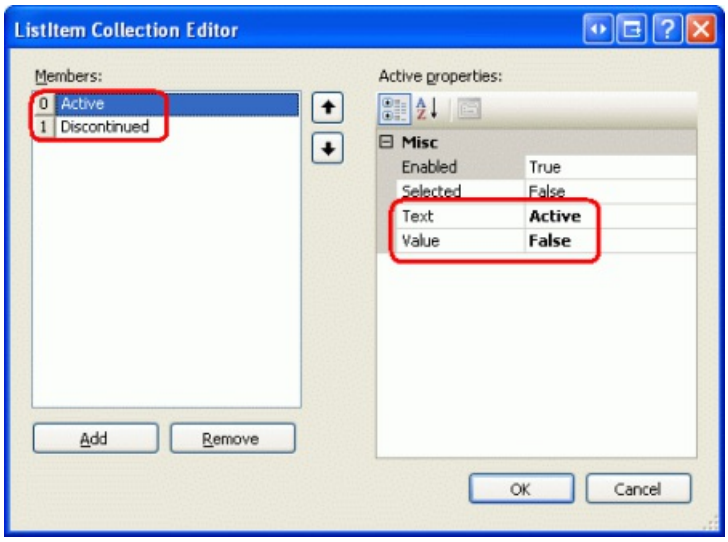


Figure 13: Add Active and Discontinued Options to the RadioButtonList (Click to view full-size image)

Since the RadioButtonList in the `ItemTemplate` shouldn't be editable, set its `Enabled` property to `False`, leaving the `Enabled` property to `True` (the default) for the RadioButtonList in the `EditItemTemplate`. This will make the radio buttons in the non-edited row as read-only, but will allow the user to change the RadioButton values for the edited row.

We still need to assign the RadioButtonList controls' `SelectedValue` properties so that the appropriate radio button is selected based upon the product's `Discontinued` data field. As with the DropDownLists examined earlier in this tutorial, this databinding syntax can either be added directly into the declarative markup or through the Edit DataBindings link in the RadioButtonLists' smart tags.

After adding the two RadioButtonLists and configuring them, the `Discontinued` TemplateField's declarative markup should look like:

```
<asp:TemplateField HeaderText="Discontinued" SortExpression="Discontinued">
    <ItemTemplate>
        <asp:RadioButtonList ID="DiscontinuedChoice" runat="server"
          Enabled="False" SelectedValue='<%# Bind("Discontinued") %>'>
            <asp:ListItem Value="False">Active</asp:ListItem>
            <asp:ListItem Value="True">Discontinued</asp:ListItem>
        </asp:RadioButtonList>
    </ItemTemplate>
    <EditItemTemplate>
        <asp:RadioButtonList ID="DiscontinuedChoice" runat="server"
            SelectedValue='<%# Bind("Discontinued") %>'>
            <asp:ListItem Value="False">Active</asp:ListItem>
            <asp:ListItem Value="True">Discontinued</asp:ListItem>
        </asp:RadioButtonList>
    </EditItemTemplate>
</asp:TemplateField>
```

With these changes, the `Discontinued` column has been transformed from a list of checkboxes to a list of radio button pairs (see Figure 14). When editing a product, the appropriate radio button is selected and the product's discontinued status can be updated by selecting the other radio button and clicking Update.

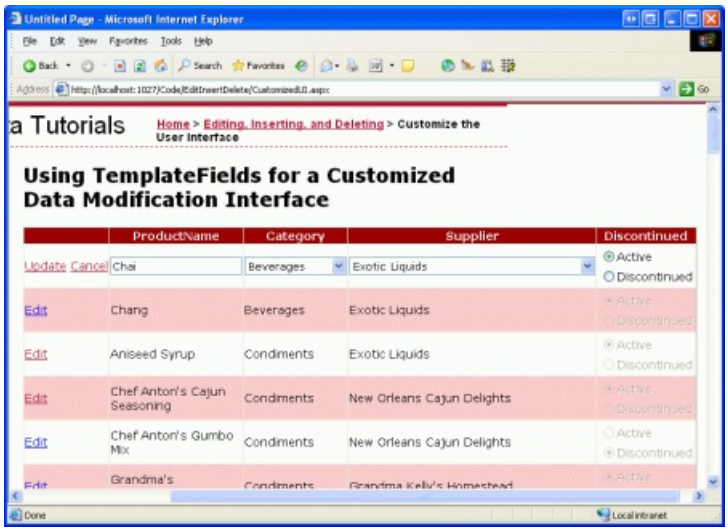


Figure 14: The Discontinued CheckBoxes Have Been Replaced by Radio Button Pairs (Click to view full-size image)

> **NOTE**
>
> Since the `Discontinued` column in the `Products` database cannot have `NULL` values, we do not need to worry about capturing `NULL` information in the interface. If, however, `Discontinued` column could contain `NULL` values we'd want to add a third radio button to the list with its `Value` set to an empty string (`Value=""`), just like with the category and supplier DropDownLists.

Summary

While the BoundField and CheckBoxField automatically render read-only, editing, and inserting interfaces, they lack the ability for customization. Often, though, we'll need to customize the editing or inserting interface, perhaps adding validation controls (as we saw in the preceding tutorial) or by customizing the data collection user interface (as we saw in this tutorial). Customizing the interface with a TemplateField can be summed up in the following steps:

1. Add a TemplateField or convert an existing BoundField or CheckBoxField into a TemplateField
2. Augment the interface as needed
3. Bind the appropriate data fields to the newly added Web controls using two-way databinding

In addition to using the built-in ASP.NET Web controls, you can also customize the templates of a TemplateField with custom, compiled server controls and User Controls.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Implementing Optimistic Concurrency (VB)

2/20/2020 • 39 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> For a web application that allows multiple users to edit data, there is the risk that two users may be editing the same data at the same time. In this tutorial we'll implement optimistic concurrency control to handle this risk.

Introduction

For web applications that only allow users to view data, or for those that include only a single user who can modify data, there's no threat of two concurrent users accidentally overwriting one another's changes. For web applications that allow multiple users to update or delete data, however, there's the potential for one user's modifications to clash with another concurrent user's. Without any concurrency policy in place, when two users are simultaneously editing a single record, the user who commits her changes last will override the changes made by the first.

For example, imagine that two users, Jisun and Sam, were both visiting a page in our application that allowed visitors to update and delete the products through a GridView control. Both click the Edit button in the GridView around the same time. Jisun changes the product name to "Chai Tea" and clicks the Update button. The net result is an `UPDATE` statement that is sent to the database, which sets *all* of the product's updateable fields (even though Jisun only updated one field, `ProductName`). At this point in time, the database has the values "Chai Tea," the category Beverages, the supplier Exotic Liquids, and so on for this particular product. However, the GridView on Sam's screen still shows the product name in the editable GridView row as "Chai". A few seconds after Jisun's changes have been committed, Sam updates the category to Condiments and clicks Update. This results in an `UPDATE` statement sent to the database that sets the product name to "Chai," the `CategoryID` to the corresponding Beverages category ID, and so on. Jisun's changes to the product name have been overwritten. Figure 1 graphically depicts this series of events.

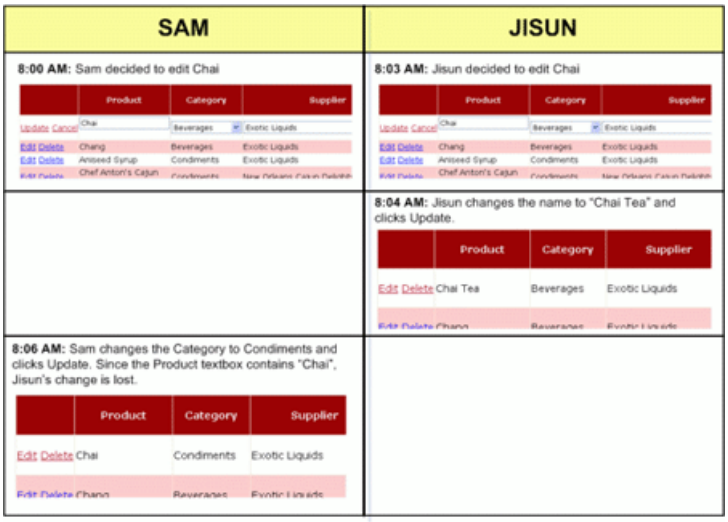


Figure 1: When Two Users Simultaneously Update a Record There s Potential for One User 's Changes to Overwrite the Other 's (Click to view full-size image)

Similarly, when two users are visiting a page, one user might be in the midst of updating a record when it is

deleted by another user. Or, between when a user loads a page and when they click the Delete button, another user may have modified the contents of that record.

There are three concurrency control strategies available:

- **Do Nothing** -if concurrent users are modifying the same record, let the last commit win (the default behavior)
- **Optimistic Concurrency** - assume that while there may be concurrency conflicts every now and then, the vast majority of the time such conflicts won't arise; therefore, if a conflict does arise, simply inform the user that their changes can't be saved because another user has modified the same data
- **Pessimistic Concurrency** - assume that concurrency conflicts are commonplace and that users won't tolerate being told their changes weren't saved due to another user's concurrent activity; therefore, when one user starts updating a record, lock it, thereby preventing any other users from editing or deleting that record until the user commits their modifications

All of our tutorials thus far have used the default concurrency resolution strategy - namely, we've let the last write win. In this tutorial we'll examine how to implement optimistic concurrency control.

> **NOTE**
>
> We won't look at pessimistic concurrency examples in this tutorial series. Pessimistic concurrency is rarely used because such locks, if not properly relinquished, can prevent other users from updating data. For example, if a user locks a record for editing and then leaves for the day before unlocking it, no other user will be able to update that record until the original user returns and completes his update. Therefore, in situations where pessimistic concurrency is used, there's typically a timeout that, if reached, cancels the lock. Ticket sales websites, which lock a particular seating location for short period while the user completes the order process, is an example of pessimistic concurrency control.

Step 1: Looking at How Optimistic Concurrency is Implemented

Optimistic concurrency control works by ensuring that the record being updated or deleted has the same values as it did when the updating or deleting process started. For example, when clicking the Edit button in an editable GridView, the record's values are read from the database and displayed in TextBoxes and other Web controls. These original values are saved by the GridView. Later, after the user makes her changes and clicks the Update button, the original values plus the new values are sent to the Business Logic Layer, and then down to the Data Access Layer. The Data Access Layer must issue a SQL statement that will only update the record if the original values that the user started editing are identical to the values still in the database. Figure 2 depicts this sequence of events.

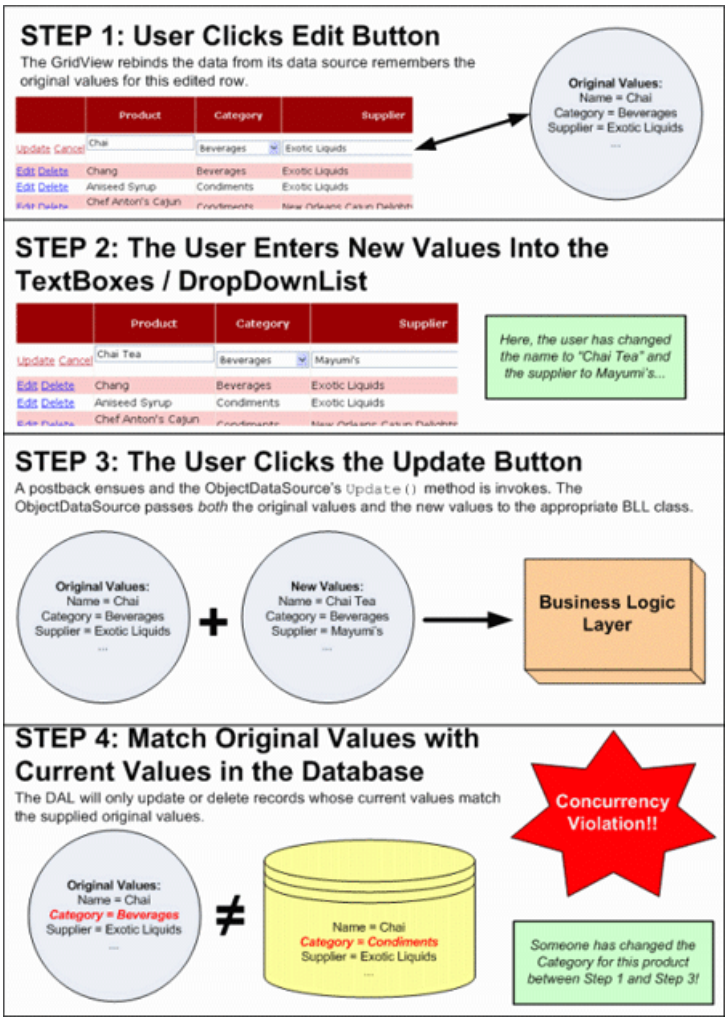


Figure 2: For the Update or Delete to Succeed, the Original Values Must Be Equal to the Current Database Values (Click to view full-size image)

There are various approaches to implementing optimistic concurrency (see Peter A. Bromberg's Optimistic Concurrency Updating Logic for a brief look at a number of options). The ADO.NET Typed DataSet provides one implementation that can be configured with just the tick of a checkbox. Enabling optimistic concurrency for a TableAdapter in the Typed DataSet augments the TableAdapter's `UPDATE` and `DELETE` statements to include a comparison of all of the original values in the `WHERE` clause. The following `UPDATE` statement, for example, updates the name and price of a product only if the current database values are equal to the values that were originally retrieved when updating the record in the GridView. The `@ProductName` and `@UnitPrice` parameters contain the new values entered by the user, whereas `@original_ProductName` and `@original_UnitPrice` contain the values that were originally loaded into the GridView when the Edit button was clicked:

```
UPDATE Products SET
    ProductName = @ProductName,
    UnitPrice = @UnitPrice
WHERE
    ProductID = @original_ProductID AND
    ProductName = @original_ProductName AND
    UnitPrice = @original_UnitPrice
```

NOTE

This `UPDATE` statement has been simplified for readability. In practice, the `UnitPrice` check in the `WHERE` clause would be more involved since `UnitPrice` can contain `NULL` s and checking if `NULL = NULL` always returns False (instead you must use `IS NULL`).

In addition to using a different underlying `UPDATE` statement, configuring a TableAdapter to use optimistic concurrency also modifies the signature of its DB direct methods. Recall from our first tutorial, *Creating a Data Access Layer*, that DB direct methods were those that accepts a list of scalar values as input parameters (rather than a strongly-typed DataRow or DataTable instance). When using optimistic concurrency, the DB direct `Update()` and `Delete()` methods include input parameters for the original values as well. Moreover, the code in the BLL for using the batch update pattern (the `Update()` method overloads that accept DataRows and DataTables rather than scalar values) must be changed as well.

Rather than extend our existing DAL's TableAdapters to use optimistic concurrency (which would necessitate changing the BLL to accommodate), let's instead create a new Typed DataSet named `NorthwindOptimisticConcurrency`, to which we'll add a `Products` TableAdapter that uses optimistic concurrency. Following that, we'll create a `ProductsOptimisticConcurrencyBLL` Business Logic Layer class that has the appropriate modifications to support the optimistic concurrency DAL. Once this groundwork has been laid, we'll be ready to create the ASP.NET page.

Step 2: Creating a Data Access Layer That Supports Optimistic Concurrency

To create a new Typed DataSet, right-click on the `DAL` folder within the `App_Code` folder and add a new DataSet named `NorthwindOptimisticConcurrency`. As we saw in the first tutorial, doing so will add a new TableAdapter to the Typed DataSet, automatically launching the TableAdapter Configuration Wizard. In the first screen, we're prompted to specify the database to connect to - connect to the same Northwind database using the `NORTHWNDConnectionString` setting from `Web.config`.

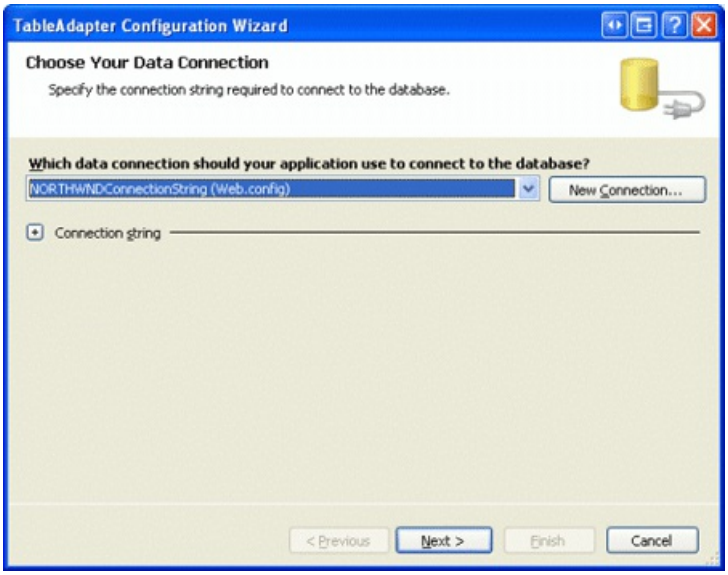


Figure 3: Connect to the Same Northwind Database (Click to view full-size image)

Next, we are prompted as to how to query the data: through an ad-hoc SQL statement, a new stored procedure, or an existing stored procedure. Since we used ad-hoc SQL queries in our original DAL, use this option here as well.

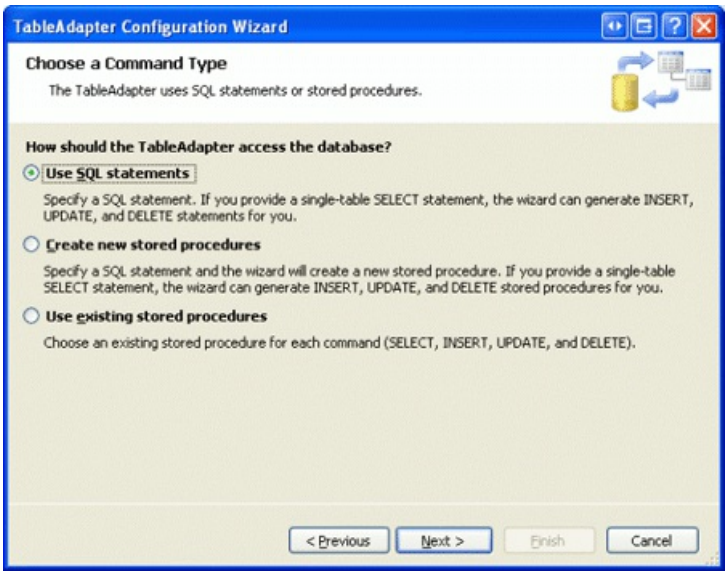


Figure 4: Specify the Data to Retrieve Using an Ad-Hoc SQL Statement (Click to view full-size image)

On the following screen, enter the SQL query to use to retrieve the product information. Let's use the exact same SQL query used for the `Products` TableAdapter from our original DAL, which returns all of the `Product` columns along with the product's supplier and category names:

```
SELECT   ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit,
           UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued,
           (SELECT CategoryName FROM Categories
              WHERE Categories.CategoryID = Products.CategoryID)
              as CategoryName,
           (SELECT CompanyName FROM Suppliers
              WHERE Suppliers.SupplierID = Products.SupplierID)
              as SupplierName
FROM     Products
```

Figure 5: Use the Same SQL Query from the `Products` TableAdapter in the Original DAL (Click to view full-size image)

Before moving onto the next screen, click the Advanced Options button. To have this TableAdapter employ optimistic concurrency control, simply check the "Use optimistic concurrency" checkbox.

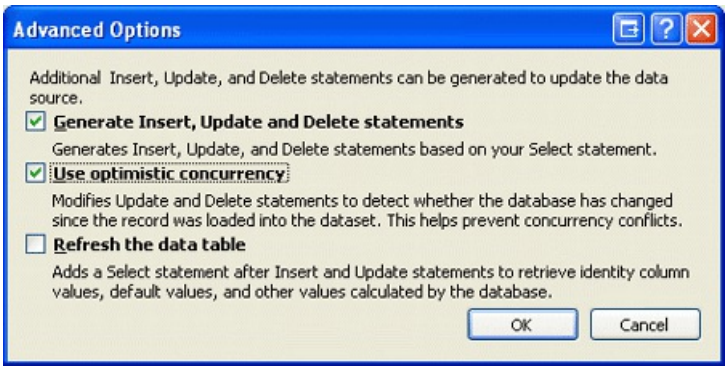


Figure 6: Enable Optimistic Concurrency Control by Checking the "Use optimistic concurrency" CheckBox (Click to view full-size image)

Lastly, indicate that the TableAdapter should use the data access patterns that both fill a DataTable and return a DataTable; also indicate that the DB direct methods should be created. Change the method name for the Return a DataTable pattern from GetData to GetProducts, so as to mirror the naming conventions we used in our original DAL.

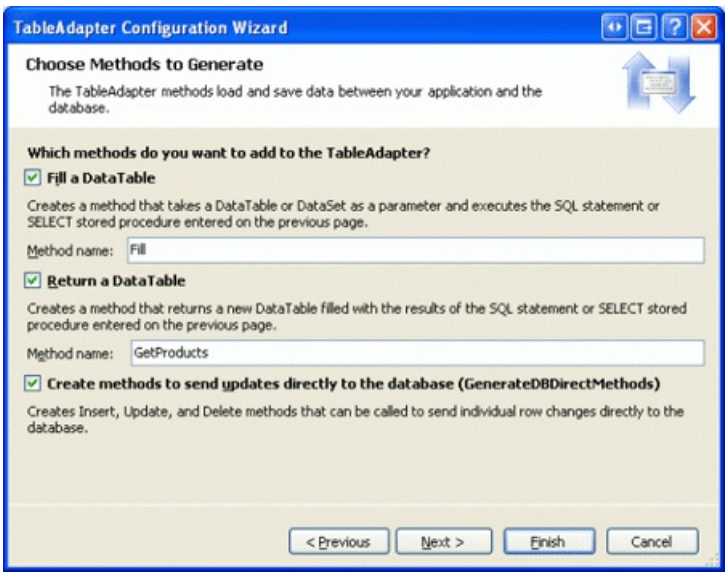


Figure 7: Have the TableAdapter Utilize All Data Access Patterns (Click to view full-size image)

After completing the wizard, the DataSet Designer will include a strongly-typed `Products` DataTable and TableAdapter. Take a moment to rename the DataTable from `Products` to `ProductsOptimisticConcurrency`, which you can do by right-clicking on the DataTable's title bar and choosing Rename from the context menu.

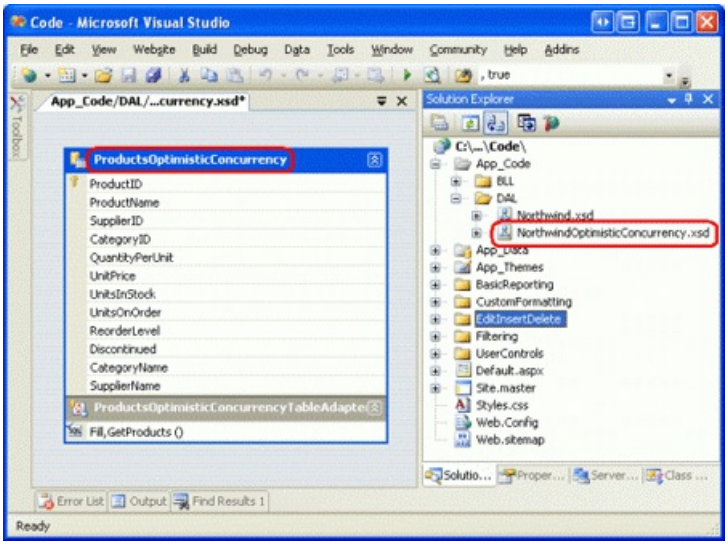


Figure 8: A DataTable and TableAdapter Have Been Added to the Typed DataSet (Click to view full-size image)

To see the differences between the `UPDATE` and `DELETE` queries between the `ProductsOptimisticConcurrency` TableAdapter (which uses optimistic concurrency) and the Products TableAdapter (which doesn't), click on the TableAdapter and go to the Properties window. In the `DeleteCommand` and `UpdateCommand` properties' `CommandText` subproperties you can see the actual SQL syntax that is sent to the database when the DAL's update or delete-related methods are invoked. For the `ProductsOptimisticConcurrency` TableAdapter the `DELETE` statement used is:

```
DELETE FROM [Products]
    WHERE (([ProductID] = @Original_ProductID)
    AND ([ProductName] = @Original_ProductName)
    AND ((@IsNull_SupplierID = 1 AND [SupplierID] IS NULL)
       OR ([SupplierID] = @Original_SupplierID))
    AND ((@IsNull_CategoryID = 1 AND [CategoryID] IS NULL)
       OR ([CategoryID] = @Original_CategoryID))
    AND ((@IsNull_QuantityPerUnit = 1 AND [QuantityPerUnit] IS NULL)
       OR ([QuantityPerUnit] = @Original_QuantityPerUnit))
    AND ((@IsNull_UnitPrice = 1 AND [UnitPrice] IS NULL)
       OR ([UnitPrice] = @Original_UnitPrice))
    AND ((@IsNull_UnitsInStock = 1 AND [UnitsInStock] IS NULL)
       OR ([UnitsInStock] = @Original_UnitsInStock))
    AND ((@IsNull_UnitsOnOrder = 1 AND [UnitsOnOrder] IS NULL)
       OR ([UnitsOnOrder] = @Original_UnitsOnOrder))
    AND ((@IsNull_ReorderLevel = 1 AND [ReorderLevel] IS NULL)
       OR ([ReorderLevel] = @Original_ReorderLevel))
    AND ([Discontinued] = @Original_Discontinued))
```

Whereas the `DELETE` statement for the Product TableAdapter in our original DAL is the much simpler:

```
DELETE FROM [Products] WHERE (([ProductID] = @Original_ProductID))
```

As you can see, the `WHERE` clause in the `DELETE` statement for the TableAdapter that uses optimistic concurrency includes a comparison between each of the `Product` table's existing column values and the original values at the time the GridView (or DetailsView or FormView) was last populated. Since all fields other than `ProductID`, `ProductName`, and `Discontinued` can have `NULL` values, additional parameters and checks are included to correctly compare `NULL` values in the `WHERE` clause.

We won't be adding any additional DataTables to the optimistic concurrency-enabled DataSet for this tutorial, as our ASP.NET page will only provide updating and deleting product information. However, we do still need to add the `GetProductByProductID(productID)` method to the `ProductsOptimisticConcurrency` TableAdapter.

To accomplish this, right-click on the TableAdapter's title bar (the area right above the `Fill` and `GetProducts` method names) and choose Add Query from the context menu. This will launch the TableAdapter Query Configuration Wizard. As with our TableAdapter's initial configuration, opt to create the `GetProductByProductID(productID)` method using an ad-hoc SQL statement (see Figure 4). Since the `GetProductByProductID(productID)` method returns information about a particular product, indicate that this query is a `SELECT` query type that returns rows.

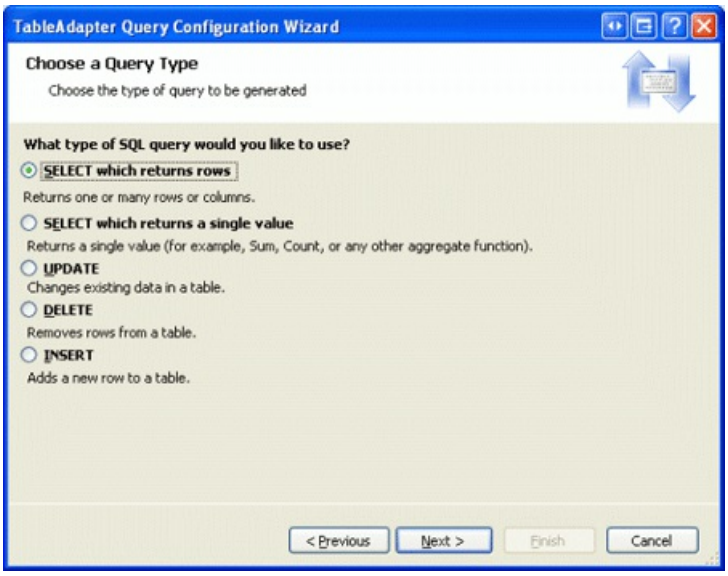


Figure 9: Mark the Query Type as a "`SELECT` which returns rows" (Click to view full-size image)

On the next screen we're prompted for the SQL query to use, with the TableAdapter's default query pre-loaded. Augment the existing query to include the clause `WHERE ProductID = @ProductID`, as shown in Figure 10.

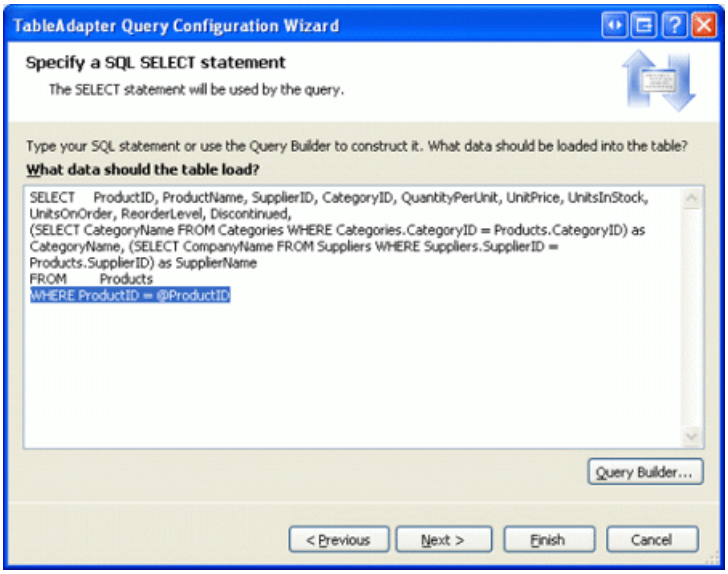


Figure 10: Add a `WHERE` Clause to the Pre-Loaded Query to Return a Specific Product Record (Click to view full-size image)

Finally, change the generated method names to `FillByProductID` and `GetProductByProductID`.

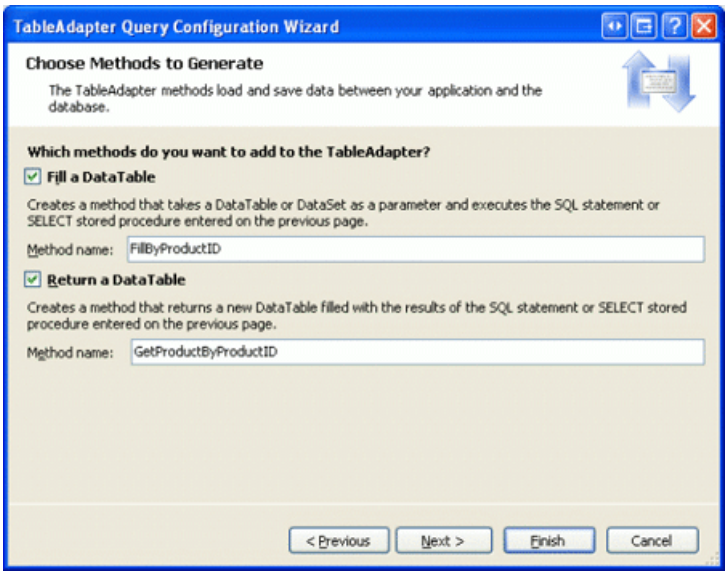


Figure 11: Rename the Methods to `FillByProductID` and `GetProductByProductID` (Click to view full-size image)

With this wizard complete, the TableAdapter now contains two methods for retrieving data: `GetProducts()`, which returns *all* products; and `GetProductByProductID(productID)`, which returns the specified product.

Step 3: Creating a Business Logic Layer for the Optimistic Concurrency-Enabled DAL

Our existing `ProductsBLL` class has examples of using both the batch update and DB direct patterns. The `AddProduct` method and `UpdateProduct` overloads both use the batch update pattern, passing in a `ProductRow` instance to the TableAdapter's Update method. The `DeleteProduct` method, on the other hand, uses the DB direct pattern, calling the TableAdapter's `Delete(productID)` method.

With the new `ProductsOptimisticConcurrency` TableAdapter, the DB direct methods now require that the original values also be passed in. For example, the `Delete` method now expects ten input parameters: the original `ProductID`, `ProductName`, `SupplierID`, `CategoryID`, `QuantityPerUnit`, `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, `ReorderLevel`, and `Discontinued`. It uses these additional input parameters' values in `WHERE` clause of the `DELETE` statement sent to the database, only deleting the specified record if the database's current values map up to the original ones.

While the method signature for the TableAdapter's `Update` method used in the batch update pattern hasn't changed, the code needed to record the original and new values has. Therefore, rather than attempt to use the optimistic concurrency-enabled DAL with our existing `ProductsBLL` class, let's create a new Business Logic Layer class for working with our new DAL.

Add a class named `ProductsOptimisticConcurrencyBLL` to the `BLL` folder within the `App_Code` folder.

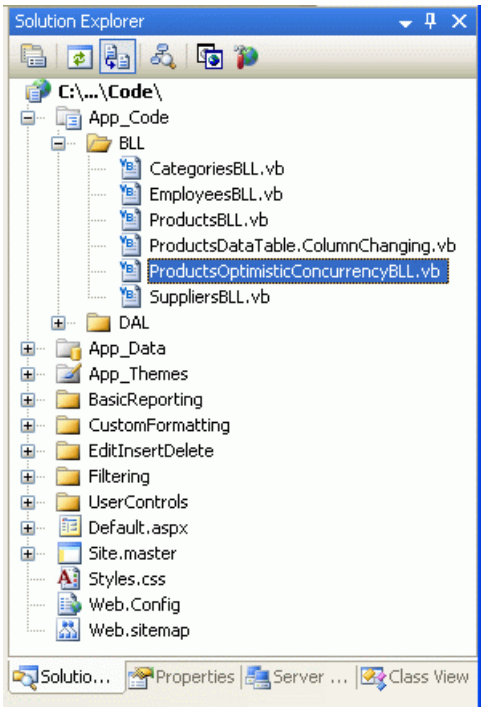


Figure 12: Add the `ProductsOptimisticConcurrencyBLL` Class to the BLL Folder

Next, add the following code to the `ProductsOptimisticConcurrencyBLL` class:

```
Imports NorthwindOptimisticConcurrencyTableAdapters
<System.ComponentModel.DataObject()> _
Public Class ProductsOptimisticConcurrencyBLL
    Private _productsAdapter As ProductsOptimisticConcurrencyTableAdapter = Nothing
    Protected ReadOnly Property Adapter() As ProductsOptimisticConcurrencyTableAdapter
        Get
            If _productsAdapter Is Nothing Then
                _productsAdapter = New ProductsOptimisticConcurrencyTableAdapter()
            End If
            Return _productsAdapter
        End Get
    End Property
    <System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Select, True)> _
    Public Function GetProducts() As _
        NorthwindOptimisticConcurrency.ProductsOptimisticConcurrencyDataTable
        Return Adapter.GetProducts()
    End Function
End Class
```

Note the using `NorthwindOptimisticConcurrencyTableAdapters` statement above the start of the class declaration. The `NorthwindOptimisticConcurrencyTableAdapters` namespace contains the `ProductsOptimisticConcurrencyTableAdapter` class, which provides the DAL's methods. Also before the class declaration you'll find the `System.ComponentModel.DataObject` attribute, which instructs Visual Studio to include this class in the ObjectDataSource wizard's drop-down list.

The `ProductsOptimisticConcurrencyBLL`'s `Adapter` property provides quick access to an instance of the `ProductsOptimisticConcurrencyTableAdapter` class, and follows the pattern used in our original BLL classes (`ProductsBLL`, `CategoriesBLL`, and so on). Finally, the `GetProducts()` method simply calls down into the DAL's `GetProducts()` method and returns a `ProductsOptimisticConcurrencyDataTable` object populated with a `ProductsOptimisticConcurrencyRow` instance for each product record in the database.

Deleting a Product Using the DB Direct Pattern with Optimistic

Concurrency

When using the DB direct pattern against a DAL that uses optimistic concurrency, the methods must be passed the new and original values. For deleting, there are no new values, so only the original values need be passed in. In our BLL, then, we must accept all of the original parameters as input parameters. Let's have the `DeleteProduct` method in the `ProductsOptimisticConcurrencyBLL` class use the DB direct method. This means that this method needs to take in all ten product data fields as input parameters, and pass these to the DAL, as shown in the following code:

```
<System.ComponentModel.DataObjectMethodAttribute _
(System.ComponentModel.DataObjectMethodType.Delete, True)> _
Public Function DeleteProduct( _
    ByVal original_productID As Integer, ByVal original_productName As String, _
    ByVal original_supplierID As Nullable(Of Integer), _
    ByVal original_categoryID As Nullable(Of Integer), _
    ByVal original_quantityPerUnit As String, _
    ByVal original_unitPrice As Nullable(Of Decimal), _
    ByVal original_unitsInStock As Nullable(Of Short), _
    ByVal original_unitsOnOrder As Nullable(Of Short), _
    ByVal original_reorderLevel As Nullable(Of Short), _
    ByVal original_discontinued As Boolean) _
    As Boolean
    Dim rowsAffected As Integer = Adapter.Delete(
                                    original_productID, _
                                    original_productName, _
                                    original_supplierID, _
                                    original_categoryID, _
                                    original_quantityPerUnit, _
                                    original_unitPrice, _
                                    original_unitsInStock, _
                                    original_unitsOnOrder, _
                                    original_reorderLevel, _
                                    original_discontinued)
    ' Return true if precisely one row was deleted, otherwise false
    Return rowsAffected = 1
End Function
```

If the original values - those values that were last loaded into the GridView (or DetailsView or FormView) - differ from the values in the database when the user clicks the Delete button the `WHERE` clause won't match up with any database record and no records will be affected. Hence, the TableAdapter's `Delete` method will return `0` and the BLL's `DeleteProduct` method will return `false`.

Updating a Product Using the Batch Update Pattern with Optimistic Concurrency

As noted earlier, the TableAdapter's `Update` method for the batch update pattern has the same method signature regardless of whether or not optimistic concurrency is employed. Namely, the `Update` method expects a DataRow, an array of DataRows, a DataTable, or a Typed DataSet. There are no additional input parameters for specifying the original values. This is possible because the DataTable keeps track of the original and modified values for its DataRow(s). When the DAL issues its `UPDATE` statement, the `@original_ColumnName` parameters are populated with the DataRow's original values, whereas the `@ColumnName` parameters are populated with the DataRow's modified values.

In the `ProductsBLL` class (which uses our original, non-optimistic concurrency DAL), when using the batch update pattern to update product information our code performs the following sequence of events:

1. Read the current database product information into a `ProductRow` instance using the TableAdapter's `GetProductByProductID(productID)` method

2. Assign the new values to the `ProductRow` instance from Step 1
3. Call the TableAdapter's `Update` method, passing in the `ProductRow` instance

This sequence of steps, however, won't correctly support optimistic concurrency because the `ProductRow` populated in Step 1 is populated directly from the database, meaning that the original values used by the DataRow are those that currently exist in the database, and not those that were bound to the GridView at the start of the editing process. Instead, when using an optimistic concurrency-enabled DAL, we need to alter the `UpdateProduct` method overloads to use the following steps:

1. Read the current database product information into a `ProductsOptimisticConcurrencyRow` instance using the TableAdapter's `GetProductByProductID(productID)` method
2. Assign the *original* values to the `ProductsOptimisticConcurrencyRow` instance from Step 1
3. Call the `ProductsOptimisticConcurrencyRow` instance's `AcceptChanges()` method, which instructs the DataRow that its current values are the "original" ones
4. Assign the *new* values to the `ProductsOptimisticConcurrencyRow` instance
5. Call the TableAdapter's `Update` method, passing in the `ProductsOptimisticConcurrencyRow` instance

Step 1 reads in all of the current database values for the specified product record. This step is superfluous in the `UpdateProduct` overload that updates *all* of the product columns (as these values are overwritten in Step 2), but is essential for those overloads where only a subset of the column values are passed in as input parameters. Once the original values have been assigned to the `ProductsOptimisticConcurrencyRow` instance, the `AcceptChanges()` method is called, which marks the current DataRow values as the original values to be used in the `@original_ColumnName` parameters in the `UPDATE` statement. Next, the new parameter values are assigned to the `ProductsOptimisticConcurrencyRow` and, finally, the `Update` method is invoked, passing in the DataRow.

The following code shows the `UpdateProduct` overload that accepts all product data fields as input parameters. While not shown here, the `ProductsOptimisticConcurrencyBLL` class included in the download for this tutorial also contains an `UpdateProduct` overload that accepts just the product's name and price as input parameters.

```
Protected Sub AssignAllProductValues( _
    ByVal product As NorthwindOptimisticConcurrency.ProductsOptimisticConcurrencyRow, _
    ByVal productName As String, ByVal supplierID As Nullable(Of Integer), _
    ByVal categoryID As Nullable(Of Integer), ByVal quantityPerUnit As String, _
    ByVal unitPrice As Nullable(Of Decimal), ByVal unitsInStock As Nullable(Of Short), _
    ByVal unitsOnOrder As Nullable(Of Short), ByVal reorderLevel As Nullable(Of Short), _
    ByVal discontinued As Boolean)
    product.ProductName = productName
    If Not supplierID.HasValue Then
        product.SetSupplierIDNull()
    Else
        product.SupplierID = supplierID.Value
    End If
    If Not categoryID.HasValue Then
        product.SetCategoryIDNull()
    Else
        product.CategoryID = categoryID.Value
    End If
    If quantityPerUnit Is Nothing Then
        product.SetQuantityPerUnitNull()
    Else
        product.QuantityPerUnit = quantityPerUnit
    End If
    If Not unitPrice.HasValue Then
        product.SetUnitPriceNull()
    Else
        product.UnitPrice = unitPrice.Value
    End If
    If Not unitsInStock.HasValue Then
        product.SetUnitsInStockNull()
    Else
```

```
        product.UnitsInStock = unitsInStock.Value
    End If
    If Not unitsOnOrder.HasValue Then
        product.SetUnitsOnOrderNull()
    Else
        product.UnitsOnOrder = unitsOnOrder.Value
    End If
    If Not reorderLevel.HasValue Then
        product.SetReorderLevelNull()
    Else
        product.ReorderLevel = reorderLevel.Value
    End If
    product.Discontinued = discontinued
End Sub
<System.ComponentModel.DataObjectMethodAttribute( _
System.ComponentModel.DataObjectMethodType.Update, True)> _
Public Function UpdateProduct(
    ByVal productName As String, ByVal supplierID As Nullable(Of Integer), _
    ByVal categoryID As Nullable(Of Integer), ByVal quantityPerUnit As String, _
    ByVal unitPrice As Nullable(Of Decimal), ByVal unitsInStock As Nullable(Of Short), _
    ByVal unitsOnOrder As Nullable(Of Short), ByVal reorderLevel As Nullable(Of Short), _
    ByVal discontinued As Boolean, ByVal productID As Integer, _
    _
    ByVal original_productName As String, _
    ByVal original_supplierID As Nullable(Of Integer), _
    ByVal original_categoryID As Nullable(Of Integer), _
    ByVal original_quantityPerUnit As String, _
    ByVal original_unitPrice As Nullable(Of Decimal), _
    ByVal original_unitsInStock As Nullable(Of Short), _
    ByVal original_unitsOnOrder As Nullable(Of Short), _
    ByVal original_reorderLevel As Nullable(Of Short), _
    ByVal original_discontinued As Boolean, _
    ByVal original_productID As Integer) _
    As Boolean
    'STEP 1: Read in the current database product information
    Dim products As _
        NorthwindOptimisticConcurrency.ProductsOptimisticConcurrencyDataTable = _
        Adapter.GetProductByProductID(original_productID)
    If products.Count = 0 Then
        ' no matching record found, return false
        Return False
    End If
    Dim product As _
        NorthwindOptimisticConcurrency.ProductsOptimisticConcurrencyRow = products(0)
    'STEP 2: Assign the original values to the product instance
    AssignAllProductValues( _
        product, original_productName, original_supplierID, _
        original_categoryID, original_quantityPerUnit, original_unitPrice, _
        original_unitsInStock, original_unitsOnOrder, original_reorderLevel, _
        original_discontinued)
    'STEP 3: Accept the changes
    product.AcceptChanges()
    'STEP 4: Assign the new values to the product instance
    AssignAllProductValues( _
        product, productName, supplierID, categoryID, quantityPerUnit, unitPrice, _
        unitsInStock, unitsOnOrder, reorderLevel, discontinued)
    'STEP 5: Update the product record
    Dim rowsAffected As Integer = Adapter.Update(product)
    ' Return true if precisely one row was updated, otherwise false
    Return rowsAffected = 1
End Function
```

Step 4: Passing the Original and New Values From the ASP.NET Page to the BLL Methods

With the DAL and BLL complete, all that remains is to create an ASP.NET page that can utilize the optimistic concurrency logic built in to the system. Specifically, the data Web control (the GridView, DetailsView, or FormView) must remember its original values and the ObjectDataSource must pass both sets of values to the Business Logic Layer. Furthermore, the ASP.NET page must be configured to gracefully handle concurrency violations.

Start by opening the `OptimisticConcurrency.aspx` page in the `EditInsertDelete` folder and adding a GridView to the Designer, setting its `ID` property to `ProductsGrid`. From the GridView's smart tag, opt to create a new ObjectDataSource named `ProductsOptimisticConcurrencyDataSource`. Since we want this ObjectDataSource to use the DAL that supports optimistic concurrency, configure it to use the `ProductsOptimisticConcurrencyBLL` object.

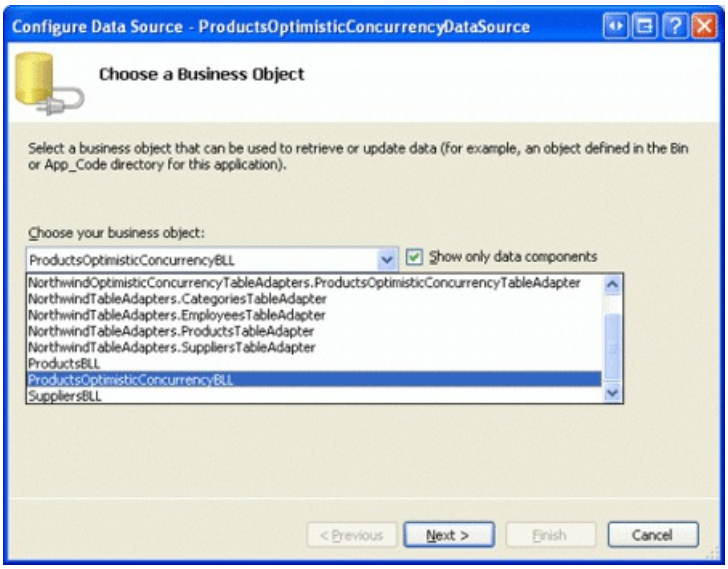


Figure 13: Have the ObjectDataSource Use the `ProductsOptimisticConcurrencyBLL` Object (Click to view full-size image)

Choose the `GetProducts`, `UpdateProduct`, and `DeleteProduct` methods from drop-down lists in the wizard. For the UpdateProduct method, use the overload that accepts all of the product's data fields.

Configuring the ObjectDataSource Control's Properties

After completing the wizard, the ObjectDataSource's declarative markup should look like the following:

```
<asp:ObjectDataSource ID="ProductsOptimisticConcurrencyDataSource" runat="server"
    DeleteMethod="DeleteProduct" OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsOptimisticConcurrencyBLL"
    UpdateMethod="UpdateProduct">
    <DeleteParameters>
        <asp:Parameter Name="original_productID" Type="Int32" />
        <asp:Parameter Name="original_productName" Type="String" />
        <asp:Parameter Name="original_supplierID" Type="Int32" />
        <asp:Parameter Name="original_categoryID" Type="Int32" />
        <asp:Parameter Name="original_quantityPerUnit" Type="String" />
        <asp:Parameter Name="original_unitPrice" Type="Decimal" />
        <asp:Parameter Name="original_unitsInStock" Type="Int16" />
        <asp:Parameter Name="original_unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="original_reorderLevel" Type="Int16" />
        <asp:Parameter Name="original_discontinued" Type="Boolean" />
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="unitsInStock" Type="Int16" />
        <asp:Parameter Name="unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="reorderLevel" Type="Int16" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
        <asp:Parameter Name="productID" Type="Int32" />
        <asp:Parameter Name="original_productName" Type="String" />
        <asp:Parameter Name="original_supplierID" Type="Int32" />
        <asp:Parameter Name="original_categoryID" Type="Int32" />
        <asp:Parameter Name="original_quantityPerUnit" Type="String" />
        <asp:Parameter Name="original_unitPrice" Type="Decimal" />
        <asp:Parameter Name="original_unitsInStock" Type="Int16" />
        <asp:Parameter Name="original_unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="original_reorderLevel" Type="Int16" />
        <asp:Parameter Name="original_discontinued" Type="Boolean" />
        <asp:Parameter Name="original_productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

As you can see, the `DeleteParameters` collection contains a `Parameter` instance for each of the ten input parameters in the `ProductsOptimisticConcurrencyBLL` class's `DeleteProduct` method. Likewise, the `UpdateParameters` collection contains a `Parameter` instance for each of the input parameters in `UpdateProduct`.

For those previous tutorials that involved data modification, we'd remove the ObjectDataSource's `OldValuesParameterFormatString` property at this point, since this property indicates that the BLL method expects the old (or original) values to be passed in as well as the new values. Furthermore, this property value indicates the input parameter names for the original values. Since we are passing in the original values into the BLL, do *not* remove this property.

> **NOTE**
> The value of the `OldValuesParameterFormatString` property must map to the input parameter names in the BLL that expect the original values. Since we named these parameters `original_productName`, `original_supplierID`, and so on, you can leave the `OldValuesParameterFormatString` property value as `original_{0}`. If, however, the BLL methods' input parameters had names like `old_productName`, `old_supplierID`, and so on, you'd need to update the `OldValuesParameterFormatString` property to `old_{0}`.

There's one final property setting that needs to be made in order for the ObjectDataSource to correctly pass the original values to the BLL methods. The ObjectDataSource has a ConflictDetection property that can be assigned

to one of two values:

- `OverwriteChanges` - the default value; does not send the original values to the BLL methods' original input parameters
- `CompareAllValues` - does send the original values to the BLL methods; choose this option when using optimistic concurrency

Take a moment to set the `ConflictDetection` property to `CompareAllValues`.

Configuring the GridView's Properties and Fields

With the ObjectDataSource's properties properly configured, let's turn our attention to setting up the GridView. First, since we want the GridView to support editing and deleting, click the Enable Editing and Enable Deleting checkboxes from the GridView's smart tag. This will add a CommandField whose `ShowEditButton` and `ShowDeleteButton` are both set to `true`.

When bound to the `ProductsOptimisticConcurrencyDataSource` ObjectDataSource, the GridView contains a field for each of the product's data fields. While such a GridView can be edited, the user experience is anything but acceptable. The `CategoryID` and `SupplierID` BoundFields will render as TextBoxes, requiring the user to enter the appropriate category and supplier as ID numbers. There will be no formatting for the numeric fields and no validation controls to ensure that the product's name has been supplied and that the unit price, units in stock, units on order, and reorder level values are both proper numeric values and are greater than or equal to zero.

As we discussed in the *Adding Validation Controls to the Editing and Inserting Interfaces* and *Customizing the Data Modification Interface* tutorials, the user interface can be customized by replacing the BoundFields with TemplateFields. I've modified this GridView and its editing interface in the following ways:

- Removed the `ProductID`, `SupplierName`, and `CategoryName` BoundFields
- Converted the `ProductName` BoundField to a TemplateField and added a RequiredFieldValidation control.
- Converted the `CategoryID` and `SupplierID` BoundFields to TemplateFields, and adjusted the editing interface to use DropDownLists rather than TextBoxes. In these TemplateFields' `ItemTemplates`, the `CategoryName` and `SupplierName` data fields are displayed.
- Converted the `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, and `ReorderLevel` BoundFields to TemplateFields and added CompareValidator controls.

Since we've already examined how to accomplish these tasks in previous tutorials, I'll just list the final declarative syntax here and leave the implementation as practice.

```
<asp:GridView ID="ProductsGrid" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsOptimisticConcurrencyDataSource"
    OnRowUpdated="ProductsGrid_RowUpdated">
    <Columns>
        <asp:CommandField ShowDeleteButton="True" ShowEditButton="True" />
        <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
            <EditItemTemplate>
                <asp:TextBox ID="EditProductName" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:TextBox>
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                    ControlToValidate="EditProductName"
                    ErrorMessage="You must enter a product name."
                    runat="server">*</asp:RequiredFieldValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Category" SortExpression="CategoryName">
            <EditItemTemplate>
```

```
            <asp:DropDownList ID="EditCategoryID" runat="server"
                DataSourceID="CategoriesDataSource" AppendDataBoundItems="true"
                DataTextField="CategoryName" DataValueField="CategoryID"
                SelectedValue='<%# Bind("CategoryID") %>'>
                <asp:ListItem Value=">(None)</asp:ListItem>
            </asp:DropDownList><asp:ObjectDataSource ID="CategoriesDataSource"
                runat="server" OldValuesParameterFormatString="original_{0}"
                SelectMethod="GetCategories" TypeName="CategoriesBLL">
            </asp:ObjectDataSource>
        </EditItemTemplate>
        <ItemTemplate>
            <asp:Label ID="Label2" runat="server"
                Text='<%# Bind("CategoryName") %>'></asp:Label>
        </ItemTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Supplier" SortExpression="SupplierName">
        <EditItemTemplate>
            <asp:DropDownList ID="EditSuppliersID" runat="server"
                DataSourceID="SuppliersDataSource" AppendDataBoundItems="true"
                DataTextField="CompanyName" DataValueField="SupplierID"
                SelectedValue='<%# Bind("SupplierID") %>'>
                <asp:ListItem Value=">(None)</asp:ListItem>
            </asp:DropDownList><asp:ObjectDataSource ID="SuppliersDataSource"
                runat="server" OldValuesParameterFormatString="original_{0}"
                SelectMethod="GetSuppliers" TypeName="SuppliersBLL">
            </asp:ObjectDataSource>
        </EditItemTemplate>
        <ItemTemplate>
            <asp:Label ID="Label3" runat="server"
                Text='<%# Bind("SupplierName") %>'></asp:Label>
        </ItemTemplate>
    </asp:TemplateField>
    <asp:BoundField DataField="QuantityPerUnit" HeaderText="Qty/Unit"
        SortExpression="QuantityPerUnit" />
    <asp:TemplateField HeaderText="Price" SortExpression="UnitPrice">
        <EditItemTemplate>
            <asp:TextBox ID="EditUnitPrice" runat="server"
                Text='<%# Bind("UnitPrice", "{0:N2}") %>' Columns="8" />
            <asp:CompareValidator ID="CompareValidator1" runat="server"
                ControlToValidate="EditUnitPrice"
                ErrorMessage="Unit price must be a valid currency value without the
                currency symbol and must have a value greater than or equal to zero."
                Operator="GreaterThanEqual" Type="Currency"
                ValueToCompare="0">*</asp:CompareValidator>
        </EditItemTemplate>
        <ItemTemplate>
            <asp:Label ID="Label4" runat="server"
                Text='<%# Bind("UnitPrice", "{0:C}") %>'></asp:Label>
        </ItemTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Units In Stock" SortExpression="UnitsInStock">
        <EditItemTemplate>
            <asp:TextBox ID="EditUnitsInStock" runat="server"
                Text='<%# Bind("UnitsInStock") %>' Columns="6"></asp:TextBox>
            <asp:CompareValidator ID="CompareValidator2" runat="server"
                ControlToValidate="EditUnitsInStock"
                ErrorMessage="Units in stock must be a valid number
                    greater than or equal to zero."
                Operator="GreaterThanEqual" Type="Integer"
                ValueToCompare="0">*</asp:CompareValidator>
        </EditItemTemplate>
        <ItemTemplate>
            <asp:Label ID="Label5" runat="server"
                Text='<%# Bind("UnitsInStock", "{0:N0}") %>'></asp:Label>
        </ItemTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Units On Order" SortExpression="UnitsOnOrder">
        <EditItemTemplate>
            <asp:TextBox ID="EditUnitsOnOrder" runat="server"
```

```
                <asp:TextBox ID="EditUnitsOnOrder" runat="server"
                    Text='<%# Bind("UnitsOnOrder") %>' Columns="6"></asp:TextBox>
                <asp:CompareValidator ID="CompareValidator3" runat="server"
                    ControlToValidate="EditUnitsOnOrder"
                    ErrorMessage="Units on order must be a valid numeric value
                        greater than or equal to zero."
                    Operator="GreaterThanEqual" Type="Integer"
                    ValueToCompare="0">*</asp:CompareValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label6" runat="server"
                    Text='<%# Bind("UnitsOnOrder", "{0:N0}") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Reorder Level" SortExpression="ReorderLevel">
            <EditItemTemplate>
                <asp:TextBox ID="EditReorderLevel" runat="server"
                    Text='<%# Bind("ReorderLevel") %>' Columns="6"></asp:TextBox>
                <asp:CompareValidator ID="CompareValidator4" runat="server"
                    ControlToValidate="EditReorderLevel"
                    ErrorMessage="Reorder level must be a valid numeric value
                        greater than or equal to zero."
                    Operator="GreaterThanEqual" Type="Integer"
                    ValueToCompare="0">*</asp:CompareValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label7" runat="server"
                    Text='<%# Bind("ReorderLevel", "{0:N0}") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
```

We're very close to having a fully-working example. However, there are a few subtleties that will creep up and cause us problems. Additionally, we still need some interface that alerts the user when a concurrency violation has occurred.

> **NOTE**
>
> In order for a data Web control to correctly pass the original values to the ObjectDataSource (which are then passed to the BLL), it's vital that the GridView's `EnableViewState` property is set to `true` (the default). If you disable view state, the original values are lost on postback.

Passing the Correct Original Values to the ObjectDataSource

There are a couple of problems with the way the GridView has been configured. If the ObjectDataSource's `ConflictDetection` property is set to `CompareAllValues` (as is ours), when the ObjectDataSource's `Update()` or `Delete()` methods are invoked by the GridView (or DetailsView or FormView), the ObjectDataSource attempts to copy the GridView's original values into its appropriate `Parameter` instances. Refer back to Figure 2 for a graphical representation of this process.

Specifically, the GridView's original values are assigned the values in the two-way databinding statements each time the data is bound to the GridView. Therefore, it's essential that the required original values all are captured via two-way databinding and that they are provided in a convertible format.

To see why this is important, take a moment to visit our page in a browser. As expected, the GridView lists each product with an Edit and Delete button in the leftmost column.

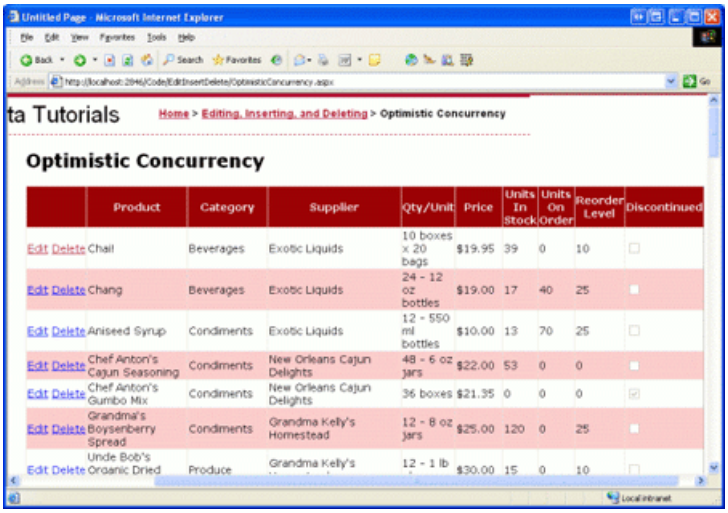


Figure 14: The Products are Listed in a GridView (Click to view full-size image)

If you click the Delete button for any product, a `FormatException` is thrown.

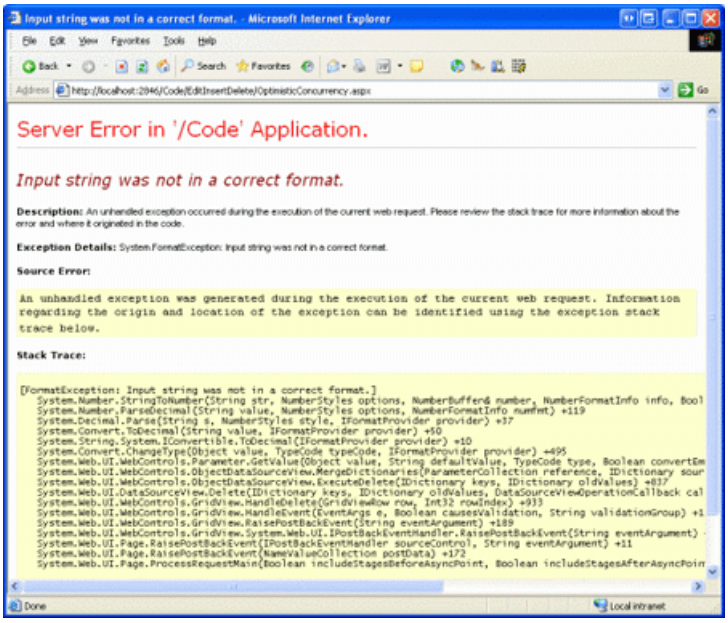


Figure 15: Attempting to Delete Any Product Results in a `FormatException` (Click to view full-size image)

The `FormatException` is raised when the ObjectDataSource attempts to read in the original `UnitPrice` value. Since the `ItemTemplate` has the `UnitPrice` formatted as a currency (`<%# Bind("UnitPrice", "{0:C}") %>`), it includes a currency symbol, like $19.95. The `FormatException` occurs as the ObjectDataSource attempts to convert this string into a `decimal`. To circumvent this problem, we have a number of options:

- Remove the currency formatting from the `ItemTemplate`. That is, instead of using `<%# Bind("UnitPrice", "{0:C}") %>`, simply use `<%# Bind("UnitPrice") %>`. The downside of this is that the price is no longer formatted.
- Display the `UnitPrice` formatted as a currency in the `ItemTemplate`, but use the `Eval` keyword to accomplish this. Recall that `Eval` performs one-way databinding. We still need to provide the `UnitPrice` value for the original values, so we'll still need a two-way databinding statement in the `ItemTemplate`, but this can be placed in a Label Web control whose `Visible` property is set to `false`. We could use the following markup in the ItemTemplate:

```
<ItemTemplate>
    <asp:Label ID="DummyUnitPrice" runat="server"
        Text='<%# Bind("UnitPrice") %>' Visible="false"></asp:Label>
    <asp:Label ID="Label4" runat="server"
        Text='<%# Eval("UnitPrice", "{0:C}") %>'></asp:Label>
</ItemTemplate>
```

- Remove the currency formatting from the `ItemTemplate`, using `<%# Bind("UnitPrice") %>`. In the GridView's `RowDataBound` event handler, programmatically access the Label Web control within which the `UnitPrice` value is displayed and set its `Text` property to the formatted version.
- Leave the `UnitPrice` formatted as a currency. In the GridView's `RowDeleting` event handler, replace the existing original `UnitPrice` value ($19.95) with an actual decimal value using `Decimal.Parse`. We saw how to accomplish something similar in the `RowUpdating` event handler in the *Handling BLL- and DAL-Level Exceptions in an ASP.NET Page* tutorial.

For my example I chose to go with the second approach, adding a hidden Label Web control whose `Text` property is two-way data bound to the unformatted `UnitPrice` value.

After solving this problem, try clicking the Delete button for any product again. This time you'll get an `InvalidOperationException` when the ObjectDataSource attempts to invoke the BLL's `UpdateProduct` method.

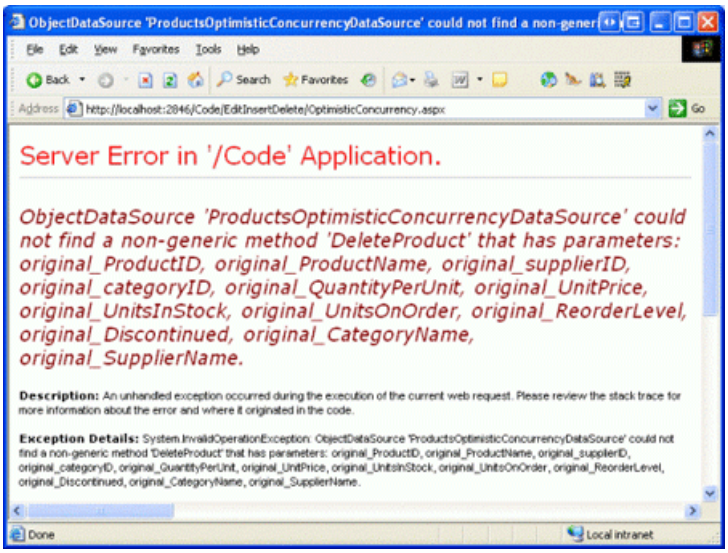


Figure 16: The ObjectDataSource Cannot Find a Method with the Input Parameters it Wants to Send (Click to view full-size image)

Looking at the exception's message, it's clear that the ObjectDataSource wants to invoke a BLL `DeleteProduct` method that includes `original_CategoryName` and `original_SupplierName` input parameters. This is because the `ItemTemplate` s for the `CategoryID` and `SupplierID` TemplateFields currently contain two-way Bind statements with the `CategoryName` and `SupplierName` data fields. Instead, we need to include `Bind` statements with the `CategoryID` and `SupplierID` data fields. To accomplish this, replace the existing Bind statements with `Eval` statements, and then add hidden Label controls whose `Text` properties are bound to the `CategoryID` and `SupplierID` data fields using two-way databinding, as shown below:

```
<asp:TemplateField HeaderText="Category" SortExpression="CategoryName">
    <EditItemTemplate>
        ...
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="DummyCategoryID" runat="server"
            Text='<%# Bind("CategoryID") %>' Visible="False"></asp:Label>
        <asp:Label ID="Label2" runat="server"
            Text='<%# Eval("CategoryName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="Supplier" SortExpression="SupplierName">
    <EditItemTemplate>
        ...
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="DummySupplierID" runat="server"
            Text='<%# Bind("SupplierID") %>' Visible="False"></asp:Label>
        <asp:Label ID="Label3" runat="server"
            Text='<%# Eval("SupplierName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

With these changes, we are now able to successfully delete and edit product information! In Step 5 we'll look at how to verify that concurrency violations are being detected. But for now, take a few minutes to try updating and deleting a few records to ensure that updating and deleting for a single user works as expected.

Step 5: Testing the Optimistic Concurrency Support

In order to verify that concurrency violations are being detected (rather than resulting in data being blindly overwritten), we need to open two browser windows to this page. In both browser instances, click on the Edit button for Chai. Then, in just one of the browsers, change the name to "Chai Tea" and click Update. The update should succeed and return the GridView to its pre-editing state, with "Chai Tea" as the new product name.

In the other browser window instance, however, the product name TextBox still shows "Chai". In this second browser window, update the `UnitPrice` to `25.00`. Without optimistic concurrency support, clicking update in the second browser instance would change the product name back to "Chai", thereby overwriting the changes made by the first browser instance. With optimistic concurrency employed, however, clicking the Update button in the second browser instance results in a DBConcurrencyException.

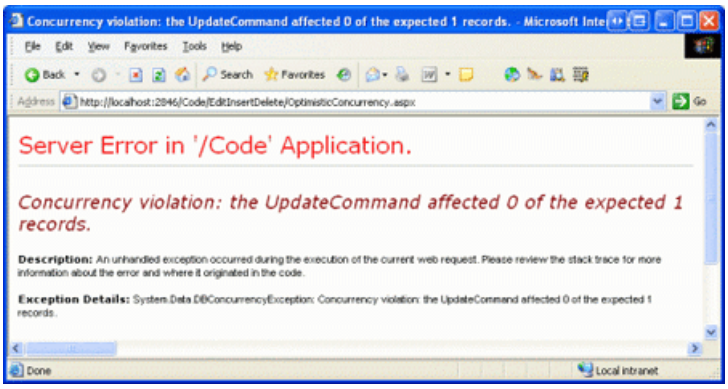


Figure 17: When a Concurrency Violation is Detected, a `DBConcurrencyException` is Thrown (Click to view full-size image)

The `DBConcurrencyException` is only thrown when the DAL's batch update pattern is utilized. The DB direct pattern does not raise an exception, it merely indicates that no rows were affected. To illustrate this, return both browser instances' GridView to their pre-editing state. Next, in the first browser instance, click the Edit button and change the product name from "Chai Tea" back to "Chai" and click Update. In the second browser window, click the Delete

button for Chai.

Upon clicking Delete, the page posts back, the GridView invokes the ObjectDataSource's `Delete()` method, and the ObjectDataSource calls down into the `ProductsOptimisticConcurrencyBLL` class's `DeleteProduct` method, passing along the original values. The original `ProductName` value for the second browser instance is "Chai Tea", which doesn't match up with the current `ProductName` value in the database. Therefore the `DELETE` statement issued to the database affects zero rows since there's no record in the database that the `WHERE` clause satisfies. The `DeleteProduct` method returns `false` and the ObjectDataSource's data is rebound to the GridView.

From the end user's perspective, clicking on the Delete button for Chai Tea in the second browser window caused the screen to flash and, upon coming back, the product is still there, although now it's listed as "Chai" (the product name change made by the first browser instance). If the user clicks the Delete button again, the Delete will succeed, as the GridView's original `ProductName` value ("Chai") now matches up with the value in the database.

In both of these cases, the user experience is far from ideal. We clearly don't want to show the user the nitty-gritty details of the `DBConcurrencyException` exception when using the batch update pattern. And the behavior when using the DB direct pattern is somewhat confusing as the users command failed, but there was no precise indication of why.

To remedy these two issues, we can create Label Web controls on the page that provide an explanation to why an update or delete failed. For the batch update pattern, we can determine whether or not a `DBConcurrencyException` exception occurred in the GridView's post-level event handler, displaying the warning label as needed. For the DB direct method, we can examine the return value of the BLL method (which is `true` if one row was affected, `false` otherwise) and display an informational message as needed.

Step 6: Adding Informational Messages and Displaying Them in the Face of a Concurrency Violation

When a concurrency violation occurs, the behavior exhibited depends on whether the DAL's batch update or DB direct pattern was used. Our tutorial uses both patterns, with the batch update pattern being used for updating and the DB direct pattern used for deleting. To get started, let's add two Label Web controls to our page that explain that a concurrency violation occurred when attempting to delete or update data. Set the Label control's `Visible` and `EnableViewState` properties to `false`; this will cause them to be hidden on each page visit except for those particular page visits where their `Visible` property is programmatically set to `true`.

```
<asp:Label ID="DeleteConflictMessage" runat="server" Visible="False"
    EnableViewState="False" CssClass="Warning"
    Text="The record you attempted to delete has been modified by another user
           since you last visited this page. Your delete was cancelled to allow
           you to review the other user's changes and determine if you want to
           continue deleting this record." />
<asp:Label ID="UpdateConflictMessage" runat="server" Visible="False"
    EnableViewState="False" CssClass="Warning"
    Text="The record you attempted to update has been modified by another user
           since you started the update process. Your changes have been replaced
           with the current values. Please review the existing values and make
           any needed changes." />
```

In addition to setting their `Visible`, `EnabledViewState`, and `Text` properties, I've also set the `CssClass` property to `Warning`, which causes the Label's to be displayed in a large, red, italic, bold font. This CSS `Warning` class was defined and added to Styles.css back in the *Examining the Events Associated with Inserting, Updating, and Deleting* tutorial.

After adding these Labels, the Designer in Visual Studio should look similar to Figure 18.

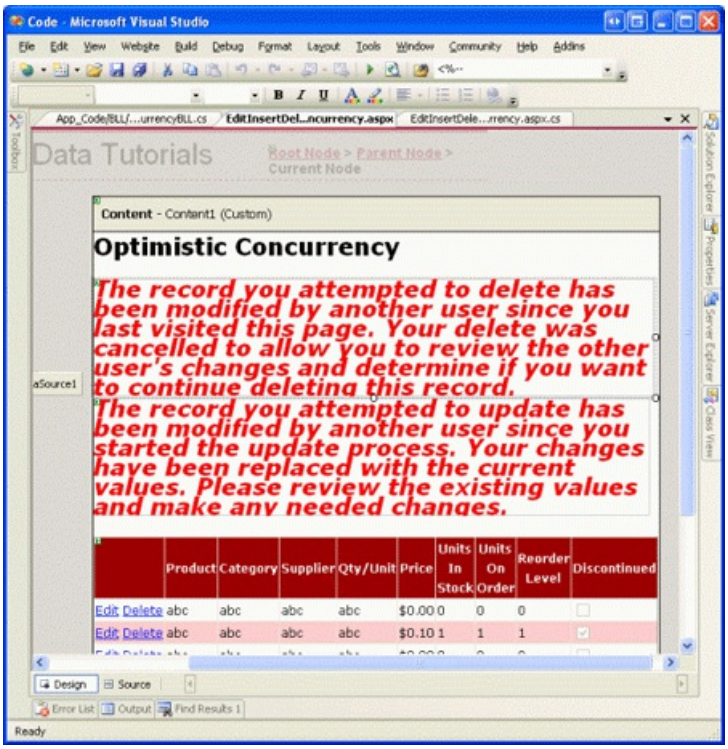


Figure 18: Two Label Controls Have Been Added to the Page (Click to view full-size image)

With these Label Web controls in place, we're ready to examine how to determine when a concurrency violation has occurred, at which point the appropriate Label's `Visible` property can be set to `true`, displaying the informational message.

Handling Concurrency Violations When Updating

Let's first look at how to handle concurrency violations when using the batch update pattern. Since such violations with the batch update pattern cause a `DBConcurrencyException` exception to be thrown, we need to add code to our ASP.NET page to determine whether a `DBConcurrencyException` exception occurred during the update process. If so, we should display a message to the user explaining that their changes were not saved because another user had modified the same data between when they started editing the record and when they clicked the Update button.

As we saw in the *Handling BLL- and DAL-Level Exceptions in an ASP.NET Page* tutorial, such exceptions can be detected and suppressed in the data Web control's post-level event handlers. Therefore, we need to create an event handler for the GridView's `RowUpdated` event that checks if a `DBConcurrencyException` exception has been thrown. This event handler is passed a reference to any exception that was raised during the updating process, as shown in the event handler code below:

```
Protected Sub ProductsGrid_RowUpdated _
        (ByVal sender As Object, ByVal e As GridViewUpdatedEventArgs) _
        Handles ProductsGrid.RowUpdated
    If e.Exception IsNot Nothing AndAlso e.Exception.InnerException IsNot Nothing Then
        If TypeOf e.Exception.InnerException Is System.Data.DBConcurrencyException Then
            ' Display the warning message and note that the exception has
            ' been handled...
            UpdateConflictMessage.Visible = True
            e.ExceptionHandled = True
        End If
    End If
End Sub
```

In the face of a `DBConcurrencyException` exception, this event handler displays the `UpdateConflictMessage` Label control and indicates that the exception has been handled. With this code in place, when a concurrency violation

occurs when updating a record, the user's changes are lost, since they would have overwritten another user's modifications at the same time. In particular, the GridView is returned to its pre-editing state and bound to the current database data. This will update the GridView row with the other user's changes, which were previously not visible. Additionally, the `UpdateConflictMessage` Label control will explain to the user what just happened. This sequence of events is detailed in Figure 19.

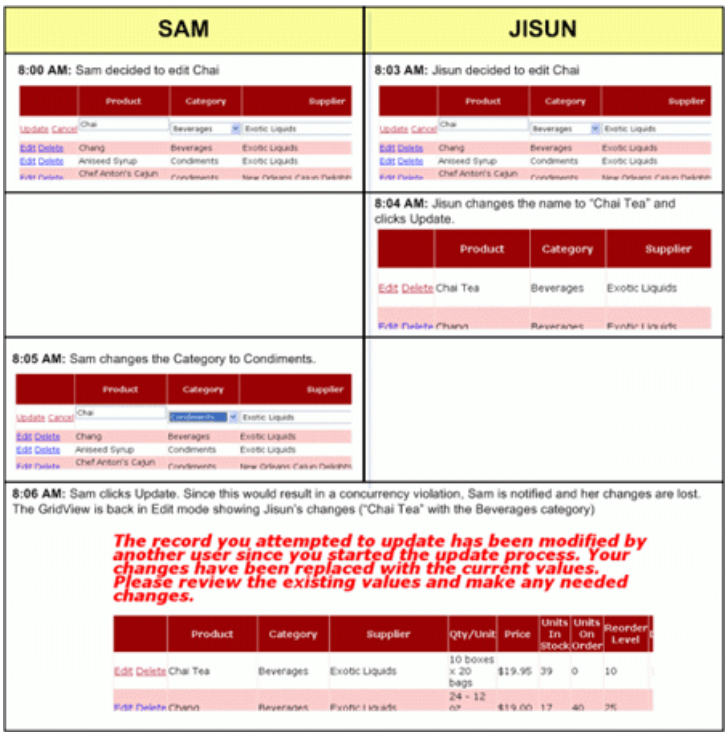


Figure 19: A User s Updates are Lost in the Face of a Concurrency Violation (Click to view full-size image)

> **NOTE**
>
> Alternatively, rather than returning the GridView to the pre-editing state, we could leave the GridView in its editing state by setting the `KeepInEditMode` property of the passed-in `GridViewUpdatedEventArgs` object to true. If you take this approach, however, be certain to rebind the data to the GridView (by invoking its `DataBind()` method) so that the other user's values are loaded into the editing interface. The code available for download with this tutorial has these two lines of code in the `RowUpdated` event handler commented out; simply uncomment these lines of code to have the GridView remain in edit mode after a concurrency violation.

Responding to Concurrency Violations When Deleting

With the DB direct pattern, there is no exception raised in the face of a concurrency violation. Instead, the database statement simply affects no records, as the WHERE clause does not match with any record. All of the data modification methods created in the BLL have been designed such that they return a Boolean value indicating whether or not they affected precisely one record. Therefore, to determine if a concurrency violation occurred when deleting a record, we can examine the return value of the BLL's `DeleteProduct` method.

The return value for a BLL method can be examined in the ObjectDataSource's post-level event handlers through the `ReturnValue` property of the `ObjectDataSourceStatusEventArgs` object passed into the event handler. Since we are interested in determining the return value from the `DeleteProduct` method, we need to create an event handler for the ObjectDataSource's `Deleted` event. The `ReturnValue` property is of type `object` and can be `null` if an exception was raised and the method was interrupted before it could return a value. Therefore, we should first ensure that the `ReturnValue` property is not `null` and is a Boolean value. Assuming this check passes, we show the `DeleteConflictMessage` Label control if the `ReturnValue` is `false`. This can be accomplished by using the following code:

```
Protected Sub ProductsOptimisticConcurrencyDataSource_Deleted _
        (ByVal sender As Object, ByVal e As ObjectDataSourceStatusEventArgs) _
        Handles ProductsOptimisticConcurrencyDataSource.Deleted
    If e.ReturnValue IsNot Nothing AndAlso TypeOf e.ReturnValue Is Boolean Then
        Dim deleteReturnValue As Boolean = CType(e.ReturnValue, Boolean)
        If deleteReturnValue = False Then
            ' No row was deleted, display the warning message
            DeleteConflictMessage.Visible = True
        End If
    End If
End Sub
```

In the face of a concurrency violation, the user's delete request is canceled. The GridView is refreshed, showing the changes that occurred for that record between the time the user loaded the page and when he clicked the Delete button. When such a violation transpires, the `DeleteConflictMessage` Label is shown, explaining what just happened (see Figure 20).

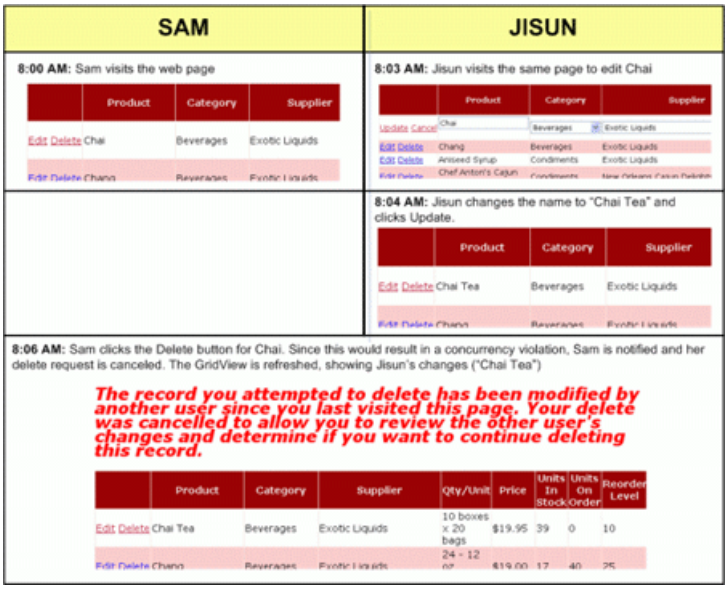


Figure 20: A User s Delete is Canceled in the Face of a Concurrency Violation (Click to view full-size image)

Summary

Opportunities for concurrency violations exist in every application that allows multiple, concurrent users to update or delete data. If such violations are not accounted for, when two users simultaneously update the same data whoever gets in the last write "wins," overwriting the other user's changes changes. Alternatively, developers can implement either optimistic or pessimistic concurrency control. Optimistic concurrency control assumes that concurrency violations are infrequent and simply disallows an update or delete command that would constitute a concurrency violation. Pessimistic concurrency control assumes that concurrency violations are frequent and simply rejecting one user's update or delete command is not acceptable. With pessimistic concurrency control, updating a record involves locking it, thereby preventing any other users from modifying or deleting the record while it is locked.

The Typed DataSet in .NET provides functionality for supporting optimistic concurrency control. In particular, the `UPDATE` and `DELETE` statements issued to the database include all of the table's columns, thereby ensuring that the update or delete will only occur if the record's current data matches with the original data the user had when performing their update or delete. Once the DAL has been configured to support optimistic concurrency, the BLL methods need to be updated. Additionally, the ASP.NET page that calls down into the BLL must be configured such that the ObjectDataSource retrieves the original values from its data Web control and passes them down into the BLL.

As we saw in this tutorial, implementing optimistic concurrency control in an ASP.NET web application involves

updating the DAL and BLL and adding support in the ASP.NET page. Whether or not this added work is a wise investment of your time and effort depends on your application. If you infrequently have concurrent users updating data, or the data they are updating is different from one another, then concurrency control is not a key issue. If, however, you routinely have multiple users on your site working with the same data, concurrency control can help prevent one user's updates or deletes from unwittingly overwriting another's.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Adding Client-Side Confirmation When Deleting (VB)

2/20/2020 • 11 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In the interfaces we've created so far, a user can accidentally delete data by clicking the Delete button when they meant to click the Edit button. In this tutorial we'll add a client-side confirmation dialog box that appears when the Delete button is clicked.

Introduction

Over the past several tutorials we ve seen how to use our application architecture, ObjectDataSource, and the data Web controls in concert to provide inserting, editing, and deleting capabilities. The deleting interfaces we ve examined thus far have been composed of a Delete button that, when clicked, causes a postback and invokes the ObjectDataSource s `Delete()` method. The `Delete()` method then invokes the configured method from the Business Logic Layer, which propagates the call down to the Data Access Layer, issuing the actual `DELETE` statement to the database.

While this user interface enables visitors to delete records through the GridView, DetailsView, or FormView controls, it lacks any sort of confirmation when the user clicks the Delete button. If a user accidentally clicks the Delete button when they meant to click Edit, the record they meant to update will instead be deleted. To help prevent this, in this tutorial we'll add a client-side confirmation dialog box that appears when the Delete button is clicked.

The JavaScript `confirm(string)` function displays its string input parameter as the text inside a modal dialog box that comes equipped with two buttons - OK and Cancel (see Figure 1). The `confirm(string)` function returns a Boolean value depending on what button is clicked (`true` , if the user clicks OK, and `false` if they click Cancel).

Figure 1: The JavaScript `confirm(string)` Method Displays a Modal, Client-Side Messagebox

During a form submission, if a value of `false` is returned from a client-side event handler then the form submission is cancelled. Using this feature, we can have the Delete button s client-side `onclick` event handler return the value of a call to `confirm("Are you sure you want to delete this product?")`. If the user clicks Cancel, `confirm(string)` will return false, thereby causing the form submission to cancel. With no postback, the product whose Delete button was clicked won't be deleted. If, however, the user clicks OK in the confirmation dialog box, the postback will continue unabated and the product will be deleted. Consult Using JavaScript s `confirm()` Method to Control Form Submission for more information on this technique.

Adding the necessary client-side script differs slightly if using templates than when using a CommandField. Therefore, in this tutorial we will look at both a FormView and GridView example.

NOTE

Using client-side confirmation techniques, like the ones discussed in this tutorial, assumes that your users are visiting with browsers that support JavaScript and that they have JavaScript enabled. If either of these assumptions are not true for a particular user, clicking the Delete button will immediately cause a postback (not displaying a confirm messagebox).

Step 1: Creating a FormView That Supports Deletion

Start by adding a FormView to the `ConfirmationOnDelete.aspx` page in the `EditInsertDelete` folder, binding it to a new ObjectDataSource that pulls back the product information via the `ProductsBLL` class s `GetProducts()` method. Also configure the ObjectDataSource so that the `ProductsBLL` class s `DeleteProduct(productID)` method is mapped to the ObjectDataSource s `Delete()` method; ensure that the INSERT and UPDATE tabs drop-down lists are set to (None). Finally, check the Enable Paging checkbox in the FormView s smart tag.

After these steps, the new ObjectDataSource s declarative markup will look like the following:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    DeleteMethod="DeleteProduct" OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
    <DeleteParameters>
        <asp:Parameter Name="productID" Type="Int32" />
    </DeleteParameters>
</asp:ObjectDataSource>
```

As in our past examples that did not use optimistic concurrency, take a moment to clear out the ObjectDataSource s `OldValuesParameterFormatString` property.

Since it has been bound to an ObjectDataSource control that only supports deleting, the FormView s `ItemTemplate` offers only the Delete button, lacking the New and Update buttons. The FormView s declarative markup, however, includes a superfluous `EditItemTemplate` and `InsertItemTemplate`, which can be removed. Take a moment to customize the `ItemTemplate` so that is shows only a subset of the product data fields. I ve configured mine to show the product s name in an `<h3>` heading above its supplier and category names (along with the Delete button).

```
<asp:FormView ID="FormView1" AllowPaging="True" DataKeyNames="ProductID"
    DataSourceID="ObjectDataSource1" runat="server">
    <ItemTemplate>
        <h3><i><%# Eval("ProductName") %></i></h3>
        <b>Category:</b>
        <asp:Label ID="CategoryNameLabel" runat="server"
            Text='<%# Eval("CategoryName") %>'>
        </asp:Label><br />
        <b>Supplier:</b>
        <asp:Label ID="SupplierNameLabel" runat="server"
            Text='<%# Eval("SupplierName") %>'>
        </asp:Label><br />
        <asp:LinkButton ID="DeleteButton" runat="server" CausesValidation="False"
            CommandName="Delete" Text="Delete">
        </asp:LinkButton>
    </ItemTemplate>
</asp:FormView>
```

With these changes, we have a fully functional web page that allows a user to toggle through the products one at a time, with the ability to delete a product by simply clicking the Delete button. Figure 2 shows a screen shot of our progress thus far when viewed through a browser.

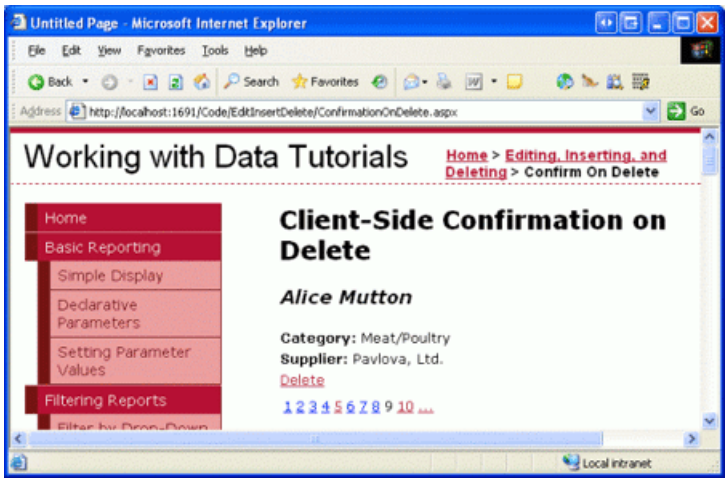


Figure 2: The FormView Shows Information About a Single Product (Click to view full-size image)

Step 2: Calling the confirm(string) Function from the Delete Buttons Client-Side onclick Event

With the FormView created, the final step is to configure the Delete button such that when it s clicked by the visitor, the JavaScript `confirm(string)` function is invoked. Adding client-side script to a Button, LinkButton, or ImageButton s client-side `onclick` event can be accomplished through the use of the `OnClientClick property`, which is new to ASP.NET 2.0. Since we want to have the value of the `confirm(string)` function returned, simply set this property to: `return confirm('Are you certain that you want to delete this product?');`

After this change the Delete LinkButton s declarative syntax should look something like:

```
<asp:LinkButton ID="DeleteButton" runat="server" CausesValidation="False"
    CommandName="Delete" Text="Delete"
    OnClientClick="return confirm('Are you certain you want to delete this product?');">
</asp:LinkButton>
```

That s all there is to it! Figure 3 shows a screen shot of this confirmation in action. Clicking the Delete button brings up the confirm dialog box. If the user clicks Cancel, the postback is cancelled and the product is not deleted. If, however, the user clicks OK, the postback continues and the ObjectDataSource s `Delete()` method is invoked, culminating in the database record being deleted.

> **NOTE**
>
> The string passed into the `confirm(string)` JavaScript function is delimited with apostrophes (rather than quotation marks). In JavaScript, strings can be delimited using either character. We use apostrophes here so that the delimiters for the string passed into `confirm(string)` do not introduce an ambiguity with the delimiters used for the `OnClientClick` property value.

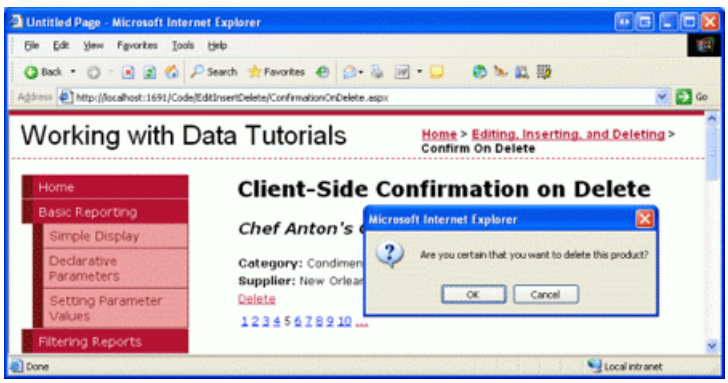


Figure 3: A Confirmation is Now Displayed When Clicking the Delete Button (Click to view full-size image)

Step 3: Configuring the OnClientClick Property for the Delete Button in a CommandField

When working with a Button, LinkButton, or ImageButton directly in a template, a confirmation dialog box can be associated with it by simply configuring its `OnClientClick` property to return the results of the JavaScript `confirm(string)` function. However, the CommandField - which adds a field of Delete buttons to a GridView or DetailsView - does not have an `OnClientClick` property that can be set declaratively. Instead, we must programmatically reference the Delete button in the GridView or DetailsView s appropriate `DataBound` event handler, and then set its `OnClientClick` property there.

> **NOTE**
>
> When setting the Delete button s `OnClientClick` property in the appropriate `DataBound` event handler, we have access to the data was bound to the current record. This means we can extend the confirmation message to include details about the particular record, such as, "Are you sure you want to delete the Chai product?" Such customization is also possible in templates using databinding syntax.

To practice setting the `OnClientClick` property for the Delete button(s) in a CommandField, let s add a GridView to the page. Configure this GridView to use the same ObjectDataSource control that the FormView uses. Also limit the GridView s BoundFields to only include the product s name, category, and supplier. Lastly, check the Enable Deleting checkbox from the GridView s smart tag. This will add a CommandField to the GridView s `Columns` collection with its `ShowDeleteButton` property set to `true`.

After making these changes, your GridView s declarative markup should look like the following:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:CommandField ShowDeleteButton="True" />
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category" ReadOnly="True"
            SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier" ReadOnly="True"
            SortExpression="SupplierName" />
    </Columns>
</asp:GridView>
```

The CommandField contains a single Delete LinkButton instance that can be accessed programmatically from the GridView s `RowDataBound` event handler. Once referenced, we can set its `OnClientClick` property accordingly. Create an event handler for the `RowDataBound` event using the following code:

```
Protected Sub GridView1_RowDataBound(ByVal sender As Object, _
    ByVal e As System.Web.UI.WebControls.GridViewRowEventArgs) _
    Handles GridView1.RowDataBound

    If e.Row.RowType = DataControlRowType.DataRow Then
        ' reference the Delete LinkButton
        Dim db As LinkButton = CType(e.Row.Cells(0).Controls(0), LinkButton)

        ' Get information about the product bound to the row
        Dim product As Northwind.ProductsRow = _
            CType(CType(e.Row.DataItem, System.Data.DataRowView).Row, _
            Northwind.ProductsRow)

        db.OnClientClick = String.Format( _
            "return confirm('Are you certain you want to delete the {0} product?');", _
            product.ProductName.Replace("'", "\'"))
    End If
End Sub
```

This event handler works with data rows (those that will have the Delete button) and begins by programmatically referencing the Delete button. In general use the following pattern:

```
Dim obj As ButtonType = _
    CType(e.Row.Cells(commandFieldIndex).Controls(controlIndex), ButtonType)
```

ButtonType is the type of button being used by the CommandField - Button, LinkButton, or ImageButton. By default, the CommandField uses LinkButtons, but this can be customized via the CommandField s `ButtonType property`. The *commandFieldIndex* is the ordinal index of the CommandField within the GridView s `Columns` collection, whereas the *controlIndex* is the index of the Delete button within the CommandField s `Controls` collection. The *controlIndex* value depends on the button s position relative to other buttons in the CommandField. For example, if the only button displayed in the CommandField is the Delete button, use an index of 0. If, however, there s an Edit button that precedes the Delete button, use an index of 2. The reason an index of 2 is used is because two controls are added by the CommandField before the Delete button: the Edit button and a LiteralControl that s used to add some space between the Edit and Delete buttons.

For our particular example, the CommandField uses LinkButtons and, being the left-most field, has a *commandFieldIndex* of 0. Since there are no other buttons but the Delete button in the CommandField, we use a *controlIndex* of 0.

After referencing the Delete button in the CommandField, we next grab information about the product bound to the current GridView row. Finally, we set the Delete button s `OnClientClick` property to the appropriate JavaScript, which includes the product s name. Since the JavaScript string passed into the `confirm(string)` function is delimited using apostrophes we must escape any apostrophes that appear within the product s name. In particular, any apostrophes in the product s name are escaped with "`\'`".

With these changes complete, clicking on a Delete button in the GridView displays a customized confirmation dialog box (see Figure 4). As with the confirmation messagebox from the FormView, if the user clicks Cancel the postback is cancelled, thereby preventing the deletion from occurring.

NOTE

This technique can also be used to programmatically access the Delete button in the CommandField in a DetailsView. For the DetailsView, however, you d create an event handler for the `DataBound` event, since the DetailsView does not have a `RowDataBound` event.

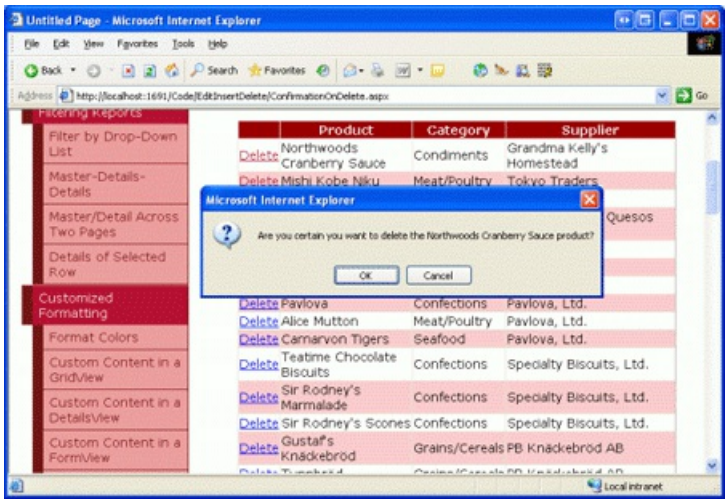


Figure 4: Clicking the GridView s Delete Button Displays a Customized Confirmation Dialog Box (Click to view full-size image)

Using TemplateFields

One of the disadvantages of the CommandField is that its buttons must be accessed through indexing and that the resulting object must be cast to the appropriate button type (Button, LinkButton, or ImageButton). Using "magic numbers" and hard-coded types invites problems that cannot be discovered until runtime. For example, if you, or another developer, adds new buttons to the CommandField at some point in the future (such as an Edit button) or changes the `ButtonType` property, the existing code will still compile without error, but visiting the page may cause an exception or unexpected behavior, depending on how your code was written and what changes were made.

An alternative approach is to convert the GridView and DetailsView s CommandFields into TemplateFields. This will generate a TemplateField with an `ItemTemplate` that has a LinkButton (or Button or ImageButton) for each button in the CommandField. These buttons `OnClientClick` properties can be assigned declaratively, as we saw with the FormView, or can be programmatically accessed in the appropriate `DataBound` event handler using the following pattern:

```
Dim obj As ButtonType = CType(e.Row.FindControl("controlID"), ButtonType)
```

Where *controlID* is the value of the button s `ID` property. While this pattern still requires a hard-coded type for the cast, it removes the need for indexing, allowing for the layout to change without resulting in a runtime error.

Summary

The JavaScript `confirm(string)` function is a commonly used technique for controlling form submission workflow. When executed, the function displays a modal, client-side dialog box that includes two buttons, OK and Cancel. If the user clicks OK, the `confirm(string)` function returns `true`; clicking Cancel returns `false`. This functionality, coupled with a browser s behavior to cancel a form submission if an event handler during the submission process returns `false`, can be used to display a confirmation messagebox when deleting a record.

The `confirm(string)` function can be associated with a Button Web control s client-side `onclick` event handler through the control s `OnClientClick` property. When working with a Delete button in a template - either in one of the FormView s templates or in a TemplateField in the DetailsView or GridView - this property can be set either declaratively or programmatically, as we saw in this tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Limiting Data Modification Functionality Based on the User (VB)

2/20/2020 • 18 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

In a web application that allows users to edit data, different user accounts may have different data-editing privileges. In this tutorial we'll examine how to dynamically adjust the data modification capabilities based on the visiting user.

Introduction

A number of web applications support user accounts and provide different options, reports, and functionality based on the logged in user. For example, with our tutorials we might want to allow users from the supplier companies to log on to the site and update general information about their products - their name and quantity per unit, perhaps - along with supplier information, such as their company name, address, the contact person s information, and so on. Furthermore, we might want to include some user accounts for people from our company so that they can log on and update product information like units on stock, reorder level, and so forth. Our web application might also allow anonymous users to visit (people who have not logged on), but would limit them to just viewing data. With such a user account system in place, we would want the data Web controls in our ASP.NET pages to offer the inserting, editing, and deleting capabilities appropriate for the currently logged on user.

In this tutorial we'll examine how to dynamically adjust the data modification capabilities based on the visiting user. In particular, we'll create a page that displays the suppliers information in an editable DetailsView along with a GridView that lists the products provided by the supplier. If the user visiting the page is from our company, they can: view any supplier s information; edit their address; and edit the information for any product provided by the supplier. If, however, the user is from a particular company, they can only view and edit their own address information and can only edit their products that have not been marked as discontinued.

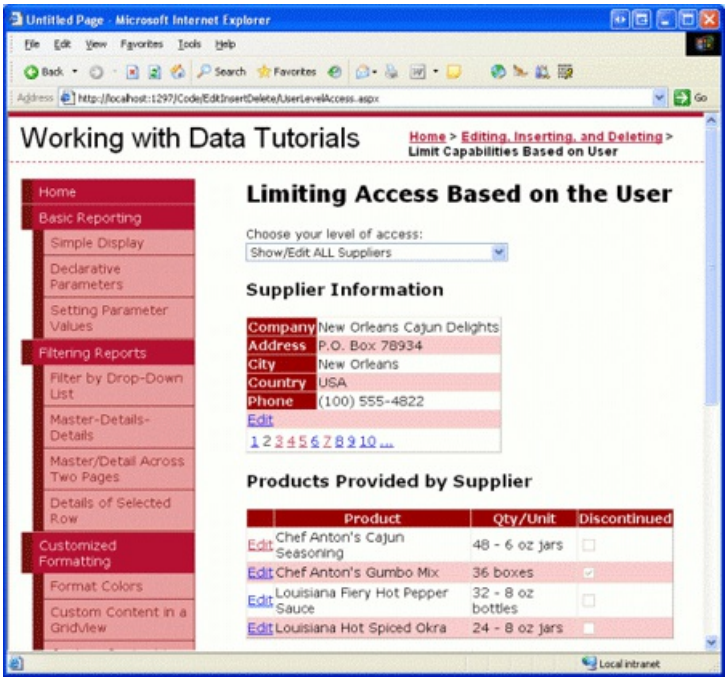


Figure 1: A User from Our Company Can Edit Any Supplier s Information (Click to view full-size image)

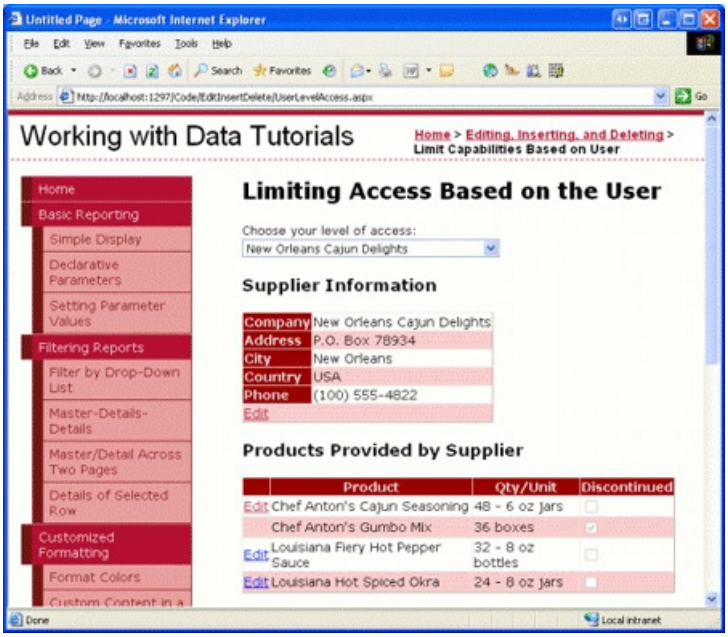


Figure 2: A User from a Particular Supplier Can Only View and Edit their Information (Click to view full-size image)

Let s get started!

> **NOTE**
>
> ASP.NET 2.0 s membership system provides a standardized, extensible platform for creating, managing, and validating user accounts. Since an examination of the membership system is beyond the scope of these tutorials, this tutorial instead "fakes" membership by allowing anonymous visitors to choose whether they are from a particular supplier or from our company. For more on membership, refer to my Examining ASP.NET 2.0 s Membership, Roles, and Profile article series.

Step 1: Allowing the User to Specify their Access Rights

In a real-world web application, a user s account information would include whether they worked for our company or for a particular supplier, and this information would be programmatically accessible from our ASP.NET pages once the user has logged on to the site. This information could be captured through ASP.NET 2.0 s roles system, as user-level account information through the profile system, or through some custom means.

Since the aim of this tutorial is to demonstrate adjusting the data modification capabilities based on the logged on user, and is not meant to showcase ASP.NET 2.0 s membership, roles, and profile systems, we'll use a very simply mechanism to determine the capabilities for the user visiting the page - a DropDownList from which the user can indicate if they should be able to view and edit any of the suppliers information or, alternatively, what particular supplier s information they can view and edit. If the user indicates that she can view and edit all supplier information (the default), she can page through all suppliers, edit any supplier s address information, and edit the name and quantity per unit for any product provided by the selected supplier. If the user indicates that she can only view and edit a particular supplier, however, then she can only view the details and products for that one supplier and can only update the name and quantity per unit information for those products that are *not* discontinued.

Our first step in this tutorial, then, is to create this DropDownList and populate it with the suppliers in the system. Open the `UserLevelAccess.aspx` page in the `EditInsertDelete` folder, add a DropDownList whose `ID` property is set to `Suppliers`, and bind this DropDownList to a new ObjectDataSource named `AllSuppliersDataSource`.

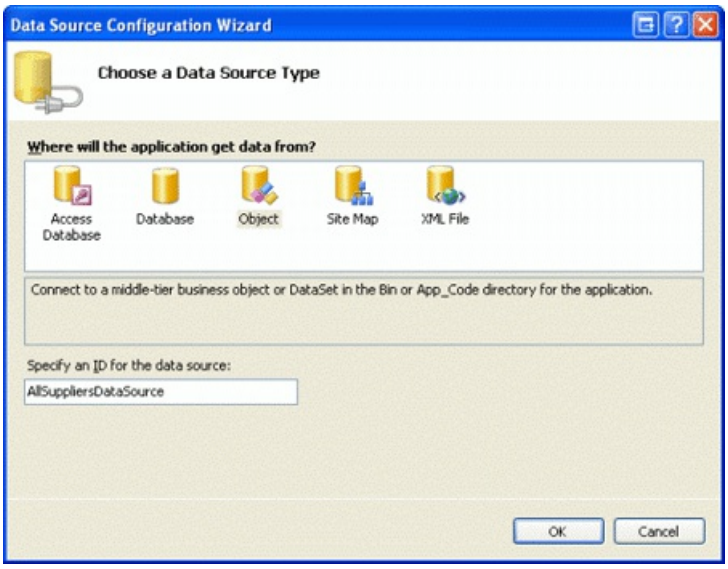


Figure 3: Create a New ObjectDataSource Named `AllSuppliersDataSource` (Click to view full-size image)

Since we want this DropDownList to include all suppliers, configure the ObjectDataSource to invoke the `SuppliersBLL` class s `GetSuppliers()` method. Also ensure that the ObjectDataSource s `Update()` method is mapped to the `SuppliersBLL` class s `UpdateSupplierAddress` method, as this ObjectDataSource will also be used by the DetailsView we'll be adding in Step 2.

After completing the ObjectDataSource wizard, complete the steps by configuring the `Suppliers` DropDownList such that it shows the `CompanyName` data field and uses the `SupplierID` data field as the value for each `ListItem`.

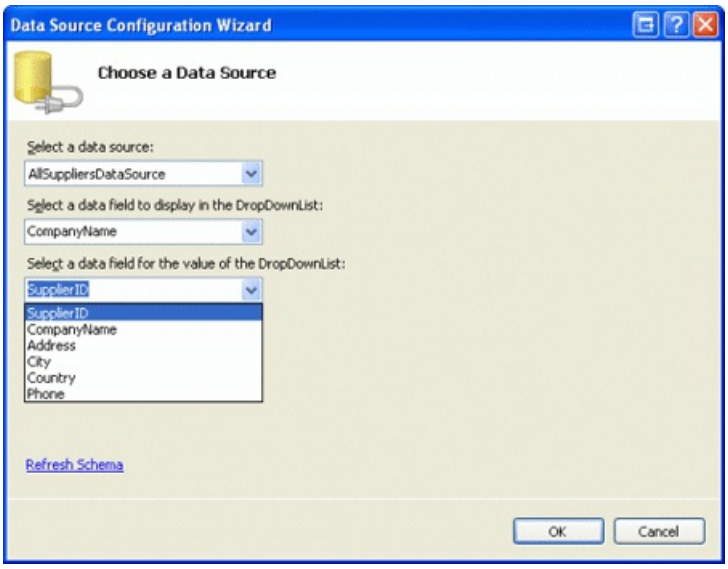


Figure 4: Configure the `Suppliers` DropDownList to Use the `CompanyName` and `SupplierID` Data Fields (Click to view full-size image)

At this point, the DropDownList lists the company names of the suppliers in the database. However, we also need to include a "Show/Edit ALL Suppliers" option to the DropDownList. To accomplish this, set the `Suppliers` DropDownList s `AppendDataBoundItems` property to `true` and then add a `ListItem` whose `Text` property is "Show/Edit ALL Suppliers" and whose value is `-1`. This can be added directly through the declarative markup or through the Designer by going to the Properties window and clicking on the ellipses in the DropDownList s `Items` property.

> **NOTE**
>
> Refer back to the *Master/Detail Filtering With a DropDownList* tutorial for a more detailed discussion on adding a Select All item to a databound DropDownList.

After the `AppendDataBoundItems` property has been set and the `ListItem` added, the DropDownList s declarative markup should look like:

```
<asp:DropDownList ID="Suppliers" runat="server" AppendDataBoundItems="True"
    DataSourceID="AllSuppliersDataSource" DataTextField="CompanyName"
    DataValueField="SupplierID">
    <asp:ListItem Value="-1">Show/Edit ALL Suppliers</asp:ListItem>
</asp:DropDownList>
```

Figure 5 shows a screen shot of our current progress, when viewed through a browser.

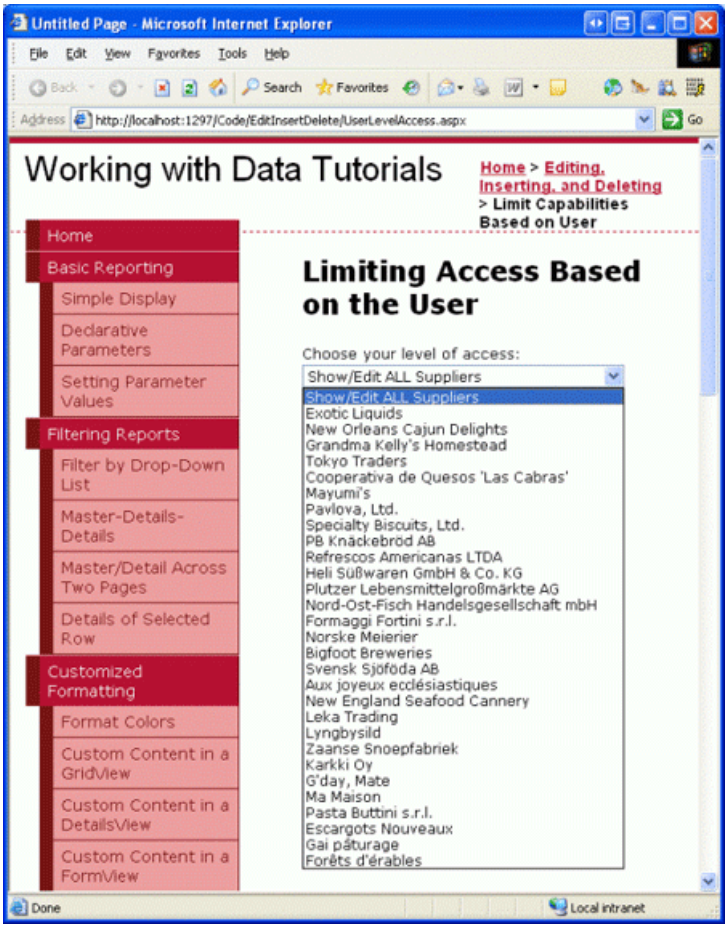


Figure 5: The `Suppliers` DropDownList Contains a Show ALL `ListItem`, Plus One for Each Supplier (Click to view full-size image)

Since we want to update the user interface immediately after the user has changed their selection, set the `Suppliers` DropDownList s `AutoPostBack` property to `true`. In Step 2 we'll create a DetailsView control that will show the information for the supplier(s) based on the DropDownList selection. Then, in Step 3, we'll create an event handler for this DropDownList s `SelectedIndexChanged` event, in which we'll add code that binds the appropriate supplier information to the DetailsView based upon the selected supplier.

Step 2: Adding a DetailsView Control

Let s use a DetailsView to show supplier information. For the user who can view and edit all suppliers, the DetailsView will support paging, allowing the user to step through the supplier information one record at a time.

If the user works for a particular supplier, however, the DetailsView will show only that particular supplier s information and will not include a paging interface. In either case, the DetailsView needs to allow the user to edit the supplier s address, city, and country fields.

Add a DetailsView to the page beneath the `Suppliers` DropDownList, set its `ID` property to `SupplierDetails`, and bind it to the `AllSuppliersDataSource` ObjectDataSource created in the previous step. Next, check the Enable Paging and Enable Editing checkboxes from the DetailsView s smart tag.

> **NOTE**
> If you don t see an Enable Editing option in the DetailsView s smart tag it s because you did not map the ObjectDataSource s `Update()` method to the `SuppliersBLL` class s `UpdateSupplierAddress` method. Take a moment to go back and make this configuration change, after which the Enable Editing option should appear in the DetailsView s smart tag.

Since the `SuppliersBLL` class s `UpdateSupplierAddress` method only accepts four parameters - `supplierID`, `address`, `city`, and `country` - modify the DetailsView s BoundFields so that the `CompanyName` and `Phone` BoundFields are read-only. Additionally, remove the `SupplierID` BoundField altogether. Finally, the `AllSuppliersDataSource` ObjectDataSource currently has its `OldValuesParameterFormatString` property set to `original_{0}`. Take a moment to remove this property setting from the declarative syntax altogether, or set it to the default value, `{0}`.

After configuring the `SupplierDetails` DetailsView and `AllSuppliersDataSource` ObjectDataSource, we will have the following declarative markup:

```
<asp:ObjectDataSource ID="AllSuppliersDataSource" runat="server"
    SelectMethod="GetSuppliers" TypeName="SuppliersBLL"
    UpdateMethod="UpdateSupplierAddress">
    <UpdateParameters>
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="address" Type="String" />
        <asp:Parameter Name="city" Type="String" />
        <asp:Parameter Name="country" Type="String" />
    </UpdateParameters>
</asp:ObjectDataSource>
<asp:DetailsView ID="SupplierDetails" runat="server" AllowPaging="True"
    AutoGenerateRows="False" DataKeyNames="SupplierID"
    DataSourceID="AllSuppliersDataSource">
    <Fields>
        <asp:BoundField DataField="CompanyName" HeaderText="Company"
            ReadOnly="True" SortExpression="CompanyName" />
        <asp:BoundField DataField="Address" HeaderText="Address"
            SortExpression="Address" />
        <asp:BoundField DataField="City" HeaderText="City"
            SortExpression="City" />
        <asp:BoundField DataField="Country" HeaderText="Country"
            SortExpression="Country" />
        <asp:BoundField DataField="Phone" HeaderText="Phone" ReadOnly="True"
            SortExpression="Phone" />
        <asp:CommandField ShowEditButton="True" />
    </Fields>
</asp:DetailsView>
```

At this point the DetailsView can be paged through and the selected supplier s address information can be updated, regardless of the selection made in the `Suppliers` DropDownList (see Figure 6).

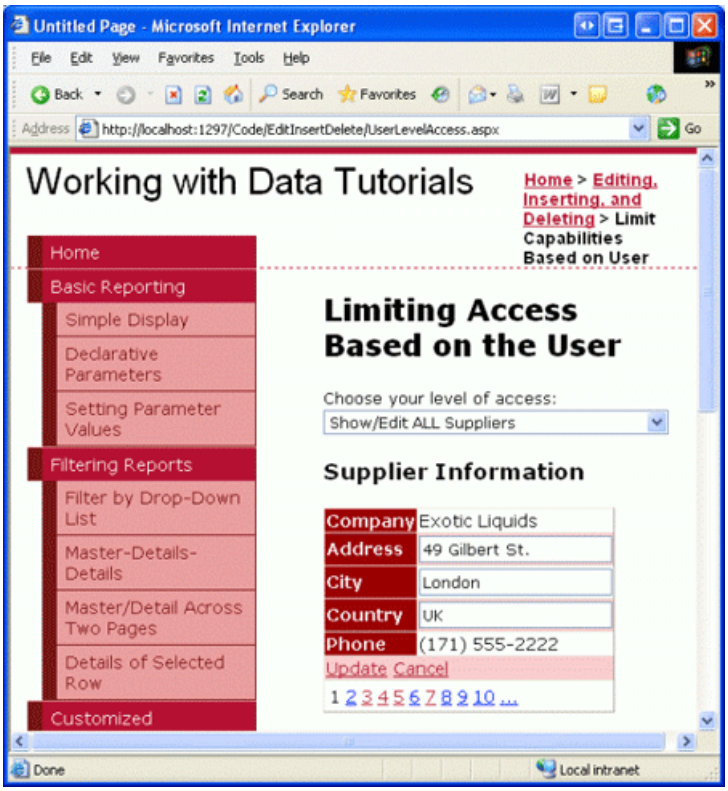


Figure 6: Any Suppliers Information Can Be Viewed, and Its Address Updated (Click to view full-size image)

Step 3: Displaying Only the Selected Supplier s Information

Our page currently displays the information for all suppliers regardless of whether a particular supplier has been selected from the `Suppliers` DropDownList. In order to display just the supplier information for the selected supplier we need to add another ObjectDataSource to our page, one that retrieves information about a particular supplier.

Add a new ObjectDataSource to the page, naming it `SingleSupplierDataSource`. From its smart tag, click the Configure Data Source link and have it use the `SuppliersBLL` class s `GetSupplierBySupplierID(supplierID)` method. As with the `AllSuppliersDataSource` ObjectDataSource, have the `SingleSupplierDataSource` ObjectDataSource s `Update()` method mapped to the `SuppliersBLL` class s `UpdateSupplierAddress` method.

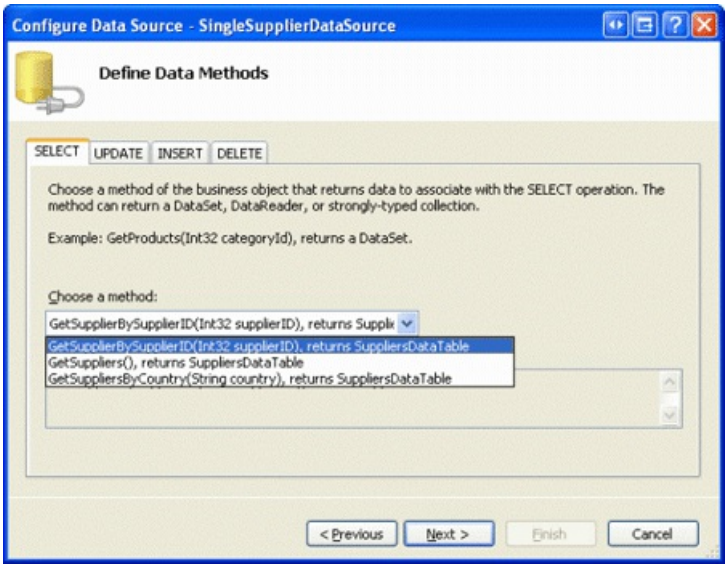


Figure 7: Configure the `SingleSupplierDataSource` ObjectDataSource to Use the `GetSupplierBySupplierID(supplierID)` Method (Click to view full-size image)

Next, we re prompted to specify the parameter source for the `GetSupplierBySupplierID(supplierID)` method s

`supplierID` input parameter. Since we want to show the information for the supplier selected from the DropDownList, use the `Suppliers` DropDownList s `SelectedValue` property as the parameter source.

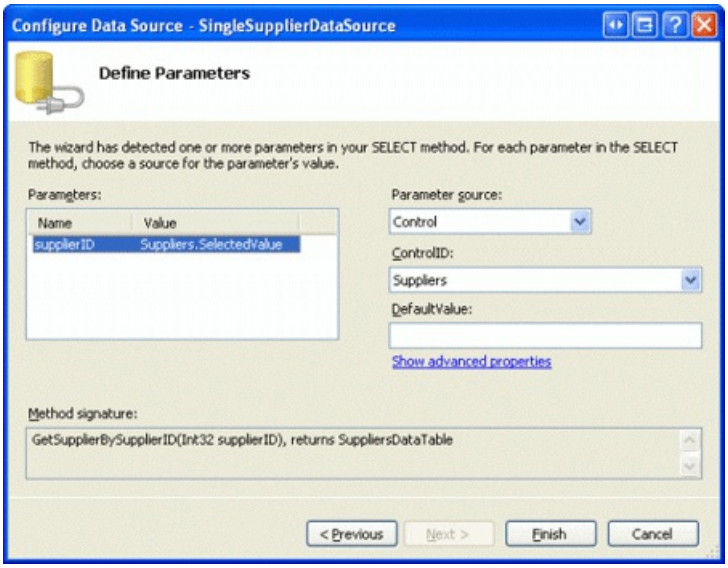


Figure 8: Use the `Suppliers` DropDownList as the `supplierID` Parameter Source (Click to view full-size image)

Even with this second ObjectDataSource added, the DetailsView control is currently configured to always use the `AllSuppliersDataSource` ObjectDataSource. We need to add logic to adjust the data source used by the DetailsView depending on the `Suppliers` DropDownList item selected. To accomplish this, create a `SelectedIndexChanged` event handler for the Suppliers DropDownList. This can most easily be created by double-clicking the DropDownList in the Designer. This event handler needs to determine what data source to use and must rebind the data to the DetailsView. This is accomplished with the following code:

```
Protected Sub Suppliers_SelectedIndexChanged _
    (ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles Suppliers.SelectedIndexChanged
    If Suppliers.SelectedValue = "-1" Then
        ' The "Show/Edit ALL" option has been selected
        SupplierDetails.DataSourceID = "AllSuppliersDataSource"
        ' Reset the page index to show the first record
        SupplierDetails.PageIndex = 0
    Else
        ' The user picked a particular supplier
        SupplierDetails.DataSourceID = "SingleSupplierDataSource"
    End If
    ' Ensure that the DetailsView and GridView are in read-only mode
    SupplierDetails.ChangeMode(DetailsViewMode.ReadOnly)
    ' Need to "refresh" the DetailsView
    SupplierDetails.DataBind()
End Sub
```

The event handler begins by determining whether the "Show/Edit ALL Suppliers" option was selected. If it was, it sets the `SupplierDetails` DetailsView s `DataSourceID` to `AllSuppliersDataSource` and returns the user to the first record in the set of suppliers by setting the `PageIndex` property to 0. If, however, the user has selected a particular supplier from the DropDownList, the DetailsView s `DataSourceID` is assigned to `SingleSuppliersDataSource`. Regardless of what data source is used, the `SuppliersDetails` mode is reverted back to the read-only mode and the data is rebound to the DetailsView by a call to the `SuppliersDetails` control s `DataBind()` method.

With this event handler in place, the DetailsView control now shows the selected supplier, unless the "Show/Edit ALL Suppliers" option was selected, in which case all of the suppliers can be viewed through the paging interface. Figure 9 shows the page with the "Show/Edit ALL Suppliers" option selected; note that the paging interface is present, allowing the user to visit and update any supplier. Figure 10 shows the page with the Ma Maison supplier selected. Only Ma Maison s information is viewable and editable in this case.

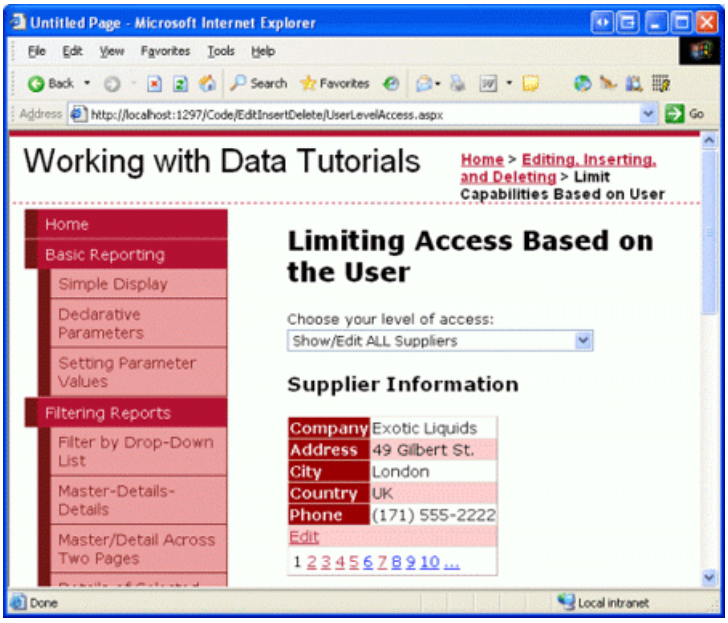


Figure 9: All of the Suppliers Information Can Be Viewed and Edited (Click to view full-size image)

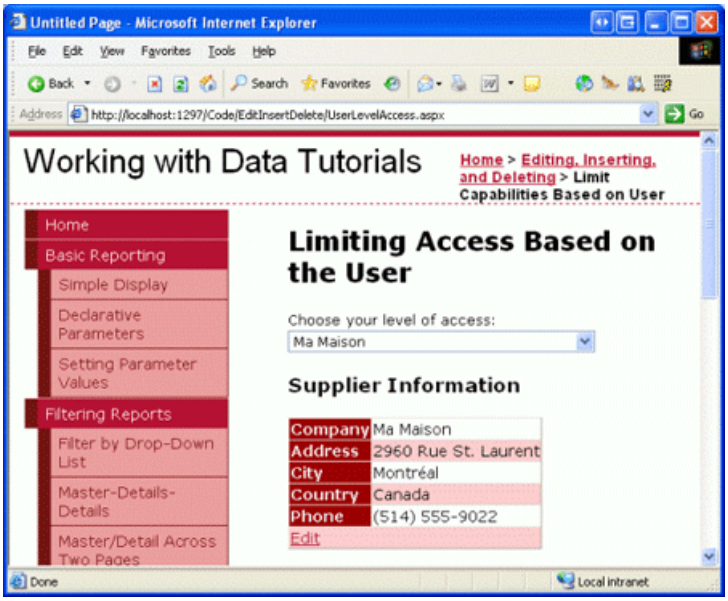


Figure 10: Only the Selected Supplier s Information Can Be Viewed and Edited (Click to view full-size image)

> **NOTE**
>
> For this tutorial, both the DropDownList and DetailsView control s `EnableViewState` must be set to `true` (the default) because the DropDownList s `SelectedIndex` and the DetailsView s `DataSourceID` property s changes must be remembered across postbacks.

Step 4: Listing the Suppliers Products in an Editable GridView

With the DetailsView complete, our next step is to include an editable GridView that lists those products provided by the selected supplier. This GridView should allow edits to only the `ProductName` and `QuantityPerUnit` fields. Moreover, if the user visiting the page is from a particular supplier, it should only allow updates to those products that are *not* discontinued. To accomplish this we'll need to first add an overload of the `ProductsBLL` class s `UpdateProducts` method that takes in just the `ProductID`, `ProductName`, and `QuantityPerUnit` fields as inputs. We ve stepped through this process beforehand in numerous tutorials, so let s just look at the code here, which should be added to `ProductsBLL`:

```
<System.ComponentModel.DataObjectMethodAttribute _
(System.ComponentModel.DataObjectMethodType.Update, False)> _
Public Function UpdateProduct(ByVal productName As String, _
    ByVal quantityPerUnit As String, ByVal productID As Integer) As Boolean
    Dim products As Northwind.ProductsDataTable = Adapter.GetProductByProductID(productID)
    If products.Count = 0 Then
        ' no matching record found, return false
        Return False
    End If
    Dim product As Northwind.ProductsRow = products(0)
    product.ProductName = productName
    If quantityPerUnit Is Nothing Then
        product.SetQuantityPerUnitNull()
    Else
        product.QuantityPerUnit = quantityPerUnit
    End If
    ' Update the product record
    Dim rowsAffected As Integer = Adapter.Update(product)
    ' Return true if precisely one row was updated, otherwise false
    Return rowsAffected = 1
End Function
```

With this overload created, we're ready to add the GridView control and its associated ObjectDataSource. Add a new GridView to the page, set its `ID` property to `ProductsBySupplier`, and configure it to use a new ObjectDataSource named `ProductsBySupplierDataSource`. Since we want this GridView to list those products by the selected supplier, use the `ProductsBLL` class's `GetProductsBySupplierID(supplierID)` method. Also map the `Update()` method to the new `UpdateProduct` overload we just created.

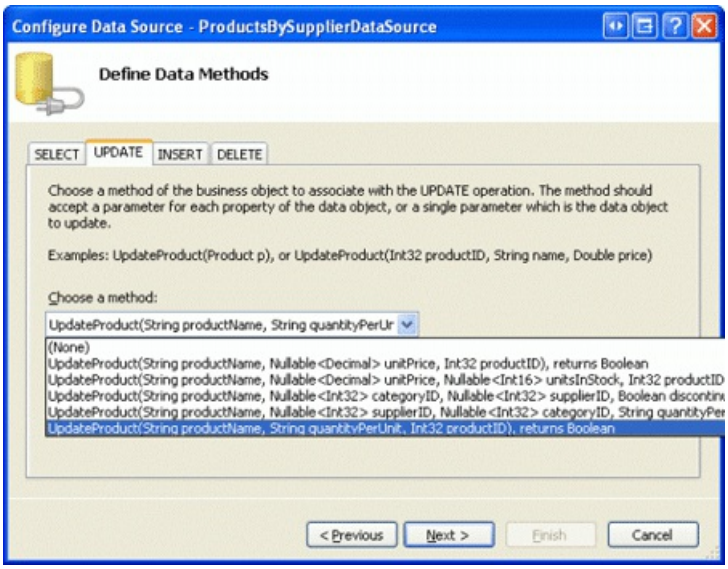


Figure 11: Configure the ObjectDataSource to Use the `UpdateProduct` Overload Just Created (Click to view full-size image)

We're prompted to select the parameter source for the `GetProductsBySupplierID(supplierID)` method's `supplierID` input parameter. Since we want to show the products for the supplier selected in the DetailsView, use the `SuppliersDetails` DetailsView control's `SelectedValue` property as the parameter source.

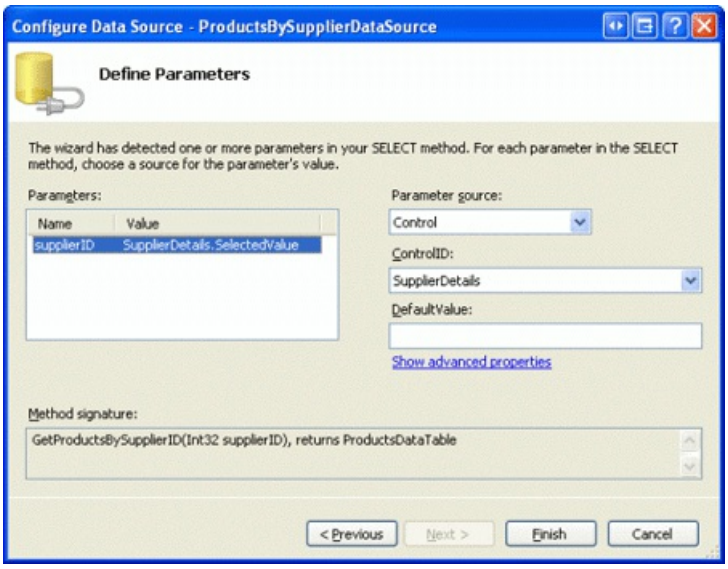


Figure 12: Use the `SuppliersDetails` DetailsView s `SelectedValue` Property as the Parameter Source (Click to view full-size image)

Returning to the GridView, remove all of the GridView fields except for `ProductName`, `QuantityPerUnit`, and `Discontinued`, marking the `Discontinued` CheckBoxField as read-only. Also, check the Enable Editing option from the GridView s smart tag. After these changes have been made, the declarative markup for the GridView and ObjectDataSource should look similar to the following:

```
<asp:GridView ID="ProductsBySupplier" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsBySupplierDataSource">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="QuantityPerUnit" HeaderText="Qty/Unit"
            SortExpression="QuantityPerUnit" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            ReadOnly="True" SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsBySupplierDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}" TypeName="ProductsBLL"
    SelectMethod="GetProductsBySupplierID" UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
    <SelectParameters>
        <asp:ControlParameter ControlID="SupplierDetails" Name="supplierID"
            PropertyName="SelectedValue" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

As with our previous ObjectDataSources, this one s `OldValuesParameterFormatString` property is set to `original_{0}`, which will cause problems when attempting to update a product s name or quantity per unit. Remove this property from the declarative syntax altogether or set it to its default, `{0}`.

With this configuration complete, our page now lists the products provided by the supplier selected in the GridView (see Figure 13). Currently *any* product s name or quantity per unit can be updated. However, we need to update our page logic so that such functionality is prohibited for discontinued products for users associated with a particular supplier. We'll tackle this final piece in Step 5.

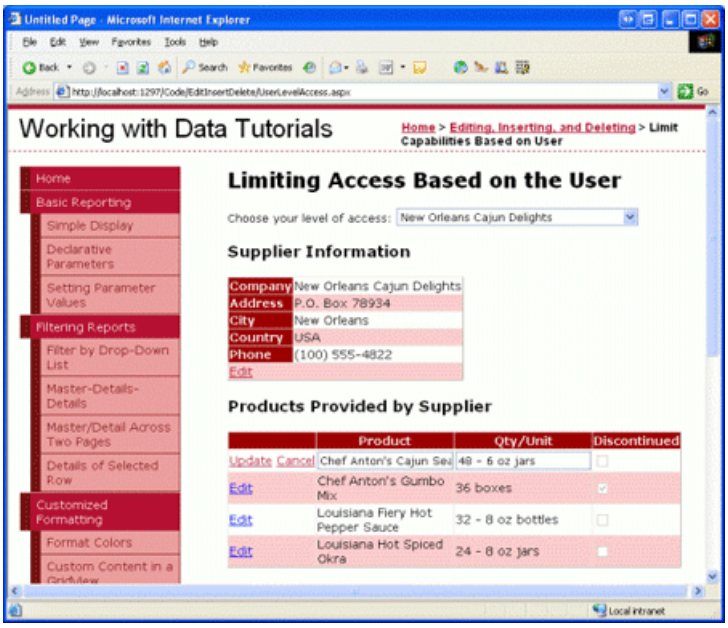


Figure 13: The Products Provided by the Selected Supplier are Displayed (Click to view full-size image)

> **NOTE**
>
> With the addition of this editable GridView the `Suppliers` DropDownList s `SelectedIndexChanged` event handler should be updated to return the GridView to a read-only state. Otherwise, if a different supplier is selected while in the middle of editing product information, the corresponding index in the GridView for the new supplier will also be editable. To prevent this, simply set the GridView s `EditIndex` property to `-1` in the `SelectedIndexChanged` event handler.

Also, recall that it is important that the GridView s view state be enabled (the default behavior). If you set the GridView s `EnableViewState` property to `false`, you run the risk of having concurrent users unintentionally deleting or editing records. See WARNING: Concurrency Issue with ASP.NET 2.0 GridViews/DetailsView/FormViews that Support Editing and/or Deleting and Whose View State is Disabled for more information.

Step 5: Disallow Editing for Discontinued Products When Show/Edit ALL Suppliers is Not Selected

While the `ProductsBySupplier` GridView is fully functional, it currently grants too much access to those users who are from a particular supplier. Per our business rules, such users should not be able to update discontinued products. To enforce this, we can hide (or disable) the Edit button in those GridView rows with discontinued products when the page is being visited by a user from a supplier.

Create an event handler for the GridView s `RowDataBound` event. In this event handler we need to determine whether or not the user is associated with a particular supplier, which, for this tutorial, can be determined by checking the Suppliers DropDownList s `SelectedValue` property - if it s something other than -1, then the user is associated with a particular supplier. For such users, we then need to determine whether or not the product is discontinued. We can grab a reference to the actual `ProductRow` instance bound to the GridView row via the `e.Row.DataItem` property, as discussed in the *Displaying Summary Information in the GridView s Footer* tutorial. If the product is discontinued, we can grab a programmatic reference to the Edit button in the GridView s CommandField using the techniques discussed in the previous tutorial, *Adding Client-Side Confirmation When Deleting*. Once we have a reference we can then hide or disable the button.

```
Protected Sub ProductsBySupplier_RowDataBound _
    (ByVal sender As Object, ByVal e As System.Web.UI.WebControls.GridViewRowEventArgs) _
    Handles ProductsBySupplier.RowDataBound
    If e.Row.RowType = DataControlRowType.DataRow Then
        ' Is this a supplier-specific user?
        If Suppliers.SelectedValue <> "-1" Then
            ' Get a reference to the ProductRow
            Dim product As Northwind.ProductsRow = _
                CType(CType(e.Row.DataItem, System.Data.DataRowView).Row, _
                Northwind.ProductsRow)
            ' Is this product discontinued?
            If product.Discontinued Then
                ' Get a reference to the Edit LinkButton
                Dim editButton As LinkButton = _
                    CType(e.Row.Cells(0).Controls(0), LinkButton)
                ' Hide the Edit button
                editButton.Visible = False
            End If
        End If
    End If
End Sub
```

With this event handler in place, when visiting this page as a user from a specific supplier those products that are discontinued are not editable, as the Edit button is hidden for these products. For example, Chef Anton s Gumbo Mix is a discontinued product for the New Orleans Cajun Delights supplier. When visiting the page for this particular supplier, the Edit button for this product is hidden from sight (see Figure 14). However, when visiting using the "Show/Edit ALL Suppliers," the Edit button is available (see Figure 15).

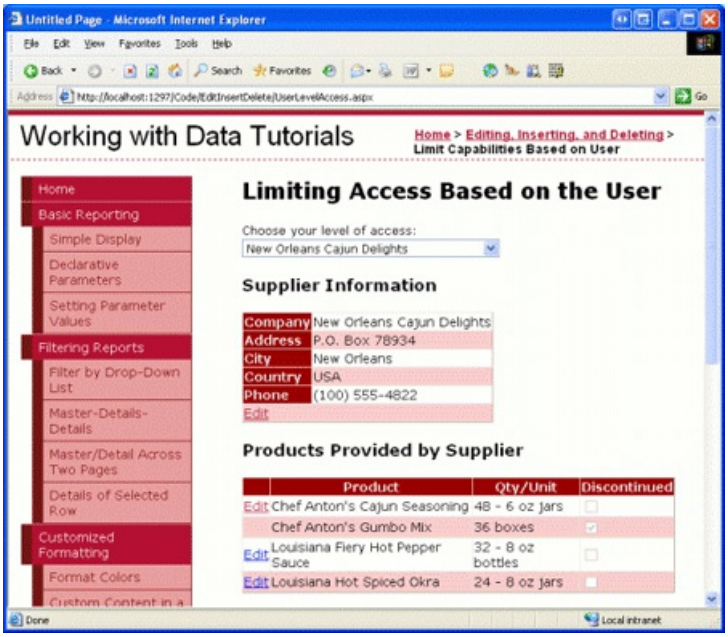


Figure 14: For Supplier-Specific Users the Edit Button for Chef Anton s Gumbo Mix is Hidden (Click to view full-size image)

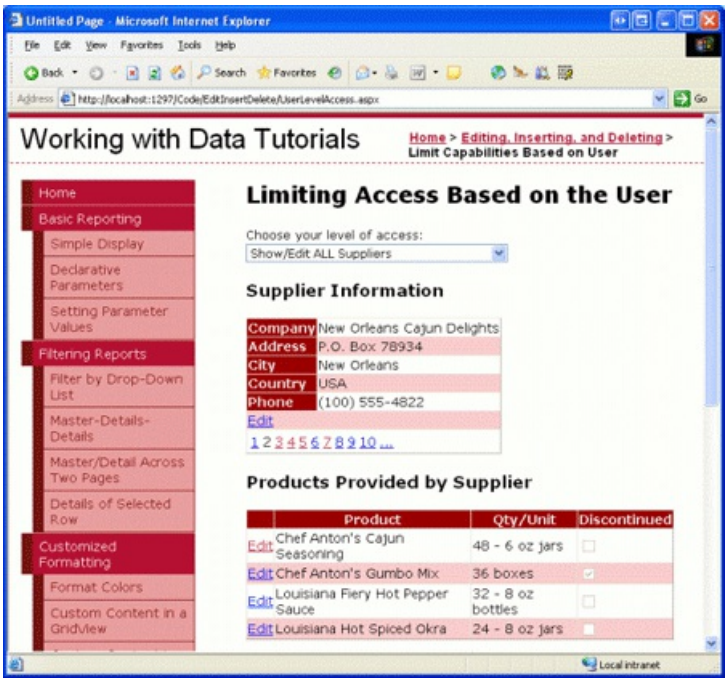


Figure 15: For Show/Edit ALL Suppliers Users, the Edit Button for Chef Anton s Gumbo Mix is Displayed (Click to view full-size image)

Checking for Access Rights in the Business Logic Layer

In this tutorial the ASP.NET page handles all logic with regards to what information the user can see and what products he can update. Ideally, this logic would also be present at the Business Logic Layer. For example, the `SuppliersBLL` class s `GetSuppliers()` method (which returns all suppliers) might include a check to ensure that the currently logged on user is *not* associated with a specific supplier. Likewise, the `UpdateSupplierAddress` method could include a check to ensure that the currently logged on user either worked for our company (and therefore can update all suppliers address information) or is associated with the supplier whose data is being updated.

I did not include such BLL-layer checks here because in our tutorial the user s rights are determined by a DropDownList on the page, which the BLL classes cannot access. When using the membership system or one of the out-of-the box authentication schemes provided by ASP.NET (such as Windows authentication), the currently logged on user s information and roles information can be accessed from the BLL, thereby making such access rights checks possible at both the presentation and BLL layers.

Summary

Most sites that provide user accounts need to customize the data modification interface based upon the logged in user. Administrative users may be able to delete and edit any record, whereas non-administrative users may be limited to only updating or deleting records they created themselves. Whatever the scenario may be, the data Web controls, ObjectDataSource, and Business Logic Layer classes can be extended to add or deny certain functionality based on the logged on user. In this tutorial we saw how to limit the viewable and editable data depending on whether the user was associated with a particular supplier or if they worked for our company.

This tutorial concludes our examination of inserting, updating, and deleting data using the GridView, DetailsView, and FormView controls. Starting with the next tutorial, we'll turn our attention to adding paging and sorting support.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS

Paging and Sorting

2/20/2020 • 2 minutes to read • Edit Online

In these tutorials you add some initial sorting and paging functionality to data reports.

- Paging and Sorting Report Data (C#)
- Efficiently Paging Through Large Amounts of Data (C#)
- Sorting Custom Paged Data (C#)
- Creating a Customized Sorting User Interface (C#)
- Paging and Sorting Report Data (VB)
- Efficiently Paging Through Large Amounts of Data (VB)
- Sorting Custom Paged Data (VB)
- Creating a Customized Sorting User Interface (VB)

Paging and Sorting Report Data (C#)

2/20/2020 • 19 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> Paging and sorting are two very common features when displaying data in an online application. In this tutorial we'll take a first look at adding sorting and paging to our reports, which we will then build upon in future tutorials.

Introduction

Paging and sorting are two very common features when displaying data in an online application. For example, when searching for ASP.NET books at an online bookstore, there may be hundreds of such books, but the report listing the search results lists only ten matches per page. Moreover, the results can be sorted by title, price, page count, author name, and so on. While the past 23 tutorials have examined how to build a variety of reports, including interfaces that permit adding, editing, and deleting data, we ve not looked at how to sort data and the only paging examples we ve seen have been with the DetailsView and FormView controls.

In this tutorial we'll see how to add sorting and paging to our reports, which can be accomplished by simply checking a few checkboxes. Unfortunately, this simplistic implementation has its drawbacks the sorting interface leaves a bit to be desired and the paging routines are not designed for efficiently paging through large result sets. Future tutorials will explore how to overcome the limitations of the out-of-the-box paging and sorting solutions.

Step 1: Adding the Paging and Sorting Tutorial Web Pages

Before we start this tutorial, let s first take a moment to add the ASP.NET pages we'll need for this tutorial and the next three. Start by creating a new folder in the project named `PagingAndSorting`. Next, add the following five ASP.NET pages to this folder, having all of them configured to use the master page `Site.master`:

- `Default.aspx`
- `SimplePagingSorting.aspx`
- `EfficientPaging.aspx`
- `SortParameter.aspx`
- `CustomSortingUI.aspx`

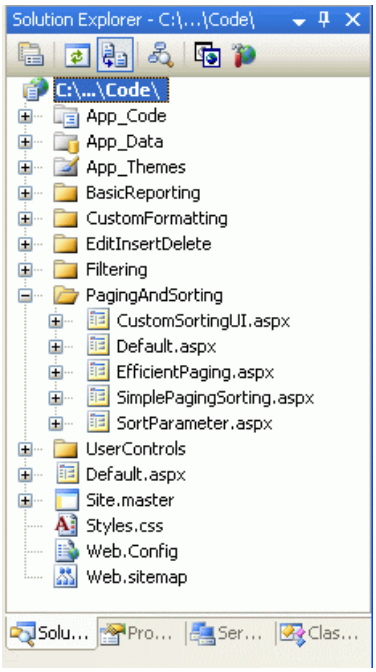


Figure 1: Create a PagingAndSorting Folder and Add the Tutorial ASP.NET Pages

Next, open the `Default.aspx` page and drag the `SectionLevelTutorialListing.ascx` User Control from the `UserControls` folder onto the Design surface. This User Control, which we created in the Master Pages and Site Navigation tutorial, enumerates the site map and displays those tutorials in the current section in a bulleted list.

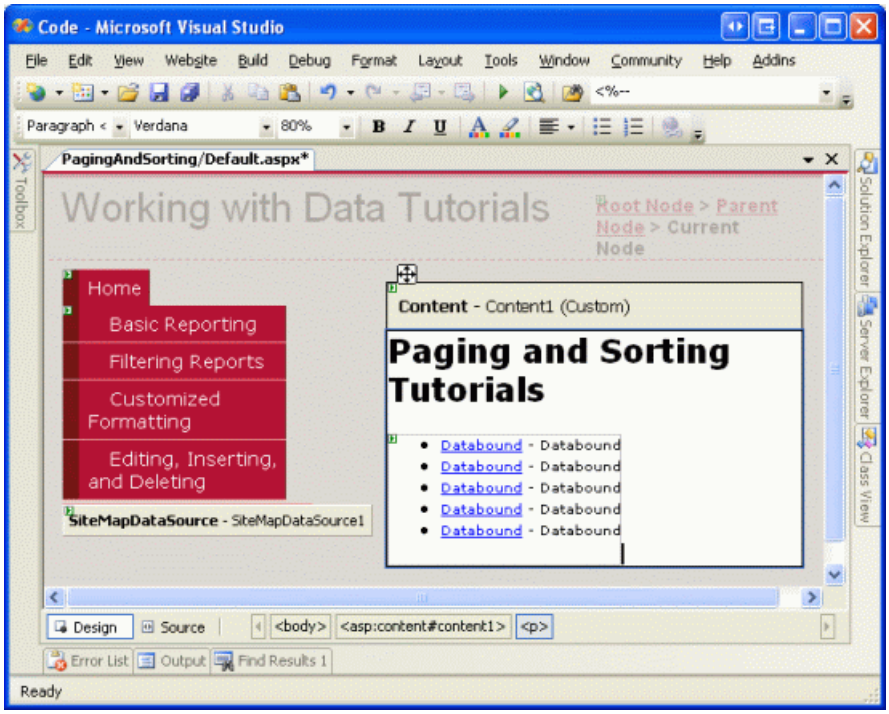


Figure 2: Add the SectionLevelTutorialListing.ascx User Control to Default.aspx

In order to have the bulleted list display the paging and sorting tutorials we'll be creating, we need to add them to the site map. Open the `Web.sitemap` file and add the following markup after the Editing, Inserting, and Deleting site map node markup:

```
<siteMapNode title="Paging and Sorting" url="~/PagingAndSorting/Default.aspx"
    description="Samples of Reports that Provide Paging and Sorting Capabilities">
    <siteMapNode url="~/PagingAndSorting/SimplePagingSorting.aspx"
        title="Simple Paging & Sorting Examples"
        description="Examines how to add simple paging and sorting support." />
    <siteMapNode url="~/PagingAndSorting/EfficientPaging.aspx"
        title="Efficiently Paging Through Large Result Sets"
        description="Learn how to efficiently page through large result sets." />
    <siteMapNode url="~/PagingAndSorting/SortParameter.aspx"
        title="Sorting Data at the BLL or DAL"
        description="Illustrates how to perform sorting logic in the Business Logic
        Layer or Data Access Layer." />
    <siteMapNode url="~/PagingAndSorting/CustomSortingUI.aspx"
        title="Customizing the Sorting User Interface"
        description="Learn how to customize and improve the sorting user interface." />
</siteMapNode>
```

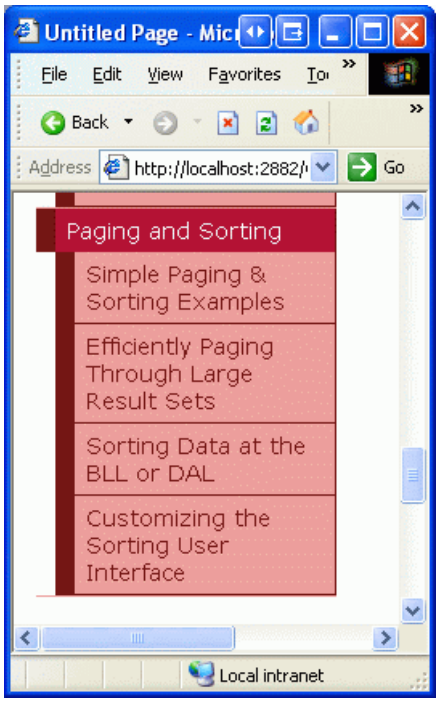


Figure 3: Update the Site Map to Include the New ASP.NET Pages

Step 2: Displaying Product Information in a GridView

Before we actually implement paging and sorting capabilities, let s first create a standard non-sortable, non-pageable GridView that lists the product information. This is a task we ve done many times before throughout this tutorial series so these steps should be familiar. Start by opening the `SimplePagingSorting.aspx` page and drag a GridView control from the Toolbox onto the Designer, setting its `ID` property to `Products`. Next, create a new ObjectDataSource that uses the ProductsBLL class s `GetProducts()` method to return all of the product information.

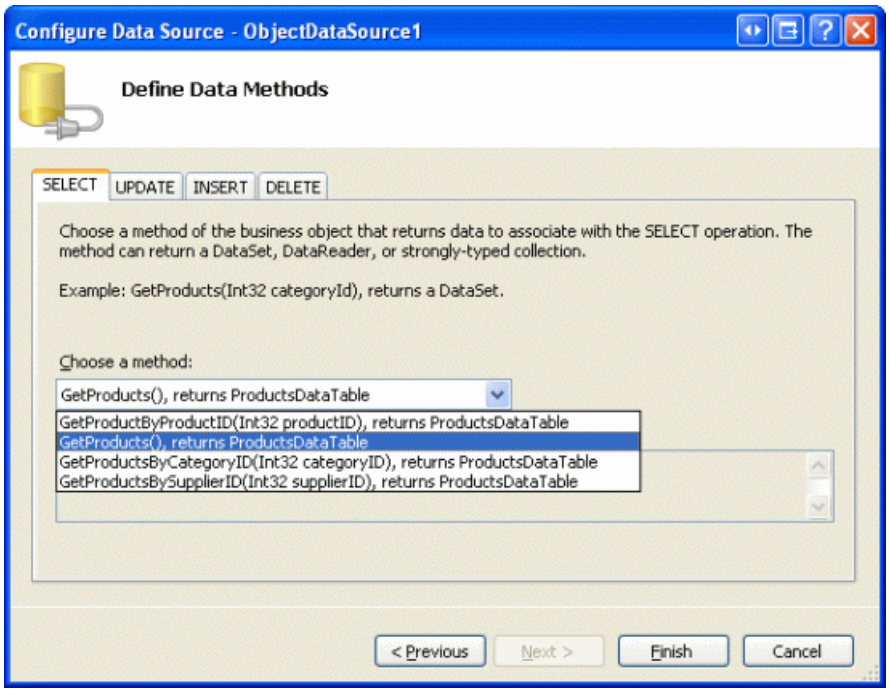


Figure 4: Retrieve Information About All of the Products Using the GetProducts() Method

Since this report is a read-only report, there s no need to map the ObjectDataSource s `Insert()`, `Update()`, or `Delete()` methods to corresponding `ProductsBLL` methods; therefore, choose (None) from the drop-down list for the UPDATE, INSERT, and DELETE tabs.

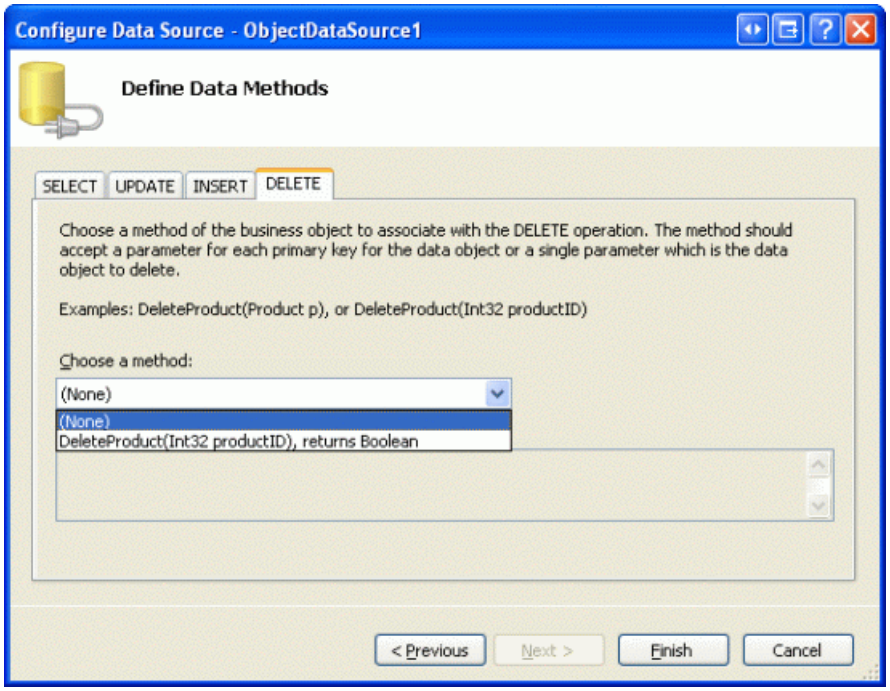


Figure 5: Choose the (None) Option in the Drop-Down List in the UPDATE, INSERT, and DELETE Tabs

Next, let s customize the GridView s fields so that only the products names, suppliers, categories, prices, and discontinued statuses are displayed. Furthermore, feel free to make any field-level formatting changes, such as adjusting the `HeaderText` properties or formatting the price as a currency. After these changes, your GridView s declarative markup should look similar to the following:

```
<asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
            ReadOnly="True" SortExpression="SupplierName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
            SortExpression="UnitPrice" DataFormatString="{0:C}"
            HtmlEncode="False" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
```

Figure 6 shows our progress thus far when viewed through a browser. Note that the page lists all of the products in one screen, showing each product s name, category, supplier, price, and discontinued status.

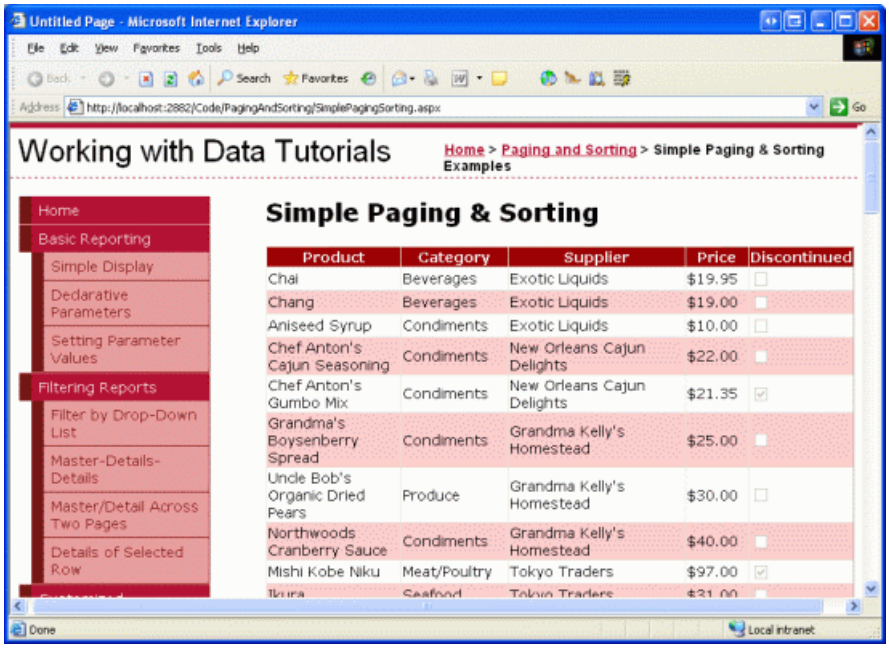


Figure 6: Each of the Products are Listed (Click to view full-size image)

Step 3: Adding Paging Support

Listing *all* of the products on one screen can lead to information overload for the user perusing the data. To help make the results more manageable, we can break up the data into smaller pages of data and allow the user to step through the data one page at a time. To accomplish this simply check the Enable Paging checkbox from the GridView s smart tag (this sets the GridView s `AllowPaging` property to `true`).

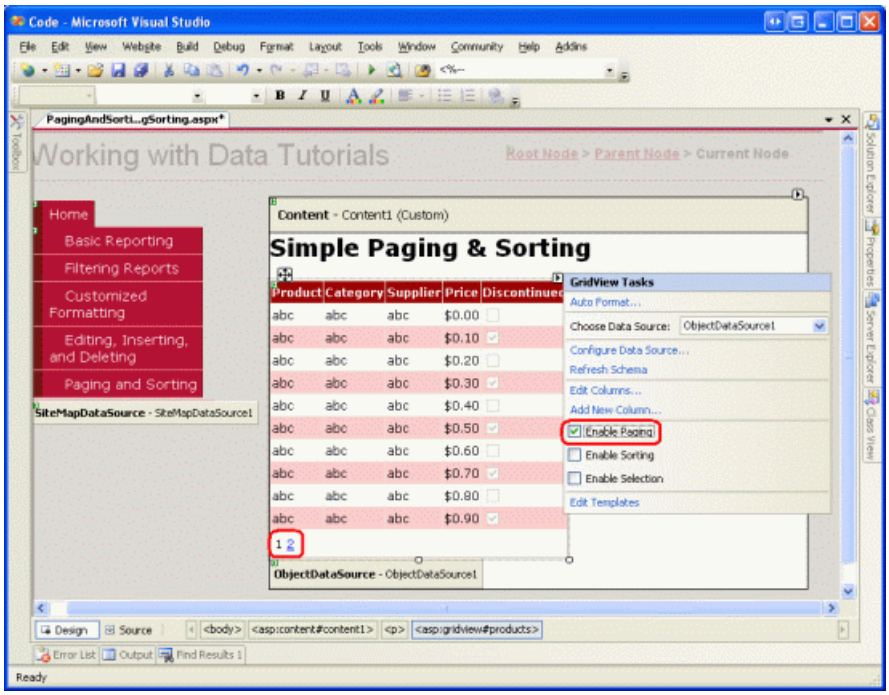


Figure 7: Check the Enable Paging Checkbox to Add Paging Support (Click to view full-size image)

Enabling paging limits the number of records shown per page and adds a *paging interface* to the GridView. The default paging interface, shown in Figure 7, is a series of page numbers, allowing the user to quickly navigate from one page of data to another. This paging interface should look familiar, as we've seen it when adding paging support to the DetailsView and FormView controls in past tutorials.

Both the DetailsView and FormView controls only show a single record per page. The GridView, however, consults its `PageSize` property to determine how many records to show per page (this property defaults to a value of 10).

This GridView, DetailsView, and FormView's paging interface can be customized using the following properties:

- `PagerStyle` indicates the style information for the paging interface; can specify settings like `BackColor`, `ForeColor`, `CssClass`, `HorizontalAlign`, and so on.
- `PagerSettings` contains a bevy of properties that can customize the functionality of the paging interface; `PageButtonCount` indicates the maximum number of numeric page numbers displayed in the paging interface (the default is 10); the `Mode` property indicates how the paging interface operates and can be set to:
  - `NextPrevious` shows a Next and Previous buttons, allowing the user to step forwards or backwards one page at a time
  - `NextPreviousFirstLast` in addition to Next and Previous buttons, First and Last buttons are also included, allowing the user to quickly move to the first or last page of data
  - `Numeric` shows a series of page numbers, allowing the user to immediately jump to any page
  - `NumericFirstLast` in addition to the page numbers, includes First and Last buttons, allowing the user to quickly move to the first or last page of data; the First/Last buttons are only shown if all of the numeric page numbers cannot fit

Moreover, the GridView, DetailsView, and FormView all offer the `PageIndex` and `PageCount` properties, which indicate the current page being viewed and the total number of pages of data, respectively. The `PageIndex` property is indexed starting at 0, meaning that when viewing the first page of data `PageIndex` will equal 0. `PageCount`, on the other hand, starts counting at 1, meaning that `PageIndex` is limited to the values between 0 and `PageCount - 1`.

Let's take a moment to improve the default appearance of our GridView's paging interface. Specifically, let's have

the paging interface right-aligned with a light gray background. Rather than setting these properties directly through the GridView s `PagerStyle` property, let s create a CSS class in `Styles.css` named `PagerRowStyle` and then assign the `PagerStyle` s `CssClass` property through our Theme. Start by opening `Styles.css` and adding the following CSS class definition:

```
.PagerRowStyle
{
    background-color: #ddd;
    text-align: right;
}
```

Next, open the `GridView.skin` file in the `DataWebControls` folder within the `App_Themes` folder. As we discussed in the *Master Pages and Site Navigation* tutorial, Skin files can be used to specify the default property values for a Web control. Therefore, augment the existing settings to include setting the `PagerStyle` s `CssClass` property to `PagerRowStyle`. Also, let s configure the paging interface to show at most five numeric page buttons using the `NumericFirstLast` paging interface.

```
<asp:GridView runat="server" CssClass="DataWebControlStyle">
   <AlternatingRowStyle CssClass="AlternatingRowStyle" />
   <RowStyle CssClass="RowStyle" />
   <HeaderStyle CssClass="HeaderStyle" />
   <FooterStyle CssClass="FooterStyle" />
   <SelectedRowStyle CssClass="SelectedRowStyle" />
   <PagerStyle CssClass="PagerRowStyle" />
   <PagerSettings Mode="NumericFirstLast" PageButtonCount="5" />
</asp:GridView>
```

The Paging User Experience

Figure 8 shows the web page when visited through a browser after the GridView s Enable Paging checkbox has been checked and the `PagerStyle` and `PagerSettings` configurations have been made through the `GridView.skin` file. Note how only ten records are shown, and the paging interface indicates that we are viewing the first page of data.

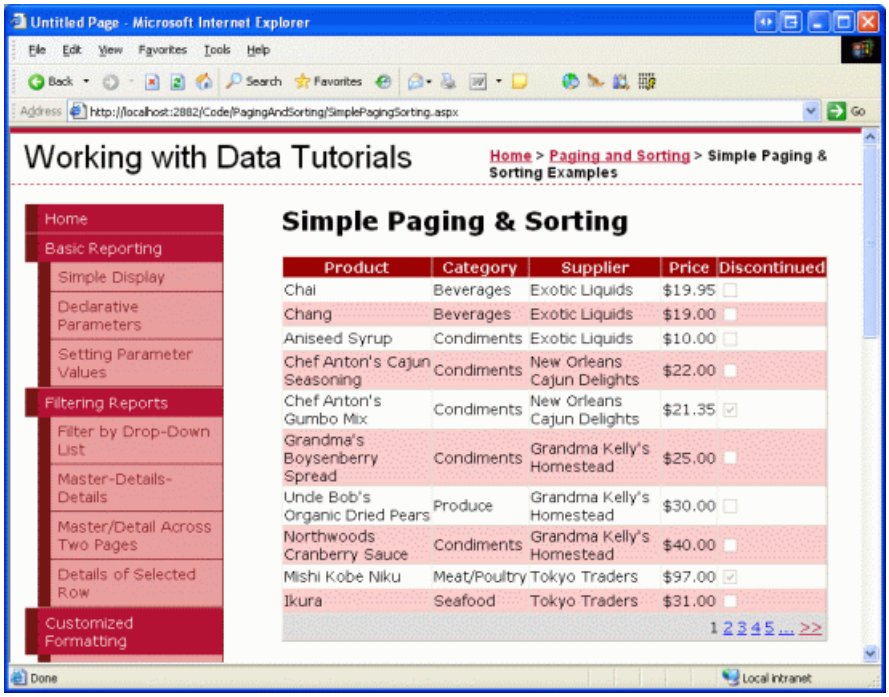


Figure 8: With Paging Enabled, Only a Subset of the Records are Displayed at a Time (Click to view full-size image)

When the user clicks on one of the page numbers in the paging interface, a postback ensues and the page reloads showing that requested page s records. Figure 9 shows the results after opting to view the final page of data. Notice that the final page only has one record; this is because there are 81 records in total, resulting in eight pages of 10 records per page plus one page with a lone record.

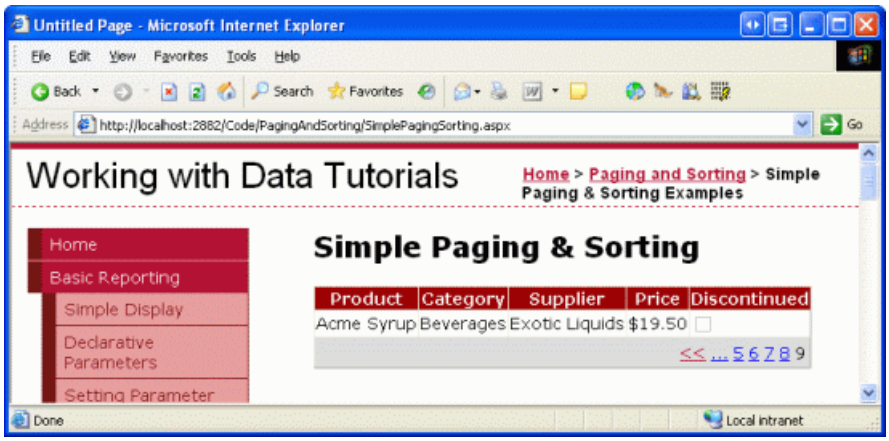


Figure 9: Clicking On a Page Number Causes a Postback and Shows the Appropriate Subset of Records (Click to view full-size image)

Paging s Server-Side Workflow

When the end user clicks on a button in the paging interface, a postback ensues and the following server-side workflow begins:

1. The GridView s (or DetailsView or FormView) `PageIndexChanging` event fires
2. The ObjectDataSource re-requests *all* of the data from the BLL; the GridView s `PageIndex` and `PageSize` property values are used to determine what records returned from the BLL need to be displayed in the GridView
3. The GridView s `PageIndexChanged` event fires

In Step 2, the ObjectDataSource re-requests all of the data from its data source. This style of paging is commonly referred to as *default paging*, as it s the paging behavior used by default when setting the `AllowPaging` property to `true`. With default paging the data Web control naively retrieves all records for each page of data, even though only a subset of records are actually rendered into the HTML that s sent to the browser. Unless the database data is cached by the BLL or ObjectDataSource, default paging is unworkable for sufficiently large result sets or for web applications with many concurrent users.

In the next tutorial we'll examine how to implement *custom paging*. With custom paging you can specifically instruct the ObjectDataSource to only retrieve the precise set of records needed for the requested page of data. As you can imagine, custom paging greatly improves the efficiency of paging through large result sets.

> **NOTE**
> While default paging is not suitable when paging through sufficiently large result sets or for sites with many simultaneous users, realize that custom paging requires more changes and effort to implement and is not as simple as checking a checkbox (as is default paging). Therefore, default paging may be the ideal choice for small, low-traffic websites or when paging through relatively small result sets, as it s much easier and quicker to implement.

For example, if we know that we'll never have more than 100 products in our database, the minimal performance gain enjoyed by custom paging is likely offset by the effort required to implement it. If, however, we may one day have thousands or tens of thousands of products, *not* implementing custom paging would greatly hamper the scalability of our application.

Step 4: Customizing the Paging Experience

The data Web controls provide a number of properties that can be used to enhance the user s paging experience. The `PageCount` property, for example, indicates how many total pages there are, while the `PageIndex` property indicates the current page being visited and can be set to quickly move a user to a specific page. To illustrate how to use these properties to improve upon the user s paging experience, let s add a Label Web control to our page that informs the user what page they re currently visiting, along with a DropDownList control that allows them to quickly jump to any given page.

First, add a Label Web control to your page, set its `ID` property to `PagingInformation`, and clear out its `Text` property. Next, create an event handler for the GridView s `DataBound` event and add the following code:

```
protected void Products_DataBound(object sender, EventArgs e)
{
    PagingInformation.Text = string.Format("You are viewing page {0} of {1}...",
        Products.PageIndex + 1, Products.PageCount);
}
```

This event handler assigns the `PagingInformation` Label s `Text` property to a message informing the user the page they are currently visiting `Products.PageIndex + 1` out of how many total pages `Products.PageCount` (we add 1 to the `Products.PageIndex` property because `PageIndex` is indexed starting at 0). I chose the assign this Label s `Text` property in the `DataBound` event handler as opposed to the `PageIndexChanged` event handler because the `DataBound` event fires every time data is bound to the GridView whereas the `PageIndexChanged` event handler only fires when the page index is changed. When the GridView is initially data bound on the first page visit, the `PageIndexChanging` event doesn t fire (whereas the `DataBound` event does).

With this addition, the user is now shown a message indicating what page they are visiting and how many total pages of data there are.

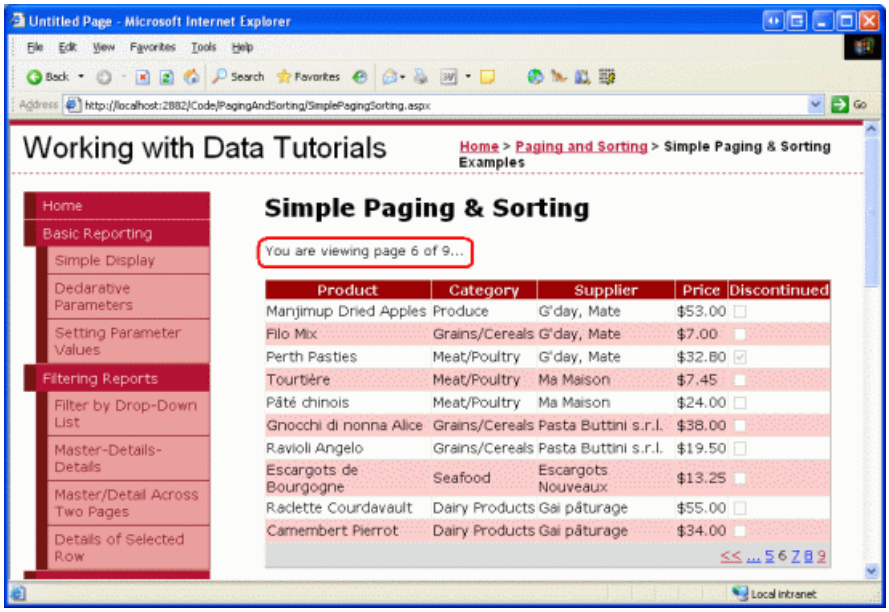


Figure 10: The Current Page Number and Total Number of Pages are Displayed (Click to view full-size image)

In addition to the Label control, let s also add a DropDownList control that lists the page numbers in the GridView with the currently viewed page selected. The idea here is that the user can quickly jump from the current page to another by simply selecting the new page index from the DropDownList. Start by adding a DropDownList to the Designer, setting its `ID` property to `PageList` and checking the Enable AutoPostBack option from its smart tag.

Next, return to the `DataBound` event handler and add the following code:

```
// Clear out all of the items in the DropDownList
PageList.Items.Clear();
// Add a ListItem for each page
for (int i = 0; i < Products.PageCount; i++)
{
    // Add the new ListItem
    ListItem pageListItem = new ListItem(string.Concat("Page ", i + 1), i.ToString());
    PageList.Items.Add(pageListItem);
    // select the current item, if needed
    if (i == Products.PageIndex)
        pageListItem.Selected = true;
}
```

This code begins by clearing out the items in the `PageList` DropDownList. This may seem superfluous, since one wouldn't expect the number of pages to change, but other users may be using the system simultaneously, adding or removing records from the `Products` table. Such insertions or deletions could alter the number of pages of data.

Next, we need to create the page numbers again and have the one that maps to the current GridView `PageIndex` selected by default. We accomplish this with a loop from 0 to `PageCount - 1`, adding a new `ListItem` in each iteration and setting its `Selected` property to true if the current iteration index equals the GridView s `PageIndex` property.

Finally, we need to create an event handler for the DropDownList s `SelectedIndexChanged` event, which fires each time the user pick a different item from the list. To create this event handler, simply double-click the DropDownList in the Designer, then add the following code:

```
protected void PageList_SelectedIndexChanged(object sender, EventArgs e)
{
    // Jump to the specified page
    Products.PageIndex = Convert.ToInt32(PageList.SelectedValue);
}
```

As Figure 11 shows, merely changing the GridView s `PageIndex` property causes the data to be rebound to the GridView. In the GridView s `DataBound` event handler, the appropriate DropDownList `ListItem` is selected.

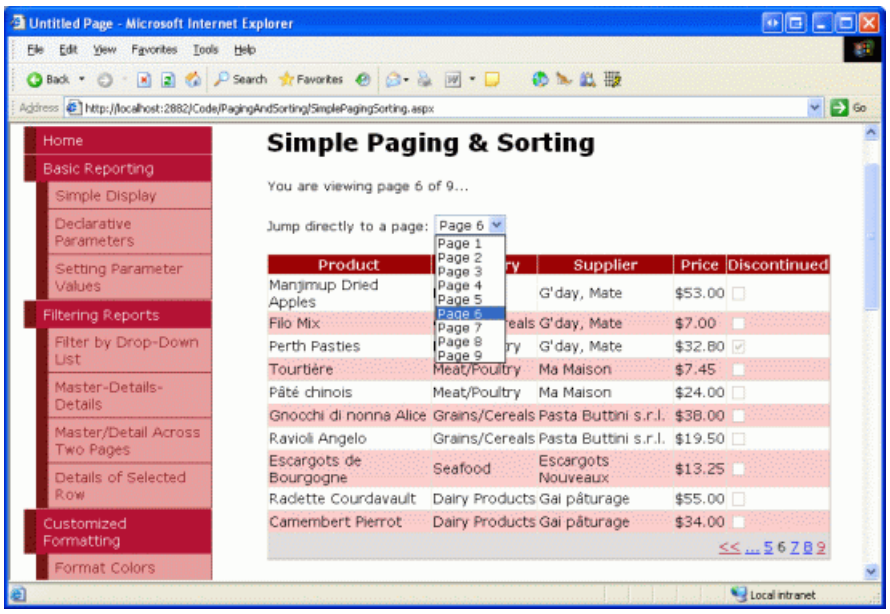


Figure 11: The User is Automatically Taken to the Sixth Page When Selecting the Page 6 Drop-Down List Item (Click to view full-size image)

Step 5: Adding Bi-Directional Sorting Support

Adding bi-directional sorting support is as simple as adding paging support simply check the Enable Sorting option from the GridView s smart tag (which sets the GridView s `AllowSorting` property to `true`). This renders each of the headers of the GridView s fields as LinkButtons that, when clicked, cause a postback and return the data sorted by the clicked column in ascending order. Clicking the same header LinkButton again re-sorts the data in descending order.

> **NOTE**
>
> If you are using a custom Data Access Layer rather than a Typed DataSet, you may not have an Enable Sorting option in the GridView s smart tag. Only GridViews bound to data sources that natively support sorting have this checkbox available. The Typed DataSet provides out-of-the-box sorting support since the ADO.NET DataTable provides a `Sort` method that, when invoked, sorts the DataTable s DataRows using the criteria specified.

If your DAL does not return objects that natively support sorting you will need to configure the ObjectDataSource to pass sorting information to the Business Logic Layer, which can either sort the data or have the data sorted by the DAL. We'll explore how to sort data at the Business Logic and Data Access Layers in a future tutorial.

The sorting LinkButtons are rendered as HTML hyperlinks, whose current colors (blue for an unvisited link and a dark red for a visited link) clash with the background color of the header row. Instead, let s have all header row links displayed in white, regardless of whether they ve been visited or not. This can be accomplished by adding the following to the `Styles.css` class:

```
.HeaderStyle a, .HeaderStyle a:visited
{
    color: White;
}
```

This syntax indicates to use white text when displaying those hyperlinks within an element that uses the HeaderStyle class.

After this CSS addition, when visiting the page through a browser your screen should look similar to Figure 12. In particular, Figure 12 shows the results after the Price field s header link has been clicked.

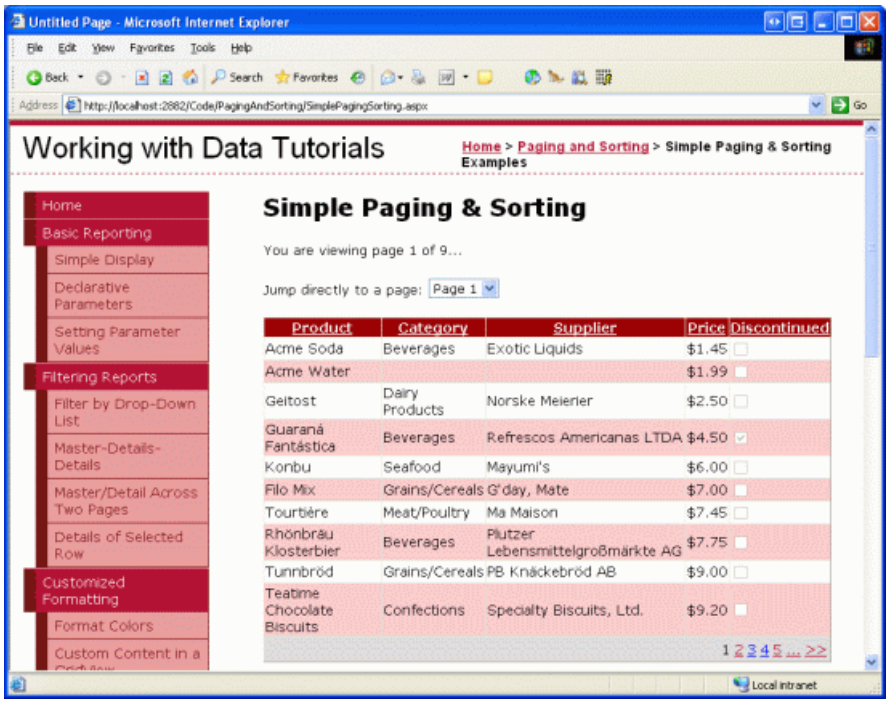


Figure 12: The Results Have Been Sorted by the UnitPrice in Ascending Order (Click to view full-size image)

Examining the Sorting Workflow

All GridView fields the BoundField, CheckBoxField, TemplateField, and so on have a `SortExpression` property that indicates the expression that should be used to sort the data when that field s sorting header link is clicked. The GridView also has a `SortExpression` property. When a sorting header LinkButton is clicked, the GridView assigns that field s `SortExpression` value to its `SortExpression` property. Next, the data is re-retrieved from the ObjectDataSource and sorted according to the GridView s `SortExpression` property. The following list details the sequence of steps that transpires when an end user sorts the data in a GridView:

1. The GridView s Sorting event fires
2. The GridView s `SortExpression` property is set to the `SortExpression` of the field whose sorting header LinkButton was clicked
3. The ObjectDataSource re-retrieves all of the data from the BLL and then sorts the data using the GridView s `SortExpression`
4. The GridView s `PageIndex` property is reset to 0, meaning that when sorting the user is returned to the first page of data (assuming paging support has been implemented)
5. The GridView s `Sorted` event fires

Like with default paging, the default sorting option re-retrieves *all* of the records from the BLL. When using sorting without paging or when using sorting with default paging, there s no way to circumvent this performance hit (short of caching the database data). However, as we'll see in a future tutorial, it s possible to efficiently sort data when using custom paging.

When binding an ObjectDataSource to the GridView through the drop-down list in the GridView s smart tag, each GridView field automatically has its `SortExpression` property assigned to the name of the data field in the `ProductsRow` class. For example, the `ProductName` BoundField s `SortExpression` is set to `ProductName`, as shown in the following declarative markup:

```
<asp:BoundField DataField="ProductName" HeaderText="Product"
    SortExpression="ProductName" />
```

A field can be configured so that it s not sortable by clearing out its `SortExpression` property (assigning it to an empty string). To illustrate this, imagine that we didn't want to let our customers sort our products by price. The `UnitPrice` BoundField s `SortExpression` property can be removed either from the declarative markup or through the Fields dialog box (which is accessible by clicking on the Edit Columns link in the GridView s smart tag).

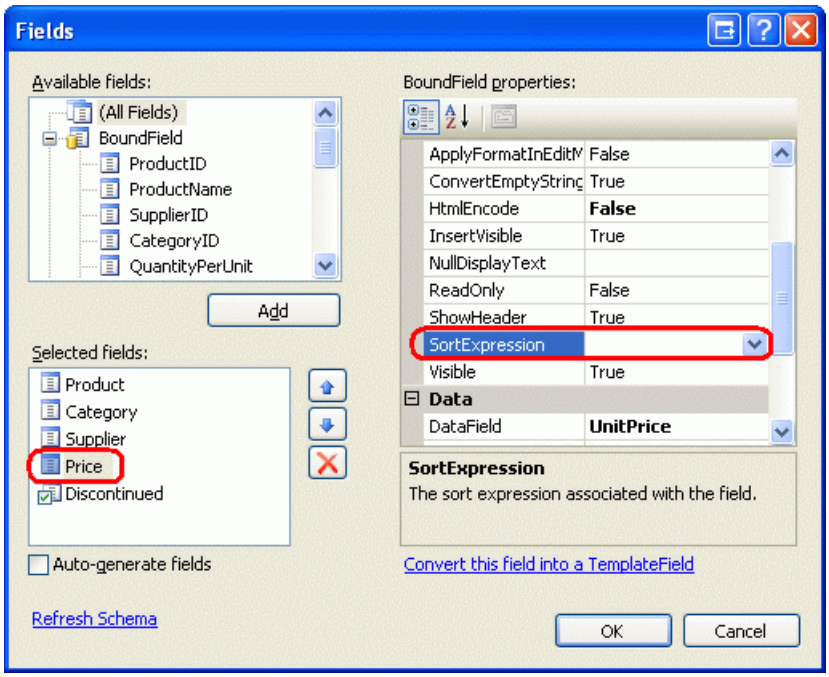


Figure 13: The Results Have Been Sorted by the UnitPrice in Ascending Order

Once the `SortExpression` property has been removed for the `UnitPrice` BoundField, the header is rendered as text rather than as a link, thereby preventing users from sorting the data by price.

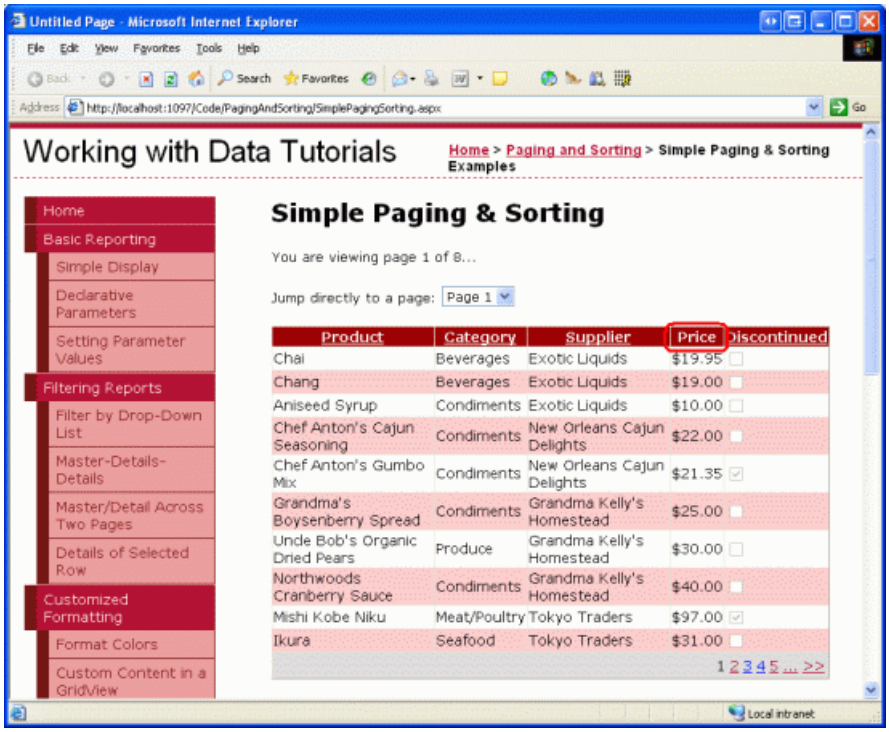


Figure 14: By Removing the SortExpression Property, Users Can No Longer Sort the Products By Price (Click to view full-size image)

Programmatically Sorting the GridView

You can also sort the contents of the GridView programmatically by using the GridView s `Sort` method. Simply pass in the `SortExpression` value to sort by along with the `SortDirection` (`Ascending` or `Descending`), and the GridView s data will be re-sorted.

Imagine that the reason we turned off sorting by the `UnitPrice` was because we were worried that our customers would simply buy only the lowest-priced products. However, we want to encourage them to buy the most expensive products, so we d like them to be able to sort the products by price, but only from the most

expensive price to the least.

To accomplish this add a Button Web control to the page, set its `ID` property to `SortPriceDescending`, and its `Text` property to Sort by Price. Next, create an event handler for the Button s `Click` event by double-clicking the Button control in the Designer. Add the following code to this event handler:

```
protected void SortPriceDescending_Click(object sender, EventArgs e)
{
    // Sort by UnitPrice in descending order
    Products.Sort("UnitPrice", SortDirection.Descending);
}
```

Clicking this Button returns the user to the first page with the products sorted by price, from most expensive to least expensive (see Figure 15).

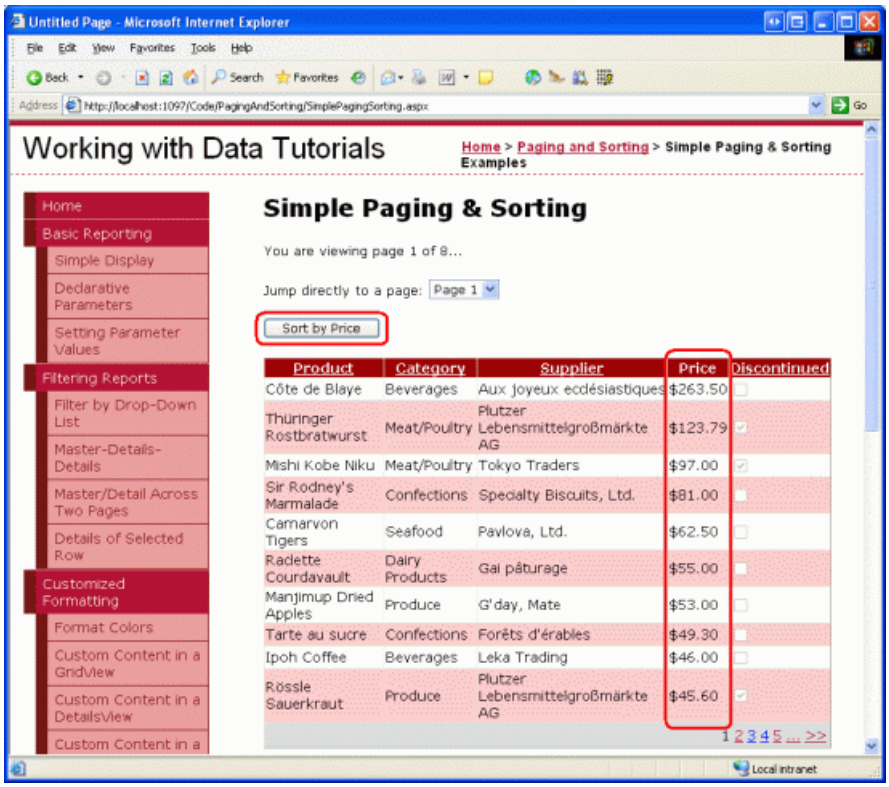


Figure 15: Clicking the Button Orders the Products From the Most Expensive to the Least (Click to view full-size image)

Summary

In this tutorial we saw how to implement default paging and sorting capabilities, both of which were as easy as checking a checkbox! When a user sorts or pages through data, a similar workflow unfolds:

1. A postback ensues
2. The data Web control s pre-level event fires (`PageIndexChanging` or `Sorting`)
3. All of the data is re-retrieved by the ObjectDataSource
4. The data Web control s post-level event fires (`PageIndexChanged` or `Sorted`)

While implementing basic paging and sorting is a breeze, more effort must be exerted to utilize the more efficient custom paging or to further enhance the paging or sorting interface. Future tutorials will explore these topics.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

NEXT

Efficiently Paging Through Large Amounts of Data (C#)

9/16/2020 • 29 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> The default paging option of a data presentation control is unsuitable when working with large amounts of data, as its underlying data source control retrieves all records, even though only a subset of data is displayed. In such circumstances, we must turn to custom paging.

Introduction

As we discussed in the preceding tutorial, paging can be implemented in one of two ways:

- **Default Paging** can be implemented by simply checking the Enable Paging option in the data Web control s smart tag; however, whenever viewing a page of data, the ObjectDataSource retrieves *all* of the records, even though only a subset of them are displayed in the page
- **Custom Paging** improves the performance of default paging by retrieving only those records from the database that need to be displayed for the particular page of data requested by the user; however, custom paging involves a bit more effort to implement than default paging

Due to the ease of implementation just check a checkbox and you re done! default paging is an attractive option. Its naive approach in retrieving all of the records, though, makes it an implausible choice when paging through sufficiently large amounts of data or for sites with many concurrent users. In such circumstances, we must turn to custom paging in order to provide a responsive system.

The challenge of custom paging is being able to write a query that returns the precise set of records needed for a particular page of data. Fortunately, Microsoft SQL Server 2005 provides a new keyword for ranking results, which enables us to write a query that can efficiently retrieve the proper subset of records. In this tutorial we'll see how to use this new SQL Server 2005 keyword to implement custom paging in a GridView control. While the user interface for custom paging is identical to that for default paging, stepping from one page to the next using custom paging can be several orders of magnitude faster than default paging.

> **NOTE**
> The exact performance gain exhibited by custom paging depends on the total number of records being paged through and the load being placed on the database server. At the end of this tutorial we'll look at some rough metrics that showcase the benefits in performance obtained through custom paging.

Step 1: Understanding the Custom Paging Process

When paging through data, the precise records displayed in a page depend upon the page of data being requested and the number of records displayed per page. For example, imagine that we wanted to page through the 81 products, displaying 10 products per page. When viewing the first page, we d want products 1 through 10; when viewing the second page we d be interested in products 11 through 20, and so on.

There are three variables that dictate what records need to be retrieved and how the paging interface should be rendered:

- **Start Row Index** the index of the first row in the page of data to display; this index can be calculated by multiplying the page index by the records to display per page and adding one. For example, when paging through records 10 at a time, for the first page (whose page index is 0), the Start Row Index is 0 * 10 + 1, or 1; for the second page (whose page index is 1), the Start Row Index is 1 * 10 + 1, or 11.
- **Maximum Rows** the maximum number of records to display per page. This variable is referred to as maximum rows since for the last page there may be fewer records returned than the page size. For example, when paging through the 81 products 10 records per page, the ninth and final page will have just one record. No page, though, will show more records than the Maximum Rows value.
- **Total Record Count** the total number of records being paged through. While this variable isn t needed to determine what records to retrieve for a given page, it does dictate the paging interface. For example, if there are 81 products being paged through, the paging interface knows to display nine page numbers in the paging UI.

With default paging, the Start Row Index is computed as the product of the page index and the page size plus one, whereas the Maximum Rows is simply the page size. Since default paging retrieves all of the records from the database when rendering any page of data, the index for each row is known, thereby making moving to Start Row Index row a trivial task. Moreover, the Total Record Count is readily available, as it s simply the number of records in the DataTable (or whatever object is being used to hold the database results).

Given the Start Row Index and Maximum Rows variables, a custom paging implementation must only return the precise subset of records starting at the Start Row Index and up to Maximum Rows number of records after that. Custom paging provides two challenges:

- We must be able to efficiently associate a row index with each row in the entire data being paged through so that we can start returning records at the specified Start Row Index
- We need to provide the total number of records being paged through

In the next two steps we'll examine the SQL script needed to respond to these two challenges. In addition to the SQL script, we'll also need to implement methods in the DAL and BLL.

Step 2: Returning the Total Number of Records Being Paged Through

Before we examine how to retrieve the precise subset of records for the page being displayed, let s first look at how to return the total number of records being paged through. This information is needed in order to properly configure the paging user interface. The total number of records returned by a particular SQL query can be obtained by using the `COUNT` aggregate function. For example, to determine the total number of records in the `Products` table, we can use the following query:

```
SELECT COUNT(*)
FROM Products
```

Let s add a method to our DAL that returns this information. In particular, we'll create a DAL method called `TotalNumberOfProducts()` that executes the `SELECT` statement shown above.

Start by opening the `Northwind.xsd` Typed DataSet file in the `App_Code/DAL` folder. Next, right-click on the `ProductsTableAdapter` in the Designer and choose Add Query. As we ve seen in previous tutorials, this will allow us to add a new method to the DAL that, when invoked, will execute a particular SQL statement or stored procedure. As with our TableAdapter methods in previous tutorials, for this one opt to use an ad-hoc SQL statement.

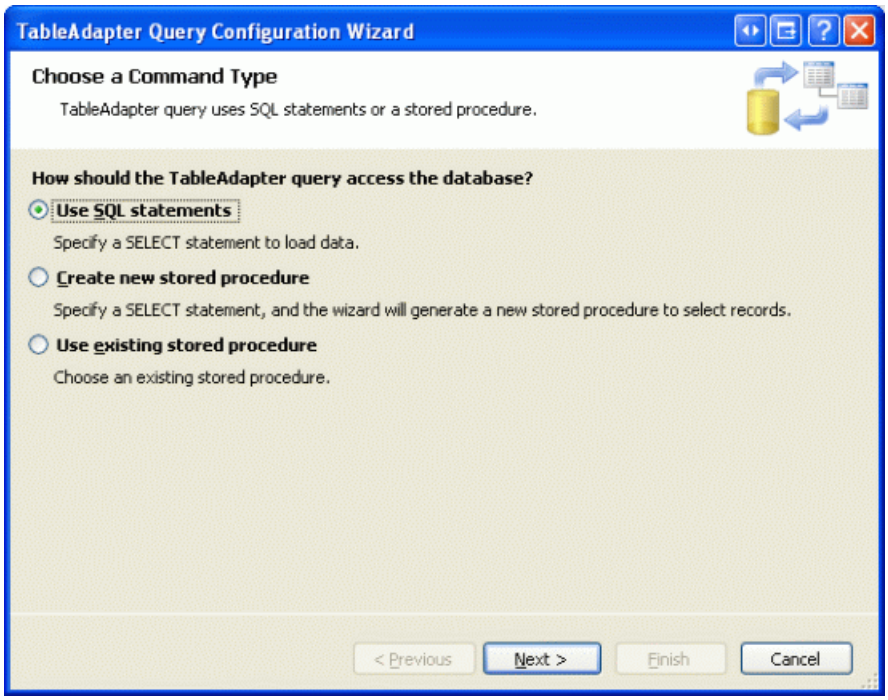


Figure 1: Use an Ad-Hoc SQL Statement

On the next screen we can specify what type of query to create. Since this query will return a single, scalar value the total number of records in the `Products` table choose the `SELECT` which returns a singe value option.

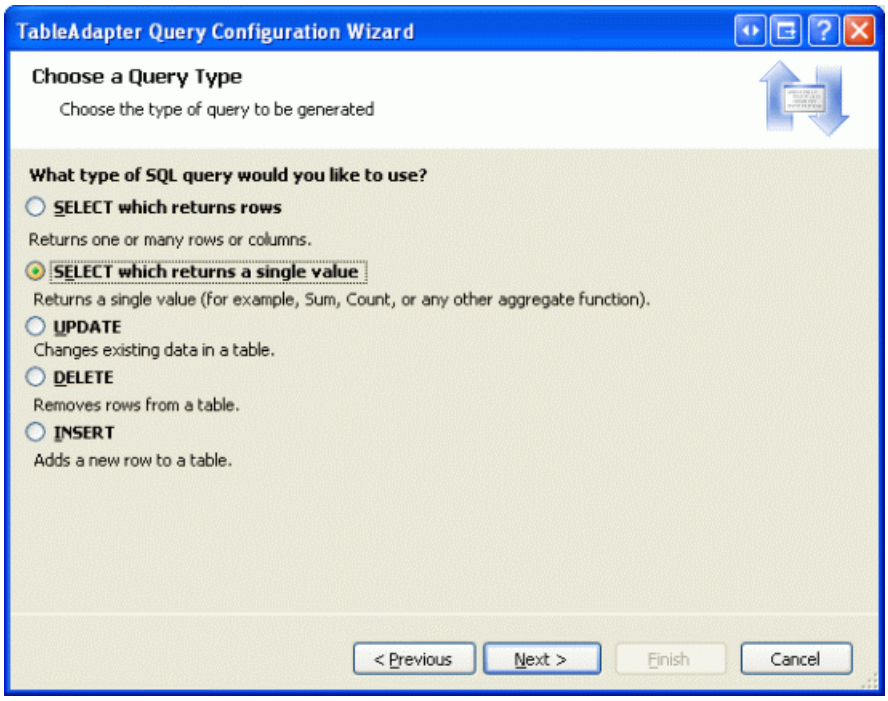


Figure 2: Configure the Query to Use a SELECT Statement that Returns a Single Value

After indicating the type of query to use, we must next specify the query.

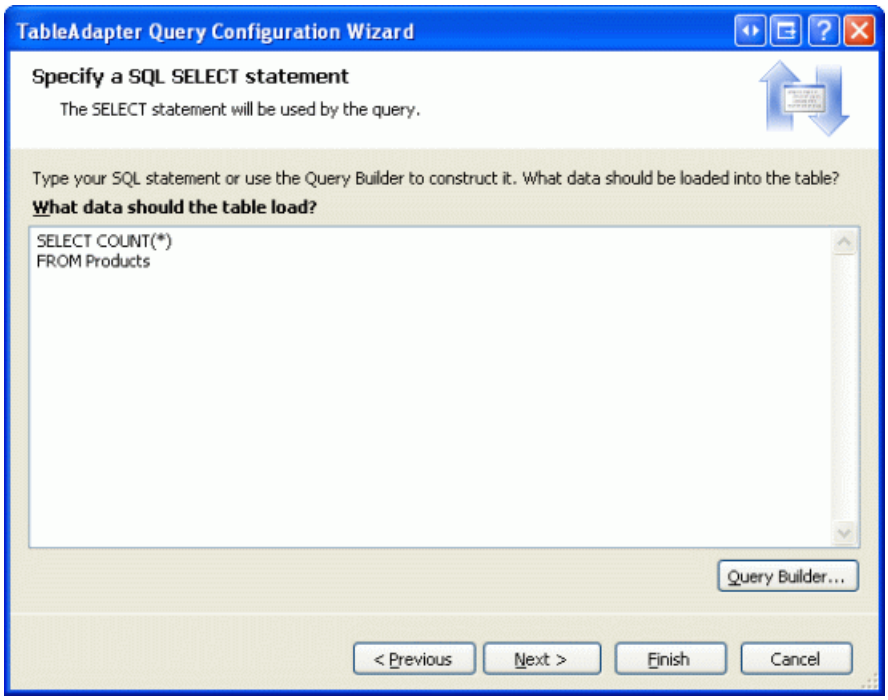


Figure 3: Use the SELECT COUNT(*) FROM Products Query

Finally, specify the name for the method. As aforementioned, let s use `TotalNumberOfProducts`.

Figure 4: Name the DAL Method TotalNumberOfProducts

After clicking Finish, the wizard will add the `TotalNumberOfProducts` method to the DAL. The scalar returning methods in the DAL return nullable types, in case the result from the SQL query is `NULL`. Our `COUNT` query, however, will always return a non-`NULL` value; regardless, the DAL method returns a nullable integer.

In addition to the DAL method, we also need a method in the BLL. Open the `ProductsBLL` class file and add a `TotalNumberOfProducts` method that simply calls down to the DAL s `TotalNumberOfProducts` method:

```
public int TotalNumberOfProducts()
{
    return Adapter.TotalNumberOfProducts().GetValueOrDefault();
}
```

The DAL s `TotalNumberOfProducts` method returns a nullable integer; however, we ve created the `ProductsBLL` class s `TotalNumberOfProducts` method so that it returns a standard integer. Therefore, we need to have the `ProductsBLL` class s `TotalNumberOfProducts` method return the value portion of the nullable integer returned by the DAL s `TotalNumberOfProducts` method. The call to `GetValueOrDefault()` returns the value of the nullable integer, if it exists; if the nullable integer is `null`, however, it returns the default integer value, 0.

Step 3: Returning the Precise Subset of Records

Our next task is to create methods in the DAL and BLL that accept the Start Row Index and Maximum Rows variables discussed earlier and return the appropriate records. Before we do that, let s first look at the needed SQL script. The challenge facing us is that we must be able to efficiently assign an index to each row in the entire results being paged through so that we can return just those records starting at the Start Row Index (and up to the Maximum Records number of records).

This is not a challenge if there is already a column in the database table that serves as a row index. At first glance we might think that the `Products` table s `ProductID` field would suffice, as the first product has `ProductID` of 1, the second a 2, and so on. However, deleting a product leaves a gap in the sequence, nullifying this approach.

There are two general techniques used to efficiently associate a row index with the data to page through, thereby enabling the precise subset of records to be retrieved:

- **Using SQL Server 2005 s `ROW_NUMBER()` Keyword** new to SQL Server 2005, the `ROW_NUMBER()` keyword associates a ranking with each returned record based on some ordering. This ranking can be used as a row index for each row.
- **Using a Table Variable and `SET ROWCOUNT`** SQL Server s `SET ROWCOUNT` statement can be used to specify how many total records a query should process before terminating; table variables are local T-SQL variables that can hold tabular data, akin to temporary tables. This approach works equally well with both Microsoft SQL Server 2005 and SQL Server 2000 (whereas the `ROW_NUMBER()` approach only works with SQL Server 2005).

  The idea here is to create a table variable that has an `IDENTITY` column and columns for the primary keys of the table whose data is being paged through. Next, the contents of the table whose data is being paged through is dumped into the table variable, thereby associating a sequential row index (via the `IDENTITY` column) for each record in the table. Once the table variable has been populated, a `SELECT` statement on the table variable, joined with the underlying table, can be executed to pull out the particular records. The `SET ROWCOUNT` statement is used to intelligently limit the number of records that need to be dumped into the table variable.

  This approach s efficiency is based on the page number being requested, as the `SET ROWCOUNT` value is assigned the value of Start Row Index plus the Maximum Rows. When paging through low-numbered pages such as the first few pages of data this approach is very efficient. However, it exhibits default paging-like performance when retrieving a page near the end.

This tutorial implements custom paging using the `ROW_NUMBER()` keyword. For more information on using the table variable and `SET ROWCOUNT` technique, see A More Efficient Method for Paging Through Large Result Sets.

The `ROW_NUMBER()` keyword associated a ranking with each record returned over a particular ordering using the following syntax:

```
SELECT columnList,
       ROW_NUMBER() OVER(orderByClause)
FROM TableName
```

`ROW_NUMBER()` returns a numerical value that specifies the rank for each record with regards to the indicated

ordering. For example, to see the rank for each product, ordered from the most expensive to the least, we could use the following query:

```
SELECT ProductName, UnitPrice,
       ROW_NUMBER() OVER(ORDER BY UnitPrice DESC) AS PriceRank
FROM Products
```

Figure 5 shows this query s results when run through the query window in Visual Studio. Note that the products are ordered by price, along with a price rank for each row.

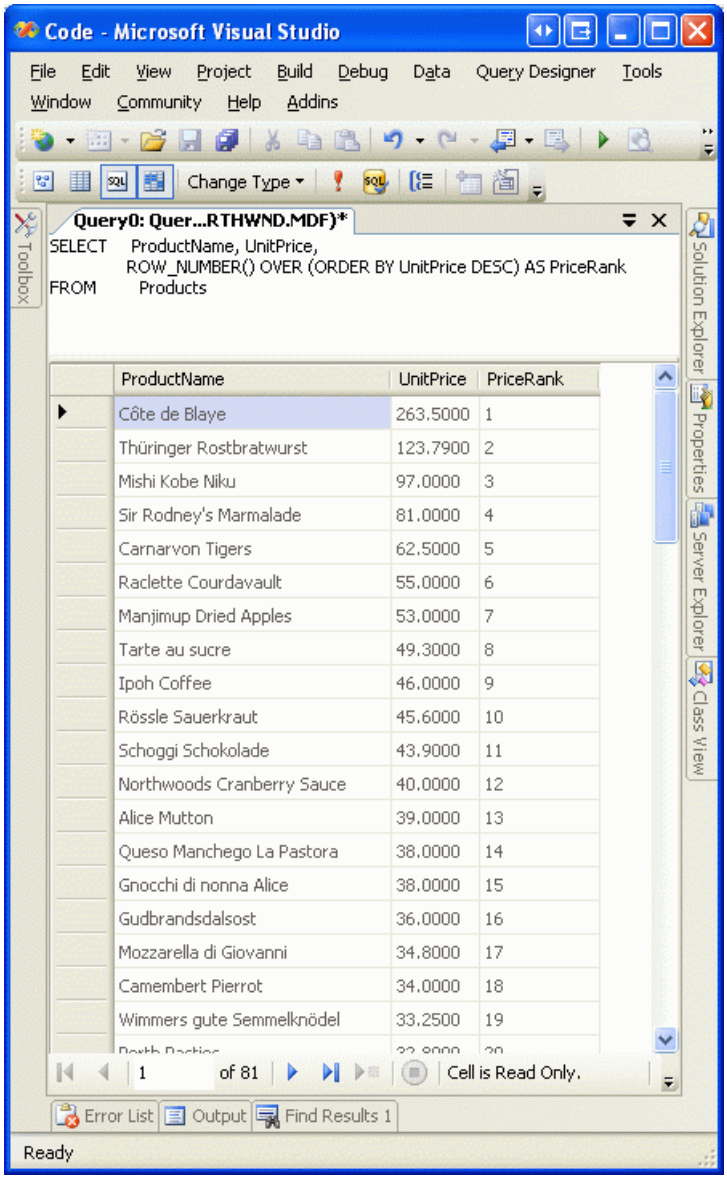


Figure 5: The Price Rank is Included for Each Returned Record

> **NOTE**
>
> `ROW_NUMBER()` is just one of the many new ranking functions available in SQL Server 2005. For a more thorough discussion of `ROW_NUMBER()`, along with the other ranking functions, read Returning Ranked Results with Microsoft SQL Server 2005.

When ranking the results by the specified `ORDER BY` column in the `OVER` clause (`UnitPrice`, in the above example), SQL Server must sort the results. This is a quick operation if there is a clustered index over the column(s) the results are being ordered by, or if there is a covering index, but can be more costly otherwise. To help improve performance for sufficiently large queries, consider adding a non-clustered index for the column by

which the results are ordered by. See Ranking Functions and Performance in SQL Server 2005 for a more detailed look at the performance considerations.

The ranking information returned by `ROW_NUMBER()` cannot directly be used in the `WHERE` clause. However, a derived table can be used to return the `ROW_NUMBER()` result, which can then appear in the `WHERE` clause. For example, the following query uses a derived table to return the ProductName and UnitPrice columns, along with the `ROW_NUMBER()` result, and then uses a `WHERE` clause to only return those products whose price rank is between 11 and 20:

```
SELECT PriceRank, ProductName, UnitPrice
FROM
   (SELECT ProductName, UnitPrice,
       ROW_NUMBER() OVER(ORDER BY UnitPrice DESC) AS PriceRank
    FROM Products
   ) AS ProductsWithRowNumber
WHERE PriceRank BETWEEN 11 AND 20
```

Extending this concept a bit further, we can utilize this approach to retrieve a specific page of data given the desired Start Row Index and Maximum Rows values:

```
SELECT PriceRank, ProductName, UnitPrice
FROM
   (SELECT ProductName, UnitPrice,
       ROW_NUMBER() OVER(ORDER BY UnitPrice DESC) AS PriceRank
    FROM Products
   ) AS ProductsWithRowNumber
WHERE PriceRank > <i>StartRowIndex</i> AND
    PriceRank <= (<i>StartRowIndex</i> + <i>MaximumRows</i>)
```

NOTE

As we will see later on in this tutorial, the `StartRowIndex` supplied by the ObjectDataSource is indexed starting at zero, whereas the `ROW_NUMBER()` value returned by SQL Server 2005 is indexed starting at 1. Therefore, the `WHERE` clause returns those records where `PriceRank` is strictly greater than `StartRowIndex` and less than or equal to `StartRowIndex` + `MaximumRows`.

Now that we ve discussed how `ROW_NUMBER()` can be used to retrieve a particular page of data given the Start Row Index and Maximum Rows values, we now need to implement this logic as methods in the DAL and BLL.

When creating this query we must decide the ordering by which the results will be ranked; let s sort the products by their name in alphabetical order. This means that with the custom paging implementation in this tutorial we will not be able to create a custom paged report than can also be sorted. In the next tutorial, though, we'll see how such functionality can be provided.

In the previous section we created the DAL method as an ad-hoc SQL statement. Unfortunately, the T-SQL parser in Visual Studio used by the TableAdapter wizard doesn t like the `OVER` syntax used by the `ROW_NUMBER()` function. Therefore, we must create this DAL method as a stored procedure. Select the Server Explorer from the View menu (or hit Ctrl+Alt+S) and expand the `NORTHWND.MDF` node. To add a new stored procedure, right-click on the Stored Procedures node and choose Add a New Stored Procedure (see Figure 6).

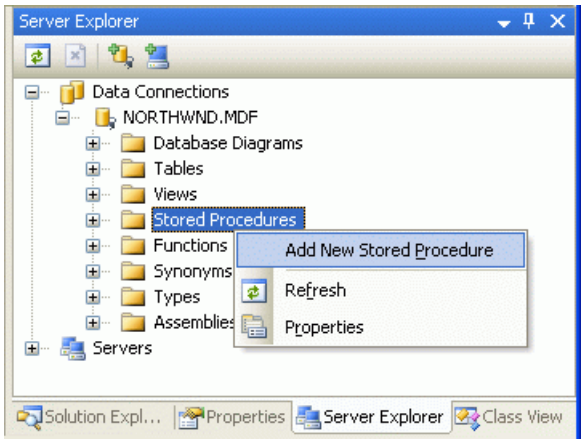


Figure 6: Add a New Stored Procedure for Paging Through the Products

This stored procedure should accept two integer input parameters - `@startRowIndex` and `@maximumRows` and use the `ROW_NUMBER()` function ordered by the `ProductName` field, returning only those rows greater than the specified `@startRowIndex` and less than or equal to `@startRowIndex` + `@maximumRow` s. Enter the following script into the new stored procedure and then click the Save icon to add the stored procedure to the database.

```
CREATE PROCEDURE dbo.GetProductsPaged
(
    @startRowIndex int,
    @maximumRows int
)
AS
    SELECT     ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit,
               UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued,
               CategoryName, SupplierName
FROM
   (
       SELECT ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit,
              UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued,
              (SELECT CategoryName
               FROM Categories
               WHERE Categories.CategoryID = Products.CategoryID) AS CategoryName,
              (SELECT CompanyName
               FROM Suppliers
               WHERE Suppliers.SupplierID = Products.SupplierID) AS SupplierName,
              ROW_NUMBER() OVER (ORDER BY ProductName) AS RowRank
        FROM Products
    ) AS ProductsWithRowNumbers
WHERE RowRank > @startRowIndex AND RowRank <= (@startRowIndex + @maximumRows)
```

After creating the stored procedure, take a moment to test it out. Right-click on the `GetProductsPaged` stored procedure name in the Server Explorer and choose the Execute option. Visual Studio will then prompt you for the input parameters, `@startRowIndex` and `@maximumRow` s (see Figure 7). Try different values and examine the results.

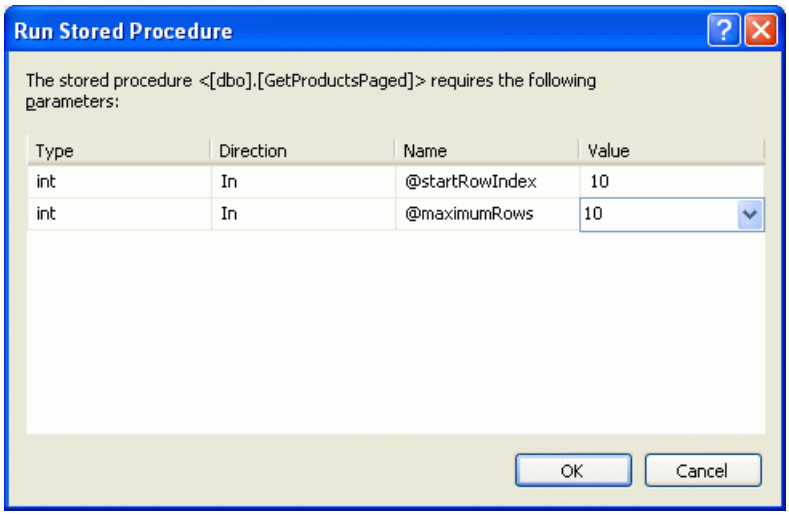


Figure 7: Enter a Value for the @startRowIndex and @maximumRows Parameters

After choosing these input parameters values, the Output window will show the results. Figure 8 shows the results when passing in 10 for both the `@startRowIndex` and `@maximumRows` parameters.

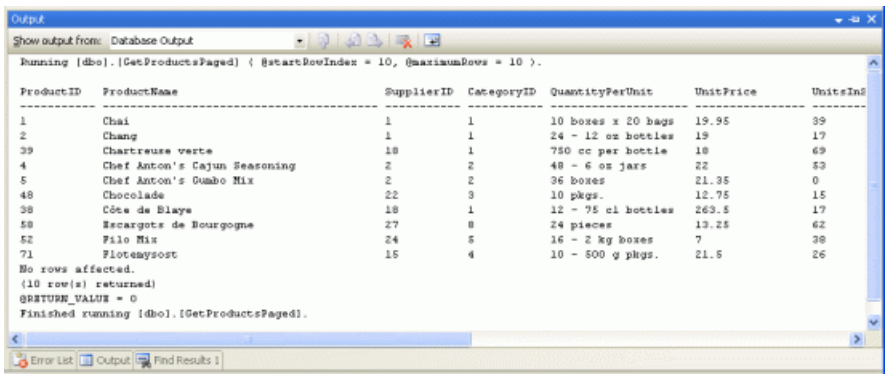


Figure 8: The Records That Would Appear in the Second Page of Data are Returned (Click to view full-size image)

With this stored procedure created, we re ready to create the `ProductsTableAdapter` method. Open the `Northwind.xsd` Typed DataSet, right-click in the `ProductsTableAdapter`, and choose the Add Query option. Instead of creating the query using an ad-hoc SQL statement, create it using an existing stored procedure.

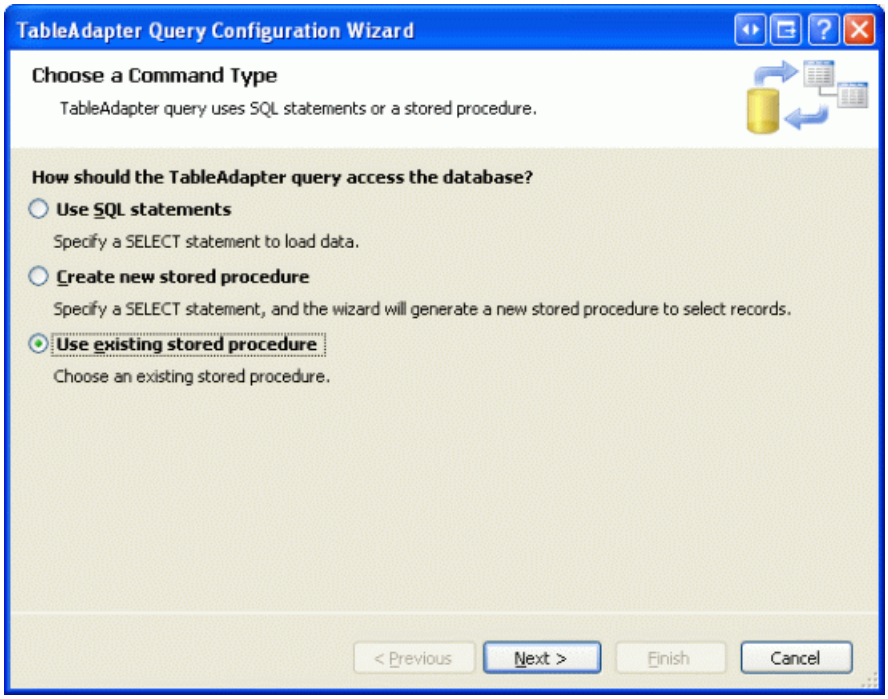


Figure 9: Create the DAL Method Using an Existing Stored Procedure

Next, we are prompted to select the stored procedure to invoke. Pick the `GetProductsPaged` stored procedure from the drop-down list.

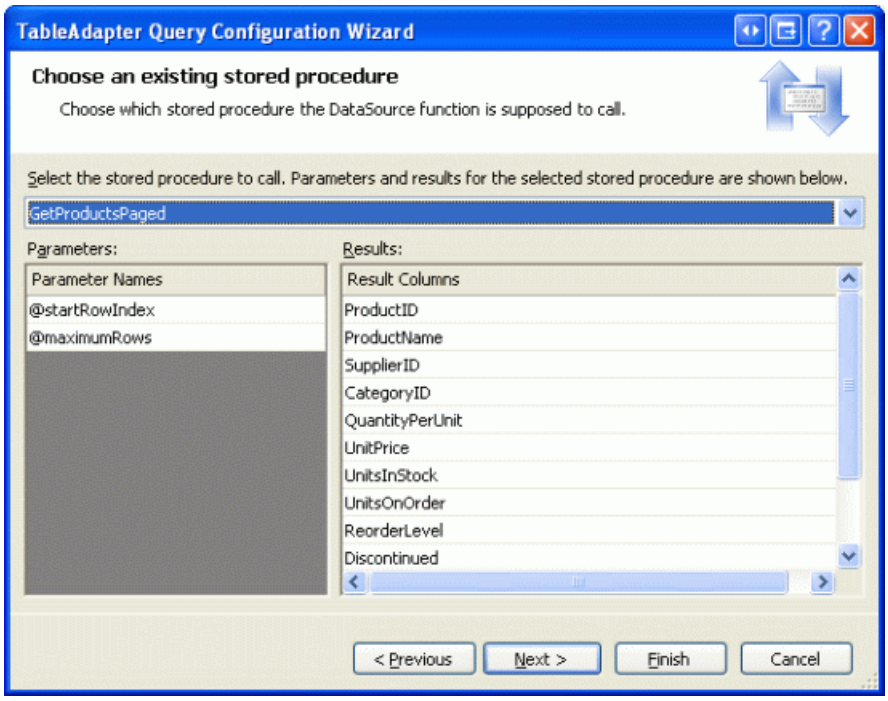


Figure 10: Choose the GetProductsPaged Stored Procedure from the Drop-Down List

The next screen then asks you what kind of data is returned by the stored procedure: tabular data, a single value, or no value. Since the `GetProductsPaged` stored procedure can return multiple records, indicate that it returns tabular data.

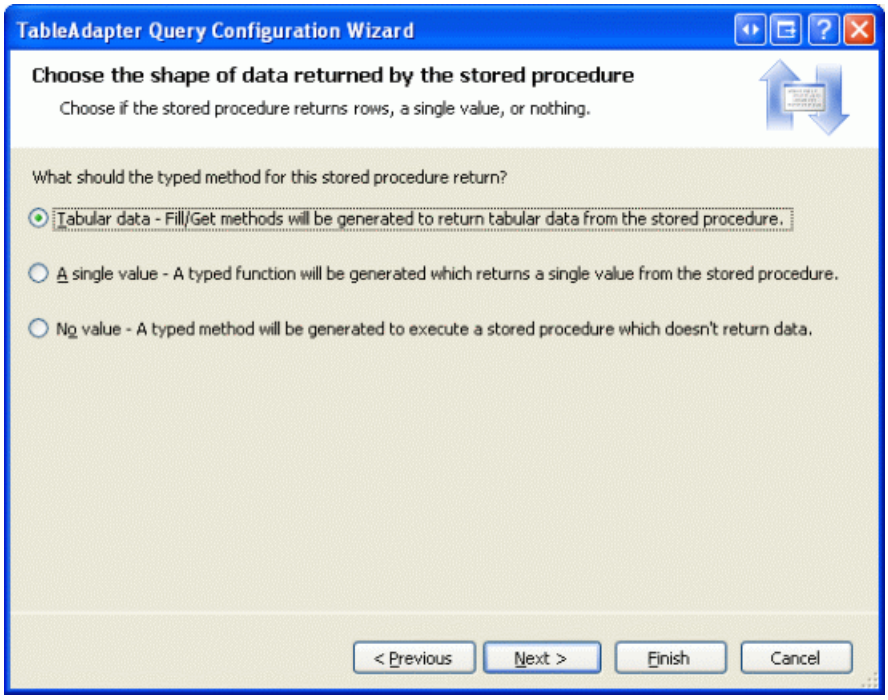


Figure 11: Indicate that the Stored Procedure Returns Tabular Data

Finally, indicate the names of the methods you want to have created. As with our previous tutorials, go ahead and create methods using both the Fill a DataTable and Return a DataTable. Name the first method `FillPaged` and the second `GetProductsPaged`.

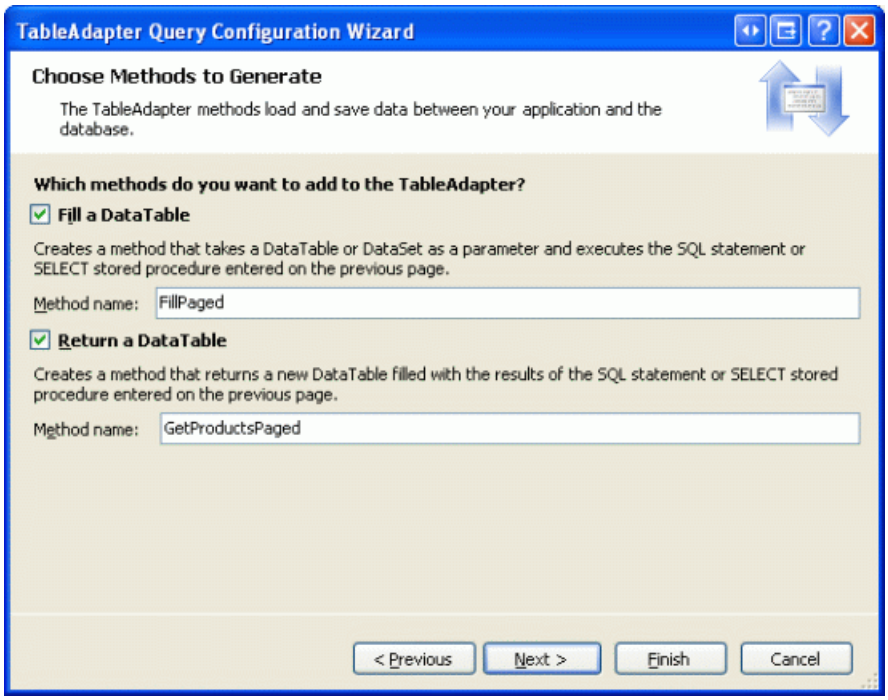


Figure 12: Name the Methods FillPaged and GetProductsPaged

In addition to created a DAL method to return a particular page of products, we also need to provide such functionality in the BLL. Like the DAL method, the BLL s GetProductsPaged method must accept two integer inputs for specifying the Start Row Index and Maximum Rows, and must return just those records that fall within the specified range. Create such a BLL method in the ProductsBLL class that merely calls down into the DAL s GetProductsPaged method, like so:

```
[System.ComponentModel.DataObjectMethodAttribute(
    System.ComponentModel.DataObjectMethodType.Select, false)]
public Northwind.ProductsDataTable GetProductsPaged(int startRowIndex, int maximumRows)
{
    return Adapter.GetProductsPaged(startRowIndex, maximumRows);
}
```

You can use any name for the BLL method s input parameters, but, as we will see shortly, choosing to use `startRowIndex` and `maximumRows` saves us from an extra bit of work when configuring an ObjectDataSource to use this method.

Step 4: Configuring the ObjectDataSource to Use Custom Paging

With the BLL and DAL methods for accessing a particular subset of records complete, we re ready to create a GridView control that pages through its underlying records using custom paging. Start by opening the `EfficientPaging.aspx` page in the `PagingAndSorting` folder, add a GridView to the page, and configure it to use a new ObjectDataSource control. In our past tutorials, we often had the ObjectDataSource configured to use the `ProductsBLL` class s `GetProducts` method. This time, however, we want to use the `GetProductsPaged` method instead, since the `GetProducts` method returns *all* of the products in the database whereas `GetProductsPaged` returns just a particular subset of records.

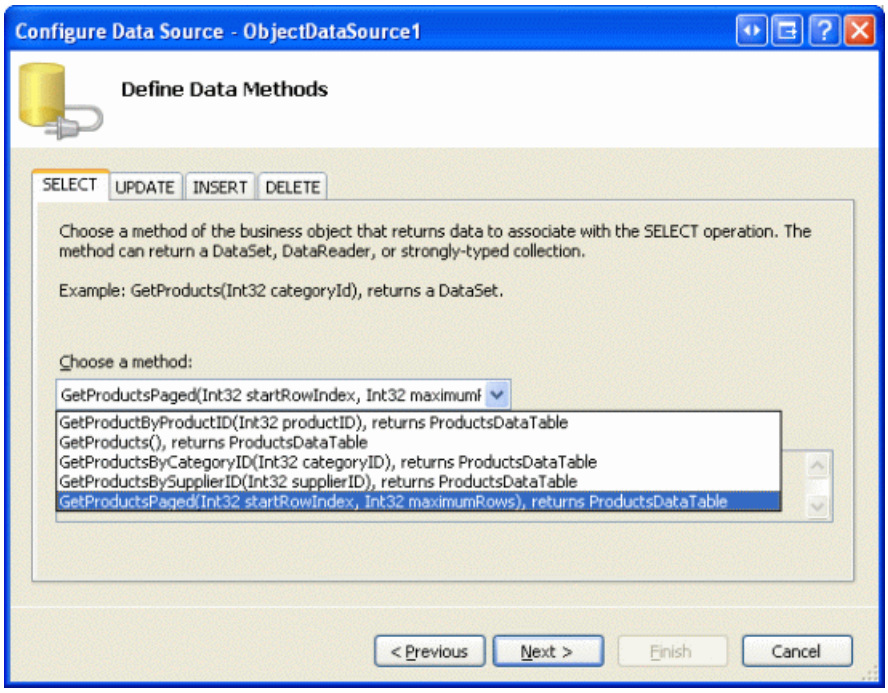


Figure 13: Configure the ObjectDataSource to Use the ProductsBLL Class s GetProductsPaged Method

Since we re creating a read-only GridView, take a moment to set the method drop-down list in the INSERT, UPDATE, and DELETE tabs to (None).

Next, the ObjectDataSource wizard prompts us for the sources of the `GetProductsPaged` method s `startRowIndex` and `maximumRows` input parameters values. These input parameters will actually be set by the GridView automatically, so simply leave the source set to None and click Finish.

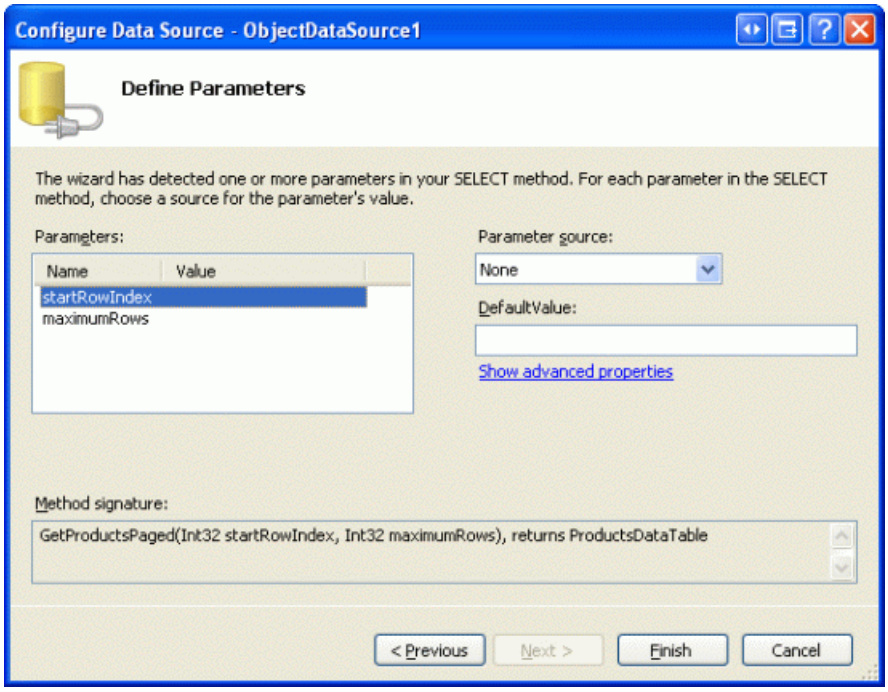


Figure 14: Leave the Input Parameter Sources as None

After completing the ObjectDataSource wizard, the GridView will contain a BoundField or CheckBoxField for each of the product data fields. Feel free to tailor the GridView s appearance as you see fit. I ve opted to display only the `ProductName`, `CategoryName`, `SupplierName`, `QuantityPerUnit`, and `UnitPrice` BoundFields. Also, configure the GridView to support paging by checking the Enable Paging checkbox in its smart tag. After these changes, the GridView and ObjectDataSource declarative markup should look similar to the following:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1" AllowPaging="True">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
            SortExpression="SupplierName" />
        <asp:BoundField DataField="QuantityPerUnit" HeaderText="Qty/Unit"
            SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
            HeaderText="Price" HtmlEncode="False" SortExpression="UnitPrice" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}" SelectMethod="GetProductsPaged"
    TypeName="ProductsBLL">
    <SelectParameters>
        <asp:Parameter Name="startRowIndex" Type="Int32" />
        <asp:Parameter Name="maximumRows" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

If you visit the page through a browser, however, the GridView is no where to be found.

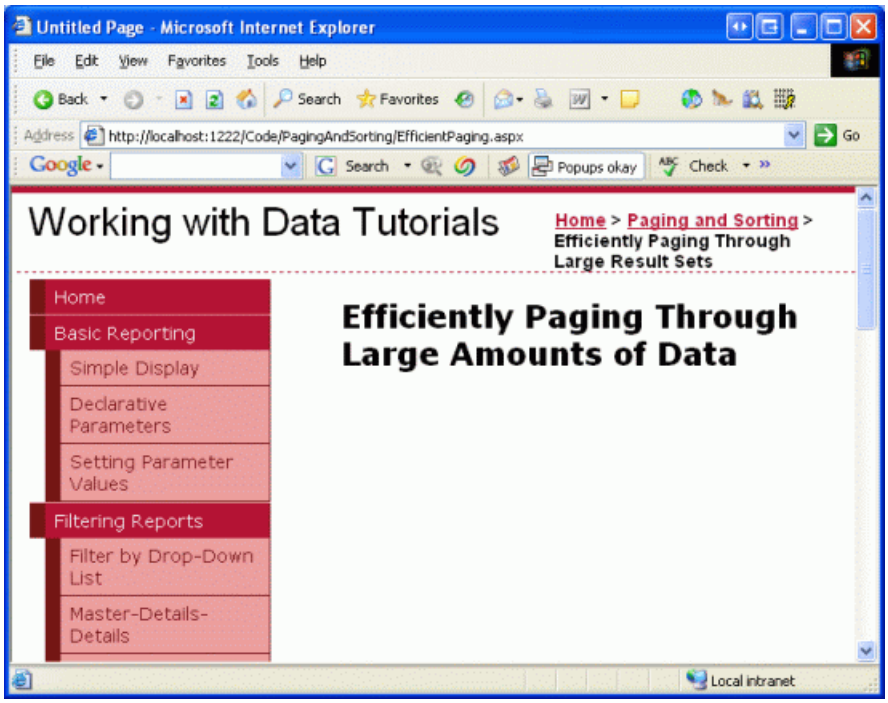


Figure 15: The GridView is Not Displayed

The GridView is missing because the ObjectDataSource is currently using 0 as the values for both of the `GetProductsPaged` `startRowIndex` and `maximumRows` input parameters. Hence, the resulting SQL query is returning no records and therefore the GridView is not displayed.

To remedy this, we need to configure the ObjectDataSource to use custom paging. This can be accomplished in the following steps:

1. **Set the ObjectDataSource s** `EnablePaging` **property to** `true` this indicates to the ObjectDataSource that it must pass to the `SelectMethod` two additional parameters: one to specify the Start Row Index (`StartRowIndexParameterName`), and one to specify the Maximum Rows (`MaximumRowsParameterName`).
2. **Set the ObjectDataSource s** `StartRowIndexParameterName` **and** `MaximumRowsParameterName` **Properties Accordingly** the `StartRowIndexParameterName` and `MaximumRowsParameterName` properties indicate the names of

the input parameters passed into the `SelectMethod` for custom paging purposes. By default, these parameter names are `startIndexRow` and `maximumRows`, which is why, when creating the `GetProductsPaged` method in the BLL, I used these values for the input parameters. If you chose to use different parameter names for the BLL s `GetProductsPaged` method such as `startIndex` and `maxRows`, for example you would need to set the ObjectDataSource s `StartRowIndexParameterName` and `MaximumRowsParameterName` properties accordingly (such as startIndex for `StartRowIndexParameterName` and maxRows for `MaximumRowsParameterName`).

3. **Set the ObjectDataSource s `SelectCountMethod` Property to the Name of the Method that Returns the Total Number of Records Being Paged Through (`TotalNumberOfProducts`)** recall that the `ProductsBLL` class s `TotalNumberOfProducts` method returns the total number of records being paged through using a DAL method that executes a `SELECT COUNT(*) FROM Products` query. This information is needed by the ObjectDataSource in order to correctly render the paging interface.
4. **Remove the `startRowIndex` and `maximumRows` `<asp:Parameter>` Elements from the ObjectDataSource s Declarative Markup** when configuring the ObjectDataSource through the wizard, Visual Studio automatically added two `<asp:Parameter>` elements for the `GetProductsPaged` method s input parameters. By setting `EnablePaging` to `true`, these parameters will be passed automatically; if they also appear in the declarative syntax, the ObjectDataSource will attempt to pass *four* parameters to the `GetProductsPaged` method and two parameters to the `TotalNumberOfProducts` method. If you forget to remove these `<asp:Parameter>` elements, when visiting the page through a browser you'll get an error message like: *ObjectDataSource 'ObjectDataSource1' could not find a non-generic method 'TotalNumberOfProducts' that has parameters: startRowIndex, maximumRows*.

After making these changes, the ObjectDataSource s declarative syntax should look like the following:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}" TypeName="ProductsBLL"
    SelectMethod="GetProductsPaged" EnablePaging="True"
    SelectCountMethod="TotalNumberOfProducts">
</asp:ObjectDataSource>
```

Note that the `EnablePaging` and `SelectCountMethod` properties have been set and the `<asp:Parameter>` elements have been removed. Figure 16 shows a screen shot of the Properties window after these changes have been made.

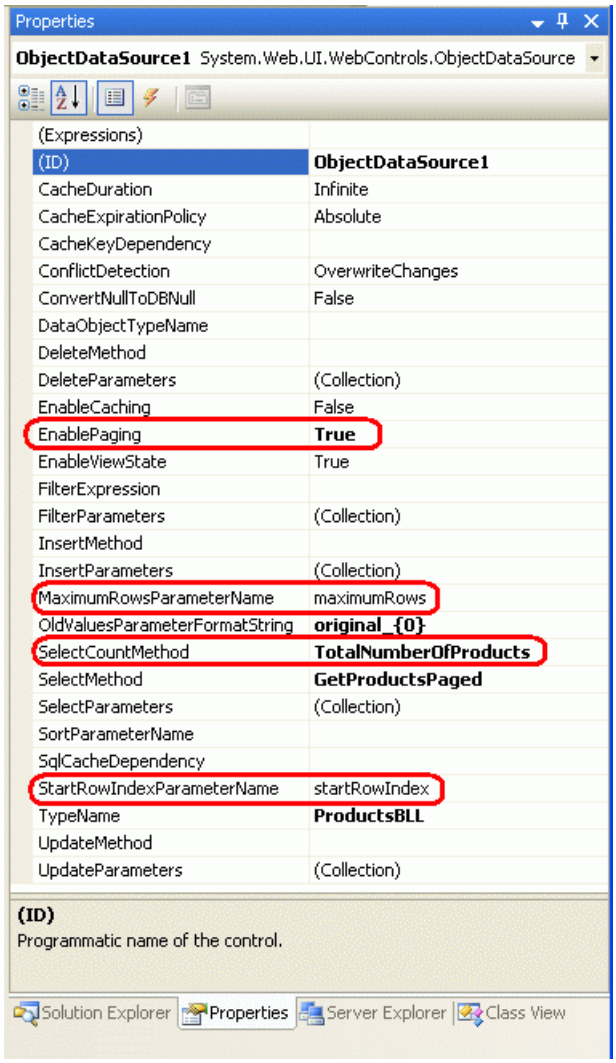


Figure 16: To Use Custom Paging, Configure the ObjectDataSource Control

After making these changes, visit this page through a browser. You should see 10 products listed, ordered alphabetically. Take a moment to step through the data one page at a time. While there is no visual difference from the end user s perspective between default paging and custom paging, custom paging more efficiently pages through large amounts of data as it only retrieves those records that need to be displayed for a given page.

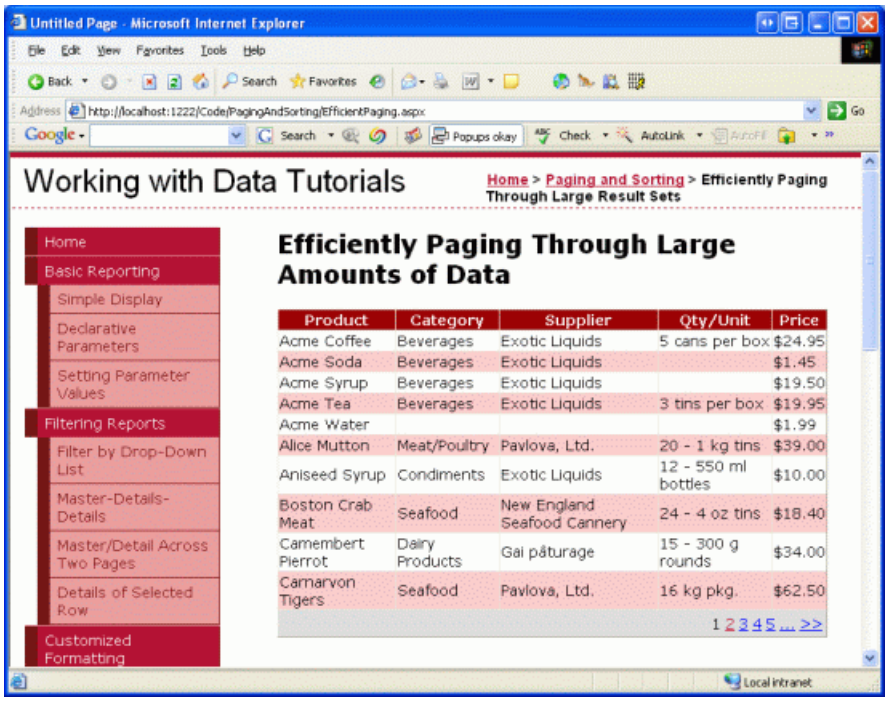


Figure 17: The Data, Ordered by the Product s Name, is Paged Using Custom Paging (Click to view full-size

image)

> **NOTE**
>
> With custom paging, the page count value returned by the ObjectDataSource's `SelectCountMethod` is stored in the GridView's view state. Other GridView variables the `PageIndex`, `EditIndex`, `SelectedIndex`, `DataKeys` collection, and so on are stored in *control state*, which is persisted regardless of the value of the GridView's `EnableViewState` property. Since the `PageCount` value is persisted across postbacks using view state, when using a paging interface that includes a link to take you to the last page, it is imperative that the GridView's view state be enabled. (If your paging interface does not include a direct link to the last page, then you may disable view state.)

Clicking the last page link causes a postback and instructs the GridView to update its `PageIndex` property. If the last page link is clicked, the GridView assigns its `PageIndex` property to a value one less than its `PageCount` property. With view state disabled, the `PageCount` value is lost across postbacks and the `PageIndex` is assigned the maximum integer value instead. Next, the GridView attempts to determine the starting row index by multiplying the `PageSize` and `PageCount` properties. This results in an `OverflowException` since the product exceeds the maximum allowed integer size.

Implement Custom Paging and Sorting

Our current custom paging implementation requires that the order by which the data is paged through be specified statically when creating the `GetProductsPaged` stored procedure. However, you may have noted that the GridView s smart tag contains an Enable Sorting checkbox in addition to the Enable Paging option. Unfortunately, adding sorting support to the GridView with our current custom paging implementation will only sort the records on the currently viewed page of data. For example, if you configure the GridView to also support paging and then, when viewing the first page of data, sort by product name in descending order, it will reverse the order of the products on page 1. As Figure 18 shows, such shows Carnarvon Tigers as the first product when sorting in reverse alphabetical order, which ignores the 71 other products that come after Carnarvon Tigers, alphabetically; only those records on the first page are considered in the sorting.

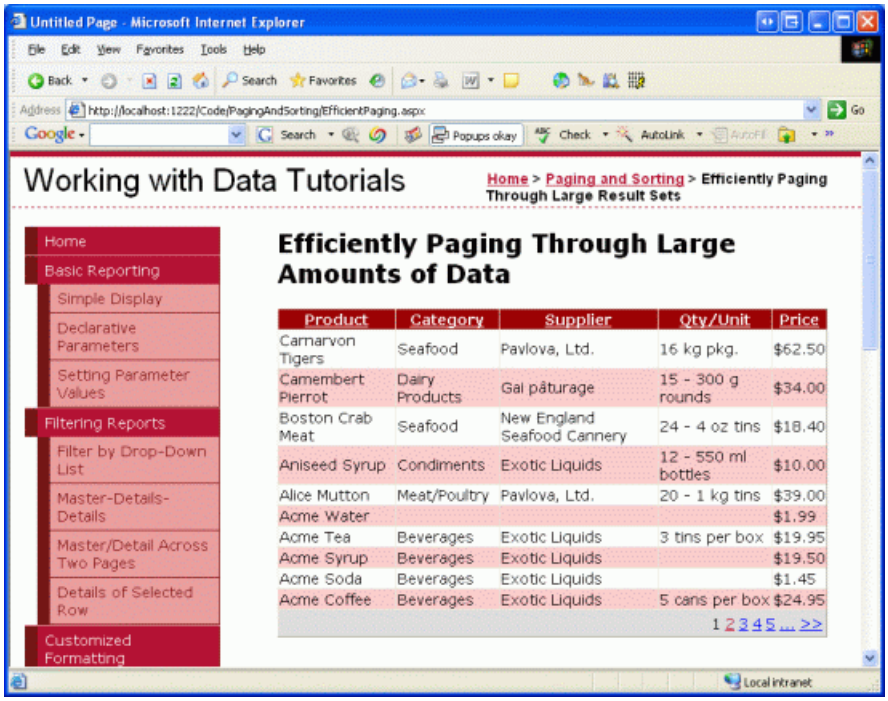


Figure 18: Only the Data Shown on the Current Page is Sorted (Click to view full-size image)

The sorting only applies to the current page of data because the sorting is occurring after the data has been retrieved from the BLL s `GetProductsPaged` method, and this method only returns those records for the specific page. To implement sorting correctly, we need to pass the sort expression to the `GetProductsPaged` method so

that the data can be ranked appropriately before returning the specific page of data. We'll see how to accomplish this in our next tutorial.

Implementing Custom Paging and Deleting

If you enabling deleting functionality in a GridView whose data is paged using custom paging techniques you will find that when deleting the last record from the last page, the GridView disappears rather than appropriately decrementing the GridView s `PageIndex`. To reproduce this bug, enable deleting for the tutorial just we just created. Go to the last page (page 9), where you should see a single product since we are paging through 81 products, 10 products at a time. Delete this product.

Upon deleting the last product, the GridView *should* automatically go to the eighth page, and such functionality is exhibited with default paging. With custom paging, however, after deleting that last product on the last page, the GridView simply disappears from the screen altogether. The precise reason *why* this happens is a bit beyond the scope of this tutorial; see Deleting the Last Record on the Last Page from a GridView with Custom Paging for the low-level details as to the source of this problem. In summary it s due to the following sequence of steps that are performed by the GridView when the Delete button is clicked:

1. Delete the record
2. Get the appropriate records to display for the specified `PageIndex` and `PageSize`
3. Check to ensure that the `PageIndex` does not exceed the number of pages of data in the data source; if it does, automatically decrement the GridView s `PageIndex` property
4. Bind the appropriate page of data to the GridView using the records obtained in Step 2

The problem stems from the fact that in Step 2 the `PageIndex` used when grabbing the records to display is still the `PageIndex` of the last page whose sole record was just deleted. Therefore, in Step 2, *no* records are returned since that last page of data no longer contains any records. Then, in Step 3, the GridView realizes that its `PageIndex` property is greater than the total number of pages in the data source (since we ve deleted the last record in the last page) and therefore decrements its `PageIndex` property. In Step 4 the GridView attempts to bind itself to the data retrieved in Step 2; however, in Step 2 no records were returned, therefore resulting in an empty GridView. With default paging, this problem doesn t surface because in Step 2 *all* records are retrieved from the data source.

To fix this we have two options. The first is to create an event handler for the GridView s `RowDeleted` event handler that determines how many records were displayed in the page that was just deleted. If there was only one record, then the record just deleted must have been the last one and we need to decrement the GridView s `PageIndex`. Of course, we only want to update the `PageIndex` if the delete operation was actually successful, which can be determined by ensuring that the `e.Exception` property is `null`.

This approach works because it updates the `PageIndex` after Step 1 but before Step 2. Therefore, in Step 2, the appropriate set of records is returned. To accomplish this, use code like the following:

```
protected void GridView1_RowDeleted(object sender, GridViewDeletedEventArgs e)
{
    // If we just deleted the last row in the GridView, decrement the PageIndex
    if (e.Exception == null && GridView1.Rows.Count == 1)
        // we just deleted the last row
        GridView1.PageIndex = Math.Max(0, GridView1.PageIndex - 1);
}
```

An alternative workaround is to create an event handler for the ObjectDataSource s `RowDeleted` event and to set the `AffectedRows` property to a value of 1. After deleting the record in Step 1 (but before re-retrieving the data in Step 2), the GridView updates its `PageIndex` property if one or more rows were affected by the operation. However, the `AffectedRows` property is not set by the ObjectDataSource and therefore this step is omitted. One

way to have this step executed is to manually set the `AffectedRows` property if the delete operation completes successfully. This can be accomplished using code like the following:

```
protected void ObjectDataSource1_Deleted(
    object sender, ObjectDataSourceStatusEventArgs e)
{
    // If we get back a Boolean value from the DeleteProduct method and it's true,
    // then we successfully deleted the product. Set AffectedRows to 1
    if (e.ReturnValue is bool && ((bool)e.ReturnValue) == true)
        e.AffectedRows = 1;
}
```

The code for both of these events handlers can be found in code-behind class of the `EfficientPaging.aspx` example.

Comparing the Performance of Default and Custom Paging

Since custom paging only retrieves the needed records, whereas default paging returns *all* of the records for each page being viewed, it s clear that custom paging is more efficient than default paging. But just how much more efficient is custom paging? What sort of performance gains can be seen by moving from default paging to custom paging?

Unfortunately, there s no one size fits all answer here. The performance gain depends on a number of factors, the most prominent two being the number of records being paged through and the load placed on the database server and communication channels between the web server and database server. For small tables with just a few dozen records, the performance difference may be negligible. For large tables, with thousands to hundreds of thousands of rows, though, the performance difference is acute.

An article of mine, Custom Paging in ASP.NET 2.0 with SQL Server 2005, contains some performance tests I ran to exhibit the differences in performance between these two paging techniques when paging through a database table with 50,000 records. In these tests I examined both the time to execute the query at the SQL Server level (using SQL Profiler) and at the ASP.NET page using ASP.NET s tracing features. Keep in mind that these tests were run on my development box with a single active user, and therefore are unscientific and do not mimic typical website load patterns. Regardless, the results illustrate the relative differences in execution time for default and custom paging when working with sufficiently large amounts of data.

	AVG. DURATION (SEC)	READS
Default Paging SQL Profiler	1.411	383
Custom Paging SQL Profiler	0.002	29
Default Paging ASP.NET Trace	2.379	*N/A*
Custom Paging ASP.NET Trace	0.029	*N/A*

As you can see, retrieving a particular page of data required 354 less reads on average and completed in a fraction of the time. At the ASP.NET page, custom the page was able to render in close to 1/100th of the time it took when using default paging. See my article for more information on these results along with code and a database you can download to reproduce these tests in your own environment.

Summary

Default paging is a cinch to implement just check the Enable Paging checkbox in the data Web control s smart tag but such simplicity comes at the cost of performance. With default paging, when a user requests any page of data

all records are returned, even though only a tiny fraction of them may be shown. To combat this performance overhead, the ObjectDataSource offers an alternative paging option custom paging.

While custom paging improves upon default paging s performance issues by retrieving only those records that need to be displayed, it s more involved to implement custom paging. First, a query must be written that correctly (and efficiently) accesses the specific subset of records requested. This can be accomplished in a number of ways; the one we examined in this tutorial is to use SQL Server 2005 s new `ROW_NUMBER()` function to rank results, and then to return just those results whose ranking falls within a specified range. Furthermore, we need to add a means to determine the total number of records being paged through. After creating these DAL and BLL methods, we also need to configure the ObjectDataSource so that it can determine how many total records are being paged through and can correctly pass the Start Row Index and Maximum Rows values to the BLL.

While implementing custom paging does require a number of steps and is not nearly as simple as default paging, custom paging is a necessity when paging through sufficiently large amounts of data. As the results examined showed, custom paging can shed seconds off of the ASP.NET page render time and can lighten the load on the database server by one ore more orders of magnitude.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Sorting Custom Paged Data (C#)

2/20/2020 • 12 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In the previous tutorial we learned how to implement custom paging when presenting data on a web page. In this tutorial we see how to extend the preceding example to include support for sorting custom paging.

Introduction

Compared to default paging, custom paging can improve the performance of paging through data by several orders of magnitude, making custom paging the de facto paging implementation choice when paging through large amounts of data. Implementing custom paging is more involved than implementing default paging, however, especially when adding sorting to the mix. In this tutorial we'll extend the example from the preceding one to include support for sorting *and* custom paging.

> **NOTE**
>
> Since this tutorial builds upon the preceding one, before beginning take a moment to copy the declarative syntax within the `<asp:Content>` element from the preceding tutorial s web page (`EfficientPaging.aspx`) and paste it between the `<asp:Content>` element in the `SortParameter.aspx` page. Refer back to Step 1 of the Adding Validation Controls to the Editing and Inserting Interfaces tutorial for a more detailed discussion on replicating the functionality of one ASP.NET page to another.

Step 1: Reexamining the Custom Paging Technique

For custom paging to work properly, we must implement some technique that can efficiently grab a particular subset of records given the Start Row Index and Maximum Rows parameters. There are a handful of techniques that can be used to achieve this aim. In the preceding tutorial we looked at accomplishing this using Microsoft SQL Server 2005 s new `ROW_NUMBER()` ranking function. In short, the `ROW_NUMBER()` ranking function assigns a row number to each row returned by a query that is ranked by a specified sort order. The appropriate subset of records is then obtained by returning a particular section of the numbered results. The following query illustrates how to use this technique to return those products numbered 11 through 20 when ranking the results ordered alphabetically by the `ProductName`:

```
SELECT ProductID, ProductName, ...
FROM
   (SELECT ProductID, ProductName, ..., ROW_NUMBER() OVER
        (ORDER BY ProductName) AS RowRank
    FROM Products) AS ProductsWithRowNumbers
WHERE RowRank > 10 AND RowRank <= 20
```

This technique works well for paging using a specific sort order (`ProductName` sorted alphabetically, in this case), but the query needs to be modified to show the results sorted by a different sort expression. Ideally, the above query could be rewritten to use a parameter in the `OVER` clause, like so:

```
SELECT ProductID, ProductName, ...
FROM
   (SELECT ProductID, ProductName, ..., ROW_NUMBER() OVER
        (ORDER BY @sortExpression) AS RowRank
    FROM Products) AS ProductsWithRowNumbers
WHERE RowRank > 10 AND RowRank <= 20
```

Unfortunately, parameterized `ORDER BY` clauses are not allowed. Instead, we must create a stored procedure that accepts a `@sortExpression` input parameter, but uses one of the following workarounds:

- Write hard-coded queries for each of the sort expressions that may be used; then, use `IF/ELSE` T-SQL statements to determine which query to execute.
- Use a `CASE` statement to provide dynamic `ORDER BY` expressions based on the `@sortExpressio` n input parameter; see the Used to Dynamically Sort Query Results section in The Power of SQL `CASE` Statements for more information.
- Craft the appropriate query as a string in the stored procedure and then use the `sp_executesql` system stored procedure to execute the dynamic query.

Each of these workarounds has some drawbacks. The first option is not as maintainable as the other two as it requires that you create a query for each possible sort expression. Therefore, if later you decide to add new, sortable fields to the GridView you will also need to go back and update the stored procedure. The second approach has some subtleties that introduce performance concerns when sorting by non-string database columns and also suffers from the same maintainability issues as the first. And the third choice, which uses dynamic SQL, introduces the risk for a SQL injection attack if an attacker is able to execute the stored procedure passing in the input parameter values of their choosing.

While none of these approaches is perfect, I think the third option is the best of the three. With its use of dynamic SQL, it offers a level of flexibility the other two do not. Furthermore, a SQL injection attack can only be exploited if an attacker is able to execute the stored procedure passing in the input parameters of his choice. Since the DAL uses parameterized queries, ADO.NET will protect those parameters that are sent to the database through the architecture, meaning that the SQL injection attack vulnerability only exists if the attacker can directly execute the stored procedure.

To implement this functionality, create a new stored procedure in the Northwind database named `GetProductsPagedAndSorted`. This stored procedure should accept three input parameters: `@sortExpression`, an input parameter of type `nvarchar(100`) that specifies how the results should be sorted and is injected directly after the `ORDER BY` text in the `OVER` clause; and `@startRowIndex` and `@maximumRows`, the same two integer input parameters from the `GetProductsPaged` stored procedure examined in the preceding tutorial. Create the `GetProductsPagedAndSorted` stored procedure using the following script:

```
CREATE PROCEDURE dbo.GetProductsPagedAndSorted
(
    @sortExpression nvarchar(100),
    @startRowIndex int,
    @maximumRows int
)
AS
-- Make sure a @sortExpression is specified
IF LEN(@sortExpression) = 0
    SET @sortExpression = 'ProductID'
-- Issue query
DECLARE @sql nvarchar(4000)
SET @sql = 'SELECT ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit,
            UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued,
            CategoryName, SupplierName
            FROM (SELECT ProductID, ProductName, p.SupplierID, p.CategoryID,
                    QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
                    ReorderLevel, Discontinued,
                  c.CategoryName, s.CompanyName AS SupplierName,
                   ROW_NUMBER() OVER (ORDER BY ' + @sortExpression + ') AS RowRank
            FROM Products AS p
                    INNER JOIN Categories AS c ON
                        c.CategoryID = p.CategoryID
                    INNER JOIN Suppliers AS s ON
                        s.SupplierID = p.SupplierID) AS ProductsWithRowNumbers
            WHERE     RowRank > ' + CONVERT(nvarchar(10), @startRowIndex) +
                ' AND RowRank <= (' + CONVERT(nvarchar(10), @startRowIndex) + ' + '
                + CONVERT(nvarchar(10), @maximumRows) + ')'
-- Execute the SQL query
EXEC sp_executesql @sql
```

The stored procedure starts by ensuring that a value for the `@sortExpression` parameter has been specified. If it is missing, the results are ranked by `ProductID`. Next, the dynamic SQL query is constructed. Note that the dynamic SQL query here differs slightly from our previous queries used to retrieve all rows from the Products table. In prior examples, we obtained each product s associated category s and supplier s names using a subquery. This decision was made back in the Creating a Data Access Layer tutorial and was done in lieu of using `JOIN` s because the TableAdapter cannot automatically create the associated insert, update, and delete methods for such queries. The `GetProductsPagedAndSorted` stored procedure, however, must use `JOIN` s for the results to be ordered by the category or supplier names.

This dynamic query is built up by concatenating the static query portions and the `@sortExpression`, `@startRowIndex`, and `@maximumRows` parameters. Since `@startRowIndex` and `@maximumRows` are integer parameters, they must be converted into nvarchars in order to be correctly concatenated. Once this dynamic SQL query has been constructed, it is executed via `sp_executesql`.

Take a moment to test this stored procedure with different values for the `@sortExpression`, `@startRowIndex`, and `@maximumRows` parameters. From the Server Explorer, right-click on the stored procedure name and choose Execute. This will bring up the Run Stored Procedure dialog box into which you can enter the input parameters (see Figure 1). To sort the results by the category name, use CategoryName for the `@sortExpression` parameter value; to sort by the supplier s company name, use CompanyName. After providing the parameters values, click OK. The results are displayed in the Output window. Figure 2 shows the results when returning products ranked 11 through 20 when ordering by the `UnitPrice` in descending order.

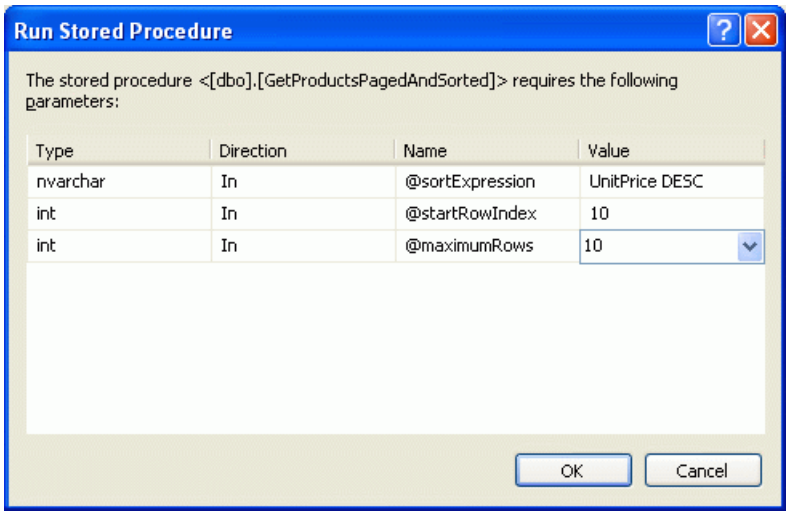


Figure 1: Try Different Values for the Stored Procedure s Three Input Parameters

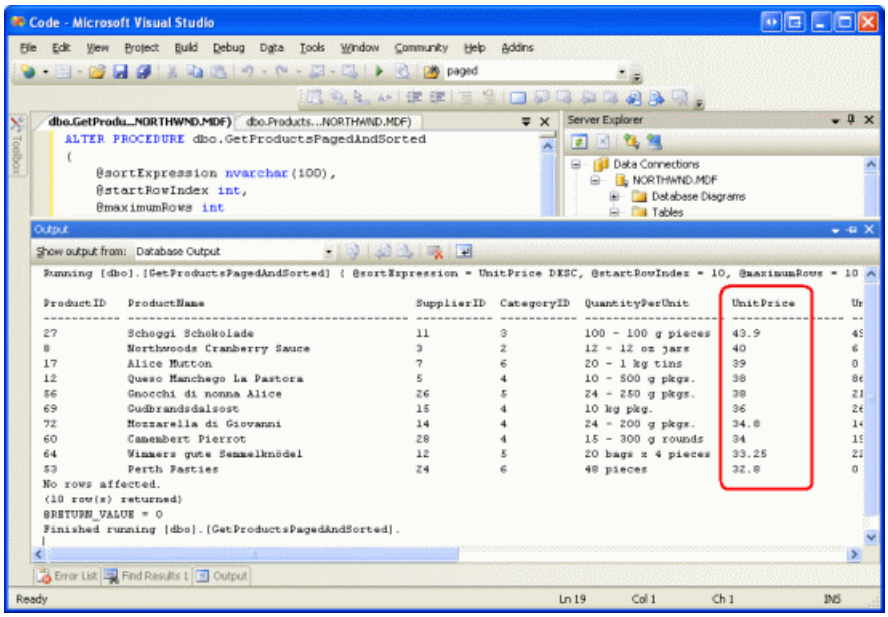


Figure 2: The Stored Procedure s Results are Shown in the Output Window (Click to view full-size image)

> **NOTE**
>
> When ranking the results by the specified `ORDER BY` column in the `OVER` clause, SQL Server must sort the results. This is a quick operation if there is a clustered index over the column(s) the results are being ordered by or if there is a covering index, but can be more costly otherwise. To improve performance for sufficiently large queries, consider adding a non-clustered index for the column by which the results are ordered by. Refer to Ranking Functions and Performance in SQL Server 2005 for more details.

Step 2: Augmenting the Data Access and Business Logic Layers

With the `GetProductsPagedAndSorted` stored procedure created, our next step is to provide a means to execute that stored procedure through our application architecture. This entails adding an appropriate method to both the DAL and BLL. Let s start by adding a method to the DAL. Open the `Northwind.xsd` Typed DataSet, right-click on the `ProductsTableAdapter`, and choose the Add Query option from the context menu. As we did in the preceding tutorial, we want to configure this new DAL method to use an existing stored procedure - `GetProductsPagedAndSorted`, in this case. Start by indicating that you want the new TableAdapter method to use an existing stored procedure.

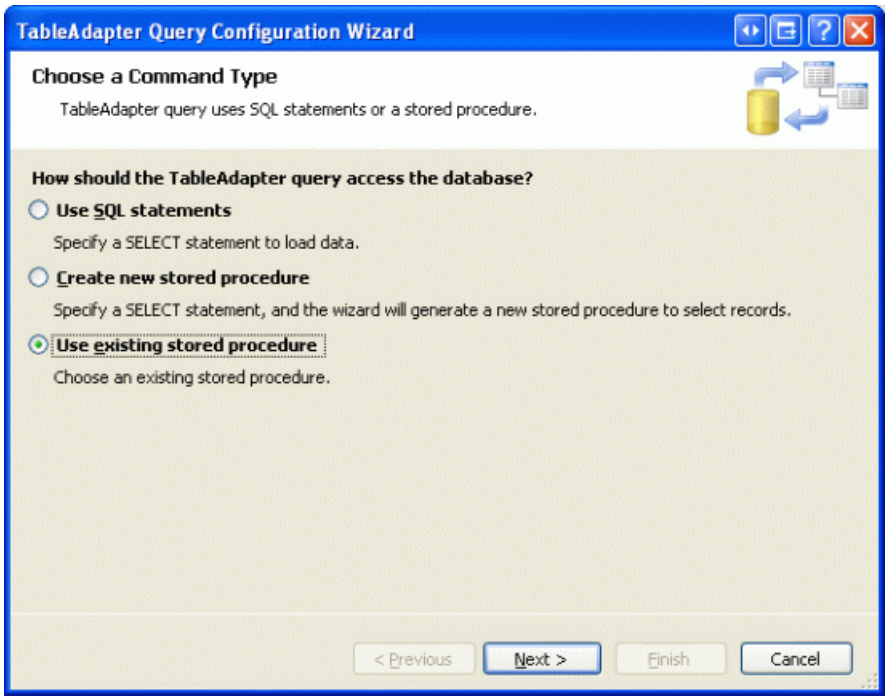


Figure 3: Choose to Use an Existing Stored Procedure

To specify the stored procedure to use, select the `GetProductsPagedAndSorted` stored procedure from the drop-down list in the next screen.

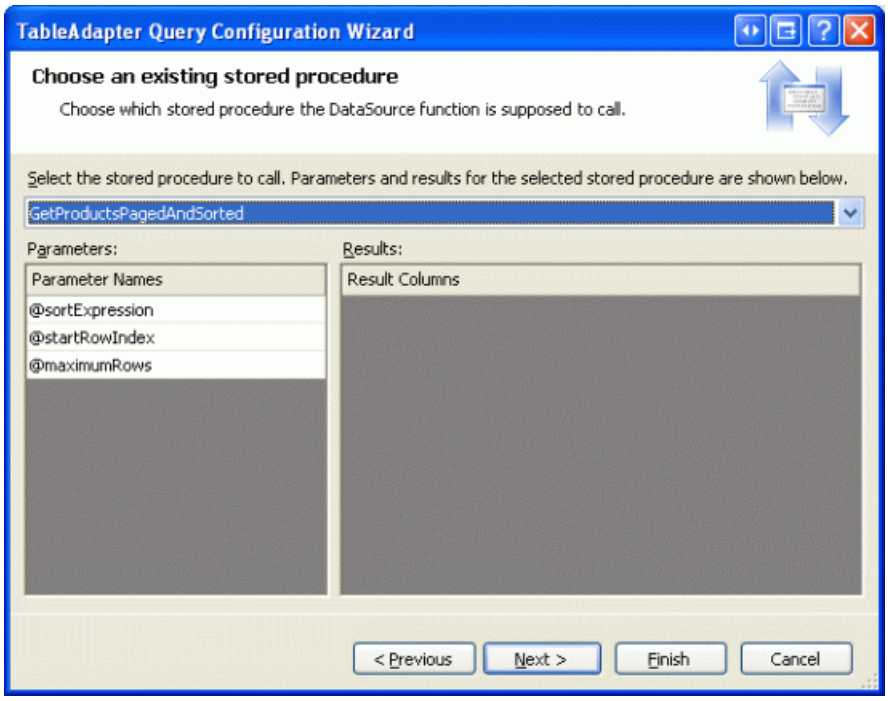


Figure 4: Use the GetProductsPagedAndSorted Stored Procedure

This stored procedure returns a set of records as its results so, in the next screen, indicate that it returns tabular data.

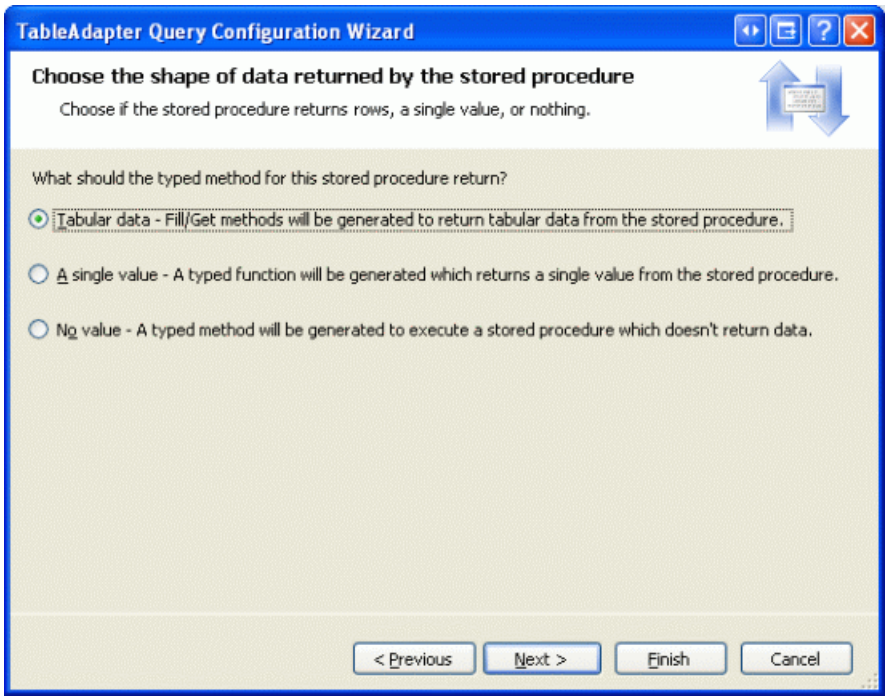


Figure 5: Indicate that the Stored Procedure Returns Tabular Data

Finally, create DAL methods that use both the Fill a DataTable and Return a DataTable patterns, naming the methods `FillPagedAndSorted` and `GetProductsPagedAndSorted`, respectively.

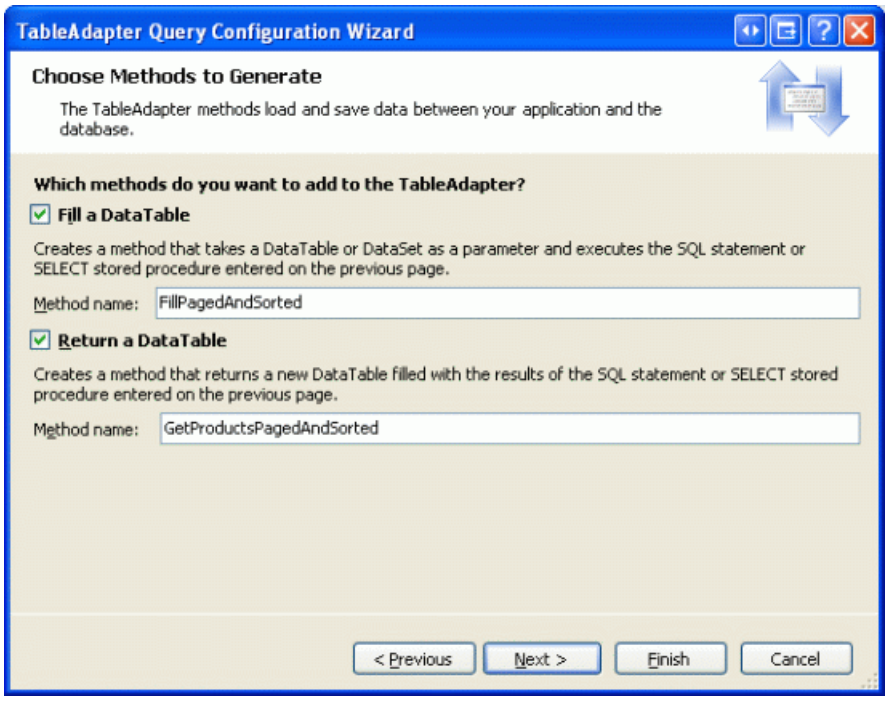


Figure 6: Choose the Methods Names

Now that we ve extended the DAL, we re ready to turn to the BLL. Open the `ProductsBLL` class file and add a new method, `GetProductsPagedAndSorted`. This method needs to accept three input parameters `sortExpression`, `startRowIndex`, and `maximumRows` and should simply call down into the DAL s `GetProductsPagedAndSorted` method, like so:

```
[System.ComponentModel.DataObjectMethodAttribute(
    System.ComponentModel.DataObjectMethodType.Select, false)]
public Northwind.ProductsDataTable GetProductsPagedAndSorted(
    string sortExpression, int startRowIndex, int maximumRows)
{
    return Adapter.GetProductsPagedAndSorted
        (sortExpression, startRowIndex, maximumRows);
}
```

Step 3: Configuring the ObjectDataSource to Pass in the SortExpression Parameter

Having augmented the DAL and BLL to include methods that utilize the `GetProductsPagedAndSorted` stored procedure, all that remains is to configure the ObjectDataSource in the `SortParameter.aspx` page to use the new BLL method and to pass in the `SortExpression` parameter based on the column that the user has requested to sort the results by.

Start by changing the ObjectDataSource s `SelectMethod` from `GetProductsPaged` to `GetProductsPagedAndSorted`. This can be done through the Configure Data Source wizard, from the Properties window, or directly through the declarative syntax. Next, we need to provide a value for the ObjectDataSource s `SortParameterName` property. If this property is set, the ObjectDataSource attempts to pass in the GridView s `SortExpression` property to the `SelectMethod`. In particular, the ObjectDataSource looks for an input parameter whose name is equal to the value of the `SortParameterName` property. Since the BLL s `GetProductsPagedAndSorted` method has the sort expression input parameter named `sortExpression`, set the ObjectDataSource s `SortExpression` property to sortExpression .

After making these two changes, the ObjectDataSource s declarative syntax should look similar to the following:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}" TypeName="ProductsBLL"
    SelectMethod="GetProductsPagedAndSorted" EnablePaging="True"
    SelectCountMethod="TotalNumberOfProducts" SortParameterName="sortExpression">
</asp:ObjectDataSource>
```

> **NOTE**
>
> As with the preceding tutorial, ensure that the ObjectDataSource does *not* include the sortExpression, startRowIndex, or maximumRows input parameters in its SelectParameters collection.

To enable sorting in the GridView, simply check the Enable Sorting checkbox in the GridView s smart tag, which sets the GridView s `AllowSorting` property to `true` and causing the header text for each column to be rendered as a LinkButton. When the end user clicks on one of the header LinkButtons, a postback ensues and the following steps transpire:

1. The GridView updates its `SortExpression` property to the value of the `SortExpression` of the field whose header link was clicked
2. The ObjectDataSource invokes the BLL s `GetProductsPagedAndSorted` method, passing in the GridView s `SortExpression` property as the value for the method s `sortExpression` input parameter (along with the appropriate `startRowIndex` and `maximumRows` input parameter values)
3. The BLL invokes the DAL s `GetProductsPagedAndSorted` method
4. The DAL executes the `GetProductsPagedAndSorted` stored procedure, passing in the `@sortExpression` parameter (along with the `@startRowIndex` and `@maximumRows` input parameter values)
5. The stored procedure returns the appropriate subset of data to the BLL, which returns it to the

ObjectDataSource; this data is then bound to the GridView, rendered into HTML, and sent down to the end user

Figure 7 shows the first page of results when sorted by the `UnitPrice` in ascending order.

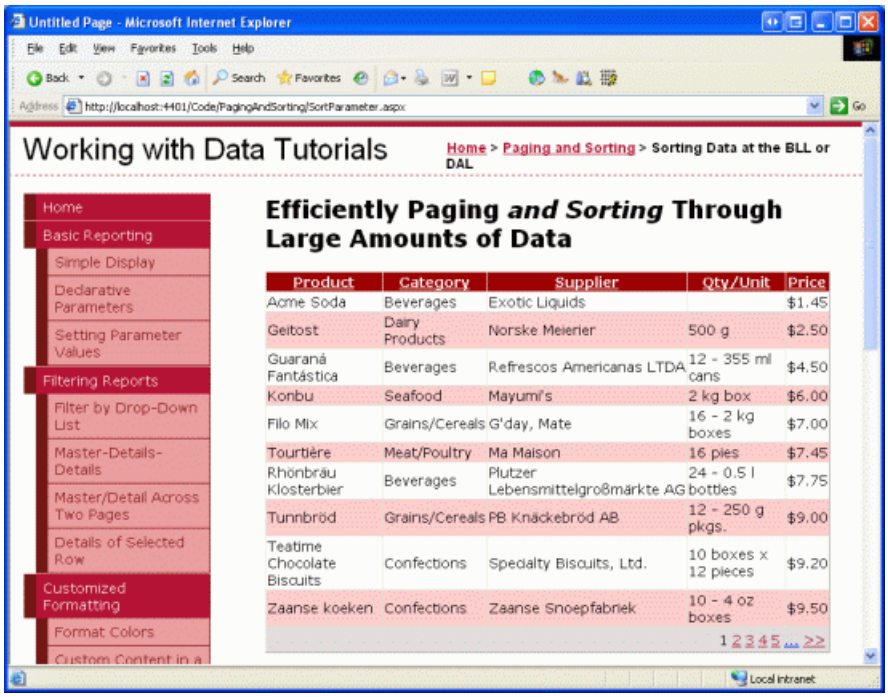


Figure 7: The Results are Sorted by the UnitPrice (Click to view full-size image)

While the current implementation can correctly sort the results by product name, category name, quantity per unit, and unit price, attempting to order the results by the supplier name results in a runtime exception (see Figure 8).

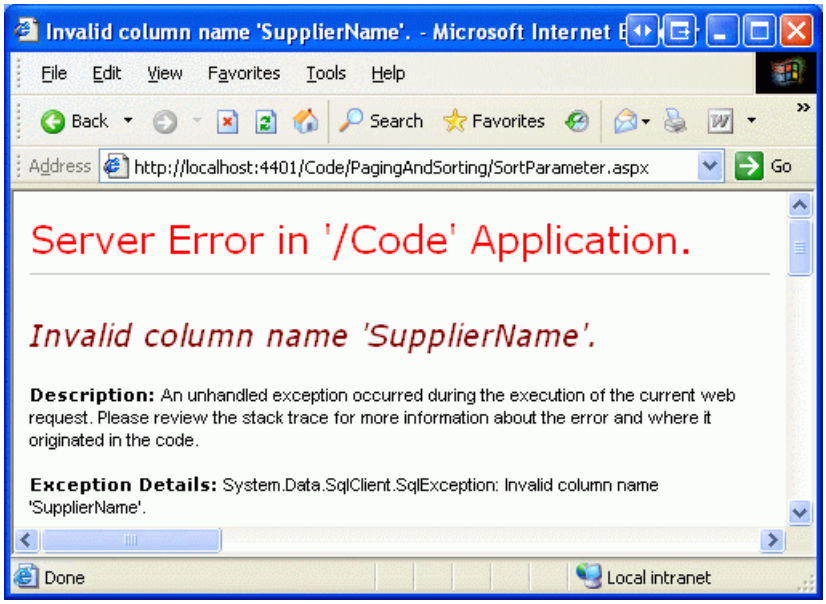


Figure 8: Attempting to Sort the Results by the Supplier Results in the Following Runtime Exception

This exception occurs because the `SortExpression` of the GridView s `SupplierName` BoundField is set to `SupplierName`. However, the supplier s name in the `Suppliers` table is actually called `CompanyName` we have been aliased this column name as `SupplierName`. However, the `OVER` clause used by the `ROW_NUMBER()` function cannot use the alias and must use the actual column name. Therefore, change the `SupplierName` BoundField s `SortExpression` from SupplierName to CompanyName (see Figure 9). As Figure 10 shows, after this change the results can be sorted by the supplier.

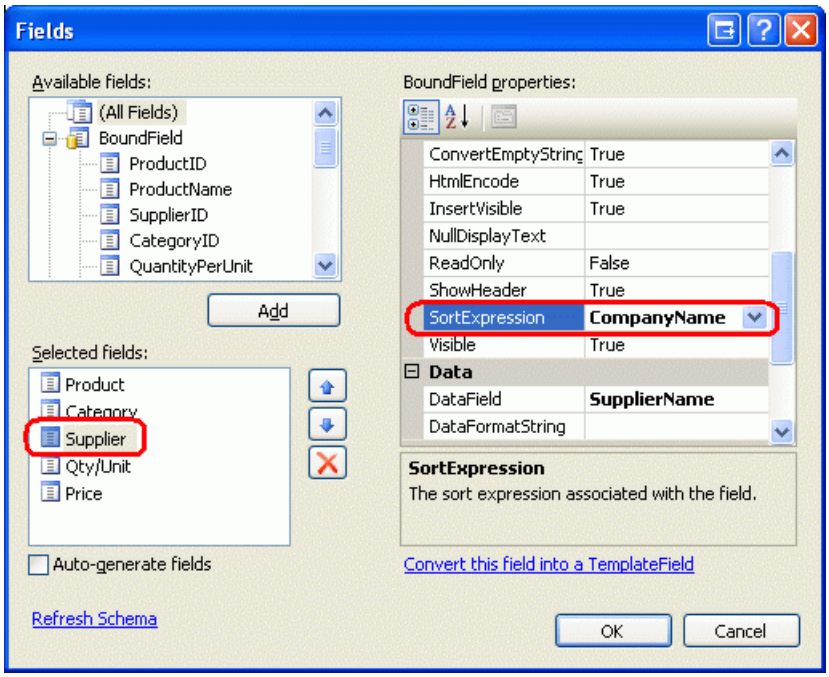


Figure 9: Change the SupplierName BoundField s SortExpression to CompanyName

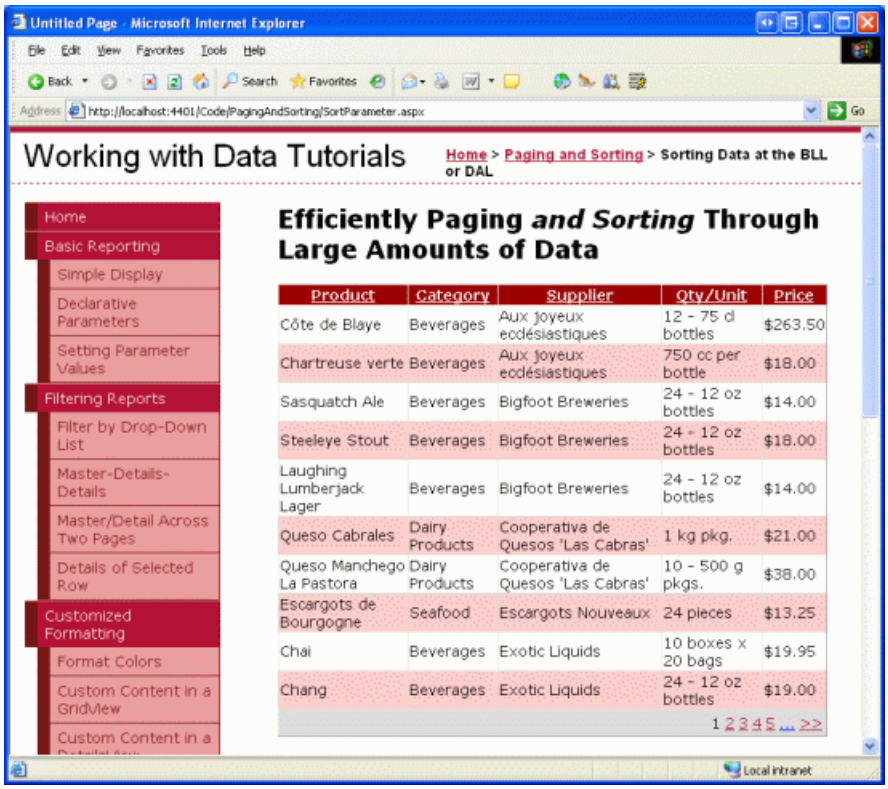


Figure 10: The Results Can Now Be Sorted by Supplier (Click to view full-size image)

Summary

The custom paging implementation we examined in the preceding tutorial required that the order by which the results were to be sorted be specified at design time. In short, this meant that the custom paging implementation we implemented could not, at the same time, provide sorting capabilities. In this tutorial we overcame this limitation by extending the stored procedure from the first to include a `@sortExpression` input parameter by which the results could be sorted.

After creating this stored procedure and creating new methods in the DAL and BLL, we were able to implement a GridView that offered both sorting and custom paging by configuring the ObjectDataSource to pass in the GridView s current `SortExpression` property to the BLL `SelectMethod`.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Carlos Santos. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Creating a Customized Sorting User Interface (C#)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

When displaying a long list of sorted data, it can be very helpful to group related data by introducing separator rows. In this tutorial we'll see how to create such a sorting user interface.

Introduction

When displaying a long list of sorted data where there are only a handful of different values in the sorted column, an end user might find it hard to discern where, exactly, the difference boundaries occur. For example, there are 81 products in the database, but only nine different category choices (eight unique categories plus the `NULL` option). Consider the case of a user who is interested in examining the products that fall under the Seafood category. From a page that lists *all* of the products in a single GridView, the user might decide her best bet is to sort the results by category, which will group together all of the Seafood products together. After sorting by category, the user then needs to hunt through the list, looking for where the Seafood-grouped products start and end. Since the results are ordered alphabetically by the category name finding the Seafood products is not difficult, but it still requires closely scanning the list of items in the grid.

To help highlight the boundaries between sorted groups, many websites employ a user interface that adds a separator between such groups. Separators like the ones shown in Figure 1 enables a user to more quickly find a particular group and identify its boundaries, as well as ascertain what distinct groups exist in the data.

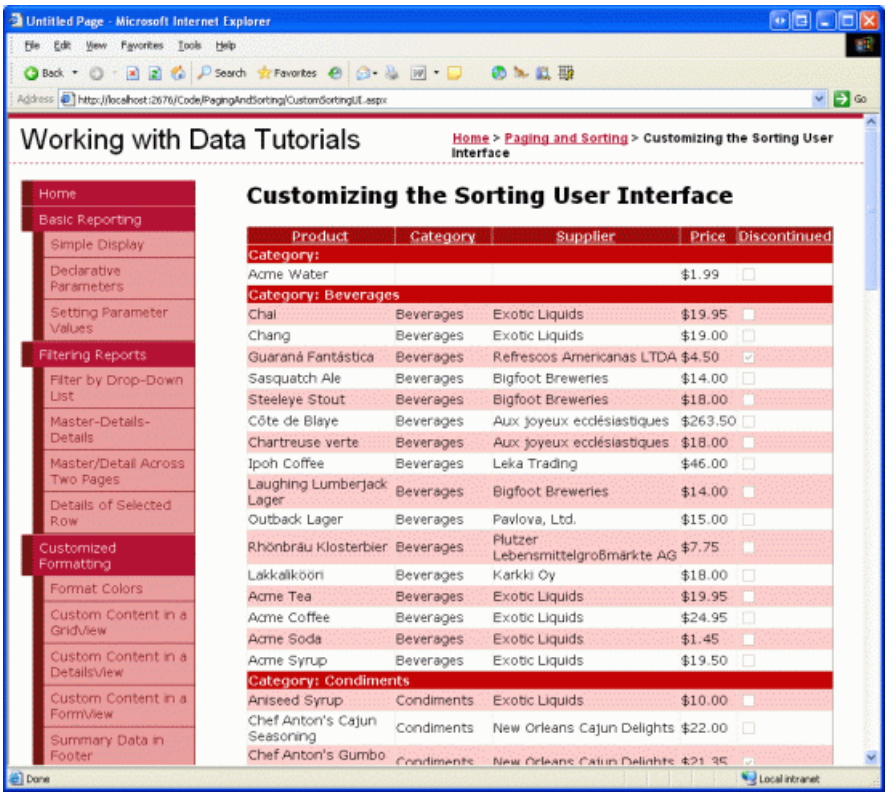


Figure 1: Each Category Group is Clearly Identified (Click to view full-size image)

In this tutorial we'll see how to create such a sorting user interface.

Step 1: Creating a Standard, Sortable GridView

Before we explore how to augment the GridView to provide the enhanced sorting interface, let s first create a standard, sortable GridView that lists the products. Start by opening the `CustomSortingUI.aspx` page in the `PagingAndSorting` folder. Add a GridView to the page, set its `ID` property to `ProductList`, and bind it to a new ObjectDataSource. Configure the ObjectDataSource to use the `ProductsBLL` class s `GetProducts()` method for selecting records.

Next, configure the GridView such that it only contains the `ProductName`, `CategoryName`, `SupplierName`, and `UnitPrice` BoundFields and the Discontinued CheckBoxField. Finally, configure the GridView to support sorting by checking the Enable Sorting checkbox in the GridView s smart tag (or by setting its `AllowSorting` property to `true`). After making these additions to the `CustomSortingUI.aspx` page, the declarative markup should look similar to the following:

```
<asp:GridView ID="ProductList" runat="server" AllowSorting="True"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
            ReadOnly="True" SortExpression="SupplierName" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:C}"
            HeaderText="Price" HtmlEncode="False" SortExpression="UnitPrice" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}" SelectMethod="GetProducts"
    TypeName="ProductsBLL"></asp:ObjectDataSource>
```

Take a moment to view our progress thus far in a browser. Figure 2 shows the sortable GridView when its data is sorted by category in alphabetical order.

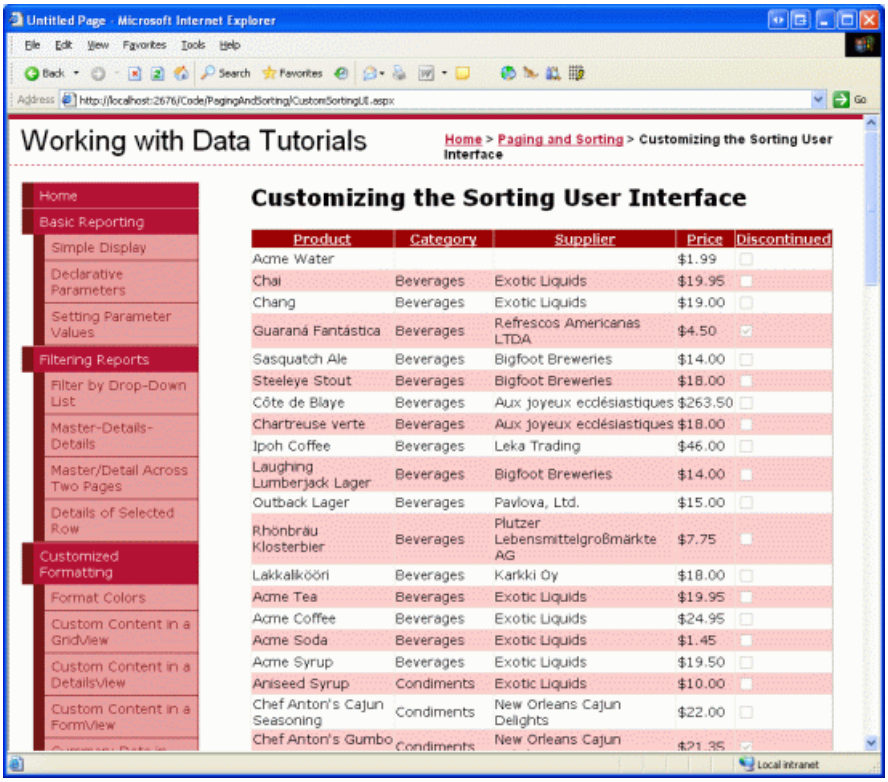


Figure 2: The Sortable GridView s Data is Ordered by Category (Click to view full-size image)

Step 2: Exploring Techniques for Adding the Separator Rows

With the generic, sortable GridView complete, all that remains is to be able to add the separator rows in the GridView before each unique sorted group. But how can such rows be injected into the GridView? Essentially, we need to iterate through the GridView s rows, determine where the differences occur between the values in the sorted column, and then add the appropriate separator row. When thinking about this problem, it seems natural that the solution lies somewhere in the GridView s `RowDataBound` event handler. As we discussed in the Custom Formatting Based Upon Data tutorial, this event handler is commonly used when applying row-level formatting based on the row s data. However, the `RowDataBound` event handler is not the solution here, as rows cannot be added to the GridView programmatically from this event handler. The GridView s `Rows` collection, in fact, is read-only.

To add additional rows to the GridView we have three choices:

- Add these metadata separator rows to the actual data that is bound to the GridView
- After the GridView has been bound to the data, add additional `TableRow` instances to the GridView s control collection
- Create a custom server control that extends the GridView control and overrides those methods responsible for constructing the GridView s structure

Creating a custom server control would be the best approach if this functionality was needed on many web pages or across several websites. However, it would entail quite a bit of code and a thorough exploration into the depths of the GridView s internal workings. Therefore, we'll not consider that option for this tutorial.

The other two options adding separator rows to the actual data being bound to the GridView and manipulating the GridView s control collection after its been bound - attack the problem differently and merit a discussion.

Adding Rows to the Data Bound to the GridView

When the GridView is bound to a data source, it creates a `GridViewRow` for each record returned by the data source. Therefore, we can inject the needed separator rows by adding separator records to the data source before binding it to the GridView. Figure 3 illustrates this concept.

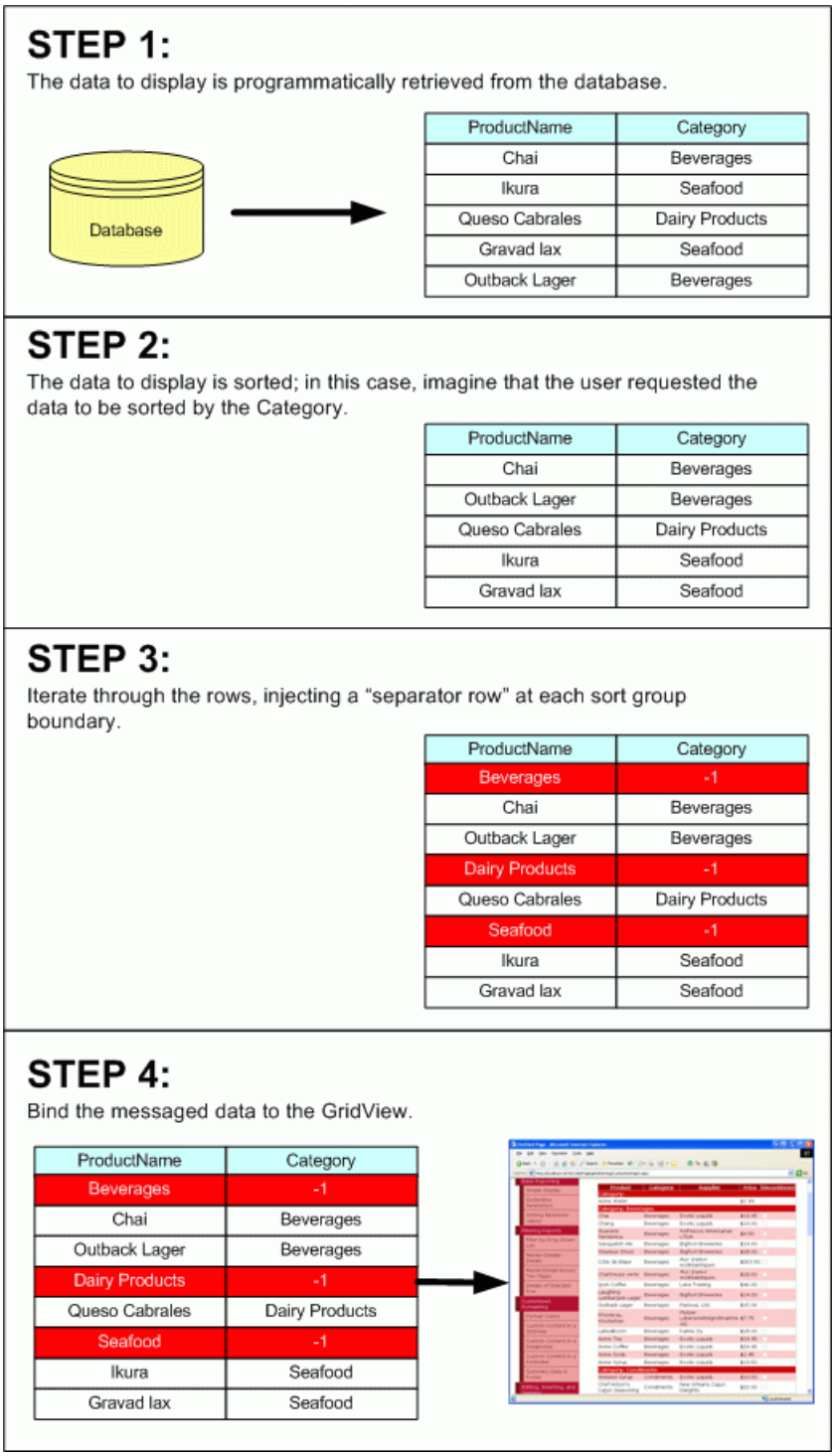


Figure 3: One Technique Involves Adding Separator Rows to the Data Source

I use the term separator records in quotes because there is no special separator record; rather, we must somehow flag that a particular record in the data source serves as a separator rather than a normal data row. For our examples, we re binding a `ProductsDataTable` instance to the GridView, which is composed of `ProductRows`. We might flag a record as a separator row by setting its `CategoryID` property to `-1` (since such a value couldn't exist normally).

To utilize this technique we d need to perform the following steps:

1. Programmatically retrieve the data to bind to the GridView (a `ProductsDataTable` instance)
2. Sort the data based on the GridView s `SortExpression` and `SortDirection` properties
3. Iterate through the `ProductsRows` in the `ProductsDataTable`, looking for where the differences in the sorted column lie
4. At each group boundary, inject a separator record `ProductsRow` instance into the DataTable, one that has it s

`CategoryID` set to `-1` (or whatever designation was decided upon to mark a record as a separator record)

5. After injecting the separator rows, programmatically bind the data to the GridView

In addition to these five steps, we d also need to provide an event handler for the GridView s `RowDataBound` event. Here, we d check each `DataRow` and determine if it was a separator row, one whose `CategoryID` setting was `-1`. If so, we d probably want to adjust its formatting or the text displayed in the cell(s).

Using this technique for injecting the sorting group boundaries requires a bit more work than outlined above, as you need to also provide an event handler for the GridView s `Sorting` event and keep track of the `SortExpression` and `SortDirection` values.

Manipulating the GridView s Control Collection After It s Been Databound

Rather than messaging the data before binding it to the GridView, we can add the separator rows *after* the data has been bound to the GridView. The process of data binding builds up the GridView s control hierarchy, which in reality is simply a `Table` instance composed of a collection of rows, each of which is composed of a collection of cells. Specifically, the GridView s control collection contains a `Table` object at its root, a `GridViewRow` (which is derived from the `TableRow` class) for each record in the `DataSource` bound to the GridView, and a `TableCell` object in each `GridViewRow` instance for each data field in the `DataSource`.

To add separator rows between each sorting group, we can directly manipulate this control hierarchy once it has been created. We can be confident that the GridView s control hierarchy has been created for the last time by the time the page is being rendered. Therefore, this approach overrides the `Page` class s `Render` method, at which point the GridView s final control hierarchy is updated to include the needed separator rows. Figure 4 illustrates this process.

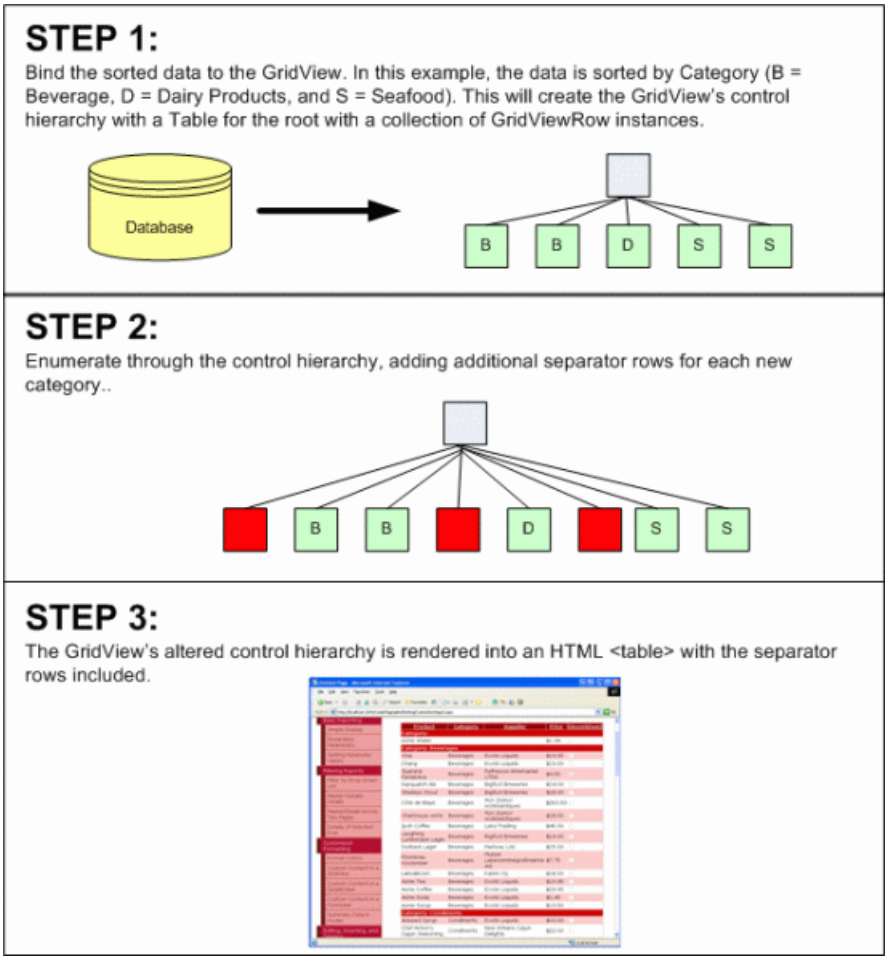


Figure 4: An Alternate Technique Manipulates the GridView s Control Hierarchy (Click to view full-size image)

For this tutorial, we'll use this latter approach to customize the sorting user experience.

> **NOTE**
> The code I m presenting in this tutorial is based on the example provided in Teemu Keiski s blog entry, Playing a Bit with GridView Sort Grouping.

Step 3: Adding the Separator Rows to the GridView s Control Hierarchy

Since we only want to add the separator rows to the GridView s control hierarchy after its control hierarchy has been created and created for the last time on that page visit, we want to perform this addition at the end of the page lifecycle, but before the actual GridView control hierarchy has been rendered into HTML. The latest possible point at which we can accomplish this is the `Page` class s `Render` event, which we can override in our code-behind class using the following method signature:

```
protected override void Render(HtmlTextWriter writer)
{
    // Add code to manipulate the GridView control hierarchy
    base.Render(writer);
}
```

When the `Page` class s original `Render` method is invoked `base.Render(writer)` each of the controls in the page will be rendered, generating the markup based on their control hierarchy. Therefore it is imperative that we both call `base.Render(writer)`, so that the page is rendered, and that we manipulate the GridView s control hierarchy prior to calling `base.Render(writer)`, so that the separator rows have been added to the GridView s control hierarchy before it s been rendered.

To inject the sort group headers we first need to ensure that the user has requested that the data be sorted. By default, the GridView s contents are not sorted, and therefore we don t need to enter any group sorting headers.

> **NOTE**
> If you want the GridView to be sorted by a particular column when the page is first loaded, call the GridView s `Sort` method on the first page visit (but not on subsequent postbacks). To accomplish this, add this call in the `Page_Load` event handler within an `if (!Page.IsPostBack)` conditional. Refer back to the Paging and Sorting Report Data tutorial information for more on the `Sort` method.

Assuming that the data has been sorted, our next task is to determine what column the data was sorted by and then to scan the rows looking for differences in that column s values. The following code ensures that the data has been sorted and finds the column by which the data has been sorted:

```
protected override void Render(HtmlTextWriter writer)
{
    // Only add the sorting UI if the GridView is sorted
    if (!string.IsNullOrEmpty(ProductList.SortExpression))
    {
        // Determine the index and HeaderText of the column that
        //the data is sorted by
        int sortColumnIndex = -1;
        string sortColumnHeaderText = string.Empty;
        for (int i = 0; i < ProductList.Columns.Count; i++)
        {
            if (ProductList.Columns[i].SortExpression.CompareTo(ProductList.SortExpression)
                == 0)
            {
                sortColumnIndex = i;
                sortColumnHeaderText = ProductList.Columns[i].HeaderText;
                break;
            }
        }
        // TODO: Scan the rows for differences in the sorted column's values
}
```

If the GridView has yet to be sorted, the GridView s `SortExpression` property will not have been set. Therefore, we only want to add the separator rows if this property has some value. If it does, we next need to determine the index of the column by which the data was sorted. This is accomplished by looping through the GridView s `Columns` collection, searching for the column whose `SortExpression` property equals the GridView s `SortExpression` property. In addition to the column s index, we also grab the `HeaderText` property, which is used when displaying the separator rows.

With the index of the column by which the data is sorted, the final step is to enumerate the rows of the GridView. For each row we need to determine whether the sorted column s value differs from the previous row s sorted column s value. If so, we need to inject a new `GridViewRow` instance into the control hierarchy. This is accomplished with the following code:

```
protected override void Render(HtmlTextWriter writer)
{
    // Only add the sorting UI if the GridView is sorted
    if (!string.IsNullOrEmpty(ProductList.SortExpression))
    {
        // ... Code for finding the sorted column index removed for brevity ...
        // Reference the Table the GridView has been rendered into
        Table gridTable = (Table)ProductList.Controls[0];
        // Enumerate each TableRow, adding a sorting UI header if
        // the sorted value has changed
        string lastValue = string.Empty;
        foreach (GridViewRow gvr in ProductList.Rows)
        {
            string currentValue = gvr.Cells[sortColumnIndex].Text;
            if (lastValue.CompareTo(currentValue) != 0)
            {
                // there's been a change in value in the sorted column
                int rowIndex = gridTable.Rows.GetRowIndex(gvr);
                // Add a new sort header row
                GridViewRow sortRow = new GridViewRow(rowIndex, rowIndex,
                    DataControlRowType.DataRow, DataControlRowState.Normal);
                TableCell sortCell = new TableCell();
                sortCell.ColumnSpan = ProductList.Columns.Count;
                sortCell.Text = string.Format("{0}: {1}",
                    sortColumnHeaderText, currentValue);
                sortCell.CssClass = "SortHeaderRowStyle";
                // Add sortCell to sortRow, and sortRow to gridTable
                sortRow.Cells.Add(sortCell);
                gridTable.Controls.AddAt(rowIndex, sortRow);
                // Update lastValue
                lastValue = currentValue;
            }
        }
    }
    base.Render(writer);
}
```

This code starts by programmatically referencing the `Table` object found at the root of the GridView s control hierarchy and creating a string variable named `lastValue`. `lastValue` is used to compare the current row s sorted column value with the previous row s value. Next, the GridView s `Rows` collection is enumerated and for each row the value of the sorted column is stored in the `currentValue` variable.

> **NOTE**
> To determine the value of the particular row s sorted column I use the cell s `Text` property. This works well for BoundFields, but will not work as desired for TemplateFields, CheckBoxFields, and so on. We'll look at how to account for alternate GridView fields shortly.

The `currentValue` and `lastValue` variables are then compared. If they differ we need to add a new separator row to the control hierarchy. This is accomplished by determining the index of the `GridViewRow` in the `Table` object s `Rows` collection, creating new `GridViewRow` and `TableCell` instances, and then adding the `TableCell` and `GridViewRow` to the control hierarchy.

Note that the separator row s lone `TableCell` is formatted such that it spans the entire width of the GridView, is formatted using the `SortHeaderRowStyle` CSS class, and has its `Text` property such that it shows both the sort group name (such as Category) and the group s value (such as Beverages). Finally, `lastValue` is updated to the value of `currentValue`.

The CSS class used to format the sorting group header row `SortHeaderRowStyle` needs to be specified in the `Styles.css` file. Feel free to use whatever style settings appeal to you; I used the following:

```
.SortHeaderRowStyle
{
    background-color: #c00;
    text-align: left;
    font-weight: bold;
    color: White;
}
```

With the current code, the sorting interface adds sort group headers when sorting by any BoundField (see Figure 5, which shows a screenshot when sorting by supplier). However, when sorting by any other field type (such as a CheckBoxField or TemplateField), the sort group headers are nowhere to be found (see Figure 6).

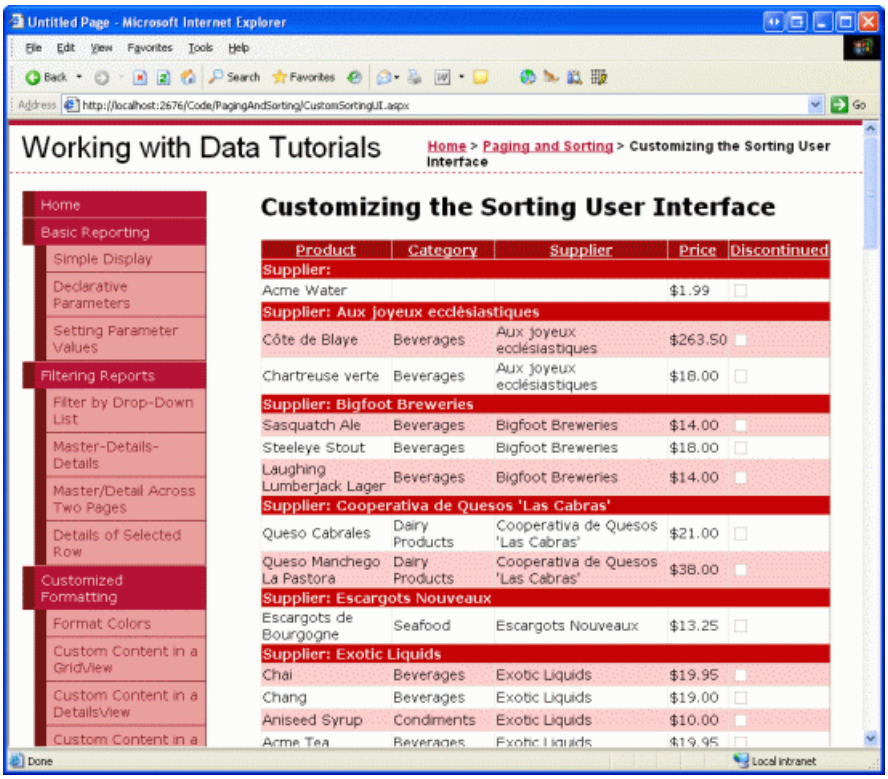


Figure 5: The Sorting Interface Includes Sort Group Headers When Sorting by BoundFields (Click to view full-size image)

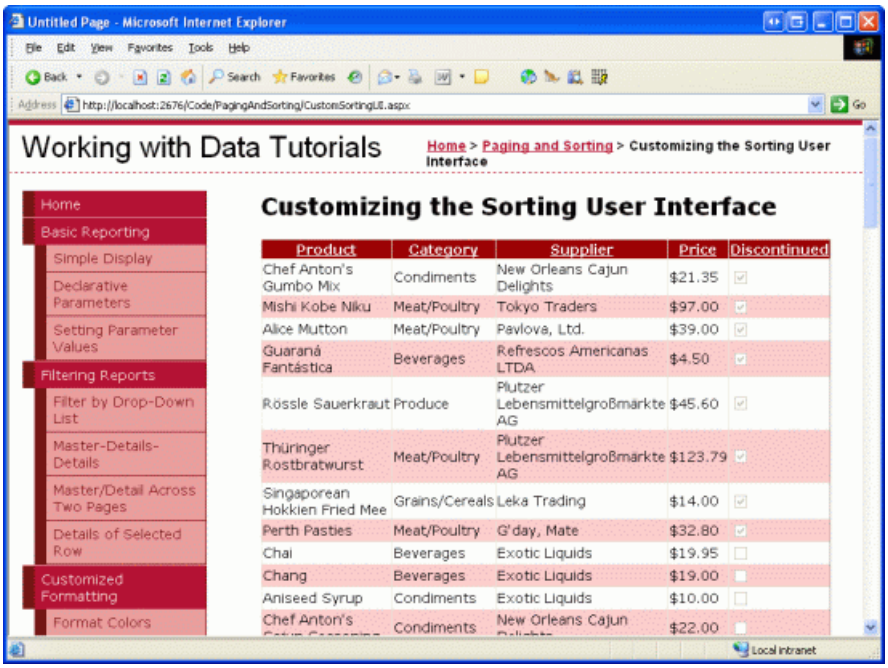


Figure 6: The Sort Group Headers are Missing When Sorting a CheckBoxField (Click to view full-size image)

The reason the sort group headers are missing when sorting by a CheckBoxField is because the code currently uses just the `TableCell` s `Text` property to determine the value of the sorted column for each row. For CheckBoxFields, the `TableCell` s `Text` property is an empty string; instead, the value is available through a CheckBox Web control that resides within the `TableCell` s `Controls` collection.

To handle field types other than BoundFields, we need to augment the code where the `currentValue` variable is assigned to check for the existence of a CheckBox in the `TableCell` s `Controls` collection. Instead of using `currentValue = gvr.Cells[sortColumnIndex].Text`, replace this code with the following:

```
string currentValue = string.Empty;
if (gvr.Cells[sortColumnIndex].Controls.Count > 0)
{
    if (gvr.Cells[sortColumnIndex].Controls[0] is CheckBox)
    {
        if (((CheckBox)gvr.Cells[sortColumnIndex].Controls[0]).Checked)
            currentValue = "Yes";
        else
            currentValue = "No";
    }
    // ... Add other checks here if using columns with other
    //      Web controls in them (Calendars, DropDownLists, etc.) ...
}
else
    currentValue = gvr.Cells[sortColumnIndex].Text;
```

This code examines the sorted column `TableCell` for the current row to determine if there are any controls in the `Controls` collection. If there are, and the first control is a CheckBox, the `currentValue` variable is set to Yes or No, depending on the CheckBox s `Checked` property. Otherwise, the value is taken from the `TableCell` s `Text` property. This logic can be replicated to handle sorting for any TemplateFields that may exist in the GridView.

With the above code addition, the sort group headers are now present when sorting by the Discontinued CheckBoxField (see Figure 7).

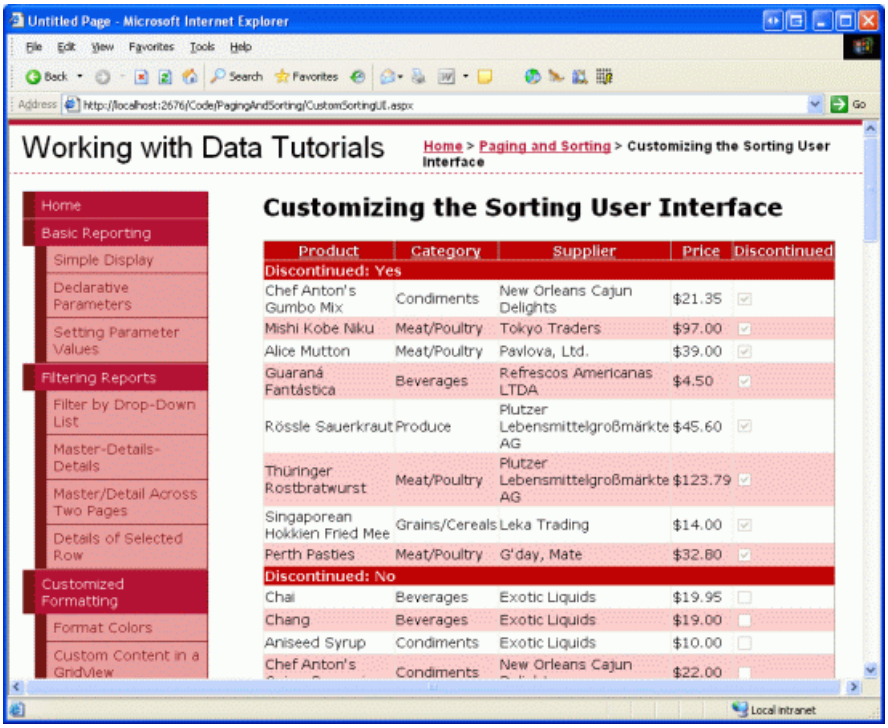


Figure 7: The Sort Group Headers are Now Present When Sorting a CheckBoxField (Click to view full-size image)

NOTE

If you have products with `NULL` database values for the `CategoryID`, `SupplierID`, or `UnitPrice` fields, those values will appear as empty strings in the GridView by default, meaning the separator row s text for those products with `NULL` values will read like Category: (that is, there s no name after Category: like with Category: Beverages). If you want a value displayed here you can either set the BoundFields `NullDisplayText` property to the text you want displayed or you can add a conditional statement in the Render method when assigning the `currentValue` to the separator row s `Text` property.

Summary

The GridView does not include many built-in options for customizing the sorting interface. However, with a bit of low-level code, it s possible to tweak the GridView s control hierarchy to create a more customized interface. In this tutorial we saw how to add a sort group separator row for a sortable GridView, which more easily identifies the distinct groups and those groups boundaries. For additional examples of customized sorting interfaces, check out Scott Guthrie s A Few ASP.NET 2.0 GridView Sorting Tips and Tricks blog entry.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Paging and Sorting Report Data (VB)

2/20/2020 • 19 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> Paging and sorting are two very common features when displaying data in an online application. In this tutorial we'll take a first look at adding sorting and paging to our reports, which we will then build upon in future tutorials.

Introduction

Paging and sorting are two very common features when displaying data in an online application. For example, when searching for ASP.NET books at an online bookstore, there may be hundreds of such books, but the report listing the search results lists only ten matches per page. Moreover, the results can be sorted by title, price, page count, author name, and so on. While the past 23 tutorials have examined how to build a variety of reports, including interfaces that permit adding, editing, and deleting data, we ve not looked at how to sort data and the only paging examples we ve seen have been with the DetailsView and FormView controls.

In this tutorial we'll see how to add sorting and paging to our reports, which can be accomplished by simply checking a few checkboxes. Unfortunately, this simplistic implementation has its drawbacks the sorting interface leaves a bit to be desired and the paging routines are not designed for efficiently paging through large result sets. Future tutorials will explore how to overcome the limitations of the out-of-the-box paging and sorting solutions.

Step 1: Adding the Paging and Sorting Tutorial Web Pages

Before we start this tutorial, let s first take a moment to add the ASP.NET pages we'll need for this tutorial and the next three. Start by creating a new folder in the project named `PagingAndSorting`. Next, add the following five ASP.NET pages to this folder, having all of them configured to use the master page `Site.master`:

- `Default.aspx`
- `SimplePagingSorting.aspx`
- `EfficientPaging.aspx`
- `SortParameter.aspx`
- `CustomSortingUI.aspx`

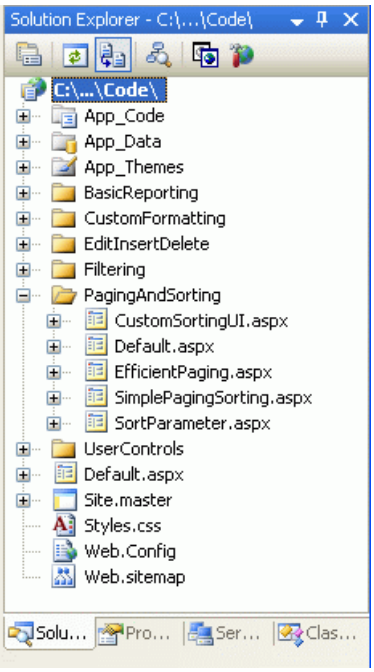


Figure 1: Create a PagingAndSorting Folder and Add the Tutorial ASP.NET Pages

Next, open the `Default.aspx` page and drag the `SectionLevelTutorialListing.ascx` User Control from the `UserControls` folder onto the Design surface. This User Control, which we created in the Master Pages and Site Navigation tutorial, enumerates the site map and displays those tutorials in the current section in a bulleted list.

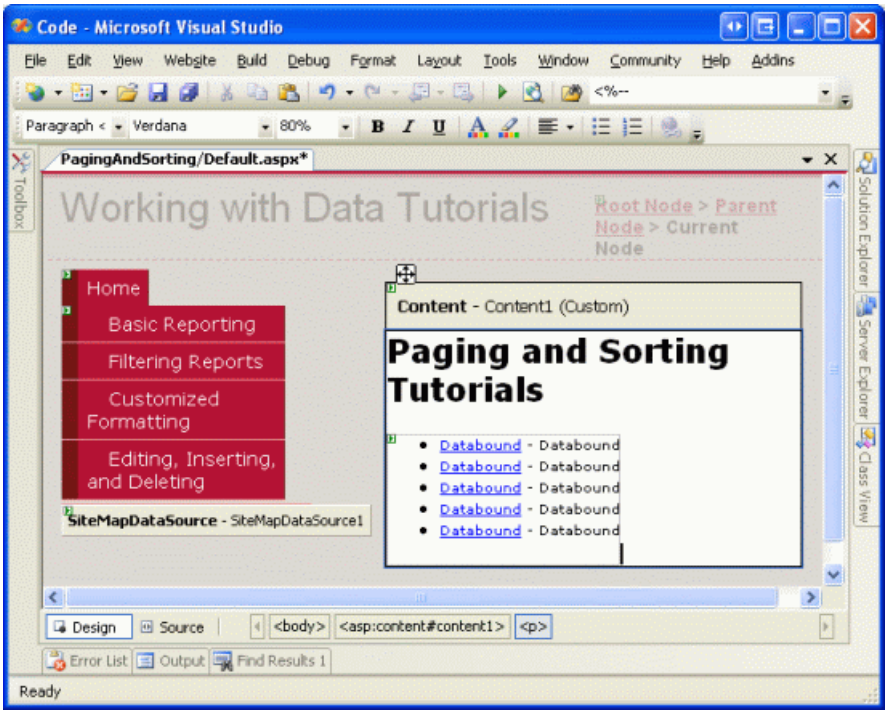


Figure 2: Add the SectionLevelTutorialListing.ascx User Control to Default.aspx

In order to have the bulleted list display the paging and sorting tutorials we'll be creating, we need to add them to the site map. Open the `Web.sitemap` file and add the following markup after the Editing, Inserting, and Deleting site map node markup:

```
<siteMapNode title="Paging and Sorting" url="~/PagingAndSorting/Default.aspx"
    description="Samples of Reports that Provide Paging and Sorting Capabilities">
    <siteMapNode url="~/PagingAndSorting/SimplePagingSorting.aspx"
        title="Simple Paging & Sorting Examples"
        description="Examines how to add simple paging and sorting support." />
    <siteMapNode url="~/PagingAndSorting/EfficientPaging.aspx"
        title="Efficiently Paging Through Large Result Sets"
        description="Learn how to efficiently page through large result sets." />
    <siteMapNode url="~/PagingAndSorting/SortParameter.aspx"
        title="Sorting Data at the BLL or DAL"
        description="Illustrates how to perform sorting logic in the Business Logic
        Layer or Data Access Layer." />
    <siteMapNode url="~/PagingAndSorting/CustomSortingUI.aspx"
        title="Customizing the Sorting User Interface"
        description="Learn how to customize and improve the sorting user interface." />
</siteMapNode>
```

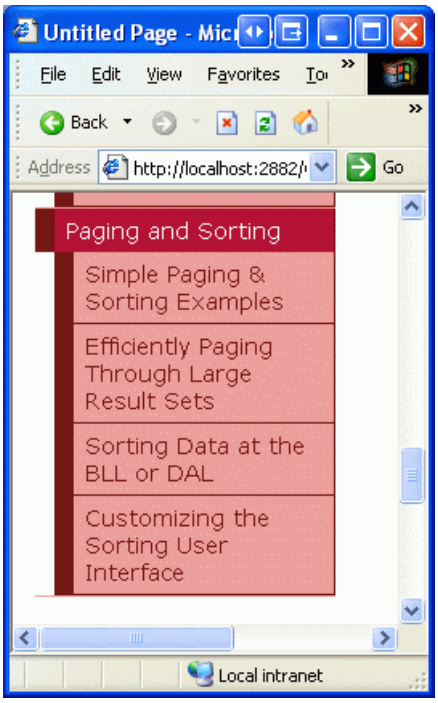


Figure 3: Update the Site Map to Include the New ASP.NET Pages

Step 2: Displaying Product Information in a GridView

Before we actually implement paging and sorting capabilities, let s first create a standard non-sortable, non-pageable GridView that lists the product information. This is a task we ve done many times before throughout this tutorial series so these steps should be familiar. Start by opening the `SimplePagingSorting.aspx` page and drag a GridView control from the Toolbox onto the Designer, setting its `ID` property to `Products`. Next, create a new ObjectDataSource that uses the ProductsBLL class s `GetProducts()` method to return all of the product information.

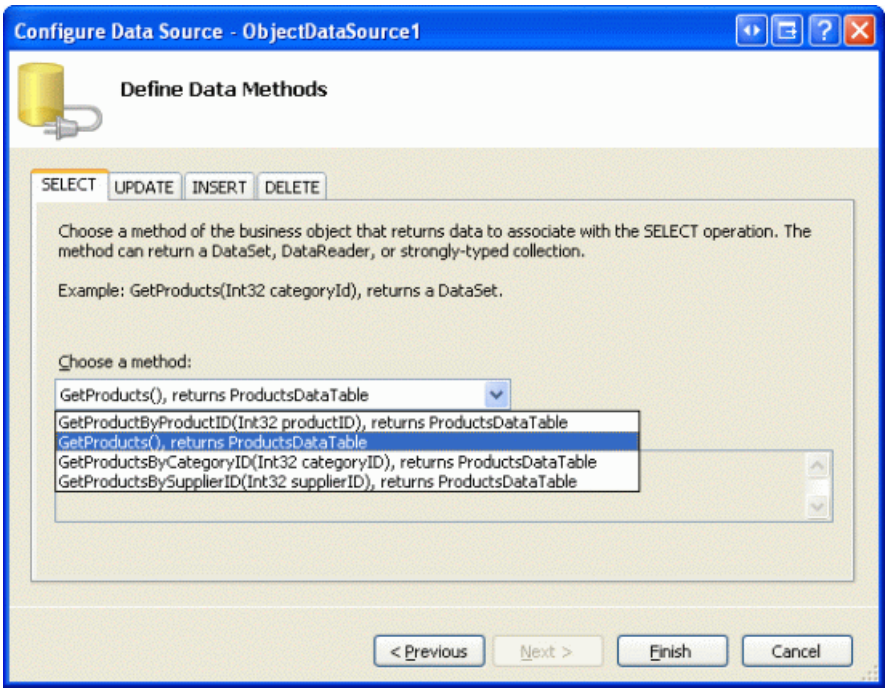


Figure 4: Retrieve Information About All of the Products Using the GetProducts() Method

Since this report is a read-only report, there s no need to map the ObjectDataSource s `Insert()`, `Update()`, or `Delete()` methods to corresponding `ProductsBLL` methods; therefore, choose (None) from the drop-down list for the UPDATE, INSERT, and DELETE tabs.

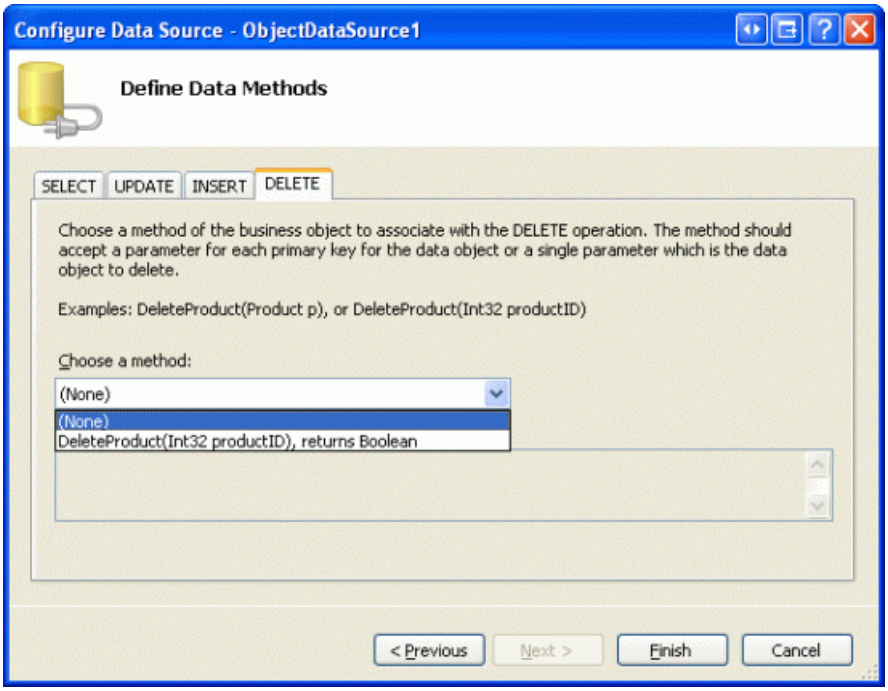


Figure 5: Choose the (None) Option in the Drop-Down List in the UPDATE, INSERT, and DELETE Tabs

Next, let s customize the GridView s fields so that only the products names, suppliers, categories, prices, and discontinued statuses are displayed. Furthermore, feel free to make any field-level formatting changes, such as adjusting the `HeaderText` properties or formatting the price as a currency. After these changes, your GridView s declarative markup should look similar to the following:

```
<asp:GridView ID="Products" AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ObjectDataSource1" EnableViewState="False" runat="server">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            SortExpression="CategoryName" ReadOnly="True" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
            SortExpression="SupplierName" ReadOnly="True" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
            SortExpression="UnitPrice" DataFormatString="{0:C}"
            HtmlEncode="False" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
```

Figure 6 shows our progress thus far when viewed through a browser. Note that the page lists all of the products in one screen, showing each product s name, category, supplier, price, and discontinued status.

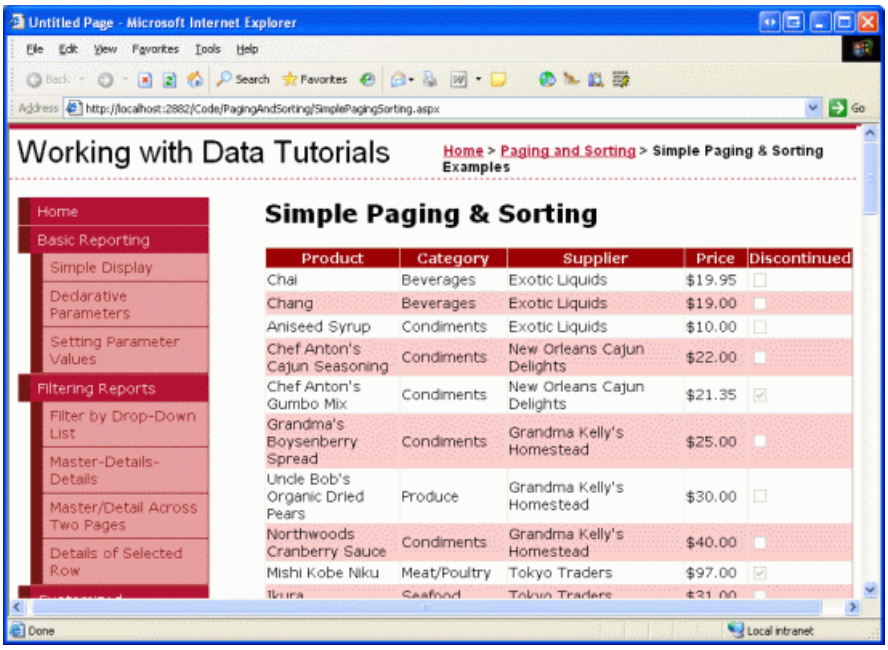


Figure 6: Each of the Products are Listed (Click to view full-size image)

Step 3: Adding Paging Support

Listing *all* of the products on one screen can lead to information overload for the user perusing the data. To help make the results more manageable, we can break up the data into smaller pages of data and allow the user to step through the data one page at a time. To accomplish this simply check the Enable Paging checkbox from the GridView s smart tag (this sets the GridView s `AllowPaging` property to `true`).

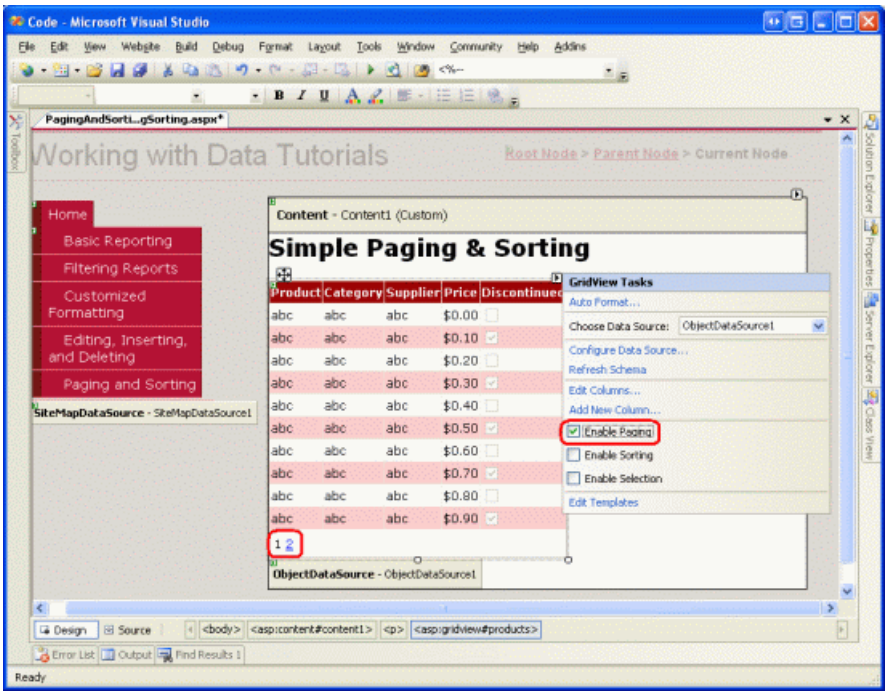


Figure 7: Check the Enable Paging Checkbox to Add Paging Support (Click to view full-size image)

Enabling paging limits the number of records shown per page and adds a *paging interface* to the GridView. The default paging interface, shown in Figure 7, is a series of page numbers, allowing the user to quickly navigate from one page of data to another. This paging interface should look familiar, as we ve seen it when adding paging support to the DetailsView and FormView controls in past tutorials.

Both the DetailsView and FormView controls only show a single record per page. The GridView, however, consults its `PageSize` property to determine how many records to show per page (this property defaults to a value of 10).

This GridView, DetailsView, and FormView s paging interface can be customized using the following properties:

- `PagerStyle` indicates the style information for the paging interface; can specify settings like `BackColor`, `ForeColor`, `CssClass`, `HorizontalAlign`, and so on.
- `PagerSettings` contains a bevy of properties that can customize the functionality of the paging interface; `PageButtonCount` indicates the maximum number of numeric page numbers displayed in the paging interface (the default is 10); the `Mode` property indicates how the paging interface operates and can be set to:
  - `NextPrevious` shows a Next and Previous buttons, allowing the user to step forwards or backwards one page at a time
  - `NextPreviousFirstLast` in addition to Next and Previous buttons, First and Last buttons are also included, allowing the user to quickly move to the first or last page of data
  - `Numeric` shows a series of page numbers, allowing the user to immediately jump to any page
  - `NumericFirstLast` in addition to the page numbers, includes First and Last buttons, allowing the user to quickly move to the first or last page of data; the First/Last buttons are only shown if all of the numeric page numbers cannot fit

Moreover, the GridView, DetailsView, and FormView all offer the `PageIndex` and `PageCount` properties, which indicate the current page being viewed and the total number of pages of data, respectively. The `PageIndex` property is indexed starting at 0, meaning that when viewing the first page of data `PageIndex` will equal 0. `PageCount`, on the other hand, starts counting at 1, meaning that `PageIndex` is limited to the values between 0 and `PageCount - 1`.

Let s take a moment to improve the default appearance of our GridView s paging interface. Specifically, let s have

the paging interface right-aligned with a light gray background. Rather than setting these properties directly through the GridView s `PagerStyle` property, let s create a CSS class in `Styles.css` named `PagerRowStyle` and then assign the `PagerStyle` s `CssClass` property through our Theme. Start by opening `Styles.css` and adding the following CSS class definition:

```
.PagerRowStyle
{
    background-color: #ddd;
    text-align: right;
}
```

Next, open the `GridView.skin` file in the `DataWebControls` folder within the `App_Themes` folder. As we discussed in the *Master Pages and Site Navigation* tutorial, Skin files can be used to specify the default property values for a Web control. Therefore, augment the existing settings to include setting the `PagerStyle` s `CssClass` property to `PagerRowStyle`. Also, let s configure the paging interface to show at most five numeric page buttons using the `NumericFirstLast` paging interface.

```
<asp:GridView runat="server" CssClass="DataWebControlStyle">
   <AlternatingRowStyle CssClass="AlternatingRowStyle" />
   <RowStyle CssClass="RowStyle" />
   <HeaderStyle CssClass="HeaderStyle" />
   <FooterStyle CssClass="FooterStyle" />
   <SelectedRowStyle CssClass="SelectedRowStyle" />
   <PagerStyle CssClass="PagerRowStyle" />
   <PagerSettings Mode="NumericFirstLast" PageButtonCount="5" />
</asp:GridView>
```

The Paging User Experience

Figure 8 shows the web page when visited through a browser after the GridView s Enable Paging checkbox has been checked and the `PagerStyle` and `PagerSettings` configurations have been made through the `GridView.skin` file. Note how only ten records are shown, and the paging interface indicates that we are viewing the first page of data.

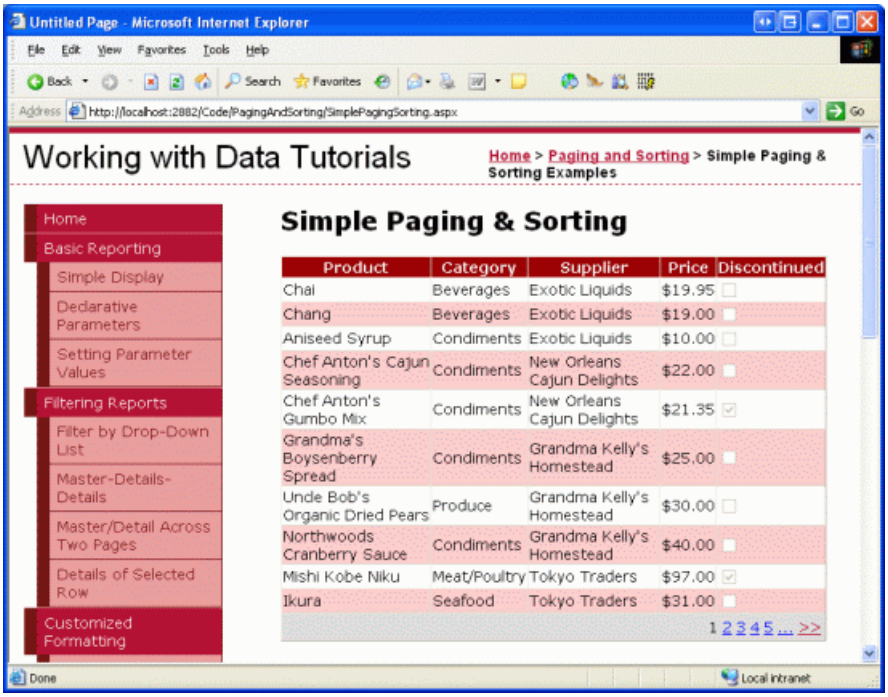


Figure 8: With Paging Enabled, Only a Subset of the Records are Displayed at a Time (Click to view full-size image)

When the user clicks on one of the page numbers in the paging interface, a postback ensues and the page reloads showing that requested page s records. Figure 9 shows the results after opting to view the final page of data. Notice that the final page only has one record; this is because there are 81 records in total, resulting in eight pages of 10 records per page plus one page with a lone record.

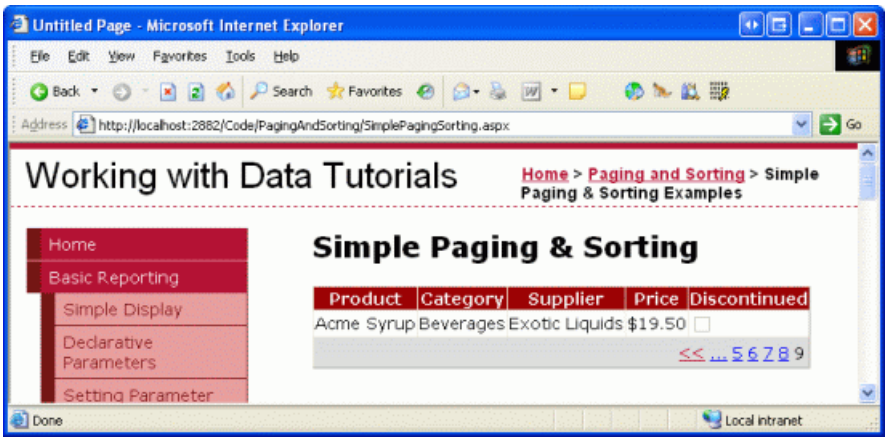


Figure 9: Clicking On a Page Number Causes a Postback and Shows the Appropriate Subset of Records (Click to view full-size image)

Paging s Server-Side Workflow

When the end user clicks on a button in the paging interface, a postback ensues and the following server-side workflow begins:

1. The GridView s (or DetailsView or FormView) `PageIndexChanging` event fires
2. The ObjectDataSource re-requests *all* of the data from the BLL; the GridView s `PageIndex` and `PageSize` property values are used to determine what records returned from the BLL need to be displayed in the GridView
3. The GridView s `PageIndexChanged` event fires

In Step 2, the ObjectDataSource re-requests all of the data from its data source. This style of paging is commonly referred to as *default paging*, as it s the paging behavior used by default when setting the `AllowPaging` property to `true`. With default paging the data Web control naively retrieves all records for each page of data, even though only a subset of records are actually rendered into the HTML that s sent to the browser. Unless the database data is cached by the BLL or ObjectDataSource, default paging is unworkable for sufficiently large result sets or for web applications with many concurrent users.

In the next tutorial we'll examine how to implement *custom paging*. With custom paging you can specifically instruct the ObjectDataSource to only retrieve the precise set of records needed for the requested page of data. As you can imagine, custom paging greatly improves the efficiency of paging through large result sets.

> **NOTE**
> While default paging is not suitable when paging through sufficiently large result sets or for sites with many simultaneous users, realize that custom paging requires more changes and effort to implement and is not as simple as checking a checkbox (as is default paging). Therefore, default paging may be the ideal choice for small, low-traffic websites or when paging through relatively small result sets, as it s much easier and quicker to implement.

For example, if we know that we'll never have more than 100 products in our database, the minimal performance gain enjoyed by custom paging is likely offset by the effort required to implement it. If, however, we may one day have thousands or tens of thousands of products, *not* implementing custom paging would greatly hamper the scalability of our application.

Step 4: Customizing the Paging Experience

The data Web controls provide a number of properties that can be used to enhance the user s paging experience. The `PageCount` property, for example, indicates how many total pages there are, while the `PageIndex` property indicates the current page being visited and can be set to quickly move a user to a specific page. To illustrate how to use these properties to improve upon the user s paging experience, let s add a Label Web control to our page that informs the user what page they re currently visiting, along with a DropDownList control that allows them to quickly jump to any given page.

First, add a Label Web control to your page, set its `ID` property to `PagingInformation`, and clear out its `Text` property. Next, create an event handler for the GridView s `DataBound` event and add the following code:

```
Protected Sub Products_DataBound(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles Products.DataBound
    PagingInformation.Text = String.Format("You are viewing page {0} of {1}...", _
        Products.PageIndex + 1, Products.PageCount)
End Sub
```

This event handler assigns the `PagingInformation` Label s `Text` property to a message informing the user the page they are currently visiting `Products.PageIndex + 1` out of how many total pages `Products.PageCount` (we add 1 to the `Products.PageIndex` property because `PageIndex` is indexed starting at 0). I chose the assign this Label s `Text` property in the `DataBound` event handler as opposed to the `PageIndexChanged` event handler because the `DataBound` event fires every time data is bound to the GridView whereas the `PageIndexChanged` event handler only fires when the page index is changed. When the GridView is initially data bound on the first page visit, the `PageIndexChanging` event doesn t fire (whereas the `DataBound` event does).

With this addition, the user is now shown a message indicating what page they are visiting and how many total pages of data there are.

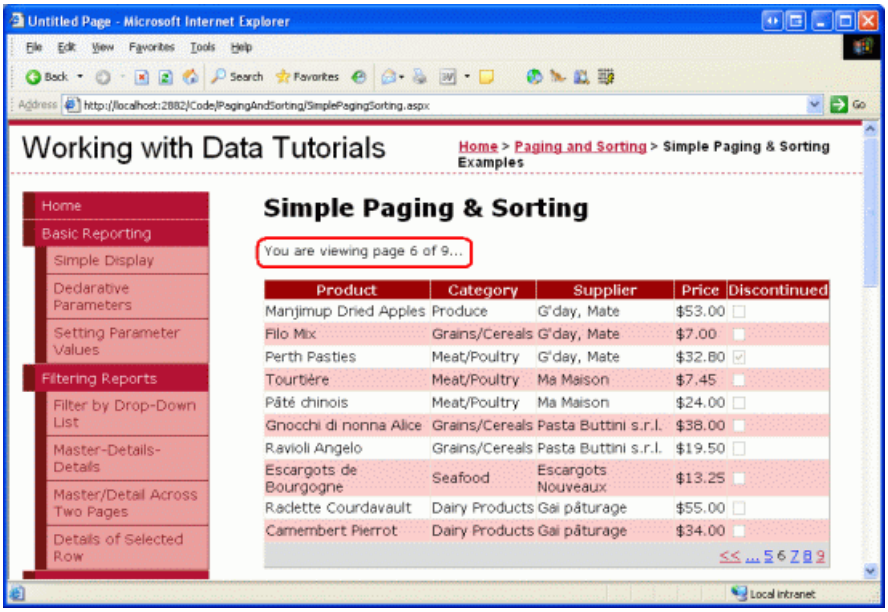


Figure 10: The Current Page Number and Total Number of Pages are Displayed (Click to view full-size image)

In addition to the Label control, let s also add a DropDownList control that lists the page numbers in the GridView with the currently viewed page selected. The idea here is that the user can quickly jump from the current page to another by simply selecting the new page index from the DropDownList. Start by adding a DropDownList to the Designer, setting its `ID` property to `PageList` and checking the Enable AutoPostBack option from its smart tag.

Next, return to the `DataBound` event handler and add the following code:

```
' Clear out all of the items in the DropDownList
PageList.Items.Clear()
' Add a ListItem for each page
For i As Integer = 0 To Products.PageCount - 1
    ' Add the new ListItem
    Dim pageListItem As New ListItem(String.Concat("Page ", i + 1), i.ToString())
    PageList.Items.Add(pageListItem)
    ' select the current item, if needed
    If i = Products.PageIndex Then
        pageListItem.Selected = True
    End If
Next
```

This code begins by clearing out the items in the `PageList` DropDownList. This may seem superfluous, since one wouldn't expect the number of pages to change, but other users may be using the system simultaneously, adding or removing records from the `Products` table. Such insertions or deletions could alter the number of pages of data.

Next, we need to create the page numbers again and have the one that maps to the current GridView `PageIndex` selected by default. We accomplish this with a loop from 0 to `PageCount - 1`, adding a new `ListItem` in each iteration and setting its `Selected` property to true if the current iteration index equals the GridView s `PageIndex` property.

Finally, we need to create an event handler for the DropDownList s `SelectedIndexChanged` event, which fires each time the user pick a different item from the list. To create this event handler, simply double-click the DropDownList in the Designer, then add the following code:

```
Protected Sub PageList_SelectedIndexChanged(sender As Object, e As System.EventArgs) _
    Handles PageList.SelectedIndexChanged
        ' Jump to the specified page
        Products.PageIndex = Convert.ToInt32(PageList.SelectedValue)
End Sub
```

As Figure 11 shows, merely changing the GridView s `PageIndex` property causes the data to be rebound to the GridView. In the GridView s `DataBound` event handler, the appropriate DropDownList `ListItem` is selected.

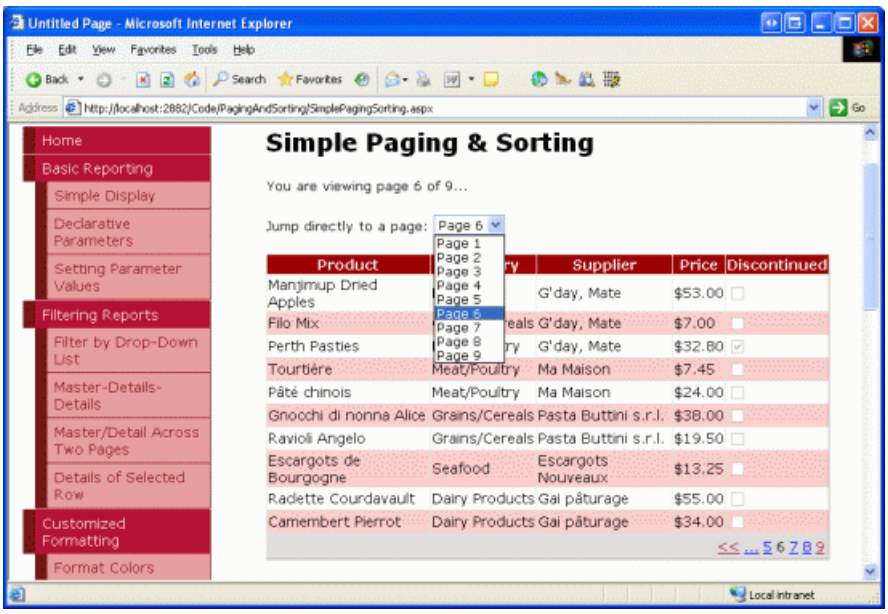


Figure 11: The User is Automatically Taken to the Sixth Page When Selecting the Page 6 Drop-Down List Item (Click to view full-size image)

Step 5: Adding Bi-Directional Sorting Support

Adding bi-directional sorting support is as simple as adding paging support simply check the Enable Sorting option from the GridView s smart tag (which sets the GridView s `AllowSorting` property to `true`). This renders each of the headers of the GridView s fields as LinkButtons that, when clicked, cause a postback and return the data sorted by the clicked column in ascending order. Clicking the same header LinkButton again re-sorts the data in descending order.

> **NOTE**
>
> If you are using a custom Data Access Layer rather than a Typed DataSet, you may not have an Enable Sorting option in the GridView s smart tag. Only GridViews bound to data sources that natively support sorting have this checkbox available. The Typed DataSet provides out-of-the-box sorting support since the ADO.NET DataTable provides a `Sort` method that, when invoked, sorts the DataTable s DataRows using the criteria specified.

If your DAL does not return objects that natively support sorting you will need to configure the ObjectDataSource to pass sorting information to the Business Logic Layer, which can either sort the data or have the data sorted by the DAL. We'll explore how to sort data at the Business Logic and Data Access Layers in a future tutorial.

The sorting LinkButtons are rendered as HTML hyperlinks, whose current colors (blue for an unvisited link and a dark red for a visited link) clash with the background color of the header row. Instead, let s have all header row links displayed in white, regardless of whether they ve been visited or not. This can be accomplished by adding the following to the `Styles.css` class:

```
.HeaderStyle a, .HeaderStyle a:visited
{
    color: White;
}
```

This syntax indicates to use white text when displaying those hyperlinks within an element that uses the HeaderStyle class.

After this CSS addition, when visiting the page through a browser your screen should look similar to Figure 12. In particular, Figure 12 shows the results after the Price field s header link has been clicked.

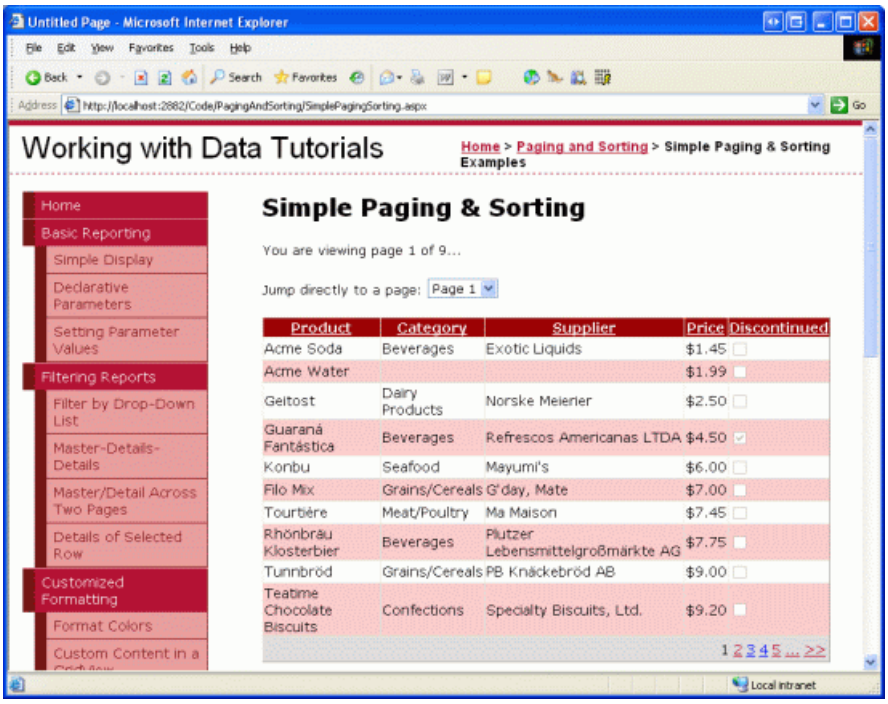


Figure 12: The Results Have Been Sorted by the UnitPrice in Ascending Order (Click to view full-size image)

Examining the Sorting Workflow

All GridView fields the BoundField, CheckBoxField, TemplateField, and so on have a `SortExpression` property that indicates the expression that should be used to sort the data when that field s sorting header link is clicked. The GridView also has a `SortExpression` property. When a sorting header LinkButton is clicked, the GridView assigns that field s `SortExpression` value to its `SortExpression` property. Next, the data is re-retrieved from the ObjectDataSource and sorted according to the GridView s `SortExpression` property. The following list details the sequence of steps that transpires when an end user sorts the data in a GridView:

1. The GridView s Sorting event fires
2. The GridView s `SortExpression` property is set to the `SortExpression` of the field whose sorting header LinkButton was clicked
3. The ObjectDataSource re-retrieves all of the data from the BLL and then sorts the data using the GridView s `SortExpression`
4. The GridView s `PageIndex` property is reset to 0, meaning that when sorting the user is returned to the first page of data (assuming paging support has been implemented)
5. The GridView s `Sorted` event fires

Like with default paging, the default sorting option re-retrieves *all* of the records from the BLL. When using sorting without paging or when using sorting with default paging, there s no way to circumvent this performance hit (short of caching the database data). However, as we'll see in a future tutorial, it s possible to efficiently sort data when using custom paging.

When binding an ObjectDataSource to the GridView through the drop-down list in the GridView s smart tag, each GridView field automatically has its `SortExpression` property assigned to the name of the data field in the `ProductsRow` class. For example, the `ProductName` BoundField s `SortExpression` is set to `ProductName`, as shown in the following declarative markup:

```
<asp:BoundField DataField="ProductName" HeaderText="Product"
    SortExpression="ProductName" />
```

A field can be configured so that it s not sortable by clearing out its `SortExpression` property (assigning it to an empty string). To illustrate this, imagine that we didn't want to let our customers sort our products by price. The `UnitPrice` BoundField s `SortExpression` property can be removed either from the declarative markup or through the Fields dialog box (which is accessible by clicking on the Edit Columns link in the GridView s smart tag).

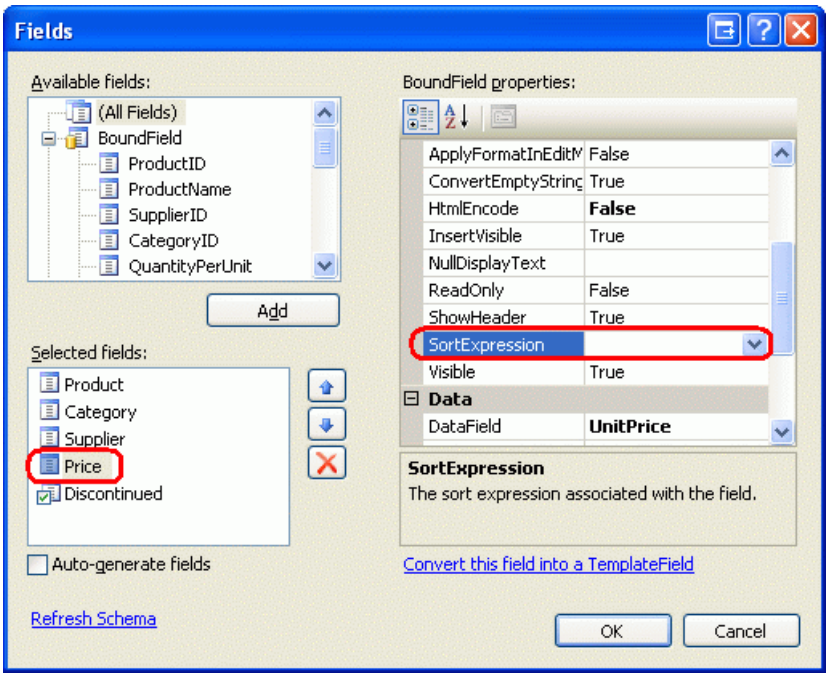


Figure 13: The Results Have Been Sorted by the UnitPrice in Ascending Order

Once the `SortExpression` property has been removed for the `UnitPrice` BoundField, the header is rendered as text rather than as a link, thereby preventing users from sorting the data by price.

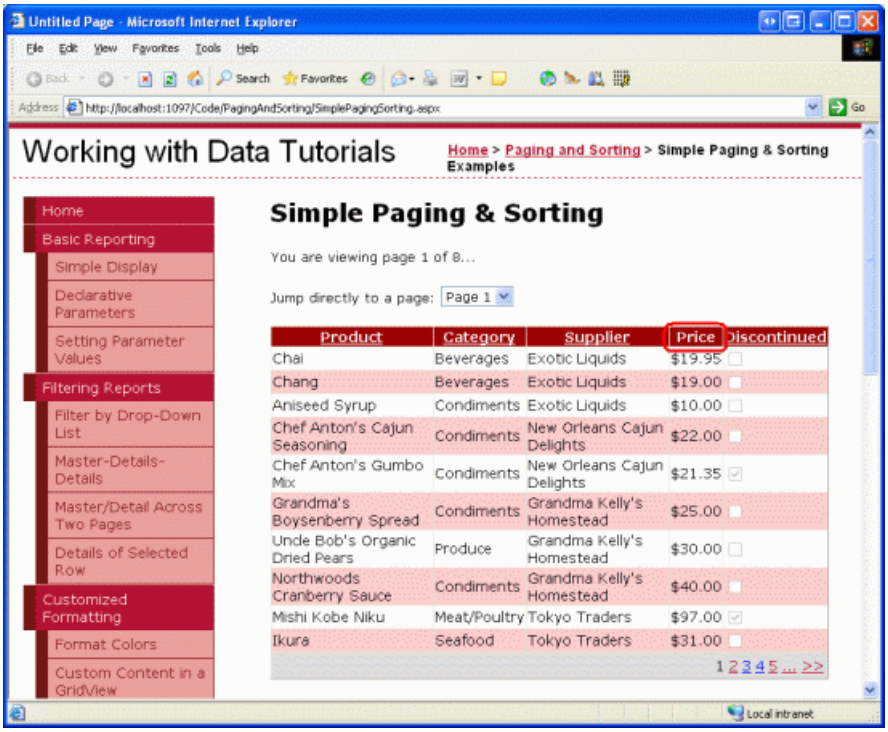


Figure 14: By Removing the SortExpression Property, Users Can No Longer Sort the Products By Price (Click to view full-size image)

Programmatically Sorting the GridView

You can also sort the contents of the GridView programmatically by using the GridView s `Sort` method. Simply pass in the `SortExpression` value to sort by along with the `SortDirection` (`Ascending` or `Descending`), and the GridView s data will be re-sorted.

Imagine that the reason we turned off sorting by the `UnitPrice` was because we were worried that our customers would simply buy only the lowest-priced products. However, we want to encourage them to buy the most expensive products, so we d like them to be able to sort the products by price, but only from the most

expensive price to the least.

To accomplish this add a Button Web control to the page, set its `ID` property to `SortPriceDescending`, and its `Text` property to Sort by Price. Next, create an event handler for the Button s `Click` event by double-clicking the Button control in the Designer. Add the following code to this event handler:

```
Protected Sub SortPriceDescending_Click(sender As Object, e As System.EventArgs) _
    Handles SortPriceDescending.Click
        'Sort by UnitPrice in descending order
        Products.Sort("UnitPrice", SortDirection.Descending)
End Sub
```

Clicking this Button returns the user to the first page with the products sorted by price, from most expensive to least expensive (see Figure 15).

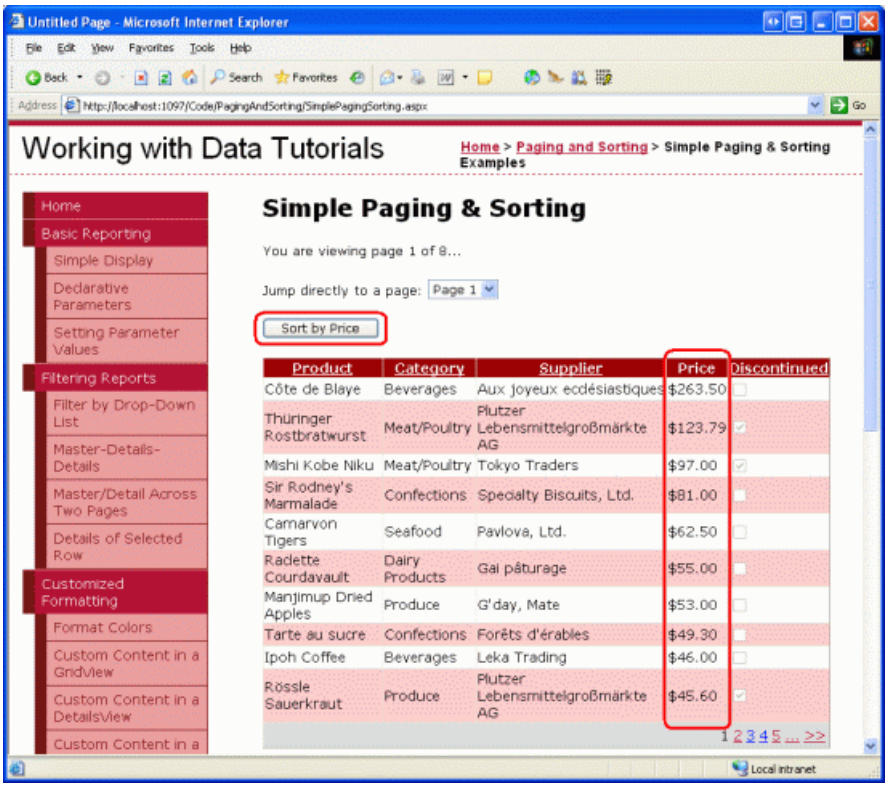


Figure 15: Clicking the Button Orders the Products From the Most Expensive to the Least (Click to view full-size image)

Summary

In this tutorial we saw how to implement default paging and sorting capabilities, both of which were as easy as checking a checkbox! When a user sorts or pages through data, a similar workflow unfolds:

1. A postback ensues
2. The data Web control s pre-level event fires (`PageIndexChanging` or `Sorting`)
3. All of the data is re-retrieved by the ObjectDataSource
4. The data Web control s post-level event fires (`PageIndexChanged` or `Sorted`)

While implementing basic paging and sorting is a breeze, more effort must be exerted to utilize the more efficient custom paging or to further enhance the paging or sorting interface. Future tutorials will explore these topics.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Efficiently Paging Through Large Amounts of Data (VB)

9/16/2020 • 29 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> The default paging option of a data presentation control is unsuitable when working with large amounts of data, as its underlying data source control retrieves all records, even though only a subset of data is displayed. In such circumstances, we must turn to custom paging.

Introduction

As we discussed in the preceding tutorial, paging can be implemented in one of two ways:

- **Default Paging** can be implemented by simply checking the Enable Paging option in the data Web control s smart tag; however, whenever viewing a page of data, the ObjectDataSource retrieves *all* of the records, even though only a subset of them are displayed in the page
- **Custom Paging** improves the performance of default paging by retrieving only those records from the database that need to be displayed for the particular page of data requested by the user; however, custom paging involves a bit more effort to implement than default paging

Due to the ease of implementation just check a checkbox and you re done! default paging is an attractive option. Its naive approach in retrieving all of the records, though, makes it an implausible choice when paging through sufficiently large amounts of data or for sites with many concurrent users. In such circumstances, we must turn to custom paging in order to provide a responsive system.

The challenge of custom paging is being able to write a query that returns the precise set of records needed for a particular page of data. Fortunately, Microsoft SQL Server 2005 provides a new keyword for ranking results, which enables us to write a query that can efficiently retrieve the proper subset of records. In this tutorial we'll see how to use this new SQL Server 2005 keyword to implement custom paging in a GridView control. While the user interface for custom paging is identical to that for default paging, stepping from one page to the next using custom paging can be several orders of magnitude faster than default paging.

> **NOTE**
> The exact performance gain exhibited by custom paging depends on the total number of records being paged through and the load being placed on the database server. At the end of this tutorial we'll look at some rough metrics that showcase the benefits in performance obtained through custom paging.

Step 1: Understanding the Custom Paging Process

When paging through data, the precise records displayed in a page depend upon the page of data being requested and the number of records displayed per page. For example, imagine that we wanted to page through the 81 products, displaying 10 products per page. When viewing the first page, we d want products 1 through 10; when viewing the second page we d be interested in products 11 through 20, and so on.

There are three variables that dictate what records need to be retrieved and how the paging interface should be rendered:

- **Start Row Index** the index of the first row in the page of data to display; this index can be calculated by multiplying the page index by the records to display per page and adding one. For example, when paging through records 10 at a time, for the first page (whose page index is 0), the Start Row Index is 0 * 10 + 1, or 1; for the second page (whose page index is 1), the Start Row Index is 1 * 10 + 1, or 11.
- **Maximum Rows** the maximum number of records to display per page. This variable is referred to as maximum rows since for the last page there may be fewer records returned than the page size. For example, when paging through the 81 products 10 records per page, the ninth and final page will have just one record. No page, though, will show more records than the Maximum Rows value.
- **Total Record Count** the total number of records being paged through. While this variable isn t needed to determine what records to retrieve for a given page, it does dictate the paging interface. For example, if there are 81 products being paged through, the paging interface knows to display nine page numbers in the paging UI.

With default paging, the Start Row Index is computed as the product of the page index and the page size plus one, whereas the Maximum Rows is simply the page size. Since default paging retrieves all of the records from the database when rendering any page of data, the index for each row is known, thereby making moving to Start Row Index row a trivial task. Moreover, the Total Record Count is readily available, as it s simply the number of records in the DataTable (or whatever object is being used to hold the database results).

Given the Start Row Index and Maximum Rows variables, a custom paging implementation must only return the precise subset of records starting at the Start Row Index and up to Maximum Rows number of records after that. Custom paging provides two challenges:

- We must be able to efficiently associate a row index with each row in the entire data being paged through so that we can start returning records at the specified Start Row Index
- We need to provide the total number of records being paged through

In the next two steps we'll examine the SQL script needed to respond to these two challenges. In addition to the SQL script, we'll also need to implement methods in the DAL and BLL.

Step 2: Returning the Total Number of Records Being Paged Through

Before we examine how to retrieve the precise subset of records for the page being displayed, let s first look at how to return the total number of records being paged through. This information is needed in order to properly configure the paging user interface. The total number of records returned by a particular SQL query can be obtained by using the `COUNT` aggregate function. For example, to determine the total number of records in the `Products` table, we can use the following query:

```
SELECT COUNT(*)
FROM Products
```

Let s add a method to our DAL that returns this information. In particular, we'll create a DAL method called `TotalNumberOfProducts()` that executes the `SELECT` statement shown above.

Start by opening the `Northwind.xsd` Typed DataSet file in the `App_Code/DAL` folder. Next, right-click on the `ProductsTableAdapter` in the Designer and choose Add Query. As we ve seen in previous tutorials, this will allow us to add a new method to the DAL that, when invoked, will execute a particular SQL statement or stored procedure. As with our TableAdapter methods in previous tutorials, for this one opt to use an ad-hoc SQL statement.

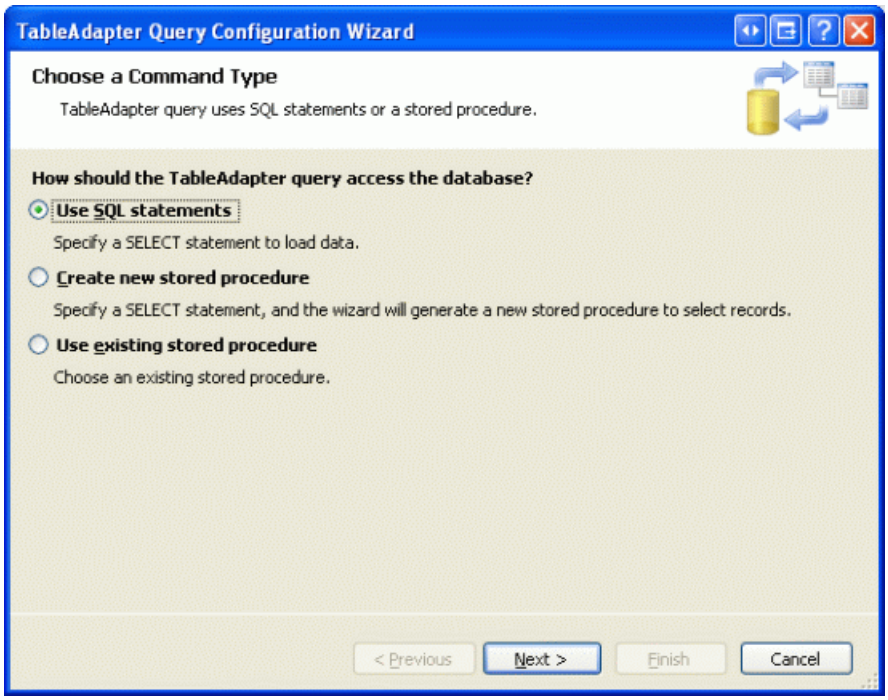


Figure 1: Use an Ad-Hoc SQL Statement

On the next screen we can specify what type of query to create. Since this query will return a single, scalar value the total number of records in the `Products` table choose the `SELECT` which returns a singe value option.

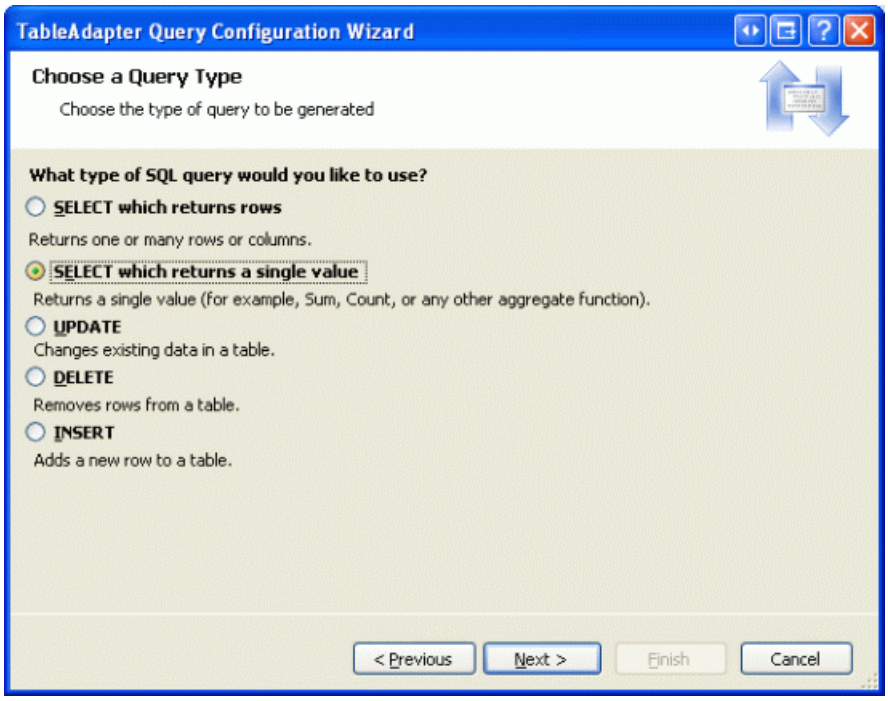


Figure 2: Configure the Query to Use a SELECT Statement that Returns a Single Value

After indicating the type of query to use, we must next specify the query.

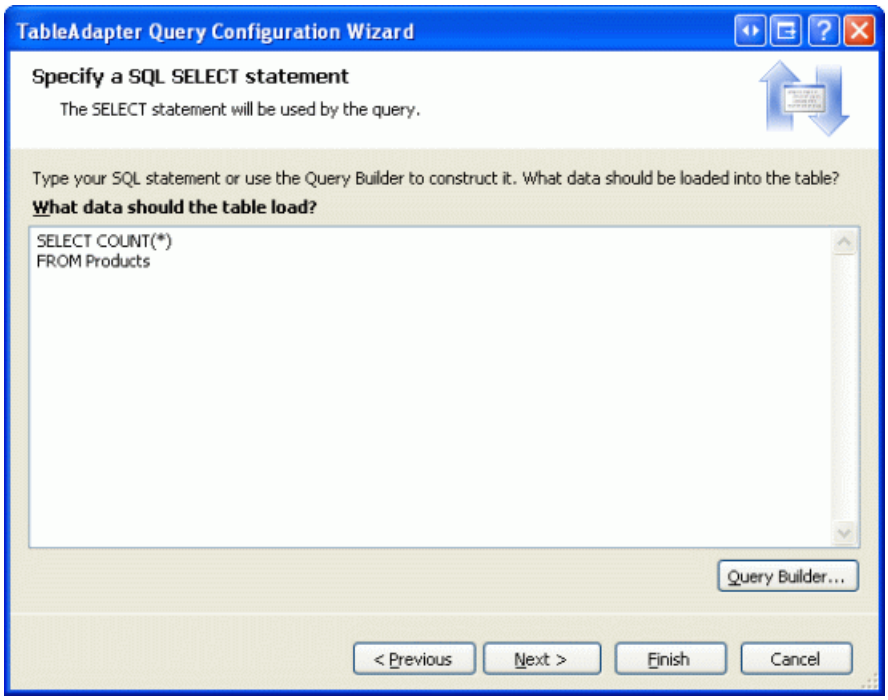


Figure 3: Use the SELECT COUNT(*) FROM Products Query

Finally, specify the name for the method. As aforementioned, let s use `TotalNumberOfProducts`.

Figure 4: Name the DAL Method TotalNumberOfProducts

After clicking Finish, the wizard will add the `TotalNumberOfProducts` method to the DAL. The scalar returning methods in the DAL return nullable types, in case the result from the SQL query is `NULL`. Our `COUNT` query, however, will always return a non-`NULL` value; regardless, the DAL method returns a nullable integer.

In addition to the DAL method, we also need a method in the BLL. Open the `ProductsBLL` class file and add a `TotalNumberOfProducts` method that simply calls down to the DAL s `TotalNumberOfProducts` method:

```
Public Function TotalNumberOfProducts() As Integer
    Return Adapter.TotalNumberOfProducts().GetValueOrDefault()
End Function
```

The DAL s `TotalNumberOfProducts` method returns a nullable integer; however, we ve created the `ProductsBLL` class s `TotalNumberOfProducts` method so that it returns a standard integer. Therefore, we need to have the `ProductsBLL` class s `TotalNumberOfProducts` method return the value portion of the nullable integer returned by the DAL s `TotalNumberOfProducts` method. The call to `GetValueOrDefault()` returns the value of the nullable integer, if it exists; if the nullable integer is `null`, however, it returns the default integer value, 0.

Step 3: Returning the Precise Subset of Records

Our next task is to create methods in the DAL and BLL that accept the Start Row Index and Maximum Rows variables discussed earlier and return the appropriate records. Before we do that, let s first look at the needed SQL script. The challenge facing us is that we must be able to efficiently assign an index to each row in the entire results being paged through so that we can return just those records starting at the Start Row Index (and up to the Maximum Records number of records).

This is not a challenge if there is already a column in the database table that serves as a row index. At first glance we might think that the `Products` table s `ProductID` field would suffice, as the first product has `ProductID` of 1, the second a 2, and so on. However, deleting a product leaves a gap in the sequence, nullifying this approach.

There are two general techniques used to efficiently associate a row index with the data to page through, thereby enabling the precise subset of records to be retrieved:

- **Using SQL Server 2005 s `ROW_NUMBER()` Keyword** new to SQL Server 2005, the `ROW_NUMBER()` keyword associates a ranking with each returned record based on some ordering. This ranking can be used as a row index for each row.
- **Using a Table Variable and `SET ROWCOUNT`** SQL Server s `SET ROWCOUNT` statement can be used to specify how many total records a query should process before terminating; table variables are local T-SQL variables that can hold tabular data, akin to temporary tables. This approach works equally well with both Microsoft SQL Server 2005 and SQL Server 2000 (whereas the `ROW_NUMBER()` approach only works with SQL Server 2005).

  The idea here is to create a table variable that has an `IDENTITY` column and columns for the primary keys of the table whose data is being paged through. Next, the contents of the table whose data is being paged through is dumped into the table variable, thereby associating a sequential row index (via the `IDENTITY` column) for each record in the table. Once the table variable has been populated, a `SELECT` statement on the table variable, joined with the underlying table, can be executed to pull out the particular records. The `SET ROWCOUNT` statement is used to intelligently limit the number of records that need to be dumped into the table variable.

  This approach s efficiency is based on the page number being requested, as the `SET ROWCOUNT` value is assigned the value of Start Row Index plus the Maximum Rows. When paging through low-numbered pages such as the first few pages of data this approach is very efficient. However, it exhibits default paging-like performance when retrieving a page near the end.

This tutorial implements custom paging using the `ROW_NUMBER()` keyword. For more information on using the table variable and `SET ROWCOUNT` technique, see A More Efficient Method for Paging Through Large Result Sets.

The `ROW_NUMBER()` keyword associated a ranking with each record returned over a particular ordering using the following syntax:

```
SELECT columnList,
       ROW_NUMBER() OVER(orderByClause)
FROM TableName
```

`ROW_NUMBER()` returns a numerical value that specifies the rank for each record with regards to the indicated

ordering. For example, to see the rank for each product, ordered from the most expensive to the least, we could use the following query:

```
SELECT ProductName, UnitPrice,
       ROW_NUMBER() OVER(ORDER BY UnitPrice DESC) AS PriceRank
FROM Products
```

Figure 5 shows this query s results when run through the query window in Visual Studio. Note that the products are ordered by price, along with a price rank for each row.

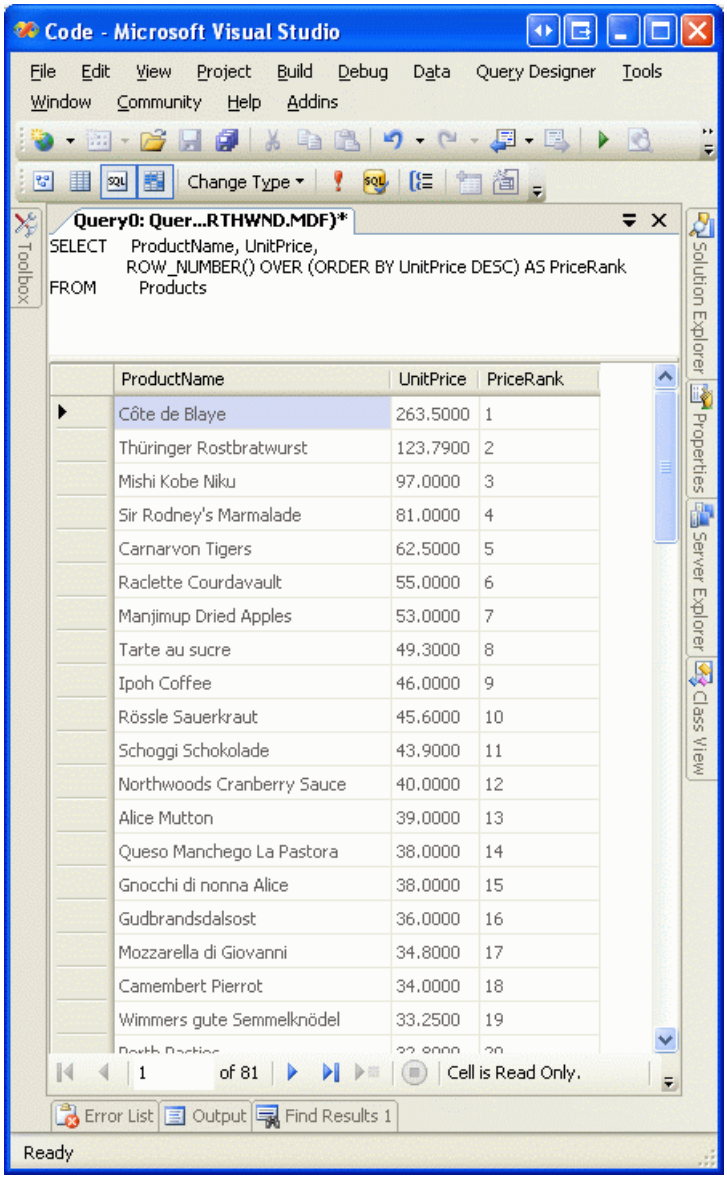


Figure 5: The Price Rank is Included for Each Returned Record

> **NOTE**
>
> `ROW_NUMBER()` is just one of the many new ranking functions available in SQL Server 2005. For a more thorough discussion of `ROW_NUMBER()`, along with the other ranking functions, read Returning Ranked Results with Microsoft SQL Server 2005.

When ranking the results by the specified `ORDER BY` column in the `OVER` clause (`UnitPrice`, in the above example), SQL Server must sort the results. This is a quick operation if there is a clustered index over the column(s) the results are being ordered by, or if there is a covering index, but can be more costly otherwise. To help improve performance for sufficiently large queries, consider adding a non-clustered index for the column by

which the results are ordered by. See Ranking Functions and Performance in SQL Server 2005 for a more detailed look at the performance considerations.

The ranking information returned by `ROW_NUMBER()` cannot directly be used in the `WHERE` clause. However, a derived table can be used to return the `ROW_NUMBER()` result, which can then appear in the `WHERE` clause. For example, the following query uses a derived table to return the ProductName and UnitPrice columns, along with the `ROW_NUMBER()` result, and then uses a `WHERE` clause to only return those products whose price rank is between 11 and 20:

```
SELECT PriceRank, ProductName, UnitPrice
FROM
   (SELECT ProductName, UnitPrice,
       ROW_NUMBER() OVER(ORDER BY UnitPrice DESC) AS PriceRank
    FROM Products
   ) AS ProductsWithRowNumber
WHERE PriceRank BETWEEN 11 AND 20
```

Extending this concept a bit further, we can utilize this approach to retrieve a specific page of data given the desired Start Row Index and Maximum Rows values:

```
SELECT PriceRank, ProductName, UnitPrice
FROM
   (SELECT ProductName, UnitPrice,
       ROW_NUMBER() OVER(ORDER BY UnitPrice DESC) AS PriceRank
    FROM Products
   ) AS ProductsWithRowNumber
WHERE PriceRank > <i>StartRowIndex</i> AND
    PriceRank <= (<i>StartRowIndex</i> + <i>MaximumRows</i>)
```

NOTE

As we will see later on in this tutorial, the `StartRowIndex` supplied by the ObjectDataSource is indexed starting at zero, whereas the `ROW_NUMBER()` value returned by SQL Server 2005 is indexed starting at 1. Therefore, the `WHERE` clause returns those records where `PriceRank` is strictly greater than `StartRowIndex` and less than or equal to `StartRowIndex` + `MaximumRows`.

Now that we ve discussed how `ROW_NUMBER()` can be used to retrieve a particular page of data given the Start Row Index and Maximum Rows values, we now need to implement this logic as methods in the DAL and BLL.

When creating this query we must decide the ordering by which the results will be ranked; let s sort the products by their name in alphabetical order. This means that with the custom paging implementation in this tutorial we will not be able to create a custom paged report than can also be sorted. In the next tutorial, though, we'll see how such functionality can be provided.

In the previous section we created the DAL method as an ad-hoc SQL statement. Unfortunately, the T-SQL parser in Visual Studio used by the TableAdapter wizard doesn t like the `OVER` syntax used by the `ROW_NUMBER()` function. Therefore, we must create this DAL method as a stored procedure. Select the Server Explorer from the View menu (or hit Ctrl+Alt+S) and expand the `NORTHWND.MDF` node. To add a new stored procedure, right-click on the Stored Procedures node and choose Add a New Stored Procedure (see Figure 6).

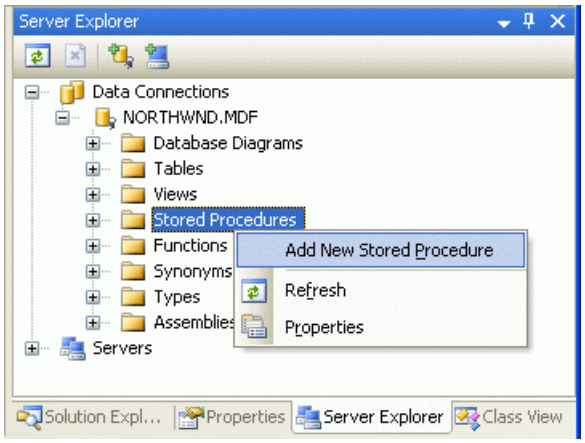


Figure 6: Add a New Stored Procedure for Paging Through the Products

This stored procedure should accept two integer input parameters - `@startRowIndex` and `@maximumRows` and use the `ROW_NUMBER()` function ordered by the `ProductName` field, returning only those rows greater than the specified `@startRowIndex` and less than or equal to `@startRowIndex` + `@maximumRow` s. Enter the following script into the new stored procedure and then click the Save icon to add the stored procedure to the database.

```
CREATE PROCEDURE dbo.GetProductsPaged
(
    @startRowIndex int,
    @maximumRows int
)
AS
    SELECT     ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit,
               UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued,
               CategoryName, SupplierName
FROM
   (
       SELECT ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit,
              UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued,
              (SELECT CategoryName
               FROM Categories
               WHERE Categories.CategoryID = Products.CategoryID) AS CategoryName,
              (SELECT CompanyName
               FROM Suppliers
               WHERE Suppliers.SupplierID = Products.SupplierID) AS SupplierName,
              ROW_NUMBER() OVER (ORDER BY ProductName) AS RowRank
        FROM Products
    ) AS ProductsWithRowNumbers
WHERE RowRank > @startRowIndex AND RowRank <= (@startRowIndex + @maximumRows)
```

After creating the stored procedure, take a moment to test it out. Right-click on the `GetProductsPaged` stored procedure name in the Server Explorer and choose the Execute option. Visual Studio will then prompt you for the input parameters, `@startRowIndex` and `@maximumRow` s (see Figure 7). Try different values and examine the results.

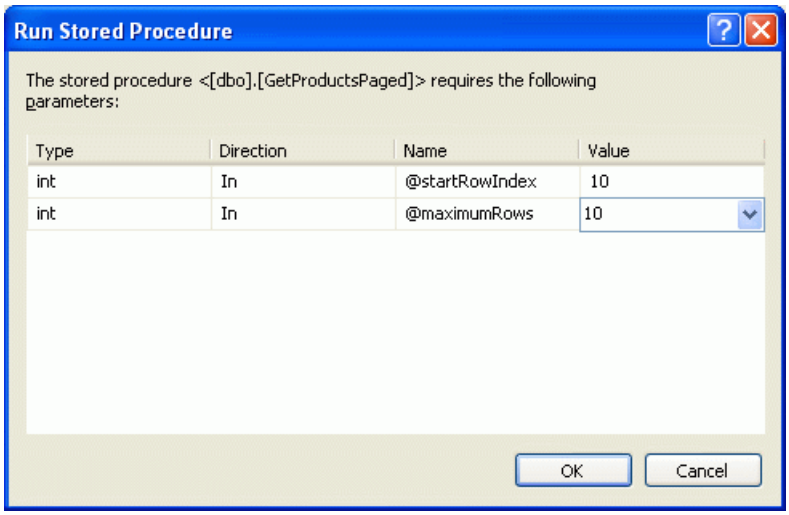


Figure 7: Enter a Value for the @startRowIndex and @maximumRows Parameters

After choosing these input parameters values, the Output window will show the results. Figure 8 shows the results when passing in 10 for both the `@startRowIndex` and `@maximumRows` parameters.

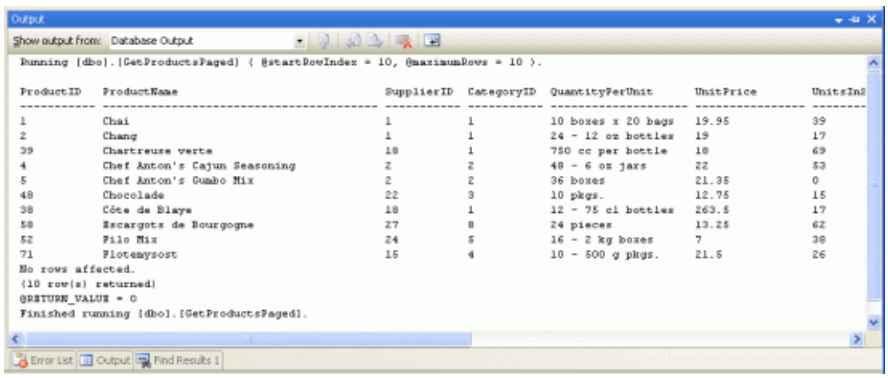


Figure 8: The Records That Would Appear in the Second Page of Data are Returned (Click to view full-size image)

With this stored procedure created, we re ready to create the `ProductsTableAdapter` method. Open the `Northwind.xsd` Typed DataSet, right-click in the `ProductsTableAdapter`, and choose the Add Query option. Instead of creating the query using an ad-hoc SQL statement, create it using an existing stored procedure.

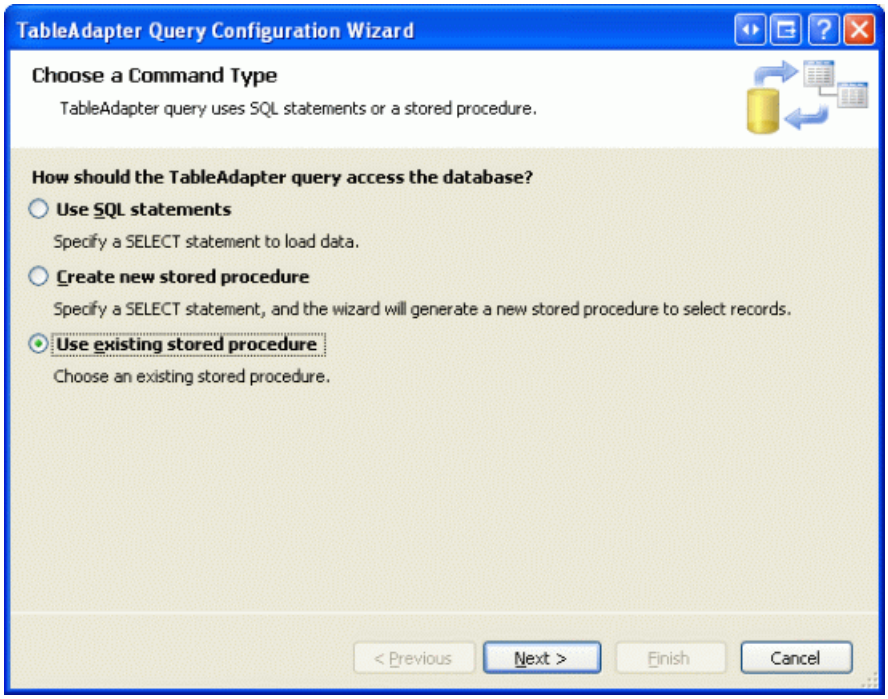


Figure 9: Create the DAL Method Using an Existing Stored Procedure

Next, we are prompted to select the stored procedure to invoke. Pick the `GetProductsPaged` stored procedure from the drop-down list.

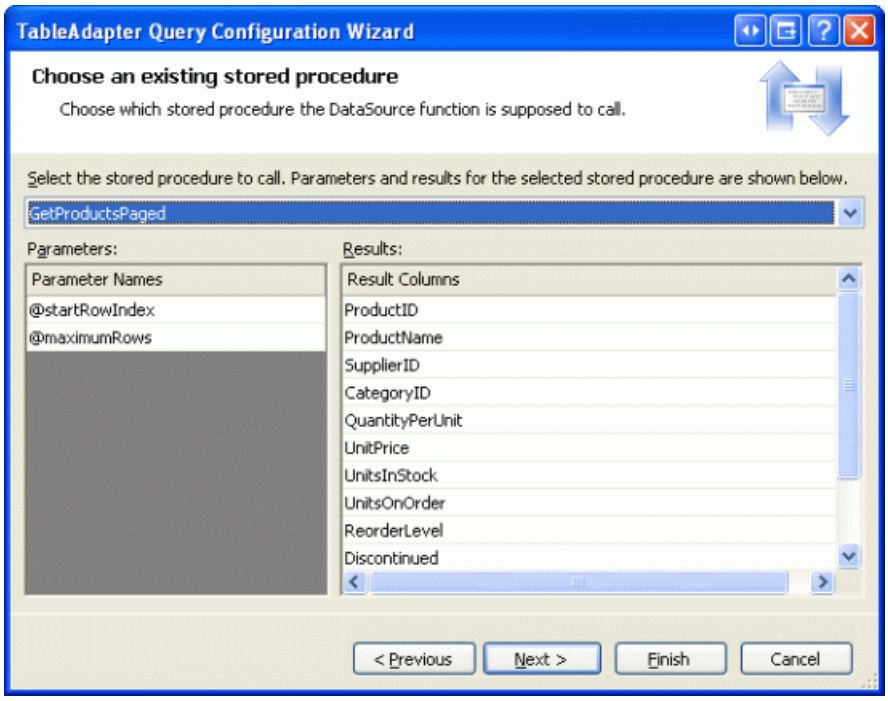


Figure 10: Choose the GetProductsPaged Stored Procedure from the Drop-Down List

The next screen then asks you what kind of data is returned by the stored procedure: tabular data, a single value, or no value. Since the `GetProductsPaged` stored procedure can return multiple records, indicate that it returns tabular data.

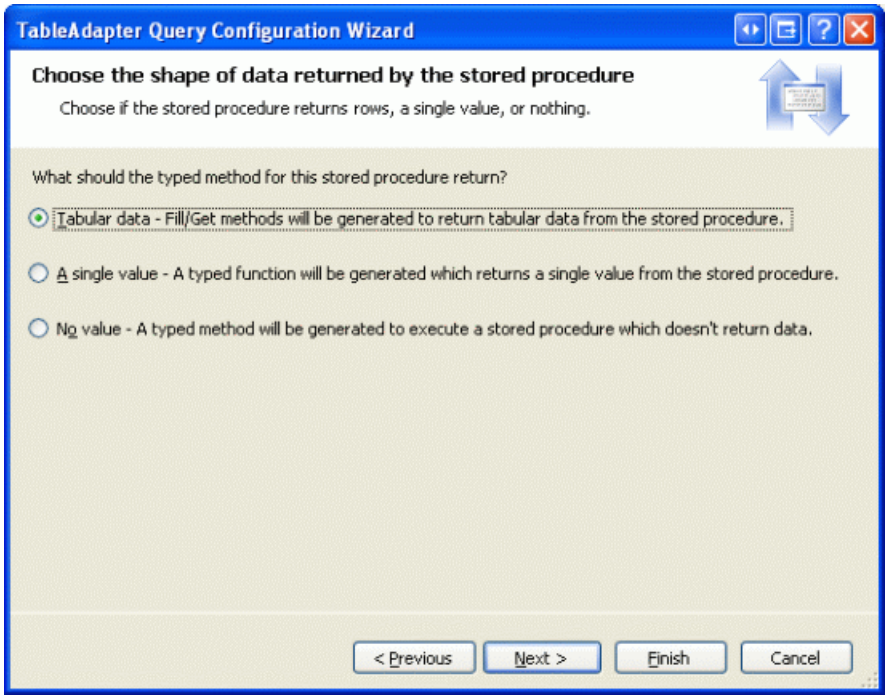


Figure 11: Indicate that the Stored Procedure Returns Tabular Data

Finally, indicate the names of the methods you want to have created. As with our previous tutorials, go ahead and create methods using both the Fill a DataTable and Return a DataTable. Name the first method `FillPaged` and the second `GetProductsPaged`.

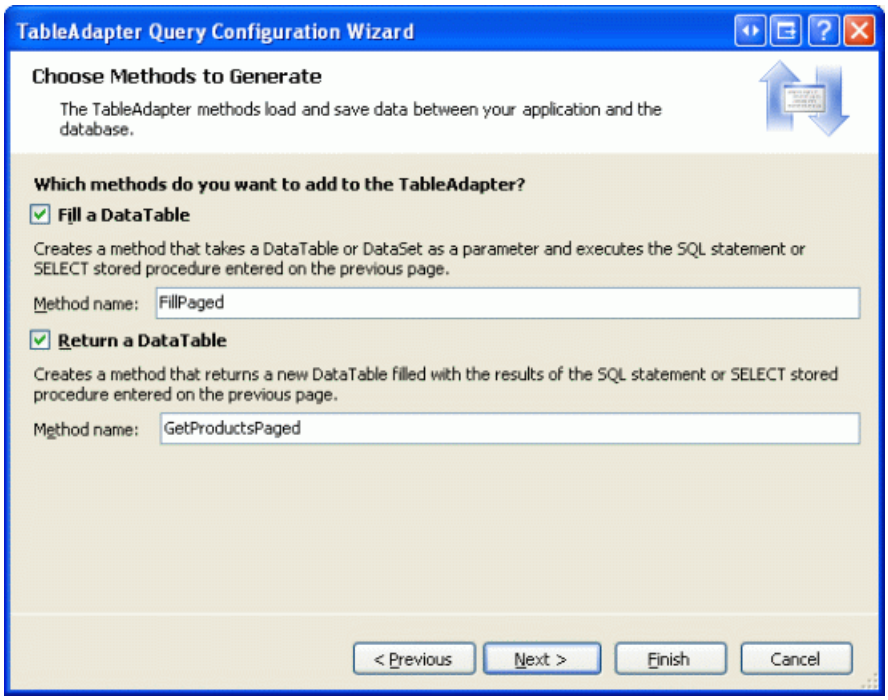


Figure 12: Name the Methods FillPaged and GetProductsPaged

In addition to created a DAL method to return a particular page of products, we also need to provide such functionality in the BLL. Like the DAL method, the BLL s GetProductsPaged method must accept two integer inputs for specifying the Start Row Index and Maximum Rows, and must return just those records that fall within the specified range. Create such a BLL method in the ProductsBLL class that merely calls down into the DAL s GetProductsPaged method, like so:

```
<System.ComponentModel.DataObjectMethodAttribute( _
    System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetProductsPaged(startRowIndex As Integer, maximumRows As Integer) _
    As Northwind.ProductsDataTable
    Return Adapter.GetProductsPaged(startRowIndex, maximumRows)
End Function
```

You can use any name for the BLL method s input parameters, but, as we will see shortly, choosing to use `startRowIndex` and `maximumRows` saves us from an extra bit of work when configuring an ObjectDataSource to use this method.

Step 4: Configuring the ObjectDataSource to Use Custom Paging

With the BLL and DAL methods for accessing a particular subset of records complete, we re ready to create a GridView control that pages through its underlying records using custom paging. Start by opening the `EfficientPaging.aspx` page in the `PagingAndSorting` folder, add a GridView to the page, and configure it to use a new ObjectDataSource control. In our past tutorials, we often had the ObjectDataSource configured to use the `ProductsBLL` class s `GetProducts` method. This time, however, we want to use the `GetProductsPaged` method instead, since the `GetProducts` method returns *all* of the products in the database whereas `GetProductsPaged` returns just a particular subset of records.

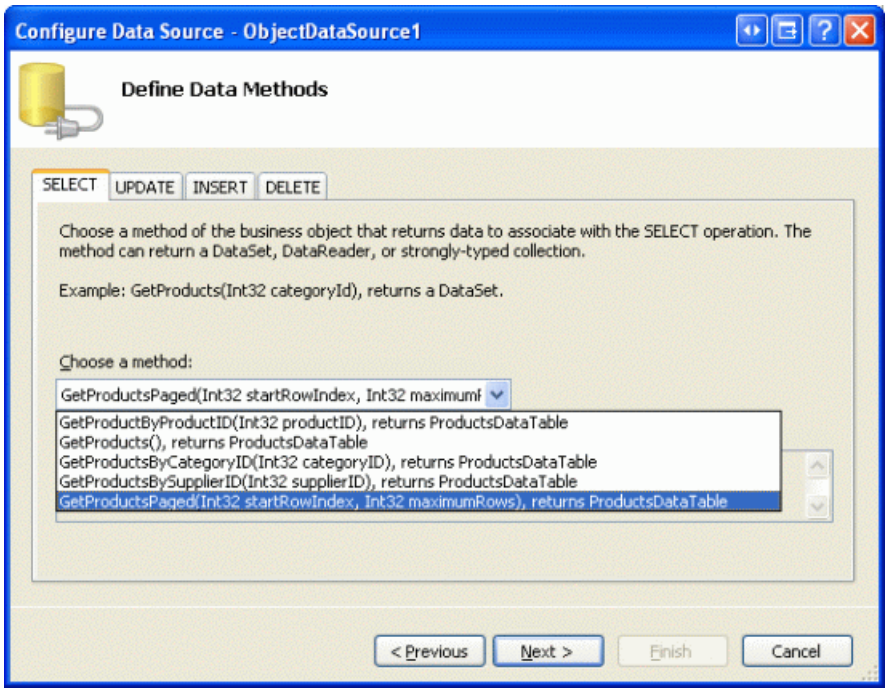


Figure 13: Configure the ObjectDataSource to Use the ProductsBLL Class s GetProductsPaged Method

Since we re creating a read-only GridView, take a moment to set the method drop-down list in the INSERT, UPDATE, and DELETE tabs to (None).

Next, the ObjectDataSource wizard prompts us for the sources of the `GetProductsPaged` method s `startRowIndex` and `maximumRows` input parameters values. These input parameters will actually be set by the GridView automatically, so simply leave the source set to None and click Finish.

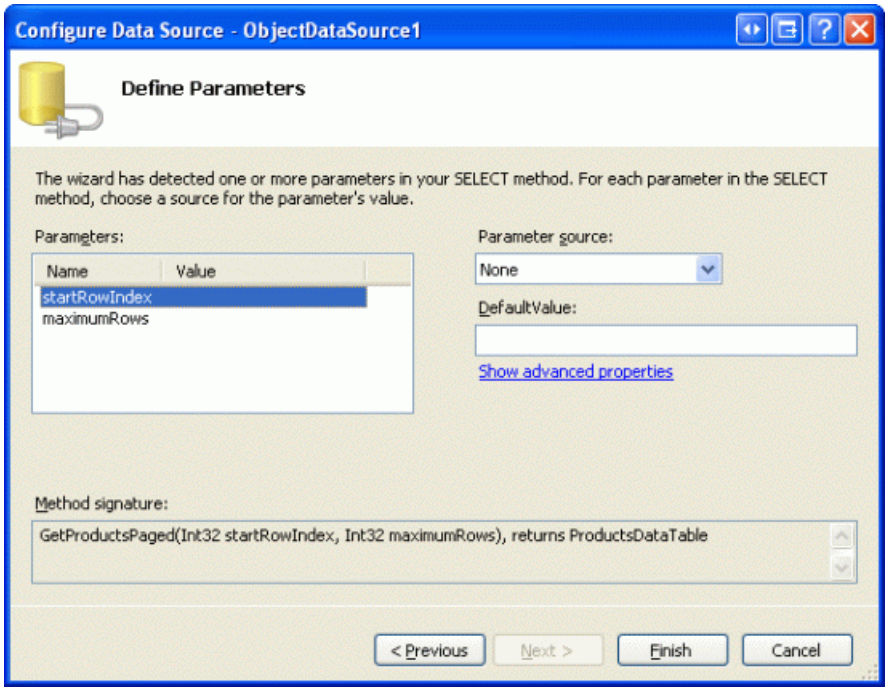


Figure 14: Leave the Input Parameter Sources as None

After completing the ObjectDataSource wizard, the GridView will contain a BoundField or CheckBoxField for each of the product data fields. Feel free to tailor the GridView s appearance as you see fit. I ve opted to display only the `ProductName`, `CategoryName`, `SupplierName`, `QuantityPerUnit`, and `UnitPrice` BoundFields. Also, configure the GridView to support paging by checking the Enable Paging checkbox in its smart tag. After these changes, the GridView and ObjectDataSource declarative markup should look similar to the following:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ObjectDataSource1" AllowPaging="True">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
            SortExpression="SupplierName" />
        <asp:BoundField DataField="QuantityPerUnit" HeaderText="Qty/Unit"
            SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
            HeaderText="Price" HtmlEncode="False" SortExpression="UnitPrice" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}" SelectMethod="GetProductsPaged"
    TypeName="ProductsBLL">
    <SelectParameters>
        <asp:Parameter Name="startRowIndex" Type="Int32" />
        <asp:Parameter Name="maximumRows" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

If you visit the page through a browser, however, the GridView is no where to be found.

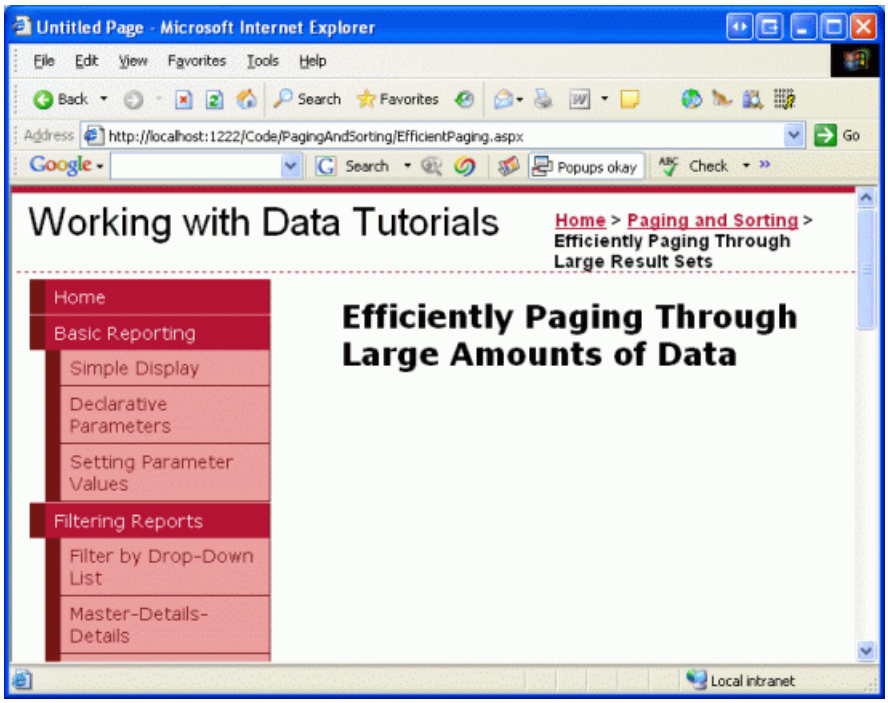


Figure 15: The GridView is Not Displayed

The GridView is missing because the ObjectDataSource is currently using 0 as the values for both of the `GetProductsPaged` `startRowIndex` and `maximumRows` input parameters. Hence, the resulting SQL query is returning no records and therefore the GridView is not displayed.

To remedy this, we need to configure the ObjectDataSource to use custom paging. This can be accomplished in the following steps:

1. **Set the ObjectDataSource s** `EnablePaging` **property to** `true` this indicates to the ObjectDataSource that it must pass to the `SelectMethod` two additional parameters: one to specify the Start Row Index (`StartRowIndexParameterName`), and one to specify the Maximum Rows (`MaximumRowsParameterName`).
2. **Set the ObjectDataSource s** `StartRowIndexParameterName` **and** `MaximumRowsParameterName` **Properties Accordingly** the `StartRowIndexParameterName` and `MaximumRowsParameterName` properties indicate the names of

the input parameters passed into the `SelectMethod` for custom paging purposes. By default, these parameter names are `startIndexRow` and `maximumRows`, which is why, when creating the `GetProductsPaged` method in the BLL, I used these values for the input parameters. If you chose to use different parameter names for the BLL s `GetProductsPaged` method such as `startIndex` and `maxRows`, for example you would need to set the ObjectDataSource s `StartRowIndexParameterName` and `MaximumRowsParameterName` properties accordingly (such as startIndex for `StartRowIndexParameterName` and maxRows for `MaximumRowsParameterName`).

3. **Set the ObjectDataSource s `SelectCountMethod` Property to the Name of the Method that Returns the Total Number of Records Being Paged Through (`TotalNumberOfProducts`)** recall that the `ProductsBLL` class s `TotalNumberOfProducts` method returns the total number of records being paged through using a DAL method that executes a `SELECT COUNT(*) FROM Products` query. This information is needed by the ObjectDataSource in order to correctly render the paging interface.
4. **Remove the `startRowIndex` and `maximumRows` `<asp:Parameter>` Elements from the ObjectDataSource s Declarative Markup** when configuring the ObjectDataSource through the wizard, Visual Studio automatically added two `<asp:Parameter>` elements for the `GetProductsPaged` method s input parameters. By setting `EnablePaging` to `true`, these parameters will be passed automatically; if they also appear in the declarative syntax, the ObjectDataSource will attempt to pass *four* parameters to the `GetProductsPaged` method and two parameters to the `TotalNumberOfProducts` method. If you forget to remove these `<asp:Parameter>` elements, when visiting the page through a browser you'll get an error message like: *ObjectDataSource 'ObjectDataSource1' could not find a non-generic method 'TotalNumberOfProducts' that has parameters: startRowIndex, maximumRows*.

After making these changes, the ObjectDataSource s declarative syntax should look like the following:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}" SelectMethod="GetProductsPaged"
    TypeName="ProductsBLL" EnablePaging="True" SelectCountMethod="
    TotalNumberOfProducts">
</asp:ObjectDataSource>
```

Note that the `EnablePaging` and `SelectCountMethod` properties have been set and the `<asp:Parameter>` elements have been removed. Figure 16 shows a screen shot of the Properties window after these changes have been made.

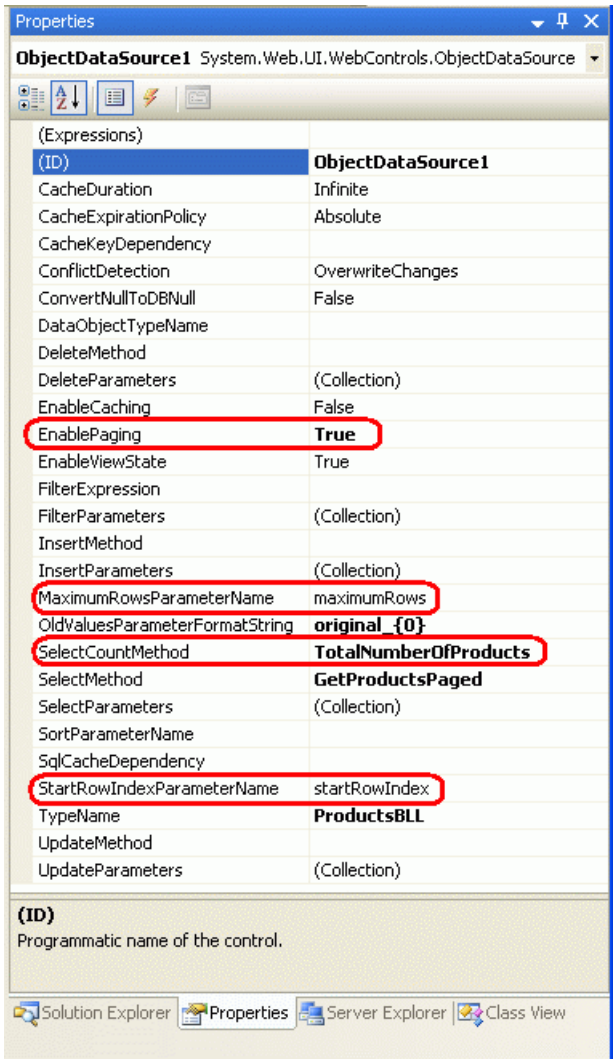


Figure 16: To Use Custom Paging, Configure the ObjectDataSource Control

After making these changes, visit this page through a browser. You should see 10 products listed, ordered alphabetically. Take a moment to step through the data one page at a time. While there is no visual difference from the end user s perspective between default paging and custom paging, custom paging more efficiently pages through large amounts of data as it only retrieves those records that need to be displayed for a given page.

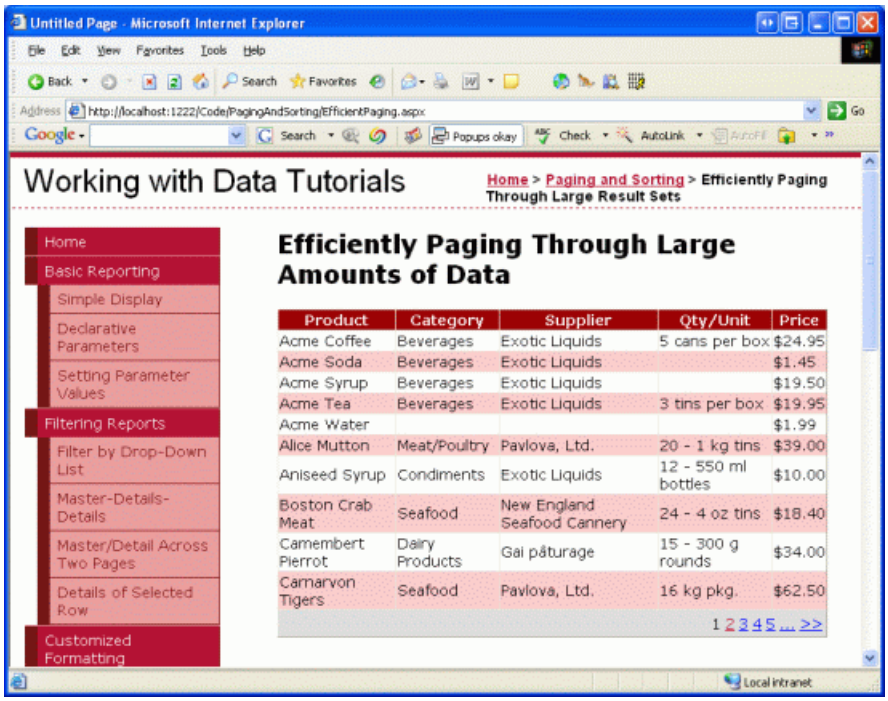


Figure 17: The Data, Ordered by the Product s Name, is Paged Using Custom Paging (Click to view full-size

image)

> **NOTE**
>
> With custom paging, the page count value returned by the ObjectDataSource's `SelectCountMethod` is stored in the GridView's view state. Other GridView variables the `PageIndex`, `EditIndex`, `SelectedIndex`, `DataKeys` collection, and so on are stored in *control state*, which is persisted regardless of the value of the GridView's `EnableViewState` property. Since the `PageCount` value is persisted across postbacks using view state, when using a paging interface that includes a link to take you to the last page, it is imperative that the GridView's view state be enabled. (If your paging interface does not include a direct link to the last page, then you may disable view state.)

Clicking the last page link causes a postback and instructs the GridView to update its `PageIndex` property. If the last page link is clicked, the GridView assigns its `PageIndex` property to a value one less than its `PageCount` property. With view state disabled, the `PageCount` value is lost across postbacks and the `PageIndex` is assigned the maximum integer value instead. Next, the GridView attempts to determine the starting row index by multiplying the `PageSize` and `PageCount` properties. This results in an `OverflowException` since the product exceeds the maximum allowed integer size.

Implement Custom Paging and Sorting

Our current custom paging implementation requires that the order by which the data is paged through be specified statically when creating the `GetProductsPaged` stored procedure. However, you may have noted that the GridView s smart tag contains an Enable Sorting checkbox in addition to the Enable Paging option. Unfortunately, adding sorting support to the GridView with our current custom paging implementation will only sort the records on the currently viewed page of data. For example, if you configure the GridView to also support paging and then, when viewing the first page of data, sort by product name in descending order, it will reverse the order of the products on page 1. As Figure 18 shows, such shows Carnarvon Tigers as the first product when sorting in reverse alphabetical order, which ignores the 71 other products that come after Carnarvon Tigers, alphabetically; only those records on the first page are considered in the sorting.

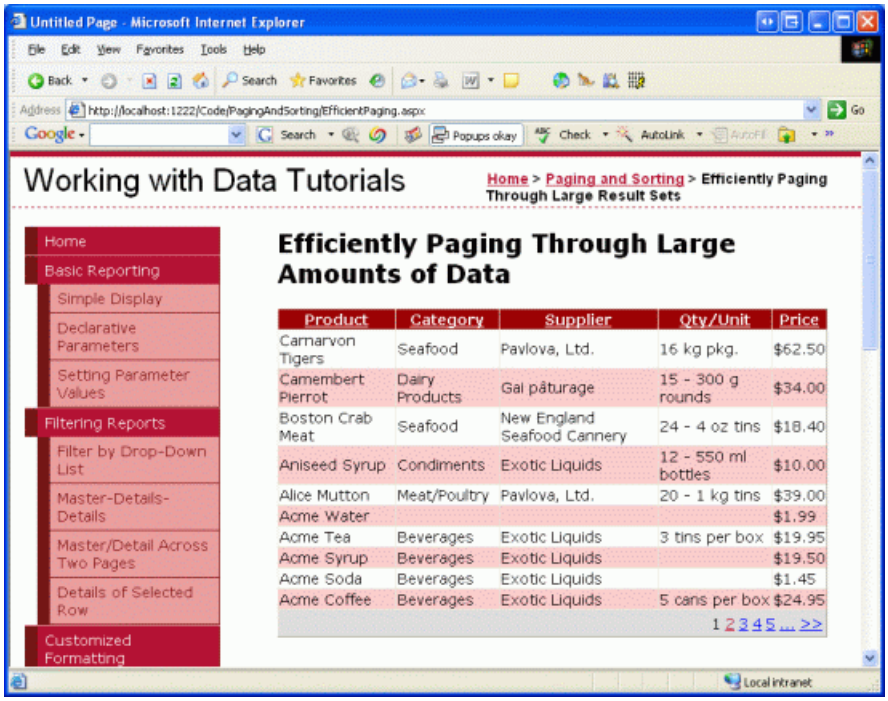


Figure 18: Only the Data Shown on the Current Page is Sorted (Click to view full-size image)

The sorting only applies to the current page of data because the sorting is occurring after the data has been retrieved from the BLL s `GetProductsPaged` method, and this method only returns those records for the specific page. To implement sorting correctly, we need to pass the sort expression to the `GetProductsPaged` method so

that the data can be ranked appropriately before returning the specific page of data. We'll see how to accomplish this in our next tutorial.

Implementing Custom Paging and Deleting

If you enabling deleting functionality in a GridView whose data is paged using custom paging techniques you will find that when deleting the last record from the last page, the GridView disappears rather than appropriately decrementing the GridView s `PageIndex`. To reproduce this bug, enable deleting for the tutorial just we just created. Go to the last page (page 9), where you should see a single product since we are paging through 81 products, 10 products at a time. Delete this product.

Upon deleting the last product, the GridView *should* automatically go to the eighth page, and such functionality is exhibited with default paging. With custom paging, however, after deleting that last product on the last page, the GridView simply disappears from the screen altogether. The precise reason *why* this happens is a bit beyond the scope of this tutorial; see Deleting the Last Record on the Last Page from a GridView with Custom Paging for the low-level details as to the source of this problem. In summary it s due to the following sequence of steps that are performed by the GridView when the Delete button is clicked:

1. Delete the record
2. Get the appropriate records to display for the specified `PageIndex` and `PageSize`
3. Check to ensure that the `PageIndex` does not exceed the number of pages of data in the data source; if it does, automatically decrement the GridView s `PageIndex` property
4. Bind the appropriate page of data to the GridView using the records obtained in Step 2

The problem stems from the fact that in Step 2 the `PageIndex` used when grabbing the records to display is still the `PageIndex` of the last page whose sole record was just deleted. Therefore, in Step 2, *no* records are returned since that last page of data no longer contains any records. Then, in Step 3, the GridView realizes that its `PageIndex` property is greater than the total number of pages in the data source (since we ve deleted the last record in the last page) and therefore decrements its `PageIndex` property. In Step 4 the GridView attempts to bind itself to the data retrieved in Step 2; however, in Step 2 no records were returned, therefore resulting in an empty GridView. With default paging, this problem doesn t surface because in Step 2 *all* records are retrieved from the data source.

To fix this we have two options. The first is to create an event handler for the GridView s `RowDeleted` event handler that determines how many records were displayed in the page that was just deleted. If there was only one record, then the record just deleted must have been the last one and we need to decrement the GridView s `PageIndex`. Of course, we only want to update the `PageIndex` if the delete operation was actually successful, which can be determined by ensuring that the `e.Exception` property is `null`.

This approach works because it updates the `PageIndex` after Step 1 but before Step 2. Therefore, in Step 2, the appropriate set of records is returned. To accomplish this, use code like the following:

```
Protected Sub GridView1_RowDeleted(sender As Object, e As GridViewDeletedEventArgs) _
    Handles GridView1.RowDeleted
    ' If we just deleted the last row in the GridView, decrement the PageIndex
    If e.Exception Is Nothing AndAlso GridView1.Rows.Count = 1 Then
        ' we just deleted the last row
        GridView1.PageIndex = Math.Max(0, GridView1.PageIndex - 1)
    End If
End Sub
```

An alternative workaround is to create an event handler for the ObjectDataSource s `RowDeleted` event and to set the `AffectedRows` property to a value of 1. After deleting the record in Step 1 (but before re-retrieving the data in Step 2), the GridView updates its `PageIndex` property if one or more rows were affected by the operation. However, the `AffectedRows` property is not set by the ObjectDataSource and therefore this step is omitted. One

way to have this step executed is to manually set the `AffectedRows` property if the delete operation completes successfully. This can be accomplished using code like the following:

```
Protected Sub ObjectDataSource1_Deleted( _
    sender As Object, e As ObjectDataSourceStatusEventArgs) _
    Handles ObjectDataSource1.Deleted
    ' If we get back a Boolean value from the DeleteProduct method and it's true, then
    ' we successfully deleted the product. Set AffectedRows to 1
    If TypeOf e.ReturnValue Is Boolean AndAlso CType(e.ReturnValue, Boolean) = True Then
        e.AffectedRows = 1
    End If
End Sub
```

The code for both of these events handlers can be found in code-behind class of the `EfficientPaging.aspx` example.

Comparing the Performance of Default and Custom Paging

Since custom paging only retrieves the needed records, whereas default paging returns *all* of the records for each page being viewed, it s clear that custom paging is more efficient than default paging. But just how much more efficient is custom paging? What sort of performance gains can be seen by moving from default paging to custom paging?

Unfortunately, there s no one size fits all answer here. The performance gain depends on a number of factors, the most prominent two being the number of records being paged through and the load placed on the database server and communication channels between the web server and database server. For small tables with just a few dozen records, the performance difference may be negligible. For large tables, with thousands to hundreds of thousands of rows, though, the performance difference is acute.

An article of mine, Custom Paging in ASP.NET 2.0 with SQL Server 2005, contains some performance tests I ran to exhibit the differences in performance between these two paging techniques when paging through a database table with 50,000 records. In these tests I examined both the time to execute the query at the SQL Server level (using SQL Profiler) and at the ASP.NET page using ASP.NET s tracing features. Keep in mind that these tests were run on my development box with a single active user, and therefore are unscientific and do not mimic typical website load patterns. Regardless, the results illustrate the relative differences in execution time for default and custom paging when working with sufficiently large amounts of data.

	AVG. DURATION (SEC)	READS
Default Paging SQL Profiler	1.411	383
Custom Paging SQL Profiler	0.002	29
Default Paging ASP.NET Trace	2.379	*N/A*
Custom Paging ASP.NET Trace	0.029	*N/A*

As you can see, retrieving a particular page of data required 354 less reads on average and completed in a fraction of the time. At the ASP.NET page, custom the page was able to render in close to 1/100th of the time it took when using default paging. See my article for more information on these results along with code and a database you can download to reproduce these tests in your own environment.

Summary

Default paging is a cinch to implement just check the Enable Paging checkbox in the data Web control s smart tag

but such simplicity comes at the cost of performance. With default paging, when a user requests any page of data *all* records are returned, even though only a tiny fraction of them may be shown. To combat this performance overhead, the ObjectDataSource offers an alternative paging option custom paging.

While custom paging improves upon default paging s performance issues by retrieving only those records that need to be displayed, it s more involved to implement custom paging. First, a query must be written that correctly (and efficiently) accesses the specific subset of records requested. This can be accomplished in a number of ways; the one we examined in this tutorial is to use SQL Server 2005 s new `ROW_NUMBER()` function to rank results, and then to return just those results whose ranking falls within a specified range. Furthermore, we need to add a means to determine the total number of records being paged through. After creating these DAL and BLL methods, we also need to configure the ObjectDataSource so that it can determine how many total records are being paged through and can correctly pass the Start Row Index and Maximum Rows values to the BLL.

While implementing custom paging does require a number of steps and is not nearly as simple as default paging, custom paging is a necessity when paging through sufficiently large amounts of data. As the results examined showed, custom paging can shed seconds off of the ASP.NET page render time and can lighten the load on the database server by one ore more orders of magnitude.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Sorting Custom Paged Data (VB)

2/20/2020 • 12 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In the previous tutorial we learned how to implement custom paging when presenting data on a web page. In this tutorial we see how to extend the preceding example to include support for sorting custom paging.

Introduction

Compared to default paging, custom paging can improve the performance of paging through data by several orders of magnitude, making custom paging the de facto paging implementation choice when paging through large amounts of data. Implementing custom paging is more involved than implementing default paging, however, especially when adding sorting to the mix. In this tutorial we'll extend the example from the preceding one to include support for sorting *and* custom paging.

> **NOTE**
>
> Since this tutorial builds upon the preceding one, before beginning take a moment to copy the declarative syntax within the `<asp:Content>` element from the preceding tutorial s web page (`EfficientPaging.aspx`) and paste it between the `<asp:Content>` element in the `SortParameter.aspx` page. Refer back to Step 1 of the Adding Validation Controls to the Editing and Inserting Interfaces tutorial for a more detailed discussion on replicating the functionality of one ASP.NET page to another.

Step 1: Reexamining the Custom Paging Technique

For custom paging to work properly, we must implement some technique that can efficiently grab a particular subset of records given the Start Row Index and Maximum Rows parameters. There are a handful of techniques that can be used to achieve this aim. In the preceding tutorial we looked at accomplishing this using Microsoft SQL Server 2005 s new `ROW_NUMBER()` ranking function. In short, the `ROW_NUMBER()` ranking function assigns a row number to each row returned by a query that is ranked by a specified sort order. The appropriate subset of records is then obtained by returning a particular section of the numbered results. The following query illustrates how to use this technique to return those products numbered 11 through 20 when ranking the results ordered alphabetically by the `ProductName`:

```
SELECT ProductID, ProductName, ...
FROM
   (SELECT ProductID, ProductName, ..., ROW_NUMBER() OVER
    (ORDER BY ProductName) AS RowRank
    FROM Products) AS ProductsWithRowNumbers
WHERE RowRank > 10 AND RowRank <= 20
```

This technique works well for paging using a specific sort order (`ProductName` sorted alphabetically, in this case), but the query needs to be modified to show the results sorted by a different sort expression. Ideally, the above query could be rewritten to use a parameter in the `OVER` clause, like so:

```
SELECT ProductID, ProductName, ...
FROM
   (SELECT ProductID, ProductName, ..., ROW_NUMBER() OVER
    (ORDER BY @sortExpression) AS RowRank
    FROM Products) AS ProductsWithRowNumbers
WHERE RowRank > 10 AND RowRank <= 20
```

Unfortunately, parameterized `ORDER BY` clauses are not allowed. Instead, we must create a stored procedure that accepts a `@sortExpression` input parameter, but uses one of the following workarounds:

- Write hard-coded queries for each of the sort expressions that may be used; then, use `IF/ELSE` T-SQL statements to determine which query to execute.
- Use a `CASE` statement to provide dynamic `ORDER BY` expressions based on the `@sortExpressio` n input parameter; see the Used to Dynamically Sort Query Results section in The Power of SQL `CASE` Statements for more information.
- Craft the appropriate query as a string in the stored procedure and then use the `sp_executesql` system stored procedure to execute the dynamic query.

Each of these workarounds has some drawbacks. The first option is not as maintainable as the other two as it requires that you create a query for each possible sort expression. Therefore, if later you decide to add new, sortable fields to the GridView you will also need to go back and update the stored procedure. The second approach has some subtleties that introduce performance concerns when sorting by non-string database columns and also suffers from the same maintainability issues as the first. And the third choice, which uses dynamic SQL, introduces the risk for a SQL injection attack if an attacker is able to execute the stored procedure passing in the input parameter values of their choosing.

While none of these approaches is perfect, I think the third option is the best of the three. With its use of dynamic SQL, it offers a level of flexibility the other two do not. Furthermore, a SQL injection attack can only be exploited if an attacker is able to execute the stored procedure passing in the input parameters of his choice. Since the DAL uses parameterized queries, ADO.NET will protect those parameters that are sent to the database through the architecture, meaning that the SQL injection attack vulnerability only exists if the attacker can directly execute the stored procedure.

To implement this functionality, create a new stored procedure in the Northwind database named `GetProductsPagedAndSorted`. This stored procedure should accept three input parameters: `@sortExpression`, an input parameter of type `nvarchar(100`) that specifies how the results should be sorted and is injected directly after the `ORDER BY` text in the `OVER` clause; and `@startRowIndex` and `@maximumRows`, the same two integer input parameters from the `GetProductsPaged` stored procedure examined in the preceding tutorial. Create the `GetProductsPagedAndSorted` stored procedure using the following script:

```
CREATE PROCEDURE dbo.GetProductsPagedAndSorted
(
    @sortExpression nvarchar(100),
    @startRowIndex int,
    @maximumRows int
)
AS
-- Make sure a @sortExpression is specified
IF LEN(@sortExpression) = 0
    SET @sortExpression = 'ProductID'
-- Issue query
DECLARE @sql nvarchar(4000)
SET @sql = 'SELECT ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit,
                   UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued,
                   CategoryName, SupplierName
            FROM (SELECT ProductID, ProductName, p.SupplierID, p.CategoryID,
                         QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
                         ReorderLevel, Discontinued,
                  c.CategoryName, s.CompanyName AS SupplierName,
                   ROW_NUMBER() OVER (ORDER BY ' + @sortExpression + ') AS RowRank
            FROM Products AS p
                    INNER JOIN Categories AS c ON
                        c.CategoryID = p.CategoryID
                    INNER JOIN Suppliers AS s ON
                        s.SupplierID = p.SupplierID) AS ProductsWithRowNumbers
            WHERE     RowRank > ' + CONVERT(nvarchar(10), @startRowIndex) +
                ' AND RowRank <= (' + CONVERT(nvarchar(10), @startRowIndex) + ' + '
                + CONVERT(nvarchar(10), @maximumRows) + ')'
-- Execute the SQL query
EXEC sp_executesql @sql
```

The stored procedure starts by ensuring that a value for the `@sortExpression` parameter has been specified. If it is missing, the results are ranked by `ProductID`. Next, the dynamic SQL query is constructed. Note that the dynamic SQL query here differs slightly from our previous queries used to retrieve all rows from the Products table. In prior examples, we obtained each product s associated category s and supplier s names using a subquery. This decision was made back in the Creating a Data Access Layer tutorial and was done in lieu of using `JOIN` s because the TableAdapter cannot automatically create the associated insert, update, and delete methods for such queries. The `GetProductsPagedAndSorted` stored procedure, however, must use `JOIN` s for the results to be ordered by the category or supplier names.

This dynamic query is built up by concatenating the static query portions and the `@sortExpression`, `@startRowIndex`, and `@maximumRows` parameters. Since `@startRowIndex` and `@maximumRows` are integer parameters, they must be converted into nvarchars in order to be correctly concatenated. Once this dynamic SQL query has been constructed, it is executed via `sp_executesql`.

Take a moment to test this stored procedure with different values for the `@sortExpression`, `@startRowIndex`, and `@maximumRows` parameters. From the Server Explorer, right-click on the stored procedure name and choose Execute. This will bring up the Run Stored Procedure dialog box into which you can enter the input parameters (see Figure 1). To sort the results by the category name, use CategoryName for the `@sortExpression` parameter value; to sort by the supplier s company name, use CompanyName. After providing the parameters values, click OK. The results are displayed in the Output window. Figure 2 shows the results when returning products ranked 11 through 20 when ordering by the `UnitPrice` in descending order.

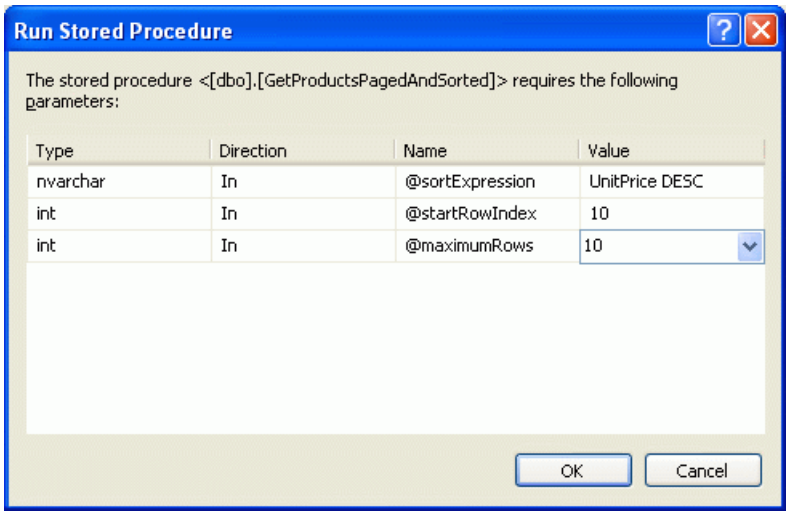


Figure 1: Try Different Values for the Stored Procedure s Three Input Parameters

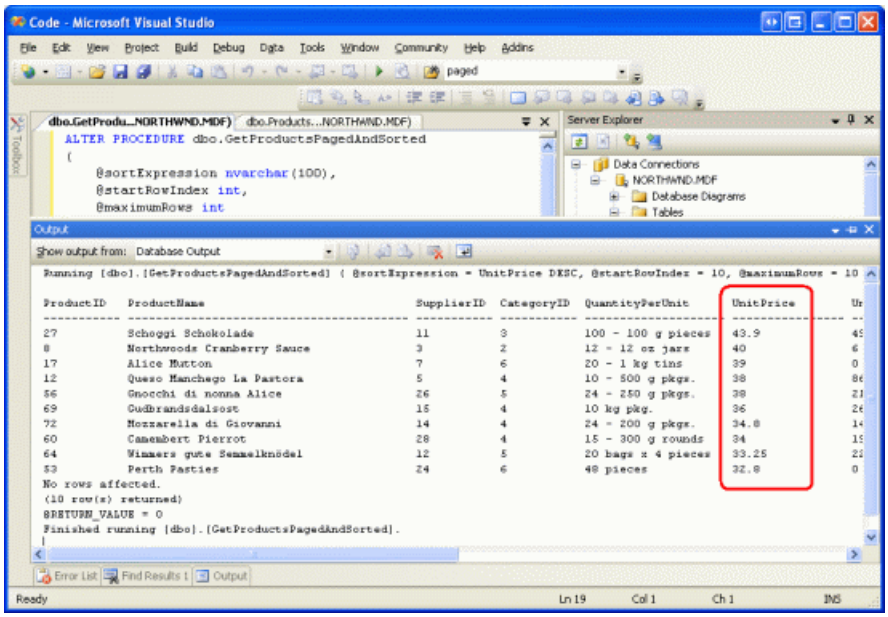


Figure 2: The Stored Procedure s Results are Shown in the Output Window (Click to view full-size image)

> **NOTE**
>
> When ranking the results by the specified `ORDER BY` column in the `OVER` clause, SQL Server must sort the results. This is a quick operation if there is a clustered index over the column(s) the results are being ordered by or if there is a covering index, but can be more costly otherwise. To improve performance for sufficiently large queries, consider adding a non-clustered index for the column by which the results are ordered by. Refer to Ranking Functions and Performance in SQL Server 2005 for more details.

Step 2: Augmenting the Data Access and Business Logic Layers

With the `GetProductsPagedAndSorted` stored procedure created, our next step is to provide a means to execute that stored procedure through our application architecture. This entails adding an appropriate method to both the DAL and BLL. Let s start by adding a method to the DAL. Open the `Northwind.xsd` Typed DataSet, right-click on the `ProductsTableAdapter`, and choose the Add Query option from the context menu. As we did in the preceding tutorial, we want to configure this new DAL method to use an existing stored procedure - `GetProductsPagedAndSorted`, in this case. Start by indicating that you want the new TableAdapter method to use an existing stored procedure.

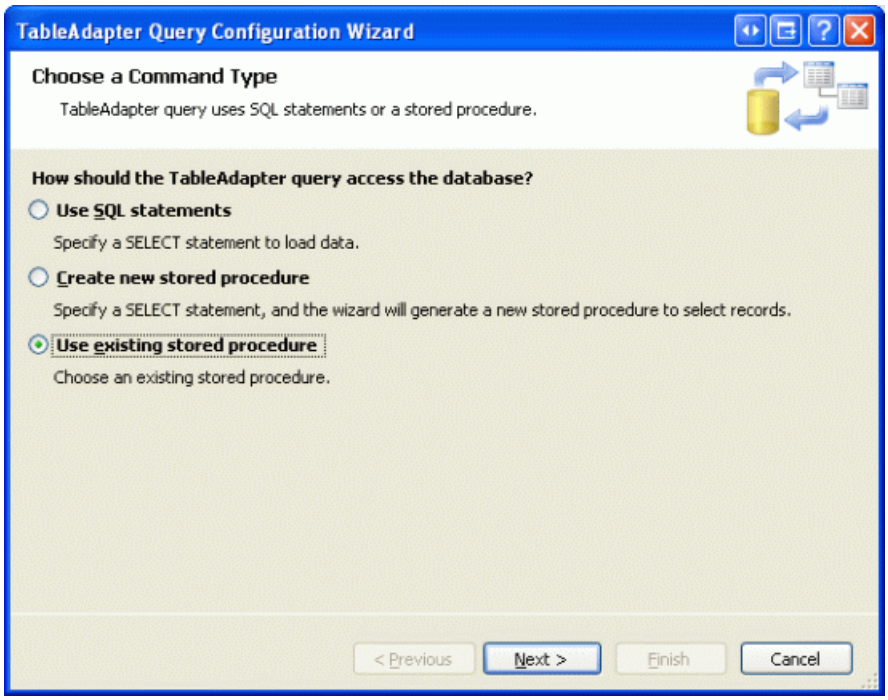


Figure 3: Choose to Use an Existing Stored Procedure

To specify the stored procedure to use, select the `GetProductsPagedAndSorted` stored procedure from the drop-down list in the next screen.

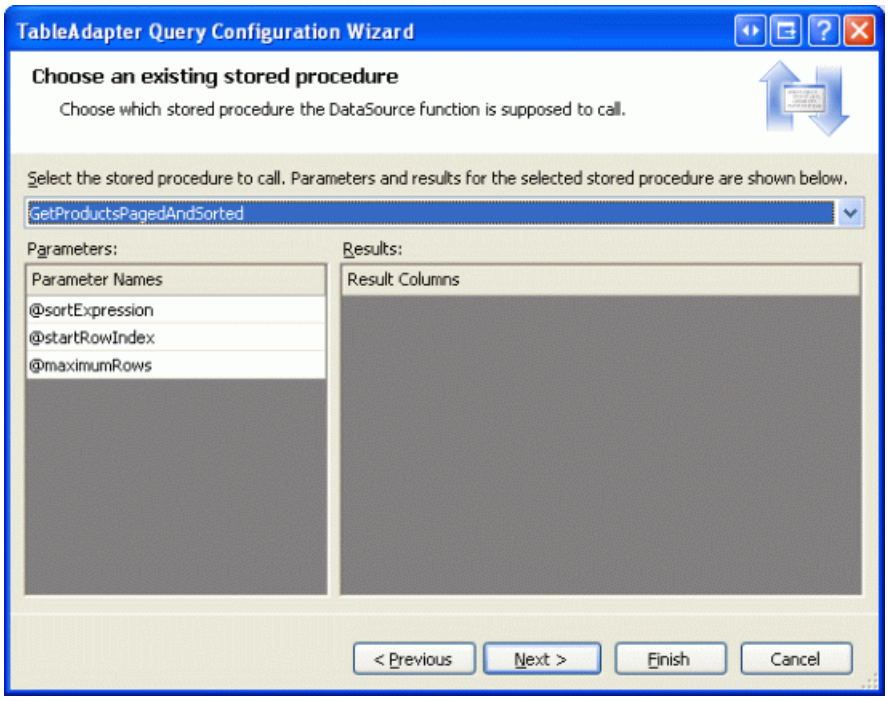


Figure 4: Use the GetProductsPagedAndSorted Stored Procedure

This stored procedure returns a set of records as its results so, in the next screen, indicate that it returns tabular data.

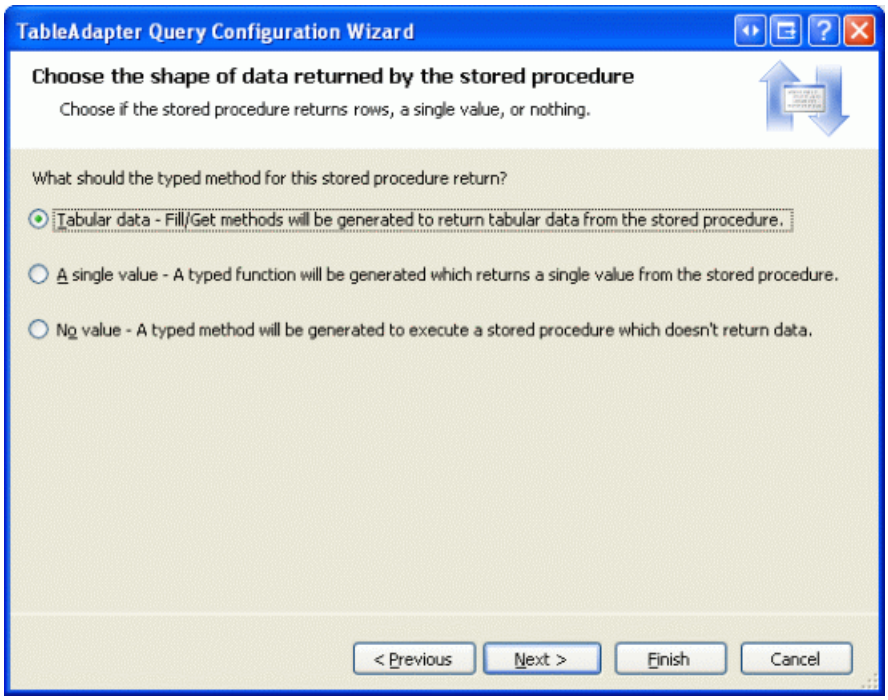


Figure 5: Indicate that the Stored Procedure Returns Tabular Data

Finally, create DAL methods that use both the Fill a DataTable and Return a DataTable patterns, naming the methods `FillPagedAndSorted` and `GetProductsPagedAndSorted`, respectively.

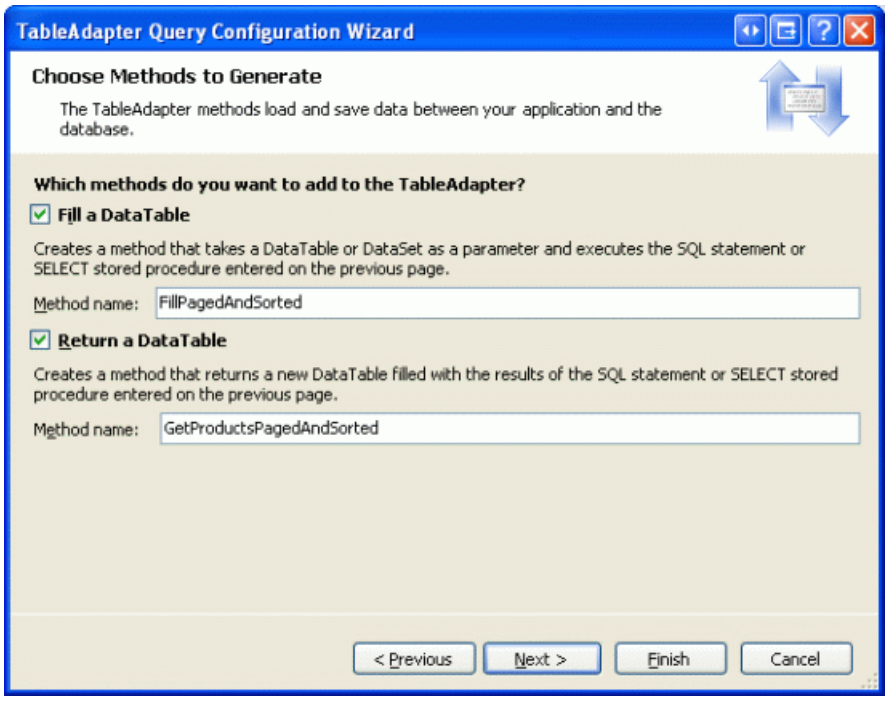


Figure 6: Choose the Methods Names

Now that we ve extended the DAL, we re ready to turn to the BLL. Open the `ProductsBLL` class file and add a new method, `GetProductsPagedAndSorted`. This method needs to accept three input parameters `sortExpression`, `startRowIndex`, and `maximumRows` and should simply call down into the DAL s `GetProductsPagedAndSorted` method, like so:

```
<System.ComponentModel.DataObjectMethodAttribute( _
    System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetProductsPagedAndSorted(ByVal sortExpression As String, _
    ByVal startRowIndex As Integer, ByVal maximumRows As Integer) _
    As Northwind.ProductsDataTable
    Return Adapter.GetProductsPagedAndSorted(sortExpression, startRowIndex, maximumRows)
End Function
```

Step 3: Configuring the ObjectDataSource to Pass in the SortExpression Parameter

Having augmented the DAL and BLL to include methods that utilize the `GetProductsPagedAndSorted` stored procedure, all that remains is to configure the ObjectDataSource in the `SortParameter.aspx` page to use the new BLL method and to pass in the `SortExpression` parameter based on the column that the user has requested to sort the results by.

Start by changing the ObjectDataSource s `SelectMethod` from `GetProductsPaged` to `GetProductsPagedAndSorted`. This can be done through the Configure Data Source wizard, from the Properties window, or directly through the declarative syntax. Next, we need to provide a value for the ObjectDataSource s `SortParameterName` property. If this property is set, the ObjectDataSource attempts to pass in the GridView s `SortExpression` property to the `SelectMethod`. In particular, the ObjectDataSource looks for an input parameter whose name is equal to the value of the `SortParameterName` property. Since the BLL s `GetProductsPagedAndSorted` method has the sort expression input parameter named `sortExpression`, set the ObjectDataSource s `SortExpression` property to sortExpression .

After making these two changes, the ObjectDataSource s declarative syntax should look similar to the following:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}" TypeName="ProductsBLL"
    SelectMethod="GetProductsPagedAndSorted" EnablePaging="True"
    SelectCountMethod="TotalNumberOfProducts" SortParameterName="sortExpression">
</asp:ObjectDataSource>
```

> **NOTE**
> As with the preceding tutorial, ensure that the ObjectDataSource does *not* include the sortExpression, startRowIndex, or maximumRows input parameters in its SelectParameters collection.

To enable sorting in the GridView, simply check the Enable Sorting checkbox in the GridView s smart tag, which sets the GridView s `AllowSorting` property to `true` and causing the header text for each column to be rendered as a LinkButton. When the end user clicks on one of the header LinkButtons, a postback ensues and the following steps transpire:

1. The GridView updates its `SortExpression` property to the value of the `SortExpression` of the field whose header link was clicked
2. The ObjectDataSource invokes the BLL s `GetProductsPagedAndSorted` method, passing in the GridView s `SortExpression` property as the value for the method s `sortExpression` input parameter (along with the appropriate `startRowIndex` and `maximumRows` input parameter values)
3. The BLL invokes the DAL s `GetProductsPagedAndSorted` method
4. The DAL executes the `GetProductsPagedAndSorted` stored procedure, passing in the `@sortExpression` parameter (along with the `@startRowIndex` and `@maximumRows` input parameter values)
5. The stored procedure returns the appropriate subset of data to the BLL, which returns it to the ObjectDataSource; this data is then bound to the GridView, rendered into HTML, and sent down to the end user

Figure 7 shows the first page of results when sorted by the `UnitPrice` in ascending order.

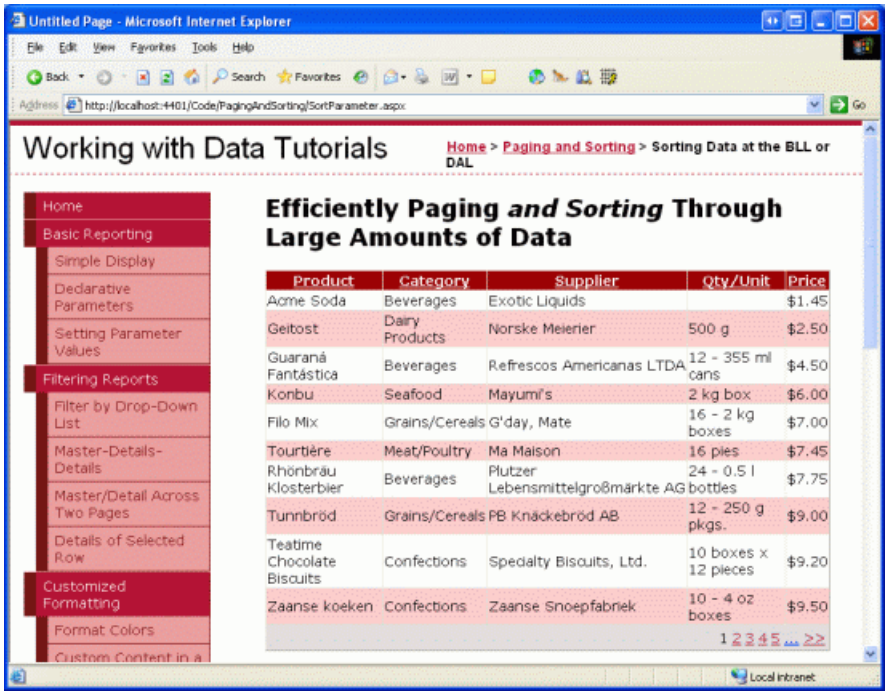


Figure 7: The Results are Sorted by the UnitPrice (Click to view full-size image)

While the current implementation can correctly sort the results by product name, category name, quantity per unit, and unit price, attempting to order the results by the supplier name results in a runtime exception (see Figure 8).

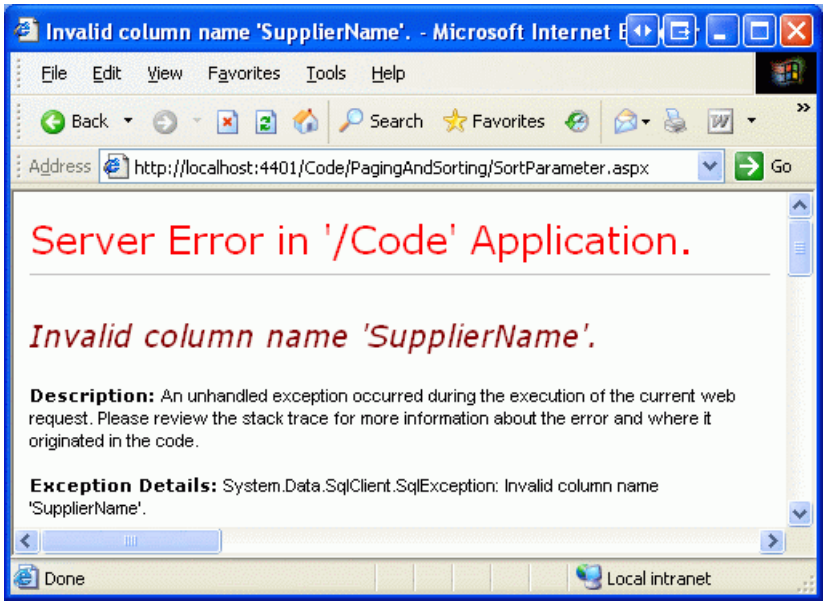


Figure 8: Attempting to Sort the Results by the Supplier Results in the Following Runtime Exception

This exception occurs because the `SortExpression` of the GridView s `SupplierName` BoundField is set to `SupplierName`. However, the supplier s name in the `Suppliers` table is actually called `CompanyName` we have been aliased this column name as `SupplierName`. However, the `OVER` clause used by the `ROW_NUMBER()` function cannot use the alias and must use the actual column name. Therefore, change the `SupplierName` BoundField s `SortExpression` from SupplierName to CompanyName (see Figure 9). As Figure 10 shows, after this change the results can be sorted by the supplier.

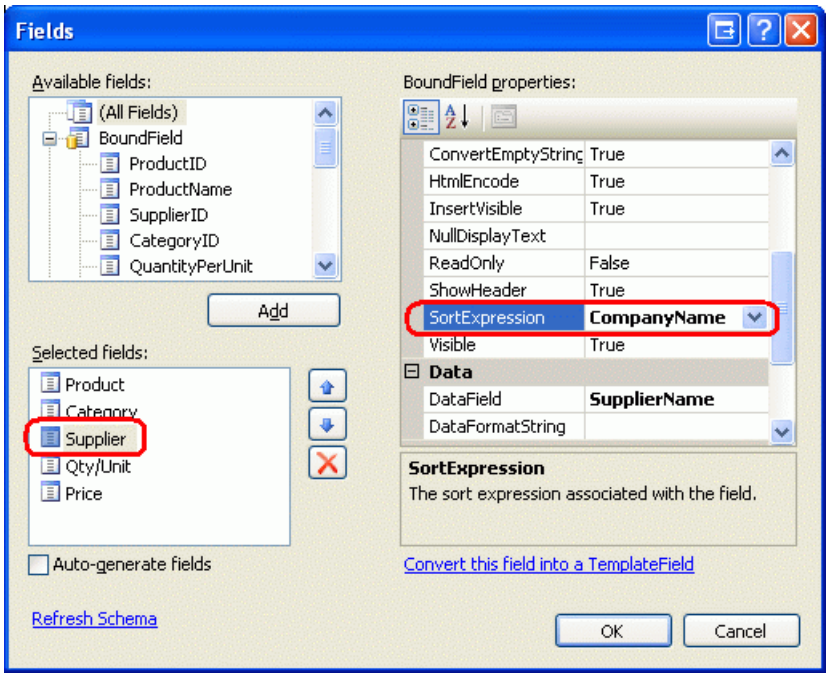


Figure 9: Change the SupplierName BoundField s SortExpression to CompanyName

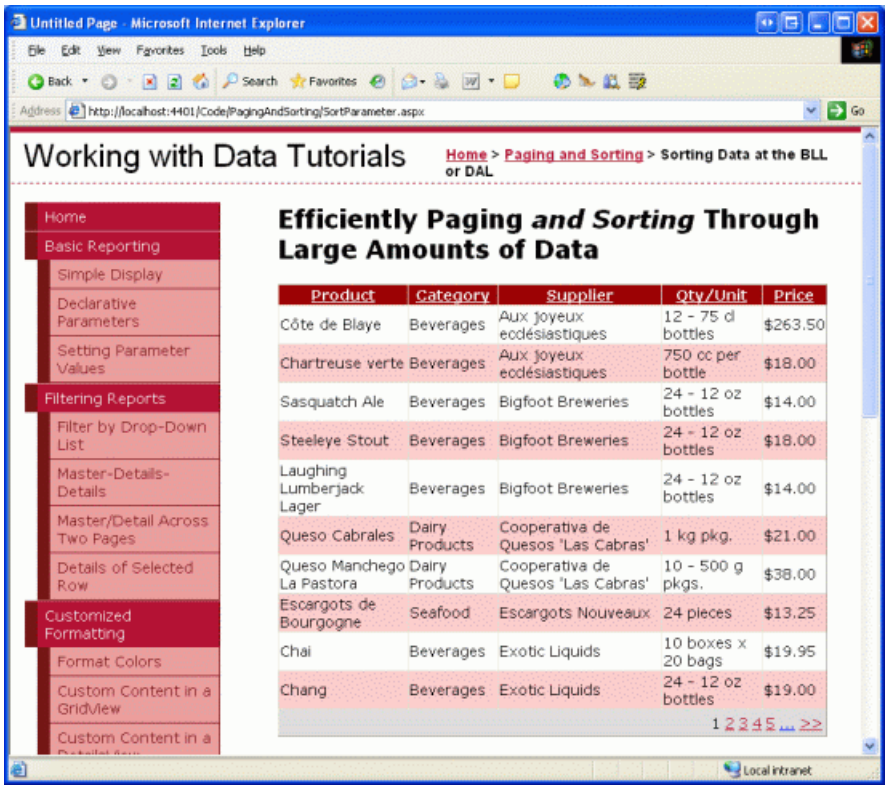


Figure 10: The Results Can Now Be Sorted by Supplier (Click to view full-size image)

Summary

The custom paging implementation we examined in the preceding tutorial required that the order by which the results were to be sorted be specified at design time. In short, this meant that the custom paging implementation we implemented could not, at the same time, provide sorting capabilities. In this tutorial we overcame this limitation by extending the stored procedure from the first to include a `@sortExpression` input parameter by which the results could be sorted.

After creating this stored procedure and creating new methods in the DAL and BLL, we were able to implement a GridView that offered both sorting and custom paging by configuring the ObjectDataSource to pass in the GridView s current `SortExpression` property to the BLL `SelectMethod`.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Carlos Santos. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Creating a Customized Sorting User Interface (VB)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> When displaying a long list of sorted data, it can be very helpful to group related data by introducing separator rows. In this tutorial we'll see how to create such a sorting user interface.

Introduction

When displaying a long list of sorted data where there are only a handful of different values in the sorted column, an end user might find it hard to discern where, exactly, the difference boundaries occur. For example, there are 81 products in the database, but only nine different category choices (eight unique categories plus the `NULL` option). Consider the case of a user who is interested in examining the products that fall under the Seafood category. From a page that lists *all* of the products in a single GridView, the user might decide her best bet is to sort the results by category, which will group together all of the Seafood products together. After sorting by category, the user then needs to hunt through the list, looking for where the Seafood-grouped products start and end. Since the results are ordered alphabetically by the category name finding the Seafood products is not difficult, but it still requires closely scanning the list of items in the grid.

To help highlight the boundaries between sorted groups, many websites employ a user interface that adds a separator between such groups. Separators like the ones shown in Figure 1 enables a user to more quickly find a particular group and identify its boundaries, as well as ascertain what distinct groups exist in the data.

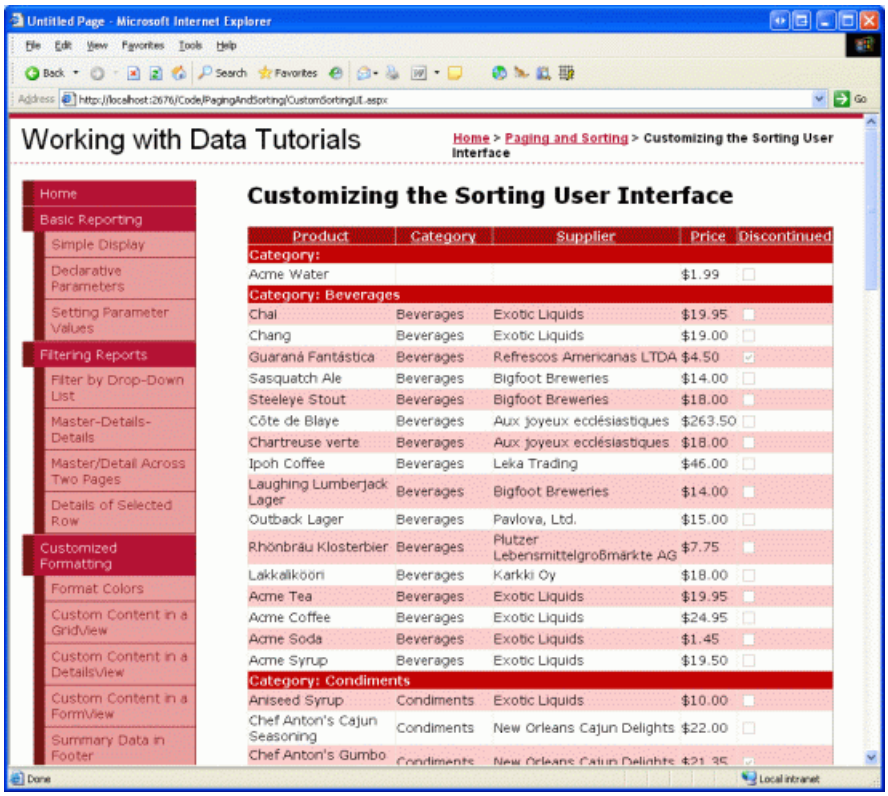


Figure 1: Each Category Group is Clearly Identified (Click to view full-size image)

In this tutorial we'll see how to create such a sorting user interface.

Step 1: Creating a Standard, Sortable GridView

Before we explore how to augment the GridView to provide the enhanced sorting interface, let s first create a standard, sortable GridView that lists the products. Start by opening the `CustomSortingUI.aspx` page in the `PagingAndSorting` folder. Add a GridView to the page, set its `ID` property to `ProductList`, and bind it to a new ObjectDataSource. Configure the ObjectDataSource to use the `ProductsBLL` class s `GetProducts()` method for selecting records.

Next, configure the GridView such that it only contains the `ProductName`, `CategoryName`, `SupplierName`, and `UnitPrice` BoundFields and the Discontinued CheckBoxField. Finally, configure the GridView to support sorting by checking the Enable Sorting checkbox in the GridView s smart tag (or by setting its `AllowSorting` property to `true`). After making these additions to the `CustomSortingUI.aspx` page, the declarative markup should look similar to the following:

```
<asp:GridView ID="ProductList" runat="server" AllowSorting="True"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
            ReadOnly="True" SortExpression="SupplierName" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:C}"
            HeaderText="Price" HtmlEncode="False" SortExpression="UnitPrice" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}" SelectMethod="GetProducts"
    TypeName="ProductsBLL"></asp:ObjectDataSource>
```

Take a moment to view our progress thus far in a browser. Figure 2 shows the sortable GridView when its data is sorted by category in alphabetical order.

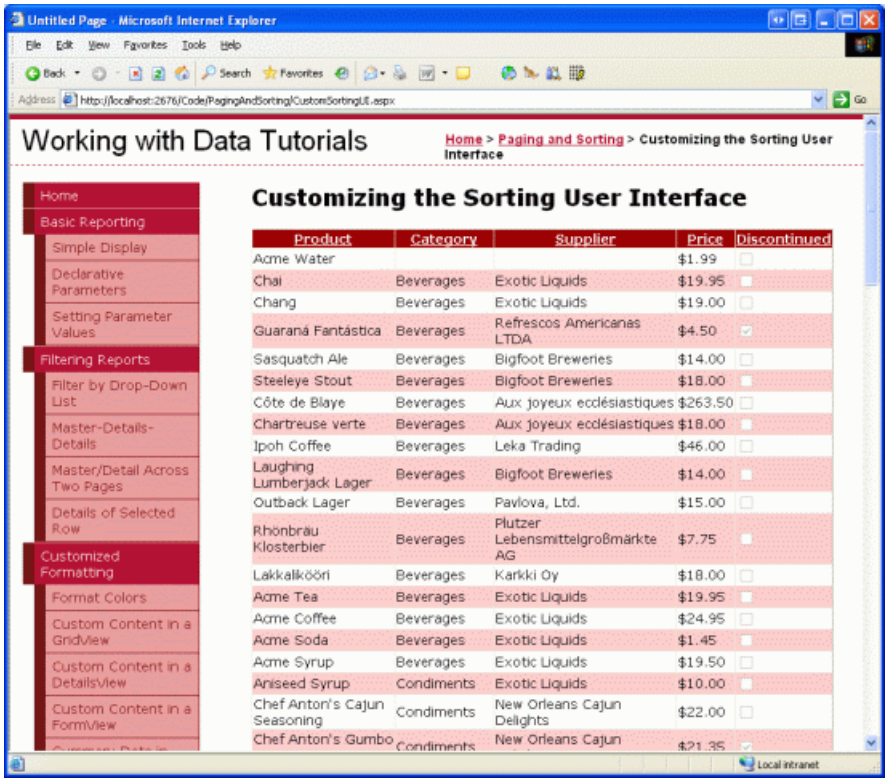


Figure 2: The Sortable GridView s Data is Ordered by Category (Click to view full-size image)

Step 2: Exploring Techniques for Adding the Separator Rows

With the generic, sortable GridView complete, all that remains is to be able to add the separator rows in the GridView before each unique sorted group. But how can such rows be injected into the GridView? Essentially, we need to iterate through the GridView s rows, determine where the differences occur between the values in the sorted column, and then add the appropriate separator row. When thinking about this problem, it seems natural that the solution lies somewhere in the GridView s `RowDataBound` event handler. As we discussed in the Custom Formatting Based Upon Data tutorial, this event handler is commonly used when applying row-level formatting based on the row s data. However, the `RowDataBound` event handler is not the solution here, as rows cannot be added to the GridView programmatically from this event handler. The GridView s `Rows` collection, in fact, is read-only.

To add additional rows to the GridView we have three choices:

- Add these metadata separator rows to the actual data that is bound to the GridView
- After the GridView has been bound to the data, add additional `TableRow` instances to the GridView s control collection
- Create a custom server control that extends the GridView control and overrides those methods responsible for constructing the GridView s structure

Creating a custom server control would be the best approach if this functionality was needed on many web pages or across several websites. However, it would entail quite a bit of code and a thorough exploration into the depths of the GridView s internal workings. Therefore, we'll not consider that option for this tutorial.

The other two options adding separator rows to the actual data being bound to the GridView and manipulating the GridView s control collection after its been bound - attack the problem differently and merit a discussion.

Adding Rows to the Data Bound to the GridView

When the GridView is bound to a data source, it creates a `GridViewRow` for each record returned by the data source. Therefore, we can inject the needed separator rows by adding separator records to the data source before binding it to the GridView. Figure 3 illustrates this concept.

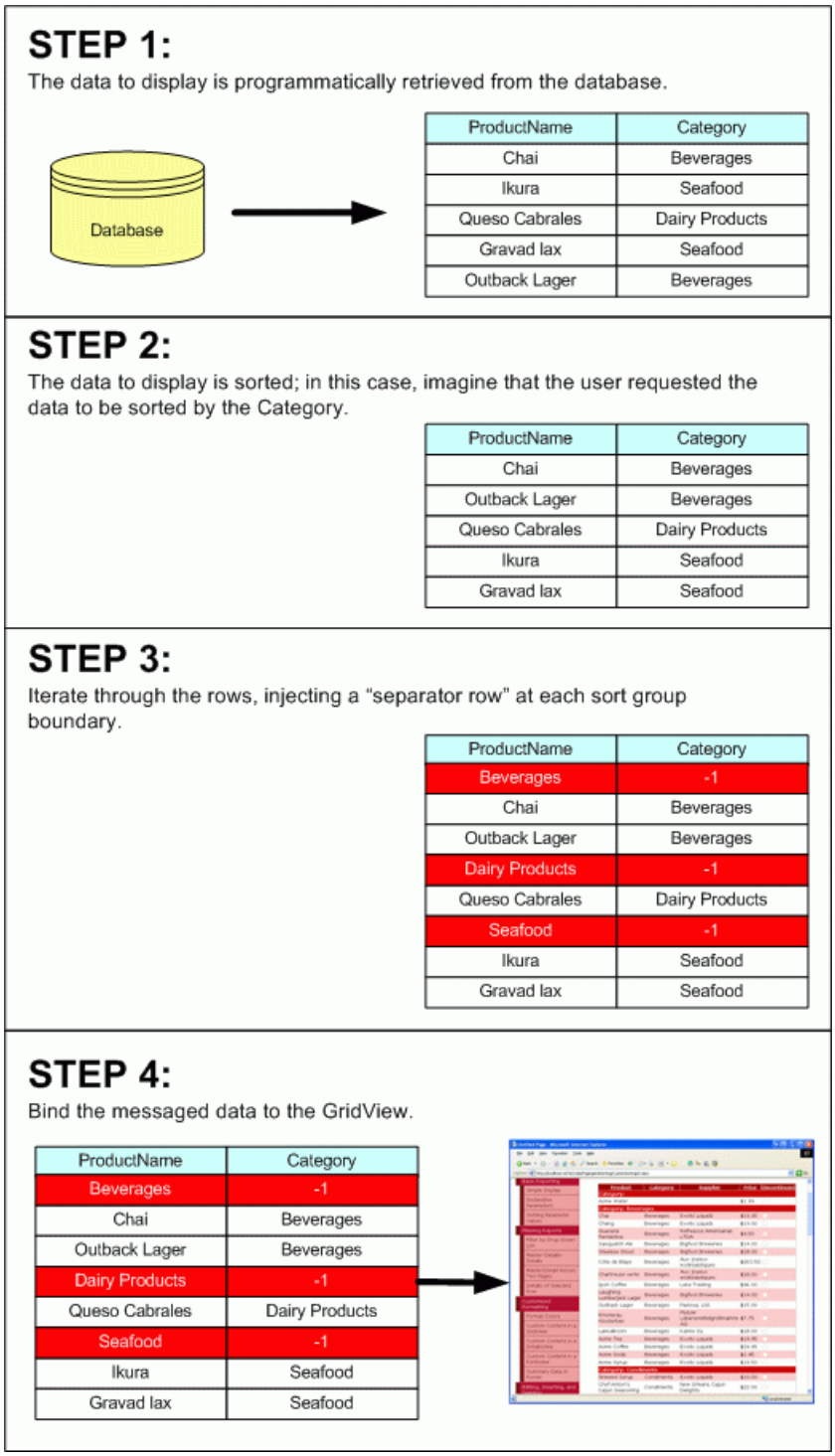


Figure 3: One Technique Involves Adding Separator Rows to the Data Source

I use the term separator records in quotes because there is no special separator record; rather, we must somehow flag that a particular record in the data source serves as a separator rather than a normal data row. For our examples, we re binding a `ProductsDataTable` instance to the GridView, which is composed of `ProductRows`. We might flag a record as a separator row by setting its `CategoryID` property to `-1` (since such a value couldn't exist normally).

To utilize this technique we d need to perform the following steps:

1. Programmatically retrieve the data to bind to the GridView (a `ProductsDataTable` instance)
2. Sort the data based on the GridView s `SortExpression` and `SortDirection` properties
3. Iterate through the `ProductsRows` in the `ProductsDataTable`, looking for where the differences in the sorted column lie
4. At each group boundary, inject a separator record `ProductsRow` instance into the DataTable, one that has it s

`CategoryID` set to `-1` (or whatever designation was decided upon to mark a record as a separator record)

5. After injecting the separator rows, programmatically bind the data to the GridView

In addition to these five steps, we d also need to provide an event handler for the GridView s `RowDataBound` event. Here, we d check each `DataRow` and determine if it was a separator row, one whose `CategoryID` setting was `-1`. If so, we d probably want to adjust its formatting or the text displayed in the cell(s).

Using this technique for injecting the sorting group boundaries requires a bit more work than outlined above, as you need to also provide an event handler for the GridView s `Sorting` event and keep track of the `SortExpression` and `SortDirection` values.

Manipulating the GridView s Control Collection After It s Been Databound

Rather than messaging the data before binding it to the GridView, we can add the separator rows *after* the data has been bound to the GridView. The process of data binding builds up the GridView s control hierarchy, which in reality is simply a `Table` instance composed of a collection of rows, each of which is composed of a collection of cells. Specifically, the GridView s control collection contains a `Table` object at its root, a `GridViewRow` (which is derived from the `TableRow` class) for each record in the `DataSource` bound to the GridView, and a `TableCell` object in each `GridViewRow` instance for each data field in the `DataSource`.

To add separator rows between each sorting group, we can directly manipulate this control hierarchy once it has been created. We can be confident that the GridView s control hierarchy has been created for the last time by the time the page is being rendered. Therefore, this approach overrides the `Page` class s `Render` method, at which point the GridView s final control hierarchy is updated to include the needed separator rows. Figure 4 illustrates this process.

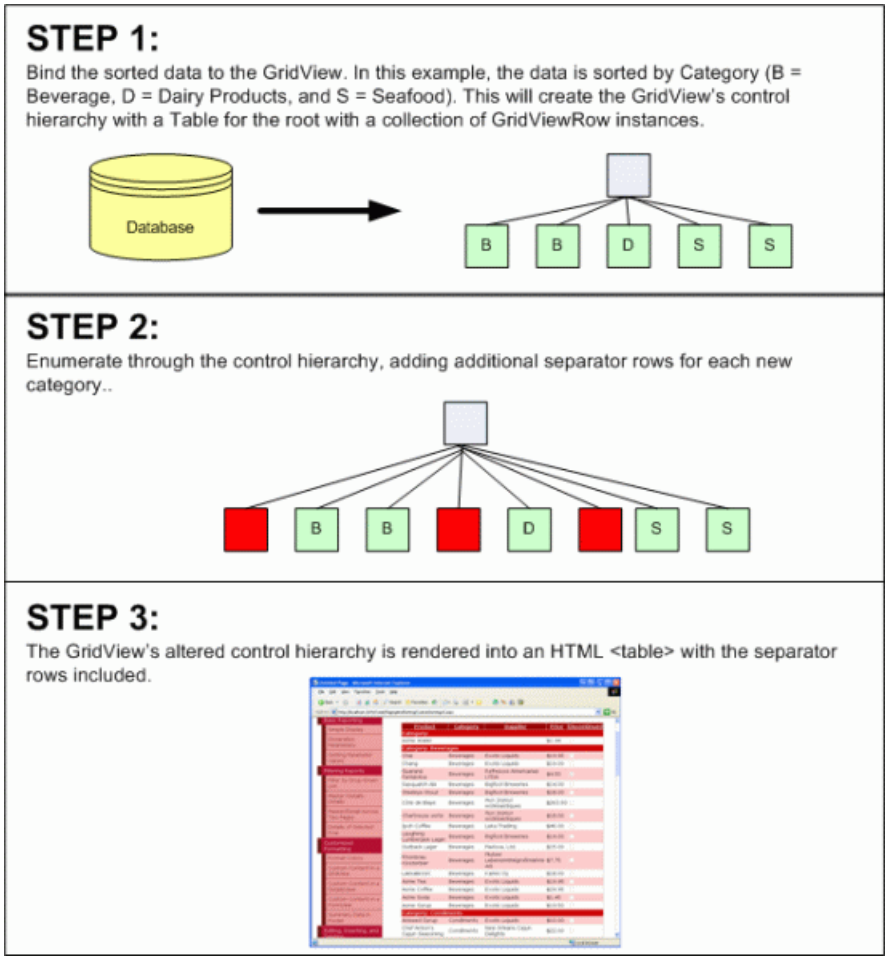


Figure 4: An Alternate Technique Manipulates the GridView s Control Hierarchy (Click to view full-size image)

For this tutorial, we'll use this latter approach to customize the sorting user experience.

> **NOTE**
> The code I m presenting in this tutorial is based on the example provided in Teemu Keiski s blog entry, Playing a Bit with GridView Sort Grouping.

Step 3: Adding the Separator Rows to the GridView s Control Hierarchy

Since we only want to add the separator rows to the GridView s control hierarchy after its control hierarchy has been created and created for the last time on that page visit, we want to perform this addition at the end of the page lifecycle, but before the actual GridView control hierarchy has been rendered into HTML. The latest possible point at which we can accomplish this is the `Page` class s `Render` event, which we can override in our code-behind class using the following method signature:

```
Protected Overrides Sub Render(ByVal writer As HtmlTextWriter)
   ' Add code to manipulate the GridView control hierarchy
   MyBase.Render(writer)
End Sub
```

When the `Page` class s original `Render` method is invoked `base.Render(writer)` each of the controls in the page will be rendered, generating the markup based on their control hierarchy. Therefore it is imperative that we both call `base.Render(writer)`, so that the page is rendered, and that we manipulate the GridView s control hierarchy prior to calling `base.Render(writer)`, so that the separator rows have been added to the GridView s control hierarchy before it s been rendered.

To inject the sort group headers we first need to ensure that the user has requested that the data be sorted. By default, the GridView s contents are not sorted, and therefore we don t need to enter any group sorting headers.

> **NOTE**
> If you want the GridView to be sorted by a particular column when the page is first loaded, call the GridView s `Sort` method on the first page visit (but not on subsequent postbacks). To accomplish this, add this call in the `Page_Load` event handler within an `if (!Page.IsPostBack)` conditional. Refer back to the Paging and Sorting Report Data tutorial information for more on the `Sort` method.

Assuming that the data has been sorted, our next task is to determine what column the data was sorted by and then to scan the rows looking for differences in that column s values. The following code ensures that the data has been sorted and finds the column by which the data has been sorted:

```
Protected Overrides Sub Render(ByVal writer As HtmlTextWriter)
    ' Only add the sorting UI if the GridView is sorted
    If Not String.IsNullOrEmpty(ProductList.SortExpression) Then
        ' Determine the index and HeaderText of the column that
        'the data is sorted by
        Dim sortColumnIndex As Integer = -1
        Dim sortColumnHeaderText As String = String.Empty
        For i As Integer = 0 To ProductList.Columns.Count - 1
            If ProductList.Columns(i).SortExpression.CompareTo( _
                ProductList.SortExpression) = 0 Then
                sortColumnIndex = i
                sortColumnHeaderText = ProductList.Columns(i).HeaderText
                Exit For
            End If
        Next
        ' TODO: Scan the rows for differences in the sorted column's values
End Sub
```

If the GridView has yet to be sorted, the GridView s `SortExpression` property will not have been set. Therefore, we only want to add the separator rows if this property has some value. If it does, we next need to determine the index of the column by which the data was sorted. This is accomplished by looping through the GridView s `Columns` collection, searching for the column whose `SortExpression` property equals the GridView s `SortExpression` property. In addition to the column s index, we also grab the `HeaderText` property, which is used when displaying the separator rows.

With the index of the column by which the data is sorted, the final step is to enumerate the rows of the GridView. For each row we need to determine whether the sorted column s value differs from the previous row s sorted column s value. If so, we need to inject a new `GridViewRow` instance into the control hierarchy. This is accomplished with the following code:

```
Protected Overrides Sub Render(ByVal writer As HtmlTextWriter)
    ' Only add the sorting UI if the GridView is sorted
    If Not String.IsNullOrEmpty(ProductList.SortExpression) Then
        ' ... Code for finding the sorted column index removed for brevity ...
        ' Reference the Table the GridView has been rendered into
        Dim gridTable As Table = CType(ProductList.Controls(0), Table)
        ' Enumerate each TableRow, adding a sorting UI header if
        ' the sorted value has changed
        Dim lastValue As String = String.Empty
        For Each gvr As GridViewRow In ProductList.Rows
            Dim currentValue As String = gvr.Cells(sortColumnIndex).Text
            If lastValue.CompareTo(currentValue) <> 0 Then
                ' there's been a change in value in the sorted column
                Dim rowIndex As Integer = gridTable.Rows.GetRowIndex(gvr)
                ' Add a new sort header row
                Dim sortRow As New GridViewRow(rowIndex, rowIndex, _
                    DataControlRowType.DataRow, DataControlRowState.Normal)
                Dim sortCell As New TableCell()
                sortCell.ColumnSpan = ProductList.Columns.Count
                sortCell.Text = String.Format("{0}: {1}", _
                    sortColumnHeaderText, currentValue)
                sortCell.CssClass = "SortHeaderRowStyle"
                ' Add sortCell to sortRow, and sortRow to gridTable
                sortRow.Cells.Add(sortCell)
                gridTable.Controls.AddAt(rowIndex, sortRow)
                ' Update lastValue
                lastValue = currentValue
            End If
        Next
    End If
    MyBase.Render(writer)
End Sub
```

This code starts by programmatically referencing the `Table` object found at the root of the GridView s control hierarchy and creating a string variable named `lastValue`. `lastValue` is used to compare the current row s sorted column value with the previous row s value. Next, the GridView s `Rows` collection is enumerated and for each row the value of the sorted column is stored in the `currentValue` variable.

> **NOTE**
> To determine the value of the particular row s sorted column I use the cell s `Text` property. This works well for BoundFields, but will not work as desired for TemplateFields, CheckBoxFields, and so on. We'll look at how to account for alternate GridView fields shortly.

The `currentValue` and `lastValue` variables are then compared. If they differ we need to add a new separator row to the control hierarchy. This is accomplished by determining the index of the `GridViewRow` in the `Table` object s `Rows` collection, creating new `GridViewRow` and `TableCell` instances, and then adding the `TableCell` and `GridViewRow` to the control hierarchy.

Note that the separator row s lone `TableCell` is formatted such that it spans the entire width of the GridView, is formatted using the `SortHeaderRowStyle` CSS class, and has its `Text` property such that it shows both the sort group name (such as Category) and the group s value (such as Beverages). Finally, `lastValue` is updated to the value of `currentValue`.

The CSS class used to format the sorting group header row `SortHeaderRowStyle` needs to be specified in the `Styles.css` file. Feel free to use whatever style settings appeal to you; I used the following:

```
.SortHeaderRowStyle
{
    background-color: #c00;
    text-align: left;
    font-weight: bold;
    color: White;
}
```

With the current code, the sorting interface adds sort group headers when sorting by any BoundField (see Figure 5, which shows a screenshot when sorting by supplier). However, when sorting by any other field type (such as a CheckBoxField or TemplateField), the sort group headers are nowhere to be found (see Figure 6).

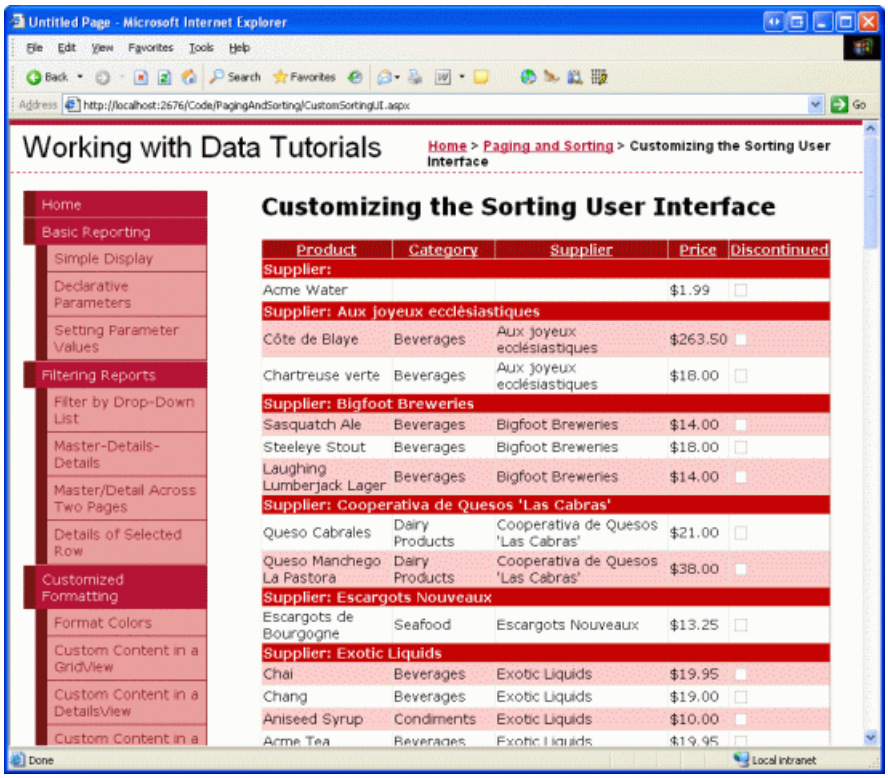


Figure 5: The Sorting Interface Includes Sort Group Headers When Sorting by BoundFields (Click to view full-size image)

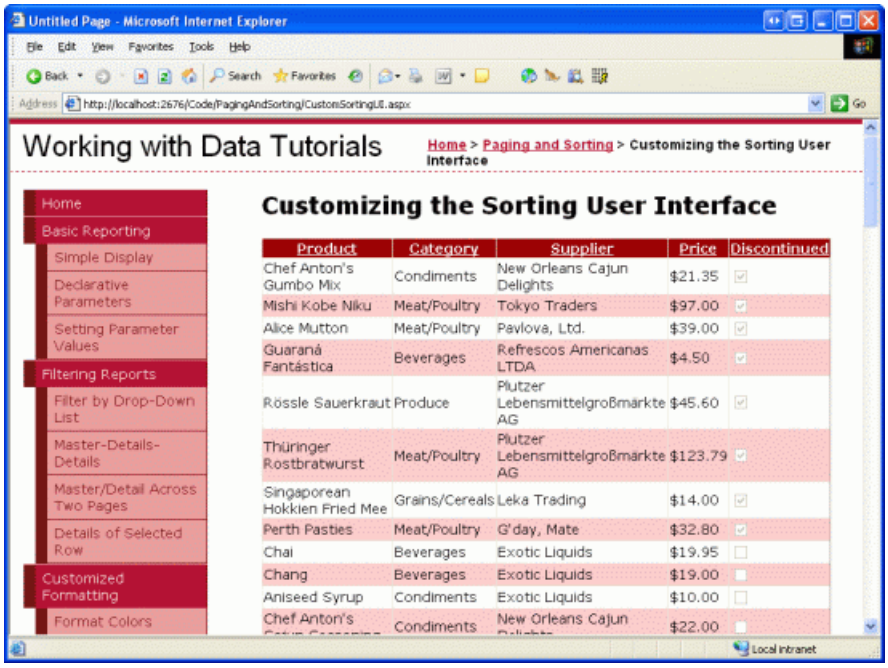


Figure 6: The Sort Group Headers are Missing When Sorting a CheckBoxField (Click to view full-size image)

The reason the sort group headers are missing when sorting by a CheckBoxField is because the code currently uses just the `TableCell` s `Text` property to determine the value of the sorted column for each row. For CheckBoxFields, the `TableCell` s `Text` property is an empty string; instead, the value is available through a CheckBox Web control that resides within the `TableCell` s `Controls` collection.

To handle field types other than BoundFields, we need to augment the code where the `currentValue` variable is assigned to check for the existence of a CheckBox in the `TableCell` s `Controls` collection. Instead of using `currentValue = gvr.Cells(sortColumnIndex).Text`, replace this code with the following:

```
Dim currentValue As String = String.Empty
If gvr.Cells(sortColumnIndex).Controls.Count > 0 Then
    If TypeOf gvr.Cells(sortColumnIndex).Controls(0) Is CheckBox Then
        If CType(gvr.Cells(sortColumnIndex).Controls(0), CheckBox).Checked Then
            currentValue = "Yes"
        Else
            currentValue = "No"
        End If
        ' ... Add other checks here if using columns with other
        '      Web controls in them (Calendars, DropDownLists, etc.) ...
    End If
Else
    currentValue = gvr.Cells(sortColumnIndex).Text
End If
```

This code examines the sorted column `TableCell` for the current row to determine if there are any controls in the `Controls` collection. If there are, and the first control is a CheckBox, the `currentValue` variable is set to Yes or No, depending on the CheckBox s `Checked` property. Otherwise, the value is taken from the `TableCell` s `Text` property. This logic can be replicated to handle sorting for any TemplateFields that may exist in the GridView.

With the above code addition, the sort group headers are now present when sorting by the Discontinued CheckBoxField (see Figure 7).

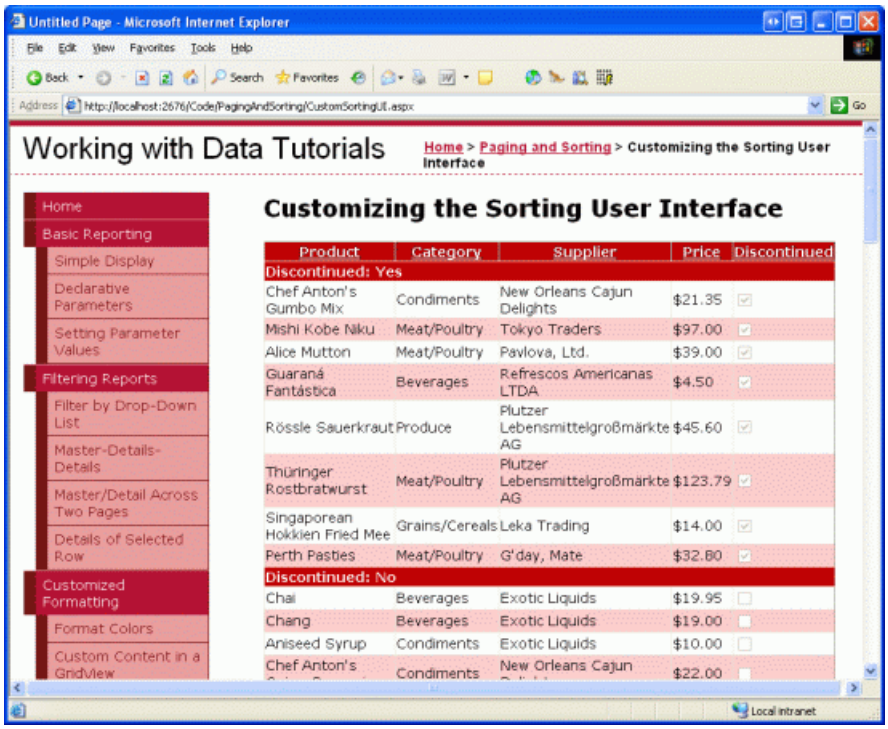


Figure 7: The Sort Group Headers are Now Present When Sorting a CheckBoxField (Click to view full-size image)

> **NOTE**
>
> If you have products with `NULL` database values for the `CategoryID`, `SupplierID`, or `UnitPrice` fields, those values will appear as empty strings in the GridView by default, meaning the separator row s text for those products with `NULL` values will read like Category: (that is, there s no name after Category: like with Category: Beverages). If you want a value displayed here you can either set the BoundFields `NullDisplayText` property to the text you want displayed or you can add a conditional statement in the Render method when assigning the `currentValue` to the separator row s `Text` property.

Summary

The GridView does not include many built-in options for customizing the sorting interface. However, with a bit of low-level code, it s possible to tweak the GridView s control hierarchy to create a more customized interface. In this tutorial we saw how to add a sort group separator row for a sortable GridView, which more easily identifies the distinct groups and those groups boundaries. For additional examples of customized sorting interfaces, check out Scott Guthrie s A Few ASP.NET 2.0 GridView Sorting Tips and Tricks blog entry.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS

Custom Button Actions

2/20/2020 • 2 minutes to read • Edit Online

This short set of tutorials shows how to add buttons to the GridView or DetailsView control that can perform any actions that your application needs.

- Adding and Responding to Buttons to a GridView (C#)
- Adding and Responding to Buttons to a GridView (VB)

Adding and Responding to Buttons to a GridView (C#)

2/20/2020 • 20 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll look at how to add custom buttons, both to a template and to the fields of a GridView or DetailsView control. In particular, we'll build an interface that has a FormView that allows the user to page through the suppliers.

Introduction

While many reporting scenarios involve read-only access to the report data, it's not uncommon for reports to include the ability to perform actions based upon the data displayed. Typically this involved adding a Button, LinkButton, or ImageButton Web control with each record displayed in the report that, when clicked, causes a postback and invokes some server-side code. Editing and deleting the data on a record-by-record basis is the most common example. In fact, as we saw starting with the Overview of Inserting, Updating, and Deleting Data tutorial, editing and deleting is so common that the GridView, DetailsView, and FormView controls can support such functionality without the need for writing a single line of code.

In addition to Edit and Delete buttons, the GridView, DetailsView, and FormView controls can also include Buttons, LinkButtons, or ImageButtons that, when clicked, perform some custom server-side logic. In this tutorial we'll look at how to add custom buttons, both to a template and to the fields of a GridView or DetailsView control. In particular, we'll build an interface that has a FormView that allows the user to page through the suppliers. For a given supplier, the FormView will show information about the supplier along with a Button Web control that, if clicked, will mark all of their associated products as discontinued. Additionally, a GridView lists those products provided by the selected supplier, with each row containing Increase Price and Discount Price Buttons that, if clicked, raise or reduce the product's `UnitPrice` by 10% (see Figure 1).

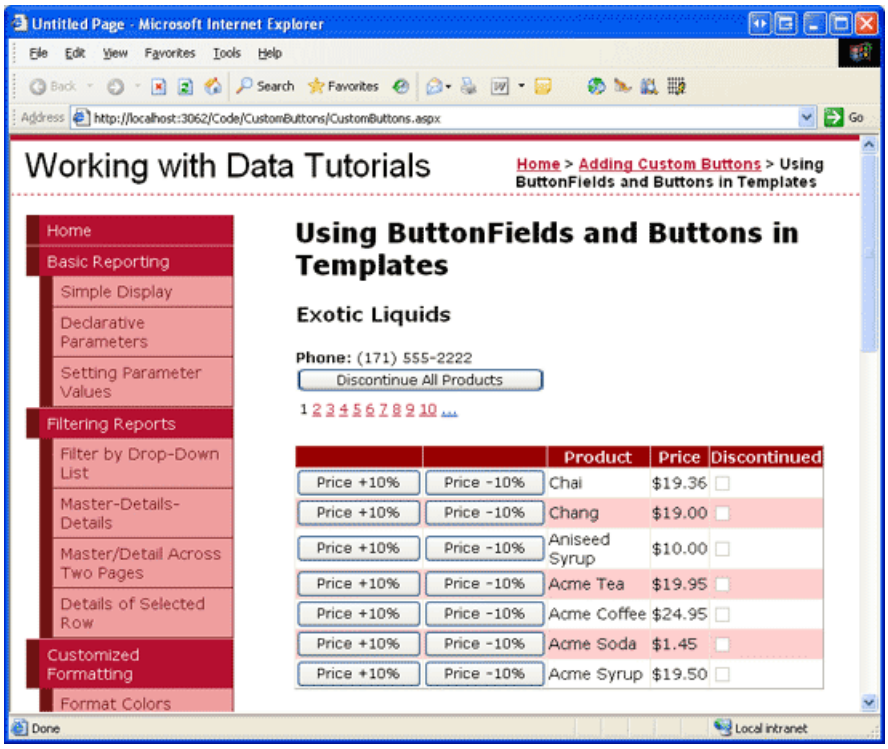


Figure 1: Both the FormView and GridView Contain Buttons That Perform Custom Actions (Click to view full-size image)

Step 1: Adding the Button Tutorial Web Pages

Before we look at how to add a custom buttons, let's first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial. Start by adding a new folder named `CustomButtons`. Next, add the following two ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `CustomButtons.aspx`

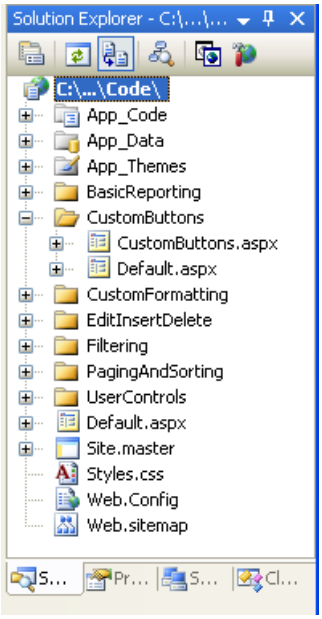


Figure 2: Add the ASP.NET Pages for the Custom Buttons-Related Tutorials

Like in the other folders, `Default.aspx` in the `CustomButtons` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page's Design view.

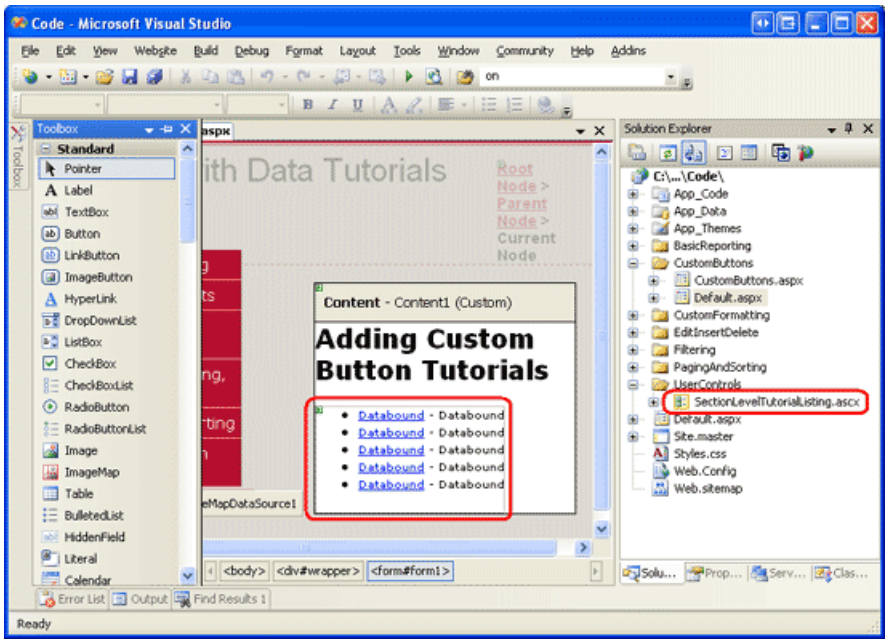


Figure 3: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add the pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Paging and Sorting `<siteMapNode>`:

```
<siteMapNode
    title="Adding Custom Buttons"
    description="Samples of Reports that Include Buttons for Performing
                 Server-Side Actions"
    url="~/CustomButtons/Default.aspx">
    <siteMapNode
        title="Using ButtonFields and Buttons in Templates"
        description="Examines how to add custom Buttons, LinkButtons,
                     or ImageButtons as ButtonFields or within templates."
        url="~/CustomButtons/CustomButtons.aspx" />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the editing, inserting, and deleting tutorials.

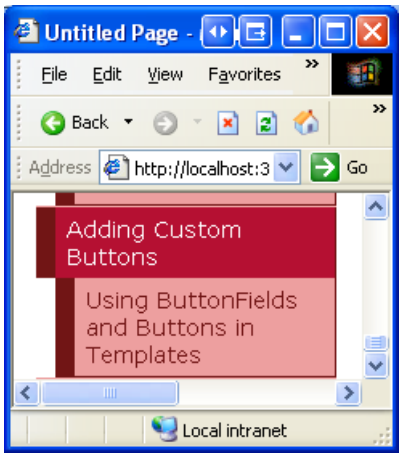


Figure 4: The Site Map Now Includes the Entry for the Custom Buttons Tutorial

Step 2: Adding a FormView that Lists the Suppliers

Let's get started with this tutorial by adding the FormView that lists the suppliers. As discussed in the Introduction, this FormView will allow the user to page through the suppliers, showing the products provided by the supplier in

a GridView. Additionally, this FormView will include a Button that, when clicked, will mark all of the supplier's products as discontinued. Before we concern ourselves with adding the custom button to the FormView, let's first just create the FormView so that it displays the supplier information.

Start by opening the `CustomButtons.aspx` page in the `CustomButtons` folder. Add a FormView to the page by dragging it from the Toolbox onto the Designer and set its `ID` property to `Suppliers`. From the FormView's smart tag, opt to create a new ObjectDataSource named `SuppliersDataSource`.

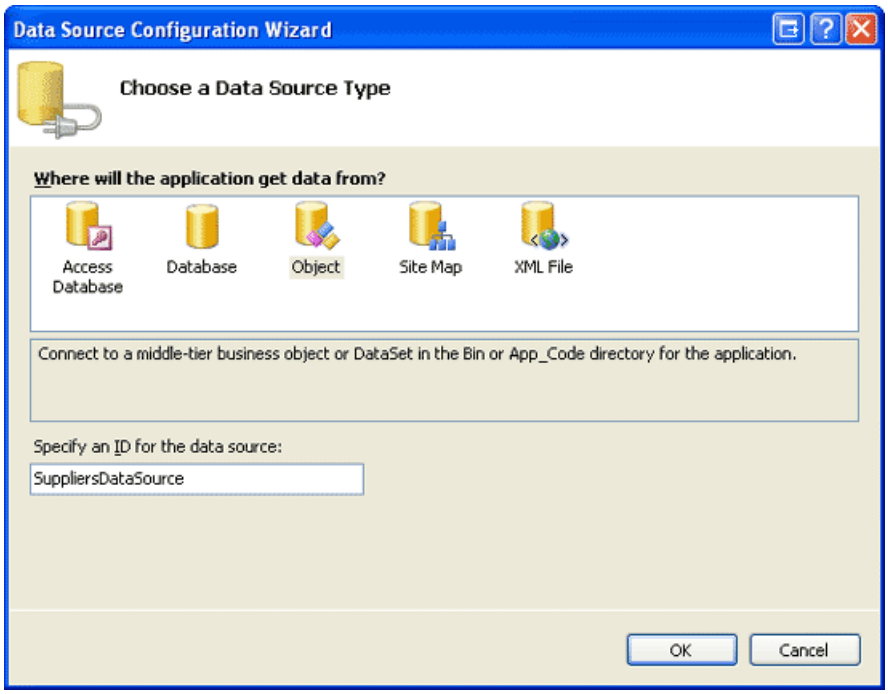


Figure 5: Create a New ObjectDataSource Named `SuppliersDataSource` (Click to view full-size image)

Configure this new ObjectDataSource such that it queries from the `SuppliersBLL` class's `GetSuppliers()` method (see Figure 6). Since this FormView does not provide an interface for updating the supplier information, select the (None) option from the drop-down list in the UPDATE tab.

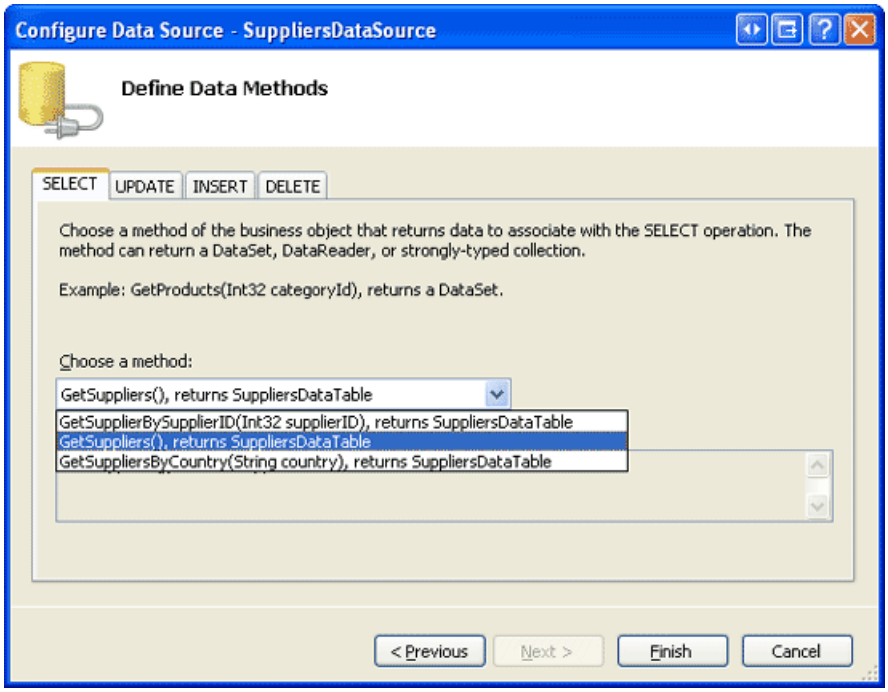


Figure 6: Configure the Data Source to use the `SuppliersBLL` Class's `GetSuppliers()` Method (Click to view full-size image)

After configuring the ObjectDataSource, Visual Studio will generate an `InsertItemTemplate`, `EditItemTemplate`,

and `ItemTemplate` for the FormView. Remove the `InsertItemTemplate` and `EditItemTemplate` and modify the `ItemTemplate` so that it displays just the supplier's company name and phone number. Finally, turn on paging support for the FormView by checking the Enable Paging checkbox from its smart tag (or by setting its `AllowPaging` property to `True`). After these changes your page's declarative markup should look similar to the following:

```
<asp:FormView ID="Suppliers" runat="server" DataKeyNames="SupplierID"
    DataSourceID="SuppliersDataSource" EnableViewState="False" AllowPaging="True">
    <ItemTemplate>
        <h3>
            <asp:Label ID="CompanyName" runat="server"
                Text='<%# Bind("CompanyName") %>' />
        </h3>
        <b>Phone:</b>
        <asp:Label ID="PhoneLabel" runat="server" Text='<%# Bind("Phone") %>' />
    </ItemTemplate>
</asp:FormView>
<asp:ObjectDataSource ID="SuppliersDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetSuppliers" TypeName="SuppliersBLL">
</asp:ObjectDataSource>
```

Figure 7 shows the CustomButtons.aspx page when viewed through a browser.

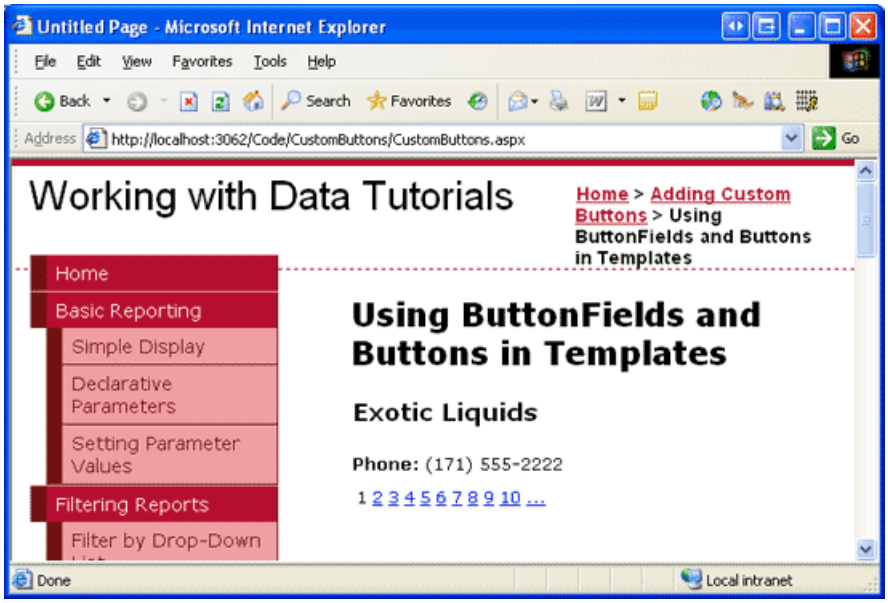


Figure 7: The FormView Lists the `CompanyName` and `Phone` Fields from the Currently Selected Supplier (Click to view full-size image)

Step 3: Adding a GridView that Lists the Selected Supplier's Products

Before we add the Discontinue All Products Button to the FormView's template, let's first add a GridView beneath the FormView that lists the products provided by the selected supplier. To accomplish this, add a GridView to the page, set its `ID` property to `SuppliersProducts`, and add a new ObjectDataSource named `SuppliersProductsDataSource`.

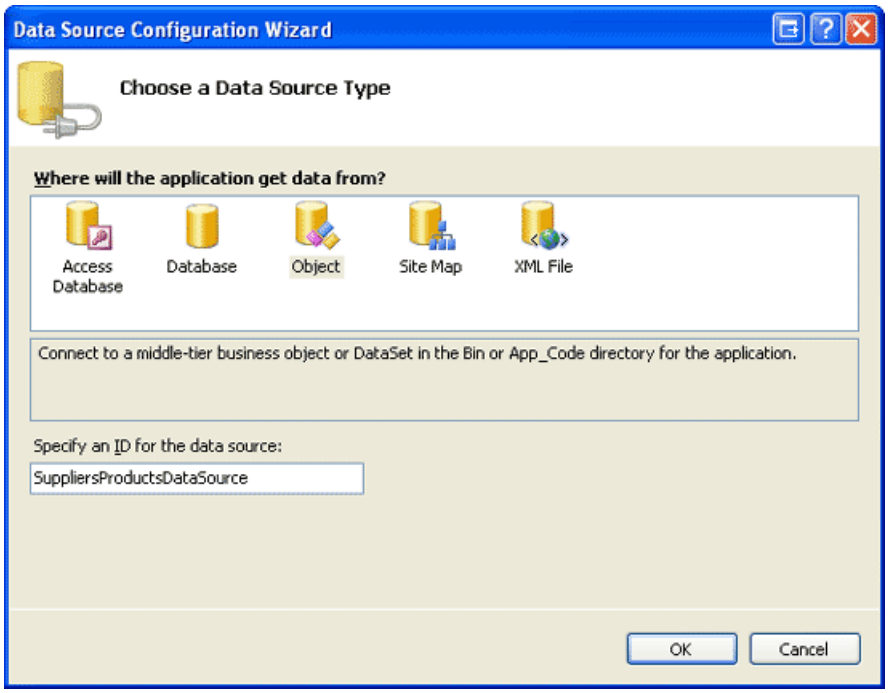


Figure 8: Create a New ObjectDataSource Named `SuppliersProductsDataSource` (Click to view full-size image)

Configure this ObjectDataSource to use the ProductsBLL class's `GetProductsBySupplierID(supplierID)` method (see Figure 9). While this GridView will allow for a product's price to be adjusted, it won't be using the built-in editing or deleting features from the GridView. Therefore, we can set the drop-down list to (None) for the ObjectDataSource's UPDATE, INSERT, and DELETE tabs.

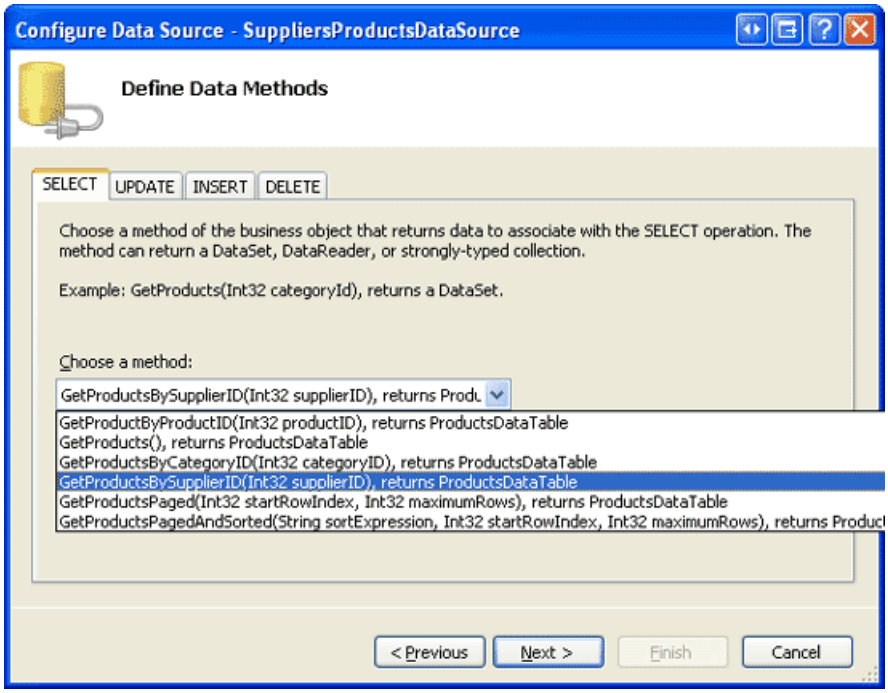


Figure 9: Configure the Data Source to use the `ProductsBLL` Class's `GetProductsBySupplierID(supplierID)` Method (Click to view full-size image)

Since the `GetProductsBySupplierID(supplierID)` method accepts an input parameter, the ObjectDataSource wizard prompts us for the source of this parameter value. To pass in the `SupplierID` value from the FormView, set the Parameter source drop-down list to Control and the ControlID drop-down list to `Suppliers` (the ID of the FormView created in Step 2).

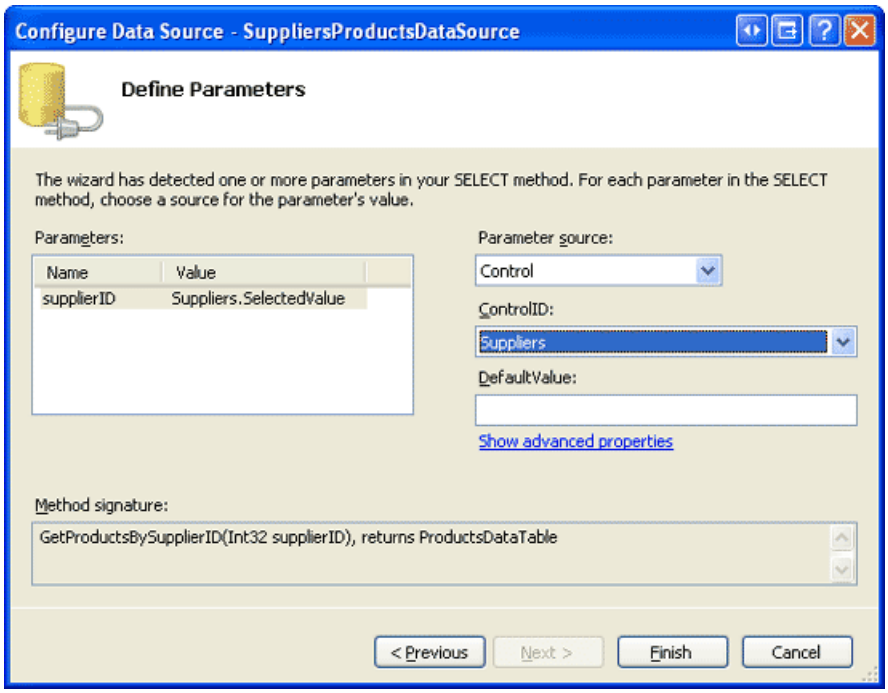


Figure 10: Indicate that the `supplierID` Parameter Should Come from the `Suppliers` FormView Control (Click to view full-size image)

After completing the ObjectDataSource wizard, the GridView will contain a BoundField or CheckBoxField for each of the product's data fields. Let's trim this down to show just the `ProductName` and `UnitPrice` BoundFields along with the `Discontinued` CheckBoxField; furthermore, let's format the `UnitPrice` BoundField such that its text is formatted as a currency. Your GridView and `SuppliersProductsDataSource` ObjectDataSource's declarative markup should look similar to the following markup:

```
<asp:GridView ID="SuppliersProducts" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="SuppliersProductsDataSource"
    EnableViewState="False" runat="server">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
            SortExpression="UnitPrice" DataFormatString="{0:C}"
            HtmlEncode="False" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="SuppliersProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProductsBySupplierID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:ControlParameter ControlID="Suppliers" Name="supplierID"
            PropertyName="SelectedValue" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

At this point our tutorial displays a master/details report, allowing the user to pick a supplier from the FormView at the top and to view the products provided by that supplier through the GridView at the bottom. Figure 11 shows a screen shot of this page when selecting the Tokyo Traders supplier from the FormView.

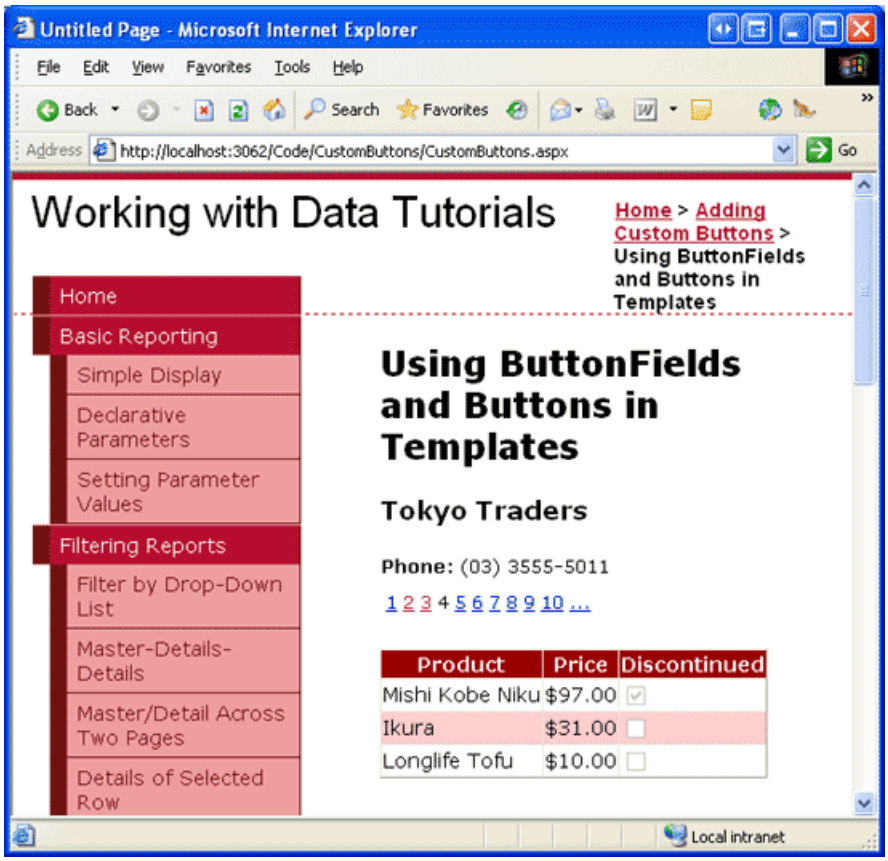


Figure 11: The Selected Supplier's Products are Displayed in the GridView (Click to view full-size image)

Step 4: Creating DAL and BLL Methods to Discontinue All Products for a Supplier

Before we can add a Button to the FormView that, when clicked, discontinues all of the supplier's products, we first need to add a method to both the DAL and BLL that performs this action. In particular, this method will be named `DiscontinueAllProductsForSupplier(supplierID)`. When the FormView's Button is clicked, we'll invoke this method in the Business Logic Layer, passing in the selected supplier's `SupplierID`; the BLL will then call down to the corresponding Data Access Layer method, which will issue an `UPDATE` statement to the database that discontinues the specified supplier's products.

As we have done in our previous tutorials, we'll use a bottom-up approach, starting with creating the DAL method, then the BLL method, and finally implementing the functionality in the ASP.NET page. Open the `Northwind.xsd` Typed DataSet in the `App_Code/DAL` folder and add a new method to the `ProductsTableAdapter` (right-click on the `ProductsTableAdapter` and choose Add Query). Doing so will bring up the TableAdapter Query Configuration wizard, which walks us through the process of adding the new method. Start by indicating that our DAL method will use an ad-hoc SQL statement.

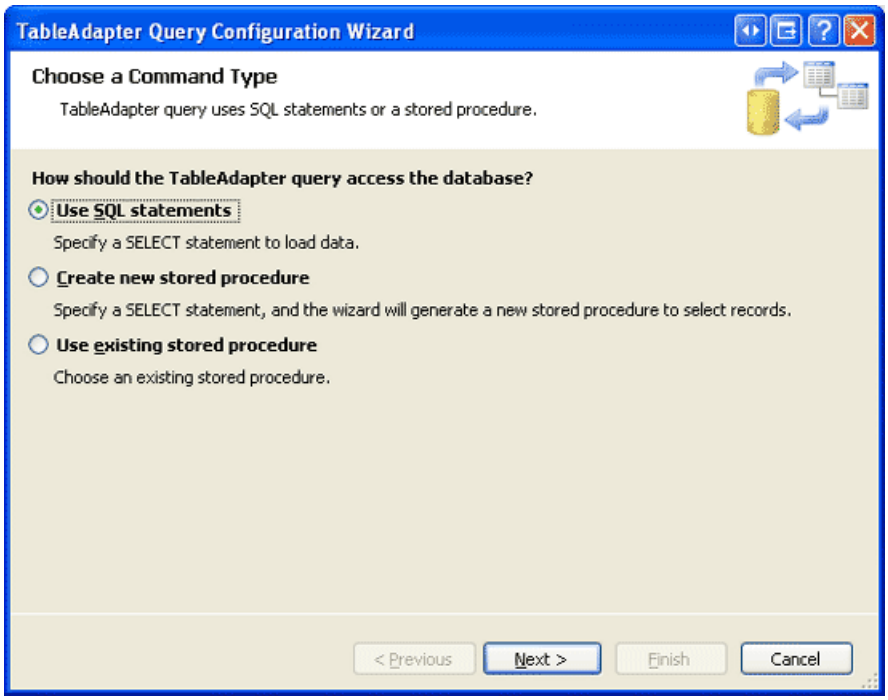


Figure 12: Create the DAL Method Using an Ad-Hoc SQL Statement (Click to view full-size image)

Next, the wizard prompts us as to what type of query to create. Since the `DiscontinueAllProductsForSupplier(supplierID)` method will need to update the `Products` database table, setting the `Discontinued` field to 1 for all products provided by the specified *`supplierID`*, we need to create a query that updates data.

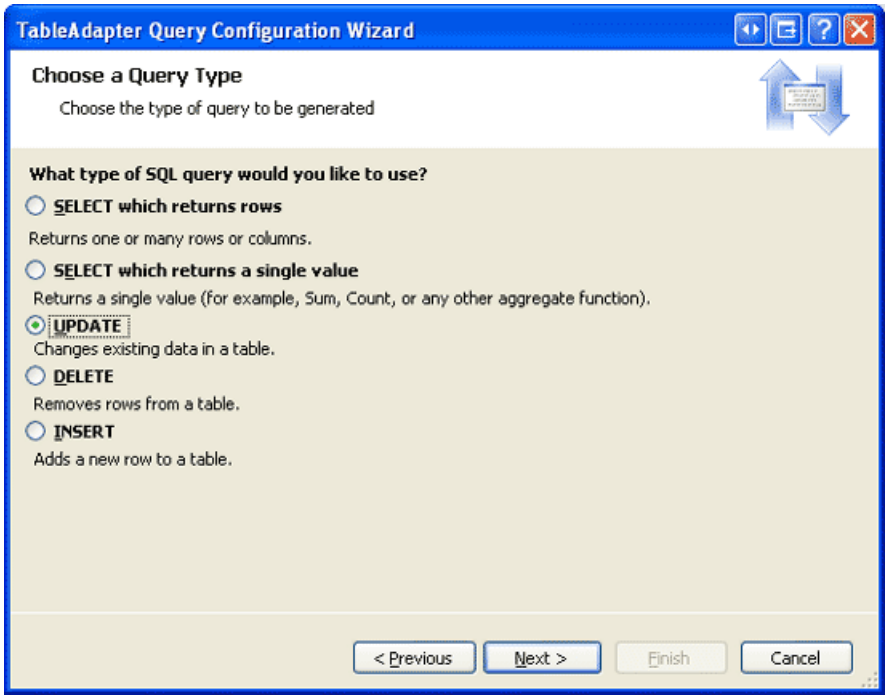


Figure 13: Choose the UPDATE Query Type (Click to view full-size image)

The next wizard screen provides the TableAdapter's existing `UPDATE` statement, which updates each of the fields defined in the `Products` DataTable. Replace this query text with the following statement:

```
UPDATE [Products] SET
   Discontinued = 1
WHERE SupplierID = @SupplierID
```

After entering this query and clicking Next, the last wizard screen asks for the new method's name use

`DiscontinueAllProductsForSupplier`. Complete the wizard by clicking the Finish button. Upon returning to the DataSet Designer you should see a new method in the `ProductsTableAdapter` named `DiscontinueAllProductsForSupplier(@SupplierID)`.

Figure 14: Name the New DAL Method `DiscontinueAllProductsForSupplier` (Click to view full-size image)

With the `DiscontinueAllProductsForSupplier(supplierID)` method created in the Data Access Layer, our next task is to create the `DiscontinueAllProductsForSupplier(supplierID)` method in the Business Logic Layer. To accomplish this, open the `ProductsBLL` class file and add the following:

```
public int DiscontinueAllProductsForSupplier(int supplierID)
{
    return Adapter.DiscontinueAllProductsForSupplier(supplierID);
}
```

This method simply calls down to the `DiscontinueAllProductsForSupplier(supplierID)` method in the DAL, passing along the provided *`supplierID`* parameter value. If there were any business rules that only allowed a supplier's products to be discontinued under certain circumstances, those rules should be implemented here, in the BLL.

> **NOTE**
> Unlike the `UpdateProduct` overloads in the `ProductsBLL` class, the `DiscontinueAllProductsForSupplier(supplierID)` method signature does not include the `DataObjectMethodAttribute` attribute (`<System.ComponentModel.DataObjectMethodAttribute(System.ComponentModel.DataObjectMethodType.Update, Boolean)>`). This precludes the `DiscontinueAllProductsForSupplier(supplierID)` method from the ObjectDataSource's Configure Data Source wizard's drop-down list in the UPDATE tab. I ve omitted this attribute because we'll be calling the `DiscontinueAllProductsForSupplier(supplierID)` method directly from an event handler in our ASP.NET page.

Step 5: Adding a Discontinue All Products Button to the FormView

With the `DiscontinueAllProductsForSupplier(supplierID)` method in the BLL and DAL complete, the final step for adding the ability to discontinue all products for the selected supplier is to add a Button Web control to the FormView's `ItemTemplate`. Let's add such a Button below the supplier's phone number with the button text, Discontinue All Products and an `ID` property value of `DiscontinueAllProductsForSupplier`. You can add this

Button Web control through the Designer by clicking on the Edit Templates link in the FormView's smart tag (see Figure 15), or directly through the declarative syntax.

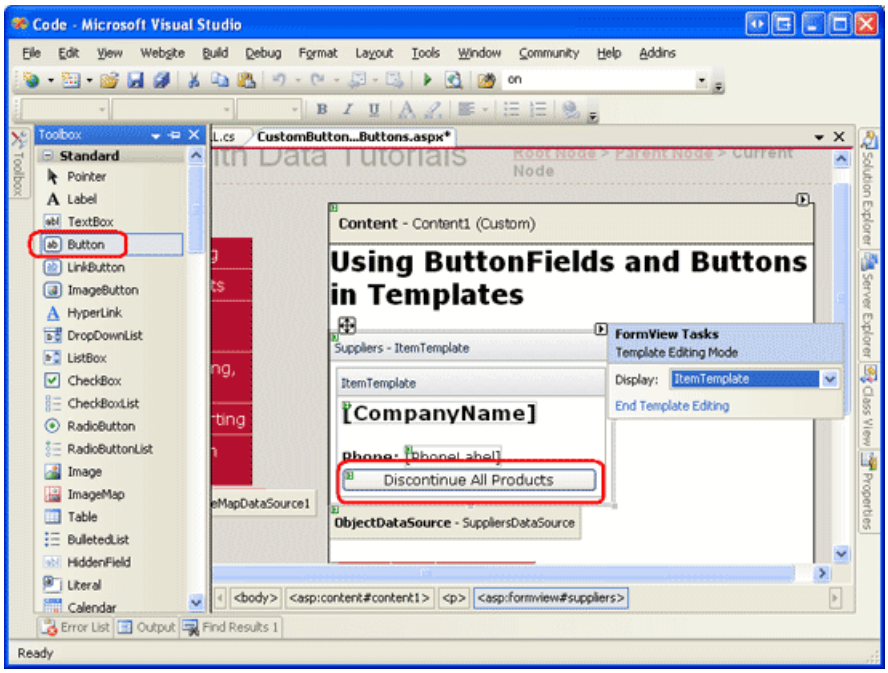


Figure 15: Add a Discontinue All Products Button Web Control to the FormView's `ItemTemplate` (Click to view full-size image)

When the Button is clicked by a user visiting the page, a postback ensues and the FormView's `ItemCommand` event fires. To execute custom code in response to this Button being clicked, we can create an event handler for this event. Understand, though, that the `ItemCommand` event fires whenever *any* Button, LinkButton, or ImageButton Web control is clicked within the FormView. That means that when the user moves from one page to another in the FormView, the `ItemCommand` event fires; same thing when the user clicks New, Edit, or Delete in a FormView that supports inserting, updating, or deleting.

Since the `ItemCommand` fires regardless of what button is clicked, in the event handler we need a way to determine if the Discontinue All Products Button was clicked or if it was some other button. To accomplish this, we can set the Button Web control's `CommandName` property to some identifying value. When the Button is clicked, this `CommandName` value is passed into the `ItemCommand` event handler, enabling us to determine whether the Discontinue All Products Button was the button clicked. Set the Discontinue All Products Button's `CommandName` property to DiscontinueProducts .

Finally, let's use a client-side confirm dialog box to ensure that the user really wants to discontinue the selected supplier's products. As we saw in the Adding Client-Side Confirmation When Deleting tutorial, this can be accomplished with a bit of JavaScript. In particular, set the Button Web control's OnClientClick property to

```
return confirm('This will mark _all_ of this supplier\'s products as discontinued. Are you certain you want to do this?');
```

After making these changes, the FormView's declarative syntax should look like the following:

```
<asp:FormView ID="Suppliers" runat="server" DataKeyNames="SupplierID"
    DataSourceID="SuppliersDataSource" EnableViewState="False"
    AllowPaging="True">
    <ItemTemplate>
        <h3><asp:Label ID="CompanyName" runat="server"
            Text='<%# Bind("CompanyName") %>'></asp:Label></h3>
        <b>Phone:</b>
        <asp:Label ID="PhoneLabel" runat="server" Text='<%# Bind("Phone") %>' />
        <br />
        <asp:Button ID="DiscontinueAllProductsForSupplier" runat="server"
            CommandName="DiscontinueProducts" Text="Discontinue All Products"
            OnClientClick="return confirm('This will mark _all_ of this supplier\'s
                products as discontinued. Are you certain you want to do this?');" />
    </ItemTemplate>
</asp:FormView>
```

Next, create an event handler for the FormView's `ItemCommand` event. In this event handler we need to first determine whether the Discontinue All Products Button was clicked. If so, we want to create an instance of the `ProductsBLL` class and invoke its `DiscontinueAllProductsForSupplier(supplierID)` method, passing in the `SupplierID` of the selected FormView:

```
protected void Suppliers_ItemCommand(object sender, FormViewCommandEventArgs e)
{
    if (e.CommandName.CompareTo("DiscontinueProducts") == 0)
    {
        // The "Discontinue All Products" Button was clicked.
        // Invoke the ProductsBLL.DiscontinueAllProductsForSupplier(supplierID) method
        // First, get the SupplierID selected in the FormView
        int supplierID = (int)Suppliers.SelectedValue;
        // Next, create an instance of the ProductsBLL class
        ProductsBLL productInfo = new ProductsBLL();
        // Finally, invoke the DiscontinueAllProductsForSupplier(supplierID) method
        productInfo.DiscontinueAllProductsForSupplier(supplierID);
    }
}
```

Note that the `SupplierID` of the current selected supplier in the FormView can be accessed using the FormView's `SelectedValue` property. The `SelectedValue` property returns the first data key value for the record being displayed in the FormView. The FormView's `DataKeyNames` property, which indicates the data fields from which the data key values are pulled from, was automatically set to `SupplierID` by Visual Studio when binding the ObjectDataSource to the FormView back in Step 2.

With the `ItemCommand` event handler created, take a moment to test out the page. Browse to the Cooperativa de Quesos 'Las Cabras' supplier (it's the fifth supplier in the FormView for me). This supplier provides two products, Queso Cabrales and Queso Manchego La Pastora, both of which are *not* discontinued.

Imagine that Cooperativa de Quesos 'Las Cabras' has gone out of business and therefore its products are to be discontinued. Click the Discontinue All Products Button. This will display the client-side confirm dialog box (see Figure 16).

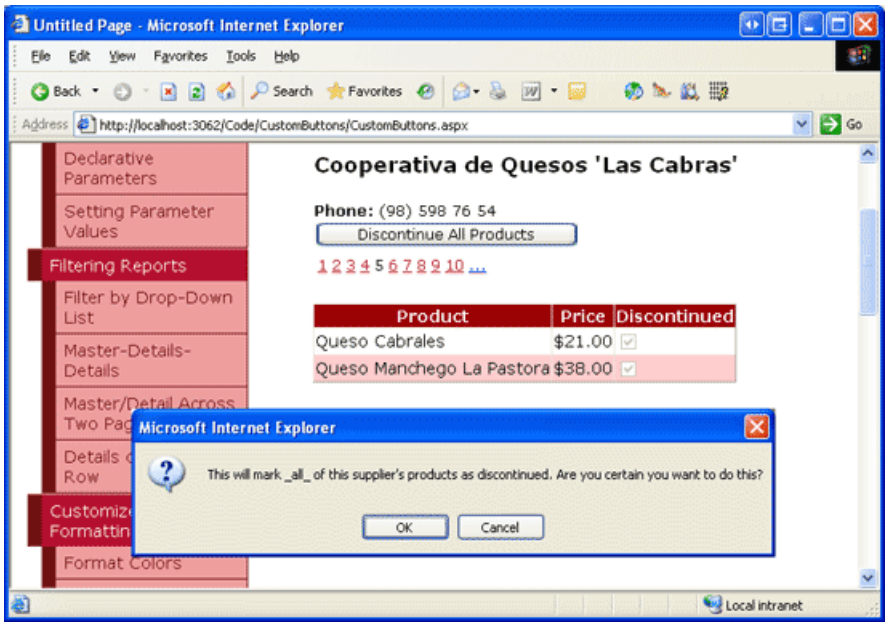


Figure 16: Cooperativa de Quesos Las Cabras Supplies Two Active Products (Click to view full-size image)

If you click OK in the client-side confirm dialog box, the form submission will proceed, causing a postback in which the FormView's `ItemCommand` event will fire. The event handler we created will then execute, invoking the `DiscontinueAllProductsForSupplier(supplierID)` method and discontinuing both the Queso Cabrales and Queso Manchego La Pastora products.

If you have disabled the GridView's view state, the GridView is being rebound to the underlying data store on every postback, and therefore will immediately be updated to reflect that these two products are now discontinued (see Figure 17). If, however, you have not disabled view state in the GridView, you will need to manually rebind the data to the GridView after making this change. To accomplish this, simply make a call to the GridView's `DataBind()` method immediately after invoking the `DiscontinueAllProductsForSupplier(supplierID)` method.

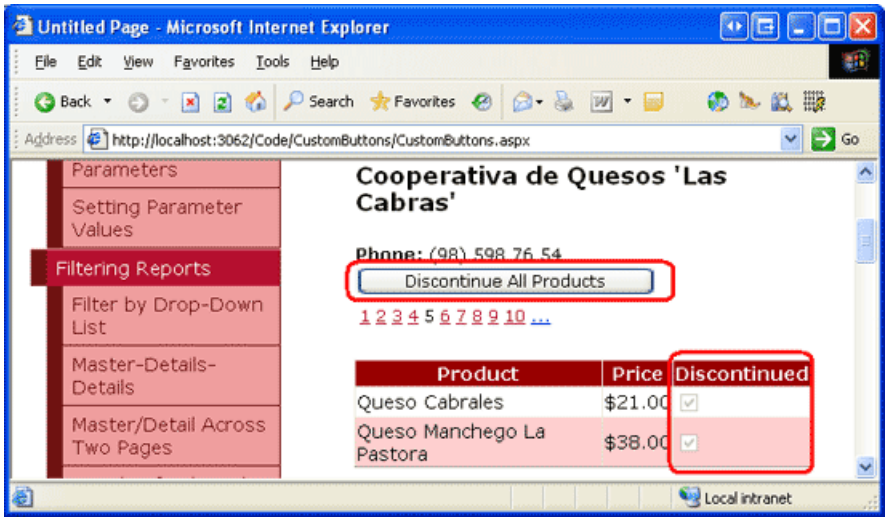


Figure 17: After Clicking the Discontinue All Products Button, the Supplier's Products are Updated Accordingly (Click to view full-size image)

Step 6: Creating an UpdateProduct Overload in the Business Logic Layer for Adjusting a Product's Price

Like with the Discontinue All Products Button in the FormView, in order to add buttons for increasing and decreasing the price for a product in the GridView we need to first add the appropriate Data Access Layer and Business Logic Layer methods. Since we already have a method that updates a single product row in the DAL, we

can provide such functionality by creating a new overload for the `UpdateProduct` method in the BLL.

Our past `UpdateProduct` overloads have taken in some combination of product fields as scalar input values and have then updated just those fields for the specified product. For this overload we'll vary slightly from this standard and instead pass in the product's `ProductID` and the percentage by which to adjust the `UnitPrice` (as opposed to passing in the new, adjusted `UnitPrice` itself). This approach will simplify the code we need to write in the ASP.NET page code-behind class, since we don't have to bother with determining the current product's `UnitPrice`.

The `UpdateProduct` overload for this tutorial is shown below:

```
public bool UpdateProduct(decimal unitPriceAdjustmentPercentage, int productID)
{
    Northwind.ProductsDataTable products = Adapter.GetProductByProductID(productID);
    if (products.Count == 0)
        // no matching record found, return false
        return false;
    Northwind.ProductsRow product = products[0];
    // Adjust the UnitPrice by the specified percentage (if it's not NULL)
    if (!product.IsUnitPriceNull())
        product.UnitPrice *= unitPriceAdjustmentPercentage;
    // Update the product record
    int rowsAffected = Adapter.Update(product);
    // Return true if precisely one row was updated, otherwise false
    return rowsAffected == 1;
}
```

This overload retrieves information about the specified product through the DAL's `GetProductByProductID(productID)` method. It then checks to see whether the product's `UnitPrice` is assigned a database `NULL` value. If it is, the price is left unaltered. If, however, there is a non-`NULL` `UnitPrice` value, the method updates the product's `UnitPrice` by the specified percent (`unitPriceAdjustmentPercent`).

Step 7: Adding the Increase and Decrease Buttons to the GridView

The GridView (and DetailsView) are both made up of a collection of fields. In addition to BoundFields, CheckBoxFields, and TemplateFields, ASP.NET includes the ButtonField, which, as its name implies, renders as a column with a Button, LinkButton, or ImageButton for each row. Similar to the FormView, clicking *any* button within the GridView paging buttons, Edit or Delete buttons, sorting buttons, and so on causes a postback and raises the GridView's `RowCommand` event.

The ButtonField has a `CommandName` property that assigns the specified value to each of its Buttons `CommandName` properties. Like with the FormView, the `CommandName` value is used by the `RowCommand` event handler to determine which button was clicked.

Let's add two new ButtonFields to the GridView, one with a button text Price +10% and the other with the text Price -10%. To add these ButtonFields, click on the Edit Columns link from the GridView's smart tag, select the ButtonField field type from the list in the upper left and click the Add button.

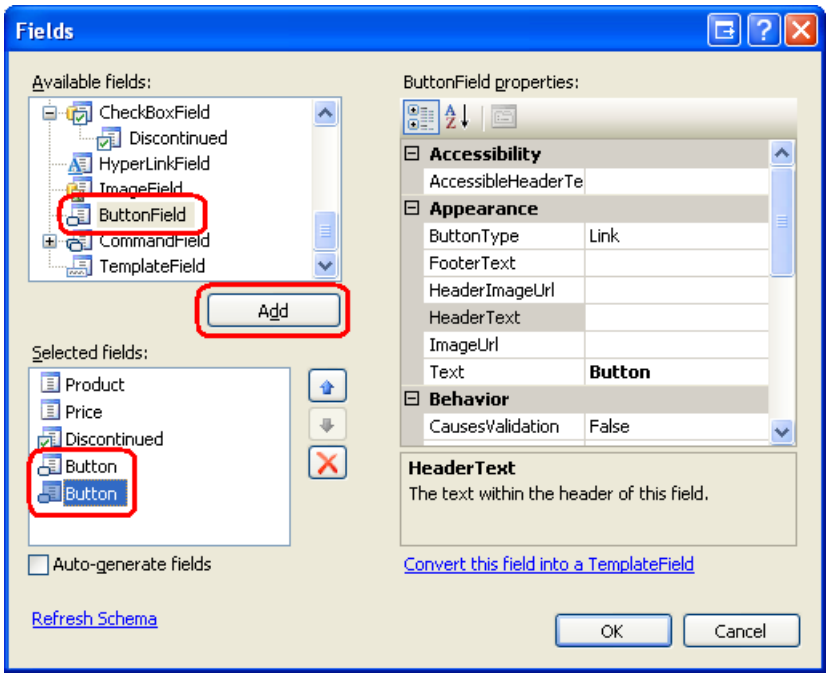


Figure 18: Add Two ButtonFields to the GridView

Move the two ButtonFields so that they appear as the first two GridView fields. Next, set the `Text` properties of these two ButtonFields to Price +10% and Price -10% and the `CommandName` properties to IncreasePrice and DecreasePrice, respectively. By default, a ButtonField renders its column of buttons as LinkButtons. This can be changed, however, through the ButtonField's `ButtonType` property. Let's have these two ButtonFields rendered as regular push buttons; therefore, set the `ButtonType` property to `Button`. Figure 19 shows the Fields dialog box after these changes have been made; following that is the GridView's declarative markup.

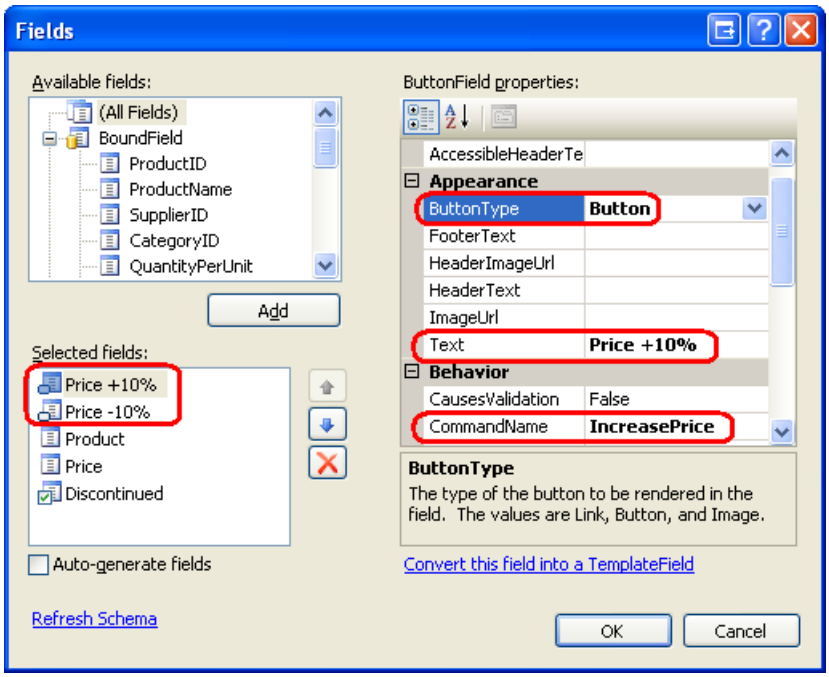


Figure 19: Configure the ButtonFields `Text`, `CommandName`, and `ButtonType` Properties

```
<asp:GridView ID="SuppliersProducts" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="SuppliersProductsDataSource"
    EnableViewState="False">
    <Columns>
        <asp:ButtonField ButtonType="Button" CommandName="IncreasePrice"
            Text="Price +10%" />
        <asp:ButtonField ButtonType="Button" CommandName="DecreasePrice"
            Text="Price -10%" />
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
            SortExpression="UnitPrice" DataFormatString="{0:C}"
            HtmlEncode="False" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
```

With these ButtonFields created, the final step is to create an event handler for the GridView's `RowCommand` event. This event handler, if fired because either the Price +10% or Price -10% buttons were clicked, needs to determine the `ProductID` for the row whose button was clicked and then invoke the `ProductsBLL` class's `UpdateProduct` method, passing in the appropriate `UnitPrice` percentage adjustment along with the `ProductID`. The following code performs these tasks:

```
protected void SuppliersProducts_RowCommand(object sender, GridViewCommandEventArgs e)
{
    if (e.CommandName.CompareTo("IncreasePrice") == 0 ||
        e.CommandName.CompareTo("DecreasePrice") == 0)
    {
        // The Increase Price or Decrease Price Button has been clicked
        // Determine the ID of the product whose price was adjusted
        int productID =
            (int)SuppliersProducts.DataKeys[Convert.ToInt32(e.CommandArgument)].Value;
        // Determine how much to adjust the price
        decimal percentageAdjust;
        if (e.CommandName.CompareTo("IncreasePrice") == 0)
            percentageAdjust = 1.1M;
        else
            percentageAdjust = 0.9M;

        // Adjust the price
        ProductsBLL productInfo = new ProductsBLL();
        productInfo.UpdateProduct(percentageAdjust, productID);
    }
}
```

In order to determine the `ProductID` for the row whose Price +10% or Price -10% button was clicked, we need to consult the GridView's `DataKeys` collection. This collection holds the values of the fields specified in the `DataKeyNames` property for each GridView row. Since the GridView's `DataKeyNames` property was set to ProductID by Visual Studio when binding the ObjectDataSource to the GridView, `DataKeys(rowIndex).Value` provides the `ProductID` for the specified *rowIndex*.

The ButtonField automatically passes in the *rowIndex* of the row whose button was clicked through the `e.CommandArgument` parameter. Therefore, to determine the `ProductID` for the row whose Price +10% or Price -10% button was clicked, we use: `Convert.ToInt32(SuppliersProducts.DataKeys(Convert.ToInt32(e.CommandArgument)).Value)`.

As with the Discontinue All Products button, if you have disabled the GridView's view state, the GridView is being rebound to the underlying data store on every postback, and therefore will immediately be updated to reflect a price change that occurs from clicking either of the buttons. If, however, you have not disabled view state in the

GridView, you will need to manually rebind the data to the GridView after making this change. To accomplish this, simply make a call to the GridView's `DataBind()` method immediately after invoking the `UpdateProduct` method.

Figure 20 shows the page when viewing the products provided by Grandma Kelly's Homestead. Figure 21 shows the results after the Price +10% button has been clicked twice for Grandma's Boysenberry Spread and the Price -10% button once for Northwoods Cranberry Sauce.

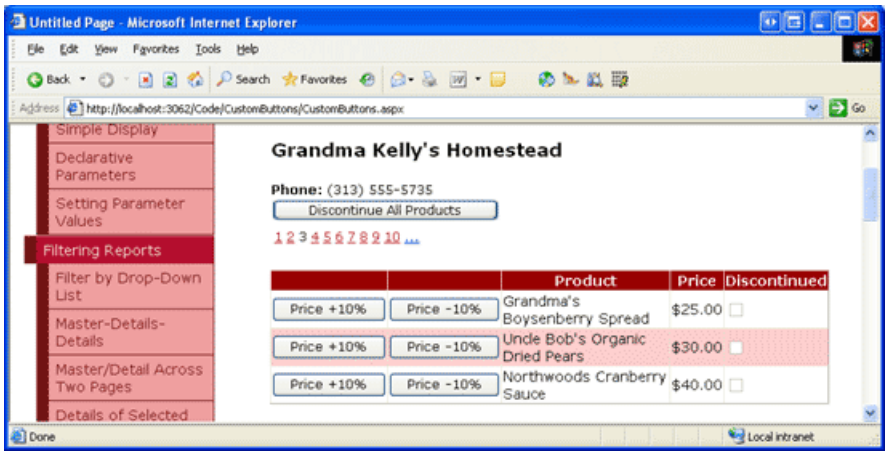


Figure 20: The GridView Includes Price +10% and Price -10% Buttons (Click to view full-size image)

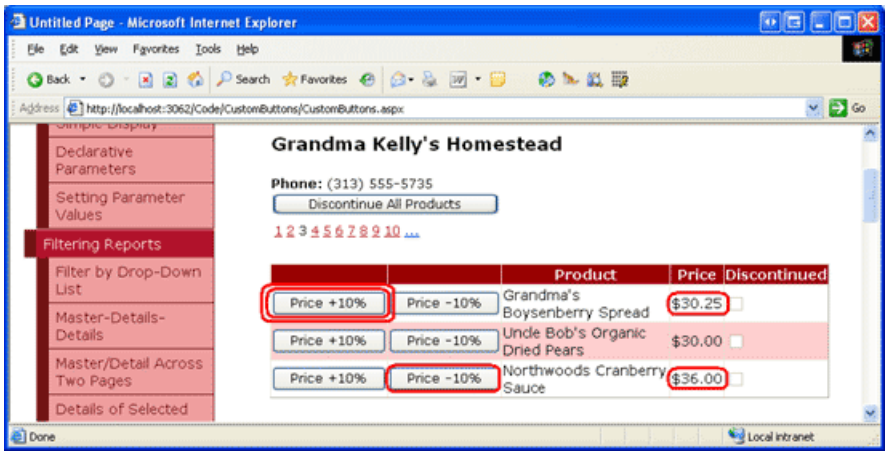


Figure 21: The Prices for the First and Third Product Have Been Updated via the Price +10% and Price -10% Buttons (Click to view full-size image)

> **NOTE**
>
> The GridView (and DetailsView) can also have Buttons, LinkButtons, or ImageButtons added to their TemplateFields. As with the BoundField, these Buttons, when clicked, will induce a postback, raising the GridView's `RowCommand` event. When adding buttons in a TemplateField, however, the Button's `CommandArgument` is not automatically set to the index of the row as it is when using ButtonFields. If you need to determine the row index of the button that was clicked within the `RowCommand` event handler, you'll need to manually set the Button's `CommandArgument` property in its declarative syntax within the TemplateField, using code like:
>
> `<asp:Button runat="server" ... CommandArgument='<%# ((GridViewRow) Container).RowIndex %>'`.

Summary

The GridView, DetailsView, and FormView controls all can include Buttons, LinkButtons, or ImageButtons. Such buttons, when clicked, cause a postback and raise the `ItemCommand` event in the FormView and DetailsView controls and the `RowCommand` event in the GridView. These data Web controls have built-in functionality to handle common command-related actions, such as deleting or editing records. However, we can also use buttons that, when clicked, respond with executing our own custom code.

To accomplish this, we need to create an event handler for the `ItemCommand` or `RowCommand` event. In this event handler we first check the incoming `CommandName` value to determine which button was clicked and then take appropriate custom action. In this tutorial we saw how to use buttons and ButtonFields to discontinue all products for a specified supplier or to increase or decrease the price of a particular product by 10%.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

NEXT

Adding and Responding to Buttons to a GridView (VB)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll look at how to add custom buttons, both to a template and to the fields of a GridView or DetailsView control. In particular, we'll build an interface that has a FormView that allows the user to page through the suppliers.

Introduction

While many reporting scenarios involve read-only access to the report data, it s not uncommon for reports to include the ability to perform actions based upon the data displayed. Typically this involved adding a Button, LinkButton, or ImageButton Web control with each record displayed in the report that, when clicked, causes a postback and invokes some server-side code. Editing and deleting the data on a record-by-record basis is the most common example. In fact, as we saw starting with the Overview of Inserting, Updating, and Deleting Data tutorial, editing and deleting is so common that the GridView, DetailsView, and FormView controls can support such functionality without the need for writing a single line of code.

In addition to Edit and Delete buttons, the GridView, DetailsView, and FormView controls can also include Buttons, LinkButtons, or ImageButtons that, when clicked, perform some custom server-side logic. In this tutorial we'll look at how to add custom buttons, both to a template and to the fields of a GridView or DetailsView control. In particular, we'll build an interface that has a FormView that allows the user to page through the suppliers. For a given supplier, the FormView will show information about the supplier along with a Button Web control that, if clicked, will mark all of their associated products as discontinued. Additionally, a GridView lists those products provided by the selected supplier, with each row containing Increase Price and Discount Price Buttons that, if clicked, raise or reduce the product s `UnitPrice` by 10% (see Figure 1).

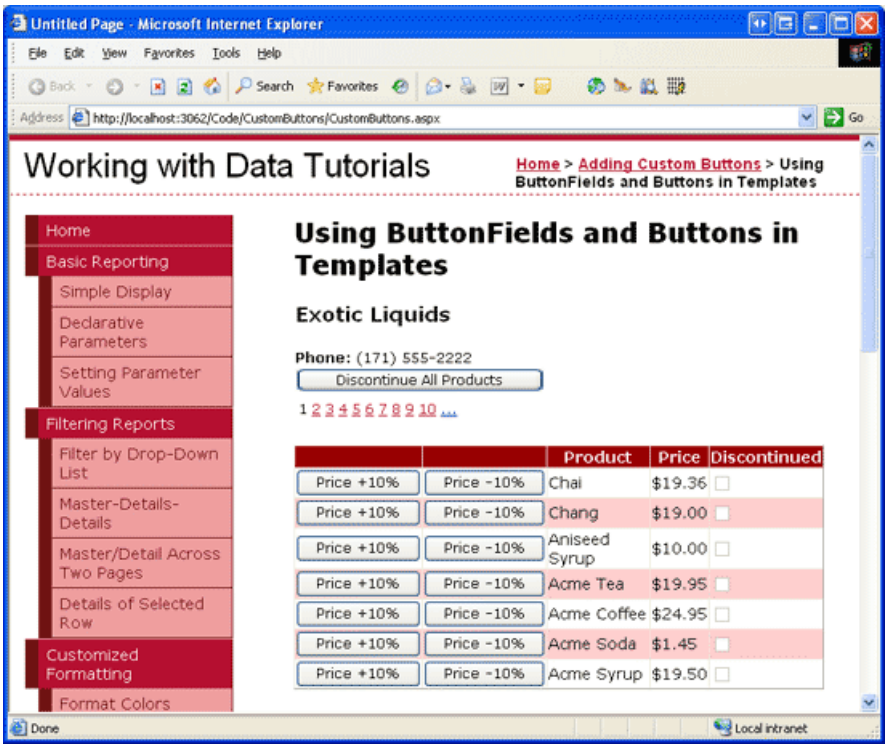


Figure 1: Both the FormView and GridView Contain Buttons That Perform Custom Actions (Click to view full-size image)

Step 1: Adding the Button Tutorial Web Pages

Before we look at how to add a custom buttons, let s first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial. Start by adding a new folder named `CustomButtons`. Next, add the following two ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `CustomButtons.aspx`

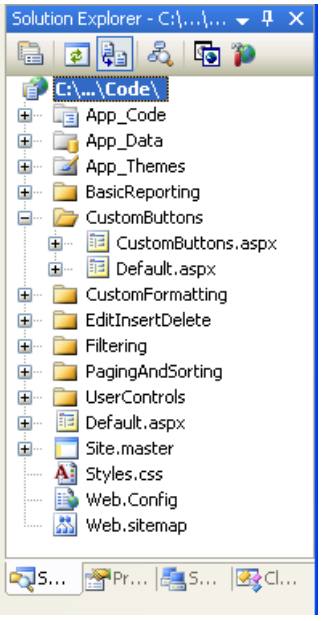


Figure 2: Add the ASP.NET Pages for the Custom Buttons-Related Tutorials

Like in the other folders, `Default.aspx` in the `CustomButtons` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

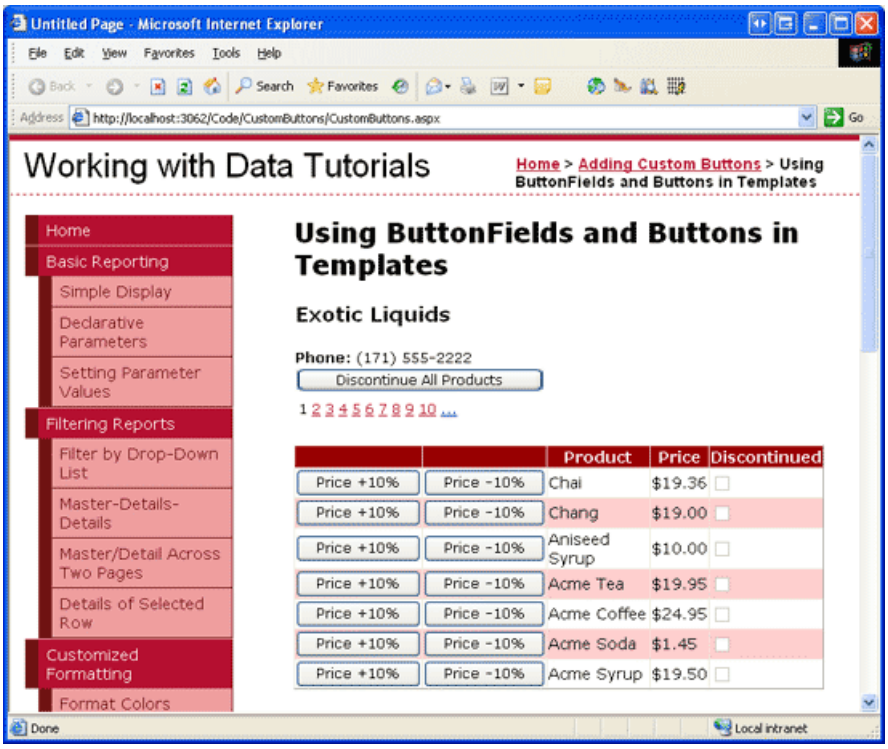


Figure 3: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add the pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Paging and Sorting `<siteMapNode>`:

```
<siteMapNode
    title="Adding Custom Buttons"
    description="Samples of Reports that Include Buttons for Performing
                 Server-Side Actions"
    url="~/CustomButtons/Default.aspx">
    <siteMapNode
        title="Using ButtonFields and Buttons in Templates"
        description="Examines how to add custom Buttons, LinkButtons,
                     or ImageButtons as ButtonFields or within templates."
        url="~/CustomButtons/CustomButtons.aspx" />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the editing, inserting, and deleting tutorials.

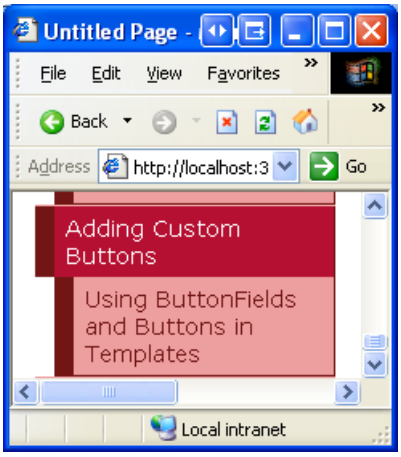


Figure 4: The Site Map Now Includes the Entry for the Custom Buttons Tutorial

Step 2: Adding a FormView that Lists the Suppliers

Let s get started with this tutorial by adding the FormView that lists the suppliers. As discussed in the Introduction, this FormView will allow the user to page through the suppliers, showing the products provided by the supplier in a GridView. Additionally, this FormView will include a Button that, when clicked, will mark all of the supplier s products as discontinued. Before we concern ourselves with adding the custom button to the FormView, let s first just create the FormView so that it displays the supplier information.

Start by opening the `CustomButtons.aspx` page in the `CustomButtons` folder. Add a FormView to the page by dragging it from the Toolbox onto the Designer and set its `ID` property to `Suppliers`. From the FormView s smart tag, opt to create a new ObjectDataSource named `SuppliersDataSource`.

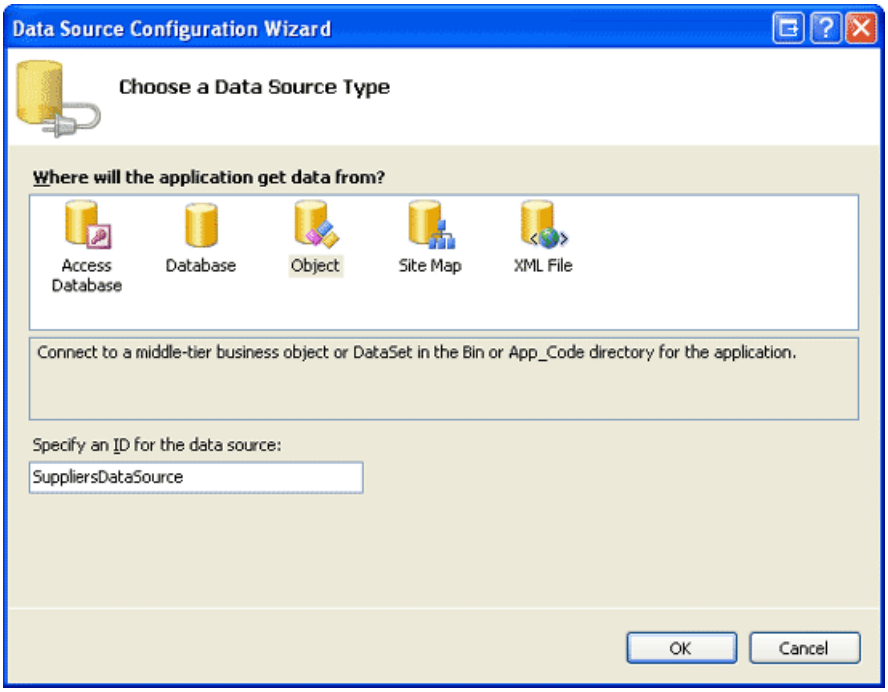


Figure 5: Create a New ObjectDataSource Named `SuppliersDataSource` (Click to view full-size image)

Configure this new ObjectDataSource such that it queries from the `SuppliersBLL` class s `GetSuppliers()` method (see Figure 6). Since this FormView does not provide an interface for updating the supplier information, select the (None) option from the drop-down list in the UPDATE tab.

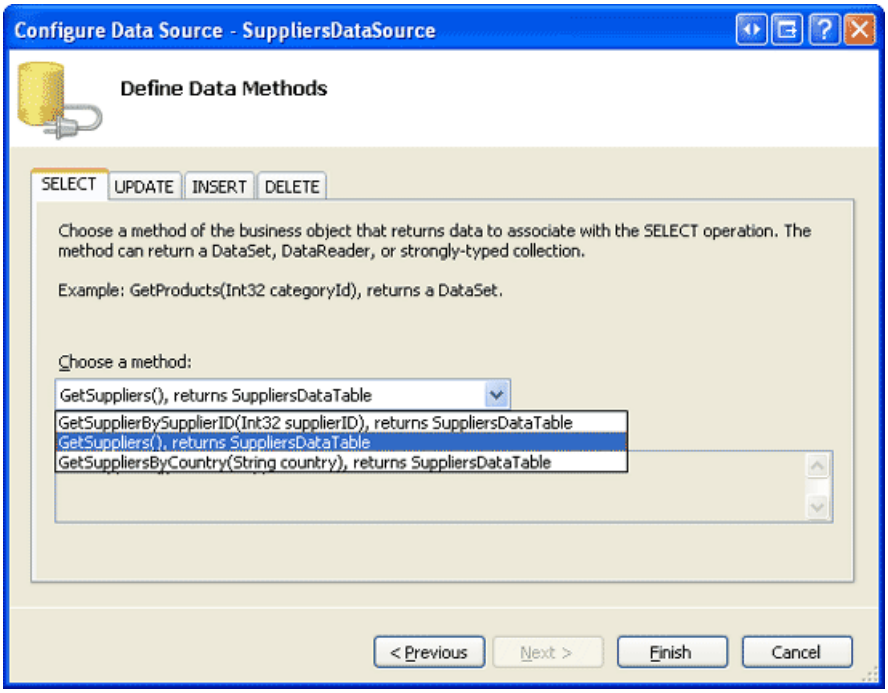


Figure 6: Configure the Data Source to use the `SuppliersBLL` Class s `GetSuppliers()` Method (Click to view full-size image)

After configuring the ObjectDataSource, Visual Studio will generate an `InsertItemTemplate`, `EditItemTemplate`, and `ItemTemplate` for the FormView. Remove the `InsertItemTemplate` and `EditItemTemplate` and modify the `ItemTemplate` so that it displays just the supplier s company name and phone number. Finally, turn on paging support for the FormView by checking the Enable Paging checkbox from its smart tag (or by setting its `AllowPaging` property to `True`). After these changes your page s declarative markup should look similar to the following:

```
<asp:FormView ID="Suppliers" runat="server" DataKeyNames="SupplierID"
    DataSourceID="SuppliersDataSource" EnableViewState="False" AllowPaging="True">
    <ItemTemplate>
        <h3>
            <asp:Label ID="CompanyName" runat="server"
                Text='<%# Bind("CompanyName") %>' />
        </h3>
        <b>Phone:</b>
        <asp:Label ID="PhoneLabel" runat="server" Text='<%# Bind("Phone") %>' />
    </ItemTemplate>
</asp:FormView>
<asp:ObjectDataSource ID="SuppliersDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetSuppliers" TypeName="SuppliersBLL">
</asp:ObjectDataSource>
```

Figure 7 shows the CustomButtons.aspx page when viewed through a browser.

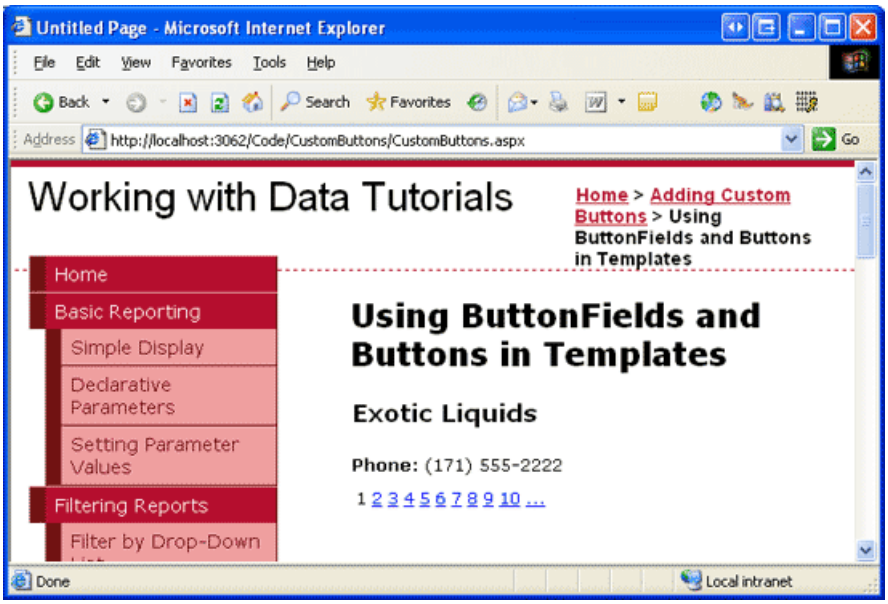


Figure 7: The FormView Lists the `CompanyName` and `Phone` Fields from the Currently Selected Supplier (Click to view full-size image)

Step 3: Adding a GridView that Lists the Selected Supplier s Products

Before we add the Discontinue All Products Button to the FormView s template, let s first add a GridView beneath the FormView that lists the products provided by the selected supplier. To accomplish this, add a GridView to the page, set its `ID` property to `SuppliersProducts`, and add a new ObjectDataSource named `SuppliersProductsDataSource`.

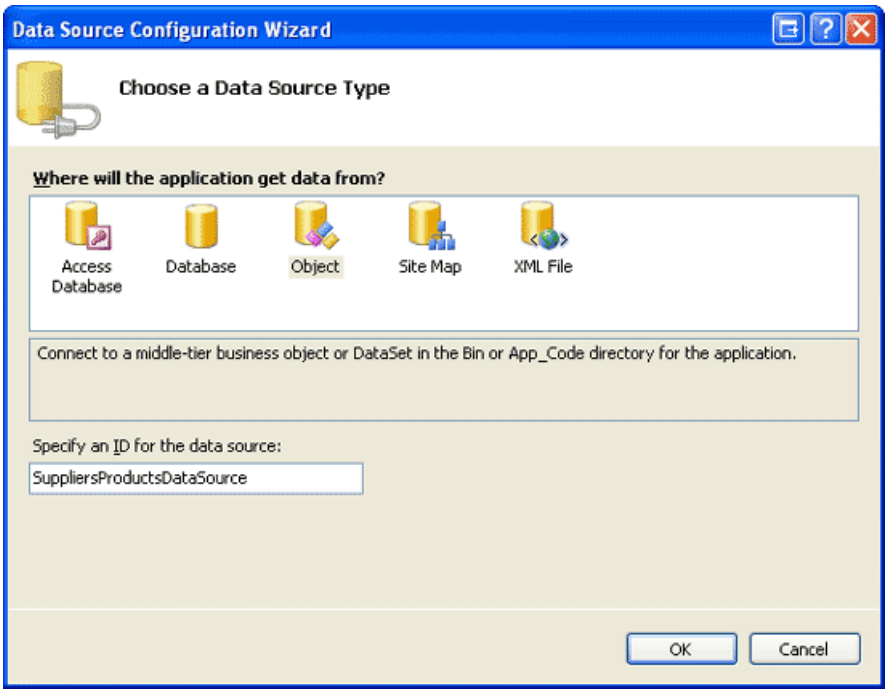


Figure 8: Create a New ObjectDataSource Named `SuppliersProductsDataSource` (Click to view full-size image)

Configure this ObjectDataSource to use the ProductsBLL class s `GetProductsBySupplierID(supplierID)` method (see Figure 9). While this GridView will allow for a product s price to be adjusted, it won't be using the built-in editing or deleting features from the GridView. Therefore, we can set the drop-down list to (None) for the ObjectDataSource s UPDATE, INSERT, and DELETE tabs.

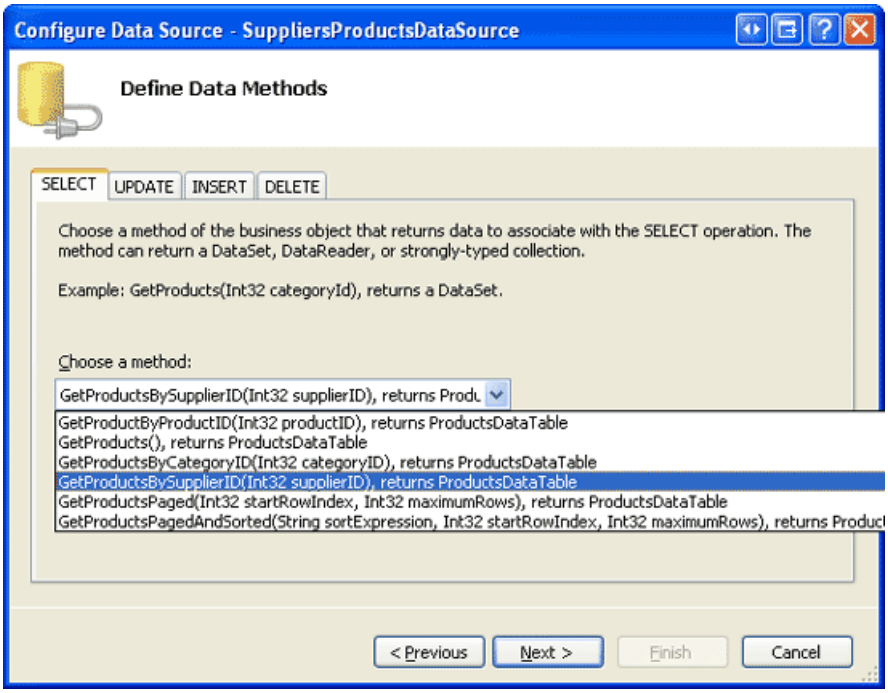


Figure 9: Configure the Data Source to use the `ProductsBLL` Class s `GetProductsBySupplierID(supplierID)` Method (Click to view full-size image)

Since the `GetProductsBySupplierID(supplierID)` method accepts an input parameter, the ObjectDataSource wizard prompts us for the source of this parameter value. To pass in the `SupplierID` value from the FormView, set the Parameter source drop-down list to Control and the ControlID drop-down list to `Suppliers` (the ID of the FormView created in Step 2).

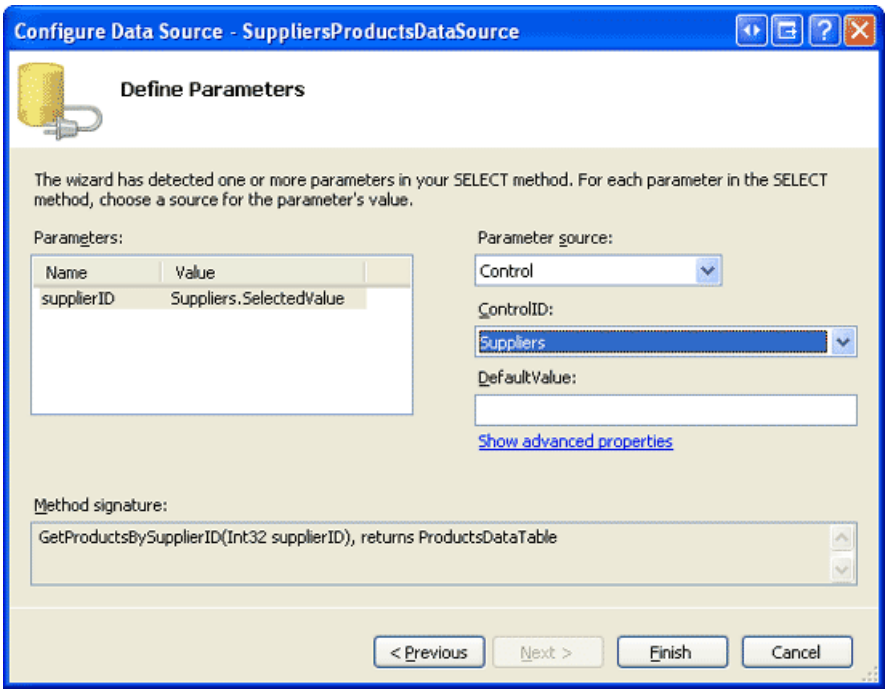


Figure 10: Indicate that the `supplierID` Parameter Should Come from the `Suppliers` FormView Control (Click to view full-size image)

After completing the ObjectDataSource wizard, the GridView will contain a BoundField or CheckBoxField for each of the product s data fields. Let s trim this down to show just the `ProductName` and `UnitPrice` BoundFields along with the `Discontinued` CheckBoxField; furthermore, let s format the `UnitPrice` BoundField such that its text is formatted as a currency. Your GridView and `SuppliersProductsDataSource` ObjectDataSource s declarative markup should look similar to the following markup:

```
<asp:GridView ID="SuppliersProducts" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="SuppliersProductsDataSource"
    EnableViewState="False" runat="server">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
            SortExpression="UnitPrice" DataFormatString="{0:C}"
            HtmlEncode="False" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="SuppliersProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProductsBySupplierID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:ControlParameter ControlID="Suppliers" Name="supplierID"
            PropertyName="SelectedValue" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

At this point our tutorial displays a master/details report, allowing the user to pick a supplier from the FormView at the top and to view the products provided by that supplier through the GridView at the bottom. Figure 11 shows a screen shot of this page when selecting the Tokyo Traders supplier from the FormView.

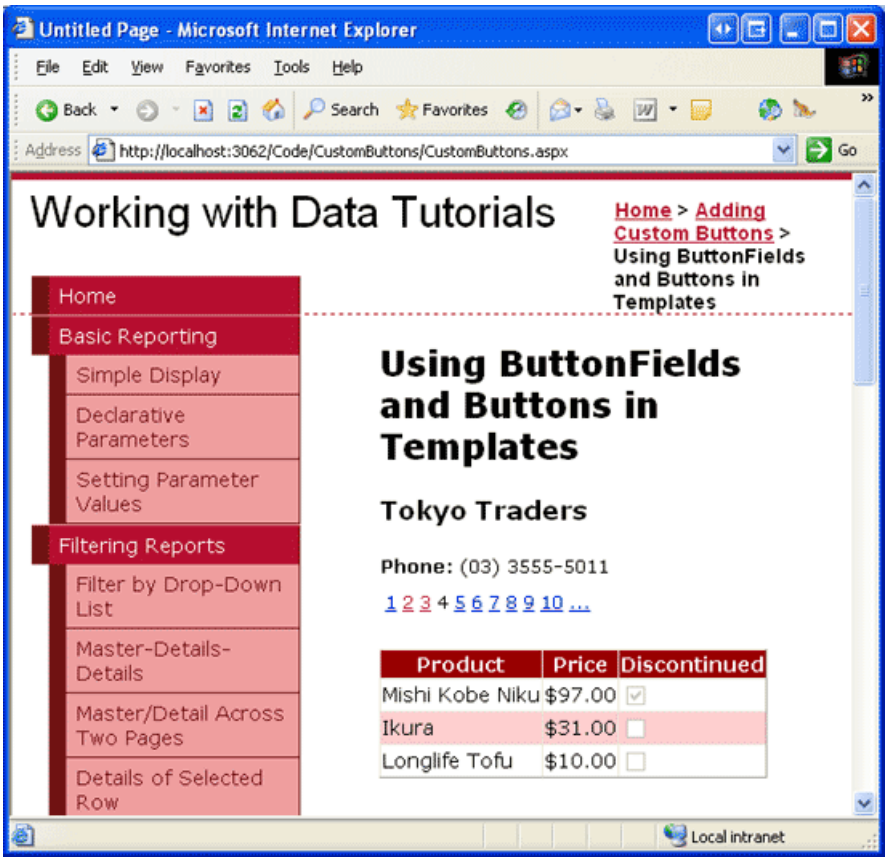


Figure 11: The Selected Supplier s Products are Displayed in the GridView (Click to view full-size image)

Step 4: Creating DAL and BLL Methods to Discontinue All Products for a Supplier

Before we can add a Button to the FormView that, when clicked, discontinues all of the supplier s products, we first need to add a method to both the DAL and BLL that performs this action. In particular, this method will be named `DiscontinueAllProductsForSupplier(supplierID)`. When the FormView s Button is clicked, we'll invoke this method in the Business Logic Layer, passing in the selected supplier s `SupplierID`; the BLL will then call down to the corresponding Data Access Layer method, which will issue an `UPDATE` statement to the database that discontinues the specified supplier s products.

As we have done in our previous tutorials, we'll use a bottom-up approach, starting with creating the DAL method, then the BLL method, and finally implementing the functionality in the ASP.NET page. Open the `Northwind.xsd` Typed DataSet in the `App_Code/DAL` folder and add a new method to the `ProductsTableAdapter` (right-click on the `ProductsTableAdapter` and choose Add Query). Doing so will bring up the TableAdapter Query Configuration wizard, which walks us through the process of adding the new method. Start by indicating that our DAL method will use an ad-hoc SQL statement.

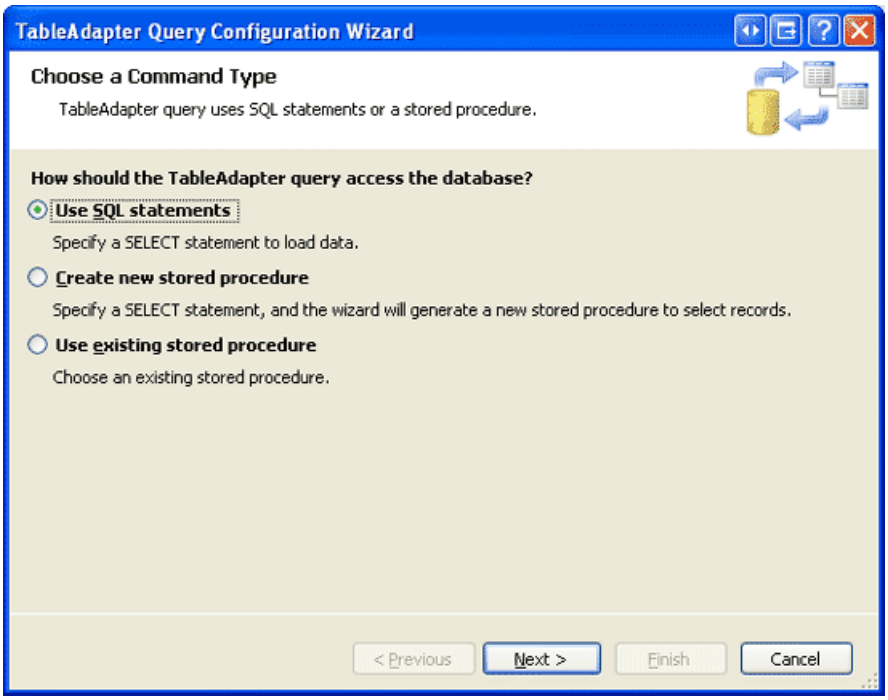


Figure 12: Create the DAL Method Using an Ad-Hoc SQL Statement (Click to view full-size image)

Next, the wizard prompts us as to what type of query to create. Since the `DiscontinueAllProductsForSupplier(supplierID)` method will need to update the `Products` database table, setting the `Discontinued` field to 1 for all products provided by the specified *`supplierID`*, we need to create a query that updates data.

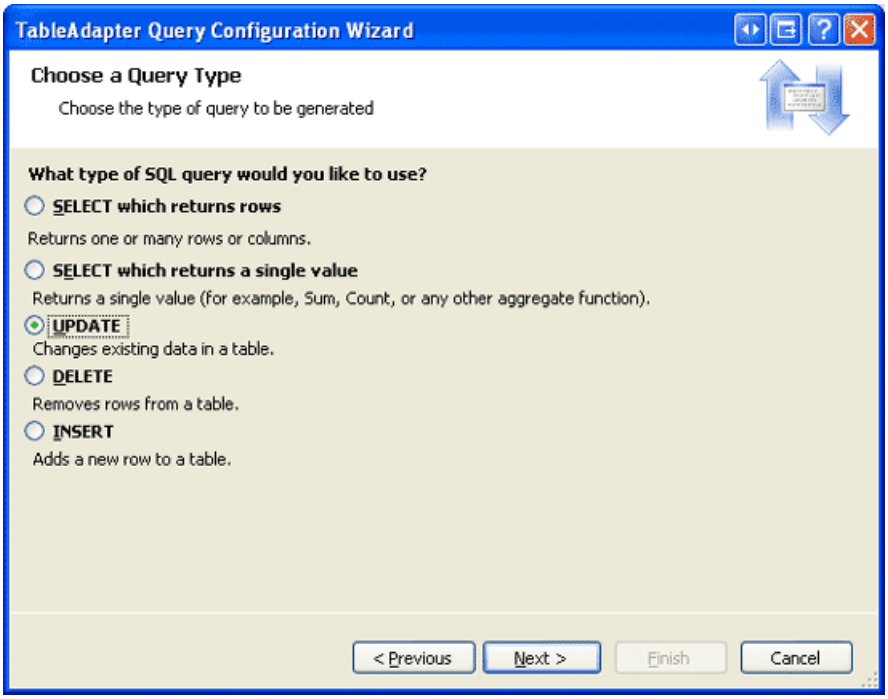


Figure 13: Choose the UPDATE Query Type (Click to view full-size image)

The next wizard screen provides the TableAdapter s existing `UPDATE` statement, which updates each of the fields defined in the `Products` DataTable. Replace this query text with the following statement:

```
UPDATE [Products] SET
   Discontinued = 1
WHERE SupplierID = @SupplierID
```

After entering this query and clicking Next, the last wizard screen asks for the new method s name use

`DiscontinueAllProductsForSupplier`. Complete the wizard by clicking the Finish button. Upon returning to the DataSet Designer you should see a new method in the `ProductsTableAdapter` named `DiscontinueAllProductsForSupplier(@SupplierID)`.

Figure 14: Name the New DAL Method `DiscontinueAllProductsForSupplier` (Click to view full-size image)

With the `DiscontinueAllProductsForSupplier(supplierID)` method created in the Data Access Layer, our next task is to create the `DiscontinueAllProductsForSupplier(supplierID)` method in the Business Logic Layer. To accomplish this, open the `ProductsBLL` class file and add the following:

```
Public Function DiscontinueAllProductsForSupplier(supplierID As Integer) As Integer
    Return Adapter.DiscontinueAllProductsForSupplier(supplierID)
End Function
```

This method simply calls down to the `DiscontinueAllProductsForSupplier(supplierID)` method in the DAL, passing along the provided *`supplierID`* parameter value. If there were any business rules that only allowed a supplier s products to be discontinued under certain circumstances, those rules should be implemented here, in the BLL.

> **NOTE**
> Unlike the `UpdateProduct` overloads in the `ProductsBLL` class, the `DiscontinueAllProductsForSupplier(supplierID)` method signature does not include the `DataObjectMethodAttribute` attribute (
> `<System.ComponentModel.DataObjectMethodAttribute(System.ComponentModel.DataObjectMethodType.Update, Boolean)>`
>). This precludes the `DiscontinueAllProductsForSupplier(supplierID)` method from the ObjectDataSource s Configure Data Source wizard s drop-down list in the UPDATE tab. I ve omitted this attribute because we'll be calling the `DiscontinueAllProductsForSupplier(supplierID)` method directly from an event handler in our ASP.NET page.

Step 5: Adding a Discontinue All Products Button to the FormView

With the `DiscontinueAllProductsForSupplier(supplierID)` method in the BLL and DAL complete, the final step for adding the ability to discontinue all products for the selected supplier is to add a Button Web control to the FormView s `ItemTemplate`. Let s add such a Button below the supplier s phone number with the button text, Discontinue All Products and an `ID` property value of `DiscontinueAllProductsForSupplier`. You can add this Button Web control through the Designer by clicking on the Edit Templates link in the FormView s smart tag (see

Figure 15), or directly through the declarative syntax.

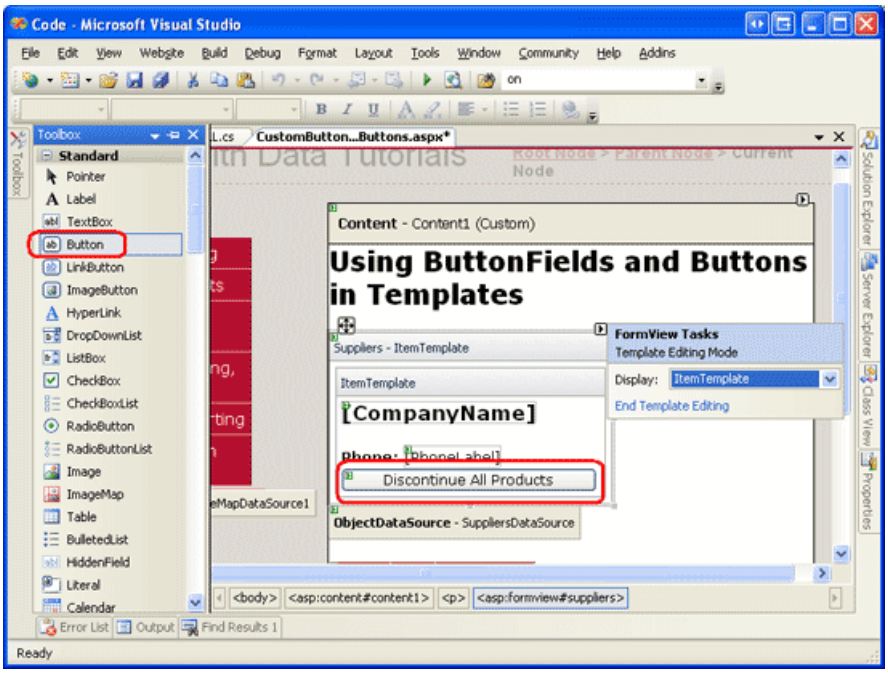


Figure 15: Add a Discontinue All Products Button Web Control to the FormView s `ItemTemplate` (Click to view full-size image)

When the Button is clicked by a user visiting the page, a postback ensues and the FormView s `ItemCommand` event fires. To execute custom code in response to this Button being clicked, we can create an event handler for this event. Understand, though, that the `ItemCommand` event fires whenever *any* Button, LinkButton, or ImageButton Web control is clicked within the FormView. That means that when the user moves from one page to another in the FormView, the `ItemCommand` event fires; same thing when the user clicks New, Edit, or Delete in a FormView that supports inserting, updating, or deleting.

Since the `ItemCommand` fires regardless of what button is clicked, in the event handler we need a way to determine if the Discontinue All Products Button was clicked or if it was some other button. To accomplish this, we can set the Button Web control s `CommandName` property to some identifying value. When the Button is clicked, this `CommandName` value is passed into the `ItemCommand` event handler, enabling us to determine whether the Discontinue All Products Button was the button clicked. Set the Discontinue All Products Button s `CommandName` property to DiscontinueProducts .

Finally, let s use a client-side confirm dialog box to ensure that the user really wants to discontinue the selected supplier s products. As we saw in the Adding Client-Side Confirmation When Deleting tutorial, this can be accomplished with a bit of JavaScript. In particular, set the Button Web control s OnClientClick property to

```
return confirm('This will mark _all_ of this supplier\'s products as discontinued. Are you certain you want to do this?');
```

After making these changes, the FormView s declarative syntax should look like the following:

```
<asp:FormView ID="Suppliers" runat="server" DataKeyNames="SupplierID"
    DataSourceID="SuppliersDataSource" EnableViewState="False"
    AllowPaging="True">
    <ItemTemplate>
        <h3><asp:Label ID="CompanyName" runat="server"
            Text='<%# Bind("CompanyName") %>'></asp:Label></h3>
        <b>Phone:</b>
        <asp:Label ID="PhoneLabel" runat="server" Text='<%# Bind("Phone") %>' />
        <br />
        <asp:Button ID="DiscontinueAllProductsForSupplier" runat="server"
            CommandName="DiscontinueProducts" Text="Discontinue All Products"
            OnClientClick="return confirm('This will mark _all_ of this supplier\'s
                products as discontinued. Are you certain you want to do this?');" />
    </ItemTemplate>
</asp:FormView>
```

Next, create an event handler for the FormView s `ItemCommand` event. In this event handler we need to first determine whether the Discontinue All Products Button was clicked. If so, we want to create an instance of the `ProductsBLL` class and invoke its `DiscontinueAllProductsForSupplier(supplierID)` method, passing in the `SupplierID` of the selected FormView:

```
Protected Sub Suppliers_ItemCommand(sender As Object, e As FormViewCommandEventArgs) _
    Handles Suppliers.ItemCommand
    If e.CommandName.CompareTo("DiscontinueProducts") = 0 Then
        ' The "Discontinue All Products" Button was clicked.
        ' Invoke the ProductsBLL.DiscontinueAllProductsForSupplier(supplierID) method
        ' First, get the SupplierID selected in the FormView
        Dim supplierID As Integer = CType(Suppliers.SelectedValue, Integer)
        ' Next, create an instance of the ProductsBLL class
        Dim productInfo As New ProductsBLL()
        ' Finally, invoke the DiscontinueAllProductsForSupplier(supplierID) method
        productInfo.DiscontinueAllProductsForSupplier(supplierID)
    End If
End Sub
```

Note that the `SupplierID` of the current selected supplier in the FormView can be accessed using the FormView s `SelectedValue` property. The `SelectedValue` property returns the first data key value for the record being displayed in the FormView. The FormView s `DataKeyNames` property, which indicates the data fields from which the data key values are pulled from, was automatically set to `SupplierID` by Visual Studio when binding the ObjectDataSource to the FormView back in Step 2.

With the `ItemCommand` event handler created, take a moment to test out the page. Browse to the Cooperativa de Quesos 'Las Cabras' supplier (it s the fifth supplier in the FormView for me). This supplier provides two products, Queso Cabrales and Queso Manchego La Pastora, both of which are *not* discontinued.

Imagine that Cooperativa de Quesos 'Las Cabras' has gone out of business and therefore its products are to be discontinued. Click the Discontinue All Products Button. This will display the client-side confirm dialog box (see Figure 16).

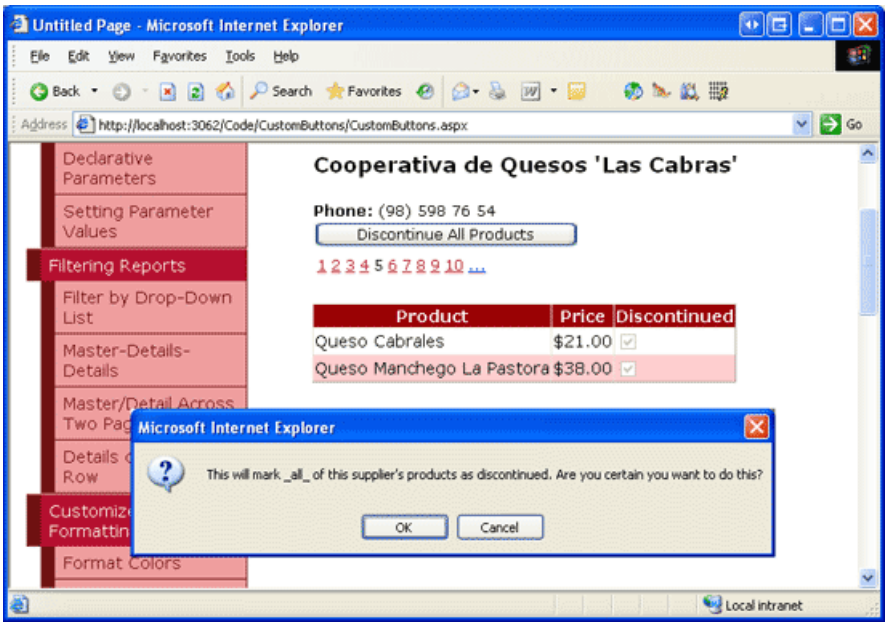


Figure 16: Cooperativa de Quesos Las Cabras Supplies Two Active Products (Click to view full-size image)

If you click OK in the client-side confirm dialog box, the form submission will proceed, causing a postback in which the FormView s `ItemCommand` event will fire. The event handler we created will then execute, invoking the `DiscontinueAllProductsForSupplier(supplierID)` method and discontinuing both the Queso Cabrales and Queso Manchego La Pastora products.

If you have disabled the GridView s view state, the GridView is being rebound to the underlying data store on every postback, and therefore will immediately be updated to reflect that these two products are now discontinued (see Figure 17). If, however, you have not disabled view state in the GridView, you will need to manually rebind the data to the GridView after making this change. To accomplish this, simply make a call to the GridView s `DataBind()` method immediately after invoking the `DiscontinueAllProductsForSupplier(supplierID)` method.

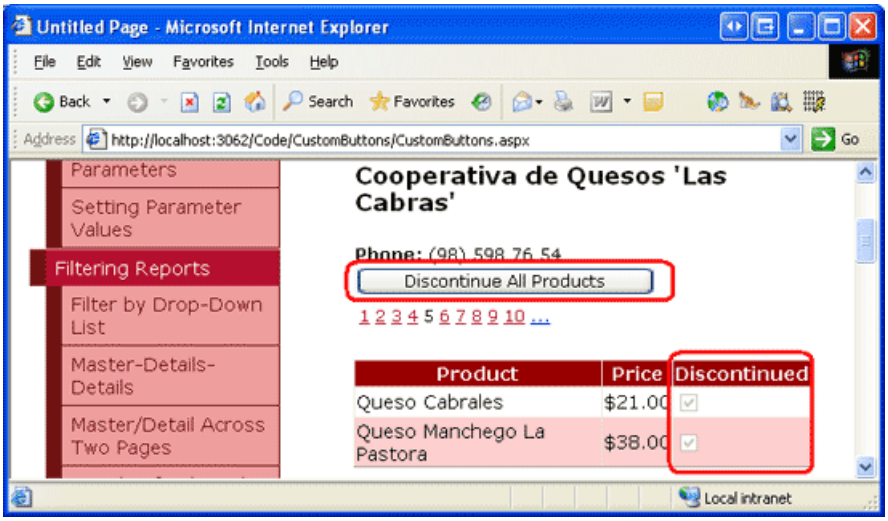


Figure 17: After Clicking the Discontinue All Products Button, the Supplier s Products are Updated Accordingly (Click to view full-size image)

Step 6: Creating an UpdateProduct Overload in the Business Logic Layer for Adjusting a Product s Price

Like with the Discontinue All Products Button in the FormView, in order to add buttons for increasing and decreasing the price for a product in the GridView we need to first add the appropriate Data Access Layer and Business Logic Layer methods. Since we already have a method that updates a single product row in the DAL, we

can provide such functionality by creating a new overload for the `UpdateProduct` method in the BLL.

Our past `UpdateProduct` overloads have taken in some combination of product fields as scalar input values and have then updated just those fields for the specified product. For this overload we'll vary slightly from this standard and instead pass in the product s `ProductID` and the percentage by which to adjust the `UnitPrice` (as opposed to passing in the new, adjusted `UnitPrice` itself). This approach will simplify the code we need to write in the ASP.NET page code-behind class, since we don t have to bother with determining the current product s `UnitPrice`.

The `UpdateProduct` overload for this tutorial is shown below:

```
Public Function UpdateProduct _
    (unitPriceAdjustmentPercentage As Decimal, productID As Integer) As Boolean
    Dim products As Northwind.ProductsDataTable = Adapter.GetProductByProductID(productID)
    If products.Count = 0 Then
        ' no matching record found, return false
        Return False
    End If
    Dim product As Northwind.ProductsRow = products(0)
    ' Adjust the UnitPrice by the specified percentage (if it's not NULL)
    If Not product.IsUnitPriceNull() Then
        product.UnitPrice *= unitPriceAdjustmentPercentage
    End If
    ' Update the product record
    Dim rowsAffected As Integer = Adapter.Update(product)
    ' Return true if precisely one row was updated, otherwise false
    Return rowsAffected = 1
End Function
```

This overload retrieves information about the specified product through the DAL s `GetProductByProductID(productID)` method. It then checks to see whether the product s `UnitPrice` is assigned a database `NULL` value. If it is, the price is left unaltered. If, however, there is a non- `NULL` `UnitPrice` value, the method updates the product s `UnitPrice` by the specified percent (`unitPriceAdjustmentPercent`).

Step 7: Adding the Increase and Decrease Buttons to the GridView

The GridView (and DetailsView) are both made up of a collection of fields. In addition to BoundFields, CheckBoxFields, and TemplateFields, ASP.NET includes the ButtonField, which, as its name implies, renders as a column with a Button, LinkButton, or ImageButton for each row. Similar to the FormView, clicking *any* button within the GridView paging buttons, Edit or Delete buttons, sorting buttons, and so on causes a postback and raises the GridView s `RowCommand` event.

The ButtonField has a `CommandName` property that assigns the specified value to each of its Buttons `CommandName` properties. Like with the FormView, the `CommandName` value is used by the `RowCommand` event handler to determine which button was clicked.

Let s add two new ButtonFields to the GridView, one with a button text Price +10% and the other with the text Price -10%. To add these ButtonFields, click on the Edit Columns link from the GridView s smart tag, select the ButtonField field type from the list in the upper left and click the Add button.

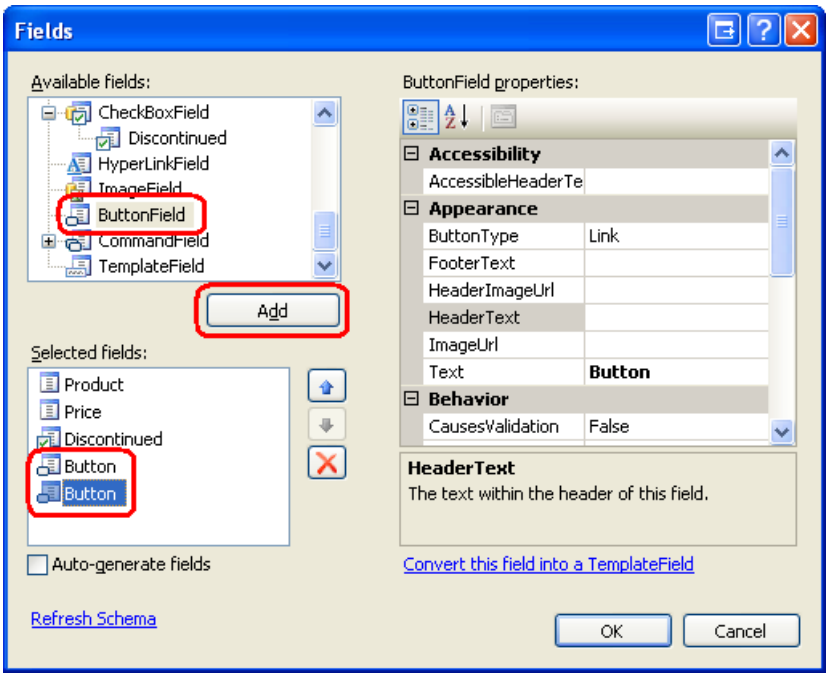


Figure 18: Add Two ButtonFields to the GridView

Move the two ButtonFields so that they appear as the first two GridView fields. Next, set the `Text` properties of these two ButtonFields to Price +10% and Price -10% and the `CommandName` properties to IncreasePrice and DecreasePrice, respectively. By default, a ButtonField renders its column of buttons as LinkButtons. This can be changed, however, through the ButtonField s `ButtonType` property. Let s have these two ButtonFields rendered as regular push buttons; therefore, set the `ButtonType` property to `Button`. Figure 19 shows the Fields dialog box after these changes have been made; following that is the GridView s declarative markup.

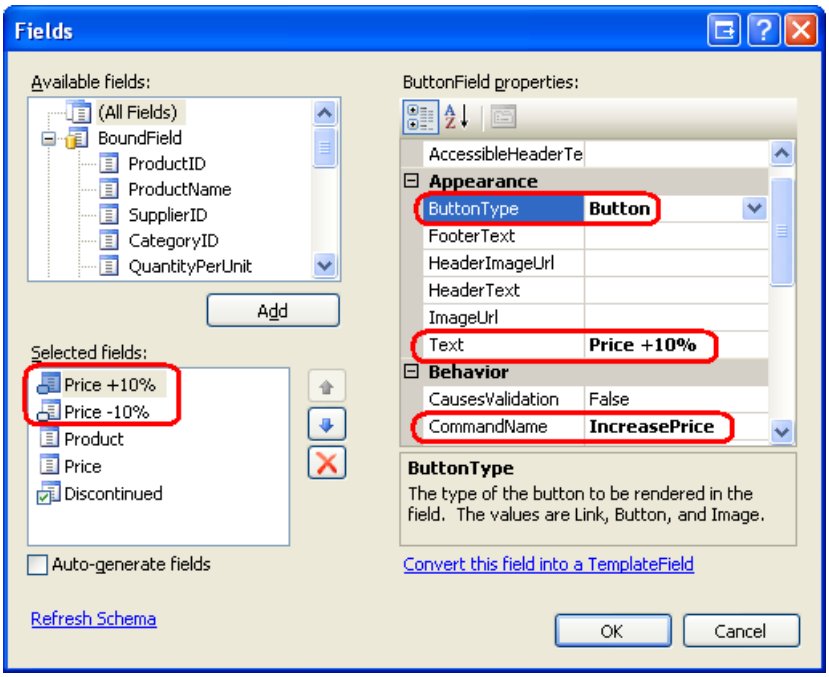


Figure 19: Configure the ButtonFields `Text`, `CommandName`, and `ButtonType` Properties

```
<asp:GridView ID="SuppliersProducts" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="SuppliersProductsDataSource"
    EnableViewState="False">
    <Columns>
        <asp:ButtonField ButtonType="Button" CommandName="IncreasePrice"
            Text="Price +10%" />
        <asp:ButtonField ButtonType="Button" CommandName="DecreasePrice"
            Text="Price -10%" />
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
            SortExpression="UnitPrice" DataFormatString="{0:C}"
            HtmlEncode="False" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
```

With these ButtonFields created, the final step is to create an event handler for the GridView s `RowCommand` event. This event handler, if fired because either the Price +10% or Price -10% buttons were clicked, needs to determine the `ProductID` for the row whose button was clicked and then invoke the `ProductsBLL` class s `UpdateProduct` method, passing in the appropriate `UnitPrice` percentage adjustment along with the `ProductID`. The following code performs these tasks:

```
Protected Sub SuppliersProducts_RowCommand _
    (sender As Object, e As GridViewCommandEventArgs) _
        Handles SuppliersProducts.RowCommand
    If e.CommandName.CompareTo("IncreasePrice") = 0 OrElse _
       e.CommandName.CompareTo("DecreasePrice") = 0 Then
        ' The Increase Price or Decrease Price Button has been clicked
        ' Determine the ID of the product whose price was adjusted
        Dim productID As Integer = Convert.ToInt32( _
            SuppliersProducts.DataKeys(Convert.ToInt32(e.CommandArgument)).Value)
        ' Determine how much to adjust the price
        Dim percentageAdjust As Decimal
        If e.CommandName.CompareTo("IncreasePrice") = 0 Then
            percentageAdjust = 1.1
        Else
            percentageAdjust = 0.9
        End If
        ' Adjust the price
        Dim productInfo As New ProductsBLL()
        productInfo.UpdateProduct(percentageAdjust, productID)
    End If
End Sub
```

In order to determine the `ProductID` for the row whose Price +10% or Price -10% button was clicked, we need to consult the GridView s `DataKeys` collection. This collection holds the values of the fields specified in the `DataKeyNames` property for each GridView row. Since the GridView s `DataKeyNames` property was set to ProductID by Visual Studio when binding the ObjectDataSource to the GridView, `DataKeys(rowIndex).Value` provides the `ProductID` for the specified *rowIndex*.

The ButtonField automatically passes in the *rowIndex* of the row whose button was clicked through the `e.CommandArgument` parameter. Therefore, to determine the `ProductID` for the row whose Price +10% or Price -10% button was clicked, we use: `Convert.ToInt32(SuppliersProducts.DataKeys(Convert.ToInt32(e.CommandArgument)).Value)`.

As with the Discontinue All Products button, if you have disabled the GridView s view state, the GridView is being rebound to the underlying data store on every postback, and therefore will immediately be updated to reflect a price change that occurs from clicking either of the buttons. If, however, you have not disabled view state in the

GridView, you will need to manually rebind the data to the GridView after making this change. To accomplish this, simply make a call to the GridView s `DataBind()` method immediately after invoking the `UpdateProduct` method.

Figure 20 shows the page when viewing the products provided by Grandma Kelly's Homestead. Figure 21 shows the results after the Price +10% button has been clicked twice for Grandma's Boysenberry Spread and the Price -10% button once for Northwoods Cranberry Sauce.

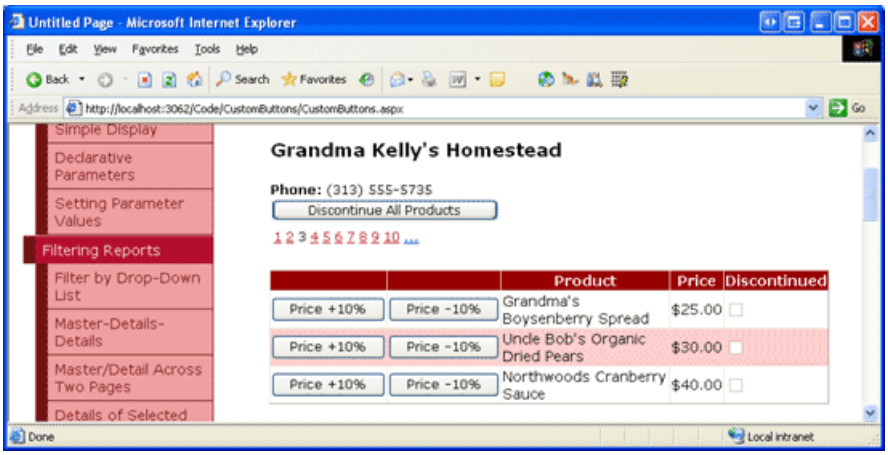


Figure 20: The GridView Includes Price +10% and Price -10% Buttons (Click to view full-size image)

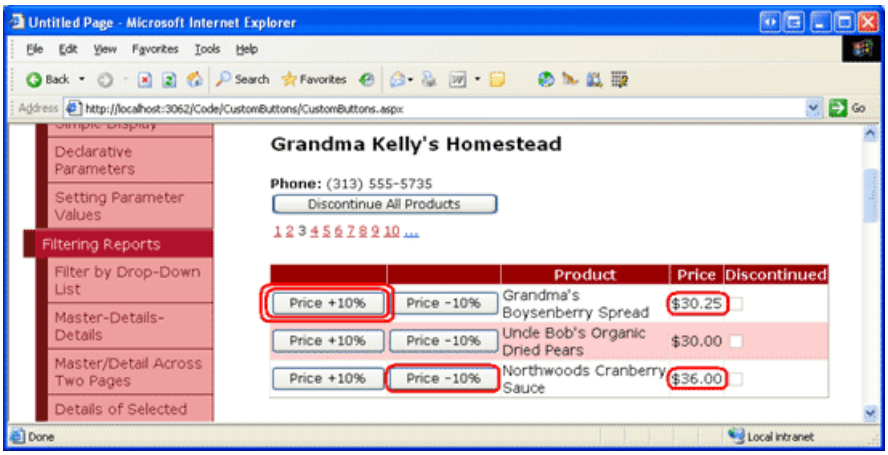


Figure 21: The Prices for the First and Third Product Have Been Updated via the Price +10% and Price -10% Buttons (Click to view full-size image)

> **NOTE**
>
> The GridView (and DetailsView) can also have Buttons, LinkButtons, or ImageButtons added to their TemplateFields. As with the BoundField, these Buttons, when clicked, will induce a postback, raising the GridView s `RowCommand` event. When adding buttons in a TemplateField, however, the Button s `CommandArgument` is not automatically set to the index of the row as it is when using ButtonFields. If you need to determine the row index of the button that was clicked within the `RowCommand` event handler, you'll need to manually set the Button s `CommandArgument` property in its declarative syntax within the TemplateField, using code like:
>
> `<asp:Button runat="server" ... CommandArgument='<%# CType(Container, GridViewRow).RowIndex %>' />`.

Summary

The GridView, DetailsView, and FormView controls all can include Buttons, LinkButtons, or ImageButtons. Such buttons, when clicked, cause a postback and raise the `ItemCommand` event in the FormView and DetailsView controls and the `RowCommand` event in the GridView. These data Web controls have built-in functionality to handle common command-related actions, such as deleting or editing records. However, we can also use buttons that, when clicked, respond with executing our own custom code.

To accomplish this, we need to create an event handler for the `ItemCommand` or `RowCommand` event. In this event handler we first check the incoming `CommandName` value to determine which button was clicked and then take appropriate custom action. In this tutorial we saw how to use buttons and ButtonFields to discontinue all products for a specified supplier or to increase or decrease the price of a particular product by 10%.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS

Displaying Data with the DataList and Repeater

2/20/2020 • 2 minutes to read • Edit Online

These tutorials show alternative ways to display data by using the DataList and Repeater controls.

- Displaying Data with the DataList and Repeater Controls (C#)
- Formatting the DataList and Repeater Based Upon Data (C#)
- Showing Multiple Records per Row with the DataList Control (C#)
- Nested Data Web Controls (C#)
- Displaying Data with the DataList and Repeater Controls (VB)
- Formatting the DataList and Repeater Based Upon Data (VB)
- Showing Multiple Records per Row with the DataList Control (VB)
- Nested Data Web Controls (VB)

Displaying Data with the DataList and Repeater Controls (C#)

2/20/2020 • 19 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In the preceding tutorials we have used the GridView control to display data. Starting with this tutorial, we look at building common reporting patterns with the DataList and Repeater controls, starting with the basics of displaying data with these controls.

Introduction

In all of the examples throughout the past 28 tutorials, if we needed to display multiple records from a data source we turned to the GridView control. The GridView renders a row for each record in the data source, displaying the record s data fields in columns. While the GridView makes it a snap to display, page through, sort, edit, and delete data, its appearance is a bit boxy. Moreover, the markup responsible for the GridView s structure is fixed it includes an HTML `<table>` with a table row (`<tr>`) for each record and a table cell (`<td>`) for each field.

To provide a greater degree of customization in the appearance and rendered markup when displaying multiple records, ASP.NET 2.0 offers the DataList and Repeater controls (both of which were also available in ASP.NET version 1.x). The DataList and Repeater controls render their content using templates rather than BoundFields, CheckBoxFields, ButtonFields, and so on. Like the GridView, the DataList renders as an HTML `<table>`, but allows for multiple data source records to be displayed per table row. The Repeater, on the other hand, renders no additional markup than what you explicitly specify, and is an ideal candidate when you need precise control over the markup emitted.

Over the next dozen or so tutorials, we'll look at building common reporting patterns with the DataList and Repeater controls, starting with the basics of displaying data with these controls templates. We'll see how to format these controls, how to alter the layout of data source records in the DataList, common master/details scenarios, ways to edit and delete data, how to page through records, and so on.

Step 1: Adding the DataList and Repeater Tutorial Web Pages

Before we start this tutorial, let s first take a moment to add the ASP.NET pages we'll need for this tutorial and the next few tutorials dealing with displaying data using the DataList and Repeater. Start by creating a new folder in the project named `DataListRepeaterBasics`. Next, add the following five ASP.NET pages to this folder, having all of them configured to use the master page `Site.master`:

- `Default.aspx`
- `Basics.aspx`
- `Formatting.aspx`
- `RepeatColumnAndDirection.aspx`
- `NestedControls.aspx`

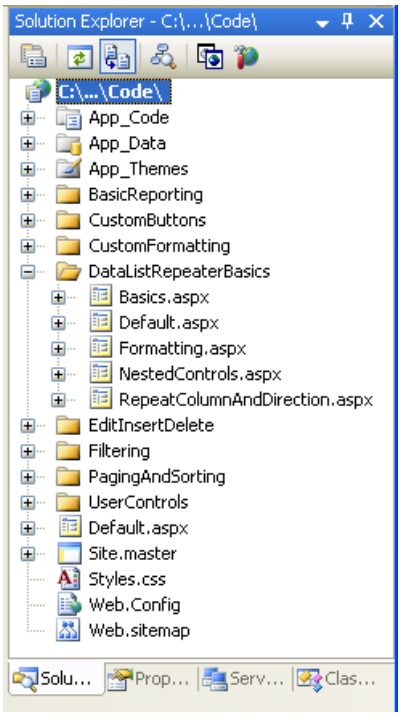


Figure 1: Create a `DataListRepeaterBasics` Folder and Add the Tutorial ASP.NET Pages

Open the `Default.aspx` page and drag the `SectionLevelTutorialListing.ascx` User Control from the `UserControls` folder onto the Design surface. This User Control, which we created in the Master Pages and Site Navigation tutorial, enumerates the site map and displays the tutorials from the current section in a bulleted list.

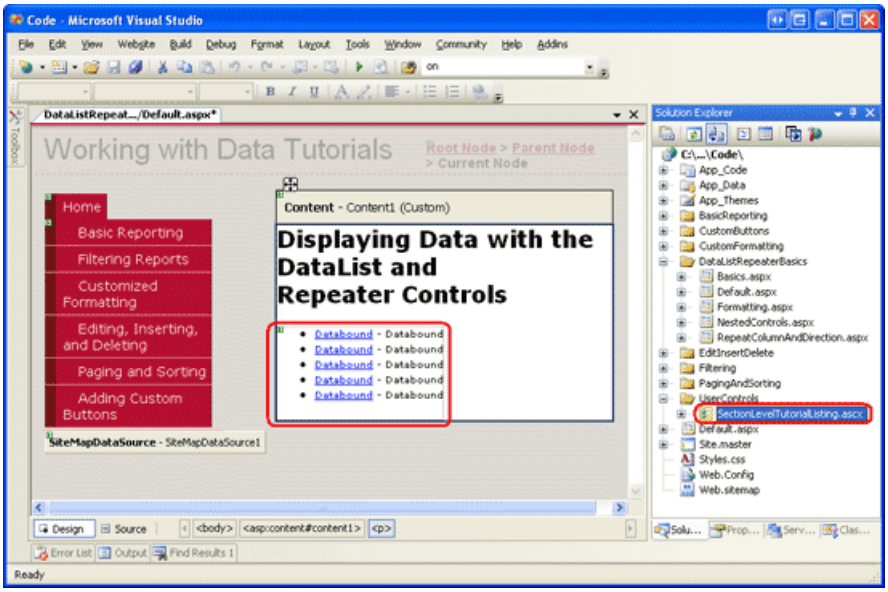


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

In order to have the bulleted list display the DataList and Repeater tutorials we'll be creating, we need to add them to the site map. Open the `Web.sitemap` file and add the following markup after the Adding Custom Buttons site map node markup:

```
<siteMapNode
    title="Displaying Data with the DataList and Repeater"
    description="Samples of Reports that Use the DataList and Repeater Controls"
    url="~/DataListRepeaterBasics/Default.aspx" >
    <siteMapNode
        title="Basic Examples"
        description="Examines the basics for displaying data using the
                     DataList and Repeater controls."
        url="~/DataListRepeaterBasics/Basics.aspx"  />
    <siteMapNode
        title="Formatting"
        description="Learn how to format the DataList and the Web controls within
                     the DataList and Repeater's templates."
        url="~/DataListRepeaterBasics/Formatting.aspx" />
    <siteMapNode
        title="Adjusting the DataList s Layout"
        description="Illustrates how to alter the DataList's layout, showing
                     multiple data source records per table row."
        url="~/DataListRepeaterBasics/RepeatColumnAndDirection.aspx" />
    <siteMapNode
        title="Nesting a Repeater within a DataList"
        description="Learn how to nest a Repeater within the template of a DataList."
        url="~/DataListRepeaterBasics/NestedControls.aspx" />
</siteMapNode>
```

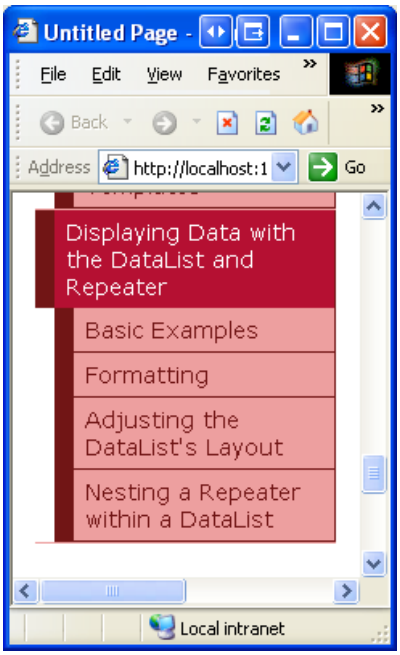


Figure 3: Update the Site Map to Include the New ASP.NET Pages

Step 2: Displaying Product Information with the DataList

Similar to the FormView, the DataList control s rendered output depends upon templates rather than BoundFields, CheckBoxFields, and so on. Unlike the FormView, the DataList is designed to display a set of records rather than a solitary one. Let s begin this tutorial with a look at binding product information to a DataList. Start by opening the `Basics.aspx` page in the `DataListRepeaterBasics` folder. Next, drag a DataList from the Toolbox onto the Designer. As Figure 4 illustrates, before specifying the DataList s templates, the Designer displays it as a gray box.

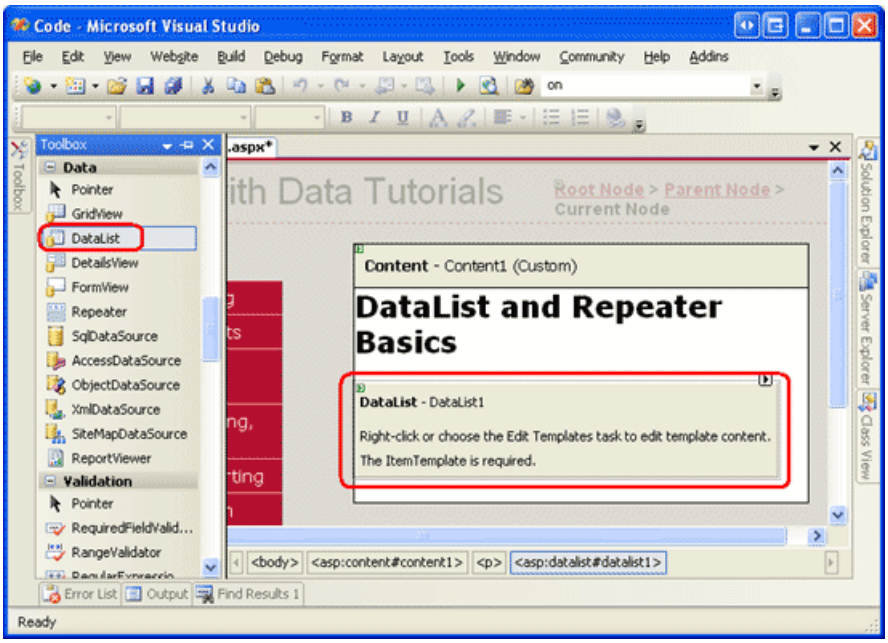


Figure 4: Drag the DataList From the Toolbox Onto the Designer (Click to view full-size image)

From the DataList s smart tag, add a new ObjectDataSource and configure it to use the `ProductsBLL` class s `GetProducts` method. Since we re creating a read-only DataList in this tutorial, set the drop-down list to (None) in the wizard s INSERT, UPDATE, and DELETE tabs.

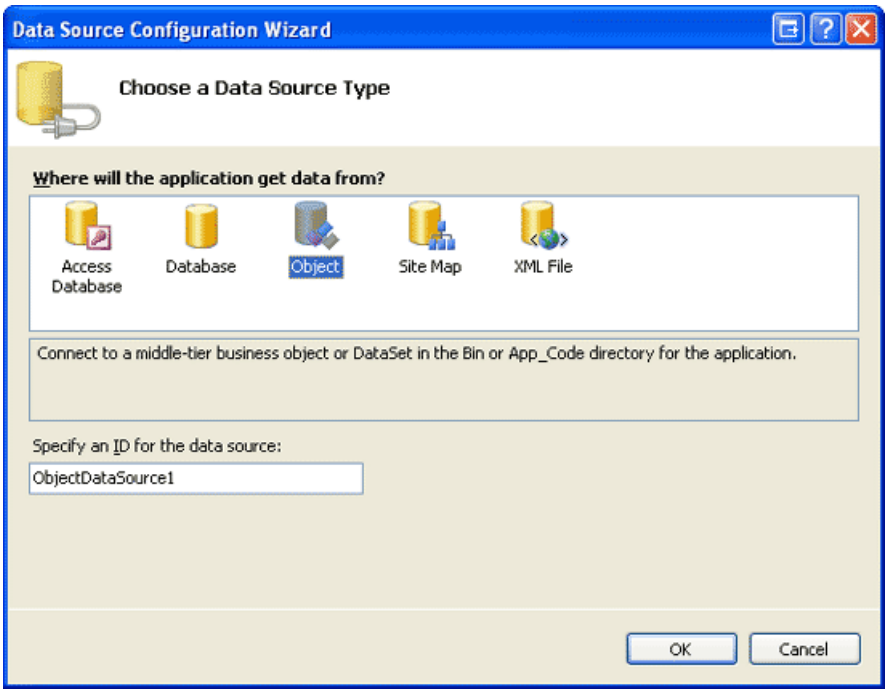


Figure 5: Opt to Create a New ObjectDataSource (Click to view full-size image)

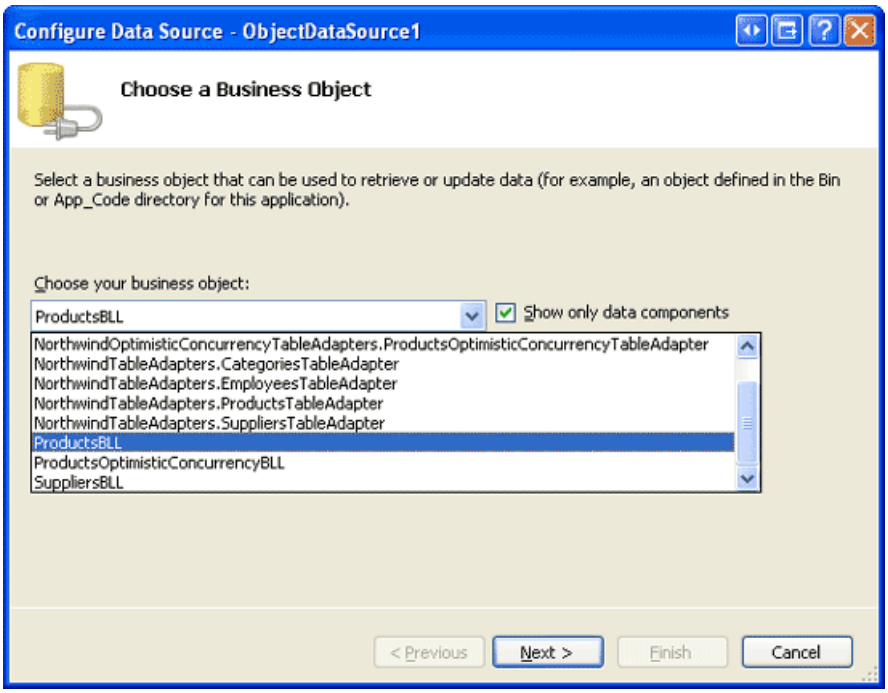


Figure 6: Configure the ObjectDataSource to Use the `ProductsBLL` Class (Click to view full-size image)

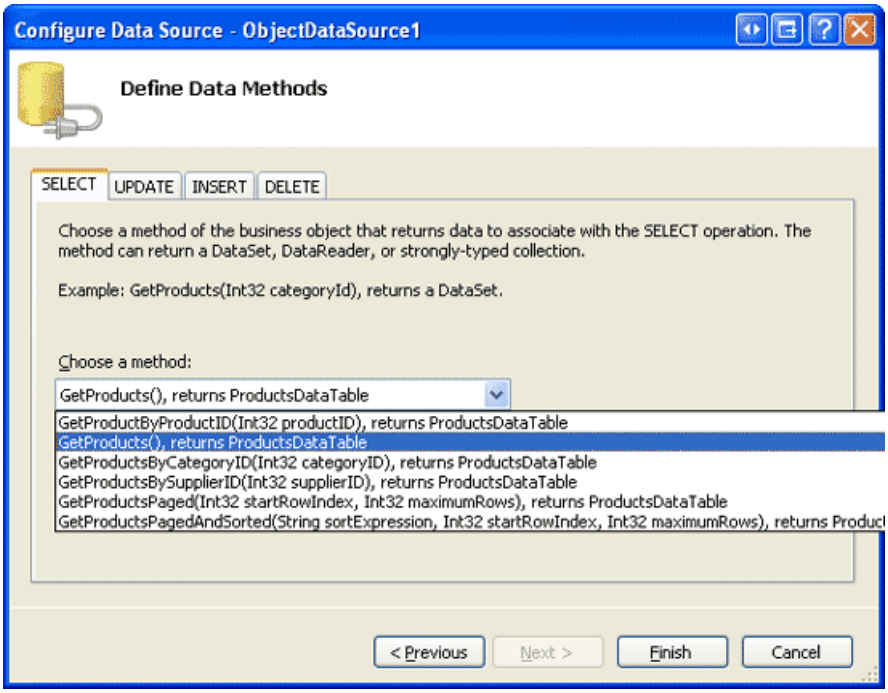


Figure 7: Retrieve Information About All of the Products Using the `GetProducts` Method (Click to view full-size image)

After configuring the ObjectDataSource and associating it with the DataList through its smart tag, Visual Studio will automatically create an `ItemTemplate` in the DataList that displays the name and value of each data field returned by the data source (see the markup below). This default `ItemTemplate` s appearance is identical to that of the templates automatically created when binding a data source to the FormView through the Designer.

```
<asp:DataList ID="DataList1" runat="server" DataKeyField="ProductID"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <ItemTemplate>
        ProductID:       <asp:Label ID="ProductIDLabel" runat="server"
                            Text='<%# Eval("ProductID") %>' /><br />
        ProductName:     <asp:Label ID="ProductNameLabel" runat="server"
                            Text='<%# Eval("ProductName") %>' /><br />
        SupplierID:      <asp:Label ID="SupplierIDLabel" runat="server"
                            Text='<%# Eval("SupplierID") %>' /><br />
        CategoryID:      <asp:Label ID="CategoryIDLabel" runat="server"
                            Text='<%# Eval("CategoryID") %>'/><br />
        QuantityPerUnit: <asp:Label ID="QuantityPerUnitLabel" runat="server"
                            Text='<%# Eval("QuantityPerUnit") %>' /><br />
        UnitPrice:       <asp:Label ID="UnitPriceLabel" runat="server"
                            Text='<%# Eval("UnitPrice") %>' /><br />
        UnitsInStock:    <asp:Label ID="UnitsInStockLabel" runat="server"
                            Text='<%# Eval("UnitsInStock") %>' /><br />
        UnitsOnOrder:    <asp:Label ID="UnitsOnOrderLabel" runat="server"
                            Text='<%# Eval("UnitsOnOrder") %>' /><br />
        ReorderLevel:    <asp:Label ID="ReorderLevelLabel" runat="server"
                            Text='<%# Eval("ReorderLevel") %>' /><br />
        Discontinued:    <asp:Label ID="DiscontinuedLabel" runat="server"
                            Text='<%# Eval("Discontinued") %>' /><br />
        CategoryName:    <asp:Label ID="CategoryNameLabel" runat="server"
                            Text='<%# Eval("CategoryName") %>' /><br />
        SupplierName:    <asp:Label ID="SupplierNameLabel" runat="server"
                            Text='<%# Eval("SupplierName") %>' /><br />
        <br />
    </ItemTemplate>
</asp:DataList>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
</asp:ObjectDataSource>
```

NOTE

Recall that when binding a data source to a FormView control through the FormView s smart tag, Visual Studio created an `ItemTemplate`, `InsertItemTemplate`, and `EditItemTemplate`. With the DataList, however, only an `ItemTemplate` is created. This is because the DataList does not have the same built-in editing and inserting support offered by the FormView. The DataList does contain edit- and delete-related events, and editing and deleting support can be added with a bit of code, but there s no simple out-of-the-box support as with the FormView. We'll see how to include editing and deleting support with the DataList in a future tutorial.

Let s take a moment to improve the appearance of this template. Rather than displaying all of the data fields, let s only display the product s name, supplier, category, quantity per unit, and unit price. Moreover, let s display the name in an `<h4>` heading and lay out the remaining fields using a `<table>` beneath the heading.

To make these changes you can either use the template editing features in the Designer from the DataList s smart tag click on the Edit Templates link or you can modify the template manually through the page s declarative syntax. If you use the Edit Templates option in the Designer, your resulting markup may not match the following markup exactly, but when viewed through a browser should look very similar to the screen shot shown in Figure 8.

```
<asp:DataList ID="DataList1" runat="server" DataKeyField="ProductID"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <ItemTemplate>
        <h4><asp:Label ID="ProductNameLabel" runat="server"
            Text='<%# Eval("ProductName") %>' /></h4>
        <table border="0">
            <tr>
                <td class="ProductPropertyLabel">Category:</td>
                <td><asp:Label ID="CategoryNameLabel" runat="server"
                    Text='<%# Eval("CategoryName") %>' /></td>
                <td class="ProductPropertyLabel">Supplier:</td>
                <td><asp:Label ID="SupplierNameLabel" runat="server"
                    Text='<%# Eval("SupplierName") %>' /></td>
            </tr>
            <tr>
                <td class="ProductPropertyLabel">Qty/Unit:</td>
                <td><asp:Label ID="QuantityPerUnitLabel" runat="server"
                    Text='<%# Eval("QuantityPerUnit") %>' /></td>
                <td class="ProductPropertyLabel">Price:</td>
                <td><asp:Label ID="UnitPriceLabel" runat="server"
                    Text='<%# Eval("UnitPrice", "{0:C}") %>' /></td>
            </tr>
        </table>
    </ItemTemplate>
</asp:DataList>
```

NOTE

The example above uses Label Web controls whose `Text` property is assigned the value of the databinding syntax. Alternatively, we could have omitted the Labels altogether, typing in just the databinding syntax. That is, instead of using `<asp:Label ID="CategoryNameLabel" runat="server" Text='<%# Eval("CategoryName") %>' />` we could have instead used the declarative syntax `<%# Eval("CategoryName") %>`.

Leaving in the Label Web controls, however, offer two advantages. First, it provides an easier means for formatting the data based on the data, as we'll see in the next tutorial. Second, the Edit Templates option in the Designer doesn t display declarative databinding syntax that appears outside of some Web control. Instead, the Edit Templates interface is designed to facilitate working with static markup and Web controls and assumes that any databinding will be done through the Edit DataBindings dialog box, which is accessible from the Web controls smart tags.

Therefore, when working with the DataList, which provides the option of editing the templates through the Designer, I prefer to use Label Web controls so that the content is accessible through the Edit Templates interface. As we'll see shortly, the Repeater requires that the template s contents be edited from the Source view. Consequently, when crafting the Repeater s templates I'll often omit the Label Web controls unless I know I'll need to format the appearance of the data bound text based on programmatic logic.

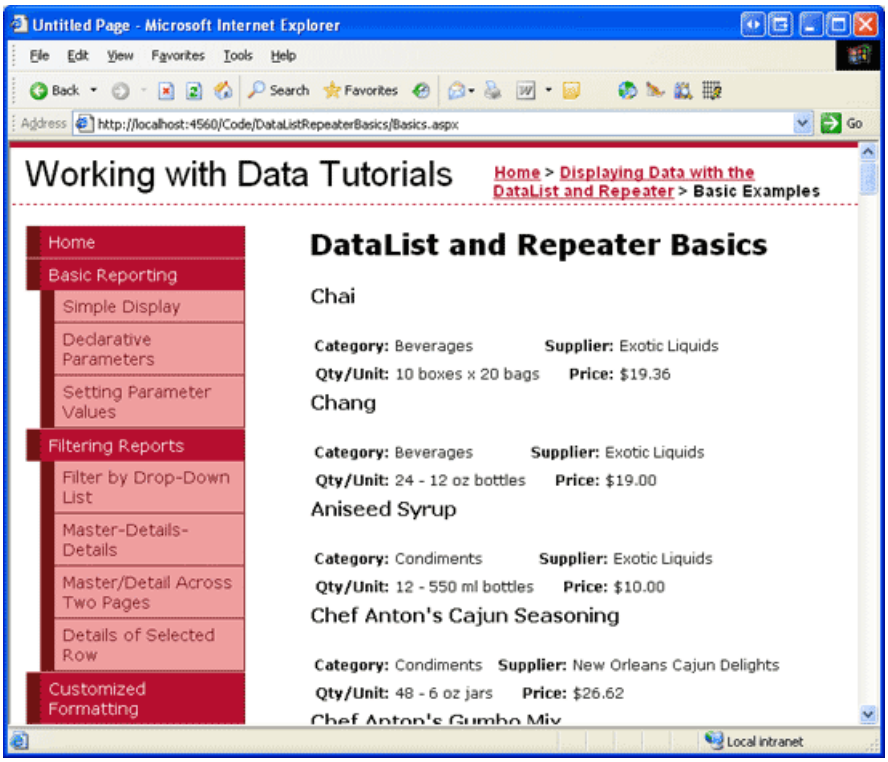


Figure 8: Each Product s Output is Rendered Using the DataList s `ItemTemplate` (Click to view full-size image)

Step 3: Improving the Appearance of the DataList

Like the GridView, the DataList offers a number of style-related properties, such as `Font`, `ForeColor`, `BackColor`, `CssClass`, `ItemStyle`, `AlternatingItemStyle`, `SelectedItemStyle`, and so on. When working with the GridView and DetailsView controls, we created Skin files in the `DataWebControls` Theme that pre-defined the `CssClass` properties for these two controls and the `CssClass` property for several of their subproperties (`RowStyle`, `HeaderStyle`, and so on). Let s do the same for the DataList.

As discussed in the Displaying Data With the ObjectDataSource tutorial, a Skin file specifies the default appearance-related properties for a Web control; a Theme is a collection of Skin, CSS, image, and JavaScript files that define a particular look and feel for a website. In the *Displaying Data With the ObjectDataSource* tutorial, we created a `DataWebControls` Theme (which is implemented as a folder within the `App_Themes` folder) that has, currently, two Skin files - `GridView.skin` and `DetailsView.skin`. Let s add a third Skin file to specify the pre-defined style settings for the DataList.

To add a Skin file, right-click on the `App_Themes/DataWebControls` folder, choose Add a New Item, and select the Skin File option from the list. Name the file `DataList.skin`.

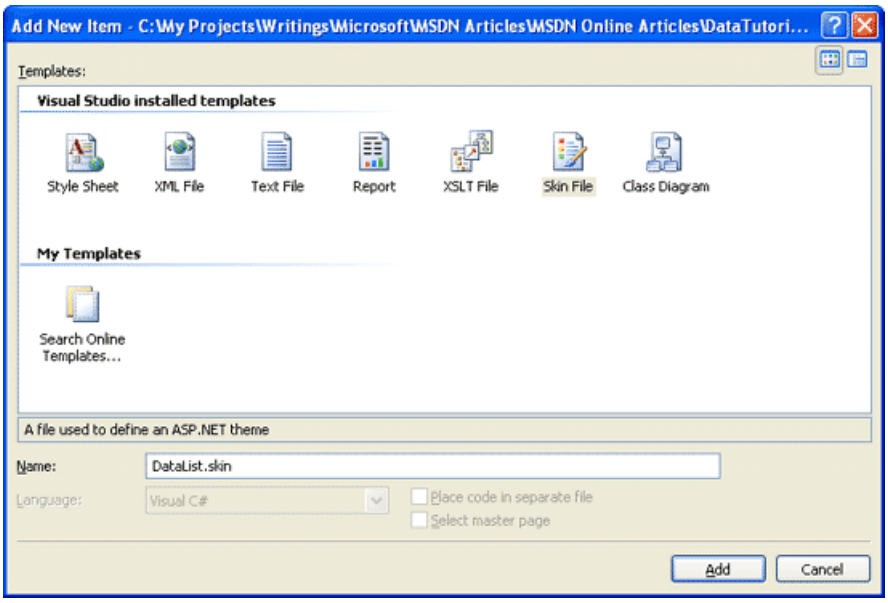


Figure 9: Create a New Skin File Named `DataList.skin` (Click to view full-size image)

Use the following markup for the `DataList.skin` file:

```
<asp:DataList runat="server" CssClass="DataWebControlStyle">
   <AlternatingItemStyle CssClass="AlternatingRowStyle" />
   <ItemStyle CssClass="RowStyle" />
   <HeaderStyle CssClass="HeaderStyle" />
   <FooterStyle CssClass="FooterStyle" />
   <SelectedItemStyle CssClass="SelectedRowStyle" />
</asp:DataList>
```

These settings assign the same CSS classes to the appropriate DataList properties as were used with the GridView and DetailsView controls. The CSS classes used here `DataWebControlStyle`, `AlternatingRowStyle`, `RowStyle`, and so on are defined in the `Styles.css` file and were added in previous tutorials.

With the addition of this Skin file, the DataList s appearance is updated in the Designer (you may need to refresh the Designer view to see the effects of the new Skin file; from the View menu, choose Refresh). As Figure 10 shows, each alternating product has a light pink background color.

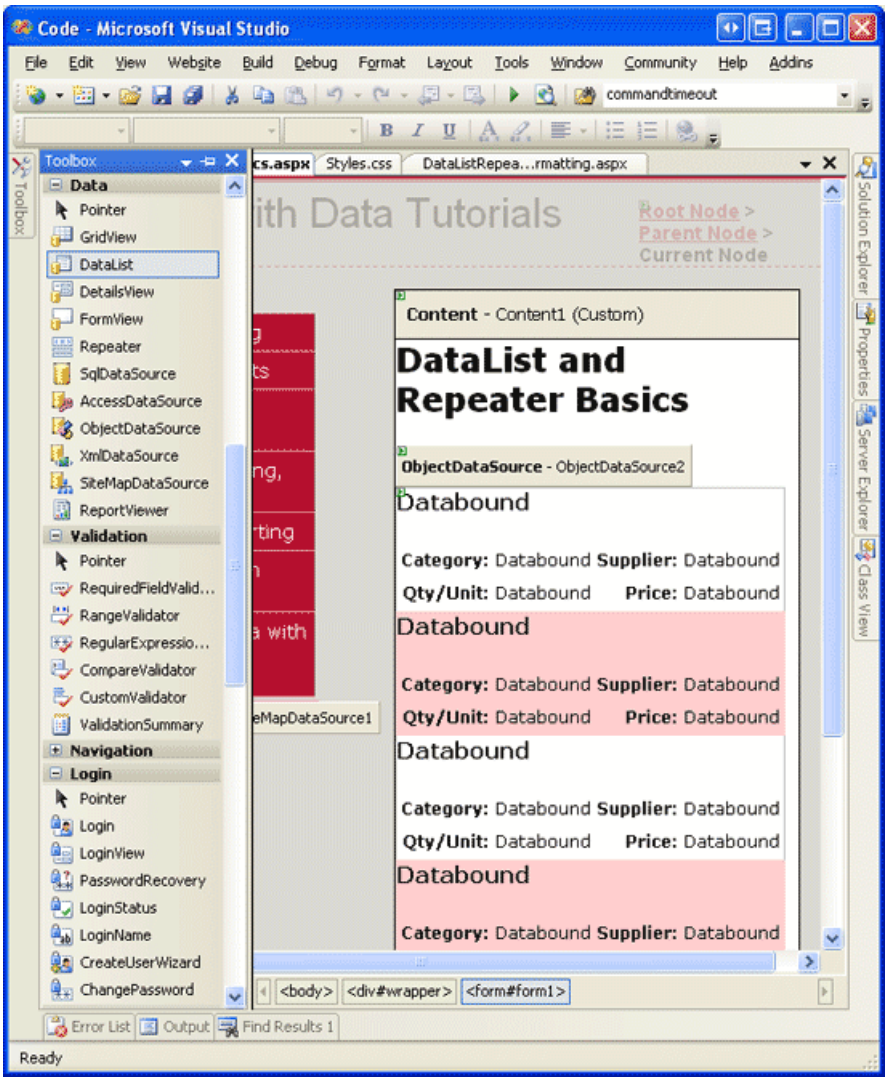


Figure 10: Create a New Skin File Named `DataList.skin` (Click to view full-size image)

Step 4: Exploring the DataList s Other Templates

In addition to the `ItemTemplate`, the DataList supports six other optional templates:

- `HeaderTemplate` if provided, adds a header row to the output and is used to render this row
- `AlternatingItemTemplate` used to render alternating items
- `SelectedItemTemplate` used to render the selected item; the selected item is the item whose index corresponds to the DataList s `SelectedIndex` property
- `EditItemTemplate` used to render the item being edited
- `SeparatorTemplate` if provided, adds a separator between each item and is used to render this separator
- `FooterTemplate` - if provided, adds a footer row to the output and is used to render this row

When specifying the `HeaderTemplate` or `FooterTemplate`, the DataList adds an additional header or footer row to the rendered output. Like with the GridView s header and footer rows, the header and footer in a DataList are not bound to data. Therefore, any databinding syntax in the `HeaderTemplate` or `FooterTemplate` that attempts to access bound data will return a blank string.

NOTE

As we saw in the Displaying Summary Information in the GridView s Footer tutorial, while the header and footer rows don t support databinding syntax, data-specific information can be injected directly into these rows from the GridView s `RowDataBound` event handler. This technique can be used to both calculate running totals or other information from the data bound to the control as well as assign that information to the footer. This same concept can be applied to the DataList and Repeater controls; the only difference is that for the DataList and Repeater create an event handler for the `ItemDataBound` event (instead of for the `RowDataBound` event).

For our example, let s have the title Product Information displayed at the top of the DataList s results in an `<h3>` heading. To accomplish this, add a `HeaderTemplate` with the appropriate markup. From the Designer, this can be accomplished by clicking on the Edit Templates link in the DataList s smart tag, choosing the Header Template from the drop-down list, and typing in the text after picking the Heading 3 option from the style drop-down list (see Figure 11).

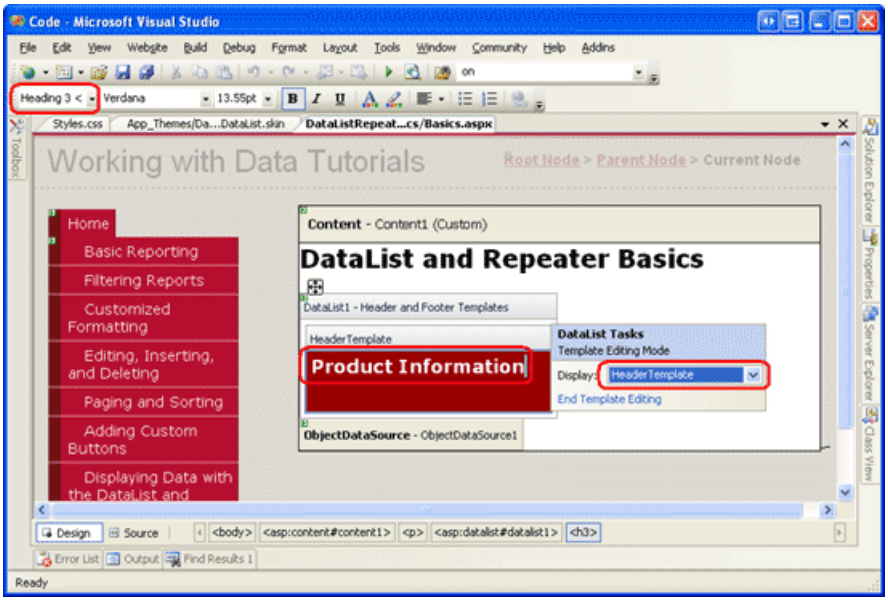


Figure 11: Add a `HeaderTemplate` with the Text Product Information (Click to view full-size image)

Alternatively, this can be added declaratively by entering the following markup within the `<asp:DataList>` tags:

```
<HeaderTemplate>
   <h3>Product Information</h3>
</HeaderTemplate>
```

To add a bit of space between each product listing, let s add a `SeparatorTemplate` that includes a line between each section. The horizontal rule tag (`<hr>`), adds such a divider. Create the `SeparatorTemplate` so that it has the following markup:

```
<SeparatorTemplate>
    <hr />
</SeparatorTemplate>
```

NOTE

Like the `HeaderTemplate` and `FooterTemplates` , the `SeparatorTemplate` is not bound to any record from the data source and therefore cannot directly access the data source records bound to the DataList.

After making this addition, when viewing the page through a browser it should look similar to Figure 12. Note the header row and the line between each product listing.

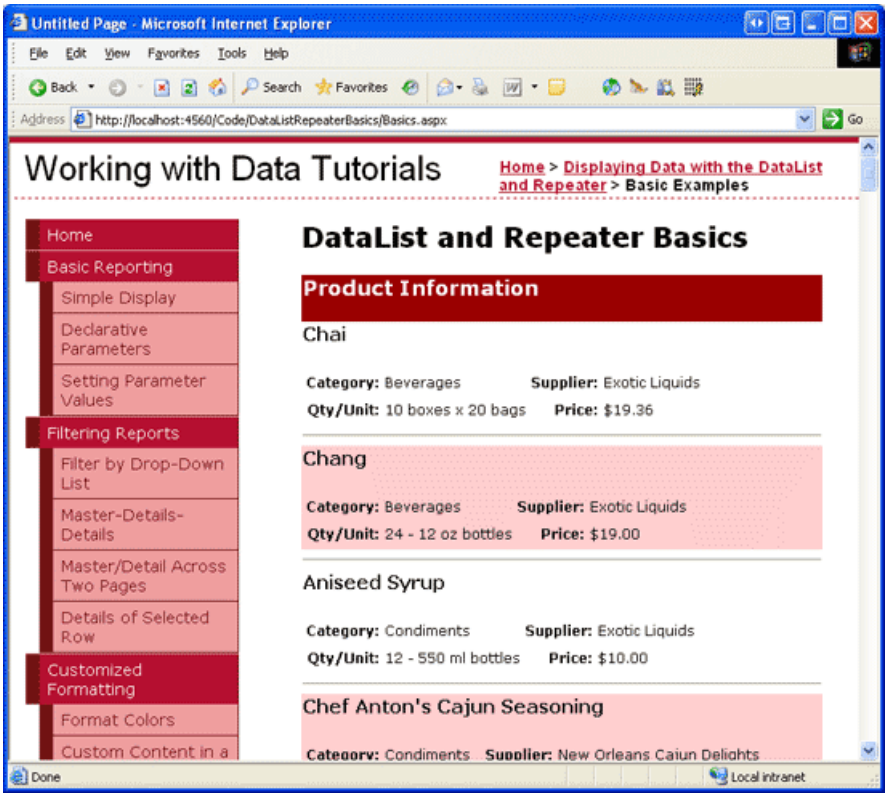


Figure 12: The DataList Includes a Header Row and a Horizontal Rule Between Each Product Listing (Click to view full-size image)

Step 5: Rendering Specific Markup with the Repeater Control

If you do a View/Source from your browser when visiting the DataList example from Figure 12, you'll see that the DataList emits an HTML `<table>` that contains a table row (`<tr>`) with a single table cell (`<td>`) for each item bound to the DataList. This output, in fact, is identical to what would be emitted from a GridView with a single TemplateField. As we'll see in a future tutorial, the DataList does allow further customization of the output, enabling us to display multiple data source records per table row.

What if you don t want to emit an HTML `<table>`, though? For total and complete control over the markup generated by a data Web control, we must use the Repeater control. Like the DataList, the Repeater is constructed based upon templates. The Repeater, however, only offers the following five templates:

- `HeaderTemplate` if provided, adds the specified markup before the items
- `ItemTemplate` used to render items
- `AlternatingItemTemplate` if provided, used to render alternating items
- `SeparatorTemplate` if provided, adds the specified markup between each item
- `FooterTemplate` - if provided, adds the specified markup after the items

In ASP.NET 1.x, the Repeater control was commonly used to display a bulleted list whose data came from some data source. In such a case, the `HeaderTemplate` and `FooterTemplates` would contain the opening and closing `<ul>` tags, respectively, while the `ItemTemplate` would contain `<li>` elements with databinding syntax. This approach can still be used in ASP.NET 2.0 as we saw in two examples in the Master Pages and Site Navigation tutorial:

- In the `Site.master` master page, a Repeater was used to display a bulleted list of the top-level site map contents (Basic Reporting, Filtering Reports, Customized Formatting, and so on); another, nested Repeater was used to display the children sections of the top-level sections

- In `SectionLevelTutorialListing.ascx`, a Repeater was used to display a bulleted list of the children sections of the current site map section

> **NOTE**
>
> ASP.NET 2.0 introduces the new BulletedList control, which can be bound to a data source control in order to display a simple bulleted list. With the BulletedList control we do not need to specify any of the list-related HTML; instead, we simply indicate the data field to display as the text for each list item.

The Repeater serves as a catch all data Web control. If there is not an existing control that generates the needed markup, the Repeater control can be used. To illustrate using the Repeater, let s have the list of categories displayed above the Product Information DataList created in Step 2. In particular, let s have the categories displayed in a single-row HTML `<table>` with each category displayed as a column in the table.

To accomplish this, start by dragging a Repeater control from the Toolbox onto the Designer, above the Product Information DataList. As with the DataList, the Repeater initially displays as a gray box until its templates have been defined.

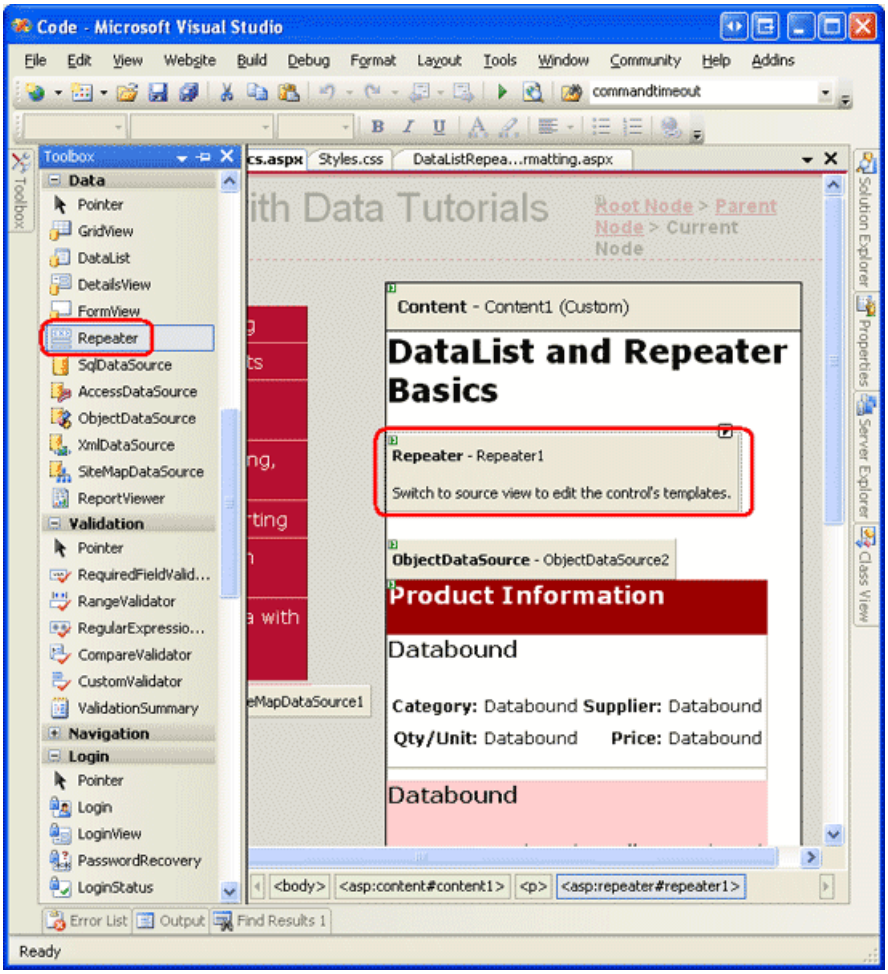


Figure 13: Add a Repeater to the Designer (Click to view full-size image)

There s only one option in the Repeater s smart tag: Choose Data Source. Opt to create a new ObjectDataSource and configure it to use the `CategoriesBLL` class s `GetCategories` method.

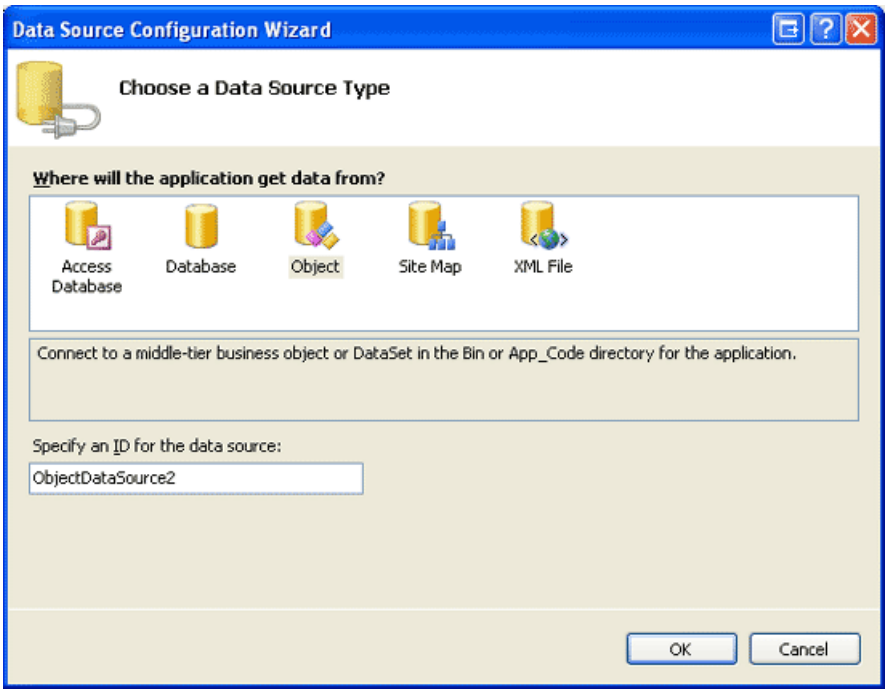


Figure 14: Create a New ObjectDataSource (Click to view full-size image)

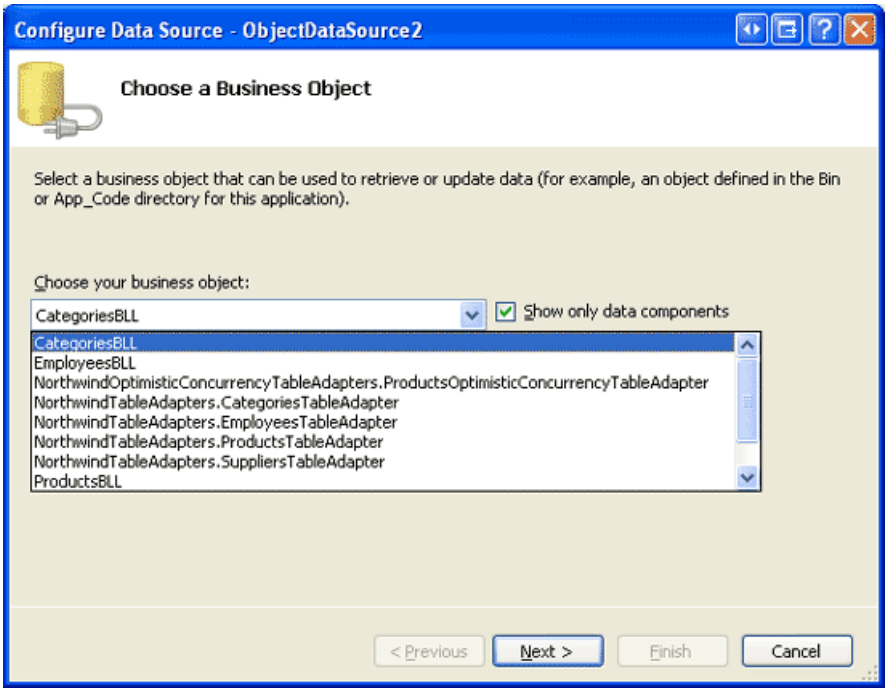


Figure 15: Configure the ObjectDataSource to Use the `CategoriesBLL` Class (Click to view full-size image)

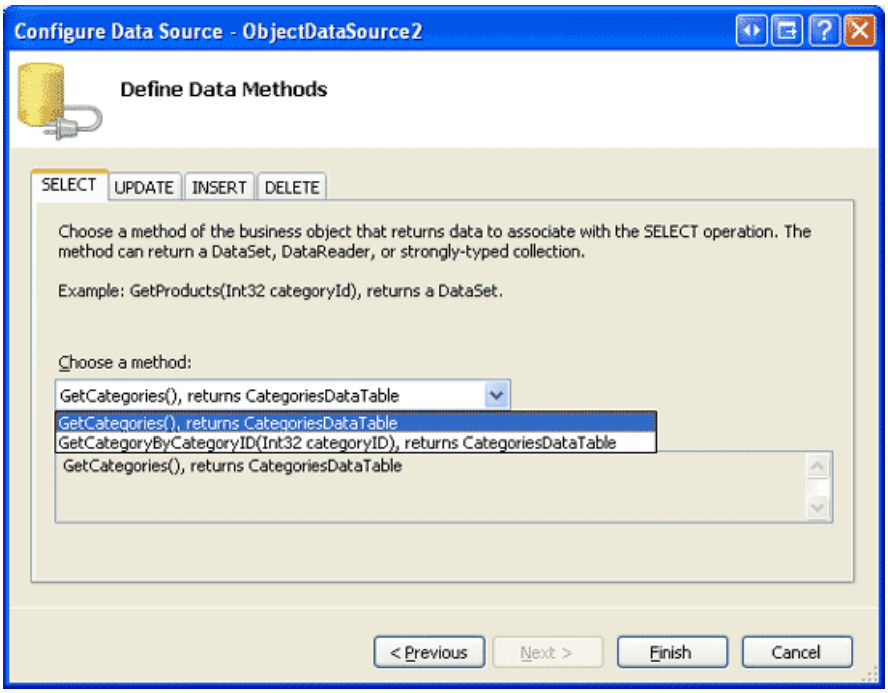


Figure 16: Retrieve Information About All of the Categories Using the `GetCategories` Method (Click to view full-size image)

Unlike the DataList, Visual Studio does not automatically create an ItemTemplate for the Repeater after binding it to a data source. Furthermore, the Repeater s templates cannot be configured through the Designer and must be specified declaratively.

To display the categories as a single-row `<table>` with a column for each category, we need the Repeater to emit markup similar to the following:

```
<table>
   <tr>
      <td>Category 1</td>
      <td>Category 2</td>
      ...
      <td>Category N</td>
   </tr>
</table>
```

Since the `<td>Category X</td>` text is the portion that repeats, this will appear in the Repeater s ItemTemplate. The markup that appears before it - `<table><tr>` - will be placed in the `HeaderTemplate` while the ending markup - `</tr></table>` - will placed in the `FooterTemplate`. To enter these template settings, go to the declarative portion of the ASP.NET page by clicking on the Source button in the lower left corner and type in the following syntax:

```
<asp:Repeater ID="Repeater1" runat="server" DataSourceID="ObjectDataSource2"
    EnableViewState="False">
    <HeaderTemplate>
        <table>
            <tr>
    </HeaderTemplate>
    <ItemTemplate>
                <td><%# Eval("CategoryName") %></td>
    </ItemTemplate>
    <FooterTemplate>
            </tr>
        </table>
    </FooterTemplate>
</asp:Repeater>
```

The Repeater emits the precise markup as specified by its templates, nothing more, nothing less. Figure 17 shows the Repeater s output when viewed through a browser.

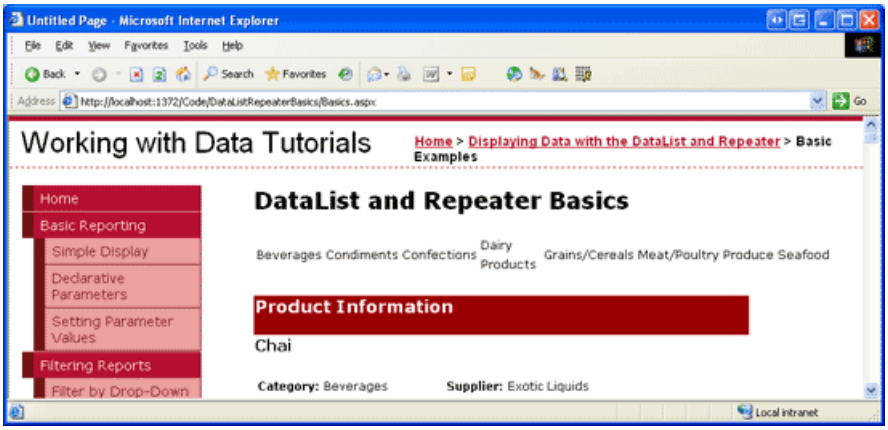


Figure 17: A Single-Row HTML `<table>` Lists Each Category in a Separate Column (Click to view full-size image)

Step 6: Improving the Appearance of the Repeater

Since the Repeater emits precisely the markup specified by its templates, it should come as no surprise that there are no style-related properties for the Repeater. To alter the appearance of the content generated by the Repeater, we must manually add the needed HTML or CSS content directly to the Repeater s templates.

For our example, let s have the category columns alternate background colors, like with the alternating rows in the DataList. To accomplish this, we need to assign the `RowStyle` CSS class to each Repeater item and the `AlternatingRowStyle` CSS class to each alternating Repeater item through the `ItemTemplate` and `AlternatingItemTemplate` templates, like so:

```
<ItemTemplate>
    <td class="RowStyle">&lt;%# Eval("CategoryName") %&gt;</td>
</ItemTemplate>
<AlternatingItemTemplate>
    <td class="AlternatingRowStyle">&lt;%# Eval("CategoryName") %&gt;</td>
</AlternatingItemTemplate>
```

Let s also add a header row to the output with the text Product Categories. Since we don t know how many columns our resulting `<table>` will be comprised of, the simplest way to generate a header row that is guaranteed to span all columns is to use *two* `<table>` s. The first `<table>` will contain two rows the header row and a row that will contain the second, single-row `<table>` that has a column for each category in the system. That is, we want to emit the following markup:

```
<table>
   <tr>
      <th>Product Categories</th>
   </tr>
   <tr>
      <td>
         <table>
            <tr>
               <td>Category 1</td>
               <td>Category 2</td>
               ...
               <td>Category N</td>
            </tr>
         </table>
      </td>
   </tr>
</table>
```

The following `HeaderTemplate` and `FooterTemplate` result in the desired markup:

```
<asp:Repeater ID="Repeater1" runat="server" DataSourceID="ObjectDataSource2"
    EnableViewState="False">
    <HeaderTemplate>
        <table cellpadding="0" cellspacing="0">
            <tr>
                <th class="HeaderStyle">Product Categories</th>
            </tr>
            <tr>
                <td>
                    <table cellpadding="4" cellspacing="0">
                        <tr>
    </HeaderTemplate>
    <ItemTemplate>
                            <td class="RowStyle"><%# Eval("CategoryName") %></td>
    </ItemTemplate>
    <AlternatingItemTemplate>
                            <td class="AlternatingRowStyle">
                                <%# Eval("CategoryName") %></td>
    </AlternatingItemTemplate>
    <FooterTemplate>
                        </tr>
                    </table>
                </td>
            </tr>
        </table>
    </FooterTemplate>
</asp:Repeater>
```

Figure 18 shows the Repeater after these changes have been made.

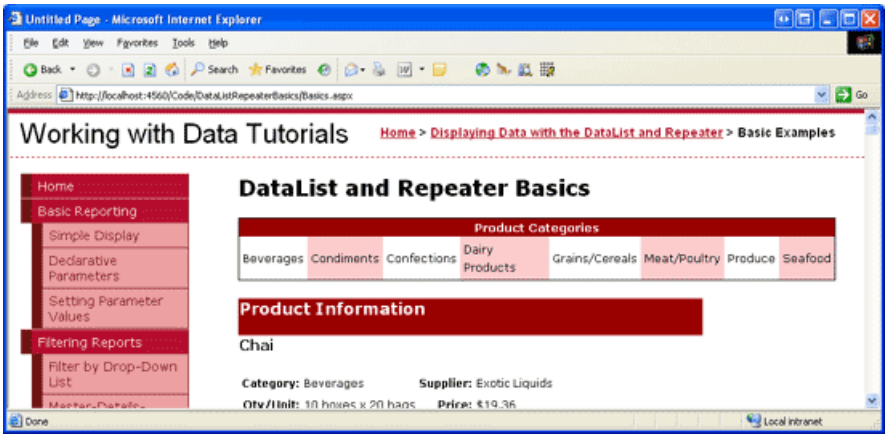


Figure 18: The Category Columns Alternate in Background Color and Includes a Header Row (Click to view full-size image)

Summary

While the GridView control makes it easy to display, edit, delete, sort, and page through data, the appearance is very boxy and grid-like. For more control over the appearance, we need to turn to either the DataList or Repeater controls. Both of these controls display a set of records using templates instead of BoundFields, CheckBoxFields, and so on.

The DataList renders as an HTML `<table>` that, by default, displays each data source record in a single table row, just like a GridView with a single TemplateField. As we will see in a future tutorial, however, the DataList does permit multiple records to be displayed per table row. The Repeater, on the other hand, strictly emits the markup specified in its templates; it does not add any additional markup and therefore is commonly used to display data in HTML elements other than a `<table>` (such as in a bulleted list).

While the DataList and Repeater offer more flexibility in their rendered output, they lack many of the built-in features found in the GridView. As we'll examine in upcoming tutorials, some of these features can be plugged back in without too much effort, but do keep in mind that using the DataList or Repeater in lieu of the GridView does limit the features you can use without having to implement those features yourself.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Yaakov Ellis, Liz Shulok, Randy Schmidt, and Stacy Park. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Formatting the DataList and Repeater Based Upon Data (C#)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll step through examples of how we format the appearance of the DataList and Repeater controls, either by using formatting functions within templates or by handling the DataBound event.

Introduction

As we saw in the preceding tutorial, the DataList offers a number of style-related properties that affect its appearance. In particular, we saw how to assign default CSS classes to the DataList s `HeaderStyle`, `ItemStyle`, `AlternatingItemStyle`, and `SelectedItemStyle` properties. In addition to these four properties, the DataList includes a number of other style-related properties, such as `Font`, `ForeColor`, `BackColor`, and `BorderWidth`, to name a few. The Repeater control does not contain any style-related properties. Any such style settings must be made directly within the markup in the Repeater s templates.

Often, though, how data should be formatted depends on the data itself. For example, when listing products we might want to display the product information in a light gray font color if it is discontinued, or we may want to highlight the `UnitsInStock` value if it is zero. As we saw in previous tutorials, the GridView, DetailsView, and FormView offer two distinct ways to format their appearance based on their data:

- **The `DataBound` event** create an event handler for the appropriate `DataBound` event, which fires after the data has been bound to each item (for the GridView it was the `RowDataBound` event; for the DataList and Repeater it is the `ItemDataBound` event). In that event handler, the data just bound can be examined and formatting decisions made. We examined this technique in the Custom Formatting Based Upon Data tutorial.
- **Formatting Functions in Templates** when using TemplateFields in the DetailsView or GridView controls, or a template in the FormView control, we can add a formatting function to the ASP.NET page s code-behind class, the Business Logic Layer, or any other class library that is accessible from the web application. This formatting function can accept an arbitrary number of input parameters, but must return the HTML to render in the template. Formatting functions were first examined in the Using TemplateFields in the GridView Control tutorial.

Both of these formatting techniques are available with the DataList and Repeater controls. In this tutorial we'll step through examples using both techniques for both controls.

Using the `ItemDataBound` Event Handler

When data is bound to a DataList, either from a data source control or through programmatically assigning data to the control s `DataSource` property and calling its `DataBind()` method, the DataList s `DataBinding` event fires, the data source enumerated, and each data record is bound to the DataList. For each record in the data source, the DataList creates a `DataListItem` object that is then bound to the current record. During this process, the DataList raises two events:

- `ItemCreated` fires after the `DataListItem` has been created
- `ItemDataBound` fires after the current record has been bound to the `DataListItem`

The following steps outline the data binding process for the DataList control.

1. The DataList s `DataBinding` event fires
2. The data is bound to the DataList

   For each record in the data source

   a. Create a `DataListItem` object
   b. Fire the `ItemCreated` event
   c. Bind the record to the `DataListItem`
   d. Fire the `ItemDataBound` event
   e. Add the `DataListItem` to the `Items` collection

When binding data to the Repeater control, it progresses through the exact same sequence of steps. The only difference is that instead of `DataListItem` instances being created, the Repeater uses `RepeaterItem` s.

> **NOTE**
> The astute reader may have noticed a slight anomaly between the sequence of steps that transpire when the DataList and Repeater are bound to data versus when the GridView is bound to data. At the tail end of the data binding process, the GridView raises the `DataBound` event; however, neither the DataList nor Repeater control have such an event. This is because the DataList and Repeater controls were created back in the ASP.NET 1.x timeframe, before the pre- and post-level event handler pattern had become common.

Like with the GridView, one option for formatting based on the data is to create an event handler for the `ItemDataBound` event. This event handler would inspect the data that had just been bound to the `DataListItem` or `RepeaterItem` and affect the formatting of the control as needed.

For the DataList control, formatting changes for the entire item can be implemented using the `DataListItem` s style-related properties, which include the standard `Font` , `ForeColor` , `BackColor` , `CssClass` , and so on. To affect the formatting of particular Web controls within the DataList s template, we need to programmatically access and modify the style of those Web controls. We saw how to accomplish this back in the *Custom Formatting Based Upon Data* tutorial. Like the Repeater control, the `RepeaterItem` class has no style-related properties; therefore, all style-related changes made to a `RepeaterItem` in the `ItemDataBound` event handler must be done by programmatically accessing and updating Web controls within the template.

Since the `ItemDataBound` formatting technique for the DataList and Repeater are virtually identical, our example will focus on using the DataList.

Step 1: Displaying Product Information in the DataList

Before we worry about the formatting, let s first create a page that uses a DataList to display product information. In the previous tutorial we created a DataList whose `ItemTemplate` displayed each product s name, category, supplier, quantity per unit, and price. Let s repeat this functionality here in this tutorial. To accomplish this, you can either recreate the DataList and its ObjectDataSource from scratch, or you can copy over those controls from the page created in the previous tutorial (`Basics.aspx`) and paste them into the page for this tutorial (`Formatting.aspx`).

Once you have replicated the DataList and ObjectDataSource functionality from `Basics.aspx` into `Formatting.aspx` , take a moment to change the DataList s `ID` property from `DataList1` to a more descriptive `ItemDataBoundFormattingExample` . Next, view the DataList in a browser. As Figure 1 shows, the only formatting difference between each product is that the background color alternates.

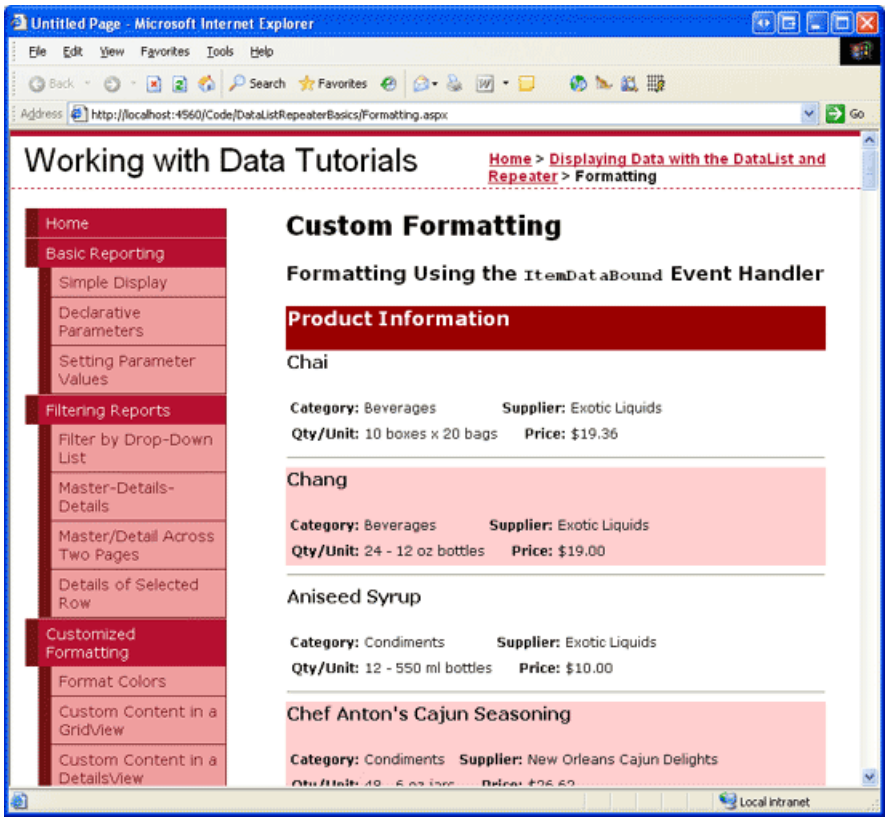


Figure 1: The Products are Listed in the DataList Control (Click to view full-size image)

For this tutorial, let s format the DataList such that any products with a price less than $20.00 will have both its name and unit price highlighted yellow.

Step 2: Programmatically Determining the Value of the Data in the ItemDataBound Event Handler

Since only those products with a price under $20.00 will have the custom formatting applied, we must be able to determine each product s price. When binding data to a DataList, the DataList enumerates the records in its data source and, for each record, creates a `DataListItem` instance, binding the data source record to the `DataListItem`. After the particular record s data has been bound to the current `DataListItem` object, the DataList s `ItemDataBound` event is fired. We can create an event handler for this event to inspect the data values for the current `DataListItem` and, based upon those values, make any formatting changes necessary.

Create an `ItemDataBound` event for the DataList and add the following code:

```
protected void ItemDataBoundFormattingExample_ItemDataBound
    (object sender, DataListItemEventArgs e)
{
    if (e.Item.ItemType == ListItemType.Item ||
        e.Item.ItemType == ListItemType.AlternatingItem)
    {
        // Programmatically reference the ProductsRow instance bound
        // to this DataListItem
        Northwind.ProductsRow product =
            (Northwind.ProductsRow)((System.Data.DataRowView)e.Item.DataItem).Row;
        // See if the UnitPrice is not NULL and less than $20.00
        if (!product.IsUnitPriceNull() && product.UnitPrice < 20)
        {
            // TODO: Highlight the product's name and price
        }
    }
}
```

While the concept and semantics behind the DataList s `ItemDataBound` event handler are the same as those used by the GridView s `RowDataBound` event handler in the *Custom Formatting Based Upon Data* tutorial, the syntax differs slightly. When the `ItemDataBound` event fires, the `DataListItem` just bound to data is passed into corresponding event handler via `e.Item` (instead of `e.Row`, as with the GridView s `RowDataBound` event handler). The DataList s `ItemDataBound` event handler fires for *each* row added to the DataList, including header rows, footer rows, and separator rows. However, the product information is only bound to the data rows. Therefore, when using the `ItemDataBound` event to inspect the data bound to the DataList, we need to first ensure that we re working with a data item. This can be accomplished by checking the `DataListItem` s `ItemType` property, which can have one of the following eight values:

- `AlternatingItem`
- `EditItem`
- `Footer`
- `Header`
- `Item`
- `Pager`
- `SelectedItem`
- `Separator`

Both `Item` and `AlternatingItem``DataListItem` s makeup the DataList s data items. Assuming we re working with an `Item` or `AlternatingItem`, we access the actual `ProductsRow` instance that was bound to the current `DataListItem`. The `DataListItem` s `DataItem` property contains a reference to the `DataRowView` object, whose `Row` property provides a reference to the actual `ProductsRow` object.

Next, we check the `ProductsRow` instance s `UnitPrice` property. Since the Products table s `UnitPrice` field allows `NULL` values, before attempting to access the `UnitPrice` property we should first check to see if it has a `NULL` value using the `IsUnitPriceNull()` method. If the `UnitPrice` value is not `NULL`, we then check to see if it s less than $20.00. If it is indeed under $20.00, we then need to apply the custom formatting.

Step 3: Highlighting the Product s Name and Price

Once we know that a product s price is less than $20.00, all that remains is to highlight its name and price. To accomplish this, we must first programmatically reference the Label controls in the `ItemTemplate` that display the product s name and price. Next, we need to have them display a yellow background. This formatting information can be applied by directly modifying the Labels `BackColor` properties (`LabelID.BackColor = Color.Yellow`); ideally, though, all display-related matters should be expressed through cascading stylesheets. In fact, we already have a stylesheet that provides the desired formatting defined in `Styles.css` - `AffordablePriceEmphasis`, which was created and discussed in the *Custom Formatting Based Upon Data* tutorial.

To apply the formatting, simply set the two Label Web controls `CssClass` properties to `AffordablePriceEmphasis`, as shown in the following code:

```
// Highlight the product name and unit price Labels
// First, get a reference to the two Label Web controls
Label ProductNameLabel = (Label)e.Item.FindControl("ProductNameLabel");
Label UnitPriceLabel = (Label)e.Item.FindControl("UnitPriceLabel");
// Next, set their CssClass properties
if (ProductNameLabel != null)
    ProductNameLabel.CssClass = "AffordablePriceEmphasis";
if (UnitPriceLabel != null)
    UnitPriceLabel.CssClass = "AffordablePriceEmphasis";
```

With the `ItemDataBound` event handler completed, revisit the `Formatting.aspx` page in a browser. As Figure 2

illustrates, those products with a price under $20.00 have both their name and price highlighted.

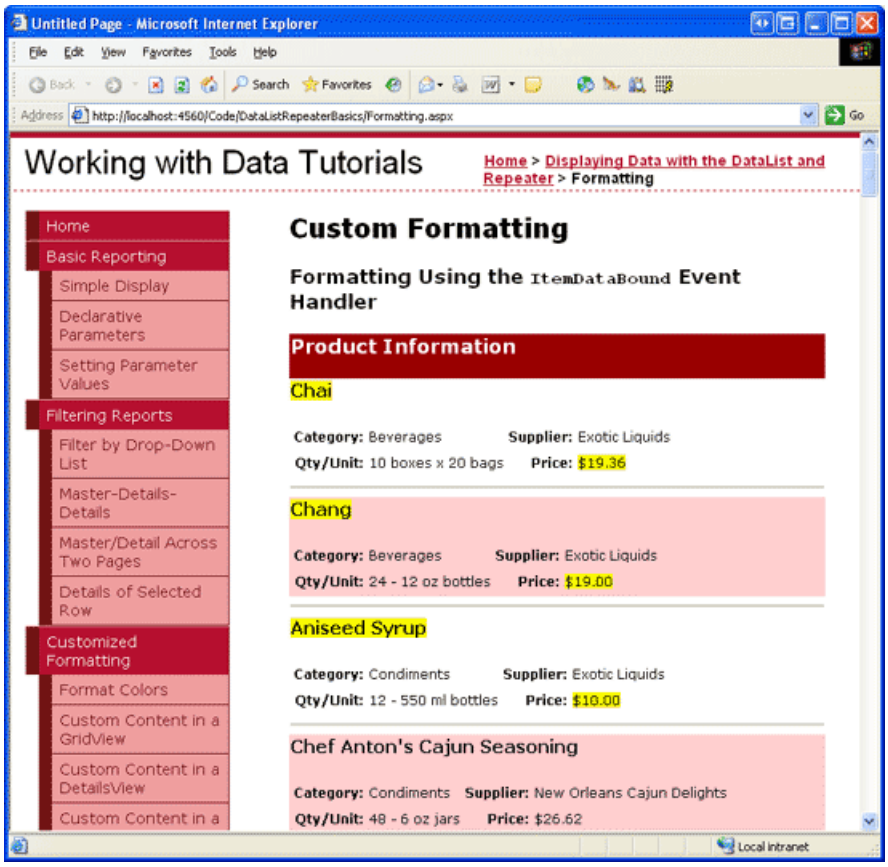


Figure 2: Those Products Less than $20.00 are Highlighted (Click to view full-size image)

> **NOTE**
>
> Since the DataList is rendered as an HTML `<table>`, its `DataListItem` instances have style-related properties that can be set to apply a specific style to the entire item. For example, if we wanted to highlight the *entire* item yellow when its price was less than $20.00, we could have replaced the code that referenced the Labels and set their `CssClass` properties with the following line of code: `e.Item.CssClass = "AffordablePriceEmphasis"` (see Figure 3).

The `RepeaterItem` s that make up the Repeater control, however, don t offer such style-level properties. Therefore, applying custom formatting to the Repeater requires the application of style properties to the Web controls within the Repeater s templates, just like we did in Figure 2.

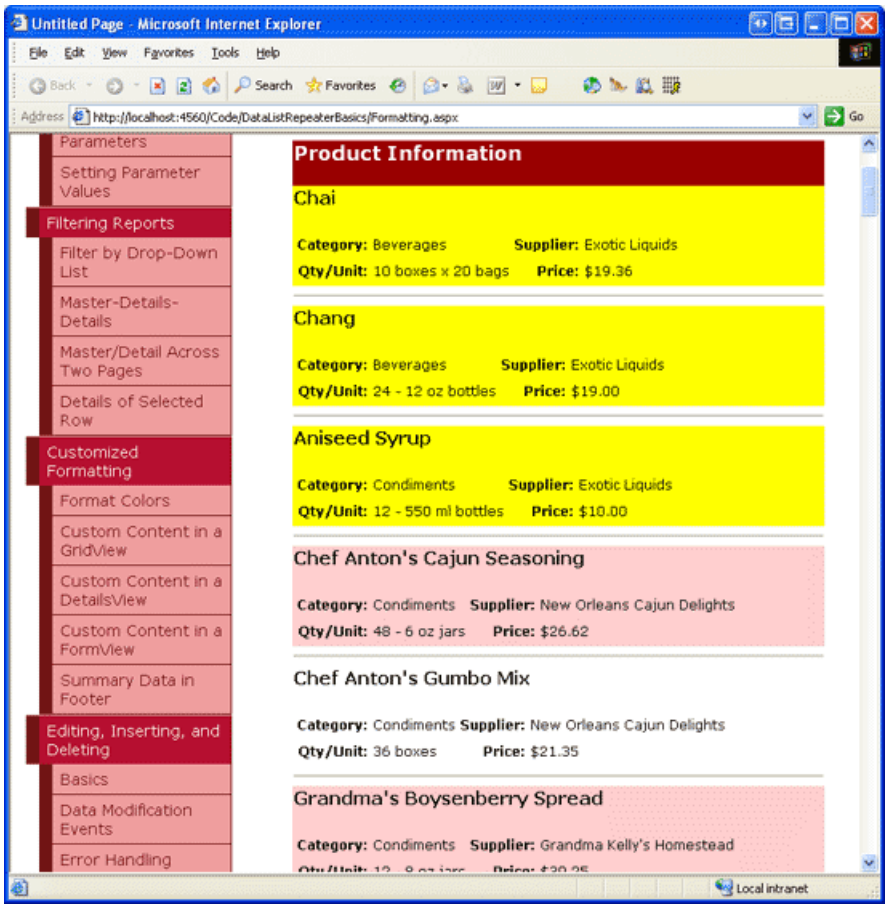


Figure 3: The Entire Product Item is Highlighted for Products Under $20.00 (Click to view full-size image)

Using Formatting Functions from Within the Template

In the *Using TemplateFields in the GridView Control* tutorial we saw how to use a formatting function within a GridView TemplateField to apply custom formatting based upon the data bound to the GridView s rows. A formatting function is a method that can be invoked from a template and returns the HTML to be emitted in its place. Formatting functions can reside in the ASP.NET page s code-behind class or can be centralized into class files in the `App_Code` folder or in a separate Class Library project. Moving the formatting function out of the ASP.NET page s code-behind class is ideal if you plan on using the same formatting function in multiple ASP.NET pages or in other ASP.NET web applications.

To demonstrate formatting functions, let s have the product information include the text [DISCONTINUED] next to the product s name if it s discontinued. Also, let s have the price highlighted yellow if it s less than $20.00 (as we did in the `ItemDataBound` event handler example); if the price is $20.00 or higher, let s not display the actual price, but instead the text, Please call for a price quote. Figure 4 shows a screen shot of the products listing with these formatting rules applied.

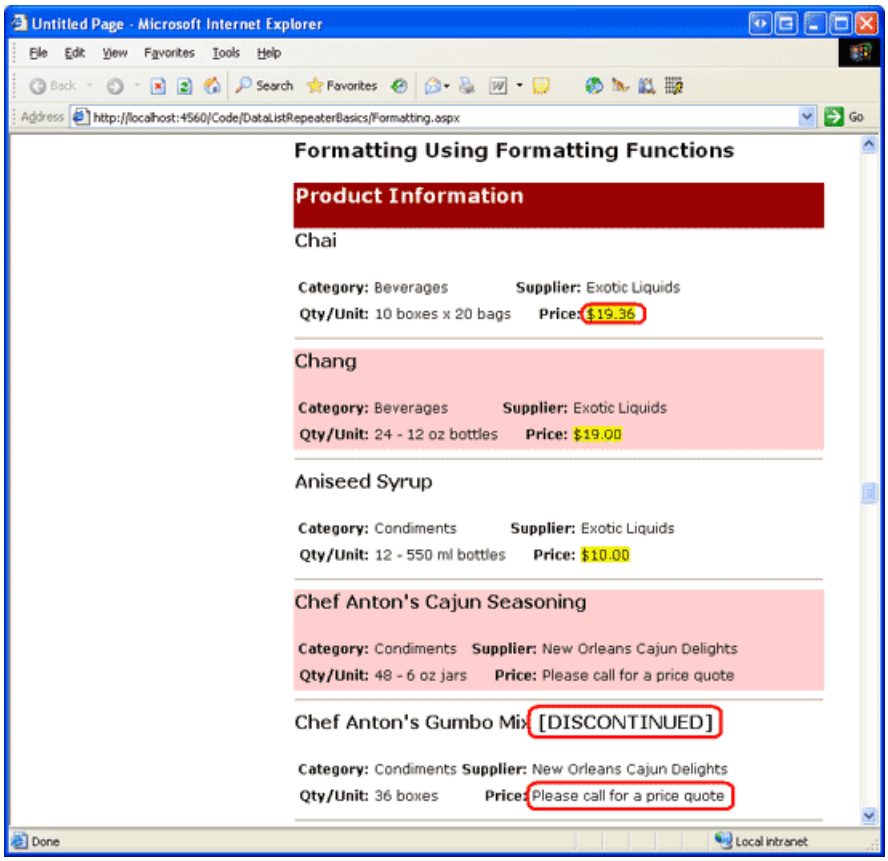


Figure 4: For Expensive Products, the Price is Replaced with the Text, Please call for a price quote (Click to view full-size image)

Step 1: Create the Formatting Functions

For this example we need two formatting functions, one that displays the product name along with the text [DISCONTINUED], if needed, and another that displays either a highlighted price if it s less than $20.00, or the text, Please call for a price quote otherwise. Let s create these functions in the ASP.NET page s code-behind class and name them `DisplayProductNameAndDiscontinuedStatus` and `DisplayPrice`. Both methods need to return the HTML to render as a string and both need to be marked `Protected` (or `Public`) in order to be invoked from the ASP.NET page s declarative syntax portion. The code for these two methods follows:

```
protected string DisplayProductNameAndDiscontinuedStatus
    (string productName, bool discontinued)
{
    // Return just the productName if discontinued is false
    if (!discontinued)
        return productName;
    else
        // otherwise, return the productName appended with the text "[DISCONTINUED]"
        return string.Concat(productName, " [DISCONTINUED]");
}
protected string DisplayPrice(Northwind.ProductsRow product)
{
    // If price is less than $20.00, return the price, highlighted
    if (!product.IsUnitPriceNull() && product.UnitPrice < 20)
        return string.Concat("<span class=\"AffordablePriceEmphasis\">",
                             product.UnitPrice.ToString("C"), "</span>");
    else
        // Otherwise return the text, "Please call for a price quote"
        return "<span>Please call for a price quote</span>";
}
```

Note that the `DisplayProductNameAndDiscontinuedStatus` method accepts the values of the `productName` and

`discontinued` data fields as scalar values, whereas the `DisplayPrice` method accepts a `ProductsRow` instance (rather than a `unitPrice` scalar value). Either approach will work; however, if the formatting function is working with scalar values that can contain database `NULL` values (such as `UnitPrice`; neither `ProductName` nor `Discontinued` allow `NULL` values), special care must be taken in handling these scalar inputs.

In particular, the input parameter must be of type `Object` since the incoming value might be a `DBNull` instance instead of the expected data type. Additionally, a check must be made to determine whether or not the incoming value is a database `NULL` value. That is, if we wanted the `DisplayPrice` method to accept the price as a scalar value, we d have to use the following code:

```
protected string DisplayPrice(object unitPrice)
{
    // If price is less than $20.00, return the price, highlighted
    if (!Convert.IsDBNull(unitPrice) && ((decimal) unitPrice) < 20)
        return string.Concat("<span class=\"AffordablePriceEmphasis\">",
                             ((decimal) unitPrice).ToString("C"), "</span>");
    else
        // Otherwise return the text, "Please call for a price quote"
        return "<span>Please call for a price quote</span>";
}
```

Note that the `unitPrice` input parameter is of type `Object` and that the conditional statement has been modified to ascertain if `unitPrice` is `DBNull` or not. Furthermore, since the `unitPrice` input parameter is passed in as an `Object`, it must be cast to a decimal value.

Step 2: Calling the Formatting Function from the DataList s ItemTemplate

With the formatting functions added to our ASP.NET page s code-behind class, all that remains is to invoke these formatting functions from the DataList s `ItemTemplate`. To call a formatting function from a template, place the function call within the databinding syntax:

```
<%# MethodName(inputParameter1, inputParameter2, ...) %>
```

In the DataList s `ItemTemplate` the `ProductNameLabel` Label Web control currently displays the product s name by assigning its `Text` property the result of `<%# Eval("ProductName") %>`. In order to have it display the name plus the text [DISCONTINUED], if needed, update the declarative syntax so that it instead assigns the `Text` property the value of the `DisplayProductNameAndDiscontinuedStatus` method. When doing so, we must pass in the product s name and discontinued values using the `Eval("columnName")` syntax. `Eval` returns a value of type `Object`, but the `DisplayProductNameAndDiscontinuedStatus` method expects input parameters of type `String` and `Boolean`; therefore, we must cast the values returned by the `Eval` method to the expected input parameter types, like so:

```
<h4>
    <asp:Label ID="ProductNameLabel" runat="server"
        Text='<%# DisplayProductNameAndDiscontinuedStatus((string) Eval("ProductName"),
              (bool) Eval("Discontinued")) %>'>
    </asp:Label>
</h4>
```

To display the price, we can simply set the `UnitPriceLabel` Label s `Text` property to the value returned by the `DisplayPrice` method, just like we did for displaying the product s name and [DISCONTINUED] text. However, instead of passing in the `UnitPrice` as a scalar input parameter, we instead pass in the entire `ProductsRow` instance:

```
<asp:Label ID="UnitPriceLabel" runat="server"
    Text='<%# DisplayPrice((Northwind.ProductsRow)
          ((System.Data.DataRowView) Container.DataItem).Row) %>'>
</asp:Label>
```

With the calls to the formatting functions in place, take a moment to view our progress in a browser. Your screen should look similar to Figure 5, with the discontinued products including the text [DISCONTINUED] and those products costing more than $20.00 having their price replaced with the text Please call for a price quote .

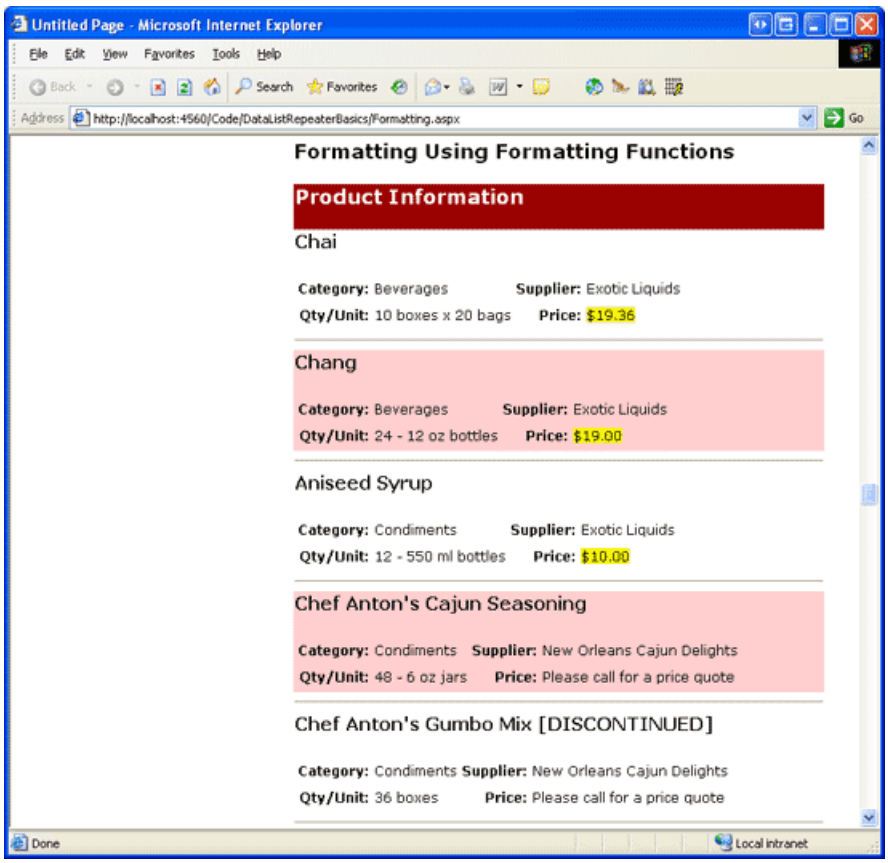


Figure 5: For Expensive Products, the Price is Replaced with the Text, Please call for a price quote (Click to view full-size image)

Summary

Formatting the contents of a DataList or Repeater control based upon the data can be accomplished using two techniques. The first technique is to create an event handler for the `ItemDataBound` event, which fires as each record in the data source is bound to a new `DataListItem` or `RepeaterItem`. In the `ItemDataBound` event handler, the current item s data can be examined and then formatting can be applied to the contents of the template or, for `DataListItem` s, to the entire item itself.

Alternatively, custom formatting can be realized through formatting functions. A formatting function is a method that can be invoked from the DataList or Repeater s templates that returns the HTML to emit in its place. Often, the HTML returned by a formatting function is determined by the values being bound to the current item. These values can be passed into the formatting function, either as scalar values or by passing in the entire object being bound to the item (such as the `ProductsRow` instance).

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with

Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Yaakov Ellis, Randy Schmidt, and Liz Shulok. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Showing Multiple Records per Row with the DataList Control (C#)

2/20/2020 • 6 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this short tutorial we'll explore how to customize the DataList's layout through its RepeatColumns and RepeatDirection properties.

Introduction

The DataList examples we've seen in the past two tutorials have rendered each record from its data source as a row in a single-column HTML `<table>`. While this is the default DataList behavior, it is very easy to customize the DataList display such that the data source items are spread across a multi-column, multi-row table. Moreover, it's possible to have all of the data source items displayed in a single-row, multi-column DataList.

We can customize the DataList's layout through its `RepeatColumns` and `RepeatDirection` properties, which, respectively, indicate how many columns are rendered and whether those items are laid out vertically or horizontally. Figure 1, for example, shows a DataList that displays product information in a table with three columns.

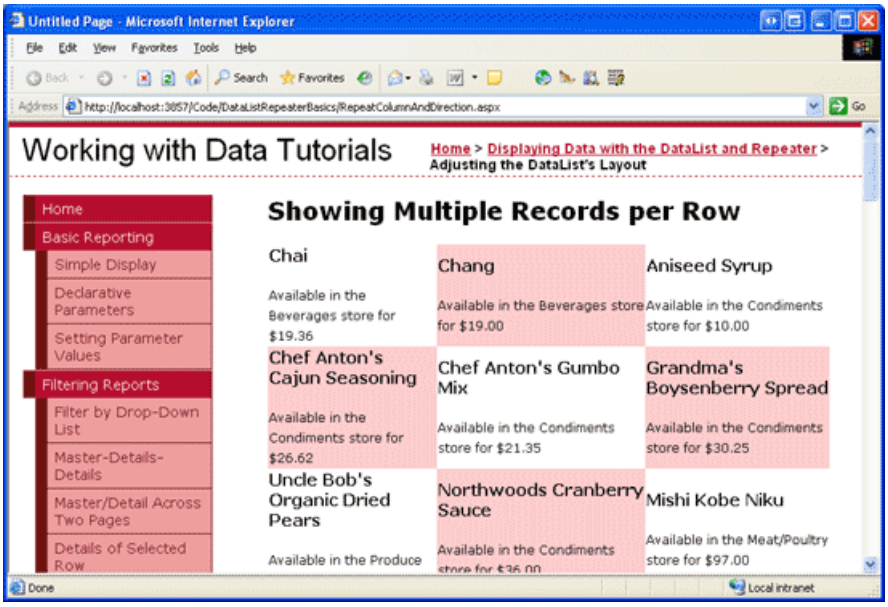


Figure 1: The DataList Shows Three Products per Row (Click to view full-size image)

By showing multiple data source items per row, the DataList can more effectively utilize horizontal screen space. In this short tutorial we'll explore these two DataList properties.

Step 1: Displaying Product Information in a DataList

Before we examine the `RepeatColumns` and `RepeatDirection` properties, let's first create a DataList on our page that lists product information using the standard single-column, multi-row table layout. For this example, let's display the product's name, category, and price using the following markup:

```
<h4>Product Name</h4>
Available in the Category Name store for Price
```

We ve seen how to bind data to a DataList in previous examples, so I'll move through these steps quickly. Start by opening the `RepeatColumnAndDirection.aspx` page in the `DataListRepeaterBasics` folder and drag a DataList from the Toolbox onto the Designer. From the DataList s smart tag, opt to create a new ObjectDataSource and configure it to pull its data from the `ProductsBLL` class s `GetProducts` method, choosing the (None) option from the wizard s INSERT, UPDATE, and DELETE tabs.

After creating and binding the new ObjectDataSource to the DataList, Visual Studio will automatically create an `ItemTemplate` that displays the name and value for each of the product data fields. Adjust the `ItemTemplate` either directly through the declarative markup or from the Edit Templates option in the DataList s smart tag so that it uses the markup shown above, replacing the *Product Name*, *Category Name*, and *Price* text with Label controls that use the appropriate databinding syntax to assign values to their `Text` properties. After updating the `ItemTemplate`, your page s declarative markup should look similar to the following:

```
<asp:DataList ID="DataList1" runat="server" DataKeyField="ProductID"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <ItemTemplate>
        <h4>
            <asp:Label runat="server" ID="ProductNameLabel"
                Text='<%# Eval("ProductName") %>'></asp:Label>
        </h4>
        Available in the
            <asp:Label runat="server" ID="CategoryNameLabel"
                Text='<%# Eval("CategoryName") %>' />
        store for
            <asp:Label runat="server" ID="UnitPriceLabel"
                Text='<%# Eval("UnitPrice", "{0:C}") %>' />
    </ItemTemplate>
</asp:DataList>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
</asp:ObjectDataSource>
```

Notice that I ve included a format specifier in the `Eval` databinding syntax for the `UnitPrice`, formatting the returned value as a currency - `Eval("UnitPrice", "{0:C}").`

Take a moment to visit your page in a browser. As Figure 2 shows, the DataList renders as a single-column, multi-row table of products.

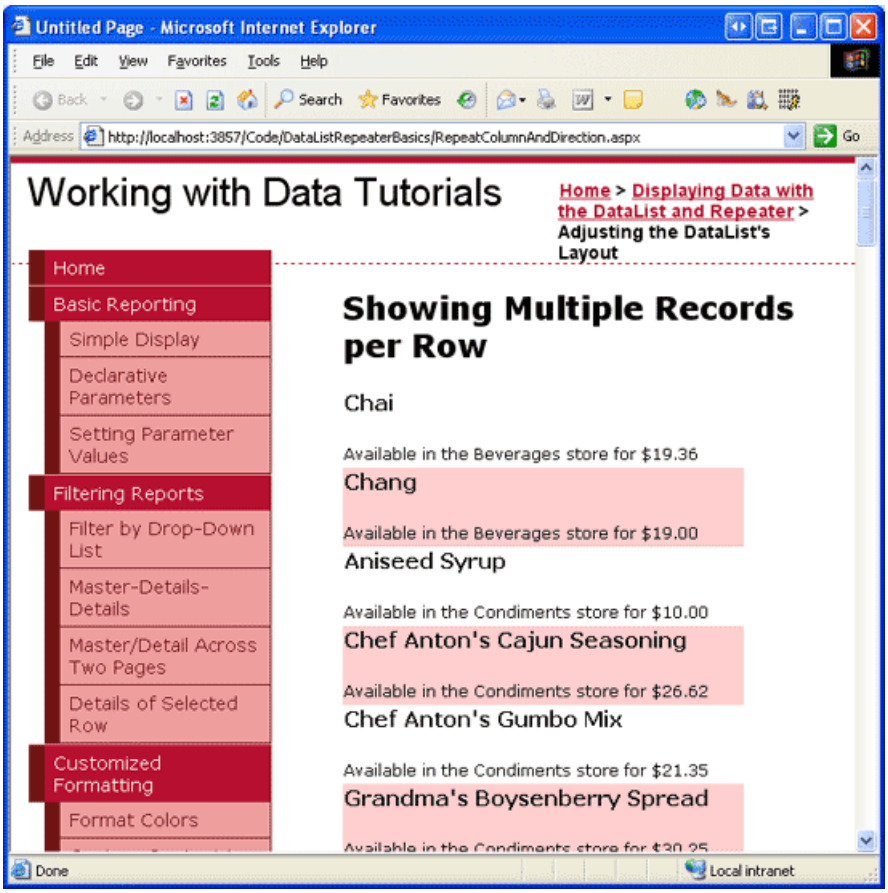


Figure 2: By Default, the DataList Renders as a Single-Column, Multi-Row Table (Click to view full-size image)

Step 2: Changing the DataList s Layout Direction

While the default behavior for the DataList is to lay out its items vertically in a single-column, multi-row table, this behavior can easily be changed through the DataList s `RepeatDirection` property. The `RepeatDirection` property can accept one of two possible values: `Horizontal` or `Vertical` (the default).

By changing the `RepeatDirection` property from `Vertical` to `Horizontal`, the DataList renders its records in a single row, creating one column per data source item. To illustrate this effect, click on the DataList in the Designer and then, from the Properties window, change the `RepeatDirection` property from `Vertical` to `Horizontal`. Immediately upon doing so, the Designer adjusts the DataList s layout, creating a single-row, multi-column interface (see Figure 3).

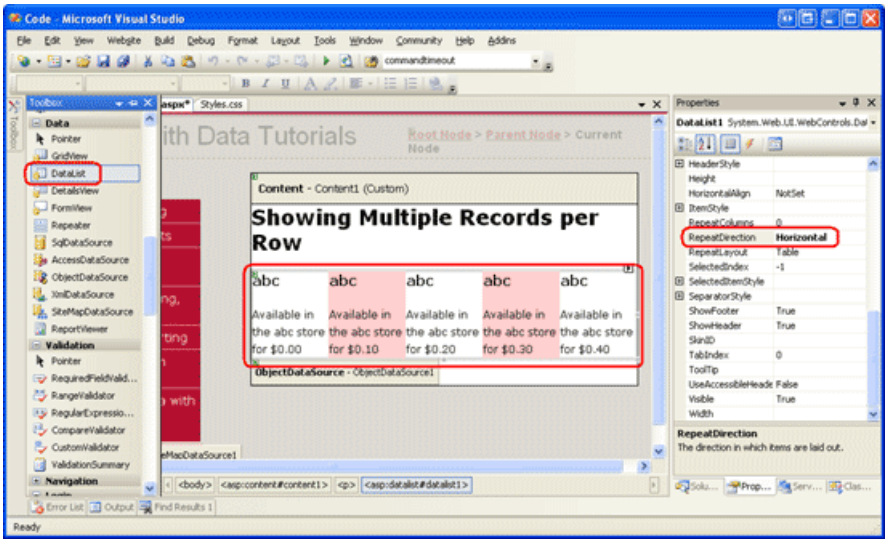


Figure 3: The `RepeatDirection` Property Dictates How the Direction the DataList s Items are Laid Out (Click to view full-size image)

When displaying small amounts of data, a single-row, multi-column table might be an ideal way to maximize screen real estate. For larger volumes of data, however, a single row will require numerous columns, which pushes those items that can t fit on the screen off to the right. Figure 4 shows the products when rendered in a single-row DataList. Since there are many products (over 80), the user will have to scroll far to the right to view information about each of the products.

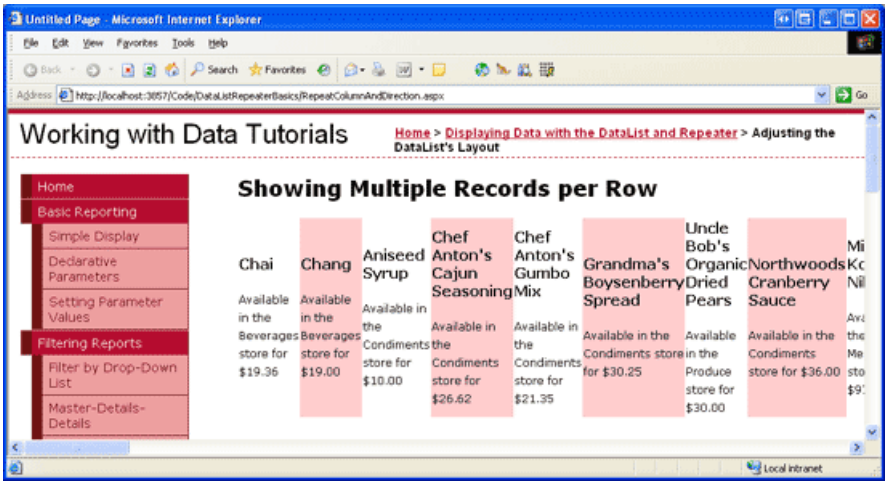


Figure 4: For Sufficiently Large Data Sources, a Single Column DataList Will Require Horizontal Scrolling (Click to view full-size image)

Step 3: Displaying Data in a Multi-Column, Multi-Row Table

To create a multi-column, multi-row DataList, we need to set the `RepeatColumns` property to the number of columns to display. By default, the `RepeatColumns` property is set to 0, which will cause the DataList to display all of its items in a single row or a column (depending on the value of the `RepeatDirection` property).

For our example, let s display three products per table row. Therefore, set the `RepeatColumns` property to 3. After making this change, take a moment to view the results in a browser. As Figure 5 shows, the products are now listed in a three-column, multi-row table.

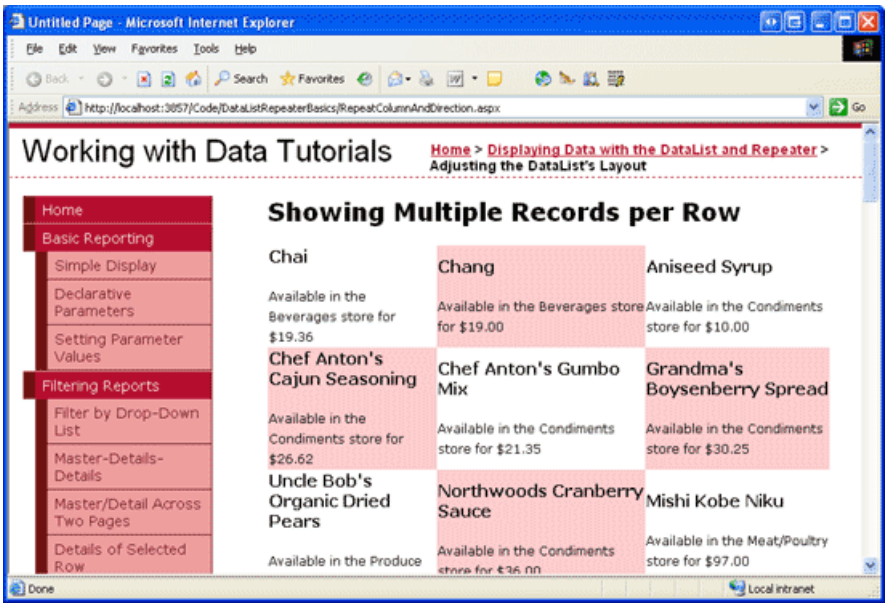


Figure 5: Three Products are Displayed per Row (Click to view full-size image)

The `RepeatDirection` property affects how the items in the DataList are laid out. Figure 5 shows the results with the `RepeatDirection` property set to `Horizontal`. Note that the first three products Chai, Chang, and Aniseed Syrup are laid out from left to right, top to bottom. The next three products (starting with Chef Anton s Cajun Seasoning) appear in a row beneath the first three. Changing the `RepeatDirection` property back to `Vertical`,

however, lays out these products from top to bottom, left to right, as Figure 6 illustrates.

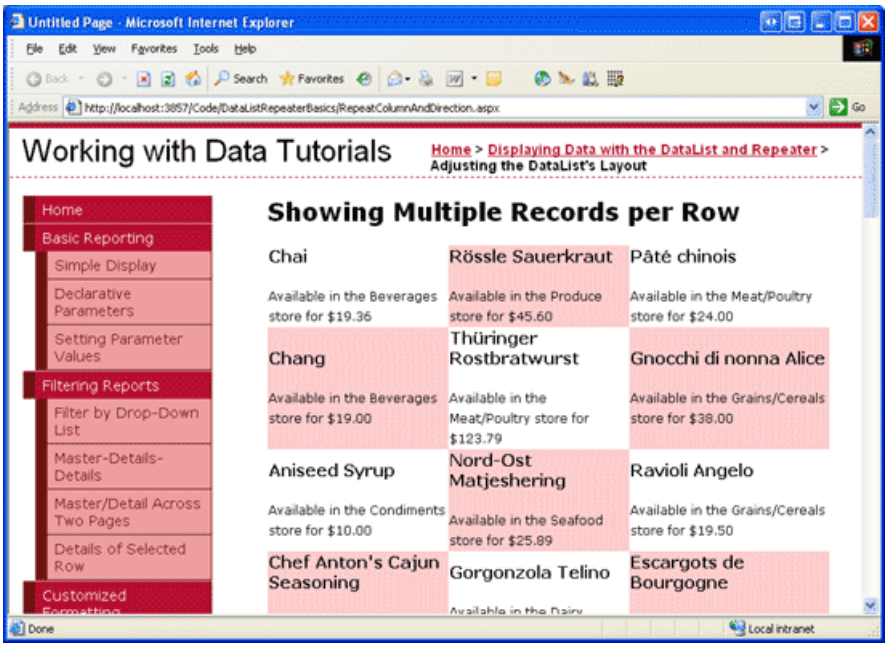


Figure 6: Here, the Products are Laid Out Vertically (Click to view full-size image)

The number of rows displayed in the resulting table depends on the number of total records bound to the DataList. Precisely, it s the ceiling of the total number of data source items divided by the `RepeatColumns` property value. Since the `Products` table currently has 84 products, which is divisible by 3, there are 28 rows. If the number of items in the data source and the `RepeatColumns` property value are not divisible, then the last row or column will have blank cells. If the `RepeatDirection` is set to `Vertical`, then the last column will have empty cells; if `RepeatDirection` is `Horizontal`, then the last row will have the empty cells.

Summary

The DataList, by default, lists its items in a single-column, multi-row table, which mimics the layout of a GridView with a single TemplateField. While this default layout is acceptable, we can maximize screen real estate by displaying multiple data source items per row. Accomplishing this is simply a matter of setting the DataList s `RepeatColumns` property to the number of columns to display per row. Additionally, the DataList s `RepeatDirection` property can be used to indicate whether the contents of the multi-column, multi-row table should be laid out horizontally from left to right, top to bottom or vertically from top to bottom, left to right.

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was John Suru. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Nested Data Web Controls (C#)

2/20/2020 • 13 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

In this tutorial we will explore how to use a Repeater nested inside another Repeater. The examples will illustrate how to populate the inner Repeater both declaratively and programmatically.

Introduction

In addition to static HTML and databinding syntax, templates can also include Web controls and User Controls. These Web controls can have their properties assigned via declarative, databinding syntax, or can be accessed programmatically in the appropriate server-side event handlers.

By embedding controls within a template, the appearance and user experience can be customized and improved upon. For example, in the Using TemplateFields in the GridView Control tutorial, we saw how to customize the GridView s display by adding a Calendar control in a TemplateField to show an employee s hire date; in the Adding Validation Controls to the Editing and Inserting Interfaces and Customizing the Data Modification Interface tutorials, we saw how to customize the editing and inserting interfaces by adding validation controls, TextBoxes, DropDownLists, and other Web controls.

Templates can also contain other data Web controls. That is, we can have a DataList that contains another DataList (or Repeater or GridView or DetailsView, and so on) within its templates. The challenge with such an interface is binding the appropriate data to the inner data Web control. There are a few different approaches available, ranging from declarative options using the ObjectDataSource to programmatic ones.

In this tutorial we will explore how to use a Repeater nested inside another Repeater. The outer Repeater will contain an item for each category in the database, displaying the category s name and description. Each category item s inner Repeater will display information for each product belonging to that category (see Figure 1) in a bulleted list. Our examples will illustrate how to populate the inner Repeater both declaratively and programmatically.

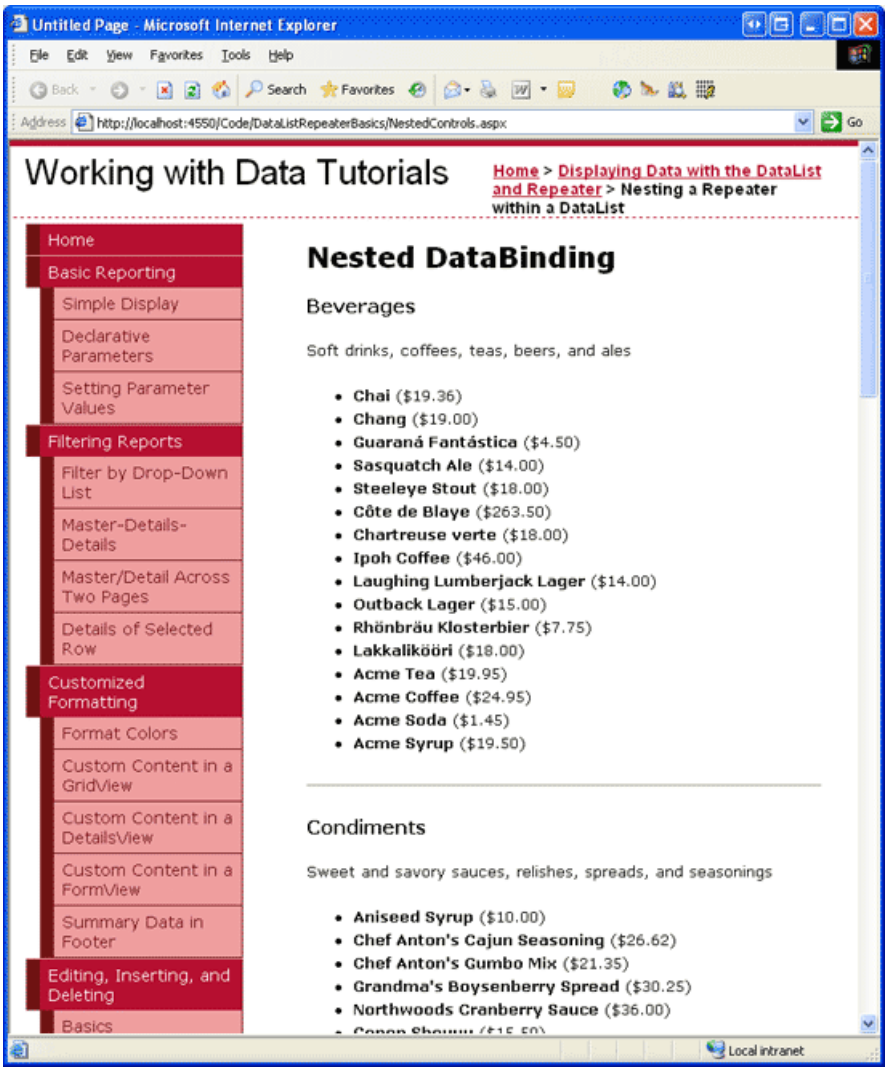


Figure 1: Each Category, Along with its Products, are Listed (Click to view full-size image)

Step 1: Creating the Category Listing

When building a page that uses nested data Web controls, I find it helpful to design, create, and test the outermost data Web control first, without even worrying about the inner nested control. Therefore, let s start by walking through the steps necessary to add a Repeater to the page that lists the name and description for each category.

Start by opening the `NestedControls.aspx` page in the `DataListRepeaterBasics` folder and add a Repeater control to the page, setting its `ID` property to `CategoryList`. From the Repeater s smart tag, choose to create a new ObjectDataSource named `CategoriesDataSource`.

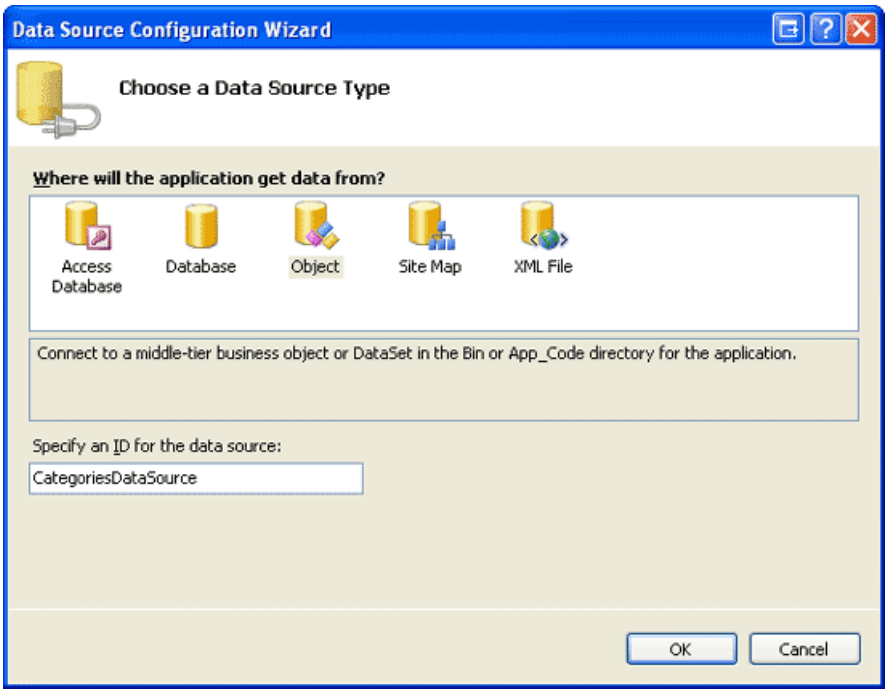


Figure 2: Name the New ObjectDataSource `CategoriesDataSource` (Click to view full-size image)

Configure the ObjectDataSource so that it pulls its data from the `CategoriesBLL` class s `GetCategories` method.

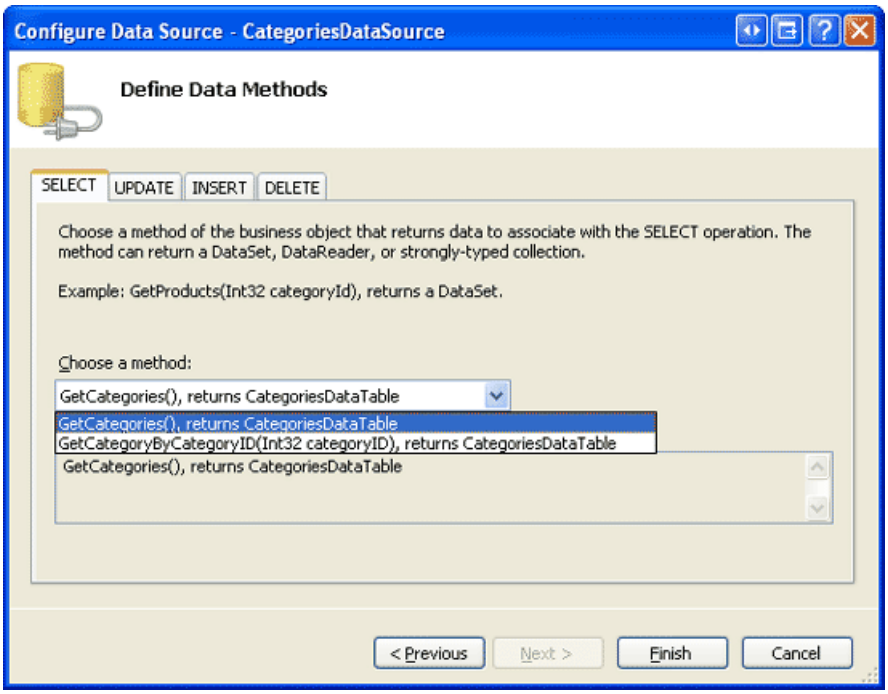


Figure 3: Configure the ObjectDataSource to Use the `CategoriesBLL` Class s `GetCategories` Method (Click to view full-size image)

To specify the Repeater s template content we need to go to the Source view and manually enter the declarative syntax. Add an `ItemTemplate` that displays the category s name in an `<h4>` element and the category s description in a paragraph element (`<p>`). Furthermore, let s separate each category with a horizontal rule (`<hr>`). After making these changes your page should contain declarative syntax for the Repeater and ObjectDataSource that is similar to the following:

```
<asp:Repeater ID="CategoryList" DataSourceID="CategoriesDataSource"
    EnableViewState="False" runat="server">
    <ItemTemplate>
        <h4><%# Eval("CategoryName") %></h4>
        <p><%# Eval("Description") %></p>
    </ItemTemplate>
    <SeparatorTemplate>
        <hr />
    </SeparatorTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

Figure 4 shows our progress when viewed through a browser.

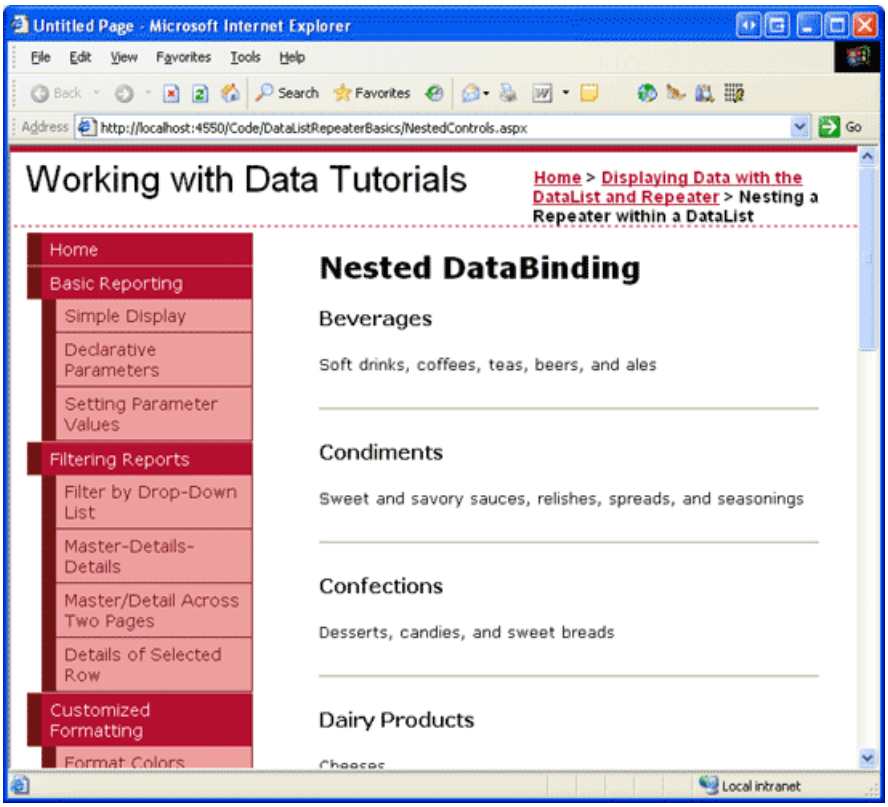


Figure 4: Each Category s Name and Description is Listed, Separated by a Horizontal Rule (Click to view full-size image)

Step 2: Adding the Nested Product Repeater

With the category listing complete, our next task is to add a Repeater to the `CategoryList` s `ItemTemplate` that displays information about those products belonging to the appropriate category. There are a number of ways we can retrieve the data for this inner Repeater, two of which we'll explore shortly. For now, let s just create the products Repeater within the `CategoryList` Repeater s `ItemTemplate`. Specifically, let s have the product Repeater display each product in a bulleted list with each list item including the product s name and price.

To create this Repeater we need to manually enter the inner Repeater s declarative syntax and templates into the `CategoryList` s `ItemTemplate`. Add the following markup within the `CategoryList` Repeater s `ItemTemplate`:

```
<asp:Repeater ID="ProductsByCategoryList" EnableViewState="False"
    runat="server">
    <HeaderTemplate>
        <ul>
    </HeaderTemplate>
    <ItemTemplate>
        <li><strong><%# Eval("ProductName") %></strong>
            (<%# Eval("UnitPrice", "{0:C}") %>)</li>
    </ItemTemplate>
    <FooterTemplate>
        </ul>
    </FooterTemplate>
</asp:Repeater>
```

Step 3: Binding the Category-Specific Products to the ProductsByCategoryList Repeater

If you visit the page through a browser at this point, your screen will look the same as in Figure 4 because we ve yet to bind any data to the Repeater. There are a few ways that we can grab the appropriate product records and bind them to the Repeater, some more efficient than others. The main challenge here is getting back the appropriate products for the specified category.

The data to bind to the inner Repeater control can either be accessed declaratively, through an ObjectDataSource in the `CategoryList` Repeater s `ItemTemplate`, or programmatically, from the ASP.NET page s code-behind page. Similarly, this data can be bound to the inner Repeater either declaratively - through the inner Repeater s `DataSourceID` property or through declarative databinding syntax or programmatically by referencing the inner Repeater in the `CategoryList` Repeater s `ItemDataBound` event handler, programmatically setting its `DataSource` property, and calling its `DataBind()` method. Let s explore each of these approaches.

Accessing the Data Declaratively with an ObjectDataSource Control and the `ItemDataBound` Event Handler

Since we ve used the ObjectDataSource extensively throughout this tutorial series, the most natural choice for accessing data for this example is to stick with the ObjectDataSource. The `ProductsBLL` class has a `GetProductsByCategoryID(categoryID)` method that returns information about those products that belong to the specified *`categoryID`*. Therefore, we can add an ObjectDataSource to the `CategoryList` Repeater s `ItemTemplate` and configure it to access its data from this class s method.

Unfortunately, the Repeater doesn t allow its templates to be edited through the Design view so we need to add the declarative syntax for this ObjectDataSource control by hand. The following syntax shows the `CategoryList` Repeater s `ItemTemplate` after adding this new ObjectDataSource (`ProductsByCategoryDataSource`):

```
<h4><%# Eval("CategoryName") %></h4>
<p><%# Eval("Description") %></p>
<asp:Repeater ID="ProductsByCategoryList" EnableViewState="False"
        DataSourceID="ProductsByCategoryDataSource" runat="server">
    <HeaderTemplate>
        <ul>
    </HeaderTemplate>
    <ItemTemplate>
        <li><strong><%# Eval("ProductName") %></strong> -
                sold as <%# Eval("QuantityPerUnit") %> at
                <%# Eval("UnitPrice", "{0:C}") %></li>
    </ItemTemplate>
    <FooterTemplate>
        </ul>
    </FooterTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="ProductsByCategoryDataSource" runat="server"
           SelectMethod="GetProductsByCategoryID" TypeName="ProductsBLL">
   <SelectParameters>
        <asp:Parameter Name="CategoryID" Type="Int32" />
   </SelectParameters>
</asp:ObjectDataSource>
```

When using the ObjectDataSource approach we need to set the `ProductsByCategoryList` Repeater s `DataSourceID` property to the `ID` of the ObjectDataSource (`ProductsByCategoryDataSource`). Also, notice that our ObjectDataSource has an `<asp:Parameter>` element that specifies the *`categoryID`* value that will be passed into the `GetProductsByCategoryID(categoryID)` method. But how do we specify this value? Ideally, we d be able to just set the `DefaultValue` property of the `<asp:Parameter>` element using databinding syntax, like so:

```
<asp:Parameter Name="CategoryID" Type="Int32"
     DefaultValue='<%# Eval("CategoryID")' />
```

Unfortunately, databinding syntax is only valid in controls that have a `DataBinding` event. The `Parameter` class lacks such an event and therefore the above syntax is illegal and will result in a runtime error.

To set this value, we need to create an event handler for the `CategoryList` Repeater s `ItemDataBound` event. Recall that the `ItemDataBound` event fires once for each item bound to the Repeater. Therefore, each time this event fires for the outer Repeater we can assign the current `CategoryID` value to the `ProductsByCategoryDataSource` ObjectDataSource s `CategoryID` parameter.

Create an event handler for the `CategoryList` Repeater s `ItemDataBound` event with the following code:

```
protected void CategoryList_ItemDataBound(object sender, RepeaterItemEventArgs e)
{
    if (e.Item.ItemType == ListItemType.AlternatingItem ||
        e.Item.ItemType == ListItemType.Item)
    {
        // Reference the CategoriesRow object being bound to this RepeaterItem
        Northwind.CategoriesRow category =
            (Northwind.CategoriesRow)((System.Data.DataRowView)e.Item.DataItem).Row;
        // Reference the ProductsByCategoryDataSource ObjectDataSource
        ObjectDataSource ProductsByCategoryDataSource =
            (ObjectDataSource)e.Item.FindControl("ProductsByCategoryDataSource");
        // Set the CategoryID Parameter value
        ProductsByCategoryDataSource.SelectParameters["CategoryID"].DefaultValue =
            category.CategoryID.ToString();
    }
}
```

This event handler starts by ensuring that we re dealing with a data item rather than the header, footer, or

separator item. Next, we reference the actual `CategoriesRow` instance that has just been bound to the current `RepeaterItem`. Finally, we reference the ObjectDataSource in the `ItemTemplate` and assign its `CategoryID` parameter value to the `CategoryID` of the current `RepeaterItem`.

With this event handler, the `ProductsByCategoryList` Repeater in each `RepeaterItem` is bound to those products in the `RepeaterItem` s category. Figure 5 shows a screen shot of the resulting output.

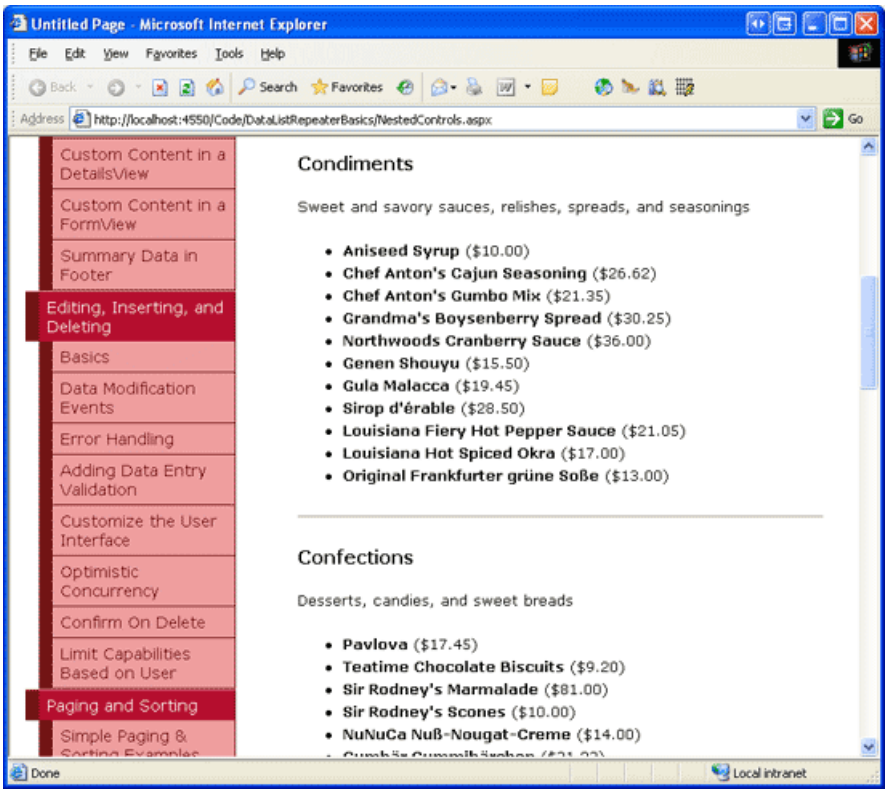


Figure 5: The Outer Repeater Lists Each Category; the Inner One Lists the Products for that Category (Click to view full-size image)

Accessing the Products by Category Data Programmatically

Instead of using an ObjectDataSource to retrieve the products for the current category, we could create a method in our ASP.NET page s code-behind class (or in the `App_Code` folder or in a separate Class Library project) that returns the appropriate set of products when passed in a `CategoryID`. Imagine that we had such a method in our ASP.NET page s code-behind class and that it was named `GetProductsInCategory(categoryID)`. With this method in place we could bind the products for the current category to the inner Repeater using the following declarative syntax:

```
<asp:Repeater runat="server" ID="ProductsByCategoryList" EnableViewState="False"
      DataSource='<%# GetProductsInCategory((int)(Eval("CategoryID"))) %>'>
  ...
</asp:Repeater>
```

The Repeater s `DataSource` property uses the databinding syntax to indicate that its data comes from the `GetProductsInCategory(categoryID)` method. Since `Eval("CategoryID")` returns a value of type `Object`, we cast the object to an `Integer` before passing it into the `GetProductsInCategory(categoryID)` method. Note that the `CategoryID` accessed here via the databinding syntax is the `CategoryID` in the *outer* Repeater (`CategoryList`), the one that s bound to the records in the `Categories` table. Therefore, we know that `CategoryID` cannot be a database `NULL` value, which is why we can blindly cast the `Eval` method without checking if we re dealing with a `DBNull`.

With this approach, we need to create the `GetProductsInCategory(categoryID)` method and have it retrieve the

appropriate set of products given the supplied `categoryID`. We can do this by simply returning the `ProductsDataTable` returned by the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method. Let s create the `GetProductsInCategory(categoryID)` method in the code-behind class for our `NestedControls.aspx` page. Do so using the following code:

```
protected Northwind.ProductsDataTable GetProductsInCategory(int categoryID)
{
    // Create an instance of the ProductsBLL class
    ProductsBLL productAPI = new ProductsBLL();
    // Return the products in the category
    return productAPI.GetProductsByCategoryID(categoryID);
}
```

This method simply creates an instance of the `ProductsBLL` method and returns the results of the `GetProductsByCategoryID(categoryID)` method. Note that the method must be marked `Public` or `Protected`; if the method is marked `Private`, it will not be accessible from the ASP.NET page s declarative markup.

After making these changes to use this new technique, take a moment to view the page through a browser. The output should be identical to the output when using the ObjectDataSource and `ItemDataBound` event handler approach (refer back to Figure 5 to see a screen shot).

NOTE

It may seem like busywork to create the `GetProductsInCategory(categoryID)` method in the ASP.NET page s code-behind class. After all, this method simply creates an instance of the `ProductsBLL` class and returns the results of its `GetProductsByCategoryID(categoryID)` method. Why not just call this method directly from the databinding syntax in the inner Repeater, like: `DataSource='<%# ProductsBLL.GetProductsByCategoryID((int)(Eval("CategoryID"))) %>'`? Although this syntax won't work with our current implementation of the `ProductsBLL` class (since the `GetProductsByCategoryID(categoryID)` method is an instance method), you could modify `ProductsBLL` to include a static `GetProductsByCategoryID(categoryID)` method or have the class include a static `Instance()` method to return a new instance of the `ProductsBLL` class.

While such modifications would eliminate the need for the `GetProductsInCategory(categoryID)` method in the ASP.NET page s code-behind class, the code-behind class method gives us more flexibility in working with the data retrieved, as we'll see shortly.

Retrieving All of the Product Information at Once

The two pervious techniques we ve examined grab those products for the current category by making a call to the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method (the first approach did so through an ObjectDataSource, the second through the `GetProductsInCategory(categoryID)` method in the code-behind class). Each time this method is invoked, the Business Logic Layer calls down to the Data Access Layer, which queries the database with a SQL statement that returns rows from the `Products` table whose `CategoryID` field matches the supplied input parameter.

Given *N* categories in the system, this approach nets *N* + 1 calls to the database one database query to get all of the categories and then *N* calls to get the products specific to each category. We can, however, retrieve all the needed data in just two database calls one call to get all of the categories and another to get all of the products. Once we have all of the products, we can filter those products so that only the products matching the current `CategoryID` are bound to that category s inner Repeater.

To provide this functionality, we only need to make a slight modification to the `GetProductsInCategory(categoryID)` method in our ASP.NET page s code-behind class. Rather than blindly returning the results of the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method, we can instead first access *all* of the products (if they haven't

been accessed already) and then return just the filtered view of the products based on the passed-in `CategoryID`.

```
private Northwind.ProductsDataTable allProducts = null;
protected Northwind.ProductsDataTable GetProductsInCategory(int categoryID)
{
    // First, see if we've yet to have accessed all of the product information
    if (allProducts == null)
    {
        ProductsBLL productAPI = new ProductsBLL();
        allProducts = productAPI.GetProducts();
    }
    // Return the filtered view
    allProducts.DefaultView.RowFilter = "CategoryID = " + categoryID;
    return allProducts;
}
```

Note the addition of the page-level variable, `allProducts`. This holds information about all of the products and is populated the first time the `GetProductsInCategory(categoryID)` method is invoked. After ensuring that the `allProducts` object has been created and populated, the method filters the DataTable s results such that only those rows whose `CategoryID` matches the specified `CategoryID` are accessible. This approach reduces the number of times the database is accessed from $N + 1$ down to two.

This enhancement does not introduce any change to the rendered markup of the page, nor does it bring back fewer records than the other approach. It simply reduces the number of calls to the database.

> **NOTE**
> One might intuitively reason that reducing the number of database accesses would assuredly improve performance. However, this might not be the case. If you have a large number of products whose `CategoryID` is `NULL`, for example, then the call to the `GetProducts` method returns a number of products that are never displayed. Moreover, returning all of the products can be wasteful if you re only showing a subset of the categories, which might be the case if you have implemented paging.

As always, when it comes to analyzing the performance of two techniques, the only surefire measure is to run controlled tests tailored for your application s common case scenarios.

Summary

In this tutorial we saw how to nest one data Web control within another, specifically examining how to have an outer Repeater display an item for each category with an inner Repeater listing the products for each category in a bulleted list. The main challenge in building a nested user interface lies in accessing and binding the correct data to the inner data Web control. There are a variety of techniques available, two of which we examined in this tutorial. The first approach examined used an ObjectDataSource in the outer data Web control s `ItemTemplate` that was bound to the inner data Web control through its `DataSourceID` property. The second technique accessed the data via a method in the ASP.NET page s code-behind class. This method can then be bound to the inner data Web control s `DataSource` property through databinding syntax.

While the nested user interface examined in this tutorial used a Repeater nested within a Repeater, these techniques can be extended to the other data Web controls. You can nest a Repeater within a GridView, or a GridView within a DataList, and so on.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with

Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Zack Jones and Liz Shulok. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Displaying Data with the DataList and Repeater Controls (VB)

2/20/2020 • 19 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In the preceding tutorials we have used the GridView control to display data. Starting with this tutorial, we look at building common reporting patterns with the DataList and Repeater controls, starting with the basics of displaying data with these controls.

Introduction

In all of the examples throughout the past 28 tutorials, if we needed to display multiple records from a data source we turned to the GridView control. The GridView renders a row for each record in the data source, displaying the record s data fields in columns. While the GridView makes it a snap to display, page through, sort, edit, and delete data, its appearance is a bit boxy. Moreover, the markup responsible for the GridView s structure is fixed it includes an HTML `<table>` with a table row (`<tr>`) for each record and a table cell (`<td>`) for each field.

To provide a greater degree of customization in the appearance and rendered markup when displaying multiple records, ASP.NET 2.0 offers the DataList and Repeater controls (both of which were also available in ASP.NET version 1.x). The DataList and Repeater controls render their content using templates rather than BoundFields, CheckBoxFields, ButtonFields, and so on. Like the GridView, the DataList renders as an HTML `<table>`, but allows for multiple data source records to be displayed per table row. The Repeater, on the other hand, renders no additional markup than what you explicitly specify, and is an ideal candidate when you need precise control over the markup emitted.

Over the next dozen or so tutorials, we'll look at building common reporting patterns with the DataList and Repeater controls, starting with the basics of displaying data with these controls templates. We'll see how to format these controls, how to alter the layout of data source records in the DataList, common master/details scenarios, ways to edit and delete data, how to page through records, and so on.

Step 1: Adding the DataList and Repeater Tutorial Web Pages

Before we start this tutorial, let s first take a moment to add the ASP.NET pages we'll need for this tutorial and the next few tutorials dealing with displaying data using the DataList and Repeater. Start by creating a new folder in the project named `DataListRepeaterBasics`. Next, add the following five ASP.NET pages to this folder, having all of them configured to use the master page `Site.master`:

- `Default.aspx`
- `Basics.aspx`
- `Formatting.aspx`
- `RepeatColumnAndDirection.aspx`
- `NestedControls.aspx`

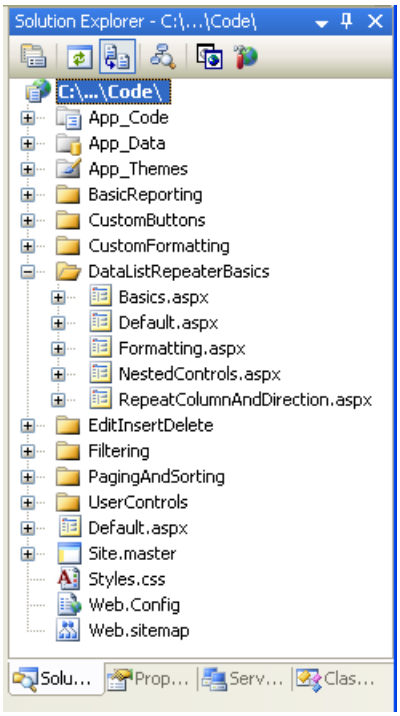


Figure 1: Create a `DataListRepeaterBasics` Folder and Add the Tutorial ASP.NET Pages

Open the `Default.aspx` page and drag the `SectionLevelTutorialListing.ascx` User Control from the `UserControls` folder onto the Design surface. This User Control, which we created in the Master Pages and Site Navigation tutorial, enumerates the site map and displays the tutorials from the current section in a bulleted list.

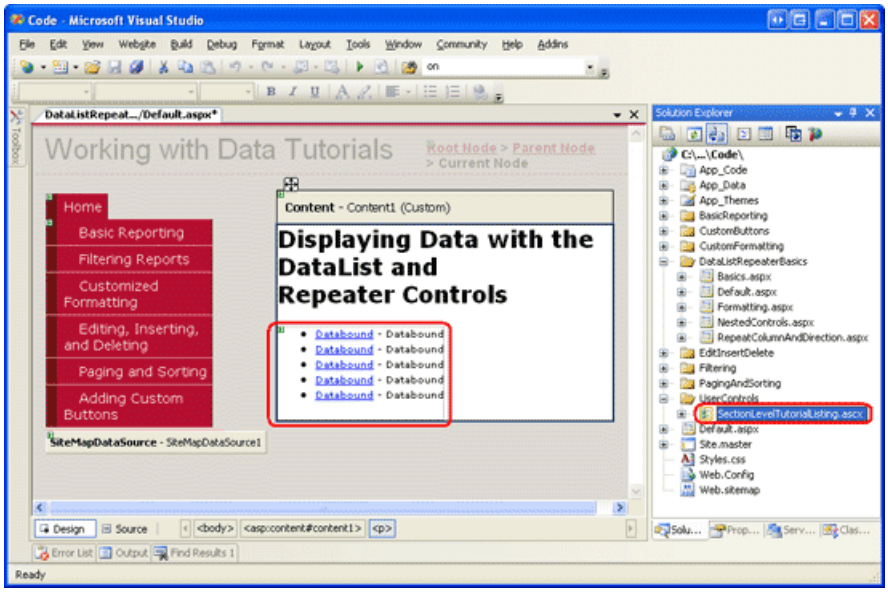


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

In order to have the bulleted list display the DataList and Repeater tutorials we'll be creating, we need to add them to the site map. Open the `Web.sitemap` file and add the following markup after the Adding Custom Buttons site map node markup:

```
<siteMapNode
    title="Displaying Data with the DataList and Repeater"
    description="Samples of Reports that Use the DataList and Repeater Controls"
    url="~/DataListRepeaterBasics/Default.aspx" >
    <siteMapNode
        title="Basic Examples"
        description="Examines the basics for displaying data using the
                     DataList and Repeater controls."
        url="~/DataListRepeaterBasics/Basics.aspx"  />
    <siteMapNode
        title="Formatting"
        description="Learn how to format the DataList and the Web controls within
                     the DataList and Repeater's templates."
        url="~/DataListRepeaterBasics/Formatting.aspx" />
    <siteMapNode
        title="Adjusting the DataList s Layout"
        description="Illustrates how to alter the DataList's layout, showing
                     multiple data source records per table row."
        url="~/DataListRepeaterBasics/RepeatColumnAndDirection.aspx" />
    <siteMapNode
        title="Nesting a Repeater within a DataList"
        description="Learn how to nest a Repeater within the template of a DataList."
        url="~/DataListRepeaterBasics/NestedControls.aspx" />
</siteMapNode>
```

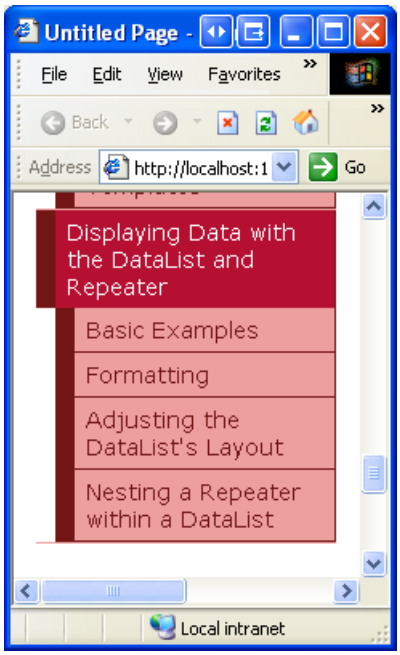


Figure 3: Update the Site Map to Include the New ASP.NET Pages

Step 2: Displaying Product Information with the DataList

Similar to the FormView, the DataList control s rendered output depends upon templates rather than BoundFields, CheckBoxFields, and so on. Unlike the FormView, the DataList is designed to display a set of records rather than a solitary one. Let s begin this tutorial with a look at binding product information to a DataList. Start by opening the `Basics.aspx` page in the `DataListRepeaterBasics` folder. Next, drag a DataList from the Toolbox onto the Designer. As Figure 4 illustrates, before specifying the DataList s templates, the Designer displays it as a gray box.

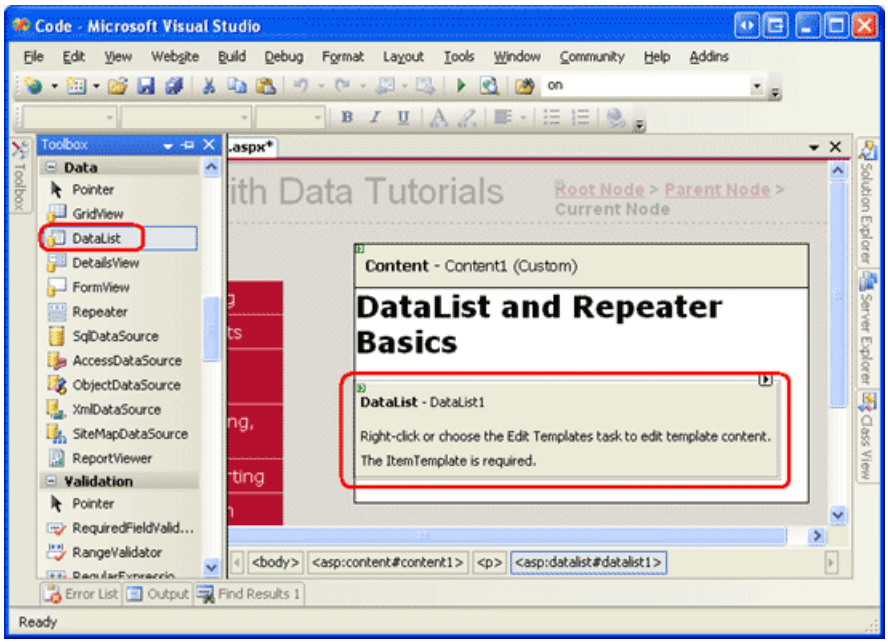


Figure 4: Drag the DataList From the Toolbox Onto the Designer (Click to view full-size image)

From the DataList s smart tag, add a new ObjectDataSource and configure it to use the `ProductsBLL` class s `GetProducts` method. Since we re creating a read-only DataList in this tutorial, set the drop-down list to (None) in the wizard s INSERT, UPDATE, and DELETE tabs.

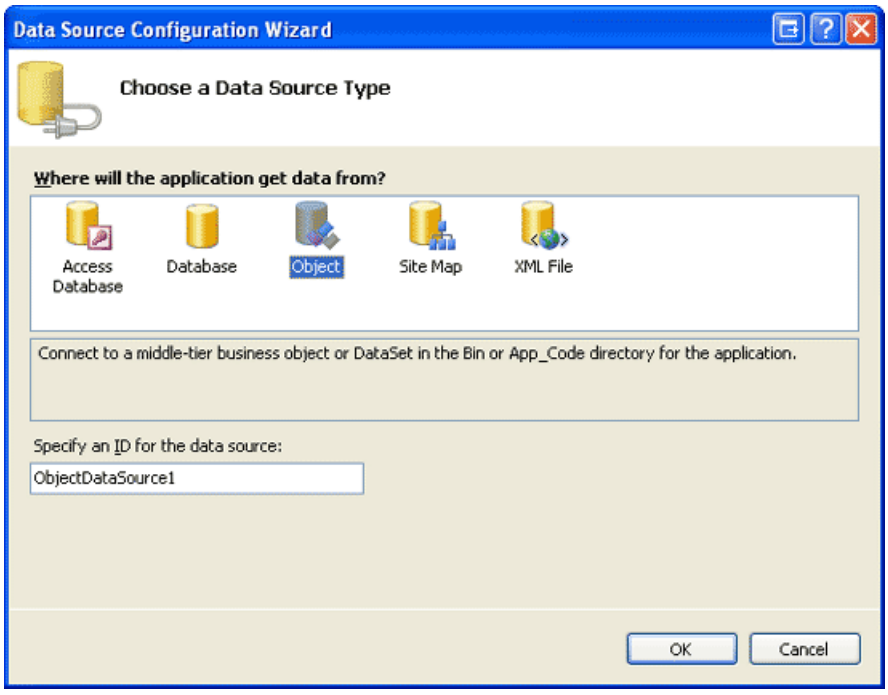


Figure 5: Opt to Create a New ObjectDataSource (Click to view full-size image)

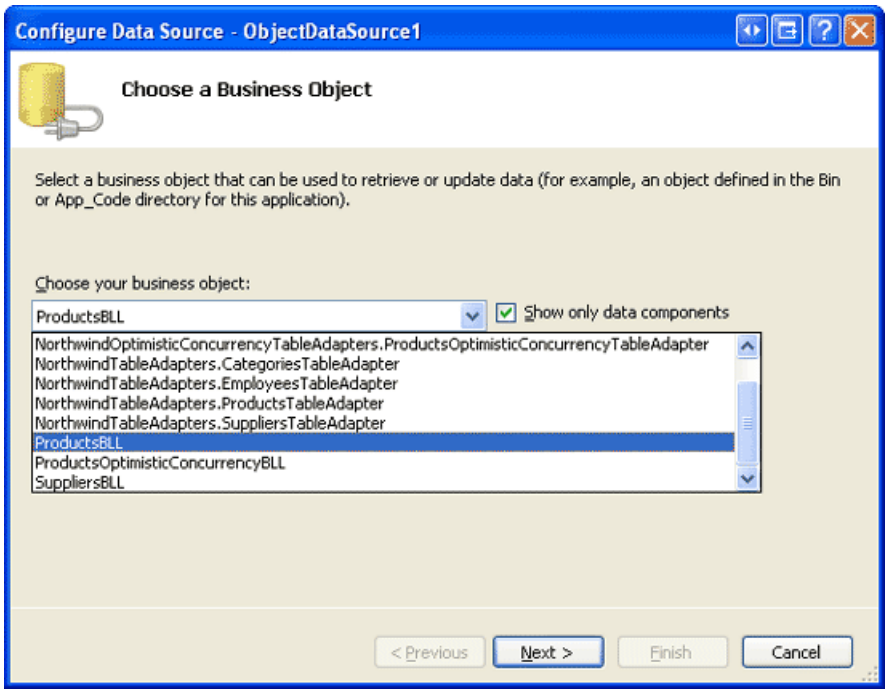


Figure 6: Configure the ObjectDataSource to Use the `ProductsBLL` Class (Click to view full-size image)

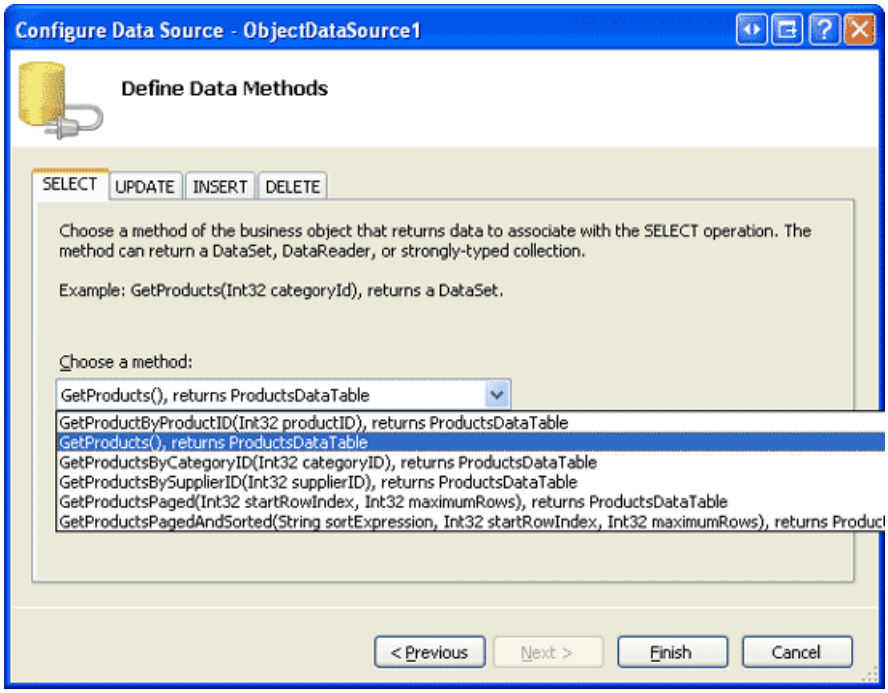


Figure 7: Retrieve Information About All of the Products Using the `GetProducts` Method (Click to view full-size image)

After configuring the ObjectDataSource and associating it with the DataList through its smart tag, Visual Studio will automatically create an `ItemTemplate` in the DataList that displays the name and value of each data field returned by the data source (see the markup below). This default `ItemTemplate` s appearance is identical to that of the templates automatically created when binding a data source to the FormView through the Designer.

```
<asp:DataList ID="DataList1" runat="server" DataKeyField="ProductID"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <ItemTemplate>
        ProductID:       <asp:Label ID="ProductIDLabel" runat="server"
                            Text='<%# Eval("ProductID") %>' /><br />
        ProductName:     <asp:Label ID="ProductNameLabel" runat="server"
                            Text='<%# Eval("ProductName") %>' /><br />
        SupplierID:      <asp:Label ID="SupplierIDLabel" runat="server"
                            Text='<%# Eval("SupplierID") %>' /><br />
        CategoryID:      <asp:Label ID="CategoryIDLabel" runat="server"
                            Text='<%# Eval("CategoryID") %>'/><br />
        QuantityPerUnit: <asp:Label ID="QuantityPerUnitLabel" runat="server"
                            Text='<%# Eval("QuantityPerUnit") %>' /><br />
        UnitPrice:       <asp:Label ID="UnitPriceLabel" runat="server"
                            Text='<%# Eval("UnitPrice") %>' /><br />
        UnitsInStock:    <asp:Label ID="UnitsInStockLabel" runat="server"
                            Text='<%# Eval("UnitsInStock") %>' /><br />
        UnitsOnOrder:    <asp:Label ID="UnitsOnOrderLabel" runat="server"
                            Text='<%# Eval("UnitsOnOrder") %>' /><br />
        ReorderLevel:    <asp:Label ID="ReorderLevelLabel" runat="server"
                            Text='<%# Eval("ReorderLevel") %>' /><br />
        Discontinued:    <asp:Label ID="DiscontinuedLabel" runat="server"
                            Text='<%# Eval("Discontinued") %>' /><br />
        CategoryName:    <asp:Label ID="CategoryNameLabel" runat="server"
                            Text='<%# Eval("CategoryName") %>' /><br />
        SupplierName:    <asp:Label ID="SupplierNameLabel" runat="server"
                            Text='<%# Eval("SupplierName") %>' /><br />
        <br />
    </ItemTemplate>
</asp:DataList>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
</asp:ObjectDataSource>
```

NOTE

Recall that when binding a data source to a FormView control through the FormView s smart tag, Visual Studio created an `ItemTemplate`, `InsertItemTemplate`, and `EditItemTemplate`. With the DataList, however, only an `ItemTemplate` is created. This is because the DataList does not have the same built-in editing and inserting support offered by the FormView. The DataList does contain edit- and delete-related events, and editing and deleting support can be added with a bit of code, but there s no simple out-of-the-box support as with the FormView. We'll see how to include editing and deleting support with the DataList in a future tutorial.

Let s take a moment to improve the appearance of this template. Rather than displaying all of the data fields, let s only display the product s name, supplier, category, quantity per unit, and unit price. Moreover, let s display the name in an `<h4>` heading and lay out the remaining fields using a `<table>` beneath the heading.

To make these changes you can either use the template editing features in the Designer from the DataList s smart tag click on the Edit Templates link or you can modify the template manually through the page s declarative syntax. If you use the Edit Templates option in the Designer, your resulting markup may not match the following markup exactly, but when viewed through a browser should look very similar to the screen shot shown in Figure 8.

```
<asp:DataList ID="DataList1" runat="server" DataKeyField="ProductID"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <ItemTemplate>
        <h4><asp:Label ID="ProductNameLabel" runat="server"
            Text='<%# Eval("ProductName") %>' /></h4>
        <table border="0">
            <tr>
                <td class="ProductPropertyLabel">Category:</td>
                <td><asp:Label ID="CategoryNameLabel" runat="server"
                    Text='<%# Eval("CategoryName") %>' /></td>
                <td class="ProductPropertyLabel">Supplier:</td>
                <td><asp:Label ID="SupplierNameLabel" runat="server"
                    Text='<%# Eval("SupplierName") %>' /></td>
            </tr>
            <tr>
                <td class="ProductPropertyLabel">Qty/Unit:</td>
                <td><asp:Label ID="QuantityPerUnitLabel" runat="server"
                    Text='<%# Eval("QuantityPerUnit") %>' /></td>
                <td class="ProductPropertyLabel">Price:</td>
                <td><asp:Label ID="UnitPriceLabel" runat="server"
                    Text='<%# Eval("UnitPrice", "{0:C}") %>' /></td>
            </tr>
        </table>
    </ItemTemplate>
</asp:DataList>
```

NOTE

The example above uses Label Web controls whose `Text` property is assigned the value of the databinding syntax. Alternatively, we could have omitted the Labels altogether, typing in just the databinding syntax. That is, instead of using `<asp:Label ID="CategoryNameLabel" runat="server" Text='<%# Eval("CategoryName") %>' />` we could have instead used the declarative syntax `<%# Eval("CategoryName") %>`.

Leaving in the Label Web controls, however, offer two advantages. First, it provides an easier means for formatting the data based on the data, as we'll see in the next tutorial. Second, the Edit Templates option in the Designer doesn t display declarative databinding syntax that appears outside of some Web control. Instead, the Edit Templates interface is designed to facilitate working with static markup and Web controls and assumes that any databinding will be done through the Edit DataBindings dialog box, which is accessible from the Web controls smart tags.

Therefore, when working with the DataList, which provides the option of editing the templates through the Designer, I prefer to use Label Web controls so that the content is accessible through the Edit Templates interface. As we'll see shortly, the Repeater requires that the template s contents be edited from the Source view. Consequently, when crafting the Repeater s templates I'll often omit the Label Web controls unless I know I'll need to format the appearance of the data bound text based on programmatic logic.

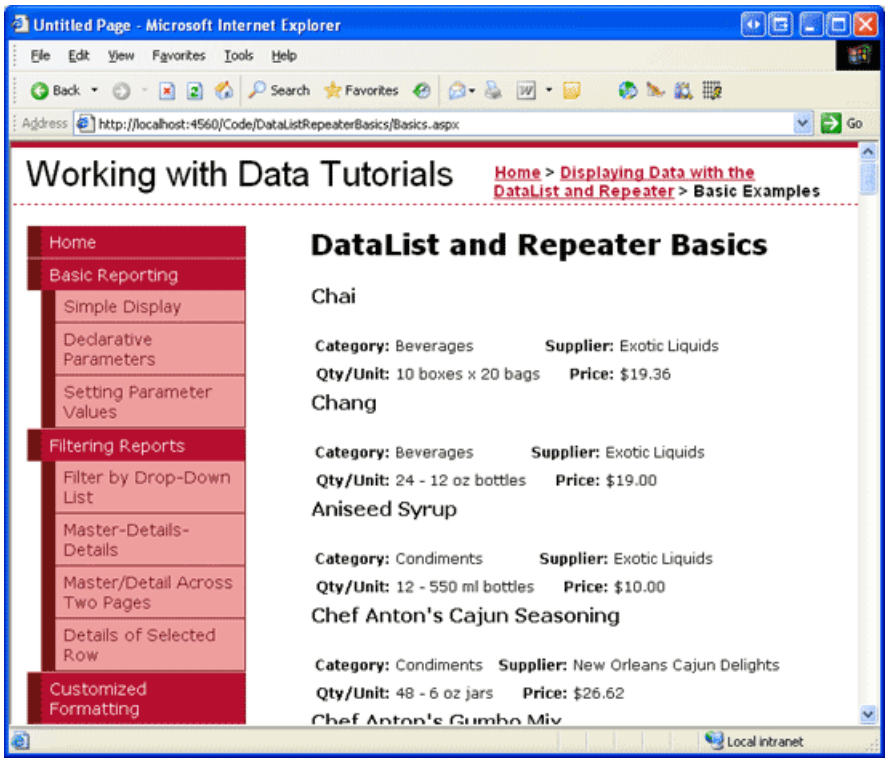


Figure 8: Each Product s Output is Rendered Using the DataList s `ItemTemplate` (Click to view full-size image)

Step 3: Improving the Appearance of the DataList

Like the GridView, the DataList offers a number of style-related properties, such as `Font`, `ForeColor`, `BackColor`, `CssClass`, `ItemStyle`, `AlternatingItemStyle`, `SelectedItemStyle`, and so on. When working with the GridView and DetailsView controls, we created Skin files in the `DataWebControls` Theme that pre-defined the `CssClass` properties for these two controls and the `CssClass` property for several of their subproperties (`RowStyle`, `HeaderStyle`, and so on). Let s do the same for the DataList.

As discussed in the Displaying Data With the ObjectDataSource tutorial, a Skin file specifies the default appearance-related properties for a Web control; a Theme is a collection of Skin, CSS, image, and JavaScript files that define a particular look and feel for a website. In the *Displaying Data With the ObjectDataSource* tutorial, we created a `DataWebControls` Theme (which is implemented as a folder within the `App_Themes` folder) that has, currently, two Skin files - `GridView.skin` and `DetailsView.skin`. Let s add a third Skin file to specify the pre-defined style settings for the DataList.

To add a Skin file, right-click on the `App_Themes/DataWebControls` folder, choose Add a New Item, and select the Skin File option from the list. Name the file `DataList.skin`.

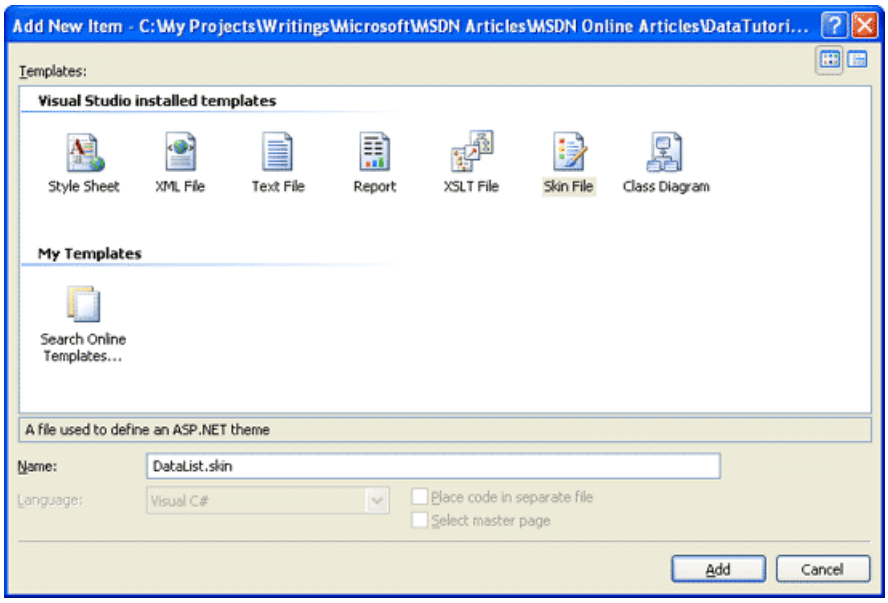


Figure 9: Create a New Skin File Named `DataList.skin` (Click to view full-size image)

Use the following markup for the `DataList.skin` file:

```
<asp:DataList runat="server" CssClass="DataWebControlStyle">
   <AlternatingItemStyle CssClass="AlternatingRowStyle" />
   <ItemStyle CssClass="RowStyle" />
   <HeaderStyle CssClass="HeaderStyle" />
   <FooterStyle CssClass="FooterStyle" />
   <SelectedItemStyle CssClass="SelectedRowStyle" />
</asp:DataList>
```

These settings assign the same CSS classes to the appropriate DataList properties as were used with the GridView and DetailsView controls. The CSS classes used here `DataWebControlStyle`, `AlternatingRowStyle`, `RowStyle`, and so on are defined in the `Styles.css` file and were added in previous tutorials.

With the addition of this Skin file, the DataList s appearance is updated in the Designer (you may need to refresh the Designer view to see the effects of the new Skin file; from the View menu, choose Refresh). As Figure 10 shows, each alternating product has a light pink background color.

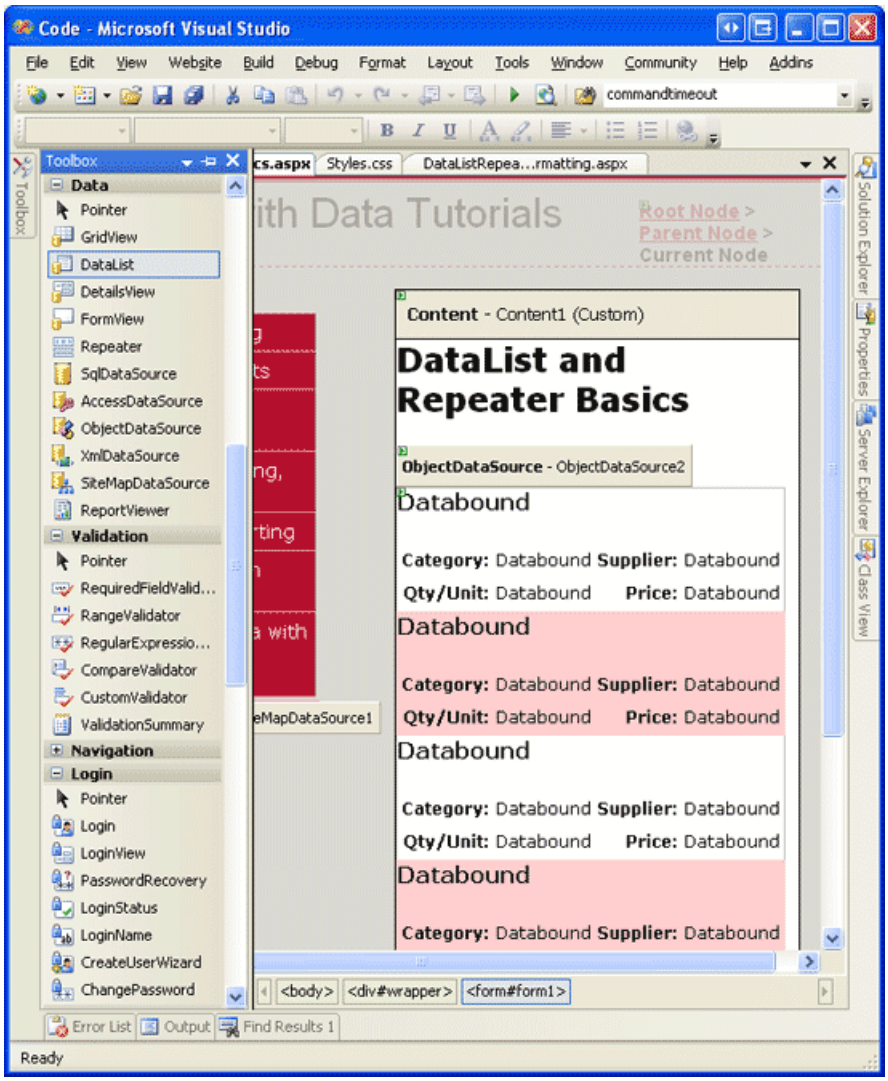


Figure 10: Create a New Skin File Named `DataList.skin` (Click to view full-size image)

Step 4: Exploring the DataList s Other Templates

In addition to the `ItemTemplate`, the DataList supports six other optional templates:

- `HeaderTemplate` if provided, adds a header row to the output and is used to render this row
- `AlternatingItemTemplate` used to render alternating items
- `SelectedItemTemplate` used to render the selected item; the selected item is the item whose index corresponds to the DataList s `SelectedIndex` property
- `EditItemTemplate` used to render the item being edited
- `SeparatorTemplate` if provided, adds a separator between each item and is used to render this separator
- `FooterTemplate` - if provided, adds a footer row to the output and is used to render this row

When specifying the `HeaderTemplate` or `FooterTemplate`, the DataList adds an additional header or footer row to the rendered output. Like with the GridView s header and footer rows, the header and footer in a DataList are not bound to data. Therefore, any databinding syntax in the `HeaderTemplate` or `FooterTemplate` that attempts to access bound data will return a blank string.

NOTE

As we saw in the Displaying Summary Information in the GridView s Footer tutorial, while the header and footer rows don t support databinding syntax, data-specific information can be injected directly into these rows from the GridView s `RowDataBound` event handler. This technique can be used to both calculate running totals or other information from the data bound to the control as well as assign that information to the footer. This same concept can be applied to the DataList and Repeater controls; the only difference is that for the DataList and Repeater create an event handler for the `ItemDataBound` event (instead of for the `RowDataBound` event).

For our example, let s have the title Product Information displayed at the top of the DataList s results in an `<h3>` heading. To accomplish this, add a `HeaderTemplate` with the appropriate markup. From the Designer, this can be accomplished by clicking on the Edit Templates link in the DataList s smart tag, choosing the Header Template from the drop-down list, and typing in the text after picking the Heading 3 option from the style drop-down list (see Figure 11).

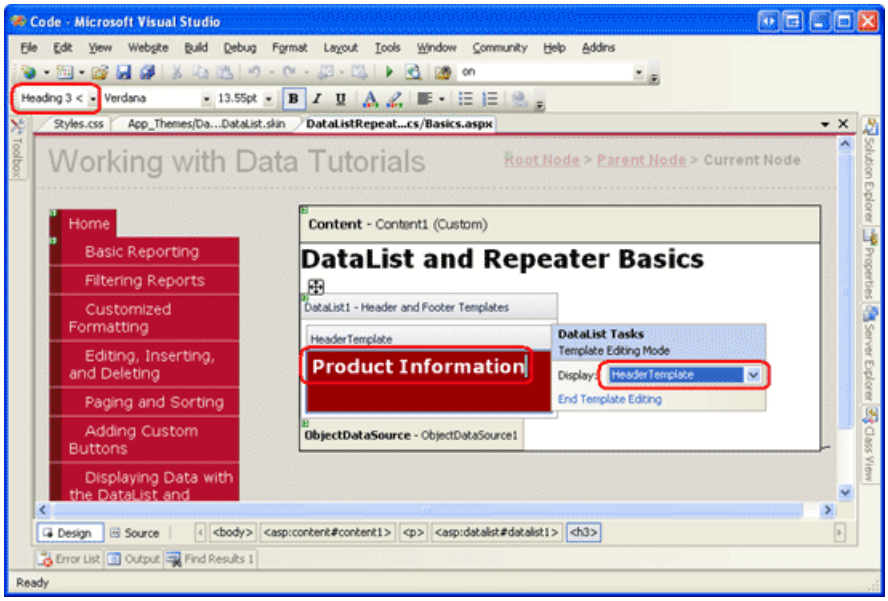


Figure 11: Add a `HeaderTemplate` with the Text Product Information (Click to view full-size image)

Alternatively, this can be added declaratively by entering the following markup within the `<asp:DataList>` tags:

```
<HeaderTemplate>
   <h3>Product Information</h3>
</HeaderTemplate>
```

To add a bit of space between each product listing, let s add a `SeparatorTemplate` that includes a line between each section. The horizontal rule tag (`<hr>`), adds such a divider. Create the `SeparatorTemplate` so that it has the following markup:

```
<SeparatorTemplate>
    <hr />
</SeparatorTemplate>
```

NOTE

Like the `HeaderTemplate` and `FooterTemplates` , the `SeparatorTemplate` is not bound to any record from the data source and therefore cannot directly access the data source records bound to the DataList.

After making this addition, when viewing the page through a browser it should look similar to Figure 12. Note the header row and the line between each product listing.

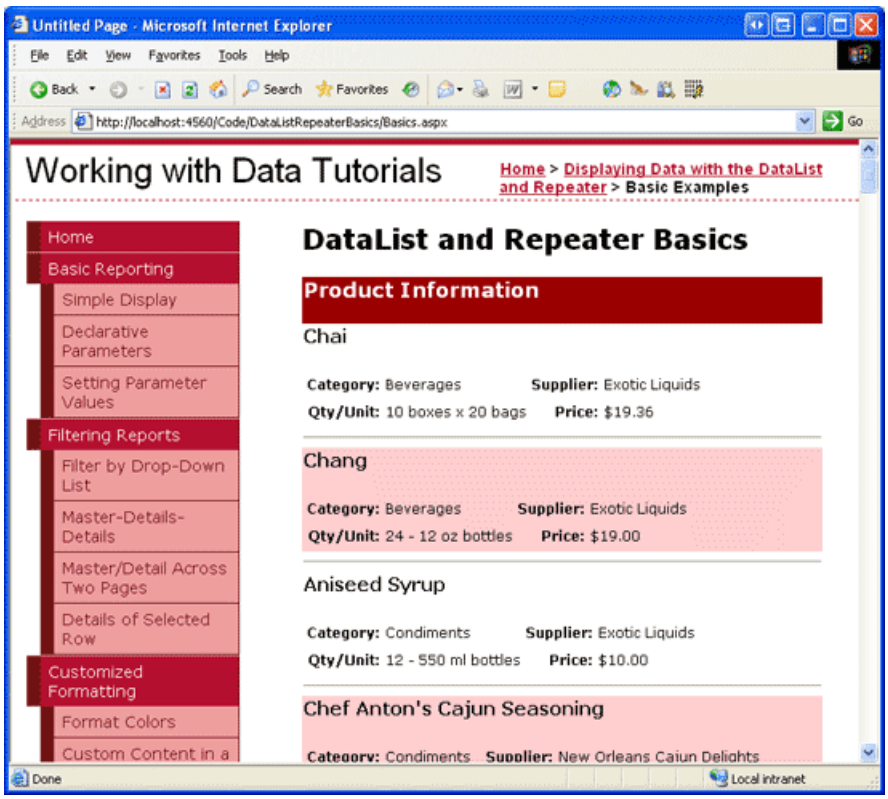


Figure 12: The DataList Includes a Header Row and a Horizontal Rule Between Each Product Listing (Click to view full-size image)

Step 5: Rendering Specific Markup with the Repeater Control

If you do a View/Source from your browser when visiting the DataList example from Figure 12, you'll see that the DataList emits an HTML `<table>` that contains a table row (`<tr>`) with a single table cell (`<td>`) for each item bound to the DataList. This output, in fact, is identical to what would be emitted from a GridView with a single TemplateField. As we'll see in a future tutorial, the DataList does allow further customization of the output, enabling us to display multiple data source records per table row.

What if you don t want to emit an HTML `<table>`, though? For total and complete control over the markup generated by a data Web control, we must use the Repeater control. Like the DataList, the Repeater is constructed based upon templates. The Repeater, however, only offers the following five templates:

- `HeaderTemplate` if provided, adds the specified markup before the items
- `ItemTemplate` used to render items
- `AlternatingItemTemplate` if provided, used to render alternating items
- `SeparatorTemplate` if provided, adds the specified markup between each item
- `FooterTemplate` - if provided, adds the specified markup after the items

In ASP.NET 1.x, the Repeater control was commonly used to display a bulleted list whose data came from some data source. In such a case, the `HeaderTemplate` and `FooterTemplates` would contain the opening and closing `<ul>` tags, respectively, while the `ItemTemplate` would contain `<li>` elements with databinding syntax. This approach can still be used in ASP.NET 2.0 as we saw in two examples in the Master Pages and Site Navigation tutorial:

- In the `Site.master` master page, a Repeater was used to display a bulleted list of the top-level site map contents (Basic Reporting, Filtering Reports, Customized Formatting, and so on); another, nested Repeater was used to display the children sections of the top-level sections

- In `SectionLevelTutorialListing.ascx`, a Repeater was used to display a bulleted list of the children sections of the current site map section

> **NOTE**
>
> ASP.NET 2.0 introduces the new BulletedList control, which can be bound to a data source control in order to display a simple bulleted list. With the BulletedList control we do not need to specify any of the list-related HTML; instead, we simply indicate the data field to display as the text for each list item.

The Repeater serves as a catch all data Web control. If there is not an existing control that generates the needed markup, the Repeater control can be used. To illustrate using the Repeater, let s have the list of categories displayed above the Product Information DataList created in Step 2. In particular, let s have the categories displayed in a single-row HTML `<table>` with each category displayed as a column in the table.

To accomplish this, start by dragging a Repeater control from the Toolbox onto the Designer, above the Product Information DataList. As with the DataList, the Repeater initially displays as a gray box until its templates have been defined.

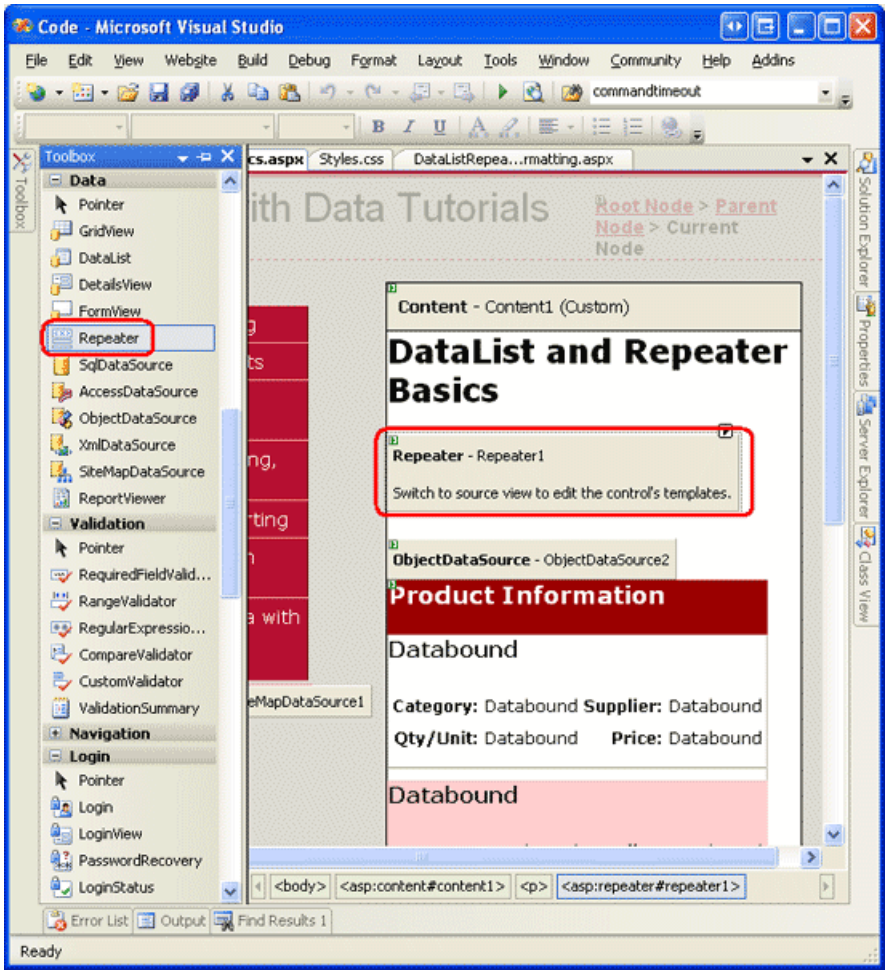


Figure 13: Add a Repeater to the Designer (Click to view full-size image)

There s only one option in the Repeater s smart tag: Choose Data Source. Opt to create a new ObjectDataSource and configure it to use the `CategoriesBLL` class s `GetCategories` method.

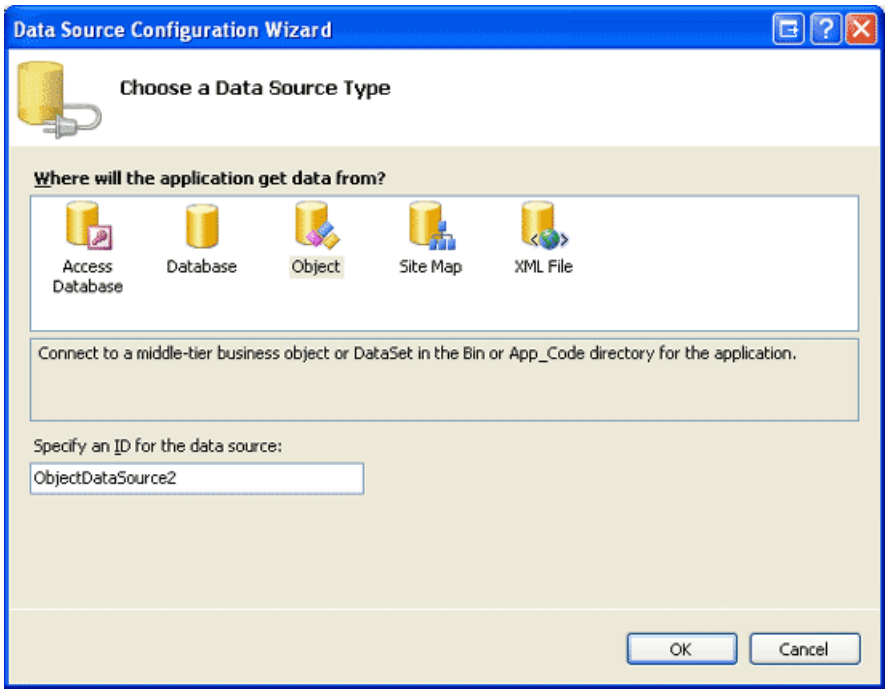


Figure 14: Create a New ObjectDataSource (Click to view full-size image)

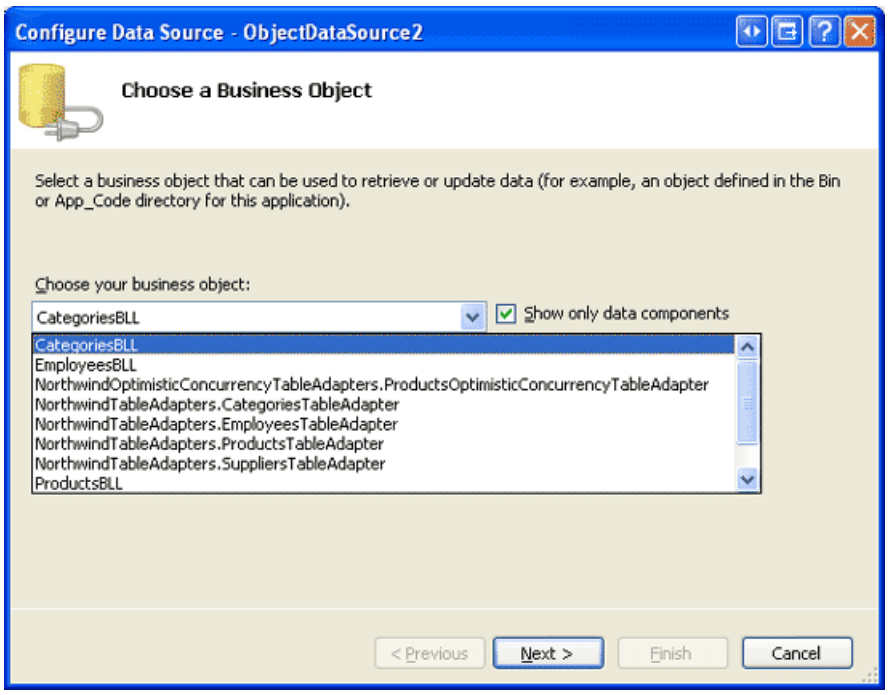


Figure 15: Configure the ObjectDataSource to Use the `CategoriesBLL` Class (Click to view full-size image)

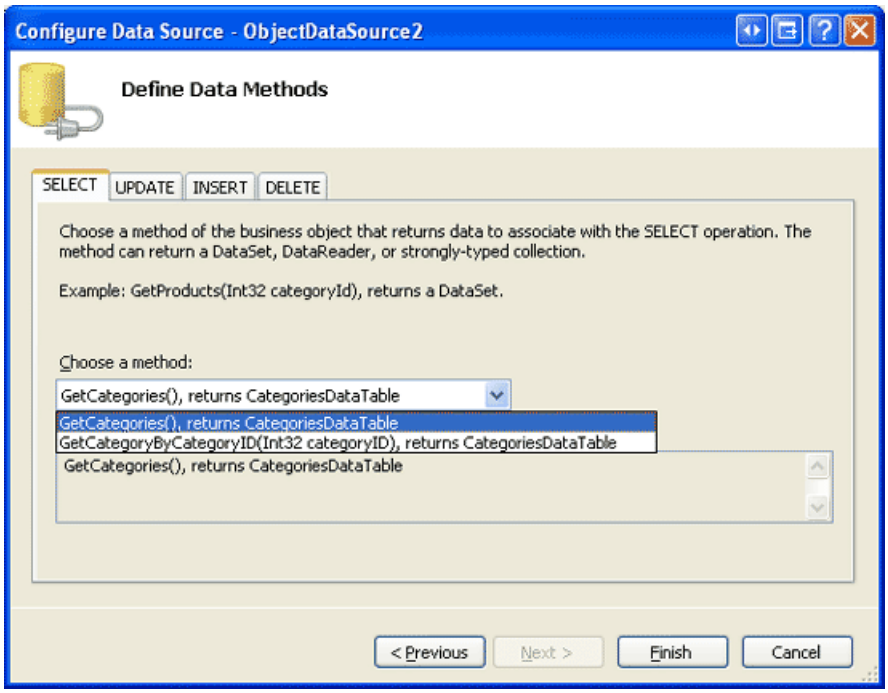


Figure 16: Retrieve Information About All of the Categories Using the `GetCategories` Method (Click to view full-size image)

Unlike the DataList, Visual Studio does not automatically create an ItemTemplate for the Repeater after binding it to a data source. Furthermore, the Repeater s templates cannot be configured through the Designer and must be specified declaratively.

To display the categories as a single-row `<table>` with a column for each category, we need the Repeater to emit markup similar to the following:

```
<table>
   <tr>
      <td>Category 1</td>
      <td>Category 2</td>
      ...
      <td>Category N</td>
   </tr>
</table>
```

Since the `<td>Category X</td>` text is the portion that repeats, this will appear in the Repeater s ItemTemplate. The markup that appears before it - `<table><tr>` - will be placed in the `HeaderTemplate` while the ending markup - `</tr></table>` - will placed in the `FooterTemplate`. To enter these template settings, go to the declarative portion of the ASP.NET page by clicking on the Source button in the lower left corner and type in the following syntax:

```
<asp:Repeater ID="Repeater1" runat="server" DataSourceID="ObjectDataSource2"
    EnableViewState="False">
    <HeaderTemplate>
        <table>
            <tr>
    </HeaderTemplate>
    <ItemTemplate>
                <td><%# Eval("CategoryName") %></td>
    </ItemTemplate>
    <FooterTemplate>
            </tr>
        </table>
    </FooterTemplate>
</asp:Repeater>
```

The Repeater emits the precise markup as specified by its templates, nothing more, nothing less. Figure 17 shows the Repeater s output when viewed through a browser.

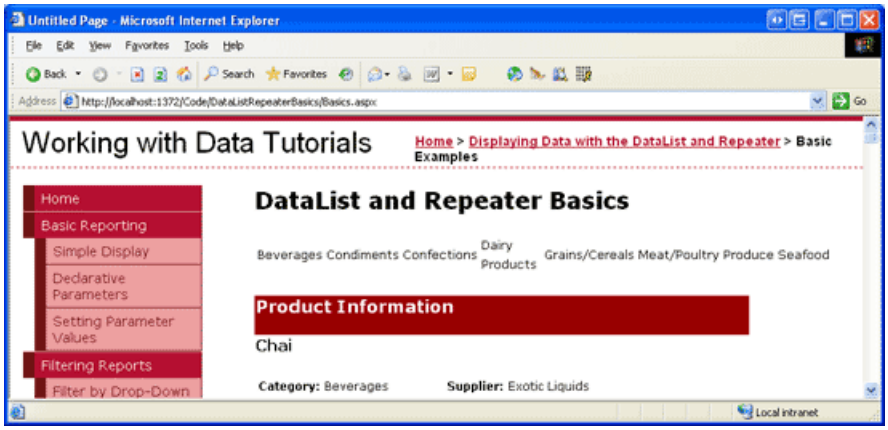


Figure 17: A Single-Row HTML `<table>` Lists Each Category in a Separate Column (Click to view full-size image)

Step 6: Improving the Appearance of the Repeater

Since the Repeater emits precisely the markup specified by its templates, it should come as no surprise that there are no style-related properties for the Repeater. To alter the appearance of the content generated by the Repeater, we must manually add the needed HTML or CSS content directly to the Repeater s templates.

For our example, let s have the category columns alternate background colors, like with the alternating rows in the DataList. To accomplish this, we need to assign the `RowStyle` CSS class to each Repeater item and the `AlternatingRowStyle` CSS class to each alternating Repeater item through the `ItemTemplate` and `AlternatingItemTemplate` templates, like so:

```
<ItemTemplate>
    <td class="RowStyle">&lt;%# Eval("CategoryName") %&gt;</td>
</ItemTemplate>
<AlternatingItemTemplate>
    <td class="AlternatingRowStyle">&lt;%# Eval("CategoryName") %&gt;</td>
</AlternatingItemTemplate>
```

Let s also add a header row to the output with the text Product Categories. Since we don t know how many columns our resulting `<table>` will be comprised of, the simplest way to generate a header row that is guaranteed to span all columns is to use *two* `<table>` s. The first `<table>` will contain two rows the header row and a row that will contain the second, single-row `<table>` that has a column for each category in the system. That is, we want to emit the following markup:

```
<table>
   <tr>
      <th>Product Categories</th>
   </tr>
   <tr>
      <td>
         <table>
            <tr>
               <td>Category 1</td>
               <td>Category 2</td>
               ...
               <td>Category N</td>
            </tr>
         </table>
      </td>
   </tr>
</table>
```

The following `HeaderTemplate` and `FooterTemplate` result in the desired markup:

```
<asp:Repeater ID="Repeater1" runat="server" DataSourceID="ObjectDataSource2"
    EnableViewState="False">
    <HeaderTemplate>
        <table cellpadding="0" cellspacing="0">
            <tr>
                <th class="HeaderStyle">Product Categories</th>
            </tr>
            <tr>
                <td>
                    <table cellpadding="4" cellspacing="0">
                        <tr>
    </HeaderTemplate>
    <ItemTemplate>
                            <td class="RowStyle"><%# Eval("CategoryName") %></td>
    </ItemTemplate>
    <AlternatingItemTemplate>
                            <td class="AlternatingRowStyle">
                                <%# Eval("CategoryName") %></td>
    </AlternatingItemTemplate>
    <FooterTemplate>
                        </tr>
                    </table>
                </td>
            </tr>
        </table>
    </FooterTemplate>
</asp:Repeater>
```

Figure 18 shows the Repeater after these changes have been made.

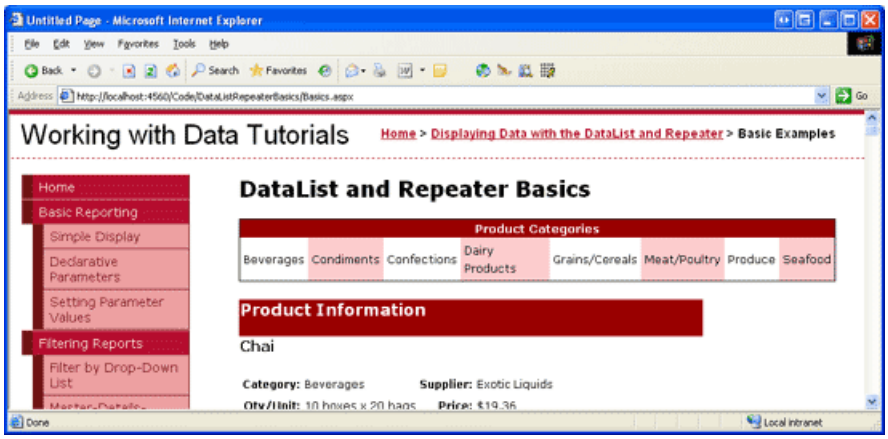


Figure 18: The Category Columns Alternate in Background Color and Includes a Header Row (Click to view full-size image)

Summary

While the GridView control makes it easy to display, edit, delete, sort, and page through data, the appearance is very boxy and grid-like. For more control over the appearance, we need to turn to either the DataList or Repeater controls. Both of these controls display a set of records using templates instead of BoundFields, CheckBoxFields, and so on.

The DataList renders as an HTML `<table>` that, by default, displays each data source record in a single table row, just like a GridView with a single TemplateField. As we will see in a future tutorial, however, the DataList does permit multiple records to be displayed per table row. The Repeater, on the other hand, strictly emits the markup specified in its templates; it does not add any additional markup and therefore is commonly used to display data in HTML elements other than a `<table>` (such as in a bulleted list).

While the DataList and Repeater offer more flexibility in their rendered output, they lack many of the built-in features found in the GridView. As we'll examine in upcoming tutorials, some of these features can be plugged back in without too much effort, but do keep in mind that using the DataList or Repeater in lieu of the GridView does limit the features you can use without having to implement those features yourself.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Yaakov Ellis, Liz Shulok, Randy Schmidt, and Stacy Park. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Formatting the DataList and Repeater Based Upon Data (VB)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll step through examples of how we format the appearance of the DataList and Repeater controls, either by using formatting functions within templates or by handling the DataBound event.

Introduction

As we saw in the preceding tutorial, the DataList offers a number of style-related properties that affect its appearance. In particular, we saw how to assign default CSS classes to the DataList s `HeaderStyle`, `ItemStyle`, `AlternatingItemStyle`, and `SelectedItemStyle` properties. In addition to these four properties, the DataList includes a number of other style-related properties, such as `Font`, `ForeColor`, `BackColor`, and `BorderWidth`, to name a few. The Repeater control does not contain any style-related properties. Any such style settings must be made directly within the markup in the Repeater s templates.

Often, though, how data should be formatted depends on the data itself. For example, when listing products we might want to display the product information in a light gray font color if it is discontinued, or we may want to highlight the `UnitsInStock` value if it is zero. As we saw in previous tutorials, the GridView, DetailsView, and FormView offer two distinct ways to format their appearance based on their data:

- **The `DataBound` event** create an event handler for the appropriate `DataBound` event, which fires after the data has been bound to each item (for the GridView it was the `RowDataBound` event; for the DataList and Repeater it is the `ItemDataBound` event). In that event handler, the data just bound can be examined and formatting decisions made. We examined this technique in the Custom Formatting Based Upon Data tutorial.
- **Formatting Functions in Templates** when using TemplateFields in the DetailsView or GridView controls, or a template in the FormView control, we can add a formatting function to the ASP.NET page s code-behind class, the Business Logic Layer, or any other class library that is accessible from the web application. This formatting function can accept an arbitrary number of input parameters, but must return the HTML to render in the template. Formatting functions were first examined in the Using TemplateFields in the GridView Control tutorial.

Both of these formatting techniques are available with the DataList and Repeater controls. In this tutorial we'll step through examples using both techniques for both controls.

Using the `ItemDataBound` Event Handler

When data is bound to a DataList, either from a data source control or through programmatically assigning data to the control s `DataSource` property and calling its `DataBind()` method, the DataList s `DataBinding` event fires, the data source enumerated, and each data record is bound to the DataList. For each record in the data source, the DataList creates a `DataListItem` object that is then bound to the current record. During this process, the DataList raises two events:

- `ItemCreated` fires after the `DataListItem` has been created
- `ItemDataBound` fires after the current record has been bound to the `DataListItem`

The following steps outline the data binding process for the DataList control.

1. The DataList s `DataBinding` event fires
2. The data is bound to the DataList

   For each record in the data source

   a. Create a `DataListItem` object
   b. Fire the `ItemCreated` event
   c. Bind the record to the `DataListItem`
   d. Fire the `ItemDataBound` event
   e. Add the `DataListItem` to the `Items` collection

When binding data to the Repeater control, it progresses through the exact same sequence of steps. The only difference is that instead of `DataListItem` instances being created, the Repeater uses `RepeaterItem` s.

> **NOTE**
> The astute reader may have noticed a slight anomaly between the sequence of steps that transpire when the DataList and Repeater are bound to data versus when the GridView is bound to data. At the tail end of the data binding process, the GridView raises the `DataBound` event; however, neither the DataList nor Repeater control have such an event. This is because the DataList and Repeater controls were created back in the ASP.NET 1.x timeframe, before the pre- and post-level event handler pattern had become common.

Like with the GridView, one option for formatting based on the data is to create an event handler for the `ItemDataBound` event. This event handler would inspect the data that had just been bound to the `DataListItem` or `RepeaterItem` and affect the formatting of the control as needed.

For the DataList control, formatting changes for the entire item can be implemented using the `DataListItem` s style-related properties, which include the standard `Font` , `ForeColor` , `BackColor` , `CssClass` , and so on. To affect the formatting of particular Web controls within the DataList s template, we need to programmatically access and modify the style of those Web controls. We saw how to accomplish this back in the *Custom Formatting Based Upon Data* tutorial. Like the Repeater control, the `RepeaterItem` class has no style-related properties; therefore, all style-related changes made to a `RepeaterItem` in the `ItemDataBound` event handler must be done by programmatically accessing and updating Web controls within the template.

Since the `ItemDataBound` formatting technique for the DataList and Repeater are virtually identical, our example will focus on using the DataList.

Step 1: Displaying Product Information in the DataList

Before we worry about the formatting, let s first create a page that uses a DataList to display product information. In the previous tutorial we created a DataList whose `ItemTemplate` displayed each product s name, category, supplier, quantity per unit, and price. Let s repeat this functionality here in this tutorial. To accomplish this, you can either recreate the DataList and its ObjectDataSource from scratch, or you can copy over those controls from the page created in the previous tutorial (`Basics.aspx`) and paste them into the page for this tutorial (`Formatting.aspx`).

Once you have replicated the DataList and ObjectDataSource functionality from `Basics.aspx` into `Formatting.aspx` , take a moment to change the DataList s `ID` property from `DataList1` to a more descriptive `ItemDataBoundFormattingExample` . Next, view the DataList in a browser. As Figure 1 shows, the only formatting difference between each product is that the background color alternates.

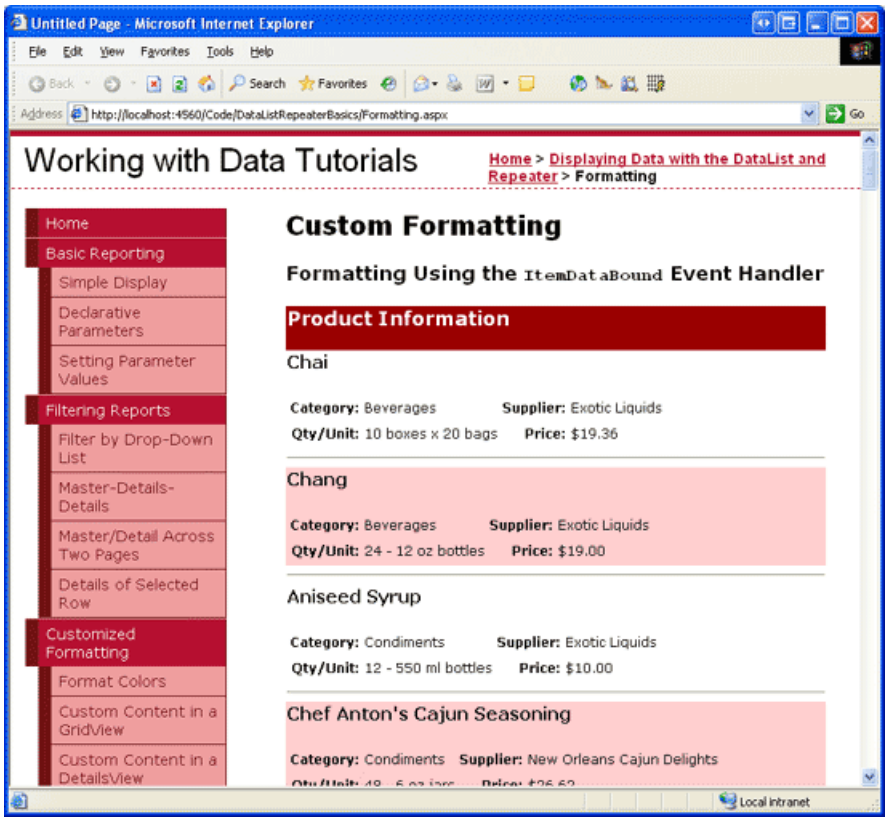


Figure 1: The Products are Listed in the DataList Control (Click to view full-size image)

For this tutorial, let s format the DataList such that any products with a price less than $20.00 will have both its name and unit price highlighted yellow.

Step 2: Programmatically Determining the Value of the Data in the ItemDataBound Event Handler

Since only those products with a price under $20.00 will have the custom formatting applied, we must be able to determine each product s price. When binding data to a DataList, the DataList enumerates the records in its data source and, for each record, creates a `DataListItem` instance, binding the data source record to the `DataListItem`. After the particular record s data has been bound to the current `DataListItem` object, the DataList s `ItemDataBound` event is fired. We can create an event handler for this event to inspect the data values for the current `DataListItem` and, based upon those values, make any formatting changes necessary.

Create an `ItemDataBound` event for the DataList and add the following code:

```
Protected Sub ItemDataBoundFormattingExample_ItemDataBound _
    (sender As Object, e As DataListItemEventArgs) _
    Handles ItemDataBoundFormattingExample.ItemDataBound
    If e.Item.ItemType = ListItemType.Item OrElse _
       e.Item.ItemType = ListItemType.AlternatingItem Then
        ' Programmatically reference the ProductsRow instance
        ' bound to this DataListItem
        Dim product As Northwind.ProductsRow = _
            CType(CType(e.Item.DataItem, System.Data.DataRowView).Row, _
                Northwind.ProductsRow)
        ' See if the UnitPrice is not NULL and less than $20.00
        If Not product.IsUnitPriceNull() AndAlso product.UnitPrice < 20 Then
            ' TODO: Highlight the product's name and price
        End If
    End If
End Sub
```

While the concept and semantics behind the DataList s `ItemDataBound` event handler are the same as those used by the GridView s `RowDataBound` event handler in the *Custom Formatting Based Upon Data* tutorial, the syntax differs slightly. When the `ItemDataBound` event fires, the `DataListItem` just bound to data is passed into corresponding event handler via `e.Item` (instead of `e.Row`, as with the GridView s `RowDataBound` event handler). The DataList s `ItemDataBound` event handler fires for *each* row added to the DataList, including header rows, footer rows, and separator rows. However, the product information is only bound to the data rows. Therefore, when using the `ItemDataBound` event to inspect the data bound to the DataList, we need to first ensure that we re working with a data item. This can be accomplished by checking the `DataListItem` s `ItemType` property, which can have one of the following eight values:

- `AlternatingItem`
- `EditItem`
- `Footer`
- `Header`
- `Item`
- `Pager`
- `SelectedItem`
- `Separator`

Both `Item` and `AlternatingItem``DataListItem` s makeup the DataList s data items. Assuming we re working with an `Item` or `AlternatingItem`, we access the actual `ProductsRow` instance that was bound to the current `DataListItem`. The `DataListItem` s `DataItem` property contains a reference to the `DataRowView` object, whose `Row` property provides a reference to the actual `ProductsRow` object.

Next, we check the `ProductsRow` instance s `UnitPrice` property. Since the Products table s `UnitPrice` field allows `NULL` values, before attempting to access the `UnitPrice` property we should first check to see if it has a `NULL` value using the `IsUnitPriceNull()` method. If the `UnitPrice` value is not `NULL`, we then check to see if it s less than $20.00. If it is indeed under $20.00, we then need to apply the custom formatting.

Step 3: Highlighting the Product s Name and Price

Once we know that a product s price is less than $20.00, all that remains is to highlight its name and price. To accomplish this, we must first programmatically reference the Label controls in the `ItemTemplate` that display the product s name and price. Next, we need to have them display a yellow background. This formatting information can be applied by directly modifying the Labels `BackColor` properties (`LabelID.BackColor = Color.Yellow`); ideally, though, all display-related matters should be expressed through cascading stylesheets. In fact, we already have a stylesheet that provides the desired formatting defined in `Styles.css` - `AffordablePriceEmphasis`, which was created and discussed in the *Custom Formatting Based Upon Data* tutorial.

To apply the formatting, simply set the two Label Web controls `CssClass` properties to `AffordablePriceEmphasis`, as shown in the following code:

```
' Highlight the product name and unit price Labels
' First, get a reference to the two Label Web controls
Dim ProductNameLabel As Label = CType(e.Item.FindControl("ProductNameLabel"), Label)
Dim UnitPriceLabel As Label = CType(e.Item.FindControl("UnitPriceLabel"), Label)
' Next, set their CssClass properties
If ProductNameLabel IsNot Nothing Then
    ProductNameLabel.CssClass = "AffordablePriceEmphasis"
End If
If UnitPriceLabel IsNot Nothing Then
    UnitPriceLabel.CssClass = "AffordablePriceEmphasis"
End If
```

With the `ItemDataBound` event handler completed, revisit the `Formatting.aspx` page in a browser. As Figure 2 illustrates, those products with a price under $20.00 have both their name and price highlighted.

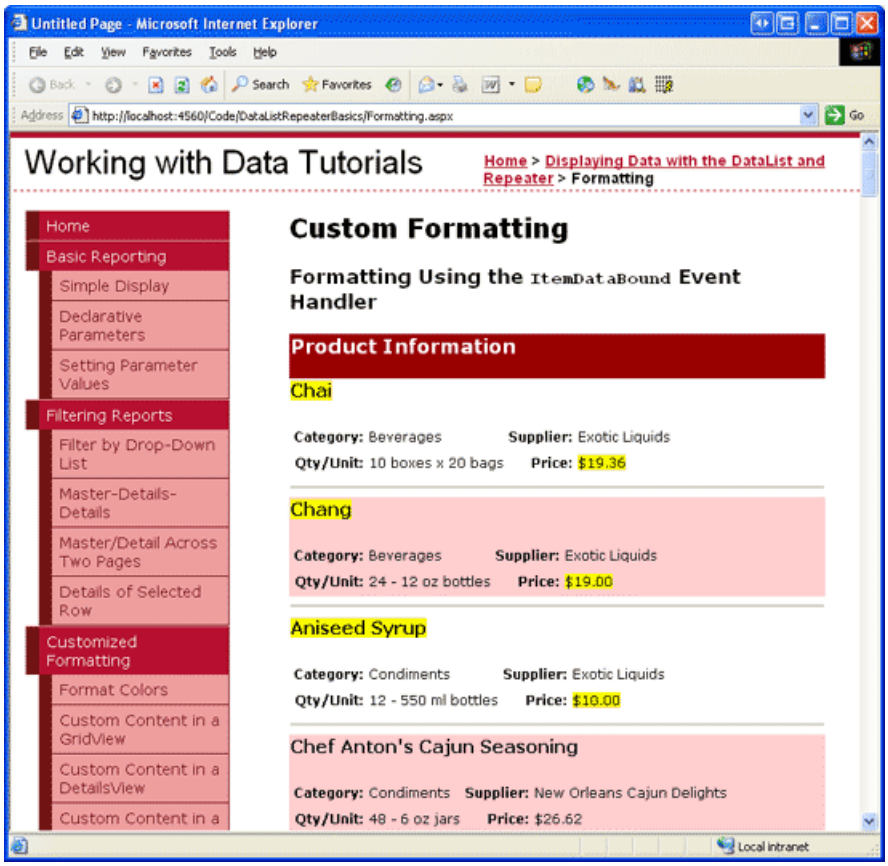


Figure 2: Those Products Less than $20.00 are Highlighted (Click to view full-size image)

> **NOTE**
>
> Since the DataList is rendered as an HTML `<table>`, its `DataListItem` instances have style-related properties that can be set to apply a specific style to the entire item. For example, if we wanted to highlight the *entire* item yellow when its price was less than $20.00, we could have replaced the code that referenced the Labels and set their `CssClass` properties with the following line of code: `e.Item.CssClass = "AffordablePriceEmphasis"` (see Figure 3).

The `RepeaterItem` s that make up the Repeater control, however, don t offer such style-level properties. Therefore, applying custom formatting to the Repeater requires the application of style properties to the Web controls within the Repeater s templates, just like we did in Figure 2.

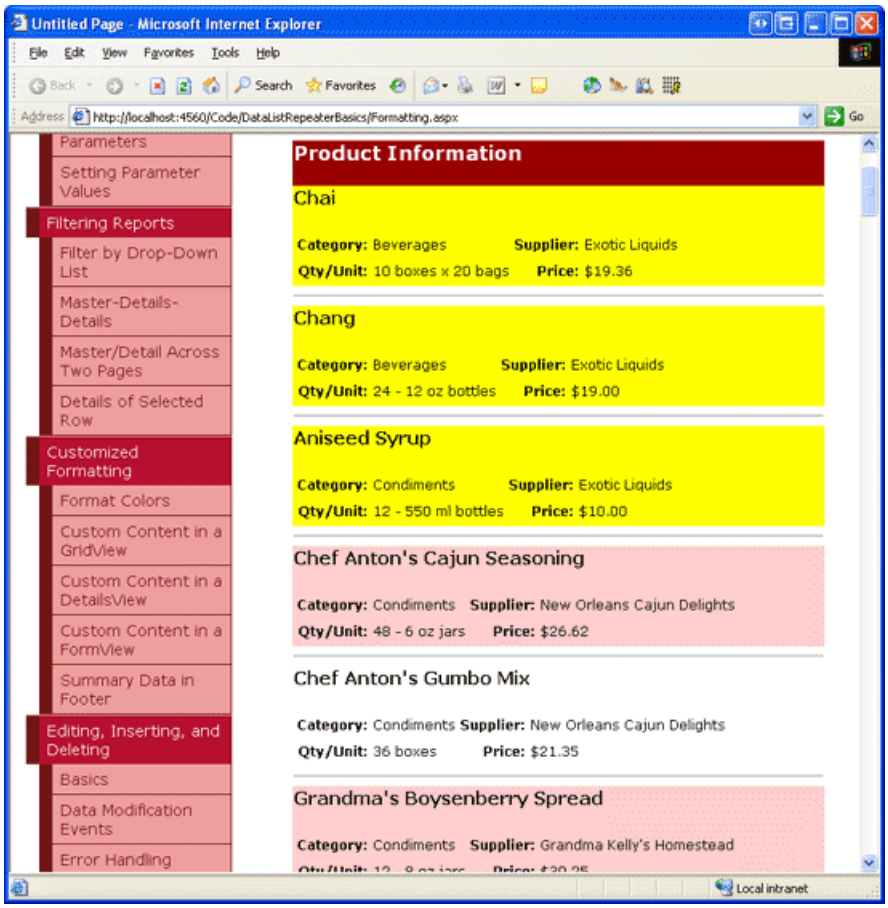


Figure 3: The Entire Product Item is Highlighted for Products Under $20.00 (Click to view full-size image)

Using Formatting Functions from Within the Template

In the *Using TemplateFields in the GridView Control* tutorial we saw how to use a formatting function within a GridView TemplateField to apply custom formatting based upon the data bound to the GridView s rows. A formatting function is a method that can be invoked from a template and returns the HTML to be emitted in its place. Formatting functions can reside in the ASP.NET page s code-behind class or can be centralized into class files in the `App_Code` folder or in a separate Class Library project. Moving the formatting function out of the ASP.NET page s code-behind class is ideal if you plan on using the same formatting function in multiple ASP.NET pages or in other ASP.NET web applications.

To demonstrate formatting functions, let s have the product information include the text [DISCONTINUED] next to the product s name if it s discontinued. Also, let s have the price highlighted yellow if it s less than $20.00 (as we did in the `ItemDataBound` event handler example); if the price is $20.00 or higher, let s not display the actual price, but instead the text, Please call for a price quote. Figure 4 shows a screen shot of the products listing with these formatting rules applied.

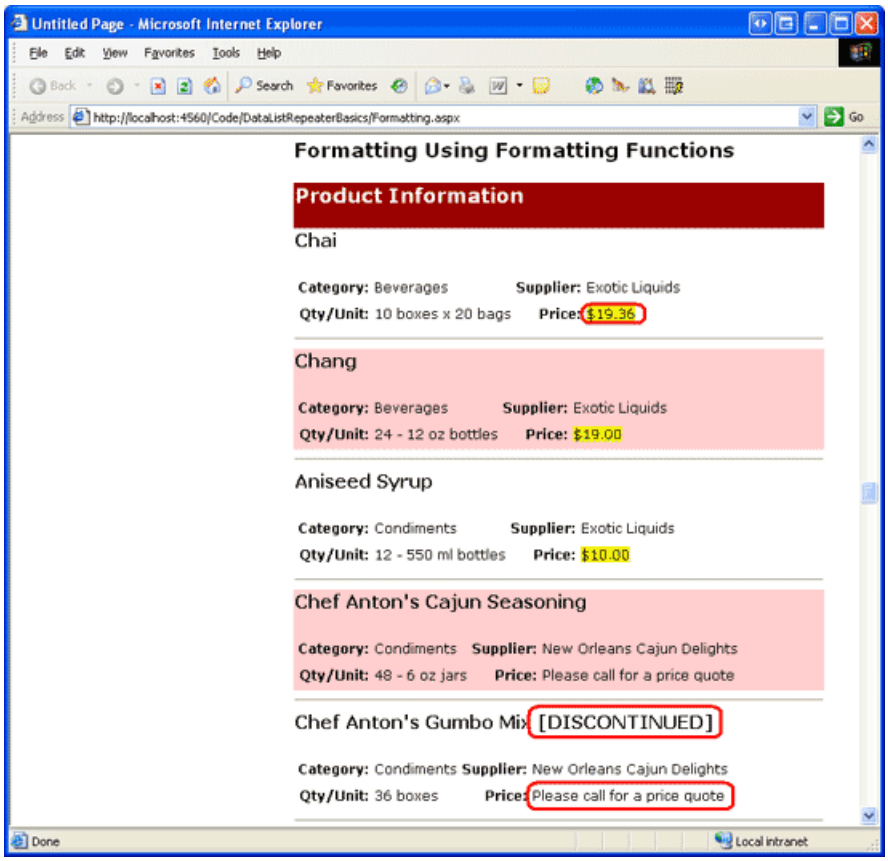


Figure 4: For Expensive Products, the Price is Replaced with the Text, Please call for a price quote (Click to view full-size image)

Step 1: Create the Formatting Functions

For this example we need two formatting functions, one that displays the product name along with the text [DISCONTINUED], if needed, and another that displays either a highlighted price if it s less than $20.00, or the text, Please call for a price quote otherwise. Let s create these functions in the ASP.NET page s code-behind class and name them `DisplayProductNameAndDiscontinuedStatus` and `DisplayPrice`. Both methods need to return the HTML to render as a string and both need to be marked `Protected` (or `Public`) in order to be invoked from the ASP.NET page s declarative syntax portion. The code for these two methods follows:

```
Protected Function DisplayProductNameAndDiscontinuedStatus _
    (productName As String, discontinued As Boolean) As String
    ' Return just the productName if discontinued is false
    If Not discontinued Then
        Return productName
    Else
        ' otherwise, return the productName appended with the text "[DISCONTINUED]"
        Return String.Concat(productName, " [DISCONTINUED]")
    End If
End Function
Protected Function DisplayPrice(product As Northwind.ProductsRow) As String
    ' If price is less than $20.00, return the price, highlighted
    If Not product.IsUnitPriceNull() AndAlso product.UnitPrice < 20 Then
        Return String.Concat("<span class="AffordablePriceEmphasis">", _
                             product.UnitPrice.ToString("C"), "</span>")
    Else
        ' Otherwise return the text, "Please call for a price quote"
        Return "<span>Please call for a price quote</span>"
    End If
End Function
```

Note that the `DisplayProductNameAndDiscontinuedStatus` method accepts the values of the `productName` and

`discontinued` data fields as scalar values, whereas the `DisplayPrice` method accepts a `ProductsRow` instance (rather than a `unitPrice` scalar value). Either approach will work; however, if the formatting function is working with scalar values that can contain database `NULL` values (such as `UnitPrice`; neither `ProductName` nor `Discontinued` allow `NULL` values), special care must be taken in handling these scalar inputs.

In particular, the input parameter must be of type `Object` since the incoming value might be a `DBNull` instance instead of the expected data type. Additionally, a check must be made to determine whether or not the incoming value is a database `NULL` value. That is, if we wanted the `DisplayPrice` method to accept the price as a scalar value, we d have to use the following code:

```
Protected Function DisplayPrice(ByVal unitPrice As Object) As String
    ' If price is less than $20.00, return the price, highlighted
    If Not Convert.IsDBNull(unitPrice) AndAlso CType(unitPrice, Decimal) < 20 Then
        Return String.Concat("<span class="AffordablePriceEmphasis">", _
            CType(unitPrice, Decimal).ToString("C"), "</span>")
    Else
        ' Otherwise return the text, "Please call for a price quote"
        Return "<span>Please call for a price quote</span>"
    End If
End Function
```

Note that the `unitPrice` input parameter is of type `Object` and that the conditional statement has been modified to ascertain if `unitPrice` is `DBNull` or not. Furthermore, since the `unitPrice` input parameter is passed in as an `Object`, it must be cast to a decimal value.

Step 2: Calling the Formatting Function from the DataList s ItemTemplate

With the formatting functions added to our ASP.NET page s code-behind class, all that remains is to invoke these formatting functions from the DataList s `ItemTemplate`. To call a formatting function from a template, place the function call within the databinding syntax:

```
<%# MethodName(inputParameter1, inputParameter2, ...) %>
```

In the DataList s `ItemTemplate` the `ProductNameLabel` Label Web control currently displays the product s name by assigning its `Text` property the result of `<%# Eval("ProductName") %>`. In order to have it display the name plus the text [DISCONTINUED], if needed, update the declarative syntax so that it instead assigns the `Text` property the value of the `DisplayProductNameAndDiscontinuedStatus` method. When doing so, we must pass in the product s name and discontinued values using the `Eval("columnName")` syntax. `Eval` returns a value of type `Object`, but the `DisplayProductNameAndDiscontinuedStatus` method expects input parameters of type `String` and `Boolean`; therefore, we must cast the values returned by the `Eval` method to the expected input parameter types, like so:

```
<h4>
    <asp:Label ID="ProductNameLabel" runat="server"
        Text='<%# DisplayProductNameAndDiscontinuedStatus((string) Eval("ProductName"),
              (bool) Eval("Discontinued")) %>'>
    </asp:Label>
</h4>
```

To display the price, we can simply set the `UnitPriceLabel` Label s `Text` property to the value returned by the `DisplayPrice` method, just like we did for displaying the product s name and [DISCONTINUED] text. However, instead of passing in the `UnitPrice` as a scalar input parameter, we instead pass in the entire `ProductsRow` instance:

```
<asp:Label ID="UnitPriceLabel" runat="server"
    Text='<%# DisplayPrice((Northwind.ProductsRow)
          ((System.Data.DataRowView) Container.DataItem).Row) %>'>
</asp:Label>
```

With the calls to the formatting functions in place, take a moment to view our progress in a browser. Your screen should look similar to Figure 5, with the discontinued products including the text [DISCONTINUED] and those products costing more than $20.00 having their price replaced with the text Please call for a price quote .

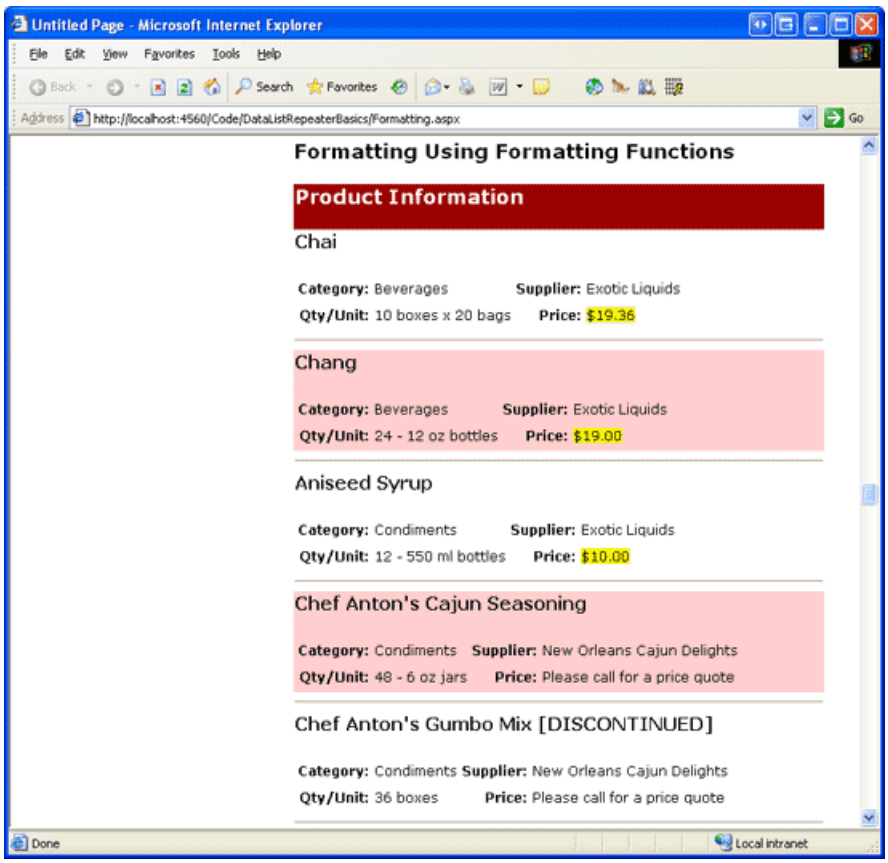


Figure 5: For Expensive Products, the Price is Replaced with the Text, Please call for a price quote (Click to view full-size image)

Summary

Formatting the contents of a DataList or Repeater control based upon the data can be accomplished using two techniques. The first technique is to create an event handler for the `ItemDataBound` event, which fires as each record in the data source is bound to a new `DataListItem` or `RepeaterItem`. In the `ItemDataBound` event handler, the current item s data can be examined and then formatting can be applied to the contents of the template or, for `DataListItem` s, to the entire item itself.

Alternatively, custom formatting can be realized through formatting functions. A formatting function is a method that can be invoked from the DataList or Repeater s templates that returns the HTML to emit in its place. Often, the HTML returned by a formatting function is determined by the values being bound to the current item. These values can be passed into the formatting function, either as scalar values or by passing in the entire object being bound to the item (such as the `ProductsRow` instance).

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with

Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Yaakov Ellis, Randy Schmidt, and Liz Shulok. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Showing Multiple Records per Row with the DataList Control (VB)

2/20/2020 • 6 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this short tutorial we'll explore how to customize the DataList's layout through its RepeatColumns and RepeatDirection properties.

Introduction

The DataList examples we've seen in the past two tutorials have rendered each record from its data source as a row in a single-column HTML `<table>`. While this is the default DataList behavior, it is very easy to customize the DataList display such that the data source items are spread across a multi-column, multi-row table. Moreover, it's possible to have all of the data source items displayed in a single-row, multi-column DataList.

We can customize the DataList's layout through its `RepeatColumns` and `RepeatDirection` properties, which, respectively, indicate how many columns are rendered and whether those items are laid out vertically or horizontally. Figure 1, for example, shows a DataList that displays product information in a table with three columns.

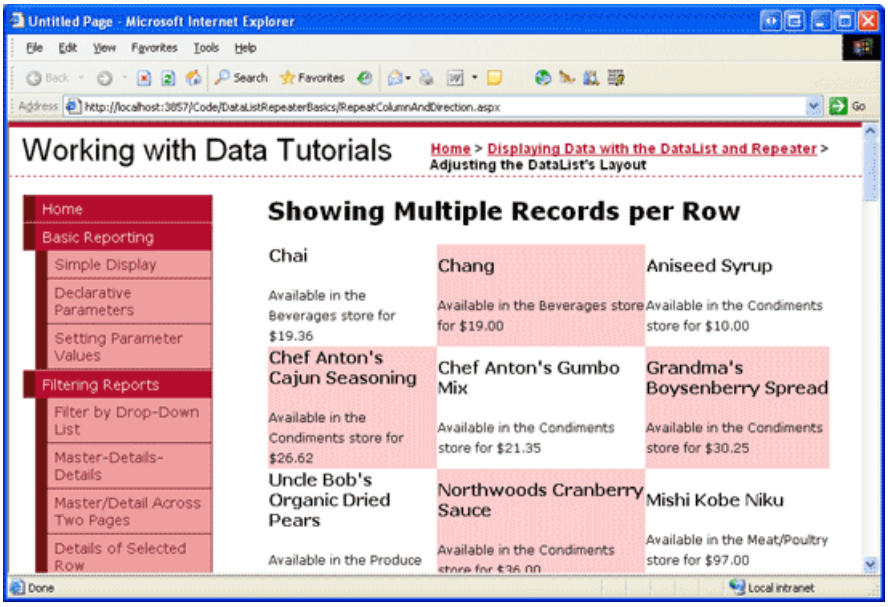


Figure 1: The DataList Shows Three Products per Row (Click to view full-size image)

By showing multiple data source items per row, the DataList can more effectively utilize horizontal screen space. In this short tutorial we'll explore these two DataList properties.

Step 1: Displaying Product Information in a DataList

Before we examine the `RepeatColumns` and `RepeatDirection` properties, let's first create a DataList on our page that lists product information using the standard single-column, multi-row table layout. For this example, let's display the product's name, category, and price using the following markup:

```
<h4>Product Name</h4>
Available in the Category Name store for Price
```

We ve seen how to bind data to a DataList in previous examples, so I'll move through these steps quickly. Start by opening the `RepeatColumnAndDirection.aspx` page in the `DataListRepeaterBasics` folder and drag a DataList from the Toolbox onto the Designer. From the DataList s smart tag, opt to create a new ObjectDataSource and configure it to pull its data from the `ProductsBLL` class s `GetProducts` method, choosing the (None) option from the wizard s INSERT, UPDATE, and DELETE tabs.

After creating and binding the new ObjectDataSource to the DataList, Visual Studio will automatically create an `ItemTemplate` that displays the name and value for each of the product data fields. Adjust the `ItemTemplate` either directly through the declarative markup or from the Edit Templates option in the DataList s smart tag so that it uses the markup shown above, replacing the *Product Name*, *Category Name*, and *Price* text with Label controls that use the appropriate databinding syntax to assign values to their `Text` properties. After updating the `ItemTemplate`, your page s declarative markup should look similar to the following:

```
<asp:DataList ID="DataList1" runat="server" DataKeyField="ProductID"
    DataSourceID="ObjectDataSource1" EnableViewState="False">
    <ItemTemplate>
        <h4>
            <asp:Label runat="server" ID="ProductNameLabel"
                Text='<%# Eval("ProductName") %>'></asp:Label>
        </h4>
        Available in the
            <asp:Label runat="server" ID="CategoryNameLabel"
                Text='<%# Eval("CategoryName") %>' />
        store for
            <asp:Label runat="server" ID="UnitPriceLabel"
                Text='<%# Eval("UnitPrice", "{0:C}") %>' />
    </ItemTemplate>
</asp:DataList>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
</asp:ObjectDataSource>
```

Notice that I ve included a format specifier in the `Eval` databinding syntax for the `UnitPrice`, formatting the returned value as a currency - `Eval("UnitPrice", "{0:C}").`

Take a moment to visit your page in a browser. As Figure 2 shows, the DataList renders as a single-column, multi-row table of products.

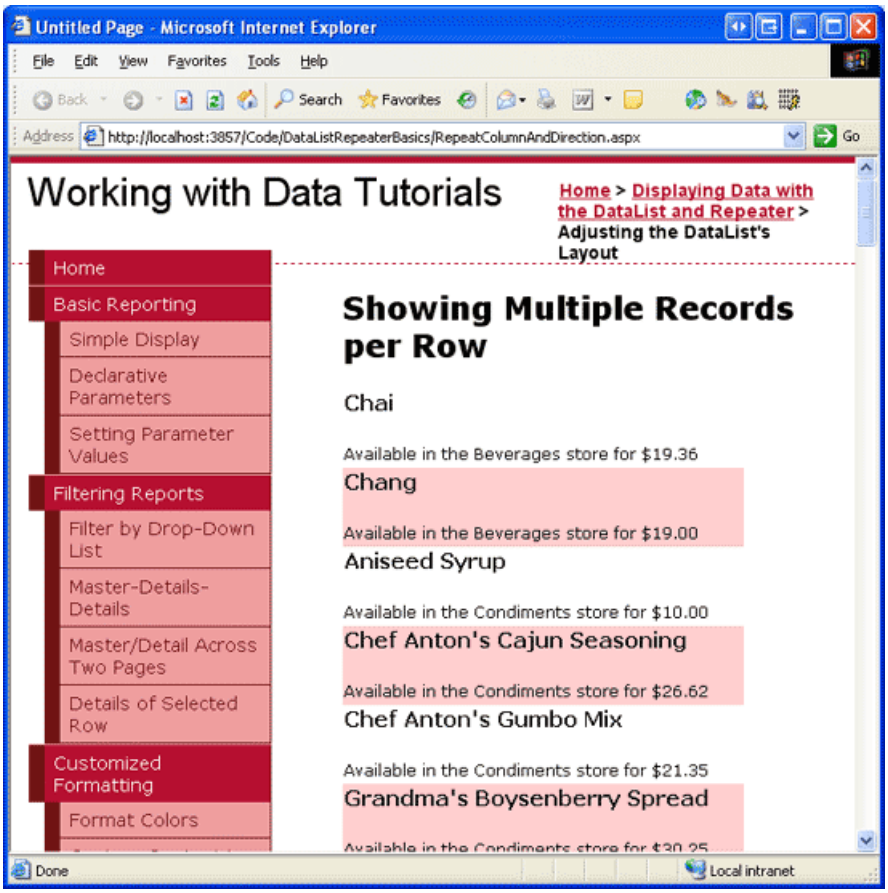


Figure 2: By Default, the DataList Renders as a Single-Column, Multi-Row Table (Click to view full-size image)

Step 2: Changing the DataList s Layout Direction

While the default behavior for the DataList is to lay out its items vertically in a single-column, multi-row table, this behavior can easily be changed through the DataList s `RepeatDirection` property. The `RepeatDirection` property can accept one of two possible values: `Horizontal` or `Vertical` (the default).

By changing the `RepeatDirection` property from `Vertical` to `Horizontal`, the DataList renders its records in a single row, creating one column per data source item. To illustrate this effect, click on the DataList in the Designer and then, from the Properties window, change the `RepeatDirection` property from `Vertical` to `Horizontal`. Immediately upon doing so, the Designer adjusts the DataList s layout, creating a single-row, multi-column interface (see Figure 3).

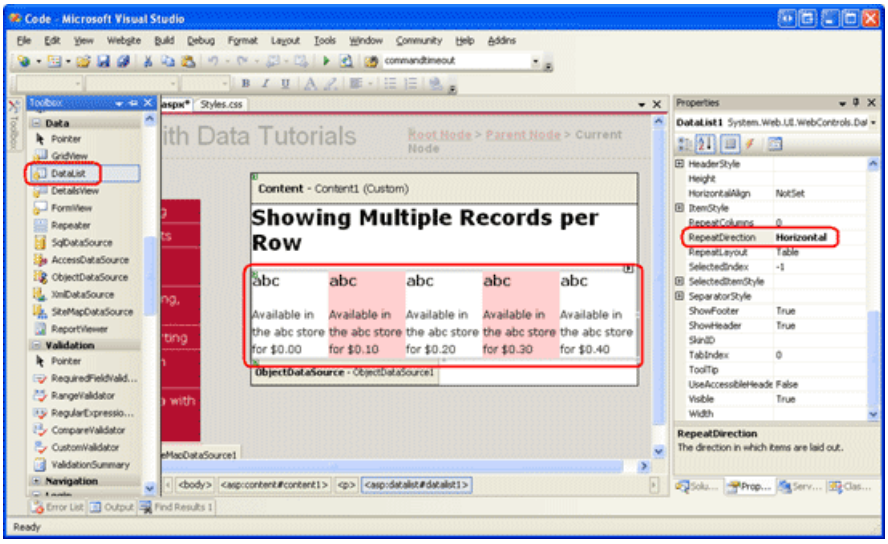


Figure 3: The `RepeatDirection` Property Dictates How the Direction the DataList s Items are Laid Out (Click to view full-size image)

When displaying small amounts of data, a single-row, multi-column table might be an ideal way to maximize screen real estate. For larger volumes of data, however, a single row will require numerous columns, which pushes those items that can t fit on the screen off to the right. Figure 4 shows the products when rendered in a single-row DataList. Since there are many products (over 80), the user will have to scroll far to the right to view information about each of the products.

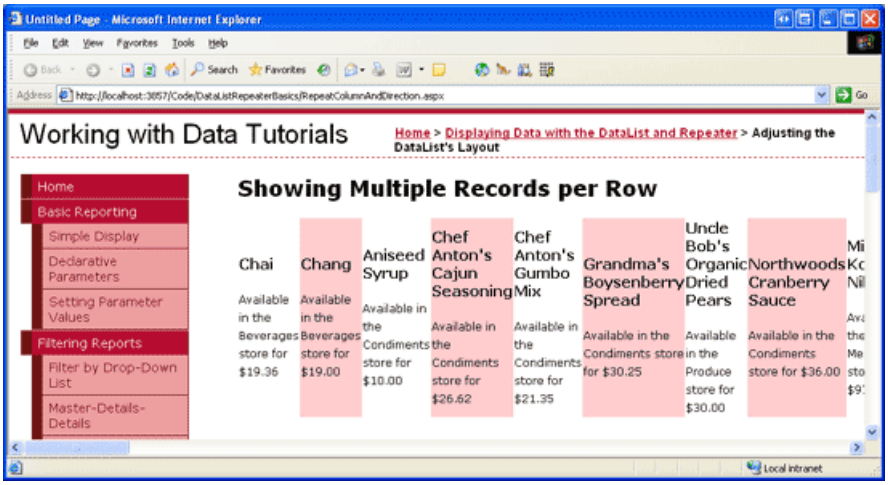


Figure 4: For Sufficiently Large Data Sources, a Single Column DataList Will Require Horizontal Scrolling (Click to view full-size image)

Step 3: Displaying Data in a Multi-Column, Multi-Row Table

To create a multi-column, multi-row DataList, we need to set the `RepeatColumns` property to the number of columns to display. By default, the `RepeatColumns` property is set to 0, which will cause the DataList to display all of its items in a single row or a column (depending on the value of the `RepeatDirection` property).

For our example, let s display three products per table row. Therefore, set the `RepeatColumns` property to 3. After making this change, take a moment to view the results in a browser. As Figure 5 shows, the products are now listed in a three-column, multi-row table.

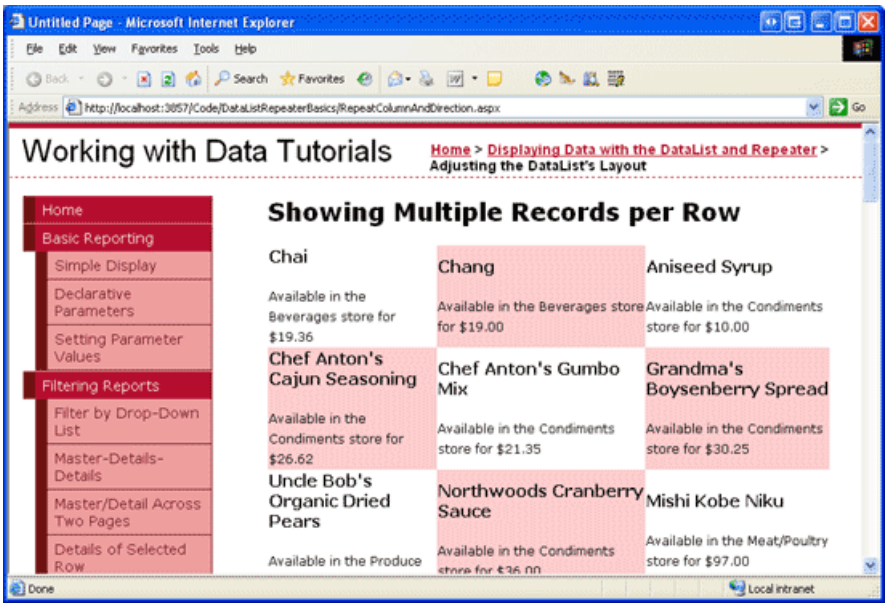


Figure 5: Three Products are Displayed per Row (Click to view full-size image)

The `RepeatDirection` property affects how the items in the DataList are laid out. Figure 5 shows the results with the `RepeatDirection` property set to `Horizontal`. Note that the first three products Chai, Chang, and Aniseed Syrup are laid out from left to right, top to bottom. The next three products (starting with Chef Anton s Cajun Seasoning) appear in a row beneath the first three. Changing the `RepeatDirection` property back to `Vertical`,

however, lays out these products from top to bottom, left to right, as Figure 6 illustrates.

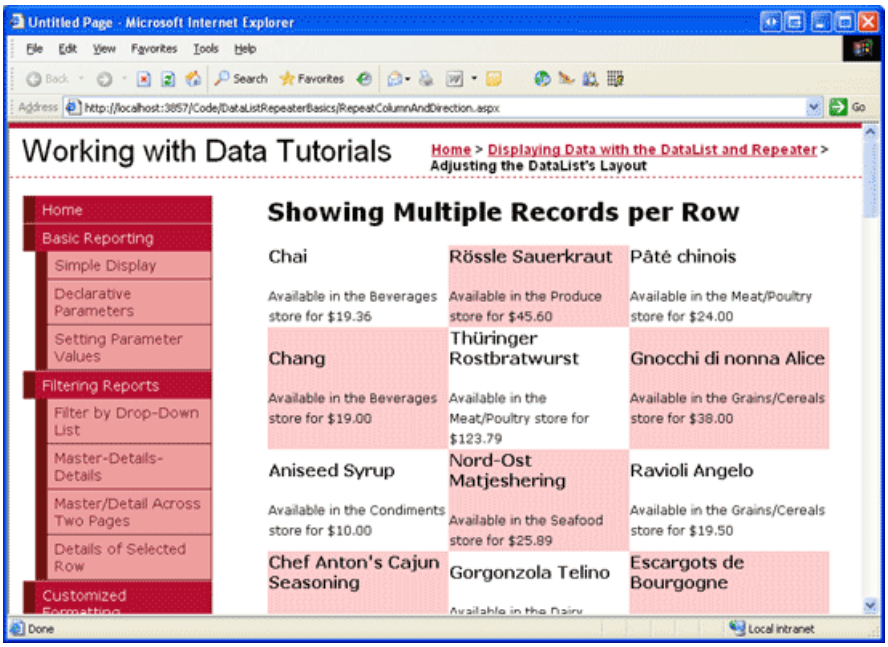


Figure 6: Here, the Products are Laid Out Vertically (Click to view full-size image)

The number of rows displayed in the resulting table depends on the number of total records bound to the DataList. Precisely, it s the ceiling of the total number of data source items divided by the `RepeatColumns` property value. Since the `Products` table currently has 84 products, which is divisible by 3, there are 28 rows. If the number of items in the data source and the `RepeatColumns` property value are not divisible, then the last row or column will have blank cells. If the `RepeatDirection` is set to `Vertical`, then the last column will have empty cells; if `RepeatDirection` is `Horizontal`, then the last row will have the empty cells.

Summary

The DataList, by default, lists its items in a single-column, multi-row table, which mimics the layout of a GridView with a single TemplateField. While this default layout is acceptable, we can maximize screen real estate by displaying multiple data source items per row. Accomplishing this is simply a matter of setting the DataList s `RepeatColumns` property to the number of columns to display per row. Additionally, the DataList s `RepeatDirection` property can be used to indicate whether the contents of the multi-column, multi-row table should be laid out horizontally from left to right, top to bottom or vertically from top to bottom, left to right.

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was John Suru. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Nested Data Web Controls (VB)

2/20/2020 • 14 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we will explore how to use a Repeater nested inside another Repeater. The examples will illustrate how to populate the inner Repeater both declaratively and programmatically.

Introduction

In addition to static HTML and databinding syntax, templates can also include Web controls and User Controls. These Web controls can have their properties assigned via declarative, databinding syntax, or can be accessed programmatically in the appropriate server-side event handlers.

By embedding controls within a template, the appearance and user experience can be customized and improved upon. For example, in the Using TemplateFields in the GridView Control tutorial, we saw how to customize the GridView s display by adding a Calendar control in a TemplateField to show an employee s hire date; in the Adding Validation Controls to the Editing and Inserting Interfaces and Customizing the Data Modification Interface tutorials, we saw how to customize the editing and inserting interfaces by adding validation controls, TextBoxes, DropDownLists, and other Web controls.

Templates can also contain other data Web controls. That is, we can have a DataList that contains another DataList (or Repeater or GridView or DetailsView, and so on) within its templates. The challenge with such an interface is binding the appropriate data to the inner data Web control. There are a few different approaches available, ranging from declarative options using the ObjectDataSource to programmatic ones.

In this tutorial we will explore how to use a Repeater nested inside another Repeater. The outer Repeater will contain an item for each category in the database, displaying the category s name and description. Each category item s inner Repeater will display information for each product belonging to that category (see Figure 1) in a bulleted list. Our examples will illustrate how to populate the inner Repeater both declaratively and programmatically.

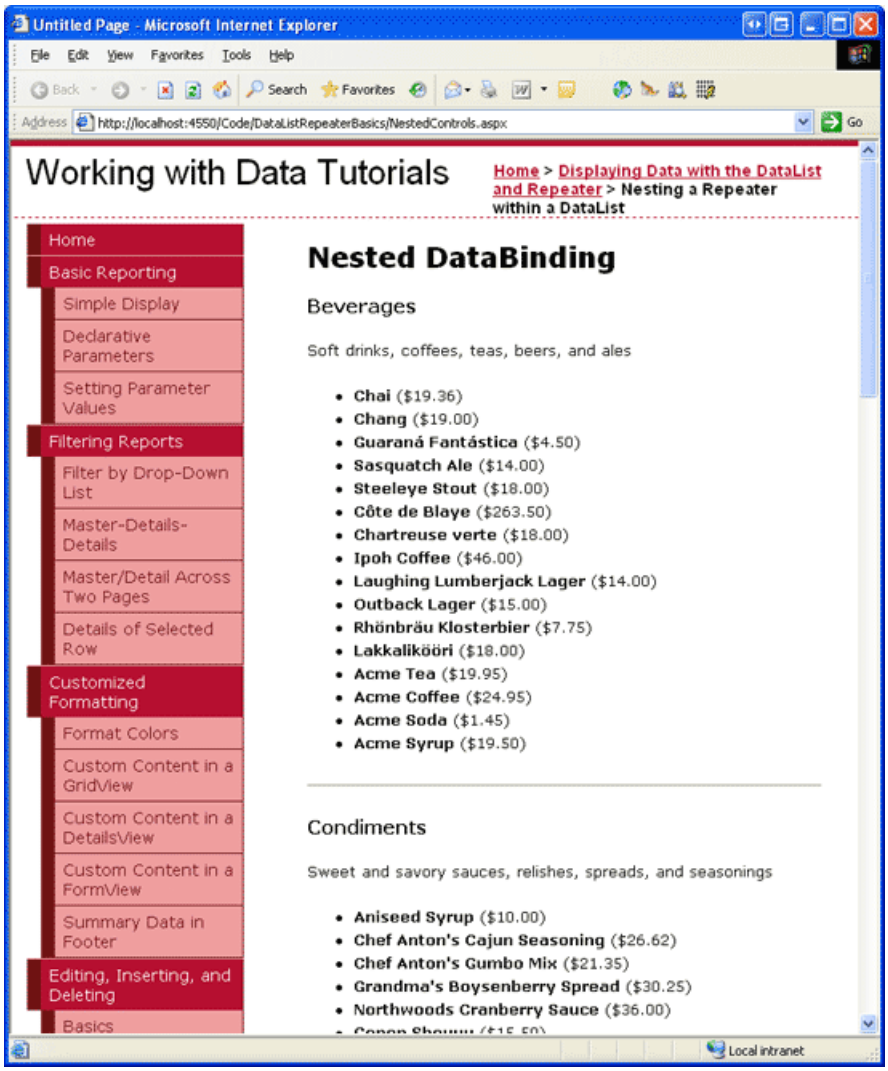


Figure 1: Each Category, Along with its Products, are Listed (Click to view full-size image)

Step 1: Creating the Category Listing

When building a page that uses nested data Web controls, I find it helpful to design, create, and test the outermost data Web control first, without even worrying about the inner nested control. Therefore, let s start by walking through the steps necessary to add a Repeater to the page that lists the name and description for each category.

Start by opening the `NestedControls.aspx` page in the `DataListRepeaterBasics` folder and add a Repeater control to the page, setting its `ID` property to `CategoryList`. From the Repeater s smart tag, choose to create a new ObjectDataSource named `CategoriesDataSource`.

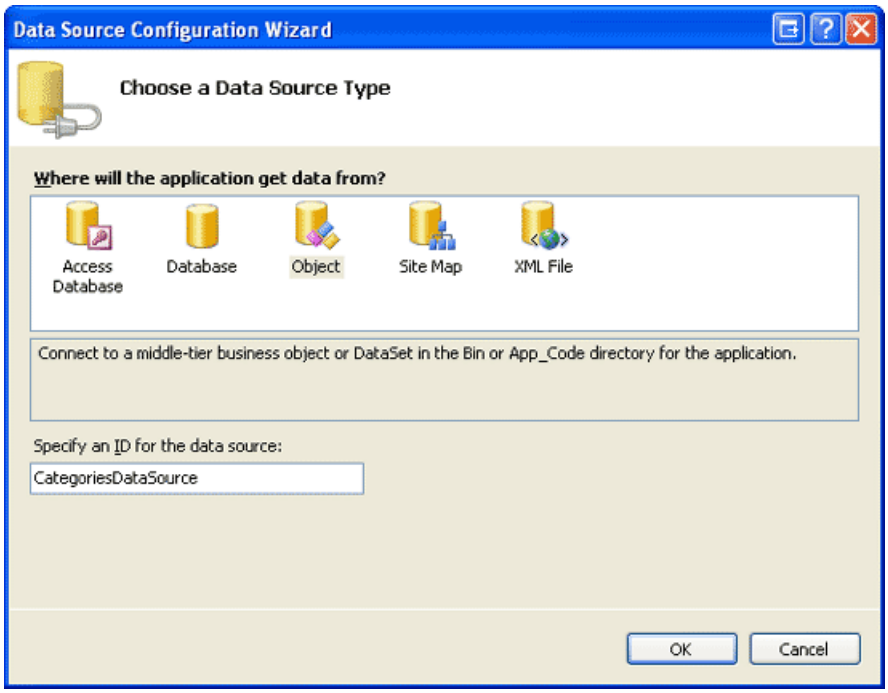


Figure 2: Name the New ObjectDataSource `CategoriesDataSource` (Click to view full-size image)

Configure the ObjectDataSource so that it pulls its data from the `CategoriesBLL` class s `GetCategories` method.

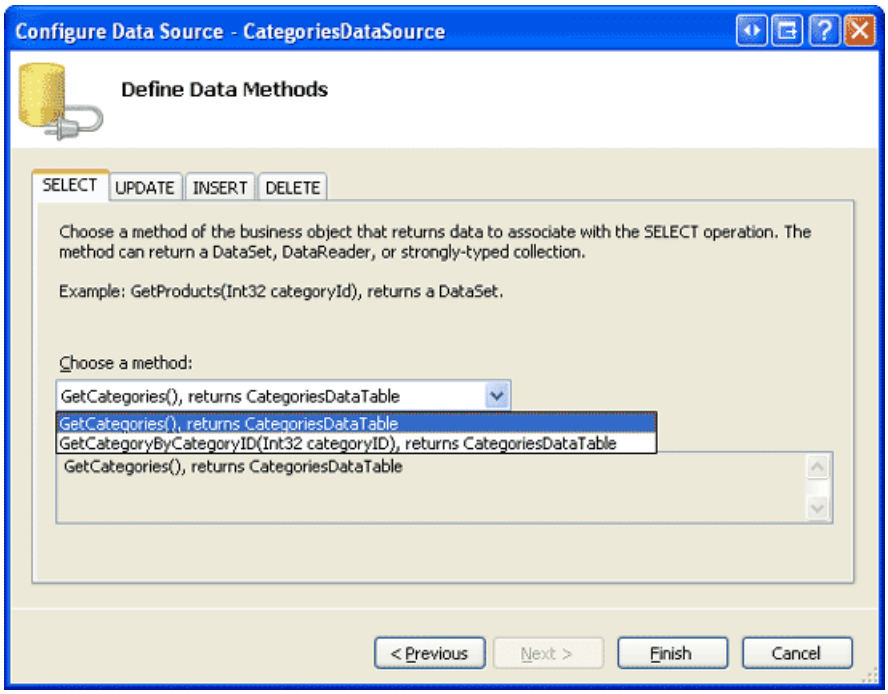


Figure 3: Configure the ObjectDataSource to Use the `CategoriesBLL` Class s `GetCategories` Method (Click to view full-size image)

To specify the Repeater s template content we need to go to the Source view and manually enter the declarative syntax. Add an `ItemTemplate` that displays the category s name in an `<h4>` element and the category s description in a paragraph element (`<p>`). Furthermore, let s separate each category with a horizontal rule (`<hr>`). After making these changes your page should contain declarative syntax for the Repeater and ObjectDataSource that is similar to the following:

```
<asp:Repeater ID="CategoryList" DataSourceID="CategoriesDataSource"
    EnableViewState="False" runat="server">
    <ItemTemplate>
        <h4><%# Eval("CategoryName") %></h4>
        <p><%# Eval("Description") %></p>
    </ItemTemplate>
    <SeparatorTemplate>
        <hr />
    </SeparatorTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

Figure 4 shows our progress when viewed through a browser.

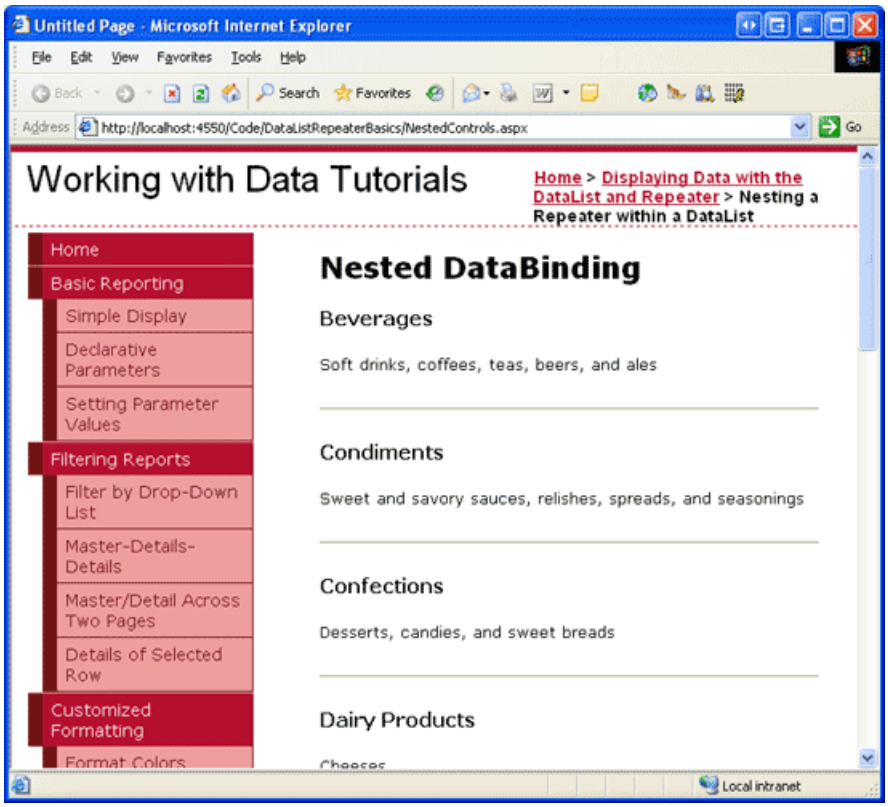


Figure 4: Each Category s Name and Description is Listed, Separated by a Horizontal Rule (Click to view full-size image)

Step 2: Adding the Nested Product Repeater

With the category listing complete, our next task is to add a Repeater to the `CategoryList` s `ItemTemplate` that displays information about those products belonging to the appropriate category. There are a number of ways we can retrieve the data for this inner Repeater, two of which we'll explore shortly. For now, let s just create the products Repeater within the `CategoryList` Repeater s `ItemTemplate`. Specifically, let s have the product Repeater display each product in a bulleted list with each list item including the product s name and price.

To create this Repeater we need to manually enter the inner Repeater s declarative syntax and templates into the `CategoryList` s `ItemTemplate`. Add the following markup within the `CategoryList` Repeater s `ItemTemplate`:

```
<asp:Repeater ID="ProductsByCategoryList" EnableViewState="False"
    runat="server">
    <HeaderTemplate>
        <ul>
    </HeaderTemplate>
    <ItemTemplate>
        <li><strong><%# Eval("ProductName") %></strong>
            (<%# Eval("UnitPrice", "{0:C}") %>)</li>
    </ItemTemplate>
    <FooterTemplate>
        </ul>
    </FooterTemplate>
</asp:Repeater>
```

Step 3: Binding the Category-Specific Products to the ProductsByCategoryList Repeater

If you visit the page through a browser at this point, your screen will look the same as in Figure 4 because we ve yet to bind any data to the Repeater. There are a few ways that we can grab the appropriate product records and bind them to the Repeater, some more efficient than others. The main challenge here is getting back the appropriate products for the specified category.

The data to bind to the inner Repeater control can either be accessed declaratively, through an ObjectDataSource in the `CategoryList` Repeater s `ItemTemplate`, or programmatically, from the ASP.NET page s code-behind page. Similarly, this data can be bound to the inner Repeater either declaratively - through the inner Repeater s `DataSourceID` property or through declarative databinding syntax or programmatically by referencing the inner Repeater in the `CategoryList` Repeater s `ItemDataBound` event handler, programmatically setting its `DataSource` property, and calling its `DataBind()` method. Let s explore each of these approaches.

Accessing the Data Declaratively with an ObjectDataSource Control and the `ItemDataBound` Event Handler

Since we ve used the ObjectDataSource extensively throughout this tutorial series, the most natural choice for accessing data for this example is to stick with the ObjectDataSource. The `ProductsBLL` class has a `GetProductsByCategoryID(categoryID)` method that returns information about those products that belong to the specified *`categoryID`*. Therefore, we can add an ObjectDataSource to the `CategoryList` Repeater s `ItemTemplate` and configure it to access its data from this class s method.

Unfortunately, the Repeater doesn t allow its templates to be edited through the Design view so we need to add the declarative syntax for this ObjectDataSource control by hand. The following syntax shows the `CategoryList` Repeater s `ItemTemplate` after adding this new ObjectDataSource (`ProductsByCategoryDataSource`):

```
<h4><%# Eval("CategoryName") %></h4>
<p><%# Eval("Description") %></p>
<asp:Repeater ID="ProductsByCategoryList" EnableViewState="False"
        DataSourceID="ProductsByCategoryDataSource" runat="server">
    <HeaderTemplate>
        <ul>
    </HeaderTemplate>
    <ItemTemplate>
        <li><strong><%# Eval("ProductName") %></strong> -
                sold as <%# Eval("QuantityPerUnit") %> at
                <%# Eval("UnitPrice", "{0:C}") %></li>
    </ItemTemplate>
    <FooterTemplate>
        </ul>
    </FooterTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="ProductsByCategoryDataSource" runat="server"
           SelectMethod="GetProductsByCategoryID" TypeName="ProductsBLL">
   <SelectParameters>
        <asp:Parameter Name="CategoryID" Type="Int32" />
   </SelectParameters>
</asp:ObjectDataSource>
```

When using the ObjectDataSource approach we need to set the `ProductsByCategoryList` Repeater s `DataSourceID` property to the `ID` of the ObjectDataSource (`ProductsByCategoryDataSource`). Also, notice that our ObjectDataSource has an `<asp:Parameter>` element that specifies the *`categoryID`* value that will be passed into the `GetProductsByCategoryID(categoryID)` method. But how do we specify this value? Ideally, we d be able to just set the `DefaultValue` property of the `<asp:Parameter>` element using databinding syntax, like so:

```
<asp:Parameter Name="CategoryID" Type="Int32"
    DefaultValue='<%# Eval("CategoryID")' />
```

Unfortunately, databinding syntax is only valid in controls that have a `DataBinding` event. The `Parameter` class lacks such an event and therefore the above syntax is illegal and will result in a runtime error.

To set this value, we need to create an event handler for the `CategoryList` Repeater s `ItemDataBound` event. Recall that the `ItemDataBound` event fires once for each item bound to the Repeater. Therefore, each time this event fires for the outer Repeater we can assign the current `CategoryID` value to the `ProductsByCategoryDataSource` ObjectDataSource s `CategoryID` parameter.

Create an event handler for the `CategoryList` Repeater s `ItemDataBound` event with the following code:

```
Protected Sub CategoryList_ItemDataBound(sender As Object, e As RepeaterItemEventArgs) _
    Handles CategoryList.ItemDataBound
    If e.Item.ItemType = ListItemType.AlternatingItem _
        OrElse e.Item.ItemType = ListItemType.Item Then
        ' Reference the CategoriesRow object being bound to this RepeaterItem
        Dim category As Northwind.CategoriesRow = _
            CType(CType(e.Item.DataItem, System.Data.DataRowView).Row, _
                Northwind.CategoriesRow)
        ' Reference the ProductsByCategoryDataSource ObjectDataSource
        Dim ProductsByCategoryDataSource As ObjectDataSource = _
            CType(e.Item.FindControl("ProductsByCategoryDataSource"), _
                ObjectDataSource)
        ' Set the CategoryID Parameter value
        ProductsByCategoryDataSource.SelectParameters("CategoryID").DefaultValue = _
            category.CategoryID.ToString()
    End If
End Sub
```

This event handler starts by ensuring that we re dealing with a data item rather than the header, footer, or separator item. Next, we reference the actual `CategoriesRow` instance that has just been bound to the current `RepeaterItem`. Finally, we reference the ObjectDataSource in the `ItemTemplate` and assign its `CategoryID` parameter value to the `CategoryID` of the current `RepeaterItem`.

With this event handler, the `ProductsByCategoryList` Repeater in each `RepeaterItem` is bound to those products in the `RepeaterItem` s category. Figure 5 shows a screen shot of the resulting output.

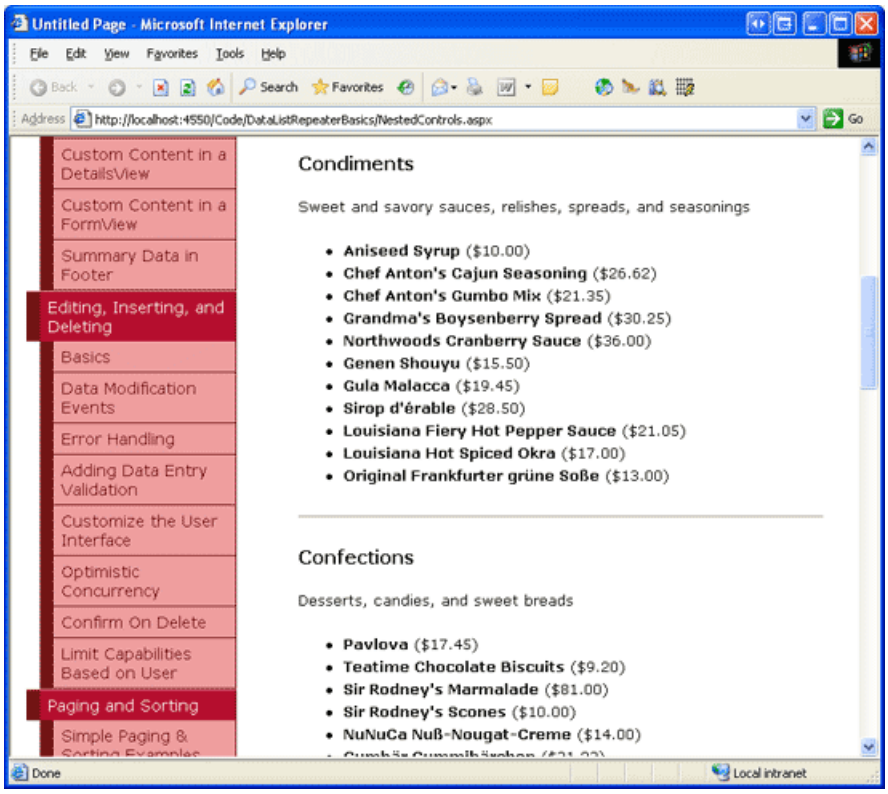


Figure 5: The Outer Repeater Lists Each Category; the Inner One Lists the Products for that Category (Click to view full-size image)

Accessing the Products by Category Data Programmatically

Instead of using an ObjectDataSource to retrieve the products for the current category, we could create a method in our ASP.NET page s code-behind class (or in the `App_Code` folder or in a separate Class Library project) that returns the appropriate set of products when passed in a `CategoryID`. Imagine that we had such a method in our ASP.NET page s code-behind class and that it was named `GetProductsInCategory(categoryID)`. With this method in place we could bind the products for the current category to the inner Repeater using the following declarative syntax:

```
<asp:Repeater runat="server" ID="ProductsByCategoryList" EnableViewState="False"
      DataSource='<%# GetProductsInCategory(CType(Eval("CategoryID"), Integer)) %>'>
  ...
</asp:Repeater>
```

The Repeater s `DataSource` property uses the databinding syntax to indicate that its data comes from the `GetProductsInCategory(categoryID)` method. Since `Eval("CategoryID")` returns a value of type `Object`, we cast the object to an `Integer` before passing it into the `GetProductsInCategory(categoryID)` method. Note that the `CategoryID` accessed here via the databinding syntax is the `CategoryID` in the *outer* Repeater (`CategoryList`), the one that s bound to the records in the `Categories` table. Therefore, we know that `CategoryID` cannot be a database `NULL` value, which is why we can blindly cast the `Eval` method without checking if we re dealing with a `DBNull`.

With this approach, we need to create the `GetProductsInCategory(categoryID)` method and have it retrieve the appropriate set of products given the supplied *`categoryID`*. We can do this by simply returning the `ProductsDataTable` returned by the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method. Let s create the `GetProductsInCategory(categoryID)` method in the code-behind class for our `NestedControls.aspx` page. Do so using the following code:

```
Protected Function GetProductsInCategory(ByVal categoryID As Integer) _
    As Northwind.ProductsDataTable
    ' Create an instance of the ProductsBLL class
    Dim productAPI As ProductsBLL = New ProductsBLL()
    ' Return the products in the category
    Return productAPI.GetProductsByCategoryID(categoryID)
End Function
```

This method simply creates an instance of the `ProductsBLL` method and returns the results of the `GetProductsByCategoryID(categoryID)` method. Note that the method must be marked `Public` or `Protected`; if the method is marked `Private`, it will not be accessible from the ASP.NET page s declarative markup.

After making these changes to use this new technique, take a moment to view the page through a browser. The output should be identical to the output when using the ObjectDataSource and `ItemDataBound` event handler approach (refer back to Figure 5 to see a screen shot).

NOTE

It may seem like busywork to create the `GetProductsInCategory(categoryID)` method in the ASP.NET page s code-behind class. After all, this method simply creates an instance of the `ProductsBLL` class and returns the results of its `GetProductsByCategoryID(categoryID)` method. Why not just call this method directly from the databinding syntax in the inner Repeater, like: `DataSource='<%# ProductsBLL.GetProductsByCategoryID(CType(Eval("CategoryID"), Integer)) %>'`? Although this syntax won't work with our current implementation of the `ProductsBLL` class (since the `GetProductsByCategoryID(categoryID)` method is an instance method), you could modify `ProductsBLL` to include a static `GetProductsByCategoryID(categoryID)` method or have the class include a static `Instance()` method to return a new instance of the `ProductsBLL` class.

While such modifications would eliminate the need for the `GetProductsInCategory(categoryID)` method in the ASP.NET page s code-behind class, the code-behind class method gives us more flexibility in working with the data retrieved, as we'll see shortly.

Retrieving All of the Product Information at Once

The two pervious techniques we ve examined grab those products for the current category by making a call to the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method (the first approach did so through an ObjectDataSource, the second through the `GetProductsInCategory(categoryID)` method in the code-behind class). Each time this method is invoked, the Business Logic Layer calls down to the Data Access Layer, which queries the database with a SQL statement that returns rows from the `Products` table whose `CategoryID` field matches the supplied input parameter.

Given *N* categories in the system, this approach nets *N* + 1 calls to the database one database query to get all of the categories and then *N* calls to get the products specific to each category. We can, however, retrieve all the needed data in just two database calls one call to get all of the categories and another to get all of the products. Once we have all of the products, we can filter those products so that only the products matching the current `CategoryID` are bound to that category s inner Repeater.

To provide this functionality, we only need to make a slight modification to the `GetProductsInCategory(categoryID)`

method in our ASP.NET page s code-behind class. Rather than blindly returning the results of the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method, we can instead first access *all* of the products (if they haven't been accessed already) and then return just the filtered view of the products based on the passed-in `CategoryID`.

```
Private allProducts As Northwind.ProductsDataTable = Nothing
Protected Function GetProductsInCategory(ByVal categoryID As Integer) _
    As Northwind.ProductsDataTable
    ' First, see if we've yet to have accessed all of the product information
    If allProducts Is Nothing Then
        Dim productAPI As ProductsBLL = New ProductsBLL()
        allProducts = productAPI.GetProducts()
    End If
    ' Return the filtered view
    allProducts.DefaultView.RowFilter = "CategoryID = " & categoryID
    Return allProducts
End Function
```

Note the addition of the page-level variable, `allProducts`. This holds information about all of the products and is populated the first time the `GetProductsInCategory(categoryID)` method is invoked. After ensuring that the `allProducts` object has been created and populated, the method filters the DataTable s results such that only those rows whose `CategoryID` matches the specified `CategoryID` are accessible. This approach reduces the number of times the database is accessed from *N* + 1 down to two.

This enhancement does not introduce any change to the rendered markup of the page, nor does it bring back fewer records than the other approach. It simply reduces the number of calls to the database.

> **NOTE**
> One might intuitively reason that reducing the number of database accesses would assuredly improve performance. However, this might not be the case. If you have a large number of products whose `CategoryID` is `NULL`, for example, then the call to the `GetProducts` method returns a number of products that are never displayed. Moreover, returning all of the products can be wasteful if you re only showing a subset of the categories, which might be the case if you have implemented paging.

As always, when it comes to analyzing the performance of two techniques, the only surefire measure is to run controlled tests tailored for your application s common case scenarios.

Summary

In this tutorial we saw how to nest one data Web control within another, specifically examining how to have an outer Repeater display an item for each category with an inner Repeater listing the products for each category in a bulleted list. The main challenge in building a nested user interface lies in accessing and binding the correct data to the inner data Web control. There are a variety of techniques available, two of which we examined in this tutorial. The first approach examined used an ObjectDataSource in the outer data Web control s `ItemTemplate` that was bound to the inner data Web control through its `DataSourceID` property. The second technique accessed the data via a method in the ASP.NET page s code-behind class. This method can then be bound to the inner data Web control s `DataSource` property through databinding syntax.

While the nested user interface examined in this tutorial used a Repeater nested within a Repeater, these techniques can be extended to the other data Web controls. You can nest a Repeater within a GridView, or a GridView within a DataList, and so on.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Zack Jones and Liz Shulok. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Filtering Scenarios with the DataList and Repeater

2/20/2020 • 2 minutes to read • Edit Online

In these tutorials you see how to display master/detail reports using drop-down lists for the master records and a DataList control for the details.

- Master/Detail Filtering With a DropDownList (C#)
- Master/Detail Filtering Across Two Pages (C#)
- Master/Detail Using a Bulleted List of Master Records with a Details DataList (C#)
- Master/Detail Filtering With a DropDownList (VB)
- Master/Detail Filtering Across Two Pages (VB)
- Master/Detail Using a Bulleted List of Master Records with a Details DataList (VB)

Master/Detail Filtering With a DropDownList (C#)

2/20/2020 • 9 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we see how to display master/detail reports in a single web page using DropDownLists to display the "master" records and a DataList to display the "details".

Introduction

The master/detail report, which we first created using a GridView in the earlier Master/Detail Filtering With a DropDownList tutorial, begins by showing some set of "master" records. The user can then drill down into one of the master records, thereby viewing that master record's "details." Master/detail reports are an ideal choice for visualizing one-to-many relationships and for displaying detailed information from particularly "wide" tables (ones that have a lot of columns). We've explored how to implement master/detail reports using the GridView and DetailsView controls in previous tutorials. In this tutorial and the next two, we'll reexamine these concepts, but focus on using DataList and Repeater controls instead.

In this tutorial, we'll look at using a DropDownList to contain the "master" records, with the "details" records displayed in a DataList.

Step 1: Adding the Master/Detail Tutorial Web Pages

Before we start this tutorial, let's first take a moment to add the folder and ASP.NET pages we'll need for this tutorial and the next two dealing with master/detail reports using the DataList and Repeater controls. Start by creating a new folder in the project named `DataListRepeaterFiltering`. Next, add the following five ASP.NET pages to this folder, having all of them configured to use the master page `Site.master`:

- `Default.aspx`
- `FilterByDropDownList.aspx`
- `CategoryListMaster.aspx`
- `ProductsForCategoryDetails.aspx`
- `CategoriesAndProducts.aspx`

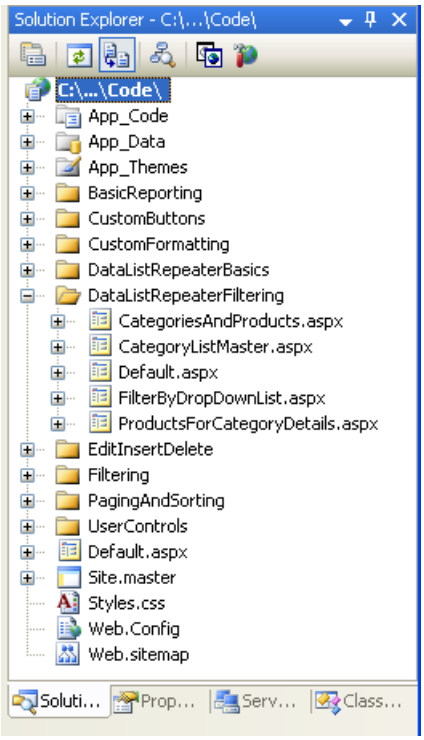


Figure 1: Create a `DataListRepeaterFiltering` Folder and Add the Tutorial ASP.NET Pages

Next, open the `Default.aspx` page and drag the `SectionLevelTutorialListing.ascx` User Control from the `UserControls` folder onto the Design surface. This User Control, which we created in the Master Pages and Site Navigation tutorial, enumerates the site map and displays the tutorials from the current section in a bulleted list.

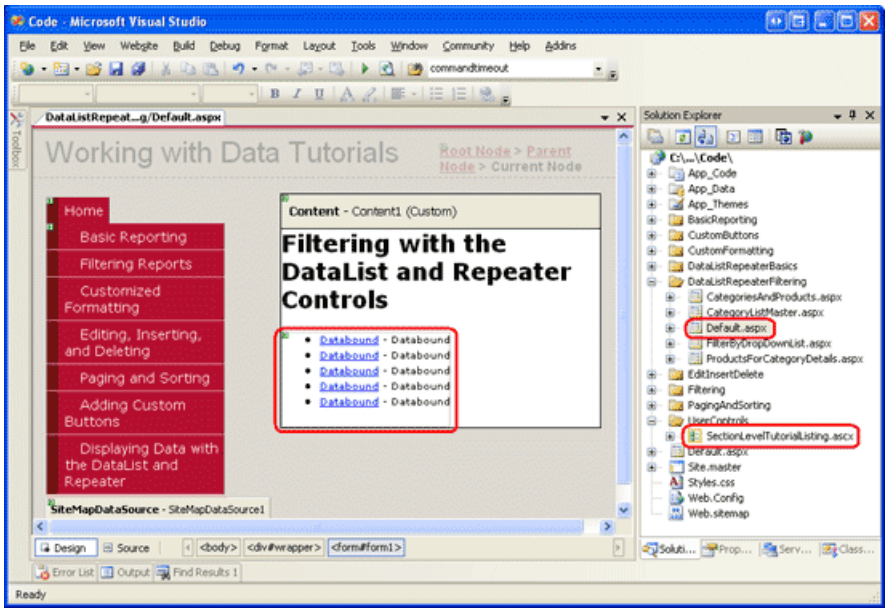


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

In order to have the bulleted list display the master/detail tutorials we'll be creating, we need to add them to the site map. Open the `Web.sitemap` file and add the following markup after the "Displaying Data with the DataList and Repeater" site map node markup:

```
<siteMapNode
    title="Master/Detail Reports with the DataList and Repeater"
    description="Samples of Reports that Use the DataList and Repeater Controls"
    url="~/DataListRepeaterFiltering/Default.aspx">

    <siteMapNode
        title="Filter by Drop-Down List"
        description="Filter results using a drop-down list."
        url="~/DataListRepeaterFiltering/FilterByDropDownList.aspx" />

    <siteMapNode
        title="Master/Detail Across Two Pages"
        description="Master records on one page, detail records on another."
        url="~/DataListRepeaterFiltering/CategoryListMaster.aspx" />

    <siteMapNode
        title="Maser/Detail on One Page"
        description="Master records in the left column, details on the right,
                     both on the same page."
        url="~/DataListRepeaterFiltering/CategoriesAndProducts.aspx" />

</siteMapNode>
```

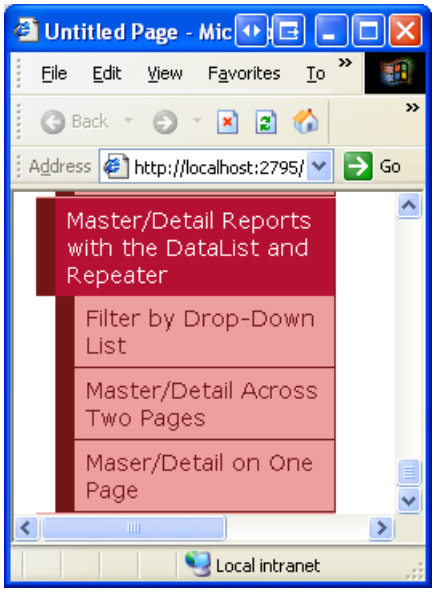


Figure 3: Update the Site Map to Include the New ASP.NET Pages

Step 2: Displaying the Categories in a DropDownList

Our master/detail report will list the categories in a DropDownList, with the selected list item's products displayed further down in the page in a DataList. The first task ahead of us, then, is to have the categories displayed in a DropDownList. Start by opening the `FilterByDropDownList.aspx` page in the `DataListRepeaterFiltering` folder and drag a DropDownList from the Toolbox onto the page's designer. Next, set the DropDownList's `ID` property to `Categories`. Click on the Choose Data Source link from the DropDownList's smart tag and create a new ObjectDataSource named `CategoriesDataSource`.

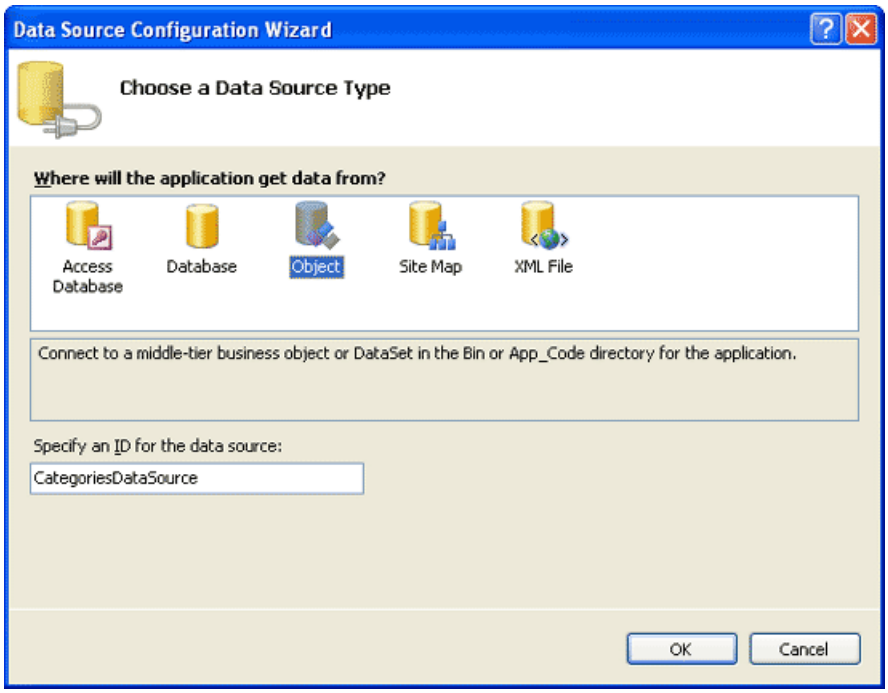


Figure 4: Add a New ObjectDataSource Named `CategoriesDataSource` (Click to view full-size image)

Configure the new ObjectDataSource such that it invokes the `CategoriesBLL` class's `GetCategories()` method. After configuring the ObjectDataSource we still need to specify what data source field should be displayed in the DropDownList and which one should be associated as the value for each list item. Have the `CategoryName` field as the display and `CategoryID` as the value for each list item.

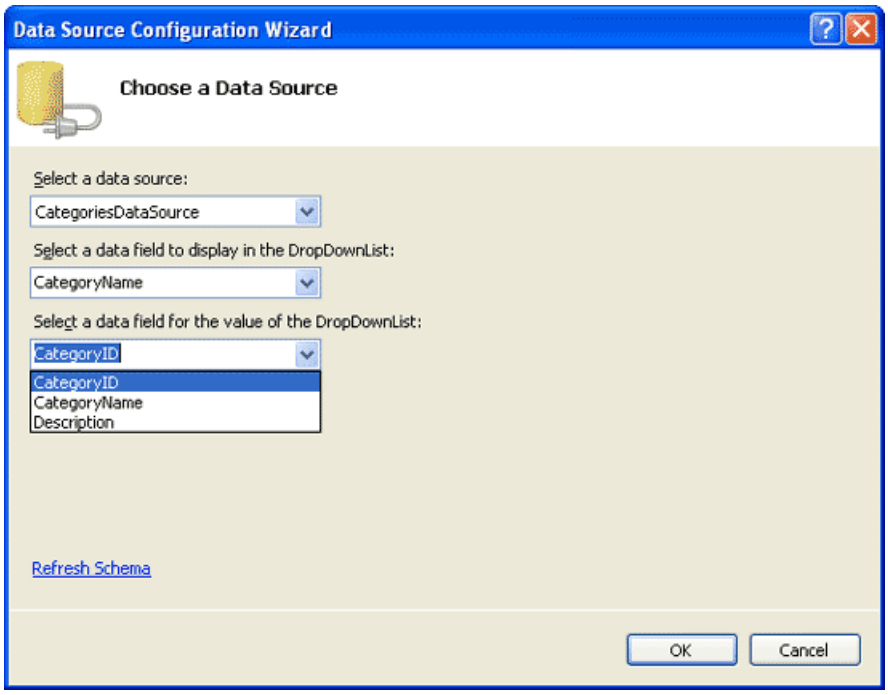


Figure 5: Have the DropDownList Display the `CategoryName` Field and Use `CategoryID` as the Value (Click to view full-size image)

At this point we have a DropDownList control that's populated with the records from the `Categories` table (all accomplished in about six seconds). Figure 6 shows our progress thus far when viewed through a browser.

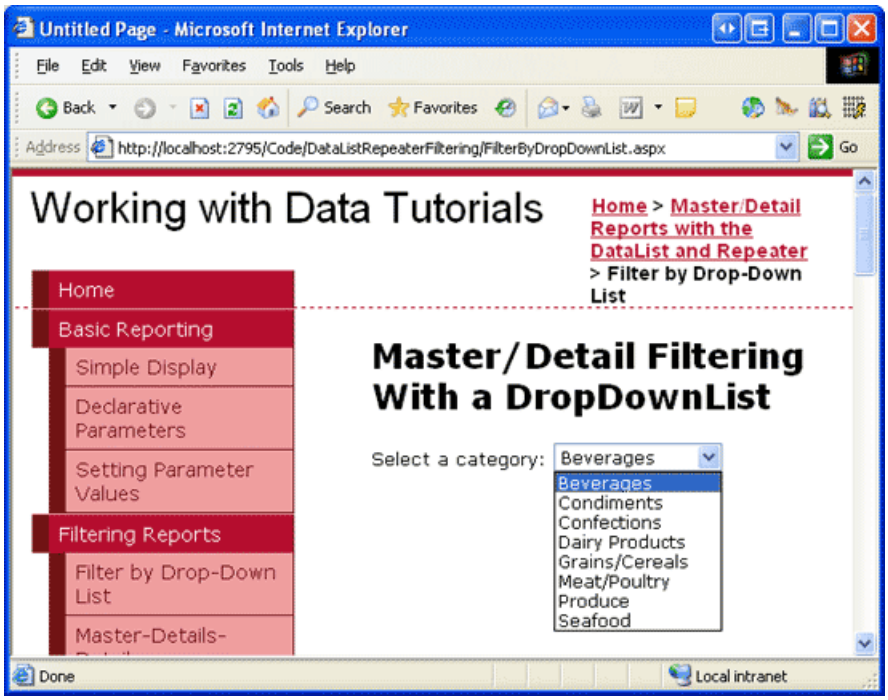


Figure 6: A Drop-Down Lists the Current Categories (Click to view full-size image)

Step 2: Adding the Products DataList

The last step in our master/detail report is to list the products associated with the selected category. To accomplish this, add a DataList to the page and create a new ObjectDataSource named `ProductsByCategoryDataSource`. Have the `ProductsByCategoryDataSource` control retrieve its data from the `ProductsBLL` class's `GetProductsByCategoryID(categoryID)` method. Since this master/detail report is read-only, choose the (None) option in the INSERT, UPDATE, and DELETE tabs.

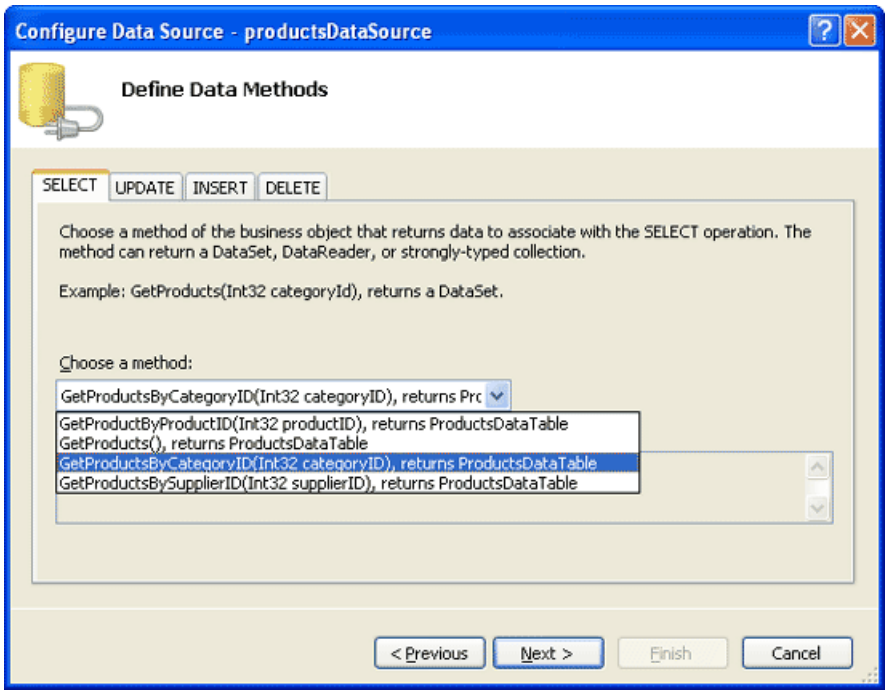


Figure 7: Select the `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

After clicking Next, the ObjectDataSource wizard prompts us for the source of the value for the `GetProductsByCategoryID(categoryID)` method's *`categoryID`* parameter. To use the value of the selected `categories` DropDownList item set the Parameter source to Control and the ControlID to `Categories`.

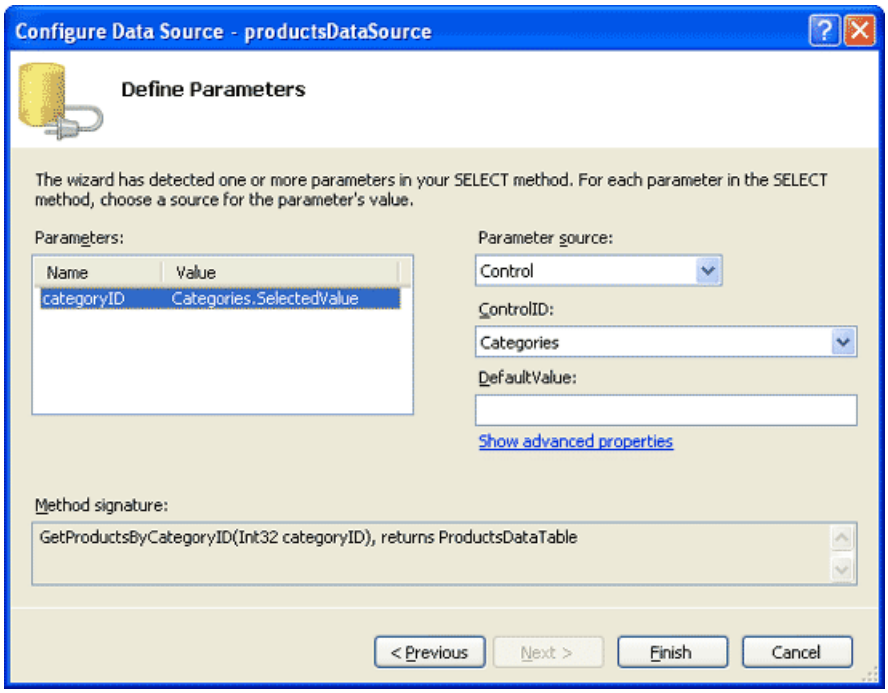


Figure 8: Set the `categoryID` Parameter to the Value of the `Categories` DropDownList (Click to view full-size image)

Upon completing the Configure Data Source wizard, Visual Studio will automatically generate an `ItemTemplate` for the DataList that displays the name and value of each data field. Let's enhance the DataList to instead use an `ItemTemplate` that displays just the product's name, category, supplier, quantity per unit, and price along with a `SeparatorTemplate` that injects an `<hr>` element between each item. I'm going to use the `ItemTemplate` from an example in the Displaying Data with the DataList and Repeater Controls tutorial, but feel free to use whatever template markup you find most visually appealing.

After making these changes, your DataList and its ObjectDataSource's markup should look similar to the following:

```
<asp:DataList ID="DataList1" runat="server" DataKeyField="ProductID"
    DataSourceID="ProductsByCategoryDataSource" EnableViewState="False">
    <ItemTemplate>
        <h4>
            <asp:Label ID="ProductNameLabel" runat="server"
                Text='<%# Eval("ProductName") %>' />
        </h4>
        <table border="0">
            <tr>
                <td class="ProductPropertyLabel">Category:</td>
                <td><asp:Label ID="CategoryNameLabel" runat="server"
                    Text='<%# Eval("CategoryName") %>' /></td>
                <td class="ProductPropertyLabel">Supplier:</td>
                <td><asp:Label ID="SupplierNameLabel" runat="server"
                    Text='<%# Eval("SupplierName") %>' /></td>
            </tr>
            <tr>
                <td class="ProductPropertyLabel">Qty/Unit:</td>
                <td><asp:Label ID="QuantityPerUnitLabel" runat="server"
                    Text='<%# Eval("QuantityPerUnit") %>' /></td>
                <td class="ProductPropertyLabel">Price:</td>
                <td><asp:Label ID="UnitPriceLabel" runat="server"
                    Text='<%# Eval("UnitPrice", "{0:C}") %>' /></td>
            </tr>
        </table>
    </ItemTemplate>
    <SeparatorTemplate>
        <hr />
    </SeparatorTemplate>
</asp:DataList>

<asp:ObjectDataSource ID="ProductsByCategoryDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProductsByCategoryID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:ControlParameter ControlID="Categories" Name="categoryID"
            PropertyName="SelectedValue" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

Take a moment to check out our progress in a browser. When first visiting the page, those products belonging to the selected category (Beverages) are displayed (as shown in Figure 9), but changing the DropDownList doesn't update the data. This is because a postback must occur for the DataList to update. To accomplish this we can either set the DropDownList's `AutoPostBack` property to `true` or add a Button Web control to the page. For this tutorial, I've opted to set the DropDownList's `AutoPostBack` property to `true`.

Figures 9 and 10 illustrate the master/detail report in action.

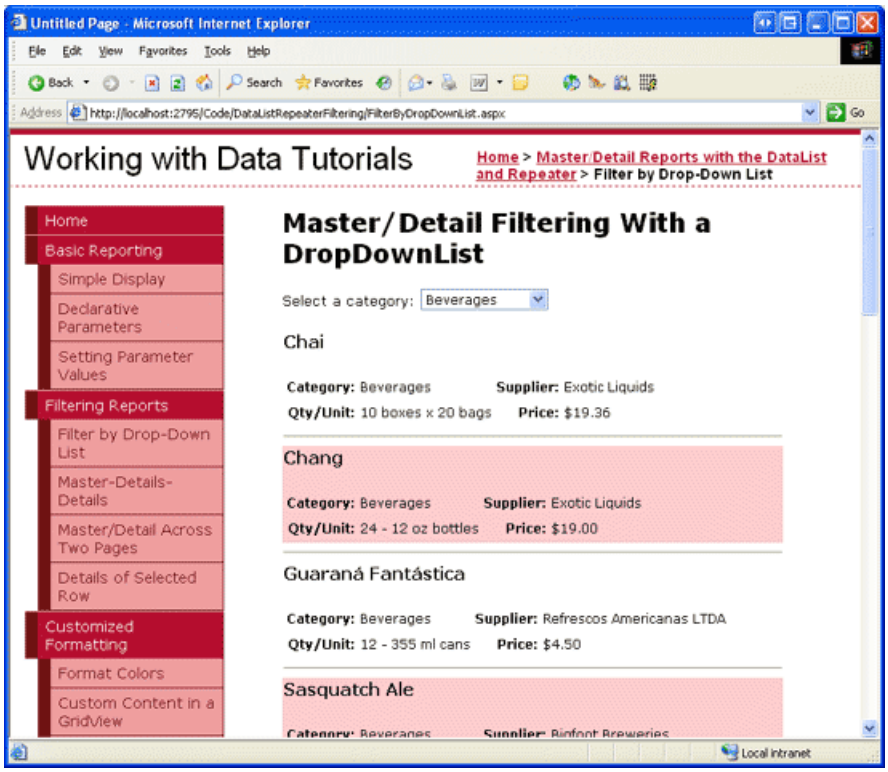


Figure 9: When First Visiting the Page, the Beverage Products are Displayed (Click to view full-size image)

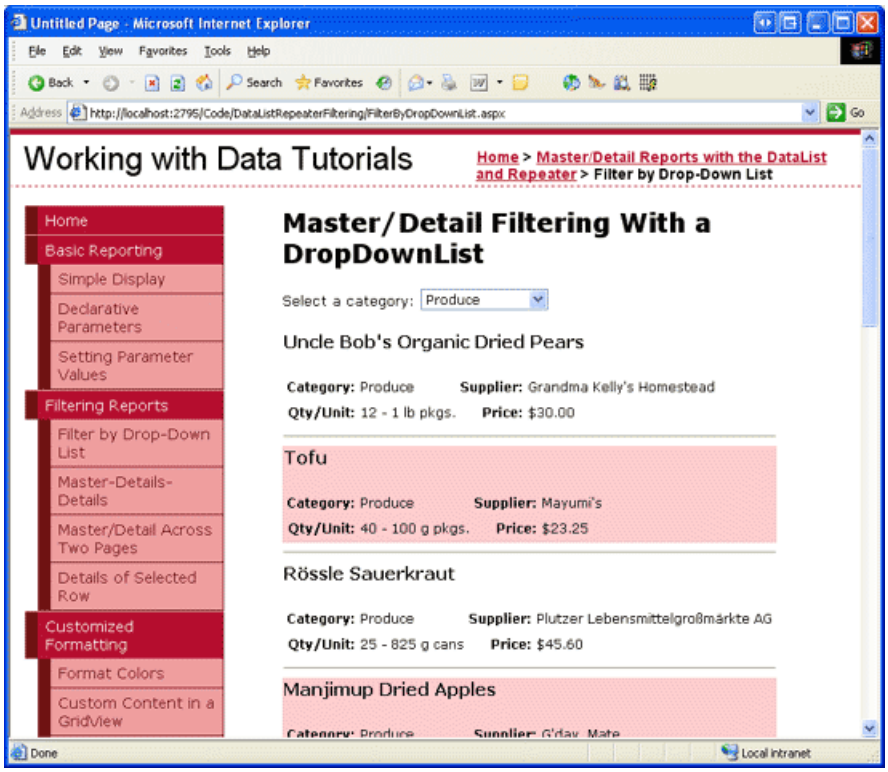


Figure 10: Selecting a New Product (Produce) Automatically Causes a PostBack, Updating the DataList (Click to view full-size image)

Adding a "-- Choose a Category --" List Item

When first visiting the `FilterByDropDownList.aspx` page the categories DropDownList's first list item (Beverages) is selected by default, showing the beverage products in the DataList. In the *Master/Detail Filtering With a DropDownList* tutorial we added a "-- Choose a Category --" option to the DropDownList that was selected by default and, when selected, displayed *all* of the products in the database. Such an approach was manageable when listing the products in a GridView, as each product row took up a small amount of screen real estate. With the DataList, however, each product's information consumes a much larger chunk of the screen. Let's still add a "--

Choose a Category --" option and have it selected by default, but instead of having it show all products when selected, let's configure it so that it shows no products.

To add a new list item to the DropDownList, go to the Properties window and click on the ellipses in the `Items` property. Add a new list item with the `Text` "-- Choose a Category --" and the `Value` `0`.

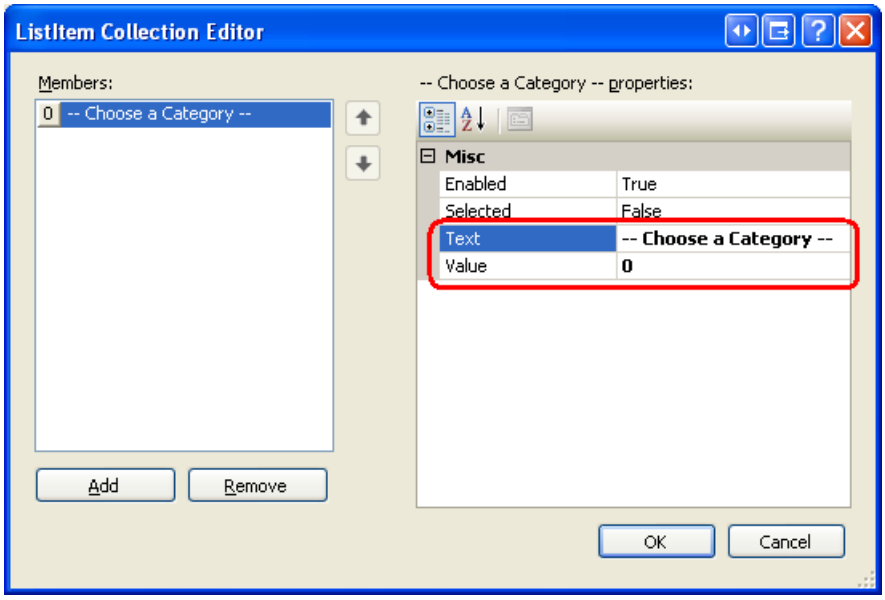


Figure 11: Add a "-- Choose a Category --" List Item

Alternatively, you can add the list item by adding the following markup to the DropDownList:

```
<asp:DropDownList ID="categories" runat="server" AutoPostBack="True"
    DataSourceID="CategoriesDataSource" DataTextField="CategoryName"
    DataValueField="CategoryID" EnableViewState="False">

    <asp:ListItem Value="0">-- Choose a Category --</asp:ListItem>

</asp:DropDownList>
```

Additionally, we need to set the DropDownList control's `AppendDataBoundItems` to `true` because if it's set to `false` (the default), when the categories are bound to the DropDownList from the ObjectDataSource they'll overwrite any manually-added list items.

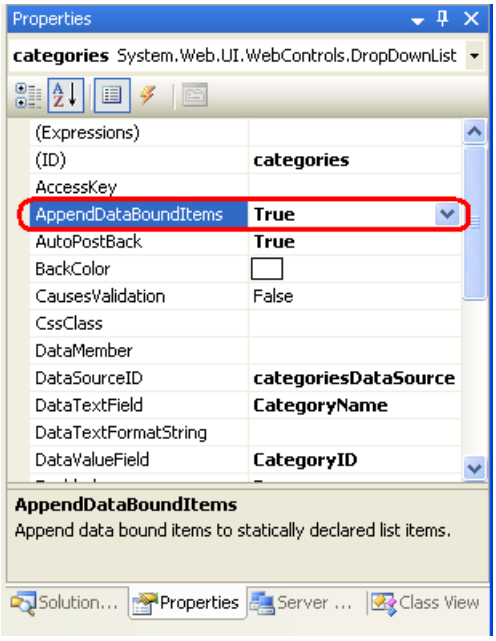


Figure 12: Set the `AppendDataBoundItems` Property to True

The reason we chose the value `0` for the "-- Choose a Category --" list item is because there are no categories in the system with a value of `0`, hence no product records will be returned when the "-- Choose a Category --" list item is selected. To confirm this, take a moment to visit the page through a browser. As Figure 13 shows, when initially viewing the page the "-- Choose a Category --" list item is selected and no products are displayed.

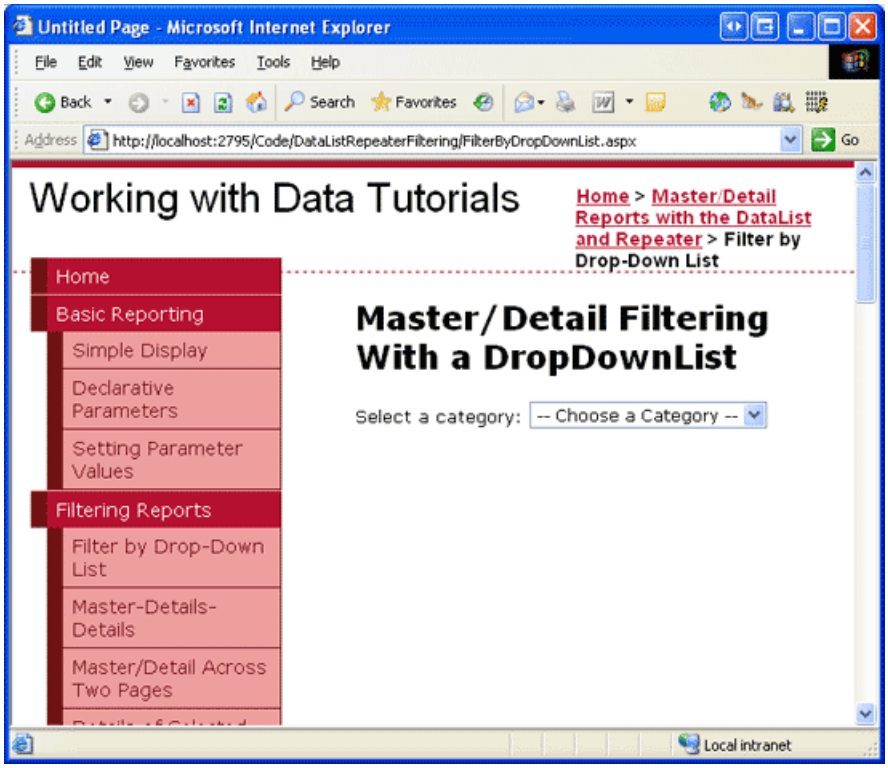


Figure 13: When the "-- Choose a Category --" List Item is Selected, No Products are Displayed (Click to view full-size image)

If you'd rather display *all* of the products when the "-- Choose a Category --" option is selected, use a value of `-1` instead. The astute reader will recall that back in the *Master/Detail Filtering With a DropDownList* tutorial we updated the `ProductsBLL` class's `GetProductsByCategoryID(categoryID)` method so that if a `categoryID` value of `-1` was passed in, all product records were returned.

Summary

When displaying hierarchically-related data, it often helps to present the data using master/detail reports, from which the user can start perusing the data from the top of the hierarchy and drill down into details. In this tutorial we examined building a simple master/detail report showing a selected category's products. This was accomplished by using a DropDownList for the list of categories and a DataList for the products belonging to the selected category.

In the next tutorial we'll look at separating the master and details records across two pages. In the first page, a list of "master" records will be displayed, with a link to view the details. Clicking on the link will whisk the user to the second page, which will display the details for the selected master record.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via

his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To…

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Randy Schmidt. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Master/Detail Filtering Across Two Pages (C#)

2/20/2020 • 13 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we look at how to separate a master/detail report across two pages. In the "master" page we use a Repeater control to render a list of categories that, when clicked, will take the user to the "details" page where a two-column DataList shows those products belonging to the selected category.

Introduction

In the preceding tutorial we saw how to display master/detail reports in a single web page using DropDownLists to display the "master" records and a DataList to display the "details." Another common pattern used for master/detail reports is to have the master records on one web page and the details on another. In the earlier Master/Detail Filtering Across Two Pages tutorial, we examined this pattern using a GridView to display all of the suppliers in the system. This GridView included a HyperLinkField, which rendered as a link to a second page, passing along the `SupplierID` in the querystring. The second page used a GridView to list those products provided by the selected supplier.

Such two-page master/detail reports can be accomplished using DataList and Repeater controls as well. The only difference is that neither the DataList nor the Repeater provides support for the HyperLinkField control. Instead, we must add a HyperLink Web control or an anchor HTML element (`<a>`) within the control's `ItemTemplate`. The HyperLink's `NavigateUrl` property or the anchor's `href` attribute can then be customized using declarative or programmatic approaches.

In this tutorial we'll explore an example that lists the categories in a bulleted list on one page using a Repeater control. Each list item will include the category's name and description, with the category name displayed as a link to a second page. Clicking on this link will whisk the user to the second page, where a DataList will show those products that belong to the selected category.

Step 1: Displaying the Categories in a Bulleted List

The first step in creating any master/detail report is to start by displaying the "master" records. Therefore, our first task is to display the categories in the "master" page. Open the `CategoryListMaster.aspx` page in the `DataListRepeaterFiltering` folder, add a Repeater control, and, from the smart tag, opt to add a new ObjectDataSource. Configure the new ObjectDataSource so that it accesses its data from the `CategoriesBLL` class's `GetCategories` method (see Figure 1).

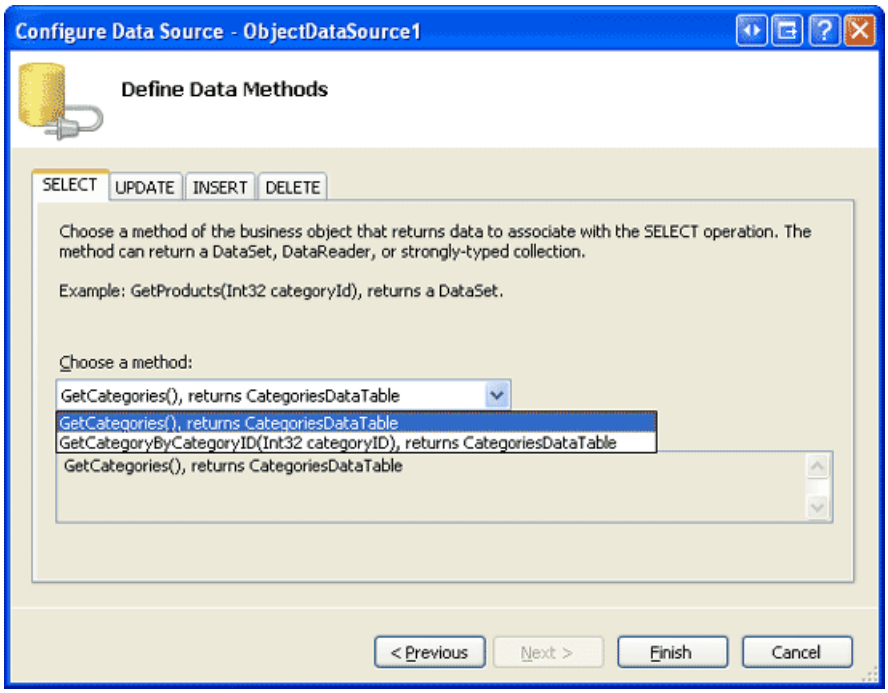


Figure 1: Configure the ObjectDataSource to Use the `CategoriesBLL` Class's `GetCategories` Method (Click to view full-size image)

Next, define the Repeater's templates such that it displays each category name and description as an item in a bulleted list. Let's not yet worry about having each category link to the details page. The following shows the declarative markup for the Repeater and ObjectDataSource:

```
<asp:Repeater ID="Repeater1" runat="server" DataSourceID="ObjectDataSource1"
    EnableViewState="False">
    <HeaderTemplate>
        <ul>
    </HeaderTemplate>

    <ItemTemplate>
        <li><%# Eval("CategoryName") %> - <%# Eval("Description") %></li>
    </ItemTemplate>

    <FooterTemplate>
        </ul>
    </FooterTemplate>
</asp:Repeater>

<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

With this markup complete, take a moment to view our progress through a browser. As Figure 2 shows, the Repeater renders as a bulleted list showing each category's name and description.

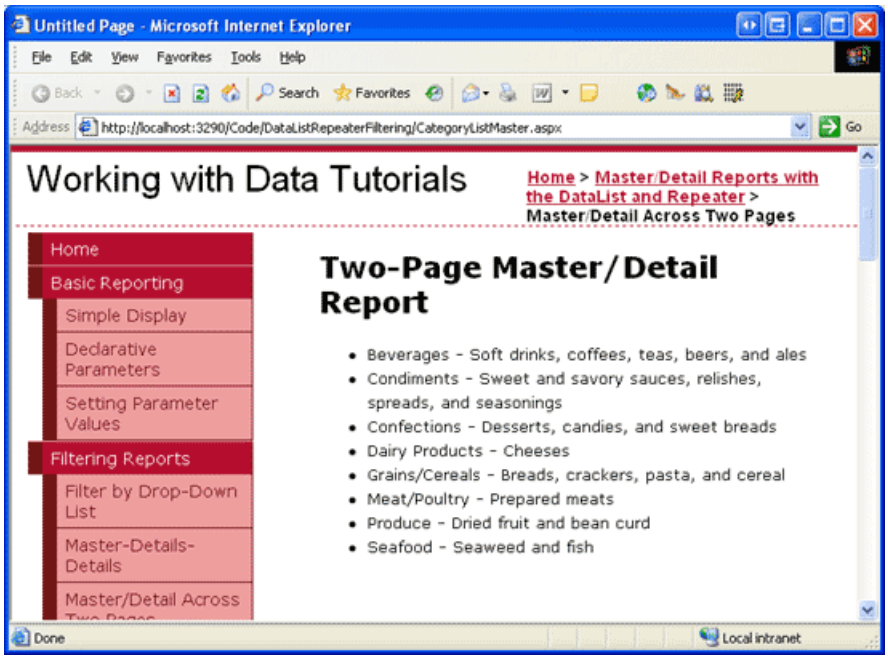


Figure 2: Each Category is Displayed as a Bulleted List Item (Click to view full-size image)

Step 2: Turning the Category Name into a Link to the Details Page

To allow a user to display the "details" information for a given category, we need to add a link to each bulleted list item that, when clicked, will take the user to the second page (`ProductsForCategoryDetails.aspx`). This second page will then display the products for the selected category using a DataList. In order to determine the category whose link was clicked, we need to pass the clicked category's `CategoryID` to the second page through some mechanism. The simplest, most straightforward way to transfer scalar data from one page to another is through the querystring, which is the option we'll use in this tutorial. In particular, the `ProductsForCategoryDetails.aspx` page will expect the selected *`categoryID`* value to be passed through a querystring field named `CategoryID`. For example, to view the products for the Beverages category, which has a `CategoryID` of 1, a user would visit `ProductsForCategoryDetails.aspx?CategoryID=1`.

To create a hyperlink for each bulleted list item in the Repeater we need to either add a HyperLink Web control or an HTML anchor element (`<a>`) to the `ItemTemplate`. In scenarios where the hyperlink is displayed the same for each row, either approach will suffice. For Repeaters, I prefer using the anchor element. To use the anchor element, update the Repeater's ItemTemplate to:

```
<li>
    <a href='ProductsForCategoryDetails.aspx?CategoryID=<%# Eval("CategoryID") %>'>
        <%# Eval("CategoryName") %>
    </a> - <%# Eval("Description") %>
</li>
```

Note that the `CategoryID` can be injected directly within the anchor element's `href` attribute; however, to do so be certain to delimit the `href` attribute's value with apostrophes (and note quotation marks) since the `Eval` method within the `href` attribute delimits its string (`"CategoryID"`) with quotation marks. Alternatively, a HyperLink Web control can be used instead:

```
<li>
    <asp:HyperLink runat="server" Text='<%# Eval("CategoryName") %>'
        NavigateUrl='<%# "ProductsForCategoryDetails.aspx?CategoryID=" &
            Eval("CategoryID") %>'>
    </asp:HyperLink>
    - <%# Eval("Description") %>
</li>
```

Note how the static portion of the URL — `ProductsForCategoryDetails.aspx?CategoryID` — is appended to the result of `Eval("CategoryID")` directly within the databinding syntax using string concatenation.

One benefit of using the HyperLink control is that it can be programmatically accessed from the Repeater's `ItemDataBound` event handler, if needed. For example, you might want to display the category name as text rather than as a link for categories with no associated products. Such a check could be programmatically performed in the `ItemDataBound` event handler; for categories with no associated products, the HyperLink's `NavigateUrl` property could be set to a blank string, thereby resulting in that particular category name rendering as plain text (rather than as a link). Refer back to the Formatting the DataList and Repeater Based Upon Data tutorial for more information on formatting the DataList and Repeater's contents based on programmatic logic through the `ItemDataBound` event handler.

If you are following along, feel free to use either the anchor element or HyperLink control approach in your page. Regardless of the approach, when viewing the page through a browser each category name should be rendered as a link to `ProductsForCategoryDetails.aspx`, passing in the applicable `CategoryID` value (see Figure 3).

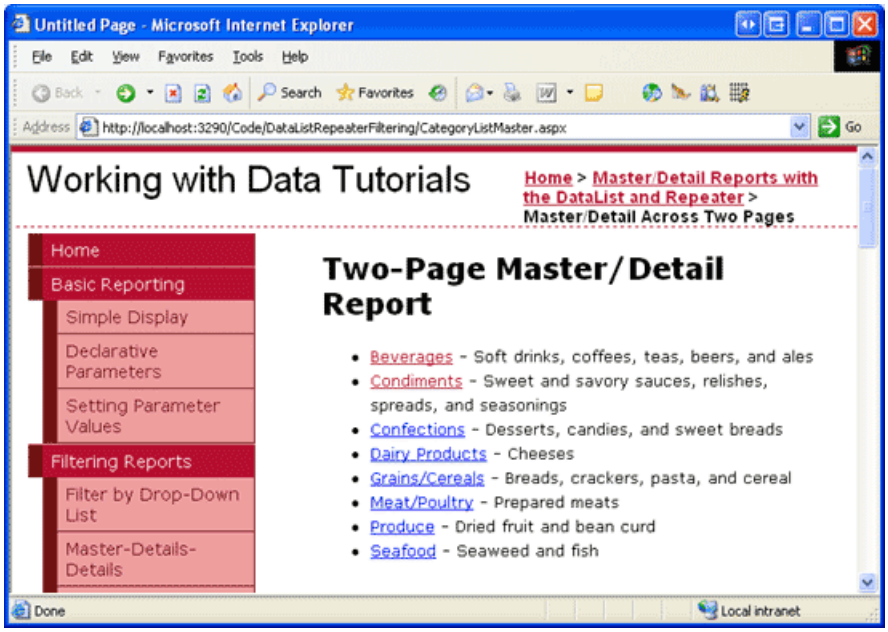


Figure 3: The Category Names Now Link to `ProductsForCategoryDetails.aspx` (Click to view full-size image)

Step 3: Listing the Products that Belong to the Selected Category

With the `CategoryListMaster.aspx` page complete, we're ready to turn our attention to implementing the "details" page, `ProductsForCategoryDetails.aspx`. Open this page, drag a DataList from the Toolbox onto the Designer, and set its `ID` property to `ProductsInCategory`. Next, from the DataList's smart tag choose to add a new ObjectDataSource to the page, naming it `ProductsInCategoryDataSource`. Configure it such that it calls the `ProductsBLL` class's `GetProductsByCategoryID(categoryID)` method; set the drop-down lists in the INSERT, UPDATE, and DELETE tabs to (None).

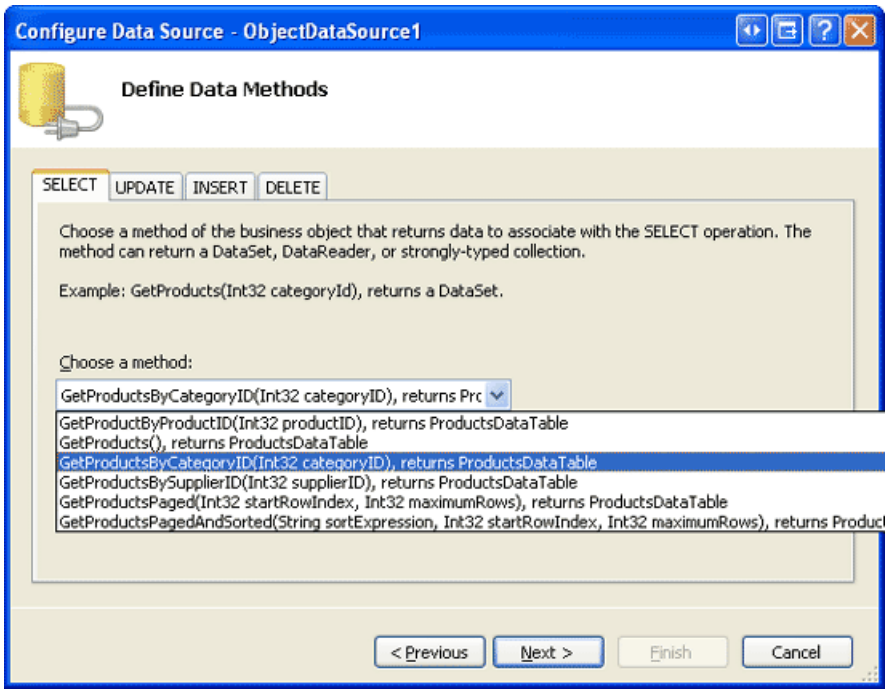


Figure 4: Configure the ObjectDataSource to Use the `ProductsBLL` Class's `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

Since the `GetProductsByCategoryID(categoryID)` method accepts an input parameter (*`categoryID`*), the Choose Data Source wizard offers us an opportunity to specify the parameter's source. Set the parameter source to QueryString using the QueryStringField `CategoryID`.

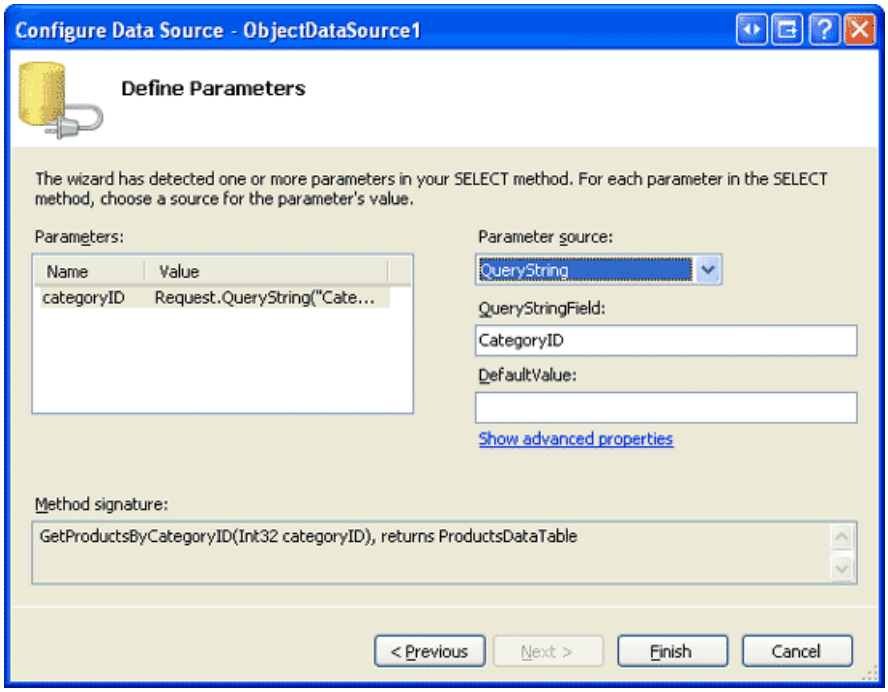


Figure 5: Use the Querystring Field `CategoryID` as the Parameter's Source (Click to view full-size image)

As we've seen in previous tutorials, after completing the Choose Data Source wizard, Visual Studio automatically creates an `ItemTemplate` for the DataList that lists each data field name and value. Replace this template with one that lists only the product's name, supplier, and price. Also, set the DataList's `RepeatColumns` property to 2. After these changes, your DataList and ObjectDataSource's declarative markup should look similar to the following:

```
<asp:DataList ID="ProductsInCategory" runat="server" DataKeyField="ProductID"
    RepeatColumns="2" DataSourceID="ProductsInCategoryDataSource"
    EnableViewState="False">
    <ItemTemplate>
        <h5><%# Eval("ProductName") %></h5>
        <p>
            Supplied by <%# Eval("SupplierName") %><br />
            <%# Eval("UnitPrice", "{0:C}") %>
        </p>
    </ItemTemplate>
</asp:DataList>

<asp:ObjectDataSource ID="ProductsInCategoryDataSource"
    OldValuesParameterFormatString="original_{0}" runat="server"
    SelectMethod="GetProductsByCategoryID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:QueryStringParameter Name="categoryID" QueryStringField="CategoryID"
            Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

To view this page in action, start from the `CategoryListMaster.aspx` page; next, click on a link in the categories bulleted list. Doing so will take you to `ProductsForCategoryDetails.aspx`, passing along the `CategoryID` through the querystring. The `ProductsInCategoryDataSource` ObjectDataSource in `ProductsForCategoryDetails.aspx` will then get just those products for the specified category and display them in the DataList, which renders two products per row. Figure 6 shows a screenshot of `ProductsForCategoryDetails.aspx` when viewing the Beverages.

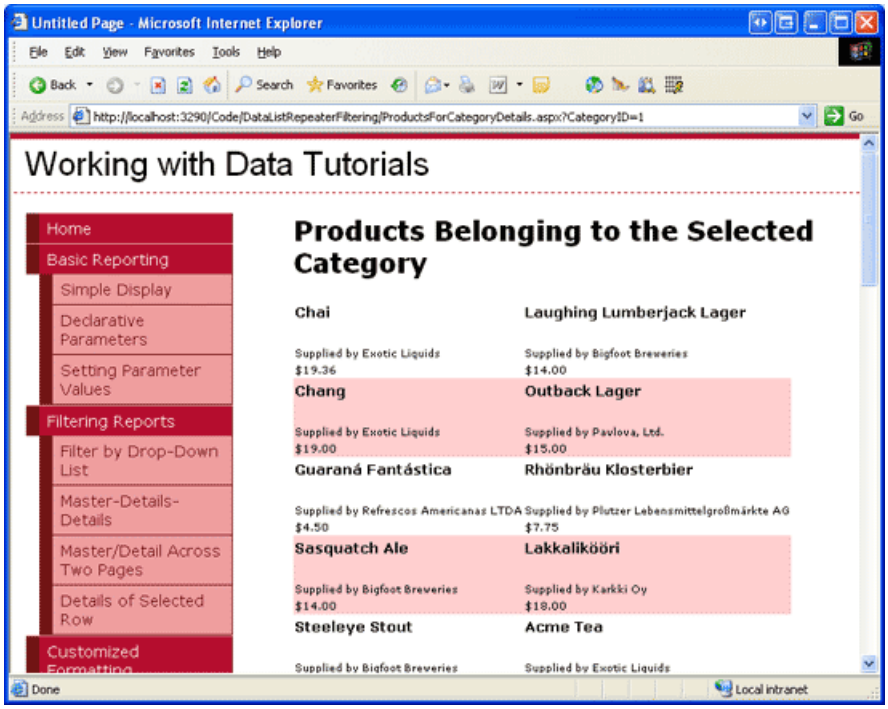


Figure 6: The Beverages are Displayed, Two per Row (Click to view full-size image)

Step 4: Displaying Category Information on ProductsForCategoryDetails.aspx

When a user clicks on a category in `CategoryListMaster.aspx`, they are taken to `ProductsForCategoryDetails.aspx` and shown the products that belong to the selected category. However, in `ProductsForCategoryDetails.aspx` there are no visual cues as to what category was selected. A user that meant to click Beverages, but accidentally clicked Condiments, has no way of realizing their mistake once they reach `ProductsForCategoryDetails.aspx`. To alleviate this potential problem, we can display information about the selected category — its name and description — at

the top of the `ProductsForCategoryDetails.aspx` page.

To accomplish this, add a FormView above the Repeater control in `ProductsForCategoryDetails.aspx`. Next, add a new ObjectDataSource to the page from the FormView's smart tag named `CategoryDataSource` and configure it to use the `CategoriesBLL` class's `GetCategoryByCategoryID(categoryID)` method.

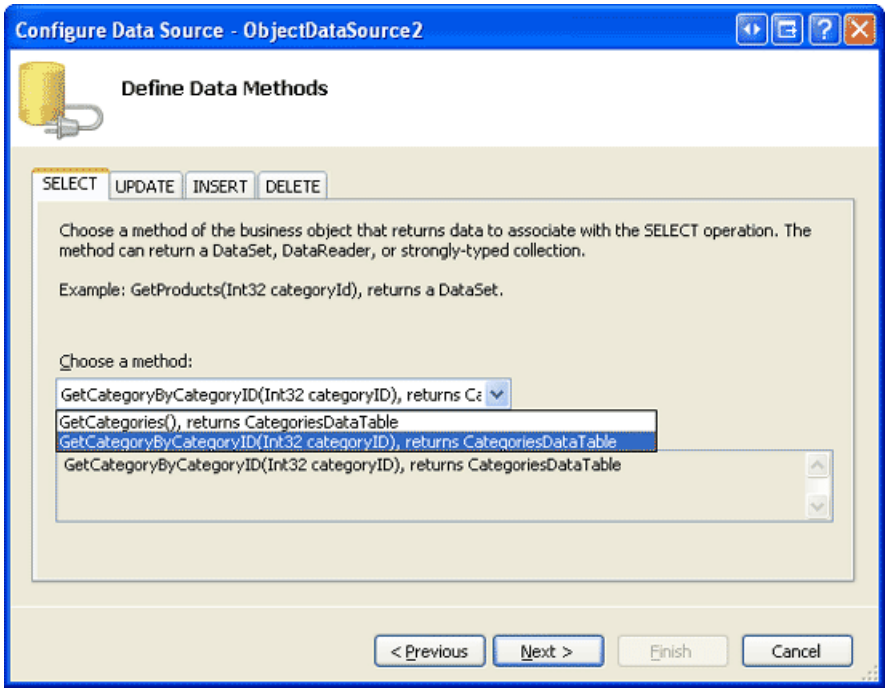


Figure 7: Access Information about the Category through the `CategoriesBLL` Class's `GetCategoryByCategoryID(categoryID)` Method (Click to view full-size image)

As with the `ProductsInCategoryDataSource` ObjectDataSource added in Step 3, the `CategoryDataSource`'s Configure Data Source wizard prompts us for a source for the `GetCategoryByCategoryID(categoryID)` method's input parameter. Use the exact same settings as before, setting the parameter source to QueryString and the QueryStringField value to `CategoryID` (refer back to Figure 5).

After completing the wizard, Visual Studio automatically creates an `ItemTemplate`, `EditItemTemplate`, and `InsertItemTemplate` for the FormView. Since we're providing a read-only interface, feel free to remove the `EditItemTemplate` and `InsertItemTemplate`. Also, feel free to customize the FormView's `ItemTemplate`. After removing the superfluous templates and customizing the ItemTemplate, your FormView and ObjectDataSource's declarative markup should look similar to the following:

```
<asp:FormView ID="FormView1" runat="server" DataKeyNames="CategoryID"
    DataSourceID="CategoryDataSource" EnableViewState="False" Width="100%">
    <ItemTemplate>
        <h3>
            <asp:Label ID="CategoryNameLabel" runat="server"
                Text='<%# Bind("CategoryName") %>' />
        </h3>
        <p>
            <asp:Label ID="DescriptionLabel" runat="server"
                Text='<%# Bind("Description") %>' />
        </p>
    </ItemTemplate>
</asp:FormView>

<asp:ObjectDataSource ID="CategoryDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategoryByCategoryID" TypeName="CategoriesBLL">
    <SelectParameters>
        <asp:QueryStringParameter Name="categoryID" Type="Int32"
            QueryStringField="CategoryID" />
    </SelectParameters>
</asp:ObjectDataSource>
```

Figure 8 shows a screen shot when viewing this page through a browser.

> **NOTE**
>
> In addition to the FormView, I've also added a HyperLink control above the FormView that will take the user back to the list of categories (`CategoryListMaster.aspx`). Feel free to place this link elsewhere or to omit it altogether.

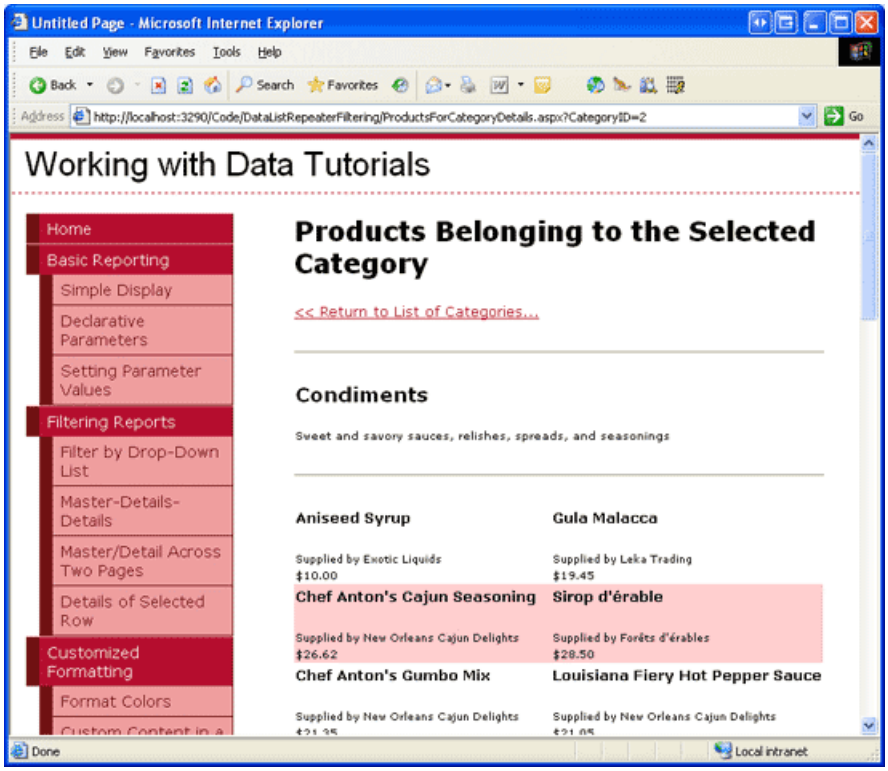


Figure 8: Category Information is Now Displayed at the Top of the Page (Click to view full-size image)

Step 5: Displaying a Message if No Products Belong to the Selected Category

The `CategoryListMaster.aspx` page lists all categories in the system, regardless of whether there are any

associated products. If a user clicks on a category with no associated products, the DataList in `ProductsForCategoryDetails.aspx` will not be rendered, as its data source will not have any items. As we've seen in past tutorials, the GridView provides an `EmptyDataText` property that can be used to specify a text message to display if there are no records in its data source. Unfortunately, neither the DataList nor Repeater has such a property.

In order to display a message informing the user that there are no matching products for the selected category, we need to add a Label control to the page whose `Text` property is assigned the message to display in the event that there are no matching products. We then need to programmatically set its `Visible` property based on whether or not the DataList contains any items.

To accomplish this, start by adding a Label beneath the DataList. Set its `ID` property to `NoProductsMessage` and its `Text` property to "There are no products for the selected category..." Next, we need to programmatically set this Label's `Visible` property based on whether or not any data was bound to the `ProductsInCategory` DataList. This assignment must be made after the data has been bound to the DataList. For the GridView, DetailsView, and FormView, we could create an event handler for the control's `DataBound` event, which fires after databinding has completed. However, neither the DataList nor the Repeater has a `DataBound` event available.

For this particular example we can assign the Label's `Visible` property in the `Page_Load` event handler, since the data will have been assigned to the DataList prior to the page's `Load` event. However, this approach would not work in the general case, as the data from the ObjectDataSource might be bound to the DataList later in the page's lifecycle. For example, if the displayed data is based upon the value in another control, such as it is when displaying a master/detail report using a DropDownList to hold the "master" records, the data may not rebound to the data Web control until the `PreRender` stage in the page's life cycle.

One solution which will work for all cases is to assign the `Visible` property to `False` in the DataList's `ItemDataBound` (or `ItemCreated`) event handler when binding an item type of `Item` or `AlternatingItem`. In such a case we know that there is at least one data item in the data source and therefore can hide the `NoProductsMessage` Label. In addition to this event handler, we also need an event handler for the DataList's `DataBinding` event, where we initialize the Label's `Visible` property to `True`. Since the `DataBinding` event fires before the `ItemDataBound` events, the Label's `Visible` property will initially be set to `True`; if there are any data items, however, it will be set to `False`. The following code implements this logic:

```
protected void ProductsInCategory_DataBinding(object sender, EventArgs e)
{
    // Show the Label
    NoProductsMessage.Visible = true;
}

protected void ProductsInCategory_ItemDataBound(object sender, DataListItemEventArgs e)
{
    // If we have a data item, hide the Label
    if (e.Item.ItemType == ListItemType.Item ||
        e.Item.ItemType == ListItemType.AlternatingItem)
        NoProductsMessage.Visible = false;
}
```

All of the categories in the Northwind database are associated with one or more products. To test this feature, I've manually adjusted the Northwind database for this tutorial, reassigning all products associated with the Produce category (`CategoryID` = 7) to the Seafood category (`CategoryID` = 8). This can be accomplished from the Server Explorer by choosing New Query and using the following `UPDATE` statement:

```
UPDATE Products SET
    CategoryID = 8
WHERE CategoryID = 7
```

After updating the database accordingly, return to the `CategoryListMaster.aspx` page and click on the Produce link. Since there are no longer any products belonging to the Produce category, you should see the "There are no products for the selected category..." message, as shown in Figure 9.

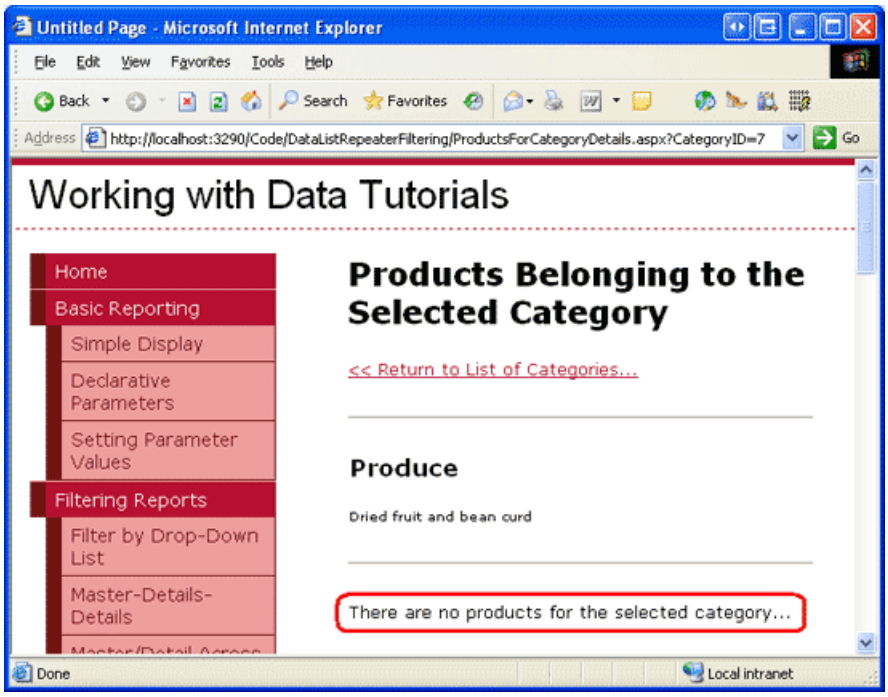


Figure 9: A Message is Displayed if there are No Products Belonging to the Selected Category (Click to view full-size image)

Summary

While master/detail reports can display both the master and detail records on a single page, in many websites they are separated out across two web pages. In this tutorial we looked at how to implement such a master/detail report by having the categories listed in a bulleted list using a Repeater in the "master" web page and the associated products listed in the "details" page. Each list item in the master web page contained a link to the details page that passed along the row's `CategoryID` value.

In the details page retrieving those products for the specified supplier was accomplished through the `ProductsBLL` class's `GetProductsByCategoryID(categoryID)` method. The `categoryID` parameter value was specified declaratively using the `CategoryID` querystring value as the parameter source. We also looked at how to display category details in the details page using a FormView and how to display a message if there were no products belonging to the selected category.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Zack Jones and Liz Shulok. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Master/Detail Using a Bulleted List of Master Records with a Details DataList (C#)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll compress the two-page master/detail report of the previous tutorial into a single page, showing a bulleted list of category names on the left side of the screen and the selected category's products on the right of the screen.

Introduction

In the preceding tutorial we looked at how to separate a master/detail report across two pages. In the master page we used a Repeater control to render a bulleted list of categories. Each category name was a hyperlink that, when clicked, would take the user to the details page, where a two-column DataList showed those products belonging to the selected category.

In this tutorial we'll compress the two-page tutorial into a single page, showing a bulleted list of category names on the left side of the screen with each category name rendered as a LinkButton. Clicking one of the category name LinkButtons induces a postback and binds the selected category s products to a two-column DataList on the right of the screen. In addition to displaying each category s name, the Repeater on the left shows how many total products there are for a given category (see Figure 1).

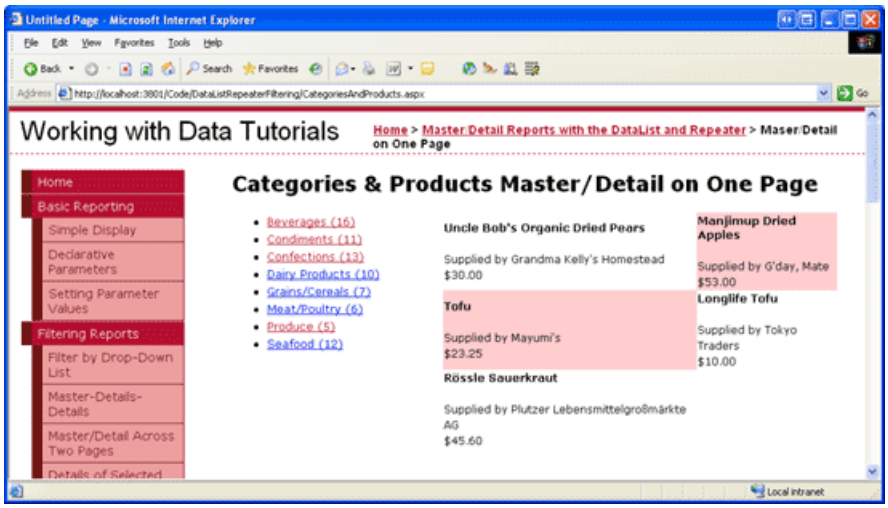


Figure 1: The Category s Name and Total Number of Products are Displayed on the Left (Click to view full-size image)

Step 1: Displaying a Repeater in the Left Portion of the Screen

For this tutorial we need to have the bulleted list of categories appear to the left of the selected category s products. Content within a web page can be positioned using standard HTML elements paragraph tags, non-breaking spaces, `<table>` s, and so on or through cascading stylesheet (CSS) techniques. All of our tutorials thus far have used CSS techniques for positioning. When we built the navigation user interface in our master page in the Master Pages and Site Navigation tutorial we used *absolute positioning*, indicating the precise pixel offset for the navigation list and the main content. Alternatively, CSS can be used to position one element to the right or

left of another through *floating*. We can have the bulleted list of categories appear to the left of the selected category s products by floating the Repeater to the left of the DataList

Open the `CategoriesAndProducts.aspx` page from the `DataListRepeaterFiltering` folder and add to the page a Repeater and a DataList. Set the Repeater s `ID` to `Categories` and the DataList s to `CategoryProducts`. Go to the Source view and put the Repeater and DataList controls within their own `<div>` elements. That is, enclose the Repeater within a `<div>` element first and then the DataList in its own `<div>` element directly after the Repeater. Your markup at this point should look similar to the following:

```
<div>
    <asp:Repeater ID="Categories" runat="server">
    </asp:Repeater>
</div>
<div>
    <asp:DataList ID="CategoryProducts" runat="server">
    </asp:DataList>
</div>
```

To float the Repeater to the left of the DataList, we need to use the `float` CSS style attribute, like so:

```
<div>
    Repeater
</div>
<div>
    DataList
</div>
```

The `float: left;` floats the first `<div>` element to the left of the second one. The `width` and `padding-right` settings indicate the first `<div>` s `width` and how much padding is added between the `<div>` element s content and its right margin. For more information on floating elements in CSS check out the Floatutorial.

Rather than specify the style setting directly through the first `<p>` element s `style` attribute, let s instead create a new CSS class in `Styles.css` named `FloatLeft`:

```
.FloatLeft
{
    float: left;
    width: 33%;
    padding-right: 10px;
}
```

Then we can replace the `<div>` with `<div class="FloatLeft">`.

After adding the CSS class and configuring the markup in the `CategoriesAndProducts.aspx` page, go to the Designer. You should see the Repeater floating to the left of the DataList (although right now both just appear as gray boxes since we ve yet to configure their data sources or templates).

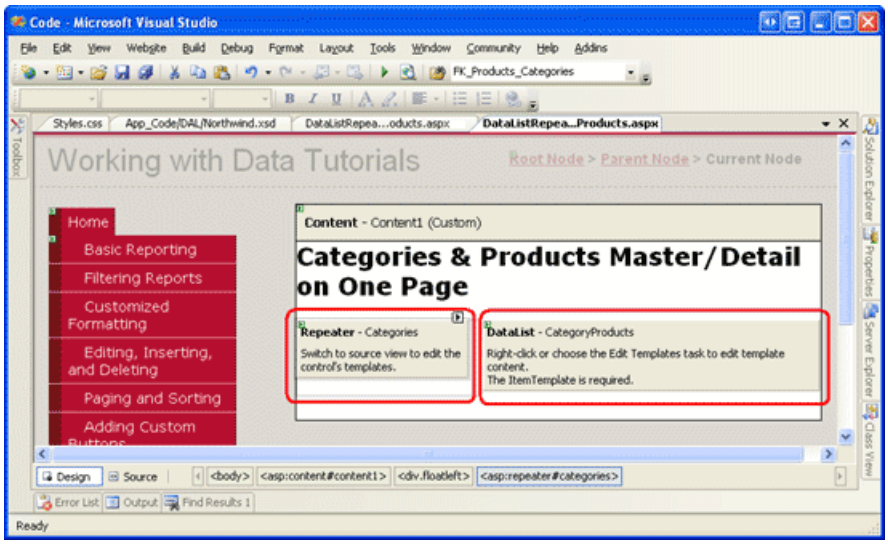


Figure 2: The Repeater is Floated to the Left of the DataList (Click to view full-size image)

Step 2: Determining the Number of Products for Each Category

With the Repeater and DataList s surrounding markup complete, we re ready to bind the category data to the Repeater control. However, as the bulleted list of categories in Figure 1 shows, in addition to each category s name we also need to display the number of products associated with the category. To access this information we can either:

- **Determine this information from the ASP.NET page s code-behind class.** Given a particular `categoryID` we can determine the number of associated products by calling the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method. This method returns a `ProductsDataTable` object whose `Count` property indicates how many `ProductsRow` s exists, which is the number of products for the specified `categoryID`. We can create an `ItemDataBound` event handler for the Repeater that, for each category bound to the Repeater, calls the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method and includes its count in the output.
- **Update the `CategoriesDataTable` in the Typed DataSet to include a `NumberOfProducts` column.** We can then update the `GetCategories()` method in the `CategoriesDataTable` to include this information or, alternatively, leave `GetCategories()` as-is and create a new `CategoriesDataTable` method called `GetCategoriesAndNumberOfProducts()`.

Let s explore both of these techniques. The first approach is simpler to implement since we don t need to update the Data Access Layer; however, it requires more communications with the database. The call to the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method in the `ItemDataBound` event handler adds an extra database call for each category displayed in the Repeater. With this technique there are $N + 1$ database calls, where N is the number of categories displayed in the Repeater. With the second approach, the product count is returned with information about each category from the `CategoriesBLL` class s `GetCategories()` (or `GetCategoriesAndNumberOfProducts()`) method, thereby resulting in just one trip to the database.

Determining the Number of Products in the ItemDataBound Event Handler

Determining the number of products for each category in the Repeater s `ItemDataBound` event handler does not require any modifications to our existing Data Access Layer. All modifications can be made directly within the `CategoriesAndProducts.aspx` page. Start by adding a new ObjectDataSource named `CategoriesDataSource` via the Repeater s smart tag. Next, configure the `CategoriesDataSource` ObjectDataSource so that it retrieves its data from the `CategoriesBLL` class s `GetCategories()` method.

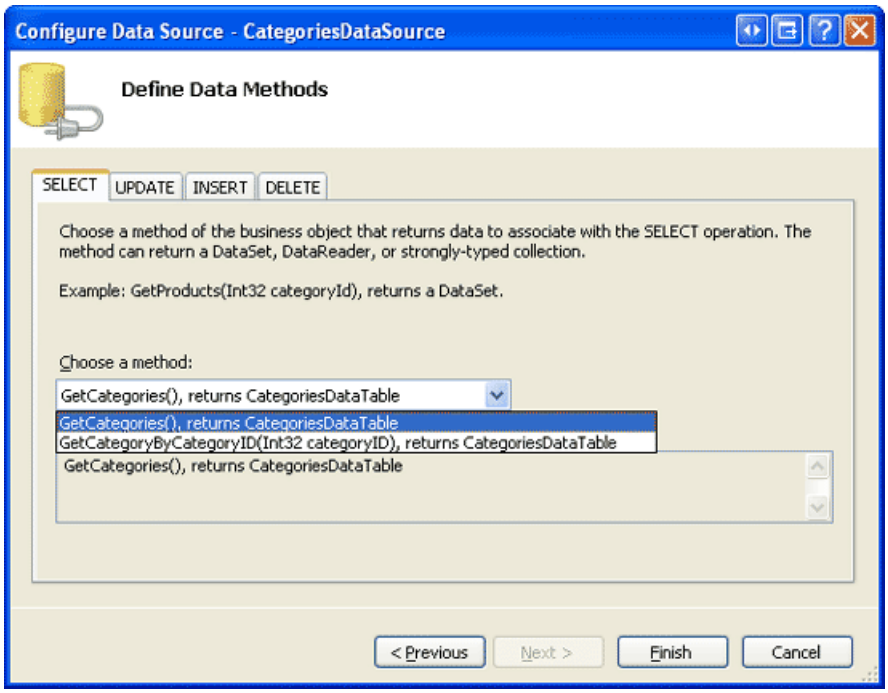


Figure 3: Configure the ObjectDataSource to Use the `CategoriesBLL` class s `GetCategories()` Method (Click to view full-size image)

Each item in the `Categories` Repeater needs to be clickable and, when clicked, cause the `CategoryProducts` DataList to display those products for the selected category. This can be accomplished by making each category a hyperlink, linking back to this same page (`CategoriesAndProducts.aspx`), but passing the `CategoryID` through the querystring, much like we saw in the previous tutorial. The advantage of this approach is that a page displaying a particular category s products can be bookmarked and indexed by a search engine.

Alternatively, we can make each category a LinkButton, which is the approach we'll use for this tutorial. The LinkButton renders in the user s browser as a hyperlink but, when clicked, induces a postback; on postback, the DataList s ObjectDataSource needs to be refreshed to display those products belonging to the selected category. For this tutorial, using a hyperlink makes more sense than using a LinkButton; however, there may be other scenarios where using a LinkButton is more advantageous. While the hyperlink approach would be ideal for this example, let s instead explore using the LinkButton. As we'll see, using a LinkButton introduces some challenges that would not otherwise arise with a hyperlink. Therefore, using a LinkButton in this tutorial will highlight these challenges and help provide solutions for those scenarios where we may want to use a LinkButton instead of a hyperlink.

> **NOTE**
>
> You are encouraged to repeat this tutorial using a HyperLink control or `<a>` element in lieu of the LinkButton.

The following markup shows the declarative syntax for the Repeater and the ObjectDataSource. Note that the Repeater s templates render a bulleted list with each item as a LinkButton:

```
<asp:Repeater ID="Categories" runat="server" DataSourceID="CategoriesDataSource">
    <HeaderTemplate>
        <ul>
    </HeaderTemplate>
    <ItemTemplate>
        <li><asp:LinkButton runat="server" ID="ViewCategory" /></li>
    </ItemTemplate>
    <FooterTemplate>
        </ul>
    </FooterTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

NOTE

For this tutorial the Repeater must have its view state enabled (note the omission of the `EnableViewState="False"` from the Repeater s declarative syntax). In step 3 we'll be creating an event handler for the Repeater s `ItemCommand` event in which we'll be updating the DataList s ObjectDataSource s `SelectParameters` collection. The Repeater s `ItemCommand`, however, won't fire if view state is disabled. See A Stumper of an ASP.NET Question and its solution for more information on why view state must be enabled for a Repeater s `ItemCommand` event to fire.

The LinkButton with the `ID` property value of `ViewCategory` does not have its `Text` property set. If we had just wanted to display the category name, we would have set the Text property declaratively, through databinding syntax, like so:

```
<asp:LinkButton runat="server" ID="ViewCategory"
    Text='<%# Eval("CategoryName") %>' />
```

However, we want to show both the category s name *and* the number of products belonging to that category. This information can be retrieved from the Repeater s `ItemDataBound` event handler by making a call to the `ProductBLL` class s `GetCategoriesByProductID(categoryID)` method and determining how many records are returned in the resulting `ProductsDataTable`, as the following code illustrates:

```
protected void Categories_ItemDataBound(object sender, RepeaterItemEventArgs e)
{
    // Make sure we're working with a data item...
    if (e.Item.ItemType == ListItemType.Item ||
        e.Item.ItemType == ListItemType.AlternatingItem)
    {
        // Reference the CategoriesRow instance bound to this RepeaterItem
        Northwind.CategoriesRow category =
            (Northwind.CategoriesRow) ((System.Data.DataRowView) e.Item.DataItem).Row;
        // Determine how many products are in this category
        NorthwindTableAdapters.ProductsTableAdapter productsAPI =
            new NorthwindTableAdapters.ProductsTableAdapter();
        int productCount =
            productsAPI.GetProductsByCategoryID(category.CategoryID).Count;
        // Reference the ViewCategory LinkButton and set its Text property
        LinkButton ViewCategory = (LinkButton)e.Item.FindControl("ViewCategory");
        ViewCategory.Text =
            string.Format("{0} ({1:N0})", category.CategoryName, productCount);
    }
}
```

We start out by ensuring that we re working with a data item (one whose `ItemType` is `Item` or `AlternatingItem`)

and then reference the `CategoriesRow` instance that has just been bound to the current `RepeaterItem`. Next, we determine the number of products for this category by creating an instance of the `ProductsBLL` class, calling its `GetCategoriesByProductID(categoryID)` method, and determining the number of records returned using the `Count` property. Finally, the `ViewCategory` LinkButton in the ItemTemplate is references and its `Text` property is set to *CategoryName* (*NumberOfProductsInCategory*), where *NumberOfProductsInCategory* is formatted as a number with zero decimal places.

> **NOTE**
>
> Alternatively, we could have added a *formatting function* to the ASP.NET page s code-behind class that accepts a category s `CategoryName` and `CategoryID` values and returns the `CategoryName` concatenated with the number of products in the category (as determined by calling the `GetCategoriesByProductID(categoryID)` method). The results of such a formatting function could be declaratively assigned to the LinkButton s Text property replacing the need for the `ItemDataBound` event handler. Refer to the Using TemplateFields in the GridView Control or Formatting the DataList and Repeater Based Upon Data tutorials for more information on using formatting functions.

After adding this event handler, take a moment to test the page through a browser. Note how each category is listed in a bulleted list, displaying the category s name and the number of products associated with the category (see Figure 4).

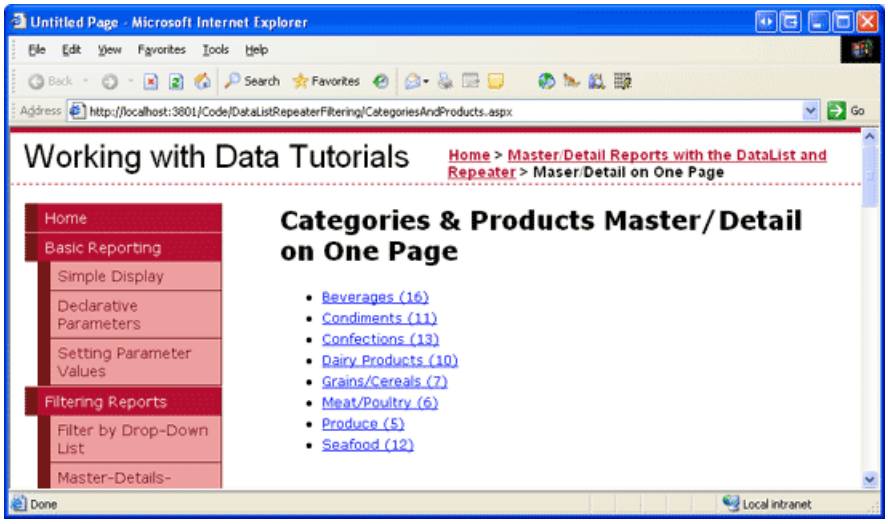


Figure 4: Each Category s Name and Number of Products are Displayed (Click to view full-size image)

Updating the `CategoriesDataTable` and `CategoriesTableAdapter` to Include the Number of Products for Each Category

Rather than determining the number of products for each category as it s bound to the Repeater, we can streamline this process by adjusting the `CategoriesDataTable` and `CategoriesTableAdapter` in the Data Access Layer to include this information natively. To achieve this, we must add a new column to `CategoriesDataTable` to hold the number of associated products. To add a new column to a DataTable, open the Typed DataSet (`App_Code\DAL\Northwind.xsd`), right-click on the DataTable to modify, and choose Add / Column. Add a new column to the `CategoriesDataTable` (see Figure 5).

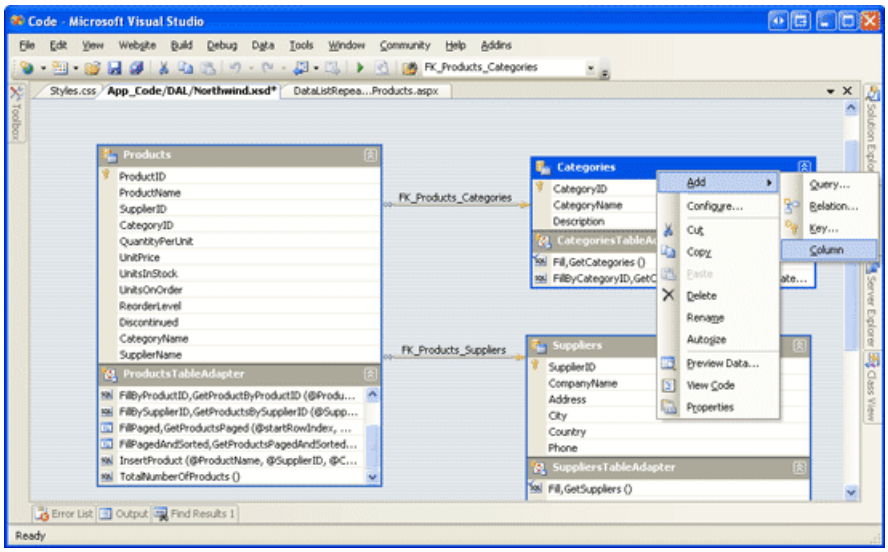


Figure 5: Add a New Column to the `CategoriesDataSource` (Click to view full-size image)

This will add a new column named `Column1`, which you can change by simply typing in a different name. Rename this new column to `NumberOfProducts`. Next, we need to configure this column s properties. Click on the new column and go to the Properties window. Change the column s `DataType` property from `System.String` to `System.Int32` and set the `ReadOnly` property to `True`, as shown in Figure 6.

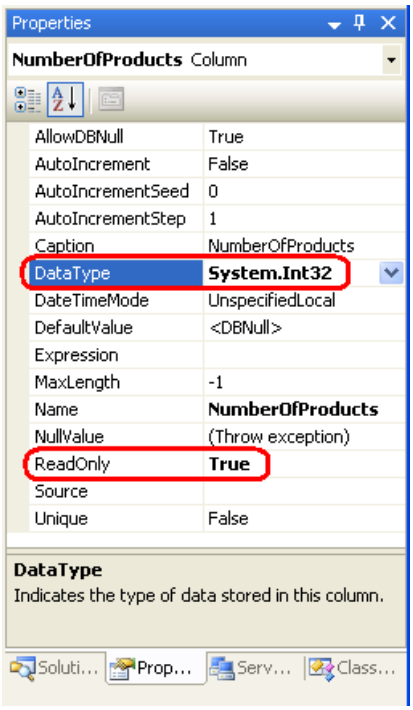


Figure 6: Set the `DataType` and `ReadOnly` Properties of the New Column

While the `CategoriesDataTable` now has a `NumberOfProducts` column, its value is not set by any of the corresponding TableAdapter s queries. We can update the `GetCategories()` method to return this information if we want such information returned every time category information is retrieved. If, however, we only need to grab the number of associated products for the categories in rare instances (such as just for this tutorial), then we can leave `GetCategories()` as-is and create a new method that returns this information. Let s use this latter approach, creating a new method named `GetCategoriesAndNumberOfProducts()`.

To add this new `GetCategoriesAndNumberOfProducts()` method, right-click on the `CategoriesTableAdapter` and choose New Query. This brings up the TableAdapter Query Configuration Wizard, which we ve used numerous times in previous tutorials. For this method, start the wizard by indicating that the query uses an ad-hoc SQL statement that returns rows.

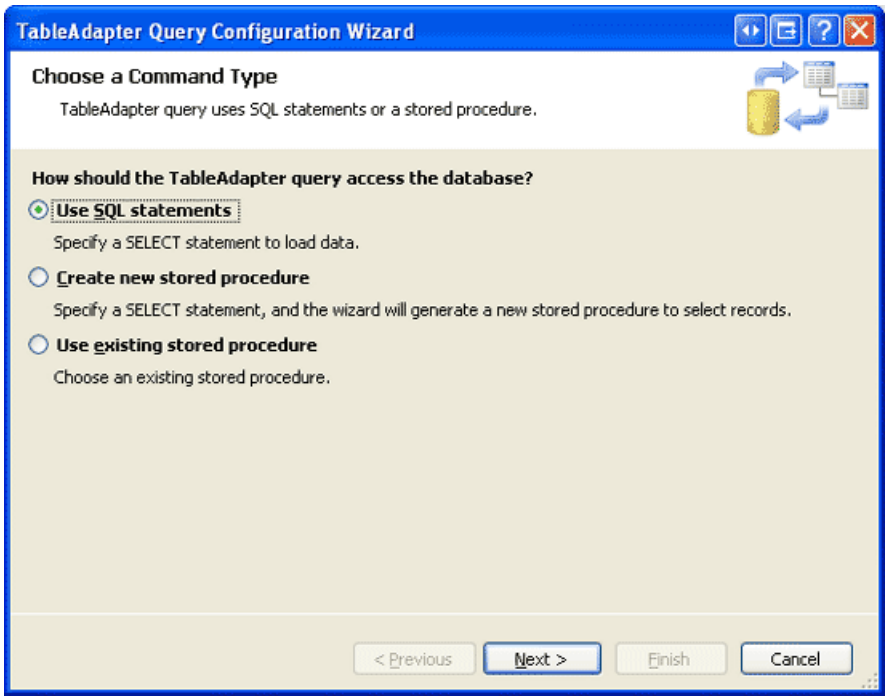


Figure 7: Create the Method Using an Ad-Hoc SQL Statement (Click to view full-size image)

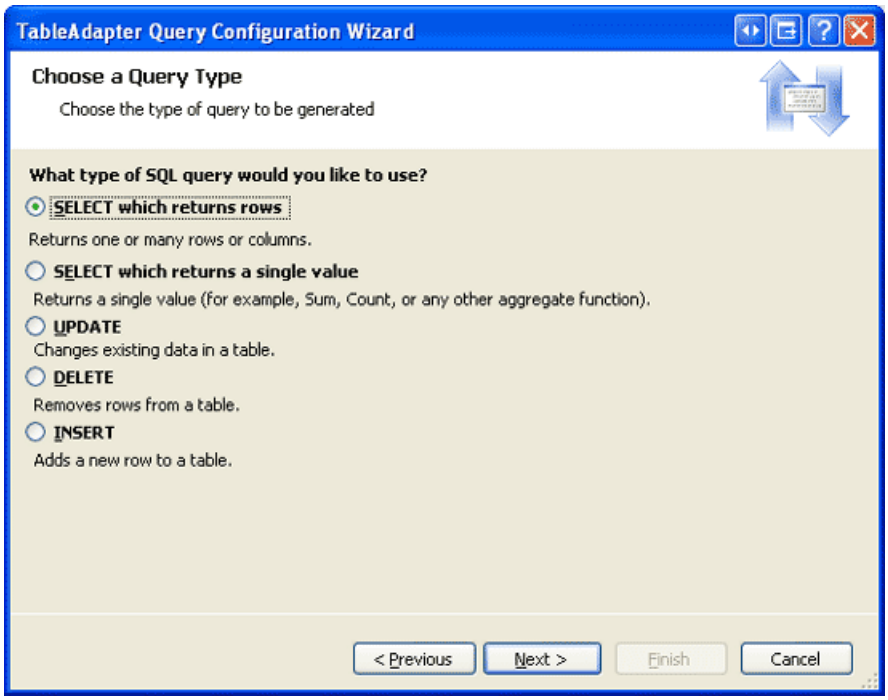


Figure 8: The SQL Statement Returns Rows (Click to view full-size image)

The next wizard screen prompts us for the query to use. To return each category s `CategoryID`, `CategoryName`, and `Description` fields, along with the number of products associated with the category, use the following `SELECT` statement:

```
SELECT CategoryID, CategoryName, Description,
       (SELECT COUNT(*) FROM Products p WHERE p.CategoryID = c.CategoryID)
            as NumberOfProducts
FROM Categories c
```

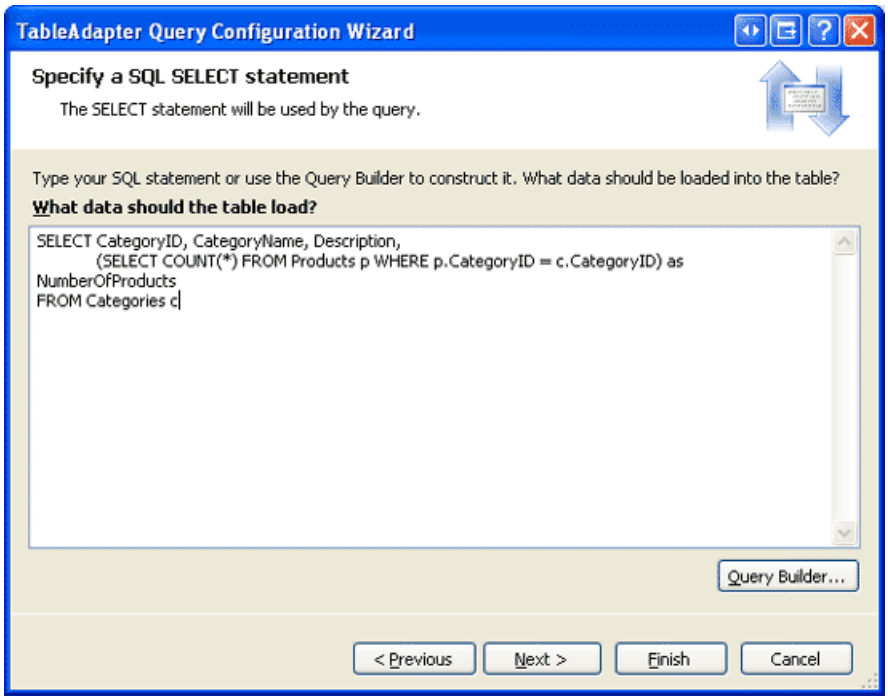


Figure 9: Specify the Query to Use (Click to view full-size image)

Note that the subquery that computes the number of products associated with the category is aliased as `NumberOfProducts`. This naming match causes the value returned by this subquery to be associated with the `CategoriesDataTable` s `NumberOfProducts` column.

After entering this query, the last step is to choose the name for the new method. Use `FillWithNumberOfProducts` and `GetCategoriesAndNumberOfProducts` for the Fill a DataTable and Return a DataTable patterns, respectively.

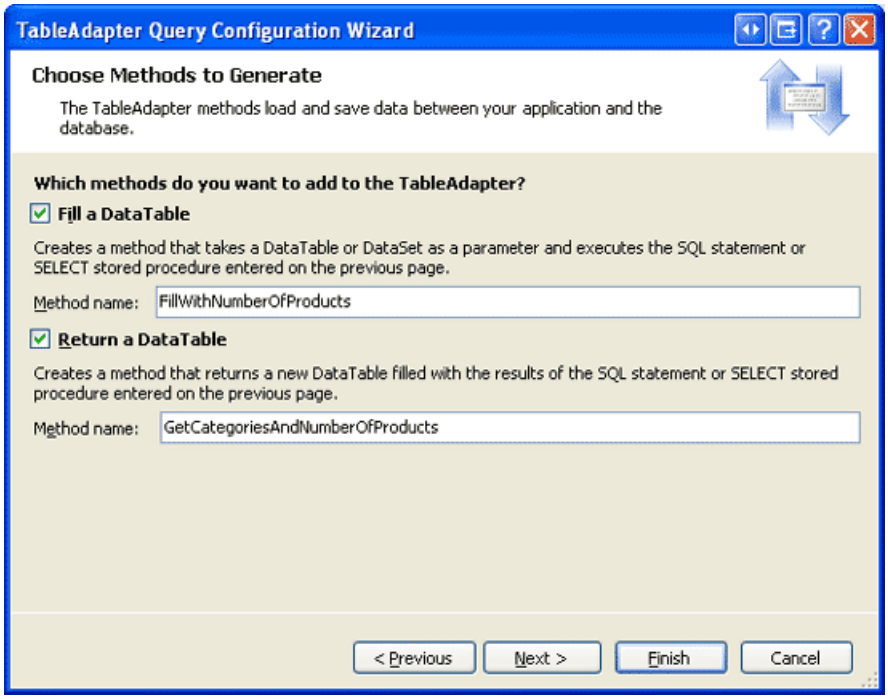


Figure 10: Name the New TableAdapter s Methods `FillWithNumberOfProducts` and `GetCategoriesAndNumberOfProducts` (Click to view full-size image)

At this point the Data Access Layer has been extended to include the number of products per category. Since all our presentation layer routes all calls to the DAL through a separate Business Logic Layer we need to add a corresponding `GetCategoriesAndNumberOfProducts` method to the `CategoriesBLL` class:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Select, false)]
public Northwind.CategoriesDataTable GetCategoriesAndNumberOfProducts()
{
    return Adapter.GetCategoriesAndNumberOfProducts();
}
```

With the DAL and BLL complete, we're ready to bind this data to the `Categories` Repeater in `CategoriesAndProducts.aspx`! If you've already created an ObjectDataSource for the Repeater from the Determining the Number of Products in the `ItemDataBound` Event Handler section, delete this ObjectDataSource and remove the Repeater's `DataSourceID` property setting; also unwire the Repeater's `ItemDataBound` event from the event handler by removing the `Handles Categories.OnItemDataBound` syntax in the ASP.NET code-behind class.

With the Repeater back in its original state, add a new ObjectDataSource named `CategoriesDataSource` via the Repeater's smart tag. Configure the ObjectDataSource to use the `CategoriesBLL` class, but instead of having it use the `GetCategories()` method, have it use `GetCategoriesAndNumberOfProducts()` instead (see Figure 11).

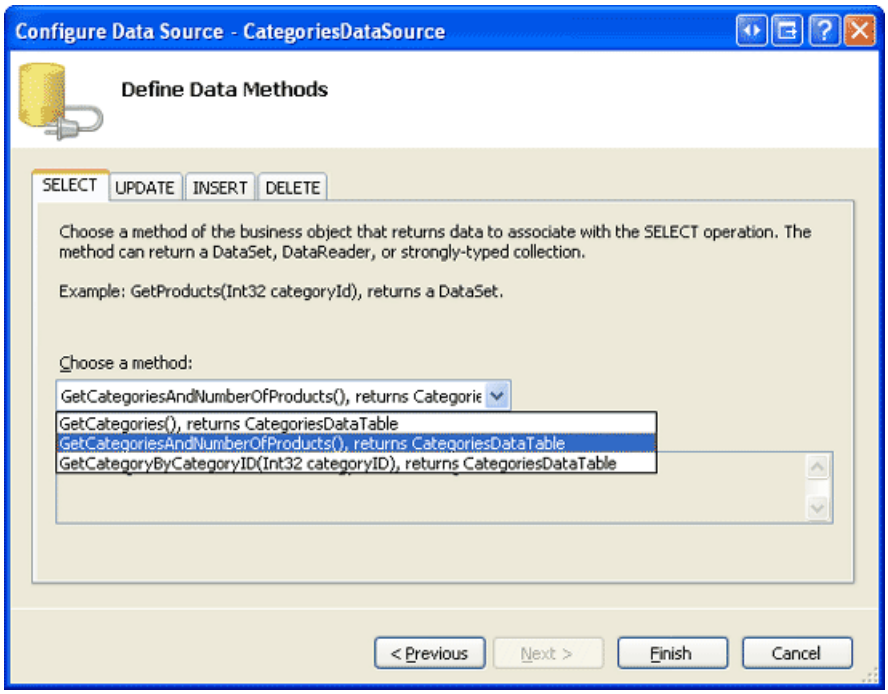


Figure 11: Configure the ObjectDataSource to Use the `GetCategoriesAndNumberOfProducts` Method (Click to view full-size image)

Next, update the `ItemTemplate` so that the LinkButton's `Text` property is declaratively assigned using databinding syntax and includes both the `CategoryName` and `NumberOfProducts` data fields. The complete declarative markup for the Repeater and the `CategoriesDataSource` ObjectDataSource follow:

```
<asp:Repeater ID="Categories" runat="server" DataSourceID="CategoriesDataSource">
    <HeaderTemplate>
        <ul>
    </HeaderTemplate>
    <ItemTemplate>
        <li><asp:LinkButton runat="server" ID="ViewCategory"
                Text='<%# String.Format("{0} ({1:N0})", _
                    Eval("CategoryName"), Eval("NumberOfProducts")) %>' />
        </li>
    </ItemTemplate>
    <FooterTemplate>
        </ul>
    </FooterTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategoriesAndNumberOfProducts" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

The output rendered by updating the DAL to include a `NumberOfProducts` column is the same as using the `ItemDataBound` event handler approach (refer back to Figure 4 to see a screen shot of the Repeater showing the category names and number of products).

Step 3: Displaying the Selected Category s Products

At this point we have the `Categories` Repeater displaying the list of categories along with the number of products in each category. The Repeater uses a LinkButton for each category that, when clicked, causes a postback, at which point we need to display those products for the selected category in the `CategoryProducts` DataList.

One challenge facing us is how to have the DataList display just those products for the selected category. In the Master/Detail Using a Selectable Master GridView with a Details DetailsView tutorial we saw how to build a GridView whose rows could be selected, with the selected row s details being displayed in a DetailsView on the same page. The GridView s ObjectDataSource returned information about all products using the `ProductsBLL` s `GetProducts()` method while the DetailsView s ObjectDataSource retrieved information about the selected product using the `GetProductsByProductID(productID)` method. The *`productID`* parameter value was provided declaratively by associating it with the value of the GridView s `SelectedValue` property. Unfortunately, the Repeater does not have a `SelectedValue` property and cannot serve as a parameter source.

> **NOTE**
> This is one of those challenges that appear when using the LinkButton in a Repeater. Had we used a hyperlink to pass in the `CategoryID` through the querystring instead, we could use that QueryString field as the source for the parameter s value.

Before we worry about the lack of a `SelectedValue` property for the Repeater, though, let s first bind the DataList to an ObjectDataSource and specify its `ItemTemplate`.

From the DataList s smart tag, opt to add a new ObjectDataSource named `CategoryProductsDataSource` and configure it to use the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method. Since the DataList in this tutorial offers a read-only interface, feel free to set the drop-down lists in the INSERT, UPDATE, and DELETE tabs to (None).

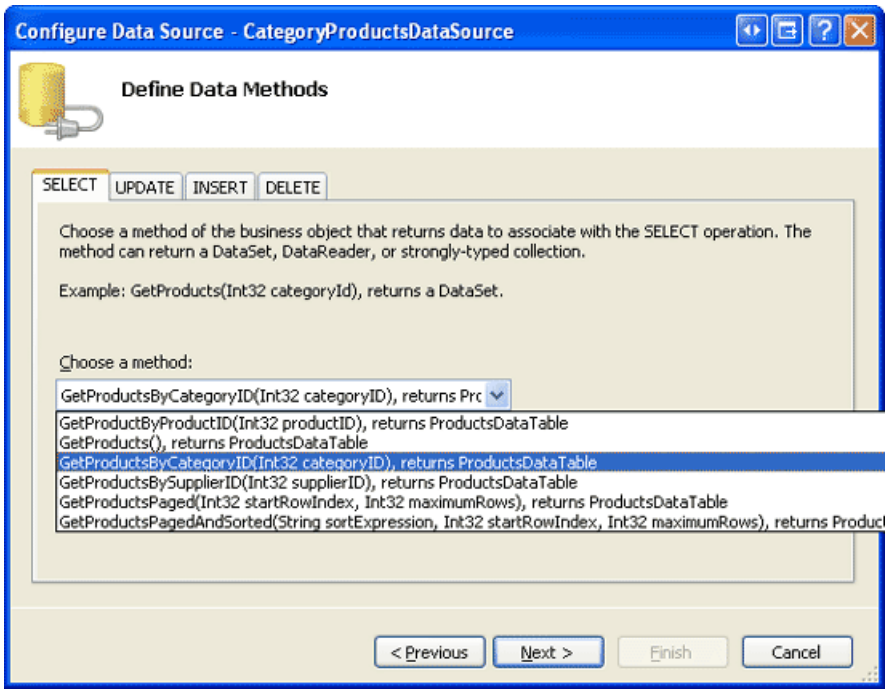


Figure 12: Configure the ObjectDataSource to Use `ProductsBLL` Class s `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

Since the `GetProductsByCategoryID(categoryID)` method expects an input parameter (*`categoryID`*), the Configure Data Source wizard allows us to specify the parameter s source. Had the categories been listed in a GridView or a DataList, we d set the Parameter source drop-down list to Control and the ControlID to the `ID` of the data Web control. However, since the Repeater lacks a `SelectedValue` property it cannot be used as a parameter source. If you check, you'll find that the ControlID drop-down list only contains one control `ID``CategoryProducts`, the `ID` of the DataList.

For now, set the Parameter source drop-down list to None. We'll end up programmatically assigning this parameter value when a category LinkButton is clicked in the Repeater.

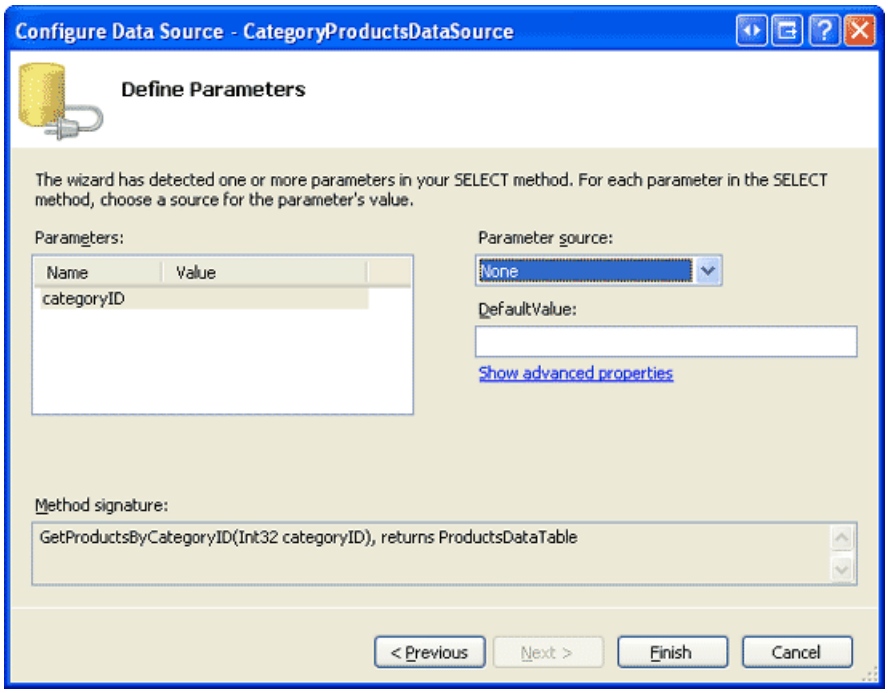


Figure 13: Do Not Specify a Parameter Source for the *`categoryID`* Parameter (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio auto-generates the DataList s `ItemTemplate`. Replace this default `ItemTemplate` with the template we used in the preceding tutorial; also, set the DataList s

`RepeatColumns` property to 2. After making these changes the declarative markup for your DataList and its associated ObjectDataSource should look like the following:

```
<asp:DataList ID="CategoryProducts" runat="server" DataKeyField="ProductID"
    DataSourceID="CategoryProductsDataSource" RepeatColumns="2"
    EnableViewState="False">
    <ItemTemplate>
        <h5><%# Eval("ProductName") %></h5>
        <p>
            Supplied by <%# Eval("SupplierName") %><br />
            <%# Eval("UnitPrice", "{0:C}") %>
        </p>
    </ItemTemplate>
</asp:DataList>
<asp:ObjectDataSource ID="CategoryProductsDataSource"
    OldValuesParameterFormatString="original_{0}"  runat="server"
    SelectMethod="GetProductsByCategoryID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:Parameter Name="categoryID" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

Currently, the `CategoryProductsDataSource` ObjectDataSource s *`categoryID`* parameter is never set, so no products are displayed when viewing the page. What we need to do is have this parameter value set based on the `CategoryID` of the clicked category in the Repeater. This introduces two challenges: first, how do we determine when a LinkButton in the Repeater s `ItemTemplate` has been clicked; and second, how can we determine the `CategoryID` of the corresponding category whose LinkButton was clicked?

The LinkButton like the Button and ImageButton controls has a `Click` event and a `Command` event. The `Click` event is designed to simply note that the LinkButton has been clicked. At times, however, in addition to noting that the LinkButton has been clicked we also need to pass some extra information to the event handler. If this is the case, the LinkButton s `CommandName` and `CommandArgument` properties can be assigned this extra information. Then, when the LinkButton is clicked, its `Command` event fires (instead of its `Click` event) and the event handler is passed the values of the `CommandName` and `CommandArgument` properties.

When a `Command` event is raised from within a template in the Repeater, the Repeater s `ItemCommand` event fires and is passed the `CommandName` and `CommandArgument` values of the clicked LinkButton (or Button or ImageButton). Therefore, to determine when a category LinkButton in the Repeater has been clicked, we need to do the following:

1. Set the `CommandName` property of the LinkButton in the Repeater s `ItemTemplate` to some value (I ve used ListProducts). By setting this `CommandName` value, the LinkButton s `Command` event fires when the LinkButton is clicked.
2. Set the LinkButton s `CommandArgument` property to the value of the current item s `CategoryID`.
3. Create an event handler for the Repeater s `ItemCommand` event. In the event handler, set the `CategoryProductsDataSource` ObjectDataSource s `CategoryID` parameter to the value of the passed-in `CommandArgument`.

The following `ItemTemplate` markup for the Categories Repeater implements steps 1 and 2. Note how the `CommandArgument` value is assigned the data item s `CategoryID` using databinding syntax:

```
<ItemTemplate>
    <li>
        <asp:LinkButton CommandName="ListProducts"  runat="server"
            CommandArgument='<%# Eval("CategoryID") %>' ID="ViewCategory"
            Text='<%# string.Format("{0} ({1:N0})", _
                Eval("CategoryName"), Eval("NumberOfProducts")) %>'>
        </asp:LinkButton>
    </li>
</ItemTemplate>
```

Whenever creating an `ItemCommand` event handler it is prudent to always first check the incoming `CommandName` value because *any* `Command` event raised by *any* Button, LinkButton, or ImageButton within the Repeater will cause the `ItemCommand` event to fire. While we currently only have one such LinkButton now, in the future we (or another developer on our team) might add additional button Web controls to the Repeater that, when clicked, raises the same `ItemCommand` event handler. Therefore, it s best to always make sure you check the `CommandName` property and only proceed with your programmatic logic if it matches up to the value expected.

After ensuring that the passed-in `CommandName` value equals ListProducts, the event handler then assigns the `CategoryProductsDataSource` ObjectDataSource s `CategoryID` parameter to the value of the passed-in `CommandArgument`. This modification to the ObjectDataSource s `SelectParameters` automatically causes the DataList to rebind itself to the data source, showing the products for the newly selected category.

```
protected void Categories_ItemCommand(object source, RepeaterCommandEventArgs e)
{
    // If it's the "ListProducts" command that has been issued...
    if (string.Compare(e.CommandName, "ListProducts", true) == 0)
    {
        // Set the CategoryProductsDataSource ObjectDataSource's CategoryID parameter
        // to the CategoryID of the category that was just clicked (e.CommandArgument)...
        CategoryProductsDataSource.SelectParameters["CategoryID"].DefaultValue =
            e.CommandArgument.ToString();
    }
}
```

With these additions, our tutorial is complete! Take a moment to test it out in a browser. Figure 14 shows the screen when first visiting the page. Since a category has yet to be selected, no products are displayed. Clicking on a category, such as Produce, displays those products in the Product category in a two-column view (see Figure 15).

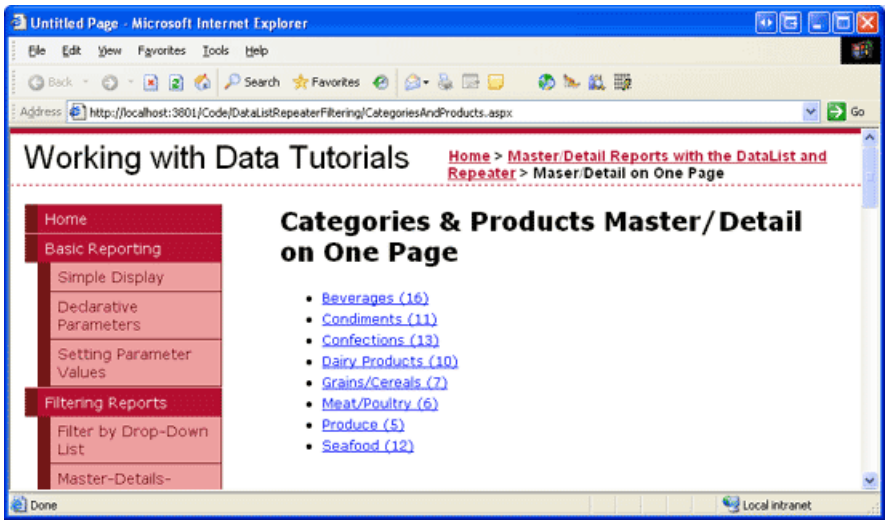


Figure 14: No Products are Displayed When First Visiting the Page (Click to view full-size image)

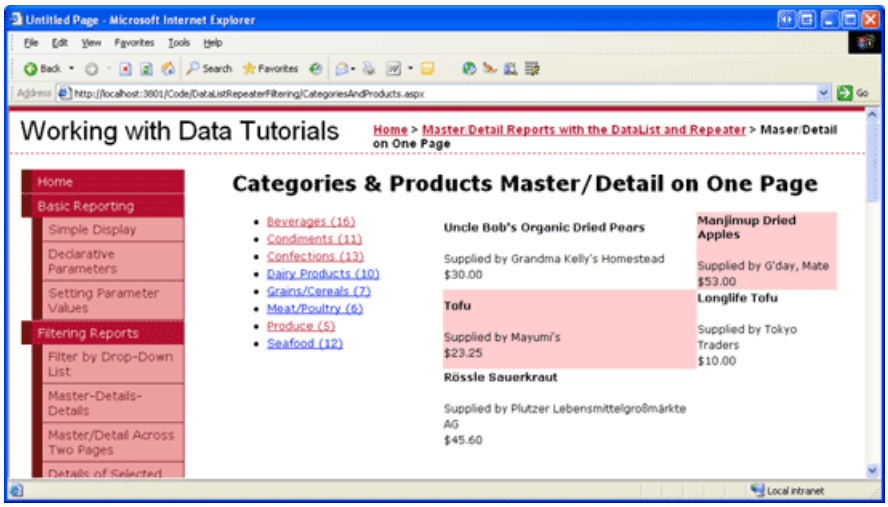


Figure 15: Clicking the Produce Category Lists the Matching Products to the Right (Click to view full-size image)

Summary

As we saw in this tutorial and the preceding one, master/detail reports can be spread out across two pages, or consolidated on one. Displaying a master/details report on a single page, however, introduces some challenges on how best to layout the master and details records on the page. In the *Master/Detail Using a Selectable Master GridView with a Details DetailsView* tutorial we had the details records appear above the master records; in this tutorial we used CSS techniques to have the master records float to the left of the details.

Along with displaying master/details reports, we also had the opportunity to explore how to retrieve the number of products associated with each category as well as how to perform server-side logic when a LinkButton (or Button or ImageButton) is clicked from within a Repeater.

This tutorial completes our examination of master/detail reports with the DataList and Repeater. Our next set of tutorials will illustrate how to add editing and deleting capabilities to the DataList control.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Floatutorial a tutorial on floating CSS elements with CSS
- CSS Positioning more information on positioning elements with CSS
- Laying Out Content with HTML using `<table>` s and other HTML elements for positioning

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Zack Jones. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Master/Detail Filtering With a DropDownList (VB)

2/20/2020 • 9 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we see how to display master/detail reports in a single web page using DropDownLists to display the "master" records and a DataList to display the "details".

Introduction

The master/detail report, which we first created using a GridView in the earlier Master/Detail Filtering With a DropDownList tutorial, begins by showing some set of "master" records. The user can then drill down into one of the master records, thereby viewing that master record's "details." Master/detail reports are an ideal choice for visualizing one-to-many relationships and for displaying detailed information from particularly "wide" tables (ones that have a lot of columns). We've explored how to implement master/detail reports using the GridView and DetailsView controls in previous tutorials. In this tutorial and the next two, we'll reexamine these concepts, but focus on using DataList and Repeater controls instead.

In this tutorial, we'll look at using a DropDownList to contain the "master" records, with the "details" records displayed in a DataList.

Step 1: Adding the Master/Detail Tutorial Web Pages

Before we start this tutorial, let's first take a moment to add the folder and ASP.NET pages we'll need for this tutorial and the next two dealing with master/detail reports using the DataList and Repeater controls. Start by creating a new folder in the project named `DataListRepeaterFiltering`. Next, add the following five ASP.NET pages to this folder, having all of them configured to use the master page `Site.master`:

- `Default.aspx`
- `FilterByDropDownList.aspx`
- `CategoryListMaster.aspx`
- `ProductsForCategoryDetails.aspx`
- `CategoriesAndProducts.aspx`

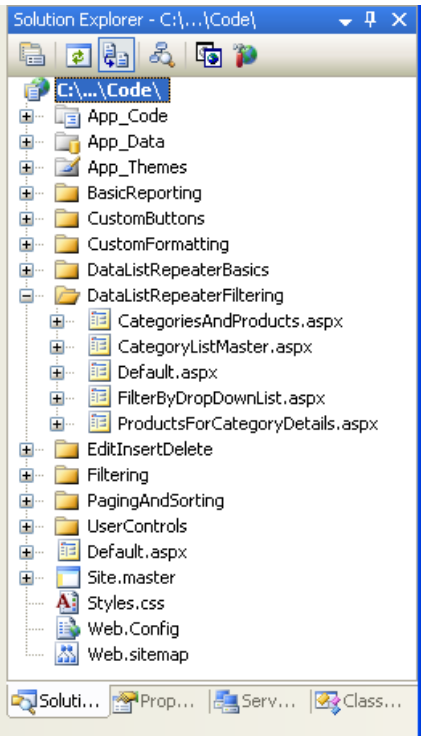


Figure 1: Create a `DataListRepeaterFiltering` Folder and Add the Tutorial ASP.NET Pages

Next, open the `Default.aspx` page and drag the `SectionLevelTutorialListing.ascx` User Control from the `UserControls` folder onto the Design surface. This User Control, which we created in the Master Pages and Site Navigation tutorial, enumerates the site map and displays the tutorials from the current section in a bulleted list.

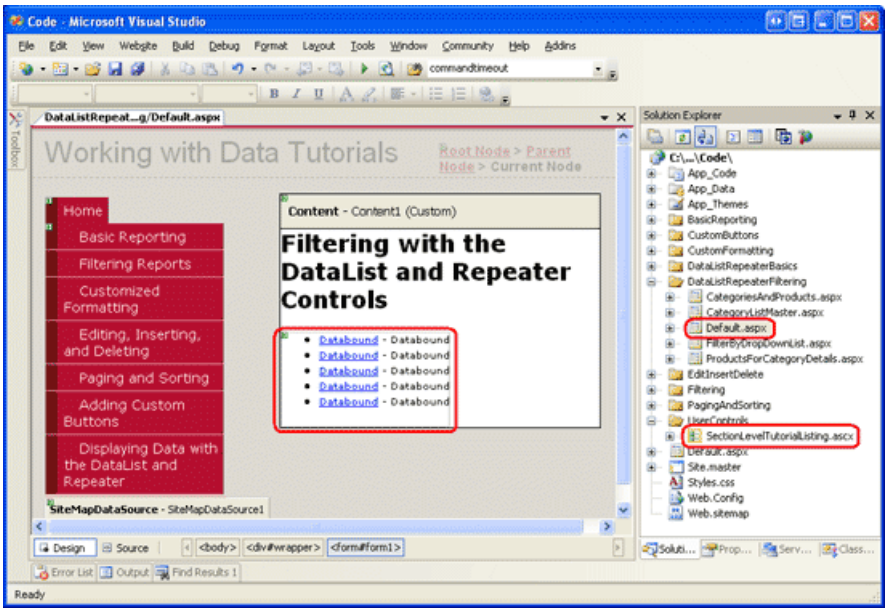


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

In order to have the bulleted list display the master/detail tutorials we'll be creating, we need to add them to the site map. Open the `Web.sitemap` file and add the following markup after the "Displaying Data with the DataList and Repeater" site map node markup:

```
<siteMapNode
    title="Master/Detail Reports with the DataList and Repeater"
    description="Samples of Reports that Use the DataList and Repeater Controls"
    url="~/DataListRepeaterFiltering/Default.aspx">

    <siteMapNode
        title="Filter by Drop-Down List"
        description="Filter results using a drop-down list."
        url="~/DataListRepeaterFiltering/FilterByDropDownList.aspx" />

    <siteMapNode
        title="Master/Detail Across Two Pages"
        description="Master records on one page, detail records on another."
        url="~/DataListRepeaterFiltering/CategoryListMaster.aspx" />

    <siteMapNode
        title="Maser/Detail on One Page"
        description="Master records in the left column, details on the right,
                     both on the same page."
        url="~/DataListRepeaterFiltering/CategoriesAndProducts.aspx" />

</siteMapNode>
```

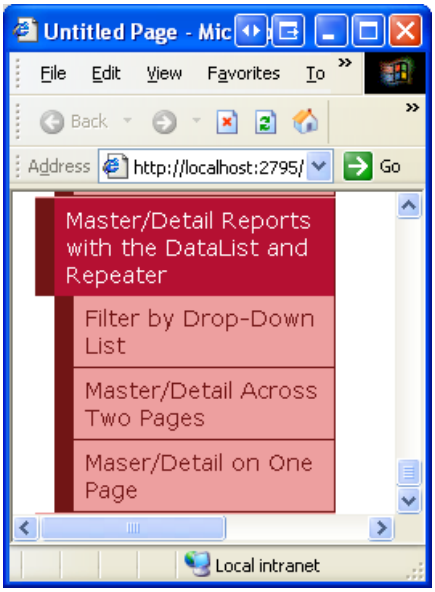


Figure 3: Update the Site Map to Include the New ASP.NET Pages

Step 2: Displaying the Categories in a DropDownList

Our master/detail report will list the categories in a DropDownList, with the selected list item's products displayed further down in the page in a DataList. The first task ahead of us, then, is to have the categories displayed in a DropDownList. Start by opening the `FilterByDropDownList.aspx` page in the `DataListRepeaterFiltering` folder and drag a DropDownList from the Toolbox onto the page's designer. Next, set the DropDownList's `ID` property to `Categories`. Click on the Choose Data Source link from the DropDownList's smart tag and create a new ObjectDataSource named `CategoriesDataSource`.

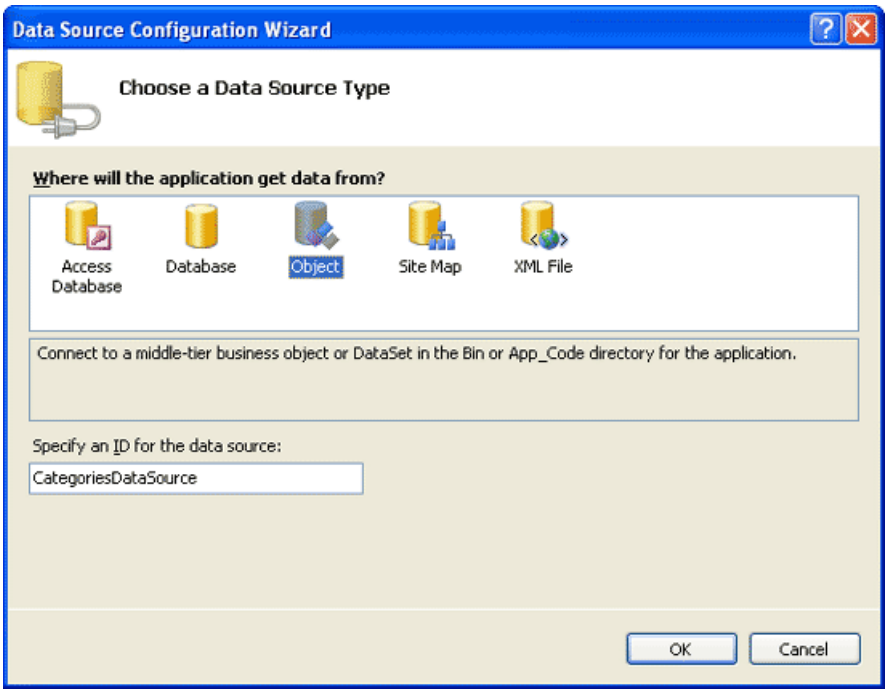


Figure 4: Add a New ObjectDataSource Named `CategoriesDataSource` (Click to view full-size image)

Configure the new ObjectDataSource such that it invokes the `CategoriesBLL` class's `GetCategories()` method. After configuring the ObjectDataSource we still need to specify what data source field should be displayed in the DropDownList and which one should be associated as the value for each list item. Have the `CategoryName` field as the display and `CategoryID` as the value for each list item.

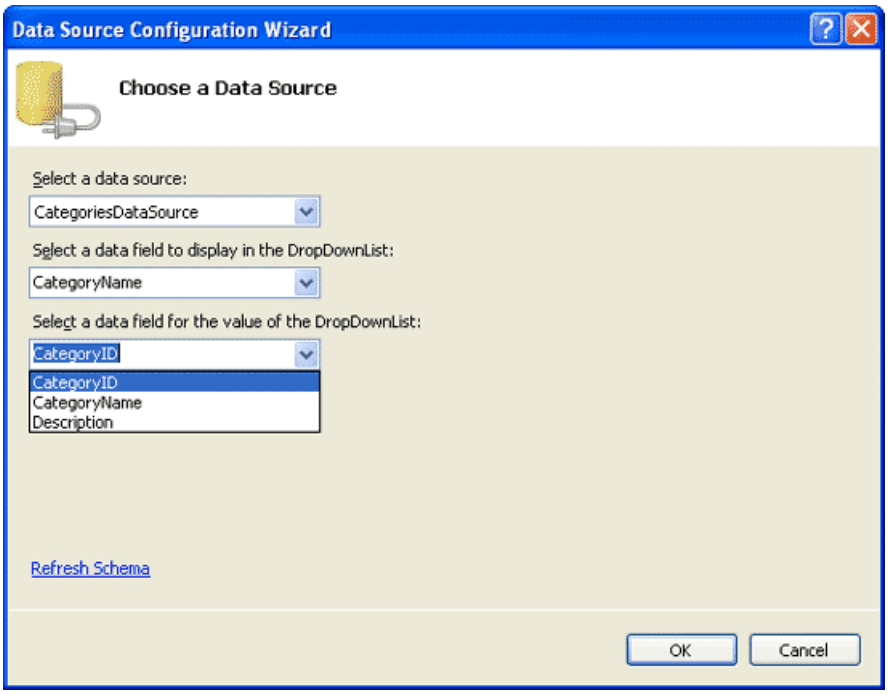


Figure 5: Have the DropDownList Display the `CategoryName` Field and Use `CategoryID` as the Value (Click to view full-size image)

At this point we have a DropDownList control that's populated with the records from the `Categories` table (all accomplished in about six seconds). Figure 6 shows our progress thus far when viewed through a browser.

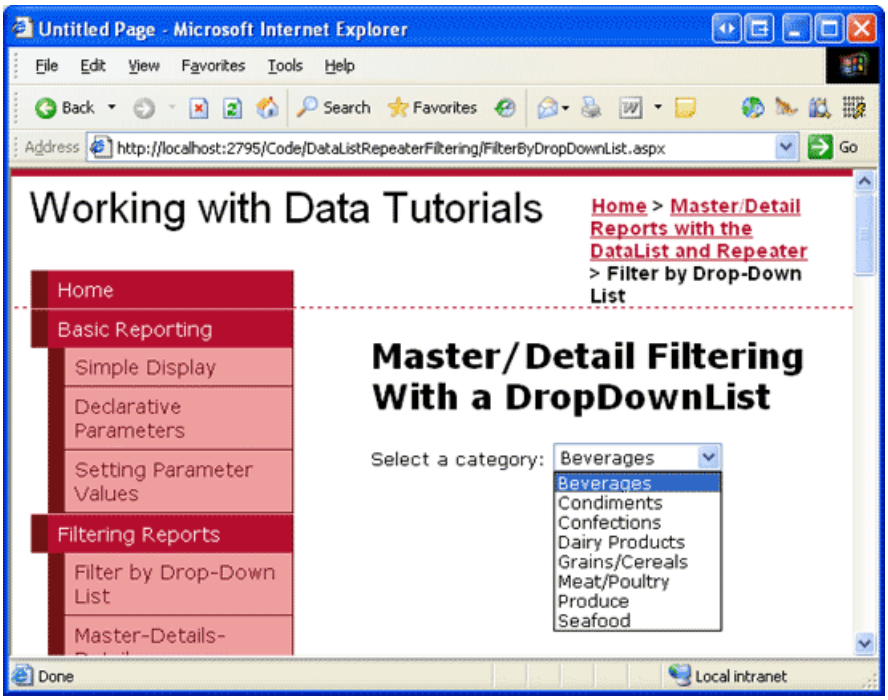


Figure 6: A Drop-Down Lists the Current Categories (Click to view full-size image)

Step 2: Adding the Products DataList

The last step in our master/detail report is to list the products associated with the selected category. To accomplish this, add a DataList to the page and create a new ObjectDataSource named `ProductsByCategoryDataSource`. Have the `ProductsByCategoryDataSource` control retrieve its data from the `ProductsBLL` class's `GetProductsByCategoryID(categoryID)` method. Since this master/detail report is read-only, choose the (None) option in the INSERT, UPDATE, and DELETE tabs.

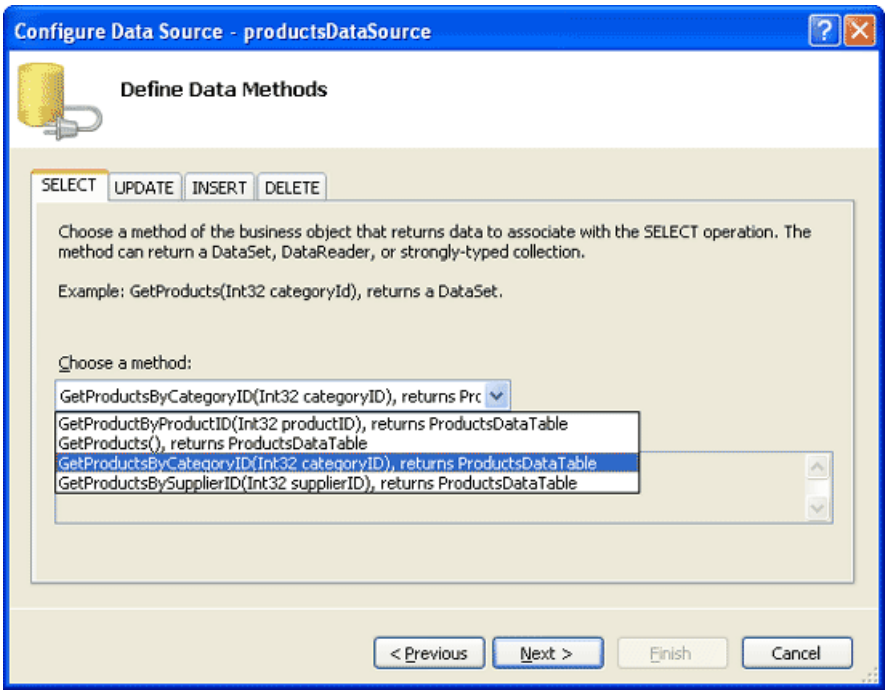


Figure 7: Select the `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

After clicking Next, the ObjectDataSource wizard prompts us for the source of the value for the `GetProductsByCategoryID(categoryID)` method's *`categoryID`* parameter. To use the value of the selected `categories` DropDownList item set the Parameter source to Control and the ControlID to `Categories`.

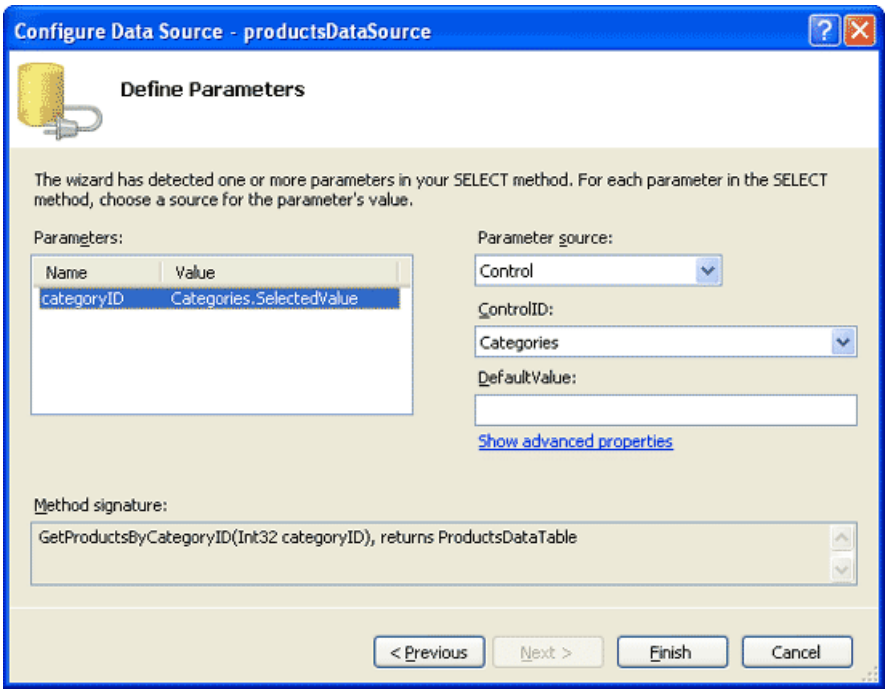


Figure 8: Set the `categoryID` Parameter to the Value of the `Categories` DropDownList (Click to view full-size image)

Upon completing the Configure Data Source wizard, Visual Studio will automatically generate an `ItemTemplate` for the DataList that displays the name and value of each data field. Let's enhance the DataList to instead use an `ItemTemplate` that displays just the product's name, category, supplier, quantity per unit, and price along with a `SeparatorTemplate` that injects an `<hr>` element between each item. I'm going to use the `ItemTemplate` from an example in the Displaying Data with the DataList and Repeater Controls tutorial, but feel free to use whatever template markup you find most visually appealing.

After making these changes, your DataList and its ObjectDataSource's markup should look similar to the following:

```
<asp:DataList ID="DataList1" runat="server" DataKeyField="ProductID"
    DataSourceID="ProductsByCategoryDataSource" EnableViewState="False">
    <ItemTemplate>
        <h4>
            <asp:Label ID="ProductNameLabel" runat="server"
                Text='<%# Eval("ProductName") %>' />
        </h4>
        <table border="0">
            <tr>
                <td class="ProductPropertyLabel">Category:</td>
                <td><asp:Label ID="CategoryNameLabel" runat="server"
                    Text='<%# Eval("CategoryName") %>' /></td>
                <td class="ProductPropertyLabel">Supplier:</td>
                <td><asp:Label ID="SupplierNameLabel" runat="server"
                    Text='<%# Eval("SupplierName") %>' /></td>
            </tr>
            <tr>
                <td class="ProductPropertyLabel">Qty/Unit:</td>
                <td><asp:Label ID="QuantityPerUnitLabel" runat="server"
                    Text='<%# Eval("QuantityPerUnit") %>' /></td>
                <td class="ProductPropertyLabel">Price:</td>
                <td><asp:Label ID="UnitPriceLabel" runat="server"
                    Text='<%# Eval("UnitPrice", "{0:C}") %>' /></td>
            </tr>
        </table>
    </ItemTemplate>
    <SeparatorTemplate>
        <hr />
    </SeparatorTemplate>
</asp:DataList>

<asp:ObjectDataSource ID="ProductsByCategoryDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProductsByCategoryID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:ControlParameter ControlID="Categories" Name="categoryID"
            PropertyName="SelectedValue" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

Take a moment to check out our progress in a browser. When first visiting the page, those products belonging to the selected category (Beverages) are displayed (as shown in Figure 9), but changing the DropDownList doesn't update the data. This is because a postback must occur for the DataList to update. To accomplish this we can either set the DropDownList's `AutoPostBack` property to `true` or add a Button Web control to the page. For this tutorial, I've opted to set the DropDownList's `AutoPostBack` property to `true`.

Figures 9 and 10 illustrate the master/detail report in action.

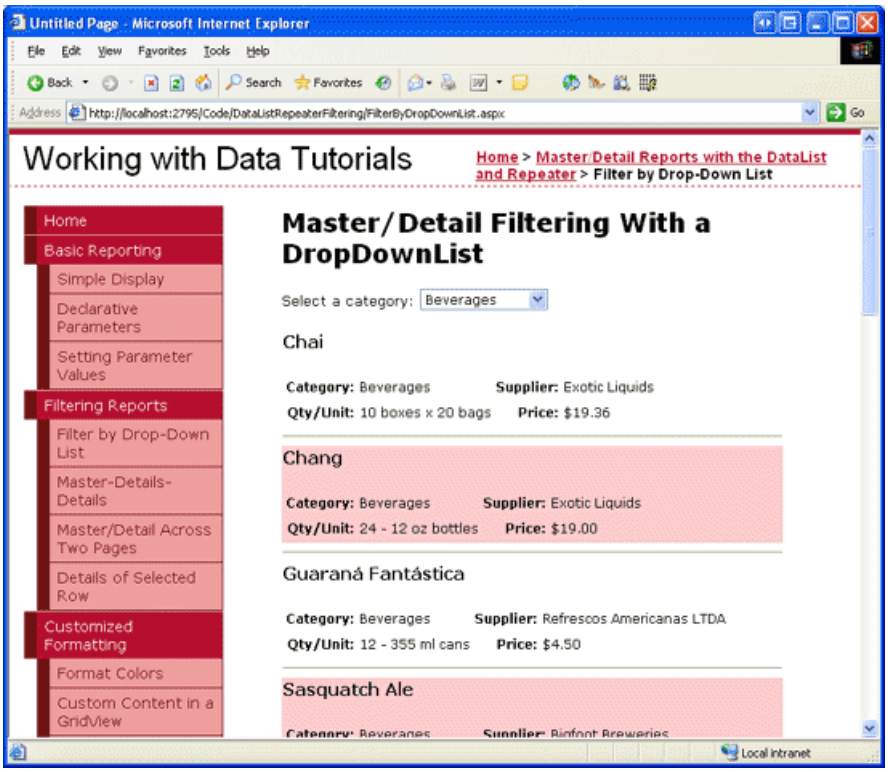


Figure 9: When First Visiting the Page, the Beverage Products are Displayed (Click to view full-size image)

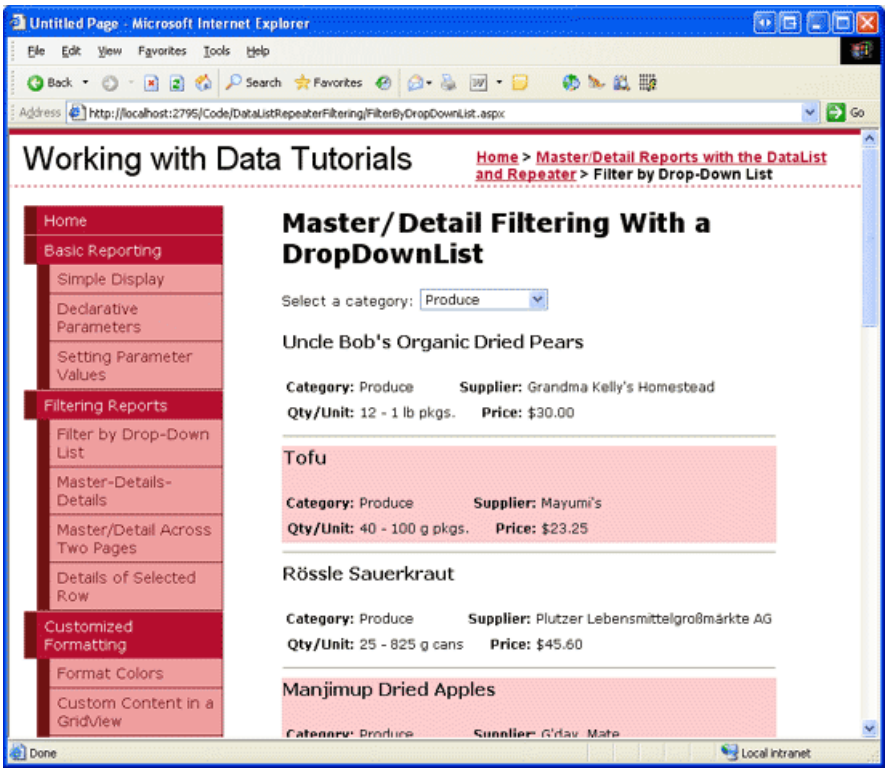


Figure 10: Selecting a New Product (Produce) Automatically Causes a PostBack, Updating the DataList (Click to view full-size image)

Adding a "-- Choose a Category --" List Item

When first visiting the `FilterByDropDownList.aspx` page the categories DropDownList's first list item (Beverages) is selected by default, showing the beverage products in the DataList. In the *Master/Detail Filtering With a DropDownList* tutorial we added a "-- Choose a Category --" option to the DropDownList that was selected by default and, when selected, displayed *all* of the products in the database. Such an approach was manageable when listing the products in a GridView, as each product row took up a small amount of screen real estate. With the DataList, however, each product's information consumes a much larger chunk of the screen. Let's still add a "--

Choose a Category --" option and have it selected by default, but instead of having it show all products when selected, let's configure it so that it shows no products.

To add a new list item to the DropDownList, go to the Properties window and click on the ellipses in the `Items` property. Add a new list item with the `Text` "-- Choose a Category --" and the `Value` `0`.

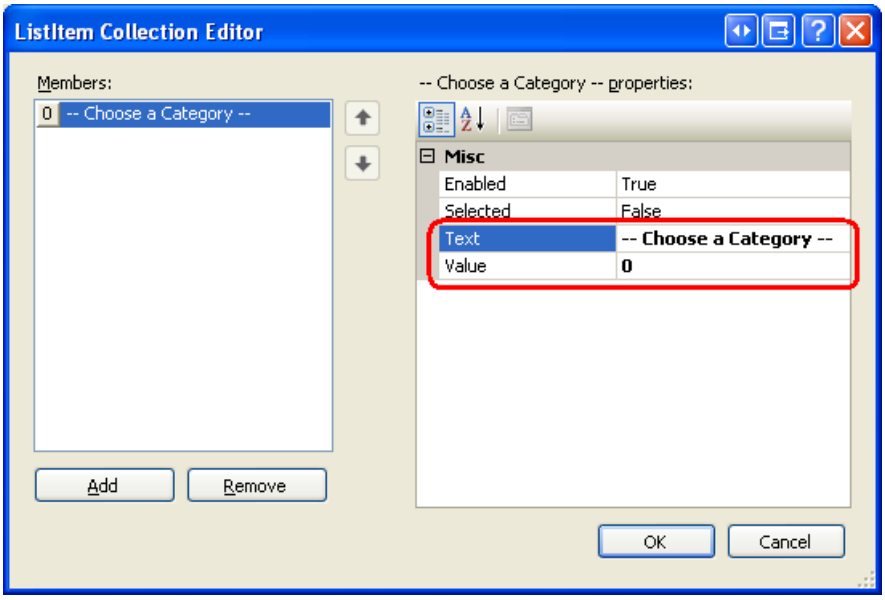


Figure 11: Add a "-- Choose a Category --" List Item

Alternatively, you can add the list item by adding the following markup to the DropDownList:

```
<asp:DropDownList ID="categories" runat="server" AutoPostBack="True"
    DataSourceID="CategoriesDataSource" DataTextField="CategoryName"
    DataValueField="CategoryID" EnableViewState="False">

    <asp:ListItem Value="0">-- Choose a Category --</asp:ListItem>

</asp:DropDownList>
```

Additionally, we need to set the DropDownList control's `AppendDataBoundItems` to `true` because if it's set to `false` (the default), when the categories are bound to the DropDownList from the ObjectDataSource they'll overwrite any manually-added list items.

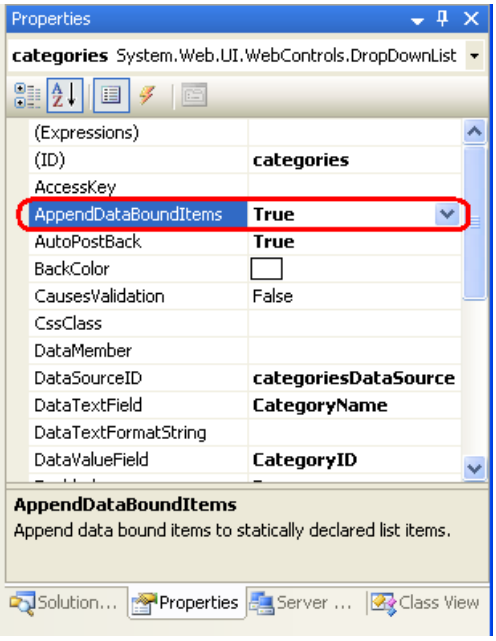


Figure 12: Set the `AppendDataBoundItems` Property to True

The reason we chose the value `0` for the "-- Choose a Category --" list item is because there are no categories in the system with a value of `0`, hence no product records will be returned when the "-- Choose a Category --" list item is selected. To confirm this, take a moment to visit the page through a browser. As Figure 13 shows, when initially viewing the page the "-- Choose a Category --" list item is selected and no products are displayed.

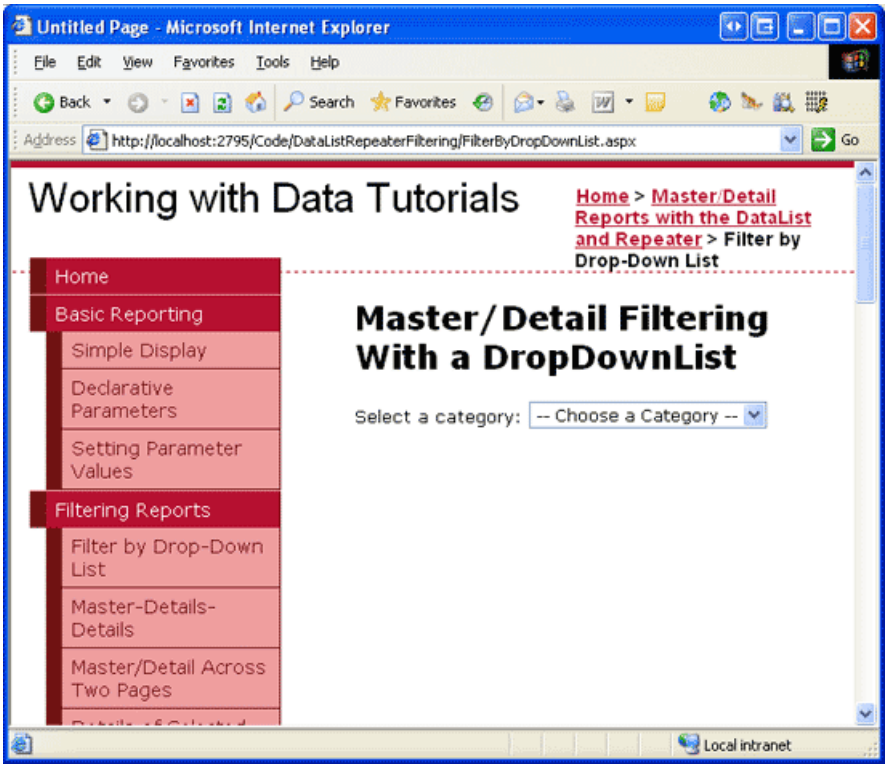


Figure 13: When the "-- Choose a Category --" List Item is Selected, No Products are Displayed (Click to view full-size image)

If you'd rather display *all* of the products when the "-- Choose a Category --" option is selected, use a value of `-1` instead. The astute reader will recall that back in the *Master/Detail Filtering With a DropDownList* tutorial we updated the `ProductsBLL` class's `GetProductsByCategoryID(categoryID)` method so that if a `categoryID` value of `-1` was passed in, all product records were returned.

Summary

When displaying hierarchically-related data, it often helps to present the data using master/detail reports, from which the user can start perusing the data from the top of the hierarchy and drill down into details. In this tutorial we examined building a simple master/detail report showing a selected category's products. This was accomplished by using a DropDownList for the list of categories and a DataList for the products belonging to the selected category.

In the next tutorial we'll look at separating the master and details records across two pages. In the first page, a list of "master" records will be displayed, with a link to view the details. Clicking on the link will whisk the user to the second page, which will display the details for the selected master record.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via

his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To…

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Randy Schmidt. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Master/Detail Filtering Across Two Pages (VB)

2/20/2020 • 13 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we look at how to separate a master/detail report across two pages. In the "master" page we use a Repeater control to render a list of categories that, when clicked, will take the user to the "details" page where a two-column DataList shows those products belonging to the selected category.

Introduction

In the Master/Detail Filtering Across Two Pages tutorial, we examined this pattern using a GridView to display all of the suppliers in the system. This GridView included a HyperLinkField, which rendered as a link to a second page, passing along the `SupplierID` in the querystring. The second page used a GridView to list those products provided by the selected supplier.

Such two-page master/detail reports can be accomplished using DataList and Repeater controls as well. The only difference is that neither the DataList nor the Repeater provides support for the HyperLinkField control. Instead, we must add a HyperLink Web control or an anchor HTML element (`<a>`) within the control's `ItemTemplate`. The HyperLink's `NavigateUrl` property or the anchor's `href` attribute can then be customized using declarative or programmatic approaches.

In this tutorial we'll explore an example that lists the categories in a bulleted list on one page using a Repeater control. Each list item will include the category's name and description, with the category name displayed as a link to a second page. Clicking on this link will whisk the user to the second page, where a DataList will show those products that belong to the selected category.

Step 1: Displaying the Categories in a Bulleted List

The first step in creating any master/detail report is to start by displaying the "master" records. Therefore, our first task is to display the categories in the "master" page. Open the `CategoryListMaster.aspx` page in the `DataListRepeaterFiltering` folder, add a Repeater control, and, from the smart tag, opt to add a new ObjectDataSource. Configure the new ObjectDataSource so that it accesses its data from the `CategoriesBLL` class's `GetCategories` method (see Figure 1).

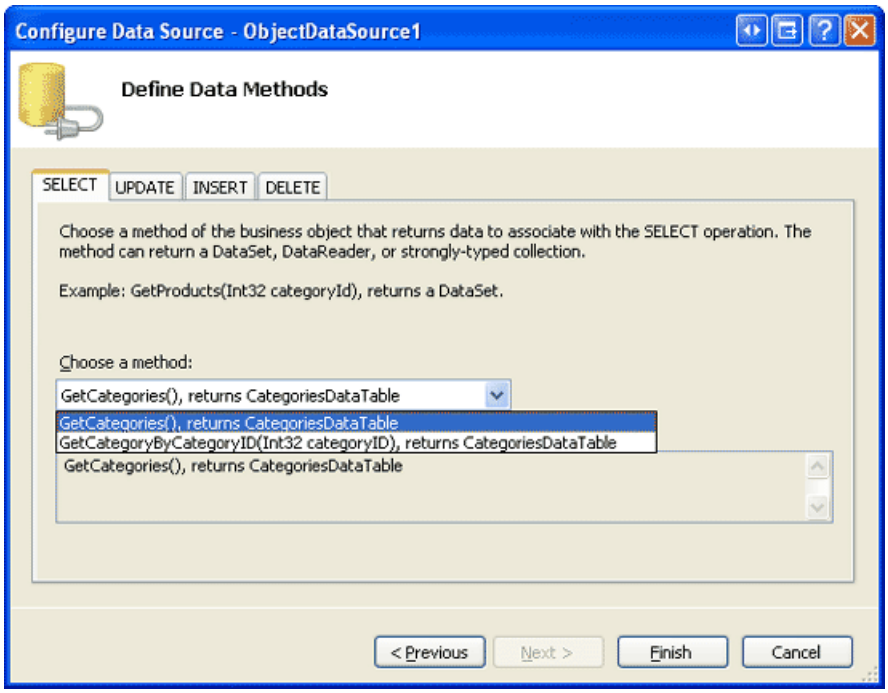


Figure 1: Configure the ObjectDataSource to Use the `CategoriesBLL` Class's `GetCategories` Method (Click to view full-size image)

Next, define the Repeater's templates such that it displays each category name and description as an item in a bulleted list. Let's not yet worry about having each category link to the details page. The following shows the declarative markup for the Repeater and ObjectDataSource:

```
<asp:Repeater ID="Repeater1" runat="server" DataSourceID="ObjectDataSource1"
    EnableViewState="False">
    <HeaderTemplate>
        <ul>
    </HeaderTemplate>

    <ItemTemplate>
        <li><%# Eval("CategoryName") %> - <%# Eval("Description") %></li>
    </ItemTemplate>

    <FooterTemplate>
        </ul>
    </FooterTemplate>
</asp:Repeater>

<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

With this markup complete, take a moment to view our progress through a browser. As Figure 2 shows, the Repeater renders as a bulleted list showing each category's name and description.

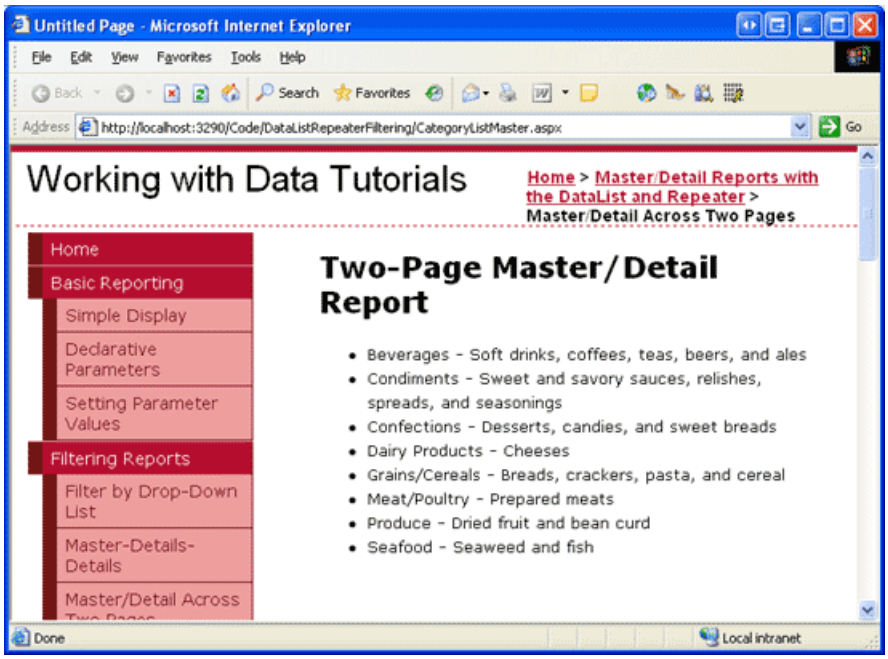


Figure 2: Each Category is Displayed as a Bulleted List Item (Click to view full-size image)

Step 2: Turning the Category Name into a Link to the Details Page

To allow a user to display the "details" information for a given category, we need to add a link to each bulleted list item that, when clicked, will take the user to the second page (`ProductsForCategoryDetails.aspx`). This second page will then display the products for the selected category using a DataList. In order to determine the category whose link was clicked, we need to pass the clicked category's `CategoryID` to the second page through some mechanism. The simplest, most straightforward way to transfer scalar data from one page to another is through the querystring, which is the option we'll use in this tutorial. In particular, the `ProductsForCategoryDetails.aspx` page will expect the selected *`categoryID`* value to be passed through a querystring field named `CategoryID`. For example, to view the products for the Beverages category, which has a `CategoryID` of 1, a user would visit `ProductsForCategoryDetails.aspx?CategoryID=1`.

To create a hyperlink for each bulleted list item in the Repeater we need to either add a HyperLink Web control or an HTML anchor element (`<a>`) to the `ItemTemplate`. In scenarios where the hyperlink is displayed the same for each row, either approach will suffice. For Repeaters, I prefer using the anchor element. To use the anchor element, update the Repeater's ItemTemplate to:

```
<li>
    <a href='ProductsForCategoryDetails.aspx?CategoryID=<%# Eval("CategoryID") %>'>
        <%# Eval("CategoryName") %>
    </a> - <%# Eval("Description") %>
</li>
```

Note that the `CategoryID` can be injected directly within the anchor element's `href` attribute; however, to do so be certain to delimit the `href` attribute's value with apostrophes (and note quotation marks) since the `Eval` method within the `href` attribute delimits its string (`"CategoryID"`) with quotation marks. Alternatively, a HyperLink Web control can be used instead:

```
<li>
    <asp:HyperLink runat="server" Text='<%# Eval("CategoryName") %>'
        NavigateUrl='<%# "ProductsForCategoryDetails.aspx?CategoryID=" &
            Eval("CategoryID") %>'>
    </asp:HyperLink>
    - <%# Eval("Description") %>
</li>
```

Note how the static portion of the URL — `ProductsForCategoryDetails.aspx?CategoryID` — is appended to the result of `Eval("CategoryID")` directly within the databinding syntax using string concatenation.

One benefit of using the HyperLink control is that it can be programmatically accessed from the Repeater's `ItemDataBound` event handler, if needed. For example, you might want to display the category name as text rather than as a link for categories with no associated products. Such a check could be programmatically performed in the `ItemDataBound` event handler; for categories with no associated products, the HyperLink's `NavigateUrl` property could be set to a blank string, thereby resulting in that particular category name rendering as plain text (rather than as a link). Refer back to the Formatting the DataList and Repeater Based Upon Data tutorial for more information on formatting the DataList and Repeater's contents based on programmatic logic through the `ItemDataBound` event handler.

If you are following along, feel free to use either the anchor element or HyperLink control approach in your page. Regardless of the approach, when viewing the page through a browser each category name should be rendered as a link to `ProductsForCategoryDetails.aspx`, passing in the applicable `CategoryID` value (see Figure 3).

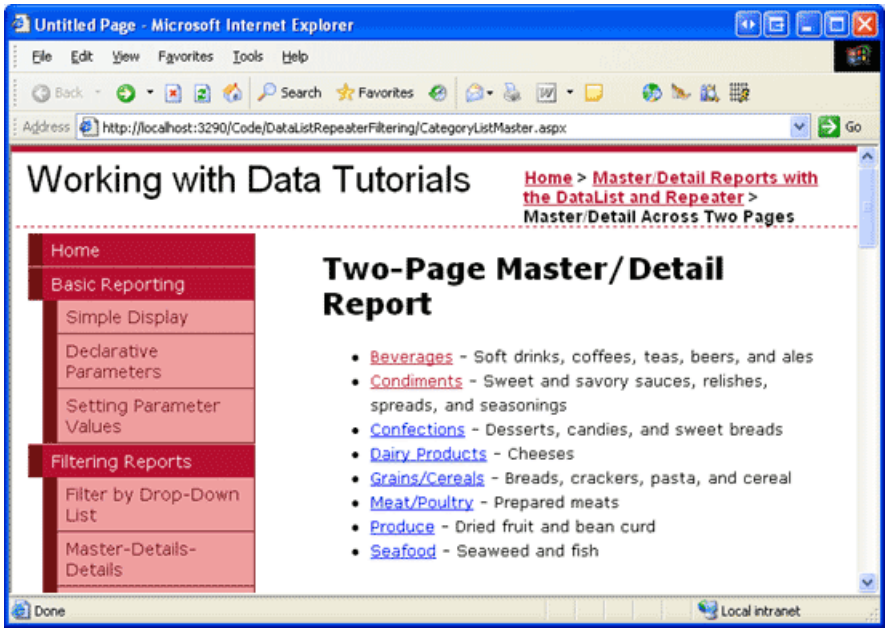


Figure 3: The Category Names Now Link to `ProductsForCategoryDetails.aspx` (Click to view full-size image)

Step 3: Listing the Products that Belong to the Selected Category

With the `CategoryListMaster.aspx` page complete, we're ready to turn our attention to implementing the "details" page, `ProductsForCategoryDetails.aspx`. Open this page, drag a DataList from the Toolbox onto the Designer, and set its `ID` property to `ProductsInCategory`. Next, from the DataList's smart tag choose to add a new ObjectDataSource to the page, naming it `ProductsInCategoryDataSource`. Configure it such that it calls the `ProductsBLL` class's `GetProductsByCategoryID(categoryID)` method; set the drop-down lists in the INSERT, UPDATE, and DELETE tabs to (None).

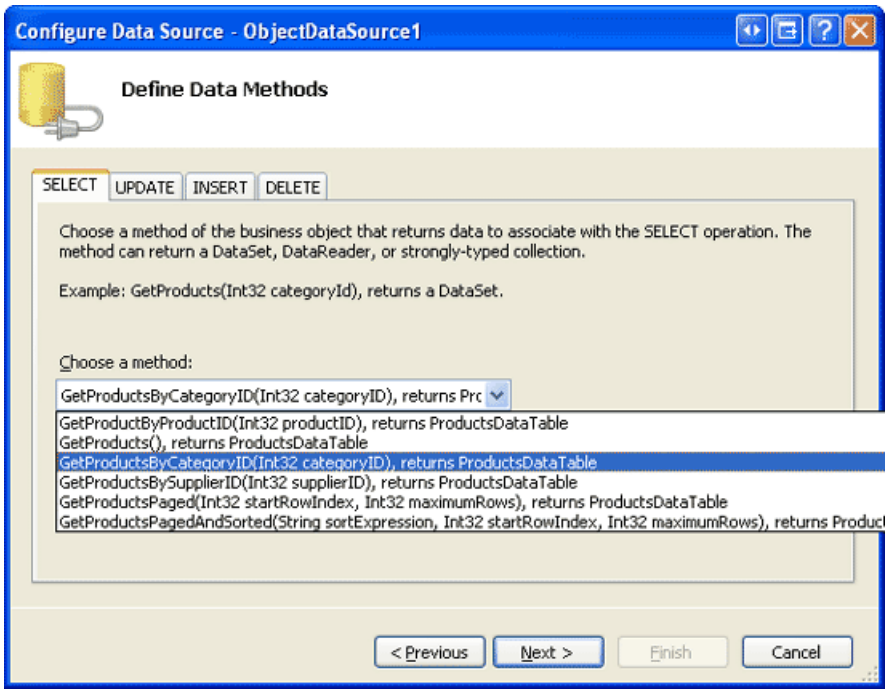


Figure 4: Configure the ObjectDataSource to Use the `ProductsBLL` Class's `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

Since the `GetProductsByCategoryID(categoryID)` method accepts an input parameter (*`categoryID`*), the Choose Data Source wizard offers us an opportunity to specify the parameter's source. Set the parameter source to QueryString using the QueryStringField `CategoryID`.

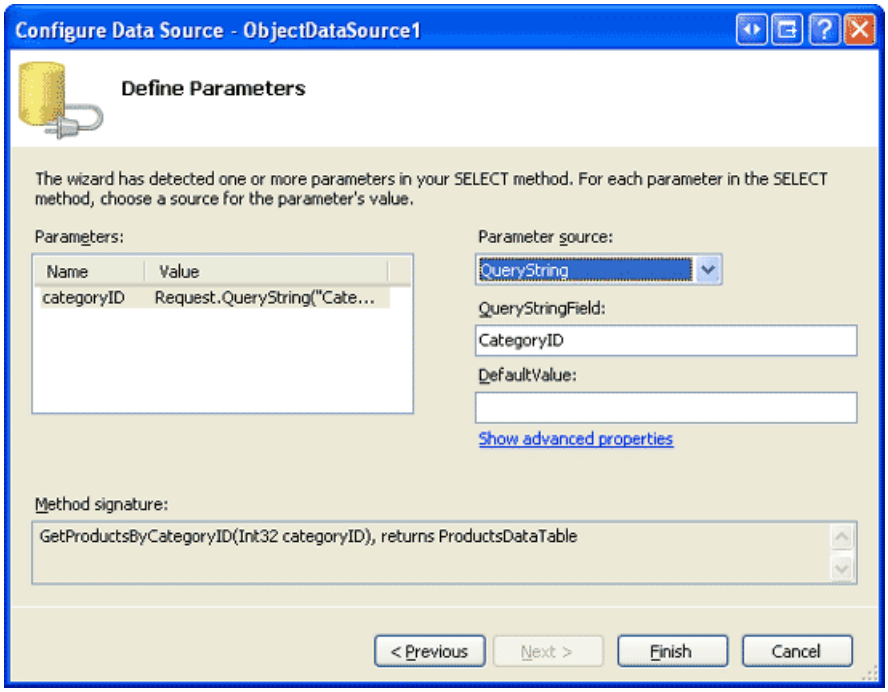


Figure 5: Use the Querystring Field `CategoryID` as the Parameter's Source (Click to view full-size image)

As we've seen in previous tutorials, after completing the Choose Data Source wizard, Visual Studio automatically creates an `ItemTemplate` for the DataList that lists each data field name and value. Replace this template with one that lists only the product's name, supplier, and price. Also, set the DataList's `RepeatColumns` property to 2. After these changes, your DataList and ObjectDataSource's declarative markup should look similar to the following:

```
<asp:DataList ID="ProductsInCategory" DataKeyField="ProductID" RepeatColumns="2"
    DataSourceID="ProductsInCategoryDataSource" EnableViewState="False"
    runat="server">
    <ItemTemplate>
        <h5><%# Eval("ProductName") %></h5>
        <p>
            Supplied by <%# Eval("SupplierName") %><br />
            <%# Eval("UnitPrice", "{0:C}") %>
        </p>
    </ItemTemplate>
</asp:DataList>

<asp:ObjectDataSource ID="ProductsInCategoryDataSource"
    OldValuesParameterFormatString="original_{0}" runat="server"
    SelectMethod="GetProductsByCategoryID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:QueryStringParameter Name="categoryID" QueryStringField="CategoryID"
            Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

To view this page in action, start from the `CategoryListMaster.aspx` page; next, click on a link in the categories bulleted list. Doing so will take you to `ProductsForCategoryDetails.aspx`, passing along the `CategoryID` through the querystring. The `ProductsInCategoryDataSource` ObjectDataSource in `ProductsForCategoryDetails.aspx` will then get just those products for the specified category and display them in the DataList, which renders two products per row. Figure 6 shows a screenshot of `ProductsForCategoryDetails.aspx` when viewing the Beverages.

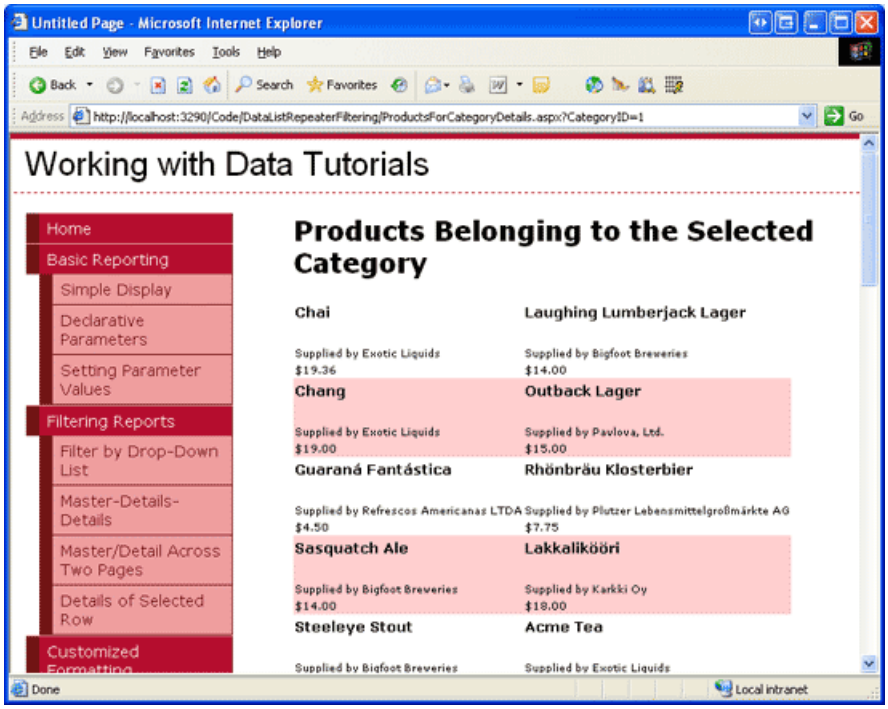


Figure 6: The Beverages are Displayed, Two per Row (Click to view full-size image)

Step 4: Displaying Category Information on ProductsForCategoryDetails.aspx

When a user clicks on a category in `CategoryListMaster.aspx`, they are taken to `ProductsForCategoryDetails.aspx` and shown the products that belong to the selected category. However, in `ProductsForCategoryDetails.aspx` there are no visual cues as to what category was selected. A user that meant to click Beverages, but accidentally clicked Condiments, has no way of realizing their mistake once they reach `ProductsForCategoryDetails.aspx`. To alleviate this potential problem, we can display information about the selected category — its name and description — at

the top of the `ProductsForCategoryDetails.aspx` page.

To accomplish this, add a FormView above the Repeater control in `ProductsForCategoryDetails.aspx`. Next, add a new ObjectDataSource to the page from the FormView's smart tag named `CategoryDataSource` and configure it to use the `CategoriesBLL` class's `GetCategoryByCategoryID(categoryID)` method.

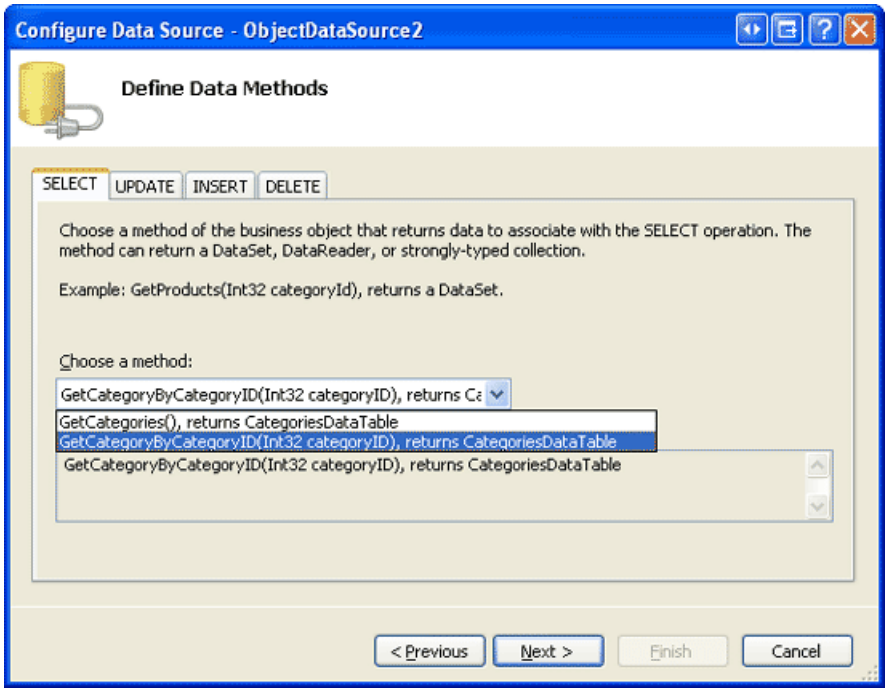


Figure 7: Access Information about the Category through the `CategoriesBLL` Class's `GetCategoryByCategoryID(categoryID)` Method (Click to view full-size image)

As with the `ProductsInCategoryDataSource` ObjectDataSource added in Step 3, the `CategoryDataSource`'s Configure Data Source wizard prompts us for a source for the `GetCategoryByCategoryID(categoryID)` method's input parameter. Use the exact same settings as before, setting the parameter source to QueryString and the QueryStringField value to `CategoryID` (refer back to Figure 5).

After completing the wizard, Visual Studio automatically creates an `ItemTemplate`, `EditItemTemplate`, and `InsertItemTemplate` for the FormView. Since we're providing a read-only interface, feel free to remove the `EditItemTemplate` and `InsertItemTemplate`. Also, feel free to customize the FormView's `ItemTemplate`. After removing the superfluous templates and customizing the ItemTemplate, your FormView and ObjectDataSource's declarative markup should look similar to the following:

```
<asp:FormView ID="FormView1" runat="server" DataKeyNames="CategoryID"
    DataSourceID="CategoryDataSource" EnableViewState="False" Width="100%">
    <ItemTemplate>
        <h3>
            <asp:Label ID="CategoryNameLabel" runat="server"
                Text='<%# Bind("CategoryName") %>' />
        </h3>
        <p>
            <asp:Label ID="DescriptionLabel" runat="server"
                Text='<%# Bind("Description") %>' />
        </p>
    </ItemTemplate>
</asp:FormView>

<asp:ObjectDataSource ID="CategoryDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategoryByCategoryID" TypeName="CategoriesBLL">
    <SelectParameters>
        <asp:QueryStringParameter Name="categoryID" Type="Int32"
            QueryStringField="CategoryID" />
    </SelectParameters>
</asp:ObjectDataSource>
```

Figure 8 shows a screen shot when viewing this page through a browser.

> **NOTE**
>
> In addition to the FormView, I've also added a HyperLink control above the FormView that will take the user back to the list of categories (`CategoryListMaster.aspx`). Feel free to place this link elsewhere or to omit it altogether.

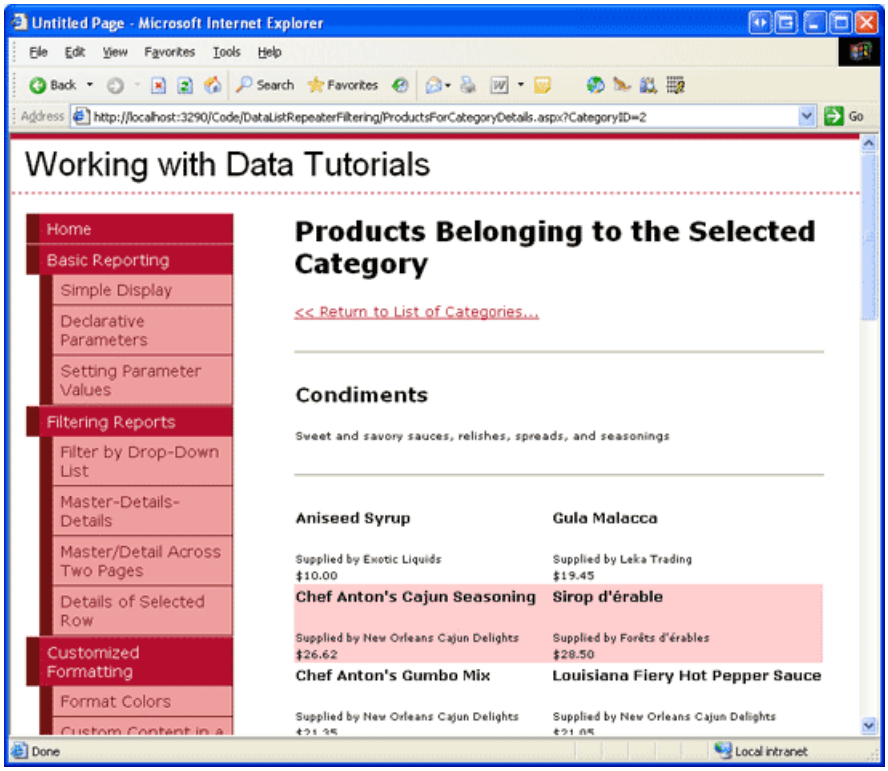


Figure 8: Category Information is Now Displayed at the Top of the Page (Click to view full-size image)

Step 5: Displaying a Message if No Products Belong to the Selected Category

The `CategoryListMaster.aspx` page lists all categories in the system, regardless of whether there are any

associated products. If a user clicks on a category with no associated products, the DataList in `ProductsForCategoryDetails.aspx` will not be rendered, as its data source will not have any items. As we've seen in past tutorials, the GridView provides an `EmptyDataText` property that can be used to specify a text message to display if there are no records in its data source. Unfortunately, neither the DataList nor Repeater has such a property.

In order to display a message informing the user that there are no matching products for the selected category, we need to add a Label control to the page whose `Text` property is assigned the message to display in the event that there are no matching products. We then need to programmatically set its `Visible` property based on whether or not the DataList contains any items.

To accomplish this, start by adding a Label beneath the DataList. Set its `ID` property to `NoProductsMessage` and its `Text` property to "There are no products for the selected category..." Next, we need to programmatically set this Label's `Visible` property based on whether or not any data was bound to the `ProductsInCategory` DataList. This assignment must be made after the data has been bound to the DataList. For the GridView, DetailsView, and FormView, we could create an event handler for the control's `DataBound` event, which fires after databinding has completed. However, neither the DataList nor the Repeater has a `DataBound` event available.

For this particular example we can assign the Label's `Visible` property in the `Page_Load` event handler, since the data will have been assigned to the DataList prior to the page's `Load` event. However, this approach would not work in the general case, as the data from the ObjectDataSource might be bound to the DataList later in the page's lifecycle. For example, if the displayed data is based upon the value in another control, such as it is when displaying a master/detail report using a DropDownList to hold the "master" records, the data may not rebound to the data Web control until the `PreRender` stage in the page's life cycle.

One solution which will work for all cases is to assign the `Visible` property to `False` in the DataList's `ItemDataBound` (or `ItemCreated`) event handler when binding an item type of `Item` or `AlternatingItem`. In such a case we know that there is at least one data item in the data source and therefore can hide the `NoProductsMessage` Label. In addition to this event handler, we also need an event handler for the DataList's `DataBinding` event, where we initialize the Label's `Visible` property to `True`. Since the `DataBinding` event fires before the `ItemDataBound` events, the Label's `Visible` property will initially be set to `True`; if there are any data items, however, it will be set to `False`. The following code implements this logic:

```
Protected Sub ProductsInCategory_DataBinding(sender As Object, e As EventArgs) _
    Handles ProductsInCategory.DataBinding
    'Show the Label
    NoProductsMessage.Visible = True
End Sub

Protected Sub ProductsInCategory_ItemDataBound(s As Object, e As DataListItemEventArgs) _
    Handles ProductsInCategory.ItemDataBound
    'If we have a data item, hide the Label
    If e.Item.ItemType = ListItemType.Item OrElse e.Item.ItemType = _
        ListItemType.AlternatingItem Then

        NoProductsMessage.Visible = False
    End If
End Sub
```

All of the categories in the Northwind database are associated with one or more products. To test this feature, I've manually adjusted the Northwind database for this tutorial, reassigning all products associated with the Produce category (`CategoryID` = 7) to the Seafood category (`CategoryID` = 8). This can be accomplished from the Server Explorer by choosing New Query and using the following `UPDATE` statement:

```
UPDATE Products SET
    CategoryID = 8
WHERE CategoryID = 7
```

After updating the database accordingly, return to the `CategoryListMaster.aspx` page and click on the Produce link. Since there are no longer any products belonging to the Produce category, you should see the "There are no products for the selected category..." message, as shown in Figure 9.

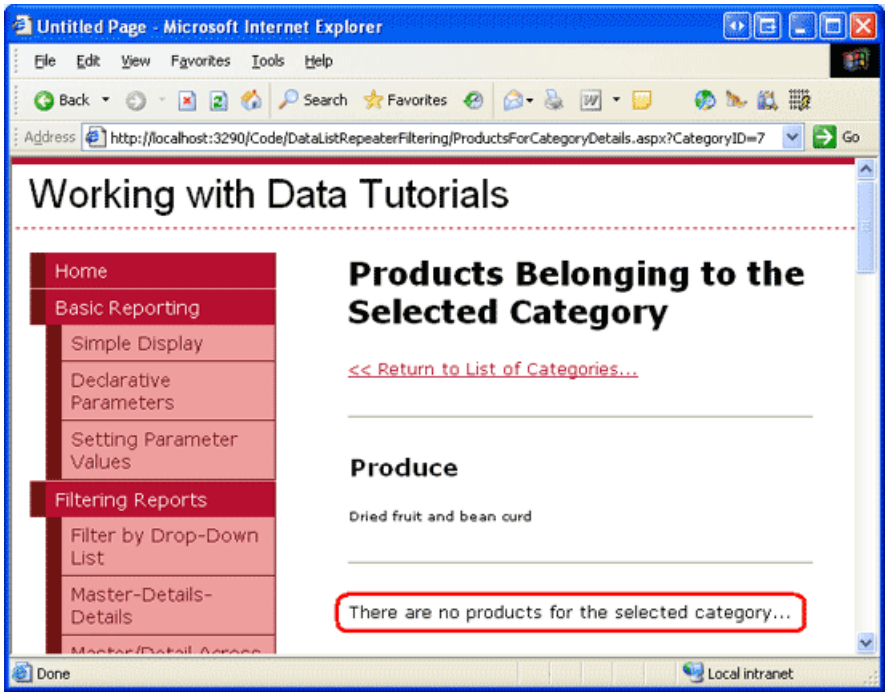


Figure 9: A Message is Displayed if there are No Products Belonging to the Selected Category (Click to view full-size image)

Summary

While master/detail reports can display both the master and detail records on a single page, in many websites they are separated out across two web pages. In this tutorial we looked at how to implement such a master/detail report by having the categories listed in a bulleted list using a Repeater in the "master" web page and the associated products listed in the "details" page. Each list item in the master web page contained a link to the details page that passed along the row's `CategoryID` value.

In the details page retrieving those products for the specified supplier was accomplished through the `ProductsBLL` class's `GetProductsByCategoryID(categoryID)` method. The `categoryID` parameter value was specified declaratively using the `CategoryID` querystring value as the parameter source. We also looked at how to display category details in the details page using a FormView and how to display a message if there were no products belonging to the selected category.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Zack Jones and Liz Shulok. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Master/Detail Using a Bulleted List of Master Records with a Details DataList (VB)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll compress the two-page master/detail report of the previous tutorial into a single page, showing a bulleted list of category names on the left side of the screen and the selected category's products on the right of the screen.

Introduction

In the preceding tutorial we looked at how to separate a master/detail report across two pages. In the master page we used a Repeater control to render a bulleted list of categories. Each category name was a hyperlink that, when clicked, would take the user to the details page, where a two-column DataList showed those products belonging to the selected category.

In this tutorial we'll compress the two-page tutorial into a single page, showing a bulleted list of category names on the left side of the screen with each category name rendered as a LinkButton. Clicking one of the category name LinkButtons induces a postback and binds the selected category s products to a two-column DataList on the right of the screen. In addition to displaying each category s name, the Repeater on the left shows how many total products there are for a given category (see Figure 1).

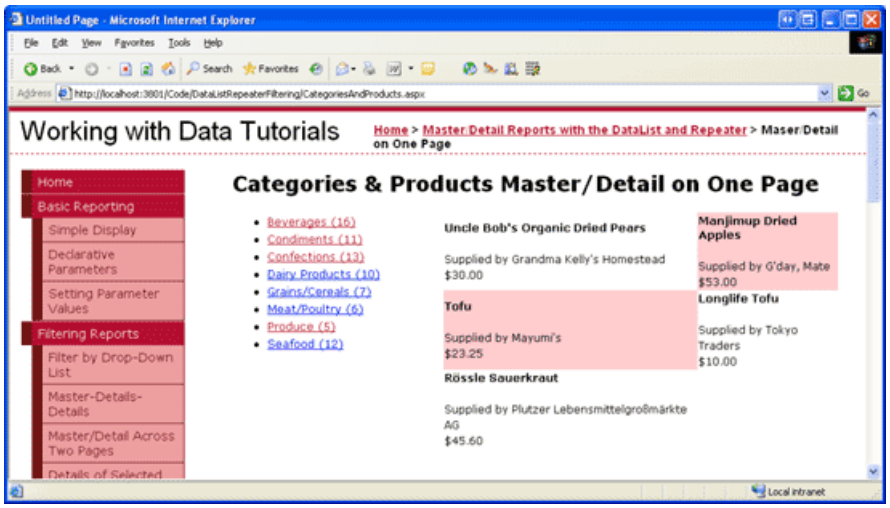


Figure 1: The Category s Name and Total Number of Products are Displayed on the Left (Click to view full-size image)

Step 1: Displaying a Repeater in the Left Portion of the Screen

For this tutorial we need to have the bulleted list of categories appear to the left of the selected category s products. Content within a web page can be positioned using standard HTML elements paragraph tags, non-breaking spaces, `<table>` s, and so on or through cascading stylesheet (CSS) techniques. All of our tutorials thus far have used CSS techniques for positioning. When we built the navigation user interface in our master page in the Master Pages and Site Navigation tutorial we used *absolute positioning*, indicating the precise pixel offset for the navigation list and the main content. Alternatively, CSS can be used to position one element to the right or left

of another through *floating*. We can have the bulleted list of categories appear to the left of the selected category s products by floating the Repeater to the left of the DataList

Open the `CategoriesAndProducts.aspx` page from the `DataListRepeaterFiltering` folder and add to the page a Repeater and a DataList. Set the Repeater s `ID` to `Categories` and the DataList s to `CategoryProducts`. Go to the Source view and put the Repeater and DataList controls within their own `<div>` elements. That is, enclose the Repeater within a `<div>` element first and then the DataList in its own `<div>` element directly after the Repeater. Your markup at this point should look similar to the following:

```
<div>
    <asp:Repeater ID="Categories" runat="server">
    </asp:Repeater>
</div>
<div>
    <asp:DataList ID="CategoryProducts" runat="server">
    </asp:DataList>
</div>
```

To float the Repeater to the left of the DataList, we need to use the `float` CSS style attribute, like so:

```
<div>
    Repeater
</div>
<div>
    DataList
</div>
```

The `float: left;` floats the first `<div>` element to the left of the second one. The `width` and `padding-right` settings indicate the first `<div>` s `width` and how much padding is added between the `<div>` element s content and its right margin. For more information on floating elements in CSS check out the Floatutorial.

Rather than specify the style setting directly through the first `<p>` element s `style` attribute, let s instead create a new CSS class in `Styles.css` named `FloatLeft`:

```
.FloatLeft
{
    float: left;
    width: 33%;
    padding-right: 10px;
}
```

Then we can replace the `<div>` with `<div class="FloatLeft">`.

After adding the CSS class and configuring the markup in the `CategoriesAndProducts.aspx` page, go to the Designer. You should see the Repeater floating to the left of the DataList (although right now both just appear as gray boxes since we ve yet to configure their data sources or templates).

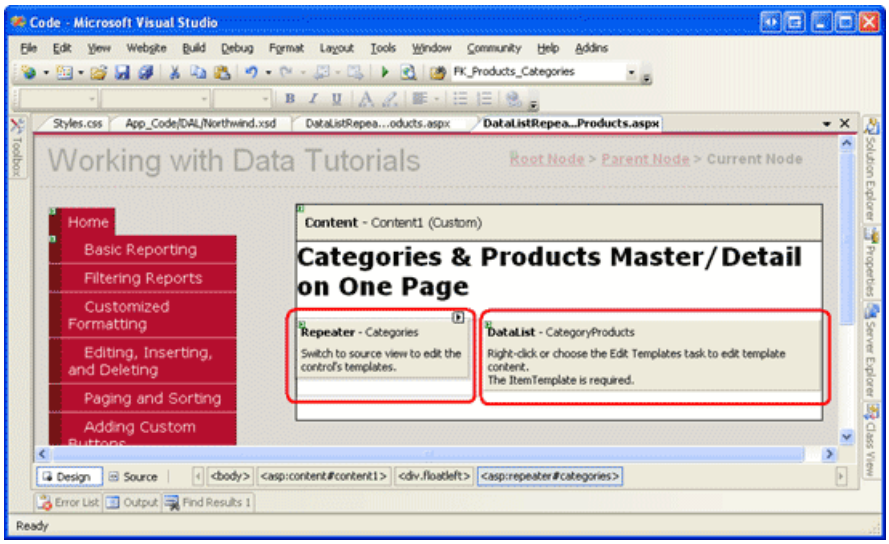


Figure 2: The Repeater is Floated to the Left of the DataList (Click to view full-size image)

Step 2: Determining the Number of Products for Each Category

With the Repeater and DataList s surrounding markup complete, we re ready to bind the category data to the Repeater control. However, as the bulleted list of categories in Figure 1 shows, in addition to each category s name we also need to display the number of products associated with the category. To access this information we can either:

- **Determine this information from the ASP.NET page s code-behind class.** Given a particular `categoryID` we can determine the number of associated products by calling the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method. This method returns a `ProductsDataTable` object whose `Count` property indicates how many `ProductsRow` s exists, which is the number of products for the specified `categoryID`. We can create an `ItemDataBound` event handler for the Repeater that, for each category bound to the Repeater, calls the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method and includes its count in the output.
- **Update the `CategoriesDataTable` in the Typed DataSet to include a `NumberOfProducts` column.** We can then update the `GetCategories()` method in the `CategoriesDataTable` to include this information or, alternatively, leave `GetCategories()` as-is and create a new `CategoriesDataTable` method called `GetCategoriesAndNumberOfProducts()`.

Let s explore both of these techniques. The first approach is simpler to implement since we don t need to update the Data Access Layer; however, it requires more communications with the database. The call to the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method in the `ItemDataBound` event handler adds an extra database call for each category displayed in the Repeater. With this technique there are $N + 1$ database calls, where N is the number of categories displayed in the Repeater. With the second approach, the product count is returned with information about each category from the `CategoriesBLL` class s `GetCategories()` (or `GetCategoriesAndNumberOfProducts()`) method, thereby resulting in just one trip to the database.

Determining the Number of Products in the ItemDataBound Event Handler

Determining the number of products for each category in the Repeater s `ItemDataBound` event handler does not require any modifications to our existing Data Access Layer. All modifications can be made directly within the `CategoriesAndProducts.aspx` page. Start by adding a new ObjectDataSource named `CategoriesDataSource` via the Repeater s smart tag. Next, configure the `CategoriesDataSource` ObjectDataSource so that it retrieves its data from the `CategoriesBLL` class s `GetCategories()` method.

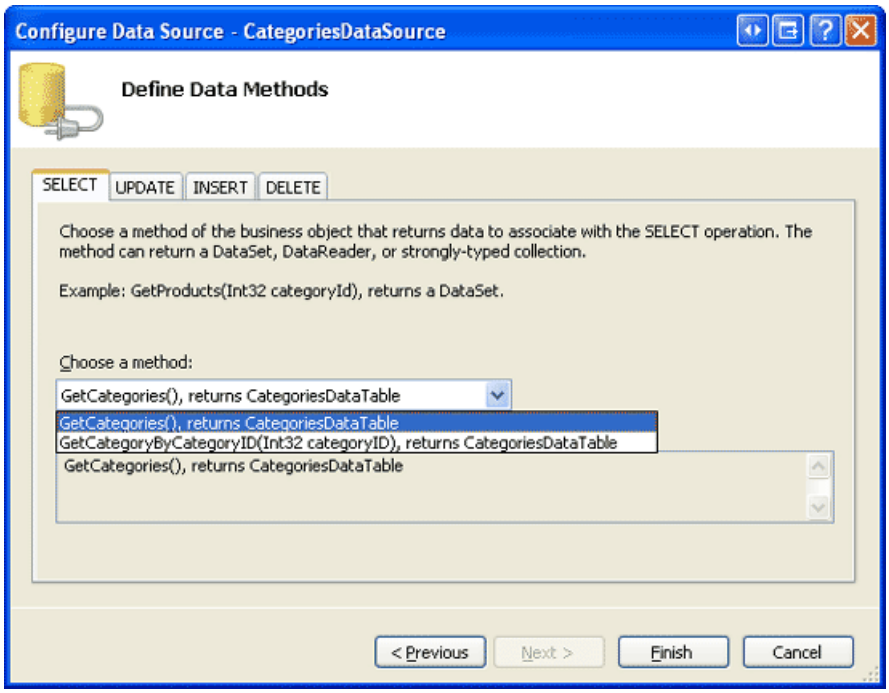


Figure 3: Configure the ObjectDataSource to Use the `CategoriesBLL` class s `GetCategories()` Method (Click to view full-size image)

Each item in the `Categories` Repeater needs to be clickable and, when clicked, cause the `CategoryProducts` DataList to display those products for the selected category. This can be accomplished by making each category a hyperlink, linking back to this same page (`CategoriesAndProducts.aspx`), but passing the `CategoryID` through the querystring, much like we saw in the previous tutorial. The advantage of this approach is that a page displaying a particular category s products can be bookmarked and indexed by a search engine.

Alternatively, we can make each category a LinkButton, which is the approach we'll use for this tutorial. The LinkButton renders in the user s browser as a hyperlink but, when clicked, induces a postback; on postback, the DataList s ObjectDataSource needs to be refreshed to display those products belonging to the selected category. For this tutorial, using a hyperlink makes more sense than using a LinkButton; however, there may be other scenarios where using a LinkButton is more advantageous. While the hyperlink approach would be ideal for this example, let s instead explore using the LinkButton. As we'll see, using a LinkButton introduces some challenges that would not otherwise arise with a hyperlink. Therefore, using a LinkButton in this tutorial will highlight these challenges and help provide solutions for those scenarios where we may want to use a LinkButton instead of a hyperlink.

> **NOTE**
>
> You are encouraged to repeat this tutorial using a HyperLink control or `<a>` element in lieu of the LinkButton.

The following markup shows the declarative syntax for the Repeater and the ObjectDataSource. Note that the Repeater s templates render a bulleted list with each item as a LinkButton:

```
<asp:Repeater ID="Categories" runat="server" DataSourceID="CategoriesDataSource">
    <HeaderTemplate>
        <ul>
    </HeaderTemplate>
    <ItemTemplate>
        <li><asp:LinkButton runat="server" ID="ViewCategory"></asp:LinkButton></li>
    </ItemTemplate>
    <FooterTemplate>
        </ul>
    </FooterTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

NOTE

For this tutorial the Repeater must have its view state enabled (note the omission of the `EnableViewState="False"` from the Repeater s declarative syntax). In step 3 we'll be creating an event handler for the Repeater s `ItemCommand` event in which we'll be updating the DataList s ObjectDataSource s `SelectParameters` collection. The Repeater s `ItemCommand`, however, won't fire if view state is disabled. See A Stumper of an ASP.NET Question and its solution for more information on why view state must be enabled for a Repeater s `ItemCommand` event to fire.

The LinkButton with the `ID` property value of `ViewCategory` does not have its `Text` property set. If we had just wanted to display the category name, we would have set the Text property declaratively, through databinding syntax, like so:

```
<asp:LinkButton runat="server" ID="ViewCategory"
    Text='<%# Eval("CategoryName") %>' />
```

However, we want to show both the category s name *and* the number of products belonging to that category. This information can be retrieved from the Repeater s `ItemDataBound` event handler by making a call to the `ProductBLL` class s `GetCategoriesByProductID(categoryID)` method and determining how many records are returned in the resulting `ProductsDataTable`, as the following code illustrates:

```
Protected Sub Categories_ItemDataBound(sender As Object, e As RepeaterItemEventArgs)
    ' Make sure we're working with a data item...
    If e.Item.ItemType = ListItemType.Item OrElse _
        e.Item.ItemType = ListItemType.AlternatingItem Then
        ' Reference the CategoriesRow instance bound to this RepeaterItem
        Dim category As Northwind.CategoriesRow = _
            CType(CType(e.Item.DataItem, System.Data.DataRowView).Row, _
                Northwind.CategoriesRow)
        ' Determine how many products are in this category
        Dim productsAPI As New NorthwindTableAdapters.ProductsTableAdapter
        Dim productCount As Integer = _
            productsAPI.GetProductsByCategoryID(category.CategoryID).Count
        ' Reference the ViewCategory LinkButton and set its Text property
        Dim ViewCategory As LinkButton = _
            CType(e.Item.FindControl("ViewCategory"), LinkButton)
        ViewCategory.Text = _
            String.Format("{0} ({1:N0})", category.CategoryName, productCount)
    End If
End Sub
```

We start out by ensuring that we re working with a data item (one whose `ItemType` is `Item` or `AlternatingItem`)

and then reference the `CategoriesRow` instance that has just been bound to the current `RepeaterItem`. Next, we determine the number of products for this category by creating an instance of the `ProductsBLL` class, calling its `GetCategoriesByProductID(categoryID)` method, and determining the number of records returned using the `Count` property. Finally, the `ViewCategory` LinkButton in the ItemTemplate is references and its `Text` property is set to *CategoryName* (*NumberOfProductsInCategory*), where *NumberOfProductsInCategory* is formatted as a number with zero decimal places.

> **NOTE**
>
> Alternatively, we could have added a *formatting function* to the ASP.NET page s code-behind class that accepts a category s `CategoryName` and `CategoryID` values and returns the `CategoryName` concatenated with the number of products in the category (as determined by calling the `GetCategoriesByProductID(categoryID)` method). The results of such a formatting function could be declaratively assigned to the LinkButton s Text property replacing the need for the `ItemDataBound` event handler. Refer to the Using TemplateFields in the GridView Control or Formatting the DataList and Repeater Based Upon Data tutorials for more information on using formatting functions.

After adding this event handler, take a moment to test the page through a browser. Note how each category is listed in a bulleted list, displaying the category s name and the number of products associated with the category (see Figure 4).

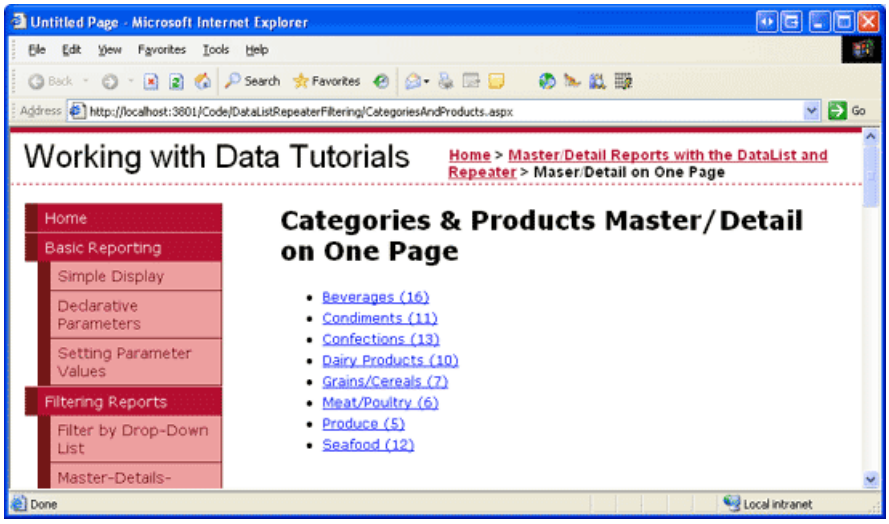


Figure 4: Each Category s Name and Number of Products are Displayed (Click to view full-size image)

Updating the `CategoriesDataTable` and `CategoriesTableAdapter` to Include the Number of Products for Each Category

Rather than determining the number of products for each category as it s bound to the Repeater, we can streamline this process by adjusting the `CategoriesDataTable` and `CategoriesTableAdapter` in the Data Access Layer to include this information natively. To achieve this, we must add a new column to `CategoriesDataTable` to hold the number of associated products. To add a new column to a DataTable, open the Typed DataSet (`App_Code\DAL\Northwind.xsd`), right-click on the DataTable to modify, and choose Add / Column. Add a new column to the `CategoriesDataTable` (see Figure 5).

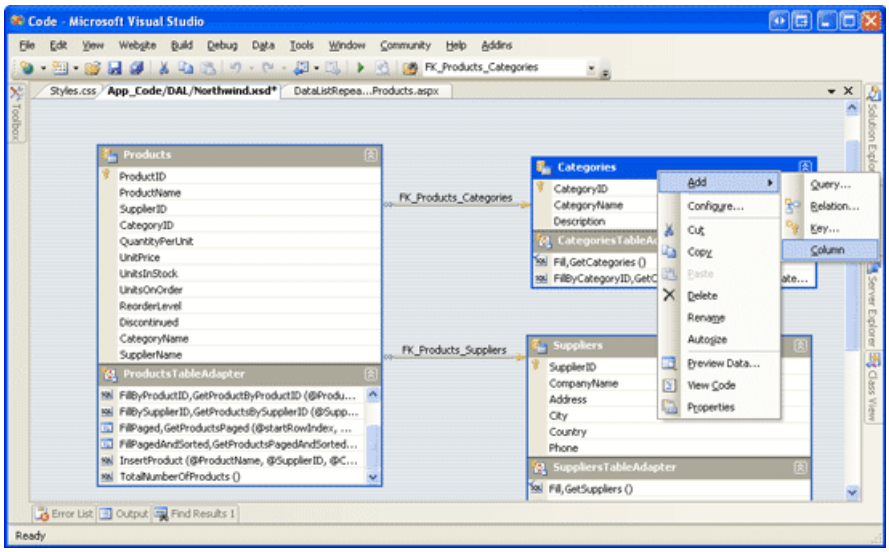


Figure 5: Add a New Column to the `CategoriesDataSource` (Click to view full-size image)

This will add a new column named `Column1`, which you can change by simply typing in a different name. Rename this new column to `NumberOfProducts`. Next, we need to configure this column s properties. Click on the new column and go to the Properties window. Change the column s `DataType` property from `System.String` to `System.Int32` and set the `ReadOnly` property to `True`, as shown in Figure 6.

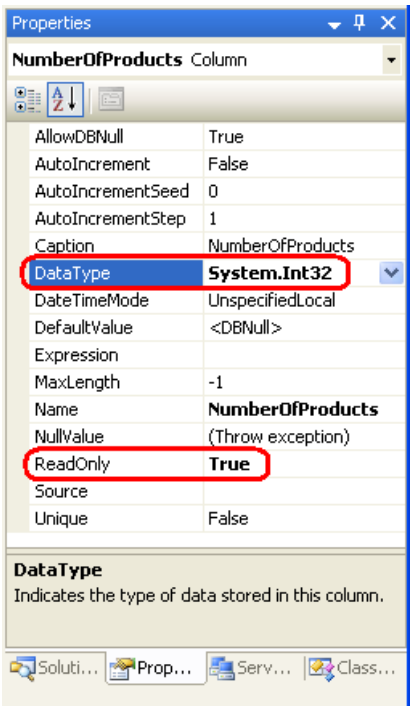


Figure 6: Set the `DataType` and `ReadOnly` Properties of the New Column

While the `CategoriesDataTable` now has a `NumberOfProducts` column, its value is not set by any of the corresponding TableAdapter s queries. We can update the `GetCategories()` method to return this information if we want such information returned every time category information is retrieved. If, however, we only need to grab the number of associated products for the categories in rare instances (such as just for this tutorial), then we can leave `GetCategories()` as-is and create a new method that returns this information. Let s use this latter approach, creating a new method named `GetCategoriesAndNumberOfProducts()`.

To add this new `GetCategoriesAndNumberOfProducts()` method, right-click on the `CategoriesTableAdapter` and choose New Query. This brings up the TableAdapter Query Configuration Wizard, which we ve used numerous times in previous tutorials. For this method, start the wizard by indicating that the query uses an ad-hoc SQL statement that returns rows.

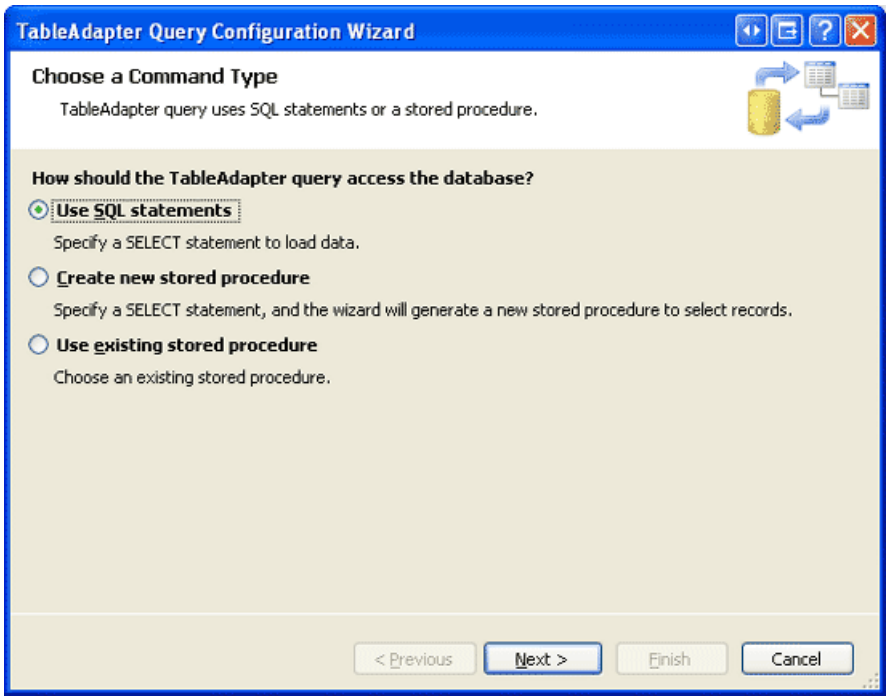


Figure 7: Create the Method Using an Ad-Hoc SQL Statement (Click to view full-size image)

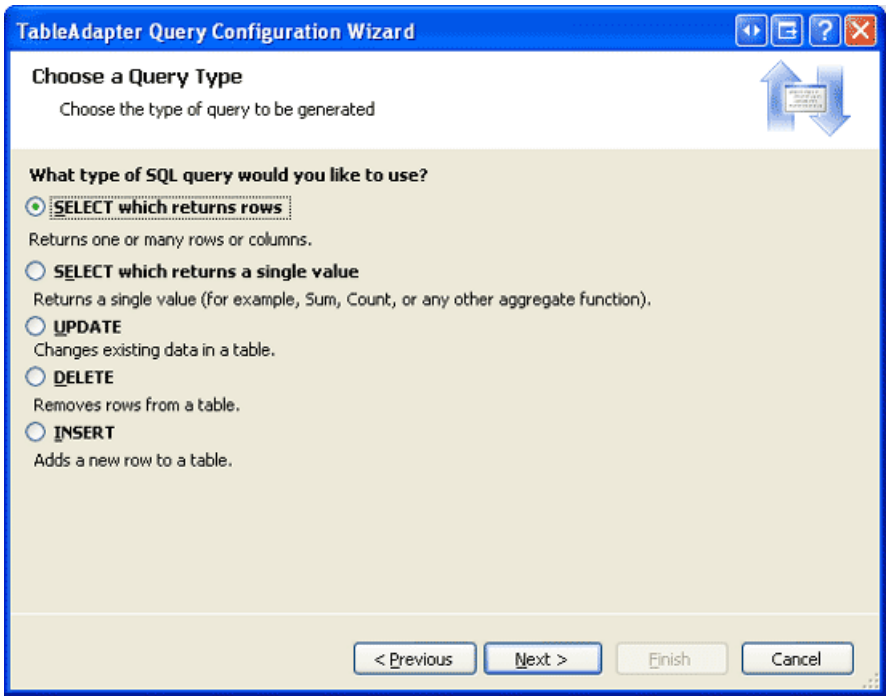


Figure 8: The SQL Statement Returns Rows (Click to view full-size image)

The next wizard screen prompts us for the query to use. To return each category s `CategoryID`, `CategoryName`, and `Description` fields, along with the number of products associated with the category, use the following `SELECT` statement:

```
SELECT CategoryID, CategoryName, Description,
       (SELECT COUNT(*) FROM Products p WHERE p.CategoryID = c.CategoryID)
            as NumberOfProducts
FROM Categories c
```

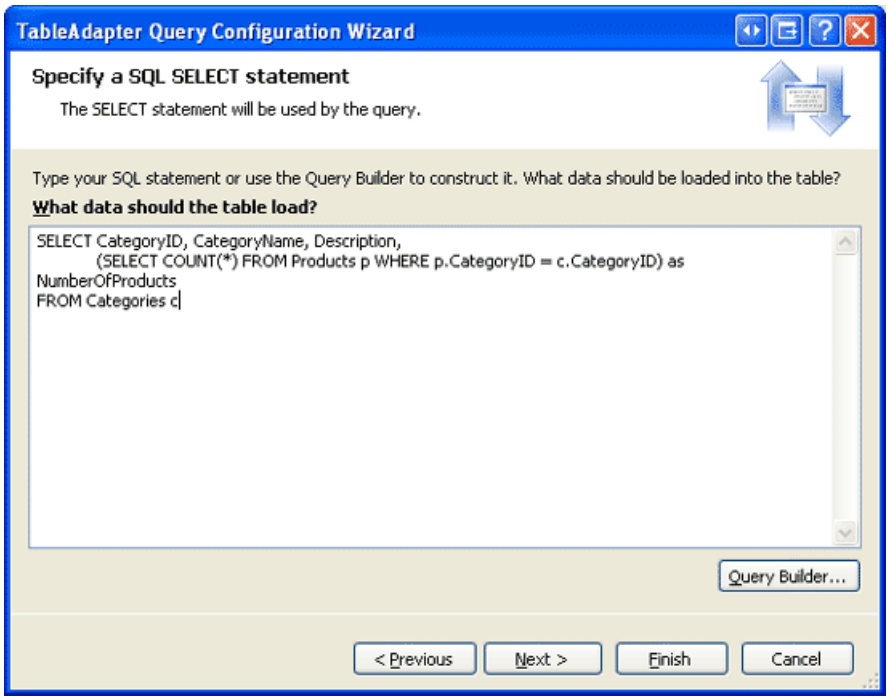


Figure 9: Specify the Query to Use (Click to view full-size image)

Note that the subquery that computes the number of products associated with the category is aliased as `NumberOfProducts`. This naming match causes the value returned by this subquery to be associated with the `CategoriesDataTable` s `NumberOfProducts` column.

After entering this query, the last step is to choose the name for the new method. Use `FillWithNumberOfProducts` and `GetCategoriesAndNumberOfProducts` for the Fill a DataTable and Return a DataTable patterns, respectively.

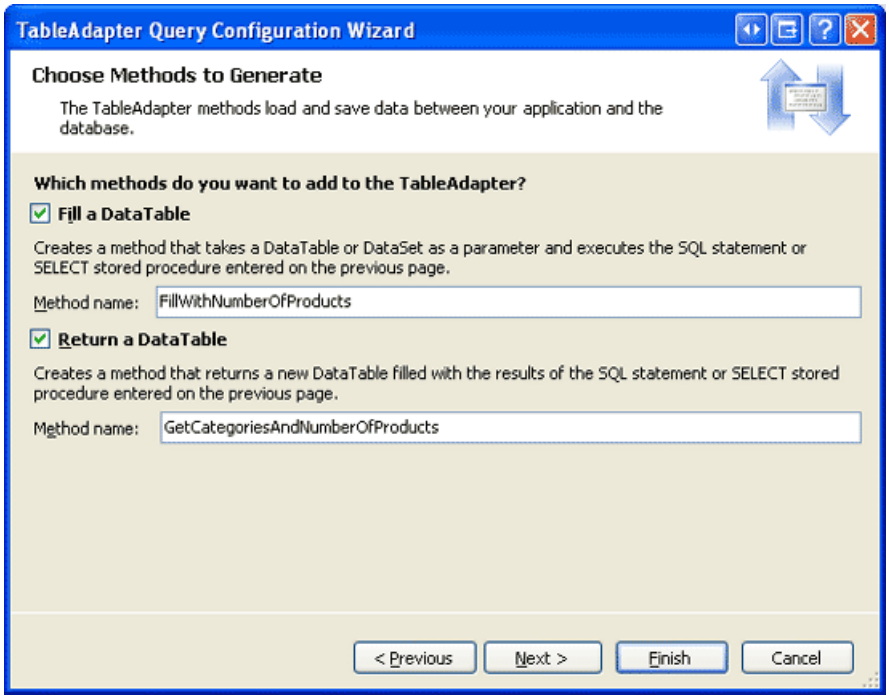


Figure 10: Name the New TableAdapter s Methods `FillWithNumberOfProducts` and `GetCategoriesAndNumberOfProducts` (Click to view full-size image)

At this point the Data Access Layer has been extended to include the number of products per category. Since all our presentation layer routes all calls to the DAL through a separate Business Logic Layer we need to add a corresponding `GetCategoriesAndNumberOfProducts` method to the `CategoriesBLL` class:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetCategoriesAndNumberOfProducts() As Northwind.CategoriesDataTable
    Return Adapter.GetCategoriesAndNumberOfProducts()
End Function
```

With the DAL and BLL complete, we re ready to bind this data to the `Categories` Repeater in `CategoriesAndProducts.aspx`! If you ve already created an ObjectDataSource for the Repeater from the Determining the Number of Products in the `ItemDataBound` Event Handler section, delete this ObjectDataSource and remove the Repeater s `DataSourceID` property setting; also unwire the Repeater s `ItemDataBound` event from the event handler by removing the `Handles Categories.OnItemDataBound` syntax in the ASP.NET code-behind class.

With the Repeater back in its original state, add a new ObjectDataSource named `CategoriesDataSource` via the Repeater s smart tag. Configure the ObjectDataSource to use the `CategoriesBLL` class, but instead of having it use the `GetCategories()` method, have it use `GetCategoriesAndNumberOfProducts()` instead (see Figure 11).

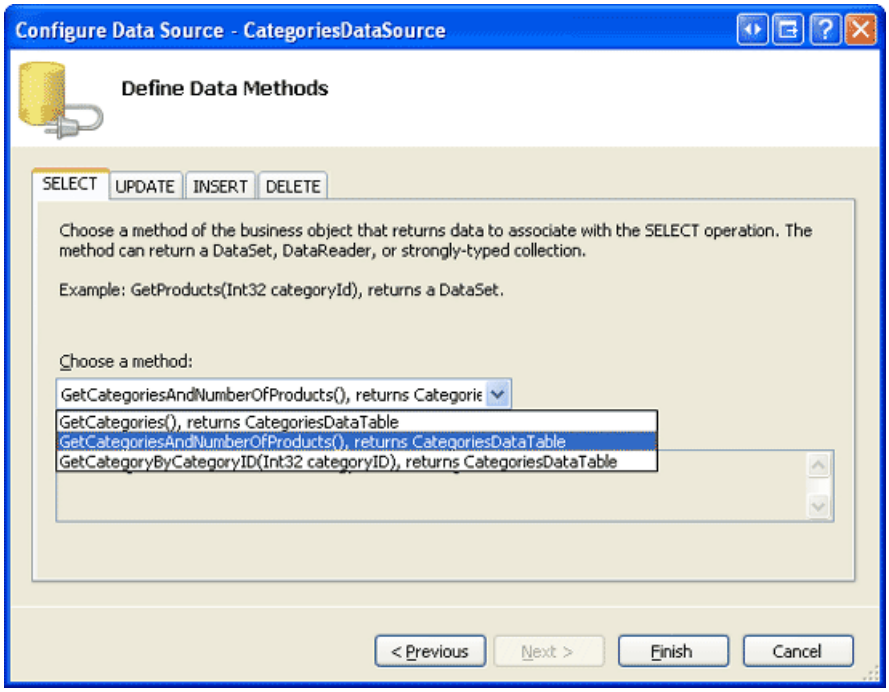


Figure 11: Configure the ObjectDataSource to Use the `GetCategoriesAndNumberOfProducts` Method (Click to view full-size image)

Next, update the `ItemTemplate` so that the LinkButton s `Text` property is declaratively assigned using databinding syntax and includes both the `CategoryName` and `NumberOfProducts` data fields. The complete declarative markup for the Repeater and the `CategoriesDataSource` ObjectDataSource follow:

```
<asp:Repeater ID="Categories" runat="server" DataSourceID="CategoriesDataSource">
    <HeaderTemplate>
        <ul>
    </HeaderTemplate>
    <ItemTemplate>
        <li><asp:LinkButton runat="server" ID="ViewCategory"
                Text='<%# String.Format("{0} ({1:N0})", _
                    Eval("CategoryName"), Eval("NumberOfProducts")) %>' />
        </li>
    </ItemTemplate>
    <FooterTemplate>
        </ul>
    </FooterTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategoriesAndNumberOfProducts" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

The output rendered by updating the DAL to include a `NumberOfProducts` column is the same as using the `ItemDataBound` event handler approach (refer back to Figure 4 to see a screen shot of the Repeater showing the category names and number of products).

Step 3: Displaying the Selected Category s Products

At this point we have the `Categories` Repeater displaying the list of categories along with the number of products in each category. The Repeater uses a LinkButton for each category that, when clicked, causes a postback, at which point we need to display those products for the selected category in the `CategoryProducts` DataList.

One challenge facing us is how to have the DataList display just those products for the selected category. In the Master/Detail Using a Selectable Master GridView with a Details DetailsView tutorial we saw how to build a GridView whose rows could be selected, with the selected row s details being displayed in a DetailsView on the same page. The GridView s ObjectDataSource returned information about all products using the `ProductsBLL` s `GetProducts()` method while the DetailsView s ObjectDataSource retrieved information about the selected product using the `GetProductsByProductID(productID)` method. The *`productID`* parameter value was provided declaratively by associating it with the value of the GridView s `SelectedValue` property. Unfortunately, the Repeater does not have a `SelectedValue` property and cannot serve as a parameter source.

> **NOTE**
> This is one of those challenges that appear when using the LinkButton in a Repeater. Had we used a hyperlink to pass in the `CategoryID` through the querystring instead, we could use that QueryString field as the source for the parameter s value.

Before we worry about the lack of a `SelectedValue` property for the Repeater, though, let s first bind the DataList to an ObjectDataSource and specify its `ItemTemplate`.

From the DataList s smart tag, opt to add a new ObjectDataSource named `CategoryProductsDataSource` and configure it to use the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method. Since the DataList in this tutorial offers a read-only interface, feel free to set the drop-down lists in the INSERT, UPDATE, and DELETE tabs to (None).

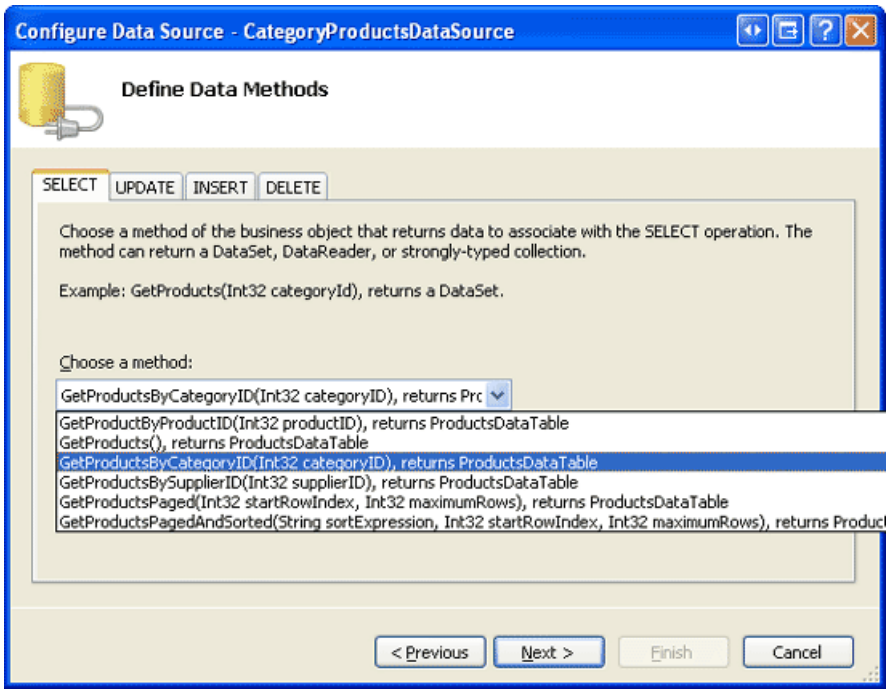


Figure 12: Configure the ObjectDataSource to Use `ProductsBLL` Class s `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

Since the `GetProductsByCategoryID(categoryID)` method expects an input parameter (*`categoryID`*), the Configure Data Source wizard allows us to specify the parameter s source. Had the categories been listed in a GridView or a DataList, we d set the Parameter source drop-down list to Control and the ControlID to the `ID` of the data Web control. However, since the Repeater lacks a `SelectedValue` property it cannot be used as a parameter source. If you check, you'll find that the ControlID drop-down list only contains one control `ID`` `CategoryProducts`, the `ID` of the DataList.

For now, set the Parameter source drop-down list to None. We'll end up programmatically assigning this parameter value when a category LinkButton is clicked in the Repeater.

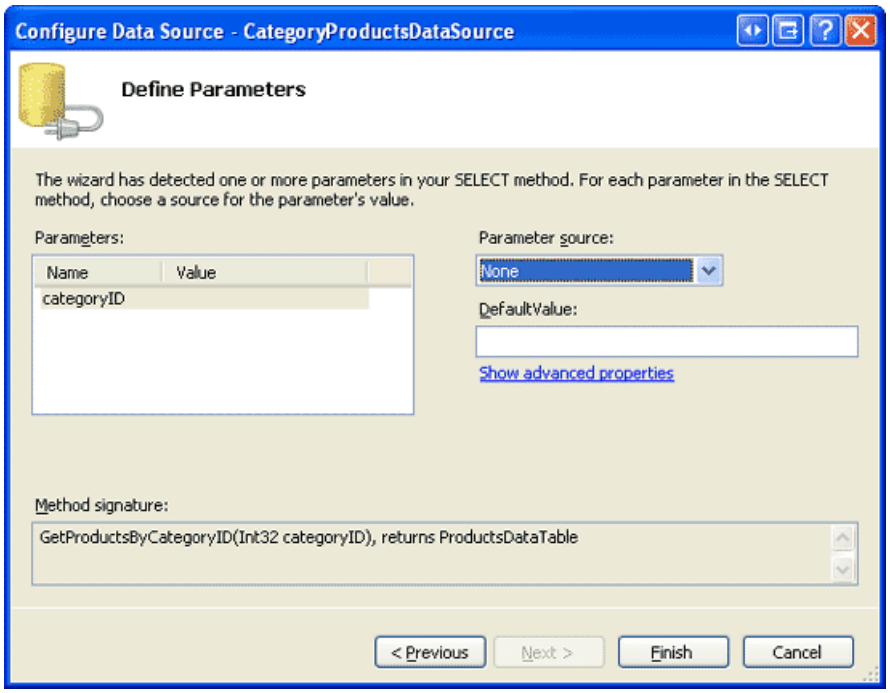


Figure 13: Do Not Specify a Parameter Source for the *`categoryID`* Parameter (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio auto-generates the DataList s `ItemTemplate`. Replace this default `ItemTemplate` with the template we used in the preceding tutorial; also, set the DataList s

`RepeatColumns` property to 2. After making these changes the declarative markup for your DataList and its associated ObjectDataSource should look like the following:

```
<asp:DataList ID="CategoryProducts" runat="server" DataKeyField="ProductID"
    DataSourceID="CategoryProductsDataSource" RepeatColumns="2"
    EnableViewState="False">
    <ItemTemplate>
        <h5><%# Eval("ProductName") %></h5>
        <p>
            Supplied by <%# Eval("SupplierName") %><br />
            <%# Eval("UnitPrice", "{0:C}") %>
        </p>
    </ItemTemplate>
</asp:DataList>
<asp:ObjectDataSource ID="CategoryProductsDataSource"
    OldValuesParameterFormatString="original_{0}"  runat="server"
    SelectMethod="GetProductsByCategoryID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:Parameter Name="categoryID" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

Currently, the `CategoryProductsDataSource` ObjectDataSource s *`categoryID`* parameter is never set, so no products are displayed when viewing the page. What we need to do is have this parameter value set based on the `CategoryID` of the clicked category in the Repeater. This introduces two challenges: first, how do we determine when a LinkButton in the Repeater s `ItemTemplate` has been clicked; and second, how can we determine the `CategoryID` of the corresponding category whose LinkButton was clicked?

The LinkButton like the Button and ImageButton controls has a `Click` event and a `Command` event. The `Click` event is designed to simply note that the LinkButton has been clicked. At times, however, in addition to noting that the LinkButton has been clicked we also need to pass some extra information to the event handler. If this is the case, the LinkButton s `CommandName` and `CommandArgument` properties can be assigned this extra information. Then, when the LinkButton is clicked, its `Command` event fires (instead of its `Click` event) and the event handler is passed the values of the `CommandName` and `CommandArgument` properties.

When a `Command` event is raised from within a template in the Repeater, the Repeater s `ItemCommand` event fires and is passed the `CommandName` and `CommandArgument` values of the clicked LinkButton (or Button or ImageButton). Therefore, to determine when a category LinkButton in the Repeater has been clicked, we need to do the following:

1. Set the `CommandName` property of the LinkButton in the Repeater s `ItemTemplate` to some value (I ve used ListProducts). By setting this `CommandName` value, the LinkButton s `Command` event fires when the LinkButton is clicked.
2. Set the LinkButton s `CommandArgument` property to the value of the current item s `CategoryID`.
3. Create an event handler for the Repeater s `ItemCommand` event. In the event handler, set the `CategoryProductsDataSource` ObjectDataSource s `CategoryID` parameter to the value of the passed-in `CommandArgument`.

The following `ItemTemplate` markup for the Categories Repeater implements steps 1 and 2. Note how the `CommandArgument` value is assigned the data item s `CategoryID` using databinding syntax:

```
<ItemTemplate>
    <li>
        <asp:LinkButton CommandName="ListProducts"  runat="server"
            CommandArgument='<%# Eval("CategoryID") %>' ID="ViewCategory"
            Text='<%# string.Format("{0} ({1:N0})", _
                Eval("CategoryName"), Eval("NumberOfProducts")) %>'>
        </asp:LinkButton>
    </li>
</ItemTemplate>
```

Whenever creating an `ItemCommand` event handler it is prudent to always first check the incoming `CommandName` value because *any* `Command` event raised by *any* Button, LinkButton, or ImageButton within the Repeater will cause the `ItemCommand` event to fire. While we currently only have one such LinkButton now, in the future we (or another developer on our team) might add additional button Web controls to the Repeater that, when clicked, raises the same `ItemCommand` event handler. Therefore, it s best to always make sure you check the `CommandName` property and only proceed with your programmatic logic if it matches up to the value expected.

After ensuring that the passed-in `CommandName` value equals ListProducts, the event handler then assigns the `CategoryProductsDataSource` ObjectDataSource s `CategoryID` parameter to the value of the passed-in `CommandArgument`. This modification to the ObjectDataSource s `SelectParameters` automatically causes the DataList to rebind itself to the data source, showing the products for the newly selected category.

```
Protected Sub Categories_ItemCommand(source As Object, e As RepeaterCommandEventArgs) _
    Handles Categories.ItemCommand
    ' If it's the "ListProducts" command that has been issued...
    If String.Compare(e.CommandName, "ListProducts", True) = 0 Then
        ' Set the CategoryProductsDataSource ObjectDataSource's CategoryID parameter
        ' to the CategoryID of the category that was just clicked (e.CommandArgument)...
        CategoryProductsDataSource.SelectParameters("CategoryID").DefaultValue = _
            e.CommandArgument.ToString()
    End If
End Sub
```

With these additions, our tutorial is complete! Take a moment to test it out in a browser. Figure 14 shows the screen when first visiting the page. Since a category has yet to be selected, no products are displayed. Clicking on a category, such as Produce, displays those products in the Product category in a two-column view (see Figure 15).

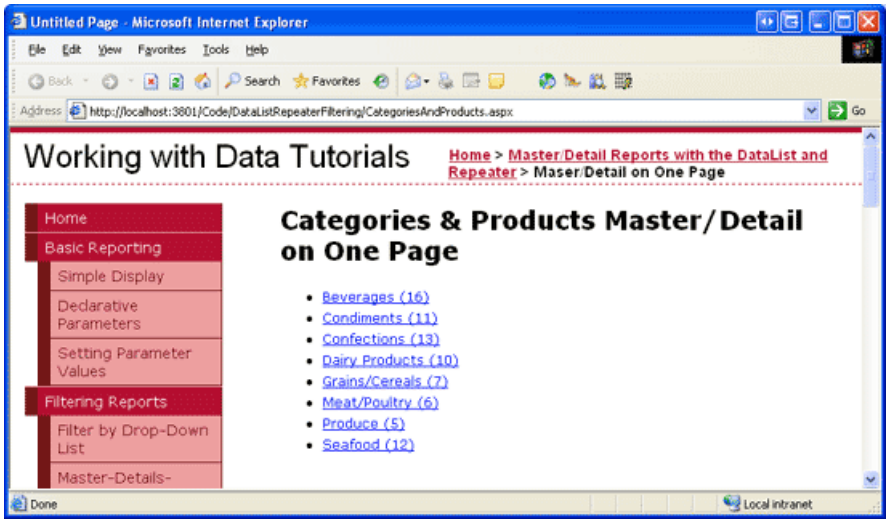


Figure 14: No Products are Displayed When First Visiting the Page (Click to view full-size image)

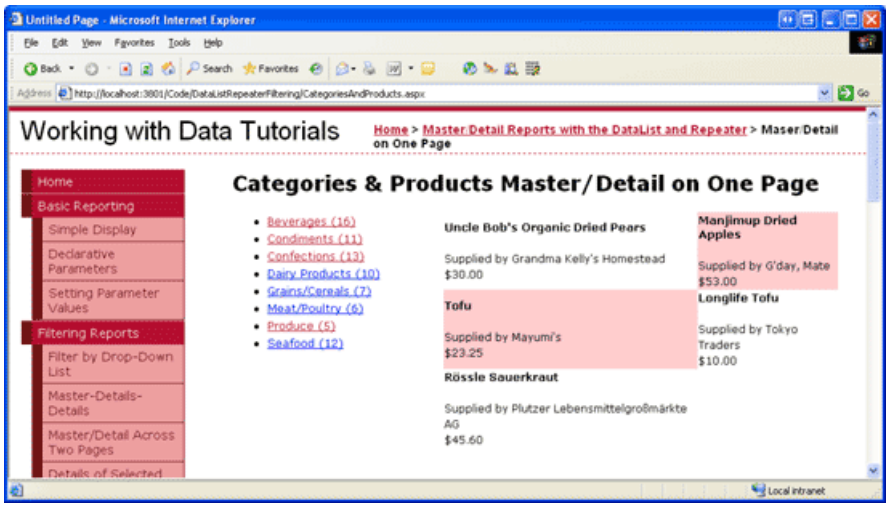


Figure 15: Clicking the Produce Category Lists the Matching Products to the Right (Click to view full-size image)

Summary

As we saw in this tutorial and the preceding one, master/detail reports can be spread out across two pages, or consolidated on one. Displaying a master/details report on a single page, however, introduces some challenges on how best to layout the master and details records on the page. In the *Master/Detail Using a Selectable Master GridView with a Details DetailsView* tutorial we had the details records appear above the master records; in this tutorial we used CSS techniques to have the master records float to the left of the details.

Along with displaying master/details reports, we also had the opportunity to explore how to retrieve the number of products associated with each category as well as how to perform server-side logic when a LinkButton (or Button or ImageButton) is clicked from within a Repeater.

This tutorial completes our examination of master/detail reports with the DataList and Repeater. Our next set of tutorials will illustrate how to add editing and deleting capabilities to the DataList control.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Floatutorial a tutorial on floating CSS elements with CSS
- CSS Positioning more information on positioning elements with CSS
- Laying Out Content with HTML using `<table>` s and other HTML elements for positioning

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Zack Jones. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Editing and Deleting Data Through the DataList

2/20/2020 • 2 minutes to read • Edit Online

These tutorials show how to tweak the DataList control so that it lets uses modify data.

- Overview of Editing and Deleting Data in the DataList (C#)
- Performing Batch Updates (C#)
- Handling BLL- and DAL-Level Exceptions (C#)
- Adding Validation Controls to the DataList's Editing Interface (C#)
- Customizing the DataList's Editing Interface (C#)
- Overview of Editing and Deleting Data in the DataList (VB)
- Performing Batch Updates (VB)
- Handling BLL- and DAL-Level Exceptions (VB)
- Adding Validation Controls to the DataList's Editing Interface (VB)
- Customizing the DataList's Editing Interface (VB)

An Overview of Editing and Deleting Data in the DataList (C#)

2/20/2020 • 23 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> While the DataList lacks built-in editing and deleting capabilities, in this tutorial we'll see how to create a DataList that supports editing and deleting of its underlying data.

Introduction

In the An Overview of Inserting, Updating, and Deleting Data tutorial we looked at how to insert, update, and delete data using the application architecture, an ObjectDataSource, and the GridView, DetailsView, and FormView controls. With the ObjectDataSource and these three data Web controls, implementing simple data modification interfaces was a snap and involved merely ticking a checkbox from a smart tag. No code needed to be written.

Unfortunately, the DataList lacks the built-in editing and deleting capabilities inherent in the GridView control. This missing functionality is due in part to the fact that the DataList is a relic from the previous version of ASP.NET, when declarative data source controls and code-free data modification pages were unavailable. While the DataList in ASP.NET 2.0 does not offer the same out of the box data modification capabilities as the GridView, we can use ASP.NET 1.x techniques to include such functionality. This approach requires a bit of code but, as we'll see in this tutorial, the DataList has some events and properties in place to aid in this process.

In this tutorial we'll see how to create a DataList that supports editing and deleting of its underlying data. Future tutorials will examine more advanced editing and deleting scenarios, including input field validation, gracefully handling exceptions raised from the Data Access or Business Logic Layers, and so on.

> **NOTE**
> Like the DataList, the Repeater control lacks the out of the box functionality for inserting, updating, or deleting. While such functionality can be added, the DataList includes properties and events not found in the Repeater that simplify adding such capabilities. Therefore, this tutorial and future ones that look at editing and deleting will focus strictly on the DataList.

Step 1: Creating the Editing and Deleting Tutorials Web Pages

Before we start exploring how to update and delete data from a DataList, let s first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial and the next several ones. Start by adding a new folder named `EditDeleteDataList`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `Basics.aspx`
- `BatchUpdate.aspx`
- `ErrorHandling.aspx`
- `UIValidation.aspx`
- `CustomizedUI.aspx`

- `OptimisticConcurrency.aspx`
- `ConfirmationOnDelete.aspx`
- `UserLevelAccess.aspx`

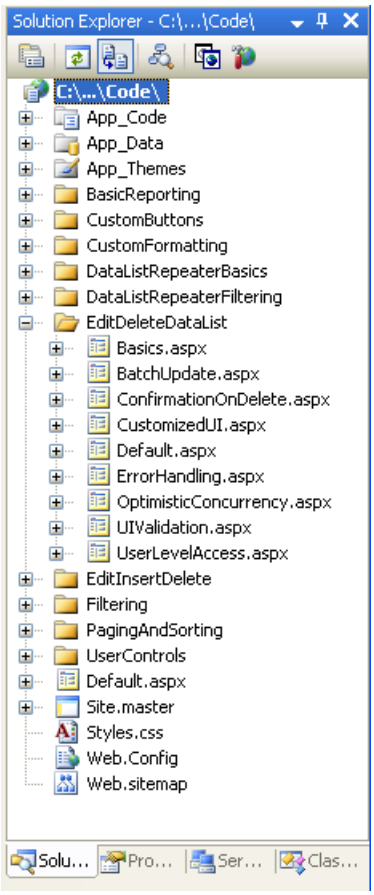


Figure 1: Add the ASP.NET Pages for the Tutorials

Like in the other folders, `Default.aspx` in the `EditDeleteDataList` folder lists the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

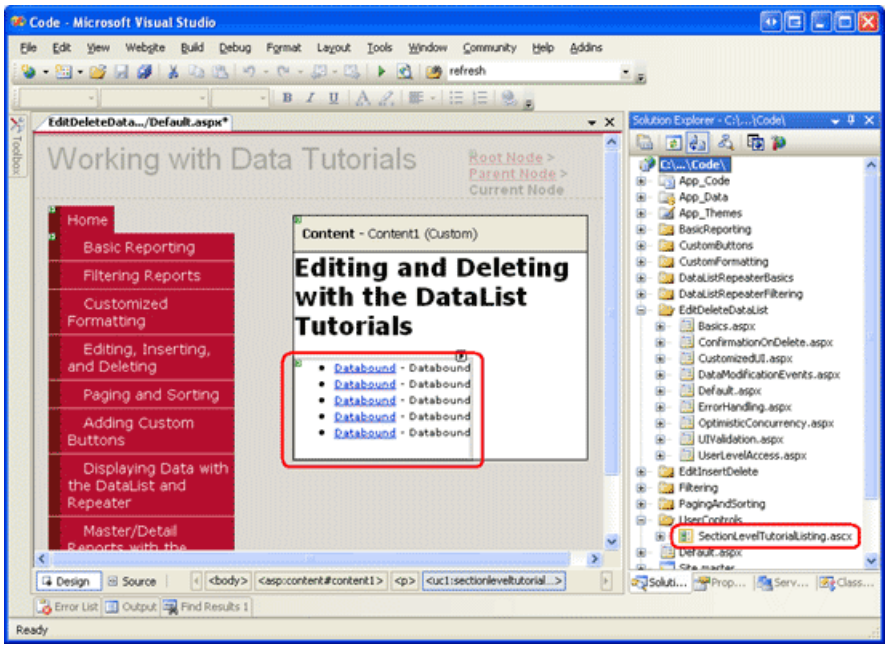


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add the pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the

Master/Detail Reports with the DataList and Repeater `<siteMapNode>`:

```
<siteMapNode
    title="Editing and Deleting with the DataList"
    description="Samples of Reports that Provide Editing and Deleting Capabilities"
    url="~/EditDeleteDataList/Default.aspx" >
    <siteMapNode
        title="Basics"
        description="Examines the basics of editing and deleting with the
                     DataList control."
        url="~/EditDeleteDataList/Basics.aspx" />
    <siteMapNode
        title="Batch Update"
        description="Examines how to update multiple records at once in a
                     fully-editable DataList."
        url="~/EditDeleteDataList/BatchUpdate.aspx" />
    <siteMapNode
        title="Error Handling"
        description="Learn how to gracefully handle exceptions raised during the
                     data modification workflow."
        url="~/EditDeleteDataList/ErrorHandling.aspx" />
    <siteMapNode
        title="Adding Data Entry Validation"
        description="Help prevent data entry errors by providing validation."
        url="~/EditDeleteDataList/UIValidation.aspx" />
    <siteMapNode
        title="Customize the User Interface"
        description="Customize the editing user interfaces."
        url="~/EditDeleteDataList/CustomizedUI.aspx" />
    <siteMapNode
        title="Optimistic Concurrency"
        description="Learn how to help prevent simultaneous users from
                     overwritting one another s changes."
        url="~/EditDeleteDataList/OptimisticConcurrency.aspx" />
    <siteMapNode
        title="Confirm On Delete"
        description="Prompt a user for confirmation when deleting a record."
        url="~/EditDeleteDataList/ConfirmationOnDelete.aspx" />
    <siteMapNode
        title="Limit Capabilities Based on User"
        description="Learn how to limit the data modification functionality
                     based on the user s role or permissions."
        url="~/EditDeleteDataList/UserLevelAccess.aspx" />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the DataList editing and deleting tutorials.

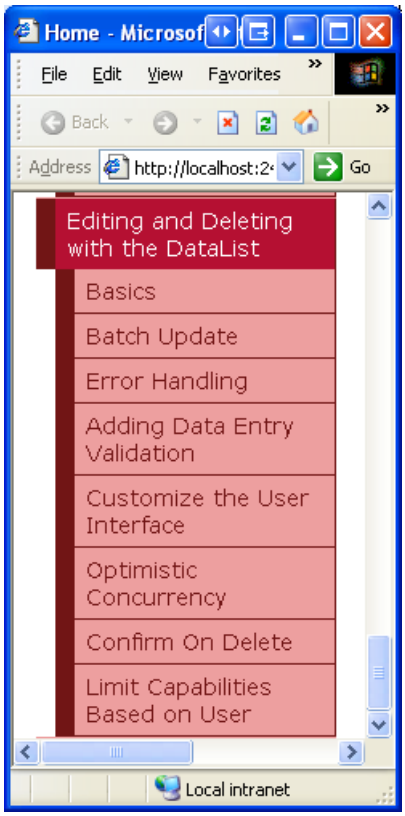


Figure 3: The Site Map Now Includes Entries for the DataList Editing and Deleting Tutorials

Step 2: Examining Techniques for Updating and Deleting Data

Editing and deleting data with the GridView is so easy because, underneath the covers, the GridView and ObjectDataSource work in concert. As discussed in the Examining the Events Associated with Inserting, Updating, and Deleting tutorial, when a row s Update button is clicked, the GridView automatically assigns its fields that used two-way databinding to the `UpdateParameters` collection of its ObjectDataSource and then invokes that ObjectDataSource s `Update()` method.

Sadly, the DataList does not provide any of this built-in functionality. It is our responsibility to ensure that the user s values are assigned to the ObjectDataSource s parameters and that its `Update()` method is called. To aid us in this endeavor, the DataList provides the following properties and events:

- **The** `DataKeyField` **property** when updating or deleting, we need to be able to uniquely identify each item in the DataList. Set this property to the primary key field of the displayed data. Doing so will populate the DataList s `DataKeys` collection with the specified `DataKeyField` value for each DataList item.
- **The** `EditCommand` **event** fires when a Button, LinkButton, or ImageButton whose `CommandName` property is set to Edit is clicked.
- **The** `CancelCommand` **event** fires when a Button, LinkButton, or ImageButton whose `CommandName` property is set to Cancel is clicked.
- **The** `UpdateCommand` **event** fires when a Button, LinkButton, or ImageButton whose `CommandName` property is set to Update is clicked.
- **The** `DeleteCommand` **event** fires when a Button, LinkButton, or ImageButton whose `CommandName` property is set to Delete is clicked.

Using these properties and events, there are four approaches we can use to update and delete data from the DataList:

1. **Using ASP.NET 1.x Techniques** the DataList existed prior to ASP.NET 2.0 and ObjectDataSources and was able to update and delete data entirely through programmatic means. This technique ditches the ObjectDataSource altogether and requires that we bind the data to the DataList directly from the Business

Logic Layer, both in retrieving the data to display and when updating or deleting a record.

2. **Using a Single ObjectDataSource Control on the Page for Selecting, Updating, and Deleting** while the DataList lacks the GridView s inherent editing and deleting capabilities, there s no reason we can t add them in ourselves. With this approach, we use an ObjectDataSource just like in the GridView examples, but must create an event handler for the DataList s `UpdateCommand` event where we set the ObjectDataSource s parameters and call its `Update()` method.
3. **Using an ObjectDataSource Control for Selecting, but Updating and Deleting Directly Against the BLL** when using option 2, we need to write a bit of code in the `UpdateCommand` event, assigning parameter values and so on. Instead, we can stick with using the ObjectDataSource for selecting, but make the updating and deleting calls directly against the BLL (like with option 1). In my opinion, updating data by interfacing directly with the BLL leads to more readable code than assigning the ObjectDataSource s `UpdateParameters` and calling its `Update()` method.
4. **Using Declarative Means through Multiple ObjectDataSources** the previous three approaches all require a bit of code. If you d rather keep using as much declarative syntax as possible, a final option is to include multiple ObjectDataSources on the page. The first ObjectDataSource retrieves the data from the BLL and binds it to the DataList. For updating, another ObjectDataSource is added, but added directly within the DataList s `EditItemTemplate`. To include deleting support, yet another ObjectDataSource would be needed in the `ItemTemplate`. With this approach, these embedded ObjectDataSource s use `ControlParameters` to declaratively bind the ObjectDataSource s parameters to the user input controls (rather than having to specify them programmatically in the DataList s `UpdateCommand` event handler). This approach still requires a bit of code we need to call the embedded ObjectDataSource s `Update()` or `Delete()` command but requires far less than with the other three approaches. The downside here is that the multiple ObjectDataSources do clutter up the page, detracting from the overall readability.

If forced to only ever use one of these approaches, I d choose option 1 because it provides the most flexibility and because the DataList was originally designed to accommodate this pattern. While the DataList was extended to work with the ASP.NET 2.0 data source controls, it does not have all of the extensibility points or features of the official ASP.NET 2.0 data Web controls (the GridView, DetailsView, and FormView). Options 2 through 4 are not without merit, though.

This and the future editing and deleting tutorials will use an ObjectDataSource for retrieving the data to display and direct calls to the BLL to update and delete data (option 3).

Step 3: Adding the DataList and Configuring its ObjectDataSource

In this tutorial we will create a DataList that lists product information and, for each product, provides the user the ability to edit the name and price and to delete the product altogether. In particular, we will retrieve the records to display using an ObjectDataSource, but perform the update and delete actions by interfacing directly with the BLL. Before we worry about implementing the editing and deleting capabilities to the DataList, let s first get the page to display the products in a read-only interface. Since we ve examined these steps in previous tutorials, I'll proceed through them quickly.

Start by opening the `Basics.aspx` page in the `EditDeleteDataList` folder and, from the Design view, add a DataList to the page. Next, from the DataList s smart tag, create a new ObjectDataSource. Since we are working with product data, configure it to use the `ProductsBLL` class. To retrieve *all* products, choose the `GetProducts()` method in the SELECT tab.

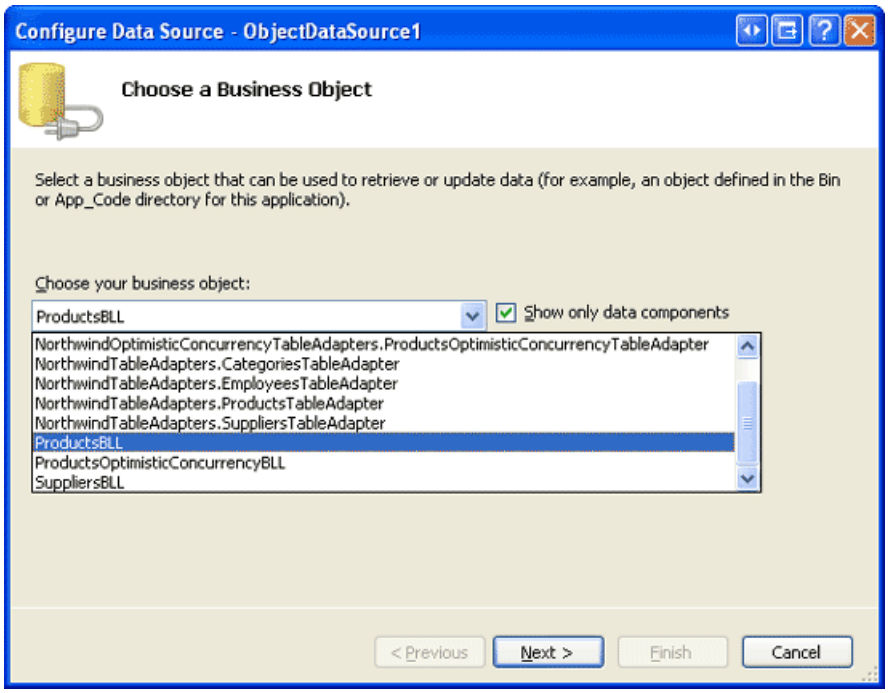


Figure 4: Configure the ObjectDataSource to Use the `ProductsBLL` Class (Click to view full-size image)

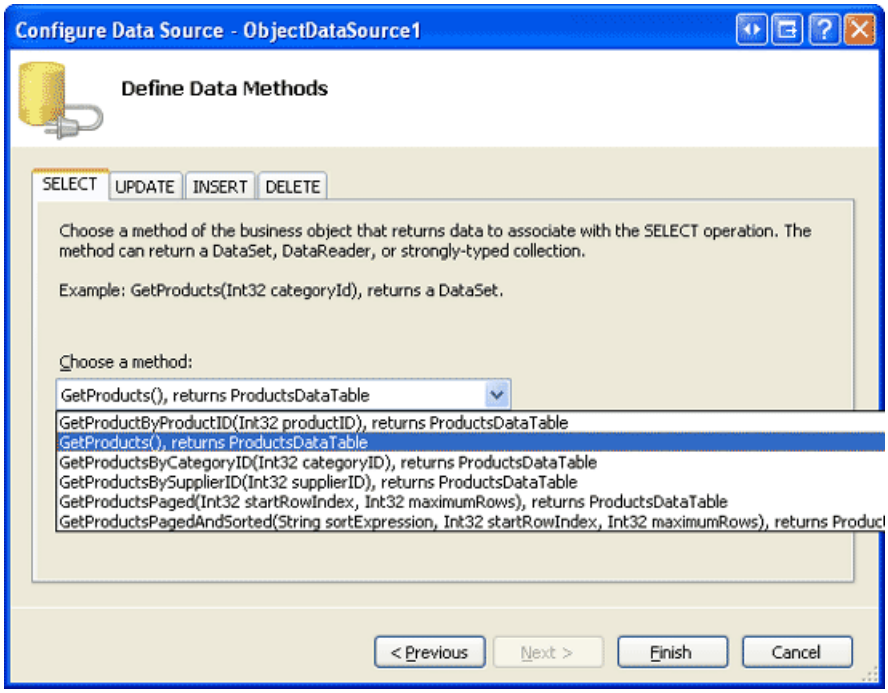


Figure 5: Return the Product Information Using the `GetProducts()` Method (Click to view full-size image)

The DataList, like the GridView, is not designed for inserting new data; therefore, select the (None) option from the drop-down list in the INSERT tab. Also choose (None) for the UPDATE and DELETE tabs since the updates and deletes will be performed programmatically through the BLL.

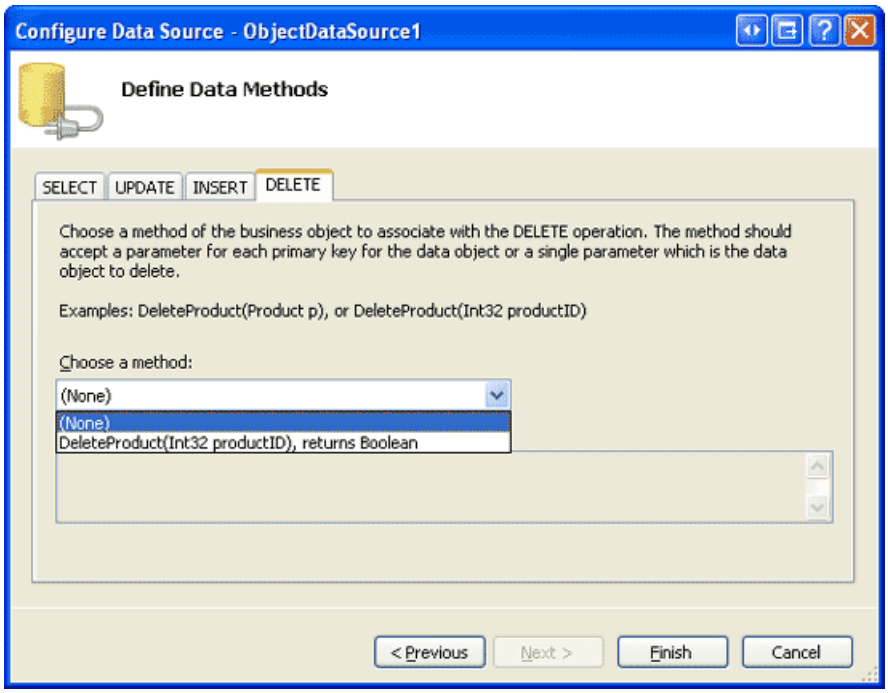


Figure 6: Confirm that the Drop-Down Lists in the ObjectDataSource s INSERT, UPDATE, and DELETE Tabs are Set to (None) (Click to view full-size image)

After configuring the ObjectDataSource, click Finish, returning to the Designer. As we ve seen in past examples, when completing the ObjectDataSource configuration, Visual Studio automatically creates an `ItemTemplate` for the DropDownList, displaying each of the data fields. Replace this `ItemTemplate` with one that displays only the product s name and price. Also, set the `RepeatColumns` property to 2.

> **NOTE**
> As discussed in the *Overview of Inserting, Updating, and Deleting Data* tutorial, when modifying data using the ObjectDataSource our architecture requires that we remove the `OldValuesParameterFormatString` property from the ObjectDataSource s declarative markup (or reset it to its default value, `{0}`). In this tutorial, however, we are using the ObjectDataSource only for retrieving data. Therefore, we do not need to modify the ObjectDataSource s `OldValuesParameterFormatString` property value (although it doesn t hurt to do so).

After replacing the default DataList `ItemTemplate` with a customized one, the declarative markup on your page should look similar to the following:

```
<asp:DataList ID="DataList1" runat="server" DataKeyField="ProductID"
    DataSourceID="ObjectDataSource1" RepeatColumns="2">
    <ItemTemplate>
        <h5>
            <asp:Label runat="server" ID="ProductNameLabel"
                Text='<%# Eval("ProductName") %>'></asp:Label>
        </h5>
        Price: <asp:Label runat="server" ID="Label1"
                    Text='<%# Eval("UnitPrice", "{0:C}") %>' />
        <br />
        <br />
    </ItemTemplate>
</asp:DataList>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    SelectMethod="GetProducts" TypeName="ProductsBLL"
    OldValuesParameterFormatString="original_{0}">
</asp:ObjectDataSource>
```

Take a moment to view our progress through a browser. As Figure 7 shows, the DataList displays the product

name and unit price for each product in two columns.

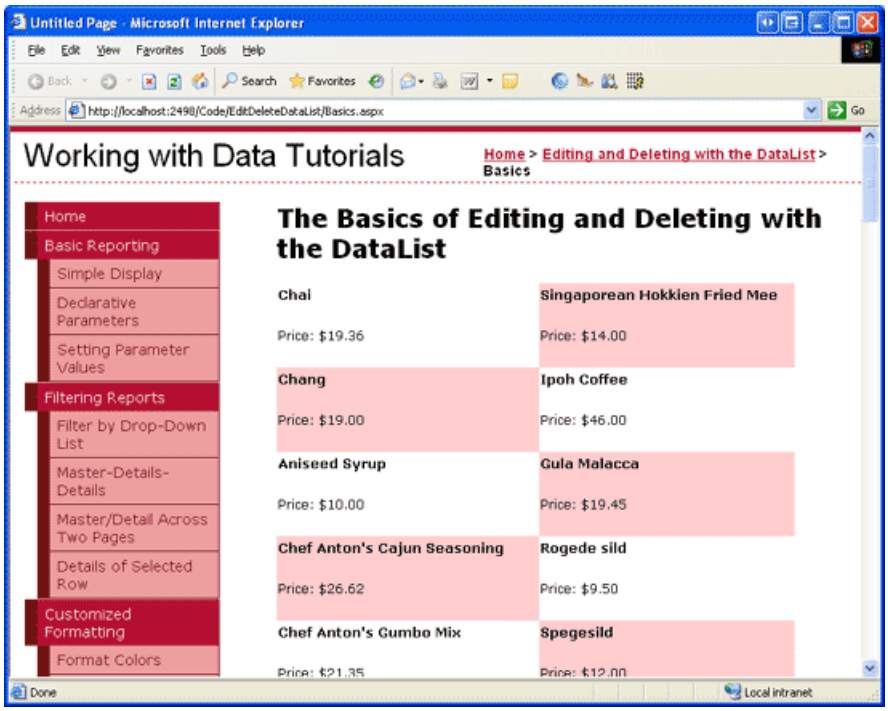


Figure 7: The Products Names and Prices are Displayed in a Two-Column DataList (Click to view full-size image)

> **NOTE**
>
> The DataList has a number of properties that are required for the updating and deleting process, and these values are stored in view state. Therefore, when building a DataList that supports editing or deleting data, it is essential that the DataList s view state be enabled.
>
> The astute reader may recall that we were able to disable view state when creating editable GridViews, DetailsViews, and FormViews. This is because ASP.NET 2.0 Web controls can include *control state*, which is state persisted across postbacks like view state, but deemed essential.

Disabling view state in the GridView merely omits trivial state information, but maintains the control state (which includes the state necessary for editing and deleting). The DataList, having been created in the ASP.NET 1.x timeframe, does not utilize control state and therefore must have view state enabled. See Control State vs. View State for more information on the purpose of control state and how it differs from view state.

Step 4: Adding an Editing User Interface

The GridView control is composed of a collection of fields (BoundFields, CheckBoxFields, TemplateFields, and so on). These fields can adjust their rendered markup depending on their mode. For example, when in read-only mode, a BoundField displays its data field value as text; when in edit mode, it renders a TextBox Web control whose `Text` property is assigned the data field value.

The DataList, on the other hand, renders its items using templates. Read-only items are rendered using the `ItemTemplate` whereas items in edit mode are rendered via the `EditItemTemplate`. At this point, our DataList has only an `ItemTemplate`. To support item-level editing functionality we need to add an `EditItemTemplate` that contains the markup to be displayed for the editable item. For this tutorial, we'll use TextBox Web controls for editing the product s name and unit price.

The `EditItemTemplate` can be created either declaratively or through the Designer (by selecting the Edit Templates option from the DataList s smart tag). To use the Edit Templates option, first click the Edit Templates link in the smart tag and then select the `EditItemTemplate` item from the drop-down list.

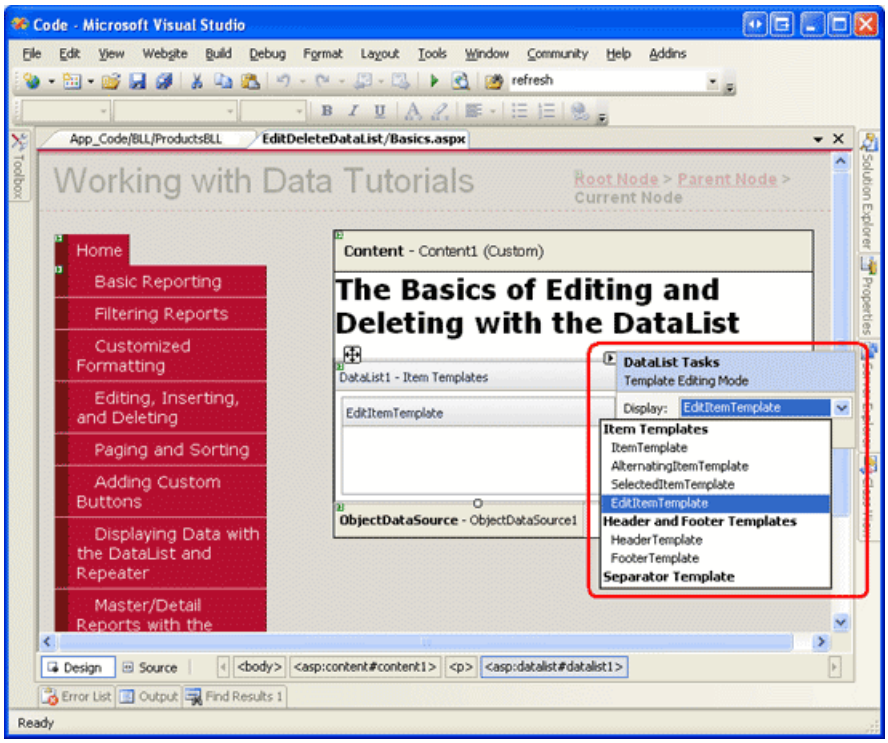


Figure 8: Opt to Work with the DataList s `EditItemTemplate` (Click to view full-size image)

Next, type in Product name: and Price: and then drag two TextBox controls from the Toolbox into the `EditItemTemplate` interface on the Designer. Set the TextBoxes `ID` properties to `ProductName` and `UnitPrice`.

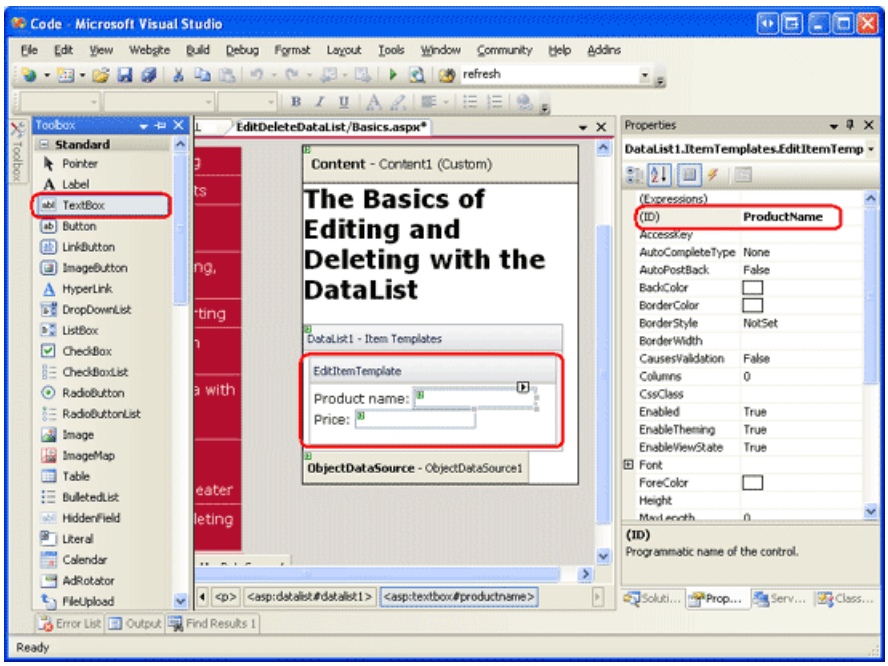


Figure 9: Add a TextBox for the Product s Name and Price (Click to view full-size image)

We need to bind the corresponding product data field values to the `Text` properties of the two TextBoxes. From the TextBoxes smart tags, click on the Edit DataBindings link and then associate the appropriate data field with the `Text` property, as shown in Figure 10.

> **NOTE**
>
> When binding the `UnitPrice` data field to the price TextBox s `Text` field, you may format it as a currency value (`{0:C}`), a general number (`{0:N}`), or leave it unformatted.

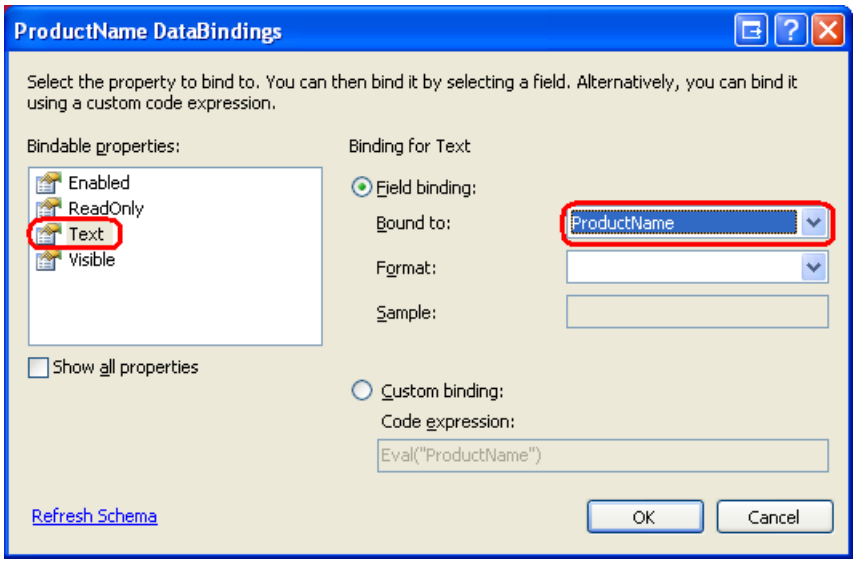


Figure 10: Bind the `ProductName` and `UnitPrice` Data Fields to the `Text` Properties of the TextBoxes

Notice how the Edit DataBindings dialog box in Figure 10 does *not* include the Two-way databinding checkbox that is present when editing a TemplateField in the GridView or DetailsView, or a template in the FormView. The two-way databinding feature allowed the value entered into the input Web control to be automatically assigned to the corresponding ObjectDataSource s `InsertParameters` or `UpdateParameters` when inserting or updating data. The DataList does not support two-way databinding as we'll see later on in this tutorial, after the user makes her changes and is ready to update the data, we will need to programmatically access these TextBoxes `Text` properties and pass in their values to the appropriate `UpdateProduct` method in the `ProductsBLL` class.

Finally, we need to add Update and Cancel buttons to the `EditItemTemplate`. As we saw in the Master/Detail Using a Bulleted List of Master Records with a Details DataList tutorial, when a Button, LinkButton, or ImageButton whose `CommandName` property is set is clicked from within a Repeater or DataList, the Repeater or DataList s `ItemCommand` event is raised. For the DataList, if the `CommandName` property is set to a certain value, an additional event may be raised as well. The special `CommandName` property values include, among others:

- Cancel raises the `CancelCommand` event
- Edit raises the `EditCommand` event
- Update raises the `UpdateCommand` event

Keep in mind that these events are raised *in addition to* the `ItemCommand` event.

Add to the `EditItemTemplate` two Button Web controls, one whose `CommandName` is set to Update and the other s set to Cancel. After adding these two Button Web controls the Designer should look similar to the following:

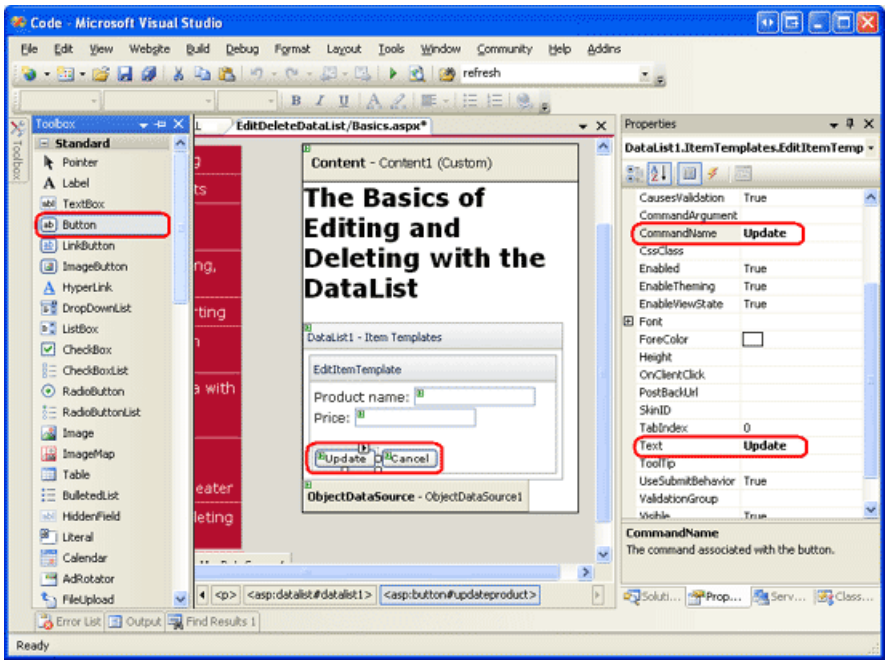


Figure 11: Add Update and Cancel Buttons to the `EditItemTemplate` (Click to view full-size image)

With the `EditItemTemplate` complete your DataList s declarative markup should look similar to the following:

```
<asp:DataList ID="DataList1" runat="server" DataKeyField="ProductID"
    DataSourceID="ObjectDataSource1" RepeatColumns="2">
    <ItemTemplate>
        <h5>
            <asp:Label runat="server" ID="ProductNameLabel"
                Text='<%# Eval("ProductName") %>' />
        </h5>
        Price: <asp:Label runat="server" ID="Label1"
                    Text='<%# Eval("UnitPrice", "{0:C}") %>' />
        <br />
        <br />
    </ItemTemplate>
    <EditItemTemplate>
        Product name:
            <asp:TextBox ID="ProductName" runat="server"
                Text='<%# Eval("ProductName") %>' /><br />
        Price:
            <asp:TextBox ID="UnitPrice" runat="server"
                Text='<%# Eval("UnitPrice", "{0:C}") %>' /><br />
        <br />
        <asp:Button ID="UpdateProduct" runat="server"
            CommandName="Update" Text="Update" />
        <asp:Button ID="CancelUpdate" runat="server"
            CommandName="Cancel" Text="Cancel" />
    </EditItemTemplate>
</asp:DataList>
```

Step 5: Adding the Plumbing to Enter Edit Mode

At this point our DataList has an editing interface defined via its `EditItemTemplate`; however, there s currently no way for a user visiting our page to indicate that he wants to edit a product s information. We need to add an Edit button to each product that, when clicked, renders that DataList item in edit mode. Start by adding an Edit button to the `ItemTemplate`, either through the Designer or declaratively. Be certain to set the Edit button s `CommandName` property to Edit .

After you have added this Edit button, take a moment to view the page through a browser. With this addition, each product listing should include an Edit button.

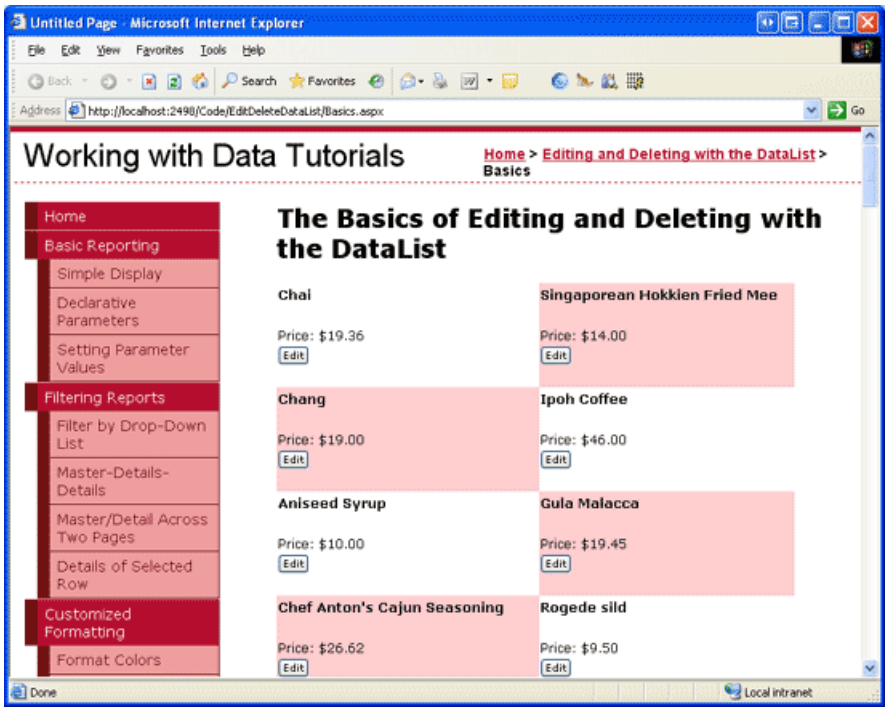


Figure 12: Add Update and Cancel Buttons to the `EditItemTemplate` (Click to view full-size image)

Clicking the button causes a postback, but does *not* bring the product listing into edit mode. To make the product editable, we need to:

1. Set the DataList s `EditItemIndex` property to the index of the `DataListItem` whose Edit button was just clicked.
2. Rebind the data to the DataList. When the DataList is re-rendered, the `DataListItem` whose `ItemIndex` corresponds with the DataList s `EditItemIndex` will render using its `EditItemTemplate`.

Since the DataList s `EditCommand` event is fired when the Edit button is clicked, create an `EditCommand` event handler with the following code:

```
protected void DataList1_EditCommand(object source, DataListCommandEventArgs e)
{
    // Set the DataList's EditItemIndex property to the
    // index of the DataListItem that was clicked
    DataList1.EditItemIndex = e.Item.ItemIndex;
    // Rebind the data to the DataList
    DataList1.DataBind();
}
```

The `EditCommand` event handler is passed in an object of type `DataListCommandEventArgs` as its second input parameter, which includes a reference to the `DataListItem` whose Edit button was clicked (`e.Item`). The event handler first sets the DataList s `EditItemIndex` to the `ItemIndex` of the editable `DataListItem` and then rebinds the data to the DataList by calling the DataList s `DataBind()` method.

After adding this event handler, revisit the page in a browser. Clicking the Edit button now makes the clicked product editable (see Figure 13).

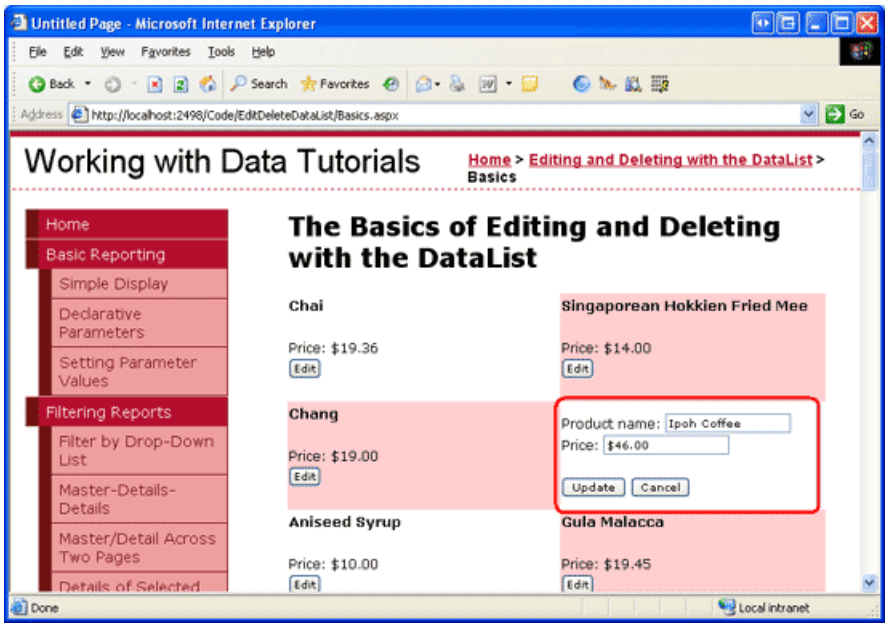


Figure 13: Clicking the Edit Button Makes the Product Editable (Click to view full-size image)

Step 6: Saving the User s Changes

Clicking the edited product s Update or Cancel buttons does nothing at this point; to add this functionality we need to create event handlers for the DataList s `UpdateCommand` and `CancelCommand` events. Start by creating the `CancelCommand` event handler, which will execute when the edited product s Cancel button is clicked and it tasked with returning the DataList to its pre-editing state.

To have the DataList render all of its items in the read-only mode, we need to:

1. Set the DataList s `EditItemIndex` property to the index of a non-existent `DataListItem` index. `-1` is a safe choice, since the `DataListItem` indexes start at `0`.
2. Rebind the data to the DataList. Since no `DataListItem` `ItemIndex` es correspond to the DataList s `EditItemIndex`, the entire DataList will be rendered in a read-only mode.

These steps can be accomplished with the following event handler code:

```
protected void DataList1_CancelCommand(object source, DataListCommandEventArgs e)
{
    // Set the DataList's EditItemIndex property to -1
    DataList1.EditItemIndex = -1;
    // Rebind the data to the DataList
    DataList1.DataBind();
}
```

With this addition, clicking the Cancel button returns the DataList to its pre-editing state.

The last event handler we need to complete is the `UpdateCommand` event handler. This event handler needs to:

1. Programmatically access the user-entered product name and price as well as the edited product s `ProductID`.
2. Initiate the update process by calling the appropriate `UpdateProduct` overload in the `ProductsBLL` class.
3. Set the DataList s `EditItemIndex` property to the index of a non-existent `DataListItem` index. `-1` is a safe choice, since the `DataListItem` indexes start at `0`.
4. Rebind the data to the DataList. Since no `DataListItem` `ItemIndex` es correspond to the DataList s `EditItemIndex`, the entire DataList will be rendered in a read-only mode.

Steps 1 and 2 are responsible for saving the user s changes; steps 3 and 4 return the DataList to its pre-editing

state after the changes have been saved and are identical to the steps performed in the `CancelCommand` event handler.

To get the updated product name and price, we need to use the `FindControl` method to programmatically reference the TextBox Web controls within the `EditItemTemplate`. We also need to get the edited product s `ProductID` value. When we initially bound the ObjectDataSource to the DataList, Visual Studio assigned the DataList s `DataKeyField` property to the primary key value from the data source (`ProductID`). This value can then be retrieved from the DataList s `DataKeys` collection. Take a moment to ensure that the `DataKeyField` property is indeed set to `ProductID`.

The following code implements the four steps:

```
protected void DataList1_UpdateCommand(object source, DataListCommandEventArgs e)
{
    // Read in the ProductID from the DataKeys collection
    int productID = Convert.ToInt32(DataList1.DataKeys[e.Item.ItemIndex]);
    // Read in the product name and price values
    TextBox productName = (TextBox)e.Item.FindControl("ProductName");
    TextBox unitPrice = (TextBox)e.Item.FindControl("UnitPrice");
    string productNameValue = null;
    if (productName.Text.Trim().Length > 0)
        productNameValue = productName.Text.Trim();
    decimal? unitPriceValue = null;
    if (unitPrice.Text.Trim().Length > 0)
        unitPriceValue = Decimal.Parse(unitPrice.Text.Trim(),
            System.Globalization.NumberStyles.Currency);
    // Call the ProductsBLL's UpdateProduct method...
    ProductsBLL productsAPI = new ProductsBLL();
    productsAPI.UpdateProduct(productNameValue, unitPriceValue, productID);
    // Revert the DataList back to its pre-editing state
    DataList1.EditItemIndex = -1;
    DataList1.DataBind();
}
```

The event handler starts by reading in the edited product s `ProductID` from the `DataKeys` collection. Next, the two TextBoxes in the `EditItemTemplate` are referenced and their `Text` properties stored in local variables, `productNameValue` and `unitPriceValue`. We use the `Decimal.Parse()` method to read the value from the `UnitPrice` TextBox so that if the value entered has a currency symbol, it can still be correctly converted into a `Decimal` value.

> **NOTE**
> The values from the `ProductName` and `UnitPrice` TextBoxes are only assigned to the productNameValue and unitPriceValue variables if the TextBoxes Text properties have a value specified. Otherwise, a value of `Nothing` is used for the variables, which has the effect of updating the data with a database `NULL` value. That is, our code treats converts empty strings to database `NULL` values, which is the default behavior of the editing interface in the GridView, DetailsView, and FormView controls.

After reading the values, the `ProductsBLL` class s `UpdateProduct` method is called, passing in the product s name, price, and `ProductID`. The event handler completes by returning the DataList to its pre-editing state using the exact same logic as in the `CancelCommand` event handler.

With the `EditCommand`, `CancelCommand`, and `UpdateCommand` event handlers complete, a visitor can edit the name and price of a product. Figures 14-16 show this editing workflow in action.

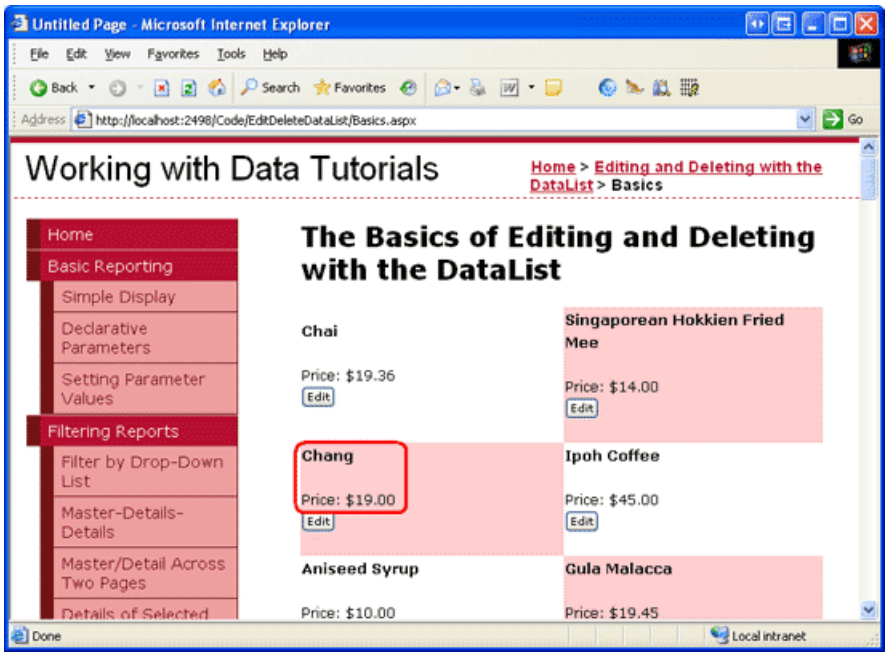


Figure 14: When First Visiting the Page, All Products are in Read-Only Mode (Click to view full-size image)

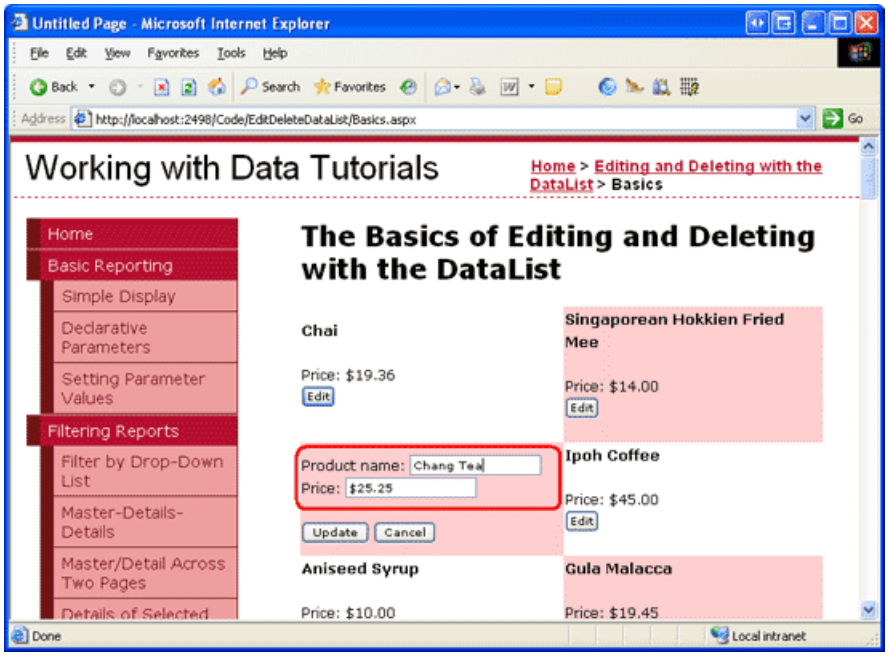


Figure 15: To Update a Product s Name or Price, Click the Edit Button (Click to view full-size image)

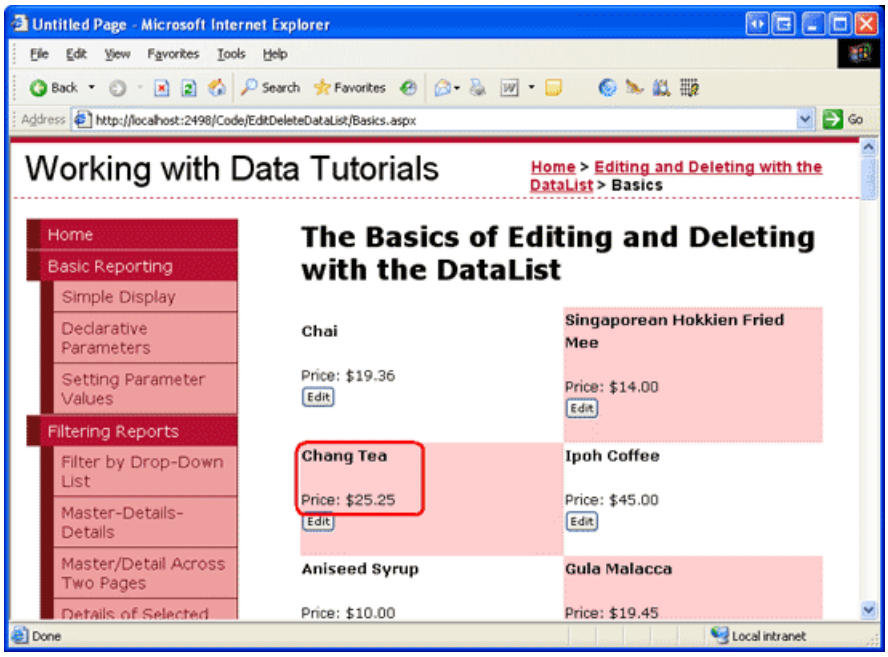


Figure 16: After Changing the Value, Click Update to Return to the Read-Only Mode (Click to view full-size image)

Step 7: Adding Delete Capabilities

The steps for adding delete capabilities to a DataList are similar to those for adding editing capabilities. In short, we need to add a Delete button to the `ItemTemplate` that, when clicked:

1. Reads in the corresponding product s `ProductID` via the `DataKeys` collection.
2. Performs the delete by calling the `ProductsBLL` class s `DeleteProduct` method.
3. Rebinds the data to the DataList.

Let s start by adding a Delete button to the `ItemTemplate`.

When clicked, a Button whose `CommandName` is Edit, Update, or Cancel raises the DataList s `ItemCommand` event along with an additional event (for example, when using Edit the `EditCommand` event is raised as well). Similarly, any Button, LinkButton, or ImageButton in the DataList whose `CommandName` property is set to Delete causes the `DeleteCommand` event to fire (along with `ItemCommand`).

Add a Delete button next to the Edit button in the `ItemTemplate`, setting its `CommandName` property to Delete. After adding this Button control your DataList s `ItemTemplate` declarative syntax should look like:

```
<ItemTemplate>
    <h5>
        <asp:Label runat="server" ID="ProductNameLabel"
            Text='<%# Eval("ProductName") %>' />
    </h5>
    Price: <asp:Label runat="server" ID="Label1"
                Text='<%# Eval("UnitPrice", "{0:C}") %>' />
    <br />
    <asp:Button runat="server" id="EditProduct" CommandName="Edit"
        Text="Edit" />

    <asp:Button runat="server" id="DeleteProduct" CommandName="Delete"
        Text="Delete" />
    <br />
    <br />
</ItemTemplate>
```

Next, create an event handler for the DataList s `DeleteCommand` event, using the following code:

```
protected void DataList1_DeleteCommand(object source, DataListCommandEventArgs e)
{
    // Read in the ProductID from the DataKeys collection
    int productID = Convert.ToInt32(DataList1.DataKeys[e.Item.ItemIndex]);
    // Delete the data
    ProductsBLL productsAPI = new ProductsBLL();
    productsAPI.DeleteProduct(productID);
    // Rebind the data to the DataList
    DataList1.DataBind();
}
```

Clicking the Delete button causes a postback and fires the DataList s `DeleteCommand` event. In the event handler, the clicked product s `ProductID` value is accessed from the `DataKeys` collection. Next, the product is deleted by calling the `ProductsBLL` class s `DeleteProduct` method.

After deleting the product, it s important that we rebind the data to the DataList (`DataList1.DataBind()`), otherwise the DataList will continue to show the product that was just deleted.

Summary

While the DataList lacks the point and click editing and deleting support enjoyed by the GridView, with a short bit of code it can be enhanced to include these features. In this tutorial we saw how to create a two-column listing of products that could be deleted and whose name and price could be edited. Adding editing and deleting support is a matter of including the appropriate Web controls in the `ItemTemplate` and `EditItemTemplate`, creating the corresponding event handlers, reading the user-entered and primary key values, and interfacing with the Business Logic Layer.

While we have added basic editing and deleting capabilities to the DataList, it lacks more advanced features. For example, there is no input field validation - if a user enters a price of Too expensive, an exception will be thrown by `Decimal.Parse` when attempting to convert Too expensive into a `Decimal`. Similarly, if there is a problem in updating the data at the Business Logic or Data Access Layers the user will be presented with the standard error screen. Without any sort of confirmation on the Delete button, accidentally deleting a product is all too likely.

In future tutorials we'll see how to improve the editing user experience.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Zack Jones, Ken Pespisa, and Randy Schmidt. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Performing Batch Updates (C#)

2/20/2020 • 11 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> Learn how to create a fully-editable DataList where all of its items are in edit mode and whose values can be saved by clicking an "Update All" button on the page.

Introduction

In the preceding tutorial we examined how to create an item-level DataList. Like the standard editable GridView, each item in the DataList included an Edit button that, when clicked, would make the item editable. While this item-level editing works well for data that is only updated occasionally, certain use case scenarios require the user to edit many records. If a user needs to edit dozens of records and is forced to click Edit, make their changes, and click Update for each one, the amount of clicking can hamper her productivity. In such situations, a better option is to provide a fully-editable DataList, one where *all* of its items are in edit mode and whose values can be edited by clicking an Update All button on the page (see Figure 1).

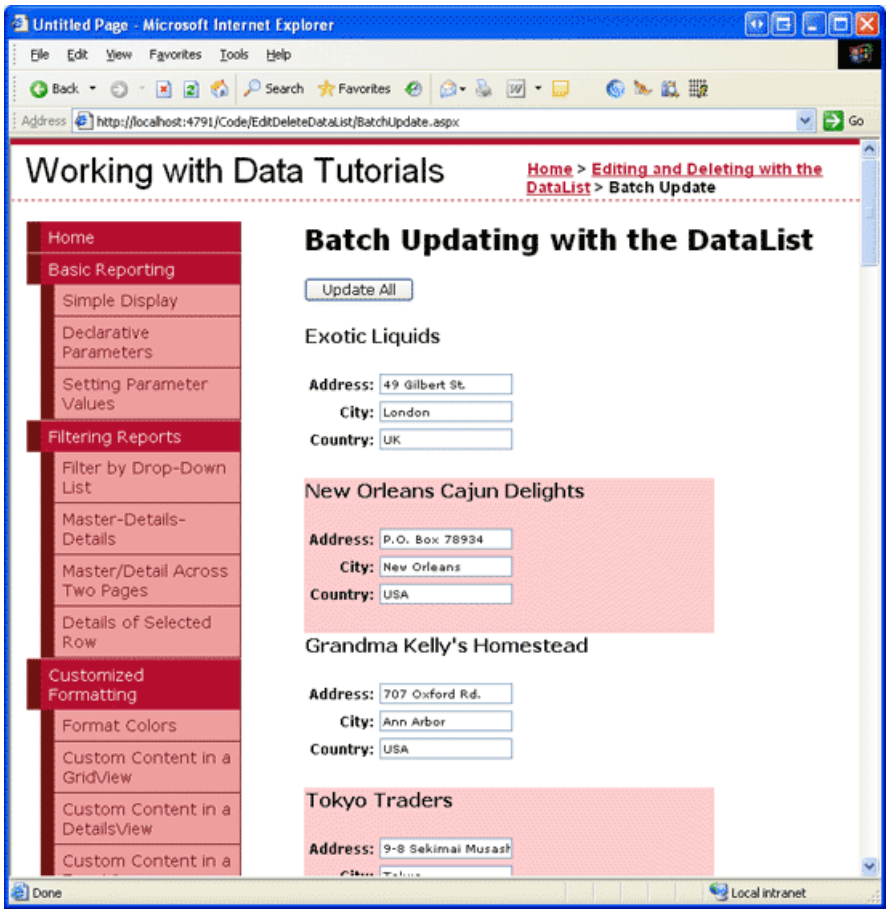


Figure 1: Each Item in a Fully Editable DataList can be Modified (Click to view full-size image)

In this tutorial we'll examine how to enable users to update suppliers address information using a fully editable DataList.

Step 1: Create the Editable User Interface in the DataList s

ItemTemplate

In the preceding tutorial, where we creating a standard, item-level editable DataList, we used two templates:

- `ItemTemplate` contained the read-only user interface (the Label Web controls for displaying each product s name and price).
- `EditItemTemplate` contained the edit mode user interface (the two TextBox Web controls).

The DataList s `EditItemIndex` property dictates what `DataListItem` (if any) is rendered using the `EditItemTemplate`. In particular, the `DataListItem` whose `ItemIndex` value matches the DataList s `EditItemIndex` property is rendered using the `EditItemTemplate`. This model works well when only one item can be edited at a time, but falls apart when creating a fully-editable DataList.

For a fully editable DataList, we want *all* of the `DataListItem` s to render using the editable interface. The simplest way to accomplish this is to define the editable interface in the `ItemTemplate`. For modifying the suppliers address information, the editable interface contains the supplier name as text and then TextBoxes for the address, city, and country values.

Start by opening the `BatchUpdate.aspx` page, add a DataList control, and set its `ID` property to `Suppliers`. From the DataList s smart tag, opt to add a new ObjectDataSource control named `SuppliersDataSource`.

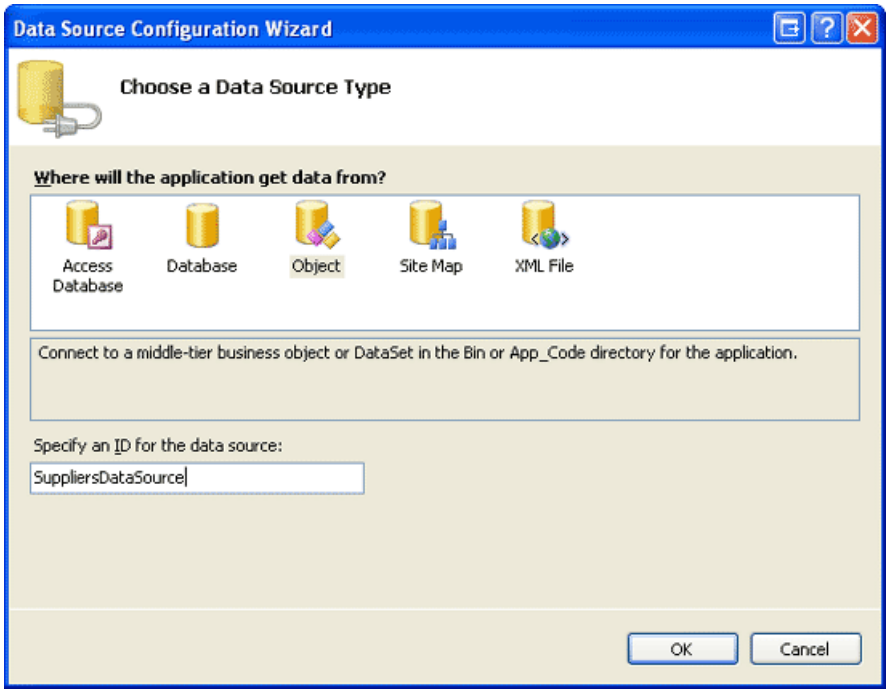


Figure 2: Create a New ObjectDataSource Named `SuppliersDataSource` (Click to view full-size image)

Configure the ObjectDataSource to retrieve data using the `SuppliersBLL` class s `GetSuppliers()` method (see Figure 3). As with the preceding tutorial, rather than updating the supplier information through the ObjectDataSource, we'll work directly with the Business Logic Layer. Therefore, set the drop-down list to (None) in the UPDATE tab (see Figure 4).

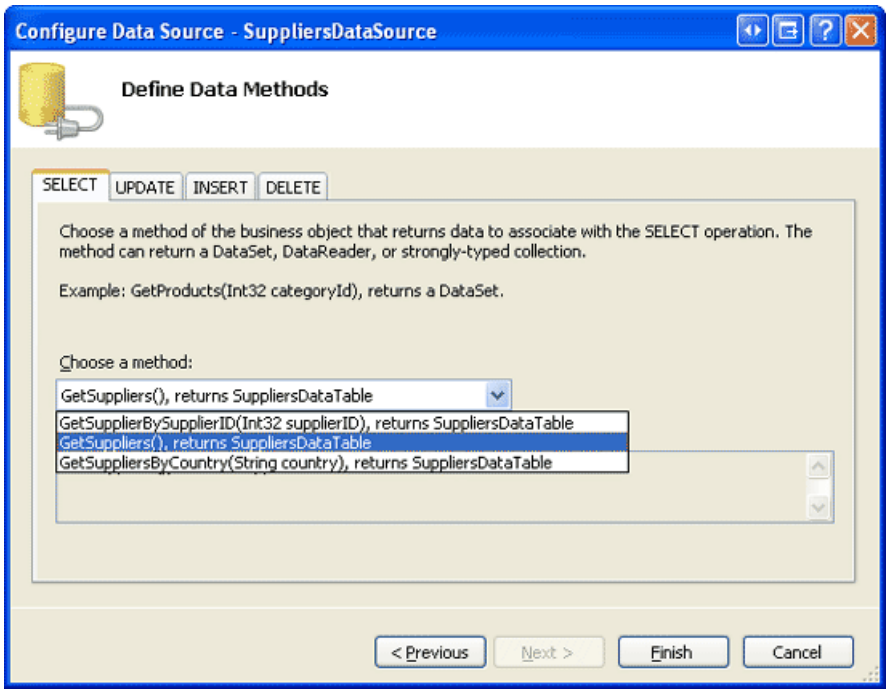


Figure 3: Retrieve Supplier Information Using the `GetSuppliers()` Method (Click to view full-size image)

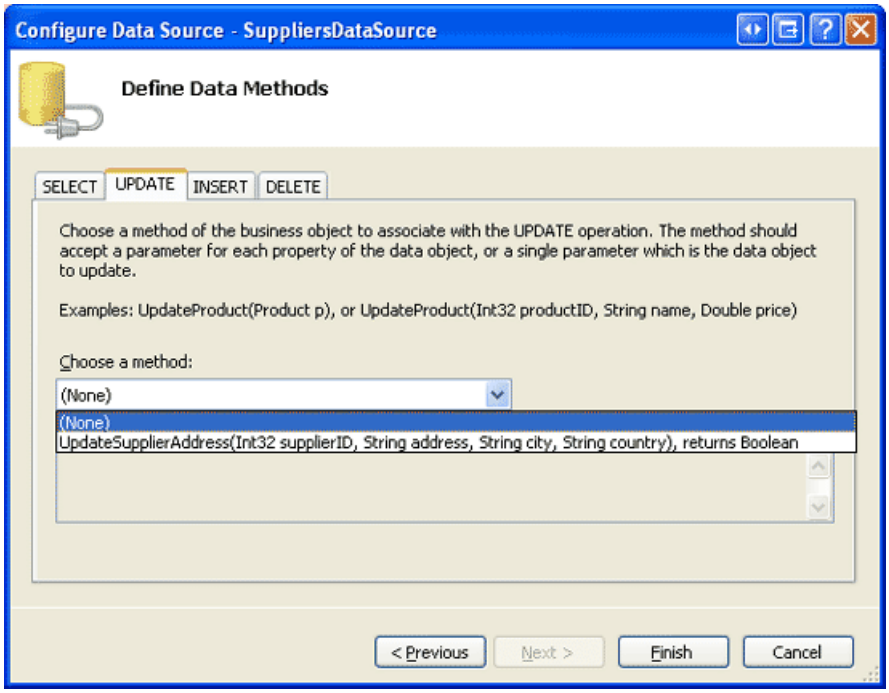


Figure 4: Set the Drop-Down List to (None) in the UPDATE Tab (Click to view full-size image)

After completing the wizard, Visual Studio automatically generates the DataList s `ItemTemplate` to display each data field returned by the data source in a Label Web control. We need to modify this template so that it provides the editing interface instead. The `ItemTemplate` can be customized through the Designer using the Edit Templates option from the DataList s smart tag or directly through the declarative syntax.

Take a moment to create an editing interface that displays the supplier s name as text, but includes TextBoxes for the supplier s address, city, and country values. After making these changes, your page s declarative syntax should look similar to the following:

```
<asp:DataList ID="Suppliers" runat="server" DataKeyField="SupplierID"
    DataSourceID="SuppliersDataSource">
    <ItemTemplate>
        <h4><asp:Label ID="CompanyNameLabel" runat="server"
            Text='<%# Eval("CompanyName") %>' /></h4>
        <table border="0">
            <tr>
                <td class="SupplierPropertyLabel">Address:</td>
                <td class="SupplierPropertyValue">
                    <asp:TextBox ID="Address" runat="server"
                        Text='<%# Eval("Address") %>' />
                </td>
            </tr>
            <tr>
                <td class="SupplierPropertyLabel">City:</td>
                <td class="SupplierPropertyValue">
                    <asp:TextBox ID="City" runat="server"
                        Text='<%# Eval("City") %>' />
                </td>
            </tr>
            <tr>
                <td class="SupplierPropertyLabel">Country:</td>
                <td class="SupplierPropertyValue">
                    <asp:TextBox ID="Country" runat="server"
                        Text='<%# Eval("Country") %>' />
                </td>
            </tr>
        </table>
        <br />
    </ItemTemplate>
</asp:DataList>
<asp:ObjectDataSource ID="SuppliersDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetSuppliers" TypeName="SuppliersBLL">
</asp:ObjectDataSource>
```

NOTE

As with the preceding tutorial, the DataList in this tutorial must have its view state enabled.

In the `ItemTemplate` I m using two new CSS classes, `SupplierPropertyLabel` and `SupplierPropertyValue`, which have been added to the `Styles.css` class and configured to use the same style settings as the `ProductPropertyLabel` and `ProductPropertyValue` CSS classes.

```
.ProductPropertyLabel, .SupplierPropertyLabel
{
    font-weight: bold;
    text-align: right;
}
.ProductPropertyValue, .SupplierPropertyValue
{
    padding-right: 35px;
}
```

After making these changes, visit this page through a browser. As Figure 5 shows, each DataList item displays the supplier name as text and uses TextBoxes to display the address, city, and country.

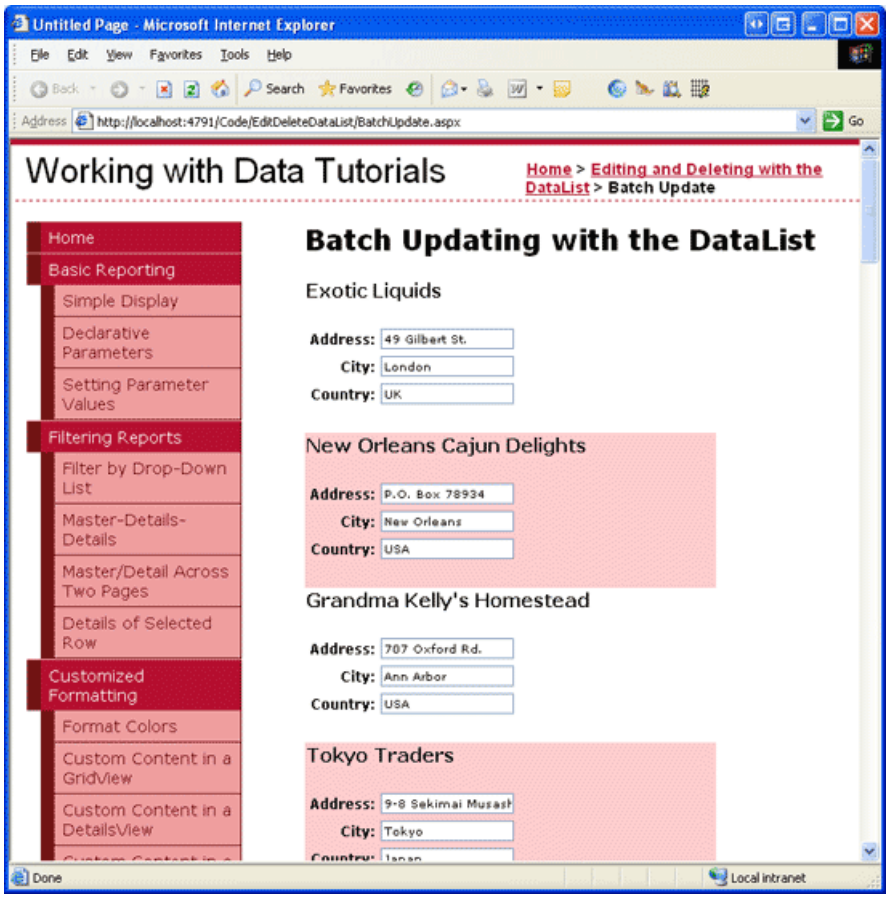


Figure 5: Each Supplier in the DataList is Editable (Click to view full-size image)

Step 2: Adding an Update All Button

While each supplier in Figure 5 has its address, city, and country fields displayed in a TextBox, there currently is no Update button available. Rather than having an Update button per item, with fully editable DataLists there is typically a single Update All button on the page that, when clicked, updates *all* of the records in the DataList. For this tutorial, let s add two Update All buttons - one at the top of the page and one at the bottom (although clicking either button will have the same effect).

Start by adding a Button Web control above the DataList and set its `ID` property to `UpdateAll1`. Next, add the second Button Web control beneath the DataList, setting its `ID` to `UpdateAll2`. Set the `Text` properties for the two Buttons to Update All. Lastly, create event handlers for both Buttons `Click` events. Rather than duplicating the update logic in each of the event handlers, let s refactor that logic to a third method, `UpdateAllSupplierAddresses`, having the event handlers simply invoking this third method.

```
protected void UpdateAll1_Click(object sender, EventArgs e)
{
    UpdateAllSupplierAddresses();
}
protected void UpdateAll2_Click(object sender, EventArgs e)
{
    UpdateAllSupplierAddresses();
}
private void UpdateAllSupplierAddresses()
{
    // TODO: Write code to update _all_ of the supplier addresses in the DataList
}
```

Figure 6 shows the page after the Update All buttons have been added.

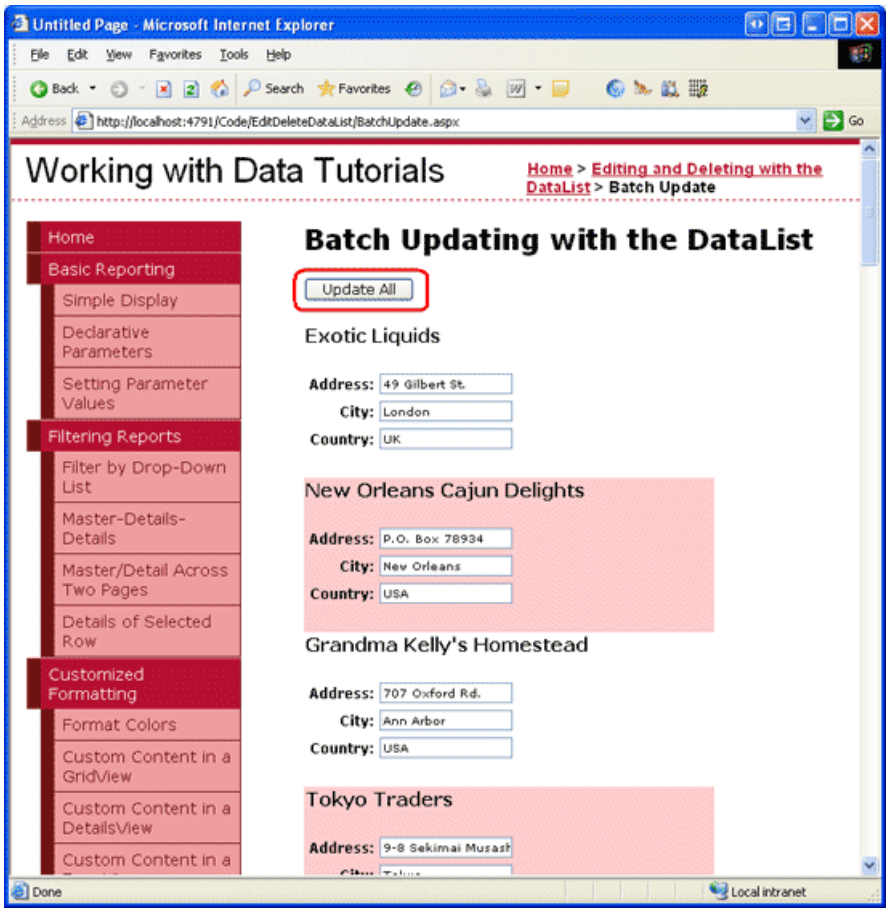


Figure 6: Two Update All Buttons have been Added to the Page (Click to view full-size image)

Step 3: Updating All of the Suppliers Address Information

With all of the DataList s items displaying the editing interface and with the addition of the Update All buttons, all that remains is writing the code to perform the batch update. Specifically, we need to loop through the DataList s items and call the `SuppliersBLL` class s `UpdateSupplierAddress` method for each one.

The collection of `DataListItem` instances that makeup the DataList can be accessed via the DataList s `Items` property. With a reference to a `DataListItem`, we can grab the corresponding `SupplierID` from the `DataKeys` collection and programmatically reference the TextBox Web controls within the `ItemTemplate` as the following code illustrates:

```
private void UpdateAllSupplierAddresses()
{
    // Create an instance of the SuppliersBLL class
    SuppliersBLL suppliersAPI = new SuppliersBLL();
    // Iterate through the DataList's items
    foreach (DataListItem item in Suppliers.Items)
    {
        // Get the supplierID from the DataKeys collection
        int supplierID = Convert.ToInt32(Suppliers.DataKeys[item.ItemIndex]);
        // Read in the user-entered values
        TextBox address = (TextBox)item.FindControl("Address");
        TextBox city = (TextBox)item.FindControl("City");
        TextBox country = (TextBox)item.FindControl("Country");
        string addressValue = null, cityValue = null, countryValue = null;
        if (address.Text.Trim().Length > 0)
            addressValue = address.Text.Trim();
        if (city.Text.Trim().Length > 0)
              cityValue = city.Text.Trim();
        if (country.Text.Trim().Length > 0)
            countryValue = country.Text.Trim();
        // Call the SuppliersBLL class's UpdateSupplierAddress method
        suppliersAPI.UpdateSupplierAddress
            (supplierID, addressValue, cityValue, countryValue);
    }
}
```

When the user clicks one of the Update All buttons, the `UpdateAllSupplierAddresses` method iterates through each `DataListItem` in the `Suppliers` DataList and calls the `SuppliersBLL` class s `UpdateSupplierAddress` method, passing in the corresponding values. A non-entered value for address, city, or country passes is a value of `Nothing` to `UpdateSupplierAddress` (rather than a blank string), which results in a database `NULL` for the underlying record s fields.

> **NOTE**
> As an enhancement, you may want to add a status Label Web control to the page that provides some confirmation message after the batch update is performed.

Updating Only Those Addresses That Have Been Modified

The batch update algorithm used for this tutorial calls the `UpdateSupplierAddress` method for *every* supplier in the DataList, regardless of whether their address information has been changed. While such blind updates aren't usually a performance issue, they can lead to superfluous records if you re auditing changes to the database table. For example, if you use triggers to record all `UPDATE` s to the `Suppliers` table to an auditing table, every time a user clicks the Update All button a new audit record will be created for each supplier in the system, regardless of whether the user made any changes.

The ADO.NET DataTable and DataAdapter classes are designed to support batch updates where only modified, deleted, and new records results in any database communication. Each row in the DataTable has a `RowState` property that indicates whether the row has been added to the DataTable, deleted from it, modified, or remains unchanged. When a DataTable is initially populated, all rows are marked unchanged. Changing the value of any of the row s columns marks the row as modified.

In the `SuppliersBLL` class we update the specified supplier s address information by first reading in the single supplier record into a `SuppliersDataTable` and then set the `Address`, `City`, and `Country` column values using the following code:

```
public bool UpdateSupplierAddress
    (int supplierID, string address, string city, string country)
{
    Northwind.SuppliersDataTable suppliers =
        Adapter.GetSupplierBySupplierID(supplierID);
    if (suppliers.Count == 0)
        // no matching record found, return false
        return false;
    else
    {
        Northwind.SuppliersRow supplier = suppliers[0];
        if (address == null)
            supplier.SetAddressNull();
        else
            supplier.Address = address;
        if (city == null)
            supplier.SetCityNull();
        else
            supplier.City = city;
        if (country == null)
            supplier.SetCountryNull();
        else
            supplier.Country = country;
        // Update the supplier Address-related information
        int rowsAffected = Adapter.Update(supplier);
        // Return true if precisely one row was updated,
        // otherwise false
        return rowsAffected == 1;
    }
}
```

This code naively assigns the passed-in address, city, and country values to the `SuppliersRow` in the `SuppliersDataTable` regardless of whether or not the values have changed. These modifications cause the `SuppliersRow` s `RowState` property to be marked as modified. When the Data Access Layer s `Update` method is called, it sees that the `SupplierRow` has been modified and therefore sends an `UPDATE` command to the database.

Imagine, however, that we added code to this method to only assign the passed-in address, city, and country values if they differ from the `SuppliersRow` s existing values. In the case where the address, city, and country are the same as the existing data, no changes will be made and the `SupplierRow` s `RowState` will be left marked as unchanged. The net result is that when the DAL s `Update` method is called, no database call will be made because the `SuppliersRow` has not been modified.

To enact this change, replace the statements that blindly assign the passed-in address, city, and country values with the following code:

```
// Only assign the values to the SupplierRow's column values if they differ
if (address == null && !supplier.IsAddressNull())
    supplier.SetAddressNull();
else if ((address != null && supplier.IsAddressNull()) ||
         (!supplier.IsAddressNull() &&
         string.Compare(supplier.Address, address) != 0))
    supplier.Address = address;
if (city == null && !supplier.IsCityNull())
    supplier.SetCityNull();
else if ((city != null && supplier.IsCityNull()) ||
         (!supplier.IsCityNull() && string.Compare(supplier.City, city) != 0))
    supplier.City = city;
if (country == null && !supplier.IsCountryNull())
    supplier.SetCountryNull();
else if ((country != null && supplier.IsCountryNull()) ||
         (!supplier.IsCountryNull() &&
         string.Compare(supplier.Country, country) != 0))
    supplier.Country = country;
```

With this added code, the DAL s `Update` method sends an `UPDATE` statement to the database for only those records whose address-related values have changed.

Alternatively, we could keep track of whether there are any differences between the passed-in address fields and the database data and, if there are none, simply bypass the call to the DAL s `Update` method. This approach works well if you re using the DB direct method, since the DB direct method isn t passed a `SuppliersRow` instance whose `RowState` can be checked to determine whether a database call is actually needed.

> **NOTE**
> Each time the `UpdateSupplierAddress` method is invoked, a call is made to the database to retrieve information about the updated record. Then, if there are any changes in data, another call to the database is made to update the table row. This workflow could be optimized by creating an `UpdateSupplierAddress` method overload that accepts an `EmployeesDataTable` instance that has *all* of the changes from the `BatchUpdate.aspx` page. Then, it could make one call to the database to get all of the records from the `Suppliers` table. The two resultsets could then be enumerated and only those records where changes have occurred could be updated.

Summary

In this tutorial we saw how to create a fully editable DataList, allowing a user to quickly modify the address information for multiple suppliers. We started by defining the editing interface a TextBox Web control for the supplier s address, city, and country values in the DataList s `ItemTemplate`. Next, we added Update All buttons above and below the DataList. After a user has made his changes and clicked one of the Update All buttons, the `DataListItem` s are enumerated and a call to the `SuppliersBLL` class s `UpdateSupplierAddress` method is made.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Zack Jones and

Ken Pespisa. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Handling BLL- and DAL-Level Exceptions (C#)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial, we'll see how to tactfully handle exceptions raised during an editable DataList's updating workflow.

Introduction

In the Overview of Editing and Deleting Data in the DataList tutorial, we created a DataList that offered simple editing and deleting capabilities. While fully functional, it was hardly user-friendly, as any error that occurred during the editing or deleting process resulted in an unhandled exception. For example, omitting the product s name or, when editing a product, entering a price value of Very affordable!, throws an exception. Since this exception is not caught in code, it bubbles up to the ASP.NET runtime, which then displays the exception s details in the web page.

As we saw in the Handling BLL- and DAL-Level Exceptions in an ASP.NET Page tutorial, if an exception is raised from the depths of the Business Logic or Data Access Layers, the exception details are returned to the ObjectDataSource and then to the GridView. We saw how to gracefully handle these exceptions by creating `Updated` or `RowUpdated` event handlers for the ObjectDataSource or GridView, checking for an exception, and then indicating that the exception was handled.

Our DataList tutorials, however, aren't using the ObjectDataSource for updating and deleting data. Instead, we are working directly against the BLL. In order to detect exceptions originating from the BLL or DAL, we need to implement exception handling code within the code-behind of our ASP.NET page. In this tutorial, we'll see how to more tactfully handle exceptions raised during an editable DataList s updating workflow.

> **NOTE**
>
> In the *An Overview of Editing and Deleting Data in the DataList* tutorial we discussed different techniques for editing and deleting data from the DataList, Some techniques involved using an ObjectDataSource for updating and deleting. If you employ these techniques, you can handle exceptions from the BLL or DAL through the ObjectDataSource s `Updated` or `Deleted` event handlers.

Step 1: Creating an Editable DataList

Before we worry about handling exceptions that occur during the updating workflow, let s first create an editable DataList. Open the `ErrorHandling.aspx` page in the `EditDeleteDataList` folder, add a DataList to the Designer, set its `ID` property to `Products`, and add a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to use the `ProductsBLL` class s `GetProducts()` method for selecting records; set the drop-down lists in the INSERT, UPDATE, and DELETE tabs to (None).

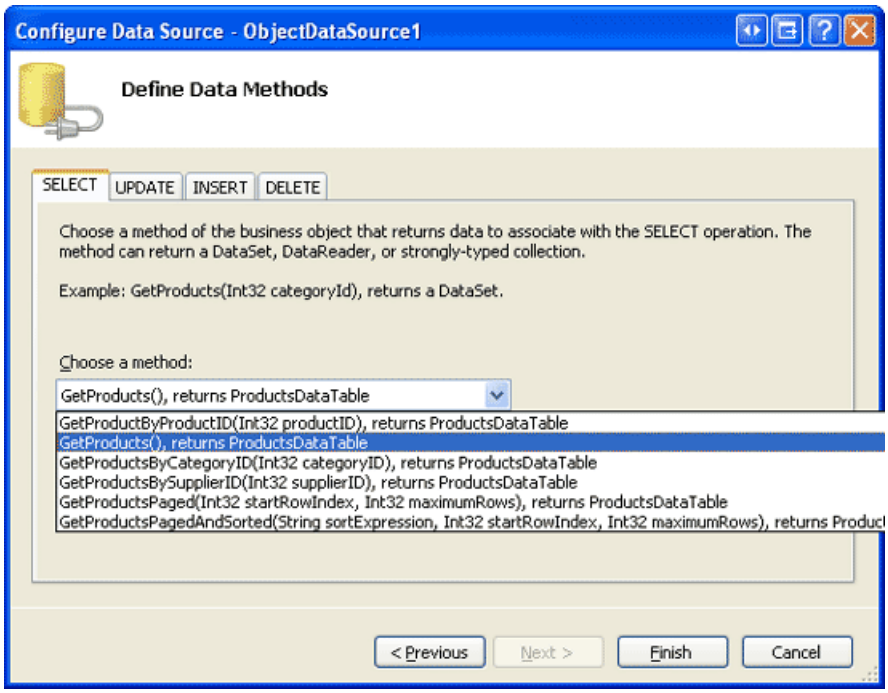


Figure 1: Return the Product Information Using the `GetProducts()` Method (Click to view full-size image)

After completing the ObjectDataSource wizard, Visual Studio will automatically create an `ItemTemplate` for the DataList. Replace this with an `ItemTemplate` that displays each product s name and price and includes an Edit button. Next, create an `EditItemTemplate` with a TextBox Web control for name and price and Update and Cancel buttons. Finally, set the DataList s `RepeatColumns` property to 2.

After these changes, your page s declarative markup should look similar to the following. Double-check to make certain that the Edit, Cancel, and Update buttons have their `CommandName` properties set to Edit, Cancel, and Update, respectively.

```
<asp:DataList ID="Products" runat="server" DataKeyField="ProductID"
    DataSourceID="ProductsDataSource" RepeatColumns="2">
    <ItemTemplate>
        <h5>
            <asp:Label runat="server" ID="ProductNameLabel"
                Text='<%# Eval("ProductName") %>' />
        </h5>
        Price:
            <asp:Label runat="server" ID="Label1"
                Text='<%# Eval("UnitPrice", "{0:C}") %>' />
        <br />
            <asp:Button runat="server" id="EditProduct" CommandName="Edit"
                Text="Edit" />
        <br />
        <br />
    </ItemTemplate>
    <EditItemTemplate>
        Product name:
            <asp:TextBox ID="ProductName" runat="server"
                Text='<%# Eval("ProductName") %>' />
        <br />
        Price:
            <asp:TextBox ID="UnitPrice" runat="server"
                Text='<%# Eval("UnitPrice", "{0:C}") %>' />
        <br />
        <br />
            <asp:Button ID="UpdateProduct" runat="server" CommandName="Update"
                Text="Update" />
            <asp:Button ID="CancelUpdate" runat="server" CommandName="Cancel"
                Text="Cancel" />
    </EditItemTemplate>
</asp:DataList>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    SelectMethod="GetProducts" TypeName="ProductsBLL"
    OldValuesParameterFormatString="original_{0}">
</asp:ObjectDataSource>
```

NOTE

For this tutorial the DataList s view state must be enabled.

Take a moment to view our progress through a browser (see Figure 2).

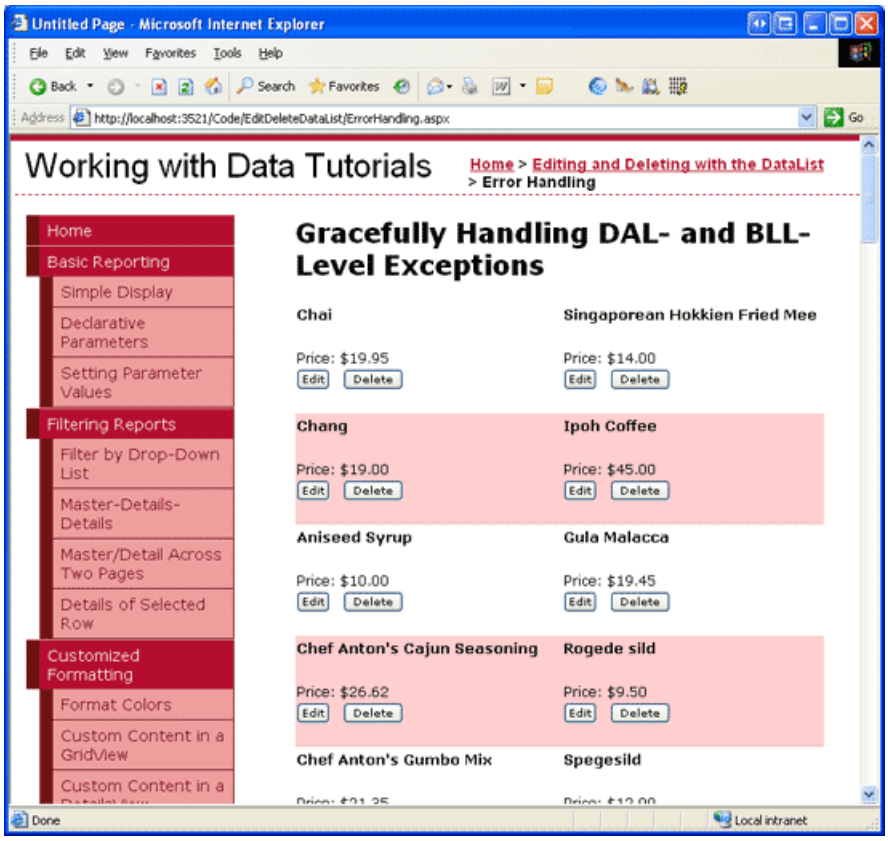


Figure 2: Each Product Includes an Edit Button (Click to view full-size image)

Currently, the Edit button only causes a postback it doesn t yet make the product editable. To enable editing, we need to create event handlers for the DataList s `EditCommand`, `CancelCommand`, and `UpdateCommand` events. The `EditCommand` and `CancelCommand` events simply update the DataList s `EditItemIndex` property and rebind the data to the DataList:

```
protected void Products_EditCommand(object source, DataListCommandEventArgs e)
{
    // Set the DataList's EditItemIndex property to the
    // index of the DataListItem that was clicked
    Products.EditItemIndex = e.Item.ItemIndex;
    // Rebind the data to the DataList
    Products.DataBind();
}
protected void Products_CancelCommand(object source, DataListCommandEventArgs e)
{
    // Set the DataList's EditItemIndex property to -1
    Products.EditItemIndex = -1;
    // Rebind the data to the DataList
    Products.DataBind();
}
```

The `UpdateCommand` event handler is a bit more involved. It needs to read in the edited product s `ProductID` from the `DataKeys` collection along with the product s name and price from the TextBoxes in the `EditItemTemplate`, and then call the `ProductsBLL` class s `UpdateProduct` method before returning the DataList to its pre-editing state.

For now, let s just use the exact same code from the `UpdateCommand` event handler in the *Overview of Editing and Deleting Data in the DataList* tutorial. We'll add the code to gracefully handle exceptions in step 2.

```
protected void Products_UpdateCommand(object source, DataListCommandEventArgs e)
{
    // Read in the ProductID from the DataKeys collection
    int productID = Convert.ToInt32(Products.DataKeys[e.Item.ItemIndex]);
    // Read in the product name and price values
    TextBox productName = (TextBox)e.Item.FindControl("ProductName");
    TextBox unitPrice = (TextBox)e.Item.FindControl("UnitPrice");
    string productNameValue = null;
    if (productName.Text.Trim().Length > 0)
        productNameValue = productName.Text.Trim();
    decimal? unitPriceValue = null;
    if (unitPrice.Text.Trim().Length > 0)
        unitPriceValue = Decimal.Parse(unitPrice.Text.Trim(),
            System.Globalization.NumberStyles.Currency);
    // Call the ProductsBLL's UpdateProduct method...
    ProductsBLL productsAPI = new ProductsBLL();
    productsAPI.UpdateProduct(productNameValue, unitPriceValue, productID);
    // Revert the DataList back to its pre-editing state
    Products.EditItemIndex = -1;
    Products.DataBind();
}
```

In the face of invalid input which can be in the form of an improperly formatted unit price, an illegal unit price value like -$5.00, or the omission of the product s name an exception will be raised. Since the `UpdateCommand` event handler does not include any exception handling code at this point, the exception will bubble up to the ASP.NET runtime, where it will be displayed to the end user (see Figure 3).

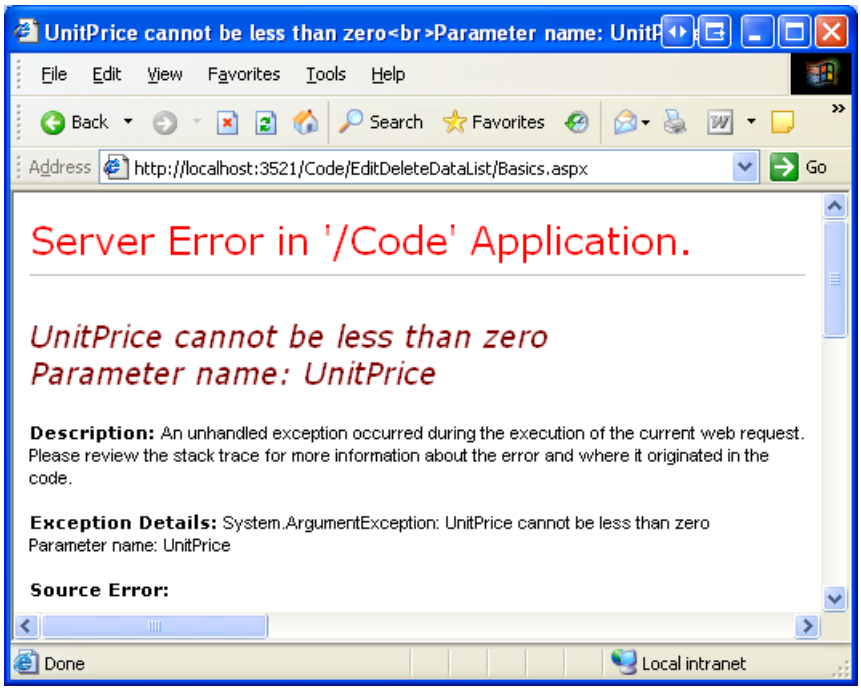


Figure 3: When an Unhandled Exception Occurs, the End User Sees an Error Page

Step 2: Gracefully Handling Exceptions in the UpdateCommand Event Handler

During the updating workflow, exceptions can occur in the `UpdateCommand` event handler, the BLL, or the DAL. For example, if a user enters a price of Too expensive, the `Decimal.Parse` statement in the `UpdateCommand` event handler will throw a `FormatException` exception. If the user omits the product s name or if the price has a negative value, the DAL will raise an exception.

When an exception occurs, we want to display an informative message within the page itself. Add a Label Web control to the page whose `ID` is set to `ExceptionDetails`. Configure the Label s text to display in a red, extra-

large, bold and italic font by assigning its `CssClass` property to the `Warning` CSS class, which is defined in the `Styles.css` file.

When an error occurs, we only want the Label to be displayed once. That is, on subsequent postbacks, the Label s warning message should disappear. This can be accomplished by either clearing out the Label s `Text` property or settings its `Visible` property to `False` in the `Page_Load` event handler (as we did back in the Handling BLL- and DAL-Level Exceptions in an ASP.NET Page tutorial) or by disabling the Label s view state support. Let s use the latter option.

```
<asp:Label ID="ExceptionDetails" EnableViewState="False" CssClass="Warning"
    runat="server" />
```

When an exception is raised, we'll assign the details of the exception to the `ExceptionDetails` Label control s `Text` property. Since its view state is disabled, on subsequent postbacks the `Text` property s programmatic changes will be lost, reverting back to the default text (an empty string), thereby hiding the warning message.

To determine when an error has been raised in order to display a helpful message on the page, we need to add a `Try ... Catch` block to the `UpdateCommand` event handler. The `Try` portion contains code that may lead to an exception, while the `Catch` block contains code that is executed in the face of an exception. Check out the Exception Handling Fundamentals section in the .NET Framework documentation for more information on the `Try ... Catch` block.

```
protected void Products_UpdateCommand(object source, DataListCommandEventArgs e)
{
    // Handle any exceptions raised during the editing process
    try
    {
        // Read in the ProductID from the DataKeys collection
        int productID = Convert.ToInt32(Products.DataKeys[e.Item.ItemIndex]);
        ... Some code omitted for brevity ...
    }
    catch (Exception ex)
    {
        // TODO: Display information about the exception in ExceptionDetails
    }
}
```

When an exception of any type is thrown by code within the `Try` block, the `Catch` block s code will begin executing. The type of exception that is thrown `DbException`, `NoNullAllowedException`, `ArgumentException`, and so on depends on what, exactly, precipitated the error in the first place. If there s a problem at the database level, a `DbException` will be thrown. If an illegal value is entered for the `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, or `ReorderLevel` fields, an `ArgumentException` will be thrown, as we added code to validate these field values in the `ProductsDataTable` class (see the Creating a Business Logic Layer tutorial).

We can provide a more helpful explanation to the end user by basing the message text on the type of exception caught. The following code which was used in a nearly identical form back in the Handling BLL- and DAL-Level Exceptions in an ASP.NET Page tutorial provides this level of detail:

```
private void DisplayExceptionDetails(Exception ex)
{
    // Display a user-friendly message
    ExceptionDetails.Text = "There was a problem updating the product. ";
    if (ex is System.Data.Common.DbException)
        ExceptionDetails.Text += "Our database is currently experiencing problems.
            Please try again later.";
    else if (ex is NoNullAllowedException)
        ExceptionDetails.Text += "There are one or more required fields that are
            missing.";
    else if (ex is ArgumentException)
    {
        string paramName = ((ArgumentException)ex).ParamName;
        ExceptionDetails.Text +=
            string.Concat("The ", paramName, " value is illegal.");
    }
    else if (ex is ApplicationException)
        ExceptionDetails.Text += ex.Message;
}
```

To complete this tutorial, simply call the `DisplayExceptionDetails` method from the `Catch` block passing in the caught `Exception` instance (`ex`).

With the `Try ... Catch` block in place, users are presented with a more informative error message, as Figures 4 and 5 show. Note that in the face of an exception the DataList remains in edit mode. This is because once the exception occurs, the control flow is immediately redirected to the `Catch` block, bypassing the code that returns the DataList to its pre-editing state.

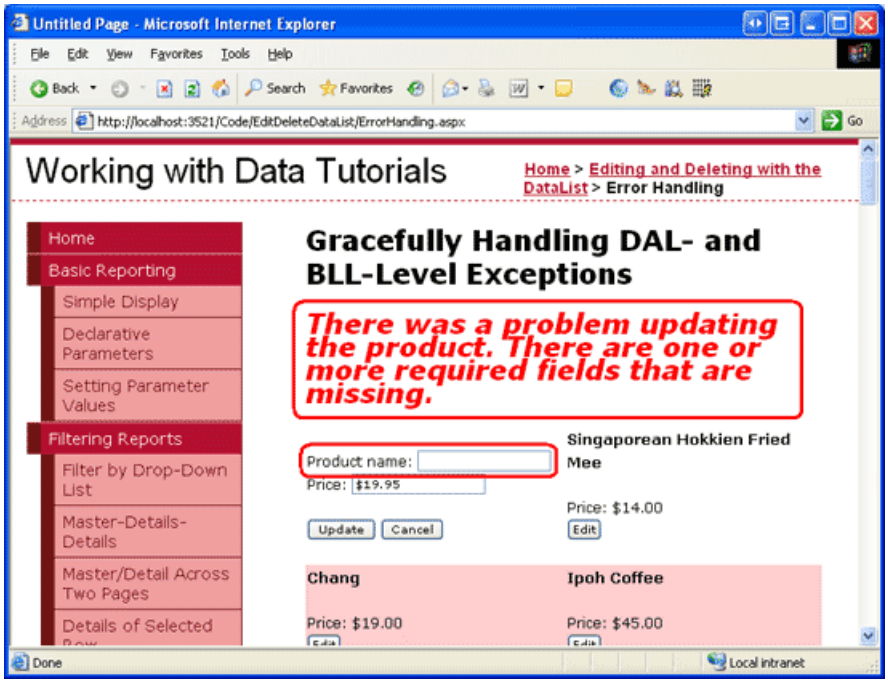


Figure 4: An Error Message is Displayed if a User Omits a Required Field (Click to view full-size image)

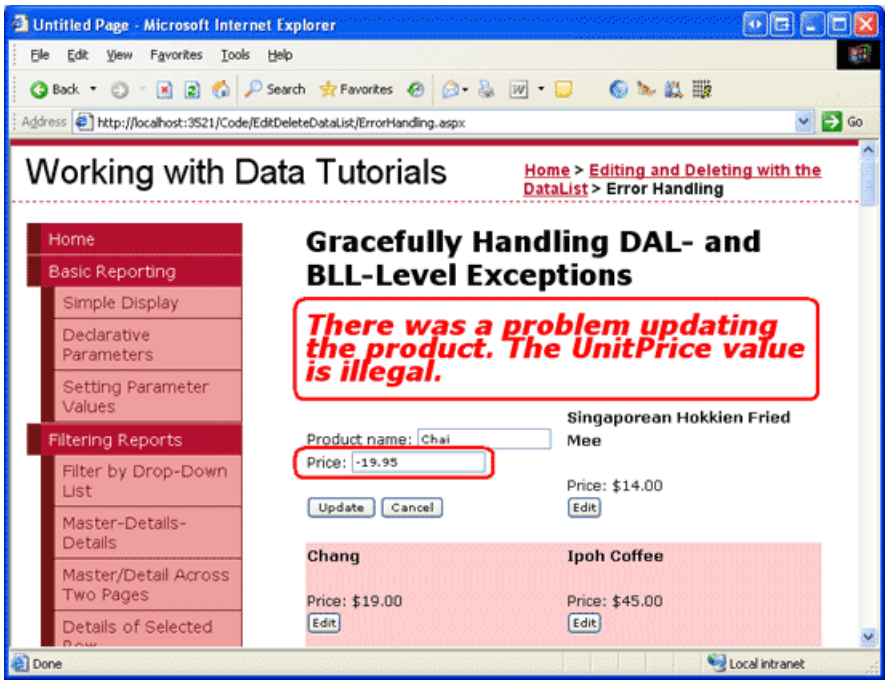


Figure 5: An Error Message is Displayed When Entering a Negative Price (Click to view full-size image)

Summary

The GridView and ObjectDataSource provide post-level event handlers that include information about any exceptions that were raised during the updating and deleting workflow, as well as properties that can be set to indicate whether or not the exception has been handled. These features, however, are unavailable when working with the DataList and using the BLL directly. Instead, we are responsible for implementing exception handling.

In this tutorial we saw how to add exception handling to an editable DataList s updating workflow by adding a `Try ... Catch` block to the `UpdateCommand` event handler. If an exception is raised during the updating workflow, the `Catch` block s code executes, displaying helpful information in the `ExceptionDetails` Label.

At this point, the DataList makes no effort to prevent exceptions from happening in the first place. Even though we know that a negative price will result in an exception, we haven't yet added any functionality to proactively prevent a user from entering such invalid input. In our next tutorial we'll see how to help reduce the exceptions caused by invalid user input by adding validation controls in the `EditItemTemplate`.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Design Guidelines for Exceptions
- Error Logging Modules and Handlers (ELMAH) (an open-source library for logging errors)
- Enterprise Library for .NET Framework 2.0 (includes the Exception Management Application Block)

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Ken Pespisa. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Adding Validation Controls to the DataList's Editing Interface (C#)

2/20/2020 • 14 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll see how easy it is to add validation controls to the DataList's EditItemTemplate in order to provide a more foolproof editing user interface.

Introduction

In the DataList editing tutorials thus far, the DataLists editing interfaces have not included any proactive user input validation even though invalid user input such as a missing product name or negative price results in an exception. In the preceding tutorial we examined how to add exception handling code to the DataList s `UpdateCommand` event handler in order to catch and gracefully display information about any exceptions that were raised. Ideally, however, the editing interface would include validation controls to prevent a user from entering such invalid data in the first place.

In this tutorial we'll see how easy it is to add validation controls to the DataList s `EditItemTemplate` in order to provide a more foolproof editing user interface. Specifically, this tutorial takes the example created in the preceding tutorial and augments the editing interface to include appropriate validation.

Step 1: Replicating the Example fromHandling BLL- and DAL-Level Exceptions

In the Handling BLL- and DAL-Level Exceptions tutorial we created a page that listed the names and prices of the products in a two-column, editable DataList. Our goal for this tutorial is to augment the DataList s editing interface to include validation controls. In particular, our validation logic will:

- Require that the product s name be provided
- Ensure that the value entered for the price is a valid currency format
- Ensure that the value entered for the price is greater than or equal to zero, since a negative `UnitPrice` value is illegal

Before we can look at augmenting the previous example to include validation, we first need to replicate the example from the `ErrorHandling.aspx` page in the `EditDeleteDataList` folder to the page for this tutorial, `UIValidation.aspx`. To achieve this we need to copy over both the `ErrorHandling.aspx` page s declarative markup and its source code. First copy over the declarative markup by performing the following steps:

1. Open the `ErrorHandling.aspx` page in Visual Studio
2. Go to the page s declarative markup (click on the Source button at the bottom of the page)
3. Copy the text within the `<asp:Content>` and `</asp:Content>` tags (lines 3 through 32), as shown in Figure 1.

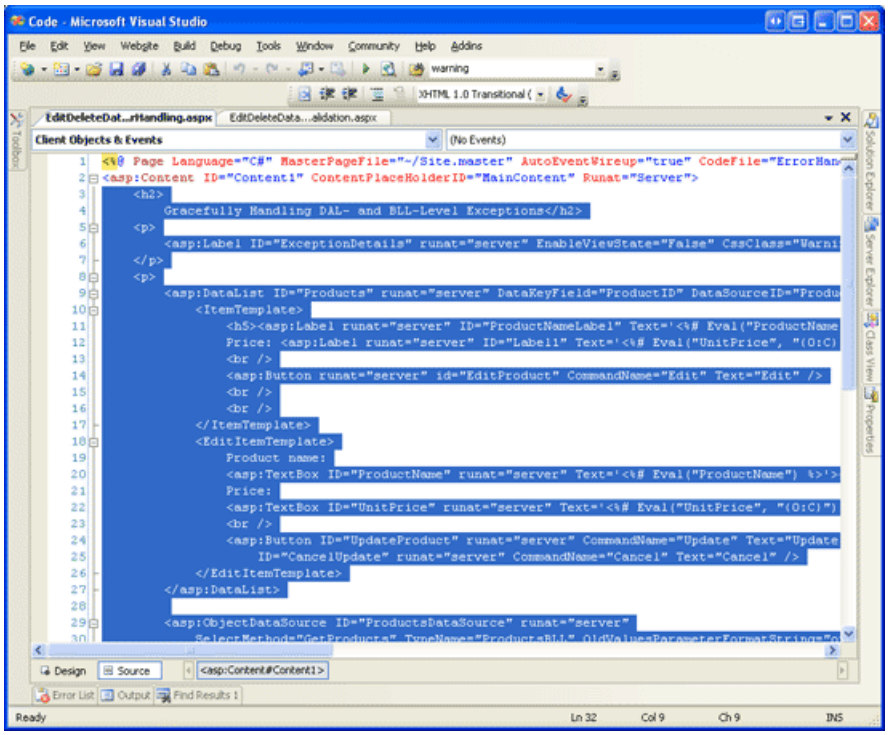


Figure 1: Copy the Text Within the `<asp:Content>` Control (Click to view full-size image)

1. Open the `UIValidation.aspx` page
2. Go to the page s declarative markup
3. Paste the text within the `<asp:Content>` control.

To copy over the source code, open the `ErrorHandling.aspx.vb` page and copy just the text *within* the `EditDeleteDataList_ErrorHandling` class. Copy the three event handlers (`Products_EditCommand` , `Products_CancelCommand` , and `Products_UpdateCommand`) along with the `DisplayExceptionDetails` method, but do **not** copy the class declaration or `using` statements. Paste the copied text *within* the `EditDeleteDataList_UIValidation` class in `UIValidation.aspx.vb` .

After moving over the content and code from `ErrorHandling.aspx` to `UIValidation.aspx` , take a moment to test out the pages in a browser. You should see the same output and experience the same functionality in each of these two pages (see Figure 2).

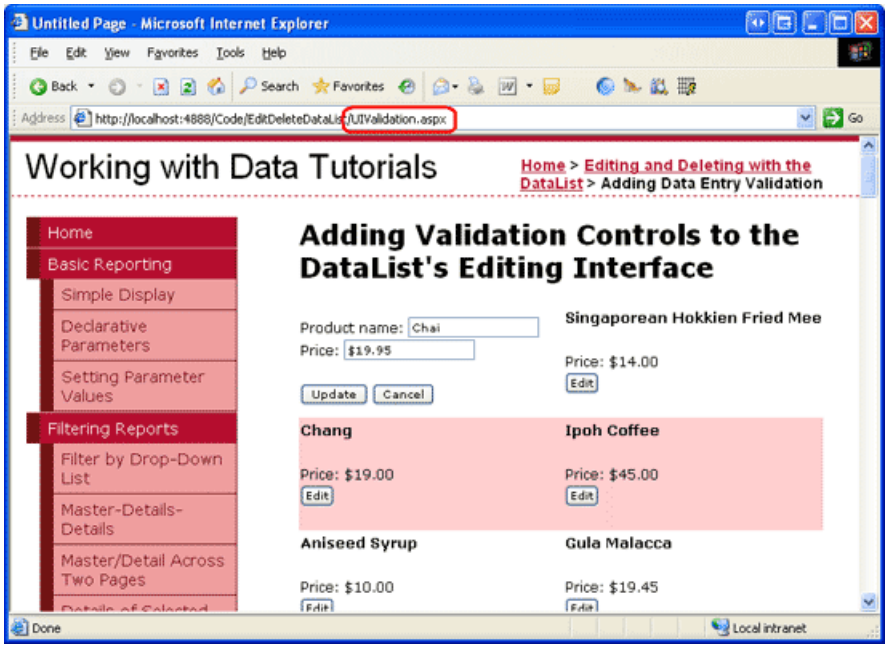


Figure 2: The `UIValidation.aspx` Page Mimics the Functionality in `ErrorHandling.aspx` (Click to view full-size

image)

Step 2: Adding the Validation Controls to the DataList s EditItemTemplate

When constructing data entry forms, it is important that users enter any required fields and that all their provided inputs are legal, properly-formatted values. To help ensure that a user s inputs are valid, ASP.NET provides five built-in validation controls that are designed to validate the value of a single input Web control:

- RequiredFieldValidator ensures that a value has been provided
- CompareValidator validates a value against another Web control value or a constant value, or ensures that the value s format is legal for a specified data type
- RangeValidator ensures that a value is within a range of values
- RegularExpressionValidator validates a value against a regular expression
- CustomValidator validates a value against a custom, user-defined method

For more information on these five controls refer back to the Adding Validation Controls to the Editing and Inserting Interfaces tutorial or check out the Validation Controls section of the ASP.NET Quickstart Tutorials.

For our tutorial we'll need to use a RequiredFieldValidator to ensure that a value for the product name has been supplied and a CompareValidator to ensure that the entered price has a value greater than or equal to 0 and is presented in a valid currency format.

> **NOTE**
> While ASP.NET 1.x had these same five validation controls, ASP.NET 2.0 has added a number of improvements, the main two being client-side script support for browsers in addition to Internet Explorer and the ability to partition validation controls on a page into validation groups. For more information on the new validation control features in 2.0, refer to Dissecting the Validation Controls in ASP.NET 2.0.

Let s start by adding the necessary validation controls to the DataList s `EditItemTemplate`. This task can be performed through the Designer by clicking on the Edit Templates link from the DataList s smart tag, or through the declarative syntax. Let s step through the process using the Edit Templates option from the Design view. After choosing to edit the DataList s `EditItemTemplate`, add a RequiredFieldValidator by dragging it from the Toolbox into the template editing interface, placing it after the `ProductName` TextBox.

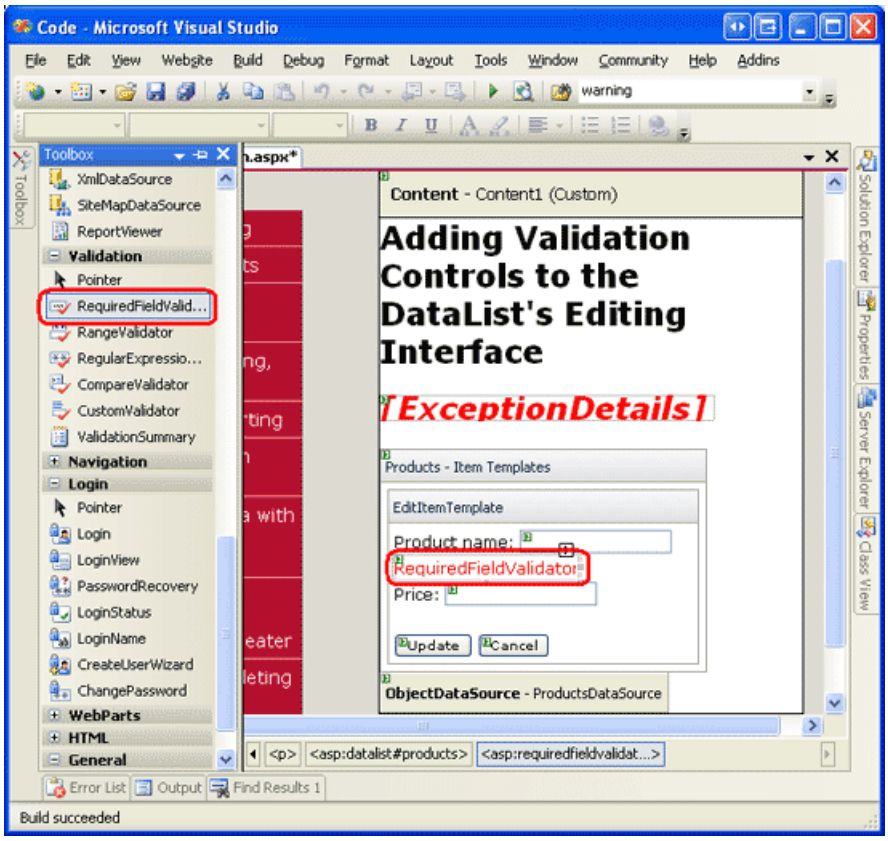


Figure 3: Add a RequiredFieldValidator to the `EditItemTemplate After` the `ProductName` TextBox (Click to view full-size image)

All validation controls work by validating the input of a single ASP.NET Web control. Therefore, we need to indicate that the RequiredFieldValidator we just added should validate against the `ProductName` TextBox; this is done by setting the validation control s `ControlToValidate` property to the `ID` of the appropriate Web control (`ProductName` , in this instance). Next, set the `ErrorMessage` property to You must provide the product s name and the `Text` property to *. The `Text` property value, if provided, is the text that is displayed by the validation control if the validation fails. The `ErrorMessage` property value, which is required, is used by the ValidationSummary control; if the `Text` property value is omitted, the `ErrorMessage` property value is displayed by the validation control on invalid input.

After setting these three properties of the RequiredFieldValidator, your screen should look similar to Figure 4.

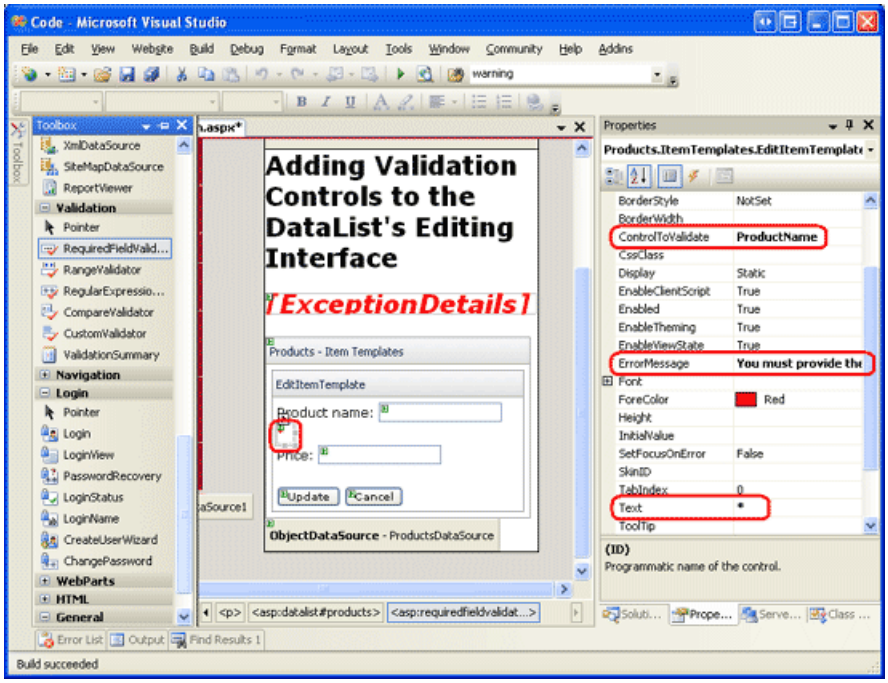


Figure 4: Set the RequiredFieldValidator s `ControlToValidate` , `ErrorMessage` , and `Text` Properties (Click to view full-size image)

With the RequiredFieldValidator added to the `EditItemTemplate` , all that remains is to add the necessary validation for the product s price TextBox. Since the `UnitPrice` is optional when editing a record, we don t need to add a RequiredFieldValidator. We do, however, need to add a CompareValidator to ensure that the `UnitPrice` , if supplied, is properly formatted as a currency and is greater than or equal to 0.

Add the CompareValidator into the `EditItemTemplate` and set its `ControlToValidate` property to `UnitPrice` , its `ErrorMessage` property to The price must be greater than or equal to zero and cannot include the currency symbol, and its `Text` property to *. To indicate that the `UnitPrice` value must be greater than or equal to 0, set the CompareValidator s `Operator` property to `GreaterThanEqual` , its `ValueToCompare` property to 0, and its `Type` property to `Currency` .

After adding these two validation controls, the DataList s `EditItemTemplate` s declarative syntax should look similar to the following:

```
<EditItemTemplate>
    Product name:
        <asp:TextBox ID="ProductName" runat="server"
            Text='<%# Eval("ProductName") %>'></asp:TextBox>
        <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
            ControlToValidate="ProductName"
            ErrorMessage="You must provide the product's name"
            runat="server">*</asp:RequiredFieldValidator>
    <br />
    Price:
        <asp:TextBox ID="UnitPrice" runat="server"
            Text='<%# Eval("UnitPrice", "{0:C}") %>'></asp:TextBox>
        <asp:CompareValidator ID="CompareValidator1"
            ControlToValidate="UnitPrice"
            ErrorMessage="The price must be greater than or equal to zero
                          and cannot include the currency symbol"
            Operator="GreaterThanEqual" Type="Currency" ValueToCompare="0"
            runat="server">*</asp:CompareValidator><br />
    <br />
    <asp:Button ID="UpdateProduct" runat="server" CommandName="Update"
        Text="Update" />
    <asp:Button ID="CancelUpdate" runat="server" CommandName="Cancel"
        Text="Cancel" />
</EditItemTemplate>
```

After making these changes, open the page in a browser. If you attempt to omit the name or enter an invalid price value when editing a product, an asterisk appears next to the textbox. As Figure 5 shows, a price value that includes the currency symbol such as $19.95 is considered invalid. The CompareValidator s `Currency` `Type` allows for digit separators (such as commas or periods, depending on the culture settings) and a leading plus or minus sign, but does *not* permit a currency symbol. This behavior may perplex users as the editing interface currently renders the `UnitPrice` using the currency format.

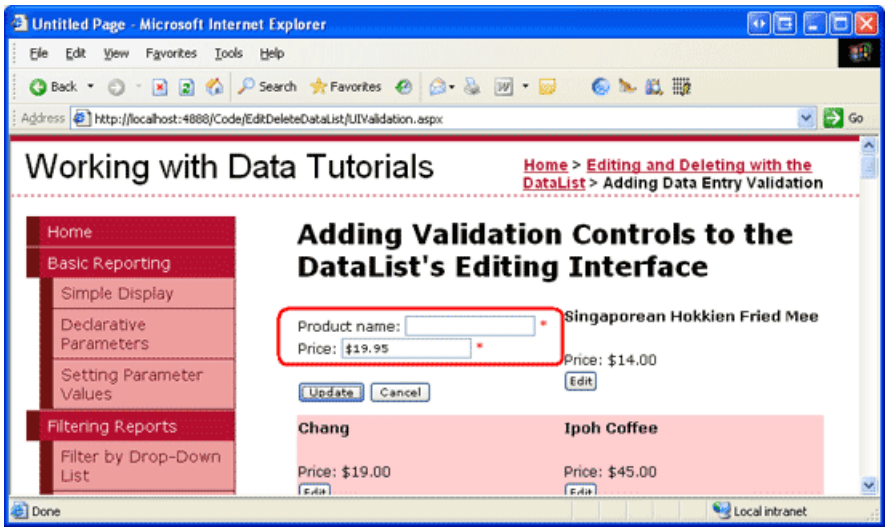


Figure 5: An Asterisk Appears Next to the Textboxes with Invalid Input (Click to view full-size image)

While the validation works as-is, the user has to manually remove the currency symbol when editing a record, which is not acceptable. Moreover, if there are invalid inputs in the editing interface neither the Update nor Cancel buttons, when clicked, will invoke a postback. Ideally, the Cancel button would return the DataList to its pre-editing state regardless of the validity of the user s inputs. Also, we need to ensure that the page s data is valid before updating the product information in the DataList s `UpdateCommand` event handler, as the validation controls client-side logic can be bypassed by users whose browsers either don t support JavaScript or have its support disabled.

Removing the Currency Symbol from the EditItemTemplate s UnitPrice TextBox

When using the CompareValidator s `Currency``Type`, the input being validated must not include any currency symbols. The presence of such symbols causes the CompareValidator to mark the input as invalid. However, our editing interface currently includes a currency symbol in the `UnitPrice` TextBox, meaning the user must explicitly remove the currency symbol before saving their changes. To remedy this we have three options:

1. Configure the `EditItemTemplate` so that the `UnitPrice` TextBox value is not formatted as a currency.
2. Allow the user to enter a currency symbol by removing the CompareValidator and replacing it with a RegularExpressionValidator that checks for a properly formatted currency value. The challenge here is that the regular expression to validate a currency value is not as straightforward as the CompareValidator and would require writing code if we wanted to incorporate culture settings.
3. Remove the validation control altogether and rely on custom server-side validation logic in the GridView s `RowUpdating` event handler.

Let s go with option 1 for this tutorial. Currently the `UnitPrice` is formatted as a currency value due to the databinding expression for the TextBox in the `EditItemTemplate`: `<%# Eval("UnitPrice", "{0:c}") %>`. Change the `Eval` statement to `Eval("UnitPrice", "{0:n2}")`, which formats the result as a number with two digits of precision. This can be done directly through the declarative syntax or by clicking on the Edit DataBindings link from the `UnitPrice` TextBox in the DataList s `EditItemTemplate`.

With this change, the formatted price in the editing interface includes commas as the group separator and a period as the decimal separator, but leaves off the currency symbol.

> **NOTE**
>
> When removing the currency format from the editable interface, I find it helpful to put the currency symbol as text outside the TextBox. This serves as a hint to the user that they do not need to provide the currency symbol.

Fixing the Cancel Button

By default, the validation Web controls emit JavaScript to perform validation on the client-side. When a Button, LinkButton, or ImageButton is clicked, the validation controls on the page are checked on the client-side before the postback occurs. If there is any invalid data, the postback is cancelled. For certain Buttons, though, the validity of the data might be immaterial; in such a case, having the postback cancelled due to invalid data is a nuisance.

The Cancel button is such an example. Imagine that a user enters invalid data, such as omitting the product s name, and then decides she doesn t want to save the product after all and hits the Cancel button. Currently, the Cancel button triggers the validation controls on the page, which report that the product name is missing and prevent the postback. Our user has to type some text into the `ProductName` TextBox just to cancel out of the editing process.

Fortunately, the Button, LinkButton, and ImageButton have a `CausesValidation` property that can indicate whether or not clicking the Button should initiate the validation logic (defaults to `True`). Set the Cancel Button s `CausesValidation` property to `False`.

Ensuring the Inputs are Valid in the UpdateCommand Event Handler

Due to the client-side script emitted by the validation controls, if a user enters invalid input the validation controls cancel any postbacks initiated by Button, LinkButton, or ImageButton controls whose `CausesValidation` properties are `True` (the default). However, if a user is visiting with an antiquated browser or one whose JavaScript support has been disabled, the client-side validation checks will not execute.

All of the ASP.NET validation controls repeat their validation logic immediately upon postback and report the overall validity of the page s inputs via the `Page.IsValid` property. However, the page flow is not interrupted or stopped in any way based on the value of `Page.IsValid`. As developers, it is our responsibility to ensure that the `Page.IsValid` property has a value of `True` before proceeding with code that assumes valid input data.

If a user has JavaScript disabled, visits our page, edits a product, enters a price value of Too expensive, and clicks the Update button, the client-side validation will be bypassed and a postback will ensue. On postback, the ASP.NET page s `UpdateCommand` event handler executes and an exception is raised when attempting to parse Too expensive to a `Decimal`. Since we have exception handling, such an exception will be handled gracefully, but we could prevent the invalid data from slipping through in the first place by only proceeding with the `UpdateCommand` event handler if `Page.IsValid` has a value of `True`.

Add the following code to the start of the `UpdateCommand` event handler, immediately before the `Try` block:

```
if (!Page.IsValid)
    return;
```

With this addition, the product will attempt to be updated only if the submitted data is valid. Most users won't be able to postback invalid data due to the validation controls client-side scripts, but users whose browsers don t support JavaScript or that have JavaScript support disabled, can bypass the client-side checks and submit invalid data.

> **NOTE**
> The astute reader will recall that when updating data with the GridView, we didn't need to explicitly check the `Page.IsValid` property in our page s code-behind class. This is because the GridView consults the `Page.IsValid` property for us and only proceeds with the update only if it returns a value of `True`.

Step 3: Summarizing Data Entry Problems

In addition to the five validation controls, ASP.NET includes the ValidationSummary control, which displays the `ErrorMessage` s of those validation controls that detected invalid data. This summary data can be displayed as text on the web page or through a modal, client-side messagebox. Let s enhance this tutorial to include a client-side messagebox summarizing any validation problems.

To accomplish this, drag a ValidationSummary control from the Toolbox onto the Designer. The location of the ValidationSummary control doesn t really matter, since we re going to configure it to only display the summary as a messagebox. After adding the control, set its `ShowSummary` property to `False` and its `ShowMessageBox` property to `True`. With this addition, any validation errors are summarized in a client-side messagebox (see Figure 6).

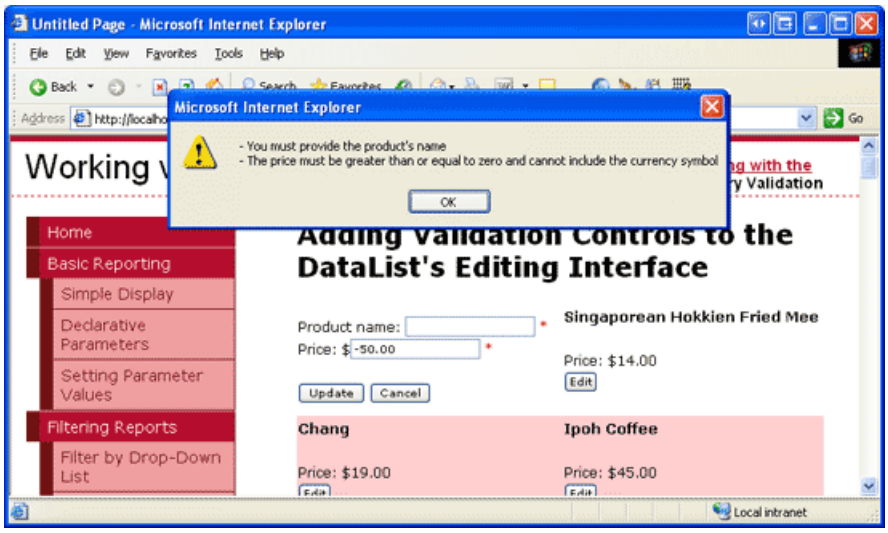


Figure 6: The Validation Errors are Summarized in a Client-Side Messagebox (Click to view full-size image)

Summary

In this tutorial we saw how to reduce the likelihood of exceptions by using validation controls to proactively ensure that our users inputs are valid before attempting to use them in the updating workflow. ASP.NET provides five validation Web controls that are designed to inspect a particular Web control s input and report back on the input s validity. In this tutorial we used two of those five controls the RequiredFieldValidator and the CompareValidator to ensure that the product s name was supplied and that the price had a currency format with a value greater than or equal to zero.

Adding validation controls to the DataList s editing interface is as simple as dragging them onto the `EditItemTemplate` from the Toolbox and setting a handful of properties. By default, the validation controls automatically emit client-side validation script; they also provide server-side validation on postback, storing the cumulative result in the `Page.IsValid` property. To bypass the client-side validation when a Button, LinkButton, or ImageButton is clicked, set the button s `CausesValidation` property to `False`. Also, before performing any tasks with the data submitted on postback, ensure that the `Page.IsValid` property returns `True`.

All of the DataList editing tutorials we ve examined so far have had very simple editing interfaces a TextBox for the product s name and another for the price. The editing interface, however, can contain a mix of different Web controls, such as DropDownLists, Calendars, RadioButtons, CheckBoxes, and so on. In our next tutorial we'll look at building an interface that uses a variety of Web controls.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via

his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Dennis Patterson, Ken Pespisa, and Liz Shulok. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Customizing the DataList's Editing Interface (C#)

2/20/2020 • 13 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we will create a richer editing interface for the DataList, one that includes DropDownLists and a CheckBox.

Introduction

The markup and Web controls in the DataList s `EditItemTemplate` define its editable interface. In all of the editable DataList examples we ve examined thus far, the editable interface has been composed of TextBox Web controls. In the preceding tutorial we improved the edit-time user experience by adding validation controls.

The `EditItemTemplate` can be further expanded to include Web controls other than the TextBox, such as DropDownLists, RadioButtonLists, Calendars, and so on. As with TextBoxes, when customizing the editing interface to include other Web controls, employ the following steps:

1. Add the Web control to the `EditItemTemplate`.
2. Use databinding syntax to assign the corresponding data field value to the appropriate property.
3. In the `UpdateCommand` event handler, programmatically access the Web control value and pass it into the appropriate BLL method.

In this tutorial we will create a richer editing interface for the DataList, one that includes DropDownLists and a CheckBox. In particular, we'll create a DataList that lists product information and permits the product s name, supplier, category, and discontinued status to be updated (see Figure 1).

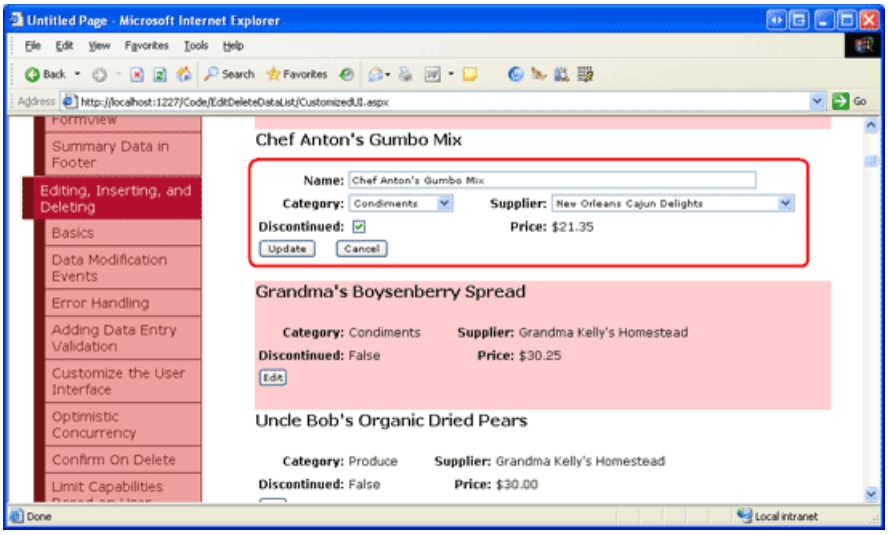


Figure 1: The Editing Interface Includes a TextBox, Two DropDownLists, and a CheckBox (Click to view full-size image)

Step 1: Displaying Product Information

Before we can create the DataList s editable interface, we first need to build the read-only interface. Start by opening the `CustomizedUI.aspx` page from the `EditDeleteDataList` folder and, from the Designer, add a DataList

to the page, setting its `ID` property to `Products`. From the DataList s smart tag, create a new ObjectDataSource. Name this new ObjectDataSource `ProductsDataSource` and configure it to retrieve data from the `ProductsBLL` class s `GetProducts` method. As with the previous editable DataList tutorials, we'll update the edited product s information by going directly to the Business Logic Layer. Accordingly, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None).

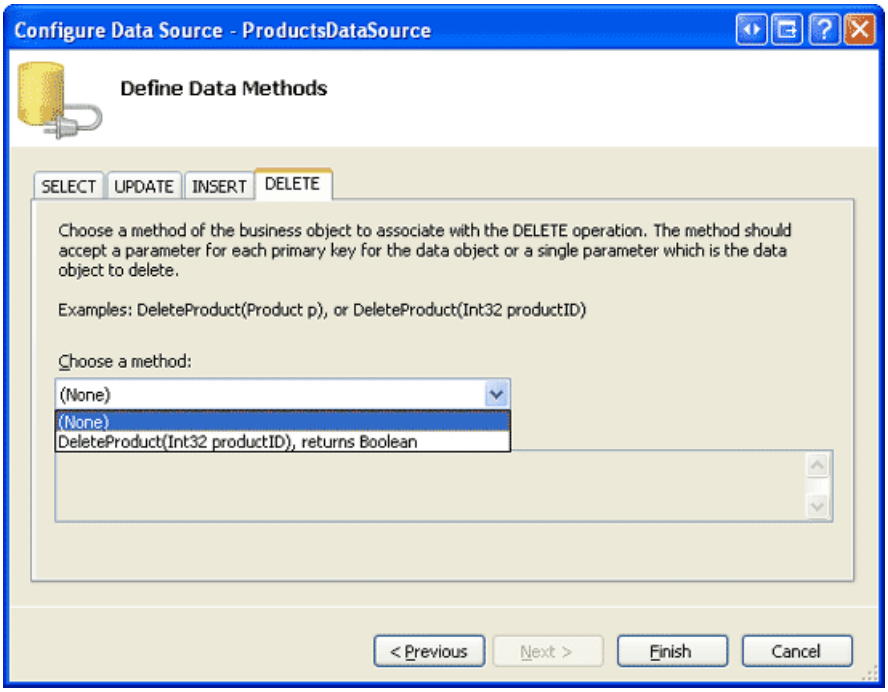


Figure 2: Set the UPDATE, INSERT, and DELETE Tabs Drop-Down Lists to (None) (Click to view full-size image)

After configuring the ObjectDataSource, Visual Studio will create a default `ItemTemplate` for the DataList that lists the name and value for each of the data fields returned. Modify the `ItemTemplate` so that the template lists the product name in an `<h4>` element along with the category name, supplier name, price, and discontinued status. Moreover, add an Edit button, making sure that its `CommandName` property is set to Edit. The declarative markup for my `ItemTemplate` follows:

```
<ItemTemplate>
    <h4>
        <asp:Label ID="ProductNameLabel" runat="server"
            Text='<%# Eval("ProductName") %>' />
    </h4>
    <table border="0">
        <tr>
            <td class="ProductPropertyLabel">Category:</td>
            <td class="ProductPropertyValue">
                <asp:Label ID="CategoryNameLabel" runat="server"
                    Text='<%# Eval("CategoryName") %>' />
            </td>
            <td class="ProductPropertyLabel">Supplier:</td>
            <td class="ProductPropertyValue">
                <asp:Label ID="SupplierNameLabel" runat="server"
                    Text='<%# Eval("SupplierName") %>' />
            </td>
        </tr>
        <tr>
            <td class="ProductPropertyLabel">Discontinued:</td>
            <td class="ProductPropertyValue">
                <asp:Label ID="DiscontinuedLabel" runat="server"
                    Text='<%# Eval("Discontinued") %>' />
            </td>
            <td class="ProductPropertyLabel">Price:</td>
            <td class="ProductPropertyValue">
                <asp:Label ID="UnitPriceLabel" runat="server"
                    Text='<%# Eval("UnitPrice", "{0:C}") %>' />
            </td>
        </tr>
        <tr>
            <td colspan="4">
                <asp:Button runat="Server" ID="EditButton"
                    Text="Edit" CommandName="Edit" />
            </td>
        </tr>
    </table>
    <br />
</ItemTemplate>
```

The above markup lays out the product information using an <h4> heading for the product s name and a four-column `<table>` for the remaining fields. The `ProductPropertyLabel` and `ProductPropertyValue` CSS classes, defined in `Styles.css`, have been discussed in previous tutorials. Figure 3 shows our progress when viewed through a browser.

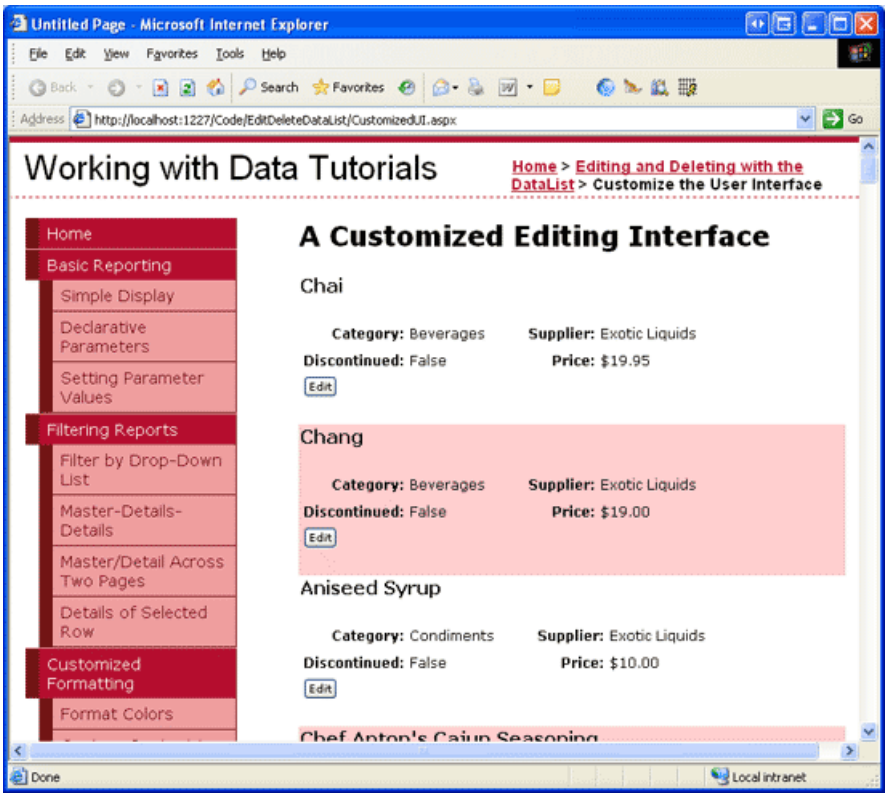


Figure 3: The Name, Supplier, Category, Discontinued Status, and Price of Each Product is Displayed (Click to view full-size image)

Step 2: Adding the Web Controls to the Editing Interface

The first step in building the customized DataList editing interface is to add the needed Web controls to the `EditItemTemplate`. In particular, we need a DropDownList for the category, another for the supplier, and a CheckBox for the discontinued state. Since the product s price is not editable in this example, we can continue displaying it using a Label Web control.

To customize the editing interface, click on the Edit Templates link from the DataList s smart tag and choose the `EditItemTemplate` option from the drop-down list. Add a DropDownList to the `EditItemTemplate` and set its `ID` to `Categories`.

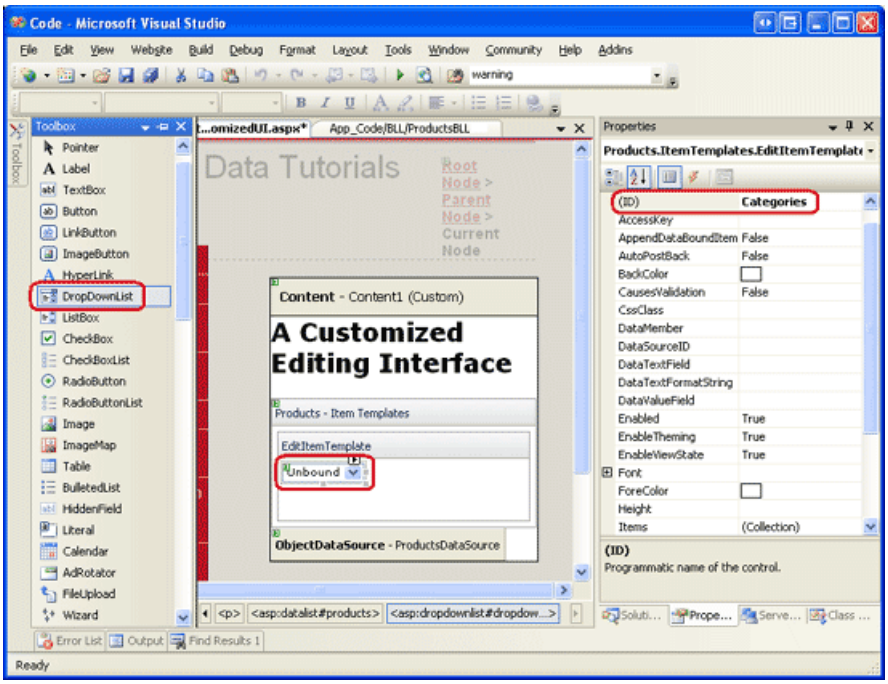


Figure 4: Add a DropDownList for the Categories (Click to view full-size image)

Next, from the DropDownList s smart tag, select the Choose Data Source option and create a new ObjectDataSource named `CategoriesDataSource`. Configure this ObjectDataSource to use the `CategoriesBLL` class s `GetCategories()` method (see Figure 5). Next, the DropDownList s Data Source Configuration Wizard prompts for the data fields to use for each `ListItem` s `Text` and `Value` properties. Have the DropDownList display the `CategoryName` data field and use the `CategoryID` as the value, as shown in Figure 6.

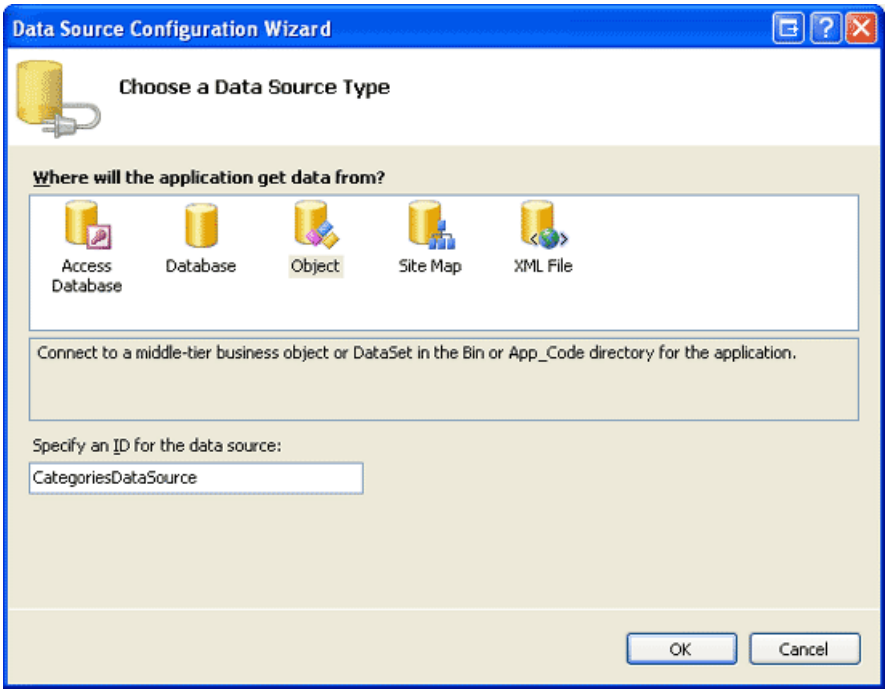


Figure 5: Create a New ObjectDataSource Named `CategoriesDataSource` (Click to view full-size image)

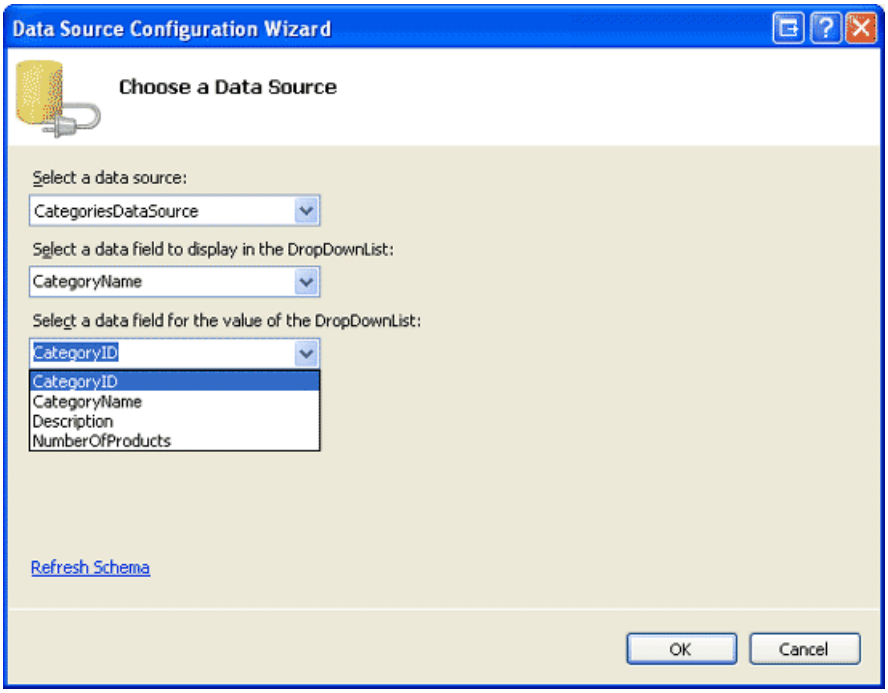


Figure 6: Configure the DropDownList s Display and Value Fields (Click to view full-size image)

Repeat this series of steps to create a DropDownList for the suppliers. Set the `ID` for this DropDownList to `Suppliers` and name its ObjectDataSource `SuppliersDataSource`.

After adding the two DropDownLists, add a CheckBox for the discontinued state and a TextBox for the product s name. Set the `ID` s for the CheckBox and TextBox to `Discontinued` and `ProductName`, respectively. Add a RequiredFieldValidator to ensure that the user provides a value for the product s name.

Lastly, add the Update and Cancel buttons. Remember that for these two buttons it is imperative that their

`CommandName` properties are set to Update and Cancel, respectively.

Feel free to lay out the editing interface however you like. I ve opted to use the same four-column `<table>` layout from the read-only interface, as the following declarative syntax and screen shot illustrates:

```
<EditItemTemplate>
    <h4>
        <asp:Label ID="ProductNameLabel" runat="server"
            Text='<%# Eval("ProductName") %>' />
    </h4>
    <table border="0">
        <tr>
            <td class="ProductPropertyLabel">Name:</td>
            <td colspan="3" class="ProductPropertyValue">
                <asp:TextBox runat="server" ID="ProductName" Width="90%" />
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                    ControlToValidate="ProductName"
                    ErrorMessage="You must enter a name for the product."
                    runat="server">*</asp:RequiredFieldValidator>
            </td>
        </tr>
        <tr>
            <td class="ProductPropertyLabel">Category:</td>
            <td class="ProductPropertyValue">
                <asp:DropDownList ID="Categories" runat="server"
                    DataSourceID="CategoriesDataSource"
                    DataTextField="CategoryName" DataValueField="CategoryID" />
            </td>
            <td class="ProductPropertyLabel">Supplier:</td>
            <td class="ProductPropertyValue">
                <asp:DropDownList ID="Suppliers" DataTextField="CompanyName"
                    DataSourceID="SuppliersDataSource"
                    DataValueField="SupplierID" runat="server" />
            </td>
        </tr>
        <tr>
            <td class="ProductPropertyLabel">Discontinued:</td>
            <td class="ProductPropertyValue">
                <asp:CheckBox runat="server" id="Discontinued" />
            </td>
            <td class="ProductPropertyLabel">Price:</td>
            <td class="ProductPropertyValue">
                <asp:Label ID="UnitPriceLabel" runat="server"
                    Text='<%# Eval("UnitPrice", "{0:C}") %>' />
            </td>
        </tr>
        <tr>
            <td colspan="4">
                <asp:Button runat="Server" ID="UpdateButton" CommandName="Update"
                    Text="Update" />

                <asp:Button runat="Server" ID="CancelButton" CommandName="Cancel"
                    Text="Cancel" CausesValidation="False" />
            </td>
        </tr>
    </table>
    <br />
    <asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
        OldValuesParameterFormatString="original_{0}" SelectMethod="GetCategories"
        TypeName="CategoriesBLL">
    </asp:ObjectDataSource>
    <asp:ObjectDataSource ID="SuppliersDataSource" runat="server"
        OldValuesParameterFormatString="original_{0}" SelectMethod="GetSuppliers"
        TypeName="SuppliersBLL">
    </asp:ObjectDataSource>
</EditItemTemplate>
```

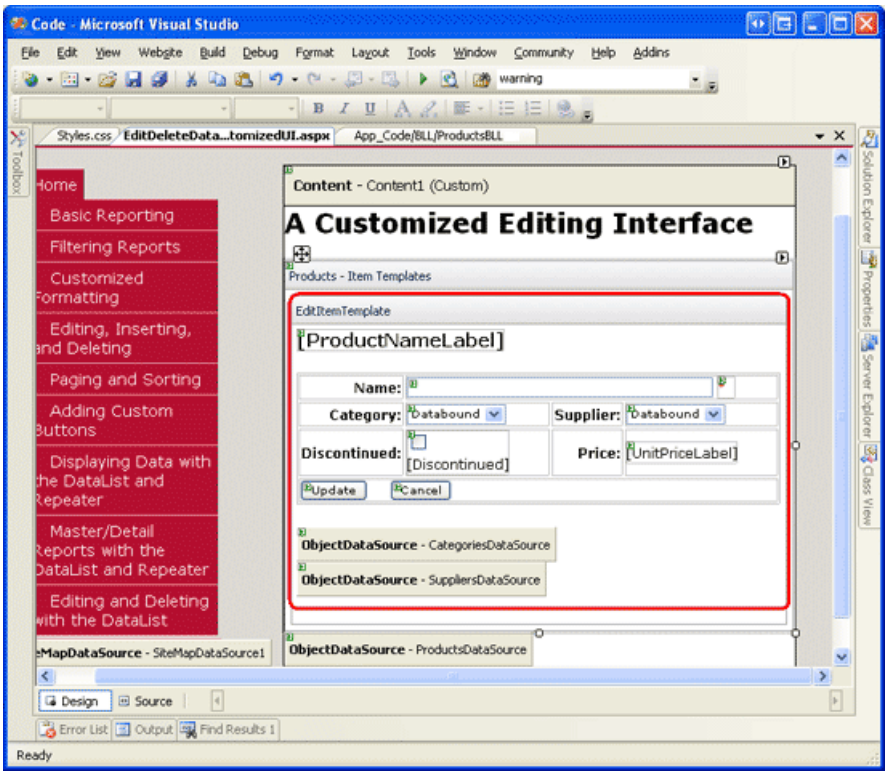


Figure 7: The Editing Interface is Laid Out like the Read-Only Interface (Click to view full-size image)

Step 3: Creating the EditCommand and CancelCommand Event Handlers

Currently, there is no databinding syntax in the `EditItemTemplate` (except for the `UnitPriceLabel`, which was copied over verbatim from the `ItemTemplate`). We'll add the databinding syntax momentarily, but first let s create the event handlers for the DataList s `EditCommand` and `CancelCommand` events. Recall that the responsibility of the `EditCommand` event handler is to render the editing interface for the DataList item whose Edit button was clicked, whereas the `CancelCommand` s job is to return the DataList to its pre-editing state.

Create these two event handlers and have them use the following code:

```
protected void Products_EditCommand(object source, DataListCommandEventArgs e)
{
    // Set the DataList's EditItemIndex property and rebind the data
    Products.EditItemIndex = e.Item.ItemIndex;
    Products.DataBind();
}
protected void Products_CancelCommand(object source, DataListCommandEventArgs e)
{
    // Return to DataList to its pre-editing state
    Products.EditItemIndex = -1;
    Products.DataBind();
}
```

With these two event handlers in place, clicking the Edit button displays the editing interface and clicking the Cancel button returns the edited item to its read-only mode. Figure 8 shows the DataList after the Edit button has been clicked for Chef Anton s Gumbo Mix. Since we ve yet to add any databinding syntax to the editing interface, the `ProductName` TextBox is blank, the `Discontinued` CheckBox unchecked, and the first items selected from the `Categories` and `Suppliers` DropDownLists.

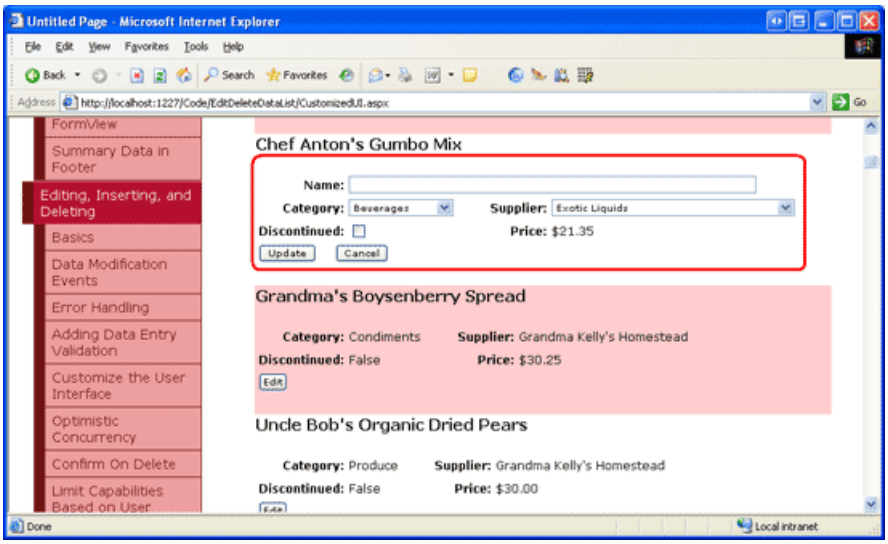


Figure 8: Clicking the Edit Button Displays the Editing Interface (Click to view full-size image)

Step 4: Adding the DataBinding Syntax to the Editing Interface

To have the editing interface display the current product s values, we need to use databinding syntax to assign the data field values to the appropriate Web control values. The databinding syntax can be applied through the Designer by going to the Edit Templates screen and selecting the Edit DataBindings link from the Web controls smart tags. Alternatively, the databinding syntax can be added directly to the declarative markup.

Assign the `ProductName` data field value to the `ProductName` TextBox s `Text` property, the `CategoryID` and `SupplierID` data field values to the `Categories` and `Suppliers` DropDownLists `SelectedValue` properties, and the `Discontinued` data field value to the `Discontinued` CheckBox s `Checked` property. After making these changes, either through the Designer or directly through the declarative markup, revisit the page through a browser and click the Edit button for Chef Anton s Gumbo Mix. As Figure 9 shows, the databinding syntax has added the current values into the TextBox, DropDownLists, and CheckBox.

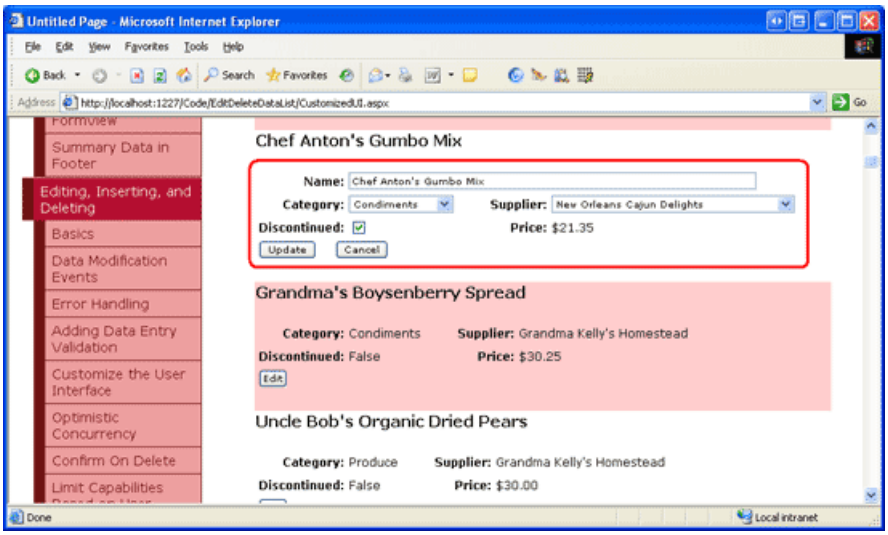


Figure 9: Clicking the Edit Button Displays the Editing Interface (Click to view full-size image)

Step 5: Saving the User s Changes in the UpdateCommand Event Handler

When the user edits a product and clicks the Update button, a postback occurs and the DataList s `UpdateCommand` event fires. In the event handler, we need to read the values from the Web controls in the `EditItemTemplate` and interface with the BLL to update the product in the database. As we ve seen in previous tutorials, the `ProductID` of

the updated product is accessible through the `DataKeys` collection. The user-entered fields are accessed by programmatically referencing the Web controls using `FindControl("controlID")`, as the following code shows:

```
protected void Products_UpdateCommand(object source, DataListCommandEventArgs e)
{
    // Make sure the page is valid...
    if (!Page.IsValid)
        return;
    // Read in the ProductID from the DataKeys collection
    int productID = Convert.ToInt32(Products.DataKeys[e.Item.ItemIndex]);
    // Read in the product name and price values
    TextBox productName = (TextBox)e.Item.FindControl("ProductName");
    DropDownList categories = (DropDownList)e.Item.FindControl("Categories");
    DropDownList suppliers = (DropDownList)e.Item.FindControl("Suppliers");
    CheckBox discontinued = (CheckBox)e.Item.FindControl("Discontinued");
    string productNameValue = null;
    if (productName.Text.Trim().Length > 0)
        productNameValue = productName.Text.Trim();
    int categoryIDValue = Convert.ToInt32(categories.SelectedValue);
    int supplierIDValue = Convert.ToInt32(suppliers.SelectedValue);
    bool discontinuedValue = discontinued.Checked;
    // Call the ProductsBLL's UpdateProduct method...
    ProductsBLL productsAPI = new ProductsBLL();
    productsAPI.UpdateProduct(productNameValue, categoryIDValue, supplierIDValue,
                              discontinuedValue, productID);
    // Revert the DataList back to its pre-editing state
    Products.EditItemIndex = -1;
    Products.DataBind();
}
```

The code starts out by consulting the `Page.IsValid` property to ensure that all the validation controls on the page are valid. If `Page.IsValid` is `True`, the edited product s `ProductID` value is read from the `DataKeys` collection and the data entry Web controls in the `EditItemTemplate` are programmatically referenced. Next, the values from these Web controls are read into variables that are then passed into the appropriate `UpdateProduct` overload. After updating the data, the DataList is returned to its pre-editing state.

> **NOTE**
> I ve omitted the exception handling logic added in the Handling BLL- and DAL-Level Exceptions tutorial in order to keep the code and this example focused. As an exercise, add this functionality after completing this tutorial.

Step 6: Handling NULL CategoryID and SupplierID Values

The Northwind database allows for `NULL` values for the `Products` table s `CategoryID` and `SupplierID` columns. However, our editing interface doesn t currently accommodate `NULL` values. If we attempt to edit a product that has a `NULL` value for either its `CategoryID` or `SupplierID` columns, we'll get an `ArgumentOutOfRangeException` with an error message similar to: *'Categories' has a SelectedValue which is invalid because it does not exist in the list of items.* Also, there s currently no way to change a product s category or supplier value from a non-`NULL` value to a `NULL` one.

To support `NULL` values for the category and supplier DropDownLists, we need to add an additional `ListItem`. I ve chosen to use (None) as the `Text` value for this `ListItem`, but you can change it to something else (such as an empty string) if you d like. Finally, remember to set the DropDownLists `AppendDataBoundItems` to `True`; if you forget to do so, the categories and suppliers bound to the DropDownList will overwrite the statically-added `ListItem`.

After making these changes, the DropDownLists markup in the DataList s `EditItemTemplate` should look similar to the following:

```
<asp:DropDownList ID="Categories" DataSourceID="CategoriesDataSource"
    DataTextField="CategoryName" DataValueField="CategoryID" runat="server"
    SelectedValue='<%# Eval("CategoryID") %>' AppendDataBoundItems="True">
    <asp:ListItem Value=" Selected="True">(None)</asp:ListItem>
</asp:DropDownList>
...
<asp:DropDownList ID="Suppliers" DataSourceID="SuppliersDataSource"
    DataTextField="CompanyName" DataValueField="SupplierID" runat="server"
    SelectedValue='<%# Eval("SupplierID") %>' AppendDataBoundItems="True">
    <asp:ListItem Value=" Selected="True">(None)</asp:ListItem>
</asp:DropDownList>
```

NOTE

Static `ListItem` s can be added to a DropDownList through the Designer or directly through the declarative syntax. When adding a DropDownList item to represent a database `NULL` value, be sure to add the `ListItem` through the declarative syntax. If you use the `ListItem` Collection Editor in the Designer, the generated declarative syntax will omit the `Value` setting altogether when assigned a blank string, creating declarative markup like: `<asp:ListItem>(None)</asp:ListItem>`. While this may look harmless, the missing `Value` causes the DropDownList to use the `Text` property value in its place. That means that if this `NULL` `ListItem` is selected, the value (None) will be attempted to be assigned to the product data field (`CategoryID` or `SupplierID` , in this tutorial), which will result in an exception. By explicitly setting `Value=""` , a `NULL` value will be assigned to the product data field when the `NULL` `ListItem` is selected.

Take a moment to view our progress through a browser. When editing a product, note that the `Categories` and `Suppliers` DropDownLists both have a (None) option at the start of the DropDownList.

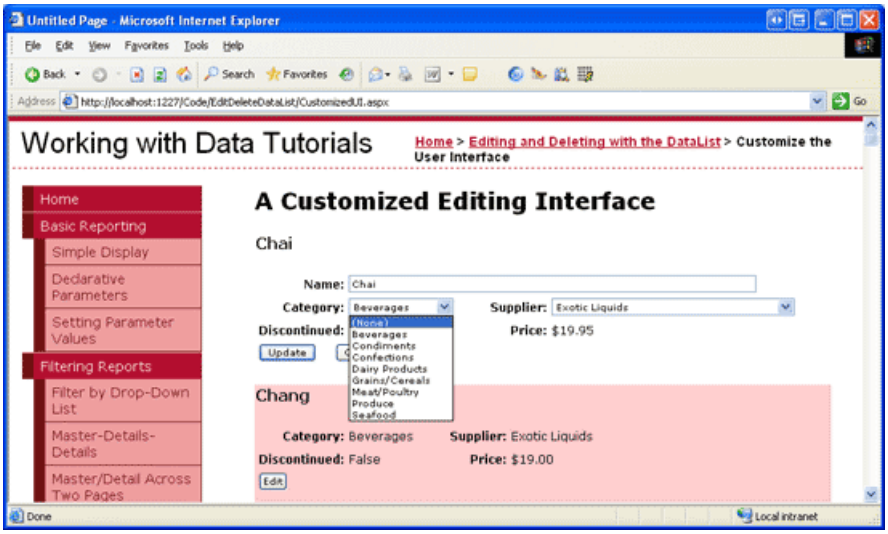


Figure 10: The `Categories` and `Suppliers` DropDownLists include a (None) Option (Click to view full-size image)

To save the (None) option as a database `NULL` value, we need to return to the `UpdateCommand` event handler. Change the `categoryIDValue` and `supplierIDValue` variables to be nullable integers and assign them a value other than `Nothing` only if the DropDownList s `SelectedValue` is not an empty string:

```
int? categoryIDValue = null;
if (!string.IsNullOrEmpty(categories.SelectedValue))
    categoryIDValue = Convert.ToInt32(categories.SelectedValue);
int? supplierIDValue = null;
if (!string.IsNullOrEmpty(suppliers.SelectedValue))
    supplierIDValue = Convert.ToInt32(suppliers.SelectedValue);
```

With this change, a value of `Nothing` will be passed into the `UpdateProduct` BLL method if the user selected the

(None) option from either of the drop-down lists, which corresponds to a `NULL` database value.

Summary

In this tutorial we saw how to create a more complex DataList editing interface that included three different input Web controls a TextBox, two DropDownLists, and a CheckBox along with validation controls. When building the editing interface, the steps are the same regardless of the Web controls being used: start by adding the Web controls to the DataList s `EditItemTemplate`; use databinding syntax to assign the corresponding data field values with the appropriate Web control properties; and, in the `UpdateCommand` event handler, programmatically access the Web controls and their appropriate properties, passing their values into the BLL.

When creating an editing interface, whether it s composed of just TextBoxes or a collection of different Web controls, be sure to correctly handle database `NULL` values. When accounting for `NULL` s, it is imperative that you not only correctly display an existing `NULL` value in the editing interface, but also that you offer a means for marking a value as `NULL`. For DropDownLists in DataLists, that typically means adding a static `ListItem` whose `Value` property is explicitly set to an empty string (`Value=""`), and adding a bit of code to the `UpdateCommand` event handler to determine whether or not the `NULL``ListItem` was selected.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Dennis Patterson, David Suru, and Randy Schmidt. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

An Overview of Editing and Deleting Data in the DataList (VB)

2/20/2020 • 23 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> While the DataList lacks built-in editing and deleting capabilities, in this tutorial we'll see how to create a DataList that supports editing and deleting of its underlying data.

Introduction

In the An Overview of Inserting, Updating, and Deleting Data tutorial we looked at how to insert, update, and delete data using the application architecture, an ObjectDataSource, and the GridView, DetailsView, and FormView controls. With the ObjectDataSource and these three data Web controls, implementing simple data modification interfaces was a snap and involved merely ticking a checkbox from a smart tag. No code needed to be written.

Unfortunately, the DataList lacks the built-in editing and deleting capabilities inherent in the GridView control. This missing functionality is due in part to the fact that the DataList is a relic from the previous version of ASP.NET, when declarative data source controls and code-free data modification pages were unavailable. While the DataList in ASP.NET 2.0 does not offer the same out of the box data modification capabilities as the GridView, we can use ASP.NET 1.x techniques to include such functionality. This approach requires a bit of code but, as we'll see in this tutorial, the DataList has some events and properties in place to aid in this process.

In this tutorial we'll see how to create a DataList that supports editing and deleting of its underlying data. Future tutorials will examine more advanced editing and deleting scenarios, including input field validation, gracefully handling exceptions raised from the Data Access or Business Logic Layers, and so on.

> **NOTE**
> Like the DataList, the Repeater control lacks the out of the box functionality for inserting, updating, or deleting. While such functionality can be added, the DataList includes properties and events not found in the Repeater that simplify adding such capabilities. Therefore, this tutorial and future ones that look at editing and deleting will focus strictly on the DataList.

Step 1: Creating the Editing and Deleting Tutorials Web Pages

Before we start exploring how to update and delete data from a DataList, let s first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial and the next several ones. Start by adding a new folder named `EditDeleteDataList`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `Basics.aspx`
- `BatchUpdate.aspx`
- `ErrorHandling.aspx`
- `UIValidation.aspx`
- `CustomizedUI.aspx`

- `OptimisticConcurrency.aspx`
- `ConfirmationOnDelete.aspx`
- `UserLevelAccess.aspx`

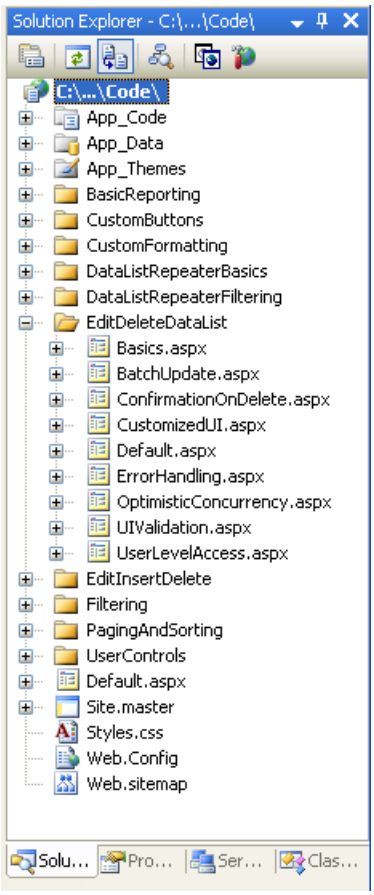


Figure 1: Add the ASP.NET Pages for the Tutorials

Like in the other folders, `Default.aspx` in the `EditDeleteDataList` folder lists the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

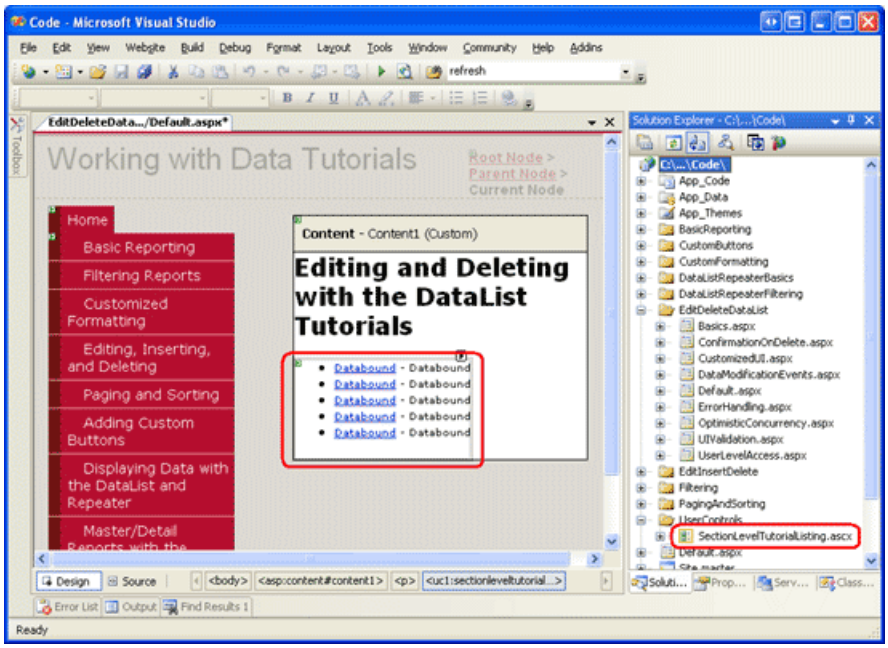


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add the pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the

Master/Detail Reports with the DataList and Repeater `<siteMapNode>`:

```
<siteMapNode
    title="Editing and Deleting with the DataList"
    description="Samples of Reports that Provide Editing and Deleting Capabilities"
    url="~/EditDeleteDataList/Default.aspx" >
    <siteMapNode
        title="Basics"
        description="Examines the basics of editing and deleting with the
                     DataList control."
        url="~/EditDeleteDataList/Basics.aspx" />
    <siteMapNode
        title="Batch Update"
        description="Examines how to update multiple records at once in a
                     fully-editable DataList."
        url="~/EditDeleteDataList/BatchUpdate.aspx" />
    <siteMapNode
        title="Error Handling"
        description="Learn how to gracefully handle exceptions raised during the
                     data modification workflow."
        url="~/EditDeleteDataList/ErrorHandling.aspx" />
    <siteMapNode
        title="Adding Data Entry Validation"
        description="Help prevent data entry errors by providing validation."
        url="~/EditDeleteDataList/UIValidation.aspx" />
    <siteMapNode
        title="Customize the User Interface"
        description="Customize the editing user interfaces."
        url="~/EditDeleteDataList/CustomizedUI.aspx" />
    <siteMapNode
        title="Optimistic Concurrency"
        description="Learn how to help prevent simultaneous users from
                     overwritting one another s changes."
        url="~/EditDeleteDataList/OptimisticConcurrency.aspx" />
    <siteMapNode
        title="Confirm On Delete"
        description="Prompt a user for confirmation when deleting a record."
        url="~/EditDeleteDataList/ConfirmationOnDelete.aspx" />
    <siteMapNode
        title="Limit Capabilities Based on User"
        description="Learn how to limit the data modification functionality
                     based on the user s role or permissions."
        url="~/EditDeleteDataList/UserLevelAccess.aspx" />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the DataList editing and deleting tutorials.

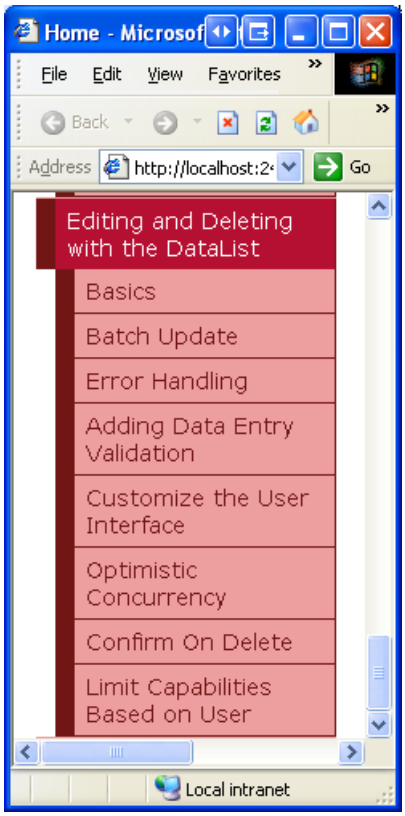


Figure 3: The Site Map Now Includes Entries for the DataList Editing and Deleting Tutorials

Step 2: Examining Techniques for Updating and Deleting Data

Editing and deleting data with the GridView is so easy because, underneath the covers, the GridView and ObjectDataSource work in concert. As discussed in the Examining the Events Associated with Inserting, Updating, and Deleting tutorial, when a row s Update button is clicked, the GridView automatically assigns its fields that used two-way databinding to the `UpdateParameters` collection of its ObjectDataSource and then invokes that ObjectDataSource s `Update()` method.

Sadly, the DataList does not provide any of this built-in functionality. It is our responsibility to ensure that the user s values are assigned to the ObjectDataSource s parameters and that its `Update()` method is called. To aid us in this endeavor, the DataList provides the following properties and events:

- **The** `DataKeyField` **property** when updating or deleting, we need to be able to uniquely identify each item in the DataList. Set this property to the primary key field of the displayed data. Doing so will populate the DataList s `DataKeys` collection with the specified `DataKeyField` value for each DataList item.
- **The** `EditCommand` **event** fires when a Button, LinkButton, or ImageButton whose `CommandName` property is set to Edit is clicked.
- **The** `CancelCommand` **event** fires when a Button, LinkButton, or ImageButton whose `CommandName` property is set to Cancel is clicked.
- **The** `UpdateCommand` **event** fires when a Button, LinkButton, or ImageButton whose `CommandName` property is set to Update is clicked.
- **The** `DeleteCommand` **event** fires when a Button, LinkButton, or ImageButton whose `CommandName` property is set to Delete is clicked.

Using these properties and events, there are four approaches we can use to update and delete data from the DataList:

1. **Using ASP.NET 1.x Techniques** the DataList existed prior to ASP.NET 2.0 and ObjectDataSources and was able to update and delete data entirely through programmatic means. This technique ditches the ObjectDataSource altogether and requires that we bind the data to the DataList directly from the Business

Logic Layer, both in retrieving the data to display and when updating or deleting a record.

2. **Using a Single ObjectDataSource Control on the Page for Selecting, Updating, and Deleting** while the DataList lacks the GridView s inherent editing and deleting capabilities, there s no reason we can t add them in ourselves. With this approach, we use an ObjectDataSource just like in the GridView examples, but must create an event handler for the DataList s `UpdateCommand` event where we set the ObjectDataSource s parameters and call its `Update()` method.
3. **Using an ObjectDataSource Control for Selecting, but Updating and Deleting Directly Against the BLL** when using option 2, we need to write a bit of code in the `UpdateCommand` event, assigning parameter values and so on. Instead, we can stick with using the ObjectDataSource for selecting, but make the updating and deleting calls directly against the BLL (like with option 1). In my opinion, updating data by interfacing directly with the BLL leads to more readable code than assigning the ObjectDataSource s `UpdateParameters` and calling its `Update()` method.
4. **Using Declarative Means through Multiple ObjectDataSources** the previous three approaches all require a bit of code. If you d rather keep using as much declarative syntax as possible, a final option is to include multiple ObjectDataSources on the page. The first ObjectDataSource retrieves the data from the BLL and binds it to the DataList. For updating, another ObjectDataSource is added, but added directly within the DataList s `EditItemTemplate`. To include deleting support, yet another ObjectDataSource would be needed in the `ItemTemplate`. With this approach, these embedded ObjectDataSource s use `ControlParameters` to declaratively bind the ObjectDataSource s parameters to the user input controls (rather than having to specify them programmatically in the DataList s `UpdateCommand` event handler). This approach still requires a bit of code we need to call the embedded ObjectDataSource s `Update()` or `Delete()` command but requires far less than with the other three approaches. The downside here is that the multiple ObjectDataSources do clutter up the page, detracting from the overall readability.

If forced to only ever use one of these approaches, I d choose option 1 because it provides the most flexibility and because the DataList was originally designed to accommodate this pattern. While the DataList was extended to work with the ASP.NET 2.0 data source controls, it does not have all of the extensibility points or features of the official ASP.NET 2.0 data Web controls (the GridView, DetailsView, and FormView). Options 2 through 4 are not without merit, though.

This and the future editing and deleting tutorials will use an ObjectDataSource for retrieving the data to display and direct calls to the BLL to update and delete data (option 3).

Step 3: Adding the DataList and Configuring its ObjectDataSource

In this tutorial we will create a DataList that lists product information and, for each product, provides the user the ability to edit the name and price and to delete the product altogether. In particular, we will retrieve the records to display using an ObjectDataSource, but perform the update and delete actions by interfacing directly with the BLL. Before we worry about implementing the editing and deleting capabilities to the DataList, let s first get the page to display the products in a read-only interface. Since we ve examined these steps in previous tutorials, I'll proceed through them quickly.

Start by opening the `Basics.aspx` page in the `EditDeleteDataList` folder and, from the Design view, add a DataList to the page. Next, from the DataList s smart tag, create a new ObjectDataSource. Since we are working with product data, configure it to use the `ProductsBLL` class. To retrieve *all* products, choose the `GetProducts()` method in the SELECT tab.

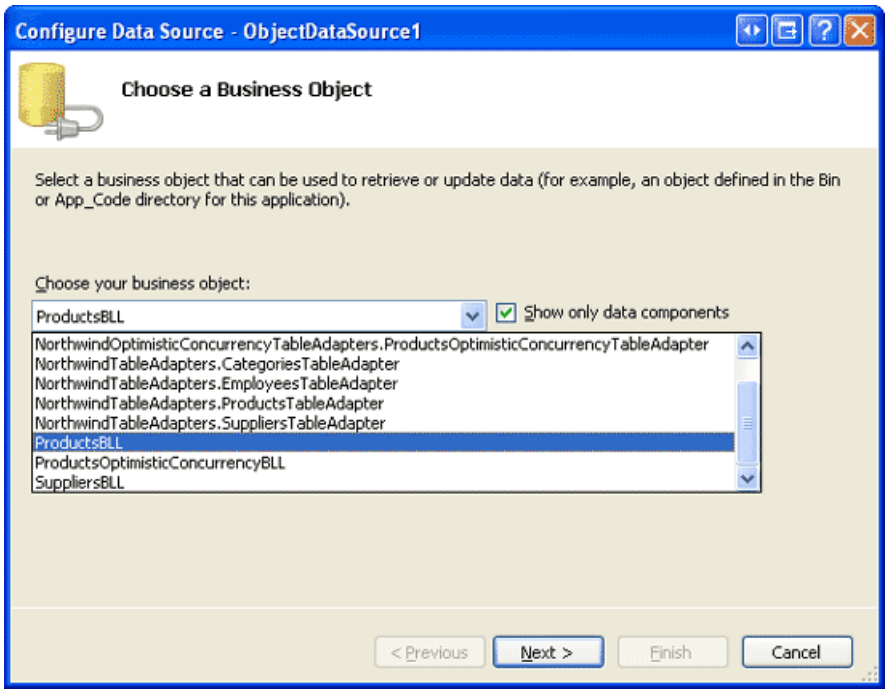


Figure 4: Configure the ObjectDataSource to Use the `ProductsBLL` Class (Click to view full-size image)

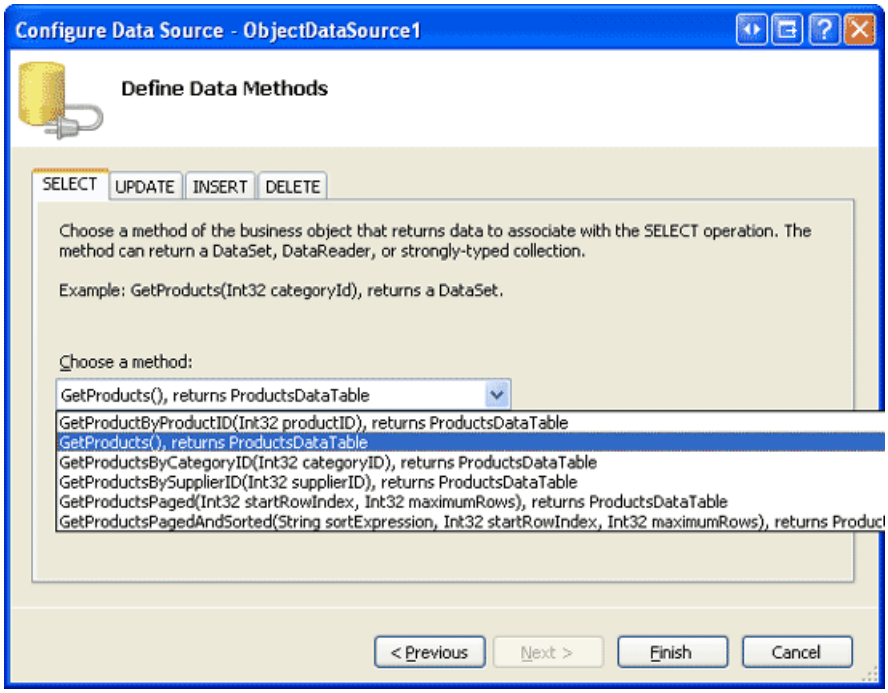


Figure 5: Return the Product Information Using the `GetProducts()` Method (Click to view full-size image)

The DataList, like the GridView, is not designed for inserting new data; therefore, select the (None) option from the drop-down list in the INSERT tab. Also choose (None) for the UPDATE and DELETE tabs since the updates and deletes will be performed programmatically through the BLL.

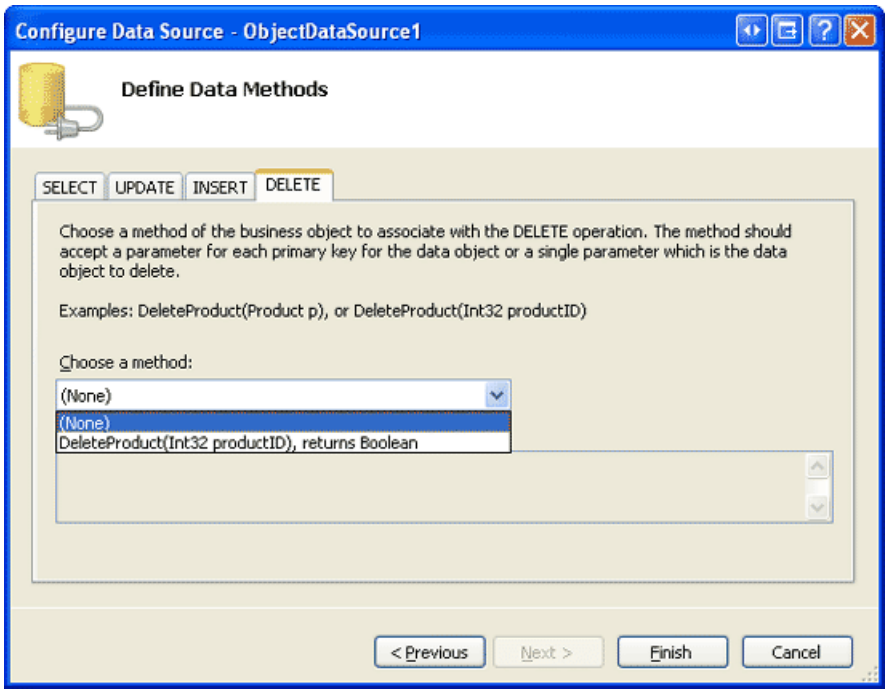


Figure 6: Confirm that the Drop-Down Lists in the ObjectDataSource s INSERT, UPDATE, and DELETE Tabs are Set to (None) (Click to view full-size image)

After configuring the ObjectDataSource, click Finish, returning to the Designer. As we ve seen in past examples, when completing the ObjectDataSource configuration, Visual Studio automatically creates an `ItemTemplate` for the DropDownList, displaying each of the data fields. Replace this `ItemTemplate` with one that displays only the product s name and price. Also, set the `RepeatColumns` property to 2.

> **NOTE**
> As discussed in the *Overview of Inserting, Updating, and Deleting Data* tutorial, when modifying data using the ObjectDataSource our architecture requires that we remove the `OldValuesParameterFormatString` property from the ObjectDataSource s declarative markup (or reset it to its default value, `{0}`). In this tutorial, however, we are using the ObjectDataSource only for retrieving data. Therefore, we do not need to modify the ObjectDataSource s `OldValuesParameterFormatString` property value (although it doesn t hurt to do so).

After replacing the default DataList `ItemTemplate` with a customized one, the declarative markup on your page should look similar to the following:

```
<asp:DataList ID="DataList1" runat="server" DataKeyField="ProductID"
    DataSourceID="ObjectDataSource1" RepeatColumns="2">
    <ItemTemplate>
        <h5>
            <asp:Label runat="server" ID="ProductNameLabel"
                Text='<%# Eval("ProductName") %>'></asp:Label>
        </h5>
        Price: <asp:Label runat="server" ID="Label1"
                    Text='<%# Eval("UnitPrice", "{0:C}") %>' />
        <br />
        <br />
    </ItemTemplate>
</asp:DataList>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    SelectMethod="GetProducts" TypeName="ProductsBLL"
    OldValuesParameterFormatString="original_{0}">
</asp:ObjectDataSource>
```

Take a moment to view our progress through a browser. As Figure 7 shows, the DataList displays the product

name and unit price for each product in two columns.

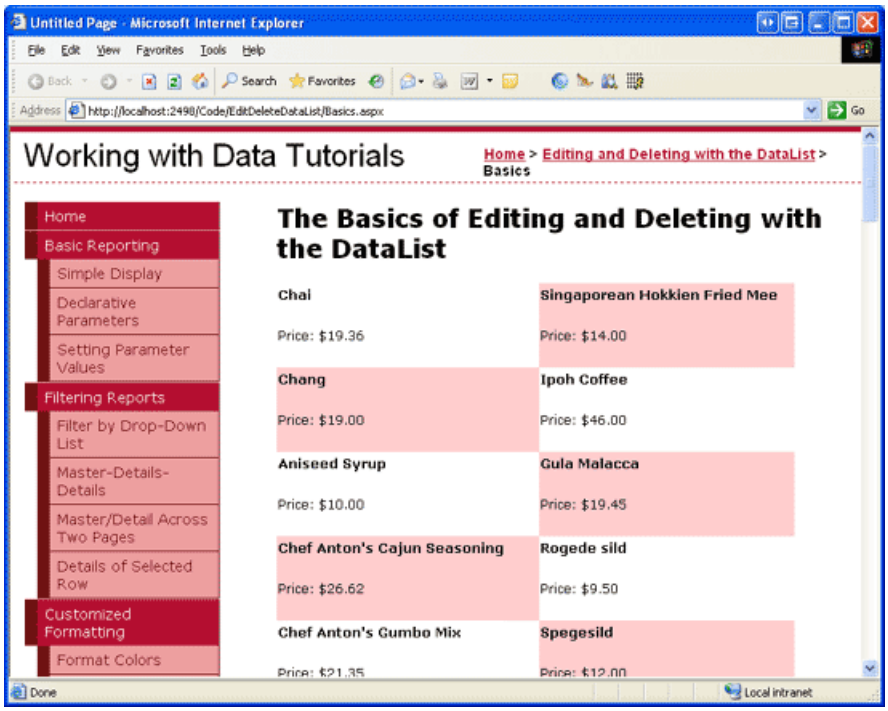


Figure 7: The Products Names and Prices are Displayed in a Two-Column DataList (Click to view full-size image)

> **NOTE**
>
> The DataList has a number of properties that are required for the updating and deleting process, and these values are stored in view state. Therefore, when building a DataList that supports editing or deleting data, it is essential that the DataList s view state be enabled.
>
> The astute reader may recall that we were able to disable view state when creating editable GridViews, DetailsViews, and FormViews. This is because ASP.NET 2.0 Web controls can include *control state*, which is state persisted across postbacks like view state, but deemed essential.

Disabling view state in the GridView merely omits trivial state information, but maintains the control state (which includes the state necessary for editing and deleting). The DataList, having been created in the ASP.NET 1.x timeframe, does not utilize control state and therefore must have view state enabled. See Control State vs. View State for more information on the purpose of control state and how it differs from view state.

Step 4: Adding an Editing User Interface

The GridView control is composed of a collection of fields (BoundFields, CheckBoxFields, TemplateFields, and so on). These fields can adjust their rendered markup depending on their mode. For example, when in read-only mode, a BoundField displays its data field value as text; when in edit mode, it renders a TextBox Web control whose `Text` property is assigned the data field value.

The DataList, on the other hand, renders its items using templates. Read-only items are rendered using the `ItemTemplate` whereas items in edit mode are rendered via the `EditItemTemplate`. At this point, our DataList has only an `ItemTemplate`. To support item-level editing functionality we need to add an `EditItemTemplate` that contains the markup to be displayed for the editable item. For this tutorial, we'll use TextBox Web controls for editing the product s name and unit price.

The `EditItemTemplate` can be created either declaratively or through the Designer (by selecting the Edit Templates option from the DataList s smart tag). To use the Edit Templates option, first click the Edit Templates link in the smart tag and then select the `EditItemTemplate` item from the drop-down list.

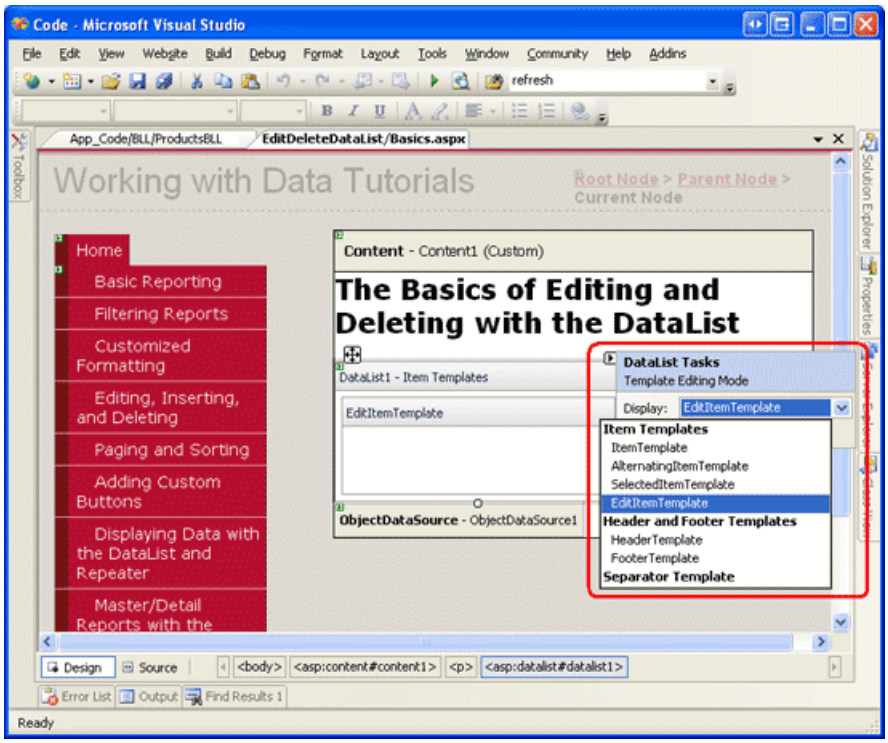


Figure 8: Opt to Work with the DataList s `EditItemTemplate` (Click to view full-size image)

Next, type in Product name: and Price: and then drag two TextBox controls from the Toolbox into the `EditItemTemplate` interface on the Designer. Set the TextBoxes `ID` properties to `ProductName` and `UnitPrice`.

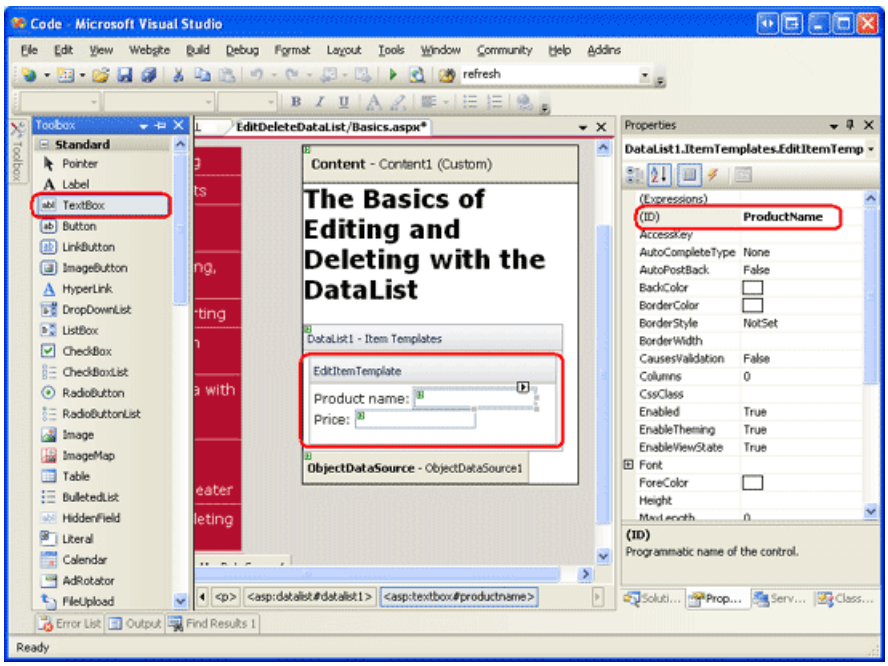


Figure 9: Add a TextBox for the Product s Name and Price (Click to view full-size image)

We need to bind the corresponding product data field values to the `Text` properties of the two TextBoxes. From the TextBoxes smart tags, click on the Edit DataBindings link and then associate the appropriate data field with the `Text` property, as shown in Figure 10.

> **NOTE**
>
> When binding the `UnitPrice` data field to the price TextBox s `Text` field, you may format it as a currency value (`{0:C}`), a general number (`{0:N}`), or leave it unformatted.

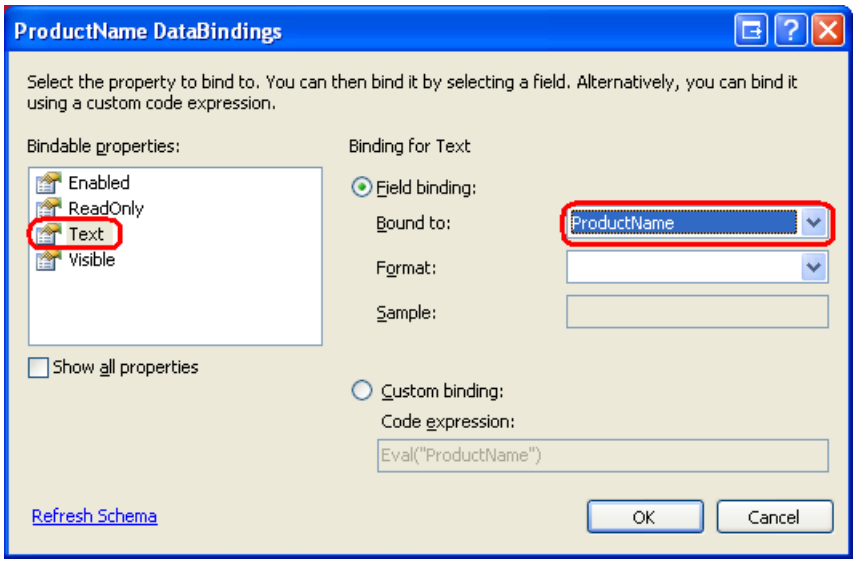


Figure 10: Bind the `ProductName` and `UnitPrice` Data Fields to the `Text` Properties of the TextBoxes

Notice how the Edit DataBindings dialog box in Figure 10 does *not* include the Two-way databinding checkbox that is present when editing a TemplateField in the GridView or DetailsView, or a template in the FormView. The two-way databinding feature allowed the value entered into the input Web control to be automatically assigned to the corresponding ObjectDataSource s `InsertParameters` or `UpdateParameters` when inserting or updating data. The DataList does not support two-way databinding as we'll see later on in this tutorial, after the user makes her changes and is ready to update the data, we will need to programmatically access these TextBoxes `Text` properties and pass in their values to the appropriate `UpdateProduct` method in the `ProductsBLL` class.

Finally, we need to add Update and Cancel buttons to the `EditItemTemplate`. As we saw in the Master/Detail Using a Bulleted List of Master Records with a Details DataList tutorial, when a Button, LinkButton, or ImageButton whose `CommandName` property is set is clicked from within a Repeater or DataList, the Repeater or DataList s `ItemCommand` event is raised. For the DataList, if the `CommandName` property is set to a certain value, an additional event may be raised as well. The special `CommandName` property values include, among others:

- Cancel raises the `CancelCommand` event
- Edit raises the `EditCommand` event
- Update raises the `UpdateCommand` event

Keep in mind that these events are raised *in addition to* the `ItemCommand` event.

Add to the `EditItemTemplate` two Button Web controls, one whose `CommandName` is set to Update and the other s set to Cancel. After adding these two Button Web controls the Designer should look similar to the following:

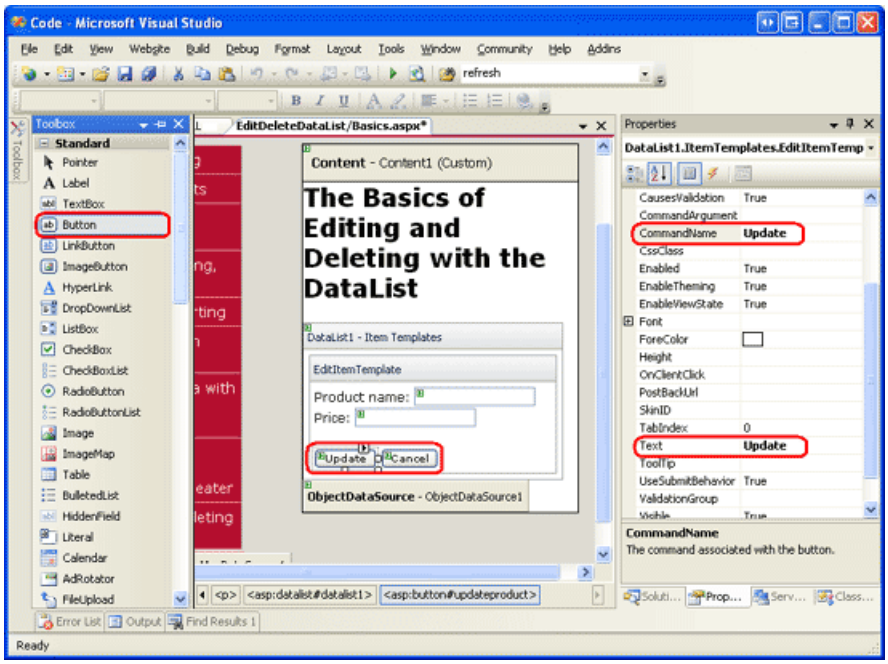


Figure 11: Add Update and Cancel Buttons to the `EditItemTemplate` (Click to view full-size image)

With the `EditItemTemplate` complete your DataList s declarative markup should look similar to the following:

```
<asp:DataList ID="DataList1" runat="server" DataKeyField="ProductID"
    DataSourceID="ObjectDataSource1" RepeatColumns="2">
    <ItemTemplate>
        <h5>
            <asp:Label runat="server" ID="ProductNameLabel"
                Text='<%# Eval("ProductName") %>' />
        </h5>
        Price: <asp:Label runat="server" ID="Label1"
                    Text='<%# Eval("UnitPrice", "{0:C}") %>' />
        <br />
        <br />
    </ItemTemplate>
    <EditItemTemplate>
        Product name:
            <asp:TextBox ID="ProductName" runat="server"
                Text='<%# Eval("ProductName") %>' /><br />
        Price:
            <asp:TextBox ID="UnitPrice" runat="server"
                Text='<%# Eval("UnitPrice", "{0:C}") %>' /><br />
        <br />
        <asp:Button ID="UpdateProduct" runat="server"
            CommandName="Update" Text="Update" />
        <asp:Button ID="CancelUpdate" runat="server"
            CommandName="Cancel" Text="Cancel" />
    </EditItemTemplate>
</asp:DataList>
```

Step 5: Adding the Plumbing to Enter Edit Mode

At this point our DataList has an editing interface defined via its `EditItemTemplate`; however, there s currently no way for a user visiting our page to indicate that he wants to edit a product s information. We need to add an Edit button to each product that, when clicked, renders that DataList item in edit mode. Start by adding an Edit button to the `ItemTemplate`, either through the Designer or declaratively. Be certain to set the Edit button s `CommandName` property to Edit .

After you have added this Edit button, take a moment to view the page through a browser. With this addition, each product listing should include an Edit button.

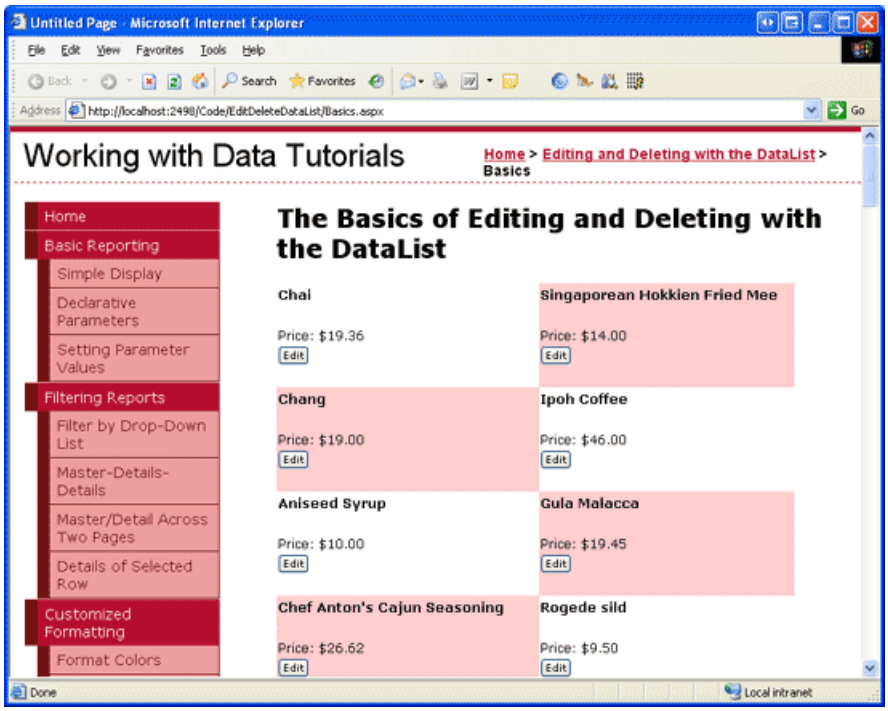


Figure 12: Add Update and Cancel Buttons to the `EditItemTemplate` (Click to view full-size image)

Clicking the button causes a postback, but does *not* bring the product listing into edit mode. To make the product editable, we need to:

1. Set the DataList s `EditItemIndex` property to the index of the `DataListItem` whose Edit button was just clicked.
2. Rebind the data to the DataList. When the DataList is re-rendered, the `DataListItem` whose `ItemIndex` corresponds with the DataList s `EditItemIndex` will render using its `EditItemTemplate`.

Since the DataList s `EditCommand` event is fired when the Edit button is clicked, create an `EditCommand` event handler with the following code:

```
Protected Sub DataList1_EditCommand(source As Object, e As DataListCommandEventArgs) _
    Handles DataList1.EditCommand
    ' Set the DataList's EditItemIndex property to the
    ' index of the DataListItem that was clicked
    DataList1.EditItemIndex = e.Item.ItemIndex
    ' Rebind the data to the DataList
    DataList1.DataBind()
End Sub
```

The `EditCommand` event handler is passed in an object of type `DataListCommandEventArgs` as its second input parameter, which includes a reference to the `DataListItem` whose Edit button was clicked (`e.Item`). The event handler first sets the DataList s `EditItemIndex` to the `ItemIndex` of the editable `DataListItem` and then rebinds the data to the DataList by calling the DataList s `DataBind()` method.

After adding this event handler, revisit the page in a browser. Clicking the Edit button now makes the clicked product editable (see Figure 13).

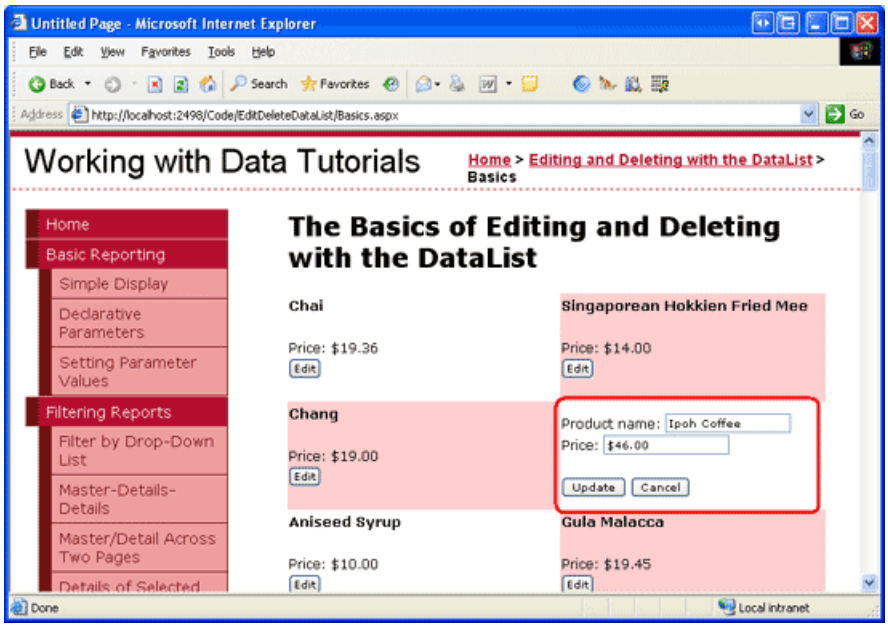


Figure 13: Clicking the Edit Button Makes the Product Editable (Click to view full-size image)

Step 6: Saving the User s Changes

Clicking the edited product s Update or Cancel buttons does nothing at this point; to add this functionality we need to create event handlers for the DataList s `UpdateCommand` and `CancelCommand` events. Start by creating the `CancelCommand` event handler, which will execute when the edited product s Cancel button is clicked and it tasked with returning the DataList to its pre-editing state.

To have the DataList render all of its items in the read-only mode, we need to:

1. Set the DataList s `EditItemIndex` property to the index of a non-existent `DataListItem` index. `-1` is a safe choice, since the `DataListItem` indexes start at `0`.
2. Rebind the data to the DataList. Since no `DataListItem` `ItemIndex` es correspond to the DataList s `EditItemIndex`, the entire DataList will be rendered in a read-only mode.

These steps can be accomplished with the following event handler code:

```
Protected Sub DataList1_CancelCommand(source As Object, e As DataListCommandEventArgs) _
    Handles DataList1.CancelCommand
    ' Set the DataList's EditItemIndex property to -1
    DataList1.EditItemIndex = -1
    ' Rebind the data to the DataList
    DataList1.DataBind()
End Sub
```

With this addition, clicking the Cancel button returns the DataList to its pre-editing state.

The last event handler we need to complete is the `UpdateCommand` event handler. This event handler needs to:

1. Programmatically access the user-entered product name and price as well as the edited product s `ProductID`.
2. Initiate the update process by calling the appropriate `UpdateProduct` overload in the `ProductsBLL` class.
3. Set the DataList s `EditItemIndex` property to the index of a non-existent `DataListItem` index. `-1` is a safe choice, since the `DataListItem` indexes start at `0`.
4. Rebind the data to the DataList. Since no `DataListItem` `ItemIndex` es correspond to the DataList s `EditItemIndex`, the entire DataList will be rendered in a read-only mode.

Steps 1 and 2 are responsible for saving the user s changes; steps 3 and 4 return the DataList to its pre-editing

state after the changes have been saved and are identical to the steps performed in the `CancelCommand` event handler.

To get the updated product name and price, we need to use the `FindControl` method to programmatically reference the TextBox Web controls within the `EditItemTemplate`. We also need to get the edited product s `ProductID` value. When we initially bound the ObjectDataSource to the DataList, Visual Studio assigned the DataList s `DataKeyField` property to the primary key value from the data source (`ProductID`). This value can then be retrieved from the DataList s `DataKeys` collection. Take a moment to ensure that the `DataKeyField` property is indeed set to `ProductID`.

The following code implements the four steps:

```
Protected Sub DataList1_UpdateCommand(source As Object, e As DataListCommandEventArgs) _
    Handles DataList1.UpdateCommand
    ' Read in the ProductID from the DataKeys collection
    Dim productID As Integer = Convert.ToInt32(DataList1.DataKeys(e.Item.ItemIndex))
    ' Read in the product name and price values
    Dim productName As TextBox = CType(e.Item.FindControl("ProductName"), TextBox)
    Dim unitPrice As TextBox = CType(e.Item.FindControl("UnitPrice"), TextBox)
    Dim productNameValue As String = Nothing
    If productName.Text.Trim().Length > 0 Then
        productNameValue = productName.Text.Trim()
    End If
    Dim unitPriceValue As Nullable(Of Decimal) = Nothing
    If unitPrice.Text.Trim().Length > 0 Then
        unitPriceValue = Decimal.Parse(unitPrice.Text.Trim(), NumberStyles.Currency)
    End If
    ' Call the ProductsBLL's UpdateProduct method...
    Dim productsAPI As New ProductsBLL()
    productsAPI.UpdateProduct(productNameValue, unitPriceValue, productID)
    ' Revert the DataList back to its pre-editing state
    DataList1.EditItemIndex = -1
    DataList1.DataBind()
End Sub
```

The event handler starts by reading in the edited product s `ProductID` from the `DataKeys` collection. Next, the two TextBoxes in the `EditItemTemplate` are referenced and their `Text` properties stored in local variables, `productNameValue` and `unitPriceValue`. We use the `Decimal.Parse()` method to read the value from the `UnitPrice` TextBox so that if the value entered has a currency symbol, it can still be correctly converted into a `Decimal` value.

> **NOTE**
> The values from the `ProductName` and `UnitPrice` TextBoxes are only assigned to the productNameValue and unitPriceValue variables if the TextBoxes Text properties have a value specified. Otherwise, a value of `Nothing` is used for the variables, which has the effect of updating the data with a database `NULL` value. That is, our code treats converts empty strings to database `NULL` values, which is the default behavior of the editing interface in the GridView, DetailsView, and FormView controls.

After reading the values, the `ProductsBLL` class s `UpdateProduct` method is called, passing in the product s name, price, and `ProductID`. The event handler completes by returning the DataList to its pre-editing state using the exact same logic as in the `CancelCommand` event handler.

With the `EditCommand`, `CancelCommand`, and `UpdateCommand` event handlers complete, a visitor can edit the name and price of a product. Figures 14-16 show this editing workflow in action.

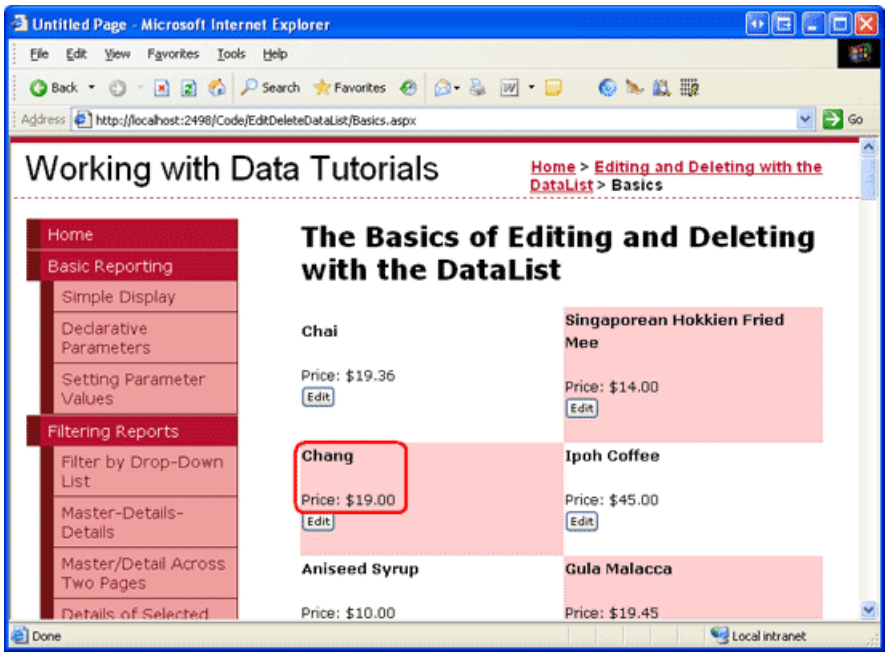


Figure 14: When First Visiting the Page, All Products are in Read-Only Mode (Click to view full-size image)

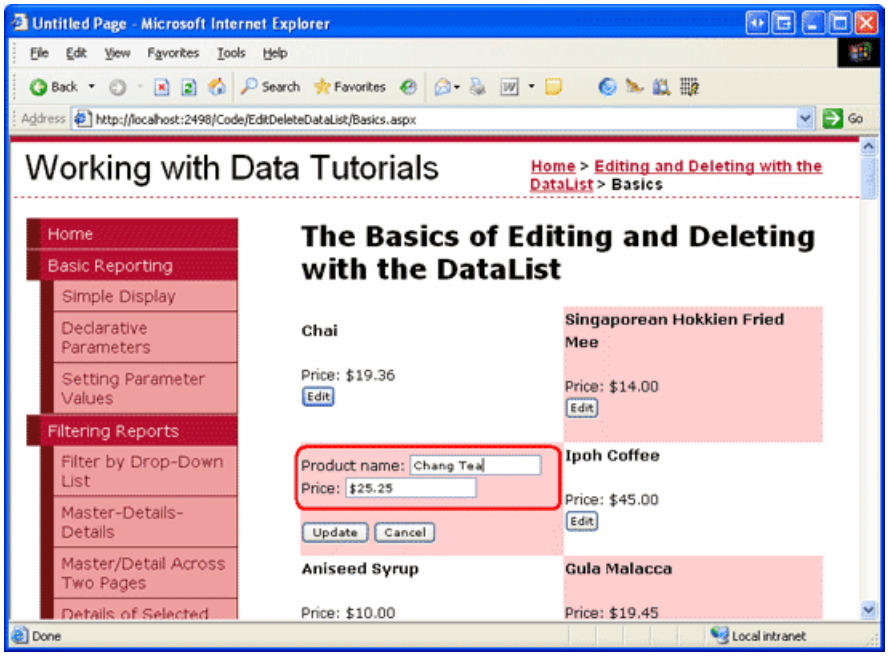


Figure 15: To Update a Product s Name or Price, Click the Edit Button (Click to view full-size image)

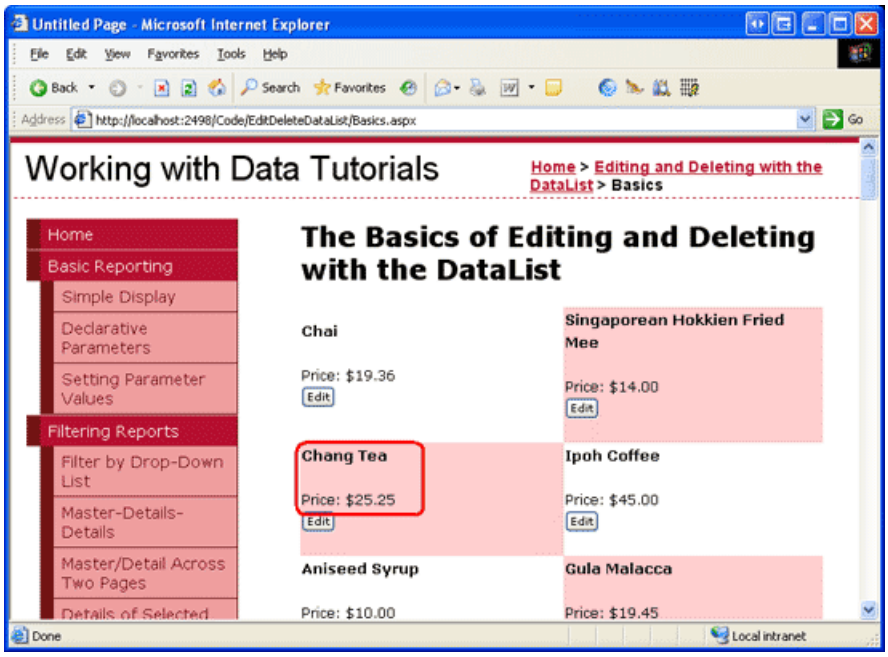


Figure 16: After Changing the Value, Click Update to Return to the Read-Only Mode (Click to view full-size image)

Step 7: Adding Delete Capabilities

The steps for adding delete capabilities to a DataList are similar to those for adding editing capabilities. In short, we need to add a Delete button to the `ItemTemplate` that, when clicked:

1. Reads in the corresponding product s `ProductID` via the `DataKeys` collection.
2. Performs the delete by calling the `ProductsBLL` class s `DeleteProduct` method.
3. Rebinds the data to the DataList.

Let s start by adding a Delete button to the `ItemTemplate`.

When clicked, a Button whose `CommandName` is Edit, Update, or Cancel raises the DataList s `ItemCommand` event along with an additional event (for example, when using Edit the `EditCommand` event is raised as well). Similarly, any Button, LinkButton, or ImageButton in the DataList whose `CommandName` property is set to Delete causes the `DeleteCommand` event to fire (along with `ItemCommand`).

Add a Delete button next to the Edit button in the `ItemTemplate`, setting its `CommandName` property to Delete. After adding this Button control your DataList s `ItemTemplate` declarative syntax should look like:

```
<ItemTemplate>
    <h5>
        <asp:Label runat="server" ID="ProductNameLabel"
            Text='<%# Eval("ProductName") %>' />
    </h5>
    Price: <asp:Label runat="server" ID="Label1"
            Text='<%# Eval("UnitPrice", "{0:C}") %>' />
    <br />
    <asp:Button runat="server" id="EditProduct" CommandName="Edit"
        Text="Edit" />

    <asp:Button runat="server" id="DeleteProduct" CommandName="Delete"
        Text="Delete" />
    <br />
    <br />
</ItemTemplate>
```

Next, create an event handler for the DataList s `DeleteCommand` event, using the following code:

```
Protected Sub DataList1_DeleteCommand(source As Object, e As DataListCommandEventArgs) _
    Handles DataList1.DeleteCommand
    ' Read in the ProductID from the DataKeys collection
    Dim productID As Integer = Convert.ToInt32(DataList1.DataKeys(e.Item.ItemIndex))
    ' Delete the data
    Dim productsAPI As New ProductsBLL()
    productsAPI.DeleteProduct(productID)
    ' Rebind the data to the DataList
    DataList1.DataBind()
End Sub
```

Clicking the Delete button causes a postback and fires the DataList s `DeleteCommand` event. In the event handler, the clicked product s `ProductID` value is accessed from the `DataKeys` collection. Next, the product is deleted by calling the `ProductsBLL` class s `DeleteProduct` method.

After deleting the product, it s important that we rebind the data to the DataList (`DataList1.DataBind()`), otherwise the DataList will continue to show the product that was just deleted.

Summary

While the DataList lacks the point and click editing and deleting support enjoyed by the GridView, with a short bit of code it can be enhanced to include these features. In this tutorial we saw how to create a two-column listing of products that could be deleted and whose name and price could be edited. Adding editing and deleting support is a matter of including the appropriate Web controls in the `ItemTemplate` and `EditItemTemplate`, creating the corresponding event handlers, reading the user-entered and primary key values, and interfacing with the Business Logic Layer.

While we have added basic editing and deleting capabilities to the DataList, it lacks more advanced features. For example, there is no input field validation - if a user enters a price of Too expensive, an exception will be thrown by `Decimal.Parse` when attempting to convert Too expensive into a `Decimal`. Similarly, if there is a problem in updating the data at the Business Logic or Data Access Layers the user will be presented with the standard error screen. Without any sort of confirmation on the Delete button, accidentally deleting a product is all too likely.

In future tutorials we'll see how to improve the editing user experience.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Zack Jones, Ken Pespisa, and Randy Schmidt. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Performing Batch Updates (VB)

2/20/2020 • 12 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> Learn how to create a fully-editable DataList where all of its items are in edit mode and whose values can be saved by clicking an "Update All" button on the page.

Introduction

In the preceding tutorial we examined how to create an item-level DataList. Like the standard editable GridView, each item in the DataList included an Edit button that, when clicked, would make the item editable. While this item-level editing works well for data that is only updated occasionally, certain use case scenarios require the user to edit many records. If a user needs to edit dozens of records and is forced to click Edit, make their changes, and click Update for each one, the amount of clicking can hamper her productivity. In such situations, a better option is to provide a fully-editable DataList, one where *all* of its items are in edit mode and whose values can be edited by clicking an Update All button on the page (see Figure 1).

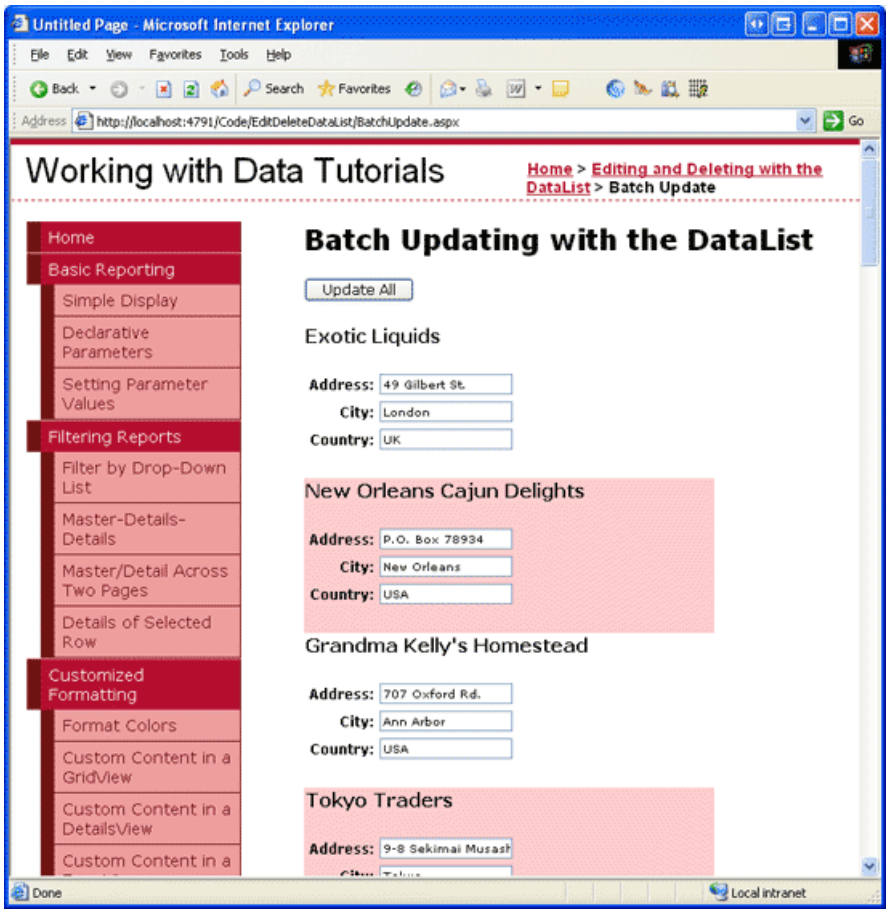


Figure 1: Each Item in a Fully Editable DataList can be Modified (Click to view full-size image)

In this tutorial we'll examine how to enable users to update suppliers address information using a fully editable DataList.

Step 1: Create the Editable User Interface in the DataList s

ItemTemplate

In the preceding tutorial, where we creating a standard, item-level editable DataList, we used two templates:

- `ItemTemplate` contained the read-only user interface (the Label Web controls for displaying each product s name and price).
- `EditItemTemplate` contained the edit mode user interface (the two TextBox Web controls).

The DataList s `EditItemIndex` property dictates what `DataListItem` (if any) is rendered using the `EditItemTemplate`. In particular, the `DataListItem` whose `ItemIndex` value matches the DataList s `EditItemIndex` property is rendered using the `EditItemTemplate`. This model works well when only one item can be edited at a time, but falls apart when creating a fully-editable DataList.

For a fully editable DataList, we want *all* of the `DataListItem` s to render using the editable interface. The simplest way to accomplish this is to define the editable interface in the `ItemTemplate`. For modifying the suppliers address information, the editable interface contains the supplier name as text and then TextBoxes for the address, city, and country values.

Start by opening the `BatchUpdate.aspx` page, add a DataList control, and set its `ID` property to `Suppliers`. From the DataList s smart tag, opt to add a new ObjectDataSource control named `SuppliersDataSource`.

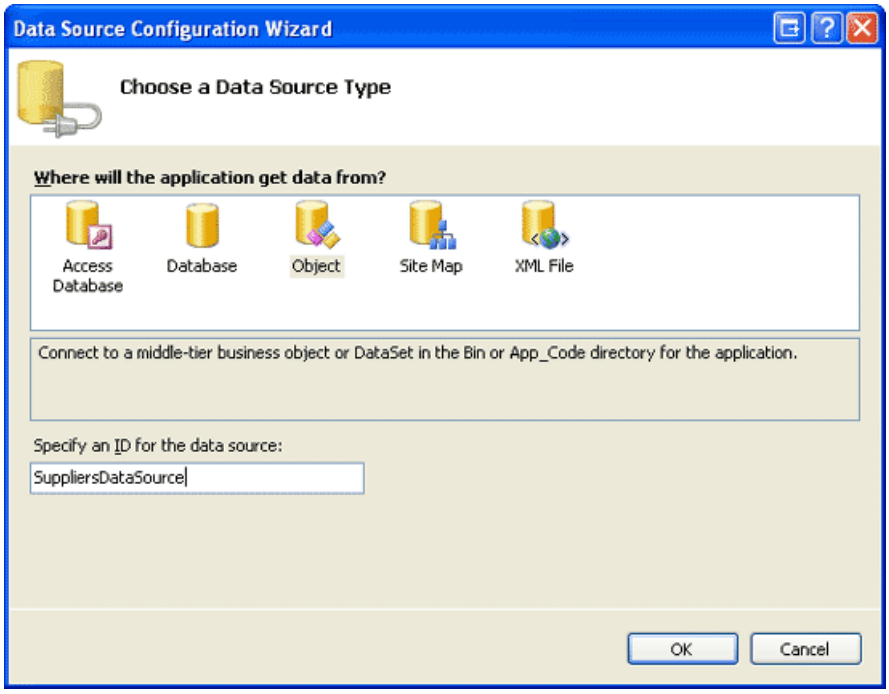


Figure 2: Create a New ObjectDataSource Named `SuppliersDataSource` (Click to view full-size image)

Configure the ObjectDataSource to retrieve data using the `SuppliersBLL` class s `GetSuppliers()` method (see Figure 3). As with the preceding tutorial, rather than updating the supplier information through the ObjectDataSource, we'll work directly with the Business Logic Layer. Therefore, set the drop-down list to (None) in the UPDATE tab (see Figure 4).

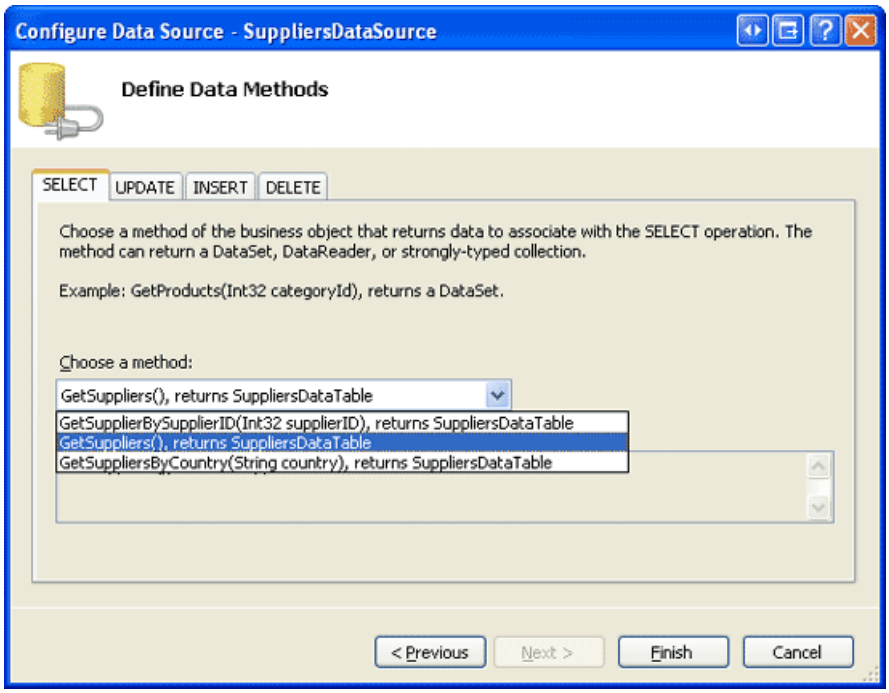


Figure 3: Retrieve Supplier Information Using the `GetSuppliers()` Method (Click to view full-size image)

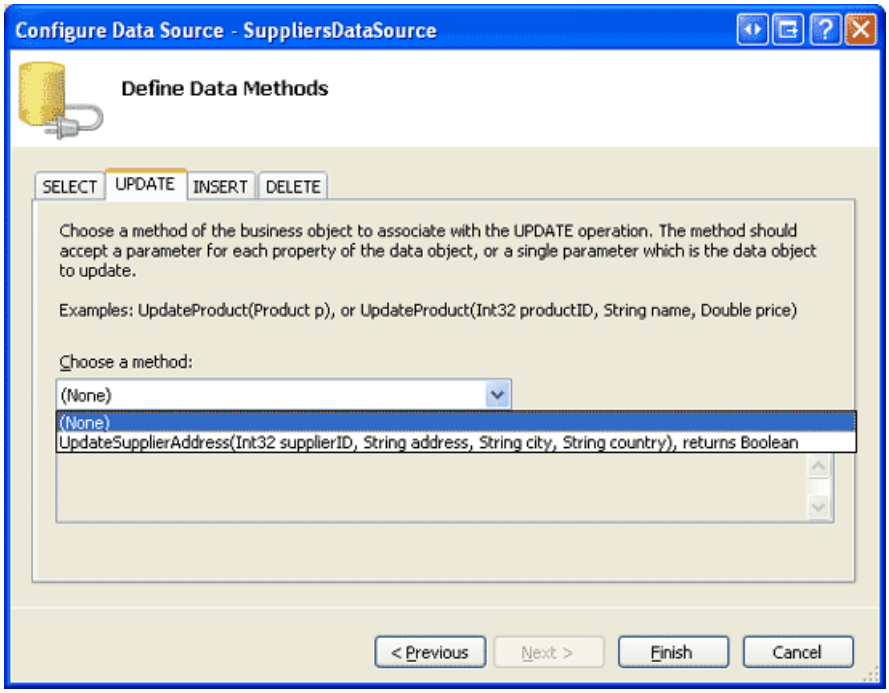


Figure 4: Set the Drop-Down List to (None) in the UPDATE Tab (Click to view full-size image)

After completing the wizard, Visual Studio automatically generates the DataList s `ItemTemplate` to display each data field returned by the data source in a Label Web control. We need to modify this template so that it provides the editing interface instead. The `ItemTemplate` can be customized through the Designer using the Edit Templates option from the DataList s smart tag or directly through the declarative syntax.

Take a moment to create an editing interface that displays the supplier s name as text, but includes TextBoxes for the supplier s address, city, and country values. After making these changes, your page s declarative syntax should look similar to the following:

```
<asp:DataList ID="Suppliers" runat="server" DataKeyField="SupplierID"
    DataSourceID="SuppliersDataSource">
    <ItemTemplate>
        <h4><asp:Label ID="CompanyNameLabel" runat="server"
            Text='<%# Eval("CompanyName") %>' /></h4>
        <table border="0">
            <tr>
                <td class="SupplierPropertyLabel">Address:</td>
                <td class="SupplierPropertyValue">
                    <asp:TextBox ID="Address" runat="server"
                        Text='<%# Eval("Address") %>' />
                </td>
            </tr>
            <tr>
                <td class="SupplierPropertyLabel">City:</td>
                <td class="SupplierPropertyValue">
                    <asp:TextBox ID="City" runat="server"
                        Text='<%# Eval("City") %>' />
                </td>
            </tr>
            <tr>
                <td class="SupplierPropertyLabel">Country:</td>
                <td class="SupplierPropertyValue">
                    <asp:TextBox ID="Country" runat="server"
                        Text='<%# Eval("Country") %>' />
                </td>
            </tr>
        </table>
        <br />
    </ItemTemplate>
</asp:DataList>
<asp:ObjectDataSource ID="SuppliersDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetSuppliers" TypeName="SuppliersBLL">
</asp:ObjectDataSource>
```

NOTE

As with the preceding tutorial, the DataList in this tutorial must have its view state enabled.

In the `ItemTemplate` I m using two new CSS classes, `SupplierPropertyLabel` and `SupplierPropertyValue`, which have been added to the `Styles.css` class and configured to use the same style settings as the `ProductPropertyLabel` and `ProductPropertyValue` CSS classes.

```
.ProductPropertyLabel, .SupplierPropertyLabel
{
    font-weight: bold;
    text-align: right;
}
.ProductPropertyValue, .SupplierPropertyValue
{
    padding-right: 35px;
}
```

After making these changes, visit this page through a browser. As Figure 5 shows, each DataList item displays the supplier name as text and uses TextBoxes to display the address, city, and country.

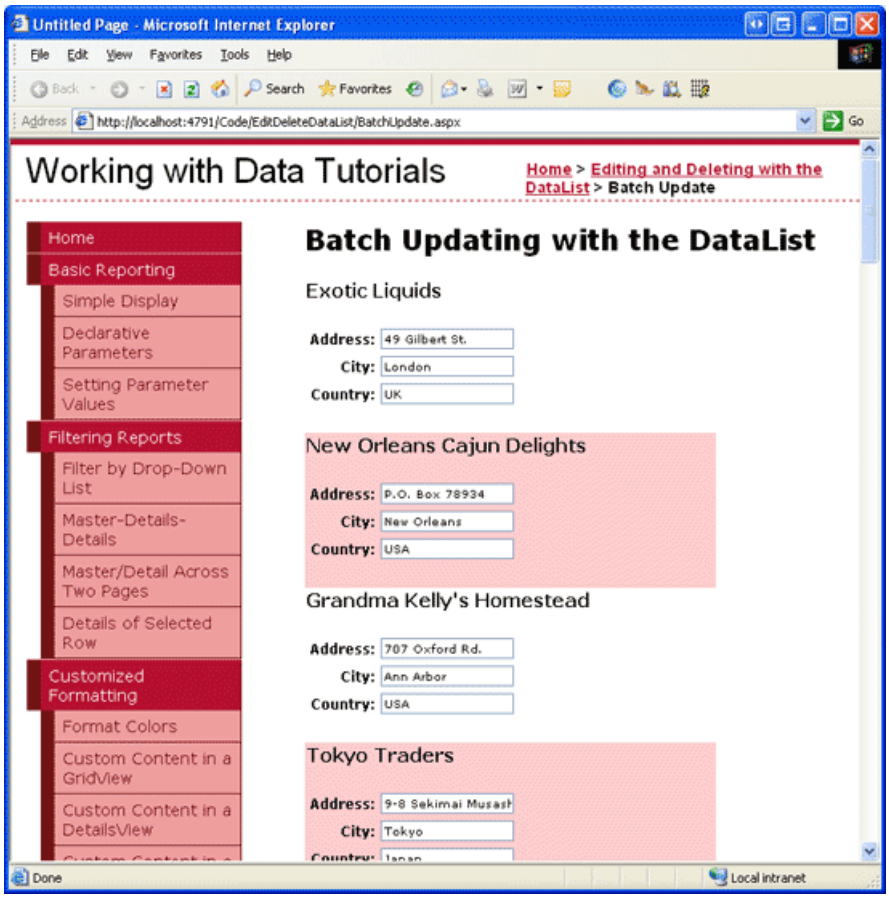


Figure 5: Each Supplier in the DataList is Editable (Click to view full-size image)

Step 2: Adding an Update All Button

While each supplier in Figure 5 has its address, city, and country fields displayed in a TextBox, there currently is no Update button available. Rather than having an Update button per item, with fully editable DataLists there is typically a single Update All button on the page that, when clicked, updates *all* of the records in the DataList. For this tutorial, let s add two Update All buttons - one at the top of the page and one at the bottom (although clicking either button will have the same effect).

Start by adding a Button Web control above the DataList and set its `ID` property to `UpdateAll1`. Next, add the second Button Web control beneath the DataList, setting its `ID` to `UpdateAll2`. Set the `Text` properties for the two Buttons to Update All. Lastly, create event handlers for both Buttons `Click` events. Rather than duplicating the update logic in each of the event handlers, let s refactor that logic to a third method, `UpdateAllSupplierAddresses`, having the event handlers simply invoking this third method.

```
Protected Sub UpdateAll1_Click(sender As Object, e As EventArgs) _
    Handles UpdateAll1.Click
    UpdateAllSupplierAddresses()
End Sub
Protected Sub UpdateAll2_Click(sender As Object, e As EventArgs) _
    Handles UpdateAll2.Click
    UpdateAllSupplierAddresses()
End Sub
Private Sub UpdateAllSupplierAddresses()
    ' TODO: Write code to update _all_ of the supplier addresses in the DataList
End Sub
```

Figure 6 shows the page after the Update All buttons have been added.

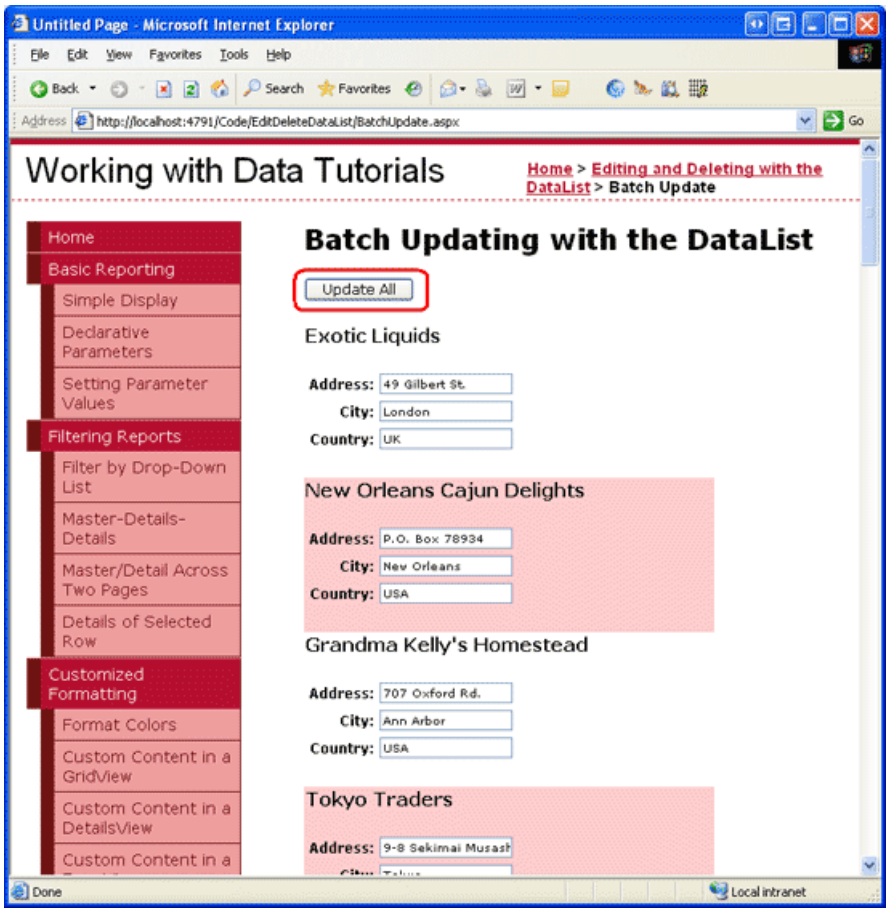


Figure 6: Two Update All Buttons have been Added to the Page (Click to view full-size image)

Step 3: Updating All of the Suppliers Address Information

With all of the DataList s items displaying the editing interface and with the addition of the Update All buttons, all that remains is writing the code to perform the batch update. Specifically, we need to loop through the DataList s items and call the `SuppliersBLL` class s `UpdateSupplierAddress` method for each one.

The collection of `DataListItem` instances that makeup the DataList can be accessed via the DataList s `Items` property. With a reference to a `DataListItem`, we can grab the corresponding `SupplierID` from the `DataKeys` collection and programmatically reference the TextBox Web controls within the `ItemTemplate` as the following code illustrates:

```
Private Sub UpdateAllSupplierAddresses()
    ' Create an instance of the SuppliersBLL class
    Dim suppliersAPI As New SuppliersBLL()
    ' Iterate through the DataList's items
    For Each item As DataListItem In Suppliers.Items
        ' Get the supplierID from the DataKeys collection
        Dim supplierID As Integer = Convert.ToInt32(Suppliers.DataKeys(item.ItemIndex))
        ' Read in the user-entered values
        Dim address As TextBox = CType(item.FindControl("Address"), TextBox)
        Dim city As TextBox = CType(item.FindControl("City"), TextBox)
        Dim country As TextBox = CType(item.FindControl("Country"), TextBox)
        Dim addressValue As String = Nothing, _
            cityValue As String = Nothing, _
            countryValue As String = Nothing
        If address.Text.Trim().Length > 0 Then
            addressValue = address.Text.Trim()
        End If
        If city.Text.Trim().Length > 0 Then
            cityValue = city.Text.Trim()
        End If
        If country.Text.Trim().Length > 0 Then
            countryValue = country.Text.Trim()
        End If
        ' Call the SuppliersBLL class's UpdateSupplierAddress method
        suppliersAPI.UpdateSupplierAddress _
            (supplierID, addressValue, cityValue, countryValue)
    Next
End Sub
```

When the user clicks one of the Update All buttons, the `UpdateAllSupplierAddresses` method iterates through each `DataListItem` in the `Suppliers` DataList and calls the `SuppliersBLL` class s `UpdateSupplierAddress` method, passing in the corresponding values. A non-entered value for address, city, or country passes is a value of `Nothing` to `UpdateSupplierAddress` (rather than a blank string), which results in a database `NULL` for the underlying record s fields.

> **NOTE**
> As an enhancement, you may want to add a status Label Web control to the page that provides some confirmation message after the batch update is performed.

Updating Only Those Addresses That Have Been Modified

The batch update algorithm used for this tutorial calls the `UpdateSupplierAddress` method for *every* supplier in the DataList, regardless of whether their address information has been changed. While such blind updates aren't usually a performance issue, they can lead to superfluous records if you re auditing changes to the database table. For example, if you use triggers to record all `UPDATE` s to the `Suppliers` table to an auditing table, every time a user clicks the Update All button a new audit record will be created for each supplier in the system, regardless of whether the user made any changes.

The ADO.NET DataTable and DataAdapter classes are designed to support batch updates where only modified, deleted, and new records results in any database communication. Each row in the DataTable has a `RowState` property that indicates whether the row has been added to the DataTable, deleted from it, modified, or remains unchanged. When a DataTable is initially populated, all rows are marked unchanged. Changing the value of any of the row s columns marks the row as modified.

In the `SuppliersBLL` class we update the specified supplier s address information by first reading in the single supplier record into a `SuppliersDataTable` and then set the `Address`, `City`, and `Country` column values using the following code:

```
Public Function UpdateSupplierAddress _
    (supplierID As Integer, address As String, city As String, country As String) _
    As Boolean
    Dim suppliers As Northwind.SuppliersDataTable = _
        Adapter.GetSupplierBySupplierID(supplierID)
    If suppliers.Count = 0 Then
        ' no matching record found, return false
        Return False
    Else
        Dim supplier As Northwind.SuppliersRow = suppliers(0)
        If address Is Nothing Then
            supplier.SetAddressNull()
        Else
            supplier.Address = address
        End If
        If city Is Nothing Then
            supplier.SetCityNull()
        Else
            supplier.City = city
        End If
        If country Is Nothing Then
            supplier.SetCountryNull()
        Else
            supplier.Country = country
        End If
        ' Update the supplier Address-related information
        Dim rowsAffected As Integer = Adapter.Update(supplier)
        ' Return true if precisely one row was updated, otherwise false
        Return rowsAffected = 1
    End If
End Function
```

This code naively assigns the passed-in address, city, and country values to the `SuppliersRow` in the `SuppliersDataTable` regardless of whether or not the values have changed. These modifications cause the `SuppliersRow` s `RowState` property to be marked as modified. When the Data Access Layer s `Update` method is called, it sees that the `SupplierRow` has been modified and therefore sends an `UPDATE` command to the database.

Imagine, however, that we added code to this method to only assign the passed-in address, city, and country values if they differ from the `SuppliersRow` s existing values. In the case where the address, city, and country are the same as the existing data, no changes will be made and the `SupplierRow` s `RowState` will be left marked as unchanged. The net result is that when the DAL s `Update` method is called, no database call will be made because the `SuppliersRow` has not been modified.

To enact this change, replace the statements that blindly assign the passed-in address, city, and country values with the following code:

```
' Only assign the values to the SupplierRow's column values if they differ
If address Is Nothing AndAlso Not supplier.IsAddressNull() Then
    supplier.SetAddressNull()
ElseIf (address IsNot Nothing AndAlso supplier.IsAddressNull) _
    OrElse (Not supplier.IsAddressNull() AndAlso _
                String.Compare(supplier.Address, address) <> 0) Then
    supplier.Address = address
End If
If city Is Nothing AndAlso Not supplier.IsCityNull() Then
    supplier.SetCityNull()
ElseIf (city IsNot Nothing AndAlso supplier.IsCityNull) _
    OrElse (Not supplier.IsCityNull() AndAlso _
                String.Compare(supplier.City, city) <> 0) Then
    supplier.City = city
End If
If country Is Nothing AndAlso Not supplier.IsCountryNull() Then
    supplier.SetCountryNull()
ElseIf (country IsNot Nothing AndAlso supplier.IsCountryNull) _
    OrElse (Not supplier.IsCountryNull() AndAlso _
                String.Compare(supplier.Country, country) <> 0) Then
    supplier.Country = country
End If
```

With this added code, the DAL s `Update` method sends an `UPDATE` statement to the database for only those records whose address-related values have changed.

Alternatively, we could keep track of whether there are any differences between the passed-in address fields and the database data and, if there are none, simply bypass the call to the DAL s `Update` method. This approach works well if you re using the DB direct method, since the DB direct method isn t passed a `SuppliersRow` instance whose `RowState` can be checked to determine whether a database call is actually needed.

> **NOTE**
> Each time the `UpdateSupplierAddress` method is invoked, a call is made to the database to retrieve information about the updated record. Then, if there are any changes in data, another call to the database is made to update the table row. This workflow could be optimized by creating an `UpdateSupplierAddress` method overload that accepts an `EmployeesDataTable` instance that has *all* of the changes from the `BatchUpdate.aspx` page. Then, it could make one call to the database to get all of the records from the `Suppliers` table. The two resultsets could then be enumerated and only those records where changes have occurred could be updated.

Summary

In this tutorial we saw how to create a fully editable DataList, allowing a user to quickly modify the address information for multiple suppliers. We started by defining the editing interface a TextBox Web control for the supplier s address, city, and country values in the DataList s `ItemTemplate`. Next, we added Update All buttons above and below the DataList. After a user has made his changes and clicked one of the Update All buttons, the `DataListItem` s are enumerated and a call to the `SuppliersBLL` class s `UpdateSupplierAddress` method is made.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Zack Jones and Ken Pespisa. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Handling BLL- and DAL-Level Exceptions (VB)

2/20/2020 • 10 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial, we'll see how to tactfully handle exceptions raised during an editable DataList's updating workflow.

Introduction

In the Overview of Editing and Deleting Data in the DataList tutorial, we created a DataList that offered simple editing and deleting capabilities. While fully functional, it was hardly user-friendly, as any error that occurred during the editing or deleting process resulted in an unhandled exception. For example, omitting the product s name or, when editing a product, entering a price value of Very affordable!, throws an exception. Since this exception is not caught in code, it bubbles up to the ASP.NET runtime, which then displays the exception s details in the web page.

As we saw in the Handling BLL- and DAL-Level Exceptions in an ASP.NET Page tutorial, if an exception is raised from the depths of the Business Logic or Data Access Layers, the exception details are returned to the ObjectDataSource and then to the GridView. We saw how to gracefully handle these exceptions by creating `Updated` or `RowUpdated` event handlers for the ObjectDataSource or GridView, checking for an exception, and then indicating that the exception was handled.

Our DataList tutorials, however, aren't using the ObjectDataSource for updating and deleting data. Instead, we are working directly against the BLL. In order to detect exceptions originating from the BLL or DAL, we need to implement exception handling code within the code-behind of our ASP.NET page. In this tutorial, we'll see how to more tactfully handle exceptions raised during an editable DataList s updating workflow.

> **NOTE**
>
> In the *An Overview of Editing and Deleting Data in the DataList* tutorial we discussed different techniques for editing and deleting data from the DataList, Some techniques involved using an ObjectDataSource for updating and deleting. If you employ these techniques, you can handle exceptions from the BLL or DAL through the ObjectDataSource s `Updated` or `Deleted` event handlers.

Step 1: Creating an Editable DataList

Before we worry about handling exceptions that occur during the updating workflow, let s first create an editable DataList. Open the `ErrorHandling.aspx` page in the `EditDeleteDataList` folder, add a DataList to the Designer, set its `ID` property to `Products`, and add a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to use the `ProductsBLL` class s `GetProducts()` method for selecting records; set the drop-down lists in the INSERT, UPDATE, and DELETE tabs to (None).

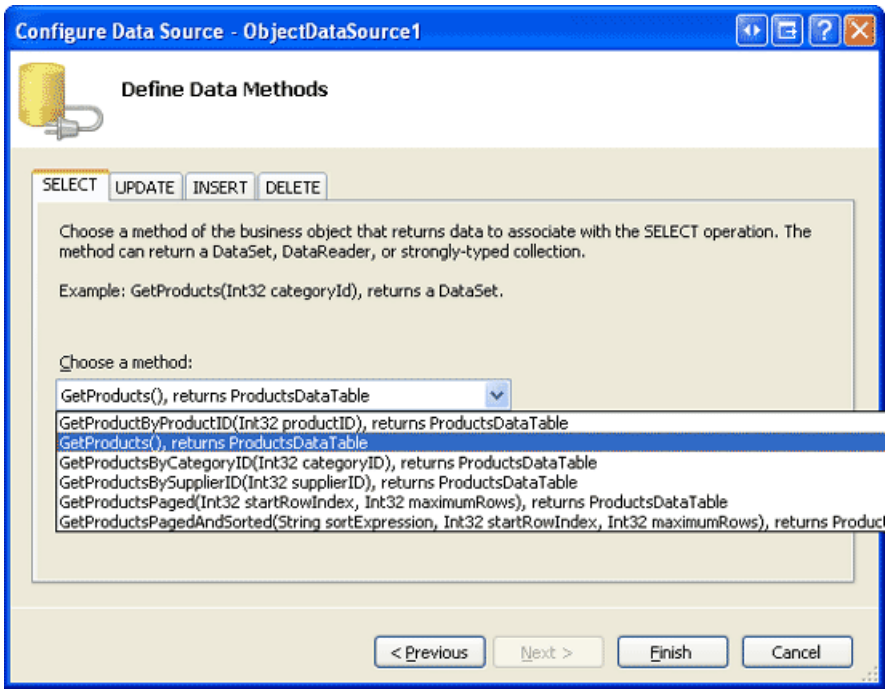


Figure 1: Return the Product Information Using the `GetProducts()` Method (Click to view full-size image)

After completing the ObjectDataSource wizard, Visual Studio will automatically create an `ItemTemplate` for the DataList. Replace this with an `ItemTemplate` that displays each product s name and price and includes an Edit button. Next, create an `EditItemTemplate` with a TextBox Web control for name and price and Update and Cancel buttons. Finally, set the DataList s `RepeatColumns` property to 2.

After these changes, your page s declarative markup should look similar to the following. Double-check to make certain that the Edit, Cancel, and Update buttons have their `CommandName` properties set to Edit, Cancel, and Update, respectively.

```
<asp:DataList ID="Products" runat="server" DataKeyField="ProductID"
    DataSourceID="ProductsDataSource" RepeatColumns="2">
    <ItemTemplate>
        <h5>
            <asp:Label runat="server" ID="ProductNameLabel"
                Text='<%# Eval("ProductName") %>' />
        </h5>
        Price:
            <asp:Label runat="server" ID="Label1"
                Text='<%# Eval("UnitPrice", "{0:C}") %>' />
        <br />
            <asp:Button runat="server" id="EditProduct" CommandName="Edit"
                Text="Edit" />
        <br />
        <br />
    </ItemTemplate>
    <EditItemTemplate>
        Product name:
            <asp:TextBox ID="ProductName" runat="server"
                Text='<%# Eval("ProductName") %>' />
        <br />
        Price:
            <asp:TextBox ID="UnitPrice" runat="server"
                Text='<%# Eval("UnitPrice", "{0:C}") %>' />
        <br />
        <br />
            <asp:Button ID="UpdateProduct" runat="server" CommandName="Update"
                Text="Update" />
            <asp:Button ID="CancelUpdate" runat="server" CommandName="Cancel"
                Text="Cancel" />
    </EditItemTemplate>
</asp:DataList>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    SelectMethod="GetProducts" TypeName="ProductsBLL"
    OldValuesParameterFormatString="original_{0}">
</asp:ObjectDataSource>
```

NOTE

For this tutorial the DataList s view state must be enabled.

Take a moment to view our progress through a browser (see Figure 2).

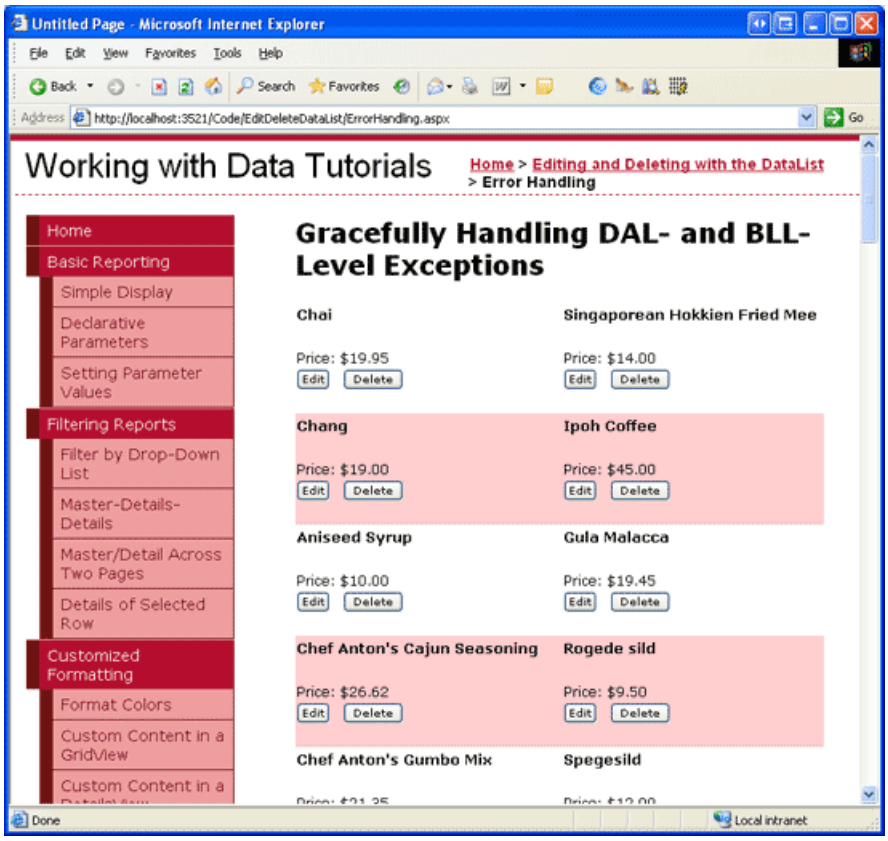


Figure 2: Each Product Includes an Edit Button (Click to view full-size image)

Currently, the Edit button only causes a postback it doesn t yet make the product editable. To enable editing, we need to create event handlers for the DataList s `EditCommand`, `CancelCommand`, and `UpdateCommand` events. The `EditCommand` and `CancelCommand` events simply update the DataList s `EditItemIndex` property and rebind the data to the DataList:

```
Protected Sub Products_EditCommand(source As Object, e As DataListCommandEventArgs) _
    Handles Products.EditCommand
    ' Set the DataList's EditItemIndex property to the
    ' index of the DataListItem that was clicked
    Products.EditItemIndex = e.Item.ItemIndex
    ' Rebind the data to the DataList
    Products.DataBind()
End Sub
Protected Sub Products_CancelCommand(source As Object, e As DataListCommandEventArgs) _
    Handles Products.CancelCommand
    ' Set the DataList's EditItemIndex property to -1
    Products.EditItemIndex = -1
    ' Rebind the data to the DataList
    Products.DataBind()
End Sub
```

The `UpdateCommand` event handler is a bit more involved. It needs to read in the edited product s `ProductID` from the `DataKeys` collection along with the product s name and price from the TextBoxes in the `EditItemTemplate`, and then call the `ProductsBLL` class s `UpdateProduct` method before returning the DataList to its pre-editing state.

For now, let s just use the exact same code from the `UpdateCommand` event handler in the *Overview of Editing and Deleting Data in the DataList* tutorial. We'll add the code to gracefully handle exceptions in step 2.

```
Protected Sub Products_UpdateCommand(source As Object, e As DataListCommandEventArgs) _
    Handles Products.UpdateCommand
    ' Read in the ProductID from the DataKeys collection
    Dim productID As Integer = Convert.ToInt32(Products.DataKeys(e.Item.ItemIndex))
    ' Read in the product name and price values
    Dim productName As TextBox = CType(e.Item.FindControl("ProductName"), TextBox)
    Dim unitPrice As TextBox = CType(e.Item.FindControl("UnitPrice"), TextBox)
    Dim productNameValue As String = Nothing
    If productName.Text.Trim().Length > 0 Then
        productNameValue = productName.Text.Trim()
    End If
    Dim unitPriceValue As Nullable(Of Decimal) = Nothing
    If unitPrice.Text.Trim().Length > 0 Then
        unitPriceValue = Decimal.Parse(unitPrice.Text.Trim(), _
                         System.Globalization.NumberStyles.Currency)
    End If
    ' Call the ProductsBLL's UpdateProduct method...
    Dim productsAPI As New ProductsBLL()
    productsAPI.UpdateProduct(productNameValue, unitPriceValue, productID)
    ' Revert the DataList back to its pre-editing state
    Products.EditItemIndex = -1
    Products.DataBind()
End Sub
```

In the face of invalid input which can be in the form of an improperly formatted unit price, an illegal unit price value like -$5.00, or the omission of the product s name an exception will be raised. Since the `UpdateCommand` event handler does not include any exception handling code at this point, the exception will bubble up to the ASP.NET runtime, where it will be displayed to the end user (see Figure 3).

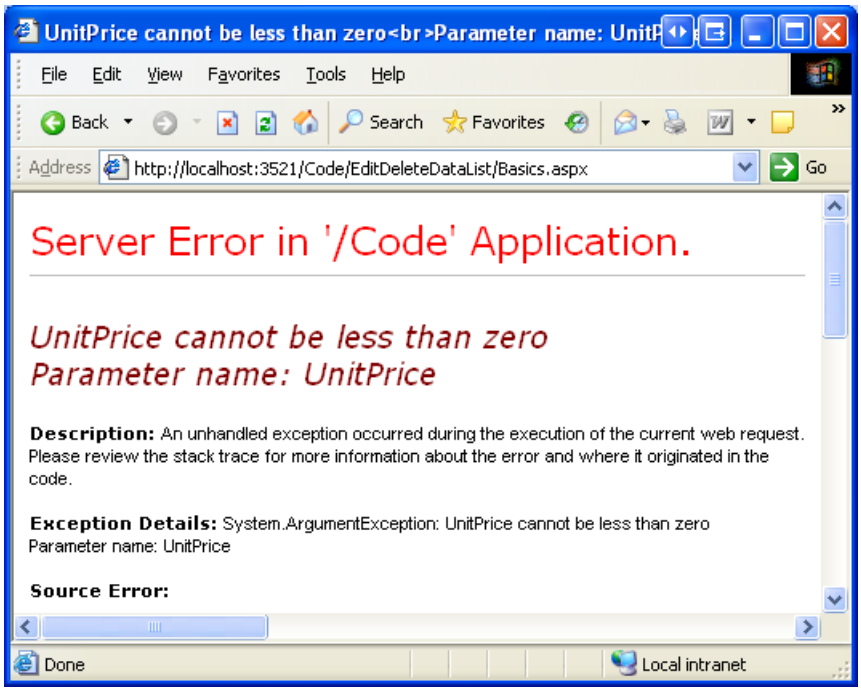


Figure 3: When an Unhandled Exception Occurs, the End User Sees an Error Page

Step 2: Gracefully Handling Exceptions in the UpdateCommand Event Handler

During the updating workflow, exceptions can occur in the `UpdateCommand` event handler, the BLL, or the DAL. For example, if a user enters a price of Too expensive, the `Decimal.Parse` statement in the `UpdateCommand` event handler will throw a `FormatException` exception. If the user omits the product s name or if the price has a negative value, the DAL will raise an exception.

When an exception occurs, we want to display an informative message within the page itself. Add a Label Web control to the page whose `ID` is set to `ExceptionDetails`. Configure the Label s text to display in a red, extra-large, bold and italic font by assigning its `CssClass` property to the `Warning` CSS class, which is defined in the `Styles.css` file.

When an error occurs, we only want the Label to be displayed once. That is, on subsequent postbacks, the Label s warning message should disappear. This can be accomplished by either clearing out the Label s `Text` property or settings its `Visible` property to `False` in the `Page_Load` event handler (as we did back in the Handling BLL- and DAL-Level Exceptions in an ASP.NET Page tutorial) or by disabling the Label s view state support. Let s use the latter option.

```
<asp:Label ID="ExceptionDetails" EnableViewState="False" CssClass="Warning"
    runat="server" />
```

When an exception is raised, we'll assign the details of the exception to the `ExceptionDetails` Label control s `Text` property. Since its view state is disabled, on subsequent postbacks the `Text` property s programmatic changes will be lost, reverting back to the default text (an empty string), thereby hiding the warning message.

To determine when an error has been raised in order to display a helpful message on the page, we need to add a `Try ... Catch` block to the `UpdateCommand` event handler. The `Try` portion contains code that may lead to an exception, while the `Catch` block contains code that is executed in the face of an exception. Check out the Exception Handling Fundamentals section in the .NET Framework documentation for more information on the `Try ... Catch` block.

```
Protected Sub Products_UpdateCommand(source As Object, e As DataListCommandEventArgs) _
    Handles Products.UpdateCommand
    ' Handle any exceptions raised during the editing process
    Try
        ' Read in the ProductID from the DataKeys collection
        Dim productID As Integer = _
            Convert.ToInt32(Products.DataKeys(e.Item.ItemIndex))
        ... Some code omitted for brevity ...
    Catch ex As Exception
        ' TODO: Display information about the exception in ExceptionDetails
    End Try
End Sub
```

When an exception of any type is thrown by code within the `Try` block, the `Catch` block s code will begin executing. The type of exception that is thrown `DbException`, `NoNullAllowedException`, `ArgumentException`, and so on depends on what, exactly, precipitated the error in the first place. If there s a problem at the database level, a `DbException` will be thrown. If an illegal value is entered for the `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, or `ReorderLevel` fields, an `ArgumentException` will be thrown, as we added code to validate these field values in the `ProductsDataTable` class (see the Creating a Business Logic Layer tutorial).

We can provide a more helpful explanation to the end user by basing the message text on the type of exception caught. The following code which was used in a nearly identical form back in the Handling BLL- and DAL-Level Exceptions in an ASP.NET Page tutorial provides this level of detail:

```
Private Sub DisplayExceptionDetails(ByVal ex As Exception)
    ' Display a user-friendly message
    ExceptionDetails.Text = "There was a problem updating the product. "
    If TypeOf ex Is System.Data.Common.DbException Then
        ExceptionDetails.Text += "Our database is currently experiencing problems." + _
                                 "Please try again later."
    ElseIf TypeOf ex Is System.Data.NoNullAllowedException Then
        ExceptionDetails.Text+="There are one or more required fields that are missing."
    ElseIf TypeOf ex Is ArgumentException Then
        Dim paramName As String = CType(ex, ArgumentException).ParamName
        ExceptionDetails.Text+=String.Concat("The ", paramName, " value is illegal.")
    ElseIf TypeOf ex Is ApplicationException Then
        ExceptionDetails.Text += ex.Message
    End If
End Sub
```

To complete this tutorial, simply call the `DisplayExceptionDetails` method from the `Catch` block passing in the caught `Exception` instance (`ex`).

With the `Try ... Catch` block in place, users are presented with a more informative error message, as Figures 4 and 5 show. Note that in the face of an exception the DataList remains in edit mode. This is because once the exception occurs, the control flow is immediately redirected to the `Catch` block, bypassing the code that returns the DataList to its pre-editing state.

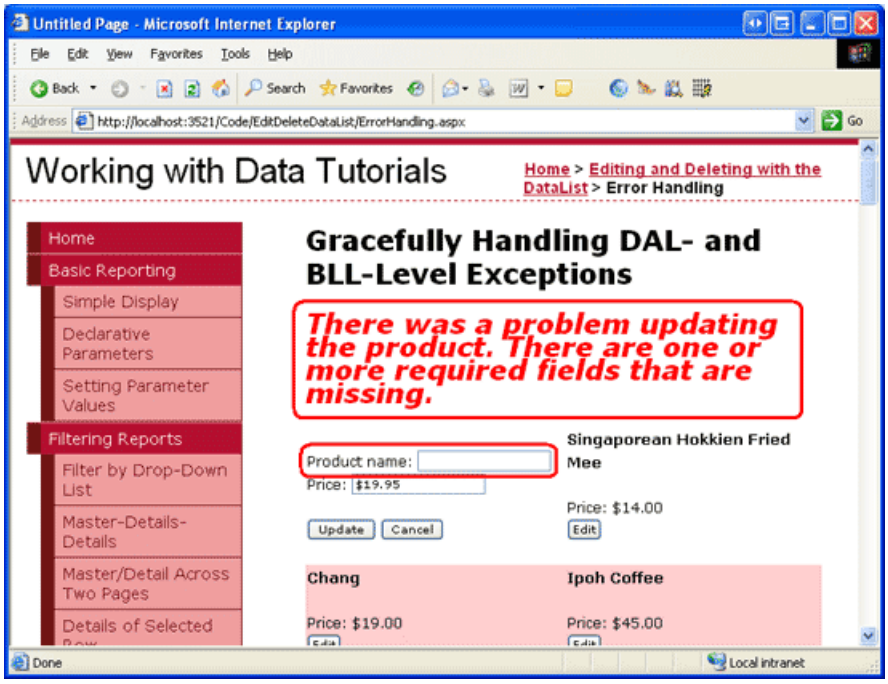


Figure 4: An Error Message is Displayed if a User Omits a Required Field (Click to view full-size image)

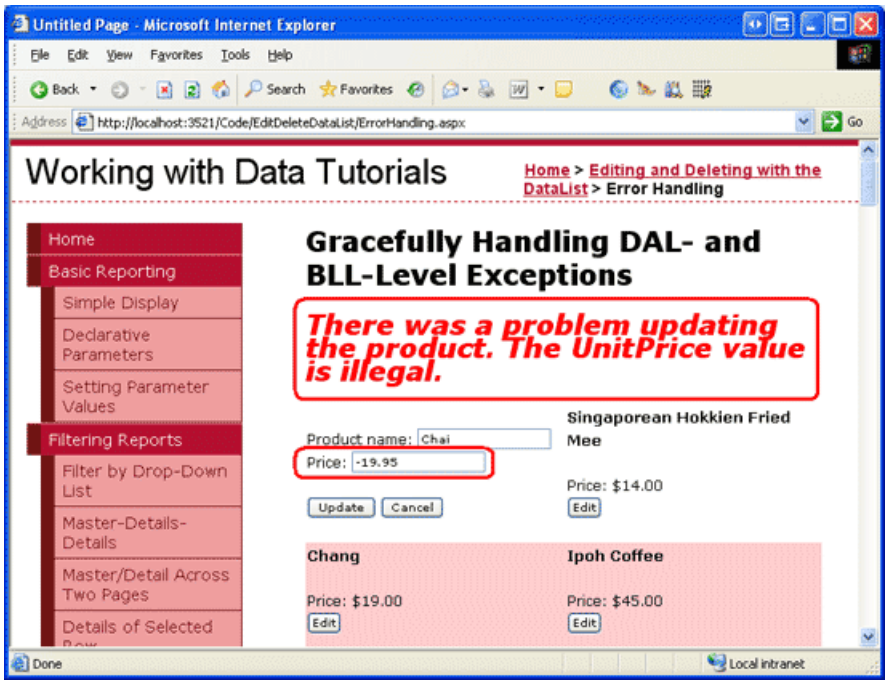


Figure 5: An Error Message is Displayed When Entering a Negative Price (Click to view full-size image)

Summary

The GridView and ObjectDataSource provide post-level event handlers that include information about any exceptions that were raised during the updating and deleting workflow, as well as properties that can be set to indicate whether or not the exception has been handled. These features, however, are unavailable when working with the DataList and using the BLL directly. Instead, we are responsible for implementing exception handling.

In this tutorial we saw how to add exception handling to an editable DataList s updating workflow by adding a `Try ... Catch` block to the `UpdateCommand` event handler. If an exception is raised during the updating workflow, the `Catch` block s code executes, displaying helpful information in the `ExceptionDetails` Label.

At this point, the DataList makes no effort to prevent exceptions from happening in the first place. Even though we know that a negative price will result in an exception, we haven't yet added any functionality to proactively prevent a user from entering such invalid input. In our next tutorial we'll see how to help reduce the exceptions caused by invalid user input by adding validation controls in the `EditItemTemplate`.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Design Guidelines for Exceptions
- Error Logging Modules and Handlers (ELMAH) (an open-source library for logging errors)
- Enterprise Library for .NET Framework 2.0 (includes the Exception Management Application Block)

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Ken Pespisa. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Adding Validation Controls to the DataList's Editing Interface (VB)

2/20/2020 • 14 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll see how easy it is to add validation controls to the DataList's EditItemTemplate in order to provide a more foolproof editing user interface.

Introduction

In the DataList editing tutorials thus far, the DataLists editing interfaces have not included any proactive user input validation even though invalid user input such as a missing product name or negative price results in an exception. In the preceding tutorial we examined how to add exception handling code to the DataList s `UpdateCommand` event handler in order to catch and gracefully display information about any exceptions that were raised. Ideally, however, the editing interface would include validation controls to prevent a user from entering such invalid data in the first place.

In this tutorial we'll see how easy it is to add validation controls to the DataList s `EditItemTemplate` in order to provide a more foolproof editing user interface. Specifically, this tutorial takes the example created in the preceding tutorial and augments the editing interface to include appropriate validation.

Step 1: Replicating the Example fromHandling BLL- and DAL-Level Exceptions

In the Handling BLL- and DAL-Level Exceptions tutorial we created a page that listed the names and prices of the products in a two-column, editable DataList. Our goal for this tutorial is to augment the DataList s editing interface to include validation controls. In particular, our validation logic will:

- Require that the product s name be provided
- Ensure that the value entered for the price is a valid currency format
- Ensure that the value entered for the price is greater than or equal to zero, since a negative `UnitPrice` value is illegal

Before we can look at augmenting the previous example to include validation, we first need to replicate the example from the `ErrorHandling.aspx` page in the `EditDeleteDataList` folder to the page for this tutorial, `UIValidation.aspx`. To achieve this we need to copy over both the `ErrorHandling.aspx` page s declarative markup and its source code. First copy over the declarative markup by performing the following steps:

1. Open the `ErrorHandling.aspx` page in Visual Studio
2. Go to the page s declarative markup (click on the Source button at the bottom of the page)
3. Copy the text within the `<asp:Content>` and `</asp:Content>` tags (lines 3 through 32), as shown in Figure 1.

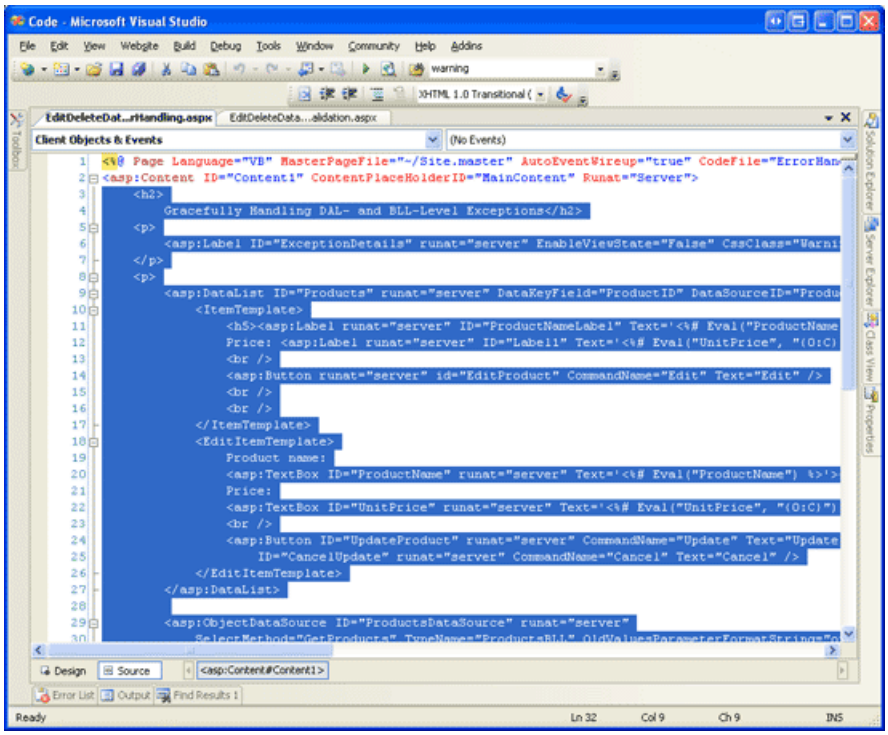


Figure 2: Copy the Text Within the `<asp:Content>` Control (Click to view full-size image)

1. Open the `UIValidation.aspx` page
2. Go to the page s declarative markup
3. Paste the text within the `<asp:Content>` control.

To copy over the source code, open the `ErrorHandling.aspx.vb` page and copy just the text *within* the `EditDeleteDataList_ErrorHandling` class. Copy the three event handlers (`Products_EditCommand`, `Products_CancelCommand`, and `Products_UpdateCommand`) along with the `DisplayExceptionDetails` method, but do **not** copy the class declaration or `using` statements. Paste the copied text *within* the `EditDeleteDataList_UIValidation` class in `UIValidation.aspx.vb`.

After moving over the content and code from `ErrorHandling.aspx` to `UIValidation.aspx`, take a moment to test out the pages in a browser. You should see the same output and experience the same functionality in each of these two pages (see Figure 2).

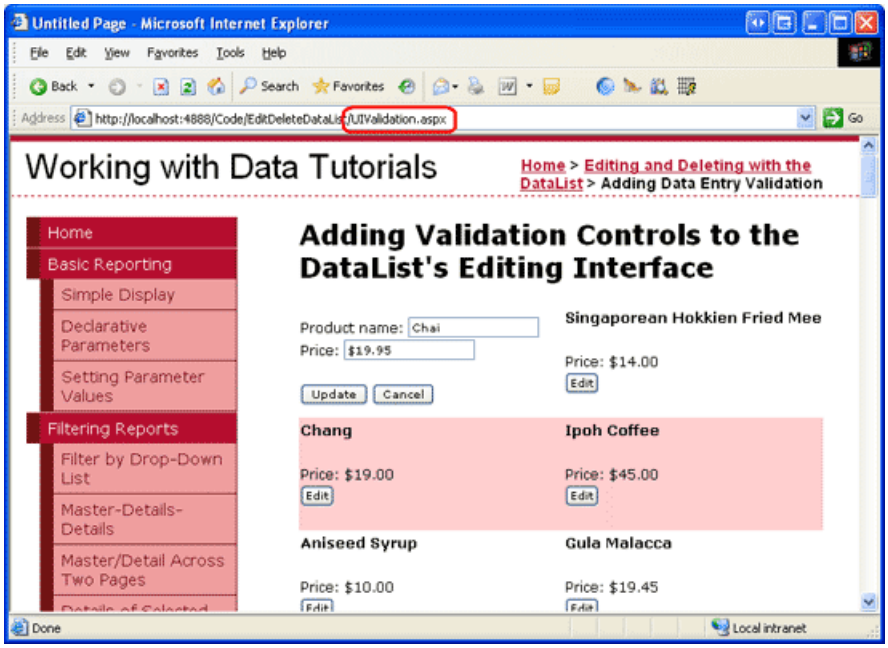


Figure 2: The `UIValidation.aspx` Page Mimics the Functionality in `ErrorHandling.aspx` (Click to view full-size

image)

Step 2: Adding the Validation Controls to the DataList s EditItemTemplate

When constructing data entry forms, it is important that users enter any required fields and that all their provided inputs are legal, properly-formatted values. To help ensure that a user s inputs are valid, ASP.NET provides five built-in validation controls that are designed to validate the value of a single input Web control:

- RequiredFieldValidator ensures that a value has been provided
- CompareValidator validates a value against another Web control value or a constant value, or ensures that the value s format is legal for a specified data type
- RangeValidator ensures that a value is within a range of values
- RegularExpressionValidator validates a value against a regular expression
- CustomValidator validates a value against a custom, user-defined method

For more information on these five controls refer back to the Adding Validation Controls to the Editing and Inserting Interfaces tutorial or check out the Validation Controls section of the ASP.NET Quickstart Tutorials.

For our tutorial we'll need to use a RequiredFieldValidator to ensure that a value for the product name has been supplied and a CompareValidator to ensure that the entered price has a value greater than or equal to 0 and is presented in a valid currency format.

> **NOTE**
> While ASP.NET 1.x had these same five validation controls, ASP.NET 2.0 has added a number of improvements, the main two being client-side script support for browsers in addition to Internet Explorer and the ability to partition validation controls on a page into validation groups. For more information on the new validation control features in 2.0, refer to Dissecting the Validation Controls in ASP.NET 2.0.

Let s start by adding the necessary validation controls to the DataList s `EditItemTemplate`. This task can be performed through the Designer by clicking on the Edit Templates link from the DataList s smart tag, or through the declarative syntax. Let s step through the process using the Edit Templates option from the Design view. After choosing to edit the DataList s `EditItemTemplate`, add a RequiredFieldValidator by dragging it from the Toolbox into the template editing interface, placing it after the `ProductName` TextBox.

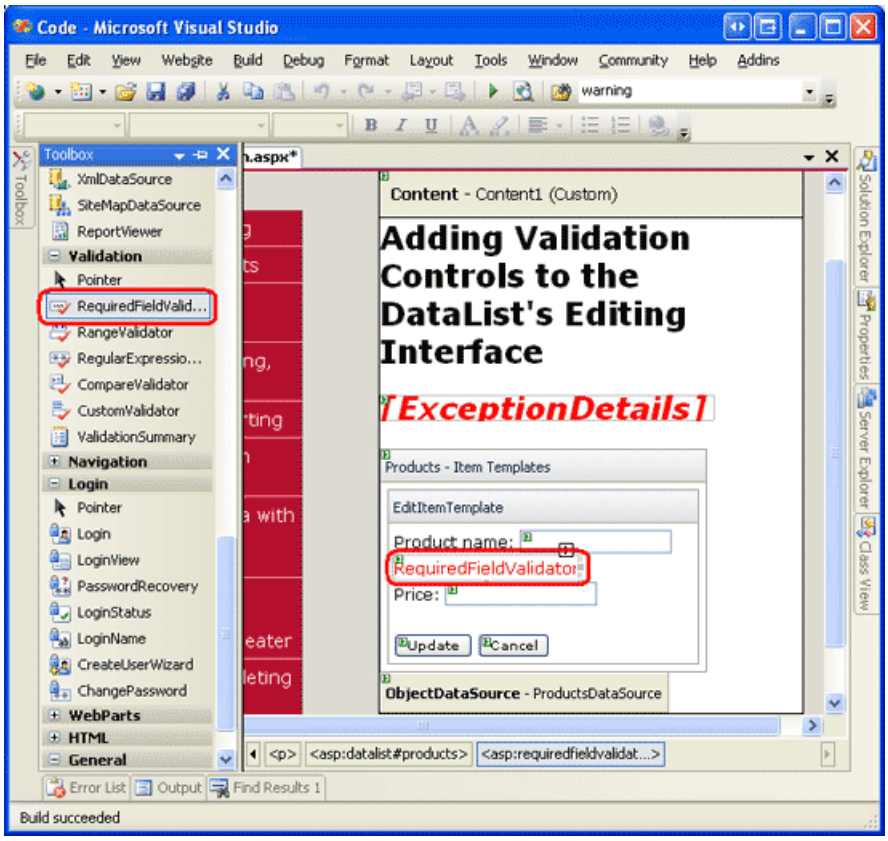


Figure 3: Add a RequiredFieldValidator to the `EditItemTemplate After` the `ProductName` TextBox (Click to view full-size image)

All validation controls work by validating the input of a single ASP.NET Web control. Therefore, we need to indicate that the RequiredFieldValidator we just added should validate against the `ProductName` TextBox; this is done by setting the validation control s `ControlToValidate` property to the `ID` of the appropriate Web control (`ProductName` , in this instance). Next, set the `ErrorMessage` property to You must provide the product s name and the `Text` property to *. The `Text` property value, if provided, is the text that is displayed by the validation control if the validation fails. The `ErrorMessage` property value, which is required, is used by the ValidationSummary control; if the `Text` property value is omitted, the `ErrorMessage` property value is displayed by the validation control on invalid input.

After setting these three properties of the RequiredFieldValidator, your screen should look similar to Figure 4.

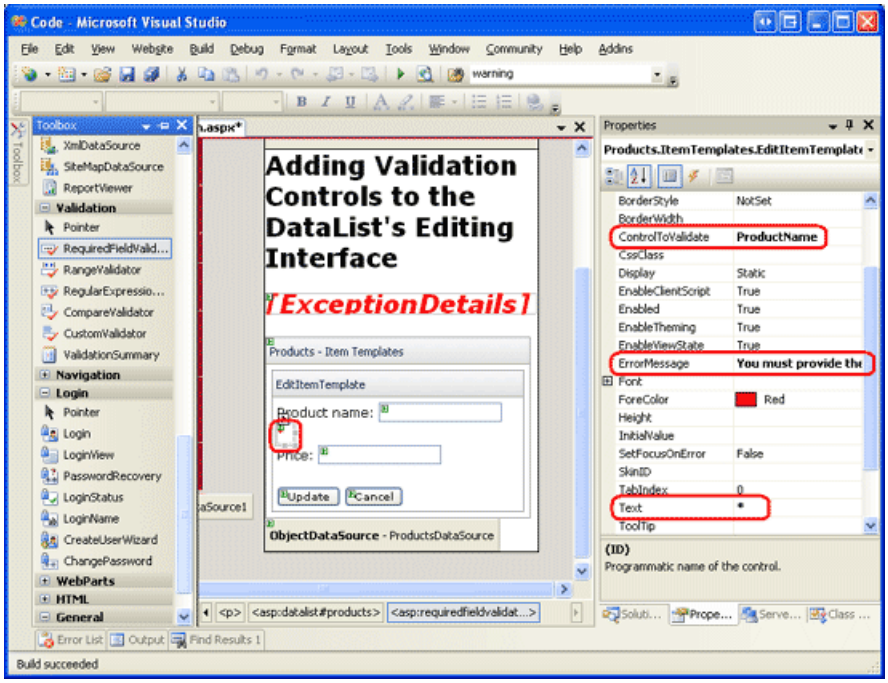


Figure 4: Set the RequiredFieldValidator s `ControlToValidate` , `ErrorMessage` , and `Text` Properties (Click to view full-size image)

With the RequiredFieldValidator added to the `EditItemTemplate` , all that remains is to add the necessary validation for the product s price TextBox. Since the `UnitPrice` is optional when editing a record, we don t need to add a RequiredFieldValidator. We do, however, need to add a CompareValidator to ensure that the `UnitPrice` , if supplied, is properly formatted as a currency and is greater than or equal to 0.

Add the CompareValidator into the `EditItemTemplate` and set its `ControlToValidate` property to `UnitPrice` , its `ErrorMessage` property to The price must be greater than or equal to zero and cannot include the currency symbol, and its `Text` property to *. To indicate that the `UnitPrice` value must be greater than or equal to 0, set the CompareValidator s `Operator` property to `GreaterThanEqual` , its `ValueToCompare` property to 0, and its `Type` property to `Currency` .

After adding these two validation controls, the DataList s `EditItemTemplate` s declarative syntax should look similar to the following:

```
<EditItemTemplate>
    Product name:
        <asp:TextBox ID="ProductName" runat="server"
            Text='<%# Eval("ProductName") %>'></asp:TextBox>
        <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
            ControlToValidate="ProductName"
            ErrorMessage="You must provide the product's name"
            runat="server">*</asp:RequiredFieldValidator>
    <br />
    Price:
        <asp:TextBox ID="UnitPrice" runat="server"
            Text='<%# Eval("UnitPrice", "{0:C}") %>'></asp:TextBox>
        <asp:CompareValidator ID="CompareValidator1"
            ControlToValidate="UnitPrice"
            ErrorMessage="The price must be greater than or equal to zero
                          and cannot include the currency symbol"
            Operator="GreaterThanEqual" Type="Currency" ValueToCompare="0"
            runat="server">*</asp:CompareValidator><br />
    <br />
    <asp:Button ID="UpdateProduct" runat="server" CommandName="Update"
        Text="Update" />
    <asp:Button ID="CancelUpdate" runat="server" CommandName="Cancel"
        Text="Cancel" />
</EditItemTemplate>
```

After making these changes, open the page in a browser. If you attempt to omit the name or enter an invalid price value when editing a product, an asterisk appears next to the textbox. As Figure 5 shows, a price value that includes the currency symbol such as $19.95 is considered invalid. The CompareValidator s `Currency` `Type` allows for digit separators (such as commas or periods, depending on the culture settings) and a leading plus or minus sign, but does *not* permit a currency symbol. This behavior may perplex users as the editing interface currently renders the `UnitPrice` using the currency format.

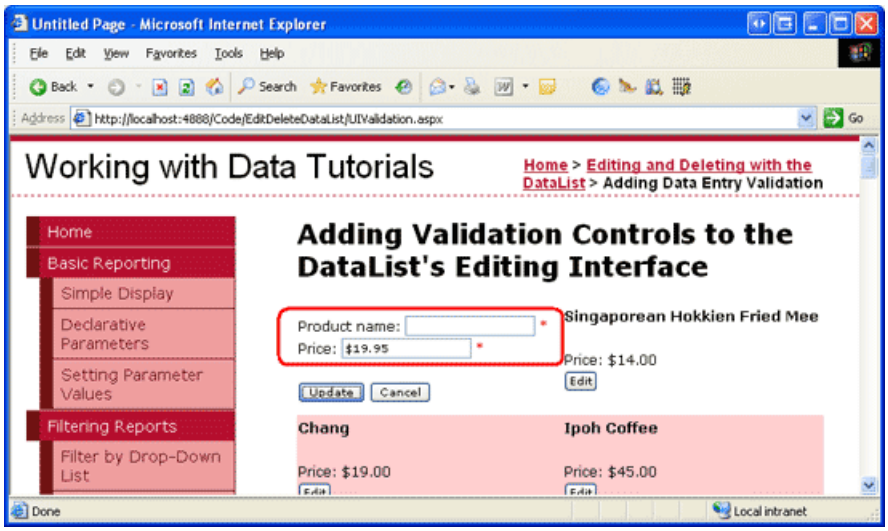


Figure 5: An Asterisk Appears Next to the Textboxes with Invalid Input (Click to view full-size image)

While the validation works as-is, the user has to manually remove the currency symbol when editing a record, which is not acceptable. Moreover, if there are invalid inputs in the editing interface neither the Update nor Cancel buttons, when clicked, will invoke a postback. Ideally, the Cancel button would return the DataList to its pre-editing state regardless of the validity of the user s inputs. Also, we need to ensure that the page s data is valid before updating the product information in the DataList s `UpdateCommand` event handler, as the validation controls client-side logic can be bypassed by users whose browsers either don t support JavaScript or have its support disabled.

Removing the Currency Symbol from the EditItemTemplate s UnitPrice TextBox

When using the CompareValidator s `Currency``Type`, the input being validated must not include any currency symbols. The presence of such symbols causes the CompareValidator to mark the input as invalid. However, our editing interface currently includes a currency symbol in the `UnitPrice` TextBox, meaning the user must explicitly remove the currency symbol before saving their changes. To remedy this we have three options:

1. Configure the `EditItemTemplate` so that the `UnitPrice` TextBox value is not formatted as a currency.
2. Allow the user to enter a currency symbol by removing the CompareValidator and replacing it with a RegularExpressionValidator that checks for a properly formatted currency value. The challenge here is that the regular expression to validate a currency value is not as straightforward as the CompareValidator and would require writing code if we wanted to incorporate culture settings.
3. Remove the validation control altogether and rely on custom server-side validation logic in the GridView s `RowUpdating` event handler.

Let s go with option 1 for this tutorial. Currently the `UnitPrice` is formatted as a currency value due to the databinding expression for the TextBox in the `EditItemTemplate`: `<%# Eval("UnitPrice", "{0:c}") %>`. Change the `Eval` statement to `Eval("UnitPrice", "{0:n2}")`, which formats the result as a number with two digits of precision. This can be done directly through the declarative syntax or by clicking on the Edit DataBindings link from the `UnitPrice` TextBox in the DataList s `EditItemTemplate`.

With this change, the formatted price in the editing interface includes commas as the group separator and a period as the decimal separator, but leaves off the currency symbol.

> **NOTE**
>
> When removing the currency format from the editable interface, I find it helpful to put the currency symbol as text outside the TextBox. This serves as a hint to the user that they do not need to provide the currency symbol.

Fixing the Cancel Button

By default, the validation Web controls emit JavaScript to perform validation on the client-side. When a Button, LinkButton, or ImageButton is clicked, the validation controls on the page are checked on the client-side before the postback occurs. If there is any invalid data, the postback is cancelled. For certain Buttons, though, the validity of the data might be immaterial; in such a case, having the postback cancelled due to invalid data is a nuisance.

The Cancel button is such an example. Imagine that a user enters invalid data, such as omitting the product s name, and then decides she doesn t want to save the product after all and hits the Cancel button. Currently, the Cancel button triggers the validation controls on the page, which report that the product name is missing and prevent the postback. Our user has to type some text into the `ProductName` TextBox just to cancel out of the editing process.

Fortunately, the Button, LinkButton, and ImageButton have a `CausesValidation` property that can indicate whether or not clicking the Button should initiate the validation logic (defaults to `True`). Set the Cancel Button s `CausesValidation` property to `False`.

Ensuring the Inputs are Valid in the UpdateCommand Event Handler

Due to the client-side script emitted by the validation controls, if a user enters invalid input the validation controls cancel any postbacks initiated by Button, LinkButton, or ImageButton controls whose `CausesValidation` properties are `True` (the default). However, if a user is visiting with an antiquated browser or one whose JavaScript support has been disabled, the client-side validation checks will not execute.

All of the ASP.NET validation controls repeat their validation logic immediately upon postback and report the overall validity of the page s inputs via the `Page.IsValid` property. However, the page flow is not interrupted or stopped in any way based on the value of `Page.IsValid`. As developers, it is our responsibility to ensure that the `Page.IsValid` property has a value of `True` before proceeding with code that assumes valid input data.

If a user has JavaScript disabled, visits our page, edits a product, enters a price value of Too expensive, and clicks the Update button, the client-side validation will be bypassed and a postback will ensue. On postback, the ASP.NET page s `UpdateCommand` event handler executes and an exception is raised when attempting to parse Too expensive to a `Decimal`. Since we have exception handling, such an exception will be handled gracefully, but we could prevent the invalid data from slipping through in the first place by only proceeding with the `UpdateCommand` event handler if `Page.IsValid` has a value of `True`.

Add the following code to the start of the `UpdateCommand` event handler, immediately before the `Try` block:

```
If Not Page.IsValid Then
    Exit Sub
End If
```

With this addition, the product will attempt to be updated only if the submitted data is valid. Most users won't be able to postback invalid data due to the validation controls client-side scripts, but users whose browsers don t support JavaScript or that have JavaScript support disabled, can bypass the client-side checks and submit invalid data.

> **NOTE**
>
> The astute reader will recall that when updating data with the GridView, we didn't need to explicitly check the `Page.IsValid` property in our page s code-behind class. This is because the GridView consults the `Page.IsValid` property for us and only proceeds with the update only if it returns a value of `True`.

Step 3: Summarizing Data Entry Problems

In addition to the five validation controls, ASP.NET includes the ValidationSummary control, which displays the `ErrorMessage` s of those validation controls that detected invalid data. This summary data can be displayed as text on the web page or through a modal, client-side messagebox. Let s enhance this tutorial to include a client-side messagebox summarizing any validation problems.

To accomplish this, drag a ValidationSummary control from the Toolbox onto the Designer. The location of the ValidationSummary control doesn t really matter, since we re going to configure it to only display the summary as a messagebox. After adding the control, set its `ShowSummary` property to `False` and its `ShowMessageBox` property to `True`. With this addition, any validation errors are summarized in a client-side messagebox (see Figure 6).

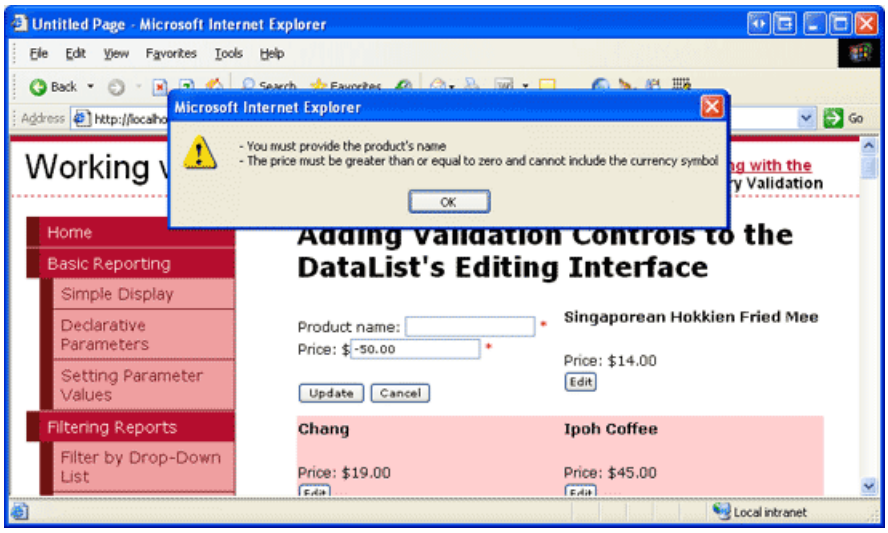


Figure 6: The Validation Errors are Summarized in a Client-Side Messagebox (Click to view full-size image)

Summary

In this tutorial we saw how to reduce the likelihood of exceptions by using validation controls to proactively ensure that our users inputs are valid before attempting to use them in the updating workflow. ASP.NET provides five validation Web controls that are designed to inspect a particular Web control s input and report back on the input s validity. In this tutorial we used two of those five controls the RequiredFieldValidator and the CompareValidator to ensure that the product s name was supplied and that the price had a currency format with a value greater than or equal to zero.

Adding validation controls to the DataList s editing interface is as simple as dragging them onto the `EditItemTemplate` from the Toolbox and setting a handful of properties. By default, the validation controls automatically emit client-side validation script; they also provide server-side validation on postback, storing the cumulative result in the `Page.IsValid` property. To bypass the client-side validation when a Button, LinkButton, or ImageButton is clicked, set the button s `CausesValidation` property to `False`. Also, before performing any tasks with the data submitted on postback, ensure that the `Page.IsValid` property returns `True`.

All of the DataList editing tutorials we ve examined so far have had very simple editing interfaces a TextBox for the product s name and another for the price. The editing interface, however, can contain a mix of different Web controls, such as DropDownLists, Calendars, RadioButtons, CheckBoxes, and so on. In our next tutorial we'll look at building an interface that uses a variety of Web controls.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest

book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Dennis Patterson, Ken Pespisa, and Liz Shulok. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Customizing the DataList's Editing Interface (VB)

2/20/2020 • 13 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we will create a richer editing interface for the DataList, one that includes DropDownLists and a CheckBox.

Introduction

The markup and Web controls in the DataList s `EditItemTemplate` define its editable interface. In all of the editable DataList examples we ve examined thus far, the editable interface has been composed of TextBox Web controls. In the preceding tutorial we improved the edit-time user experience by adding validation controls.

The `EditItemTemplate` can be further expanded to include Web controls other than the TextBox, such as DropDownLists, RadioButtonLists, Calendars, and so on. As with TextBoxes, when customizing the editing interface to include other Web controls, employ the following steps:

1. Add the Web control to the `EditItemTemplate`.
2. Use databinding syntax to assign the corresponding data field value to the appropriate property.
3. In the `UpdateCommand` event handler, programmatically access the Web control value and pass it into the appropriate BLL method.

In this tutorial we will create a richer editing interface for the DataList, one that includes DropDownLists and a CheckBox. In particular, we'll create a DataList that lists product information and permits the product s name, supplier, category, and discontinued status to be updated (see Figure 1).

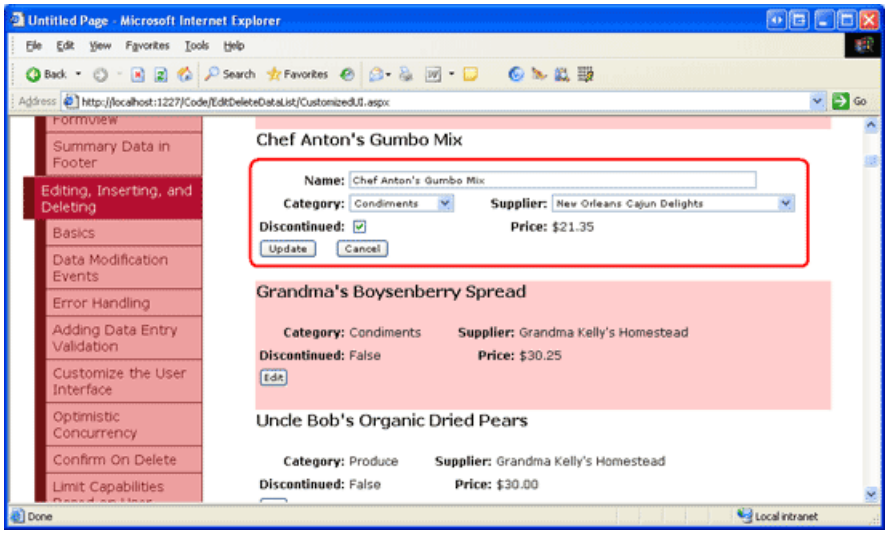


Figure 1: The Editing Interface Includes a TextBox, Two DropDownLists, and a CheckBox (Click to view full-size image)

Step 1: Displaying Product Information

Before we can create the DataList s editable interface, we first need to build the read-only interface. Start by opening the `CustomizedUI.aspx` page from the `EditDeleteDataList` folder and, from the Designer, add a DataList

to the page, setting its `ID` property to `Products`. From the DataList s smart tag, create a new ObjectDataSource. Name this new ObjectDataSource `ProductsDataSource` and configure it to retrieve data from the `ProductsBLL` class s `GetProducts` method. As with the previous editable DataList tutorials, we'll update the edited product s information by going directly to the Business Logic Layer. Accordingly, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None).

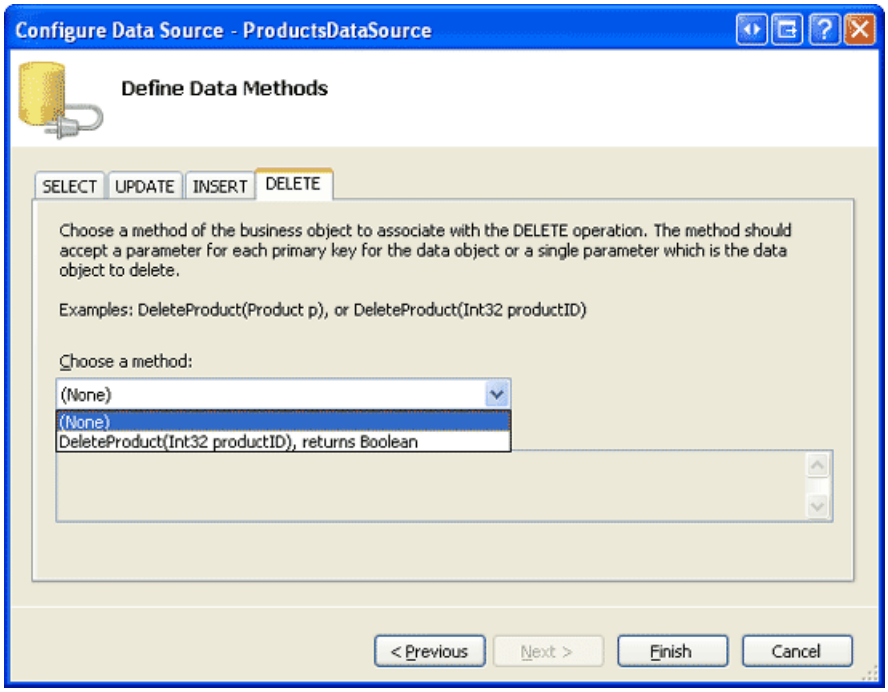


Figure 2: Set the UPDATE, INSERT, and DELETE Tabs Drop-Down Lists to (None) (Click to view full-size image)

After configuring the ObjectDataSource, Visual Studio will create a default `ItemTemplate` for the DataList that lists the name and value for each of the data fields returned. Modify the `ItemTemplate` so that the template lists the product name in an `<h4>` element along with the category name, supplier name, price, and discontinued status. Moreover, add an Edit button, making sure that its `CommandName` property is set to Edit. The declarative markup for my `ItemTemplate` follows:

```
<ItemTemplate>
    <h4>
        <asp:Label ID="ProductNameLabel" runat="server"
            Text='<%# Eval("ProductName") %>' />
    </h4>
    <table border="0">
        <tr>
            <td class="ProductPropertyLabel">Category:</td>
            <td class="ProductPropertyValue">
                <asp:Label ID="CategoryNameLabel" runat="server"
                    Text='<%# Eval("CategoryName") %>' />
            </td>
            <td class="ProductPropertyLabel">Supplier:</td>
            <td class="ProductPropertyValue">
                <asp:Label ID="SupplierNameLabel" runat="server"
                    Text='<%# Eval("SupplierName") %>' />
            </td>
        </tr>
        <tr>
            <td class="ProductPropertyLabel">Discontinued:</td>
            <td class="ProductPropertyValue">
                <asp:Label ID="DiscontinuedLabel" runat="server"
                    Text='<%# Eval("Discontinued") %>' />
            </td>
            <td class="ProductPropertyLabel">Price:</td>
            <td class="ProductPropertyValue">
                <asp:Label ID="UnitPriceLabel" runat="server"
                    Text='<%# Eval("UnitPrice", "{0:C}") %>' />
            </td>
        </tr>
        <tr>
            <td colspan="4">
                <asp:Button runat="Server" ID="EditButton"
                    Text="Edit" CommandName="Edit" />
            </td>
        </tr>
    </table>
    <br />
</ItemTemplate>
```

The above markup lays out the product information using an <h4> heading for the product s name and a four-column `<table>` for the remaining fields. The `ProductPropertyLabel` and `ProductPropertyValue` CSS classes, defined in `Styles.css`, have been discussed in previous tutorials. Figure 3 shows our progress when viewed through a browser.

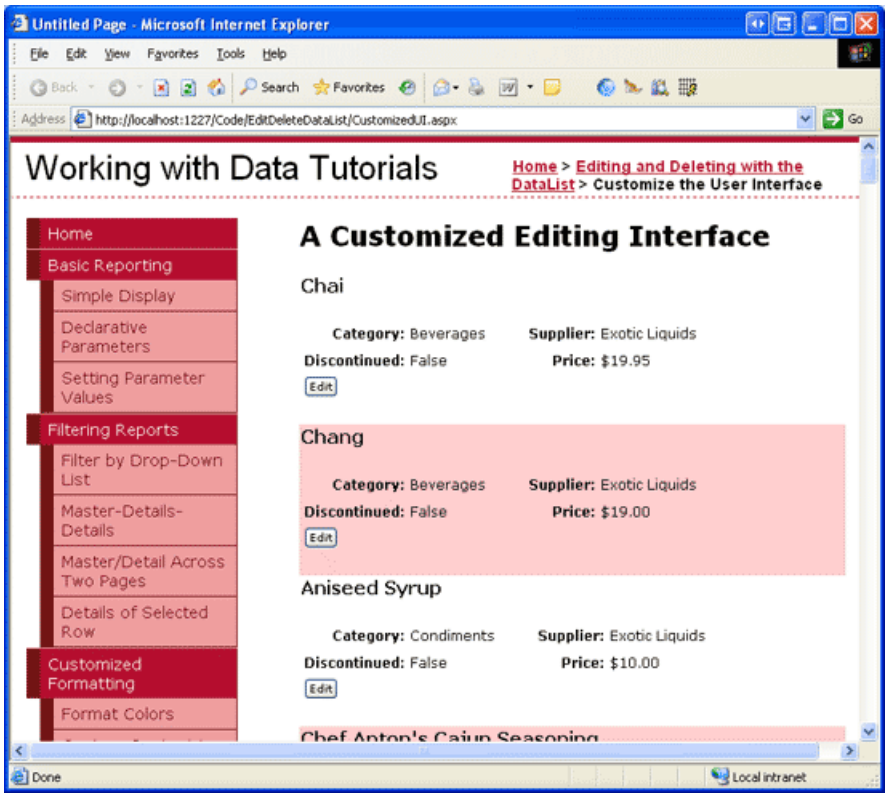


Figure 3: The Name, Supplier, Category, Discontinued Status, and Price of Each Product is Displayed (Click to view full-size image)

Step 2: Adding the Web Controls to the Editing Interface

The first step in building the customized DataList editing interface is to add the needed Web controls to the `EditItemTemplate`. In particular, we need a DropDownList for the category, another for the supplier, and a CheckBox for the discontinued state. Since the product s price is not editable in this example, we can continue displaying it using a Label Web control.

To customize the editing interface, click on the Edit Templates link from the DataList s smart tag and choose the `EditItemTemplate` option from the drop-down list. Add a DropDownList to the `EditItemTemplate` and set its `ID` to `Categories`.

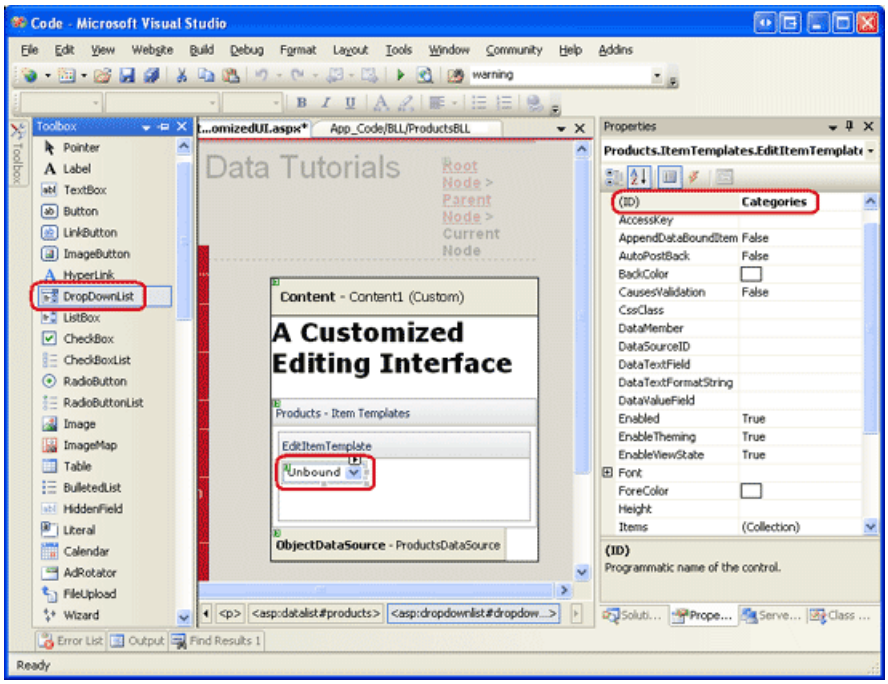


Figure 4: Add a DropDownList for the Categories (Click to view full-size image)

Next, from the DropDownList s smart tag, select the Choose Data Source option and create a new ObjectDataSource named `CategoriesDataSource`. Configure this ObjectDataSource to use the `CategoriesBLL` class s `GetCategories()` method (see Figure 5). Next, the DropDownList s Data Source Configuration Wizard prompts for the data fields to use for each `ListItem` s `Text` and `Value` properties. Have the DropDownList display the `CategoryName` data field and use the `CategoryID` as the value, as shown in Figure 6.

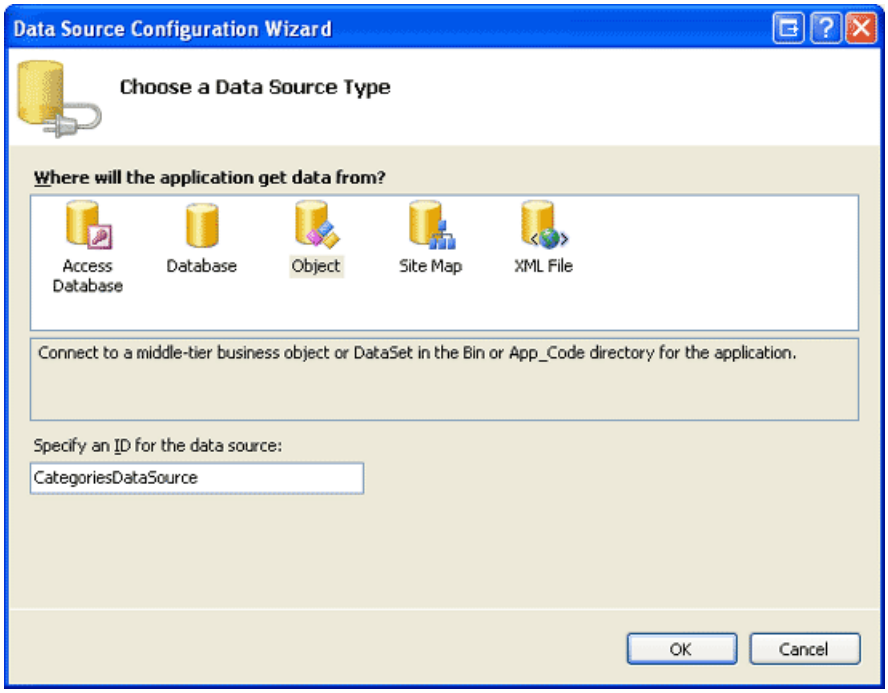


Figure 5: Create a New ObjectDataSource Named `CategoriesDataSource` (Click to view full-size image)

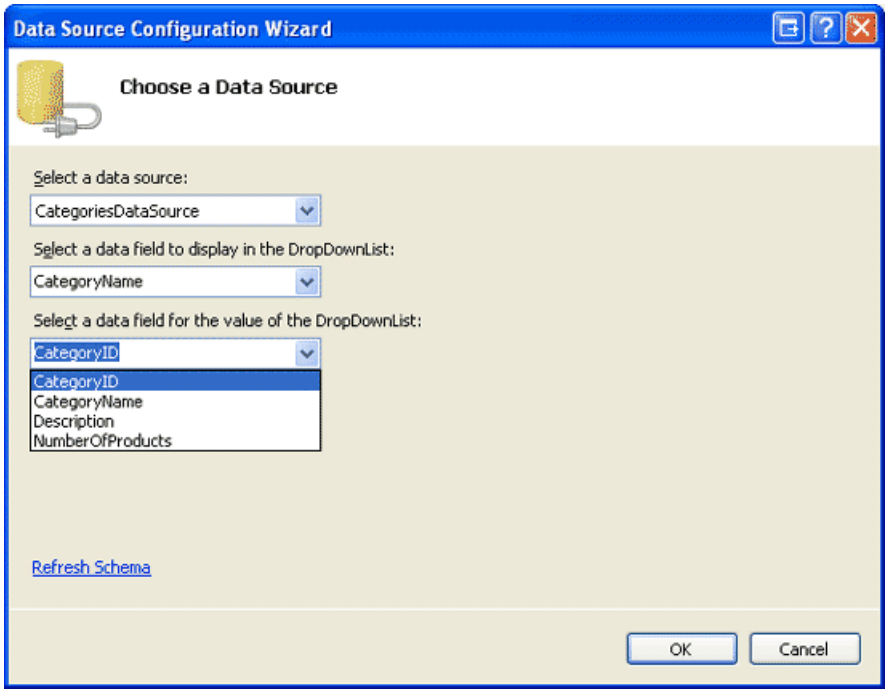


Figure 6: Configure the DropDownList s Display and Value Fields (Click to view full-size image)

Repeat this series of steps to create a DropDownList for the suppliers. Set the `ID` for this DropDownList to `Suppliers` and name its ObjectDataSource `SuppliersDataSource`.

After adding the two DropDownLists, add a CheckBox for the discontinued state and a TextBox for the product s name. Set the `ID` s for the CheckBox and TextBox to `Discontinued` and `ProductName`, respectively. Add a RequiredFieldValidator to ensure that the user provides a value for the product s name.

Lastly, add the Update and Cancel buttons. Remember that for these two buttons it is imperative that their

`CommandName` properties are set to Update and Cancel, respectively.

Feel free to lay out the editing interface however you like. I ve opted to use the same four-column `<table>` layout from the read-only interface, as the following declarative syntax and screen shot illustrates:

```
<EditItemTemplate>
    <h4>
        <asp:Label ID="ProductNameLabel" runat="server"
            Text='<%# Eval("ProductName") %>' />
    </h4>
    <table border="0">
        <tr>
            <td class="ProductPropertyLabel">Name:</td>
            <td colspan="3" class="ProductPropertyValue">
                <asp:TextBox runat="server" ID="ProductName" Width="90%" />
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                    ControlToValidate="ProductName"
                    ErrorMessage="You must enter a name for the product."
                    runat="server">*</asp:RequiredFieldValidator>
            </td>
        </tr>
        <tr>
            <td class="ProductPropertyLabel">Category:</td>
            <td class="ProductPropertyValue">
                <asp:DropDownList ID="Categories" runat="server"
                    DataSourceID="CategoriesDataSource"
                    DataTextField="CategoryName" DataValueField="CategoryID" />
            </td>
            <td class="ProductPropertyLabel">Supplier:</td>
            <td class="ProductPropertyValue">
                <asp:DropDownList ID="Suppliers"
                    DataSourceID="SuppliersDataSource" DataTextField="CompanyName"
                    DataValueField="SupplierID" runat="server" />
            </td>
        </tr>
        <tr>
            <td class="ProductPropertyLabel">Discontinued:</td>
            <td class="ProductPropertyValue">
                <asp:CheckBox runat="server" id="Discontinued" />
            </td>
            <td class="ProductPropertyLabel">Price:</td>
            <td class="ProductPropertyValue">
                <asp:Label ID="UnitPriceLabel" runat="server"
                    Text='<%# Eval("UnitPrice", "{0:C}") %>' />
            </td>
        </tr>
        <tr>
            <td colspan="4">
                <asp:Button runat="Server" ID="UpdateButton" CommandName="Update"
                    Text="Update" />

                <asp:Button runat="Server" ID="CancelButton" CommandName="Cancel"
                    Text="Cancel" CausesValidation="False" />
            </td>
        </tr>
    </table>
    <br />
    <asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
        OldValuesParameterFormatString="original_{0}" SelectMethod="GetCategories"
        TypeName="CategoriesBLL">
    </asp:ObjectDataSource>
    <asp:ObjectDataSource ID="SuppliersDataSource" runat="server"
        OldValuesParameterFormatString="original_{0}" SelectMethod="GetSuppliers"
        TypeName="SuppliersBLL">
    </asp:ObjectDataSource>
</EditItemTemplate>
```

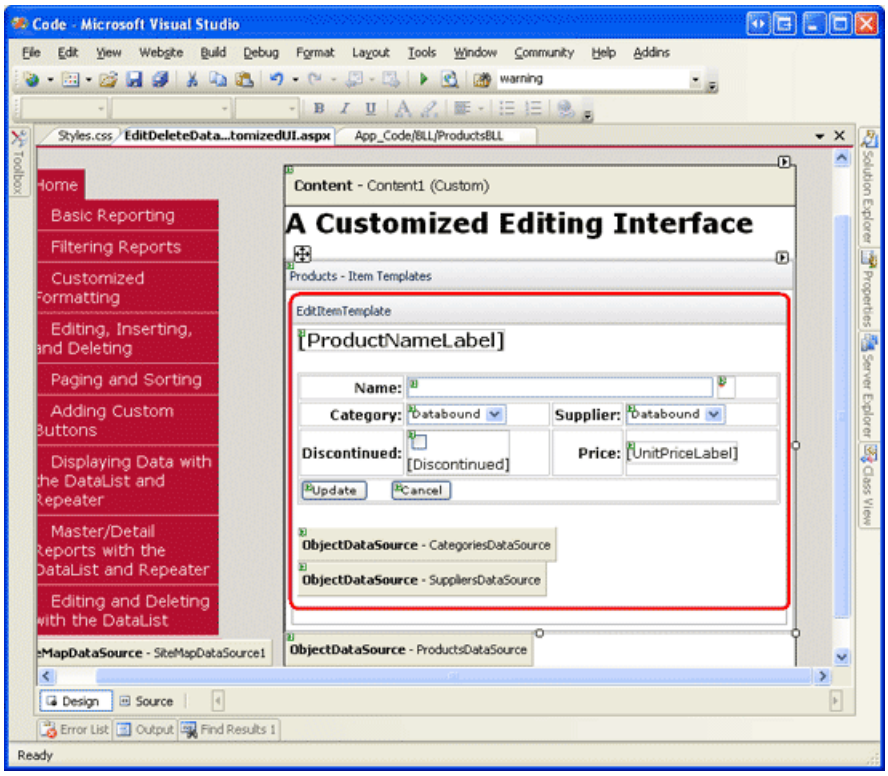


Figure 7: The Editing Interface is Laid Out like the Read-Only Interface (Click to view full-size image)

Step 3: Creating the EditCommand and CancelCommand Event Handlers

Currently, there is no databinding syntax in the `EditItemTemplate` (except for the `UnitPriceLabel`, which was copied over verbatim from the `ItemTemplate`). We'll add the databinding syntax momentarily, but first let s create the event handlers for the DataList s `EditCommand` and `CancelCommand` events. Recall that the responsibility of the `EditCommand` event handler is to render the editing interface for the DataList item whose Edit button was clicked, whereas the `CancelCommand` s job is to return the DataList to its pre-editing state.

Create these two event handlers and have them use the following code:

```
Protected Sub Products_EditCommand(source As Object, e As DataListCommandEventArgs) _
    Handles Products.EditCommand
    ' Set the DataList's EditItemIndex property to the
    ' index of the DataListItem that was clicked
    Products.EditItemIndex = e.Item.ItemIndex
    ' Rebind the data to the DataList
    Products.DataBind()
End Sub
Protected Sub Products_CancelCommand(source As Object, e As DataListCommandEventArgs) _
    Handles Products.CancelCommand
    ' Set the DataList's EditItemIndex property to -1
    Products.EditItemIndex = -1
    ' Rebind the data to the DataList
    Products.DataBind()
End Sub
```

With these two event handlers in place, clicking the Edit button displays the editing interface and clicking the Cancel button returns the edited item to its read-only mode. Figure 8 shows the DataList after the Edit button has been clicked for Chef Anton s Gumbo Mix. Since we ve yet to add any databinding syntax to the editing interface, the `ProductName` TextBox is blank, the `Discontinued` CheckBox unchecked, and the first items selected from the `Categories` and `Suppliers` DropDownLists.

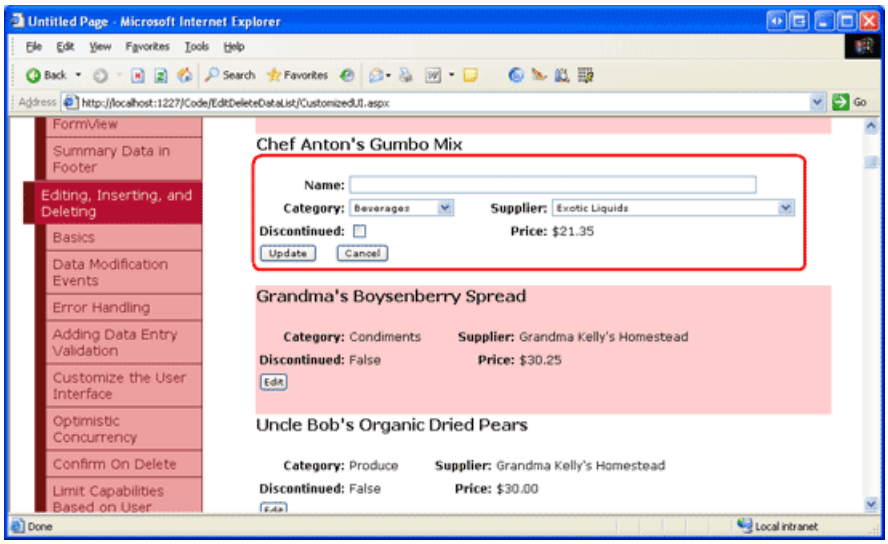


Figure 8: Clicking the Edit Button Displays the Editing Interface (Click to view full-size image)

Step 4: Adding the DataBinding Syntax to the Editing Interface

To have the editing interface display the current product s values, we need to use databinding syntax to assign the data field values to the appropriate Web control values. The databinding syntax can be applied through the Designer by going to the Edit Templates screen and selecting the Edit DataBindings link from the Web controls smart tags. Alternatively, the databinding syntax can be added directly to the declarative markup.

Assign the `ProductName` data field value to the `ProductName` TextBox s `Text` property, the `CategoryID` and `SupplierID` data field values to the `Categories` and `Suppliers` DropDownLists `SelectedValue` properties, and the `Discontinued` data field value to the `Discontinued` CheckBox s `Checked` property. After making these changes, either through the Designer or directly through the declarative markup, revisit the page through a browser and click the Edit button for Chef Anton s Gumbo Mix. As Figure 9 shows, the databinding syntax has added the current values into the TextBox, DropDownLists, and CheckBox.

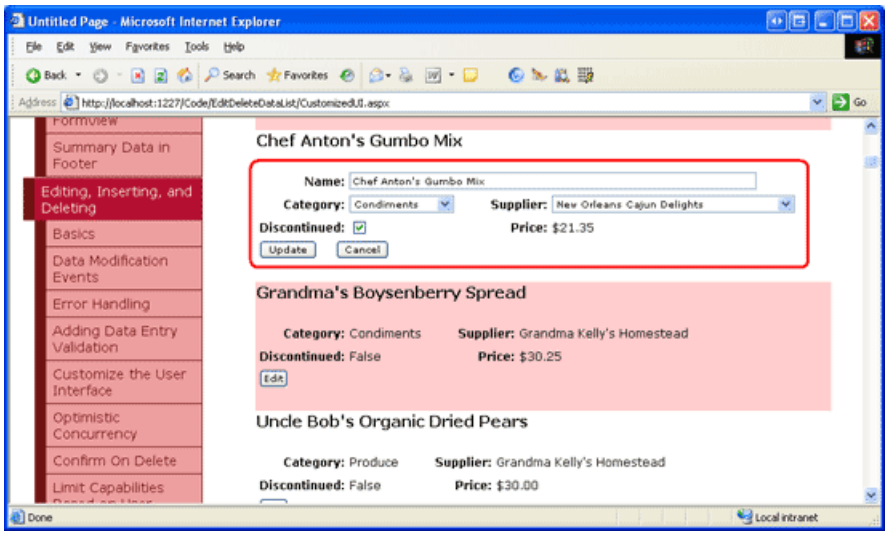


Figure 9: Clicking the Edit Button Displays the Editing Interface (Click to view full-size image)

Step 5: Saving the User s Changes in the UpdateCommand Event Handler

When the user edits a product and clicks the Update button, a postback occurs and the DataList s `UpdateCommand` event fires. In the event handler, we need to read the values from the Web controls in the `EditItemTemplate` and interface with the BLL to update the product in the database. As we ve seen in previous tutorials, the `ProductID` of

the updated product is accessible through the `DataKeys` collection. The user-entered fields are accessed by programmatically referencing the Web controls using `FindControl("controlID")`, as the following code shows:

```
Protected Sub Products_UpdateCommand(source As Object, e As DataListCommandEventArgs) _
    Handles Products.UpdateCommand
    If Not Page.IsValid Then
        Exit Sub
    End If
    ' Read in the ProductID from the DataKeys collection
    Dim productID As Integer = Convert.ToInt32(Products.DataKeys(e.Item.ItemIndex))
    ' Read in the product name and price values
    Dim productName As TextBox = CType(e.Item.FindControl("ProductName"), TextBox)
    Dim categories As DropDownList=CType(e.Item.FindControl("Categories"), DropDownList)
    Dim suppliers As DropDownList = CType(e.Item.FindControl("Suppliers"), DropDownList)
    Dim discontinued As CheckBox = CType(e.Item.FindControl("Discontinued"), CheckBox)
    Dim productNameValue As String = Nothing
    If productName.Text.Trim().Length > 0 Then
        productNameValue = productName.Text.Trim()
    End If
    Dim categoryIDValue As Integer = Convert.ToInt32(categories.SelectedValue)
    Dim supplierIDValue As Integer = Convert.ToInt32(suppliers.SelectedValue)
    Dim discontinuedValue As Boolean = discontinued.Checked
    ' Call the ProductsBLL's UpdateProduct method...
    Dim productsAPI As New ProductsBLL()
    productsAPI.UpdateProduct(productNameValue, categoryIDValue, supplierIDValue, _
        discontinuedValue, productID)
    ' Revert the DataList back to its pre-editing state
    Products.EditItemIndex = -1
    Products.DataBind()
End Sub
```

The code starts out by consulting the `Page.IsValid` property to ensure that all the validation controls on the page are valid. If `Page.IsValid` is `True`, the edited product s `ProductID` value is read from the `DataKeys` collection and the data entry Web controls in the `EditItemTemplate` are programmatically referenced. Next, the values from these Web controls are read into variables that are then passed into the appropriate `UpdateProduct` overload. After updating the data, the DataList is returned to its pre-editing state.

> **NOTE**
> I ve omitted the exception handling logic added in the Handling BLL- and DAL-Level Exceptions tutorial in order to keep the code and this example focused. As an exercise, add this functionality after completing this tutorial.

Step 6: Handling NULL CategoryID and SupplierID Values

The Northwind database allows for `NULL` values for the `Products` table s `CategoryID` and `SupplierID` columns. However, our editing interface doesn t currently accommodate `NULL` values. If we attempt to edit a product that has a `NULL` value for either its `CategoryID` or `SupplierID` columns, we'll get an `ArgumentOutOfRangeException` with an error message similar to: *'Categories' has a SelectedValue which is invalid because it does not exist in the list of items.* Also, there s currently no way to change a product s category or supplier value from a non-`NULL` value to a `NULL` one.

To support `NULL` values for the category and supplier DropDownLists, we need to add an additional `ListItem`. I ve chosen to use (None) as the `Text` value for this `ListItem`, but you can change it to something else (such as an empty string) if you d like. Finally, remember to set the DropDownLists `AppendDataBoundItems` to `True`; if you forget to do so, the categories and suppliers bound to the DropDownList will overwrite the statically-added `ListItem`.

After making these changes, the DropDownLists markup in the DataList s `EditItemTemplate` should look similar to

the following:

```
<asp:DropDownList ID="Categories" DataSourceID="CategoriesDataSource"
    DataTextField="CategoryName" DataValueField="CategoryID"  runat="server"
    SelectedValue='<%# Eval("CategoryID") %>' AppendDataBoundItems="True">
    <asp:ListItem Value=" Selected="True">(None)</asp:ListItem>
</asp:DropDownList>
...
<asp:DropDownList ID="Suppliers" DataSourceID="SuppliersDataSource"
    DataTextField="CompanyName" DataValueField="SupplierID" runat="server"
    SelectedValue='<%# Eval("SupplierID") %>' AppendDataBoundItems="True">
    <asp:ListItem Value=" Selected="True">(None)</asp:ListItem>
</asp:DropDownList>
```

NOTE

Static `ListItem` s can be added to a DropDownList through the Designer or directly through the declarative syntax. When adding a DropDownList item to represent a database `NULL` value, be sure to add the `ListItem` through the declarative syntax. If you use the `ListItem` Collection Editor in the Designer, the generated declarative syntax will omit the `Value` setting altogether when assigned a blank string, creating declarative markup like: `<asp:ListItem>(None)</asp:ListItem>`. While this may look harmless, the missing `Value` causes the DropDownList to use the `Text` property value in its place. That means that if this `NULL` `ListItem` is selected, the value (None) will be attempted to be assigned to the product data field (`CategoryID` or `SupplierID`, in this tutorial), which will result in an exception. By explicitly setting `Value=""`, a `NULL` value will be assigned to the product data field when the `NULL` `ListItem` is selected.

Take a moment to view our progress through a browser. When editing a product, note that the `Categories` and `Suppliers` DropDownLists both have a (None) option at the start of the DropDownList.

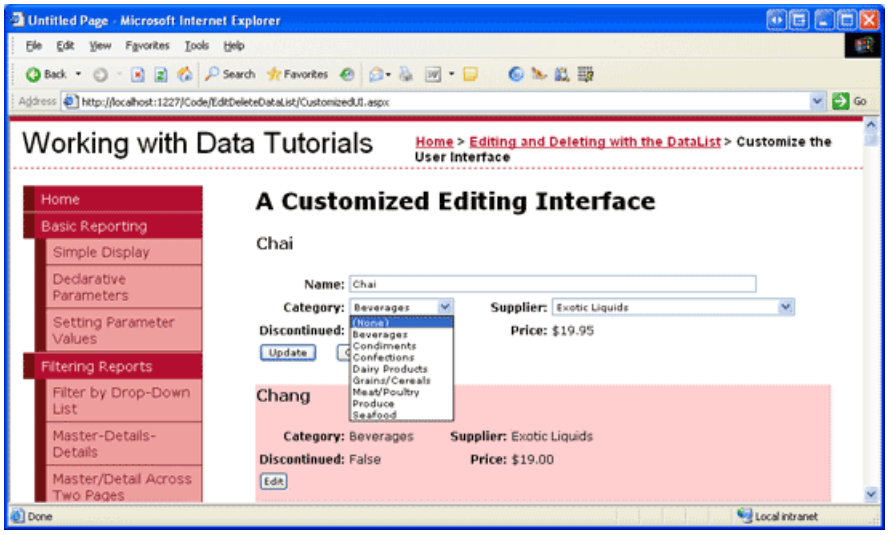


Figure 10: The `Categories` and `Suppliers` DropDownLists include a (None) Option (Click to view full-size image)

To save the (None) option as a database `NULL` value, we need to return to the `UpdateCommand` event handler. Change the `categoryIDValue` and `supplierIDValue` variables to be nullable integers and assign them a value other than `Nothing` only if the DropDownList s `SelectedValue` is not an empty string:

```
Dim categoryIDValue As Nullable(Of Integer) = Nothing
If Not String.IsNullOrEmpty(categories.SelectedValue) Then
    categoryIDValue = Convert.ToInt32(categories.SelectedValue)
End If
Dim supplierIDValue As Nullable(Of Integer) = Nothing
If Not String.IsNullOrEmpty(suppliers.SelectedValue) Then
    supplierIDValue = Convert.ToInt32(suppliers.SelectedValue)
End If
```

With this change, a value of `Nothing` will be passed into the `UpdateProduct` BLL method if the user selected the (None) option from either of the drop-down lists, which corresponds to a `NULL` database value.

Summary

In this tutorial we saw how to create a more complex DataList editing interface that included three different input Web controls a TextBox, two DropDownLists, and a CheckBox along with validation controls. When building the editing interface, the steps are the same regardless of the Web controls being used: start by adding the Web controls to the DataList s `EditItemTemplate`; use databinding syntax to assign the corresponding data field values with the appropriate Web control properties; and, in the `UpdateCommand` event handler, programmatically access the Web controls and their appropriate properties, passing their values into the BLL.

When creating an editing interface, whether it s composed of just TextBoxes or a collection of different Web controls, be sure to correctly handle database `NULL` values. When accounting for `NULL` s, it is imperative that you not only correctly display an existing `NULL` value in the editing interface, but also that you offer a means for marking a value as `NULL`. For DropDownLists in DataLists, that typically means adding a static `ListItem` whose `Value` property is explicitly set to an empty string (`Value=""`), and adding a bit of code to the `UpdateCommand` event handler to determine whether or not the `NULL``ListItem` was selected.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Dennis Patterson, David Suru, and Randy Schmidt. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Paging and Sorting with the DataList and Repeater

2/20/2020 • 2 minutes to read • Edit Online

These tutorials show how to add paging support to the DataList or Repeater control to create a very flexible UI for paging and sorting.

- Paging Report Data in a DataList or Repeater Control (C#)
- Sorting Data in a DataList or Repeater Control (C#)
- Paging Report Data in a DataList or Repeater Control (VB)
- Sorting Data in a DataList or Repeater Control (VB)

Paging Report Data in a DataList or Repeater Control (C#)

2/20/2020 • 19 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> While neither the DataList nor Repeater offer automatic paging or sorting support, this tutorial shows how to add paging support to the DataList or Repeater, which allows for much more flexible paging and data display interfaces.

Introduction

Paging and sorting are two very common features when displaying data in an online application. For example, when searching for ASP.NET books at an online bookstore, there may be hundreds of such books, but the report listing the search results lists only ten matches per page. Moreover, the results can be sorted by title, price, page count, author name, and so on. As we discussed in the Paging and Sorting Report Data tutorial, the GridView, DetailsView, and FormView controls all provide built-in paging support that can be enabled at the tick of a checkbox. The GridView also includes sorting support.

Unfortunately, neither the DataList nor Repeater offer automatic paging or sorting support. In this tutorial we'll examine how to add paging support to the DataList or Repeater. We must manually create the paging interface, display the appropriate page of records, and remember the page being visited across postbacks. While this does take more time and code than with the GridView, DetailsView, or FormView, the DataList and Repeater allow for much more flexible paging and data display interfaces.

> **NOTE**
>
> This tutorial focuses exclusively on paging. In the next tutorial we'll turn our attention to adding sorting capabilities.

Step 1: Adding the Paging and Sorting Tutorial Web Pages

Before we start this tutorial, let s first take a moment to add the ASP.NET pages we'll need for this tutorial and the next one. Start by creating a new folder in the project named `PagingSortingDataListRepeater`. Next, add the following five ASP.NET pages to this folder, having all of them configured to use the master page `Site.master`:

- `Default.aspx`
- `Paging.aspx`
- `Sorting.aspx`
- `SortingWithDefaultPaging.aspx`
- `SortingWithCustomPaging.aspx`

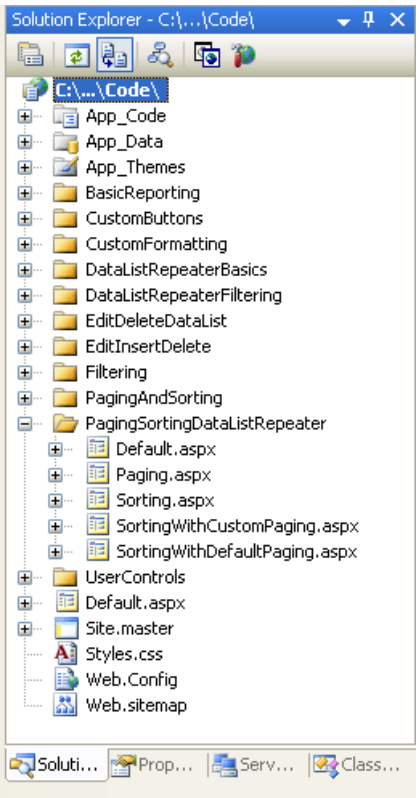


Figure 1: Create a `PagingSortingDataListRepeater` Folder and Add the Tutorial ASP.NET Pages

Next, open the `Default.aspx` page and drag the `SectionLevelTutorialListing.ascx` User Control from the `UserControls` folder onto the Design surface. This User Control, which we created in the Master Pages and Site Navigation tutorial, enumerates the site map and displays those tutorials in the current section in a bulleted list.

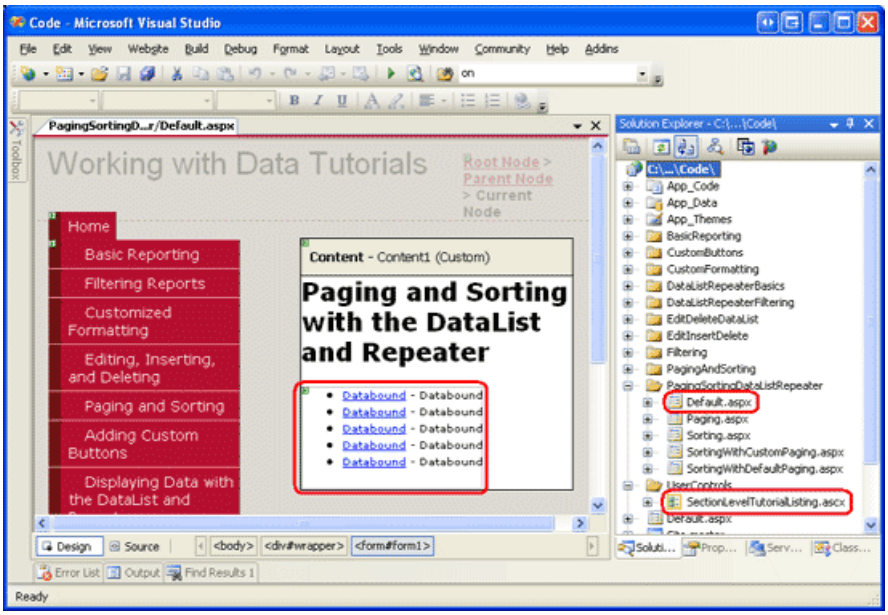


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

In order to have the bulleted list display the paging and sorting tutorials we'll be creating, we need to add them to the site map. Open the `Web.sitemap` file and add the following markup after the Editing and Deleting with the DataList site map node markup:

```
<siteMapNode
    url="~/PagingSortingDataListRepeater/Default.aspx"
    title="Paging and Sorting with the DataList and Repeater"
    description="Paging and Sorting the Data in the DataList and Repeater Controls">
    <siteMapNode
        url="~/PagingSortingDataListRepeater/Paging.aspx"
        title="Paging"
        description="Learn how to page through the data shown
                     in the DataList and Repeater controls." />
    <siteMapNode
        url="~/PagingSortingDataListRepeater/Sorting.aspx"
        title="Sorting"
        description="Sort the data displayed in a DataList or
                     Repeater control." />
    <siteMapNode
        url="~/PagingSortingDataListRepeater/SortingWithDefaultPaging.aspx"
        title="Sorting with Default Paging"
        description="Create a DataList or Repeater control that is paged using
                     default paging and can be sorted." />
    <siteMapNode
        url="~/PagingSortingDataListRepeater/SortingWithCustomPaging.aspx"
        title="Sorting with Custom Paging"
        description="Learn how to sort the data displayed in a DataList or
                     Repeater control that uses custom paging." />
</siteMapNode>
```

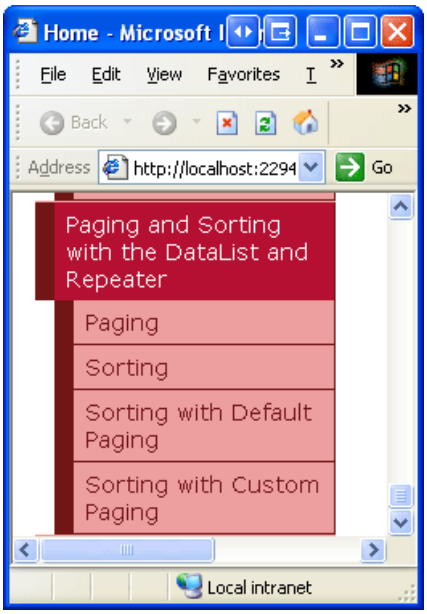


Figure 3: Update the Site Map to Include the New ASP.NET Pages

A Review of Paging

In previous tutorials we saw how to page through the data in the GridView, DetailsView, and FormView controls. These three controls offer a simple form of paging called *default paging* that can be implemented by simply checking the Enable Paging option in the control s smart tag. With default paging, each time a page of data is requested either on the first page visit or when the user navigates to a different page of data the GridView, DetailsView, or FormView control re-requests *all* of the data from the ObjectDataSource. It then snips out the particular set of records to display given the requested page index and the number of records to display per page. We discussed default paging in detail in the Paging and Sorting Report Data tutorial.

Since default paging re-requests all records for each page, it is not practical when paging through sufficiently large amounts of data. For example, imagine paging through 50,000 records with a page size of 10. Each time the user moves to a new page, all 50,000 records must be retrieved from the database, even though only ten of them are displayed.

Custom paging solves the performance concerns of default paging by grabbing only the precise subset of records to display on the requested page. When implementing custom paging, we must write the SQL query that will efficiently return just the correct set of records. We saw how to create such a query using SQL Server 2005 s new `ROW_NUMBER()` keyword back in the Efficiently Paging Through Large Amounts of Data tutorial.

To implement default paging in the DataList or Repeater controls, we can use the `PagedDataSource` class as a wrapper around the `ProductsDataTable` whose contents are being paged. The `PagedDataSource` class has a `DataSource` property that can be assigned to any enumerable object and `PageSize` and `CurrentPageIndex` properties that indicate how many records to show per page and the current page index. Once these properties have been set, the `PagedDataSource` can be used as the data source of any data Web control. The `PagedDataSource`, when enumerated, will only return the appropriate subset of records of its inner `DataSource` based on the `PageSize` and `CurrentPageIndex` properties. Figure 4 depicts the functionality of the `PagedDataSource` class.

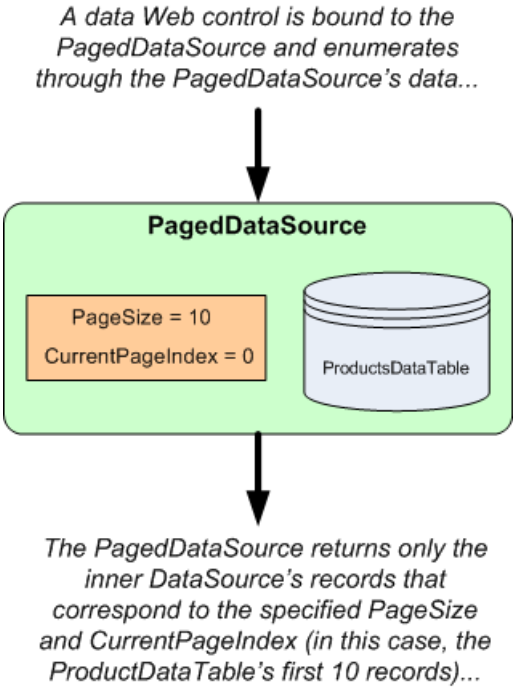


Figure 4: The `PagedDataSource` Wraps an Enumerable Object with a Pageable Interface

The `PagedDataSource` object can be created and configured directly from the Business Logic Layer and bound to a DataList or Repeater through an ObjectDataSource, or can be created and configured directly in the ASP.NET page s code-behind class. If the latter approach is used, we must forgo using the ObjectDataSource and instead bind the paged data to the DataList or Repeater programmatically.

The `PagedDataSource` object also has properties to support custom paging. However, we can bypass using a `PagedDataSource` for custom paging because we already have BLL methods in the `ProductsBLL` class designed for custom paging that return the precise records to display.

In this tutorial we'll look at implementing default paging in a DataList by adding a new method to the `ProductsBLL` class that returns an appropriately configured `PagedDataSource` object. In the next tutorial, we'll see how to use custom paging.

Step 2: Adding a Default Paging Method in the Business Logic Layer

The `ProductsBLL` class currently has a method for returning all product information `GetProducts()` and one for returning a particular subset of products at a starting index `GetProductsPaged(startRowIndex, maximumRows)`. With default paging, the GridView, DetailsView, and FormView controls all use the `GetProducts()` method to retrieve all products, but then use a `PagedDataSource` internally to display only the correct subset of records. To replicate this functionality with the DataList and Repeater controls, we can create a new method in the BLL that mimics this behavior.

Add a method to the `ProductsBLL` class named `GetProductsAsPagedDataSource` that takes in two integer input parameters:

- `pageIndex` the index of the page to display, indexed at zero, and
- `pageSize` the number of records to display per page.

`GetProductsAsPagedDataSource` starts by retrieving *all* of the records from `GetProducts()`. It then creates a `PagedDataSource` object, setting its `CurrentPageIndex` and `PageSize` properties to the values of the passed-in `pageIndex` and `pageSize` parameters. The method concludes by returning this configured `PagedDataSource`:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Select, false)]
public PagedDataSource GetProductsAsPagedDataSource(int pageIndex, int pageSize)
{
    // Get ALL of the products
    Northwind.ProductsDataTable products = GetProducts();
    // Limit the results through a PagedDataSource
    PagedDataSource pagedData = new PagedDataSource();
    pagedData.DataSource = products.Rows;
    pagedData.AllowPaging = true;
    pagedData.CurrentPageIndex = pageIndex;
    pagedData.PageSize = pageSize;
    return pagedData;
}
```

Step 3: Displaying Product Information in a DataList Using Default Paging

With the `GetProductsAsPagedDataSource` method added to the `ProductsBLL` class, we can now create a DataList or Repeater that provides default paging. Start by opening the `Paging.aspx` page in the `PagingSortingDataListRepeater` folder and drag a DataList from the Toolbox onto the Designer, setting the DataList s `ID` property to `ProductsDefaultPaging`. From the DataList s smart tag, create a new ObjectDataSource named `ProductsDefaultPagingDataSource` and configure it so that it retrieves data using the `GetProductsAsPagedDataSource` method.

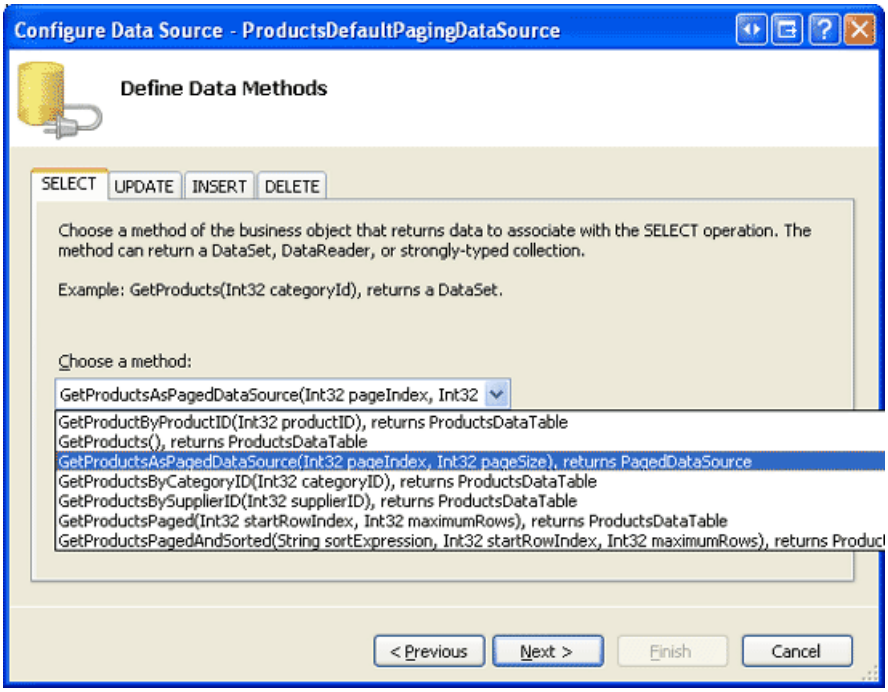


Figure 5: Create an ObjectDataSource and Configure it to Use the `GetProductsAsPagedDataSource()` Method (Click to view full-size image)

Set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) .

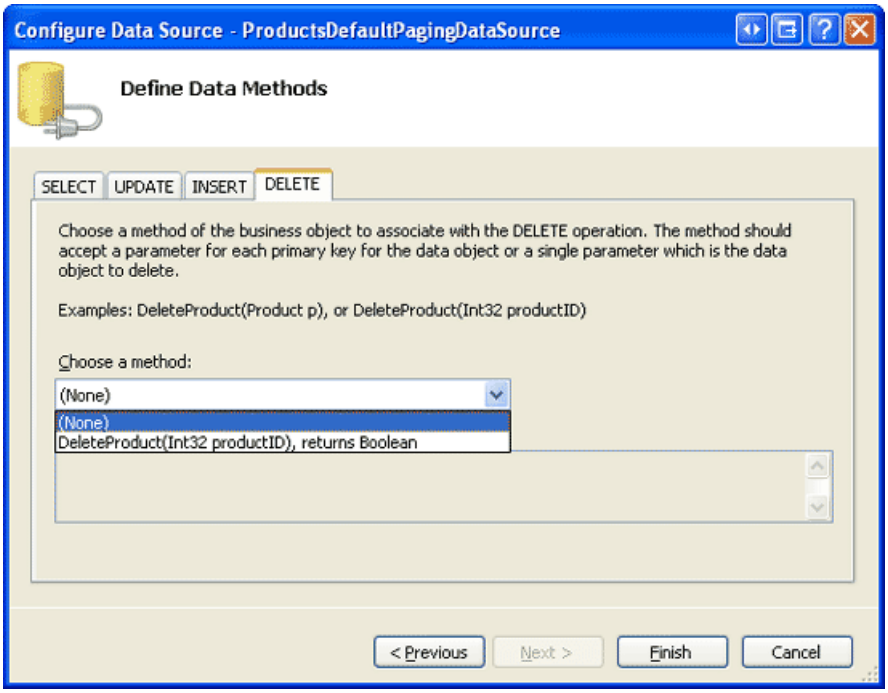


Figure 6: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE tabs to (None) (Click to view full-size image)

Since the `GetProductsAsPagedDataSource` method expects two input parameters, the wizard prompts us for the source of these parameter values.

The page index and page size values must be remembered across postbacks. They can be stored in view state, persisted to the querystring, stored in session variables, or remembered using some other technique. For this tutorial we'll use the querystring, which has the advantage of allowing a particular page of data to be bookmarked.

In particular, use the querystring fields pageIndex and pageSize for the `pageIndex` and `pageSize` parameters, respectively (see Figure 7). Take a moment to set the default values for these parameters, as the querystring values won't be present when a user first visits this page. For `pageIndex`, set the default value to 0 (which will show the first page of data) and `pageSize` s default value to 4.

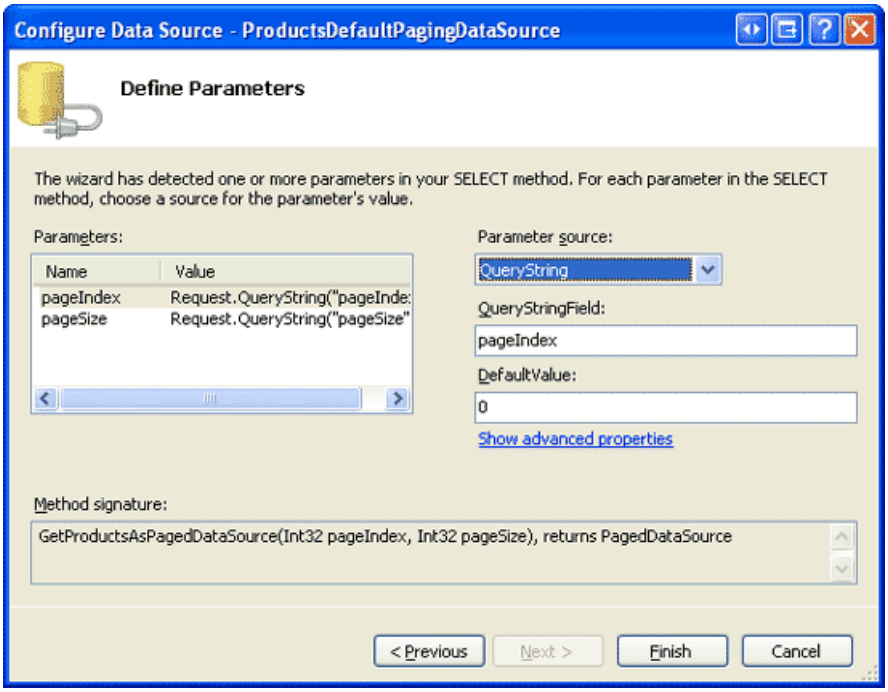


Figure 7: Use the QueryString as the Source for the `pageIndex` and `pageSize` Parameters (Click to view full-size image)

After configuring the ObjectDataSource, Visual Studio automatically creates an `ItemTemplate` for the DataList. Customize the `ItemTemplate` so that only the product s name, category, and supplier are shown. Also set the DataList s `RepeatColumns` property to 2, its `Width` to 100%, and its `ItemStyle` s `Width` to 50%. These width settings will provide equal spacing for the two columns.

After making these changes, the DataList and ObjectDataSource s markup should look similar to the following:

```
<asp:DataList ID="ProductsDefaultPaging" runat="server" Width="100%"
    DataKeyField="ProductID" DataSourceID="ProductsDefaultPagingDataSource"
    RepeatColumns="2" EnableViewState="False">
    <ItemTemplate>
        <h4><asp:Label ID="ProductNameLabel" runat="server"
            Text='<%# Eval("ProductName") %>'></asp:Label></h4>
        Category:
        <asp:Label ID="CategoryNameLabel" runat="server"
            Text='<%# Eval("CategoryName") %>'></asp:Label><br />
        Supplier:
        <asp:Label ID="SupplierNameLabel" runat="server"
            Text='<%# Eval("SupplierName") %>'></asp:Label><br />
        <br />
        <br />
    </ItemTemplate>
    <ItemStyle Width="50%" />
</asp:DataList>
<asp:ObjectDataSource ID="ProductsDefaultPagingDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}" TypeName="ProductsBLL"
    SelectMethod="GetProductsAsPagedDataSource">
    <SelectParameters>
        <asp:QueryStringParameter DefaultValue="0" Name="pageIndex"
             QueryStringField="pageIndex" Type="Int32" />
        <asp:QueryStringParameter DefaultValue="4" Name="pageSize"
             QueryStringField="pageSize" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

> **NOTE**
> Since we are not performing any update or delete functionality in this tutorial, you may disable the DataList s view state to reduce the rendered page size.

When initially visiting this page through a browser, neither the `pageIndex` nor `pageSize` querystring parameters are provided. Hence, the default values of 0 and 4 are used. As Figure 8 shows, this results in a DataList that displays the first four products.

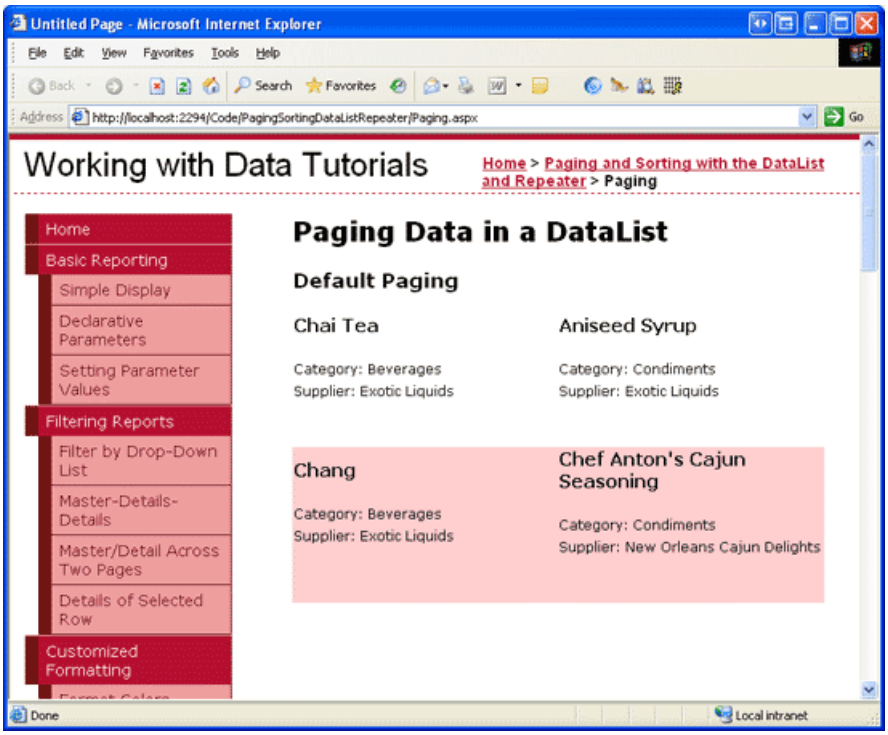


Figure 8: The First Four Products are Listed (Click to view full-size image)

Without a paging interface, there s currently no straightforward means for a user to navigate to the second page of data. We'll create a paging interface in step 4. For now, though, paging can only be accomplished by directly specifying the paging criteria in the querystring. For example, to view the second page, change the URL in the browser s Address bar from `Paging.aspx` to `Paging.aspx?pageIndex=2` and hit Enter. This causes the second page of data to be displayed (see Figure 9).

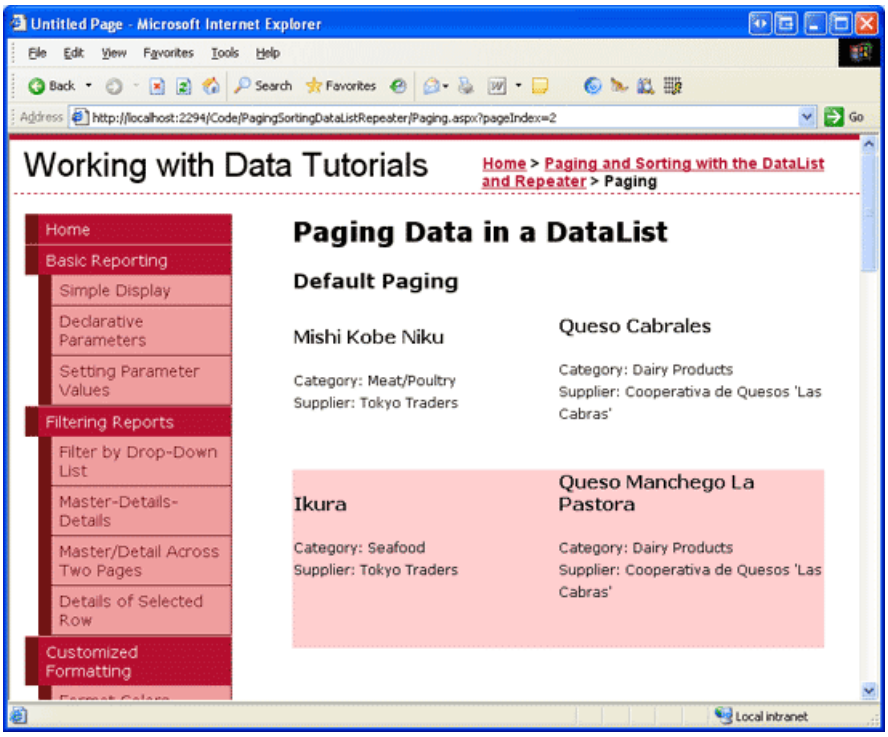


Figure 9: The Second Page of Data is Displayed (Click to view full-size image)

Step 4: Creating the Paging Interface

There are a variety of different paging interfaces that can be implemented. The GridView, DetailsView, and FormView controls provide four different interfaces to choose among:

- **Next, Previous** users can move one page at a time, to either the next or previous one.

- **Next, Previous, First, Last** in addition to Next and Previous buttons, this interface includes First and Last buttons for moving to the very first or very last page.
- **Numeric** lists the page numbers in the paging interface, allowing a user to quickly jump to a particular page.
- **Numeric, First, Last** in addition to the numeric page numbers, includes buttons for moving to the very first or very last page.

For the DataList and Repeater, we are responsible for deciding upon a paging interface and implementing it. This involves creating the needed Web controls in the page and displaying the requested page when a particular paging interface button is clicked. Additionally, certain paging interface controls may need to be disabled. For example, when viewing the first page of data using the Next, Previous, First, Last interface, both the First and Previous buttons would be disabled.

For this tutorial, let s use the Next, Previous, First, Last interface. Add four Button Web controls to the page and set their `ID` s to `FirstPage`, `PrevPage`, `NextPage`, and `LastPage`. Set the `Text` properties to << First, < Prev, Next >, and Last >> .

```
<asp:Button runat="server" ID="FirstPage" Text="<< First" />
<asp:Button runat="server" ID="PrevPage" Text="< Prev" />
<asp:Button runat="server" ID="NextPage" Text="Next >" />
<asp:Button runat="server" ID="LastPage" Text="Last >>" />
```

Next, create a `Click` event handler for each of these Buttons. In a moment we'll add the code necessary to display the requested page.

Remembering the Total Number of Records Being Paged Through

Regardless of the paging interface selected, we need to compute and remember the total number of records being paged through. The total row count (in conjunction with the page size) determines how many total pages of data are being paged through, which determines what paging interface controls are added or are enabled. In the Next, Previous, First, Last interface that we are building, the page count is used in two ways:

- To determine whether we are viewing the last page, in which case the Next and Last buttons are disabled.
- If the user clicks the Last button we need to whisk them to the last page, whose index is one less than the page count.

The page count is calculated as the ceiling of the total row count divided by the page size. For example, if we are paging through 79 records with four records per page, then the page count is 20 (the ceiling of 79 / 4). If we are using the Numeric paging interface, this information informs us as to how many numeric page buttons to display; if our paging interface includes Next or Last buttons, the page count is used to determine when to disable the Next or Last buttons.

If the paging interface includes a Last button, it is imperative that the total number of records being paged through be remembered across postbacks so that when the Last button is clicked we can determine the last page index. To facilitate this, create a `TotalRowCount` property in the ASP.NET page s code-behind class that persists its value to view state:

```
private int TotalRowCount
{
    get
    {
        object o = ViewState["TotalRowCount"];
        if (o == null)
            return -1;
        else
            return (int)o;
    }
    set
    {
        ViewState["TotalRowCount"] = value;
    }
}
```

In addition to `TotalRowCount`, take a minute to create read-only page-level properties for easily accessing the page index, page size, and page count:

```
private int PageIndex
{
    get
    {
        if (!string.IsNullOrEmpty(Request.QueryString["pageIndex"]))
            return Convert.ToInt32(Request.QueryString["pageIndex"]);
        else
            return 0;
    }
}
private int PageSize
{
    get
    {
        if (!string.IsNullOrEmpty(Request.QueryString["pageSize"]))
            return Convert.ToInt32(Request.QueryString["pageSize"]);
        else
            return 4;
    }
}
private int PageCount
{
    get
    {
        if (TotalRowCount <= 0 || PageSize <= 0)
            return 1;
        else
            return ((TotalRowCount + PageSize) - 1) / PageSize;
    }
}
```

Determining the Total Number of Records Being Paged Through

The `PagedDataSource` object returned from the ObjectDataSource s `Select()` method has within it *all* of the product records, even though only a subset of them are displayed in the DataList. The `PagedDataSource` s `Count` property returns only the number of items that will be displayed in the DataList; the `DataSourceCount` property returns the total number of items within the `PagedDataSource`. Therefore, we need to assign the ASP.NET page s `TotalRowCount` property the value of the `PagedDataSource` s `DataSourceCount` property.

To accomplish this, create an event handler for the ObjectDataSource s `Selected` event. In the `Selected` event handler we have access to the return value of the ObjectDataSource s `Select()` method in this case, the `PagedDataSource`.

```
protected void ProductsDefaultPagingDataSource_Selected
    (object sender, ObjectDataSourceStatusEventArgs e)
{
    // Reference the PagedDataSource bound to the DataList
    PagedDataSource pagedData = (PagedDataSource)e.ReturnValue;
    // Remember the total number of records being paged through
    // across postbacks
    TotalRowCount = pagedData.DataSourceCount;
}
```

Displaying the Requested Page of Data

When the user clicks one of the buttons in the paging interface, we need to display the requested page of data. Since the paging parameters are specified via the querystring, to show the requested page of data use `Response.Redirect(url)` to have the user s browser re-request the `Paging.aspx` page with the appropriate paging parameters. For example, to display the second page of data, we would redirect the user to `Paging.aspx?pageIndex=1`.

To facilitate this, create a `RedirectUser(sendUserToPageIndex)` method that redirects the user to `Paging.aspx?pageIndex=sendUserToPageIndex`. Then, call this method from the four Button `Click` event handlers. In the `FirstPage` `Click` event handler, call `RedirectUser(0)`, to send them to the first page; in the `PrevPage` `Click` event handler, use `PageIndex - 1` as the page index; and so on.

```
protected void FirstPage_Click(object sender, EventArgs e)
{
    // Send the user to the first page
    RedirectUser(0);
}
protected void PrevPage_Click(object sender, EventArgs e)
{
    // Send the user to the previous page
    RedirectUser(PageIndex - 1);
}
protected void NextPage_Click(object sender, EventArgs e)
{
    // Send the user to the next page
    RedirectUser(PageIndex + 1);
}
protected void LastPage_Click(object sender, EventArgs e)
{
    // Send the user to the last page
    RedirectUser(PageCount - 1);
}
private void RedirectUser(int sendUserToPageIndex)
{
    // Send the user to the requested page
    Response.Redirect(string.Format("Paging.aspx?pageIndex={0}&pageSize={1}",
        sendUserToPageIndex, PageSize));
}
```

With the `Click` event handlers complete, the DataList s records can be paged through by clicking the buttons. Take a moment to try it out!

Disabling Paging Interface Controls

Currently, all four buttons are enabled regardless of the page being viewed. However, we want to disable the First and Previous buttons when showing the first page of data, and the Next and Last buttons when showing the last page. The `PagedDataSource` object returned by the ObjectDataSource s `Select()` method has properties

`IsFirstPage` and `IsLastPage` that we can examine to determine if we are viewing the first or last page of data.

Add the following to the ObjectDataSource s `Selected` event handler:

```
// Configure the paging interface based on the data in the PagedDataSource
FirstPage.Enabled = !pagedData.IsFirstPage;
PrevPage.Enabled = !pagedData.IsFirstPage;
NextPage.Enabled = !pagedData.IsLastPage;
LastPage.Enabled = !pagedData.IsLastPage;
```

With this addition, the First and Previous buttons will be disabled when viewing the first page, while the Next and Last buttons will be disabled when viewing the last page.

Let s complete the paging interface by informing the user what page they re currently viewing and how many total pages exist. Add a Label Web control to the page and set its `ID` property to `CurrentPageNumber`. Set its `Text` property in the ObjectDataSource s Selected event handler such that it includes the current page being viewed (`PageIndex + 1`) and the total number of pages (`PageCount`).

```
// Display the current page being viewed...
CurrentPageNumber.Text = string.Format("You are viewing page {0} of {1}...",
    PageIndex + 1, PageCount);
```

Figure 10 shows `Paging.aspx` when first visited. Since the querystring is empty, the DataList defaults to showing the first four products; the First and Previous buttons are disabled. Clicking Next displays the next four records (see Figure 11); the First and Previous buttons are now enabled.

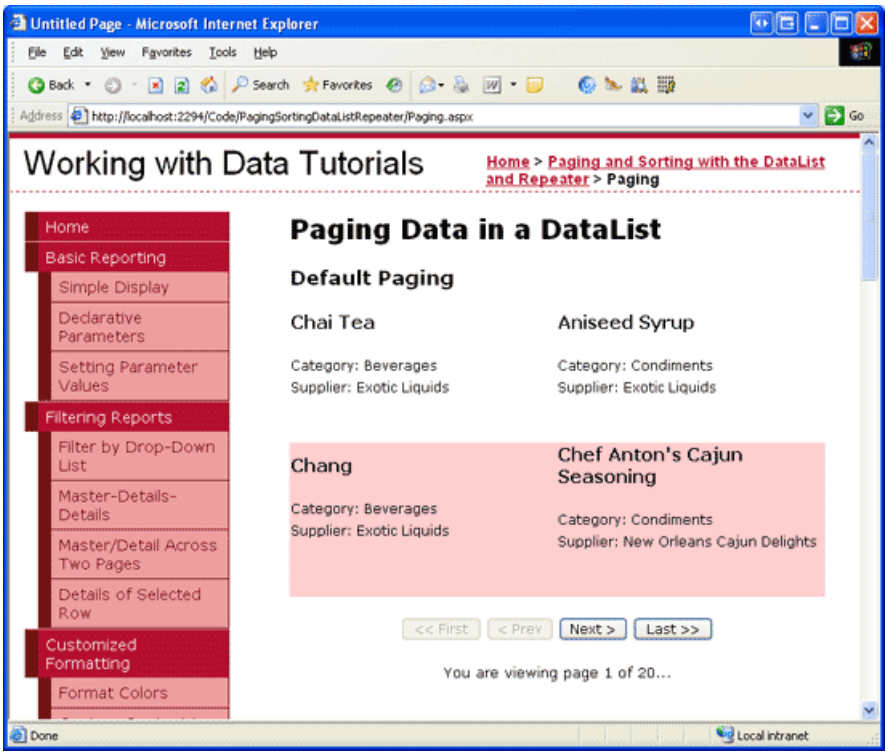


Figure 10: The First Page of Data is Displayed (Click to view full-size image)

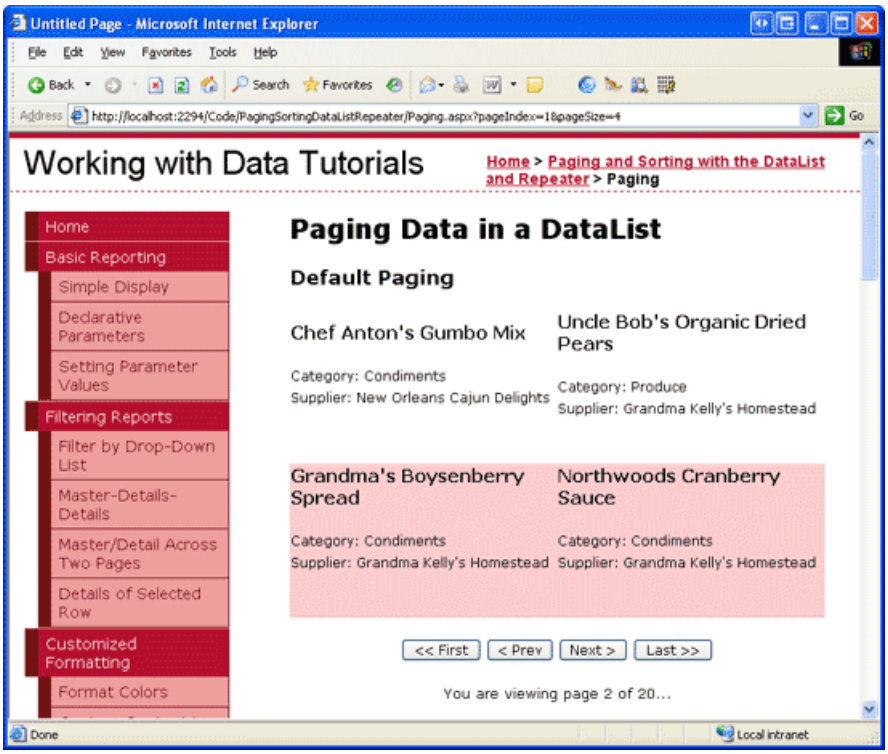


Figure 11: The Second Page of Data is Displayed (Click to view full-size image)

> **NOTE**
>
> The paging interface can be further enhanced by allowing the user to specify how many pages to view per page. For example, a DropDownList could be added listing page size options like 5, 10, 25, 50, and All. Upon selecting a page size, the user would need to be redirected back to `Paging.aspx?pageIndex=0&pageSize=selectedPageSize`. I leave implementing this enhancement as an exercise for the reader.

Using Custom Paging

The DataList pages through its data using the inefficient default paging technique. When paging through sufficiently large amounts of data, it is imperative that custom paging be used. Although the implementation details differ slightly, the concepts behind implementing custom paging in a DataList are the same as with default paging. With custom paging, use the `ProductBLL` class s `GetProductsPaged` method (instead of `GetProductsAsPagedDataSource`). As discussed in the Efficiently Paging Through Large Amounts of Data tutorial, `GetProductsPaged` must be passed the start row index and maximum number of rows to return. These parameters can be maintained through the querystring just like the `pageIndex` and `pageSize` parameters used in default paging.

Since there s no `PagedDataSource` with custom paging, alternative techniques must be used to determine the total number of records being paged through and whether we re displaying the first or last page of data. The `TotalNumberOfProducts()` method in the `ProductsBLL` class returns the total number of products being paged through. To determine if the first page of data is being viewed, examine the start row index if it is zero, then the first page is being viewed. The last page is being viewed if the start row index plus the maximum rows to return is greater than or equal to the total number of records being paged through.

We'll explore implementing custom paging in greater detail in the next tutorial.

Summary

While neither the DataList nor Repeater offers the out of the box paging support found in the GridView, DetailsView, and FormView controls, such functionality can be added with minimal effort. The easiest way to

implement default paging is to wrap the entire set of products within a `PagedDataSource` and then bind the `PagedDataSource` to the DataList or Repeater. In this tutorial we added the `GetProductsAsPagedDataSource` method to the `ProductsBLL` class to return the `PagedDataSource`. The `ProductsBLL` class already contains the methods needed for custom paging `GetProductsPaged` and `TotalNumberOfProducts`.

Along with retrieving either the precise set of records to display for custom paging or all records in a `PagedDataSource` for default paging, we also need to manually add the paging interface. For this tutorial we created a Next, Previous, First, Last interface with four Button Web controls. Also, a Label control displaying the current page number and total number of pages was added.

In the next tutorial we'll see how to add sorting support to the DataList and Repeater. We'll also see how to create a DataList that can be both paged and sorted (with examples using default and custom paging).

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Liz Shulok, Ken Pespisa, and Bernadette Leigh. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Sorting Data in a DataList or Repeater Control (C#)

2/20/2020 • 26 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll examine how to include sorting support in the DataList and Repeater, as well as how to construct a DataList or Repeater whose data can be paged and sorted.

Introduction

In the previous tutorial we examined how to add paging support to a DataList. We created a new method in the `ProductsBLL` class (`GetProductsAsPagedDataSource`) that returned a `PagedDataSource` object. When bound to a DataList or Repeater, the DataList or Repeater would display just the requested page of data. This technique is similar to what is used internally by the GridView, DetailsView, and FormView controls to provide their built-in default paging functionality.

In addition to offering paging support, the GridView also includes out of the box sorting support. Neither the DataList nor Repeater provides built-in sorting functionality; however, sorting features can be added with a bit of code. In this tutorial we'll examine how to include sorting support in the DataList and Repeater, as well as how to construct a DataList or Repeater whose data can be paged and sorted.

A Review of Sorting

As we saw in the Paging and Sorting Report Data tutorial, the GridView control provides out of the box sorting support. Each GridView field can have an associated `SortExpression`, which indicates the data field by which to sort the data. When the GridView s `AllowSorting` property is set to `true`, each GridView field that has a `SortExpression` property value has its header rendered as a LinkButton. When a user clicks a particular GridView field s header, a postback occurs and the data is sorted according to the clicked field s `SortExpression`.

The GridView control has a `SortExpression` property as well, which stores the `SortExpression` of the GridView field the data is sorted by. Additionally, a `SortDirection` property indicates whether the data is to be sorted in ascending or descending order (if a user clicks a particular GridView field s header link twice in succession, the sort order is toggled).

When the GridView is bound to its data source control, it hands off its `SortExpression` and `SortDirection` properties to the data source control. The data source control retrieves the data and then sorts it according to the supplied `SortExpression` and `SortDirection` properties. After sorting the data, the data source control returns it to the GridView.

To replicate this functionality with the DataList or Repeater controls, we must:

- Create a sorting interface
- Remember the data field to sort by and whether to sort in ascending or descending order
- Instruct the ObjectDataSource to sort the data by a particular data field

We'll tackle these three tasks in steps 3 and 4. Following that, we'll examine how to include both paging and sorting support in a DataList or Repeater.

Step 2: Displaying the Products in a Repeater

Before we worry about implementing any of the sorting-related functionality, let s start by listing the products in a Repeater control. Start by opening the `Sorting.aspx` page in the `PagingSortingDataListRepeater` folder. Add a Repeater control to the web page, setting its `ID` property to `SortableProducts`. From the Repeater s smart tag, create a new ObjectDataSource named `ProductsDataSource` and configure it to retrieve data from the `ProductsBLL` class s `GetProducts()` method. Select the (None) option from the drop-down lists in the INSERT, UPDATE, and DELETE tabs.

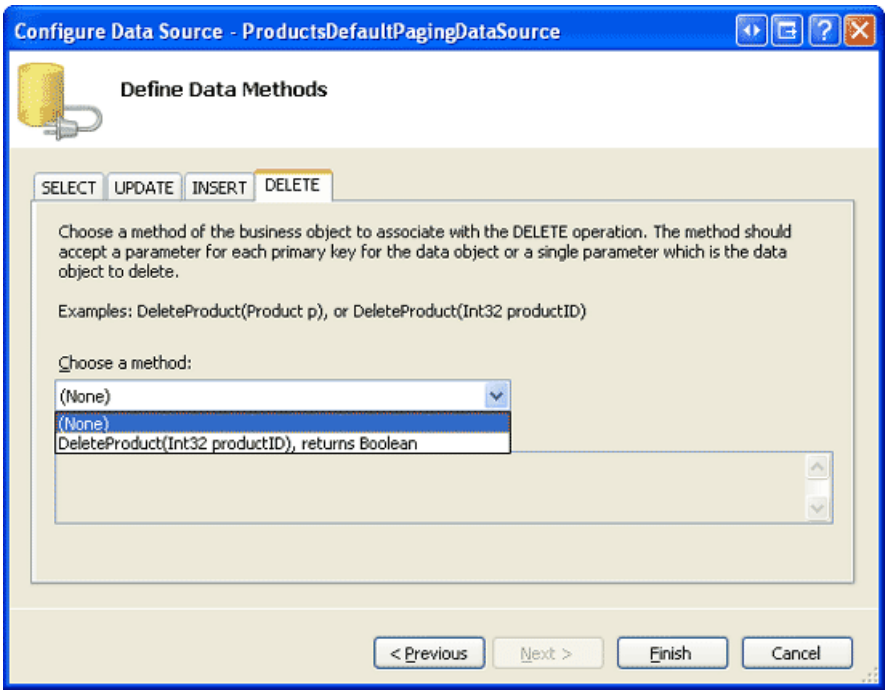


Figure 1: Create an ObjectDataSource and Configure it to Use the `GetProductsAsPagedDataSource()` Method (Click to view full-size image)

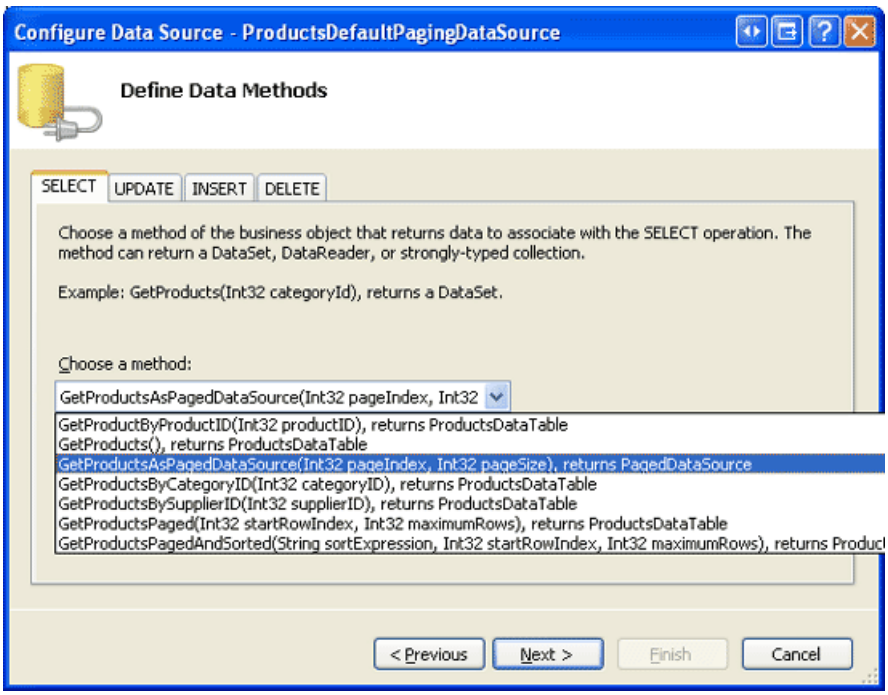


Figure 2: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE tabs to (None) (Click to view full-size image)

Unlike with the DataList, Visual Studio does not automatically create an `ItemTemplate` for the Repeater control after binding it to a data source. Furthermore, we must add this `ItemTemplate` declaratively, as the Repeater control s smart tag lacks the Edit Templates option found in the DataList s. Let s use the same `ItemTemplate` from the previous tutorial, which displayed the product s name, supplier, and category.

After adding the `ItemTemplate`, the Repeater and ObjectDataSource s declarative markup should look similar to the following:

```
<asp:Repeater ID="SortableProducts" DataSourceID="ProductsDataSource"
    EnableViewState="False" runat="server">
    <ItemTemplate>
        <h4><asp:Label ID="ProductNameLabel" runat="server"
            Text='<%# Eval("ProductName") %>'></asp:Label></h4>
        Category:
        <asp:Label ID="CategoryNameLabel" runat="server"
            Text='<%# Eval("CategoryName") %>'></asp:Label><br />
        Supplier:
        <asp:Label ID="SupplierNameLabel" runat="server"
            Text='<%# Eval("SupplierName") %>'></asp:Label><br />
        <br />
        <br />
    </ItemTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}" TypeName="ProductsBLL"
    SelectMethod="GetProducts">
</asp:ObjectDataSource>
```

Figure 3 shows this page when viewed through a browser.

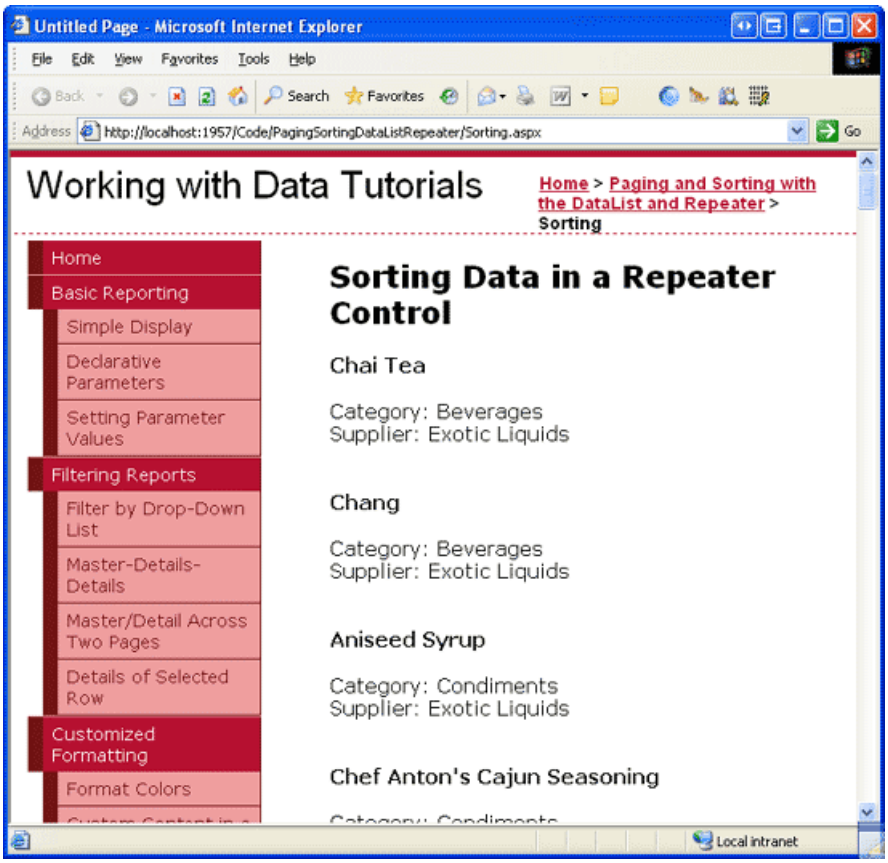


Figure 3: Each Product s Name, Supplier, and Category is Displayed (Click to view full-size image)

Step 3: Instructing the ObjectDataSource to Sort the Data

To sort the data displayed in the Repeater, we need to inform the ObjectDataSource of the sort expression by which the data should be sorted. Before the ObjectDataSource retrieves its data, it first fires its `Selecting` event, which provides an opportunity for us to specify a sort expression. The `Selecting` event handler is passed an object of type `ObjectDataSourceSelectingEventArgs`, which has a property named `Arguments` of type `DataSourceSelectArguments`. The `DataSourceSelectArguments` class is designed to pass data-related requests from a

consumer of data to the data source control, and includes a `SortExpression` property.

To pass sorting information from the ASP.NET page to the ObjectDataSource, create an event handler for the `Selecting` event and use the following code:

```
protected void ProductsDataSource_Selecting
    (object sender, ObjectDataSourceSelectingEventArgs e)
{
    e.Arguments.SortExpression = sortExpression;
}
```

The *sortExpression* value should be assigned the name of the data field to sort the data by (such as ProductName). There is no sort direction-related property, so if you want to sort the data in descending order, append the string DESC to the *sortExpression* value (such as ProductName DESC).

Go ahead and try some different hard-coded values for *sortExpression* and test the results in a browser. As Figure 4 shows, when using ProductName DESC as the *sortExpression*, the products are sorted by their name in reverse alphabetical order.

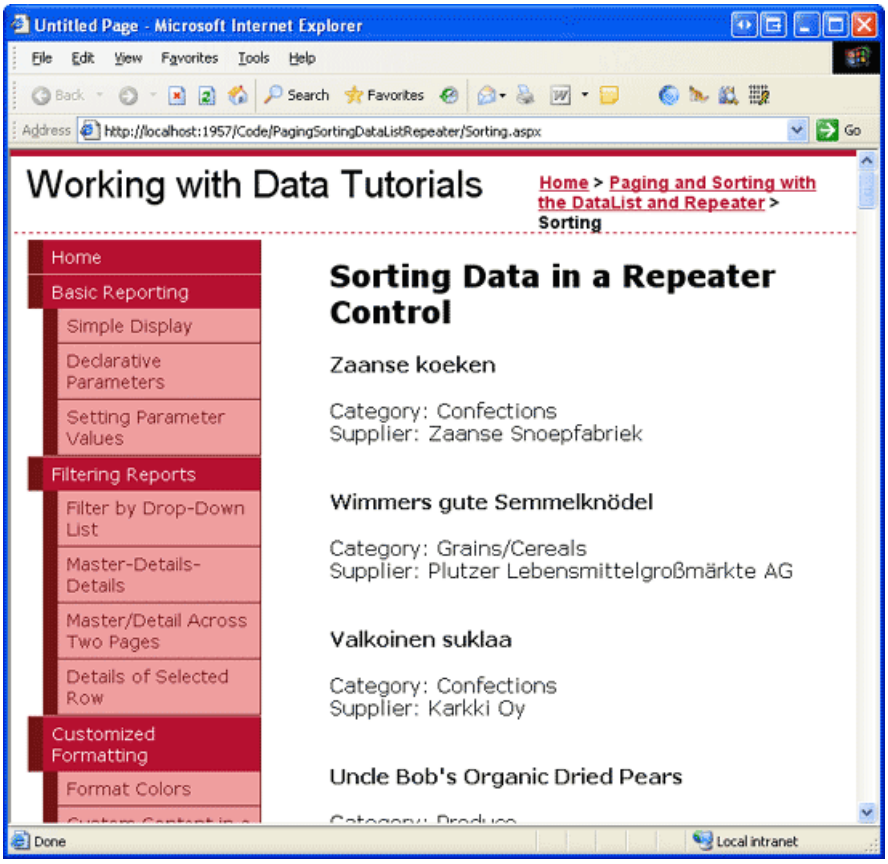


Figure 4: The Products are Sorted by their Name in Reverse Alphabetical Order (Click to view full-size image)

Step 4: Creating the Sorting Interface and Remembering the Sort Expression and Direction

Turning on sorting support in the GridView converts each sortable field s header text into a LinkButton that, when clicked, sorts the data accordingly. Such a sorting interface makes sense for the GridView, where its data is neatly laid out in columns. For the DataList and Repeater controls, however, a different sorting interface is needed. A common sorting interface for a list of data (as opposed to a grid of data), is a drop-down list that provides the fields by which the data can be sorted. Let s implement such an interface for this tutorial.

Add a DropDownList Web control above the `SortableProducts` Repeater and set its `ID` property to `SortBy`. From the Properties window, click the ellipses in the `Items` property to bring up the ListItem Collection Editor. Add

`ListItem` s to sort the data by the `ProductName` , `CategoryName` , and `SupplierName` fields. Also add a `ListItem` to sort the products by their name in reverse alphabetical order.

The `ListItem` `Text` properties can be set to any value (such as Name), but the `Value` properties must be set to the name of the data field (such as ProductName). To sort the results in descending order, append the string DESC to the data field name, like ProductName DESC .

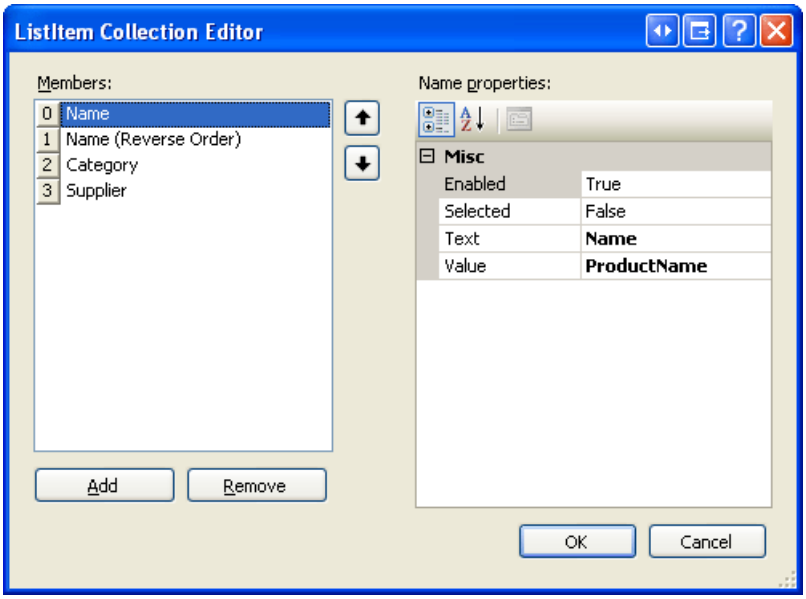


Figure 5: Add a `ListItem` for Each of the Sortable Data Fields

Finally, add a Button Web control to the right of the DropDownList. Set its `ID` to `RefreshRepeater` and its `Text` property to Refresh .

After creating the `ListItem` s and adding the Refresh button, the DropDownList and Button s declarative syntax should look similar to the following:

```
<asp:DropDownList ID="SortBy" runat="server">
    <asp:ListItem Value="ProductName">Name</asp:ListItem>
    <asp:ListItem Value="ProductName DESC">Name (Reverse Order)
        </asp:ListItem>
    <asp:ListItem Value="CategoryName">Category</asp:ListItem>
    <asp:ListItem Value="SupplierName">Supplier</asp:ListItem>
</asp:DropDownList>
<asp:Button runat="server" ID="RefreshRepeater" Text="Refresh" />
```

With the sorting DropDownList complete, we next need to update the ObjectDataSource s `Selecting` event handler so that it uses the selected `SortBy``ListItem` s `Value` property as opposed to a hard-coded sort expression.

```
protected void ProductsDataSource_Selecting
    (object sender, ObjectDataSourceSelectingEventArgs e)
{
    // Have the ObjectDataSource sort the results by the selected
    // sort expression
    e.Arguments.SortExpression = SortBy.SelectedValue;
}
```

At this point when first visiting the page the products will initially be sorted by the `ProductName` data field, as it s the `SortBy` `ListItem` selected by default (see Figure 6). Selecting a different sorting option such as Category and clicking Refresh will cause a postback and re-sort the data by the category name, as Figure 7 shows.

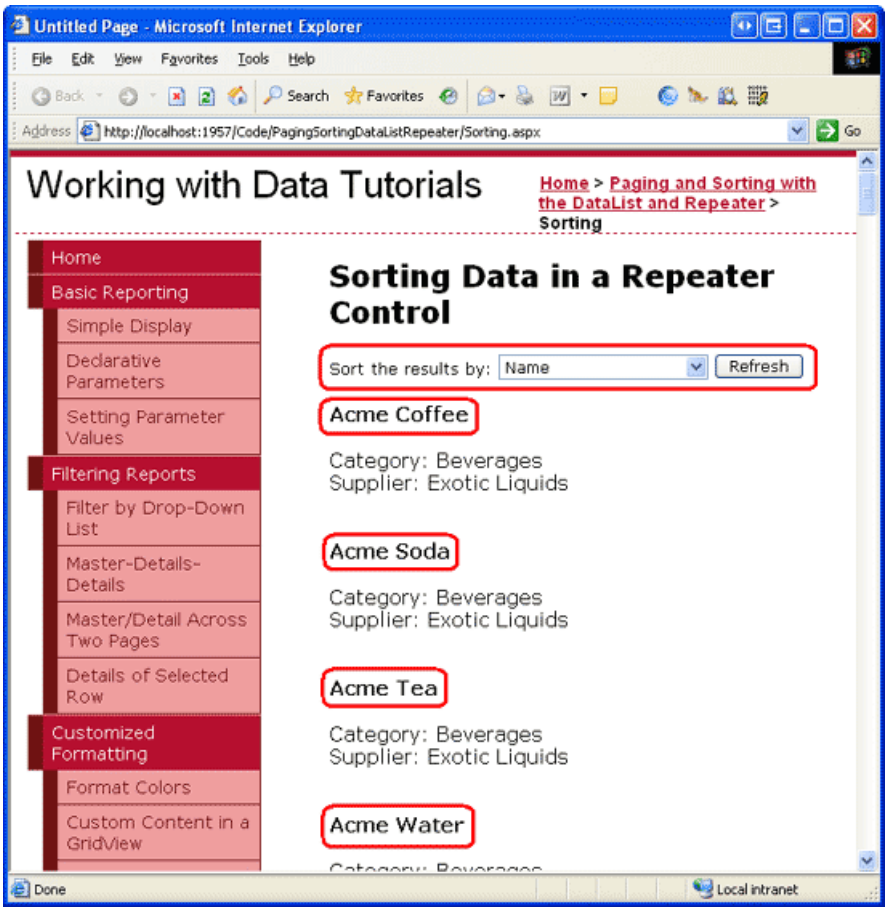


Figure 6: The Products are Initially Sorted by their Name (Click to view full-size image)

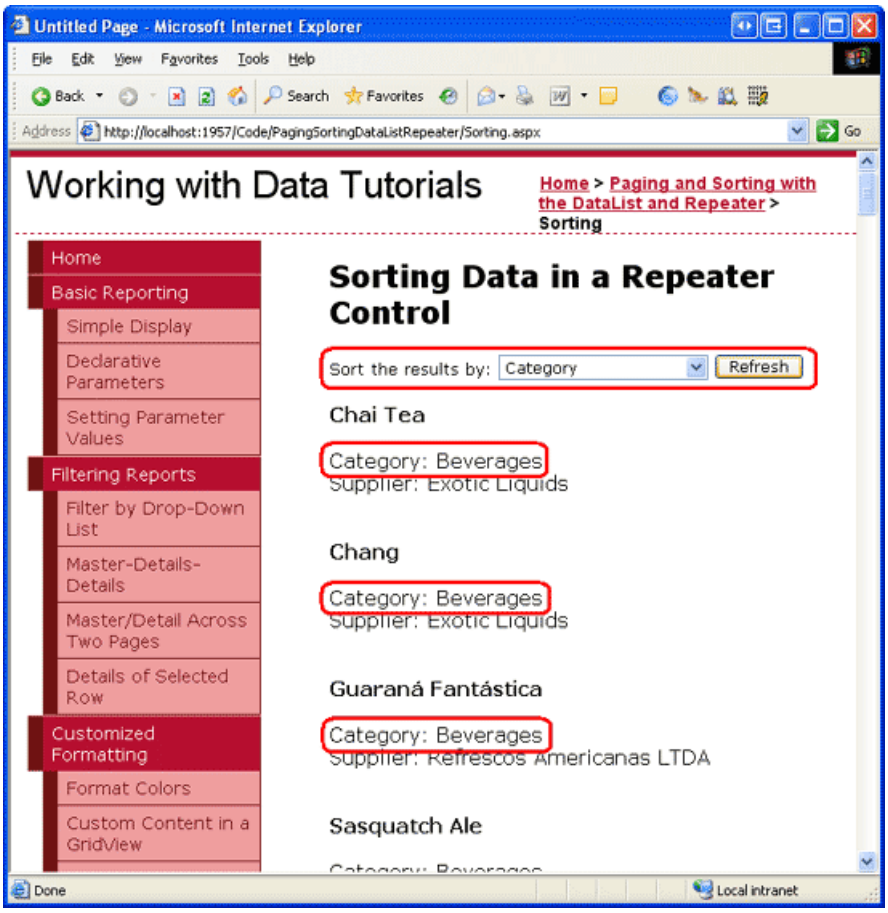


Figure 7: The Products are Now Sorted by Category (Click to view full-size image)

> **NOTE**
>
> Clicking the Refresh button causes the data to automatically be re-sorted because the Repeater s view state has been disabled, thereby causing the Repeater to rebind to its data source on every postback. If you ve left the Repeater s view state enabled, changing the sorting drop-down list won't have any affect on the sort order. To remedy this, create an event handler for the Refresh Button s `Click` event and rebind the Repeater to its data source (by calling the Repeater s `DataBind()` method).

Remembering the Sort Expression and Direction

When creating a sortable DataList or Repeater on a page where non-sort related postbacks may occur, it s imperative that the sort expression and direction be remembered across postbacks. For example, imagine that we updated the Repeater in this tutorial to include a Delete button with each product. When the user clicks the Delete button we d run some code to delete the selected product and then rebind the data to the Repeater. If the sort details are not persisted across postback, the data displayed on screen will revert to the original sort order.

For this tutorial, the DropDownList implicitly saves the sort expression and direction in its view state for us. If we were using a different sorting interface one with, say, LinkButtons that provided the various sorting options we d need to take care to remember the sort order across postbacks. This could be accomplished by storing the sorting parameters in the page s view state, by including the sort parameter in the querystring, or through some other state persistence technique.

Future examples in this tutorial explore how to persist the sorting details in the page s view state.

Step 5: Adding Sorting Support to a DataList that Uses Default Paging

In the preceding tutorial we examined how to implement default paging with a DataList. Let s extend this previous example to include the ability to sort the paged data. Start by opening the `SortingWithDefaultPaging.aspx` and `Paging.aspx` pages in the `PagingSortingDataListRepeater` folder. From the `Paging.aspx` page, click on the Source button to view the page s declarative markup. Copy the selected text (see Figure 8) and paste it into the declarative markup of `SortingWithDefaultPaging.aspx` between the `<asp:Content>` tags.

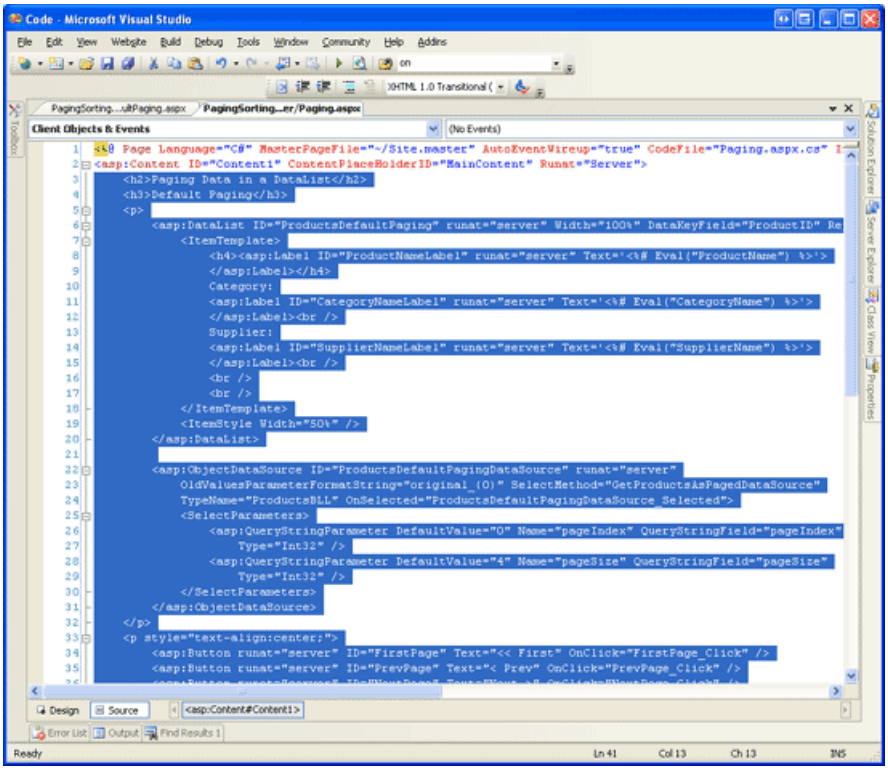


Figure 8: Replicate the Declarative Markup in the `<asp:Content>` Tags from `Paging.aspx` to

`SortingWithDefaultPaging.aspx` (Click to view full-size image)

After copying the declarative markup, copy the methods and properties in the `Paging.aspx` page s code-behind class to the code-behind class for `SortingWithDefaultPaging.aspx`. Next, take a moment to view the `SortingWithDefaultPaging.aspx` page in a browser. It should exhibit the same functionality and appearance as `Paging.aspx`.

Enhancing ProductsBLL to Include a Default Paging and Sorting Method

In the previous tutorial we created a `GetProductsAsPagedDataSource(pageIndex, pageSize)` method in the `ProductsBLL` class that returned a `PagedDataSource` object. This `PagedDataSource` object was populated with *all* of the products (via the BLL s `GetProducts()` method), but when bound to the DataList only those records corresponding to the specified *pageIndex* and *pageSize* input parameters were displayed.

Earlier in this tutorial we added sorting support by specifying the sort expression from the ObjectDataSource s `Selecting` event handler. This works well when the ObjectDataSource is returned an object that can be sorted, like the `ProductsDataTable` returned by the `GetProducts()` method. However, the `PagedDataSource` object returned by the `GetProductsAsPagedDataSource` method does not support sorting of its inner data source. Instead, we need to sort the results returned from the `GetProducts()` method *before* we put it in the `PagedDataSource`.

To accomplish this, create a new method in the `ProductsBLL` class, `GetProductsSortedAsPagedDataSource(sortExpression, pageIndex, pageSize)`. To sort the `ProductsDataTable` returned by the `GetProducts()` method, specify the `Sort` property of its default `DataTableView`:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Select, false)]
public PagedDataSource GetProductsSortedAsPagedDataSource
    (string sortExpression, int pageIndex, int pageSize)
{
    // Get ALL of the products
    Northwind.ProductsDataTable products = GetProducts();
    // Sort the products
    products.DefaultView.Sort = sortExpression;
    // Limit the results through a PagedDataSource
    PagedDataSource pagedData = new PagedDataSource();
    pagedData.DataSource = products.DefaultView;
    pagedData.AllowPaging = true;
    pagedData.CurrentPageIndex = pageIndex;
    pagedData.PageSize = pageSize;
    return pagedData;
}
```

The `GetProductsSortedAsPagedDataSource` method differs only slightly from the `GetProductsAsPagedDataSource` method created in the previous tutorial. In particular, `GetProductsSortedAsPagedDataSource` accepts an additional input parameter `sortExpression` and assigns this value to the `Sort` property of the `ProductDataTable` s `DefaultView`. A few lines of code later, the `PagedDataSource` object s DataSource is assigned the `ProductDataTable` s `DefaultView`.

Calling the GetProductsSortedAsPagedDataSource Method and Specifying the Value for the SortExpression Input Parameter

With the `GetProductsSortedAsPagedDataSource` method complete, the next step is to provide the value for this parameter. The ObjectDataSource in `SortingWithDefaultPaging.aspx` is currently configured to call the `GetProductsAsPagedDataSource` method and passes in the two input parameters through its two

`QueryStringParameters`, which are specified in the `SelectParameters` collection. These two `QueryStringParameters` indicate that the source for the `GetProductsAsPagedDataSource` method s *pageIndex* and *pageSize* parameters come from the querystring fields `pageIndex` and `pageSize`.

Update the ObjectDataSource s `SelectMethod` property so that it invokes the new `GetProductsSortedAsPagedDataSource` method. Then, add a new `QueryStringParameter` so that the *sortExpression* input parameter is accessed from the querystring field `sortExpression`. Set the `QueryStringParameter` s `DefaultValue` to ProductName .

After these changes, the ObjectDataSource s declarative markup should look like:

```
<asp:ObjectDataSource ID="ProductsDefaultPagingDataSource"
    OldValuesParameterFormatString="original_{0}" TypeName="ProductsBLL"
    SelectMethod="GetProductsSortedAsPagedDataSource"
    OnSelected="ProductsDefaultPagingDataSource_Selected" runat="server">
    <SelectParameters>
        <asp:QueryStringParameter DefaultValue="ProductName"
            Name="sortExpression" QueryStringField="sortExpression"
            Type="String" />
        <asp:QueryStringParameter DefaultValue="0" Name="pageIndex"
            QueryStringField="pageIndex" Type="Int32" />
        <asp:QueryStringParameter DefaultValue="4" Name="pageSize"
            QueryStringField="pageSize" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

At this point, the `SortingWithDefaultPaging.aspx` page will sort its results alphabetically by the product name (see Figure 9). This is because, by default, a value of ProductName is passed in as the `GetProductsSortedAsPagedDataSource` method s *sortExpression* parameter.

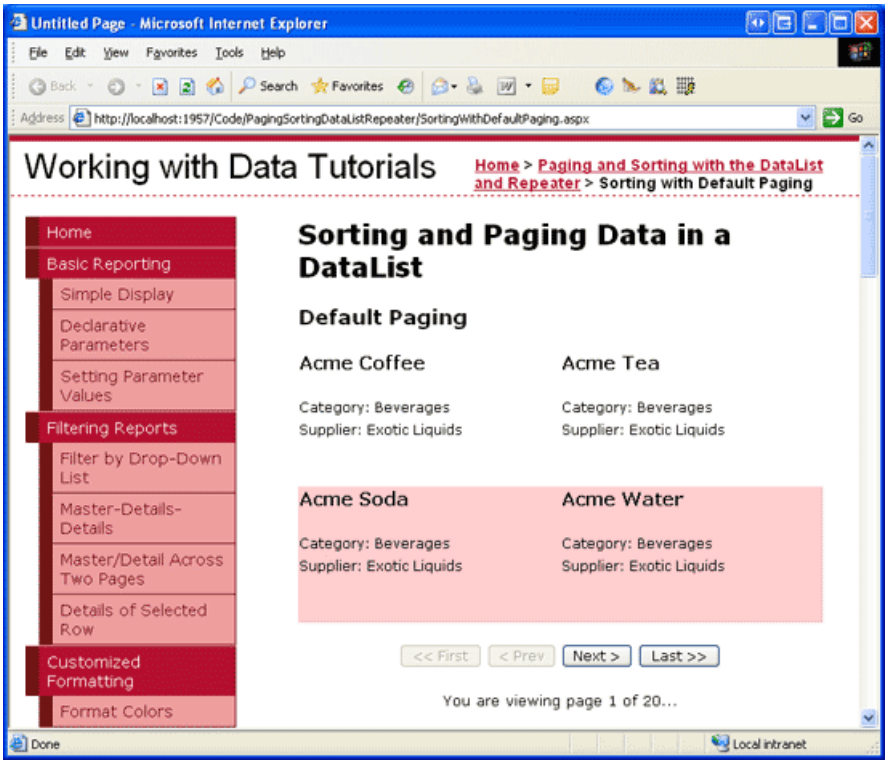


Figure 9: By Default, the Results are Sorted by `ProductName` (Click to view full-size image)

If you manually add a `sortExpression` querystring field such as `SortingWithDefaultPaging.aspx?sortExpression=CategoryName` the results will be sorted by the specified `sortExpression`. However, this `sortExpression` parameter is not included in the querystring when moving to a different page of data. In fact, clicking on the Next or Last page buttons takes us back to `Paging.aspx`!

Furthermore, there s currently no sorting interface. The only way a user can change the sort order of the paged data is by manipulating the querystring directly.

Creating the Sorting Interface

We first need to update the `RedirectUser` method to send the user to `SortingWithDefaultPaging.aspx` (instead of `Paging.aspx`) and to include the `sortExpression` value in the querystring. We should also add a read-only, page-level named `SortExpression` property. This property, similar to the `PageIndex` and `PageSize` properties created in the previous tutorial, returns the value of the `sortExpression` querystring field if it exists, and the default value (ProductName) otherwise.

Currently the `RedirectUser` method accepts only a single input parameter the index of the page to display. However, there may be times when we want to redirect the user to a particular page of data using a sort expression other than what s specified in the querystring. In a moment we'll create the sorting interface for this page, which will include a series of Button Web controls for sorting the data by a specified column. When one of those Buttons is clicked, we want to redirect the user passing in the appropriate sort expression value. To provide this functionality, create two versions of the `RedirectUser` method. The first one should accept just the page index to display, while the second one accepts the page index and sort expression.

```
private string SortExpression
{
    get
    {
        if (!string.IsNullOrEmpty(Request.QueryString["sortExpression"]))
            return Request.QueryString["sortExpression"];
        else
            return "ProductName";
    }
}
private void RedirectUser(int sendUserToPageIndex)
{
    // Use the SortExpression property to get the sort expression
    // from the querystring
    RedirectUser(sendUserToPageIndex, SortExpression);
}
private void RedirectUser(int sendUserToPageIndex, string sendUserSortingBy)
{
   // Send the user to the requested page with the requested sort expression
   Response.Redirect(string.Format(
      "SortingWithDefaultPaging.aspx?pageIndex={0}&pageSize={1}&sortExpression={2}",
      sendUserToPageIndex, PageSize, sendUserSortingBy));
}
```

In the first example in this tutorial, we created a sorting interface using a DropDownList. For this example, let s use three Button Web controls positioned above the DataList one for sorting by `ProductName`, one for `CategoryName`, and one for `SupplierName`. Add the three Button Web controls, setting their `ID` and `Text` properties appropriately:

```
<p>
    <asp:Button runat="server" id="SortByProductName"
        Text="Sort by Product Name" />
    <asp:Button runat="server" id="SortByCategoryName"
        Text="Sort by Category" />
    <asp:Button runat="server" id="SortBySupplierName"
        Text="Sort by Supplier" />
</p>
```

Next, create a `Click` event handler for each. The event handlers should call the `RedirectUser` method, returning

the user to the first page using the appropriate sort expression.

```
protected void SortByProductName_Click(object sender, EventArgs e)
{
    // Sort by ProductName
    RedirectUser(0, "ProductName");
}
protected void SortByCategoryName_Click(object sender, EventArgs e)
{
    // Sort by CategoryName
    RedirectUser(0, "CategoryName");
}
protected void SortBySupplierName_Click(object sender, EventArgs e)
{
    // Sort by SupplierName
    RedirectUser(0, "SupplierName");
}
```

When first visiting the page, the data is sorted by the product name alphabetically (refer back to Figure 9). Click the Next button to advance to the second page of data and then click the Sort by Category button. This returns us to the first page of data, sorted by category name (see Figure 10). Likewise, clicking the Sort by Supplier button sorts the data by supplier starting from the first page of data. The sort choice is remembered as the data is paged through. Figure 11 shows the page after sorting by category and then advancing to the thirteenth page of data.

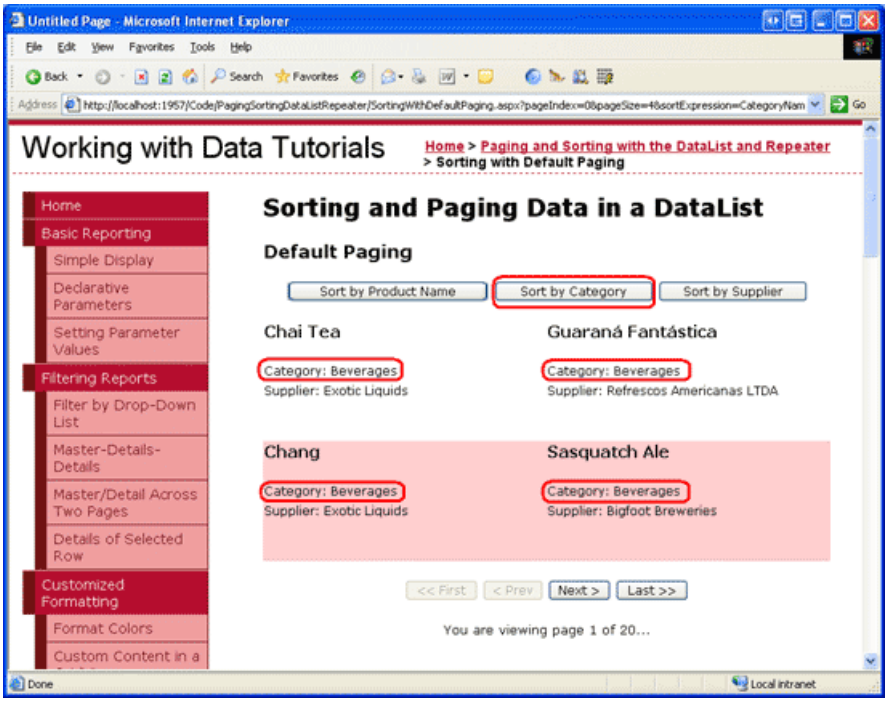


Figure 10: The Products are Sorted by Category (Click to view full-size image)

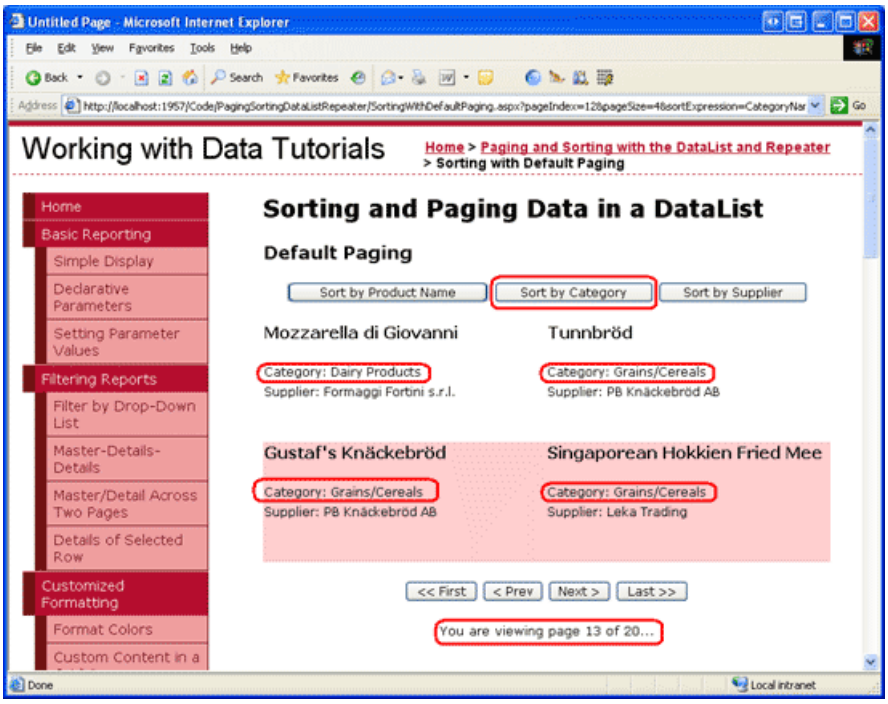


Figure 11: The Sort Expression is Remembered When Paging Through the Data (Click to view full-size image)

Step 6: Custom Paging Through Records in a Repeater

The DataList example examined in step 5 pages through its data using the inefficient default paging technique. When paging through sufficiently large amounts of data, it is imperative that custom paging be used. Back in the Efficiently Paging Through Large Amounts of Data and Sorting Custom Paged Data tutorials, we examined the differences between default and custom paging and created methods in the BLL for utilizing custom paging and sorting custom paged data. In particular, in these two previous tutorials we added the following three methods to the `ProductsBLL` class:

- `GetProductsPaged(startRowIndex, maximumRows)` returns a particular subset of records starting at *startRowIndex* and not exceeding *maximumRows*.
- `GetProductsPagedAndSorted(sortExpression, startRowIndex, maximumRows)` returns a particular subset of records sorted by the specified *sortExpression* input parameter.
- `TotalNumberOfProducts()` provides the total number of records in the `Products` database table.

These methods can be used to efficiently page and sort through data using a DataList or Repeater control. To illustrate this, let s start by creating a Repeater control with custom paging support; we'll then add sorting capabilities.

Open the `SortingWithCustomPaging.aspx` page in the `PagingSortingDataListRepeater` folder and add a Repeater to the page, setting its `ID` property to `Products`. From the Repeater s smart tag, create a new ObjectDataSource named `ProductsDataSource`. Configure it to select its data from the `ProductsBLL` class s `GetProductsPaged` method.

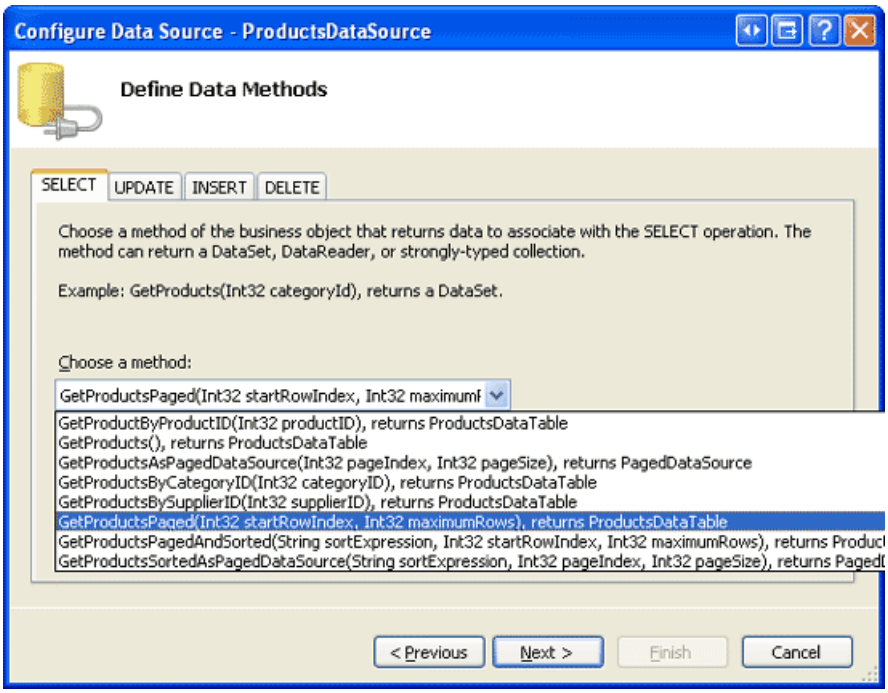


Figure 12: Configure the ObjectDataSource to Use the `ProductsBLL` Class s `GetProductsPaged` Method (Click to view full-size image)

Set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) and then click the Next button. The Configure Data Source wizard now prompts for the sources of the `GetProductsPaged` method s *startRowIndex* and *maximumRows* input parameters. In actuality, these input parameters are ignored. Instead, the *startRowIndex* and *maximumRows* values will be passed in through the `Arguments` property in the ObjectDataSource s `Selecting` event handler, just like how we specified the *sortExpression* in this tutorial s first demo. Therefore, leave the parameter source drop-down lists in the wizard set at None .

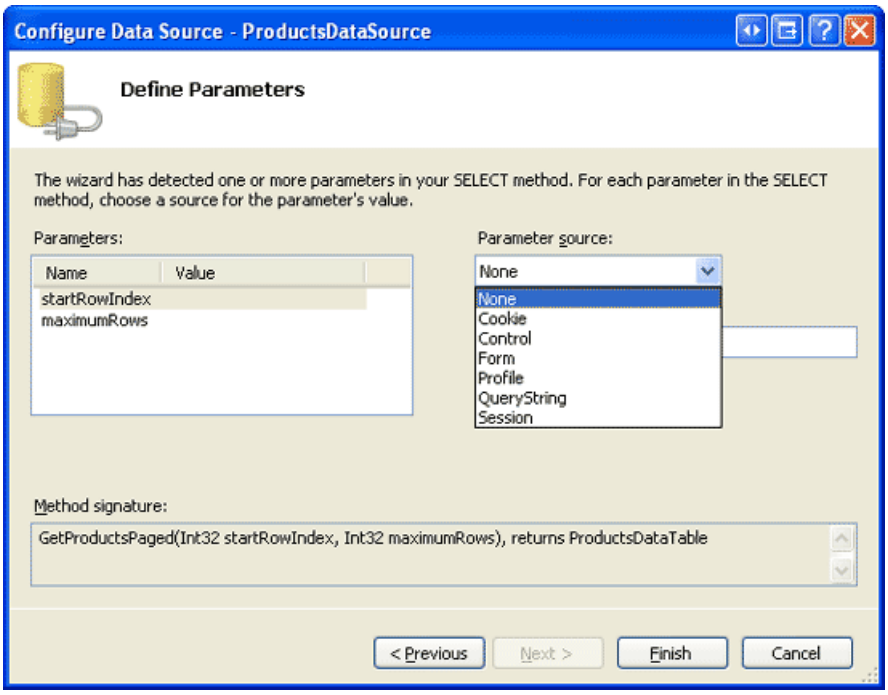


Figure 13: Leave the Parameter Sources Set to None (Click to view full-size image)

NOTE

Do *not* set the ObjectDataSource s `EnablePaging` property to `true`. This will cause the ObjectDataSource to automatically include its own *startRowIndex* and *maximumRows* parameters to the `SelectMethod` s existing parameter list. The `EnablePaging` property is useful when binding custom paged data to a GridView, DetailsView, or FormView control because these controls expect certain behavior from the ObjectDataSource that s only available when `EnablePaging` property is `true`. Since we have to manually add the paging support for the DataList and Repeater, leave this property set to `false` (the default), as we'll bake in the needed functionality directly within our ASP.NET page.

Finally, define the Repeater s `ItemTemplate` so that the product s name, category, and supplier are shown. After these changes, the Repeater and ObjectDataSource s declarative syntax should look similar to the following:

```
<asp:Repeater ID="Products" runat="server" DataSourceID="ProductsDataSource"
    EnableViewState="False">
    <ItemTemplate>
        <h4><asp:Label ID="ProductNameLabel" runat="server"
            Text='<%# Eval("ProductName") %>'></asp:Label></h4>
        Category:
        <asp:Label ID="CategoryNameLabel" runat="server"
            Text='<%# Eval("CategoryName") %>'></asp:Label><br />
        Supplier:
        <asp:Label ID="SupplierNameLabel" runat="server"
            Text='<%# Eval("SupplierName") %>'></asp:Label><br />
        <br />
        <br />
    </ItemTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProductsPaged" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:Parameter Name="startRowIndex" Type="Int32" />
        <asp:Parameter Name="maximumRows" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

Take a moment to visit the page through a browser and note that no records are returned. This is because we ve yet to specify the *startRowIndex* and *maximumRows* parameter values; therefore, values of 0 are being passed in for both. To specify these values, create an event handler for the ObjectDataSource s `Selecting` event and set these parameters values programmatically to hard-coded values of 0 and 5, respectively:

```
protected void ProductsDataSource_Selecting
    (object sender, ObjectDataSourceSelectingEventArgs e)
{
    e.InputParameters["startRowIndex"] = 0;
    e.InputParameters["maximumRows"] = 5;
}
```

With this change, the page, when viewed through a browser, shows the first five products.

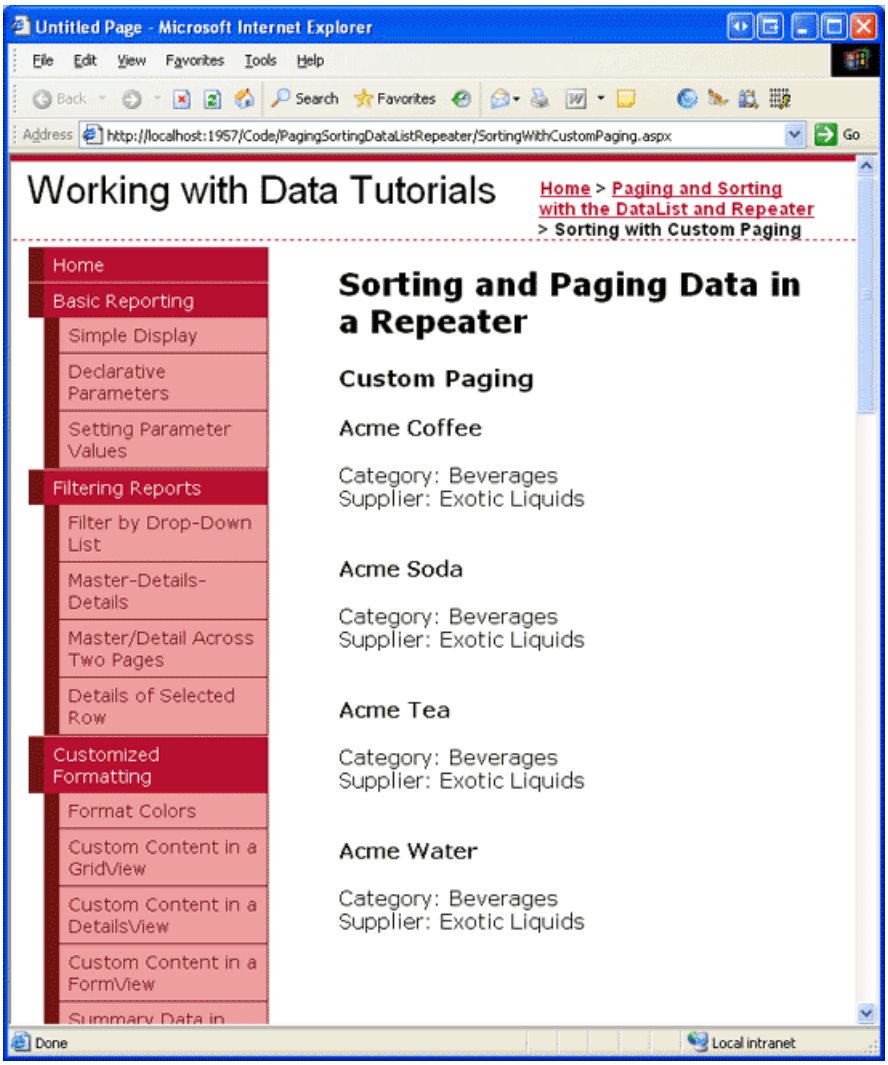


Figure 14: The First Five Records are Displayed (Click to view full-size image)

> **NOTE**
>
> The products listed in Figure 14 happen to be sorted by product name because the `GetProductsPaged` stored procedure that performs the efficient custom paging query orders the results by `ProductName`.

In order to allow the user to step through the pages, we need to keep track of the start row index and maximum rows and remember these values across postbacks. In the default paging example we used querystring fields to persist these values; for this demo, let s persist this information in the page s view state. Create the following two properties:

```
private int StartRowIndex
{
    get
    {
        object o = ViewState["StartRowIndex"];
        if (o == null)
            return 0;
        else
            return (int)o;
    }
    set
    {
        ViewState["StartRowIndex"] = value;
    }
}
private int MaximumRows
{
    get
    {
        object o = ViewState["MaximumRows"];
        if (o == null)
            return 5;
        else
            return (int)o;
    }
    set
    {
        ViewState["MaximumRows"] = value;
    }
}
```

Next, update the code in the Selecting event handler so that it uses the `StartRowIndex` and `MaximumRows` properties instead of the hard-coded values of 0 and 5:

```
e.InputParameters["startRowIndex"] = StartRowIndex;
e.InputParameters["maximumRows"] = MaximumRows;
```

At this point our page still shows just the first five records. However, with these properties in place, we re ready to create our paging interface.

Adding the Paging Interface

Let s use the same First, Previous, Next, Last paging interface used in the default paging example, including the Label Web control that displays what page of data is being viewed and how many total pages exist. Add the four Button Web controls and Label below the Repeater.

```
<p>
    <asp:Button runat="server" ID="FirstPage" Text="<< First" />
    <asp:Button runat="server" ID="PrevPage" Text="< Prev" />
    <asp:Button runat="server" ID="NextPage" Text="Next >" />
    <asp:Button runat="server" ID="LastPage" Text="Last >>" />
</p>
<p>
    <asp:Label runat="server" ID="CurrentPageNumber"></asp:Label>
</p>
```

Next, create `Click` event handlers for the four Buttons. When one of these Buttons is clicked, we need to update the `StartRowIndex` and rebind the data to the Repeater. The code for the First, Previous, and Next buttons is simple enough, but for the Last button how do we determine the start row index for the last page of data? To compute

this index as well as being able to determine whether the Next and Last buttons should be enabled we need to know how many records in total are being paged through. We can determine this by calling the `ProductsBLL` class s `TotalNumberOfProducts()` method. Let s create a read-only, page-level property named `TotalRowCount` that returns the results of the `TotalNumberOfProducts()` method:

```
private int TotalRowCount
{
    get
    {
        // Return the value from the TotalNumberOfProducts() method
        ProductsBLL productsAPI = new ProductsBLL();
        return productsAPI.TotalNumberOfProducts();
    }
}
```

With this property we can now determine the last page s start row index. Specifically, it s the integer result of the `TotalRowCount` minus 1 divided by `MaximumRows`, multiplied by `MaximumRows`. We can now write the `Click` event handlers for the four paging interface buttons:

```
protected void FirstPage_Click(object sender, EventArgs e)
{
    // Return to StartRowIndex of 0 and rebind data
    StartRowIndex = 0;
    Products.DataBind();
}
protected void PrevPage_Click(object sender, EventArgs e)
{
    // Subtract MaximumRows from StartRowIndex and rebind data
    StartRowIndex -= MaximumRows;
    Products.DataBind();
}
protected void NextPage_Click(object sender, EventArgs e)
{
    // Add MaximumRows to StartRowIndex and rebind data
    StartRowIndex += MaximumRows;
    Products.DataBind();
}
protected void LastPage_Click(object sender, EventArgs e)
{
    // Set StartRowIndex = to last page's starting row index and rebind data
    StartRowIndex = ((TotalRowCount - 1) / MaximumRows) * MaximumRows;
    Products.DataBind();
}
```

Finally, we need to disable the First and Previous buttons in the paging interface when viewing the first page of data and the Next and Last buttons when viewing the last page. To accomplish this, add the following code to the ObjectDataSource s `Selecting` event handler:

```
// Disable the paging interface buttons, if needed
FirstPage.Enabled = StartRowIndex != 0;
PrevPage.Enabled = StartRowIndex != 0;
int LastPageStartRowIndex = ((TotalRowCount - 1) / MaximumRows) * MaximumRows;
NextPage.Enabled = StartRowIndex < LastPageStartRowIndex;
LastPage.Enabled = StartRowIndex < LastPageStartRowIndex;
```

After adding these `Click` event handlers and the code to enable or disable the paging interface elements based on the current start row index, test the page in a browser. As Figure 15 illustrates, when first visiting the page the First and Previous buttons will are disabled. Clicking Next shows the second page of data, while clicking Last displays the final page (see Figures 16 and 17). When viewing the last page of data both the Next and Last buttons

are disabled.

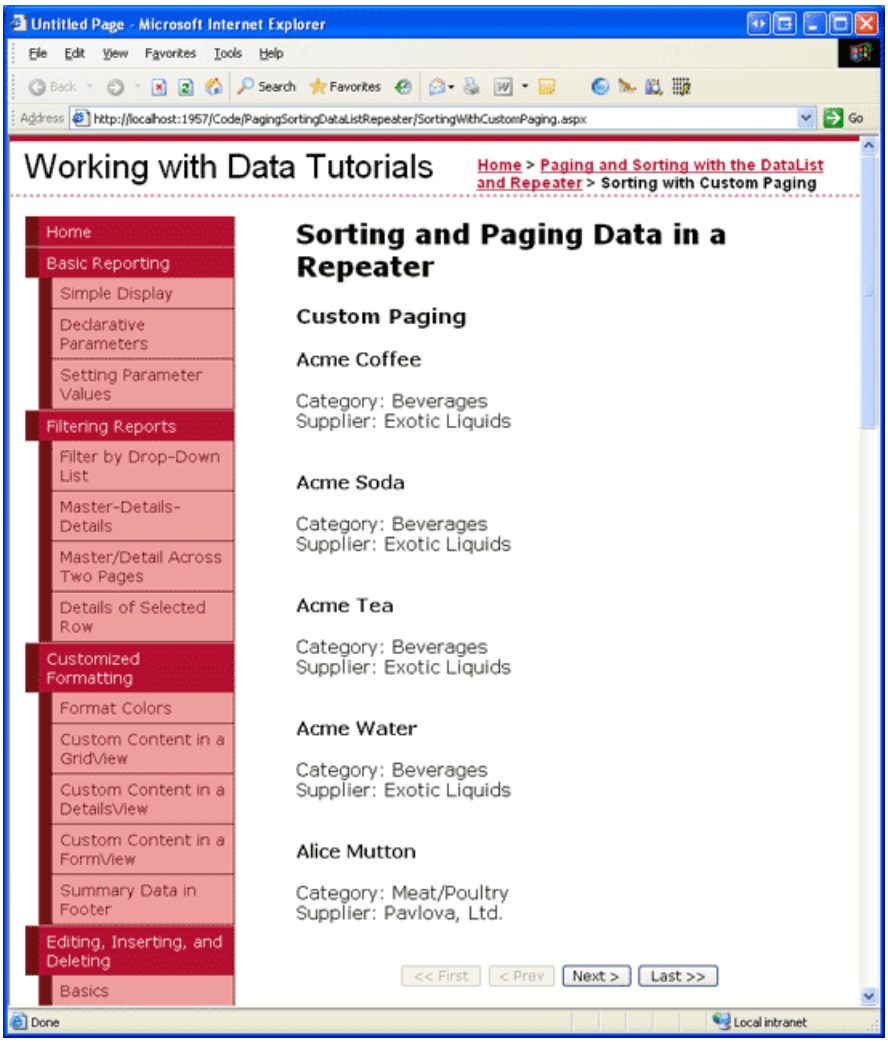


Figure 15: The Previous and Last Buttons are Disabled When Viewing the First Page of Products (Click to view full-size image)

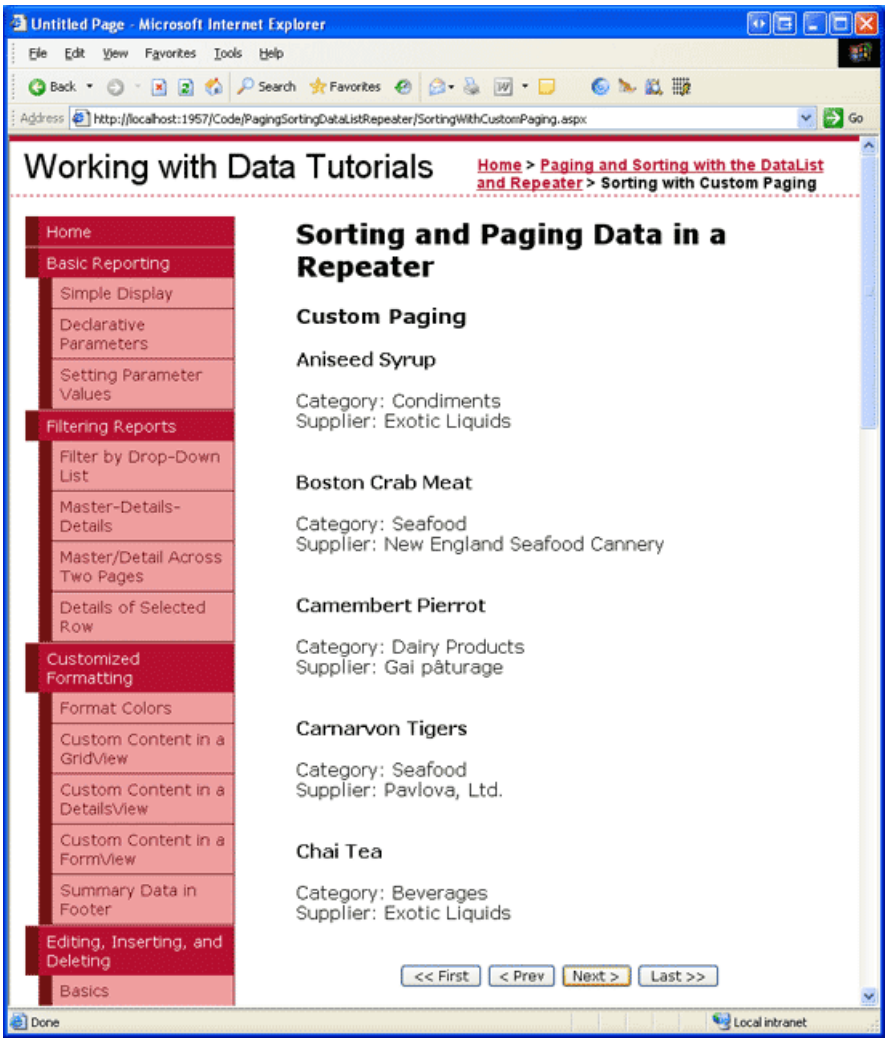


Figure 16: The Second Page of Products are Displayed (Click to view full-size image)

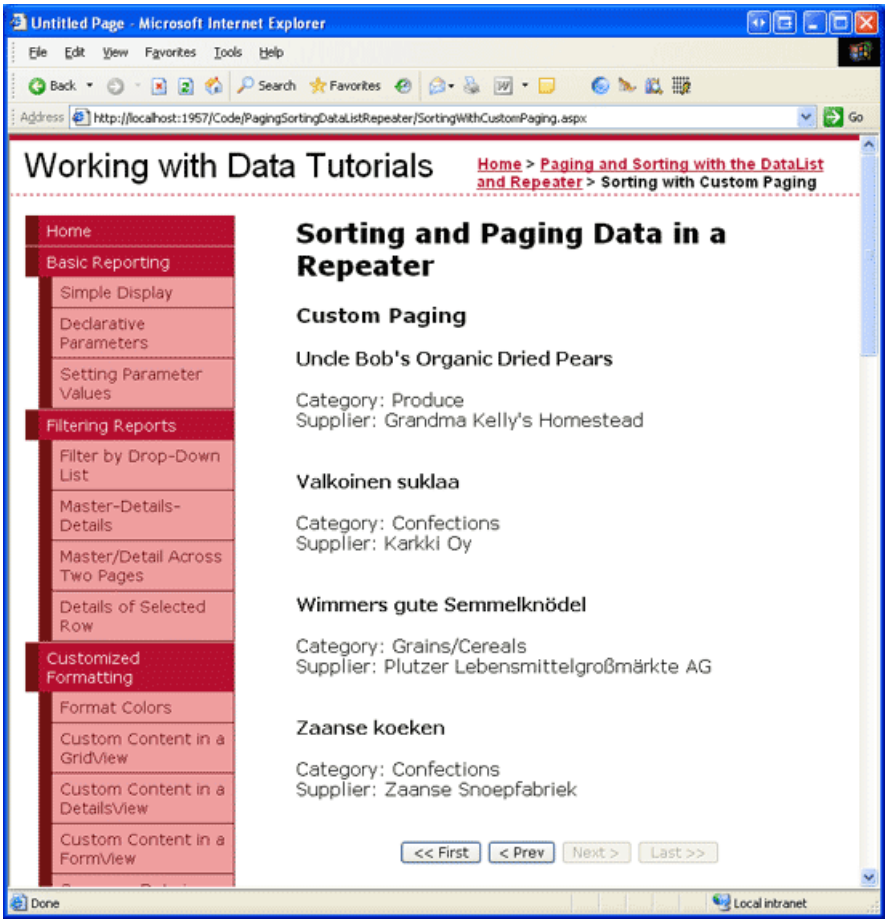


Figure 17: Clicking Last Displays the Final Page of Data (Click to view full-size image)

Step 7: Including Sorting Support with the Custom Paged Repeater

Now that custom paging has been implemented, we re ready to include sorting support. The `ProductsBLL` class s `GetProductsPagedAndSorted` method has the same *startRowIndex* and *maximumRows* input parameters as `GetProductsPaged`, but permits an additional *sortExpression* input parameter. To use the `GetProductsPagedAndSorted` method from `SortingWithCustomPaging.aspx`, we need to perform the following steps:

1. Change the ObjectDataSource s `SelectMethod` property from `GetProductsPaged` to `GetProductsPagedAndSorted`.
2. Add a *sortExpression* `Parameter` object to the ObjectDataSource s `SelectParameters` collection.
3. Create a private, page-level `SortExpression` property that persists its value across postbacks through the page s view state.
4. Update the ObjectDataSource s `Selecting` event handler to assign the ObjectDataSource s *sortExpression* parameter the value of the page-level `SortExpression` property.
5. Create the sorting interface.

Start by updating the ObjectDataSource s `SelectMethod` property and adding a *sortExpression* `Parameter`. Make sure that the *sortExpression* `Parameter` s `Type` property is set to `String`. After completing these first two tasks, the ObjectDataSource s declarative markup should look like the following:

```
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}" TypeName="ProductsBLL"
    SelectMethod="GetProductsPagedAndSorted"
    OnSelecting="ProductsDataSource_Selecting">
    <SelectParameters>
        <asp:Parameter Name="sortExpression" Type="String" />
        <asp:Parameter Name="startRowIndex" Type="Int32" />
        <asp:Parameter Name="maximumRows" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

Next, we need a page-level `SortExpression` property whose value is serialized to view state. If no sort expression value has been set, use ProductName as the default:

```
private string SortExpression
{
    get
    {
        object o = ViewState["SortExpression"];
        if (o == null)
            return "ProductName";
        else
            return o.ToString();
    }
    set
    {
        ViewState["SortExpression"] = value;
    }
}
```

Before the ObjectDataSource invokes the `GetProductsPagedAndSorted` method we need to set the *sortExpression* `Parameter` to the value of the `SortExpression` property. In the `Selecting` event handler, add the following line of code:

```
e.InputParameters["sortExpression"] = SortExpression;
```

All that remains is to implement the sorting interface. As we did in the last example, let s have the sorting interface implemented using three Button Web controls that allow the user to sort the results by product name, category, or supplier.

```
<asp:Button runat="server" id="SortByProductName"
    Text="Sort by Product Name" />
<asp:Button runat="server" id="SortByCategoryName"
    Text="Sort by Category" />
<asp:Button runat="server" id="SortBySupplierName"
    Text="Sort by Supplier" />
```

Create `Click` event handlers for these three Button controls. In the event handler, reset the `StartRowIndex` to 0, set the `SortExpression` to the appropriate value, and rebind the data to the Repeater:

```
protected void SortByProductName_Click(object sender, EventArgs e)
{
    StartRowIndex = 0;
    SortExpression = "ProductName";
    Products.DataBind();
}
protected void SortByCategoryName_Click(object sender, EventArgs e)
{
    StartRowIndex = 0;
    SortExpression = "CategoryName";
    Products.DataBind();
}
protected void SortBySupplierName_Click(object sender, EventArgs e)
{
    StartRowIndex = 0;
    SortExpression = "CompanyName";
    Products.DataBind();
}
```

That s all there is to it! While there were a number of steps to get custom paging and sorting implemented, the steps were very similar to those needed for default paging. Figure 18 shows the products when viewing the last page of data when sorted by category.

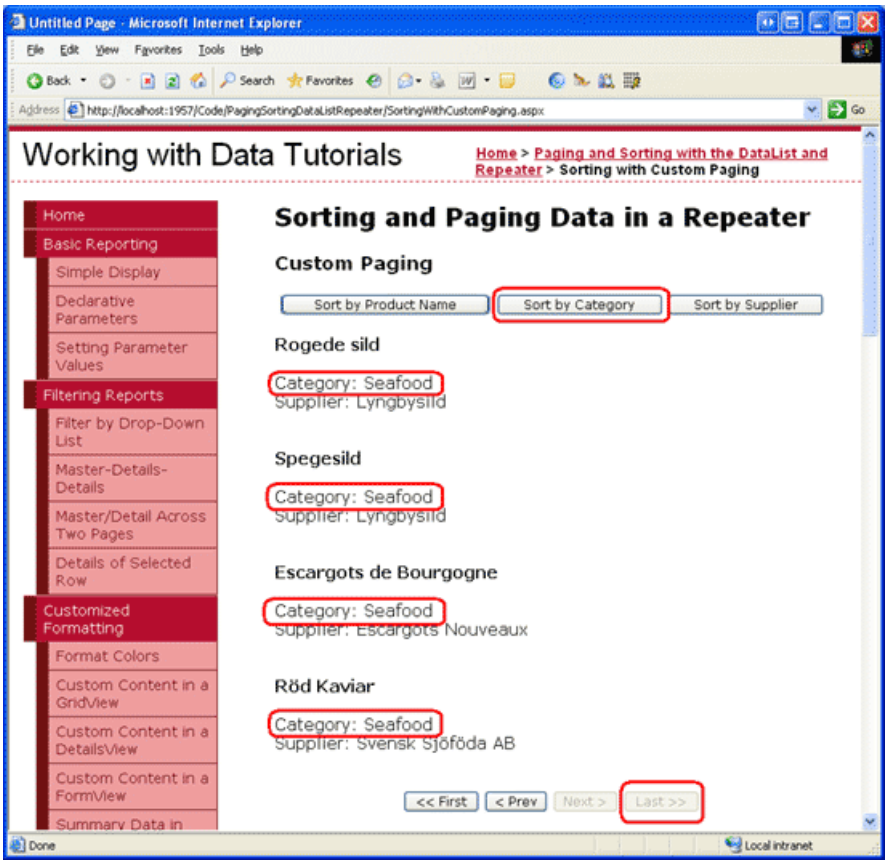


Figure 18: The Last Page of Data, Sorted by Category, is Displayed (Click to view full-size image)

> **NOTE**
>
> In previous examples, when sorting by the supplier SupplierName was used as the sort expression. However, for the custom paging implementation, we need to use CompanyName. This is because the stored procedure responsible for implementing custom paging `GetProductsPagedAndSorted` passes the sort expression into the `ROW_NUMBER()` keyword, The `ROW_NUMBER()` keyword requires the actual column name rather than an alias. Therefore, we must use `CompanyName` (the name of the column in the `Suppliers` table) rather than the alias used in the `SELECT` query (`SupplierName`) for the sort expression.

Summary

Neither the DataList nor Repeater offer built-in sorting support, but with a bit of code and a custom sorting interface, such functionality can be added. When implementing sorting, but not paging, the sort expression can be specified through the `DataSourceSelectArguments` object passed into the ObjectDataSource s `Select` method. This `DataSourceSelectArguments` object s `SortExpression` property can be assigned in the ObjectDataSource s `Selecting` event handler.

To add sorting capabilities to a DataList or Repeater that already provides paging support, the easiest approach is to customize the Business Logic Layer to include a method that accepts a sort expression. This information can then be passed in through a parameter in the ObjectDataSource s `SelectParameters`.

This tutorial completes our examination of paging and sorting with the DataList and Repeater controls. Our next and final tutorial will examine how to add Button Web controls to the DataList and Repeater s templates in order to provide some custom, user-initiated functionality on a per-item basis.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was David Suru. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Paging Report Data in a DataList or Repeater Control (VB)

2/20/2020 • 19 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> While neither the DataList nor Repeater offer automatic paging or sorting support, this tutorial shows how to add paging support to the DataList or Repeater, which allows for much more flexible paging and data display interfaces.

Introduction

Paging and sorting are two very common features when displaying data in an online application. For example, when searching for ASP.NET books at an online bookstore, there may be hundreds of such books, but the report listing the search results lists only ten matches per page. Moreover, the results can be sorted by title, price, page count, author name, and so on. As we discussed in the Paging and Sorting Report Data tutorial, the GridView, DetailsView, and FormView controls all provide built-in paging support that can be enabled at the tick of a checkbox. The GridView also includes sorting support.

Unfortunately, neither the DataList nor Repeater offer automatic paging or sorting support. In this tutorial we'll examine how to add paging support to the DataList or Repeater. We must manually create the paging interface, display the appropriate page of records, and remember the page being visited across postbacks. While this does take more time and code than with the GridView, DetailsView, or FormView, the DataList and Repeater allow for much more flexible paging and data display interfaces.

> **NOTE**
>
> This tutorial focuses exclusively on paging. In the next tutorial we'll turn our attention to adding sorting capabilities.

Step 1: Adding the Paging and Sorting Tutorial Web Pages

Before we start this tutorial, let s first take a moment to add the ASP.NET pages we'll need for this tutorial and the next one. Start by creating a new folder in the project named `PagingSortingDataListRepeater`. Next, add the following five ASP.NET pages to this folder, having all of them configured to use the master page `Site.master`:

- `Default.aspx`
- `Paging.aspx`
- `Sorting.aspx`
- `SortingWithDefaultPaging.aspx`
- `SortingWithCustomPaging.aspx`

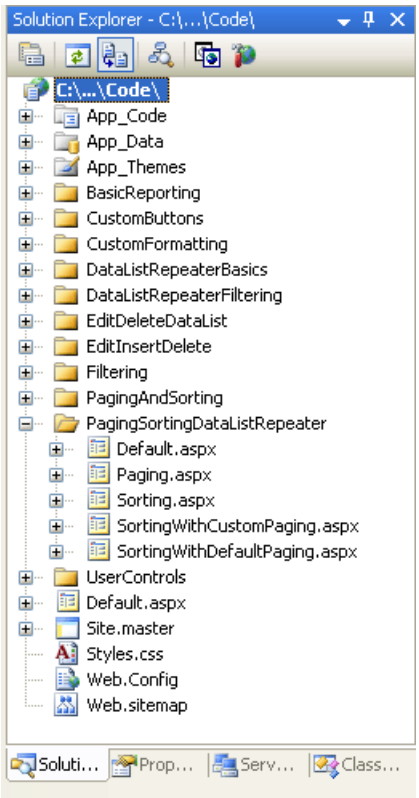


Figure 1: Create a `PagingSortingDataListRepeater` Folder and Add the Tutorial ASP.NET Pages

Next, open the `Default.aspx` page and drag the `SectionLevelTutorialListing.ascx` User Control from the `UserControls` folder onto the Design surface. This User Control, which we created in the Master Pages and Site Navigation tutorial, enumerates the site map and displays those tutorials in the current section in a bulleted list.

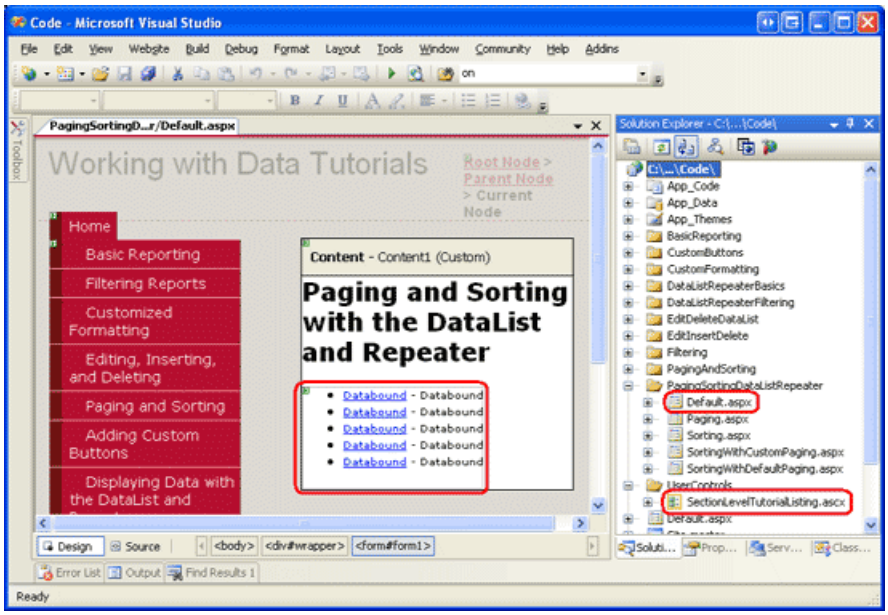


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

In order to have the bulleted list display the paging and sorting tutorials we'll be creating, we need to add them to the site map. Open the `Web.sitemap` file and add the following markup after the Editing and Deleting with the DataList site map node markup:

```
<siteMapNode
    url="~/PagingSortingDataListRepeater/Default.aspx"
    title="Paging and Sorting with the DataList and Repeater"
    description="Paging and Sorting the Data in the DataList and Repeater Controls">
    <siteMapNode
        url="~/PagingSortingDataListRepeater/Paging.aspx"
        title="Paging"
        description="Learn how to page through the data shown
                     in the DataList and Repeater controls." />
    <siteMapNode
        url="~/PagingSortingDataListRepeater/Sorting.aspx"
        title="Sorting"
        description="Sort the data displayed in a DataList or
                     Repeater control." />
    <siteMapNode
        url="~/PagingSortingDataListRepeater/SortingWithDefaultPaging.aspx"
        title="Sorting with Default Paging"
        description="Create a DataList or Repeater control that is paged using
                     default paging and can be sorted." />
    <siteMapNode
        url="~/PagingSortingDataListRepeater/SortingWithCustomPaging.aspx"
        title="Sorting with Custom Paging"
        description="Learn how to sort the data displayed in a DataList or
                     Repeater control that uses custom paging." />
</siteMapNode>
```

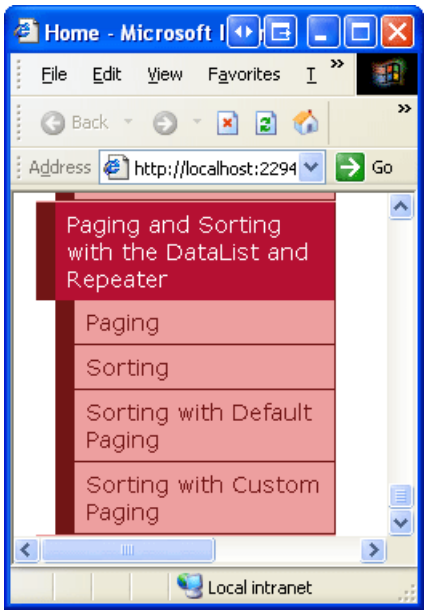


Figure 3: Update the Site Map to Include the New ASP.NET Pages

A Review of Paging

In previous tutorials we saw how to page through the data in the GridView, DetailsView, and FormView controls. These three controls offer a simple form of paging called *default paging* that can be implemented by simply checking the Enable Paging option in the control s smart tag. With default paging, each time a page of data is requested either on the first page visit or when the user navigates to a different page of data the GridView, DetailsView, or FormView control re-requests *all* of the data from the ObjectDataSource. It then snips out the particular set of records to display given the requested page index and the number of records to display per page. We discussed default paging in detail in the Paging and Sorting Report Data tutorial.

Since default paging re-requests all records for each page, it is not practical when paging through sufficiently large amounts of data. For example, imagine paging through 50,000 records with a page size of 10. Each time the user moves to a new page, all 50,000 records must be retrieved from the database, even though only ten of them are displayed.

Custom paging solves the performance concerns of default paging by grabbing only the precise subset of records to display on the requested page. When implementing custom paging, we must write the SQL query that will efficiently return just the correct set of records. We saw how to create such a query using SQL Server 2005 s new `ROW_NUMBER()` keyword back in the Efficiently Paging Through Large Amounts of Data tutorial.

To implement default paging in the DataList or Repeater controls, we can use the `PagedDataSource` class as a wrapper around the `ProductsDataTable` whose contents are being paged. The `PagedDataSource` class has a `DataSource` property that can be assigned to any enumerable object and `PageSize` and `CurrentPageIndex` properties that indicate how many records to show per page and the current page index. Once these properties have been set, the `PagedDataSource` can be used as the data source of any data Web control. The `PagedDataSource` , when enumerated, will only return the appropriate subset of records of its inner `DataSource` based on the `PageSize` and `CurrentPageIndex` properties. Figure 4 depicts the functionality of the `PagedDataSource` class.

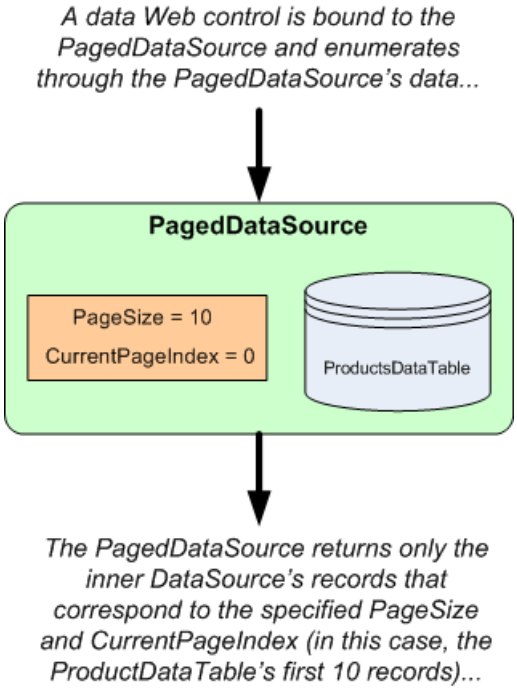


Figure 4: The `PagedDataSource` Wraps an Enumerable Object with a Pageable Interface

The `PagedDataSource` object can be created and configured directly from the Business Logic Layer and bound to a DataList or Repeater through an ObjectDataSource, or can be created and configured directly in the ASP.NET page s code-behind class. If the latter approach is used, we must forgo using the ObjectDataSource and instead bind the paged data to the DataList or Repeater programmatically.

The `PagedDataSource` object also has properties to support custom paging. However, we can bypass using a `PagedDataSource` for custom paging because we already have BLL methods in the `ProductsBLL` class designed for custom paging that return the precise records to display.

In this tutorial we'll look at implementing default paging in a DataList by adding a new method to the `ProductsBLL` class that returns an appropriately configured `PagedDataSource` object. In the next tutorial, we'll see how to use custom paging.

Step 2: Adding a Default Paging Method in the Business Logic Layer

The `ProductsBLL` class currently has a method for returning all product information `GetProducts()` and one for returning a particular subset of products at a starting index `GetProductsPaged(startRowIndex, maximumRows)`. With default paging, the GridView, DetailsView, and FormView controls all use the `GetProducts()` method to retrieve all products, but then use a `PagedDataSource` internally to display only the correct subset of records. To replicate this functionality with the DataList and Repeater controls, we can create a new method in the BLL that mimics this behavior.

Add a method to the `ProductsBLL` class named `GetProductsAsPagedDataSource` that takes in two integer input parameters:

- `pageIndex` the index of the page to display, indexed at zero, and
- `pageSize` the number of records to display per page.

`GetProductsAsPagedDataSource` starts by retrieving *all* of the records from `GetProducts()`. It then creates a `PagedDataSource` object, setting its `CurrentPageIndex` and `PageSize` properties to the values of the passed-in `pageIndex` and `pageSize` parameters. The method concludes by returning this configured `PagedDataSource`:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetProductsAsPagedDataSource(ByVal pageIndex As Integer, _
    ByVal pageSize As Integer) As PagedDataSource
    ' Get ALL of the products
    Dim products As Northwind.ProductsDataTable = GetProducts()
    ' Limit the results through a PagedDataSource
    Dim pagedData As New PagedDataSource()
    pagedData.DataSource = products.Rows
    pagedData.AllowPaging = True
    pagedData.CurrentPageIndex = pageIndex
    pagedData.PageSize = pageSize
    Return pagedData
End Function
```

Step 3: Displaying Product Information in a DataList Using Default Paging

With the `GetProductsAsPagedDataSource` method added to the `ProductsBLL` class, we can now create a DataList or Repeater that provides default paging. Start by opening the `Paging.aspx` page in the `PagingSortingDataListRepeater` folder and drag a DataList from the Toolbox onto the Designer, setting the DataList s `ID` property to `ProductsDefaultPaging`. From the DataList s smart tag, create a new ObjectDataSource named `ProductsDefaultPagingDataSource` and configure it so that it retrieves data using the `GetProductsAsPagedDataSource` method.

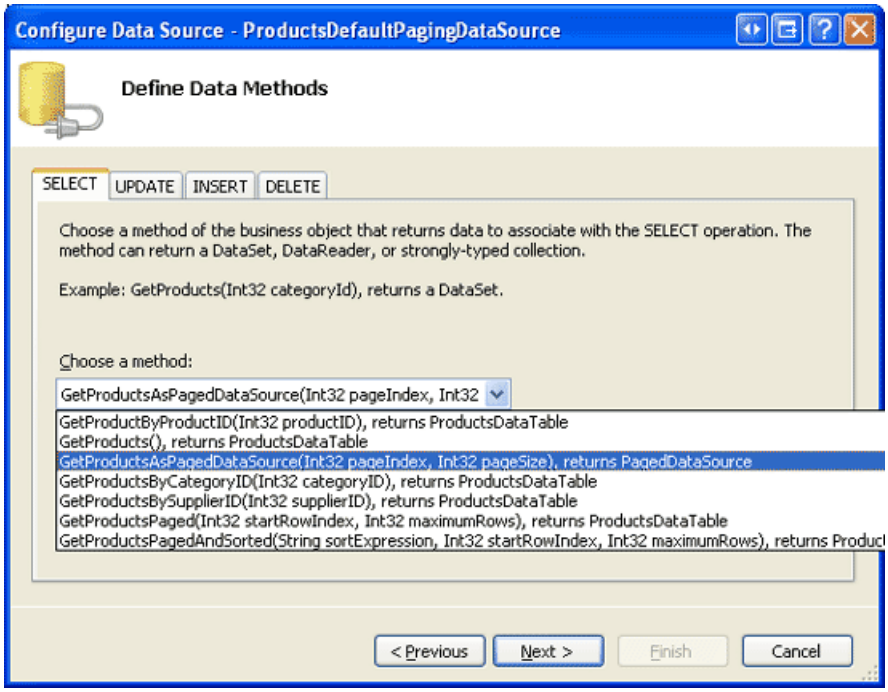


Figure 5: Create an ObjectDataSource and Configure it to Use the `GetProductsAsPagedDataSource()` Method (Click to view full-size image)

Set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) .

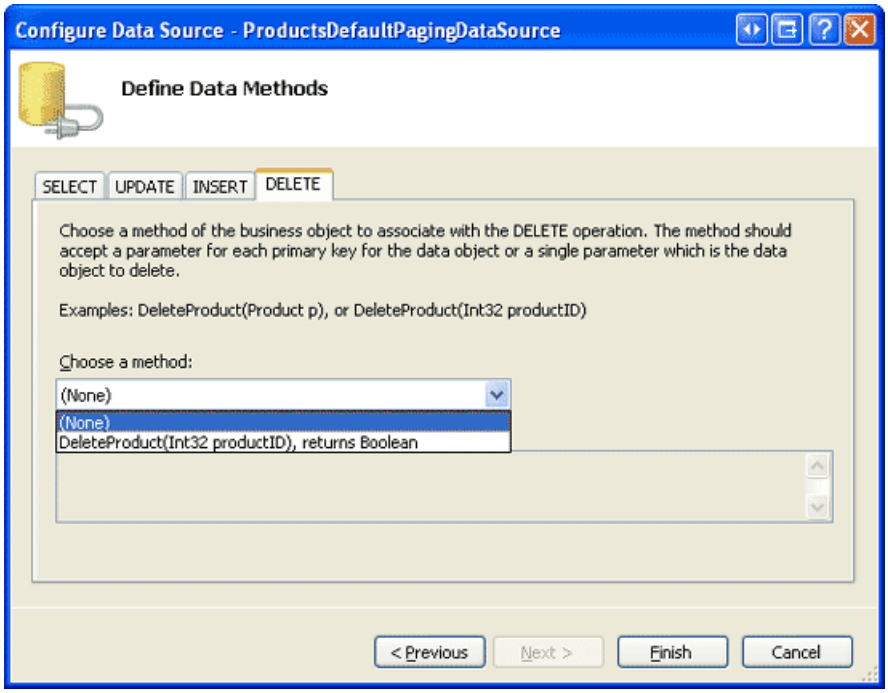


Figure 6: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE tabs to (None) (Click to view full-size image)

Since the `GetProductsAsPagedDataSource` method expects two input parameters, the wizard prompts us for the source of these parameter values.

The page index and page size values must be remembered across postbacks. They can be stored in view state, persisted to the querystring, stored in session variables, or remembered using some other technique. For this tutorial we'll use the querystring, which has the advantage of allowing a particular page of data to be bookmarked.

In particular, use the querystring fields pageIndex and pageSize for the `pageIndex` and `pageSize` parameters, respectively (see Figure 7). Take a moment to set the default values for these parameters, as the querystring values won't be present when a user first visits this page. For `pageIndex`, set the default value to 0 (which will show the first page of data) and `pageSize` s default value to 4.

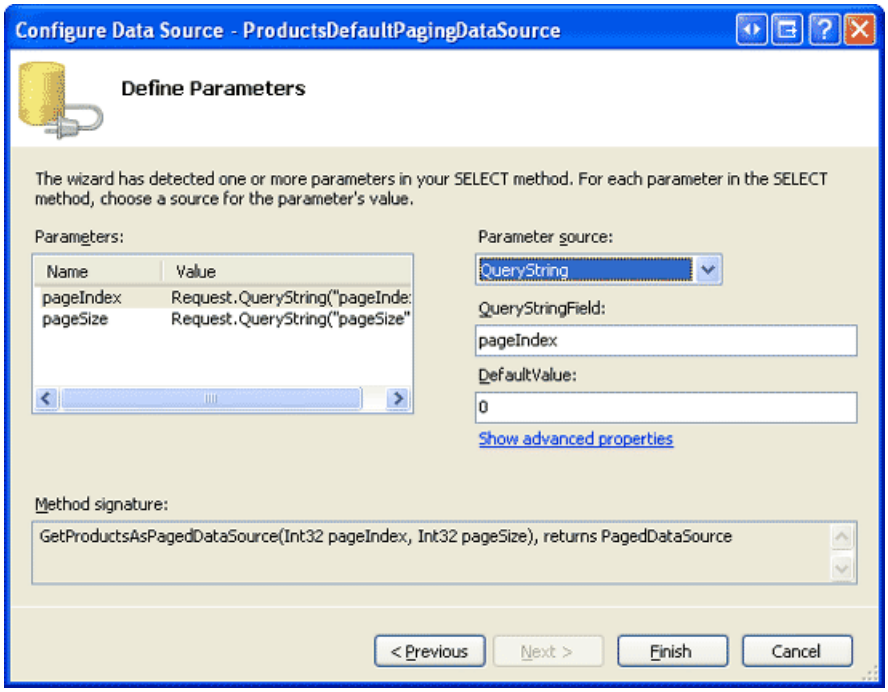


Figure 7: Use the QueryString as the Source for the `pageIndex` and `pageSize` Parameters (Click to view full-size image)

After configuring the ObjectDataSource, Visual Studio automatically creates an `ItemTemplate` for the DataList. Customize the `ItemTemplate` so that only the product s name, category, and supplier are shown. Also set the DataList s `RepeatColumns` property to 2, its `Width` to 100%, and its `ItemStyle` s `Width` to 50%. These width settings will provide equal spacing for the two columns.

After making these changes, the DataList and ObjectDataSource s markup should look similar to the following:

```
<asp:DataList ID="ProductsDefaultPaging" runat="server" Width="100%"
    DataKeyField="ProductID" DataSourceID="ProductsDefaultPagingDataSource"
    RepeatColumns="2" EnableViewState="False">
    <ItemTemplate>
        <h4><asp:Label ID="ProductNameLabel" runat="server"
            Text='<%# Eval("ProductName") %>'></asp:Label></h4>
        Category:
        <asp:Label ID="CategoryNameLabel" runat="server"
            Text='<%# Eval("CategoryName") %>'></asp:Label><br />
        Supplier:
        <asp:Label ID="SupplierNameLabel" runat="server"
            Text='<%# Eval("SupplierName") %>'></asp:Label><br />
        <br />
        <br />
    </ItemTemplate>
    <ItemStyle Width="50%" />
</asp:DataList>
<asp:ObjectDataSource ID="ProductsDefaultPagingDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}" TypeName="ProductsBLL"
    SelectMethod="GetProductsAsPagedDataSource">
    <SelectParameters>
        <asp:QueryStringParameter DefaultValue="0" Name="pageIndex"
             QueryStringField="pageIndex" Type="Int32" />
        <asp:QueryStringParameter DefaultValue="4" Name="pageSize"
             QueryStringField="pageSize" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

NOTE

Since we are not performing any update or delete functionality in this tutorial, you may disable the DataList s view state to reduce the rendered page size.

When initially visiting this page through a browser, neither the `pageIndex` nor `pageSize` querystring parameters are provided. Hence, the default values of 0 and 4 are used. As Figure 8 shows, this results in a DataList that displays the first four products.

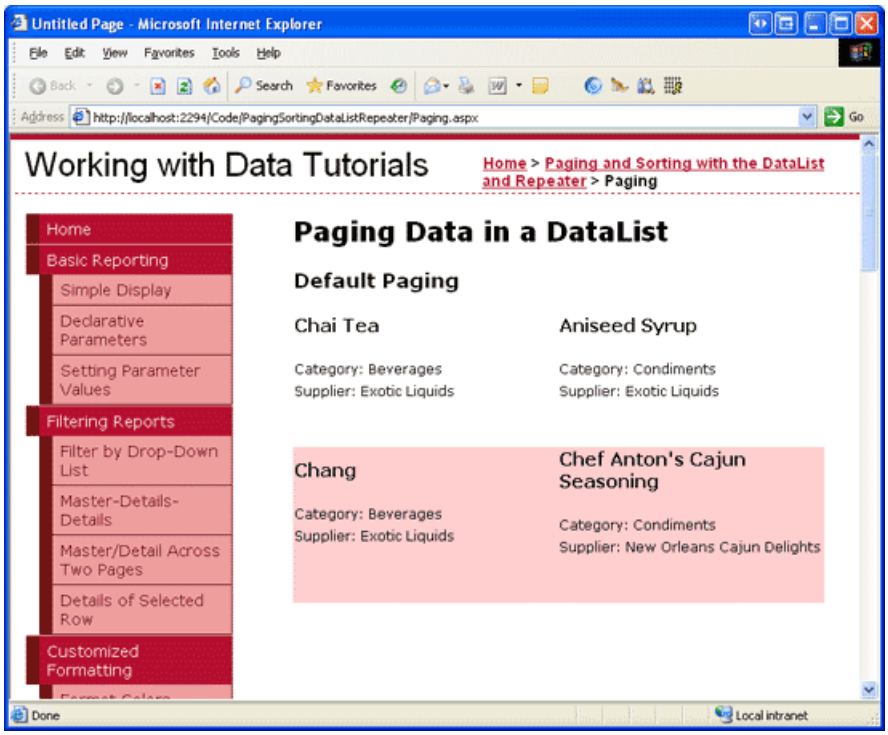


Figure 8: The First Four Products are Listed (Click to view full-size image)

Without a paging interface, there s currently no straightforward means for a user to navigate to the second page of data. We'll create a paging interface in step 4. For now, though, paging can only be accomplished by directly specifying the paging criteria in the querystring. For example, to view the second page, change the URL in the browser s Address bar from `Paging.aspx` to `Paging.aspx?pageIndex=2` and hit Enter. This causes the second page of data to be displayed (see Figure 9).

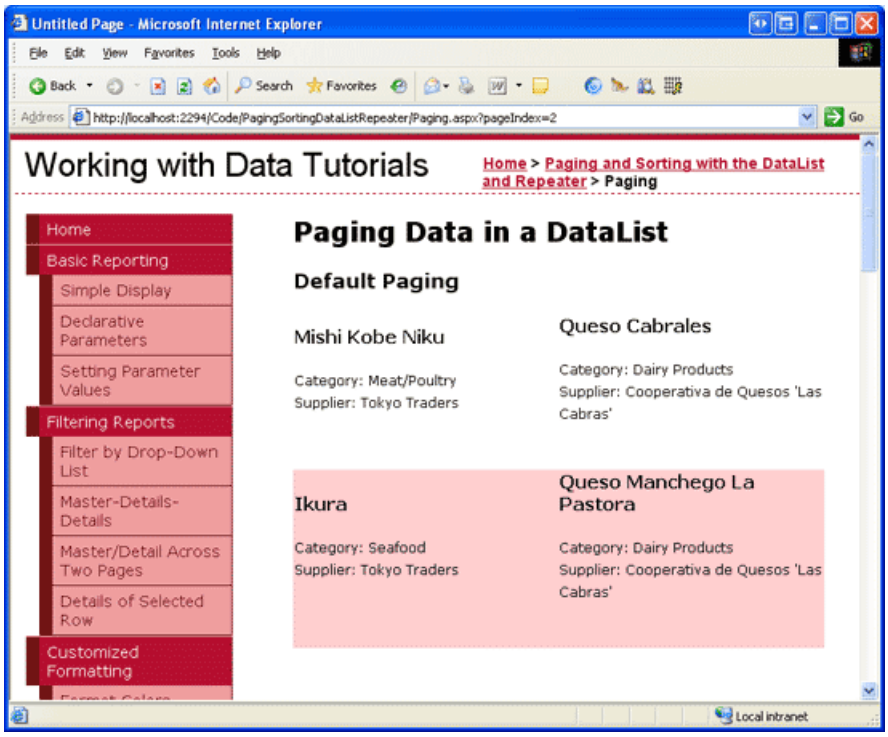


Figure 9: The Second Page of Data is Displayed (Click to view full-size image)

Step 4: Creating the Paging Interface

There are a variety of different paging interfaces that can be implemented. The GridView, DetailsView, and FormView controls provide four different interfaces to choose among:

- **Next, Previous** users can move one page at a time, to either the next or previous one.

- **Next, Previous, First, Last** in addition to Next and Previous buttons, this interface includes First and Last buttons for moving to the very first or very last page.
- **Numeric** lists the page numbers in the paging interface, allowing a user to quickly jump to a particular page.
- **Numeric, First, Last** in addition to the numeric page numbers, includes buttons for moving to the very first or very last page.

For the DataList and Repeater, we are responsible for deciding upon a paging interface and implementing it. This involves creating the needed Web controls in the page and displaying the requested page when a particular paging interface button is clicked. Additionally, certain paging interface controls may need to be disabled. For example, when viewing the first page of data using the Next, Previous, First, Last interface, both the First and Previous buttons would be disabled.

For this tutorial, let s use the Next, Previous, First, Last interface. Add four Button Web controls to the page and set their `ID` s to `FirstPage`, `PrevPage`, `NextPage`, and `LastPage`. Set the `Text` properties to << First, < Prev, Next >, and Last >> .

```
<asp:Button runat="server" ID="FirstPage" Text="<< First" />
<asp:Button runat="server" ID="PrevPage" Text="< Prev" />
<asp:Button runat="server" ID="NextPage" Text="Next >" />
<asp:Button runat="server" ID="LastPage" Text="Last >>" />
```

Next, create a `Click` event handler for each of these Buttons. In a moment we'll add the code necessary to display the requested page.

Remembering the Total Number of Records Being Paged Through

Regardless of the paging interface selected, we need to compute and remember the total number of records being paged through. The total row count (in conjunction with the page size) determines how many total pages of data are being paged through, which determines what paging interface controls are added or are enabled. In the Next, Previous, First, Last interface that we are building, the page count is used in two ways:

- To determine whether we are viewing the last page, in which case the Next and Last buttons are disabled.
- If the user clicks the Last button we need to whisk them to the last page, whose index is one less than the page count.

The page count is calculated as the ceiling of the total row count divided by the page size. For example, if we are paging through 79 records with four records per page, then the page count is 20 (the ceiling of 79 / 4). If we are using the Numeric paging interface, this information informs us as to how many numeric page buttons to display; if our paging interface includes Next or Last buttons, the page count is used to determine when to disable the Next or Last buttons.

If the paging interface includes a Last button, it is imperative that the total number of records being paged through be remembered across postbacks so that when the Last button is clicked we can determine the last page index. To facilitate this, create a `TotalRowCount` property in the ASP.NET page s code-behind class that persists its value to view state:

```
Private Property TotalRowCount() As Integer
    Get
        Dim o As Object = ViewState("TotalRowCount")
        If (o Is Nothing) Then
            Return -1
        Else
            Return Convert.ToInt32(o)
        End If
    End Get
    set(Value as Integer)
        ViewState("TotalRowCount") = value
    End Set
End Property
```

In addition to `TotalRowCount`, take a minute to create read-only page-level properties for easily accessing the page index, page size, and page count:

```
Private ReadOnly Property PageIndex() As Integer
    Get
        If (Not String.IsNullOrEmpty(Request.QueryString("pageIndex"))) Then
            Return Convert.ToInt32(Request.QueryString("pageIndex"))
        Else
            Return 0
        End If
    End Get
End Property
Private ReadOnly Property PageSize() As Integer
    Get
        If (Not String.IsNullOrEmpty(Request.QueryString("pageSize"))) Then
            Return Convert.ToInt32(Request.QueryString("pageSize"))
        Else
            Return 4
        End If
    End Get
End Property
Private ReadOnly Property PageCount() As Integer
    Get
        If TotalRowCount <= 0 OrElse PageSize <= 0 Then
            Return 1
        Else
            Return ((TotalRowCount + PageSize) - 1) / PageSize
        End If
    End Get
End Property
```

Determining the Total Number of Records Being Paged Through

The `PagedDataSource` object returned from the ObjectDataSource s `Select()` method has within it *all* of the product records, even though only a subset of them are displayed in the DataList. The `PagedDataSource` s `Count` property returns only the number of items that will be displayed in the DataList; the `DataSourceCount` property returns the total number of items within the `PagedDataSource`. Therefore, we need to assign the ASP.NET page s `TotalRowCount` property the value of the `PagedDataSource` s `DataSourceCount` property.

To accomplish this, create an event handler for the ObjectDataSource s `Selected` event. In the `Selected` event handler we have access to the return value of the ObjectDataSource s `Select()` method in this case, the `PagedDataSource`.

```
Protected Sub ProductsDefaultPagingDataSource_Selected(ByVal sender As Object, _
    ByVal e As System.Web.UI.WebControls.ObjectDataSourceStatusEventArgs) _
    Handles ProductsDefaultPagingDataSource.Selected
    ' Reference the PagedDataSource bound to the DataList
    Dim pagedData As PagedDataSource = CType(e.ReturnValue, PagedDataSource)
    ' Remember the total number of records being paged through across postbacks
    TotalRowCount = pagedData.DataSourceCount
End Sub
```

Displaying the Requested Page of Data

When the user clicks one of the buttons in the paging interface, we need to display the requested page of data. Since the paging parameters are specified via the querystring, to show the requested page of data use `Response.Redirect(url)` to have the user s browser re-request the `Paging.aspx` page with the appropriate paging parameters. For example, to display the second page of data, we would redirect the user to `Paging.aspx?pageIndex=1`.

To facilitate this, create a `RedirectUser(sendUserToPageIndex)` method that redirects the user to `Paging.aspx?pageIndex=sendUserToPageIndex`. Then, call this method from the four Button `Click` event handlers. In the `FirstPage` `Click` event handler, call `RedirectUser(0)`, to send them to the first page; in the `PrevPage` `Click` event handler, use `PageIndex - 1` as the page index; and so on.

```
Protected Sub FirstPage_Click(ByVal sender As Object, ByVal e As EventArgs) _
    Handles FirstPage.Click
    ' Send the user to the first page
    RedirectUser(0)
End Sub
Protected Sub PrevPage_Click(ByVal sender As Object, ByVal e As EventArgs) _
    Handles PrevPage.Click
    ' Send the user to the previous page
    RedirectUser(PageIndex - 1)
End Sub
Protected Sub NextPage_Click(ByVal sender As Object, ByVal e As EventArgs) _
    Handles NextPage.Click
    ' Send the user to the next page
    RedirectUser(PageIndex + 1)
End Sub
Protected Sub LastPage_Click(ByVal sender As Object, ByVal e As EventArgs) _
    Handles LastPage.Click
    ' Send the user to the last page
    RedirectUser(PageCount - 1)
End Sub
Private Sub RedirectUser(ByVal sendUserToPageIndex As Integer)
    ' Send the user to the requested page
    Response.Redirect(String.Format("Paging.aspx?pageIndex={0}&pageSize={1}", _
        sendUserToPageIndex, PageSize))
End Sub
```

With the `Click` event handlers complete, the DataList s records can be paged through by clicking the buttons. Take a moment to try it out!

Disabling Paging Interface Controls

Currently, all four buttons are enabled regardless of the page being viewed. However, we want to disable the First and Previous buttons when showing the first page of data, and the Next and Last buttons when showing the last page. The `PagedDataSource` object returned by the ObjectDataSource s `Select()` method has properties `IsFirstPage` and `IsLastPage` that we can examine to determine if we are viewing the first or last page of data.

Add the following to the ObjectDataSource s `Selected` event handler:

```
' Configure the paging interface based on the data in the PagedDataSource
FirstPage.Enabled = Not pagedData.IsFirstPage
PrevPage.Enabled = Not pagedData.IsFirstPage
NextPage.Enabled = Not pagedData.IsLastPage
LastPage.Enabled = Not pagedData.IsLastPage
```

With this addition, the First and Previous buttons will be disabled when viewing the first page, while the Next and Last buttons will be disabled when viewing the last page.

Let s complete the paging interface by informing the user what page they re currently viewing and how many total pages exist. Add a Label Web control to the page and set its `ID` property to `CurrentPageNumber`. Set its `Text` property in the ObjectDataSource s Selected event handler such that it includes the current page being viewed (`PageIndex + 1`) and the total number of pages (`PageCount`).

```
' Display the current page being viewed...
CurrentPageNumber.Text = String.Format("You are viewing page {0} of {1}...", _
    PageIndex + 1, PageCount)
```

Figure 10 shows `Paging.aspx` when first visited. Since the querystring is empty, the DataList defaults to showing the first four products; the First and Previous buttons are disabled. Clicking Next displays the next four records (see Figure 11); the First and Previous buttons are now enabled.

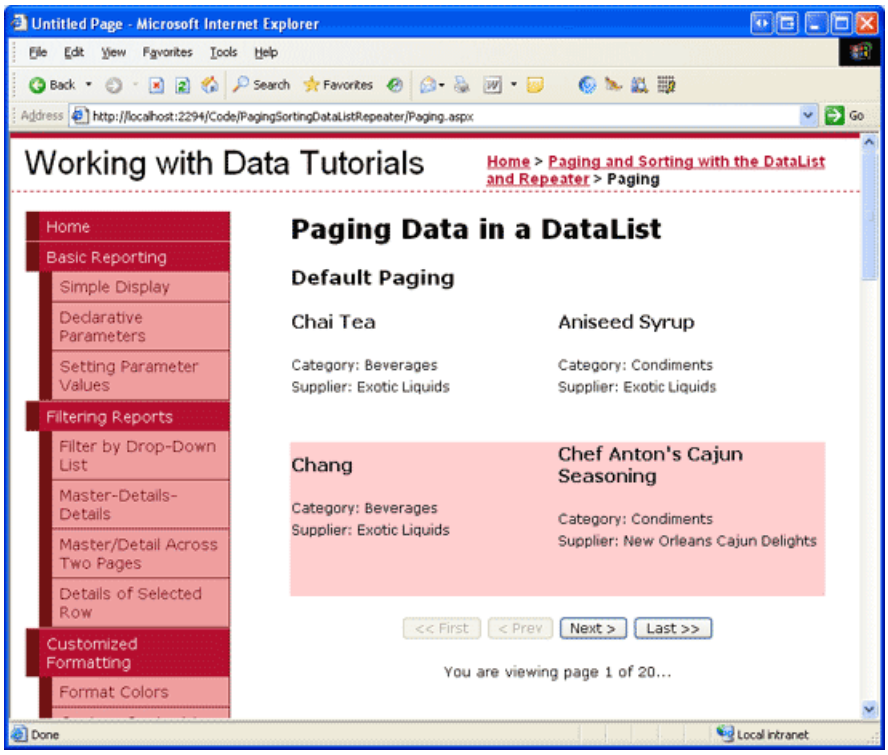


Figure 10: The First Page of Data is Displayed (Click to view full-size image)

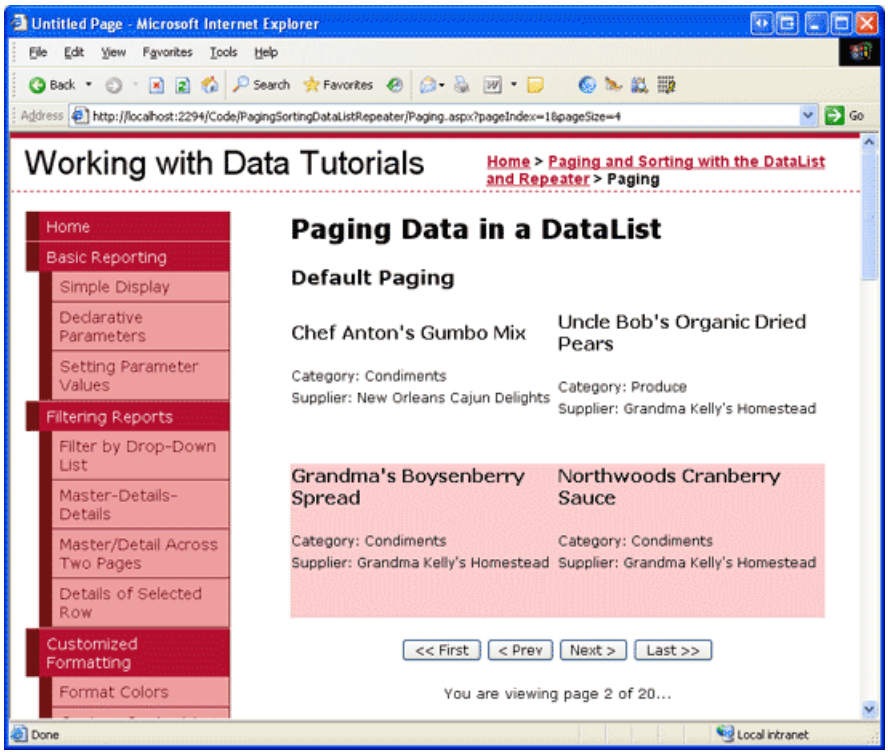


Figure 11: The Second Page of Data is Displayed (Click to view full-size image)

> **NOTE**
>
> The paging interface can be further enhanced by allowing the user to specify how many pages to view per page. For example, a DropDownList could be added listing page size options like 5, 10, 25, 50, and All. Upon selecting a page size, the user would need to be redirected back to `Paging.aspx?pageIndex=0&pageSize=selectedPageSize`. I leave implementing this enhancement as an exercise for the reader.

Using Custom Paging

The DataList pages through its data using the inefficient default paging technique. When paging through sufficiently large amounts of data, it is imperative that custom paging be used. Although the implementation details differ slightly, the concepts behind implementing custom paging in a DataList are the same as with default paging. With custom paging, use the `ProductBLL` class s `GetProductsPaged` method (instead of `GetProductsAsPagedDataSource`). As discussed in the Efficiently Paging Through Large Amounts of Data tutorial, `GetProductsPaged` must be passed the start row index and maximum number of rows to return. These parameters can be maintained through the querystring just like the `pageIndex` and `pageSize` parameters used in default paging.

Since there s no `PagedDataSource` with custom paging, alternative techniques must be used to determine the total number of records being paged through and whether we re displaying the first or last page of data. The `TotalNumberOfProducts()` method in the `ProductsBLL` class returns the total number of products being paged through. To determine if the first page of data is being viewed, examine the start row index if it is zero, then the first page is being viewed. The last page is being viewed if the start row index plus the maximum rows to return is greater than or equal to the total number of records being paged through.

We'll explore implementing custom paging in greater detail in the next tutorial.

Summary

While neither the DataList nor Repeater offers the out of the box paging support found in the GridView, DetailsView, and FormView controls, such functionality can be added with minimal effort. The easiest way to

implement default paging is to wrap the entire set of products within a `PagedDataSource` and then bind the `PagedDataSource` to the DataList or Repeater. In this tutorial we added the `GetProductsAsPagedDataSource` method to the `ProductsBLL` class to return the `PagedDataSource`. The `ProductsBLL` class already contains the methods needed for custom paging `GetProductsPaged` and `TotalNumberOfProducts`.

Along with retrieving either the precise set of records to display for custom paging or all records in a `PagedDataSource` for default paging, we also need to manually add the paging interface. For this tutorial we created a Next, Previous, First, Last interface with four Button Web controls. Also, a Label control displaying the current page number and total number of pages was added.

In the next tutorial we'll see how to add sorting support to the DataList and Repeater. We'll also see how to create a DataList that can be both paged and sorted (with examples using default and custom paging).

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Liz Shulok, Ken Pespisa, and Bernadette Leigh. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Sorting Data in a DataList or Repeater Control (VB)

2/20/2020 • 26 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll examine how to include sorting support in the DataList and Repeater, as well as how to construct a DataList or Repeater whose data can be paged and sorted.

Introduction

In the previous tutorial we examined how to add paging support to a DataList. We created a new method in the `ProductsBLL` class (`GetProductsAsPagedDataSource`) that returned a `PagedDataSource` object. When bound to a DataList or Repeater, the DataList or Repeater would display just the requested page of data. This technique is similar to what is used internally by the GridView, DetailsView, and FormView controls to provide their built-in default paging functionality.

In addition to offering paging support, the GridView also includes out of the box sorting support. Neither the DataList nor Repeater provides built-in sorting functionality; however, sorting features can be added with a bit of code. In this tutorial we'll examine how to include sorting support in the DataList and Repeater, as well as how to construct a DataList or Repeater whose data can be paged and sorted.

A Review of Sorting

As we saw in the Paging and Sorting Report Data tutorial, the GridView control provides out of the box sorting support. Each GridView field can have an associated `SortExpression` , which indicates the data field by which to sort the data. When the GridView s `AllowSorting` property is set to `true` , each GridView field that has a `SortExpression` property value has its header rendered as a LinkButton. When a user clicks a particular GridView field s header, a postback occurs and the data is sorted according to the clicked field s `SortExpression` .

The GridView control has a `SortExpression` property as well, which stores the `SortExpression` of the GridView field the data is sorted by. Additionally, a `SortDirection` property indicates whether the data is to be sorted in ascending or descending order (if a user clicks a particular GridView field s header link twice in succession, the sort order is toggled).

When the GridView is bound to its data source control, it hands off its `SortExpression` and `SortDirection` properties to the data source control. The data source control retrieves the data and then sorts it according to the supplied `SortExpression` and `SortDirection` properties. After sorting the data, the data source control returns it to the GridView.

To replicate this functionality with the DataList or Repeater controls, we must:

- Create a sorting interface
- Remember the data field to sort by and whether to sort in ascending or descending order
- Instruct the ObjectDataSource to sort the data by a particular data field

We'll tackle these three tasks in steps 3 and 4. Following that, we'll examine how to include both paging and sorting support in a DataList or Repeater.

Step 2: Displaying the Products in a Repeater

Before we worry about implementing any of the sorting-related functionality, let s start by listing the products in a Repeater control. Start by opening the `Sorting.aspx` page in the `PagingSortingDataListRepeater` folder. Add a Repeater control to the web page, setting its `ID` property to `SortableProducts`. From the Repeater s smart tag, create a new ObjectDataSource named `ProductsDataSource` and configure it to retrieve data from the `ProductsBLL` class s `GetProducts()` method. Select the (None) option from the drop-down lists in the INSERT, UPDATE, and DELETE tabs.

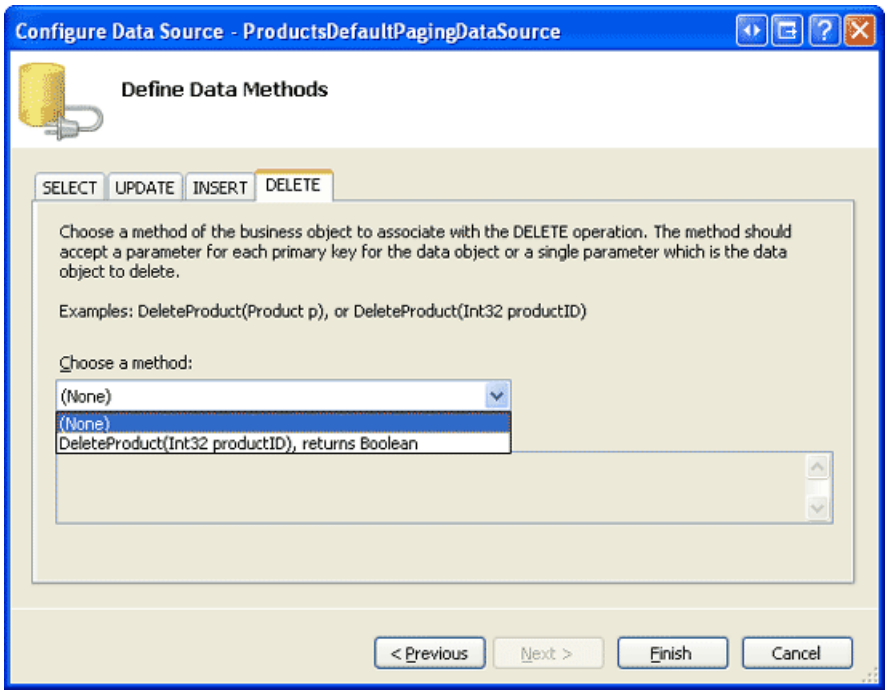


Figure 1: Create an ObjectDataSource and Configure it to Use the `GetProductsAsPagedDataSource()` Method (Click to view full-size image)

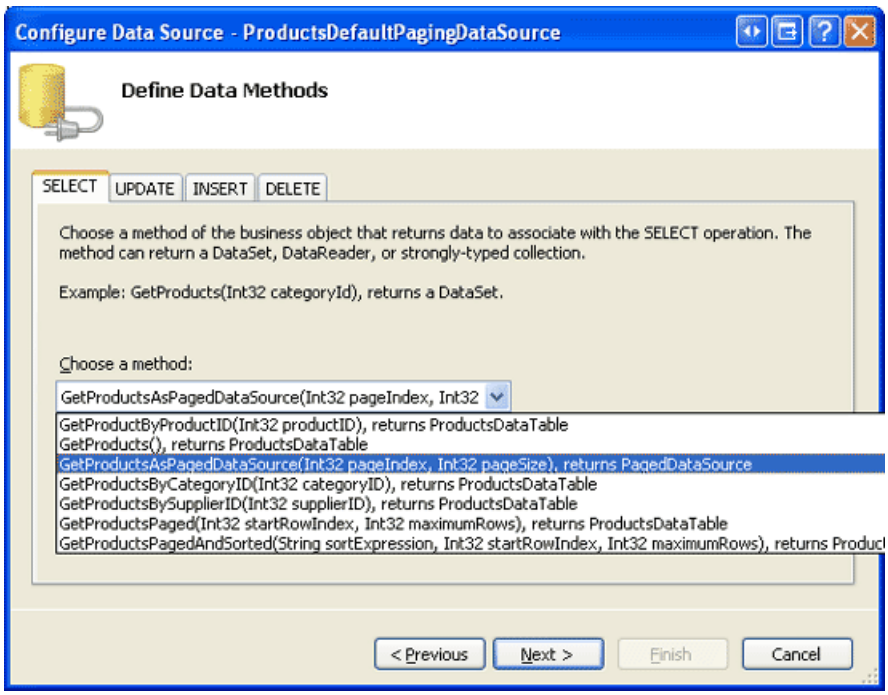


Figure 2: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE tabs to (None) (Click to view full-size image)

Unlike with the DataList, Visual Studio does not automatically create an `ItemTemplate` for the Repeater control after binding it to a data source. Furthermore, we must add this `ItemTemplate` declaratively, as the Repeater control s smart tag lacks the Edit Templates option found in the DataList s. Let s use the same `ItemTemplate` from the previous tutorial, which displayed the product s name, supplier, and category.

After adding the `ItemTemplate`, the Repeater and ObjectDataSource s declarative markup should look similar to the following:

```
<asp:Repeater ID="SortableProducts" DataSourceID="ProductsDataSource"
    EnableViewState="False" runat="server">
    <ItemTemplate>
        <h4><asp:Label ID="ProductNameLabel" runat="server"
            Text='<%# Eval("ProductName") %>'></asp:Label></h4>
        Category:
        <asp:Label ID="CategoryNameLabel" runat="server"
            Text='<%# Eval("CategoryName") %>'></asp:Label><br />
        Supplier:
        <asp:Label ID="SupplierNameLabel" runat="server"
            Text='<%# Eval("SupplierName") %>'></asp:Label><br />
        <br />
        <br />
    </ItemTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}" TypeName="ProductsBLL"
    SelectMethod="GetProducts">
</asp:ObjectDataSource>
```

Figure 3 shows this page when viewed through a browser.

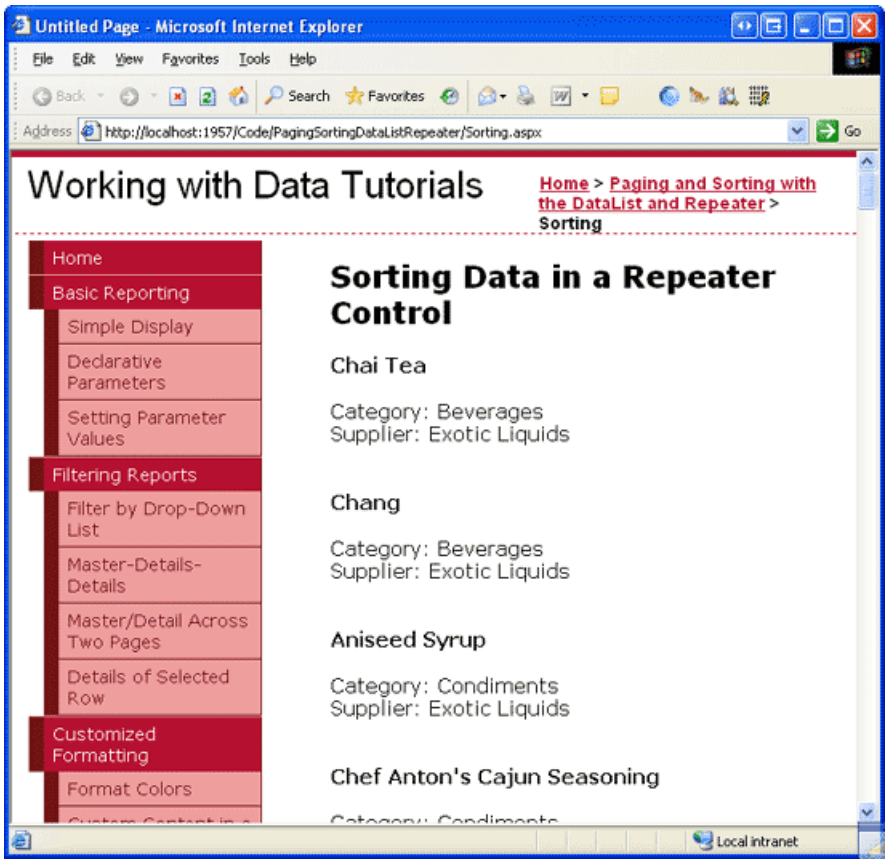


Figure 3: Each Product s Name, Supplier, and Category is Displayed (Click to view full-size image)

Step 3: Instructing the ObjectDataSource to Sort the Data

To sort the data displayed in the Repeater, we need to inform the ObjectDataSource of the sort expression by which the data should be sorted. Before the ObjectDataSource retrieves its data, it first fires its `Selecting` event, which provides an opportunity for us to specify a sort expression. The `Selecting` event handler is passed an object of type `ObjectDataSourceSelectingEventArgs`, which has a property named `Arguments` of type `DataSourceSelectArguments`. The `DataSourceSelectArguments` class is designed to pass data-related requests from a

consumer of data to the data source control, and includes a `SortExpression` property.

To pass sorting information from the ASP.NET page to the ObjectDataSource, create an event handler for the `Selecting` event and use the following code:

```
Protected Sub ProductsDataSource_Selecting(ByVal sender As Object, _
    ByVal e As ObjectDataSourceSelectingEventArgs) _
    Handles ProductsDataSource.Selecting
    e.Arguments.SortExpression = sortExpression
End Sub
```

The *sortExpression* value should be assigned the name of the data field to sort the data by (such as ProductName). There is no sort direction-related property, so if you want to sort the data in descending order, append the string DESC to the *sortExpression* value (such as ProductName DESC).

Go ahead and try some different hard-coded values for *sortExpression* and test the results in a browser. As Figure 4 shows, when using ProductName DESC as the *sortExpression*, the products are sorted by their name in reverse alphabetical order.

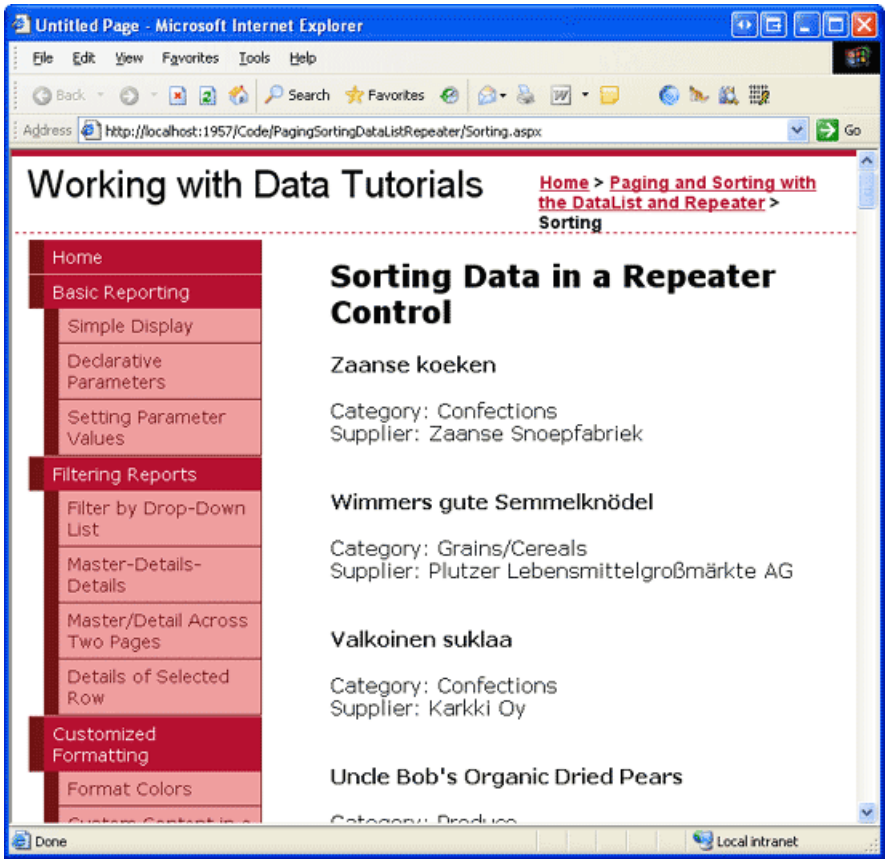


Figure 4: The Products are Sorted by their Name in Reverse Alphabetical Order (Click to view full-size image)

Step 4: Creating the Sorting Interface and Remembering the Sort Expression and Direction

Turning on sorting support in the GridView converts each sortable field s header text into a LinkButton that, when clicked, sorts the data accordingly. Such a sorting interface makes sense for the GridView, where its data is neatly laid out in columns. For the DataList and Repeater controls, however, a different sorting interface is needed. A common sorting interface for a list of data (as opposed to a grid of data), is a drop-down list that provides the fields by which the data can be sorted. Let s implement such an interface for this tutorial.

Add a DropDownList Web control above the `SortableProducts` Repeater and set its `ID` property to `SortBy`. From the Properties window, click the ellipses in the `Items` property to bring up the ListItem Collection Editor. Add

`ListItem` s to sort the data by the `ProductName` , `CategoryName` , and `SupplierName` fields. Also add a `ListItem` to sort the products by their name in reverse alphabetical order.

The `ListItem` `Text` properties can be set to any value (such as Name), but the `Value` properties must be set to the name of the data field (such as ProductName). To sort the results in descending order, append the string DESC to the data field name, like ProductName DESC .

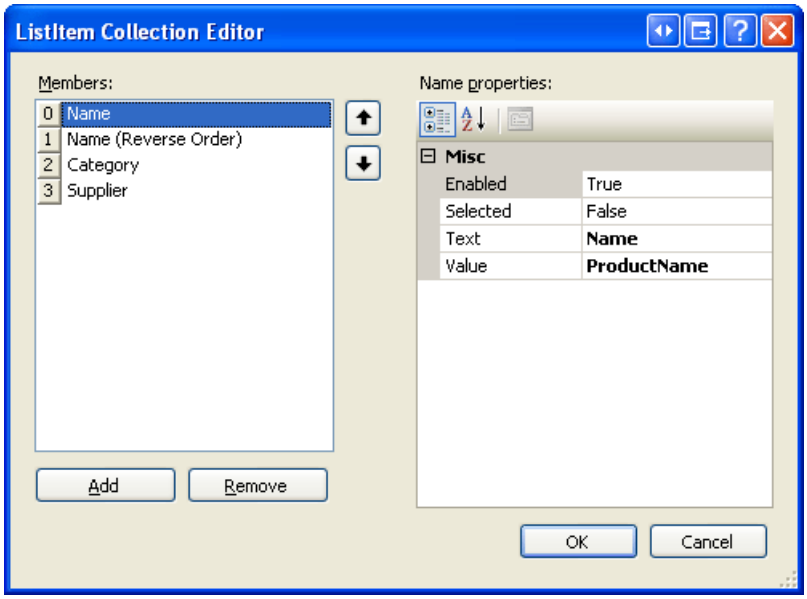


Figure 5: Add a `ListItem` for Each of the Sortable Data Fields

Finally, add a Button Web control to the right of the DropDownList. Set its `ID` to `RefreshRepeater` and its `Text` property to Refresh .

After creating the `ListItem` s and adding the Refresh button, the DropDownList and Button s declarative syntax should look similar to the following:

```
<asp:DropDownList ID="SortBy" runat="server">
    <asp:ListItem Value="ProductName">Name</asp:ListItem>
    <asp:ListItem Value="ProductName DESC">Name (Reverse Order)
        </asp:ListItem>
    <asp:ListItem Value="CategoryName">Category</asp:ListItem>
    <asp:ListItem Value="SupplierName">Supplier</asp:ListItem>
</asp:DropDownList>
<asp:Button runat="server" ID="RefreshRepeater" Text="Refresh" />
```

With the sorting DropDownList complete, we next need to update the ObjectDataSource s `Selecting` event handler so that it uses the selected `SortBy``ListItem` s `Value` property as opposed to a hard-coded sort expression.

```
Protected Sub ProductsDataSource_Selecting _
    (ByVal sender As Object, ByVal e As ObjectDataSourceSelectingEventArgs) _
    Handles ProductsDataSource.Selecting
    ' Have the ObjectDataSource sort the results by the
    ' selected sort expression
    e.Arguments.SortExpression = SortBy.SelectedValue
End Sub
```

At this point when first visiting the page the products will initially be sorted by the `ProductName` data field, as it s the `SortBy` `ListItem` selected by default (see Figure 6). Selecting a different sorting option such as Category and clicking Refresh will cause a postback and re-sort the data by the category name, as Figure 7 shows.

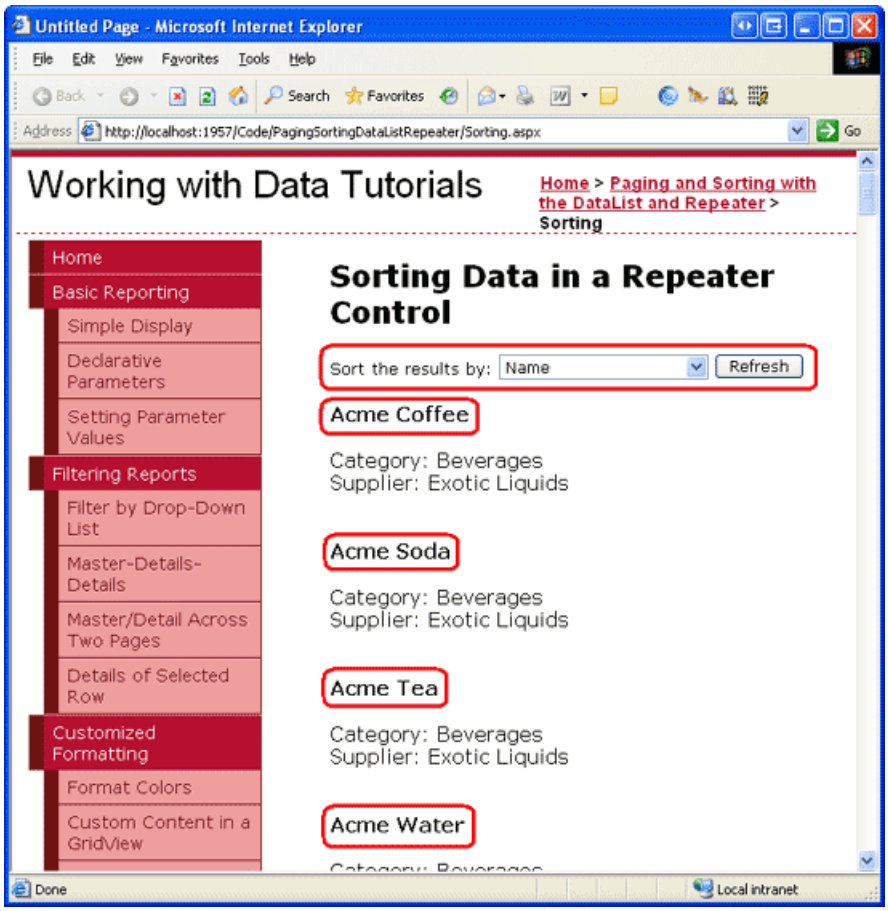


Figure 6: The Products are Initially Sorted by their Name (Click to view full-size image)

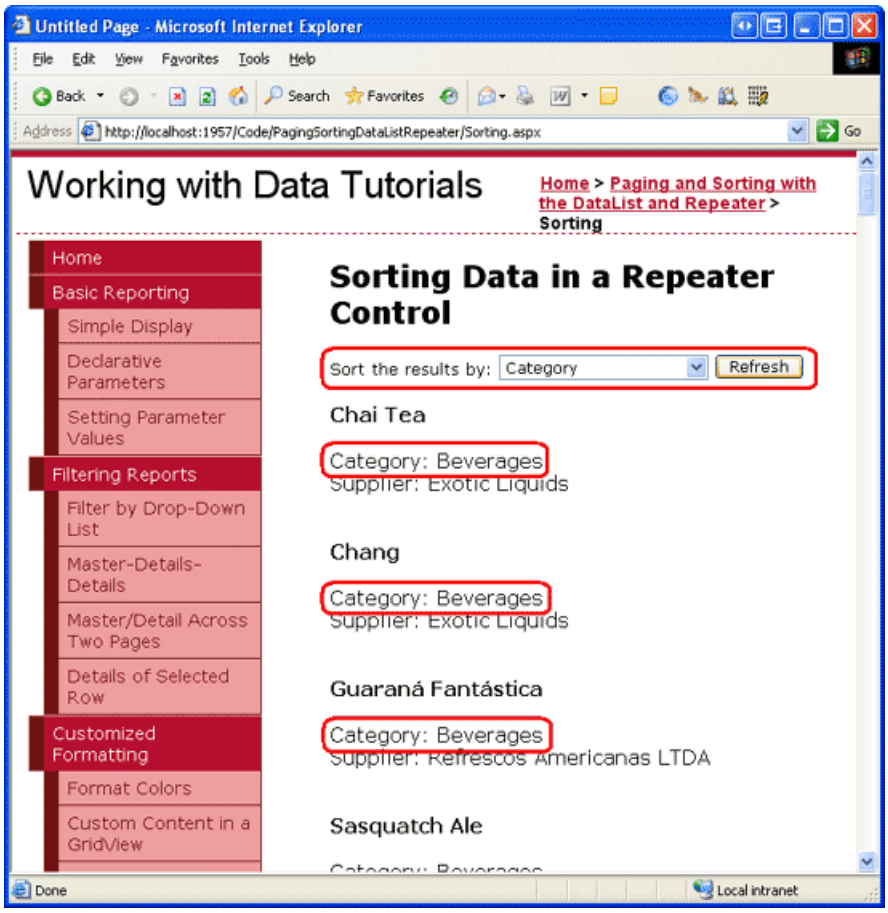


Figure 7: The Products are Now Sorted by Category (Click to view full-size image)

NOTE

Clicking the Refresh button causes the data to automatically be re-sorted because the Repeater s view state has been disabled, thereby causing the Repeater to rebind to its data source on every postback. If you ve left the Repeater s view state enabled, changing the sorting drop-down list won't have any affect on the sort order. To remedy this, create an event handler for the Refresh Button s `Click` event and rebind the Repeater to its data source (by calling the Repeater s `DataBind()` method).

Remembering the Sort Expression and Direction

When creating a sortable DataList or Repeater on a page where non-sort related postbacks may occur, it s imperative that the sort expression and direction be remembered across postbacks. For example, imagine that we updated the Repeater in this tutorial to include a Delete button with each product. When the user clicks the Delete button we d run some code to delete the selected product and then rebind the data to the Repeater. If the sort details are not persisted across postback, the data displayed on screen will revert to the original sort order.

For this tutorial, the DropDownList implicitly saves the sort expression and direction in its view state for us. If we were using a different sorting interface one with, say, LinkButtons that provided the various sorting options we d need to take care to remember the sort order across postbacks. This could be accomplished by storing the sorting parameters in the page s view state, by including the sort parameter in the querystring, or through some other state persistence technique.

Future examples in this tutorial explore how to persist the sorting details in the page s view state.

Step 5: Adding Sorting Support to a DataList that Uses Default Paging

In the preceding tutorial we examined how to implement default paging with a DataList. Let s extend this previous example to include the ability to sort the paged data. Start by opening the `SortingWithDefaultPaging.aspx` and `Paging.aspx` pages in the `PagingSortingDataListRepeater` folder. From the `Paging.aspx` page, click on the Source button to view the page s declarative markup. Copy the selected text (see Figure 8) and paste it into the declarative markup of `SortingWithDefaultPaging.aspx` between the `<asp:Content>` tags.

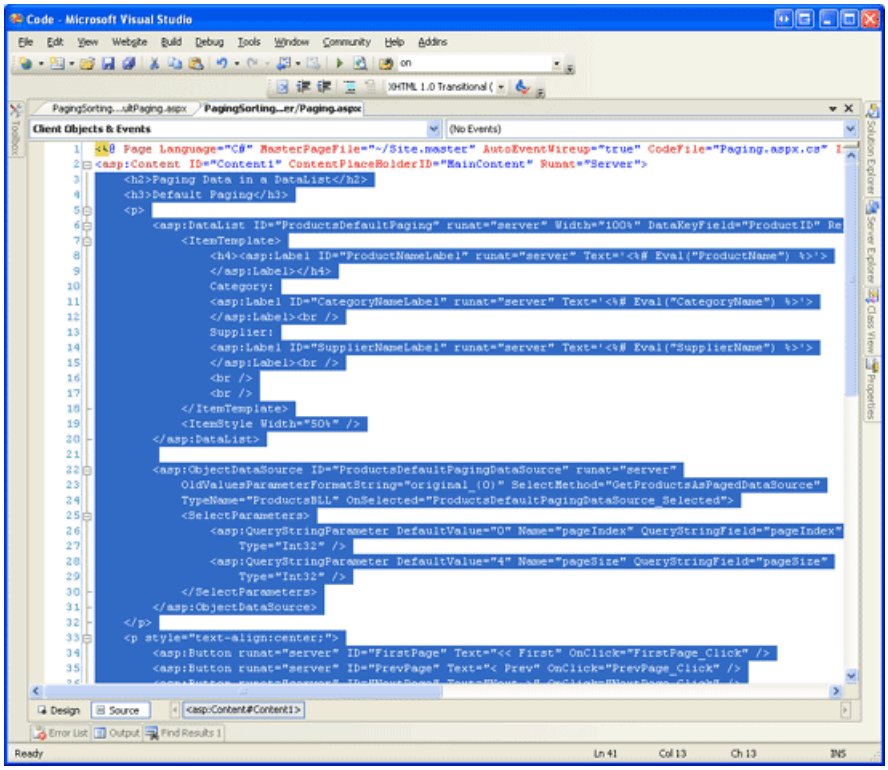


Figure 8: Replicate the Declarative Markup in the `<asp:Content>` Tags from `Paging.aspx` to

`SortingWithDefaultPaging.aspx` (Click to view full-size image)

After copying the declarative markup, copy the methods and properties in the `Paging.aspx` page s code-behind class to the code-behind class for `SortingWithDefaultPaging.aspx`. Next, take a moment to view the `SortingWithDefaultPaging.aspx` page in a browser. It should exhibit the same functionality and appearance as `Paging.aspx`.

Enhancing ProductsBLL to Include a Default Paging and Sorting Method

In the previous tutorial we created a `GetProductsAsPagedDataSource(pageIndex, pageSize)` method in the `ProductsBLL` class that returned a `PagedDataSource` object. This `PagedDataSource` object was populated with *all* of the products (via the BLL s `GetProducts()` method), but when bound to the DataList only those records corresponding to the specified *pageIndex* and *pageSize* input parameters were displayed.

Earlier in this tutorial we added sorting support by specifying the sort expression from the ObjectDataSource s `Selecting` event handler. This works well when the ObjectDataSource is returned an object that can be sorted, like the `ProductsDataTable` returned by the `GetProducts()` method. However, the `PagedDataSource` object returned by the `GetProductsAsPagedDataSource` method does not support sorting of its inner data source. Instead, we need to sort the results returned from the `GetProducts()` method *before* we put it in the `PagedDataSource`.

To accomplish this, create a new method in the `ProductsBLL` class, `GetProductsSortedAsPagedDataSource(sortExpression, pageIndex, pageSize)`. To sort the `ProductsDataTable` returned by the `GetProducts()` method, specify the `Sort` property of its default `DataTableView`:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetProductsSortedAsPagedDataSource _
    (sortExpression As String, pageIndex As Integer, pageSize As Integer) _
    As PagedDataSource
    ' Get ALL of the products
    Dim products As Northwind.ProductsDataTable = GetProducts()
    'Sort the products
    products.DefaultView.Sort = sortExpression
    ' Limit the results through a PagedDataSource
    Dim pagedData As New PagedDataSource()
    pagedData.DataSource = products.DefaultView
    pagedData.AllowPaging = True
    pagedData.CurrentPageIndex = pageIndex
    pagedData.PageSize = pageSize
    Return pagedData
End Function
```

The `GetProductsSortedAsPagedDataSource` method differs only slightly from the `GetProductsAsPagedDataSource` method created in the previous tutorial. In particular, `GetProductsSortedAsPagedDataSource` accepts an additional input parameter `sortExpression` and assigns this value to the `Sort` property of the `ProductDataTable` s `DefaultView`. A few lines of code later, the `PagedDataSource` object s DataSource is assigned the `ProductDataTable` s `DefaultView`.

Calling the GetProductsSortedAsPagedDataSource Method and Specifying the Value for the SortExpression Input Parameter

With the `GetProductsSortedAsPagedDataSource` method complete, the next step is to provide the value for this parameter. The ObjectDataSource in `SortingWithDefaultPaging.aspx` is currently configured to call the `GetProductsAsPagedDataSource` method and passes in the two input parameters through its two

`QueryStringParameters`, which are specified in the `SelectParameters` collection. These two `QueryStringParameters` indicate that the source for the `GetProductsAsPagedDataSource` method s *pageIndex* and *pageSize* parameters come from the querystring fields `pageIndex` and `pageSize`.

Update the ObjectDataSource s `SelectMethod` property so that it invokes the new `GetProductsSortedAsPagedDataSource` method. Then, add a new `QueryStringParameter` so that the *sortExpression* input parameter is accessed from the querystring field `sortExpression`. Set the `QueryStringParameter` s `DefaultValue` to ProductName .

After these changes, the ObjectDataSource s declarative markup should look like:

```
<asp:ObjectDataSource ID="ProductsDefaultPagingDataSource"
    OldValuesParameterFormatString="original_{0}" TypeName="ProductsBLL"
    SelectMethod="GetProductsSortedAsPagedDataSource"
    OnSelected="ProductsDefaultPagingDataSource_Selected" runat="server">
    <SelectParameters>
        <asp:QueryStringParameter DefaultValue="ProductName"
            Name="sortExpression" QueryStringField="sortExpression"
            Type="String" />
        <asp:QueryStringParameter DefaultValue="0" Name="pageIndex"
            QueryStringField="pageIndex" Type="Int32" />
        <asp:QueryStringParameter DefaultValue="4" Name="pageSize"
            QueryStringField="pageSize" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

At this point, the `SortingWithDefaultPaging.aspx` page will sort its results alphabetically by the product name (see Figure 9). This is because, by default, a value of ProductName is passed in as the `GetProductsSortedAsPagedDataSource` method s *sortExpression* parameter.

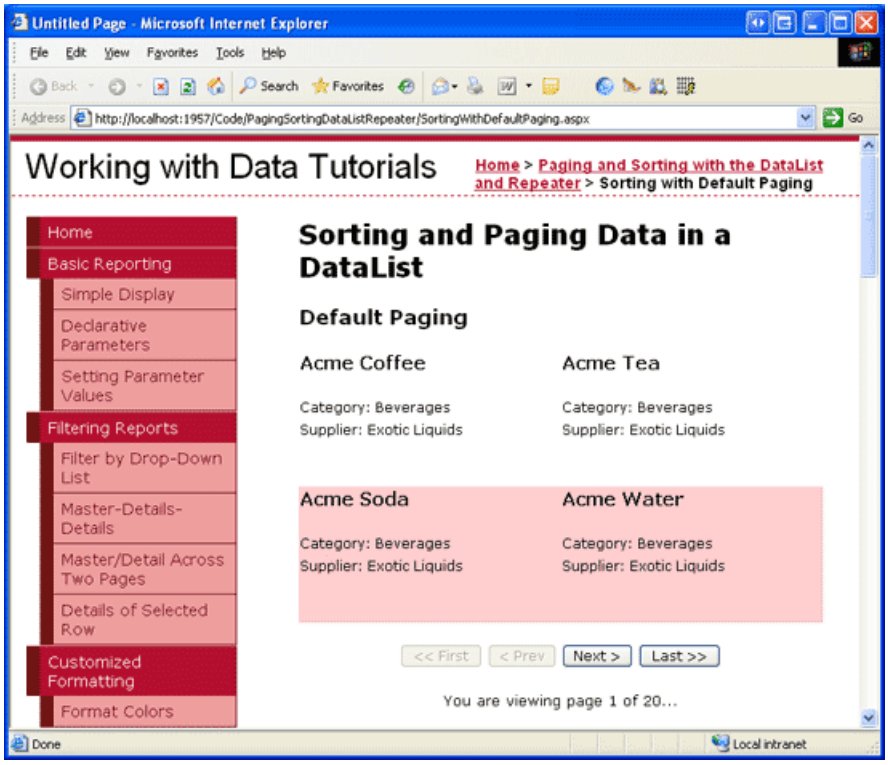


Figure 9: By Default, the Results are Sorted by `ProductName` (Click to view full-size image)

If you manually add a `sortExpression` querystring field such as `SortingWithDefaultPaging.aspx?sortExpression=CategoryName` the results will be sorted by the specified `sortExpression`. However, this `sortExpression` parameter is not included in the querystring when moving to a different page of data. In fact, clicking on the Next or Last page buttons takes us back to `Paging.aspx`!

Furthermore, there s currently no sorting interface. The only way a user can change the sort order of the paged data is by manipulating the querystring directly.

Creating the Sorting Interface

We first need to update the `RedirectUser` method to send the user to `SortingWithDefaultPaging.aspx` (instead of `Paging.aspx`) and to include the `sortExpression` value in the querystring. We should also add a read-only, page-level named `SortExpression` property. This property, similar to the `PageIndex` and `PageSize` properties created in the previous tutorial, returns the value of the `sortExpression` querystring field if it exists, and the default value (ProductName) otherwise.

Currently the `RedirectUser` method accepts only a single input parameter the index of the page to display. However, there may be times when we want to redirect the user to a particular page of data using a sort expression other than what s specified in the querystring. In a moment we'll create the sorting interface for this page, which will include a series of Button Web controls for sorting the data by a specified column. When one of those Buttons is clicked, we want to redirect the user passing in the appropriate sort expression value. To provide this functionality, create two versions of the `RedirectUser` method. The first one should accept just the page index to display, while the second one accepts the page index and sort expression.

```
Private ReadOnly Property SortExpression() As String
    Get
        If Not String.IsNullOrEmpty(Request.QueryString("sortExpression")) Then
            Return Request.QueryString("sortExpression")
        Else
            Return "ProductName"
        End If
    End Get
End Property
Private Sub RedirectUser(ByVal sendUserToPageIndex As Integer)
    ' Use the SortExpression property to get the sort expression
    ' from the querystring
    RedirectUser(sendUserToPageIndex, SortExpression)
End Sub
Private Sub RedirectUser(ByVal sendUserToPageIndex As Integer,
    ByVal sendUserSortingBy As String)
    ' Send the user to the requested page with the
    ' requested sort expression
    Response.Redirect(String.Format("SortingWithDefaultPaging.aspx?" & _
        "pageIndex={0}&pageSize={1}&sortExpression={2}", _
        sendUserToPageIndex, PageSize, sendUserSortingBy))
End Sub
```

In the first example in this tutorial, we created a sorting interface using a DropDownList. For this example, let s use three Button Web controls positioned above the DataList one for sorting by `ProductName`, one for `CategoryName`, and one for `SupplierName`. Add the three Button Web controls, setting their `ID` and `Text` properties appropriately:

```
<p>
    <asp:Button runat="server" id="SortByProductName"
        Text="Sort by Product Name" />
    <asp:Button runat="server" id="SortByCategoryName"
        Text="Sort by Category" />
    <asp:Button runat="server" id="SortBySupplierName"
        Text="Sort by Supplier" />
</p>
```

Next, create a `Click` event handler for each. The event handlers should call the `RedirectUser` method, returning the user to the first page using the appropriate sort expression.

```
Protected Sub SortByProductName_Click(sender As Object, e As EventArgs) _
    Handles SortByProductName.Click
    'Sort by ProductName
    RedirectUser(0, "ProductName")
End Sub
Protected Sub SortByCategoryName_Click(sender As Object, e As EventArgs) _
    Handles SortByCategoryName.Click
    'Sort by CategoryName
    RedirectUser(0, "CategoryName")
End Sub
Protected Sub SortBySupplierName_Click(sender As Object, e As EventArgs) _
    Handles SortBySupplierName.Click
    'Sort by SupplierName
    RedirectUser(0, "SupplierName")
End Sub
```

When first visiting the page, the data is sorted by the product name alphabetically (refer back to Figure 9). Click the Next button to advance to the second page of data and then click the Sort by Category button. This returns us to the first page of data, sorted by category name (see Figure 10). Likewise, clicking the Sort by Supplier button sorts the data by supplier starting from the first page of data. The sort choice is remembered as the data is paged through. Figure 11 shows the page after sorting by category and then advancing to the thirteenth page of data.

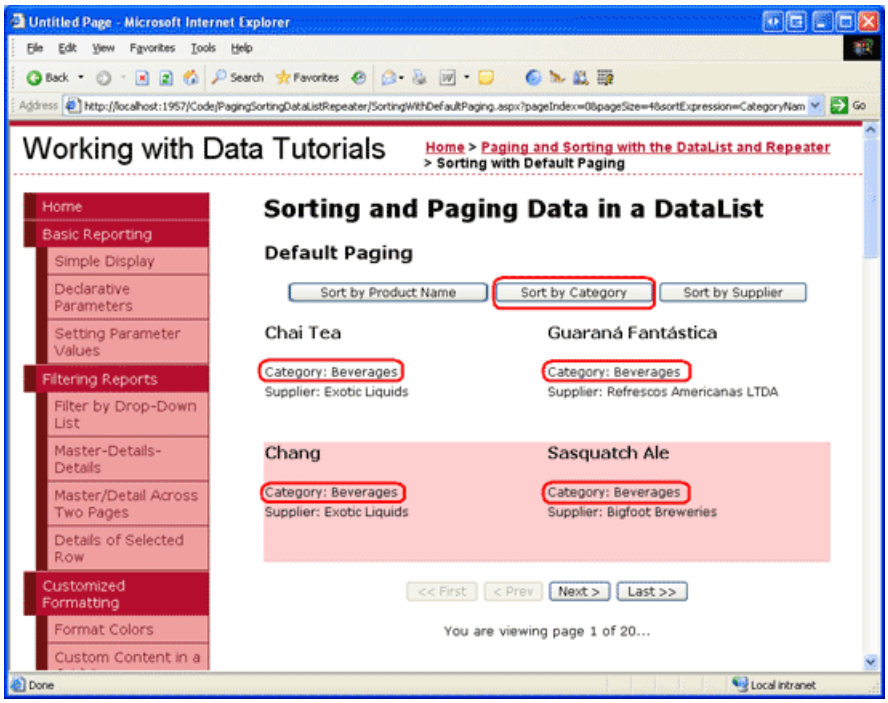


Figure 10: The Products are Sorted by Category (Click to view full-size image)

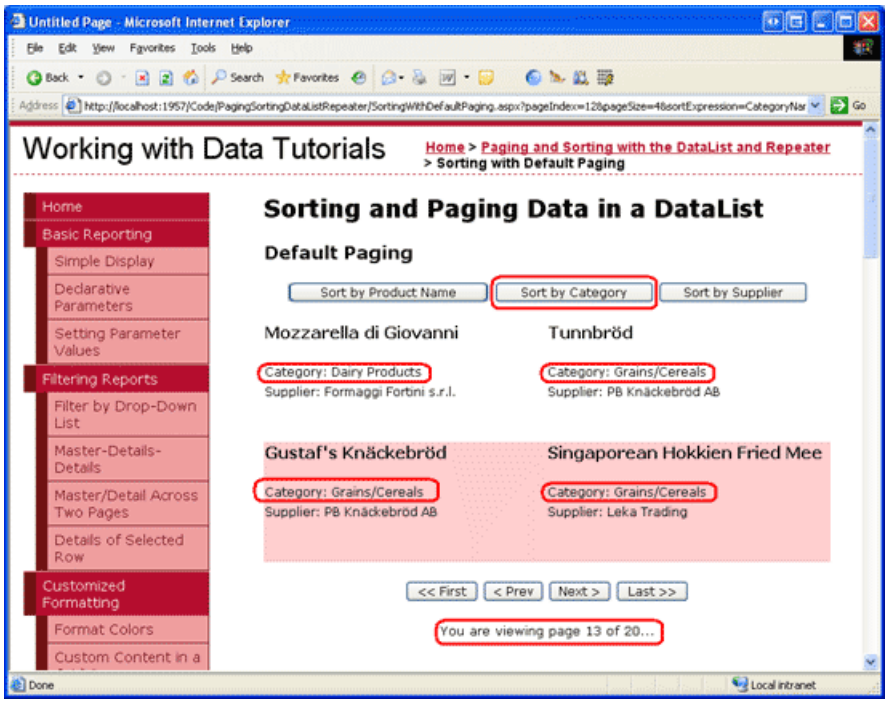


Figure 11: The Sort Expression is Remembered When Paging Through the Data (Click to view full-size image)

Step 6: Custom Paging Through Records in a Repeater

The DataList example examined in step 5 pages through its data using the inefficient default paging technique. When paging through sufficiently large amounts of data, it is imperative that custom paging be used. Back in the Efficiently Paging Through Large Amounts of Data and Sorting Custom Paged Data tutorials, we examined the differences between default and custom paging and created methods in the BLL for utilizing custom paging and sorting custom paged data. In particular, in these two previous tutorials we added the following three methods to the `ProductsBLL` class:

- `GetProductsPaged(startRowIndex, maximumRows)` returns a particular subset of records starting at *startRowIndex* and not exceeding *maximumRows*.
- `GetProductsPagedAndSorted(sortExpression, startRowIndex, maximumRows)` returns a particular subset of records sorted by the specified *sortExpression* input parameter.
- `TotalNumberOfProducts()` provides the total number of records in the `Products` database table.

These methods can be used to efficiently page and sort through data using a DataList or Repeater control. To illustrate this, let s start by creating a Repeater control with custom paging support; we'll then add sorting capabilities.

Open the `SortingWithCustomPaging.aspx` page in the `PagingSortingDataListRepeater` folder and add a Repeater to the page, setting its `ID` property to `Products`. From the Repeater s smart tag, create a new ObjectDataSource named `ProductsDataSource`. Configure it to select its data from the `ProductsBLL` class s `GetProductsPaged` method.

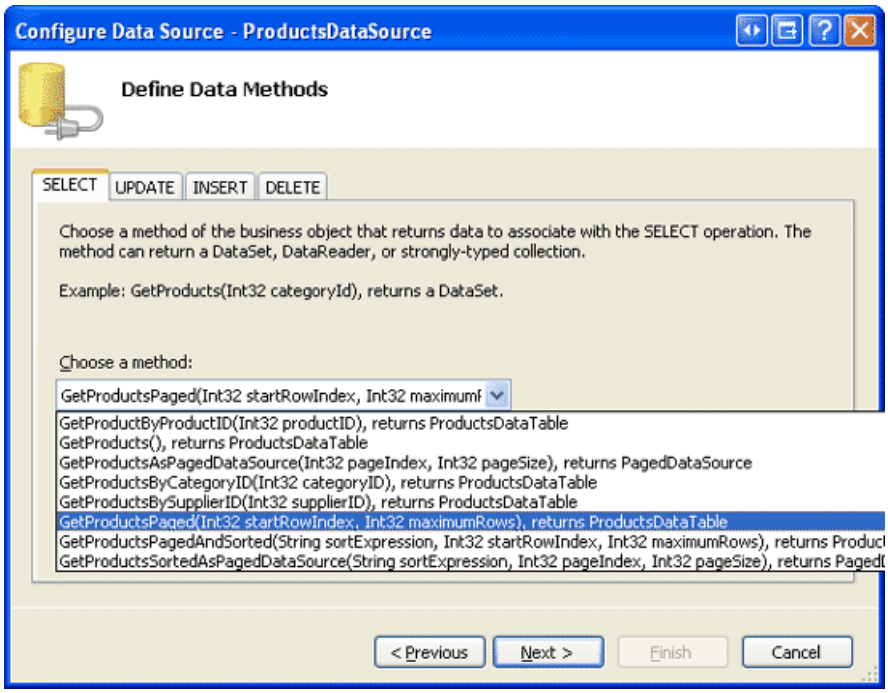


Figure 12: Configure the ObjectDataSource to Use the `ProductsBLL` Class s `GetProductsPaged` Method (Click to view full-size image)

Set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) and then click the Next button. The Configure Data Source wizard now prompts for the sources of the `GetProductsPaged` method s *startRowIndex* and *maximumRows* input parameters. In actuality, these input parameters are ignored. Instead, the *startRowIndex* and *maximumRows* values will be passed in through the `Arguments` property in the ObjectDataSource s `Selecting` event handler, just like how we specified the *sortExpression* in this tutorial s first demo. Therefore, leave the parameter source drop-down lists in the wizard set at None .

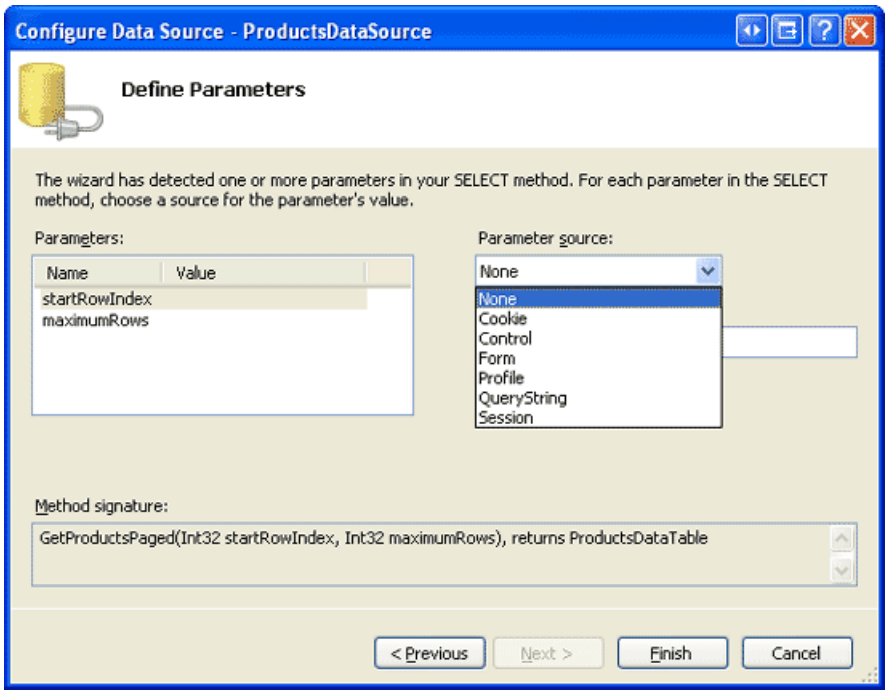


Figure 13: Leave the Parameter Sources Set to None (Click to view full-size image)

NOTE

Do *not* set the ObjectDataSource s `EnablePaging` property to `true`. This will cause the ObjectDataSource to automatically include its own *startRowIndex* and *maximumRows* parameters to the `SelectMethod` s existing parameter list. The `EnablePaging` property is useful when binding custom paged data to a GridView, DetailsView, or FormView control because these controls expect certain behavior from the ObjectDataSource that s only available when `EnablePaging` property is `true`. Since we have to manually add the paging support for the DataList and Repeater, leave this property set to `false` (the default), as we'll bake in the needed functionality directly within our ASP.NET page.

Finally, define the Repeater s `ItemTemplate` so that the product s name, category, and supplier are shown. After these changes, the Repeater and ObjectDataSource s declarative syntax should look similar to the following:

```
<asp:Repeater ID="Products" runat="server" DataSourceID="ProductsDataSource"
    EnableViewState="False">
    <ItemTemplate>
        <h4><asp:Label ID="ProductNameLabel" runat="server"
            Text='<%# Eval("ProductName") %>'></asp:Label></h4>
        Category:
        <asp:Label ID="CategoryNameLabel" runat="server"
            Text='<%# Eval("CategoryName") %>'></asp:Label><br />
        Supplier:
        <asp:Label ID="SupplierNameLabel" runat="server"
            Text='<%# Eval("SupplierName") %>'></asp:Label><br />
        <br />
        <br />
    </ItemTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProductsPaged" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:Parameter Name="startRowIndex" Type="Int32" />
        <asp:Parameter Name="maximumRows" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

Take a moment to visit the page through a browser and note that no records are returned. This is because we ve yet to specify the *startRowIndex* and *maximumRows* parameter values; therefore, values of 0 are being passed in for both. To specify these values, create an event handler for the ObjectDataSource s `Selecting` event and set these parameters values programmatically to hard-coded values of 0 and 5, respectively:

```
Protected Sub ProductsDataSource_Selecting(sender As Object, _
    e As ObjectDataSourceSelectingEventArgs) _
    Handles ProductsDataSource.Selecting
    e.InputParameters("startRowIndex") = 0
    e.InputParameters("maximumRows") = 5
End Sub
```

With this change, the page, when viewed through a browser, shows the first five products.

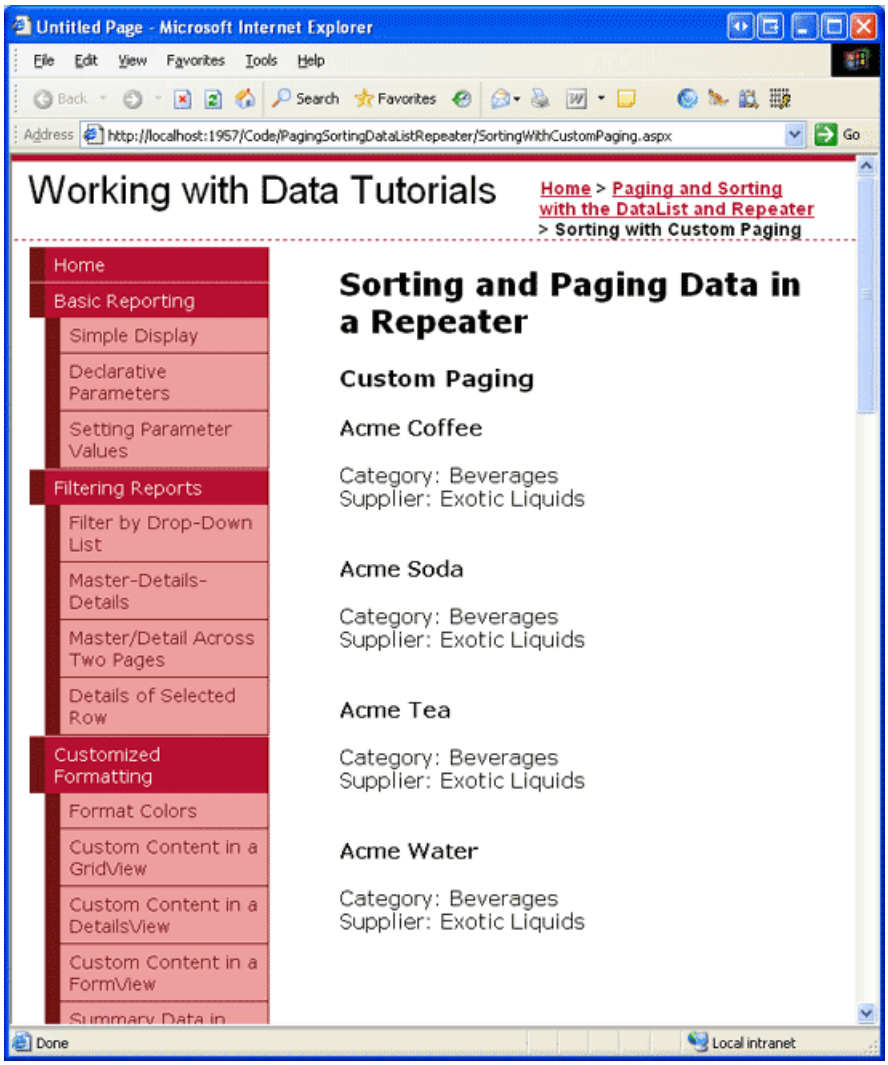


Figure 14: The First Five Records are Displayed (Click to view full-size image)

> **NOTE**
>
> The products listed in Figure 14 happen to be sorted by product name because the `GetProductsPaged` stored procedure that performs the efficient custom paging query orders the results by `ProductName`.

In order to allow the user to step through the pages, we need to keep track of the start row index and maximum rows and remember these values across postbacks. In the default paging example we used querystring fields to persist these values; for this demo, let s persist this information in the page s view state. Create the following two properties:

```
Private Property StartRowIndex() As Integer
    Get
        Dim o As Object = ViewState("StartRowIndex")
        If o Is Nothing Then
            Return 0
        Else
            Return CType(o, Integer)
        End If
    End Get
    Set(ByVal value As Integer)
        ViewState("StartRowIndex") = value
    End Set
End Property
Private Property MaximumRows() As Integer
    Get
        Dim o As Object = ViewState("MaximumRows")
        If o Is Nothing Then
            Return 5
        Else
            Return CType(o, Integer)
        End If
    End Get
    Set(ByVal value As Integer)
        ViewState("MaximumRows") = value
    End Set
End Property
```

Next, update the code in the Selecting event handler so that it uses the `StartRowIndex` and `MaximumRows` properties instead of the hard-coded values of 0 and 5:

```
e.InputParameters("startRowIndex") = 0
e.InputParameters("maximumRows") = 5
```

At this point our page still shows just the first five records. However, with these properties in place, we re ready to create our paging interface.

Adding the Paging Interface

Let s use the same First, Previous, Next, Last paging interface used in the default paging example, including the Label Web control that displays what page of data is being viewed and how many total pages exist. Add the four Button Web controls and Label below the Repeater.

```
<p>
    <asp:Button runat="server" ID="FirstPage" Text="<< First" />
    <asp:Button runat="server" ID="PrevPage" Text="< Prev" />
    <asp:Button runat="server" ID="NextPage" Text="Next >" />
    <asp:Button runat="server" ID="LastPage" Text="Last >>" />
</p>
<p>
    <asp:Label runat="server" ID="CurrentPageNumber"></asp:Label>
</p>
```

Next, create `Click` event handlers for the four Buttons. When one of these Buttons is clicked, we need to update the `StartRowIndex` and rebind the data to the Repeater. The code for the First, Previous, and Next buttons is simple enough, but for the Last button how do we determine the start row index for the last page of data? To compute this index as well as being able to determine whether the Next and Last buttons should be enabled we need to know how many records in total are being paged through. We can determine this by calling the `ProductsBLL` class s `TotalNumberOfProducts()` method. Let s create a read-only, page-level property named `TotalRowCount` that

returns the results of the `TotalNumberOfProducts()` method:

```
Private ReadOnly Property TotalRowCount() As Integer
    Get
        'Return the value from the TotalNumberOfProducts() method
        Dim productsAPI As New ProductsBLL()
        Return productsAPI.TotalNumberOfProducts()
    End Get
End Property
```

With this property we can now determine the last page s start row index. Specifically, it s the integer result of the `TotalRowCount` minus 1 divided by `MaximumRows`, multiplied by `MaximumRows`. We can now write the `Click` event handlers for the four paging interface buttons:

```
Protected Sub FirstPage_Click(sender As Object, e As EventArgs) _
    Handles FirstPage.Click
    'Return to StartRowIndex of 0 and rebind data
    StartRowIndex = 0
    Products.DataBind()
End Sub
Protected Sub PrevPage_Click(sender As Object, e As EventArgs) _
    Handles PrevPage.Click
    'Subtract MaximumRows from StartRowIndex and rebind data
    StartRowIndex -= MaximumRows
    Products.DataBind()
End Sub
Protected Sub NextPage_Click(sender As Object, e As EventArgs) _
    Handles NextPage.Click
    'Add MaximumRows to StartRowIndex and rebind data
    StartRowIndex += MaximumRows
    Products.DataBind()
End Sub
Protected Sub LastPage_Click(sender As Object, e As EventArgs) _
    Handles LastPage.Click
    'Set StartRowIndex = to last page's starting row index and rebind data
    StartRowIndex = ((TotalRowCount - 1) \ MaximumRows) * MaximumRows
    Products.DataBind()
End Sub
```

Finally, we need to disable the First and Previous buttons in the paging interface when viewing the first page of data and the Next and Last buttons when viewing the last page. To accomplish this, add the following code to the ObjectDataSource s `Selecting` event handler:

```
' Disable the paging interface buttons, if needed
FirstPage.Enabled = StartRowIndex <> 0
PrevPage.Enabled = StartRowIndex <> 0
Dim LastPageStartRowIndex As Integer = _
    ((TotalRowCount - 1) \ MaximumRows) * MaximumRows
NextPage.Enabled = (StartRowIndex < LastPageStartRowIndex)
LastPage.Enabled = (StartRowIndex < LastPageStartRowIndex)
```

After adding these `Click` event handlers and the code to enable or disable the paging interface elements based on the current start row index, test the page in a browser. As Figure 15 illustrates, when first visiting the page the First and Previous buttons will are disabled. Clicking Next shows the second page of data, while clicking Last displays the final page (see Figures 16 and 17). When viewing the last page of data both the Next and Last buttons are disabled.

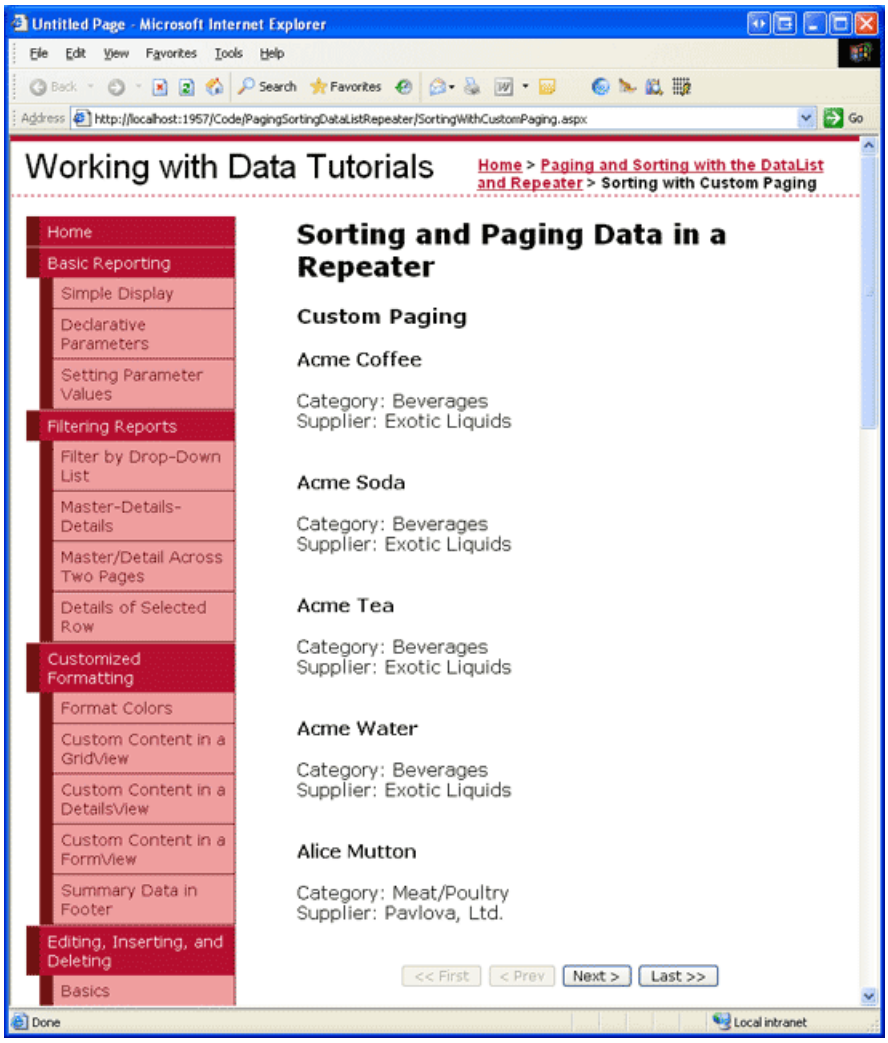


Figure 15: The Previous and Last Buttons are Disabled When Viewing the First Page of Products (Click to view full-size image)

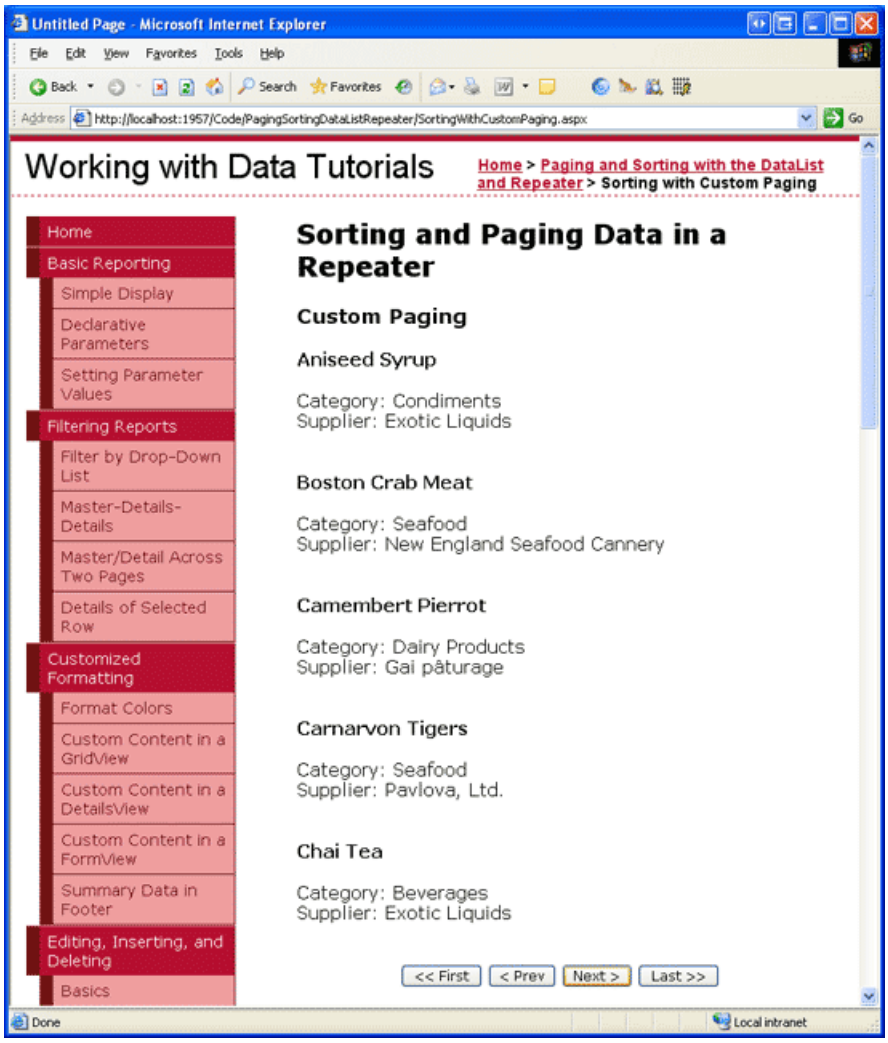


Figure 16: The Second Page of Products are Displayed (Click to view full-size image)

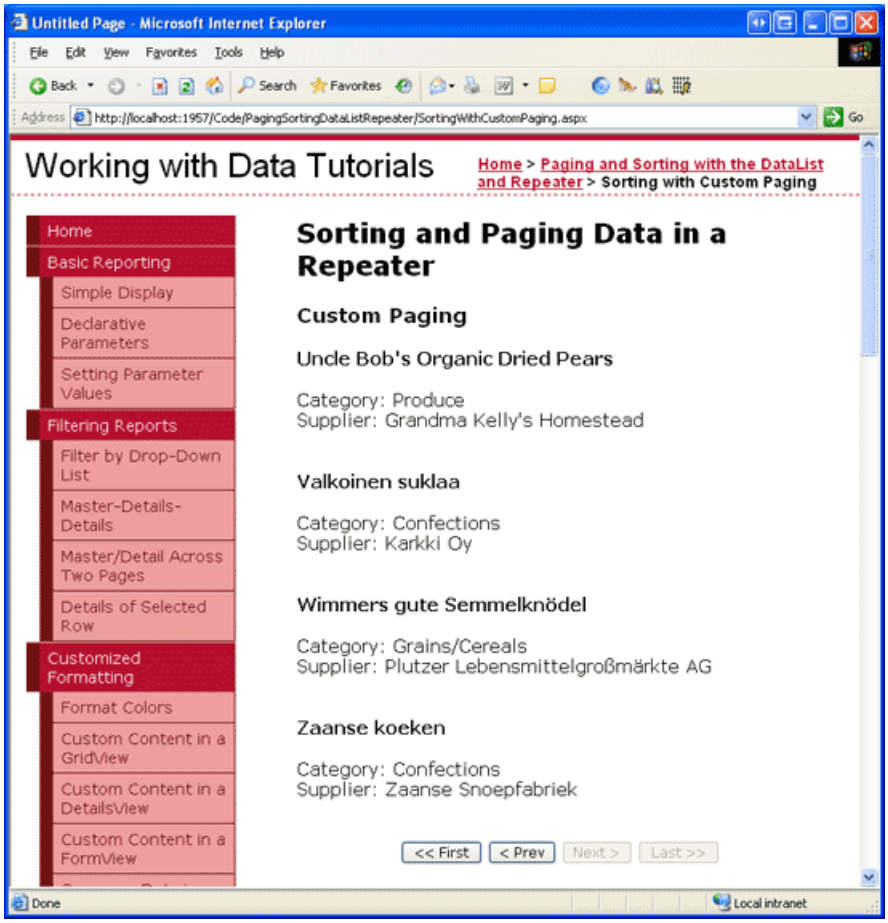


Figure 17: Clicking Last Displays the Final Page of Data (Click to view full-size image)

Step 7: Including Sorting Support with the Custom Paged Repeater

Now that custom paging has been implemented, we re ready to include sorting support. The `ProductsBLL` class s `GetProductsPagedAndSorted` method has the same *startRowIndex* and *maximumRows* input parameters as `GetProductsPaged`, but permits an additional *sortExpression* input parameter. To use the `GetProductsPagedAndSorted` method from `SortingWithCustomPaging.aspx`, we need to perform the following steps:

1. Change the ObjectDataSource s `SelectMethod` property from `GetProductsPaged` to `GetProductsPagedAndSorted`.
2. Add a *sortExpression* `Parameter` object to the ObjectDataSource s `SelectParameters` collection.
3. Create a private, page-level `SortExpression` property that persists its value across postbacks through the page s view state.
4. Update the ObjectDataSource s `Selecting` event handler to assign the ObjectDataSource s *sortExpression* parameter the value of the page-level `SortExpression` property.
5. Create the sorting interface.

Start by updating the ObjectDataSource s `SelectMethod` property and adding a *sortExpression* `Parameter`. Make sure that the *sortExpression* `Parameter` s `Type` property is set to `String`. After completing these first two tasks, the ObjectDataSource s declarative markup should look like the following:

```
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}" TypeName="ProductsBLL"
    SelectMethod="GetProductsPagedAndSorted"
    OnSelecting="ProductsDataSource_Selecting">
    <SelectParameters>
        <asp:Parameter Name="sortExpression" Type="String" />
        <asp:Parameter Name="startRowIndex" Type="Int32" />
        <asp:Parameter Name="maximumRows" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

Next, we need a page-level `SortExpression` property whose value is serialized to view state. If no sort expression value has been set, use ProductName as the default:

```
Private Property SortExpression() As String
    Get
        Dim o As Object = ViewState("SortExpression")
        If o Is Nothing Then
            Return "ProductName"
        Else
            Return o.ToString()
        End If
    End Get
    Set(ByVal value As String)
        ViewState("SortExpression") = value
    End Set
End Property
```

Before the ObjectDataSource invokes the `GetProductsPagedAndSorted` method we need to set the *sortExpression* `Parameter` to the value of the `SortExpression` property. In the `Selecting` event handler, add the following line of code:

```
e.InputParameters("sortExpression") = SortExpression
```

All that remains is to implement the sorting interface. As we did in the last example, let s have the sorting interface

implemented using three Button Web controls that allow the user to sort the results by product name, category, or supplier.

```
<asp:Button runat="server" id="SortByProductName"
    Text="Sort by Product Name" />
<asp:Button runat="server" id="SortByCategoryName"
    Text="Sort by Category" />
<asp:Button runat="server" id="SortBySupplierName"
    Text="Sort by Supplier" />
```

Create `Click` event handlers for these three Button controls. In the event handler, reset the `StartRowIndex` to 0, set the `SortExpression` to the appropriate value, and rebind the data to the Repeater:

```
Protected Sub SortByProductName_Click(sender As Object, e As EventArgs) _
    Handles SortByProductName.Click
    StartRowIndex = 0
    SortExpression = "ProductName"
    Products.DataBind()
End Sub
Protected Sub SortByCategoryName_Click(sender As Object, e As EventArgs) _
    Handles SortByCategoryName.Click
    StartRowIndex = 0
    SortExpression = "CategoryName"
    Products.DataBind()
End Sub
Protected Sub SortBySupplierName_Click(sender As Object, e As EventArgs) _
    Handles SortBySupplierName.Click
    StartRowIndex = 0
    SortExpression = "CompanyName"
    Products.DataBind()
End Sub
```

That s all there is to it! While there were a number of steps to get custom paging and sorting implemented, the steps were very similar to those needed for default paging. Figure 18 shows the products when viewing the last page of data when sorted by category.

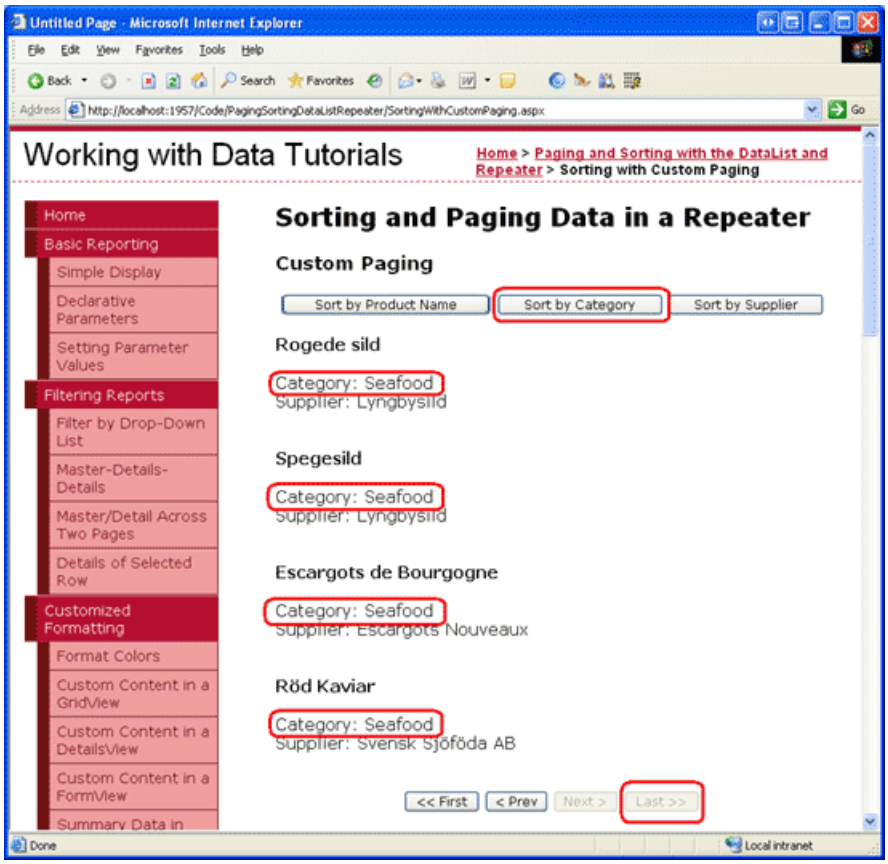


Figure 18: The Last Page of Data, Sorted by Category, is Displayed (Click to view full-size image)

> **NOTE**
>
> In previous examples, when sorting by the supplier SupplierName was used as the sort expression. However, for the custom paging implementation, we need to use CompanyName. This is because the stored procedure responsible for implementing custom paging `GetProductsPagedAndSorted` passes the sort expression into the `ROW_NUMBER()` keyword, The `ROW_NUMBER()` keyword requires the actual column name rather than an alias. Therefore, we must use `CompanyName` (the name of the column in the `Suppliers` table) rather than the alias used in the `SELECT` query (`SupplierName`) for the sort expression.

Summary

Neither the DataList nor Repeater offer built-in sorting support, but with a bit of code and a custom sorting interface, such functionality can be added. When implementing sorting, but not paging, the sort expression can be specified through the `DataSourceSelectArguments` object passed into the ObjectDataSource s `Select` method. This `DataSourceSelectArguments` object s `SortExpression` property can be assigned in the ObjectDataSource s `Selecting` event handler.

To add sorting capabilities to a DataList or Repeater that already provides paging support, the easiest approach is to customize the Business Logic Layer to include a method that accepts a sort expression. This information can then be passed in through a parameter in the ObjectDataSource s `SelectParameters`.

This tutorial completes our examination of paging and sorting with the DataList and Repeater controls. Our next and final tutorial will examine how to add Button Web controls to the DataList and Repeater s templates in order to provide some custom, user-initiated functionality on a per-item basis.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was David Suru. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Custom Button Actions with the DataList and Repeater

2/20/2020 • 2 minutes to read • Edit Online

> These tutorials show how to add buttons to the DataList and Repeater controls to add custom functionality.

- Custom Buttons in the DataList and Repeater (C#)
- Custom Buttons in the DataList and Repeater (VB)

Custom Buttons in the DataList and Repeater (C#)

2/20/2020 • 9 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll build an interface that uses a Repeater to list the categories in the system, with each category providing a button to show its associated products using a BulletedList control.

Introduction

Throughout the past seventeen DataList and Repeater tutorials, we ve created both read-only examples and editing and deleting examples. To facilitate editing and deleting capabilities within a DataList, we added buttons to the DataList s `ItemTemplate` that, when clicked, caused a postback and raised a DataList event corresponding to the button s `CommandName` property. For example, adding a button to the `ItemTemplate` with a `CommandName` property value of Edit causes the DataList s `EditCommand` to fire on postback; one with the `CommandName` Delete raises the `DeleteCommand`.

In addition to Edit and Delete buttons, the DataList and Repeater controls can also include Buttons, LinkButtons, or ImageButtons that, when clicked, perform some custom server-side logic. In this tutorial we'll build an interface that uses a Repeater to list the categories in the system. For each category, the Repeater will include a button to show the category s associated products using a BulletedList control (see Figure 1).

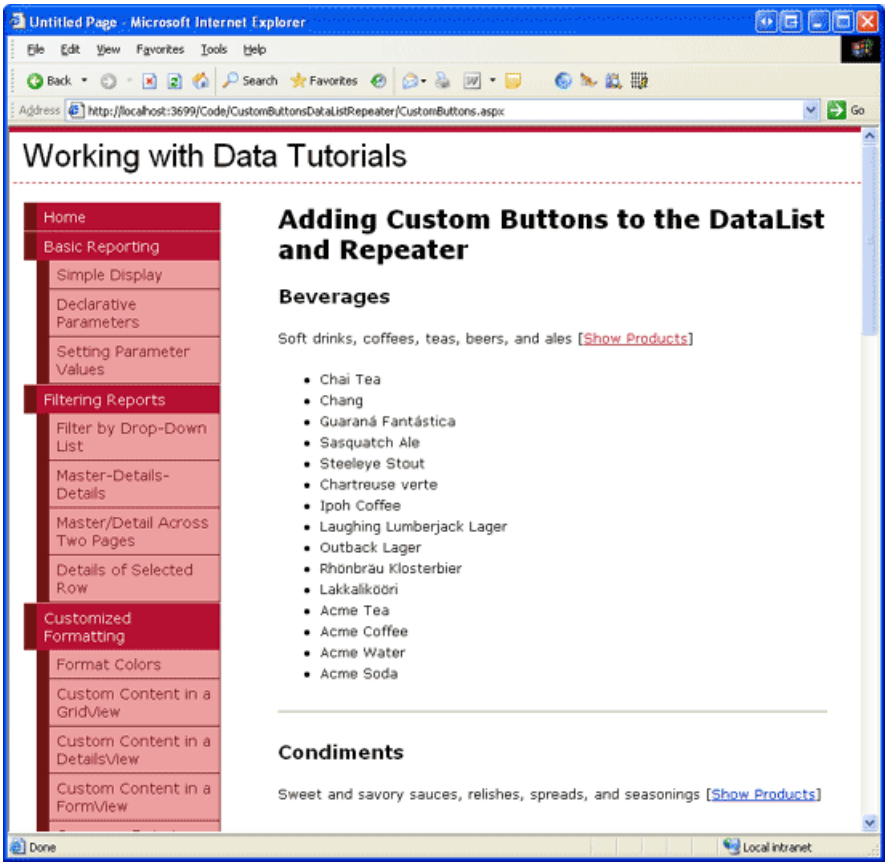


Figure 1: Clicking the Show Products Link Displays the Category s Products in a Bulleted List (Click to view full-size image)

Step 1: Adding the Custom Button Tutorial Web Pages

Before we look at how to add a custom button, let s first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial. Start by adding a new folder named `CustomButtonsDataListRepeater`. Next, add the following two ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `CustomButtons.aspx`

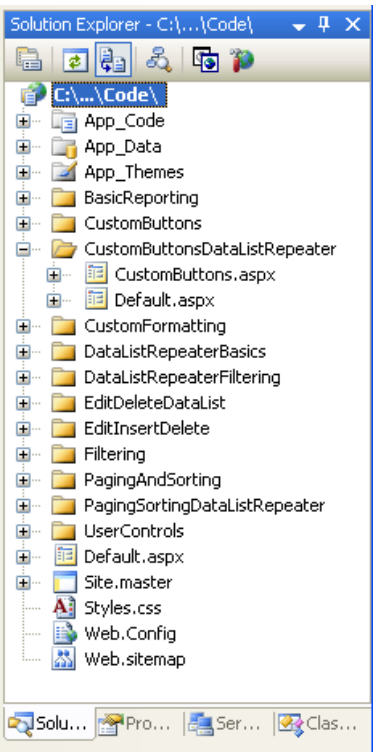


Figure 2: Add the ASP.NET Pages for the Custom Buttons-Related Tutorials

Like in the other folders, `Default.aspx` in the `CustomButtonsDataListRepeater` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

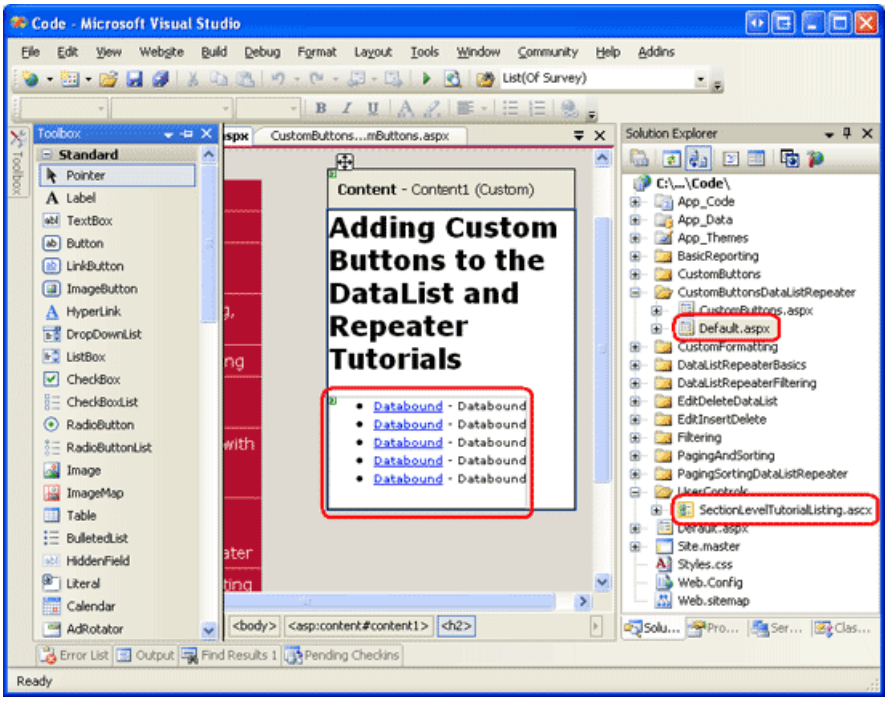


Figure 3: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add the pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Paging and Sorting with the DataList and Repeater `<siteMapNode>`:

```
<siteMapNode
    url="~/CustomButtonsDataListRepeater/Default.aspx"
    title="Adding Custom Buttons to the DataList and Repeater"
    description="Samples of DataList and Repeater Reports that Include
                 Buttons for Performing Server-Side Actions">
    <siteMapNode
        url="~/CustomButtonsDataListRepeater/CustomButtons.aspx"
        title="Using Custom Buttons in the DataList and Repeater's Templates"
        description="Examines how to add custom Buttons, LinkButtons,
                     or ImageButtons within templates." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the editing, inserting, and deleting tutorials.

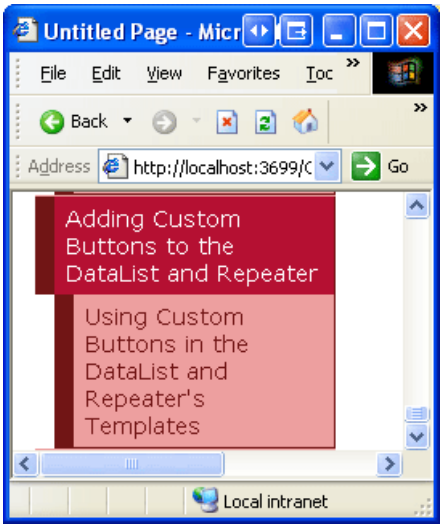


Figure 4: The Site Map Now Includes the Entry for the Custom Buttons Tutorial

Step 2: Adding the List of Categories

For this tutorial we need to create a Repeater that lists all categories along with a Show Products LinkButton that, when clicked, displays the associated category s products in a bulleted list. Let s first create a simple Repeater that lists the categories in the system. Start by opening the `CustomButtons.aspx` page in the `CustomButtonsDataListRepeater` folder. Drag a Repeater from the Toolbox onto the Designer and set its `ID` property to `Categories`. Next, create a new data source control from the Repeater s smart tag. Specifically, create a new ObjectDataSource control named `CategoriesDataSource` that selects its data from the `CategoriesBLL` class s `GetCategories()` method.

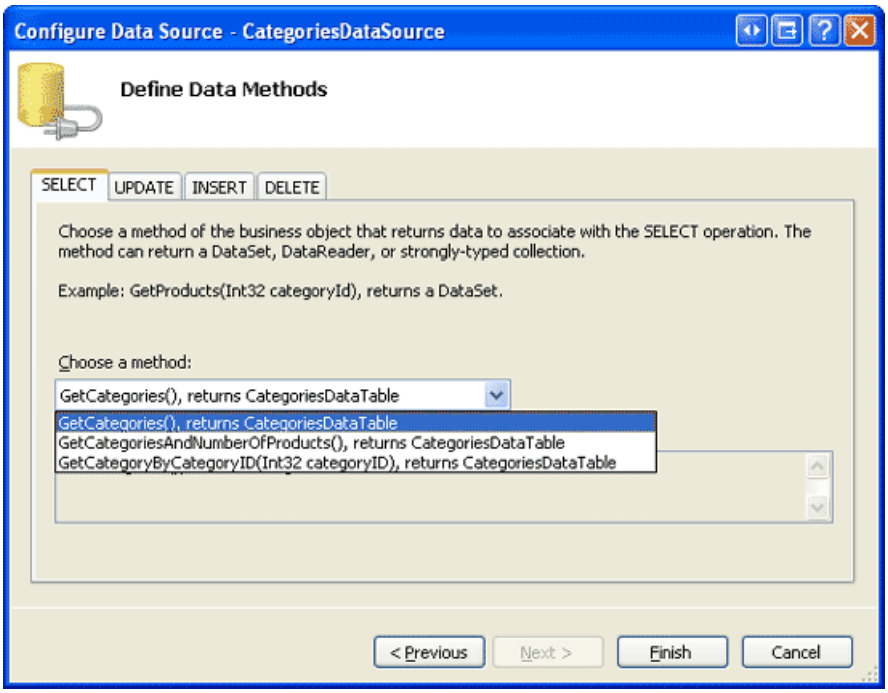


Figure 5: Configure the ObjectDataSource to Use the `CategoriesBLL` Class s `GetCategories()` Method (Click to view full-size image)

Unlike the DataList control, for which Visual Studio creates a default `ItemTemplate` based on the data source, the Repeater s templates must be manually defined. Furthermore, the Repeater s templates must be created and edited declaratively (that is, there s no Edit Templates option in the Repeater s smart tag).

Click on the Source tab in the bottom left corner and add an `ItemTemplate` that displays the category s name in an `<h3>` element and its description in a paragraph tag; include a `SeparatorTemplate` that displays a horizontal rule (`<hr />`) between each category. Also add a LinkButton with its `Text` property set to Show Products. After completing these steps, your page s declarative markup should look like the following:

```
<asp:Repeater ID="Categories" DataSourceID="CategoriesDataSource"
    runat="server">
    <ItemTemplate>
        <h3><%# Eval("CategoryName") %></h3>
        <p>
            <%# Eval("Description") %>
            [<asp:LinkButton runat="server" ID="ShowProducts">
                Show Products</asp:LinkButton>]
        </p>
    </ItemTemplate>
    <SeparatorTemplate><hr /></SeparatorTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

Figure 6 shows the page when viewed through a browser. Each category name and description is listed. The Show Products button, when clicked, causes a postback but does not yet perform any action.

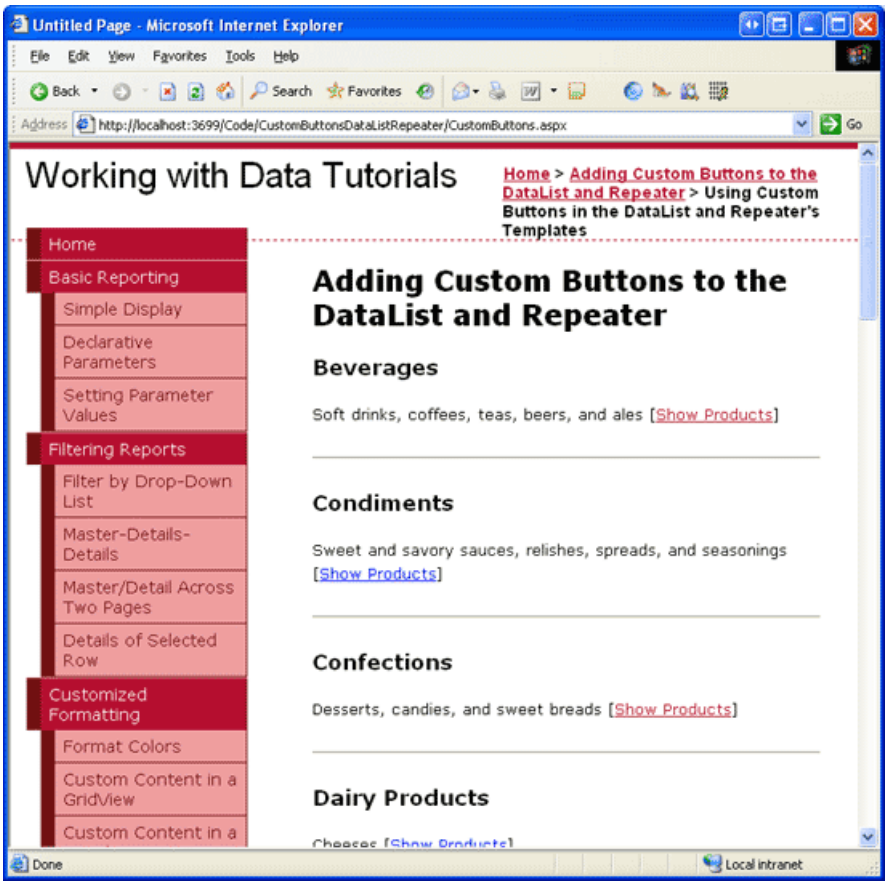


Figure 6: Each Category s Name and Description is Displayed, Along with a Show Products LinkButton (Click to view full-size image)

Step 3: Executing Server-Side Logic When the Show Products LinkButton is Clicked

Anytime a Button, LinkButton, or ImageButton within a template in a DataList or Repeater is clicked, a postback occurs and the DataList or Repeater s `ItemCommand` event fires. In addition to the `ItemCommand` event, the DataList control may also raise another, more specific event if the button s `CommandName` property is set to one of the reserved strings (Delete, Edit, Cancel, Update, or Select), but the `ItemCommand` event is *always* fired.

When a button is clicked within a DataList or Repeater, oftentimes we need to pass along which button was clicked (in the case that there may be multiple buttons within the control, such as both an Edit and Delete button) and perhaps some additional information (such as the primary key value of the item whose button was clicked). The Button, LinkButton, and ImageButton provide two properties whose values are passed to the `ItemCommand` event handler:

- `CommandName` a string typically used to identify each button in the template
- `CommandArgument` commonly used to hold the value of some data field, such as the primary key value

For this example, set the LinkButton s `CommandName` property to ShowProducts and bind the current record s primary key value `CategoryID` to the `CommandArgument` property using the databinding syntax `CategoryArgument='<%# Eval("CategoryID") %>'`. After specifying these two properties, the LinkButton s declarative syntax should look like the following:

```
<asp:LinkButton runat="server" CommandName="ShowProducts"
    CommandArgument='<%# Eval("CategoryID") %>' ID="ShowProducts">
    Show Products</asp:LinkButton>
```

When the button is clicked, a postback occurs and the DataList or Repeater s `ItemCommand` event fires. The event

handler is passed the button s `CommandName` and `CommandArgument` values.

Create an event handler for the Repeater s `ItemCommand` event and note the second parameter passed into the event handler (named `e`). This second parameter is of type `RepeaterCommandEventArgs` and has the following four properties:

- `CommandArgument` the value of the clicked button s `CommandArgument` property
- `CommandName` the value of the button s `CommandName` property
- `CommandSource` a reference to the button control that was clicked
- `Item` a reference to the `RepeaterItem` that contains the button that was clicked; each record bound to the Repeater is manifested as a `RepeaterItem`

Since the selected category s `CategoryID` is passed in via the `CommandArgument` property, we can get the set of products associated with the selected category in the `ItemCommand` event handler. These products can then be bound to a BulletedList control in the `ItemTemplate` (which we ve yet to add). All that remains, then, is to add the BulletedList, reference it in the `ItemCommand` event handler, and bind to it the set of products for the selected category, which we'll tackle in Step 4.

> **NOTE**
> The DataList s `ItemCommand` event handler is passed an object of type `DataListCommandEventArgs`, which offers the same four properties as the `RepeaterCommandEventArgs` class.

Step 4: Displaying the Selected Category s Products in a Bulleted List

The selected category s products can be displayed within the Repeater s `ItemTemplate` using any number of controls. We could add another nested Repeater, a DataList, a DropDownList, a GridView, and so on. Since we want to display the products as a bulleted list, though, we'll use the BulletedList control. Returning to the `CustomButtons.aspx` page s declarative markup, add a BulletedList control to the `ItemTemplate` after the Show Products LinkButton. Set the BulletedLists s `ID` to `ProductsInCategory`. The BulletedList displays the value of the data field specified via the `DataTextField` property; since this control will have product information bound to it, set the `DataTextField` property to `ProductName`.

```
<asp:BulletedList ID="ProductsInCategory" DataTextField="ProductName"
    runat="server"></asp:BulletedList>
```

In the `ItemCommand` event handler, reference this control using `e.Item.FindControl("ProductsInCategory")` and bind it to the set of products associated with the selected category.

```
protected void Categories_ItemCommand(object source, RepeaterCommandEventArgs e)
{
    if (e.CommandName == "ShowProducts")
    {
        // Determine the CategoryID
        int categoryID = Convert.ToInt32(e.CommandArgument);
        // Get the associated products from the ProudctsBLL and bind
        // them to the BulletedList
        BulletedList products =
            (BulletedList)e.Item.FindControl("ProductsInCategory");
        ProductsBLL productsAPI = new ProductsBLL();
        products.DataSource =
            productsAPI.GetProductsByCategoryID(categoryID);
        products.DataBind());
    }
}
```

Before performing any action in the `ItemCommand` event handler, it s prudent to first check the value of the incoming `CommandName`. Since the `ItemCommand` event handler fires when *any* button is clicked, if there are multiple buttons in the template use the `CommandName` value to discern what action to take. Checking the `CommandName` here is moot, since we only have a single button, but it is a good habit to form. Next, the `CategoryID` of the selected category is retrieved from the `CommandArgument` property. The BulletedList control in the template is then referenced and bound to the results of the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method.

In previous tutorials that used the buttons within a DataList, such as An Overview of Editing and Deleting Data in the DataList, we determined the primary key value of a given item via the `DataKeys` collection. While this approach works well with the DataList, the Repeater does not have a `DataKeys` property. Instead, we must use an alternative approach for supplying the primary key value, such as through the button s `CommandArgument` property or by assigning the primary key value to a hidden Label Web control within the template and reading its value back in the `ItemCommand` event handler using `e.Item.FindControl("LabelID")`.

After completing the `ItemCommand` event handler, take a moment to test out this page in a browser. As Figure 7 shows, clicking the Show Products link causes a postback and displays the products for the selected category in a BulletedList. Furthermore, note that this product information remains, even if other categories Show Products links are clicked.

NOTE

If you want to modify the behavior of this report, such that the only one category s products are listed at a time, simply set the BulletedList control s `EnableViewState` property to `False`.

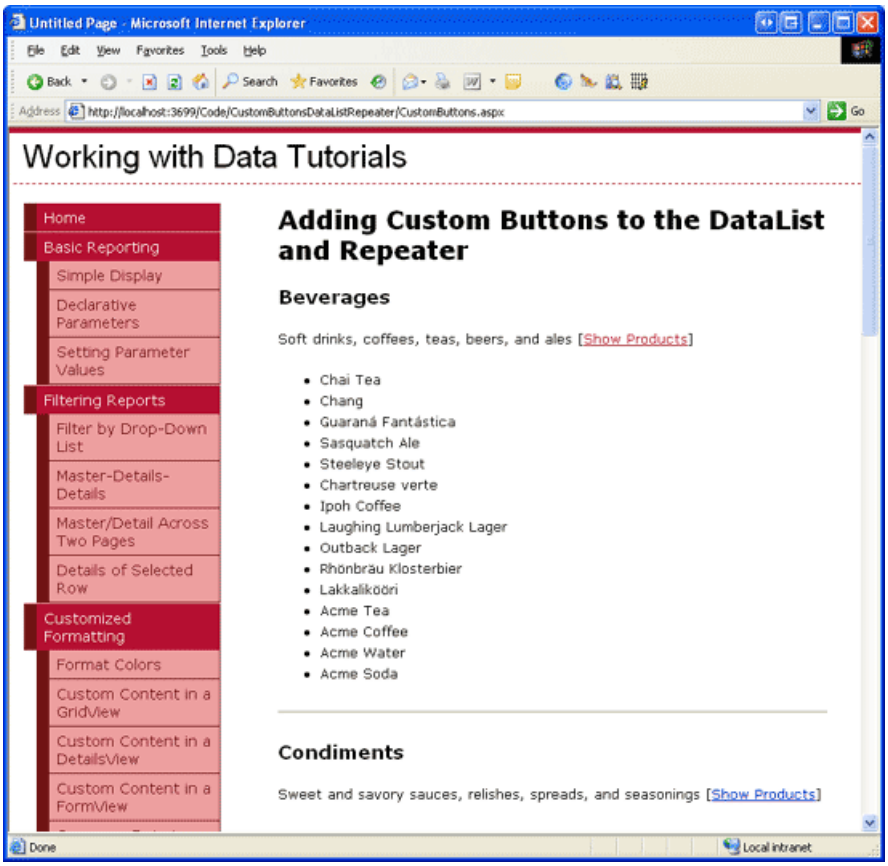


Figure 7: A BulletedList is used to Display the Products of the Selected Category (Click to view full-size image)

Summary

The DataList and Repeater controls can include any number of Buttons, LinkButtons, or ImageButtons within their templates. Such buttons, when clicked, cause a postback and raise the `ItemCommand` event. To associate custom server-side action with a button being clicked, create an event handler for the `ItemCommand` event. In this event handler first check the incoming `CommandName` value to determine which button was clicked. Additional information can optionally be supplied through the button s `CommandArgument` property.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Dennis Patterson. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Custom Buttons in the DataList and Repeater (VB)

2/20/2020 • 9 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we'll build an interface that uses a Repeater to list the categories in the system, with each category providing a button to show its associated products using a BulletedList control.

Introduction

Throughout the past seventeen DataList and Repeater tutorials, we ve created both read-only examples and editing and deleting examples. To facilitate editing and deleting capabilities within a DataList, we added buttons to the DataList s `ItemTemplate` that, when clicked, caused a postback and raised a DataList event corresponding to the button s `CommandName` property. For example, adding a button to the `ItemTemplate` with a `CommandName` property value of Edit causes the DataList s `EditCommand` to fire on postback; one with the `CommandName` Delete raises the `DeleteCommand`.

In addition to Edit and Delete buttons, the DataList and Repeater controls can also include Buttons, LinkButtons, or ImageButtons that, when clicked, perform some custom server-side logic. In this tutorial we'll build an interface that uses a Repeater to list the categories in the system. For each category, the Repeater will include a button to show the category s associated products using a BulletedList control (see Figure 1).

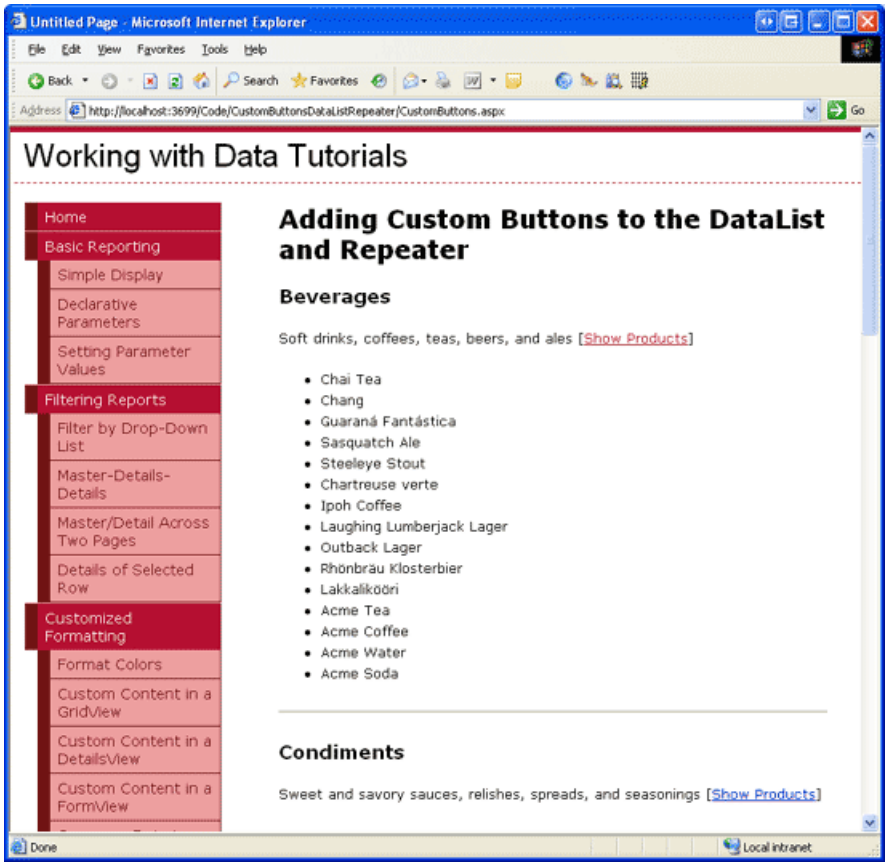


Figure 1: Clicking the Show Products Link Displays the Category s Products in a Bulleted List (Click to view full-size image)

Step 1: Adding the Custom Button Tutorial Web Pages

Before we look at how to add a custom button, let s first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial. Start by adding a new folder named `CustomButtonsDataListRepeater`. Next, add the following two ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `CustomButtons.aspx`

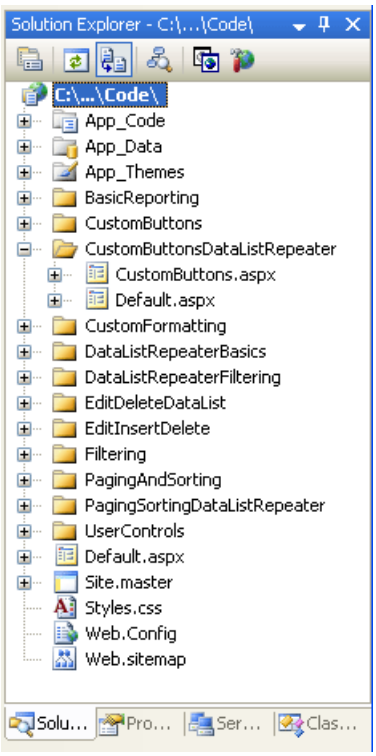


Figure 2: Add the ASP.NET Pages for the Custom Buttons-Related Tutorials

Like in the other folders, `Default.aspx` in the `CustomButtonsDataListRepeater` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

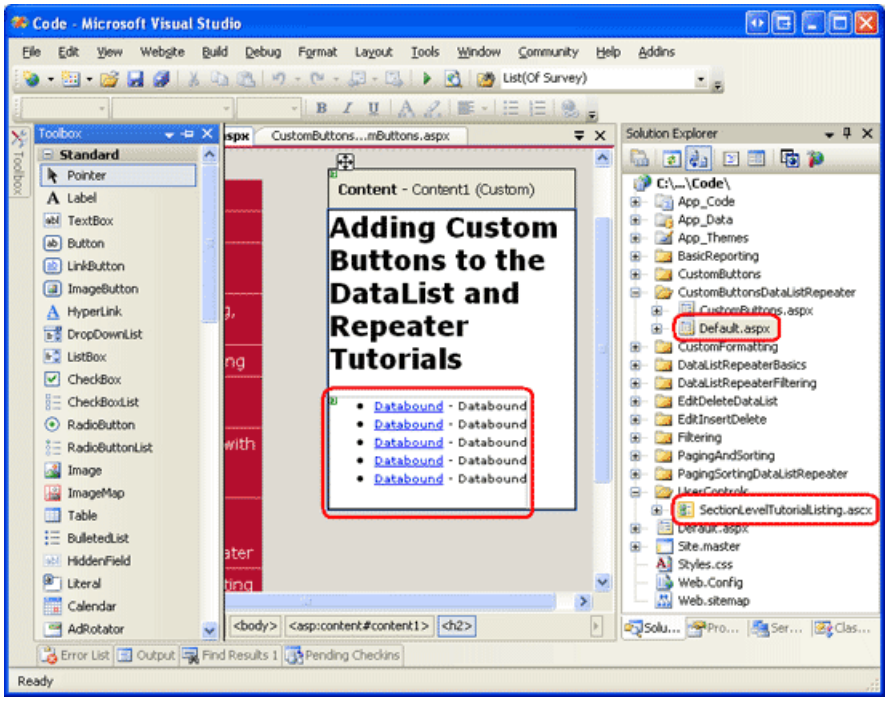


Figure 3: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add the pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Paging and Sorting with the DataList and Repeater `<siteMapNode>`:

```
<siteMapNode
    url="~/CustomButtonsDataListRepeater/Default.aspx"
    title="Adding Custom Buttons to the DataList and Repeater"
    description="Samples of DataList and Repeater Reports that Include
                 Buttons for Performing Server-Side Actions">
    <siteMapNode
        url="~/CustomButtonsDataListRepeater/CustomButtons.aspx"
        title="Using Custom Buttons in the DataList and Repeater's Templates"
        description="Examines how to add custom Buttons, LinkButtons,
                     or ImageButtons within templates." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the editing, inserting, and deleting tutorials.

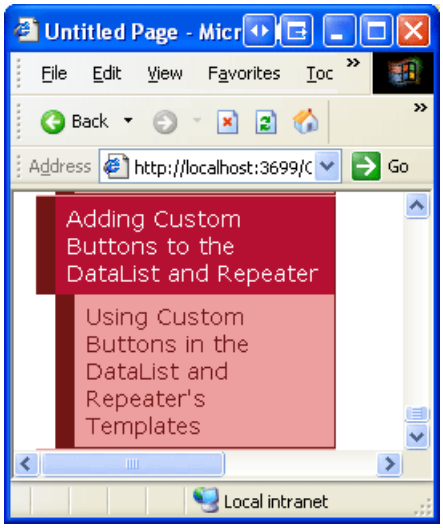


Figure 4: The Site Map Now Includes the Entry for the Custom Buttons Tutorial

Step 2: Adding the List of Categories

For this tutorial we need to create a Repeater that lists all categories along with a Show Products LinkButton that, when clicked, displays the associated category s products in a bulleted list. Let s first create a simple Repeater that lists the categories in the system. Start by opening the `CustomButtons.aspx` page in the `CustomButtonsDataListRepeater` folder. Drag a Repeater from the Toolbox onto the Designer and set its `ID` property to `Categories`. Next, create a new data source control from the Repeater s smart tag. Specifically, create a new ObjectDataSource control named `CategoriesDataSource` that selects its data from the `CategoriesBLL` class s `GetCategories()` method.

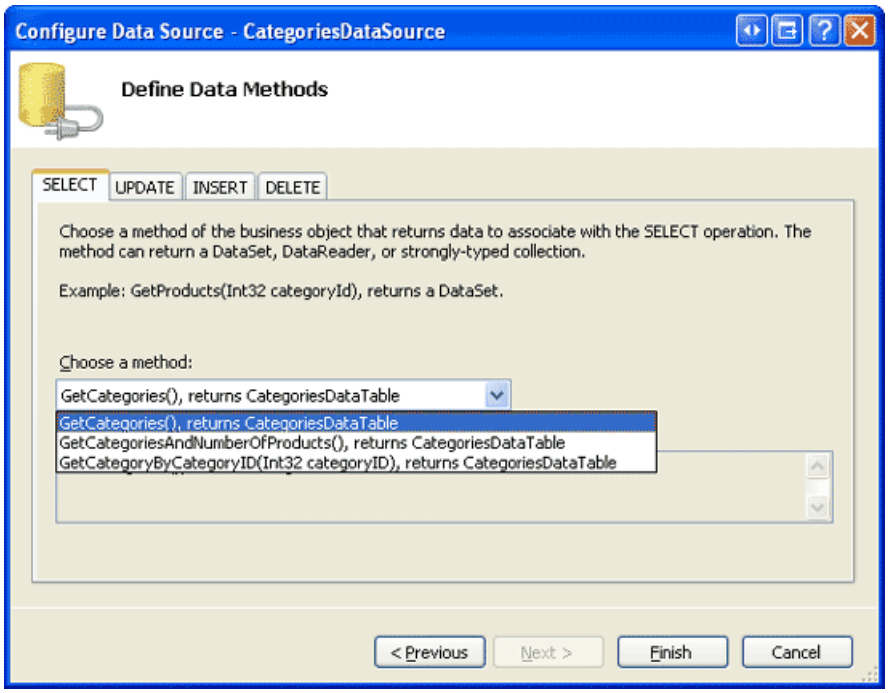


Figure 5: Configure the ObjectDataSource to Use the `CategoriesBLL` Class s `GetCategories()` Method (Click to view full-size image)

Unlike the DataList control, for which Visual Studio creates a default `ItemTemplate` based on the data source, the Repeater s templates must be manually defined. Furthermore, the Repeater s templates must be created and edited declaratively (that is, there s no Edit Templates option in the Repeater s smart tag).

Click on the Source tab in the bottom left corner and add an `ItemTemplate` that displays the category s name in an `<h3>` element and its description in a paragraph tag; include a `SeparatorTemplate` that displays a horizontal rule (`<hr />`) between each category. Also add a LinkButton with its `Text` property set to Show Products. After completing these steps, your page s declarative markup should look like the following:

```
<asp:Repeater ID="Categories" DataSourceID="CategoriesDataSource"
    runat="server">
    <ItemTemplate>
        <h3><%# Eval("CategoryName") %></h3>
        <p>
            <%# Eval("Description") %>
            [<asp:LinkButton runat="server" ID="ShowProducts">
                Show Products</asp:LinkButton>]
        </p>
    </ItemTemplate>
    <SeparatorTemplate><hr /></SeparatorTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

Figure 6 shows the page when viewed through a browser. Each category name and description is listed. The Show Products button, when clicked, causes a postback but does not yet perform any action.

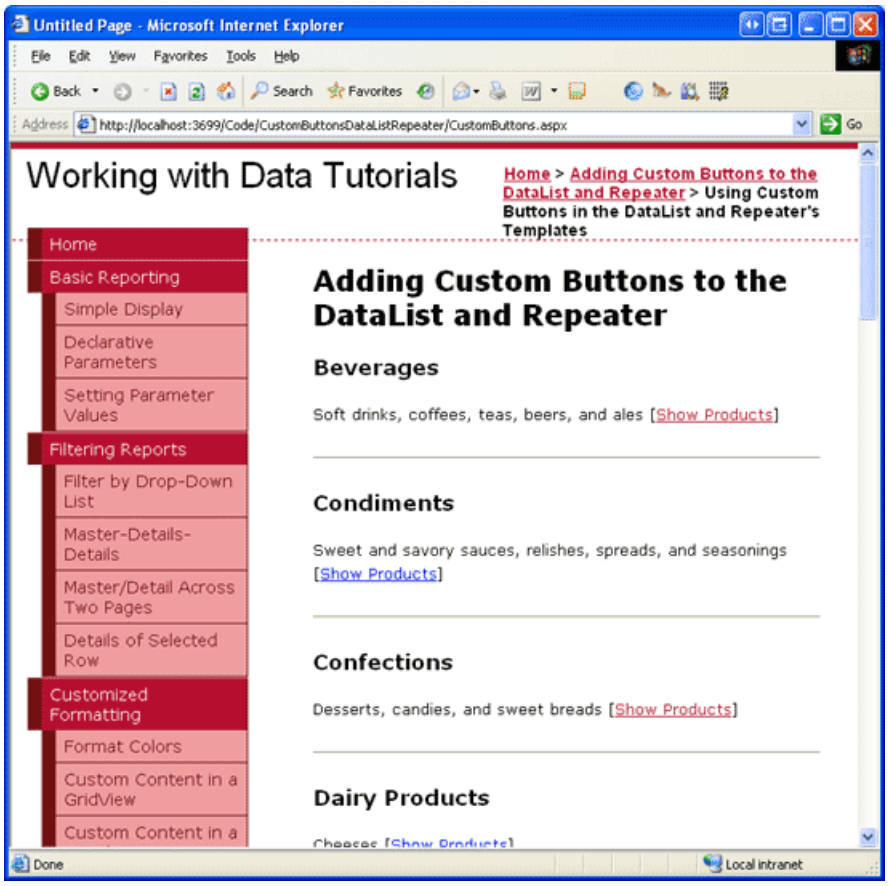


Figure 6: Each Category s Name and Description is Displayed, Along with a Show Products LinkButton (Click to view full-size image)

Step 3: Executing Server-Side Logic When the Show Products LinkButton is Clicked

Anytime a Button, LinkButton, or ImageButton within a template in a DataList or Repeater is clicked, a postback occurs and the DataList or Repeater s `ItemCommand` event fires. In addition to the `ItemCommand` event, the DataList control may also raise another, more specific event if the button s `CommandName` property is set to one of the reserved strings (Delete, Edit, Cancel, Update, or Select), but the `ItemCommand` event is *always* fired.

When a button is clicked within a DataList or Repeater, oftentimes we need to pass along which button was clicked (in the case that there may be multiple buttons within the control, such as both an Edit and Delete button) and perhaps some additional information (such as the primary key value of the item whose button was clicked). The Button, LinkButton, and ImageButton provide two properties whose values are passed to the `ItemCommand` event handler:

- `CommandName` a string typically used to identify each button in the template
- `CommandArgument` commonly used to hold the value of some data field, such as the primary key value

For this example, set the LinkButton s `CommandName` property to ShowProducts and bind the current record s primary key value `CategoryID` to the `CommandArgument` property using the databinding syntax `CategoryArgument='<%# Eval("CategoryID") %>'`. After specifying these two properties, the LinkButton s declarative syntax should look like the following:

```
<asp:LinkButton runat="server" CommandName="ShowProducts"
    CommandArgument='<%# Eval("CategoryID") %>' ID="ShowProducts">
    Show Products</asp:LinkButton>
```

When the button is clicked, a postback occurs and the DataList or Repeater s `ItemCommand` event fires. The event

handler is passed the button s `CommandName` and `CommandArgument` values.

Create an event handler for the Repeater s `ItemCommand` event and note the second parameter passed into the event handler (named `e`). This second parameter is of type `RepeaterCommandEventArgs` and has the following four properties:

- `CommandArgument` the value of the clicked button s `CommandArgument` property
- `CommandName` the value of the button s `CommandName` property
- `CommandSource` a reference to the button control that was clicked
- `Item` a reference to the `RepeaterItem` that contains the button that was clicked; each record bound to the Repeater is manifested as a `RepeaterItem`

Since the selected category s `CategoryID` is passed in via the `CommandArgument` property, we can get the set of products associated with the selected category in the `ItemCommand` event handler. These products can then be bound to a BulletedList control in the `ItemTemplate` (which we ve yet to add). All that remains, then, is to add the BulletedList, reference it in the `ItemCommand` event handler, and bind to it the set of products for the selected category, which we'll tackle in Step 4.

> **NOTE**
> The DataList s `ItemCommand` event handler is passed an object of type `DataListCommandEventArgs`, which offers the same four properties as the `RepeaterCommandEventArgs` class.

Step 4: Displaying the Selected Category s Products in a Bulleted List

The selected category s products can be displayed within the Repeater s `ItemTemplate` using any number of controls. We could add another nested Repeater, a DataList, a DropDownList, a GridView, and so on. Since we want to display the products as a bulleted list, though, we'll use the BulletedList control. Returning to the `CustomButtons.aspx` page s declarative markup, add a BulletedList control to the `ItemTemplate` after the Show Products LinkButton. Set the BulletedLists s `ID` to `ProductsInCategory`. The BulletedList displays the value of the data field specified via the `DataTextField` property; since this control will have product information bound to it, set the `DataTextField` property to `ProductName`.

```
<asp:BulletedList ID="ProductsInCategory" DataTextField="ProductName"
    runat="server"></asp:BulletedList>
```

In the `ItemCommand` event handler, reference this control using `e.Item.FindControl("ProductsInCategory")` and bind it to the set of products associated with the selected category.

```
Protected Sub Categories_ItemCommand _
    (source As Object, e As RepeaterCommandEventArgs) _
    Handles Categories.ItemCommand
    If e.CommandName = "ShowProducts" Then
        ' Determine the CategoryID
        Dim categoryID As Integer = Convert.ToInt32(e.CommandArgument)
        ' Get the associated products from the ProudctsBLL and
        ' bind them to the BulletedList
        Dim products As BulletedList = _
            CType(e.Item.FindControl("ProductsInCategory"), BulletedList)
        Dim productsAPI As New ProductsBLL()
        products.DataSource = productsAPI.GetProductsByCategoryID(categoryID)
        products.DataBind()
    End If
End Sub
```

Before performing any action in the `ItemCommand` event handler, it s prudent to first check the value of the incoming `CommandName`. Since the `ItemCommand` event handler fires when *any* button is clicked, if there are multiple buttons in the template use the `CommandName` value to discern what action to take. Checking the `CommandName` here is moot, since we only have a single button, but it is a good habit to form. Next, the `CategoryID` of the selected category is retrieved from the `CommandArgument` property. The BulletedList control in the template is then referenced and bound to the results of the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method.

In previous tutorials that used the buttons within a DataList, such as An Overview of Editing and Deleting Data in the DataList, we determined the primary key value of a given item via the `DataKeys` collection. While this approach works well with the DataList, the Repeater does not have a `DataKeys` property. Instead, we must use an alternative approach for supplying the primary key value, such as through the button s `CommandArgument` property or by assigning the primary key value to a hidden Label Web control within the template and reading its value back in the `ItemCommand` event handler using `e.Item.FindControl("LabelID")`.

After completing the `ItemCommand` event handler, take a moment to test out this page in a browser. As Figure 7 shows, clicking the Show Products link causes a postback and displays the products for the selected category in a BulletedList. Furthermore, note that this product information remains, even if other categories Show Products links are clicked.

> **NOTE**
>
> If you want to modify the behavior of this report, such that the only one category s products are listed at a time, simply set the BulletedList control s `EnableViewState` property to `False`.

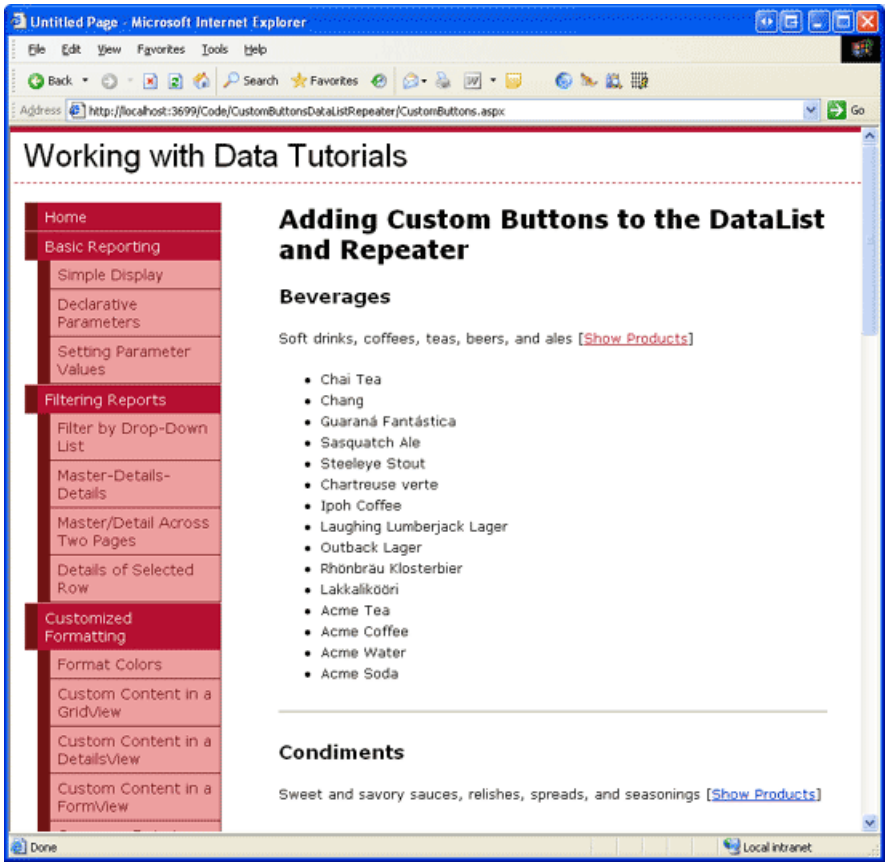


Figure 7: A BulletedList is used to Display the Products of the Selected Category (Click to view full-size image)

Summary

The DataList and Repeater controls can include any number of Buttons, LinkButtons, or ImageButtons within their templates. Such buttons, when clicked, cause a postback and raise the `ItemCommand` event. To associate custom

server-side action with a button being clicked, create an event handler for the `ItemCommand` event. In this event handler first check the incoming `CommandName` value to determine which button was clicked. Additional information can optionally be supplied through the button s `CommandArgument` property.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Dennis Patterson. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Accessing the Database Directly from an ASP.NET Page

2/20/2020 • 2 minutes to read • Edit Online

These tutorials show how to use the SqlDataSource control to query the database directly when you don't need a strict separation of the presentation and data access layers.

- Querying Data with the SqlDataSource Control (C#)
- Using Parameterized Queries with the SqlDataSource (C#)
- Inserting, Updating, and Deleting Data with the SqlDataSource (C#)
- Implementing Optimistic Concurrency with the SqlDataSource (C#)
- Querying Data with the SqlDataSource Control (VB)
- Using Parameterized Queries with the SqlDataSource (VB)
- Inserting, Updating, and Deleting Data with the SqlDataSource (VB)
- Implementing Optimistic Concurrency with the SqlDataSource (VB)

Querying Data with the SqlDataSource Control (C#)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In the preceding tutorials we used the ObjectDataSource control to fully separate the presentation layer from the Data Access layer. Starting with this tutorial, we learn how the SqlDataSource control can be used for simple applications that do not require such a strict separation of presentation and data access.

Introduction

All of the tutorials we ve examined so far have used a tiered architecture consisting of presentation, Business Logic, and Data Access layers. The Data Access Layer (DAL) was crafted in the first tutorial (Creating a Data Access Layer) and the Business Logic Layer in the second (Creating a Business Logic Layer). Starting with the Displaying Data With the ObjectDataSource tutorial, we saw how to use ASP.NET 2.0 s new ObjectDataSource control to declaratively interface with the architecture from the presentation layer.

While all of the tutorials so far have used the architecture to work with data, it is also possible to access, insert, update, and delete database data directly from an ASP.NET page, bypassing the architecture. Doing so places the specific database queries and business logic directly in the web page. For sufficiently large or complex applications, designing, implementing, and using a tiered architecture is vitally important for the success, updatability, and maintainability of the application. Developing a robust architecture, however, may be unnecessary when creating exceedingly simple, one-off applications.

ASP.NET 2.0 provides five built-in data source controls SqlDataSource, AccessDataSource, ObjectDataSource, XmlDataSource, and SiteMapDataSource. The SqlDataSource can be used to access and modify data directly from a relational database, including Microsoft SQL Server, Microsoft Access, Oracle, MySQL, and others. In this tutorial and the next three, we'll examine how to work with the SqlDataSource control, exploring how to query and filter database data, as well as how to use the SqlDataSource to insert, update, and delete data.

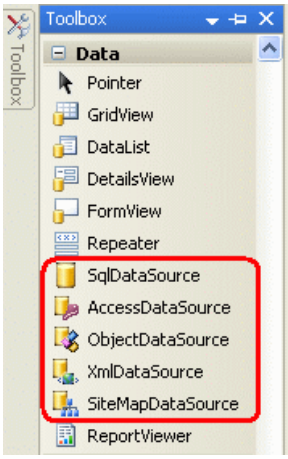


Figure 1: ASP.NET 2.0 Includes Five Built-In Data Source Controls

Comparing the ObjectDataSource and SqlDataSource

Conceptually, both the ObjectDataSource and SqlDataSource controls are simply proxies to data. As discussed in the Displaying Data With the ObjectDataSource tutorial, the ObjectDataSource has properties that indicate the

object type that provides the data and the methods to invoke to select, insert, update, and delete data from the underlying object type. Once the ObjectDataSource s properties have been configured, a data Web control such as a GridView, DetailsView, or DataList can be bound to the control, using the ObjectDataSource s `Select()`, `Insert()`, `Delete()`, and `Update()` methods to interact with the underlying architecture.

The SqlDataSource provides the same functionality, but operates against a relational database rather than an object library. With the SqlDataSource, we must specify the database connection string and the ad-hoc SQL queries or stored procedures to execute to insert, update, delete, and retrieve data. The SqlDataSource s `Select()`, `Insert()`, `Update()`, and `Delete()` methods, when invoked, connect to the specified database and issue the appropriate SQL query. As the following diagram illustrates, the these methods do the grunt work of connecting to a database, issuing a query, and returning the results.

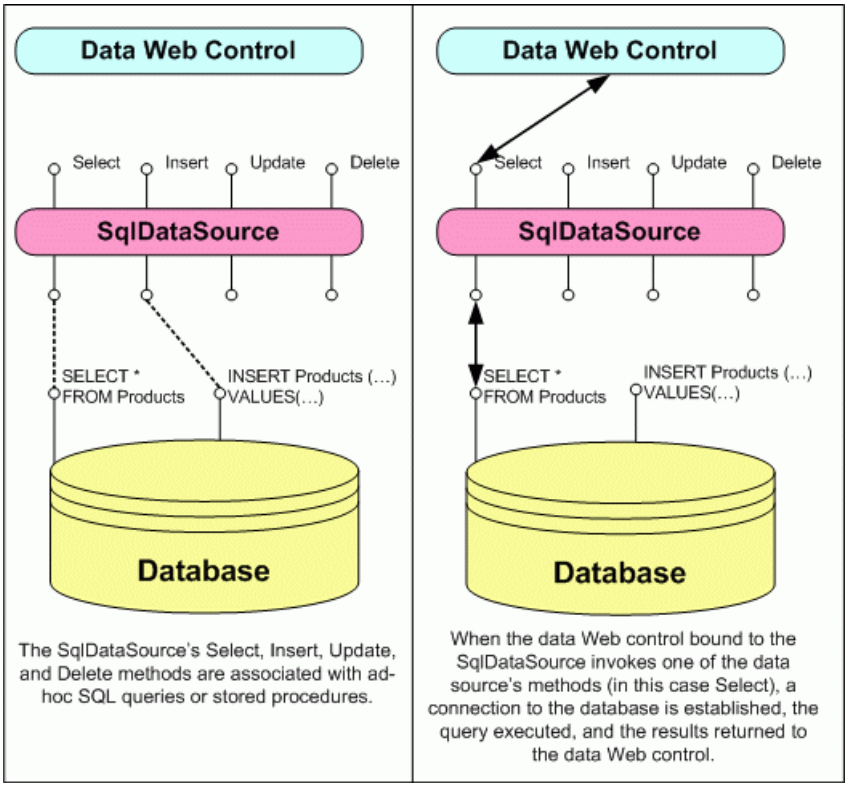


Figure 2: The SqlDataSource Serves as a Proxy to the Database

> **NOTE**
> In this tutorial we'll focus on retrieving data from the database. In the Inserting, Updating, and Deleting Data with the SqlDataSource Control tutorial, we'll see how to configure the SqlDataSource to support inserting, updating, and deleting.

The SqlDataSource and AccessDataSource Controls

In addition to the SqlDataSource control, ASP.NET 2.0 also includes an AccessDataSource control. These two different controls lead many developers new to ASP.NET 2.0 to suspect that the AccessDataSource control is designed to work exclusively with Microsoft Access with the SqlDataSource control designed to work exclusively with Microsoft SQL Server. While the AccessDataSource is designed to work specifically with Microsoft Access, the SqlDataSource control works with *any* relational database that can be accessed through .NET. This includes any OleDb- or ODBC-compliant data stores, such as Microsoft SQL Server, Microsoft Access, Oracle, Informix, MySQL, and PostgreSQL, among many others.

The sole difference between the AccessDataSource and SqlDataSource controls is how the database connection information is specified. The AccessDataSource control needs just the file path to the Access database file. The SqlDataSource, on the other hand, requires a complete connection string.

Step 1: Creating the SqlDataSource Web Pages

Before we start exploring how to work directly with database data using the SqlDataSource control, let s first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial and the next three. Start by adding a new folder named `SqlDataSource`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `Querying.aspx`
- `ParameterizedQueries.aspx`
- `InsertUpdateDelete.aspx`
- `OptimisticConcurrency.aspx`

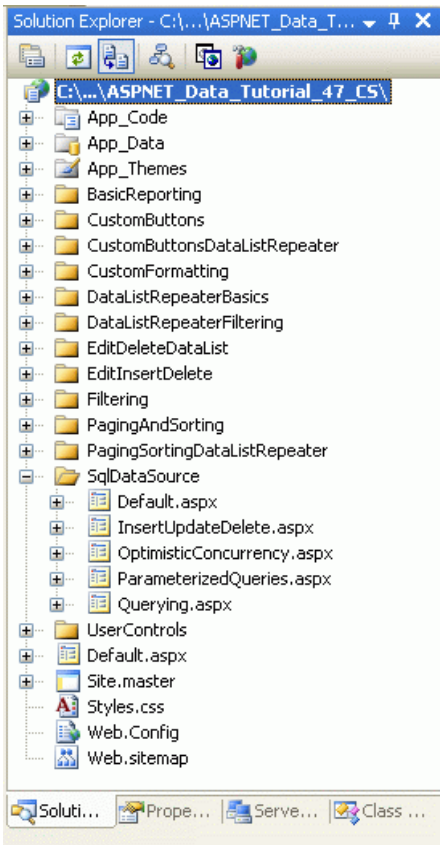


Figure 3: Add the ASP.NET Pages for the SqlDataSource-Related Tutorials

Like in the other folders, `Default.aspx` in the `SqlDataSource` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

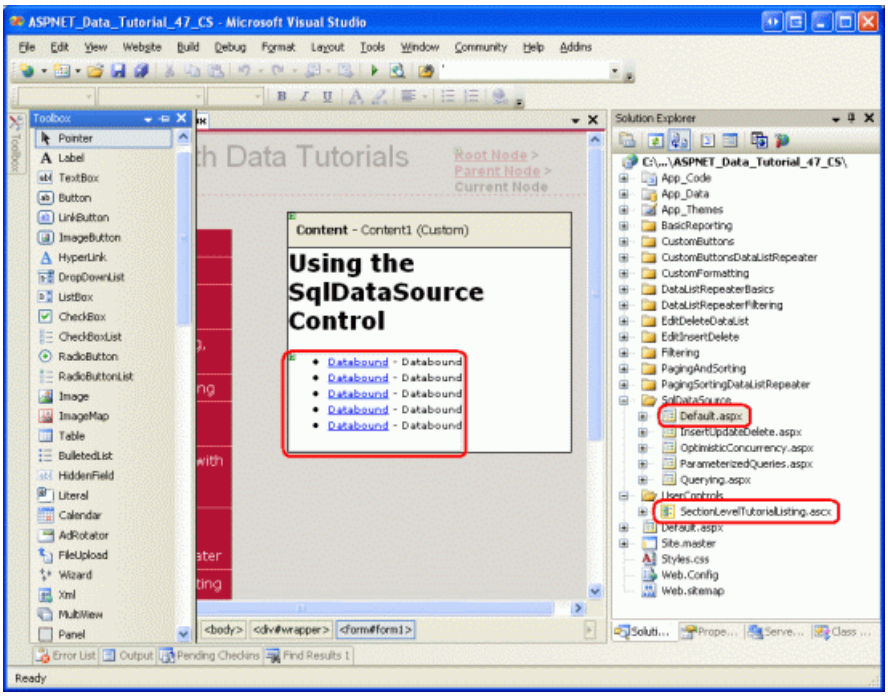


Figure 4: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add these four pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Adding Custom Buttons to the DataList and Repeater `<siteMapNode>`:

```
<siteMapNode url="~/SqlDataSource/Default.aspx"
    title="Using the SqlDataSource Control"
    description="Work directly with database data using the SqlDataSource control.">
    <siteMapNode url="~/SqlDataSource/Querying.aspx" title="Retrieving Database Data"
        description="Examines how to query data from a database that can then be
                     displayed  through a data Web control."/>
    <siteMapNode url="~/SqlDataSource/ParameterizedQueries.aspx"
        title="Parameterized Queries"
        description="Learn how to specify parameterized WHERE clauses in the
                     SqlDataSource's SELECT statement." />
    <siteMapNode url="~/SqlDataSource/InsertUpdateDelete.aspx"
        title="Inserting, Updating, and Deleting Database Data"
        description="See how to configure the SqlDataSource to include INSERT, UPDATE,
                      and DELETE statements." />
    <siteMapNode url="~/SqlDataSource/OptimisticConcurrency.aspx"
        title="Using Optimistic Concurrency"
        description="Explore how to augment the SqlDataSource to include support for
                     optimistic concurrency." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the editing, inserting, and deleting tutorials.

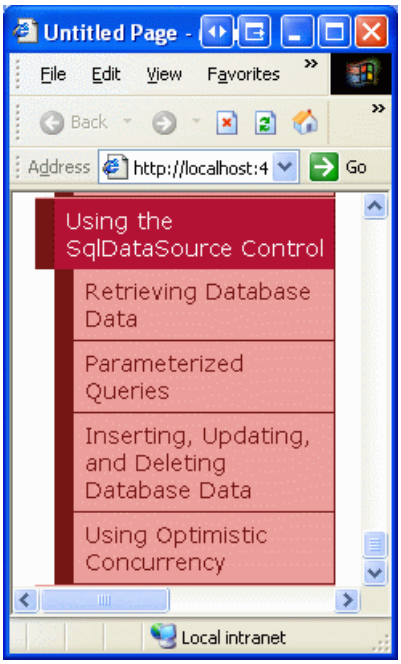


Figure 5: The Site Map Now Includes Entries for the SqlDataSource Tutorials

Step 2: Adding and Configuring the SqlDataSource Control

Start by opening the `Querying.aspx` page in the `SqlDataSource` folder and switch to Design view. Drag a SqlDataSource control from the Toolbox onto the Designer and set its `ID` to `ProductsDataSource`. As with the ObjectDataSource, the SqlDataSource does not produce any rendered output and therefore appears as a gray box on the design surface. To configure the SqlDataSource, click on the Configure Data Source link from the SqlDataSource s smart tag.

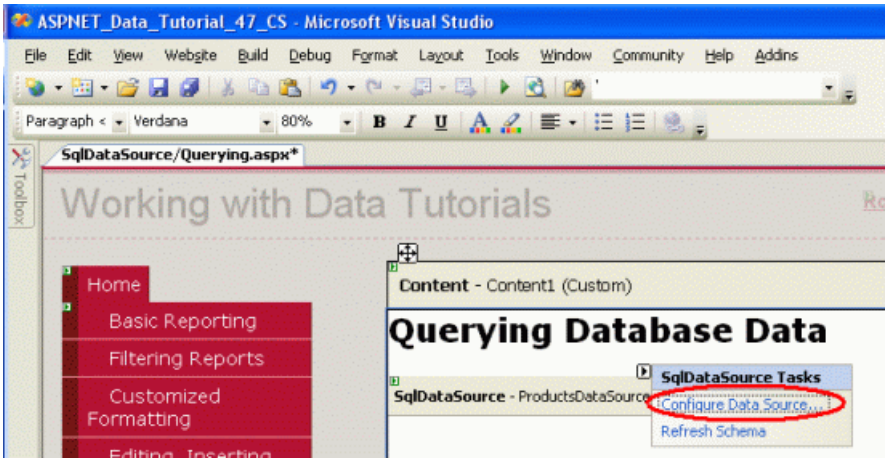


Figure 6: Click on the Configure Data Source Link from the SqlDataSource s Smart Tag

This brings up the SqlDataSource control s Configure Data Source wizard. While the wizard s steps differ from the ObjectDataSource control s, the end goal is the same to provide the details on how to retrieve, insert, update, and delete data through the data source. For the SqlDataSource this entails specifying the underlying database to use and providing the ad-hoc SQL statements or stored procedures.

The first wizard step prompts us for the database. The drop-down list includes those databases found in the web application s `App_Data` folder and those that have been added to the Data Connections node in the Server Explorer. Since we ve already added a connection string for the `NORTHWIND.MDF` database in the `App_Data` folder to our project s `Web.config` file, the drop-down list includes a reference to that connection string, `NORTHWINDConnectionString`. Choose this item from the drop-down list and click Next.

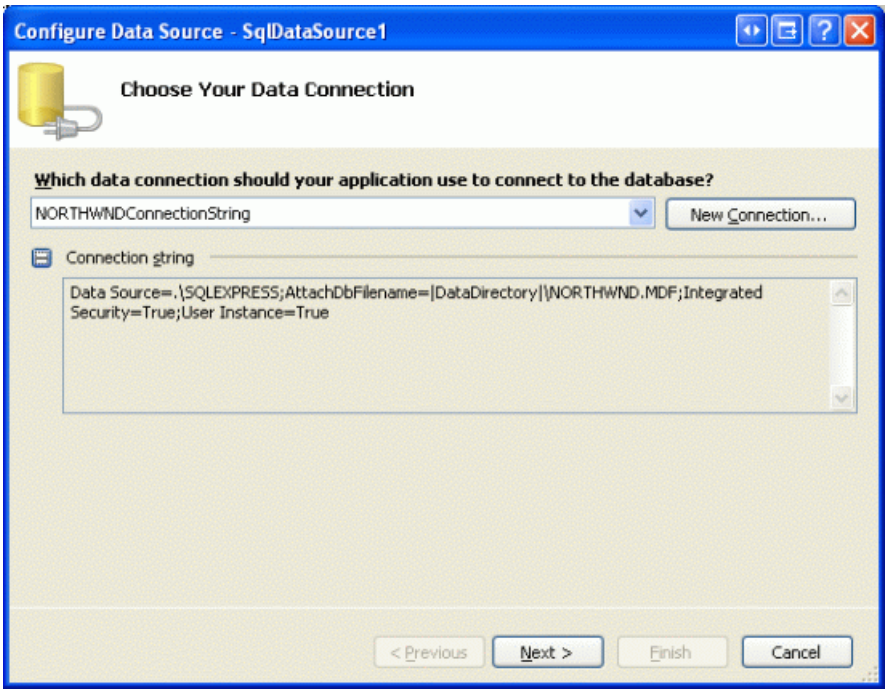


Figure 7: Choose the `NORTHWINDConnectionString` from the Drop-Down List

After choosing the database, the wizard asks for the query to return data. We can either specify the columns of a table or view to return or can enter a custom SQL statement or specify a stored procedure. You can toggle between this choice through the Specify a custom SQL statement or stored procedure and Specify columns from a table or view radio buttons.

> **NOTE**
> For this first example, let s use the Specify columns from a table or view option. We'll return to the wizard later in this tutorial and explore the Specify a custom SQL statement or stored procedure option.

Figure 8 shows the Configure the Select Statement screen when the Specify columns from a table or view radio button is selected. The drop-down list contains the set of tables and views in the Northwind database, with the selected table or view s columns displayed in the checkbox list below. For this example, let s return the `ProductID`, `ProductName`, and `UnitPrice` columns from the `Products` table. As Figure 8 shows, after making these selections the wizard shows the resulting SQL statement `SELECT [ProductID], [ProductName], [UnitPrice] FROM [Products]`.

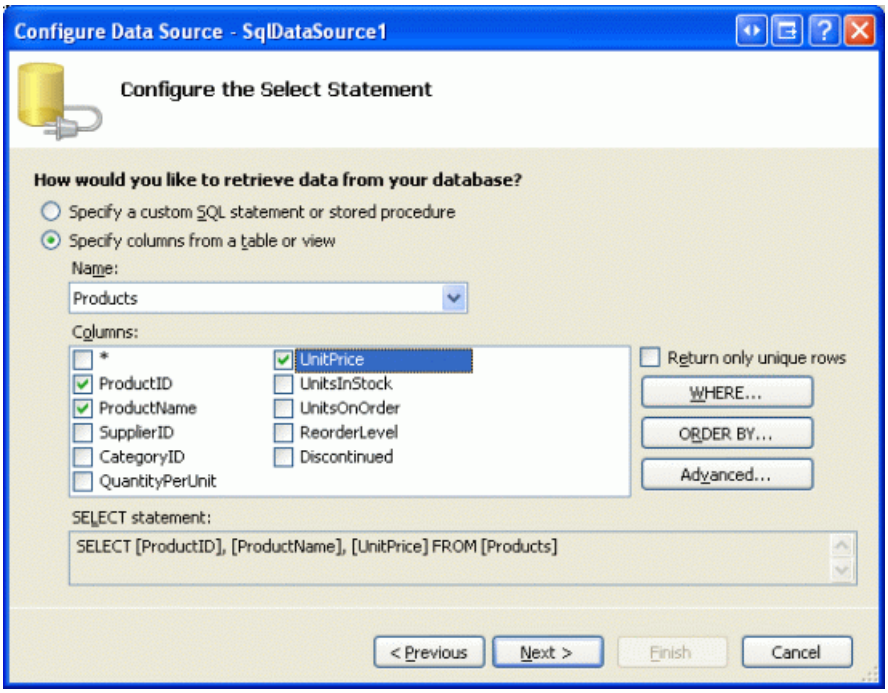


Figure 8: Return Data from the `Products` Table

Once you have configured the wizard to return the `ProductID`, `ProductName`, and `UnitPrice` columns from the `Products` table, click the Next button. This final screen provides an opportunity to examine the results of the query configured from the previous step. Clicking the Test Query button executes the configured `SELECT` statement and displays the results in a grid.

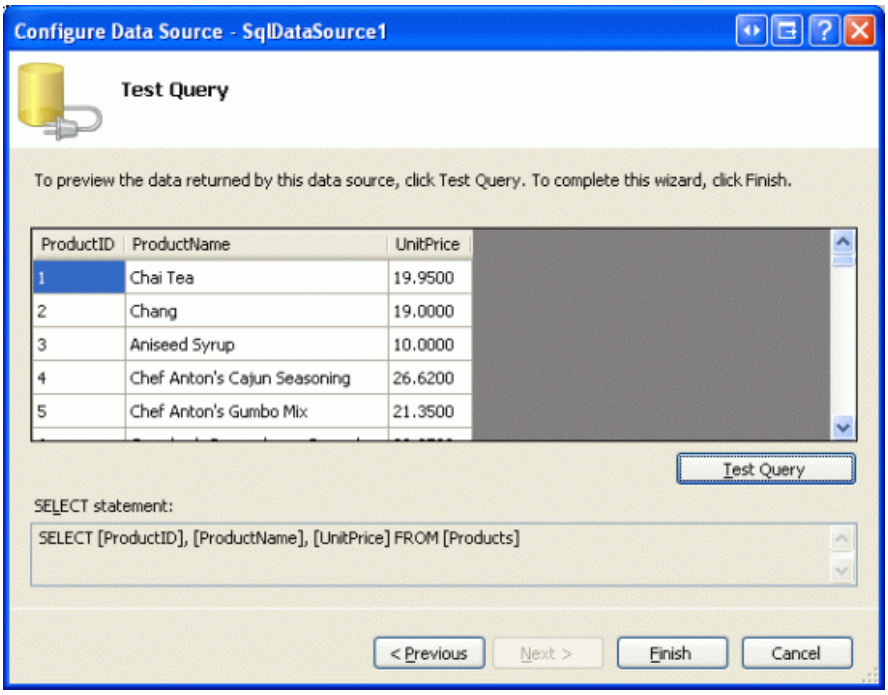


Figure 9: Click the Test Query Button to Review Your `SELECT` Query

To complete the wizard, click Finish.

Like with the ObjectDataSource, the SqlDataSource s wizard merely assigns values to the control s properties, namely the `ConnectionString` and `SelectCommand` properties. After completing the wizard, your SqlDataSource control s declarative markup should look similar to the following:

```
<asp:SqlDataSource ID="ProductsDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand="SELECT [ProductID], [ProductName], [UnitPrice] FROM [Products]">
</asp:SqlDataSource>
```

The `ConnectionString` property provides information on how to connect to the database. This property can be assigned a complete, hard-coded connection string value or can point to a connection string in `Web.config`. To reference a connection string value in Web.config, use the syntax `<%$ expressionPrefix:expressionValue %>`. Typically, *expressionPrefix* is ConnectionStrings and *expressionValue* is the name of the connect string in the `Web.config` `<connectionStrings>` section. However, the syntax can be used to reference `<appSettings>` elements or content from resource files. See ASP.NET Expressions Overview for more on this syntax.

The `SelectCommand` property specifies the ad-hoc SQL statement or stored procedure to execute to return the data.

Step 3: Adding a Data Web Control and Binding it to the SqlDataSource

Once the SqlDataSource has been configured, it can be bound to a data Web control, such as a GridView or DetailsView. For this tutorial, let s display the data in a GridView. From the Toolbox, drag a GridView onto the page and bind it to the `ProductsDataSource` SqlDataSource by choosing the data source from the drop-down list in the GridView s smart tag.

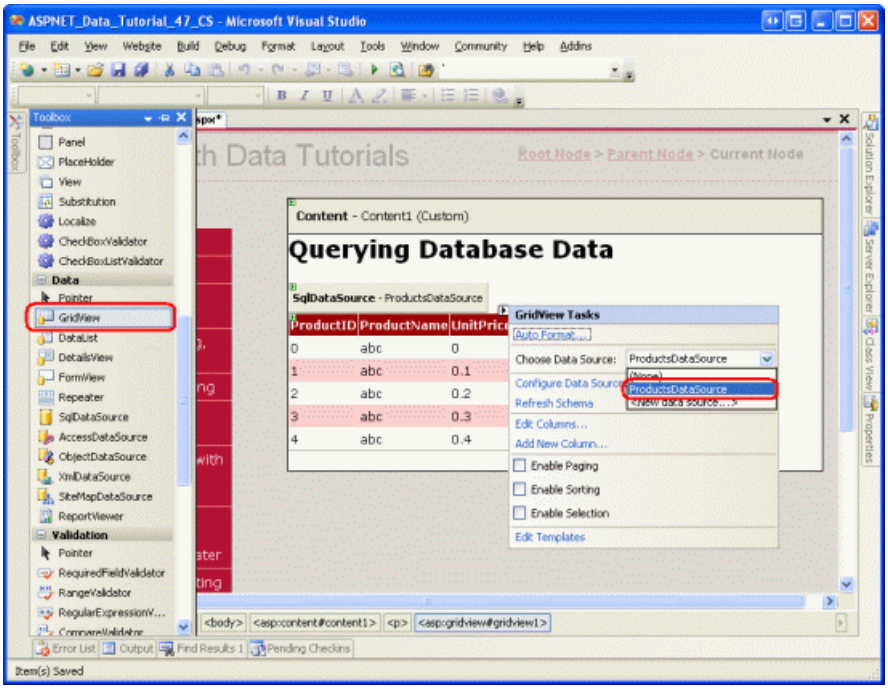


Figure 10: Add a GridView and Bind it to the SqlDataSource Control (Click to view full-size image)

Once you ve selected the SqlDataSource control from the drop-down list in the GridView s smart tag, Visual Studio will automatically add a BoundField or CheckBoxField to the GridView for each of the columns returned by the data source control. Since the SqlDataSource returns three database columns `ProductID`, `ProductName`, and `UnitPrice` there are three fields in the GridView.

Take a moment to configure the GridView s three BoundFields. Change the `ProductName` field s `HeaderText` property to Product Name and the `UnitPrice` field s to Price. Also format the `UnitPrice` field as a currency. After making these modifications, your GridView s declarative markup should look similar to the following:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="ProductID"
            InsertVisible="False" ReadOnly="True" SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName" HeaderText="Product Name"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
            SortExpression="UnitPrice" DataFormatString="{0:c}"
            HtmlEncode="False" />
    </Columns>
</asp:GridView>
```

Visit this page through a browser. As Figure 11 shows, the GridView lists each product s `ProductID`, `ProductName`, and `UnitPrice` values.

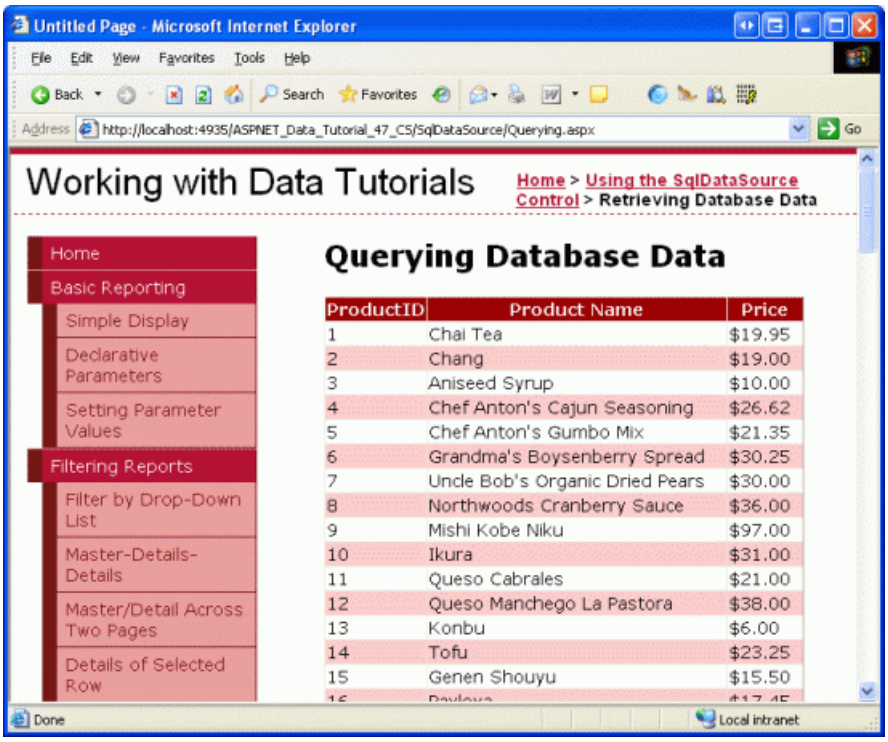


Figure 11: The GridView Displays Each Product s `ProductID`, `ProductName`, and `UnitPrice` Values (Click to view full-size image)

When the page is visited the GridView invokes its data source control s `Select()` method. When we were using the ObjectDataSource control, this called the `ProductsBLL` class s `GetProducts()` method. With the SqlDataSource, however, the `Select()` method establishes a connection to the specified database and issues the `SelectCommand` (`SELECT [ProductID], [ProductName], [UnitPrice] FROM [Products]`, in this example). The SqlDataSource returns its results which the GridView then enumerates, creating a row in the GridView for each database record returned.

The Built-In Data Web Control Features and the SqlDataSource Control

In general, the features inherent to the data Web controls paging, sorting, editing, deleting, inserting, and so on are specific to the data Web control and are not dependent on the data source control used. That is, the GridView can utilize its built-in paging, sorting, editing, and deleting whether it is bound to an ObjectDataSource or a SqlDataSource. However, certain data Web control features are sensitive to the data source control being used or to the data source control s configuration.

For example, in the Efficiently Paging Through Large Amounts of Data tutorial we discussed how, by default, the paging logic for the data Web controls naively returns *all* records from the underlying data source and then

displays only the appropriate subset of records given the current page index and the number of records to display per page. This model is highly inefficient when paging through sufficiently large result sets. Fortunately, the ObjectDataSource can be configured to support custom paging, which returns only the precise subset of records to display. The SqlDataSource control, however, lacks the properties for implementing custom paging.

Another subtlety with paging and sorting arises with the SqlDataSource. By default, the data returned from a SqlDataSource can be paged or sorted through the GridView. To demonstrate this, check the Enable Paging and Enable Sorting options in the GridView s smart tag in `Querying.aspx` and verify that this works as expected.

Sorting and paging works because the SqlDataSource retrieves the database data into a loosely-typed DataSet. The total number of records returned by the query an essential aspect to implementing paging can be ascertained from the DataSet. Additionally, the DataSet s results can be sorted through a DataView. These capabilities are automatically used by the SqlDataSource when the GridView requests paged or sorted data.

The SqlDataSource can be configured to return a DataReader instead of a DataSet by changing its `DataSourceMode` property from `DataSet` (the default) to `DataReader`. Using a DataReader might be preferred in situations when passing the SqlDataSource s results to existing code that expects a DataReader. Furthermore, since DataReaders are considerably simpler objects than DataSets, they offer better performance. If you make this change, however, the data Web control can neither sort nor page since the SqlDataSource cannot ascertain how many records are returned by the query, nor does the DataReader offer any techniques for sorting the returned data.

Step 4: Using a Custom SQL Statement or Stored Procedure

When configuring the SqlDataSource control, the query used to return data can be specified in one of two approaches as a custom SQL statement or stored procedure, or as columns from an existing table or view. In Step 2 we examined selecting columns from the `Products` table. Let s look at using a custom SQL statement.

Add another GridView control to the `Querying.aspx` page and choose to create a new data source from the drop-down list in the smart tag. Next, indicate that the data will be pulled from a database this will create a new SqlDataSource control. Name the control `ProductsWithCategoryInfoDataSource`.

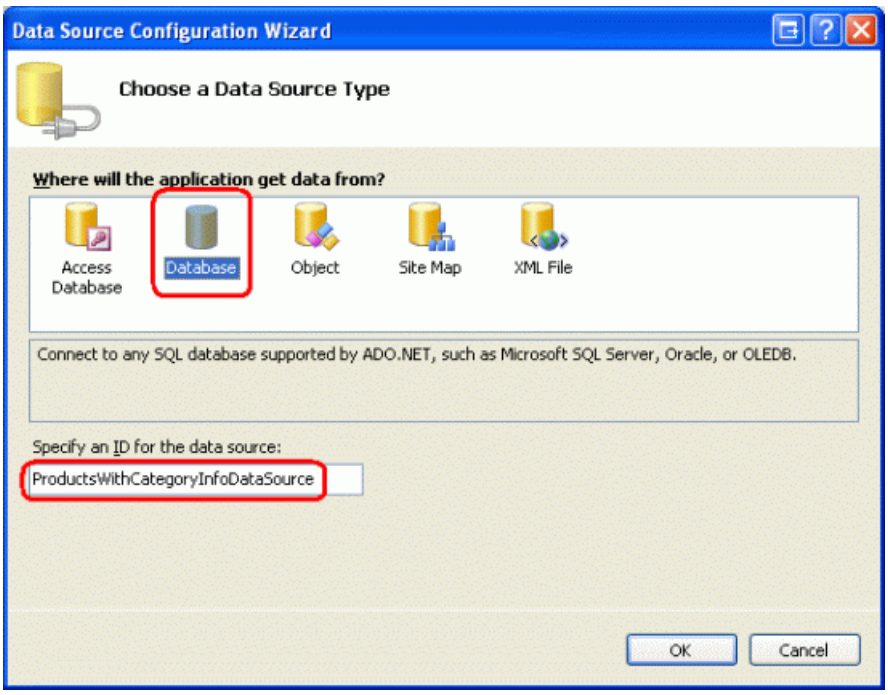


Figure 12: Create a New SqlDataSource Control Named `ProductsWithCategoryInfoDataSource`

The next screen asks us to specify the database. As we did back in Figure 7, select the `NORTHWINDConnectionString` from the drop-down list and click Next. In the Configure the Select Statement screen, choose the Specify a custom SQL statement or stored procedure radio button and click Next. This will bring up the Define Custom Statements or Stored Procedures screen, which offers tabs labeled SELECT, UPDATE, INSERT, and DELETE. In each tab you can

enter a custom SQL statement into the textbox or choose a stored procedure from the drop-down list. In this tutorial we will look at entering a custom SQL statement; the next tutorial includes an example that uses a stored procedure.

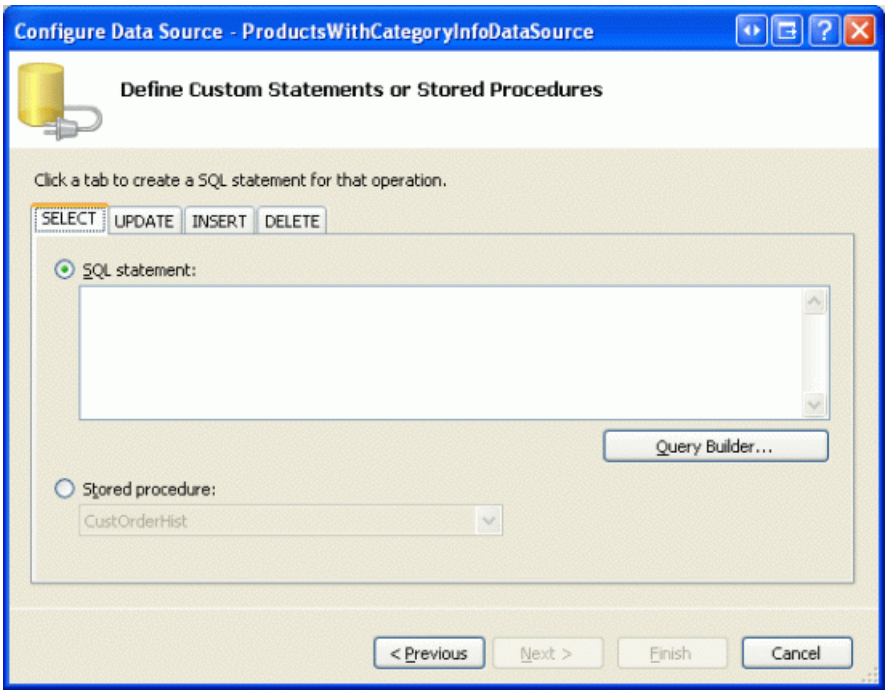


Figure 13: Enter a Custom SQL Statement or Pick a Stored Procedure

The custom SQL statement can be entered by hand into the textbox or can be constructed graphically by clicking the Query Builder button. From either the Query Builder or the textbox, use the following query to return the `ProductID` and `ProductName` fields from the `Products` table using a `JOIN` to retrieve the product s `CategoryName` from the `Categories` table:

```
SELECT Products.ProductID, Products.ProductName, Categories.CategoryName
FROM Categories
    INNER JOIN Products ON
        Categories.CategoryID = Products.CategoryID
```

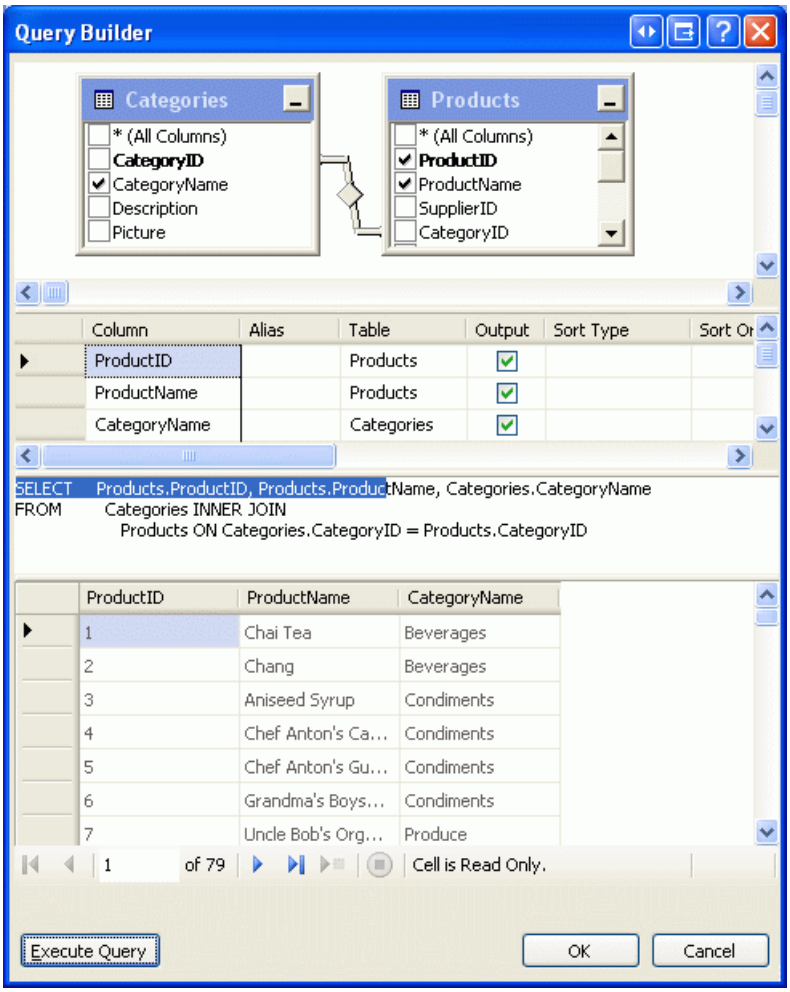


Figure 14: You can Graphically Construct the Query Using the Query Builder

After specifying the query, click Next to proceed to the Test Query screen. Click Finish to complete the SqlDataSource wizard.

After completing the wizard, the GridView will have three BoundFields added to it displaying the `ProductID`, `ProductName`, and `CategoryName` columns returned from the query and resulting in the following declarative markup:

```
<asp:GridView ID="GridView2" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsWithCategoryInfoDataSource"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="ProductID"
            InsertVisible="False" ReadOnly="True" SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName" HeaderText="ProductName"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="CategoryName"
            SortExpression="CategoryName" />
    </Columns>
</asp:GridView>
<asp:SqlDataSource ID="ProductsWithCategoryInfoDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand="
        SELECT Products.ProductID, Products.ProductName, Categories.CategoryName
        FROM Categories
        INNER JOIN Products ON Categories.CategoryID = Products.CategoryID">
</asp:SqlDataSource>
```

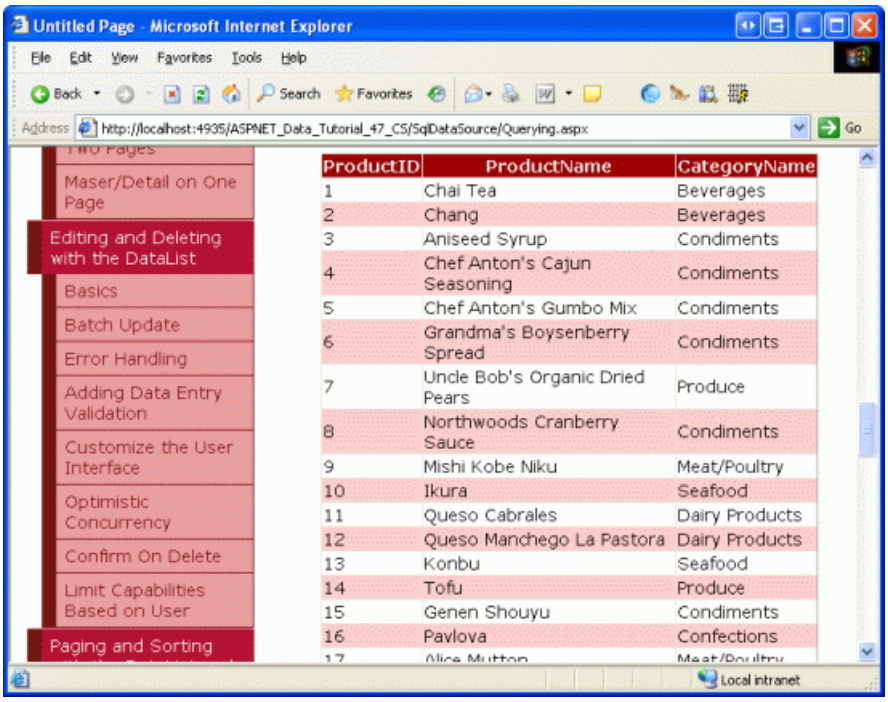


Figure 15: The GridView Shows Each Product s ID, Name, and Associated Category Name (Click to view full-size image)

Summary

In this tutorial we saw how to query and display data using the SqlDataSource control. Like the ObjectDataSource, the SqlDataSource serves as a proxy, providing a declarative approach to accessing data. Its properties specify the database to connect to and the SQL `SELECT` query to execute; they can be specified through the Properties window or by using the Configure DataSource wizard.

The `SELECT` query examples we examined in this tutorial returned all of the records from the specified query. The SqlDataSource control, however, can include a `WHERE` clause with parameters whose values are assigned programmatically or are automatically pulled from a specified source. We'll examine how to create and use parameterized queries in the next tutorial!

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Accessing Relational Database Data
- SqlDataSource Control Overview
- ASP.NET Quickstart Tutorials: The SqlDataSource Control
- The Web.config `<connectionStrings>` Element
- Database Connection String Reference

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Susan Connery, Bernadette Leigh, and David Suru. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Using Parameterized Queries with the SqlDataSource (C#)

2/20/2020 • 20 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

In this tutorial, we continue our look at the SqlDataSource control and learn how to define parameterized queries. The parameters can be specified both declaratively and programmatically, and can be pulled from a number of locations such as the querystring, Session state, other controls, and more.

Introduction

In the previous tutorial we saw how to use the SqlDataSource control to retrieve data directly from a database. Using the Configure Data Source wizard, we could choose the database and then either: pick the columns to return from a table or view; enter a custom SQL statement; or use a stored procedure. Whether selecting columns from a table or view or entering a custom SQL statement, the SqlDataSource control s `SelectCommand` property is assigned the resulting ad-hoc SQL `SELECT` statement and it is this `SELECT` statement that is executed when the SqlDataSource s `Select()` method is invoked (either programmatically or automatically from a data Web control).

The SQL `SELECT` statements used in the previous tutorial s demos lacked `WHERE` clauses. In a `SELECT` statement, the `WHERE` clause can be used to limit the results returned. For example, to display the names of products costing more than $50.00, we could use the following query:

```
SELECT ProductName
FROM Products
WHERE UnitPrice > 50.00
```

Typically, the values used in a `WHERE` clause are determine by some external source, such as a querystring value, a session variable, or user input from a Web control on the page. Ideally, such inputs are specified through the use of *parameters*. With Microsoft SQL Server, parameters are denoted using `@parameterName`, as in:

```
SELECT ProductName
FROM Products
WHERE UnitPrice > @Price
```

The SqlDataSource supports parameterized queries, both for `SELECT` statements and `INSERT`, `UPDATE`, and `DELETE` statements. Moreover, the parameter values can be automatically pulled from a variety of sources the querystring, session state, controls on the page, and so on or can be assigned programmatically. In this tutorial, we'll see how to define parameterized queries as well as how to specify the parameter values both declaratively and programmatically.

NOTE

In the previous tutorial we compared the ObjectDataSource which has been our tool of choice over the first 46 tutorials with the SqlDataSource, noting their conceptual similarities. These similarities also extend to parameters. The ObjectDataSource s parameters mapped to the input parameters for the methods in the Business Logic Layer. With the SqlDataSource, the parameters are defined directly within the SQL query. Both controls have collections of parameters for their `Select()`, `Insert()`, `Update()`, and `Delete()` methods, and both can have these parameter values populated from pre-defined sources (querystring values, session variables, and so on) or assigned programmatically.

Creating a Parameterized Query

The SqlDataSource control s Configure Data Source wizard offers three avenues for defining the command to execute to retrieve database records:

- By picking the columns from an existing table or view,
- By entering a custom SQL statement, or
- By choosing a stored procedure

When picking columns from an existing table or view, the parameters for the `WHERE` clause must be specified through the Add `WHERE` Clause dialog box. When creating a custom SQL statement, however, you can enter the parameters directly into the `WHERE` clause (using `@parameterName` to denote each parameter). A stored procedure consists of one or more SQL statements, and these statements can be parameterized. The parameters used in the SQL statements, however, must be passed in as input parameters to the stored procedure.

Since creating a parameterized query depends on how the SqlDataSource s `SelectCommand` is specified, let s take a look at all three approaches. To get started, open the `ParameterizedQueries.aspx` page in the `SqlDataSource` folder, drag a SqlDataSource control from the Toolbox onto the Designer, and set its `ID` to `Products25BucksAndUnderDataSource`. Next, click the Configure Data Source link from the control s smart tag. Select the database to use (`NORTHWINDConnectionString`) and click Next.

Step 1: Adding a `WHERE` Clause When Picking the Columns from a Table or View

When selecting the data to return from the database with the SqlDataSource control, the Configure Data Source wizard allows us to simply pick the columns to return from an existing table or view (see Figure 1). Doing so automatically builds up a SQL `SELECT` statement, which is what is sent to the database when the SqlDataSource s `Select()` method is invoked. As we did in the previous tutorial, select the Products table from the drop-down list and check the `ProductID`, `ProductName`, and `UnitPrice` columns.

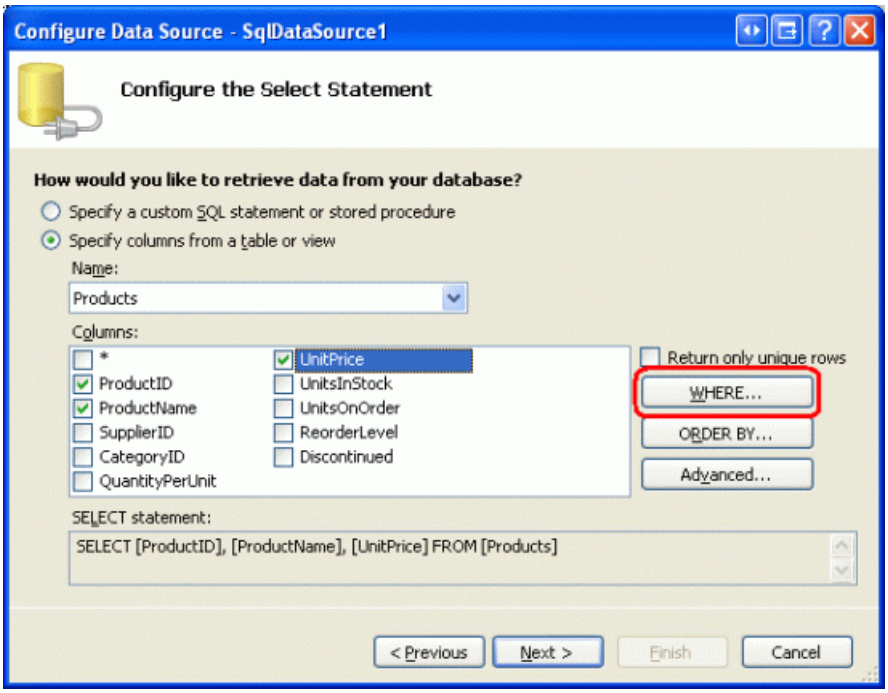


Figure 1: Pick the Columns to Return from a Table or View (Click to view full-size image)

To include a `WHERE` clause in the `SELECT` statement, click the `WHERE` button, which brings up the Add `WHERE` Clause dialog box (see Figure 2). To add a parameter to limit the results returned by the `SELECT` query, first choose the column to filter the data by. Next, choose the operator to use for filtering (=, <, <=, >, and so on). Finally, choose the source of the parameter s value, such as from the querystring or session state. After configuring the parameter, click the Add button to include it in the `SELECT` query.

For this example, let s only return those results where the `UnitPrice` value is less than or equal to $25.00. Therefore, pick `UnitPrice` from the Column drop-down list and <= from the Operator drop-down list. When using a hard-coded parameter value (such as $25.00) or if the parameter value is to be specified programmatically, select None from the Source drop-down list. Next, enter the hard-coded parameter value in the Value textbox 25.00 and complete the process by clicking the Add button.

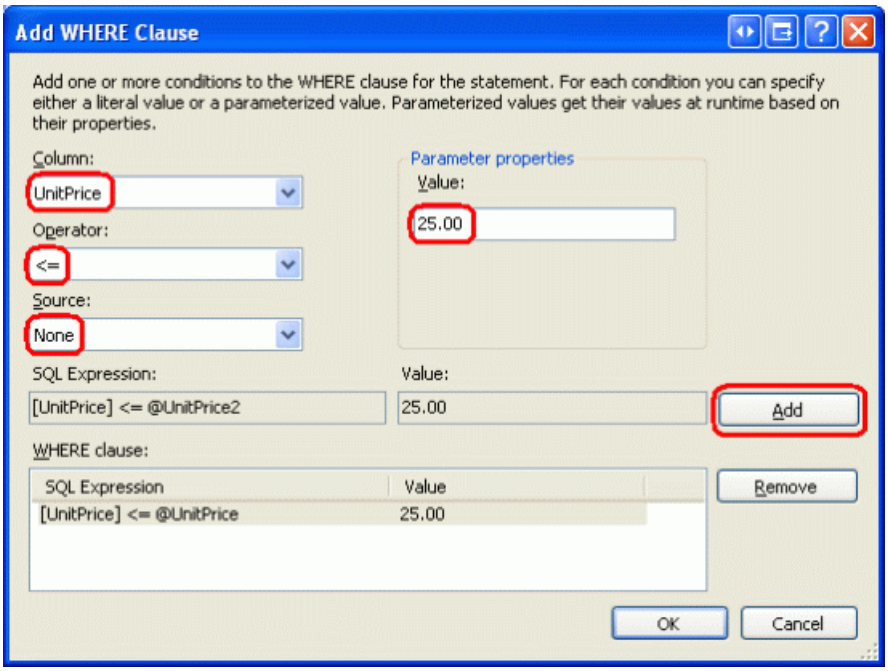


Figure 2: Limit the Results Returned from the Add `WHERE` Clause Dialog Box (Click to view full-size image)

After adding the parameter, click OK to return to the Configure Data Source wizard. The `SELECT` statement at the bottom of the wizard should now include a `WHERE` clause with a parameter named `@UnitPrice`:

```
SELECT [ProductID], [ProductName], [UnitPrice]
FROM [Products]
WHERE ([UnitPrice] <= @UnitPrice)
```

NOTE

If you specify multiple conditions in the `WHERE` clause from the Add `WHERE` Clause dialog box, the wizard joins them with the `AND` operator. If you need to include an `OR` in the `WHERE` clause (such as `WHERE UnitPrice <= @UnitPrice OR Discontinued = 1`) then you have to build the `SELECT` statement through the custom SQL statement screen.

Complete configuring the SqlDataSource (click Next, then Finish) and then inspect the SqlDataSource s declarative markup. The markup now includes a `<SelectParameters>` collection, which spells out the sources for the parameters in the `SelectCommand`.

```
<asp:SqlDataSource ID="Products25BucksAndUnderDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand=
        "SELECT [ProductID], [ProductName], [UnitPrice]
        FROM [Products] WHERE ([UnitPrice] <= @UnitPrice)">
    <SelectParameters>
        <asp:Parameter DefaultValue="25.00" Name="UnitPrice" Type="Decimal" />
    </SelectParameters>
</asp:SqlDataSource>
```

When the SqlDataSource s `Select()` method is invoked, the `UnitPrice` parameter value (25.00) is applied to the `@UnitPrice` parameter in the `SelectCommand` before being sent to the database. The net result is that only those products less than or equal to $25.00 are returned from the `Products` table. To confirm this, add a GridView to the page, bind it to this data source, and then view the page through a browser. You should only see those products listed that are less than or equal to $25.00, as Figure 3 confirms.

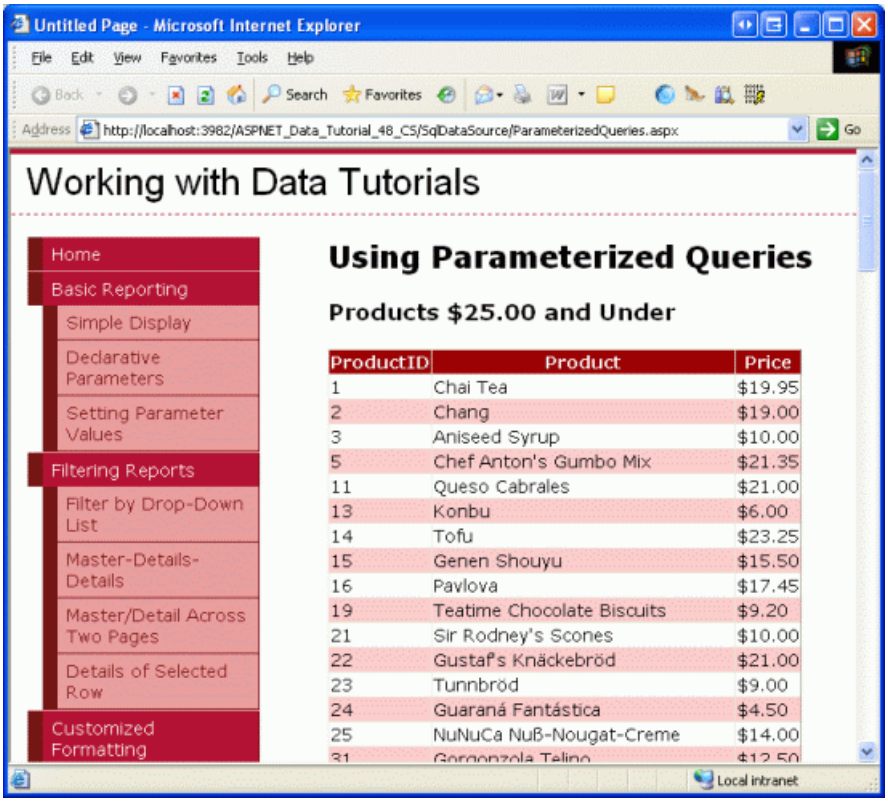


Figure 3: Only Those Products Less Than or Equal to $25.00 are Displayed (Click to view full-size image)

Step 2: Adding Parameters to a Custom SQL Statement

When adding a custom SQL statement you can enter the `WHERE` clause explicitly or specify a value in the Filter cell of the Query Builder. To demonstrate this, let s display just those products in a GridView whose prices are less than a certain threshold. Start by adding a TextBox to the `ParameterizedQueries.aspx` page to collect this threshold value from the user. Set the TextBox s `ID` property to `MaxPrice`. Add a Button Web control and set its `Text` property to Display Matching Products .

Next, drag a GridView onto the page and from its smart tag choose to create a new SqlDataSource named `ProductsFilteredByPriceDataSource`. From the Configure Data Source wizard, proceed to the Specify a custom SQL statement or stored procedure screen (see Figure 4) and enter the following query:

```
SELECT ProductName, UnitPrice
FROM Products
WHERE UnitPrice <= @MaximumPrice
```

After entering the query (either manually or through the Query Builder), click Next.

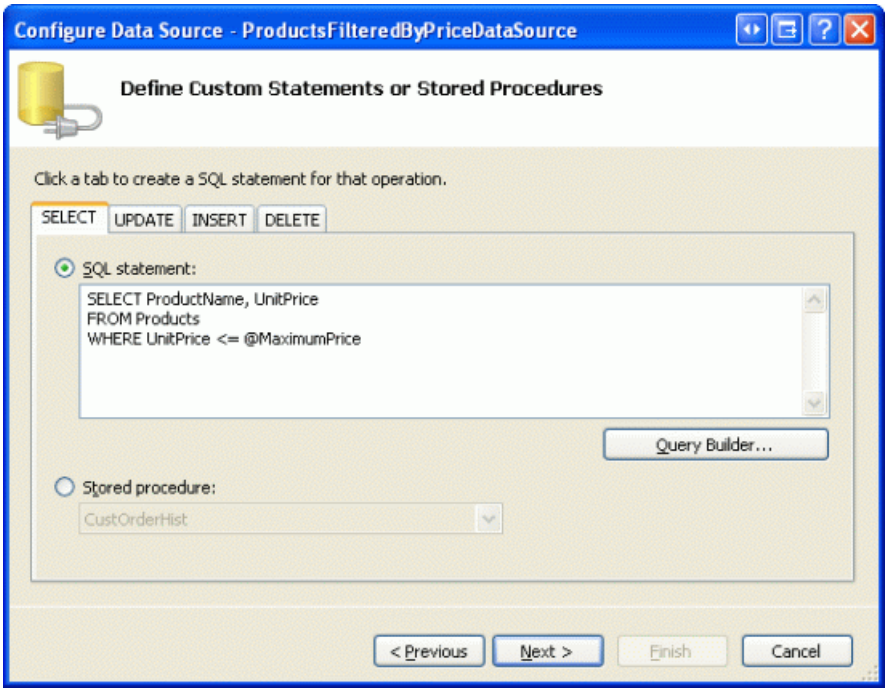


Figure 4: Return Only Those Products Less Than or Equal to a Parameter Value (Click to view full-size image)

Since the query includes parameters, the next screen in the wizard prompts us for the source of the parameters values. Choose Control from the Parameter source drop-down list and `MaxPrice` (the TextBox control s `ID` value) from the ControlID drop-down list. You can also enter an optional default value to use in the case where the user has not entered any text into the `MaxPrice` TextBox. For the time being, do not enter a default value.

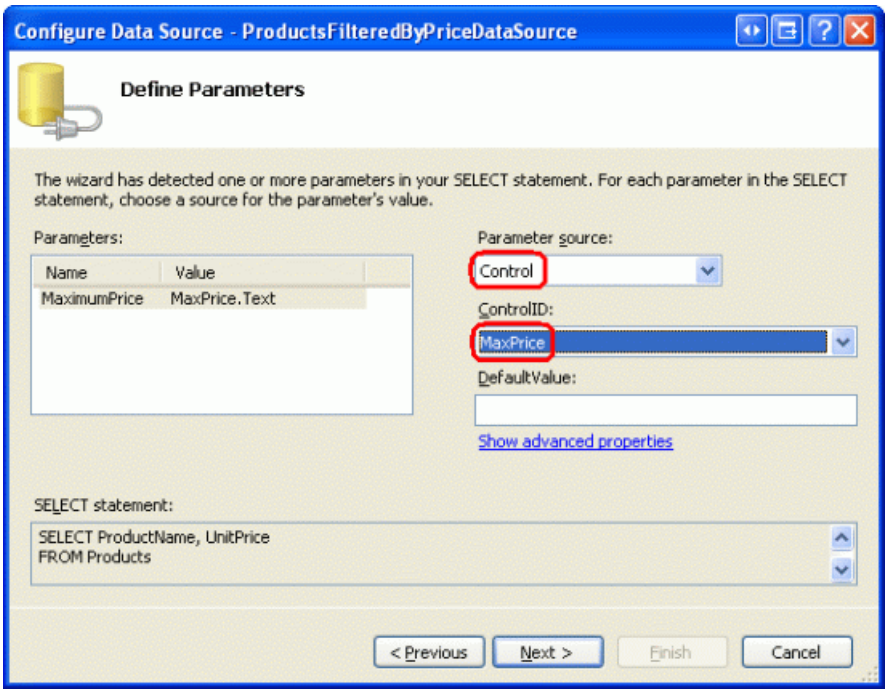


Figure 5: The `MaxPrice` TextBox s `Text` Property is Used as the Parameter Source (Click to view full-size image)

Complete the Configure Data Source wizard by clicking Next, then Finish. The declarative markup for the GridView, TextBox, Button, and SqlDataSource follows:

```
Maximum price:
$<asp:TextBox ID="MaxPrice" runat="server" Columns="5" />

<asp:Button ID="DisplayProductsLessThanButton" runat="server"
    Text="Display Matching Products" />
<asp:GridView ID="GridView2" runat="server" AutoGenerateColumns="False"
    DataSourceID="ProductsFilteredByPriceDataSource" EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
            HtmlEncode="False" DataFormatString="{0:c}"
            SortExpression="UnitPrice" />
    </Columns>
</asp:GridView>
<asp:SqlDataSource ID="ProductsFilteredByPriceDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand=
        "SELECT ProductName, UnitPrice
        FROM Products WHERE UnitPrice <= @MaximumPrice">
    <SelectParameters>
        <asp:ControlParameter ControlID="MaxPrice" Name="MaximumPrice"
            PropertyName="Text" />
    </SelectParameters>
</asp:SqlDataSource>
```

Note that the parameter within the SqlDataSource s `<SelectParameters>` section is a `ControlParameter`, which includes additional properties like `ControlID` and `PropertyName`. When the SqlDataSource s `Select()` method is invoked, the `ControlParameter` grabs the value from the specified Web control property and assigns it to the corresponding parameter in the `SelectCommand`. In this example, the `MaxPrice` s Text property is used as the `@MaxPrice` parameter value.

Take a minute to view this page through a browser. When first visiting the page or whenever the `MaxPrice` TextBox lacks a value no records are displayed in the GridView.

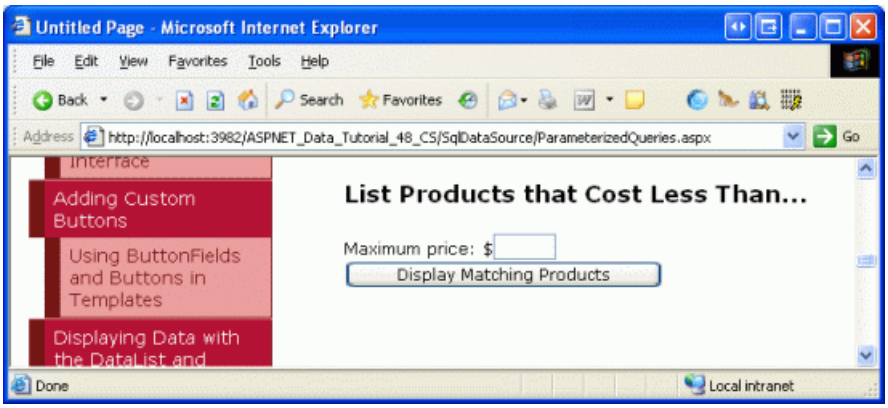


Figure 6: No Records are Displayed When the `MaxPrice` TextBox is Empty (Click to view full-size image)

The reason no products are shown is because, by default, an empty string for a parameter value is converted into a database `NULL` value. Since the comparison of `[UnitPrice] <= NULL` always evaluates as False, no results are returned.

Enter a value into the textbox, like 5.00, and click the Display Matching Products button. On postback, the SqlDataSource informs the GridView that one of its parameter sources has changed. Consequently, the GridView rebinds to the SqlDataSource, displaying those products less than or equal to $5.00.

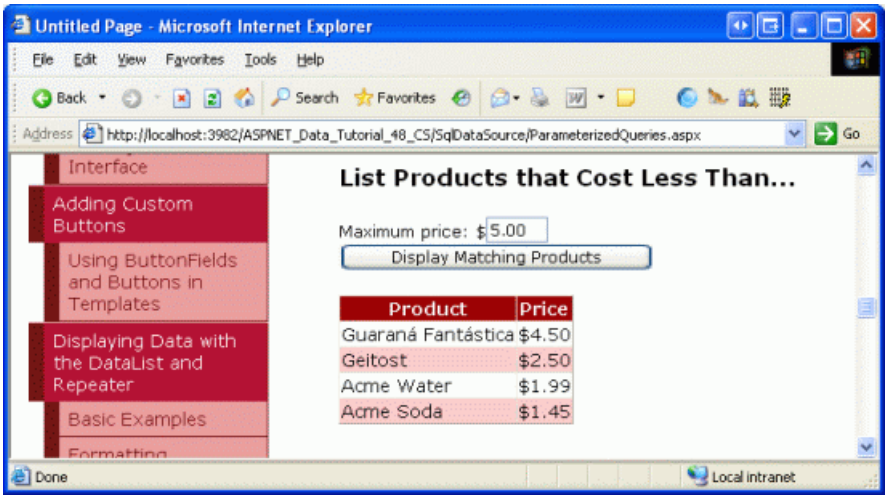


Figure 7: Products Less Than or Equal to $5.00 are Displayed (Click to view full-size image)

Initially Displaying All Products

Rather than displaying no products when the page is first loaded, we may want to display *all* products. One way to list all products whenever the `MaxPrice` TextBox is empty is to set the parameter s default value to some insanely high value, like 1000000, since it s unlikely that Northwind Traders will ever have inventory whose unit price exceeds $1,000,000. However, this approach is shortsighted and might not work in other situations.

In previous tutorials - Declarative Parameters and Master/Detail Filtering With a DropDownList we were faced with a similar problem. Our solution there was to put this logic in the Business Logic Layer. Specifically, the BLL examined the incoming value and, if it was `NULL` or some reserved value, the call was routed to the DAL method that returned all records. If the incoming value was a normal filtering value, a call was made to the DAL method that executed a SQL statement that used a parameterized `WHERE` clause with the supplied value.

Unfortunately, we bypass the architecture when using the SqlDataSource. Instead, we need to customize the SQL statement to intelligently grab all records if the `@MaximumPrice` parameter is `NULL` or some reserved value. For this exercise, let s have it so that if the `@MaximumPrice` parameter is equal to `-1.0`, then *all* of the records are to be returned (`-1.0` works as a reserved value since no product can have a negative `UnitPrice` value). To accomplish this we can use the following SQL statement:

```
SELECT ProductName, UnitPrice
FROM Products
WHERE UnitPrice <= @MaximumPrice OR @MaximumPrice = -1.0
```

This `WHERE` clause returns *all* records if the `@MaximumPrice` parameter equals `-1.0`. If the parameter value is not `-1.0`, only those products whose `UnitPrice` is less than or equal to the `@MaximumPrice` parameter value are returned. By setting the default value of the `@MaximumPrice` parameter to `-1.0`, on the first page load (or whenever the `MaxPrice` TextBox is empty), `@MaximumPrice` will have a value of `-1.0` and all products will be displayed.

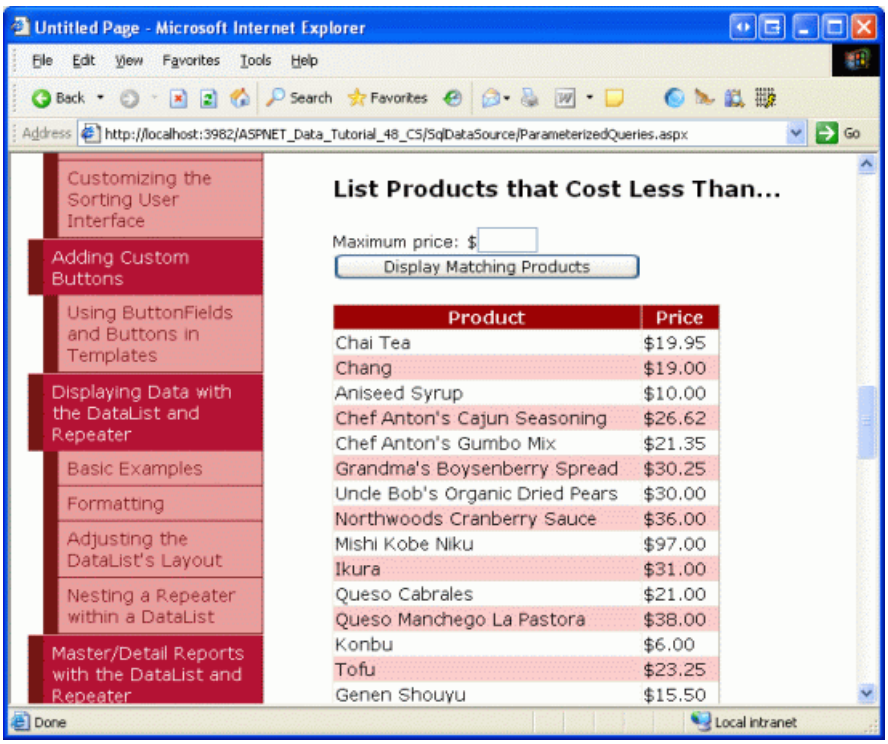


Figure 8: Now All Products are Displayed When the `MaxPrice` TextBox is Empty (Click to view full-size image)

There are a couple of caveats to note with this approach. First, realize that the parameter s data type is inferred by it s usage in the SQL query. If you change the `WHERE` clause from `@MaximumPrice = -1.0` to `@MaximumPrice = -1`, the runtime treats the parameter as an integer. If you then attempt to assign the `MaxPrice` TextBox to a decimal value (like 5.00), an error will occur because it cannot convert 5.00 to an integer. To remedy this, either make sure that you use `@MaximumPrice = -1.0` in the `WHERE` clause or, better yet, set the `ControlParameter` object s `Type` property to Decimal .

Secondly, by adding the `OR @MaximumPrice = -1.0` to the `WHERE` clause, the query engine cannot use an index on `UnitPrice` (assuming one exists), thereby resulting in a table scan. This can impact performance if there are a sufficiently large number of records in the `Products` table. A better approach would be to move this logic to a stored procedure where an `IF` statement would either perform a `SELECT` query from the `Products` table without a `WHERE` clause when all records need to be returned or one whose `WHERE` clause contains just the `UnitPrice` criteria, so that an index can be used.

Step 3: Creating and Using Parameterized Stored Procedures

Stored procedures can include a set of input parameters that can then be used in the SQL statement(s) defined within the stored procedure. When configuring the SqlDataSource to use a stored procedure that accepts input parameters, these parameter values can be specified using the same techniques as with ad-hoc SQL statements.

To illustrate using stored procedures in the SqlDataSource, let s create a new stored procedure in the Northwind database named `GetProductsByCategory`, which accepts a parameter named `@CategoryID` and returns all of the

columns of the products whose `CategoryID` column matches `@CategoryID`. To create a stored procedure, go to the Server Explorer and drill down into the `NORTHWND.MDF` database. (If you don t see the Server Explorer, bring it up by going to the View menu and selecting the Server Explorer option.)

From the `NORTHWND.MDF` database, right-click on the Stored Procedures folder, choose Add New Stored Procedure, and enter the following syntax:

```
CREATE PROCEDURE dbo.GetProductsByCategory
(
      @CategoryID int
)
AS
SELECT *
FROM Products
WHERE CategoryID = @CategoryID
```

Click the Save icon (or Ctrl+S) to save the stored procedure. You can test the stored procedure by right-clicking it from the Stored Procedures folder and choosing Execute. This will prompt you for the stored procedure s parameters (`@CategoryID`, in this instance), after which the results will be displayed in the Output window.

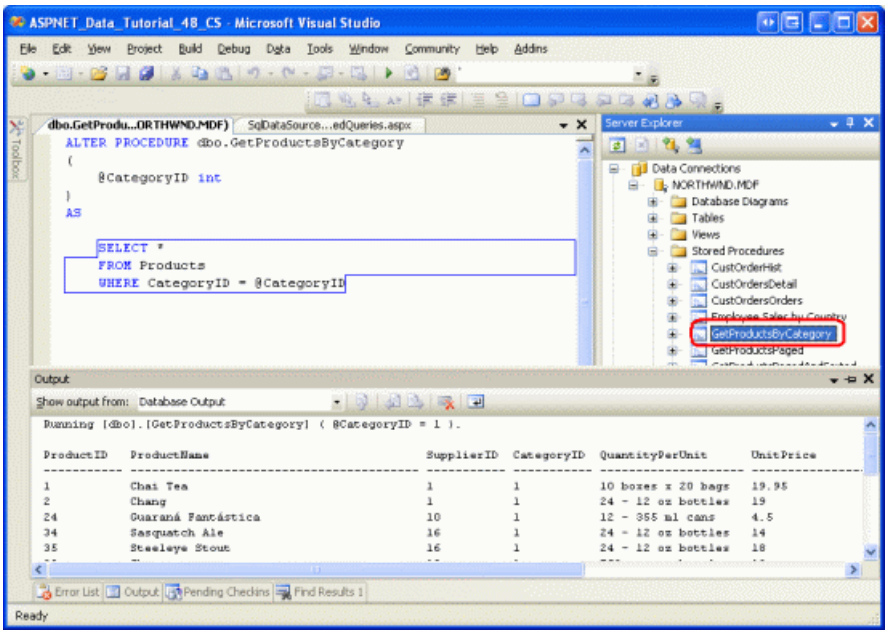


Figure 9: The `GetProductsByCategory` Stored Procedure when Executed with a `@CategoryID` of 1 (Click to view full-size image)

Let s use this stored procedure to display all products in the Beverages category in a GridView. Add a new GridView to the page and bind it to a new SqlDataSource named `BeverageProductsDataSource`. Continue to the Specify a custom SQL statement or stored procedure screen, select the Stored procedure radio button, and pick the `GetProductsByCategory` stored procedure from the drop-down list.

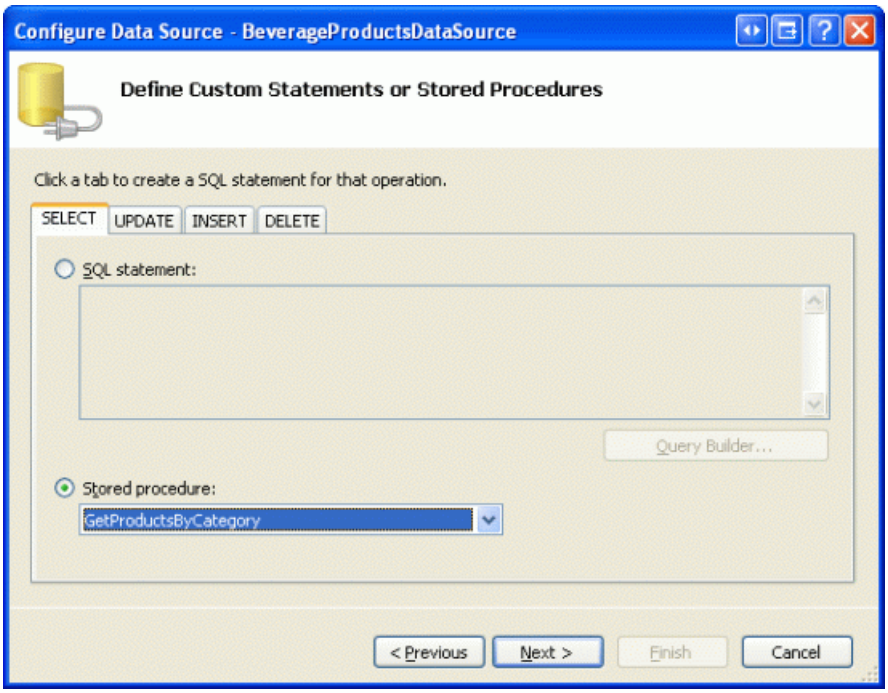


Figure 10: Select the `GetProductsByCategory` Stored Procedure from the Drop-Down List (Click to view full-size image)

Since the stored procedure accepts an input parameter (`@CategoryID`), clicking Next prompts us to specify the source for this parameter s value. The Beverages `CategoryID` is 1, so leave the Parameter source drop-down list at None and enter 1 into the DefaultValue textbox.

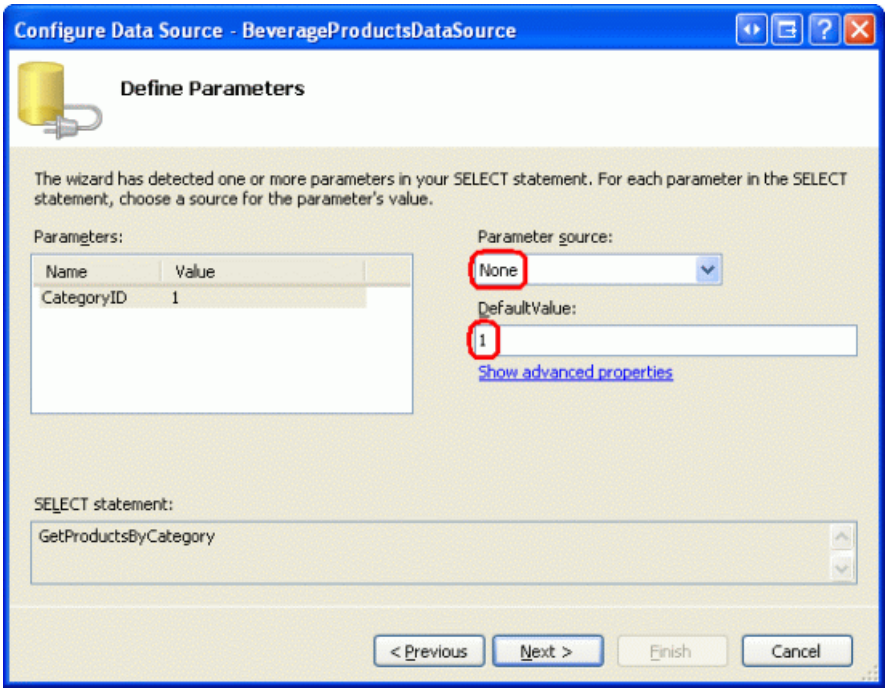
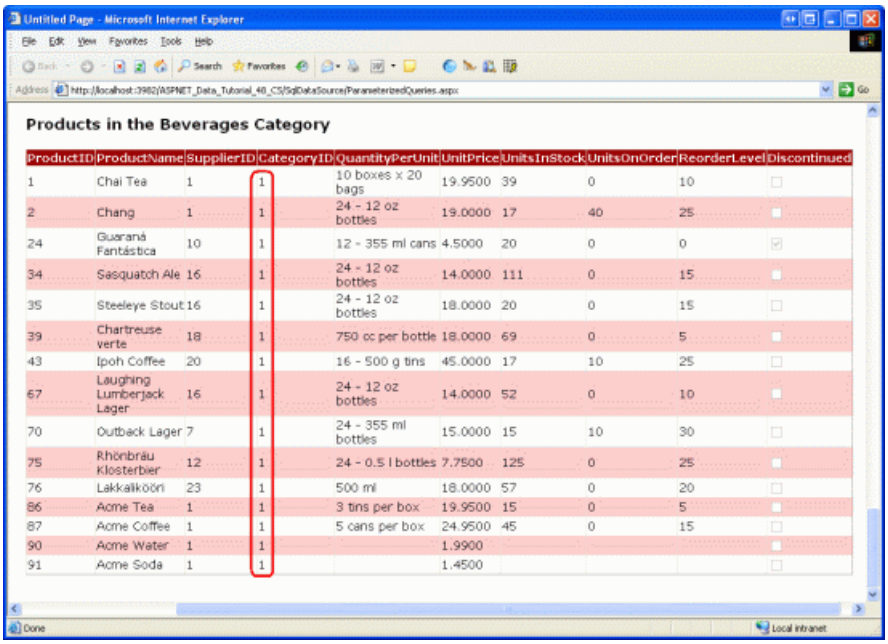


Figure 11: Use a Hard-Coded Value of 1 to Return the Products in the Beverages Category (Click to view full-size image)

As the following declarative markup shows, when using a stored procedure, the SqlDataSource s `SelectCommand` property is set to the name of the stored procedure and the `SelectCommandType` property is set to `StoredProcedure`, indicating that the `SelectCommand` is the name of a stored procedure rather than an ad-hoc SQL statement.

```
<asp:SqlDataSource ID="BeverageProductsDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand="GetProductsByCategory" SelectCommandType="StoredProcedure">
    <SelectParameters>
        <asp:Parameter DefaultValue="1" Name="CategoryID" Type="Int32" />
    </SelectParameters>
</asp:SqlDataSource>
```

Test out the page in a browser. Only those products that belong to the Beverages category are displayed, although *all* of the product fields are displayed since the `GetProductsByCategory` stored procedure returns all of the columns from the `Products` table. We could, of course, limit or customize the fields displayed in the GridView from the GridView s Edit Columns dialog box.



ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
1	Chai Tea	1	1	10 boxes x 20 bags	19.9500	39	0	10	
2	Chang	1	1	24 - 12 oz bottles	19.0000	17	40	25	
24	Guaraná Fantástica	10	1	12 - 355 ml cans	4.5000	20	0	0	
34	Sasquatch Ale	16	1	24 - 12 oz bottles	14.0000	111	0	15	
35	Steeleye Stout	16	1	24 - 12 oz bottles	18.0000	20	0	15	
39	Chartreuse verte	18	1	750 cc per bottle	18.0000	69	0	5	
43	Ipoh Coffee	20	1	16 - 500 g tins	45.0000	17	10	25	
67	Laughing Lumberjack Lager	16	1	24 - 12 oz bottles	14.0000	52	0	10	
70	Outback Lager	7	1	24 - 355 ml bottles	15.0000	15	10	30	
75	Rhönbräu Klosterbier	12	1	24 - 0.5 l bottles	7.7500	125	0	25	
76	Lakkalikööri	23	1	500 ml	18.0000	57	0	20	
86	Acme Tea	1	1	3 tins per box	19.9500	15	0	5	
87	Acme Coffee	1	1	5 cans per box	24.9500	45	0	15	
90	Acme Water	1	1		1.9900				
91	Acme Soda	1	1		1.4500				

Figure 12: All of the Beverages are Displayed (Click to view full-size image)

Step 4: Programmatically Invoking a SqlDataSource s `Select()` Statement

The examples we ve seen in the previous tutorial and this tutorial so far have bound SqlDataSource controls directly to a GridView. The SqlDataSource control s data, however, can be programmatically accessed and enumerated in code. This can be particularly useful when you need to query data to inspect it, but don t need to display it. Rather than having to write all of the boilerplate ADO.NET code to connect to the database, specify the command, and retrieve the results, you can let the SqlDataSource handle this monotonous code.

To illustrate working with the SqlDataSource s data programmatically, imagine that your boss has approached you with a request to create a web page that displays the name of a randomly selected category and its associated products. That is, when a user visits this page, we want to randomly choose a category from the `Categories` table, display the category name, and then list the products belonging to that category.

To accomplish this we need two SqlDataSource controls one to grab a random category from the `Categories` table and another to get the category s products. We'll build the SqlDataSource that retrieves a random category record in this step; Step 5 looks at crafting the SqlDataSource that retrieves the category s products.

Start by adding a SqlDataSource to `ParameterizedQueries.aspx` and set its `ID` to `RandomCategoryDataSource`. Configure it so that it uses the following SQL query:

```
SELECT TOP 1 CategoryID, CategoryName
FROM Categories
ORDER BY NEWID()
```

`ORDER BY NEWID()` returns the records sorted in random order (see Using `NEWID()` to Randomly Sort Records). `SELECT TOP 1` returns the first record from the result set. Put together, this query returns the `CategoryID` and `CategoryName` column values from a single, randomly selected category.

To display the category s `CategoryName` value, add a Label Web control to the page, set its `ID` property to `CategoryNameLabel`, and clear out its `Text` property. To programmatically retrieve the data from a SqlDataSource control, we need to invoke its `Select()` method. The `Select()` method expects a single input parameter of type `DataSourceSelectArguments`, which specifies how the data should be messaged before being returned. This can include instructions on sorting and filtering the data, and is used by the data Web controls when sorting or paging through the data from a SqlDataSource control. For our example, though, we don t need the data to be modified before being returned, and therefore will pass in the `DataSourceSelectArguments.Empty` object.

The `Select()` method returns an object that implements `IEnumerable`. The precise type returned depends on the value of the SqlDataSource control s `DataSourceMode` property. As discussed in the previous tutorial, this property can be set to a value of either `DataSet` or `DataReader`. If set to `DataSet`, the `Select()` method returns a DataView object; if set to `DataReader`, it returns an object that implements `IDataReader`. Since the `RandomCategoryDataSource` SqlDataSource has its `DataSourceMode` property set to `DataSet` (the default), we will be working with a DataView object.

The following code illustrates how to retrieve the records from the `RandomCategoryDataSource` SqlDataSource as a DataView as well as how to read the `CategoryName` column value from the first DataView row:

```
protected void Page_Load(object sender, EventArgs e)
{
    // Get the data from the SqlDataSource as a DataView
    DataView randomCategoryView =
        (DataView)RandomCategoryDataSource.Select(DataSourceSelectArguments.Empty);
    if (randomCategoryView.Count > 0)
    {
        // Assign the CategoryName value to the Label
        CategoryNameLabel.Text =
            string.Format("Here are Products in the {0} Category...",
                randomCategoryView[0]["CategoryName"].ToString());
    }
}
```

`randomCategoryView[0]` returns the first `DataRowView` in the DataView. `randomCategoryView[0]["CategoryName"]` returns the value of the `CategoryName` column in this first row. Note that the DataView is loosely-typed. To reference a particular column value we need to pass in the name of the column as a string (CategoryName, in this case). Figure 13 shows the message displayed in the `CategoryNameLabel` when viewing the page. Of course, the actual category name displayed is randomly selected by the `RandomCategoryDataSource` SqlDataSource on each visit to the page (including postbacks).

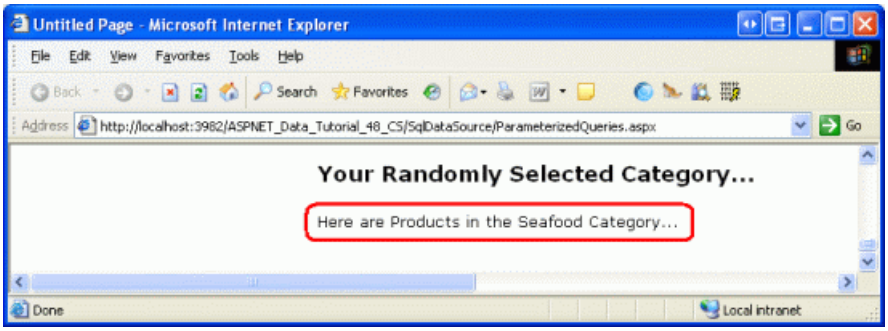


Figure 13: The Randomly Selected Category s Name is Displayed (Click to view full-size image)

> **NOTE**
>
> If the SqlDataSource control s `DataSourceMode` property had been set to `DataReader`, the return value from the `Select()` method would have needed to be cast to `IDataReader`. To read the `CategoryName` column value from the first row, we d use code like:

```
if (randomCategoryReader.Read())
{
   string categoryName = randomCategoryReader["CategoryName"].ToString();
   ...
}
```

With the SqlDataSource randomly selecting a category, we re ready to add the GridView that lists the category s products.

> **NOTE**
>
> Rather than using a Label Web control to display the category s name, we could have added a FormView or DetailsView to the page, binding it to the SqlDataSource. Using the Label, however, allowed us to explore how to programmatically invoke the SqlDataSource s `Select()` statement and work with its resulting data in code.

Step 5: Assigning Parameter Values Programmatically

All of the examples we ve seen so far in this tutorial have used either a hard-coded parameter value or one taken from one of the pre-defined parameter sources (a querystring value, a Web control on the page, and so on). However, the SqlDataSource control s parameters can also be set programmatically. To complete our current example, we need a SqlDataSource that returns all of the products belonging to a specified category. This SqlDataSource will have a `CategoryID` parameter whose value needs to be set based on the `CategoryID` column value returned by the `RandomCategoryDataSource` SqlDataSource in the `Page_Load` event handler.

Start by adding a GridView to the page and bind it to a new SqlDataSource named `ProductsByCategoryDataSource`. Much like we did in Step 3, configure the SqlDataSource so that it invokes the `GetProductsByCategory` stored procedure. Leave the Parameter source drop-down list set to None, but do not enter a default value, as we will set this default value programmatically.

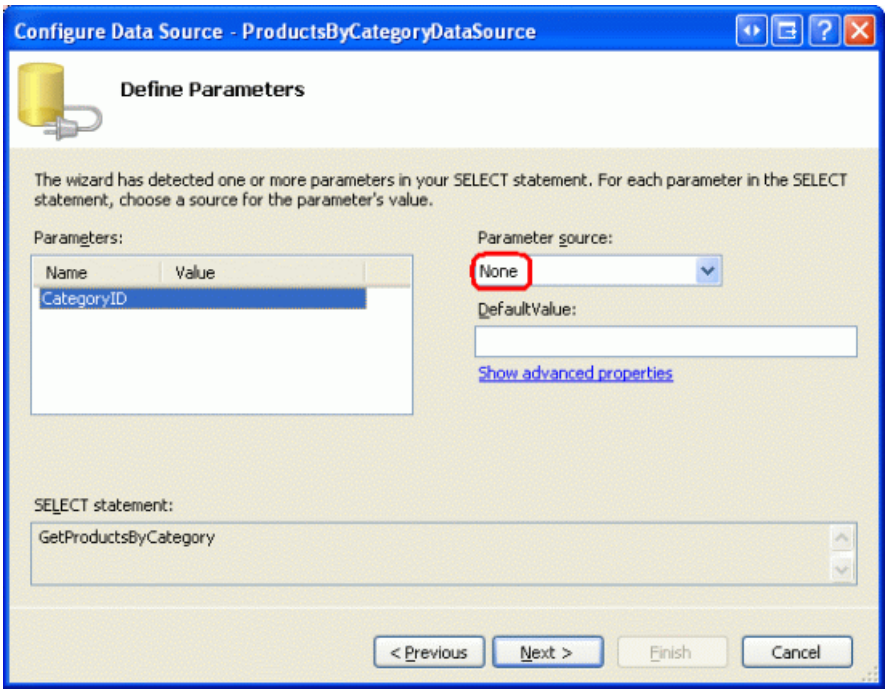


Figure 14: Do Not Specify a Parameter Source or Default Value (Click to view full-size image)

After completing the SqlDataSource wizard, the resulting declarative markup should look similar to the following:

```
<asp:SqlDataSource ID="ProductsByCategoryDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand="GetProductsByCategory" SelectCommandType="StoredProcedure">
    <SelectParameters>
        <asp:Parameter Name="CategoryID" Type="Int32" />
    </SelectParameters>
</asp:SqlDataSource>
```

We can assign the `DefaultValue` of the `CategoryID` parameter programmatically in the `Page_Load` event handler:

```
// Assign the ProductsByCategoryDataSource's
// CategoryID parameter's DefaultValue property
ProductsByCategoryDataSource.SelectParameters["CategoryID"].DefaultValue =
    randomCategoryView[0]["CategoryID"].ToString();
```

With this addition, the page includes a GridView that shows the products associated with the randomly selected category.

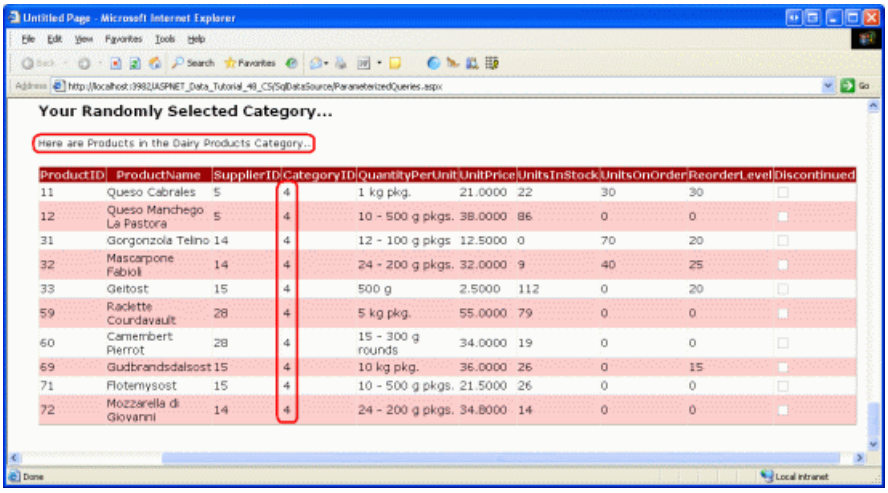


Figure 15: Do Not Specify a Parameter Source or Default Value (Click to view full-size image)

Summary

The SqlDataSource enables page developers to define parameterized queries whose parameter values can be hard-coded, pulled from pre-defined parameter sources, or assigned programmatically. In this tutorial we saw how to craft a parameterized query from the Configure Data Source wizard for both ad-hoc SQL queries and stored procedures. We also looked at using hard-coded parameter sources, a Web control as a parameter source, and programmatically specifying the parameter value.

Like with the ObjectDataSource, the SqlDataSource also provides capabilities to modify its underlying data. In the next tutorial we'll look at how to define `INSERT`, `UPDATE`, and `DELETE` statements with the SqlDataSource. Once these statements have been added, we can utilize the built-in inserting, editing, and deleting features inherent to the GridView, DetailsView, and FormView controls.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Scott Clyde, Randell Schmidt, and Ken Pespisa. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Inserting, Updating, and Deleting Data with the SqlDataSource (C#)

2/20/2020 • 16 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In previous tutorials we learned how the ObjectDataSource control allowed for inserting, updating, and deleting of data. The SqlDataSource control supports the same operations, but the approach is different, and this tutorial shows how to configure the SqlDataSource to insert, update, and delete data.

Introduction

As discussed in An Overview of Inserting, Updating, and Deleting, the GridView control provides built-in updating and deleting capabilities, while the DetailsView and FormView controls include inserting support along with editing and deleting functionality. These data modification capabilities can be plugged directly into a data source control without a line of code needing to be written. An Overview of Inserting, Updating, and Deleting examined using the ObjectDataSource to facilitate inserting, updating, and deleting with the GridView, DetailsView, and FormView controls. Alternatively, the SqlDataSource can be used in place of the ObjectDataSource.

Recall that to support inserting, updating, and deleting, with the ObjectDataSource we needed to specify the object layer methods to invoke to perform the insert, update, or delete action. With the SqlDataSource, we need to provide `INSERT`, `UPDATE`, and `DELETE` SQL statements (or stored procedures) to execute. As we'll see in this tutorial, these statements can be created manually or can be automatically generated by the SqlDataSource s Configure Data Source wizard.

> **NOTE**
>
> Since we ve already discussed the inserting, editing, and deleting capabilities of the GridView, DetailsView, and FormView controls, this tutorial will focus on configuring the SqlDataSource control to support these operations. If you need to brush up on implementing these features within the GridView, DetailsView, and FormView, return to the Editing, Inserting, and Deleting Data tutorials, starting with An Overview of Inserting, Updating, and Deleting.

Step 1: Specifying `INSERT`, `UPDATE`, and `DELETE` Statements

As we ve seen in the past two tutorials, to retrieve data from a SqlDataSource control we need to set two properties:

1. `ConnectionString`, which specifies what database to send the query to, and
2. `SelectCommand`, which specifies the ad-hoc SQL statement or stored procedure name to execute to return the results.

For `SelectCommand` values with parameters, the parameter values are specified via the SqlDataSource s `SelectParameters` collection and can include hard-coded values, common parameter source values (querystring fields, session variables, Web control values, and so on), or can be programmatically assigned. When the SqlDataSource control s `Select()` method is invoked either programmatically or automatically from a data Web control a connection to the database is established, the parameter values are assigned to the query, and the command is shuttled off to the database. The results are then returned as either a DataSet or DataReader,

depending on the value of the control s `DataSourceMode` property.

Along with selecting data, the SqlDataSource control can be used to insert, update, and delete data by supplying `INSERT`, `UPDATE`, and `DELETE` SQL statements in much the same way. Simply assign the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties the `INSERT`, `UPDATE`, and `DELETE` SQL statements to execute. If the statements have parameters (as they most always will), include them in the `InsertParameters`, `UpdateParameters`, and `DeleteParameters` collections.

Once an `InsertCommand`, `UpdateCommand`, or `DeleteCommand` value has been specified, the Enable Inserting, Enable Editing, or Enable Deleting option in the corresponding data Web control s smart tag will become available. To illustrate this, let s take an example from the `Querying.aspx` page we created in the Querying Data with the SqlDataSource Control tutorial and augment it to include delete capabilities.

Start by opening the `InsertUpdateDelete.aspx` and `Querying.aspx` pages from the `SqlDataSource` folder. From the Designer on the `Querying.aspx` page, select the SqlDataSource and GridView from the first example (the `ProductsDataSource` and `GridView1` controls). After selecting the two controls, go to the Edit menu and choose Copy (or just hit Ctrl+C). Next, go to the Designer of `InsertUpdateDelete.aspx` and paste in the controls. After you have moved the two controls over to `InsertUpdateDelete.aspx`, test out the page in a browser. You should see the values of the `ProductID`, `ProductName`, and `UnitPrice` columns for all of the records in the `Products` database table.

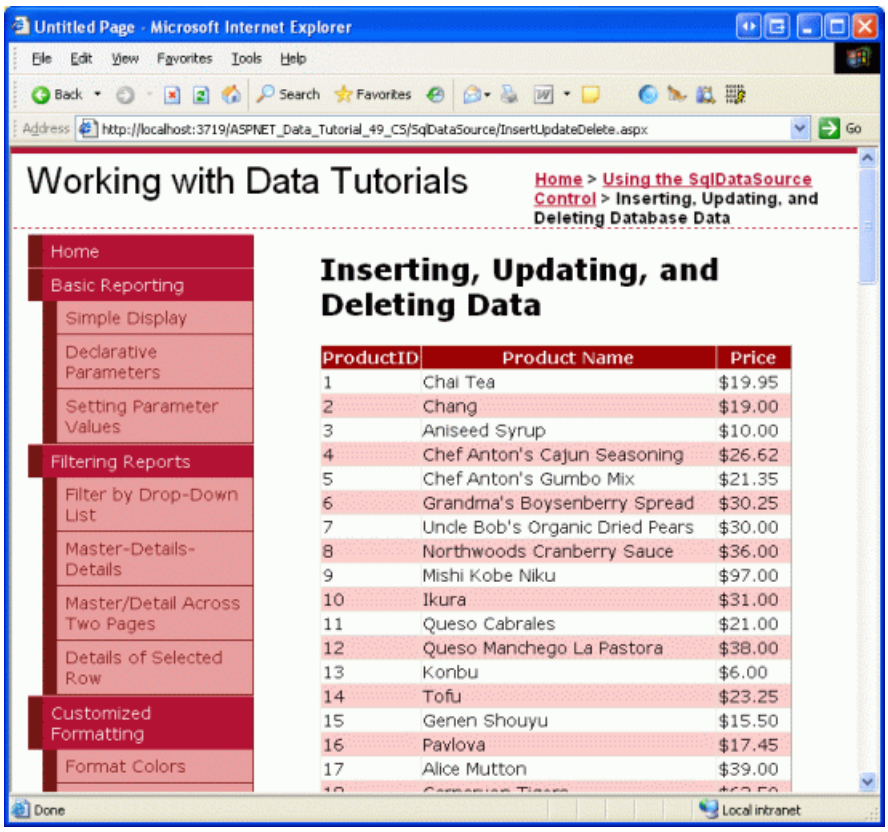


Figure 1: All of the Products are Listed, Ordered by `ProductID` (Click to view full-size image)

Adding the SqlDataSource s `DeleteCommand` and `DeleteParameters` Properties

At this point we have a SqlDataSource that simply returns all of the records from the `Products` table and a GridView that renders this data. Our goal is to extend this example to allow for the user to delete products via the GridView. To accomplish this we need to specify values for the SqlDataSource control s `DeleteCommand` and `DeleteParameters` properties and then configure the GridView to support deleting.

The `DeleteCommand` and `DeleteParameters` properties can be specified in a number of ways:

- Through the declarative syntax
- From the Properties window in the Designer
- From the Specify a custom SQL statement or stored procedure screen in the Configure Data Source wizard
- Via the Advanced button in the Specify columns from a table of view screen in the Configure Data Source wizard, which will actually automatically generate the `DELETE` SQL statement and parameter collection used in the `DeleteCommand` and `DeleteParameters` properties

We'll examine how to automatically have the `DELETE` statement created in Step 2. For now, let s use the Properties window in the Designer, although the Configure Data Source wizard or declarative syntax option would work just as well.

From the Designer in `InsertUpdateDelete.aspx`, click on the `ProductsDataSource` SqlDataSource and then bring up the Properties window (from the View menu, choose Properties window, or simply hit F4). Select the DeleteQuery property, which will bring up a set of ellipses.

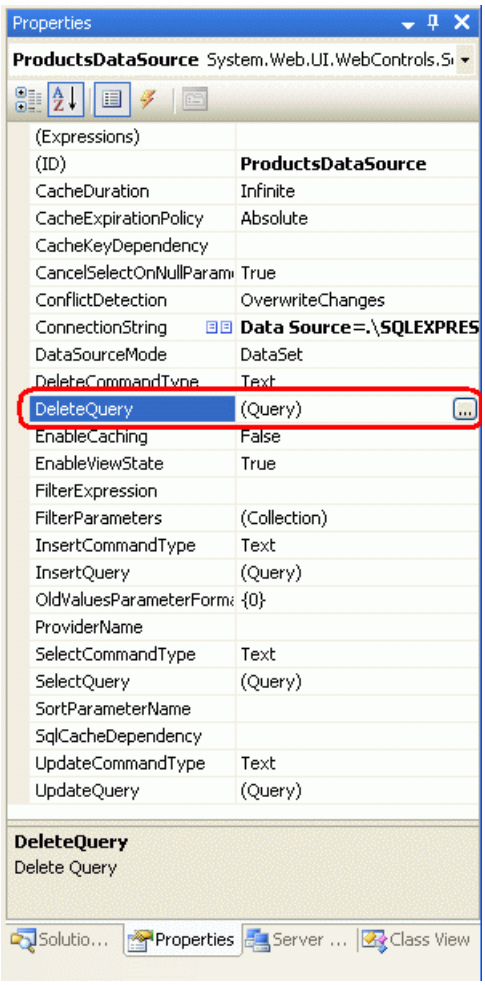


Figure 2: Select the DeleteQuery Property from the Properties Window

> **NOTE**
>
> The SqlDataSource doesn t have a DeleteQuery property. Rather, DeleteQuery is a combination of the `DeleteCommand` and `DeleteParameters` properties and is only listed in the Properties window when viewing the window through the Designer. If you are looking at the Properties window in the Source view, you'll find the `DeleteCommand` property instead.

Click the ellipses in the DeleteQuery property to bring up the Command and Parameter Editor dialog box (see Figure 3). From this dialog box you can specify the `DELETE` SQL statement and specify the parameters. Enter the following query into the `DELETE` command textbox (either manually or using the Query Builder, if you prefer):

```
DELETE FROM Products
WHERE ProductID = @ProductID
```

Next, click the Refresh Parameters button to add the `@ProductID` parameter to the list of parameters below.

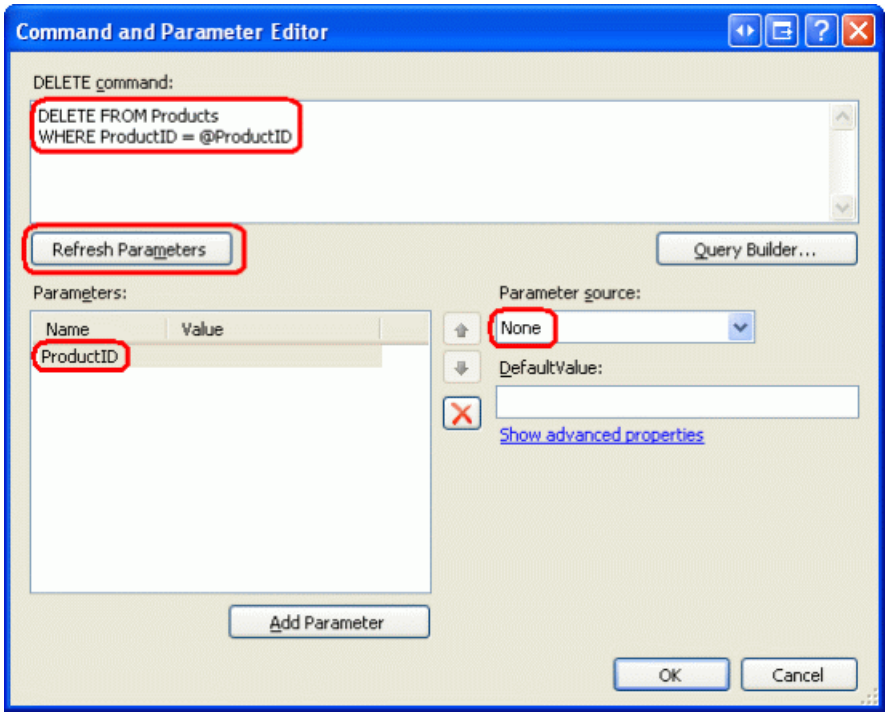


Figure 3: Select the DeleteQuery Property from the Properties Window (Click to view full-size image)

Do *not* provide a value for this parameter (leave its Parameter source at None). Once we add deleting support to the GridView, the GridView will automatically supply this parameter value, using the value of its `DataKeys` collection for the row whose Delete button was clicked.

> **NOTE**
>
> The parameter name used in the `DELETE` query *must* be the same as the name of the `DataKeyNames` value in the GridView, DetailsView, or FormView. That is, the parameter in the `DELETE` statement is purposefully named `@ProductID` (instead of, say, `@ID`), because the primary key column name in the Products table (and therefore the DataKeyNames value in the GridView) is `ProductID` .

If the parameter name and `DataKeyNames` value doesn t match, the GridView cannot automatically assign the parameter the value from the `DataKeys` collection.

After entering the delete-related information into the Command and Parameter Editor dialog box click OK and go to the Source view to examine the resulting declarative markup:

```
<asp:SqlDataSource ID="ProductsDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand=
        "SELECT [ProductID], [ProductName], [UnitPrice] FROM [Products]"
    DeleteCommand="DELETE FROM Products WHERE ProductID = @ProductID">
    <DeleteParameters>
        <asp:Parameter Name="ProductID" />
    </DeleteParameters>
</asp:SqlDataSource>
```

Note the addition of the `DeleteCommand` property as well as the `<DeleteParameters>` section and the Parameter

object named `productID`.

Configuring the GridView for Deleting

With the `DeleteCommand` property added, the GridView s smart tag now contains the Enable Deleting option. Go ahead and check this checkbox. As discussed in An Overview of Inserting, Updating, and Deleting, this causes the GridView to add a CommandField with its `ShowDeleteButton` property set to `true`. As Figure 4 shows, when the page is visited through a browser a Delete button is included. Test this page out by deleting some products.

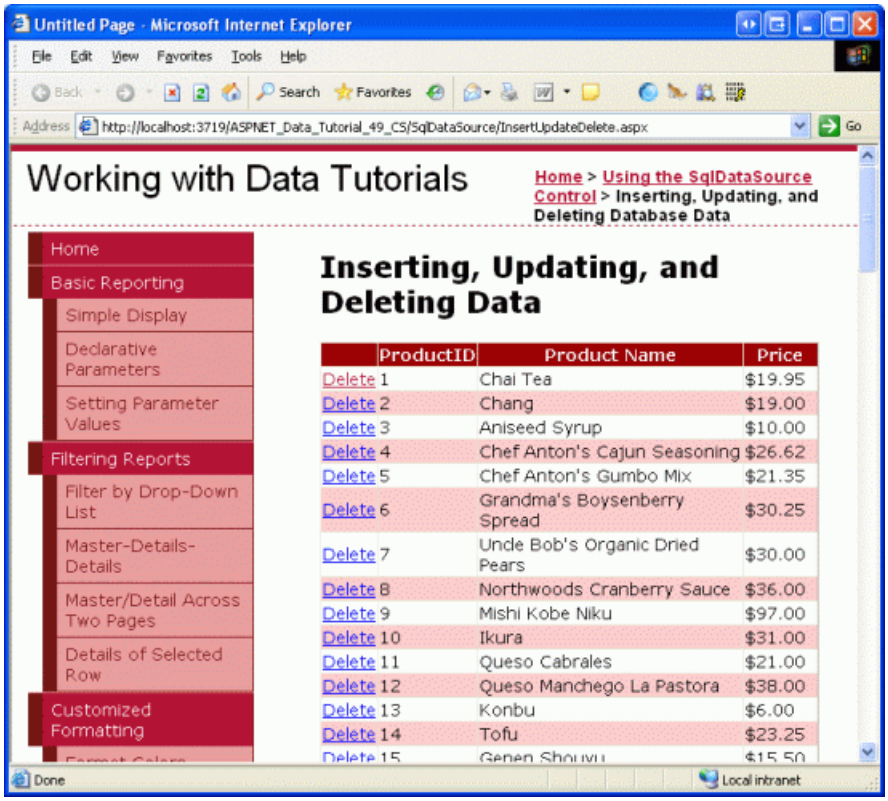


Figure 4: Each GridView Row Now Includes a Delete Button (Click to view full-size image)

Upon clicking a Delete button, a postback occurs, the GridView assigns the `ProductID` parameter the value of the `DataKeys` collection value for the row whose Delete button was clicked, and invokes the SqlDataSource s `Delete()` method. The SqlDataSource control then connects to the database and executes the `DELETE` statement. The GridView then rebinds to the SqlDataSource, getting back and displaying the current set of products (which no longer includes the just-deleted record).

> **NOTE**
>
> Since the GridView uses its `DataKeys` collection to populate the SqlDataSource parameters, it s vital that the GridView s `DataKeyNames` property be set to the column(s) that constitute the primary key and that the SqlDataSource s `SelectCommand` returns these columns. Moreover, it s important that the parameter name in the SqlDataSource s `DeleteCommand` is set to `@ProductID`. If the `DataKeyNames` property is not set or the parameter is not named `@ProductsID`, clicking the Delete button will cause a postback, but won't actually delete any record.

Figure 5 depicts this interaction graphically. Refer back to the Examining the Events Associated with Inserting, Updating, and Deleting tutorial for a more detailed discussion on the chain of events associated with inserting, updating, and deleting from a data Web control.

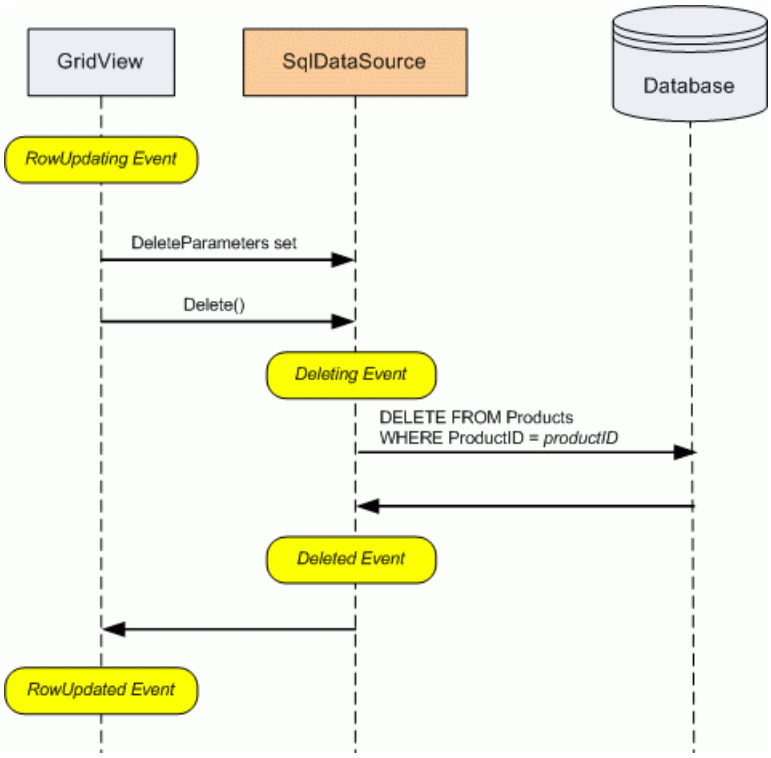


Figure 5: Clicking the Delete Button in the GridView Invokes the SqlDataSource s `Delete()` Method

Step 2: Automatically Generating the `INSERT`, `UPDATE`, and `DELETE` Statements

As Step 1 examined, `INSERT`, `UPDATE`, and `DELETE` SQL statements can be specified through the Properties window or the control s declarative syntax. However, this approach requires that we manually write out the SQL statements by hand, which can be monotonous and error-prone. Fortunately, the Configure Data Source wizard provides an option to have the `INSERT`, `UPDATE`, and `DELETE` statements automatically generated when using the Specify columns from a table of view screen.

Let s explore this automatic generation option. Add a DetailsView to the Designer in `InsertUpdateDelete.aspx` and set its `ID` property to `ManageProducts`. Next, from the DetailsView s smart tag, choose to create a new data source and create a SqlDataSource named `ManageProductsDataSource`.

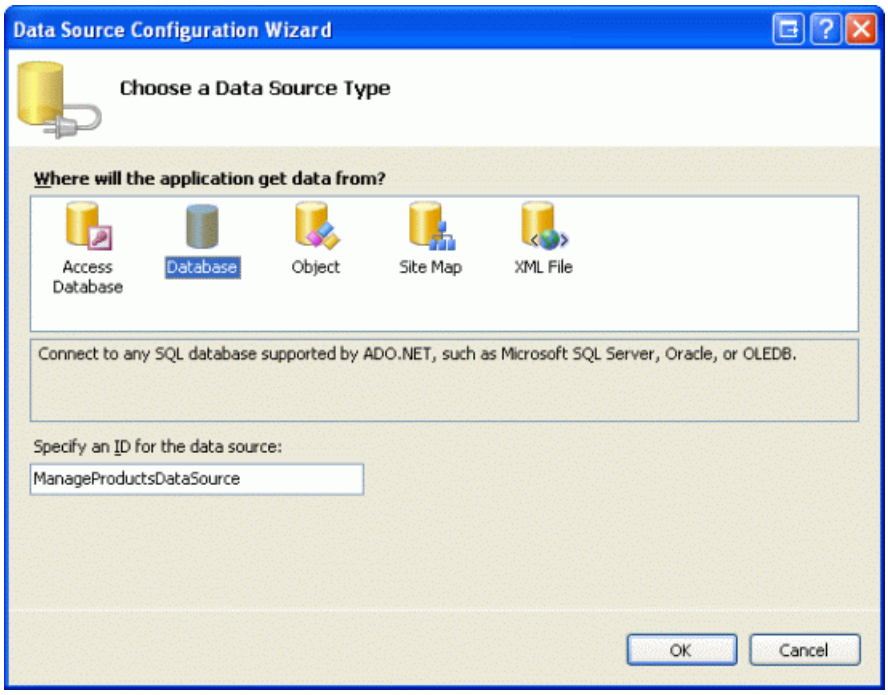


Figure 6: Create a New SqlDataSource Named `ManageProductsDataSource` (Click to view full-size image)

From the Configure Data Source wizard, opt to use the `NORTHWINDConnectionString` connection string and click Next. From the Configure the Select Statement screen, leave the Specify columns from a table or view radio button selected and pick the `Products` table from the drop-down list. Select the `ProductID`, `ProductName`, `UnitPrice`, and `Discontinued` columns from the checkbox list.

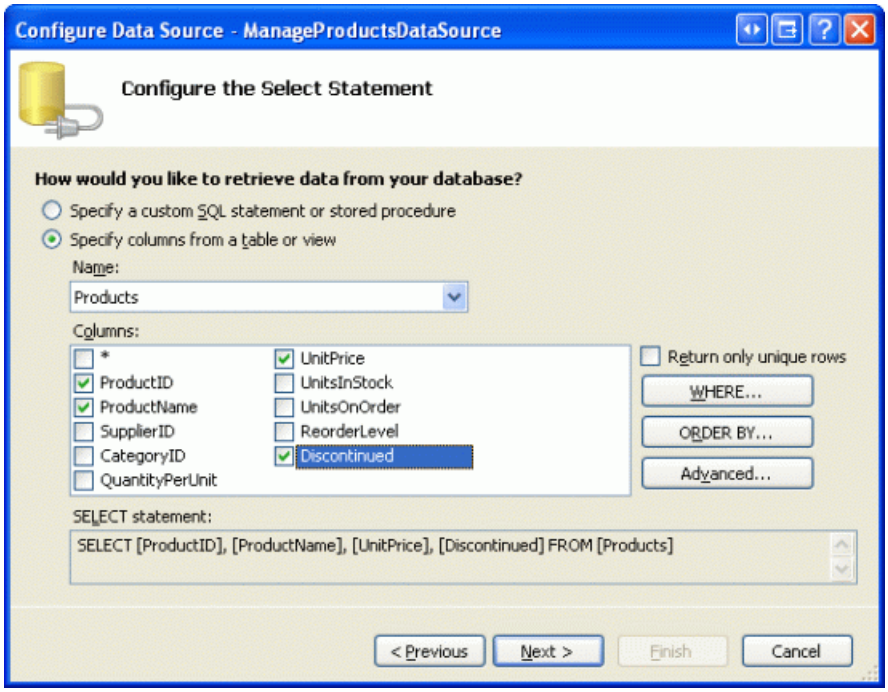


Figure 7: Using the `Products` Table, Return the `ProductID`, `ProductName`, `UnitPrice`, and `Discontinued` Columns (Click to view full-size image)

To automatically generate `INSERT`, `UPDATE`, and `DELETE` statements based on the selected table and columns, click the Advanced button and check the Generate `INSERT`, `UPDATE`, and `DELETE` statements checkbox.

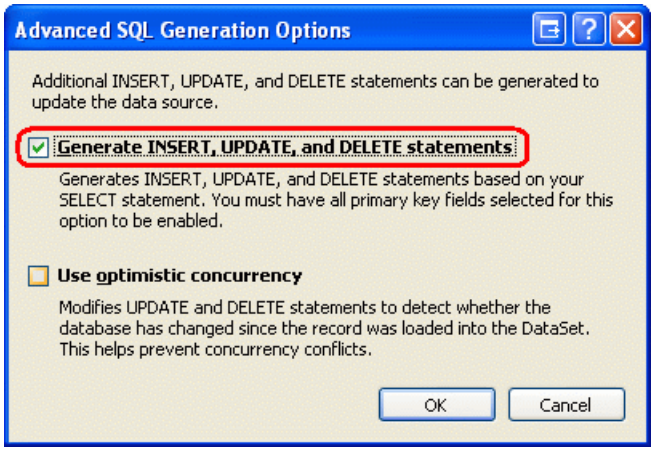


Figure 8: Check the Generate `INSERT`, `UPDATE`, and `DELETE` statements Checkbox

The Generate `INSERT`, `UPDATE`, and `DELETE` statements checkbox will only be checkable if the table selected has a primary key and the primary key column (or columns) are included in the list of returned columns. The Use optimistic concurrency checkbox, which becomes selectable once the Generate `INSERT`, `UPDATE`, and `DELETE` statements checkbox has been checked, will augment the `WHERE` clauses in the resulting `UPDATE` and `DELETE` statements to provide optimistic concurrency control. For now, leave this checkbox unchecked; we'll examine optimistic concurrency with the SqlDataSource control in the next tutorial.

After checking the Generate `INSERT`, `UPDATE`, and `DELETE` statements checkbox, click OK to return to the Configure Select Statement screen, then click Next, and then Finish, to complete the Configure Data Source

wizard. Upon completing the wizard, Visual Studio will add BoundFields to the DetailsView for the `ProductID`, `ProductName`, and `UnitPrice` columns and a CheckBoxField for the `Discontinued` column. From the DetailsView s smart tag, check the Enable Paging option so that the user visiting this page can step through the products. Also clear out the DetailsView s `Width` and `Height` properties.

Notice that the smart tag has the Enable Inserting, Enable Editing, and Enable Deleting options available. This is because the SqlDataSource contains values for its `InsertCommand`, `UpdateCommand`, and `DeleteCommand`, as the following declarative syntax shows:

```
<asp:DetailsView ID="ManageProducts" runat="server" AllowPaging="True"
    AutoGenerateRows="False" DataKeyNames="ProductID"
    DataSourceID="ManageProductsDataSource" EnableViewState="False">
    <Fields>
        <asp:BoundField DataField="ProductID" HeaderText="ProductID"
            InsertVisible="False" ReadOnly="True" SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName" HeaderText="ProductName"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
            SortExpression="UnitPrice" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Fields>
</asp:DetailsView>
<asp:SqlDataSource ID="ManageProductsDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    DeleteCommand=
        "DELETE FROM [Products] WHERE [ProductID] = @ProductID"
    InsertCommand=
        "INSERT INTO [Products] ([ProductName], [UnitPrice], [Discontinued])
         VALUES (@ProductName, @UnitPrice, @Discontinued)"
    SelectCommand=
        "SELECT [ProductID], [ProductName], [UnitPrice], [Discontinued]
         FROM [Products]"
    UpdateCommand=
        "UPDATE [Products] SET [ProductName] = @ProductName,
         [UnitPrice] = @UnitPrice, [Discontinued] = @Discontinued
         WHERE [ProductID] = @ProductID">
    <DeleteParameters>
        <asp:Parameter Name="ProductID" Type="Int32" />
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="ProductName" Type="String" />
        <asp:Parameter Name="UnitPrice" Type="Decimal" />
        <asp:Parameter Name="Discontinued" Type="Boolean" />
        <asp:Parameter Name="ProductID" Type="Int32" />
    </UpdateParameters>
    <InsertParameters>
        <asp:Parameter Name="ProductName" Type="String" />
        <asp:Parameter Name="UnitPrice" Type="Decimal" />
        <asp:Parameter Name="Discontinued" Type="Boolean" />
    </InsertParameters>
</asp:SqlDataSource>
```

Note how the SqlDataSource control has had values automatically set for its `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties. The set of columns referenced in the `InsertCommand` and `UpdateCommand` properties are based on those in the `SELECT` statement. That is, rather than having *every* Products column in the `InsertCommand` and `UpdateCommand`, there are only those columns specified in the `SelectCommand` (less `ProductID`, which is omitted because it s an `IDENTITY` column, whose value cannot be changed when edited and which is automatically assigned when inserting). Moreover, for each parameter in the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties there are corresponding parameters in the `InsertParameters`, `UpdateParameters`, and `DeleteParameters` collections.

To turn on the DetailsView s data modification features, check the Enable Inserting, Enable Editing, and Enable Deleting options in its smart tag. This adds a CommandField with its `ShowInsertButton`, `ShowEditButton`, and `ShowDeleteButton` properties set to `true`.

Visit the page in a browser and note the Edit, Delete, and New buttons included in the DetailsView. Clicking the Edit button turns the DetailsView into edit mode, which displays each BoundField whose `ReadOnly` property is set to `false` (the default) as a TextBox, and the CheckBoxField as a checkbox.

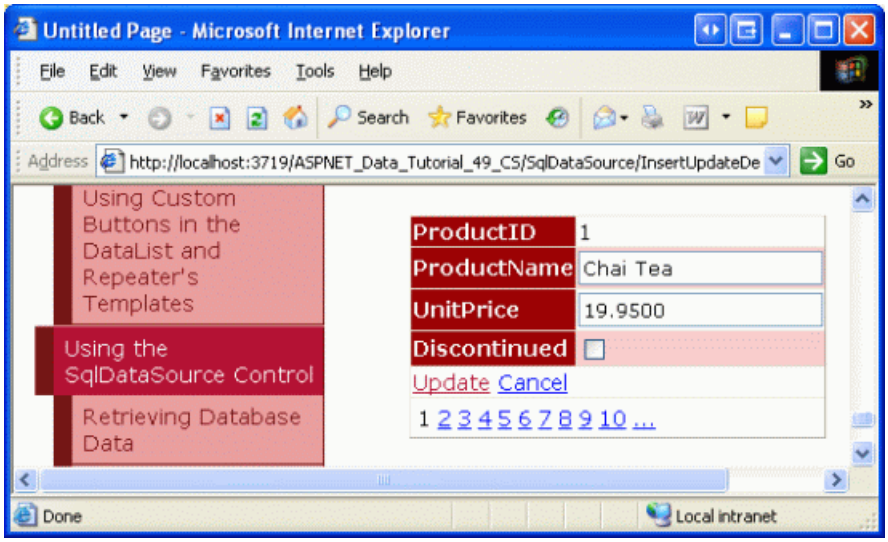


Figure 9: The DetailsView s Default Editing Interface (Click to view full-size image)

Similarly, you can delete the currently selected product or add a new product to the system. Since the `InsertCommand` statement only works with the `ProductName`, `UnitPrice`, and `Discontinued` columns, the other columns have either `NULL` or their default value assigned by the database upon insert. Just like with the ObjectDataSource, if the `InsertCommand` is missing any database table columns that don t allow `NULL` s and don t have a default value, a SQL error will occur when attempting to execute the `INSERT` statement.

> **NOTE**
>
> The DetailsView s inserting and editing interfaces lack any sort of customization or validation. To add validation controls or to customize the interfaces, you need to convert the BoundFields to TemplateFields. Refer to the Adding Validation Controls to the Editing and Inserting Interfaces and Customizing the Data Modification Interface tutorials for more information.

Also, keep in mind that for updating and deleting, the DetailsView uses the current product s `DataKey` value, which is only present if the `DataKeyNames` property is configured. If editing or deleting appears to have no effect, ensure that the `DataKeyNames` property is set.

Limitations of Automatically Generating SQL Statements

Since the Generate `INSERT`, `UPDATE`, and `DELETE` statements option is only available when picking columns from a table, for more complex queries you will have to write your own `INSERT`, `UPDATE`, and `DELETE` statements like we did in Step 1. Commonly, SQL `SELECT` statements use `JOIN` s to bring back data from one or more lookup tables for display purposes (such as bringing back the `Categories` table s `CategoryName` field when displaying product information). At the same time, we might want to allow the user to edit, update, or insert data into the core table (`Products`, in this case).

While the `INSERT`, `UPDATE`, and `DELETE` statements can be entered manually, consider the following time-saving tip. Initially setup the SqlDataSource so that it pulls back data just from the `Products` table. Use the Configure Data Source wizard s Specify columns from a table or view screen so that you can automatically generate the `INSERT`, `UPDATE`, and `DELETE` statements. Then, after completing the wizard, choose to configure the SelectQuery

from the Properties window (or, alternatively, go back to the Configure Data Source wizard, but use the Specify a custom SQL statement or stored procedure option). Then update the `SELECT` statement to include the `JOIN` syntax. This technique offers the time-saving benefits of the automatically generated SQL statements and allows for a more customized `SELECT` statement.

Another limitation of automatically generating the `INSERT`, `UPDATE`, and `DELETE` statements is that the columns in the `INSERT` and `UPDATE` statements are based on the columns returned by the `SELECT` statement. We may need to update or insert more or fewer fields, however. For example, in the example from Step 2, maybe we want to have the `UnitPrice` BoundField be read-only. In that case, it shouldn't appear in the `UpdateCommand`. Or we may want to set the value of a table field that does not appear in the GridView. For example, when adding a new record we may want the `QuantityPerUnit` value set to TODO .

If such customizations are required, you need to make them manually, either through the Properties window, the Specify a custom SQL statement or stored procedure option in the wizard, or via the declarative syntax.

> **NOTE**
> When adding parameters that do not have corresponding fields in the data Web control, keep in mind that these parameters values will need to be assigned values in some manner. These values can be: hard-coded directly in the `InsertCommand` or `UpdateCommand`; can come from some pre-defined source (the querystring, session state, Web controls on the page, and so on); or can be assigned programmatically, as we saw in the preceding tutorial.

Summary

In order for the data Web controls to utilize their built-in inserting, editing, and deleting capabilities, the data source control they are bound to must offer such functionality. For the SqlDataSource, this means that `INSERT`, `UPDATE`, and `DELETE` SQL statements must be assigned to the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties. These properties, and the corresponding parameters collections, can be added manually or generated automatically through the Configure Data Source wizard. In this tutorial we examined both techniques.

We examined using optimistic concurrency with the ObjectDataSource in the Implementing Optimistic Concurrency tutorial. The SqlDataSource control also provides optimistic concurrency support. As noted in Step 2, when automatically generating the `INSERT`, `UPDATE`, and `DELETE` statements, the wizard offers a Use optimistic concurrency option. As we'll see in the next tutorial, using optimistic concurrency with the SqlDataSource modifies the `WHERE` clauses in the `UPDATE` and `DELETE` statements to ensure that the values for the other columns haven't changed since the data was last displayed on the page.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Implementing Optimistic Concurrency with the SqlDataSource (C#)

2/20/2020 • 16 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

In this tutorial we review the essentials of optimistic concurrency control and then explore how to implement it using the SqlDataSource control.

Introduction

In the preceding tutorial we examined how to add inserting, updating, and deleting capabilities to the SqlDataSource control. In short, to provide these features we needed to specify the corresponding `INSERT`, `UPDATE`, or `DELETE` SQL statement in the control s `InsertCommand`, `UpdateCommand`, or `DeleteCommand` properties, along with the appropriate parameters in the `InsertParameters`, `UpdateParameters`, and `DeleteParameters` collections. While these properties and collections can be specified manually, the Configure Data Source wizard s Advanced button offers a Generate `INSERT`, `UPDATE`, and `DELETE` statements checkbox that will auto-create these statements based on the `SELECT` statement.

Along with the Generate `INSERT`, `UPDATE`, and `DELETE` statements checkbox, the Advanced SQL Generation Options dialog box includes a Use optimistic concurrency option (see Figure 1). When checked, the `WHERE` clauses in the autogenerated `UPDATE` and `DELETE` statements are modified to only perform the update or delete if the underlying database data hasn't been modified since the user last loaded the data into the grid.

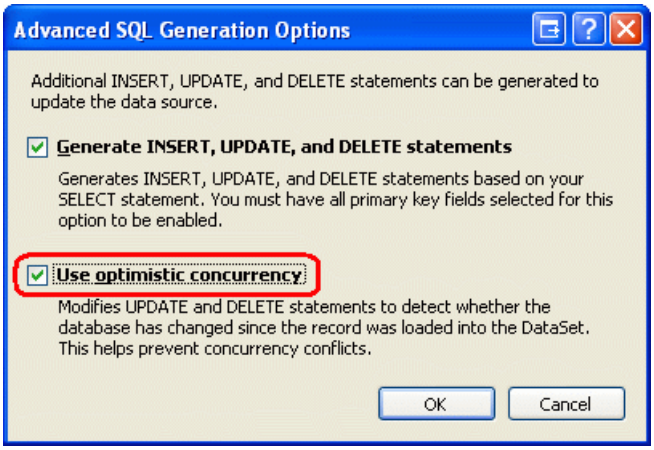


Figure 1: You Can Add Optimistic Concurrency Support from the Advanced SQL Generation Options Dialog Box

Back in the Implementing Optimistic Concurrency tutorial we examined the fundamentals of optimistic concurrency control and how to add it to the ObjectDataSource. In this tutorial we'll retouch on the essentials of optimistic concurrency control and then explore how to implement it using the SqlDataSource.

A Recap of Optimistic Concurrency

For web applications that allow multiple, simultaneous users to edit or delete the same data, there exists a possibility that one user may accidentally overwrite another s changes. In the Implementing Optimistic Concurrency tutorial I provided the following example:

Imagine that two users, Jisun and Sam, were both visiting a page in an application that allowed visitors to update and delete products through a GridView control. Both click the Edit button for Chai around the same time. Jisun changes the product name to Chai Tea and clicks the Update button. The net result is an `UPDATE` statement that is sent to the database, which sets *all* of the product s updateable fields (even though Jisun only updated one field, `ProductName`). At this point in time, the database has the values Chai Tea, the category Beverages, the supplier Exotic Liquids, and so on for this particular product. However, the GridView on Sam s screen still shows the product name in the editable GridView row as Chai. A few seconds after Jisun s changes have been committed, Sam updates the category to Condiments and clicks Update. This results in an `UPDATE` statement sent to the database that sets the product name to Chai, the `CategoryID` to the corresponding Condiments category ID, and so on. Jisun s changes to the product name have been overwritten.

Figure 2 illustrates this interaction.

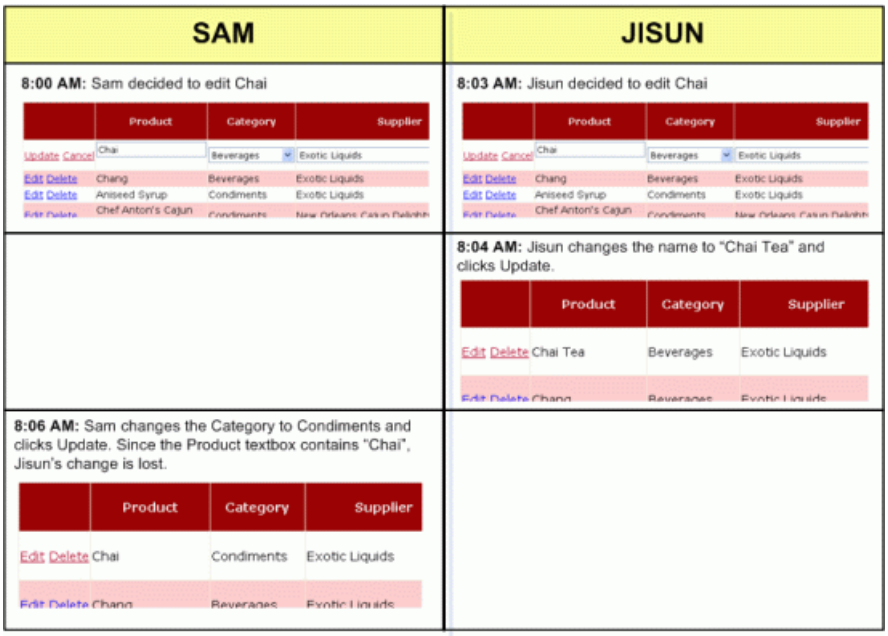


Figure 2: When Two Users Simultaneously Update a Record There s Potential for One User s Changes to Overwrite the Other s (Click to view full-size image)

To prevent this scenario from unfolding, a form of concurrency control must be implemented. Optimistic concurrency the focus of this tutorial works on the assumption that while there may be concurrency conflicts every now and then, the vast majority of the time such conflicts won't arise. Therefore, if a conflict does arise, optimistic concurrency control simply informs the user that their changes can t be saved because another user has modified the same data.

> **NOTE**
>
> For applications where it is assumed that there will be many concurrency conflicts or if such conflicts are not tolerable, then pessimistic concurrency control can be used instead. Refer back to the Implementing Optimistic Concurrency tutorial for a more thorough discussion on pessimistic concurrency control.

Optimistic concurrency control works by ensuring that the record being updated or deleted has the same values as it did when the updating or deleting process started. For example, when clicking the Edit button in an editable GridView, the record s values are read from the database and displayed in TextBoxes and other Web controls. These original values are saved by the GridView. Later, after the user makes her changes and clicks the Update button, the `UPDATE` statement used must take into account the original values plus the new values and only update the underlying database record if the original values that the user started editing are identical to the values still in the database. Figure 3 depicts this sequence of events.

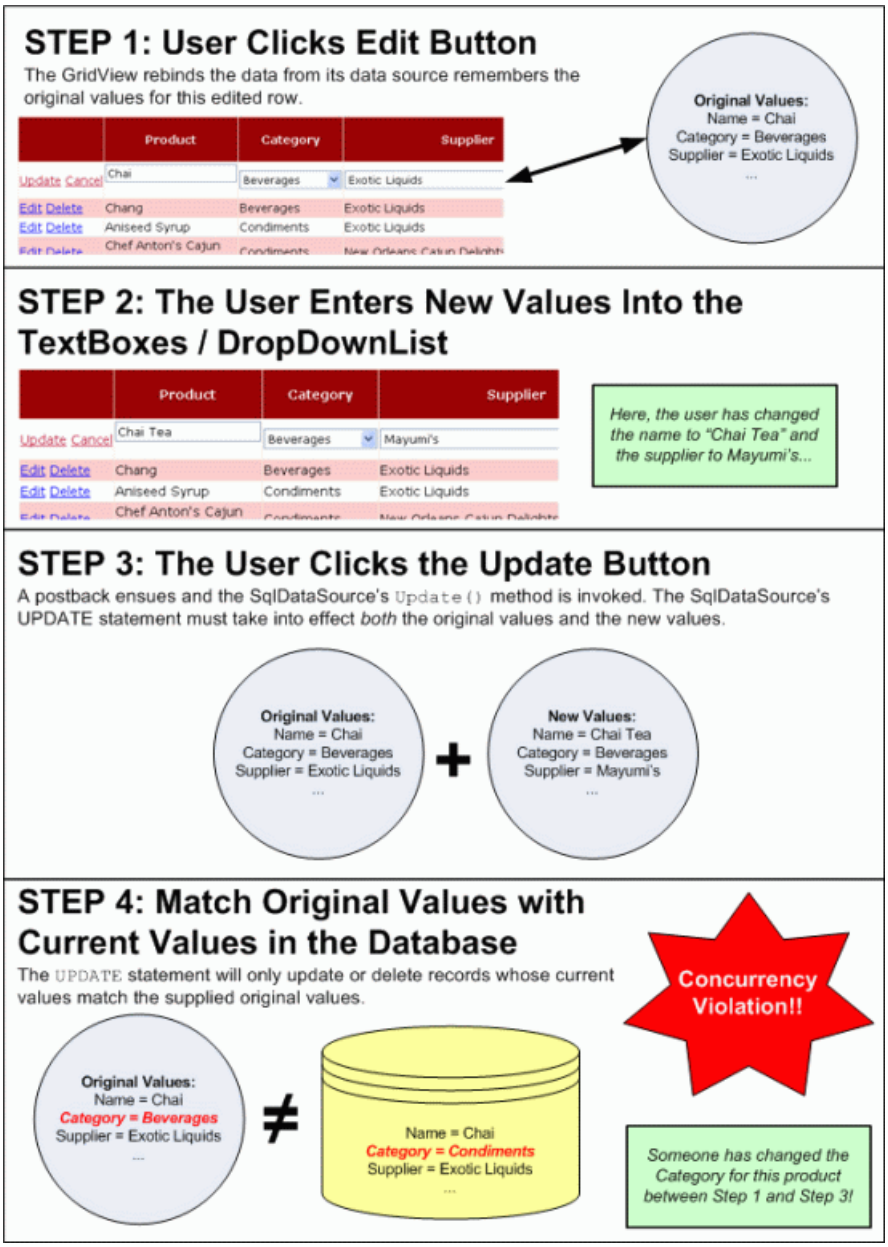


Figure 3: For the Update or Delete to Succeed, the Original Values Must Be Equal to the Current Database Values (Click to view full-size image)

There are various approaches to implementing optimistic concurrency (see Peter A. Bromberg's Optimistic Concurrency Updating Logic for a brief look at a number of options). The technique used by the SqlDataSource (as well as by the ADO.NET Typed DataSets used in our Data Access Layer) augments the `WHERE` clause to include a comparison of all of the original values. The following `UPDATE` statement, for example, updates the name and price of a product only if the current database values are equal to the values that were originally retrieved when updating the record in the GridView. The `@ProductName` and `@UnitPrice` parameters contain the new values entered by the user, whereas `@original_ProductName` and `@original_UnitPrice` contain the values that were originally loaded into the GridView when the Edit button was clicked:

```
UPDATE Products SET
    ProductName = @ProductName,
    UnitPrice = @UnitPrice
WHERE
    ProductID = @original_ProductID AND
    ProductName = @original_ProductName AND
    UnitPrice = @original_UnitPrice
```

As we'll see in this tutorial, enabling optimistic concurrency control with the SqlDataSource is as simple as checking a checkbox.

Step 1: Creating a SqlDataSource that Supports Optimistic Concurrency

Start by opening the `OptimisticConcurrency.aspx` page from the `SqlDataSource` folder. Drag a SqlDataSource control from the Toolbox onto the Designer, settings its `ID` property to `ProductsDataSourceWithOptimisticConcurrency`. Next, click on the Configure Data Source link from the control s smart tag. From the first screen in the wizard, choose to work with the `NORTHWINDConnectionString` and click Next.

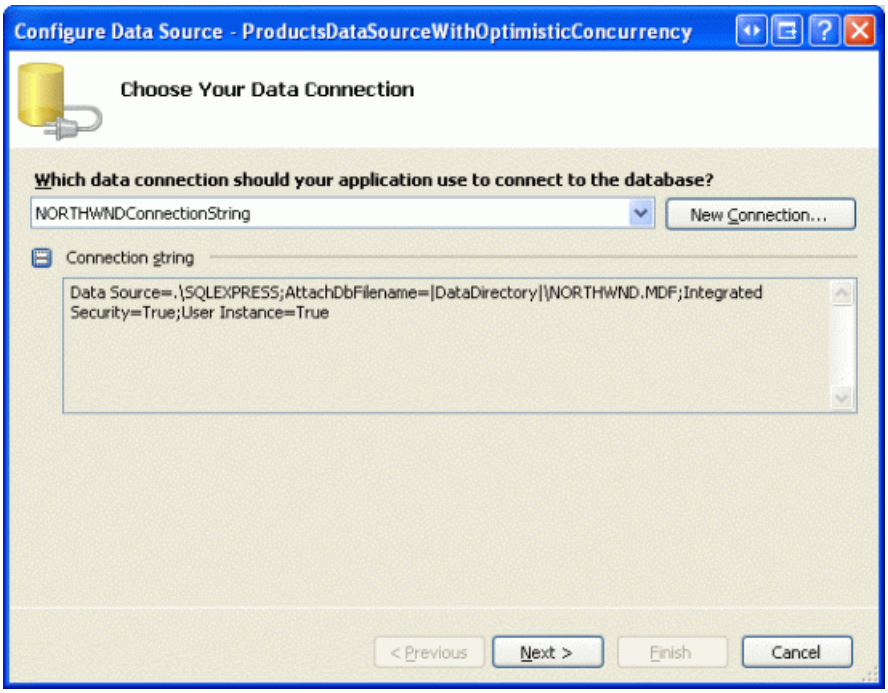


Figure 4: Choose to Work with the `NORTHWINDConnectionString` (Click to view full-size image)

For this example we'll be adding a GridView that enables users to edit the `Products` table. Therefore, from the Configure the Select Statement screen, choose the `Products` table from the drop-down list and select the `ProductID`, `ProductName`, `UnitPrice`, and `Discontinued` columns, as shown in Figure 5.

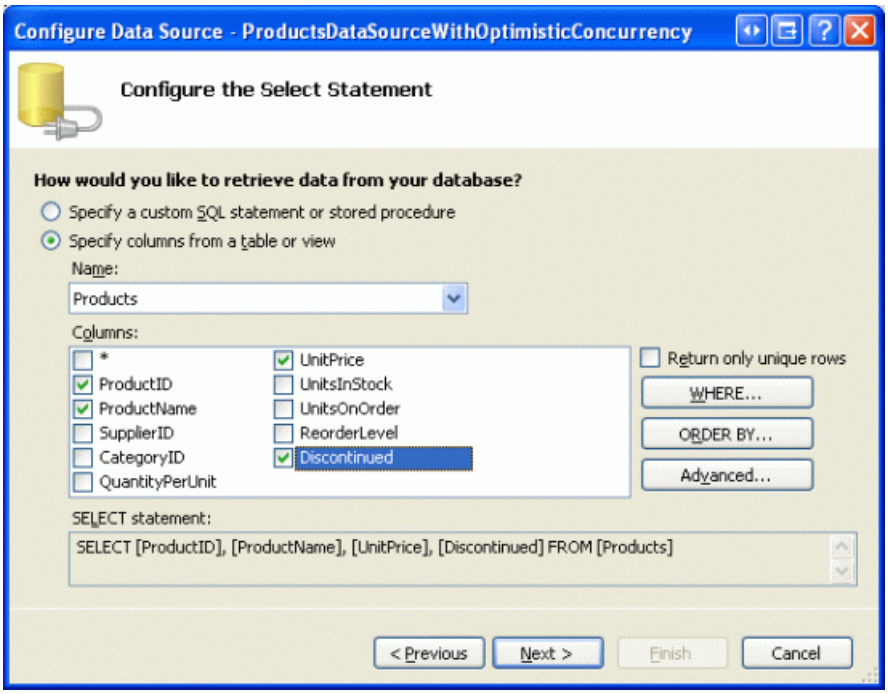


Figure 5: From the `Products` Table, Return the `ProductID`, `ProductName`, `UnitPrice`, and `Discontinued` Columns (Click to view full-size image)

After picking the columns, click the Advanced button to bring up the Advanced SQL Generation Options dialog box. Check the Generate `INSERT`, `UPDATE`, and `DELETE` statements and Use optimistic concurrency checkboxes and click OK (refer back to Figure 1 for a screenshot). Complete the wizard by clicking Next, then Finish.

After completing the Configure Data Source wizard, take a moment to examine the resulting `DeleteCommand` and `UpdateCommand` properties and the `DeleteParameters` and `UpdateParameters` collections. The easiest way to do this is to click on the Source tab in the lower left corner to see the page s declarative syntax. There you will find an `UpdateCommand` value of:

```
UPDATE [Products] SET
     [ProductName] = @ProductName,
     [UnitPrice] = @UnitPrice,
     [Discontinued] = @Discontinued
WHERE
     [ProductID] = @original_ProductID AND
     [ProductName] = @original_ProductName AND
     [UnitPrice] = @original_UnitPrice AND
     [Discontinued] = @original_Discontinued
```

With seven parameters in the `UpdateParameters` collection:

```
<asp:SqlDataSource ID="ProductsDataSourceWithOptimisticConcurrency"
    runat="server" ...>
    <DeleteParameters>
      ...
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="ProductName" Type="String" />
        <asp:Parameter Name="UnitPrice" Type="Decimal" />
        <asp:Parameter Name="Discontinued" Type="Boolean" />
        <asp:Parameter Name="original_ProductID" Type="Int32" />
        <asp:Parameter Name="original_ProductName" Type="String" />
        <asp:Parameter Name="original_UnitPrice" Type="Decimal" />
        <asp:Parameter Name="original_Discontinued" Type="Boolean" />
    </UpdateParameters>
    ...
</asp:SqlDataSource>
```

Similarly, the `DeleteCommand` property and `DeleteParameters` collection should look like the following:

```
DELETE FROM [Products]
WHERE
     [ProductID] = @original_ProductID AND
     [ProductName] = @original_ProductName AND
     [UnitPrice] = @original_UnitPrice AND
     [Discontinued] = @original_Discontinued
```

```
<asp:SqlDataSource ID="ProductsDataSourceWithOptimisticConcurrency"
    runat="server" ...>
    <DeleteParameters>
        <asp:Parameter Name="original_ProductID" Type="Int32" />
        <asp:Parameter Name="original_ProductName" Type="String" />
        <asp:Parameter Name="original_UnitPrice" Type="Decimal" />
        <asp:Parameter Name="original_Discontinued" Type="Boolean" />
    </DeleteParameters>
    <UpdateParameters>
        ...
    </UpdateParameters>
    ...
</asp:SqlDataSource>
```

In addition to augmenting the `WHERE` clauses of the `UpdateCommand` and `DeleteCommand` properties (and adding the additional parameters to the respective parameter collections), selecting the Use optimistic concurrency option adjusts two other properties:

- Changes the `ConflictDetection` property from `OverwriteChanges` (the default) to `CompareAllValues`
- Changes the `OldValuesParameterFormatString` property from {0} (the default) to original_{0} .

When the data Web control invokes the SqlDataSource s `Update()` or `Delete()` method, it passes in the original values. If the SqlDataSource s `ConflictDetection` property is set to `CompareAllValues`, these original values are added to the command. The `OldValuesParameterFormatString` property provides the naming pattern used for these original value parameters. The Configure Data Source wizard uses original_{0} and names each original parameter in the `UpdateCommand` and `DeleteCommand` properties and `UpdateParameters` and `DeleteParameters` collections accordingly.

> **NOTE**
> Since we re not using the SqlDataSource control s inserting capabilities, feel free to remove the `InsertCommand` property and its `InsertParameters` collection.

Correctly Handling `NULL` Values

Unfortunately, the augmented `UPDATE` and `DELETE` statements autogenerated by the Configure Data Source wizard when using optimistic concurrency do *not* work with records that contain `NULL` values. To see why, consider our SqlDataSource s `UpdateCommand`:

```
UPDATE [Products] SET
     [ProductName] = @ProductName,
     [UnitPrice] = @UnitPrice,
     [Discontinued] = @Discontinued
WHERE
     [ProductID] = @original_ProductID AND
     [ProductName] = @original_ProductName AND
     [UnitPrice] = @original_UnitPrice AND
     [Discontinued] = @original_Discontinued
```

The `UnitPrice` column in the `Products` table can have `NULL` values. If a particular record has a `NULL` value for `UnitPrice`, the `WHERE` clause portion `[UnitPrice] = @original_UnitPrice` will *always* evaluate to False because `NULL = NULL` always returns False. Therefore, records that contain `NULL` values cannot be edited or deleted, as the `UPDATE` and `DELETE` statements `WHERE` clauses won't return any rows to update or delete.

> **NOTE**
> This bug was first reported to Microsoft in June of 2004 in SqlDataSource Generates Incorrect SQL Statements and is reportedly scheduled to be fixed in the next version of ASP.NET.

To fix this, we have to manually update the `WHERE` clauses in both the `UpdateCommand` and `DeleteCommand` properties for **all** columns that can have `NULL` values. In general, change `[ColumnName] = @original_ColumnName` to:

```
(
   ([ColumnName] IS NULL AND @original_ColumnName IS NULL)
     OR
   ([ColumnName] = @original_ColumnName)
)
```

This modification can be made directly through the declarative markup, via the UpdateQuery or DeleteQuery options from the Properties window, or through the UPDATE and DELETE tabs in the Specify a custom SQL statement or stored procedure option in the Configure Data Source wizard. Again, this modification must be made for *every* column in the `UpdateCommand` and `DeleteCommand` s `WHERE` clause that can contain `NULL` values.

Applying this to our example results in the following modified `UpdateCommand` and `DeleteCommand` values:

```
UPDATE [Products] SET
     [ProductName] = @ProductName,
     [UnitPrice] = @UnitPrice,
     [Discontinued] = @Discontinued
WHERE
     [ProductID] = @original_ProductID AND
     [ProductName] = @original_ProductName AND
     (([UnitPrice] IS NULL AND @original_UnitPrice IS NULL)
        OR ([UnitPrice] = @original_UnitPrice)) AND
     [Discontinued] = @original_Discontinued
DELETE FROM [Products]
WHERE
     [ProductID] = @original_ProductID AND
     [ProductName] = @original_ProductName AND
     (([UnitPrice] IS NULL AND @original_UnitPrice IS NULL)
        OR ([UnitPrice] = @original_UnitPrice)) AND
     [Discontinued] = @original_Discontinued
```

Step 2: Adding a GridView with Edit and Delete Options

With the SqlDataSource configured to support optimistic concurrency, all that remains is to add a data Web control to the page that utilizes this concurrency control. For this tutorial, let s add a GridView that provides both edit and delete functionality. To accomplish this, drag a GridView from the Toolbox onto the Designer and set its `ID` to `Products`. From the GridView s smart tag, bind it to the `ProductsDataSourceWithOptimisticConcurrency` SqlDataSource control added in Step 1. Finally, check the Enable Editing and Enable Deleting options from the smart tag.

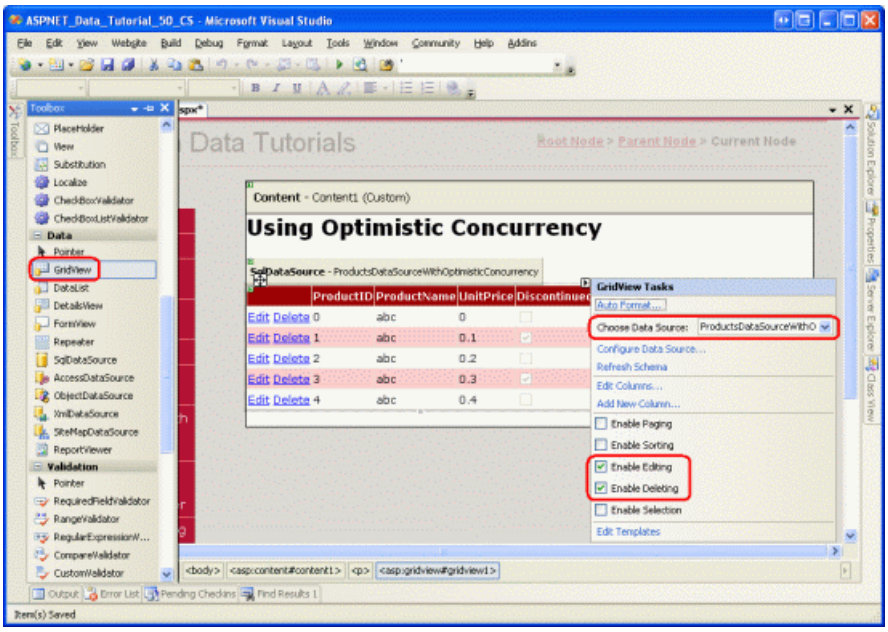


Figure 6: Bind the GridView to the SqlDataSource and Enable Editing and Deleting (Click to view full-size image)

After adding the GridView, configure its appearance by removing the `ProductID` BoundField, changing the `ProductName` BoundField s `HeaderText` property to Product, and updating the `UnitPrice` BoundField so that its `HeaderText` property is simply Price. Ideally, we d enhance the editing interface to include a RequiredFieldValidator for the `ProductName` value and a CompareValidator for the `UnitPrice` value (to ensure it s a properly formatted numeric value). Refer to the Customizing the Data Modification Interface tutorial for a more in-depth look at customizing the GridView s editing interface.

> **NOTE**
>
> The GridView s view state must be enabled since the original values passed from the GridView to the SqlDataSource are stored in view state.

After making these modifications to the GridView, the GridView and SqlDataSource declarative markup should look similar to the following:

```
<asp:SqlDataSource ID="ProductsDataSourceWithOptimisticConcurrency"
    runat="server" ConflictDetection="CompareAllValues"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    DeleteCommand=
        "DELETE FROM [Products]
         WHERE [ProductID] = @original_ProductID
         AND [ProductName] = @original_ProductName
         AND (([UnitPrice] IS NULL AND @original_UnitPrice IS NULL)
              OR ([UnitPrice] = @original_UnitPrice))
         AND [Discontinued] = @original_Discontinued"
    OldValuesParameterFormatString=
        "original_{0}"
    SelectCommand=
        "SELECT [ProductID], [ProductName], [UnitPrice], [Discontinued]
         FROM [Products]"
    UpdateCommand=
        "UPDATE [Products]
         SET [ProductName] = @ProductName, [UnitPrice] = @UnitPrice,
            [Discontinued] = @Discontinued
         WHERE [ProductID] = @original_ProductID
         AND [ProductName] = @original_ProductName
         AND (([UnitPrice] IS NULL AND @original_UnitPrice IS NULL)
            OR ([UnitPrice] = @original_UnitPrice))
        AND [Discontinued] = @original_Discontinued">
    <DeleteParameters>
        <asp:Parameter Name="original_ProductID" Type="Int32" />
        <asp:Parameter Name="original_ProductName" Type="String" />
        <asp:Parameter Name="original_UnitPrice" Type="Decimal" />
        <asp:Parameter Name="original_Discontinued" Type="Boolean" />
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="ProductName" Type="String" />
        <asp:Parameter Name="UnitPrice" Type="Decimal" />
        <asp:Parameter Name="Discontinued" Type="Boolean" />
        <asp:Parameter Name="original_ProductID" Type="Int32" />
        <asp:Parameter Name="original_ProductName" Type="String" />
        <asp:Parameter Name="original_UnitPrice" Type="Decimal" />
        <asp:Parameter Name="original_Discontinued" Type="Boolean" />
    </UpdateParameters>
</asp:SqlDataSource>
<asp:GridView ID="Products" runat="server"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ProductsDataSourceWithOptimisticConcurrency">
    <Columns>
        <asp:CommandField ShowDeleteButton="True" ShowEditButton="True" />
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
            SortExpression="UnitPrice" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
```

To see the optimistic concurrency control in action, open two browser windows and load the `OptimisticConcurrency.aspx` page in both. Click on the Edit buttons for the first product in both browsers. In one browser, change the product name and click Update. The browser will postback and the GridView will return to its pre-editing mode, showing the new product name for the record just edited.

In the second browser window, change the price (but leave the product name as its original value) and click Update. On postback, the grid returns to its pre-editing mode, but the change to the price is not recorded. The second browser shows the same value as the first one the new product name with the old price. The changes made in the second browser window were lost. Moreover, the changes were lost rather quietly, as there was no exception or message indicating that a concurrency violation just occurred.

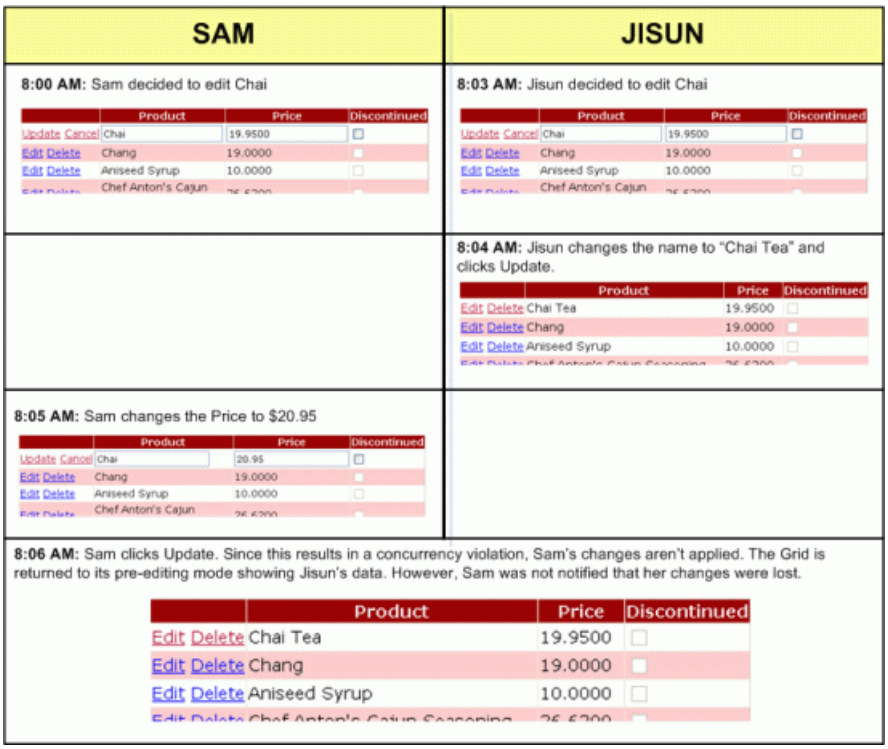


Figure 7: The Changes in the Second Browser Window Were Silently Lost (Click to view full-size image)

The reason why the second browser s changes were not committed was because the `UPDATE` statement s `WHERE` clause filtered out all records and therefore did not affect any rows. Let s look at the `UPDATE` statement again:

```
UPDATE [Products] SET
     [ProductName] = @ProductName,
     [UnitPrice] = @UnitPrice,
     [Discontinued] = @Discontinued
WHERE
     [ProductID] = @original_ProductID AND
     [ProductName] = @original_ProductName AND
     (([UnitPrice] IS NULL AND @original_UnitPrice IS NULL) OR
        ([UnitPrice] = @original_UnitPrice)) AND
     [Discontinued] = @original_Discontinued
```

When the second browser window updates the record, the original product name specified in the `WHERE` clause doesn t match up with the existing product name (since it was changed by the first browser). Therefore, the statement `[ProductName] = @original_ProductName` returns False, and the `UPDATE` does not affect any records.

> **NOTE**
> Delete works in the same manner. With two browser windows open, start by editing a given product with one, and then saving its changes. After saving the changes in the one browser, click the Delete button for the same product in the other. Since the original values don t match up in the `DELETE` statement s `WHERE` clause, the delete silently fails.

From the end user s perspective in the second browser window, after clicking the Update button the grid returns to the pre-editing mode, but their changes were lost. However, there s no visual feedback that their changes didn't stick. Ideally, if a user s changes are lost to a concurrency violation, we d notify them and, perhaps, keep the grid in edit mode. Let s look at how to accomplish this.

Step 3: Determining When a Concurrency Violation Has Occurred

Since a concurrency violation rejects the changes one has made, it would be nice to alert the user when a concurrency violation has occurred. To alert the user, let s add a Label Web control to the top of the page named

`ConcurrencyViolationMessage` whose `Text` property displays the following message: You have attempted to update or delete a record that was simultaneously updated by another user. Please review the other user's changes and then redo your update or delete. Set the Label control s `CssClass` property to Warning, which is a CSS class defined in `Styles.css` that displays text in a red, italic, bold, and large font. Finally, set the Label s `Visible` and `EnableViewState` properties to `false`. This will hide the Label except for only those postbacks where we explicitly set its `Visible` property to `true`.

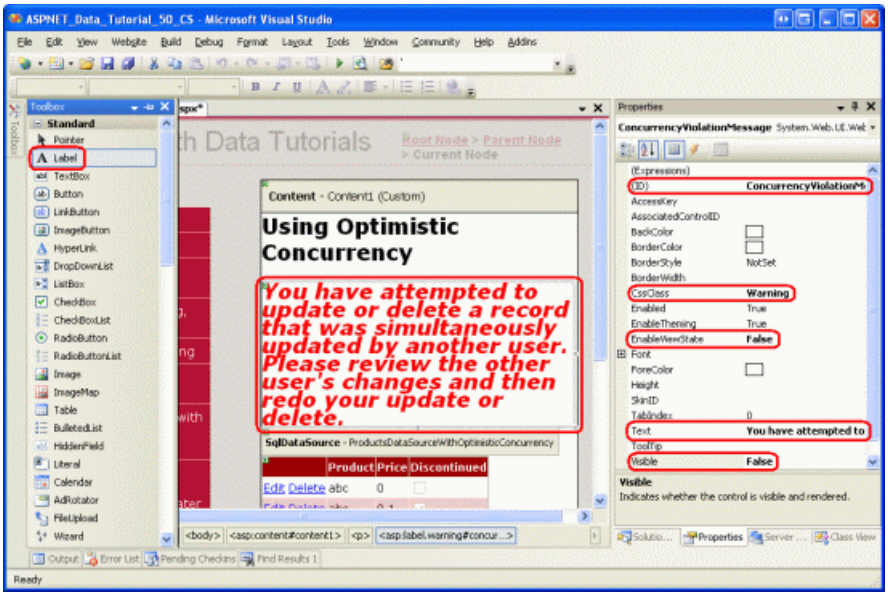


Figure 8: Add a Label Control to the Page to Display the Warning (Click to view full-size image)

When performing an update or delete, the GridView s `RowUpdated` and `RowDeleted` event handlers fire after its data source control has performed the requested update or delete. We can determine how many rows were affected by the operation from these event handlers. If zero rows were affected, we want to display the `ConcurrencyViolationMessage` Label.

Create an event handler for both the `RowUpdated` and `RowDeleted` events and add the following code:

```
protected void Products_RowUpdated(object sender, GridViewUpdatedEventArgs e)
{
    if (e.AffectedRows == 0)
    {
        ConcurrencyViolationMessage.Visible = true;
        e.KeepInEditMode = true;
        // Rebind the data to the GridView to show the latest changes
        Products.DataBind();
    }
}
protected void Products_RowDeleted(object sender, GridViewDeletedEventArgs e)
{
    if (e.AffectedRows == 0)
        ConcurrencyViolationMessage.Visible = true;
}
```

In both event handlers we check the `e.AffectedRows` property and, if it equals 0, set the `ConcurrencyViolationMessage` Label s `Visible` property to `true`. In the `RowUpdated` event handler, we also instruct the GridView to stay in edit mode by setting the `KeepInEditMode` property to true. In doing so, we need to rebind the data to the grid so that the other user s data is loaded into the editing interface. This is accomplished by calling the GridView s `DataBind()` method.

As Figure 9 shows, with these two event handlers, a very noticeable message is displayed whenever a concurrency violation occurs.

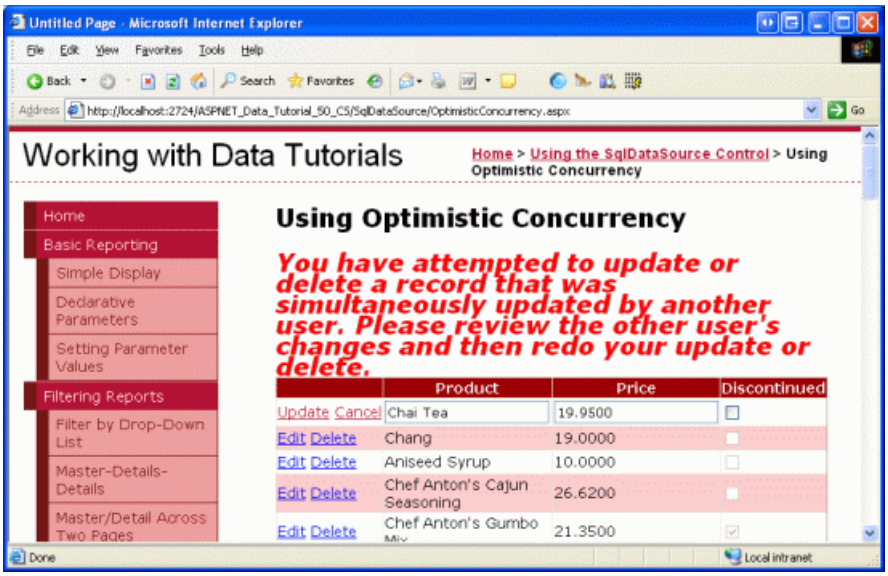


Figure 9: A Message is Displayed in the Face of a Concurrency Violation (Click to view full-size image)

Summary

When creating a web application where multiple, concurrent users may be editing the same data, it is important to consider concurrency control options. By default, the ASP.NET data Web controls and data source controls do not employ any concurrency control. As we saw in this tutorial, implementing optimistic concurrency control with the SqlDataSource is relatively quick and easy. The SqlDataSource handles most of the legwork for your adding augmented `WHERE` clauses to the autogenerated `UPDATE` and `DELETE` statements but there are a few subtleties in handling `NULL` value columns, as discussed in the Correctly Handling `NULL` Values section.

This tutorial concludes our examination of the SqlDataSource. Our remaining tutorials will return to working with data using the ObjectDataSource and tiered architecture.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Querying Data with the SqlDataSource Control (VB)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In the preceding tutorials we used the ObjectDataSource control to fully separate the presentation layer from the Data Access layer. Starting with this tutorial, we learn how the SqlDataSource control can be used for simple applications that do not require such a strict separation of presentation and data access.

Introduction

All of the tutorials we ve examined so far have used a tiered architecture consisting of presentation, Business Logic, and Data Access layers. The Data Access Layer (DAL) was crafted in the first tutorial (Creating a Data Access Layer) and the Business Logic Layer in the second (Creating a Business Logic Layer). Starting with the Displaying Data With the ObjectDataSource tutorial, we saw how to use ASP.NET 2.0 s new ObjectDataSource control to declaratively interface with the architecture from the presentation layer.

While all of the tutorials so far have used the architecture to work with data, it is also possible to access, insert, update, and delete database data directly from an ASP.NET page, bypassing the architecture. Doing so places the specific database queries and business logic directly in the web page. For sufficiently large or complex applications, designing, implementing, and using a tiered architecture is vitally important for the success, updatability, and maintainability of the application. Developing a robust architecture, however, may be unnecessary when creating exceedingly simple, one-off applications.

ASP.NET 2.0 provides five built-in data source controls SqlDataSource, AccessDataSource, ObjectDataSource, XmlDataSource, and SiteMapDataSource. The SqlDataSource can be used to access and modify data directly from a relational database, including Microsoft SQL Server, Microsoft Access, Oracle, MySQL, and others. In this tutorial and the next three, we'll examine how to work with the SqlDataSource control, exploring how to query and filter database data, as well as how to use the SqlDataSource to insert, update, and delete data.

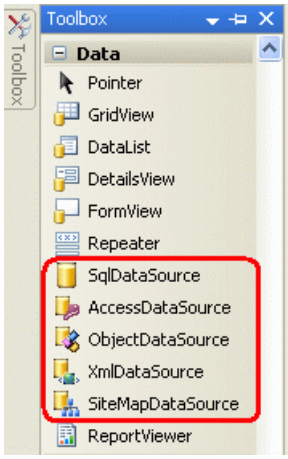


Figure 1: ASP.NET 2.0 Includes Five Built-In Data Source Controls

Comparing the ObjectDataSource and SqlDataSource

Conceptually, both the ObjectDataSource and SqlDataSource controls are simply proxies to data. As discussed in the Displaying Data With the ObjectDataSource tutorial, the ObjectDataSource has properties that indicate the

object type that provides the data and the methods to invoke to select, insert, update, and delete data from the underlying object type. Once the ObjectDataSource s properties have been configured, a data Web control such as a GridView, DetailsView, or DataList can be bound to the control, using the ObjectDataSource s `Select()`, `Insert()`, `Delete()`, and `Update()` methods to interact with the underlying architecture.

The SqlDataSource provides the same functionality, but operates against a relational database rather than an object library. With the SqlDataSource, we must specify the database connection string and the ad-hoc SQL queries or stored procedures to execute to insert, update, delete, and retrieve data. The SqlDataSource s `Select()`, `Insert()`, `Update()`, and `Delete()` methods, when invoked, connect to the specified database and issue the appropriate SQL query. As the following diagram illustrates, the these methods do the grunt work of connecting to a database, issuing a query, and returning the results.

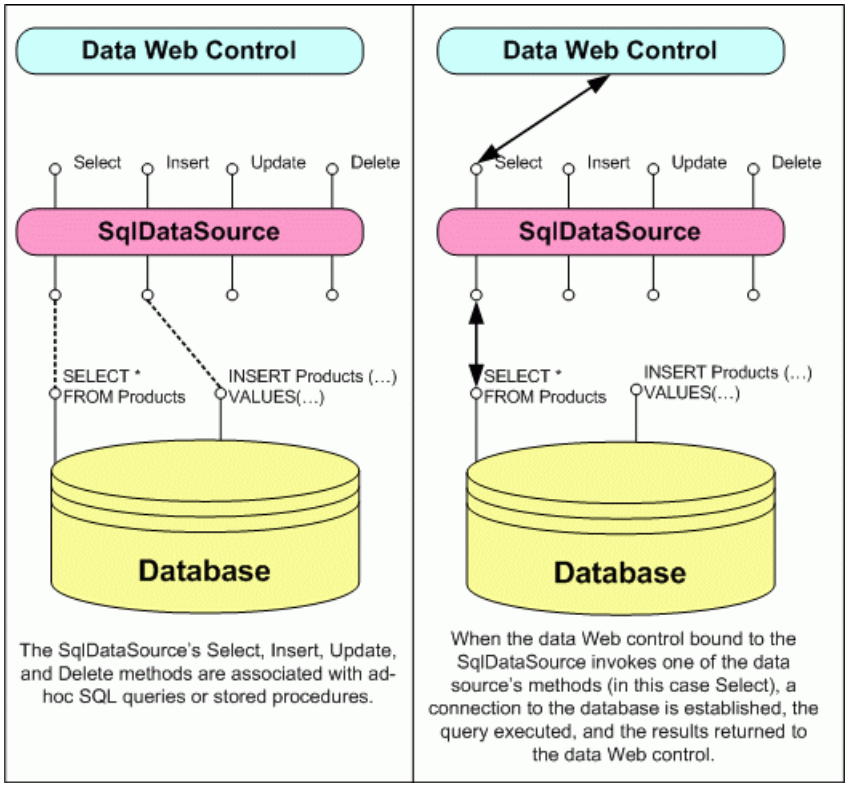


Figure 2: The SqlDataSource Serves as a Proxy to the Database

> **NOTE**
> In this tutorial we'll focus on retrieving data from the database. In the Inserting, Updating, and Deleting Data with the SqlDataSource Control tutorial, we'll see how to configure the SqlDataSource to support inserting, updating, and deleting.

The SqlDataSource and AccessDataSource Controls

In addition to the SqlDataSource control, ASP.NET 2.0 also includes an AccessDataSource control. These two different controls lead many developers new to ASP.NET 2.0 to suspect that the AccessDataSource control is designed to work exclusively with Microsoft Access with the SqlDataSource control designed to work exclusively with Microsoft SQL Server. While the AccessDataSource is designed to work specifically with Microsoft Access, the SqlDataSource control works with *any* relational database that can be accessed through .NET. This includes any OleDb- or ODBC-compliant data stores, such as Microsoft SQL Server, Microsoft Access, Oracle, Informix, MySQL, and PostgreSQL, among many others.

The sole difference between the AccessDataSource and SqlDataSource controls is how the database connection information is specified. The AccessDataSource control needs just the file path to the Access database file. The SqlDataSource, on the other hand, requires a complete connection string.

Step 1: Creating the SqlDataSource Web Pages

Before we start exploring how to work directly with database data using the SqlDataSource control, let s first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial and the next three. Start by adding a new folder named `SqlDataSource`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `Querying.aspx`
- `ParameterizedQueries.aspx`
- `InsertUpdateDelete.aspx`
- `OptimisticConcurrency.aspx`

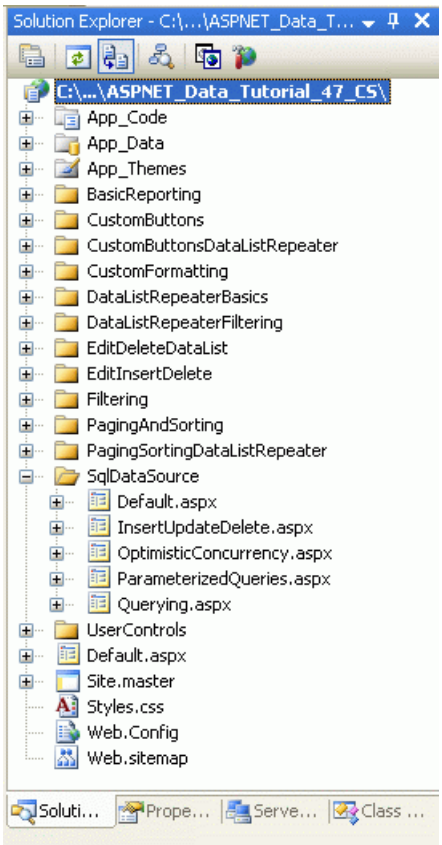


Figure 3: Add the ASP.NET Pages for the SqlDataSource-Related Tutorials

Like in the other folders, `Default.aspx` in the `SqlDataSource` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

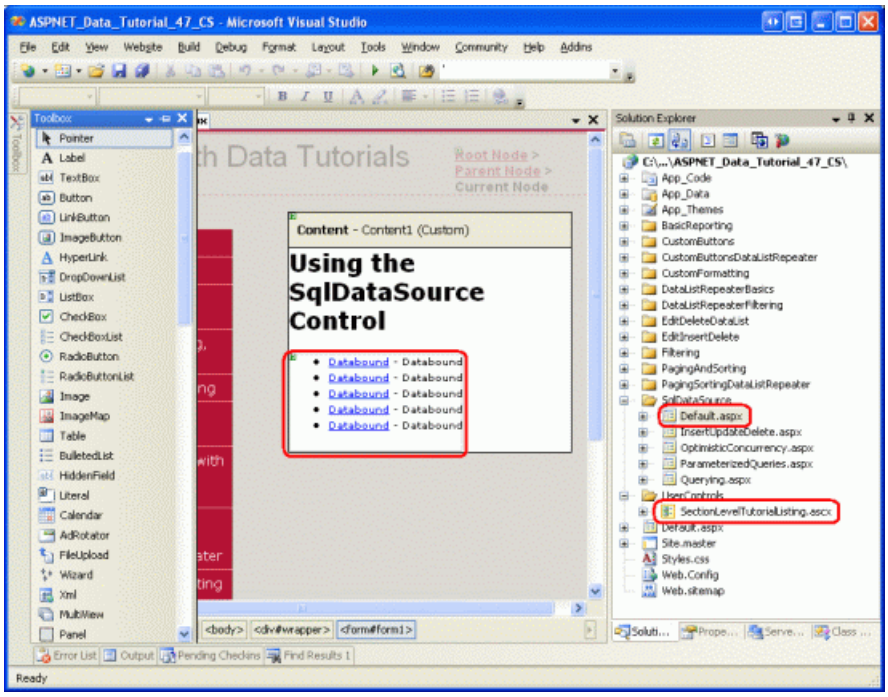


Figure 4: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add these four pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Adding Custom Buttons to the DataList and Repeater `<siteMapNode>`:

```
<siteMapNode url="~/SqlDataSource/Default.aspx"
    title="Using the SqlDataSource Control"
    description="Work directly with database data using the SqlDataSource control.">
    <siteMapNode url="~/SqlDataSource/Querying.aspx" title="Retrieving Database Data"
        description="Examines how to query data from a database that can then be
                     displayed  through a data Web control."/>
    <siteMapNode url="~/SqlDataSource/ParameterizedQueries.aspx"
        title="Parameterized Queries"
        description="Learn how to specify parameterized WHERE clauses in the
                     SqlDataSource's SELECT statement." />
    <siteMapNode url="~/SqlDataSource/InsertUpdateDelete.aspx"
        title="Inserting, Updating, and Deleting Database Data"
        description="See how to configure the SqlDataSource to include INSERT, UPDATE,
                      and DELETE statements." />
    <siteMapNode url="~/SqlDataSource/OptimisticConcurrency.aspx"
        title="Using Optimistic Concurrency"
        description="Explore how to augment the SqlDataSource to include support for
                     optimistic concurrency." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the editing, inserting, and deleting tutorials.

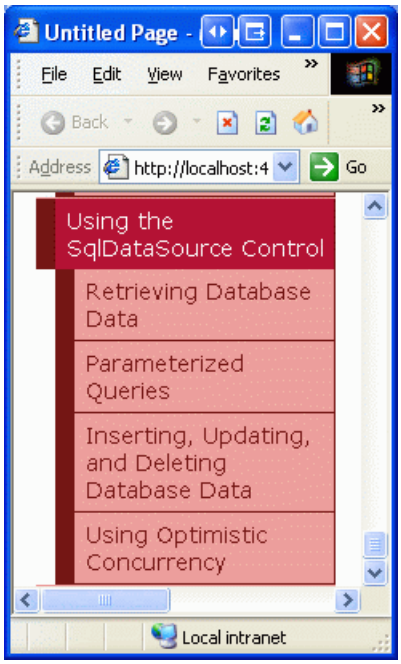


Figure 5: The Site Map Now Includes Entries for the SqlDataSource Tutorials

Step 2: Adding and Configuring the SqlDataSource Control

Start by opening the `Querying.aspx` page in the `SqlDataSource` folder and switch to Design view. Drag a SqlDataSource control from the Toolbox onto the Designer and set its `ID` to `ProductsDataSource`. As with the ObjectDataSource, the SqlDataSource does not produce any rendered output and therefore appears as a gray box on the design surface. To configure the SqlDataSource, click on the Configure Data Source link from the SqlDataSource s smart tag.

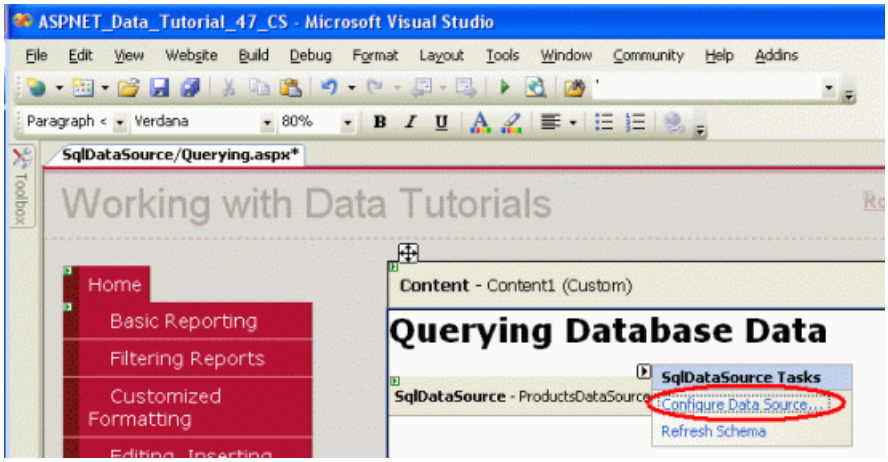


Figure 6: Click on the Configure Data Source Link from the SqlDataSource s Smart Tag

This brings up the SqlDataSource control s Configure Data Source wizard. While the wizard s steps differ from the ObjectDataSource control s, the end goal is the same to provide the details on how to retrieve, insert, update, and delete data through the data source. For the SqlDataSource this entails specifying the underlying database to use and providing the ad-hoc SQL statements or stored procedures.

The first wizard step prompts us for the database. The drop-down list includes those databases found in the web application s `App_Data` folder and those that have been added to the Data Connections node in the Server Explorer. Since we ve already added a connection string for the `NORTHWIND.MDF` database in the `App_Data` folder to our project s `Web.config` file, the drop-down list includes a reference to that connection string, `NORTHWINDConnectionString`. Choose this item from the drop-down list and click Next.

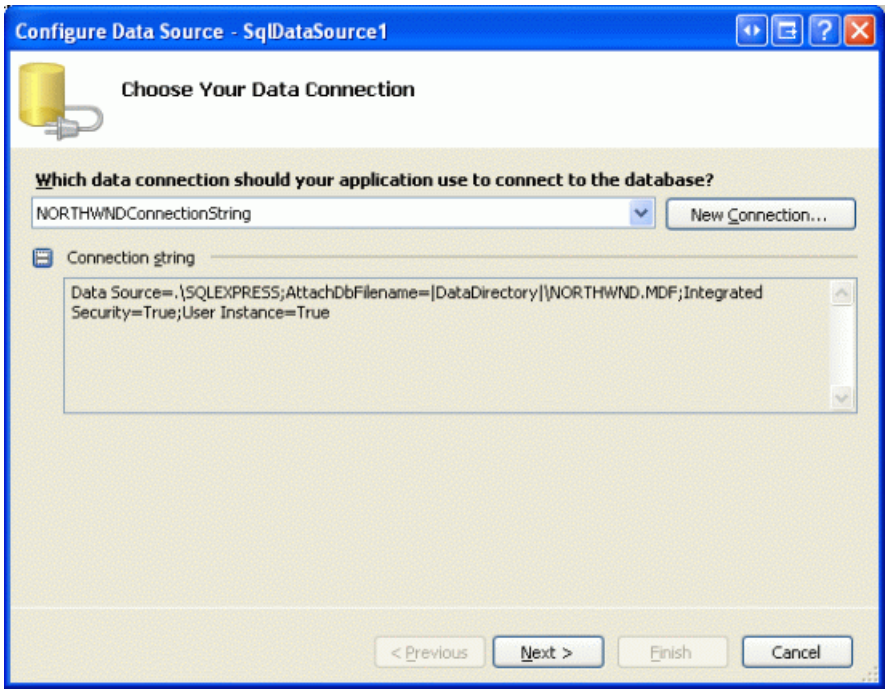


Figure 7: Choose the `NORTHWINDConnectionString` from the Drop-Down List

After choosing the database, the wizard asks for the query to return data. We can either specify the columns of a table or view to return or can enter a custom SQL statement or specify a stored procedure. You can toggle between this choice through the Specify a custom SQL statement or stored procedure and Specify columns from a table or view radio buttons.

> **NOTE**
> For this first example, let s use the Specify columns from a table or view option. We'll return to the wizard later in this tutorial and explore the Specify a custom SQL statement or stored procedure option.

Figure 8 shows the Configure the Select Statement screen when the Specify columns from a table or view radio button is selected. The drop-down list contains the set of tables and views in the Northwind database, with the selected table or view s columns displayed in the checkbox list below. For this example, let s return the `ProductID`, `ProductName`, and `UnitPrice` columns from the `Products` table. As Figure 8 shows, after making these selections the wizard shows the resulting SQL statement `SELECT [ProductID], [ProductName], [UnitPrice] FROM [Products]`.

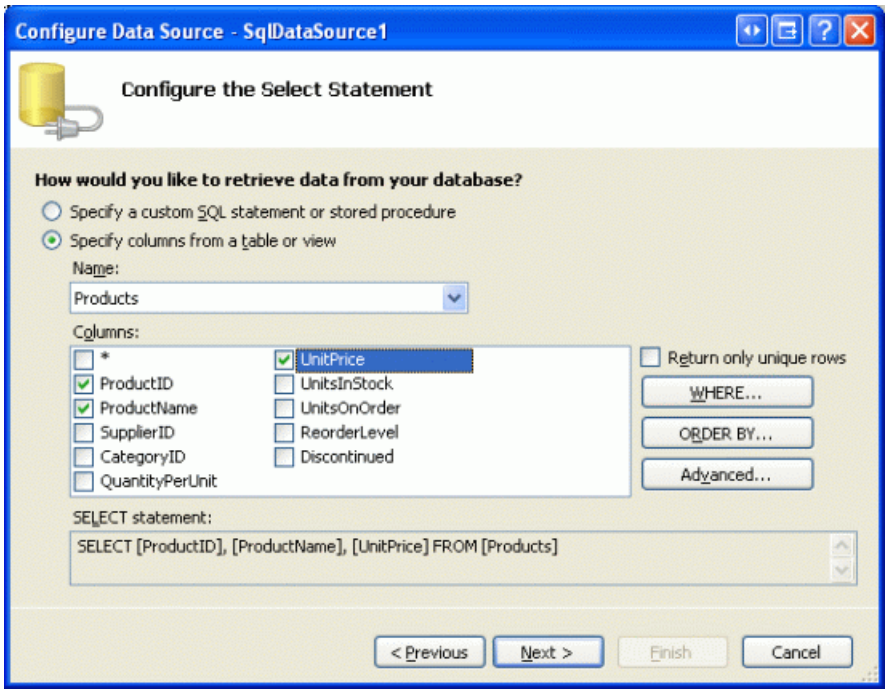


Figure 8: Return Data from the `Products` Table

Once you have configured the wizard to return the `ProductID`, `ProductName`, and `UnitPrice` columns from the `Products` table, click the Next button. This final screen provides an opportunity to examine the results of the query configured from the previous step. Clicking the Test Query button executes the configured `SELECT` statement and displays the results in a grid.

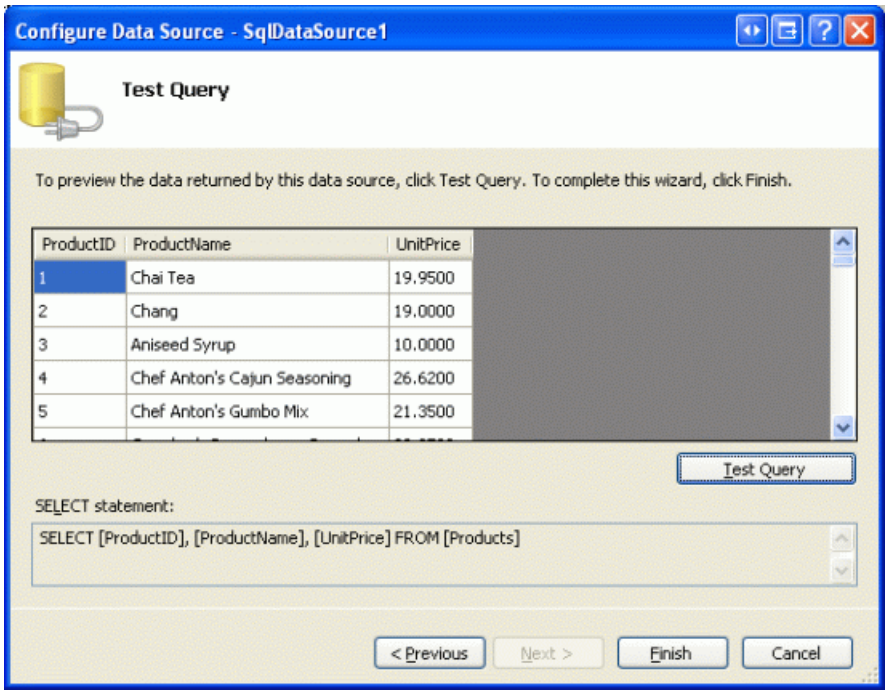


Figure 9: Click the Test Query Button to Review Your `SELECT` Query

To complete the wizard, click Finish.

Like with the ObjectDataSource, the SqlDataSource s wizard merely assigns values to the control s properties, namely the `ConnectionString` and `SelectCommand` properties. After completing the wizard, your SqlDataSource control s declarative markup should look similar to the following:

```
<asp:SqlDataSource ID="ProductsDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand="SELECT [ProductID], [ProductName], [UnitPrice] FROM [Products]">
</asp:SqlDataSource>
```

The `ConnectionString` property provides information on how to connect to the database. This property can be assigned a complete, hard-coded connection string value or can point to a connection string in `Web.config`. To reference a connection string value in Web.config, use the syntax `<%$ expressionPrefix:expressionValue %>`. Typically, *expressionPrefix* is ConnectionStrings and *expressionValue* is the name of the connect string in the `Web.config` `<connectionStrings>` section. However, the syntax can be used to reference `<appSettings>` elements or content from resource files. See ASP.NET Expressions Overview for more on this syntax.

The `SelectCommand` property specifies the ad-hoc SQL statement or stored procedure to execute to return the data.

Step 3: Adding a Data Web Control and Binding it to the SqlDataSource

Once the SqlDataSource has been configured, it can be bound to a data Web control, such as a GridView or DetailsView. For this tutorial, let s display the data in a GridView. From the Toolbox, drag a GridView onto the page and bind it to the `ProductsDataSource` SqlDataSource by choosing the data source from the drop-down list in the GridView s smart tag.

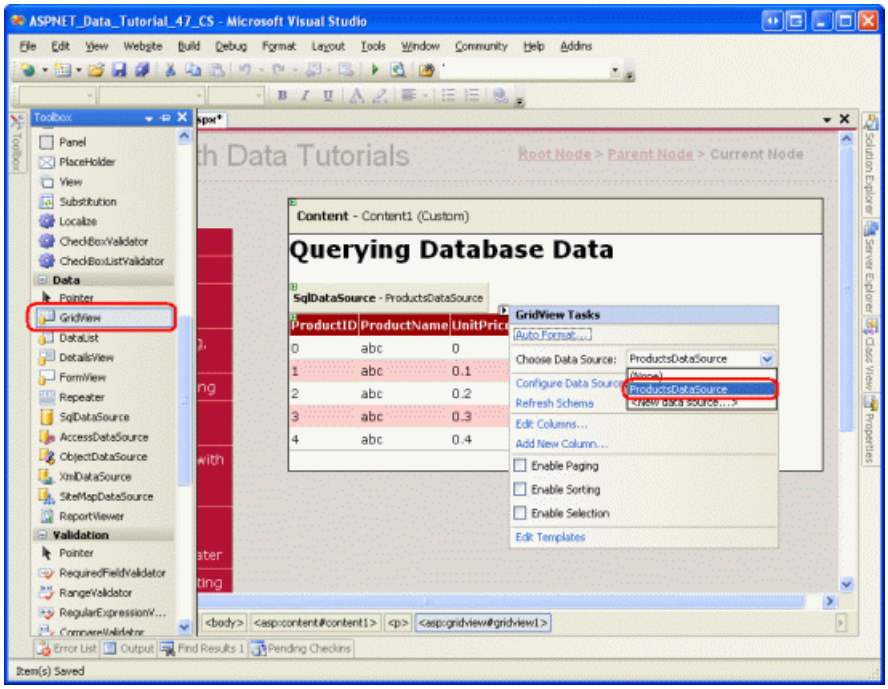


Figure 10: Add a GridView and Bind it to the SqlDataSource Control (Click to view full-size image)

Once you ve selected the SqlDataSource control from the drop-down list in the GridView s smart tag, Visual Studio will automatically add a BoundField or CheckBoxField to the GridView for each of the columns returned by the data source control. Since the SqlDataSource returns three database columns `ProductID`, `ProductName`, and `UnitPrice` there are three fields in the GridView.

Take a moment to configure the GridView s three BoundFields. Change the `ProductName` field s `HeaderText` property to Product Name and the `UnitPrice` field s to Price. Also format the `UnitPrice` field as a currency. After making these modifications, your GridView s declarative markup should look similar to the following:

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="ProductID"
            InsertVisible="False" ReadOnly="True" SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName" HeaderText="Product Name"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
            SortExpression="UnitPrice" DataFormatString="{0:c}"
            HtmlEncode="False" />
    </Columns>
</asp:GridView>
```

Visit this page through a browser. As Figure 11 shows, the GridView lists each product s `ProductID`, `ProductName`, and `UnitPrice` values.

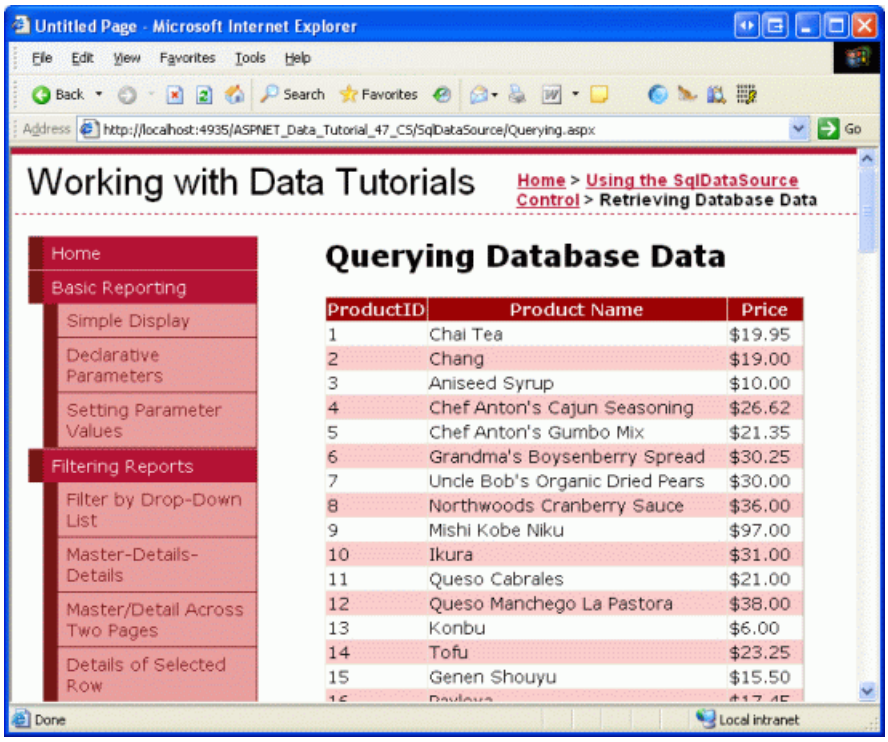


Figure 11: The GridView Displays Each Product s `ProductID`, `ProductName`, and `UnitPrice` Values (Click to view full-size image)

When the page is visited the GridView invokes its data source control s `Select()` method. When we were using the ObjectDataSource control, this called the `ProductsBLL` class s `GetProducts()` method. With the SqlDataSource, however, the `Select()` method establishes a connection to the specified database and issues the `SelectCommand` (`SELECT [ProductID], [ProductName], [UnitPrice] FROM [Products]`, in this example). The SqlDataSource returns its results which the GridView then enumerates, creating a row in the GridView for each database record returned.

The Built-In Data Web Control Features and the SqlDataSource Control

In general, the features inherent to the data Web controls paging, sorting, editing, deleting, inserting, and so on are specific to the data Web control and are not dependent on the data source control used. That is, the GridView can utilize its built-in paging, sorting, editing, and deleting whether it is bound to an ObjectDataSource or a SqlDataSource. However, certain data Web control features are sensitive to the data source control being used or to the data source control s configuration.

For example, in the Efficiently Paging Through Large Amounts of Data tutorial we discussed how, by default, the

paging logic for the data Web controls naively returns *all* records from the underlying data source and then displays only the appropriate subset of records given the current page index and the number of records to display per page. This model is highly inefficient when paging through sufficiently large result sets. Fortunately, the ObjectDataSource can be configured to support custom paging, which returns only the precise subset of records to display. The SqlDataSource control, however, lacks the properties for implementing custom paging.

Another subtlety with paging and sorting arises with the SqlDataSource. By default, the data returned from a SqlDataSource can be paged or sorted through the GridView. To demonstrate this, check the Enable Paging and Enable Sorting options in the GridView s smart tag in `Querying.aspx` and verify that this works as expected.

Sorting and paging works because the SqlDataSource retrieves the database data into a loosely-typed DataSet. The total number of records returned by the query an essential aspect to implementing paging can be ascertained from the DataSet. Additionally, the DataSet s results can be sorted through a DataView. These capabilities are automatically used by the SqlDataSource when the GridView requests paged or sorted data.

The SqlDataSource can be configured to return a DataReader instead of a DataSet by changing its `DataSourceMode` property from `DataSet` (the default) to `DataReader`. Using a DataReader might be preferred in situations when passing the SqlDataSource s results to existing code that expects a DataReader. Furthermore, since DataReaders are considerably simpler objects than DataSets, they offer better performance. If you make this change, however, the data Web control can neither sort nor page since the SqlDataSource cannot ascertain how many records are returned by the query, nor does the DataReader offer any techniques for sorting the returned data.

Step 4: Using a Custom SQL Statement or Stored Procedure

When configuring the SqlDataSource control, the query used to return data can be specified in one of two approaches as a custom SQL statement or stored procedure, or as columns from an existing table or view. In Step 2 we examined selecting columns from the `Products` table. Let s look at using a custom SQL statement.

Add another GridView control to the `Querying.aspx` page and choose to create a new data source from the drop-down list in the smart tag. Next, indicate that the data will be pulled from a database this will create a new SqlDataSource control. Name the control `ProductsWithCategoryInfoDataSource`.

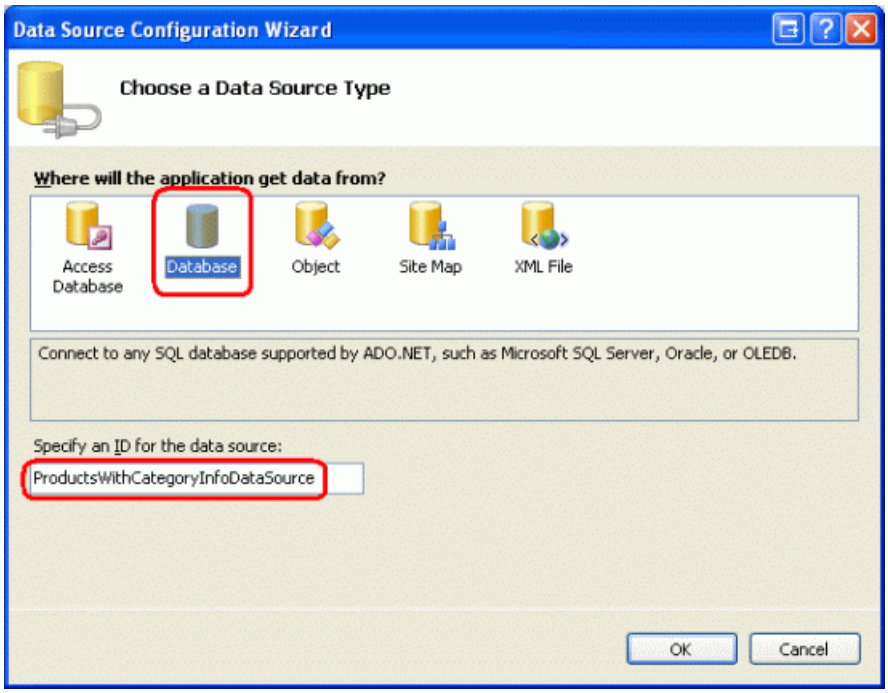


Figure 12: Create a New SqlDataSource Control Named `ProductsWithCategoryInfoDataSource`

The next screen asks us to specify the database. As we did back in Figure 7, select the `NORTHWINDConnectionString` from the drop-down list and click Next. In the Configure the Select Statement screen, choose the Specify a custom SQL statement or stored procedure radio button and click Next. This will bring up the Define Custom Statements

or Stored Procedures screen, which offers tabs labeled SELECT, UPDATE, INSERT, and DELETE. In each tab you can enter a custom SQL statement into the textbox or choose a stored procedure from the drop-down list. In this tutorial we will look at entering a custom SQL statement; the next tutorial includes an example that uses a stored procedure.

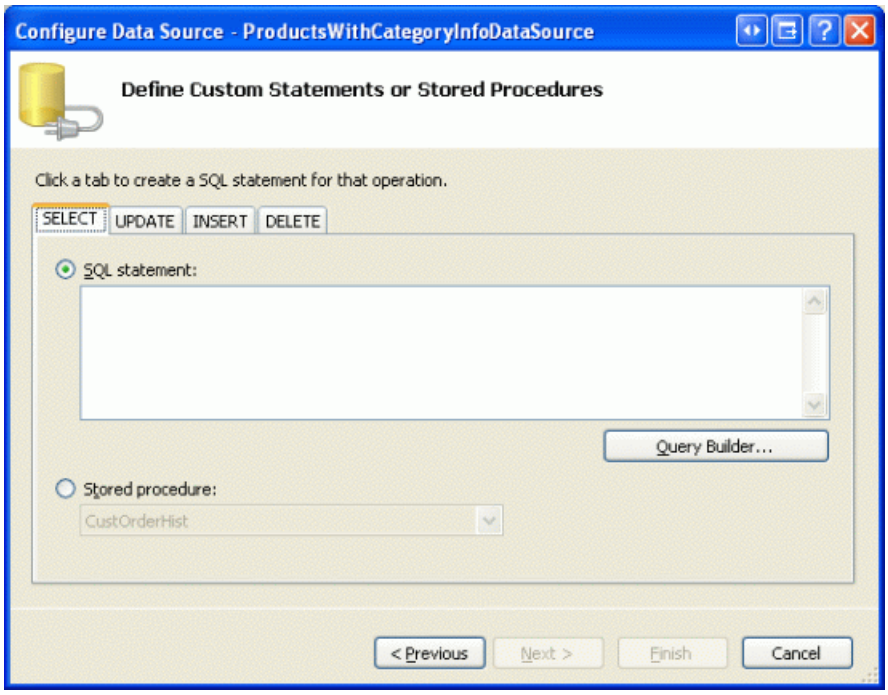


Figure 13: Enter a Custom SQL Statement or Pick a Stored Procedure

The custom SQL statement can be entered by hand into the textbox or can be constructed graphically by clicking the Query Builder button. From either the Query Builder or the textbox, use the following query to return the `ProductID` and `ProductName` fields from the `Products` table using a `JOIN` to retrieve the product s `CategoryName` from the `Categories` table:

```
SELECT Products.ProductID, Products.ProductName, Categories.CategoryName
FROM Categories
    INNER JOIN Products ON
        Categories.CategoryID = Products.CategoryID
```

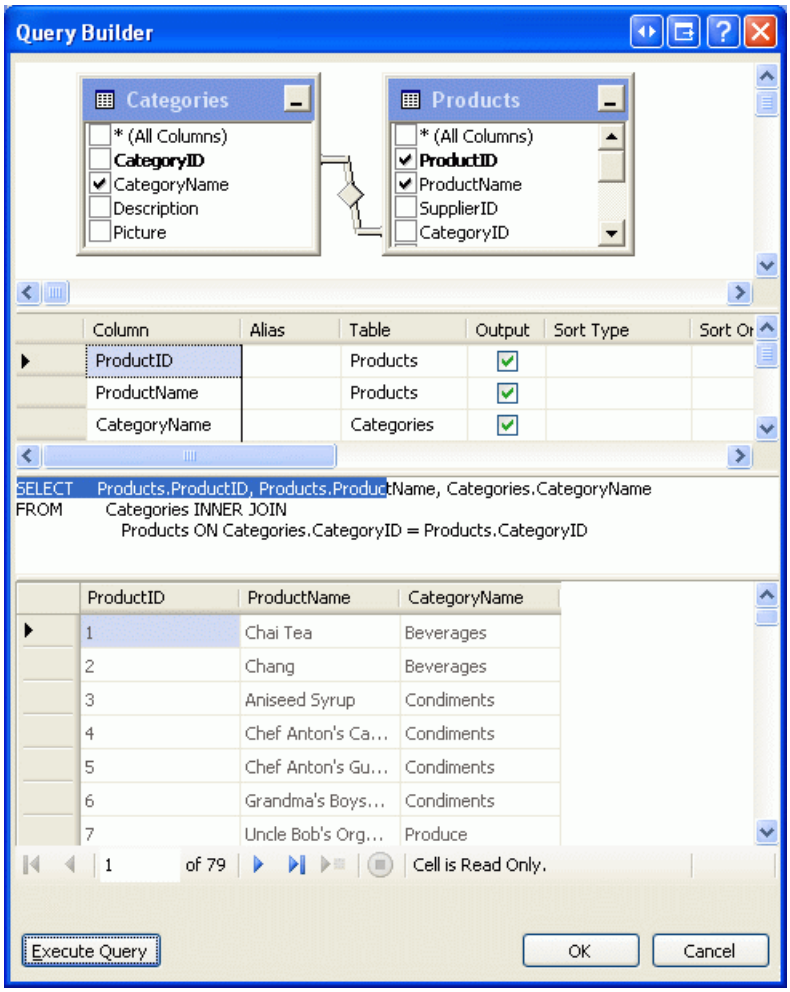


Figure 14: You can Graphically Construct the Query Using the Query Builder

After specifying the query, click Next to proceed to the Test Query screen. Click Finish to complete the SqlDataSource wizard.

After completing the wizard, the GridView will have three BoundFields added to it displaying the `ProductID`, `ProductName`, and `CategoryName` columns returned from the query and resulting in the following declarative markup:

```
<asp:GridView ID="GridView2" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsWithCategoryInfoDataSource"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="ProductID"
            InsertVisible="False" ReadOnly="True" SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName" HeaderText="ProductName"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="CategoryName"
            SortExpression="CategoryName" />
    </Columns>
</asp:GridView>
<asp:SqlDataSource ID="ProductsWithCategoryInfoDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand="
        SELECT Products.ProductID, Products.ProductName, Categories.CategoryName
        FROM Categories
        INNER JOIN Products ON Categories.CategoryID = Products.CategoryID">
</asp:SqlDataSource>
```

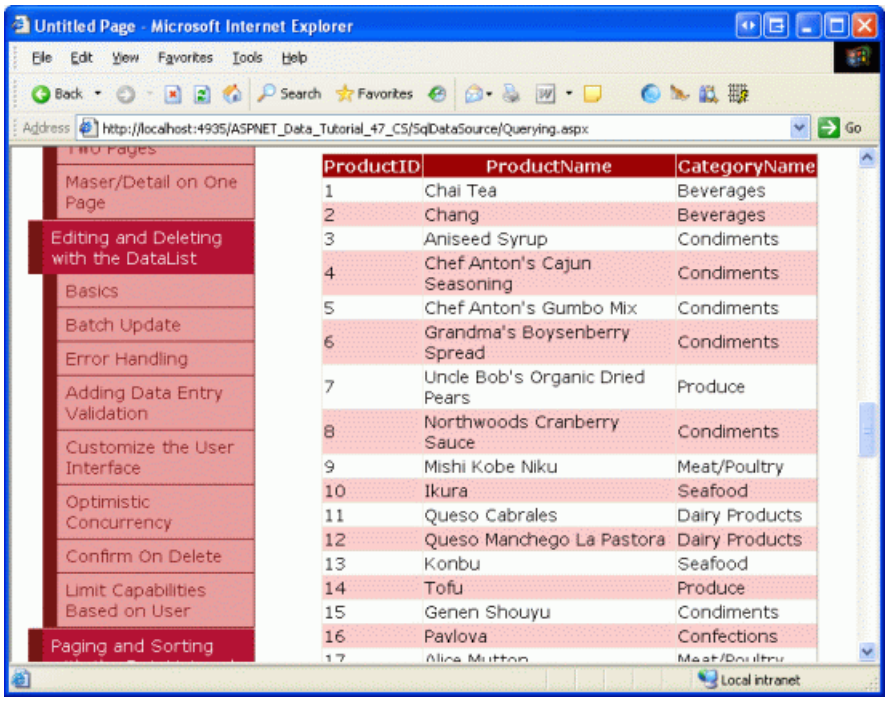


Figure 15: The GridView Shows Each Product s ID, Name, and Associated Category Name (Click to view full-size image)

Summary

In this tutorial we saw how to query and display data using the SqlDataSource control. Like the ObjectDataSource, the SqlDataSource serves as a proxy, providing a declarative approach to accessing data. Its properties specify the database to connect to and the SQL `SELECT` query to execute; they can be specified through the Properties window or by using the Configure DataSource wizard.

The `SELECT` query examples we examined in this tutorial returned all of the records from the specified query. The SqlDataSource control, however, can include a `WHERE` clause with parameters whose values are assigned programmatically or are automatically pulled from a specified source. We'll examine how to create and use parameterized queries in the next tutorial!

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Accessing Relational Database Data
- SqlDataSource Control Overview
- ASP.NET Quickstart Tutorials: The SqlDataSource Control
- The Web.config `<connectionStrings>` Element
- Database Connection String Reference

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Susan Connery, Bernadette Leigh, and David Suru. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Using Parameterized Queries with the SqlDataSource (VB)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial, we continue our look at the SqlDataSource control and learn how to define parameterized queries. The parameters can be specified both declaratively and programmatically, and can be pulled from a number of locations such as the querystring, Session state, other controls, and more.

Introduction

In the previous tutorial we saw how to use the SqlDataSource control to retrieve data directly from a database. Using the Configure Data Source wizard, we could choose the database and then either: pick the columns to return from a table or view; enter a custom SQL statement; or use a stored procedure. Whether selecting columns from a table or view or entering a custom SQL statement, the SqlDataSource control s `SelectCommand` property is assigned the resulting ad-hoc SQL `SELECT` statement and it is this `SELECT` statement that is executed when the SqlDataSource s `Select()` method is invoked (either programmatically or automatically from a data Web control).

The SQL `SELECT` statements used in the previous tutorial s demos lacked `WHERE` clauses. In a `SELECT` statement, the `WHERE` clause can be used to limit the results returned. For example, to display the names of products costing more than $50.00, we could use the following query:

```
SELECT ProductName
FROM Products
WHERE UnitPrice > 50.00
```

Typically, the values used in a `WHERE` clause are determine by some external source, such as a querystring value, a session variable, or user input from a Web control on the page. Ideally, such inputs are specified through the use of *parameters*. With Microsoft SQL Server, parameters are denoted using `@parameterName`, as in:

```
SELECT ProductName
FROM Products
WHERE UnitPrice > @Price
```

The SqlDataSource supports parameterized queries, both for `SELECT` statements and `INSERT`, `UPDATE`, and `DELETE` statements. Moreover, the parameter values can be automatically pulled from a variety of sources the querystring, session state, controls on the page, and so on or can be assigned programmatically. In this tutorial, we'll see how to define parameterized queries as well as how to specify the parameter values both declaratively and programmatically.

NOTE

In the previous tutorial we compared the ObjectDataSource which has been our tool of choice over the first 46 tutorials with the SqlDataSource, noting their conceptual similarities. These similarities also extend to parameters. The ObjectDataSource s parameters mapped to the input parameters for the methods in the Business Logic Layer. With the SqlDataSource, the parameters are defined directly within the SQL query. Both controls have collections of parameters for their `Select()`, `Insert()`, `Update()`, and `Delete()` methods, and both can have these parameter values populated from pre-defined sources (querystring values, session variables, and so on) or assigned programmatically.

Creating a Parameterized Query

The SqlDataSource control s Configure Data Source wizard offers three avenues for defining the command to execute to retrieve database records:

- By picking the columns from an existing table or view,
- By entering a custom SQL statement, or
- By choosing a stored procedure

When picking columns from an existing table or view, the parameters for the `WHERE` clause must be specified through the Add `WHERE` Clause dialog box. When creating a custom SQL statement, however, you can enter the parameters directly into the `WHERE` clause (using `@parameterName` to denote each parameter). A stored procedure consists of one or more SQL statements, and these statements can be parameterized. The parameters used in the SQL statements, however, must be passed in as input parameters to the stored procedure.

Since creating a parameterized query depends on how the SqlDataSource s `SelectCommand` is specified, let s take a look at all three approaches. To get started, open the `ParameterizedQueries.aspx` page in the `SqlDataSource` folder, drag a SqlDataSource control from the Toolbox onto the Designer, and set its `ID` to `Products25BucksAndUnderDataSource`. Next, click the Configure Data Source link from the control s smart tag. Select the database to use (`NORTHWINDConnectionString`) and click Next.

Step 1: Adding a `WHERE` Clause When Picking the Columns from a Table or View

When selecting the data to return from the database with the SqlDataSource control, the Configure Data Source wizard allows us to simply pick the columns to return from an existing table or view (see Figure 1). Doing so automatically builds up a SQL `SELECT` statement, which is what is sent to the database when the SqlDataSource s `Select()` method is invoked. As we did in the previous tutorial, select the Products table from the drop-down list and check the `ProductID`, `ProductName`, and `UnitPrice` columns.

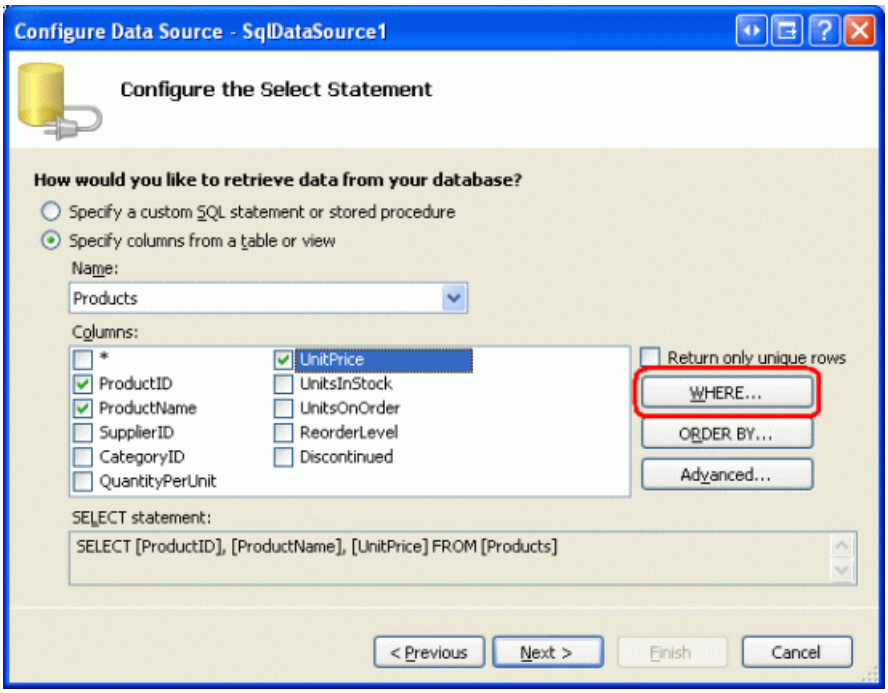


Figure 1: Pick the Columns to Return from a Table or View (Click to view full-size image)

To include a `WHERE` clause in the `SELECT` statement, click the `WHERE` button, which brings up the Add `WHERE` Clause dialog box (see Figure 2). To add a parameter to limit the results returned by the `SELECT` query, first choose the column to filter the data by. Next, choose the operator to use for filtering (=, <, <=, >, and so on). Finally, choose the source of the parameter s value, such as from the querystring or session state. After configuring the parameter, click the Add button to include it in the `SELECT` query.

For this example, let s only return those results where the `UnitPrice` value is less than or equal to $25.00. Therefore, pick `UnitPrice` from the Column drop-down list and <= from the Operator drop-down list. When using a hard-coded parameter value (such as $25.00) or if the parameter value is to be specified programmatically, select None from the Source drop-down list. Next, enter the hard-coded parameter value in the Value textbox 25.00 and complete the process by clicking the Add button.

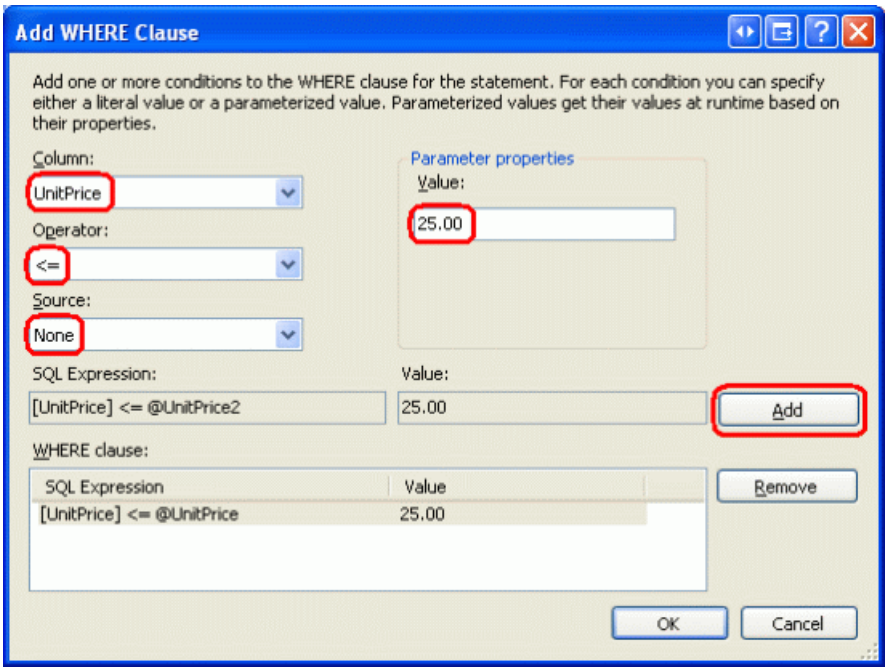


Figure 2: Limit the Results Returned from the Add `WHERE` Clause Dialog Box (Click to view full-size image)

After adding the parameter, click OK to return to the Configure Data Source wizard. The `SELECT` statement at the bottom of the wizard should now include a `WHERE` clause with a parameter named `@UnitPrice`:

```
SELECT [ProductID], [ProductName], [UnitPrice]
FROM [Products]
WHERE ([UnitPrice] <= @UnitPrice)
```

NOTE

If you specify multiple conditions in the `WHERE` clause from the Add `WHERE` Clause dialog box, the wizard joins them with the `AND` operator. If you need to include an `OR` in the `WHERE` clause (such as `WHERE UnitPrice <= @UnitPrice OR Discontinued = 1`) then you have to build the `SELECT` statement through the custom SQL statement screen.

Complete configuring the SqlDataSource (click Next, then Finish) and then inspect the SqlDataSource s declarative markup. The markup now includes a `<SelectParameters>` collection, which spells out the sources for the parameters in the `SelectCommand`.

```
<asp:SqlDataSource ID="Products25BucksAndUnderDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand=
        "SELECT [ProductID], [ProductName], [UnitPrice]
        FROM [Products] WHERE ([UnitPrice] <= @UnitPrice)">
    <SelectParameters>
        <asp:Parameter DefaultValue="25.00" Name="UnitPrice" Type="Decimal" />
    </SelectParameters>
</asp:SqlDataSource>
```

When the SqlDataSource s `Select()` method is invoked, the `UnitPrice` parameter value (25.00) is applied to the `@UnitPrice` parameter in the `SelectCommand` before being sent to the database. The net result is that only those products less than or equal to $25.00 are returned from the `Products` table. To confirm this, add a GridView to the page, bind it to this data source, and then view the page through a browser. You should only see those products listed that are less than or equal to $25.00, as Figure 3 confirms.

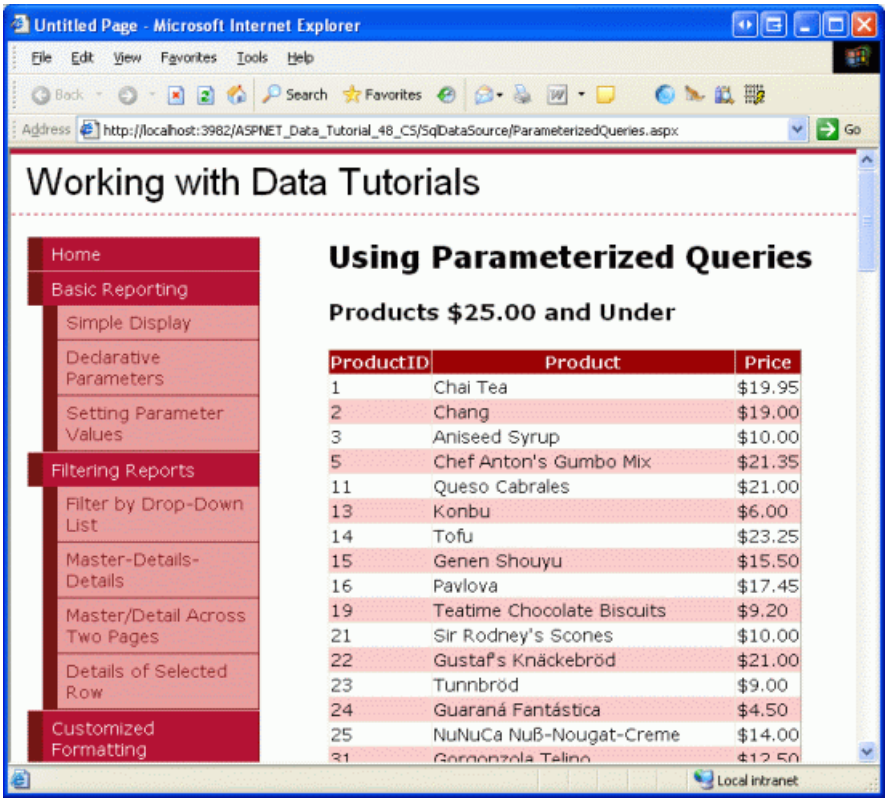


Figure 3: Only Those Products Less Than or Equal to $25.00 are Displayed (Click to view full-size image)

Step 2: Adding Parameters to a Custom SQL Statement

When adding a custom SQL statement you can enter the `WHERE` clause explicitly or specify a value in the Filter cell of the Query Builder. To demonstrate this, let s display just those products in a GridView whose prices are less than a certain threshold. Start by adding a TextBox to the `ParameterizedQueries.aspx` page to collect this threshold value from the user. Set the TextBox s `ID` property to `MaxPrice`. Add a Button Web control and set its `Text` property to Display Matching Products .

Next, drag a GridView onto the page and from its smart tag choose to create a new SqlDataSource named `ProductsFilteredByPriceDataSource`. From the Configure Data Source wizard, proceed to the Specify a custom SQL statement or stored procedure screen (see Figure 4) and enter the following query:

```
SELECT ProductName, UnitPrice
FROM Products
WHERE UnitPrice <= @MaximumPrice OR @MaximumPrice = -1.0
```

After entering the query (either manually or through the Query Builder), click Next.

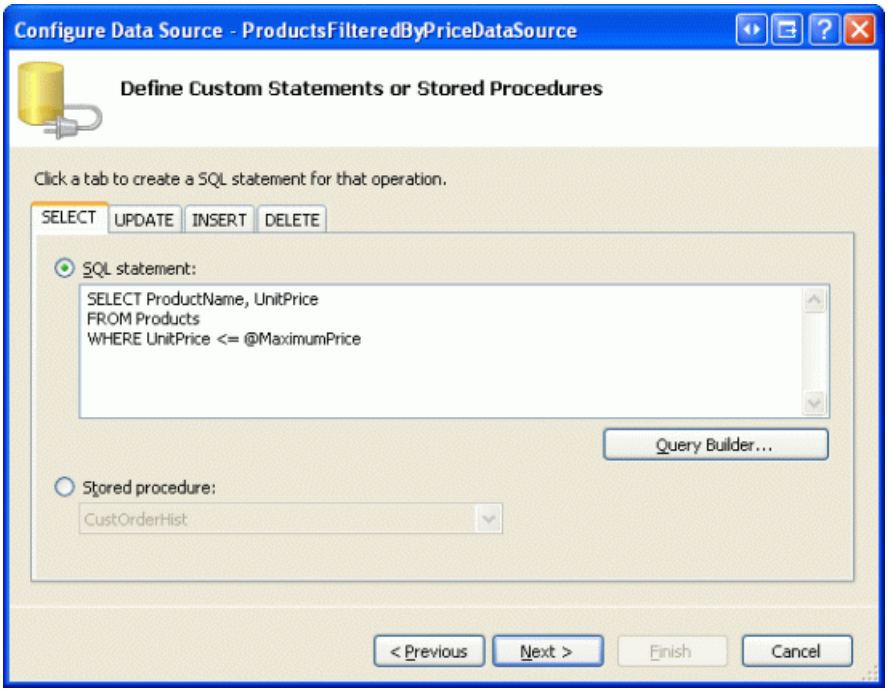


Figure 4: Return Only Those Products Less Than or Equal to a Parameter Value (Click to view full-size image)

Since the query includes parameters, the next screen in the wizard prompts us for the source of the parameters values. Choose Control from the Parameter source drop-down list and `MaxPrice` (the TextBox control s `ID` value) from the ControlID drop-down list. You can also enter an optional default value to use in the case where the user has not entered any text into the `MaxPrice` TextBox. For the time being, do not enter a default value.

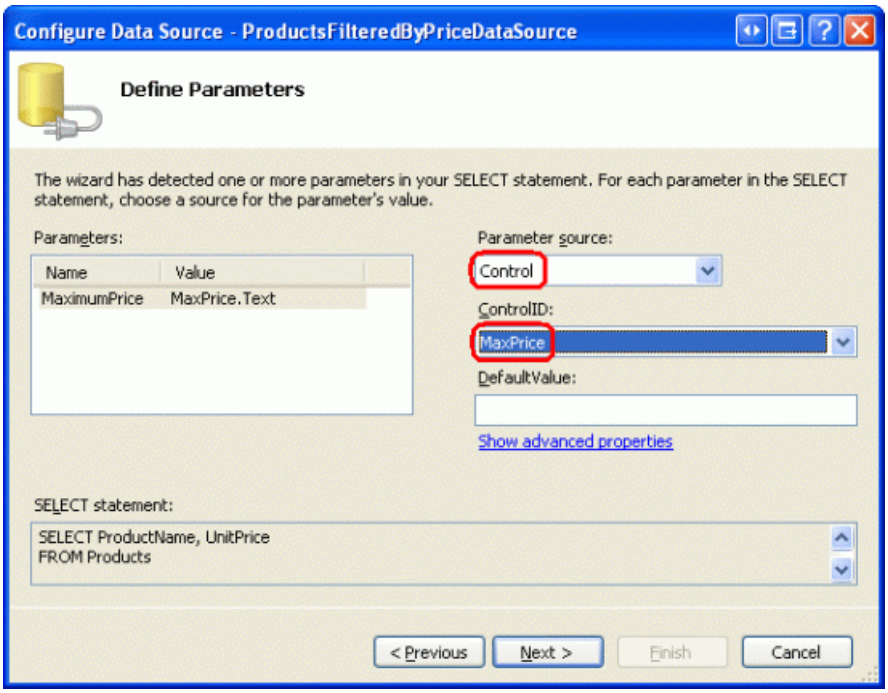


Figure 5: The `MaxPrice` TextBox s `Text` Property is Used as the Parameter Source (Click to view full-size image)

Complete the Configure Data Source wizard by clicking Next, then Finish. The declarative markup for the GridView, TextBox, Button, and SqlDataSource follows:

```
Maximum price:
$<asp:TextBox ID="MaxPrice" runat="server" Columns="5" />

<asp:Button ID="DisplayProductsLessThanButton" runat="server"
    Text="Display Matching Products" />
<asp:GridView ID="GridView2" runat="server" AutoGenerateColumns="False"
    DataSourceID="ProductsFilteredByPriceDataSource" EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
            HtmlEncode="False" DataFormatString="{0:c}"
            SortExpression="UnitPrice" />
    </Columns>
</asp:GridView>
<asp:SqlDataSource ID="ProductsFilteredByPriceDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand=
        "SELECT ProductName, UnitPrice
        FROM Products WHERE UnitPrice <= @MaximumPrice">
    <SelectParameters>
        <asp:ControlParameter ControlID="MaxPrice" Name="MaximumPrice"
            PropertyName="Text" />
    </SelectParameters>
</asp:SqlDataSource>
```

Note that the parameter within the SqlDataSource s `<SelectParameters>` section is a `ControlParameter`, which includes additional properties like `ControlID` and `PropertyName`. When the SqlDataSource s `Select()` method is invoked, the `ControlParameter` grabs the value from the specified Web control property and assigns it to the corresponding parameter in the `SelectCommand`. In this example, the `MaxPrice` s Text property is used as the `@MaxPrice` parameter value.

Take a minute to view this page through a browser. When first visiting the page or whenever the `MaxPrice` TextBox lacks a value no records are displayed in the GridView.

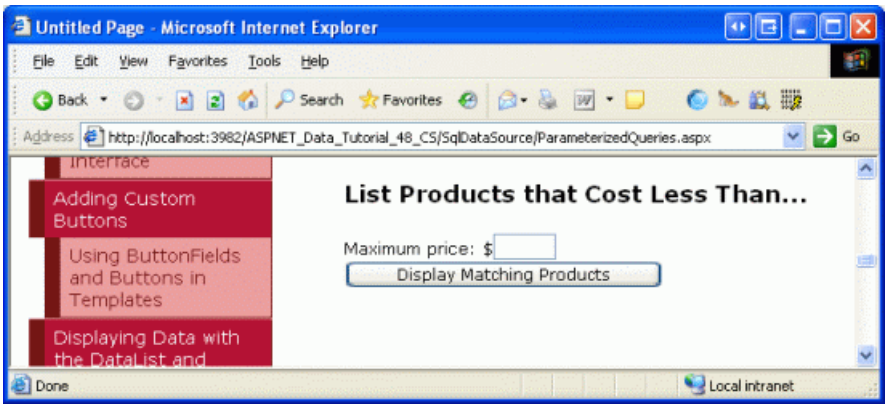


Figure 6: No Records are Displayed When the `MaxPrice` TextBox is Empty (Click to view full-size image)

The reason no products are shown is because, by default, an empty string for a parameter value is converted into a database `NULL` value. Since the comparison of `[UnitPrice] <= NULL` always evaluates as False, no results are returned.

Enter a value into the textbox, like 5.00, and click the Display Matching Products button. On postback, the SqlDataSource informs the GridView that one of its parameter sources has changed. Consequently, the GridView rebinds to the SqlDataSource, displaying those products less than or equal to $5.00.

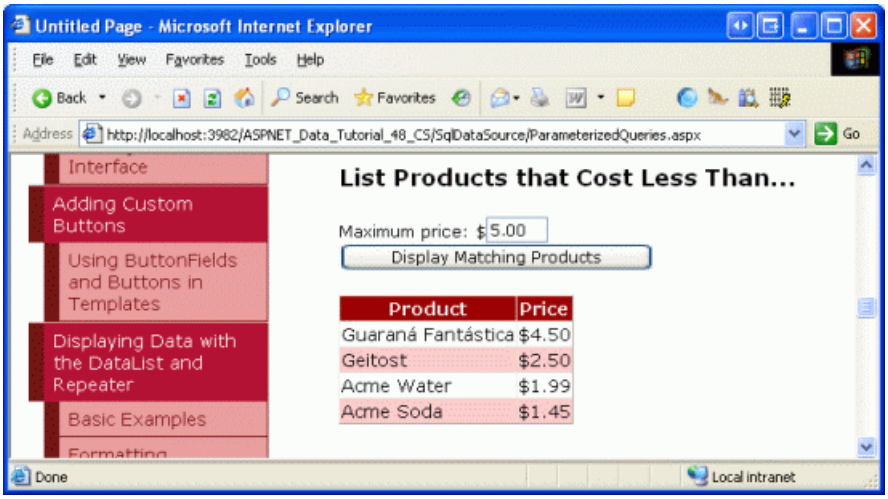


Figure 7: Products Less Than or Equal to $5.00 are Displayed (Click to view full-size image)

Initially Displaying All Products

Rather than displaying no products when the page is first loaded, we may want to display *all* products. One way to list all products whenever the `MaxPrice` TextBox is empty is to set the parameter s default value to some insanely high value, like 1000000, since it s unlikely that Northwind Traders will ever have inventory whose unit price exceeds $1,000,000. However, this approach is shortsighted and might not work in other situations.

In previous tutorials - Declarative Parameters and Master/Detail Filtering With a DropDownList we were faced with a similar problem. Our solution there was to put this logic in the Business Logic Layer. Specifically, the BLL examined the incoming value and, if it was `NULL` or some reserved value, the call was routed to the DAL method that returned all records. If the incoming value was a normal filtering value, a call was made to the DAL method that executed a SQL statement that used a parameterized `WHERE` clause with the supplied value.

Unfortunately, we bypass the architecture when using the SqlDataSource. Instead, we need to customize the SQL statement to intelligently grab all records if the `@MaximumPrice` parameter is `NULL` or some reserved value. For this exercise, let s have it so that if the `@MaximumPrice` parameter is equal to `-1.0`, then *all* of the records are to be returned (`-1.0` works as a reserved value since no product can have a negative `UnitPrice` value). To accomplish this we can use the following SQL statement:

```
SELECT ProductName, UnitPrice
FROM Products
WHERE UnitPrice <= @MaximumPrice OR @MaximumPrice = -1.0
```

This `WHERE` clause returns *all* records if the `@MaximumPrice` parameter equals `-1.0`. If the parameter value is not `-1.0`, only those products whose `UnitPrice` is less than or equal to the `@MaximumPrice` parameter value are returned. By setting the default value of the `@MaximumPrice` parameter to `-1.0`, on the first page load (or whenever the `MaxPrice` TextBox is empty), `@MaximumPrice` will have a value of `-1.0` and all products will be displayed.

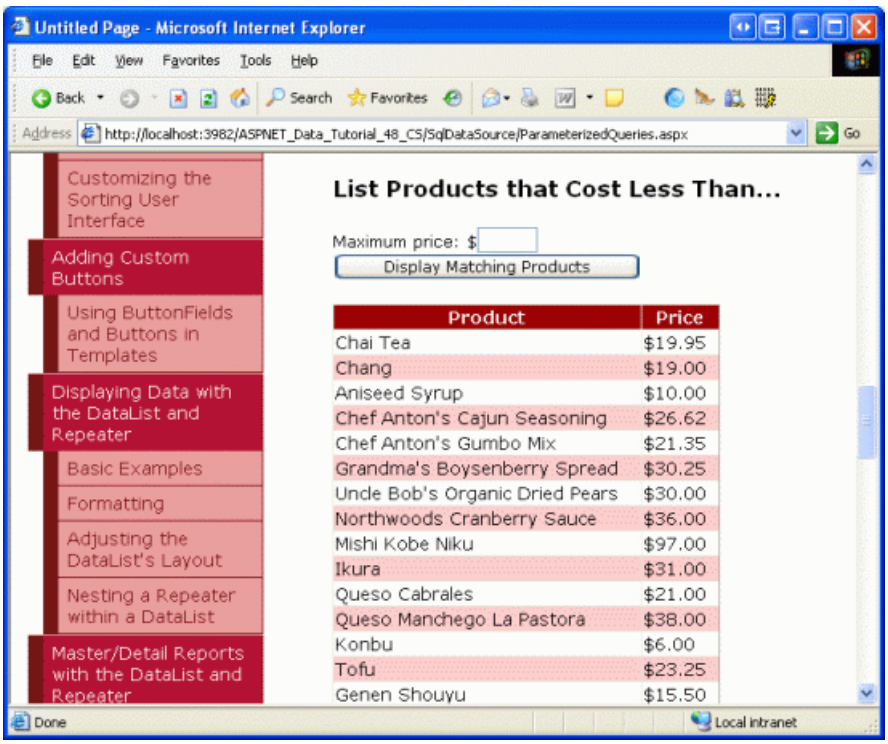


Figure 8: Now All Products are Displayed When the `MaxPrice` TextBox is Empty (Click to view full-size image)

There are a couple of caveats to note with this approach. First, realize that the parameter s data type is inferred by it s usage in the SQL query. If you change the `WHERE` clause from `@MaximumPrice = -1.0` to `@MaximumPrice = -1`, the runtime treats the parameter as an integer. If you then attempt to assign the `MaxPrice` TextBox to a decimal value (like 5.00), an error will occur because it cannot convert 5.00 to an integer. To remedy this, either make sure that you use `@MaximumPrice = -1.0` in the `WHERE` clause or, better yet, set the `ControlParameter` object s `Type` property to Decimal .

Secondly, by adding the `OR @MaximumPrice = -1.0` to the `WHERE` clause, the query engine cannot use an index on `UnitPrice` (assuming one exists), thereby resulting in a table scan. This can impact performance if there are a sufficiently large number of records in the `Products` table. A better approach would be to move this logic to a stored procedure where an `IF` statement would either perform a `SELECT` query from the `Products` table without a `WHERE` clause when all records need to be returned or one whose `WHERE` clause contains just the `UnitPrice` criteria, so that an index can be used.

Step 3: Creating and Using Parameterized Stored Procedures

Stored procedures can include a set of input parameters that can then be used in the SQL statement(s) defined within the stored procedure. When configuring the SqlDataSource to use a stored procedure that accepts input parameters, these parameter values can be specified using the same techniques as with ad-hoc SQL statements.

To illustrate using stored procedures in the SqlDataSource, let s create a new stored procedure in the Northwind database named `GetProductsByCategory`, which accepts a parameter named `@CategoryID` and returns all of the

columns of the products whose `CategoryID` column matches `@CategoryID`. To create a stored procedure, go to the Server Explorer and drill down into the `NORTHWND.MDF` database. (If you don t see the Server Explorer, bring it up by going to the View menu and selecting the Server Explorer option.)

From the `NORTHWND.MDF` database, right-click on the Stored Procedures folder, choose Add New Stored Procedure, and enter the following syntax:

```
CREATE PROCEDURE dbo.GetProductsByCategory
(
      @CategoryID int
)
AS
SELECT *
FROM Products
WHERE CategoryID = @CategoryID
```

Click the Save icon (or Ctrl+S) to save the stored procedure. You can test the stored procedure by right-clicking it from the Stored Procedures folder and choosing Execute. This will prompt you for the stored procedure s parameters (`@CategoryID`, in this instance), after which the results will be displayed in the Output window.

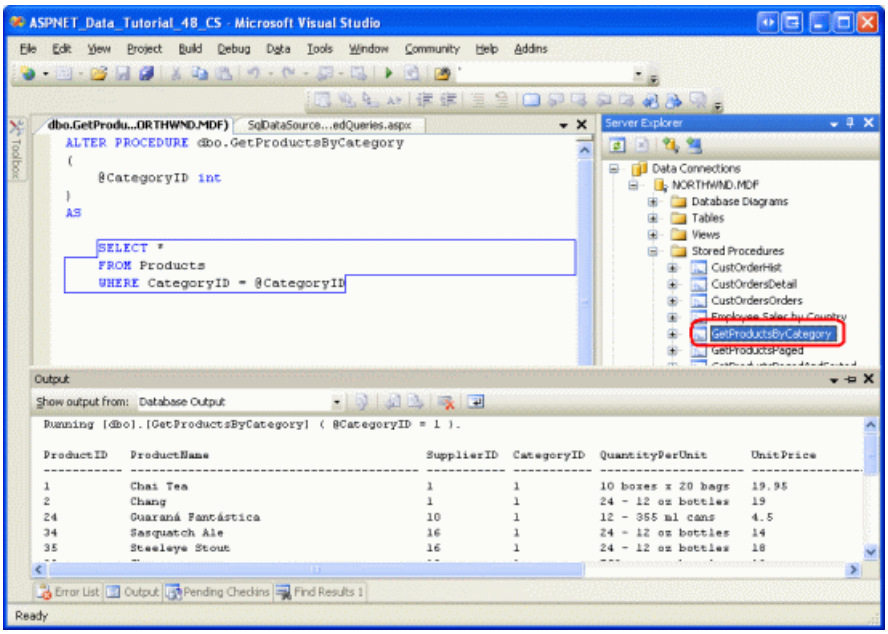


Figure 9: The `GetProductsByCategory` Stored Procedure when Executed with a `@CategoryID` of 1 (Click to view full-size image)

Let s use this stored procedure to display all products in the Beverages category in a GridView. Add a new GridView to the page and bind it to a new SqlDataSource named `BeverageProductsDataSource`. Continue to the Specify a custom SQL statement or stored procedure screen, select the Stored procedure radio button, and pick the `GetProductsByCategory` stored procedure from the drop-down list.

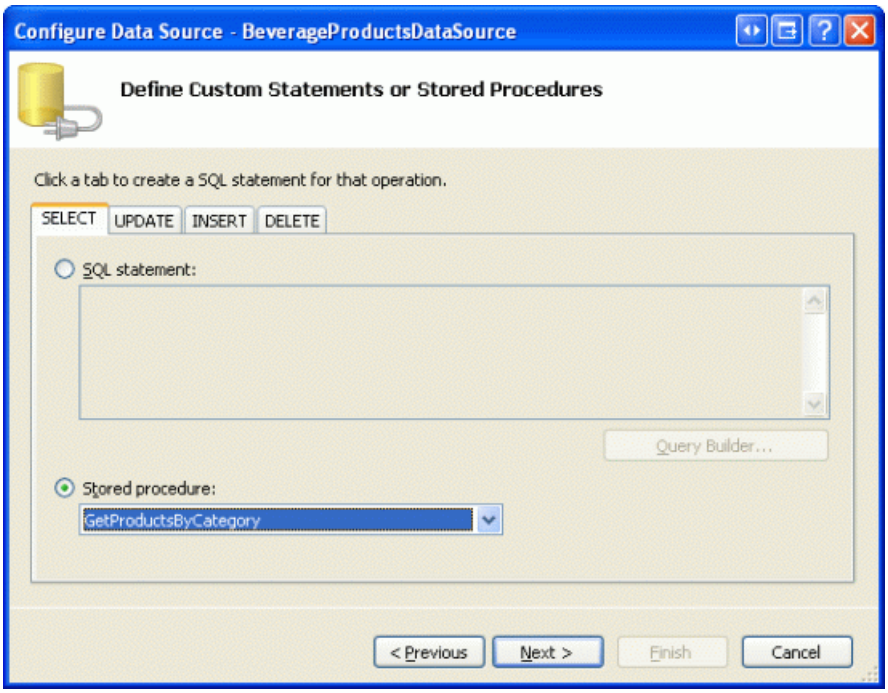


Figure 10: Select the `GetProductsByCategory` Stored Procedure from the Drop-Down List (Click to view full-size image)

Since the stored procedure accepts an input parameter (`@CategoryID`), clicking Next prompts us to specify the source for this parameter s value. The Beverages `CategoryID` is 1, so leave the Parameter source drop-down list at None and enter 1 into the DefaultValue textbox.

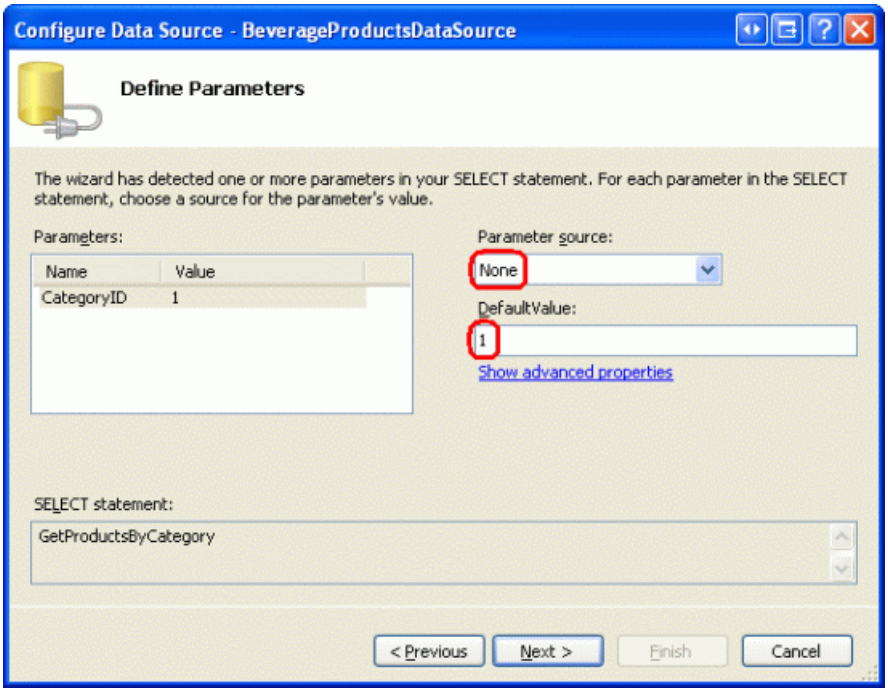
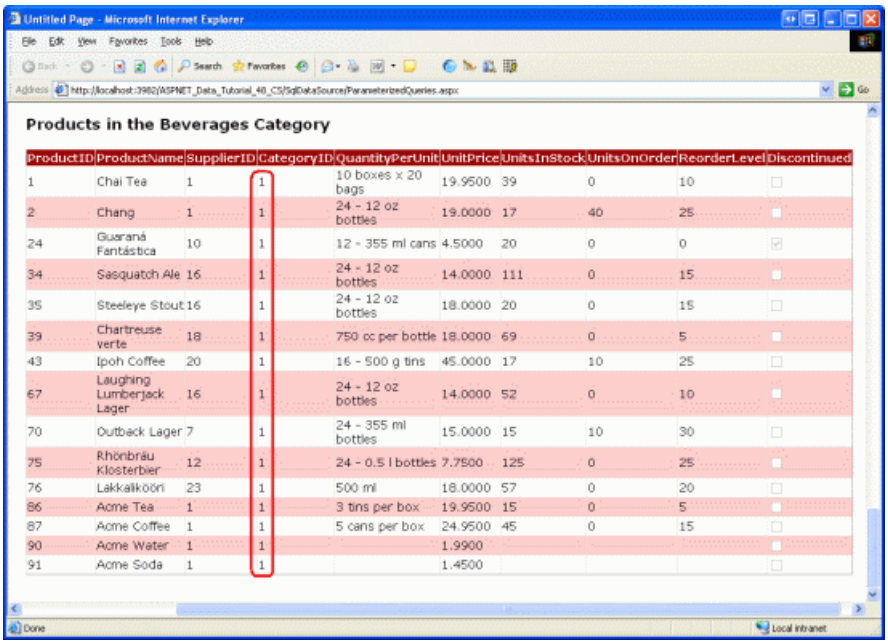


Figure 11: Use a Hard-Coded Value of 1 to Return the Products in the Beverages Category (Click to view full-size image)

As the following declarative markup shows, when using a stored procedure, the SqlDataSource s `SelectCommand` property is set to the name of the stored procedure and the `SelectCommandType` property is set to `StoredProcedure`, indicating that the `SelectCommand` is the name of a stored procedure rather than an ad-hoc SQL statement.

```
<asp:SqlDataSource ID="BeverageProductsDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand="GetProductsByCategory" SelectCommandType="StoredProcedure">
    <SelectParameters>
        <asp:Parameter DefaultValue="1" Name="CategoryID" Type="Int32" />
    </SelectParameters>
</asp:SqlDataSource>
```

Test out the page in a browser. Only those products that belong to the Beverages category are displayed, although *all* of the product fields are displayed since the `GetProductsByCategory` stored procedure returns all of the columns from the `Products` table. We could, of course, limit or customize the fields displayed in the GridView from the GridView s Edit Columns dialog box.



ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
1	Chai Tea	1	1	10 boxes x 20 bags	19.9500	39	0	10	
2	Chang	1	1	24 - 12 oz bottles	19.0000	17	40	25	
24	Guaraná Fantástica	10	1	12 - 355 ml cans	4.5000	20	0	0	
34	Sasquatch Ale	16	1	24 - 12 oz bottles	14.0000	111	0	15	
35	Steeleye Stout	16	1	24 - 12 oz bottles	18.0000	20	0	15	
39	Chartreuse verte	18	1	750 cc per bottle	18.0000	69	0	5	
43	Ipoh Coffee	20	1	16 - 500 g tins	45.0000	17	10	25	
67	Laughing Lumberjack Lager	16	1	24 - 12 oz bottles	14.0000	52	0	10	
70	Outback Lager	7	1	24 - 355 ml bottles	15.0000	15	10	30	
75	Rhönbräu Klosterbier	12	1	24 - 0.5 l bottles	7.7500	125	0	25	
76	Lakkalikööri	23	1	500 ml	18.0000	57	0	20	
86	Acme Tea	1	1	3 tins per box	19.9500	15	0	5	
87	Acme Coffee	1	1	5 cans per box	24.9500	45	0	15	
90	Acme Water	1	1		1.9900				
91	Acme Soda	1	1		1.4500				

Figure 12: All of the Beverages are Displayed (Click to view full-size image)

Step 4: Programmatically Invoking a SqlDataSource s `Select()` Statement

The examples we ve seen in the previous tutorial and this tutorial so far have bound SqlDataSource controls directly to a GridView. The SqlDataSource control s data, however, can be programmatically accessed and enumerated in code. This can be particularly useful when you need to query data to inspect it, but don t need to display it. Rather than having to write all of the boilerplate ADO.NET code to connect to the database, specify the command, and retrieve the results, you can let the SqlDataSource handle this monotonous code.

To illustrate working with the SqlDataSource s data programmatically, imagine that your boss has approached you with a request to create a web page that displays the name of a randomly selected category and its associated products. That is, when a user visits this page, we want to randomly choose a category from the `Categories` table, display the category name, and then list the products belonging to that category.

To accomplish this we need two SqlDataSource controls one to grab a random category from the `Categories` table and another to get the category s products. We'll build the SqlDataSource that retrieves a random category record in this step; Step 5 looks at crafting the SqlDataSource that retrieves the category s products.

Start by adding a SqlDataSource to `ParameterizedQueries.aspx` and set its `ID` to `RandomCategoryDataSource`. Configure it so that it uses the following SQL query:

```
SELECT TOP 1 CategoryID, CategoryName
FROM Categories
ORDER BY NEWID()
```

`ORDER BY NEWID()` returns the records sorted in random order (see Using `NEWID()` to Randomly Sort Records). `SELECT TOP 1` returns the first record from the result set. Put together, this query returns the `CategoryID` and `CategoryName` column values from a single, randomly selected category.

To display the category s `CategoryName` value, add a Label Web control to the page, set its `ID` property to `CategoryNameLabel`, and clear out its `Text` property. To programmatically retrieve the data from a SqlDataSource control, we need to invoke its `Select()` method. The `Select()` method expects a single input parameter of type `DataSourceSelectArguments`, which specifies how the data should be messaged before being returned. This can include instructions on sorting and filtering the data, and is used by the data Web controls when sorting or paging through the data from a SqlDataSource control. For our example, though, we don t need the data to be modified before being returned, and therefore will pass in the `DataSourceSelectArguments.Empty` object.

The `Select()` method returns an object that implements `IEnumerable`. The precise type returned depends on the value of the SqlDataSource control s `DataSourceMode` property. As discussed in the previous tutorial, this property can be set to a value of either `DataSet` or `DataReader`. If set to `DataSet`, the `Select()` method returns a DataView object; if set to `DataReader`, it returns an object that implements `IDataReader`. Since the `RandomCategoryDataSource` SqlDataSource has its `DataSourceMode` property set to `DataSet` (the default), we will be working with a DataView object.

The following code illustrates how to retrieve the records from the `RandomCategoryDataSource` SqlDataSource as a DataView as well as how to read the `CategoryName` column value from the first DataView row:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles Me.Load
    ' Get the data from the SqlDataSource as a DataView
    Dim randomCategoryView As DataView = CType _
        (RandomCategoryDataSource.Select(DataSourceSelectArguments.Empty), DataView)
    If randomCategoryView.Count > 0 Then
        ' Assign the CategoryName value to the Label
        CategoryNameLabel.Text = String.Format( _
            "Here are Products in the {0} Category...", _
            randomCategoryView(0)("CategoryName").ToString())
    End If
End Sub
```

`randomCategoryView(0)` returns the first `DataRowView` in the DataView. `randomCategoryView(0)("CategoryName")` returns the value of the `CategoryName` column in this first row. Note that the DataView is loosely-typed. To reference a particular column value we need to pass in the name of the column as a string (CategoryName, in this case). Figure 13 shows the message displayed in the `CategoryNameLabel` when viewing the page. Of course, the actual category name displayed is randomly selected by the `RandomCategoryDataSource` SqlDataSource on each visit to the page (including postbacks).

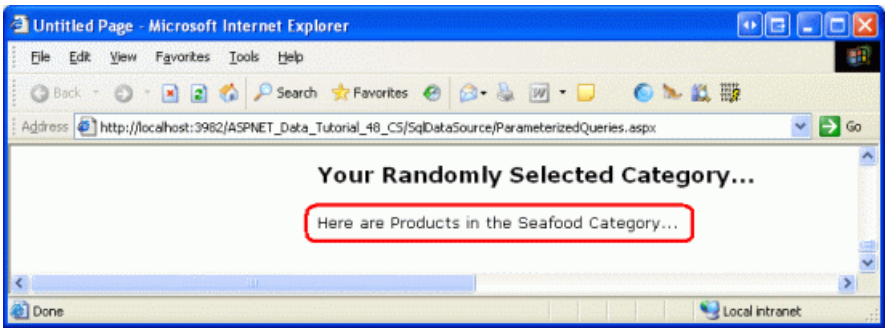


Figure 13: The Randomly Selected Category s Name is Displayed (Click to view full-size image)

> **NOTE**
>
> If the SqlDataSource control s `DataSourceMode` property had been set to `DataReader`, the return value from the `Select()` method would have needed to be cast to `IDataReader`. To read the `CategoryName` column value from the first row, we d use code like:

```
If randomCategoryReader.Read() Then
   Dim categoryName as String = randomCategoryReader("CategoryName').ToString()
   ...
End If
```

With the SqlDataSource randomly selecting a category, we re ready to add the GridView that lists the category s products.

> **NOTE**
>
> Rather than using a Label Web control to display the category s name, we could have added a FormView or DetailsView to the page, binding it to the SqlDataSource. Using the Label, however, allowed us to explore how to programmatically invoke the SqlDataSource s `Select()` statement and work with its resulting data in code.

Step 5: Assigning Parameter Values Programmatically

All of the examples we ve seen so far in this tutorial have used either a hard-coded parameter value or one taken from one of the pre-defined parameter sources (a querystring value, a Web control on the page, and so on). However, the SqlDataSource control s parameters can also be set programmatically. To complete our current example, we need a SqlDataSource that returns all of the products belonging to a specified category. This SqlDataSource will have a `CategoryID` parameter whose value needs to be set based on the `CategoryID` column value returned by the `RandomCategoryDataSource` SqlDataSource in the `Page_Load` event handler.

Start by adding a GridView to the page and bind it to a new SqlDataSource named `ProductsByCategoryDataSource`. Much like we did in Step 3, configure the SqlDataSource so that it invokes the `GetProductsByCategory` stored procedure. Leave the Parameter source drop-down list set to None, but do not enter a default value, as we will set this default value programmatically.

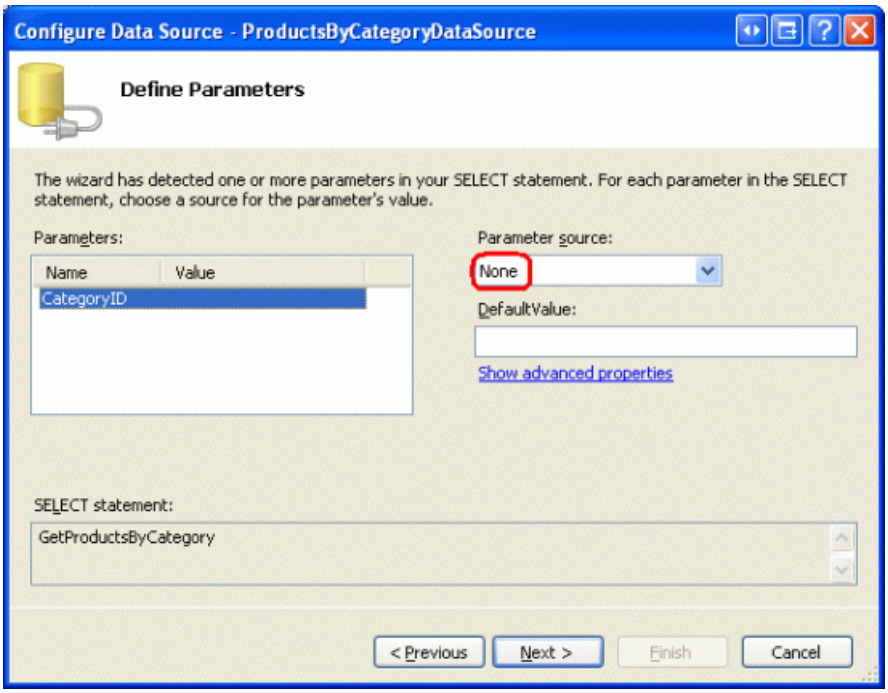


Figure 14: Do Not Specify a Parameter Source or Default Value (Click to view full-size image)

After completing the SqlDataSource wizard, the resulting declarative markup should look similar to the following:

```
<asp:SqlDataSource ID="ProductsByCategoryDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand="GetProductsByCategory" SelectCommandType="StoredProcedure">
    <SelectParameters>
        <asp:Parameter Name="CategoryID" Type="Int32" />
    </SelectParameters>
</asp:SqlDataSource>
```

We can assign the `DefaultValue` of the `CategoryID` parameter programmatically in the `Page_Load` event handler:

```
' Assign the ProductsByCategoryDataSource's
' CategoryID parameter's DefaultValue property
ProductsByCategoryDataSource.SelectParameters("CategoryID").DefaultValue = _
    randomCategoryView(0)("CategoryID").ToString()
```

With this addition, the page includes a GridView that shows the products associated with the randomly selected category.

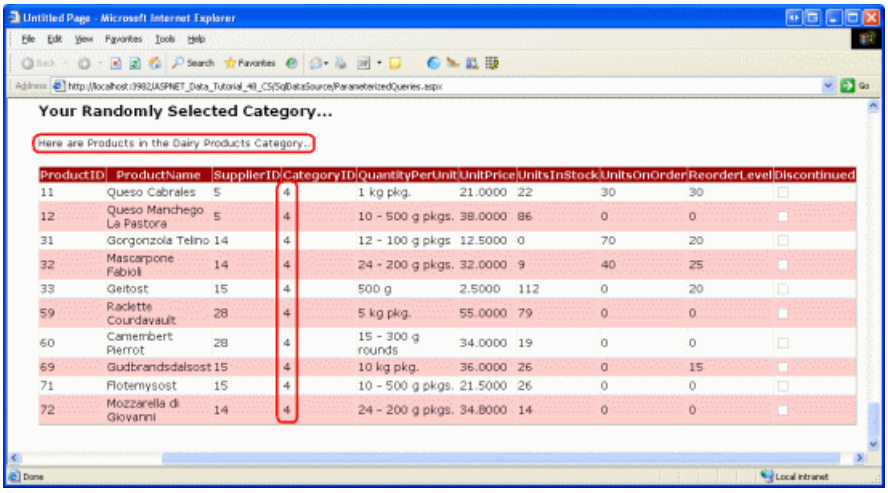


Figure 15: Do Not Specify a Parameter Source or Default Value (Click to view full-size image)

Summary

The SqlDataSource enables page developers to define parameterized queries whose parameter values can be hard-coded, pulled from pre-defined parameter sources, or assigned programmatically. In this tutorial we saw how to craft a parameterized query from the Configure Data Source wizard for both ad-hoc SQL queries and stored procedures. We also looked at using hard-coded parameter sources, a Web control as a parameter source, and programmatically specifying the parameter value.

Like with the ObjectDataSource, the SqlDataSource also provides capabilities to modify its underlying data. In the next tutorial we'll look at how to define `INSERT`, `UPDATE`, and `DELETE` statements with the SqlDataSource. Once these statements have been added, we can utilize the built-in inserting, editing, and deleting features inherent to the GridView, DetailsView, and FormView controls.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Scott Clyde, Randell Schmidt, and Ken Pespisa. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Inserting, Updating, and Deleting Data with the SqlDataSource (VB)

2/20/2020 • 16 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In previous tutorials we learned how the ObjectDataSource control allowed for inserting, updating, and deleting of data. The SqlDataSource control supports the same operations, but the approach is different, and this tutorial shows how to configure the SqlDataSource to insert, update, and delete data.

Introduction

As discussed in An Overview of Inserting, Updating, and Deleting, the GridView control provides built-in updating and deleting capabilities, while the DetailsView and FormView controls include inserting support along with editing and deleting functionality. These data modification capabilities can be plugged directly into a data source control without a line of code needing to be written. An Overview of Inserting, Updating, and Deleting examined using the ObjectDataSource to facilitate inserting, updating, and deleting with the GridView, DetailsView, and FormView controls. Alternatively, the SqlDataSource can be used in place of the ObjectDataSource.

Recall that to support inserting, updating, and deleting, with the ObjectDataSource we needed to specify the object layer methods to invoke to perform the insert, update, or delete action. With the SqlDataSource, we need to provide `INSERT`, `UPDATE`, and `DELETE` SQL statements (or stored procedures) to execute. As we'll see in this tutorial, these statements can be created manually or can be automatically generated by the SqlDataSource s Configure Data Source wizard.

> **NOTE**
>
> Since we ve already discussed the inserting, editing, and deleting capabilities of the GridView, DetailsView, and FormView controls, this tutorial will focus on configuring the SqlDataSource control to support these operations. If you need to brush up on implementing these features within the GridView, DetailsView, and FormView, return to the Editing, Inserting, and Deleting Data tutorials, starting with An Overview of Inserting, Updating, and Deleting.

Step 1: Specifying `INSERT`, `UPDATE`, and `DELETE` Statements

As we ve seen in the past two tutorials, to retrieve data from a SqlDataSource control we need to set two properties:

1. `ConnectionString`, which specifies what database to send the query to, and
2. `SelectCommand`, which specifies the ad-hoc SQL statement or stored procedure name to execute to return the results.

For `SelectCommand` values with parameters, the parameter values are specified via the SqlDataSource s `SelectParameters` collection and can include hard-coded values, common parameter source values (querystring fields, session variables, Web control values, and so on), or can be programmatically assigned. When the SqlDataSource control s `Select()` method is invoked either programmatically or automatically from a data Web control a connection to the database is established, the parameter values are assigned to the query, and the command is shuttled off to the database. The results are then returned as either a DataSet or DataReader,

depending on the value of the control s `DataSourceMode` property.

Along with selecting data, the SqlDataSource control can be used to insert, update, and delete data by supplying `INSERT`, `UPDATE`, and `DELETE` SQL statements in much the same way. Simply assign the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties the `INSERT`, `UPDATE`, and `DELETE` SQL statements to execute. If the statements have parameters (as they most always will), include them in the `InsertParameters`, `UpdateParameters`, and `DeleteParameters` collections.

Once an `InsertCommand`, `UpdateCommand`, or `DeleteCommand` value has been specified, the Enable Inserting, Enable Editing, or Enable Deleting option in the corresponding data Web control s smart tag will become available. To illustrate this, let s take an example from the `Querying.aspx` page we created in the Querying Data with the SqlDataSource Control tutorial and augment it to include delete capabilities.

Start by opening the `InsertUpdateDelete.aspx` and `Querying.aspx` pages from the `SqlDataSource` folder. From the Designer on the `Querying.aspx` page, select the SqlDataSource and GridView from the first example (the `ProductsDataSource` and `GridView1` controls). After selecting the two controls, go to the Edit menu and choose Copy (or just hit Ctrl+C). Next, go to the Designer of `InsertUpdateDelete.aspx` and paste in the controls. After you have moved the two controls over to `InsertUpdateDelete.aspx`, test out the page in a browser. You should see the values of the `ProductID`, `ProductName`, and `UnitPrice` columns for all of the records in the `Products` database table.

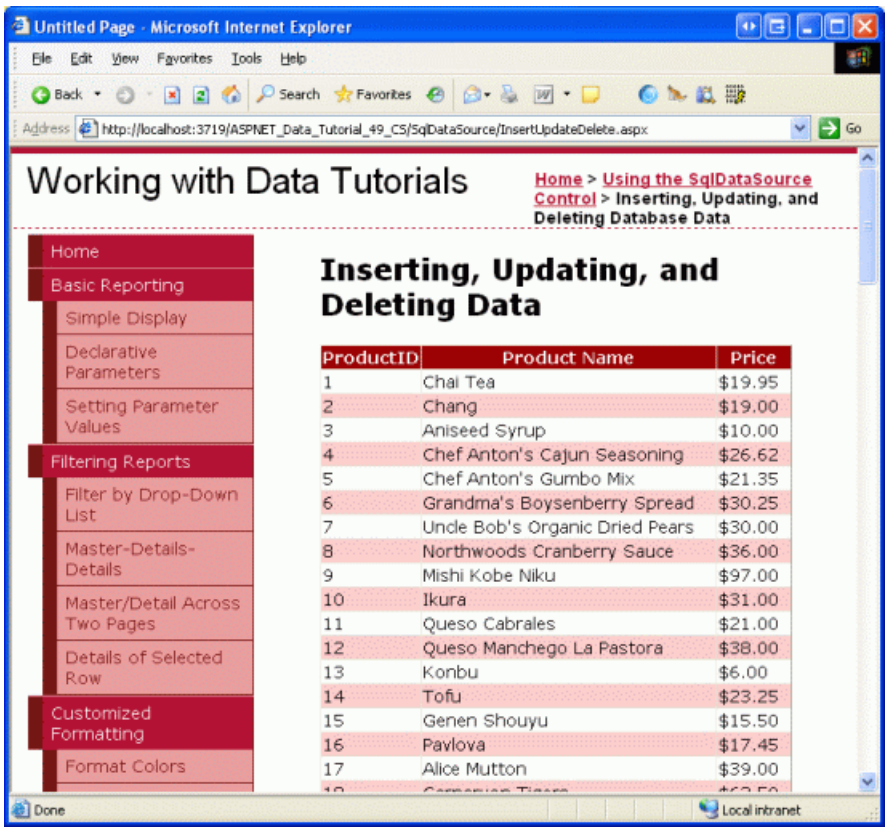


Figure 1: All of the Products are Listed, Ordered by `ProductID` (Click to view full-size image)

Adding the SqlDataSource s `DeleteCommand` and `DeleteParameters` Properties

At this point we have a SqlDataSource that simply returns all of the records from the `Products` table and a GridView that renders this data. Our goal is to extend this example to allow for the user to delete products via the GridView. To accomplish this we need to specify values for the SqlDataSource control s `DeleteCommand` and `DeleteParameters` properties and then configure the GridView to support deleting.

The `DeleteCommand` and `DeleteParameters` properties can be specified in a number of ways:

- Through the declarative syntax
- From the Properties window in the Designer
- From the Specify a custom SQL statement or stored procedure screen in the Configure Data Source wizard
- Via the Advanced button in the Specify columns from a table of view screen in the Configure Data Source wizard, which will actually automatically generate the `DELETE` SQL statement and parameter collection used in the `DeleteCommand` and `DeleteParameters` properties

We'll examine how to automatically have the `DELETE` statement created in Step 2. For now, let s use the Properties window in the Designer, although the Configure Data Source wizard or declarative syntax option would work just as well.

From the Designer in `InsertUpdateDelete.aspx`, click on the `ProductsDataSource` SqlDataSource and then bring up the Properties window (from the View menu, choose Properties window, or simply hit F4). Select the DeleteQuery property, which will bring up a set of ellipses.

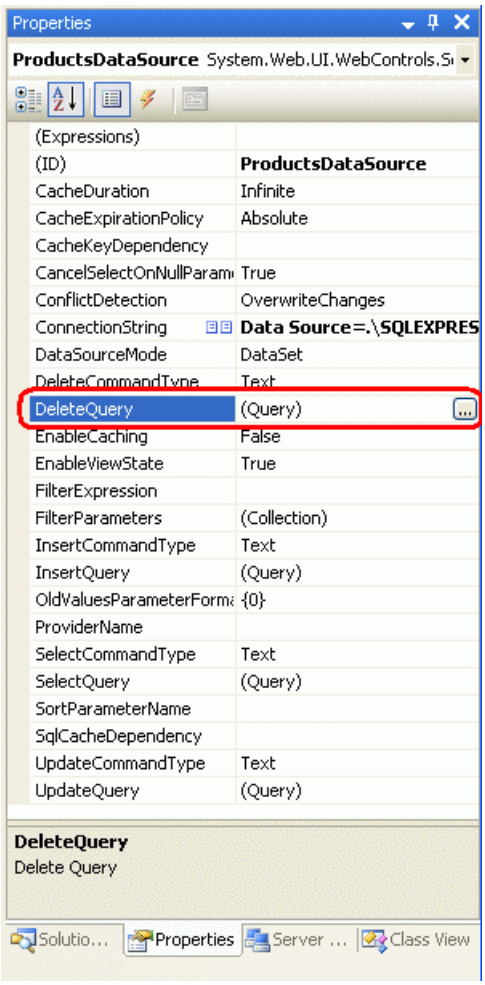


Figure 2: Select the DeleteQuery Property from the Properties Window

> **NOTE**
>
> The SqlDataSource doesn t have a DeleteQuery property. Rather, DeleteQuery is a combination of the `DeleteCommand` and `DeleteParameters` properties and is only listed in the Properties window when viewing the window through the Designer. If you are looking at the Properties window in the Source view, you'll find the `DeleteCommand` property instead.

Click the ellipses in the DeleteQuery property to bring up the Command and Parameter Editor dialog box (see Figure 3). From this dialog box you can specify the `DELETE` SQL statement and specify the parameters. Enter the following query into the `DELETE` command textbox (either manually or using the Query Builder, if you prefer):

```
DELETE FROM Products
WHERE ProductID = @ProductID
```

Next, click the Refresh Parameters button to add the `@ProductID` parameter to the list of parameters below.

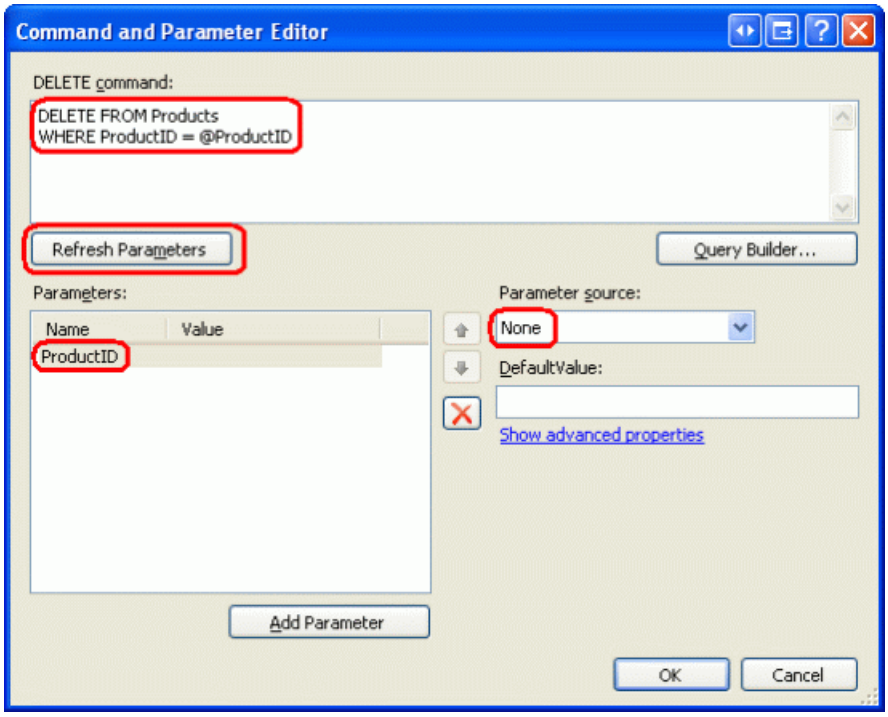


Figure 3: Select the DeleteQuery Property from the Properties Window (Click to view full-size image)

Do *not* provide a value for this parameter (leave its Parameter source at None). Once we add deleting support to the GridView, the GridView will automatically supply this parameter value, using the value of its `DataKeys` collection for the row whose Delete button was clicked.

> **NOTE**
>
> The parameter name used in the `DELETE` query *must* be the same as the name of the `DataKeyNames` value in the GridView, DetailsView, or FormView. That is, the parameter in the `DELETE` statement is purposefully named `@ProductID` (instead of, say, `@ID`), because the primary key column name in the Products table (and therefore the DataKeyNames value in the GridView) is `ProductID` .

If the parameter name and `DataKeyNames` value doesn t match, the GridView cannot automatically assign the parameter the value from the `DataKeys` collection.

After entering the delete-related information into the Command and Parameter Editor dialog box click OK and go to the Source view to examine the resulting declarative markup:

```
<asp:SqlDataSource ID="ProductsDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    SelectCommand=
        "SELECT [ProductID], [ProductName], [UnitPrice] FROM [Products]"
    DeleteCommand="DELETE FROM Products WHERE ProductID = @ProductID">
    <DeleteParameters>
        <asp:Parameter Name="ProductID" />
    </DeleteParameters>
</asp:SqlDataSource>
```

Note the addition of the `DeleteCommand` property as well as the `<DeleteParameters>` section and the Parameter

object named `productID`.

Configuring the GridView for Deleting

With the `DeleteCommand` property added, the GridView s smart tag now contains the Enable Deleting option. Go ahead and check this checkbox. As discussed in An Overview of Inserting, Updating, and Deleting, this causes the GridView to add a CommandField with its `ShowDeleteButton` property set to `True`. As Figure 4 shows, when the page is visited through a browser a Delete button is included. Test this page out by deleting some products.

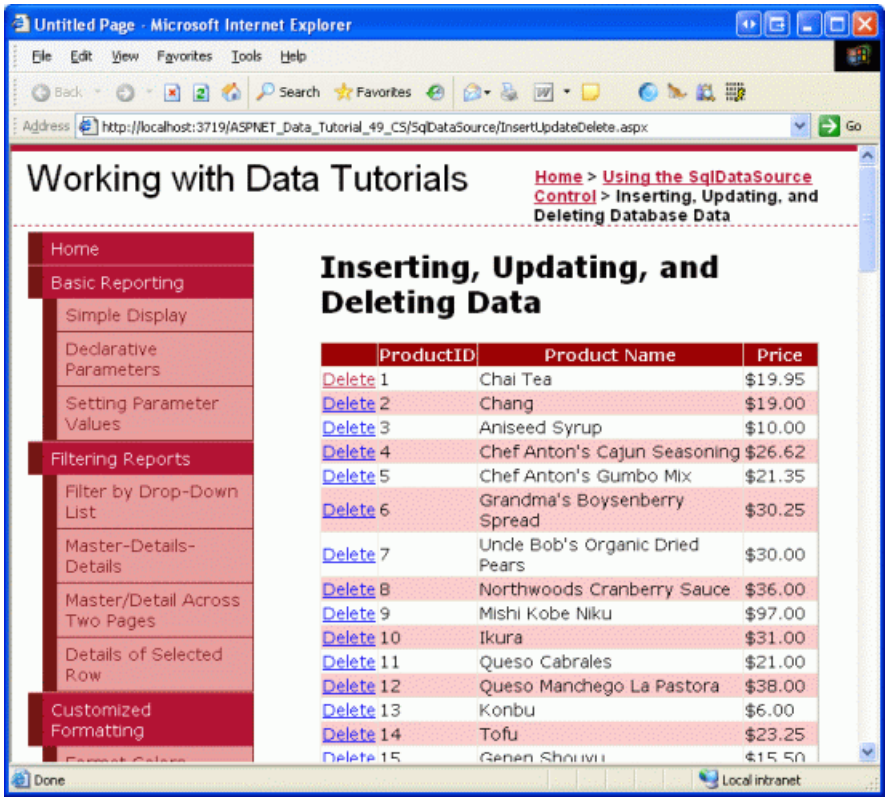


Figure 4: Each GridView Row Now Includes a Delete Button (Click to view full-size image)

Upon clicking a Delete button, a postback occurs, the GridView assigns the `ProductID` parameter the value of the `DataKeys` collection value for the row whose Delete button was clicked, and invokes the SqlDataSource s `Delete()` method. The SqlDataSource control then connects to the database and executes the `DELETE` statement. The GridView then rebinds to the SqlDataSource, getting back and displaying the current set of products (which no longer includes the just-deleted record).

> **NOTE**
>
> Since the GridView uses its `DataKeys` collection to populate the SqlDataSource parameters, it s vital that the GridView s `DataKeyNames` property be set to the column(s) that constitute the primary key and that the SqlDataSource s `SelectCommand` returns these columns. Moreover, it s important that the parameter name in the SqlDataSource s `DeleteCommand` is set to `@ProductID`. If the `DataKeyNames` property is not set or the parameter is not named `@ProductsID`, clicking the Delete button will cause a postback, but won't actually delete any record.

Figure 5 depicts this interaction graphically. Refer back to the Examining the Events Associated with Inserting, Updating, and Deleting tutorial for a more detailed discussion on the chain of events associated with inserting, updating, and deleting from a data Web control.

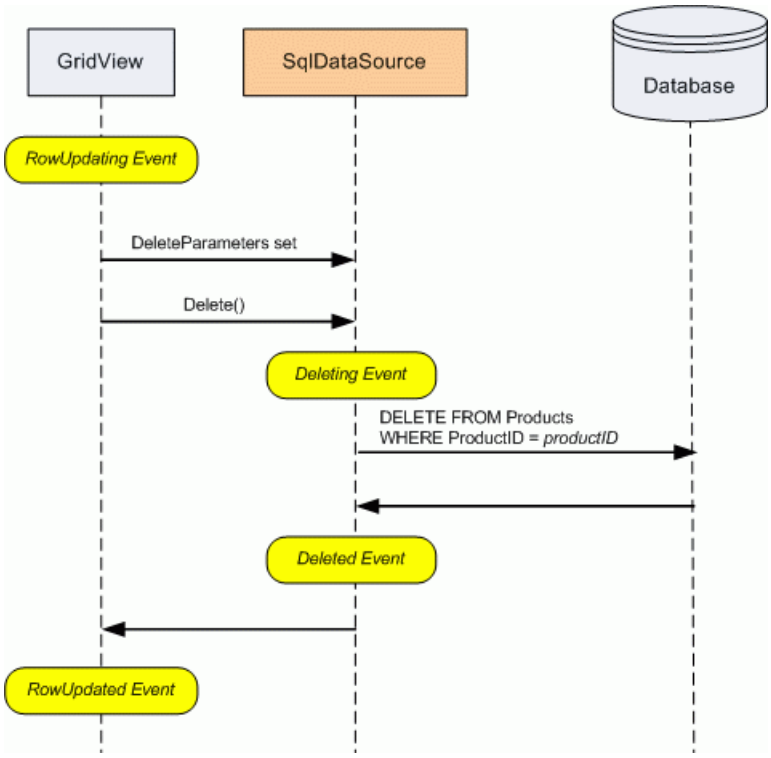


Figure 5: Clicking the Delete Button in the GridView Invokes the SqlDataSource s `Delete()` Method

Step 2: Automatically Generating the `INSERT`, `UPDATE`, and `DELETE` Statements

As Step 1 examined, `INSERT`, `UPDATE`, and `DELETE` SQL statements can be specified through the Properties window or the control s declarative syntax. However, this approach requires that we manually write out the SQL statements by hand, which can be monotonous and error-prone. Fortunately, the Configure Data Source wizard provides an option to have the `INSERT`, `UPDATE`, and `DELETE` statements automatically generated when using the Specify columns from a table of view screen.

Let s explore this automatic generation option. Add a DetailsView to the Designer in `InsertUpdateDelete.aspx` and set its `ID` property to `ManageProducts`. Next, from the DetailsView s smart tag, choose to create a new data source and create a SqlDataSource named `ManageProductsDataSource`.

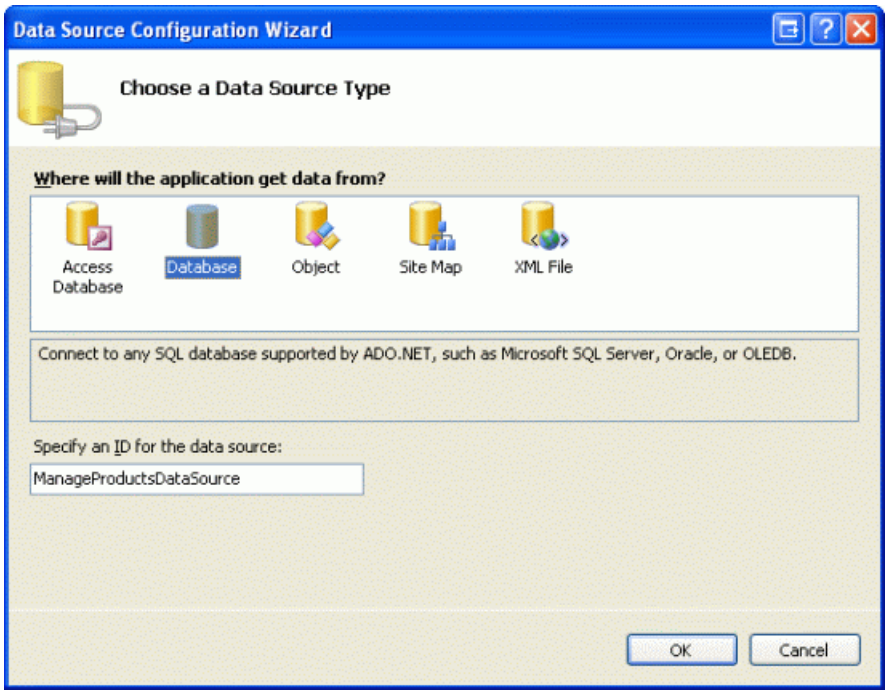


Figure 6: Create a New SqlDataSource Named `ManageProductsDataSource` (Click to view full-size image)

From the Configure Data Source wizard, opt to use the `NORTHWINDConnectionString` connection string and click Next. From the Configure the Select Statement screen, leave the Specify columns from a table or view radio button selected and pick the `Products` table from the drop-down list. Select the `ProductID`, `ProductName`, `UnitPrice`, and `Discontinued` columns from the checkbox list.

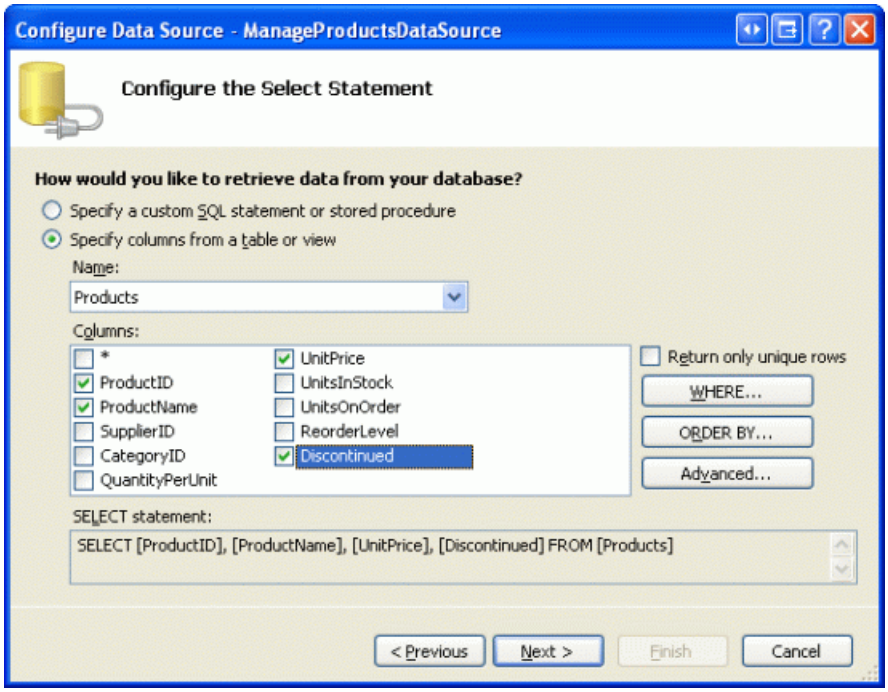


Figure 7: Using the `Products` Table, Return the `ProductID`, `ProductName`, `UnitPrice`, and `Discontinued` Columns (Click to view full-size image)

To automatically generate `INSERT`, `UPDATE`, and `DELETE` statements based on the selected table and columns, click the Advanced button and check the Generate `INSERT`, `UPDATE`, and `DELETE` statements checkbox.

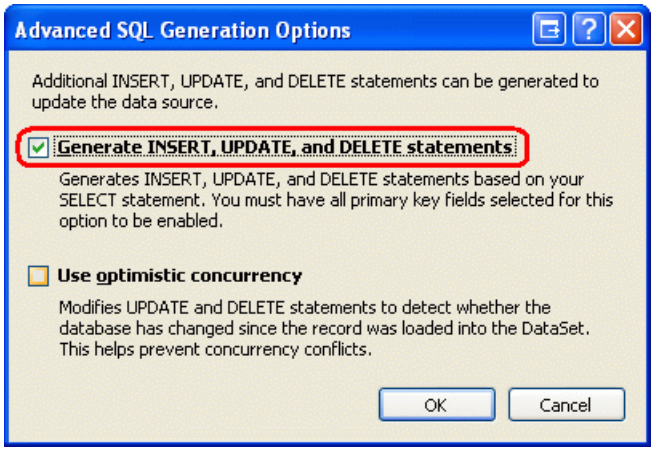


Figure 8: Check the Generate `INSERT`, `UPDATE`, and `DELETE` statements Checkbox

The Generate `INSERT`, `UPDATE`, and `DELETE` statements checkbox will only be checkable if the table selected has a primary key and the primary key column (or columns) are included in the list of returned columns. The Use optimistic concurrency checkbox, which becomes selectable once the Generate `INSERT`, `UPDATE`, and `DELETE` statements checkbox has been checked, will augment the `WHERE` clauses in the resulting `UPDATE` and `DELETE` statements to provide optimistic concurrency control. For now, leave this checkbox unchecked; we'll examine optimistic concurrency with the SqlDataSource control in the next tutorial.

After checking the Generate `INSERT`, `UPDATE`, and `DELETE` statements checkbox, click OK to return to the Configure Select Statement screen, then click Next, and then Finish, to complete the Configure Data Source

wizard. Upon completing the wizard, Visual Studio will add BoundFields to the DetailsView for the `ProductID`, `ProductName`, and `UnitPrice` columns and a CheckBoxField for the `Discontinued` column. From the DetailsView s smart tag, check the Enable Paging option so that the user visiting this page can step through the products. Also clear out the DetailsView s `Width` and `Height` properties.

Notice that the smart tag has the Enable Inserting, Enable Editing, and Enable Deleting options available. This is because the SqlDataSource contains values for its `InsertCommand`, `UpdateCommand`, and `DeleteCommand`, as the following declarative syntax shows:

```
<asp:DetailsView ID="ManageProducts" runat="server" AllowPaging="True"
    AutoGenerateRows="False" DataKeyNames="ProductID"
    DataSourceID="ManageProductsDataSource" EnableViewState="False">
    <Fields>
        <asp:BoundField DataField="ProductID" HeaderText="ProductID"
            InsertVisible="False" ReadOnly="True" SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName" HeaderText="ProductName"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
            SortExpression="UnitPrice" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Fields>
</asp:DetailsView>
<asp:SqlDataSource ID="ManageProductsDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    DeleteCommand=
        "DELETE FROM [Products] WHERE [ProductID] = @ProductID"
    InsertCommand=
        "INSERT INTO [Products] ([ProductName], [UnitPrice], [Discontinued])
         VALUES (@ProductName, @UnitPrice, @Discontinued)"
    SelectCommand=
        "SELECT [ProductID], [ProductName], [UnitPrice], [Discontinued]
         FROM [Products]"
    UpdateCommand=
        "UPDATE [Products] SET [ProductName] = @ProductName,
         [UnitPrice] = @UnitPrice, [Discontinued] = @Discontinued
         WHERE [ProductID] = @ProductID">
    <DeleteParameters>
        <asp:Parameter Name="ProductID" Type="Int32" />
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="ProductName" Type="String" />
        <asp:Parameter Name="UnitPrice" Type="Decimal" />
        <asp:Parameter Name="Discontinued" Type="Boolean" />
        <asp:Parameter Name="ProductID" Type="Int32" />
    </UpdateParameters>
    <InsertParameters>
        <asp:Parameter Name="ProductName" Type="String" />
        <asp:Parameter Name="UnitPrice" Type="Decimal" />
        <asp:Parameter Name="Discontinued" Type="Boolean" />
    </InsertParameters>
</asp:SqlDataSource>
```

Note how the SqlDataSource control has had values automatically set for its `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties. The set of columns referenced in the `InsertCommand` and `UpdateCommand` properties are based on those in the `SELECT` statement. That is, rather than having *every* Products column in the `InsertCommand` and `UpdateCommand`, there are only those columns specified in the `SelectCommand` (less `ProductID`, which is omitted because it s an `IDENTITY` column, whose value cannot be changed when edited and which is automatically assigned when inserting). Moreover, for each parameter in the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties there are corresponding parameters in the `InsertParameters`, `UpdateParameters`, and `DeleteParameters` collections.

To turn on the DetailsView s data modification features, check the Enable Inserting, Enable Editing, and Enable Deleting options in its smart tag. This adds a CommandField with its `ShowInsertButton`, `ShowEditButton`, and `ShowDeleteButton` properties set to `True`.

Visit the page in a browser and note the Edit, Delete, and New buttons included in the DetailsView. Clicking the Edit button turns the DetailsView into edit mode, which displays each BoundField whose `ReadOnly` property is set to `False` (the default) as a TextBox, and the CheckBoxField as a checkbox.

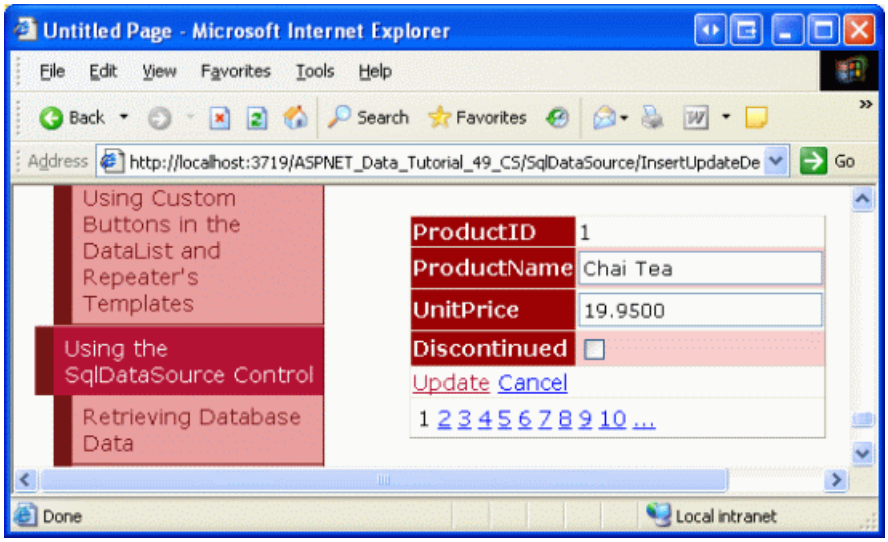


Figure 9: The DetailsView s Default Editing Interface (Click to view full-size image)

Similarly, you can delete the currently selected product or add a new product to the system. Since the `InsertCommand` statement only works with the `ProductName`, `UnitPrice`, and `Discontinued` columns, the other columns have either `NULL` or their default value assigned by the database upon insert. Just like with the ObjectDataSource, if the `InsertCommand` is missing any database table columns that don t allow `NULL` s and don t have a default value, a SQL error will occur when attempting to execute the `INSERT` statement.

> **NOTE**
>
> The DetailsView s inserting and editing interfaces lack any sort of customization or validation. To add validation controls or to customize the interfaces, you need to convert the BoundFields to TemplateFields. Refer to the Adding Validation Controls to the Editing and Inserting Interfaces and Customizing the Data Modification Interface tutorials for more information.

Also, keep in mind that for updating and deleting, the DetailsView uses the current product s `DataKey` value, which is only present if the `DataKeyNames` property is configured. If editing or deleting appears to have no effect, ensure that the `DataKeyNames` property is set.

Limitations of Automatically Generating SQL Statements

Since the Generate `INSERT`, `UPDATE`, and `DELETE` statements option is only available when picking columns from a table, for more complex queries you will have to write your own `INSERT`, `UPDATE`, and `DELETE` statements like we did in Step 1. Commonly, SQL `SELECT` statements use `JOIN` s to bring back data from one or more lookup tables for display purposes (such as bringing back the `Categories` table s `CategoryName` field when displaying product information). At the same time, we might want to allow the user to edit, update, or insert data into the core table (`Products`, in this case).

While the `INSERT`, `UPDATE`, and `DELETE` statements can be entered manually, consider the following time-saving tip. Initially setup the SqlDataSource so that it pulls back data just from the `Products` table. Use the Configure Data Source wizard s Specify columns from a table or view screen so that you can automatically generate the `INSERT`, `UPDATE`, and `DELETE` statements. Then, after completing the wizard, choose to configure the SelectQuery

from the Properties window (or, alternatively, go back to the Configure Data Source wizard, but use the Specify a custom SQL statement or stored procedure option). Then update the `SELECT` statement to include the `JOIN` syntax. This technique offers the time-saving benefits of the automatically generated SQL statements and allows for a more customized `SELECT` statement.

Another limitation of automatically generating the `INSERT`, `UPDATE`, and `DELETE` statements is that the columns in the `INSERT` and `UPDATE` statements are based on the columns returned by the `SELECT` statement. We may need to update or insert more or fewer fields, however. For example, in the example from Step 2, maybe we want to have the `UnitPrice` BoundField be read-only. In that case, it shouldn't appear in the `UpdateCommand`. Or we may want to set the value of a table field that does not appear in the GridView. For example, when adding a new record we may want the `QuantityPerUnit` value set to TODO .

If such customizations are required, you need to make them manually, either through the Properties window, the Specify a custom SQL statement or stored procedure option in the wizard, or via the declarative syntax.

> **NOTE**
> When adding parameters that do not have corresponding fields in the data Web control, keep in mind that these parameters values will need to be assigned values in some manner. These values can be: hard-coded directly in the `InsertCommand` or `UpdateCommand`; can come from some pre-defined source (the querystring, session state, Web controls on the page, and so on); or can be assigned programmatically, as we saw in the preceding tutorial.

Summary

In order for the data Web controls to utilize their built-in inserting, editing, and deleting capabilities, the data source control they are bound to must offer such functionality. For the SqlDataSource, this means that `INSERT`, `UPDATE`, and `DELETE` SQL statements must be assigned to the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties. These properties, and the corresponding parameters collections, can be added manually or generated automatically through the Configure Data Source wizard. In this tutorial we examined both techniques.

We examined using optimistic concurrency with the ObjectDataSource in the Implementing Optimistic Concurrency tutorial. The SqlDataSource control also provides optimistic concurrency support. As noted in Step 2, when automatically generating the `INSERT`, `UPDATE`, and `DELETE` statements, the wizard offers a Use optimistic concurrency option. As we'll see in the next tutorial, using optimistic concurrency with the SqlDataSource modifies the `WHERE` clauses in the `UPDATE` and `DELETE` statements to ensure that the values for the other columns haven't changed since the data was last displayed on the page.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Implementing Optimistic Concurrency with the SqlDataSource (VB)

2/20/2020 • 16 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we review the essentials of optimistic concurrency control and then explore how to implement it using the SqlDataSource control.

Introduction

In the preceding tutorial we examined how to add inserting, updating, and deleting capabilities to the SqlDataSource control. In short, to provide these features we needed to specify the corresponding `INSERT`, `UPDATE`, or `DELETE` SQL statement in the control s `InsertCommand`, `UpdateCommand`, or `DeleteCommand` properties, along with the appropriate parameters in the `InsertParameters`, `UpdateParameters`, and `DeleteParameters` collections. While these properties and collections can be specified manually, the Configure Data Source wizard s Advanced button offers a Generate `INSERT`, `UPDATE`, and `DELETE` statements checkbox that will auto-create these statements based on the `SELECT` statement.

Along with the Generate `INSERT`, `UPDATE`, and `DELETE` statements checkbox, the Advanced SQL Generation Options dialog box includes a Use optimistic concurrency option (see Figure 1). When checked, the `WHERE` clauses in the autogenerated `UPDATE` and `DELETE` statements are modified to only perform the update or delete if the underlying database data hasn't been modified since the user last loaded the data into the grid.

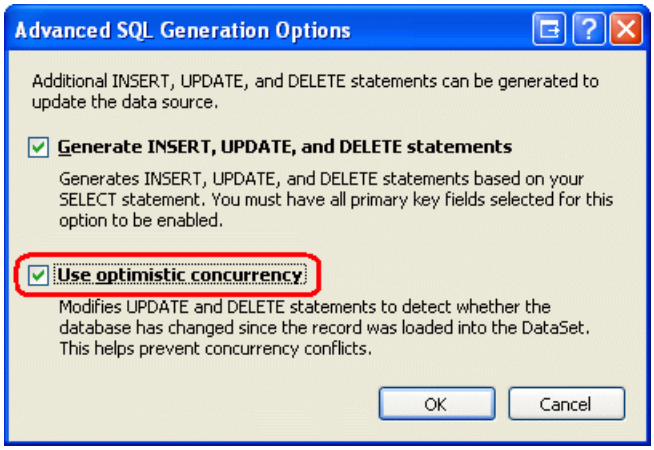


Figure 1: You Can Add Optimistic Concurrency Support from the Advanced SQL Generation Options Dialog Box

Back in the Implementing Optimistic Concurrency tutorial we examined the fundamentals of optimistic concurrency control and how to add it to the ObjectDataSource. In this tutorial we'll retouch on the essentials of optimistic concurrency control and then explore how to implement it using the SqlDataSource.

A Recap of Optimistic Concurrency

For web applications that allow multiple, simultaneous users to edit or delete the same data, there exists a possibility that one user may accidentally overwrite another s changes. In the Implementing Optimistic Concurrency tutorial I provided the following example:

Imagine that two users, Jisun and Sam, were both visiting a page in an application that allowed visitors to update and delete products through a GridView control. Both click the Edit button for Chai around the same time. Jisun changes the product name to Chai Tea and clicks the Update button. The net result is an `UPDATE` statement that is sent to the database, which sets *all* of the product s updateable fields (even though Jisun only updated one field, `ProductName`). At this point in time, the database has the values Chai Tea, the category Beverages, the supplier Exotic Liquids, and so on for this particular product. However, the GridView on Sam s screen still shows the product name in the editable GridView row as Chai. A few seconds after Jisun s changes have been committed, Sam updates the category to Condiments and clicks Update. This results in an `UPDATE` statement sent to the database that sets the product name to Chai, the `CategoryID` to the corresponding Condiments category ID, and so on. Jisun s changes to the product name have been overwritten.

Figure 2 illustrates this interaction.

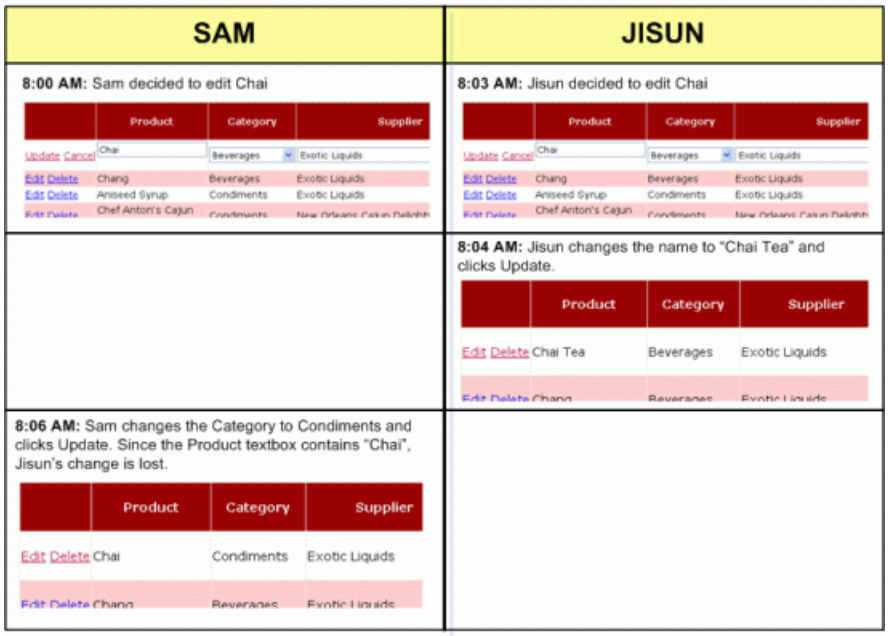


Figure 2: When Two Users Simultaneously Update a Record There s Potential for One User s Changes to Overwrite the Other s (Click to view full-size image)

To prevent this scenario from unfolding, a form of concurrency control must be implemented. Optimistic concurrency the focus of this tutorial works on the assumption that while there may be concurrency conflicts every now and then, the vast majority of the time such conflicts won't arise. Therefore, if a conflict does arise, optimistic concurrency control simply informs the user that their changes can t be saved because another user has modified the same data.

> **NOTE**
>
> For applications where it is assumed that there will be many concurrency conflicts or if such conflicts are not tolerable, then pessimistic concurrency control can be used instead. Refer back to the Implementing Optimistic Concurrency tutorial for a more thorough discussion on pessimistic concurrency control.

Optimistic concurrency control works by ensuring that the record being updated or deleted has the same values as it did when the updating or deleting process started. For example, when clicking the Edit button in an editable GridView, the record s values are read from the database and displayed in TextBoxes and other Web controls. These original values are saved by the GridView. Later, after the user makes her changes and clicks the Update button, the `UPDATE` statement used must take into account the original values plus the new values and only update the underlying database record if the original values that the user started editing are identical to the values still in the database. Figure 3 depicts this sequence of events.

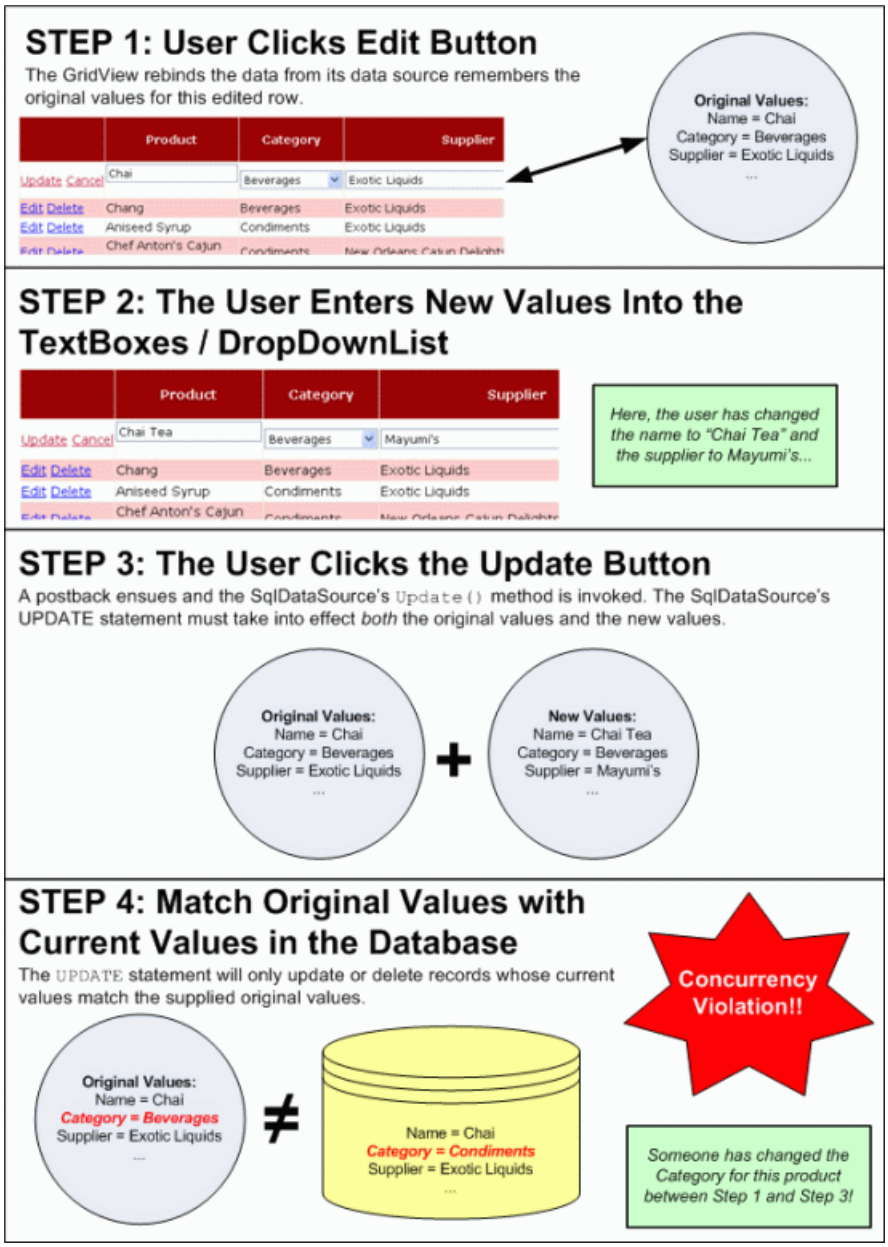


Figure 3: For the Update or Delete to Succeed, the Original Values Must Be Equal to the Current Database Values (Click to view full-size image)

There are various approaches to implementing optimistic concurrency (see Peter A. Bromberg's Optimistic Concurrency Updating Logic for a brief look at a number of options). The technique used by the SqlDataSource (as well as by the ADO.NET Typed DataSets used in our Data Access Layer) augments the `WHERE` clause to include a comparison of all of the original values. The following `UPDATE` statement, for example, updates the name and price of a product only if the current database values are equal to the values that were originally retrieved when updating the record in the GridView. The `@ProductName` and `@UnitPrice` parameters contain the new values entered by the user, whereas `@original_ProductName` and `@original_UnitPrice` contain the values that were originally loaded into the GridView when the Edit button was clicked:

```
UPDATE Products SET
    ProductName = @ProductName,
    UnitPrice = @UnitPrice
WHERE
    ProductID = @original_ProductID AND
    ProductName = @original_ProductName AND
    UnitPrice = @original_UnitPrice
```

As we'll see in this tutorial, enabling optimistic concurrency control with the SqlDataSource is as simple as checking a checkbox.

Step 1: Creating a SqlDataSource that Supports Optimistic Concurrency

Start by opening the `OptimisticConcurrency.aspx` page from the `SqlDataSource` folder. Drag a SqlDataSource control from the Toolbox onto the Designer, settings its `ID` property to `ProductsDataSourceWithOptimisticConcurrency`. Next, click on the Configure Data Source link from the control s smart tag. From the first screen in the wizard, choose to work with the `NORTHWINDConnectionString` and click Next.

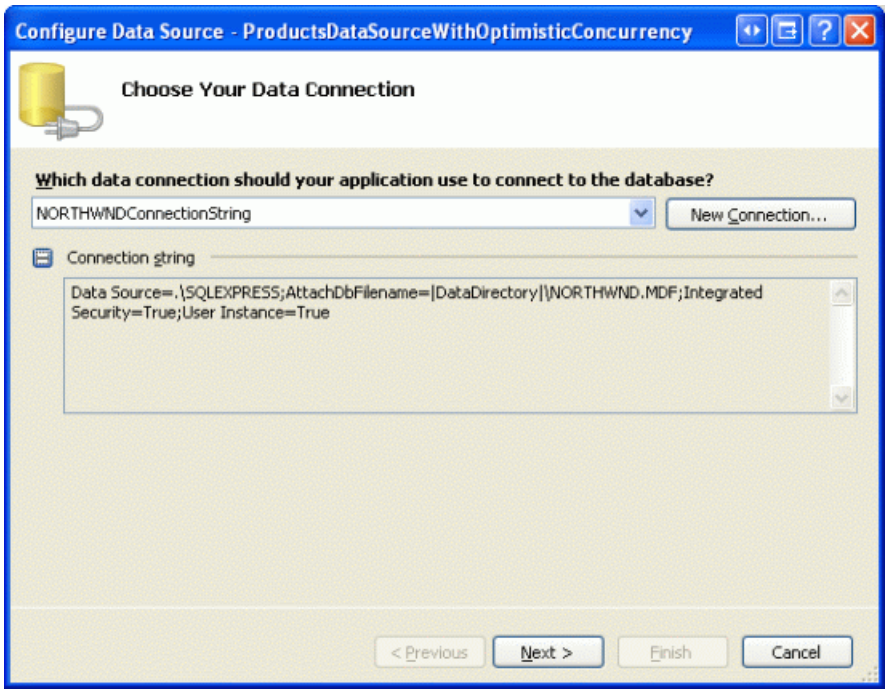


Figure 4: Choose to Work with the `NORTHWINDConnectionString` (Click to view full-size image)

For this example we'll be adding a GridView that enables users to edit the `Products` table. Therefore, from the Configure the Select Statement screen, choose the `Products` table from the drop-down list and select the `ProductID`, `ProductName`, `UnitPrice`, and `Discontinued` columns, as shown in Figure 5.

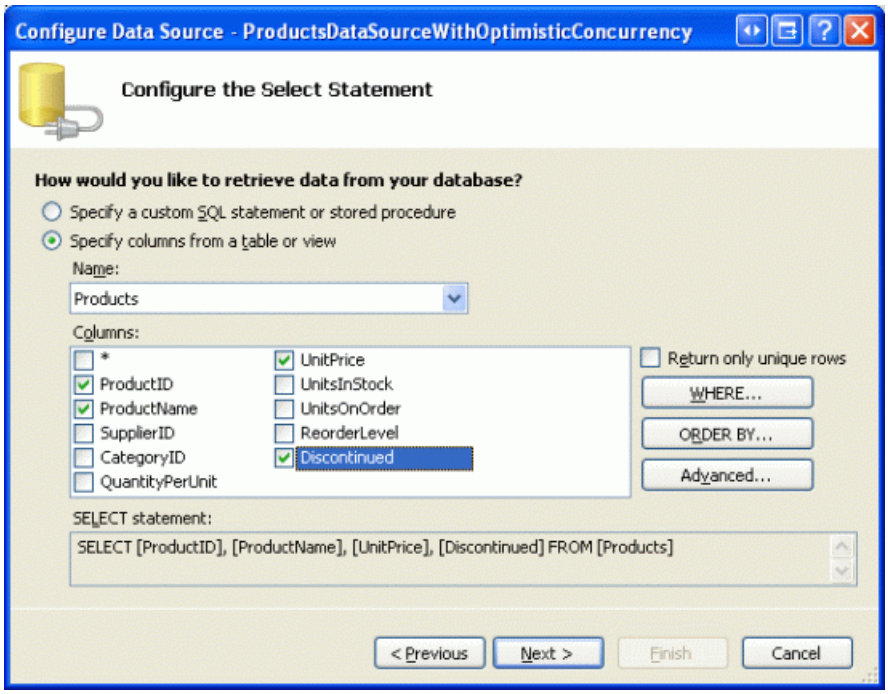


Figure 5: From the `Products` Table, Return the `ProductID`, `ProductName`, `UnitPrice`, and `Discontinued` Columns (Click to view full-size image)

After picking the columns, click the Advanced button to bring up the Advanced SQL Generation Options dialog box. Check the Generate `INSERT`, `UPDATE`, and `DELETE` statements and Use optimistic concurrency checkboxes and click OK (refer back to Figure 1 for a screenshot). Complete the wizard by clicking Next, then Finish.

After completing the Configure Data Source wizard, take a moment to examine the resulting `DeleteCommand` and `UpdateCommand` properties and the `DeleteParameters` and `UpdateParameters` collections. The easiest way to do this is to click on the Source tab in the lower left corner to see the page s declarative syntax. There you will find an `UpdateCommand` value of:

```
UPDATE [Products] SET
     [ProductName] = @ProductName,
     [UnitPrice] = @UnitPrice,
     [Discontinued] = @Discontinued
WHERE
     [ProductID] = @original_ProductID AND
     [ProductName] = @original_ProductName AND
     [UnitPrice] = @original_UnitPrice AND
     [Discontinued] = @original_Discontinued
```

With seven parameters in the `UpdateParameters` collection:

```
<asp:SqlDataSource ID="ProductsDataSourceWithOptimisticConcurrency"
    runat="server" ...>
    <DeleteParameters>
      ...
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="ProductName" Type="String" />
        <asp:Parameter Name="UnitPrice" Type="Decimal" />
        <asp:Parameter Name="Discontinued" Type="Boolean" />
        <asp:Parameter Name="original_ProductID" Type="Int32" />
        <asp:Parameter Name="original_ProductName" Type="String" />
        <asp:Parameter Name="original_UnitPrice" Type="Decimal" />
        <asp:Parameter Name="original_Discontinued" Type="Boolean" />
    </UpdateParameters>
    ...
</asp:SqlDataSource>
```

Similarly, the `DeleteCommand` property and `DeleteParameters` collection should look like the following:

```
DELETE FROM [Products]
WHERE
     [ProductID] = @original_ProductID AND
     [ProductName] = @original_ProductName AND
     [UnitPrice] = @original_UnitPrice AND
     [Discontinued] = @original_Discontinued
```

```
<asp:SqlDataSource ID="ProductsDataSourceWithOptimisticConcurrency"
    runat="server" ...>
    <DeleteParameters>
        <asp:Parameter Name="original_ProductID" Type="Int32" />
        <asp:Parameter Name="original_ProductName" Type="String" />
        <asp:Parameter Name="original_UnitPrice" Type="Decimal" />
        <asp:Parameter Name="original_Discontinued" Type="Boolean" />
    </DeleteParameters>
    <UpdateParameters>
        ...
    </UpdateParameters>
    ...
</asp:SqlDataSource>
```

In addition to augmenting the `WHERE` clauses of the `UpdateCommand` and `DeleteCommand` properties (and adding the additional parameters to the respective parameter collections), selecting the Use optimistic concurrency option adjusts two other properties:

- Changes the `ConflictDetection` property from `OverwriteChanges` (the default) to `CompareAllValues`
- Changes the `OldValuesParameterFormatString` property from {0} (the default) to original_{0} .

When the data Web control invokes the SqlDataSource s `Update()` or `Delete()` method, it passes in the original values. If the SqlDataSource s `ConflictDetection` property is set to `CompareAllValues`, these original values are added to the command. The `OldValuesParameterFormatString` property provides the naming pattern used for these original value parameters. The Configure Data Source wizard uses original_{0} and names each original parameter in the `UpdateCommand` and `DeleteCommand` properties and `UpdateParameters` and `DeleteParameters` collections accordingly.

> **NOTE**
> Since we re not using the SqlDataSource control s inserting capabilities, feel free to remove the `InsertCommand` property and its `InsertParameters` collection.

Correctly Handling `NULL` Values

Unfortunately, the augmented `UPDATE` and `DELETE` statements autogenerated by the Configure Data Source wizard when using optimistic concurrency do *not* work with records that contain `NULL` values. To see why, consider our SqlDataSource s `UpdateCommand`:

```
UPDATE [Products] SET
     [ProductName] = @ProductName,
     [UnitPrice] = @UnitPrice,
     [Discontinued] = @Discontinued
WHERE
     [ProductID] = @original_ProductID AND
     [ProductName] = @original_ProductName AND
     [UnitPrice] = @original_UnitPrice AND
     [Discontinued] = @original_Discontinued
```

The `UnitPrice` column in the `Products` table can have `NULL` values. If a particular record has a `NULL` value for `UnitPrice`, the `WHERE` clause portion `[UnitPrice] = @original_UnitPrice` will *always* evaluate to False because `NULL = NULL` always returns False. Therefore, records that contain `NULL` values cannot be edited or deleted, as the `UPDATE` and `DELETE` statements `WHERE` clauses won't return any rows to update or delete.

> **NOTE**
> This bug was first reported to Microsoft in June of 2004 in SqlDataSource Generates Incorrect SQL Statements and is reportedly scheduled to be fixed in the next version of ASP.NET.

To fix this, we have to manually update the `WHERE` clauses in both the `UpdateCommand` and `DeleteCommand` properties for **all** columns that can have `NULL` values. In general, change `[ColumnName] = @original_ColumnName` to:

```
(
   ([ColumnName] IS NULL AND @original_ColumnName IS NULL)
     OR
   ([ColumnName] = @original_ColumnName)
)
```

This modification can be made directly through the declarative markup, via the UpdateQuery or DeleteQuery options from the Properties window, or through the UPDATE and DELETE tabs in the Specify a custom SQL statement or stored procedure option in the Configure Data Source wizard. Again, this modification must be made for *every* column in the `UpdateCommand` and `DeleteCommand` s `WHERE` clause that can contain `NULL` values.

Applying this to our example results in the following modified `UpdateCommand` and `DeleteCommand` values:

```
UPDATE [Products] SET
     [ProductName] = @ProductName,
     [UnitPrice] = @UnitPrice,
     [Discontinued] = @Discontinued
WHERE
     [ProductID] = @original_ProductID AND
     [ProductName] = @original_ProductName AND
     (([UnitPrice] IS NULL AND @original_UnitPrice IS NULL)
        OR ([UnitPrice] = @original_UnitPrice)) AND
     [Discontinued] = @original_Discontinued
DELETE FROM [Products]
WHERE
     [ProductID] = @original_ProductID AND
     [ProductName] = @original_ProductName AND
     (([UnitPrice] IS NULL AND @original_UnitPrice IS NULL)
        OR ([UnitPrice] = @original_UnitPrice)) AND
     [Discontinued] = @original_Discontinued
```

Step 2: Adding a GridView with Edit and Delete Options

With the SqlDataSource configured to support optimistic concurrency, all that remains is to add a data Web control to the page that utilizes this concurrency control. For this tutorial, let s add a GridView that provides both edit and delete functionality. To accomplish this, drag a GridView from the Toolbox onto the Designer and set its `ID` to `Products`. From the GridView s smart tag, bind it to the `ProductsDataSourceWithOptimisticConcurrency` SqlDataSource control added in Step 1. Finally, check the Enable Editing and Enable Deleting options from the smart tag.

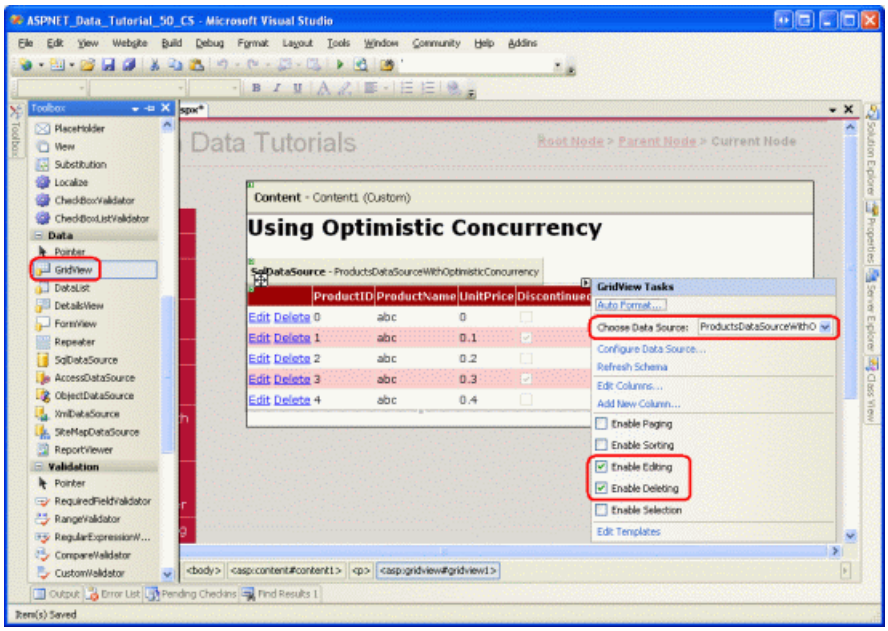


Figure 6: Bind the GridView to the SqlDataSource and Enable Editing and Deleting (Click to view full-size image)

After adding the GridView, configure its appearance by removing the `ProductID` BoundField, changing the `ProductName` BoundField s `HeaderText` property to Product, and updating the `UnitPrice` BoundField so that its `HeaderText` property is simply Price. Ideally, we d enhance the editing interface to include a RequiredFieldValidator for the `ProductName` value and a CompareValidator for the `UnitPrice` value (to ensure it s a properly formatted numeric value). Refer to the Customizing the Data Modification Interface tutorial for a more in-depth look at customizing the GridView s editing interface.

> **NOTE**
>
> The GridView s view state must be enabled since the original values passed from the GridView to the SqlDataSource are stored in view state.

After making these modifications to the GridView, the GridView and SqlDataSource declarative markup should look similar to the following:

```
<asp:SqlDataSource ID="ProductsDataSourceWithOptimisticConcurrency"
    runat="server" ConflictDetection="CompareAllValues"
    ConnectionString="<%$ ConnectionStrings:NORTHWNDConnectionString %>"
    DeleteCommand=
        "DELETE FROM [Products]
         WHERE [ProductID] = @original_ProductID
         AND [ProductName] = @original_ProductName
         AND (([UnitPrice] IS NULL AND @original_UnitPrice IS NULL)
              OR ([UnitPrice] = @original_UnitPrice))
         AND [Discontinued] = @original_Discontinued"
    OldValuesParameterFormatString=
        "original_{0}"
    SelectCommand=
        "SELECT [ProductID], [ProductName], [UnitPrice], [Discontinued]
         FROM [Products]"
    UpdateCommand=
        "UPDATE [Products]
         SET [ProductName] = @ProductName, [UnitPrice] = @UnitPrice,
            [Discontinued] = @Discontinued
         WHERE [ProductID] = @original_ProductID
         AND [ProductName] = @original_ProductName
         AND (([UnitPrice] IS NULL AND @original_UnitPrice IS NULL)
            OR ([UnitPrice] = @original_UnitPrice))
        AND [Discontinued] = @original_Discontinued">
    <DeleteParameters>
        <asp:Parameter Name="original_ProductID" Type="Int32" />
        <asp:Parameter Name="original_ProductName" Type="String" />
        <asp:Parameter Name="original_UnitPrice" Type="Decimal" />
        <asp:Parameter Name="original_Discontinued" Type="Boolean" />
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="ProductName" Type="String" />
        <asp:Parameter Name="UnitPrice" Type="Decimal" />
        <asp:Parameter Name="Discontinued" Type="Boolean" />
        <asp:Parameter Name="original_ProductID" Type="Int32" />
        <asp:Parameter Name="original_ProductName" Type="String" />
        <asp:Parameter Name="original_UnitPrice" Type="Decimal" />
        <asp:Parameter Name="original_Discontinued" Type="Boolean" />
    </UpdateParameters>
</asp:SqlDataSource>
<asp:GridView ID="Products" runat="server"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ProductsDataSourceWithOptimisticConcurrency">
    <Columns>
        <asp:CommandField ShowDeleteButton="True" ShowEditButton="True" />
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" HeaderText="Price"
            SortExpression="UnitPrice" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
```

To see the optimistic concurrency control in action, open two browser windows and load the `OptimisticConcurrency.aspx` page in both. Click on the Edit buttons for the first product in both browsers. In one browser, change the product name and click Update. The browser will postback and the GridView will return to its pre-editing mode, showing the new product name for the record just edited.

In the second browser window, change the price (but leave the product name as its original value) and click Update. On postback, the grid returns to its pre-editing mode, but the change to the price is not recorded. The second browser shows the same value as the first one the new product name with the old price. The changes made in the second browser window were lost. Moreover, the changes were lost rather quietly, as there was no exception or message indicating that a concurrency violation just occurred.

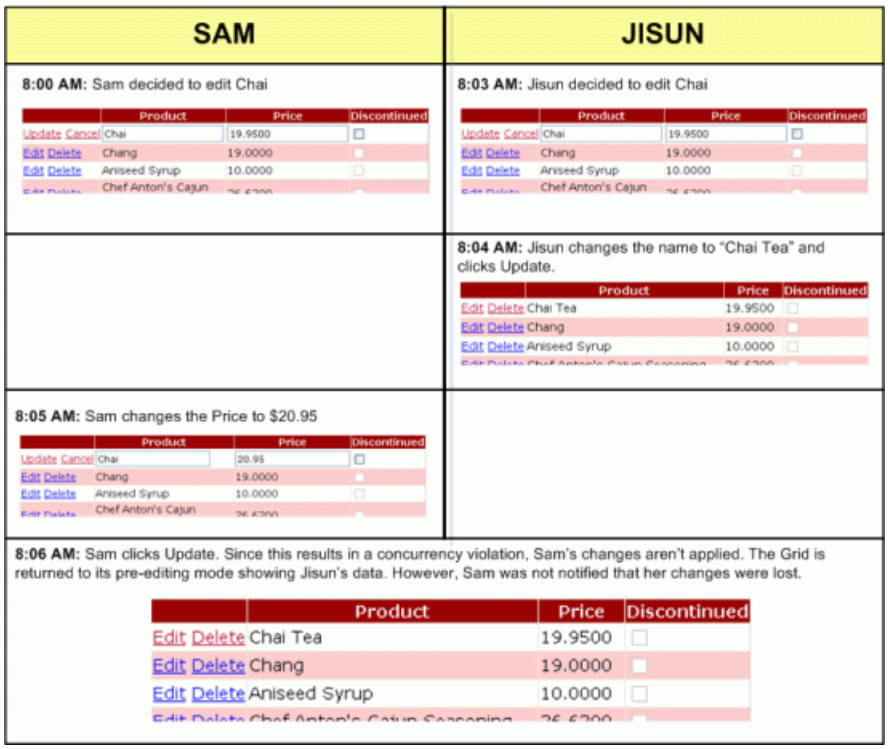


Figure 7: The Changes in the Second Browser Window Were Silently Lost (Click to view full-size image)

The reason why the second browser s changes were not committed was because the `UPDATE` statement s `WHERE` clause filtered out all records and therefore did not affect any rows. Let s look at the `UPDATE` statement again:

```
UPDATE [Products] SET
     [ProductName] = @ProductName,
     [UnitPrice] = @UnitPrice,
     [Discontinued] = @Discontinued
WHERE
     [ProductID] = @original_ProductID AND
     [ProductName] = @original_ProductName AND
     (([UnitPrice] IS NULL AND @original_UnitPrice IS NULL) OR
        ([UnitPrice] = @original_UnitPrice)) AND
     [Discontinued] = @original_Discontinued
```

When the second browser window updates the record, the original product name specified in the `WHERE` clause doesn t match up with the existing product name (since it was changed by the first browser). Therefore, the statement `[ProductName] = @original_ProductName` returns False, and the `UPDATE` does not affect any records.

> **NOTE**
> Delete works in the same manner. With two browser windows open, start by editing a given product with one, and then saving its changes. After saving the changes in the one browser, click the Delete button for the same product in the other. Since the original values don t match up in the `DELETE` statement s `WHERE` clause, the delete silently fails.

From the end user s perspective in the second browser window, after clicking the Update button the grid returns to the pre-editing mode, but their changes were lost. However, there s no visual feedback that their changes didn't stick. Ideally, if a user s changes are lost to a concurrency violation, we d notify them and, perhaps, keep the grid in edit mode. Let s look at how to accomplish this.

Step 3: Determining When a Concurrency Violation Has Occurred

Since a concurrency violation rejects the changes one has made, it would be nice to alert the user when a concurrency violation has occurred. To alert the user, let s add a Label Web control to the top of the page named

`ConcurrencyViolationMessage` whose `Text` property displays the following message: You have attempted to update or delete a record that was simultaneously updated by another user. Please review the other user's changes and then redo your update or delete. Set the Label control s `CssClass` property to Warning, which is a CSS class defined in `Styles.css` that displays text in a red, italic, bold, and large font. Finally, set the Label s `Visible` and `EnableViewState` properties to `False`. This will hide the Label except for only those postbacks where we explicitly set its `Visible` property to `True`.

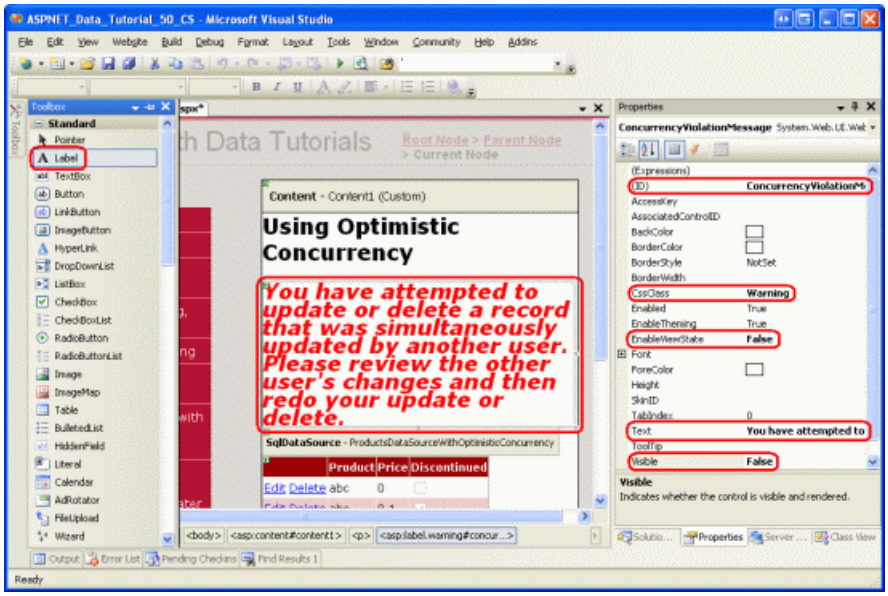


Figure 8: Add a Label Control to the Page to Display the Warning (Click to view full-size image)

When performing an update or delete, the GridView s `RowUpdated` and `RowDeleted` event handlers fire after its data source control has performed the requested update or delete. We can determine how many rows were affected by the operation from these event handlers. If zero rows were affected, we want to display the `ConcurrencyViolationMessage` Label.

Create an event handler for both the `RowUpdated` and `RowDeleted` events and add the following code:

```
Protected Sub Products_RowUpdated(sender As Object, e As GridViewUpdatedEventArgs) _
    Handles Products.RowUpdated
    If e.AffectedRows = 0 Then
        ConcurrencyViolationMessage.Visible = True
        e.KeepInEditMode = True
        ' Rebind the data to the GridView to show the latest changes
        Products.DataBind()
    End If
End Sub
Protected Sub Products_RowDeleted(sender As Object, e As GridViewDeletedEventArgs) _
    Handles Products.RowDeleted
    If e.AffectedRows = 0 Then
        ConcurrencyViolationMessage.Visible = True
    End If
End Sub
```

In both event handlers we check the `e.AffectedRows` property and, if it equals 0, set the `ConcurrencyViolationMessage` Label s `Visible` property to `True`. In the `RowUpdated` event handler, we also instruct the GridView to stay in edit mode by setting the `KeepInEditMode` property to true. In doing so, we need to rebind the data to the grid so that the other user s data is loaded into the editing interface. This is accomplished by calling the GridView s `DataBind()` method.

As Figure 9 shows, with these two event handlers, a very noticeable message is displayed whenever a concurrency violation occurs.

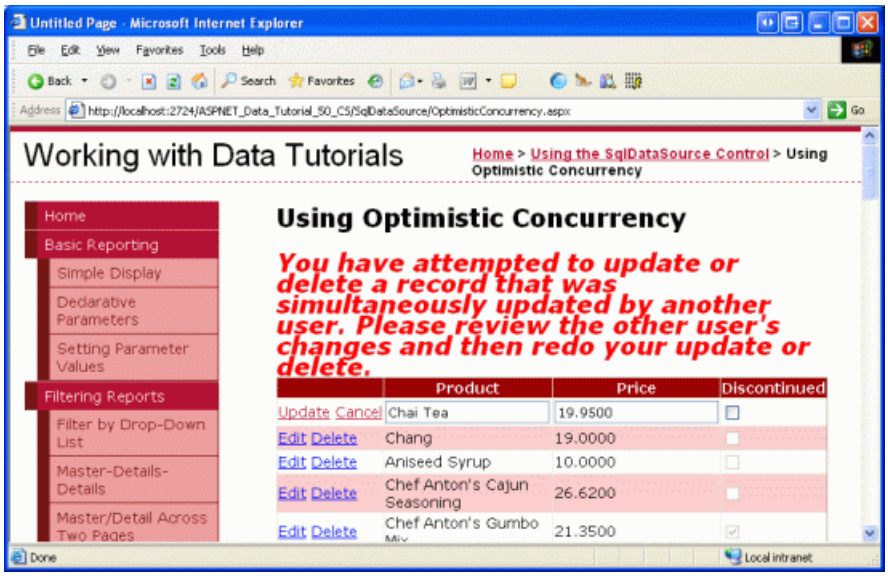


Figure 9: A Message is Displayed in the Face of a Concurrency Violation (Click to view full-size image)

Summary

When creating a web application where multiple, concurrent users may be editing the same data, it is important to consider concurrency control options. By default, the ASP.NET data Web controls and data source controls do not employ any concurrency control. As we saw in this tutorial, implementing optimistic concurrency control with the SqlDataSource is relatively quick and easy. The SqlDataSource handles most of the legwork for your adding augmented `WHERE` clauses to the autogenerated `UPDATE` and `DELETE` statements but there are a few subtleties in handling `NULL` value columns, as discussed in the Correctly Handling `NULL` Values section.

This tutorial concludes our examination of the SqlDataSource. Our remaining tutorials will return to working with data using the ObjectDataSource and tiered architecture.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS

Enhancing the GridView

2/20/2020 • 2 minutes to read • Edit Online

These tutorials show how to add a column of radio buttons to a GridView control to provide an intuitive way to select a single row.

- Adding a GridView Column of Radio Buttons (C#)
- Adding a GridView Column of Checkboxes (C#)
- Inserting a New Record from the GridView's Footer (C#)
- Adding a GridView Column of Radio Buttons (VB)
- Adding a GridView Column of Checkboxes (VB)
- Inserting a New Record from the GridView's Footer (VB)

Adding a GridView Column of Radio Buttons (C#)

2/20/2020 • 26 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> This tutorial looks at how to add a column of radio buttons to a GridView control to provide the user with a more intuitive way of selecting a single row of the GridView.

Introduction

The GridView control offers a great deal of built-in functionality. It includes a number of different fields for displaying text, images, hyperlinks, and buttons. It supports templates for further customization. With a few clicks of the mouse, it s possible to make a GridView where each row can be selected via a button, or to enable editing or deleting capabilities. Despite the plethora of provided features, there will often be situations in which additional, non-supported features will need to be added. In this tutorial and the next two we'll examine how to enhance the GridView s functionality to include additional features.

This tutorial and the next one focus on enhancing the row-selection process. As examined in the Master/Detail Using a Selectable Master GridView with a Details DetailView, we can add a CommandField to the GridView that includes a Select button. When clicked, a postback ensues and the GridView s `SelectedIndex` property is updated to the index of the row whose Select button was clicked. In the Master/Detail Using a Selectable Master GridView with a Details DetailView tutorial, we saw how to use this feature to display details for the selected GridView row.

While the Select button works in many situations, it may not work as well for others. Rather than using a button, two other user interface elements are commonly used for selection: the radio button and checkbox. We can augment the GridView so that instead of a Select button, each row contains a radio button or checkbox. In scenarios where the user can only select one of the GridView records, the radio button might be preferred over the Select button. In situations where the user can potentially select multiple records such as in a web-based email application, where a user might want to select multiple messages to delete the checkbox offers functionality that is not available from the Select button or radio button user interfaces.

This tutorial looks at how to add a column of radio buttons to the GridView. The proceeding tutorial explores using checkboxes.

Step 1: Creating the Enhancing the GridView Web Pages

Before we start enhancing the GridView to include a column of radio buttons, let s first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial and the next two. Start by adding a new folder named `EnhancedGridView`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `RadioButtonField.aspx`
- `CheckBoxField.aspx`
- `InsertThroughFooter.aspx`

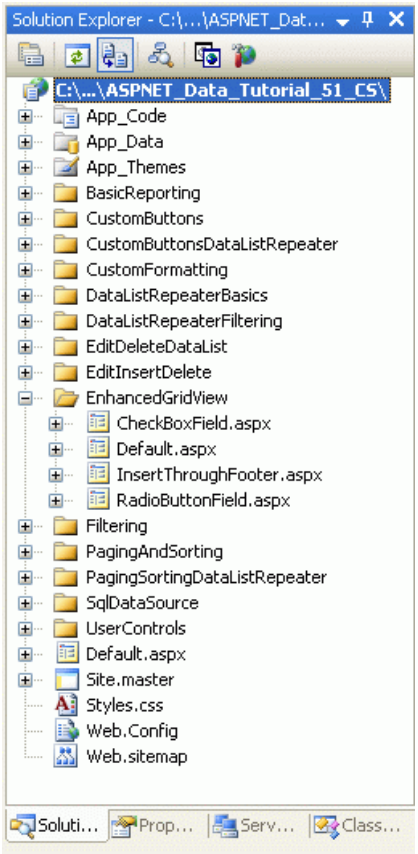


Figure 1: Add the ASP.NET Pages for the SqlDataSource-Related Tutorials

Like in the other folders, `Default.aspx` in the `EnhancedGridView` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

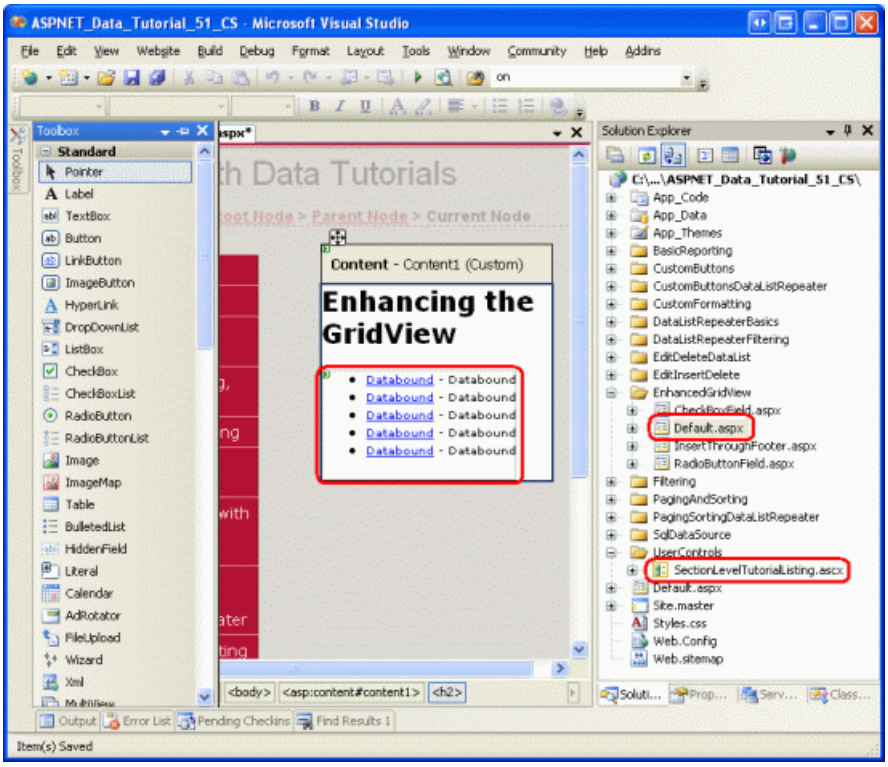


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add these four pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Using the SqlDataSource Control `<siteMapNode>`:

```
<siteMapNode
    title="Enhancing the GridView"
    url="~/EnhancedGridView/Default.aspx"
    description="Augment the user experience of the GridView control.">
    <siteMapNode
        url="~/EnhancedGridView/RadioButtonField.aspx"
        title="Selection via a Radio Button Column"
        description="Explore how to add a column of radio buttons in the GridView." />
    <siteMapNode
        url="~/EnhancedGridView/CheckBoxField.aspx"
        title="Selection via a Checkbox Column"
        description="Select multiple records in the GridView by using a column of
            checkboxes." />
    <siteMapNode
        url="~/EnhancedGridView/InsertThroughFooter.aspx"
        title="Add New Records through the Footer"
        description="Learn how to allow users to add new records through the
            GridView's footer." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the editing, inserting, and deleting tutorials.

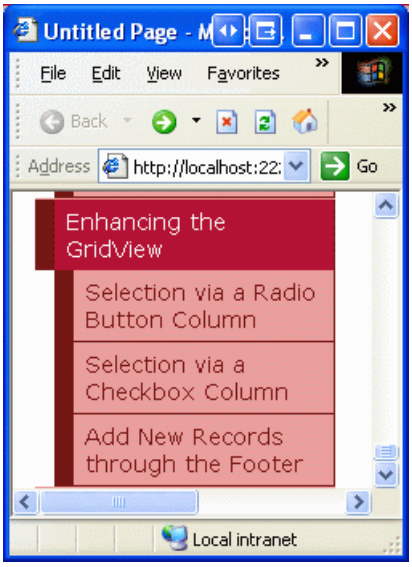


Figure 3: The Site Map Now Includes Entries for the Enhancing the GridView Tutorials

Step 2: Displaying the Suppliers in a GridView

For this tutorial let s build a GridView that lists the suppliers from the USA, with each GridView row providing a radio button. After selecting a supplier via the radio button, the user can view the supplier s products by clicking a button. While this task may sound trivial, there are a number of subtleties that make it particularly tricky. Before we delve into these subtleties, let s first get a GridView listing the suppliers.

Start by opening the `RadioButtonField.aspx` page in the `EnhancedGridView` folder by dragging a GridView from the Toolbox onto the Designer. Set the GridView s `ID` to `Suppliers` and, from its smart tag, choose to create a new data source. Specifically, create an ObjectDataSource named `SuppliersDataSource` that pulls its data from the `SuppliersBLL` object.

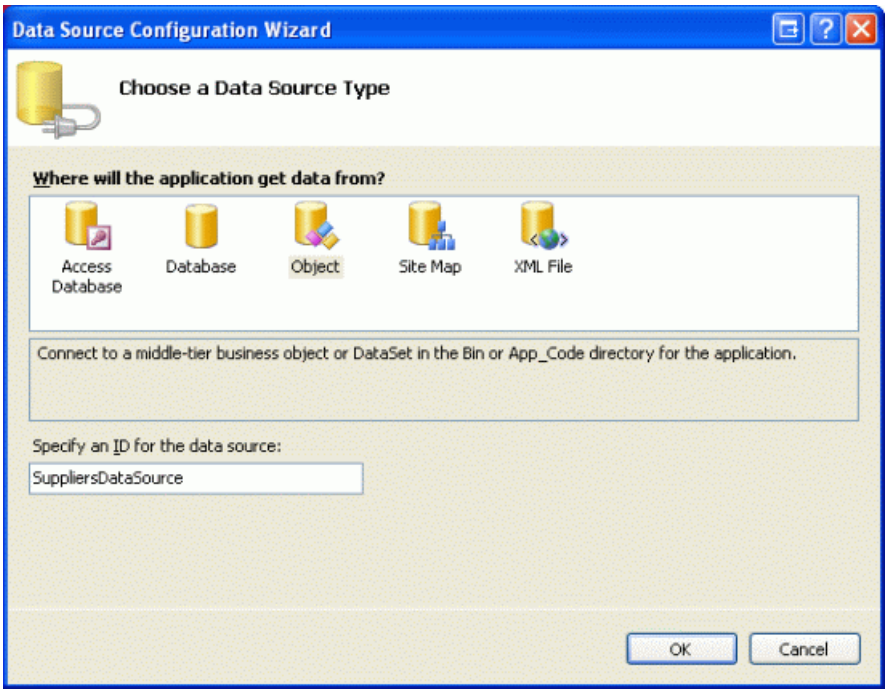


Figure 4: Create a New ObjectDataSource Named `SuppliersDataSource` (Click to view full-size image)

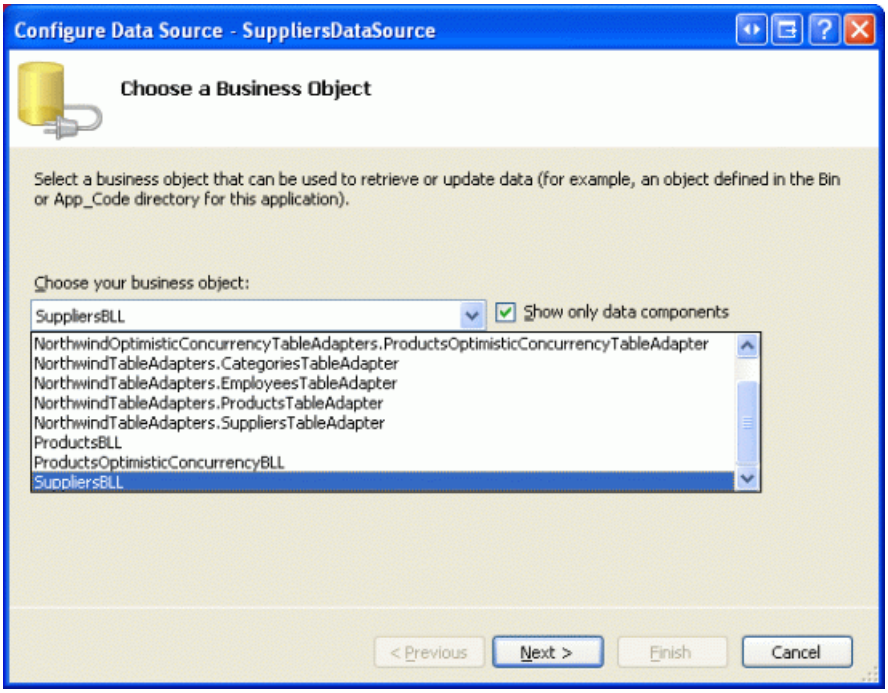


Figure 5: Configure the ObjectDataSource to Use the `SuppliersBLL` Class (Click to view full-size image)

Since we only want to list those suppliers in the USA, choose the `GetSuppliersByCountry(country)` method from the drop-down list in the SELECT tab.

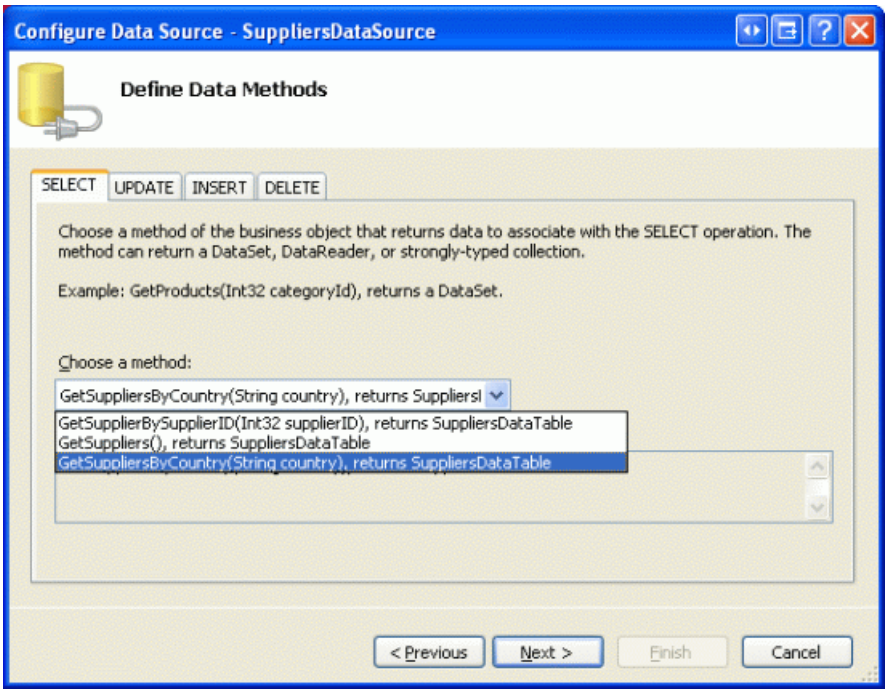


Figure 6: Configure the ObjectDataSource to Use the `SuppliersBLL` Class (Click to view full-size image)

From the UPDATE tab, select the (None) option and click Next.

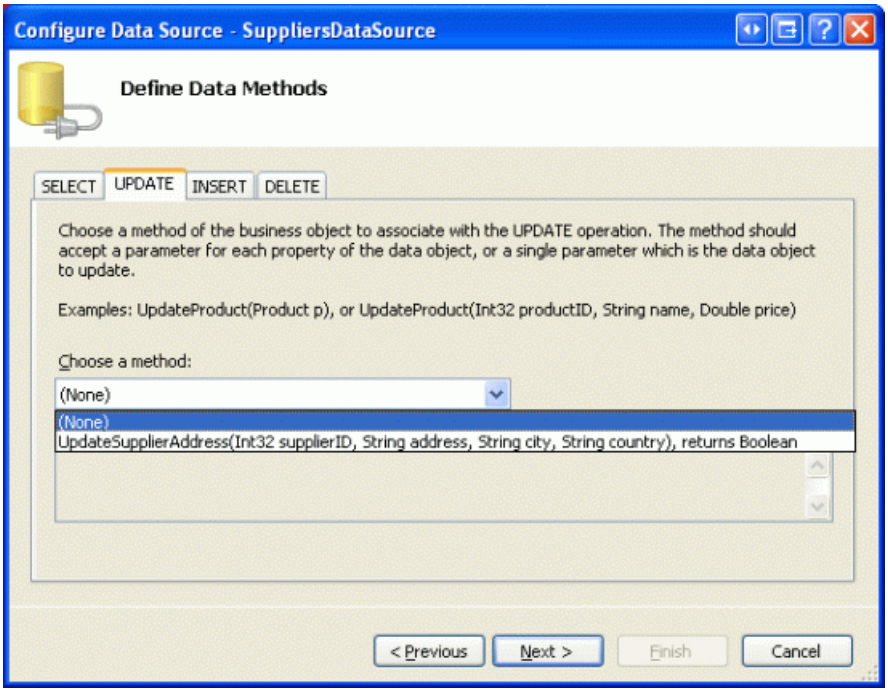


Figure 7: Configure the ObjectDataSource to Use the `SuppliersBLL` Class (Click to view full-size image)

Since the `GetSuppliersByCountry(country)` method accepts a parameter, the Configure Data Source wizard prompts us for the source of that parameter. To specify a hard coded value (USA, in this example), leave the Parameter source drop-down list set to None and enter the default value in the textbox. Click Finish to complete the wizard.

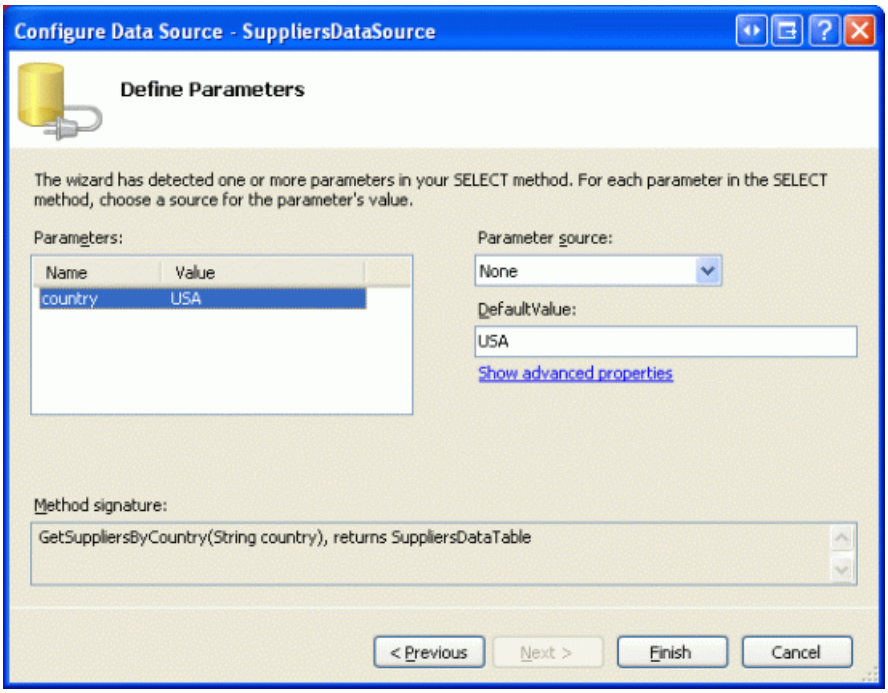


Figure 8: Use USA as the Default Value for the `country` Parameter (Click to view full-size image)

After completing the wizard, the GridView will include a BoundField for each of the supplier data fields. Remove all but the `CompanyName`, `City`, and `Country` BoundFields, and rename the `CompanyName` BoundFields `HeaderText` property to Supplier. After doing so, the GridView and ObjectDataSource declarative syntax should look similar to the following.

```
<asp:GridView ID="Suppliers" runat="server" AutoGenerateColumns="False"
    DataKeyNames="SupplierID" DataSourceID="SuppliersDataSource"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="CompanyName" HeaderText="Supplier"
            SortExpression="CompanyName" />
        <asp:BoundField DataField="City" HeaderText="City"
            SortExpression="City" />
        <asp:BoundField DataField="Country" HeaderText="Country"
            SortExpression="Country" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="SuppliersDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetSuppliersByCountry" TypeName="SuppliersBLL">
    <SelectParameters>
        <asp:Parameter DefaultValue="USA" Name="country" Type="String" />
    </SelectParameters>
</asp:ObjectDataSource>
```

For this tutorial, let s allow the user to view the selected supplier s products on the same page as the supplier list, or on a different page. To accommodate this, add two Button Web controls to the page. I ve set the `ID` s of these two Buttons to `ListProducts` and `SendToProducts`, with the idea that when `ListProducts` is clicked a postback will occur and the selected supplier s products will be listed on the same page, but when `SendToProducts` is clicked, the user will be whisked to a another page that lists the products.

Figure 9 shows the `Suppliers` GridView and the two Button Web controls when viewed through a browser.

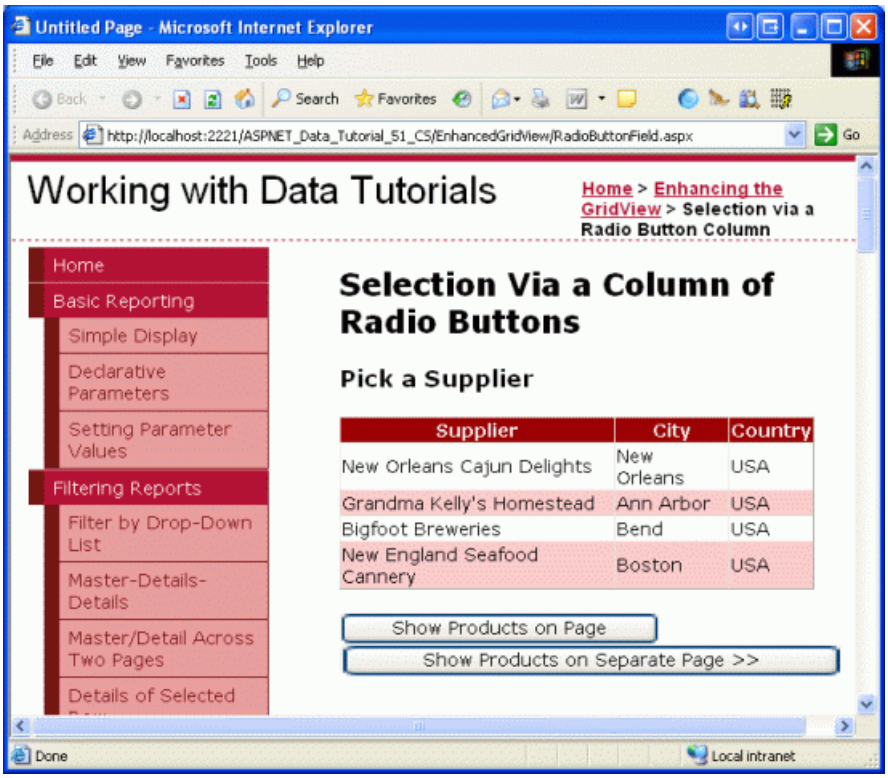


Figure 9: Those Suppliers from the USA Have Their Name, City, and Country Information Listed (Click to view full-size image)

Step 3: Adding a Column of Radio Buttons

At this point the `Suppliers` GridView has three BoundFields displaying the company name, city, and country of each supplier in the USA. It is still lacking a column of radio buttons, however. Unfortunately, the GridView doesn t include a built-in RadioButtonField, otherwise we could just add that to the grid and be done. Instead, we can add a TemplateField and configure its `ItemTemplate` to render a radio button, resulting in a radio button for each GridView row.

Initially, we might assume that the desired user interface can be implemented by adding a RadioButton Web control to the `ItemTemplate` of a TemplateField. While this will indeed add a single radio button to each row of the GridView, the radio buttons cannot be grouped and therefore are not mutually exclusive. That is, an end user is able to select multiple radio buttons simultaneously from the GridView.

Even though using a TemplateField of RadioButton Web controls does not offer the functionality we need, let s implement this approach, as it s worthwhile to examine why the resulting radio buttons are not grouped. Start by adding a TemplateField to the Suppliers GridView, making it the leftmost field. Next, from the GridView s smart tag, click the Edit Templates link and drag a RadioButton Web control from the Toolbox into the TemplateField s `ItemTemplate` (see Figure 10). Set the RadioButton s `ID` property to `RowSelector` and the `GroupName` property to `SuppliersGroup`.

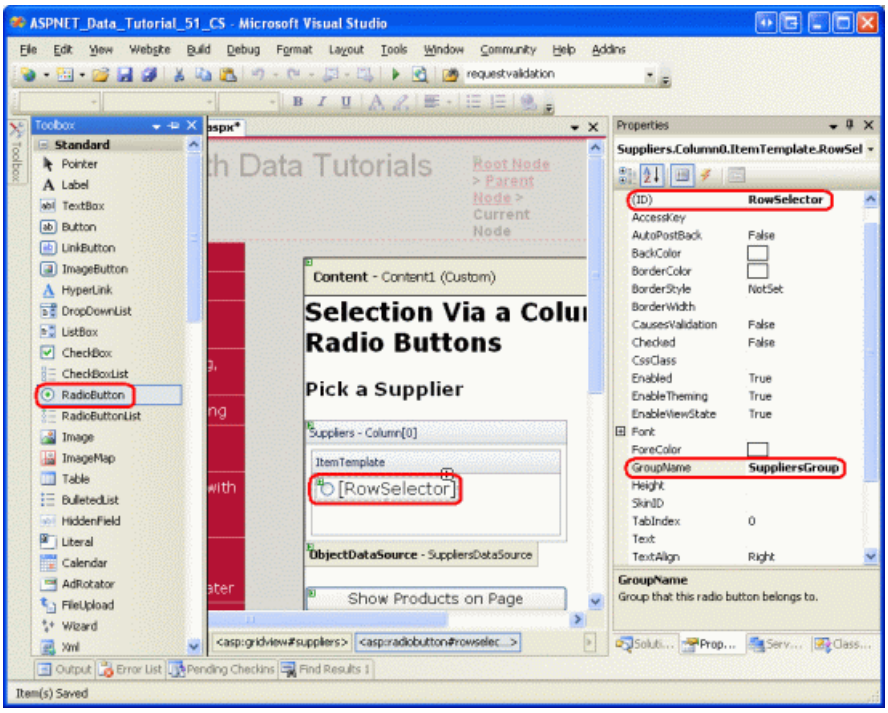


Figure 10: Add a RadioButton Web Control to the `ItemTemplate` (Click to view full-size image)

After making these additions through the Designer, your GridView s markup should look similar to the following:

```
<asp:GridView ID="Suppliers" runat="server" AutoGenerateColumns="False"
    DataKeyNames="SupplierID" DataSourceID="SuppliersDataSource"
    EnableViewState="False">
    <Columns>
        <asp:TemplateField>
            <ItemTemplate>
                <asp:RadioButton ID="RowSelector" runat="server"
                    GroupName="SuppliersGroup" />
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="CompanyName" HeaderText="Supplier"
            SortExpression="CompanyName" />
        <asp:BoundField DataField="City" HeaderText="City"
            SortExpression="City" />
        <asp:BoundField DataField="Country" HeaderText="Country"
            SortExpression="Country" />
    </Columns>
</asp:GridView>
```

The RadioButton s `GroupName` property is what is used to group a series of radio buttons. All RadioButton controls with the same `GroupName` value are considered grouped; only one radio button can be selected from a group at a time. The `GroupName` property specifies the value for the rendered radio button s `name` attribute. The browser examines the radio buttons `name` attributes to determine the radio button groupings.

With the RadioButton Web control added to the `ItemTemplate`, visit this page through a browser and click on the radio buttons in the grid s rows. Notice how the radio buttons are not grouped, making it possible to select all of the rows, as Figure 11 shows.

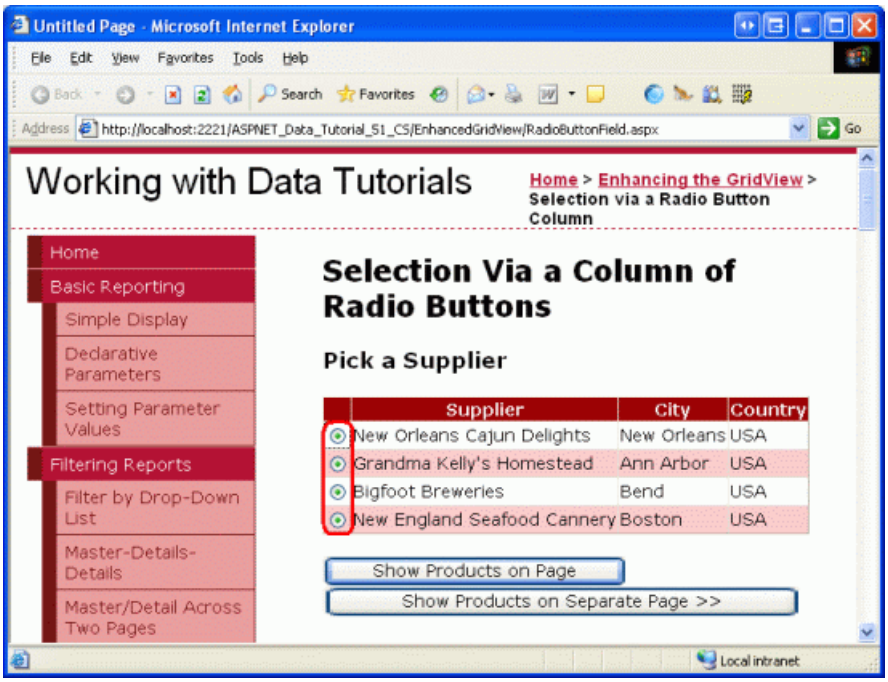


Figure 11: The GridView s Radio Buttons are Not Grouped (Click to view full-size image)

The reason the radio buttons are not grouped is because their rendered `name` attributes are different, despite having the same `GroupName` property setting. To see these differences, do a View/Source from the browser and examine the radio button markup:

```
<input id="ctl00_MainContent_Suppliers_ctl02_RowSelector"
    name="ctl00$MainContent$Suppliers$ctl02$SuppliersGroup"
    type="radio" value="RowSelector" />
<input id="ctl00_MainContent_Suppliers_ctl03_RowSelector"
    name="ctl00$MainContent$Suppliers$ctl03$SuppliersGroup"
    type="radio" value="RowSelector" />
<input id="ctl00_MainContent_Suppliers_ctl04_RowSelector"
    name="ctl00$MainContent$Suppliers$ctl04$SuppliersGroup"
    type="radio" value="RowSelector" />
<input id="ctl00_MainContent_Suppliers_ctl05_RowSelector"
    name="ctl00$MainContent$Suppliers$ctl05$SuppliersGroup"
    type="radio" value="RowSelector" />
```

Notice how both the `name` and `id` attributes are not the exact values as specified in the Properties window, but are prepended with a number of other `ID` values. The additional `ID` values added to the front of the rendered `id` and `name` attributes are the `ID` s of the radio buttons parent controls the `GridViewRow` s `ID` s, the GridView s `ID`, the Content control s `ID`, and the Web Form s `ID`. These `ID` s are added so that each rendered Web control in the GridView has a unique `id` and `name` values.

Each rendered control needs a different `name` and `id` because this is how the browser uniquely identifies each control on the client-side and how it identifies to the web server what action or change has occurred on postback. For example, imagine that we wanted to run some server-side code whenever a RadioButton s checked state was changed. We could accomplish this by setting the RadioButton s `AutoPostBack` property to `true` and creating an event handler for the `CheckChanged` event. However, if the rendered `name` and `id` values for all of the radio buttons were the same, on postback we could not determine what specific RadioButton was clicked.

The short of it is that we cannot create a column of radio buttons in a GridView using the RadioButton Web control. Instead, we must use rather archaic techniques to ensure that the appropriate markup is injected into each GridView row.

NOTE

Like the RadioButton Web control, the radio button HTML control, when added to a template, will include the unique `name` attribute, making the radio buttons in the grid ungrouped. If you are not familiar with HTML controls, feel free to disregard this note, as HTML controls are rarely used, especially in ASP.NET 2.0. But if you are interested in learning more, see K. Scott Allen s blog entry Web Controls and HTML Controls.

Using a Literal Control to Inject Radio Button Markup

In order to correctly group all of the radio buttons within the GridView, we need to manually inject the radio buttons markup into the `ItemTemplate`. Each radio button needs the same `name` attribute, but should have a unique `id` attribute (in case we want to access a radio button via client-side script). After a user selects a radio button and posts back the page, the browser will send back the value of the selected radio button s `value` attribute. Therefore, each radio button will need a unique `value` attribute. Finally, on postback we need to make sure to add the `checked` attribute to the one radio button that is selected, otherwise after the user makes a selection and posts back, the radio buttons will return to their default state (all unselected).

There are two approaches that can be taken in order to inject low-level markup into a template. One is to do a mix of markup and calls to formatting methods defined in the code-behind class. This technique was first discussed in the Using TemplateFields in the GridView Control tutorial. In our case it might look something like:

```
<input type="radio" id='<%# GetUniqueRadioButtonID(...) %>'
    name='SuppliersGroup' value='<%# GetRadioButtonValue(...) %>' ... />
```

Here, `GetUniqueRadioButton` and `GetRadioButtonValue` would be methods defined in the code-behind class that returned the appropriate `id` and `value` attribute values for each radio button. This approach works well for assigning the `id` and `value` attributes, but falls short when needing to specify the `checked` attribute value because the databinding syntax is only executed when data is first bound to the GridView. Therefore, if the GridView has view state enabled, the formatting methods will only fire when the page is first loaded (or when the GridView is explicitly rebound to the data source), and therefore the function that sets the `checked` attribute won't be called on postback. It s a rather subtle problem and a bit beyond the scope of this article, so I'll leave it at this. I do, however, encourage you to try using the above approach and work it through to the point where you'll get stuck. While such an exercise won't get you any closer to a working version, it will help foster a deeper understanding of the GridView and the databinding lifecycle.

The other approach to injecting custom, low-level markup in a template and the approach that we'll be using for this tutorial is to add a Literal control to the template. Then, in the GridView s `RowCreated` or `RowDataBound` event handler, the Literal control can be programmatically accessed and its `Text` property set to the markup to emit.

Start by removing the RadioButton from the TemplateField s `ItemTemplate`, replacing it with a Literal control. Set the Literal control s `ID` to `RadioButtonMarkup`.

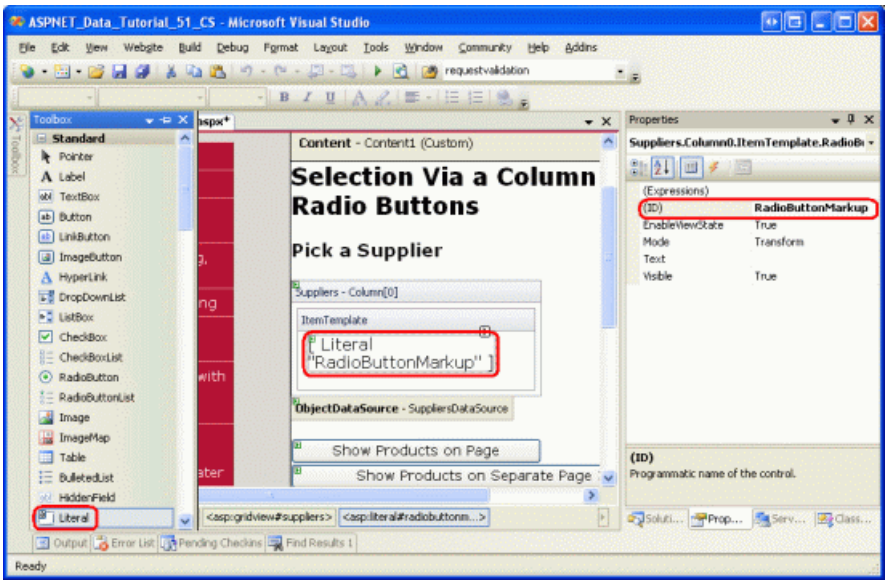


Figure 12: Add a Literal Control to the `ItemTemplate` (Click to view full-size image)

Next, create an event handler for the GridView s `RowCreated` event. The `RowCreated` event fires once for every row added, whether or not the data is being rebound to the GridView. That means that even on a postback when the data is reloaded from view state, the `RowCreated` event still fires and this is the reason we are using it instead of `RowDataBound` (which fires only when the data is explicitly bound to the data Web control).

In this event handler, we only want to proceed if we re dealing with a data row. For each data row we want to programmatically reference the `RadioButtonMarkup` Literal control and set its `Text` property to the markup to emit. As the following code shows, the markup emitted creates a radio button whose `name` attribute is set to `SuppliersGroup`, whose `id` attribute is set to `RowSelectorX`, where *X* is the index of the GridView row, and whose `value` attribute is set to the index of the GridView row.

```
protected void Suppliers_RowCreated(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        // Grab a reference to the Literal control
        Literal output = (Literal)e.Row.FindControl("RadioButtonMarkup");
        // Output the markup except for the "checked" attribute
        output.Text = string.Format(
            @"<input type="radio" name="SuppliersGroup" " +
            @"id="RowSelector{0}" value="{0}" />", e.Row.RowIndex);
    }
}
```

When a GridView row is selected and a postback occurs, we are interested in the `SupplierID` of the selected supplier. Therefore, one might think that the value of each radio button should be the actual `SupplierID` (rather than the index of the GridView row). While this may work in certain circumstances, it would be a security risk to blindly accept and process a `SupplierID`. Our GridView, for example, lists only those suppliers in the USA. However, if the `SupplierID` is passed directly from the radio button, what s to stop a mischievous user from manipulating the `SupplierID` value sent back on postback? By using the row index as the `value`, and then getting the `SupplierID` on postback from the `DataKeys` collection, we can ensure that the user is only using one of the `SupplierID` values associated with one of the GridView rows.

After adding this event handler code, take a minute to test out the page in a browser. First, note that only one radio button in the grid can be selected at a time. However, when selecting a radio button and clicking one of the buttons, a postback occurs and the radio buttons all revert to their initial state (that is, on postback, the selected radio button is no longer selected). To fix this, we need to augment the `RowCreated` event handler so that it inspects the selected radio button index sent from the postback and adds the `checked="checked"` attribute to the

emitted markup of the row index matches.

When a postback occurs, the browser sends back the `name` and `value` of the selected radio button. The value can be programmatically retrieved using `Request.Form["name"]`. The `Request.Form` property provides a `NameValueCollection` representing the form variables. The form variables are the names and values of the form fields in the web page, and are sent back by the web browser whenever a postback ensues. Because the rendered `name` attribute of the radio buttons in the GridView is `SuppliersGroup`, when the web page is posted back the browser will send `SuppliersGroup=valueOfSelectedRadioButton` back to the web server (along with the other form fields). This information can then be accessed from the `Request.Form` property using: `Request.Form["SuppliersGroup"]`.

Since we'll need to determine the selected radio button index not only in the `RowCreated` event handler, but in the `Click` event handlers for the Button Web controls, let s add a `SuppliersSelectedIndex` property to the code-behind class that returns `-1` if no radio button was selected and the selected index if one of the radio buttons is selected.

```
private int SuppliersSelectedIndex
{
    get
    {
        if (string.IsNullOrEmpty(Request.Form["SuppliersGroup"]))
            return -1;
        else
            return Convert.ToInt32(Request.Form["SuppliersGroup"]);
    }
}
```

With this property added, we know to add the `checked="checked"` markup in the `RowCreated` event handler when `SuppliersSelectedIndex` equals `e.Row.RowIndex`. Update the event handler to include this logic:

```
protected void Suppliers_RowCreated(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        // Grab a reference to the Literal control
        Literal output = (Literal)e.Row.FindControl("RadioButtonMarkup");
        // Output the markup except for the "checked" attribute
        output.Text = string.Format(
            @"<input type="radio" name="SuppliersGroup" " +
            @"id="RowSelector{0}" value="{0}"", e.Row.RowIndex);
        // See if we need to add the "checked" attribute
        if (SuppliersSelectedIndex == e.Row.RowIndex)
            output.Text += @" checked="checked"";
        // Add the closing tag
        output.Text += " />";
    }
}
```

With this change, the selected radio button remains selected after a postback. Now that we have the ability to specify what radio button is selected, we could change the behavior so that when the page was first visited, the first GridView row s radio button was selected (rather than having no radio buttons selected by default, which is the current behavior). To have the first radio button selected by default, simply change the `if (SuppliersSelectedIndex == e.Row.RowIndex)` statement to the following: `if (SuppliersSelectedIndex == e.Row.RowIndex || (!Page.IsPostBack && e.Row.RowIndex == 0))`.

At this point we have added a column of grouped radio buttons to the GridView that allows for a single GridView row to be selected and remembered across postbacks. Our next steps are to display the products provided by the selected supplier. In Step 4 we'll see how to redirect the user to another page, sending along the selected

`SupplierID`. In Step 5, we'll see how to display the selected supplier s products in a GridView on the same page.

> **NOTE**
> Rather than using a TemplateField (the focus of this lengthy Step 3), we could create a custom `DataControlField` class that renders the appropriate user interface and functionality. The `DataControlField` class is the base class from which the BoundField, CheckBoxField, TemplateField, and other built-in GridView and DetailsView fields derive. Creating a custom `DataControlField` class would mean that the column of radio buttons could be added just using declarative syntax, and would also make replicating the functionality on other web pages and other web applications significantly easier.

If you ve ever created custom, compiled controls in ASP.NET, however, you know that doing so requires a fair amount of legwork and carries with it a host of subtleties and edge cases that must be carefully handled. Therefore, we will forgo implementing a column of radio buttons as a custom `DataControlField` class for now and stick with the TemplateField option. Perhaps we'll have the chance to explore creating, using, and deploying custom `DataControlField` classes in a future tutorial!

Step 4: Displaying the Selected Supplier s Products in a Separate Page

After the user has selected a GridView row, we need to show the selected supplier s products. In some circumstances, we may want to display these products in a separate page, in others we might prefer to do it in the same page. Let s first examine how to display the products in a separate page; in Step 5 we'll look at adding a GridView to `RadioButtonField.aspx` to display the selected supplier s products.

Currently there are two Button Web controls on the page `ListProducts` and `SendToProducts`. When the `SendToProducts` Button is clicked, we want to send the user to `~/Filtering/ProductsForSupplierDetails.aspx`. This page was created in the Master/Detail Filtering Across Two Pages tutorial and displays the products for the supplier whose `SupplierID` is passed through the querystring field named `SupplierID`.

To provide this functionality, create an event handler for the `SendToProducts` Button s `Click` event. In Step 3 we added the `SuppliersSelectedIndex` property, which returns the index of the row whose radio button is selected. The corresponding `SupplierID` can be retrieved from the GridView s `DataKeys` collection and the user can then be sent to `~/Filtering/ProductsForSupplierDetails.aspx?SupplierID=SupplierID` using `Response.Redirect("url")`.

```
protected void SendToProducts_Click(object sender, EventArgs e)
{
    // Send the user to ~/Filtering/ProductsForSupplierDetails.aspx
    int supplierID =
        Convert.ToInt32(Suppliers.DataKeys[SuppliersSelectedIndex].Value);
    Response.Redirect(
        "~/Filtering/ProductsForSupplierDetails.aspx?SupplierID="
        + supplierID);
    }
}
```

This code works wonderfully as long as one of the radio buttons is selected from the GridView. If, initially, the GridView does not have any radio buttons selected, and the user clicks the `SendToProducts` button, `SuppliersSelectedIndex` will be `-1`, which will cause an exception to be thrown since `-1` is out of the index range of the `DataKeys` collection. This is not a concern, however, if you decided to update the `RowCreated` event handler as discussed in Step 3 so as to have the first radio button in the GridView initially selected.

To accommodate a `SuppliersSelectedIndex` value of `-1`, add a Label Web control to the page above the GridView. Set its `ID` property to `ChooseSupplierMsg`, its `CssClass` property to `Warning`, its `EnableViewState` and `Visible` properties to `false`, and its `Text` property to Please choose a supplier from the grid. The CSS class `Warning` displays text in a red, italic, bold, large font and is defined in `Styles.css`. By setting the `EnableViewState` and

`Visible` properties to `false`, the Label is not rendered except for only those postbacks where the control s `Visible` property is programmatically set to `true`.

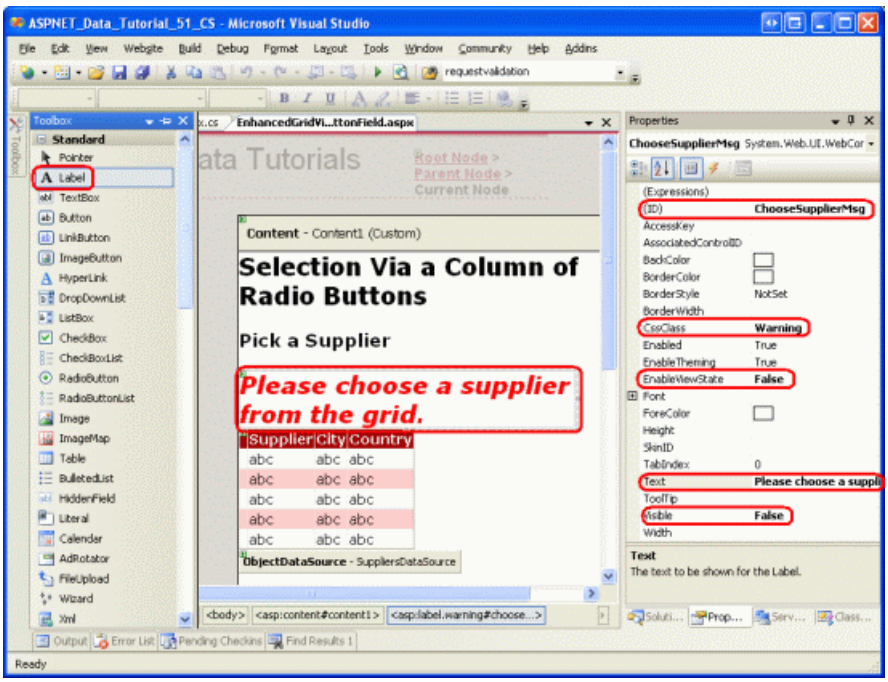


Figure 13: Add a Label Web Control Above the GridView (Click to view full-size image)

Next, augment the `Click` event handler to display the `ChooseSupplierMsg` Label if `SuppliersSelectedIndex` is less than zero, and redirect the user to `~/Filtering/ProductsForSupplierDetails.aspx?SupplierID=SupplierID` otherwise.

```
protected void SendToProducts_Click(object sender, EventArgs e)
{
    // make sure one of the radio buttons has been selected
    if (SuppliersSelectedIndex < 0)
        ChooseSupplierMsg.Visible = true;
    else
    {
        // Send the user to ~/Filtering/ProductsForSupplierDetails.aspx
        int supplierID =
            Convert.ToInt32(Suppliers.DataKeys[SuppliersSelectedIndex].Value);
        Response.Redirect(
            "~/Filtering/ProductsForSupplierDetails.aspx?SupplierID="
            + supplierID);
    }
}
```

Visit the page in a browser and click the `SendToProducts` button before selecting a supplier from the GridView. As Figure 14 shows, this displays the `ChooseSupplierMsg` label. Next, select a supplier and click the `SendToProducts` button. This will whisk you to a page that lists the products supplied by the selected supplier. Figure 15 shows the `ProductsForSupplierDetails.aspx` page when the Bigfoot Breweries supplier was selected.

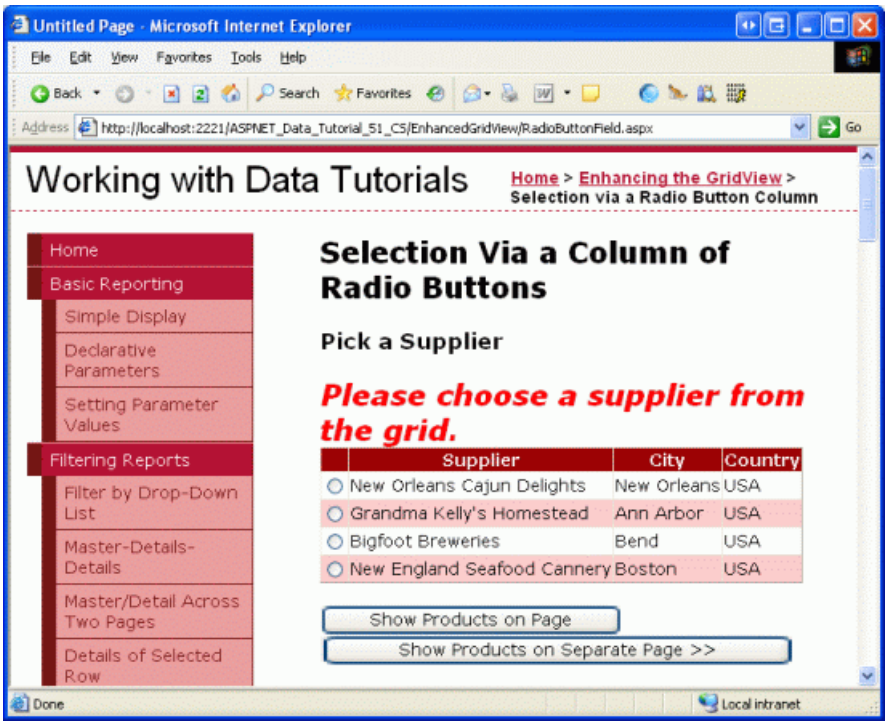


Figure 14: The `ChooseSupplierMsg` Label is Displayed if No Supplier is Selected (Click to view full-size image)

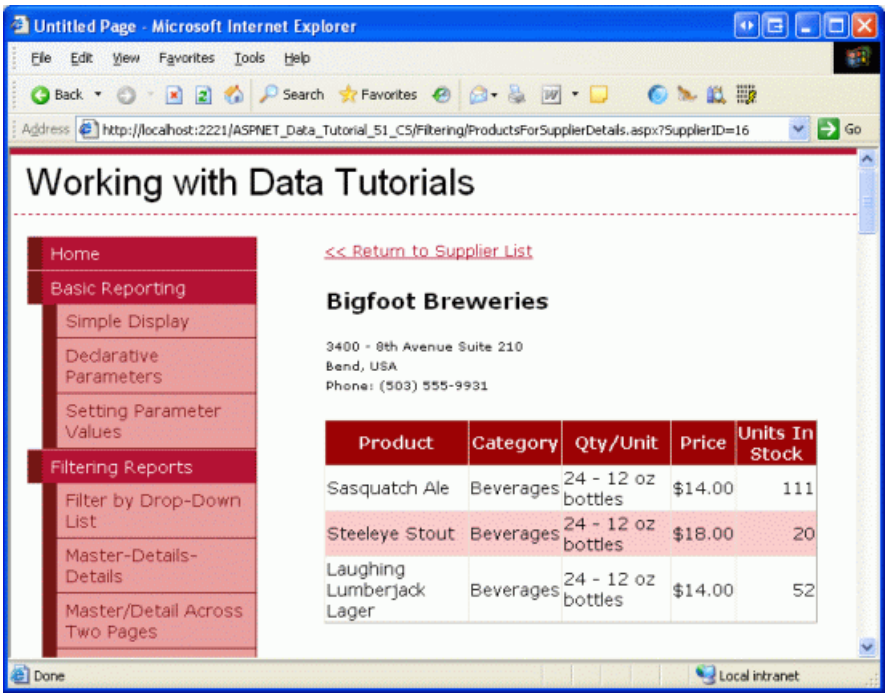


Figure 15: The Selected Supplier s Products are Displayed in `ProductsForSupplierDetails.aspx` (Click to view full-size image)

Step 5: Displaying the Selected Supplier s Products on the Same Page

In Step 4 we saw how to send the user to another web page to display the selected supplier s products. Alternatively, the selected supplier s products can be displayed on the same page. To illustrate this, we'll add another GridView to `RadioButtonField.aspx` to display the selected supplier s products.

Since we only want this GridView of products to display once a supplier has been selected, add a Panel Web control beneath the `Suppliers` GridView, setting its `ID` to `ProductsBySupplierPanel` and its `Visible` property to `false`. Within the Panel, add the text Products for the Selected Supplier, followed by a GridView named `ProductsBySupplier`. From the GridView s smart tag, choose to bind it to a new ObjectDataSource named `ProductsBySupplierDataSource`.

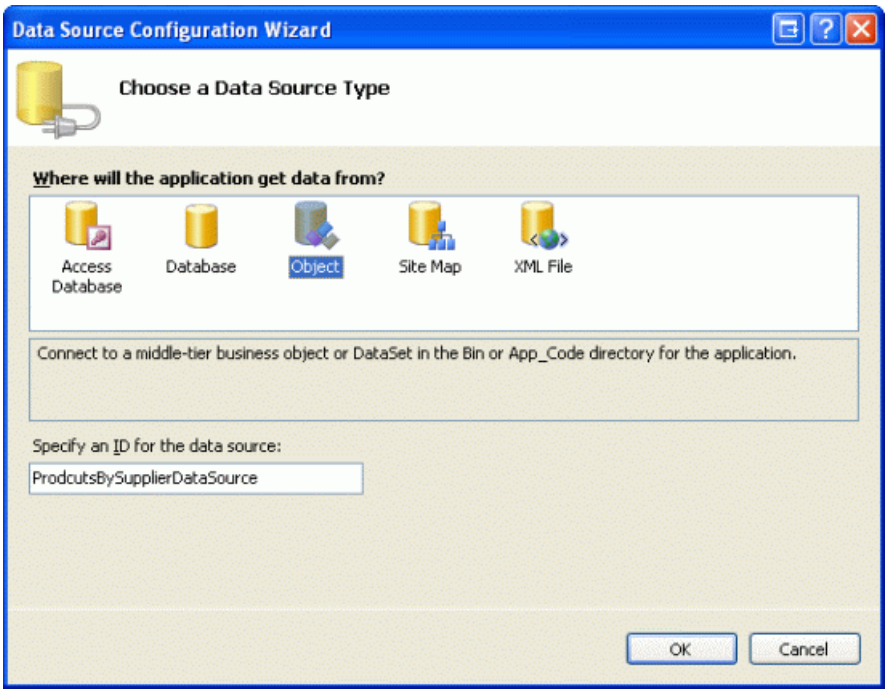


Figure 16: Bind the `ProductsBySupplier` GridView to a New ObjectDataSource (Click to view full-size image)

Next, configure the ObjectDataSource to use the `ProductsBLL` class. Since we only want to retrieve those products provided by the selected supplier, specify that the ObjectDataSource should invoke the `GetProductsBySupplierID(supplierID)` method to retrieve its data. Select (None) from the drop-down lists in the UPDATE, INSERT, and DELETE tabs.

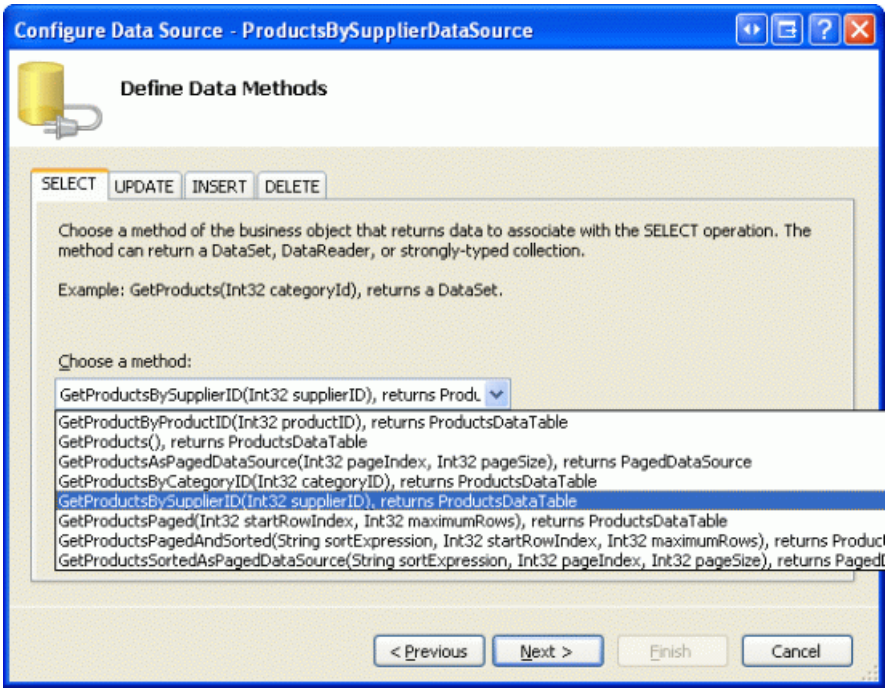


Figure 17: Configure the ObjectDataSource to Use the `GetProductsBySupplierID(supplierID)` Method (Click to view full-size image)

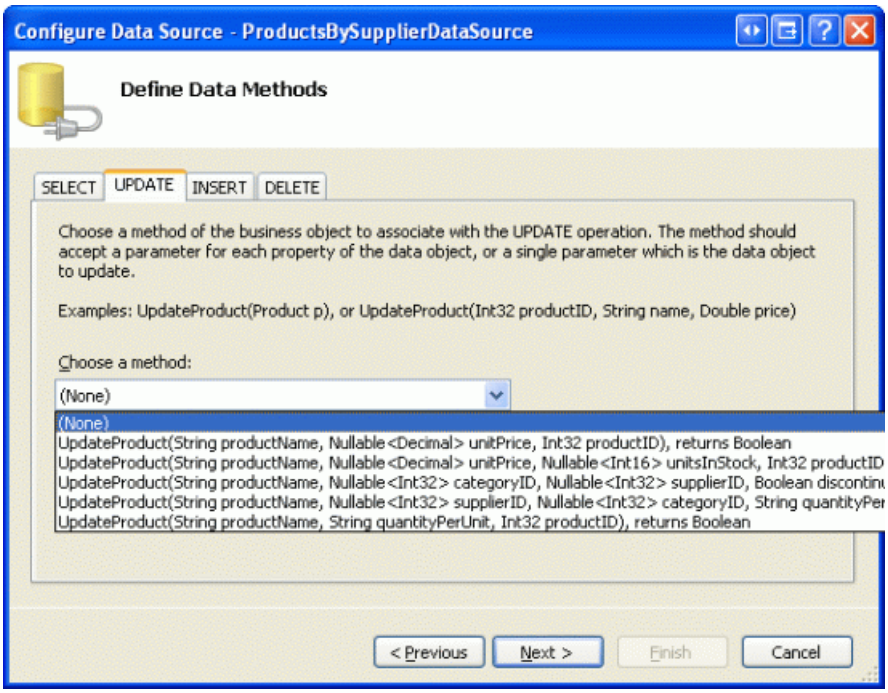


Figure 18: Set the Drop-Down Lists to (None) in the UPDATE, INSERT, and DELETE Tabs (Click to view full-size image)

After configuring the SELECT, UPDATE, INSERT, and DELETE tabs, click Next. Since the `GetProductsBySupplierID(supplierID)` method expects an input parameter, the Create Data Source wizard prompts us to specify the source for the parameter s value.

We have a couple of options here in specifying the source of the parameter s value. We could use the default Parameter object, and programmatically assign the value of the `SuppliersSelectedIndex` property to the Parameter s `DefaultValue` property in the ObjectDataSource s `Selecting` event handler. Refer back to the Programmatically Setting the ObjectDataSource's Parameter Values tutorial for a refresher on programmatically assigning values to the ObjectDataSource s parameters.

Alternatively, we can use a ControlParameter and refer to the `Suppliers` GridView s `SelectedValue` property (see Figure 19). The GridView s `SelectedValue` property returns the `DataKey` value corresponding to the `SelectedIndex` property. In order for this option to work, we need to programmatically set the GridView s `SelectedIndex` property to the selected row when the `ListProducts` button is clicked. As an added benefit, by setting the `SelectedIndex`, the selected record will take on the `SelectedRowStyle` defined in the `DataWebControls` Theme (a yellow background).

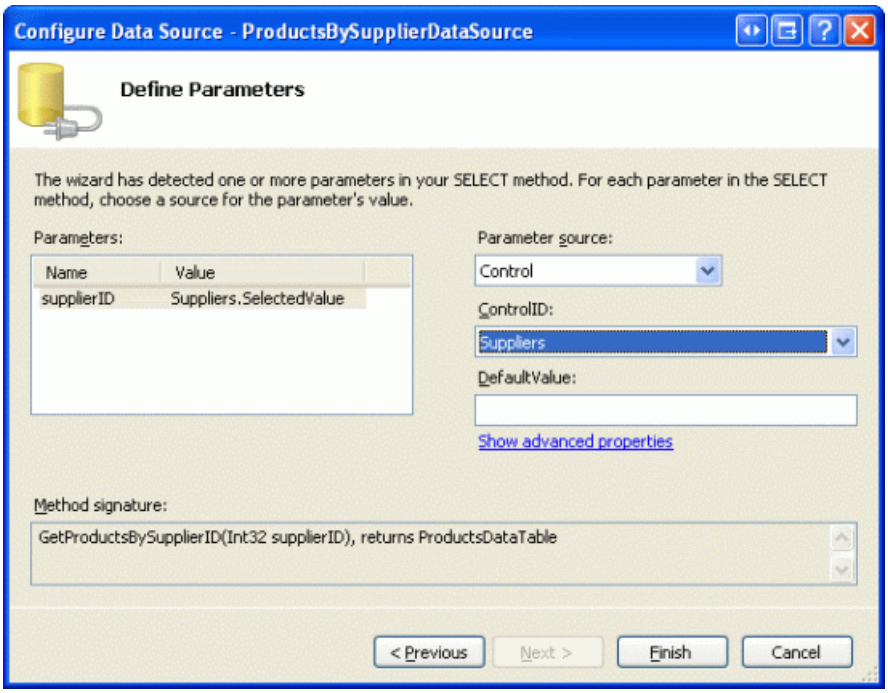


Figure 19: Use a ControlParameter to Specify the GridView s SelectedValue as the Parameter Source (Click to view full-size image)

Upon completing the wizard, Visual Studio will automatically add fields for the product s data fields. Remove all but the `ProductName`, `CategoryName`, and `UnitPrice` BoundFields, and change the `HeaderText` properties to Product, Category, and Price. Configure the `UnitPrice` BoundField so that its value is formatted as a currency. After making these changes, the Panel, GridView, and ObjectDataSource s declarative markup should look like the following:

```
<asp:Panel runat="server" ID="ProductsBySupplierPanel" Visible="False">
    <h3>
        Products for the Selected Supplier</h3>
    <p>
        <asp:GridView ID="ProductsBySupplier" runat="server"
            AutoGenerateColumns="False" DataKeyNames="ProductID"
            DataSourceID="ProductsBySupplierDataSource" EnableViewState="False">
            <Columns>
                <asp:BoundField DataField="ProductName" HeaderText="Product"
                    SortExpression="ProductName" />
                <asp:BoundField DataField="CategoryName" HeaderText="Category"
                    ReadOnly="True" SortExpression="CategoryName" />
                <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
                    HeaderText="Price" HtmlEncode="False"
                    SortExpression="UnitPrice" />
            </Columns>
        </asp:GridView>
        <asp:ObjectDataSource ID="ProductsBySupplierDataSource" runat="server"
            OldValuesParameterFormatString="original_{0}"
            SelectMethod="GetProductsBySupplierID" TypeName="ProductsBLL">
            <SelectParameters>
                <asp:ControlParameter ControlID="Suppliers" Name="supplierID"
                    PropertyName="SelectedValue" Type="Int32" />
            </SelectParameters>
        </asp:ObjectDataSource>
    </p>
</asp:Panel>
```

To complete this exercise, we need to set the GridView s `SelectedIndex` property to the `SelectedSuppliersIndex` and the `ProductsBySupplierPanel` Panel s `Visible` property to `true` when the `ListProducts` button is clicked. To accomplish this, create an event handler for the `ListProducts` Button Web control s `Click` event and add the

following code:

```
protected void ListProducts_Click(object sender, EventArgs e)
{
    // make sure one of the radio buttons has been selected
    if (SuppliersSelectedIndex < 0)
    {
        ChooseSupplierMsg.Visible = true;
        ProductsBySupplierPanel.Visible = false;
    }
    else
    {
        // Set the GridView's SelectedIndex
        Suppliers.SelectedIndex = SuppliersSelectedIndex;
        // Show the ProductsBySupplierPanel panel
        ProductsBySupplierPanel.Visible = true;
    }
}
```

If a supplier has not been selected from the GridView, the `ChooseSupplierMsg` Label is displayed and the `ProductsBySupplierPanel` Panel hidden. Otherwise, if a supplier has been selected, the `ProductsBySupplierPanel` is displayed and the GridView s `SelectedIndex` property is updated.

Figure 20 shows the results after the Bigfoot Breweries supplier has been selected and the Show Products on Page button has been clicked.

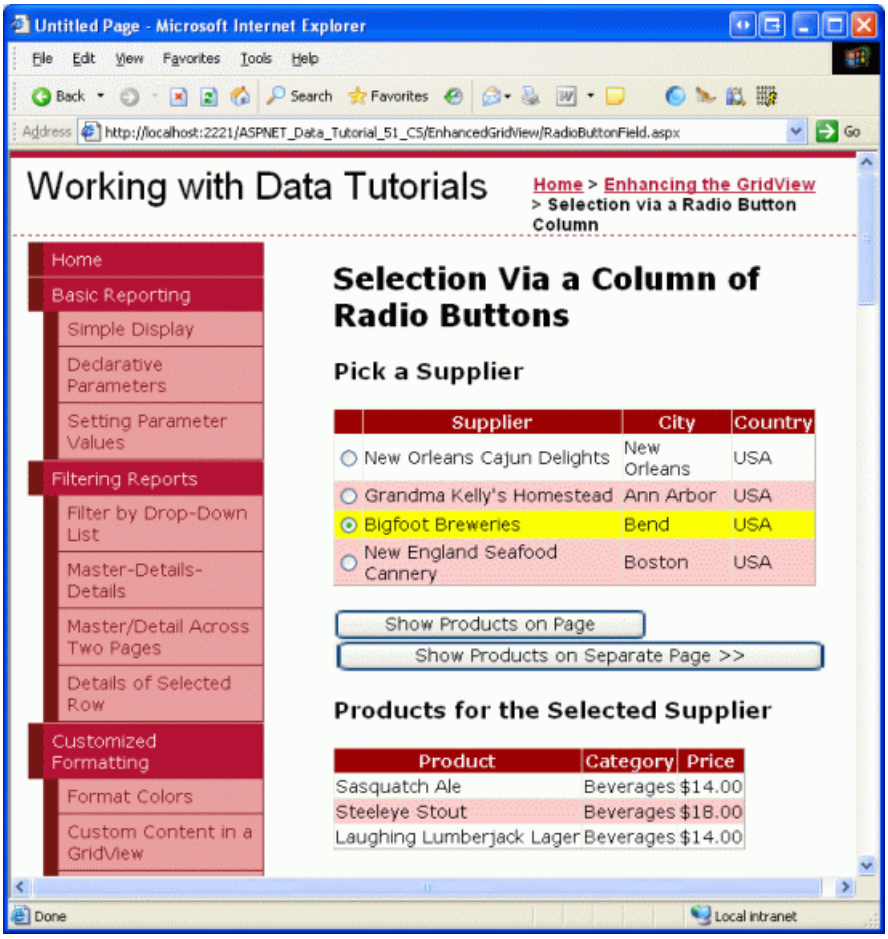


Figure 20: The Products Supplied by Bigfoot Breweries are Listed on the Same Page (Click to view full-size image)

Summary

As discussed in the Master/Detail Using a Selectable Master GridView with a Details DetailView tutorial, records can be selected from a GridView using a CommandField whose `ShowSelectButton` property is set to `true`. But the

CommandField displays its buttons either as regular push buttons, links, or images. An alternative row-selection user interface is to provide a radio button or checkbox in each GridView row. In this tutorial we examined how to add a column of radio buttons.

Unfortunately, adding a column of radio buttons isn t as straightforward or simple as one might expect. There is no built-in RadioButtonField that can be added at the click of a button, and using the RadioButton Web control within a TemplateField introduces its own set of problems. In the end, to provide such an interface we either have to create a custom `DataControlField` class or resort to injecting the appropriate HTML into a TemplateField during the `RowCreated` event.

Having explored how to add a column of radio buttons, let us turn our attention to adding a column of checkboxes. With a column of checkboxes, a user can select one or more GridView rows and then perform some operation on all of the selected rows (such as selecting a set of emails from a web-based email client, and then choosing to delete all selected emails). In the next tutorial we'll see how to add such a column.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was David Suru. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Adding a GridView Column of Checkboxes (C#)

2/20/2020 • 9 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

This tutorial looks at how to add a column of check boxes to a GridView control to provide the user with an intuitive way of selecting multiple rows of the GridView.

Introduction

In the preceding tutorial we examined how to add a column of radio buttons to the GridView for the purpose of selecting a particular record. A column of radio buttons is a suitable user interface when the user is limited to choosing at most one item from the grid. At times, however, we may want to allow the user to pick an arbitrary number of items from the grid. Web-based email clients, for example, typically display the list of messages with a column of checkboxes. The user can select an arbitrary number of messages and then perform some action, such as moving the emails to another folder or deleting them.

In this tutorial we will see how to add a column of checkboxes and how to determine what checkboxes were checked on postback. In particular, we'll build an example that closely mimics the web-based email client user interface. Our example will include a paged GridView listing the products in the `Products` database table with a checkbox in each row (see Figure 1). A Delete Selected Products button, when clicked, will delete those products selected.

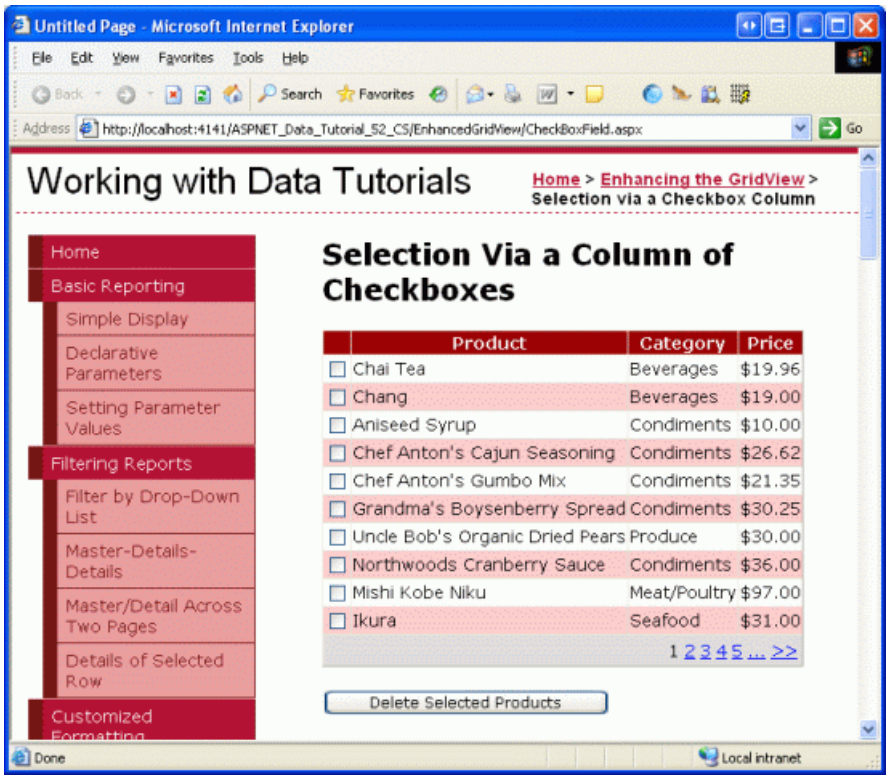


Figure 1: Each Product Row Includes a Checkbox (Click to view full-size image)

Step 1: Adding a Paged GridView that Lists Product Information

Before we worry about adding a column of checkboxes, let s first focus on listing the products in a GridView that

supports paging. Start by opening the `CheckBoxField.aspx` page in the `EnhancedGridView` folder and drag a GridView from the Toolbox onto the Designer, setting its `ID` to `Products`. Next, choose to bind the GridView to a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to use the `ProductsBLL` class, calling the `GetProducts()` method to return the data. Since this GridView will be read-only, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) .

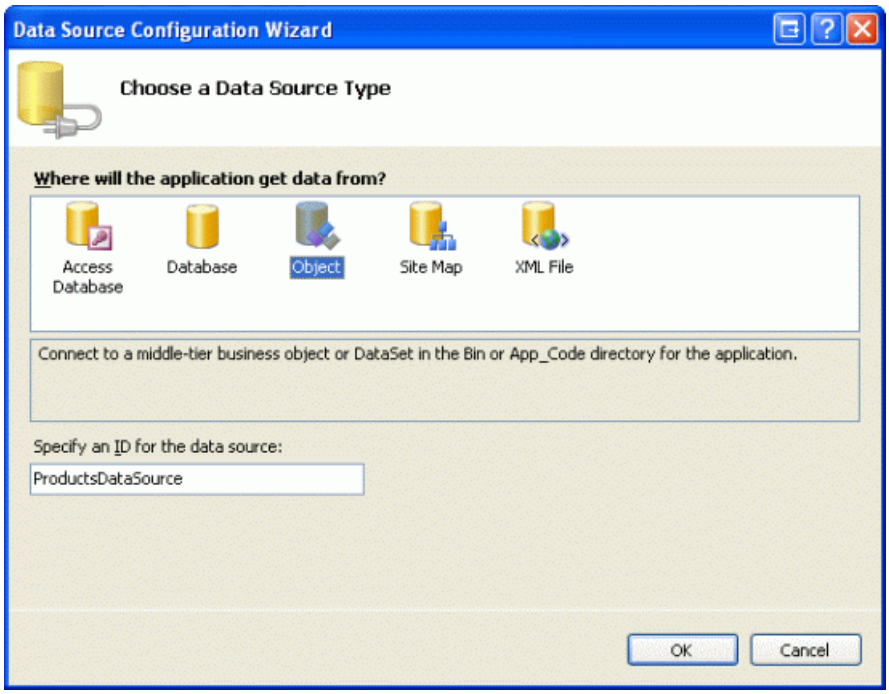


Figure 2: Create a New ObjectDataSource Named `ProductsDataSource` (Click to view full-size image)

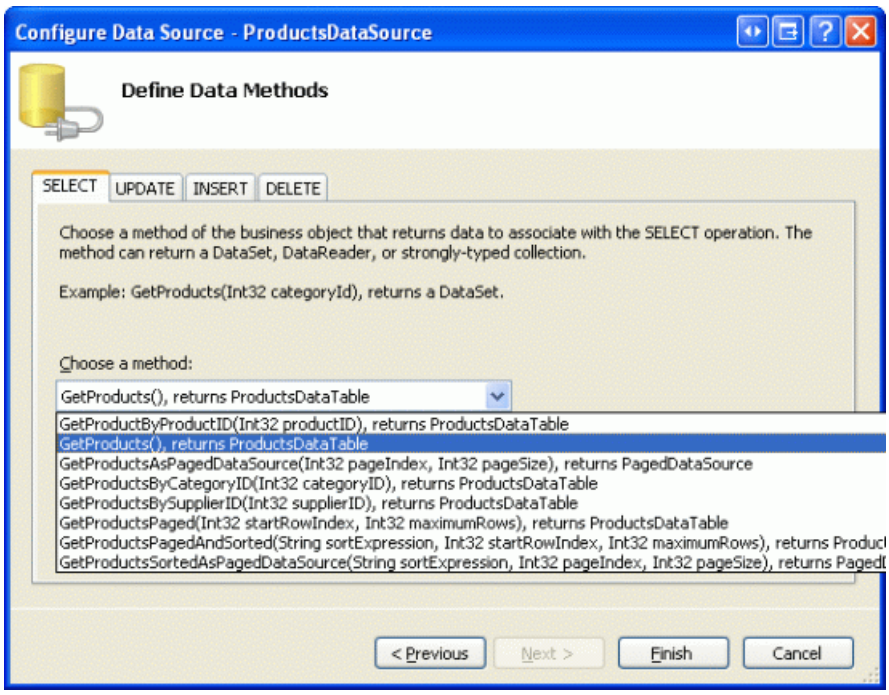


Figure 3: Configure the ObjectDataSource to Retrieve Data Using the `GetProducts()` Method (Click to view full-size image)

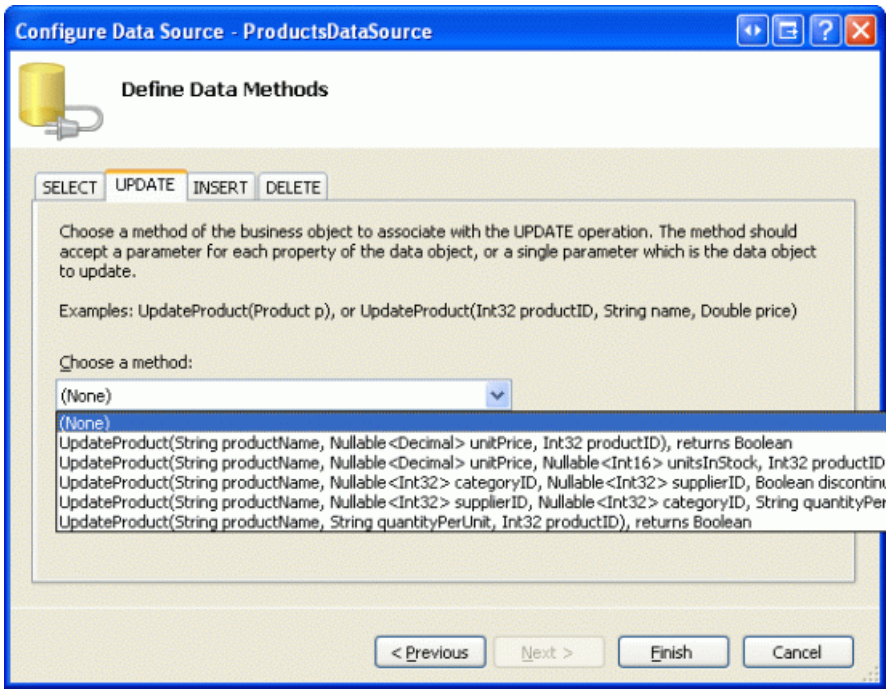


Figure 4: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE Tabs to (None) (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will automatically create BoundColumns and a CheckBoxColumn for the product-related data fields. Like we did in the previous tutorial, remove all but the `ProductName`, `CategoryName`, and `UnitPrice` BoundFields, and change the `HeaderText` properties to Product, Category, and Price. Configure the `UnitPrice` BoundField so that its value is formatted as a currency. Also configure the GridView to support paging by checking the Enable Paging checkbox from the smart tag.

Let s also add the user interface for deleting the selected products. Add a Button Web control beneath the GridView, setting its `ID` to `DeleteSelectedProducts` and its `Text` property to Delete Selected Products. Rather than actually deleting products from the database, for this example we'll just display a message stating the products that would have been deleted. To accommodate this, add a Label Web control beneath the Button. Set its ID to `DeleteResults`, clear out its `Text` property, and set its `Visible` and `EnableViewState` properties to `false`.

After making these changes, the GridView, ObjectDataSource, Button, and Label s declarative markup should similar to the following:

```
<p>
    <asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
        DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
        AllowPaging="True" EnableViewState="False">
        <Columns>
            <asp:BoundField DataField="ProductName" HeaderText="Product"
                SortExpression="ProductName" />
            <asp:BoundField DataField="CategoryName" HeaderText="Category"
                ReadOnly="True" SortExpression="CategoryName" />
            <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
                HeaderText="Price" HtmlEncode="False"
                SortExpression="UnitPrice" />
        </Columns>
    </asp:GridView>
    <asp:ObjectDataSource ID="ProductsDataSource" runat="server"
        OldValuesParameterFormatString="original_{0}"
        SelectMethod="GetProducts" TypeName="ProductsBLL">
    </asp:ObjectDataSource>
</p>
<p>
    <asp:Button ID="DeleteSelectedProducts" runat="server"
        Text="Delete Selected Products" />
</p>
<p>
    <asp:Label ID="DeleteResults" runat="server" EnableViewState="False"
        Visible="False"></asp:Label>
</p>
```

Take a moment to view the page in a browser (see Figure 5). At this point you should see the name, category, and price of the first ten products.

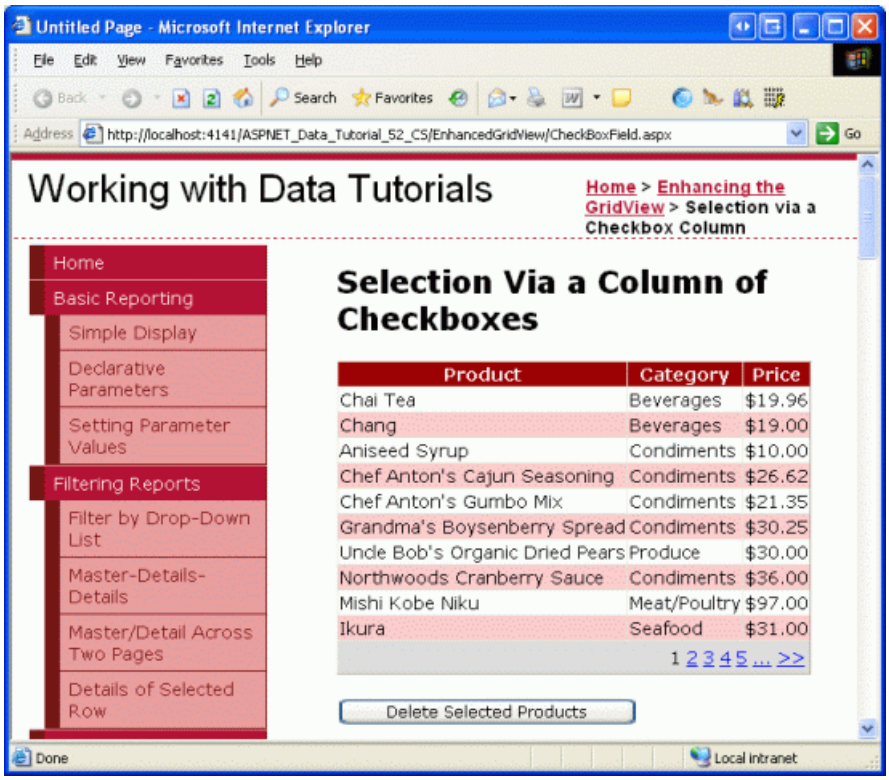


Figure 5: The Name, Category, and Price of the First Ten Products are Listed (Click to view full-size image)

Step 2: Adding a Column of Checkboxes

Since ASP.NET 2.0 includes a CheckBoxField, one might think that it could be used to add a column of checkboxes to a GridView. Unfortunately, that is not the case, as the CheckBoxField is designed to work with a Boolean data field. That is, in order to use the CheckBoxField we must specify the underlying data field whose value is consulted

to determine whether the rendered checkbox is checked. We cannot use the CheckBoxField to just include a column of unchecked checkboxes.

Instead, we must add a TemplateField and add a CheckBox Web control to its `ItemTemplate`. Go ahead and add a TemplateField to the `Products` GridView and make it the first (far-left) field. From the GridView s smart tag, click on the Edit Templates link and then drag a CheckBox Web control from the Toolbox into the `ItemTemplate`. Set this CheckBox s `ID` property to `ProductSelector`.

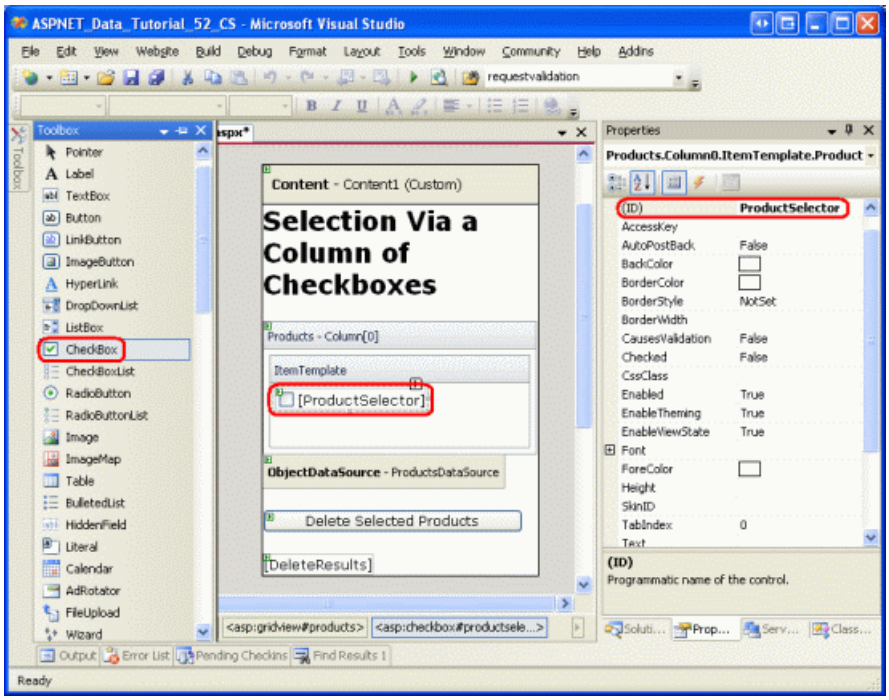


Figure 6: Add a CheckBox Web Control Named `ProductSelector` to the TemplateField s `ItemTemplate` (Click to view full-size image)

With the TemplateField and CheckBox Web control added, each row now includes a checkbox. Figure 7 shows this page, when viewed through a browser, after the TemplateField and CheckBox have been added.

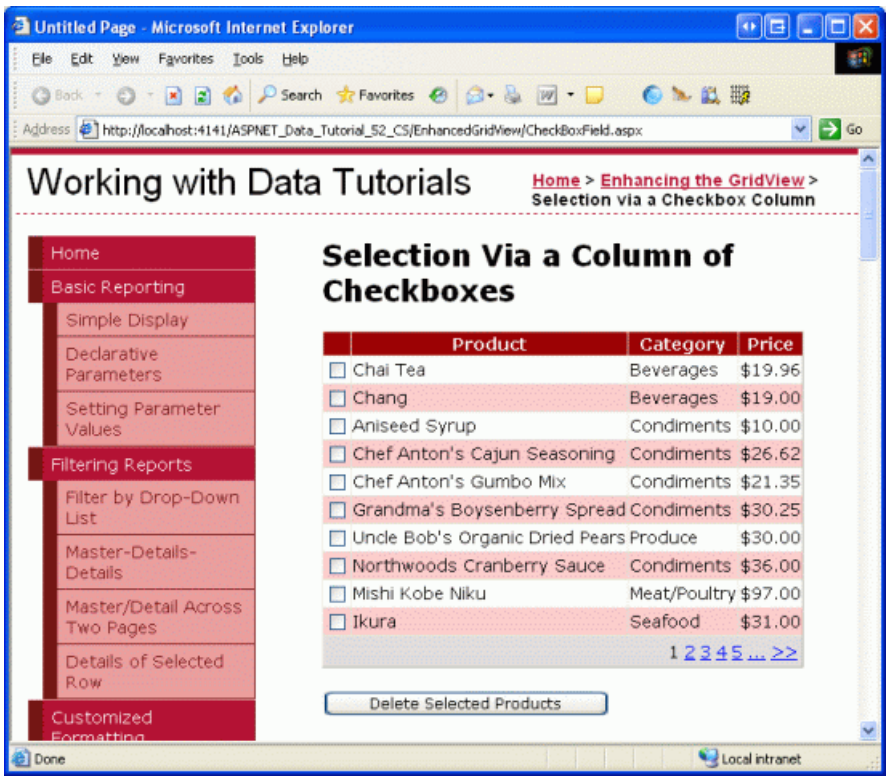


Figure 7: Each Product Row Now Includes a Checkbox (Click to view full-size image)

Step 3: Determining What Checkboxes Were Checked On Postback

At this point we have a column of checkboxes but no way to determine what checkboxes were checked on postback. When the Delete Selected Products button is clicked, though, we need to know what checkboxes were checked in order to delete those products.

The GridView s `Rows` property provides access to the data rows in the GridView. We can iterate through these rows, programmatically access the CheckBox control, and then consult its `Checked` property to determine whether the CheckBox has been selected.

Create an event handler for the `DeleteSelectedProducts` Button Web control s `Click` event and add the following code:

```
protected void DeleteSelectedProducts_Click(object sender, EventArgs e)
{
    bool atLeastOneRowDeleted = false;
    // Iterate through the Products.Rows property
    foreach (GridViewRow row in Products.Rows)
    {
        // Access the CheckBox
        CheckBox cb = (CheckBox)row.FindControl("ProductSelector");
        if (cb != null && cb.Checked)
        {
            // Delete row! (Well, not really...)
            atLeastOneRowDeleted = true;
            // First, get the ProductID for the selected row
            int productID =
                Convert.ToInt32(Products.DataKeys[row.RowIndex].Value);
            // "Delete" the row
            DeleteResults.Text += string.Format(
                "This would have deleted ProductID {0}<br />", productID);
        }
    }
    // Show the Label if at least one row was deleted...
    DeleteResults.Visible = atLeastOneRowDeleted;
}
```

The `Rows` property returns a collection of `GridViewRow` instances that makeup the GridView s data rows. The `foreach` loop here enumerates this collection. For each `GridViewRow` object, the row s CheckBox is programmatically accessed using `row.FindControl("controlID")`. If the CheckBox is checked, the row s corresponding `ProductID` value is retrieved from the `DataKeys` collection. In this exercise, we simply display an informative message in the `DeleteResults` Label, although in a working application we d instead make a call to the `ProductsBLL` class s `DeleteProduct(productID)` method.

With the addition of this event handler, clicking the Delete Selected Products button now displays the `ProductID` s of the selected products.

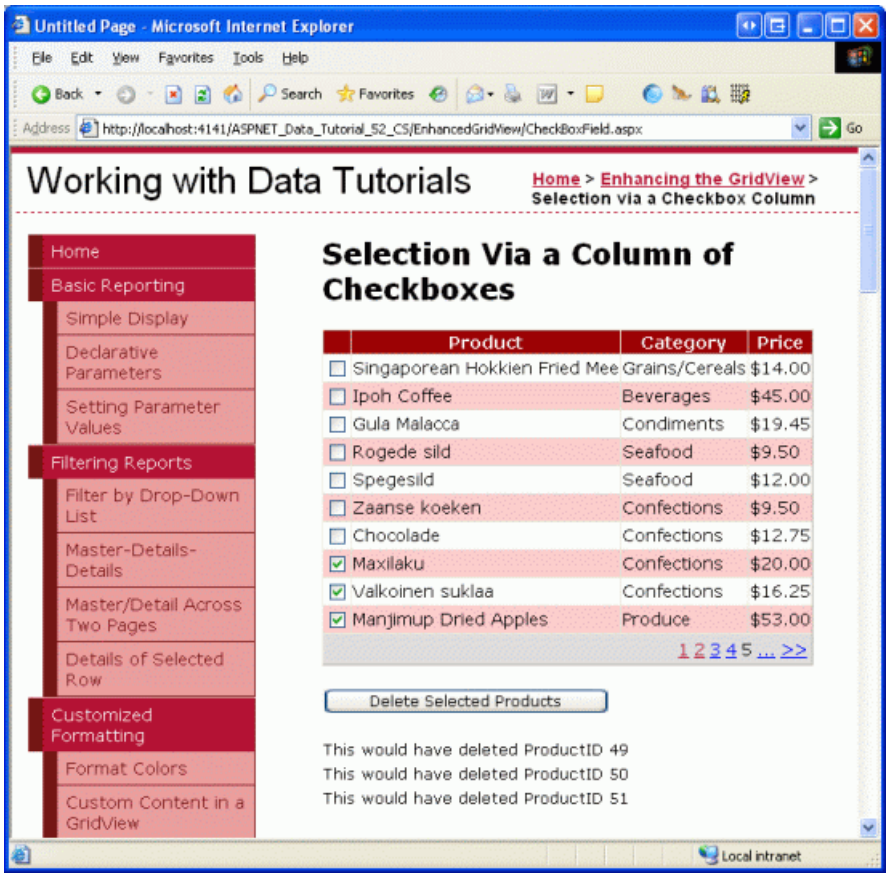


Figure 8: When the Delete Selected Products Button is Clicked the Selected Products `ProductID` s are Listed (Click to view full-size image)

Step 4: Adding Check All and Uncheck All Buttons

If a user wants to delete all products on the current page, they must check each of the ten checkboxes. We can help expedite this process by adding a Check All button that, when clicked, selects all of the checkboxes in the grid. An Uncheck All button would be equally helpful.

Add two Button Web controls to the page, placing them above the GridView. Set the first one s `ID` to `CheckAll` and its `Text` property to Check All ; set the second one s `ID` to `UncheckAll` and its `Text` property to Uncheck All .

```
<asp:Button ID="CheckAll" runat="server" Text="Check All" />

<asp:Button ID="UncheckAll" runat="server" Text="Uncheck All" />
```

Next, create a method in the code-behind class named `ToggleCheckState(checkState)` that, when invoked, enumerates the `Products` GridView s `Rows` collection and sets each CheckBox s `Checked` property to the value of the passed in *checkState* parameter.

```
private void ToggleCheckState(bool checkState)
{
    // Iterate through the Products.Rows property
    foreach (GridViewRow row in Products.Rows)
    {
        // Access the CheckBox
        CheckBox cb = (CheckBox)row.FindControl("ProductSelector");
        if (cb != null)
            cb.Checked = checkState;
    }
}
```

Next, create `Click` event handlers for the `CheckAll` and `UncheckAll` buttons. In `CheckAll` s event handler, simply call `ToggleCheckState(true)`; in `UncheckAll`, call `ToggleCheckState(false)`.

```
protected void CheckAll_Click(object sender, EventArgs e)
{
    ToggleCheckState(true);
}
protected void UncheckAll_Click(object sender, EventArgs e)
{
    ToggleCheckState(false);
}
```

With this code, clicking the Check All button causes a postback and checks all of the checkboxes in the GridView. Likewise, clicking Uncheck All unselects all checkboxes. Figure 9 shows the screen after the Check All button has been checked.

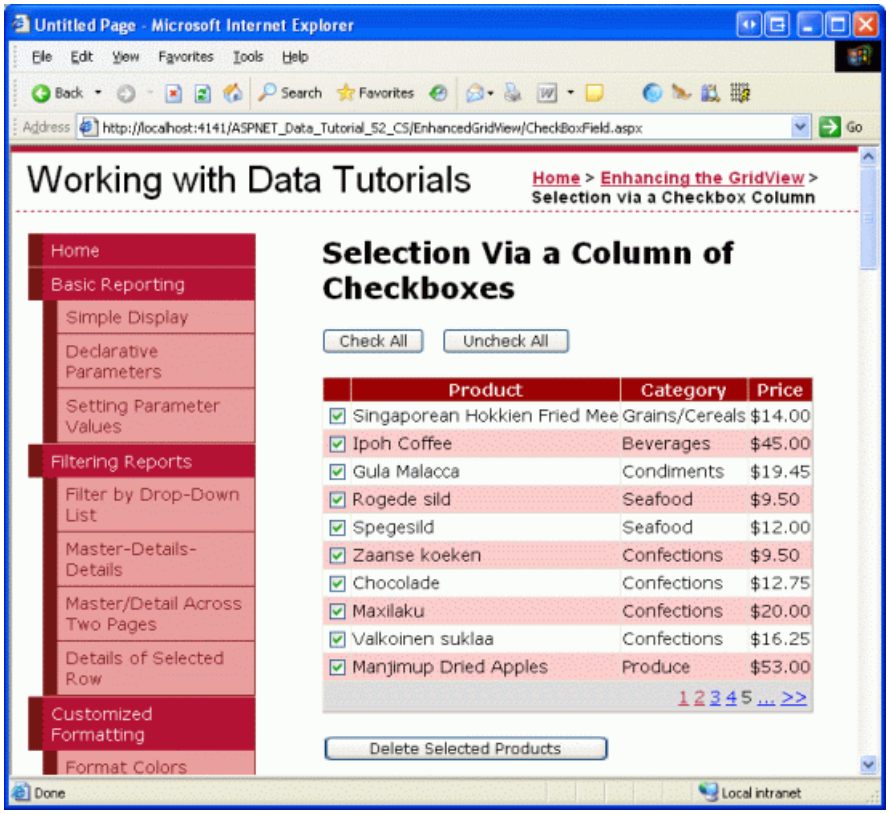


Figure 9: Clicking the Check All Button Selects All Checkboxes (Click to view full-size image)

NOTE

When displaying a column of checkboxes, one approach for selecting or unselecting all of the checkboxes is through a checkbox in the header row. Moreover, the current Check All / Uncheck All implementation requires a postback. The checkboxes can be checked or unchecked, however, entirely through client-side script, thereby providing a snappier user experience. To explore using a header row checkbox for Check All and Uncheck All in detail, along with a discussion on using client-side techniques, check out Checking All CheckBoxes in a GridView Using Client-Side Script and a Check All CheckBox.

Summary

In cases where you need to let users choose an arbitrary number of rows from a GridView before proceeding, adding a column of checkboxes is one option. As we saw in this tutorial, including a column of checkboxes in the GridView entails adding a TemplateField with a CheckBox Web control. By using a Web control (versus injecting markup directly into the template, as we did in the previous tutorial) ASP.NET automatically remembers what CheckBoxes were and were not checked across postback. We can also programmatically access the CheckBoxes in code to determine whether a given CheckBox is checked, or to change the checked state.

This tutorial and the last one looked at adding a row selector column to the GridView. In our next tutorial we'll examine how, with a bit of work, we can add inserting capabilities to the GridView.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Inserting a New Record from the GridView's Footer (C#)

2/20/2020 • 24 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> While the GridView control does not provide built-in support for inserting a new record of data, this tutorial shows how to augment the GridView to include an inserting interface.

Introduction

As discussed in the An Overview of Inserting, Updating, and Deleting Data tutorial, the GridView, DetailsView, and FormView Web controls each include built-in data modification capabilities. When used with declarative data source controls, these three Web controls can be quickly and easily configured to modify data - and in scenarios without needing to write a single line of code. Unfortunately, only the DetailsView and FormView controls provide built-in inserting, editing, and deleting capabilities. The GridView only offers editing and deleting support. However, with a little elbow grease, we can augment the GridView to include an inserting interface.

In adding inserting capabilities to the GridView, we are responsible for deciding on how new records will be added, creating the inserting interface, and writing the code to insert the new record. In this tutorial we will look at adding the inserting interface to the GridView s footer row (see Figure 1). The footer cell for each column includes the appropriate data collection user interface element (a TextBox for the product s name, a DropDownList for the supplier, and so on). We also need a column for an Add button that, when clicked, will cause a postback and insert a new record into the `Products` table using the values supplied in the footer row.

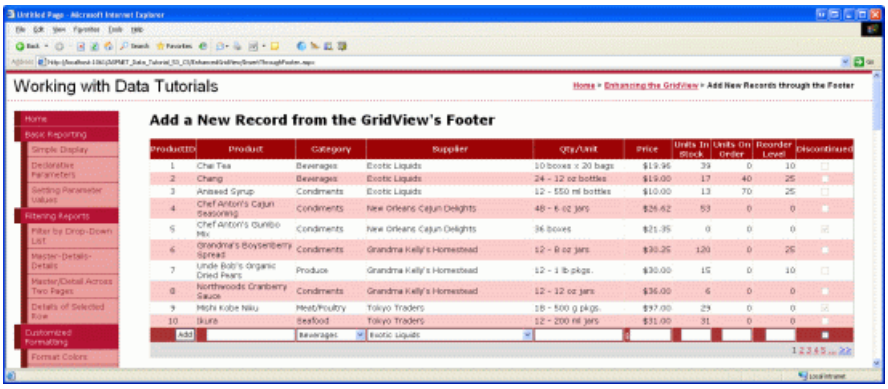


Figure 1: The Footer Row Provides an Interface for Adding New Products (Click to view full-size image)

Step 1: Displaying Product Information in a GridView

Before we concern ourselves with creating the inserting interface in the GridView s footer, let s first focus on adding a GridView to the page that lists the products in the database. Start by opening the `InsertThroughFooter.aspx` page in the `EnhancedGridView` folder and drag a GridView from the Toolbox onto the Designer, setting the GridView s `ID` property to `Products`. Next, use the GridView s smart tag to bind it to a new ObjectDataSource named `ProductsDataSource`.

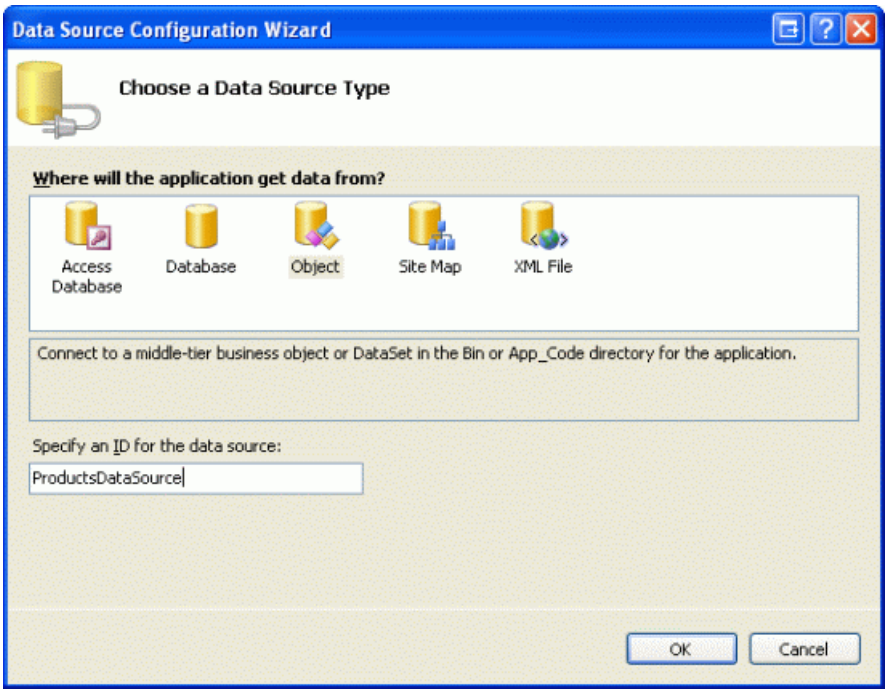


Figure 2: Create a New ObjectDataSource Named `ProductsDataSource` (Click to view full-size image)

Configure the ObjectDataSource to use the `ProductsBLL` class s `GetProducts()` method to retrieve product information. For this tutorial, let s focus strictly on adding inserting capabilities and not worry about editing and deleting. Therefore, make sure that the drop-down list in the INSERT tab is set to `AddProduct()` and that the drop-down lists in the UPDATE and DELETE tabs are set to (None) .

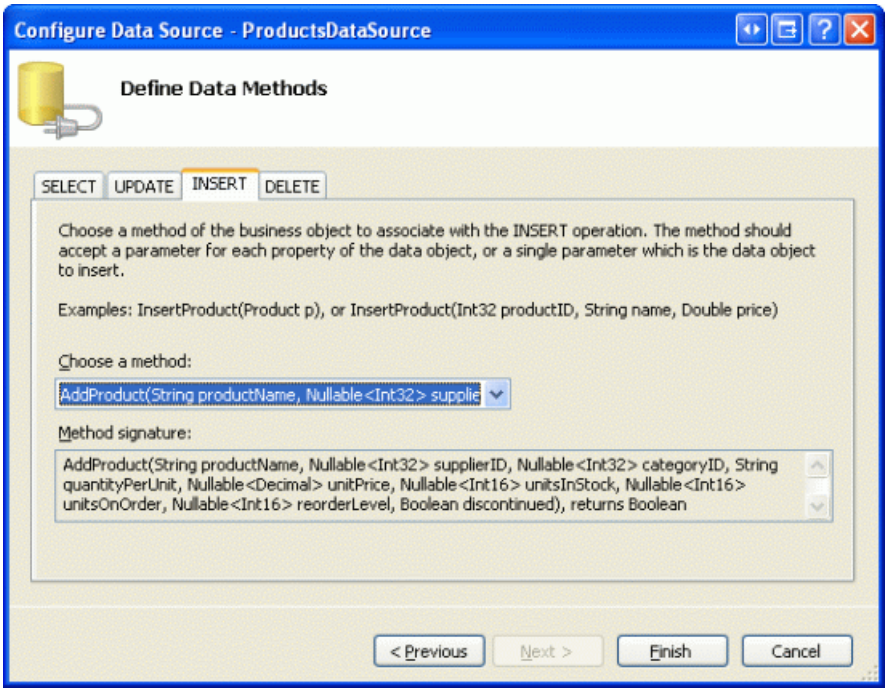


Figure 3: Map the `AddProduct` Method to the ObjectDataSource s `Insert()` Method (Click to view full-size image)

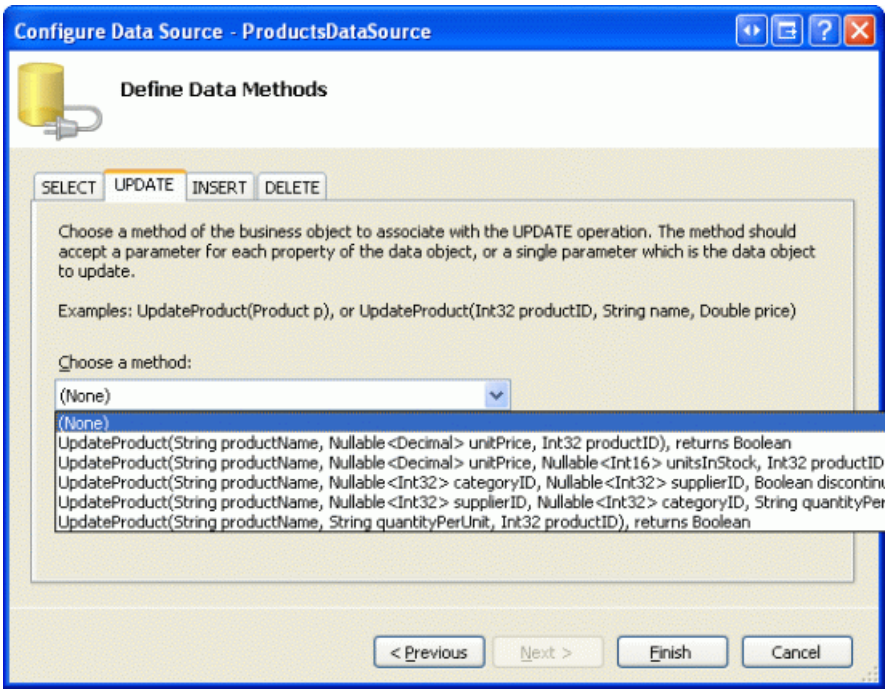


Figure 4: Set the UPDATE and DELETE Tabs Drop-Down Lists to (None) (Click to view full-size image)

After completing the ObjectDataSource s Configure Data Source wizard, Visual Studio will automatically add fields to the GridView for each of the corresponding data fields. For now, leave all of the fields added by Visual Studio. Later in this tutorial we'll come back and remove some of the fields whose values don t need to be specified when adding a new record.

Since there are close to 80 products in the database, a user will have to scroll all the way down to the bottom of the web page in order to add a new record. Therefore, let s enable paging to make the inserting interface more visible and accessible. To turn on paging, simply check the Enable Paging checkbox from the GridView s smart tag.

At this point, the GridView and ObjectDataSource s declarative markup should look similar to the following:

```
<asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
    AllowPaging="True" EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="ProductID"
            InsertVisible="False" ReadOnly="True"
            SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName" HeaderText="ProductName"
            SortExpression="ProductName" />
        <asp:BoundField DataField="SupplierID" HeaderText="SupplierID"
            SortExpression="SupplierID" />
        <asp:BoundField DataField="CategoryID" HeaderText="CategoryID"
            SortExpression="CategoryID" />
        <asp:BoundField DataField="QuantityPerUnit" HeaderText="QuantityPerUnit"
            SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
            SortExpression="UnitPrice" />
        <asp:BoundField DataField="UnitsInStock" HeaderText="UnitsInStock"
            SortExpression="UnitsInStock" />
        <asp:BoundField DataField="UnitsOnOrder" HeaderText="UnitsOnOrder"
            SortExpression="UnitsOnOrder" />
        <asp:BoundField DataField="ReorderLevel" HeaderText="ReorderLevel"
            SortExpression="ReorderLevel" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
        <asp:BoundField DataField="CategoryName" HeaderText="CategoryName"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="SupplierName"
            ReadOnly="True" SortExpression="SupplierName" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    InsertMethod="AddProduct" OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
    <InsertParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="unitsInStock" Type="Int16" />
        <asp:Parameter Name="unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="reorderLevel" Type="Int16" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
    </InsertParameters>
</asp:ObjectDataSource>
```

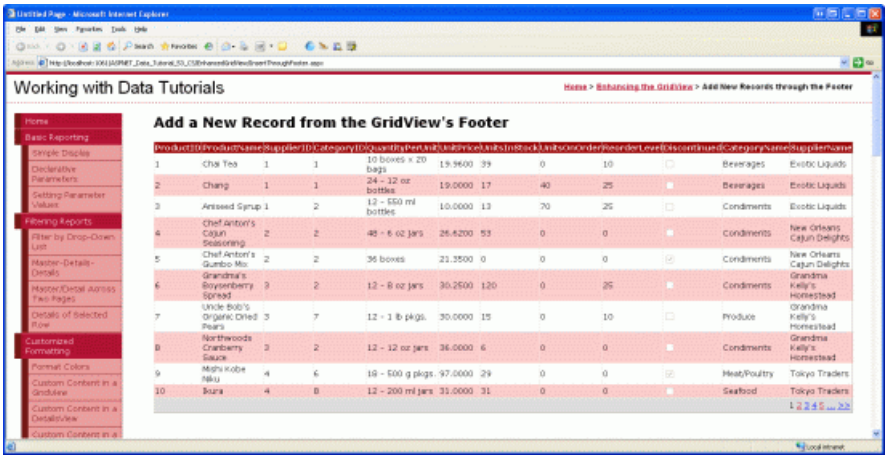


Figure 5: All Product Data Fields are Displayed in a Paged GridView (Click to view full-size image)

Step 2: Adding a Footer Row

Along with its header and data rows, the GridView includes a footer row. The header and footer rows are displayed depending on the values of the GridView s `ShowHeader` and `ShowFooter` properties. To show the footer row, simply set the `ShowFooter` property to `true`. As Figure 6 illustrates, setting the `ShowFooter` property to `true` adds a footer row to the grid.

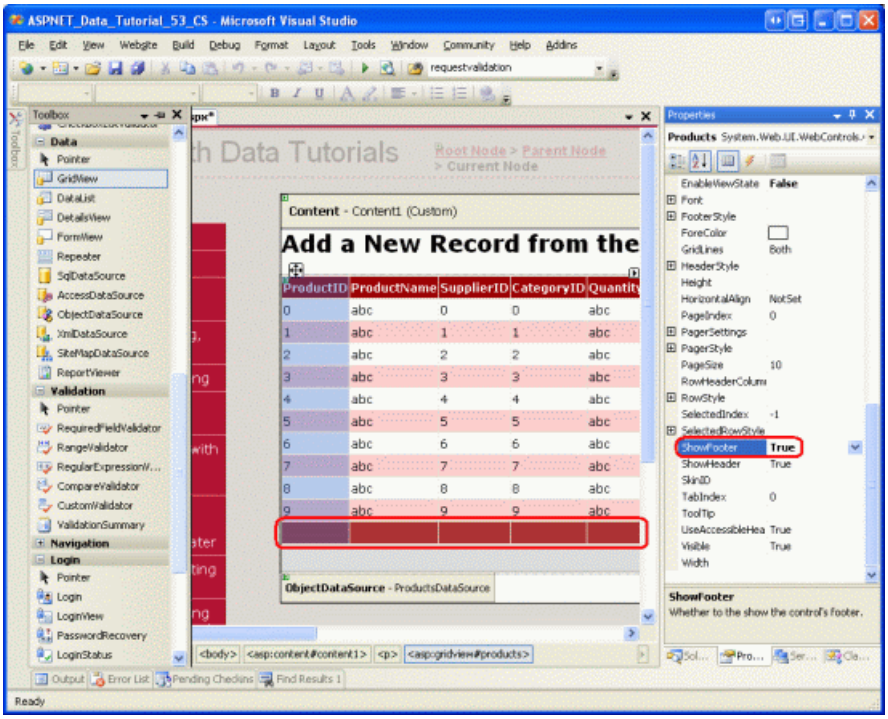


Figure 6: To Display the Footer Row, Set `ShowFooter` to `True` (Click to view full-size image)

Note that the footer row has a dark red background color. This is due to the DataWebControls Theme we created and applied to all pages back in the Displaying Data With the ObjectDataSource tutorial. Specifically, the `GridView.skin` file configures the `FooterStyle` property such that is uses the `FooterStyle` CSS class. The `FooterStyle` class is defined in `Styles.css` as follows:

```
.FooterStyle
{
    background-color: #a33;
    color: White;
    text-align: right;
}
```

> **NOTE**
>
> We ve explored using the GridView s footer row in previous tutorials. If needed, refer back to the Displaying Summary Information in the GridView's Footer tutorial for a refresher.

After setting the `ShowFooter` property to `true`, take a moment to view the output in a browser. Currently the footer row doesn t contain any text or Web controls. In Step 3 we'll modify the footer for each GridView field so that it includes the appropriate inserting interface.

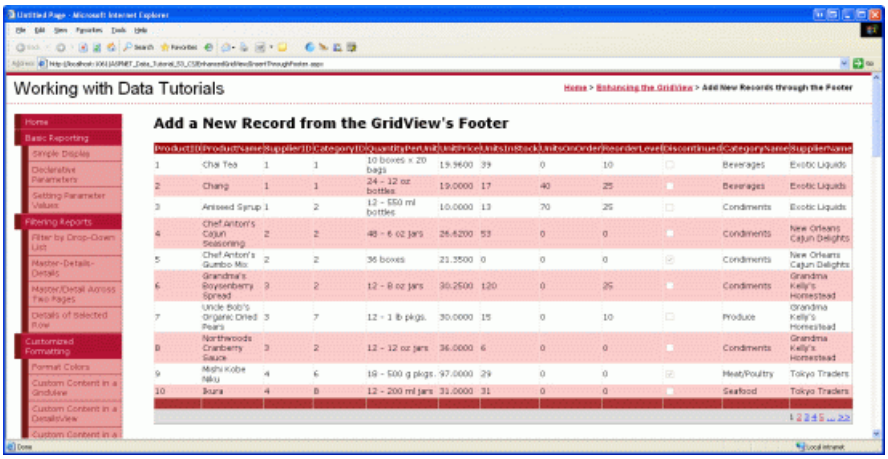


Figure 7: The Empty Footer Row is Displayed Above the Paging Interface Controls (Click to view full-size image)

Step 3: Customizing the Footer Row

Back in the Using TemplateFields in the GridView Control tutorial we saw how to greatly customize the display of a particular GridView column using TemplateFields (as opposed to BoundFields or CheckBoxFields); in Customizing the Data Modification Interface we looked at using TemplateFields to customize the editing interface in a GridView. Recall that a TemplateField is composed of a number of templates that defines the mix of markup, Web controls, and databinding syntax used for certain types of rows. The `ItemTemplate`, for example, specifies the template used for read-only rows, while the `EditItemTemplate` defines the template for the editable row.

Along with the `ItemTemplate` and `EditItemTemplate`, the TemplateField also includes a `FooterTemplate` that specifies the content for the footer row. Therefore, we can add the Web controls needed for each field s inserting interface into the `FooterTemplate`. To start, convert all of the fields in the GridView to TemplateFields. This can be done by clicking the Edit Columns link in the GridView s smart tag, selecting each field in the lower left corner, and clicking the Convert this field into a TemplateField link.

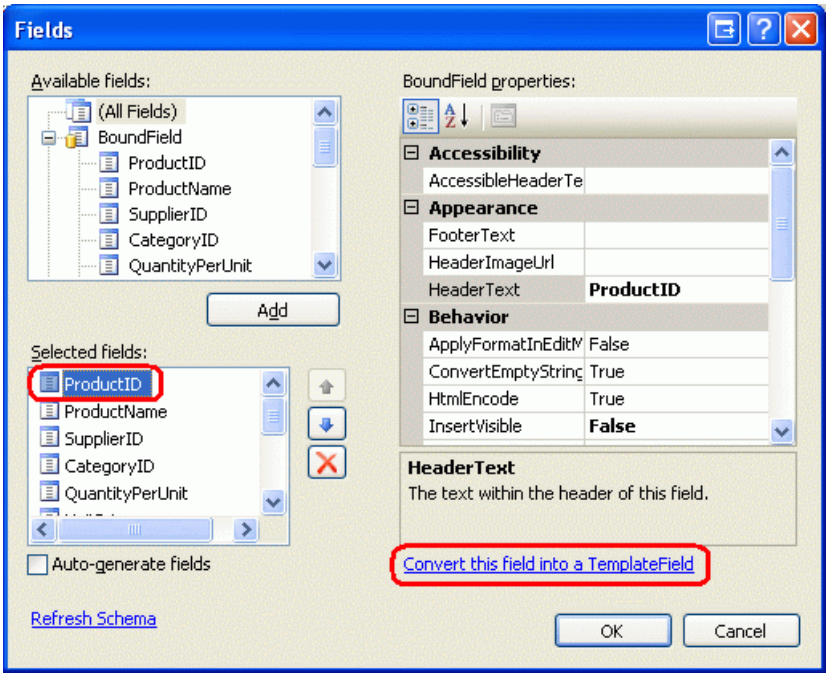


Figure 8: Convert Each Field Into a TemplateField

Clicking the Convert this field into a TemplateField turns the current field type into an equivalent TemplateField. For example, each BoundField is replaced by a TemplateField with an `ItemTemplate` that contains a Label that displays the corresponding data field and an `EditItemTemplate` that displays the data field in a TextBox. The `ProductName` BoundField has been converted into the following TemplateField markup:

```
<asp:TemplateField HeaderText="ProductName" SortExpression="ProductName">
    <EditItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
            Text='<%# Bind("ProductName") %>'></asp:TextBox>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label2" runat="server"
            Text='<%# Bind("ProductName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

Likewise, the `Discontinued` CheckBoxField has been converted into a TemplateField whose `ItemTemplate` and `EditItemTemplate` contain a CheckBox Web control (with the `ItemTemplate` s CheckBox disabled). The read-only `ProductID` BoundField has been converted into a TemplateField with a Label control in both the `ItemTemplate` and `EditItemTemplate`. In short, converting an existing GridView field into a TemplateField is a quick and easy way to switch to the more customizable TemplateField without losing any of the existing field s functionality.

Since the GridView we re working with doesn t support editing, feel free to remove the `EditItemTemplate` from each TemplateField, leaving just the `ItemTemplate`. After doing this, your GridView s declarative markup should look like the following:

```
<asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
    AllowPaging="True" EnableViewState="False" ShowFooter="True">
    <Columns>
        <asp:TemplateField HeaderText="ProductID" InsertVisible="False"
            SortExpression="ProductID">
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("ProductID") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="ProductName" SortExpression="ProductName">
            <ItemTemplate>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="SupplierID" SortExpression="SupplierID">
            <ItemTemplate>
                <asp:Label ID="Label3" runat="server"
                    Text='<%# Bind("SupplierID") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="CategoryID" SortExpression="CategoryID">
            <ItemTemplate>
                <asp:Label ID="Label4" runat="server"
                    Text='<%# Bind("CategoryID") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="QuantityPerUnit"
            SortExpression="QuantityPerUnit">
            <ItemTemplate>
                <asp:Label ID="Label5" runat="server"
                    Text='<%# Bind("QuantityPerUnit") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="UnitPrice" SortExpression="UnitPrice">
            <ItemTemplate>
                <asp:Label ID="Label6" runat="server"
                    Text='<%# Bind("UnitPrice") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="UnitsInStock"
```

```
            SortExpression="UnitsInStock">
            <ItemTemplate>
                <asp:Label ID="Label7" runat="server"
                    Text='<%# Bind("UnitsInStock") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="UnitsOnOrder"
            SortExpression="UnitsOnOrder">
            <ItemTemplate>
                <asp:Label ID="Label8" runat="server"
                    Text='<%# Bind("UnitsOnOrder") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="ReorderLevel"
            SortExpression="ReorderLevel">
            <ItemTemplate>
                <asp:Label ID="Label9" runat="server"
                    Text='<%# Bind("ReorderLevel") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Discontinued"
            SortExpression="Discontinued">
            <ItemTemplate>
                <asp:CheckBox ID="CheckBox1" runat="server"
                    Checked='<%# Bind("Discontinued") %>' Enabled="false" />
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="CategoryName"
            SortExpression="CategoryName">
            <ItemTemplate>
                <asp:Label ID="Label10" runat="server"
                    Text='<%# Bind("CategoryName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="SupplierName"
            SortExpression="SupplierName">
            <ItemTemplate>
                <asp:Label ID="Label11" runat="server"
                    Text='<%# Bind("SupplierName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
```

Now that each GridView field has been converted into a TemplateField, we can enter the appropriate inserting interface into each field s `FooterTemplate`. Some of the fields will not have an inserting interface (`ProductID`, for instance); others will vary in the Web controls used to collect the new product s information.

To create the editing interface, choose the Edit Templates link from the GridView s smart tag. Then, from the drop-down list, select the appropriate field s `FooterTemplate` and drag the appropriate control from the Toolbox onto the Designer.

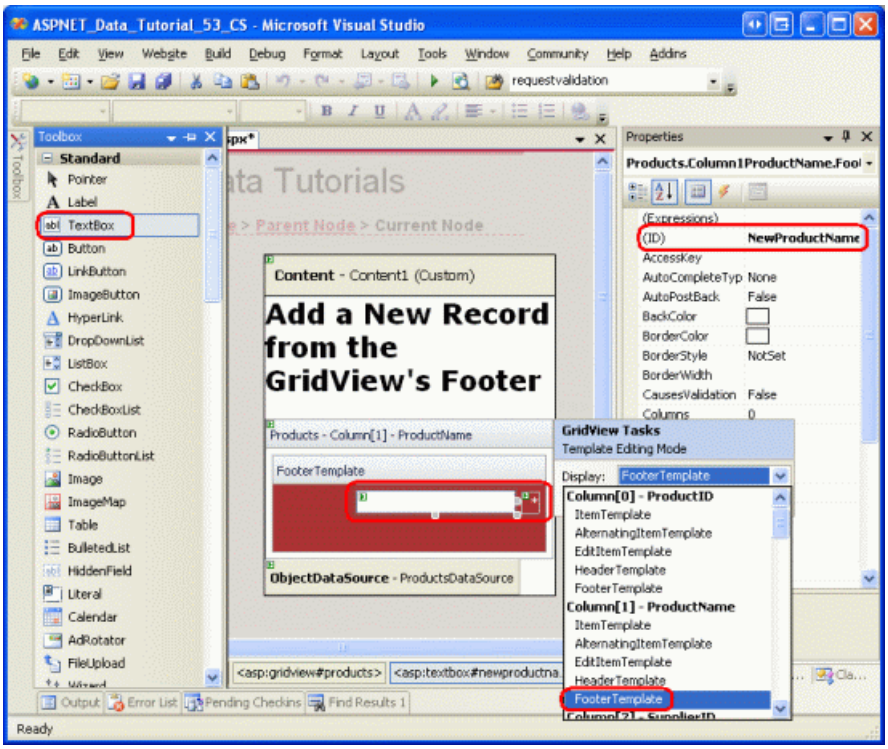


Figure 9: Add the Appropriate Inserting Interface to Each Field s `FooterTemplate` (Click to view full-size image)

The following bulleted list enumerates the GridView fields, specifying the inserting interface to add:

- `ProductID` none.
- `ProductName` add a TextBox and set its `ID` to `NewProductName`. Add a RequiredFieldValidator control as well to ensure that the user enters a value for the new product s name.
- `SupplierID` none.
- `CategoryID` none.
- `QuantityPerUnit` add a TextBox, setting its `ID` to `NewQuantityPerUnit`.
- `UnitPrice` add a TextBox named `NewUnitPrice` and a CompareValidator that ensures the value entered is a currency value greater than or equal to zero.
- `UnitsInStock` use a TextBox whose `ID` is set to `NewUnitsInStock`. Include a CompareValidator that ensures that the value entered is an integer value greater than or equal to zero.
- `UnitsOnOrder` use a TextBox whose `ID` is set to `NewUnitsOnOrder`. Include a CompareValidator that ensures that the value entered is an integer value greater than or equal to zero.
- `ReorderLevel` use a TextBox whose `ID` is set to `NewReorderLevel`. Include a CompareValidator that ensures that the value entered is an integer value greater than or equal to zero.
- `Discontinued` add a CheckBox, setting its `ID` to `NewDiscontinued`.
- `CategoryName` add a DropDownList and set its `ID` to `NewCategoryID`. Bind it to a new ObjectDataSource named `CategoriesDataSource` and configure it to use the `CategoriesBLL` class s `GetCategories()` method. Have the DropDownList s `ListItem` s display the `CategoryName` data field, using the `CategoryID` data field as their values.
- `SupplierName` add a DropDownList and set its `ID` to `NewSupplierID`. Bind it to a new ObjectDataSource named `SuppliersDataSource` and configure it to use the `SuppliersBLL` class s `GetSuppliers()` method. Have the DropDownList s `ListItem` s display the `CompanyName` data field, using the `SupplierID` data field as their values.

For each of the validation controls, clear out the `ForeColor` property so that the `FooterStyle` CSS class s white foreground color will be used in place of the default red. Also use the `ErrorMessage` property for a detailed description, but set the `Text` property to an asterisk. To prevent the validation control s text from causing the inserting interface to wrap to two lines, set the `FooterStyle` s `Wrap` property to false for each of the

`FooterTemplate` s that use a validation control. Finally, add a ValidationSummary control beneath the GridView and set its `ShowMessageBox` property to `true` and its `ShowSummary` property to `false`.

When adding a new product, we need to provide the `CategoryID` and `SupplierID`. This information is captured through the DropDownLists in the footer cells for the `CategoryName` and `SupplierName` fields. I chose to use these fields as opposed to the `CategoryID` and `SupplierID` TemplateFields because in the grid s data rows the user is likely more interested in seeing the category and supplier names rather than their ID values. Since the `CategoryID` and `SupplierID` values are now being captured in the `CategoryName` and `SupplierName` field s inserting interfaces, we can remove the `CategoryID` and `SupplierID` TemplateFields from the GridView.

Similarly, the `ProductID` is not used when adding a new product, so the `ProductID` TemplateField can be removed as well. However, let s leave the `ProductID` field in the grid. In addition to the TextBoxes, DropDownLists, CheckBoxes, and validation controls that make up the inserting interface, we'll also need an Add button that, when clicked, performs the logic to add the new product to the database. In Step 4 we'll include an Add button in the inserting interface in the `ProductID` TemplateField s `FooterTemplate`.

Feel free to improve the appearance of the various GridView fields. For example, you might want to format the `UnitPrice` values as a currency, right-align the `UnitsInStock`, `UnitsOnOrder`, and `ReorderLevel` fields, and update the `HeaderText` values for the TemplateFields.

After creating the slew of inserting interfaces in the `FooterTemplate` s, removing the `SupplierID`, and `CategoryID` TemplateFields, and improving the aesthetics of the grid through formatting and aligning the TemplateFields, your GridView s declarative markup should look similar to the following:

```
<asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
    AllowPaging="True" EnableViewState="False" ShowFooter="True">
    <Columns>
        <asp:TemplateField HeaderText="ProductID" InsertVisible="False"
            SortExpression="ProductID">
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("ProductID") %>'></asp:Label>
            </ItemTemplate>
            <ItemStyle HorizontalAlign="Center" />
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
            <ItemTemplate>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:Label>
            </ItemTemplate>
            <FooterTemplate>
                <asp:TextBox ID="NewProductName" runat="server"></asp:TextBox>
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                    runat="server" ControlToValidate="NewProductName"
                    Display="Dynamic"  ForeColor="
                    ErrorMessage="You must enter a name for the new product.">
                    * </asp:RequiredFieldValidator>
            </FooterTemplate>
            <FooterStyle Wrap="False" />
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Category" SortExpression="CategoryName">
            <ItemTemplate>
                <asp:Label ID="Label10" runat="server"
                    Text='<%# Bind("CategoryName") %>'></asp:Label>
            </ItemTemplate>
            <FooterTemplate>
                <asp:DropDownList ID="NewCategoryID" runat="server"
                    DataSourceID="CategoriesDataSource"
                    DataTextField="CategoryName" DataValueField="CategoryID">
                </asp:DropDownList>
                <asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
                    OldValuesParameterFormatString="original_{0}"
```

```
                OldValuesParameterFormatString="original_{0}"
                SelectMethod="GetCategories" TypeName="CategoriesBLL">
            </asp:ObjectDataSource>
        </FooterTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Supplier" SortExpression="SupplierName">
        <ItemTemplate>
            <asp:Label ID="Label11" runat="server"
                Text='<%# Bind("SupplierName") %>'></asp:Label>
        </ItemTemplate>
        <FooterTemplate>
            <asp:DropDownList ID="NewSupplierID" runat="server"
                DataSourceID="SuppliersDataSource"
                DataTextField="CompanyName" DataValueField="SupplierID">
            </asp:DropDownList><asp:ObjectDataSource ID="SuppliersDataSource"
                runat="server" OldValuesParameterFormatString="original_{0}"
                SelectMethod="GetSuppliers" TypeName="SuppliersBLL">
            </asp:ObjectDataSource>
        </FooterTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Qty/Unit" SortExpression="QuantityPerUnit">
        <ItemTemplate>
            <asp:Label ID="Label5" runat="server"
                Text='<%# Bind("QuantityPerUnit") %>'></asp:Label>
        </ItemTemplate>
        <FooterTemplate>
            <asp:TextBox ID="NewQuantityPerUnit" runat="server"></asp:TextBox>
        </FooterTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Price" SortExpression="UnitPrice">
        <ItemTemplate>
            <asp:Label ID="Label6" runat="server"
                Text='<%# Bind("UnitPrice", "{0:c}") %>'></asp:Label>
        </ItemTemplate>
        <FooterTemplate>
            $<asp:TextBox ID="NewUnitPrice" runat="server" Columns="8" />
            <asp:CompareValidator ID="CompareValidator1" runat="server"
                ControlToValidate="NewUnitPrice"
                ErrorMessage="You must enter a valid currency value greater than
                    or equal to 0.00. Do not include the currency symbol."
                ForeColor="" Operator="GreaterThanEqual" Type="Currency"
                ValueToCompare="0" Display="Dynamic">
                * </asp:CompareValidator>
        </FooterTemplate>
        <ItemStyle HorizontalAlign="Right" />
        <FooterStyle Wrap="False" />
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Units In Stock"
        SortExpression="Units In Stock">
        <ItemTemplate>
            <asp:Label ID="Label7" runat="server"
                Text='<%# Bind("UnitsInStock") %>'></asp:Label>
        </ItemTemplate>
        <FooterTemplate>
            <asp:TextBox ID="NewUnitsInStock" runat="server" Columns="5" />
            <asp:CompareValidator ID="CompareValidator2" runat="server"
                ControlToValidate="NewUnitsInStock" Display="Dynamic"
                ErrorMessage="You must enter a valid numeric value for units
                    in stock that's greater than or equal to zero."
                ForeColor="" Operator="GreaterThanEqual" Type="Integer"
                    ValueToCompare="0">*</asp:CompareValidator>
        </FooterTemplate>
        <ItemStyle HorizontalAlign="Right" />
        <FooterStyle Wrap="False" />
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Units On Order" SortExpression="UnitsOnOrder">
        <ItemTemplate>
            <asp:Label ID="Label8" runat="server"
                Text='<%# Bind("UnitsOnOrder") %>'></asp:Label>
```

```
            </ItemTemplate>
            <FooterTemplate>
                <asp:TextBox ID="NewUnitsOnOrder" runat="server" Columns="5" />
                <asp:CompareValidator ID="CompareValidator3" runat="server"
                    ControlToValidate="NewUnitsOnOrder" Display="Dynamic"
                    ErrorMessage="You must enter a valid numeric value for units on
                        order that's greater than or equal to zero."
                    ForeColor="" Operator="GreaterThanEqual" Type="Integer"
                    ValueToCompare="0">*</asp:CompareValidator>
            </FooterTemplate>
            <ItemStyle HorizontalAlign="Right" />
            <FooterStyle Wrap="False" />
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Reorder Level" SortExpression="ReorderLevel">
            <ItemTemplate>
                <asp:Label ID="Label9" runat="server"
                    Text='<%# Bind("ReorderLevel") %>'></asp:Label>
            </ItemTemplate>
            <FooterTemplate>
                <asp:TextBox ID="NewReorderLevel" runat="server" Columns="5" />
                <asp:CompareValidator ID="CompareValidator4" runat="server"
                    ControlToValidate="NewReorderLevel" Display="Dynamic"
                    ErrorMessage="You must enter a valid numeric value for reorder
                        level that's greater than or equal to zero."
                    ForeColor="" Operator="GreaterThanEqual" Type="Integer"
                    ValueToCompare="0">*</asp:CompareValidator>
            </FooterTemplate>
            <ItemStyle HorizontalAlign="Right" />
            <FooterStyle Wrap="False" />
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Discontinued" SortExpression="Discontinued">
            <ItemTemplate>
                <asp:CheckBox ID="CheckBox1" runat="server"
                    Checked='<%# Bind("Discontinued") %>' Enabled="false" />
            </ItemTemplate>
            <FooterTemplate>
                <asp:CheckBox ID="NewDiscontinued" runat="server" />
            </FooterTemplate>
            <ItemStyle HorizontalAlign="Center" />
            <FooterStyle HorizontalAlign="Center" />
        </asp:TemplateField>
    </Columns>
</asp:GridView>
```

When viewed through a browser, the GridView s footer row now includes the completed inserting interface (see Figure 10). At this point, the inserting interface doesn t include a means for the user to indicate that she s entered the data for the new product and wants to insert a new record into the database. Also, we ve yet to address how the data entered into the footer will translate into a new record in the `Products` database. In Step 4 we'll look at how to include an Add button to the inserting interface and how to execute code on postback when it s clicked. Step 5 shows how to insert a new record using the data from the footer.

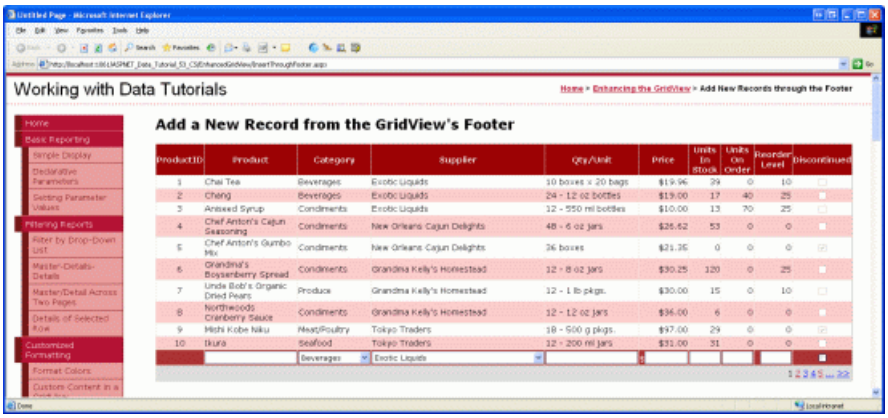


Figure 10: The GridView Footer Provides an Interface for Adding a New Record (Click to view full-size image)

Step 4: Including an Add Button in the Inserting Interface

We need to include an Add button somewhere in the inserting interface since the footer row s inserting interface currently lacks the means for the user to indicate that they have completed entering the new product s information. This could be placed in one of the existing `FooterTemplate` s or we could add a new column to the grid for this purpose. For this tutorial, let s place the Add button in the `ProductID` TemplateField s `FooterTemplate`.

From the Designer, click on the Edit Templates link in the GridView s smart tag and then choose the `ProductID` field s `FooterTemplate` from the drop-down list. Add a Button Web control (or a LinkButton or ImageButton, if you prefer) to the template, setting its ID to `AddProduct`, its `CommandName` to Insert, and its `Text` property to Add as shown in Figure 11.

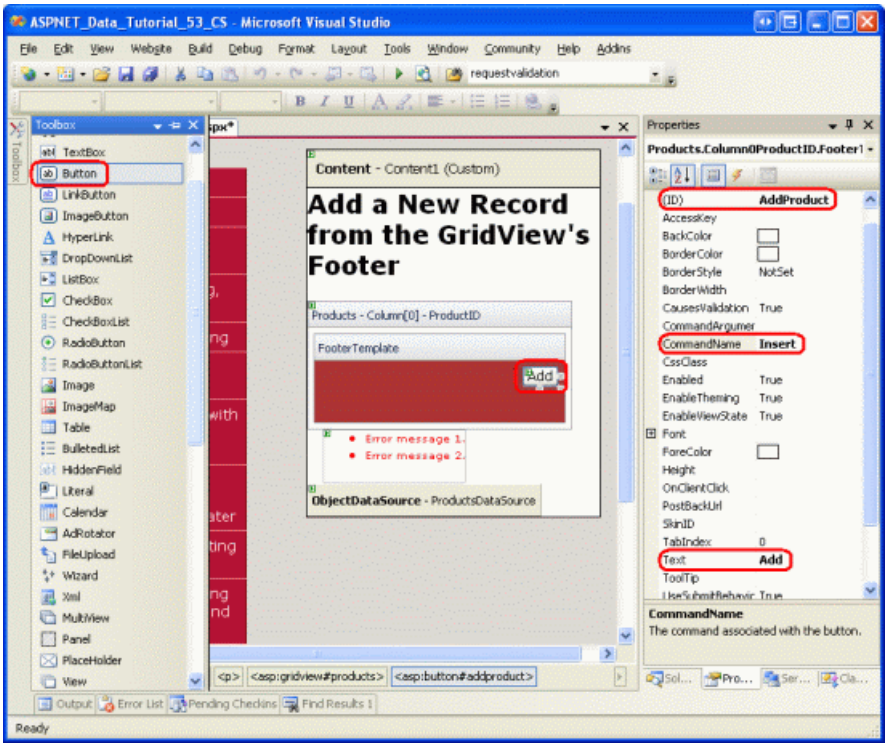


Figure 11: Place the Add Button in the `ProductID` TemplateField s `FooterTemplate` (Click to view full-size image)

Once you ve included the Add button, test out the page in a browser. Note that when clicking the Add button with invalid data in the inserting interface, the postback is short circuited and the ValidationSummary control indicates the invalid data (see Figure 12). With appropriate data entered, clicking the Add button causes a postback. No record is added to the database, however. We will need to write a bit of code to actually perform the insert.

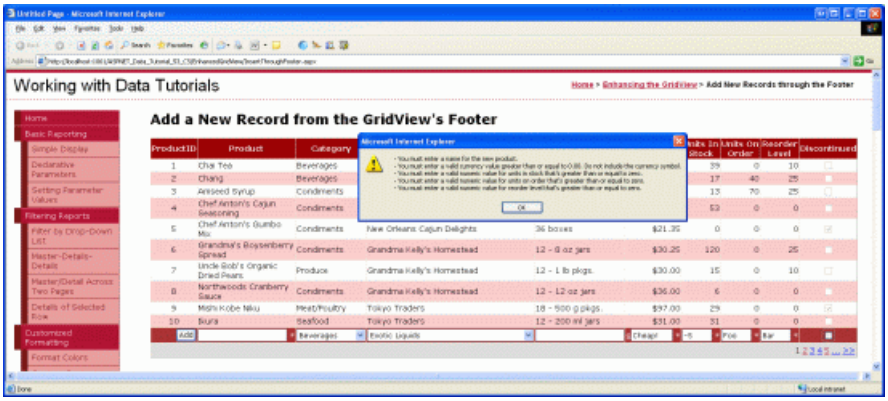


Figure 12: The Add Button s Postback is Short Circuited if There is Invalid Data in the Inserting Interface (Click to view full-size image)

NOTE

The validation controls in the inserting interface were not assigned to a validation group. This works fine so long as the inserting interface is the only set of validation controls on the page. If, however, there are other validation controls on the page (such as validation controls in the grid s editing interface), the validation controls in the inserting interface and Add button s `ValidationGroup` properties should be assigned the same value so as to associate these controls with a particular validation group. See Dissecting the Validation Controls in ASP.NET 2.0 for more information on partitioning the validation controls and buttons on a page into validation groups.

Step 5: Inserting a New Record Into the `Products` Table

When utilizing the built-in editing features of the GridView, the GridView automatically handles all of the work necessary for performing the update. In particular, when the Update button is clicked it copies the values entered from the editing interface to the parameters in the ObjectDataSource s `UpdateParameters` collection and kicks off the update by invoking the ObjectDataSource s `Update()` method. Since the GridView does not provide such built-in functionality for inserting, we must implement code that calls the ObjectDataSource s `Insert()` method and copies the values from the inserting interface to the ObjectDataSource s `InsertParameters` collection.

This insert logic should be executed after the Add button has been clicked. As discussed in the Adding and Responding to Buttons in a GridView tutorial, anytime a Button, LinkButton, or ImageButton in a GridView is clicked, the GridView s `RowCommand` event fires on postback. This event fires whether the Button, LinkButton, or ImageButton was added explicitly such as the Add button in the footer row or if it was automatically added by the GridView (such as the LinkButtons at the top of each column when Enable Sorting is selected, or the LinkButtons in the paging interface when Enable Paging is selected).

Therefore, to respond to the user clicking the Add button, we need to create an event handler for the GridView s `RowCommand` event. Since this event fires whenever *any* Button, LinkButton, or ImageButton in the GridView is clicked, it s vital that we only proceed with the inserting logic if the `CommandName` property passed into the event handler maps to the `CommandName` value of the Add button (Insert). Moreover, we should also only proceed if the validation controls report valid data. To accommodate this, create an event handler for the `RowCommand` event with the following code:

```
protected void Products_RowCommand(object sender, GridViewCommandEventArgs e)
{
    // Insert data if the CommandName == "Insert"
    // and the validation controls indicate valid data...
    if (e.CommandName == "Insert" && Page.IsValid)
    {
        // TODO: Insert new record...
    }
}
```

NOTE

You may be wondering why the event handler bothers checking the `Page.IsValid` property. After all, won't the postback be suppressed if invalid data is provided in the inserting interface? This assumption is correct as long as the user has not disabled JavaScript or has taken steps to circumvent the client-side validation logic. In short, one should never rely strictly on client-side validation; a server-side check for validity should always be performed before working with the data.

In Step 1 we created the `ProductsDataSource` ObjectDataSource such that its `Insert()` method is mapped to the `ProductsBLL` class s `AddProduct` method. To insert the new record into the `Products` table, we can simply invoke the ObjectDataSource s `Insert()` method:

```
protected void Products_RowCommand(object sender, GridViewCommandEventArgs e)
{
    // Insert data if the CommandName == "Insert"
    // and the validation controls indicate valid data...
    if (e.CommandName == "Insert" && Page.IsValid)
    {
        // Insert new record
        ProductsDataSource.Insert();
    }
}
```

Now that the `Insert()` method has been invoked, all that remains is to copy the values from the inserting interface to the parameters passed to the `ProductsBLL` class s `AddProduct` method. As we saw back in the Examining the Events Associated with Inserting, Updating, and Deleting tutorial, this can be accomplished through the ObjectDataSource s `Inserting` event. In the `Inserting` event we need to programmatically reference the controls from the `Products` GridView s footer row and assign their values to the `e.InputParameters` collection. If the user omits a value such as leaving the `ReorderLevel` TextBox blank we need to specify that the value inserted into the database should be `NULL`. Since the `AddProducts` method accepts nullable types for the nullable database fields, simply use a nullable type and set its value to `null` in the case where user input is omitted.

```
protected void ProductsDataSource_Inserting
    (object sender, ObjectDataSourceMethodEventArgs e)
{
    // Programmatically reference Web controls in the inserting interface...
    TextBox NewProductName =
        (TextBox)Products.FooterRow.FindControl("NewProductName");
    DropDownList NewCategoryID =
        (DropDownList)Products.FooterRow.FindControl("NewCategoryID");
    DropDownList NewSupplierID =
        (DropDownList)Products.FooterRow.FindControl("NewSupplierID");
    TextBox NewQuantityPerUnit =
        (TextBox)Products.FooterRow.FindControl("NewQuantityPerUnit");
    TextBox NewUnitPrice =
        (TextBox)Products.FooterRow.FindControl("NewUnitPrice");
    TextBox NewUnitsInStock =
        (TextBox)Products.FooterRow.FindControl("NewUnitsInStock");
    TextBox NewUnitsOnOrder =
        (TextBox)Products.FooterRow.FindControl("NewUnitsOnOrder");
    TextBox NewReorderLevel =
        (TextBox)Products.FooterRow.FindControl("NewReorderLevel");
    CheckBox NewDiscontinued =
        (CheckBox)Products.FooterRow.FindControl("NewDiscontinued");
    // Set the ObjectDataSource's InsertParameters values...
    e.InputParameters["productName"] = NewProductName.Text;

    e.InputParameters["supplierID"] =
        Convert.ToInt32(NewSupplierID.SelectedValue);
    e.InputParameters["categoryID"] =
        Convert.ToInt32(NewCategoryID.SelectedValue);

    string quantityPerUnit = null;
    if (!string.IsNullOrEmpty(NewQuantityPerUnit.Text))
        quantityPerUnit = NewQuantityPerUnit.Text;
    e.InputParameters["quantityPerUnit"] = quantityPerUnit;
    decimal? unitPrice = null;
    if (!string.IsNullOrEmpty(NewUnitPrice.Text))
        unitPrice = Convert.ToDecimal(NewUnitPrice.Text);
    e.InputParameters["unitPrice"] = unitPrice;
    short? unitsInStock = null;
    if (!string.IsNullOrEmpty(NewUnitsInStock.Text))
        unitsInStock = Convert.ToInt16(NewUnitsInStock.Text);
    e.InputParameters["unitsInStock"] = unitsInStock;
    short? unitsOnOrder = null;
    if (!string.IsNullOrEmpty(NewUnitsOnOrder.Text))
        unitsOnOrder = Convert.ToInt16(NewUnitsOnOrder.Text);
    e.InputParameters["unitsOnOrder"] = unitsOnOrder;
    short? reorderLevel = null;
    if (!string.IsNullOrEmpty(NewReorderLevel.Text))
        reorderLevel = Convert.ToInt16(NewReorderLevel.Text);
    e.InputParameters["reorderLevel"] = reorderLevel;

    e.InputParameters["discontinued"] = NewDiscontinued.Checked;
}
```

With the `Inserting` event handler completed, new records can be added to the `Products` database table via the GridView s footer row. Go ahead and try adding several new products.

Enhancing and Customizing the Add Operation

Currently, clicking the Add button adds a new record to the database table but does not provide any sort of visual feedback that the record has successfully been added. Ideally, a Label Web control or client-side alert box would inform the user that their insert has completed with success. I leave this as an exercise for the reader.

The GridView used in this tutorial does not apply any sort order to the listed products, nor does it allow the end

user to sort the data. Consequently, the records are ordered as they are in the database by their primary key field. Since each new record has a `ProductID` value greater than the last one, every time a new product is added it is tacked on to the end of the grid. Therefore, you may want to automatically send the user to the last page of the GridView after adding a new record. This can be accomplished by adding the following line of code after the call to `ProductsDataSource.Insert()` in the `RowCommand` event handler to indicate that the user needs to be sent to the last page after binding the data to the GridView:

```
// Indicate that the user needs to be sent to the last page
SendUserToLastPage = true;
```

`SendUserToLastPage` is a page-level Boolean variable that is initially assigned a value of `false`. In the GridView s `DataBound` event handler, if `SendUserToLastPage` is false, the `PageIndex` property is updated to send the user to the last page.

```
protected void Products_DataBound(object sender, EventArgs e)
{
    // Send user to last page of data, if needed
    if (SendUserToLastPage)
        Products.PageIndex = Products.PageCount - 1;
}
```

The reason the `PageIndex` property is set in the `DataBound` event handler (as opposed to the `RowCommand` event handler) is because when the `RowCommand` event handler fires we ve yet to add the new record to the `Products` database table. Therefore, in the `RowCommand` event handler the last page index (`PageCount - 1`) represents the last page index *before* the new product has been added. For the majority of products being added, the last page index is the same after adding the new product. But when the added product results in a new last page index, if we incorrectly update the `PageIndex` in the `RowCommand` event handler then we'll be taken to the second to last page (the last page index prior to adding the new product) as opposed to the new last page index. Since the `DataBound` event handler fires after the new product has been added and the data rebound to the grid, by setting the `PageIndex` property there we know we re getting the correct last page index.

Finally, the GridView used in this tutorial is quite wide due to the number of fields that must be collected for adding a new product. Due to this width, a DetailsView s vertical layout might be preferred. The GridView s overall width could be reduced by collecting fewer inputs. Perhaps we don t need to collect the `UnitsOnOrder`, `UnitsInStock`, and `ReorderLevel` fields when adding a new product, in which case these fields could be removed from the GridView.

To adjust the data collected, we can use one of two approaches:

- Continue to use the `AddProduct` method that expects values for the `UnitsOnOrder`, `UnitsInStock`, and `ReorderLevel` fields. In the `Inserting` event handler, provide hard-coded, default values to use for these inputs that have been removed from the inserting interface.
- Create a new overload of the `AddProduct` method in the `ProductsBLL` class that does not accept inputs for the `UnitsOnOrder`, `UnitsInStock`, and `ReorderLevel` fields. Then, in the ASP.NET page, configure the ObjectDataSource to use this new overload.

Either option will work equally as well. In past tutorials we used the latter option, creating multiple overloads for the `ProductsBLL` class s `UpdateProduct` method.

Summary

The GridView lacks the built-in inserting capabilities found in the DetailsView and FormView, but with a bit of effort an inserting interface can be added to the footer row. To display the footer row in a GridView simply set its `ShowFooter` property to `true`. The footer row content can be customized for each field by converting the field to

a TemplateField and adding the inserting interface to the `FooterTemplate`. As we saw in this tutorial, the `FooterTemplate` can contain Buttons, TextBoxes, DropDownLists, CheckBoxes, data source controls for populating data-driven Web controls (such as DropDownLists), and validation controls. Along with controls for collecting the user s input, an Add Button, LinkButton, or ImageButton is needed.

When the Add button is clicked, the ObjectDataSource s `Insert()` method is invoked to start the inserting workflow. The ObjectDataSource will then call the configured insert method (the `ProductsBLL` class s `AddProduct` method, in this tutorial). We must copy the values from the GridView s inserting interface to the ObjectDataSource s `InsertParameters` collection prior to the insert method being invoked. This can be accomplished by programmatically referencing the inserting interface Web controls in the ObjectDataSource s `Inserting` event handler.

This tutorial completes our look at techniques for enhancing the GridView s appearance. The next set of tutorials will examine how to work with binary data such as images, PDFs, Word documents, and so on and the data Web controls.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Bernadette Leigh. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Adding a GridView Column of Radio Buttons (VB)

2/20/2020 • 26 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> This tutorial looks at how to add a column of radio buttons to a GridView control to provide the user with a more intuitive way of selecting a single row of the GridView.

Introduction

The GridView control offers a great deal of built-in functionality. It includes a number of different fields for displaying text, images, hyperlinks, and buttons. It supports templates for further customization. With a few clicks of the mouse, it s possible to make a GridView where each row can be selected via a button, or to enable editing or deleting capabilities. Despite the plethora of provided features, there will often be situations in which additional, non-supported features will need to be added. In this tutorial and the next two we'll examine how to enhance the GridView s functionality to include additional features.

This tutorial and the next one focus on enhancing the row-selection process. As examined in the Master/Detail Using a Selectable Master GridView with a Details DetailView, we can add a CommandField to the GridView that includes a Select button. When clicked, a postback ensues and the GridView s `SelectedIndex` property is updated to the index of the row whose Select button was clicked. In the Master/Detail Using a Selectable Master GridView with a Details DetailView tutorial, we saw how to use this feature to display details for the selected GridView row.

While the Select button works in many situations, it may not work as well for others. Rather than using a button, two other user interface elements are commonly used for selection: the radio button and checkbox. We can augment the GridView so that instead of a Select button, each row contains a radio button or checkbox. In scenarios where the user can only select one of the GridView records, the radio button might be preferred over the Select button. In situations where the user can potentially select multiple records such as in a web-based email application, where a user might want to select multiple messages to delete the checkbox offers functionality that is not available from the Select button or radio button user interfaces.

This tutorial looks at how to add a column of radio buttons to the GridView. The proceeding tutorial explores using checkboxes.

Step 1: Creating the Enhancing the GridView Web Pages

Before we start enhancing the GridView to include a column of radio buttons, let s first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial and the next two. Start by adding a new folder named `EnhancedGridView`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `RadioButtonField.aspx`
- `CheckBoxField.aspx`
- `InsertThroughFooter.aspx`

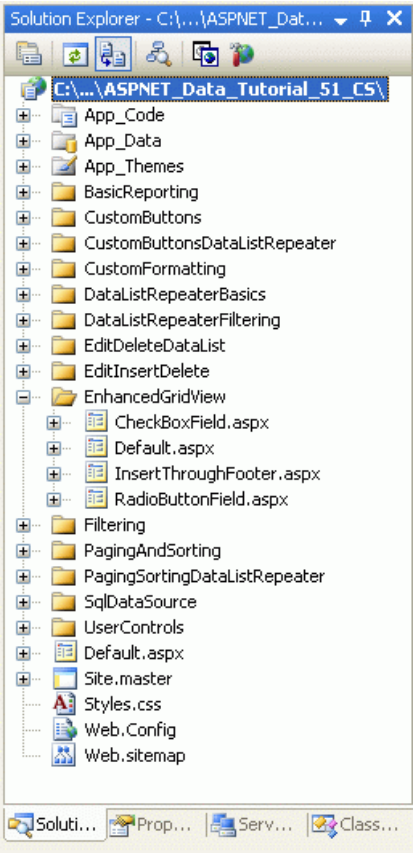


Figure 1: Add the ASP.NET Pages for the SqlDataSource-Related Tutorials

Like in the other folders, `Default.aspx` in the `EnhancedGridView` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

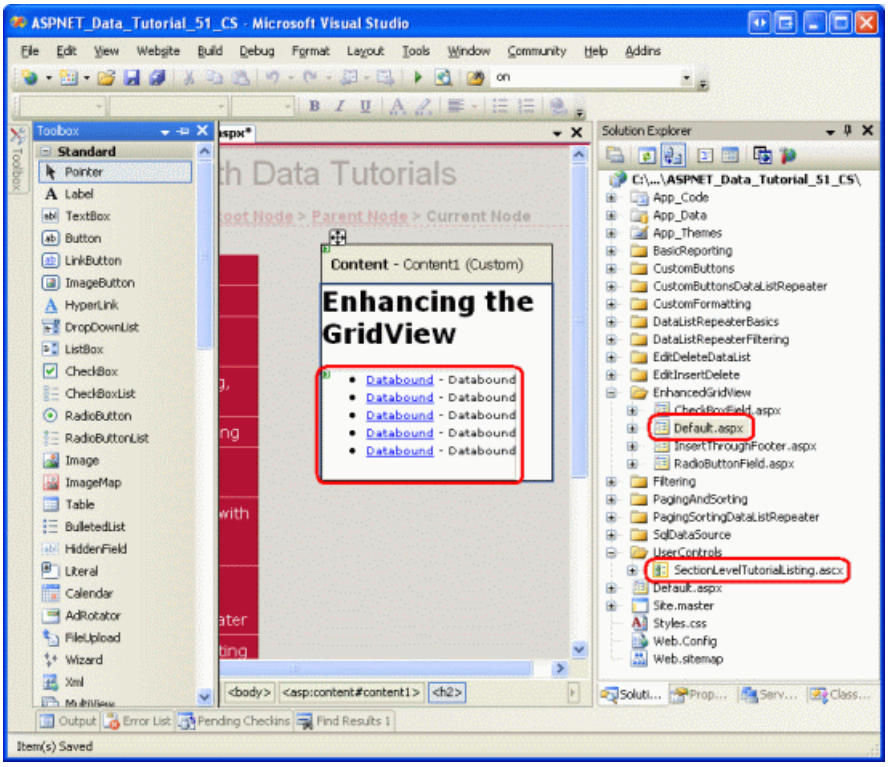


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add these four pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Using the SqlDataSource Control `<siteMapNode>`:

```
<siteMapNode
    title="Enhancing the GridView"
    url="~/EnhancedGridView/Default.aspx"
    description="Augment the user experience of the GridView control.">
    <siteMapNode
        url="~/EnhancedGridView/RadioButtonField.aspx"
        title="Selection via a Radio Button Column"
        description="Explore how to add a column of radio buttons in the GridView." />
    <siteMapNode
        url="~/EnhancedGridView/CheckBoxField.aspx"
        title="Selection via a Checkbox Column"
        description="Select multiple records in the GridView by using a column of
            checkboxes." />
    <siteMapNode
        url="~/EnhancedGridView/InsertThroughFooter.aspx"
        title="Add New Records through the Footer"
        description="Learn how to allow users to add new records through the
            GridView's footer." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the editing, inserting, and deleting tutorials.

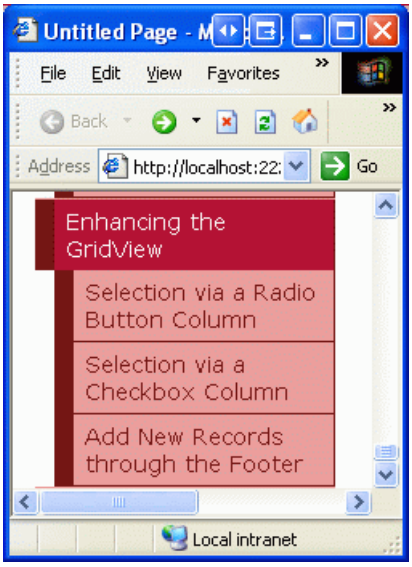


Figure 3: The Site Map Now Includes Entries for the Enhancing the GridView Tutorials

Step 2: Displaying the Suppliers in a GridView

For this tutorial let s build a GridView that lists the suppliers from the USA, with each GridView row providing a radio button. After selecting a supplier via the radio button, the user can view the supplier s products by clicking a button. While this task may sound trivial, there are a number of subtleties that make it particularly tricky. Before we delve into these subtleties, let s first get a GridView listing the suppliers.

Start by opening the `RadioButtonField.aspx` page in the `EnhancedGridView` folder by dragging a GridView from the Toolbox onto the Designer. Set the GridView s `ID` to `Suppliers` and, from its smart tag, choose to create a new data source. Specifically, create an ObjectDataSource named `SuppliersDataSource` that pulls its data from the `SuppliersBLL` object.

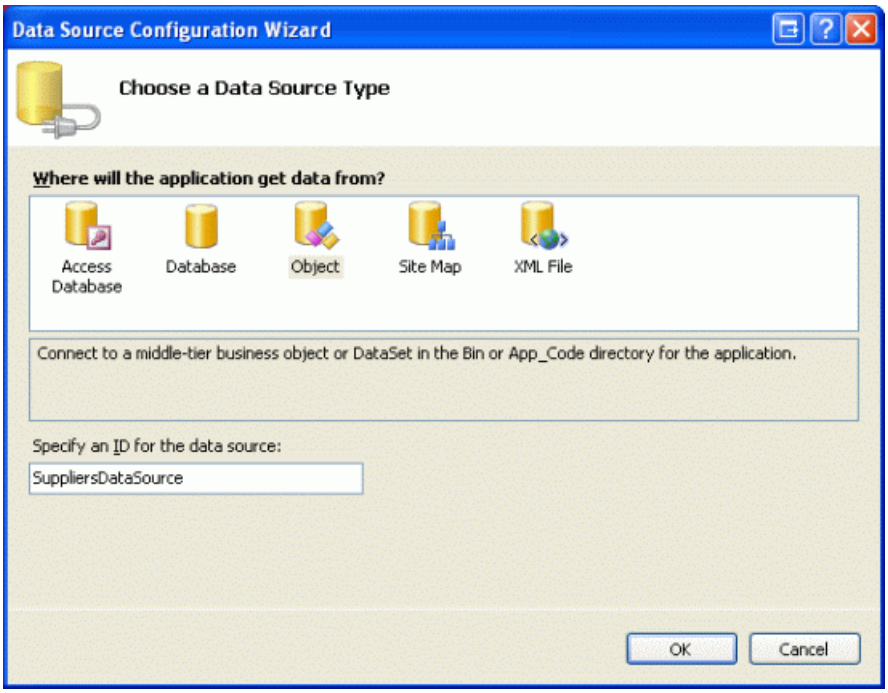


Figure 4: Create a New ObjectDataSource Named `SuppliersDataSource` (Click to view full-size image)

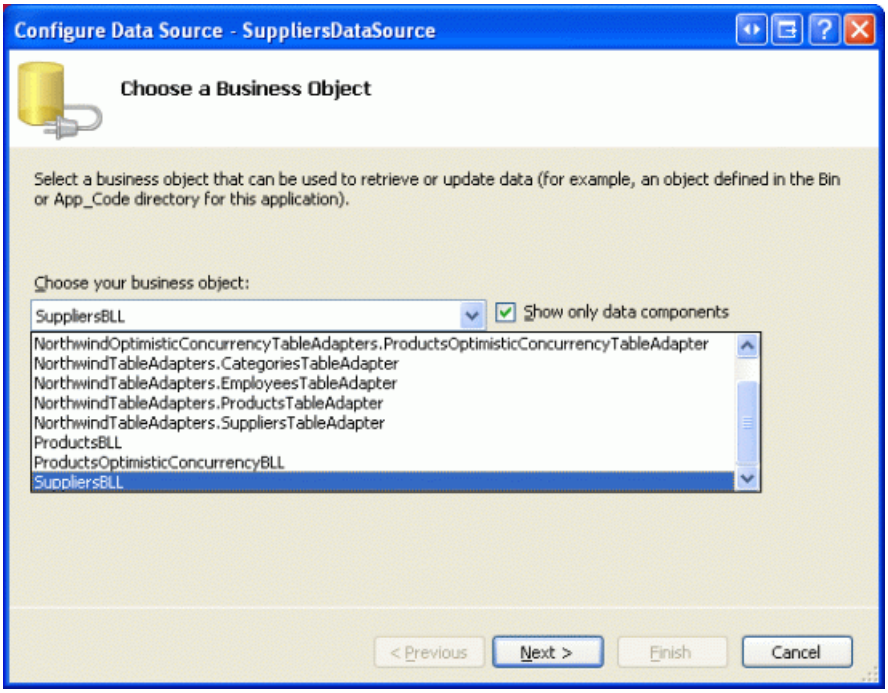


Figure 5: Configure the ObjectDataSource to Use the `SuppliersBLL` Class (Click to view full-size image)

Since we only want to list those suppliers in the USA, choose the `GetSuppliersByCountry(country)` method from the drop-down list in the SELECT tab.

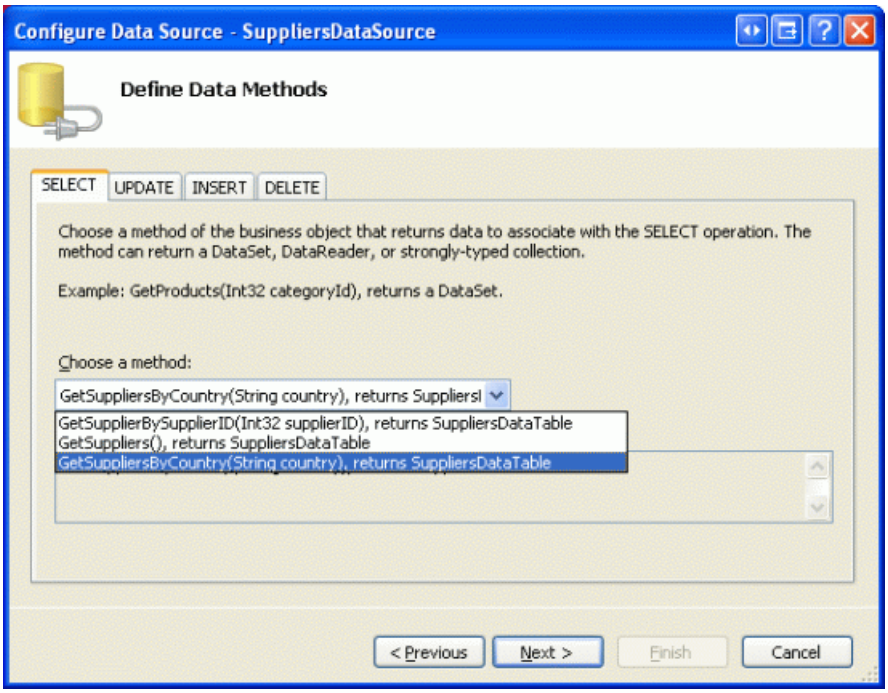


Figure 6: Configure the ObjectDataSource to Use the `SuppliersBLL` Class (Click to view full-size image)

From the UPDATE tab, select the (None) option and click Next.

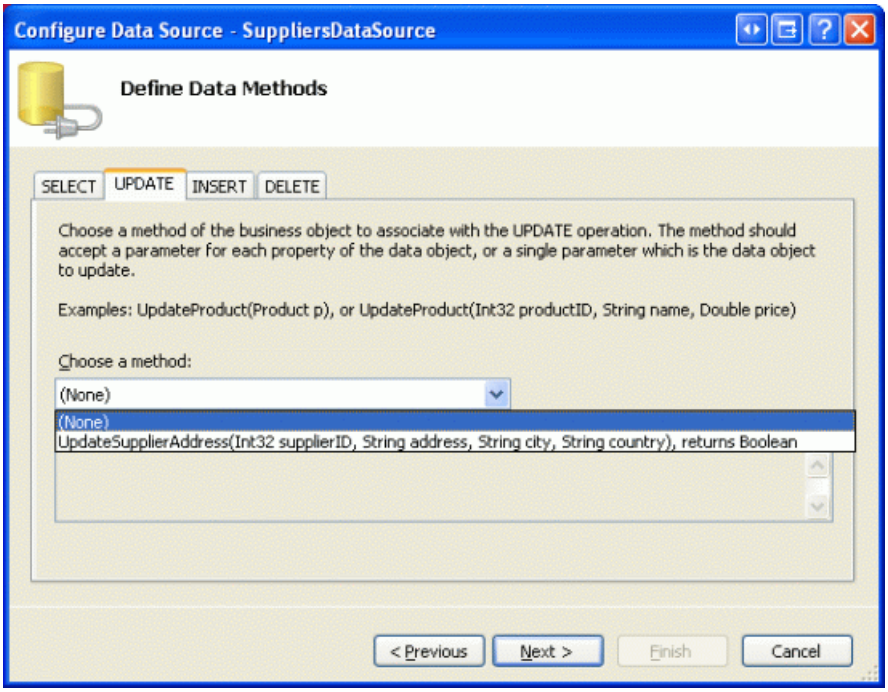


Figure 7: Configure the ObjectDataSource to Use the `SuppliersBLL` Class (Click to view full-size image)

Since the `GetSuppliersByCountry(country)` method accepts a parameter, the Configure Data Source wizard prompts us for the source of that parameter. To specify a hard coded value (USA, in this example), leave the Parameter source drop-down list set to None and enter the default value in the textbox. Click Finish to complete the wizard.

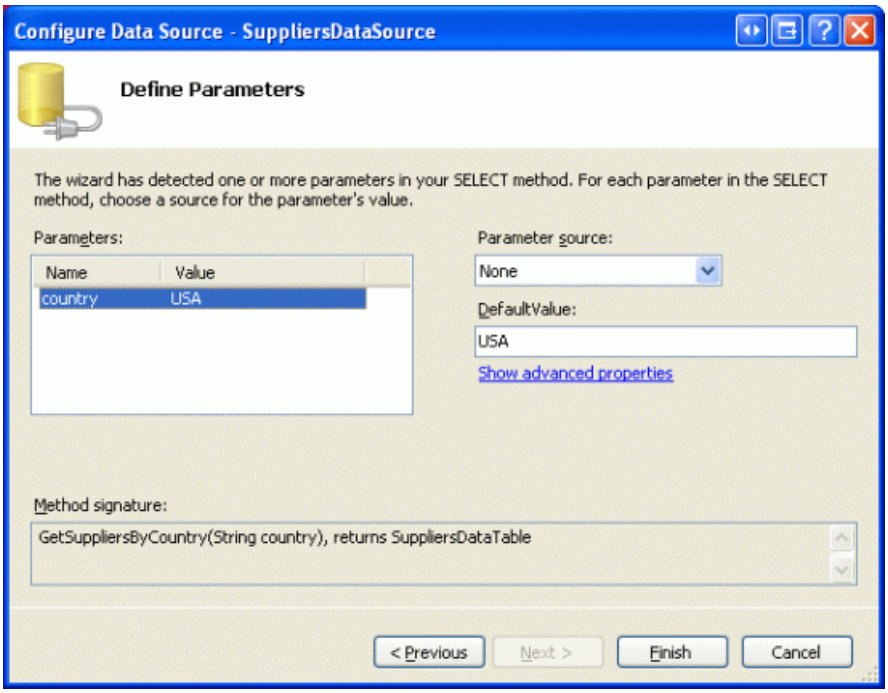


Figure 8: Use USA as the Default Value for the `country` Parameter (Click to view full-size image)

After completing the wizard, the GridView will include a BoundField for each of the supplier data fields. Remove all but the `CompanyName`, `City`, and `Country` BoundFields, and rename the `CompanyName` BoundFields `HeaderText` property to Supplier. After doing so, the GridView and ObjectDataSource declarative syntax should look similar to the following.

```
<asp:GridView ID="Suppliers" runat="server" AutoGenerateColumns="False"
    DataKeyNames="SupplierID" DataSourceID="SuppliersDataSource"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="CompanyName" HeaderText="Supplier"
            SortExpression="CompanyName" />
        <asp:BoundField DataField="City" HeaderText="City"
            SortExpression="City" />
        <asp:BoundField DataField="Country" HeaderText="Country"
            SortExpression="Country" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="SuppliersDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetSuppliersByCountry" TypeName="SuppliersBLL">
    <SelectParameters>
        <asp:Parameter DefaultValue="USA" Name="country" Type="String" />
    </SelectParameters>
</asp:ObjectDataSource>
```

For this tutorial, let s allow the user to view the selected supplier s products on the same page as the supplier list, or on a different page. To accommodate this, add two Button Web controls to the page. I ve set the `ID` s of these two Buttons to `ListProducts` and `SendToProducts`, with the idea that when `ListProducts` is clicked a postback will occur and the selected supplier s products will be listed on the same page, but when `SendToProducts` is clicked, the user will be whisked to a another page that lists the products.

Figure 9 shows the `Suppliers` GridView and the two Button Web controls when viewed through a browser.

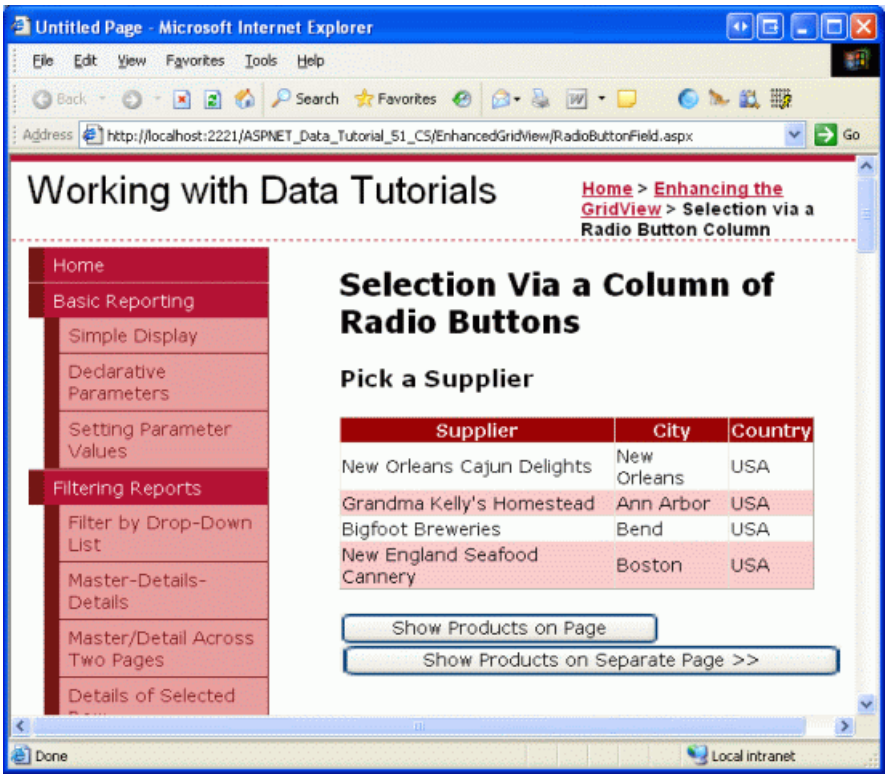


Figure 9: Those Suppliers from the USA Have Their Name, City, and Country Information Listed (Click to view full-size image)

Step 3: Adding a Column of Radio Buttons

At this point the `Suppliers` GridView has three BoundFields displaying the company name, city, and country of each supplier in the USA. It is still lacking a column of radio buttons, however. Unfortunately, the GridView doesn t include a built-in RadioButtonField, otherwise we could just add that to the grid and be done. Instead, we can add a TemplateField and configure its `ItemTemplate` to render a radio button, resulting in a radio button for each GridView row.

Initially, we might assume that the desired user interface can be implemented by adding a RadioButton Web control to the `ItemTemplate` of a TemplateField. While this will indeed add a single radio button to each row of the GridView, the radio buttons cannot be grouped and therefore are not mutually exclusive. That is, an end user is able to select multiple radio buttons simultaneously from the GridView.

Even though using a TemplateField of RadioButton Web controls does not offer the functionality we need, let s implement this approach, as it s worthwhile to examine why the resulting radio buttons are not grouped. Start by adding a TemplateField to the Suppliers GridView, making it the leftmost field. Next, from the GridView s smart tag, click the Edit Templates link and drag a RadioButton Web control from the Toolbox into the TemplateField s `ItemTemplate` (see Figure 10). Set the RadioButton s `ID` property to `RowSelector` and the `GroupName` property to `SuppliersGroup`.

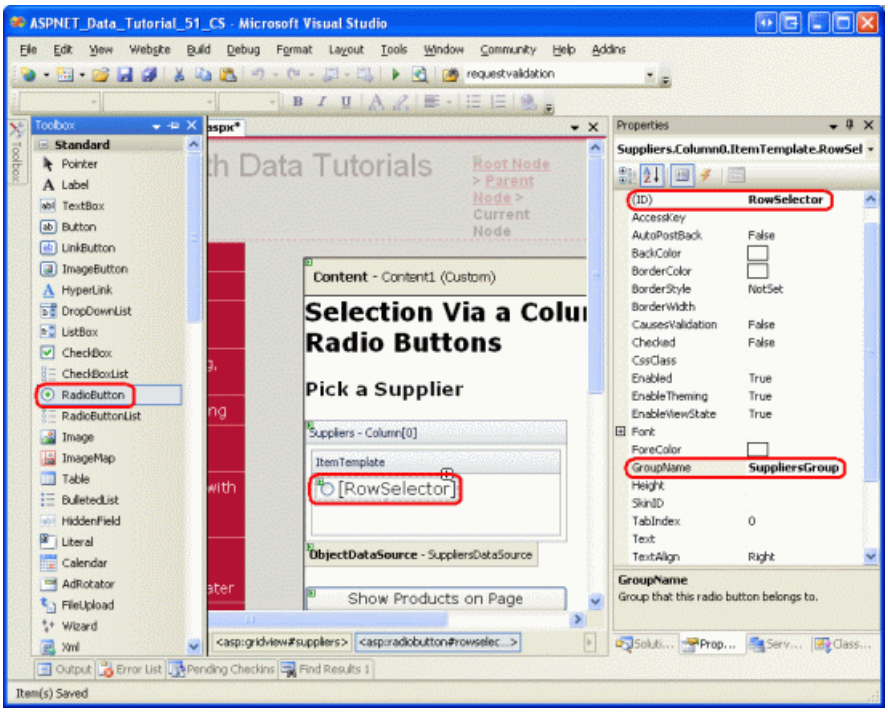


Figure 10: Add a RadioButton Web Control to the `ItemTemplate` (Click to view full-size image)

After making these additions through the Designer, your GridView s markup should look similar to the following:

```
<asp:GridView ID="Suppliers" runat="server" AutoGenerateColumns="False"
    DataKeyNames="SupplierID" DataSourceID="SuppliersDataSource"
    EnableViewState="False">
    <Columns>
        <asp:TemplateField>
            <ItemTemplate>
                <asp:RadioButton ID="RowSelector" runat="server"
                    GroupName="SuppliersGroup" />
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="CompanyName" HeaderText="Supplier"
            SortExpression="CompanyName" />
        <asp:BoundField DataField="City" HeaderText="City"
            SortExpression="City" />
        <asp:BoundField DataField="Country" HeaderText="Country"
            SortExpression="Country" />
    </Columns>
</asp:GridView>
```

The RadioButton s `GroupName` property is what is used to group a series of radio buttons. All RadioButton controls with the same `GroupName` value are considered grouped; only one radio button can be selected from a group at a time. The `GroupName` property specifies the value for the rendered radio button s `name` attribute. The browser examines the radio buttons `name` attributes to determine the radio button groupings.

With the RadioButton Web control added to the `ItemTemplate`, visit this page through a browser and click on the radio buttons in the grid s rows. Notice how the radio buttons are not grouped, making it possible to select all of the rows, as Figure 11 shows.

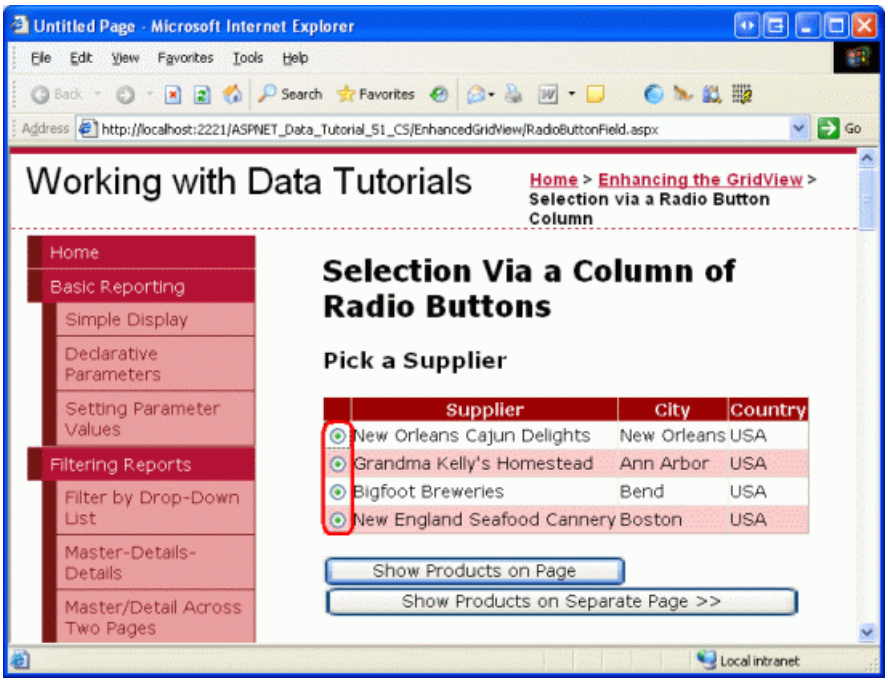


Figure 11: The GridView s Radio Buttons are Not Grouped (Click to view full-size image)

The reason the radio buttons are not grouped is because their rendered `name` attributes are different, despite having the same `GroupName` property setting. To see these differences, do a View/Source from the browser and examine the radio button markup:

```
<input id="ctl00_MainContent_Suppliers_ctl02_RowSelector"
    name="ctl00$MainContent$Suppliers$ctl02$SuppliersGroup"
    type="radio" value="RowSelector" />
<input id="ctl00_MainContent_Suppliers_ctl03_RowSelector"
    name="ctl00$MainContent$Suppliers$ctl03$SuppliersGroup"
    type="radio" value="RowSelector" />
<input id="ctl00_MainContent_Suppliers_ctl04_RowSelector"
    name="ctl00$MainContent$Suppliers$ctl04$SuppliersGroup"
    type="radio" value="RowSelector" />
<input id="ctl00_MainContent_Suppliers_ctl05_RowSelector"
    name="ctl00$MainContent$Suppliers$ctl05$SuppliersGroup"
    type="radio" value="RowSelector" />
```

Notice how both the `name` and `id` attributes are not the exact values as specified in the Properties window, but are prepended with a number of other `ID` values. The additional `ID` values added to the front of the rendered `id` and `name` attributes are the `ID` s of the radio buttons parent controls the `GridViewRow` s `ID` s, the GridView s `ID`, the Content control s `ID`, and the Web Form s `ID`. These `ID` s are added so that each rendered Web control in the GridView has a unique `id` and `name` values.

Each rendered control needs a different `name` and `id` because this is how the browser uniquely identifies each control on the client-side and how it identifies to the web server what action or change has occurred on postback. For example, imagine that we wanted to run some server-side code whenever a RadioButton s checked state was changed. We could accomplish this by setting the RadioButton s `AutoPostBack` property to `True` and creating an event handler for the `CheckChanged` event. However, if the rendered `name` and `id` values for all of the radio buttons were the same, on postback we could not determine what specific RadioButton was clicked.

The short of it is that we cannot create a column of radio buttons in a GridView using the RadioButton Web control. Instead, we must use rather archaic techniques to ensure that the appropriate markup is injected into each GridView row.

> **NOTE**
> Like the RadioButton Web control, the radio button HTML control, when added to a template, will include the unique `name` attribute, making the radio buttons in the grid ungrouped. If you are not familiar with HTML controls, feel free to disregard this note, as HTML controls are rarely used, especially in ASP.NET 2.0. But if you are interested in learning more, see K. Scott Allen s blog entry Web Controls and HTML Controls.

Using a Literal Control to Inject Radio Button Markup

In order to correctly group all of the radio buttons within the GridView, we need to manually inject the radio buttons markup into the `ItemTemplate`. Each radio button needs the same `name` attribute, but should have a unique `id` attribute (in case we want to access a radio button via client-side script). After a user selects a radio button and posts back the page, the browser will send back the value of the selected radio button s `value` attribute. Therefore, each radio button will need a unique `value` attribute. Finally, on postback we need to make sure to add the `checked` attribute to the one radio button that is selected, otherwise after the user makes a selection and posts back, the radio buttons will return to their default state (all unselected).

There are two approaches that can be taken in order to inject low-level markup into a template. One is to do a mix of markup and calls to formatting methods defined in the code-behind class. This technique was first discussed in the Using TemplateFields in the GridView Control tutorial. In our case it might look something like:

```
<input type="radio" id='<%# GetUniqueRadioButtonID(...) %>'
    name='SuppliersGroup' value='<%# GetRadioButtonValue(...) %>' ... />
```

Here, `GetUniqueRadioButton` and `GetRadioButtonValue` would be methods defined in the code-behind class that returned the appropriate `id` and `value` attribute values for each radio button. This approach works well for assigning the `id` and `value` attributes, but falls short when needing to specify the `checked` attribute value because the databinding syntax is only executed when data is first bound to the GridView. Therefore, if the GridView has view state enabled, the formatting methods will only fire when the page is first loaded (or when the GridView is explicitly rebound to the data source), and therefore the function that sets the `checked` attribute won't be called on postback. It s a rather subtle problem and a bit beyond the scope of this article, so I'll leave it at this. I do, however, encourage you to try using the above approach and work it through to the point where you'll get stuck. While such an exercise won't get you any closer to a working version, it will help foster a deeper understanding of the GridView and the databinding lifecycle.

The other approach to injecting custom, low-level markup in a template and the approach that we'll be using for this tutorial is to add a Literal control to the template. Then, in the GridView s `RowCreated` or `RowDataBound` event handler, the Literal control can be programmatically accessed and its `Text` property set to the markup to emit.

Start by removing the RadioButton from the TemplateField s `ItemTemplate`, replacing it with a Literal control. Set the Literal control s `ID` to `RadioButtonMarkup`.

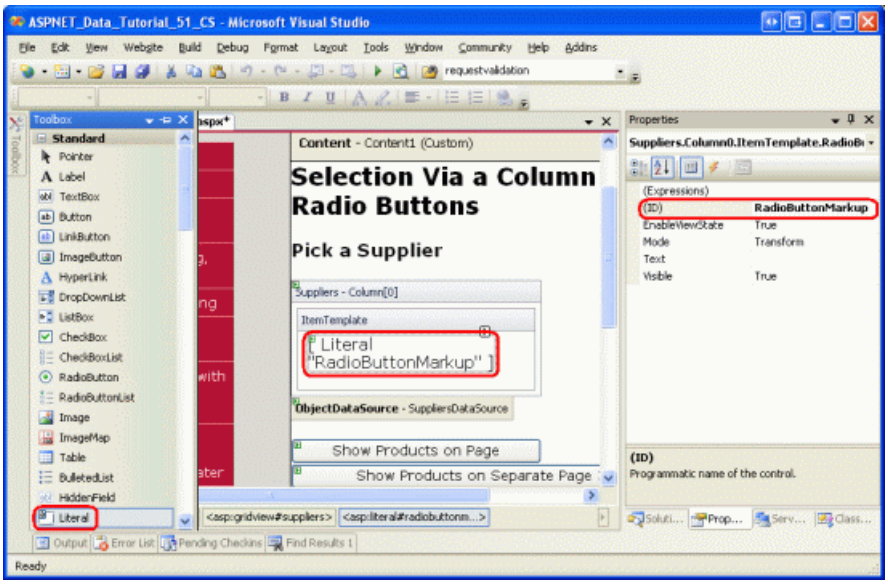


Figure 12: Add a Literal Control to the `ItemTemplate` (Click to view full-size image)

Next, create an event handler for the GridView s `RowCreated` event. The `RowCreated` event fires once for every row added, whether or not the data is being rebound to the GridView. That means that even on a postback when the data is reloaded from view state, the `RowCreated` event still fires and this is the reason we are using it instead of `RowDataBound` (which fires only when the data is explicitly bound to the data Web control).

In this event handler, we only want to proceed if we re dealing with a data row. For each data row we want to programmatically reference the `RadioButtonMarkup` Literal control and set its `Text` property to the markup to emit. As the following code shows, the markup emitted creates a radio button whose `name` attribute is set to `SuppliersGroup`, whose `id` attribute is set to `RowSelectorX`, where *X* is the index of the GridView row, and whose `value` attribute is set to the index of the GridView row.

```
Protected Sub Suppliers_RowCreated(sender As Object, e As GridViewRowEventArgs) _
    Handles Suppliers.RowCreated

    If e.Row.RowType = DataControlRowType.DataRow Then
        ' Grab a reference to the Literal control
        Dim output As Literal = _
            CType(e.Row.FindControl("RadioButtonMarkup"), Literal)
        ' Output the markup except for the "checked" attribute
        output.Text = String.Format( _
            "<input type="radio" name="SuppliersGroup" " & _
            "id="RowSelector{0}" value="{0}" />", e.Row.RowIndex)
    End If
End Sub
```

When a GridView row is selected and a postback occurs, we are interested in the `SupplierID` of the selected supplier. Therefore, one might think that the value of each radio button should be the actual `SupplierID` (rather than the index of the GridView row). While this may work in certain circumstances, it would be a security risk to blindly accept and process a `SupplierID`. Our GridView, for example, lists only those suppliers in the USA. However, if the `SupplierID` is passed directly from the radio button, what s to stop a mischievous user from manipulating the `SupplierID` value sent back on postback? By using the row index as the `value`, and then getting the `SupplierID` on postback from the `DataKeys` collection, we can ensure that the user is only using one of the `SupplierID` values associated with one of the GridView rows.

After adding this event handler code, take a minute to test out the page in a browser. First, note that only one radio button in the grid can be selected at a time. However, when selecting a radio button and clicking one of the buttons, a postback occurs and the radio buttons all revert to their initial state (that is, on postback, the selected radio button is no longer selected). To fix this, we need to augment the `RowCreated` event handler so that it

inspects the selected radio button index sent from the postback and adds the `checked="checked"` attribute to the emitted markup of the row index matches.

When a postback occurs, the browser sends back the `name` and `value` of the selected radio button. The value can be programmatically retrieved using `Request.Form("name")`. The `Request.Form` property provides a `NameValueCollection` representing the form variables. The form variables are the names and values of the form fields in the web page, and are sent back by the web browser whenever a postback ensues. Because the rendered `name` attribute of the radio buttons in the GridView is `SuppliersGroup`, when the web page is posted back the browser will send `SuppliersGroup=valueOfSelectedRadioButton` back to the web server (along with the other form fields). This information can then be accessed from the `Request.Form` property using: `Request.Form("SuppliersGroup")`.

Since we'll need to determine the selected radio button index not only in the `RowCreated` event handler, but in the `Click` event handlers for the Button Web controls, let s add a `SuppliersSelectedIndex` property to the code-behind class that returns `-1` if no radio button was selected and the selected index if one of the radio buttons is selected.

```
Private ReadOnly Property SuppliersSelectedIndex() As Integer
    Get
        If String.IsNullOrEmpty(Request.Form("SuppliersGroup")) Then
            Return -1
        Else
            Return Convert.ToInt32(Request.Form("SuppliersGroup"))
        End If
    End Get
End Property
```

With this property added, we know to add the `checked="checked"` markup in the `RowCreated` event handler when `SuppliersSelectedIndex` equals `e.Row.RowIndex`. Update the event handler to include this logic:

```
Protected Sub Suppliers_RowCreated(sender As Object, e As GridViewRowEventArgs) _
    Handles Suppliers.RowCreated

    If e.Row.RowType = DataControlRowType.DataRow Then
        ' Grab a reference to the Literal control
        Dim output As Literal = _
            CType(e.Row.FindControl("RadioButtonMarkup"), Literal)
        ' Output the markup except for the "checked" attribute
        output.Text = String.Format( _
            "<input type="radio" name="SuppliersGroup" " & _
            "id="RowSelector{0}" value="{0}"", e.Row.RowIndex)
        ' See if we need to add the "checked" attribute
        If SuppliersSelectedIndex = e.Row.RowIndex Then
            output.Text &= " checked="checked""
        End If
        ' Add the closing tag
        output.Text &= " />"
    End If
End Sub
```

With this change, the selected radio button remains selected after a postback. Now that we have the ability to specify what radio button is selected, we could change the behavior so that when the page was first visited, the first GridView row s radio button was selected (rather than having no radio buttons selected by default, which is the current behavior). To have the first radio button selected by default, simply change the `If SuppliersSelectedIndex = e.Row.RowIndex Then` statement to the following: `If SuppliersSelectedIndex = e.Row.RowIndex OrElse (Not Page.IsPostBack AndAlso e.Row.RowIndex = 0) Then`.

At this point we have added a column of grouped radio buttons to the GridView that allows for a single GridView

row to be selected and remembered across postbacks. Our next steps are to display the products provided by the selected supplier. In Step 4 we'll see how to redirect the user to another page, sending along the selected `SupplierID`. In Step 5, we'll see how to display the selected supplier s products in a GridView on the same page.

> **NOTE**
> Rather than using a TemplateField (the focus of this lengthy Step 3), we could create a custom `DataControlField` class that renders the appropriate user interface and functionality. The `DataControlField` class is the base class from which the BoundField, CheckBoxField, TemplateField, and other built-in GridView and DetailsView fields derive. Creating a custom `DataControlField` class would mean that the column of radio buttons could be added just using declarative syntax, and would also make replicating the functionality on other web pages and other web applications significantly easier.

If you ve ever created custom, compiled controls in ASP.NET, however, you know that doing so requires a fair amount of legwork and carries with it a host of subtleties and edge cases that must be carefully handled. Therefore, we will forgo implementing a column of radio buttons as a custom `DataControlField` class for now and stick with the TemplateField option. Perhaps we'll have the chance to explore creating, using, and deploying custom `DataControlField` classes in a future tutorial!

Step 4: Displaying the Selected Supplier s Products in a Separate Page

After the user has selected a GridView row, we need to show the selected supplier s products. In some circumstances, we may want to display these products in a separate page, in others we might prefer to do it in the same page. Let s first examine how to display the products in a separate page; in Step 5 we'll look at adding a GridView to `RadioButtonField.aspx` to display the selected supplier s products.

Currently there are two Button Web controls on the page `ListProducts` and `SendToProducts`. When the `SendToProducts` Button is clicked, we want to send the user to `~/Filtering/ProductsForSupplierDetails.aspx`. This page was created in the Master/Detail Filtering Across Two Pages tutorial and displays the products for the supplier whose `SupplierID` is passed through the querystring field named `SupplierID`.

To provide this functionality, create an event handler for the `SendToProducts` Button s `Click` event. In Step 3 we added the `SuppliersSelectedIndex` property, which returns the index of the row whose radio button is selected. The corresponding `SupplierID` can be retrieved from the GridView s `DataKeys` collection and the user can then be sent to `~/Filtering/ProductsForSupplierDetails.aspx?SupplierID=SupplierID` using `Response.Redirect("url")`.

```
Protected Sub SendToProducts_Click(sender As Object, e As EventArgs) _
    Handles SendToProducts.Click

    ' Send the user to ~/Filtering/ProductsForSupplierDetails.aspx
    Dim supplierID As Integer = _
        Convert.ToInt32(Suppliers.DataKeys(SuppliersSelectedIndex).Value)
    Response.Redirect( _
        "~/Filtering/ProductsForSupplierDetails.aspx?SupplierID=" & _
        supplierID)
End Sub
```

This code works wonderfully as long as one of the radio buttons is selected from the GridView. If, initially, the GridView does not have any radio buttons selected, and the user clicks the `SendToProducts` button, `SuppliersSelectedIndex` will be `-1`, which will cause an exception to be thrown since `-1` is out of the index range of the `DataKeys` collection. This is not a concern, however, if you decided to update the `RowCreated` event handler as discussed in Step 3 so as to have the first radio button in the GridView initially selected.

To accommodate a `SuppliersSelectedIndex` value of `-1`, add a Label Web control to the page above the GridView. Set its `ID` property to `ChooseSupplierMsg`, its `CssClass` property to `Warning`, its `EnableViewState` and `Visible` properties to `False`, and its `Text` property to Please choose a supplier from the grid. The CSS class `Warning`

displays text in a red, italic, bold, large font and is defined in `Styles.css`. By setting the `EnableViewState` and `Visible` properties to `False`, the Label is not rendered except for only those postbacks where the control s `Visible` property is programmatically set to `True`.

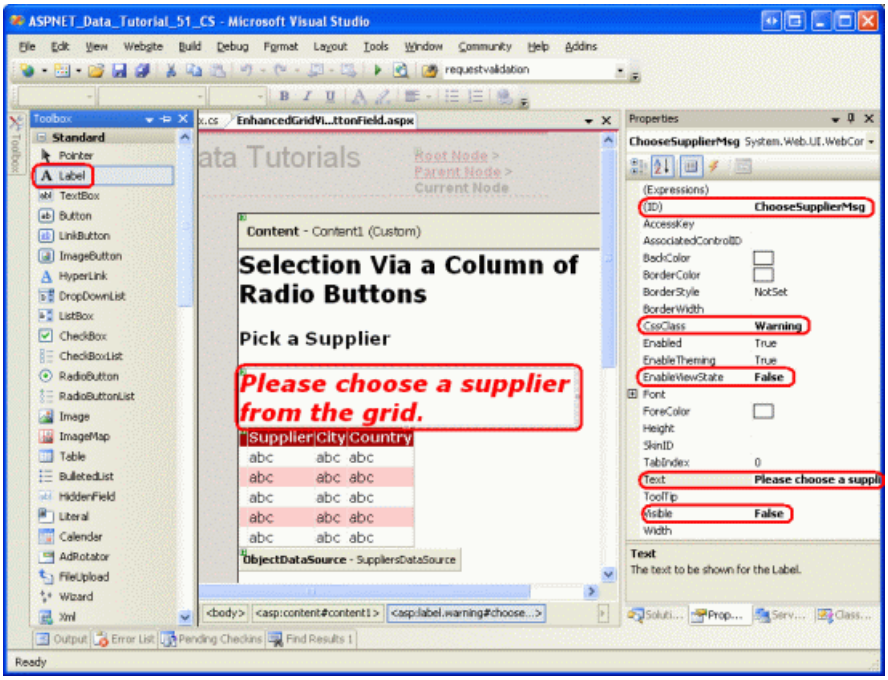


Figure 13: Add a Label Web Control Above the GridView (Click to view full-size image)

Next, augment the `Click` event handler to display the `ChooseSupplierMsg` Label if `SuppliersSelectedIndex` is less than zero, and redirect the user to `~/Filtering/ProductsForSupplierDetails.aspx?SupplierID=SupplierID` otherwise.

```
Protected Sub SendToProducts_Click(sender As Object, e As EventArgs) _
    Handles SendToProducts.Click

    ' make sure one of the radio buttons has been selected
    If SuppliersSelectedIndex < 0 Then
        ChooseSupplierMsg.Visible = True
    Else
        ' Send the user to ~/Filtering/ProductsForSupplierDetails.aspx
        Dim supplierID As Integer = _
            Convert.ToInt32(Suppliers.DataKeys(SuppliersSelectedIndex).Value)
        Response.Redirect( _
            "~/Filtering/ProductsForSupplierDetails.aspx?SupplierID=" & _
            supplierID)
    End If
End Sub
```

Visit the page in a browser and click the `SendToProducts` button before selecting a supplier from the GridView. As Figure 14 shows, this displays the `ChooseSupplierMsg` label. Next, select a supplier and click the `SendToProducts` button. This will whisk you to a page that lists the products supplied by the selected supplier. Figure 15 shows the `ProductsForSupplierDetails.aspx` page when the Bigfoot Breweries supplier was selected.

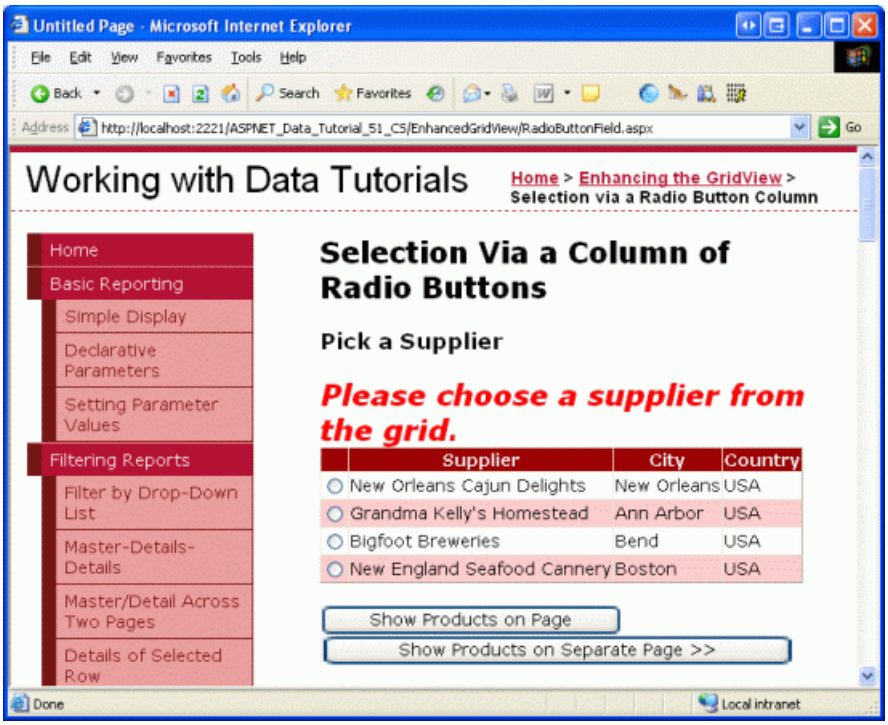


Figure 14: The `ChooseSupplierMsg` Label is Displayed if No Supplier is Selected (Click to view full-size image)

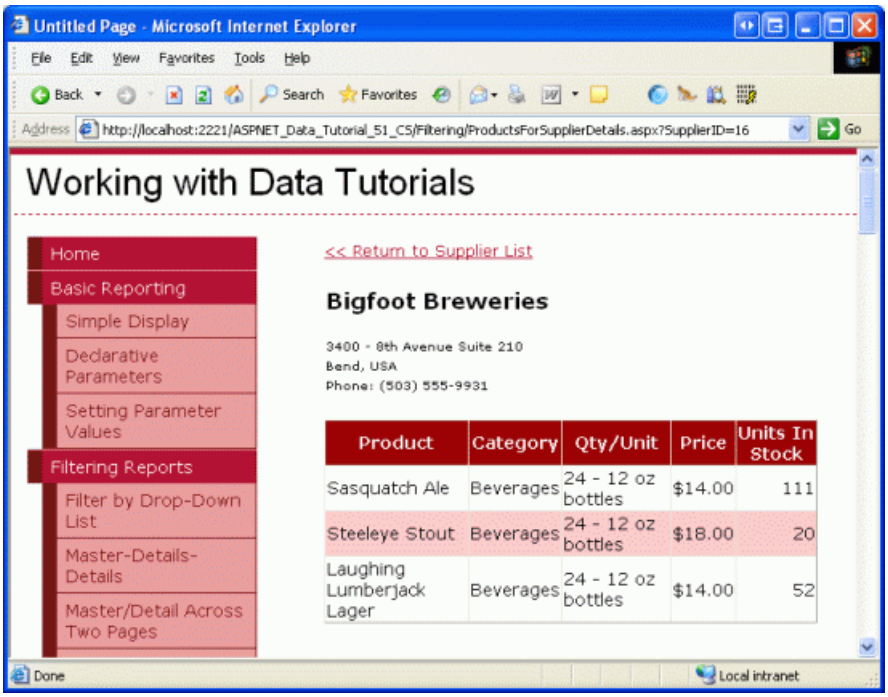


Figure 15: The Selected Supplier s Products are Displayed in `ProductsForSupplierDetails.aspx` (Click to view full-size image)

Step 5: Displaying the Selected Supplier s Products on the Same Page

In Step 4 we saw how to send the user to another web page to display the selected supplier s products. Alternatively, the selected supplier s products can be displayed on the same page. To illustrate this, we'll add another GridView to `RadioButtonField.aspx` to display the selected supplier s products.

Since we only want this GridView of products to display once a supplier has been selected, add a Panel Web control beneath the `Suppliers` GridView, setting its `ID` to `ProductsBySupplierPanel` and its `Visible` property to `False`. Within the Panel, add the text Products for the Selected Supplier, followed by a GridView named `ProductsBySupplier`. From the GridView s smart tag, choose to bind it to a new ObjectDataSource named `ProductsBySupplierDataSource`.

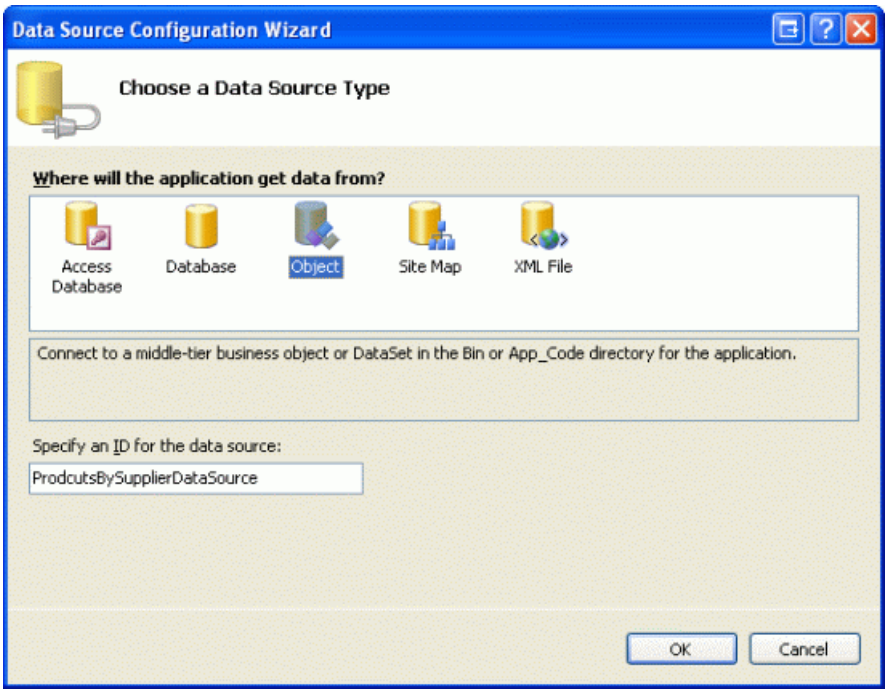


Figure 16: Bind the `ProductsBySupplier` GridView to a New ObjectDataSource (Click to view full-size image)

Next, configure the ObjectDataSource to use the `ProductsBLL` class. Since we only want to retrieve those products provided by the selected supplier, specify that the ObjectDataSource should invoke the `GetProductsBySupplierID(supplierID)` method to retrieve its data. Select (None) from the drop-down lists in the UPDATE, INSERT, and DELETE tabs.

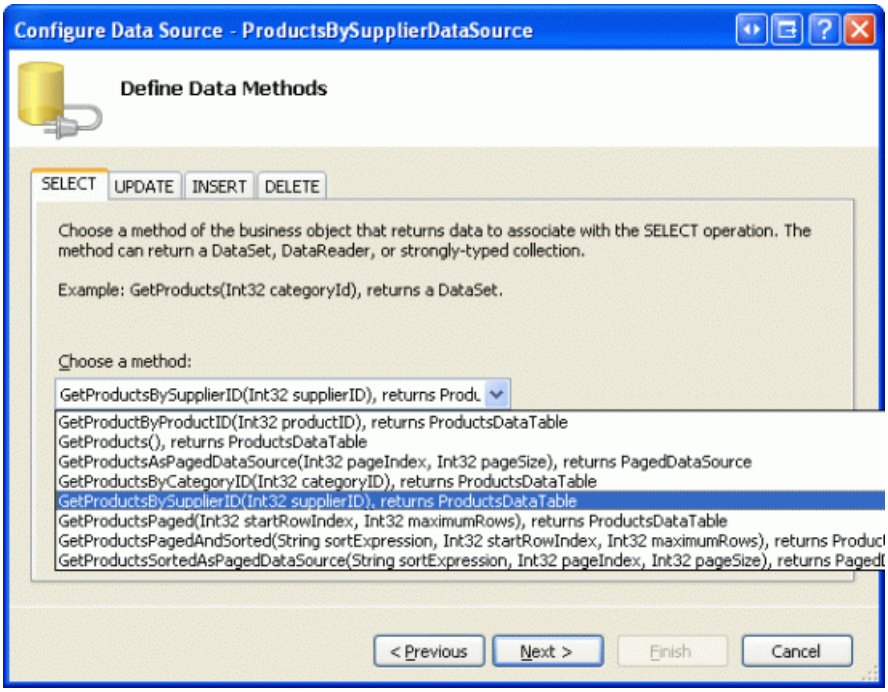


Figure 17: Configure the ObjectDataSource to Use the `GetProductsBySupplierID(supplierID)` Method (Click to view full-size image)

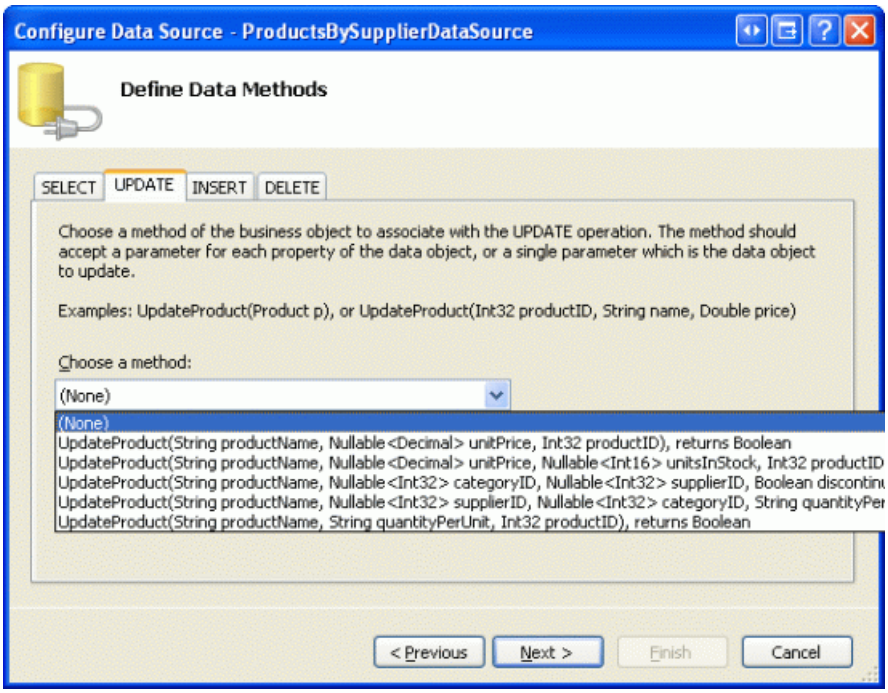


Figure 18: Set the Drop-Down Lists to (None) in the UPDATE, INSERT, and DELETE Tabs (Click to view full-size image)

After configuring the SELECT, UPDATE, INSERT, and DELETE tabs, click Next. Since the `GetProductsBySupplierID(supplierID)` method expects an input parameter, the Create Data Source wizard prompts us to specify the source for the parameter s value.

We have a couple of options here in specifying the source of the parameter s value. We could use the default Parameter object, and programmatically assign the value of the `SuppliersSelectedIndex` property to the Parameter s `DefaultValue` property in the ObjectDataSource s `Selecting` event handler. Refer back to the Programmatically Setting the ObjectDataSource's Parameter Values tutorial for a refresher on programmatically assigning values to the ObjectDataSource s parameters.

Alternatively, we can use a ControlParameter and refer to the `Suppliers` GridView s `SelectedValue` property (see Figure 19). The GridView s `SelectedValue` property returns the `DataKey` value corresponding to the `SelectedIndex` property. In order for this option to work, we need to programmatically set the GridView s `SelectedIndex` property to the selected row when the `ListProducts` button is clicked. As an added benefit, by setting the `SelectedIndex`, the selected record will take on the `SelectedRowStyle` defined in the `DataWebControls` Theme (a yellow background).

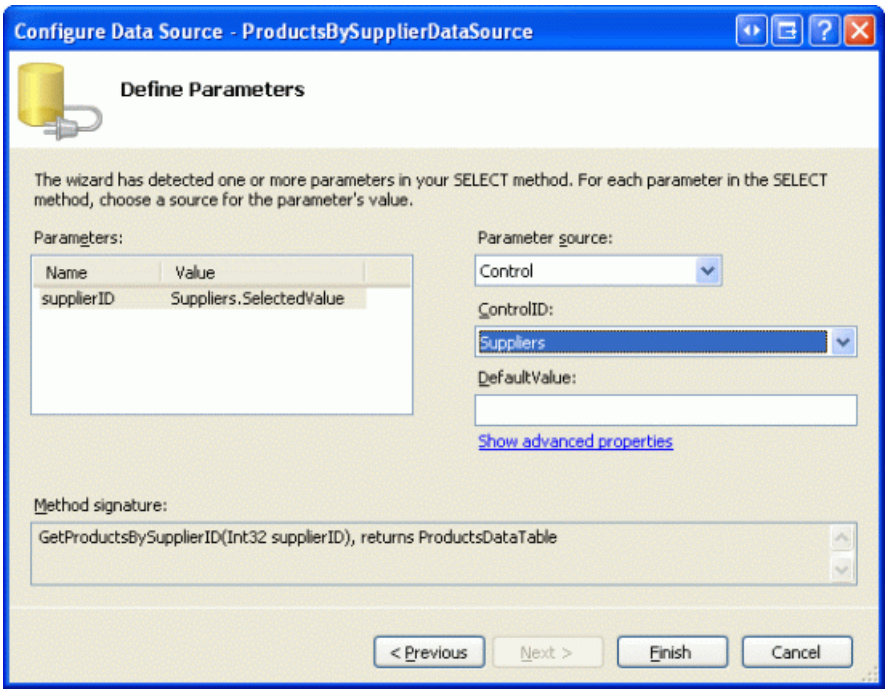


Figure 19: Use a ControlParameter to Specify the GridView s SelectedValue as the Parameter Source (Click to view full-size image)

Upon completing the wizard, Visual Studio will automatically add fields for the product s data fields. Remove all but the `ProductName`, `CategoryName`, and `UnitPrice` BoundFields, and change the `HeaderText` properties to Product, Category, and Price. Configure the `UnitPrice` BoundField so that its value is formatted as a currency. After making these changes, the Panel, GridView, and ObjectDataSource s declarative markup should look like the following:

```
<asp:Panel runat="server" ID="ProductsBySupplierPanel" Visible="False">
    <h3>
        Products for the Selected Supplier</h3>
    <p>
        <asp:GridView ID="ProductsBySupplier" runat="server"
            AutoGenerateColumns="False" DataKeyNames="ProductID"
            DataSourceID="ProductsBySupplierDataSource" EnableViewState="False">
            <Columns>
                <asp:BoundField DataField="ProductName" HeaderText="Product"
                    SortExpression="ProductName" />
                <asp:BoundField DataField="CategoryName" HeaderText="Category"
                    ReadOnly="True" SortExpression="CategoryName" />
                <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
                    HeaderText="Price" HtmlEncode="False"
                    SortExpression="UnitPrice" />
            </Columns>
        </asp:GridView>
        <asp:ObjectDataSource ID="ProductsBySupplierDataSource" runat="server"
            OldValuesParameterFormatString="original_{0}"
            SelectMethod="GetProductsBySupplierID" TypeName="ProductsBLL">
            <SelectParameters>
                <asp:ControlParameter ControlID="Suppliers" Name="supplierID"
                    PropertyName="SelectedValue" Type="Int32" />
            </SelectParameters>
        </asp:ObjectDataSource>
    </p>
</asp:Panel>
```

To complete this exercise, we need to set the GridView s `SelectedIndex` property to the `SelectedSuppliersIndex` and the `ProductsBySupplierPanel` Panel s `Visible` property to `True` when the `ListProducts` button is clicked. To accomplish this, create an event handler for the `ListProducts` Button Web control s `Click` event and add the

following code:

```
Protected Sub ListProducts_Click(sender As Object, e As EventArgs) _
    Handles ListProducts.Click

    ' make sure one of the radio buttons has been selected
    If SuppliersSelectedIndex < 0 Then
        ChooseSupplierMsg.Visible = True
        ProductsBySupplierPanel.Visible = False
    Else
        ' Set the GridView's SelectedIndex
        Suppliers.SelectedIndex = SuppliersSelectedIndex
        ' Show the ProductsBySupplierPanel panel
        ProductsBySupplierPanel.Visible = True
    End If
End Sub
```

If a supplier has not been selected from the GridView, the `ChooseSupplierMsg` Label is displayed and the `ProductsBySupplierPanel` Panel hidden. Otherwise, if a supplier has been selected, the `ProductsBySupplierPanel` is displayed and the GridView s `SelectedIndex` property is updated.

Figure 20 shows the results after the Bigfoot Breweries supplier has been selected and the Show Products on Page button has been clicked.

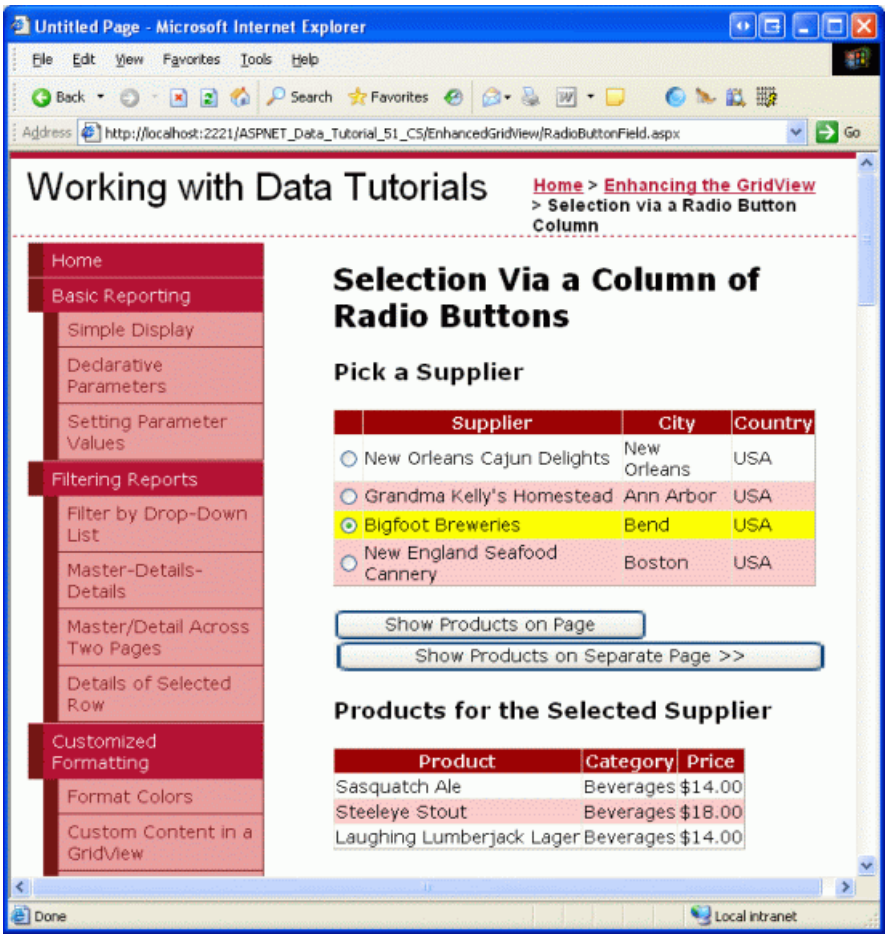


Figure 20: The Products Supplied by Bigfoot Breweries are Listed on the Same Page (Click to view full-size image)

Summary

As discussed in the Master/Detail Using a Selectable Master GridView with a Details DetailView tutorial, records can be selected from a GridView using a CommandField whose `ShowSelectButton` property is set to `True`. But the CommandField displays its buttons either as regular push buttons, links, or images. An alternative row-selection

user interface is to provide a radio button or checkbox in each GridView row. In this tutorial we examined how to add a column of radio buttons.

Unfortunately, adding a column of radio buttons isn t as straightforward or simple as one might expect. There is no built-in RadioButtonField that can be added at the click of a button, and using the RadioButton Web control within a TemplateField introduces its own set of problems. In the end, to provide such an interface we either have to create a custom `DataControlField` class or resort to injecting the appropriate HTML into a TemplateField during the `RowCreated` event.

Having explored how to add a column of radio buttons, let us turn our attention to adding a column of checkboxes. With a column of checkboxes, a user can select one or more GridView rows and then perform some operation on all of the selected rows (such as selecting a set of emails from a web-based email client, and then choosing to delete all selected emails). In the next tutorial we'll see how to add such a column.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was David Suru. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Adding a GridView Column of Checkboxes (VB)

2/20/2020 • 9 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> This tutorial looks at how to add a column of check boxes to a GridView control to provide the user with an intuitive way of selecting multiple rows of the GridView.

Introduction

In the preceding tutorial we examined how to add a column of radio buttons to the GridView for the purpose of selecting a particular record. A column of radio buttons is a suitable user interface when the user is limited to choosing at most one item from the grid. At times, however, we may want to allow the user to pick an arbitrary number of items from the grid. Web-based email clients, for example, typically display the list of messages with a column of checkboxes. The user can select an arbitrary number of messages and then perform some action, such as moving the emails to another folder or deleting them.

In this tutorial we will see how to add a column of checkboxes and how to determine what checkboxes were checked on postback. In particular, we'll build an example that closely mimics the web-based email client user interface. Our example will include a paged GridView listing the products in the `Products` database table with a checkbox in each row (see Figure 1). A Delete Selected Products button, when clicked, will delete those products selected.

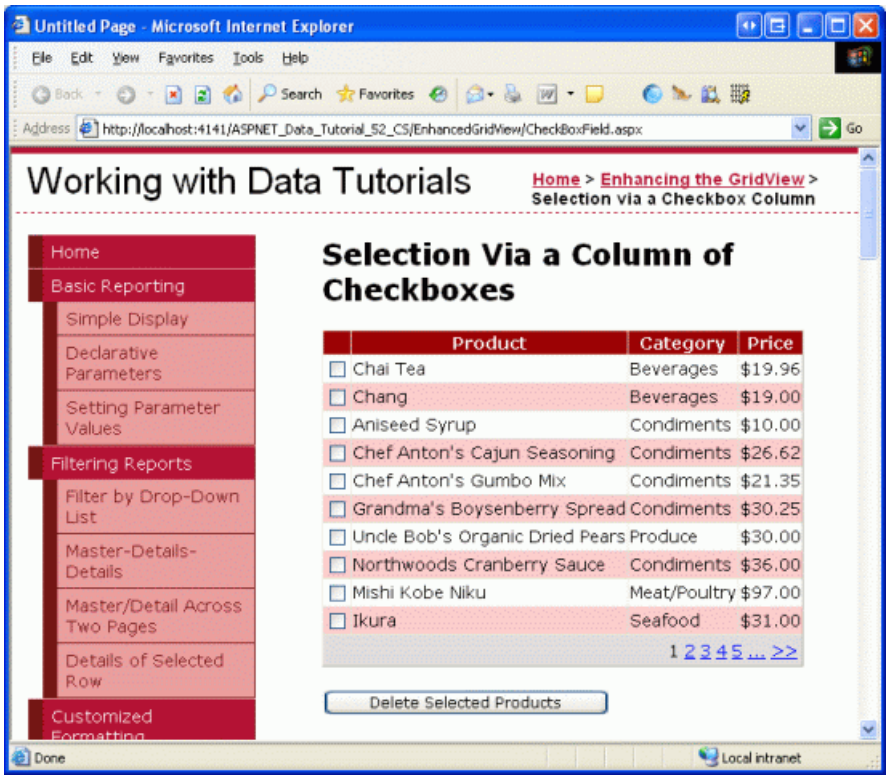


Figure 1: Each Product Row Includes a Checkbox (Click to view full-size image)

Step 1: Adding a Paged GridView that Lists Product Information

Before we worry about adding a column of checkboxes, let s first focus on listing the products in a GridView that

supports paging. Start by opening the `CheckBoxField.aspx` page in the `EnhancedGridView` folder and drag a GridView from the Toolbox onto the Designer, setting its `ID` to `Products`. Next, choose to bind the GridView to a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to use the `ProductsBLL` class, calling the `GetProducts()` method to return the data. Since this GridView will be read-only, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) .

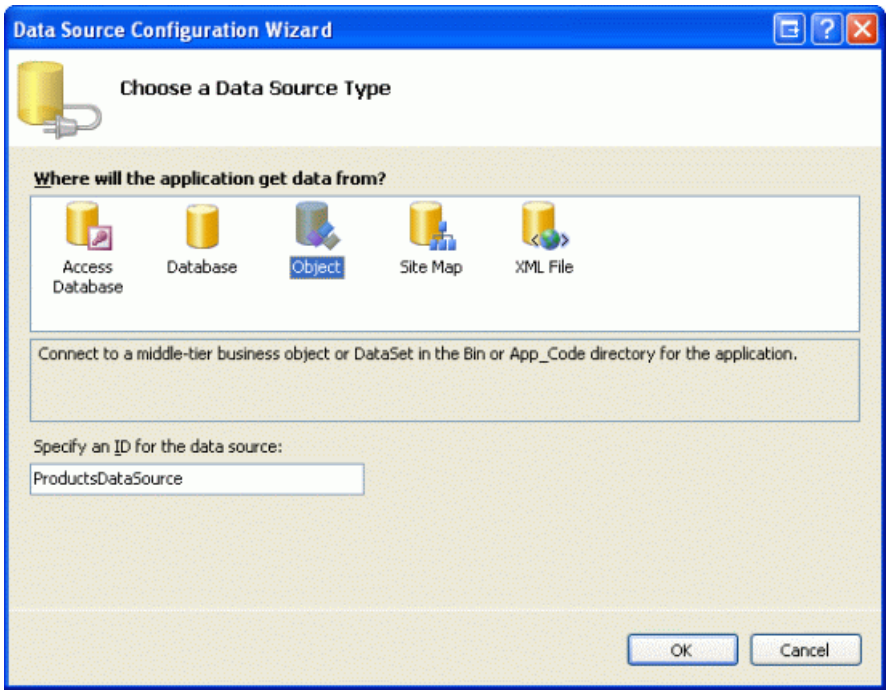


Figure 2: Create a New ObjectDataSource Named `ProductsDataSource` (Click to view full-size image)

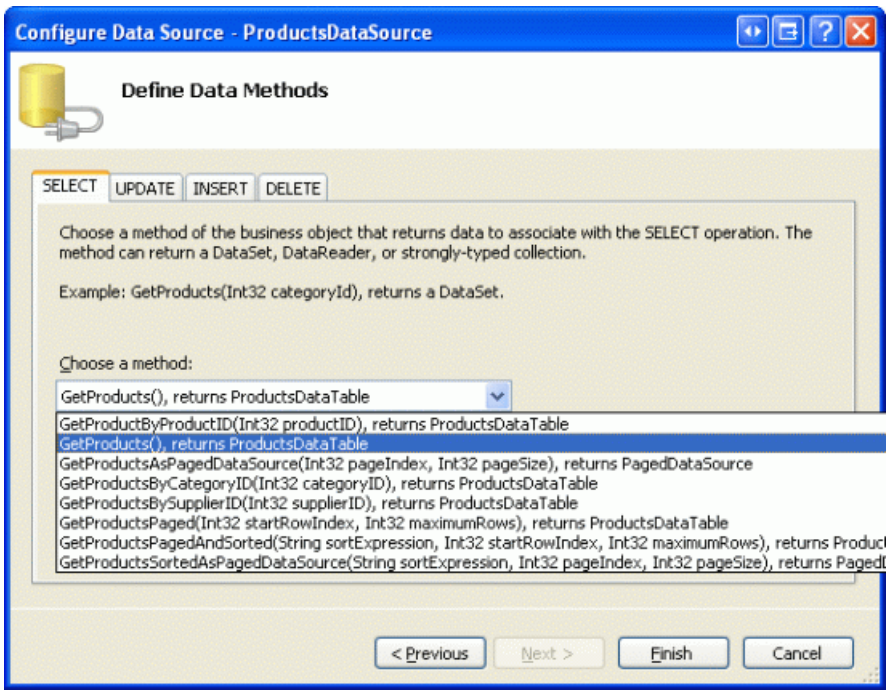


Figure 3: Configure the ObjectDataSource to Retrieve Data Using the `GetProducts()` Method (Click to view full-size image)

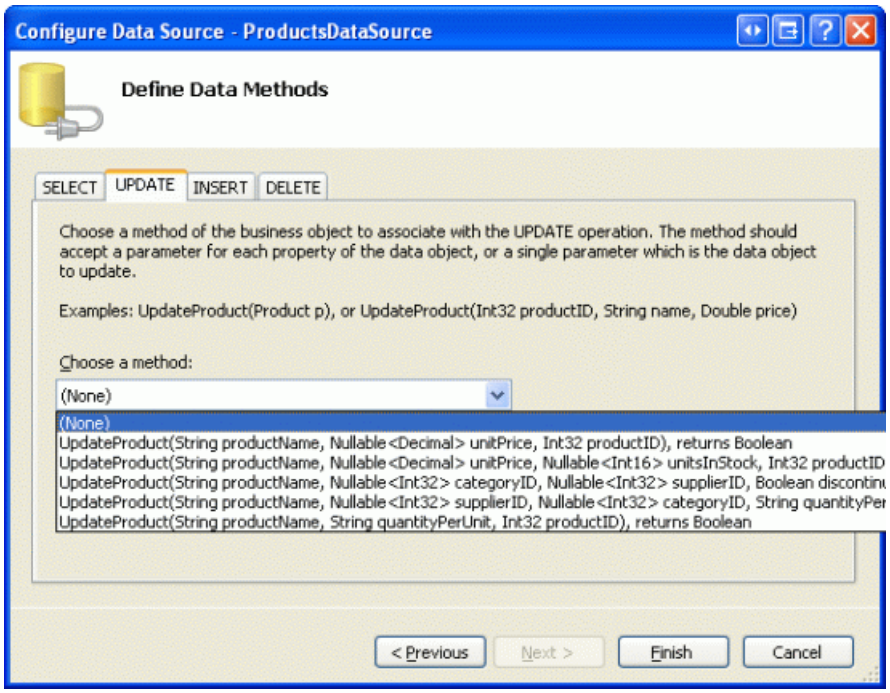


Figure 4: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE Tabs to (None) (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will automatically create BoundColumns and a CheckBoxColumn for the product-related data fields. Like we did in the previous tutorial, remove all but the `ProductName`, `CategoryName`, and `UnitPrice` BoundFields, and change the `HeaderText` properties to Product, Category, and Price. Configure the `UnitPrice` BoundField so that its value is formatted as a currency. Also configure the GridView to support paging by checking the Enable Paging checkbox from the smart tag.

Let s also add the user interface for deleting the selected products. Add a Button Web control beneath the GridView, setting its `ID` to `DeleteSelectedProducts` and its `Text` property to Delete Selected Products. Rather than actually deleting products from the database, for this example we'll just display a message stating the products that would have been deleted. To accommodate this, add a Label Web control beneath the Button. Set its ID to `DeleteResults`, clear out its `Text` property, and set its `Visible` and `EnableViewState` properties to `False`.

After making these changes, the GridView, ObjectDataSource, Button, and Label s declarative markup should similar to the following:

```
<p>
    <asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
        DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
        AllowPaging="True" EnableViewState="False">
        <Columns>
            <asp:BoundField DataField="ProductName" HeaderText="Product"
                SortExpression="ProductName" />
            <asp:BoundField DataField="CategoryName" HeaderText="Category"
                ReadOnly="True" SortExpression="CategoryName" />
            <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
                HeaderText="Price" HtmlEncode="False"
                SortExpression="UnitPrice" />
        </Columns>
    </asp:GridView>
    <asp:ObjectDataSource ID="ProductsDataSource" runat="server"
        OldValuesParameterFormatString="original_{0}"
        SelectMethod="GetProducts" TypeName="ProductsBLL">
    </asp:ObjectDataSource>
</p>
<p>
    <asp:Button ID="DeleteSelectedProducts" runat="server"
        Text="Delete Selected Products" />
</p>
<p>
    <asp:Label ID="DeleteResults" runat="server" EnableViewState="False"
        Visible="False"></asp:Label>
</p>
```

Take a moment to view the page in a browser (see Figure 5). At this point you should see the name, category, and price of the first ten products.

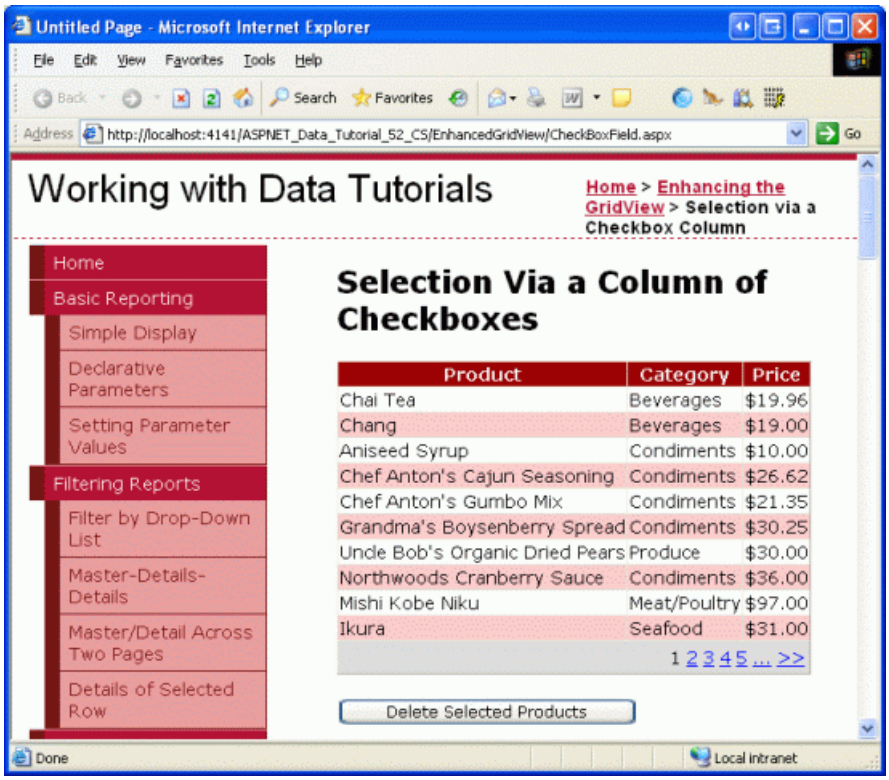


Figure 5: The Name, Category, and Price of the First Ten Products are Listed (Click to view full-size image)

Step 2: Adding a Column of Checkboxes

Since ASP.NET 2.0 includes a CheckBoxField, one might think that it could be used to add a column of checkboxes to a GridView. Unfortunately, that is not the case, as the CheckBoxField is designed to work with a Boolean data field. That is, in order to use the CheckBoxField we must specify the underlying data field whose value is consulted

to determine whether the rendered checkbox is checked. We cannot use the CheckBoxField to just include a column of unchecked checkboxes.

Instead, we must add a TemplateField and add a CheckBox Web control to its `ItemTemplate`. Go ahead and add a TemplateField to the `Products` GridView and make it the first (far-left) field. From the GridView s smart tag, click on the Edit Templates link and then drag a CheckBox Web control from the Toolbox into the `ItemTemplate`. Set this CheckBox s `ID` property to `ProductSelector`.

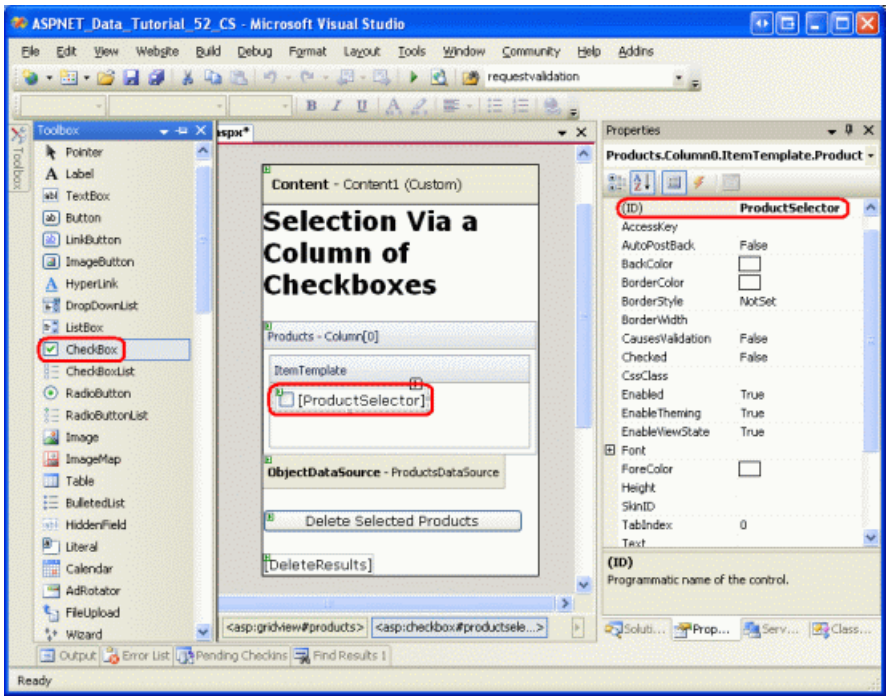


Figure 6: Add a CheckBox Web Control Named `ProductSelector` to the TemplateField s `ItemTemplate` (Click to view full-size image)

With the TemplateField and CheckBox Web control added, each row now includes a checkbox. Figure 7 shows this page, when viewed through a browser, after the TemplateField and CheckBox have been added.

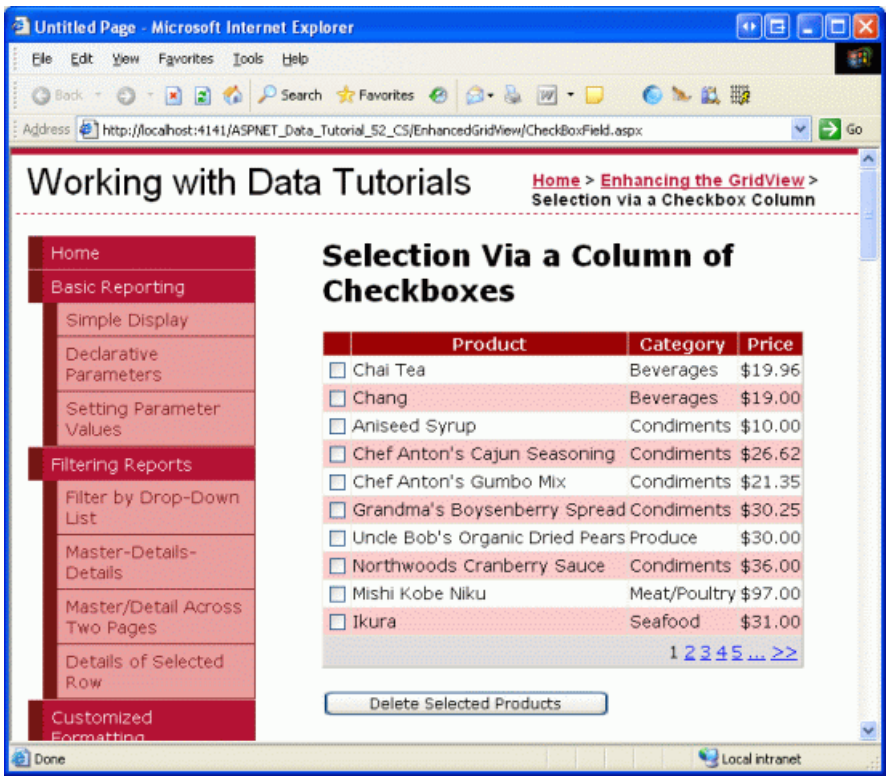


Figure 7: Each Product Row Now Includes a Checkbox (Click to view full-size image)

Step 3: Determining What Checkboxes Were Checked On Postback

At this point we have a column of checkboxes but no way to determine what checkboxes were checked on postback. When the Delete Selected Products button is clicked, though, we need to know what checkboxes were checked in order to delete those products.

The GridView s `Rows` property provides access to the data rows in the GridView. We can iterate through these rows, programmatically access the CheckBox control, and then consult its `Checked` property to determine whether the CheckBox has been selected.

Create an event handler for the `DeleteSelectedProducts` Button Web control s `Click` event and add the following code:

```
Protected Sub DeleteSelectedProducts_Click(sender As Object, e As EventArgs) _
    Handles DeleteSelectedProducts.Click

    Dim atLeastOneRowDeleted As Boolean = False
    ' Iterate through the Products.Rows property
    For Each row As GridViewRow In Products.Rows
        ' Access the CheckBox
        Dim cb As CheckBox = row.FindControl("ProductSelector")
        If cb IsNot Nothing AndAlso cb.Checked Then
            ' Delete row! (Well, not really...)
            atLeastOneRowDeleted = True
            ' First, get the ProductID for the selected row
            Dim productID As Integer = _
                Convert.ToInt32(Products.DataKeys(row.RowIndex).Value)
            ' "Delete" the row
            DeleteResults.Text &= String.Format( _
                "This would have deleted ProductID {0}<br />", productID)
            '... To actually delete the product, use ...
            ' Dim productAPI As New ProductsBLL
            ' productAPI.DeleteProduct(productID)
            '.............................................
        End If
    Next
    ' Show the Label if at least one row was deleted...
    DeleteResults.Visible = atLeastOneRowDeleted
End Sub
```

The `Rows` property returns a collection of `GridViewRow` instances that makeup the GridView s data rows. The `For Each` loop here enumerates this collection. For each `GridViewRow` object, the row s CheckBox is programmatically accessed using `row.FindControl("controlID")`. If the CheckBox is checked, the row s corresponding `ProductID` value is retrieved from the `DataKeys` collection. In this exercise, we simply display an informative message in the `DeleteResults` Label, although in a working application we d instead make a call to the `ProductsBLL` class s `DeleteProduct(productID)` method.

With the addition of this event handler, clicking the Delete Selected Products button now displays the `ProductID` s of the selected products.

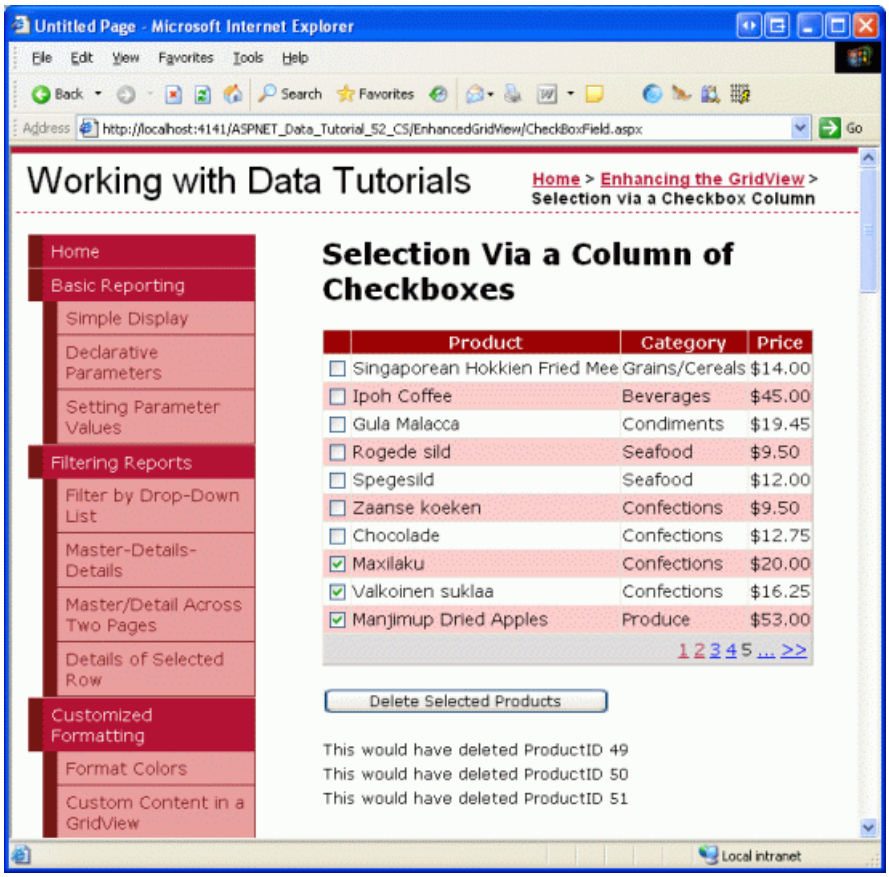


Figure 8: When the Delete Selected Products Button is Clicked the Selected Products `ProductID` s are Listed (Click to view full-size image)

Step 4: Adding Check All and Uncheck All Buttons

If a user wants to delete all products on the current page, they must check each of the ten checkboxes. We can help expedite this process by adding a Check All button that, when clicked, selects all of the checkboxes in the grid. An Uncheck All button would be equally helpful.

Add two Button Web controls to the page, placing them above the GridView. Set the first one s `ID` to `CheckAll` and its `Text` property to Check All ; set the second one s `ID` to `UncheckAll` and its `Text` property to Uncheck All .

```
<asp:Button ID="CheckAll" runat="server" Text="Check All" />

<asp:Button ID="UncheckAll" runat="server" Text="Uncheck All" />
```

Next, create a method in the code-behind class named `ToggleCheckState(checkState)` that, when invoked, enumerates the `Products` GridView s `Rows` collection and sets each CheckBox s `Checked` property to the value of the passed in *checkState* parameter.

```
Private Sub ToggleCheckState(ByVal checkState As Boolean)
    ' Iterate through the Products.Rows property
    For Each row As GridViewRow In Products.Rows
        ' Access the CheckBox
        Dim cb As CheckBox = row.FindControl("ProductSelector")
        If cb IsNot Nothing Then
            cb.Checked = checkState
        End If
    Next
End Sub
```

Next, create `Click` event handlers for the `CheckAll` and `UncheckAll` buttons. In `CheckAll` s event handler, simply call `ToggleCheckState(True)`; in `UncheckAll`, call `ToggleCheckState(False)`.

```
Protected Sub CheckAll_Click(sender As Object, e As EventArgs) _
    Handles CheckAll.Click

    ToggleCheckState(True)
End Sub
Protected Sub UncheckAll_Click(sender As Object, e As EventArgs) _
    Handles UncheckAll.Click

    ToggleCheckState(False)
End Sub
```

With this code, clicking the Check All button causes a postback and checks all of the checkboxes in the GridView. Likewise, clicking Uncheck All unselects all checkboxes. Figure 9 shows the screen after the Check All button has been checked.

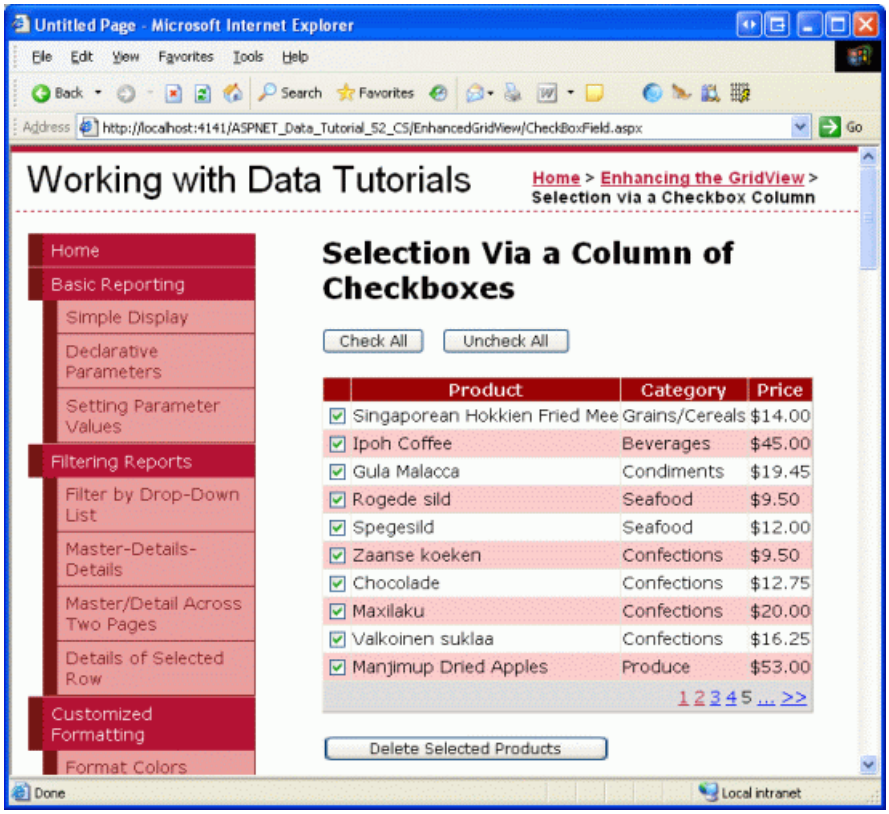


Figure 9: Clicking the Check All Button Selects All Checkboxes (Click to view full-size image)

> **NOTE**
>
> When displaying a column of checkboxes, one approach for selecting or unselecting all of the checkboxes is through a checkbox in the header row. Moreover, the current Check All / Uncheck All implementation requires a postback. The checkboxes can be checked or unchecked, however, entirely through client-side script, thereby providing a snappier user experience. To explore using a header row checkbox for Check All and Uncheck All in detail, along with a discussion on using client-side techniques, check out Checking All CheckBoxes in a GridView Using Client-Side Script and a Check All CheckBox.

Summary

In cases where you need to let users choose an arbitrary number of rows from a GridView before proceeding, adding a column of checkboxes is one option. As we saw in this tutorial, including a column of checkboxes in the GridView entails adding a TemplateField with a CheckBox Web control. By using a Web control (versus injecting

markup directly into the template, as we did in the previous tutorial) ASP.NET automatically remembers what CheckBoxes were and were not checked across postback. We can also programmatically access the CheckBoxes in code to determine whether a given CheckBox is checked, or to change the checked state.

This tutorial and the last one looked at adding a row selector column to the GridView. In our next tutorial we'll examine how, with a bit of work, we can add inserting capabilities to the GridView.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Inserting a New Record from the GridView's Footer (VB)

2/20/2020 • 25 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> While the GridView control does not provide built-in support for inserting a new record of data, this tutorial shows how to augment the GridView to include an inserting interface.

Introduction

As discussed in the An Overview of Inserting, Updating, and Deleting Data tutorial, the GridView, DetailsView, and FormView Web controls each include built-in data modification capabilities. When used with declarative data source controls, these three Web controls can be quickly and easily configured to modify data - and in scenarios without needing to write a single line of code. Unfortunately, only the DetailsView and FormView controls provide built-in inserting, editing, and deleting capabilities. The GridView only offers editing and deleting support. However, with a little elbow grease, we can augment the GridView to include an inserting interface.

In adding inserting capabilities to the GridView, we are responsible for deciding on how new records will be added, creating the inserting interface, and writing the code to insert the new record. In this tutorial we will look at adding the inserting interface to the GridView s footer row (see Figure 1). The footer cell for each column includes the appropriate data collection user interface element (a TextBox for the product s name, a DropDownList for the supplier, and so on). We also need a column for an Add button that, when clicked, will cause a postback and insert a new record into the `Products` table using the values supplied in the footer row.

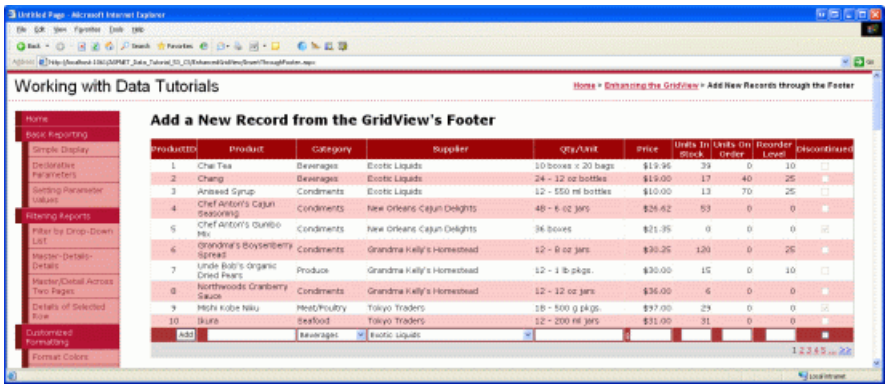


Figure 1: The Footer Row Provides an Interface for Adding New Products (Click to view full-size image)

Step 1: Displaying Product Information in a GridView

Before we concern ourselves with creating the inserting interface in the GridView s footer, let s first focus on adding a GridView to the page that lists the products in the database. Start by opening the `InsertThroughFooter.aspx` page in the `EnhancedGridView` folder and drag a GridView from the Toolbox onto the Designer, setting the GridView s `ID` property to `Products`. Next, use the GridView s smart tag to bind it to a new ObjectDataSource named `ProductsDataSource`.

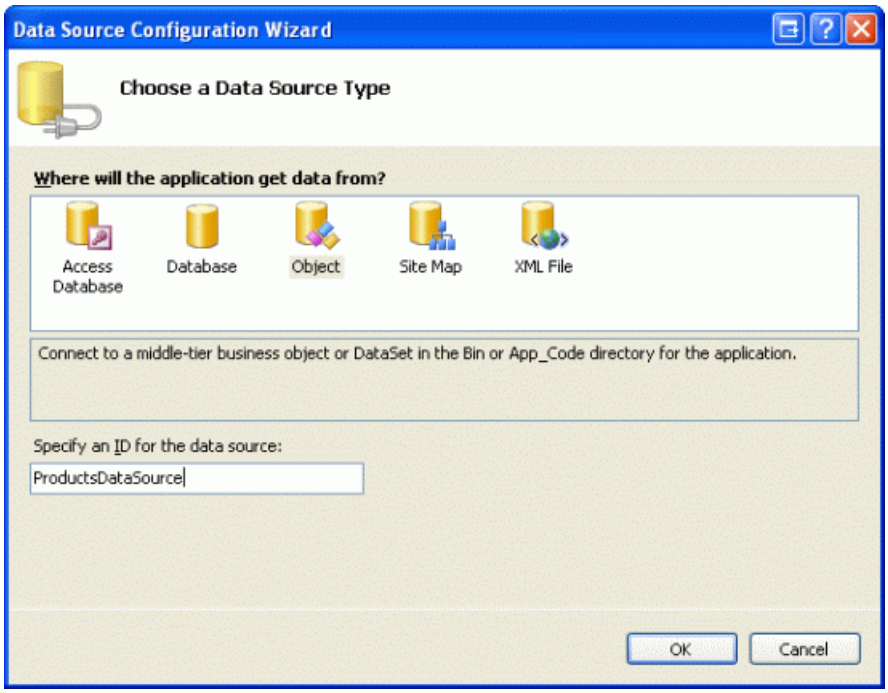


Figure 2: Create a New ObjectDataSource Named `ProductsDataSource` (Click to view full-size image)

Configure the ObjectDataSource to use the `ProductsBLL` class s `GetProducts()` method to retrieve product information. For this tutorial, let s focus strictly on adding inserting capabilities and not worry about editing and deleting. Therefore, make sure that the drop-down list in the INSERT tab is set to `AddProduct()` and that the drop-down lists in the UPDATE and DELETE tabs are set to (None) .

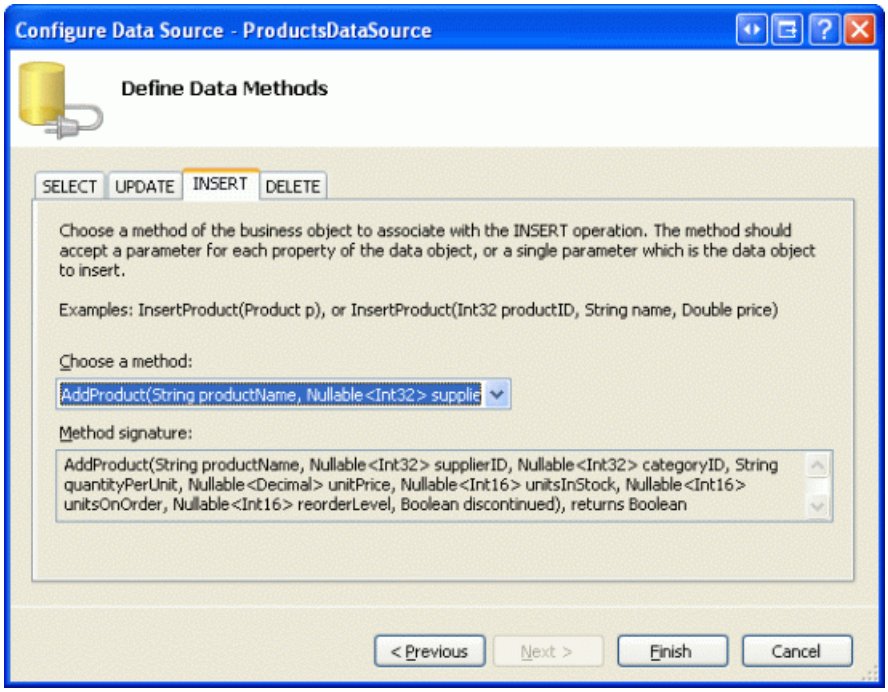


Figure 3: Map the `AddProduct` Method to the ObjectDataSource s `Insert()` Method (Click to view full-size image)

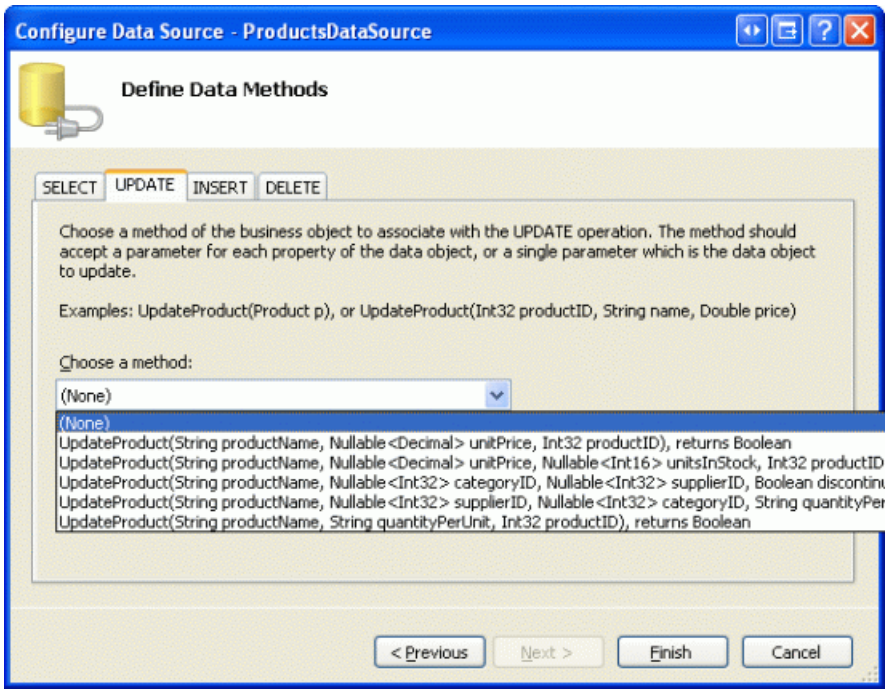


Figure 4: Set the UPDATE and DELETE Tabs Drop-Down Lists to (None) (Click to view full-size image)

After completing the ObjectDataSource s Configure Data Source wizard, Visual Studio will automatically add fields to the GridView for each of the corresponding data fields. For now, leave all of the fields added by Visual Studio. Later in this tutorial we'll come back and remove some of the fields whose values don t need to be specified when adding a new record.

Since there are close to 80 products in the database, a user will have to scroll all the way down to the bottom of the web page in order to add a new record. Therefore, let s enable paging to make the inserting interface more visible and accessible. To turn on paging, simply check the Enable Paging checkbox from the GridView s smart tag.

At this point, the GridView and ObjectDataSource s declarative markup should look similar to the following:

```
<asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
    AllowPaging="True" EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="ProductID"
            InsertVisible="False" ReadOnly="True"
            SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName" HeaderText="ProductName"
            SortExpression="ProductName" />
        <asp:BoundField DataField="SupplierID" HeaderText="SupplierID"
            SortExpression="SupplierID" />
        <asp:BoundField DataField="CategoryID" HeaderText="CategoryID"
            SortExpression="CategoryID" />
        <asp:BoundField DataField="QuantityPerUnit" HeaderText="QuantityPerUnit"
            SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
            SortExpression="UnitPrice" />
        <asp:BoundField DataField="UnitsInStock" HeaderText="UnitsInStock"
            SortExpression="UnitsInStock" />
        <asp:BoundField DataField="UnitsOnOrder" HeaderText="UnitsOnOrder"
            SortExpression="UnitsOnOrder" />
        <asp:BoundField DataField="ReorderLevel" HeaderText="ReorderLevel"
            SortExpression="ReorderLevel" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
        <asp:BoundField DataField="CategoryName" HeaderText="CategoryName"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="SupplierName"
            ReadOnly="True" SortExpression="SupplierName" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    InsertMethod="AddProduct" OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
    <InsertParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="unitsInStock" Type="Int16" />
        <asp:Parameter Name="unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="reorderLevel" Type="Int16" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
    </InsertParameters>
</asp:ObjectDataSource>
```

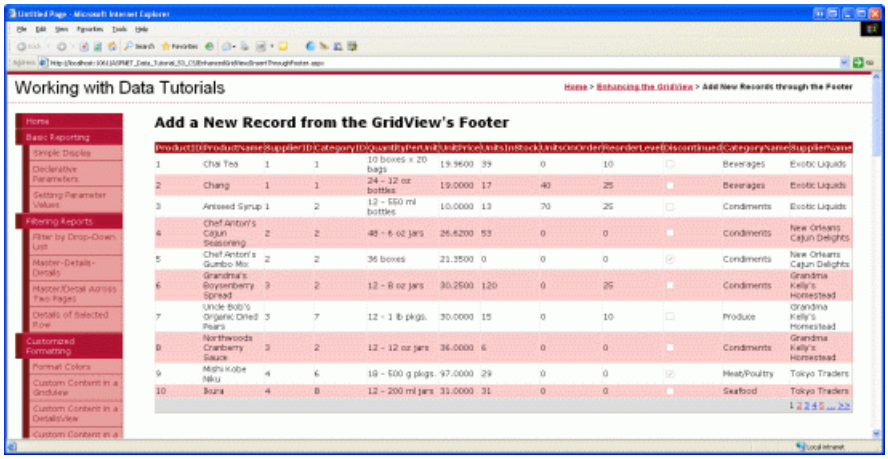


Figure 5: All Product Data Fields are Displayed in a Paged GridView (Click to view full-size image)

Step 2: Adding a Footer Row

Along with its header and data rows, the GridView includes a footer row. The header and footer rows are displayed depending on the values of the GridView s `ShowHeader` and `ShowFooter` properties. To show the footer row, simply set the `ShowFooter` property to `True`. As Figure 6 illustrates, setting the `ShowFooter` property to `True` adds a footer row to the grid.

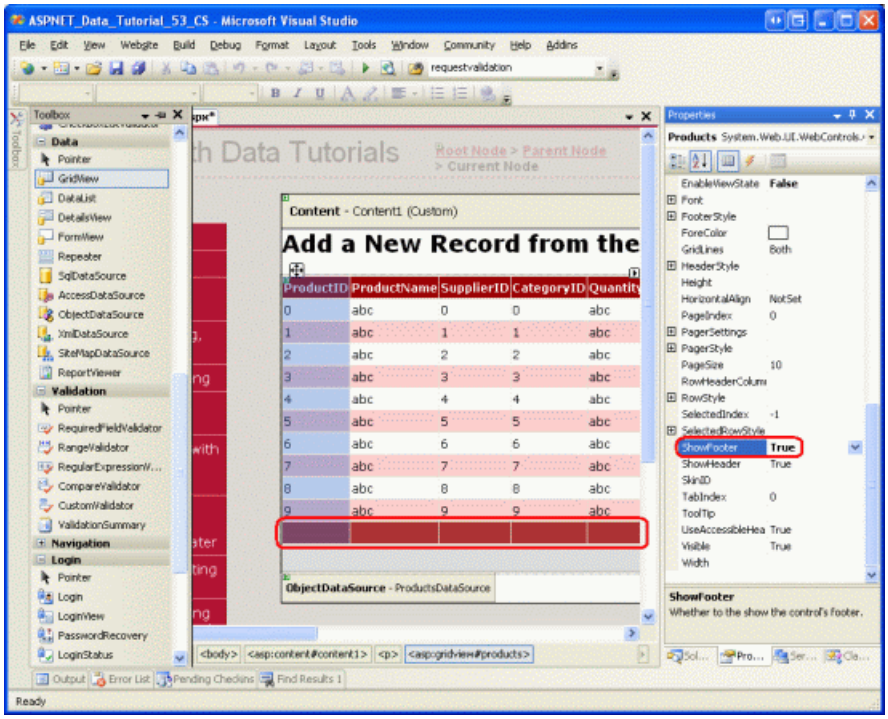


Figure 6: To Display the Footer Row, Set `ShowFooter` to `True` (Click to view full-size image)

Note that the footer row has a dark red background color. This is due to the DataWebControls Theme we created and applied to all pages back in the Displaying Data With the ObjectDataSource tutorial. Specifically, the `GridView.skin` file configures the `FooterStyle` property such that is uses the `FooterStyle` CSS class. The `FooterStyle` class is defined in `Styles.css` as follows:

```
.FooterStyle
{
    background-color: #a33;
    color: White;
    text-align: right;
}
```

> **NOTE**
>
> We ve explored using the GridView s footer row in previous tutorials. If needed, refer back to the Displaying Summary Information in the GridView's Footer tutorial for a refresher.

After setting the `ShowFooter` property to `True`, take a moment to view the output in a browser. Currently the footer row doesn t contain any text or Web controls. In Step 3 we'll modify the footer for each GridView field so that it includes the appropriate inserting interface.

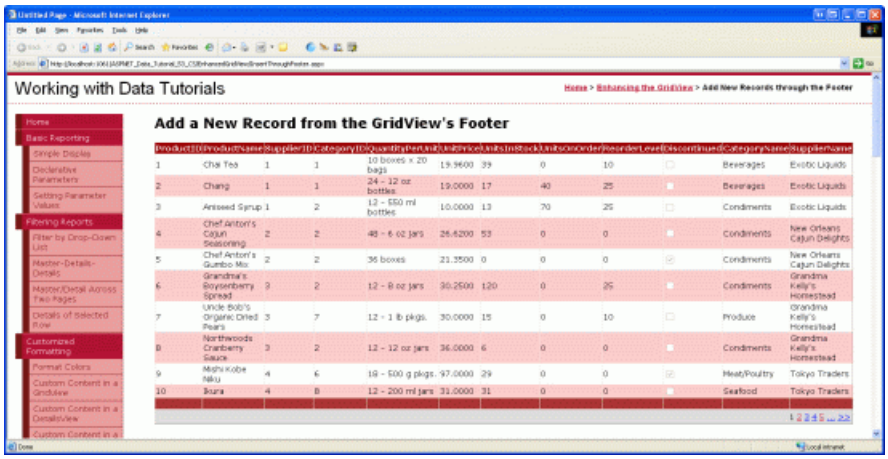


Figure 7: The Empty Footer Row is Displayed Above the Paging Interface Controls (Click to view full-size image)

Step 3: Customizing the Footer Row

Back in the Using TemplateFields in the GridView Control tutorial we saw how to greatly customize the display of a particular GridView column using TemplateFields (as opposed to BoundFields or CheckBoxFields); in Customizing the Data Modification Interface we looked at using TemplateFields to customize the editing interface in a GridView. Recall that a TemplateField is composed of a number of templates that defines the mix of markup, Web controls, and databinding syntax used for certain types of rows. The `ItemTemplate`, for example, specifies the template used for read-only rows, while the `EditItemTemplate` defines the template for the editable row.

Along with the `ItemTemplate` and `EditItemTemplate`, the TemplateField also includes a `FooterTemplate` that specifies the content for the footer row. Therefore, we can add the Web controls needed for each field s inserting interface into the `FooterTemplate`. To start, convert all of the fields in the GridView to TemplateFields. This can be done by clicking the Edit Columns link in the GridView s smart tag, selecting each field in the lower left corner, and clicking the Convert this field into a TemplateField link.

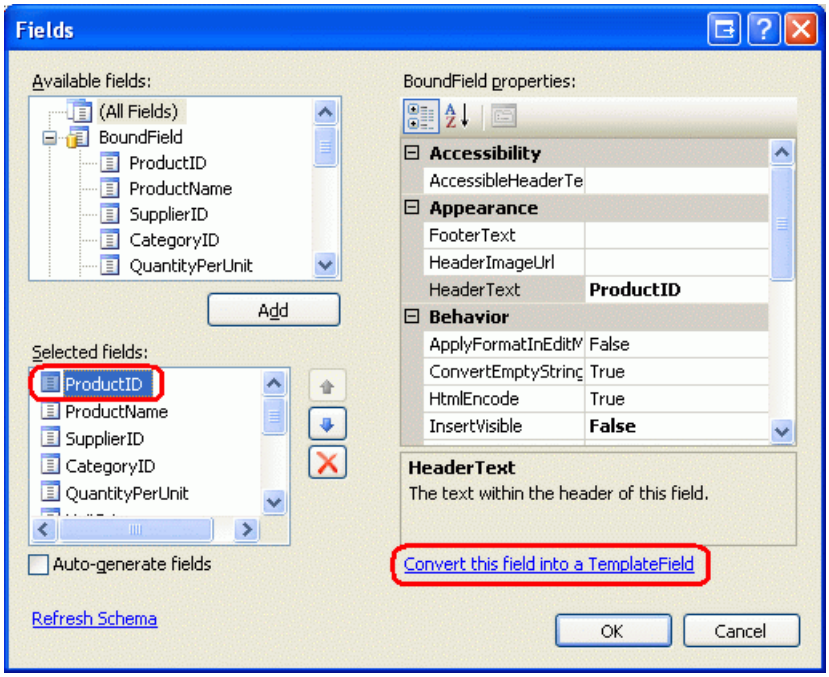


Figure 8: Convert Each Field Into a TemplateField

Clicking the Convert this field into a TemplateField turns the current field type into an equivalent TemplateField. For example, each BoundField is replaced by a TemplateField with an `ItemTemplate` that contains a Label that displays the corresponding data field and an `EditItemTemplate` that displays the data field in a TextBox. The `ProductName` BoundField has been converted into the following TemplateField markup:

```
<asp:TemplateField HeaderText="ProductName" SortExpression="ProductName">
    <EditItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
            Text='<%# Bind("ProductName") %>'></asp:TextBox>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Label ID="Label2" runat="server"
            Text='<%# Bind("ProductName") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
```

Likewise, the `Discontinued` CheckBoxField has been converted into a TemplateField whose `ItemTemplate` and `EditItemTemplate` contain a CheckBox Web control (with the `ItemTemplate` s CheckBox disabled). The read-only `ProductID` BoundField has been converted into a TemplateField with a Label control in both the `ItemTemplate` and `EditItemTemplate`. In short, converting an existing GridView field into a TemplateField is a quick and easy way to switch to the more customizable TemplateField without losing any of the existing field s functionality.

Since the GridView we re working with doesn t support editing, feel free to remove the `EditItemTemplate` from each TemplateField, leaving just the `ItemTemplate`. After doing this, your GridView s declarative markup should look like the following:

```
<asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
    AllowPaging="True" EnableViewState="False" ShowFooter="True">
    <Columns>
        <asp:TemplateField HeaderText="ProductID" InsertVisible="False"
            SortExpression="ProductID">
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("ProductID") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="ProductName" SortExpression="ProductName">
            <ItemTemplate>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="SupplierID" SortExpression="SupplierID">
            <ItemTemplate>
                <asp:Label ID="Label3" runat="server"
                    Text='<%# Bind("SupplierID") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="CategoryID" SortExpression="CategoryID">
            <ItemTemplate>
                <asp:Label ID="Label4" runat="server"
                    Text='<%# Bind("CategoryID") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="QuantityPerUnit"
            SortExpression="QuantityPerUnit">
            <ItemTemplate>
                <asp:Label ID="Label5" runat="server"
                    Text='<%# Bind("QuantityPerUnit") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="UnitPrice" SortExpression="UnitPrice">
            <ItemTemplate>
                <asp:Label ID="Label6" runat="server"
                    Text='<%# Bind("UnitPrice") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="UnitsInStock"
```

```
            SortExpression="UnitsInStock">
            <ItemTemplate>
                <asp:Label ID="Label7" runat="server"
                    Text='<%# Bind("UnitsInStock") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="UnitsOnOrder"
            SortExpression="UnitsOnOrder">
            <ItemTemplate>
                <asp:Label ID="Label8" runat="server"
                    Text='<%# Bind("UnitsOnOrder") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="ReorderLevel"
            SortExpression="ReorderLevel">
            <ItemTemplate>
                <asp:Label ID="Label9" runat="server"
                    Text='<%# Bind("ReorderLevel") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Discontinued"
            SortExpression="Discontinued">
            <ItemTemplate>
                <asp:CheckBox ID="CheckBox1" runat="server"
                    Checked='<%# Bind("Discontinued") %>' Enabled="false" />
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="CategoryName"
            SortExpression="CategoryName">
            <ItemTemplate>
                <asp:Label ID="Label10" runat="server"
                    Text='<%# Bind("CategoryName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="SupplierName"
            SortExpression="SupplierName">
            <ItemTemplate>
                <asp:Label ID="Label11" runat="server"
                    Text='<%# Bind("SupplierName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
```

Now that each GridView field has been converted into a TemplateField, we can enter the appropriate inserting interface into each field s `FooterTemplate`. Some of the fields will not have an inserting interface (`ProductID`, for instance); others will vary in the Web controls used to collect the new product s information.

To create the editing interface, choose the Edit Templates link from the GridView s smart tag. Then, from the drop-down list, select the appropriate field s `FooterTemplate` and drag the appropriate control from the Toolbox onto the Designer.

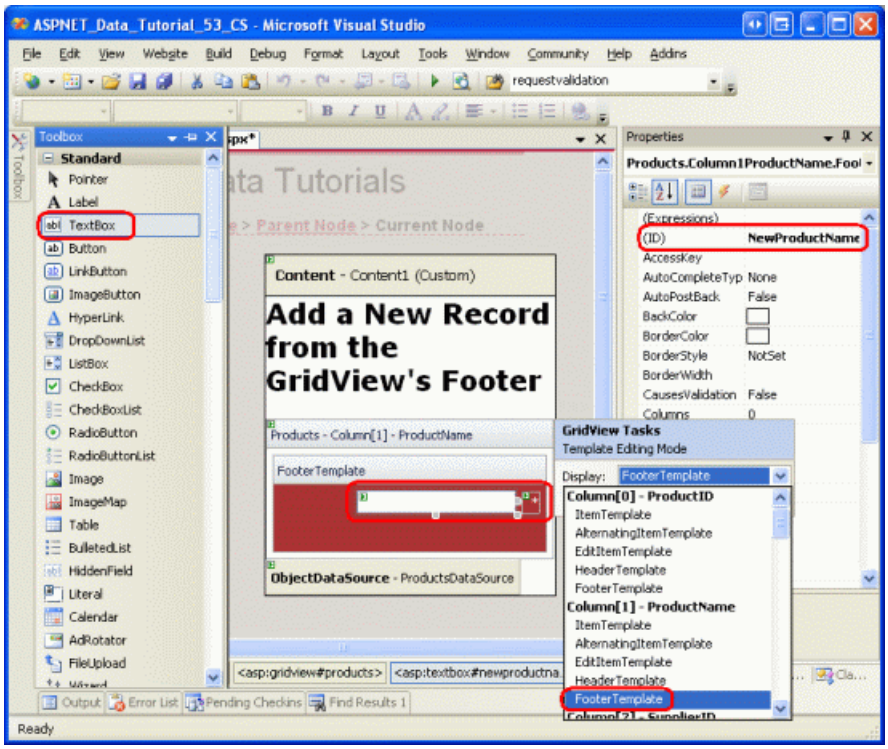


Figure 9: Add the Appropriate Inserting Interface to Each Field s `FooterTemplate` (Click to view full-size image)

The following bulleted list enumerates the GridView fields, specifying the inserting interface to add:

- `ProductID` none.
- `ProductName` add a TextBox and set its `ID` to `NewProductName`. Add a RequiredFieldValidator control as well to ensure that the user enters a value for the new product s name.
- `SupplierID` none.
- `CategoryID` none.
- `QuantityPerUnit` add a TextBox, setting its `ID` to `NewQuantityPerUnit`.
- `UnitPrice` add a TextBox named `NewUnitPrice` and a CompareValidator that ensures the value entered is a currency value greater than or equal to zero.
- `UnitsInStock` use a TextBox whose `ID` is set to `NewUnitsInStock`. Include a CompareValidator that ensures that the value entered is an integer value greater than or equal to zero.
- `UnitsOnOrder` use a TextBox whose `ID` is set to `NewUnitsOnOrder`. Include a CompareValidator that ensures that the value entered is an integer value greater than or equal to zero.
- `ReorderLevel` use a TextBox whose `ID` is set to `NewReorderLevel`. Include a CompareValidator that ensures that the value entered is an integer value greater than or equal to zero.
- `Discontinued` add a CheckBox, setting its `ID` to `NewDiscontinued`.
- `CategoryName` add a DropDownList and set its `ID` to `NewCategoryID`. Bind it to a new ObjectDataSource named `CategoriesDataSource` and configure it to use the `CategoriesBLL` class s `GetCategories()` method. Have the DropDownList s `ListItem` s display the `CategoryName` data field, using the `CategoryID` data field as their values.
- `SupplierName` add a DropDownList and set its `ID` to `NewSupplierID`. Bind it to a new ObjectDataSource named `SuppliersDataSource` and configure it to use the `SuppliersBLL` class s `GetSuppliers()` method. Have the DropDownList s `ListItem` s display the `CompanyName` data field, using the `SupplierID` data field as their values.

For each of the validation controls, clear out the `ForeColor` property so that the `FooterStyle` CSS class s white foreground color will be used in place of the default red. Also use the `ErrorMessage` property for a detailed description, but set the `Text` property to an asterisk. To prevent the validation control s text from causing the inserting interface to wrap to two lines, set the `FooterStyle` s `Wrap` property to false for each of the `FooterTemplate` s that use a validation control. Finally, add a ValidationSummary control beneath the GridView

and set its `ShowMessageBox` property to `True` and its `ShowSummary` property to `False`.

When adding a new product, we need to provide the `CategoryID` and `SupplierID`. This information is captured through the DropDownLists in the footer cells for the `CategoryName` and `SupplierName` fields. I chose to use these fields as opposed to the `CategoryID` and `SupplierID` TemplateFields because in the grid s data rows the user is likely more interested in seeing the category and supplier names rather than their ID values. Since the `CategoryID` and `SupplierID` values are now being captured in the `CategoryName` and `SupplierName` field s inserting interfaces, we can remove the `CategoryID` and `SupplierID` TemplateFields from the GridView.

Similarly, the `ProductID` is not used when adding a new product, so the `ProductID` TemplateField can be removed as well. However, let s leave the `ProductID` field in the grid. In addition to the TextBoxes, DropDownLists, CheckBoxes, and validation controls that make up the inserting interface, we'll also need an Add button that, when clicked, performs the logic to add the new product to the database. In Step 4 we'll include an Add button in the inserting interface in the `ProductID` TemplateField s `FooterTemplate`.

Feel free to improve the appearance of the various GridView fields. For example, you might want to format the `UnitPrice` values as a currency, right-align the `UnitsInStock`, `UnitsOnOrder`, and `ReorderLevel` fields, and update the `HeaderText` values for the TemplateFields.

After creating the slew of inserting interfaces in the `FooterTemplate` s, removing the `SupplierID`, and `CategoryID` TemplateFields, and improving the aesthetics of the grid through formatting and aligning the TemplateFields, your GridView s declarative markup should look similar to the following:

```
<asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
    AllowPaging="True" EnableViewState="False" ShowFooter="True">
    <Columns>
        <asp:TemplateField HeaderText="ProductID" InsertVisible="False"
            SortExpression="ProductID">
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("ProductID") %>'></asp:Label>
            </ItemTemplate>
            <ItemStyle HorizontalAlign="Center" />
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
            <ItemTemplate>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:Label>
            </ItemTemplate>
            <FooterTemplate>
                <asp:TextBox ID="NewProductName" runat="server"></asp:TextBox>
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                    runat="server" ControlToValidate="NewProductName"
                    Display="Dynamic"  ForeColor="
                    ErrorMessage="You must enter a name for the new product.">
                    * </asp:RequiredFieldValidator>
            </FooterTemplate>
            <FooterStyle Wrap="False" />
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Category" SortExpression="CategoryName">
            <ItemTemplate>
                <asp:Label ID="Label10" runat="server"
                    Text='<%# Bind("CategoryName") %>'></asp:Label>
            </ItemTemplate>
            <FooterTemplate>
                <asp:DropDownList ID="NewCategoryID" runat="server"
                    DataSourceID="CategoriesDataSource"
                    DataTextField="CategoryName" DataValueField="CategoryID">
                </asp:DropDownList>
                <asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
                    OldValuesParameterFormatString="original_{0}"
                    SelectMethod="GetCategories" TypeName="CategoriesBLL">
```

```
            </asp:ObjectDataSource>
        </FooterTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Supplier" SortExpression="SupplierName">
        <ItemTemplate>
            <asp:Label ID="Label11" runat="server"
                Text='<%# Bind("SupplierName") %>'></asp:Label>
        </ItemTemplate>
        <FooterTemplate>
            <asp:DropDownList ID="NewSupplierID" runat="server"
                DataSourceID="SuppliersDataSource"
                DataTextField="CompanyName" DataValueField="SupplierID">
            </asp:DropDownList><asp:ObjectDataSource ID="SuppliersDataSource"
                runat="server" OldValuesParameterFormatString="original_{0}"
                SelectMethod="GetSuppliers" TypeName="SuppliersBLL">
            </asp:ObjectDataSource>
        </FooterTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Qty/Unit" SortExpression="QuantityPerUnit">
        <ItemTemplate>
            <asp:Label ID="Label5" runat="server"
                Text='<%# Bind("QuantityPerUnit") %>'></asp:Label>
        </ItemTemplate>
        <FooterTemplate>
            <asp:TextBox ID="NewQuantityPerUnit" runat="server"></asp:TextBox>
        </FooterTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Price" SortExpression="UnitPrice">
        <ItemTemplate>
            <asp:Label ID="Label6" runat="server"
                Text='<%# Bind("UnitPrice", "{0:c}") %>'></asp:Label>
        </ItemTemplate>
        <FooterTemplate>
            $<asp:TextBox ID="NewUnitPrice" runat="server" Columns="8" />
            <asp:CompareValidator ID="CompareValidator1" runat="server"
                ControlToValidate="NewUnitPrice"
                ErrorMessage="You must enter a valid currency value greater than
                    or equal to 0.00. Do not include the currency symbol."
                ForeColor="" Operator="GreaterThanEqual" Type="Currency"
                ValueToCompare="0" Display="Dynamic">
                * </asp:CompareValidator>
        </FooterTemplate>
        <ItemStyle HorizontalAlign="Right" />
        <FooterStyle Wrap="False" />
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Units In Stock"
        SortExpression="Units In Stock">
        <ItemTemplate>
            <asp:Label ID="Label7" runat="server"
                Text='<%# Bind("UnitsInStock") %>'></asp:Label>
        </ItemTemplate>
        <FooterTemplate>
            <asp:TextBox ID="NewUnitsInStock" runat="server" Columns="5" />
            <asp:CompareValidator ID="CompareValidator2" runat="server"
                ControlToValidate="NewUnitsInStock" Display="Dynamic"
                ErrorMessage="You must enter a valid numeric value for units
                    in stock that's greater than or equal to zero."
                ForeColor="" Operator="GreaterThanEqual" Type="Integer"
                    ValueToCompare="0">*</asp:CompareValidator>
        </FooterTemplate>
        <ItemStyle HorizontalAlign="Right" />
        <FooterStyle Wrap="False" />
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Units On Order" SortExpression="UnitsOnOrder">
        <ItemTemplate>
            <asp:Label ID="Label8" runat="server"
                Text='<%# Bind("UnitsOnOrder") %>'></asp:Label>
        </ItemTemplate>
        <FooterTemplate>
```

```
                <asp:TextBox ID="NewUnitsOnOrder" runat="server" Columns="5" />
                <asp:CompareValidator ID="CompareValidator3" runat="server"
                    ControlToValidate="NewUnitsOnOrder" Display="Dynamic"
                    ErrorMessage="You must enter a valid numeric value for units on
                        order that's greater than or equal to zero."
                    ForeColor="" Operator="GreaterThanEqual" Type="Integer"
                    ValueToCompare="0">*</asp:CompareValidator>
            </FooterTemplate>
            <ItemStyle HorizontalAlign="Right" />
            <FooterStyle Wrap="False" />
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Reorder Level" SortExpression="ReorderLevel">
            <ItemTemplate>
                <asp:Label ID="Label9" runat="server"
                    Text='<%# Bind("ReorderLevel") %>'></asp:Label>
            </ItemTemplate>
            <FooterTemplate>
                <asp:TextBox ID="NewReorderLevel" runat="server" Columns="5" />
                <asp:CompareValidator ID="CompareValidator4" runat="server"
                    ControlToValidate="NewReorderLevel" Display="Dynamic"
                    ErrorMessage="You must enter a valid numeric value for reorder
                        level that's greater than or equal to zero."
                    ForeColor="" Operator="GreaterThanEqual" Type="Integer"
                    ValueToCompare="0">*</asp:CompareValidator>
            </FooterTemplate>
            <ItemStyle HorizontalAlign="Right" />
            <FooterStyle Wrap="False" />
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Discontinued" SortExpression="Discontinued">
            <ItemTemplate>
                <asp:CheckBox ID="CheckBox1" runat="server"
                    Checked='<%# Bind("Discontinued") %>' Enabled="false" />
            </ItemTemplate>
            <FooterTemplate>
                <asp:CheckBox ID="NewDiscontinued" runat="server" />
            </FooterTemplate>
            <ItemStyle HorizontalAlign="Center" />
            <FooterStyle HorizontalAlign="Center" />
        </asp:TemplateField>
    </Columns>
</asp:GridView>
```

When viewed through a browser, the GridView s footer row now includes the completed inserting interface (see Figure 10). At this point, the inserting interface doesn t include a means for the user to indicate that she s entered the data for the new product and wants to insert a new record into the database. Also, we ve yet to address how the data entered into the footer will translate into a new record in the `Products` database. In Step 4 we'll look at how to include an Add button to the inserting interface and how to execute code on postback when it s clicked. Step 5 shows how to insert a new record using the data from the footer.

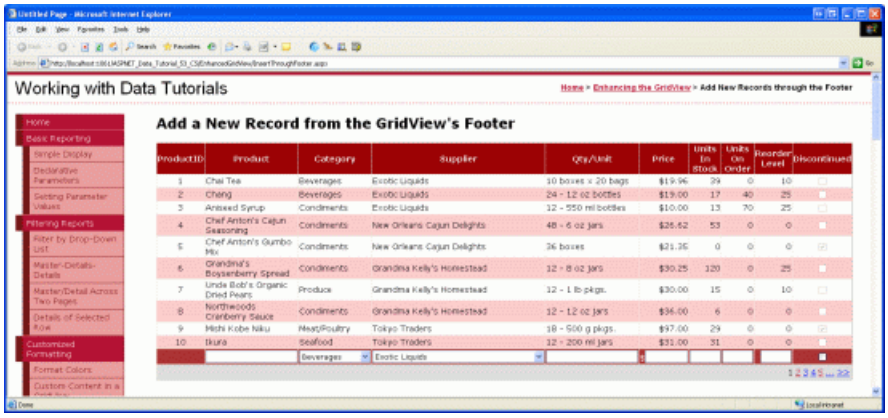


Figure 10: The GridView Footer Provides an Interface for Adding a New Record (Click to view full-size image)

Step 4: Including an Add Button in the Inserting Interface

We need to include an Add button somewhere in the inserting interface since the footer row s inserting interface currently lacks the means for the user to indicate that they have completed entering the new product s information. This could be placed in one of the existing `FooterTemplate` s or we could add a new column to the grid for this purpose. For this tutorial, let s place the Add button in the `ProductID` TemplateField s `FooterTemplate`.

From the Designer, click on the Edit Templates link in the GridView s smart tag and then choose the `ProductID` field s `FooterTemplate` from the drop-down list. Add a Button Web control (or a LinkButton or ImageButton, if you prefer) to the template, setting its ID to `AddProduct`, its `CommandName` to Insert, and its `Text` property to Add as shown in Figure 11.

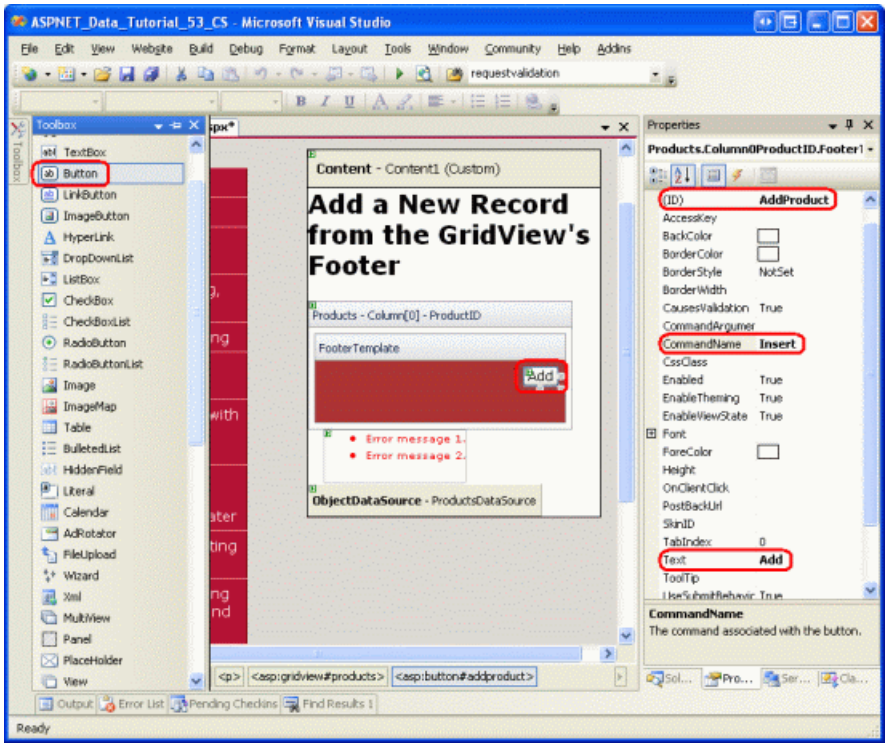


Figure 11: Place the Add Button in the `ProductID` TemplateField s `FooterTemplate` (Click to view full-size image)

Once you ve included the Add button, test out the page in a browser. Note that when clicking the Add button with invalid data in the inserting interface, the postback is short circuited and the ValidationSummary control indicates the invalid data (see Figure 12). With appropriate data entered, clicking the Add button causes a postback. No record is added to the database, however. We will need to write a bit of code to actually perform the insert.

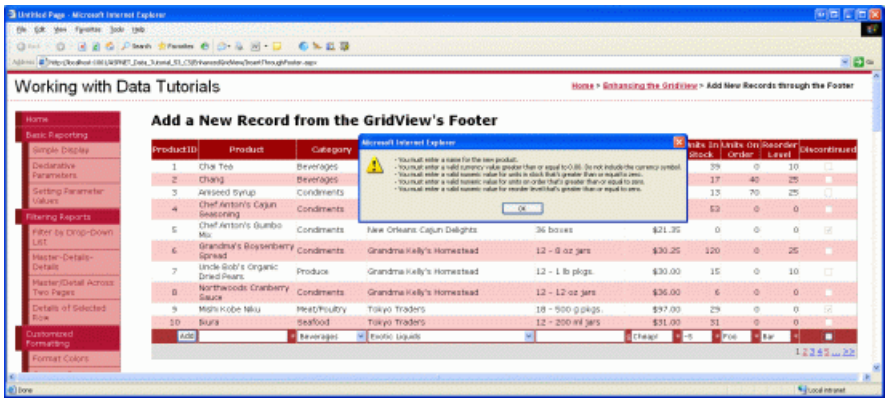


Figure 12: The Add Button s Postback is Short Circuited if There is Invalid Data in the Inserting Interface (Click to view full-size image)

NOTE

The validation controls in the inserting interface were not assigned to a validation group. This works fine so long as the inserting interface is the only set of validation controls on the page. If, however, there are other validation controls on the page (such as validation controls in the grid s editing interface), the validation controls in the inserting interface and Add button s `ValidationGroup` properties should be assigned the same value so as to associate these controls with a particular validation group. See Dissecting the Validation Controls in ASP.NET 2.0 for more information on partitioning the validation controls and buttons on a page into validation groups.

Step 5: Inserting a New Record Into the `Products` Table

When utilizing the built-in editing features of the GridView, the GridView automatically handles all of the work necessary for performing the update. In particular, when the Update button is clicked it copies the values entered from the editing interface to the parameters in the ObjectDataSource s `UpdateParameters` collection and kicks off the update by invoking the ObjectDataSource s `Update()` method. Since the GridView does not provide such built-in functionality for inserting, we must implement code that calls the ObjectDataSource s `Insert()` method and copies the values from the inserting interface to the ObjectDataSource s `InsertParameters` collection.

This insert logic should be executed after the Add button has been clicked. As discussed in the Adding and Responding to Buttons in a GridView tutorial, anytime a Button, LinkButton, or ImageButton in a GridView is clicked, the GridView s `RowCommand` event fires on postback. This event fires whether the Button, LinkButton, or ImageButton was added explicitly such as the Add button in the footer row or if it was automatically added by the GridView (such as the LinkButtons at the top of each column when Enable Sorting is selected, or the LinkButtons in the paging interface when Enable Paging is selected).

Therefore, to respond to the user clicking the Add button, we need to create an event handler for the GridView s `RowCommand` event. Since this event fires whenever *any* Button, LinkButton, or ImageButton in the GridView is clicked, it s vital that we only proceed with the inserting logic if the `CommandName` property passed into the event handler maps to the `CommandName` value of the Add button (Insert). Moreover, we should also only proceed if the validation controls report valid data. To accommodate this, create an event handler for the `RowCommand` event with the following code:

```
Protected Sub Products_RowCommand(sender As Object, e As GridViewCommandEventArgs) _
    Handles Products.RowCommand

    ' Insert data if the CommandName == "Insert"
    ' and the validation controls indicate valid data...
    If e.CommandName = "Insert" AndAlso Page.IsValid Then
        ' TODO: Insert new record...
    End If
End Sub
```

NOTE

You may be wondering why the event handler bothers checking the `Page.IsValid` property. After all, won't the postback be suppressed if invalid data is provided in the inserting interface? This assumption is correct as long as the user has not disabled JavaScript or has taken steps to circumvent the client-side validation logic. In short, one should never rely strictly on client-side validation; a server-side check for validity should always be performed before working with the data.

In Step 1 we created the `ProductsDataSource` ObjectDataSource such that its `Insert()` method is mapped to the `ProductsBLL` class s `AddProduct` method. To insert the new record into the `Products` table, we can simply invoke the ObjectDataSource s `Insert()` method:

```
Protected Sub Products_RowCommand(sender As Object, e As GridViewCommandEventArgs) _
    Handles Products.RowCommand

    ' Insert data if the CommandName == "Insert"
    ' and the validation controls indicate valid data...
    If e.CommandName = "Insert" AndAlso Page.IsValid Then
        ' Insert new record
        ProductsDataSource.Insert()
    End If
End Sub
```

Now that the `Insert()` method has been invoked, all that remains is to copy the values from the inserting interface to the parameters passed to the `ProductsBLL` class s `AddProduct` method. As we saw back in the Examining the Events Associated with Inserting, Updating, and Deleting tutorial, this can be accomplished through the ObjectDataSource s `Inserting` event. In the `Inserting` event we need to programmatically reference the controls from the `Products` GridView s footer row and assign their values to the `e.InputParameters` collection. If the user omits a value such as leaving the `ReorderLevel` TextBox blank we need to specify that the value inserted into the database should be `NULL`. Since the `AddProducts` method accepts nullable types for the nullable database fields, simply use a nullable type and set its value to `Nothing` in the case where user input is omitted.

```
Protected Sub ProductsDataSource_Inserting _
    (sender As Object, e As .ObjectDataSourceMethodEventArgs) _
    Handles ProductsDataSource.Inserting

    ' Programmatically reference Web controls in the inserting interface...
    Dim NewProductName As TextBox = _
        Products.FooterRow.FindControl("NewProductName")
    Dim NewCategoryID As DropDownList = _
        Products.FooterRow.FindControl("NewCategoryID")
    Dim NewSupplierID As DropDownList = _
        Products.FooterRow.FindControl("NewSupplierID")
    Dim NewQuantityPerUnit As TextBox = _
        Products.FooterRow.FindControl("NewQuantityPerUnit")
    Dim NewUnitPrice As TextBox = _
        Products.FooterRow.FindControl("NewUnitPrice")
    Dim NewUnitsInStock As TextBox = _
        Products.FooterRow.FindControl("NewUnitsInStock")
    Dim NewUnitsOnOrder As TextBox = _
        Products.FooterRow.FindControl("NewUnitsOnOrder")
    Dim NewReorderLevel As TextBox = _
        Products.FooterRow.FindControl("NewReorderLevel")
    Dim NewDiscontinued As CheckBox = _
        Products.FooterRow.FindControl("NewDiscontinued")
    ' Set the ObjectDataSource's InsertParameters values...
    e.InputParameters("productName") = NewProductName.Text
    e.InputParameters("supplierID") = _
        Convert.ToInt32(NewSupplierID.SelectedValue)
    e.InputParameters("categoryID") = _
        Convert.ToInt32(NewCategoryID.SelectedValue)
    Dim quantityPerUnit As String = Nothing
    If Not String.IsNullOrEmpty(NewQuantityPerUnit.Text) Then
        quantityPerUnit = NewQuantityPerUnit.Text
    End If
    e.InputParameters("quantityPerUnit") = quantityPerUnit
    Dim unitPrice As Nullable(Of Decimal) = Nothing
    If Not String.IsNullOrEmpty(NewUnitPrice.Text) Then
        unitPrice = Convert.ToDecimal(NewUnitPrice.Text)
    End If
    e.InputParameters("unitPrice") = unitPrice
    Dim unitsInStock As Nullable(Of Short) = Nothing
    If Not String.IsNullOrEmpty(NewUnitsInStock.Text) Then
        unitsInStock = Convert.ToInt16(NewUnitsInStock.Text)
    End If
    e.InputParameters("unitsInStock") = unitsInStock
    Dim unitsOnOrder As Nullable(Of Short) = Nothing
    If Not String.IsNullOrEmpty(NewUnitsOnOrder.Text) Then
        unitsOnOrder = Convert.ToInt16(NewUnitsOnOrder.Text)
    End If
    e.InputParameters("unitsOnOrder") = unitsOnOrder
    Dim reorderLevel As Nullable(Of Short) = Nothing
    If Not String.IsNullOrEmpty(NewReorderLevel.Text) Then
        reorderLevel = Convert.ToInt16(NewReorderLevel.Text)
    End If
    e.InputParameters("reorderLevel") = reorderLevel
    e.InputParameters("discontinued") = NewDiscontinued.Checked
End Sub
```

With the `Inserting` event handler completed, new records can be added to the `Products` database table via the GridView s footer row. Go ahead and try adding several new products.

Enhancing and Customizing the Add Operation

Currently, clicking the Add button adds a new record to the database table but does not provide any sort of visual feedback that the record has successfully been added. Ideally, a Label Web control or client-side alert box would

inform the user that their insert has completed with success. I leave this as an exercise for the reader.

The GridView used in this tutorial does not apply any sort order to the listed products, nor does it allow the end user to sort the data. Consequently, the records are ordered as they are in the database by their primary key field. Since each new record has a `ProductID` value greater than the last one, every time a new product is added it is tacked on to the end of the grid. Therefore, you may want to automatically send the user to the last page of the GridView after adding a new record. This can be accomplished by adding the following line of code after the call to `ProductsDataSource.Insert()` in the `RowCommand` event handler to indicate that the user needs to be sent to the last page after binding the data to the GridView:

```
' Indicate that the user needs to be sent to the last page
SendUserToLastPage = True
```

`SendUserToLastPage` is a page-level Boolean variable that is initially assigned a value of `False`. In the GridView s `DataBound` event handler, if `SendUserToLastPage` is false, the `PageIndex` property is updated to send the user to the last page.

```
Protected Sub Products_DataBound(sender As Object, e As EventArgs) _
    Handles Products.DataBound

    ' Send user to last page of data, if needed
    If SendUserToLastPage Then
        Products.PageIndex = Products.PageCount - 1
    End If
End Sub
```

The reason the `PageIndex` property is set in the `DataBound` event handler (as opposed to the `RowCommand` event handler) is because when the `RowCommand` event handler fires we ve yet to add the new record to the `Products` database table. Therefore, in the `RowCommand` event handler the last page index (`PageCount - 1`) represents the last page index *before* the new product has been added. For the majority of products being added, the last page index is the same after adding the new product. But when the added product results in a new last page index, if we incorrectly update the `PageIndex` in the `RowCommand` event handler then we'll be taken to the second to last page (the last page index prior to adding the new product) as opposed to the new last page index. Since the `DataBound` event handler fires after the new product has been added and the data rebound to the grid, by setting the `PageIndex` property there we know we re getting the correct last page index.

Finally, the GridView used in this tutorial is quite wide due to the number of fields that must be collected for adding a new product. Due to this width, a DetailsView s vertical layout might be preferred. The GridView s overall width could be reduced by collecting fewer inputs. Perhaps we don t need to collect the `UnitsOnOrder`, `UnitsInStock`, and `ReorderLevel` fields when adding a new product, in which case these fields could be removed from the GridView.

To adjust the data collected, we can use one of two approaches:

- Continue to use the `AddProduct` method that expects values for the `UnitsOnOrder`, `UnitsInStock`, and `ReorderLevel` fields. In the `Inserting` event handler, provide hard-coded, default values to use for these inputs that have been removed from the inserting interface.
- Create a new overload of the `AddProduct` method in the `ProductsBLL` class that does not accept inputs for the `UnitsOnOrder`, `UnitsInStock`, and `ReorderLevel` fields. Then, in the ASP.NET page, configure the ObjectDataSource to use this new overload.

Either option will work equally as well. In past tutorials we used the latter option, creating multiple overloads for the `ProductsBLL` class s `UpdateProduct` method.

Summary

The GridView lacks the built-in inserting capabilities found in the DetailsView and FormView, but with a bit of effort an inserting interface can be added to the footer row. To display the footer row in a GridView simply set its `ShowFooter` property to `True`. The footer row content can be customized for each field by converting the field to a TemplateField and adding the inserting interface to the `FooterTemplate`. As we saw in this tutorial, the `FooterTemplate` can contain Buttons, TextBoxes, DropDownLists, CheckBoxes, data source controls for populating data-driven Web controls (such as DropDownLists), and validation controls. Along with controls for collecting the user s input, an Add Button, LinkButton, or ImageButton is needed.

When the Add button is clicked, the ObjectDataSource s `Insert()` method is invoked to start the inserting workflow. The ObjectDataSource will then call the configured insert method (the `ProductsBLL` class s `AddProduct` method, in this tutorial). We must copy the values from the GridView s inserting interface to the ObjectDataSource s `InsertParameters` collection prior to the insert method being invoked. This can be accomplished by programmatically referencing the inserting interface Web controls in the ObjectDataSource s `Inserting` event handler.

This tutorial completes our look at techniques for enhancing the GridView s appearance. The next set of tutorials will examine how to work with binary data such as images, PDFs, Word documents, and so on and the data Web controls.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Bernadette Leigh. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Working with Binary Files

2/20/2020 • 2 minutes to read • Edit Online

These tutorials show how to let users upload binary files (like Word or PDF documents) to the server, where they can be stored in folders or in the database.

- Uploading Files (C#)
- Displaying Binary Data in the Data Web Controls (C#)
- Including a File Upload Option When Adding a New Record (C#)
- Updating and Deleting Existing Binary Data (C#)
- Uploading Files (VB)
- Displaying Binary Data in the Data Web Controls (VB)
- Including a File Upload Option When Adding a New Record (VB)
- Updating and Deleting Existing Binary Data (VB)

Uploading Files (C#)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> Learn how to allow users to upload binary files (such as Word or PDF documents) to your Web site where they may be stored in either the server's file system or the database.

Introduction

All of the tutorials we ve examined so far have worked exclusively with text data. However, many applications have data models that capture both text and binary data. An online dating site might allow users to upload a picture to associate with their profile. A recruiting website might let users upload their resume as a Microsoft Word or PDF document.

Working with binary data adds a new set of challenges. We must decide how the binary data is stored in the application. The interface used for inserting new records has to be updated to allow the user to upload a file from their computer and extra steps must be taken to display or provide a means for downloading a record s associated binary data. In this tutorial and the next three we'll explore how to hurdle these challenges. At the end of these tutorials we'll have built a fully functional application that associates a picture and PDF brochure with each category. In this particular tutorial we'll look at different techniques for storing binary data and explore how to enable users to upload a file from their computer and have it saved on the web server s file system.

> **NOTE**
>
> Binary data that is part of an application s data model is sometimes referred to as a BLOB, an acronym for Binary Large OBject. In these tutorials I have chosen to use the terminology binary data, although the term BLOB is synonymous.

Step 1: Creating the Working with Binary Data Web Pages

Before we begin to explore the challenges associated with adding support for binary data, let s first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial and the next three. Start by adding a new folder named `BinaryData`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `FileUpload.aspx`
- `DisplayOrDownloadData.aspx`
- `UploadInDetailsView.aspx`
- `UpdatingAndDeleting.aspx`

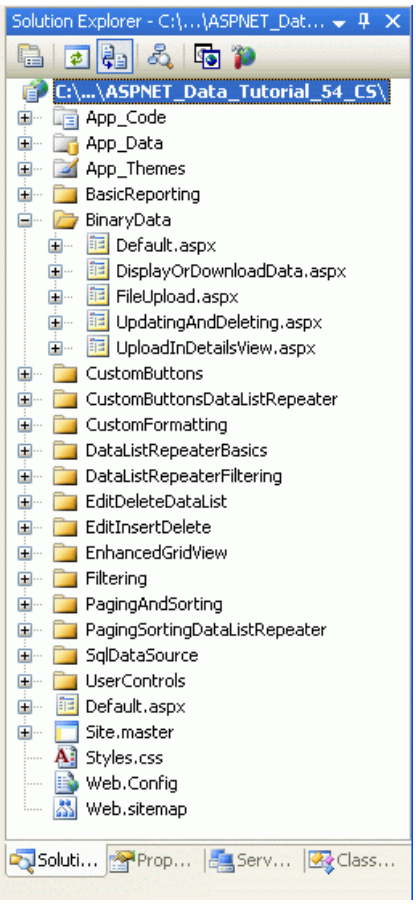


Figure 1: Add the ASP.NET Pages for the Binary Data-Related Tutorials

Like in the other folders, `Default.aspx` in the `BinaryData` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

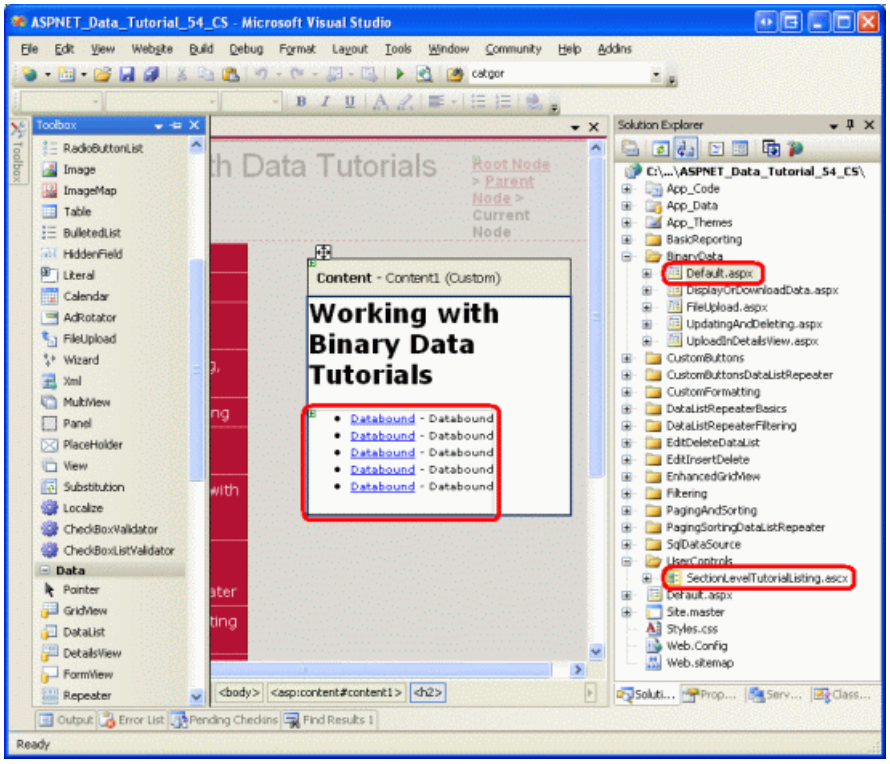


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add these pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Enhancing the GridView `<siteMapNode>`:

```
<siteMapNode
    title="Working with Binary Data"
    url="~/BinaryData/Default.aspx"
    description="Extend the data model to include collecting binary data.">

    <siteMapNode
        title="Uploading Files"
        url="~/BinaryData/FileUpload.aspx"
        description="Examine the different ways to store binary data on the
                     web server and see how to accept uploaded files from users
                     with the FileUpload control." />
    <siteMapNode
        title="Display or Download Binary Data"
        url="~/BinaryData/DisplayOrDownloadData.aspx"
        description="Let users view or download the captured binary data." />
    <siteMapNode
        title="Adding New Binary Data"
        url="~/BinaryData/UploadInDetailsView.aspx"
        description="Learn how to augment the inserting interface to
                     include a FileUpload control." />
    <siteMapNode
        title="Updating and Deleting Existing Binary Data"
        url="~/BinaryData/UpdatingAndDeleting.aspx"
        description="Learn how to update and delete existing binary data." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the Working with Binary Data tutorials.

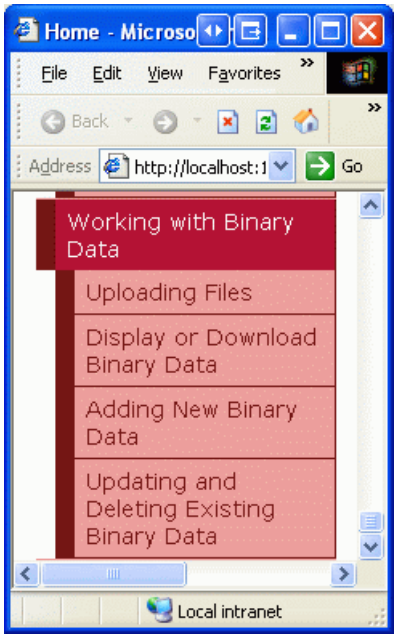


Figure 3: The Site Map Now Includes Entries for the Working with Binary Data Tutorials

Step 2: Deciding Where to Store the Binary Data

Binary data that is associated with the application s data model can be stored in one of two places: on the web server s file system with a reference to the file stored in the database; or directly within the database itself (see Figure 4). Each approach has its own set of pros and cons and merits a more detailed discussion.

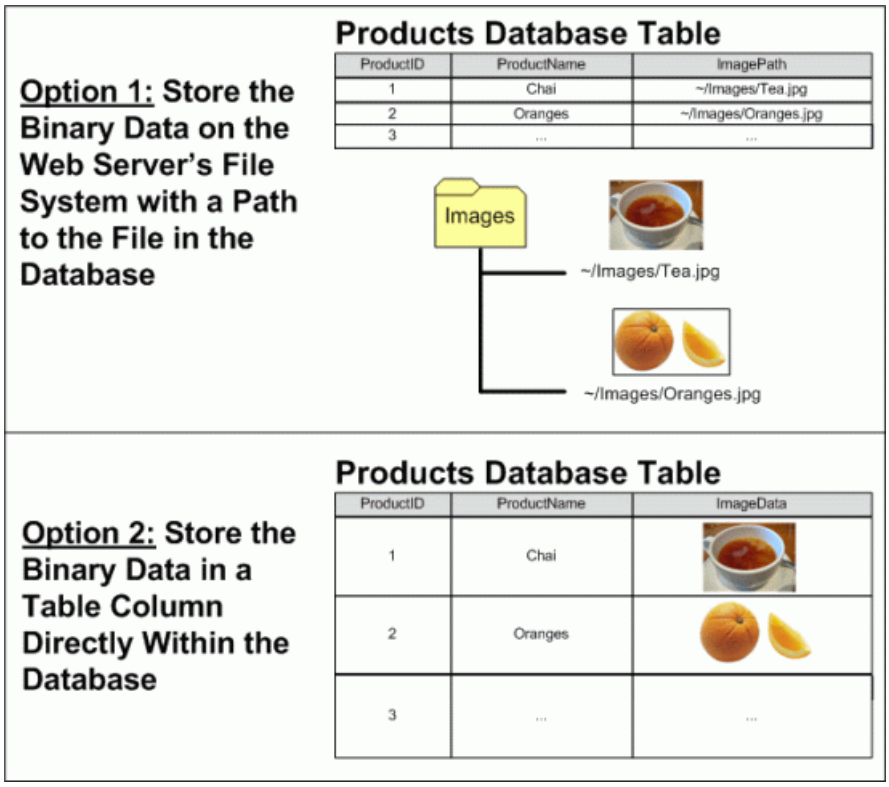


Figure 4: Binary Data Can Be Stored On the File System or Directly in the Database (Click to view full-size image)

Imagine that we wanted to extend the Northwind database to associate a picture with each product. One option would be to store these image files on the web server s file system and record the path in the `Products` table. With this approach, we d add an `ImagePath` column to the `Products` table of type `varchar(200)`, perhaps. When a user uploaded a picture for Chai, that picture might be stored on the web server s file system at `~/Images/Tea.jpg`, where `~` represents the application s physical path. That is, if the web site is rooted at the physical path `C:\Websites\Northwind\`, `~/Images/Tea.jpg` would be equivalent to `C:\Websites\Northwind\Images\Tea.jpg`. After uploading the image file, we d update the Chai record in the `Products` table so that its `ImagePath` column referenced the path of the new image. We could use `~/Images/Tea.jpg` or just `Tea.jpg` if we decided that all product images would be placed in the application s `Images` folder.

The main advantages of storing the binary data on the file system are:

- **Ease of implementation** as we'll see shortly, storing and retrieving binary data stored directly within the database involves a bit more code than when working with data through the file system. Additionally, in order for a user to view or download binary data they must be presented with a URL to that data. If the data resides on the web server s file system, the URL is straightforward. If the data is stored in the database, however, a web page needs to be created that will retrieve and return the data from the database.
- **Wider access to the binary data** the binary data may need to be accessible to other services or applications, ones that cannot pull the data from the database. For example, the images associated with each product might also need to be available to users through FTP, in which case we d want to store the binary data on the file system.
- **Performance** if the binary data is stored on the file system, the demand and network congestion between the database server and web server will be less than if the binary data is stored directly within the database.

The main disadvantage of storing binary data on the file system is that it decouples the data from the database. If a record is deleted from the `Products` table, the associated file on the web server s file system is not automatically deleted. We must write extra code to delete the file or the file system will become cluttered with unused, orphaned files. Furthermore, when backing up the database, we must make sure to make backups of the associated binary data on the file system, as well. Moving the database to another site or server poses similar challenges.

Alternatively, binary data can be stored directly in a Microsoft SQL Server 2005 database by creating a column of type `varbinary`. Like with other variable length data types, you can specify a maximum length of the binary data that can be held in this column. For example, to reserve at most 5,000 bytes use `varbinary(5000)`; `varbinary(MAX)` allows for the maximum storage size, about 2 GB.

The main advantage of storing binary data directly in the database is the tight coupling between the binary data and the database record. This greatly simplifies database administration tasks, like backups or moving the database to a different site or server. Also, deleting a record automatically deletes the corresponding binary data. There are also more subtle benefits of storing the binary data in the database. See Storing Binary Files Directly in the Database Using ASP.NET 2.0 for a more in-depth discussion.

> **NOTE**
> In Microsoft SQL Server 2000 and earlier versions, the `varbinary` data type had a maximum limit of 8,000 bytes. To store up to 2 GB of binary data the `image` data type needs to be used instead. With the addition of `MAX` in SQL Server 2005, however, the `image` data type has been deprecated. It s still supported for backwards compatibility, but Microsoft has announced that the `image` data type will be removed in a future version of SQL Server.

If you are working with an older data model you may see the `image` data type. The Northwind database s `Categories` table has a `Picture` column that can be used to store the binary data of an image file for the category. Since the Northwind database has its roots in Microsoft Access and earlier versions of SQL Server, this column is of type `image`.

For this tutorial and the next three, we'll use both approaches. The `Categories` table already has a `Picture` column for storing the binary content of an image for the category. We'll add an additional column, `BrochurePath`, to store a path to a PDF on the web server s file system that can be used to provide a print-quality, polished overview of the category.

Step 3: Adding the `BrochurePath` Column to the `Categories` Table

Currently the Categories table has only four columns: `CategoryID`, `CategoryName`, `Description`, and `Picture`. In addition to these fields, we need to add a new one that will point to the category s brochure (if one exists). To add this column, go to the Server Explorer, drill down into the Tables, right-click on the `Categories` table and choose Open Table Definition (see Figure 5). If you do not see the Server Explorer, bring it up by selecting the Server Explorer option from the View menu, or hit Ctrl+Alt+S.

Add a new `varchar(200)` column to the `Categories` table that is named `BrochurePath` and allows `NULL` s and click the Save icon (or hit Ctrl+S).

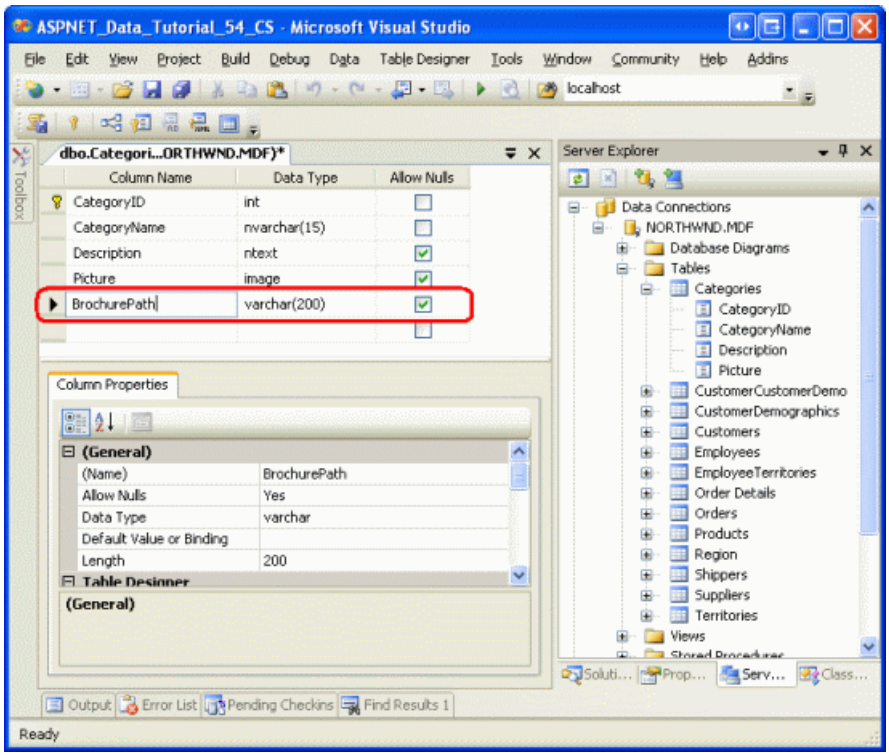


Figure 5: Add a `BrochurePath` Column to the `Categories` Table (Click to view full-size image)

Step 4: Updating the Architecture to Use the `Picture` and `BrochurePath` Columns

The `CategoriesDataTable` in the Data Access Layer (DAL) currently has four `DataColumn` s defined: `CategoryID` , `CategoryName` , `Description` , and `NumberOfProducts` . When we originally designed this DataTable in the Creating a Data Access Layer tutorial, the `CategoriesDataTable` only had the first three columns; the `NumberOfProducts` column was added in the Master/Detail Using a Bulleted List of Master Records with a Details DataList tutorial.

As discussed in *Creating a Data Access Layer*, the DataTables in the Typed DataSet make up the business objects. The TableAdapters are responsible for communicating with the database and populating the business objects with the query results. The `CategoriesDataTable` is populated by the `CategoriesTableAdapter` , which has three data retrieval methods:

- `GetCategories()` executes the TableAdapter s main query and returns the `CategoryID` , `CategoryName` , and `Description` fields of all records in the `Categories` table. The main query is what is used by the auto-generated `Insert` and `Update` methods.
- `GetCategoryByCategoryID(categoryID)` returns the `CategoryID` , `CategoryName` , and `Description` fields of the category whose `CategoryID` equals *categoryID*.
- `GetCategoriesAndNumberOfProducts()` - returns the `CategoryID` , `CategoryName` , and `Description` fields for all records in the `Categories` table. Also uses a subquery to return the number of products associated with each category.

Notice that none of these queries return the `Categories` table s `Picture` or `BrochurePath` columns; nor does the `CategoriesDataTable` provide `DataColumn` s for these fields. In order to work with the Picture and `BrochurePath` properties, we need to first add them to the `CategoriesDataTable` and then update the `CategoriesTableAdapter` class to return these columns.

Adding the `Picture` and `BrochurePath``DataColumn` s

Start by adding these two columns to the `CategoriesDataTable` . Right-click on the `CategoriesDataTable` s header,

select Add from the context menu and then choose the Column option. This will create a new `DataColumn` in the DataTable named `Column1`. Rename this column to `Picture`. From the Properties window, set the `DataColumn` s `DataType` property to `System.Byte[]` (this is not an option in the drop-down list; you need to type it in).

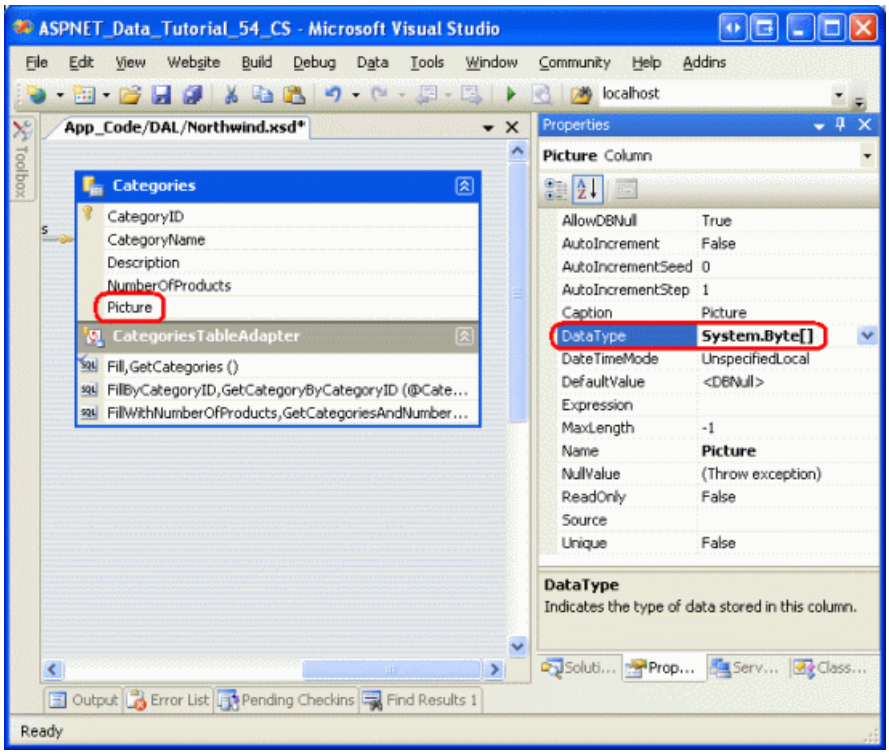


Figure 6: Create a `DataColumn` Named `Picture` whose `DataType` is `System.Byte[]` (Click to view full-size image)

Add another `DataColumn` to the DataTable, naming it `BrochurePath` using the default `DataType` value (`System.String`).

Returning the `Picture` and `BrochurePath` Values from the TableAdapter

With these two `DataColumn` s added to the `CategoriesDataTable`, we re ready to update the `CategoriesTableAdapter`. We could have both of these column values returned in the main TableAdapter query, but this would bring back the binary data every time the `GetCategories()` method was invoked. Instead, let s update the main TableAdapter query to bring back `BrochurePath` and create an additional data retrieval method that returns a particular category s `Picture` column.

To update the main TableAdapter query, right-click on the `CategoriesTableAdapter` s header and choose the Configure option from the context menu. This brings up the Table Adapter Configuration Wizard, which we ve seen in a number of past tutorials. Update the query to bring back the `BrochurePath` and click Finish.

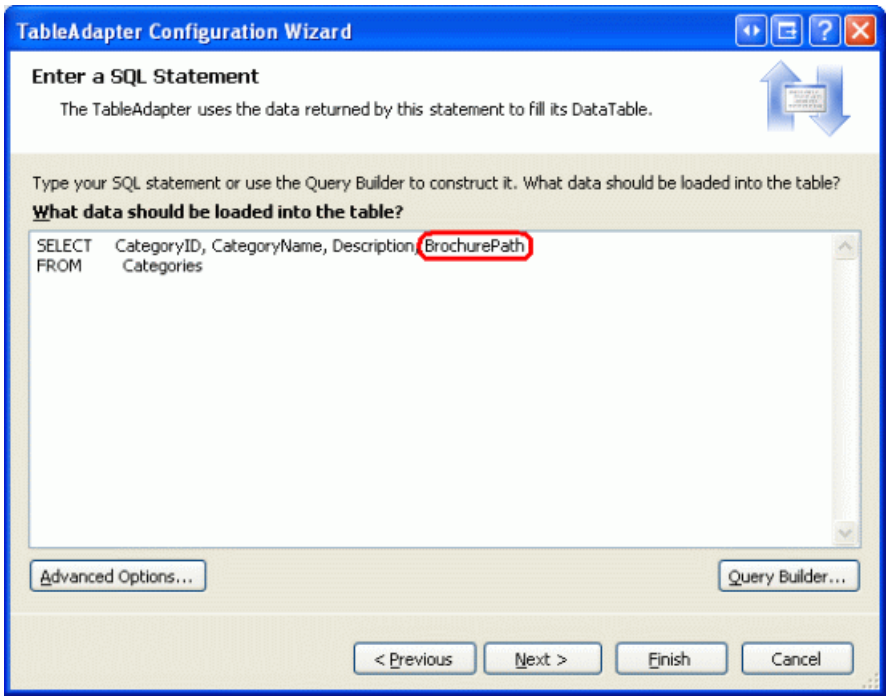


Figure 7: Update the Column List in the `SELECT` Statement to Also Return `BrochurePath` (Click to view full-size image)

When using ad-hoc SQL statements for the TableAdapter, updating the column list in the main query updates the column list for all of the `SELECT` query methods in the TableAdapter. That means the `GetCategoryByCategoryID(categoryID)` method has been updated to return the `BrochurePath` column, which might be what we intended. However, it also updated the column list in the `GetCategoriesAndNumberOfProducts()` method, removing the subquery that returns the number of products for each category! Therefore, we need to update this method s `SELECT` query. Right-click on the `GetCategoriesAndNumberOfProducts()` method, choose Configure, and revert the `SELECT` query back to its original value:

```
SELECT CategoryID, CategoryName, Description,
       (SELECT COUNT(*)
            FROM Products p
            WHERE p.CategoryID = c.CategoryID)
       as NumberOfProducts
FROM Categories c
```

Next, create a new TableAdapter method that returns a particular category s `Picture` column value. Right-click on the `CategoriesTableAdapter` s header and choose the Add Query option to launch the TableAdapter Query Configuration Wizard. The first step of this wizard asks us if we want to query data using an ad-hoc SQL statement, a new stored procedure, or an existing one. Select Use SQL statements and click Next. Since we will be returning a row, choose the SELECT which returns rows option from the second step.

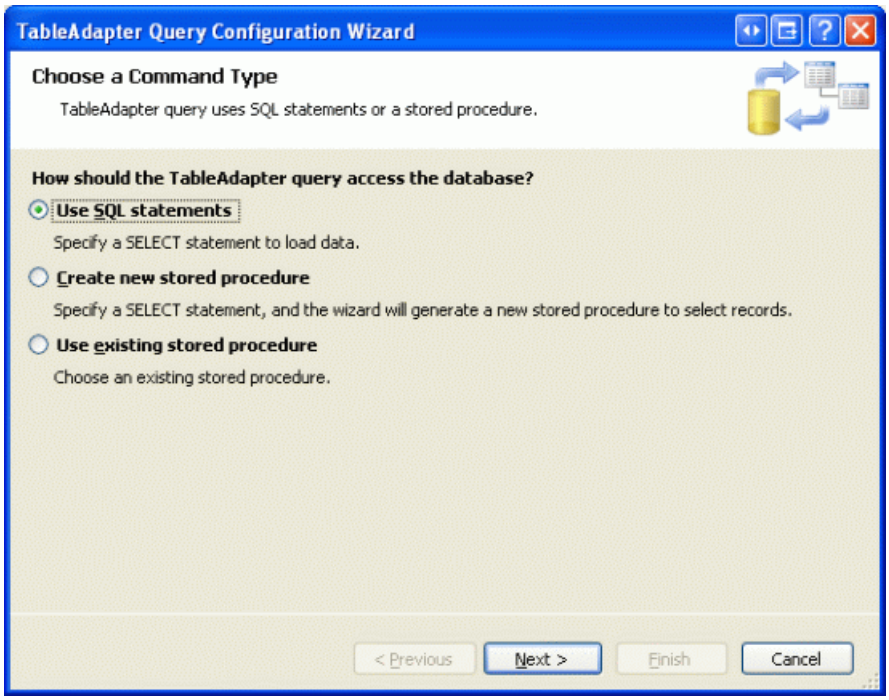


Figure 8: Select the Use SQL statements Option (Click to view full-size image)

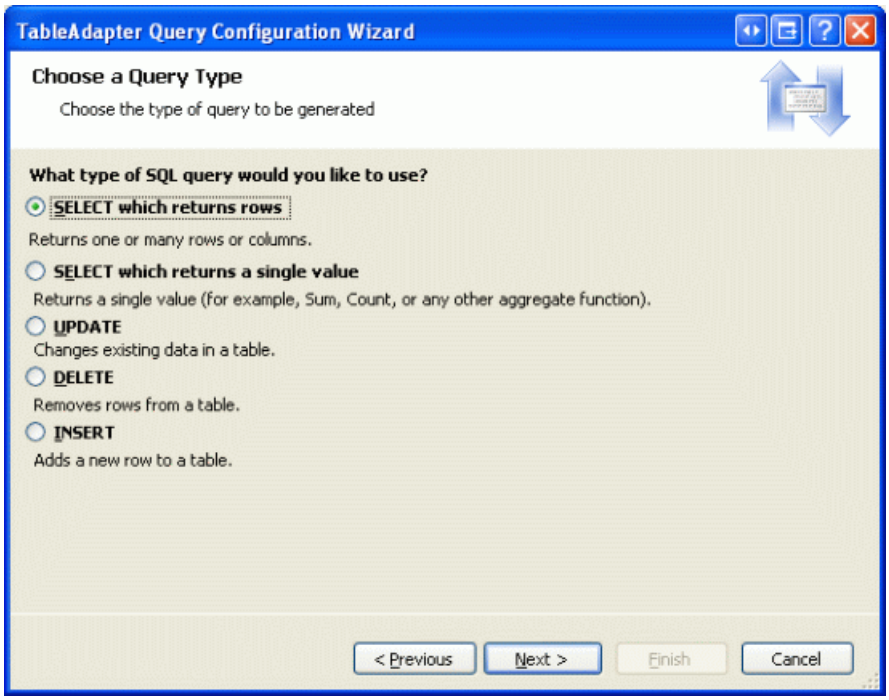


Figure 9: Since the Query Will Return a Record from the Categories Table, Choose SELECT which returns rows (Click to view full-size image)

In the third step, enter the following SQL query and click Next:

```
SELECT     CategoryID, CategoryName, Description, BrochurePath, Picture
FROM       Categories
WHERE      CategoryID = @CategoryID
```

The last step is to choose the name for the new method. Use `FillCategoryWithBinaryDataByCategoryID` and `GetCategoryWithBinaryDataByCategoryID` for the Fill a DataTable and Return a DataTable patterns, respectively. Click Finish to complete the wizard.

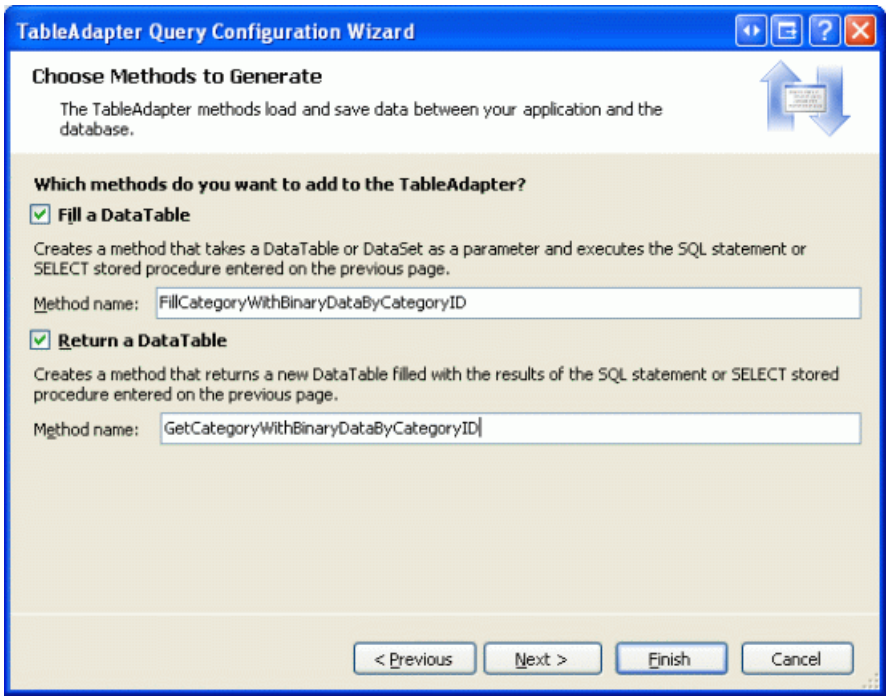


Figure 10: Choose the Names for the TableAdapter s Methods (Click to view full-size image)

> **NOTE**
>
> After completing the Table Adapter Query Configuration Wizard you may see a dialog box informing you that the new command text returns data with schema different from the schema of the main query. In short, the wizard is noting that the TableAdapter s main query `GetCategories()` returns a different schema than the one we just created. But this is what we want, so you can disregard this message.

Also, keep in mind that if you are using ad-hoc SQL statements and use the wizard to change the TableAdapter s main query at some later point in time, it will modify the `GetCategoryWithBinaryDataByCategoryID` method s `SELECT` statement s column list to include just those columns from the main query (that is, it will remove the `Picture` column from the query). You will have to manually update the column list to return the `Picture` column, similar to what we did with the `GetCategoriesAndNumberOfProducts()` method earlier in this step.

After adding the two `DataColumn` s to the `CategoriesDataTable` and the `GetCategoryWithBinaryDataByCategoryID` method to the `CategoriesTableAdapter`, these classes in the Typed DataSet Designer should look like the screenshot in Figure 11.

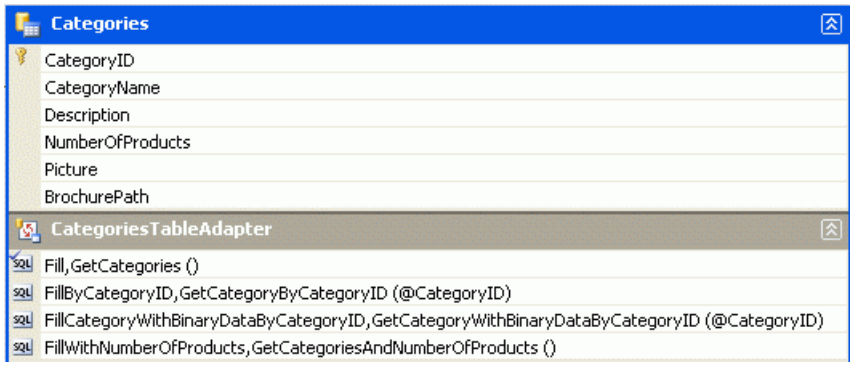


Figure 11: The DataSet Designer Includes the New Columns and Method

Updating the Business Logic Layer (BLL)

With the DAL updated, all that remains is to augment the Business Logic Layer (BLL) to include a method for the new `CategoriesTableAdapter` method. Add the following method to the `CategoriesBLL` class:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Select, false)]
public Northwind.CategoriesDataTable
    GetCategoryWithBinaryDataByCategoryID(int categoryID)
{
    return Adapter.GetCategoryWithBinaryDataByCategoryID(categoryID);
}
```

Step 5: Uploading a File From the Client to the Web Server

When collecting binary data, oftentimes this data is supplied by an end user. To capture this information, the user needs to be able to upload a file from their computer to the web server. The uploaded data then needs to be integrated with the data model, which may mean saving the file to the web server s file system and adding a path to the file in the database, or writing the binary contents directly into the database. In this step we'll look at how to allow a user to upload files from their computer to the server. In the next tutorial we'll turn our attention to integrating the uploaded file with data model.

ASP.NET 2.0 s new FileUpload Web control provides a mechanism for users to send a file from their computer to the web server. The FileUpload control renders as an `<input>` element whose `type` attribute is set to file, which browsers display as a textbox with a Browse button. Clicking the Browse button brings up a dialog box from which the user can select a file. When the form is posted back, the selected file s contents are sent along with the postback. On the server-side, information about the uploaded file is accessible through the FileUpload control s properties.

To demonstrate uploading files, open the `FileUpload.aspx` page in the `BinaryData` folder, drag a FileUpload control from the Toolbox onto the Designer, and set the control s `ID` property to `UploadTest`. Next, add a Button Web control setting its `ID` and `Text` properties to `UploadButton` and Upload Selected File, respectively. Finally, place a Label Web control beneath the Button, clear out its `Text` property and set its `ID` property to `UploadDetails`.

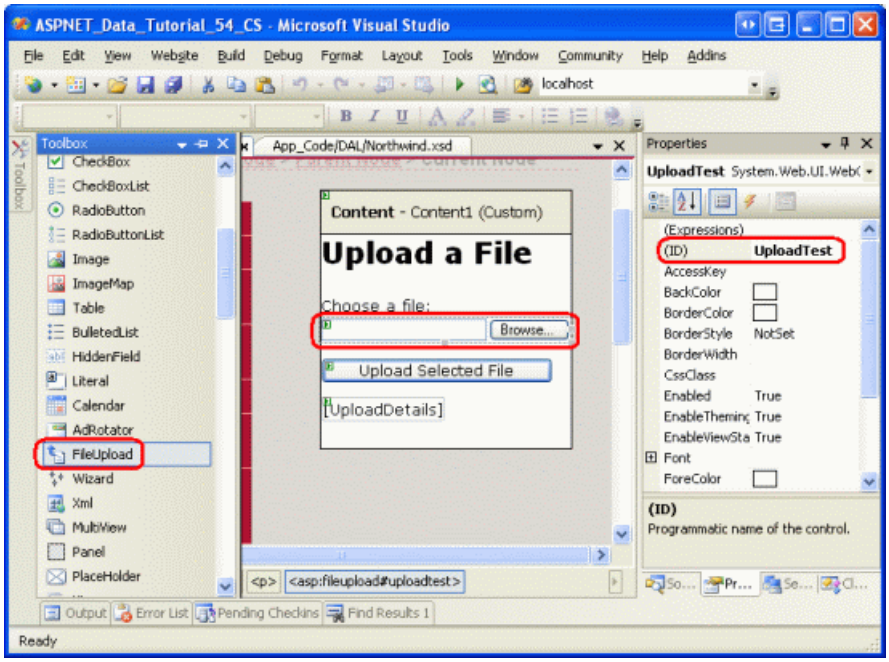


Figure 12: Add a FileUpload Control to the ASP.NET Page (Click to view full-size image)

Figure 13 shows this page when viewed through a browser. Note that clicking the Browse button brings up a file selection dialog box, allowing the user to pick a file from their computer. Once a file has been selected, clicking the Upload Selected File button causes a postback that sends the selected file s binary content to the web server.

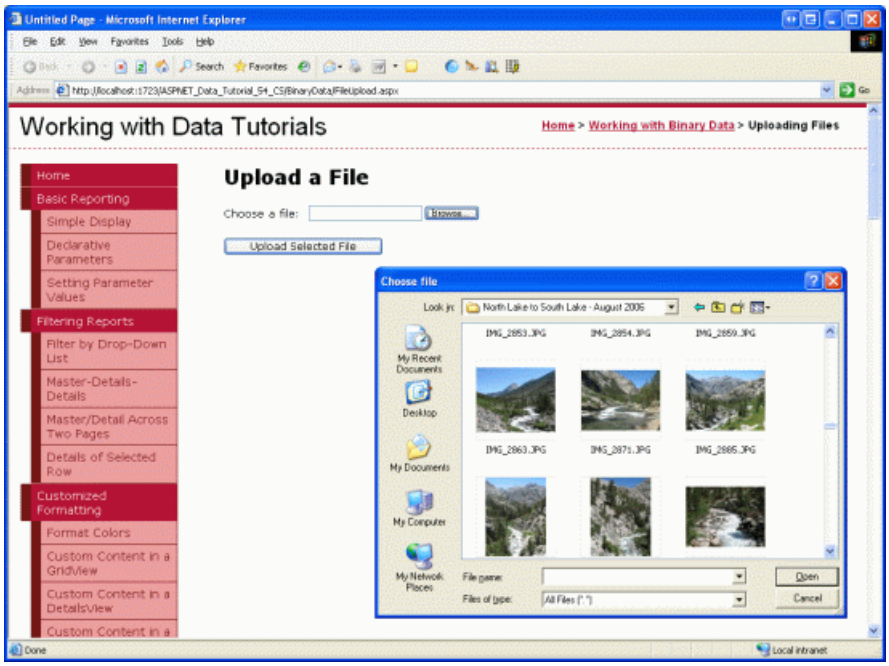


Figure 13: The User Can Select a File to Upload from their Computer to the Server (Click to view full-size image)

On postback, the uploaded file can be saved to the file system or its binary data can be worked with directly through a Stream. For this example, let s create a `~/Brochures` folder and save the uploaded file there. Start by adding the `Brochures` folder to the site as a subfolder of the root directory. Next, create an event handler for the `UploadButton` s `Click` event and add the following code:

```
protected void UploadButton_Click(object sender, EventArgs e)
{
    if (UploadTest.HasFile == false)
    {
        // No file uploaded!
        UploadDetails.Text = "Please first select a file to upload...";
    }
    else
    {
        // Display the uploaded file's details
        UploadDetails.Text = string.Format(
                @"Uploaded file: {0}<br />
                  File size (in bytes): {1:N0}<br />
                  Content-type: {2}",
                  UploadTest.FileName,
                  UploadTest.FileBytes.Length,
                  UploadTest.PostedFile.ContentType);
        // Save the file
        string filePath =
            Server.MapPath("~/Brochures/" + UploadTest.FileName);
        UploadTest.SaveAs(filePath);
    }
}
```

The FileUpload control provides a variety of properties for working with the uploaded data. For instance, the `HasFile` property indicates whether a file was uploaded by the user, while the `FileBytes` property provides access to the uploaded binary data as an array of bytes. The `Click` event handler starts by ensuring that a file has been uploaded. If a file has been uploaded, the Label shows the name of the uploaded file, its size in bytes, and its content-type.

> **NOTE**
> To ensure that the user uploads a file you can check the `HasFile` property and display a warning if it s `false`, or you may use the RequiredFieldValidator control instead.

The FileUpload s `SaveAs(filePath)` saves the uploaded file to the specified *filePath*. *filePath* must be a *physical path* (`C:\Websites\Brochures\SomeFile.pdf`) rather than a *virtual path* (`/Brochures/SomeFile.pdf`). The `Server.MapPath(virtPath)` method takes a virtual path and returns its corresponding physical path. Here, the virtual path is `~/Brochures/fileName`, where *fileName* is the name of the uploaded file. See Using Server.MapPath for more information on virtual and physical paths and using `Server.MapPath`.

After completing the `Click` event handler, take a moment to test out the page in a browser. Click the Browse button and select a file from your hard drive and then click the Upload Selected File button. The postback will send the contents of the selected file to the web server, which will then display information about the file before saving it to the `~/Brochures` folder. After uploading the file, return to Visual Studio and click the Refresh button in the Solution Explorer. You should see the file you just uploaded in the ~/Brochures folder!

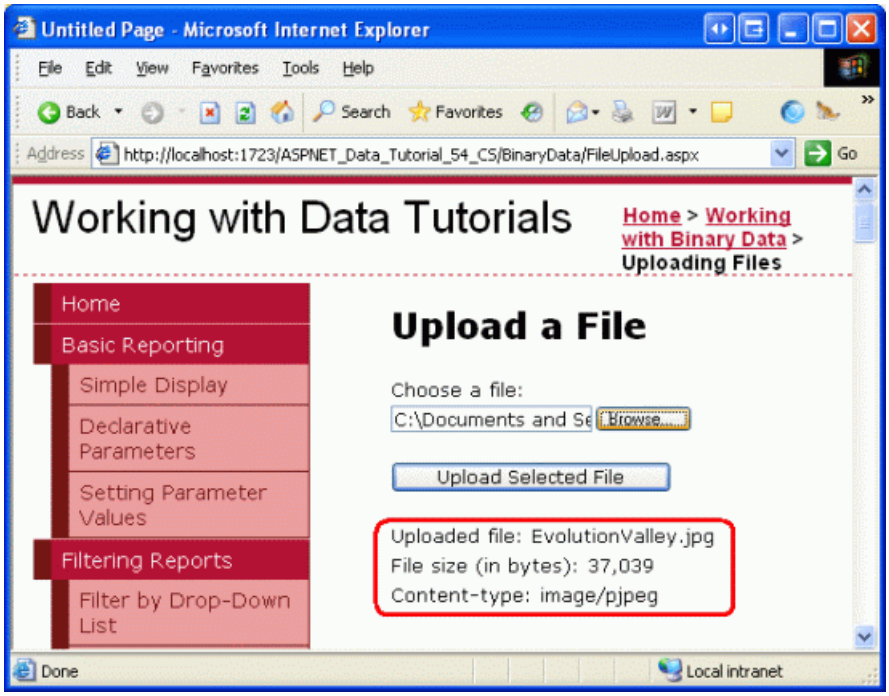


Figure 14: The File `EvolutionValley.jpg` Has Been Uploaded to the Web Server (Click to view full-size image)

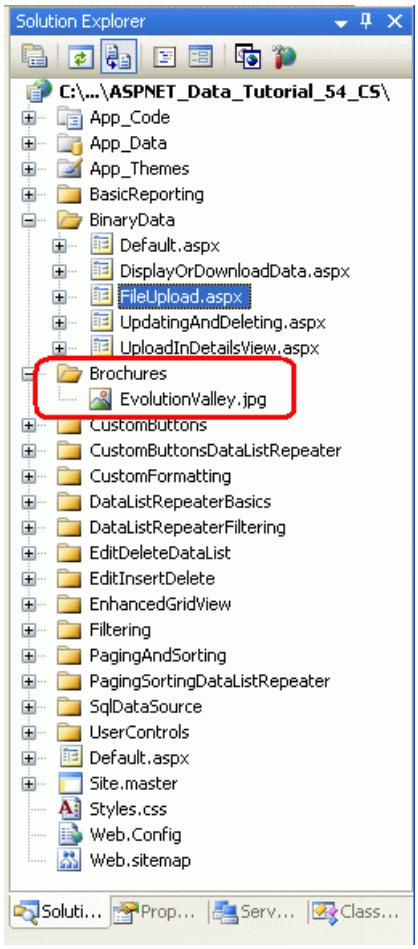


Figure 15: `EvolutionValley.jpg` Was Saved to the `~/Brochures` Folder

Subtleties with Saving Uploaded Files to the File System

There are several subtleties that must be addressed when saving uploading files to the web server s file system. First, there s the issue of security. To save a file to the file system, the security context under which the ASP.NET page is executing must have Write permissions. The ASP.NET Development Web Server runs under the context of your current user account. If you are using Microsoft s Internet Information Services (IIS) as the web server, the security context depends on the version of IIS and its configuration.

Another challenge of saving files to the file system revolves around naming the files. Currently, our page saves all of the uploaded files to the `~/Brochures` directory using the same name as the file on the client s computer. If User A uploads a brochure with the name `Brochure.pdf`, the file will be saved as `~/Brochure/Brochure.pdf`. But what if sometime later User B uploads a different brochure file that happens to have the same filename (`Brochure.pdf`)? With the code we have now, User A s file will be overwritten with User B s upload.

There are a number of techniques for resolving file name conflicts. One option is to prohibit uploading a file if there already exists one with the same name. With this approach, when User B attempts to upload a file named `Brochure.pdf`, the system would not save their file and instead display a message informing User B to rename the file and try again. Another approach is to save the file using a unique file name, which could be a globally unique identifier (GUID) or the value from the corresponding database record s primary key column(s) (assuming that the upload is associated with a particular row in the data model). In the next tutorial we'll explore these options in more detail.

Challenges Involved with Very Large Amounts of Binary Data

These tutorials assume that the binary data captured is modest in size. Working with very large amounts of binary data files that are several megabytes or larger introduces new challenges that are beyond the scope of these tutorials. For example, by default ASP.NET will reject uploads of more than 4 MB, although this can be

configured through the `<httpRuntime>` element in `Web.config`. IIS imposes its own file upload size limitations, too. See IIS Upload File Size for more information. Furthermore, the time taken to upload large files might exceed the default 110 seconds ASP.NET will wait for a request. There are also memory and performance issues that arise when working with large files.

The FileUpload control is impractical for large file uploads. As the file s contents are being posted to the server, the end user must patiently wait without any confirmation that their upload is progressing. This is not so much an issue when dealing with smaller files that can be uploaded in a few seconds, but can be an issue when dealing with larger files that may take minutes to upload. There are a variety of third-party file upload controls that are better suited for handling large uploads and many of these vendors provide progress indicators and ActiveX upload managers that present a much more polished user experience.

If your application needs to handle large files, you'll need to carefully investigate the challenges and find suitable solutions for your particular needs.

Summary

Building an application that needs to capture binary data introduces a number of challenges. In this tutorial we explored the first two: deciding where to store the binary data and allowing a user to upload binary content through a web page. Over the next three tutorials, we'll see how to associate the uploaded data with a record in the database as well as how to display the binary data alongside its text data fields.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Using Large-Value Data Types
- FileUpload Control QuickStarts
- The ASP.NET 2.0 FileUpload Server Control
- The Dark Side of File Uploads

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Teresa Murphy and Bernadette Leigh. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Displaying Binary Data in the Data Web Controls (C#)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we look at the options to present binary data on a Web page, including the display of an image file and the provision of a 'Download' link for a PDF file.

Introduction

In the preceding tutorial we explored the two techniques for associating binary data with an application s underlying data model and used the FileUpload control to upload files from a browser to the web server s file system. We ve yet to see how to associate the uploaded binary data with the data model. That is, after a file has been uploaded and saved to the file system, a path to the file must be stored in the appropriate database record. If the data is being stored directly in the database, then the uploaded binary data need not be saved to the file system, but must be injected into the database.

Before we look at associating the data with the data model, though, let s first look at how to provide the binary data to the end user. Presenting text data is simple enough, but how should binary data be presented? It depends, of course, on the type of binary data. For images, we likely want to display the image; for PDFs, Microsoft Word documents, ZIP files, and other types of binary data, providing a Download link is probably more appropriate.

In this tutorial we will look at how to present the binary data alongside its associated text data using data Web controls like the GridView and DetailsView. In the next tutorial we'll turn our attention to associating an uploaded file with the database.

Step 1: Providing `BrochurePath` Values

The `Picture` column in the `Categories` table already contains binary data for the various category images. Specifically, the `Picture` column for each record holds the binary contents of a grainy, low-quality, 16-color bitmap image. Each category image is 172 pixels wide and 120 pixels tall and consumes roughly 11 KB. What s more, the binary content in the `Picture` column includes a 78-byte OLE header that must be stripped before displaying the image. This header information is present because the Northwind database has its roots in Microsoft Access. In Access, binary data is stored using the OLE Object data type, which tacks on this header. For now, we'll see how to strip the headers from these low-quality images in order to display the picture. In a future tutorial we'll build an interface for updating a category s `Picture` column and replace these bitmap images that use OLE headers with equivalent JPG images without the unnecessary OLE headers.

In the preceding tutorial we saw how to use the FileUpload control. Therefore, you can go ahead and add brochure files to the web server s file system. Doing so, however, does not update the `BrochurePath` column in the `Categories` table. In the next tutorial we'll see how to accomplish this, but for now we need to manually provide values for this column.

In this tutorial s download you'll find seven PDF brochure files in the `~/Brochures` folder, one for each of the categories except Seafood. I purposefully omitted adding a Seafood brochure to illustrate how to handle scenarios where not all records have associated binary data. To update the `Categories` table with these values, right-click on

the `Categories` node from Server Explorer and choose Show Table Data. Then, enter the virtual paths to the brochure files for each category that has a brochure, as Figure 1 shows. Since there is no brochure for the Seafood category, leave its `BrochurePath` column s value as `NULL`.

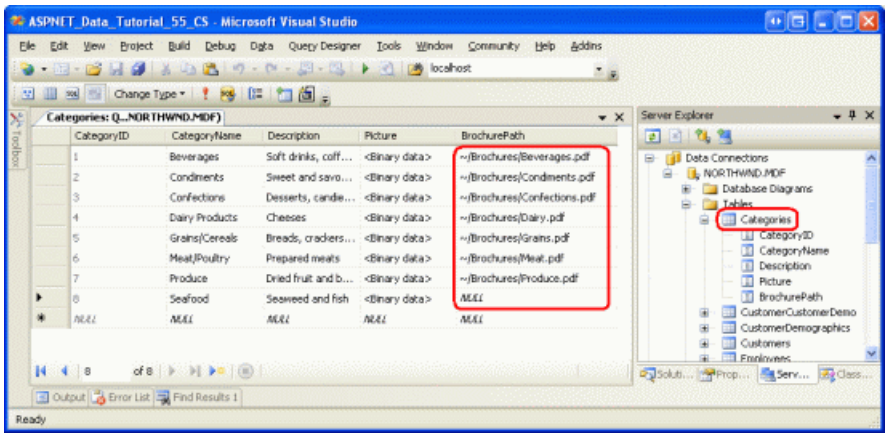


Figure 1: Manually Enter the Values for the `Categories` Table s `BrochurePath` Column (Click to view full-size image)

Step 2: Providing a Download Link for the Brochures in a GridView

With the `BrochurePath` values provided for the `Categories` table, we re ready to create a GridView that lists each category along with a link to download the category s brochure. In Step 4 we'll extend this GridView to also display the category s image.

Start by dragging a GridView from the Toolbox onto the Designer of the `DisplayOrDownloadData.aspx` page in the `BinaryData` folder. Set the GridView s `ID` to `Categories` and through the GridView s smart tag, choose to bind it to a new data source. Specifically, bind it to an ObjectDataSource named `CategoriesDataSource` that retrieves data using the `CategoriesBLL` object s `GetCategories()` method.

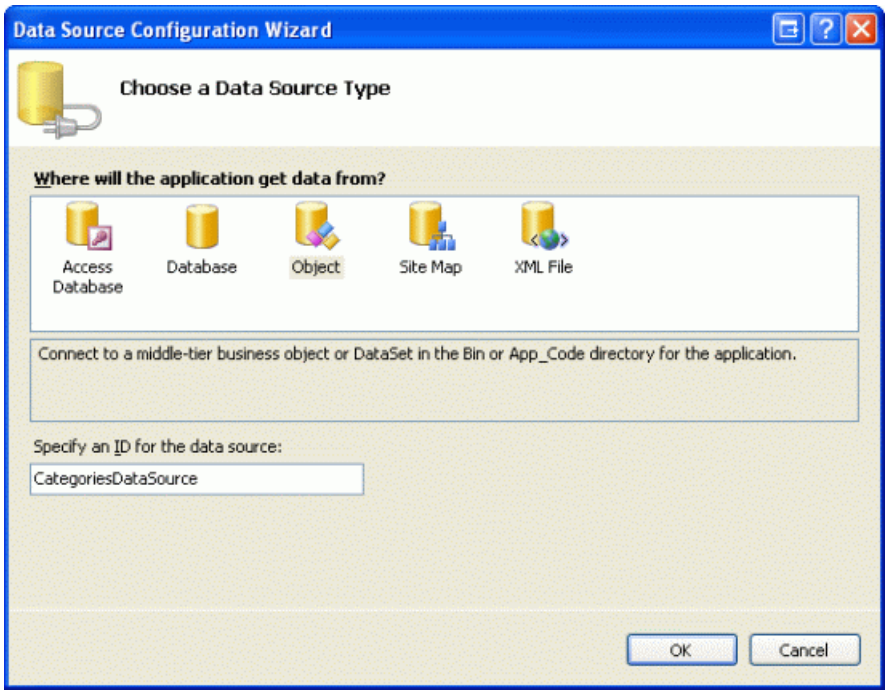


Figure 2: Create a New ObjectDataSource Named `CategoriesDataSource` (Click to view full-size image)

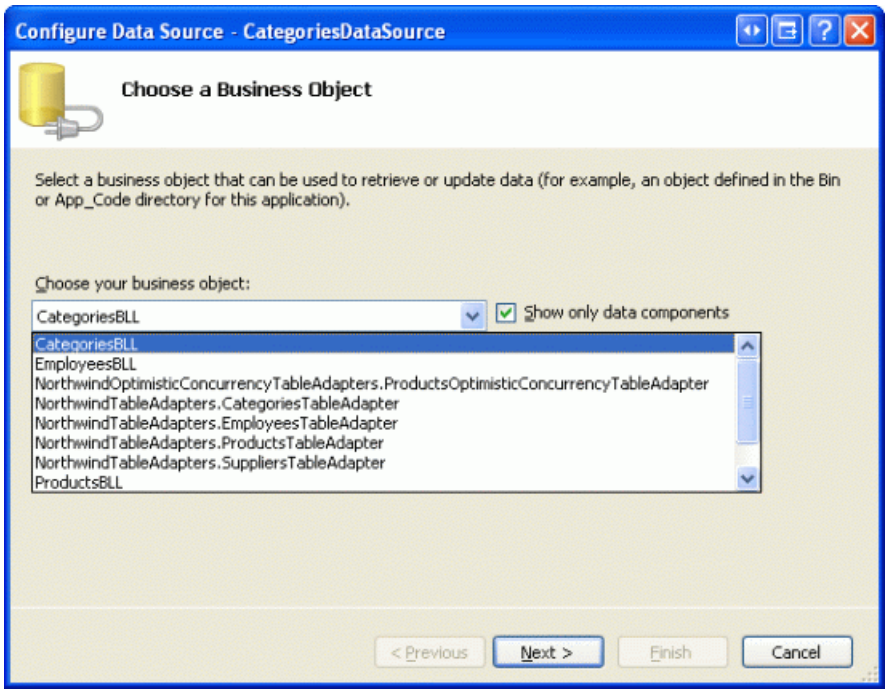


Figure 3: Configure the ObjectDataSource to Use the `CategoriesBLL` Class (Click to view full-size image)

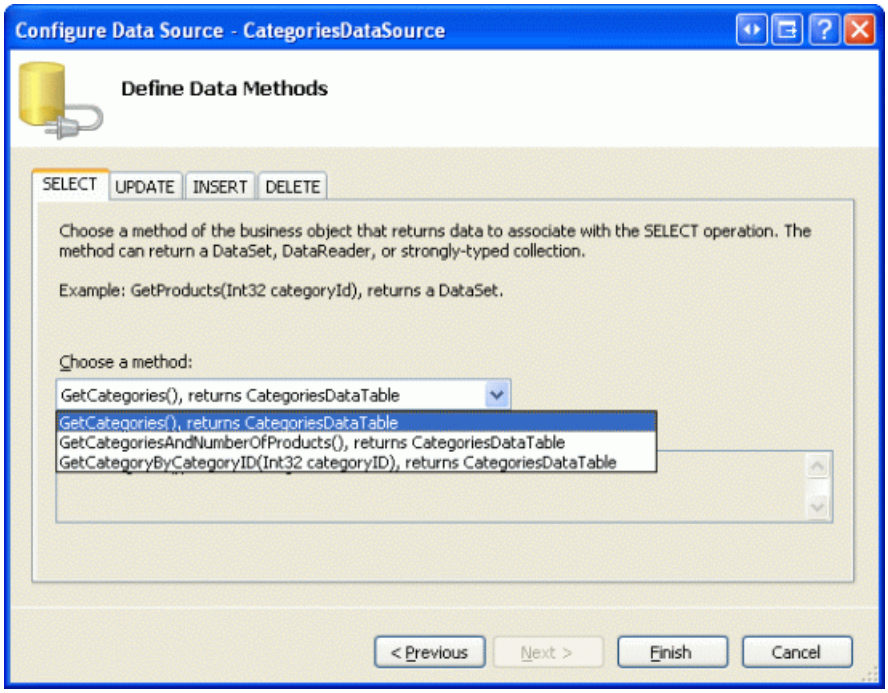


Figure 4: Retrieve the List of Categories Using the `GetCategories()` Method (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will automatically add a BoundField to the `Categories` GridView for the `CategoryID`, `CategoryName`, `Description`, `NumberOfProducts`, and `BrochurePath` `DataColumn` s. Go ahead and remove the `NumberOfProducts` BoundField since the `GetCategories()` method s query does not retrieve this information. Also remove the `CategoryID` BoundField and rename the `CategoryName` and `BrochurePath` BoundFields `HeaderText` properties to Category and Brochure, respectively. After making these changes, your GridView and ObjectDataSource s declarative markup should look like the following:

```
<asp:GridView ID="Categories" runat="server"
    AutoGenerateColumns="False" DataKeyNames="CategoryID"
    DataSourceID="CategoriesDataSource" EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            SortExpression="CategoryName" />
        <asp:BoundField DataField="Description" HeaderText="Description"
            SortExpression="Description" />
        <asp:BoundField DataField="BrochurePath" HeaderText="Brochure"
            SortExpression="BrochurePath" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

View this page through a browser (see Figure 5). Each of the eight categories is listed. The seven categories with `BrochurePath` values have the `BrochurePath` value displayed in the respective BoundField. Seafood, which has a `NULL` value for its `BrochurePath`, displays an empty cell.

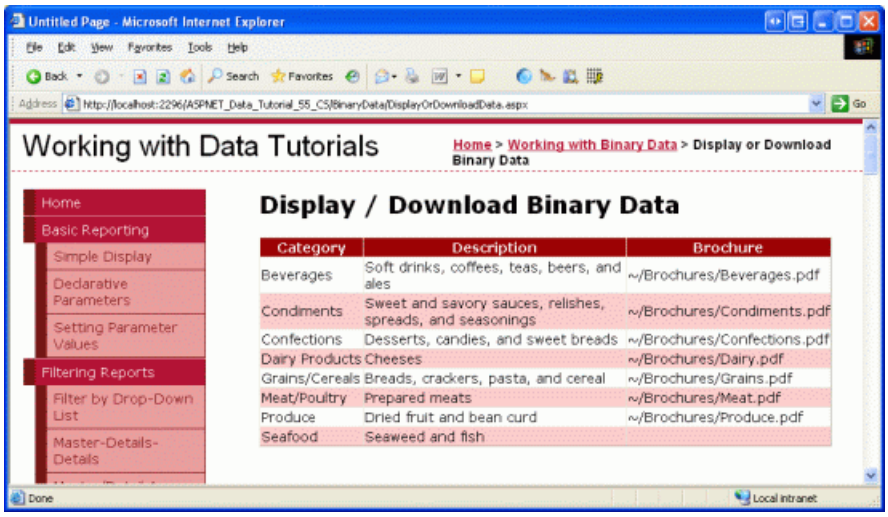


Figure 5: Each Category s Name, Description, and `BrochurePath` Value is Listed (Click to view full-size image)

Rather than displaying the text of the `BrochurePath` column, we want to create a link to the brochure. To accomplish this, remove the `BrochurePath` BoundField and replace it with a HyperLinkField. Set the new HyperLinkField s `HeaderText` property to Brochure, its `Text` property to View Brochure, and its `DataNavigateUrlFields` property to `BrochurePath`.

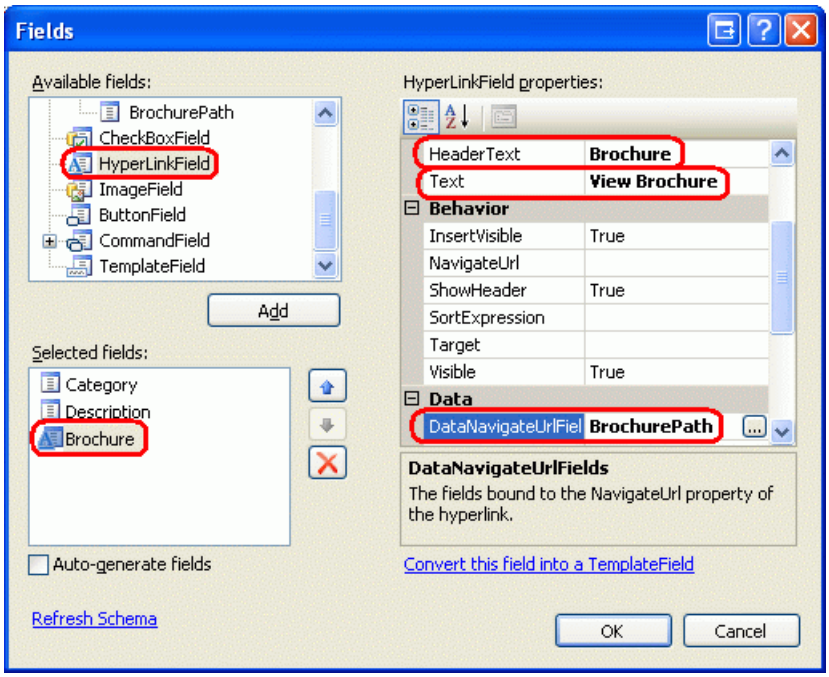


Figure 6: Add a HyperLinkField for `BrochurePath`

This will add a column of links to the GridView, as Figure 7 shows. Clicking a View Brochure link will either display the PDF directly in the browser or prompt the user to download the file, depending on whether a PDF reader is installed and the browser s settings.

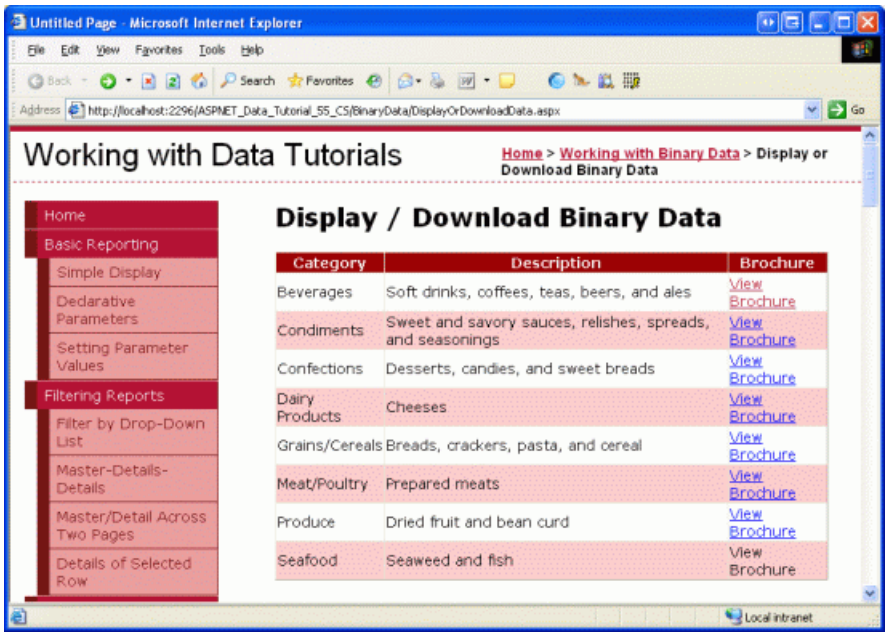


Figure 7: A Category s Brochure Can Be Viewed by Clicking the View Brochure Link (Click to view full-size image)

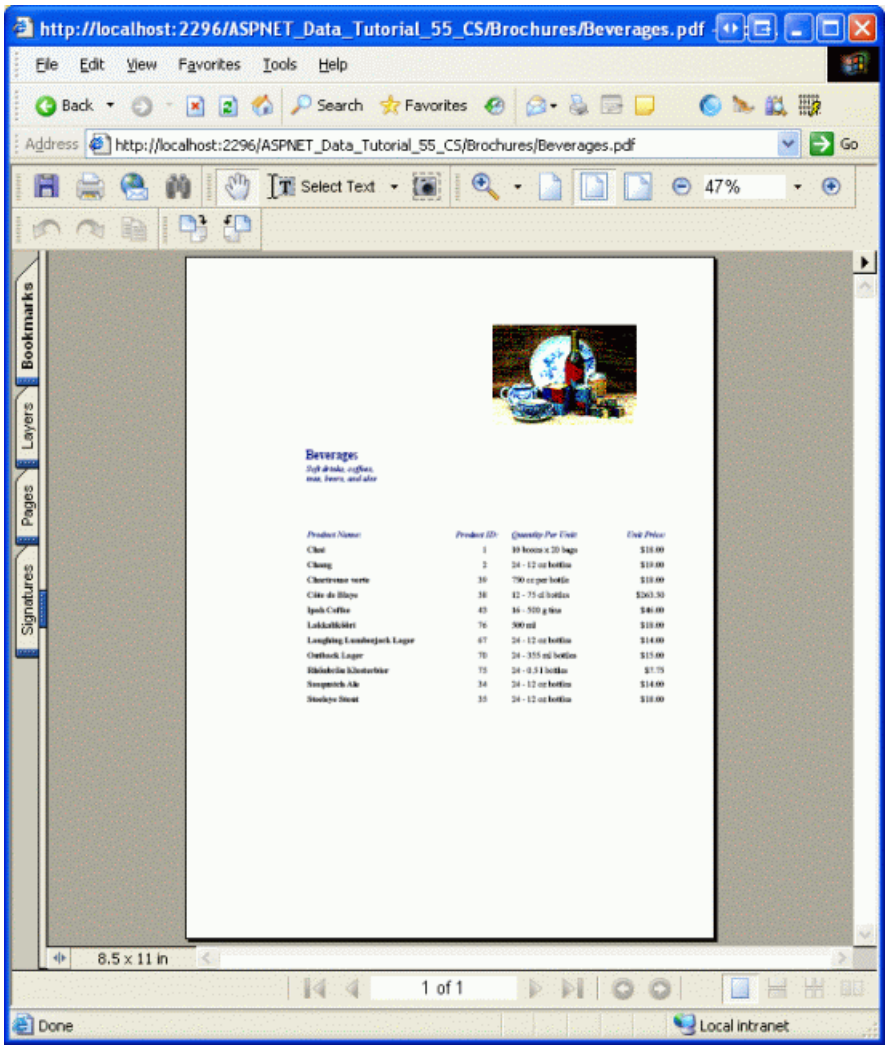


Figure 8: The Category s Brochure PDF is Displayed (Click to view full-size image)

Hiding the View Brochure Text for Categories Without a Brochure

As Figure 7 shows, the `BrochurePath` HyperLinkField displays its `Text` property value (View Brochure) for all records, regardless of whether there s a non- `NULL` value for `BrochurePath`. Of course, if `BrochurePath` is `NULL`, then the link is displayed as text only, as is the case with the Seafood category (refer back to Figure 7). Rather than displaying the text View Brochure, it might be nice to have those categories without a `BrochurePath` value display some alternate text, like No Brochure Available.

In order to provide this behavior, we need to use a TemplateField whose content is generated via a call to a page method that emits the appropriate output based on the `BrochurePath` value. We first explored this formatting technique back in the Using TemplateFields in the GridView Control tutorial.

Turn the HyperLinkField into a TemplateField by selecting the `BrochurePath` HyperLinkField and then clicking on the Convert this field into a TemplateField link in the Edit Columns dialog box.

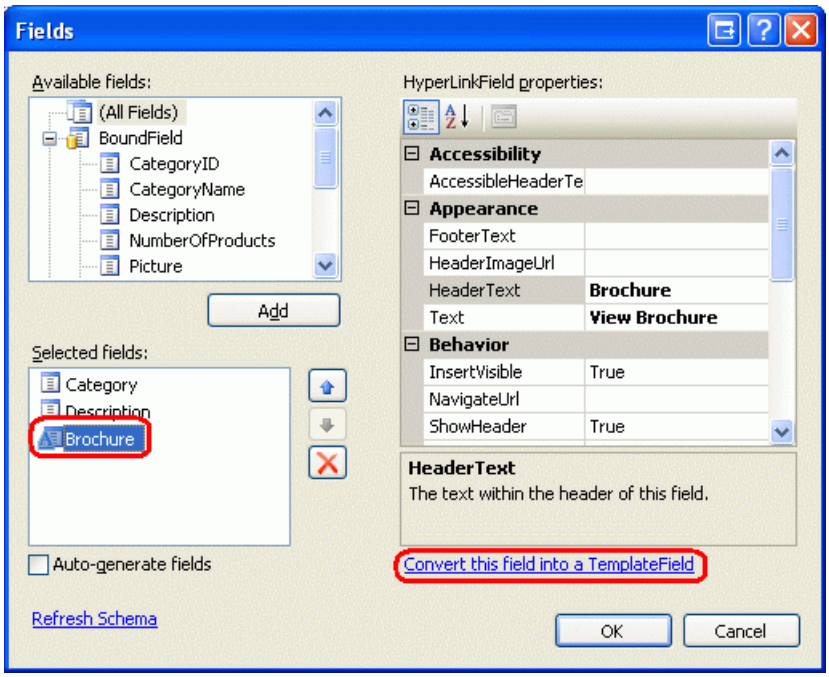


Figure 9: Convert the HyperLinkField into a TemplateField

This will create a TemplateField with an `ItemTemplate` that contains a HyperLink Web control whose `NavigateUrl` property is bound to the `BrochurePath` value. Replace this markup with a call to the method `GenerateBrochureLink`, passing in the value of `BrochurePath`:

```
<asp:TemplateField HeaderText="Brochure">
    <ItemTemplate>
        <%# GenerateBrochureLink(Eval("BrochurePath")) %>
    </ItemTemplate>
</asp:TemplateField>
```

Next, create a `protected` method in the ASP.NET page s code-behind class named `GenerateBrochureLink` that returns a `string` and accepts an `object` as an input parameter.

```
protected string GenerateBrochureLink(object BrochurePath)
{
    if (Convert.IsDBNull(BrochurePath))
        return "No Brochure Available";
    else
        return string.Format(@"<a href="{0}">View Brochure</a>",
            ResolveUrl(BrochurePath.ToString()));
}
```

This method determines if the passed-in `object` value is a database `NULL` and, if so, returns a message indicating that the category lacks a brochure. Otherwise, if there is a `BrochurePath` value, it s displayed in a hyperlink. Note that if the `BrochurePath` value is present it s passed into the `ResolveUrl(url)` method. This method resolves the passed-in *url*, replacing the `~` character with the appropriate virtual path. For example, if the application is rooted at `/Tutorial55`, `ResolveUrl("~/Brochures/Meats.pdf")` will return `/Tutorial55/Brochures/Meat.pdf`.

Figure 10 shows the page after these changes have been applied. Note that the Seafood category s `BrochurePath` field now displays the text No Brochure Available .

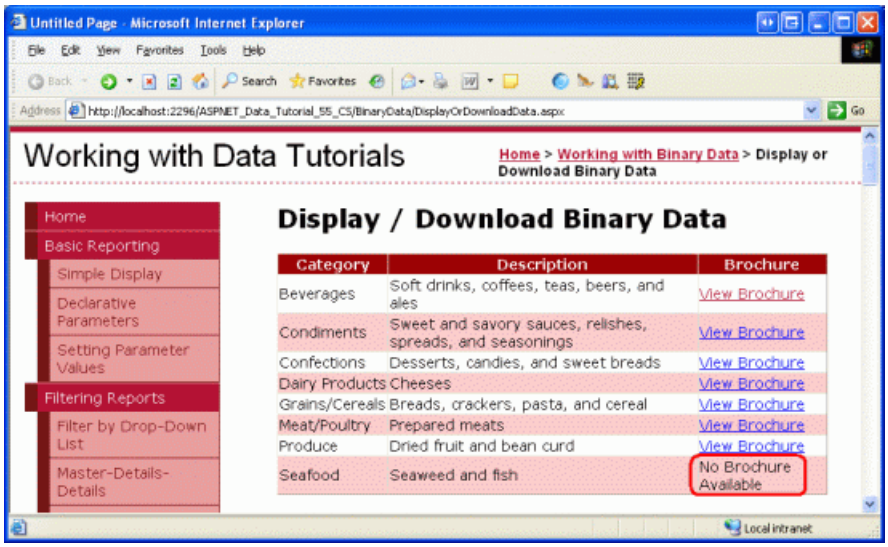


Figure 10: The Text No Brochure Available is Displayed for Those Categories Without a Brochure (Click to view full-size image)

Step 3: Adding a Web Page to Display a Category s Picture

When a user visits an ASP.NET page, they receive the ASP.NET page s HTML. The received HTML is just text and does not contain any binary data. Any additional binary data, such as images, sound files, Macromedia Flash applications, embedded Windows Media Player videos, and so forth, exist as separate resources on the web server. The HTML contains references to these files, but does not include the actual contents of the files.

For example, in HTML the `<img>` element is used to reference a picture, with the `src` attribute pointing to the image file like so:

```
<img src="MyPicture.jpg" ... />
```

When a browser receives this HTML, it makes another request to the web server to retrieve the binary contents of the image file, which it then displays in the browser. The same concept applies to any binary data. In Step 2, the brochure was not sent down to the browser as part of the page s HTML markup. Rather, the rendered HTML provided hyperlinks that, when clicked, caused the browser to request the PDF document directly.

To display or allow users to download binary data that resides within the database, we need to create a separate web page that returns the data. For our application, there s only one binary data field stored directly in the database the category s picture. Therefore, we need a page that, when called, returns the image data for a particular category.

Add a new ASP.NET page to the `BinaryData` folder named `DisplayCategoryPicture.aspx`. When doing so, leave the Select master page checkbox unchecked. This page expects a `CategoryID` value in the querystring and returns the binary data of that category s `Picture` column. Since this page returns binary data and nothing else, it does not need any markup in the HTML section. Therefore, click on the Source tab in the lower left corner and remove all of the page s markup except for the `<%@ Page %>` directive. That is, `DisplayCategoryPicture.aspx` s declarative markup should consist of a single line:

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="DisplayCategoryPicture.aspx.cs"
    Inherits="BinaryData_DisplayCategoryPicture" %>
```

If you see the `MasterPageFile` attribute in the `<%@ Page %>` directive, remove it.

In the page s code-behind class, add the following code to the `Page_Load` event handler:

```
protected void Page_Load(object sender, EventArgs e)
{
    int categoryID = Convert.ToInt32(Request.QueryString["CategoryID"]);
    // Get information about the specified category
    CategoriesBLL categoryAPI = new CategoriesBLL();
    Northwind.CategoriesDataTable categories =
        categoryAPI.GetCategoryWithBinaryDataByCategoryID(categoryID);
    Northwind.CategoriesRow category = categories[0];
    // Output HTTP headers providing information about the binary data
    Response.ContentType = "image/bmp";
    // Output the binary data
    // But first we need to strip out the OLE header
    const int OleHeaderLength = 78;
    int strippedImageLength = category.Picture.Length - OleHeaderLength;
    byte[] strippedImageData = new byte[strippedImageLength];
    Array.Copy(category.Picture, OleHeaderLength,
        strippedImageData, 0, strippedImageLength);

    Response.BinaryWrite(strippedImageData);
}
```

This code starts by reading in the `CategoryID` querystring value into a variable named `categoryID`. Next, the picture data is retrieved via a call to the `CategoriesBLL` class s `GetCategoryWithBinaryDataByCategoryID(categoryID)` method. This data is returned to the client by using the `Response.BinaryWrite(data)` method, but before this is called, the `Picture` column value s OLE header must be removed. This is accomplished by creating a `byte` array named `strippedImageData` that will hold precisely 78 characters less than what is in the `Picture` column. The `Array.Copy` method is used to copy the data from `category.Picture` starting at position 78 over to `strippedImageData`.

The `Response.ContentType` property specifies the MIME type of the content being returned so that the browser knows how to render it. Since the `Categories` table s `Picture` column is a bitmap image, the bitmap MIME type is used here (image/bmp). If you omit the MIME type, most browsers will still display the image correctly because they can infer the type based on the contents of the image file s binary data. However, it s prudent to include the MIME type when possible. See the Internet Assigned Numbers Authority s website for a complete listing of MIME media types.

With this page created, a particular category s picture can be viewed by visiting `DisplayCategoryPicture.aspx?CategoryID=categoryID`. Figure 11 shows the Beverages category s picture, which can be viewed from `DisplayCategoryPicture.aspx?CategoryID=1`.

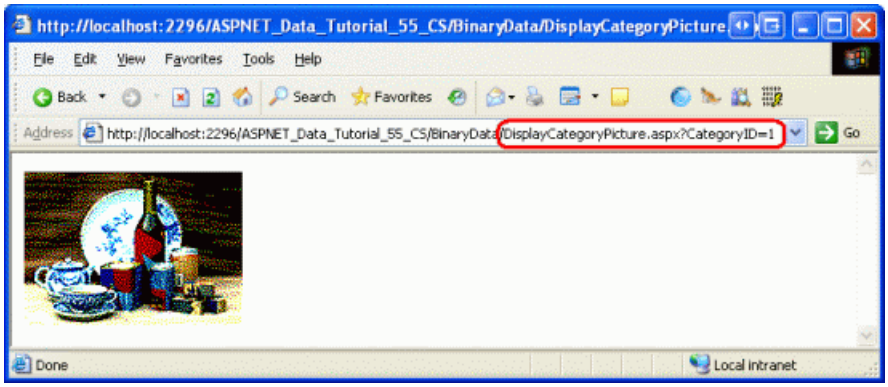


Figure 11: The Beverages Category s Picture is Displayed (Click to view full-size image)

If, when visiting `DisplayCategoryPicture.aspx?CategoryID=categoryID`, you get an exception that reads Unable to cast object of type 'System.DBNull' to type 'System.Byte[]', there are two things that may be causing this. First, the `Categories` table s `Picture` column does allow `NULL` values. The `DisplayCategoryPicture.aspx` page, however, assumes there is a non-`NULL` value present. The `Picture` property of the `CategoriesDataTable` cannot be directly

accessed if it has a `NULL` value. If you do want to allow `NULL` values for the `Picture` column, you d want to include the following condition:

```
if (category.IsPictureNull())
{
    // Display some "No Image Available" picture
    Response.Redirect("~/Images/NoPictureAvailable.gif");
}
else
{
    // Send back the binary contents of the Picture column
    // ... Set ContentType property and write out ...
    // ... data via Response.BinaryWrite ...
}
```

The above code assumes that there s some image file named `NoPictureAvailable.gif` in the `Images` folder that you want to display for those categories without a picture.

This exception could also be caused if the `CategoriesTableAdapter` s `GetCategoryWithBinaryDataByCategoryID` method s `SELECT` statement has reverted back to the main query s column list, which can happen if you are using ad-hoc SQL statements and you ve re-run the wizard for the TableAdapter s main query. Check to ensure that `GetCategoryWithBinaryDataByCategoryID` method s `SELECT` statement still includes the `Picture` column.

> **NOTE**
>
> Every time the `DisplayCategoryPicture.aspx` is visited, the database is accessed and the specified category s picture data is returned. If the category s picture hasn't changed since the user has last viewed it, though, this is wasted effort. Fortunately, HTTP allows for *conditional GETs*. With a conditional GET, the client making the HTTP request sends along an `If-Modified-Since` HTTP header that provides the date and time the client last retrieved this resource from the web server. If the content has not changed since this specified date, the web server may respond with a Not Modified status code (304) and forgo sending back the requested resource s content. In short, this technique relieves the web server from having to send back content for a resource if it has not been modified since the client last accessed it.

To implement this behavior, however, requires that you add a `PictureLastModified` column to the `Categories` table to capture when the `Picture` column was last updated as well as code to check for the `If-Modified-Since` header. For more information on the `If-Modified-Since` header and the conditional GET workflow, see HTTP Conditional GET for RSS Hackers and A Deeper Look at Performing HTTP Requests in an ASP.NET Page.

Step 4: Displaying the Category Pictures in a GridView

Now that we have a web page to display a particular category s picture, we can display it using the Image Web control or an HTML `<img>` element pointing to `DisplayCategoryPicture.aspx?CategoryID=categoryID`. Images whose URL is determined by database data can be displayed in the GridView or DetailsView using the ImageField. The ImageField contains `DataImageUrlField` and `DataImageUrlFormatString` properties that work like the HyperLinkField s `DataNavigateUrlFields` and `DataNavigateUrlFormatString` properties.

Let s augment the `Categories` GridView in `DisplayOrDownloadData.aspx` by adding an ImageField to show each category s picture. Simply add the ImageField and set its `DataImageUrlField` and `DataImageUrlFormatString` properties to `CategoryID` and `DisplayCategoryPicture.aspx?CategoryID={0}`, respectively. This will create a GridView column that renders an `<img>` element whose `src` attribute references `DisplayCategoryPicture.aspx?CategoryID={0}`, where {0} is replaced with the GridView row s `CategoryID` value.

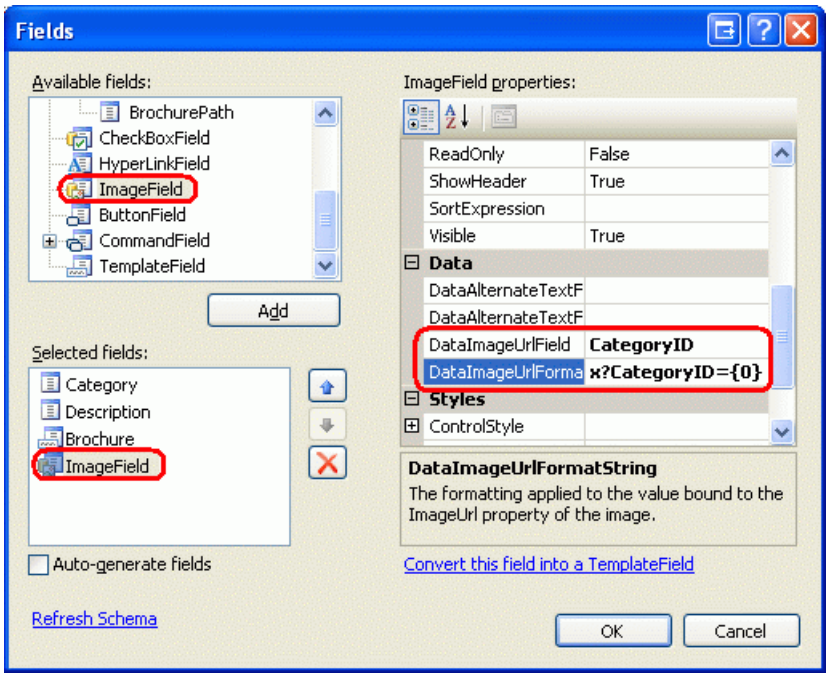


Figure 12: Add an ImageField to the GridView

After adding the ImageField, your GridView s declarative syntax should look like soothe following:

```
<asp:GridView ID="Categories" runat="server" AutoGenerateColumns="False"
    DataKeyNames="CategoryID" DataSourceID="CategoriesDataSource"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            SortExpression="CategoryName" />
        <asp:BoundField DataField="Description" HeaderText="Description"
            SortExpression="Description" />
        <asp:TemplateField HeaderText="Brochure">
            <ItemTemplate>
                <%# GenerateBrochureLink(Eval("BrochurePath")) %>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:ImageField DataImageUrlField="CategoryID"
            DataImageUrlFormatString="DisplayCategoryPicture.aspx?CategoryID={0}">
        </asp:ImageField>
    </Columns>
</asp:GridView>
```

Take a moment to view this page through a browser. Note how each record now includes a picture for the category.

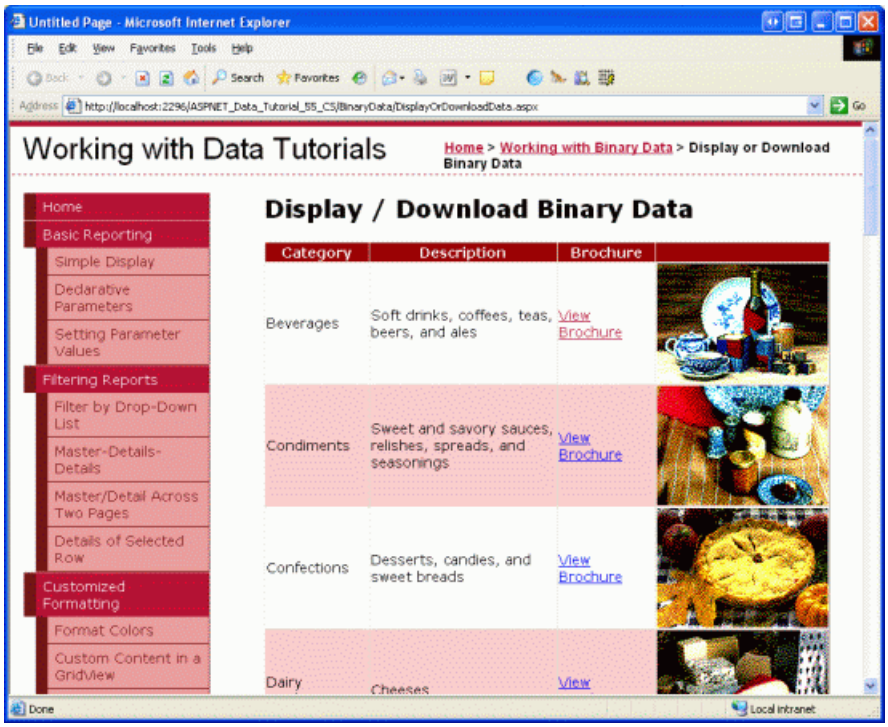


Figure 13: The Category s Picture is Displayed for Each Row (Click to view full-size image)

Summary

In this tutorial we examined how to present binary data. How the data is presented depends on the type of data. For the PDF brochure files, we offered the user a View Brochure link that, when clicked, took the user directly to the PDF file. For the category s picture, we first created a page to retrieve and return the binary data from the database and then used that page to display each category s picture in a GridView.

Now that we ve looked at how to display binary data, we re ready to examine how to perform insert, updates, and deletes against the database with the binary data. In the next tutorial we'll look at how to associate an uploaded file with its corresponding database record. In the tutorial after that, we'll see how to update existing binary data as well as how to delete the binary data when its associated record is removed.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Teresa Murphy and Dave Gardner. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Including a File Upload Option When Adding a New Record (C#)

2/20/2020 • 23 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> This tutorial shows how to create a Web interface that allows the user to both enter text data and upload binary files. To illustrate the options available to store binary data, one file will be saved in the database while the other is stored in the file system.

Introduction

In the previous two tutorials we explored techniques for storing binary data that is associated with the application s data model, looked at how to use the FileUpload control to send files from the client to the web server, and saw how to present this binary data in a data Web control. We ve yet to talk about how to associate uploaded data with the data model, though.

In this tutorial we will create a web page to add a new category. In addition to TextBoxes for the category s name and description, this page will need to include two FileUpload controls one for the new category s picture and one for the brochure. The uploaded picture will be stored directly in the new record s `Picture` column, whereas the brochure will be saved to the `~/Brochures` folder with the path to the file saved in the new record s `BrochurePath` column.

Before creating this new web page, we'll need to update the architecture. The `CategoriesTableAdapter` s main query does not retrieve the `Picture` column. Consequently, the auto-generated `Insert` method only has inputs for the `CategoryName`, `Description`, and `BrochurePath` fields. Therefore, we need to create an additional method in the TableAdapter that prompts for all four `Categories` fields. The `CategoriesBLL` class in the Business Logic Layer will also need to be updated.

Step 1: Adding an `InsertWithPicture` Method to the `CategoriesTableAdapter`

When we created the `CategoriesTableAdapter` back in the Creating a Data Access Layer tutorial, we configured it to automatically generate `INSERT`, `UPDATE`, and `DELETE` statements based on the main query. Moreover, we instructed the TableAdapter to employ the DB Direct approach, which created the methods `Insert`, `Update`, and `Delete`. These methods execute the auto-generated `INSERT`, `UPDATE`, and `DELETE` statements and, consequently, accept input parameters based on the columns returned by the main query. In the Uploading Files tutorial we augmented the `CategoriesTableAdapter` s main query to use the `BrochurePath` column.

Since the `CategoriesTableAdapter` s main query does not reference the `Picture` column, we can neither add a new record nor update an existing record with a value for the `Picture` column. In order to capture this information, we can either create a new method in the TableAdapter that is used specifically to insert a record with binary data or we can customize the auto-generated `INSERT` statement. The problem with customizing the auto-generated `INSERT` statement is that we risk having our customizations overwritten by the wizard. For example, imagine that we customized the `INSERT` statement to include use of the `Picture` column. This would update the TableAdapter s `Insert` method to include an additional input parameter for the category s picture s

binary data. We could then create a method in the Business Logic Layer to use this DAL method and invoke this BLL method through the Presentation Layer, and everything would work wonderfully. That is, until the next time we configured the TableAdapter through the TableAdapter Configuration wizard. As soon as the wizard completed, our customizations to the `INSERT` statement would be overwritten, the `Insert` method would revert to its old form, and our code would no longer compile!

> **NOTE**
> This annoyance is a non-issue when using stored procedures instead of ad-hoc SQL statements. A future tutorial will explore using stored procedures in lieu of ad-hoc SQL statements in the Data Access Layer.

To avoid this potential headache, rather than customizing the auto-generated SQL statements let s instead create a new method for the TableAdapter. This method, named `InsertWithPicture`, will accept values for the `CategoryName`, `Description`, `BrochurePath`, and `Picture` columns and execute an `INSERT` statement that stores all four values in a new record.

Open the Typed DataSet and, from the Designer, right-click on the `CategoriesTableAdapter` s header and choose Add Query from the context menu. This launches the TableAdapter Query Configuration Wizard, which begins by asking us how the TableAdapter query should access the database. Choose Use SQL statements and click Next. The next step prompts for the type of query to be generated. Since we re creating a query to add a new record to the `Categories` table, choose INSERT and click Next.

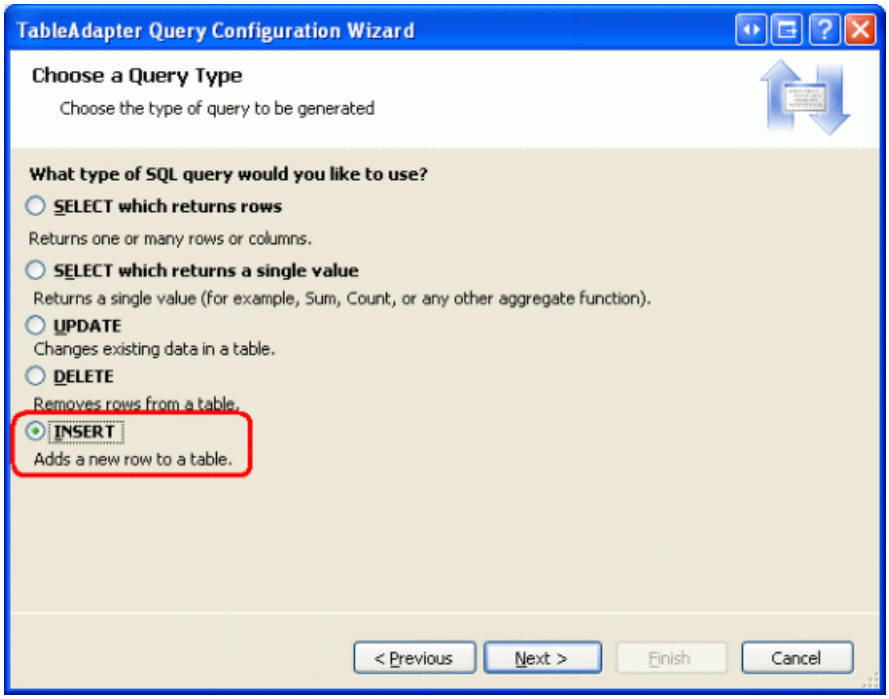


Figure 1: Select the INSERT Option (Click to view full-size image)

We now need to specify the `INSERT` SQL statement. The wizard automatically suggests an `INSERT` statement corresponding to the TableAdapter s main query. In this case, it s an `INSERT` statement that inserts the `CategoryName`, `Description`, and `BrochurePath` values. Update the statement so that the `Picture` column is included along with a `@Picture` parameter, like so:

```
INSERT INTO [Categories]
    ([CategoryName], [Description], [BrochurePath], [Picture])
VALUES
    (@CategoryName, @Description, @BrochurePath, @Picture)
```

The final screen of the wizard asks us to name the new TableAdapter method. Enter `InsertWithPicture` and click

Finish.

Figure 2: Name the New TableAdapter Method `InsertWithPicture` (Click to view full-size image)

Step 2: Updating the Business Logic Layer

Since the Presentation Layer should only interface with the Business Logic Layer rather than bypassing it to go directly to the Data Access Layer, we need to create a BLL method that invokes the DAL method we just created (`InsertWithPicture`). For this tutorial, create a method in the `CategoriesBLL` class named `InsertWithPicture` that accepts as input three `string` s and a `byte` array. The `string` input parameters are for the category s name, description, and brochure file path, while the `byte` array is for the binary contents of the category s picture. As the following code shows, this BLL method invokes the corresponding DAL method:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Insert, false)]
public void InsertWithPicture(string categoryName, string description,
    string brochurePath, byte[] picture)
{
    Adapter.InsertWithPicture(categoryName, description, brochurePath, picture);
}
```

NOTE

Make sure that you have saved the Typed DataSet before adding the `InsertWithPicture` method to the BLL. Since the `CategoriesTableAdapter` class code is auto-generated based on the Typed DataSet, if you don t first save your changes to the Typed DataSet the `Adapter` property won't know about the `InsertWithPicture` method.

Step 3: Listing the Existing Categories and their Binary Data

In this tutorial we will create a page that allows an end user to add a new category to the system, providing a picture and brochure for the new category. In the preceding tutorial we used a GridView with a TemplateField and ImageField to display each category s name, description, picture, and a link to download its brochure. Let s replicate that functionality for this tutorial, creating a page that both lists all existing categories and allows for new ones to be created.

Start by opening the `DisplayOrDownload.aspx` page from the `BinaryData` folder. Go to the Source view and copy the GridView and ObjectDataSource s declarative syntax, pasting it within the `<asp:Content>` element in `UploadInDetailsView.aspx`. Also, don t forget to copy over the `GenerateBrochureLink` method from the code-behind class of `DisplayOrDownload.aspx` to `UploadInDetailsView.aspx`.

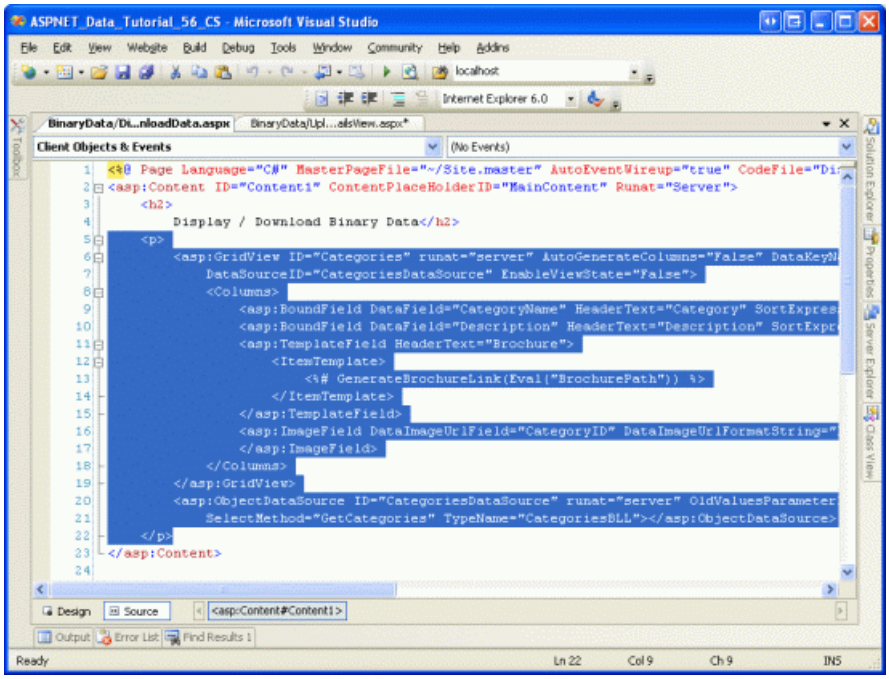


Figure 3: Copy and Paste the Declarative Syntax from `DisplayOrDownload.aspx` to `UploadInDetailsView.aspx` (Click to view full-size image)

After copying the declarative syntax and `GenerateBrochureLink` method over to the `UploadInDetailsView.aspx` page, view the page through a browser to ensure that everything was copied over correctly. You should see a GridView listing the eight categories that includes a link to download the brochure as well as the category s picture.

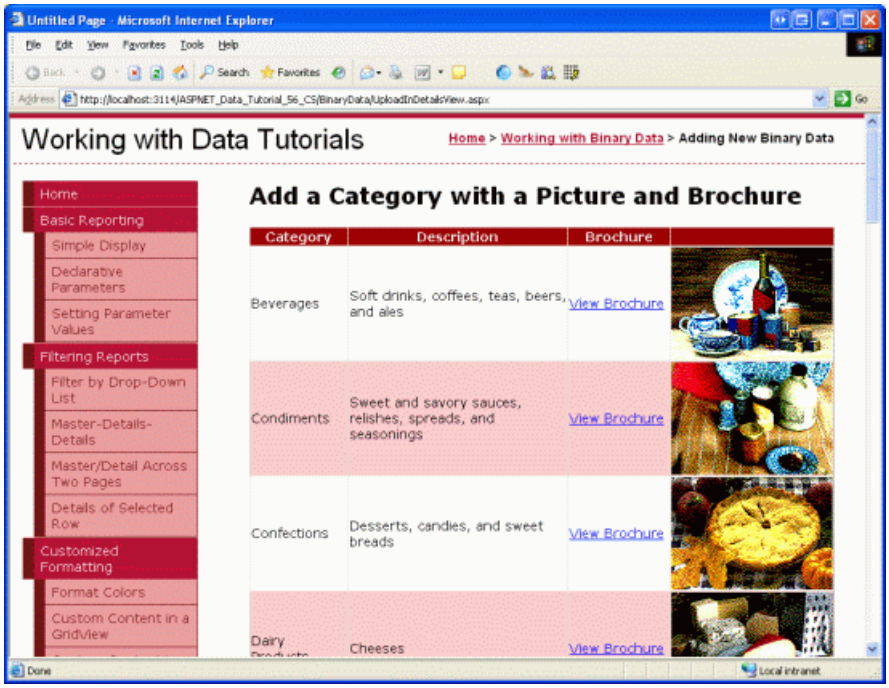


Figure 4: You Should Now See Each Category Along with Its Binary Data (Click to view full-size image)

Step 4: Configuring the `CategoriesDataSource` to Support Inserting

The `CategoriesDataSource` ObjectDataSource used by the `Categories` GridView currently does not provide the

ability to insert data. In order to support inserting through this data source control, we need to map its `Insert` method to a method in its underlying object, `CategoriesBLL`. In particular, we want to map it to the `CategoriesBLL` method we added back in Step 2, `InsertWithPicture`.

Start by clicking the Configure Data Source link from the ObjectDataSource s smart tag. The first screen shows the object the data source is configured to work with, `CategoriesBLL`. Leave this setting as-is and click Next to advance to the Define Data Methods screen. Move to the INSERT tab and pick the `InsertWithPicture` method from the drop-down list. Click Finish to complete the wizard.

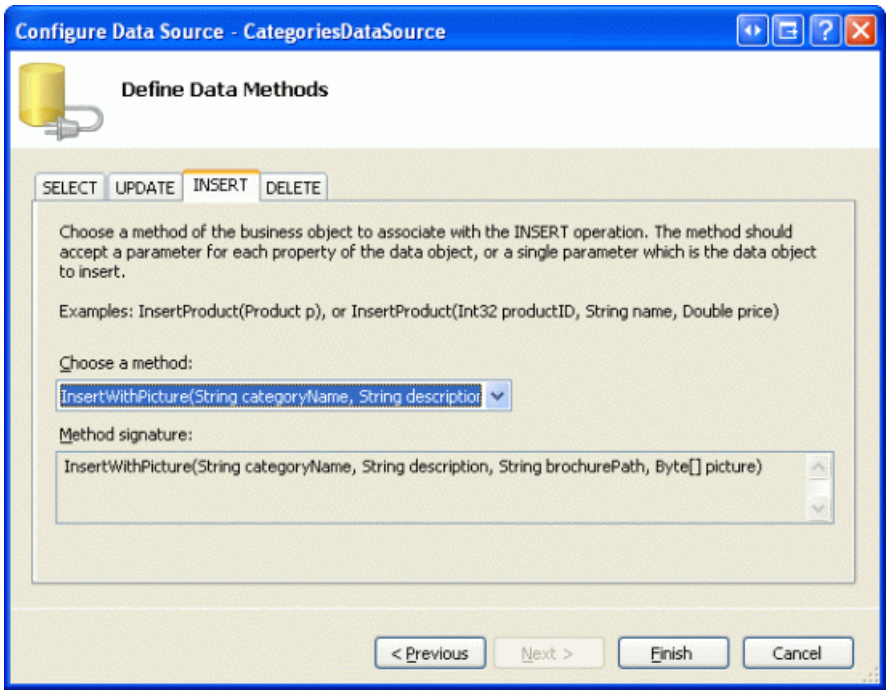


Figure 5: Configure the ObjectDataSource to use the `InsertWithPicture` Method (Click to view full-size image)

> **NOTE**
>
> Upon completing the wizard, Visual Studio may ask if you want to Refresh Fields and Keys, which will regenerate the data Web controls fields. Choose No, because choosing Yes will overwrite any field customizations you may have made.

After completing the wizard, the ObjectDataSource will now include a value for its `InsertMethod` property as well as `InsertParameters` for the four category columns, as the following declarative markup illustrates:

```
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}" SelectMethod="GetCategories"
    TypeName="CategoriesBLL" InsertMethod="InsertWithPicture">
    <InsertParameters>
        <asp:Parameter Name="categoryName" Type="String" />
        <asp:Parameter Name="description" Type="String" />
        <asp:Parameter Name="brochurePath" Type="String" />
        <asp:Parameter Name="picture" Type="Object" />
    </InsertParameters>
</asp:ObjectDataSource>
```

Step 5: Creating the Inserting Interface

As first covered in the An Overview of Inserting, Updating, and Deleting Data, the DetailsView control provides a built-in inserting interface that can be utilized when working with a data source control that supports inserting. Let s add a DetailsView control to this page above the GridView that will permanently render its inserting interface, allowing a user to quickly add a new category. Upon adding a new category in the DetailsView, the

GridView beneath it will automatically refresh and display the new category.

Start by dragging a DetailsView from the Toolbox onto the Designer above the GridView, setting its `ID` property to `NewCategory` and clearing out the `Height` and `Width` property values. From the DetailsView s smart tag, bind it to the existing `CategoriesDataSource` and then check the Enable Inserting checkbox.

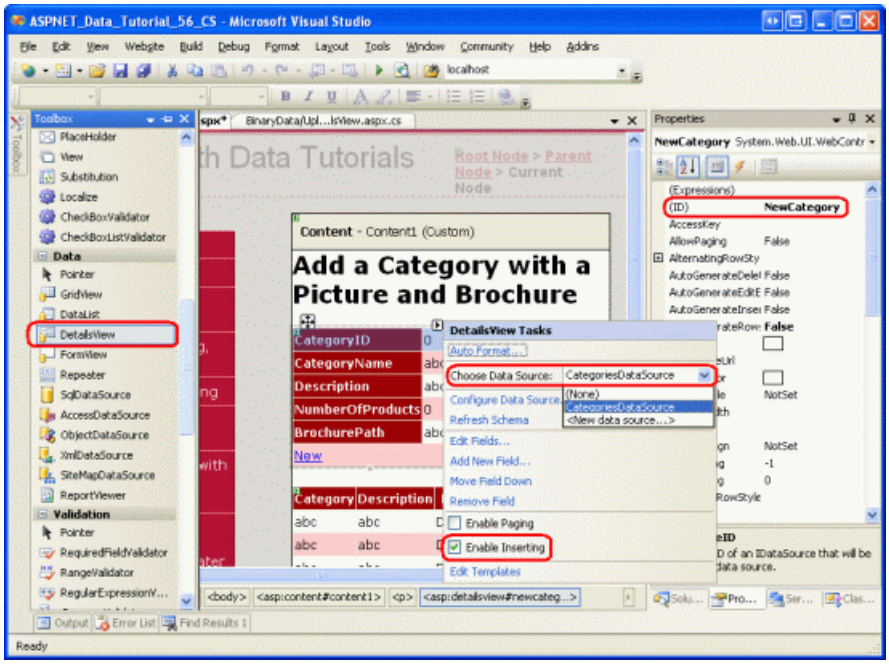


Figure 6: Bind the DetailsView to the `CategoriesDataSource` and Enable Inserting (Click to view full-size image)

To permanently render the DetailsView in its inserting interface, set its `DefaultMode` property to `Insert`.

Note that the DetailsView has five BoundFields `CategoryID`, `CategoryName`, `Description`, `NumberOfProducts`, and `BrochurePath` although the `CategoryID` BoundField is not rendered in the inserting interface because its `InsertVisible` property is set to `false`. These BoundFields exists because they are the columns returned by the `GetCategories()` method, which is what the ObjectDataSource invokes to retrieve its data. For inserting, however, we don t want to let the user specify a value for `NumberOfProducts`. Moreover, we need to allow them to upload a picture for the new category as well as upload a PDF for the brochure.

Remove the `NumberOfProducts` BoundField from the DetailsView altogether and then update the `HeaderText` properties of the `CategoryName` and `BrochurePath` BoundFields to Category and Brochure, respectively. Next, convert the `BrochurePath` BoundField into a TemplateField and add a new TemplateField for the picture, giving this new TemplateField a `HeaderText` value of Picture. Move the `Picture` TemplateField so that it is between the `BrochurePath` TemplateField and CommandField.

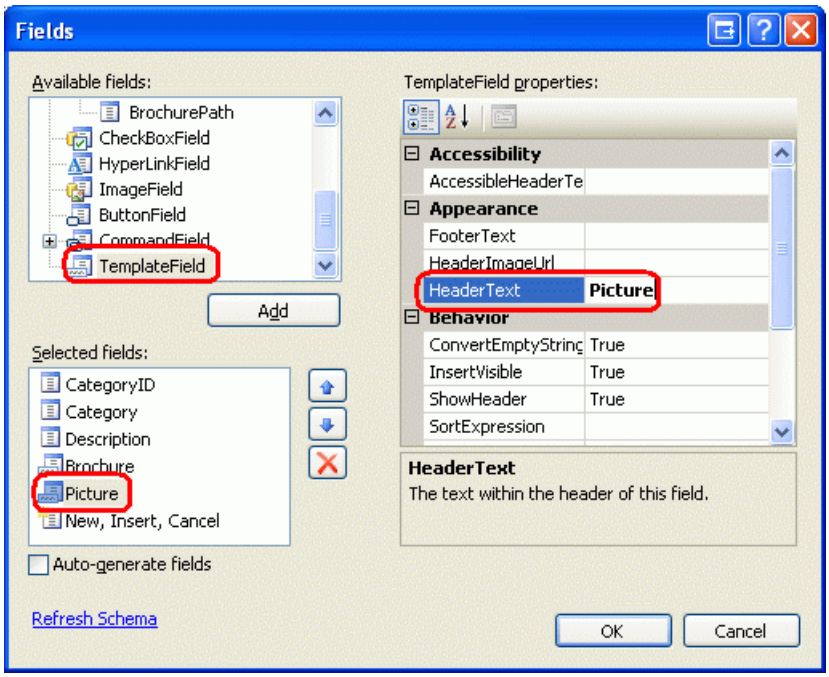


Figure 7: Bind the DetailsView to the `CategoriesDataSource` and Enable Inserting

If you converted the `BrochurePath` BoundField into a TemplateField through the Edit Fields dialog box, the TemplateField includes an `ItemTemplate`, `EditItemTemplate`, and `InsertItemTemplate`. Only the `InsertItemTemplate` is needed, however, so feel free to remove the other two templates. At this point your DetailsView s declarative syntax should look like the following:

```
<asp:DetailsView ID="NewCategory" runat="server" AutoGenerateRows="False"
    DataKeyNames="CategoryID" DataSourceID="CategoriesDataSource"
    DefaultMode="Insert">
    <Fields>
        <asp:BoundField DataField="CategoryID" HeaderText="CategoryID"
            InsertVisible="False" ReadOnly="True"
            SortExpression="CategoryID" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            SortExpression="CategoryName" />
        <asp:BoundField DataField="Description" HeaderText="Description"
            SortExpression="Description" />
        <asp:TemplateField HeaderText="Brochure" SortExpression="BrochurePath">
            <InsertItemTemplate>
                <asp:TextBox ID="TextBox1" runat="server"
                    Text='<%# Bind("BrochurePath") %>'></asp:TextBox>
            </InsertItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Picture"></asp:TemplateField>
        <asp:CommandField ShowInsertButton="True" />
    </Fields>
</asp:DetailsView>
```

Adding FileUpload Controls for the Brochure and Picture Fields

Presently, the `BrochurePath` TemplateField s `InsertItemTemplate` contains a TextBox, while the `Picture` TemplateField does not contain any templates. We need to update these two TemplateField s `InsertItemTemplate` s to use FileUpload controls.

From the DetailsView s smart tag, choose the Edit Templates option and then select the `BrochurePath` TemplateField s `InsertItemTemplate` from the drop-down list. Remove the TextBox and then drag a FileUpload control from the Toolbox into the template. Set the FileUpload control s `ID` to `BrochureUpload`. Similarly, add a FileUpload control to the `Picture` TemplateField s `InsertItemTemplate`. Set this FileUpload control s `ID` to

`PictureUpload`.

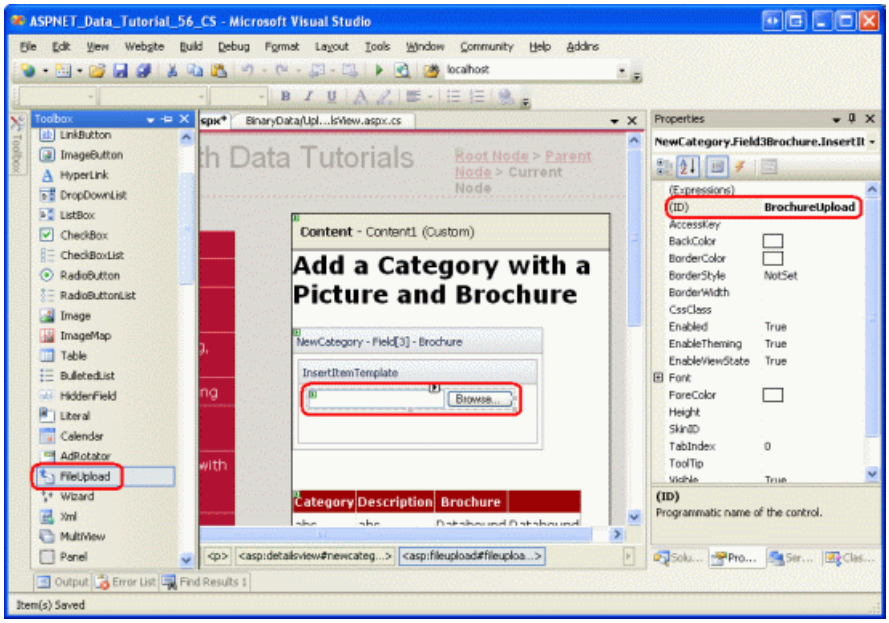


Figure 8: Add a FileUpload Control to the `InsertItemTemplate` (Click to view full-size image)

After making these additions, the two TemplateField s declarative syntax will be:

```
<asp:TemplateField HeaderText="Brochure" SortExpression="BrochurePath">
    <InsertItemTemplate>
        <asp:FileUpload ID="BrochureUpload" runat="server" />
    </InsertItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="Picture">
    <InsertItemTemplate>
        <asp:FileUpload ID="PictureUpload" runat="server" />
    </InsertItemTemplate>
</asp:TemplateField>
```

When a user adds a new category, we want to ensure that the brochure and picture are of the correct file type. For the brochure, the user must supply a PDF. For the picture, we need the user to upload an image file, but do we allow *any* image file or only image files of a particular type, such as GIFs or JPGs? In order to allow for different file types, we d need to extend the `Categories` schema to include a column that captures the file type so that this type can be sent to the client through `Response.ContentType` in `DisplayCategoryPicture.aspx`. Since we don t have such a column, it would be prudent to restrict users to only providing a specific image file type. The `Categories` table s existing images are bitmaps, but JPGs are a more appropriate file format for images served over the web.

If a user uploads an incorrect file type, we need to cancel the insert and display a message indicating the problem. Add a Label Web control beneath the DetailsView. Set its `ID` property to `UploadWarning`, clear out its `Text` property, set the `CssClass` property to Warning, and the `Visible` and `EnableViewState` properties to `false`. The `Warning` CSS class is defined in `Styles.css` and renders the text in a large, red, italicized, bold font.

> **NOTE**
>
> Ideally, the `CategoryName` and `Description` BoundFields would be converted to TemplateFields and their inserting interfaces customized. The `Description` inserting interface, for example, would likely be better suited through a multi-line textbox. And since the `CategoryName` column does not accept `NULL` values, a RequiredFieldValidator should be added to ensure the user provides a value for the new category s name. These steps are left as an exercise to the reader. Refer back to Customizing the Data Modification Interface for an in-depth look at augmenting the data modification interfaces.

Step 6: Saving the Uploaded Brochure to the Web Server s File System

When the user enters the values for a new category and clicks the Insert button, a postback occurs and the inserting workflow unfolds. First, the DetailsView s `ItemInserting` event fires. Next, the ObjectDataSource s `Insert()` method is invoked, which results in a new record being added to the `Categories` table. After that, the DetailsView s `ItemInserted` event fires.

Before the ObjectDataSource s `Insert()` method is invoked, we must first ensure that the appropriate file types were uploaded by the user and then save the brochure PDF to the web server s file system. Create an event handler for the DetailsView s `ItemInserting` event and add the following code:

```
// Reference the FileUpload control
FileUpload BrochureUpload =
    (FileUpload)NewCategory.FindControl("BrochureUpload");
if (BrochureUpload.HasFile)
{
    // Make sure that a PDF has been uploaded
    if (string.Compare(System.IO.Path.GetExtension
        (BrochureUpload.FileName), ".pdf", true) != 0)
    {
        UploadWarning.Text =
            "Only PDF documents may be used for a category's brochure.";
        UploadWarning.Visible = true;
        e.Cancel = true;
        return;
    }
}
```

The event handler starts by referencing the `BrochureUpload` FileUpload control from the DetailsView s templates. Then, if a brochure has been uploaded, the uploaded file s extension is examined. If the extension is not .PDF, then a warning is displayed, the insert is cancelled, and the execution of the event handler ends.

> **NOTE**
>
> Relying on the uploaded file s extension is not a sure-fire technique for ensuring that the uploaded file is a PDF document. The user could have a valid PDF document with the extension `.Brochure`, or could have taken a non-PDF document and given it a `.pdf` extension. The file s binary content would need to be programmatically examined in order to more conclusively verify the file type. Such thorough approaches, though, are often overkill; checking the extension is sufficient for most scenarios.

As discussed in the Uploading Files tutorial, care must be taken when saving files to the file system so that one user s upload does not overwrite another s. For this tutorial we will attempt to use the same name as the uploaded file. If there already exists a file in the `~/Brochures` directory with that same file name, however, we'll append a number at the end until a unique name is found. For example, if the user uploads a brochure file named `Meats.pdf`, but there is already a file named `Meats.pdf` in the `~/Brochures` folder, we'll change the saved file name to `Meats-1.pdf`. If that exists, we'll try `Meats-2.pdf`, and so on, until a unique file name is found.

The following code uses the `File.Exists(path)` method to determine if a file already exists with the specified file name. If so, it continues to try new file names for the brochure until no conflict is found.

```
const string BrochureDirectory = "~/Brochures/";
string brochurePath = BrochureDirectory + BrochureUpload.FileName;
string fileNameWithoutExtension =
    System.IO.Path.GetFileNameWithoutExtension(BrochureUpload.FileName);
int iteration = 1;
while (System.IO.File.Exists(Server.MapPath(brochurePath)))
{
    brochurePath = string.Concat(BrochureDirectory,
        fileNameWithoutExtension, "-", iteration, ".pdf");
    iteration++;
}
```

Once a valid file name has been found, the file needs to be saved to the file system and the ObjectDataSource s `brochurePath``InsertParameter` value needs to be updated so that this file name is written to the database. As we saw back in the *Uploading Files* tutorial, the file can be saved using the FileUpload control s `SaveAs(path)` method. To update the ObjectDataSource s `brochurePath` parameter, use the `e.Values` collection.

```
// Save the file to disk and set the value of the brochurePath parameter
BrochureUpload.SaveAs(Server.MapPath(brochurePath));
e.Values["brochurePath"] = brochurePath;
```

Step 7: Saving the Uploaded Picture to the Database

To store the uploaded picture in the new `Categories` record, we need to assign the uploaded binary content to the ObjectDataSource s `picture` parameter in the DetailsView s `ItemInserting` event. Before we make this assignment, however, we need to first make sure that the uploaded picture is a JPG and not some other image type. As in Step 6, let s use the uploaded picture s file extension to ascertain its type.

While the `Categories` table allows `NULL` values for the `Picture` column, all categories currently have a picture. Let s force the user to provide a picture when adding a new category through this page. The following code checks to ensure that a picture has been uploaded and that it has an appropriate extension.

```
// Reference the FileUpload controls
FileUpload PictureUpload = (FileUpload)NewCategory.FindControl("PictureUpload");
if (PictureUpload.HasFile)
{
    // Make sure that a JPG has been uploaded
    if (string.Compare(System.IO.Path.GetExtension(PictureUpload.FileName),
            ".jpg", true) != 0 &&
        string.Compare(System.IO.Path.GetExtension(PictureUpload.FileName),
            ".jpeg", true) != 0)
    {
        UploadWarning.Text =
            "Only JPG documents may be used for a category's picture.";
        UploadWarning.Visible = true;
        e.Cancel = true;
        return;
    }
}
else
{
    // No picture uploaded!
    UploadWarning.Text =
        "You must provide a picture for the new category.";
    UploadWarning.Visible = true;
    e.Cancel = true;
    return;
}
```

This code should be placed *before* the code from Step 6 so that if there is a problem with the picture upload, the event handler will terminate before the brochure file is saved to the file system.

Assuming that an appropriate file has been uploaded, assign the uploaded binary content to the picture parameter s value with the following line of code:

```
// Set the value of the picture parameter
e.Values["picture"] = PictureUpload.FileBytes;
```

The Complete `ItemInserting` Event Handler

For completeness, here is the `ItemInserting` event handler in its entirety:

```
protected void NewCategory_ItemInserting(object sender, DetailsViewInsertEventArgs e)
{
    // Reference the FileUpload controls
    FileUpload PictureUpload = (FileUpload)NewCategory.FindControl("PictureUpload");
    if (PictureUpload.HasFile)
    {
        // Make sure that a JPG has been uploaded
        if (string.Compare(System.IO.Path.GetExtension(PictureUpload.FileName),
                ".jpg", true) != 0 &&
            string.Compare(System.IO.Path.GetExtension(PictureUpload.FileName),
                ".jpeg", true) != 0)
        {
            UploadWarning.Text =
                "Only JPG documents may be used for a category's picture.";
            UploadWarning.Visible = true;
            e.Cancel = true;
            return;
        }
    }
    else
    {
        // No picture uploaded!
        UploadWarning.Text =
            "You must provide a picture for the new category.";
        UploadWarning.Visible = true;
        e.Cancel = true;
        return;
    }
    // Set the value of the picture parameter
    e.Values["picture"] = PictureUpload.FileBytes;


    // Reference the FileUpload controls
    FileUpload BrochureUpload =
        (FileUpload)NewCategory.FindControl("BrochureUpload");
    if (BrochureUpload.HasFile)
    {
        // Make sure that a PDF has been uploaded
        if (string.Compare(System.IO.Path.GetExtension(BrochureUpload.FileName),
            ".pdf", true) != 0)
        {
            UploadWarning.Text =
                "Only PDF documents may be used for a category's brochure.";
            UploadWarning.Visible = true;
            e.Cancel = true;
            return;
        }
        const string BrochureDirectory = "~/Brochures/";
        string brochurePath = BrochureDirectory + BrochureUpload.FileName;
        string fileNameWithoutExtension =
            System.IO.Path.GetFileNameWithoutExtension(BrochureUpload.FileName);
        int iteration = 1;
        while (System.IO.File.Exists(Server.MapPath(brochurePath)))
        {
            brochurePath = string.Concat(BrochureDirectory, fileNameWithoutExtension,
                "-", iteration, ".pdf");
            iteration++;
        }
        // Save the file to disk and set the value of the brochurePath parameter
        BrochureUpload.SaveAs(Server.MapPath(brochurePath));
        e.Values["brochurePath"] = brochurePath;
    }
}
```

Step 8: Fixing the `DisplayCategoryPicture.aspx` Page

Let s take a moment to test out the inserting interface and `ItemInserting` event handler that was created over the last few steps. Visit the `UploadInDetailsView.aspx` page through a browser and attempt to add a category, but omit the picture, or specify a non-JPG picture or a non-PDF brochure. In any of these cases, an error message will be displayed and the insert workflow cancelled.

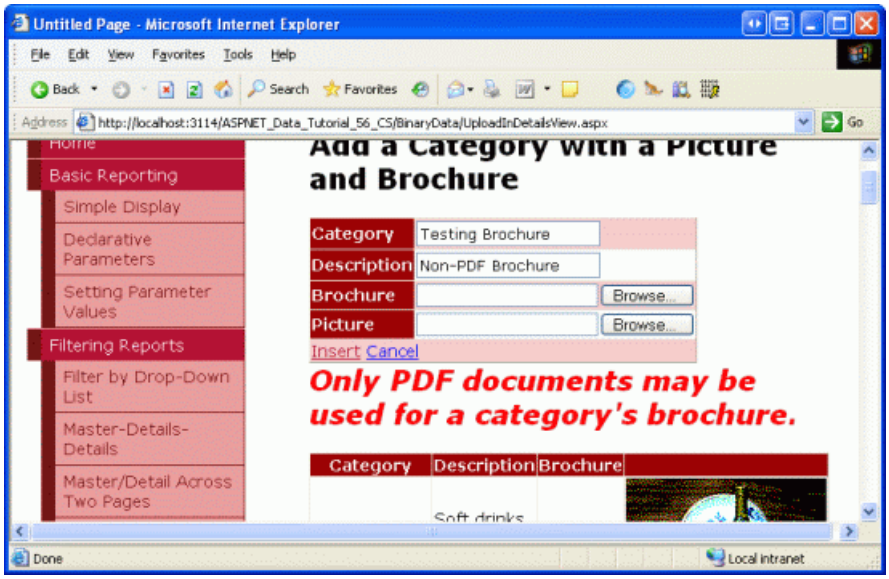


Figure 9: A Warning Message is Displayed If an Invalid File Type is Uploaded (Click to view full-size image)

Once you have verified that the page requires a picture to be uploaded and won't accept non-PDF or non-JPG files, add a new category with a valid JPG picture, leaving the Brochure field empty. After clicking the Insert button, the page will postback and a new record will be added to the `Categories` table with the uploaded image s binary contents stored directly in the database. The GridView is updated and shows a row for the newly added category, but, as Figure 10 shows, the new category s picture is not rendered correctly.

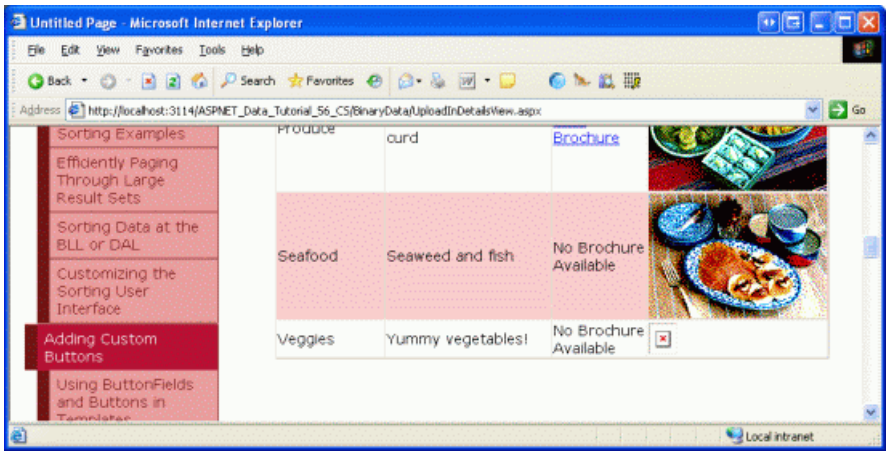


Figure 10: The New Category s Picture is not Displayed (Click to view full-size image)

The reason the new picture is not displayed is because the `DisplayCategoryPicture.aspx` page that returns a specified category s picture is configured to process bitmaps that have an OLE header. This 78 byte header is stripped from the `Picture` column s binary contents before they are sent back to the client. But the JPG file we just uploaded for the new category does not have this OLE header; therefore, valid, necessary bytes are being removed from the image s binary data.

Since there are now both bitmaps with OLE headers and JPGs in the `Categories` table, we need to update `DisplayCategoryPicture.aspx` so that it does the OLE header stripping for the original eight categories and bypasses this stripping for the newer category records. In our next tutorial we'll examine how to update an existing record s image, and we'll update all of the old category pictures so that they are JPGs. For now, though, use the following code in `DisplayCategoryPicture.aspx` to strip the OLE headers only for those original eight categories:

```
protected void Page_Load(object sender, EventArgs e)
{
    int categoryID = Convert.ToInt32(Request.QueryString["CategoryID"]);
    // Get information about the specified category
    CategoriesBLL categoryAPI = new CategoriesBLL();
    Northwind.CategoriesDataTable categories =
        categoryAPI.GetCategoryWithBinaryDataByCategoryID(categoryID);
    Northwind.CategoriesRow category = categories[0];
    if (categoryID <= 8)
    {
        // For older categories, we must strip the OLE header... images are bitmaps
        // Output HTTP headers providing information about the binary data
        Response.ContentType = "image/bmp";
        // Output the binary data
        // But first we need to strip out the OLE header
        const int OleHeaderLength = 78;
        int strippedImageLength = category.Picture.Length - OleHeaderLength;
        byte[] strippedImageData = new byte[strippedImageLength];
        Array.Copy(category.Picture, OleHeaderLength, strippedImageData,
            0, strippedImageLength);
        Response.BinaryWrite(strippedImageData);
    }
    else
    {
        // For new categories, images are JPGs...

        // Output HTTP headers providing information about the binary data
        Response.ContentType = "image/jpeg";
        // Output the binary data
        Response.BinaryWrite(category.Picture);
    }
}
```

With this change, the JPG image is now rendered correctly in the GridView.

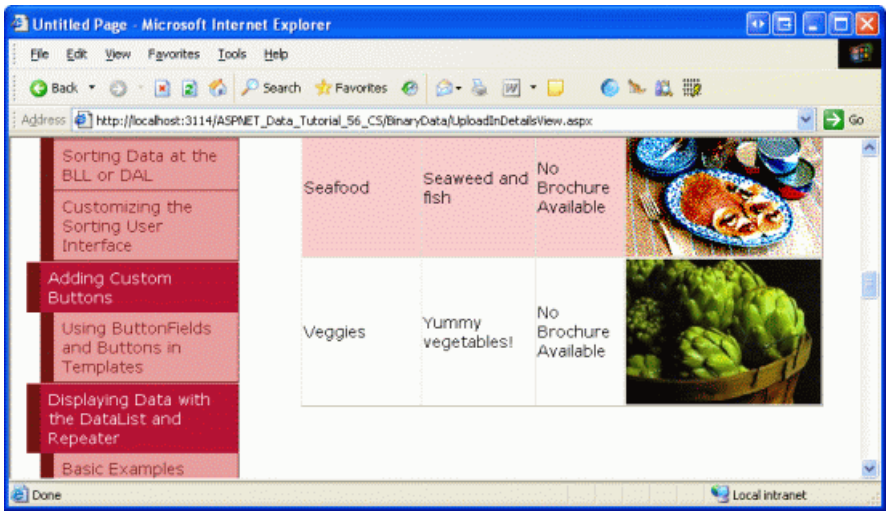


Figure 11: The JPG Images for New Categories are Correctly Rendered (Click to view full-size image)

Step 9: Deleting the Brochure in the Face of an Exception

One of the challenges of storing binary data on the web server s file system is that it introduces a disconnect between the data model and its binary data. Therefore, whenever a record is deleted, the corresponding binary data on the file system must also be removed. This can come into play when inserting, as well. Consider the following scenario: a user adds a new category, specifying a valid picture and brochure. Upon clicking the Insert button, a postback occurs and the DetailsView s `ItemInserting` event fires, saving the brochure to the web server s file system. Next, the ObjectDataSource s `Insert()` method is invoked, which calls the `CategoriesBLL` class s `InsertWithPicture` method, which calls the `CategoriesTableAdapter` s `InsertWithPicture` method.

Now, what happens if the database is offline, or if there is an error in the `INSERT` SQL statement? Clearly the INSERT will fail, so no new category row will be added to the database. But we still have the uploaded brochure file sitting on the web server s file system! This file needs to be deleted in the face of an exception during the inserting workflow.

As discussed previously in the Handling BLL- and DAL-Level Exceptions in an ASP.NET Page tutorial, when an exception is thrown from within the depths of the architecture it is bubbled up through the various layers. At the Presentation Layer, we can determine if an exception has occurred from the DetailsView s `ItemInserted` event. This event handler also provides the values of the ObjectDataSource s `InsertParameters`. Therefore, we can create an event handler for the `ItemInserted` event that checks if there was an exception and, if so, deletes the file specified by the ObjectDataSource s `brochurePath` parameter:

```
protected void NewCategory_ItemInserted
    (object sender, DetailsViewInsertedEventArgs e)
{
    if (e.Exception != null)
    {
        // Need to delete brochure file, if it exists
        if (e.Values["brochurePath"] != null)
            System.IO.File.Delete(Server.MapPath(
                e.Values["brochurePath"].ToString()));
    }
}
```

Summary

There are a number of steps that must be performed in order to provide a web-based interface for adding records that include binary data. If the binary data is being stored directly into the database, chances are you'll need to update the architecture, adding specific methods to handle the case where binary data is being inserted. Once the architecture has been updated, the next step is creating the inserting interface, which can be accomplished using a DetailsView that has been customized to include a FileUpload control for each binary data field. The uploaded data can then be saved to the web server s file system or assigned to a data source parameter in the DetailsView s `ItemInserting` event handler.

Saving binary data to the file system requires more planning than saving data directly into the database. A naming scheme must be chosen in order to avoid one user s upload overwriting another s. Also, extra steps must be taken to delete the uploaded file if the database insert fails.

We now have the ability to add new categories to the system with a brochure and picture, but we ve yet to look at how to update an existing category s binary data or how to correctly remove the binary data for a deleted category. We'll explore these two topics in the next tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Dave Gardner, Teresa Murphy, and Bernadette Leigh. Interested in reviewing my upcoming MSDN articles? If so, drop me a line

at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Updating and Deleting Existing Binary Data (C#)

2/20/2020 • 30 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In earlier tutorials we saw how the GridView control makes it simple to edit and delete text data. In this tutorial we see how the GridView control also makes it possible to edit and delete binary data, whether that binary data is saved in the database or stored in the file system.

Introduction

Over the past three tutorials we've added quite a bit of functionality for working with binary data. We started by adding a `BrochurePath` column to the `Categories` table and updated the architecture accordingly. We also added Data Access Layer and Business Logic Layer methods to work with the Categories table's existing `Picture` column, which holds the binary content's of an image file. We have built web pages to present the binary data in a GridView a download link for the brochure, with the category's picture shown in an `<img>` element and have added a DetailsView to allow users to add a new category and upload its brochure and picture data.

All that remains to be implemented is the ability to edit and delete existing categories, which we'll accomplish in this tutorial using the GridView's built-in editing and deleting features. When editing a category, the user will be able to optionally upload a new picture or have the category continue to use the existing one. For the brochure, they can either choose to use the existing brochure, to upload a new brochure, or to indicate that the category no longer has a brochure associated with it. Let's get started!

Step 1: Updating the Data Access Layer

The DAL has auto-generated `Insert`, `Update`, and `Delete` methods, but these methods were generated based on the `CategoriesTableAdapter`'s main query, which does not include the `Picture` column. Therefore, the `Insert` and `Update` methods do not include parameters for specifying the binary data for the category's picture. Like we did in the preceding tutorial, we need to create a new TableAdapter method for updating the `Categories` table when specifying binary data.

Open the Typed DataSet and, from the Designer, right-click on the `CategoriesTableAdapter`'s header and choose Add Query from the context menu to launch the TableAdapter Query Configuration Wizard. This wizard starts by asking us how the TableAdapter query should access the database. Choose Use SQL statements and click Next. The next step prompts for the type of query to be generated. Since we're creating a query to add a new record to the `Categories` table, choose UPDATE and click Next.

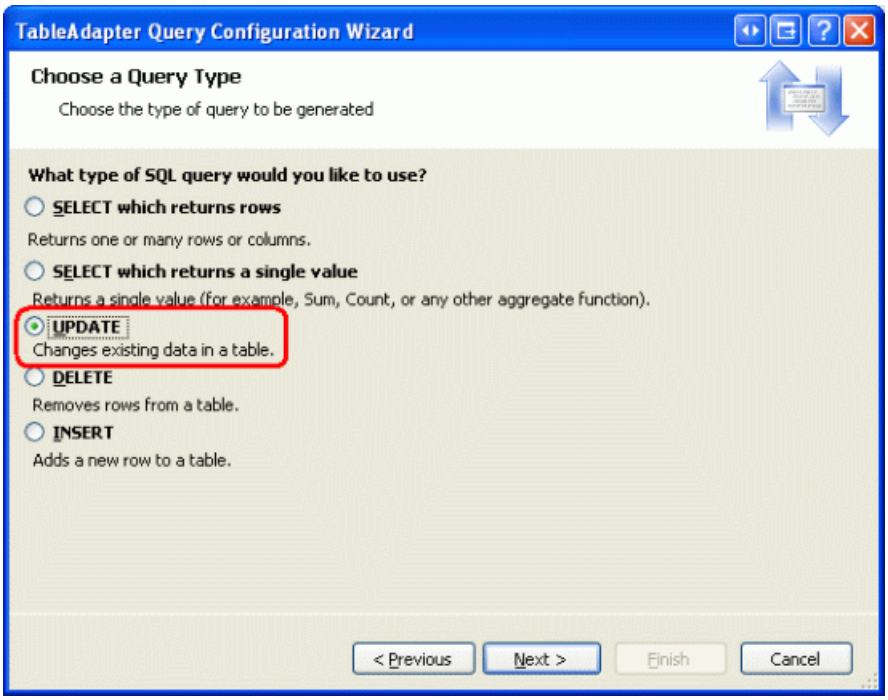


Figure 1: Select the UPDATE Option (Click to view full-size image)

We now need to specify the `UPDATE` SQL statement. The wizard automatically suggests an `UPDATE` statement corresponding to the TableAdapter s main query (one that updates the `CategoryName`, `Description`, and `BrochurePath` values). Change the statement so that the `Picture` column is included along with a `@Picture` parameter, like so:

```
UPDATE [Categories] SET
    [CategoryName] = @CategoryName,
    [Description] = @Description,
    [BrochurePath] = @BrochurePath ,
    [Picture] = @Picture
WHERE (([CategoryID] = @Original_CategoryID))
```

The final screen of the wizard asks us to name the new TableAdapter method. Enter `UpdateWithPicture` and click Finish.

Figure 2: Name the New TableAdapter Method `UpdateWithPicture` (Click to view full-size image)

Step 2: Adding the Business Logic Layer Methods

In addition to updating the DAL, we need to update the BLL to include methods for updating and deleting a category. These are the methods that will be invoked from the Presentation Layer.

For deleting a category, we can use the `CategoriesTableAdapter` s auto-generated `Delete` method. Add the following method to the `CategoriesBLL` class:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Delete, true)]
public bool DeleteCategory(int categoryID)
{
    int rowsAffected = Adapter.Delete(categoryID);
    // Return true if precisely one row was deleted, otherwise false
    return rowsAffected == 1;
}
```

For this tutorial, let s create two methods for updating a category - one that expects the binary picture data and invokes the `UpdateWithPicture` method we just added to the `CategoriesTableAdapter` and another that accepts just the `CategoryName`, `Description`, and `BrochurePath` values and uses `CategoriesTableAdapter` class s auto-generated `Update` statement. The rationale behind using two methods is that in some circumstances, a user might want to update the category s picture along with its other fields, in which case the user will have to upload the new picture. The uploaded picture s binary data can then be used in the `UPDATE` statement. In other cases, the user might only be interested in updating, say, the name and description. But if the `UPDATE` statement expects the binary data for the `Picture` column as well, then we d need to provide that information as well. This would require an extra trip to the database to bring back the picture data for the record being edited. Therefore, we want two `UPDATE` methods. The Business Logic Layer will determine which one to use based on whether picture data is provided when updating the category.

To facilitate this, add two methods to the `CategoriesBLL` class, both named `UpdateCategory`. The first one should accept three `string` s, a `byte` array, and an `int` as its input parameters; the second, just three `string` s and an `int`. The `string` input parameters are for the category s name, description, and brochure file path, the `byte` array is for the binary contents of the category s picture, and the `int` identifies the `CategoryID` of the record to update. Notice that the first overload invokes the second if the passed-in `byte` array is `null`:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Update, false)]
public bool UpdateCategory(string categoryName, string description,
    string brochurePath, byte[] picture, int categoryID)
{
    // If no picture is specified, use other overload
    if (picture == null)
        return UpdateCategory(categoryName, description, brochurePath, categoryID);
    // Update picture, as well
    int rowsAffected = Adapter.UpdateWithPicture
        (categoryName, description, brochurePath, picture, categoryID);
    // Return true if precisely one row was updated, otherwise false
    return rowsAffected == 1;
}
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Update, true)]
public bool UpdateCategory(string categoryName, string description,
    string brochurePath, int categoryID)
{
    int rowsAffected = Adapter.Update
        (categoryName, description, brochurePath, categoryID);
    // Return true if precisely one row was updated, otherwise false
    return rowsAffected == 1;
}
```

Step 3: Copying Over the Insert and View Functionality

In the preceding tutorial we created a page named `UploadInDetailsView.aspx` that listed all categories in a GridView and provided a DetailsView to add new categories to the system. In this tutorial we will extend the GridView to include editing and deleting support. Rather than continuing to work from `UploadInDetailsView.aspx`, let s instead place this tutorial s changes in the `UpdatingAndDeleting.aspx` page from the same folder, `~/BinaryData`. Copy and paste the declarative markup and code from `UploadInDetailsView.aspx` to `UpdatingAndDeleting.aspx`.

Start by opening the `UploadInDetailsView.aspx` page. Copy all of the declarative syntax within the `<asp:Content>` element, as shown in Figure 3. Next, open `UpdatingAndDeleting.aspx` and paste this markup within its `<asp:Content>` element. Similarly, copy the code from the `UploadInDetailsView.aspx` page s code-behind class to `UpdatingAndDeleting.aspx`.

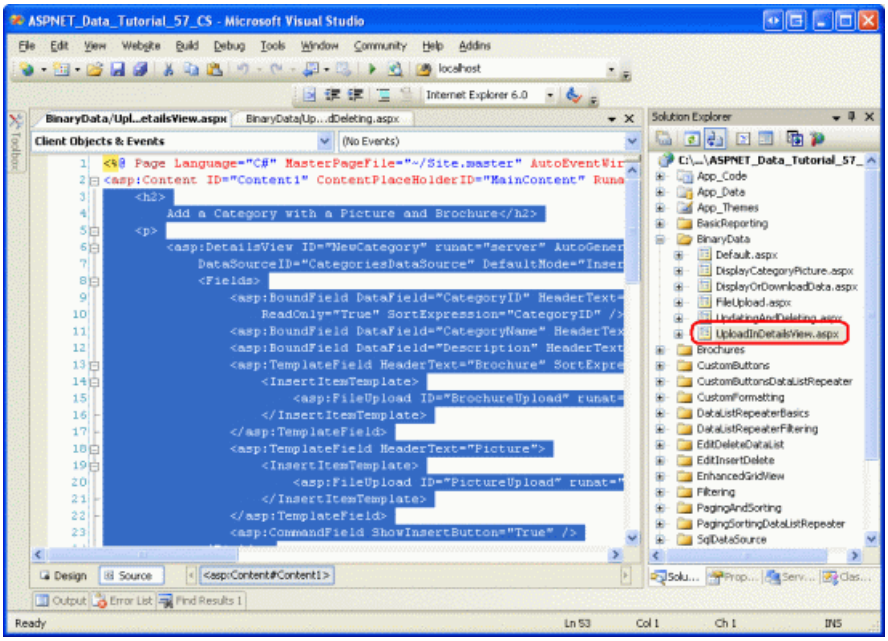


Figure 3: Copy the Declarative Markup from `UploadInDetailsView.aspx` (Click to view full-size image)

After copying over the declarative markup and code, visit `UpdatingAndDeleting.aspx`. You should see the same output and have the same user experience as with `UploadInDetailsView.aspx` page from the previous tutorial.

Step 4: Adding Deleting Support to the ObjectDataSource and GridView

As we discussed back in the An Overview of Inserting, Updating, and Deleting Data tutorial, the GridView provides built-in deleting capabilities and these capabilities can be enabled at the tick of a checkbox if the grid s underlying data source supports deleting. Currently the ObjectDataSource the GridView is bound to (`CategoriesDataSource`) does not support deleting.

To remedy this, click on the Configure Data Source option from the ObjectDataSource s smart tag to launch the wizard. The first screen shows that the ObjectDataSource is configured to work with the `CategoriesBLL` class. Hit Next. Currently, only the ObjectDataSource s `InsertMethod` and `SelectMethod` properties are specified. However, the wizard auto-populated the drop-down lists in the UPDATE and DELETE tabs with the `UpdateCategory` and `DeleteCategory` methods, respectively. This is because in the `CategoriesBLL` class we marked these methods using the `DataObjectMethodAttribute` as the default methods for updating and deleting.

For now, set the UPDATE tab s drop-down list to (None), but leave the DELETE tab s drop-down list set to `DeleteCategory`. We'll return to this wizard in Step 6 to add updating support.

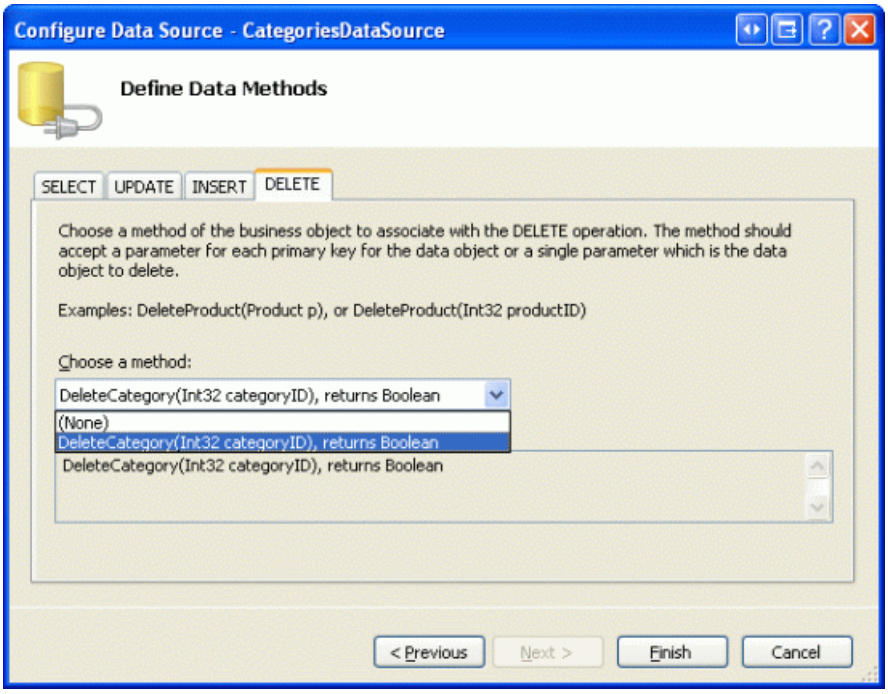


Figure 4: Configure the ObjectDataSource to Use the `DeleteCategory` Method (Click to view full-size image)

> **NOTE**
>
> Upon completing the wizard, Visual Studio may ask if you want to Refresh Fields and Keys, which will regenerate the data Web controls fields. Choose No, because choosing Yes will overwrite any field customizations you may have made.

The ObjectDataSource will now include a value for its `DeleteMethod` property as well as a `DeleteParameter`. Recall that when using the wizard to specify the methods, Visual Studio sets the ObjectDataSource s `OldValuesParameterFormatString` property to `original_{0}`, which causes problems with the update and delete method invocations. Therefore, either clear out this property altogether or reset it to the default, `{0}`. If you need to refresh your memory on this ObjectDataSource property, see the An Overview of Inserting, Updating, and Deleting Data tutorial.

After completing the wizard and fixing the `OldValuesParameterFormatString`, the ObjectDataSource s declarative markup should look similar like the following:

```
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="{0}" SelectMethod="GetCategories"
    TypeName="CategoriesBLL" InsertMethod="InsertWithPicture"
    DeleteMethod="DeleteCategory">
    <InsertParameters>
        <asp:Parameter Name="categoryName" Type="String" />
        <asp:Parameter Name="description" Type="String" />
        <asp:Parameter Name="brochurePath" Type="String" />
        <asp:Parameter Name="picture" Type="Object" />
    </InsertParameters>
    <DeleteParameters>
        <asp:Parameter Name="categoryID" Type="Int32" />
    </DeleteParameters>
</asp:ObjectDataSource>
```

After configuring the ObjectDataSource, add deleting capabilities to the GridView by checking the Enable Deleting checkbox from the GridView s smart tag. This will add a CommandField to the GridView whose `ShowDeleteButton` property is set to `true`.

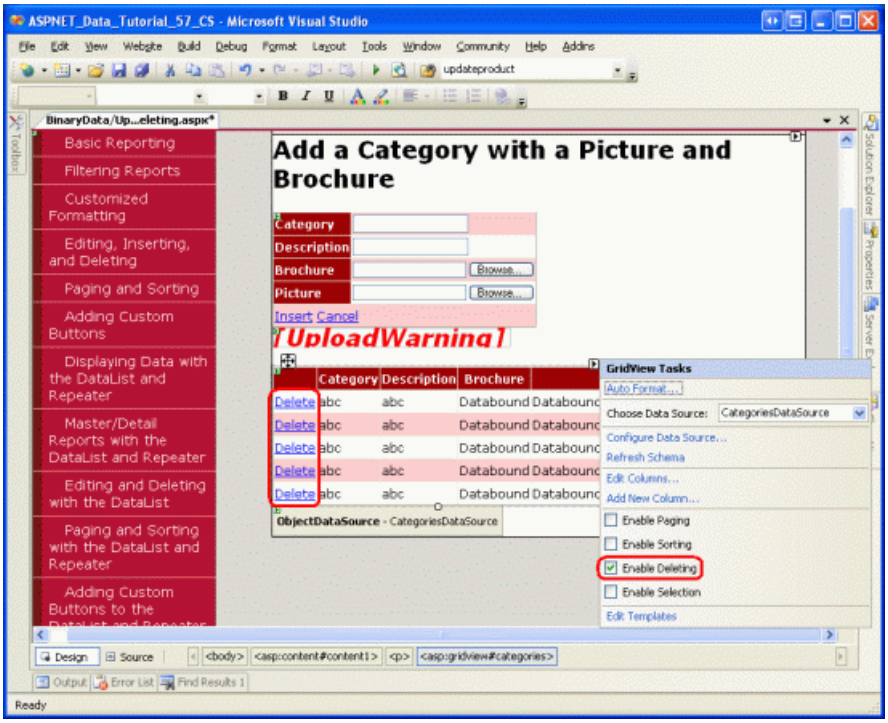


Figure 5: Enable Support for Deleting in the GridView (Click to view full-size image)

Take a moment to test out the delete functionality. There is a foreign key between the `Products` table s `CategoryID` and the `Categories` table s `CategoryID`, so you will get a foreign key constraint violation exception if you attempt to delete any of the first eight categories. To test this functionality out, add a new category, providing both a brochure and picture. My test category, shown in Figure 6, includes a test brochure file named `Test.pdf` and a test picture. Figure 7 shows the GridView after the test category has been added.

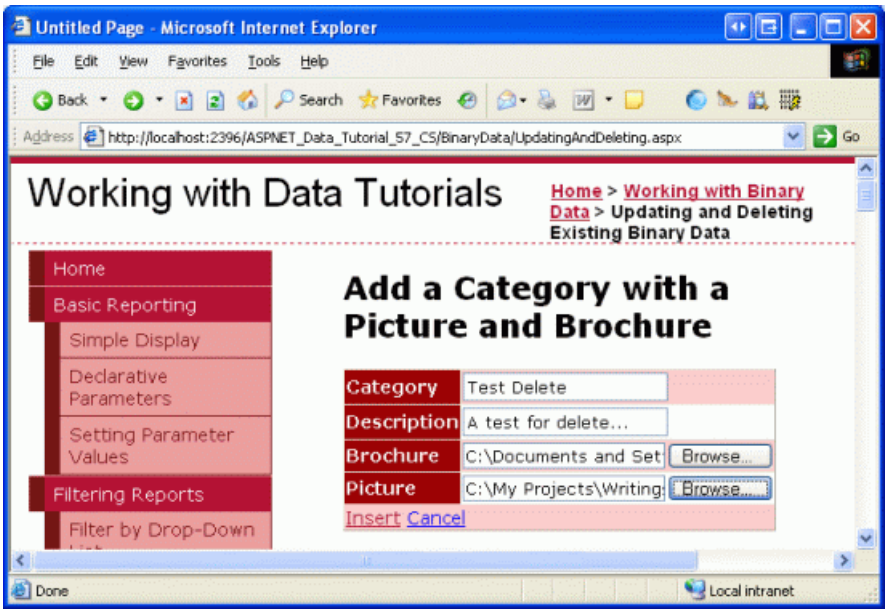


Figure 6: Add a Test Category with a Brochure and Image (Click to view full-size image)

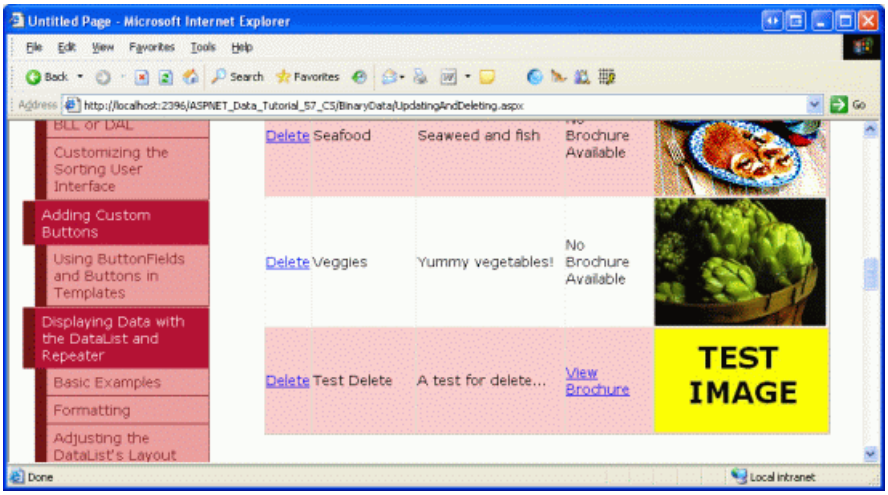


Figure 7: After Inserting the Test Category, it is Displayed in the GridView (Click to view full-size image)

In Visual Studio, refresh the Solution Explorer. You should now see a new file in the `~/Brochures` folder, `Test.pdf` (see Figure 8).

Next, click the Delete link in the Test Category row, causing the page to postback and the `CategoriesBLL` class s `DeleteCategory` method to fire. This will invoke the DAL s `Delete` method, causing the appropriate `DELETE` statement to be sent to the database. The data is then rebound to the GridView and the markup is sent back to the client with the Test Category no longer present.

While the delete workflow successfully removed the Test Category record from the `Categories` table, it did not remove its brochure file from the web server s file system. Refresh the Solution Explorer and you will see that `Test.pdf` is still sitting in the `~/Brochures` folder.

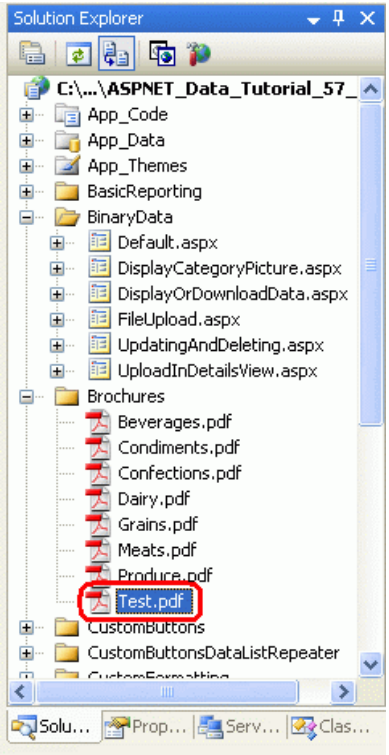


Figure 8: The `Test.pdf` File Was Not Deleted from the Web Server s File System

Step 5: Removing the Deleted Category s Brochure File

One of the downsides of storing binary data external to the database is that extra steps must be taken to clean up these files when the associated database record is deleted. The GridView and ObjectDataSource provide events that fire both before and after the delete command has been performed. We actually need to create event handlers for both the pre- and post-action events. Before the `Categories` record is deleted we need to determine its PDF file s path, but we don t want to delete the PDF before the category is deleted in case there is some exception and the category is not deleted.

The GridView s `RowDeleting` event fires before the ObjectDataSource s delete command has been invoked, while its `RowDeleted` event fires after. Create event handlers for these two events using the following code:

```
// A page variable to "remember" the deleted category's BrochurePath value
string deletedCategorysPdfPath = null;
protected void Categories_RowDeleting(object sender, GridViewDeleteEventArgs e)
{
    // Determine the PDF path for the category being deleted...
    int categoryID = Convert.ToInt32(e.Keys["CategoryID"]);
    CategoriesBLL categoryAPI = new CategoriesBLL();
    Northwind.CategoriesDataTable categories =
        categoryAPI.GetCategoryByCategoryID(categoryID);
    Northwind.CategoriesRow category = categories[0];
    if (category.IsBrochurePathNull())
        deletedCategorysPdfPath = null;
    else
        deletedCategorysPdfPath = category.BrochurePath;
}
protected void Categories_RowDeleted(object sender, GridViewDeletedEventArgs e)
{
    // Delete the brochure file if there were no problems deleting the record
    if (e.Exception == null)
    {
        // Is there a file to delete?
        if (deletedCategorysPdfPath != null)
        {
            System.IO.File.Delete(Server.MapPath(deletedCategorysPdfPath));
        }
    }
}
```

In the `RowDeleting` event handler, the `CategoryID` of the row being deleted is grabbed from the GridView s `DataKeys` collection, which can be accessed in this event handler through the `e.Keys` collection. Next, the `CategoriesBLL` class s `GetCategoryByCategoryID(categoryID)` is invoked to return information about the record being deleted. If the returned `CategoriesDataRow` object has a non- `NULL``BrochurePath` value then it is stored in the page variable `deletedCategorysPdfPath` so that the file can be deleted in the `RowDeleted` event handler.

> **NOTE**
> Rather than retrieving the `BrochurePath` details for the `Categories` record being deleted in the `RowDeleting` event handler, we could have alternatively added the `BrochurePath` to the GridView s `DataKeyNames` property and accessed the record s value through the `e.Keys` collection. Doing so would slightly increase the GridView s view state size, but would reduce the amount of code needed and save a trip to the database.

After the ObjectDataSource s underlying delete command has been invoked, the GridView s `RowDeleted` event handler fires. If there were no exceptions in deleting the data and there is a value for `deletedCategorysPdfPath`, then the PDF is deleted from the file system. Note that this extra code is not needed to clean up the category s binary data associated with its picture. That s because the picture data is stored directly in the database, so deleting the `Categories` row also deletes that category s picture data.

After adding the two event handlers, run this test case again. When deleting the category, its associated PDF is also deleted.

Updating an existing record s associated binary data provides some interesting challenges. The remainder of this tutorial delves into adding update capabilities to the brochure and picture. Step 6 explores techniques for updating the brochure information while Step 7 looks at updating the picture.

Step 6: Updating a Category s Brochure

As discussed in the An Overview of Inserting, Updating, and Deleting Data tutorial, the GridView offers built-in row-level editing support that can be implemented by the tick of a checkbox if its underlying data source is

appropriately configured. Currently, the `CategoriesDataSource` ObjectDataSource is not yet configured to include updating support, so let s add that in.

Click the Configure Data Source link from the ObjectDataSource s wizard and proceed to the second step. Because of the `DataObjectMethodAttribute` used in `CategoriesBLL`, the UPDATE drop-down list should automatically be populated with the `UpdateCategory` overload that accepts four input parameters (for all columns but `Picture`). Change this so that it uses the overload with five parameters.

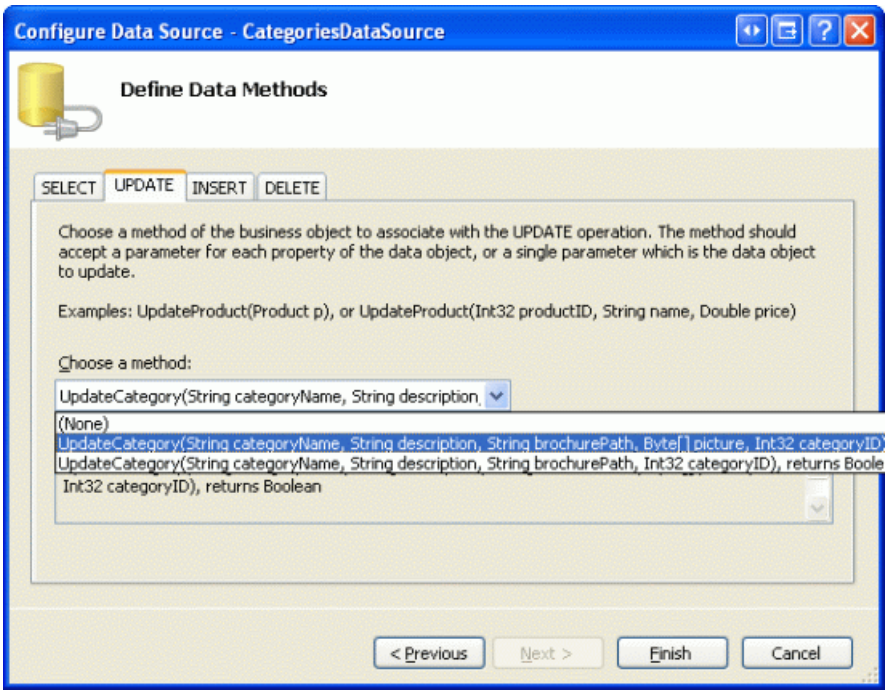


Figure 9: Configure the ObjectDataSource to Use the `UpdateCategory` Method that Includes a Parameter for `Picture` (Click to view full-size image)

The ObjectDataSource will now include a value for its `UpdateMethod` property as well as corresponding `UpdateParameter` s. As noted in Step 4, Visual Studio sets the ObjectDataSource s `OldValuesParameterFormatString` property to `original_{0}` when using the Configure Data Source wizard. This will cause problems with the update and delete method invocations. Therefore, either clear out this property altogether or reset it to the default, `{0}`.

After completing the wizard and fixing the `OldValuesParameterFormatString`, the ObjectDataSource s declarative markup should look like the following:

```
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="{0}" SelectMethod="GetCategories"
    TypeName="CategoriesBLL" InsertMethod="InsertWithPicture"
    DeleteMethod="DeleteCategory" UpdateMethod="UpdateCategory">
    <InsertParameters>
        <asp:Parameter Name="categoryName" Type="String" />
        <asp:Parameter Name="description" Type="String" />
        <asp:Parameter Name="brochurePath" Type="String" />
        <asp:Parameter Name="picture" Type="Object" />
    </InsertParameters>
    <DeleteParameters>
        <asp:Parameter Name="categoryID" Type="Int32" />
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="categoryName" Type="String" />
        <asp:Parameter Name="description" Type="String" />
        <asp:Parameter Name="brochurePath" Type="String" />
        <asp:Parameter Name="picture" Type="Object" />
        <asp:Parameter Name="categoryID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

To turn on the GridView s built-in editing features, check the Enable Editing option from the GridView s smart tag. This will set the CommandField s `ShowEditButton` property to `true`, resulting in the addition of an Edit button (and Update and Cancel buttons for the row being edited).

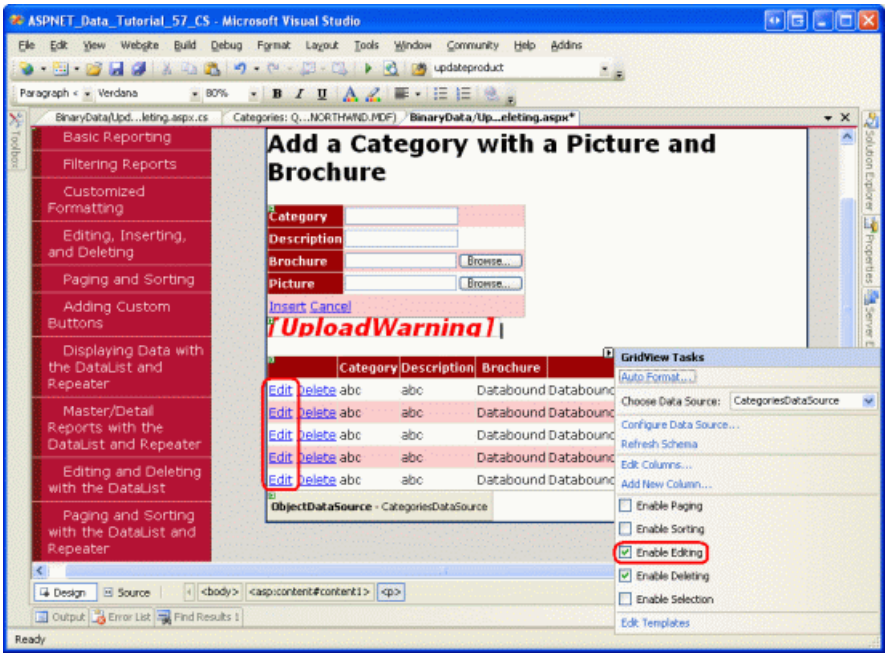


Figure 10: Configure the GridView to Support Editing (Click to view full-size image)

Visit the page through a browser and click one of the row s Edit buttons. The `CategoryName` and `Description` BoundFields are rendered as textboxes. The `BrochurePath` TemplateField lacks an `EditItemTemplate`, so it continues to show its `ItemTemplate` a link to the brochure. The `Picture` ImageField renders as a TextBox whose `Text` property is assigned the value of the ImageField s `DataImageUrlField` value, in this case `CategoryID`.

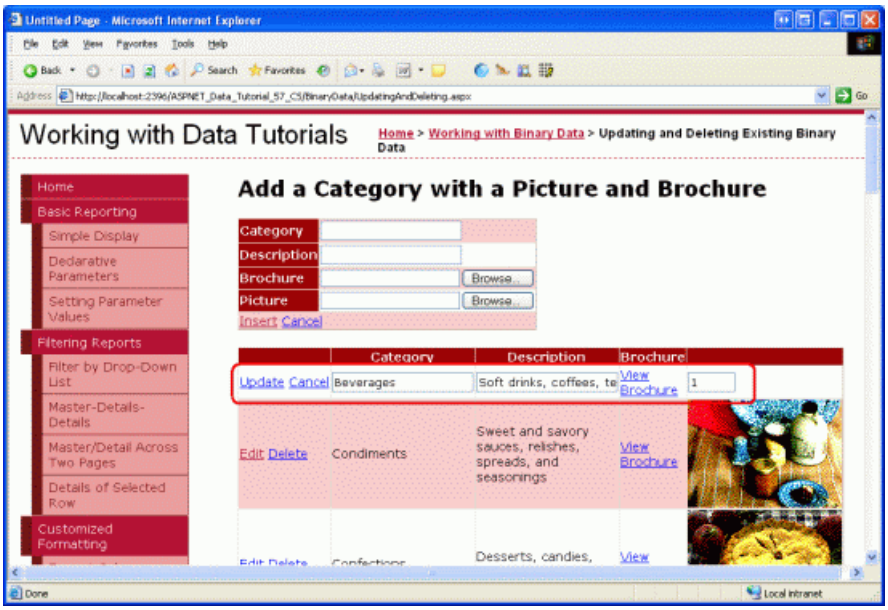


Figure 11: The GridView Lacks an Editing Interface for `BrochurePath` (Click to view full-size image)

Customizing the `BrochurePath` s Editing Interface

We need to create an editing interface for the `BrochurePath` TemplateField, one that allows the user to either:

- Leave the category s brochure as-is,
- Update the category s brochure by uploading a new brochure, or
- Remove the category s brochure altogether (in the case that the category no longer has an associated brochure).

We also need to update the `Picture` ImageField s editing interface, but we'll get to this in Step 7.

From the GridView s smart tag, click on the Edit Templates link and select the `BrochurePath` TemplateField s `EditItemTemplate` from the drop-down list. Add a RadioButtonList Web control to this template, setting its `ID` property to `BrochureOptions` and its `AutoPostBack` property to `true`. From the Properties window, click on the ellipses in the `Items` property, which will bring up the `ListItem` Collection Editor. Add the following three options with `Value` s 1, 2 and 3, respectively:

- Use current brochure
- Remove current brochure
- Upload new brochure

Set the first `ListItem` s `Selected` property to `true`.

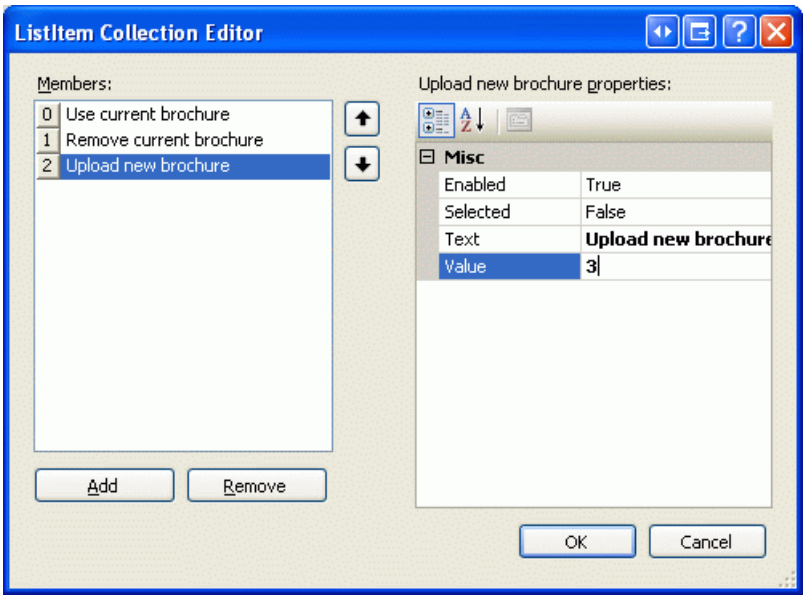


Figure 12: Add Three `ListItem` s to the RadioButtonList

Beneath the RadioButtonList, add a FileUpload control named `BrochureUpload`. Set its `Visible` property to `false`
.

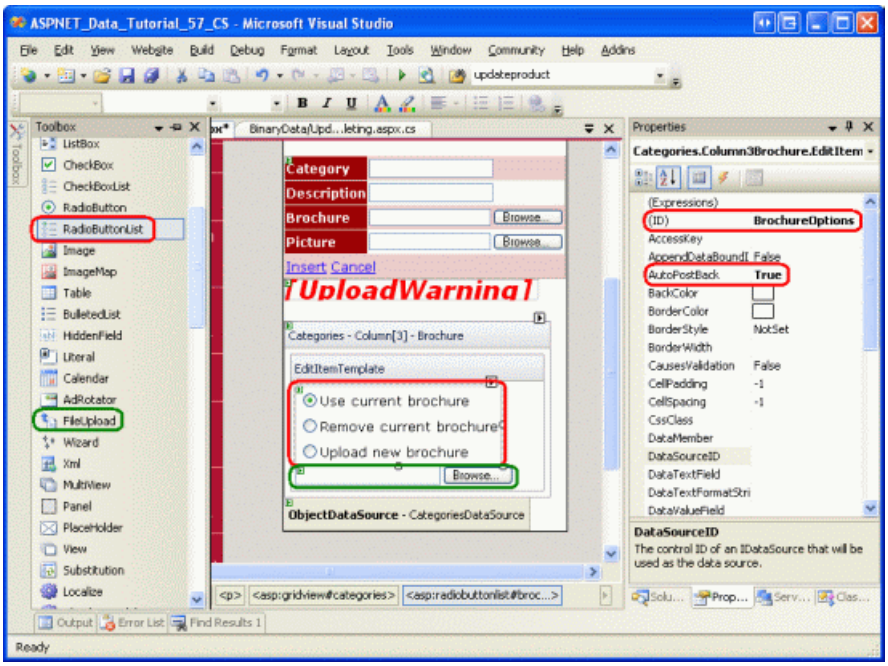


Figure 13: Add a RadioButtonList and FileUpload Control to the `EditItemTemplate` (Click to view full-size image)

This RadioButtonList provides the three options for the user. The idea is that the FileUpload control will be displayed only if the last option, Upload new brochure, is selected. To accomplish this, create an event handler for the RadioButtonList s `SelectedIndexChanged` event and add the following code:

```
protected void BrochureOptions_SelectedIndexChanged(object sender, EventArgs e)
{
    // Get a reference to the RadioButtonList and its Parent
    RadioButtonList BrochureOptions = (RadioButtonList)sender;
    Control parent = BrochureOptions.Parent;
    // Now use FindControl("controlID") to get a reference of the
    // FileUpload control
    FileUpload BrochureUpload =
        (FileUpload)parent.FindControl("BrochureUpload");
    // Only show BrochureUpload if SelectedValue = "3"
    BrochureUpload.Visible = (BrochureOptions.SelectedValue == "3");
}
```

Since the RadioButtonList and FileUpload controls are within a template, we have to write a bit of code to programmatically access these controls. The `SelectedIndexChanged` event handler is passed a reference of the RadioButtonList in the `sender` input parameter. To get the FileUpload control, we need to get the RadioButtonList s parent control and use the `FindControl("controlID")` method from there. Once we have a reference to both the RadioButtonList and FileUpload controls, the FileUpload control s `Visible` property is set to `true` only if the RadioButtonList s `SelectedValue` equals 3, which is the `Value` for the Upload new brochure `ListItem`.

With this code in place, take a moment to test out the editing interface. Click on the Edit button for a row. Initially, the Use current brochure option should be selected. Changing the selected index causes a postback. If the third option is selected, the FileUpload control is displayed, otherwise it is hidden. Figure 14 shows the editing interface when the Edit button is first clicked; Figure 15 shows the interface after the Upload new brochure option is selected.

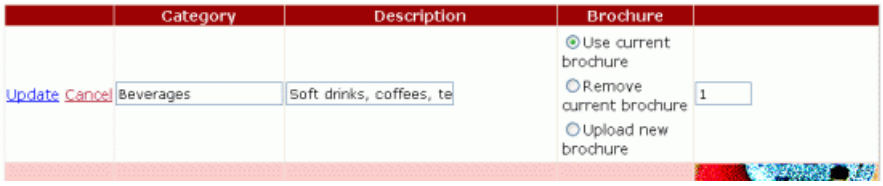


Figure 14: Initially, the Use current brochure Option is Selected (Click to view full-size image)

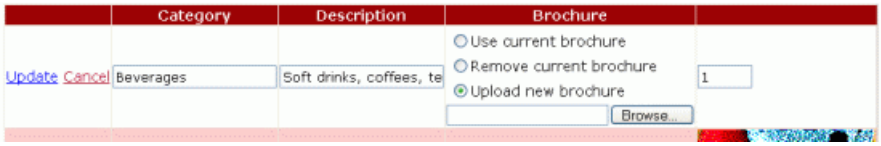


Figure 15: Choosing the Upload new brochure Option Displays the FileUpload Control (Click to view full-size image)

Saving the Brochure File and Updating the `BrochurePath` Column

When the GridView s Update button is clicked, its `RowUpdating` event fires. The ObjectDataSource s update command is invoked and then the GridView s `RowUpdated` event fires. Like with the deleting workflow, we need to create event handlers for both of these events. In the `RowUpdating` event handler, we need to determine what action to take based on the `SelectedValue` of the `BrochureOptions` RadioButtonList:

- If the `SelectedValue` is 1, we want to keep using the same `BrochurePath` setting. Therefore, we need to set the ObjectDataSource s `brochurePath` parameter to the existing `BrochurePath` value of the record being updated. The ObjectDataSource s `brochurePath` parameter can be set using `e.NewValues["brochurePath"] = value`.
- If the `SelectedValue` is 2, then we want to set the record s `BrochurePath` value to `NULL`. This can be accomplished by setting the ObjectDataSource s `brochurePath` parameter to `Nothing`, which results in a database `NULL` being used in the `UPDATE` statement. If there is an existing brochure file that is being removed, we need to delete the existing file. However, we only want to do this if the update completes without raising an

exception.

- If the `SelectedValue` is 3, then we want to ensure that the user has uploaded a PDF file and then save it to the file system and update the record s `BrochurePath` column value. Moreover, if there is an existing brochure file that is being replaced, we need to delete the previous file. However, we only want to do this if the update completes without raising an exception.

The steps needed to be completed when the RadioButtonList s `SelectedValue` is 3 are virtually identical to those used by the DetailsView s `ItemInserting` event handler. This event handler is executed when a new category record is added from the DetailsView control we added in the previous tutorial. Therefore, it behooves us to refactor this functionality out into separate methods. Specifically, I moved out the common functionality into two methods:

- `ProcessBrochureUpload(FileUpload, out bool)` accepts as input a FileUpload control instance and an output Boolean value that specifies whether the delete or edit operation should proceed or if it should be cancelled due to some validation error. This method returns the path to the saved file or `null` if no file was saved.
- `DeleteRememberedBrochurePath` deletes the file specified by the path in the page variable `deletedCategorysPdfPath` if `deletedCategorysPdfPath` is not `null`.

The code for these two methods follows. Note the similarity between `ProcessBrochureUpload` and the DetailsView s `ItemInserting` event handler from the previous tutorial. In this tutorial I have updated the DetailsView s event handlers to use these new methods. Download the code associated with this tutorial to see the modifications to the DetailsView s event handlers.

```
private string ProcessBrochureUpload
    (FileUpload BrochureUpload, out bool CancelOperation)
{
    CancelOperation = false;    // by default, do not cancel operation
    if (BrochureUpload.HasFile)
    {
        // Make sure that a PDF has been uploaded
        if (string.Compare(System.IO.Path.GetExtension(BrochureUpload.FileName),
            ".pdf", true) != 0)
        {
            UploadWarning.Text =
                "Only PDF documents may be used for a category's brochure.";
            UploadWarning.Visible = true;
            CancelOperation = true;
            return null;
        }
        const string BrochureDirectory = "~/Brochures/";
        string brochurePath = BrochureDirectory + BrochureUpload.FileName;
        string fileNameWithoutExtension =
            System.IO.Path.GetFileNameWithoutExtension(BrochureUpload.FileName);
        int iteration = 1;
        while (System.IO.File.Exists(Server.MapPath(brochurePath)))
        {
            brochurePath = string.Concat(BrochureDirectory, fileNameWithoutExtension,
                "-", iteration, ".pdf");
            iteration++;
        }
        // Save the file to disk and set the value of the brochurePath parameter
        BrochureUpload.SaveAs(Server.MapPath(brochurePath));
        return brochurePath;
    }
    else
    {
        // No file uploaded
        return null;
    }
}
private void DeleteRememberedBrochurePath()
{
    // Is there a file to delete?
    if (deletedCategorysPdfPath != null)
    {
        System.IO.File.Delete(Server.MapPath(deletedCategorysPdfPath));
    }
}
```

The GridView s `RowUpdating` and `RowUpdated` event handlers use the `ProcessBrochureUpload` and `DeleteRememberedBrochurePath` methods, as the following code shows:

```
protected void Categories_RowUpdating(object sender, GridViewUpdateEventArgs e)
{
    // Reference the RadioButtonList
    RadioButtonList BrochureOptions =
        (RadioButtonList)Categories.Rows[e.RowIndex].FindControl("BrochureOptions");
    // Get BrochurePath information about the record being updated
    int categoryID = Convert.ToInt32(e.Keys["CategoryID"]);
    CategoriesBLL categoryAPI = new CategoriesBLL();
    Northwind.CategoriesDataTable categories =
        categoryAPI.GetCategoryByCategoryID(categoryID);
    Northwind.CategoriesRow category = categories[0];
    if (BrochureOptions.SelectedValue == "1")
    {
        // Use current value for BrochurePath
        if (category.IsBrochurePathNull())
            e.NewValues["brochurePath"] = null;
        else
            e.NewValues["brochurePath"] = category.BrochurePath;
    }
    else if (BrochureOptions.SelectedValue == "2")
    {
        // Remove the current brochure (set it to NULL in the database)
        e.NewValues["brochurePath"] = null;
    }
    else if (BrochureOptions.SelectedValue == "3")
    {
        // Reference the BrochurePath FileUpload control
        FileUpload BrochureUpload =
            (FileUpload)Categories.Rows[e.RowIndex].FindControl("BrochureUpload");
        // Process the BrochureUpload
        bool cancelOperation = false;
        e.NewValues["brochurePath"] =
            ProcessBrochureUpload(BrochureUpload, out cancelOperation);
        e.Cancel = cancelOperation;
    }
    else
    {
        // Unknown value!
        throw new ApplicationException(
            string.Format("Invalid BrochureOptions value, {0}",
                BrochureOptions.SelectedValue));
    }
    if (BrochureOptions.SelectedValue == "2" ||
        BrochureOptions.SelectedValue == "3")
    {
        // "Remember" that we need to delete the old PDF file
        if (category.IsBrochurePathNull())
            deletedCategorysPdfPath = null;
        else
            deletedCategorysPdfPath = category.BrochurePath;
    }
}
protected void Categories_RowUpdated(object sender, GridViewUpdatedEventArgs e)
{
    // If there were no problems and we updated the PDF file,
    // then delete the existing one
    if (e.Exception == null)
    {
        DeleteRememberedBrochurePath();
    }
}
```

Note how the `RowUpdating` event handler uses a series of conditional statements to perform the appropriate action based on the `BrochureOptions` RadioButtonList s `SelectedValue` property value.

With this code in place, you can edit a category and have it use its current brochure, use no brochure, or upload a

new one. Go ahead and try it out. Set breakpoints in the `RowUpdating` and `RowUpdated` event handlers to get a sense of the workflow.

Step 7: Uploading a New Picture

The `Picture` ImageField s editing interface renders as a textbox populated with the value from its `DataImageUrlField` property. During the editing workflow, the GridView passes a parameter to the ObjectDataSource with the parameter s name the value of the ImageField s `DataImageUrlField` property and the parameter s value the value entered into the textbox in the editing interface. This behavior is suitable when the image is saved as a file on the file system and the `DataImageUrlField` contains the full URL of the image. With such circumstances, the editing interface displays the image s URL in the textbox, which the user can change and have saved back to the database. Granted, this default interface doesn t allow the user to upload a new image, but it does let them change the URL of the image from the current value to another. For this tutorial, however, the ImageField s default editing interface does not suffice because the `Picture` binary data is being stored directly in the database and the `DataImageUrlField` property holds just the `CategoryID`.

To better understand what happens in our tutorial when a user edits a row with an ImageField, consider the following example: a user edits a row with `CategoryID` 10, causing the `Picture` ImageField to render as a textbox with the value 10. Imagine that the user changes the value in this textbox to 50 and clicks the Update button. A postback occurs and the GridView initially creates a parameter named `CategoryID` with the value 50. However, before the GridView sends this parameter (and the `CategoryName` and `Description` parameters), it adds in the values from the `DataKeys` collection. Therefore, it overwrites the `CategoryID` parameter with the current row s underlying `CategoryID` value, 10. In short, the ImageField s editing interface has no affect on the editing workflow for this tutorial because names of the ImageField s `DataImageUrlField` property and the grid s `DataKey` value are one in the same.

While the ImageField makes it easy to display an image based on database data, we don t want to provide a textbox in the editing interface. Rather, we want to offer a FileUpload control that the end user can use to change the category s picture. Unlike the `BrochurePath` value, for these tutorials we ve decided to require that each category must have a picture. Therefore, we don t need to let the user indicate that there is no associated picture the user may either upload a new picture or leave the current picture as-is.

To customize the ImageField s editing interface, we need to convert it into a TemplateField. From the GridView s smart tag, click on the Edit Columns link, select the ImageField, and click the Convert this field into a TemplateField link.

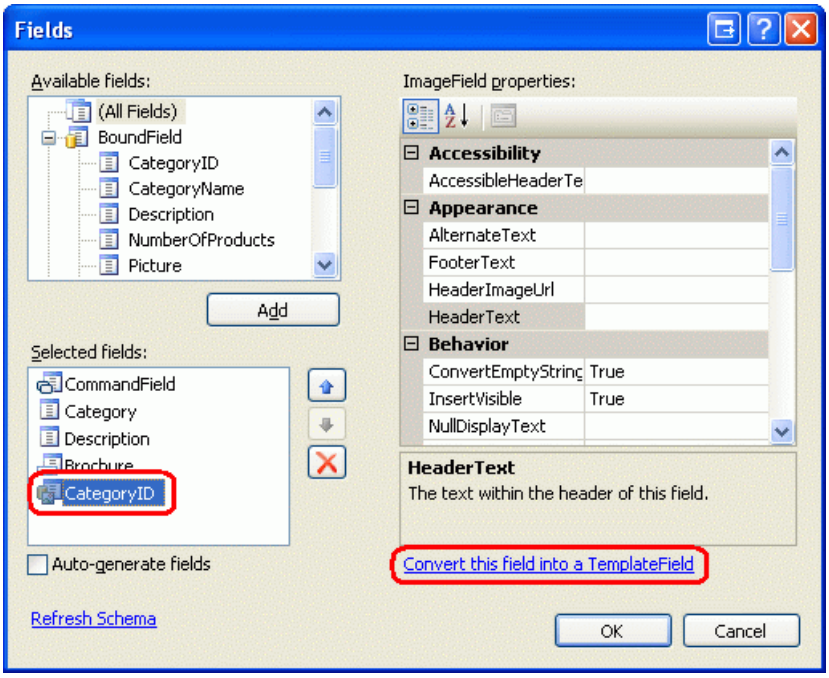


Figure 16: Convert the ImageField Into a TemplateField

Converting the ImageField into a TemplateField in this manner generates a TemplateField with two templates. As the following declarative syntax shows, the `ItemTemplate` contains an Image Web control whose `ImageUrl` property is assigned using databinding syntax based on the ImageField s `DataImageUrlField` and `DataImageUrlFormatString` properties. The `EditItemTemplate` contains a TextBox whose `Text` property is bound to the value specified by the `DataImageUrlField` property.

```
<asp:TemplateField>
    <EditItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
            Text='<%# Eval("CategoryID") %>'></asp:TextBox>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Image ID="Image1" runat="server"
            ImageUrl='<%# Eval("CategoryID",
                "DisplayCategoryPicture.aspx?CategoryID={0}") %>' />
    </ItemTemplate>
</asp:TemplateField>
```

We need to update the `EditItemTemplate` to use a FileUpload control. From the GridView s smart tag click on the Edit Templates link and then select the `Picture` TemplateField s `EditItemTemplate` from the drop-down list. In the template you should see a TextBox remove this. Next, drag a FileUpload control from the Toolbox into the template, setting its `ID` to `PictureUpload`. Also add the text To change the category s picture, specify a new picture. To keep the category s picture the same, leave the field empty to the template, as well.

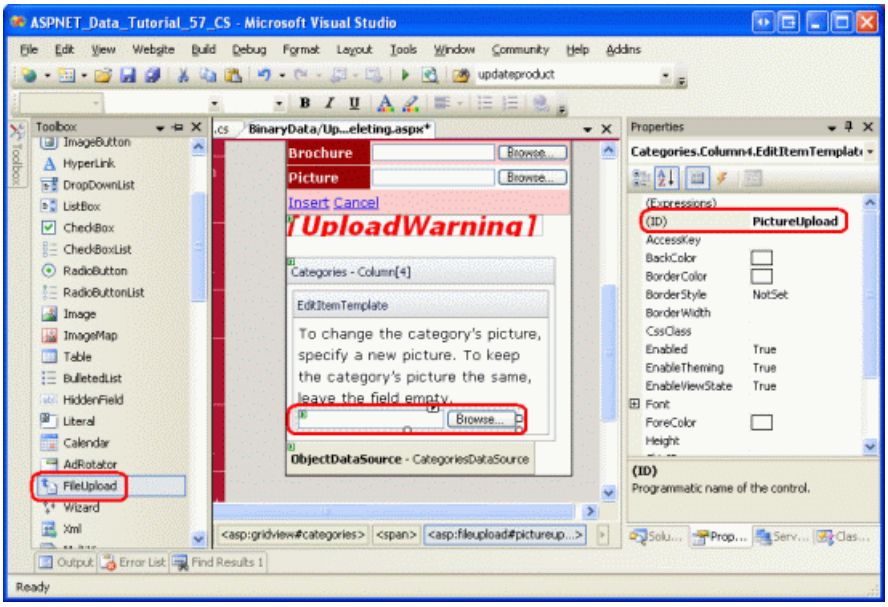


Figure 17: Add a FileUpload Control to the `EditItemTemplate` (Click to view full-size image)

After customizing the editing interface, view your progress in a browser. When viewing a row in read-only mode, the category s image is shown as it was before, but clicking on the Edit button renders the picture column as text with a FileUpload control.

Figure 18: The Editing Interface Includes a FileUpload Control (Click to view full-size image)

Recall that the ObjectDataSource is configured to call the `CategoriesBLL` class s `UpdateCategory` method that accepts as input the binary data for the picture as a `byte` array. If this array has a `null` value, however, the alternate `UpdateCategory` overload is called, which issues the `UPDATE` SQL statement that does not modify the `Picture` column, thereby leaving the category s current picture intact. Therefore, in the GridView s `RowUpdating` event handler we need to programmatically reference the `PictureUpload` FileUpload control and determine if a file was uploaded. If one was not uploaded, then we do *not* want to specify a value for the `picture` parameter. On the other hand, if a file was uploaded in the `PictureUpload` FileUpload control, we want to ensure that it is a JPG file. If it is, then we can send its binary contents to the ObjectDataSource through the `picture` parameter.

Like with the code used in Step 6, much of the code needed here already exists in the DetailsView s `ItemInserting` event handler. Therefore, I ve refactored the common functionality into a new method, `ValidPictureUpload`, and updated the `ItemInserting` event handler to use this method.

Add the following code to the start of the GridView s `RowUpdating` event handler. It s important that this code come before the code that saves the brochure file since we don t want to save the brochure to the web server s file system if an invalid picture file is uploaded.

```
// Reference the PictureUpload FileUpload
FileUpload PictureUpload =
    (FileUpload)Categories.Rows[e.RowIndex].FindControl("PictureUpload");
if (PictureUpload.HasFile)
{
    // Make sure the picture upload is valid
    if (ValidPictureUpload(PictureUpload))
    {
        e.NewValues["picture"] = PictureUpload.FileBytes;
    }
    else
    {
        // Invalid file upload, cancel update and exit event handler
        e.Cancel = true;
        return;
    }
}
```

The `ValidPictureUpload(FileUpload)` method takes in a FileUpload control as its sole input parameter and checks the uploaded file s extension to ensure that the uploaded file is a JPG; it is only called if a picture file is uploaded. If no file is uploaded, then the picture parameter is not set, and therefore uses its default value of `null`. If a picture was uploaded and `ValidPictureUpload` returns `true`, the `picture` parameter is assigned the binary data of the uploaded image; if the method returns `false`, the update workflow is cancelled and the event handler exited.

The `ValidPictureUpload(FileUpload)` method code, which was refactored from the DetailsView s `ItemInserting` event handler, follows:

```
private bool ValidPictureUpload(FileUpload PictureUpload)
{
    // Make sure that a JPG has been uploaded
    if (string.Compare(System.IO.Path.GetExtension(PictureUpload.FileName),
            ".jpg", true) != 0 &&
        string.Compare(System.IO.Path.GetExtension(PictureUpload.FileName),
            ".jpeg", true) != 0)
    {
        UploadWarning.Text =
            "Only JPG documents may be used for a category's picture.";
        UploadWarning.Visible = true;
        return false;
    }
    else
    {
        return true;
    }
}
```

Step 8: Replacing the Original Categories Pictures with JPGs

Recall that the original eight categories pictures are bitmap files wrapped in an OLE header. Now that we have added the capability to edit an existing record s picture, take a moment to replace these bitmaps with JPGs. If you want to continue to use the current category pictures, you can convert them to JPGs by performing the following steps:

1. Save the bitmap images to your hard drive. Visit the `UpdatingAndDeleting.aspx` page in your browser and for each of the first eight categories, right-click on the image and choose to save the picture.
2. Open the image in your image editor of choice. You can use Microsoft Paint, for example.
3. Save the bitmap as a JPG image.
4. Update the category s picture through the editing interface, using the JPG file.

After editing a category and uploading the JPG image, the image will not render in the browser because the `DisplayCategoryPicture.aspx` page is stripping the first 78 bytes from the pictures of the first eight categories. Fix this by removing the code that performs the OLE header stripping. After doing this, the `DisplayCategoryPicture.aspx``Page_Load` event handler should have just the following code:

```
protected void Page_Load(object sender, EventArgs e)
{
    int categoryID = Convert.ToInt32(Request.QueryString["CategoryID"]);
    // Get information about the specified category
    CategoriesBLL categoryAPI = new CategoriesBLL();
    Northwind.CategoriesDataTable categories = _
        categoryAPI.GetCategoryWithBinaryDataByCategoryID(categoryID);
    Northwind.CategoriesRow category = categories[0];
    // For new categories, images are JPGs...

    // Output HTTP headers providing information about the binary data
    Response.ContentType = "image/jpeg";
    // Output the binary data
    Response.BinaryWrite(category.Picture);
}
```

NOTE

The `UpdatingAndDeleting.aspx` page s inserting and editing interfaces could use a bit more work. The `CategoryName` and `Description` BoundFields in the DetailsView and GridView should be converted into TemplateFields. Since `CategoryName` does not allow `NULL` values, a RequiredFieldValidator should be added. And the `Description` TextBox should probably be converted into a multi-line TextBox. I leave these finishing touches as an exercise for you.

Summary

This tutorial completes our look at working with binary data. In this tutorial and the previous three, we saw how binary data can be stored on the file system or directly within the database. A user provides binary data to the system by selecting a file from their hard drive and uploading it to the web server, where it can be stored on the file system or inserted into the database. ASP.NET 2.0 includes a FileUpload control that makes providing such an interface as easy as drag and drop. However, as noted in the Uploading Files tutorial, the FileUpload control is only well-suited for relatively small file uploads, ideally not exceeding a megabyte. We also explored how to associate uploaded data with the underlying data model, as well as how to edit and delete the binary data from existing records.

Our next set of tutorials explores various caching techniques. Caching provides a means to improve an application s overall performance by taking the results from expensive operations and storing them in a location that can be more quickly accessed.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Uploading Files (VB)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> Learn how to allow users to upload binary files (such as Word or PDF documents) to your Web site where they may be stored in either the server's file system or the database.

Introduction

All of the tutorials we ve examined so far have worked exclusively with text data. However, many applications have data models that capture both text and binary data. An online dating site might allow users to upload a picture to associate with their profile. A recruiting website might let users upload their resume as a Microsoft Word or PDF document.

Working with binary data adds a new set of challenges. We must decide how the binary data is stored in the application. The interface used for inserting new records has to be updated to allow the user to upload a file from their computer and extra steps must be taken to display or provide a means for downloading a record s associated binary data. In this tutorial and the next three we'll explore how to hurdle these challenges. At the end of these tutorials we'll have built a fully functional application that associates a picture and PDF brochure with each category. In this particular tutorial we'll look at different techniques for storing binary data and explore how to enable users to upload a file from their computer and have it saved on the web server s file system.

> **NOTE**
>
> Binary data that is part of an application s data model is sometimes referred to as a BLOB, an acronym for Binary Large OBject. In these tutorials I have chosen to use the terminology binary data, although the term BLOB is synonymous.

Step 1: Creating the Working with Binary Data Web Pages

Before we begin to explore the challenges associated with adding support for binary data, let s first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial and the next three. Start by adding a new folder named `BinaryData`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `FileUpload.aspx`
- `DisplayOrDownloadData.aspx`
- `UploadInDetailsView.aspx`
- `UpdatingAndDeleting.aspx`

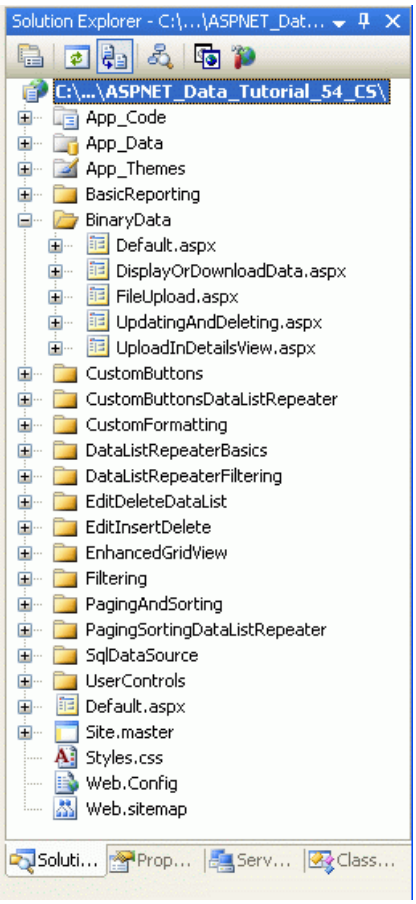


Figure 1: Add the ASP.NET Pages for the Binary Data-Related Tutorials

Like in the other folders, `Default.aspx` in the `BinaryData` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

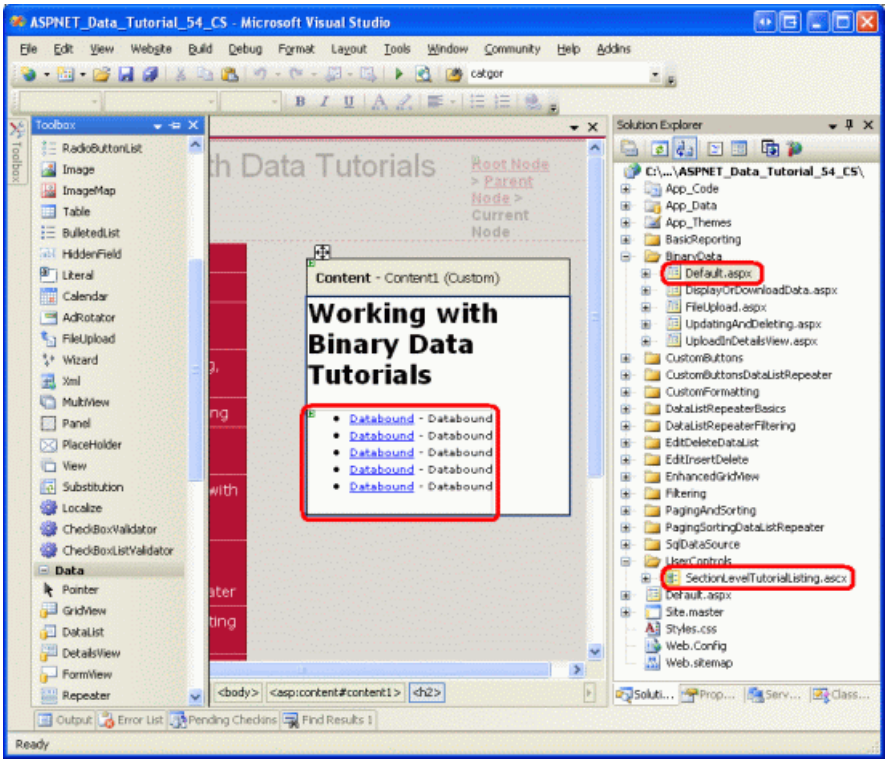


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add these pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Enhancing the GridView `<siteMapNode>`:

```
<siteMapNode
    title="Working with Binary Data"
    url="~/BinaryData/Default.aspx"
    description="Extend the data model to include collecting binary data.">

    <siteMapNode
        title="Uploading Files"
        url="~/BinaryData/FileUpload.aspx"
        description="Examine the different ways to store binary data on the
                     web server and see how to accept uploaded files from users
                     with the FileUpload control." />
    <siteMapNode
        title="Display or Download Binary Data"
        url="~/BinaryData/DisplayOrDownloadData.aspx"
        description="Let users view or download the captured binary data." />
    <siteMapNode
        title="Adding New Binary Data"
        url="~/BinaryData/UploadInDetailsView.aspx"
        description="Learn how to augment the inserting interface to
                     include a FileUpload control." />
    <siteMapNode
        title="Updating and Deleting Existing Binary Data"
        url="~/BinaryData/UpdatingAndDeleting.aspx"
        description="Learn how to update and delete existing binary data." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the Working with Binary Data tutorials.

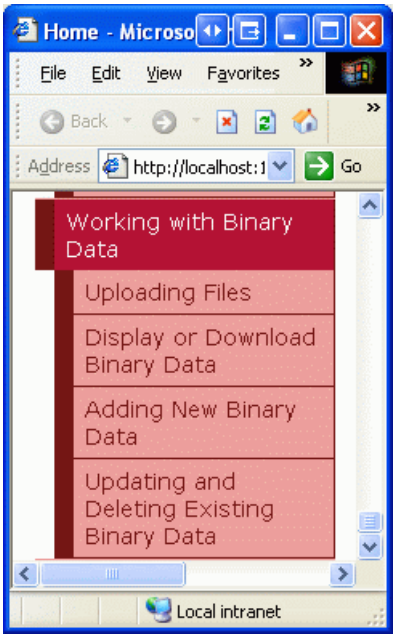


Figure 3: The Site Map Now Includes Entries for the Working with Binary Data Tutorials

Step 2: Deciding Where to Store the Binary Data

Binary data that is associated with the application s data model can be stored in one of two places: on the web server s file system with a reference to the file stored in the database; or directly within the database itself (see Figure 4). Each approach has its own set of pros and cons and merits a more detailed discussion.

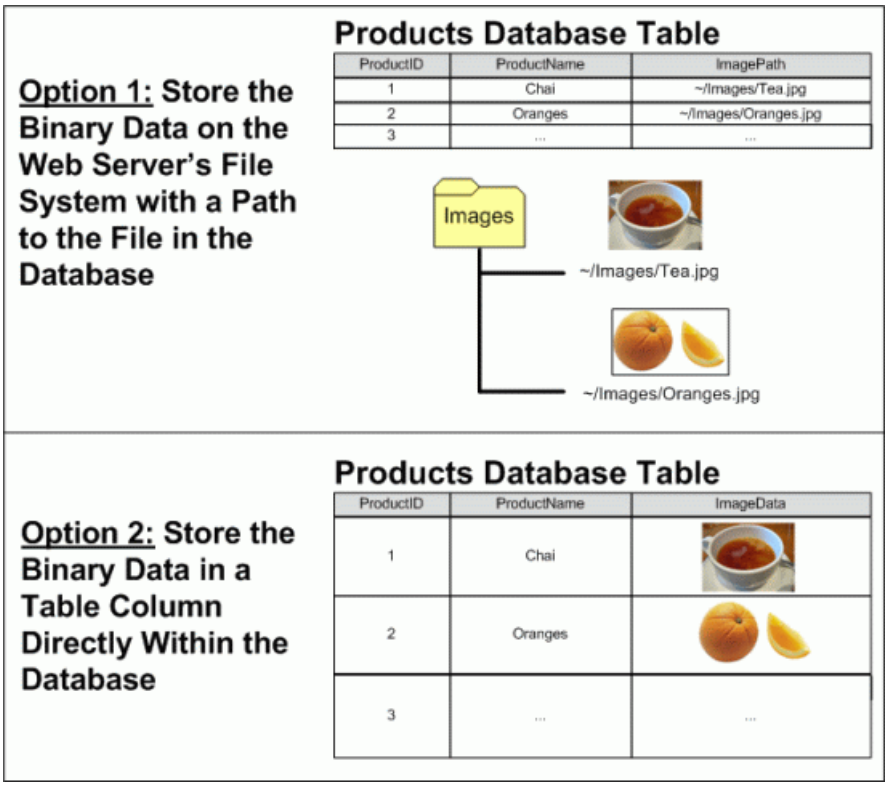


Figure 4: Binary Data Can Be Stored On the File System or Directly in the Database (Click to view full-size image)

Imagine that we wanted to extend the Northwind database to associate a picture with each product. One option would be to store these image files on the web server s file system and record the path in the `Products` table. With this approach, we d add an `ImagePath` column to the `Products` table of type `varchar(200)`, perhaps. When a user uploaded a picture for Chai, that picture might be stored on the web server s file system at `~/Images/Tea.jpg`, where `~` represents the application s physical path. That is, if the web site is rooted at the physical path `C:\Websites\Northwind\`, `~/Images/Tea.jpg` would be equivalent to `C:\Websites\Northwind\Images\Tea.jpg`. After uploading the image file, we d update the Chai record in the `Products` table so that its `ImagePath` column referenced the path of the new image. We could use `~/Images/Tea.jpg` or just `Tea.jpg` if we decided that all product images would be placed in the application s `Images` folder.

The main advantages of storing the binary data on the file system are:

- **Ease of implementation** as we'll see shortly, storing and retrieving binary data stored directly within the database involves a bit more code than when working with data through the file system. Additionally, in order for a user to view or download binary data they must be presented with a URL to that data. If the data resides on the web server s file system, the URL is straightforward. If the data is stored in the database, however, a web page needs to be created that will retrieve and return the data from the database.
- **Wider access to the binary data** the binary data may need to be accessible to other services or applications, ones that cannot pull the data from the database. For example, the images associated with each product might also need to be available to users through FTP, in which case we d want to store the binary data on the file system.
- **Performance** if the binary data is stored on the file system, the demand and network congestion between the database server and web server will be less than if the binary data is stored directly within the database.

The main disadvantage of storing binary data on the file system is that it decouples the data from the database. If a record is deleted from the `Products` table, the associated file on the web server s file system is not automatically deleted. We must write extra code to delete the file or the file system will become cluttered with unused, orphaned files. Furthermore, when backing up the database, we must make sure to make backups of the associated binary data on the file system, as well. Moving the database to another site or server poses similar challenges.

Alternatively, binary data can be stored directly in a Microsoft SQL Server 2005 database by creating a column of type `varbinary`. Like with other variable length data types, you can specify a maximum length of the binary data that can be held in this column. For example, to reserve at most 5,000 bytes use `varbinary(5000)`; `varbinary(MAX)` allows for the maximum storage size, about 2 GB.

The main advantage of storing binary data directly in the database is the tight coupling between the binary data and the database record. This greatly simplifies database administration tasks, like backups or moving the database to a different site or server. Also, deleting a record automatically deletes the corresponding binary data. There are also more subtle benefits of storing the binary data in the database. See Storing Binary Files Directly in the Database Using ASP.NET 2.0 for a more in-depth discussion.

> **NOTE**
> In Microsoft SQL Server 2000 and earlier versions, the `varbinary` data type had a maximum limit of 8,000 bytes. To store up to 2 GB of binary data the `image` data type needs to be used instead. With the addition of `MAX` in SQL Server 2005, however, the `image` data type has been deprecated. It s still supported for backwards compatibility, but Microsoft has announced that the `image` data type will be removed in a future version of SQL Server.

If you are working with an older data model you may see the `image` data type. The Northwind database s `Categories` table has a `Picture` column that can be used to store the binary data of an image file for the category. Since the Northwind database has its roots in Microsoft Access and earlier versions of SQL Server, this column is of type `image`.

For this tutorial and the next three, we'll use both approaches. The `Categories` table already has a `Picture` column for storing the binary content of an image for the category. We'll add an additional column, `BrochurePath`, to store a path to a PDF on the web server s file system that can be used to provide a print-quality, polished overview of the category.

Step 3: Adding the `BrochurePath` Column to the `Categories` Table

Currently the Categories table has only four columns: `CategoryID`, `CategoryName`, `Description`, and `Picture`. In addition to these fields, we need to add a new one that will point to the category s brochure (if one exists). To add this column, go to the Server Explorer, drill down into the Tables, right-click on the `Categories` table and choose Open Table Definition (see Figure 5). If you do not see the Server Explorer, bring it up by selecting the Server Explorer option from the View menu, or hit Ctrl+Alt+S.

Add a new `varchar(200)` column to the `Categories` table that is named `BrochurePath` and allows `NULL` s and click the Save icon (or hit Ctrl+S).

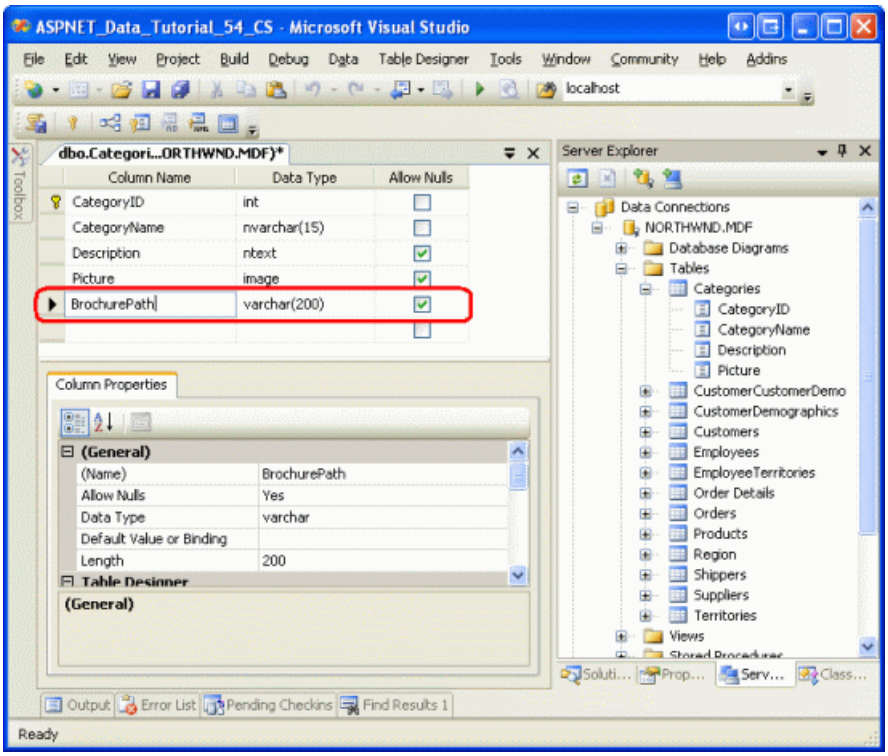


Figure 5: Add a `BrochurePath` Column to the `Categories` Table (Click to view full-size image)

Step 4: Updating the Architecture to Use the `Picture` and `BrochurePath` Columns

The `CategoriesDataTable` in the Data Access Layer (DAL) currently has four `DataColumn` s defined: `CategoryID`, `CategoryName`, `Description`, and `NumberOfProducts`. When we originally designed this DataTable in the Creating a Data Access Layer tutorial, the `CategoriesDataTable` only had the first three columns; the `NumberOfProducts` column was added in the Master/Detail Using a Bulleted List of Master Records with a Details DataList tutorial.

As discussed in *Creating a Data Access Layer*, the DataTables in the Typed DataSet make up the business objects. The TableAdapters are responsible for communicating with the database and populating the business objects with the query results. The `CategoriesDataTable` is populated by the `CategoriesTableAdapter`, which has three data retrieval methods:

- `GetCategories()` executes the TableAdapter s main query and returns the `CategoryID`, `CategoryName`, and `Description` fields of all records in the `Categories` table. The main query is what is used by the auto-generated `Insert` and `Update` methods.
- `GetCategoryByCategoryID(categoryID)` returns the `CategoryID`, `CategoryName`, and `Description` fields of the category whose `CategoryID` equals *categoryID*.
- `GetCategoriesAndNumberOfProducts()` - returns the `CategoryID`, `CategoryName`, and `Description` fields for all records in the `Categories` table. Also uses a subquery to return the number of products associated with each category.

Notice that none of these queries return the `Categories` table s `Picture` or `BrochurePath` columns; nor does the `CategoriesDataTable` provide `DataColumn` s for these fields. In order to work with the Picture and `BrochurePath` properties, we need to first add them to the `CategoriesDataTable` and then update the `CategoriesTableAdapter` class to return these columns.

Adding the `Picture` and `BrochurePath``DataColumn` s

Start by adding these two columns to the `CategoriesDataTable`. Right-click on the `CategoriesDataTable` s header,

select Add from the context menu and then choose the Column option. This will create a new `DataColumn` in the DataTable named `Column1`. Rename this column to `Picture`. From the Properties window, set the `DataColumn` s `DataType` property to `System.Byte[]` (this is not an option in the drop-down list; you need to type it in).

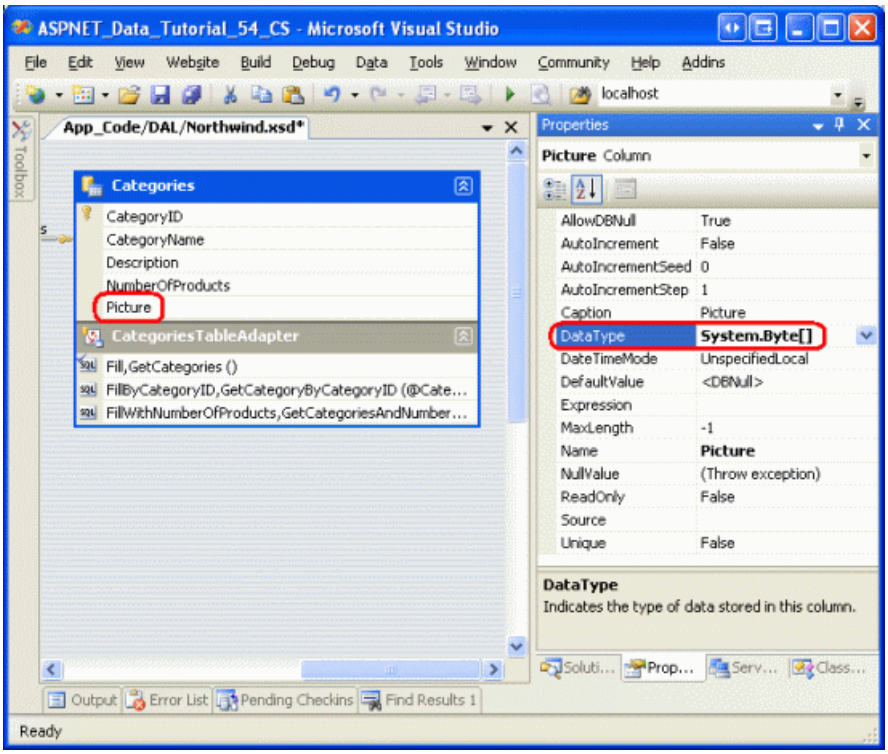


Figure 6: Create a `DataColumn` Named `Picture` whose `DataType` is `System.Byte[]` (Click to view full-size image)

Add another `DataColumn` to the DataTable, naming it `BrochurePath` using the default `DataType` value (`System.String`).

Returning the `Picture` and `BrochurePath` Values from the TableAdapter

With these two `DataColumn` s added to the `CategoriesDataTable`, we re ready to update the `CategoriesTableAdapter`. We could have both of these column values returned in the main TableAdapter query, but this would bring back the binary data every time the `GetCategories()` method was invoked. Instead, let s update the main TableAdapter query to bring back `BrochurePath` and create an additional data retrieval method that returns a particular category s `Picture` column.

To update the main TableAdapter query, right-click on the `CategoriesTableAdapter` s header and choose the Configure option from the context menu. This brings up the Table Adapter Configuration Wizard, which we ve seen in a number of past tutorials. Update the query to bring back the `BrochurePath` and click Finish.

Figure 7: Update the Column List in the `SELECT` Statement to Also Return `BrochurePath` (Click to view full-size image)

When using ad-hoc SQL statements for the TableAdapter, updating the column list in the main query updates the column list for all of the `SELECT` query methods in the TableAdapter. That means the `GetCategoryByCategoryID(categoryID)` method has been updated to return the `BrochurePath` column, which might be what we intended. However, it also updated the column list in the `GetCategoriesAndNumberOfProducts()` method, removing the subquery that returns the number of products for each category! Therefore, we need to update this method s `SELECT` query. Right-click on the `GetCategoriesAndNumberOfProducts()` method, choose Configure, and revert the `SELECT` query back to its original value:

```
SELECT CategoryID, CategoryName, Description,
       (SELECT COUNT(*)
            FROM Products p
            WHERE p.CategoryID = c.CategoryID)
       as NumberOfProducts
FROM Categories c
```

Next, create a new TableAdapter method that returns a particular category s `Picture` column value. Right-click on the `CategoriesTableAdapter` s header and choose the Add Query option to launch the TableAdapter Query Configuration Wizard. The first step of this wizard asks us if we want to query data using an ad-hoc SQL statement, a new stored procedure, or an existing one. Select Use SQL statements and click Next. Since we will be returning a row, choose the SELECT which returns rows option from the second step.

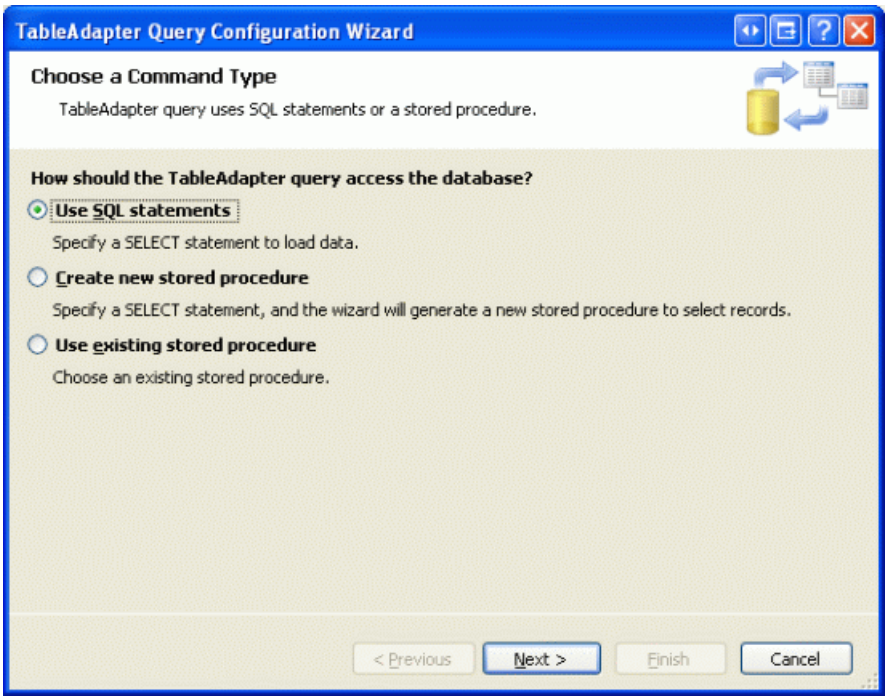


Figure 8: Select the Use SQL statements Option (Click to view full-size image)

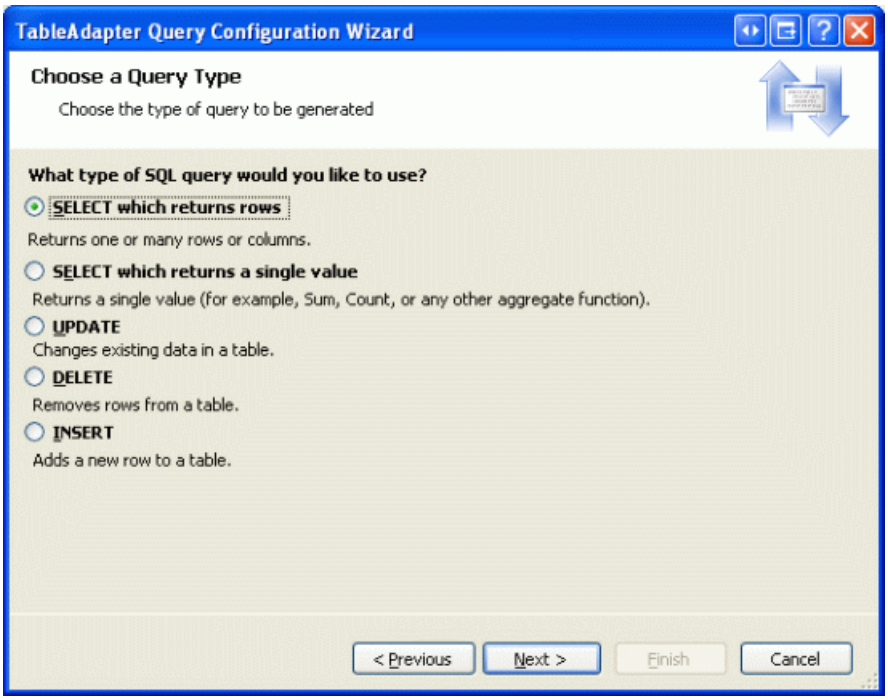


Figure 9: Since the Query Will Return a Record from the Categories Table, Choose SELECT which returns rows (Click to view full-size image)

In the third step, enter the following SQL query and click Next:

```
SELECT     CategoryID, CategoryName, Description, BrochurePath, Picture
FROM       Categories
WHERE      CategoryID = @CategoryID
```

The last step is to choose the name for the new method. Use `FillCategoryWithBinaryDataByCategoryID` and `GetCategoryWithBinaryDataByCategoryID` for the Fill a DataTable and Return a DataTable patterns, respectively. Click Finish to complete the wizard.

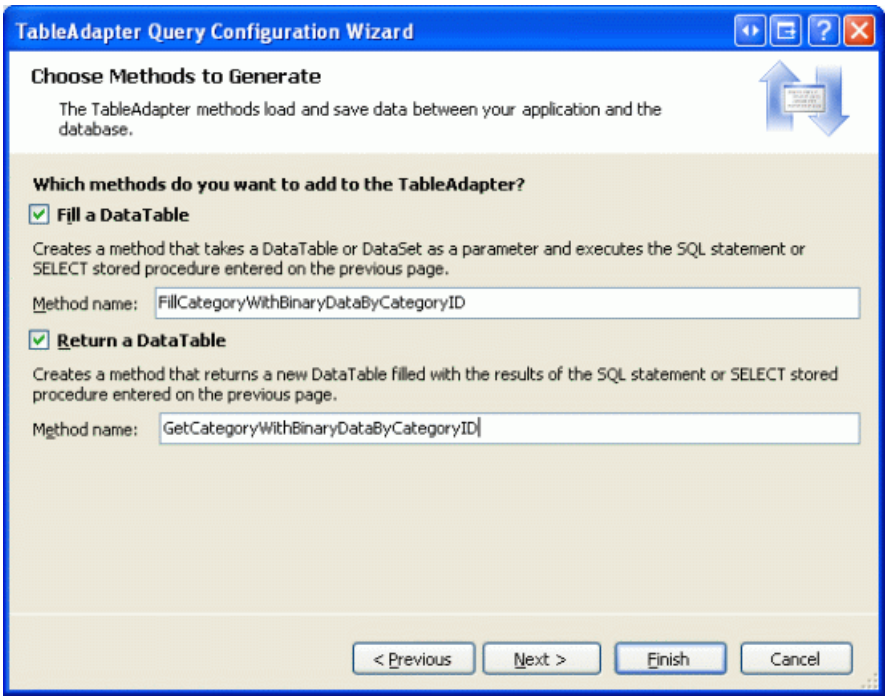


Figure 10: Choose the Names for the TableAdapter s Methods (Click to view full-size image)

> **NOTE**
>
> After completing the Table Adapter Query Configuration Wizard you may see a dialog box informing you that the new command text returns data with schema different from the schema of the main query. In short, the wizard is noting that the TableAdapter s main query `GetCategories()` returns a different schema than the one we just created. But this is what we want, so you can disregard this message.

Also, keep in mind that if you are using ad-hoc SQL statements and use the wizard to change the TableAdapter s main query at some later point in time, it will modify the `GetCategoryWithBinaryDataByCategoryID` method s `SELECT` statement s column list to include just those columns from the main query (that is, it will remove the `Picture` column from the query). You will have to manually update the column list to return the `Picture` column, similar to what we did with the `GetCategoriesAndNumberOfProducts()` method earlier in this step.

After adding the two `DataColumn` s to the `CategoriesDataTable` and the `GetCategoryWithBinaryDataByCategoryID` method to the `CategoriesTableAdapter`, these classes in the Typed DataSet Designer should look like the screenshot in Figure 11.

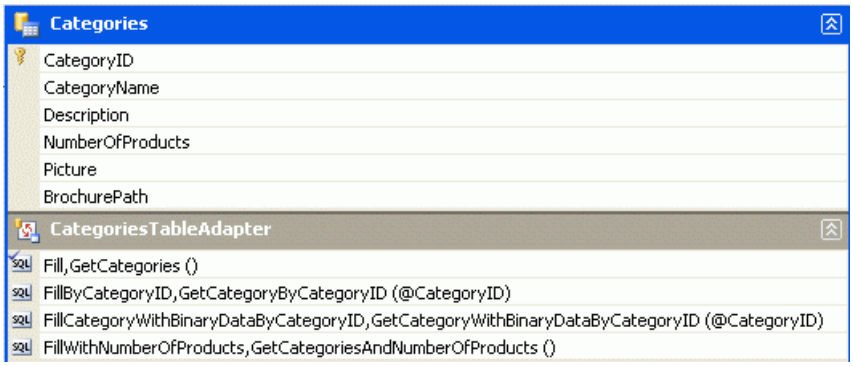


Figure 11: The DataSet Designer Includes the New Columns and Method

Updating the Business Logic Layer (BLL)

With the DAL updated, all that remains is to augment the Business Logic Layer (BLL) to include a method for the new `CategoriesTableAdapter` method. Add the following method to the `CategoriesBLL` class:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetCategoryWithBinaryDataByCategoryID(categoryID As Integer) _
    As Northwind.CategoriesDataTable

    Return Adapter.GetCategoryWithBinaryDataByCategoryID(categoryID)
End Function
```

Step 5: Uploading a File From the Client to the Web Server

When collecting binary data, oftentimes this data is supplied by an end user. To capture this information, the user needs to be able to upload a file from their computer to the web server. The uploaded data then needs to be integrated with the data model, which may mean saving the file to the web server s file system and adding a path to the file in the database, or writing the binary contents directly into the database. In this step we'll look at how to allow a user to upload files from their computer to the server. In the next tutorial we'll turn our attention to integrating the uploaded file with data model.

ASP.NET 2.0 s new FileUpload Web control provides a mechanism for users to send a file from their computer to the web server. The FileUpload control renders as an `<input>` element whose `type` attribute is set to file, which browsers display as a textbox with a Browse button. Clicking the Browse button brings up a dialog box from which the user can select a file. When the form is posted back, the selected file s contents are sent along with the postback. On the server-side, information about the uploaded file is accessible through the FileUpload control s properties.

To demonstrate uploading files, open the `FileUpload.aspx` page in the `BinaryData` folder, drag a FileUpload control from the Toolbox onto the Designer, and set the control s `ID` property to `UploadTest`. Next, add a Button Web control setting its `ID` and `Text` properties to `UploadButton` and Upload Selected File, respectively. Finally, place a Label Web control beneath the Button, clear out its `Text` property and set its `ID` property to `UploadDetails`.

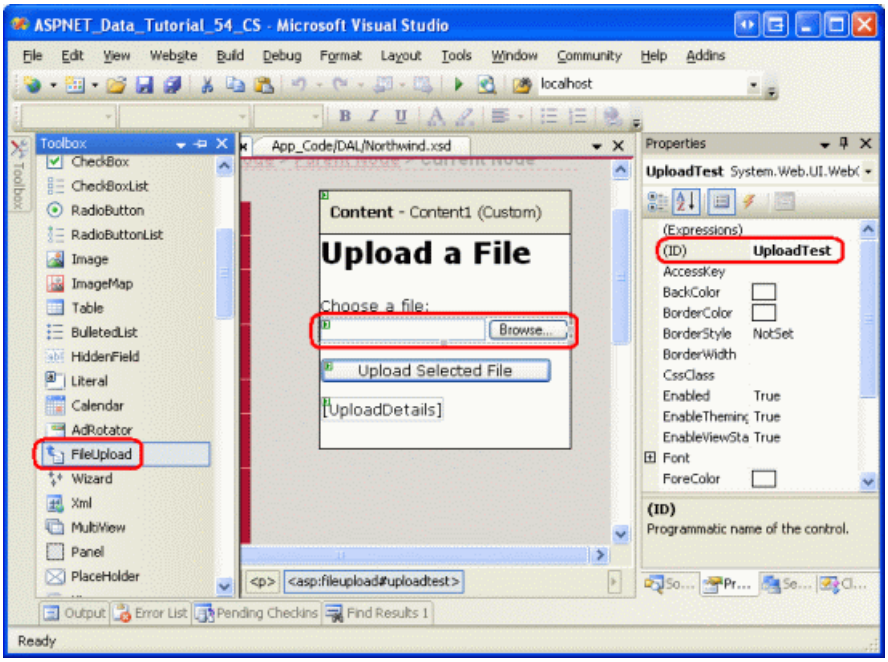


Figure 12: Add a FileUpload Control to the ASP.NET Page (Click to view full-size image)

Figure 13 shows this page when viewed through a browser. Note that clicking the Browse button brings up a file selection dialog box, allowing the user to pick a file from their computer. Once a file has been selected, clicking the Upload Selected File button causes a postback that sends the selected file s binary content to the web server.

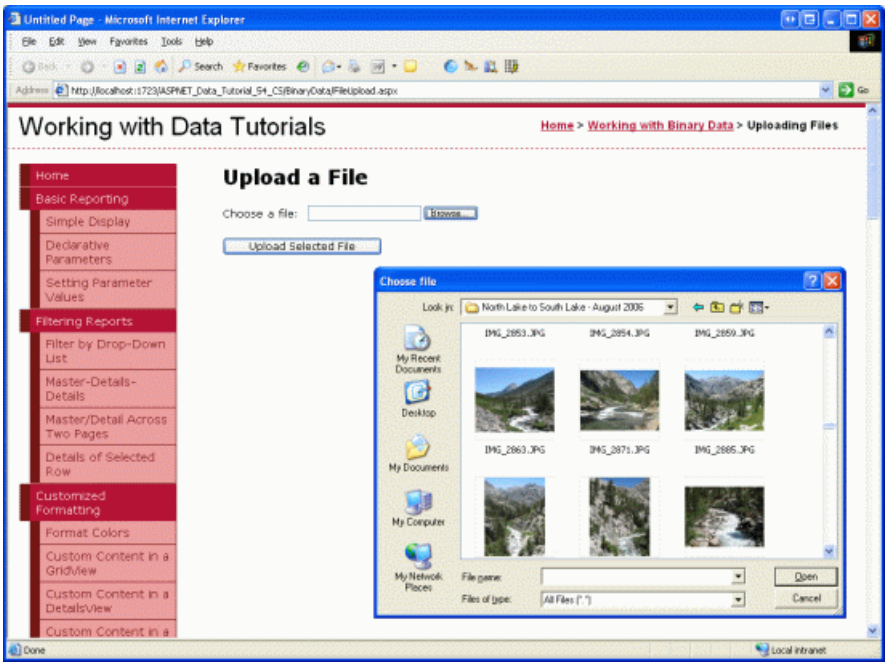


Figure 13: The User Can Select a File to Upload from their Computer to the Server (Click to view full-size image)

On postback, the uploaded file can be saved to the file system or its binary data can be worked with directly through a Stream. For this example, let s create a `~/Brochures` folder and save the uploaded file there. Start by adding the `Brochures` folder to the site as a subfolder of the root directory. Next, create an event handler for the `UploadButton` s `Click` event and add the following code:

```
Protected Sub UploadButton_Click(sender As Object, e As EventArgs) _
    Handles UploadButton.Click

    If UploadTest.HasFile = False Then
        ' No file uploaded!
        UploadDetails.Text = "Please first select a file to upload..."
    Else
        ' Display the uploaded file's details
        UploadDetails.Text = String.Format( _
                "Uploaded file: {0}<br />" & _
                "File size (in bytes): {1:N0}<br />" & _
                "Content-type: {2}", _
                UploadTest.FileName, _
                UploadTest.FileBytes.Length, _
                UploadTest.PostedFile.ContentType)
        ' Save the file
        Dim filePath As String = _
            Server.MapPath("~/Brochures/" & UploadTest.FileName)
        UploadTest.SaveAs(filePath)
    End If
End Sub
```

The FileUpload control provides a variety of properties for working with the uploaded data. For instance, the `HasFile` property indicates whether a file was uploaded by the user, while the `FileBytes` property provides access to the uploaded binary data as an array of bytes. The `Click` event handler starts by ensuring that a file has been uploaded. If a file has been uploaded, the Label shows the name of the uploaded file, its size in bytes, and its content-type.

> **NOTE**
>
> To ensure that the user uploads a file you can check the `HasFile` property and display a warning if it s `False`, or you may use the RequiredFieldValidator control instead.

The FileUpload s `SaveAs(filePath)` saves the uploaded file to the specified *filePath*. *filePath* must be a *physical path* (`C:\Websites\Brochures\SomeFile.pdf`) rather than a *virtual path* (`/Brochures/SomeFile.pdf`). The `Server.MapPath(virtPath)` method takes a virtual path and returns its corresponding physical path. Here, the virtual path is `~/Brochures/fileName` , where *fileName* is the name of the uploaded file. See Using Server.MapPath for more information on virtual and physical paths and using `Server.MapPath` .

After completing the `Click` event handler, take a moment to test out the page in a browser. Click the Browse button and select a file from your hard drive and then click the Upload Selected File button. The postback will send the contents of the selected file to the web server, which will then display information about the file before saving it to the `~/Brochures` folder. After uploading the file, return to Visual Studio and click the Refresh button in the Solution Explorer. You should see the file you just uploaded in the ~/Brochures folder!

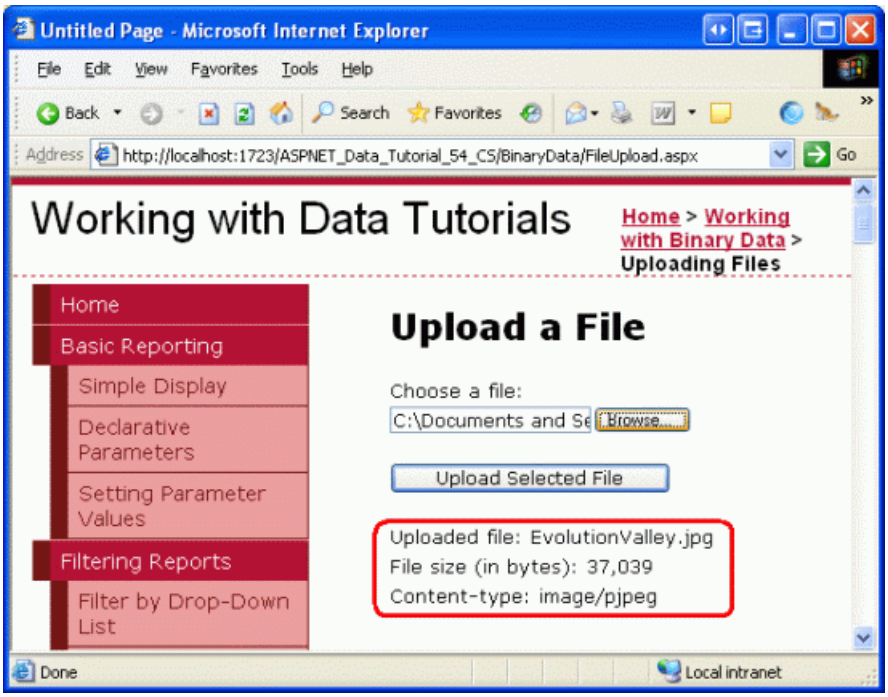


Figure 14: The File `EvolutionValley.jpg` Has Been Uploaded to the Web Server (Click to view full-size image)

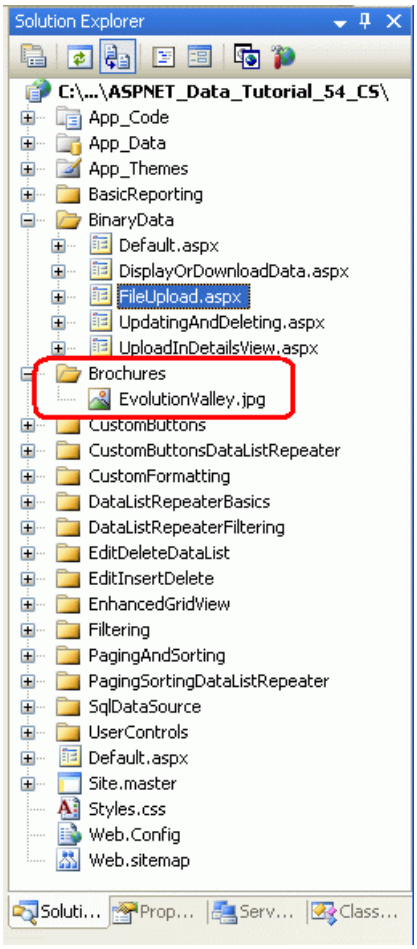


Figure 15: `EvolutionValley.jpg` Was Saved to the `~/Brochures` Folder

Subtleties with Saving Uploaded Files to the File System

There are several subtleties that must be addressed when saving uploading files to the web server s file system. First, there s the issue of security. To save a file to the file system, the security context under which the ASP.NET page is executing must have Write permissions. The ASP.NET Development Web Server runs under the context of your current user account. If you are using Microsoft s Internet Information Services (IIS) as the web server, the security context depends on the version of IIS and its configuration.

Another challenge of saving files to the file system revolves around naming the files. Currently, our page saves all of the uploaded files to the `~/Brochures` directory using the same name as the file on the client s computer. If User A uploads a brochure with the name `Brochure.pdf`, the file will be saved as `~/Brochure/Brochure.pdf`. But what if sometime later User B uploads a different brochure file that happens to have the same filename (`Brochure.pdf`)? With the code we have now, User A s file will be overwritten with User B s upload.

There are a number of techniques for resolving file name conflicts. One option is to prohibit uploading a file if there already exists one with the same name. With this approach, when User B attempts to upload a file named `Brochure.pdf`, the system would not save their file and instead display a message informing User B to rename the file and try again. Another approach is to save the file using a unique file name, which could be a globally unique identifier (GUID) or the value from the corresponding database record s primary key column(s) (assuming that the upload is associated with a particular row in the data model). In the next tutorial we'll explore these options in more detail.

Challenges Involved with Very Large Amounts of Binary Data

These tutorials assume that the binary data captured is modest in size. Working with very large amounts of binary data files that are several megabytes or larger introduces new challenges that are beyond the scope of these tutorials. For example, by default ASP.NET will reject uploads of more than 4 MB, although this can be

configured through the `<httpRuntime>` element in `Web.config`. IIS imposes its own file upload size limitations, too. See IIS Upload File Size for more information. Furthermore, the time taken to upload large files might exceed the default 110 seconds ASP.NET will wait for a request. There are also memory and performance issues that arise when working with large files.

The FileUpload control is impractical for large file uploads. As the file s contents are being posted to the server, the end user must patiently wait without any confirmation that their upload is progressing. This is not so much an issue when dealing with smaller files that can be uploaded in a few seconds, but can be an issue when dealing with larger files that may take minutes to upload. There are a variety of third-party file upload controls that are better suited for handling large uploads and many of these vendors provide progress indicators and ActiveX upload managers that present a much more polished user experience.

If your application needs to handle large files, you'll need to carefully investigate the challenges and find suitable solutions for your particular needs.

Summary

Building an application that needs to capture binary data introduces a number of challenges. In this tutorial we explored the first two: deciding where to store the binary data and allowing a user to upload binary content through a web page. Over the next three tutorials, we'll see how to associate the uploaded data with a record in the database as well as how to display the binary data alongside its text data fields.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Using Large-Value Data Types
- FileUpload Control QuickStarts
- The ASP.NET 2.0 FileUpload Server Control
- The Dark Side of File Uploads

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Teresa Murphy and Bernadette Leigh. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Displaying Binary Data in the Data Web Controls (VB)

2/20/2020 • 15 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In this tutorial we look at the options to present binary data on a Web page, including the display of an image file and the provision of a 'Download' link for a PDF file.

Introduction

In the preceding tutorial we explored the two techniques for associating binary data with an application s underlying data model and used the FileUpload control to upload files from a browser to the web server s file system. We ve yet to see how to associate the uploaded binary data with the data model. That is, after a file has been uploaded and saved to the file system, a path to the file must be stored in the appropriate database record. If the data is being stored directly in the database, then the uploaded binary data need not be saved to the file system, but must be injected into the database.

Before we look at associating the data with the data model, though, let s first look at how to provide the binary data to the end user. Presenting text data is simple enough, but how should binary data be presented? It depends, of course, on the type of binary data. For images, we likely want to display the image; for PDFs, Microsoft Word documents, ZIP files, and other types of binary data, providing a Download link is probably more appropriate.

In this tutorial we will look at how to present the binary data alongside its associated text data using data Web controls like the GridView and DetailsView. In the next tutorial we'll turn our attention to associating an uploaded file with the database.

Step 1: Providing `BrochurePath` Values

The `Picture` column in the `Categories` table already contains binary data for the various category images. Specifically, the `Picture` column for each record holds the binary contents of a grainy, low-quality, 16-color bitmap image. Each category image is 172 pixels wide and 120 pixels tall and consumes roughly 11 KB. What s more, the binary content in the `Picture` column includes a 78-byte OLE header that must be stripped before displaying the image. This header information is present because the Northwind database has its roots in Microsoft Access. In Access, binary data is stored using the OLE Object data type, which tacks on this header. For now, we'll see how to strip the headers from these low-quality images in order to display the picture. In a future tutorial we'll build an interface for updating a category s `Picture` column and replace these bitmap images that use OLE headers with equivalent JPG images without the unnecessary OLE headers.

In the preceding tutorial we saw how to use the FileUpload control. Therefore, you can go ahead and add brochure files to the web server s file system. Doing so, however, does not update the `BrochurePath` column in the `Categories` table. In the next tutorial we'll see how to accomplish this, but for now we need to manually provide values for this column.

In this tutorial s download you'll find seven PDF brochure files in the `~/Brochures` folder, one for each of the categories except Seafood. I purposefully omitted adding a Seafood brochure to illustrate how to handle scenarios where not all records have associated binary data. To update the `Categories` table with these values, right-click on

the `Categories` node from Server Explorer and choose Show Table Data. Then, enter the virtual paths to the brochure files for each category that has a brochure, as Figure 1 shows. Since there is no brochure for the Seafood category, leave its `BrochurePath` column s value as `NULL`.

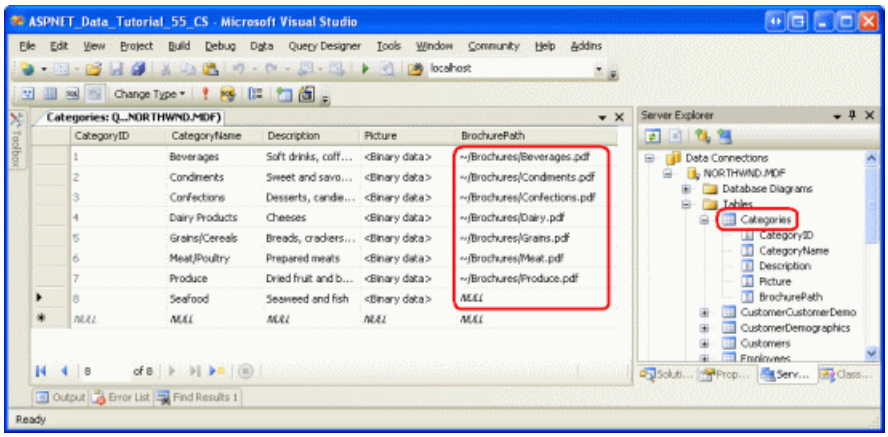


Figure 1: Manually Enter the Values for the `Categories` Table s `BrochurePath` Column (Click to view full-size image)

Step 2: Providing a Download Link for the Brochures in a GridView

With the `BrochurePath` values provided for the `Categories` table, we re ready to create a GridView that lists each category along with a link to download the category s brochure. In Step 4 we'll extend this GridView to also display the category s image.

Start by dragging a GridView from the Toolbox onto the Designer of the `DisplayOrDownloadData.aspx` page in the `BinaryData` folder. Set the GridView s `ID` to `Categories` and through the GridView s smart tag, choose to bind it to a new data source. Specifically, bind it to an ObjectDataSource named `CategoriesDataSource` that retrieves data using the `CategoriesBLL` object s `GetCategories()` method.

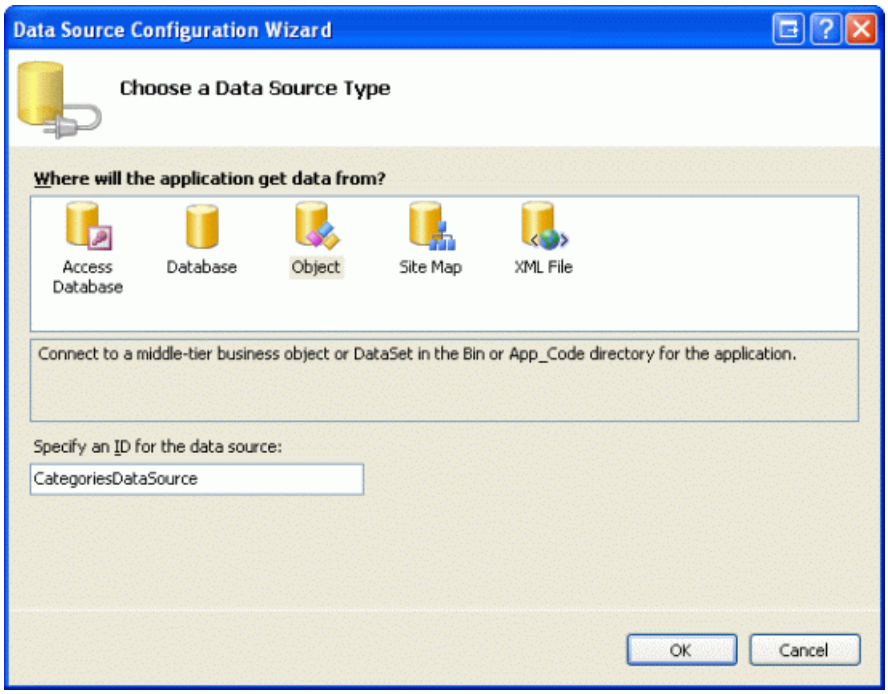


Figure 2: Create a New ObjectDataSource Named `CategoriesDataSource` (Click to view full-size image)

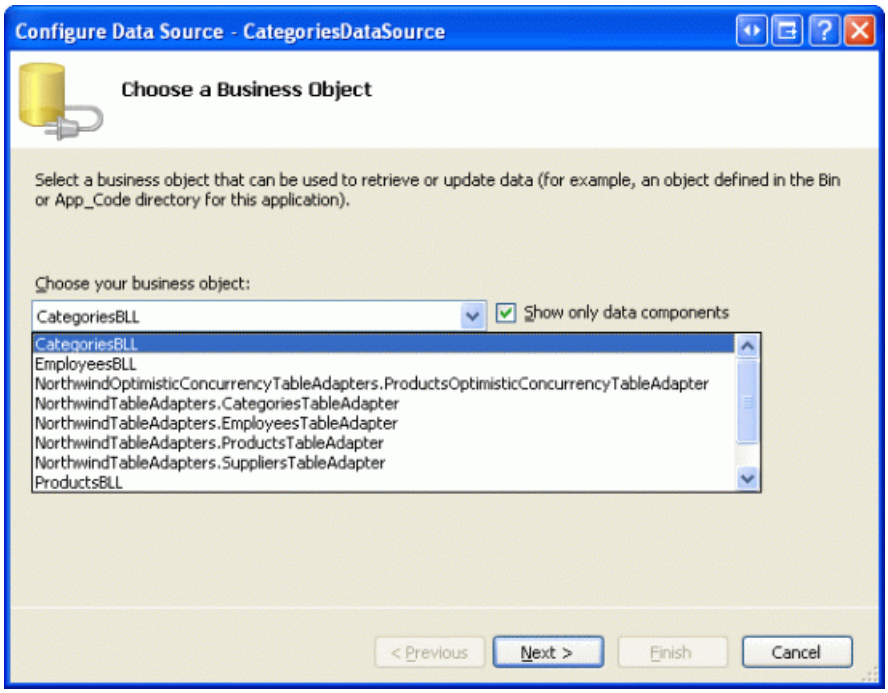


Figure 3: Configure the ObjectDataSource to Use the `CategoriesBLL` Class (Click to view full-size image)

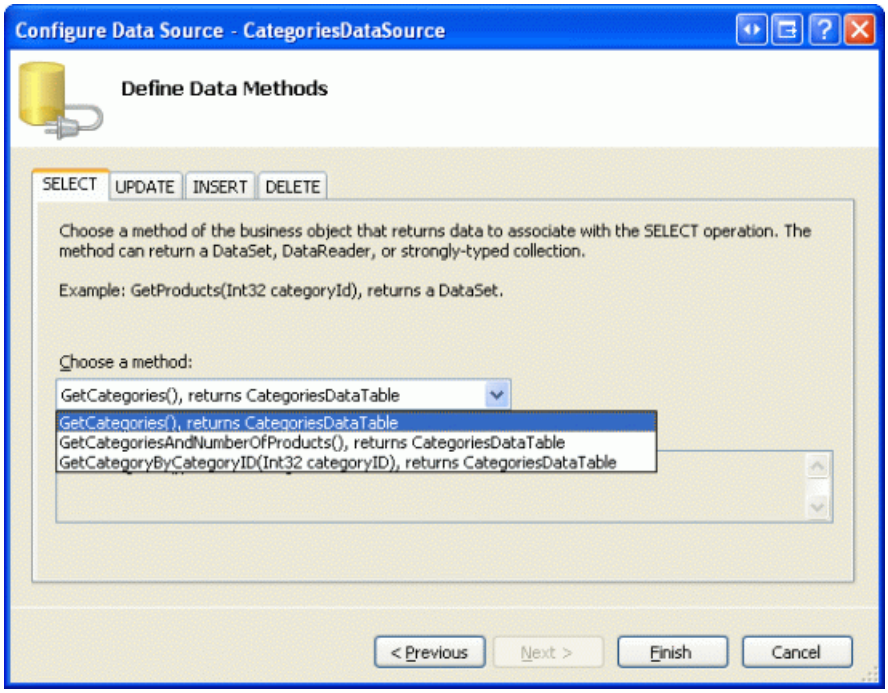


Figure 4: Retrieve the List of Categories Using the `GetCategories()` Method (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will automatically add a BoundField to the `Categories` GridView for the `CategoryID`, `CategoryName`, `Description`, `NumberOfProducts`, and `BrochurePath` `DataColumn` s. Go ahead and remove the `NumberOfProducts` BoundField since the `GetCategories()` method s query does not retrieve this information. Also remove the `CategoryID` BoundField and rename the `CategoryName` and `BrochurePath` BoundFields `HeaderText` properties to Category and Brochure, respectively. After making these changes, your GridView and ObjectDataSource s declarative markup should look like the following:

```
<asp:GridView ID="Categories" runat="server"
    AutoGenerateColumns="False" DataKeyNames="CategoryID"
    DataSourceID="CategoriesDataSource" EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            SortExpression="CategoryName" />
        <asp:BoundField DataField="Description" HeaderText="Description"
            SortExpression="Description" />
        <asp:BoundField DataField="BrochurePath" HeaderText="Brochure"
            SortExpression="BrochurePath" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

View this page through a browser (see Figure 5). Each of the eight categories is listed. The seven categories with `BrochurePath` values have the `BrochurePath` value displayed in the respective BoundField. Seafood, which has a `NULL` value for its `BrochurePath`, displays an empty cell.

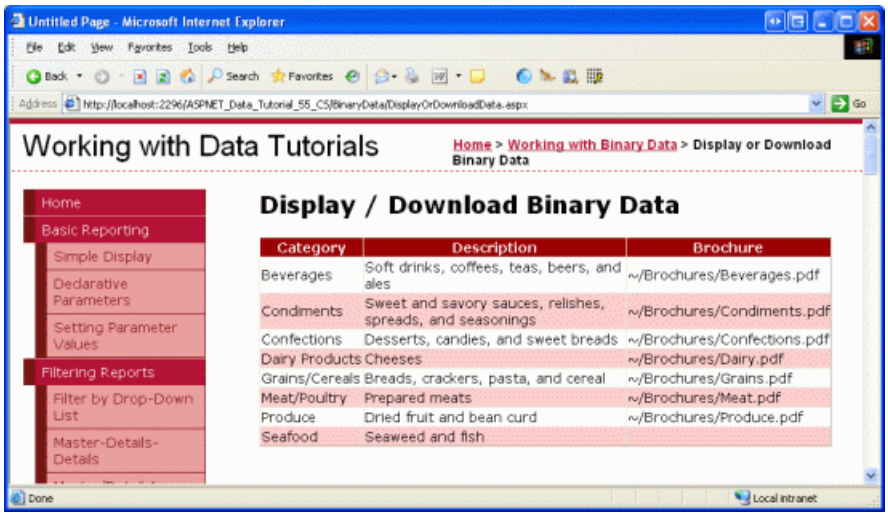


Figure 5: Each Category s Name, Description, and `BrochurePath` Value is Listed (Click to view full-size image)

Rather than displaying the text of the `BrochurePath` column, we want to create a link to the brochure. To accomplish this, remove the `BrochurePath` BoundField and replace it with a HyperLinkField. Set the new HyperLinkField s `HeaderText` property to Brochure, its `Text` property to View Brochure, and its `DataNavigateUrlFields` property to `BrochurePath`.

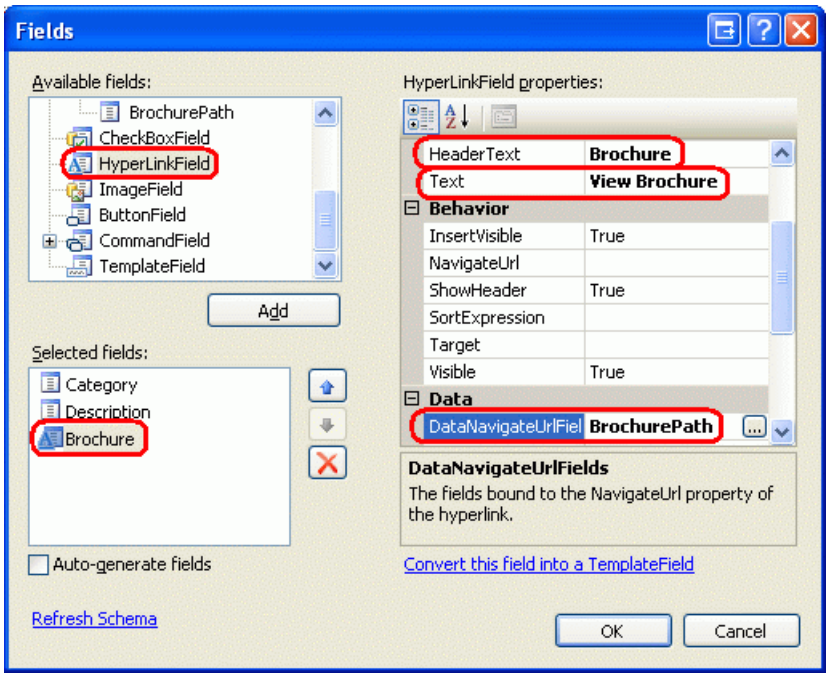


Figure 6: Add a HyperLinkField for `BrochurePath`

This will add a column of links to the GridView, as Figure 7 shows. Clicking a View Brochure link will either display the PDF directly in the browser or prompt the user to download the file, depending on whether a PDF reader is installed and the browser s settings.

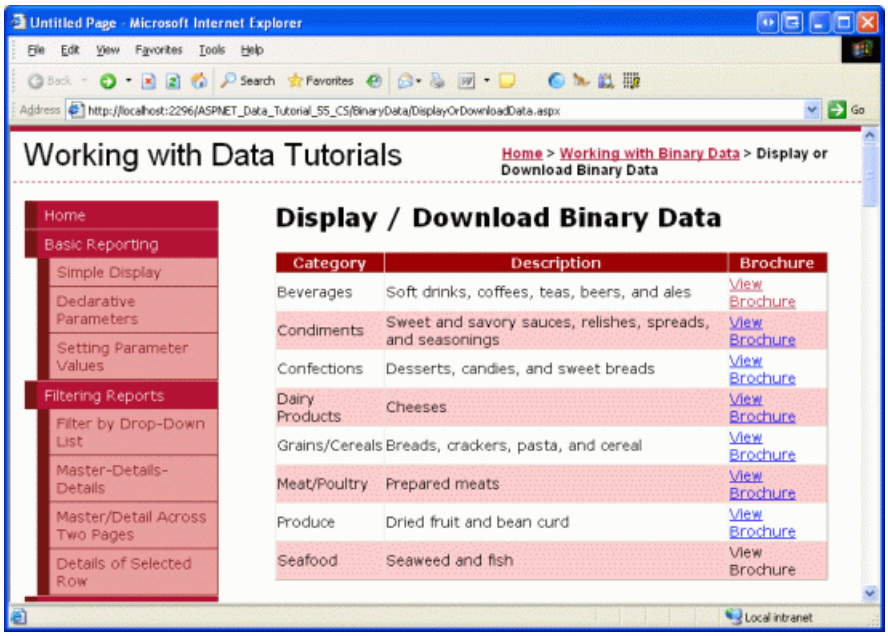


Figure 7: A Category s Brochure Can Be Viewed by Clicking the View Brochure Link (Click to view full-size image)

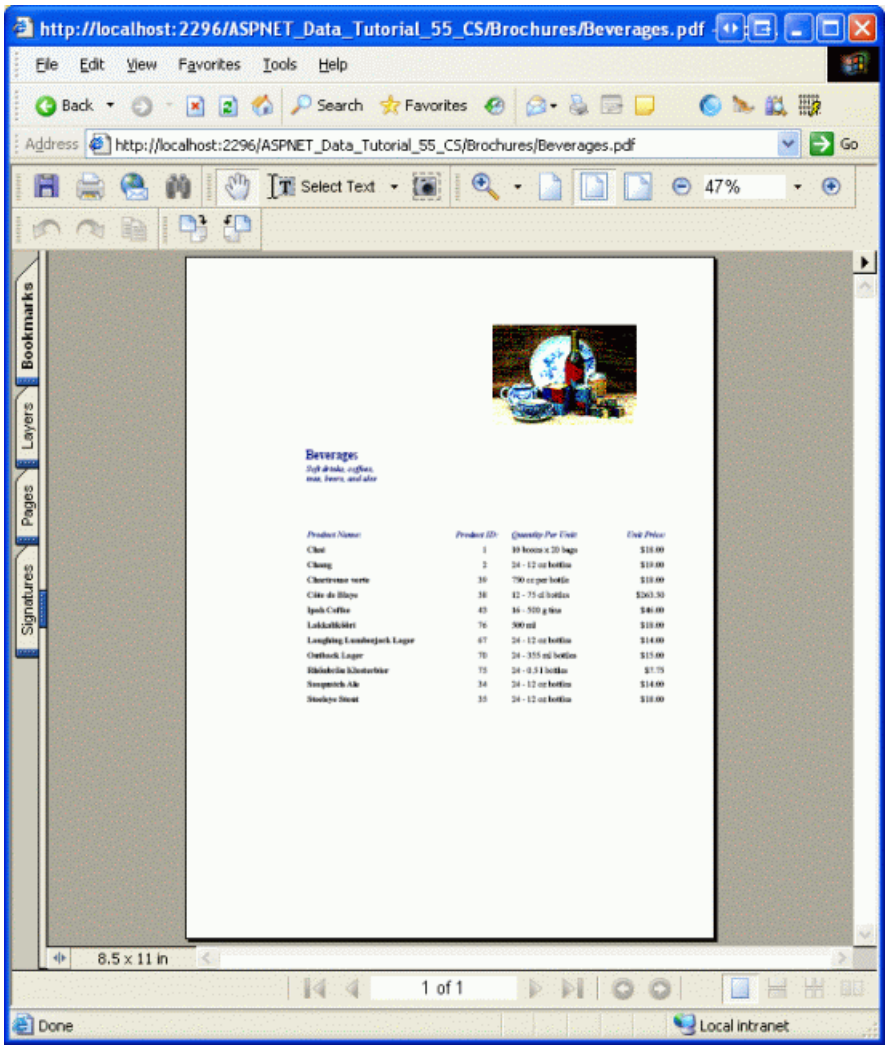


Figure 8: The Category s Brochure PDF is Displayed (Click to view full-size image)

Hiding the View Brochure Text for Categories Without a Brochure

As Figure 7 shows, the `BrochurePath` HyperLinkField displays its `Text` property value (View Brochure) for all records, regardless of whether there s a non- `NULL` value for `BrochurePath`. Of course, if `BrochurePath` is `NULL`, then the link is displayed as text only, as is the case with the Seafood category (refer back to Figure 7). Rather than displaying the text View Brochure, it might be nice to have those categories without a `BrochurePath` value display some alternate text, like No Brochure Available.

In order to provide this behavior, we need to use a TemplateField whose content is generated via a call to a page method that emits the appropriate output based on the `BrochurePath` value. We first explored this formatting technique back in the Using TemplateFields in the GridView Control tutorial.

Turn the HyperLinkField into a TemplateField by selecting the `BrochurePath` HyperLinkField and then clicking on the Convert this field into a TemplateField link in the Edit Columns dialog box.

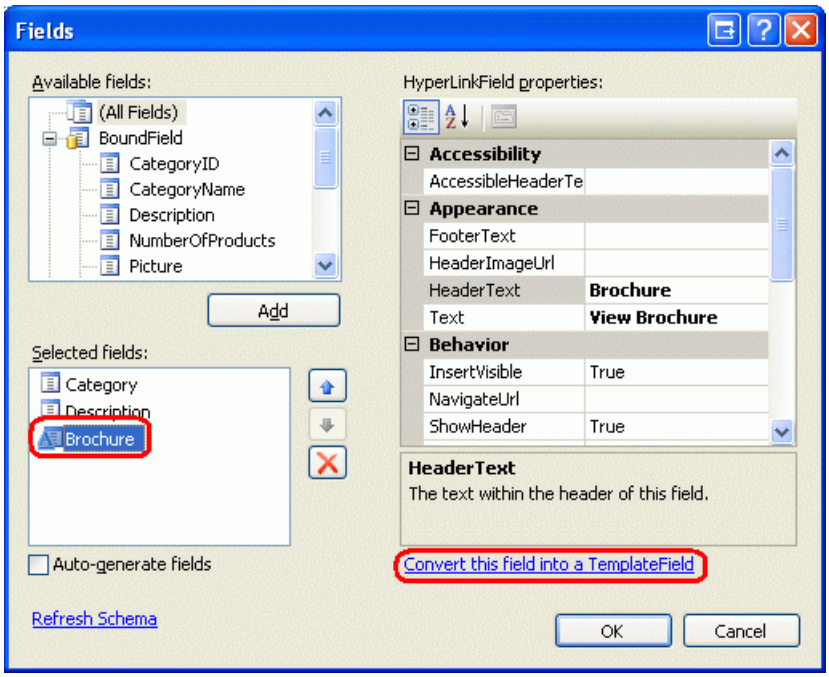


Figure 9: Convert the HyperLinkField into a TemplateField

This will create a TemplateField with an `ItemTemplate` that contains a HyperLink Web control whose `NavigateUrl` property is bound to the `BrochurePath` value. Replace this markup with a call to the method `GenerateBrochureLink`, passing in the value of `BrochurePath`:

```
<asp:TemplateField HeaderText="Brochure">
    <ItemTemplate>
        <%# GenerateBrochureLink(Eval("BrochurePath")) %>
    </ItemTemplate>
</asp:TemplateField>
```

Next, create a `Protected` method in the ASP.NET page s code-behind class named `GenerateBrochureLink` that returns a `String` and accepts an `Object` as an input parameter.

```
Protected Function GenerateBrochureLink(BrochurePath As Object) As String
    If Convert.IsDBNull(BrochurePath) Then
        Return "No Brochure Available"
    Else
        Return String.Format("<a href="{0}">View Brochure</a>", _
            ResolveUrl(BrochurePath.ToString()))
    End If
End Function
```

This method determines if the passed-in `Object` value is a database `NULL` and, if so, returns a message indicating that the category lacks a brochure. Otherwise, if there is a `BrochurePath` value, it s displayed in a hyperlink. Note that if the `BrochurePath` value is present it s passed into the `ResolveUrl(url)` method. This method resolves the passed-in *url*, replacing the `~` character with the appropriate virtual path. For example, if the application is rooted at `/Tutorial55`, `ResolveUrl("~/Brochures/Meats.pdf")` will return `/Tutorial55/Brochures/Meat.pdf`.

Figure 10 shows the page after these changes have been applied. Note that the Seafood category s `BrochurePath` field now displays the text No Brochure Available .

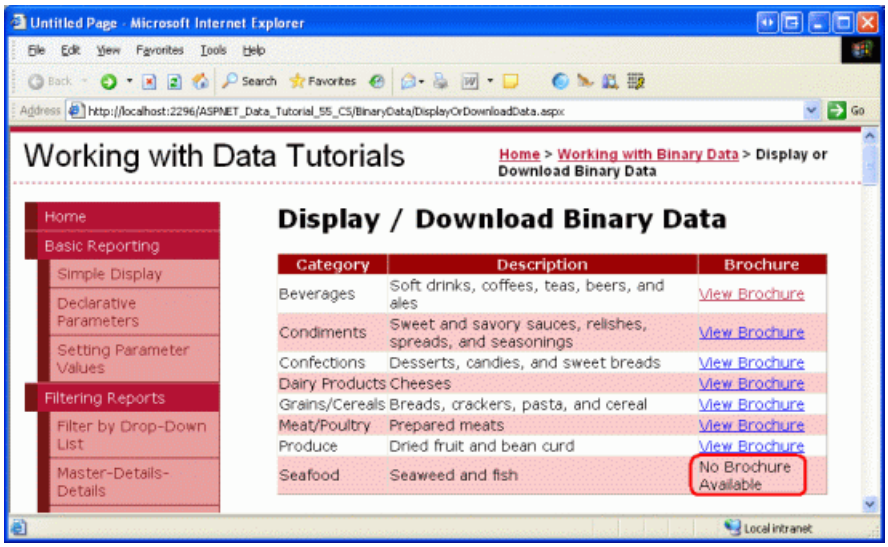


Figure 10: The Text No Brochure Available is Displayed for Those Categories Without a Brochure (Click to view full-size image)

Step 3: Adding a Web Page to Display a Category s Picture

When a user visits an ASP.NET page, they receive the ASP.NET page s HTML. The received HTML is just text and does not contain any binary data. Any additional binary data, such as images, sound files, Macromedia Flash applications, embedded Windows Media Player videos, and so forth, exist as separate resources on the web server. The HTML contains references to these files, but does not include the actual contents of the files.

For example, in HTML the `<img>` element is used to reference a picture, with the `src` attribute pointing to the image file like so:

```
<img src="MyPicture.jpg" ... />
```

When a browser receives this HTML, it makes another request to the web server to retrieve the binary contents of the image file, which it then displays in the browser. The same concept applies to any binary data. In Step 2, the brochure was not sent down to the browser as part of the page s HTML markup. Rather, the rendered HTML provided hyperlinks that, when clicked, caused the browser to request the PDF document directly.

To display or allow users to download binary data that resides within the database, we need to create a separate web page that returns the data. For our application, there s only one binary data field stored directly in the database the category s picture. Therefore, we need a page that, when called, returns the image data for a particular category.

Add a new ASP.NET page to the `BinaryData` folder named `DisplayCategoryPicture.aspx`. When doing so, leave the Select master page checkbox unchecked. This page expects a `CategoryID` value in the querystring and returns the binary data of that category s `Picture` column. Since this page returns binary data and nothing else, it does not need any markup in the HTML section. Therefore, click on the Source tab in the lower left corner and remove all of the page s markup except for the `<%@ Page %>` directive. That is, `DisplayCategoryPicture.aspx` s declarative markup should consist of a single line:

```
<%@ Page Language="VB" AutoEventWireup="true"
    CodeFile="DisplayCategoryPicture.aspx.vb"
    Inherits="BinaryData_DisplayCategoryPicture" %>
```

If you see the `MasterPageFile` attribute in the `<%@ Page %>` directive, remove it.

In the page s code-behind class, add the following code to the `Page_Load` event handler:

```
Protected Sub Page_Load(sender As Object, e As EventArgs) Handles Me.Load
    Dim categoryID As Integer = _
        Convert.ToInt32(Request.QueryString("CategoryID"))
    ' Get information about the specified category
    Dim categoryAPI As New CategoriesBLL()
    Dim categories As Northwind.CategoriesDataTable = _
        categoryAPI.GetCategoryWithBinaryDataByCategoryID(categoryID)
    Dim category As Northwind.CategoriesRow = categories(0)
    ' Output HTTP headers providing information about the binary data
    Response.ContentType = "image/bmp"
    ' Output the binary data
    ' But first we need to strip out the OLE header
    Const OleHeaderLength As Integer = 78
    Dim strippedImageLength As Integer = _
        category.Picture.Length - OleHeaderLength
    Dim strippedImageData(strippedImageLength) As Byte
    Array.Copy(category.Picture, OleHeaderLength, _
        strippedImageData, 0, strippedImageLength)
    Response.BinaryWrite(strippedImageData)
End Sub
```

This code starts by reading in the `CategoryID` querystring value into a variable named `categoryID`. Next, the picture data is retrieved via a call to the `CategoriesBLL` class s `GetCategoryWithBinaryDataByCategoryID(categoryID)` method. This data is returned to the client by using the `Response.BinaryWrite(data)` method, but before this is called, the `Picture` column value s OLE header must be removed. This is accomplished by creating a `Byte` array named `strippedImageData` that will hold precisely 78 characters less than what is in the `Picture` column. The `Array.Copy` method is used to copy the data from `category.Picture` starting at position 78 over to `strippedImageData`.

The `Response.ContentType` property specifies the MIME type of the content being returned so that the browser knows how to render it. Since the `Categories` table s `Picture` column is a bitmap image, the bitmap MIME type is used here (image/bmp). If you omit the MIME type, most browsers will still display the image correctly because they can infer the type based on the contents of the image file s binary data. However, it s prudent to include the MIME type when possible. See the Internet Assigned Numbers Authority s website for a complete listing of MIME media types.

With this page created, a particular category s picture can be viewed by visiting `DisplayCategoryPicture.aspx?CategoryID=categoryID`. Figure 11 shows the Beverages category s picture, which can be viewed from `DisplayCategoryPicture.aspx?CategoryID=1`.

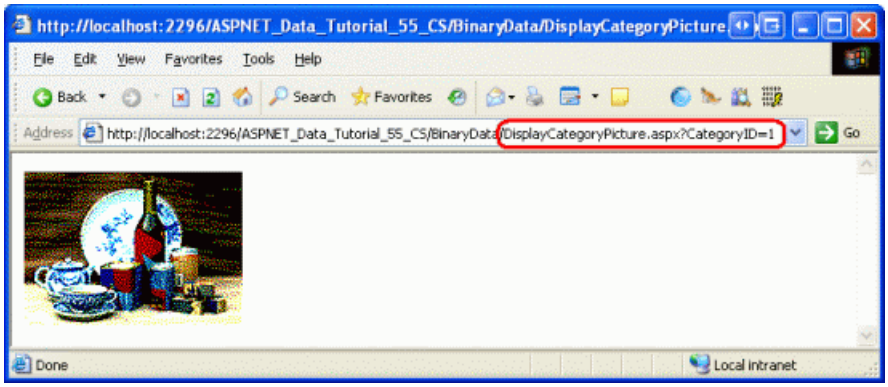


Figure 11: The Beverages Category s Picture is Displayed (Click to view full-size image)

If, when visiting `DisplayCategoryPicture.aspx?CategoryID=categoryID`, you get an exception that reads Unable to cast object of type 'System.DBNull' to type 'System.Byte[]', there are two things that may be causing this. First, the `Categories` table s `Picture` column does allow `NULL` values. The `DisplayCategoryPicture.aspx` page, however, assumes there is a non-`NULL` value present. The `Picture` property of the `CategoriesDataTable` cannot be directly

accessed if it has a `NULL` value. If you do want to allow `NULL` values for the `Picture` column, you d want to include the following condition:

```
If category.IsPictureNull() Then
    ' Display some "No Image Available" picture
    Response.Redirect("~/Images/NoPictureAvailable.gif")
Else
    ' Send back the binary contents of the Picture column
    ' ... Set ContentType property and write out ...
    ' ... data via Response.BinaryWrite ...
End If
```

The above code assumes that there s some image file named `NoPictureAvailable.gif` in the `Images` folder that you want to display for those categories without a picture.

This exception could also be caused if the `CategoriesTableAdapter` s `GetCategoryWithBinaryDataByCategoryID` method s `SELECT` statement has reverted back to the main query s column list, which can happen if you are using ad-hoc SQL statements and you ve re-run the wizard for the TableAdapter s main query. Check to ensure that `GetCategoryWithBinaryDataByCategoryID` method s `SELECT` statement still includes the `Picture` column.

> **NOTE**
>
> Every time the `DisplayCategoryPicture.aspx` is visited, the database is accessed and the specified category s picture data is returned. If the category s picture hasn't changed since the user has last viewed it, though, this is wasted effort. Fortunately, HTTP allows for *conditional GETs*. With a conditional GET, the client making the HTTP request sends along an `If-Modified-Since` HTTP header that provides the date and time the client last retrieved this resource from the web server. If the content has not changed since this specified date, the web server may respond with a Not Modified status code (304) and forgo sending back the requested resource s content. In short, this technique relieves the web server from having to send back content for a resource if it has not been modified since the client last accessed it.

To implement this behavior, however, requires that you add a `PictureLastModified` column to the `Categories` table to capture when the `Picture` column was last updated as well as code to check for the `If-Modified-Since` header. For more information on the `If-Modified-Since` header and the conditional GET workflow, see HTTP Conditional GET for RSS Hackers and A Deeper Look at Performing HTTP Requests in an ASP.NET Page.

Step 4: Displaying the Category Pictures in a GridView

Now that we have a web page to display a particular category s picture, we can display it using the Image Web control or an HTML `<img>` element pointing to `DisplayCategoryPicture.aspx?CategoryID=categoryID`. Images whose URL is determined by database data can be displayed in the GridView or DetailsView using the ImageField. The ImageField contains `DataImageUrlField` and `DataImageUrlFormatString` properties that work like the HyperLinkField s `DataNavigateUrlFields` and `DataNavigateUrlFormatString` properties.

Let s augment the `Categories` GridView in `DisplayOrDownloadData.aspx` by adding an ImageField to show each category s picture. Simply add the ImageField and set its `DataImageUrlField` and `DataImageUrlFormatString` properties to `CategoryID` and `DisplayCategoryPicture.aspx?CategoryID={0}`, respectively. This will create a GridView column that renders an `<img>` element whose `src` attribute references `DisplayCategoryPicture.aspx?CategoryID={0}`, where {0} is replaced with the GridView row s `CategoryID` value.

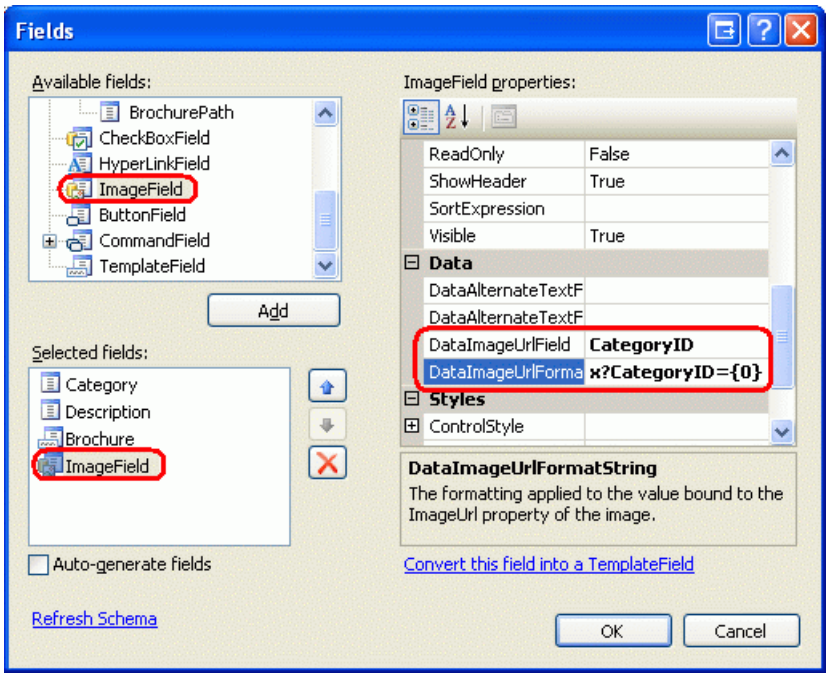


Figure 12: Add an ImageField to the GridView

After adding the ImageField, your GridView s declarative syntax should look like soothe following:

```
<asp:GridView ID="Categories" runat="server" AutoGenerateColumns="False"
    DataKeyNames="CategoryID" DataSourceID="CategoriesDataSource"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            SortExpression="CategoryName" />
        <asp:BoundField DataField="Description" HeaderText="Description"
            SortExpression="Description" />
        <asp:TemplateField HeaderText="Brochure">
            <ItemTemplate>
                <%# GenerateBrochureLink(Eval("BrochurePath")) %>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:ImageField DataImageUrlField="CategoryID"
            DataImageUrlFormatString="DisplayCategoryPicture.aspx?CategoryID={0}">
        </asp:ImageField>
    </Columns>
</asp:GridView>
```

Take a moment to view this page through a browser. Note how each record now includes a picture for the category.

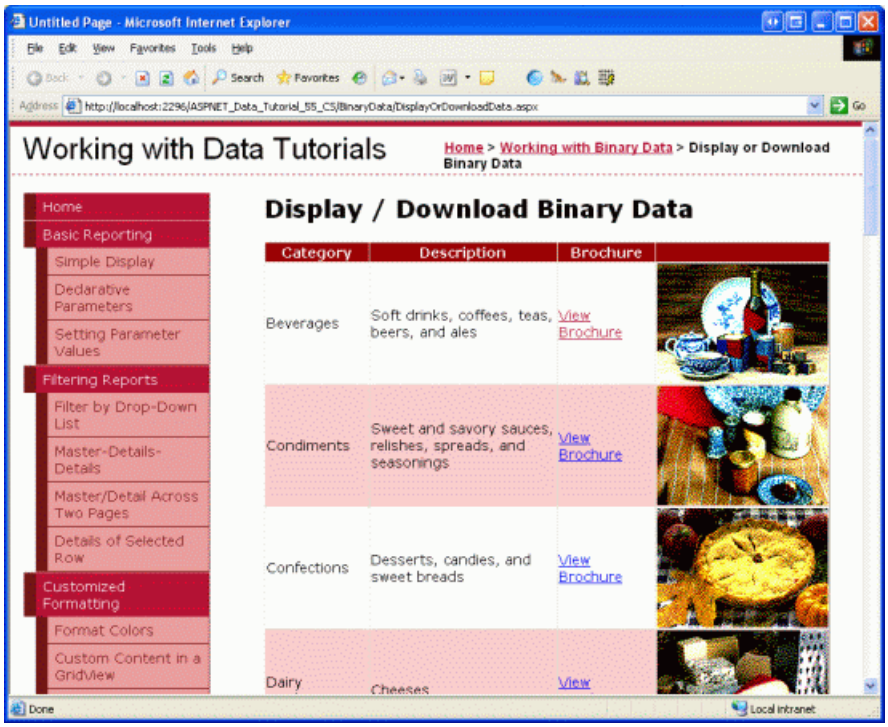


Figure 13: The Category s Picture is Displayed for Each Row (Click to view full-size image)

Summary

In this tutorial we examined how to present binary data. How the data is presented depends on the type of data. For the PDF brochure files, we offered the user a View Brochure link that, when clicked, took the user directly to the PDF file. For the category s picture, we first created a page to retrieve and return the binary data from the database and then used that page to display each category s picture in a GridView.

Now that we ve looked at how to display binary data, we re ready to examine how to perform insert, updates, and deletes against the database with the binary data. In the next tutorial we'll look at how to associate an uploaded file with its corresponding database record. In the tutorial after that, we'll see how to update existing binary data as well as how to delete the binary data when its associated record is removed.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Teresa Murphy and Dave Gardner. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Including a File Upload Option When Adding a New Record (VB)

2/20/2020 • 23 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> This tutorial shows how to create a Web interface that allows the user to both enter text data and upload binary files. To illustrate the options available to store binary data, one file will be saved in the database while the other is stored in the file system.

Introduction

In the previous two tutorials we explored techniques for storing binary data that is associated with the application s data model, looked at how to use the FileUpload control to send files from the client to the web server, and saw how to present this binary data in a data Web control. We ve yet to talk about how to associate uploaded data with the data model, though.

In this tutorial we will create a web page to add a new category. In addition to TextBoxes for the category s name and description, this page will need to include two FileUpload controls one for the new category s picture and one for the brochure. The uploaded picture will be stored directly in the new record s `Picture` column, whereas the brochure will be saved to the `~/Brochures` folder with the path to the file saved in the new record s `BrochurePath` column.

Before creating this new web page, we'll need to update the architecture. The `CategoriesTableAdapter` s main query does not retrieve the `Picture` column. Consequently, the auto-generated `Insert` method only has inputs for the `CategoryName`, `Description`, and `BrochurePath` fields. Therefore, we need to create an additional method in the TableAdapter that prompts for all four `Categories` fields. The `CategoriesBLL` class in the Business Logic Layer will also need to be updated.

Step 1: Adding an `InsertWithPicture` Method to the `CategoriesTableAdapter`

When we created the `CategoriesTableAdapter` back in the Creating a Data Access Layer tutorial, we configured it to automatically generate `INSERT`, `UPDATE`, and `DELETE` statements based on the main query. Moreover, we instructed the TableAdapter to employ the DB Direct approach, which created the methods `Insert`, `Update`, and `Delete`. These methods execute the auto-generated `INSERT`, `UPDATE`, and `DELETE` statements and, consequently, accept input parameters based on the columns returned by the main query. In the Uploading Files tutorial we augmented the `CategoriesTableAdapter` s main query to use the `BrochurePath` column.

Since the `CategoriesTableAdapter` s main query does not reference the `Picture` column, we can neither add a new record nor update an existing record with a value for the `Picture` column. In order to capture this information, we can either create a new method in the TableAdapter that is used specifically to insert a record with binary data or we can customize the auto-generated `INSERT` statement. The problem with customizing the auto-generated `INSERT` statement is that we risk having our customizations overwritten by the wizard. For example, imagine that we customized the `INSERT` statement to include use of the `Picture` column. This would update the TableAdapter s `Insert` method to include an additional input parameter for the category s picture s

binary data. We could then create a method in the Business Logic Layer to use this DAL method and invoke this BLL method through the Presentation Layer, and everything would work wonderfully. That is, until the next time we configured the TableAdapter through the TableAdapter Configuration wizard. As soon as the wizard completed, our customizations to the `INSERT` statement would be overwritten, the `Insert` method would revert to its old form, and our code would no longer compile!

> **NOTE**
> This annoyance is a non-issue when using stored procedures instead of ad-hoc SQL statements. A future tutorial will explore using stored procedures in lieu of ad-hoc SQL statements in the Data Access Layer.

To avoid this potential headache, rather than customizing the auto-generated SQL statements let s instead create a new method for the TableAdapter. This method, named `InsertWithPicture`, will accept values for the `CategoryName`, `Description`, `BrochurePath`, and `Picture` columns and execute an `INSERT` statement that stores all four values in a new record.

Open the Typed DataSet and, from the Designer, right-click on the `CategoriesTableAdapter` s header and choose Add Query from the context menu. This launches the TableAdapter Query Configuration Wizard, which begins by asking us how the TableAdapter query should access the database. Choose Use SQL statements and click Next. The next step prompts for the type of query to be generated. Since we re creating a query to add a new record to the `Categories` table, choose INSERT and click Next.

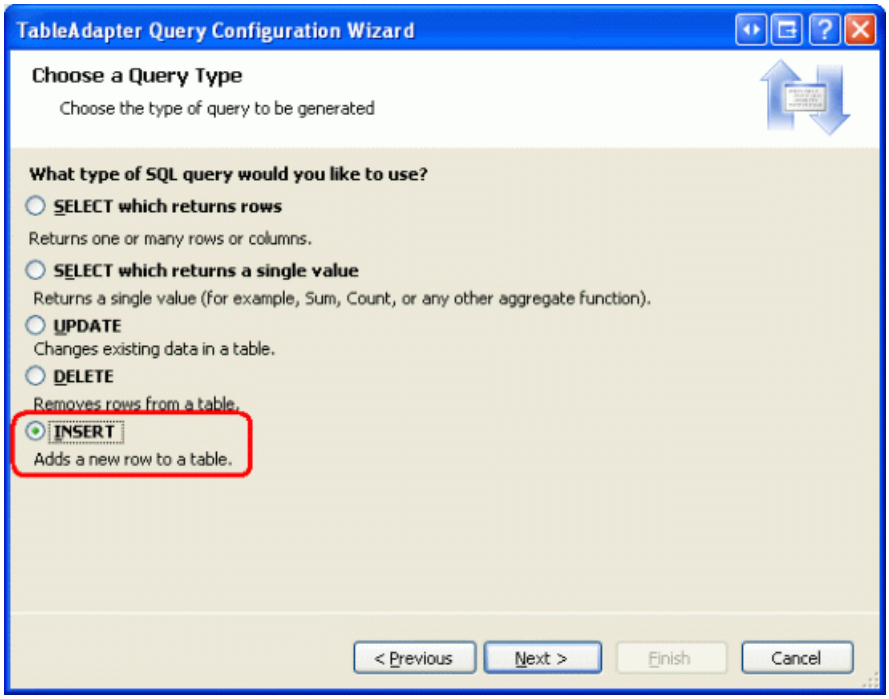


Figure 1: Select the INSERT Option (Click to view full-size image)

We now need to specify the `INSERT` SQL statement. The wizard automatically suggests an `INSERT` statement corresponding to the TableAdapter s main query. In this case, it s an `INSERT` statement that inserts the `CategoryName`, `Description`, and `BrochurePath` values. Update the statement so that the `Picture` column is included along with a `@Picture` parameter, like so:

```
INSERT INTO [Categories]
    ([CategoryName], [Description], [BrochurePath], [Picture])
VALUES
    (@CategoryName, @Description, @BrochurePath, @Picture)
```

The final screen of the wizard asks us to name the new TableAdapter method. Enter `InsertWithPicture` and click

Finish.

Figure 2: Name the New TableAdapter Method `InsertWithPicture` (Click to view full-size image)

Step 2: Updating the Business Logic Layer

Since the Presentation Layer should only interface with the Business Logic Layer rather than bypassing it to go directly to the Data Access Layer, we need to create a BLL method that invokes the DAL method we just created (`InsertWithPicture`). For this tutorial, create a method in the `CategoriesBLL` class named `InsertWithPicture` that accepts as input three `String` s and a `Byte` array. The `String` input parameters are for the category s name, description, and brochure file path, while the `Byte` array is for the binary contents of the category s picture. As the following code shows, this BLL method invokes the corresponding DAL method:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Insert, False)> _
Public Sub InsertWithPicture(categoryName As String, description As String, _
    brochurePath As String, picture() As Byte)

    Adapter.InsertWithPicture(categoryName, description, brochurePath, picture)
End Sub
```

NOTE

Make sure that you have saved the Typed DataSet before adding the `InsertWithPicture` method to the BLL. Since the `CategoriesTableAdapter` class code is auto-generated based on the Typed DataSet, if you don t first save your changes to the Typed DataSet the `Adapter` property won't know about the `InsertWithPicture` method.

Step 3: Listing the Existing Categories and their Binary Data

In this tutorial we will create a page that allows an end user to add a new category to the system, providing a picture and brochure for the new category. In the preceding tutorial we used a GridView with a TemplateField and ImageField to display each category s name, description, picture, and a link to download its brochure. Let s replicate that functionality for this tutorial, creating a page that both lists all existing categories and allows for new ones to be created.

Start by opening the `DisplayOrDownload.aspx` page from the `BinaryData` folder. Go to the Source view and copy the GridView and ObjectDataSource s declarative syntax, pasting it within the `<asp:Content>` element in `UploadInDetailsView.aspx`. Also, don t forget to copy over the `GenerateBrochureLink` method from the code-behind class of `DisplayOrDownload.aspx` to `UploadInDetailsView.aspx`.

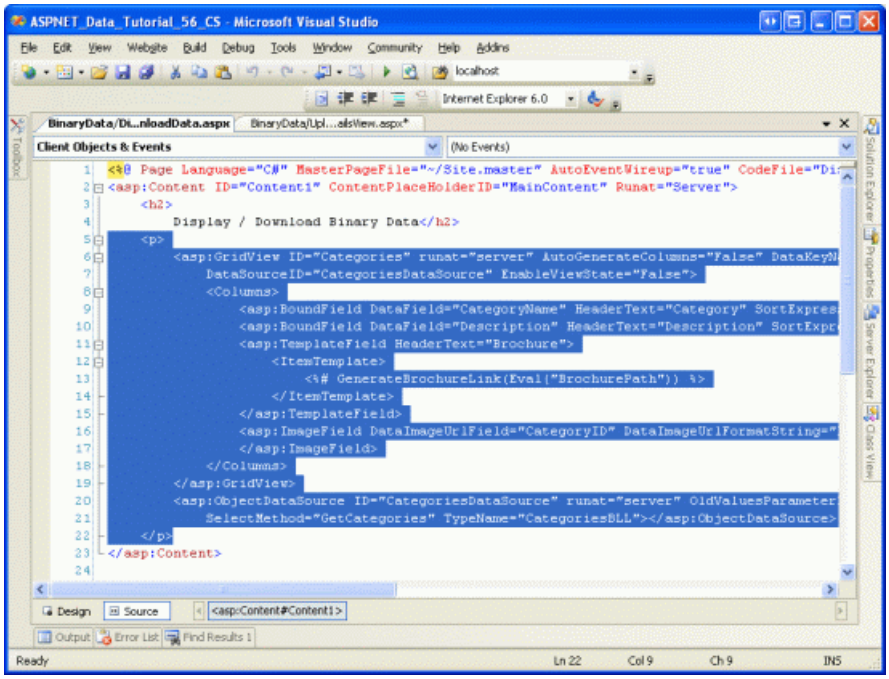


Figure 3: Copy and Paste the Declarative Syntax from `DisplayOrDownload.aspx` to `UploadInDetailsView.aspx` (Click to view full-size image)

After copying the declarative syntax and `GenerateBrochureLink` method over to the `UploadInDetailsView.aspx` page, view the page through a browser to ensure that everything was copied over correctly. You should see a GridView listing the eight categories that includes a link to download the brochure as well as the category s picture.

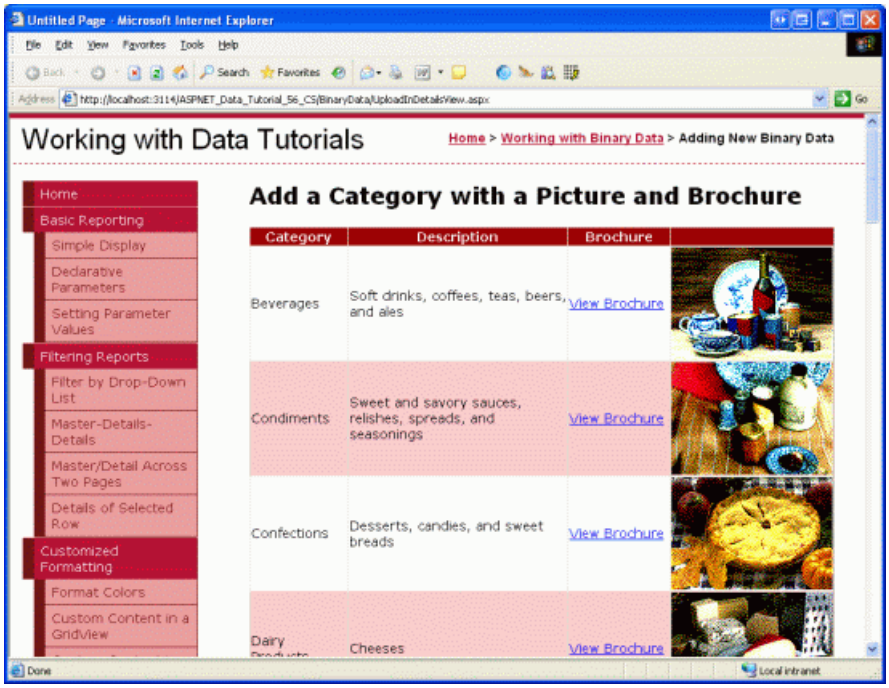


Figure 4: You Should Now See Each Category Along with Its Binary Data (Click to view full-size image)

Step 4: Configuring the `CategoriesDataSource` to Support Inserting

The `CategoriesDataSource` ObjectDataSource used by the `Categories` GridView currently does not provide the

ability to insert data. In order to support inserting through this data source control, we need to map its `Insert` method to a method in its underlying object, `CategoriesBLL`. In particular, we want to map it to the `CategoriesBLL` method we added back in Step 2, `InsertWithPicture`.

Start by clicking the Configure Data Source link from the ObjectDataSource s smart tag. The first screen shows the object the data source is configured to work with, `CategoriesBLL`. Leave this setting as-is and click Next to advance to the Define Data Methods screen. Move to the INSERT tab and pick the `InsertWithPicture` method from the drop-down list. Click Finish to complete the wizard.

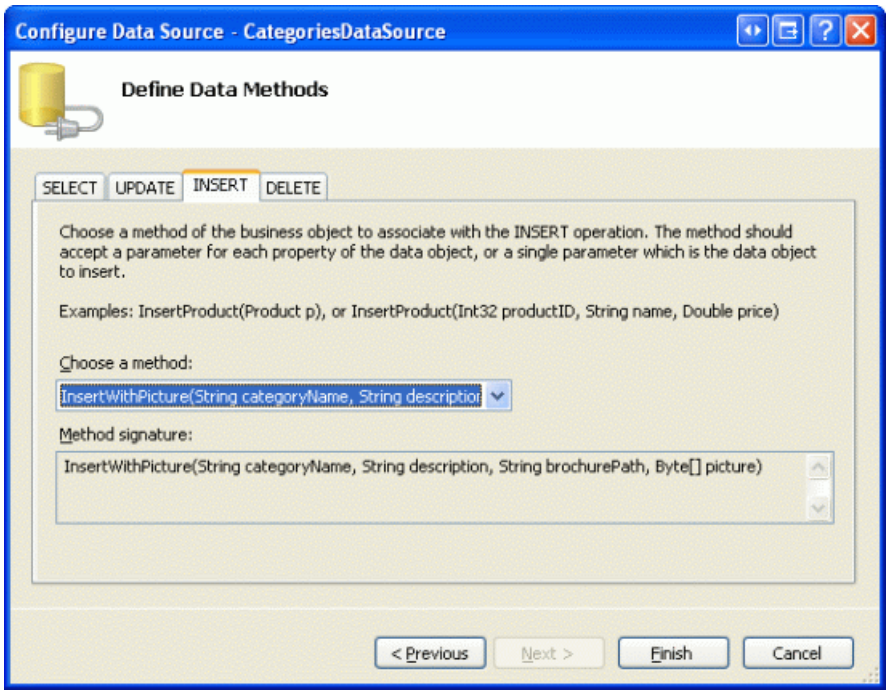


Figure 5: Configure the ObjectDataSource to use the `InsertWithPicture` Method (Click to view full-size image)

> **NOTE**
>
> Upon completing the wizard, Visual Studio may ask if you want to Refresh Fields and Keys, which will regenerate the data Web controls fields. Choose No, because choosing Yes will overwrite any field customizations you may have made.

After completing the wizard, the ObjectDataSource will now include a value for its `InsertMethod` property as well as `InsertParameters` for the four category columns, as the following declarative markup illustrates:

```
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}" SelectMethod="GetCategories"
    TypeName="CategoriesBLL" InsertMethod="InsertWithPicture">
    <InsertParameters>
        <asp:Parameter Name="categoryName" Type="String" />
        <asp:Parameter Name="description" Type="String" />
        <asp:Parameter Name="brochurePath" Type="String" />
        <asp:Parameter Name="picture" Type="Object" />
    </InsertParameters>
</asp:ObjectDataSource>
```

Step 5: Creating the Inserting Interface

As first covered in the An Overview of Inserting, Updating, and Deleting Data, the DetailsView control provides a built-in inserting interface that can be utilized when working with a data source control that supports inserting. Let s add a DetailsView control to this page above the GridView that will permanently render its inserting interface, allowing a user to quickly add a new category. Upon adding a new category in the DetailsView, the

GridView beneath it will automatically refresh and display the new category.

Start by dragging a DetailsView from the Toolbox onto the Designer above the GridView, setting its `ID` property to `NewCategory` and clearing out the `Height` and `Width` property values. From the DetailsView s smart tag, bind it to the existing `CategoriesDataSource` and then check the Enable Inserting checkbox.

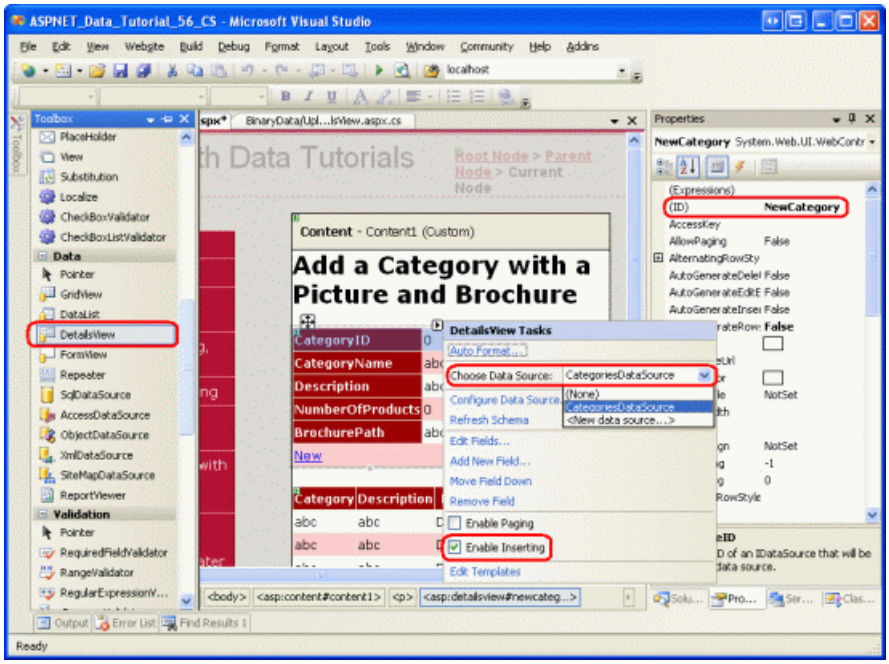


Figure 6: Bind the DetailsView to the `CategoriesDataSource` and Enable Inserting (Click to view full-size image)

To permanently render the DetailsView in its inserting interface, set its `DefaultMode` property to `Insert`.

Note that the DetailsView has five BoundFields `CategoryID`, `CategoryName`, `Description`, `NumberOfProducts`, and `BrochurePath` although the `CategoryID` BoundField is not rendered in the inserting interface because its `InsertVisible` property is set to `False`. These BoundFields exists because they are the columns returned by the `GetCategories()` method, which is what the ObjectDataSource invokes to retrieve its data. For inserting, however, we don t want to let the user specify a value for `NumberOfProducts`. Moreover, we need to allow them to upload a picture for the new category as well as upload a PDF for the brochure.

Remove the `NumberOfProducts` BoundField from the DetailsView altogether and then update the `HeaderText` properties of the `CategoryName` and `BrochurePath` BoundFields to Category and Brochure, respectively. Next, convert the `BrochurePath` BoundField into a TemplateField and add a new TemplateField for the picture, giving this new TemplateField a `HeaderText` value of Picture. Move the `Picture` TemplateField so that it is between the `BrochurePath` TemplateField and CommandField.

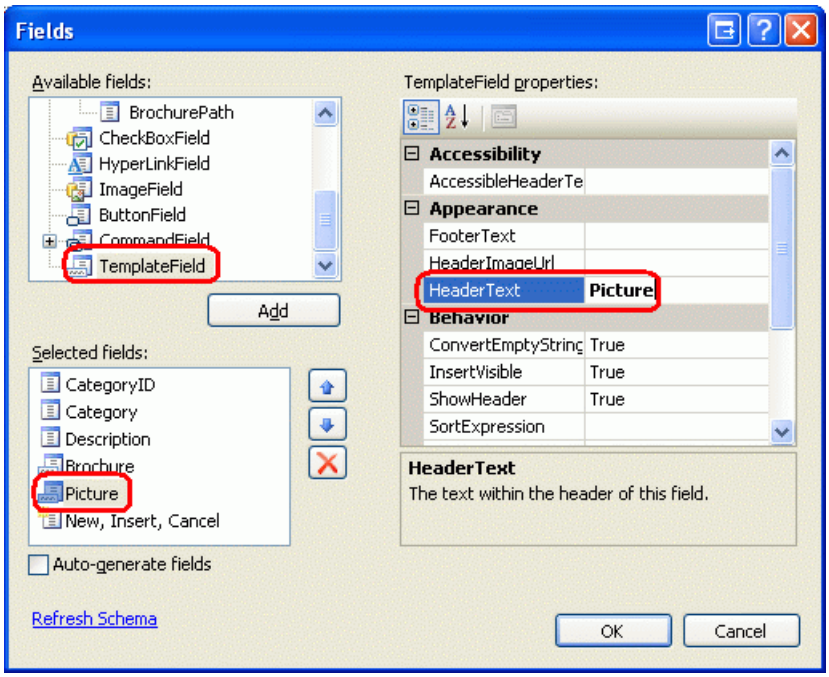


Figure 7: Bind the DetailsView to the `CategoriesDataSource` and Enable Inserting

If you converted the `BrochurePath` BoundField into a TemplateField through the Edit Fields dialog box, the TemplateField includes an `ItemTemplate`, `EditItemTemplate`, and `InsertItemTemplate`. Only the `InsertItemTemplate` is needed, however, so feel free to remove the other two templates. At this point your DetailsView s declarative syntax should look like the following:

```
<asp:DetailsView ID="NewCategory" runat="server" AutoGenerateRows="False"
    DataKeyNames="CategoryID" DataSourceID="CategoriesDataSource"
    DefaultMode="Insert">
    <Fields>
        <asp:BoundField DataField="CategoryID" HeaderText="CategoryID"
            InsertVisible="False" ReadOnly="True"
            SortExpression="CategoryID" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            SortExpression="CategoryName" />
        <asp:BoundField DataField="Description" HeaderText="Description"
            SortExpression="Description" />
        <asp:TemplateField HeaderText="Brochure" SortExpression="BrochurePath">
            <InsertItemTemplate>
                <asp:TextBox ID="TextBox1" runat="server"
                    Text='<%# Bind("BrochurePath") %>'></asp:TextBox>
            </InsertItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Picture"></asp:TemplateField>
        <asp:CommandField ShowInsertButton="True" />
    </Fields>
</asp:DetailsView>
```

Adding FileUpload Controls for the Brochure and Picture Fields

Presently, the `BrochurePath` TemplateField s `InsertItemTemplate` contains a TextBox, while the `Picture` TemplateField does not contain any templates. We need to update these two TemplateField s `InsertItemTemplate` s to use FileUpload controls.

From the DetailsView s smart tag, choose the Edit Templates option and then select the `BrochurePath` TemplateField s `InsertItemTemplate` from the drop-down list. Remove the TextBox and then drag a FileUpload control from the Toolbox into the template. Set the FileUpload control s `ID` to `BrochureUpload`. Similarly, add a FileUpload control to the `Picture` TemplateField s `InsertItemTemplate`. Set this FileUpload control s `ID` to

`PictureUpload`.

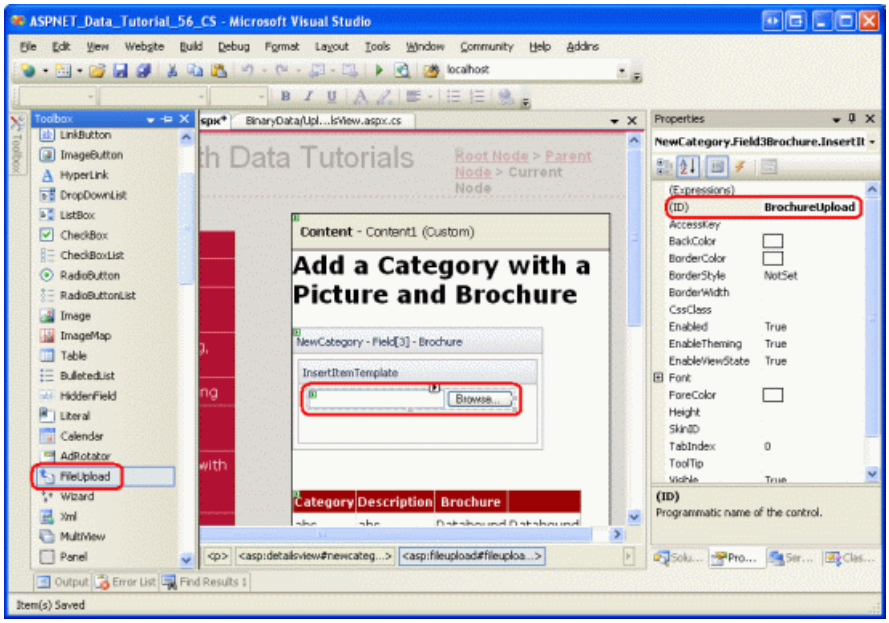


Figure 8: Add a FileUpload Control to the `InsertItemTemplate` (Click to view full-size image)

After making these additions, the two TemplateField s declarative syntax will be:

```
<asp:TemplateField HeaderText="Brochure" SortExpression="BrochurePath">
    <InsertItemTemplate>
        <asp:FileUpload ID="BrochureUpload" runat="server" />
    </InsertItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="Picture">
    <InsertItemTemplate>
        <asp:FileUpload ID="PictureUpload" runat="server" />
    </InsertItemTemplate>
</asp:TemplateField>
```

When a user adds a new category, we want to ensure that the brochure and picture are of the correct file type. For the brochure, the user must supply a PDF. For the picture, we need the user to upload an image file, but do we allow *any* image file or only image files of a particular type, such as GIFs or JPGs? In order to allow for different file types, we d need to extend the `Categories` schema to include a column that captures the file type so that this type can be sent to the client through `Response.ContentType` in `DisplayCategoryPicture.aspx`. Since we don t have such a column, it would be prudent to restrict users to only providing a specific image file type. The `Categories` table s existing images are bitmaps, but JPGs are a more appropriate file format for images served over the web.

If a user uploads an incorrect file type, we need to cancel the insert and display a message indicating the problem. Add a Label Web control beneath the DetailsView. Set its `ID` property to `UploadWarning`, clear out its `Text` property, set the `CssClass` property to Warning, and the `Visible` and `EnableViewState` properties to `False`. The `Warning` CSS class is defined in `Styles.css` and renders the text in a large, red, italicized, bold font.

> **NOTE**
>
> Ideally, the `CategoryName` and `Description` BoundFields would be converted to TemplateFields and their inserting interfaces customized. The `Description` inserting interface, for example, would likely be better suited through a multi-line textbox. And since the `CategoryName` column does not accept `NULL` values, a RequiredFieldValidator should be added to ensure the user provides a value for the new category s name. These steps are left as an exercise to the reader. Refer back to Customizing the Data Modification Interface for an in-depth look at augmenting the data modification interfaces.

Step 6: Saving the Uploaded Brochure to the Web Server s File System

When the user enters the values for a new category and clicks the Insert button, a postback occurs and the inserting workflow unfolds. First, the DetailsView s `ItemInserting` event fires. Next, the ObjectDataSource s `Insert()` method is invoked, which results in a new record being added to the `Categories` table. After that, the DetailsView s `ItemInserted` event fires.

Before the ObjectDataSource s `Insert()` method is invoked, we must first ensure that the appropriate file types were uploaded by the user and then save the brochure PDF to the web server s file system. Create an event handler for the DetailsView s `ItemInserting` event and add the following code:

```
' Reference the FileUpload controls
Dim BrochureUpload As FileUpload = _
    CType(NewCategory.FindControl("BrochureUpload"), FileUpload)
If BrochureUpload.HasFile Then
    ' Make sure that a PDF has been uploaded
    If String.Compare(System.IO.Path.GetExtension _
        (BrochureUpload.FileName), ".pdf", True) <> 0 Then
        UploadWarning.Text = _
            "Only PDF documents may be used for a category's brochure."
        UploadWarning.Visible = True
        e.Cancel = True
        Exit Sub
    End If
End If
```

The event handler starts by referencing the `BrochureUpload` FileUpload control from the DetailsView s templates. Then, if a brochure has been uploaded, the uploaded file s extension is examined. If the extension is not .PDF, then a warning is displayed, the insert is cancelled, and the execution of the event handler ends.

> **NOTE**
> Relying on the uploaded file s extension is not a sure-fire technique for ensuring that the uploaded file is a PDF document. The user could have a valid PDF document with the extension `.Brochure`, or could have taken a non-PDF document and given it a `.pdf` extension. The file s binary content would need to be programmatically examined in order to more conclusively verify the file type. Such thorough approaches, though, are often overkill; checking the extension is sufficient for most scenarios.

As discussed in the Uploading Files tutorial, care must be taken when saving files to the file system so that one user s upload does not overwrite another s. For this tutorial we will attempt to use the same name as the uploaded file. If there already exists a file in the `~/Brochures` directory with that same file name, however, we'll append a number at the end until a unique name is found. For example, if the user uploads a brochure file named `Meats.pdf`, but there is already a file named `Meats.pdf` in the `~/Brochures` folder, we'll change the saved file name to `Meats-1.pdf`. If that exists, we'll try `Meats-2.pdf`, and so on, until a unique file name is found.

The following code uses the `File.Exists(path)` method to determine if a file already exists with the specified file name. If so, it continues to try new file names for the brochure until no conflict is found.

```
Const BrochureDirectory As String = "~/Brochures/"
Dim brochurePath As String = BrochureDirectory & BrochureUpload.FileName
Dim fileNameWithoutExtension As String = _
    System.IO.Path.GetFileNameWithoutExtension(BrochureUpload.FileName)
Dim iteration As Integer = 1
While System.IO.File.Exists(Server.MapPath(brochurePath))
    brochurePath = String.Concat(BrochureDirectory, _
        fileNameWithoutExtension, "-", iteration, ".pdf")
    iteration += 1
End While
```

Once a valid file name has been found, the file needs to be saved to the file system and the ObjectDataSource s `brochurePath``InsertParameter` value needs to be updated so that this file name is written to the database. As we saw back in the *Uploading Files* tutorial, the file can be saved using the FileUpload control s `SaveAs(path)` method. To update the ObjectDataSource s `brochurePath` parameter, use the `e.Values` collection.

```
' Save the file to disk and set the value of the brochurePath parameter
BrochureUpload.SaveAs(Server.MapPath(brochurePath))
e.Values("brochurePath") = brochurePath
```

Step 7: Saving the Uploaded Picture to the Database

To store the uploaded picture in the new `Categories` record, we need to assign the uploaded binary content to the ObjectDataSource s `picture` parameter in the DetailsView s `ItemInserting` event. Before we make this assignment, however, we need to first make sure that the uploaded picture is a JPG and not some other image type. As in Step 6, let s use the uploaded picture s file extension to ascertain its type.

While the `Categories` table allows `NULL` values for the `Picture` column, all categories currently have a picture. Let s force the user to provide a picture when adding a new category through this page. The following code checks to ensure that a picture has been uploaded and that it has an appropriate extension.

```
' Reference the FileUpload controls
Dim PictureUpload As FileUpload = _
    CType(NewCategory.FindControl("PictureUpload"), FileUpload)
If PictureUpload.HasFile Then
    ' Make sure that a JPG has been uploaded
    If  String.Compare(System.IO.Path.GetExtension(PictureUpload.FileName), _
            ".jpg", True) <> 0 AndAlso _
        String.Compare(System.IO.Path.GetExtension(PictureUpload.FileName), _
            ".jpeg", True) <> 0 Then

        UploadWarning.Text = _
            "Only JPG documents may be used for a category's picture."
        UploadWarning.Visible = True
        e.Cancel = True
        Exit Sub
    End If
Else
    ' No picture uploaded!
    UploadWarning.Text = _
        "You must provide a picture for the new category."
    UploadWarning.Visible = True
    e.Cancel = True
    Exit Sub
End If
```

This code should be placed *before* the code from Step 6 so that if there is a problem with the picture upload, the event handler will terminate before the brochure file is saved to the file system.

Assuming that an appropriate file has been uploaded, assign the uploaded binary content to the picture parameter s value with the following line of code:

```
' Set the value of the picture parameter
e.Values("picture") = PictureUpload.FileBytes
```

The Complete `ItemInserting` Event Handler

For completeness, here is the `ItemInserting` event handler in its entirety:

```
Protected Sub NewCategory_ItemInserting _
    (sender As Object, e As DetailsViewInsertEventArgs) _
    Handles NewCategory.ItemInserting

    ' Reference the FileUpload controls
    Dim PictureUpload As FileUpload = _
        CType(NewCategory.FindControl("PictureUpload"), FileUpload)
    If PictureUpload.HasFile Then
        ' Make sure that a JPG has been uploaded
        If  String.Compare(System.IO.Path.GetExtension(PictureUpload.FileName), _
                ".jpg", True) <> 0 AndAlso _
            String.Compare(System.IO.Path.GetExtension(PictureUpload.FileName), _
                ".jpeg", True) <> 0 Then

            UploadWarning.Text = _
                "Only JPG documents may be used for a category's picture."
            UploadWarning.Visible = True
            e.Cancel = True
            Exit Sub
        End If
    Else
        ' No picture uploaded!
        UploadWarning.Text = _
            "You must provide a picture for the new category."
        UploadWarning.Visible = True
        e.Cancel = True
        Exit Sub
    End If
    ' Set the value of the picture parameter
    e.Values("picture") = PictureUpload.FileBytes
    ' Reference the FileUpload controls
    Dim BrochureUpload As FileUpload = _
        CType(NewCategory.FindControl("BrochureUpload"), FileUpload)
    If BrochureUpload.HasFile Then
        ' Make sure that a PDF has been uploaded
        If String.Compare(System.IO.Path.GetExtension(BrochureUpload.FileName), _
            ".pdf", True) <> 0 Then

            UploadWarning.Text = _
                "Only PDF documents may be used for a category's brochure."
            UploadWarning.Visible = True
            e.Cancel = True
            Exit Sub
        End If
        Const BrochureDirectory As String = "~/Brochures/"
        Dim brochurePath As String = BrochureDirectory & BrochureUpload.FileName
        Dim fileNameWithoutExtension As String = _
            System.IO.Path.GetFileNameWithoutExtension(BrochureUpload.FileName)
        Dim iteration As Integer = 1
        While System.IO.File.Exists(Server.MapPath(brochurePath))
            brochurePath = String.Concat(BrochureDirectory, _
                fileNameWithoutExtension, "-", iteration, ".pdf")
            iteration += 1
        End While
        ' Save the file to disk and set the value of the brochurePath parameter
        BrochureUpload.SaveAs(Server.MapPath(brochurePath))
        e.Values("brochurePath") = brochurePath
    End If
End Sub
```

Step 8: Fixing the `DisplayCategoryPicture.aspx` Page

Let s take a moment to test out the inserting interface and `ItemInserting` event handler that was created over the last few steps. Visit the `UploadInDetailsView.aspx` page through a browser and attempt to add a category, but

omit the picture, or specify a non-JPG picture or a non-PDF brochure. In any of these cases, an error message will be displayed and the insert workflow cancelled.

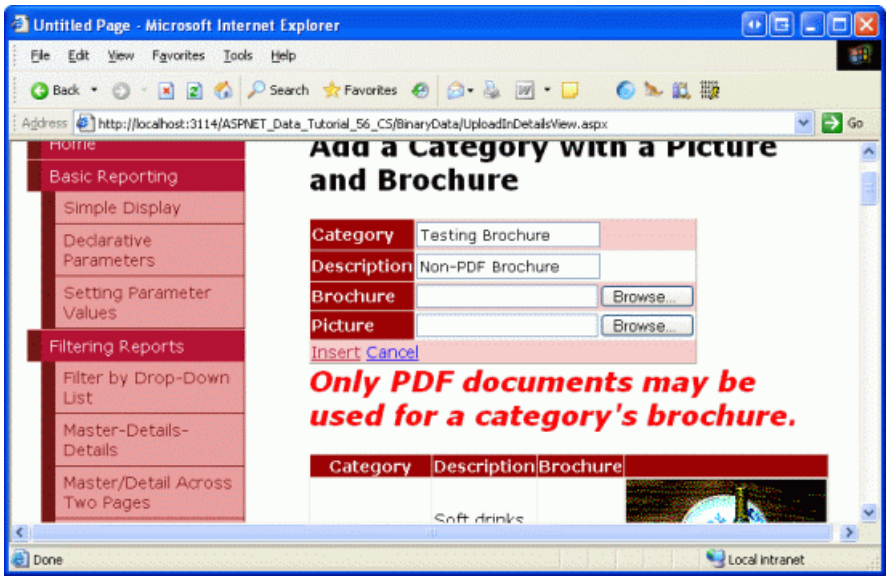


Figure 9: A Warning Message is Displayed If an Invalid File Type is Uploaded (Click to view full-size image)

Once you have verified that the page requires a picture to be uploaded and won't accept non-PDF or non-JPG files, add a new category with a valid JPG picture, leaving the Brochure field empty. After clicking the Insert button, the page will postback and a new record will be added to the `Categories` table with the uploaded image s binary contents stored directly in the database. The GridView is updated and shows a row for the newly added category, but, as Figure 10 shows, the new category s picture is not rendered correctly.

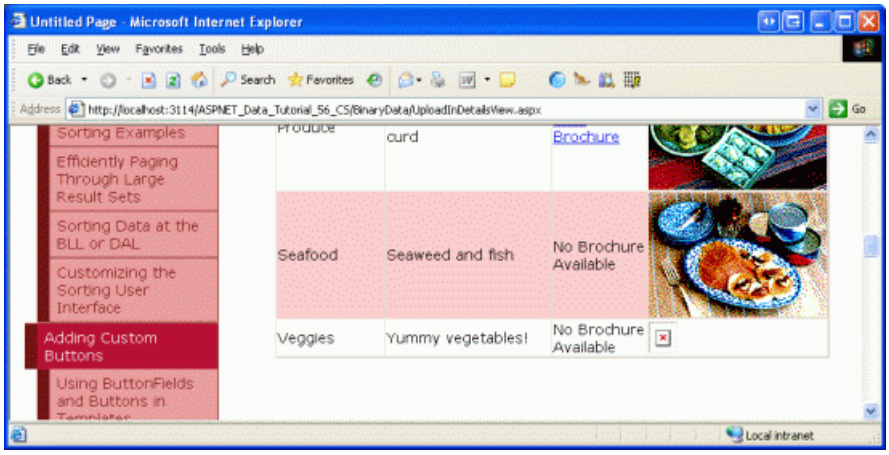


Figure 10: The New Category s Picture is not Displayed (Click to view full-size image)

The reason the new picture is not displayed is because the `DisplayCategoryPicture.aspx` page that returns a specified category s picture is configured to process bitmaps that have an OLE header. This 78 byte header is stripped from the `Picture` column s binary contents before they are sent back to the client. But the JPG file we just uploaded for the new category does not have this OLE header; therefore, valid, necessary bytes are being removed from the image s binary data.

Since there are now both bitmaps with OLE headers and JPGs in the `Categories` table, we need to update `DisplayCategoryPicture.aspx` so that it does the OLE header stripping for the original eight categories and bypasses this stripping for the newer category records. In our next tutorial we'll examine how to update an existing record s image, and we'll update all of the old category pictures so that they are JPGs. For now, though, use the following code in `DisplayCategoryPicture.aspx` to strip the OLE headers only for those original eight categories:

```
Protected Sub Page_Load(sender As Object, e As EventArgs) Handles Me.Load
    Dim categoryID As Integer = Convert.ToInt32(Request.QueryString("CategoryID"))
    ' Get information about the specified category
    Dim categoryAPI As New CategoriesBLL()
    Dim categories As Northwind.CategoriesDataTable = _
        categoryAPI.GetCategoryWithBinaryDataByCategoryID(categoryID)
    Dim category As Northwind.CategoriesRow = categories(0)
    If categoryID <= 8 Then
        ' Output HTTP headers providing information about the binary data
        Response.ContentType = "image/bmp"
        ' Output the binary data
        ' But first we need to strip out the OLE header
        Const OleHeaderLength As Integer = 78
        Dim strippedImageLength As Integer = _
            category.Picture.Length - OleHeaderLength
        Dim strippedImageData(strippedImageLength) As Byte
        Array.Copy(category.Picture, OleHeaderLength, _
            strippedImageData, 0, strippedImageLength)
        Response.BinaryWrite(strippedImageData)
    Else
        ' For new categories, images are JPGs...
        ' Output HTTP headers providing information about the binary data
        Response.ContentType = "image/jpeg"
        ' Output the binary data
        Response.BinaryWrite(category.Picture)
    End If
End Sub
```

With this change, the JPG image is now rendered correctly in the GridView.

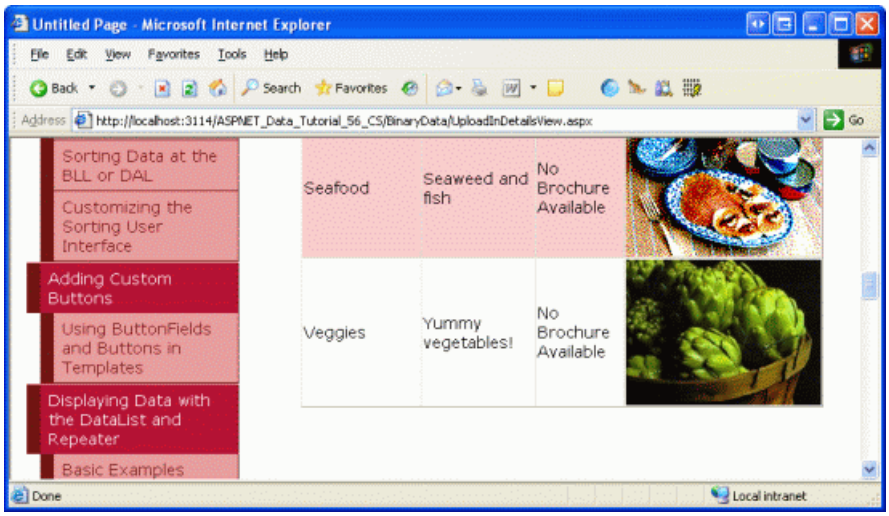


Figure 11: The JPG Images for New Categories are Correctly Rendered (Click to view full-size image)

Step 9: Deleting the Brochure in the Face of an Exception

One of the challenges of storing binary data on the web server s file system is that it introduces a disconnect between the data model and its binary data. Therefore, whenever a record is deleted, the corresponding binary data on the file system must also be removed. This can come into play when inserting, as well. Consider the following scenario: a user adds a new category, specifying a valid picture and brochure. Upon clicking the Insert button, a postback occurs and the DetailsView s `ItemInserting` event fires, saving the brochure to the web server s file system. Next, the ObjectDataSource s `Insert()` method is invoked, which calls the `CategoriesBLL` class s `InsertWithPicture` method, which calls the `CategoriesTableAdapter` s `InsertWithPicture` method.

Now, what happens if the database is offline, or if there is an error in the `INSERT` SQL statement? Clearly the INSERT will fail, so no new category row will be added to the database. But we still have the uploaded brochure file sitting on the web server s file system! This file needs to be deleted in the face of an exception during the

inserting workflow.

As discussed previously in the Handling BLL- and DAL-Level Exceptions in an ASP.NET Page tutorial, when an exception is thrown from within the depths of the architecture it is bubbled up through the various layers. At the Presentation Layer, we can determine if an exception has occurred from the DetailsView s `ItemInserted` event. This event handler also provides the values of the ObjectDataSource s `InsertParameters`. Therefore, we can create an event handler for the `ItemInserted` event that checks if there was an exception and, if so, deletes the file specified by the ObjectDataSource s `brochurePath` parameter:

```
Protected Sub NewCategory_ItemInserted _
    (sender As Object, e As DetailsViewInsertedEventArgs) _
    Handles NewCategory.ItemInserted

    If e.Exception IsNot Nothing Then
        ' Need to delete brochure file, if it exists
        If e.Values("brochurePath") IsNot Nothing Then
            System.IO.File.Delete(Server.MapPath _
                (e.Values("brochurePath").ToString()))
        End If
    End If
End Sub
```

Summary

There are a number of steps that must be performed in order to provide a web-based interface for adding records that include binary data. If the binary data is being stored directly into the database, chances are you'll need to update the architecture, adding specific methods to handle the case where binary data is being inserted. Once the architecture has been updated, the next step is creating the inserting interface, which can be accomplished using a DetailsView that has been customized to include a FileUpload control for each binary data field. The uploaded data can then be saved to the web server s file system or assigned to a data source parameter in the DetailsView s `ItemInserting` event handler.

Saving binary data to the file system requires more planning than saving data directly into the database. A naming scheme must be chosen in order to avoid one user s upload overwriting another s. Also, extra steps must be taken to delete the uploaded file if the database insert fails.

We now have the ability to add new categories to the system with a brochure and picture, but we ve yet to look at how to update an existing category s binary data or how to correctly remove the binary data for a deleted category. We'll explore these two topics in the next tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Dave Gardner, Teresa Murphy, and Bernadette Leigh. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Updating and Deleting Existing Binary Data (VB)

2/20/2020 • 30 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In earlier tutorials we saw how the GridView control makes it simple to edit and delete text data. In this tutorial we see how the GridView control also makes it possible to edit and delete binary data, whether that binary data is saved in the database or stored in the file system.

Introduction

Over the past three tutorials we've added quite a bit of functionality for working with binary data. We started by adding a `BrochurePath` column to the `Categories` table and updated the architecture accordingly. We also added Data Access Layer and Business Logic Layer methods to work with the Categories table's existing `Picture` column, which holds the binary content's of an image file. We have built web pages to present the binary data in a GridView a download link for the brochure, with the category's picture shown in an `<img>` element and have added a DetailsView to allow users to add a new category and upload its brochure and picture data.

All that remains to be implemented is the ability to edit and delete existing categories, which we'll accomplish in this tutorial using the GridView's built-in editing and deleting features. When editing a category, the user will be able to optionally upload a new picture or have the category continue to use the existing one. For the brochure, they can either choose to use the existing brochure, to upload a new brochure, or to indicate that the category no longer has a brochure associated with it. Let's get started!

Step 1: Updating the Data Access Layer

The DAL has auto-generated `Insert`, `Update`, and `Delete` methods, but these methods were generated based on the `CategoriesTableAdapter`'s main query, which does not include the `Picture` column. Therefore, the `Insert` and `Update` methods do not include parameters for specifying the binary data for the category's picture. Like we did in the preceding tutorial, we need to create a new TableAdapter method for updating the `Categories` table when specifying binary data.

Open the Typed DataSet and, from the Designer, right-click on the `CategoriesTableAdapter`'s header and choose Add Query from the context menu to launch the TableAdapter Query Configuration Wizard. This wizard starts by asking us how the TableAdapter query should access the database. Choose Use SQL statements and click Next. The next step prompts for the type of query to be generated. Since we're creating a query to add a new record to the `Categories` table, choose UPDATE and click Next.

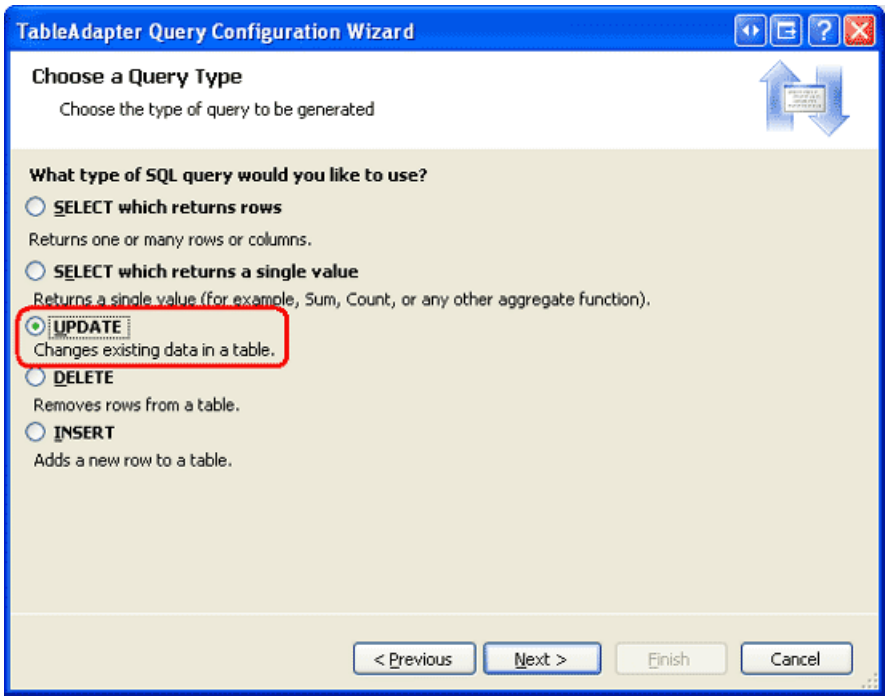


Figure 1: Select the UPDATE Option (Click to view full-size image)

We now need to specify the `UPDATE` SQL statement. The wizard automatically suggests an `UPDATE` statement corresponding to the TableAdapter s main query (one that updates the `CategoryName`, `Description`, and `BrochurePath` values). Change the statement so that the `Picture` column is included along with a `@Picture` parameter, like so:

```
UPDATE [Categories] SET
    [CategoryName] = @CategoryName,
    [Description] = @Description,
    [BrochurePath] = @BrochurePath ,
    [Picture] = @Picture
WHERE (([CategoryID] = @Original_CategoryID))
```

The final screen of the wizard asks us to name the new TableAdapter method. Enter `UpdateWithPicture` and click Finish.

Figure 2: Name the New TableAdapter Method `UpdateWithPicture` (Click to view full-size image)

Step 2: Adding the Business Logic Layer Methods

In addition to updating the DAL, we need to update the BLL to include methods for updating and deleting a category. These are the methods that will be invoked from the Presentation Layer.

For deleting a category, we can use the `CategoriesTableAdapter` s auto-generated `Delete` method. Add the following method to the `CategoriesBLL` class:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Delete, True)> _
Public Function DeleteCategory(ByVal categoryID As Integer) As Boolean
    Dim rowsAffected As Integer = Adapter.Delete(categoryID)
    ' Return true if precisely one row was deleted, otherwise false
    Return rowsAffected = 1
End Function
```

For this tutorial, let s create two methods for updating a category - one that expects the binary picture data and invokes the `UpdateWithPicture` method we just added to the `CategoriesTableAdapter` and another that accepts just the `CategoryName`, `Description`, and `BrochurePath` values and uses `CategoriesTableAdapter` class s auto-generated `Update` statement. The rationale behind using two methods is that in some circumstances, a user might want to update the category s picture along with its other fields, in which case the user will have to upload the new picture. The uploaded picture s binary data can then be used in the `UPDATE` statement. In other cases, the user might only be interested in updating, say, the name and description. But if the `UPDATE` statement expects the binary data for the `Picture` column as well, then we d need to provide that information as well. This would require an extra trip to the database to bring back the picture data for the record being edited. Therefore, we want two `UPDATE` methods. The Business Logic Layer will determine which one to use based on whether picture data is provided when updating the category.

To facilitate this, add two methods to the `CategoriesBLL` class, both named `UpdateCategory`. The first one should accept three `String` s, a `Byte` array, and an `Integer` as its input parameters; the second, just three `String` s and an `Integer`. The `String` input parameters are for the category s name, description, and brochure file path, the `Byte` array is for the binary contents of the category s picture, and the `Integer` identifies the `CategoryID` of the record to update. Notice that the first overload invokes the second if the passed-in `Byte` array is `Nothing`:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Update, False)> _
Public Function UpdateCategory(categoryName As String, description As String, _
    brochurePath As String, picture() As Byte, categoryID As Integer) As Boolean

    ' If no picture is specified, use other overload
    If picture Is Nothing Then
        Return UpdateCategory(categoryName, description, brochurePath, categoryID)
    End If
    ' Update picture, as well
    Dim rowsAffected As Integer = Adapter.UpdateWithPicture _
        (categoryName, description, brochurePath, picture, categoryID)
    ' Return true if precisely one row was updated, otherwise false
    Return rowsAffected = 1
End Function
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Update, True)> _
Public Function UpdateCategory(categoryName As String, description As String, _
    brochurePath As String, categoryID As Integer) As Boolean
    Dim rowsAffected As Integer = Adapter.Update _
        (categoryName, description, brochurePath, categoryID)
    ' Return true if precisely one row was updated, otherwise false
    Return rowsAffected = 1
End Function
```

Step 3: Copying Over the Insert and View Functionality

In the preceding tutorial we created a page named `UploadInDetailsView.aspx` that listed all categories in a GridView and provided a DetailsView to add new categories to the system. In this tutorial we will extend the GridView to include editing and deleting support. Rather than continuing to work from `UploadInDetailsView.aspx`, let s instead place this tutorial s changes in the `UpdatingAndDeleting.aspx` page from the same folder, `~/BinaryData`. Copy and paste the declarative markup and code from `UploadInDetailsView.aspx` to `UpdatingAndDeleting.aspx`.

Start by opening the `UploadInDetailsView.aspx` page. Copy all of the declarative syntax within the `<asp:Content>` element, as shown in Figure 3. Next, open `UpdatingAndDeleting.aspx` and paste this markup within its `<asp:Content>` element. Similarly, copy the code from the `UploadInDetailsView.aspx` page s code-behind class to `UpdatingAndDeleting.aspx`.

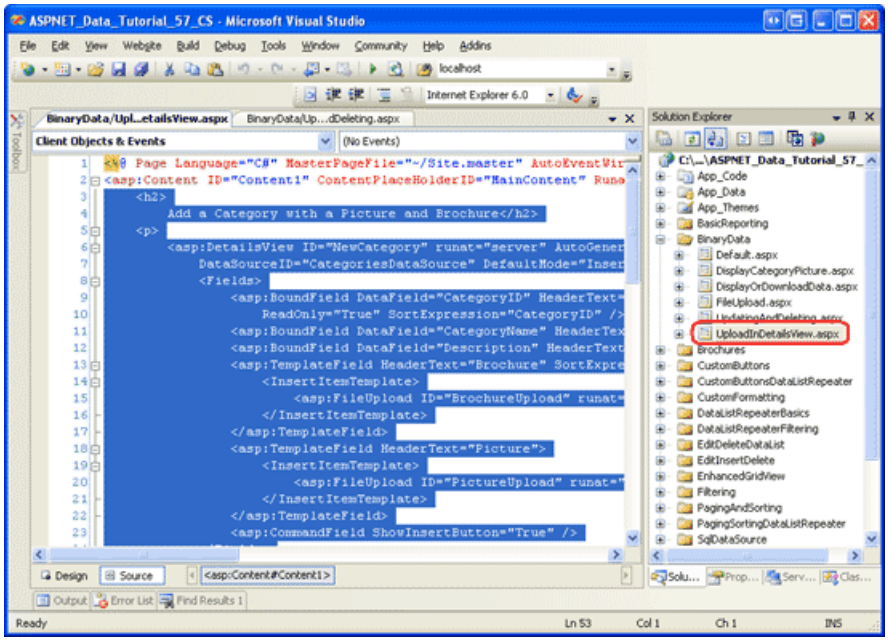


Figure 3: Copy the Declarative Markup from `UploadInDetailsView.aspx` (Click to view full-size image)

After copying over the declarative markup and code, visit `UpdatingAndDeleting.aspx`. You should see the same output and have the same user experience as with `UploadInDetailsView.aspx` page from the previous tutorial.

Step 4: Adding Deleting Support to the ObjectDataSource and GridView

As we discussed back in the An Overview of Inserting, Updating, and Deleting Data tutorial, the GridView provides built-in deleting capabilities and these capabilities can be enabled at the tick of a checkbox if the grid s underlying data source supports deleting. Currently the ObjectDataSource the GridView is bound to (`CategoriesDataSource`) does not support deleting.

To remedy this, click on the Configure Data Source option from the ObjectDataSource s smart tag to launch the wizard. The first screen shows that the ObjectDataSource is configured to work with the `CategoriesBLL` class. Hit Next. Currently, only the ObjectDataSource s `InsertMethod` and `SelectMethod` properties are specified. However, the wizard auto-populated the drop-down lists in the UPDATE and DELETE tabs with the `UpdateCategory` and `DeleteCategory` methods, respectively. This is because in the `CategoriesBLL` class we marked these methods using the `DataObjectMethodAttribute` as the default methods for updating and deleting.

For now, set the UPDATE tab s drop-down list to (None), but leave the DELETE tab s drop-down list set to `DeleteCategory`. We'll return to this wizard in Step 6 to add updating support.

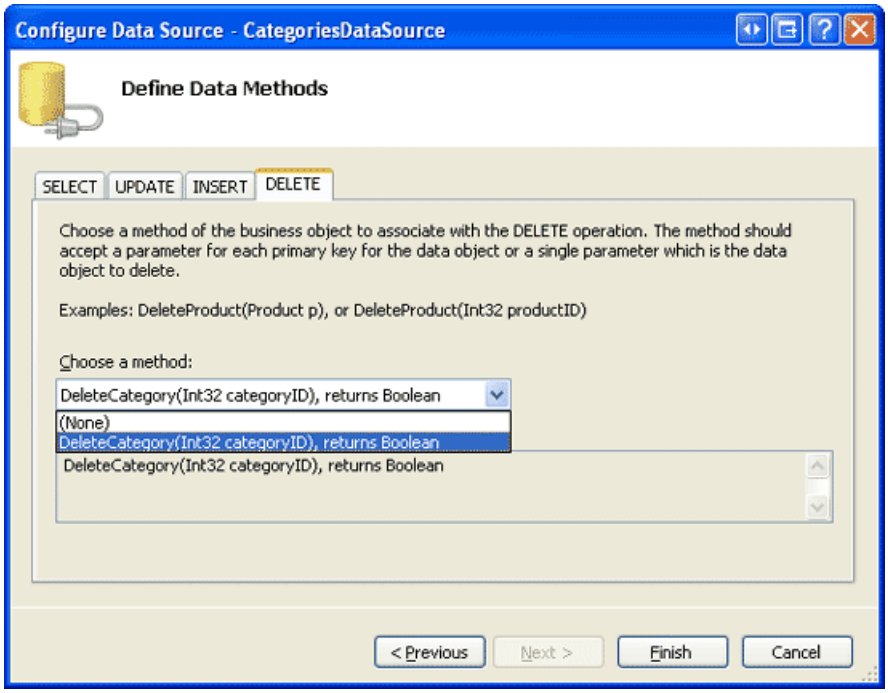


Figure 4: Configure the ObjectDataSource to Use the `DeleteCategory` Method (Click to view full-size image)

> **NOTE**
>
> Upon completing the wizard, Visual Studio may ask if you want to Refresh Fields and Keys, which will regenerate the data Web controls fields. Choose No, because choosing Yes will overwrite any field customizations you may have made.

The ObjectDataSource will now include a value for its `DeleteMethod` property as well as a `DeleteParameter`. Recall that when using the wizard to specify the methods, Visual Studio sets the ObjectDataSource s `OldValuesParameterFormatString` property to `original_{0}`, which causes problems with the update and delete method invocations. Therefore, either clear out this property altogether or reset it to the default, `{0}`. If you need to refresh your memory on this ObjectDataSource property, see the An Overview of Inserting, Updating, and Deleting Data tutorial.

After completing the wizard and fixing the `OldValuesParameterFormatString`, the ObjectDataSource s declarative markup should look similar like the following:

```
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="{0}" SelectMethod="GetCategories"
    TypeName="CategoriesBLL" InsertMethod="InsertWithPicture"
    DeleteMethod="DeleteCategory">
    <InsertParameters>
        <asp:Parameter Name="categoryName" Type="String" />
        <asp:Parameter Name="description" Type="String" />
        <asp:Parameter Name="brochurePath" Type="String" />
        <asp:Parameter Name="picture" Type="Object" />
    </InsertParameters>
    <DeleteParameters>
        <asp:Parameter Name="categoryID" Type="Int32" />
    </DeleteParameters>
</asp:ObjectDataSource>
```

After configuring the ObjectDataSource, add deleting capabilities to the GridView by checking the Enable Deleting checkbox from the GridView s smart tag. This will add a CommandField to the GridView whose `ShowDeleteButton` property is set to `True`.

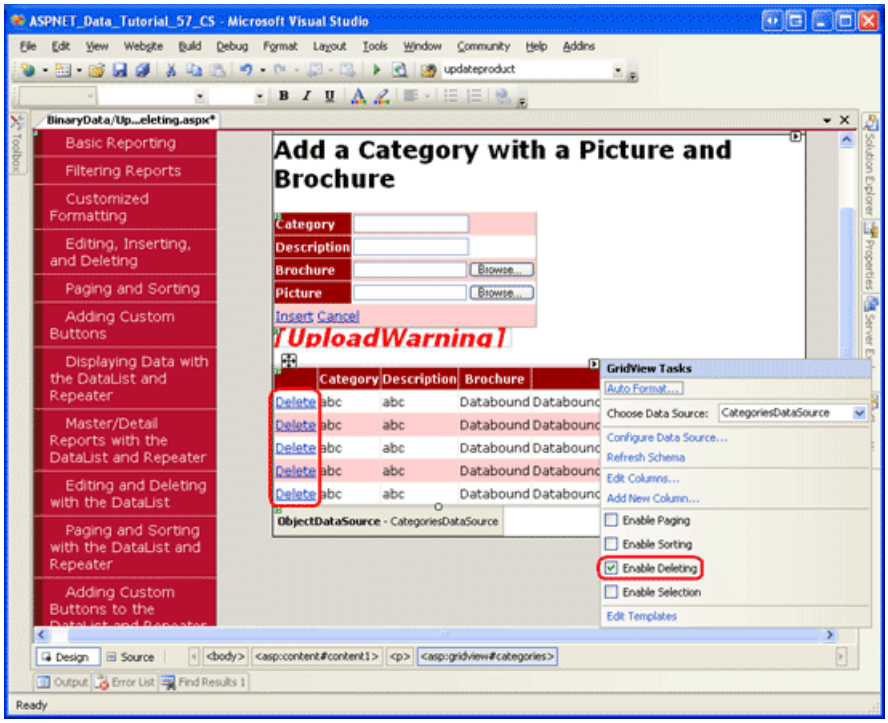


Figure 5: Enable Support for Deleting in the GridView (Click to view full-size image)

Take a moment to test out the delete functionality. There is a foreign key between the `Products` table s `CategoryID` and the `Categories` table s `CategoryID`, so you will get a foreign key constraint violation exception if you attempt to delete any of the first eight categories. To test this functionality out, add a new category, providing both a brochure and picture. My test category, shown in Figure 6, includes a test brochure file named `Test.pdf` and a test picture. Figure 7 shows the GridView after the test category has been added.

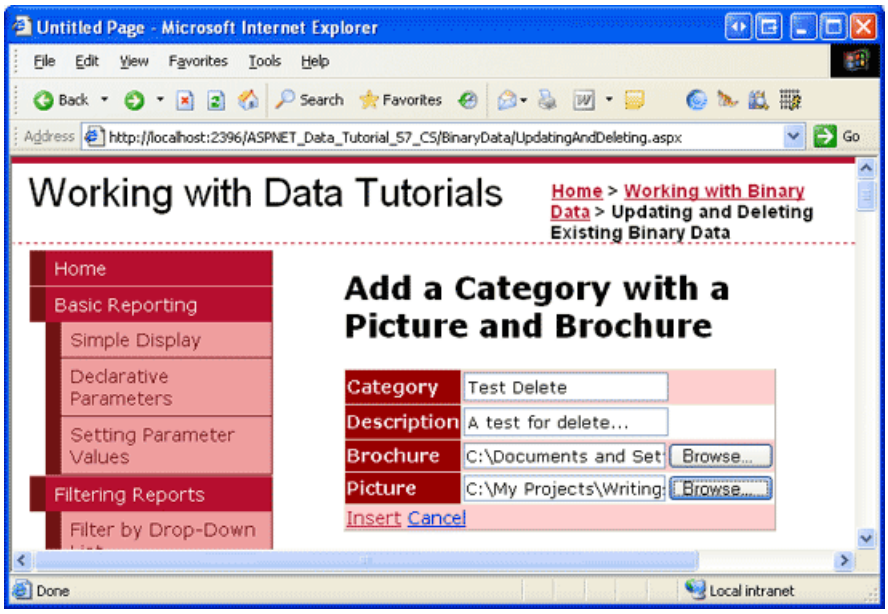


Figure 6: Add a Test Category with a Brochure and Image (Click to view full-size image)

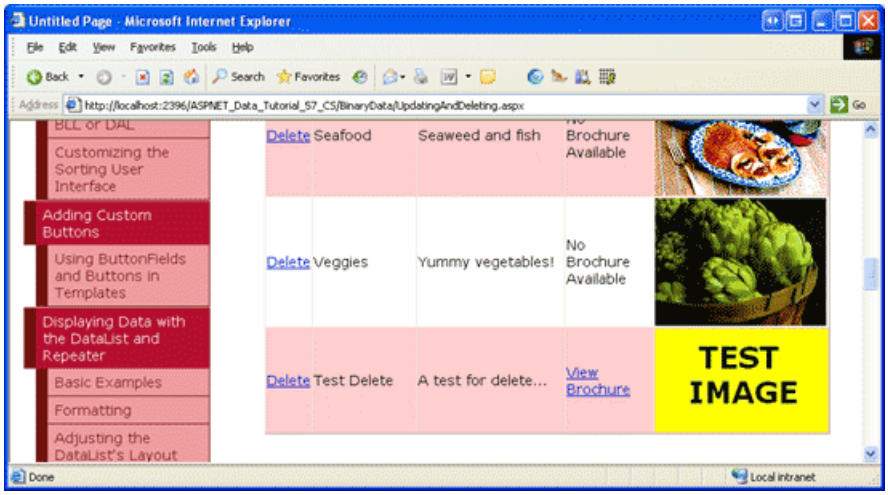


Figure 7: After Inserting the Test Category, it is Displayed in the GridView (Click to view full-size image)

In Visual Studio, refresh the Solution Explorer. You should now see a new file in the `~/Brochures` folder, `Test.pdf` (see Figure 8).

Next, click the Delete link in the Test Category row, causing the page to postback and the `CategoriesBLL` class s `DeleteCategory` method to fire. This will invoke the DAL s `Delete` method, causing the appropriate `DELETE` statement to be sent to the database. The data is then rebound to the GridView and the markup is sent back to the client with the Test Category no longer present.

While the delete workflow successfully removed the Test Category record from the `Categories` table, it did not remove its brochure file from the web server s file system. Refresh the Solution Explorer and you will see that `Test.pdf` is still sitting in the `~/Brochures` folder.

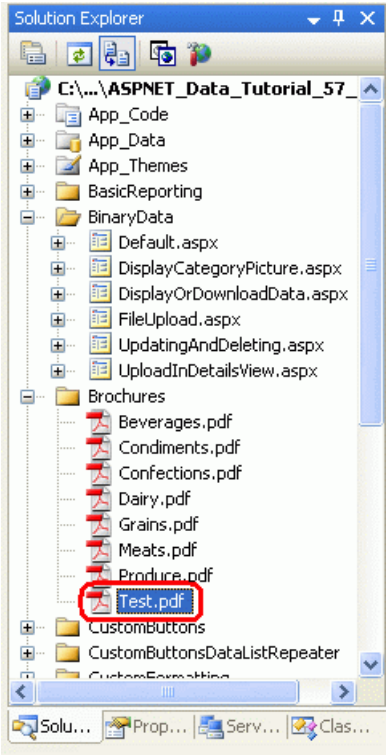


Figure 8: The `Test.pdf` File Was Not Deleted from the Web Server s File System

Step 5: Removing the Deleted Category s Brochure File

One of the downsides of storing binary data external to the database is that extra steps must be taken to clean up these files when the associated database record is deleted. The GridView and ObjectDataSource provide events that fire both before and after the delete command has been performed. We actually need to create event handlers for both the pre- and post-action events. Before the `Categories` record is deleted we need to determine its PDF file s path, but we don t want to delete the PDF before the category is deleted in case there is some exception and the category is not deleted.

The GridView s `RowDeleting` event fires before the ObjectDataSource s delete command has been invoked, while its `RowDeleted` event fires after. Create event handlers for these two events using the following code:

```
' A page variable to "remember" the deleted category's BrochurePath value
Private deletedCategorysPdfPath As String = Nothing
Protected Sub Categories_RowDeleting(sender As Object, e As GridViewDeleteEventArgs) _
    Handles Categories.RowDeleting

    ' Determine the PDF path for the category being deleted...
    Dim categoryID As Integer = Convert.ToInt32(e.Keys("CategoryID"))
    Dim categoryAPI As New CategoriesBLL()
    Dim categoriesData As Northwind.CategoriesDataTable = _
        categoryAPI.GetCategoryByCategoryID(categoryID)
    Dim category As Northwind.CategoriesRow = categoriesData(0)
    If category.IsBrochurePathNull() Then
        deletedCategorysPdfPath = Nothing
    Else
        deletedCategorysPdfPath = category.BrochurePath
    End If
End Sub
Protected Sub Categories_RowDeleted(sender As Object, e As GridViewDeletedEventArgs) _
    Handles Categories.RowDeleted

    ' Delete the brochure file if there were no problems deleting the record
    If e.Exception Is Nothing Then
        DeleteRememberedBrochurePath()
    End If
End Sub
```

In the `RowDeleting` event handler, the `CategoryID` of the row being deleted is grabbed from the GridView s `DataKeys` collection, which can be accessed in this event handler through the `e.Keys` collection. Next, the `CategoriesBLL` class s `GetCategoryByCategoryID(categoryID)` is invoked to return information about the record being deleted. If the returned `CategoriesDataRow` object has a non- `NULL``BrochurePath` value then it is stored in the page variable `deletedCategorysPdfPath` so that the file can be deleted in the `RowDeleted` event handler.

> **NOTE**
> Rather than retrieving the `BrochurePath` details for the `Categories` record being deleted in the `RowDeleting` event handler, we could have alternatively added the `BrochurePath` to the GridView s `DataKeyNames` property and accessed the record s value through the `e.Keys` collection. Doing so would slightly increase the GridView s view state size, but would reduce the amount of code needed and save a trip to the database.

After the ObjectDataSource s underlying delete command has been invoked, the GridView s `RowDeleted` event handler fires. If there were no exceptions in deleting the data and there is a value for `deletedCategorysPdfPath`, then the PDF is deleted from the file system. Note that this extra code is not needed to clean up the category s binary data associated with its picture. That s because the picture data is stored directly in the database, so deleting the `Categories` row also deletes that category s picture data.

After adding the two event handlers, run this test case again. When deleting the category, its associated PDF is also deleted.

Updating an existing record s associated binary data provides some interesting challenges. The remainder of this tutorial delves into adding update capabilities to the brochure and picture. Step 6 explores techniques for updating the brochure information while Step 7 looks at updating the picture.

Step 6: Updating a Category s Brochure

As discussed in the An Overview of Inserting, Updating, and Deleting Data tutorial, the GridView offers built-in row-level editing support that can be implemented by the tick of a checkbox if its underlying data source is appropriately configured. Currently, the `CategoriesDataSource` ObjectDataSource is not yet configured to include updating support, so let s add that in.

Click the Configure Data Source link from the ObjectDataSource s wizard and proceed to the second step. Because of the `DataObjectMethodAttribute` used in `CategoriesBLL`, the UPDATE drop-down list should automatically be populated with the `UpdateCategory` overload that accepts four input parameters (for all columns but `Picture`). Change this so that it uses the overload with five parameters.

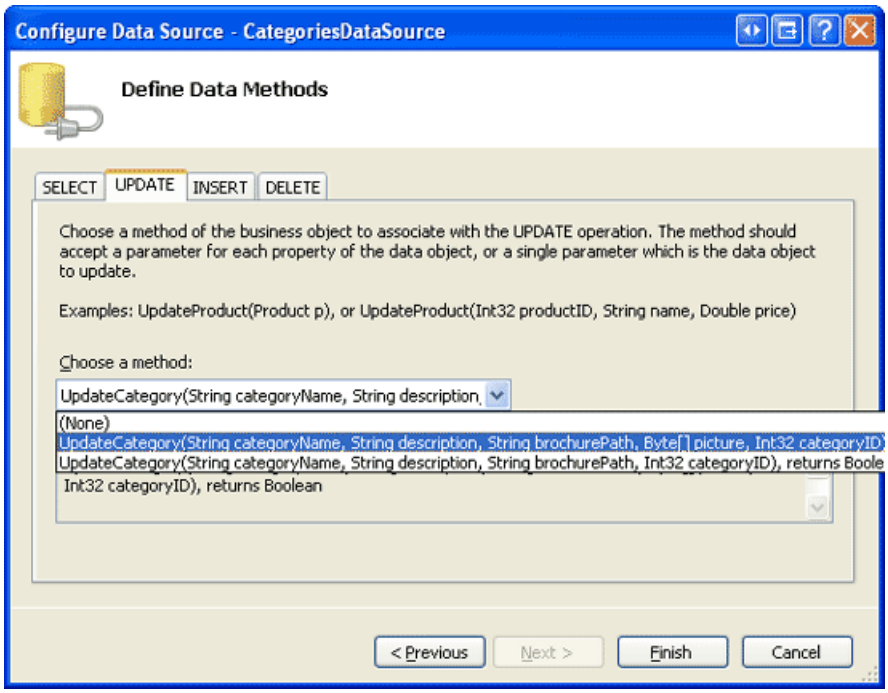


Figure 9: Configure the ObjectDataSource to Use the `UpdateCategory` Method that Includes a Parameter for `Picture` (Click to view full-size image)

The ObjectDataSource will now include a value for its `UpdateMethod` property as well as corresponding `UpdateParameter` s. As noted in Step 4, Visual Studio sets the ObjectDataSource s `OldValuesParameterFormatString` property to `original_{0}` when using the Configure Data Source wizard. This will cause problems with the update and delete method invocations. Therefore, either clear out this property altogether or reset it to the default, `{0}`.

After completing the wizard and fixing the `OldValuesParameterFormatString`, the ObjectDataSource s declarative markup should look like the following:

```
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="{0}" SelectMethod="GetCategories"
    TypeName="CategoriesBLL" InsertMethod="InsertWithPicture"
    DeleteMethod="DeleteCategory" UpdateMethod="UpdateCategory">
    <InsertParameters>
        <asp:Parameter Name="categoryName" Type="String" />
        <asp:Parameter Name="description" Type="String" />
        <asp:Parameter Name="brochurePath" Type="String" />
        <asp:Parameter Name="picture" Type="Object" />
    </InsertParameters>
    <DeleteParameters>
        <asp:Parameter Name="categoryID" Type="Int32" />
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="categoryName" Type="String" />
        <asp:Parameter Name="description" Type="String" />
        <asp:Parameter Name="brochurePath" Type="String" />
        <asp:Parameter Name="picture" Type="Object" />
        <asp:Parameter Name="categoryID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

To turn on the GridView s built-in editing features, check the Enable Editing option from the GridView s smart tag.

This will set the CommandField s `ShowEditButton` property to `True`, resulting in the addition of an Edit button (and Update and Cancel buttons for the row being edited).

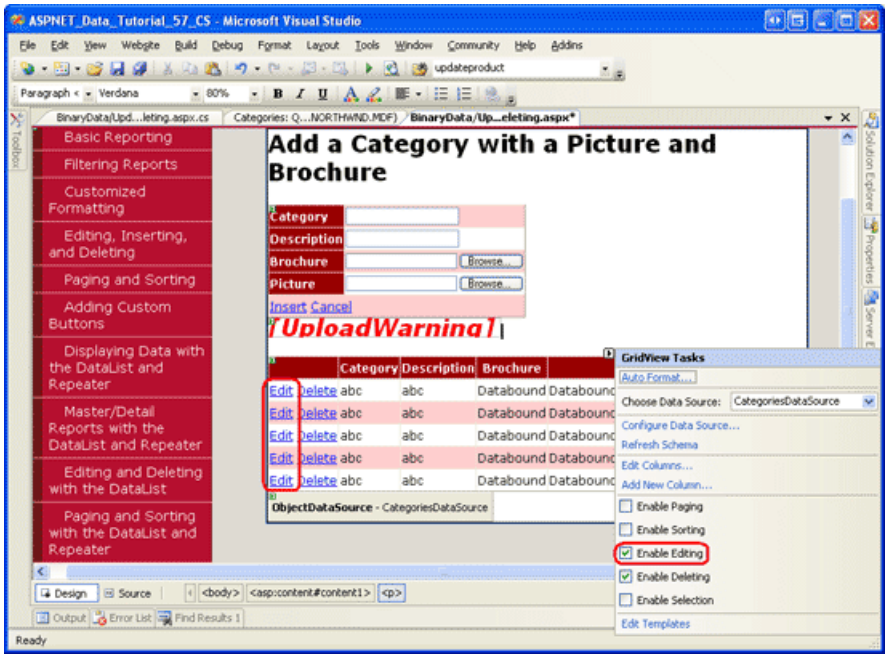


Figure 10: Configure the GridView to Support Editing (Click to view full-size image)

Visit the page through a browser and click one of the row s Edit buttons. The `CategoryName` and `Description` BoundFields are rendered as textboxes. The `BrochurePath` TemplateField lacks an `EditItemTemplate`, so it continues to show its `ItemTemplate` a link to the brochure. The `Picture` ImageField renders as a TextBox whose `Text` property is assigned the value of the ImageField s `DataImageUrlField` value, in this case `CategoryID`.

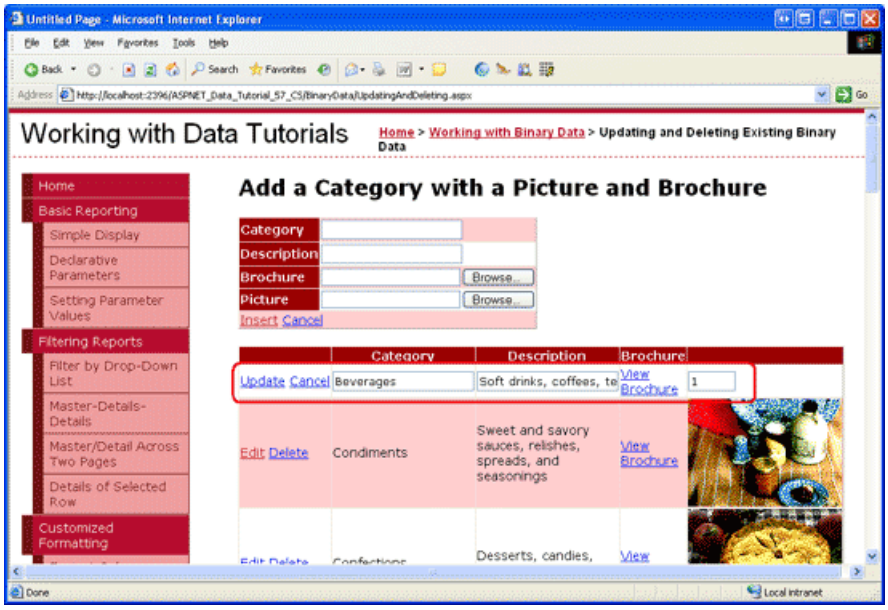


Figure 11: The GridView Lacks an Editing Interface for `BrochurePath` (Click to view full-size image)

Customizing the `BrochurePath` s Editing Interface

We need to create an editing interface for the `BrochurePath` TemplateField, one that allows the user to either:

- Leave the category s brochure as-is,
- Update the category s brochure by uploading a new brochure, or
- Remove the category s brochure altogether (in the case that the category no longer has an associated brochure).

We also need to update the `Picture` ImageField s editing interface, but we'll get to this in Step 7.

From the GridView s smart tag, click on the Edit Templates link and select the `BrochurePath` TemplateField s `EditItemTemplate` from the drop-down list. Add a RadioButtonList Web control to this template, setting its `ID` property to `BrochureOptions` and its `AutoPostBack` property to `True`. From the Properties window, click on the ellipses in the `Items` property, which will bring up the `ListItem` Collection Editor. Add the following three options with `Value` s 1, 2 and 3, respectively:

- Use current brochure
- Remove current brochure
- Upload new brochure

Set the first `ListItem` s `Selected` property to `True`.

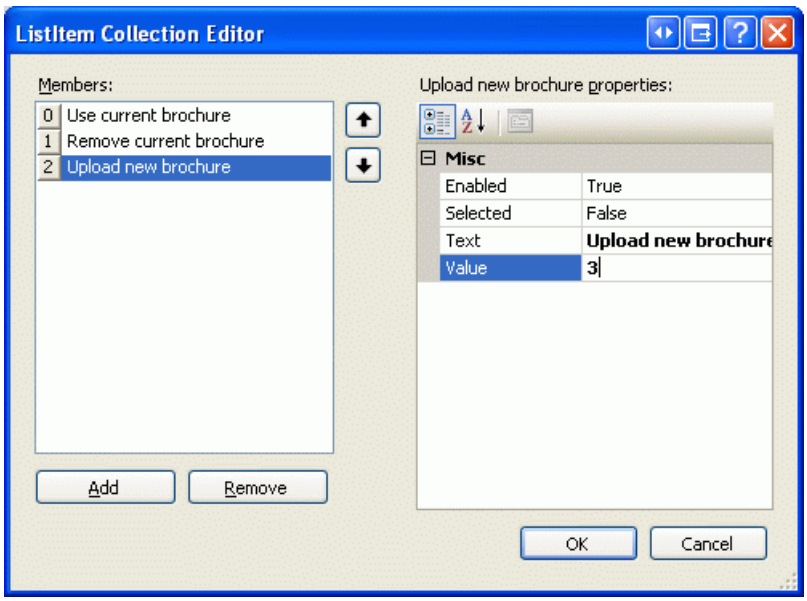


Figure 12: Add Three `ListItem` s to the RadioButtonList

Beneath the RadioButtonList, add a FileUpload control named `BrochureUpload`. Set its `Visible` property to `False`.

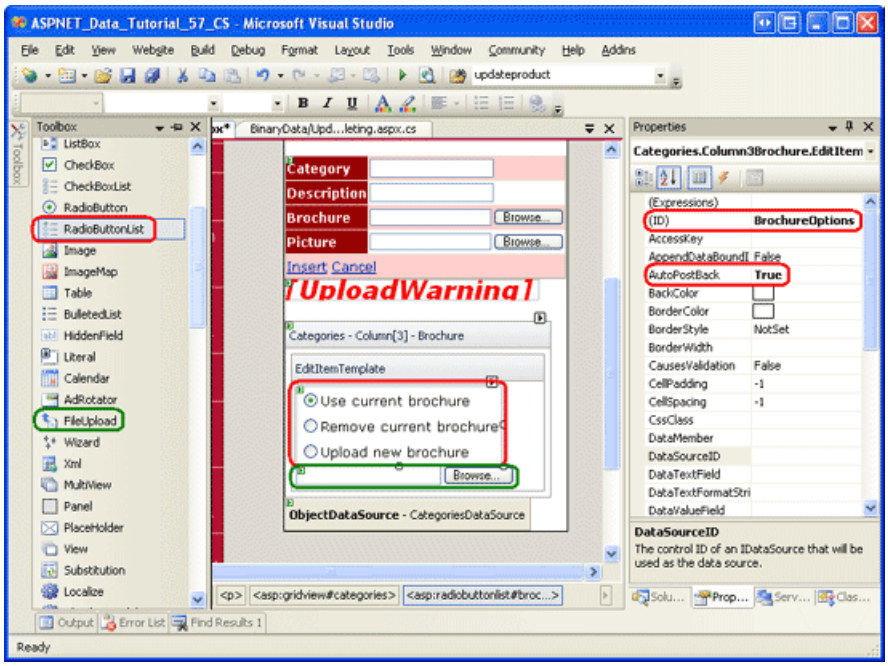


Figure 13: Add a RadioButtonList and FileUpload Control to the `EditItemTemplate` (Click to view full-size image)

This RadioButtonList provides the three options for the user. The idea is that the FileUpload control will be

displayed only if the last option, Upload new brochure, is selected. To accomplish this, create an event handler for the RadioButtonList s `SelectedIndexChanged` event and add the following code:

```
Protected Sub BrochureOptions_SelectedIndexChanged _
    (sender As Object, e As EventArgs)

    ' Get a reference to the RadioButtonList and its Parent
    Dim BrochureOptions As RadioButtonList = _
        CType(sender, RadioButtonList)
    Dim parent As Control = BrochureOptions.Parent
    ' Now use FindControl("controlID") to get a reference of the
    ' FileUpload control
    Dim BrochureUpload As FileUpload = _
        CType(parent.FindControl("BrochureUpload"), FileUpload)
    ' Only show BrochureUpload if SelectedValue = "3"
    BrochureUpload.Visible = (BrochureOptions.SelectedValue = "3")
End Sub
```

Since the RadioButtonList and FileUpload controls are within a template, we have to write a bit of code to programmatically access these controls. The `SelectedIndexChanged` event handler is passed a reference of the RadioButtonList in the `sender` input parameter. To get the FileUpload control, we need to get the RadioButtonList s parent control and use the `FindControl("controlID")` method from there. Once we have a reference to both the RadioButtonList and FileUpload controls, the FileUpload control s `Visible` property is set to `True` only if the RadioButtonList s `SelectedValue` equals 3, which is the `Value` for the Upload new brochure `ListItem`.

With this code in place, take a moment to test out the editing interface. Click on the Edit button for a row. Initially, the Use current brochure option should be selected. Changing the selected index causes a postback. If the third option is selected, the FileUpload control is displayed, otherwise it is hidden. Figure 14 shows the editing interface when the Edit button is first clicked; Figure 15 shows the interface after the Upload new brochure option is selected.

Figure 14: Initially, the Use current brochure Option is Selected (Click to view full-size image)

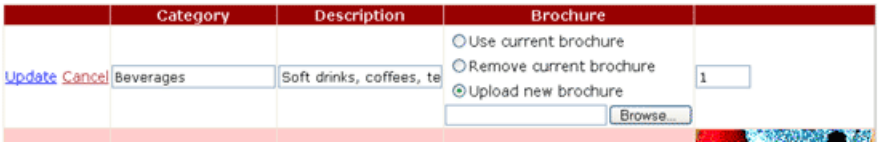


Figure 15: Choosing the Upload new brochure Option Displays the FileUpload Control (Click to view full-size image)

Saving the Brochure File and Updating the `BrochurePath` Column

When the GridView s Update button is clicked, its `RowUpdating` event fires. The ObjectDataSource s update command is invoked and then the GridView s `RowUpdated` event fires. Like with the deleting workflow, we need to create event handlers for both of these events. In the `RowUpdating` event handler, we need to determine what action to take based on the `SelectedValue` of the `BrochureOptions` RadioButtonList:

- If the `SelectedValue` is 1, we want to keep using the same `BrochurePath` setting. Therefore, we need to set the ObjectDataSource s `brochurePath` parameter to the existing `BrochurePath` value of the record being updated. The ObjectDataSource s `brochurePath` parameter can be set using `e.NewValues["brochurePath"] = value`.

- If the `SelectedValue` is 2, then we want to set the record s `BrochurePath` value to `NULL`. This can be accomplished by setting the ObjectDataSource s `brochurePath` parameter to `Nothing`, which results in a database `NULL` being used in the `UPDATE` statement. If there is an existing brochure file that is being removed, we need to delete the existing file. However, we only want to do this if the update completes without raising an exception.
- If the `SelectedValue` is 3, then we want to ensure that the user has uploaded a PDF file and then save it to the file system and update the record s `BrochurePath` column value. Moreover, if there is an existing brochure file that is being replaced, we need to delete the previous file. However, we only want to do this if the update completes without raising an exception.

The steps needed to be completed when the RadioButtonList s `SelectedValue` is 3 are virtually identical to those used by the DetailsView s `ItemInserting` event handler. This event handler is executed when a new category record is added from the DetailsView control we added in the previous tutorial. Therefore, it behooves us to refactor this functionality out into separate methods. Specifically, I moved out the common functionality into two methods:

- `ProcessBrochureUpload(FileUpload, out bool)` accepts as input a FileUpload control instance and an output Boolean value that specifies whether the delete or edit operation should proceed or if it should be cancelled due to some validation error. This method returns the path to the saved file or `null` if no file was saved.
- `DeleteRememberedBrochurePath` deletes the file specified by the path in the page variable `deletedCategorysPdfPath` if `deletedCategorysPdfPath` is not `null`.

The code for these two methods follows. Note the similarity between `ProcessBrochureUpload` and the DetailsView s `ItemInserting` event handler from the previous tutorial. In this tutorial I have updated the DetailsView s event handlers to use these new methods. Download the code associated with this tutorial to see the modifications to the DetailsView s event handlers.

```
Private Function ProcessBrochureUpload _
    (BrochureUpload As FileUpload, CancelOperation As Boolean) As String

    CancelOperation = False    ' by default, do not cancel operation
    If BrochureUpload.HasFile Then
        ' Make sure that a PDF has been uploaded
        If String.Compare(System.IO.Path.GetExtension(BrochureUpload.FileName), _
            ".pdf", True) <> 0 Then

            UploadWarning.Text = _
                "Only PDF documents may be used for a category's brochure."
            UploadWarning.Visible = True
            CancelOperation = True
            Return Nothing
        End If
        Const BrochureDirectory As String = "~/Brochures/"
        Dim brochurePath As String = BrochureDirectory + BrochureUpload.FileName
        Dim fileNameWithoutExtension As String = _
            System.IO.Path.GetFileNameWithoutExtension(BrochureUpload.FileName)
        Dim iteration As Integer = 1
        While System.IO.File.Exists(Server.MapPath(brochurePath))
            brochurePath = String.Concat(BrochureDirectory, _
                fileNameWithoutExtension, "-", iteration, ".pdf")
            iteration += 1
        End While
        ' Save the file to disk and set the value of the brochurePath parameter
        BrochureUpload.SaveAs(Server.MapPath(brochurePath))
        Return brochurePath
    Else
        ' No file uploaded
        Return Nothing
    End If
End Function
Private Sub DeleteRememberedBrochurePath()
    ' Is there a file to delete?
    If deletedCategorysPdfPath IsNot Nothing Then
        System.IO.File.Delete(Server.MapPath(deletedCategorysPdfPath))
    End If
End Sub
```

The GridView s `RowUpdating` and `RowUpdated` event handlers use the `ProcessBrochureUpload` and `DeleteRememberedBrochurePath` methods, as the following code shows:

```
Protected Sub Categories_RowUpdating _
    (sender As Object, e As GridViewUpdateEventArgs) _
    Handles Categories.RowUpdating

    ' Reference the RadioButtonList
    Dim BrochureOptions As RadioButtonList = _
        CType(Categories.Rows(e.RowIndex).FindControl("BrochureOptions"), _
            RadioButtonList)
    ' Get BrochurePath information about the record being updated
    Dim categoryID As Integer = Convert.ToInt32(e.Keys("CategoryID"))
    Dim categoryAPI As New CategoriesBLL()
    Dim categoriesData As Northwind.CategoriesDataTable = _
        categoryAPI.GetCategoryByCategoryID(categoryID)
    Dim category As Northwind.CategoriesRow = categoriesData(0)
    If BrochureOptions.SelectedValue = "1" Then
        ' Use current value for BrochurePath
        If category.IsBrochurePathNull() Then
            e.NewValues("brochurePath") = Nothing
        Else
            e.NewValues("brochurePath") = category.BrochurePath
        End If
    ElseIf BrochureOptions.SelectedValue = "2" Then
        ' Remove the current brochure (set it to NULL in the database)
        e.NewValues("brochurePath") = Nothing
    ElseIf BrochureOptions.SelectedValue = "3" Then
        ' Reference the BrochurePath FileUpload control
        Dim BrochureUpload As FileUpload = _
            CType(categories.Rows(e.RowIndex).FindControl("BrochureUpload"), _
                FileUpload)
        ' Process the BrochureUpload
        Dim cancelOperation As Boolean = False
        e.NewValues("brochurePath") = _
            ProcessBrochureUpload(BrochureUpload, cancelOperation)
        e.Cancel = cancelOperation
    Else
        ' Unknown value!
        Throw New ApplicationException( _
            String.Format("Invalid BrochureOptions value, {0}", _
                BrochureOptions.SelectedValue))
    End If
    If BrochureOptions.SelectedValue = "2" OrElse _
        BrochureOptions.SelectedValue = "3" Then

        ' "Remember" that we need to delete the old PDF file
        If (category.IsBrochurePathNull()) Then
            deletedCategorysPdfPath = Nothing
        Else
            deletedCategorysPdfPath = category.BrochurePath
        End If
    End If
End Sub
Protected Sub Categories_RowUpdated _
    (sender As Object, e As GridViewUpdatedEventArgs) _
    Handles Categories.RowUpdated

    ' If there were no problems and we updated the PDF file,
    ' then delete the existing one
    If e.Exception Is Nothing Then
        DeleteRememberedBrochurePath()
    End If
End Sub
```

Note how the `RowUpdating` event handler uses a series of conditional statements to perform the appropriate action based on the `BrochureOptions` RadioButtonList s `SelectedValue` property value.

With this code in place, you can edit a category and have it use its current brochure, use no brochure, or upload a

new one. Go ahead and try it out. Set breakpoints in the `RowUpdating` and `RowUpdated` event handlers to get a sense of the workflow.

Step 7: Uploading a New Picture

The `Picture` ImageField s editing interface renders as a textbox populated with the value from its `DataImageUrlField` property. During the editing workflow, the GridView passes a parameter to the ObjectDataSource with the parameter s name the value of the ImageField s `DataImageUrlField` property and the parameter s value the value entered into the textbox in the editing interface. This behavior is suitable when the image is saved as a file on the file system and the `DataImageUrlField` contains the full URL of the image. With such circumstances, the editing interface displays the image s URL in the textbox, which the user can change and have saved back to the database. Granted, this default interface doesn t allow the user to upload a new image, but it does let them change the URL of the image from the current value to another. For this tutorial, however, the ImageField s default editing interface does not suffice because the `Picture` binary data is being stored directly in the database and the `DataImageUrlField` property holds just the `CategoryID`.

To better understand what happens in our tutorial when a user edits a row with an ImageField, consider the following example: a user edits a row with `CategoryID` 10, causing the `Picture` ImageField to render as a textbox with the value 10. Imagine that the user changes the value in this textbox to 50 and clicks the Update button. A postback occurs and the GridView initially creates a parameter named `CategoryID` with the value 50. However, before the GridView sends this parameter (and the `CategoryName` and `Description` parameters), it adds in the values from the `DataKeys` collection. Therefore, it overwrites the `CategoryID` parameter with the current row s underlying `CategoryID` value, 10. In short, the ImageField s editing interface has no affect on the editing workflow for this tutorial because names of the ImageField s `DataImageUrlField` property and the grid s `DataKey` value are one in the same.

While the ImageField makes it easy to display an image based on database data, we don t want to provide a textbox in the editing interface. Rather, we want to offer a FileUpload control that the end user can use to change the category s picture. Unlike the `BrochurePath` value, for these tutorials we ve decided to require that each category must have a picture. Therefore, we don t need to let the user indicate that there is no associated picture the user may either upload a new picture or leave the current picture as-is.

To customize the ImageField s editing interface, we need to convert it into a TemplateField. From the GridView s smart tag, click on the Edit Columns link, select the ImageField, and click the Convert this field into a TemplateField link.

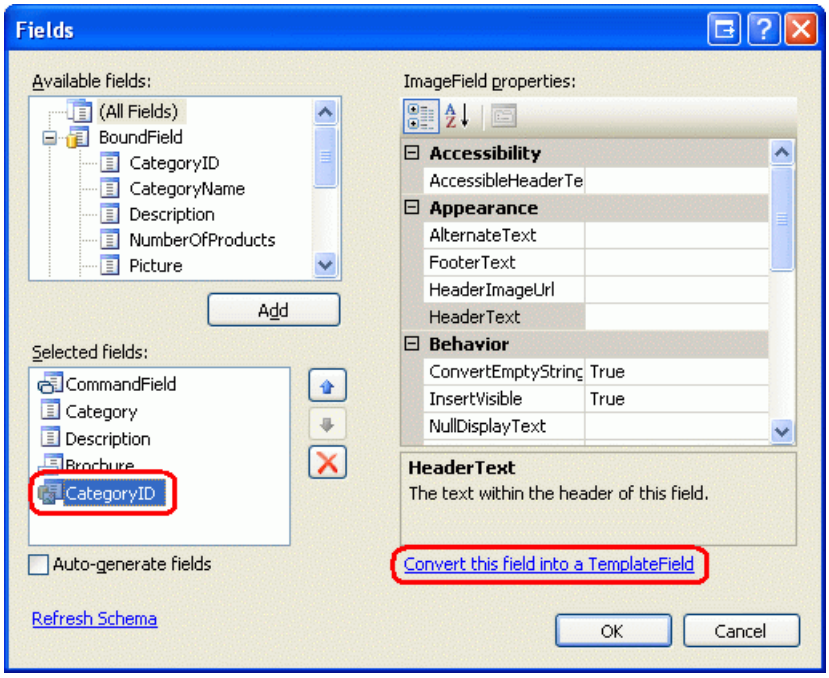


Figure 16: Convert the ImageField Into a TemplateField

Converting the ImageField into a TemplateField in this manner generates a TemplateField with two templates. As the following declarative syntax shows, the `ItemTemplate` contains an Image Web control whose `ImageUrl` property is assigned using databinding syntax based on the ImageField s `DataImageUrlField` and `DataImageUrlFormatString` properties. The `EditItemTemplate` contains a TextBox whose `Text` property is bound to the value specified by the `DataImageUrlField` property.

```
<asp:TemplateField>
    <EditItemTemplate>
        <asp:TextBox ID="TextBox1" runat="server"
            Text='<%# Eval("CategoryID") %>'></asp:TextBox>
    </EditItemTemplate>
    <ItemTemplate>
        <asp:Image ID="Image1" runat="server"
            ImageUrl='<%# Eval("CategoryID",
                "DisplayCategoryPicture.aspx?CategoryID={0}") %>' />
    </ItemTemplate>
</asp:TemplateField>
```

We need to update the `EditItemTemplate` to use a FileUpload control. From the GridView s smart tag click on the Edit Templates link and then select the `Picture` TemplateField s `EditItemTemplate` from the drop-down list. In the template you should see a TextBox remove this. Next, drag a FileUpload control from the Toolbox into the template, setting its `ID` to `PictureUpload`. Also add the text To change the category s picture, specify a new picture. To keep the category s picture the same, leave the field empty to the template, as well.

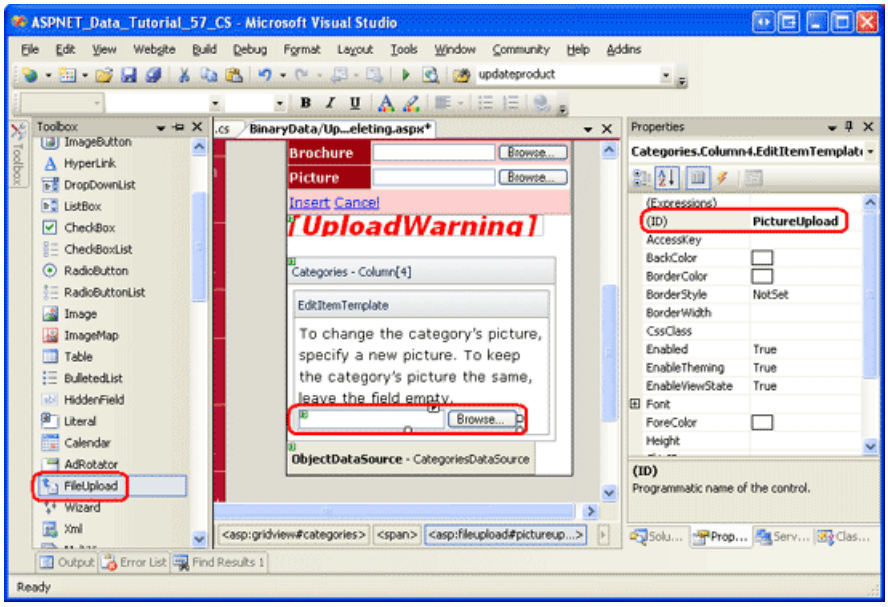


Figure 17: Add a FileUpload Control to the `EditItemTemplate` (Click to view full-size image)

After customizing the editing interface, view your progress in a browser. When viewing a row in read-only mode, the category s image is shown as it was before, but clicking on the Edit button renders the picture column as text with a FileUpload control.

Figure 18: The Editing Interface Includes a FileUpload Control (Click to view full-size image)

Recall that the ObjectDataSource is configured to call the `CategoriesBLL` class s `UpdateCategory` method that accepts as input the binary data for the picture as a `Byte` array. If this array is `Nothing`, however, the alternate `UpdateCategory` overload is called, which issues the `UPDATE` SQL statement that does not modify the `Picture` column, thereby leaving the category s current picture intact. Therefore, in the GridView s `RowUpdating` event handler we need to programmatically reference the `PictureUpload` FileUpload control and determine if a file was uploaded. If one was not uploaded, then we do *not* want to specify a value for the `picture` parameter. On the other hand, if a file was uploaded in the `PictureUpload` FileUpload control, we want to ensure that it is a JPG file. If it is, then we can send its binary contents to the ObjectDataSource through the `picture` parameter.

Like with the code used in Step 6, much of the code needed here already exists in the DetailsView s `ItemInserting` event handler. Therefore, I ve refactored the common functionality into a new method, `ValidPictureUpload`, and updated the `ItemInserting` event handler to use this method.

Add the following code to the start of the GridView s `RowUpdating` event handler. It s important that this code come before the code that saves the brochure file since we don t want to save the brochure to the web server s file system if an invalid picture file is uploaded.

```
' Reference the PictureUpload FileUpload
Dim PictureUpload As FileUpload = _
    CType(categories.Rows(e.RowIndex).FindControl("PictureUpload"), _
        FileUpload)
If PictureUpload.HasFile Then
    ' Make sure the picture upload is valid
    If ValidPictureUpload(PictureUpload) Then
        e.NewValues("picture") = PictureUpload.FileBytes
    Else
        ' Invalid file upload, cancel update and exit event handler
        e.Cancel = True
        Exit Sub
    End If
End If
```

The `ValidPictureUpload(FileUpload)` method takes in a FileUpload control as its sole input parameter and checks the uploaded file s extension to ensure that the uploaded file is a JPG; it is only called if a picture file is uploaded. If no file is uploaded, then the picture parameter is not set, and therefore uses its default value of `Nothing`. If a picture was uploaded and `ValidPictureUpload` returns `True`, the `picture` parameter is assigned the binary data of the uploaded image; if the method returns `False`, the update workflow is cancelled and the event handler exited.

The `ValidPictureUpload(FileUpload)` method code, which was refactored from the DetailsView s `ItemInserting` event handler, follows:

```
Private Function ValidPictureUpload(ByVal PictureUpload As FileUpload) As Boolean
    ' Make sure that a JPG has been uploaded
    If String.Compare(System.IO.Path.GetExtension(PictureUpload.FileName), _
        ".jpg", True) <> 0 AndAlso _
        String.Compare(System.IO.Path.GetExtension(PictureUpload.FileName), _
        ".jpeg", True) <> 0 Then

        UploadWarning.Text = _
            "Only JPG documents may be used for a category's picture."
        UploadWarning.Visible = True
        Return False
    Else
        Return True
    End If
End Function
```

Step 8: Replacing the Original Categories Pictures with JPGs

Recall that the original eight categories pictures are bitmap files wrapped in an OLE header. Now that we have added the capability to edit an existing record s picture, take a moment to replace these bitmaps with JPGs. If you want to continue to use the current category pictures, you can convert them to JPGs by performing the following steps:

1. Save the bitmap images to your hard drive. Visit the `UpdatingAndDeleting.aspx` page in your browser and for each of the first eight categories, right-click on the image and choose to save the picture.
2. Open the image in your image editor of choice. You can use Microsoft Paint, for example.
3. Save the bitmap as a JPG image.
4. Update the category s picture through the editing interface, using the JPG file.

After editing a category and uploading the JPG image, the image will not render in the browser because the `DisplayCategoryPicture.aspx` page is stripping the first 78 bytes from the pictures of the first eight categories. Fix this by removing the code that performs the OLE header stripping. After doing this, the `DisplayCategoryPicture.aspx``Page_Load` event handler should have just the following code:

```
Protected Sub Page_Load(sender As Object, e As EventArgs) Handles Me.Load
    Dim categoryID As Integer = _
        Convert.ToInt32(Request.QueryString("CategoryID"))
    ' Get information about the specified category
    Dim categoryAPI As New CategoriesBLL()
    Dim categories As Northwind.CategoriesDataTable = _
        categoryAPI.GetCategoryWithBinaryDataByCategoryID(categoryID)
    Dim category As Northwind.CategoriesRow = categories(0)
    ' For new categories, images are JPGs...
    ' Output HTTP headers providing information about the binary data
    Response.ContentType = "image/jpeg"
    ' Output the binary data
    Response.BinaryWrite(category.Picture)
End Sub
```

NOTE

The `UpdatingAndDeleting.aspx` page s inserting and editing interfaces could use a bit more work. The `CategoryName` and `Description` BoundFields in the DetailsView and GridView should be converted into TemplateFields. Since `CategoryName` does not allow `NULL` values, a RequiredFieldValidator should be added. And the `Description` TextBox should probably be converted into a multi-line TextBox. I leave these finishing touches as an exercise for you.

Summary

This tutorial completes our look at working with binary data. In this tutorial and the previous three, we saw how binary data can be stored on the file system or directly within the database. A user provides binary data to the system by selecting a file from their hard drive and uploading it to the web server, where it can be stored on the file system or inserted into the database. ASP.NET 2.0 includes a FileUpload control that makes providing such an interface as easy as drag and drop. However, as noted in the Uploading Files tutorial, the FileUpload control is only well-suited for relatively small file uploads, ideally not exceeding a megabyte. We also explored how to associate uploaded data with the underlying data model, as well as how to edit and delete the binary data from existing records.

Our next set of tutorials explores various caching techniques. Caching provides a means to improve an application s overall performance by taking the results from expensive operations and storing them in a location that can be more quickly accessed.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Caching Data

2/20/2020 • 2 minutes to read • Edit Online

This tutorial set explains caching, which can improve site performance, and how to apply caching to the Presentation Layer using the ObjectDataSource control.

- Caching Data with the ObjectDataSource (C#)
- Caching Data in the Architecture (C#)
- Caching Data at Application Startup (C#)
- Using SQL Cache Dependencies (C#)
- Caching Data with the ObjectDataSource (VB)
- Caching Data in the Architecture (VB)
- Caching Data at Application Startup (VB)
- Using SQL Cache Dependencies (VB)

Caching Data with the ObjectDataSource (C#)

2/20/2020 • 20 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

Caching can mean the difference between a slow and a fast Web application. This tutorial is the first of four that take a detailed look at caching in ASP.NET. Learn the key concepts of caching and how to apply caching to the Presentation Layer through the ObjectDataSource control.

Introduction

In computer science, *caching* is the process of taking data or information that is expensive to obtain and storing a copy of it in a location that is quicker to access. For data-driven applications, large and complex queries commonly consume the majority of the application s execution time. Such an application s performance, then, can often be improved by storing the results of expensive database queries in the application s memory.

ASP.NET 2.0 offers a variety of caching options. An entire web page or User Control s rendered markup can be cached through *output caching*. The ObjectDataSource and SqlDataSource controls provide caching capabilities as well, thereby allowing data to be cached at the control level. And ASP.NET s *data cache* provides a rich caching API that enables page developers to programmatically cache objects. In this tutorial and the next three we'll examine using the ObjectDataSource s caching features as well as the data cache. We'll also explore how to cache application-wide data at startup and how to keep cached data fresh through the use of SQL cache dependencies. These tutorials do not explore output caching. For a detailed look at output caching, see Output Caching in ASP.NET 2.0.

Caching can be applied at any place in the architecture, from the Data Access Layer up through the Presentation Layer. In this tutorial we'll look at applying caching to the Presentation Layer through the ObjectDataSource control. In the next tutorial we'll examine caching data at the Business Logic Layer.

Key Caching Concepts

Caching can greatly improve an application s overall performance and scalability by taking data that is expensive to generate and storing a copy of it in a location that can be more efficiently accessed. Since the cache holds just a copy of the actual, underlying data, it can become outdated, or *stale*, if the underlying data changes. To combat this, a page developer can indicate criteria by which the cache item will be *evicted* from the cache, using either:

- **Time-based criteria** an item may be added to the cache for an absolute or sliding duration. For example, a page developer may indicate a duration of, say, 60 seconds. With an absolute duration, the cached item is evicted 60 seconds after it was added to cache, regardless of how frequently it was accessed. With a sliding duration, the cached item is evicted 60 seconds after the last access.
- **Dependency-based criteria** a dependency can be associated with an item when added to the cache. When the item s dependency changes it is evicted from the cache. The dependency may be a file, another cache item, or a combination of the two. ASP.NET 2.0 also allows SQL cache dependencies, which enable developers to add an item to the cache and have it evicted when the underlying database data changes. We will examine SQL cache dependencies in the upcoming Using SQL Cache Dependencies tutorial.

Regardless of the eviction criteria specified, an item in the cache may be *scavenged* before the time-based or dependency-based criteria has been met. If the cache has reached its capacity, existing items must be removed before new ones can be added. Consequently, when programmatically working with cached data it s vital that you

always assume that the cached data may not be present. We'll look at the pattern to use when accessing data from the cache programmatically in our next tutorial, *Caching Data in the Architecture*.

Caching provides an economical means for squeezing more performance from an application. As Steven Smith articulates in his article ASP.NET Caching: Techniques and Best Practices:

Caching can be a good way to get good enough performance without requiring a lot of time and analysis. Memory is cheap, so if you can get the performance you need by caching the output for 30 seconds instead of spending a day or a week trying to optimize your code or database, do the caching solution (assuming 30-second old data is ok) and move on. Eventually, poor design will probably catch up to you, so of course you should try to design your applications correctly. But if you just need to get good enough performance today, caching can be an excellent [approach], buying you time to refactor your application at a later date when you have the time to do so.

While caching can provide appreciable performance enhancements, it is not applicable in all situations, such as with applications that use real-time, frequently-updating data, or where even shortly-lived stale data is unacceptable. But for the majority of applications, caching should be used. For more background on caching in ASP.NET 2.0, refer to the Caching for Performance section of the ASP.NET 2.0 QuickStart Tutorials.

Step 1: Creating the Caching Web Pages

Before we start our exploration of the ObjectDataSource s caching features, let s first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial and the next three. Start by adding a new folder named `Caching`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `ObjectDataSource.aspx`
- `FromTheArchitecture.aspx`
- `AtApplicationStartup.aspx`
- `SqlCacheDependencies.aspx`

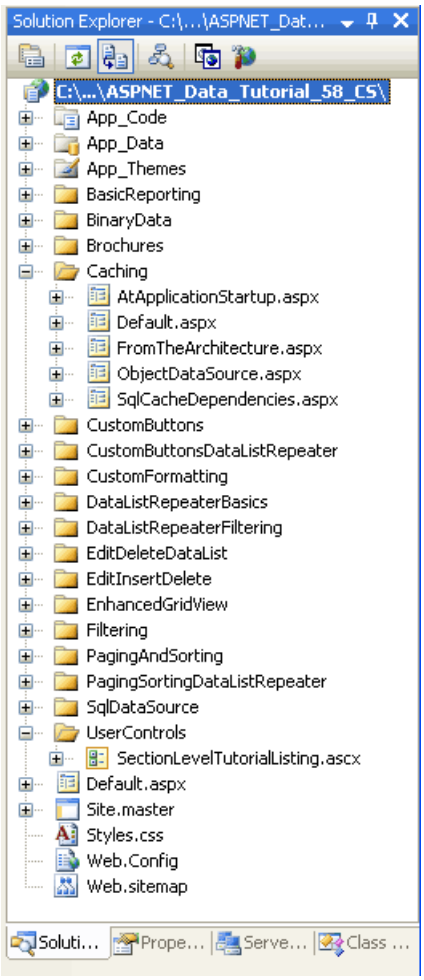


Figure 1: Add the ASP.NET Pages for the Caching-Related Tutorials

Like in the other folders, `Default.aspx` in the `Caching` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

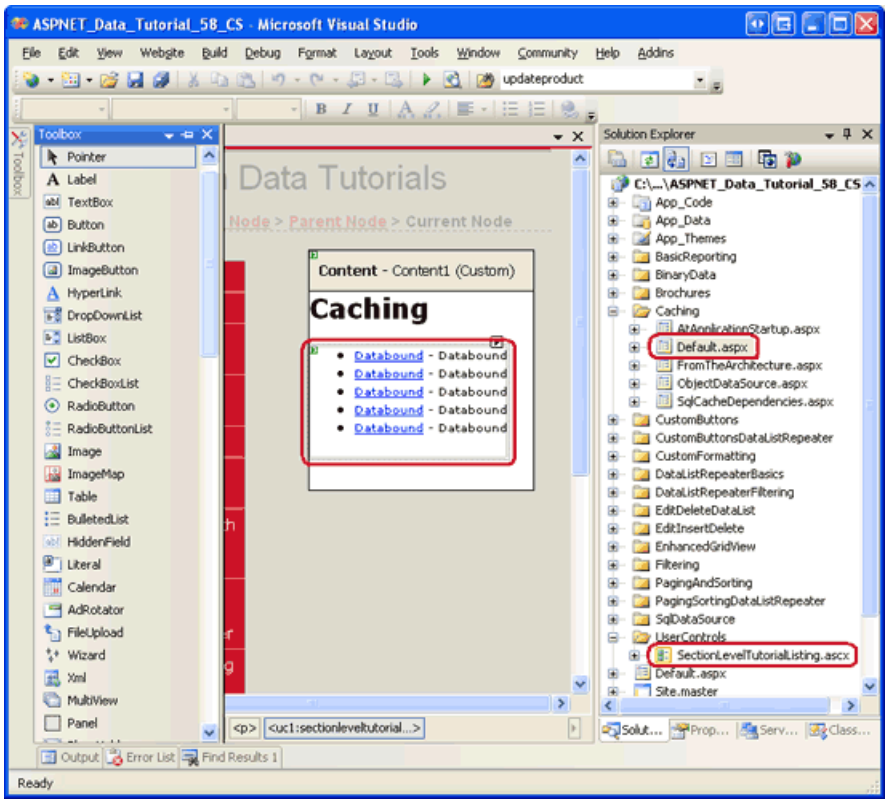


Figure 2: Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-

size image)

Lastly, add these pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Working with Binary Data `<siteMapNode>`:

```
<siteMapNode title="Caching" url="~/Caching/Default.aspx"
    description="Learn how to use the caching features of ASP.NET 2.0.">
    <siteMapNode url="~/Caching/ObjectDataSource.aspx"
        title="ObjectDataSource Caching"
        description="Explore how to cache data directly from the
            ObjectDataSource control." />
    <siteMapNode url="~/Caching/FromTheArchitecture.aspx"
        title="Caching in the Architecture"
        description="See how to cache data from within the
            architecture." />
    <siteMapNode url="~/Caching/AtApplicationStartup.aspx"
        title="Caching Data at Application Startup"
        description="Learn how to cache expensive or infrequently-changing
            queries at the start of the application." />
    <siteMapNode url="~/Caching/SqlCacheDependencies.aspx"
        title="Using SQL Cache Dependencies"
        description="Examine how to have data automatically expire from the
            cache when its underlying database data is modified." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the caching tutorials.

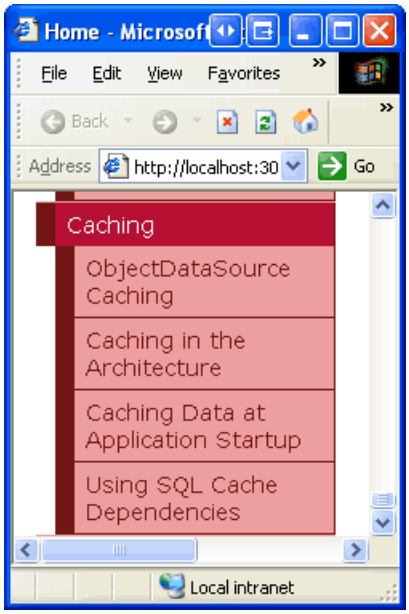


Figure 3: The Site Map Now Includes Entries for the Caching Tutorials

Step 2: Displaying a List of Products in a Web Page

This tutorial explores how to use the ObjectDataSource control s built-in caching features. Before we can look at these features, though, we first need a page to work from. Let s create a web page that uses a GridView to list product information retrieved by an ObjectDataSource from the `ProductsBLL` class.

Start by opening the `ObjectDataSource.aspx` page in the `Caching` folder. Drag a GridView from the Toolbox onto the Designer, set its `ID` property to `Products`, and, from its smart tag, choose to bind it to a new ObjectDataSource control named `ProductsDataSource`. Configure the ObjectDataSource to work with the `ProductsBLL` class.

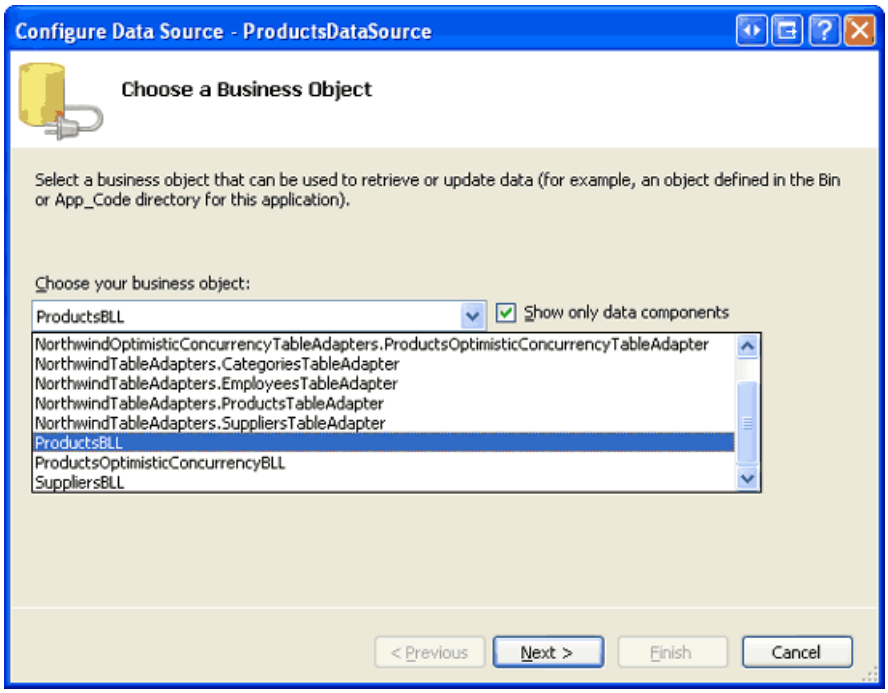


Figure 4: Configure the ObjectDataSource to Use the `ProductsBLL` Class (Click to view full-size image)

For this page, let s create an editable GridView so that we can examine what happens when data cached in the ObjectDataSource is modified through the GridView s interface. Leave the drop-down list in the SELECT tab set to its default, `GetProducts()`, but change the selected item in the UPDATE tab to the `UpdateProduct` overload that accepts `productName`, `unitPrice`, and `productID` as its input parameters.

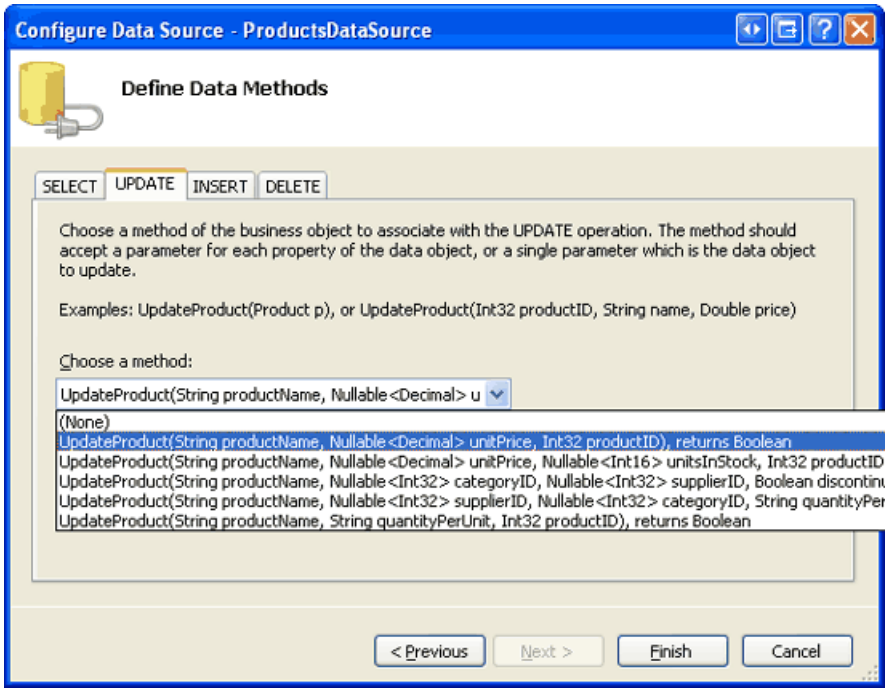


Figure 5: Set the UPDATE Tab s Drop-Down List to the Appropriate `UpdateProduct` Overload (Click to view full-size image)

Finally, set the drop-down lists in the INSERT and DELETE tabs to (None) and click Finish. Upon completing the Configure Data Source wizard, Visual Studio sets the ObjectDataSource s `OldValuesParameterFormatString` property to `original_{0}`. As discussed in the An Overview of Inserting, Updating, and Deleting Data tutorial, this property needs to be removed from the declarative syntax or set back to its default value, `{0}`, in order for our update workflow to proceed without error.

Furthermore, at the completion of the wizard Visual Studio adds a field to the GridView for each of the product data fields. Remove all but the `ProductName`, `CategoryName`, and `UnitPrice` BoundFields. Next, update the

`HeaderText` properties of each of these BoundFields to Product, Category, and Price, respectively. Since the `ProductName` field is required, convert the BoundField into a TemplateField and add a RequiredFieldValidator to the `EditItemTemplate`. Similarly, convert the `UnitPrice` BoundField into a TemplateField and add a CompareValidator to ensure that the value entered by the user is a valid currency value that s greater than or equal to zero. In addition to these modifications, feel free to perform any aesthetic changes, such as right-aligning the `UnitPrice` value, or specifying the formatting for the `UnitPrice` text in its read-only and editing interfaces.

Make the GridView editable by checking the Enable Editing checkbox in the GridView s smart tag. Also check the Enable Paging and Enable Sorting checkboxes.

> **NOTE**
>
> Need a review of how to customize the GridView s editing interface? If so, refer back to the Customizing the Data Modification Interface tutorial.

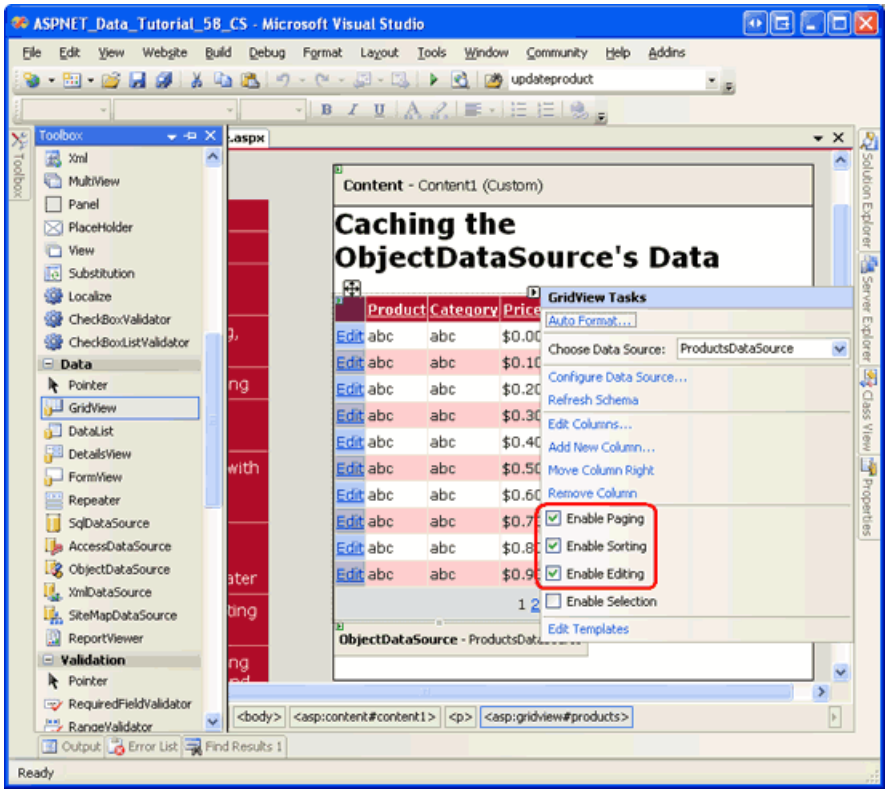


Figure 6: Enable GridView Support for Editing, Sorting, and Paging (Click to view full-size image)

After making these GridView modifications, the GridView and ObjectDataSource s declarative markup should look similar to the following:

```
<asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
    AllowPaging="True" AllowSorting="True">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
            <EditItemTemplate>
                <asp:TextBox ID="ProductName" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:TextBox>
                <asp:RequiredFieldValidator
                    ID="RequiredFieldValidator1" Display="Dynamic"
                    ControlToValidate="ProductName" SetFocusOnError="True"
                    ErrorMessage="You must provide a name for the product."
                    runat="server">*</asp:RequiredFieldValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:TemplateField HeaderText="Price" SortExpression="UnitPrice">
            <EditItemTemplate>
                $<asp:TextBox ID="UnitPrice" runat="server" Columns="8"
                    Text='<%# Bind("UnitPrice", "{0:N2}") %>'></asp:TextBox>
                <asp:CompareValidator ID="CompareValidator1"
                    ControlToValidate="UnitPrice" Display="Dynamic"
                    ErrorMessage="You must enter a valid currency value with no
                        currency symbols. Also, the value must be greater than
                        or equal to zero."
                    Operator="GreaterThanEqual" SetFocusOnError="True"
                    Type="Currency" runat="server"
                    ValueToCompare="0">*</asp:CompareValidator>
            </EditItemTemplate>
            <ItemStyle HorizontalAlign="Right" />
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("UnitPrice", "{0:c}") %>' />
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="{0}" SelectMethod="GetProducts"
    TypeName="ProductsBLL" UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

As Figure 7 shows, the editable GridView lists the name, category, and price of each of the products in the database. Take a moment to test out the page s functionality sort the results, page through them, and edit a record.

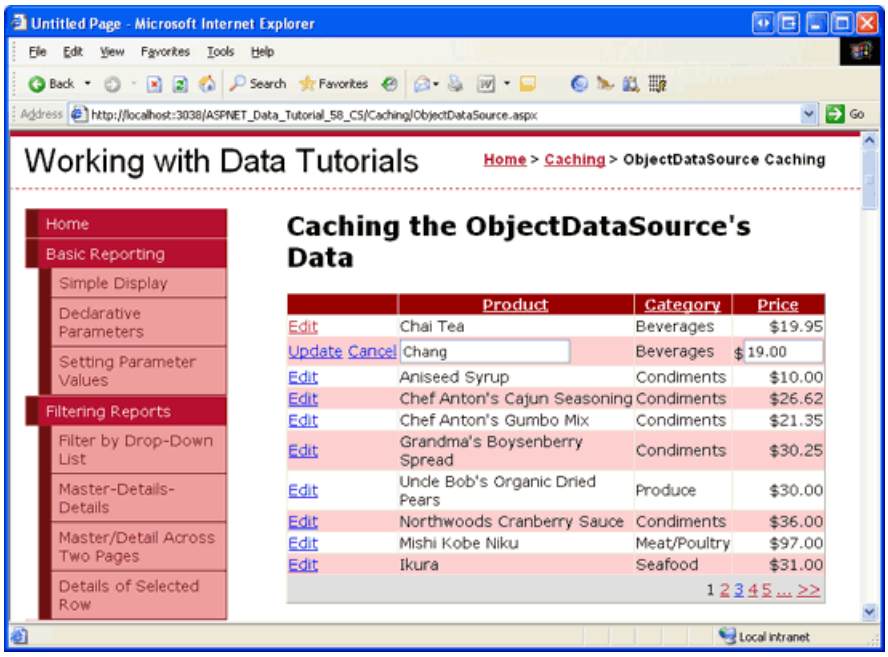


Figure 7: Each Product s Name, Category, and Price is Listed in a Sortable, Pageable, Editable GridView (Click to view full-size image)

Step 3: Examining When the ObjectDataSource is Requesting Data

The `Products` GridView retrieves its data to display by invoking the `Select` method of the `ProductsDataSource` ObjectDataSource. This ObjectDataSource creates an instance of the Business Logic Layer s `ProductsBLL` class and calls its `GetProducts()` method, which in turn calls the Data Access Layer s `ProductsTableAdapter` s `GetProducts()` method. The DAL method connects to the Northwind database and issues the configured `SELECT` query. This data is then returned to the DAL, which packages it up in a `NorthwindDataTable`. The DataTable object is returned to the BLL, which returns it to the ObjectDataSource, which returns it to the GridView. The GridView then creates a `GridViewRow` object for each `DataRow` in the DataTable, and each `GridViewRow` is eventually rendered into the HTML that is returned to the client and displayed on the visitor s browser.

This sequence of events happens each and every time the GridView needs to bind to its underlying data. That happens when the page is first visited, when moving from one page of data to another, when sorting the GridView, or when modifying the GridView s data through its built-in editing or deleting interfaces. If the GridView s view state is disabled, the GridView will be rebound on each and every postback as well. The GridView can also be explicitly rebound to its data by calling its `DataBind()` method.

To fully appreciate the frequency with which the data is retrieved from the database, let s display a message indicating when the data is being re-retrieved. Add a Label Web control above the GridView named `ODSEvents`. Clear out its `Text` property and set its `EnableViewState` property to `false`. Underneath the Label, add a Button Web control and set its `Text` property to Postback .

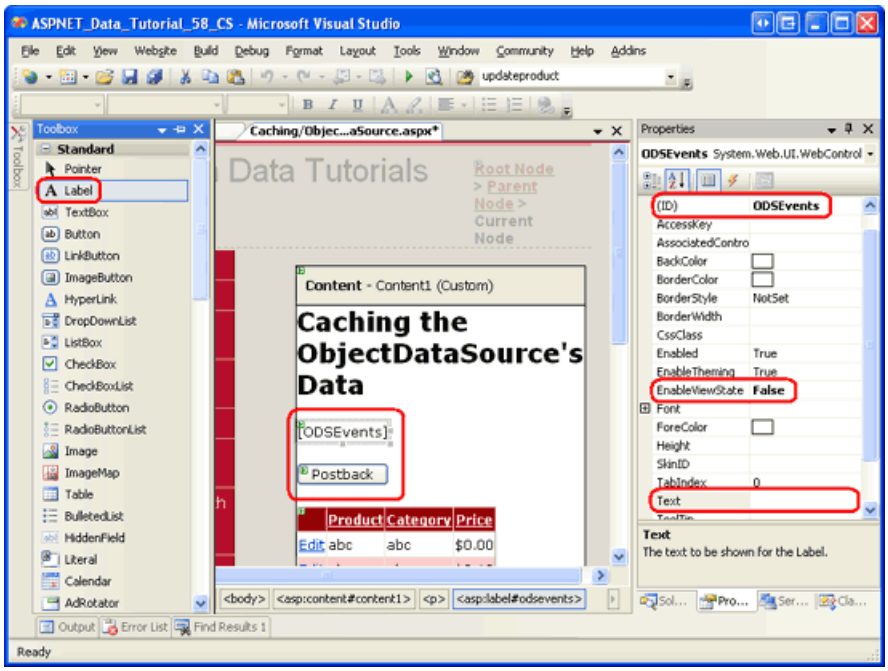


Figure 8: Add a Label and Button to the Page Above the GridView (Click to view full-size image)

During the data access workflow, the ObjectDataSource s `Selecting` event fires before the underlying object is created and its configured method invoked. Create an event handler for this event and add the following code:

```
protected void ProductsDataSource_Selecting(object sender,
    ObjectDataSourceSelectingEventArgs e)
{
    ODSEvents.Text = "-- Selecting event fired";
}
```

Each time the ObjectDataSource makes a request to the architecture for data, the Label will display the text Selecting event fired .

Visit this page in a browser. When the page is first visited, the text Selecting event fired is shown. Click the Postback button and note that the text disappears (assuming that the GridView s `EnableViewState` property is set to `true`, the default). This is because, on postback, the GridView is reconstructed from its view state and therefore doesn t turn to the ObjectDataSource for its data. Sorting, paging, or editing the data, however, causes the GridView to rebind to its data source, and therefore the Selecting event fired text reappears.

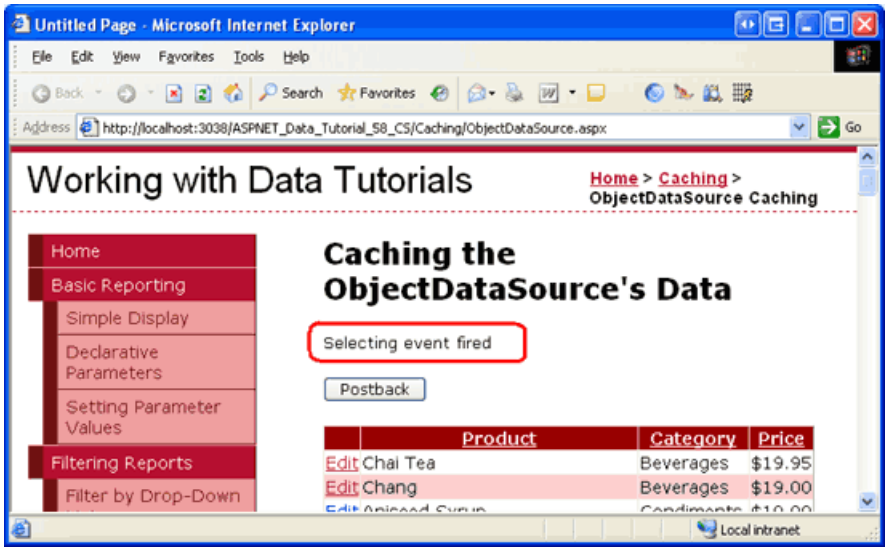


Figure 9: Whenever the GridView is Rebound to its Data Source, Selecting event fired is Displayed (Click to view full-size image)

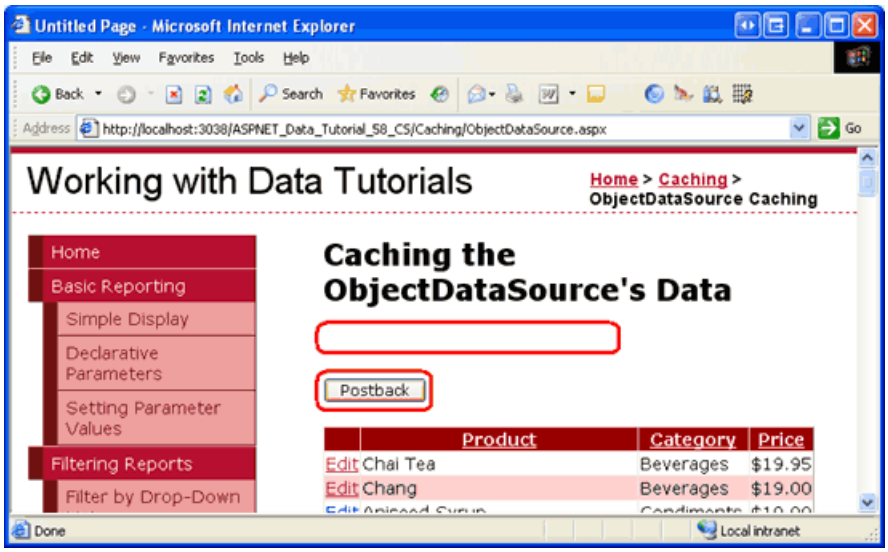


Figure 10: Clicking the Postback Button Causes the GridView to be Reconstructed from its View State (Click to view full-size image)

It may seem wasteful to retrieve the database data each time the data is paged through or sorted. After all, since we re using default paging, the ObjectDataSource has retrieved all of the records when displaying the first page. Even if the GridView does not provide sorting and paging support, the data must be retrieved from the database each time the page is first visited by any user (and on every postback, if view state is disabled). But if the GridView is showing the same data to all users, these extra database requests are superfluous. Why not cache the results returned from the `GetProducts()` method and bind the GridView to those cached results?

Step 4: Caching the Data Using the ObjectDataSource

By simply setting a few properties, the ObjectDataSource can be configured to automatically cache its retrieved data in the ASP.NET data cache. The following list summarizes the cache-related properties of the ObjectDataSource:

- EnableCaching must be set to `true` to enable caching. The default is `false`.
- CacheDuration the amount of time, in seconds, that the data is cached. The default is 0. The ObjectDataSource will only cache data if `EnableCaching` is `true` and `CacheDuration` is set to a value greater than zero.
- CacheExpirationPolicy can be set to `Absolute` or `Sliding`. If `Absolute`, the ObjectDataSource caches its retrieved data for `CacheDuration` seconds; if `Sliding`, the data expires only after it has not been accessed for `CacheDuration` seconds. The default is `Absolute`.
- CacheKeyDependency use this property to associate the ObjectDataSource s cache entries with an existing cache dependency. The ObjectDataSource s data entries can be prematurely evicted from the cache by expiring its associated `CacheKeyDependency`. This property is most commonly used to associate a SQL cache dependency with the ObjectDataSource s cache, a topic we'll explore in the future Using SQL Cache Dependencies tutorial.

Let s configure the `ProductsDataSource` ObjectDataSource to cache its data for 30 seconds on an absolute scale. Set the ObjectDataSource s `EnableCaching` property to `true` and its `CacheDuration` property to 30. Leave the `CacheExpirationPolicy` property set to its default, `Absolute`.

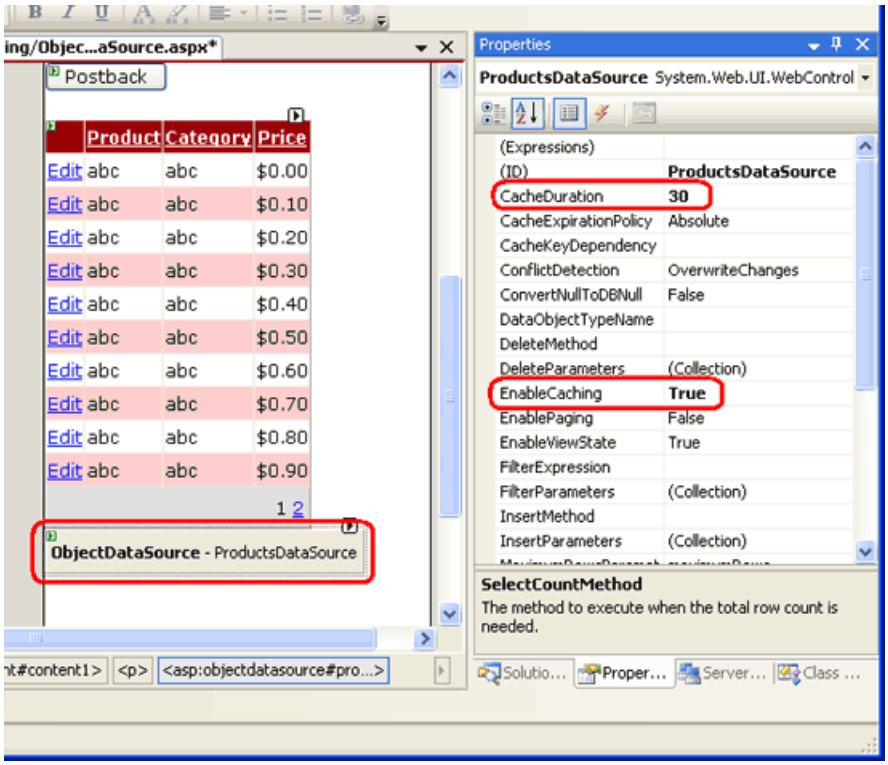


Figure 11: Configure the ObjectDataSource to Cache its Data for 30 Seconds (Click to view full-size image)

Save your changes and revisit this page in a browser. The Selecting event fired text will appear when you first visit the page, as initially the data is not in the cache. But subsequent postbacks triggered by clicking the Postback button, sorting, paging, or clicking the Edit or Cancel buttons does *not* redisplay the Selecting event fired text. This is because the `Selecting` event only fires when the ObjectDataSource gets its data from its underlying object; the `Selecting` event does not fire if the data is pulled from the data cache.

After 30 seconds, the data will be evicted from the cache. The data will also be evicted from the cache if the ObjectDataSource s `Insert`, `Update`, or `Delete` methods are invoked. Consequently, after 30 seconds have passed or the Update button has been clicked, sorting, paging, or clicking the Edit or Cancel buttons will cause the ObjectDataSource to get its data from its underlying object, displaying the Selecting event fired text when the `Selecting` event fires. These returned results are placed back into the data cache.

> **NOTE**
>
> If you see the Selecting event fired text frequently, even when you expect the ObjectDataSource to be working with cached data, it may be due to memory constraints. If there is not enough free memory, the data added to the cache by the ObjectDataSource may have been scavenged. If the ObjectDataSource doesn t appear to be correctly caching the data or only caches the data sporadically, close some applications to free memory and try again.

Figure 12 illustrates the ObjectDataSource s caching workflow. When the Selecting event fired text appears on your screen, it is because the data was not in the cache and had to be retrieved from the underlying object. When this text is missing, however, it s because the data was available from the cache. When the data is returned from the cache there s no call to the underlying object and, therefore, no database query executed.

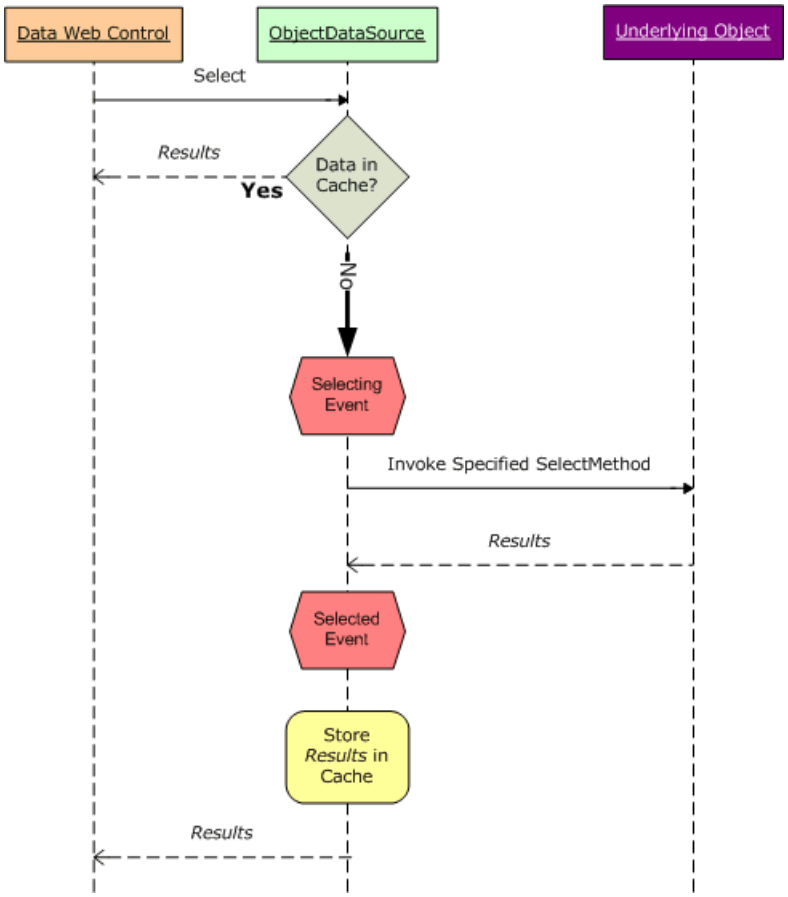


Figure 12: The ObjectDataSource Stores and Retrieves its Data from the Data Cache

Each ASP.NET application has its own data cache instance that s shared across all pages and visitors. That means that the data stored in the data cache by the ObjectDataSource is likewise shared across all users who visit the page. To verify this, open the `ObjectDataSource.aspx` page in a browser. When first visiting the page, the Selecting event fired text will appear (assuming that the data added to the cache by previous tests has, by now, been evicted). Open a second browser instance and copy and paste the URL from the first browser instance to the second. In the second browser instance, the Selecting event fired text is not shown because it s using the same cached data as the first.

When inserting its retrieved data into the cache, the ObjectDataSource uses a cache key value that includes: the `CacheDuration` and `CacheExpirationPolicy` property values; the type of the underlying business object being used by the ObjectDataSource, which is specified via the `TypeName` property (`ProductsBLL` , in this example); the value of the `SelectMethod` property and the name and values of the parameters in the `SelectParameters` collection; and the values of its `StartRowIndex` and `MaximumRows` properties, which are used when implementing custom paging.

Crafting the cache key value as a combination of these properties ensures a unique cache entry as these values change. For example, in past tutorials we ve looked at using the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` , which returns all products for a specified category. One user might come to the page and view beverages, which has a `CategoryID` of 1. If the ObjectDataSource cached its results without regard for the `SelectParameters` values, when another user came to the page to view condiments while the beverages products were in the cache, they d see the cached beverage products rather than condiments. By varying the cache key by these properties, which include the values of the `SelectParameters` , the ObjectDataSource maintains a separate cache entry for beverages and condiments.

Stale Data Concerns

The ObjectDataSource automatically evicts its items from the cache when any one of its `Insert` , `Update` , or `Delete` methods is invoked. This helps protect against stale data by clearing out the cache entries when the data is modified through the page. However, it is possible for an ObjectDataSource using caching to still display stale

data. In the simplest case, it can be due to the data changing directly within the database. Perhaps a database administrator just ran a script that modifies some of the records in the database.

This scenario could also unfold in a more subtle way. While the ObjectDataSource evicts its items from the cache when one of its data modification methods is called, the cached items removed are for the ObjectDataSource s particular combination of property values (`CacheDuration` , `TypeName` , `SelectMethod` , and so on). If you have two ObjectDataSources that use different `SelectMethods` or `SelectParameters` , but still can update the same data, then one ObjectDataSource may update a row and invalidate its own cache entries, but the corresponding row for the second ObjectDataSource will still be served from the cached. I encourage you to create pages to exhibit this functionality. Create a page that displays an editable GridView that pulls its data from an ObjectDataSource that uses caching and is configured to get data from the `ProductsBLL` class s `GetProducts()` method. Add another editable GridView and ObjectDataSource to this page (or another one), but for this second ObjectDataSource have it use the `GetProductsByCategoryID(categoryID)` method. Since the two ObjectDataSources `SelectMethod` properties differ, they ll each have their own cached values. If you edit a product in one grid, the next time you bind the data back to the other grid (by paging, sorting, and so forth), it will still serve the old, cached data and not reflect the change that was made from the other grid.

In short, only use time-based expiries if you are willing to have the potential of stale data, and use shorter expiries for scenarios where the freshness of data is important. If stale data is not acceptable, either forgo caching or use SQL cache dependencies (assuming it is database data you re caching). We'll explore SQL cache dependencies in a future tutorial.

Summary

In this tutorial we examined the ObjectDataSource s built-in caching capabilities. By simply setting a few properties, we can instruct the ObjectDataSource to cache the results returned from the specified `SelectMethod` into the ASP.NET data cache. The `CacheDuration` and `CacheExpirationPolicy` properties indicate the duration the item is cached and whether it is an absolute or sliding expiration. The `CacheKeyDependency` property associates all of the ObjectDataSource s cache entries with an existing cache dependency. This can be used to evict the ObjectDataSource s entries from the cache before the time-based expiration is reached, and is typically used with SQL cache dependencies.

Since the ObjectDataSource simply caches its values to the data cache, we could replicate the ObjectDataSource s built-in functionality programmatically. It doesn t make sense to do this at the Presentation Layer, since the ObjectDataSource offers this functionality out of the box, but we can implement caching capabilities in a separate layer of the architecture. To do so, we'll need to repeat the same logic used by the ObjectDataSource. We'll explore how to programmatically work with the data cache from within the architecture in our next tutorial.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Caching: Techniques and Best Practices
- Caching Architecture Guide for .NET Framework Applications
- Output Caching in ASP.NET 2.0

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Caching Data in the Architecture (C#)

2/20/2020 • 16 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In the previous tutorial we learned how to apply caching at the Presentation Layer. In this tutorial we learn how to take advantage of our layered architecture to cache data at the Business Logic Layer. We do this by extending the architecture to include a Caching Layer.

Introduction

As we saw in the preceding tutorial, caching the ObjectDataSource s data is as simple as setting a couple of properties. Unfortunately, the ObjectDataSource applies caching at the Presentation Layer, which tightly couples the caching policies with the ASP.NET page. One of the reasons for creating a layered architecture is to allow such couplings to be broken. The Business Logic Layer, for instance, decouples the business logic from the ASP.NET pages, while the Data Access Layer decouples the data access details. This decoupling of business logic and data access details is preferred, in part, because it makes the system more readable, more maintainable, and more flexible to change. It also allows for domain knowledge and division of labor a developer working on the Presentation Layer doesn t need to be familiar with the database s details in order to do her job. Decoupling the caching policy from the Presentation Layer offers similar benefits.

In this tutorial we will augment our architecture to include a *Caching Layer* (or CL for short) that employs our caching policy. The Caching Layer will include a `ProductsCL` class that provides access to product information with methods like `GetProducts()`, `GetProductsByCategoryID(categoryID)`, and so forth, that, when invoked, will first attempt to retrieve the data from the cache. If the cache is empty, these methods will invoke the appropriate `ProductsBLL` method in the BLL, which would in turn get the data from the DAL. The `ProductsCL` methods cache the data retrieved from the BLL before returning it.

As Figure 1 shows, the CL resides between the Presentation and Business Logic Layers.

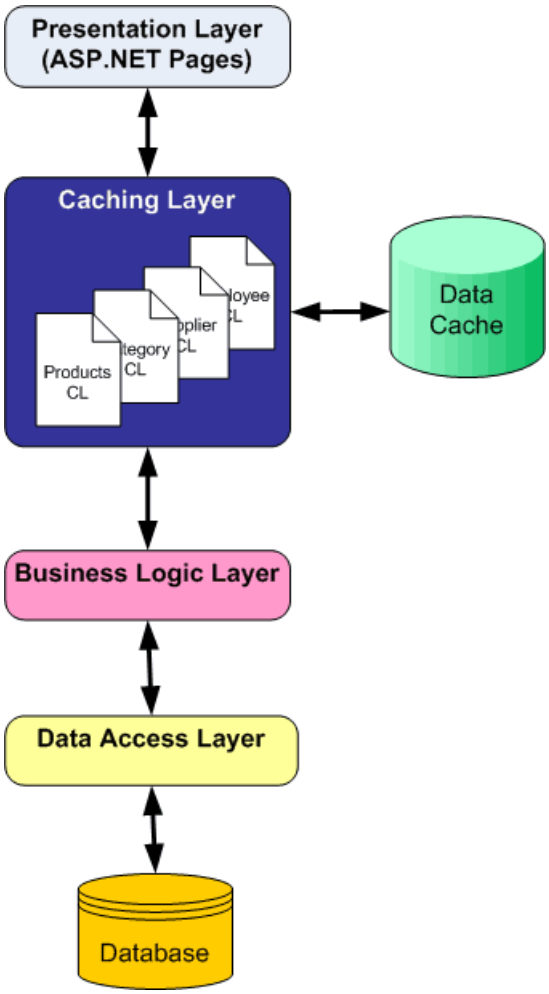


Figure 1: The Caching Layer (CL) is Another Layer in Our Architecture

Step 1: Creating the Caching Layer Classes

In this tutorial we will create a very simple CL with a single class `ProductsCL` that has only a handful of methods. Building a complete Caching Layer for the entire application would require creating `CategoriesCL`, `EmployeesCL`, and `SuppliersCL` classes, and providing a method in these Caching Layer classes for each data access or modification method in the BLL. As with the BLL and DAL, the Caching Layer should ideally be implemented as a separate Class Library project; however, we will implement it as a class in the `App_Code` folder.

To more cleanly separate the CL classes from the DAL and BLL classes, let s create a new subfolder in the `App_Code` folder. Right-click on the `App_Code` folder in the Solution Explorer, choose New Folder, and name the new folder `CL`. After creating this folder, add to it a new class named `ProductsCL.cs`.

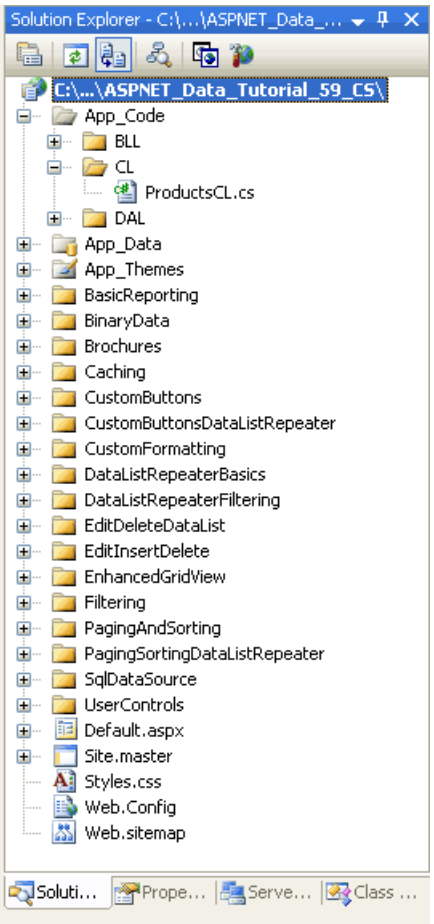


Figure 2: Add a New Folder Named `CL` and a Class Named `ProductsCL.cs`

The `ProductsCL` class should include the same set of data access and modification methods as found in its corresponding Business Logic Layer class (`ProductsBLL`). Rather than creating all of these methods, let s just build a couple here to get a feel for the patterns used by the CL. In particular, we'll add the `GetProducts()` and `GetProductsByCategoryID(categoryID)` methods in Step 3 and an `UpdateProduct` overload in Step 4. You can add the remaining `ProductsCL` methods and `CategoriesCL`, `EmployeesCL`, and `SuppliersCL` classes at your leisure.

Step 2: Reading and Writing to the Data Cache

The ObjectDataSource caching feature explored in the preceding tutorial internally uses the ASP.NET data cache to store the data retrieved from the BLL. The data cache can also be accessed programmatically from ASP.NET pages code-behind classes or from the classes in the web application s architecture. To read and write to the data cache from an ASP.NET page s code-behind class, use the following pattern:

```
// Read from the cache
object value = Cache["key"];
// Add a new item to the cache
Cache["key"] = value;
Cache.Insert(key, value);
Cache.Insert(key, value, CacheDependency);
Cache.Insert(key, value, CacheDependency, DateTime, TimeSpan);
```

The `Cache` class s `Insert` method has a number of overloads. `Cache["key"] = value` and `Cache.Insert(key, value)` are synonymous and both add an item to the cache using the specified key without a defined expiry. Typically, we want to specify an expiry when adding an item to the cache, either as a dependency, a time-based expiry, or both. Use one of the other `Insert` method s overloads to provide dependency- or time-based expiry information.

The Caching Layer s methods need to first check if the requested data is in the cache and, if so, return it from

there. If the requested data is not in the cache, the appropriate BLL method needs to be invoked. Its return value should be cached and then returned, as the following sequence diagram illustrates.

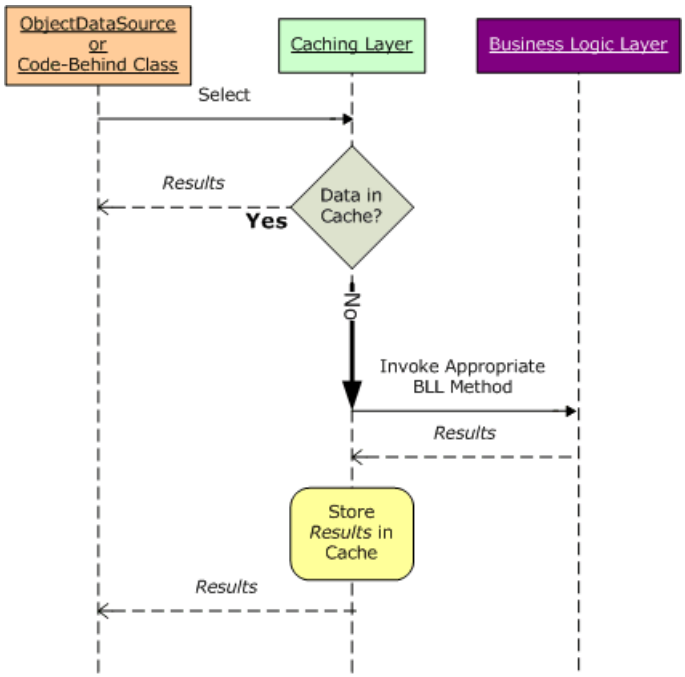


Figure 3: The Caching Layer s Methods Return Data from the Cache if it s Available

The sequence depicted in Figure 3 is accomplished in the CL classes using the following pattern:

```
Type instance = Cache["key"] as Type;
if (instance == null)
{
    instance = BllMethodToGetInstance();
    Cache.Insert(key, instance, ...);
}
return instance;
```

Here, *Type* is the type of data being stored in the cache `Northwind.ProductsDataTable`, for example while *key* is the key that uniquely identifies the cache item. If the item with the specified *key* is not in the cache, then *instance* will be `null` and the data will be retrieved from the appropriate BLL method and added to the cache. By the time `return instance` is reached, *instance* contains a reference to the data, either from the cache or pulled from the BLL.

Be sure to use the above pattern when accessing data from the cache. The following pattern, which, at first glance, looks equivalent, contains a subtle difference that introduces a race condition. Race conditions are difficult to debug because they reveal themselves sporadically and are difficult to reproduce.

```
if (Cache["key"] == null)
{
    Cache.Insert(key, BllMethodToGetInstance(), ...);
}
return Cache["key"];
```

The difference in this second, incorrect code snippet is that rather than storing a reference to the cached item in a local variable, the data cache is accessed directly in the conditional statement *and* in the `return`. Imagine that when this code is reached, `Cache["key"]` is non-`null`, but before the `return` statement is reached, the system evicts *key* from the cache. In this rare case, the code will return a `null` value rather than an object of the expected type.

> **NOTE**
> The data cache is thread-safe, so you don t need to synchronize thread access for simple reads or writes. However, if you need to perform multiple operations on data in the cache that need to be atomic, you are responsible for implementing a lock or some other mechanism to ensure thread safety. See Synchronizing Access to the ASP.NET Cache for more information.

An item can be programmatically evicted from the data cache using the `Remove` method like so:

```
Cache.Remove(key);
```

Step 3: Returning Product Information from the `ProductsCL` Class

For this tutorial let s implement two methods for returning product information from the `ProductsCL` class: `GetProducts()` and `GetProductsByCategoryID(categoryID)`. Like with the `ProductsBL` class in the Business Logic Layer, the `GetProducts()` method in the CL returns information about all of the products as a `Northwind.ProductsDataTable` object, while `GetProductsByCategoryID(categoryID)` returns all of the products from a specified category.

The following code shows a portion of the methods in the `ProductsCL` class:

```
[System.ComponentModel.DataObject]
public class ProductsCL
{
    private ProductsBLL _productsAPI = null;
    protected ProductsBLL API
    {
        get
        {
            if (_productsAPI == null)
                _productsAPI = new ProductsBLL();
            return _productsAPI;
        }
    }

   [System.ComponentModel.DataObjectMethodAttribute(DataObjectMethodType.Select, true)]
    public Northwind.ProductsDataTable GetProducts()
    {
        const string rawKey = "Products";
        // See if the item is in the cache
        Northwind.ProductsDataTable products = _
            GetCacheItem(rawKey) as Northwind.ProductsDataTable;
        if (products == null)
        {
            // Item not found in cache - retrieve it and insert it into the cache
            products = API.GetProducts();
            AddCacheItem(rawKey, products);
        }
        return products;
    }

    [System.ComponentModel.DataObjectMethodAttribute(DataObjectMethodType.Select, false)]
    public Northwind.ProductsDataTable GetProductsByCategoryID(int categoryID)
    {
        if (categoryID < 0)
            return GetProducts();
        else
        {
            string rawKey = string.Concat("ProductsByCategory-", categoryID);
            // See if the item is in the cache
            Northwind.ProductsDataTable products = _
                GetCacheItem(rawKey) as Northwind.ProductsDataTable;
            if (products == null)
            {
                // Item not found in cache - retrieve it and insert it into the cache
                products = API.GetProductsByCategoryID(categoryID);
                AddCacheItem(rawKey, products);
            }
            return products;
        }
    }
}
```

First, note the `DataObject` and `DataObjectMethodAttribute` attributes applied to the class and methods. These attributes provide information to the ObjectDataSource s wizard, indicating what classes and methods should appear in the wizard s steps. Since the CL classes and methods will be accessed from an ObjectDataSource in the Presentation Layer, I added these attributes to enhance the design-time experience. Refer back to the Creating a Business Logic Layer tutorial for a more thorough description on these attributes and their effects.

In the `GetProducts()` and `GetProductsByCategoryID(categoryID)` methods, the data returned from the `GetCacheItem(key)` method is assigned to a local variable. The `GetCacheItem(key)` method, which we'll examine shortly, returns a particular item from the cache based on the specified *key*. If no such data is found in cache, it is retrieved from the corresponding `ProductsBLL` class method and then added to the cache using the `AddCacheItem(key, value)` method.

The `GetCacheItem(key)` and `AddCacheItem(key, value)` methods interface with the data cache, reading and writing values, respectively. The `GetCacheItem(key)` method is the simpler of the two. It simply returns the value from the Cache class using the passed-in *key*:

```
private object GetCacheItem(string rawKey)
{
    return HttpRuntime.Cache[GetCacheKey(rawKey)];
}
private readonly string[] MasterCacheKeyArray = {"ProductsCache"};
private string GetCacheKey(string cacheKey)
{
    return string.Concat(MasterCacheKeyArray[0], "-", cacheKey);
}
```

`GetCacheItem(key)` does not use *key* value as supplied, but instead calls the `GetCacheKey(key)` method, which returns the *key* prepended with ProductsCache-. The `MasterCacheKeyArray`, which holds the string ProductsCache, is also used by the `AddCacheItem(key, value)` method, as we'll see momentarily.

From an ASP.NET page s code-behind class, the data cache can be accessed using the `Page` class s `Cache` property, and allows for syntax like `Cache["key"] = value`, as discussed in Step 2. From a class within the architecture, the data cache can be accessed using either `HttpRuntime.Cache` or `HttpContext.Current.Cache`. Peter Johnson's blog entry HttpRuntime.Cache vs. HttpContext.Current.Cache notes the slight performance advantage in using `HttpRuntime` instead of `HttpContext.Current`; consequently, `ProductsCL` uses `HttpRuntime`.

> **NOTE**
> If your architecture is implemented using Class Library projects then you will need to add a reference to the `System.Web` assembly in order to use the HttpRuntime and HttpContext classes.

If the item is not found in the cache, the `ProductsCL` class s methods get the data from the BLL and add it to the cache using the `AddCacheItem(key, value)` method. To add *value* to the cache we could use the following code, which uses a 60 second time expiry:

```
const double CacheDuration = 60.0;
private void AddCacheItem(string rawKey, object value)
{
    HttpRuntime.Cache.Insert(GetCacheKey(rawKey), value, null,
        DateTime.Now.AddSeconds(CacheDuration), Caching.Cache.NoSlidingExpiration);
}
```

`DateTime.Now.AddSeconds(CacheDuration)` specifies the time-based expiry 60 seconds in the future while `System.Web.Caching.Cache.NoSlidingExpiration` indicates that there s no sliding expiration. While this `Insert` method overload has input parameters for both an absolute and sliding expiry, you can only provide one of the two. If you attempt to specify both an absolute time and a time span, the `Insert` method will throw an `ArgumentException` exception.

> **NOTE**
> This implementation of the `AddCacheItem(key, value)` method currently has some shortcomings. We'll address and overcome these issues in Step 4.

Step 4: Invalidating the Cache When the Data is Modified Through the Architecture

Along with data retrieval methods, the Caching Layer needs to provide the same methods as the BLL for inserting, updating, and deleting data. The CL s data modification methods do not modify the cached data, but rather call the BLL s corresponding data modification method and then invalidate the cache. As we saw in the preceding tutorial, this is the same behavior that the ObjectDataSource applies when its caching features are enabled and its `Insert`, `Update`, or `Delete` methods are invoked.

The following `UpdateProduct` overload illustrates how to implement the data modification methods in the CL:

```
[System.ComponentModel.DataObjectMethodAttribute(DataObjectMethodType.Update, false)]
public bool UpdateProduct(string productName, decimal? unitPrice, int productID)
{
    bool result = API.UpdateProduct(productName, unitPrice, productID);
    // TODO: Invalidate the cache
    return result;
}
```

The appropriate data modification Business Logic Layer method is invoked, but before its response is returned we need to invalidate the cache. Unfortunately, invalidating the cache is not straightforward because the `ProductsCL` class s `GetProducts()` and `GetProductsByCategoryID(categoryID)` methods each add items to the cache with different keys, and the `GetProductsByCategoryID(categoryID)` method adds a different cache item for each unique *categoryID*.

When invalidating the cache, we need to remove *all* of the items that may have been added by the `ProductsCL` class. This can be accomplished by associating a *cache dependency* with the each item added to the cache in the `AddCacheItem(key, value)` method. In general, a cache dependency can be another item in the cache, a file on the file system, or data from a Microsoft SQL Server database. When the dependency changes or is removed from the cache, the cache items it is associated with are automatically evicted from the cache. For this tutorial, we want to create an additional item in the cache that serves as a cache dependency for all items added through the `ProductsCL` class. That way, all of these items can be removed from the cache by simply removing the cache dependency.

Let s update the `AddCacheItem(key, value)` method so that each item added to the cache through this method is associated with a single cache dependency:

```
private void AddCacheItem(string rawKey, object value)
{
    System.Web.Caching.Cache DataCache = HttpRuntime.Cache;
    // Make sure MasterCacheKeyArray[0] is in the cache - if not, add it
    if (DataCache[MasterCacheKeyArray[0]] == null)
        DataCache[MasterCacheKeyArray[0]] = DateTime.Now;
    // Add a CacheDependency
    System.Web.Caching.CacheDependency dependency =
        new CacheDependency(null, MasterCacheKeyArray);
    DataCache.Insert(GetCacheKey(rawKey), value, dependency,
        DateTime.Now.AddSeconds(CacheDuration),
        System.Web.Caching.Cache.NoSlidingExpiration);
}
```

`MasterCacheKeyArray` is a string array that holds a single value, ProductsCache. First, a cache item is added to the cache and assigned the current date and time. If the cache item already exists, it is updated. Next, a cache dependency is created. The `CacheDependency` class s constructor has a number of overloads, but the one being used in here expects two `string` array inputs. The first one specifies the set of files to be used as dependencies. Since we don t want to use any file-based dependencies, a value of `null` is used for the first input parameter. The second input parameter specifies the set of cache keys to use as dependencies. Here we specify our single dependency, `MasterCacheKeyArray`. The `CacheDependency` is then passed into the `Insert` method.

With this modification to `AddCacheItem(key, value)`, invaliding the cache is as simple as removing the

dependency.

```
[System.ComponentModel.DataObjectMethodAttribute(DataObjectMethodType.Update, false)]
public bool UpdateProduct(string productName, decimal? unitPrice, int productID)
{
    bool result = API.UpdateProduct(productName, unitPrice, productID);
    // Invalidate the cache
    InvalidateCache();
    return result;
}
public void InvalidateCache()
{
    // Remove the cache dependency
    HttpRuntime.Cache.Remove(MasterCacheKeyArray[0]);
}
```

Step 5: Calling the Caching Layer from the Presentation Layer

The Caching Layer s classes and methods can be used to work with data using the techniques we ve examined throughout these tutorials. To illustrate working with cached data, save your changes to the `ProductsCL` class and then open the `FromTheArchitecture.aspx` page in the `Caching` folder and add a GridView. From the GridView s smart tag, create a new ObjectDataSource. In the wizard s first step you should see the `ProductsCL` class as one of the options from the drop-down list.

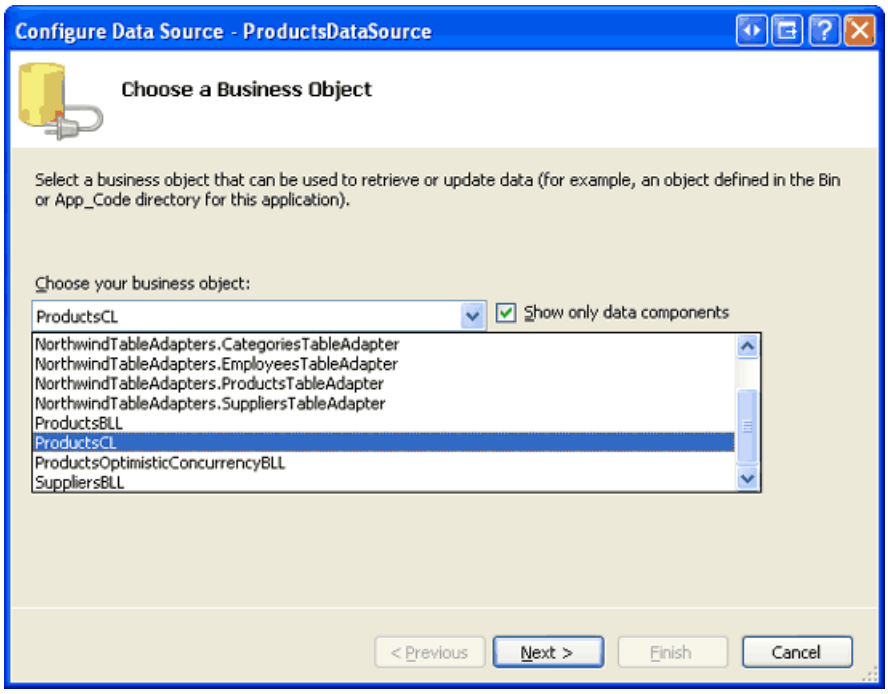


Figure 4: The `ProductsCL` Class is Included in the Business Object Drop-Down List (Click to view full-size image)

After selecting `ProductsCL`, click Next. The drop-down list in the SELECT tab has two items - `GetProducts()` and `GetProductsByCategoryID(categoryID)` and the UPDATE tab has the sole `UpdateProduct` overload. Choose the `GetProducts()` method from the SELECT tab and the `UpdateProducts` method from the UPDATE tab and click Finish.

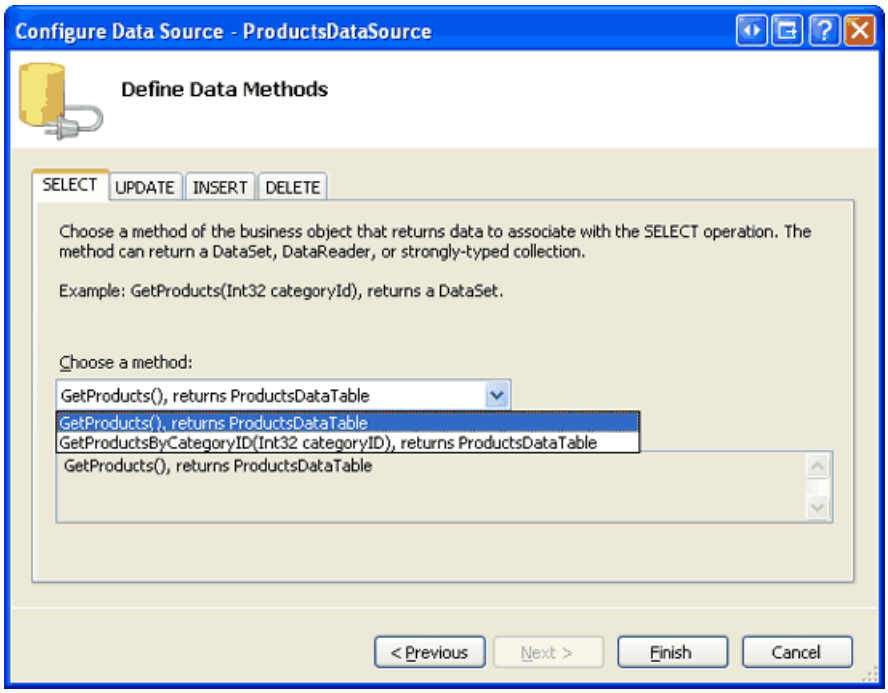


Figure 5: The `ProductsCL` Class s Methods are Listed in the Drop-Down Lists (Click to view full-size image)

After completing the wizard, Visual Studio will set the ObjectDataSource s `OldValuesParameterFormatString` property to `original_{0}` and add the appropriate fields to the GridView. Change the `OldValuesParameterFormatString` property back to its default value, `{0}`, and configure the GridView to support paging, sorting, and editing. Since the `UploadProducts` overload used by the CL accepts only the edited product s name and price, limit the GridView so that only these fields are editable.

In the preceding tutorial we defined a GridView to include fields for the `ProductName`, `CategoryName`, and `UnitPrice` fields. Feel free to replicate this formatting and structure, in which case your GridView and ObjectDataSource s declarative markup should look similar to the following:

```
<asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
    AllowPaging="True" AllowSorting="True">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
            <EditItemTemplate>
                <asp:TextBox ID="ProductName" runat="server"
                    Text='<%# Bind("ProductName") %>' />
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                    ControlToValidate="ProductName" Display="Dynamic"
                    ErrorMessage="You must provide a name for the product."
                    SetFocusOnError="True"
                    runat="server">*</asp:RequiredFieldValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:TemplateField HeaderText="Price" SortExpression="UnitPrice">
            <EditItemTemplate>
                $<asp:TextBox ID="UnitPrice" runat="server" Columns="8"
                    Text='<%# Bind("UnitPrice", "{0:N2}") %>'></asp:TextBox>
                <asp:CompareValidator ID="CompareValidator1" runat="server"
                    ControlToValidate="UnitPrice" Display="Dynamic"
                    ErrorMessage="You must enter a valid currency value with
                        no currency symbols. Also, the value must be greater than
                        or equal to zero."
                    Operator="GreaterThanEqual" SetFocusOnError="True"
                    Type="Currency" ValueToCompare="0">*</asp:CompareValidator>
            </EditItemTemplate>
            <ItemStyle HorizontalAlign="Right" />
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("UnitPrice", "{0:c}") %>' />
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="{0}" SelectMethod="GetProducts"
    TypeName="ProductsCL" UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

At this point we have a page that uses the Caching Layer. To see the cache in action, set breakpoints in the `ProductsCL` class s `GetProducts()` and `UpdateProduct` methods. Visit the page in a browser and step through the code when sorting and paging in order to see the data pulled from the cache. Then update a record and note that the cache is invalidated and, consequently, it is retrieved from the BLL when the data is rebound to the GridView.

NOTE

The Caching Layer provided in the download accompanying this article is not complete. It contains only one class, `ProductsCL`, which only sports a handful of methods. Moreover, only a single ASP.NET page uses the CL (`~/Caching/FromTheArchitecture.aspx`) all others still reference the BLL directly. If you plan on using a CL in your application, all calls from the Presentation Layer should go to the CL, which would require that the CL s classes and methods covered those classes and methods in the BLL currently used by the Presentation Layer.

Summary

While caching can be applied at the Presentation Layer with ASP.NET 2.0 s SqlDataSource and ObjectDataSource controls, ideally caching responsibilities would be delegated to a separate layer in the architecture. In this tutorial we created a Caching Layer that resides between the Presentation Layer and the Business Logic Layer. The Caching Layer needs to provide the same set of classes and methods that exist in the BLL and are called from the Presentation Layer.

The Caching Layer examples we explored in this and the preceding tutorials exhibited *reactive loading*. With reactive loading, the data is loaded into the cache only when a request for the data is made and that data is missing from the cache. Data can also be *proactively loaded* into the cache, a technique that loads the data into the cache before it is actually needed. In the next tutorial we'll see an example of proactive loading when we look at how to store static values into the cache at application startup.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Teresa Murph. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Caching Data at Application Startup (C#)

2/20/2020 • 14 minutes to read • Edit Online

by Scott Mitchell

Download PDF

In any Web application some data will be frequently used and some data will be infrequently used. We can improve the performance of our ASP.NET application by loading in advance the frequently-used data, a technique known as Caching. This tutorial demonstrates one approach to proactive loading, which is to load data into the cache at application startup.

Introduction

The two previous tutorials looked at caching data in the Presentation and Caching Layers. In Caching Data with the ObjectDataSource, we looked at using the ObjectDataSource's caching features to cache data in the Presentation Layer. Caching Data in the Architecture examined caching in a new, separate Caching Layer. Both of these tutorials used *reactive loading* in working with the data cache. With reactive loading, each time the data is requested, the system first checks if it's in the cache. If not, it grabs the data from the originating source, such as the database, and then stores it in the cache. The main advantage to reactive loading is its ease of implementation. One of its disadvantages is its uneven performance across requests. Imagine a page that uses the Caching Layer from the preceding tutorial to display product information. When this page is visited for the first time, or visited for the first time after the cached data has been evicted due to memory constraints or the specified expiry having been reached, the data must be retrieved from the database. Therefore, these users requests will take longer than users requests that can be served by the cache.

Proactive loading provides an alternative cache management strategy that smoothes out the performance across requests by loading the cached data before it's needed. Typically, proactive loading uses some process that either periodically checks or is notified when there has been an update to the underlying data. This process then updates the cache to keep it fresh. Proactive loading is especially useful if the underlying data comes from a slow database connection, a Web service, or some other particularly sluggish data source. But this approach to proactive loading is more difficult to implement, as it requires creating, managing, and deploying a process to check for changes and update the cache.

Another flavor of proactive loading, and the type we'll be exploring in this tutorial, is loading data into the cache at application startup. This approach is especially useful for caching static data, such as the records in database lookup tables.

> **NOTE**
>
> For a more in-depth look at the differences between proactive and reactive loading, as well as lists of pros, cons, and implementation recommendations, refer to the Managing the Contents of a Cache section of the Caching Architecture Guide for .NET Framework Applications.

Step 1: Determining What Data to Cache at Application Startup

The caching examples using reactive loading that we examined in the previous two tutorials work well with data that may periodically change and does not take exorbitantly long to generate. But if the cached data never changes, the expiry used by reactive loading is superfluous. Likewise, if the data being cached takes an exceedingly long time to generate, then those users whose requests find the cache empty will have to endure a

lengthy wait while the underlying data is retrieved. Consider caching static data and data that takes an exceptionally long time to generate at application startup.

While databases have many dynamic, frequently-changing values, most also have a fair amount of static data. For example, virtually all data models have one or more columns that contain a particular value from a fixed set of choices. A `Patients` database table might have a `PrimaryLanguage` column, whose set of values could be English, Spanish, French, Russian, Japanese, and so on. Oftentimes, these types of columns are implemented using *lookup tables*. Rather than storing the string English or French in the `Patients` table, a second table is created that has, commonly, two columns - a unique identifier and a string description - with a record for each possible value. The `PrimaryLanguage` column in the `Patients` table stores the corresponding unique identifier in the lookup table. In Figure 1, patient John Doe's primary language is English, while Ed Johnson's is Russian.

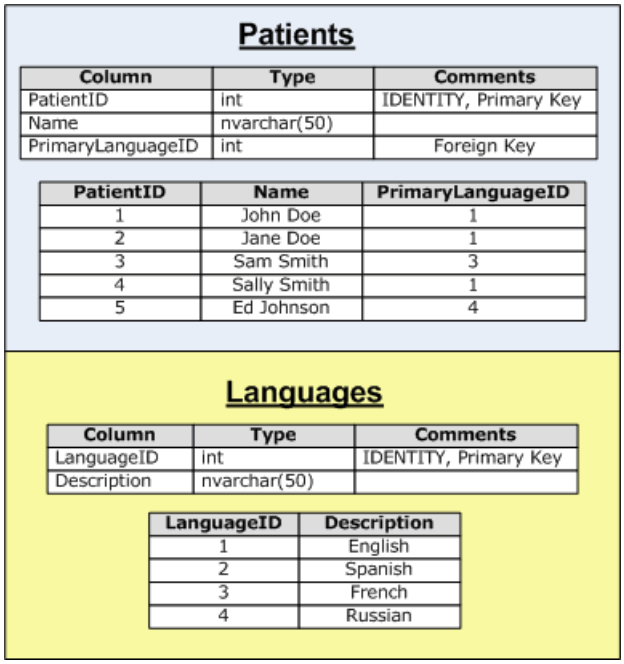


Figure 1: The `Languages` Table is a Lookup Table Used by the `Patients` Table

The user interface for editing or creating a new patient would include a drop-down list of allowable languages populated by the records in the `Languages` table. Without caching, each time this interface is visited the system must query the `Languages` table. This is wasteful and unnecessary since lookup table values change very infrequently, if ever.

We could cache the `Languages` data using the same reactive loading techniques examined in the previous tutorials. Reactive loading, however, uses a time-based expiry, which is not needed for static lookup table data. While caching using reactive loading would be better than no caching at all, the best approach would be to proactively load the lookup table data into the cache at application startup.

In this tutorial we will look at how to cache lookup table data and other static information.

Step 2: Examining the Different Ways to Cache Data

Information can be programmatically cached in an ASP.NET application using a variety of approaches. We ve already seen how to use the data cache in previous tutorials. Alternatively, objects can be programmatically cached using *static members* or *application state*.

When working with a class, typically the class must first be instantiated before its members can be accessed. For example, in order to invoke a method from one of the classes in our Business Logic Layer, we must first create an instance of the class:

```
ProductsBLL productsAPI = new ProductsBLL();
productsAPI.SomeMethod();
productsAPI.SomeProperty = "Hello, World!";
```

Before we can invoke *SomeMethod* or work with *SomeProperty*, we must first create an instance of the class using the `new` keyword. *SomeMethod* and *SomeProperty* are associated with a particular instance. The lifetime of these members is tied to the lifetime of their associated object. *Static members*, on the other hand, are variables, properties, and methods that are shared among *all* instances of the class and, consequently, have a lifetime as long as the class. Static members are denoted by the keyword `static`.

In addition to static members, data can be cached using application state. Each ASP.NET application maintains a name/value collection that's shared across all users and pages of the application. This collection can be accessed using the `HttpContext` class's `Application` property, and used from an ASP.NET page's code-behind class like so:

```
Application["key"] = value;
object value = Application["key"];
```

The data cache provides a much richer API for caching data, providing mechanisms for time- and dependency-based expiries, cache item priorities, and so forth. With static members and application state, such features must be manually added by the page developer. When caching data at application startup for the lifetime of the application, however, the data cache's advantages are moot. In this tutorial we'll look at code that uses all three techniques for caching static data.

Step 3: Caching the `Suppliers` Table Data

The Northwind database tables we've implemented to date do not include any traditional lookup tables. The four DataTables implemented in our DAL all model tables whose values are non-static. Rather than spending the time to add a new DataTable to the DAL and then a new class and methods to the BLL, for this tutorial let's just pretend that the `Suppliers` table's data is static. Therefore, we could cache this data at application startup.

To start, create a new class named `StaticCache.cs` in the `CL` folder.

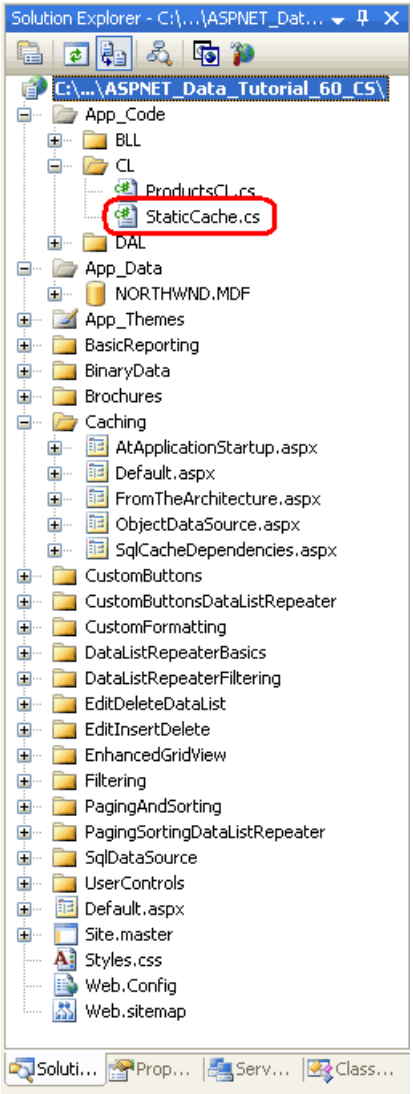


Figure 2: Create the `StaticCache.cs` Class in the `CL` Folder

We need to add a method that loads the data at startup into the appropriate cache store, as well as methods that return data from this cache.

```
[System.ComponentModel.DataObject]
public class StaticCache
{
    private static Northwind.SuppliersDataTable suppliers = null;
    public static void LoadStaticCache()
    {
        // Get suppliers - cache using a static member variable
        SuppliersBLL suppliersBLL = new SuppliersBLL();
        suppliers = suppliersBLL.GetSuppliers();
    }
    [DataObjectMethodAttribute(DataObjectMethodType.Select, true)]
    public static Northwind.SuppliersDataTable GetSuppliers()
    {
        return suppliers;
    }
}
```

The above code uses a static member variable, `suppliers`, to hold the results from the `SuppliersBLL` class's `GetSuppliers()` method, which is called from the `LoadStaticCache()` method. The `LoadStaticCache()` method is meant to be called during the application's start. Once this data has been loaded at application startup, any page that needs to work with supplier data can call the `StaticCache` class's `GetSuppliers()` method. Therefore, the call to the database to get the suppliers only happens once, at application start.

Rather than using a static member variable as the cache store, we could have alternatively used application state or the data cache. The following code shows the class retooled to use application state:

```
[System.ComponentModel.DataObject]
public class StaticCache
{
    public static void LoadStaticCache()
    {
        // Get suppliers - cache using application state
        SuppliersBLL suppliersBLL = new SuppliersBLL();
        HttpContext.Current.Application["key"] = suppliersBLL.GetSuppliers();
    }
    [DataObjectMethodAttribute(DataObjectMethodType.Select, true)]
    public static Northwind.SuppliersDataTable GetSuppliers()
    {
        return HttpContext.Current.Application["key"] as Northwind.SuppliersDataTable;
    }
}
```

In `LoadStaticCache()`, the supplier information is stored to the application variable *key*. It's returned as the appropriate type (`Northwind.SuppliersDataTable`) from `GetSuppliers()`. While application state can be accessed in the code-behind classes of ASP.NET pages using `Application["key"]`, in the architecture we must use `HttpContext.Current.Application["key"]` in order to get the current `HttpContext`.

Likewise, the data cache can be used as a cache store, as the following code shows:

```
[System.ComponentModel.DataObject]
public class StaticCache
{
    public static void LoadStaticCache()
    {
        // Get suppliers - cache using the data cache
        SuppliersBLL suppliersBLL = new SuppliersBLL();
        HttpRuntime.Cache.Insert(
          /* key */                "key",
          /* value */              suppliers,
          /* dependencies */       null,
          /* absoluteExpiration */ Cache.NoAbsoluteExpiration,
          /* slidingExpiration */  Cache.NoSlidingExpiration,
          /* priority */           CacheItemPriority.NotRemovable,
          /* onRemoveCallback */   null);
    }
    [DataObjectMethodAttribute(DataObjectMethodType.Select, true)]
    public static Northwind.SuppliersDataTable GetSuppliers()
    {
        return HttpRuntime.Cache["key"] as Northwind.SuppliersDataTable;
    }
}
```

To add an item to the data cache with no time-based expiry, use the `System.Web.Caching.Cache.NoAbsoluteExpiration` and `System.Web.Caching.Cache.NoSlidingExpiration` values as input parameters. This particular overload of the data cache's `Insert` method was selected so that we could specify the *priority* of the cache item. The priority is used to determine what items to scavenge from the cache when available memory runs low. Here we use the priority `NotRemovable`, which ensures that this cache item won't be scavenged.

> **NOTE**
> This tutorial's download implements the `StaticCache` class using the static member variable approach. The code for the application state and data cache techniques is available in the comments in the class file.

Step 4: Executing Code at Application Startup

To execute code when a web application first starts, we need to create a special file named `Global.asax`. This file can contain event handlers for application-, session-, and request-level events, and it is here where we can add code that will be executed whenever the application starts.

Add the `Global.asax` file to your web application's root directory by right-clicking on the website project name in Visual Studio's Solution Explorer and choosing Add New Item. From the Add New Item dialog box, select the Global Application Class item type and then click the Add button.

> **NOTE**
> If you already have a `Global.asax` file in your project, the Global Application Class item type will not be listed in the Add New Item dialog box.

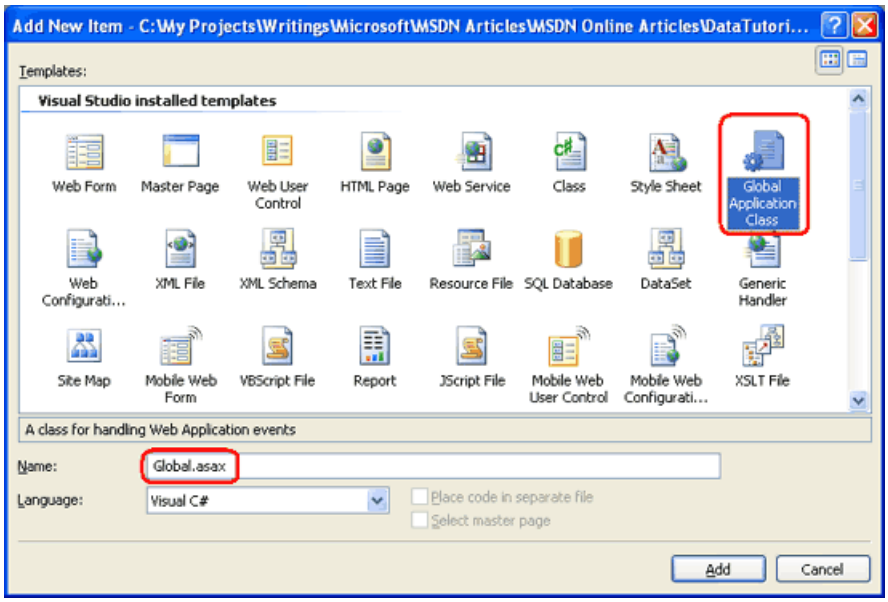


Figure 3: Add the `Global.asax` File to Your Web Application's Root Directory (Click to view full-size image)

The default `Global.asax` file template includes five methods within a server-side `<script>` tag:

- `Application_Start` executes when the web application first starts
- `Application_End` runs when the application is shutting down
- `Application_Error` executes whenever an unhandled exception reaches the application
- `Session_Start` executes when a new session is created
- `Session_End` runs when a session is expired or abandoned

The `Application_Start` event handler is called only once during an application's life cycle. The application starts the first time an ASP.NET resource is requested from the application and continues to run until the application is restarted, which can happen by modifying the contents of the `/Bin` folder, modifying `Global.asax`, modifying the contents in the `App_Code` folder, or modifying the `Web.config` file, among other causes. Refer to ASP.NET Application Life Cycle Overview for a more detailed discussion on the application life cycle.

For these tutorials we only need to add code to the `Application_Start` method, so feel free to remove the others.

In `Application_Start`, simply call the `StaticCache` class's `LoadStaticCache()` method, which will load and cache the supplier information:

```
<%@ Application Language="C#" %>
<script runat="server">
    void Application_Start(object sender, EventArgs e)
    {
        StaticCache.LoadStaticCache();
    }
</script>
```

That's all there is to it! At application startup, the `LoadStaticCache()` method will grab the supplier information from the BLL, and store it in a static member variable (or whatever cache store you ended up using in the `StaticCache` class). To verify this behavior, set a breakpoint in the `Application_Start` method and run your application. Note that the breakpoint is hit upon the application starting. Subsequent requests, however, do not cause the `Application_Start` method to execute.

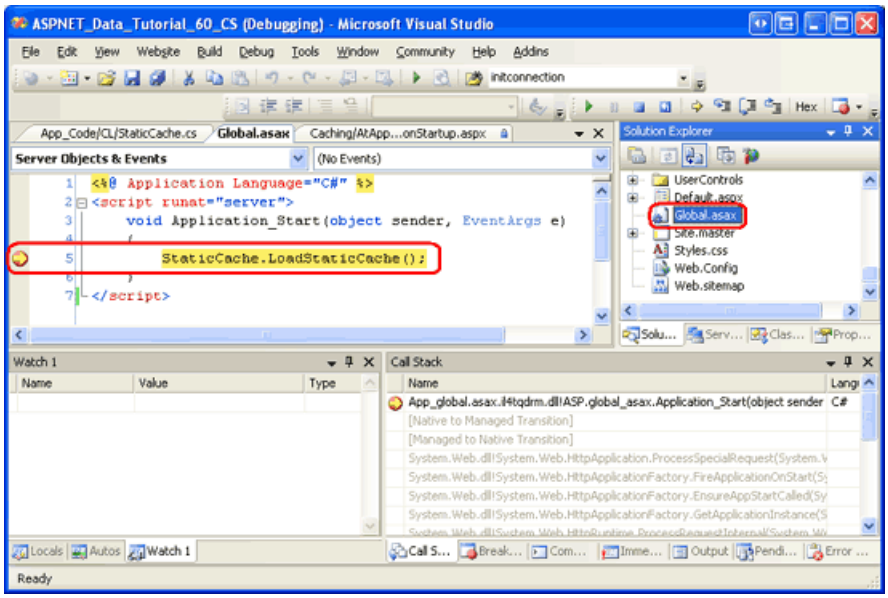


Figure 4: Use a Breakpoint to Verify that the `Application_Start` Event Handler is Being Executed (Click to view full-size image)

> **NOTE**
>
> If you do not hit the `Application_Start` breakpoint when you first start debugging, it is because your application has already started. Force the application to restart by modifying your `Global.asax` or `Web.config` files and then try again. You can simply add (or remove) a blank line at the end of one of these files to quickly restart the application.

Step 5: Displaying the Cached Data

At this point the `StaticCache` class has a version of the supplier data cached at application startup that can be accessed through its `GetSuppliers()` method. To work with this data from the Presentation Layer, we can use an ObjectDataSource or programmatically invoke the `StaticCache` class's `GetSuppliers()` method from an ASP.NET page's code-behind class. Let's look at using the ObjectDataSource and GridView controls to display the cached supplier information.

Start by opening the `AtApplicationStartup.aspx` page in the `Caching` folder. Drag a GridView from the Toolbox onto the designer, setting its `ID` property to `Suppliers`. Next, from the GridView's smart tag choose to create a new ObjectDataSource named `SuppliersCachedDataSource`. Configure the ObjectDataSource to use the `StaticCache` class's `GetSuppliers()` method.

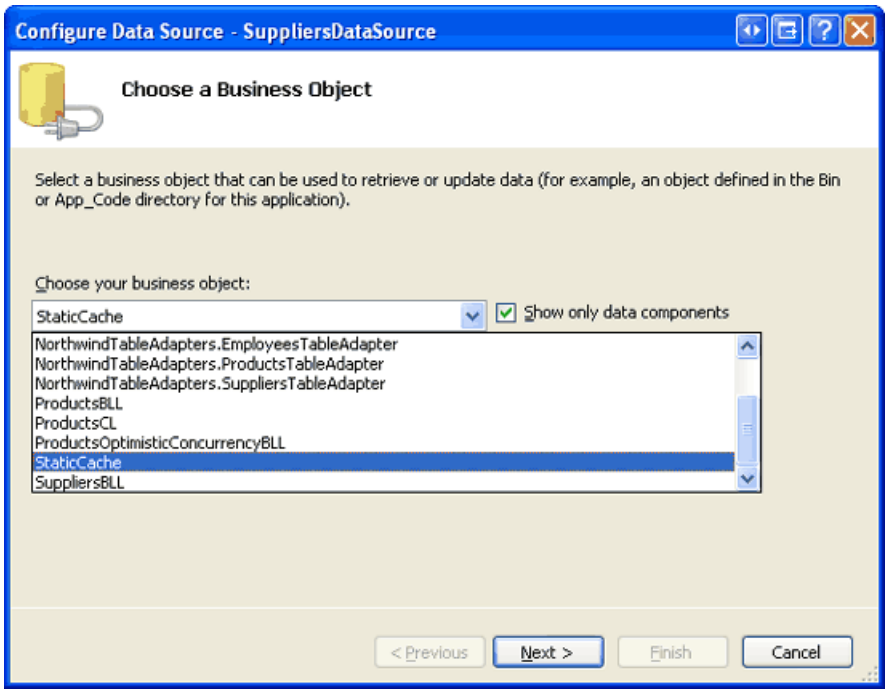


Figure 5: Configure the ObjectDataSource to use the `StaticCache` Class (Click to view full-size image)

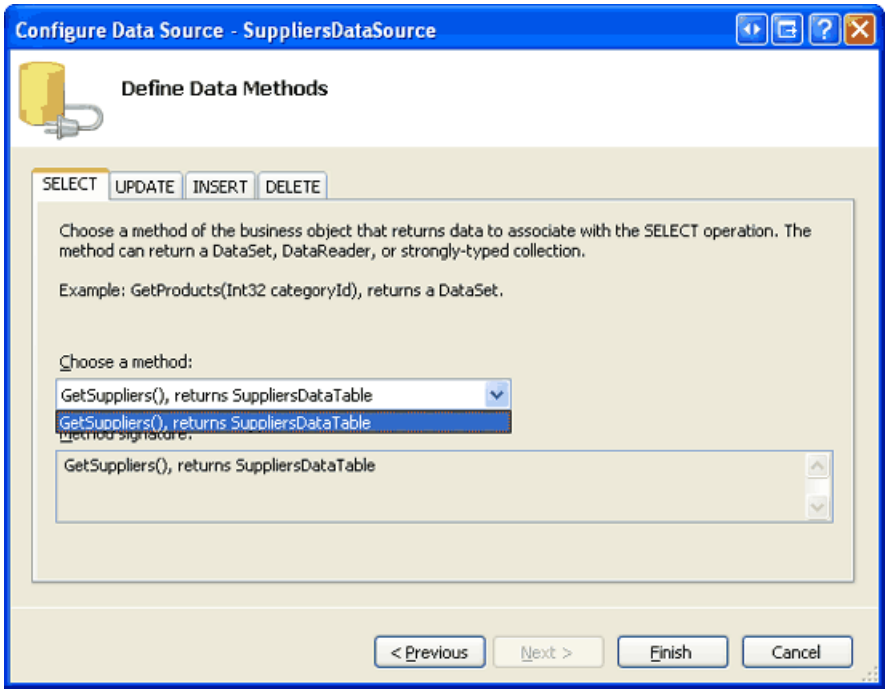


Figure 6: Use the `GetSuppliers()` Method to Retrieve the Cached Supplier Data (Click to view full-size image)

After completing the wizard, Visual Studio will automatically add BoundFields for each of the data fields in `SuppliersDataTable`. Your GridView and ObjectDataSource's declarative markup should look similar to the following:

```
<asp:GridView ID="Suppliers" runat="server" AutoGenerateColumns="False"
    DataKeyNames="SupplierID" DataSourceID="SuppliersCachedDataSource"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="SupplierID" HeaderText="SupplierID"
            InsertVisible="False" ReadOnly="True"
            SortExpression="SupplierID" />
        <asp:BoundField DataField="CompanyName" HeaderText="CompanyName"
            SortExpression="CompanyName" />
        <asp:BoundField DataField="Address" HeaderText="Address"
            SortExpression="Address" />
        <asp:BoundField DataField="City" HeaderText="City"
            SortExpression="City" />
        <asp:BoundField DataField="Country" HeaderText="Country"
            SortExpression="Country" />
        <asp:BoundField DataField="Phone" HeaderText="Phone"
            SortExpression="Phone" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="SuppliersCachedDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetSuppliers" TypeName="StaticCache" />
```

Figure 7 shows the page when viewed through a browser. The output is the same had we pulled the data from the BLL's `SuppliersBLL` class, but using the `StaticCache` class returns the supplier data as cached at application startup. You can set breakpoints in the `StaticCache` class's `GetSuppliers()` method to verify this behavior.

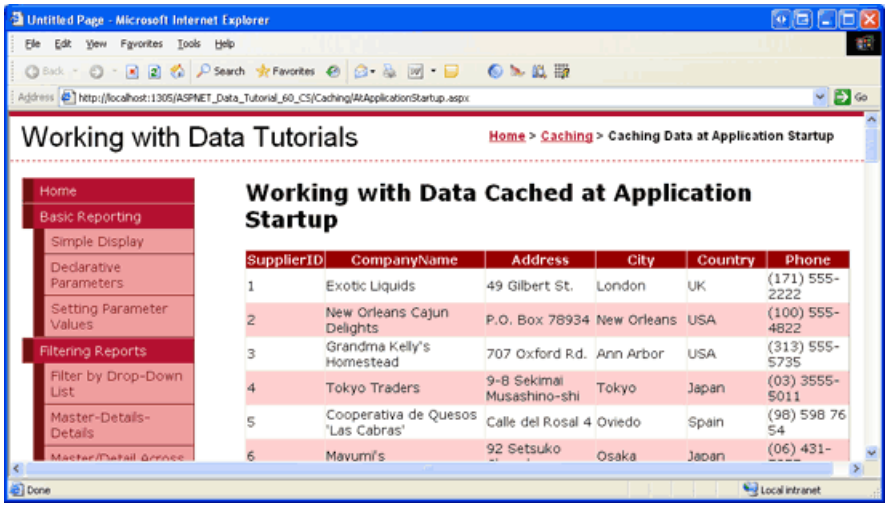


Figure 7: The Cached Supplier Data is Displayed in a GridView (Click to view full-size image)

Summary

Most every data model contains a fair amount of static data, usually implemented in the form of lookup tables. Since this information is static, there's no reason to continually access the database each time this information needs to be displayed. Furthermore, due to its static nature, when caching the data there's no need for an expiry. In this tutorial we saw how to take such data and cache it in the data cache, application state, and through a static member variable. This information is cached at application startup and remains in the cache throughout the application's lifetime.

In this tutorial and the past two, we ve looked at caching data for the duration of the application's lifetime as well as using time-based expiries. When caching database data, though, a time-based expiry may be less than ideal. Rather than periodically flushing the cache, it would be optimal to only evict the cached item when the underlying database data is modified. This ideal is possible through the use of SQL cache dependencies, which we'll examine in our next tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Teresa Murphy and Zack Jones. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Using SQL Cache Dependencies (C#)

9/16/2020 • 25 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

The simplest caching strategy is to allow cached data to expire after a specified period of time. But this simple approach means that the cached data maintains no association with its underlying data source, resulting in stale data that is held too long or current data that is expired too soon. A better approach is to use the SqlCacheDependency class so that data remains cached until its underlying data has been modified in the SQL database. This tutorial shows you how.

Introduction

The caching techniques examined in the Caching Data with the ObjectDataSource and Caching Data in the Architecture tutorials used a time-based expiry to evict the data from the cache after a specified period. This approach is the simplest way to balance the performance gains of caching against data staleness. By selecting a time expiry of *x* seconds, a page developer concedes to enjoy the performance benefits of caching for only *x* seconds, but can rest easy that her data will never be stale longer than a maximum of *x* seconds. Of course, for static data, *x* can be extended to the lifetime of the web application, as was examined in the Caching Data at Application Startup tutorial.

When caching database data, a time-based expiry is often chosen for its ease of use but is frequently an inadequate solution. Ideally, the database data would remain cached until the underlying data has been modified in the database; only then would the cache be evicted. This approach maximizes the performance benefits of caching and minimizes the duration of stale data. However, in order to enjoy these benefits there must be some system in place that knows when the underlying database data has been modified and evicts the corresponding items from the cache. Prior to ASP.NET 2.0, page developers were responsible for implementing this system.

ASP.NET 2.0 provides a `SqlCacheDependency` class and the necessary infrastructure to determine when a change has occurred in the database so that the corresponding cached items can be evicted. There are two techniques for determining when the underlying data has changed: notification and polling. After discussing the differences between notification and polling, we'll create the infrastructure necessary to support polling and then explore how to use the `SqlCacheDependency` class in declarative and programmatically scenarios.

Understanding Notification and Polling

There are two techniques that can be used to determine when the data in a database has been modified: notification and polling. With notification, the database automatically alerts the ASP.NET runtime when the results of a particular query have been changed since the query was last executed, at which point the cached items associated with the query are evicted. With polling, the database server maintains information about when particular tables have last been updated. The ASP.NET runtime periodically polls the database to check what tables have changed since they were entered into the cache. Those tables whose data has been modified have their associated cache items evicted.

The notification option requires less setup than polling and is more granular since it tracks changes at the query level rather than at the table level. Unfortunately, notifications are only available in the full editions of Microsoft SQL Server 2005 (i.e., the non-Express editions). However, the polling option can be used for all versions of Microsoft SQL Server from 7.0 to 2005. Since these tutorials use the Express edition of SQL Server 2005, we will

focus on setting up and using the polling option. Consult the Further Reading section at the end of this tutorial for further resources on SQL Server 2005 s notification capabilities.

With polling, the database must be configured to include a table named `AspNet_SqlCacheTablesForChangeNotification` that has three columns - `tableName`, `notificationCreated`, and `changeId`. This table contains a row for each table that has data that might need to be used in a SQL cache dependency in the web application. The `tableName` column specifies the name of the table while `notificationCreated` indicates the date and time the row was added to the table. The `changeId` column is of type `int` and has an initial value of 0. Its value is incremented with each modification to the table.

In addition to the `AspNet_SqlCacheTablesForChangeNotification` table, the database also needs to include triggers on each of the tables that may appear in a SQL cache dependency. These triggers are executed whenever a row is inserted, updated, or deleted and increment the table s `changeId` value in `AspNet_SqlCacheTablesForChangeNotification`.

The ASP.NET runtime tracks the current `changeId` for a table when caching data using a `SqlCacheDependency` object. The database is periodically checked and any `SqlCacheDependency` objects whose `changeId` differs from the value in the database are evicted since a differing `changeId` value indicates that there has been a change to the table since the data was cached.

Step 1: Exploring the `aspnet_regsql.exe` Command Line Program

With the polling approach the database must be setup to contain the infrastructure described above: a predefined table (`AspNet_SqlCacheTablesForChangeNotification`), a handful of stored procedures, and triggers on each of the tables that may be used in SQL cache dependencies in the web application. These tables, stored procedures, and triggers can be created through the command line program `aspnet_regsql.exe`, which is found in the `$WINDOWS$\Microsoft.NET\Framework\version` folder. To create the `AspNet_SqlCacheTablesForChangeNotification` table and associated stored procedures, run the following from the command line:

```
/* For SQL Server authentication... */
aspnet_regsql.exe -S server -U user -P password -d database -ed
/* For Windows Authentication... */
aspnet_regsql.exe -S server -E -d database -ed
```

> **NOTE**
> To execute these commands the specified database login must be in the `db_securityadmin` and `db_ddladmin` roles. To examine the T-SQL sent to the database by the `aspnet_regsql.exe` command line program, refer to this blog entry.

For example, to add the infrastructure for polling to a Microsoft SQL Server database named `pubs` on a database server named `ScottsServer` using Windows Authentication, navigate to the appropriate directory and, from the command line, enter:

```
aspnet_regsql.exe -S ScottsServer -E -d pubs -ed
```

After the database-level infrastructure has been added, we need to add the triggers to those tables that will be used in SQL cache dependencies. Use the `aspnet_regsql.exe` command line program again, but specify the table name using the `-t` switch and instead of using the `-ed` switch use `-et`, like so:

```
/* For SQL Server authentication... */
aspnet_regsql.exe -S <i>server</i>
-U <i>user</i> -P <i>password</i> -d <i>database</i> -t <i>tableName</i> -et
/* For Windows Authentication... */
aspnet_regsql.exe -S <i>server</i>
-E -d <i>database</i> -t <i>tableName</i> -et
```

To add the triggers to the `authors` and `titles` tables on the `pubs` database on `ScottsServer`, use:

```
aspnet_regsql.exe -S ScottsServer -E -d pubs -t authors -et
aspnet_regsql.exe -S ScottsServer -E -d pubs -t titles -et
```

For this tutorial add the triggers to the `Products`, `Categories`, and `Suppliers` tables. We'll look at the particular command line syntax in Step 3.

Step 2: Referencing a Microsoft SQL Server 2005 Express Edition Database in `App_Data`

The `aspnet_regsql.exe` command line program requires the database and server name in order to add the necessary polling infrastructure. But what is the database and server name for a Microsoft SQL Server 2005 Express database that resides in the `App_Data` folder? Rather than having to discover what the database and server names are, I ve found that the simplest approach is to attach the database to the `localhost\SQLExpress` database instance and rename the data using SQL Server Management Studio. If you have one of the full versions of SQL Server 2005 installed on your machine, then you likely already have SQL Server Management Studio installed on your computer. If you only have the Express edition, you can download the free Microsoft SQL Server Management Studio Express Edition.

Start by closing Visual Studio. Next, open SQL Server Management Studio and choose to connect to the `localhost\SQLExpress` server using Windows Authentication.

Figure 1: Attach to the `localhost\SQLExpress` Server

After connecting to the server, Management Studio will show the server and have subfolders for the databases, security, and so forth. Right-click on the Databases folder and choose the Attach option. This will bring up the Attach Databases dialog box (see Figure 2). Click the Add button and select the `NORTHWND.MDF` database folder in your web application s `App_Data` folder.

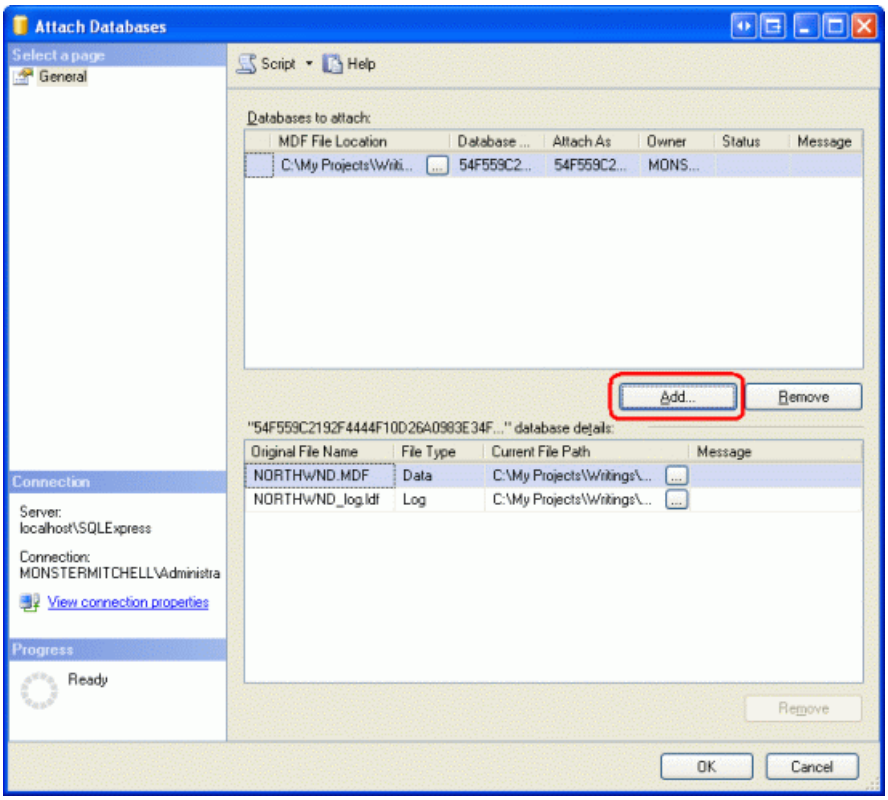


Figure 2: Attach the `NORTHWND.MDF` Database from the `App_Data` Folder (Click to view full-size image)

This will add the database to the Databases folder. The database name might be the full path to the database file or the full path prepended with a GUID. To avoid having to type in this lengthy database name when using the aspnet_regsql.exe command line tool, rename the database to a more human-friendly name by right-clicking on database just attached and choosing Rename. I ve renamed my database to DataTutorials .

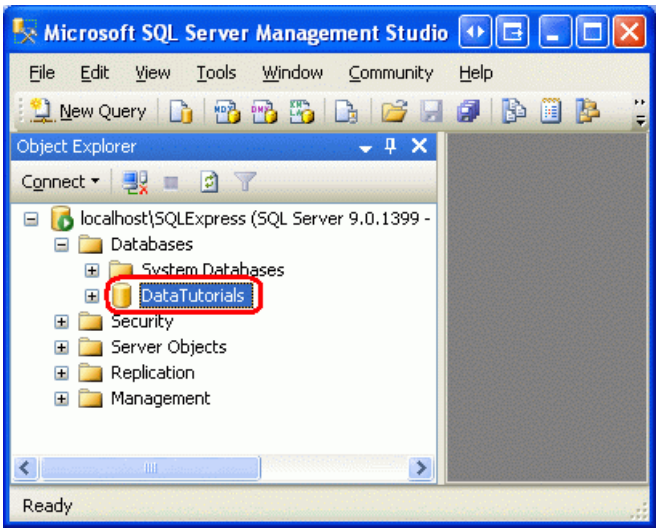


Figure 3: Rename the Attached Database to a More Human-Friendly Name

Step 3: Adding the Polling Infrastructure to the Northwind Database

Now that we have attached the `NORTHWND.MDF` database from the `App_Data` folder, we re ready to add the polling infrastructure. Assuming that you ve renamed the database to DataTutorials, run the following four commands:

```
aspnet_regsql.exe -S localhost\SQLExpress -E -d DataTutorials -ed
aspnet_regsql.exe -S localhost\SQLExpress -E -d DataTutorials -t Products -et
aspnet_regsql.exe -S localhost\SQLExpress -E -d DataTutorials -t Categories -et
aspnet_regsql.exe -S localhost\SQLExpress -E -d DataTutorials -t Suppliers -et
```

After running these four commands, right-click on the database name in Management Studio, go to the Tasks submenu, and choose Detach. Then close Management Studio and reopen Visual Studio.

Once Visual Studio has reopened, drill into the database through the Server Explorer. Note the new table (`AspNet_SqlCacheTablesForChangeNotification`), the new stored procedures, and the triggers on the `Products` , `Categories` , and `Suppliers` tables.

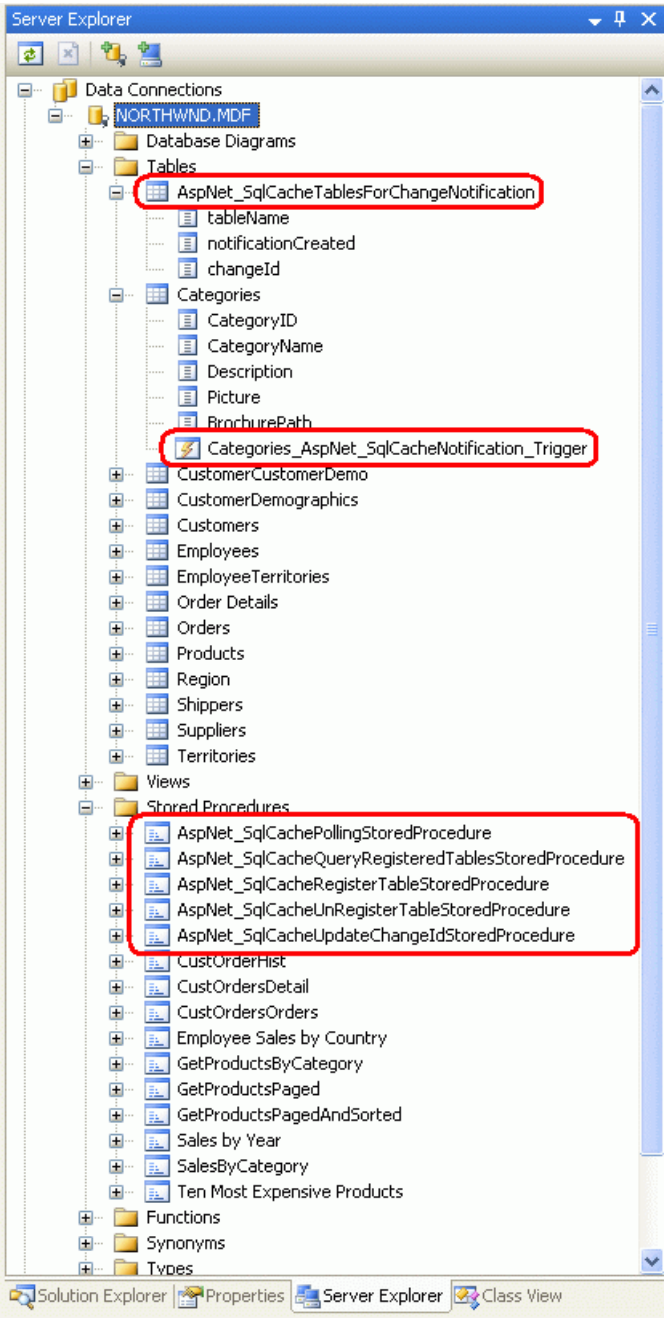


Figure 4: The Database Now Includes the Necessary Polling Infrastructure

Step 4: Configuring the Polling Service

After creating the needed tables, triggers, and stored procedures in the database, the final step is to configure the polling service, which is done through `Web.config` by specifying the databases to use and the polling frequency in milliseconds. The following markup polls the Northwind database once every second.

```
<?xml version="1.0"?>
<configuration>
   <connectionStrings>
      <add name="NORTHWNDConnectionString" connectionString=
          "Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\NORTHWND.MDF;
           Integrated Security=True;User Instance=True"
           providerName="System.Data.SqlClient"/>
   </connectionStrings>
   <system.web>
      ...
      <!-- Configure the polling service used for SQL cache dependencies -->
      <caching>
         <sqlCacheDependency enabled="true" pollTime="1000" >
            <databases>
               <add name="NorthwindDB"
                    connectionStringName="NORTHWNDConnectionString" />
            </databases>
         </sqlCacheDependency>
      </caching>
   </system.web>
</configuration>
```

The `name` value in the `<add>` element (NorthwindDB) associates a human-readable name with a particular database. When working with SQL cache dependencies, we'll need to refer to the database name defined here as well as the table that the cached data is based on. We'll see how to use the `SqlCacheDependency` class to programmatically associate SQL cache dependencies with cached data in Step 6.

Once a SQL cache dependency has been established, the polling system will connect to the databases defined in the `<databases>` elements every `pollTime` milliseconds and execute the `AspNet_SqlCachePollingStoredProcedure` stored procedure. This stored procedure - which was added back in Step 3 using the `aspnet_regsql.exe` command line tool - returns the `tableName` and `changeId` values for each record in `AspNet_SqlCacheTablesForChangeNotification`. Outdated SQL cache dependencies are evicted from the cache.

The `pollTime` setting introduces a tradeoff between performance and data staleness. A small `pollTime` value increases the number of requests to the database, but more quickly evicts stale data from the cache. A larger `pollTime` value reduces the number of database requests, but increases the delay between when the backend data changes and when the related cache items are evicted. Fortunately, the database request is executing a simple stored procedure that s returning just a few rows from a simple, lightweight table. But do experiment with different `pollTime` values to find an ideal balance between database access and data staleness for your application. The smallest `pollTime` value allowed is 500.

> **NOTE**
> The above example provides a single `pollTime` value in the `<sqlCacheDependency>` element, but you can optionally specify the `pollTime` value in the `<add>` element. This is useful if you have multiple databases specified and want to customize the polling frequency per database.

Step 5: Declaratively Working with SQL Cache Dependencies

In Steps 1 through 4 we looked at how to setup the necessary database infrastructure and configure the polling system. With this infrastructure in place, we can now add items to the data cache with an associated SQL cache dependency using either programmatic or declarative techniques. In this step we'll examine how to declaratively work with SQL cache dependencies. In Step 6 we'll look at the programmatic approach.

The Caching Data with the ObjectDataSource tutorial explored the declarative caching capabilities of the ObjectDataSource. By simply setting the `EnableCaching` property to `true` and the `CacheDuration` property to some time interval, the ObjectDataSource will automatically cache the data returned from its underlying object

for the specified interval. The ObjectDataSource can also use one or more SQL cache dependencies.

To demonstrate using SQL cache dependencies declaratively, open the `SqlCacheDependencies.aspx` page in the `Caching` folder and drag a GridView from the Toolbox onto the Designer. Set the GridView s `ID` to `ProductsDeclarative` and, from its smart tag, choose to bind it to a new ObjectDataSource named `ProductsDataSourceDeclarative`.

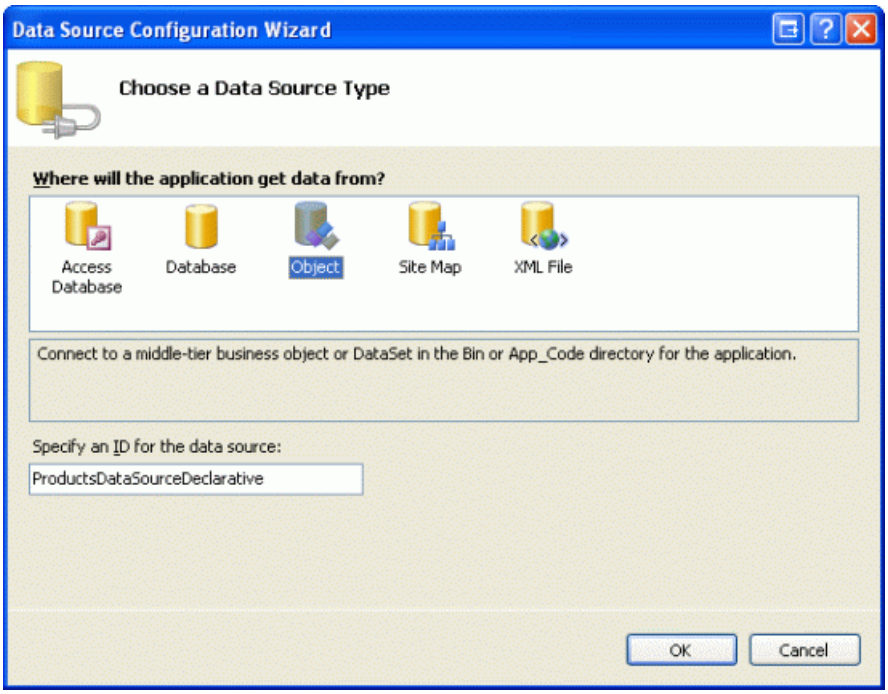


Figure 5: Create a New ObjectDataSource Named `ProductsDataSourceDeclarative` (Click to view full-size image)

Configure the ObjectDataSource to use the `ProductsBLL` class and set the drop-down list in the SELECT tab to `GetProducts()`. In the UPDATE tab, choose the `UpdateProduct` overload with three input parameters - `productName`, `unitPrice`, and `productID`. Set the drop-down lists to (None) in the INSERT and DELETE tabs.

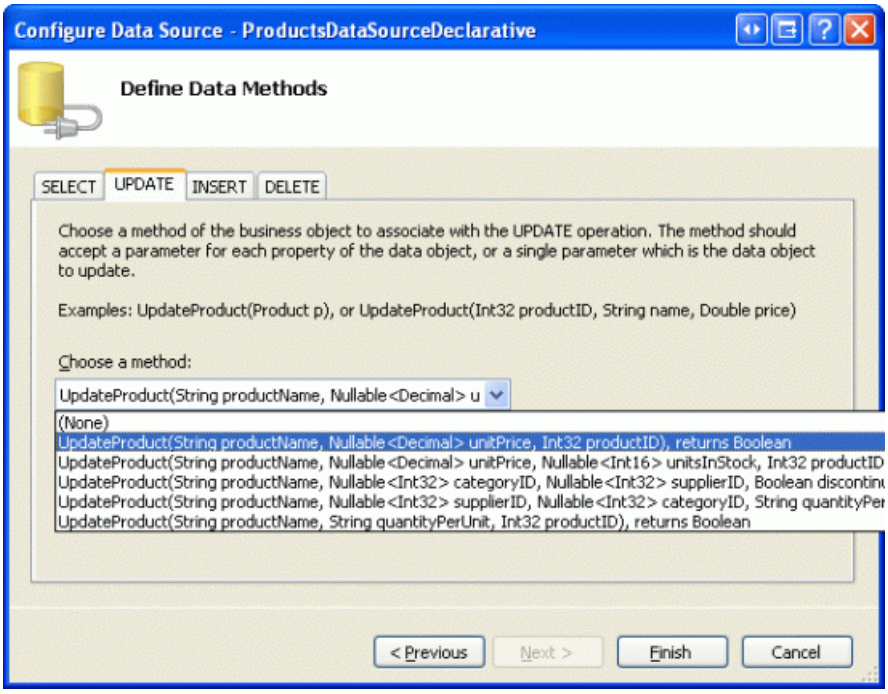


Figure 6: Use the UpdateProduct Overload with Three Input Parameters (Click to view full-size image)

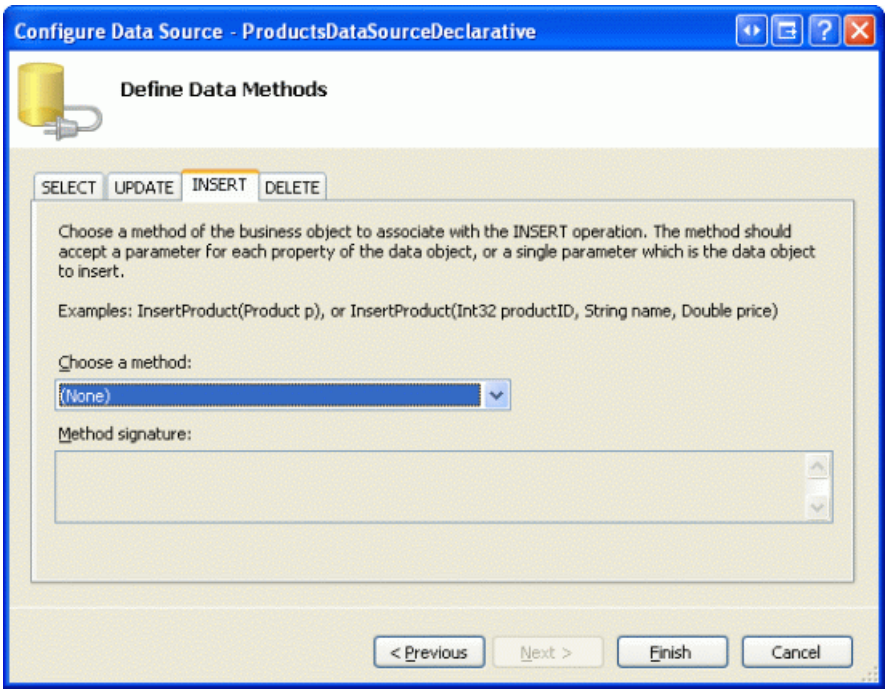


Figure 7: Set the Drop-Down List to (None) for the INSERT and DELETE Tabs (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will create BoundFields and CheckBoxFields in the GridView for each of the data fields. Remove all fields but `ProductName`, `CategoryName`, and `UnitPrice`, and format these fields as you see fit. From the GridView s smart tag, check the Enable Paging, Enable Sorting, and Enable Editing checkboxes. Visual Studio will set the ObjectDataSource s `OldValuesParameterFormatString` property to `original_{0}`. In order for the GridView s edit feature to work properly, either remove this property entirely from the declarative syntax or set it back to its default value, `{0}`.

Finally, add a Label Web control above the GridView and set its `ID` property to `ODSEvents` and its `EnableViewState` property to `false`. After making these changes, your page s declarative markup should look similar to the following. Note that I ve made a number of aesthetic customizations to the GridView fields that are not necessary to demonstrate the SQL cache dependency functionality.

```
<asp:Label ID="ODSEvents" runat="server" EnableViewState="False" />
<asp:GridView ID="ProductsDeclarative" runat="server"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ProductsDataSourceDeclarative"
    AllowPaging="True" AllowSorting="True">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
            <EditItemTemplate>
                <asp:TextBox ID="ProductName" runat="server"
                    Text='<%# Bind("ProductName") %>' />
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                    ControlToValidate="ProductName" Display="Dynamic"
                    ErrorMessage="You must provide a name for the product."
                    SetFocusOnError="True"
                    runat="server">*</asp:RequiredFieldValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Bind("ProductName") %>' />
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:TemplateField HeaderText="Price" SortExpression="UnitPrice">
            <EditItemTemplate>
                $<asp:TextBox ID="UnitPrice" runat="server" Columns="8"
                    Text='<%# Bind("UnitPrice", "{0:N2}") %>'></asp:TextBox>
                <asp:CompareValidator ID="CompareValidator1" runat="server"
                    ControlToValidate="UnitPrice"
                    ErrorMessage="You must enter a valid currency value with
                        no currency symbols. Also, the value must be greater than
                        or equal to zero."
                    Operator="GreaterThanEqual" SetFocusOnError="True"
                    Type="Currency" Display="Dynamic"
                    ValueToCompare="0">*</asp:CompareValidator>
            </EditItemTemplate>
            <ItemStyle HorizontalAlign="Right" />
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("UnitPrice", "{0:c}") %>' />
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSourceDeclarative" runat="server"
    SelectMethod="GetProducts" TypeName="ProductsBLL"
    UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

Next, create an event handler for the ObjectDataSource s `Selecting` event and in it add the following code:

```
protected void ProductsDataSourceDeclarative_Selecting
    (object sender, ObjectDataSourceSelectingEventArgs e)
{
    ODSEvents.Text = "-- Selecting event fired";
}
```

Recall that the ObjectDataSource s `Selecting` event fires only when retrieving data from its underlying object. If

the ObjectDataSource accesses the data from its own cache, this event is not fired.

Now, visit this page through a browser. Since we ve yet to implement any caching, each time you page, sort, or edit the grid the page should display the text, 'Selecting event fired, as Figure 8 shows.

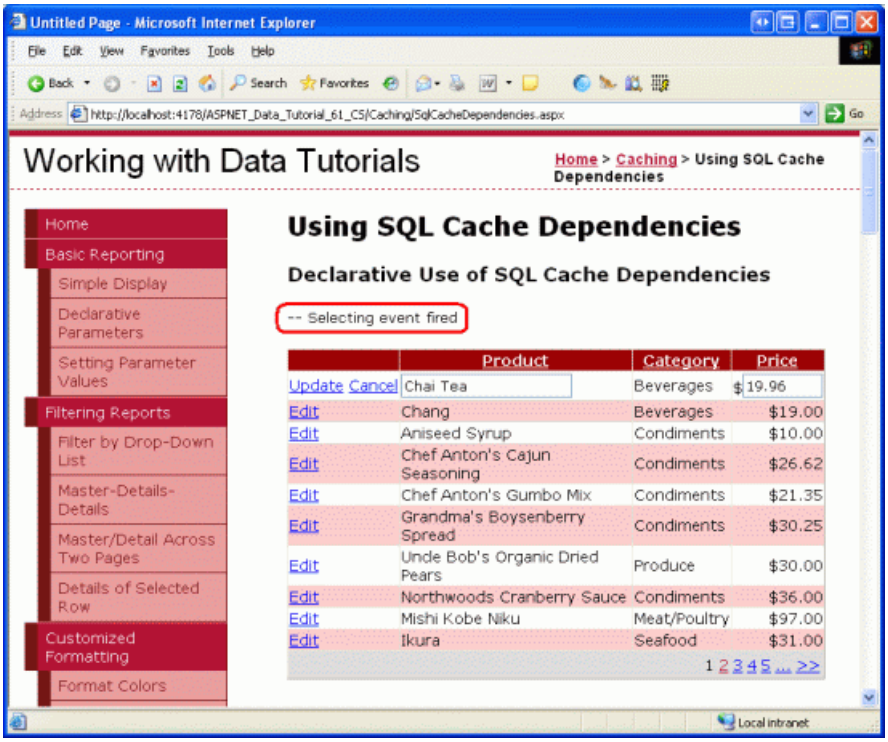


Figure 8: The ObjectDataSource s `Selecting` Event Fires Each Time the GridView is Paged, Edited, or Sorted (Click to view full-size image)

As we saw in the Caching Data with the ObjectDataSource tutorial, setting the `EnableCaching` property to `true` causes the ObjectDataSource to cache its data for the duration specified by its `CacheDuration` property. The ObjectDataSource also has a `SqlCacheDependency` property, which adds one or more SQL cache dependencies to the cached data using the pattern:

```
databaseName1:tableName1;databaseName2:tableName2;...
```

Where *databaseName* is the name of the database as specified in the `name` attribute of the `<add>` element in `Web.config`, and *tableName* is the name of the database table. For example, to create an ObjectDataSource that caches data indefinitely based on a SQL cache dependency against the Northwind s `Products` table, set the ObjectDataSource s `EnableCaching` property to `true` and its `SqlCacheDependency` property to NorthwindDB:Products .

> **NOTE**
>
> You can use a SQL cache dependency *and* a time-based expiry by setting `EnableCaching` to `true`, `CacheDuration` to the time interval, and `SqlCacheDependency` to the database and table name(s). The ObjectDataSource will evict its data when the time-based expiry is reached or when the polling system notes that the underlying database data has changed, whichever happens first.

The GridView in `SqlCacheDependencies.aspx` displays data from two tables - `Products` and `Categories` (the product s `CategoryName` field is retrieved via a `JOIN` on `Categories`). Therefore, we want to specify two SQL cache dependencies: NorthwindDB:Products;NorthwindDB:Categories .

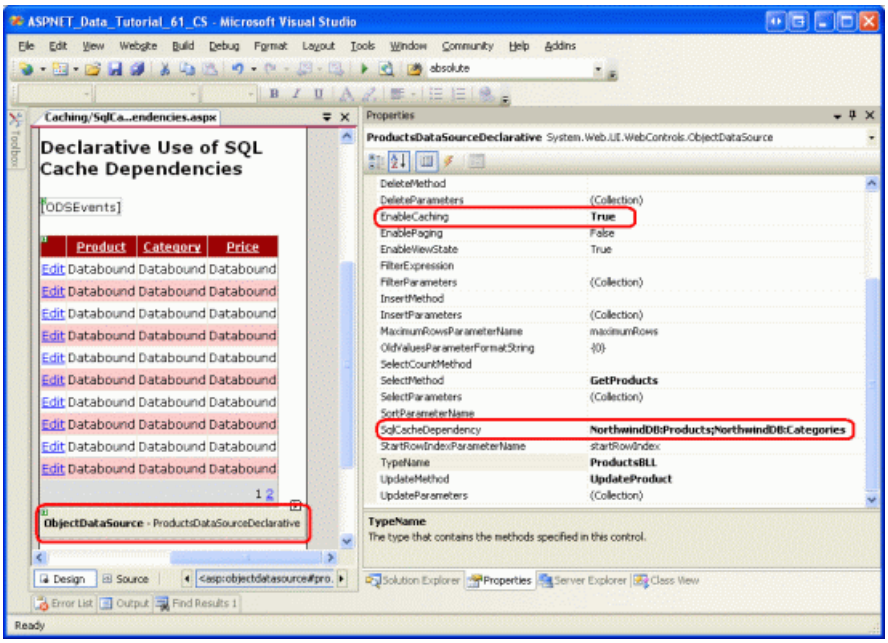


Figure 9: Configure the ObjectDataSource to Support Caching Using SQL Cache Dependencies on `Products` and `Categories` (Click to view full-size image)

After configuring the ObjectDataSource to support caching, revisit the page through a browser. Again, the text 'Selecting event fired should appear on the first page visit, but should go away when paging, sorting, or clicking the Edit or Cancel buttons. This is because after the data is loaded into the ObjectDataSource s cache, it remains there until the `Products` or `Categories` tables are modified or the data is updated through the GridView.

After paging through the grid and noting the lack of the 'Selecting event fired text, open a new browser window and navigate to the Basics tutorial in the Editing, Inserting, and Deleting section (`~/EditInsertDelete/Basics.aspx`). Update the name or price of a product. Then, from to the first browser window, view a different page of data, sort the grid, or click a row s Edit button. This time, the 'Selecting event fired should reappear, as the underlying database data has been modified (see Figure 10). If the text does not appear, wait a few moments and try again. Remember that the polling service is checking for changes to the `Products` table every `pollTime` milliseconds, so there is a delay between when the underlying data is updated and when the cached data is evicted.

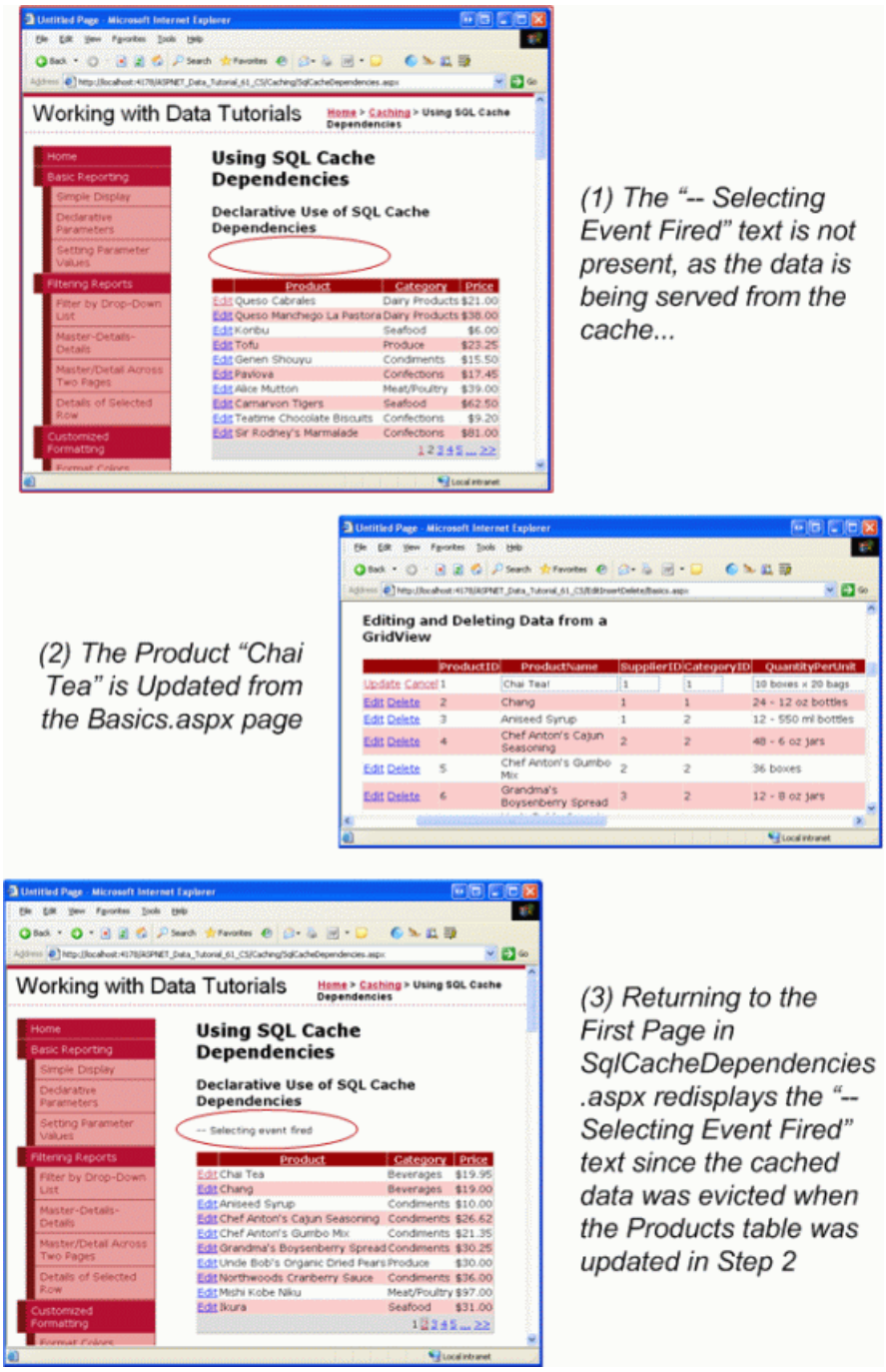


Figure 10: Modifying the Products Table Evicts the Cached Product Data (Click to view full-size image)

Step 6: Programmatically Working with the `SqlCacheDependency` Class

The Caching Data in the Architecture tutorial looked at the benefits of using a separate Caching Layer in the architecture as opposed to tightly coupling the caching with the ObjectDataSource. In that tutorial we created a `ProductsCL` class to demonstrate programmatically working with the data cache. To utilize SQL cache dependencies in the Caching Layer, use the `SqlCacheDependency` class.

With the polling system, a `SqlCacheDependency` object must be associated with a particular database and table pair. The following code, for example, creates a `SqlCacheDependency` object based on the Northwind database s `Products` table:

```
Caching.SqlCacheDependency productsTableDependency =
    new Caching.SqlCacheDependency("NorthwindDB", "Products");
```

The two input parameters to the `SqlCacheDependency` s constructor are the database and table names,

respectively. Like with the ObjectDataSource s `SqlCacheDependency` property, the database name used is the same as the value specified in the `name` attribute of the `<add>` element in `Web.config`. The table name is the actual name of the database table.

To associate a `SqlCacheDependency` with an item added to the data cache, use one of the `Insert` method overloads that accepts a dependency. The following code adds *value* to the data cache for an indefinite duration, but associates it with a `SqlCacheDependency` on the `Products` table. In short, *value* will remain in the cache until it is evicted due to memory constraints or because the polling system has detected that the `Products` table has changed since it was cached.

```
Caching.SqlCacheDependency productsTableDependency =
    new Caching.SqlCacheDependency("NorthwindDB", "Products");
Cache.Insert(key,
             value,
             productsTableDependency,
             System.Web.Caching.Cache.NoAbsoluteExpiration,
             System.Web.Caching.Cache.NoSlidingExpiration);
```

The Caching Layer s `ProductsCL` class currently caches data from the `Products` table using a time-based expiry of 60 seconds. Let s update this class so that it uses SQL cache dependencies instead. The `ProductsCL` class s `AddCacheItem` method, which is responsible for adding the data to the cache, currently contains the following code:

```
private void AddCacheItem(string rawKey, object value)
{
    System.Web.Caching.Cache DataCache = HttpRuntime.Cache;
    // Make sure MasterCacheKeyArray[0] is in the cache
    DataCache[MasterCacheKeyArray[0]] = DateTime.Now;
    // Add a CacheDependency
    Caching.CacheDependency dependency =
        new Caching.CacheDependency(null, MasterCacheKeyArray);
    DataCache.Insert(GetCacheKey(rawKey), value, dependency,
        DateTime.Now.AddSeconds(CacheDuration),
        System.Web.Caching.Cache.NoSlidingExpiration);
}
```

Update this code to use a `SqlCacheDependency` object instead of the `MasterCacheKeyArray` cache dependency:

```
private void AddCacheItem(string rawKey, object value)
{
    System.Web.Caching.Cache DataCache = HttpRuntime.Cache;
    // Add the SqlCacheDependency objects for Products
    Caching.SqlCacheDependency productsTableDependency =
        new Caching.SqlCacheDependency("NorthwindDB", "Products");
    // Add the item to the data cache using productsTableDependency
    DataCache.Insert(GetCacheKey(rawKey), value, productsTableDependency,
        Caching.Cache.NoAbsoluteExpiration, Caching.Cache.NoSlidingExpiration);
}
```

To test this functionality, add a GridView to the page beneath the existing `ProductsDeclarative` GridView. Set this new GridView s `ID` to `ProductsProgrammatic` and, through its smart tag, bind it to a new ObjectDataSource named `ProductsDataSourceProgrammatic`. Configure the ObjectDataSource to use the `ProductsCL` class, setting the drop-down lists in the SELECT and UPDATE tabs to `GetProducts` and `UpdateProduct`, respectively.

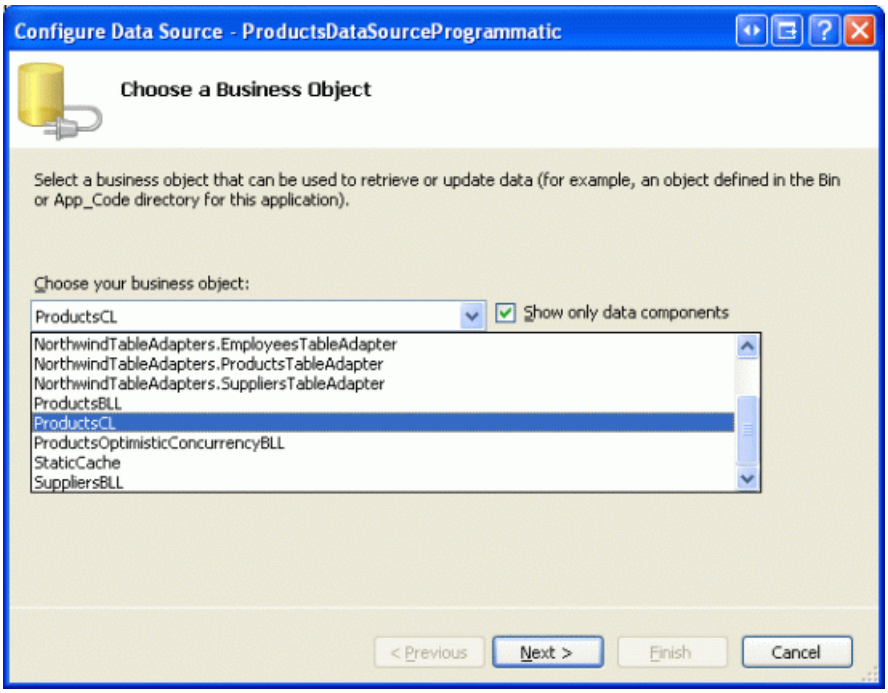


Figure 11: Configure the ObjectDataSource to Use the `ProductsCL` Class (Click to view full-size image)

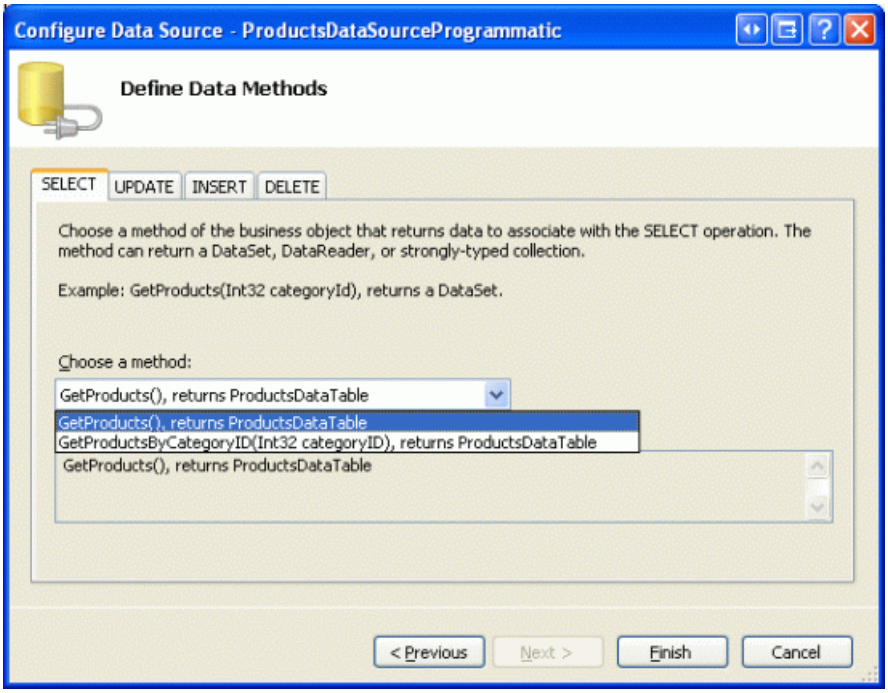


Figure 12: Select the `GetProducts` Method from the SELECT Tab s Drop-Down List (Click to view full-size image)

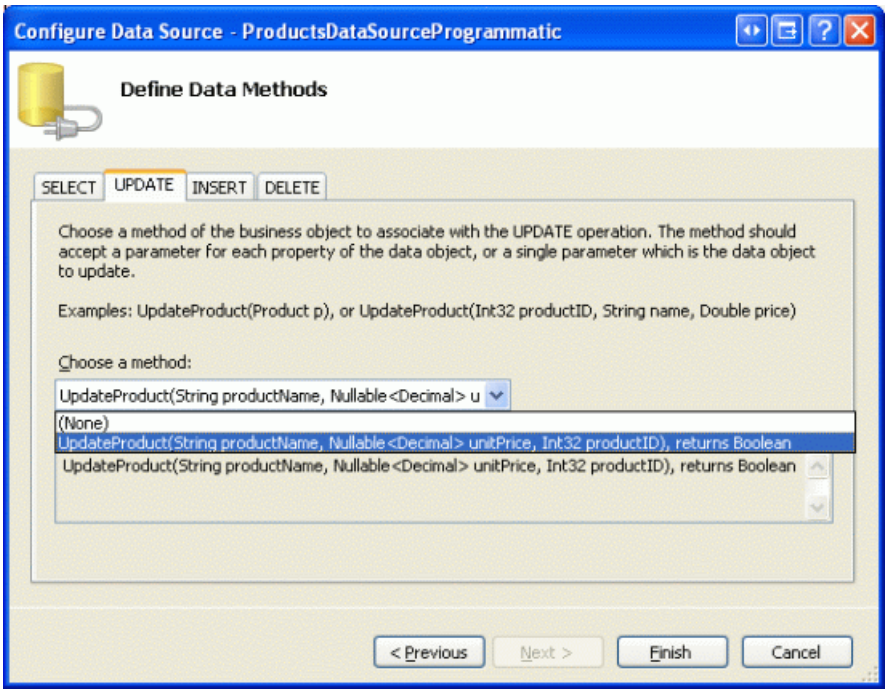


Figure 13: Choose the UpdateProduct Method from the UPDATE Tab s Drop-Down List (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will create BoundFields and CheckBoxFields in the GridView for each of the data fields. Like with the first GridView added to this page, remove all fields but `ProductName`, `CategoryName`, and `UnitPrice`, and format these fields as you see fit. From the GridView s smart tag, check the Enable Paging, Enable Sorting, and Enable Editing checkboxes. As with the `ProductsDataSourceDeclarative` ObjectDataSource, Visual Studio will set the `ProductsDataSourceProgrammatic` ObjectDataSource s `OldValuesParameterFormatString` property to `original_{0}`. In order for the GridView s edit feature to work properly, set this property back to `{0}` (or remove the property assignment from the declarative syntax altogether).

After completing these tasks, the resulting GridView and ObjectDataSource declarative markup should look like the following:

```
<asp:GridView ID="ProductsProgrammatic" runat="server"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ProductsDataSourceProgrammatic" AllowPaging="True"
    AllowSorting="True">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
            <EditItemTemplate>
                <asp:TextBox ID="ProductName" runat="server"
                    Text='<%# Bind("ProductName") %>' />
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                    ControlToValidate="ProductName" Display="Dynamic"
                    ErrorMessage="You must provide a name for the product."
                    SetFocusOnError="True"
                    runat="server">*</asp:RequiredFieldValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Bind("ProductName") %>' />
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:TemplateField HeaderText="Price" SortExpression="UnitPrice">
            <EditItemTemplate>
                $<asp:TextBox ID="UnitPrice" runat="server" Columns="8"
                    Text='<%# Bind("UnitPrice", "{0:N2}") %>'></asp:TextBox>
                <asp:CompareValidator ID="CompareValidator1" runat="server"
                    ControlToValidate="UnitPrice" Display="Dynamic"
                    ErrorMessage="You must enter a valid currency value with
                        no currency symbols. Also, the value must be greater than
                        or equal to zero."
                    Operator="GreaterThanEqual" SetFocusOnError="True"
                    Type="Currency" ValueToCompare="0">*</asp:CompareValidator>
            </EditItemTemplate>
            <ItemStyle HorizontalAlign="Right" />
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("UnitPrice", "{0:c}") %>' />
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSourceProgrammatic" runat="server"
    OldValuesParameterFormatString="{0}" SelectMethod="GetProducts"
    TypeName="ProductsCL" UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

To test the SQL cache dependency in the Caching Layer set a breakpoint in the `ProductCL` class s `AddCacheItem` method and then start debugging. When you first visit `SqlCacheDependencies.aspx`, the breakpoint should be hit as the data is requested for the first time and placed into the cache. Next, move to another page in the GridView or sort one of the columns. This causes the GridView to requery its data, but the data should be found in the cache since the `Products` database table has not been modified. If the data is repeatedly not found in the cache, make sure there is sufficient memory available on your computer and try again.

After paging through a few pages of the GridView, open a second browser window and navigate to the Basics tutorial in the Editing, Inserting, and Deleting section (`~/EditInsertDelete/Basics.aspx`). Update a record from the Products table and then, from the first browser window, view a new page or click on one of the sorting headers.

In this scenario you will see one of two things: either the breakpoint will be hit, indicating that the cached data was evicted due to the change in the database; or, the breakpoint will not be hit, meaning that `SqlCacheDependencies.aspx` is now showing stale data. If the breakpoint is not hit, it is likely because the polling service has not yet fired since the data was changed. Remember that the polling service is checking for changes to the `Products` table every `pollTime` milliseconds, so there is a delay between when the underlying data is updated and when the cached data is evicted.

> **NOTE**
>
> This delay is more likely to appear when editing one of the products through the GridView in `SqlCacheDependencies.aspx`. In the Caching Data in the Architecture tutorial we added the `MasterCacheKeyArray` cache dependency to ensure that the data being edited through the `ProductsCL` class s `UpdateProduct` method was evicted from the cache. However, we replaced this cache dependency when modifying the `AddCacheItem` method earlier in this step and therefore the `ProductsCL` class will continue to show the cached data until the polling system notes the change to the `Products` table. We'll see how to reintroduce the `MasterCacheKeyArray` cache dependency in Step 7.

Step 7: Associating Multiple Dependencies with a Cached Item

Recall that the `MasterCacheKeyArray` cache dependency is used to ensure that *all* product-related data is evicted from the cache when any single item associated within it is updated. For example, the `GetProductsByCategoryID(categoryID)` method caches `ProductsDataTables` instances for each unique *categoryID* value. If one of these objects is evicted, the `MasterCacheKeyArray` cache dependency ensures that the others are also removed. Without this cache dependency, when the cached data is modified the possibility exists that other cached product data may be out of date. Consequently, it s important that we maintain the `MasterCacheKeyArray` cache dependency when using SQL cache dependencies. However, the data cache s `Insert` method only allows for a single dependency object.

Furthermore, when working with SQL cache dependencies we may need to associate multiple database tables as dependencies. For example, the `ProductsDataTable` cached in the `ProductsCL` class contains the category and supplier names for each product, but the `AddCacheItem` method only uses a dependency on `Products`. In this situation, if the user updates the name of a category or supplier, the cached product data will remain in the cache and be out of date. Therefore, we want to make the cached product data dependent on not only the `Products` table, but on the `Categories` and `Suppliers` tables as well.

The `AggregateCacheDependency` class provides a means for associating multiple dependencies with a cache item. Start by creating an `AggregateCacheDependency` instance. Next, add the set of dependencies using the `AggregateCacheDependency` s `Add` method. When inserting the item into the data cache thereafter, pass in the `AggregateCacheDependency` instance. When *any* of the `AggregateCacheDependency` instance s dependencies change, the cached item will be evicted.

The following shows the updated code for the `ProductsCL` class s `AddCacheItem` method. The method creates the `MasterCacheKeyArray` cache dependency along with `SqlCacheDependency` objects for the `Products`, `Categories`, and `Suppliers` tables. These are all combined into one `AggregateCacheDependency` object named `aggregateDependencies`, which is then passed into the `Insert` method.

```
private void AddCacheItem(string rawKey, object value)
{
    System.Web.Caching.Cache DataCache = HttpRuntime.Cache;
    // Make sure MasterCacheKeyArray[0] is in the cache and create a depedency
    DataCache[MasterCacheKeyArray[0]] = DateTime.Now;
    Caching.CacheDependency masterCacheKeyDependency =
        new Caching.CacheDependency(null, MasterCacheKeyArray);
    // Add the SqlCacheDependency objects for Products, Categories, and Suppliers
    Caching.SqlCacheDependency productsTableDependency =
        new Caching.SqlCacheDependency("NorthwindDB", "Products");
    Caching.SqlCacheDependency categoriesTableDependency =
        new Caching.SqlCacheDependency("NorthwindDB", "Categories");
    Caching.SqlCacheDependency suppliersTableDependency =
        new Caching.SqlCacheDependency("NorthwindDB", "Suppliers");
    // Create an AggregateCacheDependency
    Caching.AggregateCacheDependency aggregateDependencies =
        new Caching.AggregateCacheDependency();
    aggregateDependencies.Add(masterCacheKeyDependency, productsTableDependency,
        categoriesTableDependency, suppliersTableDependency);
    DataCache.Insert(GetCacheKey(rawKey), value, aggregateDependencies,
        Caching.Cache.NoAbsoluteExpiration, Caching.Cache.NoSlidingExpiration);
}
```

Test this new code out. Now changes to the `Products`, `Categories`, or `Suppliers` tables cause the cached data to be evicted. Moreover, the `ProductsCL` class s `UpdateProduct` method, which is called when editing a product through the GridView, evicts the `MasterCacheKeyArray` cache dependency, which causes the cached `ProductsDataTable` to be evicted and the data to be re-retrieved on the next request.

> **NOTE**
>
> SQL cache dependencies can also be used with output caching. For a demonstration of this functionality, see: Using ASP.NET Output Caching with SQL Server.

Summary

When caching database data, the data will ideally remain in the cache until it is modified in the database. With ASP.NET 2.0, SQL cache dependencies can be created and used in both declarative and programmatic scenarios. One of the challenges with this approach is in discovering when the data has been modified. The full versions of Microsoft SQL Server 2005 provide notification capabilities that can alert an application when a query result has changed. For the Express Edition of SQL Server 2005 and older versions of SQL Server, a polling system must be used instead. Fortunately, setting up the necessary polling infrastructure is fairly straightforward.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Using Query Notifications in Microsoft SQL Server 2005
- Creating a Query Notification
- Caching in ASP.NET with the `SqlCacheDependency` Class
- ASP.NET SQL Server Registration Tool (`aspnet_regsql.exe`)
- Overview of `SqlCacheDependency`

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Marko Rangel, Teresa Murphy, and Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Caching Data with the ObjectDataSource (VB)

2/20/2020 • 20 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

Caching can mean the difference between a slow and a fast Web application. This tutorial is the first of four that take a detailed look at caching in ASP.NET. Learn the key concepts of caching and how to apply caching to the Presentation Layer through the ObjectDataSource control.

Introduction

In computer science, *caching* is the process of taking data or information that is expensive to obtain and storing a copy of it in a location that is quicker to access. For data-driven applications, large and complex queries commonly consume the majority of the application s execution time. Such an application s performance, then, can often be improved by storing the results of expensive database queries in the application s memory.

ASP.NET 2.0 offers a variety of caching options. An entire web page or User Control s rendered markup can be cached through *output caching*. The ObjectDataSource and SqlDataSource controls provide caching capabilities as well, thereby allowing data to be cached at the control level. And ASP.NET s *data cache* provides a rich caching API that enables page developers to programmatically cache objects. In this tutorial and the next three we'll examine using the ObjectDataSource s caching features as well as the data cache. We'll also explore how to cache application-wide data at startup and how to keep cached data fresh through the use of SQL cache dependencies. These tutorials do not explore output caching. For a detailed look at output caching, see Output Caching in ASP.NET 2.0.

Caching can be applied at any place in the architecture, from the Data Access Layer up through the Presentation Layer. In this tutorial we'll look at applying caching to the Presentation Layer through the ObjectDataSource control. In the next tutorial we'll examine caching data at the Business Logic Layer.

Key Caching Concepts

Caching can greatly improve an application s overall performance and scalability by taking data that is expensive to generate and storing a copy of it in a location that can be more efficiently accessed. Since the cache holds just a copy of the actual, underlying data, it can become outdated, or *stale*, if the underlying data changes. To combat this, a page developer can indicate criteria by which the cache item will be *evicted* from the cache, using either:

- **Time-based criteria** an item may be added to the cache for an absolute or sliding duration. For example, a page developer may indicate a duration of, say, 60 seconds. With an absolute duration, the cached item is evicted 60 seconds after it was added to cache, regardless of how frequently it was accessed. With a sliding duration, the cached item is evicted 60 seconds after the last access.
- **Dependency-based criteria** a dependency can be associated with an item when added to the cache. When the item s dependency changes it is evicted from the cache. The dependency may be a file, another cache item, or a combination of the two. ASP.NET 2.0 also allows SQL cache dependencies, which enable developers to add an item to the cache and have it evicted when the underlying database data changes. We will examine SQL cache dependencies in the upcoming Using SQL Cache Dependencies tutorial.

Regardless of the eviction criteria specified, an item in the cache may be *scavenged* before the time-based or dependency-based criteria has been met. If the cache has reached its capacity, existing items must be removed before new ones can be added. Consequently, when programmatically working with cached data it s vital that you

always assume that the cached data may not be present. We'll look at the pattern to use when accessing data from the cache programmatically in our next tutorial, *Caching Data in the Architecture*.

Caching provides an economical means for squeezing more performance from an application. As Steven Smith articulates in his article ASP.NET Caching: Techniques and Best Practices:

Caching can be a good way to get good enough performance without requiring a lot of time and analysis. Memory is cheap, so if you can get the performance you need by caching the output for 30 seconds instead of spending a day or a week trying to optimize your code or database, do the caching solution (assuming 30-second old data is ok) and move on. Eventually, poor design will probably catch up to you, so of course you should try to design your applications correctly. But if you just need to get good enough performance today, caching can be an excellent [approach], buying you time to refactor your application at a later date when you have the time to do so.

While caching can provide appreciable performance enhancements, it is not applicable in all situations, such as with applications that use real-time, frequently-updating data, or where even shortly-lived stale data is unacceptable. But for the majority of applications, caching should be used. For more background on caching in ASP.NET 2.0, refer to the Caching for Performance section of the ASP.NET 2.0 QuickStart Tutorials.

Step 1: Creating the Caching Web Pages

Before we start our exploration of the ObjectDataSource s caching features, let s first take a moment to create the ASP.NET pages in our website project that we'll need for this tutorial and the next three. Start by adding a new folder named `Caching`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `ObjectDataSource.aspx`
- `FromTheArchitecture.aspx`
- `AtApplicationStartup.aspx`
- `SqlCacheDependencies.aspx`

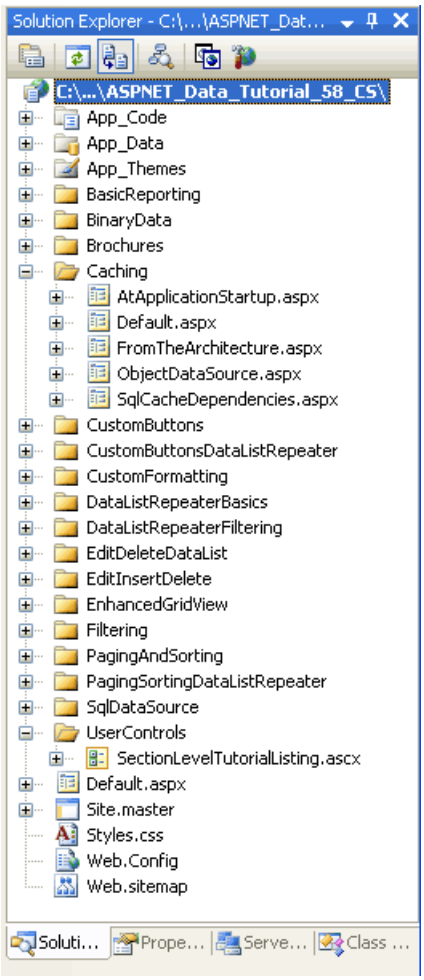


Figure 1: Add the ASP.NET Pages for the Caching-Related Tutorials

Like in the other folders, `Default.aspx` in the `Caching` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

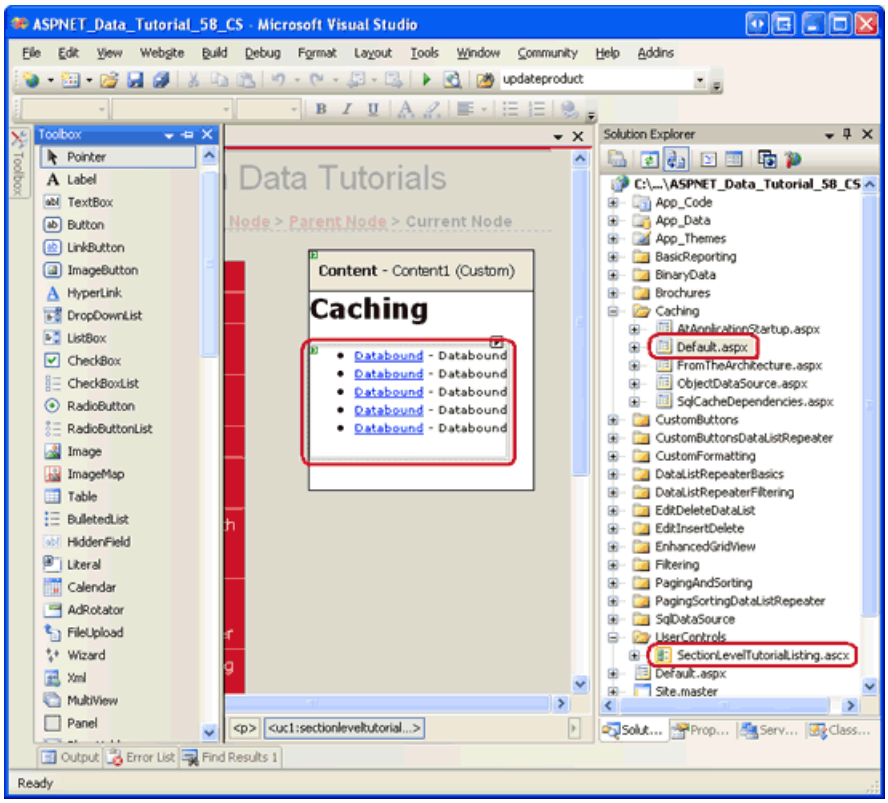


Figure 2: Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-

size image)

Lastly, add these pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Working with Binary Data `<siteMapNode>`:

```
<siteMapNode title="Caching" url="~/Caching/Default.aspx"
    description="Learn how to use the caching features of ASP.NET 2.0.">
    <siteMapNode url="~/Caching/ObjectDataSource.aspx"
        title="ObjectDataSource Caching"
        description="Explore how to cache data directly from the
            ObjectDataSource control." />
    <siteMapNode url="~/Caching/FromTheArchitecture.aspx"
        title="Caching in the Architecture"
        description="See how to cache data from within the
            architecture." />
    <siteMapNode url="~/Caching/AtApplicationStartup.aspx"
        title="Caching Data at Application Startup"
        description="Learn how to cache expensive or infrequently-changing
            queries at the start of the application." />
    <siteMapNode url="~/Caching/SqlCacheDependencies.aspx"
        title="Using SQL Cache Dependencies"
        description="Examine how to have data automatically expire from the
            cache when its underlying database data is modified." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the caching tutorials.

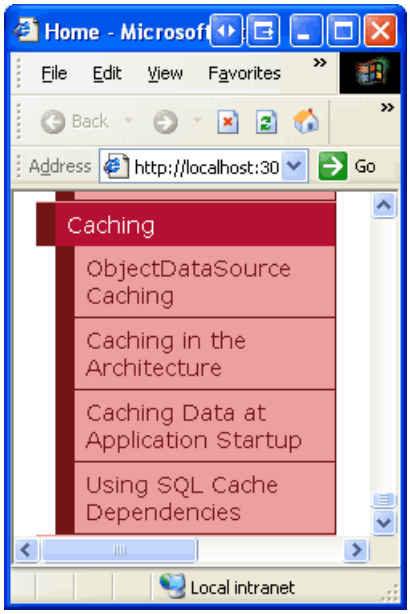


Figure 3: The Site Map Now Includes Entries for the Caching Tutorials

Step 2: Displaying a List of Products in a Web Page

This tutorial explores how to use the ObjectDataSource control s built-in caching features. Before we can look at these features, though, we first need a page to work from. Let s create a web page that uses a GridView to list product information retrieved by an ObjectDataSource from the `ProductsBLL` class.

Start by opening the `ObjectDataSource.aspx` page in the `Caching` folder. Drag a GridView from the Toolbox onto the Designer, set its `ID` property to `Products`, and, from its smart tag, choose to bind it to a new ObjectDataSource control named `ProductsDataSource`. Configure the ObjectDataSource to work with the `ProductsBLL` class.

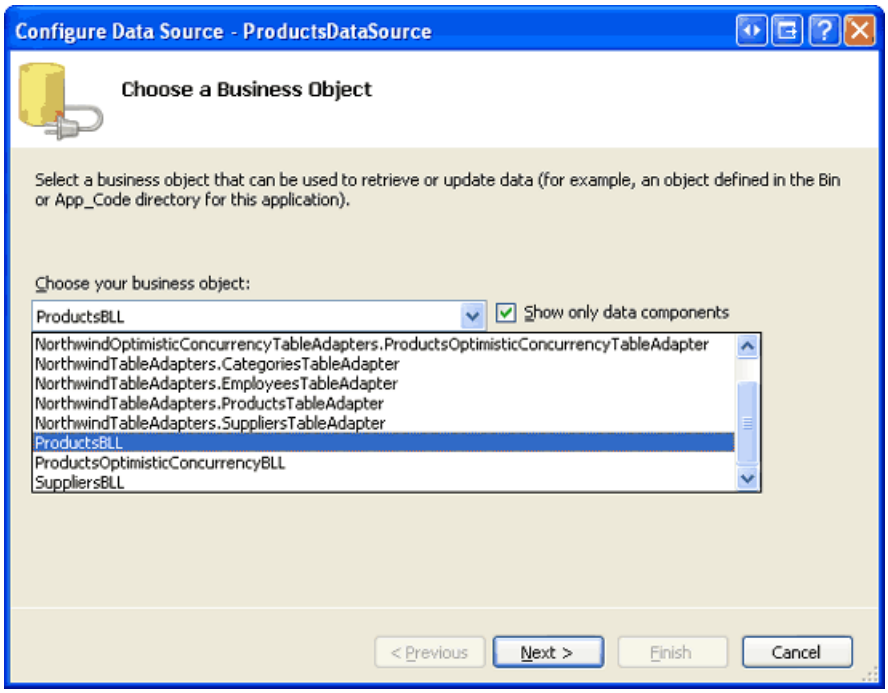


Figure 4: Configure the ObjectDataSource to Use the `ProductsBLL` Class (Click to view full-size image)

For this page, let s create an editable GridView so that we can examine what happens when data cached in the ObjectDataSource is modified through the GridView s interface. Leave the drop-down list in the SELECT tab set to its default, `GetProducts()`, but change the selected item in the UPDATE tab to the `UpdateProduct` overload that accepts `productName`, `unitPrice`, and `productID` as its input parameters.

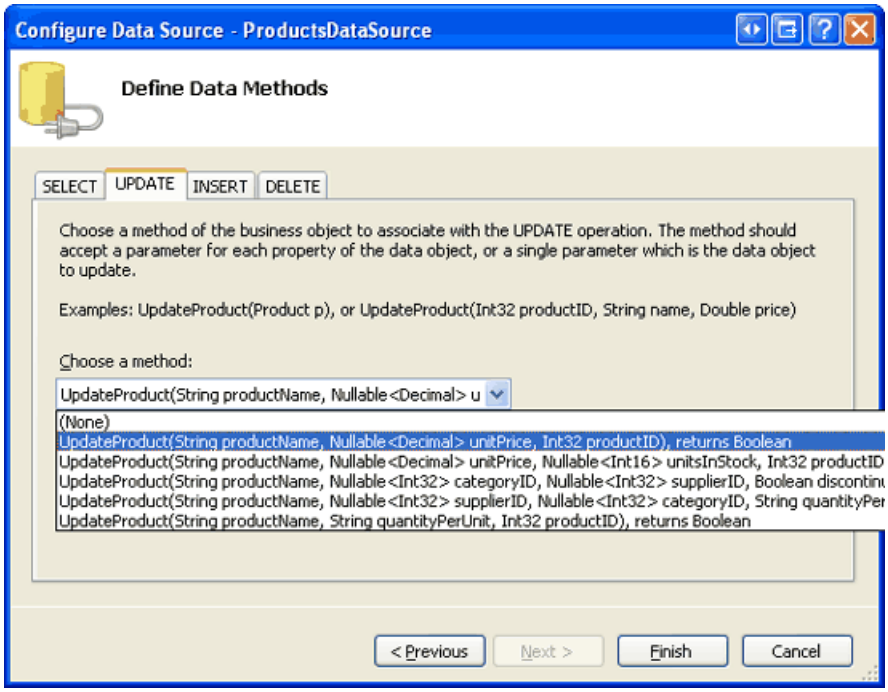


Figure 5: Set the UPDATE Tab s Drop-Down List to the Appropriate `UpdateProduct` Overload (Click to view full-size image)

Finally, set the drop-down lists in the INSERT and DELETE tabs to (None) and click Finish. Upon completing the Configure Data Source wizard, Visual Studio sets the ObjectDataSource s `OldValuesParameterFormatString` property to `original_{0}`. As discussed in the An Overview of Inserting, Updating, and Deleting Data tutorial, this property needs to be removed from the declarative syntax or set back to its default value, `{0}`, in order for our update workflow to proceed without error.

Furthermore, at the completion of the wizard Visual Studio adds a field to the GridView for each of the product data fields. Remove all but the `ProductName`, `CategoryName`, and `UnitPrice` BoundFields. Next, update the

`HeaderText` properties of each of these BoundFields to Product, Category, and Price, respectively. Since the `ProductName` field is required, convert the BoundField into a TemplateField and add a RequiredFieldValidator to the `EditItemTemplate`. Similarly, convert the `UnitPrice` BoundField into a TemplateField and add a CompareValidator to ensure that the value entered by the user is a valid currency value that s greater than or equal to zero. In addition to these modifications, feel free to perform any aesthetic changes, such as right-aligning the `UnitPrice` value, or specifying the formatting for the `UnitPrice` text in its read-only and editing interfaces.

Make the GridView editable by checking the Enable Editing checkbox in the GridView s smart tag. Also check the Enable Paging and Enable Sorting checkboxes.

> **NOTE**
>
> Need a review of how to customize the GridView s editing interface? If so, refer back to the Customizing the Data Modification Interface tutorial.

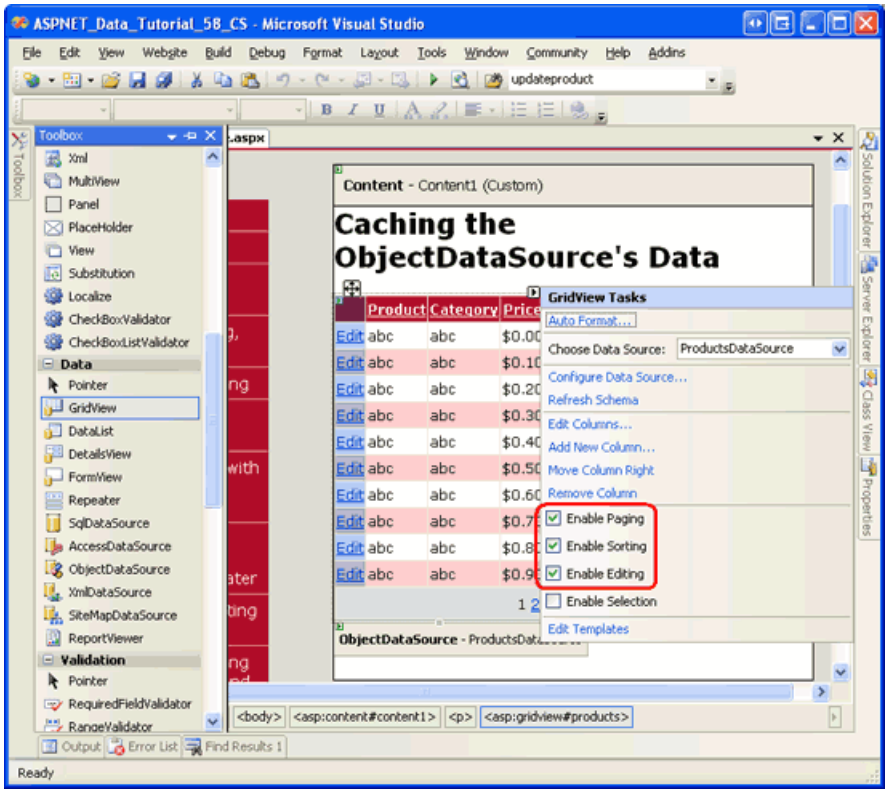


Figure 6: Enable GridView Support for Editing, Sorting, and Paging (Click to view full-size image)

After making these GridView modifications, the GridView and ObjectDataSource s declarative markup should look similar to the following:

```
<asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
    AllowPaging="True" AllowSorting="True">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
            <EditItemTemplate>
                <asp:TextBox ID="ProductName" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:TextBox>
                <asp:RequiredFieldValidator
                    ID="RequiredFieldValidator1" Display="Dynamic"
                    ControlToValidate="ProductName" SetFocusOnError="True"
                    ErrorMessage="You must provide a name for the product."
                    runat="server">*</asp:RequiredFieldValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:TemplateField HeaderText="Price" SortExpression="UnitPrice">
            <EditItemTemplate>
                $<asp:TextBox ID="UnitPrice" runat="server" Columns="8"
                    Text='<%# Bind("UnitPrice", "{0:N2}") %>'></asp:TextBox>
                <asp:CompareValidator ID="CompareValidator1"
                    ControlToValidate="UnitPrice" Display="Dynamic"
                    ErrorMessage="You must enter a valid currency value with no
                        currency symbols. Also, the value must be greater than
                        or equal to zero."
                    Operator="GreaterThanEqual" SetFocusOnError="True"
                    Type="Currency" runat="server"
                    ValueToCompare="0">*</asp:CompareValidator>
            </EditItemTemplate>
            <ItemStyle HorizontalAlign="Right" />
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("UnitPrice", "{0:c}") %>' />
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="{0}" SelectMethod="GetProducts"
    TypeName="ProductsBLL" UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

As Figure 7 shows, the editable GridView lists the name, category, and price of each of the products in the database. Take a moment to test out the page s functionality sort the results, page through them, and edit a record.

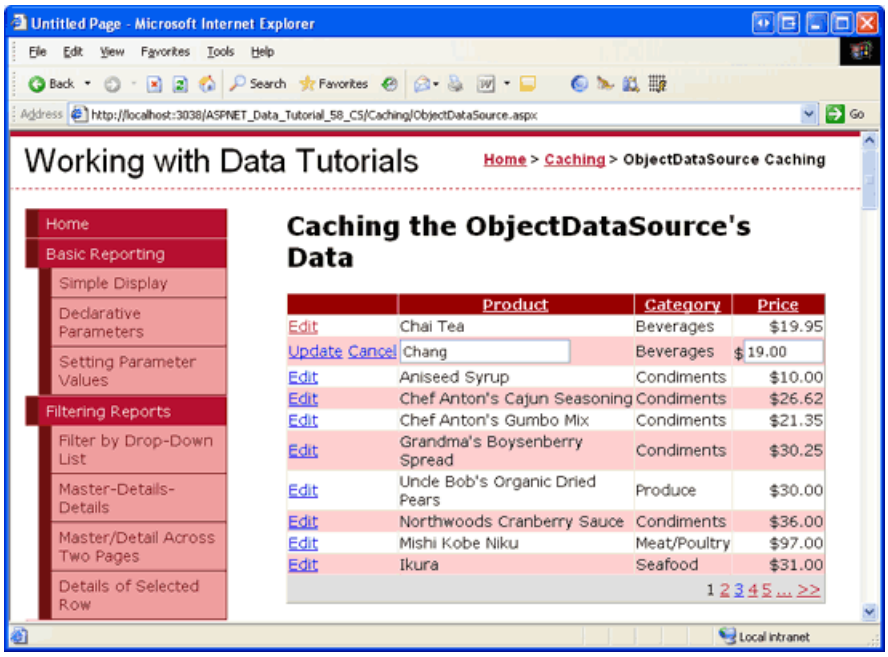


Figure 7: Each Product s Name, Category, and Price is Listed in a Sortable, Pageable, Editable GridView (Click to view full-size image)

Step 3: Examining When the ObjectDataSource is Requesting Data

The `Products` GridView retrieves its data to display by invoking the `Select` method of the `ProductsDataSource` ObjectDataSource. This ObjectDataSource creates an instance of the Business Logic Layer s `ProductsBLL` class and calls its `GetProducts()` method, which in turn calls the Data Access Layer s `ProductsTableAdapter` s `GetProducts()` method. The DAL method connects to the Northwind database and issues the configured `SELECT` query. This data is then returned to the DAL, which packages it up in a `NorthwindDataTable`. The DataTable object is returned to the BLL, which returns it to the ObjectDataSource, which returns it to the GridView. The GridView then creates a `GridViewRow` object for each `DataRow` in the DataTable, and each `GridViewRow` is eventually rendered into the HTML that is returned to the client and displayed on the visitor s browser.

This sequence of events happens each and every time the GridView needs to bind to its underlying data. That happens when the page is first visited, when moving from one page of data to another, when sorting the GridView, or when modifying the GridView s data through its built-in editing or deleting interfaces. If the GridView s view state is disabled, the GridView will be rebound on each and every postback as well. The GridView can also be explicitly rebound to its data by calling its `DataBind()` method.

To fully appreciate the frequency with which the data is retrieved from the database, let s display a message indicating when the data is being re-retrieved. Add a Label Web control above the GridView named `ODSEvents`. Clear out its `Text` property and set its `EnableViewState` property to `False`. Underneath the Label, add a Button Web control and set its `Text` property to Postback .

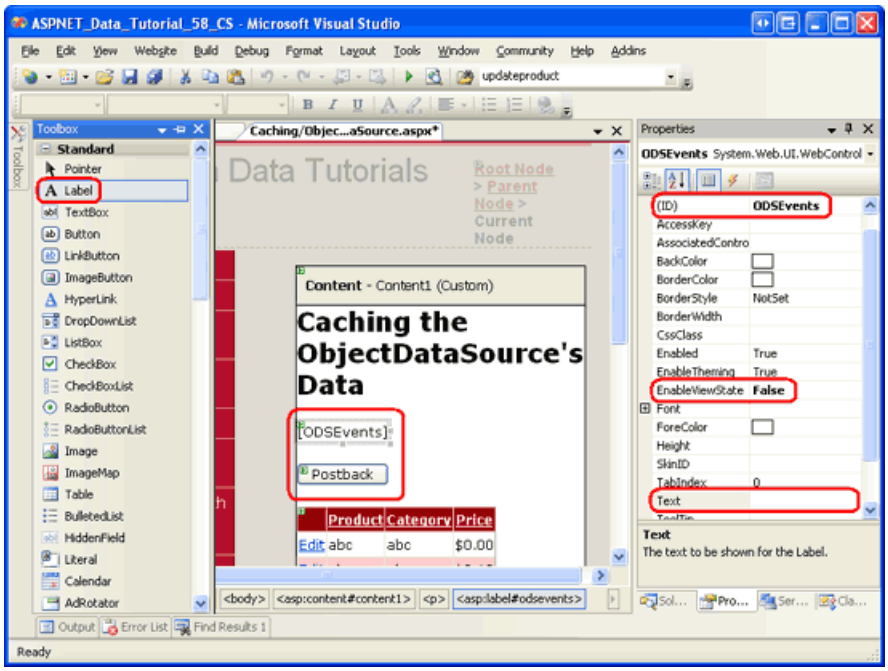


Figure 8: Add a Label and Button to the Page Above the GridView (Click to view full-size image)

During the data access workflow, the ObjectDataSource s `Selecting` event fires before the underlying object is created and its configured method invoked. Create an event handler for this event and add the following code:

```
Protected Sub ProductsDataSource_Selecting(sender As Object, _
    e As ObjectDataSourceSelectingEventArgs) _
    Handles ProductsDataSource.Selecting
    ODSEvents.Text = "-- Selecting event fired"
End Sub
```

Each time the ObjectDataSource makes a request to the architecture for data, the Label will display the text Selecting event fired .

Visit this page in a browser. When the page is first visited, the text Selecting event fired is shown. Click the Postback button and note that the text disappears (assuming that the GridView s `EnableViewState` property is set to `True` , the default). This is because, on postback, the GridView is reconstructed from its view state and therefore doesn t turn to the ObjectDataSource for its data. Sorting, paging, or editing the data, however, causes the GridView to rebind to its data source, and therefore the Selecting event fired text reappears.

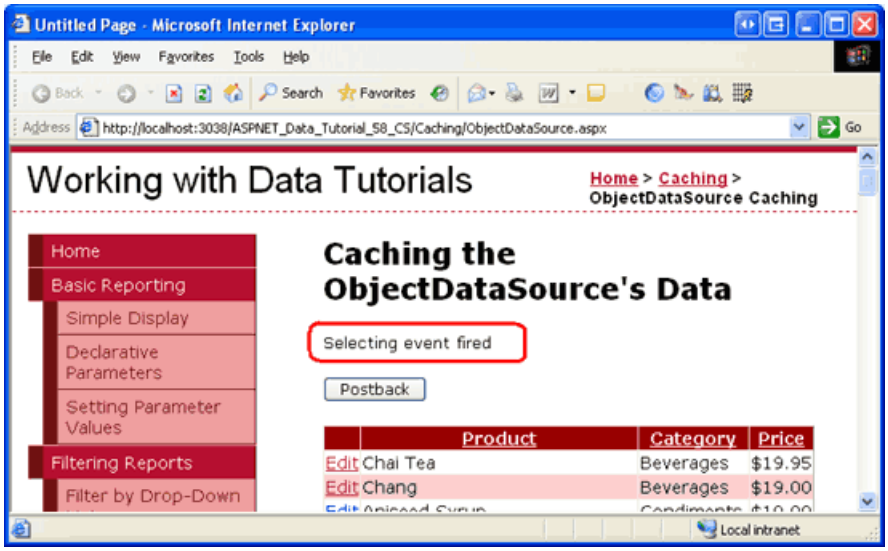


Figure 9: Whenever the GridView is Rebound to its Data Source, Selecting event fired is Displayed (Click to view full-size image)

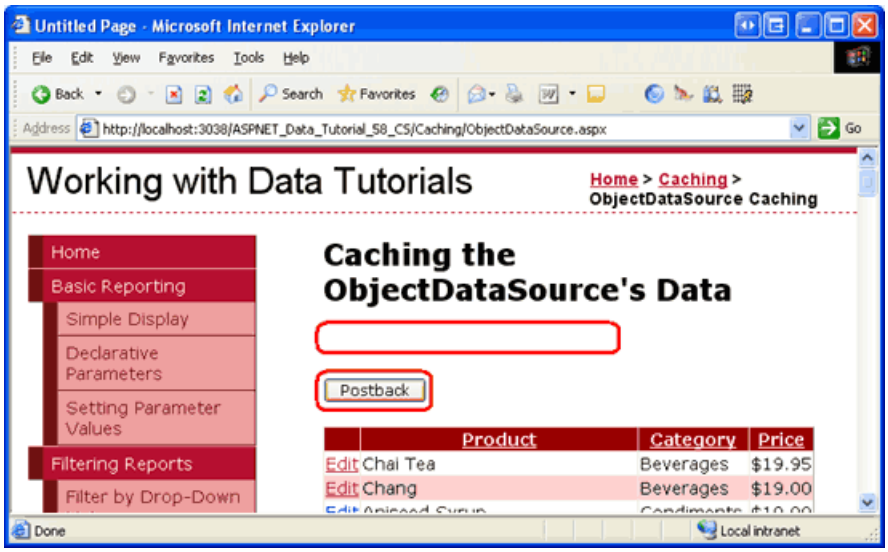


Figure 10: Clicking the Postback Button Causes the GridView to be Reconstructed from its View State (Click to view full-size image)

It may seem wasteful to retrieve the database data each time the data is paged through or sorted. After all, since we re using default paging, the ObjectDataSource has retrieved all of the records when displaying the first page. Even if the GridView does not provide sorting and paging support, the data must be retrieved from the database each time the page is first visited by any user (and on every postback, if view state is disabled). But if the GridView is showing the same data to all users, these extra database requests are superfluous. Why not cache the results returned from the `GetProducts()` method and bind the GridView to those cached results?

Step 4: Caching the Data Using the ObjectDataSource

By simply setting a few properties, the ObjectDataSource can be configured to automatically cache its retrieved data in the ASP.NET data cache. The following list summarizes the cache-related properties of the ObjectDataSource:

- EnableCaching must be set to `True` to enable caching. The default is `False`.
- CacheDuration the amount of time, in seconds, that the data is cached. The default is 0. The ObjectDataSource will only cache data if `EnableCaching` is `True` and `CacheDuration` is set to a value greater than zero.
- CacheExpirationPolicy can be set to `Absolute` or `Sliding`. If `Absolute`, the ObjectDataSource caches its retrieved data for `CacheDuration` seconds; if `Sliding`, the data expires only after it has not been accessed for `CacheDuration` seconds. The default is `Absolute`.
- CacheKeyDependency use this property to associate the ObjectDataSource s cache entries with an existing cache dependency. The ObjectDataSource s data entries can be prematurely evicted from the cache by expiring its associated `CacheKeyDependency`. This property is most commonly used to associate a SQL cache dependency with the ObjectDataSource s cache, a topic we'll explore in the future Using SQL Cache Dependencies tutorial.

Let s configure the `ProductsDataSource` ObjectDataSource to cache its data for 30 seconds on an absolute scale. Set the ObjectDataSource s `EnableCaching` property to `True` and its `CacheDuration` property to 30. Leave the `CacheExpirationPolicy` property set to its default, `Absolute`.

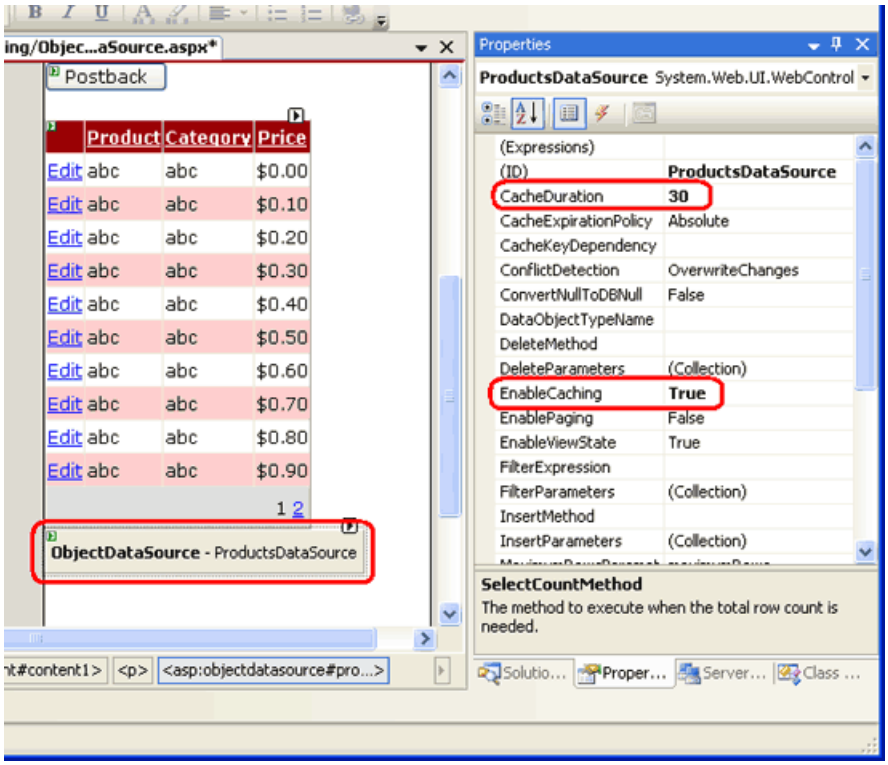


Figure 11: Configure the ObjectDataSource to Cache its Data for 30 Seconds (Click to view full-size image)

Save your changes and revisit this page in a browser. The Selecting event fired text will appear when you first visit the page, as initially the data is not in the cache. But subsequent postbacks triggered by clicking the Postback button, sorting, paging, or clicking the Edit or Cancel buttons does *not* redisplay the Selecting event fired text. This is because the `Selecting` event only fires when the ObjectDataSource gets its data from its underlying object; the `Selecting` event does not fire if the data is pulled from the data cache.

After 30 seconds, the data will be evicted from the cache. The data will also be evicted from the cache if the ObjectDataSource s `Insert`, `Update`, or `Delete` methods are invoked. Consequently, after 30 seconds have passed or the Update button has been clicked, sorting, paging, or clicking the Edit or Cancel buttons will cause the ObjectDataSource to get its data from its underlying object, displaying the Selecting event fired text when the `Selecting` event fires. These returned results are placed back into the data cache.

> **NOTE**
>
> If you see the Selecting event fired text frequently, even when you expect the ObjectDataSource to be working with cached data, it may be due to memory constraints. If there is not enough free memory, the data added to the cache by the ObjectDataSource may have been scavenged. If the ObjectDataSource doesn t appear to be correctly caching the data or only caches the data sporadically, close some applications to free memory and try again.

Figure 12 illustrates the ObjectDataSource s caching workflow. When the Selecting event fired text appears on your screen, it is because the data was not in the cache and had to be retrieved from the underlying object. When this text is missing, however, it s because the data was available from the cache. When the data is returned from the cache there s no call to the underlying object and, therefore, no database query executed.

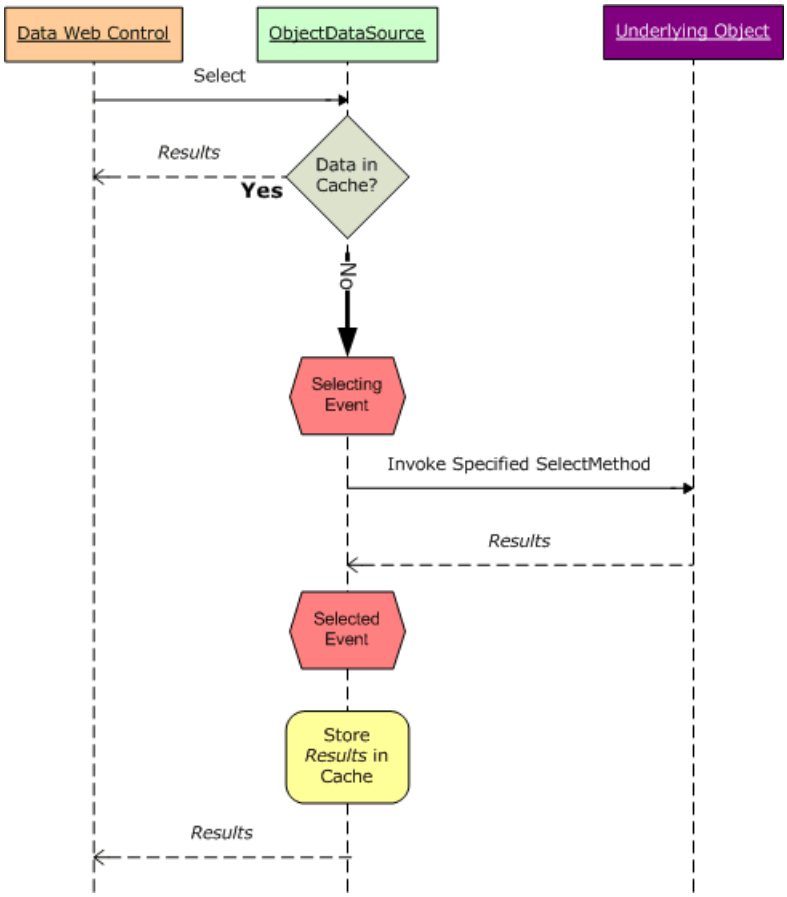


Figure 12: The ObjectDataSource Stores and Retrieves its Data from the Data Cache

Each ASP.NET application has its own data cache instance that s shared across all pages and visitors. That means that the data stored in the data cache by the ObjectDataSource is likewise shared across all users who visit the page. To verify this, open the `ObjectDataSource.aspx` page in a browser. When first visiting the page, the Selecting event fired text will appear (assuming that the data added to the cache by previous tests has, by now, been evicted). Open a second browser instance and copy and paste the URL from the first browser instance to the second. In the second browser instance, the Selecting event fired text is not shown because it s using the same cached data as the first.

When inserting its retrieved data into the cache, the ObjectDataSource uses a cache key value that includes: the `CacheDuration` and `CacheExpirationPolicy` property values; the type of the underlying business object being used by the ObjectDataSource, which is specified via the `TypeName` property (`ProductsBLL` , in this example); the value of the `SelectMethod` property and the name and values of the parameters in the `SelectParameters` collection; and the values of its `StartRowIndex` and `MaximumRows` properties, which are used when implementing custom paging.

Crafting the cache key value as a combination of these properties ensures a unique cache entry as these values change. For example, in past tutorials we ve looked at using the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` , which returns all products for a specified category. One user might come to the page and view beverages, which has a `CategoryID` of 1. If the ObjectDataSource cached its results without regard for the `SelectParameters` values, when another user came to the page to view condiments while the beverages products were in the cache, they d see the cached beverage products rather than condiments. By varying the cache key by these properties, which include the values of the `SelectParameters` , the ObjectDataSource maintains a separate cache entry for beverages and condiments.

Stale Data Concerns

The ObjectDataSource automatically evicts its items from the cache when any one of its `Insert` , `Update` , or `Delete` methods is invoked. This helps protect against stale data by clearing out the cache entries when the data

is modified through the page. However, it is possible for an ObjectDataSource using caching to still display stale data. In the simplest case, it can be due to the data changing directly within the database. Perhaps a database administrator just ran a script that modifies some of the records in the database.

This scenario could also unfold in a more subtle way. While the ObjectDataSource evicts its items from the cache when one of its data modification methods is called, the cached items removed are for the ObjectDataSource s particular combination of property values (`CacheDuration` , `TypeName` , `SelectMethod` , and so on). If you have two ObjectDataSources that use different `SelectMethods` or `SelectParameters` , but still can update the same data, then one ObjectDataSource may update a row and invalidate its own cache entries, but the corresponding row for the second ObjectDataSource will still be served from the cached. I encourage you to create pages to exhibit this functionality. Create a page that displays an editable GridView that pulls its data from an ObjectDataSource that uses caching and is configured to get data from the `ProductsBLL` class s `GetProducts()` method. Add another editable GridView and ObjectDataSource to this page (or another one), but for this second ObjectDataSource have it use the `GetProductsByCategoryID(categoryID)` method. Since the two ObjectDataSources `SelectMethod` properties differ, they ll each have their own cached values. If you edit a product in one grid, the next time you bind the data back to the other grid (by paging, sorting, and so forth), it will still serve the old, cached data and not reflect the change that was made from the other grid.

In short, only use time-based expiries if you are willing to have the potential of stale data, and use shorter expiries for scenarios where the freshness of data is important. If stale data is not acceptable, either forgo caching or use SQL cache dependencies (assuming it is database data you re caching). We'll explore SQL cache dependencies in a future tutorial.

Summary

In this tutorial we examined the ObjectDataSource s built-in caching capabilities. By simply setting a few properties, we can instruct the ObjectDataSource to cache the results returned from the specified `SelectMethod` into the ASP.NET data cache. The `CacheDuration` and `CacheExpirationPolicy` properties indicate the duration the item is cached and whether it is an absolute or sliding expiration. The `CacheKeyDependency` property associates all of the ObjectDataSource s cache entries with an existing cache dependency. This can be used to evict the ObjectDataSource s entries from the cache before the time-based expiration is reached, and is typically used with SQL cache dependencies.

Since the ObjectDataSource simply caches its values to the data cache, we could replicate the ObjectDataSource s built-in functionality programmatically. It doesn t make sense to do this at the Presentation Layer, since the ObjectDataSource offers this functionality out of the box, but we can implement caching capabilities in a separate layer of the architecture. To do so, we'll need to repeat the same logic used by the ObjectDataSource. We'll explore how to programmatically work with the data cache from within the architecture in our next tutorial.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Caching: Techniques and Best Practices
- Caching Architecture Guide for .NET Framework Applications
- Output Caching in ASP.NET 2.0

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via

his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Caching Data in the Architecture (VB)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Sample App or Download PDF

> In the previous tutorial we learned how to apply caching at the Presentation Layer. In this tutorial we learn how to take advantage of our layered architecture to cache data at the Business Logic Layer. We do this by extending the architecture to include a Caching Layer.

Introduction

As we saw in the preceding tutorial, caching the ObjectDataSource s data is as simple as setting a couple of properties. Unfortunately, the ObjectDataSource applies caching at the Presentation Layer, which tightly couples the caching policies with the ASP.NET page. One of the reasons for creating a layered architecture is to allow such couplings to be broken. The Business Logic Layer, for instance, decouples the business logic from the ASP.NET pages, while the Data Access Layer decouples the data access details. This decoupling of business logic and data access details is preferred, in part, because it makes the system more readable, more maintainable, and more flexible to change. It also allows for domain knowledge and division of labor a developer working on the Presentation Layer doesn t need to be familiar with the database s details in order to do her job. Decoupling the caching policy from the Presentation Layer offers similar benefits.

In this tutorial we will augment our architecture to include a *Caching Layer* (or CL for short) that employs our caching policy. The Caching Layer will include a `ProductsCL` class that provides access to product information with methods like `GetProducts()`, `GetProductsByCategoryID(categoryID)`, and so forth, that, when invoked, will first attempt to retrieve the data from the cache. If the cache is empty, these methods will invoke the appropriate `ProductsBLL` method in the BLL, which would in turn get the data from the DAL. The `ProductsCL` methods cache the data retrieved from the BLL before returning it.

As Figure 1 shows, the CL resides between the Presentation and Business Logic Layers.

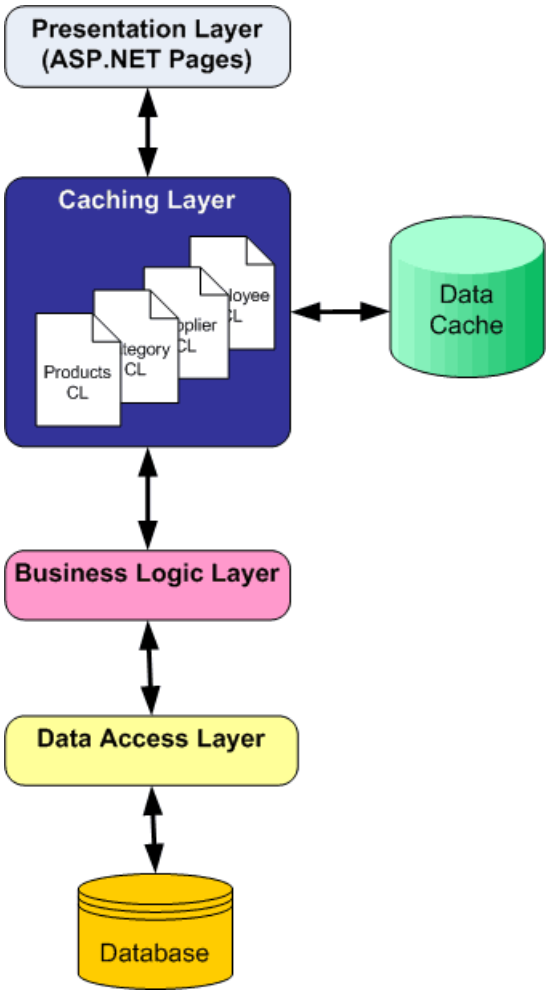


Figure 1: The Caching Layer (CL) is Another Layer in Our Architecture

Step 1: Creating the Caching Layer Classes

In this tutorial we will create a very simple CL with a single class `ProductsCL` that has only a handful of methods. Building a complete Caching Layer for the entire application would require creating `CategoriesCL`, `EmployeesCL`, and `SuppliersCL` classes, and providing a method in these Caching Layer classes for each data access or modification method in the BLL. As with the BLL and DAL, the Caching Layer should ideally be implemented as a separate Class Library project; however, we will implement it as a class in the `App_Code` folder.

To more cleanly separate the CL classes from the DAL and BLL classes, let s create a new subfolder in the `App_Code` folder. Right-click on the `App_Code` folder in the Solution Explorer, choose New Folder, and name the new folder `CL`. After creating this folder, add to it a new class named `ProductsCL.vb`.

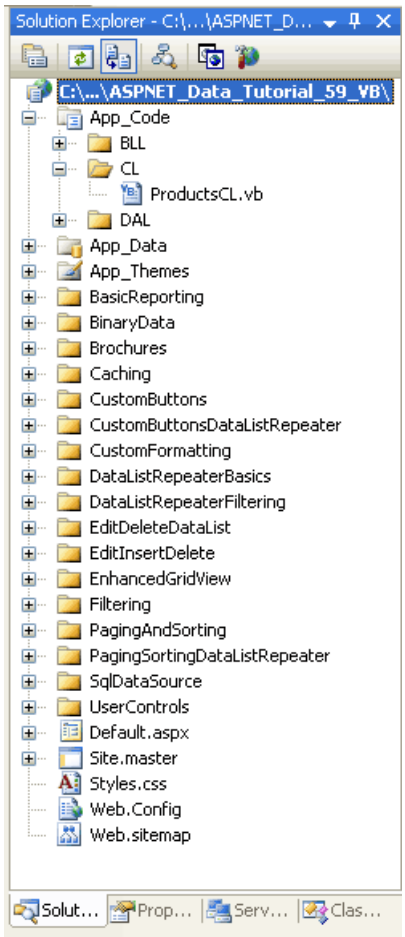


Figure 2: Add a New Folder Named `CL` and a Class Named `ProductsCL.vb`

The `ProductsCL` class should include the same set of data access and modification methods as found in its corresponding Business Logic Layer class (`ProductsBLL`). Rather than creating all of these methods, let s just build a couple here to get a feel for the patterns used by the CL. In particular, we'll add the `GetProducts()` and `GetProductsByCategoryID(categoryID)` methods in Step 3 and an `UpdateProduct` overload in Step 4. You can add the remaining `ProductsCL` methods and `CategoriesCL`, `EmployeesCL`, and `SuppliersCL` classes at your leisure.

Step 2: Reading and Writing to the Data Cache

The ObjectDataSource caching feature explored in the preceding tutorial internally uses the ASP.NET data cache to store the data retrieved from the BLL. The data cache can also be accessed programmatically from ASP.NET pages code-behind classes or from the classes in the web application s architecture. To read and write to the data cache from an ASP.NET page s code-behind class, use the following pattern:

```
' Read from the cache
Dim value as Object = Cache("key")
' Add a new item to the cache
Cache("key") = value
Cache.Insert(key, value)
Cache.Insert(key, value, CacheDependency)
Cache.Insert(key, value, CacheDependency, DateTime, TimeSpan)
```

The `Cache` class s `Insert` method has a number of overloads. `Cache("key") = value` and `Cache.Insert(key, value)` are synonymous and both add an item to the cache using the specified key without a defined expiry. Typically, we want to specify an expiry when adding an item to the cache, either as a dependency, a time-based expiry, or both. Use one of the other `Insert` method s overloads to provide dependency- or time-based expiry information.

The Caching Layer s methods need to first check if the requested data is in the cache and, if so, return it from

there. If the requested data is not in the cache, the appropriate BLL method needs to be invoked. Its return value should be cached and then returned, as the following sequence diagram illustrates.

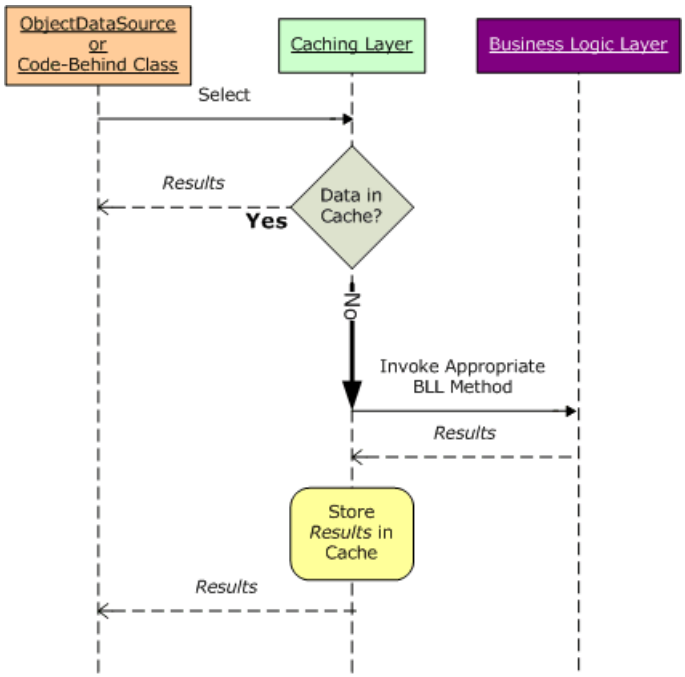


Figure 3: The Caching Layer s Methods Return Data from the Cache if it s Available

The sequence depicted in Figure 3 is accomplished in the CL classes using the following pattern:

```
Dim instance As Type = TryCast(Cache("key"), Type)
If instance Is Nothing Then
    instance = BllMethodToGetInstance()
    Cache.Insert(key, instance, ...)
End If
Return instance
```

Here, *Type* is the type of data being stored in the cache `Northwind.ProductsDataTable`, for example while *key* is the key that uniquely identifies the cache item. If the item with the specified *key* is not in the cache, then *instance* will be `Nothing` and the data will be retrieved from the appropriate BLL method and added to the cache. By the time `Return instance` is reached, *instance* contains a reference to the data, either from the cache or pulled from the BLL.

Be sure to use the above pattern when accessing data from the cache. The following pattern, which, at first glance, looks equivalent, contains a subtle difference that introduces a race condition. Race conditions are difficult to debug because they reveal themselves sporadically and are difficult to reproduce.

```
If Cache("key") Is Nothing Then
    Cache.Insert(key, BllMethodToGetInstance(), ...)
End If
Return Cache("key")
```

The difference in this second, incorrect code snippet is that rather than storing a reference to the cached item in a local variable, the data cache is accessed directly in the conditional statement *and* in the `Return`. Imagine that when this code is reached, `Cache("key")` is not `Nothing`, but before the `Return` statement is reached, the system evicts *key* from the cache. In this rare case, the code will return `Nothing` rather than an object of the expected type.

> **NOTE**
> The data cache is thread-safe, so you don't need to synchronize thread access for simple reads or writes. However, if you need to perform multiple operations on data in the cache that need to be atomic, you are responsible for implementing a lock or some other mechanism to ensure thread safety. See Synchronizing Access to the ASP.NET Cache for more information.

An item can be programmatically evicted from the data cache using the `Remove` method like so:

```
Cache.Remove(key)
```

Step 3: Returning Product Information from the `ProductsCL` Class

For this tutorial let s implement two methods for returning product information from the `ProductsCL` class: `GetProducts()` and `GetProductsByCategoryID(categoryID)`. Like with the `ProductsBL` class in the Business Logic Layer, the `GetProducts()` method in the CL returns information about all of the products as a `Northwind.ProductsDataTable` object, while `GetProductsByCategoryID(categoryID)` returns all of the products from a specified category.

The following code shows a portion of the methods in the `ProductsCL` class:

```
<System.ComponentModel.DataObject()> _
Public Class ProductsCL
    Private _productsAPI As ProductsBLL = Nothing
    Protected ReadOnly Property API() As ProductsBLL
        Get
            If _productsAPI Is Nothing Then
                _productsAPI = New ProductsBLL()
            End If
            Return _productsAPI
        End Get
    End Property
    <System.ComponentModel.DataObjectMethodAttribute _
    (DataObjectMethodType.Select, True)> _
    Public Function GetProducts() As Northwind.ProductsDataTable
        Const rawKey As String = "Products"
        ' See if the item is in the cache
        Dim products As Northwind.ProductsDataTable = _
            TryCast(GetCacheItem(rawKey), Northwind.ProductsDataTable)
        If products Is Nothing Then
            ' Item not found in cache - retrieve it and insert it into the cache
            products = API.GetProducts()
            AddCacheItem(rawKey, products)
        End If
        Return products
    End Function
    <System.ComponentModel.DataObjectMethodAttribute _
        (DataObjectMethodType.Select, False)> _
    Public Function GetProductsByCategoryID(ByVal categoryID As Integer) _
        As Northwind.ProductsDataTable
        If (categoryID < 0) Then
            Return GetProducts()
        Else
            Dim rawKey As String = String.Concat("ProductsByCategory-", categoryID)
            ' See if the item is in the cache
            Dim products As Northwind.ProductsDataTable = _
                TryCast(GetCacheItem(rawKey), Northwind.ProductsDataTable)
            If products Is Nothing Then
                ' Item not found in cache - retrieve it and insert it into the cache
                products = API.GetProductsByCategoryID(categoryID)
                AddCacheItem(rawKey, products)
            End If
            Return products
        End If
    End Function
End Class
```

First, note the `DataObject` and `DataObjectMethodAttribute` attributes applied to the class and methods. These attributes provide information to the ObjectDataSource s wizard, indicating what classes and methods should appear in the wizard s steps. Since the CL classes and methods will be accessed from an ObjectDataSource in the Presentation Layer, I added these attributes to enhance the design-time experience. Refer back to the Creating a Business Logic Layer tutorial for a more thorough description on these attributes and their effects.

In the `GetProducts()` and `GetProductsByCategoryID(categoryID)` methods, the data returned from the `GetCacheItem(key)` method is assigned to a local variable. The `GetCacheItem(key)` method, which we'll examine shortly, returns a particular item from the cache based on the specified *key*. If no such data is found in cache, it is retrieved from the corresponding `ProductsBLL` class method and then added to the cache using the `AddCacheItem(key, value)` method.

The `GetCacheItem(key)` and `AddCacheItem(key, value)` methods interface with the data cache, reading and writing values, respectively. The `GetCacheItem(key)` method is the simpler of the two. It simply returns the value from the Cache class using the passed-in *key*:

```
Private Function GetCacheItem(ByVal rawKey As String) As Object
    Return HttpRuntime.Cache(GetCacheKey(rawKey))
End Function
Private ReadOnly MasterCacheKeyArray() As String = {"ProductsCache"}
Private Function GetCacheKey(ByVal cacheKey As String) As String
    Return String.Concat(MasterCacheKeyArray(0), "-", cacheKey)
End Function
```

`GetCacheItem(key)` does not use *key* value as supplied, but instead calls the `GetCacheKey(key)` method, which returns the *key* prepended with ProductsCache-. The `MasterCacheKeyArray`, which holds the string ProductsCache, is also used by the `AddCacheItem(key, value)` method, as we'll see momentarily.

From an ASP.NET page s code-behind class, the data cache can be accessed using the `Page` class s `Cache` property, and allows for syntax like `Cache("key") = value`, as discussed in Step 2. From a class within the architecture, the data cache can be accessed using either `HttpRuntime.Cache` or `HttpContext.Current.Cache`. Peter Johnson's blog entry HttpRuntime.Cache vs. HttpContext.Current.Cache notes the slight performance advantage in using `HttpRuntime` instead of `HttpContext.Current`; consequently, `ProductsCL` uses `HttpRuntime`.

> **NOTE**
> If your architecture is implemented using Class Library projects then you will need to add a reference to the `System.Web` assembly in order to use the `HttpRuntime` and `HttpContext` classes.

If the item is not found in the cache, the `ProductsCL` class s methods get the data from the BLL and add it to the cache using the `AddCacheItem(key, value)` method. To add *value* to the cache we could use the following code, which uses a 60 second time expiry:

```
Const CacheDuration As Double = 60.0
Private Sub AddCacheItem(ByVal rawKey As String, ByVal value As Object)
    DataCache.Insert(GetCacheKey(rawKey), value, Nothing, _
        DateTime.Now.AddSeconds(CacheDuration), _
        System.Web.Caching.Cache.NoSlidingExpiration)
End Sub
```

`DateTime.Now.AddSeconds(CacheDuration)` specifies the time-based expiry 60 seconds in the future while `System.Web.Caching.Cache.NoSlidingExpiration` indicates that there s no sliding expiration. While this `Insert` method overload has input parameters for both an absolute and sliding expiry, you can only provide one of the two. If you attempt to specify both an absolute time and a time span, the `Insert` method will throw an `ArgumentException` exception.

> **NOTE**
> This implementation of the `AddCacheItem(key, value)` method currently has some shortcomings. We'll address and overcome these issues in Step 4.

Step 4: Invalidating the Cache When the Data is Modified Through the Architecture

Along with data retrieval methods, the Caching Layer needs to provide the same methods as the BLL for inserting, updating, and deleting data. The CL s data modification methods do not modify the cached data, but rather call the BLL s corresponding data modification method and then invalidate the cache. As we saw in the preceding tutorial, this is the same behavior that the ObjectDataSource applies when its caching features are

enabled and its `Insert`, `Update`, or `Delete` methods are invoked.

The following `UpdateProduct` overload illustrates how to implement the data modification methods in the CL:

```
<DataObjectMethodAttribute(DataObjectMethodType.Update, False)> _
Public Function UpdateProduct(productName As String, _
    unitPrice As Nullable(Of Decimal), productID As Integer) _
    As Boolean
    Dim result As Boolean = API.UpdateProduct(productName, unitPrice, productID)
    ' TODO: Invalidate the cache
    Return result
End Function
```

The appropriate data modification Business Logic Layer method is invoked, but before its response is returned we need to invalidate the cache. Unfortunately, invalidating the cache is not straightforward because the `ProductsCL` class s `GetProducts()` and `GetProductsByCategoryID(categoryID)` methods each add items to the cache with different keys, and the `GetProductsByCategoryID(categoryID)` method adds a different cache item for each unique *categoryID*.

When invalidating the cache, we need to remove *all* of the items that may have been added by the `ProductsCL` class. This can be accomplished by associating a *cache dependency* with the each item added to the cache in the `AddCacheItem(key, value)` method. In general, a cache dependency can be another item in the cache, a file on the file system, or data from a Microsoft SQL Server database. When the dependency changes or is removed from the cache, the cache items it is associated with are automatically evicted from the cache. For this tutorial, we want to create an additional item in the cache that serves as a cache dependency for all items added through the `ProductsCL` class. That way, all of these items can be removed from the cache by simply removing the cache dependency.

Let s update the `AddCacheItem(key, value)` method so that each item added to the cache through this method is associated with a single cache dependency:

```
Private Sub AddCacheItem(ByVal rawKey As String, ByVal value As Object)
    Dim DataCache As System.Web.Caching.Cache = HttpRuntime.Cache
    ' Make sure MasterCacheKeyArray[0] is in the cache - if not, add it
    If DataCache(MasterCacheKeyArray(0)) Is Nothing Then
        DataCache(MasterCacheKeyArray(0)) = DateTime.Now
    End If
    ' Add a CacheDependency
    Dim dependency As New Caching.CacheDependency(Nothing, MasterCacheKeyArray) _
        DataCache.Insert(GetCacheKey(rawKey), value, dependency, _
        DateTime.Now.AddSeconds(CacheDuration), _
        System.Web.Caching.Cache.NoSlidingExpiration)
End Sub
```

`MasterCacheKeyArray` is a string array that holds a single value, ProductsCache. First, a cache item is added to the cache and assigned the current date and time. If the cache item already exists, it is updated. Next, a cache dependency is created. The `CacheDependency` class s constructor has a number of overloads, but the one being used in here expects two `String` array inputs. The first one specifies the set of files to be used as dependencies. Since we don t want to use any file-based dependencies, a value of `Nothing` is used for the first input parameter. The second input parameter specifies the set of cache keys to use as dependencies. Here we specify our single dependency, `MasterCacheKeyArray`. The `CacheDependency` is then passed into the `Insert` method.

With this modification to `AddCacheItem(key, value)`, invaliding the cache is as simple as removing the dependency.

```
<DataObjectMethodAttribute(DataObjectMethodType.Update, False)> _
Public Function UpdateProduct(ByVal productName As String, _
    ByVal unitPrice As Nullable(Of Decimal), ByVal productID As Integer) _
    As Boolean
    Dim result As Boolean = API.UpdateProduct(productName, unitPrice, productID)
    ' Invalidate the cache
    InvalidateCache()
    Return result
End Function
Public Sub InvalidateCache()
    ' Remove the cache dependency
    HttpRuntime.Cache.Remove(MasterCacheKeyArray(0))
End Sub
```

Step 5: Calling the Caching Layer from the Presentation Layer

The Caching Layer s classes and methods can be used to work with data using the techniques we ve examined throughout these tutorials. To illustrate working with cached data, save your changes to the `ProductsCL` class and then open the `FromTheArchitecture.aspx` page in the `Caching` folder and add a GridView. From the GridView s smart tag, create a new ObjectDataSource. In the wizard s first step you should see the `ProductsCL` class as one of the options from the drop-down list.

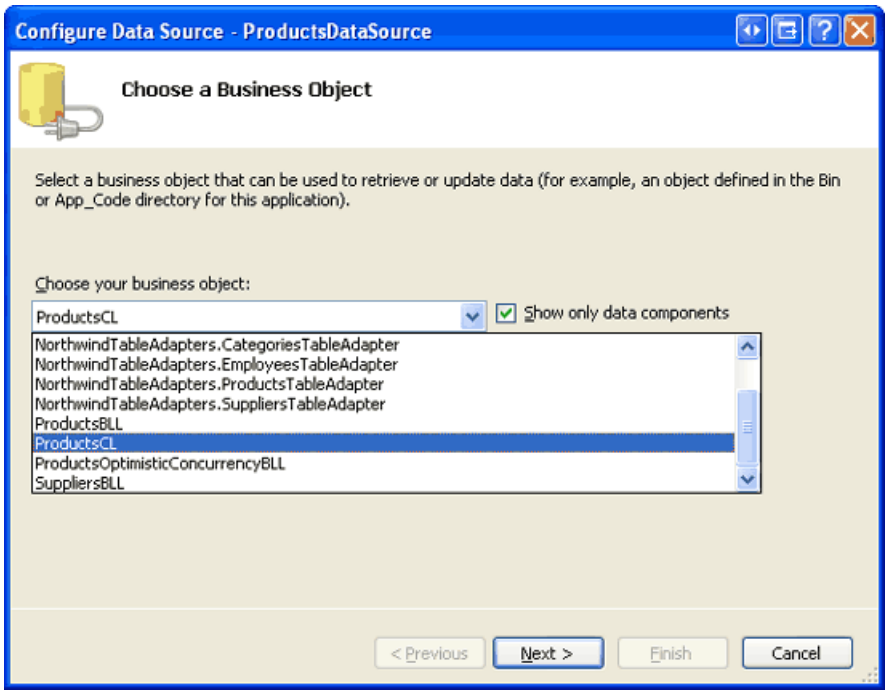


Figure 4: The `ProductsCL` Class is Included in the Business Object Drop-Down List (Click to view full-size image)

After selecting `ProductsCL`, click Next. The drop-down list in the SELECT tab has two items - `GetProducts()` and `GetProductsByCategoryID(categoryID)` and the UPDATE tab has the sole `UpdateProduct` overload. Choose the `GetProducts()` method from the SELECT tab and the `UpdateProducts` method from the UPDATE tab and click Finish.

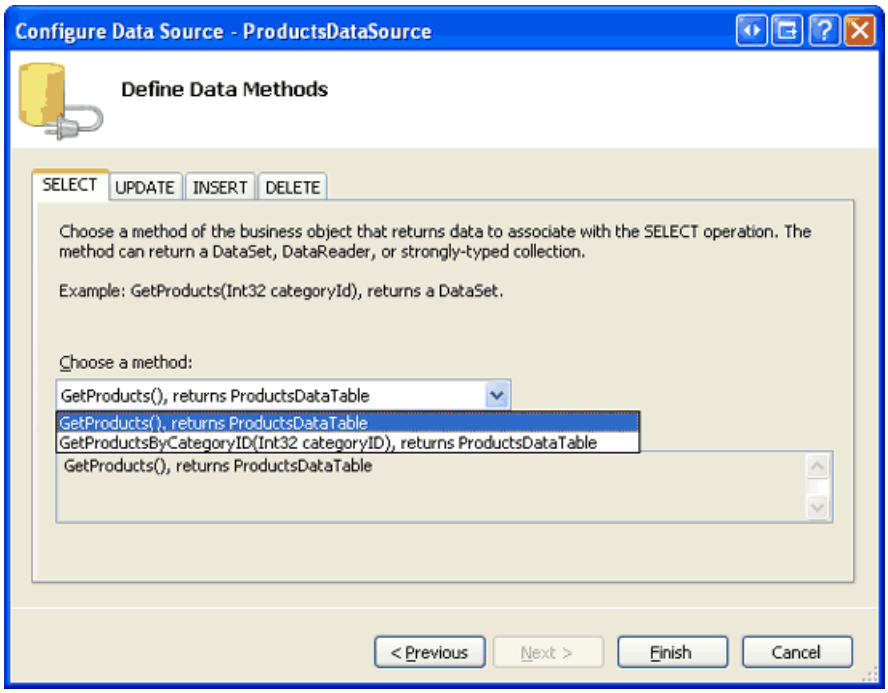


Figure 5: The `ProductsCL` Class s Methods are Listed in the Drop-Down Lists (Click to view full-size image)

After completing the wizard, Visual Studio will set the ObjectDataSource s `OldValuesParameterFormatString` property to `original_{0}` and add the appropriate fields to the GridView. Change the `OldValuesParameterFormatString` property back to its default value, `{0}`, and configure the GridView to support paging, sorting, and editing. Since the `UploadProducts` overload used by the CL accepts only the edited product s name and price, limit the GridView so that only these fields are editable.

In the preceding tutorial we defined a GridView to include fields for the `ProductName`, `CategoryName`, and `UnitPrice` fields. Feel free to replicate this formatting and structure, in which case your GridView and ObjectDataSource s declarative markup should look similar to the following:

```
<asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
    AllowPaging="True" AllowSorting="True">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
            <EditItemTemplate>
                <asp:TextBox ID="ProductName" runat="server"
                    Text='<%# Bind("ProductName") %>' />
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                    ControlToValidate="ProductName" Display="Dynamic"
                    ErrorMessage="You must provide a name for the product."
                    SetFocusOnError="True"
                    runat="server">*</asp:RequiredFieldValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Bind("ProductName") %>'></asp:Label>
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:TemplateField HeaderText="Price" SortExpression="UnitPrice">
            <EditItemTemplate>
                $<asp:TextBox ID="UnitPrice" runat="server" Columns="8"
                    Text='<%# Bind("UnitPrice", "{0:N2}") %>'></asp:TextBox>
                <asp:CompareValidator ID="CompareValidator1" runat="server"
                    ControlToValidate="UnitPrice" Display="Dynamic"
                    ErrorMessage="You must enter a valid currency value with
                        no currency symbols. Also, the value must be greater than
                        or equal to zero."
                    Operator="GreaterThanEqual" SetFocusOnError="True"
                    Type="Currency" ValueToCompare="0">*</asp:CompareValidator>
            </EditItemTemplate>
            <ItemStyle HorizontalAlign="Right" />
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("UnitPrice", "{0:c}") %>' />
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="{0}" SelectMethod="GetProducts"
    TypeName="ProductsCL" UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

At this point we have a page that uses the Caching Layer. To see the cache in action, set breakpoints in the `ProductsCL` class s `GetProducts()` and `UpdateProduct` methods. Visit the page in a browser and step through the code when sorting and paging in order to see the data pulled from the cache. Then update a record and note that the cache is invalidated and, consequently, it is retrieved from the BLL when the data is rebound to the GridView.

NOTE

The Caching Layer provided in the download accompanying this article is not complete. It contains only one class, `ProductsCL`, which only sports a handful of methods. Moreover, only a single ASP.NET page uses the CL (`~/Caching/FromTheArchitecture.aspx`) all others still reference the BLL directly. If you plan on using a CL in your application, all calls from the Presentation Layer should go to the CL, which would require that the CL s classes and methods covered those classes and methods in the BLL currently used by the Presentation Layer.

Summary

While caching can be applied at the Presentation Layer with ASP.NET 2.0 s SqlDataSource and ObjectDataSource controls, ideally caching responsibilities would be delegated to a separate layer in the architecture. In this tutorial we created a Caching Layer that resides between the Presentation Layer and the Business Logic Layer. The Caching Layer needs to provide the same set of classes and methods that exist in the BLL and are called from the Presentation Layer.

The Caching Layer examples we explored in this and the preceding tutorials exhibited *reactive loading*. With reactive loading, the data is loaded into the cache only when a request for the data is made and that data is missing from the cache. Data can also be *proactively loaded* into the cache, a technique that loads the data into the cache before it is actually needed. In the next tutorial we'll see an example of proactive loading when we look at how to store static values into the cache at application startup.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Caching Data at Application Startup (VB)

2/20/2020 • 14 minutes to read • Edit Online

by Scott Mitchell

Download PDF

In any Web application some data will be frequently used and some data will be infrequently used. We can improve the performance of our ASP.NET application by loading in advance the frequently-used data, a technique known as. This tutorial demonstrates one approach to proactive loading, which is to load data into the cache at application startup.

Introduction

The two previous tutorials looked at caching data in the Presentation and Caching Layers. In Caching Data with the ObjectDataSource, we looked at using the ObjectDataSource s caching features to cache data in the Presentation Layer. Caching Data in the Architecture examined caching in a new, separate Caching Layer. Both of these tutorials used *reactive loading* in working with the data cache. With reactive loading, each time the data is requested, the system first checks if it s in the cache. If not, it grabs the data from the originating source, such as the database, and then stores it in the cache. The main advantage to reactive loading is its ease of implementation. One of its disadvantages is its uneven performance across requests. Imagine a page that uses the Caching Layer from the preceding tutorial to display product information. When this page is visited for the first time, or visited for the first time after the cached data has been evicted due to memory constraints or the specified expiry having been reached, the data must be retrieved from the database. Therefore, these users requests will take longer than users requests that can be served by the cache.

Proactive loading provides an alternative cache management strategy that smoothes out the performance across requests by loading the cached data before it s needed. Typically, proactive loading uses some process that either periodically checks or is notified when there has been an update to the underlying data. This process then updates the cache to keep it fresh. Proactive loading is especially useful if the underlying data comes from a slow database connection, a Web service, or some other particularly sluggish data source. But this approach to proactive loading is more difficult to implement, as it requires creating, managing, and deploying a process to check for changes and update the cache.

Another flavor of proactive loading, and the type we'll be exploring in this tutorial, is loading data into the cache at application startup. This approach is especially useful for caching static data, such as the records in database lookup tables.

> **NOTE**
>
> For a more in-depth look at the differences between proactive and reactive loading, as well as lists of pros, cons, and implementation recommendations, refer to the Managing the Contents of a Cache section of the Caching Architecture Guide for .NET Framework Applications.

Step 1: Determining What Data to Cache at Application Startup

The caching examples using reactive loading that we examined in the previous two tutorials work well with data that may periodically change and does not take exorbitantly long to generate. But if the cached data never changes, the expiry used by reactive loading is superfluous. Likewise, if the data being cached takes an exceedingly long time to generate, then those users whose requests find the cache empty will have to endure a

lengthy wait while the underlying data is retrieved. Consider caching static data and data that takes an exceptionally long time to generate at application startup.

While databases have many dynamic, frequently-changing values, most also have a fair amount of static data. For example, virtually all data models have one or more columns that contain a particular value from a fixed set of choices. A `Patients` database table might have a `PrimaryLanguage` column, whose set of values could be English, Spanish, French, Russian, Japanese, and so on. Oftentimes, these types of columns are implemented using *lookup tables*. Rather than storing the string English or French in the `Patients` table, a second table is created that has, commonly, two columns - a unique identifier and a string description - with a record for each possible value. The `PrimaryLanguage` column in the `Patients` table stores the corresponding unique identifier in the lookup table. In Figure 1, patient John Doe s primary language is English, while Ed Johnson s is Russian.

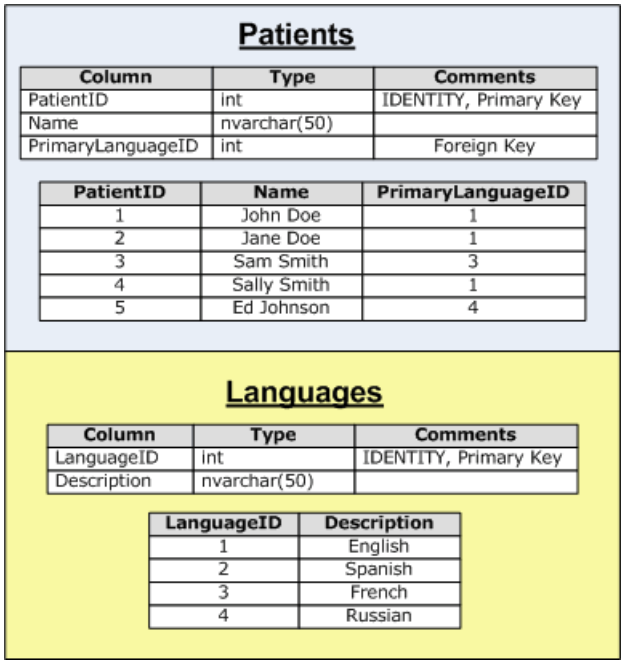


Figure 1: The `Languages` Table is a Lookup Table Used by the `Patients` Table

The user interface for editing or creating a new patient would include a drop-down list of allowable languages populated by the records in the `Languages` table. Without caching, each time this interface is visited the system must query the `Languages` table. This is wasteful and unnecessary since lookup table values change very infrequently, if ever.

We could cache the `Languages` data using the same reactive loading techniques examined in the previous tutorials. Reactive loading, however, uses a time-based expiry, which is not needed for static lookup table data. While caching using reactive loading would be better than no caching at all, the best approach would be to proactively load the lookup table data into the cache at application startup.

In this tutorial we will look at how to cache lookup table data and other static information.

Step 2: Examining the Different Ways to Cache Data

Information can be programmatically cached in an ASP.NET application using a variety of approaches. We ve already seen how to use the data cache in previous tutorials. Alternatively, objects can be programmatically cached using *static members* or *application state*.

When working with a class, typically the class must first be instantiated before its members can be accessed. For example, in order to invoke a method from one of the classes in our Business Logic Layer, we must first create an instance of the class:

```
Dim productsAPI As New ProductsBLL()
productsAPI.SomeMethod()
productsAPI.SomeProperty = "Hello, World!"
```

Before we can invoke *SomeMethod* or work with *SomeProperty*, we must first create an instance of the class using the `New` keyword. *SomeMethod* and *SomeProperty* are associated with a particular instance. The lifetime of these members is tied to the lifetime of their associated object. *Static members*, on the other hand, are variables, properties, and methods that are shared among *all* instances of the class and, consequently, have a lifetime as long as the class. Static members are denoted by the keyword `Shared`.

In addition to static members, data can be cached using application state. Each ASP.NET application maintains a name/value collection that s shared across all users and pages of the application. This collection can be accessed using the `HttpContext` class s `Application` property, and used from an ASP.NET page s code-behind class like so:

```
Application("key") = value
Dim value As Object = Application("key")
```

The data cache provides a much richer API for caching data, providing mechanisms for time- and dependency-based expiries, cache item priorities, and so forth. With static members and application state, such features must be manually added by the page developer. When caching data at application startup for the lifetime of the application, however, the data cache s advantages are moot. In this tutorial we'll look at code that uses all three techniques for caching static data.

Step 3: Caching the `Suppliers` Table Data

The Northwind database tables we ve implemented to date do not include any traditional lookup tables. The four DataTables implemented in our DAL all model tables whose values are non-static. Rather than spending the time to add a new DataTable to the DAL and then a new class and methods to the BLL, for this tutorial let s just pretend that the `Suppliers` table s data is static. Therefore, we could cache this data at application startup.

To start, create a new class named `StaticCache.cs` in the `CL` folder.

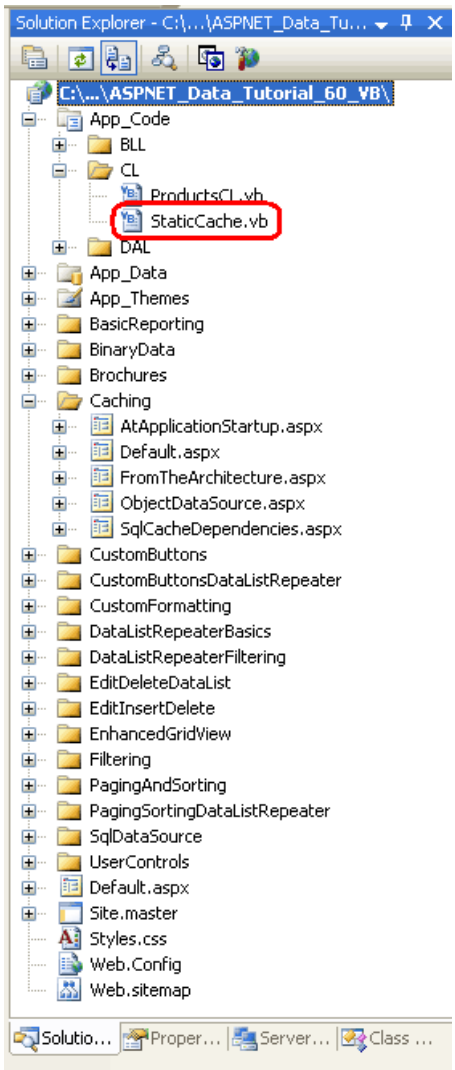


Figure 2: Create the `StaticCache.vb` Class in the `CL` Folder

We need to add a method that loads the data at startup into the appropriate cache store, as well as methods that return data from this cache.

```
<System.ComponentModel.DataObject()> _
Public Class StaticCache
    Private Shared suppliers As Northwind.SuppliersDataTable = Nothing
    Public Shared Sub LoadStaticCache()
        ' Get suppliers - cache using a static member variable
        Dim suppliersBLL As New SuppliersBLL()
        suppliers = suppliersBLL.GetSuppliers()
    End Sub

    <DataObjectMethodAttribute(DataObjectMethodType.Select, True)> _
    Public Shared Function GetSuppliers() As Northwind.SuppliersDataTable
        Return suppliers
    End Function
End Class
```

The above code uses a static member variable, `suppliers`, to hold the results from the `SuppliersBLL` class s `GetSuppliers()` method, which is called from the `LoadStaticCache()` method. The `LoadStaticCache()` method is meant to be called during the application s start. Once this data has been loaded at application startup, any page that needs to work with supplier data can call the `StaticCache` class s `GetSuppliers()` method. Therefore, the call to the database to get the suppliers only happens once, at application start.

Rather than using a static member variable as the cache store, we could have alternatively used application state or the data cache. The following code shows the class retooled to use application state:

```
<System.ComponentModel.DataObject()> _
Public Class StaticCache
    Public Shared Sub LoadStaticCache()
        ' Get suppliers - cache using application state
        Dim suppliersBLL As New SuppliersBLL()
        HttpContext.Current.Application("key") = suppliers
    End Sub

    <DataObjectMethodAttribute(DataObjectMethodType.Select, True)> _
    Public Shared Function GetSuppliers() As Northwind.SuppliersDataTable
        Return TryCast(HttpContext.Current.Application("key"), _
            Northwind.SuppliersDataTable)
    End Function
End Class
```

In `LoadStaticCache()`, the supplier information is stored to the application variable *key*. It s returned as the appropriate type (`Northwind.SuppliersDataTable`) from `GetSuppliers()`. While application state can be accessed in the code-behind classes of ASP.NET pages using `Application("key")`, in the architecture we must use `HttpContext.Current.Application("key")` in order to get the current `HttpContext`.

Likewise, the data cache can be used as a cache store, as the following code shows:

```
<System.ComponentModel.DataObject()> _
Public Class StaticCache
    Public Shared Sub LoadStaticCache()
        ' Get suppliers - cache using a static member variable
        Dim suppliersBLL As New SuppliersBLL()
        HttpRuntime.Cache.Insert("key", suppliers, Nothing, _
            Cache.NoAbsoluteExpiration, Cache.NoSlidingExpiration, _
            CacheItemPriority.NotRemovable, Nothing)
    End Sub
    <System.ComponentModel.DataObjectMethodAttribute_
    (System.ComponentModel.DataObjectMethodType.Select, True)> _
    Public Shared Function GetSuppliers() As Northwind.SuppliersDataTable
        Return TryCast(HttpRuntime.Cache("key"), Northwind.SuppliersDataTable)
    End Function
End Class
```

To add an item to the data cache with no time-based expiry, use the `System.Web.Caching.Cache.NoAbsoluteExpiration` and `System.Web.Caching.Cache.NoSlidingExpiration` values as input parameters. This particular overload of the data cache s `Insert` method was selected so that we could specify the *priority* of the cache item. The priority is used to determine what items to scavenge from the cache when available memory runs low. Here we use the priority `NotRemovable`, which ensures that this cache item won't be scavenged.

> **NOTE**
> This tutorial s download implements the `StaticCache` class using the static member variable approach. The code for the application state and data cache techniques is available in the comments in the class file.

Step 4: Executing Code at Application Startup

To execute code when a web application first starts, we need to create a special file named `Global.asax`. This file can contain event handlers for application-, session-, and request-level events, and it is here where we can add code that will be executed whenever the application starts.

Add the `Global.asax` file to your web application s root directory by right-clicking on the website project name in

Visual Studio s Solution Explorer and choosing Add New Item. From the Add New Item dialog box, select the Global Application Class item type and then click the Add button.

> **NOTE**
>
> If you already have a `Global.asax` file in your project, the Global Application Class item type will not be listed in the Add New Item dialog box.

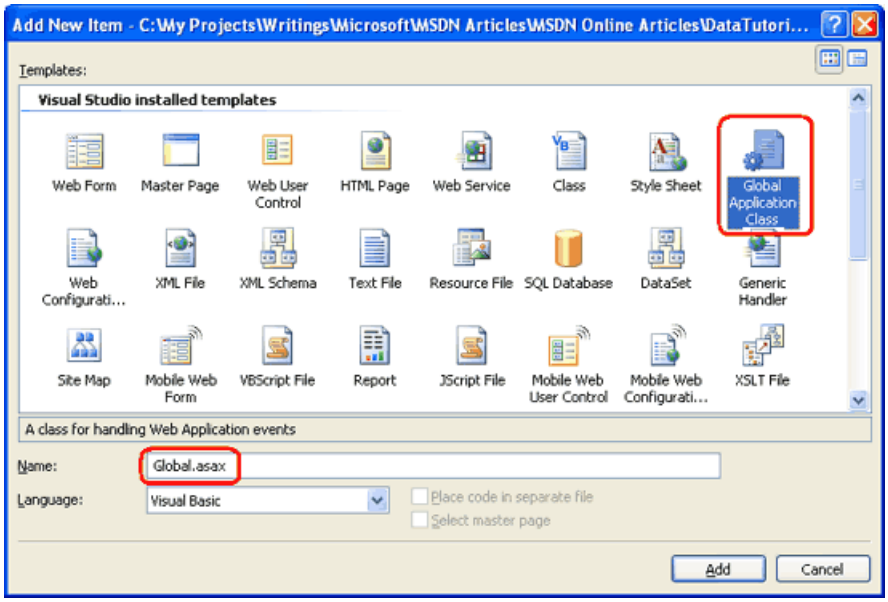


Figure 3: Add the `Global.asax` File to Your Web Application s Root Directory (Click to view full-size image)

The default `Global.asax` file template includes five methods within a server-side `<script>` tag:

- `Application_Start` executes when the web application first starts
- `Application_End` runs when the application is shutting down
- `Application_Error` executes whenever an unhandled exception reaches the application
- `Session_Start` executes when a new session is created
- `Session_End` runs when a session is expired or abandoned

The `Application_Start` event handler is called only once during an application s life cycle. The application starts the first time an ASP.NET resource is requested from the application and continues to run until the application is restarted, which can happen by modifying the contents of the `/Bin` folder, modifying `Global.asax`, modifying the contents in the `App_Code` folder, or modifying the `Web.config` file, among other causes. Refer to ASP.NET Application Life Cycle Overview for a more detailed discussion on the application life cycle.

For these tutorials we only need to add code to the `Application_Start` method, so feel free to remove the others. In `Application_Start`, simply call the `StaticCache` class s `LoadStaticCache()` method, which will load and cache the supplier information:

```
<%@ Application Language="VB" %>
<script runat="server">
    Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)
        StaticCache.LoadStaticCache()
    End Sub
</script>
```

That s all there is to it! At application startup, the `LoadStaticCache()` method will grab the supplier information from the BLL, and store it in a static member variable (or whatever cache store you ended up using in the `StaticCache` class). To verify this behavior, set a breakpoint in the `Application_Start` method and run your

application. Note that the breakpoint is hit upon the application starting. Subsequent requests, however, do not cause the `Application_Start` method to execute.

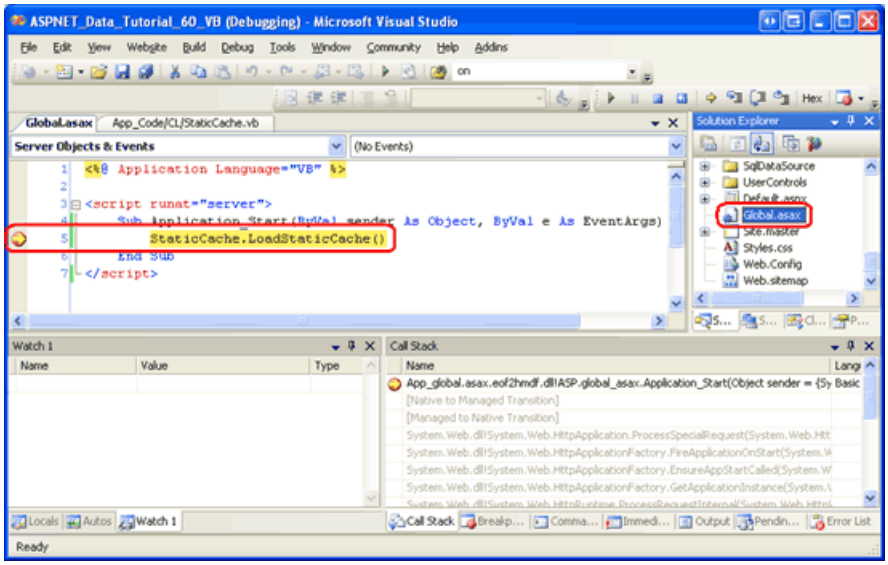


Figure 4: Use a Breakpoint to Verify that the `Application_Start` Event Handler is Being Executed (Click to view full-size image)

> **NOTE**
>
> If you do not hit the `Application_Start` breakpoint when you first start debugging, it is because your application has already started. Force the application to restart by modifying your `Global.asax` or `Web.config` files and then try again. You can simply add (or remove) a blank line at the end of one of these files to quickly restart the application.

Step 5: Displaying the Cached Data

At this point the `StaticCache` class has a version of the supplier data cached at application startup that can be accessed through its `GetSuppliers()` method. To work with this data from the Presentation Layer, we can use an ObjectDataSource or programmatically invoke the `StaticCache` class s `GetSuppliers()` method from an ASP.NET page s code-behind class. Let s look at using the ObjectDataSource and GridView controls to display the cached supplier information.

Start by opening the `AtApplicationStartup.aspx` page in the `Caching` folder. Drag a GridView from the Toolbox onto the designer, setting its `ID` property to `Suppliers`. Next, from the GridView s smart tag choose to create a new ObjectDataSource named `SuppliersCachedDataSource`. Configure the ObjectDataSource to use the `StaticCache` class s `GetSuppliers()` method.

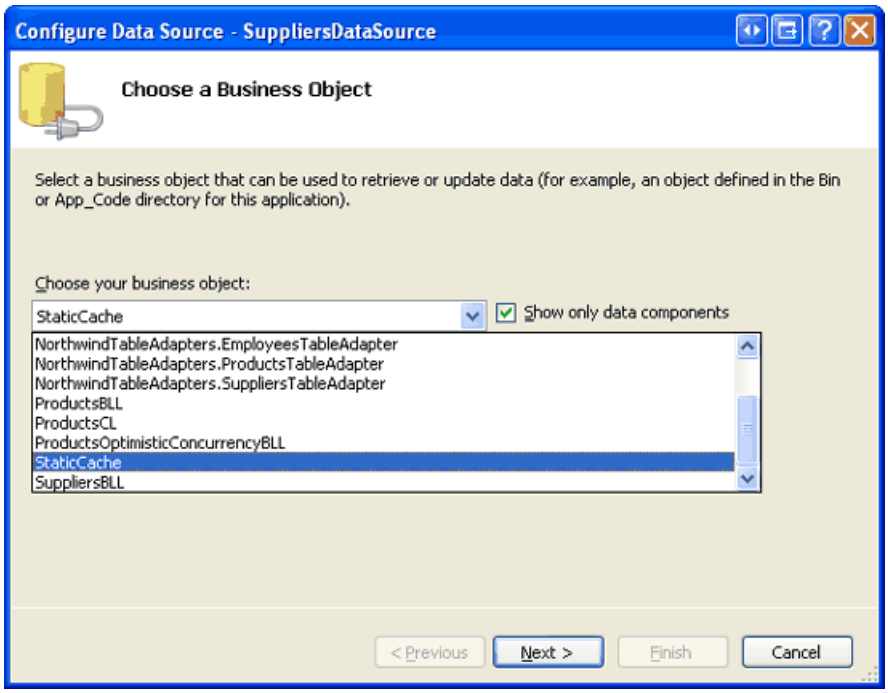


Figure 5: Configure the ObjectDataSource to use the `StaticCache` Class (Click to view full-size image)

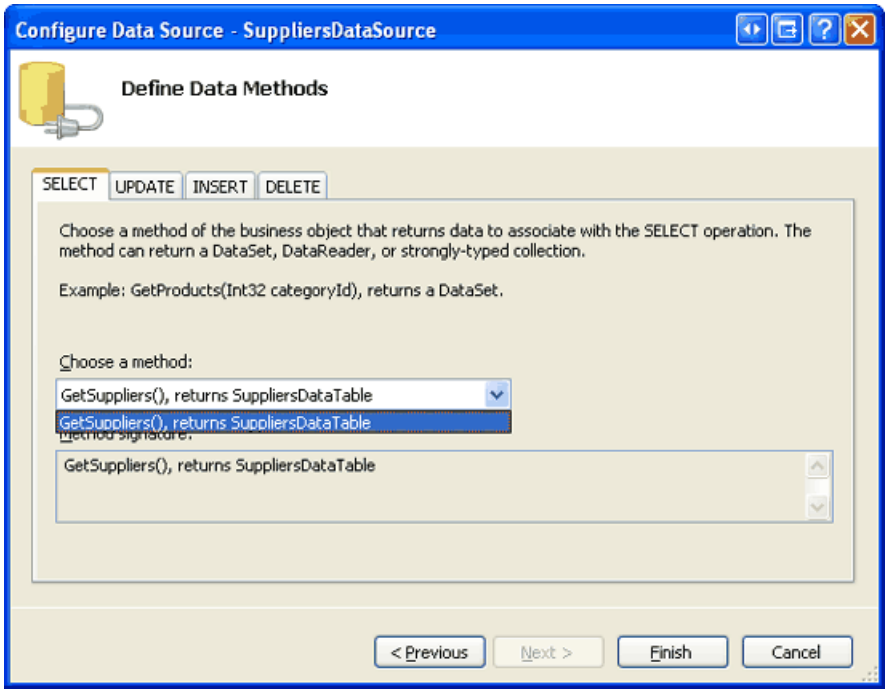


Figure 6: Use the `GetSuppliers()` Method to Retrieve the Cached Supplier Data (Click to view full-size image)

After completing the wizard, Visual Studio will automatically add BoundFields for each of the data fields in `SuppliersDataTable`. Your GridView and ObjectDataSource s declarative markup should look similar to the following:

```
<asp:GridView ID="Suppliers" runat="server" AutoGenerateColumns="False"
    DataKeyNames="SupplierID" DataSourceID="SuppliersCachedDataSource"
    EnableViewState="False">
    <Columns>
        <asp:BoundField DataField="SupplierID" HeaderText="SupplierID"
            InsertVisible="False" ReadOnly="True"
            SortExpression="SupplierID" />
        <asp:BoundField DataField="CompanyName" HeaderText="CompanyName"
            SortExpression="CompanyName" />
        <asp:BoundField DataField="Address" HeaderText="Address"
            SortExpression="Address" />
        <asp:BoundField DataField="City" HeaderText="City"
            SortExpression="City" />
        <asp:BoundField DataField="Country" HeaderText="Country"
            SortExpression="Country" />
        <asp:BoundField DataField="Phone" HeaderText="Phone"
            SortExpression="Phone" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="SuppliersCachedDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetSuppliers" TypeName="StaticCache" />
```

Figure 7 shows the page when viewed through a browser. The output is the same had we pulled the data from the BLL s `SuppliersBLL` class, but using the `StaticCache` class returns the supplier data as cached at application startup. You can set breakpoints in the `StaticCache` class s `GetSuppliers()` method to verify this behavior.

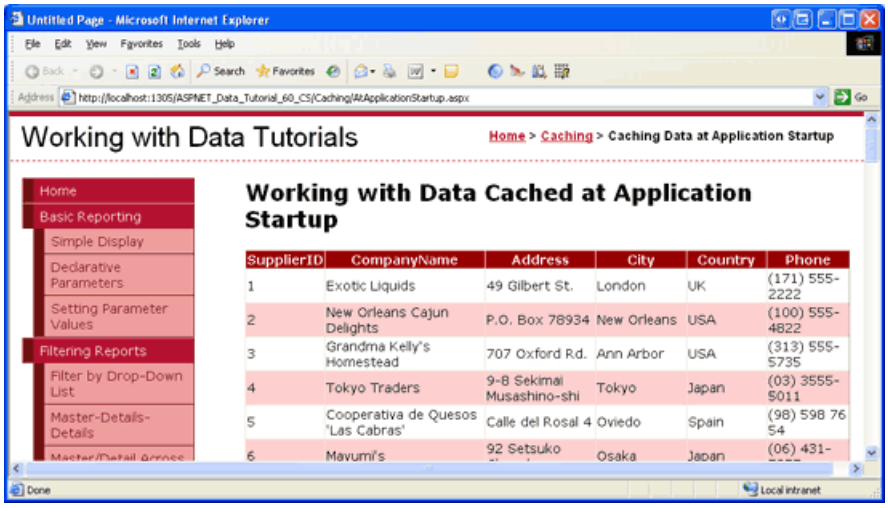


Figure 7: The Cached Supplier Data is Displayed in a GridView (Click to view full-size image)

Summary

Most every data model contains a fair amount of static data, usually implemented in the form of lookup tables. Since this information is static, there s no reason to continually access the database each time this information needs to be displayed. Furthermore, due to its static nature, when caching the data there s no need for an expiry. In this tutorial we saw how to take such data and cache it in the data cache, application state, and through a static member variable. This information is cached at application startup and remains in the cache throughout the application s lifetime.

In this tutorial and the past two, we ve looked at caching data for the duration of the application s lifetime as well as using time-based expiries. When caching database data, though, a time-based expiry may be less than ideal. Rather than periodically flushing the cache, it would be optimal to only evict the cached item when the underlying database data is modified. This ideal is possible through the use of SQL cache dependencies, which we'll examine in our next tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Teresa Murphy and Zack Jones. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Using SQL Cache Dependencies (VB)

9/16/2020 • 25 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

The simplest caching strategy is to allow cached data to expire after a specified period of time. But this simple approach means that the cached data maintains no association with its underlying data source, resulting in stale data that is held too long or current data that is expired too soon. A better approach is to use the SqlCacheDependency class so that data remains cached until its underlying data has been modified in the SQL database. This tutorial shows you how.

Introduction

The caching techniques examined in the Caching Data with the ObjectDataSource and Caching Data in the Architecture tutorials used a time-based expiry to evict the data from the cache after a specified period. This approach is the simplest way to balance the performance gains of caching against data staleness. By selecting a time expiry of *x* seconds, a page developer concedes to enjoy the performance benefits of caching for only *x* seconds, but can rest easy that her data will never be stale longer than a maximum of *x* seconds. Of course, for static data, *x* can be extended to the lifetime of the web application, as was examined in the Caching Data at Application Startup tutorial.

When caching database data, a time-based expiry is often chosen for its ease of use but is frequently an inadequate solution. Ideally, the database data would remain cached until the underlying data has been modified in the database; only then would the cache be evicted. This approach maximizes the performance benefits of caching and minimizes the duration of stale data. However, in order to enjoy these benefits there must be some system in place that knows when the underlying database data has been modified and evicts the corresponding items from the cache. Prior to ASP.NET 2.0, page developers were responsible for implementing this system.

ASP.NET 2.0 provides a `SqlCacheDependency` class and the necessary infrastructure to determine when a change has occurred in the database so that the corresponding cached items can be evicted. There are two techniques for determining when the underlying data has changed: notification and polling. After discussing the differences between notification and polling, we'll create the infrastructure necessary to support polling and then explore how to use the `SqlCacheDependency` class in declarative and programmatically scenarios.

Understanding Notification and Polling

There are two techniques that can be used to determine when the data in a database has been modified: notification and polling. With notification, the database automatically alerts the ASP.NET runtime when the results of a particular query have been changed since the query was last executed, at which point the cached items associated with the query are evicted. With polling, the database server maintains information about when particular tables have last been updated. The ASP.NET runtime periodically polls the database to check what tables have changed since they were entered into the cache. Those tables whose data has been modified have their associated cache items evicted.

The notification option requires less setup than polling and is more granular since it tracks changes at the query level rather than at the table level. Unfortunately, notifications are only available in the full editions of Microsoft SQL Server 2005 (i.e., the non-Express editions). However, the polling option can be used for all versions of Microsoft SQL Server from 7.0 to 2005. Since these tutorials use the Express edition of SQL Server 2005, we will

focus on setting up and using the polling option. Consult the Further Reading section at the end of this tutorial for further resources on SQL Server 2005 s notification capabilities.

With polling, the database must be configured to include a table named `AspNet_SqlCacheTablesForChangeNotification` that has three columns - `tableName`, `notificationCreated`, and `changeId`. This table contains a row for each table that has data that might need to be used in a SQL cache dependency in the web application. The `tableName` column specifies the name of the table while `notificationCreated` indicates the date and time the row was added to the table. The `changeId` column is of type `int` and has an initial value of 0. Its value is incremented with each modification to the table.

In addition to the `AspNet_SqlCacheTablesForChangeNotification` table, the database also needs to include triggers on each of the tables that may appear in a SQL cache dependency. These triggers are executed whenever a row is inserted, updated, or deleted and increment the table s `changeId` value in `AspNet_SqlCacheTablesForChangeNotification`.

The ASP.NET runtime tracks the current `changeId` for a table when caching data using a `SqlCacheDependency` object. The database is periodically checked and any `SqlCacheDependency` objects whose `changeId` differs from the value in the database are evicted since a differing `changeId` value indicates that there has been a change to the table since the data was cached.

Step 1: Exploring the `aspnet_regsql.exe` Command Line Program

With the polling approach the database must be setup to contain the infrastructure described above: a predefined table (`AspNet_SqlCacheTablesForChangeNotification`), a handful of stored procedures, and triggers on each of the tables that may be used in SQL cache dependencies in the web application. These tables, stored procedures, and triggers can be created through the command line program `aspnet_regsql.exe`, which is found in the `$WINDOWS$\Microsoft.NET\Framework\version` folder. To create the `AspNet_SqlCacheTablesForChangeNotification` table and associated stored procedures, run the following from the command line:

```
/* For SQL Server authentication... */
aspnet_regsql.exe -S server -U user -P password -d database -ed
/* For Windows Authentication... */
aspnet_regsql.exe -S server -E -d database -ed
```

> **NOTE**
> To execute these commands the specified database login must be in the `db_securityadmin` and `db_ddladmin` roles. To examine the T-SQL sent to the database by the `aspnet_regsql.exe` command line program, refer to this blog entry.

For example, to add the infrastructure for polling to a Microsoft SQL Server database named `pubs` on a database server named `ScottsServer` using Windows Authentication, navigate to the appropriate directory and, from the command line, enter:

```
aspnet_regsql.exe -S ScottsServer -E -d pubs -ed
```

After the database-level infrastructure has been added, we need to add the triggers to those tables that will be used in SQL cache dependencies. Use the `aspnet_regsql.exe` command line program again, but specify the table name using the `-t` switch and instead of using the `-ed` switch use `-et`, like so:

```
/* For SQL Server authentication... */
aspnet_regsql.exe -S <i>server</i>
-U <i>user</i> -P <i>password</i> -d <i>database</i> -t <i>tableName</i> -et
/* For Windows Authentication... */
aspnet_regsql.exe -S <i>server</i>
-E -d <i>database</i> -t <i>tableName</i> -et
```

To add the triggers to the `authors` and `titles` tables on the `pubs` database on `ScottsServer`, use:

```
aspnet_regsql.exe -S ScottsServer -E -d pubs -t authors -et
aspnet_regsql.exe -S ScottsServer -E -d pubs -t titles -et
```

For this tutorial add the triggers to the `Products`, `Categories`, and `Suppliers` tables. We'll look at the particular command line syntax in Step 3.

Step 2: Referencing a Microsoft SQL Server 2005 Express Edition Database in `App_Data`

The `aspnet_regsql.exe` command line program requires the database and server name in order to add the necessary polling infrastructure. But what is the database and server name for a Microsoft SQL Server 2005 Express database that resides in the `App_Data` folder? Rather than having to discover what the database and server names are, I ve found that the simplest approach is to attach the database to the `localhost\SQLExpress` database instance and rename the data using SQL Server Management Studio. If you have one of the full versions of SQL Server 2005 installed on your machine, then you likely already have SQL Server Management Studio installed on your computer. If you only have the Express edition, you can download the free Microsoft SQL Server Management Studio Express Edition.

Start by closing Visual Studio. Next, open SQL Server Management Studio and choose to connect to the `localhost\SQLExpress` server using Windows Authentication.

Figure 1: Attach to the `localhost\SQLExpress` Server

After connecting to the server, Management Studio will show the server and have subfolders for the databases, security, and so forth. Right-click on the Databases folder and choose the Attach option. This will bring up the Attach Databases dialog box (see Figure 2). Click the Add button and select the `NORTHWND.MDF` database folder in your web application s `App_Data` folder.

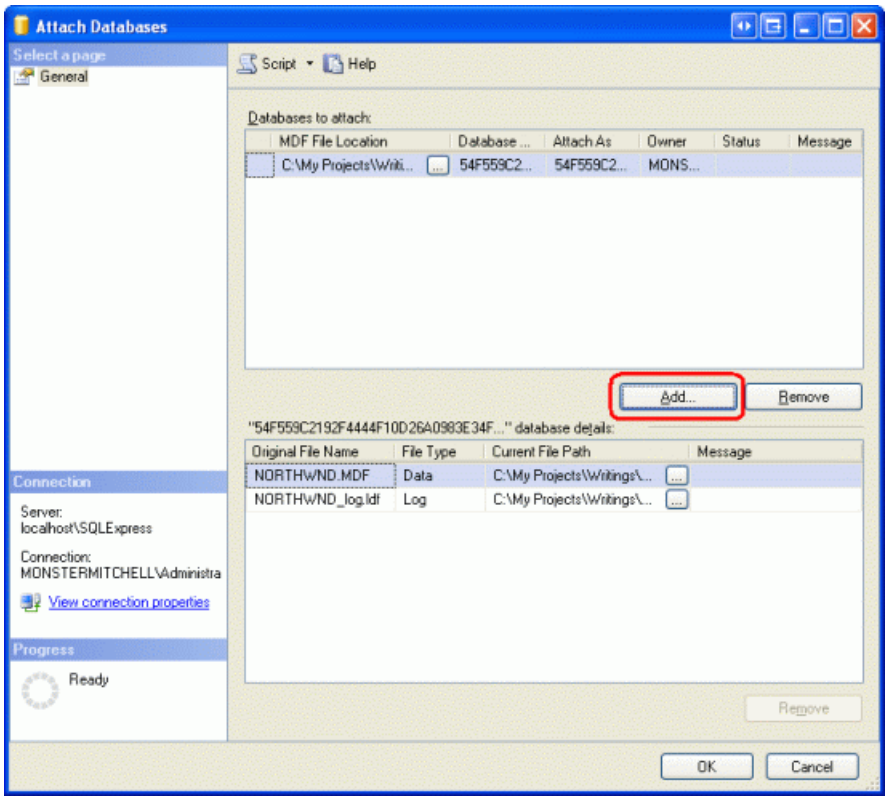


Figure 2: Attach the `NORTHWND.MDF` Database from the `App_Data` Folder (Click to view full-size image)

This will add the database to the Databases folder. The database name might be the full path to the database file or the full path prepended with a GUID. To avoid having to type in this lengthy database name when using the aspnet_regsql.exe command line tool, rename the database to a more human-friendly name by right-clicking on database just attached and choosing Rename. I ve renamed my database to DataTutorials .

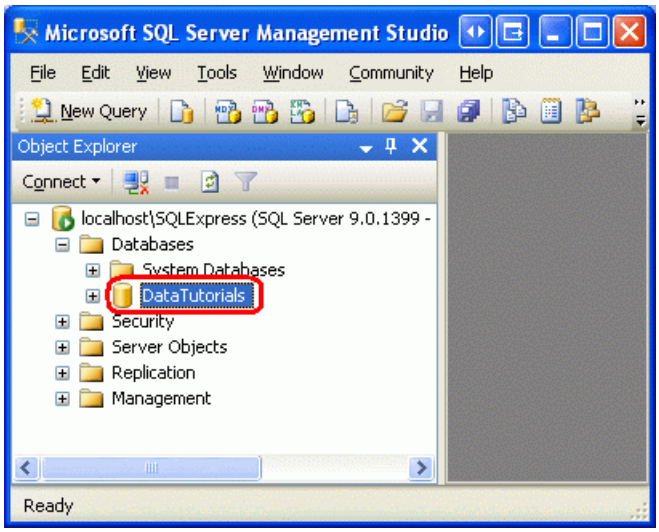


Figure 3: Rename the Attached Database to a More Human-Friendly Name

Step 3: Adding the Polling Infrastructure to the Northwind Database

Now that we have attached the `NORTHWND.MDF` database from the `App_Data` folder, we re ready to add the polling infrastructure. Assuming that you ve renamed the database to DataTutorials, run the following four commands:

```
aspnet_regsql.exe -S localhost\SQLExpress -E -d DataTutorials -ed
aspnet_regsql.exe -S localhost\SQLExpress -E -d DataTutorials -t Products -et
aspnet_regsql.exe -S localhost\SQLExpress -E -d DataTutorials -t Categories -et
aspnet_regsql.exe -S localhost\SQLExpress -E -d DataTutorials -t Suppliers -et
```

After running these four commands, right-click on the database name in Management Studio, go to the Tasks submenu, and choose Detach. Then close Management Studio and reopen Visual Studio.

Once Visual Studio has reopened, drill into the database through the Server Explorer. Note the new table (`AspNet_SqlCacheTablesForChangeNotification`), the new stored procedures, and the triggers on the `Products`, `Categories`, and `Suppliers` tables.

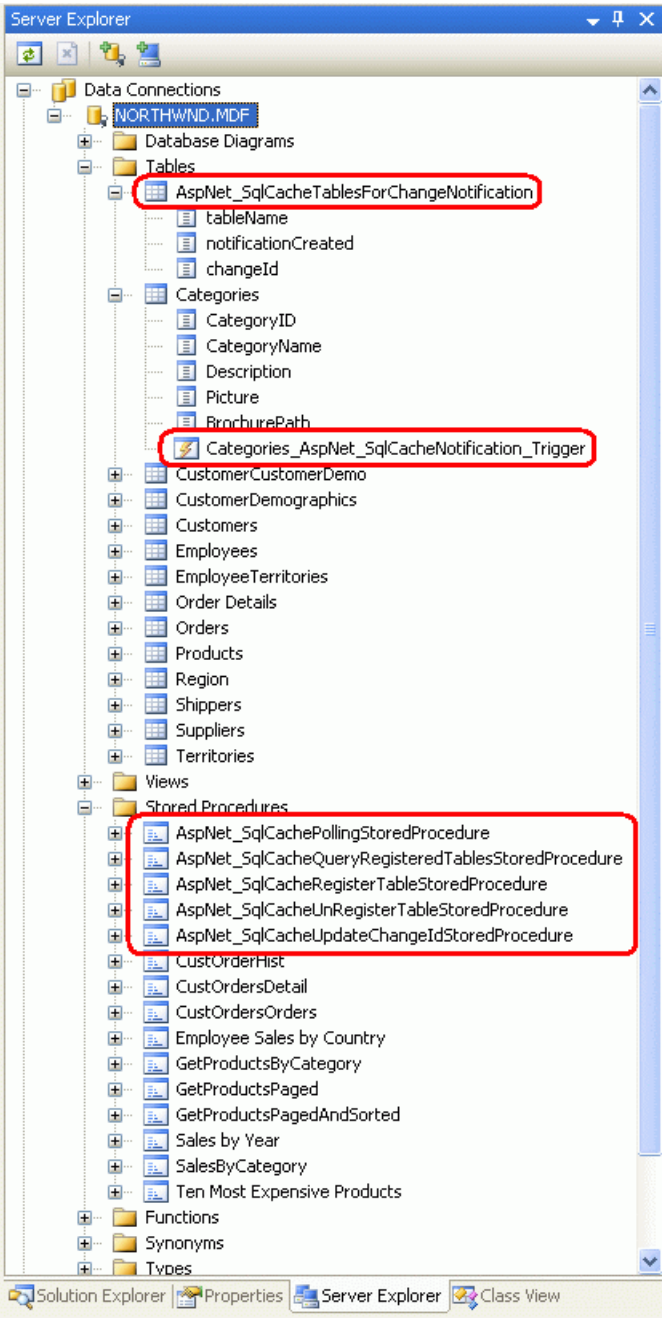


Figure 4: The Database Now Includes the Necessary Polling Infrastructure

Step 4: Configuring the Polling Service

After creating the needed tables, triggers, and stored procedures in the database, the final step is to configure the polling service, which is done through `Web.config` by specifying the databases to use and the polling frequency in milliseconds. The following markup polls the Northwind database once every second.

```
<?xml version="1.0"?>
<configuration>
   <connectionStrings>
      <add name="NORTHWNDConnectionString" connectionString=
          "Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\NORTHWND.MDF;
           Integrated Security=True;User Instance=True"
           providerName="System.Data.SqlClient"/>
   </connectionStrings>
   <system.web>
      ...
      <!-- Configure the polling service used for SQL cache dependencies -->
      <caching>
         <sqlCacheDependency enabled="true" pollTime="1000" >
            <databases>
               <add name="NorthwindDB"
                    connectionStringName="NORTHWNDConnectionString" />
            </databases>
         </sqlCacheDependency>
      </caching>
   </system.web>
</configuration>
```

The `name` value in the `<add>` element (NorthwindDB) associates a human-readable name with a particular database. When working with SQL cache dependencies, we'll need to refer to the database name defined here as well as the table that the cached data is based on. We'll see how to use the `SqlCacheDependency` class to programmatically associate SQL cache dependencies with cached data in Step 6.

Once a SQL cache dependency has been established, the polling system will connect to the databases defined in the `<databases>` elements every `pollTime` milliseconds and execute the `AspNet_SqlCachePollingStoredProcedure` stored procedure. This stored procedure - which was added back in Step 3 using the `aspnet_regsql.exe` command line tool - returns the `tableName` and `changeId` values for each record in `AspNet_SqlCacheTablesForChangeNotification`. Outdated SQL cache dependencies are evicted from the cache.

The `pollTime` setting introduces a tradeoff between performance and data staleness. A small `pollTime` value increases the number of requests to the database, but more quickly evicts stale data from the cache. A larger `pollTime` value reduces the number of database requests, but increases the delay between when the backend data changes and when the related cache items are evicted. Fortunately, the database request is executing a simple stored procedure that s returning just a few rows from a simple, lightweight table. But do experiment with different `pollTime` values to find an ideal balance between database access and data staleness for your application. The smallest `pollTime` value allowed is 500.

> **NOTE**
> The above example provides a single `pollTime` value in the `<sqlCacheDependency>` element, but you can optionally specify the `pollTime` value in the `<add>` element. This is useful if you have multiple databases specified and want to customize the polling frequency per database.

Step 5: Declaratively Working with SQL Cache Dependencies

In Steps 1 through 4 we looked at how to setup the necessary database infrastructure and configure the polling system. With this infrastructure in place, we can now add items to the data cache with an associated SQL cache dependency using either programmatic or declarative techniques. In this step we'll examine how to declaratively work with SQL cache dependencies. In Step 6 we'll look at the programmatic approach.

The Caching Data with the ObjectDataSource tutorial explored the declarative caching capabilities of the ObjectDataSource. By simply setting the `EnableCaching` property to `True` and the `CacheDuration` property to some time interval, the ObjectDataSource will automatically cache the data returned from its underlying object

for the specified interval. The ObjectDataSource can also use one or more SQL cache dependencies.

To demonstrate using SQL cache dependencies declaratively, open the `SqlCacheDependencies.aspx` page in the `Caching` folder and drag a GridView from the Toolbox onto the Designer. Set the GridView s `ID` to `ProductsDeclarative` and, from its smart tag, choose to bind it to a new ObjectDataSource named `ProductsDataSourceDeclarative`.

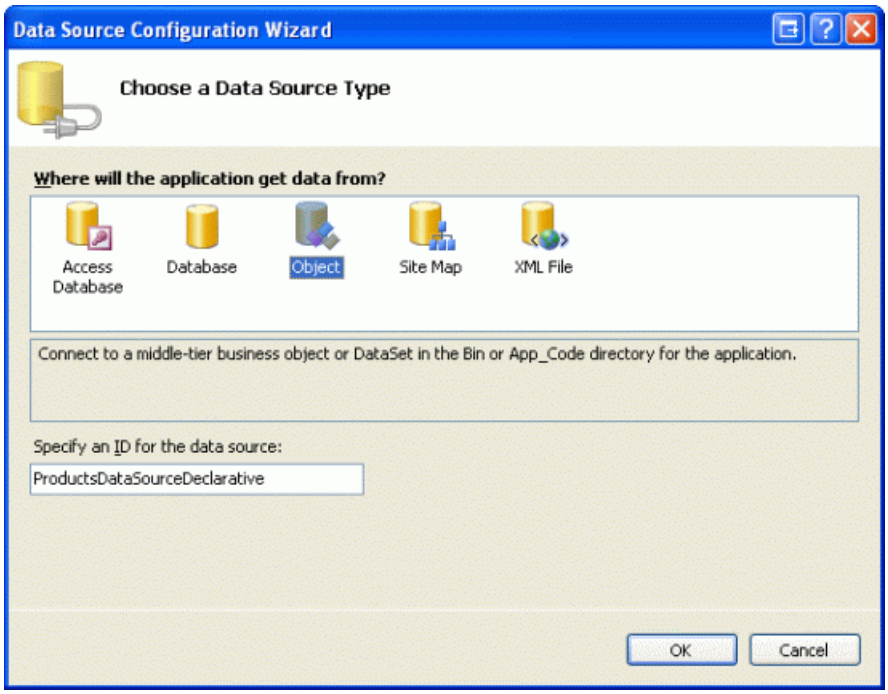


Figure 5: Create a New ObjectDataSource Named `ProductsDataSourceDeclarative` (Click to view full-size image)

Configure the ObjectDataSource to use the `ProductsBLL` class and set the drop-down list in the SELECT tab to `GetProducts()`. In the UPDATE tab, choose the `UpdateProduct` overload with three input parameters - `productName`, `unitPrice`, and `productID`. Set the drop-down lists to (None) in the INSERT and DELETE tabs.

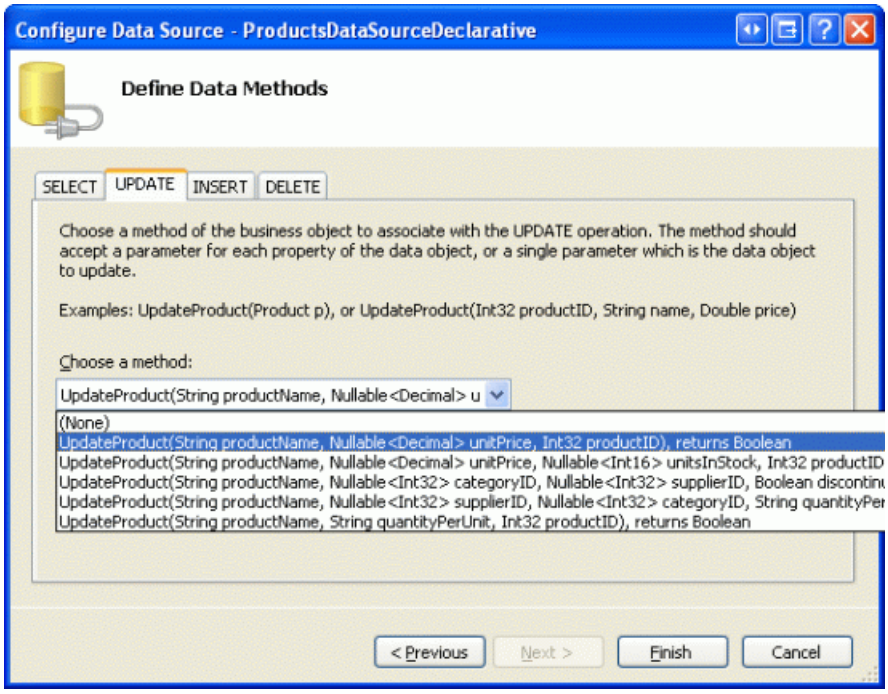


Figure 6: Use the UpdateProduct Overload with Three Input Parameters (Click to view full-size image)

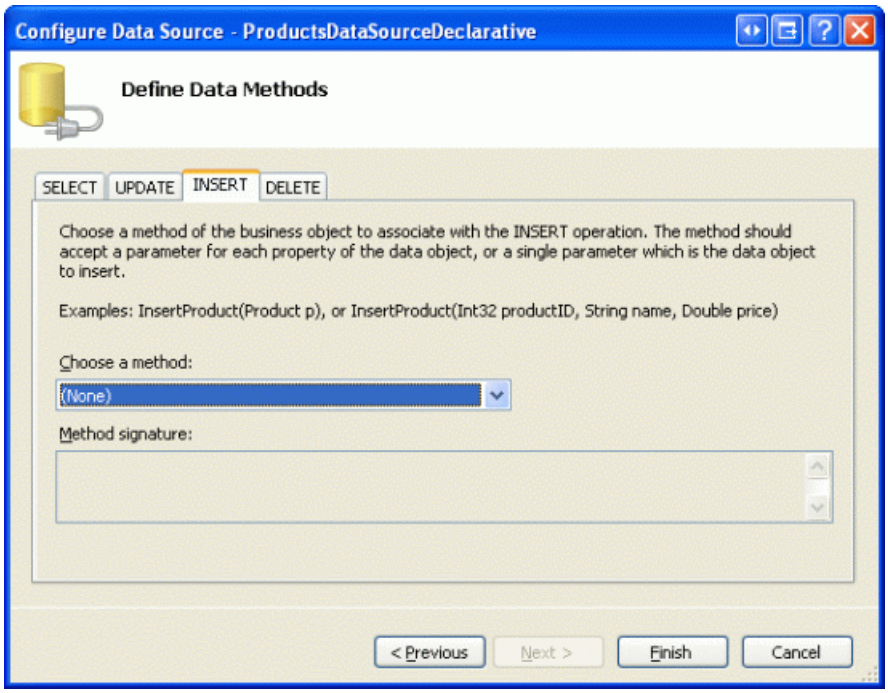


Figure 7: Set the Drop-Down List to (None) for the INSERT and DELETE Tabs (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will create BoundFields and CheckBoxFields in the GridView for each of the data fields. Remove all fields but `ProductName`, `CategoryName`, and `UnitPrice`, and format these fields as you see fit. From the GridView s smart tag, check the Enable Paging, Enable Sorting, and Enable Editing checkboxes. Visual Studio will set the ObjectDataSource s `OldValuesParameterFormatString` property to `original_{0}`. In order for the GridView s edit feature to work properly, either remove this property entirely from the declarative syntax or set it back to its default value, `{0}`.

Finally, add a Label Web control above the GridView and set its `ID` property to `ODSEvents` and its `EnableViewState` property to `False`. After making these changes, your page s declarative markup should look similar to the following. Note that I ve made a number of aesthetic customizations to the GridView fields that are not necessary to demonstrate the SQL cache dependency functionality.

```
<asp:Label ID="ODSEvents" runat="server" EnableViewState="False" />
<asp:GridView ID="ProductsDeclarative" runat="server"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ProductsDataSourceDeclarative"
    AllowPaging="True" AllowSorting="True">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
            <EditItemTemplate>
                <asp:TextBox ID="ProductName" runat="server"
                    Text='<%# Bind("ProductName") %>' />
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                    ControlToValidate="ProductName" Display="Dynamic"
                    ErrorMessage="You must provide a name for the product."
                    SetFocusOnError="True"
                    runat="server">*</asp:RequiredFieldValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Bind("ProductName") %>' />
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:TemplateField HeaderText="Price" SortExpression="UnitPrice">
            <EditItemTemplate>
                $<asp:TextBox ID="UnitPrice" runat="server" Columns="8"
                    Text='<%# Bind("UnitPrice", "{0:N2}") %>'></asp:TextBox>
                <asp:CompareValidator ID="CompareValidator1" runat="server"
                    ControlToValidate="UnitPrice"
                    ErrorMessage="You must enter a valid currency value with
                        no currency symbols. Also, the value must be greater than
                        or equal to zero."
                    Operator="GreaterThanEqual" SetFocusOnError="True"
                    Type="Currency" Display="Dynamic"
                    ValueToCompare="0">*</asp:CompareValidator>
            </EditItemTemplate>
            <ItemStyle HorizontalAlign="Right" />
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("UnitPrice", "{0:c}") %>' />
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSourceDeclarative" runat="server"
    SelectMethod="GetProducts" TypeName="ProductsBLL"
    UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

Next, create an event handler for the ObjectDataSource s `Selecting` event and in it add the following code:

```
Protected Sub ProductsDataSourceDeclarative_Selecting _
    (sender As Object, e As ObjectDataSourceSelectingEventArgs) _
    Handles ProductsDataSourceDeclarative.Selecting
    ODSEvents.Text = "-- Selecting event fired"
End Sub
```

Recall that the ObjectDataSource s `Selecting` event fires only when retrieving data from its underlying object. If

the ObjectDataSource accesses the data from its own cache, this event is not fired.

Now, visit this page through a browser. Since we ve yet to implement any caching, each time you page, sort, or edit the grid the page should display the text, 'Selecting event fired, as Figure 8 shows.

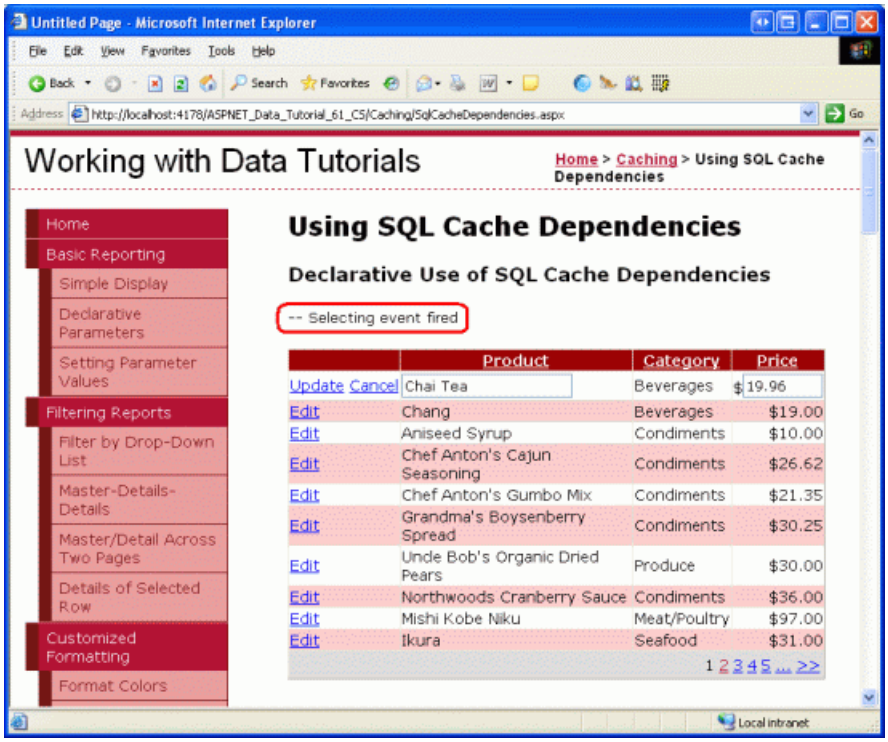


Figure 8: The ObjectDataSource s `Selecting` Event Fires Each Time the GridView is Paged, Edited, or Sorted (Click to view full-size image)

As we saw in the Caching Data with the ObjectDataSource tutorial, setting the `EnableCaching` property to `True` causes the ObjectDataSource to cache its data for the duration specified by its `CacheDuration` property. The ObjectDataSource also has a `SqlCacheDependency` property, which adds one or more SQL cache dependencies to the cached data using the pattern:

```
databaseName1:tableName1;databaseName2:tableName2;...
```

Where *databaseName* is the name of the database as specified in the `name` attribute of the `<add>` element in `Web.config`, and *tableName* is the name of the database table. For example, to create an ObjectDataSource that caches data indefinitely based on a SQL cache dependency against the Northwind s `Products` table, set the ObjectDataSource s `EnableCaching` property to `True` and its `SqlCacheDependency` property to NorthwindDB:Products .

> **NOTE**
> You can use a SQL cache dependency *and* a time-based expiry by setting `EnableCaching` to `True`, `CacheDuration` to the time interval, and `SqlCacheDependency` to the database and table name(s). The ObjectDataSource will evict its data when the time-based expiry is reached or when the polling system notes that the underlying database data has changed, whichever happens first.

The GridView in `SqlCacheDependencies.aspx` displays data from two tables - `Products` and `Categories` (the product s `CategoryName` field is retrieved via a `JOIN` on `Categories`). Therefore, we want to specify two SQL cache dependencies: NorthwindDB:Products;NorthwindDB:Categories .

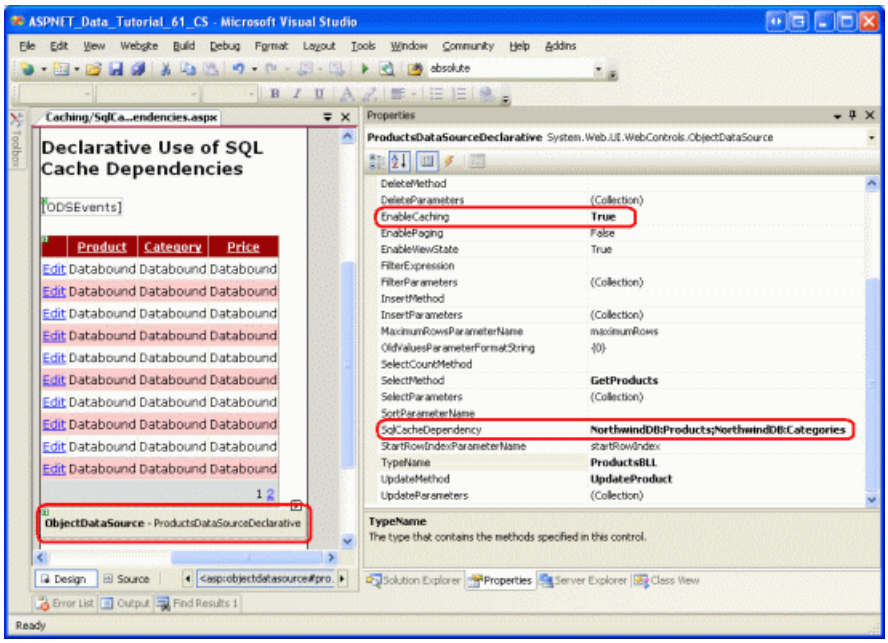


Figure 9: Configure the ObjectDataSource to Support Caching Using SQL Cache Dependencies on `Products` and `Categories` (Click to view full-size image)

After configuring the ObjectDataSource to support caching, revisit the page through a browser. Again, the text 'Selecting event fired should appear on the first page visit, but should go away when paging, sorting, or clicking the Edit or Cancel buttons. This is because after the data is loaded into the ObjectDataSource s cache, it remains there until the `Products` or `Categories` tables are modified or the data is updated through the GridView.

After paging through the grid and noting the lack of the 'Selecting event fired text, open a new browser window and navigate to the Basics tutorial in the Editing, Inserting, and Deleting section (`~/EditInsertDelete/Basics.aspx`). Update the name or price of a product. Then, from to the first browser window, view a different page of data, sort the grid, or click a row s Edit button. This time, the 'Selecting event fired should reappear, as the underlying database data has been modified (see Figure 10). If the text does not appear, wait a few moments and try again. Remember that the polling service is checking for changes to the `Products` table every `pollTime` milliseconds, so there is a delay between when the underlying data is updated and when the cached data is evicted.

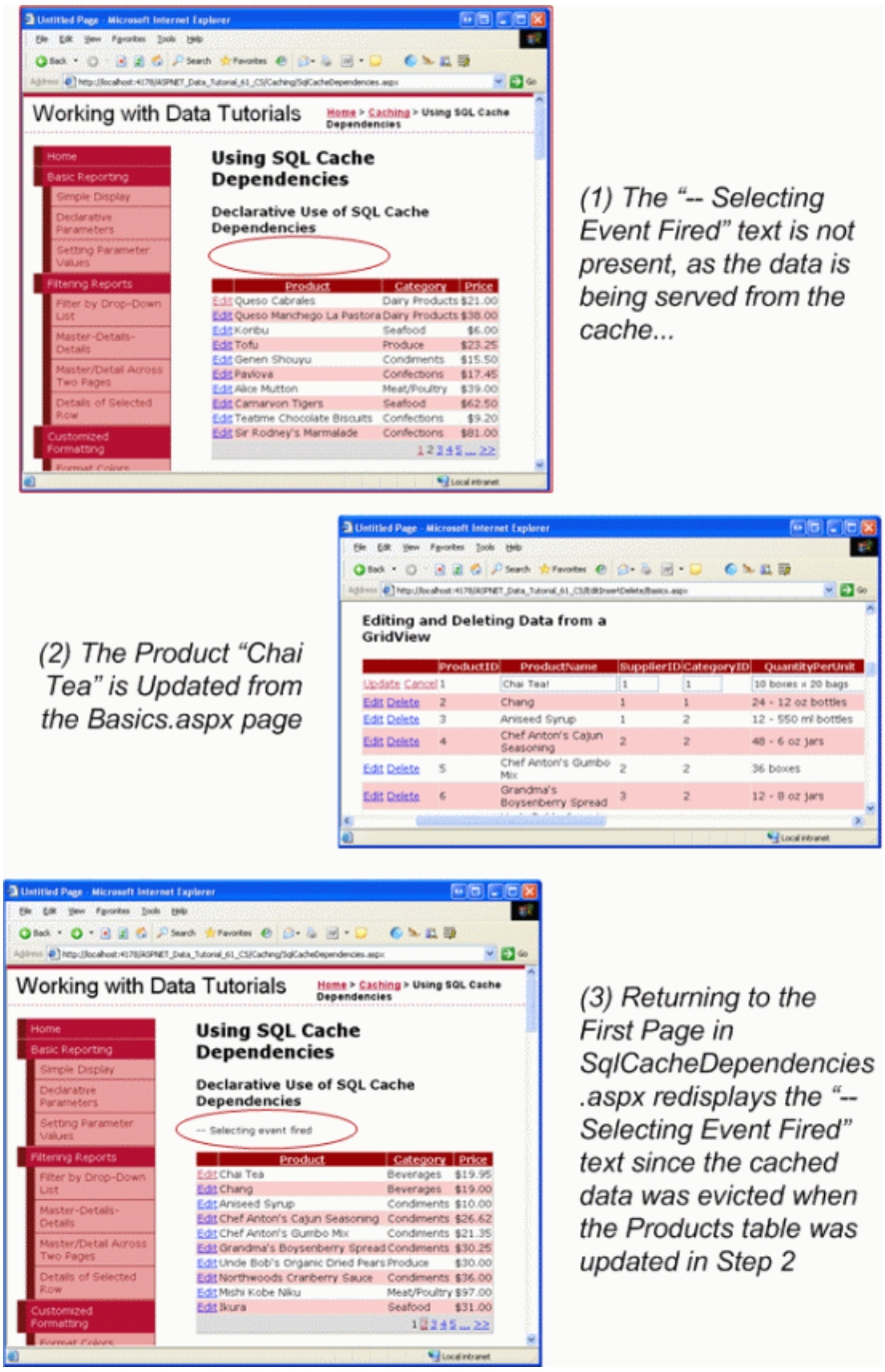


Figure 10: Modifying the Products Table Evicts the Cached Product Data (Click to view full-size image)

Step 6: Programmatically Working with the `SqlCacheDependency` Class

The Caching Data in the Architecture tutorial looked at the benefits of using a separate Caching Layer in the architecture as opposed to tightly coupling the caching with the ObjectDataSource. In that tutorial we created a `ProductsCL` class to demonstrate programmatically working with the data cache. To utilize SQL cache dependencies in the Caching Layer, use the `SqlCacheDependency` class.

With the polling system, a `SqlCacheDependency` object must be associated with a particular database and table pair. The following code, for example, creates a `SqlCacheDependency` object based on the Northwind database s `Products` table:

```
Dim productsTableDependency As _
    New Caching.SqlCacheDependency("NorthwindDB", "Products")
```

The two input parameters to the `SqlCacheDependency` s constructor are the database and table names,

respectively. Like with the ObjectDataSource s `SqlCacheDependency` property, the database name used is the same as the value specified in the `name` attribute of the `<add>` element in `Web.config`. The table name is the actual name of the database table.

To associate a `SqlCacheDependency` with an item added to the data cache, use one of the `Insert` method overloads that accepts a dependency. The following code adds *value* to the data cache for an indefinite duration, but associates it with a `SqlCacheDependency` on the `Products` table. In short, *value* will remain in the cache until it is evicted due to memory constraints or because the polling system has detected that the `Products` table has changed since it was cached.

```
Dim productsTableDependency As _
    New Caching.SqlCacheDependency("NorthwindDB", "Products")
Cache.Insert(key, _
             value, _
             productsTableDependency, _
             System.Web.Caching.Cache.NoAbsoluteExpiration, _
             System.Web.Caching.Cache.NoSlidingExpiration)
```

The Caching Layer s `ProductsCL` class currently caches data from the `Products` table using a time-based expiry of 60 seconds. Let s update this class so that it uses SQL cache dependencies instead. The `ProductsCL` class s `AddCacheItem` method, which is responsible for adding the data to the cache, currently contains the following code:

```
Private Sub AddCacheItem(ByVal rawKey As String, ByVal value As Object)
    Dim DataCache As System.Web.Caching.Cache = HttpRuntime.Cache
    ' Make sure MasterCacheKeyArray(0) is in the cache - if not, add it
    If DataCache(MasterCacheKeyArray(0)) Is Nothing Then
        DataCache(MasterCacheKeyArray(0)) = DateTime.Now
    End If
    ' Add a CacheDependency
    Dim dependency As _
        New Caching.CacheDependency(Nothing, MasterCacheKeyArray)
    DataCache.Insert(GetCacheKey(rawKey), value, dependency, _
        DateTime.Now.AddSeconds(CacheDuration), _
        Caching.Cache.NoSlidingExpiration)
End Sub
```

Update this code to use a `SqlCacheDependency` object instead of the `MasterCacheKeyArray` cache dependency:

```
Private Sub AddCacheItem(ByVal rawKey As String, ByVal value As Object)
    Dim DataCache As System.Web.Caching.Cache = HttpRuntime.Cache
    ' Add the SqlCacheDependency objects for Products
    Dim productsTableDependency As New _
        Caching.SqlCacheDependency("NorthwindDB", "Products")
    DataCache.Insert(GetCacheKey(rawKey), value, productsTableDependency, _
        Cache.NoAbsoluteExpiration, Caching.Cache.NoSlidingExpiration)
End Sub
```

To test this functionality, add a GridView to the page beneath the existing `ProductsDeclarative` GridView. Set this new GridView s `ID` to `ProductsProgrammatic` and, through its smart tag, bind it to a new ObjectDataSource named `ProductsDataSourceProgrammatic`. Configure the ObjectDataSource to use the `ProductsCL` class, setting the drop-down lists in the SELECT and UPDATE tabs to `GetProducts` and `UpdateProduct`, respectively.

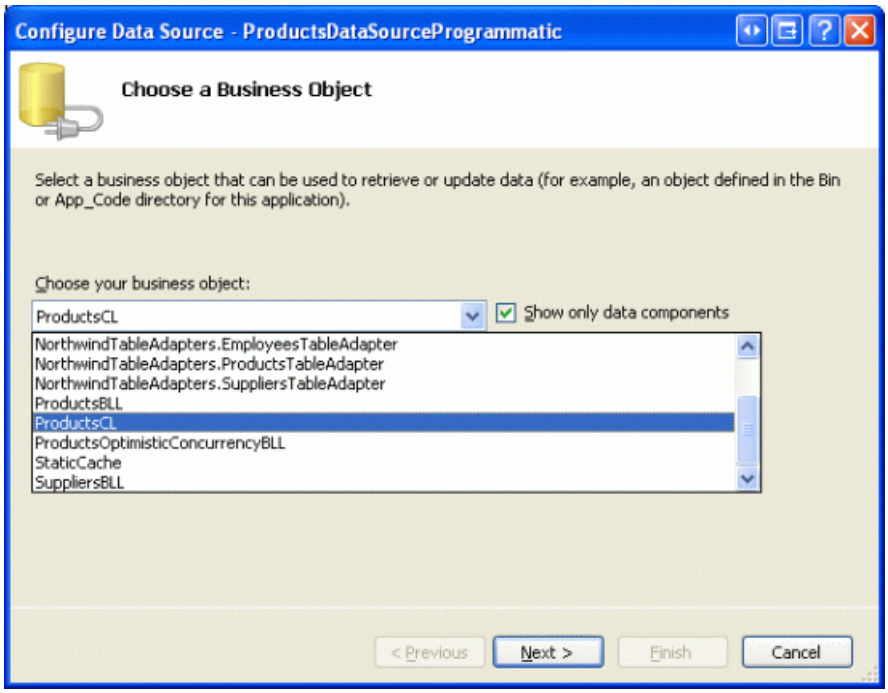


Figure 11: Configure the ObjectDataSource to Use the `ProductsCL` Class (Click to view full-size image)

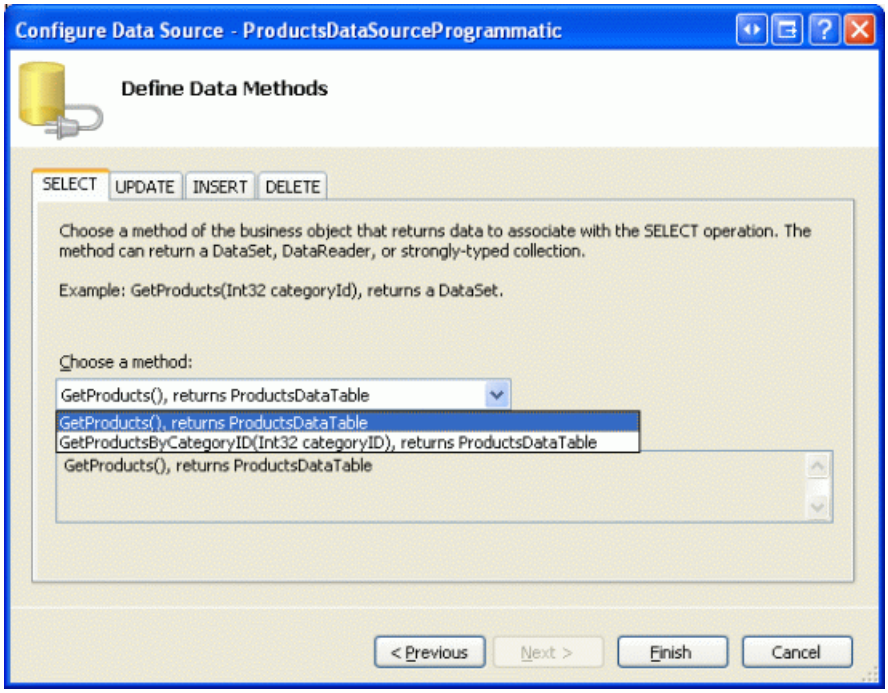


Figure 12: Select the `GetProducts` Method from the SELECT Tab s Drop-Down List (Click to view full-size image)

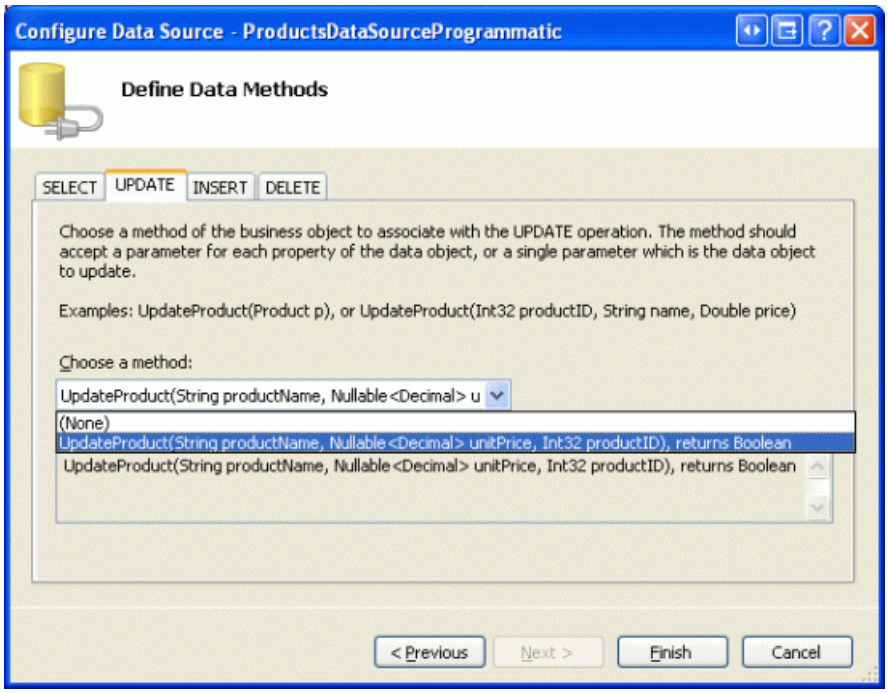


Figure 13: Choose the UpdateProduct Method from the UPDATE Tab s Drop-Down List (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will create BoundFields and CheckBoxFields in the GridView for each of the data fields. Like with the first GridView added to this page, remove all fields but `ProductName`, `CategoryName`, and `UnitPrice`, and format these fields as you see fit. From the GridView s smart tag, check the Enable Paging, Enable Sorting, and Enable Editing checkboxes. As with the `ProductsDataSourceDeclarative` ObjectDataSource, Visual Studio will set the `ProductsDataSourceProgrammatic` ObjectDataSource s `OldValuesParameterFormatString` property to `original_{0}`. In order for the GridView s edit feature to work properly, set this property back to `{0}` (or remove the property assignment from the declarative syntax altogether).

After completing these tasks, the resulting GridView and ObjectDataSource declarative markup should look like the following:

```
<asp:GridView ID="ProductsProgrammatic" runat="server"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ProductsDataSourceProgrammatic" AllowPaging="True"
    AllowSorting="True">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
            <EditItemTemplate>
                <asp:TextBox ID="ProductName" runat="server"
                    Text='<%# Bind("ProductName") %>' />
                <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                    ControlToValidate="ProductName" Display="Dynamic"
                    ErrorMessage="You must provide a name for the product."
                    SetFocusOnError="True"
                    runat="server">*</asp:RequiredFieldValidator>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label2" runat="server"
                    Text='<%# Bind("ProductName") %>' />
            </ItemTemplate>
        </asp:TemplateField>
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:TemplateField HeaderText="Price" SortExpression="UnitPrice">
            <EditItemTemplate>
                $<asp:TextBox ID="UnitPrice" runat="server" Columns="8"
                    Text='<%# Bind("UnitPrice", "{0:N2}") %>'></asp:TextBox>
                <asp:CompareValidator ID="CompareValidator1" runat="server"
                    ControlToValidate="UnitPrice" Display="Dynamic"
                    ErrorMessage="You must enter a valid currency value with no
                        currency symbols. Also, the value must be greater than
                        or equal to zero."
                    Operator="GreaterThanEqual" SetFocusOnError="True"
                    Type="Currency" ValueToCompare="0">*</asp:CompareValidator>
            </EditItemTemplate>
            <ItemStyle HorizontalAlign="Right" />
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
                    Text='<%# Bind("UnitPrice", "{0:c}") %>' />
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSourceProgrammatic" runat="server"
    OldValuesParameterFormatString="{0}" SelectMethod="GetProducts"
    TypeName="ProductsCL" UpdateMethod="UpdateProduct">
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

To test the SQL cache dependency in the Caching Layer set a breakpoint in the `ProductCL` class s `AddCacheItem` method and then start debugging. When you first visit `SqlCacheDependencies.aspx`, the breakpoint should be hit as the data is requested for the first time and placed into the cache. Next, move to another page in the GridView or sort one of the columns. This causes the GridView to requery its data, but the data should be found in the cache since the `Products` database table has not been modified. If the data is repeatedly not found in the cache, make sure there is sufficient memory available on your computer and try again.

After paging through a few pages of the GridView, open a second browser window and navigate to the Basics tutorial in the Editing, Inserting, and Deleting section (`~/EditInsertDelete/Basics.aspx`). Update a record from the Products table and then, from the first browser window, view a new page or click on one of the sorting headers.

In this scenario you will see one of two things: either the breakpoint will be hit, indicating that the cached data was evicted due to the change in the database; or, the breakpoint will not be hit, meaning that `SqlCacheDependencies.aspx` is now showing stale data. If the breakpoint is not hit, it is likely because the polling service has not yet fired since the data was changed. Remember that the polling service is checking for changes to the `Products` table every `pollTime` milliseconds, so there is a delay between when the underlying data is updated and when the cached data is evicted.

> **NOTE**
>
> This delay is more likely to appear when editing one of the products through the GridView in `SqlCacheDependencies.aspx`. In the Caching Data in the Architecture tutorial we added the `MasterCacheKeyArray` cache dependency to ensure that the data being edited through the `ProductsCL` class s `UpdateProduct` method was evicted from the cache. However, we replaced this cache dependency when modifying the `AddCacheItem` method earlier in this step and therefore the `ProductsCL` class will continue to show the cached data until the polling system notes the change to the `Products` table. We'll see how to reintroduce the `MasterCacheKeyArray` cache dependency in Step 7.

Step 7: Associating Multiple Dependencies with a Cached Item

Recall that the `MasterCacheKeyArray` cache dependency is used to ensure that *all* product-related data is evicted from the cache when any single item associated within it is updated. For example, the `GetProductsByCategoryID(categoryID)` method caches `ProductsDataTables` instances for each unique *categoryID* value. If one of these objects is evicted, the `MasterCacheKeyArray` cache dependency ensures that the others are also removed. Without this cache dependency, when the cached data is modified the possibility exists that other cached product data may be out of date. Consequently, it s important that we maintain the `MasterCacheKeyArray` cache dependency when using SQL cache dependencies. However, the data cache s `Insert` method only allows for a single dependency object.

Furthermore, when working with SQL cache dependencies we may need to associate multiple database tables as dependencies. For example, the `ProductsDataTable` cached in the `ProductsCL` class contains the category and supplier names for each product, but the `AddCacheItem` method only uses a dependency on `Products`. In this situation, if the user updates the name of a category or supplier, the cached product data will remain in the cache and be out of date. Therefore, we want to make the cached product data dependent on not only the `Products` table, but on the `Categories` and `Suppliers` tables as well.

The `AggregateCacheDependency` class provides a means for associating multiple dependencies with a cache item. Start by creating an `AggregateCacheDependency` instance. Next, add the set of dependencies using the `AggregateCacheDependency` s `Add` method. When inserting the item into the data cache thereafter, pass in the `AggregateCacheDependency` instance. When *any* of the `AggregateCacheDependency` instance s dependencies change, the cached item will be evicted.

The following shows the updated code for the `ProductsCL` class s `AddCacheItem` method. The method creates the `MasterCacheKeyArray` cache dependency along with `SqlCacheDependency` objects for the `Products`, `Categories`, and `Suppliers` tables. These are all combined into one `AggregateCacheDependency` object named `aggregateDependencies`, which is then passed into the `Insert` method.

```
Private Sub AddCacheItem(ByVal rawKey As String, ByVal value As Object)
    Dim DataCache As System.Web.Caching.Cache = HttpRuntime.Cache
    ' Make sure MasterCacheKeyArray(0) is in the cache - if not, add it.
    If DataCache(MasterCacheKeyArray(0)) Is Nothing Then
        DataCache(MasterCacheKeyArray(0)) = DateTime.Now
    End If
    'Create the CacheDependency
    Dim masterCacheKeyDependency As _
        New Caching.CacheDependency(Nothing, MasterCacheKeyArray)
    ' Add the SqlCacheDependency objects for Products, Categories, and Suppliers
    Dim productsTableDependency As _
        New Caching.SqlCacheDependency("NorthwindDB", "Products")
    Dim categoriesTableDependency As _
        New Caching.SqlCacheDependency("NorthwindDB", "Categories")
    Dim suppliersTableDependency As _
        New Caching.SqlCacheDependency("NorthwindDB", "Suppliers")
    ' Create an AggregateCacheDependency
    Dim aggregateDependencies As New Caching.AggregateCacheDependency()
    aggregateDependencies.Add(masterCacheKeyDependency, productsTableDependency, _
        categoriesTableDependency, suppliersTableDependency)
    DataCache.Insert(GetCacheKey(rawKey), value, aggregateDependencies, _
        Caching.Cache.NoAbsoluteExpiration, Caching.Cache.NoSlidingExpiration)
End Sub
```

Test this new code out. Now changes to the `Products`, `Categories`, or `Suppliers` tables cause the cached data to be evicted. Moreover, the `ProductsCL` class s `UpdateProduct` method, which is called when editing a product through the GridView, evicts the `MasterCacheKeyArray` cache dependency, which causes the cached `ProductsDataTable` to be evicted and the data to be re-retrieved on the next request.

> **NOTE**
>
> SQL cache dependencies can also be used with output caching. For a demonstration of this functionality, see: Using ASP.NET Output Caching with SQL Server.

Summary

When caching database data, the data will ideally remain in the cache until it is modified in the database. With ASP.NET 2.0, SQL cache dependencies can be created and used in both declarative and programmatic scenarios. One of the challenges with this approach is in discovering when the data has been modified. The full versions of Microsoft SQL Server 2005 provide notification capabilities that can alert an application when a query result has changed. For the Express Edition of SQL Server 2005 and older versions of SQL Server, a polling system must be used instead. Fortunately, setting up the necessary polling infrastructure is fairly straightforward.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Using Query Notifications in Microsoft SQL Server 2005
- Creating a Query Notification
- Caching in ASP.NET with the `SqlCacheDependency` Class
- ASP.NET SQL Server Registration Tool (`aspnet_regsql.exe`)
- Overview of `SqlCacheDependency`

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Marko Rangel, Teresa Murphy, and Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Database-Driven Site Maps

2/20/2020 • 2 minutes to read • Edit Online

This tutorial explains how to build a site map dynamically based on information in the database.

- Building a Custom Database-Driven Site Map Provider (C#)
- Building a Custom Database-Driven Site Map Provider (VB)

Building a Custom Database-Driven Site Map Provider (C#)

2/20/2020 • 27 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> The default site map provider in ASP.NET 2.0 retrieves its data from a static XML file. While the XML-based provider is suitable to many small and medium-sized Web sites, larger Web applications require a more dynamic site map. In this tutorial we'll build a custom site map provider that retrieves its data from the Business Logic Layer, which in turn retrieves data from the database.

Introduction

ASP.NET 2.0 s site map feature enables a page developer to define a web application s site map in some persistent medium, such as in an XML file. Once defined, the site map data can be accessed programmatically through the `SiteMap` class in the `System.Web` namespace or through a variety of navigation Web controls, such as the SiteMapPath, Menu, and TreeView controls. The site map system uses the provider model so that different site map serialization implementations can be created and plugged into a web application. The default site map provider that ships with ASP.NET 2.0 persists site map structure in an XML file. Back in the Master Pages and Site Navigation tutorial we created a file named `Web.sitemap` that contained this structure and have been updating its XML with each new tutorial section.

The default XML-based site map provider works well if the site map s structure is fairly static, such as for these tutorials. In many scenarios, however, a more dynamic site map is needed. Consider the site map shown in Figure 1, where each category and product appear as sections in the website s structure. With this site map, visiting the web page corresponding to the root node might list all of the categories, whereas visiting a particular category s web page would list that category s products and viewing a particular product s web page would show that product s details.

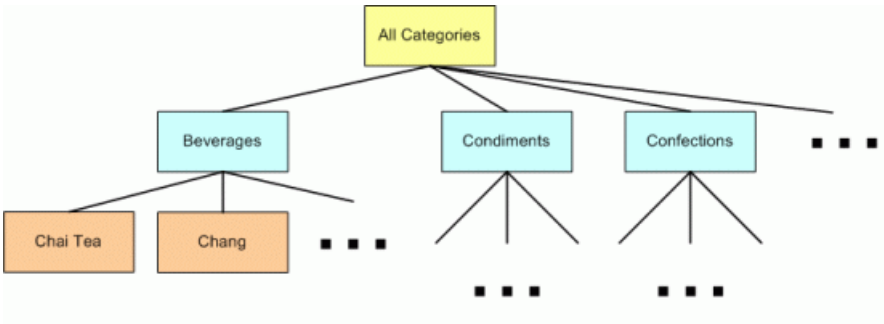


Figure 1: The Categories and Products Makeup the Site Map s Structure (Click to view full-size image)

While this category- and product-based structure could be hard-coded into the `Web.sitemap` file, the file would need to be updated each time a category or product was added, removed, or renamed. Consequently, the site map maintenance would be greatly simplified if its structure was retrieved from the database or, ideally, from the Business Logic Layer of the application s architecture. That way, as products and categories were added, renamed, or deleted, the site map would automatically update to reflect these changes.

Since ASP.NET 2.0 s site map serialization is built atop the provider model, we can create our own custom site map provider that grabs its data from an alternate data store, such as the database or architecture. In this tutorial we'll

build a custom provider that retrieves its data from the BLL. Let s get started!

> **NOTE**
> The custom site map provider created in this tutorial is tightly coupled to the application s architecture and data model. Jeff Prosise s Storing Site Maps in SQL Server and The SQL Site Map Provider You ve Been Waiting For articles examine a generalized approach to storing site map data in SQL Server.

Step 1: Creating the Custom Site Map Provider Web Pages

Before we start creating a custom site map provider, let s first add the ASP.NET pages we'll need for this tutorial. Start by adding a new folder named `SiteMapProvider`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `ProductsByCategory.aspx`
- `ProductDetails.aspx`

Also add a `CustomProviders` subfolder to the `App_Code` folder.

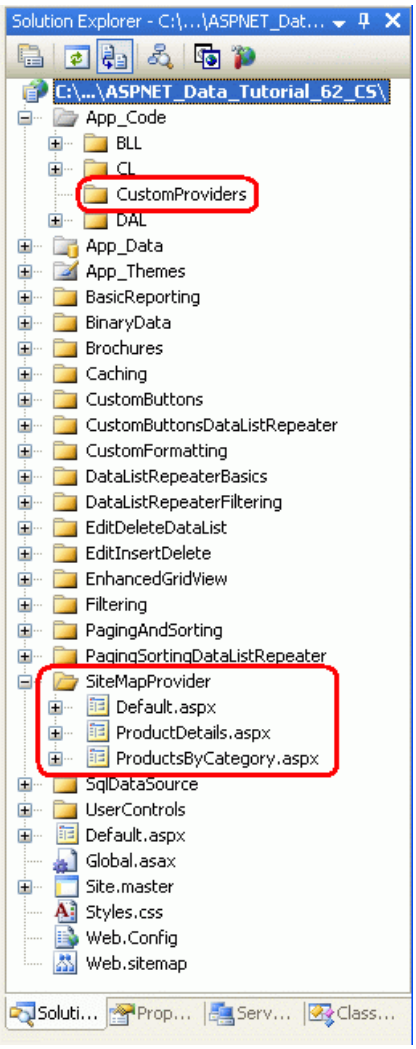


Figure 2: Add the ASP.NET Pages for the Site Map Provider-Related Tutorials

Since there is only one tutorial for this section, we don t need `Default.aspx` to list the section s tutorials. Instead, `Default.aspx` will display the categories in a GridView control. We'll tackle this in Step 2.

Next, update `Web.sitemap` to include a reference to the `Default.aspx` page. Specifically, add the following markup after the Caching `<siteMapNode>`:

```
<siteMapNode
    title="Customizing the Site Map" url="~/SiteMapProvider/Default.aspx"
    description="Learn how to create a custom provider that retrieves the site map
                 from the Northwind database." />
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes an item for the sole site map provider tutorial.

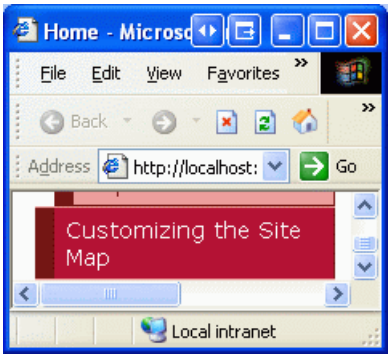


Figure 3: The Site Map Now Includes an Entry for the Site Map Provider Tutorial

This tutorial s main focus is to illustrate creating a custom site map provider and configuring a web application to use that provider. In particular, we'll build a provider that returns a site map that includes a root node along with a node for each category and product, as depicted in Figure 1. In general, each node in the site map may specify a URL. For our site map, the root node s URL will be `~/SiteMapProvider/Default.aspx`, which will list all of the categories in the database. Each category node in the site map will have a URL that points to `~/SiteMapProvider/ProductsByCategory.aspx?CategoryID=categoryID`, which will list all of the products in the specified *categoryID*. Finally, each product site map node will point to `~/SiteMapProvider/ProductDetails.aspx?ProductID=productID`, which will display the specific product s details.

To start we need to create the `Default.aspx`, `ProductsByCategory.aspx`, and `ProductDetails.aspx` pages. These pages are completed in Steps 2, 3, and 4, respectively. Since the thrust of this tutorial is on site map providers, and since past tutorials have covered creating these sorts of multi-page master/detail reports, we will hurry through Steps 2 through 4. If you need a refresher on creating master/detail reports that span multiple pages, refer back to the Master/Detail Filtering Across Two Pages tutorial.

Step 2: Displaying a List of Categories

Open the `Default.aspx` page in the `SiteMapProvider` folder and drag a GridView from the Toolbox onto the Designer, setting its `ID` to `Categories`. From the GridView s smart tag, bind it to a new ObjectDataSource named `CategoriesDataSource` and configure it so that it retrieves its data using the `CategoriesBLL` class s `GetCategories` method. Since this GridView just displays the categories and does not provide data modification capabilities, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) .

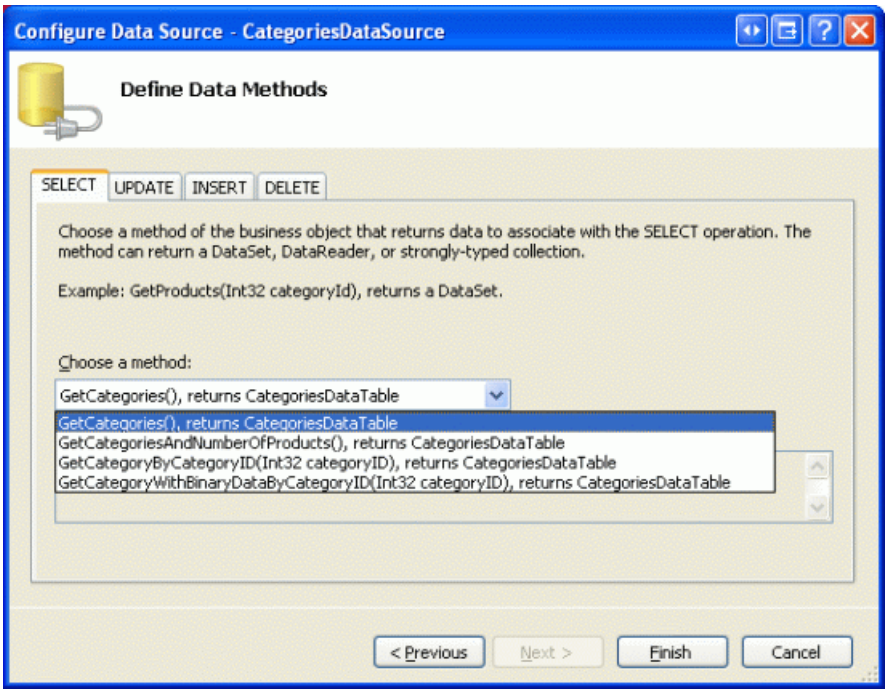


Figure 4: Configure the ObjectDataSource to Return Categories Using the `GetCategories` Method (Click to view full-size image)

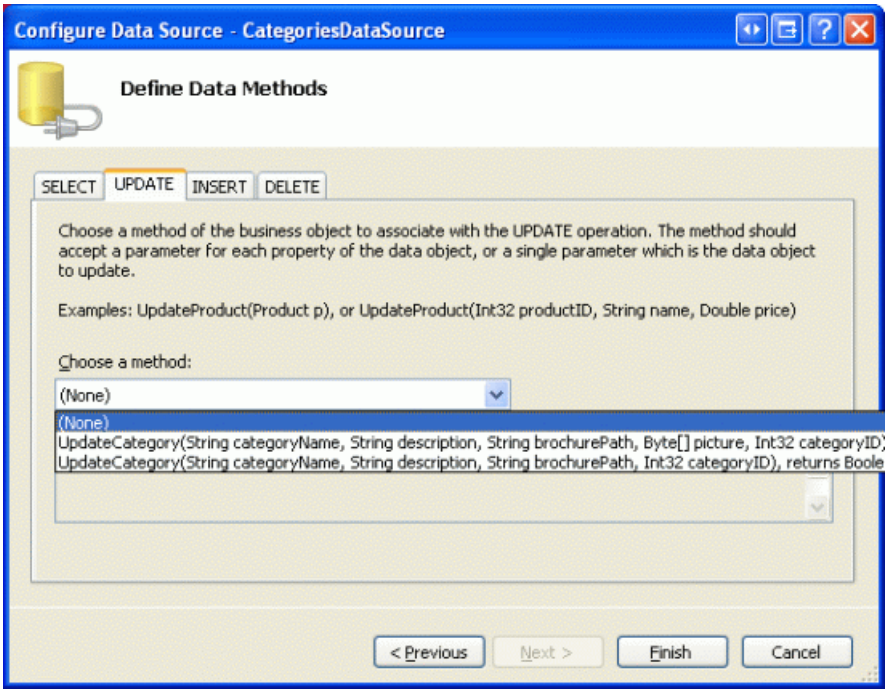


Figure 5: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE Tabs to (None) (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will add a BoundField for `CategoryID`, `CategoryName`, `Description`, `NumberOfProducts`, and `BrochurePath`. Edit the GridView so that it only contains the `CategoryName` and `Description` BoundFields and update the `CategoryName` BoundField s `HeaderText` property to Category .

Next, add a HyperLinkField and position it so that it s the left-most field. Set the `DataNavigateUrlFields` property to `CategoryID` and the `DataNavigateUrlFormatString` property to `~/SiteMapProvider/ProductsByCategory.aspx?CategoryID={0}`. Set the `Text` property to View Products .

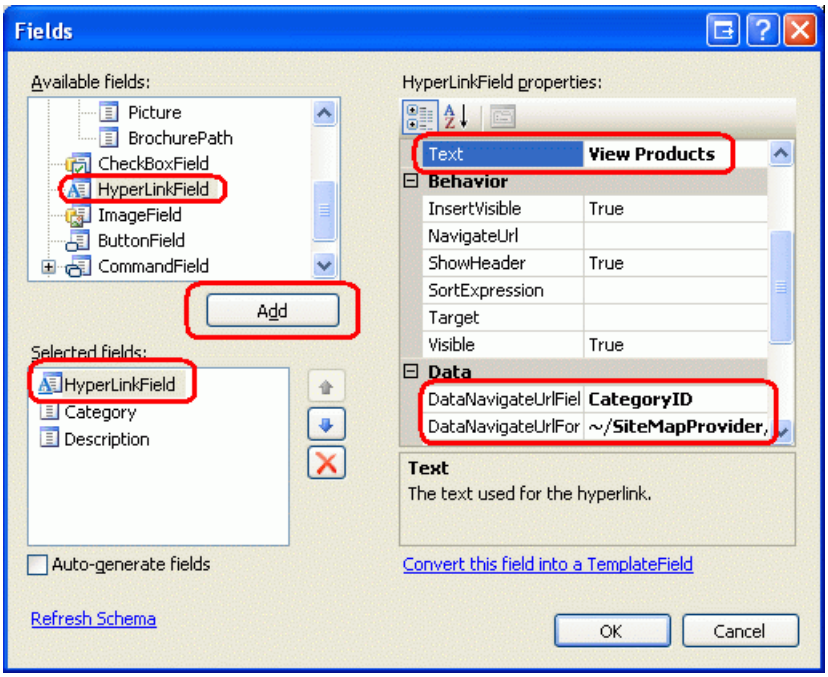


Figure 6: Add a HyperLinkField to the `Categories` GridView

After creating the ObjectDataSource and customizing the GridView s fields, the two controls declarative markup will look like the following:

```
<asp:GridView ID="Categories" runat="server" AutoGenerateColumns="False"
    DataKeyNames="CategoryID" DataSourceID="CategoriesDataSource"
    EnableViewState="False">
    <Columns>
        <asp:HyperLinkField DataNavigateUrlFields="CategoryID"
            DataNavigateUrlFormatString=
                "~/SiteMapProvider/ProductsByCategory.aspx?CategoryID={0}"
            Text="View Products" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            SortExpression="CategoryName" />
        <asp:BoundField DataField="Description" HeaderText="Description"
            SortExpression="Description" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}" SelectMethod="GetCategories"
    TypeName="CategoriesBLL"></asp:ObjectDataSource>
```

Figure 7 shows `Default.aspx` when viewed through a browser. Clicking a category s View Products link takes you to `ProductsByCategory.aspx?CategoryID=categoryID`, which we will build in Step 3.

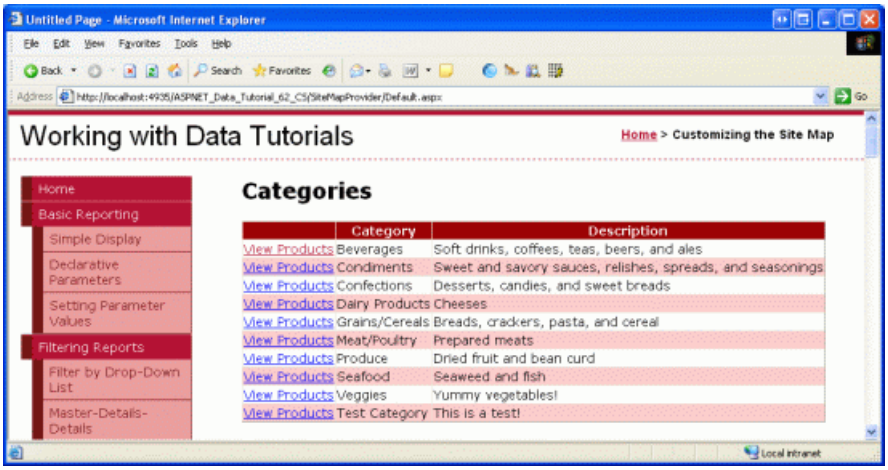


Figure 7: Each Category is Listed Along with a View Products Link (Click to view full-size image)

Step 3: Listing the Selected Category s Products

Open the `ProductsByCategory.aspx` page and add a GridView, naming it `ProductsByCategory`. From its smart tag, bind the GridView to a new ObjectDataSource named `ProductsByCategoryDataSource`. Configure the ObjectDataSource to use the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method and set the drop-down lists to (None) in the UPDATE, INSERT, and DELETE tabs.

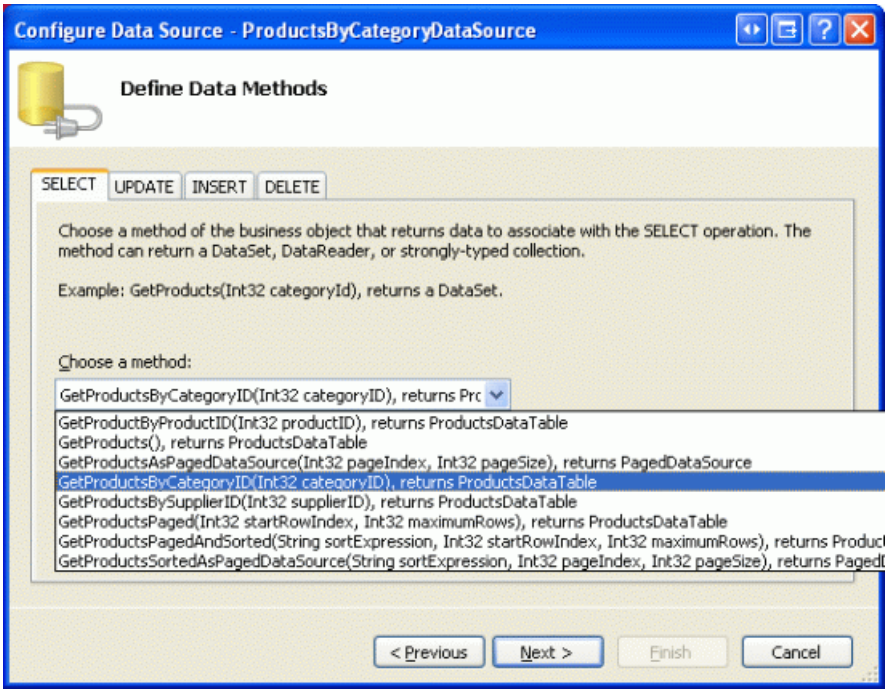


Figure 8: Use the `ProductsBLL` Class s `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

The final step in the Configure Data Source wizard prompts for a parameter source for *categoryID*. Since this information is passed through the querystring field `CategoryID`, select QueryString from the drop-down list and enter CategoryID in the QueryStringField textbox as shown in Figure 9. Click Finish to complete the wizard.

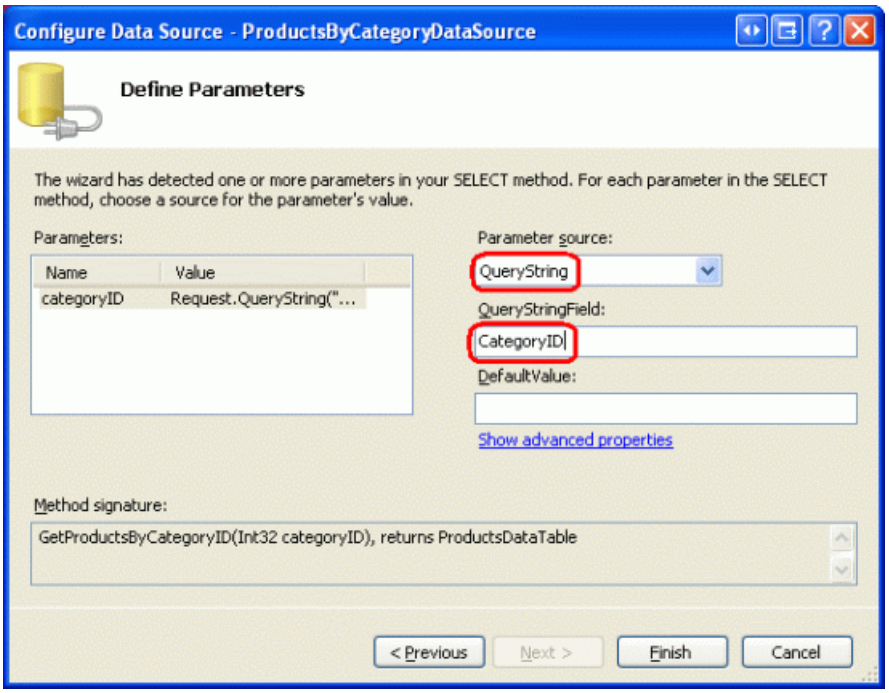


Figure 9: Use the `CategoryID` Querystring Field for the *categoryID* Parameter (Click to view full-size image)

After completing the wizard, Visual Studio will add corresponding BoundFields and a CheckBoxField to the GridView for the product data fields. Remove all but the `ProductName`, `UnitPrice`, and `SupplierName` BoundFields. Customize these three BoundFields `HeaderText` properties to read Product, Price, and Supplier, respectively. Format the `UnitPrice` BoundField as a currency.

Next, add a HyperLinkField and move it to the left-most position. Set its `Text` property to View Details, its `DataNavigateUrlFields` property to `ProductID`, and its `DataNavigateUrlFormatString` property to `~/SiteMapProvider/ProductDetails.aspx?ProductID={0}`.

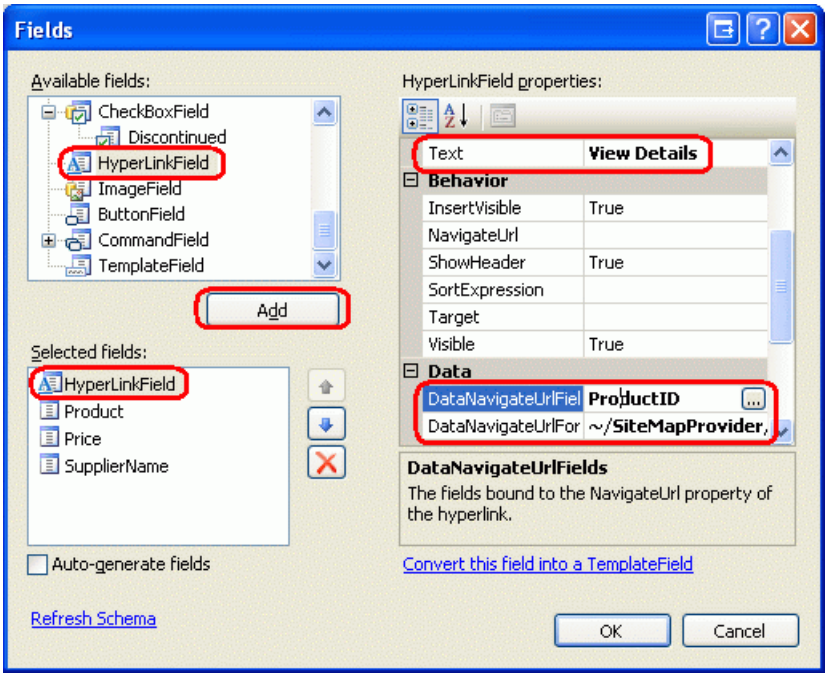


Figure 10: Add a View Details HyperLinkField that Points to `ProductDetails.aspx`

After making these customizations, the GridView and ObjectDataSource s declarative markup should resemble the following:

```
<asp:GridView ID="ProductsByCategory" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsByCategoryDataSource"
    EnableViewState="False">
    <Columns>
        <asp:HyperLinkField DataNavigateUrlFields="ProductID"
            DataNavigateUrlFormatString=
                "~/SiteMapProvider/ProductDetails.aspx?ProductID={0}"
            Text="View Details" />
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
            HeaderText="Price" HtmlEncode="False"
            SortExpression="UnitPrice" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
            ReadOnly="True" SortExpression="SupplierName" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsByCategoryDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProductsByCategoryID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:QueryStringParameter Name="categoryID"
            QueryStringField="CategoryID" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

Return to viewing `Default.aspx` through a browser and click on the View Products link for Beverages. This will

take you to `ProductsByCategory.aspx?CategoryID=1`, displaying the names, prices, and suppliers of the products in the Northwind database that belong to the Beverages category (see Figure 11). Feel free to further enhance this page to include a link to return users to the category listing page (`Default.aspx`) and a DetailsView or FormView control that displays the selected category s name and description.

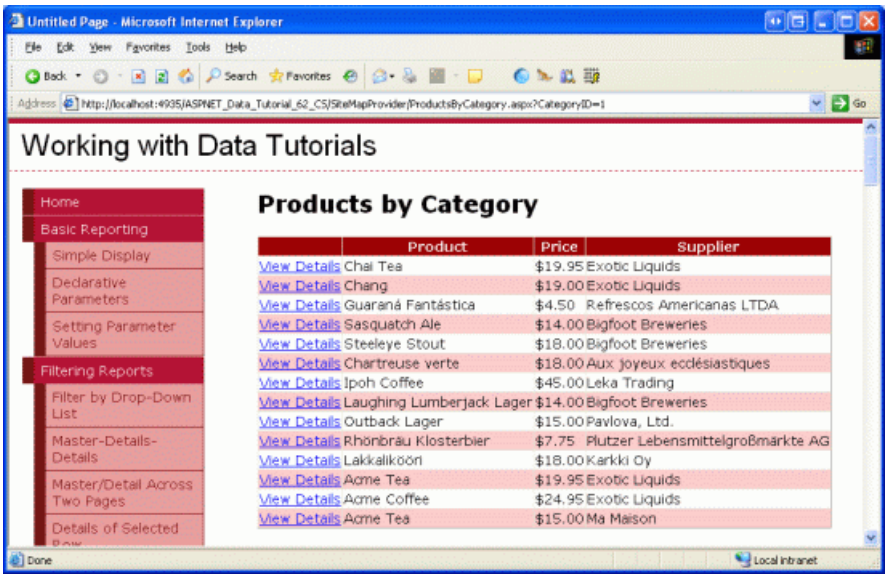


Figure 11: The Beverages Names, Prices, and Suppliers are Displayed (Click to view full-size image)

Step 4: Showing a Product s Details

The final page, `ProductDetails.aspx`, displays the selected products details. Open `ProductDetails.aspx` and drag a DetailsView from the Toolbox onto the Designer. Set the DetailsView s `ID` property to `ProductInfo` and clear out its `Height` and `Width` property values. From its smart tag, bind the DetailsView to a new ObjectDataSource named `ProductDataSource`, configuring the ObjectDataSource to pull its data from the `ProductsBLL` class s `GetProductByProductID(productID)` method. As with the previous web pages created in Steps 2 and 3, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) .

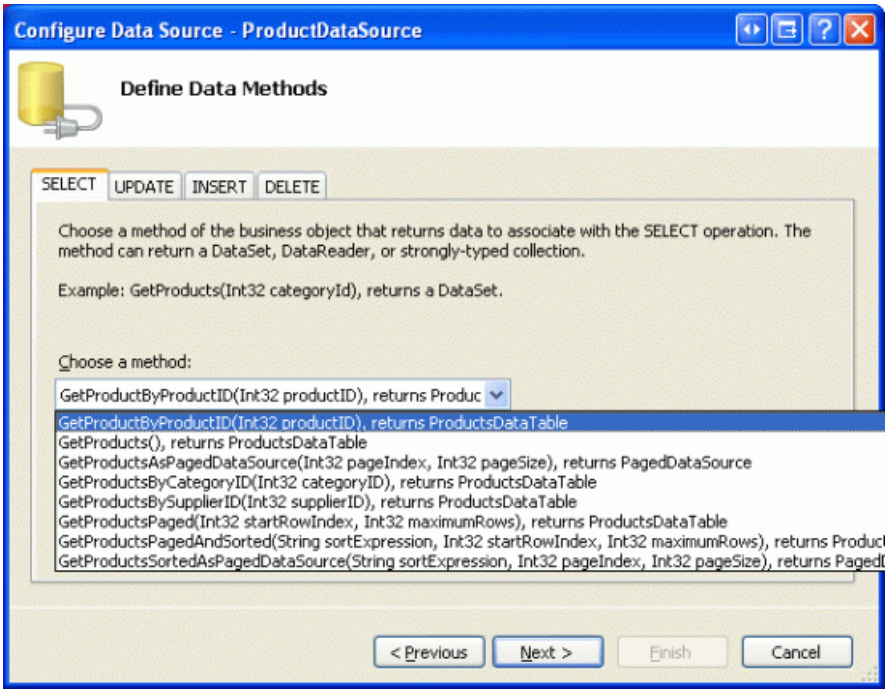


Figure 12: Configure the ObjectDataSource to Use the `GetProductByProductID(productID)` Method (Click to view full-size image)

The last step of the Configure Data Source wizard prompts for the source of the *productID* parameter. Since this

data comes through the querystring field `ProductID`, set the drop-down list to QueryString and the QueryStringField textbox to ProductID. Finally, click the Finish button to complete the wizard.

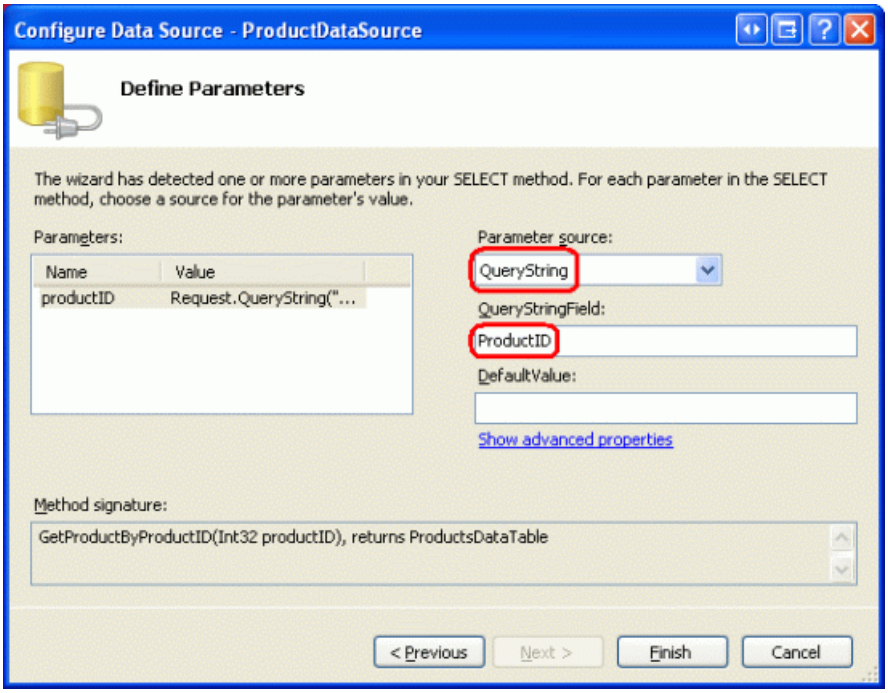


Figure 13: Configure the *productID* Parameter to Pull its Value from the `ProductID` Querystring Field (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will create corresponding BoundFields and a CheckBoxField in the DetailsView for the product data fields. Remove the `ProductID`, `SupplierID`, and `CategoryID` BoundFields and configure the remaining fields as you see fit. After a handful of aesthetic configurations, my DetailsView and ObjectDataSource s declarative markup looked like the following:

```
<asp:DetailsView ID="ProductInfo" runat="server" AutoGenerateRows="False"
    DataKeyNames="ProductID" DataSourceID="ProductDataSource"
    EnableViewState="False">
    <Fields>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
            ReadOnly="True" SortExpression="SupplierName" />
        <asp:BoundField DataField="QuantityPerUnit" HeaderText="Qty/Unit"
            SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
            HeaderText="Price" HtmlEncode="False"
            SortExpression="UnitPrice" />
        <asp:BoundField DataField="UnitsInStock" HeaderText="Units In Stock"
            SortExpression="UnitsInStock" />
        <asp:BoundField DataField="UnitsOnOrder" HeaderText="Units On Order"
            SortExpression="UnitsOnOrder" />
        <asp:BoundField DataField="ReorderLevel" HeaderText="Reorder Level"
            SortExpression="ReorderLevel" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Fields>
</asp:DetailsView>
<asp:ObjectDataSource ID="ProductDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProductByProductID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:QueryStringParameter Name="productID"
            QueryStringField="ProductID" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

To test this page, return to `Default.aspx` and click on View Products for the Beverages category. From the listing of beverage products, click on the View Details link for Chai Tea. This will take you to `ProductDetails.aspx?ProductID=1`, which shows a Chai Tea s details (see Figure 14).

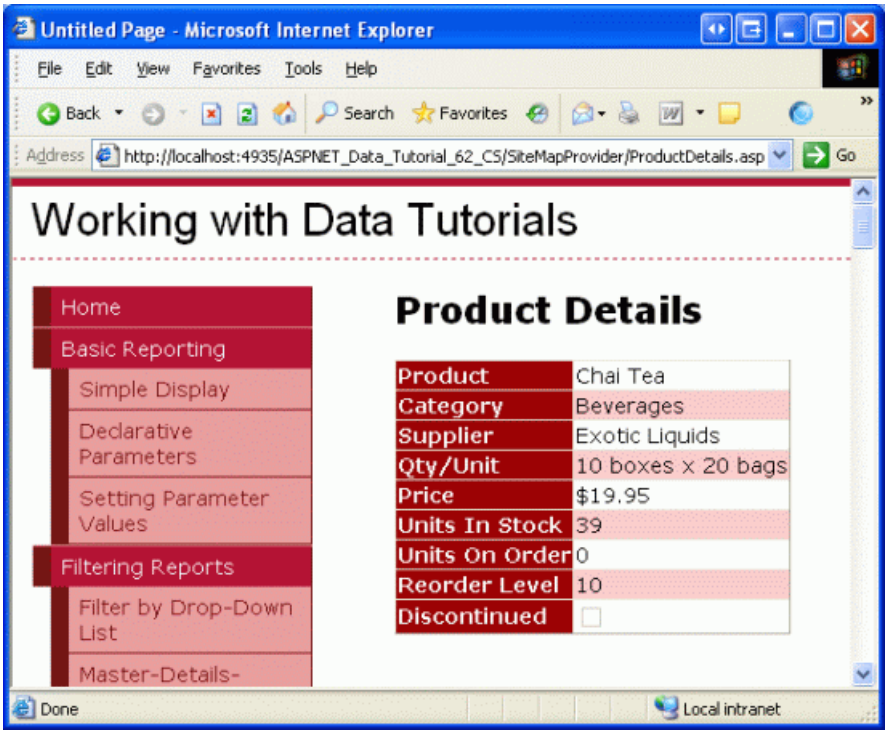


Figure 14: Chai Tea s Supplier, Category, Price, and Other Information is Displayed (Click to view full-size image)

Step 5: Understanding the Inner Workings of a Site Map Provider

The site map is represented in the web server s memory as a collection of `SiteMapNode` instances that form a hierarchy. There must be exactly one root, all non-root nodes must have exactly one parent node, and all nodes may have an arbitrary number of children. Each `SiteMapNode` object represents a section in the website s structure; these sections commonly have a corresponding web page. Consequently, the `SiteMapNode` class has properties like `Title`, `Url`, and `Description`, which provide information for the section the `SiteMapNode` represents. There is also a `Key` property that uniquely identifies each `SiteMapNode` in the hierarchy, as well as properties used to establish this hierarchy `ChildNodes`, `ParentNode`, `NextSibling`, `PreviousSibling`, and so forth.

Figure 15 shows the general site map structure from Figure 1, but with the implementation details sketched out in finer detail.

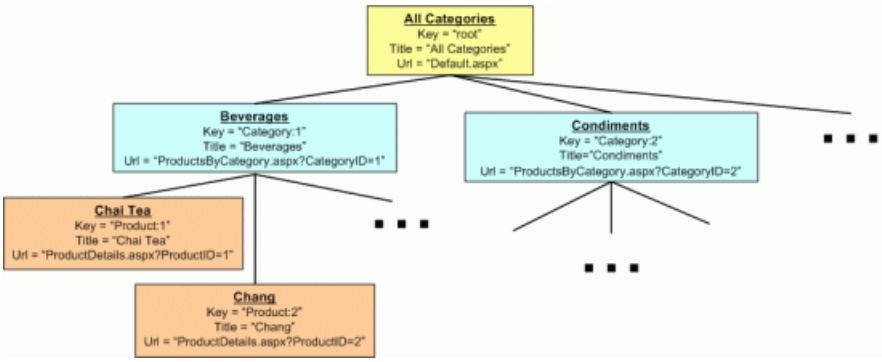


Figure 15: Each `SiteMapNode` has Properties Like `Title`, `Url`, `Key`, and So On (Click to view full-size image)

The site map is accessible through the `SiteMap` class in the `System.Web` namespace. This class s `RootNode` property returns the site map s root `SiteMapNode` instance; `CurrentNode` returns the `SiteMapNode` whose `Url` property matches the URL of the currently requested page. This class is used internally by ASP.NET 2.0 s navigation Web controls.

When the `SiteMap` class s properties are accessed, it must serialize the site map structure from some persistent medium into memory. However, the site map serialization logic is not hard coded into the `SiteMap` class. Instead, at runtime the `SiteMap` class determines which site map *provider* to use for serialization. By default, the `XmlSiteMapProvider` class is used, which reads the site map s structure from a properly-formatted XML file. However, with a little bit of work we can create our own custom site map provider.

All site map providers must be derived from the `SiteMapProvider` class, which includes the essential methods and properties needed for site map providers, but omits many of the implementation details. A second class, `StaticSiteMapProvider`, extends the `SiteMapProvider` class and contains a more robust implementation of the needed functionality. Internally, the `StaticSiteMapProvider` stores the `SiteMapNode` instances of the site map in a `Hashtable` and provides methods like `AddNode(child, parent)`, `RemoveNode(siteMapNode)`, and `Clear()` that add and remove `SiteMapNode` s to the internal `Hashtable`. `XmlSiteMapProvider` is derived from `StaticSiteMapProvider`.

When creating a custom site map provider that extends `StaticSiteMapProvider`, there are two abstract methods that must be overridden: `BuildSiteMap` and `GetRootNodeCore`. `BuildSiteMap`, as its name implies, is responsible for loading the site map structure from persistent storage and constructing it in memory. `GetRootNodeCore` returns the root node in the site map.

Before a web application can use a site map provider it must be registered in the application s configuration. By default, the `XmlSiteMapProvider` class is registered using the name `AspNetXmlSiteMapProvider`. To register additional site map providers, add the following markup to `Web.config`:

```
<configuration>
    <system.web>
        ...
        <siteMap defaultProvider="defaultProviderName">
          <providers>
            <add name="name" type="type" />
          </providers>
        </siteMap>
    </system.web>
</configuration>
```

The *name* value assigns a human-readable name to the provider while *type* specifies the fully-qualified type name of the site map provider. We'll explore concrete values for the *name* and *type* values in Step 7, after we ve created our custom site map provider.

The site map provider class is instantiated the first time it is accessed from the `SiteMap` class and remains in memory for the lifetime of the web application. Since there is only one instance of the site map provider that may be invoked from multiple, concurrent web site visitors, it is imperative that the provider s methods be *thread-safe*.

For performance and scalability reasons, it s important that we cache the in-memory site map structure and return this cached structure rather than recreating it every time the `BuildSiteMap` method is invoked. `BuildSiteMap` may be called several times per page request per user, depending on the navigation controls in use on the page and the depth of the site map structure. In any case, if we do not cache the site map structure in `BuildSiteMap` then each time it is invoked we would need to re-retrieve the product and category information from the architecture (which would result in a query to the database). As we discussed in the previous caching tutorials, cached data can become stale. To combat this, we can use either time- or SQL cache dependency-based expiries.

NOTE

A site map provider may optionally override the `Initialize` method. `Initialize` is invoked when the site map provider is first instantiated and is passed any custom attributes assigned to the provider in `Web.config` in the `<add>` element like: `<add name="name" type="type" customAttribute="value" />`. It is useful if you want to allow a page developer to specify various site map provider-related settings without having to modify the provider s code. For example, if we were reading the category and products data directly from the database as opposed to through the architecture, we d likely want to let the page developer specify the database connection string through `Web.config` rather than using a hard coded value in the provider s code. The custom site map provider we'll build in Step 6 does not override this `Initialize` method. For an example of using the `Initialize` method, refer to Jeff Prosise s Storing Site Maps in SQL Server article.

Step 6: Creating the Custom Site Map Provider

To create a custom site map provider that builds the site map from the categories and products in the Northwind database, we need to create a class that extends `StaticSiteMapProvider`. In Step 1 I asked you to add a `CustomProviders` folder in the `App_Code` folder - add a new class to this folder named `NorthwindSiteMapProvider`. Add the following code to the `NorthwindSiteMapProvider` class:

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Web.Caching;
```

```
public class NorthwindSiteMapProvider : StaticSiteMapProvider
{
    private readonly object siteMapLock = new object();
    private SiteMapNode root = null;
    public const string CacheDependencyKey =
        "NorthwindSiteMapProviderCacheDependency";
    public override SiteMapNode BuildSiteMap()
    {
        // Use a lock to make this method thread-safe
        lock (siteMapLock)
        {
            // First, see if we already have constructed the
            // rootNode. If so, return it...
            if (root != null)
                return root;
            // We need to build the site map!

            // Clear out the current site map structure
            base.Clear();
            // Get the categories and products information from the database
            ProductsBLL productsAPI = new ProductsBLL();
            Northwind.ProductsDataTable products = productsAPI.GetProducts();
            // Create the root SiteMapNode
            root = new SiteMapNode(
                this, "root", "~/SiteMapProvider/Default.aspx", "All Categories");
            AddNode(root);
            // Create SiteMapNodes for the categories and products
            foreach (Northwind.ProductsRow product in products)
            {
                // Add a new category SiteMapNode, if needed
                string categoryKey, categoryName;
                bool createUrlForCategoryNode = true;
                if (product.IsCategoryIDNull())
                {
                    categoryKey = "Category:None";
                    categoryName = "None";
                    createUrlForCategoryNode = false;
                }
                else
                {
                    categoryKey = string.Concat("Category:", product.CategoryID);
                    categoryName = product.CategoryName;
                }
                SiteMapNode categoryNode = FindSiteMapNodeFromKey(categoryKey);
                // Add the category SiteMapNode if it does not exist
                if (categoryNode == null)
                {
                    string productsByCategoryUrl = string.Empty;
                    if (createUrlForCategoryNode)
                        productsByCategoryUrl =
                            "~/SiteMapProvider/ProductsByCategory.aspx?CategoryID="
                            + product.CategoryID;
                    categoryNode = new SiteMapNode(
                        this, categoryKey, productsByCategoryUrl, categoryName);
                    AddNode(categoryNode, root);
                }
                // Add the product SiteMapNode
                string productUrl =
                    "~/SiteMapProvider/ProductDetails.aspx?ProductID="
                    + product.ProductID;
                SiteMapNode productNode = new SiteMapNode(
                    this, string.Concat("Product:", product.ProductID),
                    productUrl, product.ProductName);
                AddNode(productNode, categoryNode);
            }

            // Add a "dummy" item to the cache using a SqlCacheDependency
            // on the Products and Categories tables
            System.Web.Caching.SqlCacheDependency productsTableDependency =
```

```
                new System.Web.Caching.SqlCacheDependency("NorthwindDB", "Products");
            System.Web.Caching.SqlCacheDependency categoriesTableDependency =
                new System.Web.Caching.SqlCacheDependency("NorthwindDB", "Categories");
            // Create an AggregateCacheDependency
            System.Web.Caching.AggregateCacheDependency aggregateDependencies =
                new System.Web.Caching.AggregateCacheDependency();
            aggregateDependencies.Add(productsTableDependency, categoriesTableDependency);
            // Add the item to the cache specifying a callback function
            HttpRuntime.Cache.Insert(
                CacheDependencyKey, DateTime.Now, aggregateDependencies,
                Cache.NoAbsoluteExpiration, Cache.NoSlidingExpiration,
                CacheItemPriority.Normal,
                new CacheItemRemovedCallback(OnSiteMapChanged));
            // Finally, return the root node
            return root;
        }
    }
    protected override SiteMapNode GetRootNodeCore()
    {
        return BuildSiteMap();
    }
    protected void OnSiteMapChanged(string key, object value, CacheItemRemovedReason reason)
    {
        lock (siteMapLock)
        {
            if (string.Compare(key, CacheDependencyKey) == 0)
            {
                // Refresh the site map
                root = null;
            }
        }
    }
    public DateTime? CachedDate
    {
        get
        {
            return HttpRuntime.Cache[CacheDependencyKey] as DateTime?;
        }
    }
}
```

Let s start with exploring this class s `BuildSiteMap` method, which starts with a `lock` statement. The `lock` statement only allows one thread at a time to enter, thereby serializing access to its code and preventing two concurrent threads from stepping on one another s toes.

The class-level `SiteMapNode` variable `root` is used to cache the site map structure. When the site map is constructed for the first time, or for the first time after the underlying data has been modified, `root` will be `null` and the site map structure will be constructed. The site map s root node is assigned to `root` during the construction process so that the next time this method is called, `root` will not be `null`. Consequently, so long as `root` is not `null` the site map structure will be returned to the caller without having to recreate it.

If root is `null`, the site map structure is created from the product and category information. The site map is built by creating the `SiteMapNode` instances and then forming the hierarchy through calls to the `StaticSiteMapProvider` class s `AddNode` method. `AddNode` performs the internal bookkeeping, storing the assorted `SiteMapNode` instances in a `Hashtable`. Before we start constructing the hierarchy, we start by calling the `Clear` method, which clears out the elements from the internal `Hashtable`. Next, the `ProductsBLL` class s `GetProducts` method and the resulting `ProductsDataTable` are stored in local variables.

The site map s construction begins by creating the root node and assigning it to `root`. The overload of the `SiteMapNode` s constructor used here and throughout this `BuildSiteMap` is passed the following information:

- A reference to the site map provider (`this`).

- The `SiteMapNode` s `Key`. This required value must be unique for each `SiteMapNode`.
- The `SiteMapNode` s `Url`. `Url` is optional, but if provided, each `SiteMapNode` s `Url` value must be unique.
- The `SiteMapNode` s `Title`, which is required.

The `AddNode(root)` method call adds the `SiteMapNode` `root` to the site map as the root. Next, each `ProductRow` in the `ProductsDataTable` is enumerated. If there already exists a `SiteMapNode` for the current product s category, it is referenced. Otherwise, a new `SiteMapNode` for the category is created and added as a child of the `SiteMapNode``root` through the `AddNode(categoryNode, root)` method call. After the appropriate category `SiteMapNode` node has been found or created, a `SiteMapNode` is created for the current product and added as a child of the category `SiteMapNode` via `AddNode(productNode, categoryNode)`. Note that the category `SiteMapNode` s `Url` property value is `~/SiteMapProvider/ProductsByCategory.aspx?CategoryID=categoryID` while the product `SiteMapNode` s `Url` property is assigned `~/SiteMapNode/ProductDetails.aspx?ProductID=productID`.

> **NOTE**
> Those products that have a database `NULL` value for their `CategoryID` are grouped under a category `SiteMapNode` whose `Title` property is set to None and whose `Url` property is set to an empty string. I decided to set `Url` to an empty string since the `ProductBLL` class s `GetProductsByCategory(categoryID)` method currently lacks the capability to return just those products with a `NULL` `CategoryID` value. Also, I wanted to demonstrate how the navigation controls render a `SiteMapNode` that lacks a value for its `Url` property. I encourage you to extend this tutorial so that the None `SiteMapNode` s `Url` property points to `ProductsByCategory.aspx`, yet only displays the products with `NULL` `CategoryID` values.

After constructing the site map, an arbitrary object is added to the data cache using a SQL cache dependency on the `Categories` and `Products` tables through an `AggregateCacheDependency` object. We explored using SQL cache dependencies in the preceding tutorial, *Using SQL Cache Dependencies*. The custom site map provider, however, uses an overload of the data cache s `Insert` method that we ve yet to explore. This overload accepts as its final input parameter a delegate that is called when the object is removed from the cache. Specifically, we pass in a new `CacheItemRemovedCallback` delegate that points to the `OnSiteMapChanged` method defined further down in the `NorthwindSiteMapProvider` class.

> **NOTE**
> The in-memory representation of the site map is cached through the class-level variable `root`. Since there is only one instance of the custom site map provider class and since that instance is shared among all threads in the web application, this class variable serves as a cache. The `BuildSiteMap` method also uses the data cache, but only as a means to receive notification when the underlying database data in the `Categories` or `Products` tables changes. Note that the value put into the data cache is just the current date and time. The actual site map data is *not* put in the data cache.

The `BuildSiteMap` method completes by returning the root node of the site map.

The remaining methods are fairly straightforward. `GetRootNodeCore` is responsible for returning the root node. Since `BuildSiteMap` returns the root, `GetRootNodeCore` simply returns `BuildSiteMap` s return value. The `OnSiteMapChanged` method sets `root` back to `null` when the cache item is removed. With root set back to `null`, the next time `BuildSiteMap` is invoked, the site map structure will be rebuilt. Lastly, the `CachedDate` property returns the date and time value stored in the data cache, if such a value exists. This property can be used by a page developer to determine when the site map data was last cached.

Step 7: Registering the `NorthwindSiteMapProvider`

In order for our web application to use the `NorthwindSiteMapProvider` site map provider created in Step 6, we need to register it in the `<siteMap>` section of `Web.config`. Specifically, add the following markup within the

`<system.web>` element in `Web.config`:

```
<siteMap defaultProvider="AspNetXmlSiteMapProvider">
  <providers>
    <add name="Northwind" type="NorthwindSiteMapProvider" />
  </providers>
</siteMap>
```

This markup does two things: first, it indicates that the built-in `AspNetXmlSiteMapProvider` is the default site map provider; second, it registers the custom site map provider created in Step 6 with the human-friendly name Northwind .

> **NOTE**
> For site map providers located in the application s `App_Code` folder, the value of the `type` attribute is simply the class name. Alternatively, the custom site map provider could have been created in a separate Class Library project with the compiled assembly placed in the web application s `/Bin` directory. In that case, the `type` attribute value would be *Namespace.ClassName, AssemblyName* .

After updating `Web.config`, take a moment to view any page from the tutorials in a browser. Note that the navigation interface on the left still shows the sections and tutorials defined in `Web.sitemap`. This is because we left `AspNetXmlSiteMapProvider` as the default provider. In order to create a navigation user interface element that uses the `NorthwindSiteMapProvider`, we'll need to explicitly specify that the Northwind site map provider should be used. We'll see how to accomplish this in Step 8.

Step 8: Displaying Site Map Information Using the Custom Site Map Provider

With the custom site map provider created and registered in `Web.config`, we re ready to add navigation controls to the `Default.aspx`, `ProductsByCategory.aspx`, and `ProductDetails.aspx` pages in the `SiteMapProvider` folder. Start by opening the `Default.aspx` page and drag a `SiteMapPath` from the Toolbox onto the Designer. The SiteMapPath control is located in the Navigation section of the Toolbox.

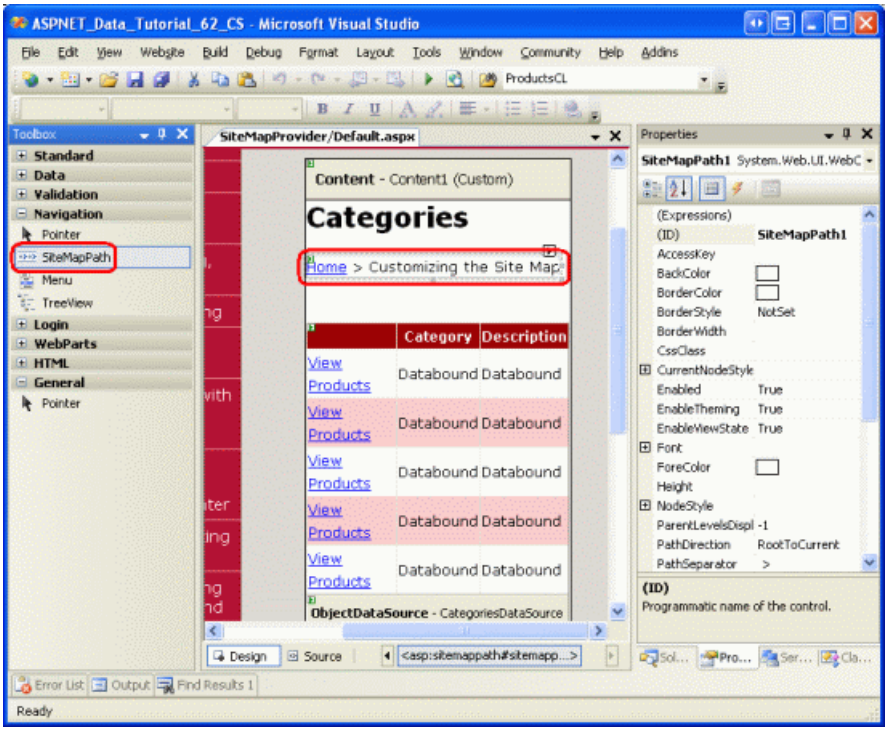


Figure 16: Add a SiteMapPath to `Default.aspx` (Click to view full-size image)

The SiteMapPath control displays a breadcrumb, indicating the current page s location within the site map. We added a SiteMapPath to the top of the master page back in the *Master Pages and Site Navigation* tutorial.

Take a moment to view this page through a browser. The SiteMapPath added in Figure 16 uses the default site map provider, pulling its data from `Web.sitemap`. Therefore, the breadcrumb shows Home > Customizing the Site Map, just like the breadcrumb in the upper-right corner.

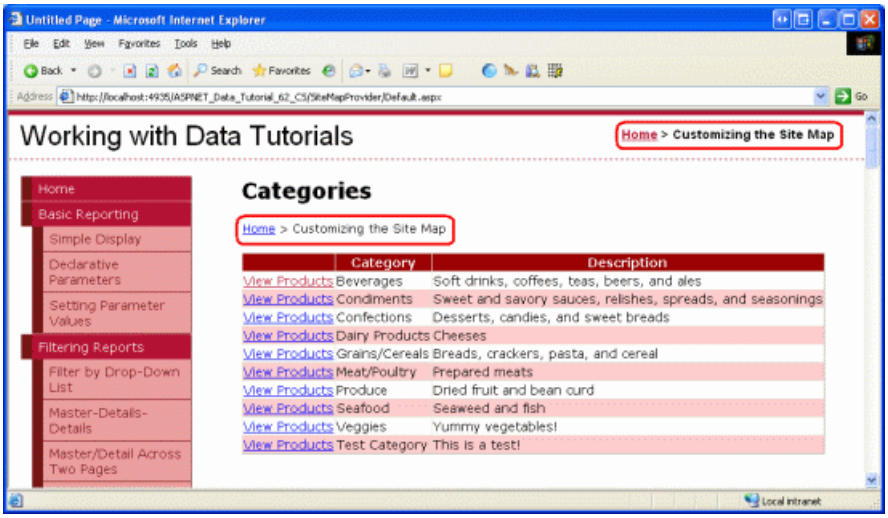


Figure 17: The Breadcrumb Uses the Default Site Map Provider (Click to view full-size image)

To have the SiteMapPath added in Figure 16 use the custom site map provider we created in Step 6, set its `SiteMapProvider` property to Northwind, the name we assigned to the `NorthwindSiteMapProvider` in `Web.config`. Unfortunately, the Designer continues to use the default site map provider, but if you visit the page through a browser after making this property change you'll see that the breadcrumb now uses the custom site map provider.

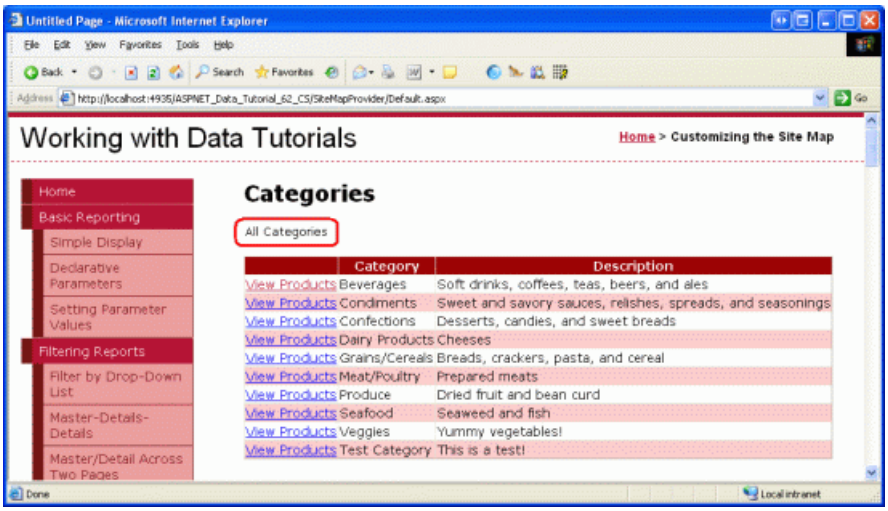


Figure 18: The Breadcrumb Now Uses the Custom Site Map Provider `NorthwindSiteMapProvider` (Click to view full-size image)

The SiteMapPath control displays a more functional user interface in the `ProductsByCategory.aspx` and `ProductDetails.aspx` pages. Add a SiteMapPath to these pages, setting the `SiteMapProvider` property in both to Northwind. From `Default.aspx` click on the View Products link for Beverages, and then on the View Details link for Chai Tea. As Figure 19 shows, the breadcrumb includes the current site map section (Chai Tea) and its ancestors: Beverages and All Categories .

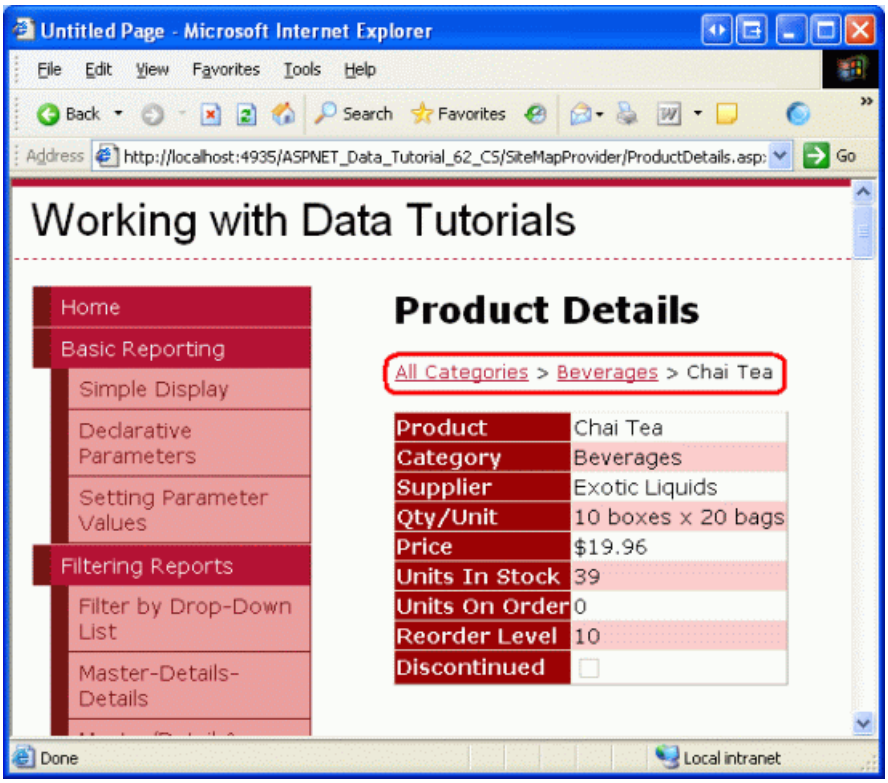


Figure 19: The Breadcrumb Now Uses the Custom Site Map Provider `NorthwindSiteMapProvider` (Click to view full-size image)

Other navigation user interface elements can be used in addition to the SiteMapPath, such as the Menu and TreeView controls. The `Default.aspx`, `ProductsByCategory.aspx`, and `ProductDetails.aspx` pages in the download for this tutorial, for example, all include Menu controls (see Figure 20). See Examining ASP.NET 2.0 s Site Navigation Features and the Using Site Navigation Controls section of the ASP.NET 2.0 QuickStarts for a more in-depth look at the navigation controls and site map system in ASP.NET 2.0.

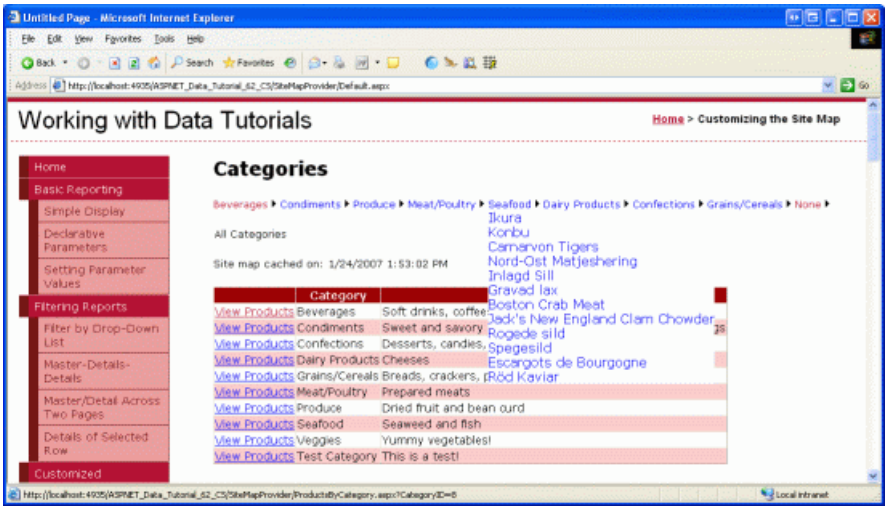


Figure 20: The Menu Control Lists Each of the Categories and Products (Click to view full-size image)

As mentioned earlier in this tutorial, the site map structure can be accessed programmatically through the `SiteMap` class. The following code returns the root `SiteMapNode` of the default provider:

```
SiteMapNode root = SiteMap.RootNode;
```

Since the `AspNetXmlSiteMapProvider` is the default provider for our application, the above code would return the root node defined in `Web.sitemap`. To reference a site map provider other than the default, use the `SiteMap` class s `Providers` property like so:

```
SiteMapNode root = SiteMap.Providers["name"].RootNode;
```

Where *name* is the name of the custom site map provider (Northwind, for our web application).

To access a member specific to a site map provider, use `SiteMap.Providers["name"]` to retrieve the provider instance and then cast it to the appropriate type. For example, to display the `NorthwindSiteMapProvider` s `CachedDate` property in an ASP.NET page, use the following code:

```
NorthwindSiteMapProvider customProvider =
    SiteMap.Providers["Northwind"] as NorthwindSiteMapProvider;
if (customProvider != null)
{
    DateTime? lastCachedDate = customProvider.CachedDate;
    if (lastCachedDate != null)
        LabelID.Text = "Site map cached on: " + lastCachedDate.Value.ToString();
    else
        LabelID.Text = "The site map is being reconstructed!";
}
```

NOTE

Be sure to test out the SQL cache dependency feature. After visiting the `Default.aspx` , `ProductsByCategory.aspx` , and `ProductDetails.aspx` pages, go to one of the tutorials in the Editing, Inserting, and Deleting section and edit the name of a category or product. Then return to one of the pages in the `SiteMapProvider` folder. Assuming enough time has passed for the polling mechanism to note the change to the underlying database, the site map should be updated to show the new product or category name.

Summary

ASP.NET 2.0 s site map features includes a `SiteMap` class, a number of built-in navigation Web controls, and a default site map provider that expects the site map information persisted to an XML file. In order to use site map information from some other source such as from a database, the application s architecture, or a remote Web service we need to create a custom site map provider. This involves creating a class that derives, directly or indirectly, from the `SiteMapProvider` class.

In this tutorial we saw how to create a custom site map provider that based the site map on the product and category information culled from the application architecture. Our provider extended the `StaticSiteMapProvider` class and entailed creating a `BuildSiteMap` method that retrieved the data, constructed the site map hierarchy, and cached the resulting structure in a class-level variable. We used a SQL cache dependency with a callback function to invalidate the cached structure when the underlying `Categories` or `Products` data is modified.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Storing Site Maps in SQL Server and The SQL Site Map Provider You ve Been Waiting For
- A Look at ASP.NET 2.0 s Provider Model
- The Provider Toolkit
- Examining ASP.NET 2.0 s Site Navigation Features

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Dave Gardner, Zack Jones, Teresa Murphy, and Bernadette Leigh. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Building a Custom Database-Driven Site Map Provider (VB)

2/20/2020 • 27 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> The default site map provider in ASP.NET 2.0 retrieves its data from a static XML file. While the XML-based provider is suitable to many small and medium-sized Web sites, larger Web applications require a more dynamic site map. In this tutorial we'll build a custom site map provider that retrieves its data from the Business Logic Layer, which in turn retrieves data from the database.

Introduction

ASP.NET 2.0 s site map feature enables a page developer to define a web application s site map in some persistent medium, such as in an XML file. Once defined, the site map data can be accessed programmatically through the `SiteMap` class in the `System.Web` namespace or through a variety of navigation Web controls, such as the SiteMapPath, Menu, and TreeView controls. The site map system uses the provider model so that different site map serialization implementations can be created and plugged into a web application. The default site map provider that ships with ASP.NET 2.0 persists site map structure in an XML file. Back in the Master Pages and Site Navigation tutorial we created a file named `Web.sitemap` that contained this structure and have been updating its XML with each new tutorial section.

The default XML-based site map provider works well if the site map s structure is fairly static, such as for these tutorials. In many scenarios, however, a more dynamic site map is needed. Consider the site map shown in Figure 1, where each category and product appear as sections in the website s structure. With this site map, visiting the web page corresponding to the root node might list all of the categories, whereas visiting a particular category s web page would list that category s products and viewing a particular product s web page would show that product s details.

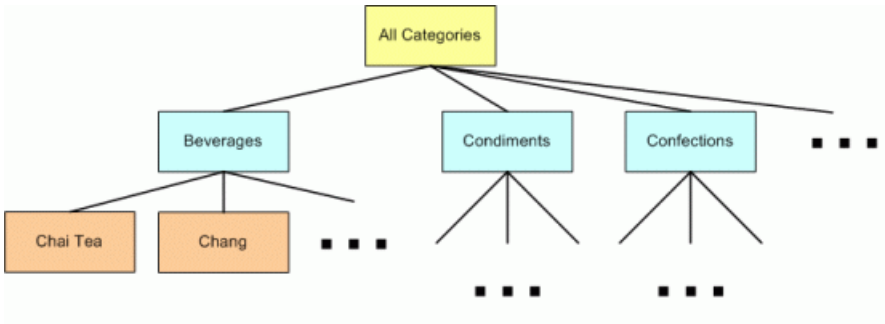


Figure 1: The Categories and Products Makeup the Site Map s Structure (Click to view full-size image)

While this category- and product-based structure could be hard-coded into the `Web.sitemap` file, the file would need to be updated each time a category or product was added, removed, or renamed. Consequently, the site map maintenance would be greatly simplified if its structure was retrieved from the database or, ideally, from the Business Logic Layer of the application s architecture. That way, as products and categories were added, renamed, or deleted, the site map would automatically update to reflect these changes.

Since ASP.NET 2.0 s site map serialization is built atop the provider model, we can create our own custom site map provider that grabs its data from an alternate data store, such as the database or architecture. In this tutorial we'll

build a custom provider that retrieves its data from the BLL. Let s get started!

> **NOTE**
> The custom site map provider created in this tutorial is tightly coupled to the application s architecture and data model. Jeff Prosise s Storing Site Maps in SQL Server and The SQL Site Map Provider You ve Been Waiting For articles examine a generalized approach to storing site map data in SQL Server.

Step 1: Creating the Custom Site Map Provider Web Pages

Before we start creating a custom site map provider, let s first add the ASP.NET pages we'll need for this tutorial. Start by adding a new folder named `SiteMapProvider`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `ProductsByCategory.aspx`
- `ProductDetails.aspx`

Also add a `CustomProviders` subfolder to the `App_Code` folder.

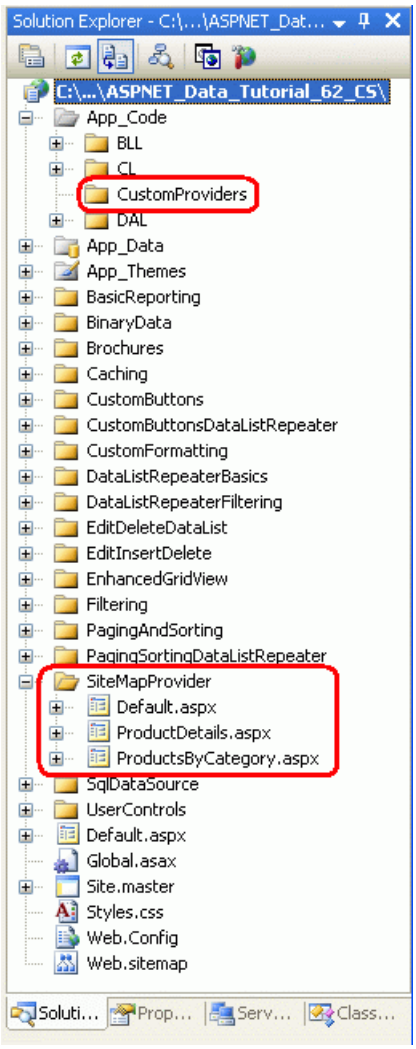


Figure 2: Add the ASP.NET Pages for the Site Map Provider-Related Tutorials

Since there is only one tutorial for this section, we don t need `Default.aspx` to list the section s tutorials. Instead, `Default.aspx` will display the categories in a GridView control. We'll tackle this in Step 2.

Next, update `Web.sitemap` to include a reference to the `Default.aspx` page. Specifically, add the following markup after the Caching `<siteMapNode>`:

```
<siteMapNode
    title="Customizing the Site Map" url="~/SiteMapProvider/Default.aspx"
    description="Learn how to create a custom provider that retrieves the site map
                 from the Northwind database." />
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes an item for the sole site map provider tutorial.

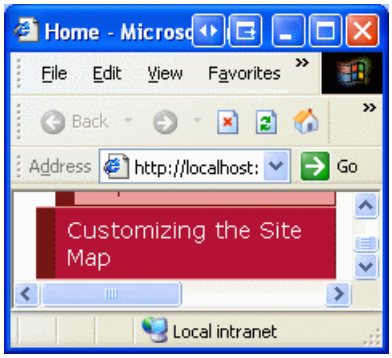


Figure 3: The Site Map Now Includes an Entry for the Site Map Provider Tutorial

This tutorial s main focus is to illustrate creating a custom site map provider and configuring a web application to use that provider. In particular, we'll build a provider that returns a site map that includes a root node along with a node for each category and product, as depicted in Figure 1. In general, each node in the site map may specify a URL. For our site map, the root node s URL will be `~/SiteMapProvider/Default.aspx`, which will list all of the categories in the database. Each category node in the site map will have a URL that points to `~/SiteMapProvider/ProductsByCategory.aspx?CategoryID=categoryID`, which will list all of the products in the specified *categoryID*. Finally, each product site map node will point to `~/SiteMapProvider/ProductDetails.aspx?ProductID=productID`, which will display the specific product s details.

To start we need to create the `Default.aspx`, `ProductsByCategory.aspx`, and `ProductDetails.aspx` pages. These pages are completed in Steps 2, 3, and 4, respectively. Since the thrust of this tutorial is on site map providers, and since past tutorials have covered creating these sorts of multi-page master/detail reports, we will hurry through Steps 2 through 4. If you need a refresher on creating master/detail reports that span multiple pages, refer back to the Master/Detail Filtering Across Two Pages tutorial.

Step 2: Displaying a List of Categories

Open the `Default.aspx` page in the `SiteMapProvider` folder and drag a GridView from the Toolbox onto the Designer, setting its `ID` to `Categories`. From the GridView s smart tag, bind it to a new ObjectDataSource named `CategoriesDataSource` and configure it so that it retrieves its data using the `CategoriesBLL` class s `GetCategories` method. Since this GridView just displays the categories and does not provide data modification capabilities, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) .

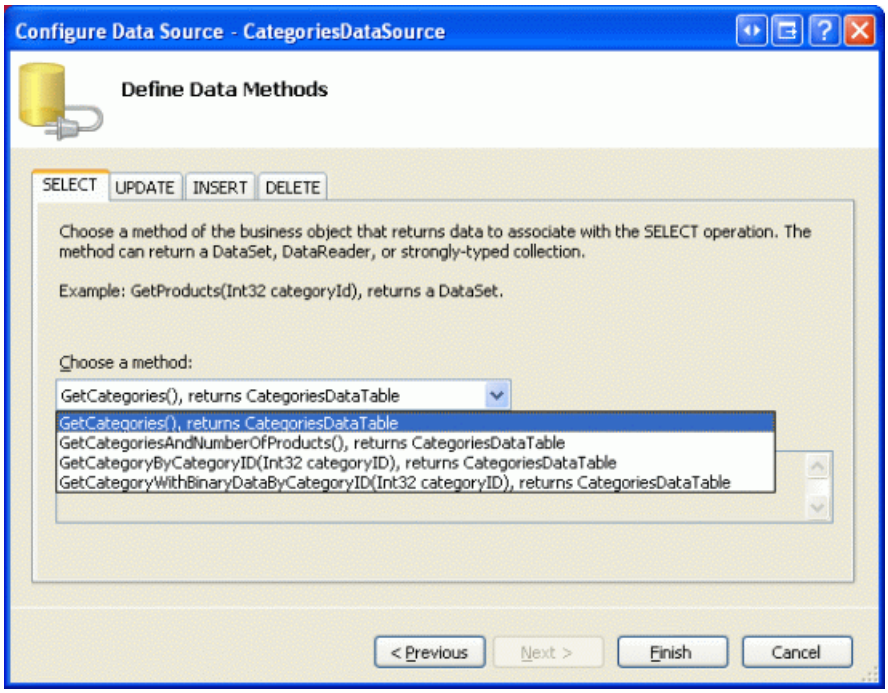


Figure 4: Configure the ObjectDataSource to Return Categories Using the `GetCategories` Method (Click to view full-size image)

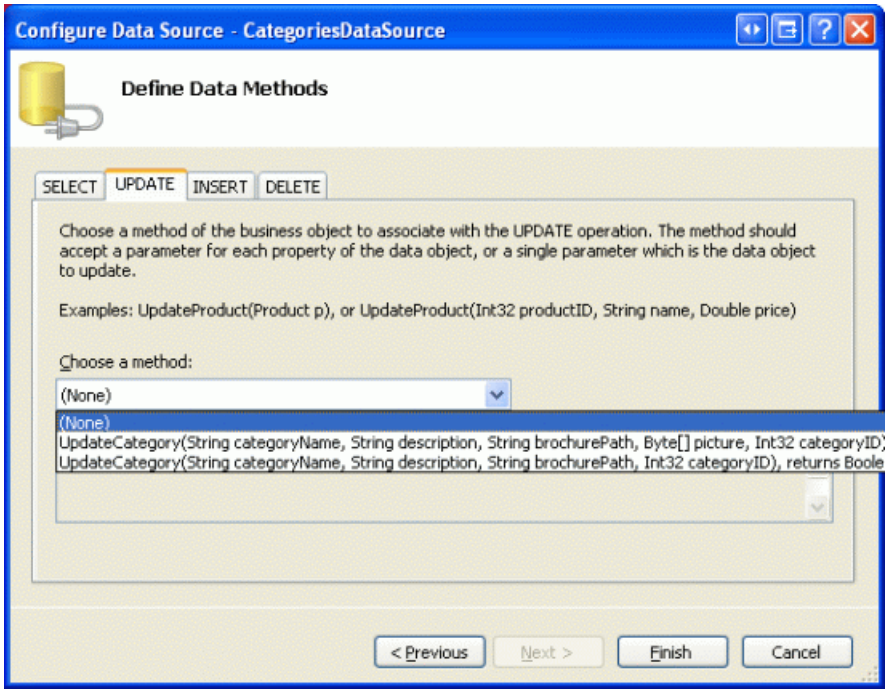


Figure 5: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE Tabs to (None) (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will add a BoundField for `CategoryID`, `CategoryName`, `Description`, `NumberOfProducts`, and `BrochurePath`. Edit the GridView so that it only contains the `CategoryName` and `Description` BoundFields and update the `CategoryName` BoundField s `HeaderText` property to Category .

Next, add a HyperLinkField and position it so that it s the left-most field. Set the `DataNavigateUrlFields` property to `CategoryID` and the `DataNavigateUrlFormatString` property to `~/SiteMapProvider/ProductsByCategory.aspx?CategoryID={0}`. Set the `Text` property to View Products .

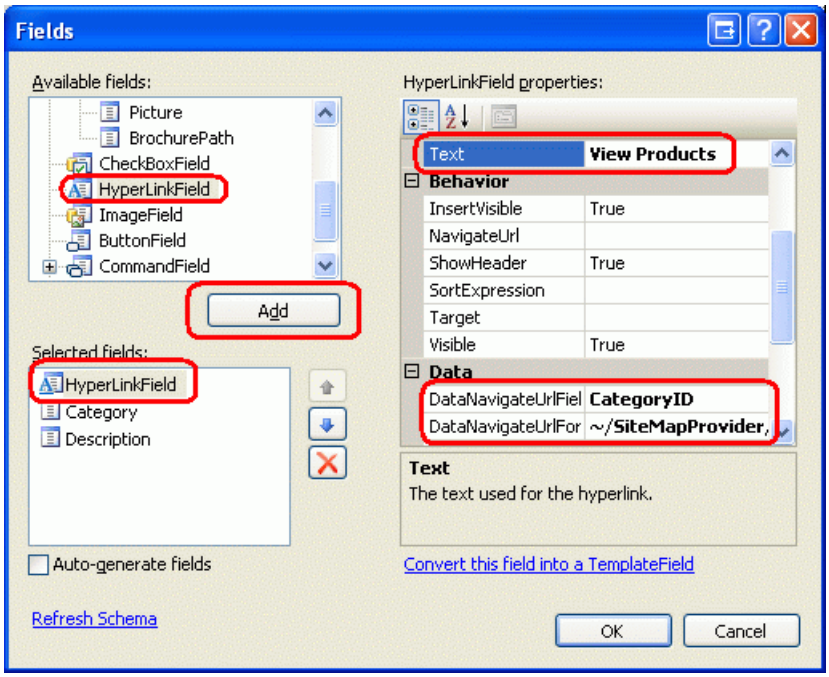


Figure 6: Add a HyperLinkField to the `Categories` GridView

After creating the ObjectDataSource and customizing the GridView s fields, the two controls declarative markup will look like the following:

```
<asp:GridView ID="Categories" runat="server" AutoGenerateColumns="False"
    DataKeyNames="CategoryID" DataSourceID="CategoriesDataSource"
    EnableViewState="False">
    <Columns>
        <asp:HyperLinkField DataNavigateUrlFields="CategoryID"
            DataNavigateUrlFormatString=
                "~/SiteMapProvider/ProductsByCategory.aspx?CategoryID={0}"
            Text="View Products" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            SortExpression="CategoryName" />
        <asp:BoundField DataField="Description" HeaderText="Description"
            SortExpression="Description" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}" SelectMethod="GetCategories"
    TypeName="CategoriesBLL"></asp:ObjectDataSource>
```

Figure 7 shows `Default.aspx` when viewed through a browser. Clicking a category s View Products link takes you to `ProductsByCategory.aspx?CategoryID=categoryID`, which we will build in Step 3.

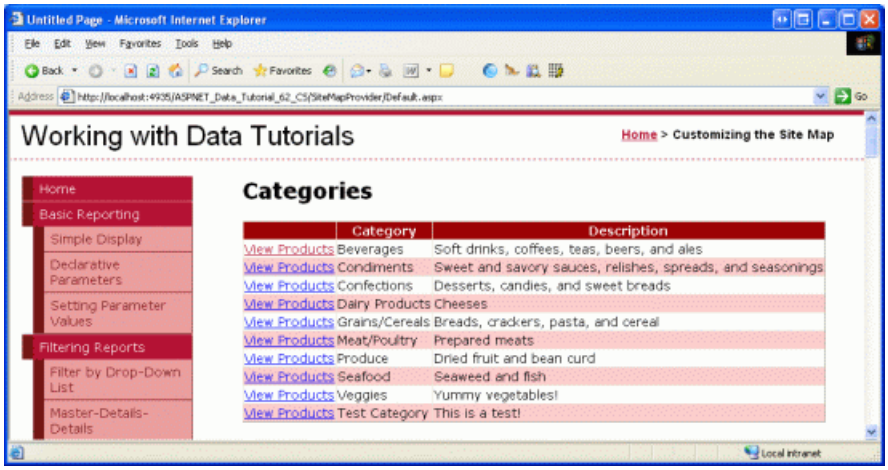


Figure 7: Each Category is Listed Along with a View Products Link (Click to view full-size image)

Step 3: Listing the Selected Category s Products

Open the `ProductsByCategory.aspx` page and add a GridView, naming it `ProductsByCategory`. From its smart tag, bind the GridView to a new ObjectDataSource named `ProductsByCategoryDataSource`. Configure the ObjectDataSource to use the `ProductsBLL` class s `GetProductsByCategoryID(categoryID)` method and set the drop-down lists to (None) in the UPDATE, INSERT, and DELETE tabs.

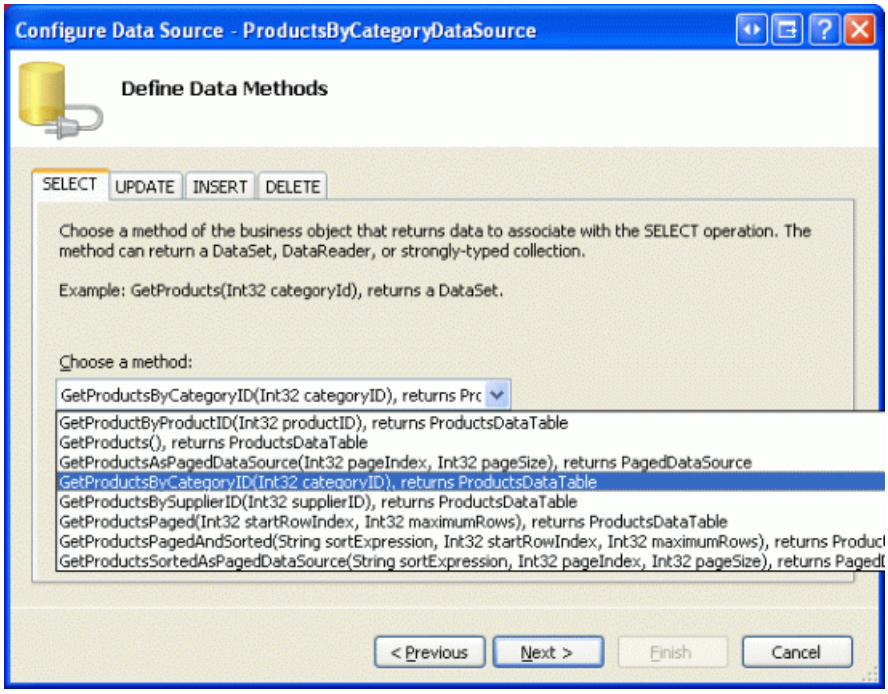


Figure 8: Use the `ProductsBLL` Class s `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

The final step in the Configure Data Source wizard prompts for a parameter source for *categoryID*. Since this information is passed through the querystring field `CategoryID`, select QueryString from the drop-down list and enter CategoryID in the QueryStringField textbox as shown in Figure 9. Click Finish to complete the wizard.

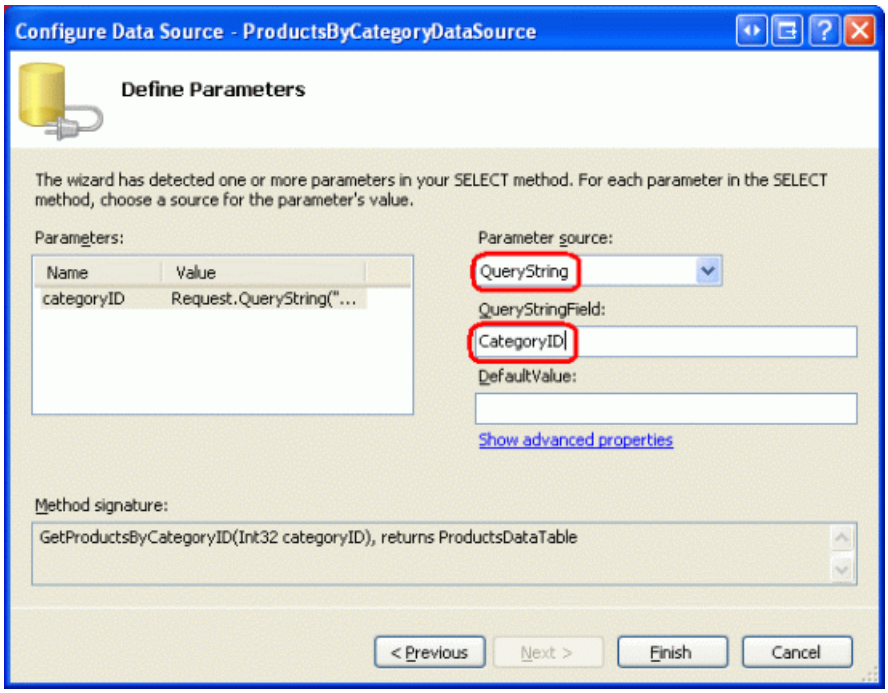


Figure 9: Use the `CategoryID` Querystring Field for the *categoryID* Parameter (Click to view full-size image)

After completing the wizard, Visual Studio will add corresponding BoundFields and a CheckBoxField to the GridView for the product data fields. Remove all but the `ProductName`, `UnitPrice`, and `SupplierName` BoundFields. Customize these three BoundFields `HeaderText` properties to read Product, Price, and Supplier, respectively. Format the `UnitPrice` BoundField as a currency.

Next, add a HyperLinkField and move it to the left-most position. Set its `Text` property to View Details, its `DataNavigateUrlFields` property to `ProductID`, and its `DataNavigateUrlFormatString` property to `~/SiteMapProvider/ProductDetails.aspx?ProductID={0}`.

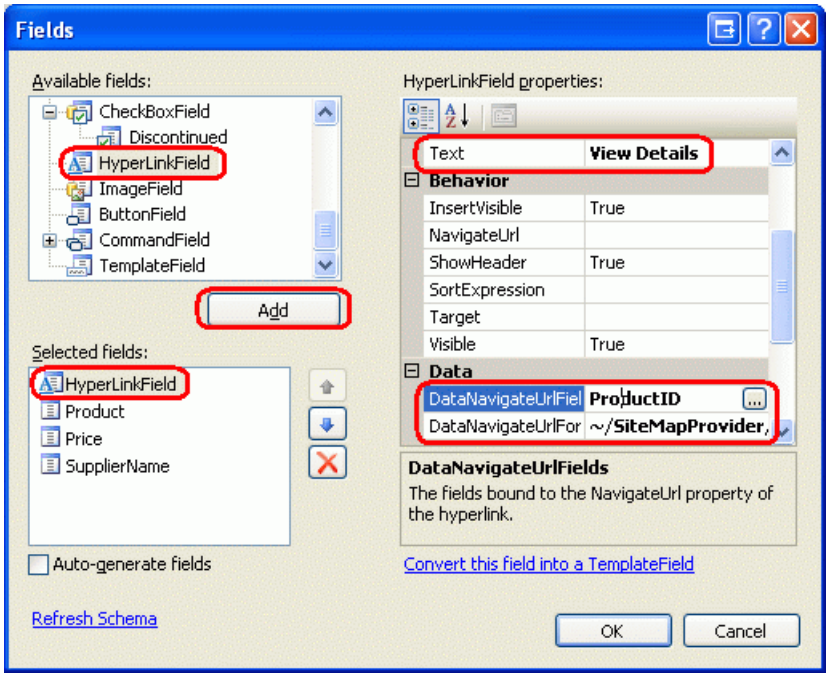


Figure 10: Add a View Details HyperLinkField that Points to `ProductDetails.aspx`

After making these customizations, the GridView and ObjectDataSource s declarative markup should resemble the following:

```
<asp:GridView ID="ProductsByCategory" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsByCategoryDataSource"
    EnableViewState="False">
    <Columns>
        <asp:HyperLinkField DataNavigateUrlFields="ProductID"
            DataNavigateUrlFormatString=
                "~/SiteMapProvider/ProductDetails.aspx?ProductID={0}"
            Text="View Details" />
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
            HeaderText="Price" HtmlEncode="False"
            SortExpression="UnitPrice" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
            ReadOnly="True" SortExpression="SupplierName" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsByCategoryDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProductsByCategoryID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:QueryStringParameter Name="categoryID"
            QueryStringField="CategoryID" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

Return to viewing `Default.aspx` through a browser and click on the View Products link for Beverages. This will

take you to `ProductsByCategory.aspx?CategoryID=1`, displaying the names, prices, and suppliers of the products in the Northwind database that belong to the Beverages category (see Figure 11). Feel free to further enhance this page to include a link to return users to the category listing page (`Default.aspx`) and a DetailsView or FormView control that displays the selected category s name and description.

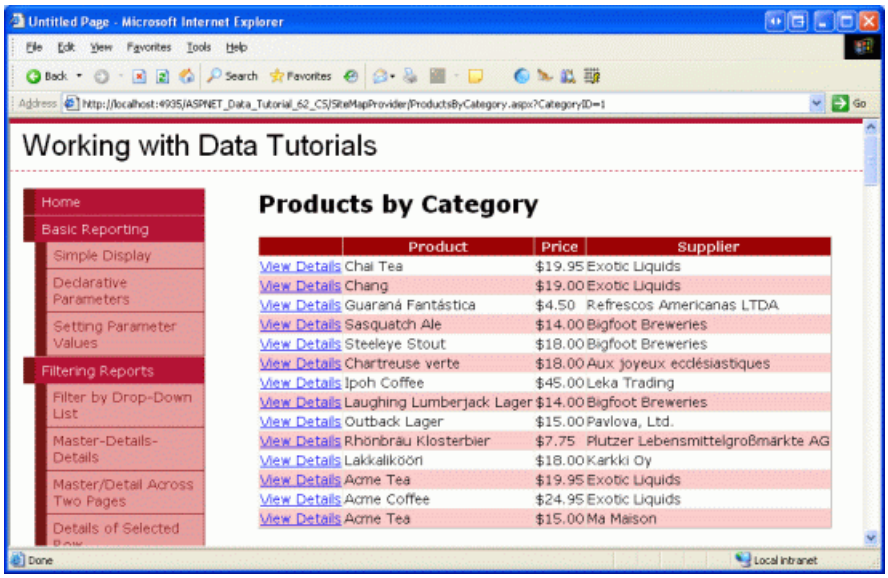


Figure 11: The Beverages Names, Prices, and Suppliers are Displayed (Click to view full-size image)

Step 4: Showing a Product s Details

The final page, `ProductDetails.aspx`, displays the selected products details. Open `ProductDetails.aspx` and drag a DetailsView from the Toolbox onto the Designer. Set the DetailsView s `ID` property to `ProductInfo` and clear out its `Height` and `Width` property values. From its smart tag, bind the DetailsView to a new ObjectDataSource named `ProductDataSource`, configuring the ObjectDataSource to pull its data from the `ProductsBLL` class s `GetProductByProductID(productID)` method. As with the previous web pages created in Steps 2 and 3, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) .

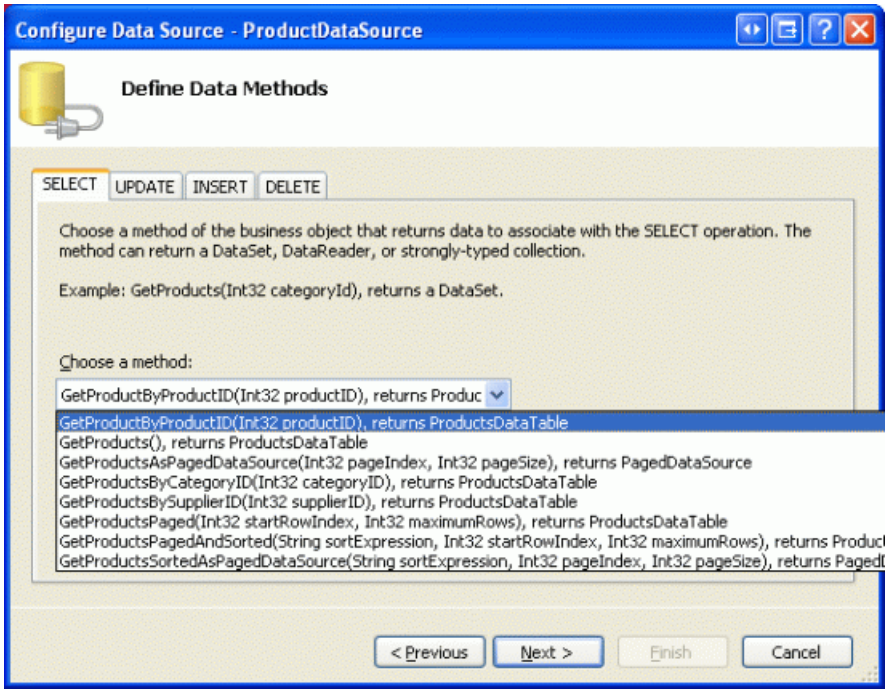


Figure 12: Configure the ObjectDataSource to Use the `GetProductByProductID(productID)` Method (Click to view full-size image)

The last step of the Configure Data Source wizard prompts for the source of the *productID* parameter. Since this

data comes through the querystring field `ProductID`, set the drop-down list to QueryString and the QueryStringField textbox to ProductID. Finally, click the Finish button to complete the wizard.

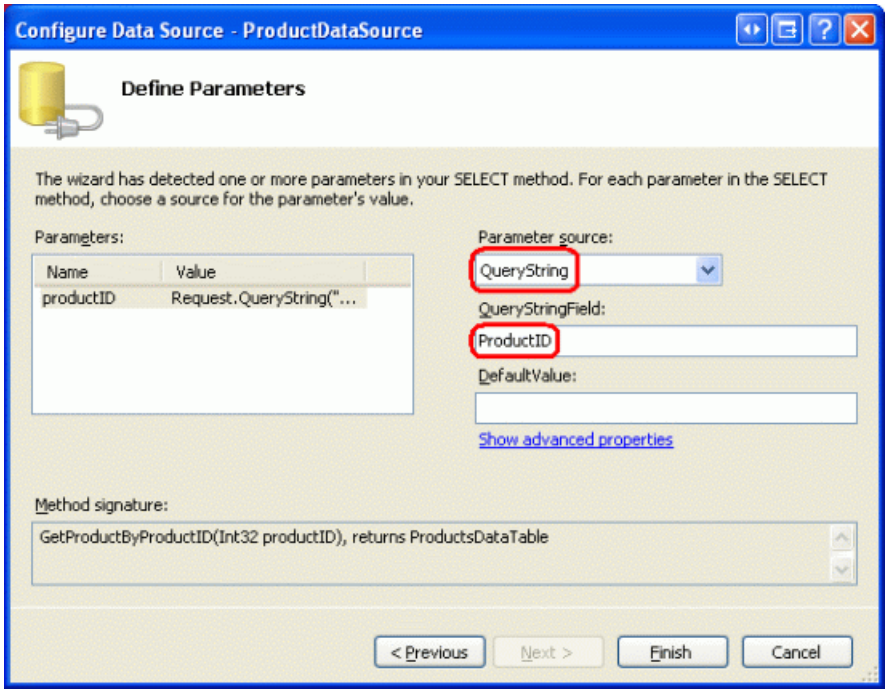


Figure 13: Configure the *productID* Parameter to Pull its Value from the `ProductID` Querystring Field (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will create corresponding BoundFields and a CheckBoxField in the DetailsView for the product data fields. Remove the `ProductID`, `SupplierID`, and `CategoryID` BoundFields and configure the remaining fields as you see fit. After a handful of aesthetic configurations, my DetailsView and ObjectDataSource s declarative markup looked like the following:

```
<asp:DetailsView ID="ProductInfo" runat="server" AutoGenerateRows="False"
    DataKeyNames="ProductID" DataSourceID="ProductDataSource"
    EnableViewState="False">
    <Fields>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            ReadOnly="True" SortExpression="CategoryName" />
        <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
            ReadOnly="True" SortExpression="SupplierName" />
        <asp:BoundField DataField="QuantityPerUnit" HeaderText="Qty/Unit"
            SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
            HeaderText="Price" HtmlEncode="False"
            SortExpression="UnitPrice" />
        <asp:BoundField DataField="UnitsInStock" HeaderText="Units In Stock"
            SortExpression="UnitsInStock" />
        <asp:BoundField DataField="UnitsOnOrder" HeaderText="Units On Order"
            SortExpression="UnitsOnOrder" />
        <asp:BoundField DataField="ReorderLevel" HeaderText="Reorder Level"
            SortExpression="ReorderLevel" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Fields>
</asp:DetailsView>
<asp:ObjectDataSource ID="ProductDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProductByProductID" TypeName="ProductsBLL">
    <SelectParameters>
        <asp:QueryStringParameter Name="productID"
            QueryStringField="ProductID" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

To test this page, return to `Default.aspx` and click on View Products for the Beverages category. From the listing of beverage products, click on the View Details link for Chai Tea. This will take you to `ProductDetails.aspx?ProductID=1`, which shows a Chai Tea s details (see Figure 14).

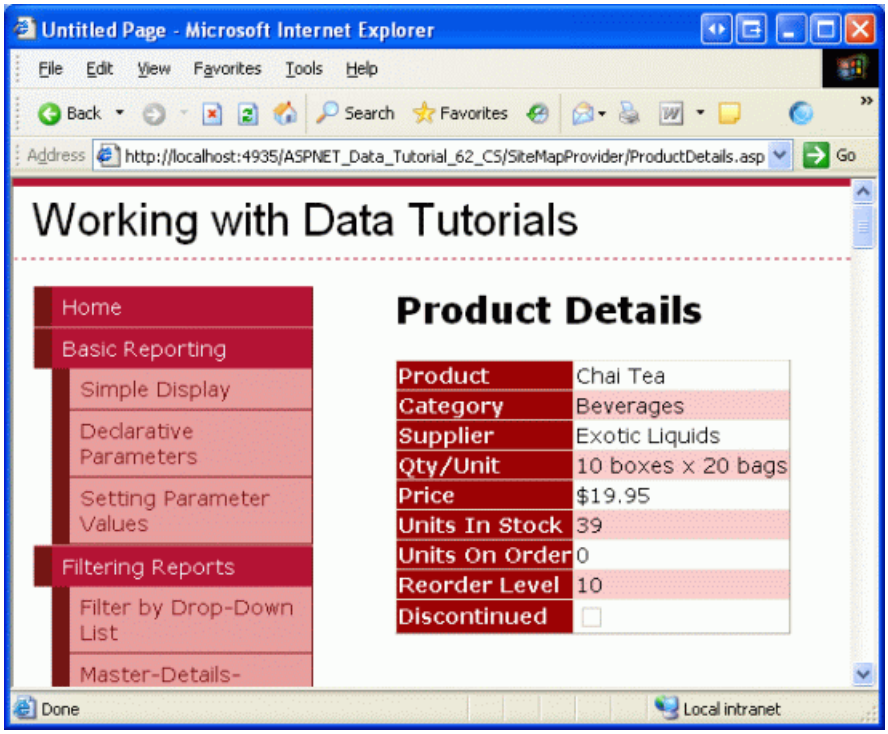


Figure 14: Chai Tea s Supplier, Category, Price, and Other Information is Displayed (Click to view full-size image)

Step 5: Understanding the Inner Workings of a Site Map Provider

The site map is represented in the web server s memory as a collection of `SiteMapNode` instances that form a hierarchy. There must be exactly one root, all non-root nodes must have exactly one parent node, and all nodes may have an arbitrary number of children. Each `SiteMapNode` object represents a section in the website s structure; these sections commonly have a corresponding web page. Consequently, the `SiteMapNode` class has properties like `Title`, `Url`, and `Description`, which provide information for the section the `SiteMapNode` represents. There is also a `Key` property that uniquely identifies each `SiteMapNode` in the hierarchy, as well as properties used to establish this hierarchy `ChildNodes`, `ParentNode`, `NextSibling`, `PreviousSibling`, and so forth.

Figure 15 shows the general site map structure from Figure 1, but with the implementation details sketched out in finer detail.

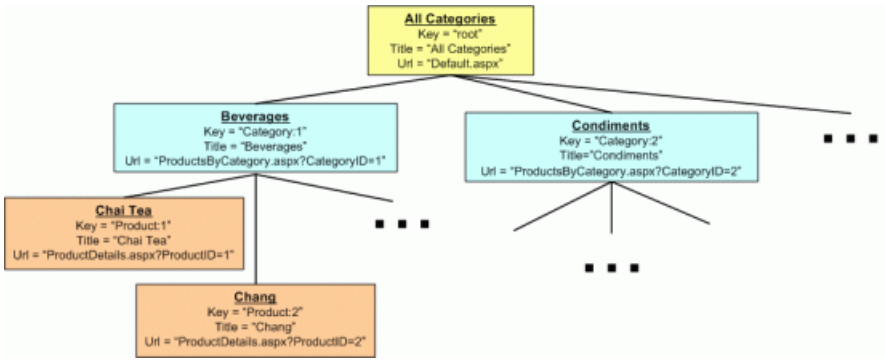


Figure 15: Each `SiteMapNode` has Properties Like `Title`, `Url`, `Key`, and So On (Click to view full-size image)

The site map is accessible through the `SiteMap` class in the `System.Web` namespace. This class s `RootNode` property returns the site map s root `SiteMapNode` instance; `CurrentNode` returns the `SiteMapNode` whose `Url` property matches the URL of the currently requested page. This class is used internally by ASP.NET 2.0 s navigation Web controls.

When the `SiteMap` class s properties are accessed, it must serialize the site map structure from some persistent medium into memory. However, the site map serialization logic is not hard coded into the `SiteMap` class. Instead, at runtime the `SiteMap` class determines which site map *provider* to use for serialization. By default, the `XmlSiteMapProvider` class is used, which reads the site map s structure from a properly-formatted XML file. However, with a little bit of work we can create our own custom site map provider.

All site map providers must be derived from the `SiteMapProvider` class, which includes the essential methods and properties needed for site map providers, but omits many of the implementation details. A second class, `StaticSiteMapProvider`, extends the `SiteMapProvider` class and contains a more robust implementation of the needed functionality. Internally, the `StaticSiteMapProvider` stores the `SiteMapNode` instances of the site map in a `Hashtable` and provides methods like `AddNode(child, parent)`, `RemoveNode(siteMapNode)`, and `Clear()` that add and remove `SiteMapNode` s to the internal `Hashtable`. `XmlSiteMapProvider` is derived from `StaticSiteMapProvider`.

When creating a custom site map provider that extends `StaticSiteMapProvider`, there are two abstract methods that must be overridden: `BuildSiteMap` and `GetRootNodeCore`. `BuildSiteMap`, as its name implies, is responsible for loading the site map structure from persistent storage and constructing it in memory. `GetRootNodeCore` returns the root node in the site map.

Before a web application can use a site map provider it must be registered in the application s configuration. By default, the `XmlSiteMapProvider` class is registered using the name `AspNetXmlSiteMapProvider`. To register additional site map providers, add the following markup to `Web.config`:

```
<configuration>
    <system.web>
        ...
        <siteMap defaultProvider="defaultProviderName">
          <providers>
            <add name="name" type="type" />
          </providers>
        </siteMap>
    </system.web>
</configuration>
```

The *name* value assigns a human-readable name to the provider while *type* specifies the fully-qualified type name of the site map provider. We'll explore concrete values for the *name* and *type* values in Step 7, after we ve created our custom site map provider.

The site map provider class is instantiated the first time it is accessed from the `SiteMap` class and remains in memory for the lifetime of the web application. Since there is only one instance of the site map provider that may be invoked from multiple, concurrent web site visitors, it is imperative that the provider s methods be *thread-safe*.

For performance and scalability reasons, it s important that we cache the in-memory site map structure and return this cached structure rather than recreating it every time the `BuildSiteMap` method is invoked. `BuildSiteMap` may be called several times per page request per user, depending on the navigation controls in use on the page and the depth of the site map structure. In any case, if we do not cache the site map structure in `BuildSiteMap` then each time it is invoked we would need to re-retrieve the product and category information from the architecture (which would result in a query to the database). As we discussed in the previous caching tutorials, cached data can become stale. To combat this, we can use either time- or SQL cache dependency-based expiries.

NOTE

A site map provider may optionally override the `Initialize` method. `Initialize` is invoked when the site map provider is first instantiated and is passed any custom attributes assigned to the provider in `Web.config` in the `<add>` element like: `<add name="name" type="type" customAttribute="value" />`. It is useful if you want to allow a page developer to specify various site map provider-related settings without having to modify the provider s code. For example, if we were reading the category and products data directly from the database as opposed to through the architecture, we d likely want to let the page developer specify the database connection string through `Web.config` rather than using a hard coded value in the provider s code. The custom site map provider we'll build in Step 6 does not override this `Initialize` method. For an example of using the `Initialize` method, refer to Jeff Prosise s Storing Site Maps in SQL Server article.

Step 6: Creating the Custom Site Map Provider

To create a custom site map provider that builds the site map from the categories and products in the Northwind database, we need to create a class that extends `StaticSiteMapProvider`. In Step 1 I asked you to add a `CustomProviders` folder in the `App_Code` folder - add a new class to this folder named `NorthwindSiteMapProvider`. Add the following code to the `NorthwindSiteMapProvider` class:

```
Imports System.Web
Imports System.Web.Caching
Public Class NorthwindSiteMapProvider
    Inherits StaticSiteMapProvider
    Private ReadOnly siteMapLock As New Object()
    Private root As SiteMapNode = Nothing
    Public Const CacheDependencyKey As String = "NorthwindSiteMapProviderCacheDependency"
    Public Overrides Function BuildSiteMap() As System.Web.SiteMapNode
        ' Use a lock to make this method thread-safe
        SyncLock siteMapLock
```

```
        ' First, see if we already have constructed the
        ' rootNode. If so, return it...
        If root IsNot Nothing Then
            Return root
        End If
        ' We need to build the site map!
        ' Clear out the current site map structure
        MyBase.Clear()
        ' Get the categories and products information from the database
        Dim productsAPI As New ProductsBLL()
        Dim products As Northwind.ProductsDataTable = productsAPI.GetProducts()
        ' Create the root SiteMapNode
        root = New SiteMapNode( _
            Me, "root", "~/SiteMapProvider/Default.aspx", "All Categories")
        AddNode(root)
        ' Create SiteMapNodes for the categories and products
        For Each product As Northwind.ProductsRow In products
            ' Add a new category SiteMapNode, if needed
            Dim categoryKey, categoryName As String
            Dim createUrlForCategoryNode As Boolean = True
            If product.IsCategoryIDNull() Then
                categoryKey = "Category:None"
                categoryName = "None"
                createUrlForCategoryNode = False
            Else
                categoryKey = String.Concat("Category:", product.CategoryID)
                categoryName = product.CategoryName
            End If
            Dim categoryNode As SiteMapNode = FindSiteMapNodeFromKey(categoryKey)
            ' Add the category SiteMapNode if it does not exist
            If categoryNode Is Nothing Then
                Dim productsByCategoryUrl As String = String.Empty
                If createUrlForCategoryNode Then
                    productsByCategoryUrl = _
                        "~/SiteMapProvider/ProductsByCategory.aspx?CategoryID=" & _
                        product.CategoryID
                End If
                categoryNode = New SiteMapNode _
                    (Me, categoryKey, productsByCategoryUrl, categoryName)
                AddNode(categoryNode, root)
            End If
            ' Add the product SiteMapNode
            Dim productUrl As String = _
                "~/SiteMapProvider/ProductDetails.aspx?ProductID=" & _
                product.ProductID
            Dim productNode As New SiteMapNode _
                (Me, String.Concat("Product:", product.ProductID), _
                productUrl, product.ProductName)
            AddNode(productNode, categoryNode)
        Next
        ' Add a "dummy" item to the cache using a SqlCacheDependency
        ' on the Products and Categories tables
        Dim productsTableDependency As New _
            System.Web.Caching.SqlCacheDependency("NorthwindDB", "Products")
        Dim categoriesTableDependency As New _
            System.Web.Caching.SqlCacheDependency("NorthwindDB", "Categories")
        ' Create an AggregateCacheDependency
        Dim aggregateDependencies As New System.Web.Caching.AggregateCacheDependency()
        aggregateDependencies.Add(productsTableDependency, categoriesTableDependency)
        ' Add the item to the cache specifying a callback function
        HttpRuntime.Cache.Insert( _
            CacheDependencyKey, DateTime.Now, aggregateDependencies, _
            Cache.NoAbsoluteExpiration, Cache.NoSlidingExpiration, _
            CacheItemPriority.Normal, AddressOf OnSiteMapChanged)
        ' Finally, return the root node
        Return root
    End SyncLock
End Function
Protected Overrides Function GetRootNodeCore() As System.Web.SiteMapNode
```

```
        Return BuildSiteMap()
    End Function
    Protected Sub OnSiteMapChanged _
    (key As String, value As Object, reason As CacheItemRemovedReason)
        SyncLock siteMapLock
            If String.Compare(key, CacheDependencyKey) = 0 Then
                ' Refresh the site map
                root = Nothing
            End If
        End SyncLock
    End Sub
    Public ReadOnly Property CachedDate() As Nullable(Of DateTime)
        Get
            Dim value As Object = HttpRuntime.Cache(CacheDependencyKey)
            If value Is Nothing OrElse Not TypeOf value Is Nullable(Of DateTime) Then
                Return Nothing
            Else
                Return CType(value, Nullable(Of DateTime))
            End If
        End Get
    End Property
End Class
```

Let s start with exploring this class s `BuildSiteMap` method, which starts with a `lock` statement. The `lock` statement only allows one thread at a time to enter, thereby serializing access to its code and preventing two concurrent threads from stepping on one another s toes.

The class-level `SiteMapNode` variable `root` is used to cache the site map structure. When the site map is constructed for the first time, or for the first time after the underlying data has been modified, `root` will be `Nothing` and the site map structure will be constructed. The site map s root node is assigned to `root` during the construction process so that the next time this method is called, `root` will not be `Nothing`. Consequently, so long as `root` is not `Nothing` the site map structure will be returned to the caller without having to recreate it.

If root is `Nothing`, the site map structure is created from the product and category information. The site map is built by creating the `SiteMapNode` instances and then forming the hierarchy through calls to the `StaticSiteMapProvider` class s `AddNode` method. `AddNode` performs the internal bookkeeping, storing the assorted `SiteMapNode` instances in a `Hashtable`. Before we start constructing the hierarchy, we start by calling the `Clear` method, which clears out the elements from the internal `Hashtable`. Next, the `ProductsBLL` class s `GetProducts` method and the resulting `ProductsDataTable` are stored in local variables.

The site map s construction begins by creating the root node and assigning it to `root`. The overload of the `SiteMapNode` s constructor used here and throughout this `BuildSiteMap` is passed the following information:

- A reference to the site map provider (`Me`).
- The `SiteMapNode` s `Key`. This required value must be unique for each `SiteMapNode`.
- The `SiteMapNode` s `Url`. `Url` is optional, but if provided, each `SiteMapNode` s `Url` value must be unique.
- The `SiteMapNode` s `Title`, which is required.

The `AddNode(root)` method call adds the `SiteMapNode` `root` to the site map as the root. Next, each `ProductRow` in the `ProductsDataTable` is enumerated. If there already exists a `SiteMapNode` for the current product s category, it is referenced. Otherwise, a new `SiteMapNode` for the category is created and added as a child of the `SiteMapNode``root` through the `AddNode(categoryNode, root)` method call. After the appropriate category `SiteMapNode` node has been found or created, a `SiteMapNode` is created for the current product and added as a child of the category `SiteMapNode` via `AddNode(productNode, categoryNode)`. Note that the category `SiteMapNode` s `Url` property value is `~/SiteMapProvider/ProductsByCategory.aspx?CategoryID=categoryID` while the product `SiteMapNode` s `Url` property is assigned `~/SiteMapNode/ProductDetails.aspx?ProductID=productID`.

NOTE

Those products that have a database `NULL` value for their `CategoryID` are grouped under a category `SiteMapNode` whose `Title` property is set to None and whose `Url` property is set to an empty string. I decided to set `Url` to an empty string since the `ProductBLL` class s `GetProductsByCategory(categoryID)` method currently lacks the capability to return just those products with a `NULL` `CategoryID` value. Also, I wanted to demonstrate how the navigation controls render a `SiteMapNode` that lacks a value for its `Url` property. I encourage you to extend this tutorial so that the None `SiteMapNode` s `Url` property points to `ProductsByCategory.aspx`, yet only displays the products with `NULL` `CategoryID` values.

After constructing the site map, an arbitrary object is added to the data cache using a SQL cache dependency on the `Categories` and `Products` tables through an `AggregateCacheDependency` object. We explored using SQL cache dependencies in the preceding tutorial, *Using SQL Cache Dependencies*. The custom site map provider, however, uses an overload of the data cache s `Insert` method that we ve yet to explore. This overload accepts as its final input parameter a delegate that is called when the object is removed from the cache. Specifically, we pass in a new `CacheItemRemovedCallback` delegate that points to the `OnSiteMapChanged` method defined further down in the `NorthwindSiteMapProvider` class.

NOTE

The in-memory representation of the site map is cached through the class-level variable `root`. Since there is only one instance of the custom site map provider class and since that instance is shared among all threads in the web application, this class variable serves as a cache. The `BuildSiteMap` method also uses the data cache, but only as a means to receive notification when the underlying database data in the `Categories` or `Products` tables changes. Note that the value put into the data cache is just the current date and time. The actual site map data is *not* put in the data cache.

The `BuildSiteMap` method completes by returning the root node of the site map.

The remaining methods are fairly straightforward. `GetRootNodeCore` is responsible for returning the root node. Since `BuildSiteMap` returns the root, `GetRootNodeCore` simply returns `BuildSiteMap` s return value. The `OnSiteMapChanged` method sets `root` back to `Nothing` when the cache item is removed. With root set back to `Nothing`, the next time `BuildSiteMap` is invoked, the site map structure will be rebuilt. Lastly, the `CachedDate` property returns the date and time value stored in the data cache, if such a value exists. This property can be used by a page developer to determine when the site map data was last cached.

Step 7: Registering the `NorthwindSiteMapProvider`

In order for our web application to use the `NorthwindSiteMapProvider` site map provider created in Step 6, we need to register it in the `<siteMap>` section of `Web.config`. Specifically, add the following markup within the `<system.web>` element in `Web.config`:

```
<siteMap defaultProvider="AspNetXmlSiteMapProvider">
  <providers>
    <add name="Northwind" type="NorthwindSiteMapProvider" />
  </providers>
</siteMap>
```

This markup does two things: first, it indicates that the built-in `AspNetXmlSiteMapProvider` is the default site map provider; second, it registers the custom site map provider created in Step 6 with the human-friendly name Northwind .

> **NOTE**
> For site map providers located in the application s `App_Code` folder, the value of the `type` attribute is simply the class name. Alternatively, the custom site map provider could have been created in a separate Class Library project with the compiled assembly placed in the web application s `/Bin` directory. In that case, the `type` attribute value would be *Namespace.ClassName, AssemblyName* .

After updating `Web.config`, take a moment to view any page from the tutorials in a browser. Note that the navigation interface on the left still shows the sections and tutorials defined in `Web.sitemap`. This is because we left `AspNetXmlSiteMapProvider` as the default provider. In order to create a navigation user interface element that uses the `NorthwindSiteMapProvider`, we'll need to explicitly specify that the Northwind site map provider should be used. We'll see how to accomplish this in Step 8.

Step 8: Displaying Site Map Information Using the Custom Site Map Provider

With the custom site map provider created and registered in `Web.config`, we re ready to add navigation controls to the `Default.aspx`, `ProductsByCategory.aspx`, and `ProductDetails.aspx` pages in the `SiteMapProvider` folder. Start by opening the `Default.aspx` page and drag a `SiteMapPath` from the Toolbox onto the Designer. The SiteMapPath control is located in the Navigation section of the Toolbox.

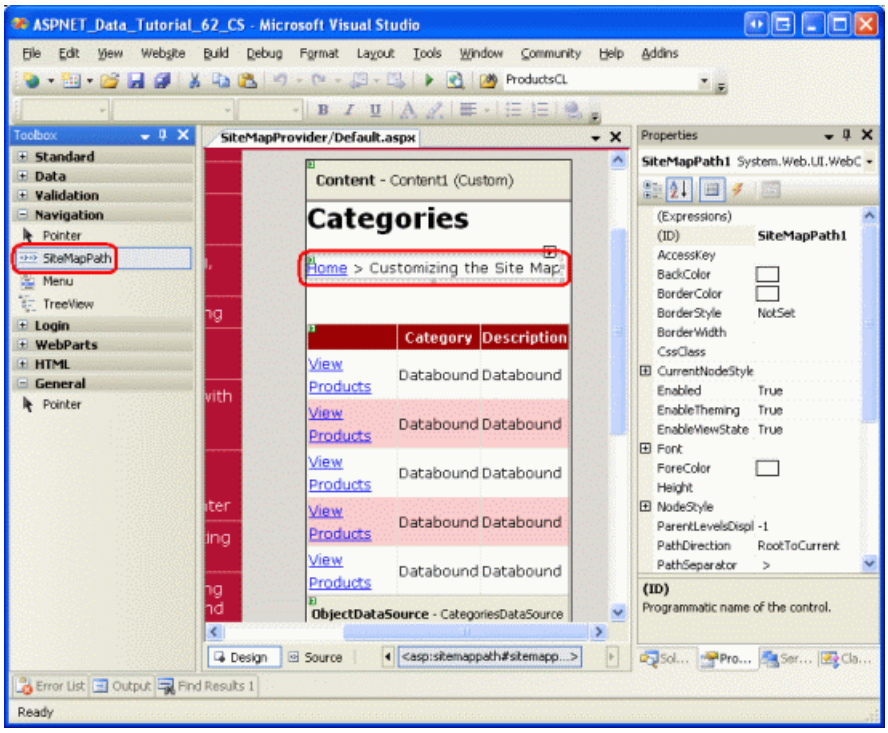


Figure 16: Add a SiteMapPath to `Default.aspx` (Click to view full-size image)

The SiteMapPath control displays a breadcrumb, indicating the current page s location within the site map. We added a SiteMapPath to the top of the master page back in the *Master Pages and Site Navigation* tutorial.

Take a moment to view this page through a browser. The SiteMapPath added in Figure 16 uses the default site map provider, pulling its data from `Web.sitemap`. Therefore, the breadcrumb shows Home > Customizing the Site Map, just like the breadcrumb in the upper-right corner.

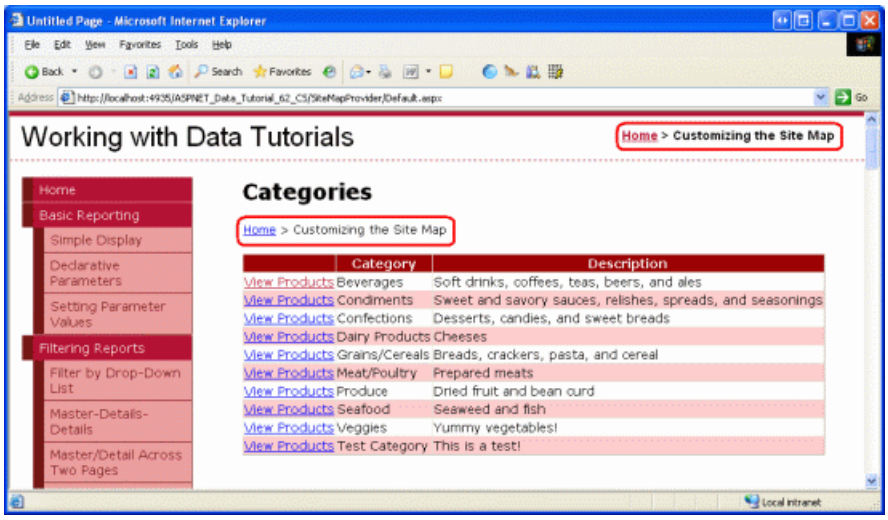


Figure 17: The Breadcrumb Uses the Default Site Map Provider (Click to view full-size image)

To have the SiteMapPath added in Figure 16 use the custom site map provider we created in Step 6, set its `SiteMapProvider` property to Northwind, the name we assigned to the `NorthwindSiteMapProvider` in `Web.config`. Unfortunately, the Designer continues to use the default site map provider, but if you visit the page through a browser after making this property change you'll see that the breadcrumb now uses the custom site map provider.

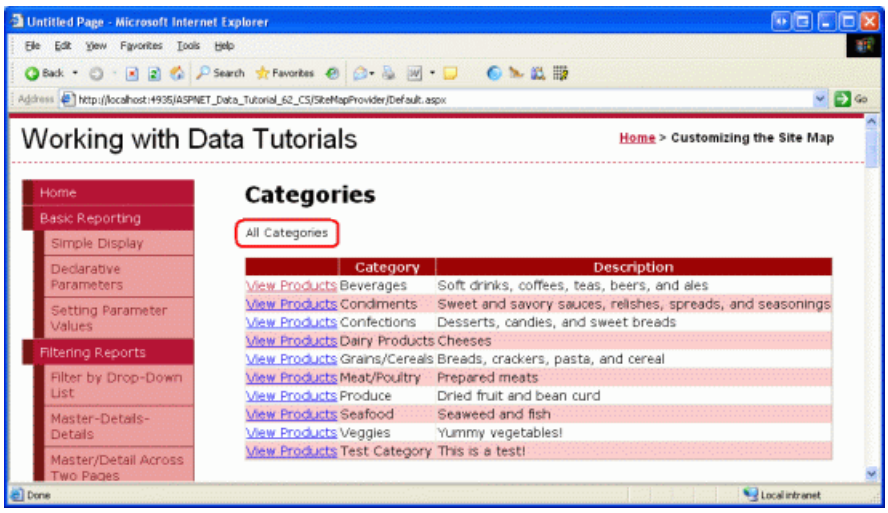


Figure 18: The Breadcrumb Now Uses the Custom Site Map Provider `NorthwindSiteMapProvider` (Click to view full-size image)

The SiteMapPath control displays a more functional user interface in the `ProductsByCategory.aspx` and `ProductDetails.aspx` pages. Add a SiteMapPath to these pages, setting the `SiteMapProvider` property in both to Northwind. From `Default.aspx` click on the View Products link for Beverages, and then on the View Details link for Chai Tea. As Figure 19 shows, the breadcrumb includes the current site map section (Chai Tea) and its ancestors: Beverages and All Categories .

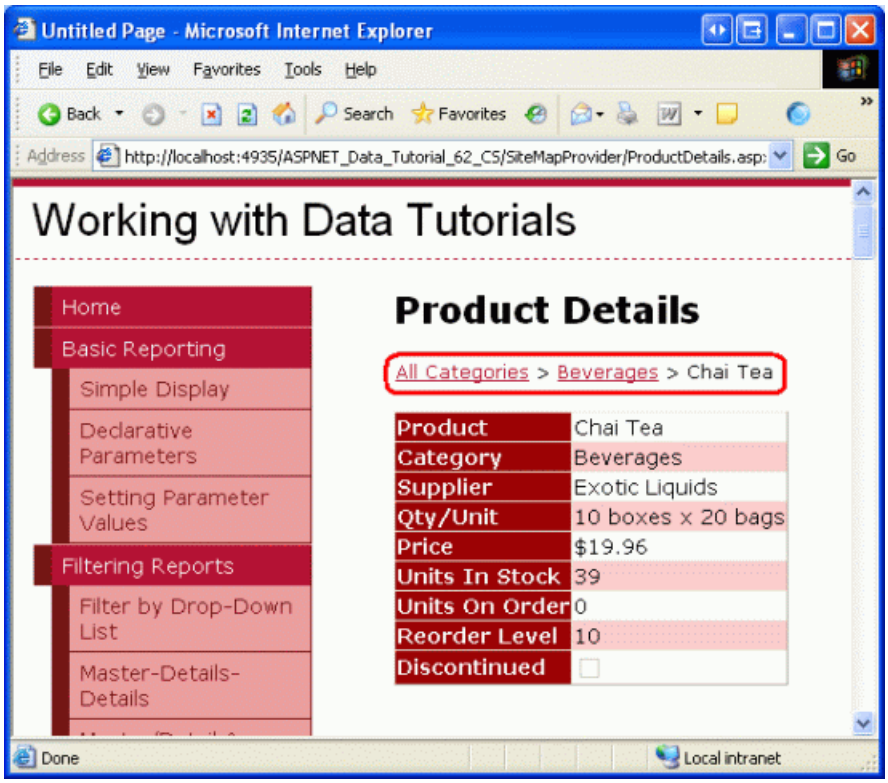


Figure 19: The Breadcrumb Now Uses the Custom Site Map Provider `NorthwindSiteMapProvider` (Click to view full-size image)

Other navigation user interface elements can be used in addition to the SiteMapPath, such as the Menu and TreeView controls. The `Default.aspx`, `ProductsByCategory.aspx`, and `ProductDetails.aspx` pages in the download for this tutorial, for example, all include Menu controls (see Figure 20). See Examining ASP.NET 2.0 s Site Navigation Features and the Using Site Navigation Controls section of the ASP.NET 2.0 QuickStarts for a more in-depth look at the navigation controls and site map system in ASP.NET 2.0.

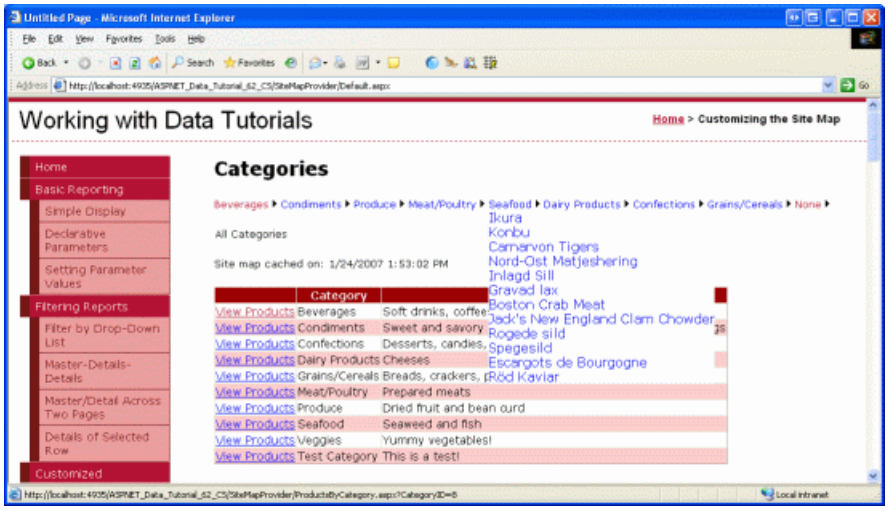


Figure 20: The Menu Control Lists Each of the Categories and Products (Click to view full-size image)

As mentioned earlier in this tutorial, the site map structure can be accessed programmatically through the `SiteMap` class. The following code returns the root `SiteMapNode` of the default provider:

```
Dim root As SiteMapNode = SiteMap.RootNode
```

Since the `AspNetXmlSiteMapProvider` is the default provider for our application, the above code would return the root node defined in `Web.sitemap`. To reference a site map provider other than the default, use the `SiteMap` class s `Providers` property like so:

```
Dim root As SiteMapNode = SiteMap.Providers("name").RootNode
```

Where *name* is the name of the custom site map provider (Northwind, for our web application).

To access a member specific to a site map provider, use `SiteMap.Providers["name"]` to retrieve the provider instance and then cast it to the appropriate type. For example, to display the `NorthwindSiteMapProvider` s `CachedDate` property in an ASP.NET page, use the following code:

```
Dim customProvider As NorthwindSiteMapProvider = _
    TryCast(SiteMap.Providers("Northwind"), NorthwindSiteMapProvider)
If customProvider IsNot Nothing Then
    Dim lastCachedDate As Nullable(Of DateTime) = customProvider.CachedDate
    If lastCachedDate.HasValue Then
        SiteMapLastCachedDate.Text = _
            "Site map cached on: " & lastCachedDate.Value.ToString()
    Else
        SiteMapLastCachedDate.Text = "The site map is being reconstructed!"
    End If
End If
```

NOTE

Be sure to test out the SQL cache dependency feature. After visiting the `Default.aspx`, `ProductsByCategory.aspx`, and `ProductDetails.aspx` pages, go to one of the tutorials in the Editing, Inserting, and Deleting section and edit the name of a category or product. Then return to one of the pages in the `SiteMapProvider` folder. Assuming enough time has passed for the polling mechanism to note the change to the underlying database, the site map should be updated to show the new product or category name.

Summary

ASP.NET 2.0 s site map features includes a `SiteMap` class, a number of built-in navigation Web controls, and a default site map provider that expects the site map information persisted to an XML file. In order to use site map information from some other source such as from a database, the application s architecture, or a remote Web service we need to create a custom site map provider. This involves creating a class that derives, directly or indirectly, from the `SiteMapProvider` class.

In this tutorial we saw how to create a custom site map provider that based the site map on the product and category information culled from the application architecture. Our provider extended the `StaticSiteMapProvider` class and entailed creating a `BuildSiteMap` method that retrieved the data, constructed the site map hierarchy, and cached the resulting structure in a class-level variable. We used a SQL cache dependency with a callback function to invalidate the cached structure when the underlying `Categories` or `Products` data is modified.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Storing Site Maps in SQL Server and The SQL Site Map Provider You ve Been Waiting For
- A Look at ASP.NET 2.0 s Provider Model
- The Provider Toolkit
- Examining ASP.NET 2.0 s Site Navigation Features

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Dave Gardner, Zack Jones, Teresa Murphy, and Bernadette Leigh. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Working with Batched Data

2/20/2020 • 2 minutes to read • Edit Online

> These tutorials look at updating, deleting, and inserting batches of data using transactions.

- Wrapping Database Modifications within a Transaction (C#)
- Batch Updating (C#)
- Batch Deleting (C#)
- Batch Inserting (C#)
- Wrapping Database Modifications within a Transaction (VB)
- Batch Updating (VB)
- Batch Deleting (VB)
- Batch Inserting (VB)

Wrapping Database Modifications within a Transaction (C#)

2/20/2020 • 23 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> This tutorial is the first of four that looks at updating, deleting, and inserting batches of data. In this tutorial we learn how database transactions allow batch modifications to be carried out as an atomic operation, which ensures that either all steps succeed or all steps fail.

Introduction

As we saw starting with the An Overview of Inserting, Updating, and Deleting Data tutorial, the GridView provides built-in support for row-level editing and deleting. With a few clicks of the mouse it is possible to create a rich data modification interface without writing a line of code, so long as you are content with editing and deleting on a per-row basis. However, in certain scenarios this is insufficient and we need to provide users with the ability to edit or delete a batch of records.

For example, most web-based email clients use a grid to list each message where each row includes a checkbox along with the email s information (subject, sender, and so forth). This interface permits the user to delete multiple messages by checking them and then clicking a Delete Selected Messages button. A batch editing interface is ideal in situations where users commonly edit many different records. Rather than forcing the user to click Edit, make their change, and then click Update for each record that needs to be modified, a batch editing interface renders each row with its editing interface. The user can quickly modify the set of rows that need to be changed and then save these changes by clicking an Update All button. In this set of tutorials we'll examine how to create interfaces for inserting, editing, and deleting batches of data.

When performing batch operations it s important to determine whether it should be possible for some of the operations in the batch to succeed while others fail. Consider a batch deleting interface - what should happen if the first selected record is deleted successfully, but the second one fails, say, because of a foreign key constraint violation? Should the first record s delete be rolled back or is it acceptable for the first record to remain deleted?

If you want the batch operation to be treated as an atomic operation, one where either all of the steps succeed or all of the steps fail, then the Data Access Layer needs to be augmented to include support for database transactions. Database transactions guarantee atomicity for the set of `INSERT`, `UPDATE`, and `DELETE` statements executed under the umbrella of the transaction and are a feature supported by most all modern database systems.

In this tutorial we'll look at how to extend the DAL to use database transactions. Subsequent tutorials will examine implementing web pages for batch inserting, updating, and deleting interfaces. Let s get started!

NOTE

When modifying data in a batch transaction, atomicity is not always needed. In some scenarios, it may be acceptable to have some data modifications succeed and others in the same batch fail, such as when deleting a set of emails from a web-based email client. If there s a database error midway through the deletion process, it s probably acceptable that those records processed without error remain deleted. In such cases, the DAL does not need to be modified to support database transactions. There are other batch operation scenarios, however, where atomicity is vital. When a customer moves her funds from one bank account to another, two operations must be performed: the funds must be deducted from the first account and then added to the second. While the bank may not mind having the first step succeed but the second step fail, its customers would understandably be upset. I encourage you to work through this tutorial and implement the enhancements to the DAL to support database transactions even if you do not plan on using them in the batch inserting, updating, and deleting interfaces we'll be building in the following three tutorials.

An Overview of Transactions

Most databases include support for *transactions*, which enable multiple database commands to be grouped into a single logical unit of work. The database commands that comprise a transaction are guaranteed to be atomic, meaning that either all commands will fail or all will succeed.

In general, transactions are implemented through SQL statements using the following pattern:

1. Indicate the start of a transaction.
2. Execute the SQL statements that comprise the transaction.
3. If there is an error in one of the statements from Step 2, rollback the transaction.
4. If all of the statements from Step 2 complete without error, commit the transaction.

The SQL statements used to create, commit, and roll back the transaction can be entered manually when writing SQL scripts or creating stored procedures, or through programmatic means using either ADO.NET or the classes in the `System.Transactions` namespace. In this tutorial we will only examine managing transactions using ADO.NET. In a future tutorial we will look at how to use stored procedures in the Data Access Layer, at which time we'll explore the SQL statements for creating, rolling back, and committing transactions. In the meantime, consult Managing Transactions in SQL Server Stored Procedures for more information.

NOTE

The `TransactionScope` class in the `System.Transactions` namespace enables developers to programmatically wrap a series of statements within the scope of a transaction and includes support for complex transactions that involve multiple sources, such as two different databases or even heterogeneous types of data stores, such as a Microsoft SQL Server database, an Oracle database, and a Web service. I ve decided to use ADO.NET transactions for this tutorial instead of the `TransactionScope` class because ADO.NET is more specific for database transactions and, in many cases, is far less resource intensive. In addition, under certain scenarios the `TransactionScope` class uses the Microsoft Distributed Transaction Coordinator (MSDTC). The configuration, implementation, and performance issues surrounding MSDTC makes it a rather specialized and advanced topic and beyond the scope of these tutorials.

When working with the SqlClient provider in ADO.NET, transactions are initiated through a call to the `SqlConnection` class s `BeginTransaction` method, which returns a `SqlTransaction` object. The data modification statements that makeup the transaction are placed within a `try...catch` block. If an error occurs in a statement in the `try` block, execution transfers to the `catch` block where the transaction can be rolled back via the `SqlTransaction` object s `Rollback` method. If all of the statements complete successfully, a call to the `SqlTransaction` object s `Commit` method at the end of the `try` block commits the transaction. The following code snippet illustrates this pattern. See Maintaining Database Consistency with Transactions for additional syntax and examples of using transactions with ADO.NET.

```
// Create the SqlTransaction object
SqlTransaction myTransaction = SqlConnectionObject.BeginTransaction();
try
{
    /*
     * ... Perform the database transaction's data modification statements...
     */
    // If we reach here, no errors, so commit the transaction
    myTransaction.Commit();
}
catch
{
    // If we reach here, there was an error, so rollback the transaction
    myTransaction.Rollback();
    throw;
}
```

By default, the TableAdapters in a Typed DataSet do not use transactions. To provide support for transactions we need to augment the TableAdapter classes to include additional methods that use the above pattern to perform a series of data modification statements within the scope of a transaction. In Step 2 we'll see how to use partial classes to add these methods.

Step 1: Creating the Working with Batched Data Web Pages

Before we start exploring how to augment the DAL to support database transactions, let s first take a moment to create the ASP.NET web pages that we will need for this tutorial and the three that follow. Start by adding a new folder named `BatchData` and then add the following ASP.NET pages, associating each page with the `Site.master` master page.

- `Default.aspx`
- `Transactions.aspx`
- `BatchUpdate.aspx`
- `BatchDelete.aspx`
- `BatchInsert.aspx`

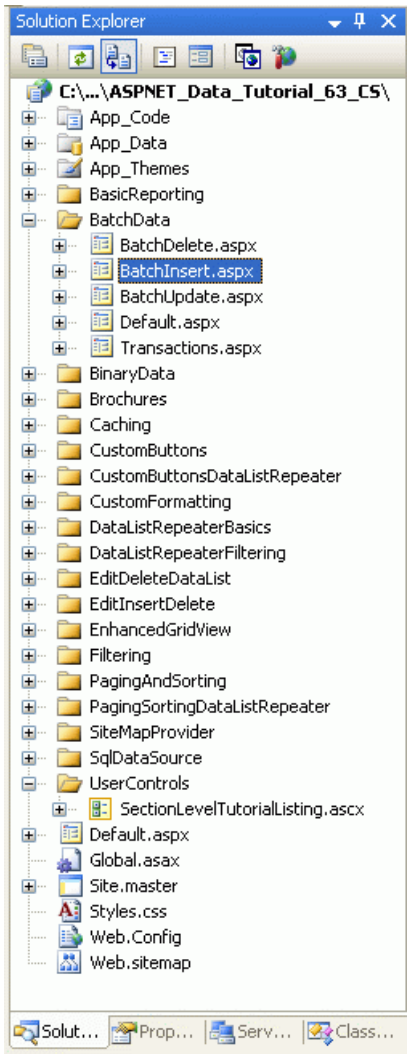


Figure 1: Add the ASP.NET Pages for the SqlDataSource-Related Tutorials

As with the other folders, `Default.aspx` will use the `SectionLevelTutorialListing.ascx` User Control to list the tutorials within its section. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

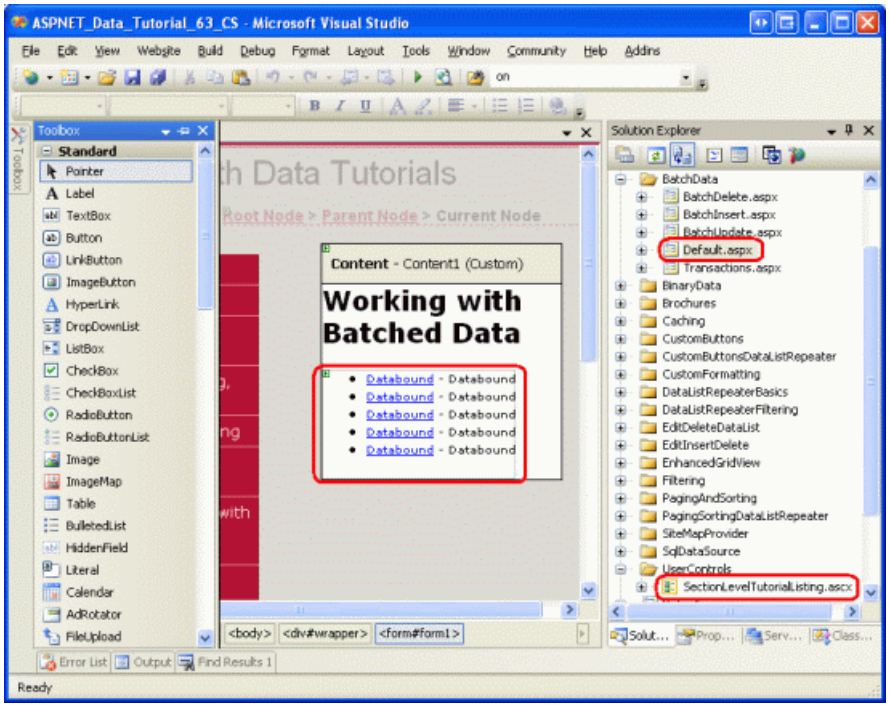


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size

image)

Lastly, add these four pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Customizing the Site Map `<siteMapNode>`:

```
<siteMapNode title="Working with Batched Data"
    url="~/BatchData/Default.aspx"
    description="Learn how to perform batch operations as opposed to
                 per-row operations.">

    <siteMapNode title="Adding Support for Transactions"
        url="~/BatchData/Transactions.aspx"
        description="See how to extend the Data Access Layer to support
                     database transactions." />
    <siteMapNode title="Batch Updating"
        url="~/BatchData/BatchUpdate.aspx"
        description="Build a batch updating interface, where each row in a
                      GridView is editable." />
    <siteMapNode title="Batch Deleting"
        url="~/BatchData/BatchDelete.aspx"
        description="Explore how to create an interface for batch deleting
                     by adding a CheckBox to each GridView row." />
    <siteMapNode title="Batch Inserting"
        url="~/BatchData/BatchInsert.aspx"
        description="Examine the steps needed to create a batch inserting
                     interface, where multiple records can be created at the
                     click of a button." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the working with batched data tutorials.

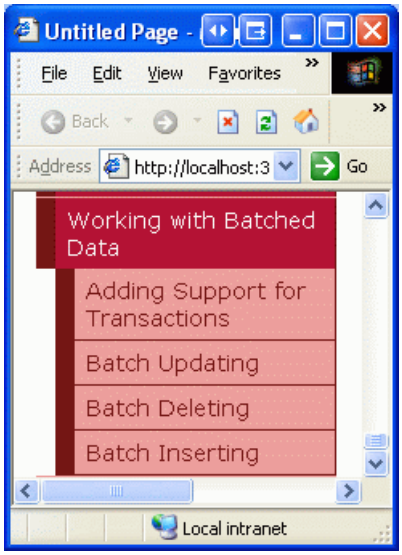


Figure 3: The Site Map Now Includes Entries for the Working with Batched Data Tutorials

Step 2: Updating the Data Access Layer to Support Database Transactions

As we discussed back in the first tutorial, Creating a Data Access Layer, the Typed DataSet in our DAL is composed of DataTables and TableAdapters. The DataTables hold data while the TableAdapters provide the functionality to read data from the database into the DataTables, to update the database with changes made to the DataTables, and so forth. Recall that the TableAdapters provide two patterns for updating data, which I referred to as Batch Update and DB-Direct. With the Batch Update pattern, the TableAdapter is passed a DataSet, DataTable, or collection of DataRows. This data is enumerated and for each inserted, modified, or deleted row, the

`InsertCommand`, `UpdateCommand`, or `DeleteCommand` is executed. With the DB-Direct pattern, the TableAdapter is instead passed the values of the columns necessary for inserting, updating, or deleting a single record. The DB Direct pattern method then uses those passed-in values to execute the appropriate `InsertCommand`, `UpdateCommand`, or `DeleteCommand` statement.

Regardless of the update pattern used, the TableAdapters auto-generated methods do not use transactions. By default each insert, update, or delete performed by the TableAdapter is treated as a single discrete operation. For instance, imagine that the DB-Direct pattern is used by some code in the BLL to insert ten records into the database. This code would call the TableAdapter s `Insert` method ten times. If the first five inserts succeed, but the sixth one resulted in an exception, the first five inserted records would remain in the database. Similarly, if the Batch Update pattern is used to perform inserts, updates, and deletes to the inserted, modified, and deleted rows in a DataTable, if the first several modifications succeeded but a later one encountered an error, those earlier modifications that completed would remain in the database.

In certain scenarios we want to ensure atomicity across a series of modifications. To accomplish this we must manually extend the TableAdapter by adding new methods that execute the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` s under the umbrella of a transaction. In Creating a Data Access Layer we looked at using partial classes to extend the functionality of the DataTables within the Typed DataSet. This technique can also be used with TableAdapters.

The Typed DataSet `Northwind.xsd` is located in the `App_Code` folder s `DAL` subfolder. Create a subfolder in the `DAL` folder named `TransactionSupport` and add a new class file named `ProductsTableAdapter.TransactionSupport.cs` (see Figure 4). This file will hold the partial implementation of the `ProductsTableAdapter` that includes methods for performing data modifications using a transaction.

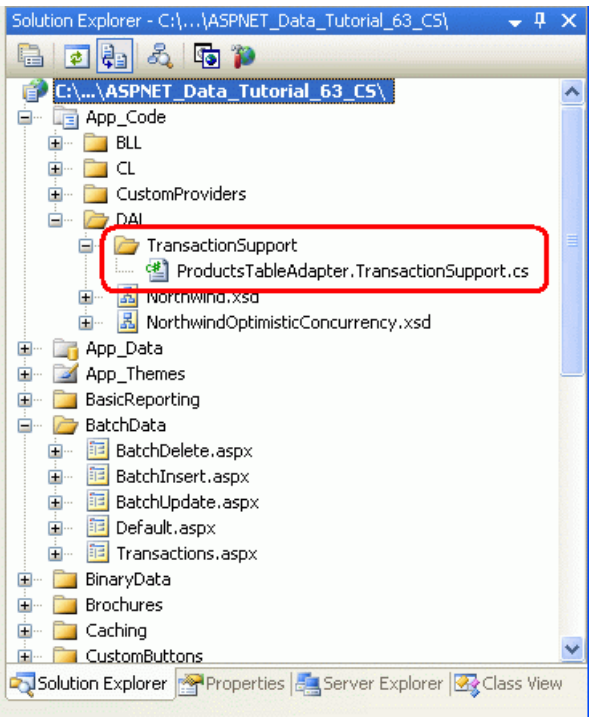


Figure 4: Add a Folder Named `TransactionSupport` and a Class File Named `ProductsTableAdapter.TransactionSupport.cs`

Enter the following code into the `ProductsTableAdapter.TransactionSupport.cs` file:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
namespace NorthwindTableAdapters
{
    public partial class ProductsTableAdapter
    {
        private SqlTransaction _transaction;
        private SqlTransaction Transaction
        {
            get
            {
                return this._transaction;
            }
            set
            {
                this._transaction = value;
            }
        }
        public void BeginTransaction()
        {
            // Open the connection, if needed
            if (this.Connection.State != ConnectionState.Open)
                this.Connection.Open();
            // Create the transaction and assign it to the Transaction property
            this.Transaction = this.Connection.BeginTransaction();
            // Attach the transaction to the Adapters
            foreach (SqlCommand command in this.CommandCollection)
            {
                command.Transaction = this.Transaction;
            }
            this.Adapter.InsertCommand.Transaction = this.Transaction;
            this.Adapter.UpdateCommand.Transaction = this.Transaction;
            this.Adapter.DeleteCommand.Transaction = this.Transaction;
        }
        public void CommitTransaction()
        {
            // Commit the transaction
            this.Transaction.Commit();
            // Close the connection
            this.Connection.Close();
        }
        public void RollbackTransaction()
        {
            // Rollback the transaction
            this.Transaction.Rollback();
            // Close the connection
            this.Connection.Close();
        }
    }
}
```

The `partial` keyword in the class declaration here indicates to the compiler that the members added within are to be added to the `ProductsTableAdapter` class in the `NorthwindTableAdapters` namespace. Note the `using System.Data.SqlClient` statement at the top of the file. Since the TableAdapter was configured to use the SqlClient provider, internally it uses a `SqlDataAdapter` object to issue its commands to the database. Consequently, we need to use the `SqlTransaction` class to begin the transaction and then to commit it or roll it back. If you are using a data store other than Microsoft SQL Server, you'll need to use the appropriate provider.

These methods provide the building blocks needed to start, rollback, and commit a transaction. They are marked `public`, enabling them to be used from within the `ProductsTableAdapter`, from another class in the DAL, or from another layer in the architecture, such as the BLL. `BeginTransaction` opens the TableAdapter s internal `SqlConnection` (if needed), begins the transaction and assigns it to the `Transaction` property, and attaches the transaction to the internal `SqlDataAdapter` s `SqlCommand` objects. `CommitTransaction` and `RollbackTransaction` call the `Transaction` object s `Commit` and `Rollback` methods, respectively, before closing the internal `Connection` object.

Step 3: Adding Methods to Update and Delete Data Under the Umbrella of a Transaction

With these methods complete, we re ready to add methods to `ProductsDataTable` or the BLL that perform a series of commands under the umbrella of a transaction. The following method uses the Batch Update pattern to update a `ProductsDataTable` instance using a transaction. It starts a transaction by calling the `BeginTransaction` method and then uses a `try...catch` block to issue the data modification statements. If the call to the `Adapter` object s `Update` method results in an exception, execution will transfer to the `catch` block where the transaction will be rolled back and the exception re-thrown. Recall that the `Update` method implements the Batch Update pattern by enumerating the rows of the supplied `ProductsDataTable` and performing the necessary `InsertCommand`, `UpdateCommand`, and `DeleteCommand` s. If any one of these commands results in an error, the transaction is rolled back, undoing the previous modifications made during the transaction s lifetime. Should the `Update` statement complete without error, the transaction is committed in its entirety.

```
public int UpdateWithTransaction(Northwind.ProductsDataTable dataTable)
{
    this.BeginTransaction();
    try
    {
        // Perform the update on the DataTable
        int returnValue = this.Adapter.Update(dataTable);
        // If we reach here, no errors, so commit the transaction
        this.CommitTransaction();
        return returnValue;
    }
    catch
    {
        // If we reach here, there was an error, so rollback the transaction
        this.RollbackTransaction();
        throw;
    }
}
```

Add the `UpdateWithTransaction` method to the `ProductsTableAdapter` class through the partial class in `ProductsTableAdapter.TransactionSupport.cs`. Alternatively, this method could be added to the Business Logic Layer s `ProductsBLL` class with a few minor syntactical changes. Namely, the keyword this in `this.BeginTransaction()`, `this.CommitTransaction()`, and `this.RollbackTransaction()` would need to be replaced with `Adapter` (recall that `Adapter` is the name of a property in `ProductsBLL` of type `ProductsTableAdapter`).

The `UpdateWithTransaction` method uses the Batch Update pattern, but a series of DB-Direct calls can also be used within the scope of a transaction, as the following method shows. The `DeleteProductsWithTransaction` method accepts as input a `List<T>` of type `int`, which are the `ProductID` s to delete. The method initiates the transaction via a call to `BeginTransaction` and then, in the `try` block, iterates through the supplied list calling the DB-Direct pattern `Delete` method for each `ProductID` value. If any of the calls to `Delete` fails, control is transferred to the `catch` block where the transaction is rolled back and the exception re-thrown. If all calls to `Delete` succeed, then transaction is committed. Add this method to the `ProductsBLL` class.

```
public void DeleteProductsWithTransaction
    (System.Collections.Generic.List<int> productIDs)
{
    // Start the transaction
    Adapter.BeginTransaction();
    try
    {
        // Delete each product specified in the list
        foreach (int productID in productIDs)
        {
            Adapter.Delete(productID);
        }
        // Commit the transaction
        Adapter.CommitTransaction();
    }
    catch
    {
        // There was an error - rollback the transaction
        Adapter.RollbackTransaction();
        throw;
    }
}
```

Applying Transactions Across Multiple TableAdapters

The transaction-related code examined in this tutorial allows for multiple statements against the `ProductsTableAdapter` to be treated as an atomic operation. But what if multiple modifications to different database tables need to be performed atomically? For instance, when deleting a category, we might first want to reassign its current products to some other category. These two steps reassigning the products and deleting the category should be executed as an atomic operation. But the `ProductsTableAdapter` includes only methods for modifying the `Products` table and the `CategoriesTableAdapter` includes only methods for modifying the `Categories` table. So how can a transaction encompass both TableAdapters?

One option is to add a method to the `CategoriesTableAdapter` named `DeleteCategoryAndReassignProducts(categoryIDtoDelete, reassignToCategoryID)` and have that method call a stored procedure that both reassigns the products and deletes the category within the scope of a transaction defined within the stored procedure. We'll look at how to begin, commit, and rollback transactions in stored procedures in a future tutorial.

Another option is to create a helper class in the DAL that contains the `DeleteCategoryAndReassignProducts(categoryIDtoDelete, reassignToCategoryID)` method. This method would create an instance of the `CategoriesTableAdapter` and the `ProductsTableAdapter` and then set these two TableAdapters `Connection` properties to the same `SqlConnection` instance. At that point, either one of the two TableAdapters would initiate the transaction with a call to `BeginTransaction`. The TableAdapters methods for reassigning the products and deleting the category would be invoked in a `try...catch` block with the transaction committed or rolled back as needed.

Step 4: Adding the `UpdateWithTransaction` Method to the Business Logic Layer

In Step 3 we added an `UpdateWithTransaction` method to the `ProductsTableAdapter` in the DAL. We should add a corresponding method to the BLL. While the Presentation Layer could call directly down to the DAL to invoke the `UpdateWithTransaction` method, these tutorials have strived to define a layered architecture that insulates the DAL from the Presentation Layer. Therefore, it behooves us to continue this approach.

Open the `ProductsBLL` class file and add a method named `UpdateWithTransaction` that simply calls down to the

corresponding DAL method. There should now be two new methods in `ProductsBLL`: `UpdateWithTransaction`, which you just added, and `DeleteProductsWithTransaction`, which was added in Step 3.

```
public int UpdateWithTransaction(Northwind.ProductsDataTable products)
{
    return Adapter.UpdateWithTransaction(products);
}
public void DeleteProductsWithTransaction
    (System.Collections.Generic.List<int> productIDs)
{
    // Start the transaction
    Adapter.BeginTransaction();
    try
    {
        // Delete each product specified in the list
        foreach (int productID in productIDs)
            Adapter.Delete(productID);
        // Commit the transaction
        Adapter.CommitTransaction();
    }
    catch
    {
        // There was an error - rollback the transaction
        Adapter.RollbackTransaction();
        throw;
    }
}
```

NOTE

These methods do not include the `DataObjectMethodAttribute` attribute assigned to most other methods in the `ProductsBLL` class because we'll be invoking these methods directly from the ASP.NET pages code-behind classes. Recall that `DataObjectMethodAttribute` is used to flag what methods should appear in the ObjectDataSource s Configure Data Source wizard and under what tab (SELECT, UPDATE, INSERT, or DELETE). Since the GridView lacks any built-in support for batch editing or deleting, we'll have to invoke these methods programmatically rather than use the code-free declarative approach.

Step 5: Atomically Updating Database Data from the Presentation Layer

To illustrate the effect that the transaction has when updating a batch of records, let s create a user interface that lists all products in a GridView and includes a Button Web control that, when clicked, reassigns the products `CategoryID` values. In particular, the category reassignment will progress so that the first several products are assigned a valid `CategoryID` value while others are purposefully assigned a non-existent `CategoryID` value. If we attempt to update the database with a product whose `CategoryID` does not match an existing category s `CategoryID`, a foreign key constraint violation will occur and an exception will be raised. What we'll see in this example is that when using a transaction the exception raised from the foreign key constraint violation will cause the previous valid `CategoryID` changes to be rolled back. When not using a transaction, however, the modifications to the initial categories will remain.

Start by opening the `Transactions.aspx` page in the `BatchData` folder and drag a GridView from the Toolbox onto the Designer. Set its `ID` to `Products` and, from its smart tag, bind it to a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to pull its data from the `ProductsBLL` class s `GetProducts` method. This will be a read-only GridView, so set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) and click Finish.

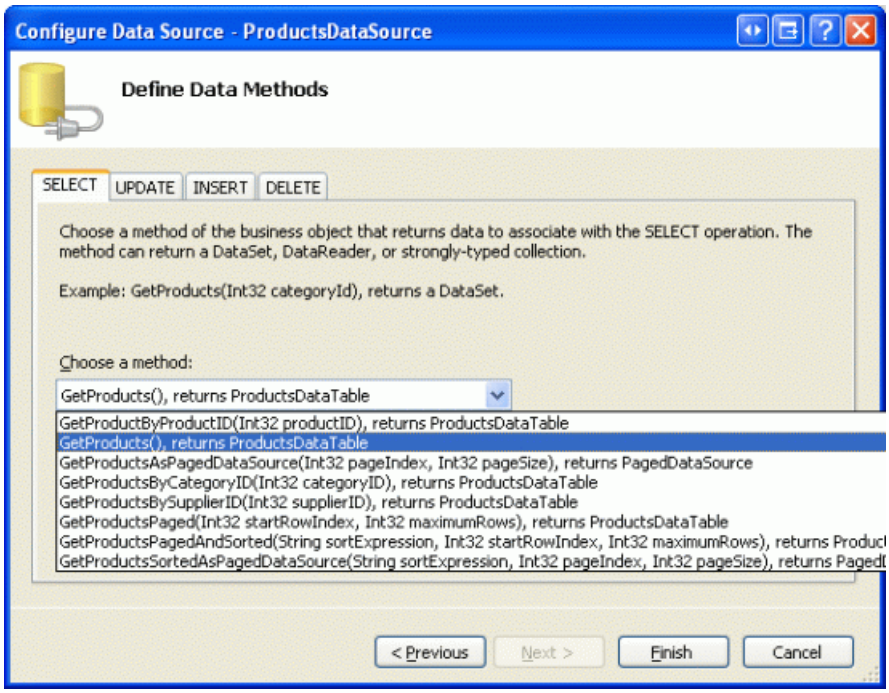


Figure 5: Figure 5: Configure the ObjectDataSource to Use the `ProductsBLL` Class s `GetProducts` Method (Click to view full-size image)

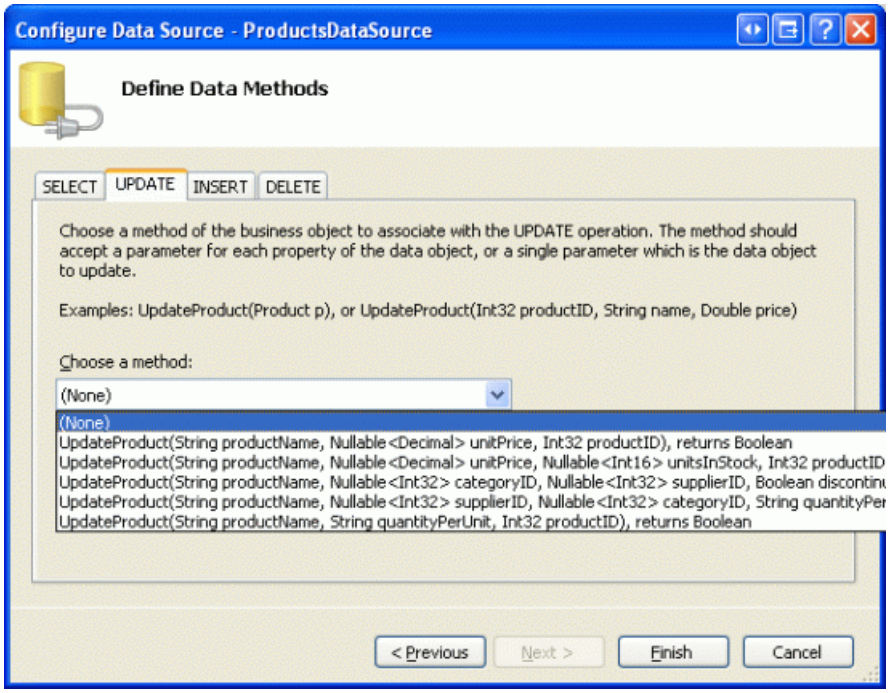


Figure 6: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE Tabs to (None) (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will create BoundFields and a CheckBoxField for the product data fields. Remove all of these fields except for `ProductID`, `ProductName`, `CategoryID`, and `CategoryName` and rename the `ProductName` and `CategoryName` BoundFields `HeaderText` properties to Product and Category, respectively. From the smart tag, check the Enable Paging option. After making these modifications, the GridView and ObjectDataSource s declarative markup should look like the following:

```
<asp:GridView ID="Products" runat="server" AllowPaging="True"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ProductsDataSource">
    <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="ProductID"
            InsertVisible="False" ReadOnly="True"
            SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryID" HeaderText="CategoryID"
            SortExpression="CategoryID" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            SortExpression="CategoryName" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
</asp:ObjectDataSource>
```

Next, add three Button Web controls above the GridView. Set the first Button s Text property to Refresh Grid, the second s to Modify Categories (WITH TRANSACTION), and the third one s to Modify Categories (WITHOUT TRANSACTION) .

```
<p>
    <asp:Button ID="RefreshGrid" runat="server" Text="Refresh Grid" />
</p>
<p>
    <asp:Button ID="ModifyCategoriesWithTransaction" runat="server"
        Text="Modify Categories (WITH TRANSACTION)" />
</p>
<p>
    <asp:Button ID="ModifyCategoriesWithoutTransaction" runat="server"
        Text="Modify Categories (WITHOUT TRANSACTION)" />
</p>
```

At this point the Design view in Visual Studio should look similar to the screen shot shown in Figure 7.

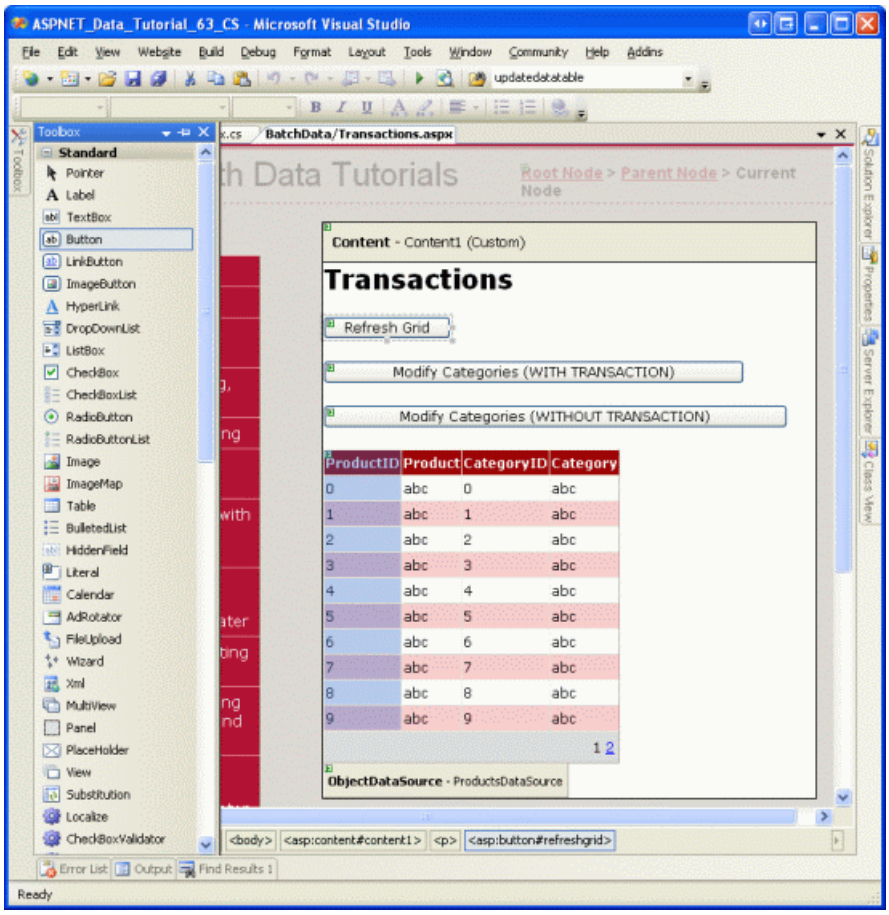


Figure 7: The Page Contains a GridView and Three Button Web Controls (Click to view full-size image)

Create event handlers for each of the three Button s `Click` events and use the following code:

```
protected void RefreshGrid_Click(object sender, EventArgs e)
{
    Products.DataBind();
}
protected void ModifyCategoriesWithTransaction_Click(object sender, EventArgs e)
{
    // Get the set of products
    ProductsBLL productsAPI = new ProductsBLL();
    Northwind.ProductsDataTable products = productsAPI.GetProducts();
    // Update each product's CategoryID
    foreach (Northwind.ProductsRow product in products)
    {
        product.CategoryID = product.ProductID;
    }
    // Update the data using a transaction
    productsAPI.UpdateWithTransaction(products);
    // Refresh the Grid
    Products.DataBind();
}
protected void ModifyCategoriesWithoutTransaction_Click(object sender, EventArgs e)
{
    // Get the set of products
    ProductsBLL productsAPI = new ProductsBLL();
    Northwind.ProductsDataTable products = productsAPI.GetProducts();
    // Update each product's CategoryID
    foreach (Northwind.ProductsRow product in products)
    {
        product.CategoryID = product.ProductID;
    }
    // Update the data WITHOUT using a transaction
    NorthwindTableAdapters.ProductsTableAdapter productsAdapter =
        new NorthwindTableAdapters.ProductsTableAdapter();
    productsAdapter.Update(products);
    // Refresh the Grid
    Products.DataBind();
}
```

The refresh Button s `Click` event handler simply rebinds the data to the GridView by calling the `Products` GridView s `DataBind` method.

The second event handler reassigns the products `CategoryID` s and uses the new transaction method from the BLL to perform the database updates under the umbrella of a transaction. Note that each product s `CategoryID` is arbitrarily set to the same value as its `ProductID`. This will work fine for the first few products, since those products have `ProductID` values that happen to map to valid `CategoryID` s. But once the `ProductID` s start getting too large, this coincidental overlap of `ProductID` s and `CategoryID` s no longer applies.

The third `Click` event handler updates the products `CategoryID` s in the same manner, but sends the update to the database using the `ProductsTableAdapter` s default `Update` method. This `Update` method does not wrap the series of commands within a transaction, so those changes are made prior to the first encountered foreign key constraint violation error will persist.

To demonstrate this behavior, visit this page through a browser. Initially you should see the first page of data as shown in Figure 8. Next, click the Modify Categories (WITH TRANSACTION) button. This will cause a postback and attempt to update all of the products `CategoryID` values, but will result in a foreign key constraint violation (see Figure 9).

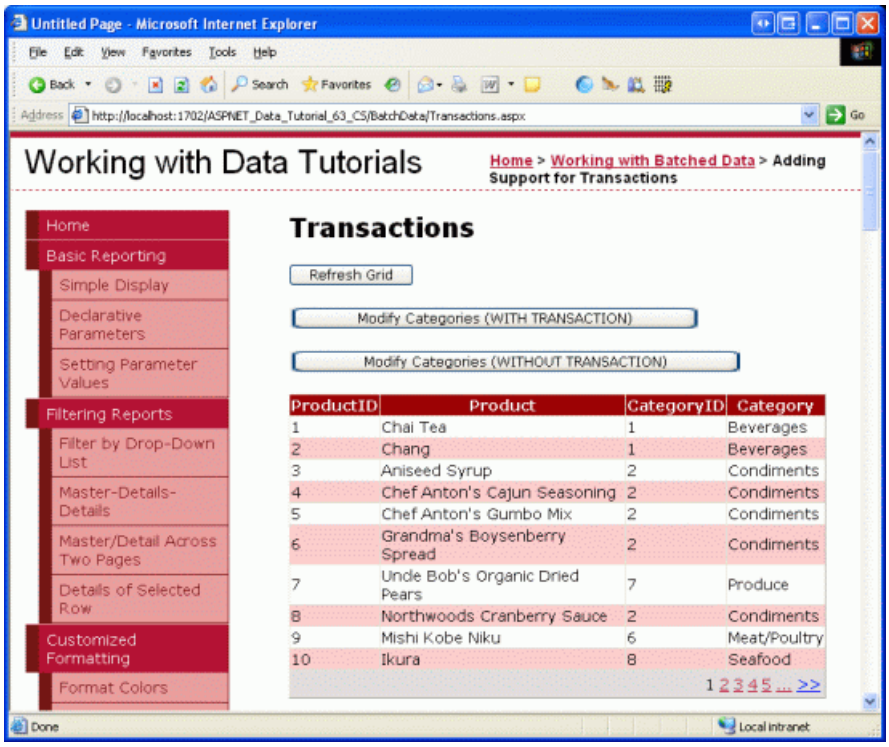


Figure 8: The Products are Displayed in a Pageable GridView (Click to view full-size image)

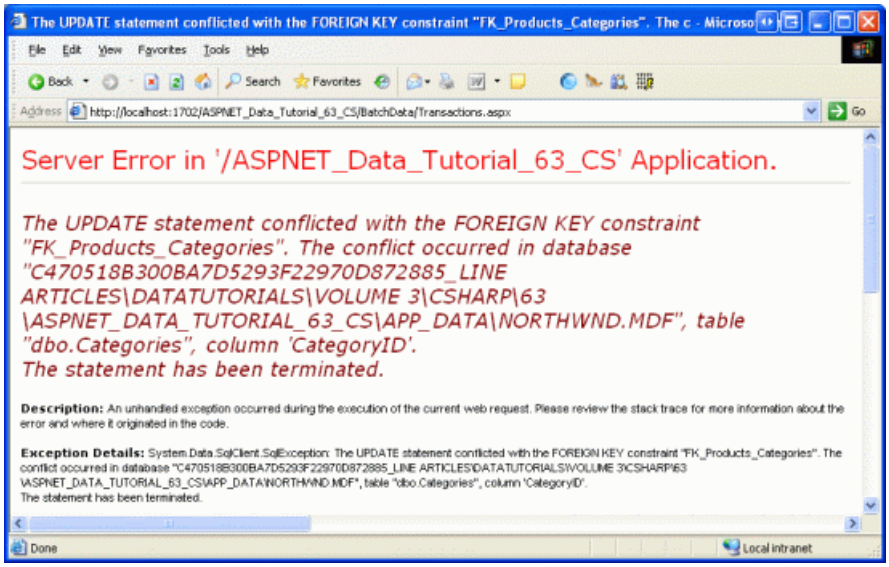


Figure 9: Reassigning the Categories Results in a Foreign Key Constraint Violation (Click to view full-size image)

Now hit your browser s Back button and then click the Refresh Grid button. Upon refreshing the data you should see the exact same output as shown in Figure 8. That is, even though some of the products `CategoryID` s were changed to legal values and updated in the database, they were rolled back when the foreign key constraint violation occurred.

Now try clicking the Modify Categories (WITHOUT TRANSACTION) button. This will result in the same foreign key constraint violation error (see Figure 9), but this time those products whose `CategoryID` values were changed to a legal value will not be rolled back. Hit your browser s Back button and then the Refresh Grid button. As Figure 10 shows, the `CategoryID` s of the first eight products have been reassigned. For example, in Figure 8, Chang had a `CategoryID` of 1, but in Figure 10 it s been reassigned to 2.

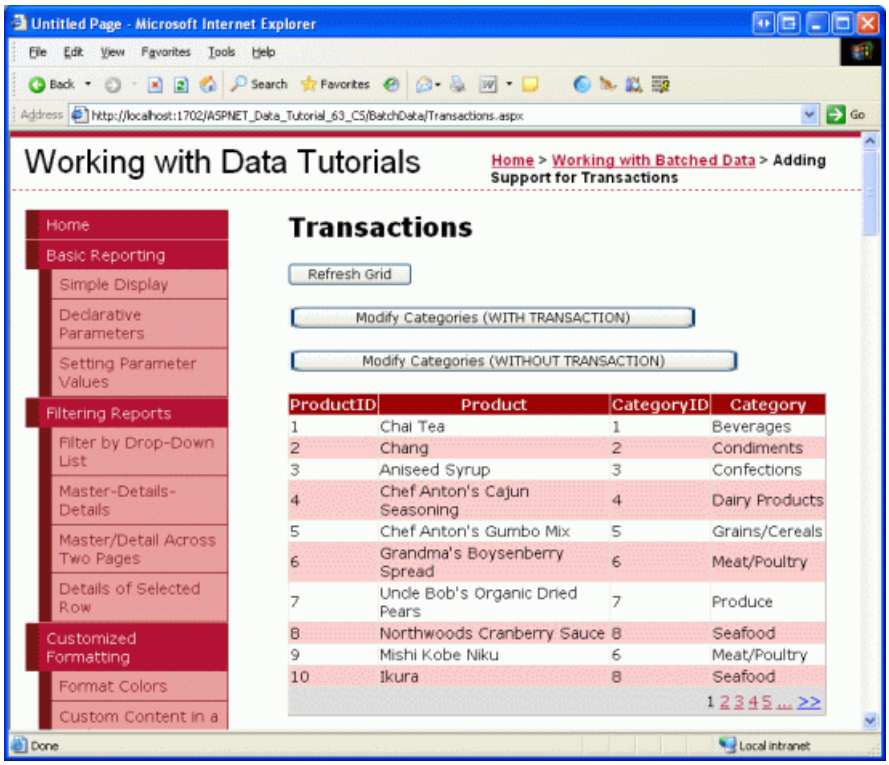


Figure 10: Some Products `CategoryID` Values were Updated While Others Were Not (Click to view full-size image)

Summary

By default, the TableAdapter s methods do not wrap the executed database statements within the scope of a transaction, but with a little work we can add methods that will create, commit, and rollback a transaction. In this tutorial we created three such methods in the `ProductsTableAdapter` class: `BeginTransaction`, `CommitTransaction`, and `RollbackTransaction`. We saw how to use these methods along with a `try...catch` block to make a series of data modification statements atomic. In particular, we created the `UpdateWithTransaction` method in the `ProductsTableAdapter`, which uses the Batch Update pattern to perform the necessary modifications to the rows of a supplied `ProductsDataTable`. We also added the `DeleteProductsWithTransaction` method to the `ProductsBLL` class in the BLL, which accepts a `List` of `ProductID` values as its input and calls the DB-Direct pattern method `Delete` for each `ProductID`. Both methods start by creating a transaction and then executing the data modification statements within a `try...catch` block. If an exception occurs, the transaction is rolled back, otherwise it is committed.

Step 5 illustrated the effect of transactional batch updates versus batch updates that neglected to use a transaction. In the next three tutorials we will build upon the foundation laid in this tutorial and create user interfaces for performing batch updates, deletes, and inserts.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Maintaining Database Consistency with Transactions
- Managing Transactions in SQL Server Stored Procedures
- Transactions Made Easy: `System.Transactions`
- TransactionScope and DataAdapters
- Using Oracle Database Transactions in .NET

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Dave Gardner, Hilton Giesenow, and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Batch Updating (C#)

2/20/2020 • 20 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Learn how to update multiple database records in a single operation. In the User Interface Layer we build a GridView where each row is editable. In the Data Access Layer we wrap the multiple Update operations within a transaction to ensure that all updates succeed or all updates are rolled back.

Introduction

In the preceding tutorial we saw how to extend the Data Access Layer to add support for database transactions. Database transactions guarantee that a series of data modification statements will be treated as one atomic operation, which ensures that all modifications will fail or all will succeed. With this low-level DAL functionality out of the way, we re ready to turn our attention to creating batch data modification interfaces.

In this tutorial we'll build a GridView where each row is editable (see Figure 1). Since each row is rendered in its editing interface, there s no need for a column of Edit, Update, and Cancel buttons. Instead, there are two Update Products buttons on the page that, when clicked, enumerate the GridView rows and update the database.

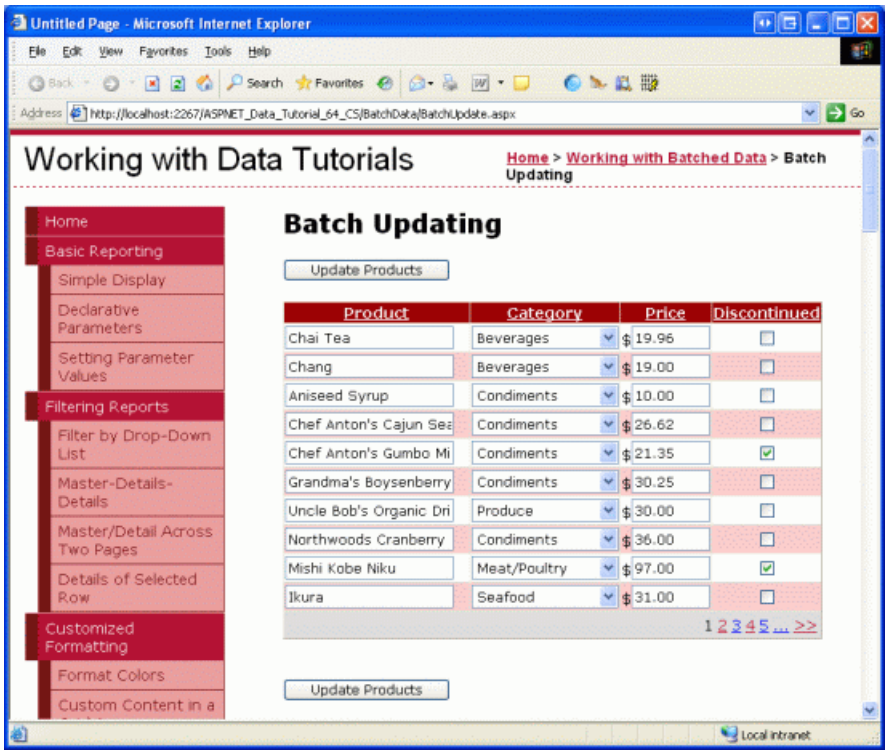


Figure 1: Each Row in the GridView is Editable (Click to view full-size image)

Let s get started!

> **NOTE**
> In the Performing Batch Updates tutorial we created a batch editing interface using the DataList control. This tutorial differs from the previous one in that is uses a GridView and the batch update is performed within the scope of a transaction. After completing this tutorial I encourage you to return to the earlier tutorial and update it to use the database transaction-related functionality added in the preceding tutorial.

Examining the Steps for Making All GridView Rows Editable

As discussed in the An Overview of Inserting, Updating, and Deleting Data tutorial, the GridView offers built-in support for editing its underlying data on a per-row basis. Internally, the GridView notes what row is editable through its `EditIndex` property. As the GridView is being bound to its data source, it checks each row to see if the index of the row equals the value of `EditIndex`. If so, that row s fields are rendered using their editing interfaces. For BoundFields, the editing interface is a TextBox whose `Text` property is assigned the value of the data field specified by the BoundField s `DataField` property. For TemplateFields, the `EditItemTemplate` is used in place of the `ItemTemplate`.

Recall that the editing workflow starts when a user clicks a row s Edit button. This causes a postback, sets the GridView s `EditIndex` property to the clicked row s index, and rebinds the data to the grid. When a row s Cancel button is clicked, on postback the `EditIndex` is set to a value of `-1` before rebinding the data to the grid. Since the GridView s rows start indexing at zero, setting `EditIndex` to `-1` has the effect of displaying the GridView in read-only mode.

The `EditIndex` property works well for per-row editing, but is not designed for batch editing. To make the entire GridView editable, we need to have each row render using its editing interface. The easiest way to accomplish this is to create where each editable field is implemented as a TemplateField with its editing interface defined in the `ItemTemplate`.

Over the next several steps we'll create a completely editable GridView. In Step 1 we'll start by creating the GridView and its ObjectDataSource and convert its BoundFields and CheckBoxField into TemplateFields. In Steps 2 and 3 we'll move the editing interfaces from the TemplateFields `EditItemTemplate` s to their `ItemTemplate` s.

Step 1: Displaying Product Information

Before we worry about creating a GridView where are rows are editable, let s start by simply displaying the product information. Open the `BatchUpdate.aspx` page in the `BatchData` folder and drag a GridView from the Toolbox onto the Designer. Set the GridView s `ID` to `ProductsGrid` and, from its smart tag, choose to bind it to a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to retrieve its data from the `ProductsBLL` class s `GetProducts` method.

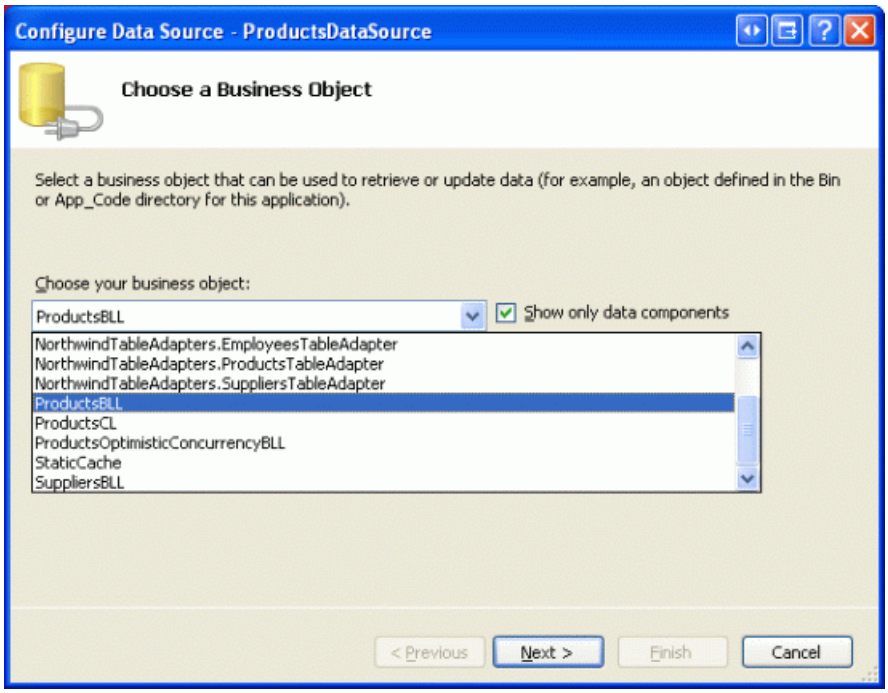


Figure 2: Configure the ObjectDataSource to Use the `ProductsBLL` Class (Click to view full-size image)

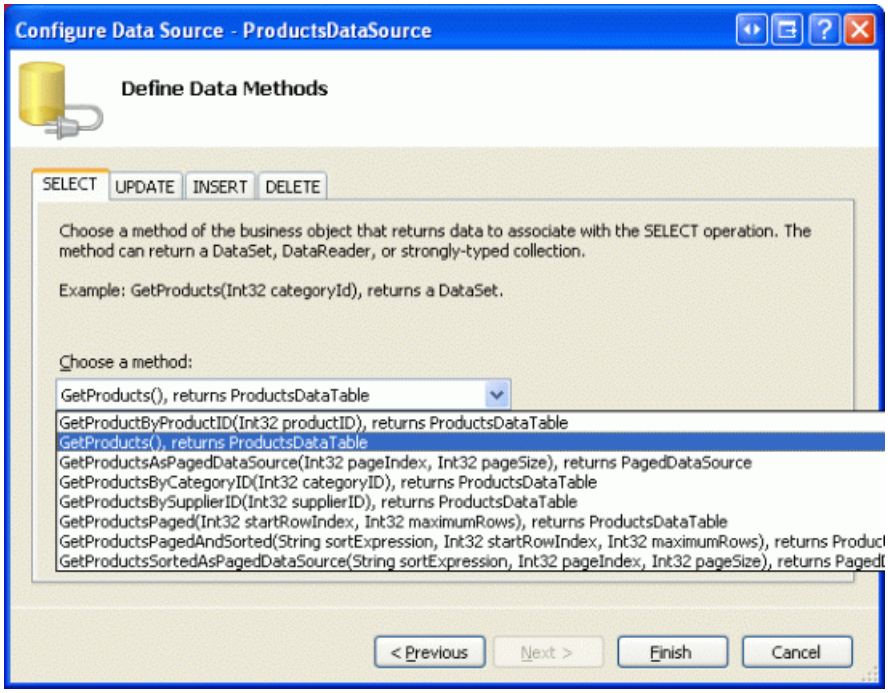


Figure 3: Retrieve the Product Data Using the `GetProducts` Method (Click to view full-size image)

Like the GridView, the ObjectDataSource s modification features are designed to work on a per-row basis. In order to update a set of records, we'll need to write a bit of code in the ASP.NET page s code-behind class that batches the data and passes it to the BLL. Therefore, set the drop-down lists in the ObjectDataSource s UPDATE, INSERT, and DELETE tabs to (None). Click Finish to complete the wizard.

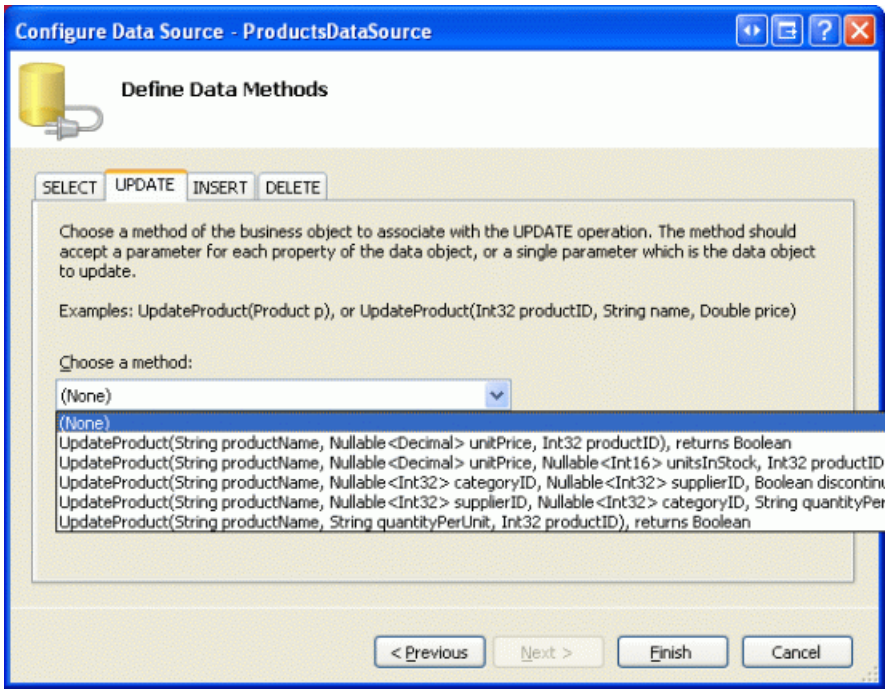


Figure 4: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE Tabs to (None) (Click to view full-size image)

After completing the Configure Data Source wizard, the ObjectDataSource s declarative markup should look like the following:

```
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
</asp:ObjectDataSource>
```

Completing the Configure Data Source wizard also causes Visual Studio to create BoundFields and a CheckBoxField for the product data fields in the GridView. For this tutorial, let s only allow the user to view and edit the product s name, category, price, and discontinued status. Remove all but the `ProductName`, `CategoryName`, `UnitPrice`, and `Discontinued` fields and rename the `HeaderText` properties of the first three fields to Product, Category, and Price, respectively. Lastly, check the Enable Paging and Enable Sorting checkboxes in the GridView s smart tag.

At this point the GridView has three BoundFields (`ProductName`, `CategoryName`, and `UnitPrice`) and a CheckBoxField (`Discontinued`). We need to convert these four fields into TemplateFields and then move the editing interface from the TemplateField s `EditItemTemplate` to its `ItemTemplate`.

> **NOTE**
>
> We explored creating and customizing TemplateFields in the Customizing the Data Modification Interface tutorial. We'll walk through the steps of converting the BoundFields and CheckBoxField into TemplateFields and defining their editing interfaces in their `ItemTemplate` s, but if you get stuck or need a refresher, don t hesitate to refer back to this earlier tutorial.

From the GridView s smart tag, click the Edit Columns link to open the Fields dialog box. Next, select each field and click the Convert this field into a TemplateField link.

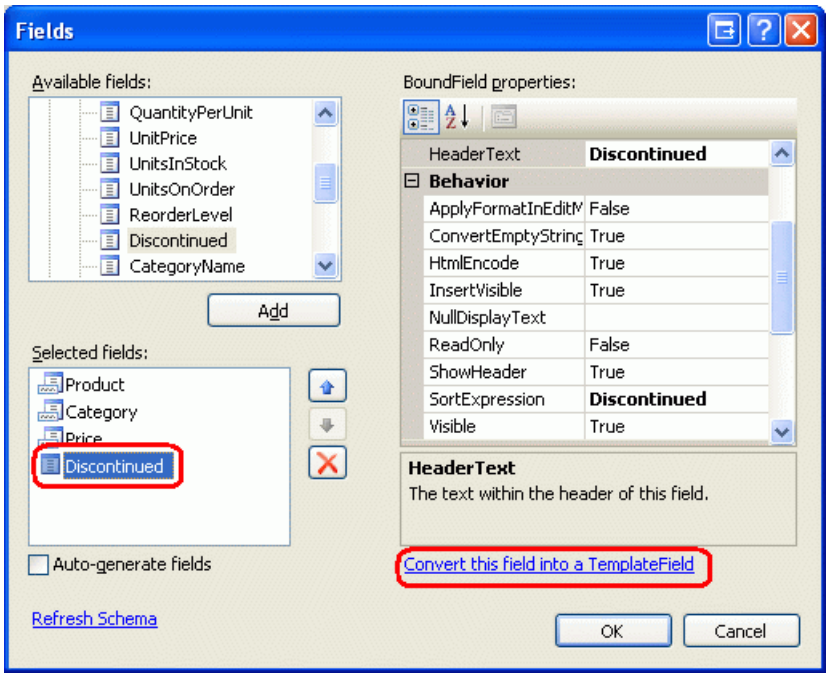


Figure 5: Convert the Existing BoundFields and CheckBoxField Into TemplateField

Now that each field is a TemplateField, we re ready to move the editing interface from the `EditItemTemplate` s to the `ItemTemplate` s.

Step 2: Creating the `ProductName` , `UnitPrice` , and `Discontinued` Editing Interfaces

Creating the `ProductName` , `UnitPrice` , and `Discontinued` editing interfaces are the topic of this step and are pretty straightforward, as each interface is already defined in the TemplateField s `EditItemTemplate` . Creating the `CategoryName` editing interface is a bit more involved since we need to create a DropDownList of the applicable categories. This `CategoryName` editing interface is tackled in Step 3.

Let s start with the `ProductName` TemplateField. Click on the Edit Templates link from the GridView s smart tag and drill down to the `ProductName` TemplateField s `EditItemTemplate` . Select the TextBox, copy it to the clipboard, and then paste it to the `ProductName` TemplateField s `ItemTemplate` . Change the TextBox s `ID` property to `ProductName` .

Next, add a RequiredFieldValidator to the `ItemTemplate` to ensure that the user provides a value for each product s name. Set the `ControlToValidate` property to ProductName, the `ErrorMessage` property to You must provide the product's name. and the `Text` property to *. After making these additions to the `ItemTemplate` , your screen should look similar to Figure 6.

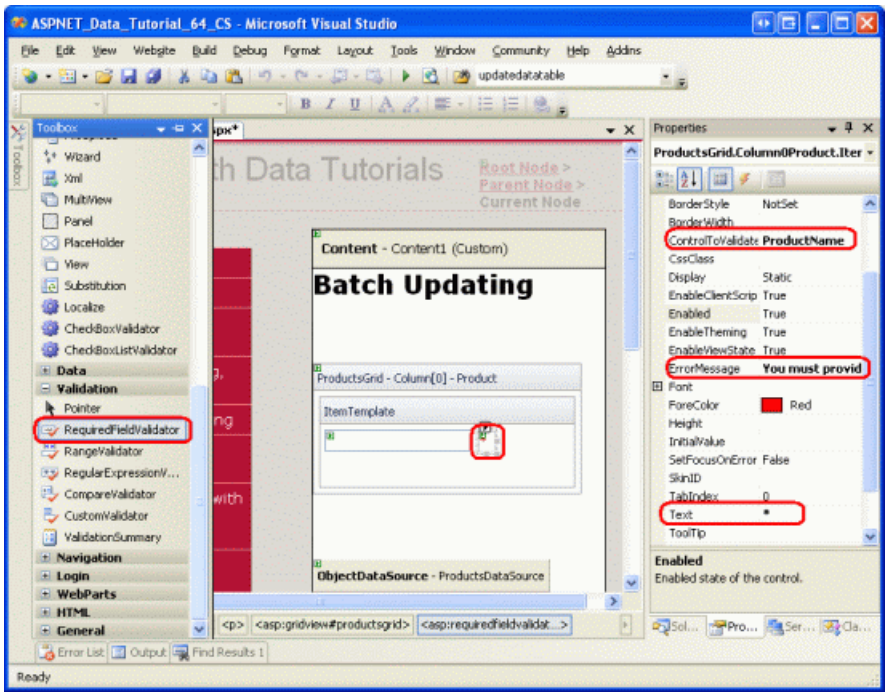


Figure 6: The `ProductName` TemplateField Now Includes a TextBox and a RequiredFieldValidator (Click to view full-size image)

For the `UnitPrice` editing interface, start by copying the TextBox from the `EditItemTemplate` to the `ItemTemplate`. Next, place a $ in front of the TextBox and set its `ID` property to UnitPrice and its `Columns` property to 8 .

Also add a CompareValidator to the `UnitPrice` s `ItemTemplate` to ensure that the value entered by the user is a valid currency value greater than or equal to $0.00. Set the validator s `ControlToValidate` property to UnitPrice, its `ErrorMessage` property to You must enter a valid currency value. Please omit any currency symbols., its `Text` property to *, its `Type` property to `Currency`, its `Operator` property to `GreaterThanEqual`, and its `ValueToCompare` property to 0 .

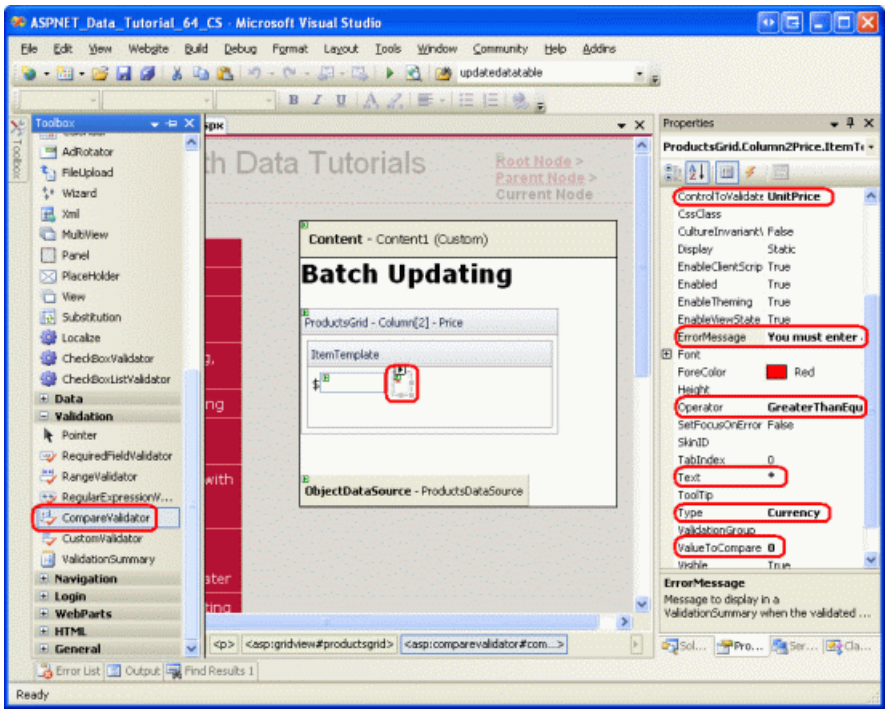


Figure 7: Add a CompareValidator to Ensure the Price Entered is a Non-Negative Currency Value (Click to view full-size image)

For the `Discontinued` TemplateField you can use the CheckBox already defined in the `ItemTemplate`. Simply set its `ID` to Discontinued and its `Enabled` property to `true`.

Step 3: Creating the `CategoryName` Editing Interface

The editing interface in the `CategoryName` TemplateField s `EditItemTemplate` contains a TextBox that displays the value of the `CategoryName` data field. We need to replace this with a DropDownList that lists the possible categories.

> **NOTE**
>
> The Customizing the Data Modification Interface tutorial contains a more thorough and complete discussion on customizing a template to include a DropDownList as opposed to a TextBox. While the steps here are complete, they are presented tersely. For a more in-depth look at creating and configuring the categories DropDownList, refer back to the Customizing the Data Modification Interface tutorial.

Drag a DropDownList from the Toolbox onto the `CategoryName` TemplateField s `ItemTemplate`, setting its `ID` to `Categories`. At this point we would usually define the DropDownLists s data source through its smart tag, creating a new ObjectDataSource. However, this will add the ObjectDataSource within the `ItemTemplate`, which will result in an ObjectDataSource instance created for each GridView row. Instead, let s create the ObjectDataSource outside of the GridView s TemplateFields. End the template editing and drag an ObjectDataSource from the Toolbox onto the Designer beneath the `ProductsDataSource` ObjectDataSource. Name the new ObjectDataSource `CategoriesDataSource` and configure it to use the `CategoriesBLL` class s `GetCategories` method.

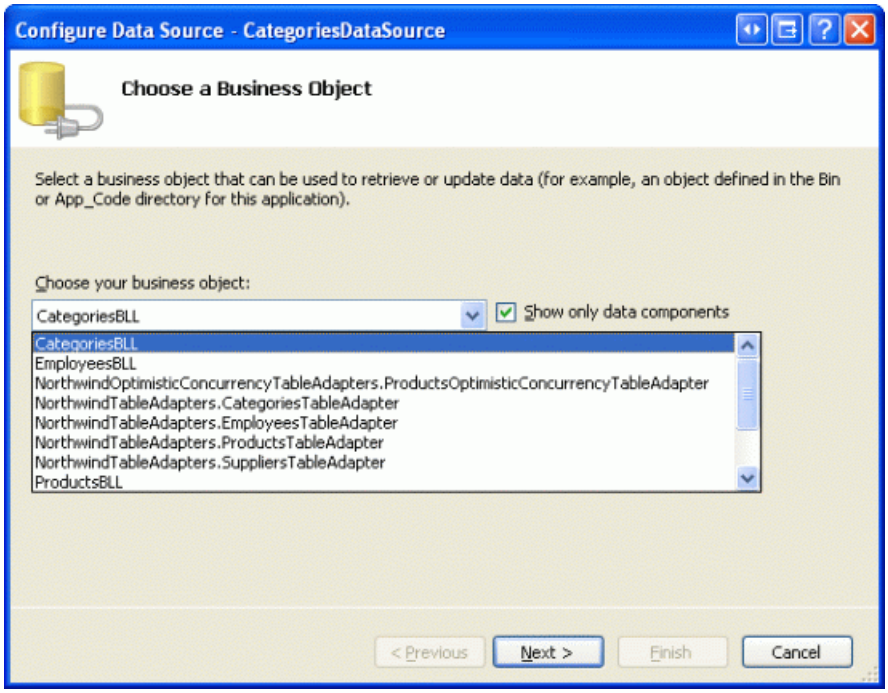


Figure 8: Configure the ObjectDataSource to Use the `CategoriesBLL` Class (Click to view full-size image)

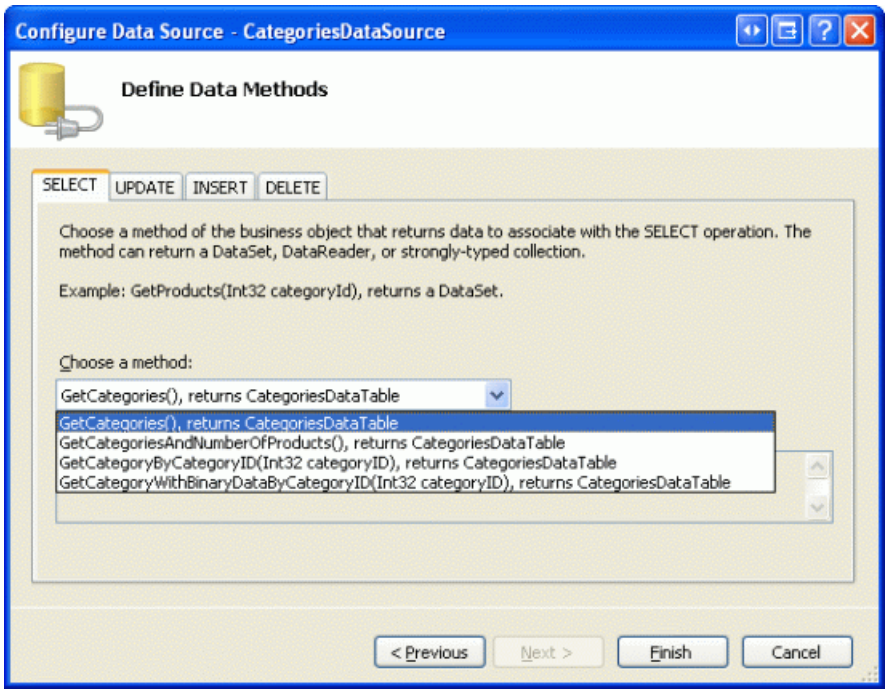


Figure 9: Retrieve the Category Data Using the `GetCategories` Method (Click to view full-size image)

Since this ObjectDataSource is used merely to retrieve data, set the drop-down lists in the UPDATE and DELETE tabs to (None). Click Finish to complete the wizard.

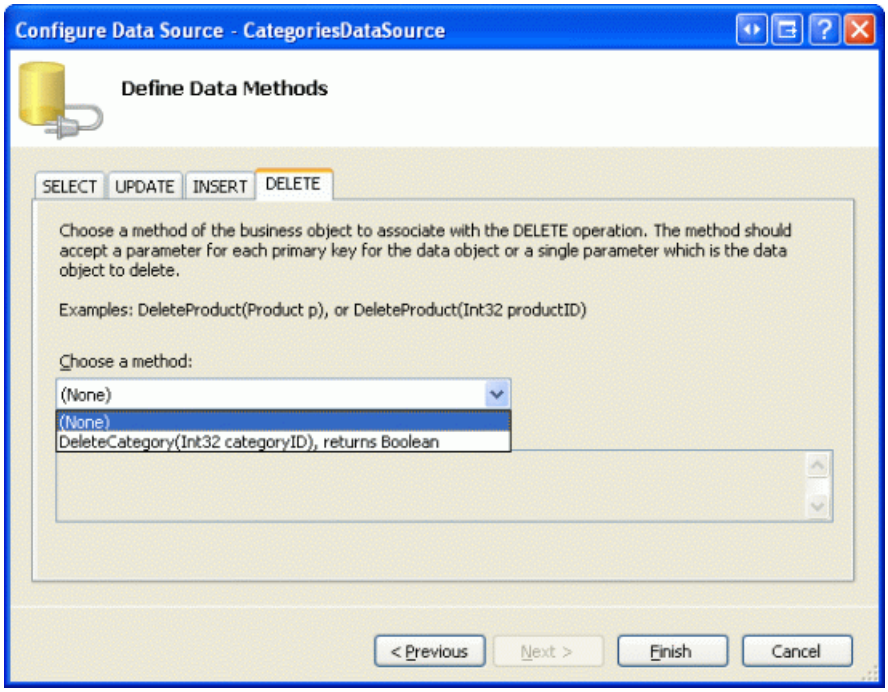


Figure 10: Set the Drop-Down Lists in the UPDATE and DELETE Tabs to (None) (Click to view full-size image)

After completing the wizard, the `CategoriesDataSource` s declarative markup should look like the following:

```
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

With the `CategoriesDataSource` created and configured, return to the `CategoryName` TemplateField s `ItemTemplate` and, from the DropDownList s smart tag, click on the Choose Data Source link. In the Data Source Configuration wizard, select the `CategoriesDataSource` option from the first drop-down list and choose to have `CategoryName` used for the display and `CategoryID` as the value.

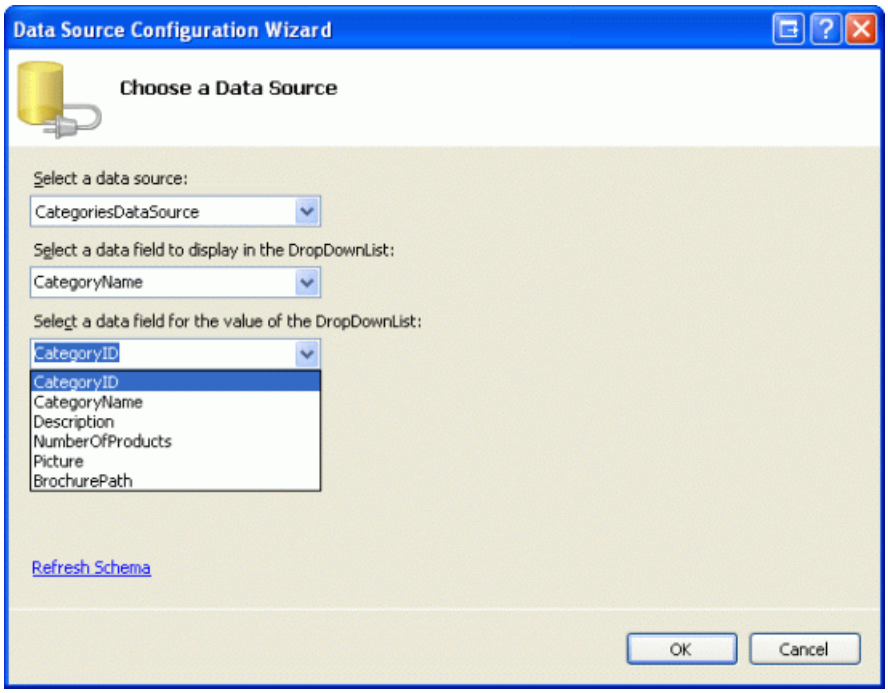


Figure 11: Bind the DropDownList to the `CategoriesDataSource` (Click to view full-size image)

At this point the `Categories` DropDownList lists all of the categories, but it does not yet automatically select the appropriate category for the product bound to the GridView row. To accomplish this we need to set the `Categories` DropDownList s `SelectedValue` to the product s `CategoryID` value. Click on the Edit DataBindings link from the DropDownList s smart tag and associate the `SelectedValue` property with the `CategoryID` data field as shown in Figure 12.

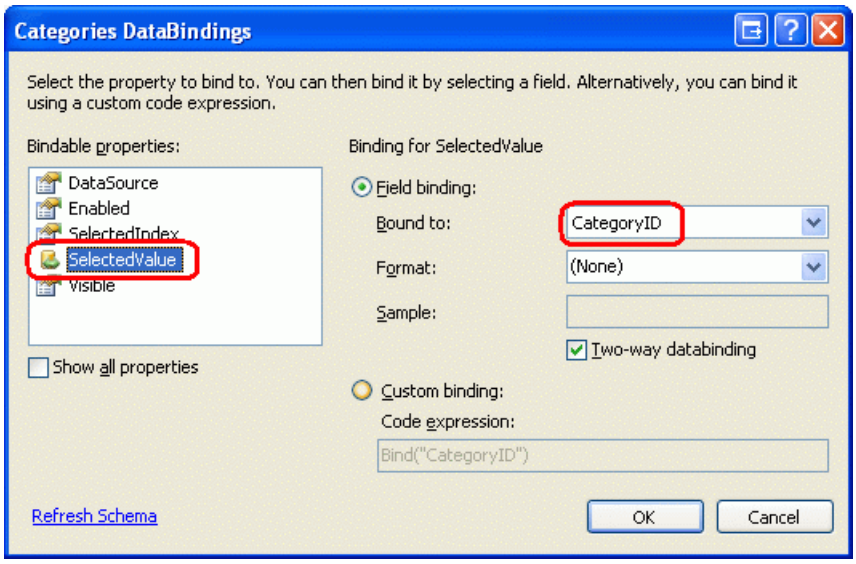


Figure 12: Bind the Product s `CategoryID` Value to the DropDownList s `SelectedValue` Property

One last problem remains: if the product doesn t have a `CategoryID` value specified then the databinding statement on `SelectedValue` will result in an exception. This is because the DropDownList contains only items for the categories and does not offer an option for those products that have a `NULL` database value for `CategoryID`. To remedy this, set the DropDownList s `AppendDataBoundItems` property to `true` and add a new item to the DropDownList, omitting the `Value` property from the declarative syntax. That is, make sure that the `Categories` DropDownList s declarative syntax looks like the following:

```
<asp:DropDownList ID="Categories" runat="server" AppendDataBoundItems="True"
    DataSourceID="CategoriesDataSource" DataTextField="CategoryName"
    DataValueField="CategoryID" SelectedValue='<%# Bind("CategoryID") %>'>
    <asp:ListItem Value="">-- Select One --</asp:ListItem>
</asp:DropDownList>
```

Note how the `<asp:ListItem Value="">` -- Select One -- has its `Value` attribute explicitly set to an empty string. Refer back to the Customizing the Data Modification Interface tutorial for a more thorough discussion on why this additional DropDownList item is needed to handle the `NULL` case and why assignment of the `Value` property to an empty string is essential.

> **NOTE**
>
> There is a potential performance and scalability issue here that is worth mentioning. Since each row has a DropDownList that uses the `CategoriesDataSource` as its data source, the `CategoriesBLL` class s `GetCategories` method will be called *n* times per page visit, where *n* is the number of rows in the GridView. These *n* calls to `GetCategories` result in *n* queries to the database. This impact on the database could be lessened by caching the returned categories either in a per-request cache or through the Caching Layer using a SQL caching dependency or a very short time-based expiry. For more information on the per-request caching option, see `HttpContext.Items` a Per-Request Cache Store.

Step 4: Completing the Editing Interface

We ve made a number of changes to the GridView s templates without pausing to view our progress. Take a moment to view our progress through a browser. As Figure 13 shows, each row is rendered using its `ItemTemplate`, which contains the cell s editing interface.

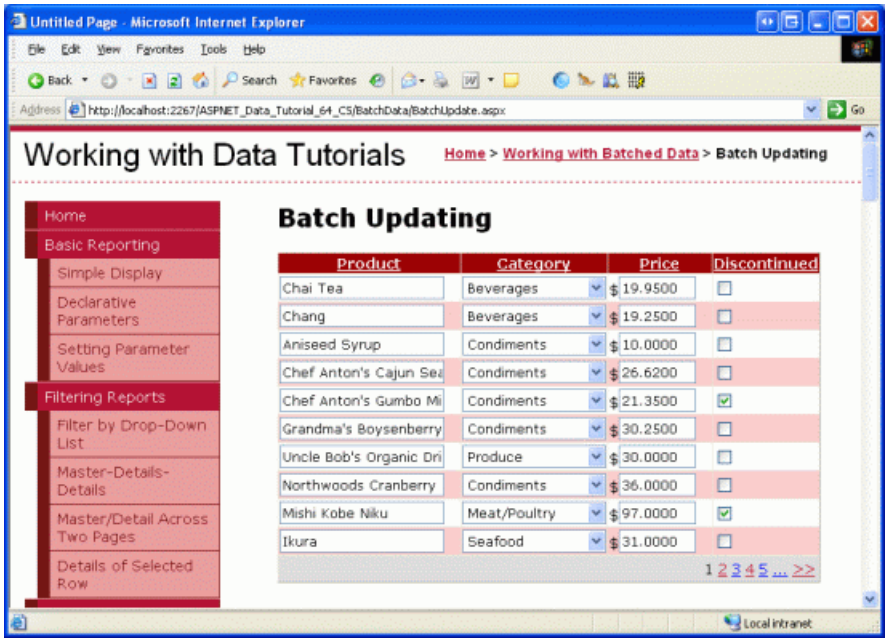


Figure 13: Each GridView Row is Editable (Click to view full-size image)

There are a few minor formatting issues that we should take care of at this point. First, note that the `UnitPrice` value contains four decimal points. To fix this, return to the `UnitPrice` TemplateField s `ItemTemplate` and, from the TextBox s smart tag, click on the Edit DataBindings link. Next, specify that the `Text` property should be formatted as a number.

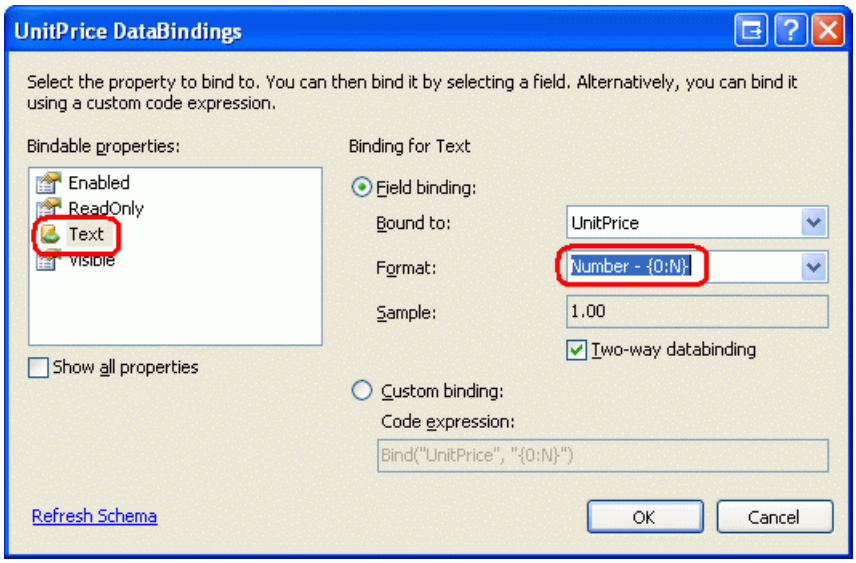


Figure 14: Format the `Text` Property as a Number

Second, let s center the checkbox in the `Discontinued` column (rather than having it left-aligned). Click on Edit Columns from the GridView s smart tag and select the `Discontinued` TemplateField from the list of fields in the bottom left corner. Drill down into `ItemStyle` and set the `HorizontalAlign` property to Center as shown in Figure 15.

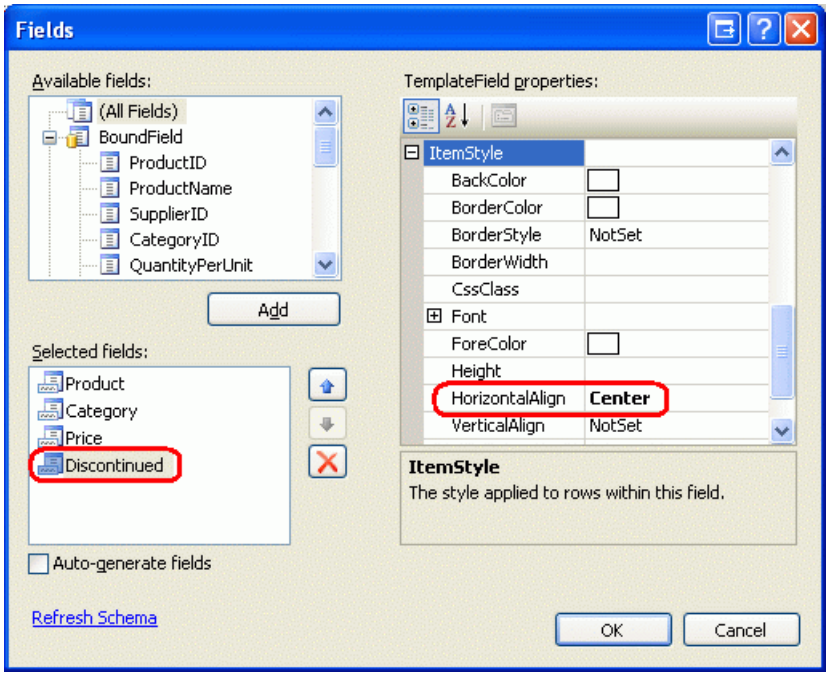


Figure 15: Center the `Discontinued` CheckBox

Next, add a ValidationSummary control to the page and set its `ShowMessageBox` property to `true` and its `ShowSummary` property to `false`. Also add the Button Web controls that, when clicked, will update the user s changes. Specifically, add two Button Web controls, one above the GridView and one below it, setting both controls `Text` properties to Update Products .

Since the GridView s editing interface is defined in its TemplateFields `ItemTemplate` s, the `EditItemTemplate` s are superfluous and may be deleted.

After making the above mentioned formatting changes, adding the Button controls, and removing the unnecessary `EditItemTemplate` s, your page s declarative syntax should look like the following:

```
<p>
    <asp:Button ID="UpdateAllProducts1" runat="server" Text="Update Products" />
```

```
</p>
<p>
    <asp:GridView ID="ProductsGrid" runat="server" AutoGenerateColumns="False"
        DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
        AllowPaging="True" AllowSorting="True">
        <Columns>
            <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
                <ItemTemplate>
                    <asp:TextBox ID="ProductName" runat="server"
                        Text='<%# Bind("ProductName") %>'></asp:TextBox>
                    <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                        ControlToValidate="ProductName"
                        ErrorMessage="You must provide the product's name."
                        runat="server">*</asp:RequiredFieldValidator>
                </ItemTemplate>
            </asp:TemplateField>
            <asp:TemplateField HeaderText="Category"
                SortExpression="CategoryName">
                <ItemTemplate>
                    <asp:DropDownList ID="Categories" runat="server"
                        AppendDataBoundItems="True"
                        DataSourceID="CategoriesDataSource"
                        DataTextField="CategoryName"
                        DataValueField="CategoryID"
                        SelectedValue='<%# Bind("CategoryID") %>'>
                        <asp:ListItem>-- Select One --</asp:ListItem>
                    </asp:DropDownList>
                </ItemTemplate>
            </asp:TemplateField>
            <asp:TemplateField HeaderText="Price"
                SortExpression="UnitPrice">
                <ItemTemplate>
                    $<asp:TextBox ID="UnitPrice" runat="server" Columns="8"
                        Text='<%# Bind("UnitPrice", "{0:N}") %>'></asp:TextBox>
                    <asp:CompareValidator ID="CompareValidator1" runat="server"
                        ControlToValidate="UnitPrice"
                        ErrorMessage="You must enter a valid currency value.
                                      Please omit any currency symbols."
                        Operator="GreaterThanEqual" Type="Currency"
                        ValueToCompare="0">*</asp:CompareValidator>
                </ItemTemplate>
            </asp:TemplateField>
            <asp:TemplateField HeaderText="Discontinued" SortExpression="Discontinued">
                <ItemTemplate>
                    <asp:CheckBox ID="Discontinued" runat="server"
                        Checked='<%# Bind("Discontinued") %>' />
                </ItemTemplate>
                <ItemStyle HorizontalAlign="Center" />
            </asp:TemplateField>
        </Columns>
    </asp:GridView>
</p>
<p>
    <asp:Button ID="UpdateAllProducts2" runat="server" Text="Update Products" />
    <asp:ObjectDataSource ID="ProductsDataSource" runat="server"
        OldValuesParameterFormatString="original_{0}"
        SelectMethod="GetProducts" TypeName="ProductsBLL">
    </asp:ObjectDataSource>
    <asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
        OldValuesParameterFormatString="original_{0}"
        SelectMethod="GetCategories" TypeName="CategoriesBLL">
    </asp:ObjectDataSource>
    <asp:ValidationSummary ID="ValidationSummary1" runat="server"
        ShowMessageBox="True" ShowSummary="False" />
</p>
```

Figure 16 shows this page when viewed through a browser after the Button Web controls have been added and

the formatting changes made.

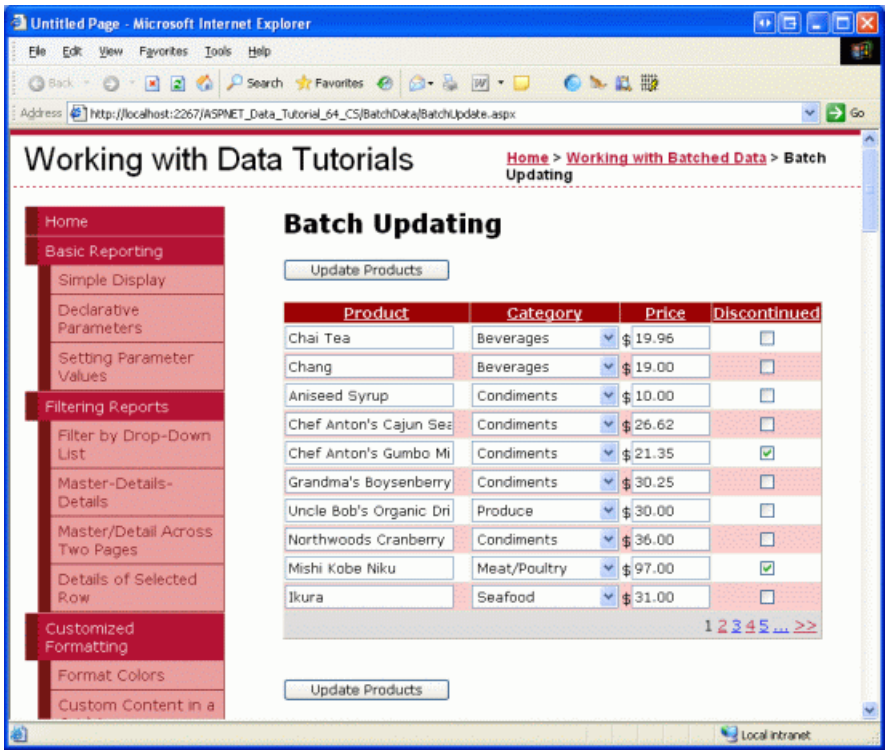


Figure 16: The Page Now Includes Two Update Products Buttons (Click to view full-size image)

Step 5: Updating the Products

When a user visits this page they will make their modifications and then click one of the two Update Products buttons. At that point we need to somehow save the user-entered values for each row into a `ProductsDataTable` instance and then pass that to a BLL method that will then pass that `ProductsDataTable` instance to the DAL s `UpdateWithTransaction` method. The `UpdateWithTransaction` method, which we created in the preceding tutorial, ensures that the batch of changes will be updated as an atomic operation.

Create a method named `BatchUpdate` in `BatchUpdate.aspx.cs` and add the following code:

```
private void BatchUpdate()
{
    // Enumerate the GridView's Rows collection and create a ProductRow
    ProductsBLL productsAPI = new ProductsBLL();
    Northwind.ProductsDataTable products = productsAPI.GetProducts();
    foreach (GridViewRow gvRow in ProductsGrid.Rows)
    {
        // Find the ProductsRow instance in products that maps to gvRow
        int productID = Convert.ToInt32(ProductsGrid.DataKeys[gvRow.RowIndex].Value);
        Northwind.ProductsRow product = products.FindByProductID(productID);
        if (product != null)
        {
            // Programmatically access the form field elements in the
            // current GridViewRow
            TextBox productName = (TextBox)gvRow.FindControl("ProductName");
            DropDownList categories =
                (DropDownList)gvRow.FindControl("Categories");
            TextBox unitPrice = (TextBox)gvRow.FindControl("UnitPrice");
            CheckBox discontinued =
                (CheckBox)gvRow.FindControl("Discontinued");
            // Assign the user-entered values to the current ProductRow
            product.ProductName = productName.Text.Trim();
            if (categories.SelectedIndex == 0)
                product.SetCategoryIDNull();
            else
                product.CategoryID = Convert.ToInt32(categories.SelectedValue);
            if (unitPrice.Text.Trim().Length == 0)
                product.SetUnitPriceNull();
            else
                product.UnitPrice = Convert.ToDecimal(unitPrice.Text);
            product.Discontinued = discontinued.Checked;
        }
    }
    // Now have the BLL update the products data using a transaction
    productsAPI.UpdateWithTransaction(products);
}
```

This method starts out by getting all of the products back in a `ProductsDataTable` via a call to the BLL s `GetProducts` method. It then enumerates the `ProductGrid` GridView s `Rows` collection. The `Rows` collection contains a `GridViewRow` instance for each row displayed in the GridView. Since we are showing at most ten rows per page, the GridView s `Rows` collection will have no more than ten items.

For each row the `ProductID` is grabbed from the `DataKeys` collection and the appropriate `ProductsRow` is selected from the `ProductsDataTable`. The four TemplateField input controls are programmatically referenced and their values assigned to the `ProductsRow` instance s properties. After each GridView row s values have been used to update the `ProductsDataTable`, it s passed to the BLL s `UpdateWithTransaction` method which, as we saw in the preceding tutorial, simply calls down into the DAL s `UpdateWithTransaction` method.

The batch update algorithm used for this tutorial updates each row in the `ProductsDataTable` that corresponds to a row in the GridView, regardless of whether the product s information has been changed. While such blind updates aren't usually a performance issue, they can lead to superfluous records if you re auditing changes to the database table. Back in the Performing Batch Updates tutorial we explored a batch updating interface with the DataList and added code that would only update those records that were actually modified by the user. Feel free to use the techniques from Performing Batch Updates to update the code in this tutorial, if desired.

> **NOTE**
> When binding the data source to the GridView through its smart tag, Visual Studio automatically assigns the data source s primary key value(s) to the GridView s `DataKeyNames` property. If you did not bind the ObjectDataSource to the GridView through the GridView s smart tag as outlined in Step 1, then you will need to manually set the GridView s `DataKeyNames` property to ProductID in order to access the `ProductID` value for each row through the `DataKeys` collection.

The code used in `BatchUpdate` is similar to that used in the BLL s `UpdateProduct` methods, the main difference being that in the `UpdateProduct` methods only a single `ProductRow` instance is retrieved from the architecture. The code that assigns the properties of the `ProductRow` is the same between the `UpdateProducts` methods and the code within the `foreach` loop in `BatchUpdate`, as is the overall pattern.

To complete this tutorial, we need to have the `BatchUpdate` method invoked when either of the Update Products buttons is clicked. Create event handlers for the `Click` events of these two Button controls and add the following code in the event handlers:

```
BatchUpdate();
ClientScript.RegisterStartupScript(this.GetType(), "message",
    "alert('The products have been updated.');", true);
```

First a call is made to `BatchUpdate`. Next, the `ClientScript property` is used to inject JavaScript that will display a messagebox that reads The products have been updated.

Take a minute to test out this code. Visit `BatchUpdate.aspx` through a browser, edit a number of rows, and click one of the Update Products buttons. Assuming there are no input validation errors, you should see a messagebox that reads The products have been updated. To verify the atomicity of the update, consider adding a random `CHECK` constraint, like one that disallows `UnitPrice` values of 1234.56. Then from `BatchUpdate.aspx`, edit a number of records, making sure to set one of the product s `UnitPrice` value to the forbidden value (1234.56). This should result in an error when clicking Update Products with the other changes during that batch operation rolled back to their original values.

An Alternative `BatchUpdate` Method

The `BatchUpdate` method we just examined retrieves *all* of the products from the BLL s `GetProducts` method and then updates just those records that appear in the GridView. This approach is ideal if the GridView does not use paging, but if it does, there may be hundreds, thousands, or tens of thousands of products, but only ten rows in the GridView. In such a case, getting all of the products from the database only to modify 10 of them is less than ideal.

For those types of situations, consider using the following `BatchUpdateAlternate` method instead:

```
private void BatchUpdateAlternate()
{
    // Enumerate the GridView's Rows collection and create a ProductRow
    ProductsBLL productsAPI = new ProductsBLL();
    Northwind.ProductsDataTable products = new Northwind.ProductsDataTable();
    foreach (GridViewRow gvRow in ProductsGrid.Rows)
    {
        // Create a new ProductRow instance
        int productID = Convert.ToInt32(ProductsGrid.DataKeys[gvRow.RowIndex].Value);

        Northwind.ProductsDataTable currentProductDataTable =
            productsAPI.GetProductByProductID(productID);
        if (currentProductDataTable.Rows.Count > 0)
        {
            Northwind.ProductsRow product = currentProductDataTable[0];
            // Programmatically access the form field elements in the
            // current GridViewRow
            TextBox productName = (TextBox)gvRow.FindControl("ProductName");
            DropDownList categories =
                (DropDownList)gvRow.FindControl("Categories");
            TextBox unitPrice = (TextBox)gvRow.FindControl("UnitPrice");
            CheckBox discontinued =
                (CheckBox)gvRow.FindControl("Discontinued");
            // Assign the user-entered values to the current ProductRow
            product.ProductName = productName.Text.Trim();
            if (categories.SelectedIndex == 0)
                product.SetCategoryIDNull();
            else
                product.CategoryID = Convert.ToInt32(categories.SelectedValue);
            if (unitPrice.Text.Trim().Length == 0)
                product.SetUnitPriceNull();
            else
                product.UnitPrice = Convert.ToDecimal(unitPrice.Text);
            product.Discontinued = discontinued.Checked;
            // Import the ProductRow into the products DataTable
            products.ImportRow(product);
        }
    }
    // Now have the BLL update the products data using a transaction
    productsAPI.UpdateProductsWithTransaction(products);
}
```

`BatchMethodAlternate` starts by creating a new empty `ProductsDataTable` named `products`. It then steps through the GridView s `Rows` collection and for each row gets the particular product information using the BLL s `GetProductByProductID(productID)` method. The retrieved `ProductsRow` instance has its properties updated in the same fashion as `BatchUpdate`, but after updating the row it is imported into the `products``ProductsDataTable` via the DataTable s `ImportRow(DataRow)` method.

After the `foreach` loop completes, `products` contains one `ProductsRow` instance for each row in the GridView. Since each of the `ProductsRow` instances have been added to the `products` (instead of updated), if we blindly pass it to the `UpdateWithTransaction` method the `ProductsTableAdapter` will try to insert each of the records into the database. Instead, we need to specify that each of these rows has been modified (not added).

This can be accomplished by adding a new method to the BLL named `UpdateProductsWithTransaction`. `UpdateProductsWithTransaction`, shown below, sets the `RowState` of each of the `ProductsRow` instances in the `ProductsDataTable` to `Modified` and then passes the `ProductsDataTable` to the DAL s `UpdateWithTransaction` method.

```
public int UpdateProductsWithTransaction(Northwind.ProductsDataTable products)
{
    // Mark each product as Modified
    products.AcceptChanges();
    foreach (Northwind.ProductsRow product in products)
        product.SetModified();
    // Update the data via a transaction
    return UpdateWithTransaction(products);
}
```

Summary

The GridView provides built-in per-row editing capabilities, but lacks support for creating fully editable interfaces. As we saw in this tutorial, such interfaces are possible, but require a bit of work. To create a GridView where every row is editable, we need to convert the GridView s fields into TemplateFields and define the editing interface within the `ItemTemplate` s. Additionally, Update All -type Button Web controls must be added to the page, separate from the GridView. These Buttons `Click` event handlers need to enumerate the GridView s `Rows` collection, store the changes in a `ProductsDataTable`, and pass the updated information into the appropriate BLL method.

In the next tutorial we'll see how to create an interface for batch deleting. In particular, each GridView row will include a checkbox and instead of Update All -type buttons, we'll have Delete Selected Rows buttons.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Teresa Murphy and David Suru. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Batch Deleting (C#)

2/20/2020 • 8 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Learn how to delete multiple database records in a single operation. In the User Interface Layer we build upon an enhanced GridView created in an earlier tutorial. In the Data Access Layer we wrap the multiple Delete operations within a transaction to ensure that all deletions succeed or all deletions are rolled back.

Introduction

The preceding tutorial explored how to create a batch editing interface using a fully-editable GridView. In situations where users are commonly editing many records at once, a batch editing interface will require far fewer postbacks and keyboard-to-mouse context switches, thereby improving the end user s efficiency. This technique is similarly useful for pages where it is common for users to delete many records in one go.

Anyone who has used an online email client is already familiar with one of the most common batch deleting interfaces: a checkbox in each row in a grid with a corresponding Delete All Checked Items button (see Figure 1). This tutorial is rather short because we ve already done all of the hard work in previous tutorials in creating both the web-based interface and a method to delete a series of records as a single atomic operation. In the Adding a GridView Column of Checkboxes tutorial we created a GridView with a column of checkboxes and in the Wrapping Database Modifications within a Transaction tutorial we created a method in the BLL that would use a transaction to delete a `List<T>` of `ProductID` values. In this tutorial, we will build upon and merge our previous experiences to create a working batch deleting example.

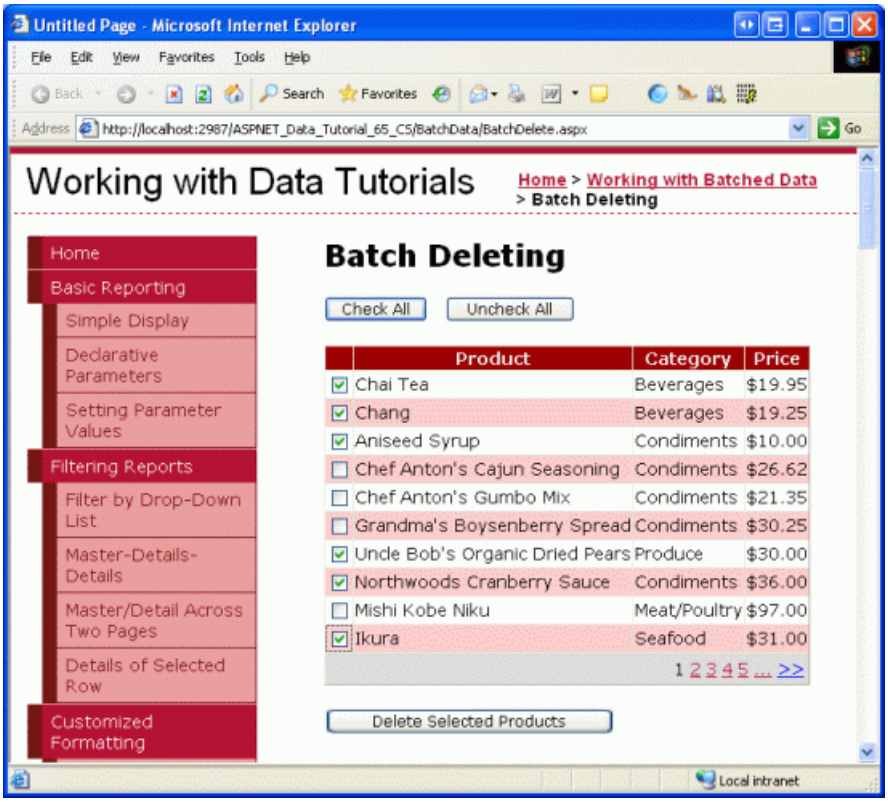


Figure 1: Each Row Includes a Checkbox (Click to view full-size image)

Step 1: Creating the Batch Deleting Interface

Since we already created the batch deleting interface in the Adding a GridView Column of Checkboxes tutorial, we can simply copy it to `BatchDelete.aspx` rather than creating it from scratch. Start by opening the `BatchDelete.aspx` page in the `BatchData` folder and the `CheckBoxField.aspx` page in the `EnhancedGridView` folder. From the `CheckBoxField.aspx` page, go to the Source view and copy the markup between the `<asp:Content>` tags as shown in Figure 2.

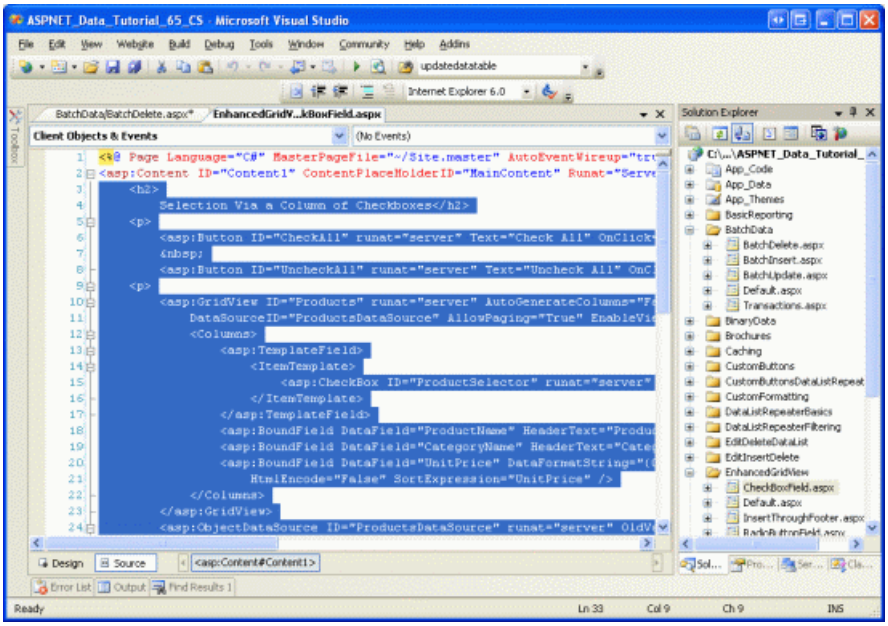


Figure 2: Copy the Declarative Markup of `CheckBoxField.aspx` to the Clipboard (Click to view full-size image)

Next, go to the Source view in `BatchDelete.aspx` and paste the contents of the clipboard within the `<asp:Content>` tags. Also copy and paste the code from within the code-behind class in `CheckBoxField.aspx.cs` to within the code-behind class in `BatchDelete.aspx.cs` (the `DeleteSelectedProducts` Button s `Click` event handler, the `ToggleCheckState` method, and the `Click` event handlers for the `CheckAll` and `UncheckAll` Buttons). After copying over this content, the `BatchDelete.aspx` page s code-behind class should contain the following code:

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
public partial class BatchData_BatchDelete : System.Web.UI.Page
{
    protected void DeleteSelectedProducts_Click(object sender, EventArgs e)
    {
        bool atLeastOneRowDeleted = false;
        // Iterate through the Products.Rows property
        foreach (GridViewRow row in Products.Rows)
        {
            // Access the CheckBox
            CheckBox cb = (CheckBox)row.FindControl("ProductSelector");
            if (cb != null && cb.Checked)
            {
                // Delete row! (Well, not really...)
                atLeastOneRowDeleted = true;
                // First, get the ProductID for the selected row
                int productID = Convert.ToInt32(Products.DataKeys[row.RowIndex].Value);
                // "Delete" the row
                DeleteResults.Text += string.Format
                    ("This would have deleted ProductID {0}<br />", productID);
                //... To actually delete the product, use ...
                //ProductsBLL productAPI = new ProductsBLL();
                //productAPI.DeleteProduct(productID);
                //.........................................
            }
        }
        // Show the Label if at least one row was deleted...
        DeleteResults.Visible = atLeastOneRowDeleted;
    }
    private void ToggleCheckState(bool checkState)
    {
        // Iterate through the Products.Rows property
        foreach (GridViewRow row in Products.Rows)
        {
            // Access the CheckBox
            CheckBox cb = (CheckBox)row.FindControl("ProductSelector");
            if (cb != null)
                cb.Checked = checkState;
        }
    }
    protected void CheckAll_Click(object sender, EventArgs e)
    {
        ToggleCheckState(true);
    }
    protected void UncheckAll_Click(object sender, EventArgs e)
    {
        ToggleCheckState(false);
    }
}
```

After copying over the declarative markup and source code, take a moment to test `BatchDelete.aspx` by viewing it through a browser. You should see a GridView listing the first ten products in a GridView with each row listing the product s name, category, and price along with a checkbox. There should be three buttons: Check All, Uncheck All, and Delete Selected Products. Clicking the Check All button selects all checkboxes, while Uncheck All clears all checkboxes. Clicking Delete Selected Products displays a message that lists the `ProductID` values of the selected products, but does not actually delete the products.

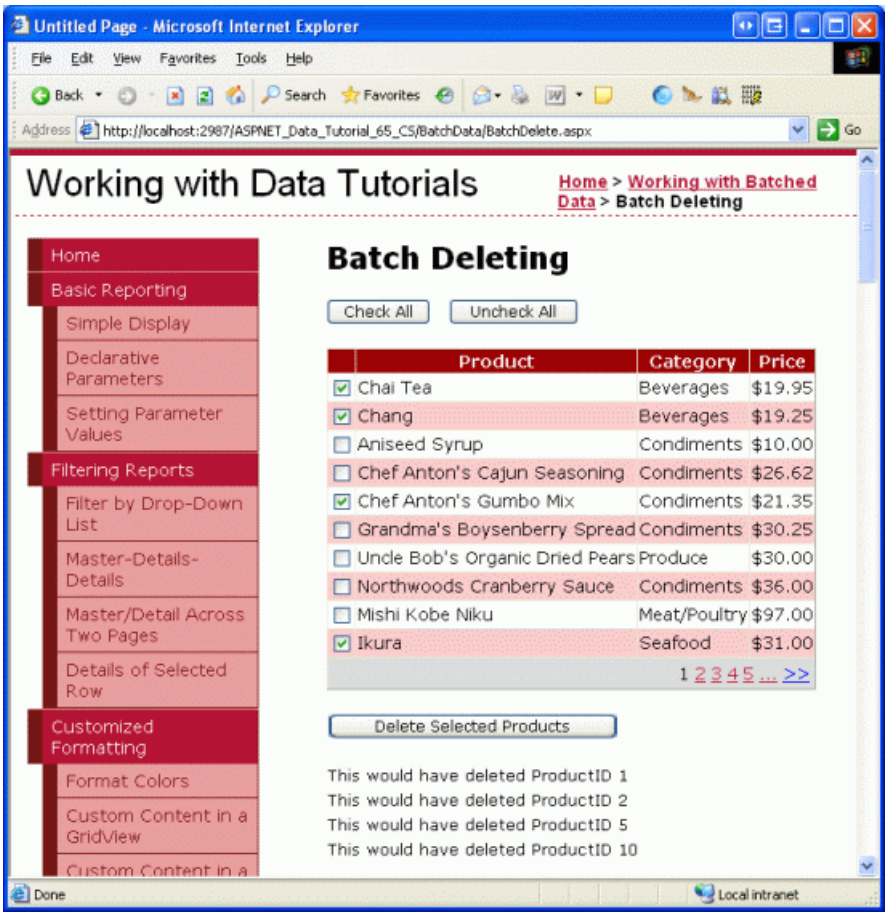


Figure 3: The Interface from `CheckBoxField.aspx` has been Moved to `BatchDeleting.aspx` (Click to view full-size image)

Step 2: Deleting the Checked Products Using Transactions

With the batch deleting interface successfully copied over to `BatchDeleting.aspx`, all that remains is to update the code so that the Delete Selected Products button deletes the checked products using the `DeleteProductsWithTransaction` method in the `ProductsBLL` class. This method, added in the Wrapping Database Modifications within a Transaction tutorial, accepts as its input a `List<T>` of `ProductID` values and deletes each corresponding `ProductID` within the scope of a transaction.

The `DeleteSelectedProducts` Button s `Click` event handler currently uses the following `foreach` loop to iterate through each GridView row:

```
// Iterate through the Products.Rows property
foreach (GridViewRow row in Products.Rows)
{
    // Access the CheckBox
    CheckBox cb = (CheckBox)row.FindControl("ProductSelector");
    if (cb != null && cb.Checked)
    {
        // Delete row! (Well, not really...)
        atLeastOneRowDeleted = true;
        // First, get the ProductID for the selected row
        int productID = Convert.ToInt32(Products.DataKeys[row.RowIndex].Value);
        // "Delete" the row
        DeleteResults.Text += string.Format
            ("This would have deleted ProductID {0}<br />", productID);
        //... To actually delete the product, use ...
        //ProductsBLL productAPI = new ProductsBLL();
        //productAPI.DeleteProduct(productID);
        //.........................................
    }
}
```

For each row, the `ProductSelector` CheckBox Web control is programmatically referenced. If it is checked, the row s `ProductID` is retrieved from the `DataKeys` collection and the `DeleteResults` Label s `Text` property is updated to include a message indicating that the row was selected for deletion.

The above code does not actually delete any records as the call to the `ProductsBLL` class s `Delete` method is commented out. Were this delete logic to be applied, the code would delete the products but not within an atomic operation. That is, if the first few deletes in the sequence succeeded, but a later one failed (perhaps due to a foreign key constraint violation), an exception would be thrown but those products already deleted would remain deleted.

In order to assure atomicity, we need to instead use the `ProductsBLL` class s `DeleteProductsWithTransaction` method. Because this method accepts a list of `ProductID` values, we need to first compile this list from the grid and then pass it as a parameter. We first create an instance of a `List<T>` of type `int`. Within the `foreach` loop we need to add the selected products `ProductID` values to this `List<T>`. After the loop this `List<T>` must be passed to the `ProductsBLL` class s `DeleteProductsWithTransaction` method. Update the `DeleteSelectedProducts` Button s `Click` event handler with the following code:

```
protected void DeleteSelectedProducts_Click(object sender, EventArgs e)
{
    // Create a List to hold the ProductID values to delete
    System.Collections.Generic.List<int> productIDsToDelete =
        new System.Collections.Generic.List<int>();
    // Iterate through the Products.Rows property
    foreach (GridViewRow row in Products.Rows)
    {
        // Access the CheckBox
        CheckBox cb = (CheckBox)row.FindControl("ProductSelector");
        if (cb != null && cb.Checked)
        {
            // Save the ProductID value for deletion
            // First, get the ProductID for the selected row
            int productID = Convert.ToInt32(Products.DataKeys[row.RowIndex].Value);
            // Add it to the List...
            productIDsToDelete.Add(productID);
            // Add a confirmation message
            DeleteResults.Text += string.Format
                ("ProductID {0} has been deleted<br />", productID);
        }
    }
    // Call the DeleteProductsWithTransaction method and show the Label
    // if at least one row was deleted...
    if (productIDsToDelete.Count > 0)
    {
        ProductsBLL productAPI = new ProductsBLL();
        productAPI.DeleteProductsWithTransaction(productIDsToDelete);
        DeleteResults.Visible = true;
        // Rebind the data to the GridView
        Products.DataBind();
    }
}
```

The updated code creates a `List<T>` of type `int` (`productIDsToDelete`) and populates it with the `ProductID` values to delete. After the `foreach` loop, if there is at least one product selected, the `ProductsBLL` class s `DeleteProductsWithTransaction` method is called and passed this list. The `DeleteResults` Label is also displayed and the data rebound to the GridView (so that the newly-deleted records no longer appear as rows in the grid).

Figure 4 shows the GridView after a number of rows have been selected for deletion. Figure 5 shows the screen immediately after the Delete Selected Products button has been clicked. Note that in Figure 5 the `ProductID` values of the deleted records are displayed in the Label beneath the GridView and those rows are no longer in the GridView.

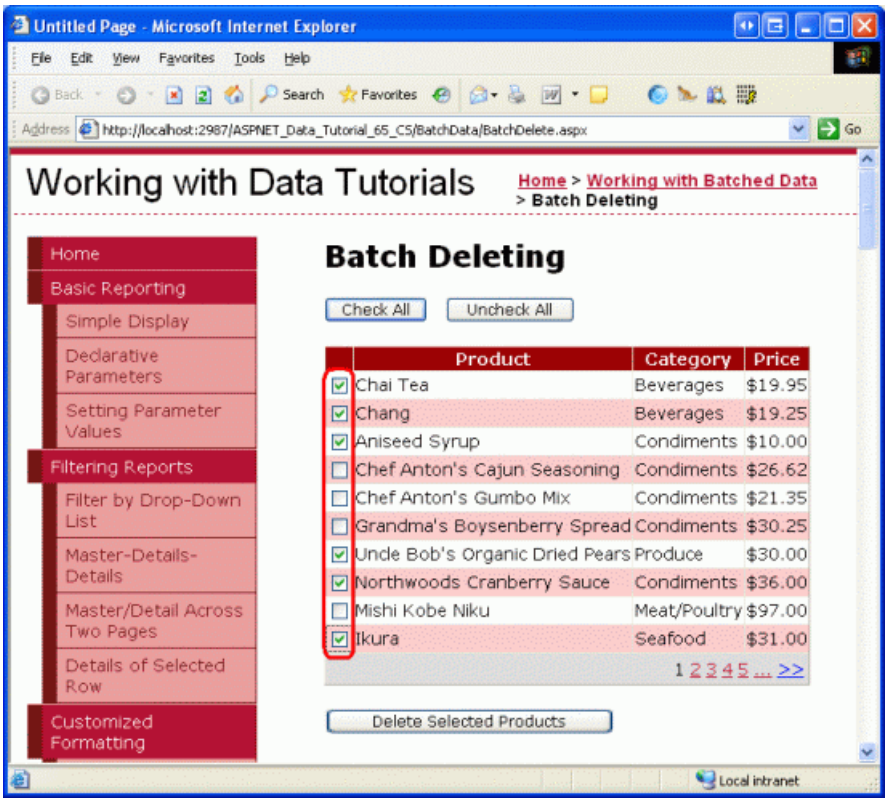


Figure 4: The Selected Products Will Be Deleted (Click to view full-size image)

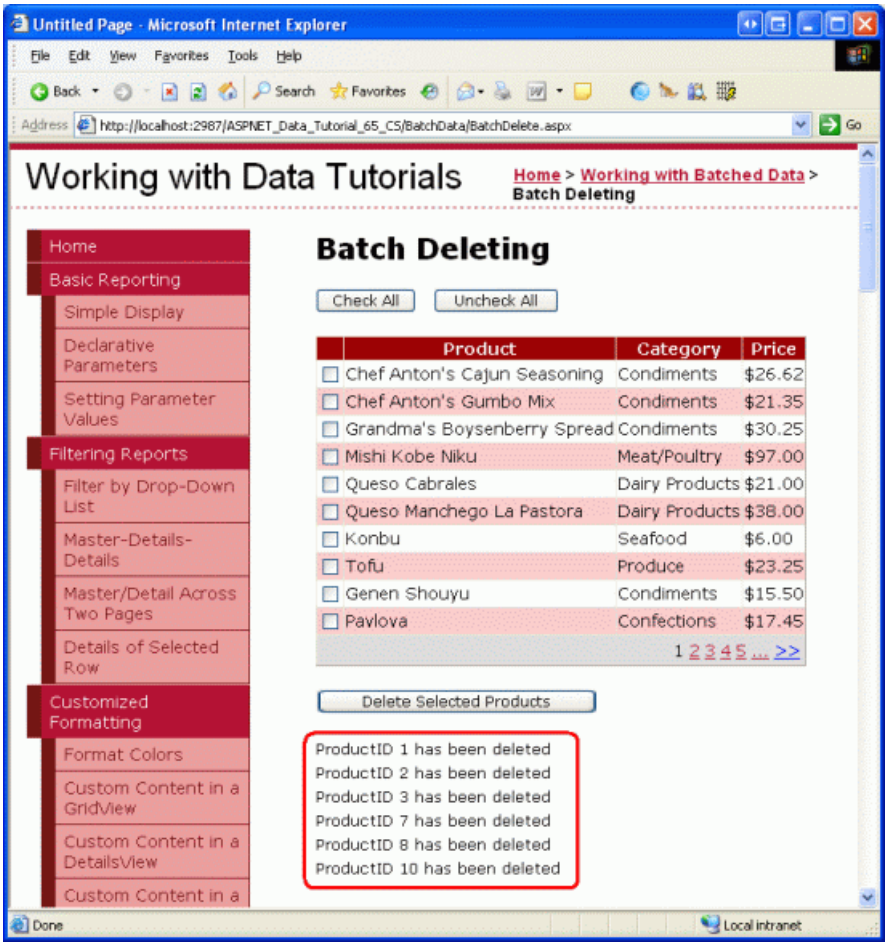


Figure 5: The Deleted Products `ProductID` Values are Listed Beneath the GridView (Click to view full-size image)

NOTE

To test the `DeleteProductsWithTransaction` method s atomicity, manually add an entry for a product in the `Order Details` table and then attempt to delete that product (along with others). You will receive a foreign key constraint violation when attempting to delete the product with an associated order, but note how the other selected products deletions are rolled back.

Summary

Creating a batch deleting interface involves adding a GridView with a column of checkboxes and a Button Web control that, when clicked, will delete all of the selected rows as a single atomic operation. In this tutorial we built such an interface by piecing together work done in two previous tutorials, Adding a GridView Column of Checkboxes and Wrapping Database Modifications within a Transaction. In the first tutorial we created a GridView with a column of checkboxes and in the latter we implemented a method in the BLL that, when passed a `List<T>` of `ProductID` values, deleted them all within the scope of a transaction.

In the next tutorial we'll create an interface for performing batch inserts.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Hilton Giesenow and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Batch Inserting (C#)

2/20/2020 • 23 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Learn how to insert multiple database records in a single operation. In the User Interface Layer we extend the GridView to allow the user to enter multiple new records. In the Data Access Layer we wrap the multiple Insert operations within a transaction to ensure that all insertions succeed or all insertions are rolled back.

Introduction

In the Batch Updating tutorial we looked at customizing the GridView control to present an interface where multiple records were editable. The user visiting the page could make a series of changes and then, with a single button click, perform a batch update. For situations where users commonly update many records in one go, such an interface can save countless clicks and keyboard-to-mouse context switches when compared to the default per-row editing features that were first explored back in the An Overview of Inserting, Updating, and Deleting Data tutorial.

This concept can also be applied when adding records. Imagine that here at Northwind Traders we commonly receive shipments from suppliers that contain a number of products for a particular category. As an example, we might receive a shipment of six different tea and coffee products from Tokyo Traders. If a user enters the six products one at a time through a DetailsView control, they will have to choose many of the same values over and over again: they will need to choose the same category (Beverages), the same supplier (Tokyo Traders), the same discontinued value (False), and the same units on order value (0). This repetitive data entry is not only time consuming, but is prone to errors.

With a little work we can create a batch inserting interface that enables the user to choose the supplier and category once, enter a series of product names and unit prices, and then click a button to add the new products to the database (see Figure 1). As each product is added, its `ProductName` and `UnitPrice` data fields are assigned the values entered in the TextBoxes, while its `CategoryID` and `SupplierID` values are assigned the values from the DropDownLists at the top fo the form. The `Discontinued` and `UnitsOnOrder` values are set to the hard-coded values of `false` and 0, respectively.

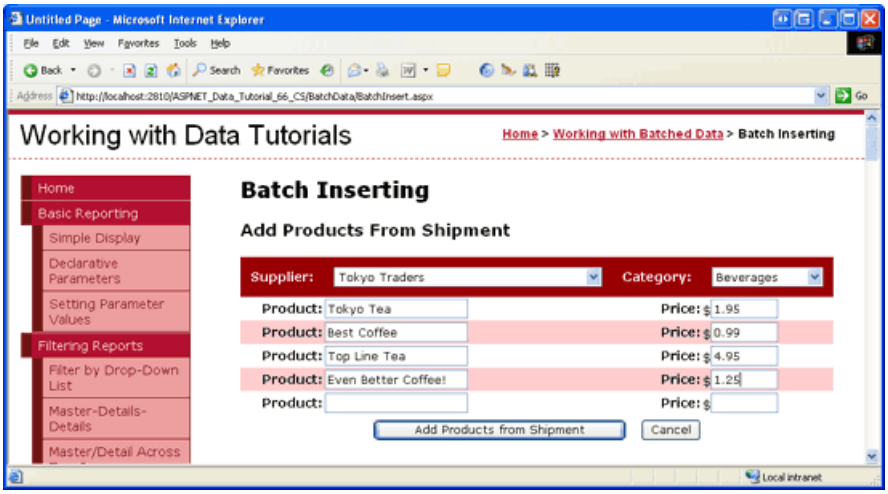


Figure 1: The Batch Inserting Interface (Click to view full-size image)

In this tutorial we will create a page that implements the batch inserting interface shown in Figure 1. As with the

previous two tutorials, we will wrap the insertions within the scope of a transaction to ensure atomicity. Let s get started!

Step 1: Creating the Display Interface

This tutorial will consist of a single page that is divided into two regions: a display region and an inserting region. The display interface, which we'll create in this step, shows the products in a GridView and includes a button titled Process Product Shipment. When this button is clicked, the display interface is replaced with the inserting interface, which is shown in Figure 1. The display interface returns after the Add Products from Shipment or Cancel buttons are clicked. We'll create the inserting interface in Step 2.

When creating a page that has two interfaces, only one of which is visible at a time, each interface typically is placed within a Panel Web control, which serves as a container for other controls. Therefore, our page will have two Panel controls one for each interface.

Start by opening the `BatchInsert.aspx` page in the `BatchData` folder and drag a Panel from the Toolbox onto the Designer (see Figure 2). Set the Panel s `ID` property to `DisplayInterface`. When adding the Panel to the Designer, its `Height` and `Width` properties are set to 50px and 125px, respectively. Clear out these property values from the Properties window.

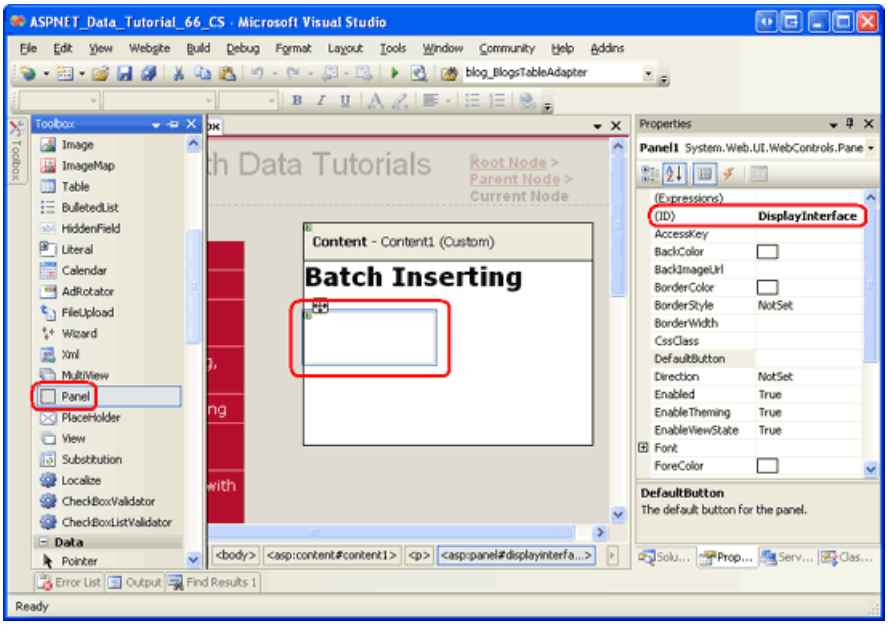


Figure 2: Drag a Panel from the Toolbox onto the Designer (Click to view full-size image)

Next, drag a Button and GridView control into the Panel. Set the Button s `ID` property to `ProcessShipment` and its `Text` property to Process Product Shipment. Set the GridView s `ID` property to `ProductsGrid` and, from its smart tag, bind it to a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to pull its data from the `ProductsBLL` class s `GetProducts` method. Since this GridView is used only to display data, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None). Click Finish to complete the Configure Data Source wizard.

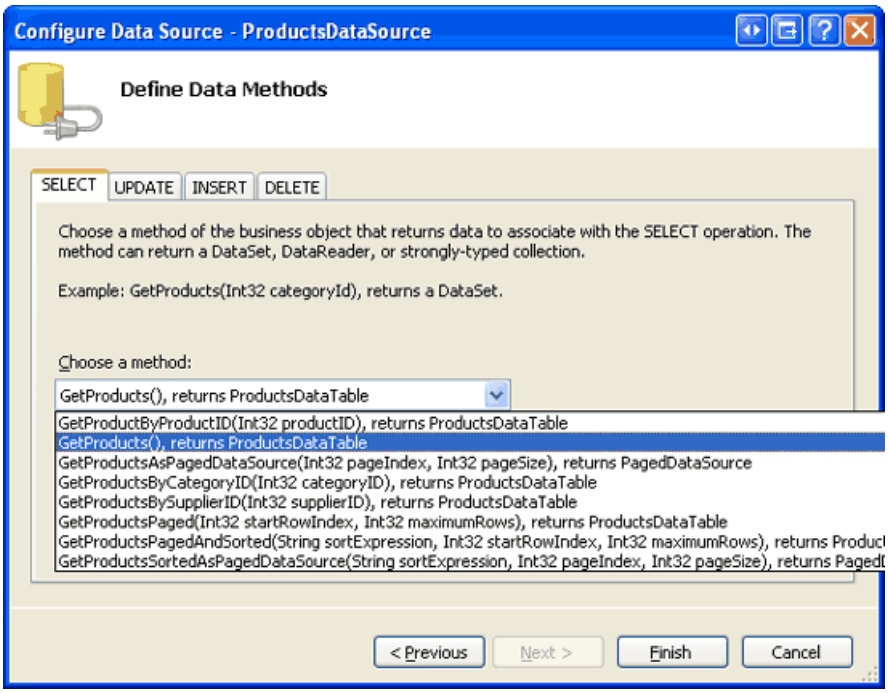


Figure 3: Display the Data Returned from the `ProductsBLL` Class s `GetProducts` Method (Click to view full-size image)

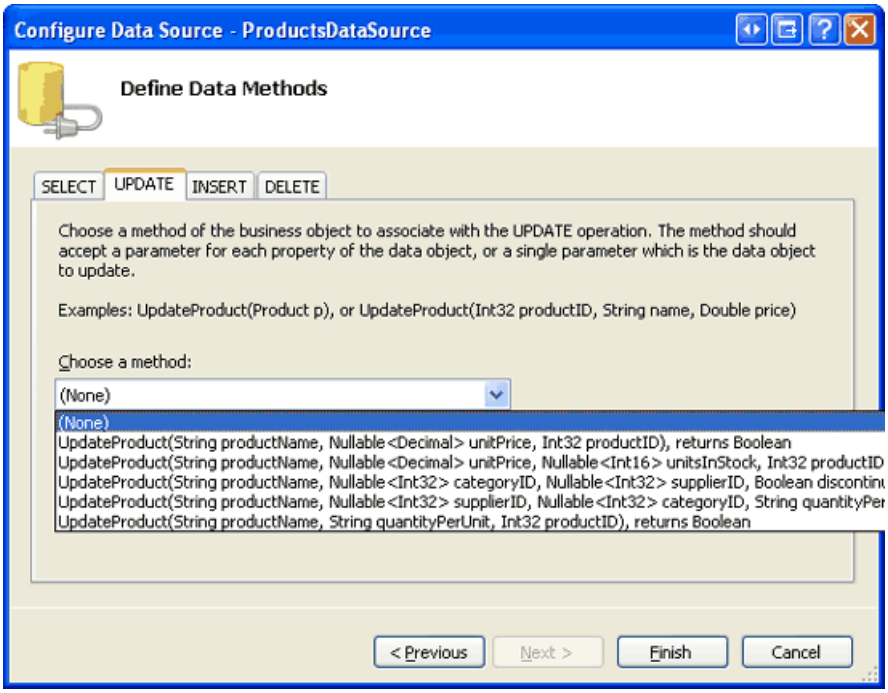


Figure 4: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE Tabs to (None) (Click to view full-size image)

After completing the ObjectDataSource wizard, Visual Studio will add BoundFields and a CheckBoxField for the product data fields. Remove all but the `ProductName`, `CategoryName`, `SupplierName`, `UnitPrice`, and `Discontinued` fields. Feel free to make any aesthetic customizations. I decided to format the `UnitPrice` field as a currency value, reordered the fields, and renamed several of the fields `HeaderText` values. Also configure the GridView to include paging and sorting support by checking the Enable Paging and Enable Sorting checkboxes in the GridView s smart tag.

After adding the Panel, Button, GridView, and ObjectDataSource controls and customizing the GridView s fields, your page s declarative markup should look similar to the following:

```
<asp:Panel ID="DisplayInterface" runat="server">
    <p>
        <asp:Button ID="ProcessShipment" runat="server"
            Text="Process Product Shipment" />
    </p>
    <asp:GridView ID="ProductsGrid" runat="server" AllowPaging="True"
        AllowSorting="True" AutoGenerateColumns="False"
        DataKeyNames="ProductID" DataSourceID="ProductsDataSource">
        <Columns>
            <asp:BoundField DataField="ProductName" HeaderText="Product"
                SortExpression="ProductName" />
            <asp:BoundField DataField="CategoryName" HeaderText="Category"
                ReadOnly="True" SortExpression="CategoryName" />
            <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
                ReadOnly="True" SortExpression="SupplierName" />
            <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
                HeaderText="Price" HtmlEncode="False"
                SortExpression="UnitPrice">
                <ItemStyle HorizontalAlign="Right" />
            </asp:BoundField>
            <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
                SortExpression="Discontinued">
                <ItemStyle HorizontalAlign="Center" />
            </asp:CheckBoxField>
        </Columns>
    </asp:GridView>
    <asp:ObjectDataSource ID="ProductsDataSource" runat="server"
        OldValuesParameterFormatString="original_{0}"
        SelectMethod="GetProducts" TypeName="ProductsBLL">
    </asp:ObjectDataSource>
</asp:Panel>
```

Note that the markup for the Button and GridView appear within the opening and closing `<asp:Panel>` tags. Since these controls are within the `DisplayInterface` Panel, we can hide them by simply setting the Panel s `Visible` property to `false`. Step 3 looks at programmatically changing the Panel s `Visible` property in response to a button click to show one interface while hiding the other.

Take a moment to view our progress through a browser. As Figure 5 shows, you should see a Process Product Shipment button above a GridView that lists the products ten at a time.

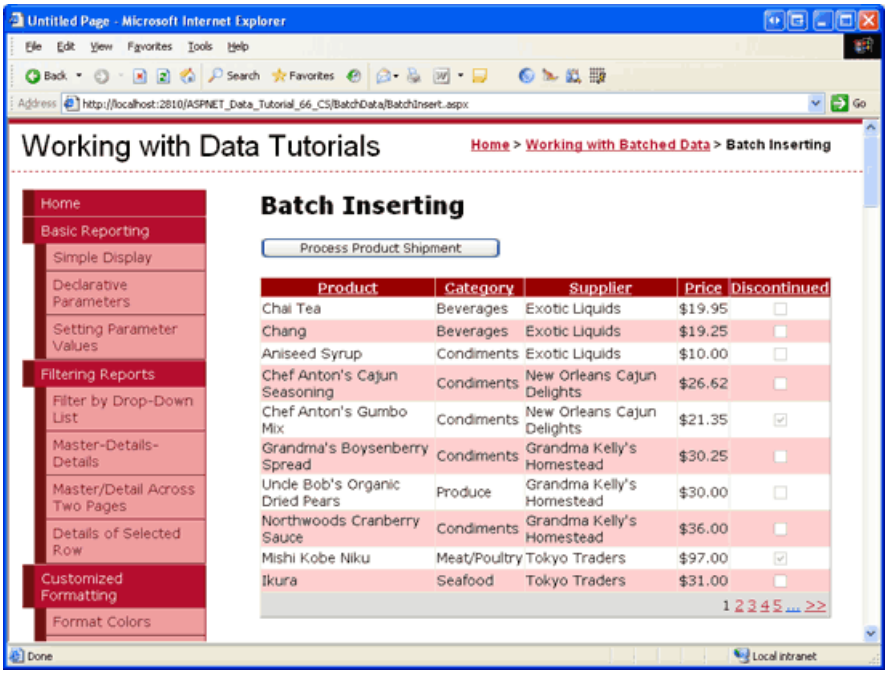


Figure 5: The GridView Lists the Products and Offers Sorting and Paging Capabilities (Click to view full-size image)

Step 2: Creating the Inserting Interface

With the display interface complete, we re ready to create the inserting interface. For this tutorial, let s create an inserting interface that prompts for a single supplier and category value and then allows the user to enter up to five product names and unit price values. With this interface, the user can add one to five new products that all share the same category and supplier, but have unique product names and prices.

Start by dragging a Panel from the Toolbox onto the Designer, placing it beneath the existing `DisplayInterface` Panel. Set the `ID` property of this newly added Panel to `InsertingInterface` and set its `Visible` property to `false`. We'll add code that sets the `InsertingInterface` Panel s `Visible` property to `true` in Step 3. Also clear out the Panel s `Height` and `Width` property values.

Next, we need to create the inserting interface that was shown back in Figure 1. This interface can be created through a variety of HTML techniques, but we will use a fairly straightforward one: a four-column, seven-row table.

> **NOTE**
>
> When entering markup for HTML `<table>` elements, I prefer to use the Source view. While Visual Studio does have tools for adding `<table>` elements through the Designer, the Designer seems all too willing to inject unasked for `style` settings into the markup. Once I have created the `<table>` markup, I usually return to the Designer to add the Web controls and set their properties. When creating tables with pre-determined columns and rows I prefer using static HTML rather than the Table Web control because any Web controls placed within a Table Web control can only be accessed using the `FindControl("controlID")` pattern. I do, however, use Table Web controls for dynamically-sized tables (ones whose rows or columns are based on some database or user-specified criteria), since the Table Web control can be constructed programmatically.

Enter the following markup within the `<asp:Panel>` tags of the `InsertingInterface` Panel:

```
<table class="DataWebControlStyle" cellspacing="0">
    <tr class="BatchInsertHeaderRow">
        <td class="BatchInsertLabel">Supplier:</td>
        <td></td>
        <td class="BatchInsertLabel">Category:</td>
        <td></td>
    </tr>
    <tr class="BatchInsertRow">
        <td class="BatchInsertLabel">Product:</td>
        <td></td>
        <td class="BatchInsertLabel">Price:</td>
        <td></td>
    </tr>
    <tr class="BatchInsertAlternatingRow">
        <td class="BatchInsertLabel">Product:</td>
        <td></td>
        <td class="BatchInsertLabel">Price:</td>
        <td></td>
    </tr>
    <tr class="BatchInsertRow">
        <td class="BatchInsertLabel">Product:</td>
        <td></td>
        <td class="BatchInsertLabel">Price:</td>
        <td></td>
    </tr>
    <tr class="BatchInsertAlternatingRow">
        <td class="BatchInsertLabel">Product:</td>
        <td></td>
        <td class="BatchInsertLabel">Price:</td>
        <td></td>
    </tr>
    <tr class="BatchInsertRow">
        <td class="BatchInsertLabel">Product:</td>
        <td></td>
        <td class="BatchInsertLabel">Price:</td>
        <td></td>
    </tr>
    <tr class="BatchInsertFooterRow">
        <td colspan="4">
        </td>
    </tr>
</table>
```

This `<table>` markup does not include any Web controls yet, we'll add those momentarily. Note that each `<tr>` element contains a particular CSS class setting: `BatchInsertHeaderRow` for the header row where the supplier and category DropDownLists will go; `BatchInsertFooterRow` for the footer row where the Add Products from Shipment and Cancel Buttons will go; and alternating `BatchInsertRow` and `BatchInsertAlternatingRow` values for the rows that will contain the product and unit price TextBox controls. I ve created corresponding CSS classes in the `Styles.css` file to give the inserting interface an appearance similar to the GridView and DetailsView controls we ve used throughout these tutorials. These CSS classes are shown below.

```
/*** Styles for ~/BatchData/BatchInsert.aspx tutorial ***/
.BatchInsertLabel
{
    font-weight: bold;
    text-align: right;
}
.BatchInsertHeaderRow td
{
    color: White;
    background-color: #900;
    padding: 11px;
}
.BatchInsertFooterRow td
{
    text-align: center;
    padding-top: 5px;
}
.BatchInsertRow
{
}
.BatchInsertAlternatingRow
{
    background-color: #fcc;
}
```

With this markup entered, return to the Design view. This `<table>` should show as a four-column, seven-row table in the Designer, as Figure 6 illustrates.

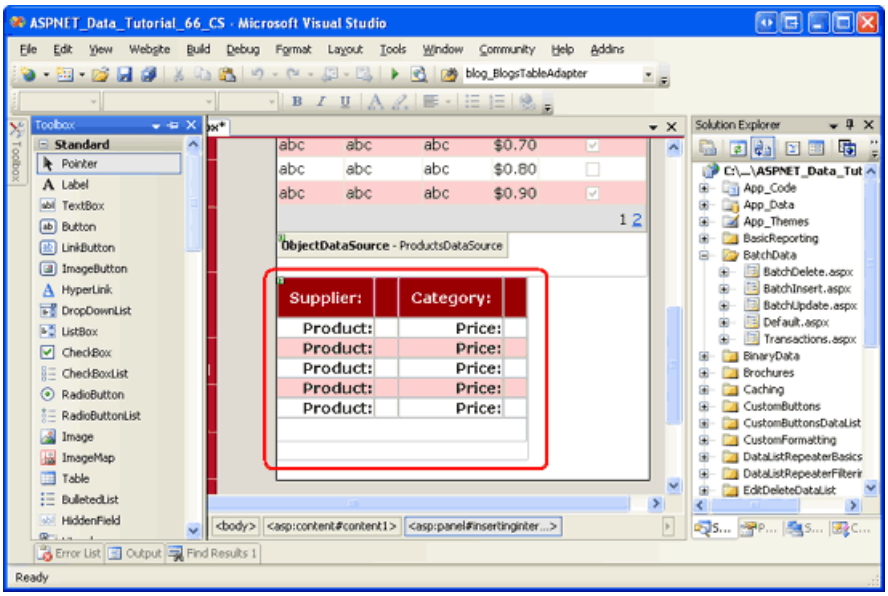


Figure 6: The Inserting Interface is Composed of a Four-Column, Seven-Row Table (Click to view full-size image)

We re now ready to add the Web controls to the inserting interface. Drag two DropDownLists from the Toolbox into the appropriate cells in the table one for the supplier and one for the category.

Set the supplier DropDownList s `ID` property to `Suppliers` and bind it to a new ObjectDataSource named `SuppliersDataSource`. Configure the new ObjectDataSource to retrieve its data from the `SuppliersBLL` class s `GetSuppliers` method and set the UPDATE tab s drop-down list to (None). Click Finish to complete the wizard.

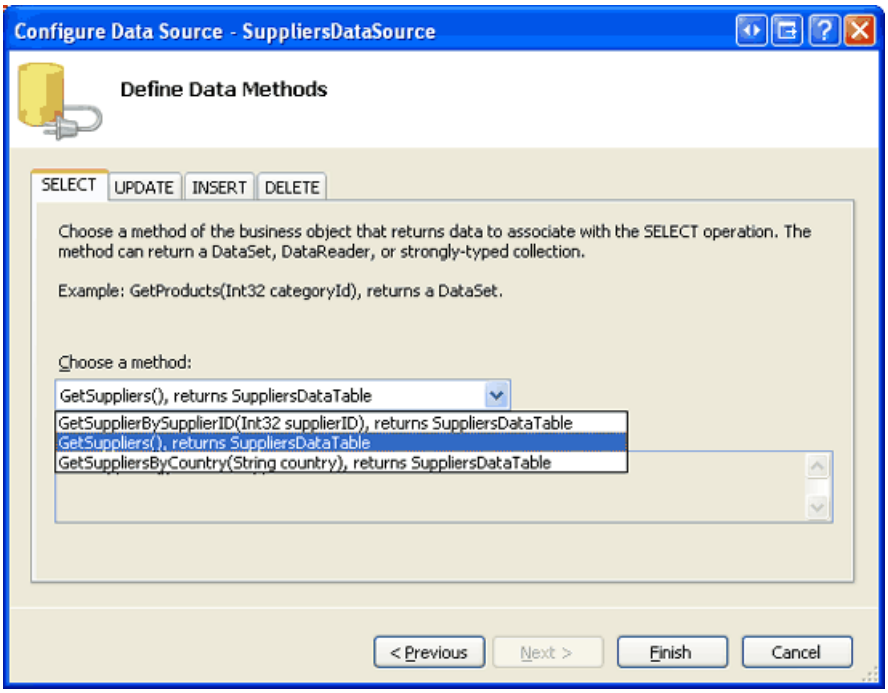


Figure 7: Configure the ObjectDataSource to Use the `SuppliersBLL` Class s `GetSuppliers` Method (Click to view full-size image)

Have the `Suppliers` DropDownList display the `CompanyName` data field and use the `SupplierID` data field as its `ListItem` s values.

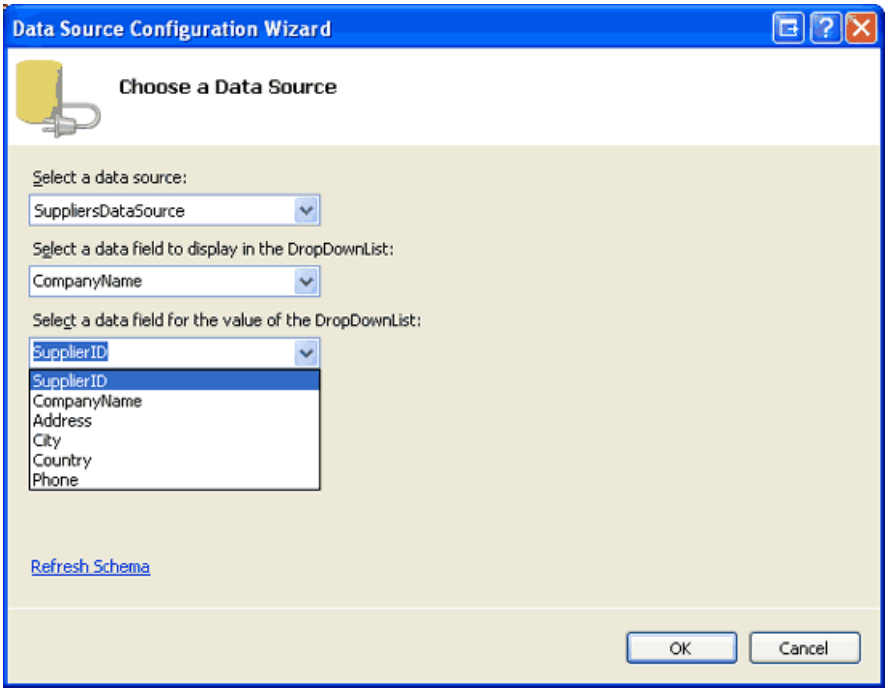


Figure 8: Display the `CompanyName` Data Field and Use `SupplierID` as the Value (Click to view full-size image)

Name the second DropDownList `Categories` and bind it to a new ObjectDataSource named `CategoriesDataSource`. Configure the `CategoriesDataSource` ObjectDataSource to use the `CategoriesBLL` class s `GetCategories` method; set the drop-down lists in the UPDATE and DELETE tabs to (None) and click Finish to complete the wizard. Finally, have the DropDownList display the `CategoryName` data field and use the `CategoryID` as the value.

After these two DropDownLists have been added and bound to appropriately configured ObjectDataSources, your screen should look similar to Figure 9.

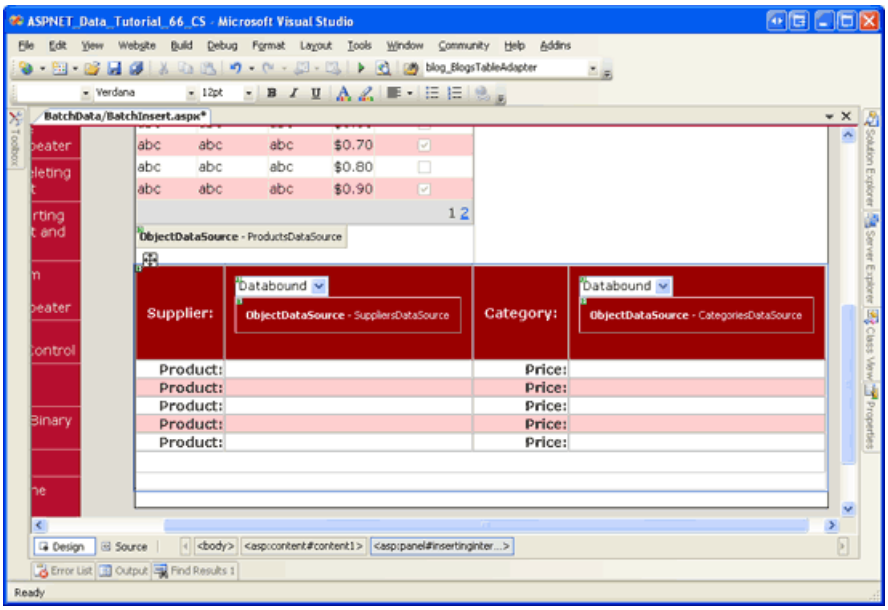


Figure 9: The Header Row Now Contains the `Suppliers` and `Categories` DropDownLists (Click to view full-size image)

We now need to create the TextBoxes to collect the name and price for each new product. Drag a TextBox control from the Toolbox onto the Designer for each of the five product name and price rows. Set the `ID` properties of the TextBoxes to `ProductName1`, `UnitPrice1`, `ProductName2`, `UnitPrice2`, `ProductName3`, `UnitPrice3`, and so on.

Add a CompareValidator after each of the unit price TextBoxes, setting the `ControlToValidate` property to the appropriate `ID`. Also set the `Operator` property to `GreaterThanEqual`, `ValueToCompare` to 0, and `Type` to `Currency`. These settings instruct the CompareValidator to ensure that the price, if entered, is a valid currency value that is greater than or equal to zero. Set the `Text` property to *, and `ErrorMessage` to The price must be greater than or equal to zero. Also, please omit any currency symbols.

> **NOTE**
>
> The inserting interface does not include any RequiredFieldValidator controls, even though the `ProductName` field in the `Products` database table does not allow `NULL` values. This is because we want to let the user enter up to five products. For example, if the user were to provide the product name and unit price for the first three rows, leaving the last two rows blank, we d just add three new products to the system. Since `ProductName` is required, however, we will need to programmatically check to ensure that if a unit price is entered that a corresponding product name value is provided. We'll tackle this check in Step 4.

When validating the user s input, the CompareValidator reports invalid data if the value contains a currency symbol. Add a $ in front of each of the unit price TextBoxes to serve as a visual cue that instructs the user to omit the currency symbol when entering the price.

Lastly, add a ValidationSummary control within the `InsertingInterface` Panel, settings its `ShowMessageBox` property to `true` and its `ShowSummary` property to `false`. With these settings, if the user enters an invalid unit price value, an asterisk will appear next to the offending TextBox controls and the ValidationSummary will display a client-side messagebox that shows the error message we specified earlier.

At this point, your screen should look similar to Figure 10.

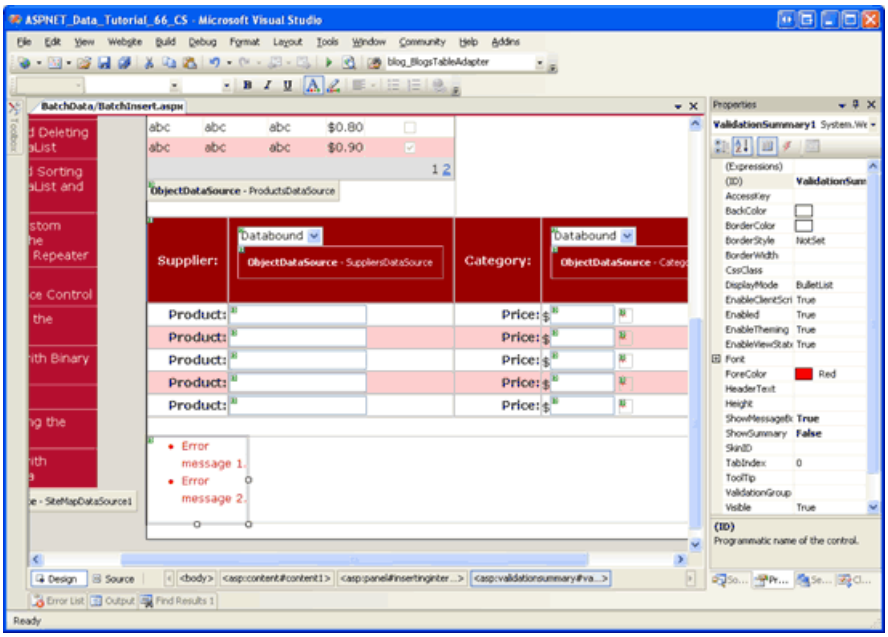


Figure 10: The Inserting Interface Now Includes TextBoxes for the Products Names and Prices (Click to view full-size image)

Next we need to add the Add Products from Shipment and Cancel buttons to the footer row. Drag two Button controls from the Toolbox into the footer of the inserting interface, setting the Buttons `ID` properties to `AddProducts` and `CancelButton` and `Text` properties to Add Products from Shipment and Cancel, respectively. In addition, set the `CancelButton` control s `CausesValidation` property to `false`.

Finally, we need to add a Label Web control that will display status messages for the two interfaces. For example, when a user successfully adds a new shipment of products, we want to return to the display interface and display a confirmation message. If, however, the user provides a price for a new product but leaves off the product name, we need to display a warning message since the `ProductName` field is required. Since we need this message to display for both interfaces, place it at the top of the page outside of the Panels.

Drag a Label Web control from the Toolbox to the top of the page in the Designer. Set the `ID` property to `StatusLabel`, clear out the `Text` property, and set the `Visible` and `EnableViewState` properties to `false`. As we have seen in previous tutorials, setting the `EnableViewState` property to `false` allows us to programmatically change the Label s property values and have them automatically revert back to their defaults on the subsequent postback. This simplifies the code for showing a status message in response to some user action that disappears on the subsequent postback. Finally, set the `StatusLabel` control s `CssClass` property to Warning, which is the name of a CSS class defined in `Styles.css` that displays text in a large, italic, bold, red font.

Figure 11 shows the Visual Studio Designer after the Label has been added and configured.

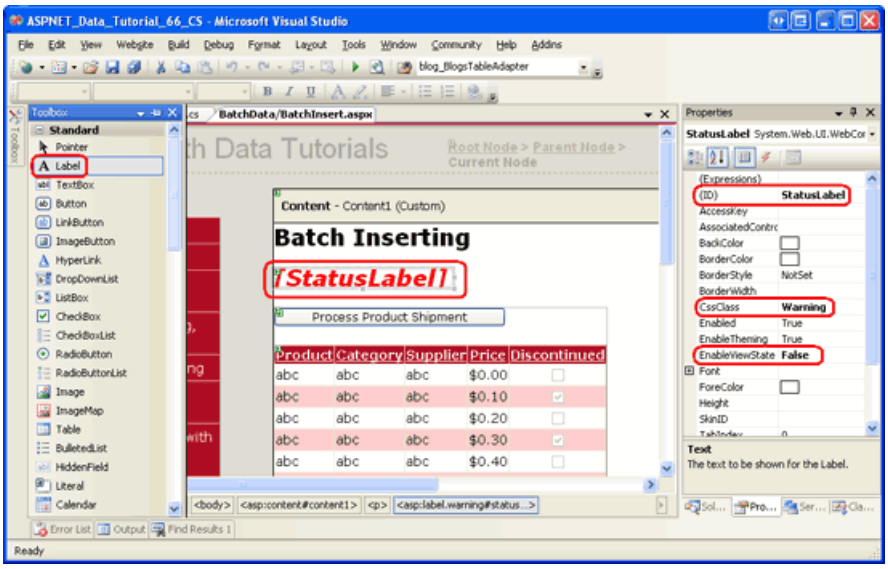


Figure 11: Place the `StatusLabel` Control Above the Two Panel Controls (Click to view full-size image)

Step 3: Switching Between the Display and Inserting Interfaces

At this point we have completed the markup for our display and inserting interfaces, but we re still left with two tasks:

- Switching between the display and inserting interfaces
- Adding the products in the shipment to the database

Currently, the display interface is visible but the inserting interface is hidden. This is because the `DisplayInterface` Panel s `Visible` property is set to `true` (the default value), while the `InsertingInterface` Panel s `Visible` property is set to `false`. To switch between the two interfaces we simply need to toggle each control s `Visible` property value.

We want to move from the display interface to the inserting interface when the Process Product Shipment button is clicked. Therefore, create an event handler for this Button s `Click` event that contains the following code:

```
protected void ProcessShipment_Click(object sender, EventArgs e)
{
    DisplayInterface.Visible = false;
    InsertingInterface.Visible = true;
}
```

This code simply hides the `DisplayInterface` Panel and shows the `InsertingInterface` Panel.

Next, create event handlers for the Add Products from Shipment and Cancel Button controls in the inserting interface. When either of these Buttons is clicked, we need to revert back to the display interface. Create `Click` event handlers for both Button controls so that they call `ReturnToDisplayInterface`, a method we will add momentarily. In addition to hiding the `InsertingInterface` Panel and showing the `DisplayInterface` Panel, the `ReturnToDisplayInterface` method needs to return the Web controls to their pre-editing state. This involves setting the DropDownLists `SelectedIndex` properties to 0 and clearing out the `Text` properties of the TextBox controls.

NOTE

Consider what might happen if we didn't return the controls to their pre-editing state before returning to the display interface. A user might click the Process Product Shipment button, enter the products from the shipment, and then click Add Products from Shipment. This would add the products and return the user to the display interface. At this point the user might want to add another shipment. Upon clicking the Process Product Shipment button they would return to the inserting interface but the DropDownList selections and TextBox values would still be populated with their previous values.

```
protected void AddProducts_Click(object sender, EventArgs e)
{
    // TODO: Save the products
    // Revert to the display interface
    ReturnToDisplayInterface();
}
protected void CancelButton_Click(object sender, EventArgs e)
{
    // Revert to the display interface
    ReturnToDisplayInterface();
}
const int firstControlID = 1;
const int lastControlID = 5;
private void ReturnToDisplayInterface()
{
    // Reset the control values in the inserting interface
    Suppliers.SelectedIndex = 0;
    Categories.SelectedIndex = 0;
    for (int i = firstControlID; i <= lastControlID; i++)
    {
        ((TextBox)InsertingInterface.FindControl("ProductName" + i.ToString())).Text =
            string.Empty;
        ((TextBox)InsertingInterface.FindControl("UnitPrice" + i.ToString())).Text =
            string.Empty;
    }
    DisplayInterface.Visible = true;
    InsertingInterface.Visible = false;
}
```

Both `Click` event handlers simply call the `ReturnToDisplayInterface` method, although we'll return to the Add Products from Shipment `Click` event handler in Step 4 and add code to save the products. `ReturnToDisplayInterface` starts by returning the `Suppliers` and `Categories` DropDownLists to their first options. The two constants `firstControlID` and `lastControlID` mark the starting and ending control index values used in naming the product name and unit price TextBoxes in the inserting interface and are used in the bounds of the `for` loop that sets the `Text` properties of the TextBox controls back to an empty string. Finally, the Panels `Visible` properties are reset so that the inserting interface is hidden and the display interface shown.

Take a moment to test out this page in a browser. When first visiting the page you should see the display interface as was shown in Figure 5. Click the Process Product Shipment button. The page will postback and you should now see the inserting interface as shown in Figure 12. Clicking either the Add Products from Shipment or Cancel buttons returns you to the display interface.

NOTE

While viewing the inserting interface, take a moment to test out the CompareValidators on the unit price TextBoxes. You should see a client-side messagebox warning when clicking the Add Products from Shipment button with invalid currency values or prices with a value less than zero.

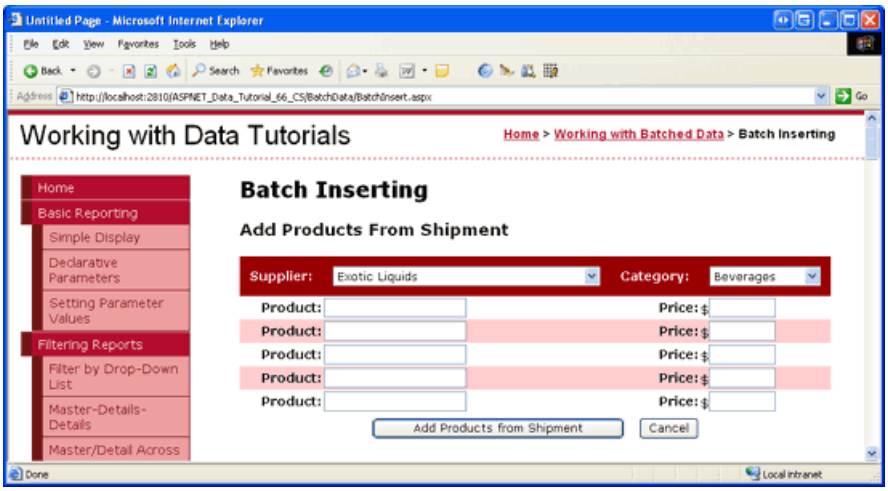


Figure 12: The Inserting Interface is Displayed After Clicking the Process Product Shipment Button (Click to view full-size image)

Step 4: Adding the Products

All that remains for this tutorial is to save the products to the database in the Add Products from Shipment Button s `Click` event handler. This can be accomplished by creating a `ProductsDataTable` and adding a `ProductsRow` instance for each of the product names supplied. Once these `ProductsRow` s have been added we will make a call to the `ProductsBLL` class s `UpdateWithTransaction` method passing in the `ProductsDataTable`. Recall that the `UpdateWithTransaction` method, which was created back in the Wrapping Database Modifications within a Transaction tutorial, passes the `ProductsDataTable` to the `ProductsTableAdapter` s `UpdateWithTransaction` method. From there, an ADO.NET transaction is started and the TableAdapter issues an `INSERT` statement to the database for each added `ProductsRow` in the DataTable. Assuming all products are added without error, the transaction is committed, otherwise it is rolled back.

The code for the Add Products from Shipment Button s `Click` event handler also needs to perform a bit of error checking. Since there are no RequiredFieldValidators used in the inserting interface, a user could enter a price for a product while omitting its name. Since the product s name is required, if such a condition unfolds we need to alert the user and not proceed with the inserts. The complete `Click` event handler code follows:

```
protected void AddProducts_Click(object sender, EventArgs e)
{
    // Make sure that the UnitPrice CompareValidators report valid data...
    if (!Page.IsValid)
        return;
    // Add new ProductsRows to a ProductsDataTable...
    Northwind.ProductsDataTable products = new Northwind.ProductsDataTable();
    for (int i = firstControlID; i <= lastControlID; i++)
    {
        // Read in the values for the product name and unit price
        string productName = ((TextBox)InsertingInterface.FindControl
            ("ProductName" + i.ToString())).Text.Trim();
        string unitPrice = ((TextBox)InsertingInterface.FindControl
            ("UnitPrice" + i.ToString())).Text.Trim();
        // Ensure that if unitPrice has a value, so does productName
        if (unitPrice.Length > 0 && productName.Length == 0)
        {
            // Display a warning and exit this event handler
            StatusLabel.Text = "If you provide a unit price you must also " +
                "include the name of the product.";
            StatusLabel.Visible = true;
            return;
        }
        // Only add the product if a product name value is provided
        if (productName.Length > 0)
        {
            // Add a new ProductsRow to the ProductsDataTable
            Northwind.ProductsRow newProduct = products.NewProductsRow();
            // Assign the values from the web page
            newProduct.ProductName = productName;
            newProduct.SupplierID = Convert.ToInt32(Suppliers.SelectedValue);
            newProduct.CategoryID = Convert.ToInt32(Categories.SelectedValue);
            if (unitPrice.Length > 0)
                newProduct.UnitPrice = Convert.ToDecimal(unitPrice);
            // Add any "default" values
            newProduct.Discontinued = false;
            newProduct.UnitsOnOrder = 0;
            products.AddProductsRow(newProduct);
        }
    }
    // If we reach here, see if there were any products added
    if (products.Count > 0)
    {
        // Add the new products to the database using a transaction
        ProductsBLL productsAPI = new ProductsBLL();
        productsAPI.UpdateWithTransaction(products);
        // Rebind the data to the grid so that the products just added are displayed
        ProductsGrid.DataBind();
        // Display a confirmation (don't use the Warning CSS class, though)
        StatusLabel.CssClass = string.Empty;
        StatusLabel.Text = string.Format(
            "{0} products from supplier {1} have been added and filed under " +
            "category {2}.", products.Count, Suppliers.SelectedItem.Text,
            Categories.SelectedItem.Text);
        StatusLabel.Visible = true;
        // Revert to the display interface
        ReturnToDisplayInterface();
    }
    else
    {
        // No products supplied!
        StatusLabel.Text = "No products were added. Please enter the product " +
            "names and unit prices in the textboxes.";
        StatusLabel.Visible = true;
    }
}
```

The event handler starts by ensuring that the `Page.IsValid` property returns a value of `true`. If it returns `false`, then that means one or more of the CompareValidators are reporting invalid data; in such a case we do not want to attempt to insert the entered products or we'll end up with an exception when attempting to assign the user-entered unit price value to the `ProductsRow` s `UnitPrice` property.

Next, a new `ProductsDataTable` instance is created (`products`). A `for` loop is used to iterate through the product name and unit price TextBoxes and the `Text` properties are read into the local variables `productName` and `unitPrice`. If the user has entered a value for the unit price but not for the corresponding product name, the `StatusLabel` displays the message If you provide a unit price you must also include the name of the product and the event handler is exited.

If a product name has been provided, a new `ProductsRow` instance is created using the `ProductsDataTable` s `NewProductsRow` method. This new `ProductsRow` instance s `ProductName` property is set to the current product name TextBox while the `SupplierID` and `CategoryID` properties are assigned to the `SelectedValue` properties of the DropDownLists in the inserting interface s header. If the user entered a value for the product s price, it is assigned to the `ProductsRow` instance s `UnitPrice` property; otherwise, the property is left unassigned, which will result in a `NULL` value for `UnitPrice` in the database. Finally, the `Discontinued` and `UnitsOnOrder` properties are assigned to the hard-coded values `false` and 0, respectively.

After the properties have been assigned to the `ProductsRow` instance it is added to the `ProductsDataTable`.

At the completion of the `for` loop, we check whether any products have been added. The user may, after all, have clicked the Add Products from Shipment before entering any product names or prices. If there is at least one product in the `ProductsDataTable`, the `ProductsBLL` class s `UpdateWithTransaction` method is called. Next, the data is rebound to the `ProductsGrid` GridView so that the newly added products will appear in the display interface. The `StatusLabel` is updated to display a confirmation message and the `ReturnToDisplayInterface` is invoked, hiding the inserting interface and showing the display interface.

If no products were entered, the inserting interface remains displayed but the message No products were added. Please enter the product names and unit prices in the textboxes is displayed.

Figure s 13, 14, and 15 show the inserting and display interfaces in action. In Figure 13, the user has entered a unit price value without a corresponding product name. Figure 14 shows the display interface after three new products have been added successfully, while Figure 15 shows two of the newly added products in the GridView (the third one is on the previous page).

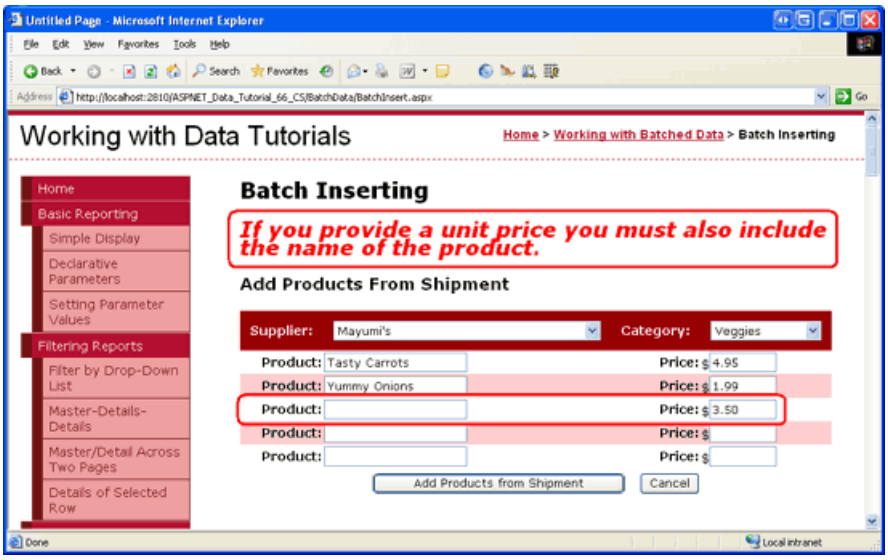


Figure 13: A Product Name is Required When Entering a Unit Price (Click to view full-size image)

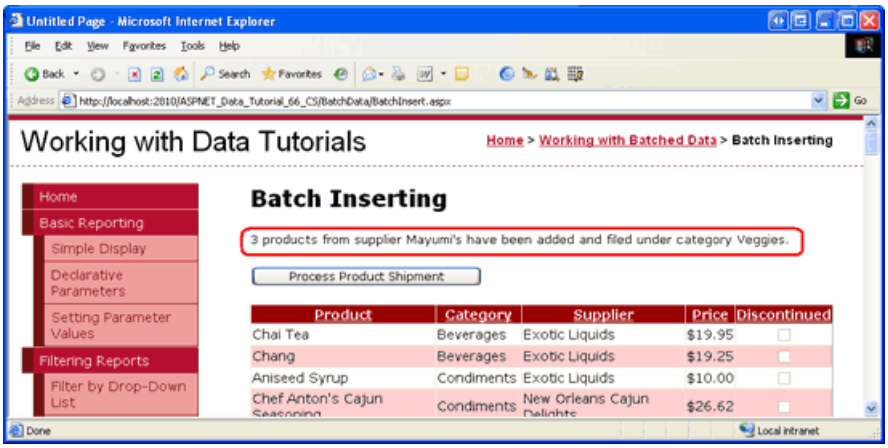


Figure 14: Three New Veggies Have Been Added for the Supplier Mayumi s (Click to view full-size image)

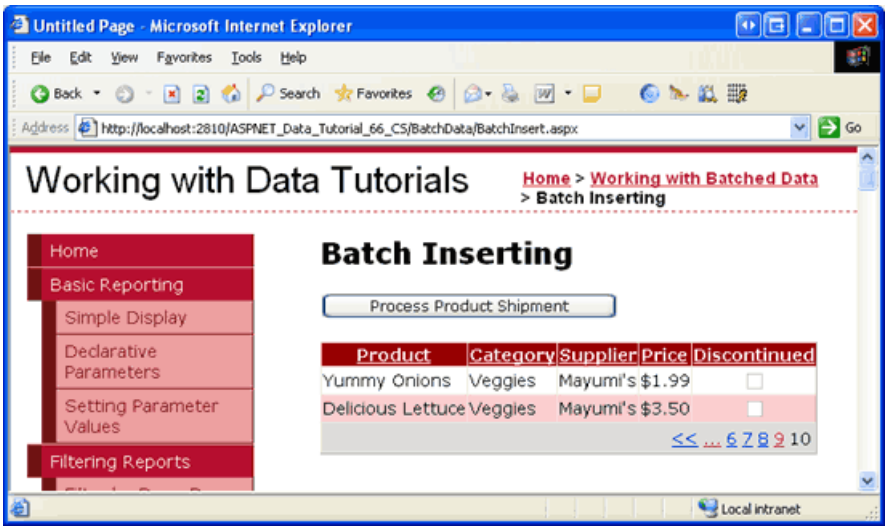


Figure 15: The New Products Can Be Found in the Last Page of the GridView (Click to view full-size image)

> **NOTE**
>
> The batch inserting logic used in this tutorial wraps the inserts within the scope of transaction. To verify this, purposefully introduce a database-level error. For example, rather than assigning the new `ProductsRow` instance s `CategoryID` property to the selected value in the `Categories` DropDownList, assign it to a value like `i * 5`. Here `i` is the loop indexer and has values ranging from 1 to 5. Therefore, when adding two or more products in batch insert the first product will have a valid `CategoryID` value (5), but subsequent products will have `CategoryID` values that do not match up to `CategoryID` values in the `Categories` table. The net effect is that while the first `INSERT` will succeed, subsequent ones will fail with a foreign key constraint violation. Since the batch insert is atomic, the first `INSERT` will be rolled back, returning the database to its state before the batch insert process began.

Summary

Over this and the previous two tutorials we have created interfaces that allow for updating, deleting, and inserting batches of data, all of which used the transaction support we added to the Data Access Layer in the Wrapping Database Modifications within a Transaction tutorial. For certain scenarios, such batch processing user interfaces greatly improve end user efficiency by cutting down on the number of clicks, postbacks, and keyboard-to-mouse context switches, while also maintaining the integrity of the underlying data.

This tutorial completes our look at working with batched data. The next set of tutorials explores a variety of advanced Data Access Layer scenarios, including using stored procedures in the TableAdapter s methods, configuring connection- and command-level settings in the DAL, encrypting connection strings, and more!

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Hilton Giesenow and S ren Jacob Lauritsen. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Wrapping Database Modifications within a Transaction (VB)

2/20/2020 • 23 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> This tutorial is the first of four that looks at updating, deleting, and inserting batches of data. In this tutorial we learn how database transactions allow batch modifications to be carried out as an atomic operation, which ensures that either all steps succeed or all steps fail.

Introduction

As we saw starting with the An Overview of Inserting, Updating, and Deleting Data tutorial, the GridView provides built-in support for row-level editing and deleting. With a few clicks of the mouse it is possible to create a rich data modification interface without writing a line of code, so long as you are content with editing and deleting on a per-row basis. However, in certain scenarios this is insufficient and we need to provide users with the ability to edit or delete a batch of records.

For example, most web-based email clients use a grid to list each message where each row includes a checkbox along with the email s information (subject, sender, and so forth). This interface permits the user to delete multiple messages by checking them and then clicking a Delete Selected Messages button. A batch editing interface is ideal in situations where users commonly edit many different records. Rather than forcing the user to click Edit, make their change, and then click Update for each record that needs to be modified, a batch editing interface renders each row with its editing interface. The user can quickly modify the set of rows that need to be changed and then save these changes by clicking an Update All button. In this set of tutorials we'll examine how to create interfaces for inserting, editing, and deleting batches of data.

When performing batch operations it s important to determine whether it should be possible for some of the operations in the batch to succeed while others fail. Consider a batch deleting interface - what should happen if the first selected record is deleted successfully, but the second one fails, say, because of a foreign key constraint violation? Should the first record s delete be rolled back or is it acceptable for the first record to remain deleted?

If you want the batch operation to be treated as an atomic operation, one where either all of the steps succeed or all of the steps fail, then the Data Access Layer needs to be augmented to include support for database transactions. Database transactions guarantee atomicity for the set of `INSERT`, `UPDATE`, and `DELETE` statements executed under the umbrella of the transaction and are a feature supported by most all modern database systems.

In this tutorial we'll look at how to extend the DAL to use database transactions. Subsequent tutorials will examine implementing web pages for batch inserting, updating, and deleting interfaces. Let s get started!

NOTE

When modifying data in a batch transaction, atomicity is not always needed. In some scenarios, it may be acceptable to have some data modifications succeed and others in the same batch fail, such as when deleting a set of emails from a web-based email client. If there s a database error midway through the deletion process, it s probably acceptable that those records processed without error remain deleted. In such cases, the DAL does not need to be modified to support database transactions. There are other batch operation scenarios, however, where atomicity is vital. When a customer moves her funds from one bank account to another, two operations must be performed: the funds must be deducted from the first account and then added to the second. While the bank may not mind having the first step succeed but the second step fail, its customers would understandably be upset. I encourage you to work through this tutorial and implement the enhancements to the DAL to support database transactions even if you do not plan on using them in the batch inserting, updating, and deleting interfaces we'll be building in the following three tutorials.

An Overview of Transactions

Most databases include support for *transactions*, which enable multiple database commands to be grouped into a single logical unit of work. The database commands that comprise a transaction are guaranteed to be atomic, meaning that either all commands will fail or all will succeed.

In general, transactions are implemented through SQL statements using the following pattern:

1. Indicate the start of a transaction.
2. Execute the SQL statements that comprise the transaction.
3. If there is an error in one of the statements from Step 2, rollback the transaction.
4. If all of the statements from Step 2 complete without error, commit the transaction.

The SQL statements used to create, commit, and roll back the transaction can be entered manually when writing SQL scripts or creating stored procedures, or through programmatic means using either ADO.NET or the classes in the `System.Transactions` namespace. In this tutorial we will only examine managing transactions using ADO.NET. In a future tutorial we will look at how to use stored procedures in the Data Access Layer, at which time we'll explore the SQL statements for creating, rolling back, and committing transactions. In the meantime, consult Managing Transactions in SQL Server Stored Procedures for more information.

NOTE

The `TransactionScope` class in the `System.Transactions` namespace enables developers to programmatically wrap a series of statements within the scope of a transaction and includes support for complex transactions that involve multiple sources, such as two different databases or even heterogeneous types of data stores, such as a Microsoft SQL Server database, an Oracle database, and a Web service. I ve decided to use ADO.NET transactions for this tutorial instead of the `TransactionScope` class because ADO.NET is more specific for database transactions and, in many cases, is far less resource intensive. In addition, under certain scenarios the `TransactionScope` class uses the Microsoft Distributed Transaction Coordinator (MSDTC). The configuration, implementation, and performance issues surrounding MSDTC makes it a rather specialized and advanced topic and beyond the scope of these tutorials.

When working with the SqlClient provider in ADO.NET, transactions are initiated through a call to the `SqlConnection` class s `BeginTransaction` method, which returns a `SqlTransaction` object. The data modification statements that makeup the transaction are placed within a `try...catch` block. If an error occurs in a statement in the `try` block, execution transfers to the `catch` block where the transaction can be rolled back via the `SqlTransaction` object s `Rollback` method. If all of the statements complete successfully, a call to the `SqlTransaction` object s `Commit` method at the end of the `try` block commits the transaction. The following code snippet illustrates this pattern. See Maintaining Database Consistency with Transactions for additional syntax and examples of using transactions with ADO.NET.

```
' Create the SqlTransaction object
Dim myTransaction As SqlTransaction = SqlConnectionObject.BeginTransaction();
Try
    '
    ' ... Perform the database transaction's data modification statements...
    '
    ' If we reach here, no errors, so commit the transaction
    myTransaction.Commit()
Catch
    ' If we reach here, there was an error, so rollback the transaction
    myTransaction.Rollback()
    Throw
End Try
```

By default, the TableAdapters in a Typed DataSet do not use transactions. To provide support for transactions we need to augment the TableAdapter classes to include additional methods that use the above pattern to perform a series of data modification statements within the scope of a transaction. In Step 2 we'll see how to use partial classes to add these methods.

Step 1: Creating the Working with Batched Data Web Pages

Before we start exploring how to augment the DAL to support database transactions, let s first take a moment to create the ASP.NET web pages that we will need for this tutorial and the three that follow. Start by adding a new folder named `BatchData` and then add the following ASP.NET pages, associating each page with the `Site.master` master page.

- `Default.aspx`
- `Transactions.aspx`
- `BatchUpdate.aspx`
- `BatchDelete.aspx`
- `BatchInsert.aspx`

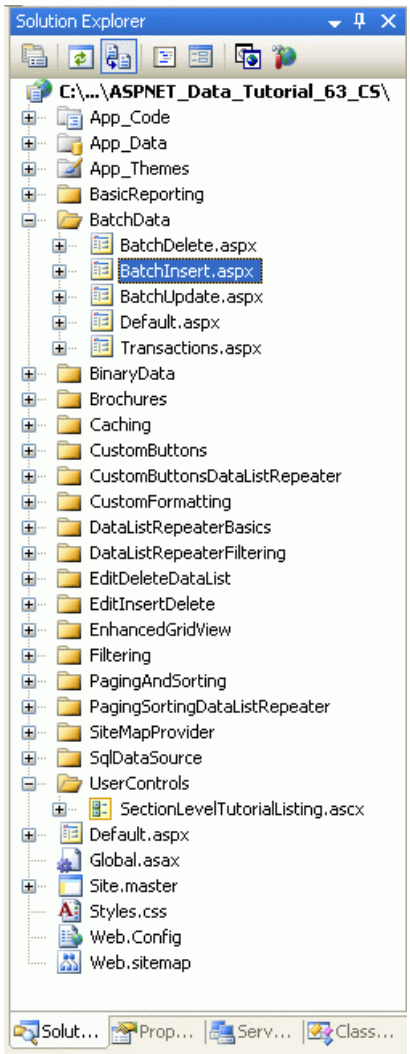


Figure 1: Add the ASP.NET Pages for the SqlDataSource-Related Tutorials

As with the other folders, `Default.aspx` will use the `SectionLevelTutorialListing.ascx` User Control to list the tutorials within its section. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

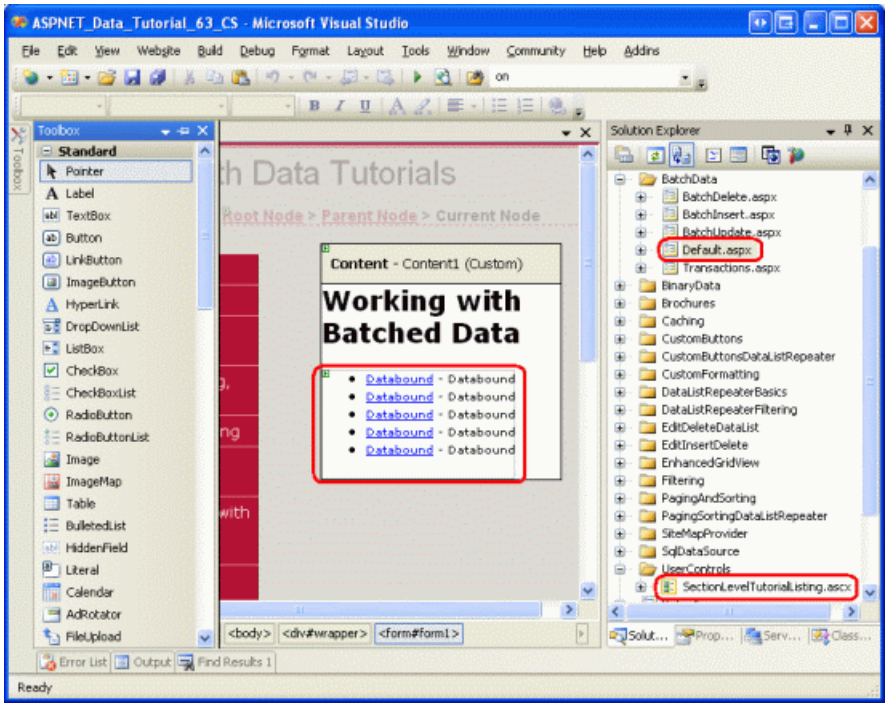


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size

image)

Lastly, add these four pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Customizing the Site Map `<siteMapNode>`:

```
<siteMapNode title="Working with Batched Data"
    url="~/BatchData/Default.aspx"
    description="Learn how to perform batch operations as opposed to
                 per-row operations.">

    <siteMapNode title="Adding Support for Transactions"
        url="~/BatchData/Transactions.aspx"
        description="See how to extend the Data Access Layer to support
                     database transactions." />
    <siteMapNode title="Batch Updating"
        url="~/BatchData/BatchUpdate.aspx"
        description="Build a batch updating interface, where each row in a
                      GridView is editable." />
    <siteMapNode title="Batch Deleting"
        url="~/BatchData/BatchDelete.aspx"
        description="Explore how to create an interface for batch deleting
                     by adding a CheckBox to each GridView row." />
    <siteMapNode title="Batch Inserting"
        url="~/BatchData/BatchInsert.aspx"
        description="Examine the steps needed to create a batch inserting
                     interface, where multiple records can be created at the
                     click of a button." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the working with batched data tutorials.

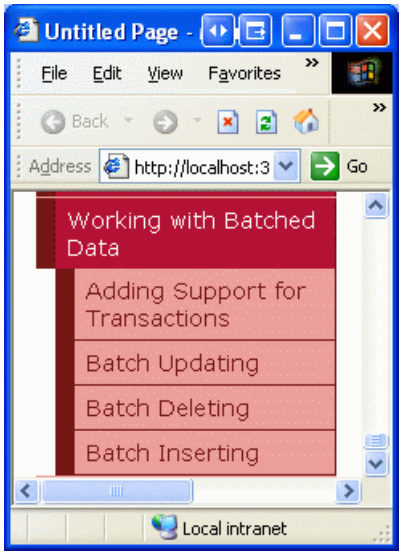


Figure 3: The Site Map Now Includes Entries for the Working with Batched Data Tutorials

Step 2: Updating the Data Access Layer to Support Database Transactions

As we discussed back in the first tutorial, Creating a Data Access Layer, the Typed DataSet in our DAL is composed of DataTables and TableAdapters. The DataTables hold data while the TableAdapters provide the functionality to read data from the database into the DataTables, to update the database with changes made to the DataTables, and so forth. Recall that the TableAdapters provide two patterns for updating data, which I referred to as Batch Update and DB-Direct. With the Batch Update pattern, the TableAdapter is passed a DataSet, DataTable, or collection of DataRows. This data is enumerated and for each inserted, modified, or deleted row, the

`InsertCommand`, `UpdateCommand`, or `DeleteCommand` is executed. With the DB-Direct pattern, the TableAdapter is instead passed the values of the columns necessary for inserting, updating, or deleting a single record. The DB Direct pattern method then uses those passed-in values to execute the appropriate `InsertCommand`, `UpdateCommand`, or `DeleteCommand` statement.

Regardless of the update pattern used, the TableAdapters auto-generated methods do not use transactions. By default each insert, update, or delete performed by the TableAdapter is treated as a single discrete operation. For instance, imagine that the DB-Direct pattern is used by some code in the BLL to insert ten records into the database. This code would call the TableAdapter s `Insert` method ten times. If the first five inserts succeed, but the sixth one resulted in an exception, the first five inserted records would remain in the database. Similarly, if the Batch Update pattern is used to perform inserts, updates, and deletes to the inserted, modified, and deleted rows in a DataTable, if the first several modifications succeeded but a later one encountered an error, those earlier modifications that completed would remain in the database.

In certain scenarios we want to ensure atomicity across a series of modifications. To accomplish this we must manually extend the TableAdapter by adding new methods that execute the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` s under the umbrella of a transaction. In Creating a Data Access Layer we looked at using partial classes to extend the functionality of the DataTables within the Typed DataSet. This technique can also be used with TableAdapters.

The Typed DataSet `Northwind.xsd` is located in the `App_Code` folder s `DAL` subfolder. Create a subfolder in the `DAL` folder named `TransactionSupport` and add a new class file named `ProductsTableAdapter.TransactionSupport.vb` (see Figure 4). This file will hold the partial implementation of the `ProductsTableAdapter` that includes methods for performing data modifications using a transaction.

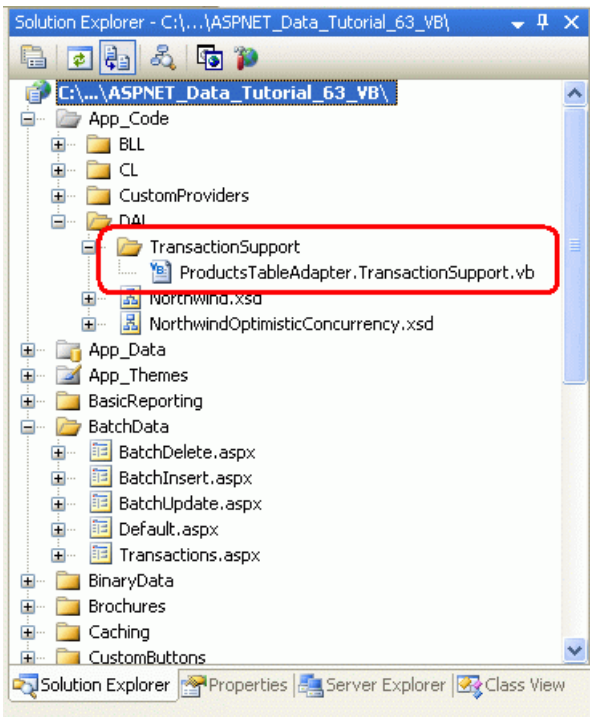


Figure 4: Add a Folder Named `TransactionSupport` and a Class File Named `ProductsTableAdapter.TransactionSupport.vb`

Enter the following code into the `ProductsTableAdapter.TransactionSupport.vb` file:

```
Imports System.Data
Imports System.Data.SqlClient
Namespace NorthwindTableAdapters
    Partial Public Class ProductsTableAdapter
        Private _transaction As SqlTransaction
        Private Property Transaction() As SqlTransaction
            Get
                Return Me._transaction
            End Get
            Set(ByVal Value As SqlTransaction)
                Me._transaction = Value
            End Set
        End Property
        Public Sub BeginTransaction()
            ' Open the connection, if needed
            If Me.Connection.State <> ConnectionState.Open Then
                Me.Connection.Open()
            End If
            ' Create the transaction and assign it to the Transaction property
            Me.Transaction = Me.Connection.BeginTransaction()
            ' Attach the transaction to the Adapters
            For Each command As SqlCommand In Me.CommandCollection
                command.Transaction = Me.Transaction
            Next
            Me.Adapter.InsertCommand.Transaction = Me.Transaction
            Me.Adapter.UpdateCommand.Transaction = Me.Transaction
            Me.Adapter.DeleteCommand.Transaction = Me.Transaction
        End Sub
        Public Sub CommitTransaction()
            ' Commit the transaction
            Me.Transaction.Commit()
            ' Close the connection
            Me.Connection.Close()
        End Sub
        Public Sub RollbackTransaction()
            ' Rollback the transaction
            Me.Transaction.Rollback()
            ' Close the connection
            Me.Connection.Close()
        End Sub
    End Class
End Namespace
```

The `Partial` keyword in the class declaration here indicates to the compiler that the members added within are to be added to the `ProductsTableAdapter` class in the `NorthwindTableAdapters` namespace. Note the `Imports System.Data.SqlClient` statement at the top of the file. Since the TableAdapter was configured to use the SqlClient provider, internally it uses a `SqlDataAdapter` object to issue its commands to the database. Consequently, we need to use the `SqlTransaction` class to begin the transaction and then to commit it or roll it back. If you are using a data store other than Microsoft SQL Server, you'll need to use the appropriate provider.

These methods provide the building blocks needed to start, rollback, and commit a transaction. They are marked `Public`, enabling them to be used from within the `ProductsTableAdapter`, from another class in the DAL, or from another layer in the architecture, such as the BLL. `BeginTransaction` opens the TableAdapter s internal `SqlConnection` (if needed), begins the transaction and assigns it to the `Transaction` property, and attaches the transaction to the internal `SqlDataAdapter` s `SqlCommand` objects. `CommitTransaction` and `RollbackTransaction` call the `Transaction` object s `Commit` and `Rollback` methods, respectively, before closing the internal `Connection` object.

Step 3: Adding Methods to Update and Delete Data Under the Umbrella of a Transaction

With these methods complete, we re ready to add methods to `ProductsDataTable` or the BLL that perform a series of commands under the umbrella of a transaction. The following method uses the Batch Update pattern to update a `ProductsDataTable` instance using a transaction. It starts a transaction by calling the `BeginTransaction` method and then uses a `Try...Catch` block to issue the data modification statements. If the call to the `Adapter` object s `Update` method results in an exception, execution will transfer to the `catch` block where the transaction will be rolled back and the exception re-thrown. Recall that the `Update` method implements the Batch Update pattern by enumerating the rows of the supplied `ProductsDataTable` and performing the necessary `InsertCommand`, `UpdateCommand`, and `DeleteCommand` s. If any one of these commands results in an error, the transaction is rolled back, undoing the previous modifications made during the transaction s lifetime. Should the `Update` statement complete without error, the transaction is committed in its entirety.

```
Public Function UpdateWithTransaction _
    (ByVal dataTable As Northwind.ProductsDataTable) As Integer

    Me.BeginTransaction()
    Try
        ' Perform the update on the DataTable
        Dim returnValue As Integer = Me.Adapter.Update(dataTable)
        ' If we reach here, no errors, so commit the transaction
        Me.CommitTransaction()
        Return returnValue
    Catch
        ' If we reach here, there was an error, so rollback the transaction
        Me.RollbackTransaction()
        Throw
    End Try
End Function
```

Add the `UpdateWithTransaction` method to the `ProductsTableAdapter` class through the partial class in `ProductsTableAdapter.TransactionSupport.vb`. Alternatively, this method could be added to the Business Logic Layer s `ProductsBLL` class with a few minor syntactical changes. Namely, the keyword `Me` in `Me.BeginTransaction()`, `Me.CommitTransaction()`, and `Me.RollbackTransaction()` would need to be replaced with `Adapter` (recall that `Adapter` is the name of a property in `ProductsBLL` of type `ProductsTableAdapter`).

The `UpdateWithTransaction` method uses the Batch Update pattern, but a series of DB-Direct calls can also be used within the scope of a transaction, as the following method shows. The `DeleteProductsWithTransaction` method accepts as input a `List(Of T)` of type `Integer`, which are the `ProductID` s to delete. The method initiates the transaction via a call to `BeginTransaction` and then, in the `Try` block, iterates through the supplied list calling the DB-Direct pattern `Delete` method for each `ProductID` value. If any of the calls to `Delete` fails, control is transferred to the `Catch` block where the transaction is rolled back and the exception re-thrown. If all calls to `Delete` succeed, then transaction is committed. Add this method to the `ProductsBLL` class.

```
Public Sub DeleteProductsWithTransaction _
    (ByVal productIDs As System.Collections.Generic.List(Of Integer))

    ' Start the transaction
    Adapter.BeginTransaction()
    Try
        ' Delete each product specified in the list
        For Each productID As Integer In productIDs
            Adapter.Delete(productID)
        Next
        ' Commit the transaction
        Adapter.CommitTransaction()
    Catch
        ' There was an error - rollback the transaction
        Adapter.RollbackTransaction()
        Throw
    End Try
End Sub
```

Applying Transactions Across Multiple TableAdapters

The transaction-related code examined in this tutorial allows for multiple statements against the `ProductsTableAdapter` to be treated as an atomic operation. But what if multiple modifications to different database tables need to be performed atomically? For instance, when deleting a category, we might first want to reassign its current products to some other category. These two steps reassigning the products and deleting the category should be executed as an atomic operation. But the `ProductsTableAdapter` includes only methods for modifying the `Products` table and the `CategoriesTableAdapter` includes only methods for modifying the `Categories` table. So how can a transaction encompass both TableAdapters?

One option is to add a method to the `CategoriesTableAdapter` named `DeleteCategoryAndReassignProducts(categoryIDtoDelete, reassignToCategoryID)` and have that method call a stored procedure that both reassigns the products and deletes the category within the scope of a transaction defined within the stored procedure. We'll look at how to begin, commit, and rollback transactions in stored procedures in a future tutorial.

Another option is to create a helper class in the DAL that contains the `DeleteCategoryAndReassignProducts(categoryIDtoDelete, reassignToCategoryID)` method. This method would create an instance of the `CategoriesTableAdapter` and the `ProductsTableAdapter` and then set these two TableAdapters `Connection` properties to the same `SqlConnection` instance. At that point, either one of the two TableAdapters would initiate the transaction with a call to `BeginTransaction`. The TableAdapters methods for reassigning the products and deleting the category would be invoked in a `Try...Catch` block with the transaction committed or rolled back as needed.

Step 4: Adding the `UpdateWithTransaction` Method to the Business Logic Layer

In Step 3 we added an `UpdateWithTransaction` method to the `ProductsTableAdapter` in the DAL. We should add a corresponding method to the BLL. While the Presentation Layer could call directly down to the DAL to invoke the `UpdateWithTransaction` method, these tutorials have strived to define a layered architecture that insulates the DAL from the Presentation Layer. Therefore, it behooves us to continue this approach.

Open the `ProductsBLL` class file and add a method named `UpdateWithTransaction` that simply calls down to the corresponding DAL method. There should now be two new methods in `ProductsBLL`: `UpdateWithTransaction`, which you just added, and `DeleteProductsWithTransaction`, which was added in Step 3.

```
Public Function UpdateWithTransaction _
    (ByVal products As Northwind.ProductsDataTable) As Integer

    Return Adapter.UpdateWithTransaction(products)
End Function
Public Sub DeleteProductsWithTransaction _
    (ByVal productIDs As System.Collections.Generic.List(Of Integer))

    ' Start the transaction
    Adapter.BeginTransaction()
    Try
        ' Delete each product specified in the list
        For Each productID As Integer In productIDs
            Adapter.Delete(productID)
        Next
        ' Commit the transaction
        Adapter.CommitTransaction()
    Catch
        ' There was an error - rollback the transaction
        Adapter.RollbackTransaction()
        Throw
    End Try
End Sub
```

NOTE

These methods do not include the `DataObjectMethodAttribute` attribute assigned to most other methods in the `ProductsBLL` class because we'll be invoking these methods directly from the ASP.NET pages code-behind classes. Recall that `DataObjectMethodAttribute` is used to flag what methods should appear in the ObjectDataSource s Configure Data Source wizard and under what tab (SELECT, UPDATE, INSERT, or DELETE). Since the GridView lacks any built-in support for batch editing or deleting, we'll have to invoke these methods programmatically rather than use the code-free declarative approach.

Step 5: Atomically Updating Database Data from the Presentation Layer

To illustrate the effect that the transaction has when updating a batch of records, let s create a user interface that lists all products in a GridView and includes a Button Web control that, when clicked, reassigns the products `CategoryID` values. In particular, the category reassignment will progress so that the first several products are assigned a valid `CategoryID` value while others are purposefully assigned a non-existent `CategoryID` value. If we attempt to update the database with a product whose `CategoryID` does not match an existing category s `CategoryID`, a foreign key constraint violation will occur and an exception will be raised. What we'll see in this example is that when using a transaction the exception raised from the foreign key constraint violation will cause the previous valid `CategoryID` changes to be rolled back. When not using a transaction, however, the modifications to the initial categories will remain.

Start by opening the `Transactions.aspx` page in the `BatchData` folder and drag a GridView from the Toolbox onto the Designer. Set its `ID` to `Products` and, from its smart tag, bind it to a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to pull its data from the `ProductsBLL` class s `GetProducts` method. This will be a read-only GridView, so set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) and click Finish.

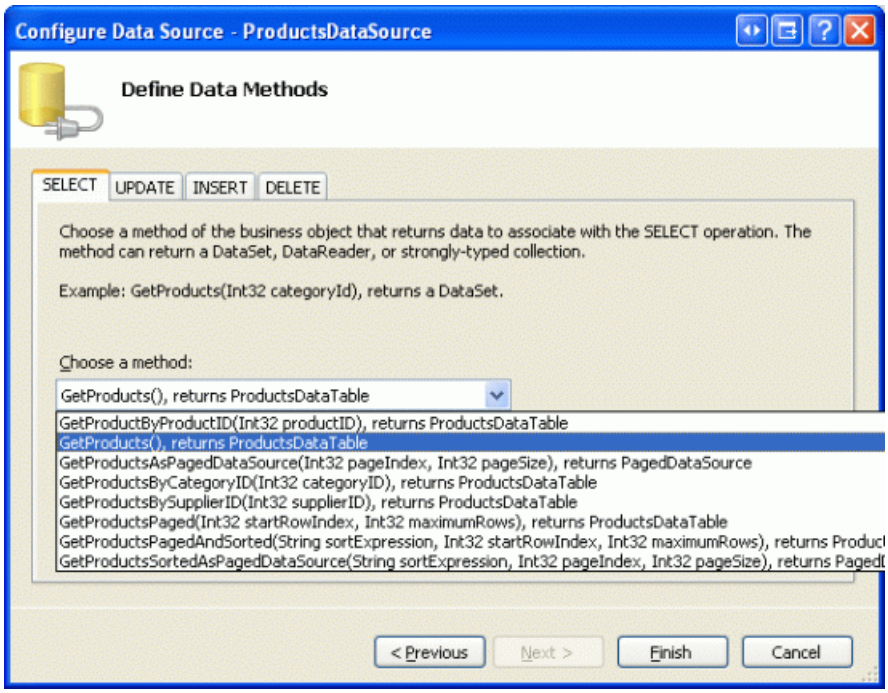


Figure 5: Configure the ObjectDataSource to Use the `ProductsBLL` Class s `GetProducts` Method (Click to view full-size image)

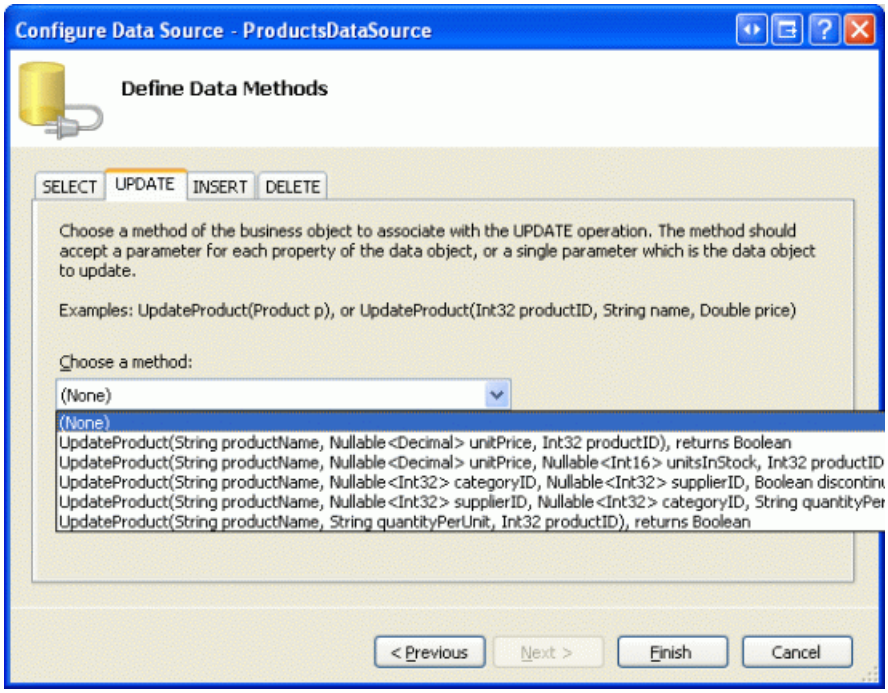


Figure 6: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE Tabs to (None) (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will create BoundFields and a CheckBoxField for the product data fields. Remove all of these fields except for `ProductID`, `ProductName`, `CategoryID`, and `CategoryName` and rename the `ProductName` and `CategoryName` BoundFields `HeaderText` properties to Product and Category, respectively. From the smart tag, check the Enable Paging option. After making these modifications, the GridView and ObjectDataSource s declarative markup should look like the following:

```
<asp:GridView ID="Products" runat="server" AllowPaging="True"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ProductsDataSource">
    <Columns>
        <asp:BoundField DataField="ProductID" HeaderText="ProductID"
            InsertVisible="False" ReadOnly="True"
            SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="CategoryID" HeaderText="CategoryID"
            SortExpression="CategoryID" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            SortExpression="CategoryName" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
</asp:ObjectDataSource>
```

Next, add three Button Web controls above the GridView. Set the first Button s Text property to Refresh Grid, the second s to Modify Categories (WITH TRANSACTION), and the third one s to Modify Categories (WITHOUT TRANSACTION) .

```
<p>
    <asp:Button ID="RefreshGrid" runat="server" Text="Refresh Grid" />
</p>
<p>
    <asp:Button ID="ModifyCategoriesWithTransaction" runat="server"
        Text="Modify Categories (WITH TRANSACTION)" />
</p>
<p>
    <asp:Button ID="ModifyCategoriesWithoutTransaction" runat="server"
        Text="Modify Categories (WITHOUT TRANSACTION)" />
</p>
```

At this point the Design view in Visual Studio should look similar to the screen shot shown in Figure 7.

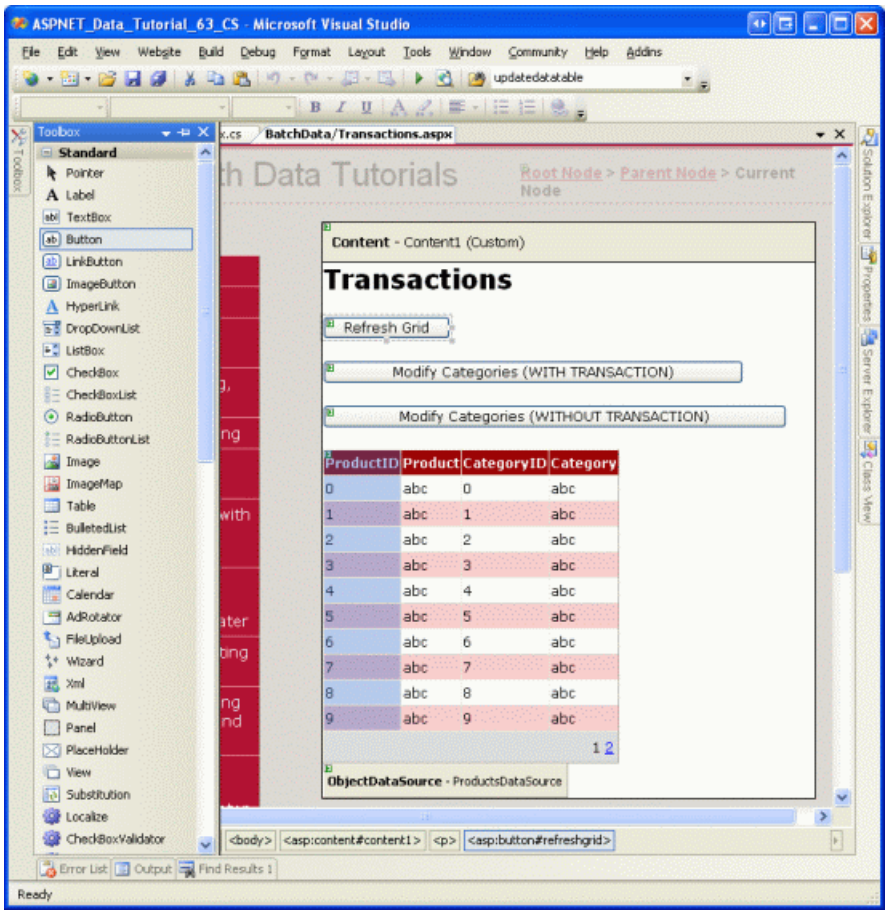


Figure 7: The Page Contains a GridView and Three Button Web Controls (Click to view full-size image)

Create event handlers for each of the three Button s `Click` events and use the following code:

```
Protected Sub RefreshGrid_Click _
    (ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles RefreshGrid.Click

    Products.DataBind()
End Sub
Protected Sub ModifyCategoriesWithTransaction_Click _
    (ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles ModifyCategoriesWithTransaction.Click

    ' Get the set of products
    Dim productsAPI As New ProductsBLL()
    Dim productsData As Northwind.ProductsDataTable = productsAPI.GetProducts()
    ' Update each product's CategoryID
    For Each product As Northwind.ProductsRow In productsData
        product.CategoryID = product.ProductID
    Next
    ' Update the data using a transaction
    productsAPI.UpdateWithTransaction(productsData)
    ' Refresh the Grid
    Products.DataBind()
End Sub
Protected Sub ModifyCategoriesWithoutTransaction_Click _
    (ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles ModifyCategoriesWithoutTransaction.Click

    ' Get the set of products
    Dim productsAPI As New ProductsBLL()
    Dim productsData As Northwind.ProductsDataTable = productsAPI.GetProducts()
    ' Update each product's CategoryID
    For Each product As Northwind.ProductsRow In productsData
        product.CategoryID = product.ProductID
    Next
    ' Update the data WITHOUT using a transaction
    Dim productsAdapter As New NorthwindTableAdapters.ProductsTableAdapter()
    productsAdapter.Update(productsData)
    ' Refresh the Grid
    Products.DataBind()
End Sub
```

The refresh Button s `Click` event handler simply rebinds the data to the GridView by calling the `Products` GridView s `DataBind` method.

The second event handler reassigns the products `CategoryID` s and uses the new transaction method from the BLL to perform the database updates under the umbrella of a transaction. Note that each product s `CategoryID` is arbitrarily set to the same value as its `ProductID`. This will work fine for the first few products, since those products have `ProductID` values that happen to map to valid `CategoryID` s. But once the `ProductID` s start getting too large, this coincidental overlap of `ProductID` s and `CategoryID` s no longer applies.

The third `Click` event handler updates the products `CategoryID` s in the same manner, but sends the update to the database using the `ProductsTableAdapter` s default `Update` method. This `Update` method does not wrap the series of commands within a transaction, so those changes are made prior to the first encountered foreign key constraint violation error will persist.

To demonstrate this behavior, visit this page through a browser. Initially you should see the first page of data as shown in Figure 8. Next, click the Modify Categories (WITH TRANSACTION) button. This will cause a postback and attempt to update all of the products `CategoryID` values, but will result in a foreign key constraint violation (see Figure 9).

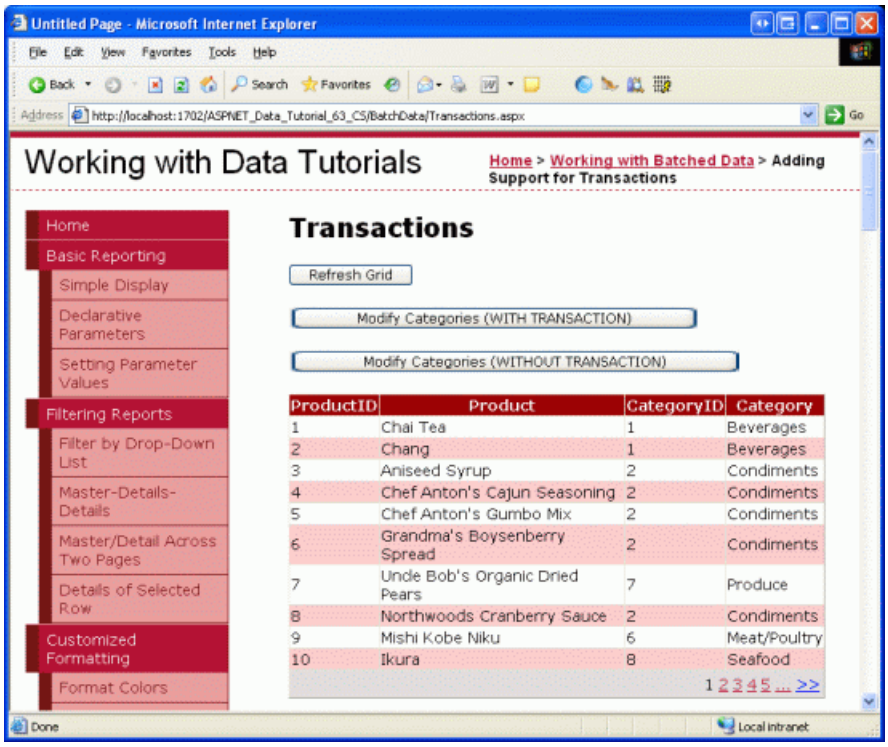


Figure 8: The Products are Displayed in a Pageable GridView (Click to view full-size image)

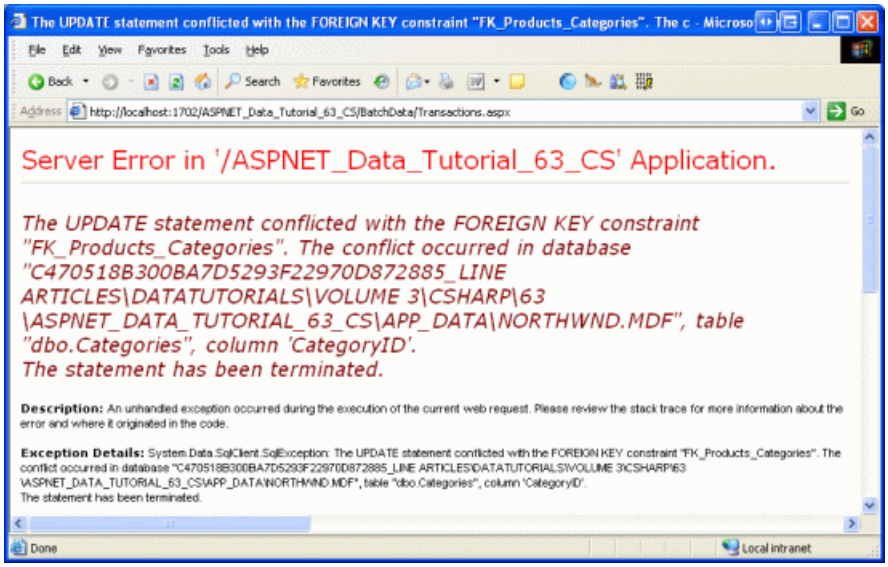


Figure 9: Reassigning the Categories Results in a Foreign Key Constraint Violation (Click to view full-size image)

Now hit your browser s Back button and then click the Refresh Grid button. Upon refreshing the data you should see the exact same output as shown in Figure 8. That is, even though some of the products `CategoryID` s were changed to legal values and updated in the database, they were rolled back when the foreign key constraint violation occurred.

Now try clicking the Modify Categories (WITHOUT TRANSACTION) button. This will result in the same foreign key constraint violation error (see Figure 9), but this time those products whose `CategoryID` values were changed to a legal value will not be rolled back. Hit your browser s Back button and then the Refresh Grid button. As Figure 10 shows, the `CategoryID` s of the first eight products have been reassigned. For example, in Figure 8, Chang had a `CategoryID` of 1, but in Figure 10 it s been reassigned to 2.

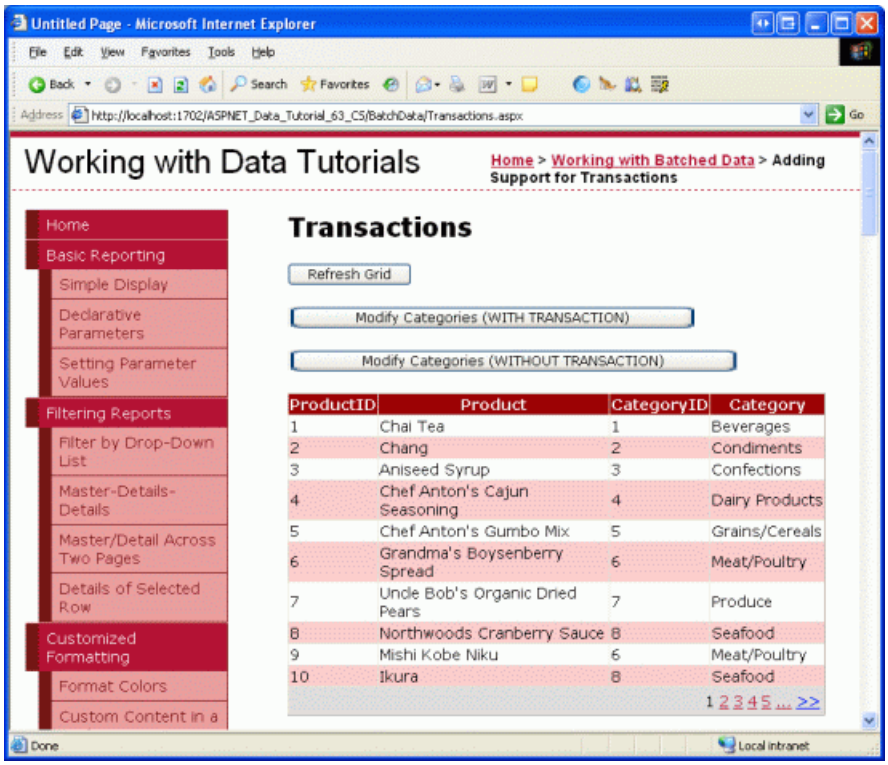


Figure 10: Some Products `CategoryID` Values were Updated While Others Were Not (Click to view full-size image)

Summary

By default, the TableAdapter s methods do not wrap the executed database statements within the scope of a transaction, but with a little work we can add methods that will create, commit, and rollback a transaction. In this tutorial we created three such methods in the `ProductsTableAdapter` class: `BeginTransaction`, `CommitTransaction`, and `RollbackTransaction`. We saw how to use these methods along with a `Try...Catch` block to make a series of data modification statements atomic. In particular, we created the `UpdateWithTransaction` method in the `ProductsTableAdapter`, which uses the Batch Update pattern to perform the necessary modifications to the rows of a supplied `ProductsDataTable`. We also added the `DeleteProductsWithTransaction` method to the `ProductsBLL` class in the BLL, which accepts a `List` of `ProductID` values as its input and calls the DB-Direct pattern method `Delete` for each `ProductID`. Both methods start by creating a transaction and then executing the data modification statements within a `Try...Catch` block. If an exception occurs, the transaction is rolled back, otherwise it is committed.

Step 5 illustrated the effect of transactional batch updates versus batch updates that neglected to use a transaction. In the next three tutorials we will build upon the foundation laid in this tutorial and create user interfaces for performing batch updates, deletes, and inserts.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Maintaining Database Consistency with Transactions
- Managing Transactions in SQL Server Stored Procedures
- Transactions Made Easy: `System.Transactions`
- TransactionScope and DataAdapters
- Using Oracle Database Transactions in .NET

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Dave Gardner, Hilton Giesenow, and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Batch Updating (VB)

2/20/2020 • 21 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Learn how to update multiple database records in a single operation. In the User Interface Layer we build a GridView where each row is editable. In the Data Access Layer we wrap the multiple Update operations within a transaction to ensure that all updates succeed or all updates are rolled back.

Introduction

In the preceding tutorial we saw how to extend the Data Access Layer to add support for database transactions. Database transactions guarantee that a series of data modification statements will be treated as one atomic operation, which ensures that all modifications will fail or all will succeed. With this low-level DAL functionality out of the way, we re ready to turn our attention to creating batch data modification interfaces.

In this tutorial we'll build a GridView where each row is editable (see Figure 1). Since each row is rendered in its editing interface, there s no need for a column of Edit, Update, and Cancel buttons. Instead, there are two Update Products buttons on the page that, when clicked, enumerate the GridView rows and update the database.

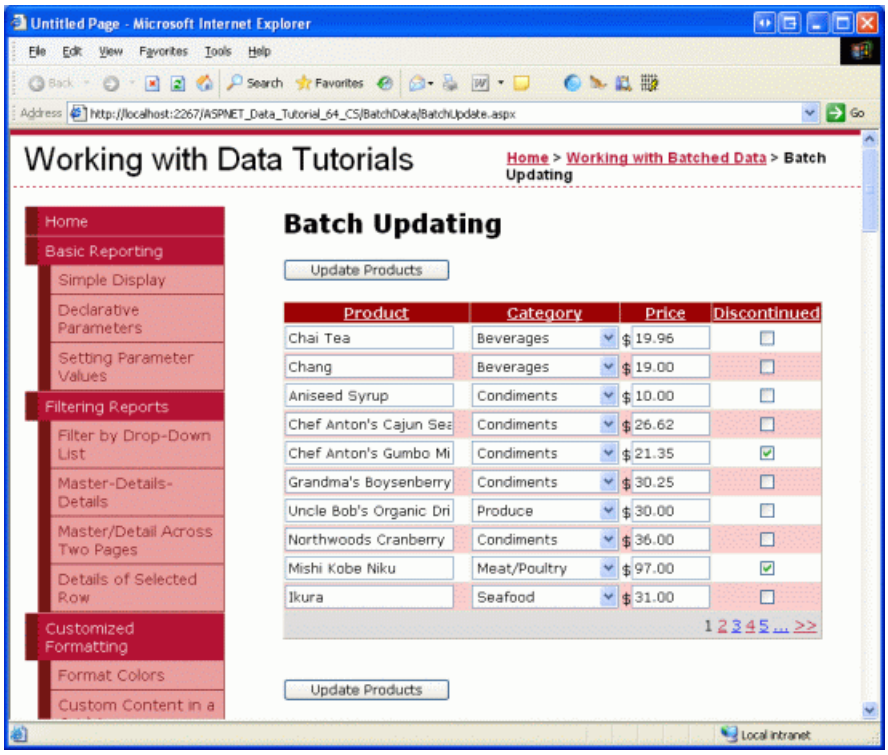


Figure 1: Each Row in the GridView is Editable (Click to view full-size image)

Let s get started!

> **NOTE**
> In the Performing Batch Updates tutorial we created a batch editing interface using the DataList control. This tutorial differs from the previous one in that is uses a GridView and the batch update is performed within the scope of a transaction. After completing this tutorial I encourage you to return to the earlier tutorial and update it to use the database transaction-related functionality added in the preceding tutorial.

Examining the Steps for Making All GridView Rows Editable

As discussed in the An Overview of Inserting, Updating, and Deleting Data tutorial, the GridView offers built-in support for editing its underlying data on a per-row basis. Internally, the GridView notes what row is editable through its `EditIndex` property. As the GridView is being bound to its data source, it checks each row to see if the index of the row equals the value of `EditIndex`. If so, that row s fields are rendered using their editing interfaces. For BoundFields, the editing interface is a TextBox whose `Text` property is assigned the value of the data field specified by the BoundField s `DataField` property. For TemplateFields, the `EditItemTemplate` is used in place of the `ItemTemplate`.

Recall that the editing workflow starts when a user clicks a row s Edit button. This causes a postback, sets the GridView s `EditIndex` property to the clicked row s index, and rebinds the data to the grid. When a row s Cancel button is clicked, on postback the `EditIndex` is set to a value of `-1` before rebinding the data to the grid. Since the GridView s rows start indexing at zero, setting `EditIndex` to `-1` has the effect of displaying the GridView in read-only mode.

The `EditIndex` property works well for per-row editing, but is not designed for batch editing. To make the entire GridView editable, we need to have each row render using its editing interface. The easiest way to accomplish this is to create where each editable field is implemented as a TemplateField with its editing interface defined in the `ItemTemplate`.

Over the next several steps we'll create a completely editable GridView. In Step 1 we'll start by creating the GridView and its ObjectDataSource and convert its BoundFields and CheckBoxField into TemplateFields. In Steps 2 and 3 we'll move the editing interfaces from the TemplateFields `EditItemTemplate` s to their `ItemTemplate` s.

Step 1: Displaying Product Information

Before we worry about creating a GridView where are rows are editable, let s start by simply displaying the product information. Open the `BatchUpdate.aspx` page in the `BatchData` folder and drag a GridView from the Toolbox onto the Designer. Set the GridView s `ID` to `ProductsGrid` and, from its smart tag, choose to bind it to a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to retrieve its data from the `ProductsBLL` class s `GetProducts` method.

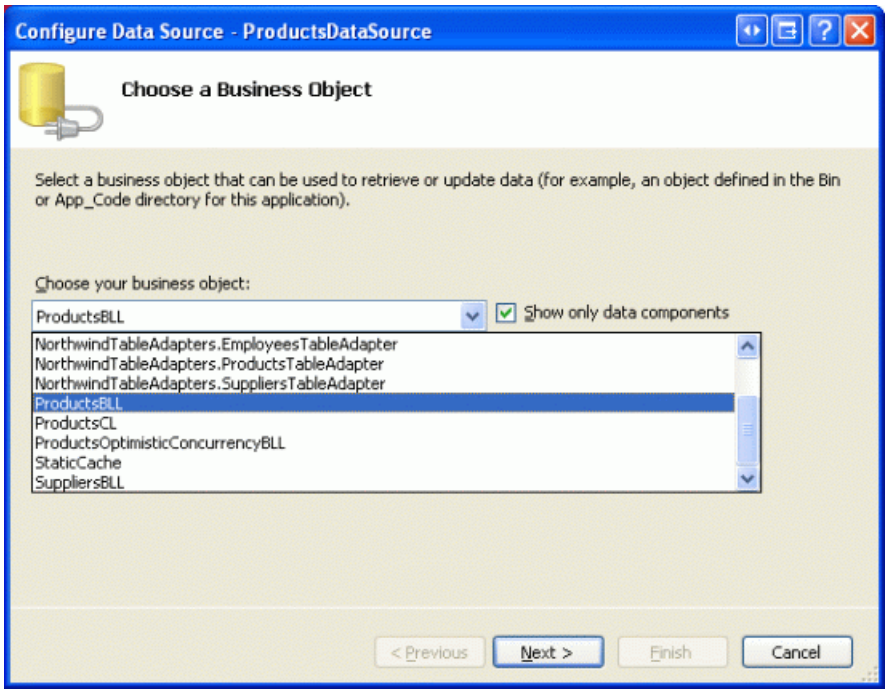


Figure 2: Configure the ObjectDataSource to Use the `ProductsBLL` Class (Click to view full-size image)

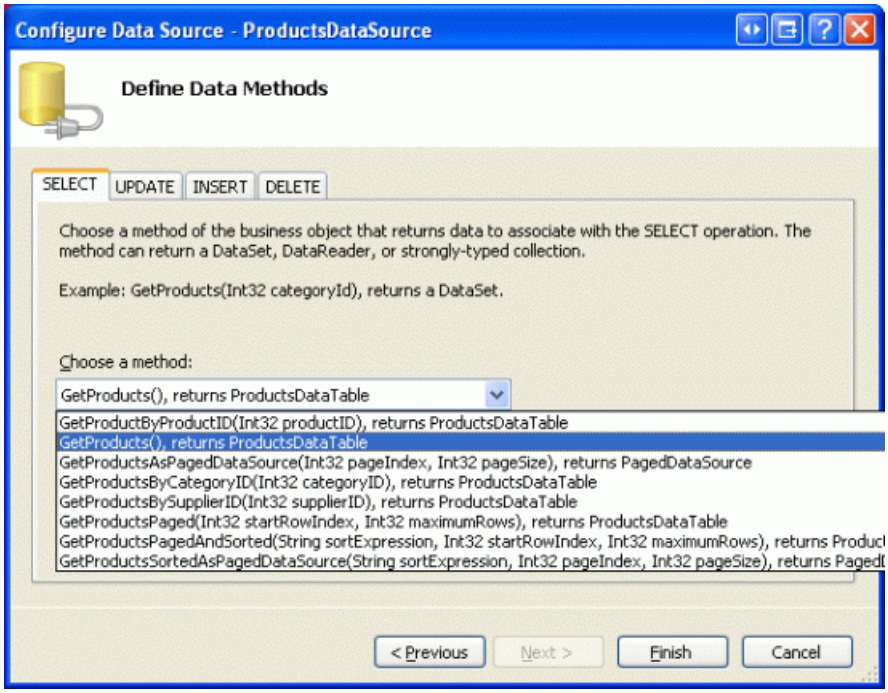


Figure 3: Retrieve the Product Data Using the `GetProducts` Method (Click to view full-size image)

Like the GridView, the ObjectDataSource s modification features are designed to work on a per-row basis. In order to update a set of records, we'll need to write a bit of code in the ASP.NET page s code-behind class that batches the data and passes it to the BLL. Therefore, set the drop-down lists in the ObjectDataSource s UPDATE, INSERT, and DELETE tabs to (None). Click Finish to complete the wizard.

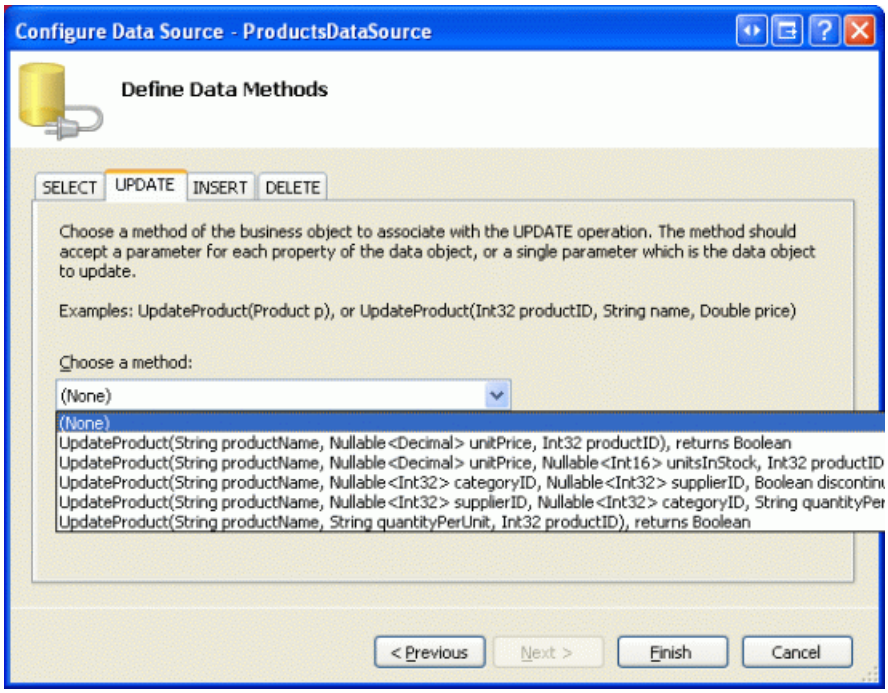


Figure 4: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE Tabs to (None) (Click to view full-size image)

After completing the Configure Data Source wizard, the ObjectDataSource s declarative markup should look like the following:

```
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProducts" TypeName="ProductsBLL">
</asp:ObjectDataSource>
```

Completing the Configure Data Source wizard also causes Visual Studio to create BoundFields and a CheckBoxField for the product data fields in the GridView. For this tutorial, let s only allow the user to view and edit the product s name, category, price, and discontinued status. Remove all but the `ProductName`, `CategoryName`, `UnitPrice`, and `Discontinued` fields and rename the `HeaderText` properties of the first three fields to Product, Category, and Price, respectively. Lastly, check the Enable Paging and Enable Sorting checkboxes in the GridView s smart tag.

At this point the GridView has three BoundFields (`ProductName`, `CategoryName`, and `UnitPrice`) and a CheckBoxField (`Discontinued`). We need to convert these four fields into TemplateFields and then move the editing interface from the TemplateField s `EditItemTemplate` to its `ItemTemplate`.

> **NOTE**
>
> We explored creating and customizing TemplateFields in the Customizing the Data Modification Interface tutorial. We'll walk through the steps of converting the BoundFields and CheckBoxField into TemplateFields and defining their editing interfaces in their `ItemTemplate` s, but if you get stuck or need a refresher, don t hesitate to refer back to this earlier tutorial.

From the GridView s smart tag, click the Edit Columns link to open the Fields dialog box. Next, select each field and click the Convert this field into a TemplateField link.

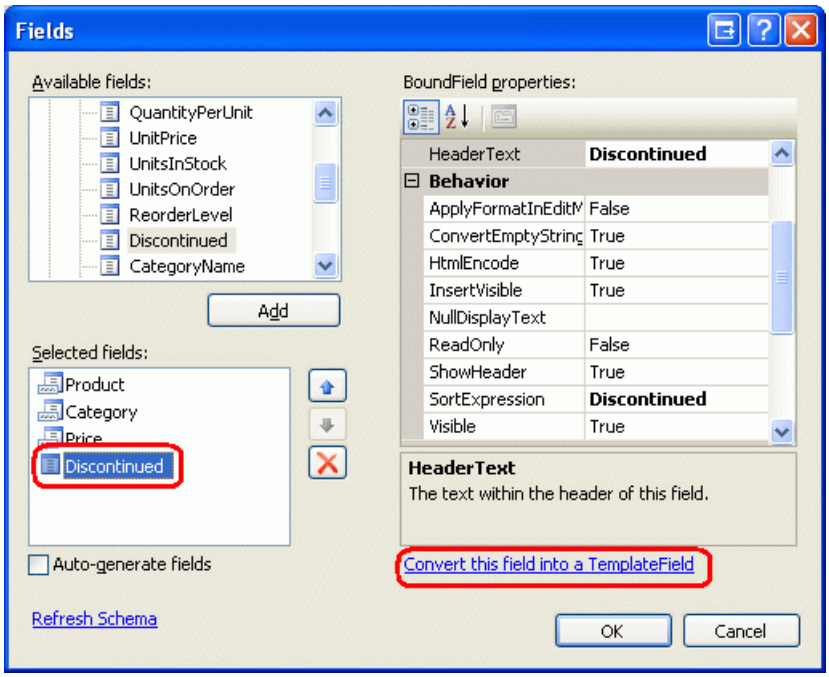


Figure 5: Convert the Existing BoundFields and CheckBoxField Into TemplateFields

Now that each field is a TemplateField, we re ready to move the editing interface from the `EditItemTemplate` s to the `ItemTemplate` s.

Step 2: Creating the `ProductName` , `UnitPrice` , and `Discontinued` Editing Interfaces

Creating the `ProductName` , `UnitPrice` , and `Discontinued` editing interfaces are the topic of this step and are pretty straightforward, as each interface is already defined in the TemplateField s `EditItemTemplate` . Creating the `CategoryName` editing interface is a bit more involved since we need to create a DropDownList of the applicable categories. This `CategoryName` editing interface is tackled in Step 3.

Let s start with the `ProductName` TemplateField. Click on the Edit Templates link from the GridView s smart tag and drill down to the `ProductName` TemplateField s `EditItemTemplate` . Select the TextBox, copy it to the clipboard, and then paste it to the `ProductName` TemplateField s `ItemTemplate` . Change the TextBox s `ID` property to `ProductName` .

Next, add a RequiredFieldValidator to the `ItemTemplate` to ensure that the user provides a value for each product s name. Set the `ControlToValidate` property to ProductName, the `ErrorMessage` property to You must provide the product's name. and the `Text` property to *. After making these additions to the `ItemTemplate` , your screen should look similar to Figure 6.

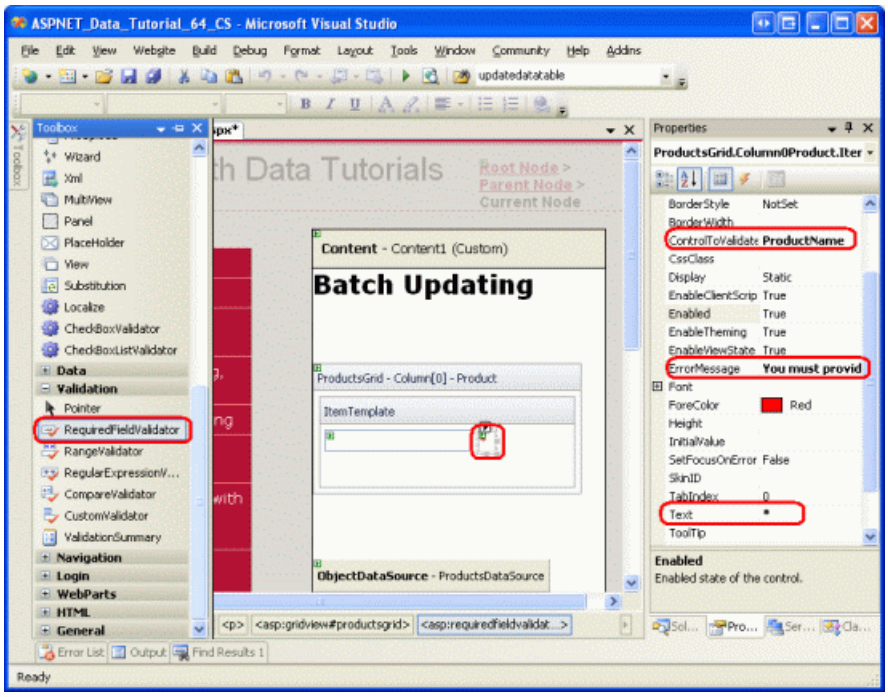


Figure 6: The `ProductName` TemplateField Now Includes a TextBox and a RequiredFieldValidator (Click to view full-size image)

For the `UnitPrice` editing interface, start by copying the TextBox from the `EditItemTemplate` to the `ItemTemplate`. Next, place a $ in front of the TextBox and set its `ID` property to UnitPrice and its `Columns` property to 8 .

Also add a CompareValidator to the `UnitPrice` s `ItemTemplate` to ensure that the value entered by the user is a valid currency value greater than or equal to $0.00. Set the validator s `ControlToValidate` property to UnitPrice, its `ErrorMessage` property to You must enter a valid currency value. Please omit any currency symbols., its `Text` property to *, its `Type` property to `Currency`, its `Operator` property to `GreaterThanEqual`, and its `ValueToCompare` property to 0 .

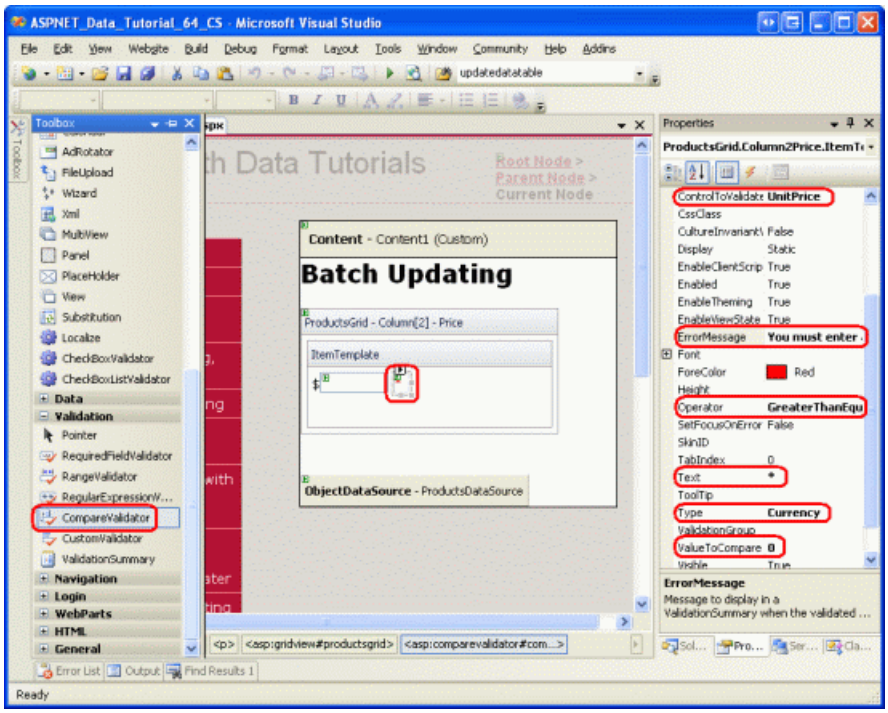


Figure 7: Add a CompareValidator to Ensure the Price Entered is a Non-Negative Currency Value (Click to view full-size image)

For the `Discontinued` TemplateField you can use the CheckBox already defined in the `ItemTemplate`. Simply set its `ID` to Discontinued and its `Enabled` property to `True`.

Step 3: Creating the `CategoryName` Editing Interface

The editing interface in the `CategoryName` TemplateField s `EditItemTemplate` contains a TextBox that displays the value of the `CategoryName` data field. We need to replace this with a DropDownList that lists the possible categories.

> **NOTE**
>
> The Customizing the Data Modification Interface tutorial contains a more thorough and complete discussion on customizing a template to include a DropDownList as opposed to a TextBox. While the steps here are complete, they are presented tersely. For a more in-depth look at creating and configuring the categories DropDownList, refer back to the Customizing the Data Modification Interface tutorial.

Drag a DropDownList from the Toolbox onto the `CategoryName` TemplateField s `ItemTemplate`, setting its `ID` to `Categories`. At this point we would usually define the DropDownLists s data source through its smart tag, creating a new ObjectDataSource. However, this will add the ObjectDataSource within the `ItemTemplate`, which will result in an ObjectDataSource instance created for each GridView row. Instead, let s create the ObjectDataSource outside of the GridView s TemplateFields. End the template editing and drag an ObjectDataSource from the Toolbox onto the Designer beneath the `ProductsDataSource` ObjectDataSource. Name the new ObjectDataSource `CategoriesDataSource` and configure it to use the `CategoriesBLL` class s `GetCategories` method.

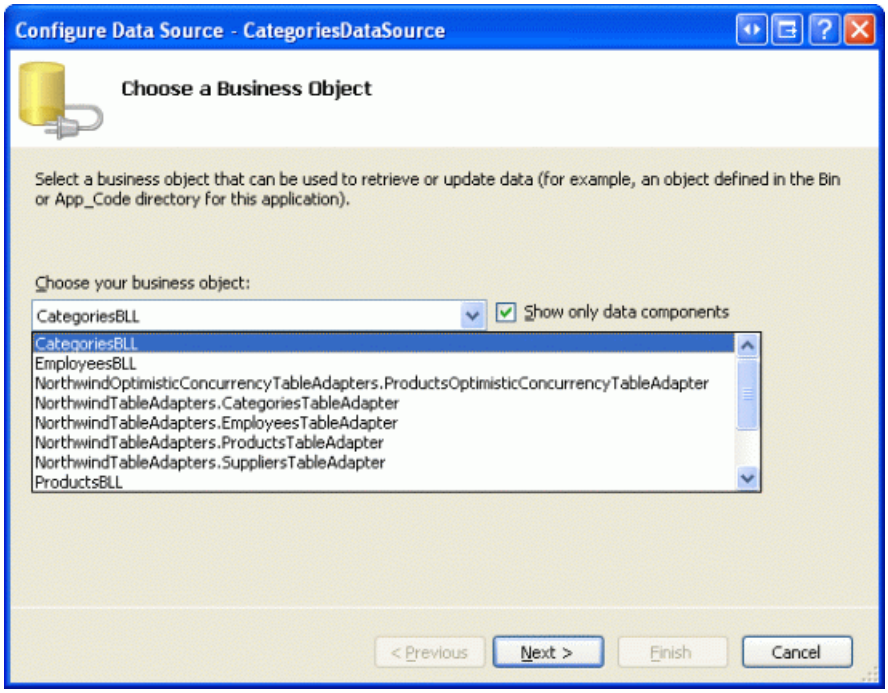


Figure 8: Configure the ObjectDataSource to Use the `CategoriesBLL` Class (Click to view full-size image)

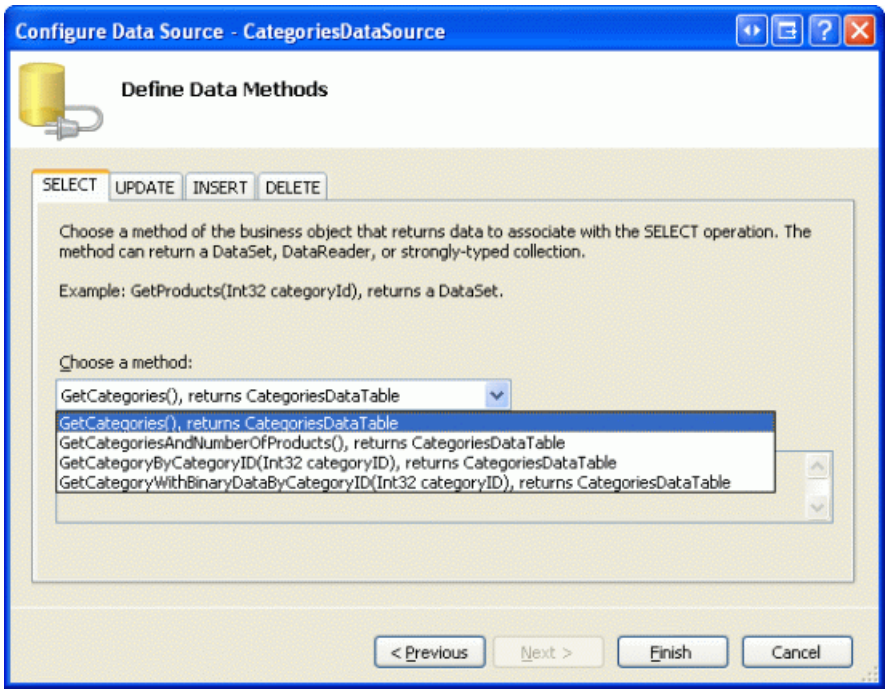


Figure 9: Retrieve the Category Data Using the `GetCategories` Method (Click to view full-size image)

Since this ObjectDataSource is used merely to retrieve data, set the drop-down lists in the UPDATE and DELETE tabs to (None). Click Finish to complete the wizard.

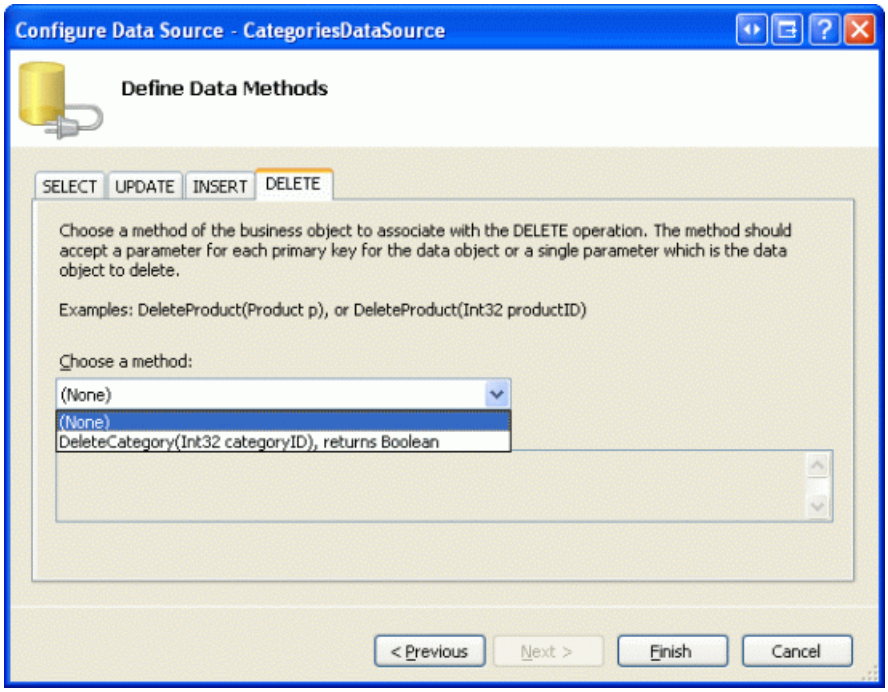


Figure 10: Set the Drop-Down Lists in the UPDATE and DELETE Tabs to (None) (Click to view full-size image)

After completing the wizard, the `CategoriesDataSource` s declarative markup should look like the following:

```
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

With the `CategoriesDataSource` created and configured, return to the `CategoryName` TemplateField s `ItemTemplate` and, from the DropDownList s smart tag, click on the Choose Data Source link. In the Data Source Configuration wizard, select the `CategoriesDataSource` option from the first drop-down list and choose to have `CategoryName` used for the display and `CategoryID` as the value.

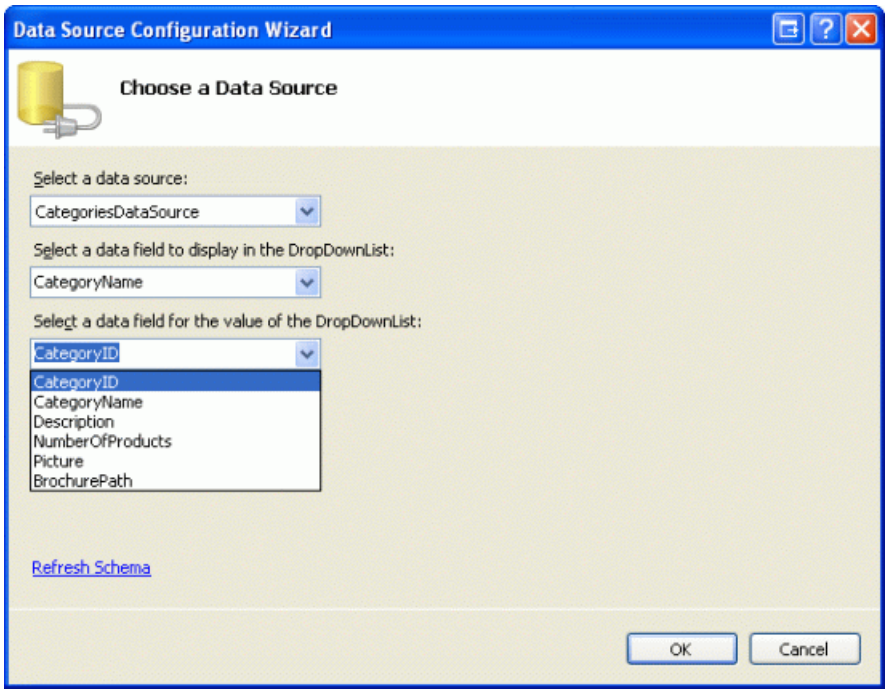


Figure 11: Bind the DropDownList to the `CategoriesDataSource` (Click to view full-size image)

At this point the `Categories` DropDownList lists all of the categories, but it does not yet automatically select the appropriate category for the product bound to the GridView row. To accomplish this we need to set the `Categories` DropDownList s `SelectedValue` to the product s `CategoryID` value. Click on the Edit DataBindings link from the DropDownList s smart tag and associate the `SelectedValue` property with the `CategoryID` data field as shown in Figure 12.

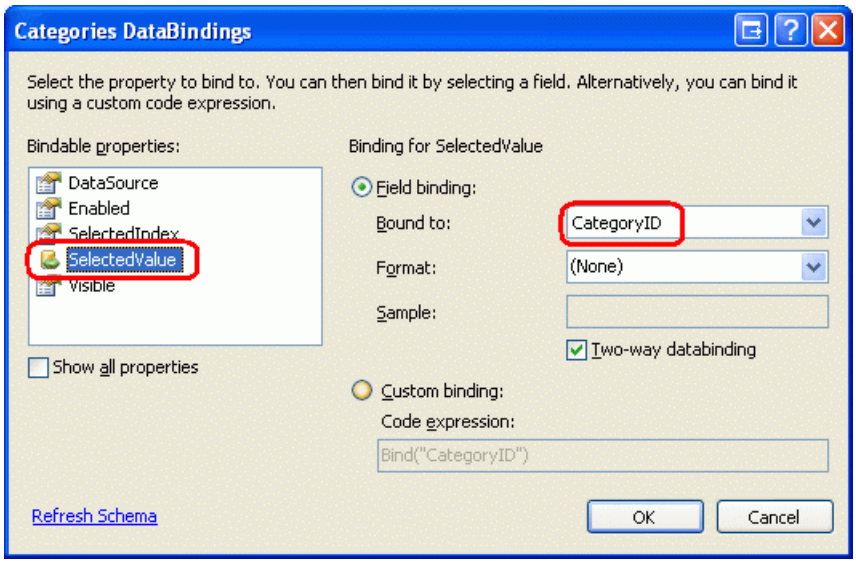


Figure 12: Bind the Product s `CategoryID` Value to the DropDownList s `SelectedValue` Property

One last problem remains: if the product doesn t have a `CategoryID` value specified then the databinding statement on `SelectedValue` will result in an exception. This is because the DropDownList contains only items for the categories and does not offer an option for those products that have a `NULL` database value for `CategoryID`. To remedy this, set the DropDownList s `AppendDataBoundItems` property to `True` and add a new item to the DropDownList, omitting the `Value` property from the declarative syntax. That is, make sure that the `Categories` DropDownList s declarative syntax looks like the following:

```
<asp:DropDownList ID="Categories" runat="server" AppendDataBoundItems="True"
    DataSourceID="CategoriesDataSource" DataTextField="CategoryName"
    DataValueField="CategoryID" SelectedValue='<%# Bind("CategoryID") %>'>
    <asp:ListItem Value="">-- Select One --</asp:ListItem>
</asp:DropDownList>
```

Note how the `<asp:ListItem Value="">` -- Select One -- has its `Value` attribute explicitly set to an empty string. Refer back to the Customizing the Data Modification Interface tutorial for a more thorough discussion on why this additional DropDownList item is needed to handle the `NULL` case and why assignment of the `Value` property to an empty string is essential.

> **NOTE**
>
> There is a potential performance and scalability issue here that is worth mentioning. Since each row has a DropDownList that uses the `CategoriesDataSource` as its data source, the `CategoriesBLL` class s `GetCategories` method will be called *n* times per page visit, where *n* is the number of rows in the GridView. These *n* calls to `GetCategories` result in *n* queries to the database. This impact on the database could be lessened by caching the returned categories either in a per-request cache or through the Caching Layer using a SQL caching dependency or a very short time-based expiry. For more information on the per-request caching option, see `HttpContext.Items` a Per-Request Cache Store.

Step 4: Completing the Editing Interface

We ve made a number of changes to the GridView s templates without pausing to view our progress. Take a moment to view our progress through a browser. As Figure 13 shows, each row is rendered using its `ItemTemplate`, which contains the cell s editing interface.

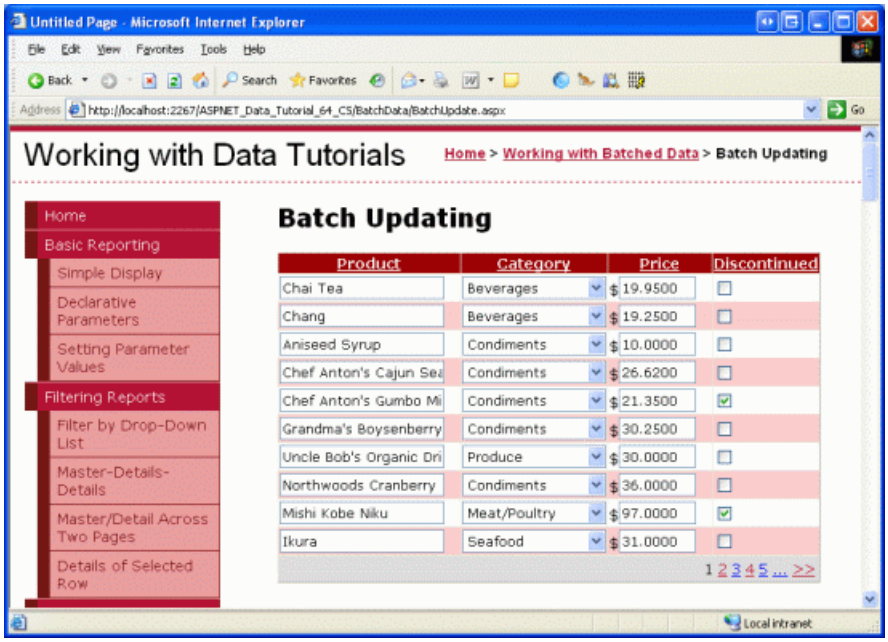


Figure 13: Each GridView Row is Editable (Click to view full-size image)

There are a few minor formatting issues that we should take care of at this point. First, note that the `UnitPrice` value contains four decimal points. To fix this, return to the `UnitPrice` TemplateField s `ItemTemplate` and, from the TextBox s smart tag, click on the Edit DataBindings link. Next, specify that the `Text` property should be formatted as a number.

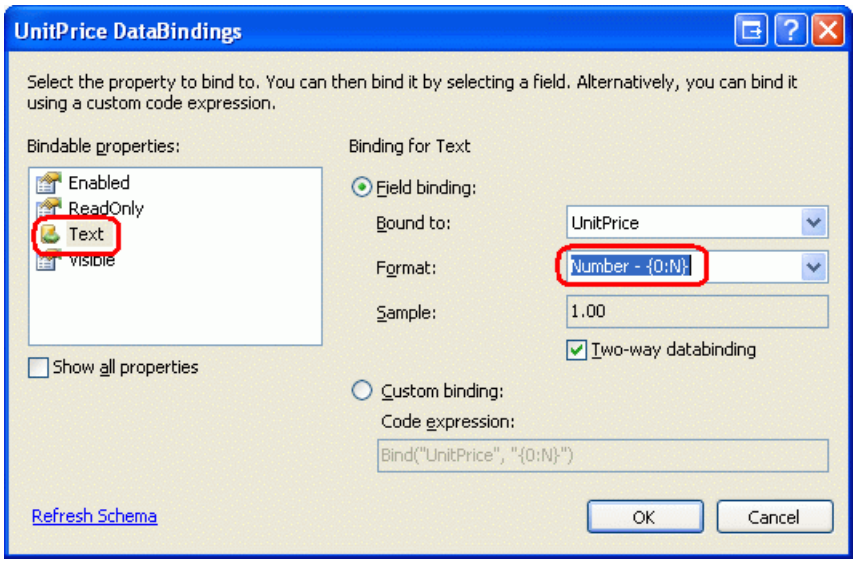


Figure 14: Format the `Text` Property as a Number

Second, let s center the checkbox in the `Discontinued` column (rather than having it left-aligned). Click on Edit Columns from the GridView s smart tag and select the `Discontinued` TemplateField from the list of fields in the bottom left corner. Drill down into `ItemStyle` and set the `HorizontalAlign` property to Center as shown in Figure 15.

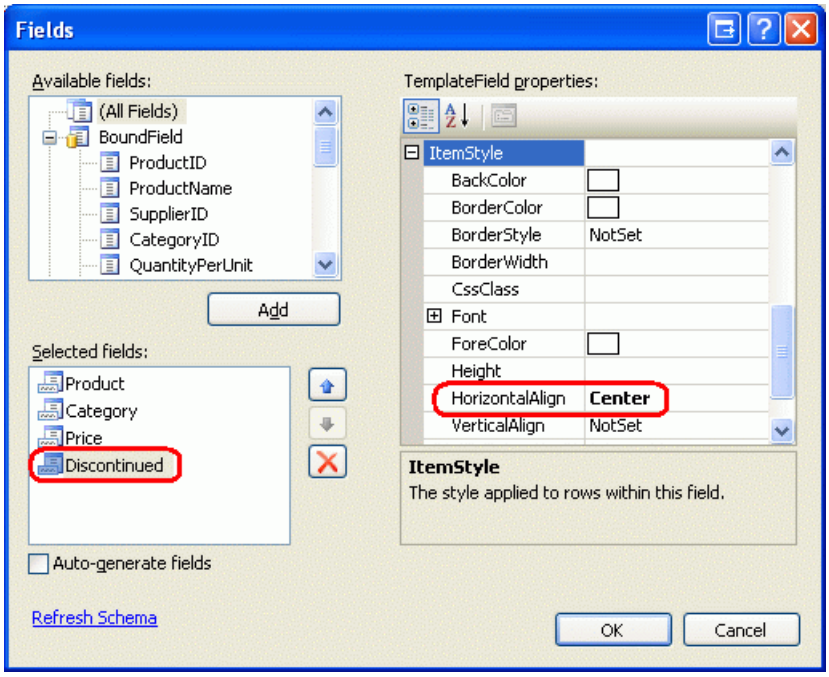


Figure 15: Center the `Discontinued` CheckBox

Next, add a ValidationSummary control to the page and set its `ShowMessageBox` property to `True` and its `ShowSummary` property to `False`. Also add the Button Web controls that, when clicked, will update the user s changes. Specifically, add two Button Web controls, one above the GridView and one below it, setting both controls `Text` properties to Update Products .

Since the GridView s editing interface is defined in its TemplateFields `ItemTemplate` s, the `EditItemTemplate` s are superfluous and may be deleted.

After making the above mentioned formatting changes, adding the Button controls, and removing the unnecessary `EditItemTemplate` s, your page s declarative syntax should look like the following:

```
<p>
    <asp:Button ID="UpdateAllProducts1" runat="server" Text="Update Products" />
```

```
</p>
<p>
    <asp:GridView ID="ProductsGrid" runat="server" AutoGenerateColumns="False"
        DataKeyNames="ProductID" DataSourceID="ProductsDataSource"
        AllowPaging="True" AllowSorting="True">
        <Columns>
            <asp:TemplateField HeaderText="Product" SortExpression="ProductName">
                <ItemTemplate>
                    <asp:TextBox ID="ProductName" runat="server"
                        Text='<%# Bind("ProductName") %>'></asp:TextBox>
                    <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
                        ControlToValidate="ProductName"
                        ErrorMessage="You must provide the product's name."
                        runat="server">*</asp:RequiredFieldValidator>
                </ItemTemplate>
            </asp:TemplateField>
            <asp:TemplateField HeaderText="Category"
                SortExpression="CategoryName">
                <ItemTemplate>
                    <asp:DropDownList ID="Categories" runat="server"
                        AppendDataBoundItems="True"
                        DataSourceID="CategoriesDataSource"
                        DataTextField="CategoryName"
                        DataValueField="CategoryID"
                        SelectedValue='<%# Bind("CategoryID") %>'>
                        <asp:ListItem>-- Select One --</asp:ListItem>
                    </asp:DropDownList>
                </ItemTemplate>
            </asp:TemplateField>
            <asp:TemplateField HeaderText="Price"
                SortExpression="UnitPrice">
                <ItemTemplate>
                    $<asp:TextBox ID="UnitPrice" runat="server" Columns="8"
                        Text='<%# Bind("UnitPrice", "{0:N}") %>'></asp:TextBox>
                    <asp:CompareValidator ID="CompareValidator1" runat="server"
                        ControlToValidate="UnitPrice"
                        ErrorMessage="You must enter a valid currency value.
                                      Please omit any currency symbols."
                        Operator="GreaterThanEqual" Type="Currency"
                        ValueToCompare="0">*</asp:CompareValidator>
                </ItemTemplate>
            </asp:TemplateField>
            <asp:TemplateField HeaderText="Discontinued" SortExpression="Discontinued">
                <ItemTemplate>
                    <asp:CheckBox ID="Discontinued" runat="server"
                        Checked='<%# Bind("Discontinued") %>' />
                </ItemTemplate>
                <ItemStyle HorizontalAlign="Center" />
            </asp:TemplateField>
        </Columns>
    </asp:GridView>
</p>
<p>
    <asp:Button ID="UpdateAllProducts2" runat="server" Text="Update Products" />
    <asp:ObjectDataSource ID="ProductsDataSource" runat="server"
        OldValuesParameterFormatString="original_{0}"
        SelectMethod="GetProducts" TypeName="ProductsBLL">
    </asp:ObjectDataSource>
    <asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
        OldValuesParameterFormatString="original_{0}"
        SelectMethod="GetCategories" TypeName="CategoriesBLL">
    </asp:ObjectDataSource>
    <asp:ValidationSummary ID="ValidationSummary1" runat="server"
        ShowMessageBox="True" ShowSummary="False" />
</p>
```

Figure 16 shows this page when viewed through a browser after the Button Web controls have been added and

the formatting changes made.

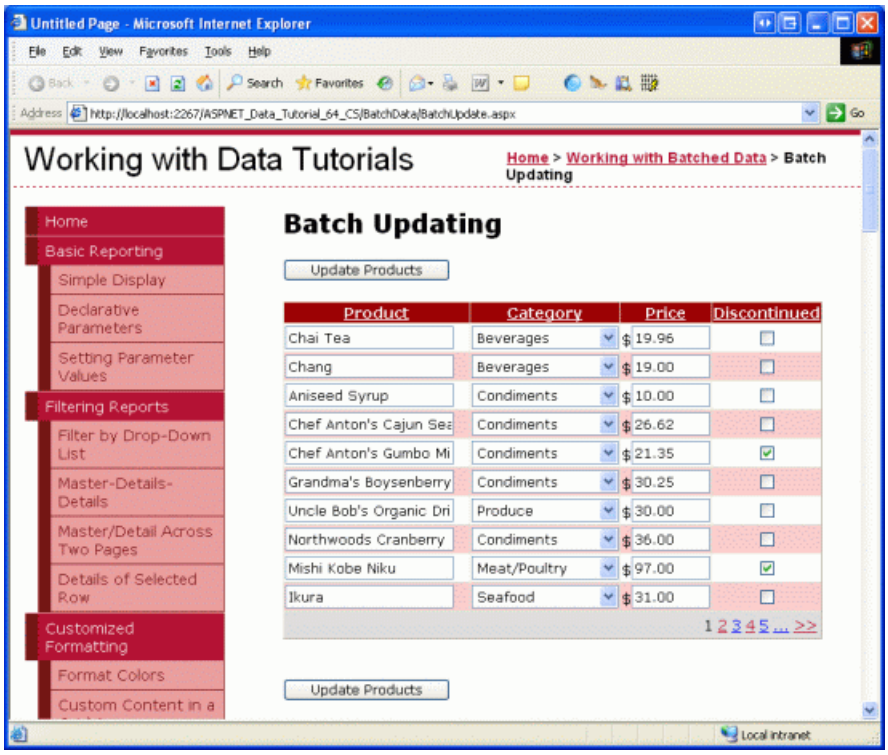


Figure 16: The Page Now Includes Two Update Products Buttons (Click to view full-size image)

Step 5: Updating the Products

When a user visits this page they will make their modifications and then click one of the two Update Products buttons. At that point we need to somehow save the user-entered values for each row into a `ProductsDataTable` instance and then pass that to a BLL method that will then pass that `ProductsDataTable` instance to the DAL s `UpdateWithTransaction` method. The `UpdateWithTransaction` method, which we created in the preceding tutorial, ensures that the batch of changes will be updated as an atomic operation.

Create a method named `BatchUpdate` in `BatchUpdate.aspx.vb` and add the following code:

```
Private Sub BatchUpdate()
    ' Enumerate the GridView's Rows collection and create a ProductRow
    Dim productsAPI As New ProductsBLL()
    Dim products As Northwind.ProductsDataTable = productsAPI.GetProducts()
    For Each gvRow As GridViewRow In ProductsGrid.Rows
        ' Find the ProductsRow instance in products that maps to gvRow
        Dim productID As Integer = _
            Convert.ToInt32(ProductsGrid.DataKeys(gvRow.RowIndex).Value)
        Dim product As Northwind.ProductsRow = products.FindByProductID(productID)
        If product IsNot Nothing Then
            ' Programmatically access the form field elements in the
            ' current GridViewRow
            Dim productName As TextBox = _
                CType(gvRow.FindControl("ProductName"), TextBox)
            Dim categories As DropDownList = _
                CType(gvRow.FindControl("Categories"), DropDownList)
            Dim unitPrice As TextBox = _
                CType(gvRow.FindControl("UnitPrice"), TextBox)
            Dim discontinued As CheckBox = _
                CType(gvRow.FindControl("Discontinued"), CheckBox)
            ' Assign the user-entered values to the current ProductRow
            product.ProductName = productName.Text.Trim()
            If categories.SelectedIndex = 0 Then
                product.SetCategoryIDNull()
            Else
                product.CategoryID = Convert.ToInt32(categories.SelectedValue)
            End If
            If unitPrice.Text.Trim().Length = 0 Then
                product.SetUnitPriceNull()
            Else
                product.UnitPrice = Convert.ToDecimal(unitPrice.Text)
            End If
            product.Discontinued = discontinued.Checked
        End If
    Next
    ' Now have the BLL update the products data using a transaction
    productsAPI.UpdateWithTransaction(products)
End Sub
```

This method starts out by getting all of the products back in a `ProductsDataTable` via a call to the BLL s `GetProducts` method. It then enumerates the `ProductGrid` GridView s `Rows` collection. The `Rows` collection contains a `GridViewRow` instance for each row displayed in the GridView. Since we are showing at most ten rows per page, the GridView s `Rows` collection will have no more than ten items.

For each row the `ProductID` is grabbed from the `DataKeys` collection and the appropriate `ProductsRow` is selected from the `ProductsDataTable`. The four TemplateField input controls are programmatically referenced and their values assigned to the `ProductsRow` instance s properties. After each GridView row s values have been used to update the `ProductsDataTable`, it s passed to the BLL s `UpdateWithTransaction` method which, as we saw in the preceding tutorial, simply calls down into the DAL s `UpdateWithTransaction` method.

The batch update algorithm used for this tutorial updates each row in the `ProductsDataTable` that corresponds to a row in the GridView, regardless of whether the product s information has been changed. While such blind updates aren't usually a performance issue, they can lead to superfluous records if you re auditing changes to the database table. Back in the Performing Batch Updates tutorial we explored a batch updating interface with the DataList and added code that would only update those records that were actually modified by the user. Feel free to use the techniques from Performing Batch Updates to update the code in this tutorial, if desired.

> **NOTE**
> When binding the data source to the GridView through its smart tag, Visual Studio automatically assigns the data source s primary key value(s) to the GridView s `DataKeyNames` property. If you did not bind the ObjectDataSource to the GridView through the GridView s smart tag as outlined in Step 1, then you will need to manually set the GridView s `DataKeyNames` property to ProductID in order to access the `ProductID` value for each row through the `DataKeys` collection.

The code used in `BatchUpdate` is similar to that used in the BLL s `UpdateProduct` methods, the main difference being that in the `UpdateProduct` methods only a single `ProductRow` instance is retrieved from the architecture. The code that assigns the properties of the `ProductRow` is the same between the `UpdateProducts` methods and the code within the `For Each` loop in `BatchUpdate`, as is the overall pattern.

To complete this tutorial, we need to have the `BatchUpdate` method invoked when either of the Update Products buttons is clicked. Create event handlers for the `Click` events of these two Button controls and add the following code in the event handlers:

```
BatchUpdate()
ClientScript.RegisterStartupScript(Me.GetType(), "message", _
    "alert('The products have been updated.');", True)
```

First a call is made to `BatchUpdate`. Next, the `ClientScript` property is used to inject JavaScript that will display a messagebox that reads The products have been updated.

Take a minute to test out this code. Visit `BatchUpdate.aspx` through a browser, edit a number of rows, and click one of the Update Products buttons. Assuming there are no input validation errors, you should see a messagebox that reads The products have been updated. To verify the atomicity of the update, consider adding a random `CHECK` constraint, like one that disallows `UnitPrice` values of 1234.56. Then from `BatchUpdate.aspx`, edit a number of records, making sure to set one of the product s `UnitPrice` value to the forbidden value (1234.56). This should result in an error when clicking Update Products with the other changes during that batch operation rolled back to their original values.

An Alternative `BatchUpdate` Method

The `BatchUpdate` method we just examined retrieves *all* of the products from the BLL s `GetProducts` method and then updates just those records that appear in the GridView. This approach is ideal if the GridView does not use paging, but if it does, there may be hundreds, thousands, or tens of thousands of products, but only ten rows in the GridView. In such a case, getting all of the products from the database only to modify 10 of them is less than ideal.

For those types of situations, consider using the following `BatchUpdateAlternate` method instead:

```
Private Sub BatchUpdateAlternate()
    ' Enumerate the GridView's Rows collection and create a ProductRow
    Dim productsAPI As New ProductsBLL()
    Dim products As New Northwind.ProductsDataTable()
    For Each gvRow As GridViewRow In ProductsGrid.Rows
        ' Create a new ProductRow instance
        Dim productID As Integer = _
            Convert.ToInt32(ProductsGrid.DataKeys(gvRow.RowIndex).Value)
        Dim currentProductDataTable As Northwind.ProductsDataTable = _
            productsAPI.GetProductByProductID(productID)
        If currentProductDataTable.Rows.Count > 0 Then
            Dim product As Northwind.ProductsRow = currentProductDataTable(0)
            Dim productName As TextBox = _
                CType(gvRow.FindControl("ProductName"), TextBox)
            Dim categories As DropDownList = _
                CType(gvRow.FindControl("Categories"), DropDownList)
            Dim unitPrice As TextBox = _
                CType(gvRow.FindControl("UnitPrice"), TextBox)
            Dim discontinued As CheckBox = _
                CType(gvRow.FindControl("Discontinued"), CheckBox)
            ' Assign the user-entered values to the current ProductRow
            product.ProductName = productName.Text.Trim()
            If categories.SelectedIndex = 0 Then
                product.SetCategoryIDNull()
            Else
                product.CategoryID = Convert.ToInt32(categories.SelectedValue)
            End If
            If unitPrice.Text.Trim().Length = 0 Then
                product.SetUnitPriceNull()
            Else
                product.UnitPrice = Convert.ToDecimal(unitPrice.Text)
            End If
            product.Discontinued = discontinued.Checked
            ' Import the ProductRow into the products DataTable
            products.ImportRow(product)
        End If
    Next
    ' Now have the BLL update the products data using a transaction
    productsAPI.UpdateProductsWithTransaction(products)
End Sub
```

`BatchMethodAlternate` starts by creating a new empty `ProductsDataTable` named `products`. It then steps through the GridView s `Rows` collection and for each row gets the particular product information using the BLL s `GetProductByProductID(productID)` method. The retrieved `ProductsRow` instance has its properties updated in the same fashion as `BatchUpdate`, but after updating the row it is imported into the `products` `ProductsDataTable` via the DataTable s `ImportRow(DataRow)` method.

After the `For Each` loop completes, `products` contains one `ProductsRow` instance for each row in the GridView. Since each of the `ProductsRow` instances have been added to the `products` (instead of updated), if we blindly pass it to the `UpdateWithTransaction` method the `ProductsTableAdapter` will try to insert each of the records into the database. Instead, we need to specify that each of these rows has been modified (not added).

This can be accomplished by adding a new method to the BLL named `UpdateProductsWithTransaction`. `UpdateProductsWithTransaction`, shown below, sets the `RowState` of each of the `ProductsRow` instances in the `ProductsDataTable` to `Modified` and then passes the `ProductsDataTable` to the DAL s `UpdateWithTransaction` method.

```
Public Function UpdateProductsWithTransaction _
    (ByVal products As Northwind.ProductsDataTable) As Integer
    ' Mark each product as Modified
    products.AcceptChanges()
    For Each product As Northwind.ProductsRow In products
        product.SetModified()
    Next
    ' Update the data via a transaction
    Return UpdateWithTransaction(products)
End Function
```

Summary

The GridView provides built-in per-row editing capabilities, but lacks support for creating fully editable interfaces. As we saw in this tutorial, such interfaces are possible, but require a bit of work. To create a GridView where every row is editable, we need to convert the GridView s fields into TemplateFields and define the editing interface within the `ItemTemplate` s. Additionally, Update All -type Button Web controls must be added to the page, separate from the GridView. These Buttons `Click` event handlers need to enumerate the GridView s `Rows` collection, store the changes in a `ProductsDataTable`, and pass the updated information into the appropriate BLL method.

In the next tutorial we'll see how to create an interface for batch deleting. In particular, each GridView row will include a checkbox and instead of Update All -type buttons, we'll have Delete Selected Rows buttons.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Teresa Murphy and David Suru. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Batch Deleting (VB)

2/20/2020 • 8 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Learn how to delete multiple database records in a single operation. In the User Interface Layer we build upon an enhanced GridView created in an earlier tutorial. In the Data Access Layer we wrap the multiple Delete operations within a transaction to ensure that all deletions succeed or all deletions are rolled back.

Introduction

The preceding tutorial explored how to create a batch editing interface using a fully-editable GridView. In situations where users are commonly editing many records at once, a batch editing interface will require far fewer postbacks and keyboard-to-mouse context switches, thereby improving the end user s efficiency. This technique is similarly useful for pages where it is common for users to delete many records in one go.

Anyone who has used an online email client is already familiar with one of the most common batch deleting interfaces: a checkbox in each row in a grid with a corresponding Delete All Checked Items button (see Figure 1). This tutorial is rather short because we ve already done all of the hard work in previous tutorials in creating both the web-based interface and a method to delete a series of records as a single atomic operation. In the Adding a GridView Column of Checkboxes tutorial we created a GridView with a column of checkboxes and in the Wrapping Database Modifications within a Transaction tutorial we created a method in the BLL that would use a transaction to delete a `List<T>` of `ProductID` values. In this tutorial, we will build upon and merge our previous experiences to create a working batch deleting example.

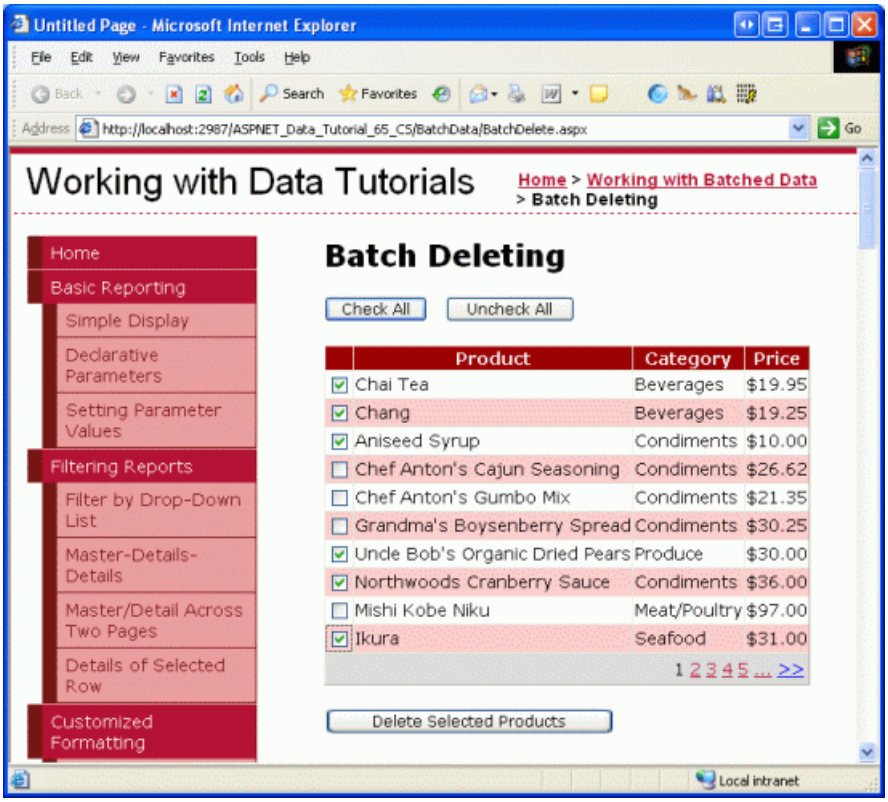


Figure 1: Each Row Includes a Checkbox (Click to view full-size image)

Step 1: Creating the Batch Deleting Interface

Since we already created the batch deleting interface in the Adding a GridView Column of Checkboxes tutorial, we can simply copy it to `BatchDelete.aspx` rather than creating it from scratch. Start by opening the `BatchDelete.aspx` page in the `BatchData` folder and the `CheckBoxField.aspx` page in the `EnhancedGridView` folder. From the `CheckBoxField.aspx` page, go to the Source view and copy the markup between the `<asp:Content>` tags as shown in Figure 2.

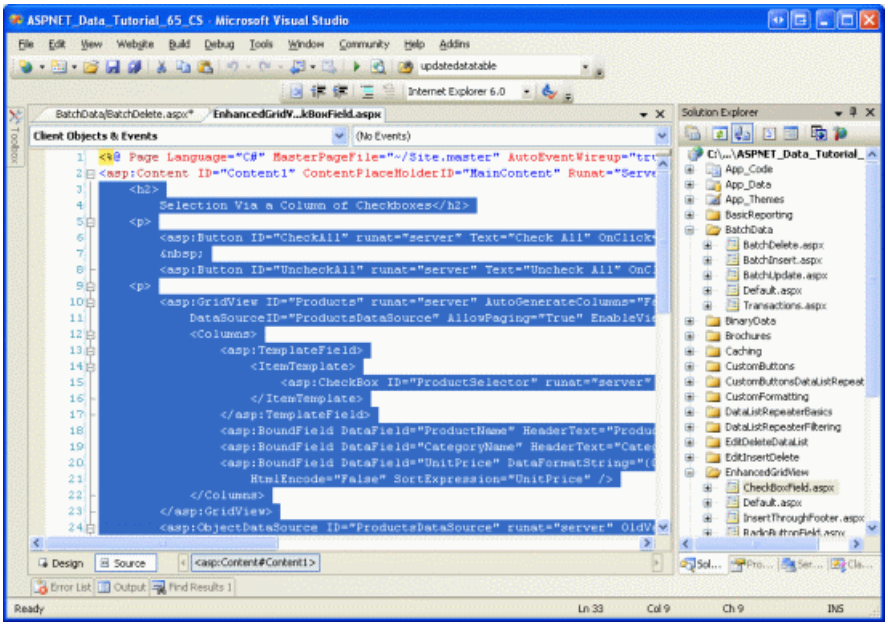


Figure 2: Copy the Declarative Markup of `CheckBoxField.aspx` to the Clipboard (Click to view full-size image)

Next, go to the Source view in `BatchDelete.aspx` and paste the contents of the clipboard within the `<asp:Content>` tags. Also copy and paste the code from within the code-behind class in `CheckBoxField.aspx.vb` to within the code-behind class in `BatchDelete.aspx.vb` (the `DeleteSelectedProducts` Button s `Click` event handler, the `ToggleCheckState` method, and the `Click` event handlers for the `CheckAll` and `UncheckAll` Buttons). After copying over this content, the `BatchDelete.aspx` page s code-behind class should contain the following code:

```
Partial Class BatchData_BatchDelete
    Inherits System.Web.UI.Page
    Protected Sub DeleteSelectedProducts_Click(sender As Object, e As EventArgs) _
        Handles DeleteSelectedProducts.Click

        Dim atLeastOneRowDeleted As Boolean = False
        ' Iterate through the Products.Rows property
        For Each row As GridViewRow In Products.Rows
            ' Access the CheckBox
            Dim cb As CheckBox = row.FindControl("ProductSelector")
            If cb IsNot Nothing AndAlso cb.Checked Then
                ' Delete row! (Well, not really...)
                atLeastOneRowDeleted = True
                ' First, get the ProductID for the selected row
                Dim productID As Integer = _
                    Convert.ToInt32(Products.DataKeys(row.RowIndex).Value)
                ' "Delete" the row
                DeleteResults.Text &= String.Format _
                    ("This would have deleted ProductID {0}<br />", productID)
                '... To actually delete the product, use ...
                ' Dim productAPI As New ProductsBLL
                ' productAPI.DeleteProduct(productID)
                '.............................................
            End If
        Next
        ' Show the Label if at least one row was deleted...
        DeleteResults.Visible = atLeastOneRowDeleted
    End Sub
    Private Sub ToggleCheckState(ByVal checkState As Boolean)
        ' Iterate through the Products.Rows property
        For Each row As GridViewRow In Products.Rows
            ' Access the CheckBox
            Dim cb As CheckBox = row.FindControl("ProductSelector")
            If cb IsNot Nothing Then
                cb.Checked = checkState
            End If
        Next
    End Sub
    Protected Sub CheckAll_Click(sender As Object, e As EventArgs) _
        Handles CheckAll.Click
        ToggleCheckState(True)
    End Sub
    Protected Sub UncheckAll_Click(sender As Object, e As EventArgs) _
        Handles UncheckAll.Click
        ToggleCheckState(False)
    End Sub
End Class
```

After copying over the declarative markup and source code, take a moment to test `BatchDelete.aspx` by viewing it through a browser. You should see a GridView listing the first ten products in a GridView with each row listing the product s name, category, and price along with a checkbox. There should be three buttons: Check All, Uncheck All, and Delete Selected Products. Clicking the Check All button selects all checkboxes, while Uncheck All clears all checkboxes. Clicking Delete Selected Products displays a message that lists the `ProductID` values of the selected products, but does not actually delete the products.

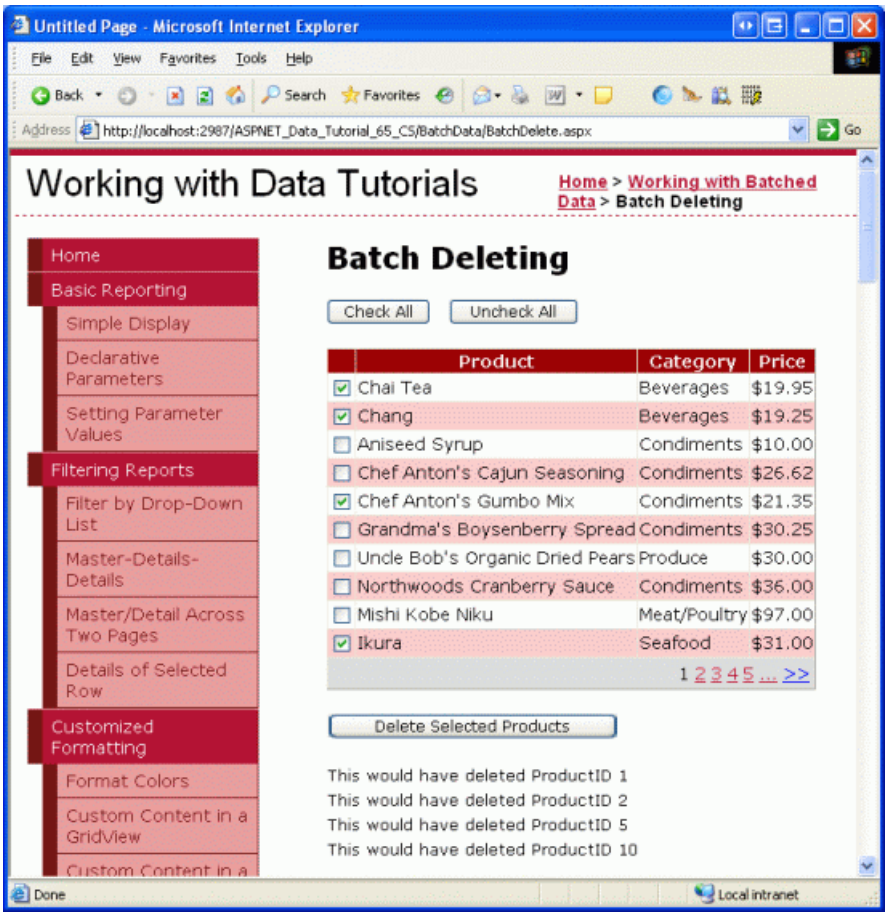


Figure 3: The Interface from `CheckBoxField.aspx` has been Moved to `BatchDeleting.aspx` (Click to view full-size image)

Step 2: Deleting the Checked Products Using Transactions

With the batch deleting interface successfully copied over to `BatchDeleting.aspx`, all that remains is to update the code so that the Delete Selected Products button deletes the checked products using the `DeleteProductsWithTransaction` method in the `ProductsBLL` class. This method, added in the Wrapping Database Modifications within a Transaction tutorial, accepts as its input a `List(Of T)` of `ProductID` values and deletes each corresponding `ProductID` within the scope of a transaction.

The `DeleteSelectedProducts` Button s `Click` event handler currently uses the following `For Each` loop to iterate through each GridView row:

```
' Iterate through the Products.Rows property
For Each row As GridViewRow In Products.Rows
    ' Access the CheckBox
    Dim cb As CheckBox = row.FindControl("ProductSelector")
    If cb IsNot Nothing AndAlso cb.Checked Then
        ' Delete row! (Well, not really...)
        atLeastOneRowDeleted = True
        ' First, get the ProductID for the selected row
        Dim productID As Integer = _
            Convert.ToInt32(Products.DataKeys(row.RowIndex).Value)
        ' "Delete" the row
        DeleteResults.Text &= String.Format _
            ("This would have deleted ProductID {0}<br />", productID)
        '... To actually delete the product, use ...
        ' Dim productAPI As New ProductsBLL
        ' productAPI.DeleteProduct(productID)
        '.............................................
    End If
Next
```

For each row, the `ProductSelector` CheckBox Web control is programmatically referenced. If it is checked, the row s `ProductID` is retrieved from the `DataKeys` collection and the `DeleteResults` Label s `Text` property is updated to include a message indicating that the row was selected for deletion.

The above code does not actually delete any records as the call to the `ProductsBLL` class s `Delete` method is commented out. Were this delete logic to be applied, the code would delete the products but not within an atomic operation. That is, if the first few deletes in the sequence succeeded, but a later one failed (perhaps due to a foreign key constraint violation), an exception would be thrown but those products already deleted would remain deleted.

In order to assure atomicity, we need to instead use the `ProductsBLL` class s `DeleteProductsWithTransaction` method. Because this method accepts a list of `ProductID` values, we need to first compile this list from the grid and then pass it as a parameter. We first create an instance of a `List(Of T)` of type `Integer`. Within the `For Each` loop we need to add the selected products `ProductID` values to this `List(Of T)`. After the loop this `List(Of T)` must be passed to the `ProductsBLL` class s `DeleteProductsWithTransaction` method. Update the `DeleteSelectedProducts` Button s `Click` event handler with the following code:

```
Protected Sub DeleteSelectedProducts_Click(sender As Object, e As EventArgs) _
    Handles DeleteSelectedProducts.Click

    ' Create a List to hold the ProductID values to delete
    Dim productIDsToDelete As New System.Collections.Generic.List(Of Integer)
    ' Iterate through the Products.Rows property
    For Each row As GridViewRow In Products.Rows
        ' Access the CheckBox
        Dim cb As CheckBox = CType(row.FindControl("ProductSelector"), CheckBox)
        If cb IsNot Nothing AndAlso cb.Checked Then
            ' Save the ProductID value for deletion
            ' First, get the ProductID for the selected row
            Dim productID As Integer = _
                Convert.ToInt32(Products.DataKeys(row.RowIndex).Value)
            ' Add it to the List...
            productIDsToDelete.Add(productID)
            ' Add a confirmation message
            DeleteResults.Text &= String.Format _
                ("ProductID {0} has been deleted<br />", productID)
        End If
    Next
    ' Call the DeleteProductsWithTransaction method and show the Label
    ' if at least one row was deleted...
    If productIDsToDelete.Count > 0 Then
        Dim productAPI As New ProductsBLL()
        productAPI.DeleteProductsWithTransaction(productIDsToDelete)
        DeleteResults.Visible = True
        ' Rebind the data to the GridView
        Products.DataBind()
    End If
End Sub
```

The updated code creates a `List(Of T)` of type `Integer` (`productIDsToDelete`) and populates it with the `ProductID` values to delete. After the `For Each` loop, if there is at least one product selected, the `ProductsBLL` class s `DeleteProductsWithTransaction` method is called and passed this list. The `DeleteResults` Label is also displayed and the data rebound to the GridView (so that the newly-deleted records no longer appear as rows in the grid).

Figure 4 shows the GridView after a number of rows have been selected for deletion. Figure 5 shows the screen immediately after the Delete Selected Products button has been clicked. Note that in Figure 5 the `ProductID` values of the deleted records are displayed in the Label beneath the GridView and those rows are no longer in the GridView.

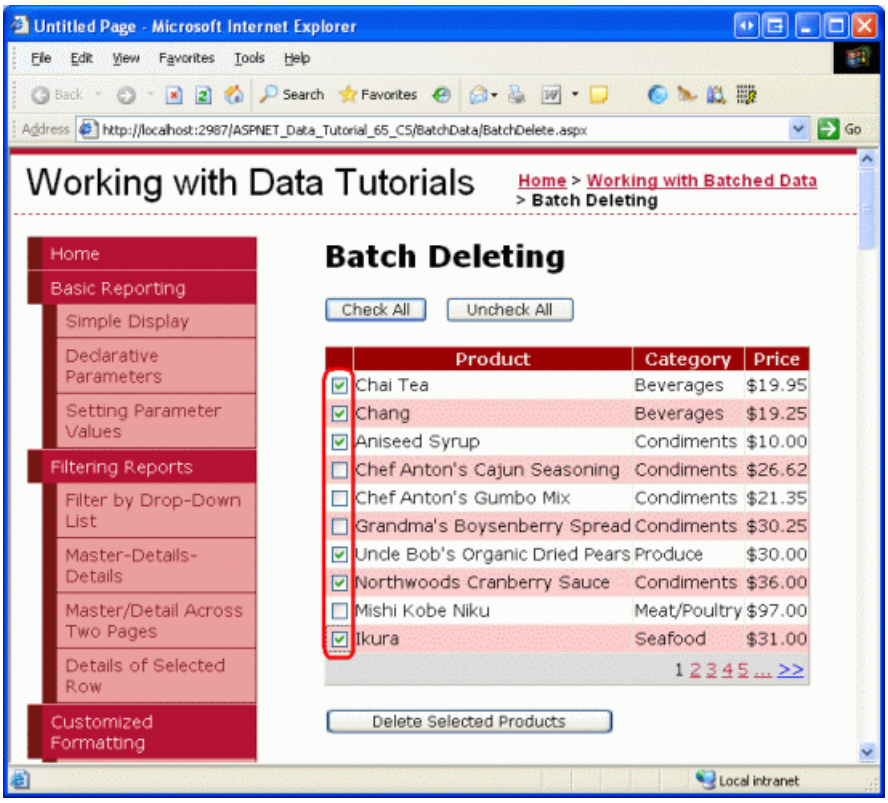


Figure 4: The Selected Products Will Be Deleted (Click to view full-size image)

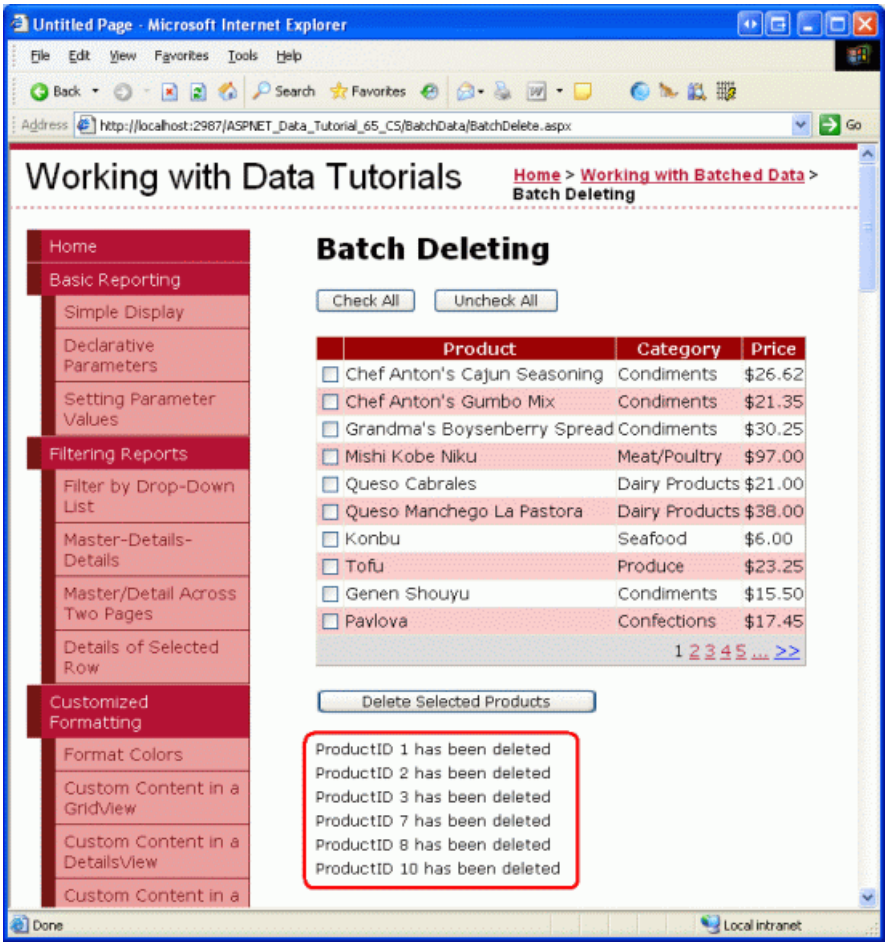


Figure 5: The Deleted Products `ProductID` Values are Listed Beneath the GridView (Click to view full-size image)

NOTE

To test the `DeleteProductsWithTransaction` method s atomicity, manually add an entry for a product in the `Order Details` table and then attempt to delete that product (along with others). You will receive a foreign key constraint violation when attempting to delete the product with an associated order, but note how the other selected products deletions are rolled back.

Summary

Creating a batch deleting interface involves adding a GridView with a column of checkboxes and a Button Web control that, when clicked, will delete all of the selected rows as a single atomic operation. In this tutorial we built such an interface by piecing together work done in two previous tutorials, Adding a GridView Column of Checkboxes and Wrapping Database Modifications within a Transaction. In the first tutorial we created a GridView with a column of checkboxes and in the latter we implemented a method in the BLL that, when passed a `List(Of T)` of `ProductID` values, deleted them all within the scope of a transaction.

In the next tutorial we'll create an interface for performing batch inserts.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Hilton Giesenow and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Batch Inserting (VB)

2/20/2020 • 23 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Learn how to insert multiple database records in a single operation. In the User Interface Layer we extend the GridView to allow the user to enter multiple new records. In the Data Access Layer we wrap the multiple Insert operations within a transaction to ensure that all insertions succeed or all insertions are rolled back.

Introduction

In the Batch Updating tutorial we looked at customizing the GridView control to present an interface where multiple records were editable. The user visiting the page could make a series of changes and then, with a single button click, perform a batch update. For situations where users commonly update many records in one go, such an interface can save countless clicks and keyboard-to-mouse context switches when compared to the default per-row editing features that were first explored back in the An Overview of Inserting, Updating, and Deleting Data tutorial.

This concept can also be applied when adding records. Imagine that here at Northwind Traders we commonly receive shipments from suppliers that contain a number of products for a particular category. As an example, we might receive a shipment of six different tea and coffee products from Tokyo Traders. If a user enters the six products one at a time through a DetailsView control, they will have to choose many of the same values over and over again: they will need to choose the same category (Beverages), the same supplier (Tokyo Traders), the same discontinued value (False), and the same units on order value (0). This repetitive data entry is not only time consuming, but is prone to errors.

With a little work we can create a batch inserting interface that enables the user to choose the supplier and category once, enter a series of product names and unit prices, and then click a button to add the new products to the database (see Figure 1). As each product is added, its `ProductName` and `UnitPrice` data fields are assigned the values entered in the TextBoxes, while its `CategoryID` and `SupplierID` values are assigned the values from the DropDownLists at the top fo the form. The `Discontinued` and `UnitsOnOrder` values are set to the hard-coded values of `False` and 0, respectively.

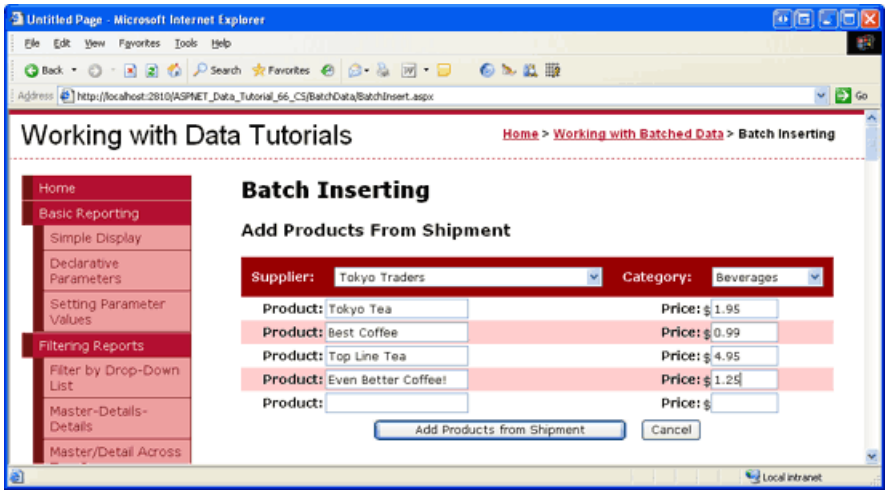


Figure 1: The Batch Inserting Interface (Click to view full-size image)

In this tutorial we will create a page that implements the batch inserting interface shown in Figure 1. As with the

previous two tutorials, we will wrap the insertions within the scope of a transaction to ensure atomicity. Let s get started!

Step 1: Creating the Display Interface

This tutorial will consist of a single page that is divided into two regions: a display region and an inserting region. The display interface, which we'll create in this step, shows the products in a GridView and includes a button titled Process Product Shipment. When this button is clicked, the display interface is replaced with the inserting interface, which is shown in Figure 1. The display interface returns after the Add Products from Shipment or Cancel buttons are clicked. We'll create the inserting interface in Step 2.

When creating a page that has two interfaces, only one of which is visible at a time, each interface typically is placed within a Panel Web control, which serves as a container for other controls. Therefore, our page will have two Panel controls one for each interface.

Start by opening the `BatchInsert.aspx` page in the `BatchData` folder and drag a Panel from the Toolbox onto the Designer (see Figure 2). Set the Panel s `ID` property to `DisplayInterface`. When adding the Panel to the Designer, its `Height` and `Width` properties are set to 50px and 125px, respectively. Clear out these property values from the Properties window.

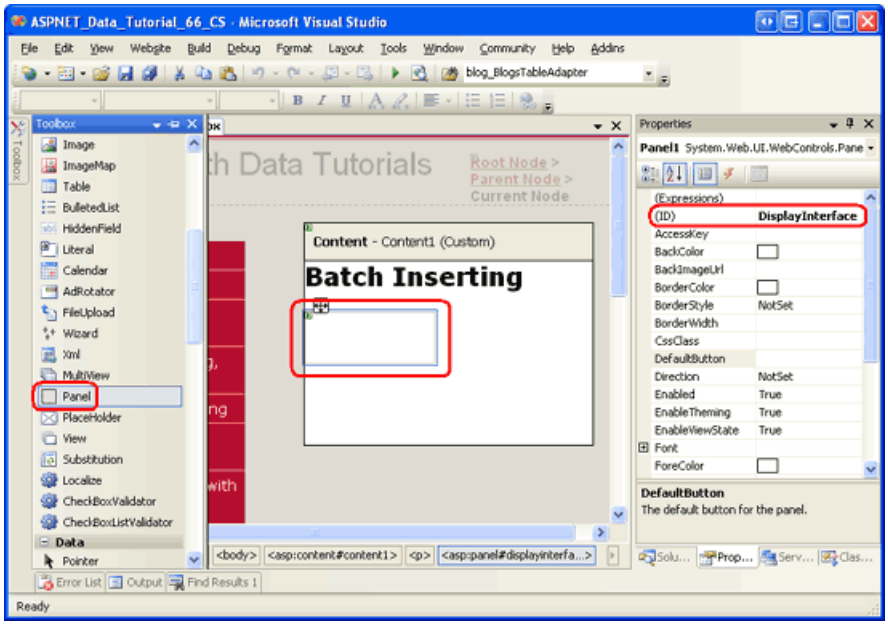


Figure 2: Drag a Panel from the Toolbox onto the Designer (Click to view full-size image)

Next, drag a Button and GridView control into the Panel. Set the Button s `ID` property to `ProcessShipment` and its `Text` property to Process Product Shipment. Set the GridView s `ID` property to `ProductsGrid` and, from its smart tag, bind it to a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to pull its data from the `ProductsBLL` class s `GetProducts` method. Since this GridView is used only to display data, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None). Click Finish to complete the Configure Data Source wizard.

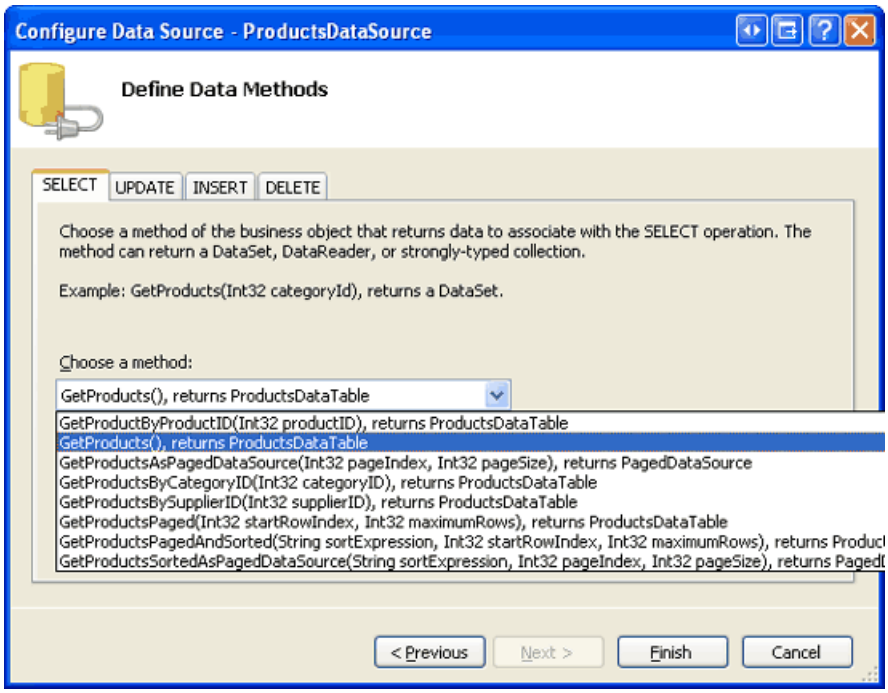


Figure 3: Display the Data Returned from the `ProductsBLL` Class s `GetProducts` Method (Click to view full-size image)

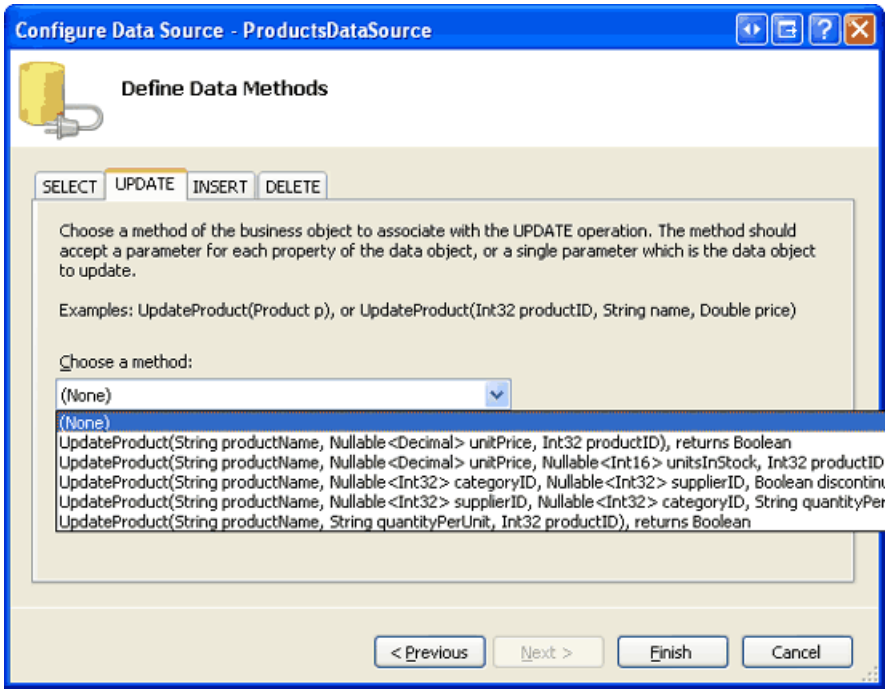


Figure 4: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE Tabs to (None) (Click to view full-size image)

After completing the ObjectDataSource wizard, Visual Studio will add BoundFields and a CheckBoxField for the product data fields. Remove all but the `ProductName`, `CategoryName`, `SupplierName`, `UnitPrice`, and `Discontinued` fields. Feel free to make any aesthetic customizations. I decided to format the `UnitPrice` field as a currency value, reordered the fields, and renamed several of the fields `HeaderText` values. Also configure the GridView to include paging and sorting support by checking the Enable Paging and Enable Sorting checkboxes in the GridView s smart tag.

After adding the Panel, Button, GridView, and ObjectDataSource controls and customizing the GridView s fields, your page s declarative markup should look similar to the following:

```
<asp:Panel ID="DisplayInterface" runat="server">
    <p>
        <asp:Button ID="ProcessShipment" runat="server"
            Text="Process Product Shipment" />
    </p>
    <asp:GridView ID="ProductsGrid" runat="server" AllowPaging="True"
        AllowSorting="True" AutoGenerateColumns="False"
        DataKeyNames="ProductID" DataSourceID="ProductsDataSource">
        <Columns>
            <asp:BoundField DataField="ProductName" HeaderText="Product"
                SortExpression="ProductName" />
            <asp:BoundField DataField="CategoryName" HeaderText="Category"
                ReadOnly="True" SortExpression="CategoryName" />
            <asp:BoundField DataField="SupplierName" HeaderText="Supplier"
                ReadOnly="True" SortExpression="SupplierName" />
            <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
                HeaderText="Price" HtmlEncode="False"
                SortExpression="UnitPrice">
                <ItemStyle HorizontalAlign="Right" />
            </asp:BoundField>
            <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
                SortExpression="Discontinued">
                <ItemStyle HorizontalAlign="Center" />
            </asp:CheckBoxField>
        </Columns>
    </asp:GridView>
    <asp:ObjectDataSource ID="ProductsDataSource" runat="server"
        OldValuesParameterFormatString="original_{0}"
        SelectMethod="GetProducts" TypeName="ProductsBLL">
    </asp:ObjectDataSource>
</asp:Panel>
```

Note that the markup for the Button and GridView appear within the opening and closing `<asp:Panel>` tags. Since these controls are within the `DisplayInterface` Panel, we can hide them by simply setting the Panel s `Visible` property to `False`. Step 3 looks at programmatically changing the Panel s `Visible` property in response to a button click to show one interface while hiding the other.

Take a moment to view our progress through a browser. As Figure 5 shows, you should see a Process Product Shipment button above a GridView that lists the products ten at a time.

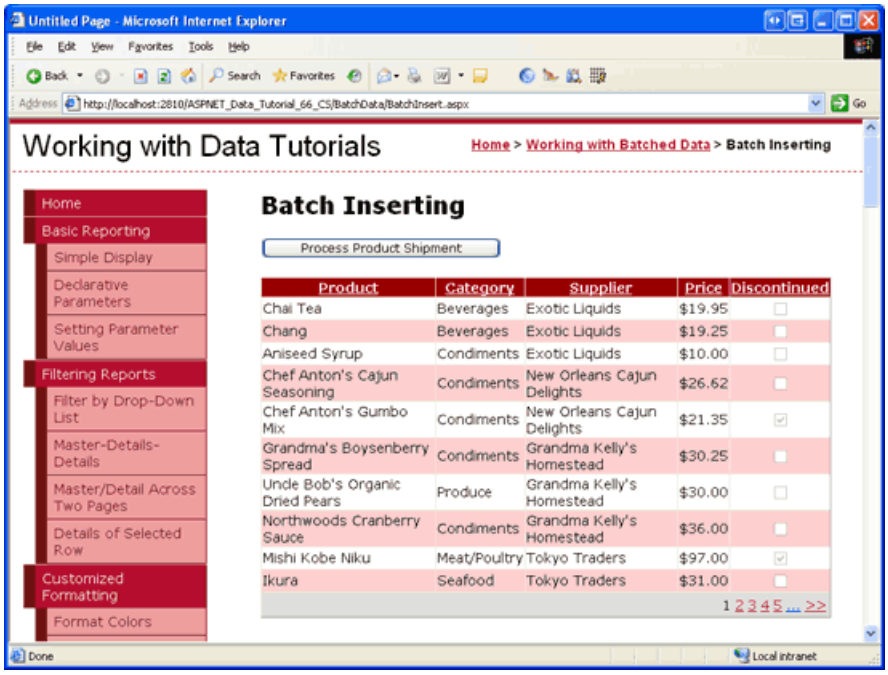


Figure 5: The GridView Lists the Products and Offers Sorting and Paging Capabilities (Click to view full-size image)

Step 2: Creating the Inserting Interface

With the display interface complete, we re ready to create the inserting interface. For this tutorial, let s create an inserting interface that prompts for a single supplier and category value and then allows the user to enter up to five product names and unit price values. With this interface, the user can add one to five new products that all share the same category and supplier, but have unique product names and prices.

Start by dragging a Panel from the Toolbox onto the Designer, placing it beneath the existing `DisplayInterface` Panel. Set the `ID` property of this newly added Panel to `InsertingInterface` and set its `Visible` property to `False`. We'll add code that sets the `InsertingInterface` Panel s `Visible` property to `True` in Step 3. Also clear out the Panel s `Height` and `Width` property values.

Next, we need to create the inserting interface that was shown back in Figure 1. This interface can be created through a variety of HTML techniques, but we will use a fairly straightforward one: a four-column, seven-row table.

> **NOTE**
>
> When entering markup for HTML `<table>` elements, I prefer to use the Source view. While Visual Studio does have tools for adding `<table>` elements through the Designer, the Designer seems all too willing to inject unasked for `style` settings into the markup. Once I have created the `<table>` markup, I usually return to the Designer to add the Web controls and set their properties. When creating tables with pre-determined columns and rows I prefer using static HTML rather than the Table Web control because any Web controls placed within a Table Web control can only be accessed using the `FindControl("controlID")` pattern. I do, however, use Table Web controls for dynamically-sized tables (ones whose rows or columns are based on some database or user-specified criteria), since the Table Web control can be constructed programmatically.

Enter the following markup within the `<asp:Panel>` tags of the `InsertingInterface` Panel:

```
<table class="DataWebControlStyle" cellspacing="0">
    <tr class="BatchInsertHeaderRow">
        <td class="BatchInsertLabel">Supplier:</td>
        <td></td>
        <td class="BatchInsertLabel">Category:</td>
        <td></td>
    </tr>
    <tr class="BatchInsertRow">
        <td class="BatchInsertLabel">Product:</td>
        <td></td>
        <td class="BatchInsertLabel">Price:</td>
        <td></td>
    </tr>
    <tr class="BatchInsertAlternatingRow">
        <td class="BatchInsertLabel">Product:</td>
        <td></td>
        <td class="BatchInsertLabel">Price:</td>
        <td></td>
    </tr>
    <tr class="BatchInsertRow">
        <td class="BatchInsertLabel">Product:</td>
        <td></td>
        <td class="BatchInsertLabel">Price:</td>
        <td></td>
    </tr>
    <tr class="BatchInsertAlternatingRow">
        <td class="BatchInsertLabel">Product:</td>
        <td></td>
        <td class="BatchInsertLabel">Price:</td>
        <td></td>
    </tr>
    <tr class="BatchInsertRow">
        <td class="BatchInsertLabel">Product:</td>
        <td></td>
        <td class="BatchInsertLabel">Price:</td>
        <td></td>
    </tr>
    <tr class="BatchInsertFooterRow">
        <td colspan="4">
        </td>
    </tr>
</table>
```

This `<table>` markup does not include any Web controls yet, we'll add those momentarily. Note that each `<tr>` element contains a particular CSS class setting: `BatchInsertHeaderRow` for the header row where the supplier and category DropDownLists will go; `BatchInsertFooterRow` for the footer row where the Add Products from Shipment and Cancel Buttons will go; and alternating `BatchInsertRow` and `BatchInsertAlternatingRow` values for the rows that will contain the product and unit price TextBox controls. I ve created corresponding CSS classes in the `Styles.css` file to give the inserting interface an appearance similar to the GridView and DetailsView controls we ve used throughout these tutorials. These CSS classes are shown below.

```
/*** Styles for ~/BatchData/BatchInsert.aspx tutorial ***/
.BatchInsertLabel
{
    font-weight: bold;
    text-align: right;
}
.BatchInsertHeaderRow td
{
    color: White;
    background-color: #900;
    padding: 11px;
}
.BatchInsertFooterRow td
{
    text-align: center;
    padding-top: 5px;
}
.BatchInsertRow
{
}
.BatchInsertAlternatingRow
{
    background-color: #fcc;
}
```

With this markup entered, return to the Design view. This `<table>` should show as a four-column, seven-row table in the Designer, as Figure 6 illustrates.

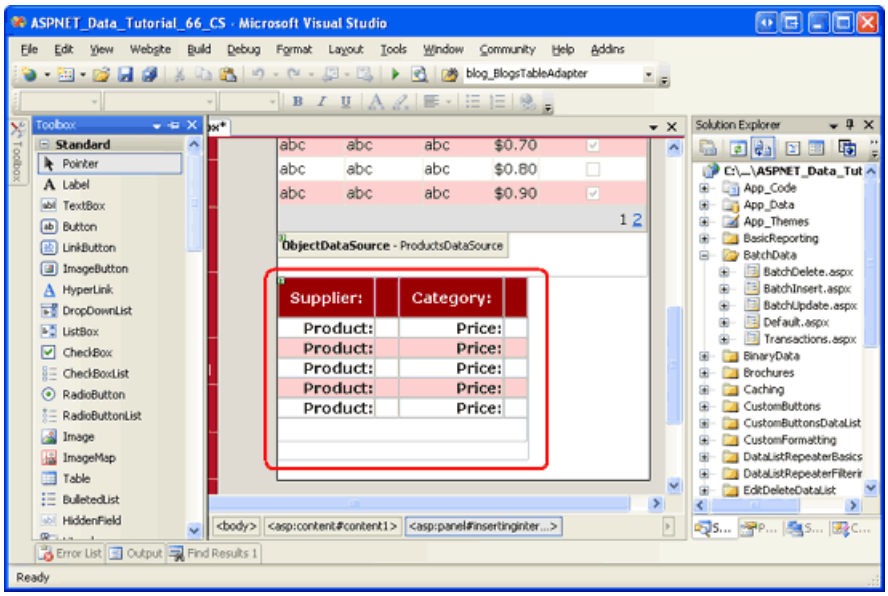


Figure 6: The Inserting Interface is Composed of a Four-Column, Seven-Row Table (Click to view full-size image)

We re now ready to add the Web controls to the inserting interface. Drag two DropDownLists from the Toolbox into the appropriate cells in the table one for the supplier and one for the category.

Set the supplier DropDownList s `ID` property to `Suppliers` and bind it to a new ObjectDataSource named `SuppliersDataSource`. Configure the new ObjectDataSource to retrieve its data from the `SuppliersBLL` class s `GetSuppliers` method and set the UPDATE tab s drop-down list to (None). Click Finish to complete the wizard.

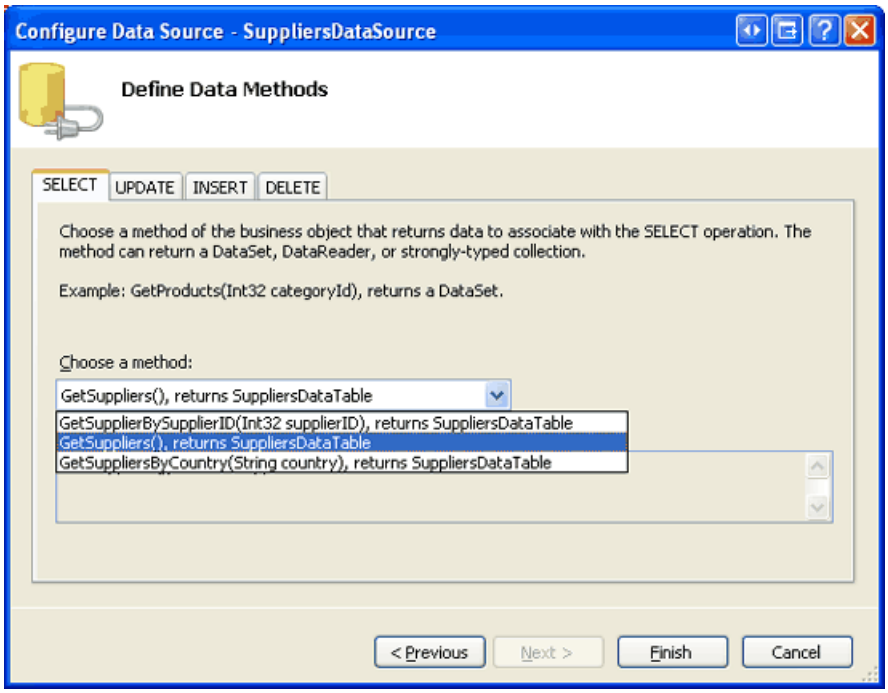


Figure 7: Configure the ObjectDataSource to Use the `SuppliersBLL` Class s `GetSuppliers` Method (Click to view full-size image)

Have the `Suppliers` DropDownList display the `CompanyName` data field and use the `SupplierID` data field as its `ListItem` s values.

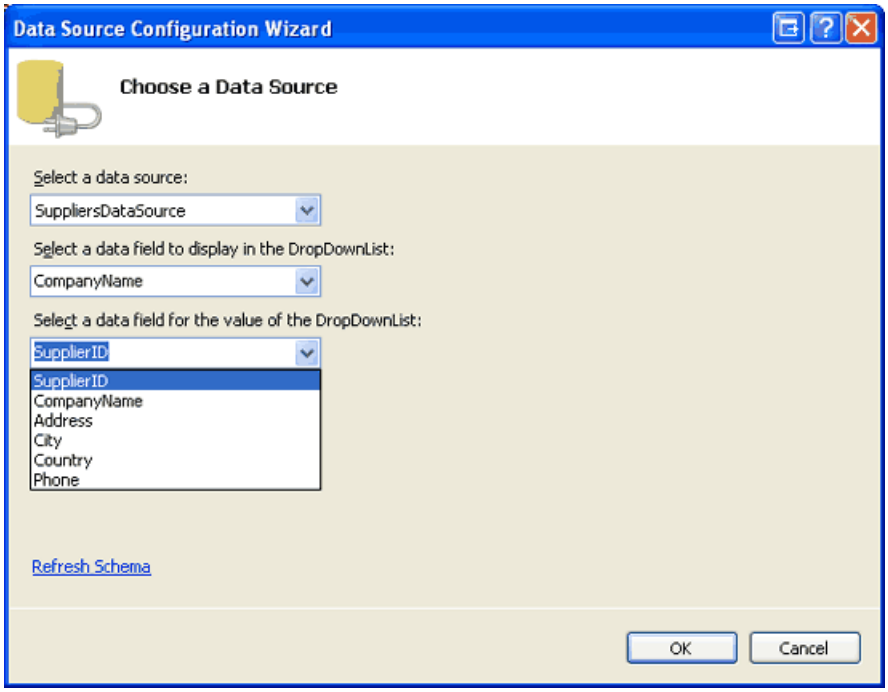


Figure 8: Display the `CompanyName` Data Field and Use `SupplierID` as the Value (Click to view full-size image)

Name the second DropDownList `Categories` and bind it to a new ObjectDataSource named `CategoriesDataSource`. Configure the `CategoriesDataSource` ObjectDataSource to use the `CategoriesBLL` class s `GetCategories` method; set the drop-down lists in the UPDATE and DELETE tabs to (None) and click Finish to complete the wizard. Finally, have the DropDownList display the `CategoryName` data field and use the `CategoryID` as the value.

After these two DropDownLists have been added and bound to appropriately configured ObjectDataSources, your screen should look similar to Figure 9.

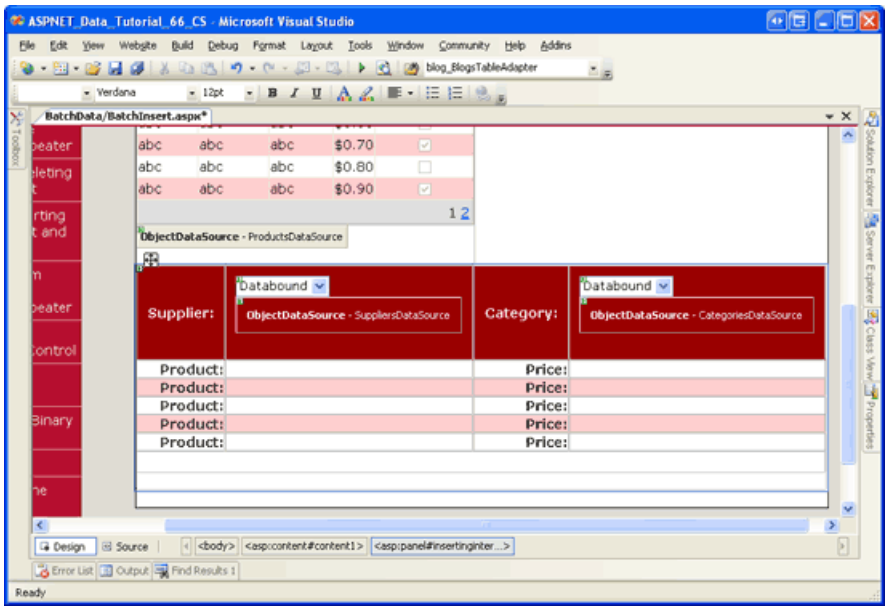


Figure 9: The Header Row Now Contains the `Suppliers` and `Categories` DropDownLists (Click to view full-size image)

We now need to create the TextBoxes to collect the name and price for each new product. Drag a TextBox control from the Toolbox onto the Designer for each of the five product name and price rows. Set the `ID` properties of the TextBoxes to `ProductName1`, `UnitPrice1`, `ProductName2`, `UnitPrice2`, `ProductName3`, `UnitPrice3`, and so on.

Add a CompareValidator after each of the unit price TextBoxes, setting the `ControlToValidate` property to the appropriate `ID`. Also set the `Operator` property to `GreaterThanEqual`, `ValueToCompare` to 0, and `Type` to `Currency`. These settings instruct the CompareValidator to ensure that the price, if entered, is a valid currency value that is greater than or equal to zero. Set the `Text` property to *, and `ErrorMessage` to The price must be greater than or equal to zero. Also, please omit any currency symbols.

> **NOTE**
>
> The inserting interface does not include any RequiredFieldValidator controls, even though the `ProductName` field in the `Products` database table does not allow `NULL` values. This is because we want to let the user enter up to five products. For example, if the user were to provide the product name and unit price for the first three rows, leaving the last two rows blank, we d just add three new products to the system. Since `ProductName` is required, however, we will need to programmatically check to ensure that if a unit price is entered that a corresponding product name value is provided. We'll tackle this check in Step 4.

When validating the user s input, the CompareValidator reports invalid data if the value contains a currency symbol. Add a $ in front of each of the unit price TextBoxes to serve as a visual cue that instructs the user to omit the currency symbol when entering the price.

Lastly, add a ValidationSummary control within the `InsertingInterface` Panel, settings its `ShowMessageBox` property to `True` and its `ShowSummary` property to `False`. With these settings, if the user enters an invalid unit price value, an asterisk will appear next to the offending TextBox controls and the ValidationSummary will display a client-side messagebox that shows the error message we specified earlier.

At this point, your screen should look similar to Figure 10.

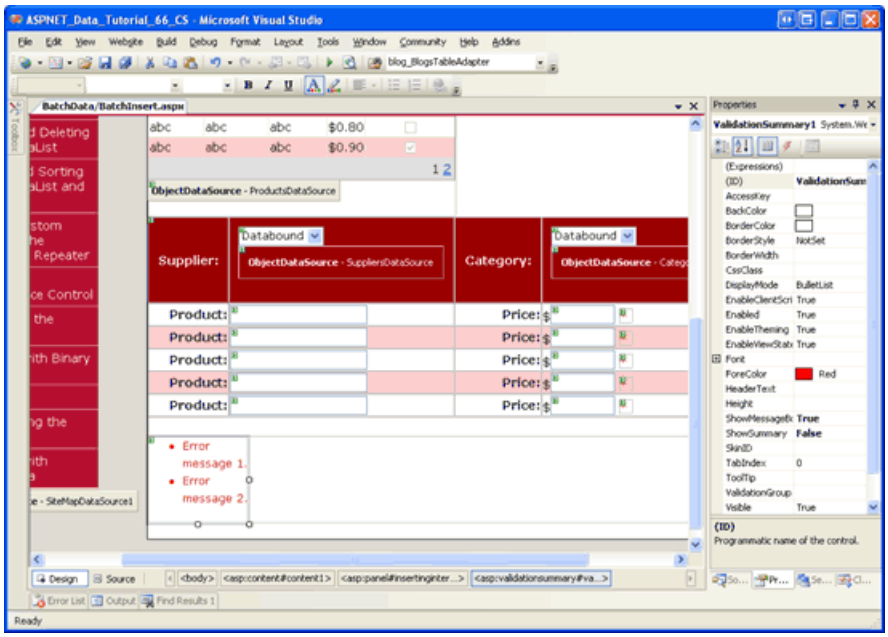


Figure 10: The Inserting Interface Now Includes TextBoxes for the Products Names and Prices (Click to view full-size image)

Next we need to add the Add Products from Shipment and Cancel buttons to the footer row. Drag two Button controls from the Toolbox into the footer of the inserting interface, setting the Buttons `ID` properties to `AddProducts` and `CancelButton` and `Text` properties to Add Products from Shipment and Cancel, respectively. In addition, set the `CancelButton` control s `CausesValidation` property to `false`.

Finally, we need to add a Label Web control that will display status messages for the two interfaces. For example, when a user successfully adds a new shipment of products, we want to return to the display interface and display a confirmation message. If, however, the user provides a price for a new product but leaves off the product name, we need to display a warning message since the `ProductName` field is required. Since we need this message to display for both interfaces, place it at the top of the page outside of the Panels.

Drag a Label Web control from the Toolbox to the top of the page in the Designer. Set the `ID` property to `StatusLabel`, clear out the `Text` property, and set the `Visible` and `EnableViewState` properties to `False`. As we have seen in previous tutorials, setting the `EnableViewState` property to `False` allows us to programmatically change the Label s property values and have them automatically revert back to their defaults on the subsequent postback. This simplifies the code for showing a status message in response to some user action that disappears on the subsequent postback. Finally, set the `StatusLabel` control s `CssClass` property to Warning, which is the name of a CSS class defined in `Styles.css` that displays text in a large, italic, bold, red font.

Figure 11 shows the Visual Studio Designer after the Label has been added and configured.

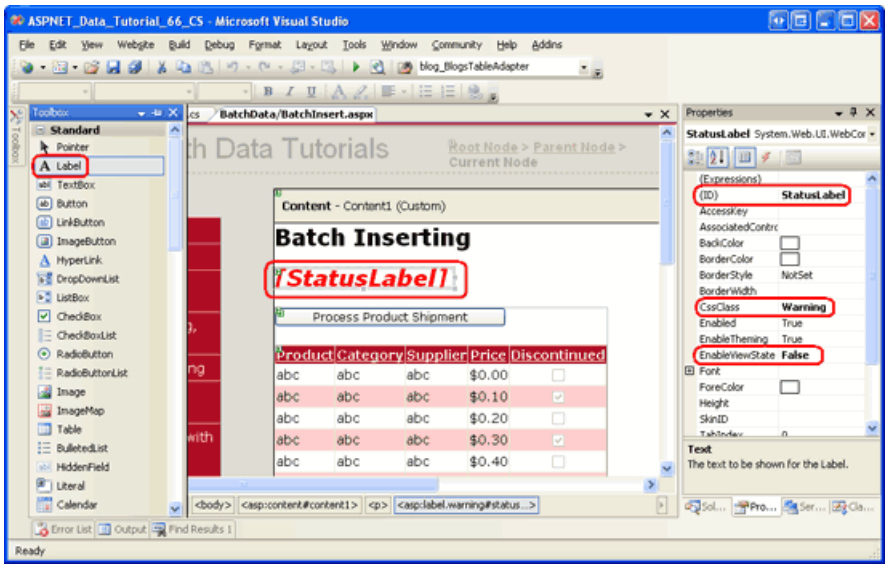


Figure 11: Place the `StatusLabel` Control Above the Two Panel Controls (Click to view full-size image)

Step 3: Switching Between the Display and Inserting Interfaces

At this point we have completed the markup for our display and inserting interfaces, but we re still left with two tasks:

- Switching between the display and inserting interfaces
- Adding the products in the shipment to the database

Currently, the display interface is visible but the inserting interface is hidden. This is because the `DisplayInterface` Panel s `Visible` property is set to `True` (the default value), while the `InsertingInterface` Panel s `Visible` property is set to `False`. To switch between the two interfaces we simply need to toggle each control s `Visible` property value.

We want to move from the display interface to the inserting interface when the Process Product Shipment button is clicked. Therefore, create an event handler for this Button s `Click` event that contains the following code:

```
Protected Sub ProcessShipment_Click(sender As Object, e As EventArgs) _
    Handles ProcessShipment.Click
    DisplayInterface.Visible = False
    InsertingInterface.Visible = True
End Sub
```

This code simply hides the `DisplayInterface` Panel and shows the `InsertingInterface` Panel.

Next, create event handlers for the Add Products from Shipment and Cancel Button controls in the inserting interface. When either of these Buttons is clicked, we need to revert back to the display interface. Create `Click` event handlers for both Button controls so that they call `ReturnToDisplayInterface`, a method we will add momentarily. In addition to hiding the `InsertingInterface` Panel and showing the `DisplayInterface` Panel, the `ReturnToDisplayInterface` method needs to return the Web controls to their pre-editing state. This involves setting the DropDownLists `SelectedIndex` properties to 0 and clearing out the `Text` properties of the TextBox controls.

NOTE

Consider what might happen if we didn't return the controls to their pre-editing state before returning to the display interface. A user might click the Process Product Shipment button, enter the products from the shipment, and then click Add Products from Shipment. This would add the products and return the user to the display interface. At this point the user might want to add another shipment. Upon clicking the Process Product Shipment button they would return to the inserting interface but the DropDownList selections and TextBox values would still be populated with their previous values.

```
Protected Sub AddProducts_Click(sender As Object, e As EventArgs) _
    Handles AddProducts.Click
    ' TODO: Save the products
    ' Revert to the display interface
    ReturnToDisplayInterface()
End Sub
Protected Sub CancelButton_Click(sender As Object, e As EventArgs) _
    Handles CancelButton.Click
    ' Revert to the display interface
    ReturnToDisplayInterface()
End Sub
Const firstControlID As Integer = 1
Const lastControlID As Integer = 5
Private Sub ReturnToDisplayInterface()
    ' Reset the control values in the inserting interface
    Suppliers.SelectedIndex = 0
    Categories.SelectedIndex = 0
    For i As Integer = firstControlID To lastControlID
        CType(InsertingInterface.FindControl _
            ("ProductName" + i.ToString()), TextBox).Text = String.Empty
        CType(InsertingInterface.FindControl _
            ("UnitPrice" + i.ToString()), TextBox).Text = String.Empty
    Next
    DisplayInterface.Visible = True
    InsertingInterface.Visible = False
End Sub
```

Both `Click` event handlers simply call the `ReturnToDisplayInterface` method, although we'll return to the Add Products from Shipment `Click` event handler in Step 4 and add code to save the products. `ReturnToDisplayInterface` starts by returning the `Suppliers` and `Categories` DropDownLists to their first options. The two constants `firstControlID` and `lastControlID` mark the starting and ending control index values used in naming the product name and unit price TextBoxes in the inserting interface and are used in the bounds of the `For` loop that sets the `Text` properties of the TextBox controls back to an empty string. Finally, the Panels `Visible` properties are reset so that the inserting interface is hidden and the display interface shown.

Take a moment to test out this page in a browser. When first visiting the page you should see the display interface as was shown in Figure 5. Click the Process Product Shipment button. The page will postback and you should now see the inserting interface as shown in Figure 12. Clicking either the Add Products from Shipment or Cancel buttons returns you to the display interface.

NOTE

While viewing the inserting interface, take a moment to test out the CompareValidators on the unit price TextBoxes. You should see a client-side messagebox warning when clicking the Add Products from Shipment button with invalid currency values or prices with a value less than zero.

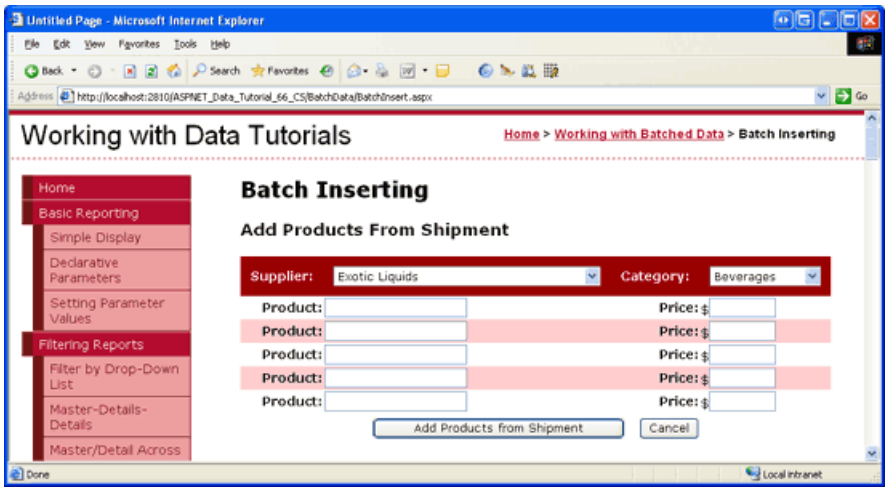


Figure 12: The Inserting Interface is Displayed After Clicking the Process Product Shipment Button (Click to view full-size image)

Step 4: Adding the Products

All that remains for this tutorial is to save the products to the database in the Add Products from Shipment Button s `Click` event handler. This can be accomplished by creating a `ProductsDataTable` and adding a `ProductsRow` instance for each of the product names supplied. Once these `ProductsRow` s have been added we will make a call to the `ProductsBLL` class s `UpdateWithTransaction` method passing in the `ProductsDataTable`. Recall that the `UpdateWithTransaction` method, which was created back in the Wrapping Database Modifications within a Transaction tutorial, passes the `ProductsDataTable` to the `ProductsTableAdapter` 's `UpdateWithTransaction` method. From there, an ADO.NET transaction is started and the TableAdapter issues an `INSERT` statement to the database for each added `ProductsRow` in the DataTable. Assuming all products are added without error, the transaction is committed, otherwise it is rolled back.

The code for the Add Products from Shipment Button s `Click` event handler also needs to perform a bit of error checking. Since there are no RequiredFieldValidators used in the inserting interface, a user could enter a price for a product while omitting its name. Since the product s name is required, if such a condition unfolds we need to alert the user and not proceed with the inserts. The complete `Click` event handler code follows:

```
Protected Sub AddProducts_Click(sender As Object, e As EventArgs) _
    Handles AddProducts.Click
    ' Make sure that the UnitPrice CompareValidators report valid data...
    If Not Page.IsValid Then Exit Sub
    ' Add new ProductsRows to a ProductsDataTable...
    Dim products As New Northwind.ProductsDataTable()
    For i As Integer = firstControlID To lastControlID
        ' Read in the values for the product name and unit price
        Dim productName As String = CType(InsertingInterface.FindControl _
            ("ProductName" + i.ToString()), TextBox).Text.Trim()
        Dim unitPrice As String = CType(InsertingInterface.FindControl _
            ("UnitPrice" + i.ToString()), TextBox).Text.Trim()
        ' Ensure that if unitPrice has a value, so does productName
        If unitPrice.Length > 0 AndAlso productName.Length = 0 Then
            ' Display a warning and exit this event handler
            StatusLabel.Text = "If you provide a unit price you must also
                                include the name of the product."
            StatusLabel.Visible = True
            Exit Sub
        End If
        ' Only add the product if a product name value is provided
        If productName.Length > 0 Then
            ' Add a new ProductsRow to the ProductsDataTable
            Dim newProduct As Northwind.ProductsRow = products.NewProductsRow()
            ' Assign the values from the web page
            newProduct.ProductName = productName
            newProduct.SupplierID = Convert.ToInt32(Suppliers.SelectedValue)
            newProduct.CategoryID = Convert.ToInt32(Categories.SelectedValue)
            If unitPrice.Length > 0 Then
                newProduct.UnitPrice = Convert.ToDecimal(unitPrice)
            End If
            ' Add any "default" values
            newProduct.Discontinued = False
            newProduct.UnitsOnOrder = 0
            products.AddProductsRow(newProduct)
        End If
    Next
    ' If we reach here, see if there were any products added
    If products.Count > 0 Then
        ' Add the new products to the database using a transaction
        Dim productsAPI As New ProductsBLL()
        productsAPI.UpdateWithTransaction(products)
        ' Rebind the data to the grid so that the products just added are displayed
        ProductsGrid.DataBind()
        ' Display a confirmation (don't use the Warning CSS class, though)
        StatusLabel.CssClass = String.Empty
        StatusLabel.Text = String.Format( _
            "{0} products from supplier {1} have been " & _
            "added and filed under category {2}.", _
            products.Count, Suppliers.SelectedItem.Text, Categories.SelectedItem.Text)
        StatusLabel.Visible = True
        ' Revert to the display interface
        ReturnToDisplayInterface()
    Else
        ' No products supplied!
        StatusLabel.Text =
            "No products were added. Please enter the " & _
            "product names and unit prices in the textboxes."
        StatusLabel.Visible = True
    End If
End Sub
```

The event handler starts by ensuring that the `Page.IsValid` property returns a value of `True`. If it returns `False`, then that means one or more of the CompareValidators are reporting invalid data; in such a case we do not want to attempt to insert the entered products or we'll end up with an exception when attempting to assign the user-

entered unit price value to the `ProductsRow` s `UnitPrice` property.

Next, a new `ProductsDataTable` instance is created (`products`). A `For` loop is used to iterate through the product name and unit price TextBoxes and the `Text` properties are read into the local variables `productName` and `unitPrice` . If the user has entered a value for the unit price but not for the corresponding product name, the `StatusLabel` displays the message If you provide a unit price you must also include the name of the product and the event handler is exited.

If a product name has been provided, a new `ProductsRow` instance is created using the `ProductsDataTable` s `NewProductsRow` method. This new `ProductsRow` instance s `ProductName` property is set to the current product name TextBox while the `SupplierID` and `CategoryID` properties are assigned to the `SelectedValue` properties of the DropDownLists in the inserting interface s header. If the user entered a value for the product s price, it is assigned to the `ProductsRow` instance s `UnitPrice` property; otherwise, the property is left unassigned, which will result in a `NULL` value for `UnitPrice` in the database. Finally, the `Discontinued` and `UnitsOnOrder` properties are assigned to the hard-coded values `False` and 0, respectively.

After the properties have been assigned to the `ProductsRow` instance it is added to the `ProductsDataTable` .

At the completion of the `For` loop, we check whether any products have been added. The user may, after all, have clicked the Add Products from Shipment before entering any product names or prices. If there is at least one product in the `ProductsDataTable` , the `ProductsBLL` class s `UpdateWithTransaction` method is called. Next, the data is rebound to the `ProductsGrid` GridView so that the newly added products will appear in the display interface. The `StatusLabel` is updated to display a confirmation message and the `ReturnToDisplayInterface` is invoked, hiding the inserting interface and showing the display interface.

If no products were entered, the inserting interface remains displayed but the message No products were added. Please enter the product names and unit prices in the textboxes is displayed.

Figure s 13, 14, and 15 show the inserting and display interfaces in action. In Figure 13, the user has entered a unit price value without a corresponding product name. Figure 14 shows the display interface after three new products have been added successfully, while Figure 15 shows two of the newly added products in the GridView (the third one is on the previous page).

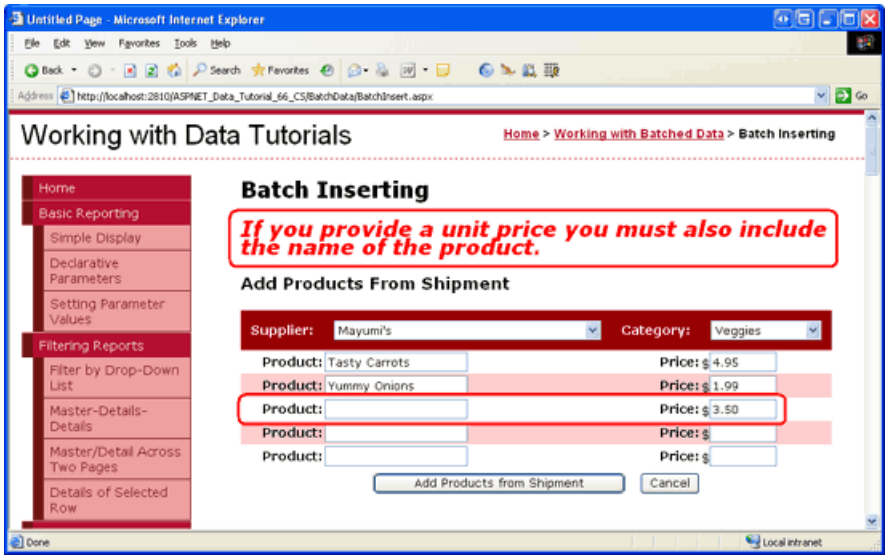


Figure 13: A Product Name is Required When Entering a Unit Price (Click to view full-size image)

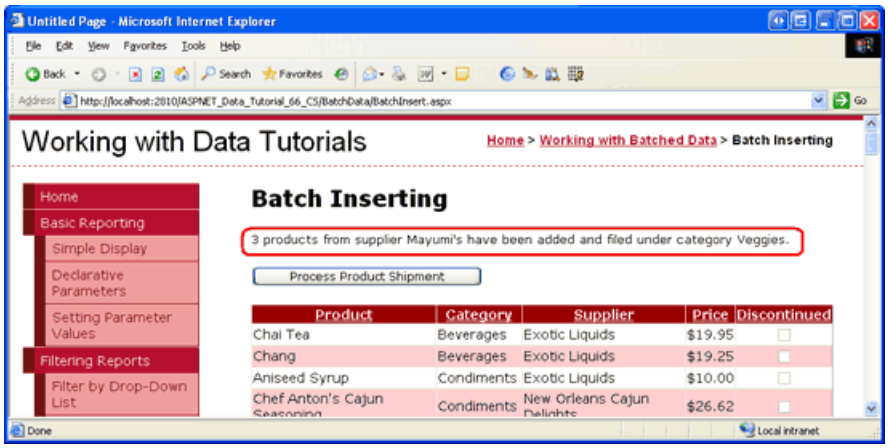


Figure 14: Three New Veggies Have Been Added for the Supplier Mayumi s (Click to view full-size image)

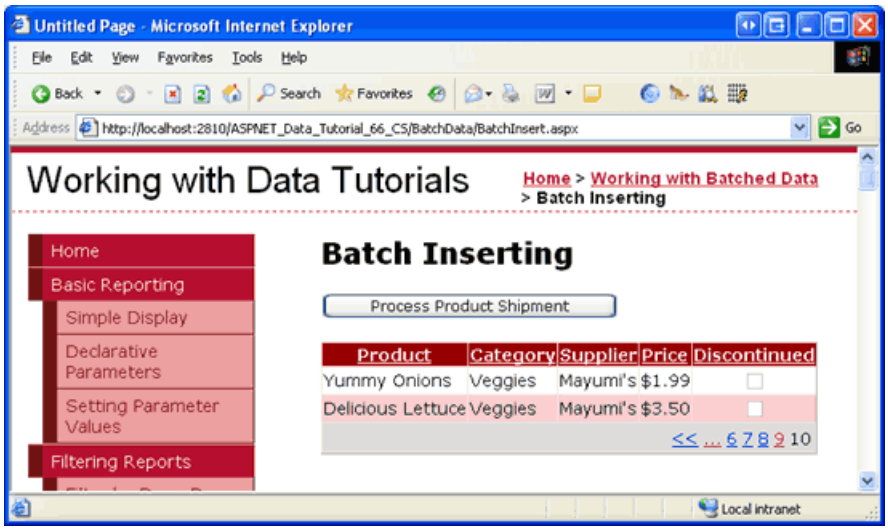


Figure 15: The New Products Can Be Found in the Last Page of the GridView (Click to view full-size image)

> **NOTE**
>
> The batch inserting logic used in this tutorial wraps the inserts within the scope of transaction. To verify this, purposefully introduce a database-level error. For example, rather than assigning the new `ProductsRow` instance s `CategoryID` property to the selected value in the `Categories` DropDownList, assign it to a value like `i * 5`. Here `i` is the loop indexer and has values ranging from 1 to 5. Therefore, when adding two or more products in batch insert the first product will have a valid `CategoryID` value (5), but subsequent products will have `CategoryID` values that do not match up to `CategoryID` values in the `Categories` table. The net effect is that while the first `INSERT` will succeed, subsequent ones will fail with a foreign key constraint violation. Since the batch insert is atomic, the first `INSERT` will be rolled back, returning the database to its state before the batch insert process began.

Summary

Over this and the previous two tutorials we have created interfaces that allow for updating, deleting, and inserting batches of data, all of which used the transaction support we added to the Data Access Layer in the Wrapping Database Modifications within a Transaction tutorial. For certain scenarios, such batch processing user interfaces greatly improve end user efficiency by cutting down on the number of clicks, postbacks, and keyboard-to-mouse context switches, while also maintaining the integrity of the underlying data.

This tutorial completes our look at working with batched data. The next set of tutorials explores a variety of advanced Data Access Layer scenarios, including using stored procedures in the TableAdapter s methods, configuring connection- and command-level settings in the DAL, encrypting connection strings, and more!

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Hilton Giesenow and S ren Jacob Lauritsen. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Advanced Data Access Scenarios

2/20/2020 • 2 minutes to read • Edit Online

These tutorials cover how to work with stored procedures and computed columns, how to use joins with database tables, and how to encrypt data connection information.

- Creating New Stored Procedures for the Typed DataSet's TableAdapters (C#)
- Using Existing Stored Procedures for the Typed DataSet's TableAdapters (C#)
- Updating the TableAdapter to Use JOINs (C#)
- Adding Additional DataTable Columns (C#)
- Working with Computed Columns (C#)
- Configuring the Data Access Layer's Connection- and Command-Level Settings (C#)
- Protecting Connection Strings and Other Configuration Information (C#)
- Debugging Stored Procedures (C#)
- Creating Stored Procedures and User-Defined Functions with Managed Code (C#)
- Creating New Stored Procedures for the Typed DataSet's TableAdapters (VB)
- Using Existing Stored Procedures for the Typed DataSet's TableAdapters (VB)
- Updating the TableAdapter to Use JOINs (VB)
- Adding Additional DataTable Columns (VB)
- Working with Computed Columns (VB)
- Configuring the Data Access Layer's Connection- and Command-Level Settings (VB)
- Protecting Connection Strings and Other Configuration Information (VB)
- Debugging Stored Procedures (VB)
- Creating Stored Procedures and User-Defined Functions with Managed Code (VB)

Creating New Stored Procedures for the Typed DataSet's TableAdapters (C#)

2/20/2020 • 26 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> In earlier tutorials we have created SQL statements in our code and passed the statements to the database to be executed. An alternative approach is to use stored procedures, where the SQL statements are pre-defined at the database. In this tutorial we learn how to have the TableAdapter Wizard generate new stored procedures for us.

Introduction

The Data Access Layer (DAL) for these tutorials uses Typed DataSets. As discussed in the Creating a Data Access Layer tutorial, Typed DataSets consist of strongly-typed DataTables and TableAdapters. The DataTables represent the logical entities in the system while the TableAdapters interface with the underlying database to perform the data access work. This includes populating the DataTables with data, executing queries that return scalar data, and inserting, updating, and deleting records from the database.

The SQL commands executed by the TableAdapters can be either ad-hoc SQL statements, such as `SELECT columnList FROM TableName`, or stored procedures. The TableAdapters in our architecture use ad-hoc SQL statements. Many developers and database administrators, however, prefer stored procedures over ad-hoc SQL statements for security, maintainability, and updatability reasons. Others ardently prefer ad-hoc SQL statements for their flexibility. In my own work I favor stored procedures over ad-hoc SQL statements, but chose to use ad-hoc SQL statements to simplify the earlier tutorials.

When defining a TableAdapter or adding new methods, the TableAdapter s wizard makes it just as easy to create new stored procedures or use existing stored procedures as it does to use ad-hoc SQL statements. In this tutorial we'll examine how to have the TableAdapter s wizard auto-generate stored procedures. In the next tutorial we will look at how to configure the TableAdapter s methods to use existing or manually-created stored procedures.

> **NOTE**
>
> See Rob Howard's blog entry Don't Use Stored Procedures Yet? and Frans Bouma s blog entry Stored Procedures are Bad, M Kay? for a lively debate on the pros and cons of stored procedures and ad-hoc SQL.

Stored Procedure Basics

Functions are a construct common to all programming languages. A function is a collection of statements that are executed when the function is called. Functions can accept input parameters and may optionally return a value. *Stored procedures* are database constructs that share many similarities with functions in programming languages. A stored procedure is made up of a set of T-SQL statements that are executed when the stored procedure is called. A stored procedure may accept zero to many input parameters and can return scalar values, output parameters, or, most commonly, result sets from `SELECT` queries.

> **NOTE**
> Stored procedures are oftentimes referred to as sprocs or SPs .

Stored procedures are created using the `CREATE PROCEDURE` T-SQL statement. For example, the following T-SQL script creates a stored procedure named `GetProductsByCategoryID` that takes in a single parameter named `@CategoryID` and returns the `ProductID`, `ProductName`, `UnitPrice`, and `Discontinued` fields of those columns in the `Products` table that have a matching `CategoryID` value:

```
CREATE PROCEDURE GetProductsByCategoryID
(
    @CategoryID int
)
AS
SELECT ProductID, ProductName, UnitPrice, Discontinued
FROM Products
WHERE CategoryID = @CategoryID
```

Once this stored procedure has been created, it can be called using the following syntax:

```
EXEC GetProductsByCategory categoryID
```

> **NOTE**
> In the next tutorial we will examine creating stored procedures through the Visual Studio IDE. For this tutorial, however, we are going to let the TableAdapter wizard automatically generate the stored procedures for us.

In addition to simply returning data, stored procedures are often used to perform multiple database commands within the scope of a single transaction. A stored procedure named `DeleteCategory`, for example, might take in a `@CategoryID` parameter and perform two `DELETE` statements: first, one to delete the related products and a second one deleting the specified category. Multiple statements within a stored procedure are *not* automatically wrapped within a transaction. Additional T-SQL commands need to be issued to ensure the stored procedure s multiple commands are treated as an atomic operation. We'll see how to wrap a stored procedure s commands within the scope of a transaction in the subsequent tutorial.

When using stored procedures within an architecture, the Data Access Layer s methods invoke a particular stored procedure rather than issuing an ad-hoc SQL statement. This centralizes the location of the SQL statements executed (on the database) rather than having it defined within the application s architecture. This centralization arguably makes it easier to find, analyze, and tune the queries and provides a much clearer picture as to where and how the database is being used.

For more information on stored procedure fundamentals, consult the resources in the Further Reading section at the end of this tutorial.

Step 1: Creating the Advanced Data Access Layer Scenarios Web Pages

Before we start our discussion on creating a DAL using stored procedures, let s first take a moment to create the ASP.NET pages in our website project that we will need for this and the next several tutorials. Start by adding a new folder named `AdvancedDAL`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `NewSprocs.aspx`

- `ExistingSprocs.aspx`
- `JOINs.aspx`
- `AddingColumns.aspx`
- `ComputedColumns.aspx`
- `EncryptingConfigSections.aspx`
- `ManagedFunctionsAndSprocs.aspx`

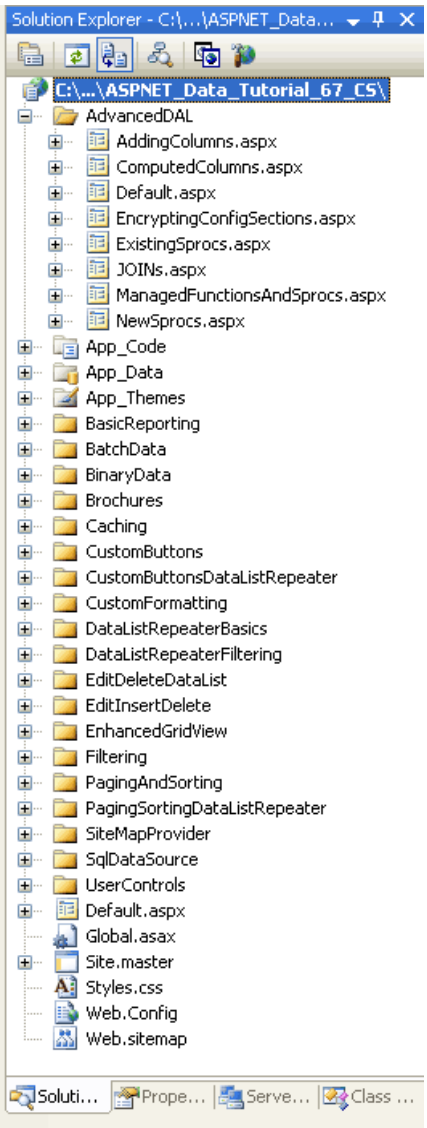


Figure 1: Add the ASP.NET Pages for the Advanced Data Access Layer Scenarios Tutorials

Like in the other folders, `Default.aspx` in the `AdvancedDAL` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

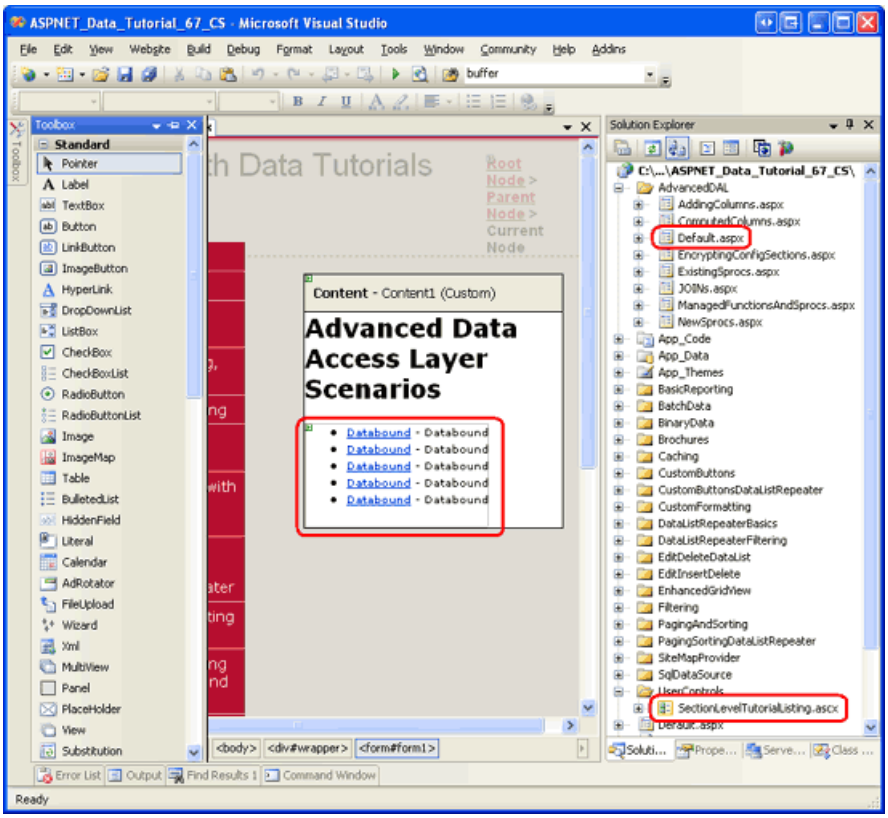


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add these pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Working with Batched Data `<siteMapNode>`:

```
<siteMapNode url="~/AdvancedDAL/Default.aspx"
    title="Advanced DAL Scenarios"
    description="Explore a number of advanced Data Access Layer scenarios.">

    <siteMapNode url="~/AdvancedDAL/NewSprocs.aspx"
        title="Creating New Stored Procedures for TableAdapters"
        description="Learn how to have the TableAdapter wizard automatically
            create and use stored procedures." />
    <siteMapNode url="~/AdvancedDAL/ExistingSprocs.aspx"
        title="Using Existing Stored Procedures for TableAdapters"
        description="See how to plug existing stored procedures into a
            TableAdapter." />
    <siteMapNode url="~/AdvancedDAL/JOINs.aspx"
        title="Returning Data Using JOINs"
        description="Learn how to augment your DataTables to work with data
            returned from multiple tables via a JOIN query." />
    <siteMapNode url="~/AdvancedDAL/AddingColumns.aspx"
        title="Adding DataColumns to a DataTable"
        description="Master adding new columns to an existing DataTable." />
    <siteMapNode url="~/AdvancedDAL/ComputedColumns.aspx"
        title="Working with Computed Columns"
        description="Explore how to work with computed columns when using
            Typed DataSets." />
    <siteMapNode url="~/AdvancedDAL/EncryptingConfigSections.aspx"
        title="Protected Connection Strings in Web.config"
        description="Protect your connection string information in
            Web.config using encryption." />
    <siteMapNode url="~/AdvancedDAL/ManagedFunctionsAndSprocs.aspx"
        title="Creating Managed SQL Functions and Stored Procedures"
        description="See how to create SQL functions and stored procedures
            using managed code." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the advanced DAL scenarios tutorials.

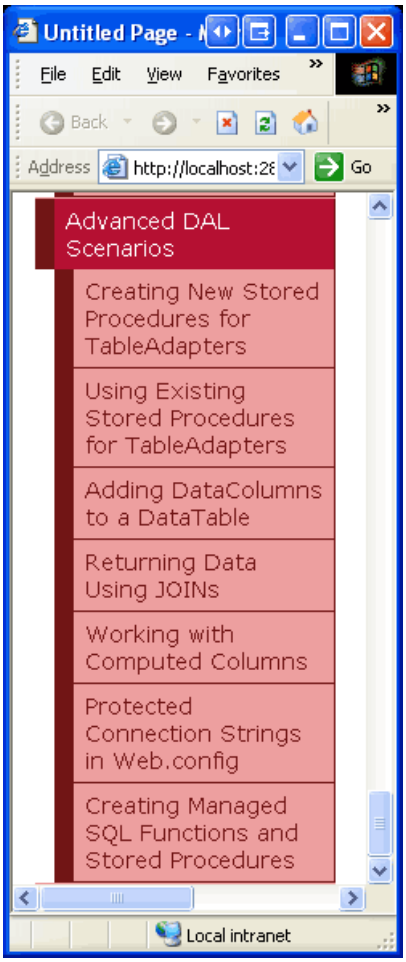


Figure 3: The Site Map Now Includes Entries for the Advanced DAL Scenarios Tutorials

Step 2: Configuring a TableAdapter to Create New Stored Procedures

To demonstrate creating a Data Access Layer that uses stored procedures instead of ad-hoc SQL statements, let s create a new Typed DataSet in the `~/App_Code/DAL` folder named `NorthwindWithSprocs.xsd`. Since we have stepped through this process in detail in previous tutorials, we will proceed quickly through the steps here. If you get stuck or need further step-by-step instructions in creating and configuring a Typed DataSet, refer back to the Creating a Data Access Layer tutorial.

Add a new DataSet to the project by right-clicking on the `DAL` folder, choosing Add New Item, and selecting the DataSet template as shown in Figure 4.

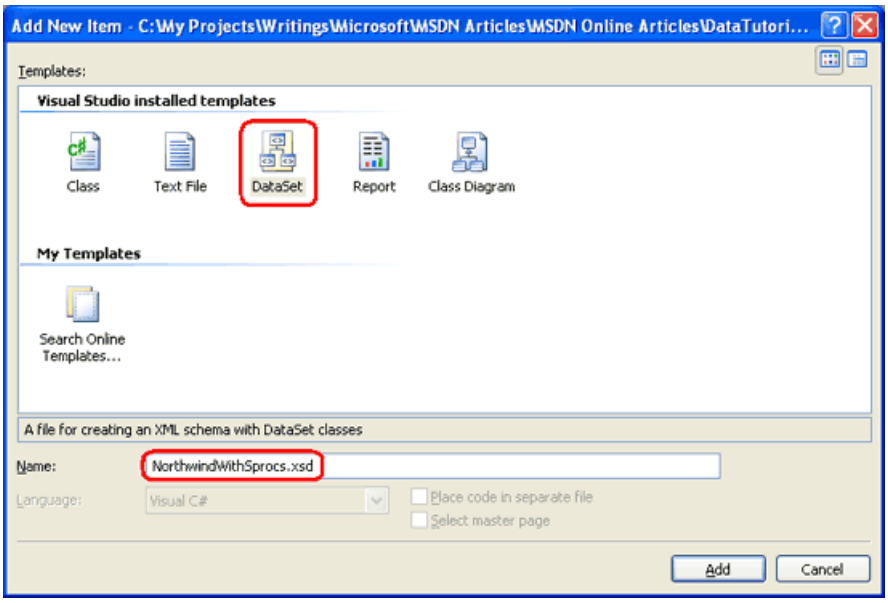


Figure 4: Add a New Typed DataSet to the Project Named `NorthwindWithSprocs.xsd` (Click to view full-size image)

This will create the new Typed DataSet, open its Designer, create a new TableAdapter, and launch the TableAdapter Configuration Wizard. The TableAdapter Configuration Wizard s first step asks us to select the database to work with. The connection string to the Northwind database should be listed in the drop-down list. Select this and click Next.

From this next screen we can choose how the TableAdapter should access the database. In previous tutorials, we selected the first option, Use SQL statements. For this tutorial, select the second option, Create new stored procedures, and click Next.

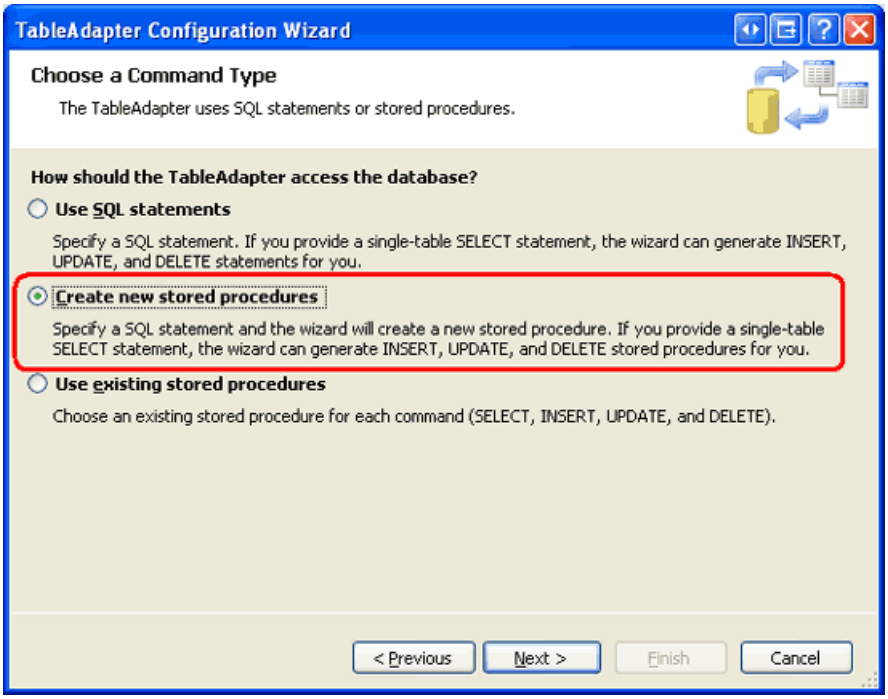


Figure 5: Instruct the TableAdapter to Create New Stored Procedures (Click to view full-size image)

Just like with using ad-hoc SQL statements, in the following step we are asked to provide the `SELECT` statement for the TableAdapter s main query. But instead of using the `SELECT` statement entered here to perform an ad-hoc query directly, the TableAdapter s wizard will create a stored procedure that contains this `SELECT` query.

Use the following `SELECT` query for this TableAdapter:

```
SELECT ProductID, ProductName, SupplierID, CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued
FROM Products
```

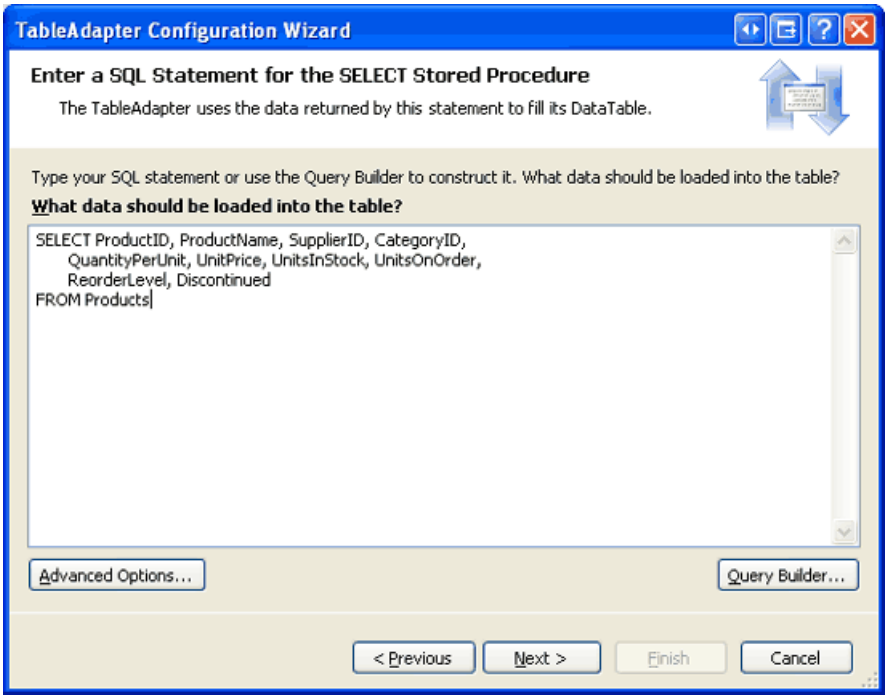


Figure 6: Enter the `SELECT` Query (Click to view full-size image)

> **NOTE**
> The query above differs slightly from the main query of the `ProductsTableAdapter` in the `Northwind` Typed DataSet. Recall that the `ProductsTableAdapter` in the `Northwind` Typed DataSet includes two correlated subqueries to bring back the category name and company name for each product s category and supplier. In the upcoming Updating the TableAdapter to Use JOINs tutorial we will look at adding this related data to this TableAdapter.

Take a moment to click the Advanced Options button. From here we can specify whether the wizard should also generate insert, update, and delete statements for the TableAdapter, whether to use optimistic concurrency, and whether the data table should be refreshed after inserts and updates. The Generate Insert, Update and Delete statements option is checked by default. Leave it checked. For this tutorial, leave the Use optimistic concurrency options unchecked.

When having the stored procedures automatically created by the TableAdapter wizard, it appears that the Refresh the data table option is ignored. Regardless of whether this checkbox is checked, the resulting insert and update stored procedures retrieve the just-inserted or just-updated record, as we will see in Step 3.

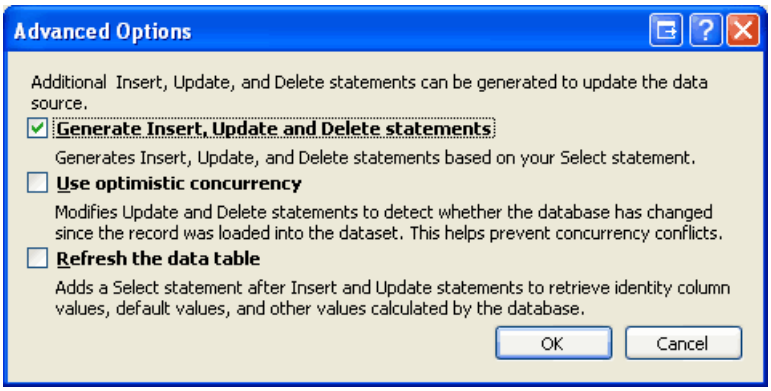


Figure 7: Leave the Generate Insert, Update and Delete statements Option Checked

> **NOTE**
>
> If the Use optimistic concurrency option is checked, the wizard will add additional conditions to the `WHERE` clause that prevent data from being updated if there were changes in other fields. Refer back to the Implementing Optimistic Concurrency tutorial for more information on using the TableAdapter s built-in optimistic concurrency control feature.

After entering the `SELECT` query and confirming that the Generate Insert, Update and Delete statements option is checked, click Next. This next screen, shown in Figure 8, prompts for the names of the stored procedures the wizard will create for selecting, inserting, updating, and deleting data. Change these stored procedures names to `Products_Select`, `Products_Insert`, `Products_Update`, and `Products_Delete`.

Figure 8: Rename the Stored Procedures (Click to view full-size image)

To see the T-SQL the TableAdapter wizard will use to create the four stored procedures, click the Preview SQL Script button. From the Preview SQL Script dialog box you may save the script to a file or copy it to the clipboard.

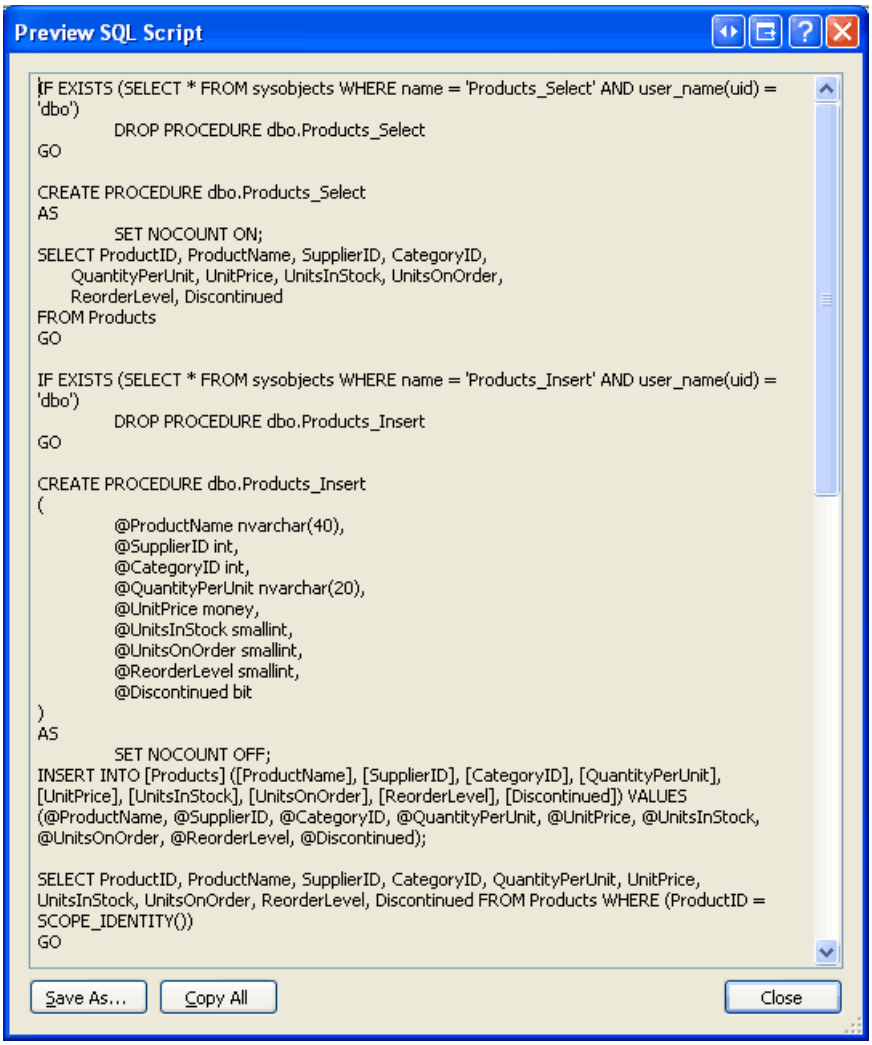


Figure 9: Preview the SQL Script Used to Generate the Stored Procedures

After naming the stored procedures, click Next to name the TableAdapter s corresponding methods. Just like when using ad-hoc SQL statements, we can create methods that fill an existing DataTable or return a new one. We can also specify whether the TableAdapter should include the DB-Direct pattern for inserting, updating, and deleting records. Leave all three checkboxes checked, but rename the Return a DataTable method to `GetProducts` (as shown in Figure 10).

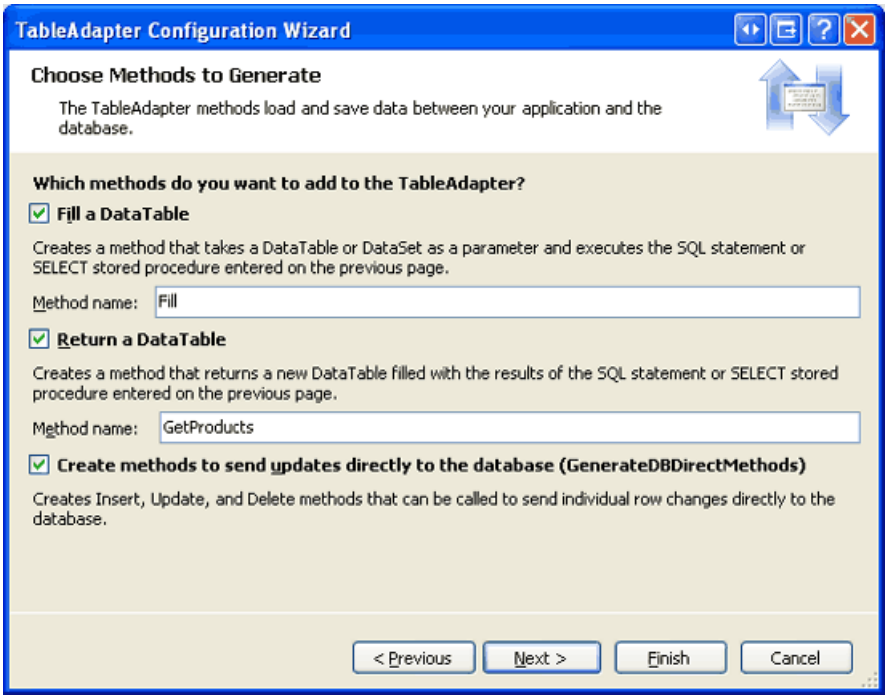


Figure 10: Name the Methods `Fill` and `GetProducts` (Click to view full-size image)

Click Next to see a summary of the steps the wizard will perform. Complete the wizard by clicking the Finish button. Once the wizard completes, you will be returned to the DataSet s Designer, which should now include the `ProductsDataTable`.

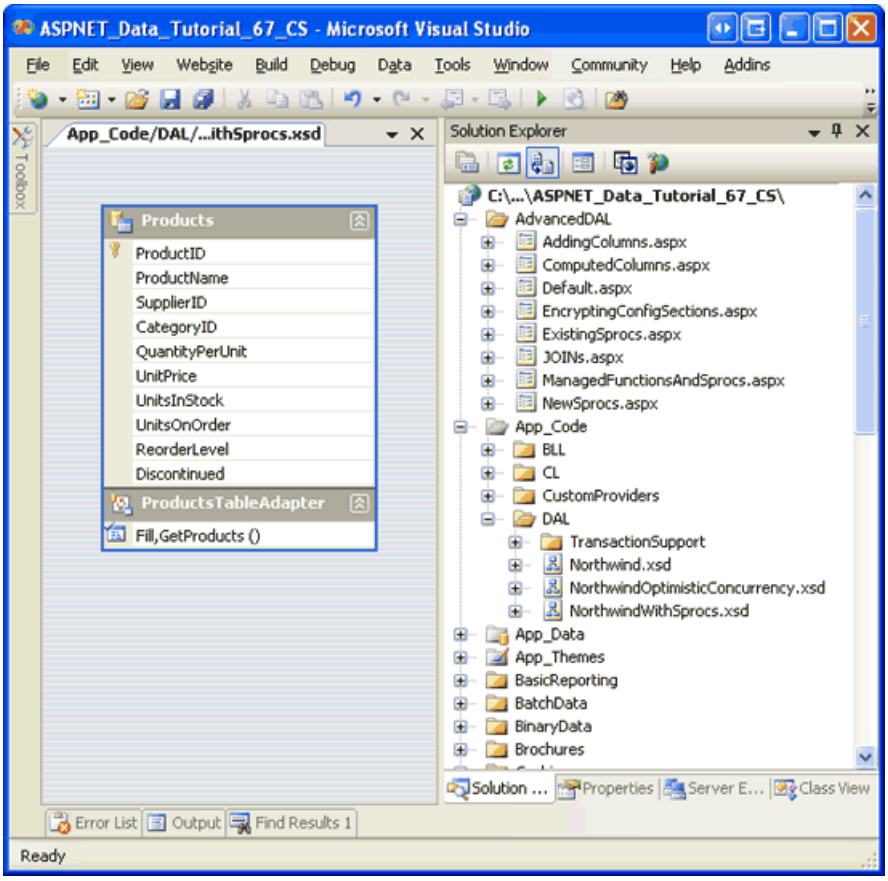


Figure 11: The DataSet s Designer Shows the Newly Added `ProductsDataTable` (Click to view full-size image)

Step 3: Examining the Newly Created Stored Procedures

The TableAdapter wizard used in Step 2 automatically created the stored procedures for selecting, inserting, updating, and deleting data. These stored procedures can be viewed or modified through Visual Studio by going to the Server Explorer and drilling down into the database s Stored Procedures folder. As Figure 12 shows, the Northwind database contains four new stored procedures: `Products_Delete`, `Products_Insert`, `Products_Select`, and `Products_Update`.

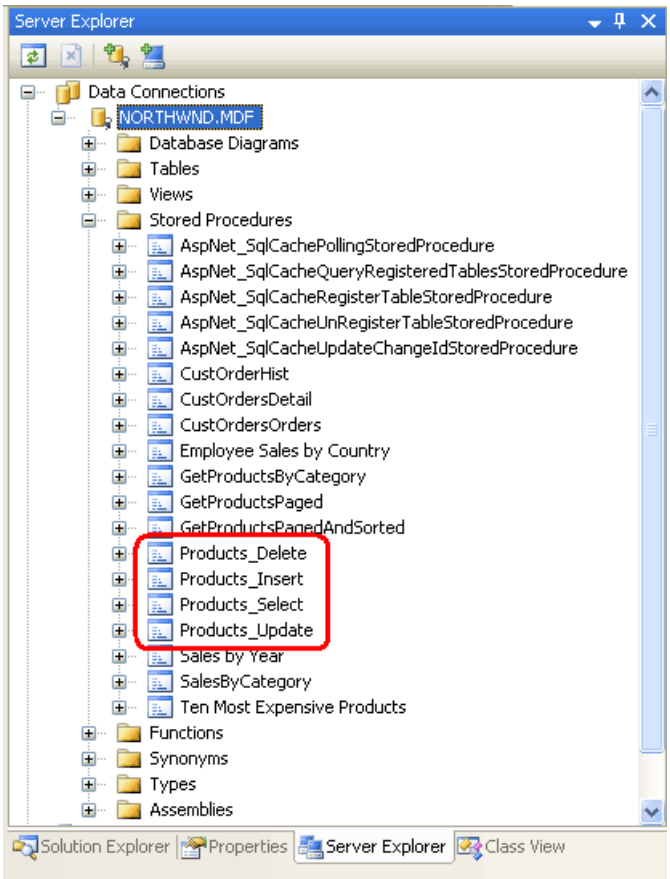


Figure 12: The Four Stored Procedures Created in Step 2 Can Be Found in the Database s Stored Procedures Folder

> **NOTE**
>
> If you do not see the Server Explorer, go to the View menu and choose the Server Explorer option. If you do not see the product-related stored procedures added from Step 2, try right-clicking on the Stored Procedures folder and choosing Refresh.

To view or modify a stored procedure, double-click its name in the Server Explorer or, alternatively, right-click on the stored procedure and choose Open. Figure 13 shows the `Products_Delete` stored procedure, when opened.

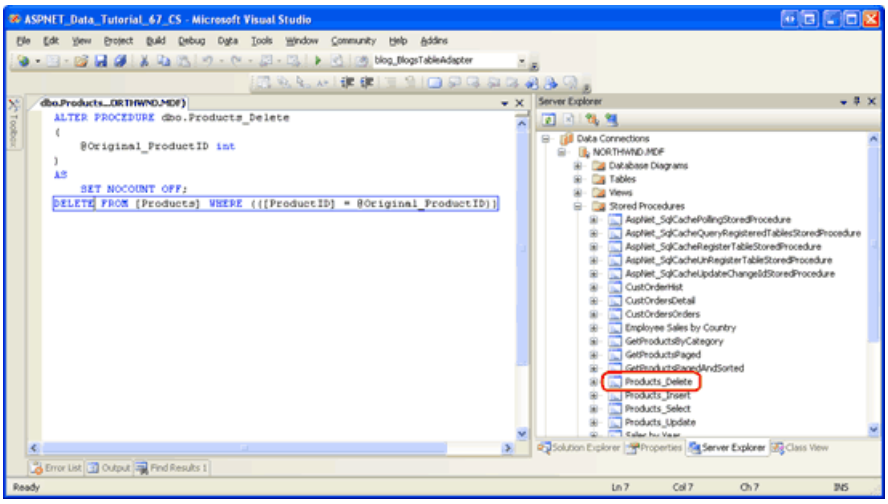


Figure 13: Stored Procedures Can Be Opened and Modified From Within Visual Studio (Click to view full-size image)

The contents of both the `Products_Delete` and `Products_Select` stored procedures are quite straightforward. The `Products_Insert` and `Products_Update` stored procedures, on the other hand, warrant a closer inspection as they

both perform a `SELECT` statement after their `INSERT` and `UPDATE` statements. For example, the following SQL makes up the `Products_Insert` stored procedure:

```
ALTER PROCEDURE dbo.Products_Insert
(
    @ProductName nvarchar(40),
    @SupplierID int,
    @CategoryID int,
    @QuantityPerUnit nvarchar(20),
    @UnitPrice money,
    @UnitsInStock smallint,
    @UnitsOnOrder smallint,
    @ReorderLevel smallint,
    @Discontinued bit
)
AS
    SET NOCOUNT OFF;
INSERT INTO [Products] ([ProductName], [SupplierID], [CategoryID], [QuantityPerUnit],
    [UnitPrice], [UnitsInStock], [UnitsOnOrder], [ReorderLevel], [Discontinued])
VALUES (@ProductName, @SupplierID, @CategoryID, @QuantityPerUnit, @UnitPrice,
    @UnitsInStock, @UnitsOnOrder, @ReorderLevel, @Discontinued);

SELECT ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit, UnitPrice,
    UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued
FROM Products
WHERE (ProductID = SCOPE_IDENTITY())
```

The stored procedure accepts as input parameters the `Products` columns that were returned by the `SELECT` query specified in the TableAdapter s wizard and these values are used in an `INSERT` statement. Following the `INSERT` statement, a `SELECT` query is used to return the `Products` column values (including the `ProductID`) of the newly added record. This refresh capability is useful when adding a new record using the Batch Update pattern as it automatically updates the newly added `ProductRow` instances `ProductID` properties with the auto-incremented values assigned by the database.

The following code illustrates this feature. It contains a `ProductsTableAdapter` and `ProductsDataTable` created for the `NorthwindWithSprocs` Typed DataSet. A new product is added to the database by creating a `ProductsRow` instance, supplying its values, and calling the TableAdapter s `Update` method, passing in the `ProductsDataTable`. Internally, the TableAdapter s `Update` method enumerates the `ProductsRow` instances in the passed-in DataTable (in this example there is only one - the one we just added), and performs the appropriate insert, update, or delete command. In this case, the `Products_Insert` stored procedure is executed, which adds a new record to the `Products` table and returns the details of the newly-added record. The `ProductsRow` instance s `ProductID` value is then updated. After the `Update` method has completed, we can access the newly-added record s `ProductID` value through the `ProductsRow` s `ProductID` property.

```
// Create the ProductsTableAdapter and ProductsDataTable
NorthwindWithSprocsTableAdapters.ProductsTableAdapter productsAPI =
    new NorthwindWithSprocsTableAdapters.ProductsTableAdapter();
NorthwindWithSprocs.ProductsDataTable products =
    new NorthwindWithSprocs.ProductsDataTable();
// Create a new ProductsRow instance and set its properties
NorthwindWithSprocs.ProductsRow product = products.NewProductsRow();
product.ProductName = "New Product";
product.CategoryID = 1;  // Beverages
product.Discontinued = false;
// Add the ProductsRow instance to the DataTable
products.AddProductsRow(product);
// Update the DataTable using the Batch Update pattern
productsAPI.Update(products);
// At this point, we can determine the value of the newly-added record's ProductID
int newlyAddedProductIDValue = product.ProductID;
```

The `Products_Update` stored procedure similarly includes a `SELECT` statement after its `UPDATE` statement.

```
ALTER PROCEDURE dbo.Products_Update
(
    @ProductName nvarchar(40),
    @SupplierID int,
    @CategoryID int,
    @QuantityPerUnit nvarchar(20),
    @UnitPrice money,
    @UnitsInStock smallint,
    @UnitsOnOrder smallint,
    @ReorderLevel smallint,
    @Discontinued bit,
    @Original_ProductID int,
    @ProductID int
)
AS
    SET NOCOUNT OFF;
UPDATE [Products]
SET [ProductName] = @ProductName, [SupplierID] = @SupplierID,
    [CategoryID] = @CategoryID, [QuantityPerUnit] = @QuantityPerUnit,
    [UnitPrice] = @UnitPrice, [UnitsInStock] = @UnitsInStock,
    [UnitsOnOrder] = @UnitsOnOrder, [ReorderLevel] = @ReorderLevel,
    [Discontinued] = @Discontinued
WHERE (([ProductID] = @Original_ProductID));

SELECT ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit,
    UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued
FROM Products
WHERE (ProductID = @ProductID)
```

Note that this stored procedure includes two input parameters for `ProductID`: `@Original_ProductID` and `@ProductID`. This functionality allows for scenarios where the primary key might be changed. For example, in an employee database, each employee record might use the employee s social security number as their primary key. In order to change an existing employee s social security number, both the new social security number and the original one must be supplied. For the `Products` table, such functionality is not needed because the `ProductID` column is an `IDENTITY` column and should never be changed. In fact, the `UPDATE` statement in the `Products_Update` stored procedure doesn t include the `ProductID` column in its column list. So, while `@Original_ProductID` is used in the `UPDATE` statement s `WHERE` clause, it is superfluous for the `Products` table and could be replaced by the `@ProductID` parameter. When modifying a stored procedure s parameters it is important that the TableAdapter method(s) that use that stored procedure are also updated.

Step 4: Modifying a Stored Procedure s Parameters and Updating the

TableAdapter

Since the `@Original_ProductID` parameter is superfluous, let s remove it from the `Products_Update` stored procedure altogether. Open the `Products_Update` stored procedure, delete the `@Original_ProductID` parameter, and, in the `WHERE` clause of the `UPDATE` statement, change the parameter name used from `@Original_ProductID` to `@ProductID`. After making these changes, the T-SQL within the stored procedure should look like the following:

```
ALTER PROCEDURE dbo.Products_Update
(
    @ProductName nvarchar(40),
    @SupplierID int,
    @CategoryID int,
    @QuantityPerUnit nvarchar(20),
    @UnitPrice money,
    @UnitsInStock smallint,
    @UnitsOnOrder smallint,
    @ReorderLevel smallint,
    @Discontinued bit,
    @ProductID int
)
AS
    SET NOCOUNT OFF;
UPDATE [Products] SET [ProductName] = @ProductName, [SupplierID] = @SupplierID,
    [CategoryID] = @CategoryID, [QuantityPerUnit] = @QuantityPerUnit,
    [UnitPrice] = @UnitPrice, [UnitsInStock] = @UnitsInStock,
    [UnitsOnOrder] = @UnitsOnOrder, [ReorderLevel] = @ReorderLevel,
    [Discontinued] = @Discontinued
WHERE (([ProductID] = @ProductID));

SELECT ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit,
    UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued
FROM Products
WHERE (ProductID = @ProductID)
```

To save these changes to the database, click the Save icon in the toolbar or hit Ctrl+S. At this point, the `Products_Update` stored procedure does not expect an `@Original_ProductID` input parameter, but the TableAdapter is configured to pass such a parameter. You can see the parameters the TableAdapter will send to the `Products_Update` stored procedure by selecting the TableAdapter in the DataSet Designer, going to the Properties window, and clicking the ellipses in the `UpdateCommand` s `Parameters` collection. This brings up the Parameters Collection Editor dialog box shown in Figure 14.

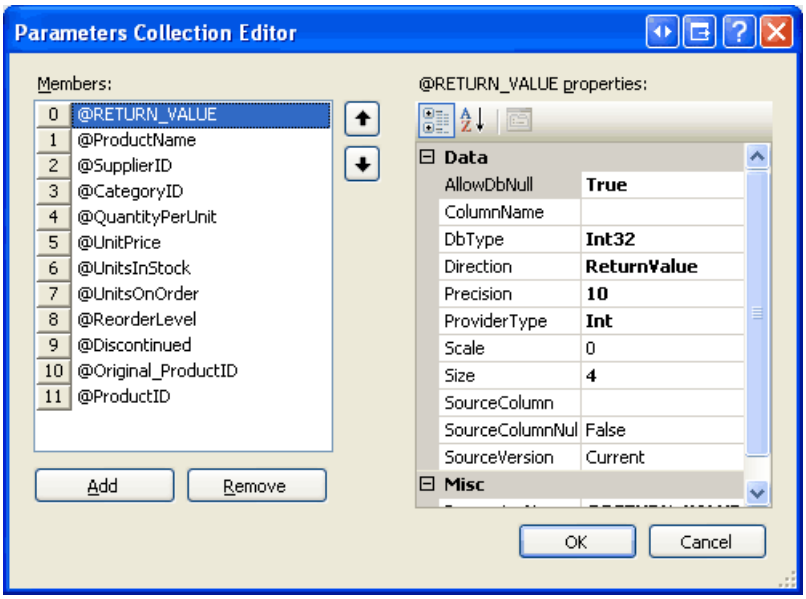


Figure 14: The Parameters Collection Editor Lists the Parameters Used Passed to the `Products_Update` Stored

Procedure

You can remove this parameter from here by simply selecting the `@Original_ProductID` parameter from the list of members and clicking the Remove button.

Alternatively, you can refresh the parameters used for all of the methods by right-clicking on the TableAdapter in the Designer and choosing Configure. This will bring up the TableAdapter Configuration wizard, listing the stored procedures used for selecting, inserting, updating, and deleting, along with the parameters the stored procedures expect to receive. If you click on the Update drop-down list you can see the `Products_Update` stored procedures expected input parameters, which now no longer includes `@Original_ProductID` (see Figure 15). Simply click Finish to automatically update the parameter collection used by the TableAdapter.

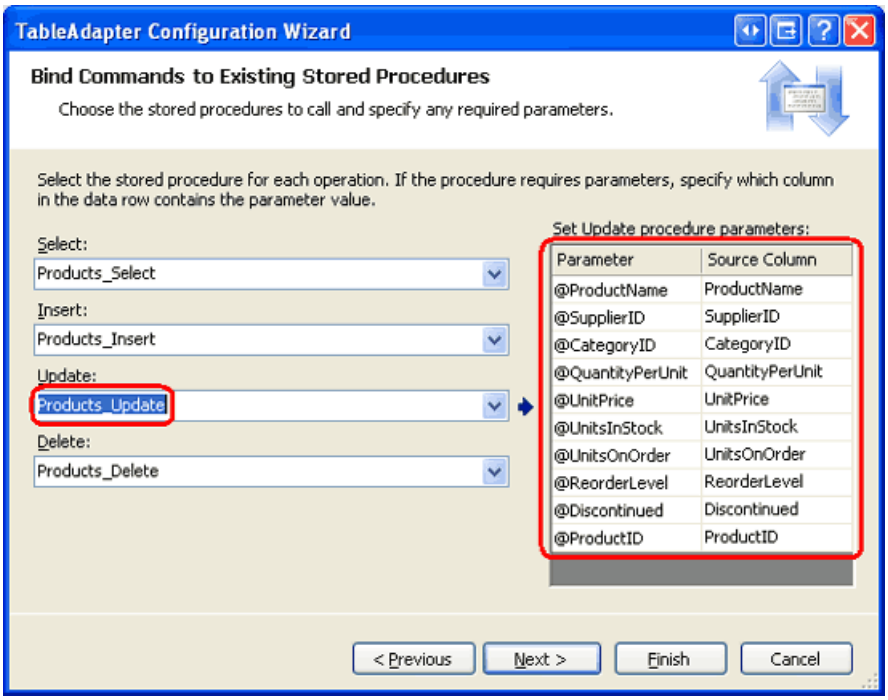


Figure 15: You Can Alternatively Use the TableAdapter s Configuration Wizard to Refresh Its Methods Parameter Collections (Click to view full-size image)

Step 5: Adding Additional TableAdapter Methods

As Step 2 illustrated, when creating a new TableAdapter it is easy to have the corresponding stored procedures automatically generated. The same is true when adding additional methods to a TableAdapter. To illustrate this, let s add a `GetProductByProductID(productID)` method to the `ProductsTableAdapter` created in Step 2. This method will take as input a `ProductID` value and return details about the specified product.

Start by right-clicking on the TableAdapter and choosing Add Query from the context menu.

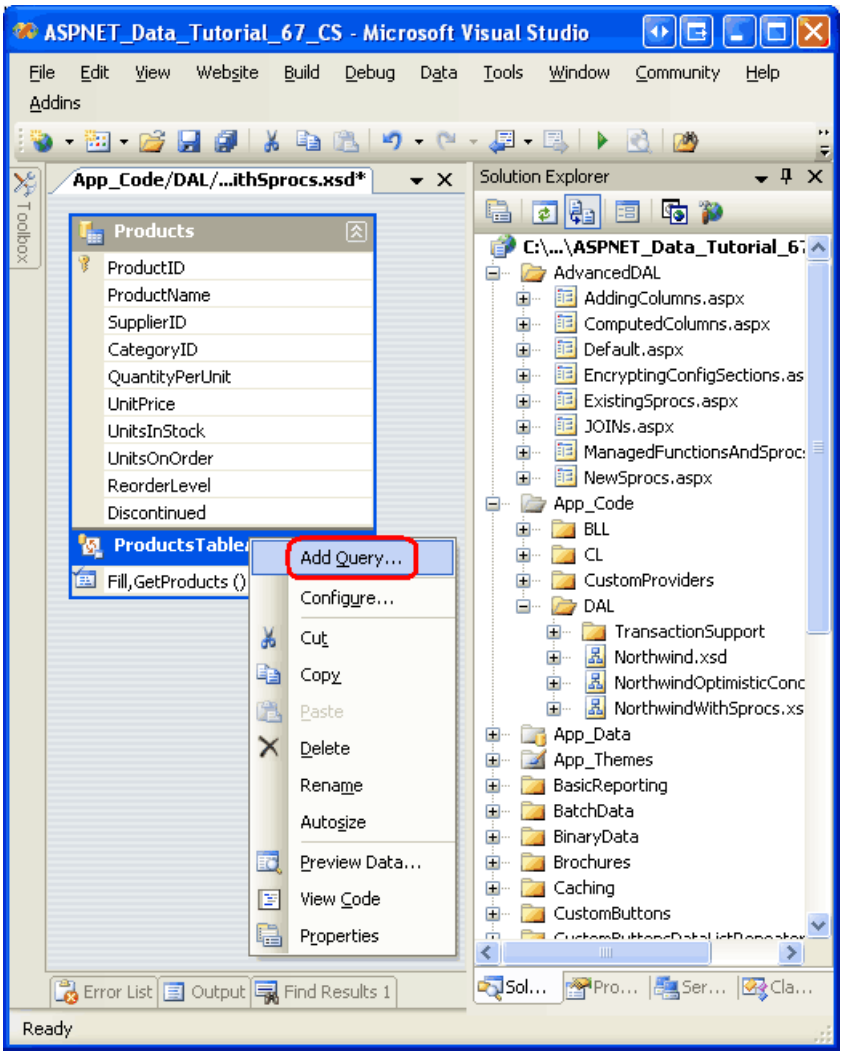


Figure 16: Add a New Query to the TableAdapter

This will start the TableAdapter Query Configuration wizard, which first prompts for how the TableAdapter should access the database. To have a new stored procedure created, choose the Create a new stored procedure option and click Next.

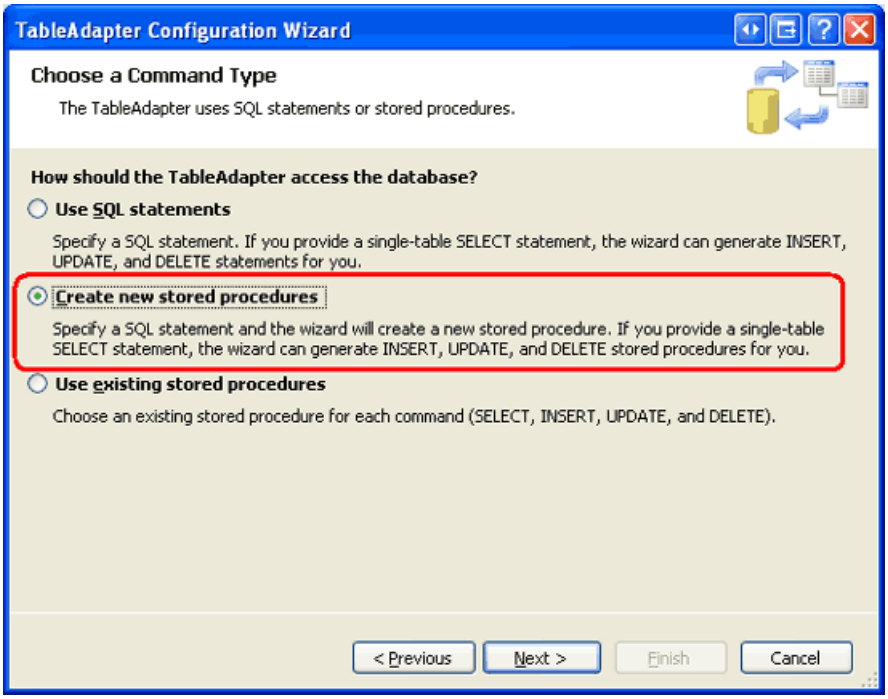


Figure 17: Choose the Create a new stored procedure Option (Click to view full-size image)

The next screen asks us to identify the type of query to execute, whether it will return a set of rows or a single scalar value, or perform an `UPDATE`, `INSERT`, or `DELETE` statement. Since the `GetProductByProductID(productID)` method will return a row, leave the SELECT which returns row option selected and hit Next.

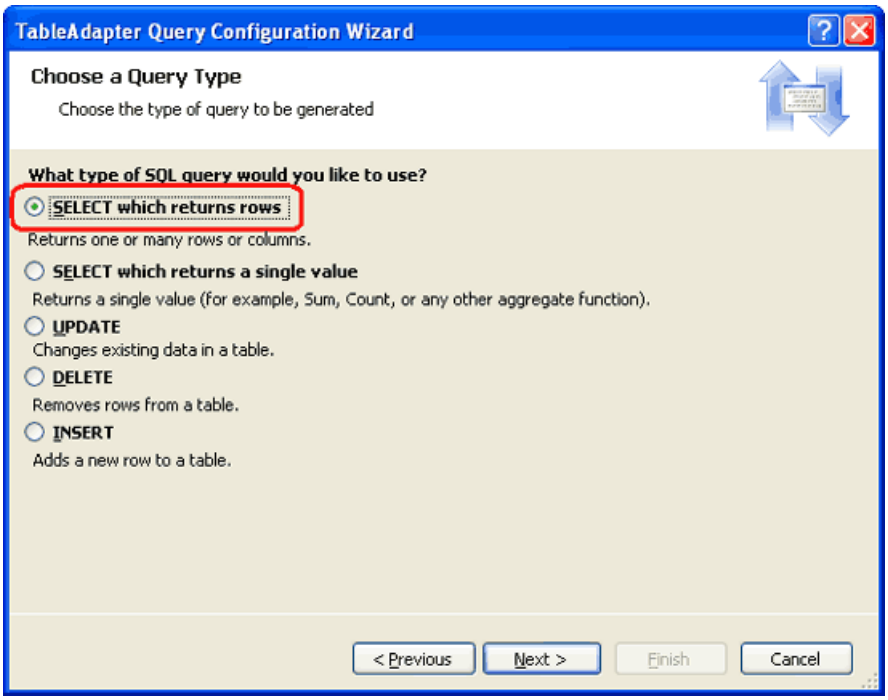


Figure 18: Choose the SELECT which returns row Option (Click to view full-size image)

The next screen displays the TableAdapter s main query, which just lists the name of the stored procedure (`dbo.Products_Select`). Replace the stored procedure name with the following `SELECT` statement, which returns all of the product fields for a specified product:

```
SELECT ProductID, ProductName, SupplierID, CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued
FROM Products
WHERE ProductID = @ProductID
```

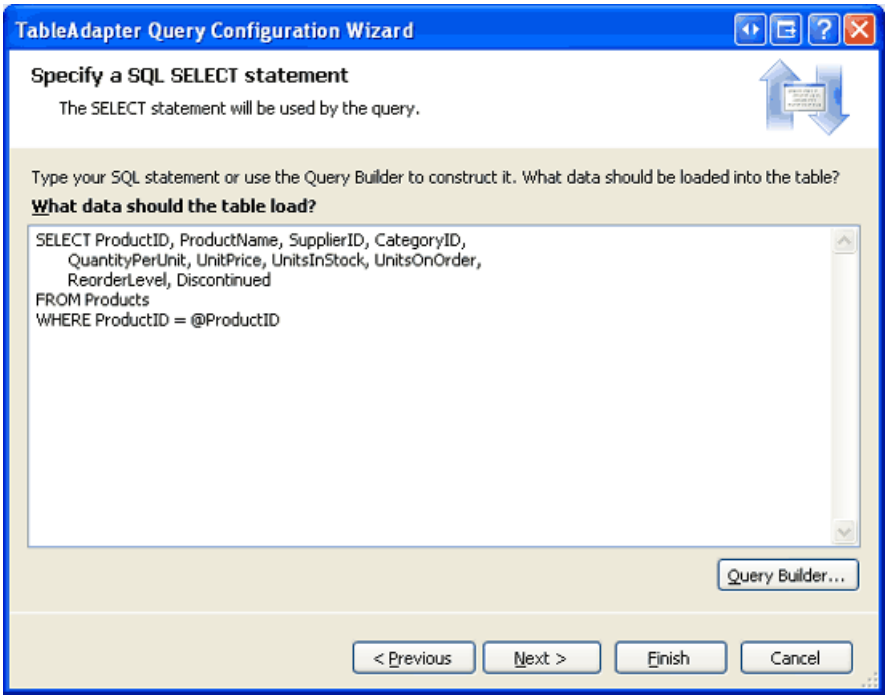


Figure 19: Replace the Stored Procedure Name with a `SELECT` Query (Click to view full-size image)

The subsequent screen asks you to name the stored procedure that will be created. Enter the name `Products_SelectByProductID` and click Next.

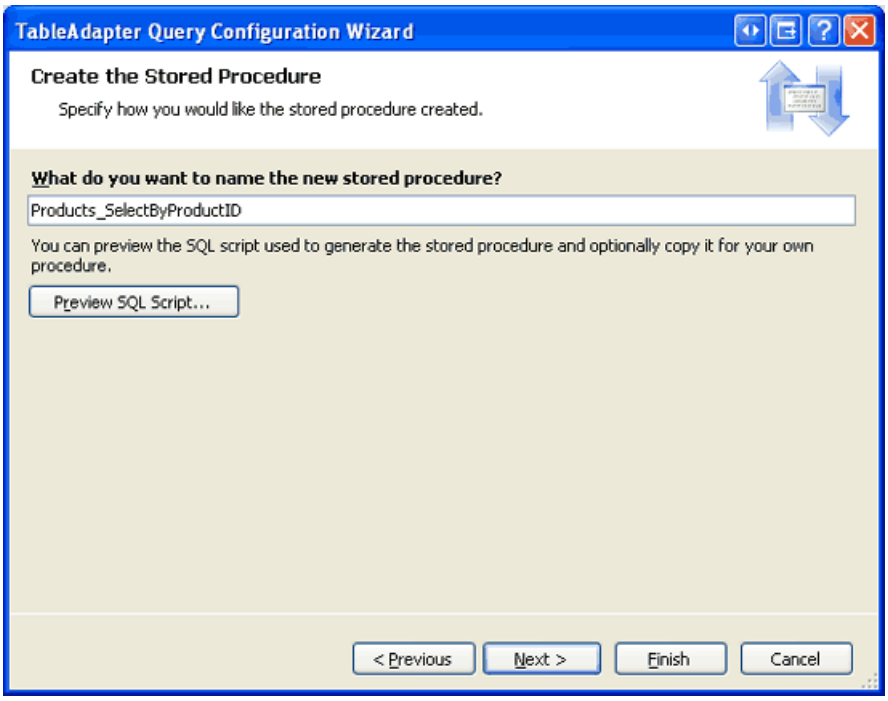


Figure 20: Name the New Stored Procedure `Products_SelectByProductID` (Click to view full-size image)

The final step of the wizard allows us to change the method names generated as well as indicate whether to use the Fill a DataTable pattern, Return a DataTable pattern, or both. For this method, leave both options checked, but rename the methods to `FillByProductID` and `GetProductByProductID`. Click Next to view a summary of the steps the wizard will perform and then click Finish to complete the wizard.

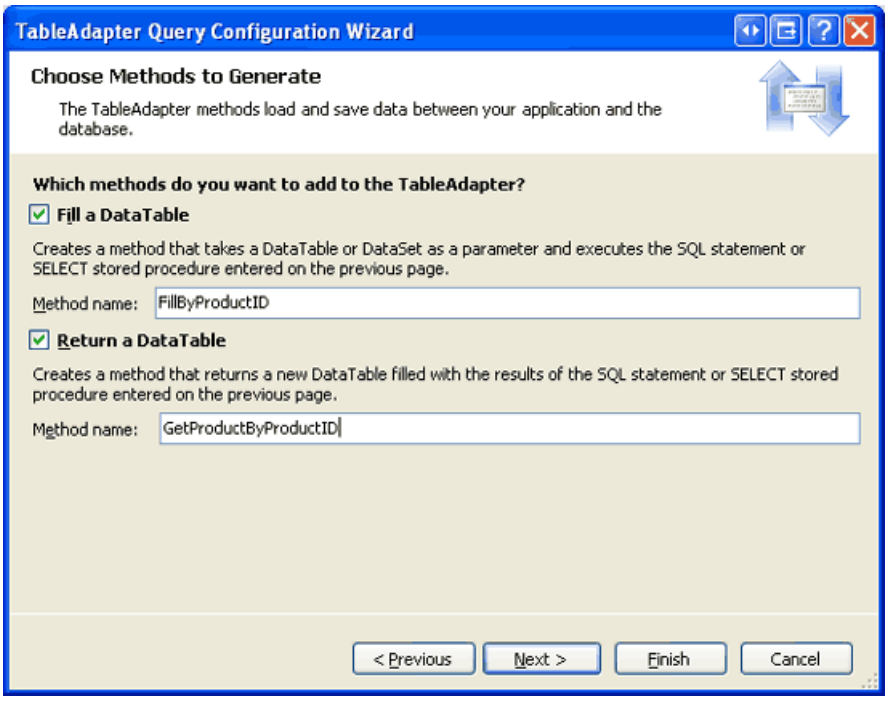


Figure 21: Rename the TableAdapter s Methods to `FillByProductID` and `GetProductByProductID` (Click to view full-size image)

After completing the wizard, the TableAdapter has a new method available, `GetProductByProductID(productID)` that, when invoked, will execute the `Products_SelectByProductID` stored procedure that was just created. Take a moment to view this new stored procedure from the Server Explorer by drilling into the Stored Procedures folder and opening `Products_SelectByProductID` (if you do not see it, right-click on the Stored Procedures folder and

choose Refresh).

Note that the `SelectByProductID` stored procedure takes `@ProductID` as an input parameter and executes the `SELECT` statement that we entered in the wizard.

```
ALTER PROCEDURE dbo.Products_SelectByProductID
(
    @ProductID int
)
AS
    SET NOCOUNT ON;
SELECT ProductID, ProductName, SupplierID, CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued
FROM Products
WHERE ProductID = @ProductID
```

Step 6: Creating a Business Logic Layer Class

Throughout the tutorial series we have strived to maintain a layered architecture in which the Presentation Layer made all of its calls to the Business Logic Layer (BLL). In order to adhere to this design decision, we first need to create a BLL class for the new Typed DataSet before we can access product data from the Presentation Layer.

Create a new class file named `ProductsBLLWithSprocs.cs` in the `~/App_Code/BLL` folder and add to it the following code:

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using NorthwindWithSprocsTableAdapters;
[System.ComponentModel.DataObject]
public class ProductsBLLWithSprocs
{
    private ProductsTableAdapter _productsAdapter = null;
    protected ProductsTableAdapter Adapter
    {
        get
        {
            if (_productsAdapter == null)
                _productsAdapter = new ProductsTableAdapter();
            return _productsAdapter;
        }
    }
    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Select, true)]
    public NorthwindWithSprocs.ProductsDataTable GetProducts()
    {
        return Adapter.GetProducts();
    }
    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Select, false)]
    public NorthwindWithSprocs.ProductsDataTable GetProductByProductID(int productID)
    {
        return Adapter.GetProductByProductID(productID);
    }
    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Insert, true)]
    public bool AddProduct
```

```
    public bool AddProduct
        (string productName, int? supplierID, int? categoryID,
         string quantityPerUnit, decimal? unitPrice, short? unitsInStock,
         short? unitsOnOrder, short? reorderLevel, bool discontinued)
    {
        // Create a new ProductRow instance
        NorthwindWithSprocs.ProductsDataTable products =
            new NorthwindWithSprocs.ProductsDataTable();
        NorthwindWithSprocs.ProductsRow product = products.NewProductsRow();
        product.ProductName = productName;
        if (supplierID == null)
            product.SetSupplierIDNull();
        else
            product.SupplierID = supplierID.Value;
        if (categoryID == null)
            product.SetCategoryIDNull();
        else
            product.CategoryID = categoryID.Value;
        if (quantityPerUnit == null)
            product.SetQuantityPerUnitNull();
        else
            product.QuantityPerUnit = quantityPerUnit;
        if (unitPrice == null)
            product.SetUnitPriceNull();
        else
            product.UnitPrice = unitPrice.Value;
        if (unitsInStock == null)
            product.SetUnitsInStockNull();
        else
            product.UnitsInStock = unitsInStock.Value;
        if (unitsOnOrder == null)
            product.SetUnitsOnOrderNull();
        else
            product.UnitsOnOrder = unitsOnOrder.Value;
        if (reorderLevel == null)
            product.SetReorderLevelNull();
        else
            product.ReorderLevel = reorderLevel.Value;
        product.Discontinued = discontinued;
        // Add the new product
        products.AddProductsRow(product);
        int rowsAffected = Adapter.Update(products);
        // Return true if precisely one row was inserted, otherwise false
        return rowsAffected == 1;
    }
    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Update, true)]
    public bool UpdateProduct
        (string productName, int? supplierID, int? categoryID, string quantityPerUnit,
        decimal? unitPrice, short? unitsInStock, short? unitsOnOrder,
        short? reorderLevel, bool discontinued, int productID)
    {
        NorthwindWithSprocs.ProductsDataTable products =
            Adapter.GetProductByProductID(productID);
        if (products.Count == 0)
            // no matching record found, return false
            return false;
        NorthwindWithSprocs.ProductsRow product = products[0];
        product.ProductName = productName;
        if (supplierID == null)
            product.SetSupplierIDNull();
        else
            product.SupplierID = supplierID.Value;
        if (categoryID == null)
            product.SetCategoryIDNull();
        else
            product.CategoryID = categoryID.Value;
        if (quantityPerUnit == null)
            product.SetQuantityPerUnitNull();
```

```
        else
            product.QuantityPerUnit = quantityPerUnit;
        if (unitPrice == null)
            product.SetUnitPriceNull();
        else
            product.UnitPrice = unitPrice.Value;
        if (unitsInStock == null)
            product.SetUnitsInStockNull();
        else
            product.UnitsInStock = unitsInStock.Value;
        if (unitsOnOrder == null)
            product.SetUnitsOnOrderNull();
        else
            product.UnitsOnOrder = unitsOnOrder.Value;
        if (reorderLevel == null)
            product.SetReorderLevelNull();
        else
            product.ReorderLevel = reorderLevel.Value;
        product.Discontinued = discontinued;
        // Update the product record
        int rowsAffected = Adapter.Update(product);
        // Return true if precisely one row was updated, otherwise false
        return rowsAffected == 1;
    }
    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Delete, true)]
    public bool DeleteProduct(int productID)
    {
        int rowsAffected = Adapter.Delete(productID);
        // Return true if precisely one row was deleted, otherwise false
        return rowsAffected == 1;
    }
}
```

This class mimics the `ProductsBLL` class semantics from earlier tutorials, but uses the `ProductsTableAdapter` and `ProductsDataTable` objects from the `NorthwindWithSprocs` DataSet. For example, rather than having a `using NorthwindTableAdapters` statement at the start of the class file as `ProductsBLL` does, the `ProductsBLLWithSprocs` class uses `using NorthwindWithSprocsTableAdapters`. Likewise, the `ProductsDataTable` and `ProductsRow` objects used in this class are prefixed with the `NorthwindWithSprocs` namespace. The `ProductsBLLWithSprocs` class provides two data access methods, `GetProducts` and `GetProductByProductID`, and methods to add, update, and delete a single product instance.

Step 7: Working with the `NorthwindWithSprocs` DataSet from the Presentation Layer

At this point we have created a DAL that uses stored procedures to access and modify the underlying database data. We have also built a rudimentary BLL with methods to retrieve all products or a particular product along with methods for adding, updating, and deleting products. To round off this tutorial, let s create an ASP.NET page that uses the BLL s `ProductsBLLWithSprocs` class for displaying, updating, and deleting records.

Open the `NewSprocs.aspx` page in the `AdvancedDAL` folder and drag a GridView from the Toolbox onto the Designer, naming it `Products`. From the GridView s smart tag choose to bind it to a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to use the `ProductsBLLWithSprocs` class, as shown in Figure 22.

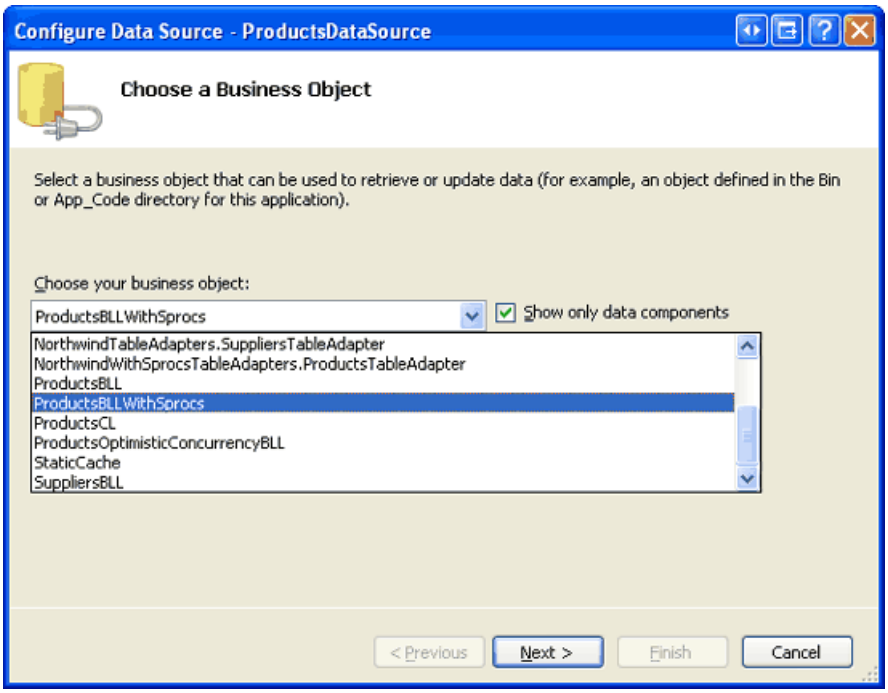


Figure 22: Configure the ObjectDataSource to Use the `ProductsBLLWithSprocs` Class (Click to view full-size image)

The drop-down list in the SELECT tab has two options, `GetProducts` and `GetProductByProductID`. Since we want to display all products in the GridView, choose the `GetProducts` method. The drop-down lists in the UPDATE, INSERT, and DELETE tabs each only have one method. Ensure that each of these drop-down lists has its appropriate method selected and then click Finish.

After the ObjectDataSource wizard has completed, Visual Studio will add BoundFields and a CheckBoxField to the GridView for the product data fields. Turn on the GridView s built-in editing and deleting features by checking the Enable Editing and Enable Deleting options present in the smart tag.

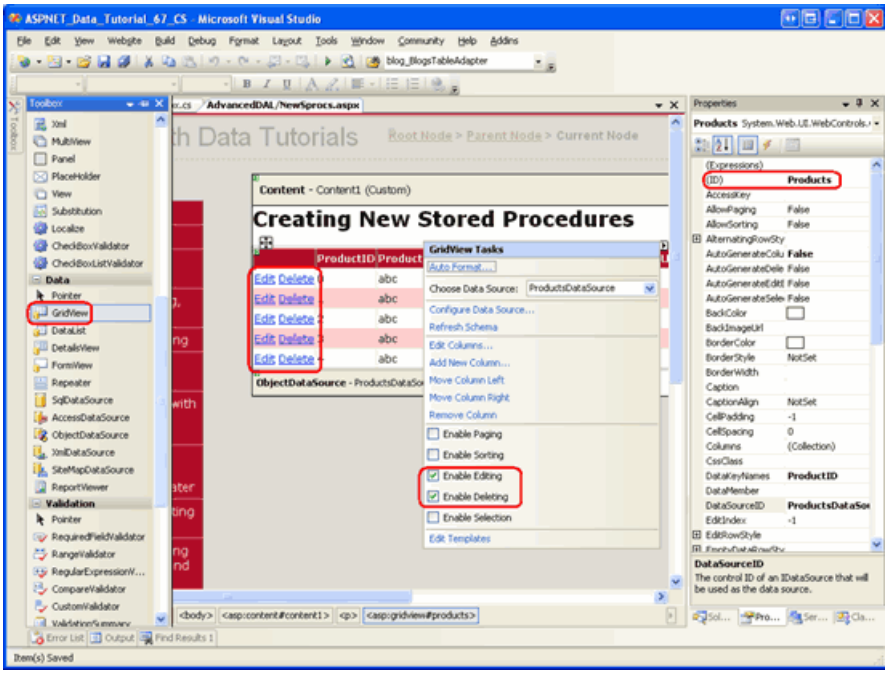


Figure 23: The Page Contains a GridView with Editing and Deleting Support Enabled (Click to view full-size image)

As we ve discussed in previous tutorials, at the completion of the ObjectDataSource s wizard, Visual Studio sets the `OldValuesParameterFormatString` property to original_{0}. This needs to be reverted to its default value of {0} in order for the data modification features to work properly given the parameters expected by the methods in

our BLL. Therefore, be sure to set the `OldValuesParameterFormatString` property to {0} or remove the property altogether from the declarative syntax.

After completing the Configure Data Source wizard, turning on editing and deleting support in the GridView, and returning the ObjectDataSource s `OldValuesParameterFormatString` property to its default value, your page s declarative markup should look similar to the following:

```
<asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource">
    <Columns>
        <asp:CommandField ShowDeleteButton="True" ShowEditButton="True" />
        <asp:BoundField DataField="ProductID" HeaderText="ProductID"
            InsertVisible="False" ReadOnly="True"
            SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName" HeaderText="ProductName"
            SortExpression="ProductName" />
        <asp:BoundField DataField="SupplierID" HeaderText="SupplierID"
            SortExpression="SupplierID" />
        <asp:BoundField DataField="CategoryID" HeaderText="CategoryID"
            SortExpression="CategoryID" />
        <asp:BoundField DataField="QuantityPerUnit" HeaderText="QuantityPerUnit"
            SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
            SortExpression="UnitPrice" />
        <asp:BoundField DataField="UnitsInStock" HeaderText="UnitsInStock"
            SortExpression="UnitsInStock" />
        <asp:BoundField DataField="UnitsOnOrder" HeaderText="UnitsOnOrder"
            SortExpression="UnitsOnOrder" />
        <asp:BoundField DataField="ReorderLevel" HeaderText="ReorderLevel"
            SortExpression="ReorderLevel" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    DeleteMethod="DeleteProduct" InsertMethod="AddProduct"
    SelectMethod="GetProducts" TypeName="ProductsBLLWithSprocs"
    UpdateMethod="UpdateProduct">
    <DeleteParameters>
        <asp:Parameter Name="productID" Type="Int32" />
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="unitsInStock" Type="Int16" />
        <asp:Parameter Name="unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="reorderLevel" Type="Int16" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
    <InsertParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="unitsInStock" Type="Int16" />
        <asp:Parameter Name="unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="reorderLevel" Type="Int16" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
    </InsertParameters>
</asp:ObjectDataSource>
```

At this point we could tidy up the GridView by customizing the editing interface to include validation, having the `CategoryID` and `SupplierID` columns render as DropDownLists, and so on. We could also add a client-side confirmation to the Delete button, and I encourage you to take the time to implement these enhancements. Since these topics have been covered in previous tutorials, however, we will not cover them again here.

Regardless of whether you enhance the GridView or not, test out the page s core features in a browser. As Figure 24 shows, the page lists the products in a GridView that provides per-row editing and deleting capabilities.

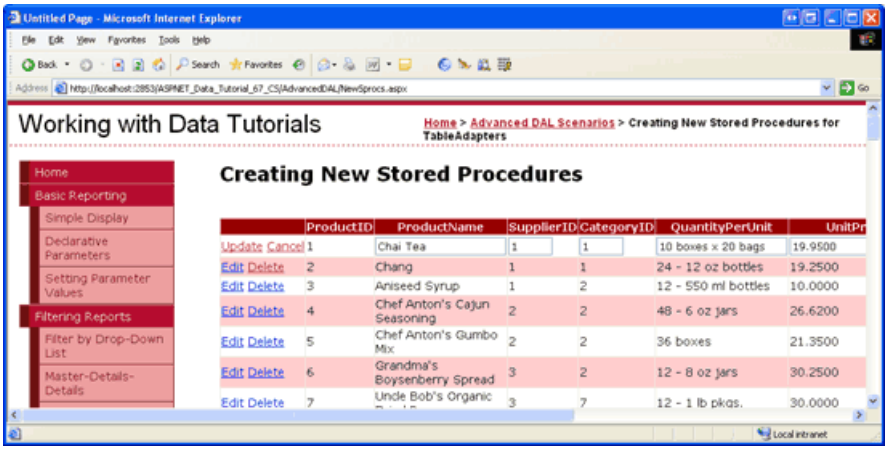


Figure 24: The Products Can Be Viewed, Edited, and Deleted from the GridView (Click to view full-size image)

Summary

The TableAdapters in a Typed DataSet can access data from the database using ad-hoc SQL statements or through stored procedures. When working with stored procedures, either existing stored procedures can be used or the TableAdapter wizard can be instructed to create new stored procedures based on a `SELECT` query. In this tutorial we explored how to have the stored procedures automatically created for us.

While having the stored procedures auto-generated helps save time, there are certain cases where the stored procedure created by the wizard doesn t align with what we would have created on our own. One example is the `Products_Update` stored procedure, which expected both `@Original_ProductID` and `@ProductID` input parameters even though the `@Original_ProductID` parameter was superfluous.

In many scenarios, the stored procedures may already have been created, or we may want to build them manually so as to have a finer degree of control over the stored procedure s commands. In either case, we would want to instruct the TableAdapter to use existing stored procedures for its methods. We shall see how to accomplish this in the next tutorial.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Creating and Maintaining Stored Procedures
- Retrieving Scalar Data from a Stored Procedure
- SQL Server Stored Procedure Basics
- Stored Procedures: An Overview
- Writing a Stored Procedure

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest

book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Geisenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Using Existing Stored Procedures for the Typed DataSet's TableAdapters (C#)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> In the previous tutorial we learned how to use the TableAdapter Wizard to generate new stored procedures. In this tutorial we learn how the same TableAdapter Wizard can work with existing stored procedures. We also learn how to manually add new stored procedures to our database.

Introduction

In the preceding tutorial we saw how the Typed DataSet s TableAdapters could be configured to use stored procedures to access data rather than ad-hoc SQL statements. In particular, we examined how to have the TableAdapter wizard automatically create these stored procedures. When porting a legacy application to ASP.NET 2.0 or when building an ASP.NET 2.0 website around an existing data model, chances are that the database already contains the stored procedures we need. Alternatively, you may prefer to create your stored procedures by hand or through some tool other than the TableAdapter wizard that auto-generates your stored procedures.

In this tutorial we will look at how to configure the TableAdapter to use existing stored procedures. Since the Northwind database only has a small set of built-in stored procedures, we will also look at the steps needed to manually add new stored procedures to the database through the Visual Studio environment. Let s get started!

> **NOTE**
>
> In the Wrapping Database Modifications within a Transaction tutorial we added methods to the TableAdapter to support transactions (`BeginTransaction` , `CommitTransaction` , and so on). Alternatively, transactions can be managed entirely within a stored procedure, which requires no modifications to the Data Access Layer code. In this tutorial we'll explore the T-SQL commands used to execute a stored procedure s statements within the scope of a transaction.

Step 1: Adding Stored Procedures to the Northwind Database

Visual Studio makes it easy to add new stored procedures to a database. Let s add a new stored procedure to the Northwind database that returns all columns from the `Products` table for those that have a particular `CategoryID` value. From the Server Explorer window, expand the Northwind database so that its folders - Database Diagrams, Tables, Views, and so forth - are displayed. As we saw in the preceding tutorial, the Stored Procedures folder contains the database s existing stored procedures. To add a new stored procedure, simply right-click the Stored Procedures folder and choose the Add New Stored Procedure option from the context menu.

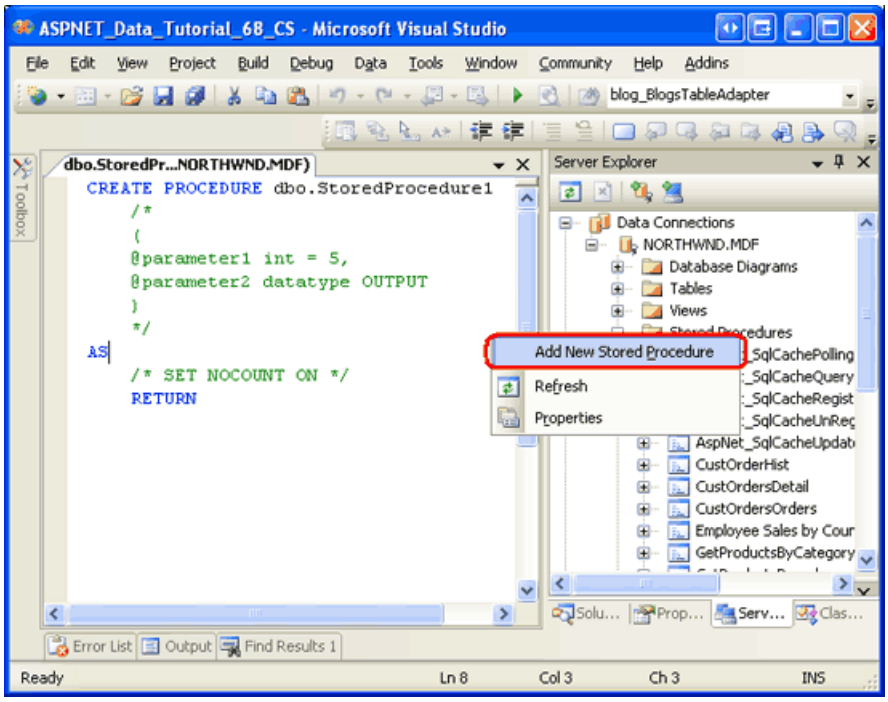


Figure 1: Right-Click the Stored Procedures Folder and Add a New Stored Procedure (Click to view full-size image)

As Figure 1 shows, selecting the Add New Stored Procedure option opens a script window in Visual Studio with the outline of the SQL script needed to create the stored procedure. It is our job to flesh out this script and execute it, at which point the stored procedure will be added to the database.

Enter the following script:

```
CREATE PROCEDURE dbo.Products_SelectByCategoryID
(
    @CategoryID int
)
AS
SELECT ProductID, ProductName, SupplierID, CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued
FROM Products
WHERE CategoryID = @CategoryID
```

This script, when executed, will add a new stored procedure to the Northwind database named `Products_SelectByCategoryID`. This stored procedure accepts a single input parameter (`@CategoryID`, of type `int`) and it returns all of the fields for those products with a matching `CategoryID` value.

To execute this `CREATE PROCEDURE` script and add the stored procedure to the database, click the Save icon in the toolbar or hit Ctrl+S. After doing so, the Stored Procedures folder refreshes, showing the newly created stored procedure. Also, the script in the window will change subtlety from `CREATE PROCEDURE dbo.Products_SelectProductByCategoryID` to `ALTER PROCEDURE dbo.Products_SelectProductByCategoryID`. `CREATE PROCEDURE` adds a new stored procedure to the database, while `ALTER PROCEDURE` updates an existing one. Since the start of the script has changed to `ALTER PROCEDURE`, changing the stored procedures input parameters or SQL statements and clicking the Save icon will update the stored procedure with these changes.

Figure 2 shows Visual Studio after the `Products_SelectByCategoryID` stored procedure has been saved.

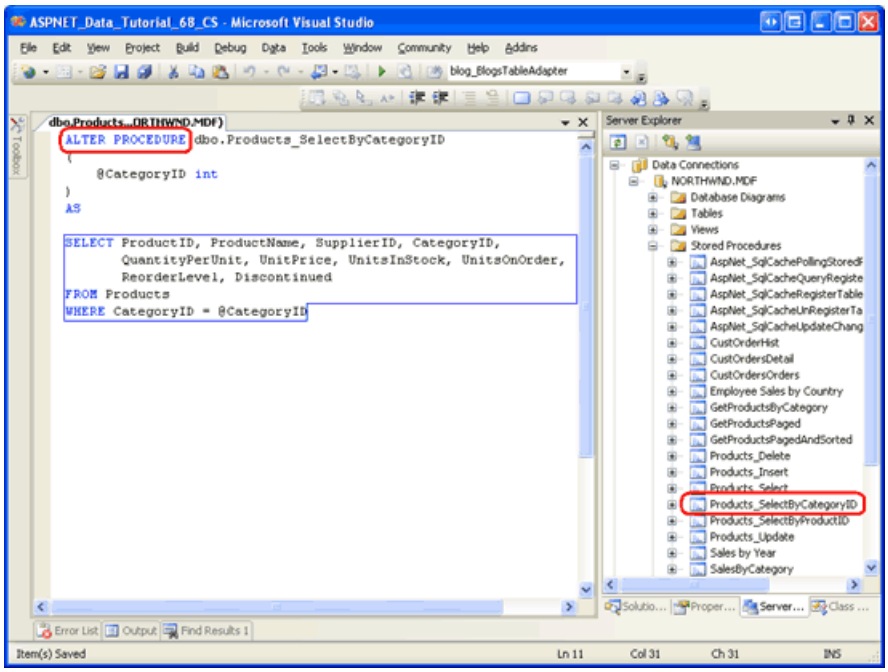


Figure 2: The Stored Procedure `Products_SelectByCategoryID` Has Been Added to the Database (Click to view full-size image)

Step 2: Configuring the TableAdapter to Use an Existing Stored Procedure

Now that the `Products_SelectByCategoryID` stored procedure has been added to the database, we can configure our Data Access Layer to use this stored procedure when one of its methods is invoked. In particular, we will add a `GetProductsByCategoryID(categoryID)` method to the `ProductsTableAdapter` in the `NorthwindWithSprocs` Typed DataSet that calls the `Products_SelectByCategoryID` stored procedure we just created.

Start by opening the `NorthwindWithSprocs` DataSet. Right-click on the `ProductsTableAdapter` and choose Add Query to launch the TableAdapter Query Configuration wizard. In the preceding tutorial we opted to have the TableAdapter create a new stored procedure for us. For this tutorial, however, we want to wire the new TableAdapter method to the existing `Products_SelectByCategoryID` stored procedure. Therefore, choose the Use existing stored procedure option from the wizard s first step and then click Next.

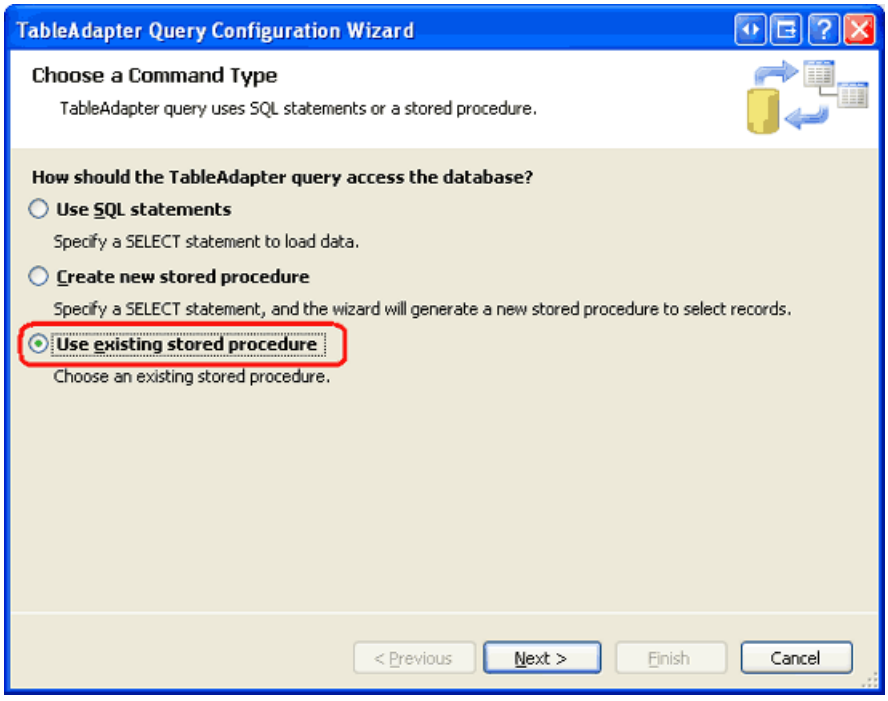


Figure 3: Choose the Use existing stored procedure Option (Click to view full-size image)

The following screen provides a drop-down list populated with the database s stored procedures. Selecting a stored procedure lists its input parameters on the left and the data fields returned (if any) on the right. Choose the `Products_SelectByCategoryID` stored procedure from the list and click Next.

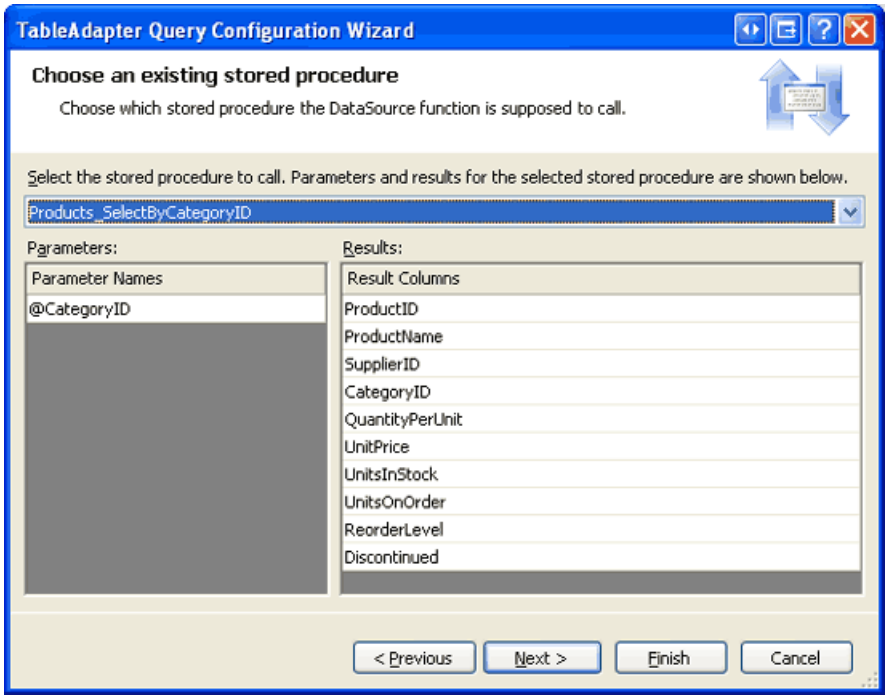


Figure 4: Pick the `Products_SelectByCategoryID` Stored Procedure (Click to view full-size image)

The next screen asks us what kind of data is returned by the stored procedure and our answer here determines the type returned by the TableAdapter s method. For example, if we indicate that tabular data is returned, the method will return a `ProductsDataTable` instance populated with the records returned by the stored procedure. In contrast, if we indicate that this stored procedure returns a single value the TableAdapter will return an `object` that is assigned the value in the first column of the first record returned by the stored procedure.

Since the `Products_SelectByCategoryID` stored procedure returns all products that belong to a particular category, choose the first answer - Tabular data - and click Next.

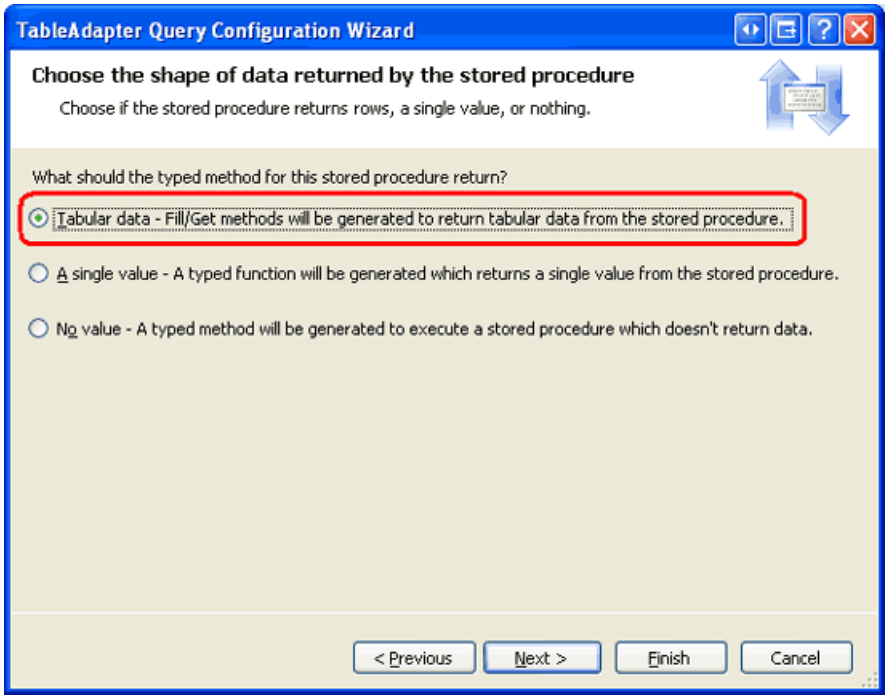


Figure 5: Indicate that the Stored Procedure Returns Tabular Data (Click to view full-size image)

All that remains is to indicate what method patterns to use followed by the names for these methods. Leave both the Fill a DataTable and Return a DataTable options checked, but rename the methods to `FillByCategoryID` and `GetProductsByCategoryID`. Then click Next to review a summary of the tasks the wizard will perform. If everything looks correct, click Finish.

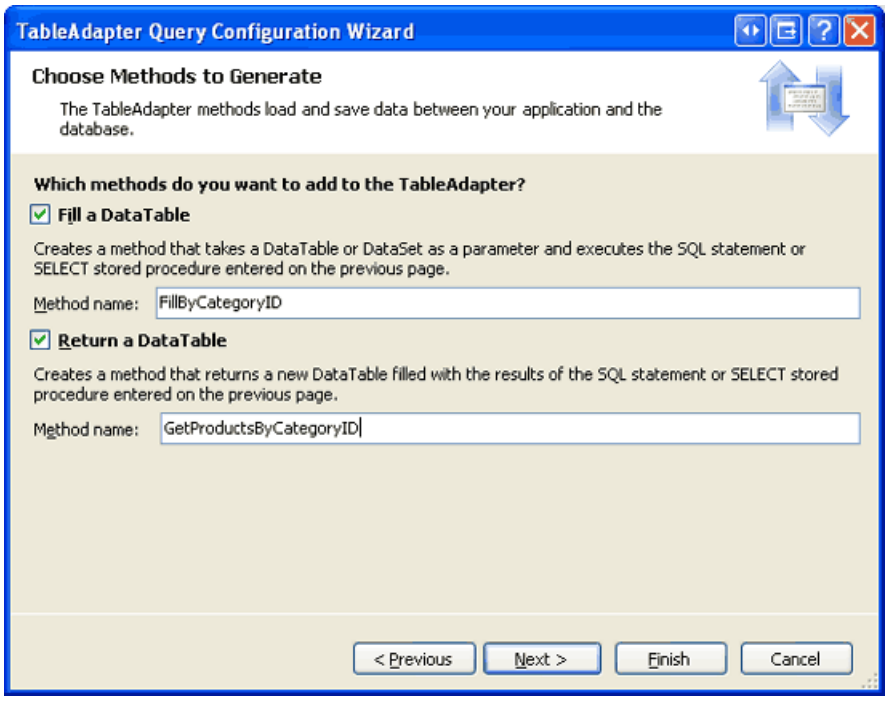


Figure 6: Name the Methods `FillByCategoryID` and `GetProductsByCategoryID` (Click to view full-size image)

> **NOTE**
>
> The TableAdapter methods we just created, `FillByCategoryID` and `GetProductsByCategoryID`, expect an input parameter of type `int`. This input parameter value is passed into the stored procedure via its `@CategoryID` parameter. If you modify the `Products_SelectByCategory` stored procedure s parameters, you'll need to also update the parameters for these TableAdapter methods. As discussed in the previous tutorial, this can be done in one of two ways: by manually adding or removing parameters from the parameters collection or by rerunning the TableAdapter wizard.

Step 3: Adding a `GetProductsByCategoryID(categoryID)` Method to the BLL

With the `GetProductsByCategoryID` DAL method complete, the next step is to provide access to this method in the Business Logic Layer. Open the `ProductsBLLWithSprocs` class file and add the following method:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Select, false)]
public NorthwindWithSprocs.ProductsDataTable GetProductByCategoryID(int categoryID)
{
    return Adapter.GetProductsByCategoryID(categoryID);
}
```

This BLL method simply returns the `ProductsDataTable` returned from the `ProductsTableAdapter` s `GetProductsByCategoryID` method. The `DataObjectMethodAttribute` attribute provides metadata used by the ObjectDataSource s Configure Data Source wizard. In particular, this method will appear in the SELECT tab s drop-down list.

Step 4: Displaying Products by Category

To test the newly added `Products_SelectByCategoryID` stored procedure and the corresponding DAL and BLL

methods, let s create an ASP.NET page that contains a DropDownList and a GridView. The DropDownList will list all of the categories in the database while the GridView will display the products belonging to the selected category.

> **NOTE**
>
> We ve created master/detail interfaces using DropDownLists in previous tutorials. For a more in-depth look at implementing such a master/detail report, refer to the Master/Detail Filtering With a DropDownList tutorial.

Open the `ExistingSprocs.aspx` page in the `AdvancedDAL` folder and drag a DropDownList from the Toolbox onto the Designer. Set the DropDownList s `ID` property to `Categories` and its `AutoPostBack` property to `true`. Next, from its smart tag, bind the DropDownList to a new ObjectDataSource named `CategoriesDataSource`. Configure the ObjectDataSource so that it retrieves its data from the `CategoriesBLL` class s `GetCategories` method. Set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) .

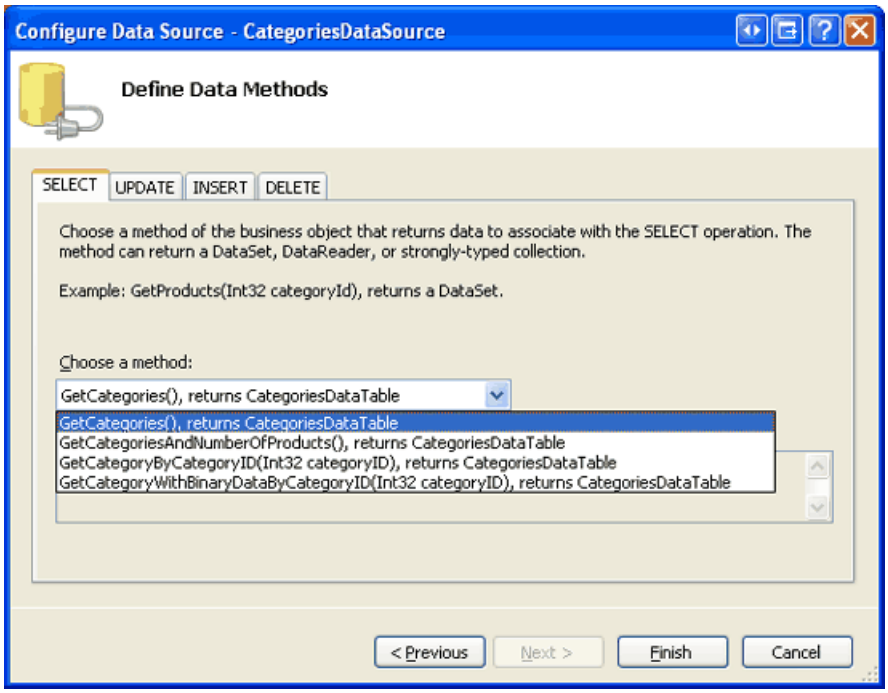


Figure 7: Retrieve Data from the `CategoriesBLL` Class s `GetCategories` Method (Click to view full-size image)

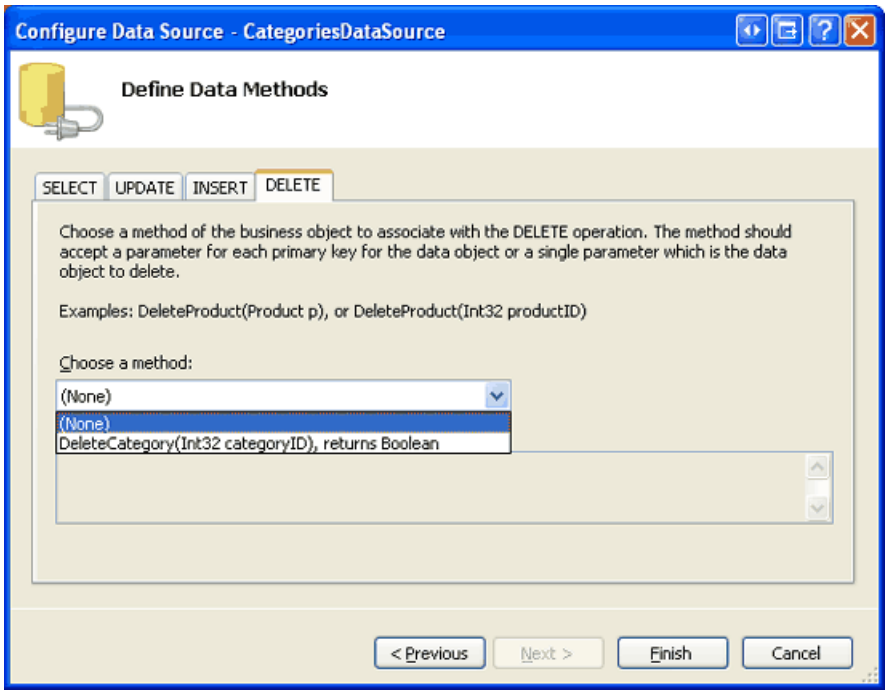


Figure 8: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE Tabs to (None) (Click to view full-size image)

After completing the ObjectDataSource wizard, configure the DropDownList to display the `CategoryName` data field and to use the `CategoryID` field as the `Value` for each `ListItem`.

At this point, the DropDownList and ObjectDataSource s declarative markup should similar to the following:

```
<asp:DropDownList ID="Categories" runat="server" AutoPostBack="True"
    DataSourceID="CategoriesDataSource" DataTextField="CategoryName"
    DataValueField="CategoryID">
</asp:DropDownList>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

Next, drag a GridView onto the Designer, placing it beneath the DropDownList. Set the GridView s `ID` to `ProductsByCategory` and, from its smart tag, bind it to a new ObjectDataSource named `ProductsByCategoryDataSource`. Configure the `ProductsByCategoryDataSource` ObjectDataSource to use the `ProductsBLLWithSprocs` class, having it retrieve its data using the `GetProductsByCategoryID(categoryID)` method. Since this GridView will only be used to display data, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) and click Next.

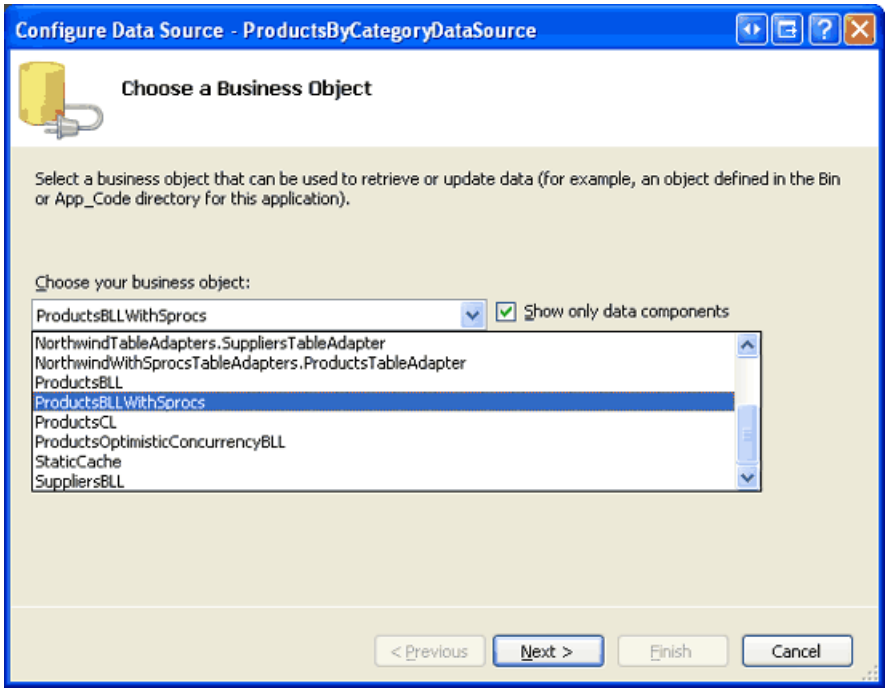


Figure 9: Configure the ObjectDataSource to Use the `ProductsBLLWithSprocs` Class (Click to view full-size image)

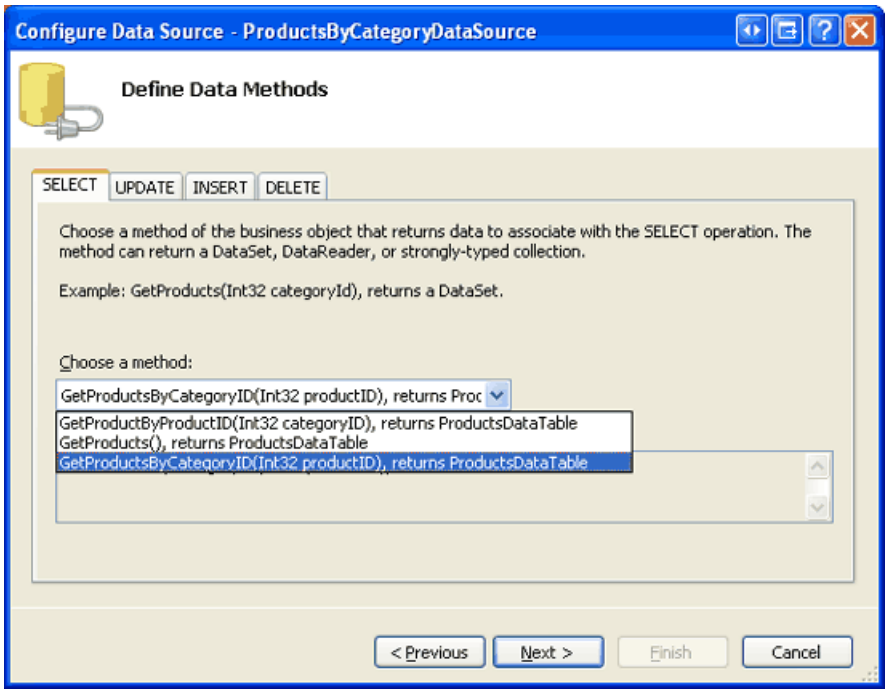


Figure 10: Retrieve Data from the `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

The method chosen in the SELECT tab expects a parameter, so the final step of the wizard prompts us for the parameter s source. Set the Parameter source drop-down list to Control and choose the `Categories` control from the ControlID drop-down list. Click Finish to complete the wizard.

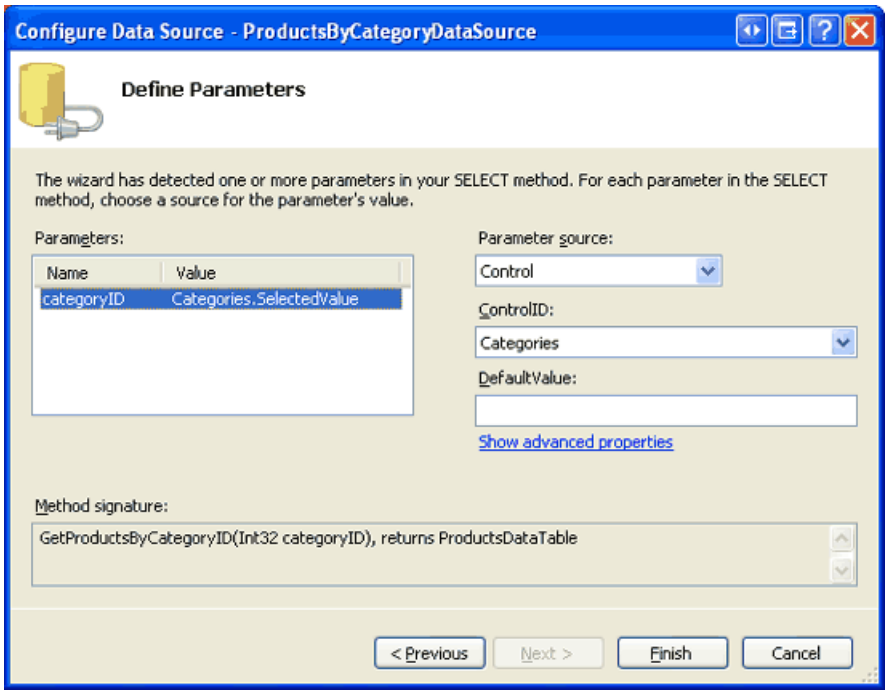


Figure 11: Use the `Categories` DropDownList as the Source of the `categoryID` Parameter (Click to view full-size image)

Upon completing the ObjectDataSource wizard, Visual Studio will add BoundFields and a CheckBoxField for each of the product data fields. Feel free to customize these fields as you see fit.

Visit the page through a browser. When visiting the page the Beverages category is selected and the corresponding products listed in the grid. Changing the drop-down list to an alternative category, as Figure 12 shows, causes a postback and reloads the grid with the products of the newly selected category.

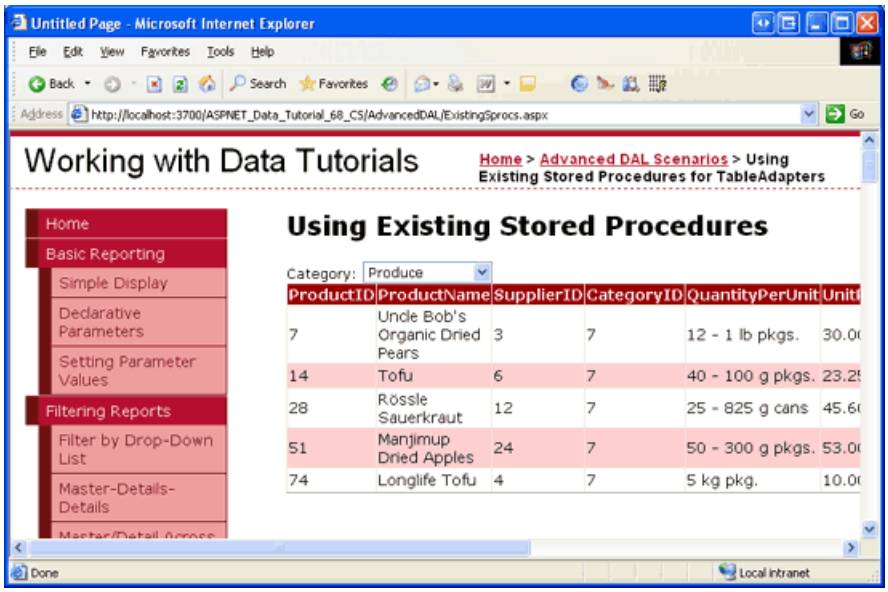


Figure 12: The Products in the Produce Category are Displayed (Click to view full-size image)

Step 5: Wrapping a Stored Procedure s Statements Within the Scope of a Transaction

In the Wrapping Database Modifications within a Transaction tutorial we discussed techniques for performing a series of database modification statements within the scope of a transaction. Recall that the modifications performed under the umbrella of a transaction either all succeed or all fail, guaranteeing atomicity. Techniques for using transactions include:

- Using the classes in the `System.Transactions` namespace,
- Having the Data Access Layer use ADO.NET classes like `SqlTransaction`, and
- Adding the T-SQL transaction commands directly within the stored procedure

The *Wrapping Database Modifications within a Transaction* tutorial used the ADO.NET classes in the DAL. The remainder of this tutorial examines how to manage a transaction using T-SQL commands from within a stored procedure.

The three key SQL commands for manually starting, committing, and rolling back a transaction are `BEGIN TRANSACTION`, `COMMIT TRANSACTION`, and `ROLLBACK TRANSACTION`, respectively. Like with the ADO.NET approach, when using transactions from within a stored procedure we need to apply the following pattern:

1. Indicate the start of a transaction.
2. Execute the SQL statements that comprise the transaction.
3. If there is an error in any one of the statements from Step 2, rollback the transaction
4. If all of the statements from Step 2 complete without error, commit the transaction.

This pattern can be implemented in T-SQL syntax using the following template:

```
BEGIN TRY
  BEGIN TRANSACTION -- Start the transaction
  ... Perform the SQL statements that makeup the transaction ...
  -- If we reach here, success!
  COMMIT TRANSACTION
END TRY
BEGIN CATCH
  -- Whoops, there was an error
  ROLLBACK TRANSACTION
  -- Raise an error with the
  -- details of the exception
  DECLARE @ErrMsg nvarchar(4000),
          @ErrSeverity int
  SELECT @ErrMsg = ERROR_MESSAGE(),
         @ErrSeverity = ERROR_SEVERITY()

  RAISERROR(@ErrMsg, @ErrSeverity, 1)
END CATCH
```

The template starts by defining a `TRY...CATCH` block, a construct new to SQL Server 2005. Like with `try...catch` blocks in C#, the SQL `TRY...CATCH` block executes the statements in the `TRY` block. If any statement raises an error, control is immediately transferred to the `CATCH` block.

If there are no errors executing the SQL statements that makeup the transaction, the `COMMIT TRANSACTION` statement commits the changes and completes the transaction. If, however, one of the statements results in an error, the `ROLLBACK TRANSACTION` in the `CATCH` block returns the database to its state prior to the start of the transaction. The stored procedure also raises an error using the RAISERROR command, which causes a `SqlException` to be raised in the application.

> **NOTE**
> Since the `TRY...CATCH` block is new to SQL Server 2005, the above template will not work if you are using older versions of Microsoft SQL Server. If you are not using SQL Server 2005, consult Managing Transactions in SQL Server Stored Procedures for a template that will work with other versions of SQL Server.

Let s look at a concrete example. A foreign key constraint exists between the `Categories` and `Products` tables, meaning that each `CategoryID` field in the `Products` table must map to a `CategoryID` value in the `Categories` table. Any action that would violate this constraint, such as attempting to delete a category that has associated products, results in a foreign key constraint violation. To verify this, revisit the Updating and Deleting Existing Binary Data example in the Working with Binary Data section (`~/BinaryData/UpdatingAndDeleting.aspx`). This page lists each category in the system along with Edit and Delete buttons (see Figure 13), but if you attempt to delete a category that has associated products - such as Beverages - the delete fails due to a foreign key constraint violation (see Figure 14).

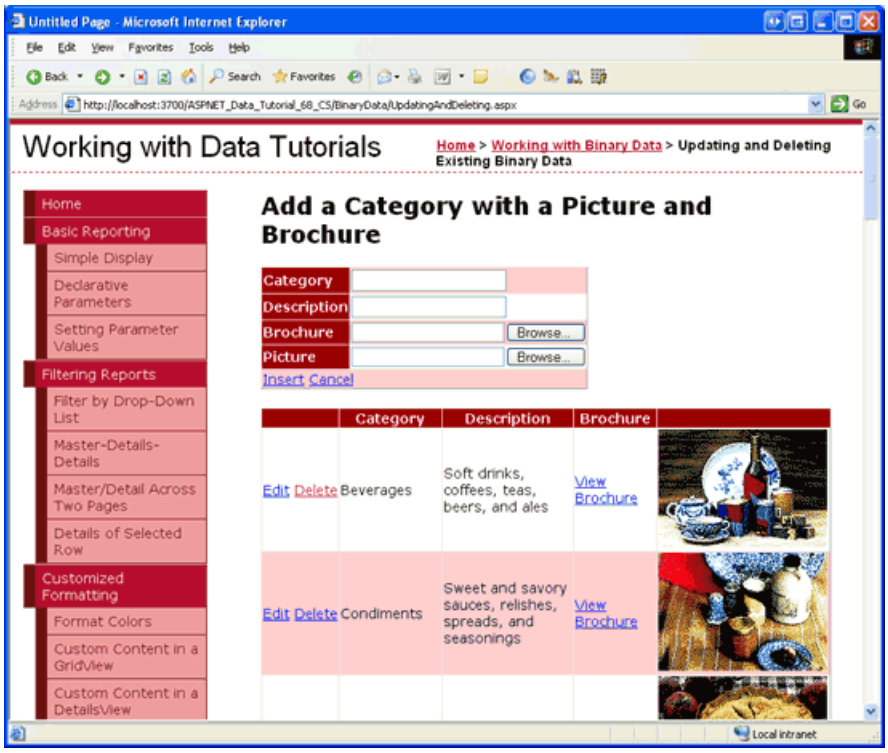


Figure 13: Each Category is Displayed in a GridView with Edit and Delete Buttons (Click to view full-size image)

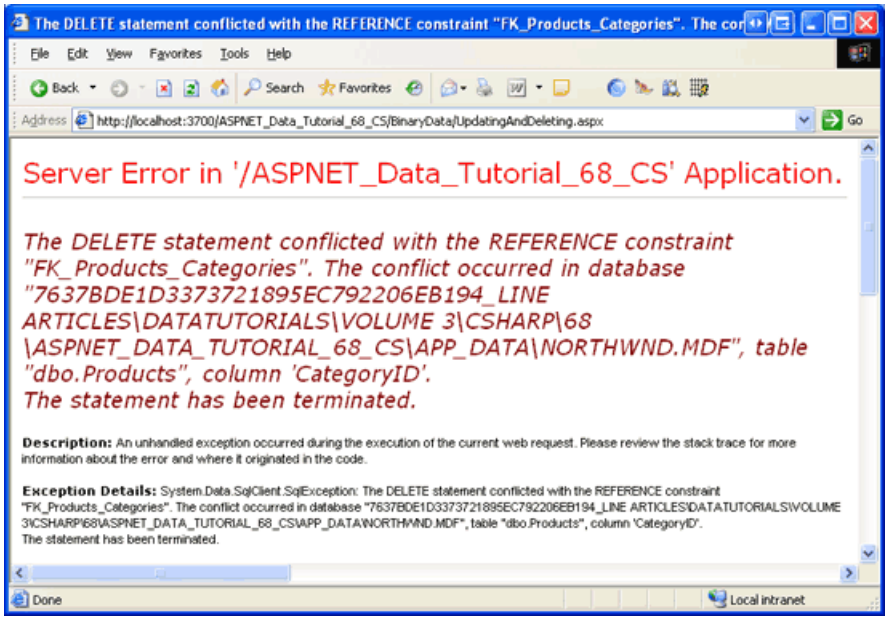


Figure 14: You Cannot Delete a Category that has Existing Products (Click to view full-size image)

Imagine, though, that we want to allow categories to be deleted regardless of whether they have associated products. Should a category with products be deleted, imagine that we want to also delete its existing products (although another option would be to simply set its products `CategoryID` values to `NULL`). This functionality could be implemented through the cascade rules of the foreign key constraint. Alternatively, we could create a stored procedure that accepts a `@CategoryID` input parameter and, when invoked, explicitly deletes all of the associated products and then the specified category.

Our first attempt at such a stored procedure might look like the following:

```
CREATE PROCEDURE dbo.Categories_Delete
(
    @CategoryID int
)
AS
-- First, delete the associated products...
DELETE FROM Products
WHERE CategoryID = @CategoryID
-- Now delete the category
DELETE FROM Categories
WHERE CategoryID = @CategoryID
```

While this will definitely delete the associated products and category, it does not do so under the umbrella of a transaction. Imagine that there is some other foreign key constraint on `Categories` that would prohibit the deletion of a particular `@CategoryID` value. The problem is that in such a case all of the products will be deleted before we attempt to delete the category. The net result is that for such a category, this stored procedure would remove all of its products while the category remained since it still has related records in some other table.

If the stored procedure were wrapped within the scope of a transaction, however, the deletes to the `Products` table would be rolled back in the face of a failed delete on `Categories`. The following stored procedure script uses a transaction to assure atomicity between the two `DELETE` statements:

```
CREATE PROCEDURE dbo.Categories_Delete
(
    @CategoryID int
)
AS
BEGIN TRY
  BEGIN TRANSACTION -- Start the transaction
  -- First, delete the associated products...
  DELETE FROM Products
  WHERE CategoryID = @CategoryID
  -- Now delete the category
  DELETE FROM Categories
  WHERE CategoryID = @CategoryID
  -- If we reach here, success!
  COMMIT TRANSACTION
END TRY
BEGIN CATCH
  -- Whoops, there was an error
  ROLLBACK TRANSACTION
  -- Raise an error with the
  -- details of the exception
  DECLARE @ErrMsg nvarchar(4000),
          @ErrSeverity int
  SELECT @ErrMsg = ERROR_MESSAGE(),
         @ErrSeverity = ERROR_SEVERITY()

  RAISERROR(@ErrMsg, @ErrSeverity, 1)
END CATCH
```

Take a moment to add the `Categories_Delete` stored procedure to the Northwind database. Refer back to Step 1 for instructions on adding stored procedures to a database.

Step 6: Updating the `CategoriesTableAdapter`

While we ve added the `Categories_Delete` stored procedure to the database, the DAL is currently configured to use ad-hoc SQL statements to perform the delete. We need to update the `CategoriesTableAdapter` and instruct it to use the `Categories_Delete` stored procedure instead.

> **NOTE**
>
> Earlier in this tutorial we were working with the `NorthwindWithSprocs` DataSet. But that DataSet only has a single entity, `ProductsDataTable`, and we need to work with categories. Therefore, for the remainder of this tutorial when I talk about the Data Access Layer I m referring to the `Northwind` DataSet, the one that we first created in the Creating a Data Access Layer tutorial.

Open the Northwind DataSet, select the `CategoriesTableAdapter`, and go to the Properties window. The Properties window lists the `InsertCommand`, `UpdateCommand`, `DeleteCommand`, and `SelectCommand` used by the TableAdapter, as well as its name and connection information. Expand the `DeleteCommand` property to see its details. As Figure 15 shows, the `DeleteCommand` s `CommandType` property is set to Text, which instructs it to send the text in the `CommandText` property as an ad-hoc SQL query.

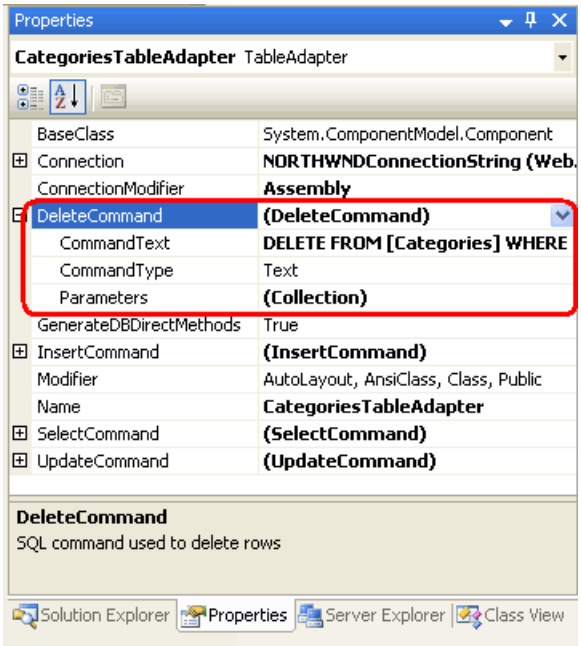


Figure 15: Select the `CategoriesTableAdapter` in the Designer to View Its Properties in the Properties Window

To change these settings, select the (DeleteCommand) text in the Properties window and choose (New) from the drop-down list. This will clear out the settings for the `CommandText`, `CommandType`, and `Parameters` properties. Next, set the `CommandType` property to `StoredProcedure` and then type in the name of the stored procedure for the `CommandText` (`dbo.Categories_Delete`). If you make sure to enter the properties in this order - first the `CommandType` and then the `CommandText` - Visual Studio will automatically populate the Parameters collection. If you do not enter these properties in this order, you will have to manually add the parameters through the Parameters Collection Editor. In either case, it s prudent to click on the ellipses in the Parameters property to bring up the Parameters Collection Editor to verify that the correct parameter settings changes have been made (see Figure 16). If you do not see any parameters in the dialog box, add the `@CategoryID` parameter manually (you do not need to add the `@RETURN_VALUE` parameter).

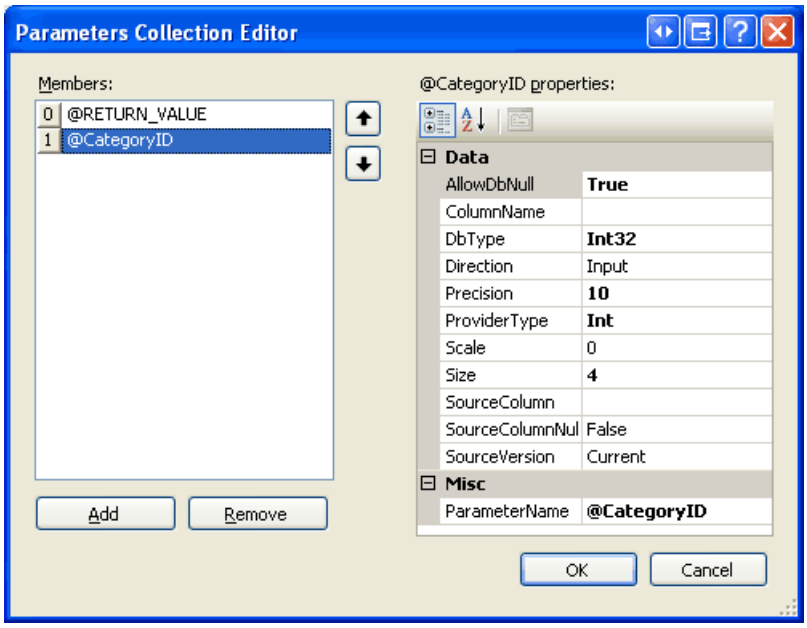


Figure 16: Ensure That the Parameters Settings are Correct

Once the DAL has been updated, deleting a category will automatically delete all of its associated products and do so under the umbrella of a transaction. To verify this, return to the Updating and Deleting Existing Binary Data page and click the Delete button for one of the categories. With one single click of the mouse, the category and all of its associated products will be deleted.

> **NOTE**
>
> Before testing the `Categories_Delete` stored procedure, which will delete a number of products along with the selected category, it might be prudent to make a backup copy of your database. If you are using the `NORTHWND.MDF` database in `App_Data`, simply close Visual Studio and copy the MDF and LDF files in `App_Data` to some other folder. After testing the functionality, you can restore the database by closing Visual Studio and replacing the current MDF and LDF files in `App_Data` with the backup copies.

Summary

While the TableAdapter s wizard will automatically generate stored procedures for us, there are times when we might already have such stored procedures created or want to create them manually or with other tools instead. To accommodate such scenarios, the TableAdapter can also be configured to point to an existing stored procedure. In this tutorial we looked at how to manually add stored procedures to a database through the Visual Studio environment and how to wire the TableAdapter s methods to these stored procedures. We also examined the T-SQL commands and script pattern used for starting, committing, and rolling back transactions from within a stored procedure.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Hilton

Geisenow, S ren Jacob Lauritsen, and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Updating the TableAdapter to Use JOINs (C#)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

When working with a database it is common to request data that is spread across multiple tables. To retrieve data from two different tables we can use either a correlated subquery or a JOIN operation. In this tutorial we compare correlated subqueries and the JOIN syntax before looking at how to create a TableAdapter that includes a JOIN in its main query.

Introduction

With relational databases the data we are interested in working with is often spread across multiple tables. For example, when displaying product information we likely want to list each product s corresponding category and supplier s names. The `Products` table has `CategoryID` and `SupplierID` values, but the actual category and supplier names are in the `Categories` and `Suppliers` tables, respectively.

To retrieve information from another, related table, we can either use *correlated subqueries* or `JOIN` *s*. A correlated subquery is a nested `SELECT` query that references columns in the outer query. For example, in the Creating a Data Access Layer tutorial we used two correlated subqueries in the `ProductsTableAdapter` s main query to return the category and supplier names for each product. A `JOIN` is a SQL construct that merges related rows from two different tables. We used a `JOIN` in the Querying Data with the SqlDataSource Control tutorial to display category information alongside each product.

The reason we have abstained from using `JOIN` s with the TableAdapters is because of limitations in the TableAdapter s wizard to auto-generate corresponding `INSERT`, `UPDATE`, and `DELETE` statements. More specifically, if the TableAdapter s main query contains any `JOIN` s, the TableAdapter cannot auto-create the ad-hoc SQL statements or stored procedures for its `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties.

In this tutorial we will briefly compare and contrast correlated subqueries and `JOIN` s before exploring how to create a TableAdapter that includes `JOIN` s in its main query.

Comparing and Contrasting Correlated Subqueries and `JOIN` s

Recall that the `ProductsTableAdapter` created in the first tutorial in the `Northwind` DataSet uses correlated subqueries to bring back each product s corresponding category and supplier name. The `ProductsTableAdapter` s main query is shown below.

```
SELECT ProductID, ProductName, SupplierID, CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued,
       (SELECT CategoryName FROM Categories WHERE Categories.CategoryID =
            Products.CategoryID) as CategoryName,
       (SELECT CompanyName FROM Suppliers WHERE Suppliers.SupplierID =
            Products.SupplierID) as SupplierName
FROM Products
```

The two correlated subqueries -

`(SELECT CategoryName FROM Categories WHERE Categories.CategoryID = Products.CategoryID)` and

`(SELECT CompanyName FROM Suppliers WHERE Suppliers.SupplierID = Products.SupplierID)` - are `SELECT` queries that return a single value per product as an additional column in the outer `SELECT` statement s column list.

Alternatively, a `JOIN` can be used to return each product s supplier and category name. The following query returns the same output as the above one, but uses `JOIN` s in place of subqueries:

```
SELECT ProductID, ProductName, Products.SupplierID, Products.CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued,
       Categories.CategoryName,
       Suppliers.CompanyName as SupplierName
FROM Products
    LEFT JOIN Categories ON
        Categories.CategoryID = Products.CategoryID
    LEFT JOIN Suppliers ON
        Suppliers.SupplierID = Products.SupplierID
```

A `JOIN` merges the records from one table with records from another table based on some criteria. In the above query, for example, the `LEFT JOIN Categories ON Categories.CategoryID = Products.CategoryID` instructs SQL Server to merge each product record with the category record whose `CategoryID` value matches the product s `CategoryID` value. The merged result allows us to work with the corresponding category fields for each product (such as `CategoryName`).

NOTE

`JOIN` s are commonly used when querying data from relational databases. If you are new to the `JOIN` syntax or need to brush up a bit on its usage, I d recommend the SQL Join tutorial at W3 Schools. Also worth reading are the `JOIN` Fundamentals and Subquery Fundamentals sections of the SQL Books Online.

Since `JOIN` s and correlated subqueries can both be used to retrieve related data from other tables, many developers are left scratching their heads and wondering which approach to use. All of the SQL gurus I ve talked to have said roughly the same thing, that it doesn t really matter performance-wise as SQL Server will produce roughly identical execution plans. Their advice, then, is to use the technique that you and your team are most comfortable with. It merits noting that after imparting this advice these experts immediately express their preference of `JOIN` s over correlated subqueries.

When building a Data Access Layer using Typed DataSets, the tools work better when using subqueries. In particular, the TableAdapter s wizard will not auto-generate corresponding `INSERT`, `UPDATE`, and `DELETE` statements if the main query contains any `JOIN` s, but will auto-generate these statements when correlated subqueries are used.

To explore this shortcoming, create a temporary Typed DataSet in the `~/App_Code/DAL` folder. During the TableAdapter Configuration wizard, choose to use ad-hoc SQL statements and enter the following `SELECT` query (see Figure 1):

```
SELECT ProductID, ProductName, Products.SupplierID, Products.CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued,
       Categories.CategoryName,
       Suppliers.CompanyName as SupplierName
FROM Products
    LEFT JOIN Categories ON
        Categories.CategoryID = Products.CategoryID
    LEFT JOIN Suppliers ON
        Suppliers.SupplierID = Products.SupplierID
```

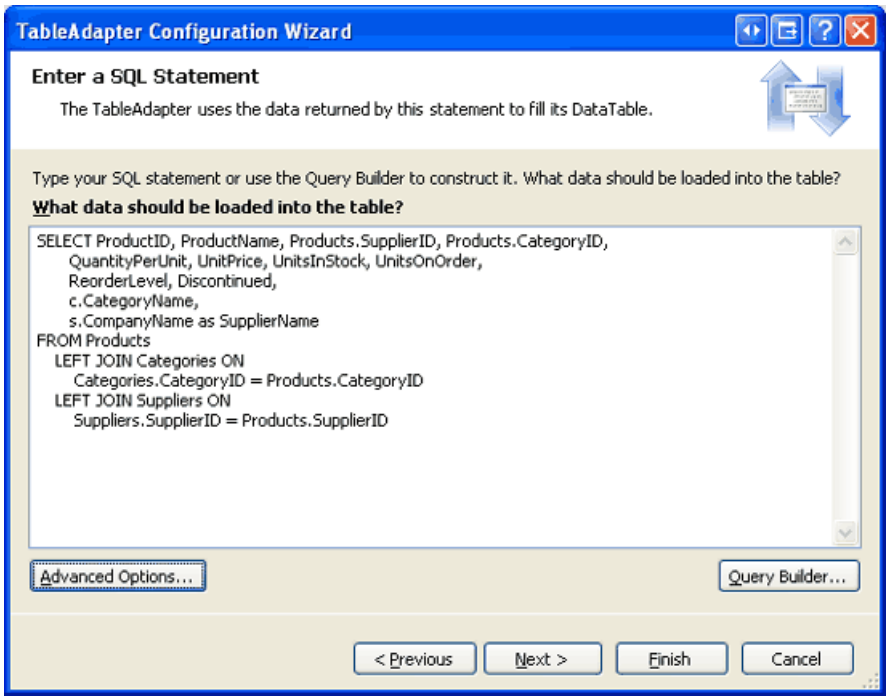


Figure 1: Enter a Main Query that Contains `JOIN` s (Click to view full-size image)

By default, the TableAdapter will automatically create `INSERT`, `UPDATE`, and `DELETE` statements based on the main query. If you click the Advanced button you can see that this feature is enabled. Despite this setting, the TableAdapter will not be able to create the `INSERT`, `UPDATE`, and `DELETE` statements because the main query contains a `JOIN`.

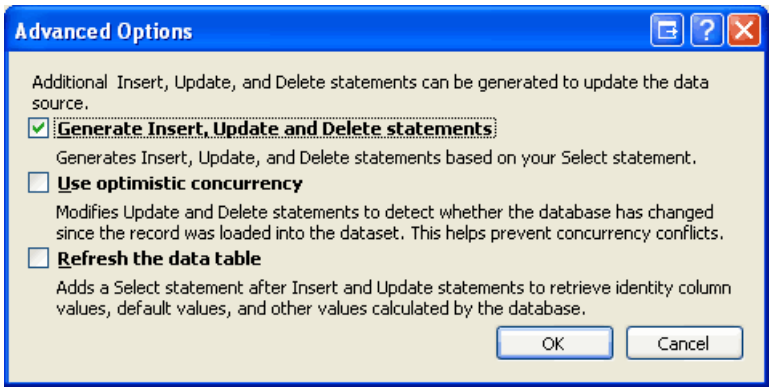


Figure 2: Enter a Main Query that Contains `JOIN` s

Click Finish to complete the wizard. At this point your DataSet s Designer will include a single TableAdapter with a DataTable with columns for each of the fields returned in the `SELECT` query s column list. This includes the `CategoryName` and `SupplierName`, as Figure 3 shows.

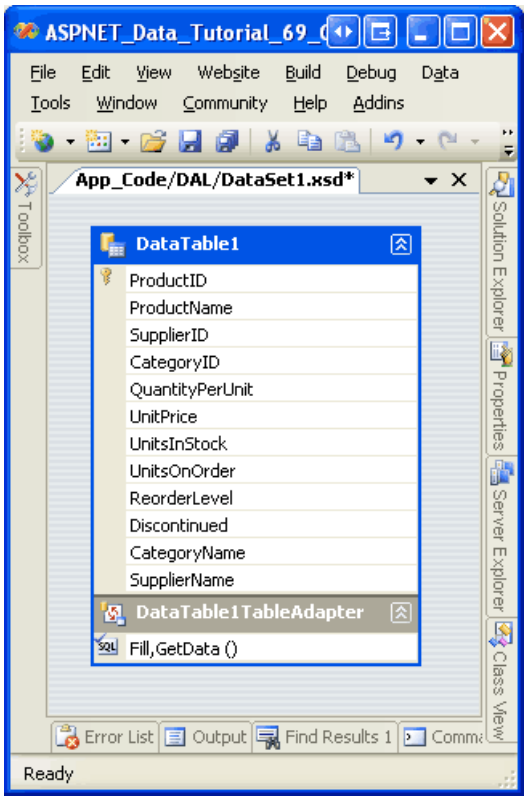


Figure 3: The DataTable Includes a Column for Each Field Returned in the Column List

While the DataTable has the appropriate columns, the TableAdapter lacks values for its `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties. To confirm this, click on the TableAdapter in the Designer and then go to the Properties window. There you will see that the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties are set to (None) .

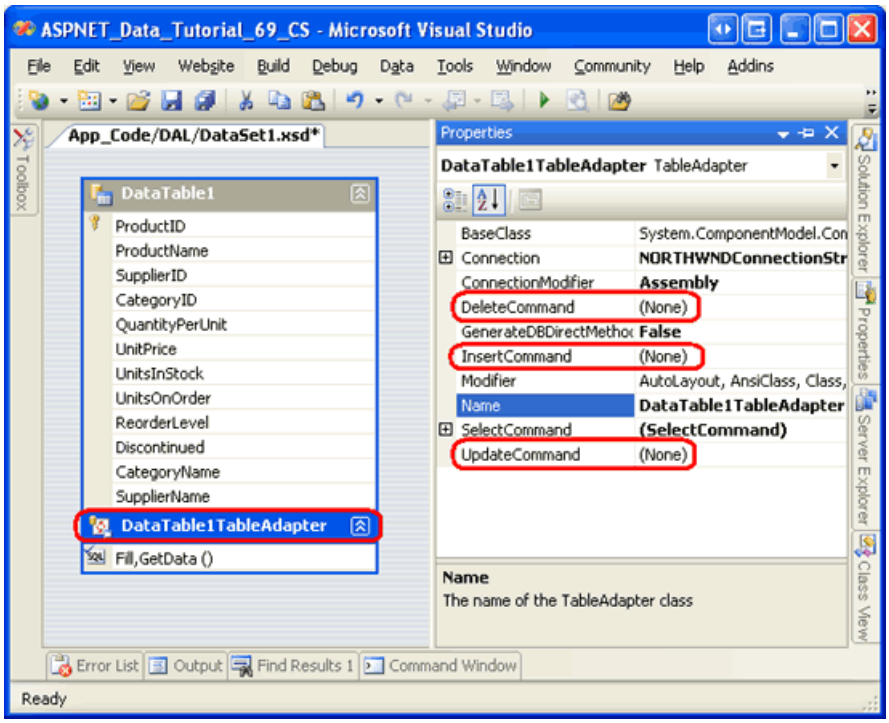


Figure 4: The `InsertCommand`, `UpdateCommand`, and `DeleteCommand` Properties are Set to (None) (Click to view full-size image)

To work around this shortcoming, we can manually provide the SQL statements and parameters for the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties via the Properties window. Alternatively, we could start by configuring the TableAdapter s main query to *not* include any `JOIN` s. This will allow the `INSERT`, `UPDATE`, and `DELETE` statements to be auto-generated for us. After completing the wizard, we could then manually

update the TableAdapter s `SelectCommand` from the Properties window so that it includes the `JOIN` syntax.

While this approach works, it is very brittle when using ad-hoc SQL queries because any time the TableAdapter s main query is re-configured through the wizard, the auto-generated `INSERT`, `UPDATE`, and `DELETE` statements are recreated. That means all of the customizations we later made would be lost if we right-clicked on the TableAdapter, chose Configure from the context menu, and completed the wizard again.

The brittleness of the TableAdapter s auto-generated `INSERT`, `UPDATE`, and `DELETE` statements is, fortunately, limited to ad-hoc SQL statements. If your TableAdapter uses stored procedures, you can customize the `SelectCommand`, `InsertCommand`, `UpdateCommand`, or `DeleteCommand` stored procedures and re-run the TableAdapter Configuration wizard without having to fear that the stored procedures will be modified.

Over the next several steps we will create a TableAdapter that, initially, uses a main query that omits any `JOIN` s so that the corresponding insert, update, and delete stored procedures will be auto-generated. We will then update the `SelectCommand` so that uses a `JOIN` that returns additional columns from related tables. Finally, we'll create a corresponding Business Logic Layer class and demonstrate using the TableAdapter in an ASP.NET web page.

Step 1: Creating the TableAdapter Using a Simplified Main Query

For this tutorial we will add a TableAdapter and strongly-typed DataTable for the `Employees` table in the `NorthwindWithSprocs` DataSet. The `Employees` table contains a `ReportsTo` field that specified the `EmployeeID` of the employee s manager. For example, employee Anne Dodsworth has a `ReportTo` value of 5, which is the `EmployeeID` of Steven Buchanan. Consequently, Anne reports to Steven, her manager. Along with reporting each employee s `ReportsTo` value, we might also want to retrieve the name of their manager. This can be accomplished using a `JOIN`. But using a `JOIN` when initially creating the TableAdapter precludes the wizard from automatically generating the corresponding insert, update, and delete capabilities. Therefore, we will start by creating a TableAdapter whose main query does not contain any `JOIN` s. Then, in Step 2, we will update the main query stored procedure to retrieve the manager s name via a `JOIN`.

Start by opening the `NorthwindWithSprocs` DataSet in the `~/App_Code/DAL` folder. Right-click on the Designer, select the Add option from the context menu, and pick the TableAdapter menu item. This will launch the TableAdapter Configuration wizard. As Figure 5 depicts, have the wizard create new stored procedures and click Next. For a refresher on creating new stored procedures from the TableAdapter s wizard, consult the Creating New Stored Procedures for the Typed DataSet s TableAdapters tutorial.

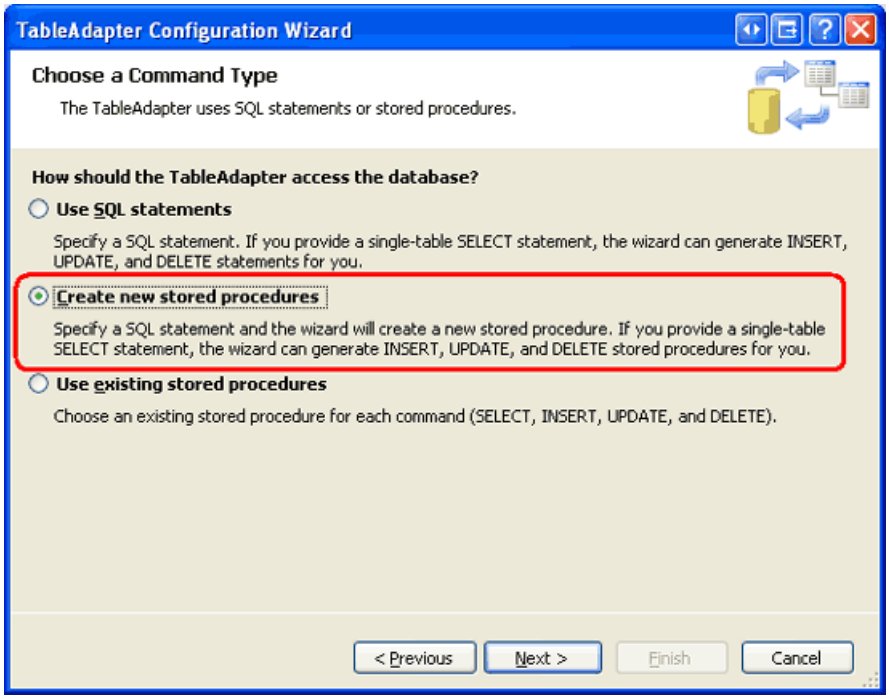


Figure 5: Select the Create new stored procedures Option (Click to view full-size image)

Use the following `SELECT` statement for the TableAdapter s main query:

```
SELECT EmployeeID, LastName, FirstName, Title, HireDate, ReportsTo, Country
FROM Employees
```

Since this query does not include any `JOIN` s, the TableAdapter wizard will automatically create stored procedures with corresponding `INSERT`, `UPDATE`, and `DELETE` statements, as well as a stored procedure for executing the main query.

The following step allows us to name the TableAdapter s stored procedures. Use the names `Employees_Select`, `Employees_Insert`, `Employees_Update`, and `Employees_Delete`, as shown in Figure 6.

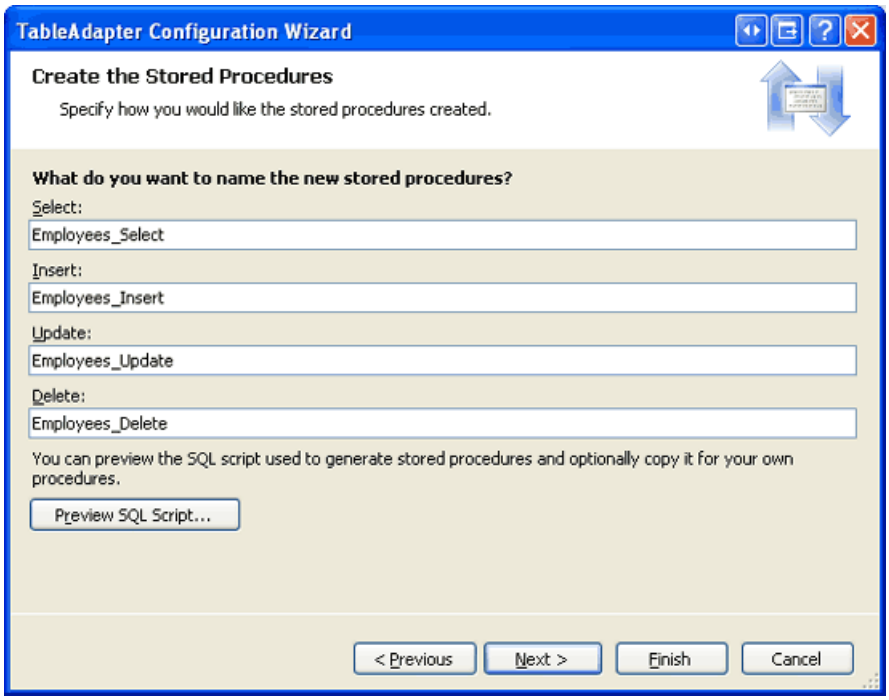


Figure 6: Name the TableAdapter s Stored Procedures (Click to view full-size image)

The final step prompts us to name the TableAdapter s methods. Use `Fill` and `GetEmployees` as the method names. Also be sure to leave the Create methods to send updates directly to the database (GenerateDBDirectMethods) checkbox checked.

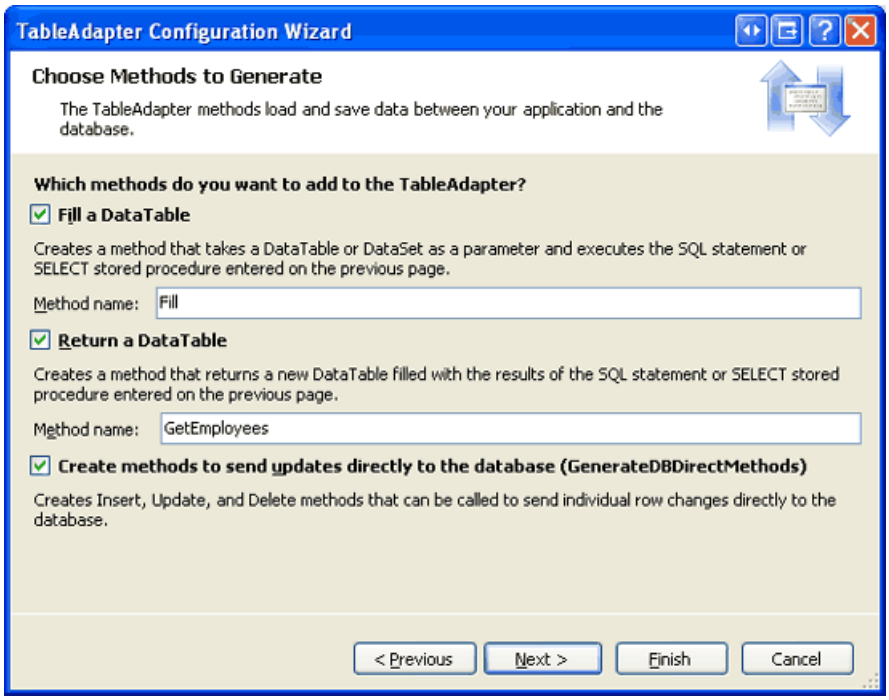


Figure 7: Name the TableAdapter s Methods `Fill` and `GetEmployees` (Click to view full-size image)

After completing the wizard, take a moment to examine the stored procedures in the database. You should see four new ones: `Employees_Select`, `Employees_Insert`, `Employees_Update`, and `Employees_Delete`. Next, inspect the `EmployeesDataTable` and `EmployeesTableAdapter` just created. The DataTable contains a column for each field returned by the main query. Click on the TableAdapter and then go to the Properties window. There you will see that the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties are correctly configured to call the corresponding stored procedures.

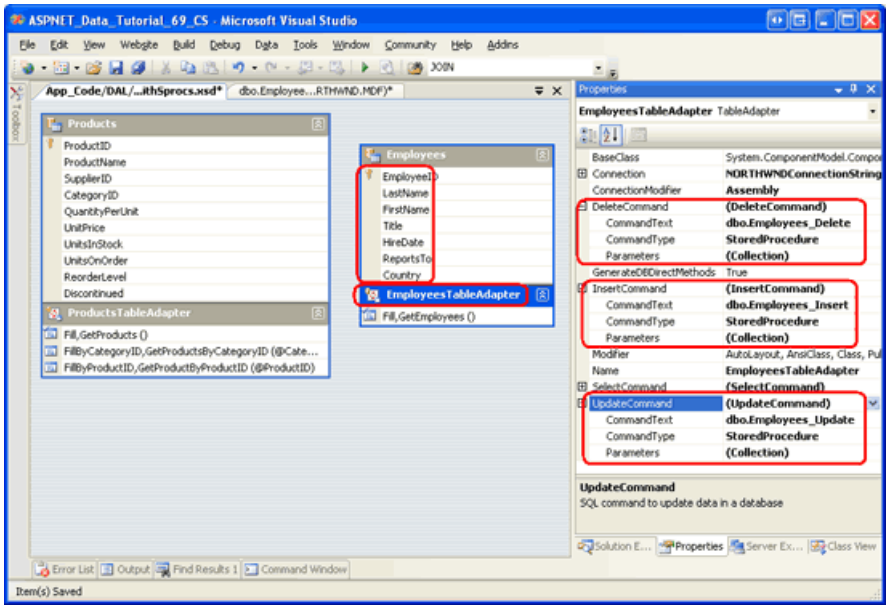


Figure 8: The TableAdapter Includes Insert, Update, and Delete Capabilities (Click to view full-size image)

With the insert, update, and delete stored procedures automatically created and the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties correctly configured, we are ready to customize the `SelectCommand` s stored procedure to return additional information about each employee s manager. Specifically, we need to update the `Employees_Select` stored procedure to use a `JOIN` and return the manager s `FirstName` and `LastName` values. After the stored procedure has been updated, we will need to update the DataTable so that it includes these additional columns. We'll tackle these two tasks in Steps 2 and 3.

Step 2: Customizing the Stored Procedure to Include a `JOIN`

Start by going to the Server Explorer, drilling down into the Northwind database s Stored Procedures folder, and opening the `Employees_Select` stored procedure. If you do not see this stored procedure, right-click on the Stored Procedures folder and choose Refresh. Update the stored procedure so that it uses a `LEFT JOIN` to return the manager s first and last name:

```
SELECT Employees.EmployeeID, Employees.LastName,
       Employees.FirstName, Employees.Title,
       Employees.HireDate, Employees.ReportsTo,
       Employees.Country,
       Manager.FirstName as ManagerFirstName,
       Manager.LastName as ManagerLastName
FROM Employees
    LEFT JOIN Employees AS Manager ON
        Employees.ReportsTo = Manager.EmployeeID
```

After updating the `SELECT` statement, save the changes by going to the File menu and choosing Save `Employees_Select`. Alternatively, you can click the Save icon in the toolbar or hit Ctrl+S. After saving your changes, right-click on the `Employees_Select` stored procedure in the Server Explorer and choose Execute. This will run the stored procedure and show its results in the Output window (see Figure 9).

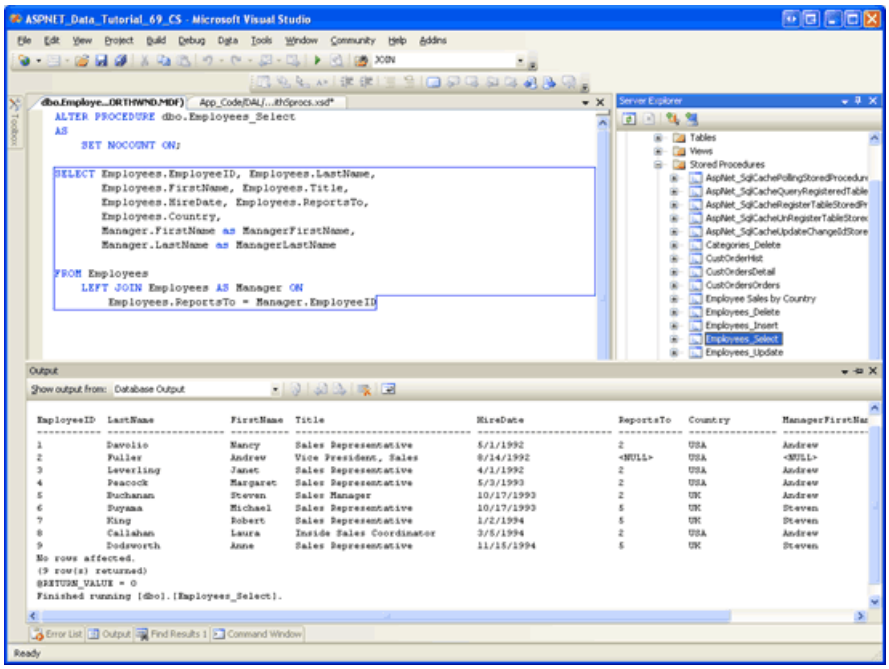


Figure 9: The Stored Procedures Results are Displayed in the Output Window (Click to view full-size image)

Step 3: Updating the DataTable s Columns

At this point, the `Employees_Select` stored procedure returns `ManagerFirstName` and `ManagerLastName` values, but the `EmployeesDataTable` is missing these columns. These missing columns can be added to the DataTable in one of two ways:

- **Manually** - right-click on the DataTable in the DataSet Designer and, from the Add menu, choose Column. You can then name the column and set its properties accordingly.
- **Automatically** - the TableAdapter Configuration wizard will update the DataTable s columns to reflect the fields returned by the `SelectCommand` stored procedure. When using ad-hoc SQL statements, the wizard will also remove the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties since the `SelectCommand` now contains a `JOIN`. But when using stored procedures, these command properties remain intact.

We have explored manually adding DataTable columns in previous tutorials, including Master/Detail Using a Bulleted List of Master Records with a Details DataList and Uploading Files, and we will look at this process again in more detail in our next tutorial. For this tutorial, however, let s use the automatic approach via the TableAdapter Configuration wizard.

Start by right-clicking on the `EmployeesTableAdapter` and selecting Configure from the context menu. This brings up the TableAdapter Configuration wizard, which lists the stored procedures used for selecting, inserting, updating, and deleting, along with their return values and parameters (if any). Figure 10 shows this wizard. Here we can see that the `Employees_Select` stored procedure now returns the `ManagerFirstName` and `ManagerLastName` fields.

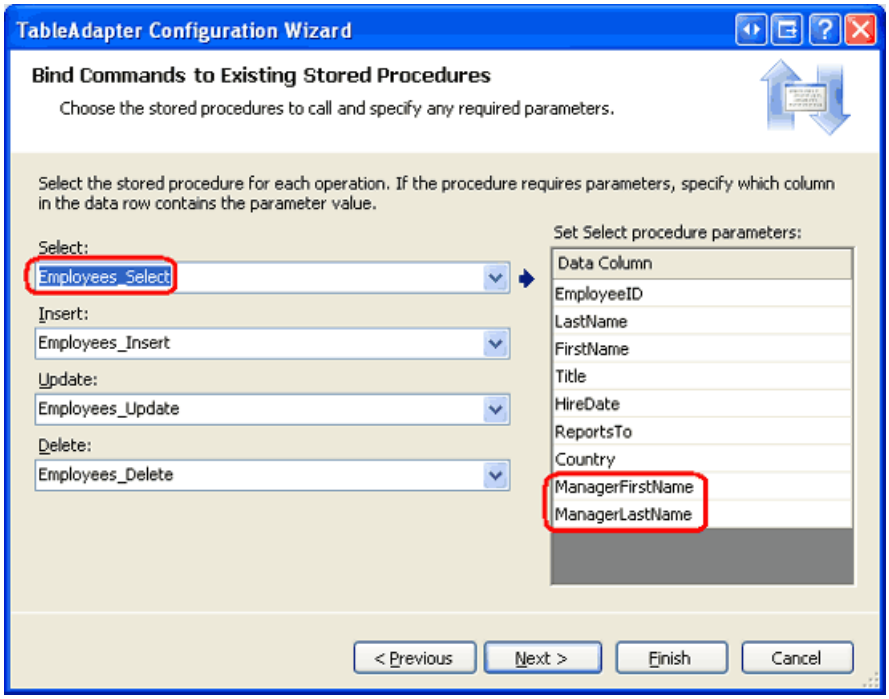


Figure 10: The Wizard Shows the Updated Column List for the `Employees_Select` Stored Procedure (Click to view full-size image)

Complete the wizard by clicking Finish. Upon returning to the DataSet Designer, the `EmployeesDataTable` includes two additional columns: `ManagerFirstName` and `ManagerLastName`.

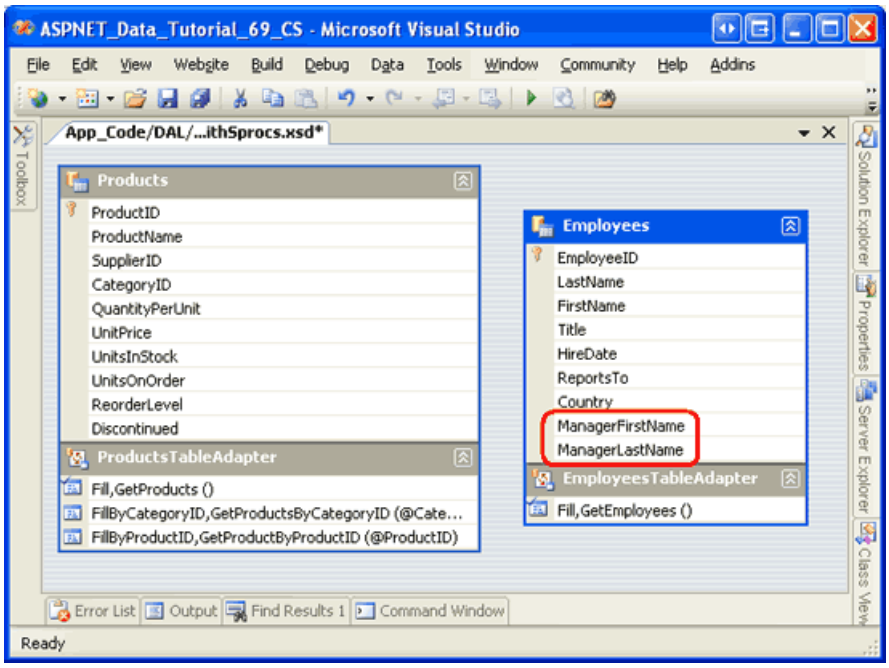


Figure 11: The `EmployeesDataTable` Contains Two New Columns (Click to view full-size image)

To illustrate that the updated `Employees_Select` stored procedure is in effect and that the insert, update, and delete capabilities of the TableAdapter are still functional, let s create a web page that allows users to view and delete employees. Before we create such a page, however, we need to first create a new class in the Business Logic Layer for working with employees from the `NorthwindWithSprocs` DataSet. In Step 4, we will create an `EmployeesBLLWithSprocs` class. In Step 5, we will use this class from an ASP.NET page.

Step 4: Implementing the Business Logic Layer

Create a new class file in the `~/App_Code/BLL` folder named `EmployeesBLLWithSprocs.cs`. This class mimics the semantics of the existing `EmployeesBLL` class, only this new one provides fewer methods and uses the `NorthwindWithSprocs` DataSet (instead of the `Northwind` DataSet). Add the following code to the `EmployeesBLLWithSprocs` class.

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using NorthwindWithSprocsTableAdapters;
[System.ComponentModel.DataObject]
public class EmployeesBLLWithSprocs
{
    private EmployeesTableAdapter _employeesAdapter = null;
    protected EmployeesTableAdapter Adapter
    {
        get
        {
            if (_employeesAdapter == null)
                _employeesAdapter = new EmployeesTableAdapter();
            return _employeesAdapter;
        }
    }
    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Select, true)]
    public NorthwindWithSprocs.EmployeesDataTable GetEmployees()
    {
        return Adapter.GetEmployees();
    }
    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Delete, true)]
    public bool DeleteEmployee(int employeeID)
    {
        int rowsAffected = Adapter.Delete(employeeID);
        // Return true if precisely one row was deleted, otherwise false
        return rowsAffected == 1;
    }
}
```

The `EmployeesBLLWithSprocs` class s `Adapter` property returns an instance of the `NorthwindWithSprocs` DataSet s `EmployeesTableAdapter`. This is used by the class s `GetEmployees` and `DeleteEmployee` methods. The `GetEmployees` method calls the `EmployeesTableAdapter` s corresponding `GetEmployees` method, which invokes the `Employees_Select` stored procedure and populates its results in an `EmployeeDataTable`. The `DeleteEmployee` method similarly calls the `EmployeesTableAdapter` s `Delete` method, which invokes the `Employees_Delete` stored procedure.

Step 5: Working with the Data in the Presentation Layer

With the `EmployeesBLLWithSprocs` class complete, we re ready to work with employee data through an ASP.NET page. Open the `JOINs.aspx` page in the `AdvancedDAL` folder and drag a GridView from the Toolbox onto the Designer, setting its `ID` property to `Employees`. Next, from the GridView s smart tag, bind the grid to a new ObjectDataSource control named `EmployeesDataSource`.

Configure the ObjectDataSource to use the `EmployeesBLLWithSprocs` class and, from the SELECT and DELETE tabs, ensure that the `GetEmployees` and `DeleteEmployee` methods are selected from the drop-down lists. Click Finish to complete the ObjectDataSource s configuration.

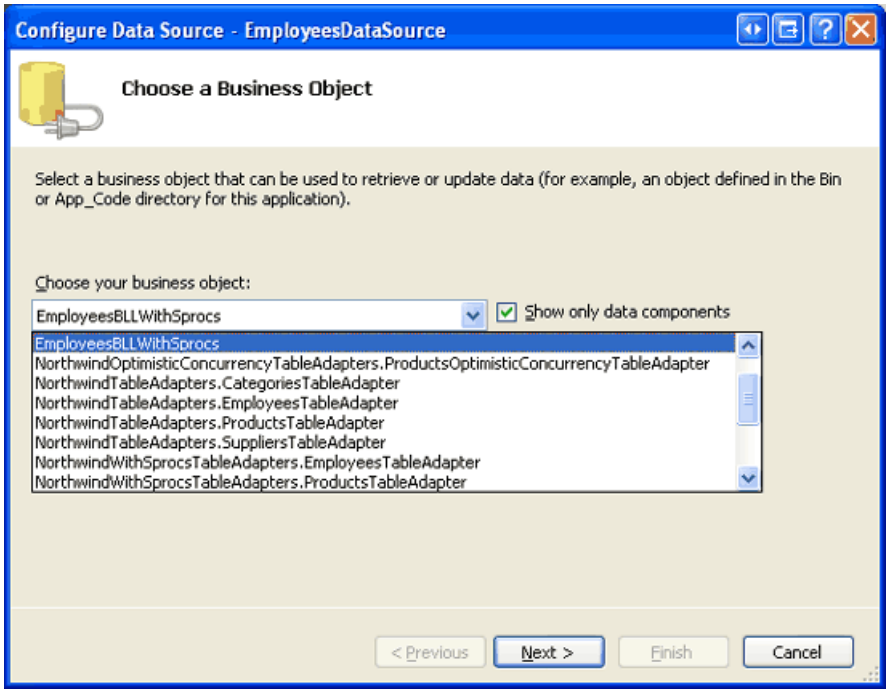


Figure 12: Configure the ObjectDataSource to Use the `EmployeesBLLWithSprocs` Class (Click to view full-size image)

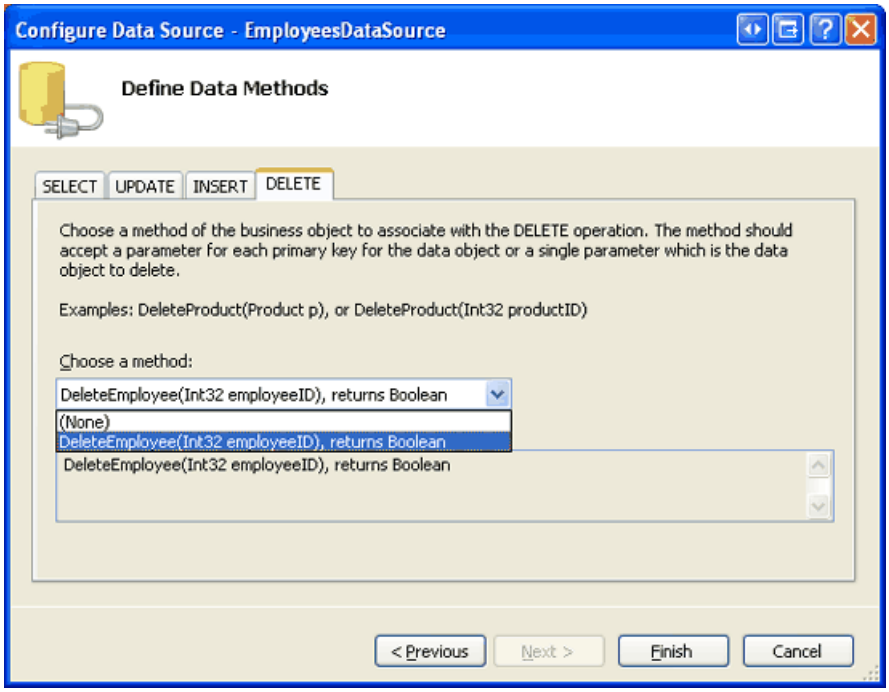


Figure 13: Have the ObjectDataSource Use the `GetEmployees` and `DeleteEmployee` Methods (Click to view full-size image)

Visual Studio will add a BoundField to the GridView for each of the `EmployeesDataTable` s columns. Remove all of these BoundFields except for `Title`, `LastName`, `FirstName`, `ManagerFirstName`, and `ManagerLastName` and rename

the `HeaderText` properties for the last four BoundFields to Last Name, First Name, Manager s First Name, and Manager s Last Name, respectively.

To allow users to delete employees from this page we need to do two things. First, instruct the GridView to provide deleting capabilities by checking the Enable Deleting option from its smart tag. Second, change the ObjectDataSource s `OldValuesParameterFormatString` property from the value set by the ObjectDataSource wizard (`original_{0}`) to its default value (`{0}`). After making these changes, your GridView and ObjectDataSource s declarative markup should look similar to the following:

```
<asp:GridView ID="Employees" runat="server" AutoGenerateColumns="False"
    DataKeyNames="EmployeeID" DataSourceID="EmployeesDataSource">
    <Columns>
        <asp:CommandField ShowDeleteButton="True" />
        <asp:BoundField DataField="Title"
            HeaderText="Title"
            SortExpression="Title" />
        <asp:BoundField DataField="LastName"
            HeaderText="Last Name"
            SortExpression="LastName" />
        <asp:BoundField DataField="FirstName"
            HeaderText="First Name"
            SortExpression="FirstName" />
        <asp:BoundField DataField="ManagerFirstName"
            HeaderText="Manager's First Name"
            SortExpression="ManagerFirstName" />
        <asp:BoundField DataField="ManagerLastName"
            HeaderText="Manager's Last Name"
            SortExpression="ManagerLastName" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="EmployeesDataSource" runat="server"
    DeleteMethod="DeleteEmployee" OldValuesParameterFormatString="{0}"
    SelectMethod="GetEmployees" TypeName="EmployeesBLLWithSprocs">
    <DeleteParameters>
        <asp:Parameter Name="employeeID" Type="Int32" />
    </DeleteParameters>
</asp:ObjectDataSource>
```

Test out the page by visiting it through a browser. As Figure 14 shows, the page will list each employee and his or her manager s name (assuming they have one).

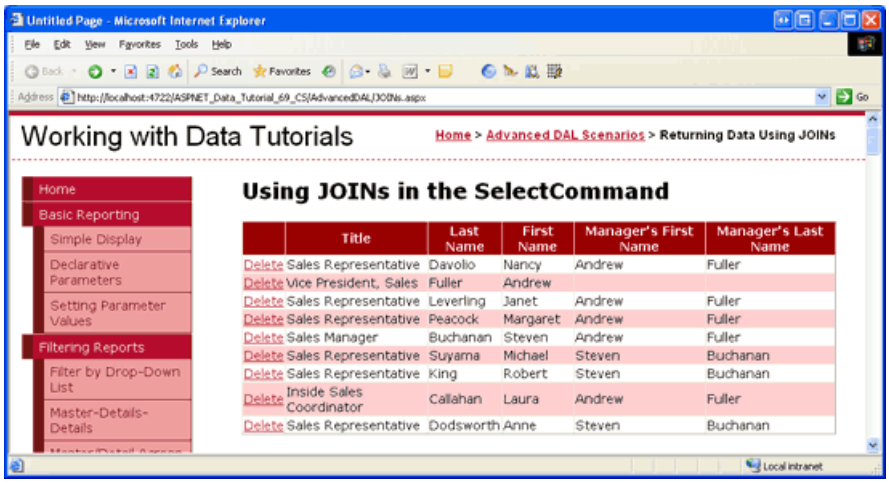


Figure 14: The `JOIN` in the `Employees_Select` Stored Procedure Returns the Manager s Name (Click to view full-size image)

Clicking the Delete button starts the deleting workflow, which culminates in the execution of the `Employees_Delete` stored procedure. However, the attempted `DELETE` statement in the stored procedure fails because of a foreign key constraint violation (see Figure 15). Specifically, each employee has one or more records

in the `Orders` table, causing the delete to fail.

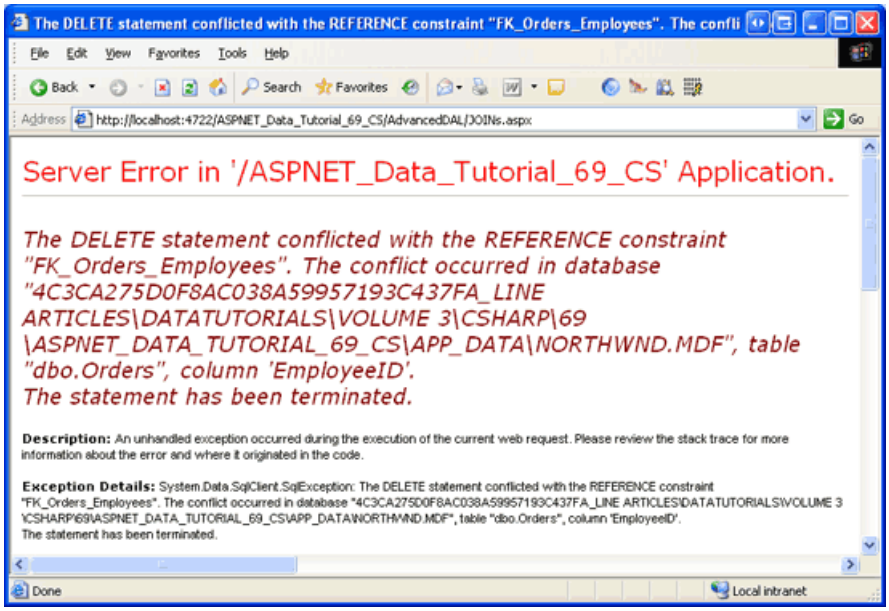


Figure 15: Deleting an Employee That has Corresponding Orders Results in a Foreign Key Constraint Violation (Click to view full-size image)

To allow an employee to be deleted you could:

- Update the foreign key constraint to cascade deletes,
- Manually delete the records from the `Orders` table for the employee(s) you want to delete, or
- Update the `Employees_Delete` stored procedure to first delete the related records from the `Orders` table before deleting the `Employees` record. We discussed this technique in the Using Existing Stored Procedures for the Typed DataSet s TableAdapters tutorial.

I leave this as an exercise for the reader.

Summary

When working with relational databases, it is common for queries to pull their data from multiple, related tables. Correlated subqueries and `JOIN` s provide two different techniques for accessing data from related tables in a query. In previous tutorials we most commonly made use of correlated subqueries because the TableAdapter cannot auto-generate `INSERT`, `UPDATE`, and `DELETE` statements for queries involving `JOIN` s. While these values can be provided manually, when using ad-hoc SQL statements any customizations will be overwritten when the TableAdapter Configuration wizard is completed.

Fortunately, TableAdapters created using stored procedures do not suffer from the same brittleness as those created using ad-hoc SQL statements. Therefore, it is feasible to create a TableAdapter whose main query uses a `JOIN` when using stored procedures. In this tutorial we saw how to create such a TableAdapter. We started by using a `JOIN`-less `SELECT` query for the TableAdapter s main query so that the corresponding insert, update, and delete stored procedures would be auto-created. With the TableAdapter s initial configuration complete, we augmented the `SelectCommand` stored procedure to use a `JOIN` and re-ran the TableAdapter Configuration wizard to update the `EmployeesDataTable` s columns.

Re-running the TableAdapter Configuration wizard automatically updated the `EmployeesDataTable` columns to reflect the data fields returned by the `Employees_Select` stored procedure. Alternatively, we could have added these columns manually to the DataTable. We will explore manually adding columns to the DataTable in the next tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Hilton Geisenow, David Suru, and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Adding Additional DataTable Columns (C#)

2/20/2020 • 14 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

When using the TableAdapter Wizard to create a Typed DataSet, the corresponding DataTable contains the columns returned by the main database query. But there are occasions when the DataTable needs to include additional columns. In this tutorial we learn why stored procedures are recommended when we need additional DataTable columns.

Introduction

When adding a TableAdapter to a Typed DataSet, the corresponding DataTable s schema is determined by the TableAdapter s main query. For example, if the main query returns data fields *A*, *B*, and *C*, the DataTable will have three corresponding columns named *A*, *B*, and *C*. In addition to its main query, a TableAdapter can include additional queries that return, perhaps, a subset of the data based on some parameter. For instance, in addition to the `ProductsTableAdapter` s main query, which returns information about all products, it also contains methods like `GetProductsByCategoryID(categoryID)` and `GetProductByProductID(productID)`, which return specific product information based on a supplied parameter.

The model of having the DataTable s schema reflect the TableAdapter s main query works well if all of the TableAdapter s methods return the same or fewer data fields than those specified in the main query. If a TableAdapter method needs to return additional data fields, then we should expand the DataTable s schema accordingly. In the Master/Detail Using a Bulleted List of Master Records with a Details DataList tutorial we added a method to the `CategoriesTableAdapter` that returned the `CategoryID`, `CategoryName`, and `Description` data fields defined in the main query plus `NumberOfProducts`, an additional data field that reported the number of products associated with each category. We manually added a new column to the `CategoriesDataTable` in order to capture the `NumberOfProducts` data field value from this new method.

As discussed in the Uploading Files tutorial, great care must be taken with TableAdapters that use ad-hoc SQL statements and have methods whose data fields do not precisely match the main query. If the TableAdapter Configuration wizard is re-run, it will update all of the TableAdapter s methods so that their data field list matches the main query. Consequently, any methods with customized column lists will revert to the main query s column list and not return the expected data. This issue does not arise when using stored procedures.

In this tutorial we will look at how to extend a DataTable s schema to include additional columns. Due to the brittleness of the TableAdapter when using ad-hoc SQL statements, in this tutorial we will use stored procedures. Refer to the Creating New Stored Procedures for the Typed DataSet s TableAdapters and Using Existing Stored Procedures for the Typed DataSet s TableAdapters tutorials for more information on configuring a TableAdapter to use stored procedures.

Step 1: Adding a `PriceQuartile` Column to the `ProductsDataTable`

In the *Creating New Stored Procedures for the Typed DataSet s TableAdapters* tutorial we created a Typed DataSet named `NorthwindWithSprocs`. This DataSet currently contains two DataTables: `ProductsDataTable` and `EmployeesDataTable`. The `ProductsTableAdapter` has the following three methods:

- `GetProducts` - the main query, which returns all records from the `Products` table

- `GetProductsByCategoryID(categoryID)` - returns all products with the specified *categoryID*.
- `GetProductByProductID(productID)` - returns the particular product with the specified *productID*.

The main query and the two additional methods all return the same set of data fields, namely all of the columns from the `Products` table. There are no correlated subqueries or `JOIN` s pulling related data from the `Categories` or `Suppliers` tables. Therefore, the `ProductsDataTable` has a corresponding column for each field in the `Products` table.

For this tutorial, let s add a method to the `ProductsTableAdapter` named `GetProductsWithPriceQuartile` that returns all of the products. In addition to the standard product data fields, `GetProductsWithPriceQuartile` will also include a `PriceQuartile` data field that indicates under which quartile the product s price falls. For example, those products whose prices are in the most expensive 25% will have a `PriceQuartile` value of 1, while those whose prices fall in the bottom 25% will have a value of 4. Before we worry about creating the stored procedure to return this information, however, we first need to update the `ProductsDataTable` to include a column to hold the `PriceQuartile` results when the `GetProductsWithPriceQuartile` method is used.

Open the `NorthwindWithSprocs` DataSet and right-click on the `ProductsDataTable`. Choose Add from the context-menu and then pick Column.

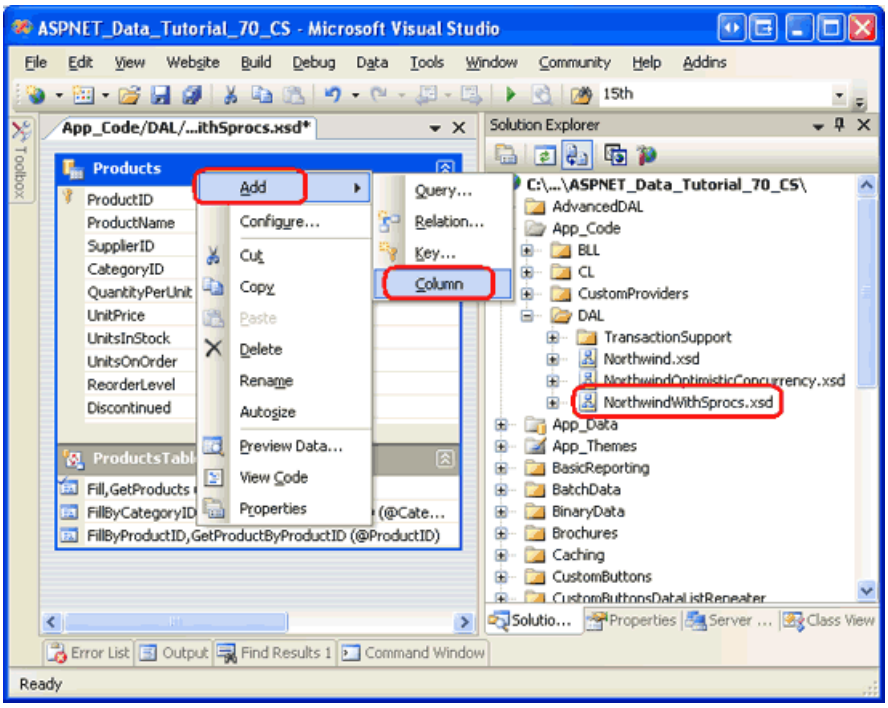


Figure 1: Add a New Column to the `ProductsDataTable` (Click to view full-size image)

This will add a new column to the DataTable named Column1 of type `System.String`. We need to update this column s name to PriceQuartile and its type to `System.Int32` since it will be used to hold a number between 1 and 4. Select the newly-added column in the `ProductsDataTable` and, from the Properties window, set the `Name` property to PriceQuartile and the `DataType` property to `System.Int32`.

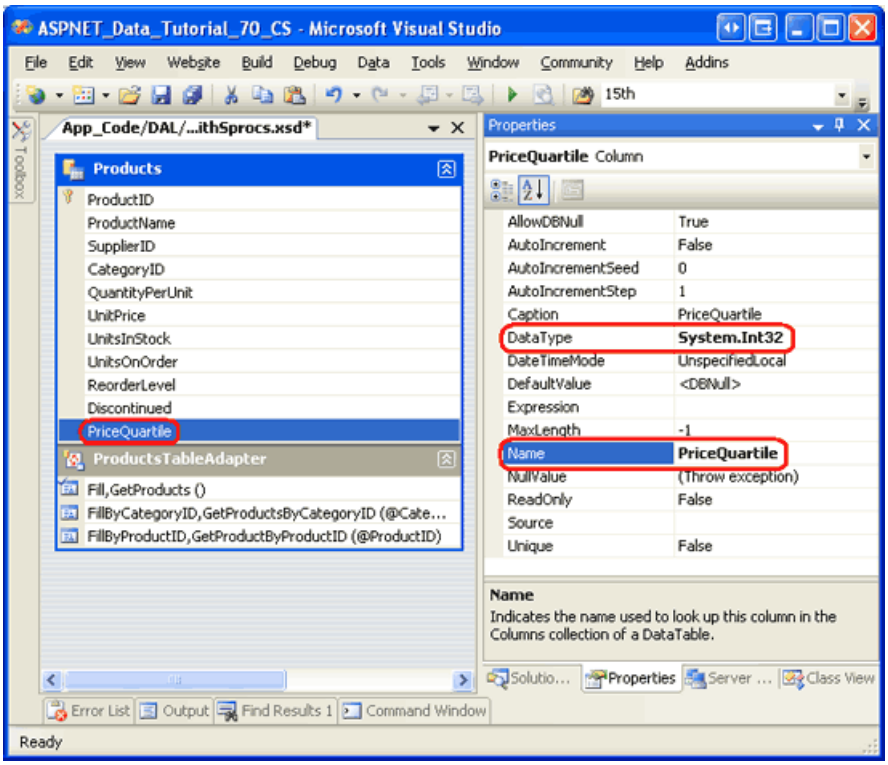


Figure 2: Set the New Column s `Name` and `DataType` Properties (Click to view full-size image)

As Figure 2 shows, there are additional properties that can be set, such as whether the values in the column must be unique, if the column is an auto-increment column, whether or not database `NULL` values are allowed, and so on. Leave these values set to their defaults.

Step 2: Creating the `GetProductsWithPriceQuartile` Method

Now that the `ProductsDataTable` has been updated to include the `PriceQuartile` column, we are ready to create the `GetProductsWithPriceQuartile` method. Start by right-clicking on the TableAdapter and choosing Add Query from the context-menu. This brings up the TableAdapter Query Configuration wizard, which first prompts us as to whether we want to use ad-hoc SQL statements or a new or existing stored procedure. Since we don t yet have a stored procedure that returns the price quartile data, let s allow the TableAdapter to create this stored procedure for us. Select the Create new stored procedure option and click Next.

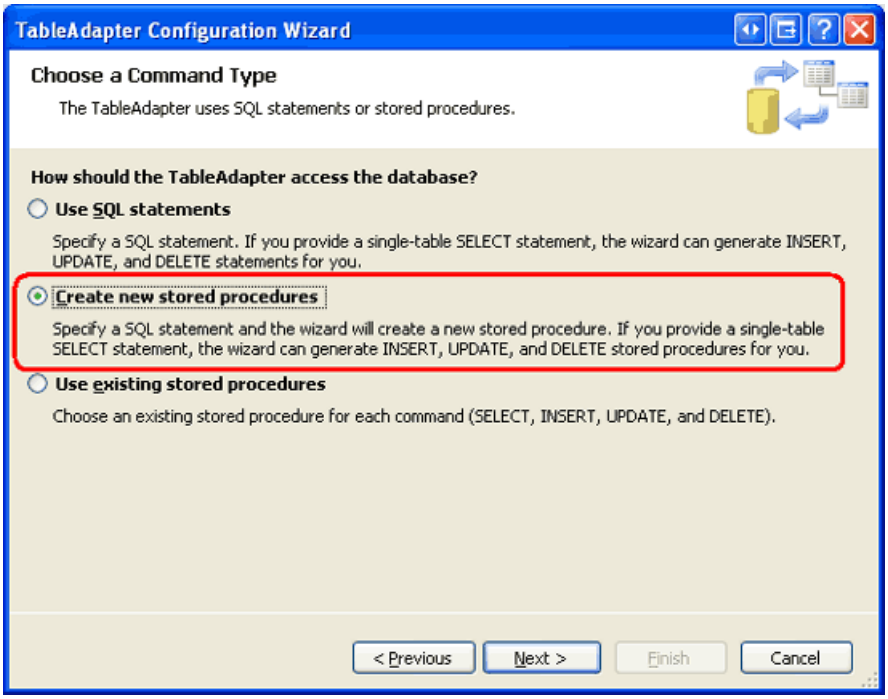


Figure 3: Instruct the TableAdapter Wizard to Create the Stored Procedure For Us (Click to view full-size image)

In the subsequent screen, shown in Figure 4, the wizard asks us what type of query to add. Since the `GetProductsWithPriceQuartile` method will return all columns and records from the `Products` table, select the SELECT which returns rows option and click Next.

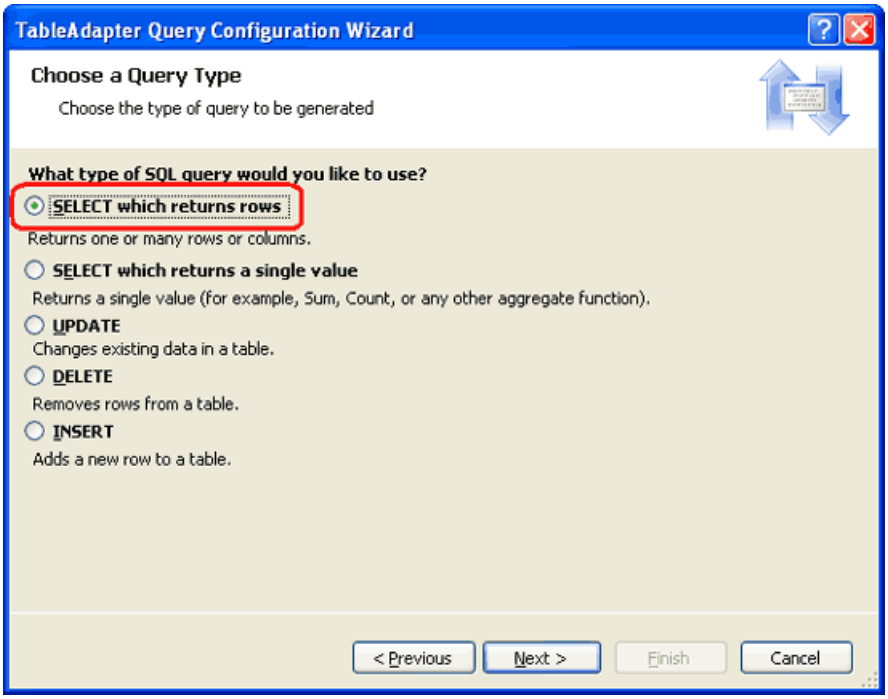


Figure 4: Our Query will be a `SELECT` Statement that Returns Multiple Rows (Click to view full-size image)

Next we are prompted for the `SELECT` query. Enter the following query into the wizard:

```
SELECT ProductID, ProductName, SupplierID, CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued,
       NTILE(4) OVER (ORDER BY UnitPrice DESC) as PriceQuartile
FROM Products
```

The above query uses SQL Server 2005 s new `NTILE` function to divide the results into four groups where the groups are determined by the `UnitPrice` values sorted in descending order.

Unfortunately, the Query Builder does not know how to parse the `OVER` keyword and will display an error when parsing the above query. Therefore, enter the above query directly in the textbox in the wizard without using the Query Builder.

> **NOTE**
> For more information on NTILE and SQL Server 2005 s other ranking functions, see Returning Ranked Results with Microsoft SQL Server 2005 and the Ranking Functions section from the SQL Server 2005 Books Online.

After entering the `SELECT` query and clicking Next, the wizard asks us to provide a name for the stored procedure it will create. Name the new stored procedure `Products_SelectWithPriceQuartile` and click Next.

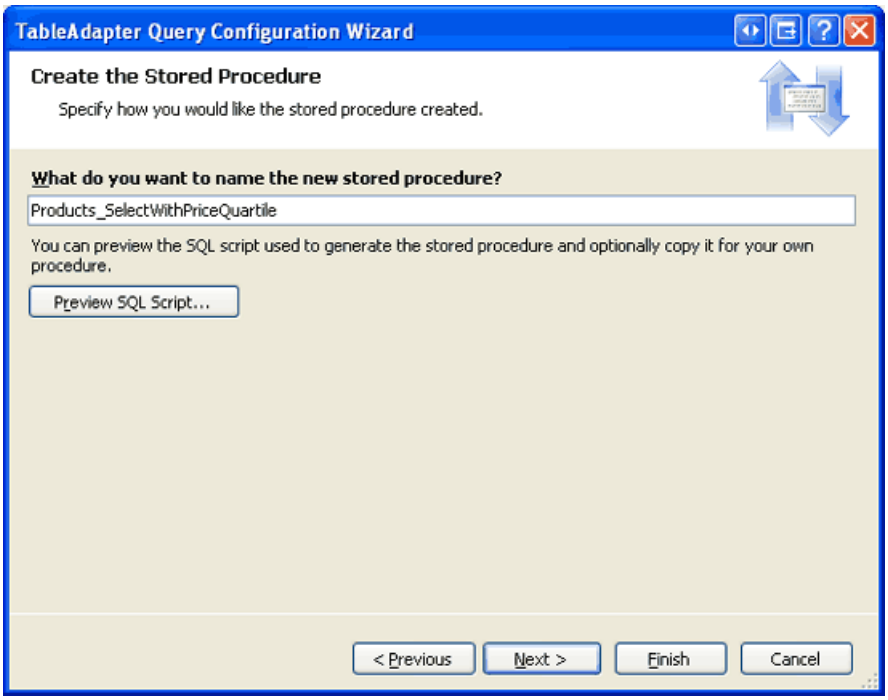


Figure 5: Name the Stored Procedure `Products_SelectWithPriceQuartile` (Click to view full-size image)

Lastly, we are prompted to name the TableAdapter methods. Leave both the Fill a DataTable and Return a DataTable checkboxes checked and name the methods `FillWithPriceQuartile` and `GetProductsWithPriceQuartile`.

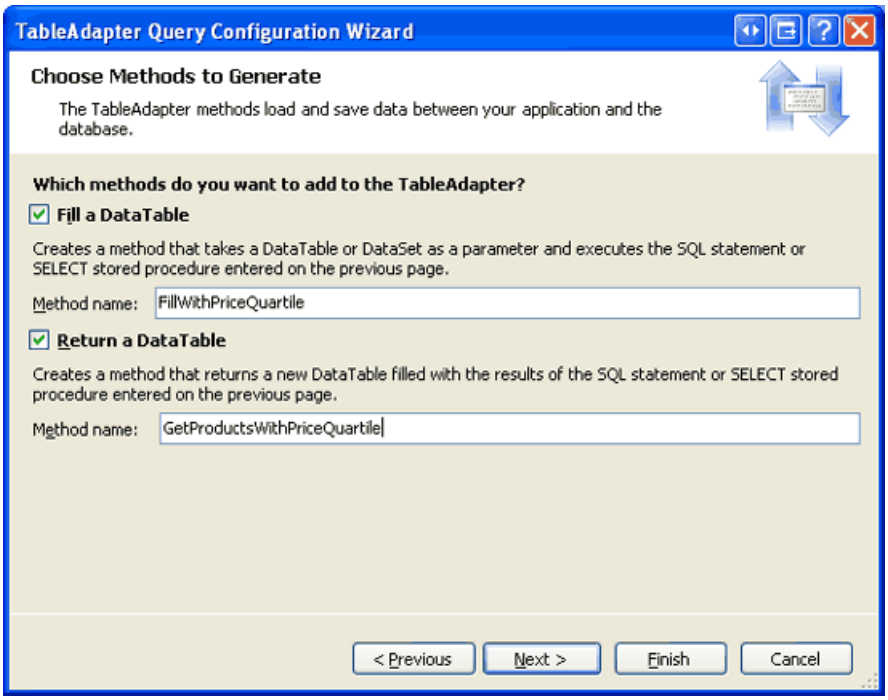


Figure 6: Name the TableAdapter s Methods and Click Finish (Click to view full-size image)

With the `SELECT` query specified and the stored procedure and TableAdapter methods named, click Finish to complete the wizard. At this point you may get a warning or two from the wizard saying that The `OVER` SQL construct or statement is not supported. These warnings can be ignored.

After completing the wizard, the TableAdapter should include the `FillWithPriceQuartile` and `GetProductsWithPriceQuartile` methods and the database should include a stored procedure named `Products_SelectWithPriceQuartile`. Take a moment to verify that the TableAdapter does indeed contain this new method and that the stored procedure has been correctly added to the database. When checking the database, if you do not see the stored procedure try right-clicking on the Stored Procedures folder and choosing Refresh.

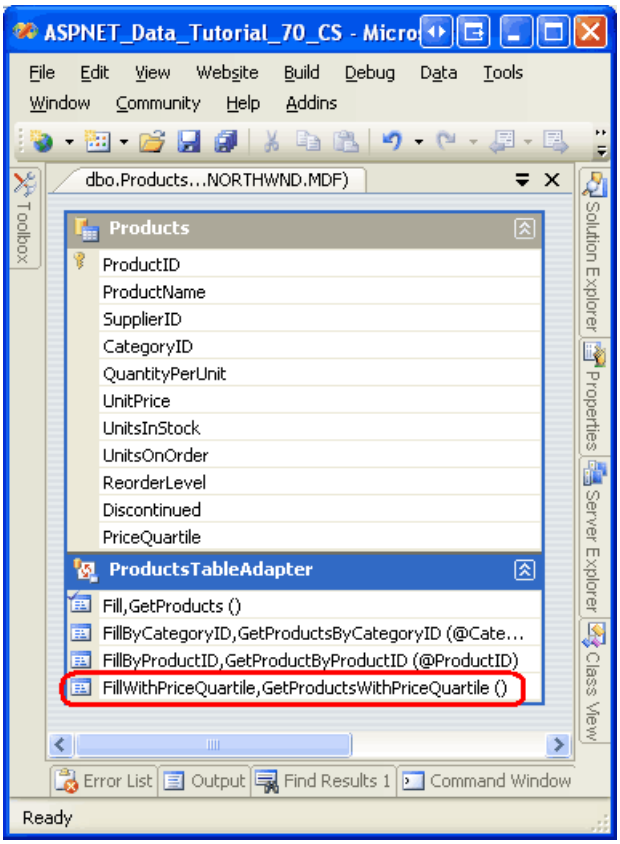


Figure 7: Verify that a New Method Has Been Added to the TableAdapter

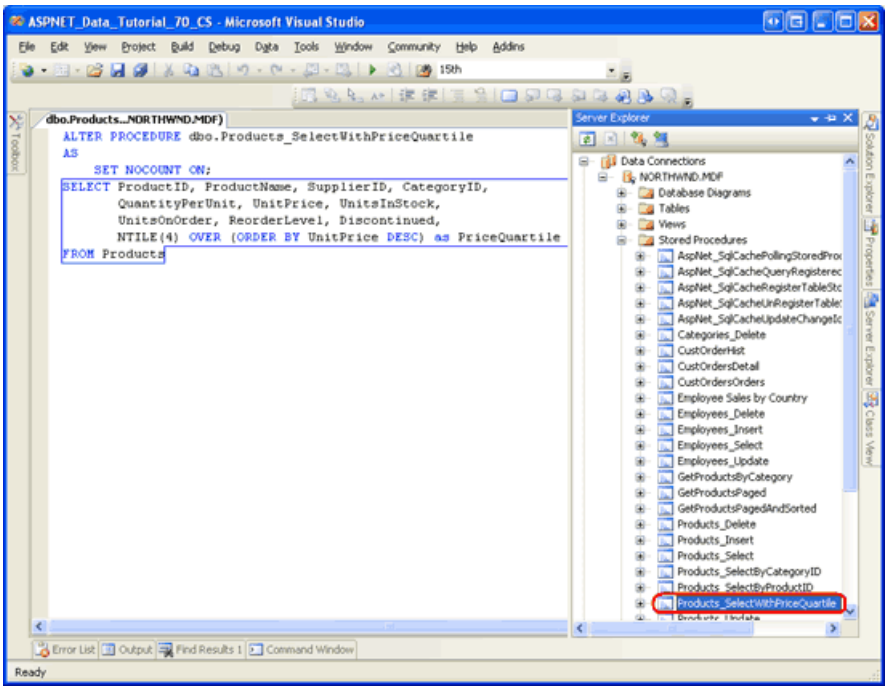


Figure 8: Ensure that the Database Contains the `Products_SelectWithPriceQuartile` Stored Procedure (Click to view full-size image)

NOTE

One of the benefits of using stored procedures instead of ad-hoc SQL statements is that re-running the TableAdapter Configuration wizard will not modify the stored procedures column lists. Verify this by right-clicking on the TableAdapter, choosing the Configure option from the context-menu to start the wizard, and then clicking Finish to complete it. Next, go to the database and view the `Products_SelectWithPriceQuartile` stored procedure. Note that its column list has not been modified. Had we been using ad-hoc SQL statements, re-running the TableAdapter Configuration wizard would have reverted this query s column list to match the main query column list, thereby removing the NTILE statement from the query used by the `GetProductsWithPriceQuartile` method.

When the Data Access Layer s `GetProductsWithPriceQuartile` method is invoked, the TableAdapter executes the `Products_SelectWithPriceQuartile` stored procedure and adds a row to the `ProductsDataTable` for each returned record. The data fields returned by the stored procedure are mapped to the `ProductsDataTable` s columns. Since there is a `PriceQuartile` data field returned from the stored procedure, its value is assigned to the `ProductsDataTable` s `PriceQuartile` column.

For those TableAdapter methods whose queries do not return a `PriceQuartile` data field, the `PriceQuartile` column s value is the value specified by its `DefaultValue` property. As Figure 2 shows, this value is set to `DBNull`, the default. If you would prefer a different default value, simply set the `DefaultValue` property accordingly. Just make sure that the `DefaultValue` value is valid given the column s `DataType` (i.e., `System.Int32` for the `PriceQuartile` column).

At this point we have performed the necessary steps for adding an additional column to a DataTable. To verify that this additional column works as expected, let s create an ASP.NET page that displays each product s name, price, and price quartile. Before we do that, though, we first need to update the Business Logic Layer to include a method that calls down to the DAL s `GetProductsWithPriceQuartile` method. We will update the BLL next, in Step 3, and then create the ASP.NET page in Step 4.

Step 3: Augmenting the Business Logic Layer

Before we use the new `GetProductsWithPriceQuartile` method from the Presentation Layer, we should first add a corresponding method to the BLL. Open the `ProductsBLLWithSprocs` class file and add the following code:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Select, false)]
public NorthwindWithSprocs.ProductsDataTable GetProductsWithPriceQuartile()
{
    return Adapter.GetProductsWithPriceQuartile();
}
```

Like the other data retrieval methods in `ProductsBLLWithSprocs`, the `GetProductsWithPriceQuartile` method simply calls the DAL s corresponding `GetProductsWithPriceQuartile` method and returns its results.

Step 4: Displaying the Price Quartile Information in an ASP.NET Web Page

With the BLL addition complete we re ready to create an ASP.NET page that shows the price quartile for each product. Open the `AddingColumns.aspx` page in the `AdvancedDAL` folder and drag a GridView from the Toolbox onto the Designer, setting its `ID` property to `Products`. From the GridView s smart tag, bind it to a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to use the `ProductsBLLWithSprocs` class s `GetProductsWithPriceQuartile` method. Since this will be a read-only grid, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) .

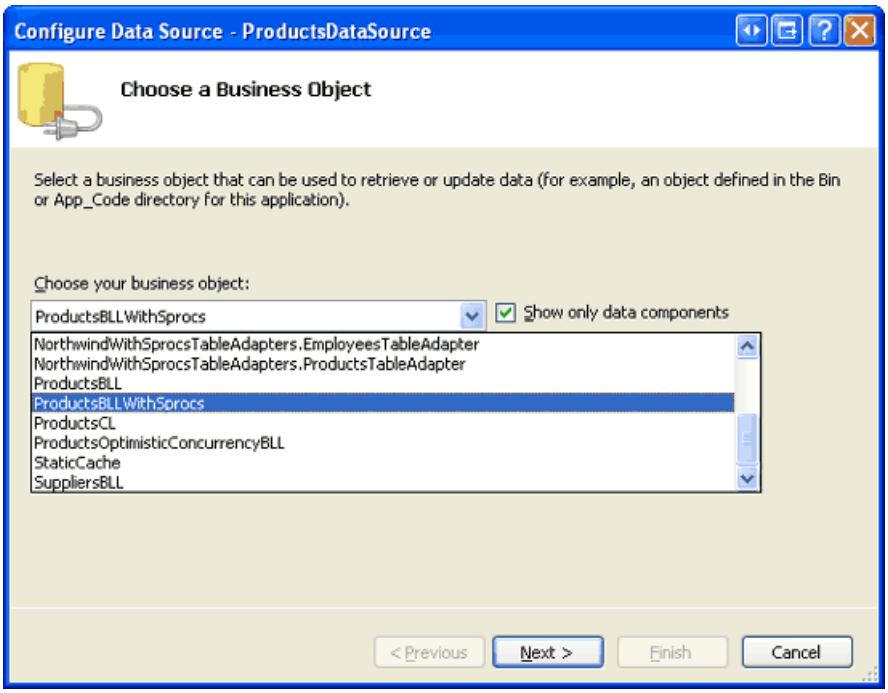


Figure 9: Configure the ObjectDataSource to Use the `ProductsBLLWithSprocs` Class (Click to view full-size image)

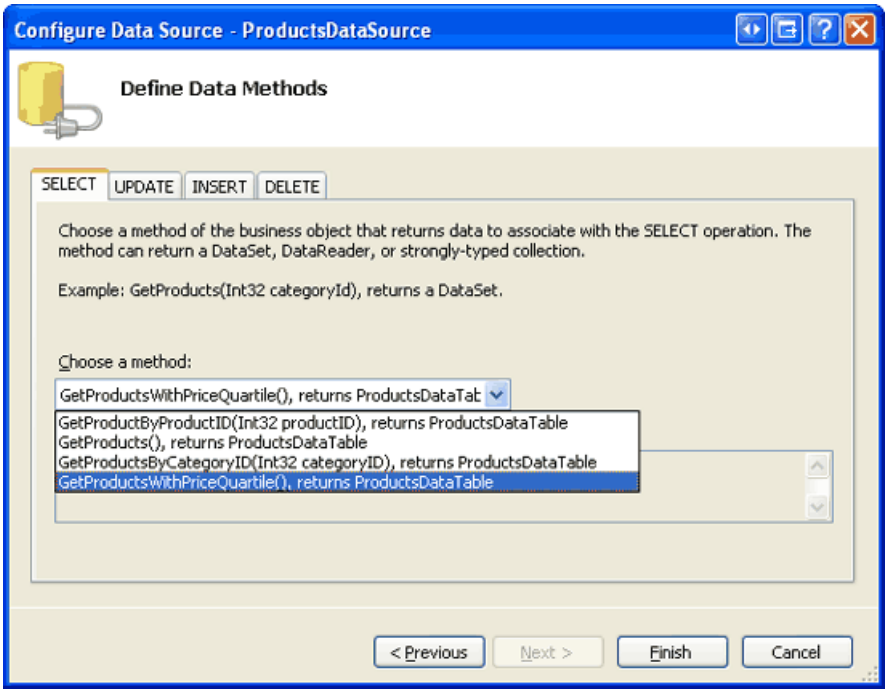


Figure 10: Retrieve Product Information from the `GetProductsWithPriceQuartile` Method (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will automatically add a BoundField or CheckBoxField to the GridView for each of the data fields returned by the method. One of these data fields is `PriceQuartile`, which is the column we added to the `ProductsDataTable` in Step 1.

Edit the GridView s fields, removing all but the `ProductName`, `UnitPrice`, and `PriceQuartile` BoundFields. Configure the `UnitPrice` BoundField to format its value as a currency and have the `UnitPrice` and `PriceQuartile` BoundFields right- and center-aligned, respectively. Finally, update the remaining BoundFields `HeaderText` properties to Product, Price, and Price Quartile, respectively. Also, check the Enable Sorting checkbox from the GridView s smart tag.

After these modifications, the GridView and ObjectDataSource s declarative markup should look like the following:

```
<asp:GridView ID="Products" runat="server" AllowSorting="True"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ProductsDataSource">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
            HeaderText="Price" HtmlEncode="False"
            SortExpression="UnitPrice">
            <ItemStyle HorizontalAlign="Right" />
        </asp:BoundField>
        <asp:BoundField DataField="PriceQuartile" HeaderText="Price Quartile"
            SortExpression="PriceQuartile">
            <ItemStyle HorizontalAlign="Center" />
        </asp:BoundField>
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProductsWithPriceQuartile"
    TypeName="ProductsBLLWithSprocs">
</asp:ObjectDataSource>
```

Figure 11 shows this page when visited through a browser. Note that, initially, the products are ordered by their price in descending order with each product assigned an appropriate `PriceQuartile` value. Of course this data can be sorted by other criteria with the Price Quartile column value still reflecting the product s ranking with respect to price (see Figure 12).

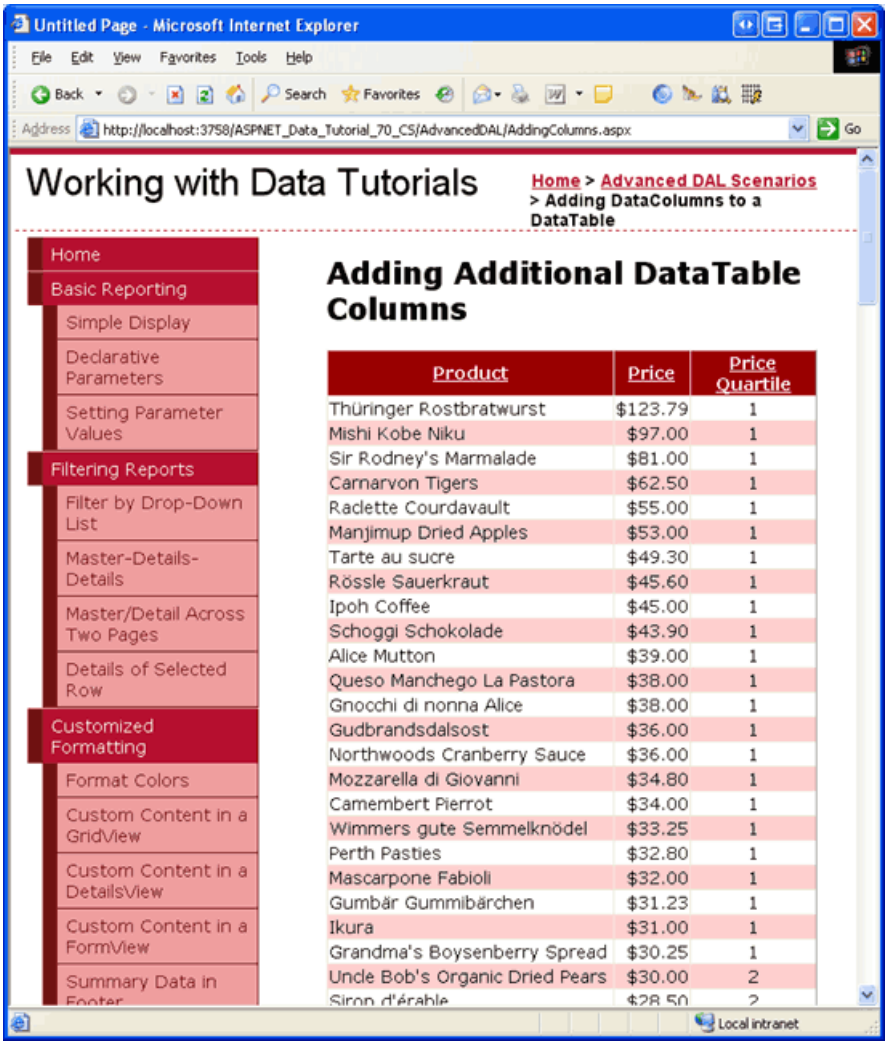


Figure 11: The Products are Ordered by their Prices (Click to view full-size image)

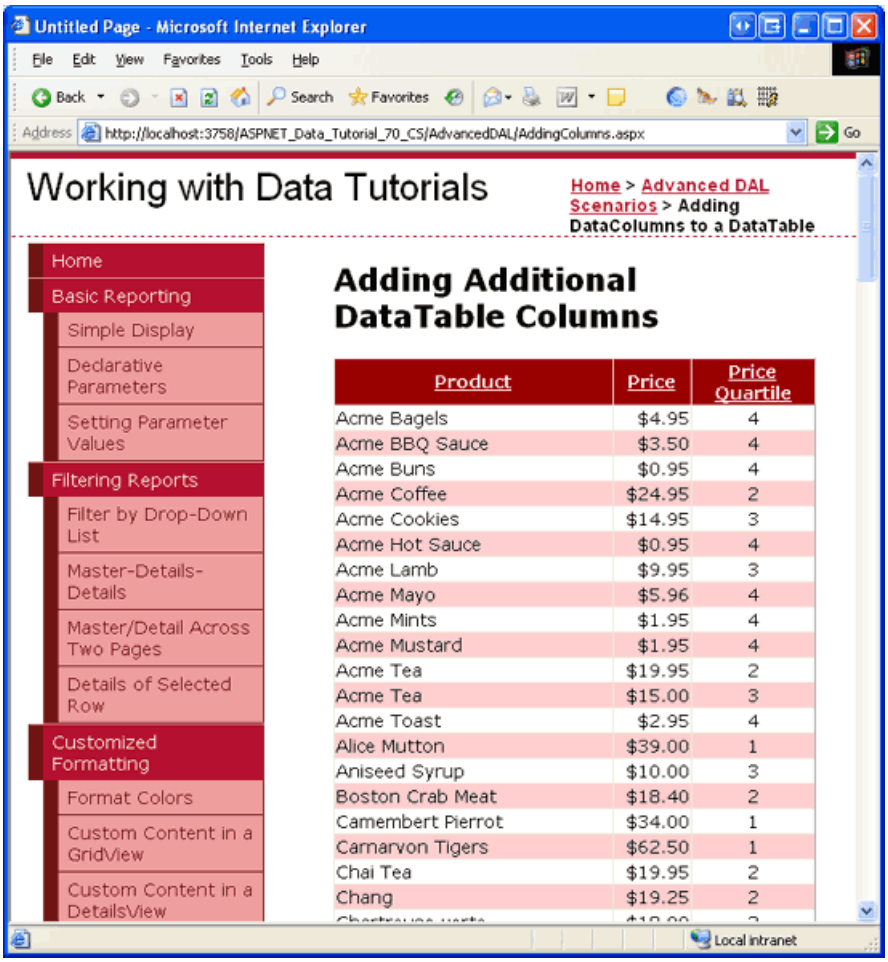


Figure 12: The Products are Ordered by their Names (Click to view full-size image)

> **NOTE**
>
> With a few lines of code we could augment the GridView so that it colored the product rows based on their `PriceQuartile` value. We might color those products in the first quartile a light green, those in the second quartile a light yellow, and so forth. I encourage you to take a moment to add this functionality. If you need a refresher on formatting a GridView, consult the Custom Formatting Based Upon Data tutorial.

An Alternative Approach - Creating Another TableAdapter

As we saw in this tutorial, when adding a method to a TableAdapter that returns data fields other than those spelled out by the main query, we can add corresponding columns to the DataTable. Such an approach, however, works well only if there are a small number of methods in the TableAdapter that return different data fields and if those alternate data fields do not vary too much from the main query.

Rather than adding columns to the DataTable, you can instead add another TableAdapter to the DataSet that contains the methods from the first TableAdapter that return different data fields. For this tutorial, rather than add the `PriceQuartile` column to the `ProductsDataTable` (where it is only used by the `GetProductsWithPriceQuartile` method), we could have added an additional TableAdapter to the DataSet named `ProductsWithPriceQuartileTableAdapter` that used the `Products_SelectWithPriceQuartile` stored procedure as its main query. ASP.NET pages that needed to get product information with the price quartile would use the `ProductsWithPriceQuartileTableAdapter`, while those that did not could continue to use the `ProductsTableAdapter`.

By adding a new TableAdapter, the DataTables remain untarnished and their columns precisely mirror the data fields returned by their TableAdapter s methods. However, additional TableAdapters can introduce repetitive tasks and functionality. For example, if those ASP.NET pages that displayed the `PriceQuartile` column also needed to provide insert, update, and delete support, the `ProductsWithPriceQuartileTableAdapter` would need to have its

`InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties properly configured. While these properties would mirror the `ProductsTableAdapter` s, this configuration introduces an extra step. Moreover, there are now two ways to update, delete, or add a product to the database - through the `ProductsTableAdapter` and `ProductsWithPriceQuartileTableAdapter` classes.

The download for this tutorial includes a `ProductsWithPriceQuartileTableAdapter` class in the `NorthwindWithSprocs` DataSet that illustrates this alternative approach.

Summary

In most scenarios, all of the methods in a TableAdapter will return the same set of data fields, but there are times when a particular method or two may need to return an additional field. For example, in the Master/Detail Using a Bulleted List of Master Records with a Details DataList tutorial we added a method to the `CategoriesTableAdapter` that, in addition to the main query s data fields, returned a `NumberOfProducts` field that reported the number of products associated with each category. In this tutorial we looked at adding a method in the `ProductsTableAdapter` that returned a `PriceQuartile` field in addition to the main query s data fields. To capture additional data fields returned by the TableAdapter s methods we need to add corresponding columns to the DataTable.

If you plan on manually adding columns to the DataTable, it is recommended that the TableAdapter use stored procedures. If the TableAdapter uses ad-hoc SQL statements, any time the TableAdapter Configuration wizard is run all of the methods data field lists revert to the data fields returned by the main query. This problem does not extend to stored procedures, which is why they are recommended and were used in this tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Randy Schmidt, Jacky Goor, Bernadette Leigh, and Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Working with Computed Columns (C#)

2/20/2020 • 18 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

When creating a database table, Microsoft SQL Server allows you to define a computed column whose value is calculated from an expression that usually references other values in the same database record. Such values are read-only at the database, which requires special considerations when working with TableAdapters. In this tutorial we learn how to meet the challenges posed by computed columns.

Introduction

Microsoft SQL Server allows for *computed columns*, which are columns whose values are calculated from an expression that usually references the values from other columns in the same table. As an example, a time tracking data model might have a table named `ServiceLog` with columns including `ServicePerformed`, `EmployeeID`, `Rate`, and `Duration`, among others. While the amount due per service item (being the rate multiplied by the duration) could be calculated through a web page or other programmatic interface, it might be handy to include a column in the `ServiceLog` table named `AmountDue` that reported this information. This column could be created as a normal column, but it would need to be updated anytime the `Rate` or `Duration` column values changed. A better approach would be to make the `AmountDue` column a computed column using the expression `Rate * Duration`. Doing so would cause SQL Server to automatically calculate the `AmountDue` column value whenever it was referenced in a query.

Since a computed column s value is determined by an expression, such columns are read-only and therefore cannot have values assigned to them in `INSERT` or `UPDATE` statements. However, when computed columns are part of the main query for a TableAdapter that uses ad-hoc SQL statements, they are automatically included in the auto-generated `INSERT` and `UPDATE` statements. Consequently, the TableAdapter s `INSERT` and `UPDATE` queries and `InsertCommand` and `UpdateCommand` properties must be updated to remove references to any computed columns.

One challenge of using computed columns with a TableAdapter that uses ad-hoc SQL statements is that the TableAdapter s `INSERT` and `UPDATE` queries are automatically regenerated any time the TableAdapter Configuration wizard is completed. Therefore, the computed columns manually removed from the `INSERT` and `UPDATE` queries will reappear if the wizard is re-run. Although TableAdapters that use stored procedures don t suffer from this brittleness, they do have their own quirks that we will address in Step 3.

In this tutorial we will add a computed column to the `Suppliers` table in the Northwind database and then create a corresponding TableAdapter to work with this table and its computed column. We will have our TableAdapter use stored procedures instead of ad-hoc SQL statements so that our customizations aren't lost when the TableAdapter Configuration wizard is used.

Let s get started!

Step 1: Adding a Computed Column to the `Suppliers` Table

The Northwind database does not have any computed columns so we will need to add one ourselves. For this tutorial let s add a computed column to the `Suppliers` table called `FullContactName` that returns the contact s name, title, and the company they work for in the following format: `ContactName` (`ContactTitle` , `CompanyName`).

This computed column might be used in reports when displaying information about suppliers.

Start by opening the `Suppliers` table definition by right-clicking on the `Suppliers` table in the Server Explorer and choosing Open Table Definition from the context-menu. This will display the columns of the table and their properties, such as their data type, whether they allow `NULL` s, and so forth. To add a computed column, start by typing in the name of the column into the table definition. Next, enter its expression into the (Formula) textbox under the Computed Column Specification section in the Column Properties window (see Figure 1). Name the computed column `FullContactName` and use the following expression:

```
ContactName + ' (' + CASE WHEN ContactTitle IS NOT NULL THEN
    ContactTitle + ', ' ELSE '' END + CompanyName + ')'
```

Note that strings can be concatenated in SQL using the `+` operator. The `CASE` statement can be used like a conditional in a traditional programming language. In the above expression the `CASE` statement can be read as: If `ContactTitle` is not `NULL` then output the `ContactTitle` value concatenated with a comma, otherwise emit nothing. For more on the usefulness of the `CASE` statement, see The Power of SQL `CASE` Statements.

> **NOTE**
>
> Instead of using a `CASE` statement here, we could have alternatively used `ISNULL(ContactTitle, '')`. `ISNULL(checkExpression, replacementValue)` returns *checkExpression* if it is non-NULL, otherwise it returns *replacementValue*. While either `ISNULL` or `CASE` will work in this instance, there are more intricate scenarios where the flexibility of the `CASE` statement cannot be matched by `ISNULL`.

After adding this computed column your screen should look like the screen shot in Figure 1.

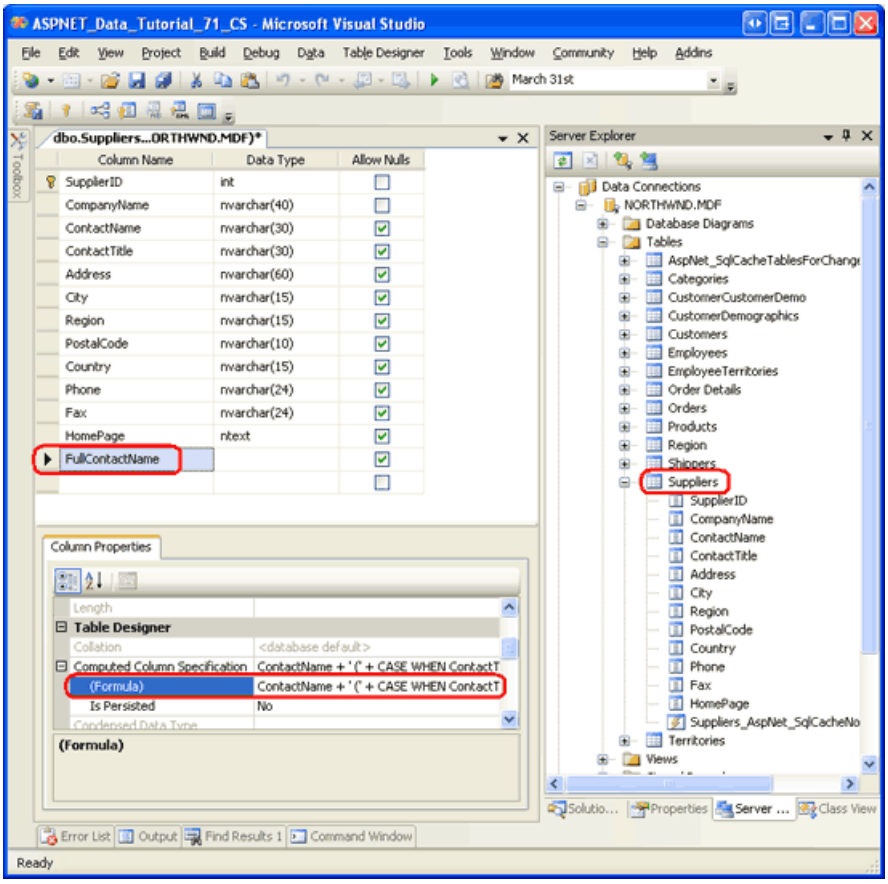


Figure 1: Add a Computed Column Named `FullContactName` to the `Suppliers` Table (Click to view full-size image)

After naming the computed column and entering its expression, save the changes to the table by clicking the Save

icon in the toolbar, by hitting Ctrl+S, or by going to the File menu and choosing Save `Suppliers`.

Saving the table should refresh the Server Explorer, including the just-added column in the `Suppliers` table s column list. Furthermore, the expression entered into the (Formula) textbox will automatically adjust to an equivalent expression that strips unnecessary whitespace, surrounds column names with brackets (`[]`), and includes parentheses to more explicitly show the order of operations:

```
((((([ContactName]+' (')+case when [ContactTitle] IS NOT NULL
    then [ContactTitle]+', ' else '' end)+[CompanyName])+')')
```

For more information on computed columns in Microsoft SQL Server, refer to the technical documentation. Also check out the How to: Specify Computed Columns for a step-by-step walkthrough of creating computed columns.

> **NOTE**
> By default, computed columns are not physically stored in the table but are instead recalculated each time they are referenced in a query. By checking the Is Persisted checkbox, however, you can instruct SQL Server to physically store the computed column in the table. Doing so allows an index to be created on the computed column, which can improve the performance of queries that use the computed column value in their `WHERE` clauses. See Creating Indexes on Computed Columns for more information.

Step 2: Viewing the Computed Column s Values

Before we start work on the Data Access Layer, let s take a minute to view the `FullContactName` values. From the Server Explorer, right-click on the `Suppliers` table name and choose New Query from the context-menu. This will bring up a Query window that prompts us to choose what tables to include in the query. Add the `Suppliers` table and click Close. Next, check the `CompanyName`, `ContactName`, `ContactTitle`, and `FullContactName` columns from the Suppliers table. Finally, click the red exclamation point icon in the Toolbar to execute the query and view the results.

As Figure 2 shows, the results include `FullContactName`, which lists the `CompanyName`, `ContactName`, and `ContactTitle` columns using the format ldquo; `ContactName` (`ContactTitle`, `CompanyName`) .

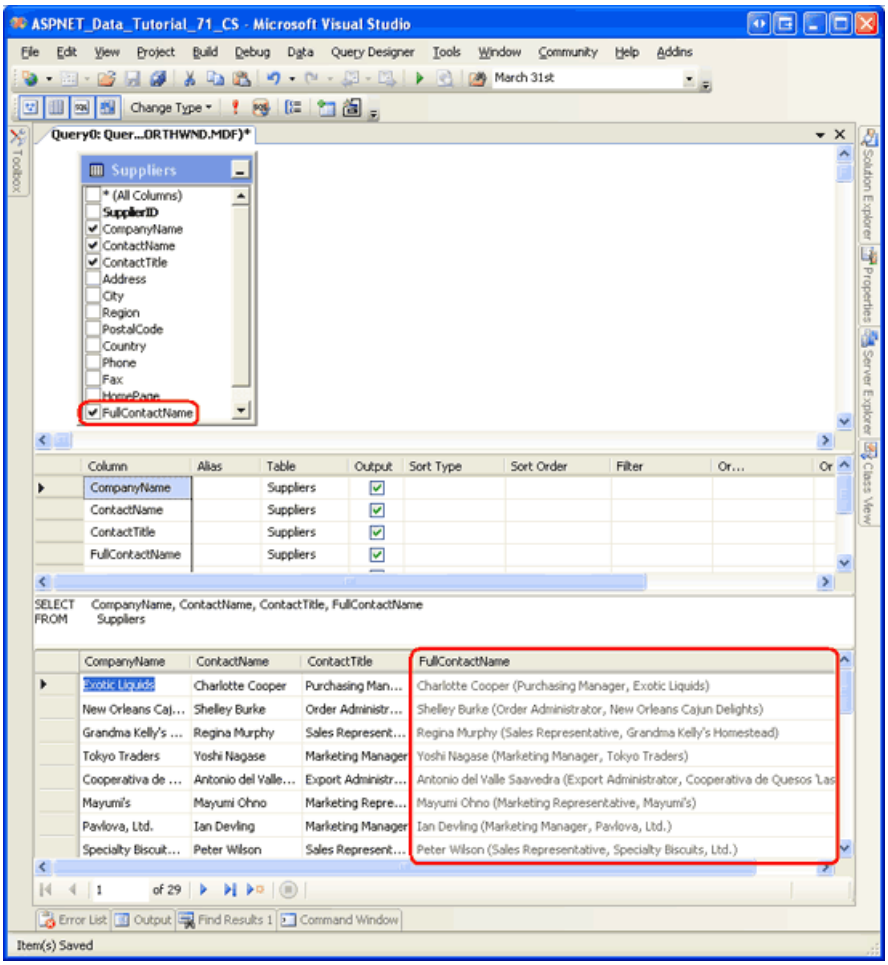


Figure 2: The `FullContactName` Uses the Format `ContactName` (`ContactTitle`, `CompanyName`) (Click to view full-size image)

Step 3: Adding the `SuppliersTableAdapter` to the Data Access Layer

In order to work with the supplier information in our application we need to first create a TableAdapter and DataTable in our DAL. Ideally, this would be accomplished using the same straightforward steps examined in earlier tutorials. However, working with computed columns introduces a few wrinkles that merit discussion.

If you are using a TableAdapter that uses ad-hoc SQL statements, you can simply include the computed column in the TableAdapter s main query via the TableAdapter Configuration wizard. This, however, will auto-generate `INSERT` and `UPDATE` statements that include the computed column. If you attempt to execute one of these methods, a `SqlException` with the message The column *ColumnName* cannot be modified because it is either a computed column or is the result of a UNION operator will be thrown. While the `INSERT` and `UPDATE` statement can be manually adjusted through the TableAdapter s `InsertCommand` and `UpdateCommand` properties, these customizations will be lost whenever the TableAdapter Configuration wizard is re-run.

Due to the brittleness of TableAdapters that use ad-hoc SQL statements, it is recommended that we use stored procedures when working with computed columns. If you are using existing stored procedures, simply configure the TableAdapter as discussed in the Using Existing Stored Procedures for the Typed DataSet s TableAdapters tutorial. If you have the TableAdapter wizard create the stored procedures for you, however, it is important to initially omit any computed columns from the main query. If you include a computed column in the main query, the TableAdapter Configuration wizard will inform you, upon completion, that it cannot create the corresponding stored procedures. In short, we need to initially configure the TableAdapter using a computed column-free main query and then manually update the corresponding stored procedure and the TableAdapter s `SelectCommand` to include the computed column. This approach is similar to the one used in the Updating the TableAdapter to Use `JOIN` *s* tutorial.

For this tutorial, let s add a new TableAdapter and have it automatically create the stored procedures for us. Consequently, we will need to initially omit the `FullContactName` computed column from the main query.

Start by opening the `NorthwindWithSprocs` DataSet in the `~/App_Code/DAL` folder. Right-click in the Designer and, from the context-menu, choose to add a new TableAdapter. This will launch the TableAdapter Configuration wizard. Specify the database to query data from (`NORTHWNDConnectionString` from `Web.config`) and click Next. Since we have not yet created any stored procedures for querying or modifying the `Suppliers` table, select the Create new stored procedures option so that the wizard will create them for us and click Next.

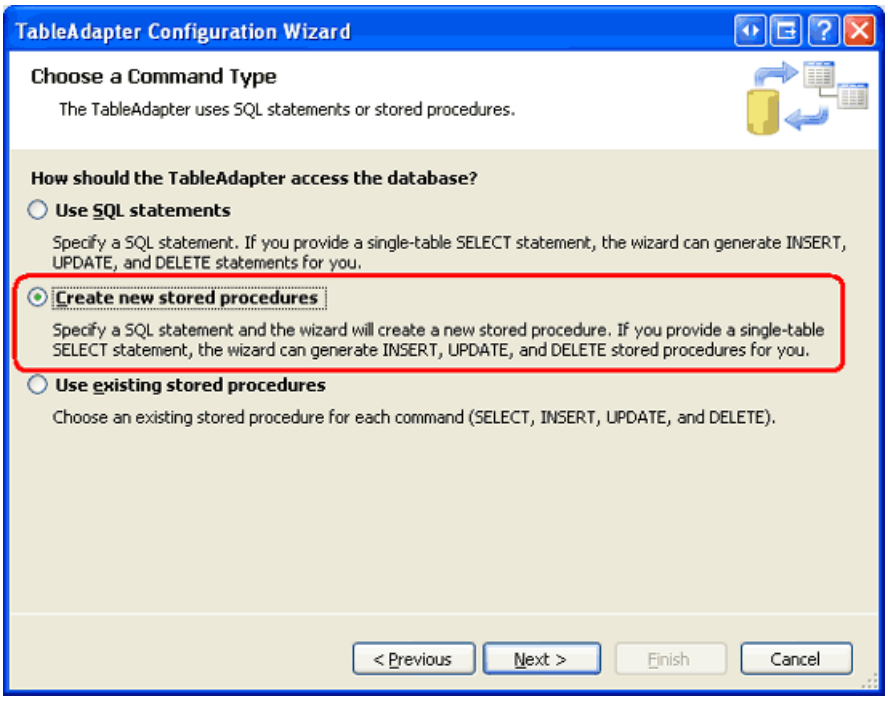


Figure 3: Choose the Create new stored procedures Option (Click to view full-size image)

The subsequent step prompts us for the main query. Enter the following query, which returns the `SupplierID` , `CompanyName` , `ContactName` , and `ContactTitle` columns for each supplier. Note that this query purposefully omits the computed column (`FullContactName`); we will update the corresponding stored procedure to include this column in Step 4.

```
SELECT SupplierID, CompanyName, ContactName, ContactTitle
FROM Suppliers
```

After entering the main query and clicking Next, the wizard allows us to name the four stored procedures it will generate. Name these stored procedures `Suppliers_Select` , `Suppliers_Insert` , `Suppliers_Update` , and `Suppliers_Delete` , as Figure 4 illustrates.

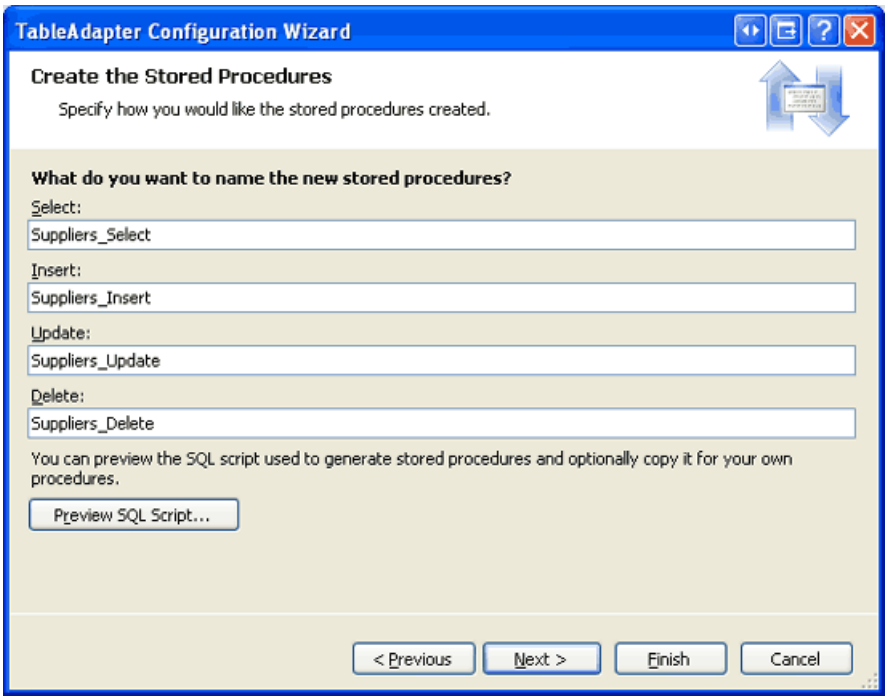


Figure 4: Customize the Names of the Auto-Generated Stored Procedures (Click to view full-size image)

The next wizard step allows us to name the TableAdapter s methods and specify the patterns used to access and update data. Leave all three checkboxes checked, but rename the `GetData` method to `GetSuppliers`. Click Finish to complete the wizard.

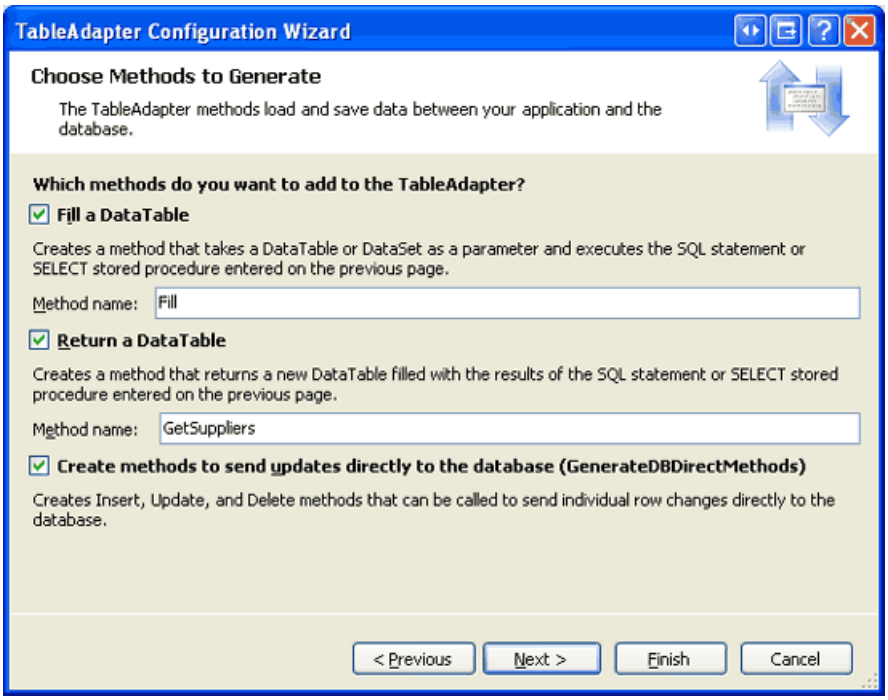


Figure 5: Rename the `GetData` Method to `GetSuppliers` (Click to view full-size image)

Upon clicking Finish, the wizard will create the four stored procedures and add the TableAdapter and corresponding DataTable to the Typed DataSet.

Step 4: Including the Computed Column in the TableAdapter s Main Query

We now need to update the TableAdapter and DataTable created in Step 3 to include the `FullContactName` computed column. This involves two steps:

1. Updating the `Suppliers_Select` stored procedure to return the `FullContactName` computed column, and
2. Updating the DataTable to include a corresponding `FullContactName` column.

Start by navigating to the Server Explorer and drilling down into the Stored Procedures folder. Open the `Suppliers_Select` stored procedure and update the `SELECT` query to include the `FullContactName` computed column:

```
SELECT SupplierID, CompanyName, ContactName, ContactTitle, FullContactName
FROM Suppliers
```

Save the changes to the stored procedure by clicking the Save icon in the Toolbar, by hitting Ctrl+S, or by choosing the Save `Suppliers_Select` option from the File menu.

Next, return to the DataSet Designer, right-click on the `SuppliersTableAdapter`, and choose Configure from the context-menu. Note that the `Suppliers_Select` column now includes the `FullContactName` column in its Data Columns collection.

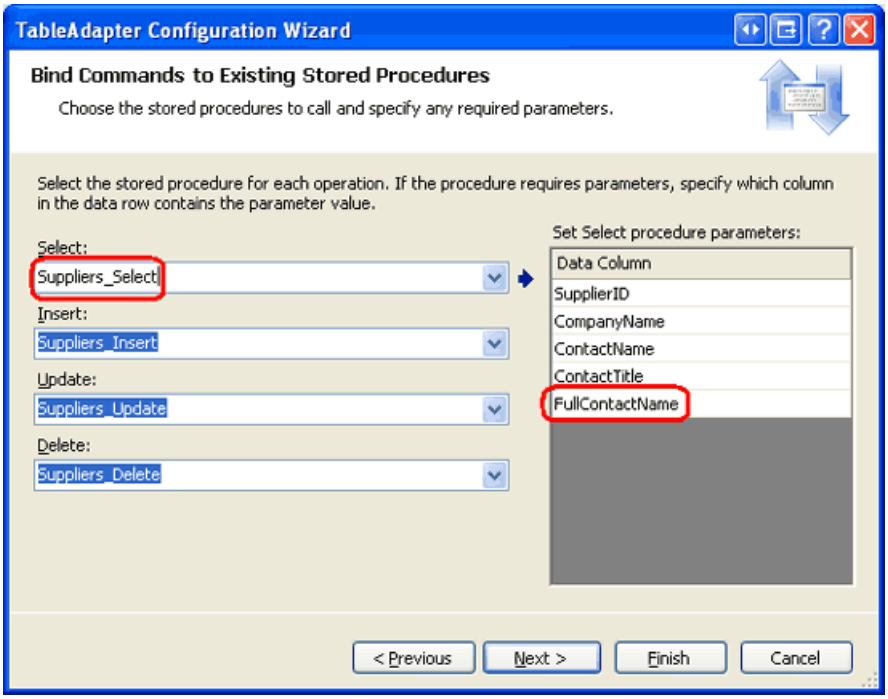


Figure 6: Run the TableAdapter s Configuration Wizard to Update the DataTable s Columns (Click to view full-size image)

Click Finish to complete the wizard. This will automatically add a corresponding column to the `SuppliersDataTable`. The TableAdapter wizard is smart enough to detect that the `FullContactName` column is a computed column and therefore read-only. Consequently, it sets the column s `ReadOnly` property to `true`. To verify this, select the column from the `SuppliersDataTable` and then go to the Properties window (see Figure 7). Note that the `FullContactName` column s `DataType` and `MaxLength` properties are also set accordingly.

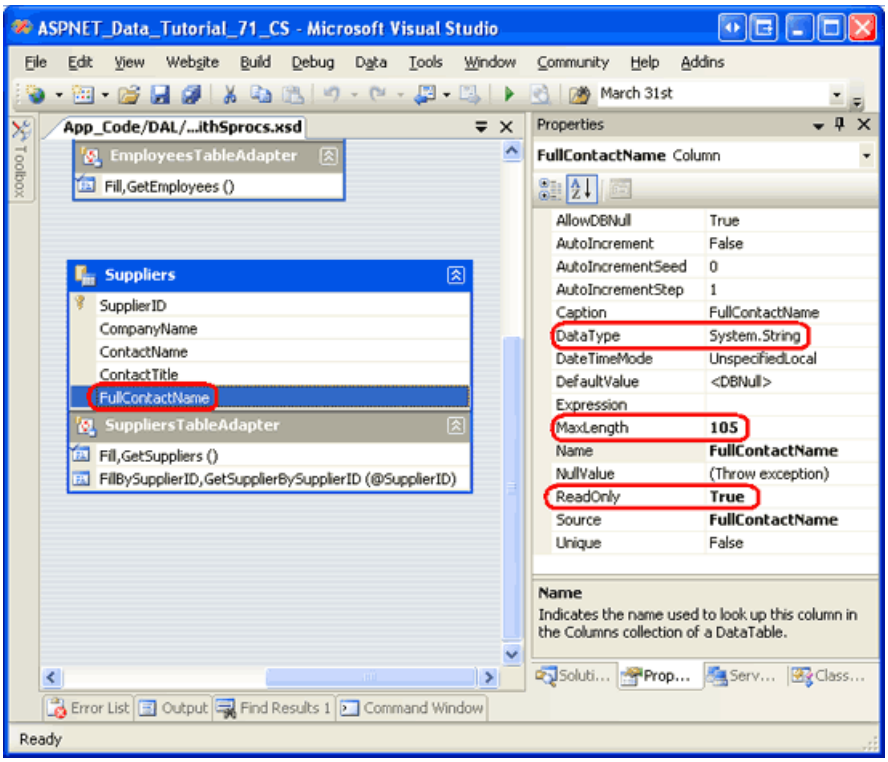


Figure 7: The `FullContactName` Column is Marked as Read-Only (Click to view full-size image)

Step 5: Adding a `GetSupplierBySupplierID` Method to the TableAdapter

For this tutorial we will create an ASP.NET page that displays the suppliers in an updateable grid. In past tutorials we have updated a single record from the Business Logic Layer by retrieving that particular record from the DAL as a strongly-typed DataTable, updating its properties, and then sending the updated DataTable back to the DAL to propagate the changes to the database. To accomplish this first step - retrieving the record being updated from the DAL - we need to first add a `GetSupplierBySupplierID(supplierID)` method to the DAL.

Right-click on the `SuppliersTableAdapter` in the DataSet Design and choose the Add Query option from the context-menu. As we did in Step 3, let the wizard generate a new stored procedure for us by selecting the Create new stored procedure option (refer back to Figure 3 for a screenshot of this wizard step). Since this method will return a record with multiple columns, indicate that we want to use a SQL query that is a SELECT which returns rows and click Next.

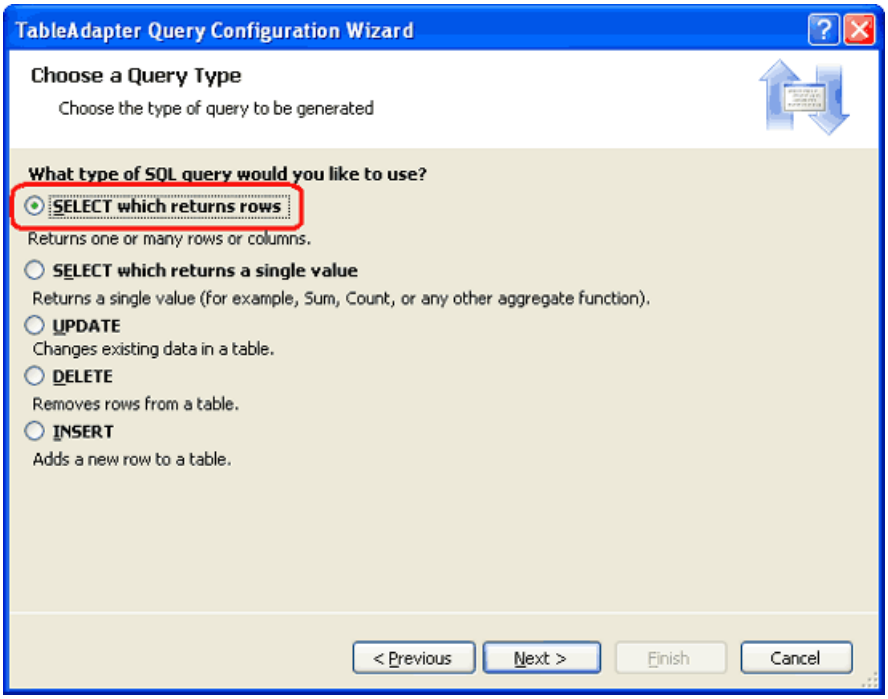


Figure 8: Choose the SELECT which returns rows Option (Click to view full-size image)

The subsequent step prompts us for the query to use for this method. Enter the following, which returns the same data fields as the main query but for a particular supplier.

```
SELECT SupplierID, CompanyName, ContactName, ContactTitle, FullContactName
FROM Suppliers
WHERE SupplierID = @SupplierID
```

The next screen asks us to name the stored procedure that will be auto-generated. Name this stored procedure `Suppliers_SelectBySupplierID` and click Next.

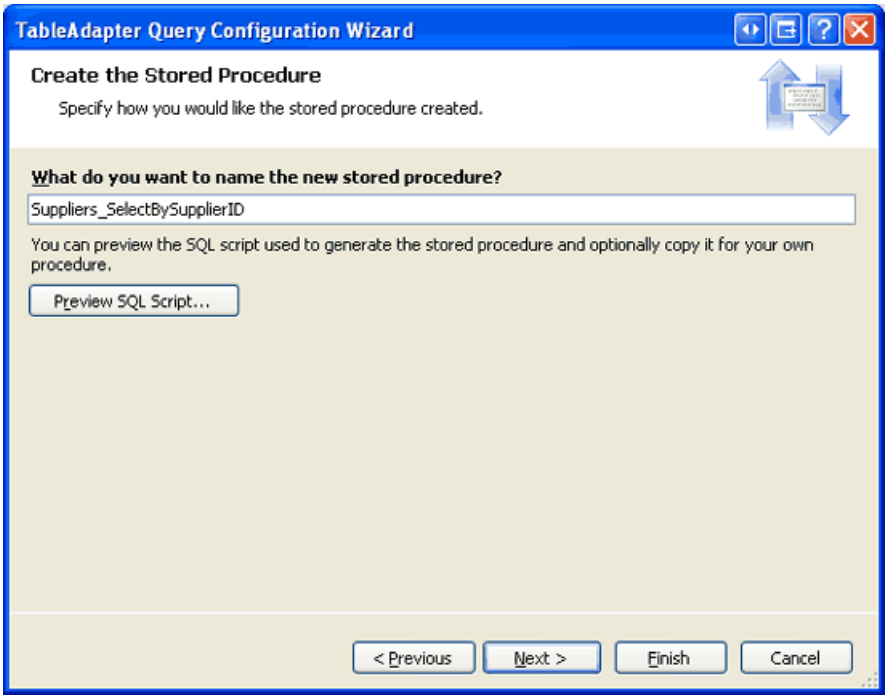


Figure 9: Name the Stored Procedure `Suppliers_SelectBySupplierID` (Click to view full-size image)

Lastly, the wizard prompts us for the data access patterns and method names to use for the TableAdapter. Leave both checkboxes checked, but rename the `FillBy` and `GetDataBy` methods to `FillBySupplierID` and `GetSupplierBySupplierID`, respectively.

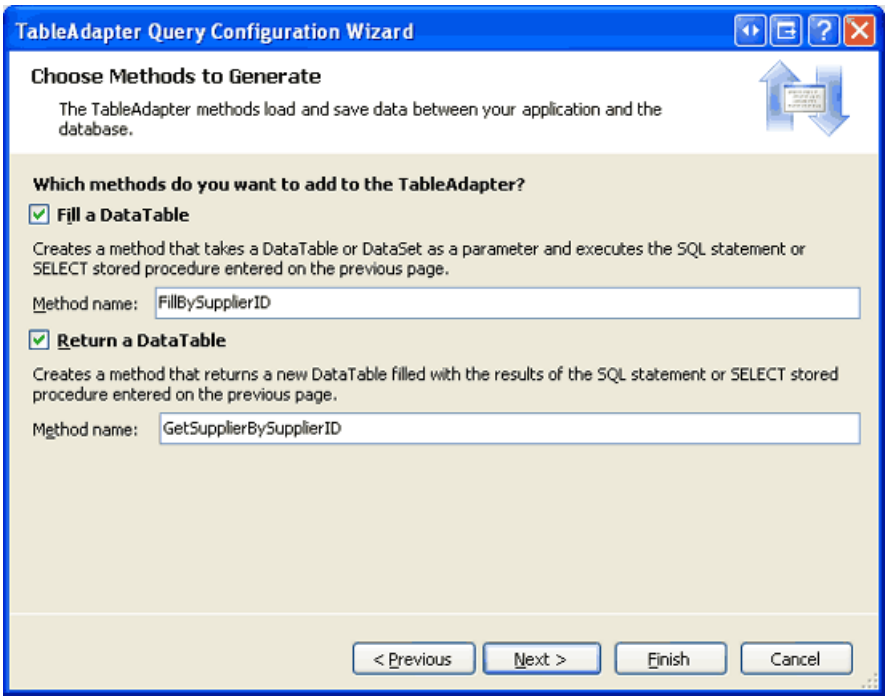


Figure 10: Name the TableAdapter Methods `FillBySupplierID` and `GetSupplierBySupplierID` (Click to view full-size image)

Click Finish to complete the wizard.

Step 6: Creating the Business Logic Layer

Before we create an ASP.NET page that uses the computed column created in Step 1, we first need to add the corresponding methods in the BLL. Our ASP.NET page, which we will create in Step 7, will allow users to view and edit suppliers. Therefore, we need our BLL to provide, at minimum, a method to get all of the suppliers and another to update a particular supplier.

Create a new class file named `SuppliersBLLWithSprocs` in the `~/App_Code/BLL` folder and add the following code:

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using NorthwindWithSprocsTableAdapters;
[System.ComponentModel.DataObject]
public class SuppliersBLLWithSprocs
{
    private SuppliersTableAdapter _suppliersAdapter = null;
    protected SuppliersTableAdapter Adapter
    {
        get
        {
            if (_suppliersAdapter == null)
                _suppliersAdapter = new SuppliersTableAdapter();
            return _suppliersAdapter;
        }
    }
    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Select, true)]
    public NorthwindWithSprocs.SuppliersDataTable GetSuppliers()
    {
        return Adapter.GetSuppliers();
    }
    [System.ComponentModel.DataObjectMethodAttribute
        (System.ComponentModel.DataObjectMethodType.Update, true)]
    public bool UpdateSupplier(string companyName, string contactName,
        string contactTitle, int supplierID)
    {
        NorthwindWithSprocs.SuppliersDataTable suppliers =
            Adapter.GetSupplierBySupplierID(supplierID);
        if (suppliers.Count == 0)
            // no matching record found, return false
            return false;
        NorthwindWithSprocs.SuppliersRow supplier = suppliers[0];
        supplier.CompanyName = companyName;
        if (contactName == null)
            supplier.SetContactNameNull();
        else
            supplier.ContactName = contactName;
        if (contactTitle == null)
            supplier.SetContactTitleNull();
        else
            supplier.ContactTitle = contactTitle;
        // Update the product record
        int rowsAffected = Adapter.Update(supplier);
        // Return true if precisely one row was updated, otherwise false
        return rowsAffected == 1;
    }
}
```

Like the other BLL classes, `SuppliersBLLWithSprocs` has a `protected` `Adapter` property that returns an instance of the `SuppliersTableAdapter` class along with two `public` methods: `GetSuppliers` and `UpdateSupplier`. The `GetSuppliers` method calls and returns the `SuppliersDataTable` returned by the corresponding `GetSupplier` method in the Data Access Layer. The `UpdateSupplier` method retrieves information about the particular supplier being updated via a call to the DAL s `GetSupplierBySupplierID(supplierID)` method. It then updates the `CategoryName`, `ContactName`, and `ContactTitle` properties and commits these changes to the database by calling the Data Access Layer s `Update` method, passing in the modified `SuppliersRow` object.

> **NOTE**
> Except for `SupplierID` and `CompanyName`, all columns in the Suppliers table allow `NULL` values. Therefore, if the passed-in `contactName` or `contactTitle` parameters are `null` we need to set the corresponding `ContactName` and `ContactTitle` properties to a `NULL` database value using the `SetContactNameNull` and `SetContactTitleNull` methods, respectively.

Step 7: Working with the Computed Column from the Presentation Layer

With the computed column added to the `Suppliers` table and the DAL and BLL updated accordingly, we are ready to build an ASP.NET page that works with the `FullContactName` computed column. Start by opening the `ComputedColumns.aspx` page in the `AdvancedDAL` folder and drag a GridView from the Toolbox onto the Designer. Set the GridView s `ID` property to `Suppliers` and, from its smart tag, bind it to a new ObjectDataSource named `SuppliersDataSource`. Configure the ObjectDataSource to use the `SuppliersBLLWithSprocs` class we added back in Step 6 and click Next.

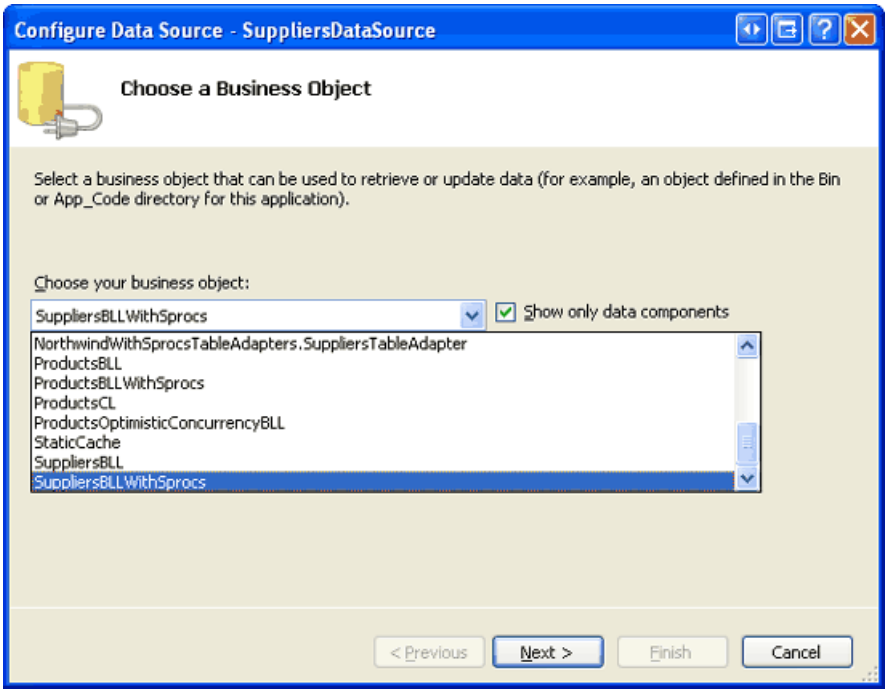


Figure 11: Configure the ObjectDataSource to Use the `SuppliersBLLWithSprocs` Class (Click to view full-size image)

There are only two methods defined in the `SuppliersBLLWithSprocs` class: `GetSuppliers` and `UpdateSupplier`. Ensure that these two methods are specified in the SELECT and UPDATE tabs, respectively, and click Finish to complete the configuration of the ObjectDataSource.

Upon completion of the Data Source Configuration wizard, Visual Studio will add a BoundField for each of the data fields returned. Remove the `SupplierID` BoundField and change the `HeaderText` properties of the `CompanyName`, `ContactName`, `ContactTitle`, and `FullContactName` BoundFields to Company, Contact Name, Title, and Full Contact Name, respectively. From the smart tag, check the Enable Editing checkbox to turn on the GridView s built-in editing capabilities.

In addition to adding BoundFields to the GridView, completion of the Data Source Wizard also causes Visual Studio to set the ObjectDataSource s `OldValuesParameterFormatString` property to original_{0}. Revert this setting back to its default value, {0} .

After making these edits to the GridView and ObjectDataSource, their declarative markup should look similar to

the following:

```
<asp:GridView ID="Suppliers" runat="server" AutoGenerateColumns="False"
    DataKeyNames="SupplierID" DataSourceID="SuppliersDataSource">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:BoundField DataField="CompanyName"
            HeaderText="Company"
            SortExpression="CompanyName" />
        <asp:BoundField DataField="ContactName"
            HeaderText="Contact Name"
            SortExpression="ContactName" />
        <asp:BoundField DataField="ContactTitle"
            HeaderText="Title"
            SortExpression="ContactTitle" />
        <asp:BoundField DataField="FullContactName"
            HeaderText="Full Contact Name"
            SortExpression="FullContactName"
            ReadOnly="True" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="SuppliersDataSource" runat="server"
    SelectMethod="GetSuppliers" TypeName="SuppliersBLLWithSprocs"
        UpdateMethod="UpdateSupplier">
    <UpdateParameters>
        <asp:Parameter Name="companyName" Type="String" />
        <asp:Parameter Name="contactName" Type="String" />
        <asp:Parameter Name="contactTitle" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

Next, visit this page through a browser. As Figure 12 shows, each supplier is listed in a grid that includes the `FullContactName` column, whose value is simply the concatenation of the other three columns formatted as `ContactName` (`ContactTitle` , `CompanyName`) .

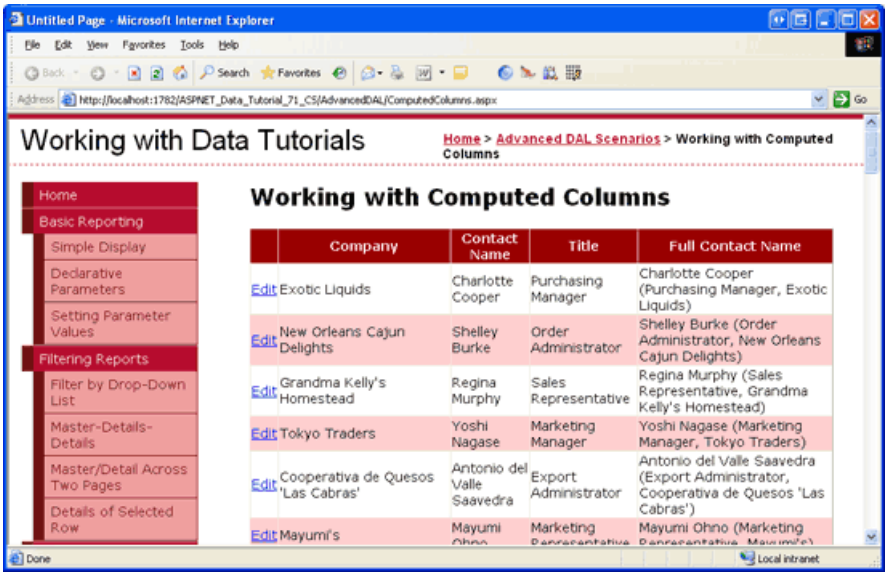


Figure 12: Each Supplier is Listed in the Grid (Click to view full-size image)

Clicking the Edit button for a particular supplier causes a postback and has that row rendered in its editing interface (see Figure 13). The first three columns render in their default editing interface - a TextBox control whose `Text` property is set to the value of the data field. The `FullContactName` column, however, remains as text. When the BoundFields were added to the GridView at the completion of the Data Source Configuration wizard, the `FullContactName` BoundField s `ReadOnly` property was set to `true` because the corresponding `FullContactName` column in the `SuppliersDataTable` has its `ReadOnly` property set to `true` . As noted in Step 4, the

`FullContactName` s `ReadOnly` property was set to `true` because the TableAdapter detected that the column was a computed column.

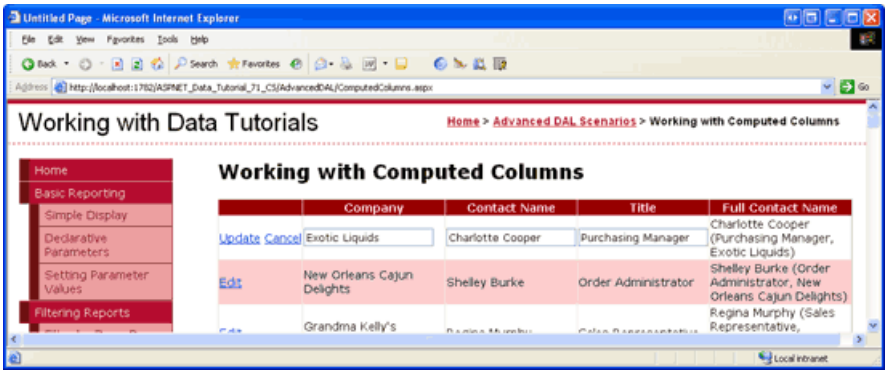


Figure 13: The `FullContactName` Column is Not Editable (Click to view full-size image)

Go ahead and update the value of one or more of the editable columns and click Update. Note how the `FullContactName` s value is automatically updated to reflect the change.

> **NOTE**
>
> The GridView currently uses BoundFields for the editable fields, resulting in the default editing interface. Since the `CompanyName` field is required, it should be converted into a TemplateField that includes a RequiredFieldValidator. I leave this as an exercise for the interested reader. Consult the Adding Validation Controls to the Editing and Inserting Interfaces tutorial for step-by-step instructions on converting a BoundField to a TemplateField and adding validation controls.

Summary

When defining the schema for a table, Microsoft SQL Server allows the inclusion of computed columns. These are columns whose values are calculated from an expression that usually references the values from other columns in the same record. Since the values for computed columns are based on an expression, they are read-only and cannot be assigned a value in an `INSERT` or `UPDATE` statement. This introduces challenges when using a computed column in the main query of a TableAdapter that tries to automatically generate corresponding `INSERT` , `UPDATE` , and `DELETE` statements.

In this tutorial we discussed techniques for circumventing the challenges posed by computed columns. In particular, we used stored procedures in our TableAdapter to overcome the brittleness inherent in TableAdapters that use ad-hoc SQL statements. When having the TableAdapter wizard create new stored procedures, it is important that we have the main query initially omit any computed columns because their presence prevents the data modification stored procedures from being generated. After the TableAdapter has been initially configured, its `SelectCommand` stored procedure can be retooled to include any computed columns.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Hilton Geisenow and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at

mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Configuring the Data Access Layer's Connection- and Command-Level Settings (C#)

2/20/2020 • 12 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> The TableAdapters within a Typed DataSet automatically take care of connecting to the database, issuing commands, and populating a DataTable with the results. There are occasions however when we want to take care of these details ourselves, and in this tutorial we learn how to access the database connection- and command-level settings in the TableAdapter.

Introduction

Throughout the tutorial series we have used Typed DataSets to implement the Data Access Layer and business objects of our layered architecture. As discussed in the first tutorial, the Typed DataSet s DataTables serve as repositories of data whereas the TableAdapters act as wrappers to communicate with the database to retrieve and modify the underlying data. The TableAdapters encapsulate the complexity involved in working with the database and saves us from having to write code to connect to the database, issue a command, or populate the results into a DataTable.

There are times, however, when we need to burrow into the depths of the TableAdapter and write code that works directly with the ADO.NET objects. In the Wrapping Database Modifications within a Transaction tutorial, for example, we added methods to the TableAdapter for beginning, committing, and rolling back ADO.NET transactions. These methods used an internal, manually-created `SqlTransaction` object that was assigned to the TableAdapter s `SqlCommand` objects.

In this tutorial we will examine how to access the database connection- and command-level settings in the TableAdapter. In particular, we will add functionality to the `ProductsTableAdapter` that enables access to the underlying connection string and command timeout settings.

Working with Data Using ADO.NET

The Microsoft .NET Framework contains a plethora of classes designed specifically to work with data. These classes, found within the `System.Data` namespace, are referred to as the *ADO.NET* classes. Some of the classes under the ADO.NET umbrella are tied to a particular *data provider*. You can think of a data provider as a communication channel that allows information to flow between the ADO.NET classes and the underlying data store. There are generalized providers, like OleDb and ODBC, as well as providers that are specially designed for a particular database system. For example, while it is possible to connect to a Microsoft SQL Server database using the OleDb provider, the SqlClient provider is much more efficient as it was designed and optimized specifically for SQL Server.

When programmatically accessing data, the following pattern is commonly used:

- Establish a connection to the database.
- Issue a command.
- For `SELECT` queries, work with the resulting records.

There are separate ADO.NET classes for performing each of these steps. To connect to a database using the

SqlClient provider, for example, use the `SqlConnection` class. To issue an `INSERT`, `UPDATE`, `DELETE`, or `SELECT` command to the database, use the `SqlCommand` class.

Except for the Wrapping Database Modifications within a Transaction tutorial, we have not had to write any low-level ADO.NET code ourselves because the TableAdapters auto-generated code includes the functionality needed to connect to the database, issue commands, retrieve data, and populate that data into DataTables. However, there may be times when we need to customize these low-level settings. Over the next few steps we will examine how to tap into the ADO.NET objects used internally by the TableAdapters.

Step 1: Examining with the Connection Property

Each TableAdapter class has a `Connection` property that specifies database connection information. This property s data type and `ConnectionString` value are determined by the selections made in the TableAdapter Configuration wizard. Recall that when we first add a TableAdapter to a Typed DataSet this wizard asks us for the database source (see Figure 1). The drop-down list in this first step includes those databases specified in the configuration file as well as any other databases in the Server Explorer s Data Connections. If the database we want to use does not exist in the drop-down list, a new database connection can be specified by clicking the New Connection button and providing the needed connection information.

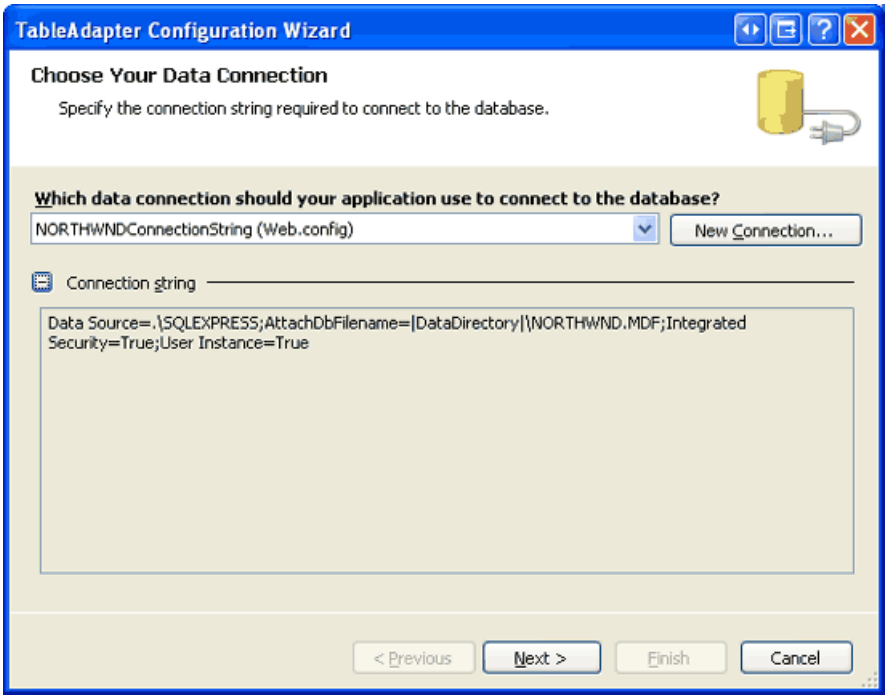


Figure 1: The First Step of the TableAdapter Configuration Wizard (Click to view full-size image)

Let s take a moment to inspect the code for the TableAdapter s `Connection` property. As noted in the Creating a Data Access Layer tutorial, we can view the auto-generated TableAdapter code by going to the Class View window, drilling down to the appropriate class, and then double-clicking the member name.

Navigate to the Class View window by going to the View menu and choosing Class View (or by typing Ctrl+Shift+C). From the top half of the Class View window, drill down to the `NorthwindTableAdapters` namespace and select the `ProductsTableAdapter` class. This will display the `ProductsTableAdapter` s members in the bottom half of the Class View, as shown in Figure 2. Double-click the `Connection` property to see its code.

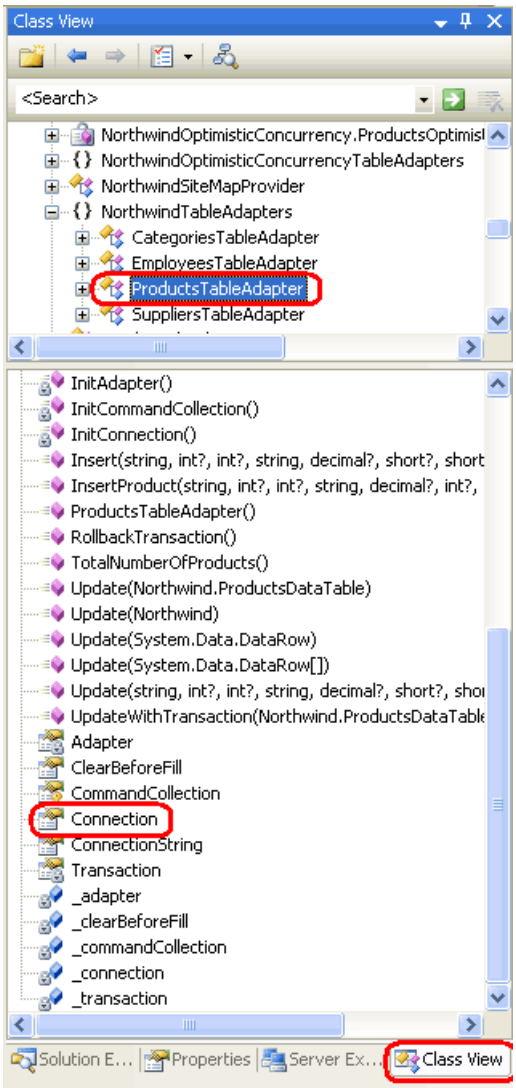


Figure 2: Double-Click the Connection Property in the Class View to View Its Auto-Generated Code

The TableAdapter s `Connection` property and other connection-related code follows:

```
private System.Data.SqlClient.SqlConnection _connection;
private void InitConnection() {
    this._connection = new System.Data.SqlClient.SqlConnection();
    this._connection.ConnectionString =
        ConfigurationManager.ConnectionStrings["NORTHWNDConnectionString"].ConnectionString;
}
internal System.Data.SqlClient.SqlConnection Connection {
    get {
        if ((this._connection == null)) {
            this.InitConnection();
        }
        return this._connection;
    }
    set {
        this._connection = value;
        if ((this.Adapter.InsertCommand != null)) {
            this.Adapter.InsertCommand.Connection = value;
        }
        if ((this.Adapter.DeleteCommand != null)) {
            this.Adapter.DeleteCommand.Connection = value;
        }
        if ((this.Adapter.UpdateCommand != null)) {
            this.Adapter.UpdateCommand.Connection = value;
        }
        for (int i = 0; (i < this.CommandCollection.Length); i = (i + 1)) {
            if ((this.CommandCollection[i] != null)) {
                ((System.Data.SqlClient.SqlCommand)
                    (this.CommandCollection[i])).Connection = value;
            }
        }
    }
}
```

When the TableAdapter class is instantiated, the member variable `_connection` is equal to `null`. When the `Connection` property is accessed, it first checks to see if the `_connection` member variable has been instantiated. If it has not, the `InitConnection` method is invoked, which instantiates `_connection` and sets its `ConnectionString` property to the connection string value specified from the TableAdapter Configuration wizard s first step.

The `Connection` property can also be assigned to a `SqlConnection` object. Doing so associates the new `SqlConnection` object with each of the TableAdapter s `SqlCommand` objects.

Step 2: Exposing Connection-Level Settings

The connection information should remain encapsulated within the TableAdapter and not be accessible to other layers in the application architecture. However, there may be scenarios when the TableAdapter s connection-level information needs to be accessible or customizable for a query, user, or ASP.NET page.

Let s extend the `ProductsTableAdapter` in the `Northwind` DataSet to include a `ConnectionString` property that can be used by the Business Logic Layer to read or change the connection string used by the TableAdapter.

> **NOTE**
>
> A *connection string* is a string that specifies database connection information, such as the provider to use, the location of the database, authentication credentials, and other database-related settings. For a list of connection string patterns used by a variety of data stores and providers, see ConnectionStrings.com.

As discussed in the Creating a Data Access Layer tutorial, the Typed DataSet s auto-generated classes can be extended through the use of partial classes. First, create a new subfolder in the project named `ConnectionAndCommandSettings` underneath the `~/App_Code/DAL` folder.

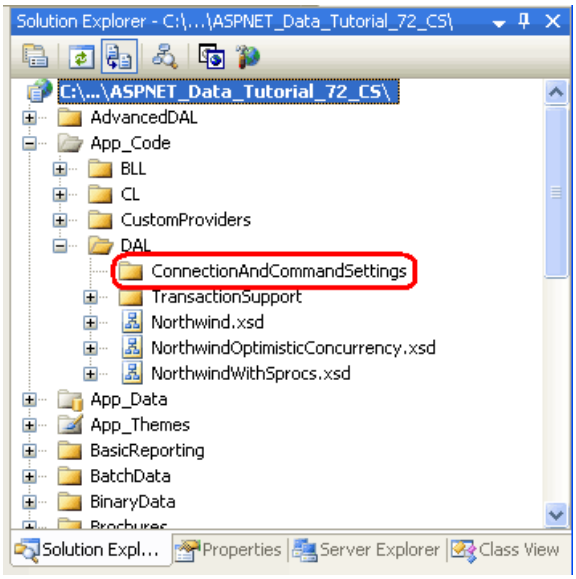


Figure 3: Add a Subfolder Named `ConnectionAndCommandSettings`

Add a new class file named `ProductsTableAdapter.ConnectionAndCommandSettings.cs` and enter the following code:

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
namespace NorthwindTableAdapters
{
    public partial class ProductsTableAdapter
    {
        public string ConnectionString
        {
            get
            {
                return this.Connection.ConnectionString;
            }
            set
            {
                this.Connection.ConnectionString = value;
            }
        }
    }
}
```

This partial class adds a `public` property named `ConnectionString` to the `ProductsTableAdapter` class that allows any layer to read or update the connection string for the TableAdapter s underlying connection.

With this partial class created (and saved), open the `ProductsBLL` class. Go to one of the existing methods and type in `Adapter` and then hit the period key to bring up IntelliSense. You should see the new `ConnectionString` property available in IntelliSense, meaning that you can programmatically read or adjust this value from the BLL.

Exposing the Entire Connection Object

This partial class exposes just one property of the underlying connection object: `ConnectionString`. If you want to make the entire connection object available beyond the confines of the TableAdapter, you can alternatively change the `Connection` property s protection level. The auto-generated code we examined in Step 1 showed that the

TableAdapter s `Connection` property is marked as `internal`, meaning that it can only be accessed by classes in the same assembly. This can be changed, however, via the TableAdapter s `ConnectionModifier` property.

Open the `Northwind` DataSet, click on the `ProductsTableAdapter` in the Designer, and navigate to the Properties window. There you will see the `ConnectionModifier` set to its default value, `Assembly`. To make the `Connection` property available outside of the Typed DataSet s assembly, change the `ConnectionModifier` property to `Public`.

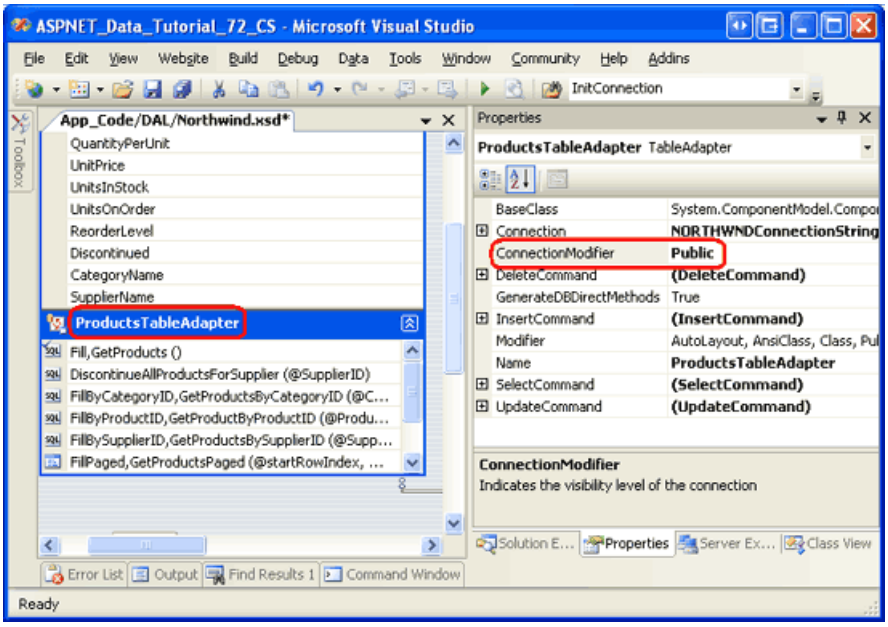


Figure 4: The `Connection` Property s Accessibility Level Can Be Configured via the `ConnectionModifier` Property (Click to view full-size image)

Save the DataSet and then return to the `ProductsBLL` class. As before, go to one of the existing methods and type in `Adapter` and then hit the period key to bring up IntelliSense. The list should include a `Connection` property, meaning that you can now programmatically read or assign any connection-level settings from the BLL.

Step 3: Examining the Command-Related Properties

A TableAdapter consists of a main query that, by default, has auto-generated `INSERT`, `UPDATE`, and `DELETE` statements. This main query s `INSERT`, `UPDATE`, and `DELETE` statements are implemented in the TableAdapter s code as an ADO.NET data adapter object via the `Adapter` property. Like with its `Connection` property, the `Adapter` property s data type is determined by the data provider used. Since these tutorials use the SqlClient provider, the `Adapter` property is of type `SqlDataAdapter`.

The TableAdapter s `Adapter` property has three properties of type `SqlCommand` that it uses to issue the `INSERT`, `UPDATE`, and `DELETE` statements:

- `InsertCommand`
- `UpdateCommand`
- `DeleteCommand`

A `SqlCommand` object is responsible for sending a particular query to the database and has properties like: `CommandText`, which contains the ad-hoc SQL statement or stored procedure to execute; and `Parameters`, which is a collection of `SqlParameter` objects. As we saw back in the Creating a Data Access Layer tutorial, these command objects can be customized through the Properties window.

In addition to its main query, the TableAdapter can include a variable number of methods that, when invoked, dispatch a specified command to the database. The main query s command object and the command objects for all additional methods are stored in the TableAdapter s `CommandCollection` property.

Let s take a moment to look at the code generated by the `ProductsTableAdapter` in the `Northwind` DataSet for these two properties and their supporting member variables and helper methods:

```
private System.Data.SqlClient.SqlDataAdapter _adapter;
private void InitAdapter() {
    this._adapter = new System.Data.SqlClient.SqlDataAdapter();

    ... Code that creates the InsertCommand, UpdateCommand, ...
    ... and DeleteCommand instances - omitted for brevity ...
}
private System.Data.SqlClient.SqlDataAdapter Adapter {
    get {
        if ((this._adapter == null)) {
            this.InitAdapter();
        }
        return this._adapter;
    }
}
private System.Data.SqlClient.SqlCommand[] _commandCollection;
private void InitCommandCollection() {
    this._commandCollection = new System.Data.SqlClient.SqlCommand[9];
    ... Code that creates the command objects for the main query and the ...
    ... ProductsTableAdapter's other eight methods - omitted for brevity ...
}
protected System.Data.SqlClient.SqlCommand[] CommandCollection {
    get {
        if ((this._commandCollection == null)) {
            this.InitCommandCollection();
        }
        return this._commandCollection;
    }
}
```

The code for the `Adapter` and `CommandCollection` properties closely mimics that of the `Connection` property. There are member variables that hold the objects used by the properties. The properties `get` accessors start by checking to see if the corresponding member variable is `null`. If so, an initialization method is called which creates an instance of the member variable and assigns the core command-related properties.

Step 4: Exposing Command-Level Settings

Ideally, the command-level information should remain encapsulated within the Data Access Layer. Should this information be needed in other layers of the architecture, however, it can be exposed through a partial class, just like with the connection-level settings.

Since the TableAdapter only has a single `Connection` property, the code for exposing connection-level settings is fairly straightforward. Things are a bit more complicated when modifying command-level settings because the TableAdapter can have multiple command objects - an `InsertCommand`, `UpdateCommand`, and `DeleteCommand`, along with a variable number of command objects in the `CommandCollection` property. When updating command-level settings, these settings will need to be propagated to all of the command objects.

For example, imagine that there were certain queries in the TableAdapter that took an extraordinary long time to execute. When using the TableAdapter to execute one of those queries, we might want to increase the command object s `CommandTimeout` property. This property specifies the number of seconds to wait for the command to execute and defaults to 30.

To allow the `CommandTimeout` property to be adjusted by the BLL, add the following `public` method to the `ProductsDataTable` using the partial class file created in Step 2 (`ProductsTableAdapter.ConnectionAndCommandSettings.cs`):

```
public void SetCommandTimeout(int timeout)
{
    if (this.Adapter.InsertCommand != null)
        this.Adapter.InsertCommand.CommandTimeout = timeout;
    if (this.Adapter.DeleteCommand != null)
        this.Adapter.DeleteCommand.CommandTimeout = timeout;
    if (this.Adapter.UpdateCommand != null)
        this.Adapter.UpdateCommand.CommandTimeout = timeout;
    for (int i = 0; i < this.CommandCollection.Length; i++)
        if (this.CommandCollection[i] != null)
            this.CommandCollection[i].CommandTimeout = timeout;
}
```

This method could be invoked from the BLL or Presentation Layer to set the command timeout for all commands issues by that TableAdapter instance.

NOTE

The `Adapter` and `CommandCollection` properties are marked as `private`, meaning they can only be accessed from code within the TableAdapter. Unlike the `Connection` property, these access modifiers are not configurable. Therefore, if you need to expose command-level properties to other layers in the architecture you must use the partial class approach discussed above to provide a `public` method or property that reads or writes to the `private` command objects.

Summary

The TableAdapters within a Typed DataSet serve to encapsulate data access details and complexity. Using TableAdapters, we do not have to worry about writing ADO.NET code to connect to the database, issue a command, or populate the results into a DataTable. It is all handled automatically for us.

However, there may be times when we need to customize the low-level ADO.NET specifics, such as changing the connection string or the default connection or command timeout values. The TableAdapter has auto-generated `Connection`, `Adapter`, and `CommandCollection` properties, but these are either `internal` or `private`, by default. This internal information can be exposed by extending the TableAdapter using partial classes to include `public` methods or properties. Alternatively, the TableAdapter s `Connection` property access modifier can be configured through the TableAdapter s `ConnectionModifier` property.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Burnadette Leigh, S ren Jacob Lauritsen, Teresa Murphy, and Hilton Geisenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Protecting Connection Strings and Other Configuration Information (C#)

2/20/2020 • 16 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

An ASP.NET application typically stores configuration information in a Web.config file. Some of this information is sensitive and warrants protection. By default this file will not be served to a Web site visitor, but an administrator or a hacker may gain access to the Web server's file system and view the contents of the file. In this tutorial we learn that ASP.NET 2.0 allows us to protect sensitive information by encrypting sections of the Web.config file.

Introduction

Configuration information for ASP.NET applications is commonly stored in an XML file named `Web.config`. Over the course of these tutorials we have updated the `Web.config` a handful of times. When creating the `Northwind` Typed DataSet in the first tutorial, for example, connection string information was automatically added to `Web.config` in the `<connectionStrings>` section. Later, in the Master Pages and Site Navigation tutorial, we manually updated `Web.config`, adding a `<pages>` element indicating that all of the ASP.NET pages in our project should use the `DataWebControls` Theme.

Since `Web.config` may contain sensitive data such as connection strings, it is important that the contents of `Web.config` be kept safe and hidden from unauthorized viewers. By default, any HTTP request to a file with the `.config` extension is handled by the ASP.NET engine, which returns the *This type of page is not served* message shown in Figure 1. This means that visitors cannot view your `Web.config` file s contents by simply entering http://www.YourServer.com/Web.config into their browser s Address bar.

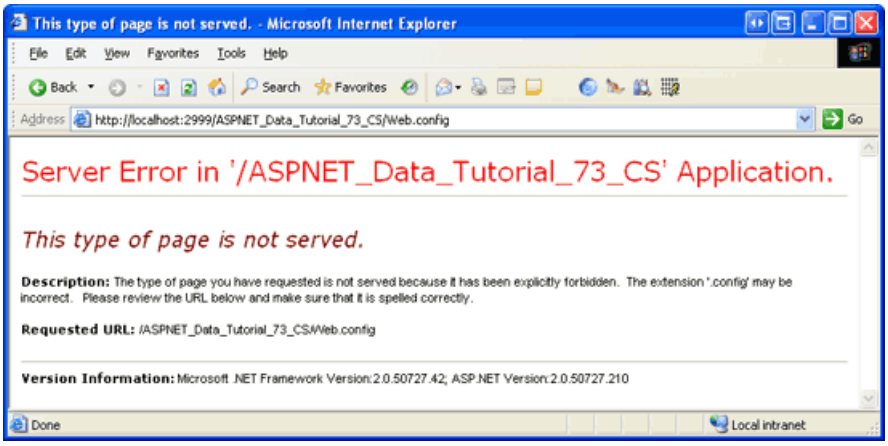


Figure 1: Visiting `Web.config` Through a Browser Returns a This type of page is not served Message (Click to view full-size image)

But what if an attacker is able to find some other exploit that allows her to view your `Web.config` file s contents? What could an attacker do with this information, and what steps can be taken to further protect the sensitive information within `Web.config`? Fortunately, most sections in `Web.config` do not contain sensitive information. What harm can an attacker perpetrate if they know the name of the default Theme used by your ASP.NET pages?

Certain `Web.config` sections, however, contain sensitive information that may include connection strings, user

names, passwords, server names, encryption keys, and so forth. This information is typically found in the following `Web.config` sections:

- `<appSettings>`
- `<connectionStrings>`
- `<identity>`
- `<sessionState>`

In this tutorial we will look at techniques for protecting such sensitive configuration information. As we will see, the .NET Framework version 2.0 includes a protected configurations system that makes programmatically encrypting and decrypting selected configuration sections a breeze.

> **NOTE**
> This tutorial concludes with a look at Microsoft s recommendations for connecting to a database from an ASP.NET application. In addition to encrypting your connection strings, you can help harden your system by ensuring that you are connecting to the database in a secure fashion.

Step 1: Exploring ASP.NET 2.0 s Protected Configuration Options

ASP.NET 2.0 includes a protected configuration system for encrypting and decrypting configuration information. This includes methods in the .NET Framework that can be used to programmatically encrypt or decrypt configuration information. The protected configuration system uses the provider model, which allows developers to choose what cryptographic implementation is used.

The .NET Framework ships with two protected configuration providers:

- `RSAProtectedConfigurationProvider` - uses the asymmetric RSA algorithm for encryption and decryption.
- `DPAPIProtectedConfigurationProvider` - uses the Windows Data Protection API (DPAPI) for encryption and decryption.

Since the protected configuration system implements the provider design pattern, it is possible to create your own protected configuration provider and plug it into your application. See Implementing a Protected Configuration Provider for more information on this process.

The RSA and DPAPI providers use keys for their encryption and decryption routines, and these keys can be stored at the machine- or user-level. Machine-level keys are ideal for scenarios where the web application runs on its own dedicated server or if there are multiple applications on a server that need to share encrypted information. User-level keys are a more secure option in shared hosting environments where other applications on the same server should not be able to decrypt your application s protected configuration sections.

In this tutorial our examples will use the DPAPI provider and machine-level keys. Specifically, we will look at encrypting the `<connectionStrings>` section in `Web.config`, although the protected configuration system can be used to encrypt most any `Web.config` section. For information on using user-level keys or using the RSA provider, consult the resources in the Further Readings section at the end of this tutorial.

NOTE

The `RSAProtectedConfigurationProvider` and `DPAPIProtectedConfigurationProvider` providers are registered in the `machine.config` file with the provider names `RsaProtectedConfigurationProvider` and `DataProtectionConfigurationProvider`, respectively. When encrypting or decrypting configuration information we will need to supply the appropriate provider name (`RsaProtectedConfigurationProvider` or `DataProtectionConfigurationProvider`) rather than the actual type name (`RSAProtectedConfigurationProvider` and `DPAPIProtectedConfigurationProvider`). You can find the `machine.config` file in the `$WINDOWS$\Microsoft.NET\Framework\version\CONFIG` folder.

Step 2: Programmatically Encrypting and Decrypting Configuration Sections

With a few lines of code we can encrypt or decrypt a particular configuration section using a specified provider. The code, as we will see shortly, simply needs to programmatically reference the appropriate configuration section, call its `ProtectSection` or `UnprotectSection` method, and then call the `Save` method to persist the changes. Moreover, the .NET Framework includes a helpful command line utility that can encrypt and decrypt configuration information. We will explore this command line utility in Step 3.

To illustrate programmatically protecting configuration information, let s create an ASP.NET page that includes buttons for encrypting and decrypting the `<connectionStrings>` section in `Web.config`.

Start by opening the `EncryptingConfigSections.aspx` page in the `AdvancedDAL` folder. Drag a TextBox control from the Toolbox onto the Designer, setting its `ID` property to `WebConfigContents`, its `TextMode` property to `MultiLine`, and its `Width` and `Rows` properties to 95% and 15, respectively. This TextBox control will display the contents of `Web.config` allowing us to quickly see if the contents are encrypted or not. Of course, in a real application you would never want to display the contents of `Web.config`.

Beneath the TextBox, add two Button controls named `EncryptConnStrings` and `DecryptConnStrings`. Set their Text properties to Encrypt Connection Strings and Decrypt Connection Strings .

At this point your screen should look similar to Figure 2.

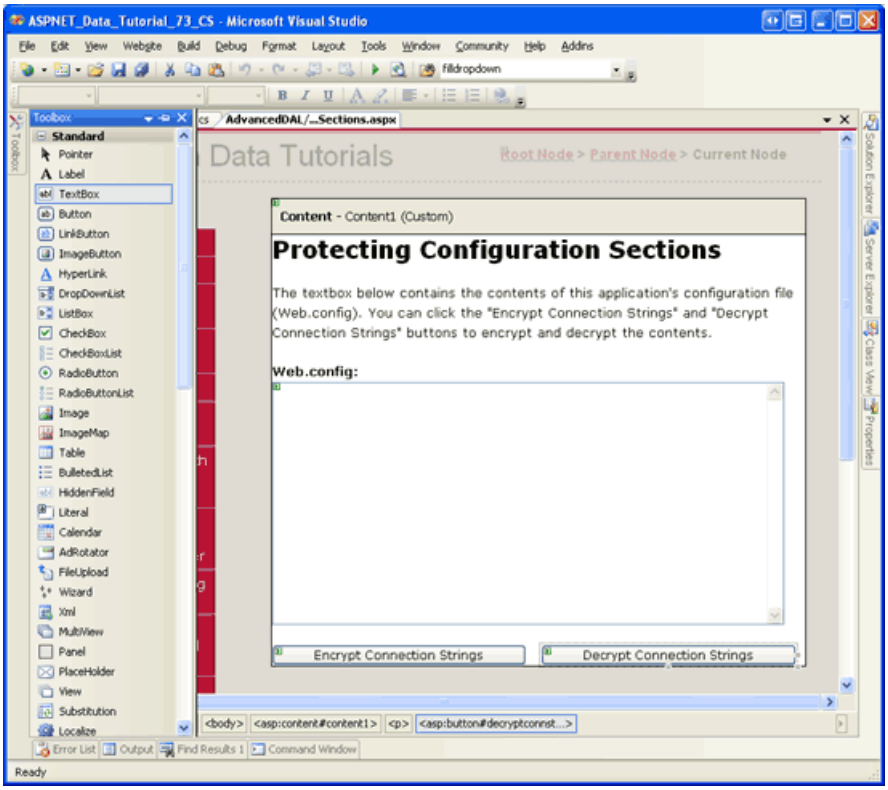


Figure 2: Add a TextBox and Two Button Web Controls to the Page (Click to view full-size image)

Next, we need to write code that loads and displays the contents of `Web.config` in the `WebConfigContents` TextBox when the page is first loaded. Add the following code to the page s code-behind class. This code adds a method named `DisplayWebConfig` and calls it from the `Page_Load` event handler when `Page.IsPostBack` is `false`:

```
protected void Page_Load(object sender, EventArgs e)
{
    // On the first page visit, call DisplayWebConfig method
    if (!Page.IsPostBack)
        DisplayWebConfig();
}
private void DisplayWebConfig()
{
    // Reads in the contents of Web.config and displays them in the TextBox
    StreamReader webConfigStream =
        File.OpenText(Path.Combine(Request.PhysicalApplicationPath, "Web.config"));
    string configContents = webConfigStream.ReadToEnd();
    webConfigStream.Close();
    WebConfigContents.Text = configContents;
}
```

The `DisplayWebConfig` method uses the `File` class to open the application s `Web.config` file, the `StreamReader` class to read its contents into a string, and the `Path` class to generate the physical path to the `Web.config` file. These three classes are all found in the `System.IO` namespace. Consequently, you will need to add a `using` `System.IO` statement to the top of the code-behind class or, alternatively, prefix these class names with `System.IO.` .

Next, we need to add event handlers for the two Button controls `Click` events and add the necessary code to encrypt and decrypt the `<connectionStrings>` section using a machine-level key with the DPAPI provider. From the Designer, double-click each of the Buttons to add a `Click` event handler in the code-behind class and then add the following code:

```
protected void EncryptConnStrings_Click(object sender, EventArgs e)
{
    // Get configuration information about Web.config
    Configuration config =
        WebConfigurationManager.OpenWebConfiguration(Request.ApplicationPath);
    // Let's work with the <connectionStrings> section
    ConfigurationSection connectionStrings = config.GetSection("connectionStrings");
    if (connectionStrings != null)
        // Only encrypt the section if it is not already protected
        if (!connectionStrings.SectionInformation.IsProtected)
        {
            // Encrypt the <connectionStrings> section using the
            // DataProtectionConfigurationProvider provider
            connectionStrings.SectionInformation.ProtectSection(
                "DataProtectionConfigurationProvider");
            config.Save();

            // Refresh the Web.config display
            DisplayWebConfig();
        }
}
protected void DecryptConnStrings_Click(object sender, EventArgs e)
{
    // Get configuration information about Web.config
    Configuration config =
        WebConfigurationManager.OpenWebConfiguration(Request.ApplicationPath);
    // Let's work with the <connectionStrings> section
    ConfigurationSection connectionStrings =
        config.GetSection("connectionStrings");
    if (connectionStrings != null)
        // Only decrypt the section if it is protected
        if (connectionStrings.SectionInformation.IsProtected)
        {
            // Decrypt the <connectionStrings> section
            connectionStrings.SectionInformation.UnprotectSection();
            config.Save();
            // Refresh the Web.config display
            DisplayWebConfig();
        }
}
```

The code used in the two event handlers is nearly identical. They both start by getting information about the current application s `Web.config` file via the `WebConfigurationManager` class s `OpenWebConfiguration` method. This method returns the web configuration file for the specified virtual path. Next, the `Web.config` file s `<connectionStrings>` section is accessed via the `Configuration` class s `GetSection(sectionName)` method, which returns a `ConfigurationSection` object.

The `ConfigurationSection` object includes a `SectionInformation` property that provides additional information and functionality regarding the configuration section. As the code above shows, we can determine whether the configuration section is encrypted by checking the `SectionInformation` property s `IsProtected` property. Moreover, the section can be encrypted or decrypted via the `SectionInformation` property s `ProtectSection(provider)` and `UnprotectSection` methods.

The `ProtectSection(provider)` method accepts as input a string specifying the name of the protected configuration provider to use when encrypting. In the `EncryptConnString` Button s event handler we pass DataProtectionConfigurationProvider into the `ProtectSection(provider)` method so that the DPAPI provider is used. The `UnprotectSection` method can determine the provider that was used to encrypt the configuration section and therefore does not require any input parameters.

After calling the `ProtectSection(provider)` or `UnprotectSection` method, you must call the `Configuration` object s `Save` method to persist the changes. Once the configuration information has been encrypted or decrypted and

the changes saved, we call `DisplayWebConfig` to load the updated `Web.config` contents into the TextBox control.

Once you have entered the above code, test it by visiting the `EncryptingConfigSections.aspx` page through a browser. You should initially see a page that lists the contents of `Web.config` with the `<connectionStrings>` section displayed in plain-text (see Figure 3).

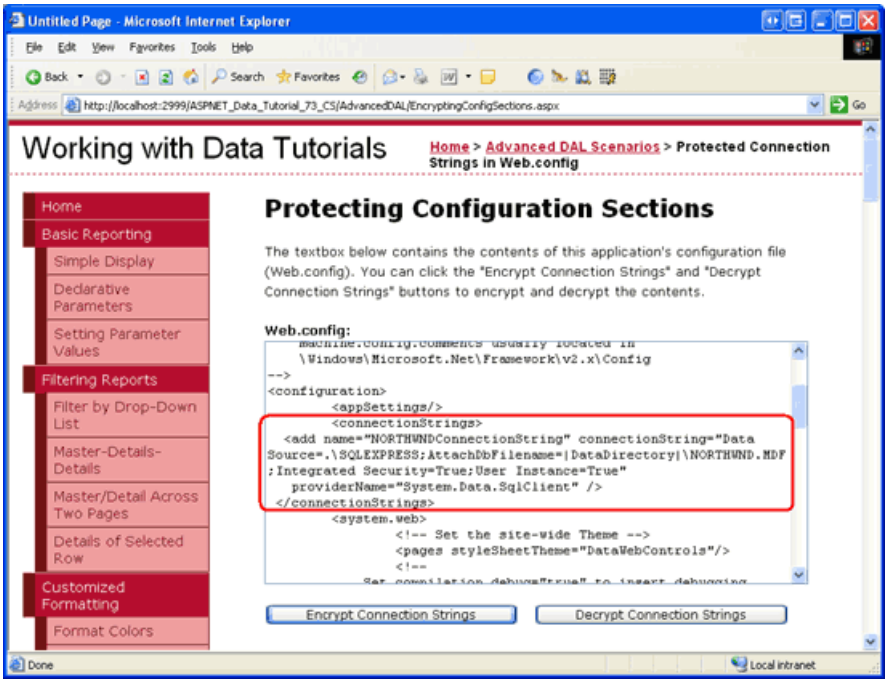


Figure 3: Add a TextBox and Two Button Web Controls to the Page (Click to view full-size image)

Now click the Encrypt Connection Strings button. If request validation is enabled, the markup posted back from the `WebConfigContents` TextBox will produce an `HttpRequestValidationException`, which displays the message, A potentially dangerous `Request.Form` value was detected from the client. Request validation, which is enabled by default in ASP.NET 2.0, prohibits postbacks that include un-encoded HTML and is designed to help prevent script-injection attacks. This check can be disabled at the page- or application-level. To turn it off for this page, set the `ValidateRequest` setting to `false` in the `@Page` directive. The `@Page` directive is found at the top of the page s declarative markup.

```
<%@ Page ValidateRequest="False" ... %>
```

For more information on request validation, its purpose, how to disable it at the page- and application-level, as well as how to HTML encode markup, see Request Validation - Preventing Script Attacks.

After disabling request validation for the page, try clicking the Encrypt Connection Strings button again. On postback, the configuration file will be accessed and its `<connectionStrings>` section encrypted using the DPAPI provider. The TextBox is then updated to display the new `Web.config` content. As Figure 4 shows, the `<connectionStrings>` information is now encrypted.

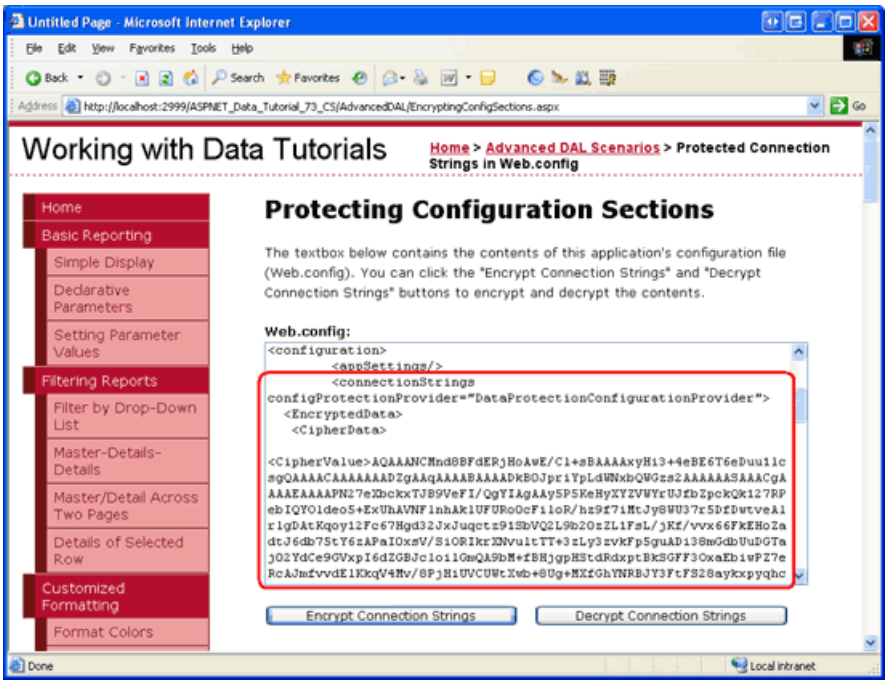


Figure 4: Clicking the Encrypt Connection Strings Button Encrypts the `<connectionString>` Section (Click to view full-size image)

The encrypted `<connectionStrings>` section generated on my computer follows, although some of the content in the `<CipherData>` element has been removed for brevity:

```
<connectionStrings
    configProtectionProvider="DataProtectionConfigurationProvider">
  <EncryptedData>
    <CipherData>
      <CipherValue>AQAAANCMnd8BFdERjHoAwE/...zChw==</CipherValue>
    </CipherData>
  </EncryptedData>
</connectionStrings>
```

NOTE

The `<connectionStrings>` element specifies the provider used to perform the encryption (`DataProtectionConfigurationProvider`). This information is used by the `UnprotectSection` method when the Decrypt Connection Strings button is clicked.

When the connection string information is accessed from `Web.config` - either by code we write, from a SqlDataSource control, or the auto-generated code from the TableAdapters in our Typed DataSets - it is automatically decrypted. In short, we do not need to add any extra code or logic to decrypt the encrypted `<connectionString>` section. To demonstrate this, visit one of the earlier tutorials at this time, such as the Simple Display tutorial from the Basic Reporting section (`~/BasicReporting/SimpleDisplay.aspx`). As Figure 5 shows, the tutorial works exactly as we would expect it, indicating that the encrypted connection string information is being automatically decrypted by the ASP.NET page.

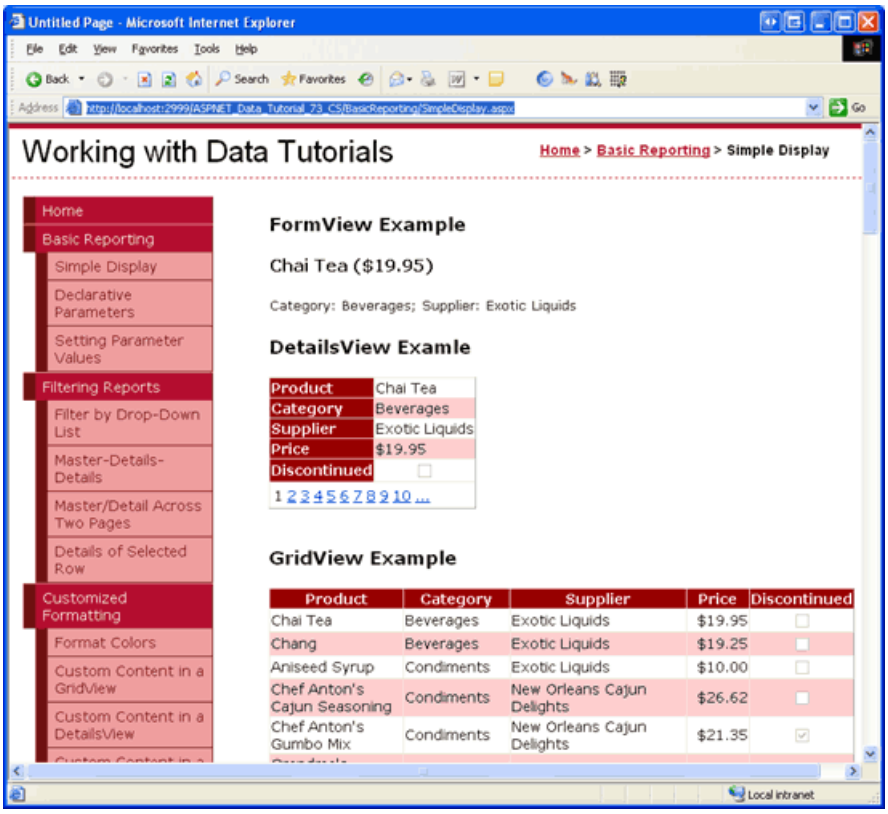


Figure 5: The Data Access Layer Automatically Decrypts the Connection String Information (Click to view full-size image)

To revert the `<connectionStrings>` section back to its plain-text representation, click the Decrypt Connection Strings button. On postback you should see the connection strings in `Web.config` in plain-text. At this point your screen should look like it did when first visiting this page (see in Figure 3).

Step 3: Encrypting Configuration Sections Using `aspnet_regiis.exe`

The .NET Framework includes a variety of command line tools in the `$WINDOWS$\Microsoft.NET\Framework\version\` folder. In the Using SQL Cache Dependencies tutorial, for instance, we looked at using the `aspnet_regsql.exe` command line tool to add the infrastructure necessary for SQL cache dependencies. Another useful command line tool in this folder is the ASP.NET IIS Registration tool (`aspnet_regiis.exe`). As its name implies, the ASP.NET IIS Registration tool is primarily used to register an ASP.NET 2.0 application with Microsoft s professional-grade Web server, IIS. In addition to its IIS-related features, the ASP.NET IIS Registration tool can also be used to encrypt or decrypt specified configuration sections in `Web.config`.

The following statement shows the general syntax used to encrypt a configuration section with the `aspnet_regiis.exe` command line tool:

```
aspnet_regiis.exe -pef section physical_directory -prov provider
```

section is the configuration section to encrypt (like connectionStrings), the *physical_directory* is the full, physical path to the web application s root directory, and *provider* is the name of the protected configuration provider to use (such as DataProtectionConfigurationProvider). Alternatively, if the web application is registered in IIS you can enter the virtual path instead of the physical path using the following syntax:

```
aspnet_regiis.exe -pe section -app virtual_directory -prov provider
```

The following `aspnet_regiis.exe` example encrypts the `<connectionStrings>` section using the DPAPI provider with a machine-level key:

```
aspnet_regiis.exe -pef
"connectionStrings" "C:\Websites\ASPNET_Data_Tutorial_73_CS"
-prov "DataProtectionConfigurationProvider"
```

Similarly, the `aspnet_regiis.exe` command line tool can be used to decrypt configuration sections. Instead of using the `-pef` switch, use `-pdf` (or instead of `-pe`, use `-pd`). Also, note that the provider name is not necessary when decrypting.

```
aspnet_regiis.exe -pdf section physical_directory
  -- or --
aspnet_regiis.exe -pd section -app virtual_directory
```

NOTE

Since we are using the DPAPI provider, which uses keys specific to the computer, you must run `aspnet_regiis.exe` from the same machine from which the web pages are being served. For example, if you run this command line program from your local development machine and then upload the encrypted Web.config file to the production server, the production server will not be able to decrypt the connection string information since it was encrypted using keys specific to your development machine. The RSA provider does not have this limitation as it is possible to export the RSA keys to another machine.

Understanding Database Authentication Options

Before any application can issue `SELECT`, `INSERT`, `UPDATE`, or `DELETE` queries to a Microsoft SQL Server database, the database first must identify the requestor. This process is known as *authentication* and SQL Server provides two methods of authentication:

- **Windows Authentication** - the process under which the application is running is used to communicate with the database. When running an ASP.NET application through Visual Studio 2005 s ASP.NET Development Server, the ASP.NET application assumes the identity of the currently logged on user. For ASP.NET applications on Microsoft Internet Information Server (IIS), ASP.NET applications usually assume the identity of `domainName``\MachineName` or `domainName``\NETWORK SERVICE`, although this can be customized.
- **SQL Authentication** - a user ID and password values are supplied as credentials for authentication. With SQL authentication, the user ID and password are provided in the connection string.

Windows authentication is preferred over SQL authentication because it is more secure. With Windows authentication the connection string is free from a username and password and if the web server and database server reside on two different machines, the credentials are not sent over the network in plain-text. With SQL authentication, however, the authentication credentials are hard-coded in the connection string and are transmitted from the web server to the database server in plain-text.

These tutorials have used Windows authentication. You can tell what authentication mode is being used by inspecting the connection string. The connection string in `Web.config` for our tutorials has been:

```
Data Source=.\SQLEXPRESS; AttachDbFilename=|DataDirectory|\NORTHWND.MDF; Integrated Security=True; User
Instance=True
```

The Integrated Security=True and lack of a username and password indicate that Windows authentication is being used. In some connection strings the term Trusted Connection=Yes or Integrated Security=SSPI is used instead of Integrated Security=True, but all three indicate the use of Windows authentication.

The following example shows a connection string that uses SQL authentication. Note the credentials embedded within the connection string:

```
Server=serverName; Database=Northwind; uid=userID; pwd=password
```

Imagine that an attacker is able to view your application s `Web.config` file. If you use SQL authentication to connect to a database that is accessible over the Internet, the attacker can use this connection string to connect to your database through SQL Management Studio or from ASP.NET pages on their own website. To help mitigate this threat, encrypt the connection string information in `Web.config` using the protected configuration system.

> **NOTE**
> For more information on the different types of authentication available in SQL Server, see Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication. For further connection string examples illustrating the differences between Windows and SQL authentication syntax, refer to ConnectionStrings.com.

Summary

By default, files with a `.config` extension in an ASP.NET application cannot be accessed through a browser. These types of files are not returned because they may contain sensitive information, such as database connection strings, usernames and passwords, and so on. The protected configuration system in .NET 2.0 helps further protect sensitive information by allowing specified configuration sections to be encrypted. There are two built-in protected configuration providers: one that uses the RSA algorithm and one that uses the Windows Data Protection API (DPAPI).

In this tutorial we looked at how to encrypt and decrypt configuration settings using the DPAPI provider. This can be accomplished both programmatically, as we saw in Step 2, as well as through the `aspnet_regiis.exe` command line tool, which was covered in Step 3. For more information on using user-level keys or using the RSA provider instead, see the resources in the Further Reading section.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Building Secure ASP.NET Application: Authentication, Authorization, and Secure Communication
- Encrypting Configuration Information in ASP.NET 2.0 Applications
- Encrypting `Web.config` Values in ASP.NET 2.0
- How To: Encrypt Configuration Sections in ASP.NET 2.0 Using DPAPI
- How To: Encrypt Configuration Sections in ASP.NET 2.0 Using RSA
- The Configuration API in .NET 2.0
- Windows Data Protection

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Teresa Murphy and Randy Schmidt. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Debugging Stored Procedures (C#)

2/20/2020 • 14 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Visual Studio Professional and Team System editions allow you to set breakpoints and step in to stored procedures within SQL Server, making debugging stored procedures as easy as debugging application code. This tutorial demonstrates direct database debugging and application debugging of stored procedures.

Introduction

Visual Studio provides a rich debugging experience. With a few keystrokes or clicks of the mouse, it s possible to use breakpoints to stop execution of a program and examine its state and control flow. Along with debugging application code, Visual Studio offers support for debugging stored procedures from within SQL Server. Just like breakpoints can be set within the code of an ASP.NET code-behind class or Business Logic Layer class, so too can they be placed within stored procedures.

In this tutorial we will look at stepping into stored procedures from the Server Explorer within Visual Studio as well as how to set breakpoints that are hit when the stored procedure is called from the running ASP.NET application.

> **NOTE**
> Unfortunately, stored procedures can only be stepped into and debugged through the Professional and Team Systems versions of Visual Studio. If you are using Visual Web Developer or the standard version of Visual Studio, you are welcome to read along as we walk through the steps necessary to debug stored procedures, but you will not be able to replicate these steps on your machine.

SQL Server Debugging Concepts

Microsoft SQL Server 2005 was designed to provide integration with the Common Language Runtime (CLR), which is the runtime used by all .NET assemblies. Consequently, SQL Server 2005 supports managed database objects. That is, you can create database objects like stored procedures and User-Defined Functions (UDFs) as methods in a C# class. This enables these stored procedures and UDFs to utilize functionality in the .NET Framework and from your own custom classes. Of course, SQL Server 2005 also provides support for T-SQL database objects.

SQL Server 2005 offers debugging support for both T-SQL and managed database objects. However, these objects can only be debugged through Visual Studio 2005 Professional and Team Systems editions. In this tutorial we will examine debugging T-SQL database objects. The subsequent tutorial looks at debugging managed database objects.

The Overview of T-SQL and CLR Debugging in SQL Server 2005 blog entry from the SQL Server 2005 CLR Integration team highlights the three ways to debug SQL Server 2005 objects from Visual Studio:

- **Direct Database Debugging (DDD)** - from Server Explorer we can step into any T-SQL database object, such as stored procedures and UDFs. We will examine DDD in Step 1.
- **Application Debugging** - we can set breakpoints within a database object and then run our ASP.NET application. When the database object is executed, the breakpoint will be hit and control turned over to the

debugger. Note that with application debugging we cannot step into a database object from application code. We must explicitly set breakpoints in those stored procedures or UDFs where we want the debugger to stop. Application debugging is examined starting in Step 2.

- **Debugging from a SQL Server Project** - Visual Studio Professional and Team Systems editions include a SQL Server Project type that is commonly used to create managed database objects. We will examine using SQL Server Projects and debugging their contents in the next tutorial.

Visual Studio can debug stored procedures on local and remote SQL Server instances. A local SQL Server instance is one that is installed on the same machine as Visual Studio. If the SQL Server database you are using is not located on your development machine, then it is considered a remote instance. For these tutorials we have been using local SQL Server instances. Debugging stored procedures on a remote SQL server instance requires more configuration steps than when debugging stored procedures on a local instance.

If you are using a local SQL Server instance, you can start with Step 1 and work through this tutorial to the end. If you are using a remote SQL Server instance, however, you will first need to ensure that when debugging you are logged to your development machine with a Windows user account that has a SQL Server login on the remote instance. Moreover, both this database login and the database login used to connect to the database from the running ASP.NET application must be members of the `sysadmin` role. See the Debugging T-SQL Database Objects on Remote Instances section at the end of this tutorial for more information on configuring Visual Studio and SQL Server to debug a remote instance.

Finally, understand that debugging support for T-SQL database objects is not as feature rich as debugging support for .NET applications. For example, breakpoint conditions and filters are not supported, only a subset of the debugging windows are available, you cannot use Edit and Continue, the Immediate window is rendered useless, and so forth. See Limitations on Debugger Commands and Features for more information.

Step 1: Directly Stepping Into a Stored Procedure

Visual Studio makes it easy to directly debug a database object. Let s look at how to use the Direct Database Debugging (DDD) feature to step into the `Products_SelectByCategoryID` stored procedure in the Northwind database. As its name implies, `Products_SelectByCategoryID` returns product information for a particular category; it was created in the Using Existing Stored Procedures for the Typed DataSet s TableAdapters tutorial. Start by going to the Server Explorer and expand the Northwind database node. Next, drill down into the Stored Procedures folder, right-click on the `Products_SelectByCategoryID` stored procedure, and choose the Step Into Stored Procedure option from the context menu. This will start the debugger.

Since the `Products_SelectByCategoryID` stored procedure expects a `@CategoryID` input parameter, we are asked to provide this value. Enter 1, which will return information about the beverages.

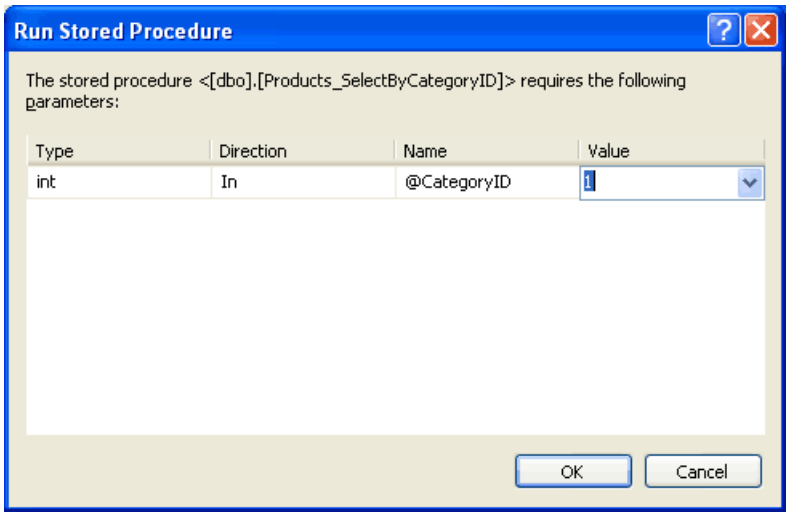


Figure 1: Use the Value 1 for the `@CategoryID` Parameter

After supplying the value for the `@CategoryID` parameter, the stored procedure is executed. Rather than running to completion, though, the debugger halts execution on the first statement. Note the yellow arrow in the margin, indicating the current location in the stored procedure. You can view and edit parameter values through the Watch window or by hovering over the parameter name in the stored procedure.

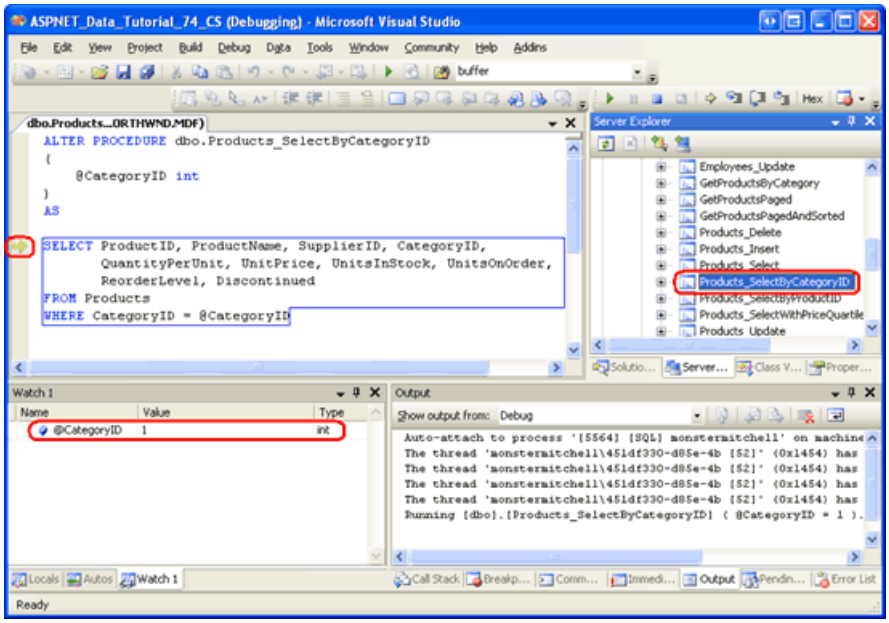


Figure 2: The Debugger has Halted on the First Statement of the Stored Procedure (Click to view full-size image)

To step through the stored procedure one statement at a time, click the Step Over button in the Toolbar or hit the F10 key. The `Products_SelectByCategoryID` stored procedure contains a single `SELECT` statement, so hitting F10 will Step over the single statement and complete the execution of the stored procedure. After the stored procedure has completed, its output will appear in the Output window and the debugger will terminate.

> **NOTE**
>
> T-SQL debugging occurs at the statement level; you cannot Step into a `SELECT` statement.

Step 2: Configuring the Website for Application Debugging

While debugging a stored procedure directly from the Server Explorer is handy, in many scenarios we are more interested in debugging the stored procedure when it is called from our ASP.NET application. We can add breakpoints to a stored procedure from within Visual Studio and then start debugging the ASP.NET application. When a stored procedure with breakpoints is invoked from the application, execution will halt at the breakpoint and we can view and change the stored procedure s parameter values and step through its statements, just like we did in Step 1.

Before we can start debugging stored procedures called from the application, we need to instruct the ASP.NET web application to integrate with the SQL Server debugger. Start by right-clicking on the website name in the Solution Explorer (`ASPNET_Data_Tutorial_74_CS`). Choose the Property Pages option from the context menu, select the Start Options item on the left, and check the SQL Server checkbox in the Debuggers section (see Figure 3).

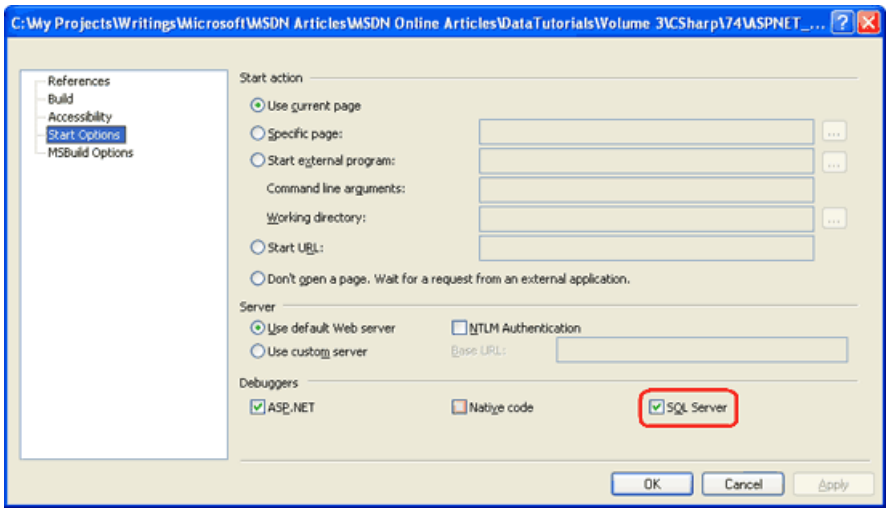


Figure 3: Check the SQL Server Checkbox in the Application s Property Pages (Click to view full-size image)

Additionally, we need to update the database connection string used by the application so that connection pooling is disabled. When a connection to a database is closed, the corresponding `SqlConnection` object is placed in a pool of available connections. When establishing a connection to a database, an available connection object can be retrieved from this pool rather than having to create and establish a new connection. This pooling of connection objects is a performance enhancement and is enabled by default. However, when debugging we want to turn off connection pooling because the debugging infrastructure is not correctly reestablished when working with a connection that was taken from the pool.

To disabled connection pooling, update the `NORTHWNDConnectionString` in `Web.config` so that it includes the setting `Pooling=false` .

```
<connectionStrings>
    <add name="NORTHWNDConnectionString" connectionString=
        "Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\NORTHWND.MDF;
            Integrated Security=True;User Instance=True;Pooling=false"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

NOTE

Once you have finished debugging SQL Server through the ASP.NET application be sure to reinstate connection pooling by removing the `Pooling` setting from the connection string (or by setting it to `Pooling=true`).

At this point the ASP.NET application has been configured to allow Visual Studio to debug SQL Server database objects when invoked through the web application. All that remains now is to add a breakpoint to a stored procedure and start debugging!

Step 3: Adding a Breakpoint and Debugging

Open the `Products_SelectByCategoryID` stored procedure and set a breakpoint at the start of the `SELECT` statement by clicking in the margin at the appropriate place or by placing your cursor at the start of the `SELECT` statement and hitting F9. As Figure 4 illustrates, the breakpoint appears as a red circle in the margin.

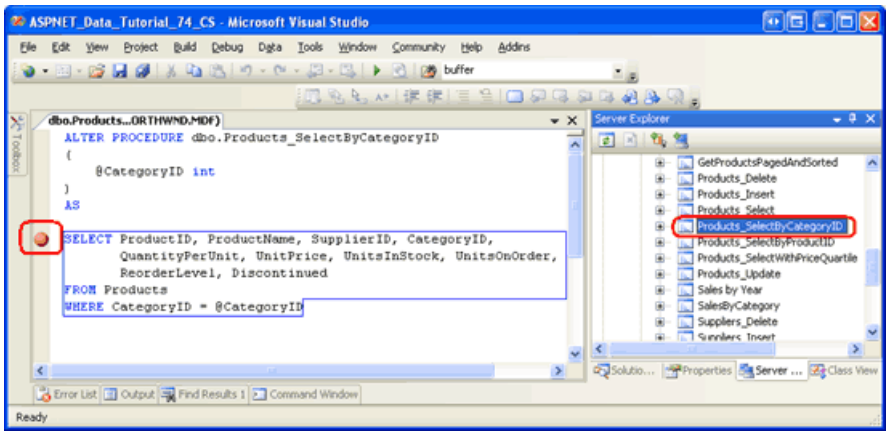


Figure 4: Set a Breakpoint in the `Products_SelectByCategoryID` Stored Procedure (Click to view full-size image)

In order for a SQL database object to be debugged through a client application, it is imperative that the database be configured to support application debugging. When you first set a breakpoint, this setting should automatically be switched on, but it is prudent to double-check. Right-click on the `NORTHWND.MDF` node in the Server Explorer. The context menu should include a checked menu item named Application Debugging .

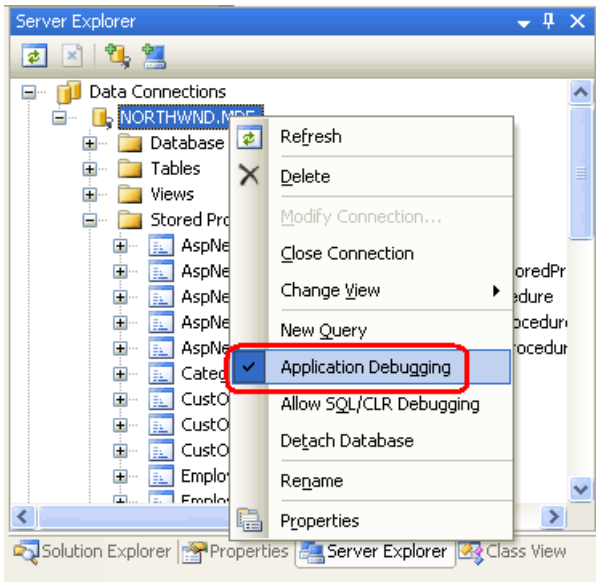


Figure 5: Ensure that the Application Debugging Option is Enabled

With the breakpoint set and the Application Debugging option enabled, we are ready to debug the stored procedure when called from the ASP.NET application. Start the debugger by going to the Debug menu and choosing Start Debugging, by hitting F5, or by clicking the green play icon in the Toolbar. This will start the debugger and launch the website.

The `Products_SelectByCategoryID` stored procedure was created in the Using Existing Stored Procedures for the Typed DataSet s TableAdapters tutorial. Its corresponding web page (`~/AdvancedDAL/ExistingSprocs.aspx`) contains a GridView that displays the results returned by this stored procedure. Visit this page through the browser. Upon reaching the page, the breakpoint in the `Products_SelectByCategoryID` stored procedure will be hit and control returned to Visual Studio. Just like in Step 1, you can step through the stored procedure s statements and view and modify the parameter values.

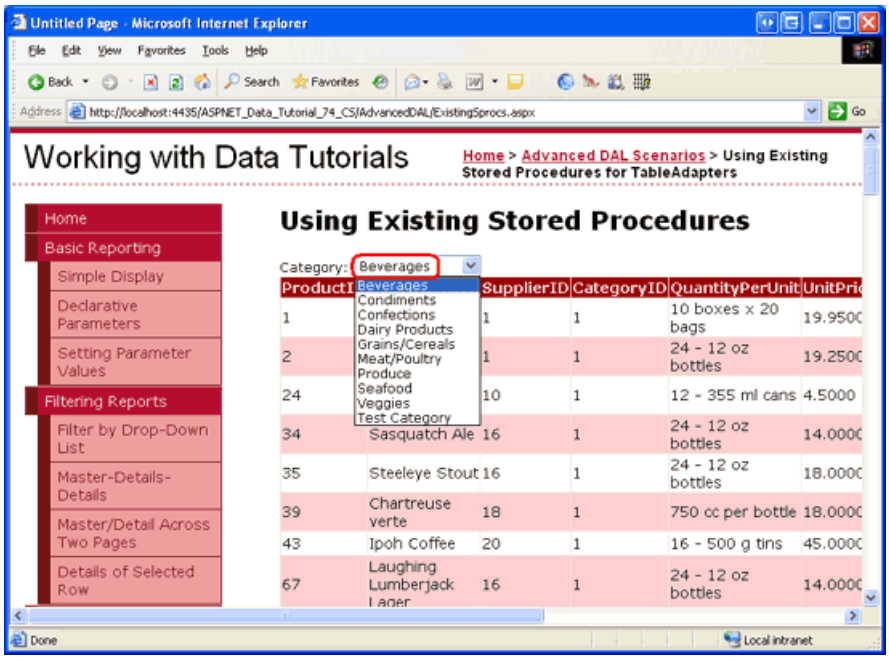


Figure 6: The `ExistingSprocs.aspx` Page Initially Displays the Beverages (Click to view full-size image)

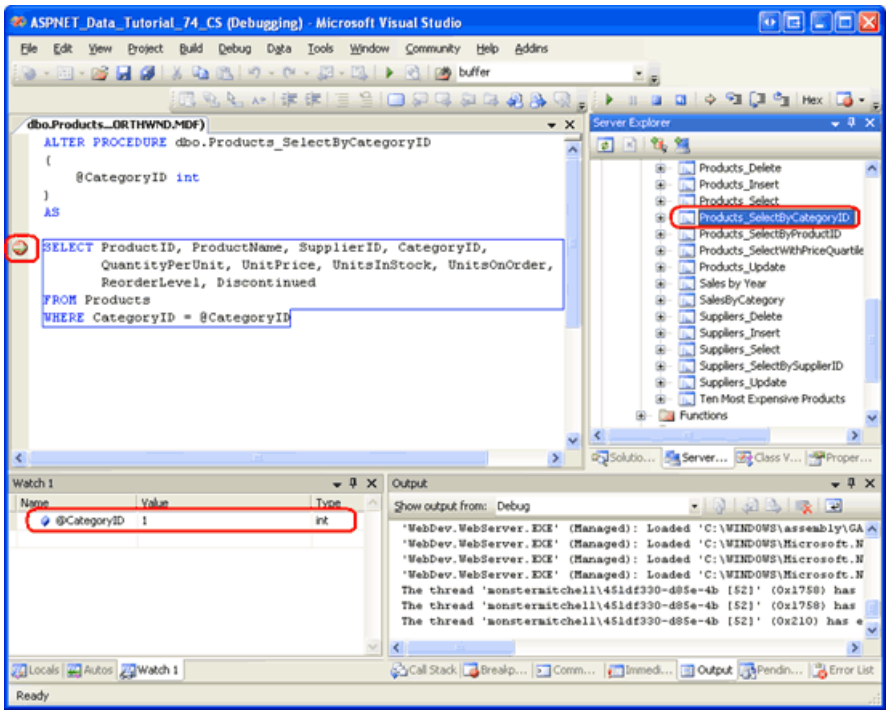


Figure 7: The Stored Procedure s Breakpoint has been Reached (Click to view full-size image)

As the Watch window in Figure 7 shows, the value of the `@CategoryID` parameter is 1. This is because the `ExistingSprocs.aspx` page initially displays products in the beverages category, which has a `CategoryID` value of 1. Choose a different category from the drop-down list. Doing so causes a postback and re-executes the `Products_SelectByCategoryID` stored procedure. The breakpoint is hit again, but this time the `@CategoryID` parameter s value reflects the selected drop-down list item s `CategoryID`.

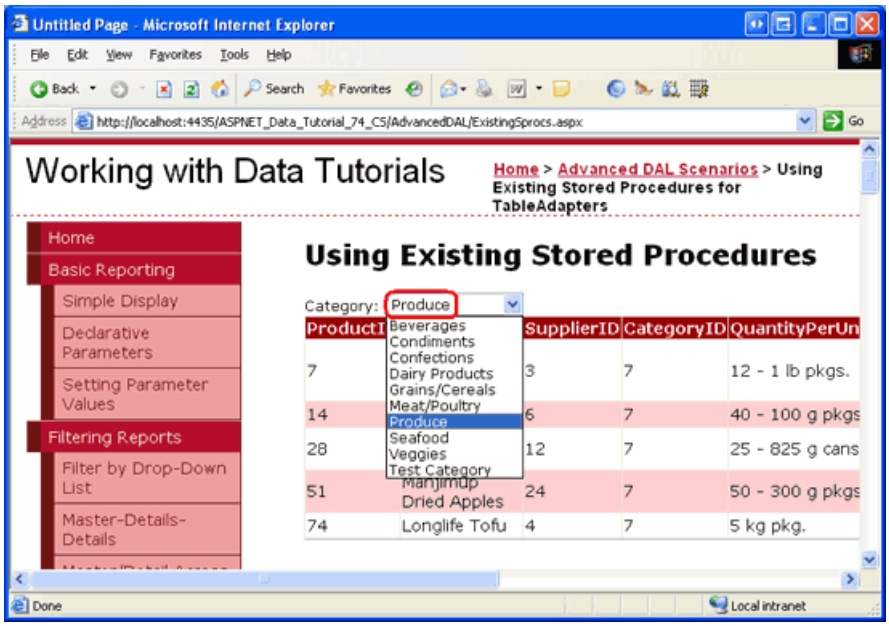


Figure 8: Choose a Different Category from the Drop-Down List (Click to view full-size image)

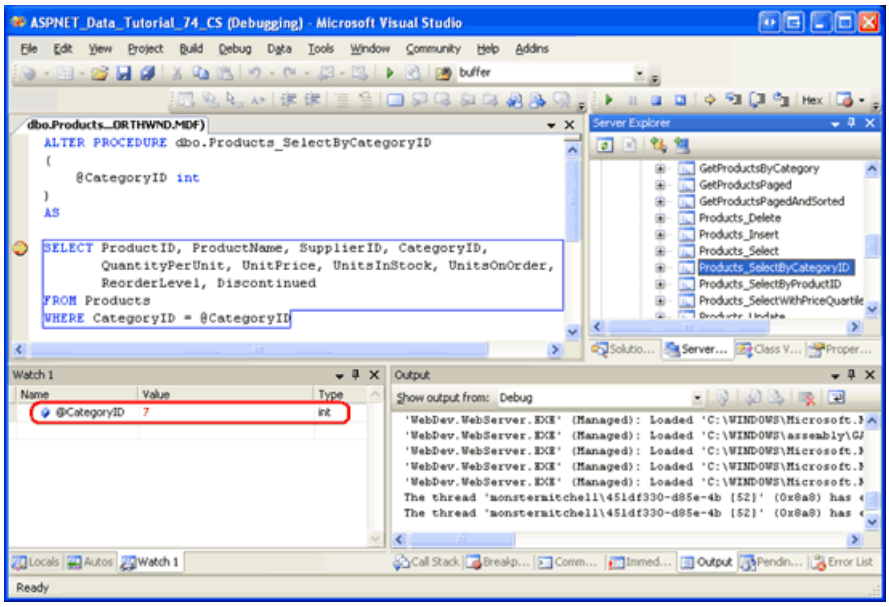


Figure 9: The `@CategoryID` Parameter Reflects the Category Selected from the Web Page (Click to view full-size image)

> **NOTE**
>
> If the breakpoint in the `Products_SelectByCategoryID` stored procedure is not hit when visiting the `ExistingSprocs.aspx` page, make sure that the SQL Server checkbox has been checked in the Debuggers section of the ASP.NET application s Properties Page, that connection pooling has been disabled, and that the database s Application Debugging option is enabled. If you re still having problems, restart Visual Studio and try again.

Debugging T-SQL Database Objects on Remote Instances

Debugging database objects through Visual Studio is fairly straightforward when the SQL Server database instance is on the same machine as Visual Studio. However, if SQL Server and Visual Studio reside on different machines then some careful configuration is required to get everything working properly. There are two core tasks we are faced with:

- Ensure that the login used to connect to the database via ADO.NET belongs to the `sysadmin` role.
- Ensure that the Windows user account used by Visual Studio on the development machine is a valid SQL

Server login account that belongs to the `sysadmin` role.

The first step is relatively straightforward. First, identify the user account used to connect to the database from the ASP.NET application and then, from SQL Server Management Studio, add that login account to the `sysadmin` role.

The second task requires that the Windows user account you use to debug the application be a valid login on the remote database. However, chances are the Windows account you logged on to your workstation with is not a valid login on SQL Server. Rather than adding your particular login account to SQL Server, a better choice would be to designate some Windows user account as the SQL Server debugging account. Then, to debug the database objects of a remote SQL Server instance, you would run Visual Studio using that Windows login account s credentials.

An example should help clarify things. Imagine that there is a Windows account named `SQLDebug` within the Windows domain. This account would need to be added to the remote SQL Server instance as a valid login and as a member of the `sysadmin` role. Then, to debug the remote SQL Server instance from Visual Studio, we would need to run Visual Studio as the `SQLDebug` user. This could be done by logging out of our workstation, logging back in as `SQLDebug`, and then launching Visual Studio, but a simpler approach would be to login to our workstation using our own credentials and then use `runas.exe` to launch Visual Studio as the `SQLDebug` user. `runas.exe` allows a particular application to be executed under the guise of a different user account. To launch Visual Studio as `SQLDebug`, you could enter the following statement at the command line:

```
runas.exe /user:SQLDebug "%PROGRAMFILES%\Microsoft Visual Studio 8\Common7\IDE\devenv.exe"
```

For a more detailed explanation on this process, see William R. Vaughn s *Hitchhiker s Guide to Visual Studio and SQL Server, Seventh Edition* as well as How To: Set SQL Server Permissions for Debugging.

> **NOTE**
> If your development machine is running Windows XP Service Pack 2 you will need to configure the Internet Connection Firewall to allow remote debugging. The How To: Enable SQL Server 2005 Debugging article notes that this involves two steps: (a) On the Visual Studio host machine, you must add `Devenv.exe` to the Exceptions list and open the TCP 135 port; and (b) On the remote (SQL) machine, you must open the TCP 135 port and add `sqlservr.exe` to the Exceptions list. If your domain policy requires network communication to be done through IPSec, you must open the UDP 4500 and UDP 500 ports.

Summary

In addition to providing debugging support for .NET application code, Visual Studio also provides a variety of debugging options for SQL Server 2005. In this tutorial we looked at two of those options: Direct Database Debugging and application debugging. To directly debug a T-SQL database object, locate the object through the Server Explorer then right-click on it and choose Step Into. This starts the debugger and halts on the first statement in the database object, at which point you can step through the object s statements and view and modify parameter values. In Step 1 we used this approach to step into the `Products_SelectByCategoryID` stored procedure.

Application debugging allows breakpoints to be set directly within the database objects. When a database object with breakpoints is invoked from a client application (such as an ASP.NET web application), the program halts as the debugger takes over. Application debugging is useful because it more clearly shows what application action causes a particular database object to be invoked. However, it requires a bit more configuration and setup than Direct Database Debugging.

Database objects can also be debugged through SQL Server Projects. We will look at using SQL Server Projects

and how to use them to create and debug managed database objects in the next tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Creating Stored Procedures and User-Defined Functions with Managed Code (C#)

9/16/2020 • 36 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Microsoft SQL Server 2005 integrates with the .NET Common Language Runtime to allow developers to create database objects through managed code. This tutorial shows how to create managed stored procedures and managed user-defined functions with your Visual Basic or C# code. We also see how these editions of Visual Studio allow you to debug such managed database objects.

Introduction

Databases like Microsoft s SQL Server 2005 use the Transact-Structured Query Language (T-SQL) for inserting, modifying, and retrieving data. Most database systems include constructs for grouping a series of SQL statements that can then be executed as a single, reusable unit. Stored procedures are one example. Another is *User-Defined Functions*(UDFs), a construct that we will examine in greater detail in Step 9.

At its core, SQL is designed for working with sets of data. The `SELECT`, `UPDATE`, and `DELETE` statements inherently apply to all records in the corresponding table and are only limited by their `WHERE` clauses. Yet there are many language features designed for working with one record at a time and for manipulating scalar data. `CURSOR` s allow for a set of records to be looped through one at a time. String manipulation functions like `LEFT`, `CHARINDEX`, and `PATINDEX` work with scalar data. SQL also includes control flow statements like `IF` and `WHILE`.

Prior to Microsoft SQL Server 2005, stored procedures and UDFs could only be defined as a collection of T-SQL statements. SQL Server 2005, however, was designed to provide integration with the Common Language Runtime (CLR), which is the runtime used by all .NET assemblies. Consequently, the stored procedures and UDFs in a SQL Server 2005 database can be created using managed code. That is, you can create a stored procedure or UDF as a method in a C# class. This enables these stored procedures and UDFs to utilize functionality in the .NET Framework and from your own custom classes.

In this tutorial we will examine how to create managed stored procedures and User-Defined Functions and how to integrate them into our Northwind database. Let s get started!

> **NOTE**
> Managed database objects offer some advantages over their SQL counterparts. Language richness and familiarity and the ability to reuse existing code and logic are the main advantages. But managed database objects are likely to be less efficient when working with sets of data that do not involve much procedural logic. For a more thorough discussion on the advantages of using managed code versus T-SQL, check out the Advantages of Using Managed Code to Create Database Objects.

Step 1: Moving the Northwind Database Out of `App_Data`

All of our tutorials thus far have used a Microsoft SQL Server 2005 Express Edition database file in the web application s `App_Data` folder. Placing the database in `App_Data` simplified distributing and running these tutorials as all of the files were located within one directory and required no additional configuration steps to test the

tutorial.

For this tutorial, however, let s move the Northwind database out of `App_Data` and explicitly register it with the SQL Server 2005 Express Edition database instance. While we can perform the steps for this tutorial with the database in the `App_Data` folder, a number of the steps are made much simpler by explicitly registering the database with the SQL Server 2005 Express Edition database instance.

The download for this tutorial has the two database files - `NORTHWND.MDF` and `NORTHWND_log.LDF` - placed in a folder named `DataFiles`. If you are following along with your own implementation of the tutorials, close Visual Studio and move the `NORTHWND.MDF` and `NORTHWND_log.LDF` files from the website s `App_Data` folder to a folder outside of the website. Once the database files have been moved to another folder we need to register the Northwind database with the SQL Server 2005 Express Edition database instance. This can be done from SQL Server Management Studio. If you have a non-Express Edition of SQL Server 2005 installed on your computer then you likely already have Management Studio installed. If you only have SQL Server 2005 Express Edition on your computer then take a moment to download and install Microsoft SQL Server Management Studio Express.

Launch SQL Server Management Studio. As Figure 1 shows, Management Studio starts by asking what server to connect to. Enter localhost\SQLExpress for the server name, choose Windows Authentication in the Authentication drop-down list, and click Connect.

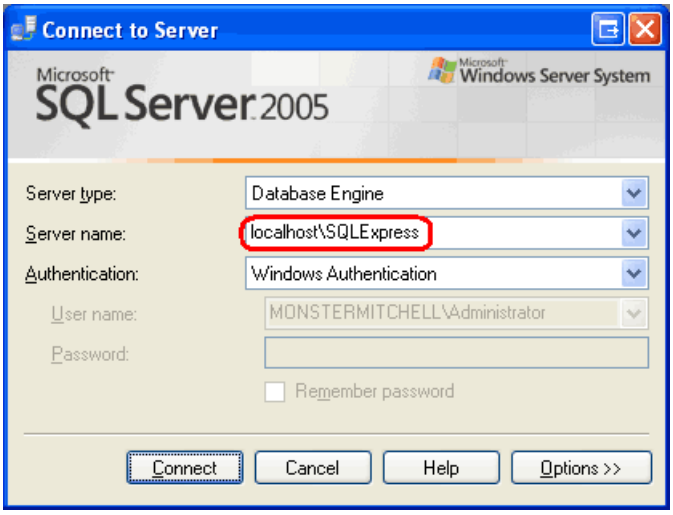


Figure 1: Connect to the Appropriate Database Instance

Once you ve connected, the Object Explorer window will list information about the SQL Server 2005 Express Edition database instance, including its databases, security information, management options, and so forth.

We need to attach the Northwind database in the `DataFiles` folder (or wherever you may have moved it) to the SQL Server 2005 Express Edition database instance. Right-click on the Databases folder and choose the Attach option from the context menu. This will bring up the Attach Databases dialog box. Click the Add button, drill down to the appropriate `NORTHWND.MDF` file, and click OK. At this point your screen should look similar to Figure 2.

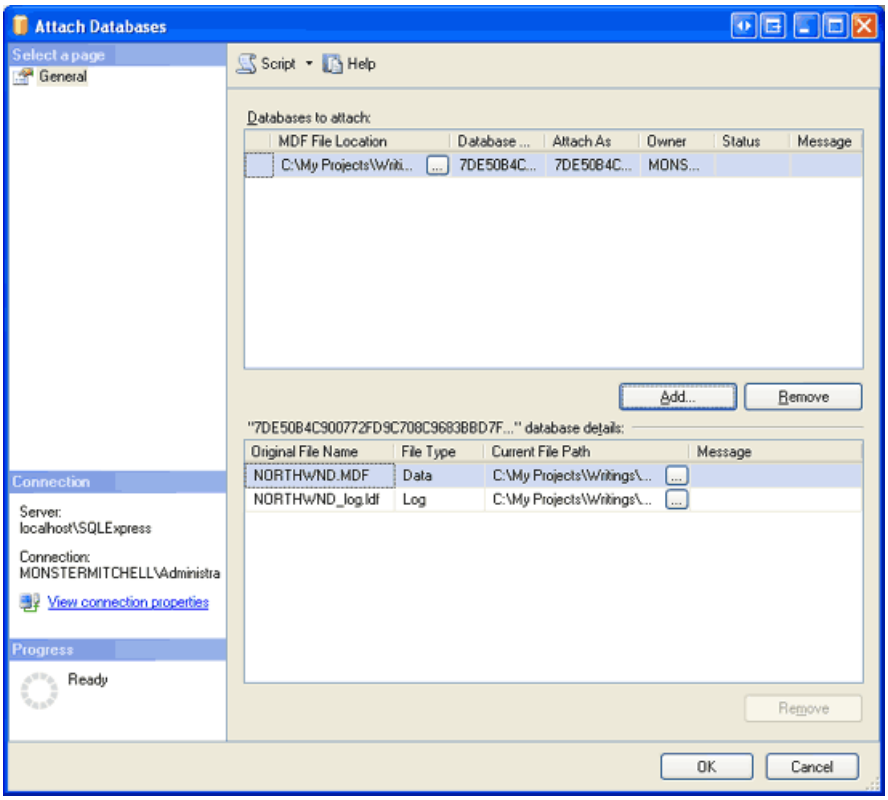


Figure 2: Connect to the Appropriate Database Instance (Click to view full-size image)

> **NOTE**
>
> When connecting to the SQL Server 2005 Express Edition instance through Management Studio the Attach Databases dialog box does not allow you to drill down into user profile directories, such as My Documents. Therefore, make sure to place the `NORTHWND.MDF` and `NORTHWND_log.LDF` files in a non-user profile directory.

Click the OK button to attach the database. The Attach Databases dialog box will close and the Object Explorer should now list the just-attached database. Chances are the Northwind database has a name like

```
9FE54661B32FDD967F51D71D0D5145CC_LINE ARTICLES\DATATUTORIALS\VOLUME
3\CSHARP\73\ASPNET_DATA_TUTORIAL_75_CS\APP_DATA\NORTHWND.MDF
```

. Rename the database to Northwind by right-clicking on the database and choosing Rename.

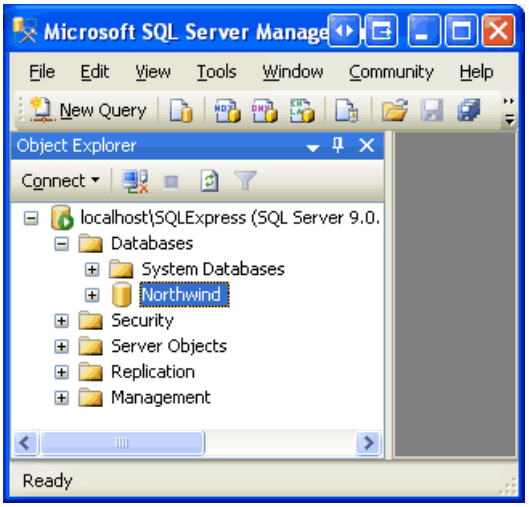


Figure 3: Rename the Database to Northwind

Step 2: Creating a New Solution and SQL Server Project in Visual Studio

To create managed stored procedures or UDFs in SQL Server 2005 we will write the stored procedure and UDF logic as C# code in a class. Once the code has been written, we will need to compile this class into an assembly (a `.dll` file), register the assembly with the SQL Server database, and then create a stored procedure or UDF object in the database that points to the corresponding method in the assembly. These steps can all be performed manually. We can create the code in any text editor, compile it from the command line using the C# compiler (`csc.exe`), register it with the database using the `CREATE ASSEMBLY` command or from Management Studio, and add the stored procedure or UDF object through similar means. Fortunately, the Professional and Team Systems versions of Visual Studio include a SQL Server Project type that automates these tasks. In this tutorial we will walk through using the SQL Server Project type to create a managed stored procedure and UDF.

> **NOTE**
>
> If you are using Visual Web Developer or the Standard edition of Visual Studio, then you will have to use the manual approach instead. Step 13 provides detailed instructions for performing these steps manually. I encourage you to read Steps 2 through 12 before reading Step 13 since these steps include important SQL Server configuration instructions that must be applied regardless of what version of Visual Studio you are using.

Start by opening Visual Studio. From the File menu, choose New Project to display the New Project dialog box (see Figure 4). Drill down to the Database project type and then, from the Templates listed on the right, choose to create a new SQL Server Project. I have chosen to name this project `ManagedDatabaseConstructs` and placed it within a Solution named `Tutorial75`.

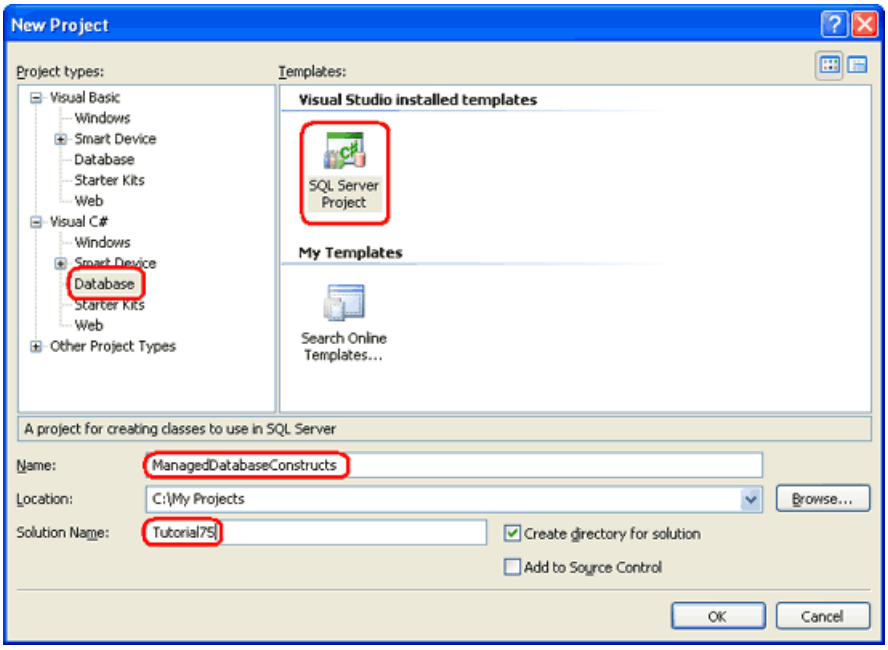


Figure 4: Create a New SQL Server Project (Click to view full-size image)

Click the OK button in the New Project dialog box to create the Solution and SQL Server Project.

A SQL Server Project is tied to a particular database. Consequently, after creating the new SQL Server Project we are immediately asked to specify this information. Figure 5 shows the New Database Reference dialog box that has been filled out to point to the Northwind database we registered in the SQL Server 2005 Express Edition database instance back in Step 1.

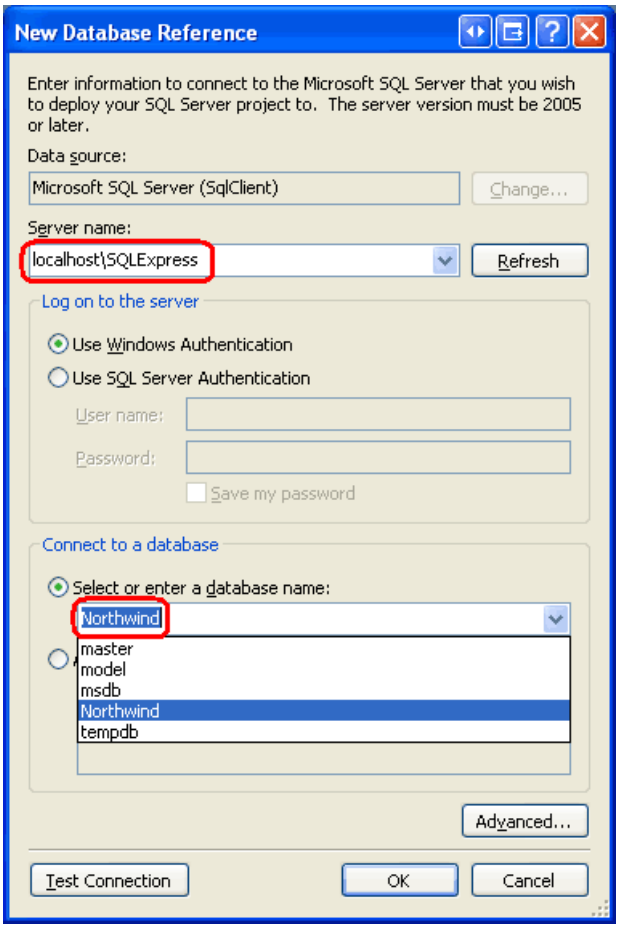


Figure 5: Associate the SQL Server Project with the Northwind Database

In order to debug the managed stored procedures and UDFs we will create within this project, we need to enable SQL/CLR debugging support for the connection. Whenever associating a SQL Server Project with a new database (as we did in Figure 5), Visual Studio asks us if we want to enable SQL/CLR debugging on the connection (see Figure 6). Click Yes.

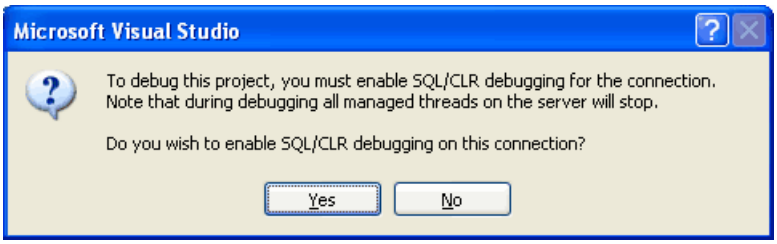


Figure 6: Enable SQL/CLR Debugging

At this point the new SQL Server Project has been added to the Solution. It contains a folder named `Test Scripts` with a file named `Test.sql`, which is used for debugging the managed database objects created in the project. We will look at debugging in Step 12.

We can now add new managed stored procedures and UDFs to this project, but before we do let s first include our existing web application in the Solution. From the File menu select the Add option and choose Existing Web Site. Browse to the appropriate website folder and click OK. As Figure 7 shows, this will update the Solution to include two projects: the website and the `ManagedDatabaseConstructs` SQL Server Project.

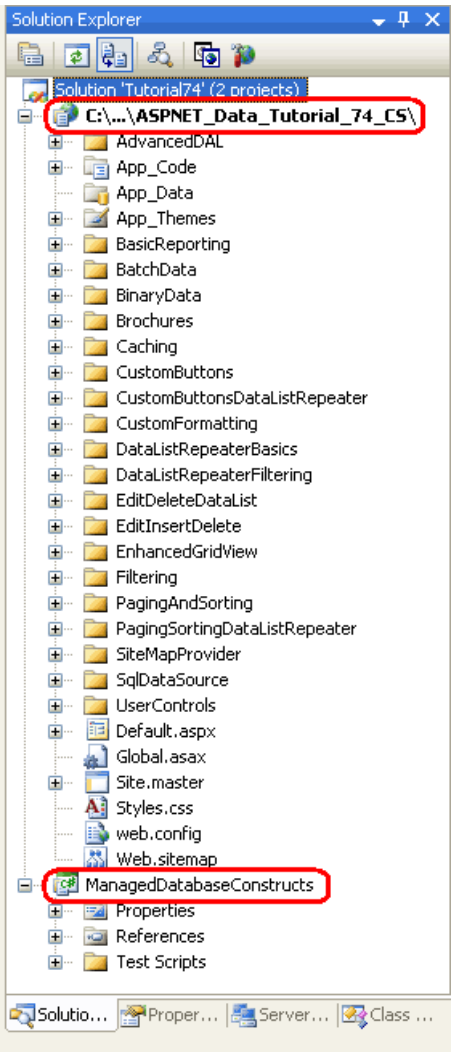


Figure 7: The Solution Explorer Now Includes Two Projects

The `NORTHWNDConnectionString` value in `Web.config` currently references the `NORTHWND.MDF` file in the `App_Data` folder. Since we removed this database from `App_Data` and explicitly registered it in the SQL Server 2005 Express Edition database instance, we need to correspondingly update the `NORTHWNDConnectionString` value. Open the `Web.config` file in the website and change the `NORTHWNDConnectionString` value so that the connection string reads: `Data Source=localhost\SQLExpress;Initial Catalog=Northwind;Integrated Security=True`. After this change, your `<connectionStrings>` section in `Web.config` should look similar to the following:

```
<connectionStrings>
    <add name="NORTHWNDConnectionString" connectionString=
        "Data Source=localhost\SQLExpress;Initial Catalog=Northwind;
            Integrated Security=True;Pooling=false"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

> **NOTE**
>
> As discussed in the preceding tutorial, when debugging a SQL Server object from a client application, such as an ASP.NET website, we need to disable connection pooling. The connection string shown above disables connection pooling (`Pooling=false`). If you do not plan on debugging the managed stored procedures and UDFs from the ASP.NET website, enable connection pooling.

Step 3: Creating a Managed Stored Procedure

To add a managed stored procedure to the Northwind database we first need to create the stored procedure as a method in the SQL Server Project. From the Solution Explorer, right-click on the `ManagedDatabaseConstructs` project name and choose to add a new item. This will display the Add New Item dialog box, which lists the types of managed database objects that can be added to the project. As Figure 8 shows, this includes stored procedures and User-Defined Functions, among others.

Let s start by adding a stored procedure that simply returns all of the products that have been discontinued. Name the new stored procedure file `GetDiscontinuedProducts.cs`.

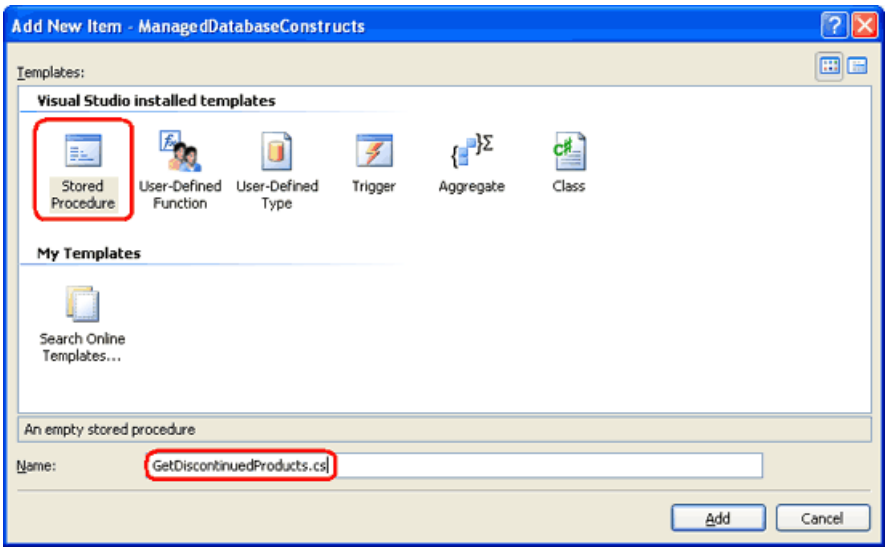


Figure 8: Add a New Stored Procedure Named `GetDiscontinuedProducts.cs` (Click to view full-size image)

This will create a new C# class file with the following content:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void GetDiscontinuedProducts()
    {
        // Put your code here
    }
};
```

Note that the stored procedure is implemented as a `static` method within a `partial` class file named `StoredProcedures`. Moreover, the `GetDiscontinuedProducts` method is decorated with the `SqlProcedure attribute`, which marks the method as a stored procedure.

The following code creates a `SqlCommand` object and sets its `CommandText` to a `SELECT` query that returns all of the columns from the `Products` table for products whose `Discontinued` field equals 1. It then executes the command and sends the results back to the client application. Add this code to the `GetDiscontinuedProducts` method.

```
// Create the command
SqlCommand myCommand = new SqlCommand();
myCommand.CommandText =
      @"SELECT ProductID, ProductName, SupplierID, CategoryID,
               QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
               ReorderLevel, Discontinued
        FROM Products
        WHERE Discontinued = 1";
// Execute the command and send back the results
SqlContext.Pipe.ExecuteAndSend(myCommand);
```

All managed database objects have access to a `SqlContext` object that represents the context of the caller. The `SqlContext` provides access to a `SqlPipe` object via its `Pipe` property. This `SqlPipe` object is used to ferry information between the SQL Server database and the calling application. As its name implies, the `ExecuteAndSend` method executes a passed-in `SqlCommand` object and sends the results back to the client application.

> **NOTE**
> Managed database objects are best suited for stored procedures and UDFs that use procedural logic rather than set-based logic. Procedural logic involves working with sets of data on a row-by-row basis or working with scalar data. The `GetDiscontinuedProducts` method we just created, however, involves no procedural logic. Therefore, it would ideally be implemented as a T-SQL stored procedure. It is implemented as a managed stored procedure to demonstrate the steps necessary for creating and deploying managed stored procedures.

Step 4: Deploying the Managed Stored Procedure

With this code complete, we are ready to deploy it to the Northwind database. Deploying a SQL Server Project compiles the code into an assembly, registers the assembly with the database, and creates the corresponding objects in the database, linking them to the appropriate methods in the assembly. The exact set of tasks performed by the Deploy option is more precisely spelled out in Step 13. Right-click on the `ManagedDatabaseConstructs` project name in the Solution Explorer and choose the Deploy option. However, deployment fails with the following error: Incorrect syntax near 'EXTERNAL'. You may need to set the compatibility level of the current database to a higher value to enable this feature. See help for the stored procedure `sp_dbcmptlevel`.

This error message occurs when attempting to register the assembly with the Northwind database. In order to register an assembly with a SQL Server 2005 database, the database s compatibility level must be set to 90. By default, new SQL Server 2005 databases have a compatibility level of 90. However, databases created using Microsoft SQL Server 2000 have a default compatibility level of 80. Since the Northwind database was initially a Microsoft SQL Server 2000 database, its compatibility level is currently set to 80 and therefore needs to be increased to 90 in order to register managed database objects.

To update the database s compatibility level, open a New Query window in Management Studio and enter:

```
exec sp_dbcmptlevel 'Northwind', 90
```

Click the Execute icon in the Toolbar to run the above query.

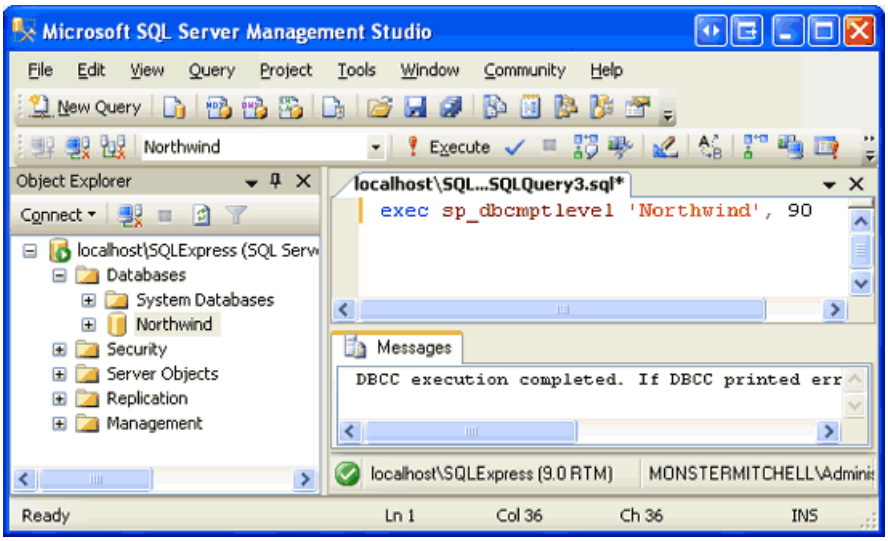


Figure 9: Update the Northwind Database s Compatibility Level (Click to view full-size image)

After updating the compatibility level, redeploy the SQL Server Project. This time the deployment should complete without error.

Return to SQL Server Management Studio, right-click on the Northwind database in the Object Explorer, and choose Refresh. Next, drill down into the Programmability folder and then expand the Assemblies folder. As Figure 10 shows, the Northwind database now includes the assembly generated by the `ManagedDatabaseConstructs` project.

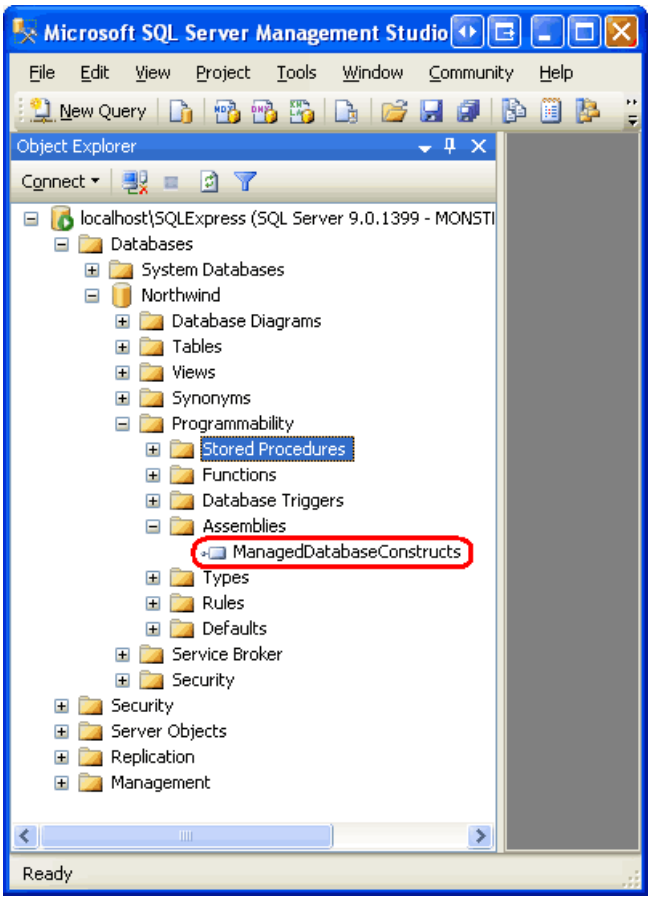


Figure 10: The `ManagedDatabaseConstructs` Assembly is Now Registered with the Northwind Database

Also expand the Stored Procedures folder. There you will see a stored procedure named `GetDiscontinuedProducts`. This stored procedure was created by the deployment process and points to the `GetDiscontinuedProducts` method in the `ManagedDatabaseConstructs` assembly. When the `GetDiscontinuedProducts` stored procedure is executed, it, in turn, executes the `GetDiscontinuedProducts` method. Since this is a managed stored procedure it cannot be

edited through Management Studio (hence the lock icon next to the stored procedure name).

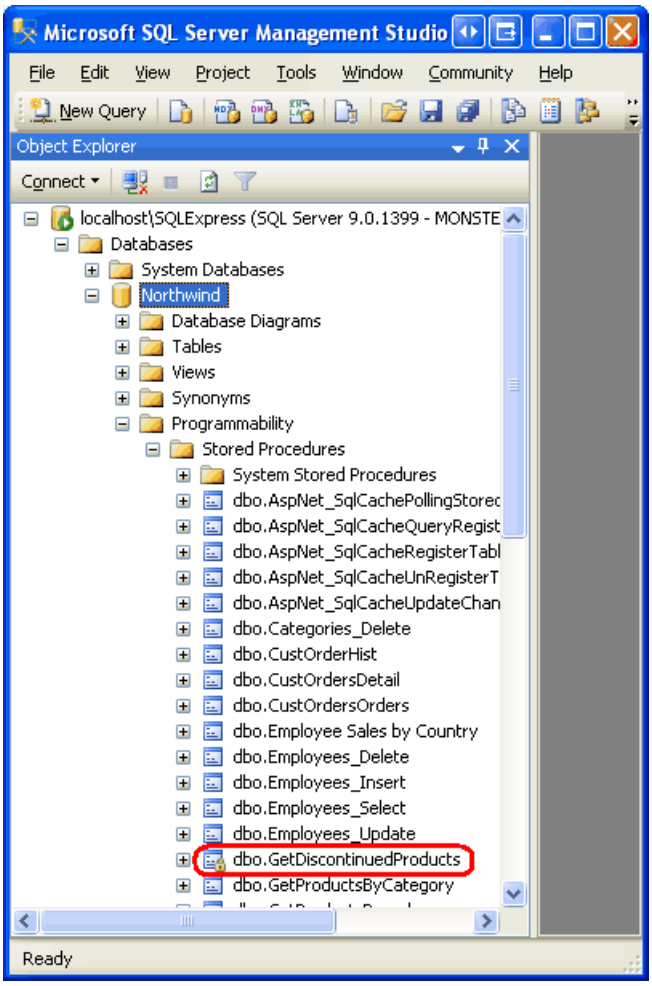


Figure 11: The `GetDiscontinuedProducts` Stored Procedure is Listed in the Stored Procedures Folder

There is still one more hurdle we have to overcome before we can call the managed stored procedure: the database is configured to prevent execution of managed code. Verify this by opening a new query window and executing the `GetDiscontinuedProducts` stored procedure. You will receive the following error message: Execution of user code in the .NET Framework is disabled. Enable 'clr enabled configuration option.

To examine the Northwind database s configuration information, enter and execute the command `exec sp_configure` in the query window. This shows that the clr enabled setting is currently set to 0.

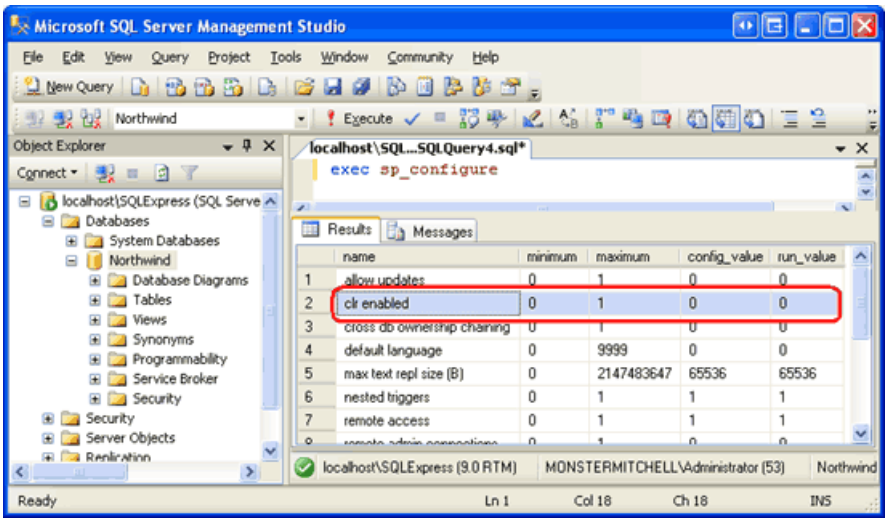


Figure 12: The clr enabled Setting is Currently Set to 0 (Click to view full-size image)

Note that each configuration setting in Figure 12 has four values listed with it: the minimum and maximum values

and the config and run values. To update the config value for the clr enabled setting, execute the following command:

```
exec sp_configure 'clr enabled', 1
```

If you re-run the `exec sp_configure` you will see that the above statement updated the clr enabled setting s config value to 1, but that the run value is still set to 0. For this configuration change to take affect we need to execute the `RECONFIGURE` command, which will set the run value to the current config value. Simply enter `RECONFIGURE` in the query window and click the Execute icon in the Toolbar. If you run `exec sp_configure` now you should see a value of 1 for the clr enabled setting s config and run values.

With the clr enabled configuration complete, we are ready to run the managed `GetDiscontinuedProducts` stored procedure. In the query window enter and execute the command `exec` `GetDiscontinuedProducts`. Invoking the stored procedure causes the corresponding managed code in the `GetDiscontinuedProducts` method to execute. This code issues a `SELECT` query to return all products that are discontinued and returns this data to the calling application, which is SQL Server Management Studio in this instance. Management Studio receives these results and displays them in the Results window.

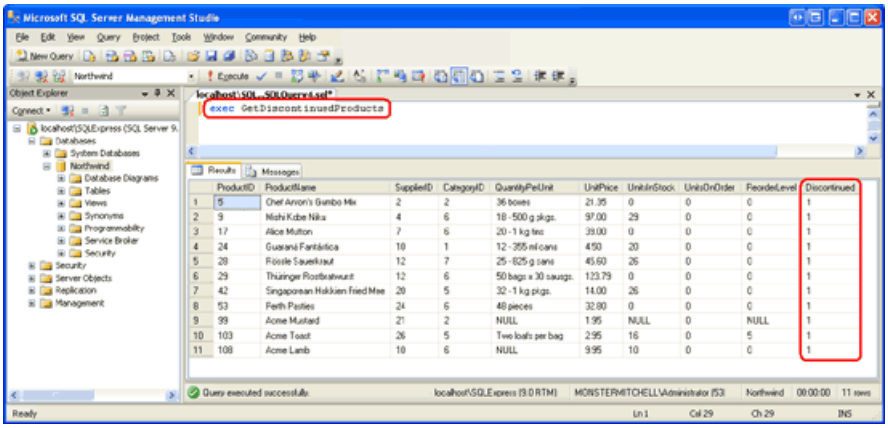


Figure 13: The `GetDiscontinuedProducts` Stored Procedure Returns All Discontinued Products (Click to view full-size image)

Step 5: Creating Managed Stored Procedures that Accept Input Parameters

Many of the queries and stored procedures we have created throughout these tutorials have used *parameters*. For example, in the Creating New Stored Procedures for the Typed DataSet s TableAdapters tutorial we created a stored procedure named `GetProductsByCategoryID` that accepted an input parameter named `@CategoryID`. The stored procedure then returned all products whose `CategoryID` field matched the value of the supplied `@CategoryID` parameter.

To create a managed stored procedure that accepts input parameters, simply specify those parameters in the method s definition. To illustrate this, let s add another managed stored procedure to the `ManagedDatabaseConstructs` project named `GetProductsWithPriceLessThan`. This managed stored procedure will accept an input parameter specifying a price and will return all products whose `UnitPrice` field is less than the parameter s value.

To add a new stored procedure to the project, right-click on the `ManagedDatabaseConstructs` project name and choose to add a new stored procedure. Name the file `GetProductsWithPriceLessThan.cs`. As we saw in Step 3, this will create a new C# class file with a method named `GetProductsWithPriceLessThan` placed within the `partial` class `StoredProcedures`.

Update the `GetProductsWithPriceLessThan` method s definition so that it accepts a `SqlMoney` input parameter

named `price` and write the code to execute and return the query results:

```
[Microsoft.SqlServer.Server.SqlProcedure]
public static void GetProductsWithPriceLessThan(SqlMoney price)
{
    // Create the command
    SqlCommand myCommand = new SqlCommand();
    myCommand.CommandText =
          @"SELECT ProductID, ProductName, SupplierID, CategoryID,
                   QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
                   ReorderLevel, Discontinued
            FROM Products
            WHERE UnitPrice < @MaxPrice";
    myCommand.Parameters.AddWithValue("@MaxPrice", price);
    // Execute the command and send back the results
    SqlContext.Pipe.ExecuteAndSend(myCommand);
}
```

The `GetProductsWithPriceLessThan` method s definition and code closely resembles the definition and code of the `GetDiscontinuedProducts` method created in Step 3. The only differences are that the `GetProductsWithPriceLessThan` method accepts as input parameter (`price`), the `SqlCommand` s query includes a parameter (`@MaxPrice`), and a parameter is added to the `SqlCommand` s `Parameters` collection is and assigned the value of the `price` variable.

After adding this code, redeploy the SQL Server Project. Next, return to SQL Server Management Studio and Refresh the Stored Procedures folder. You should see a new entry, `GetProductsWithPriceLessThan`. From a query window, enter and execute the command `exec GetProductsWithPriceLessThan 25`, which will list all products less than $25, as Figure 14 shows.

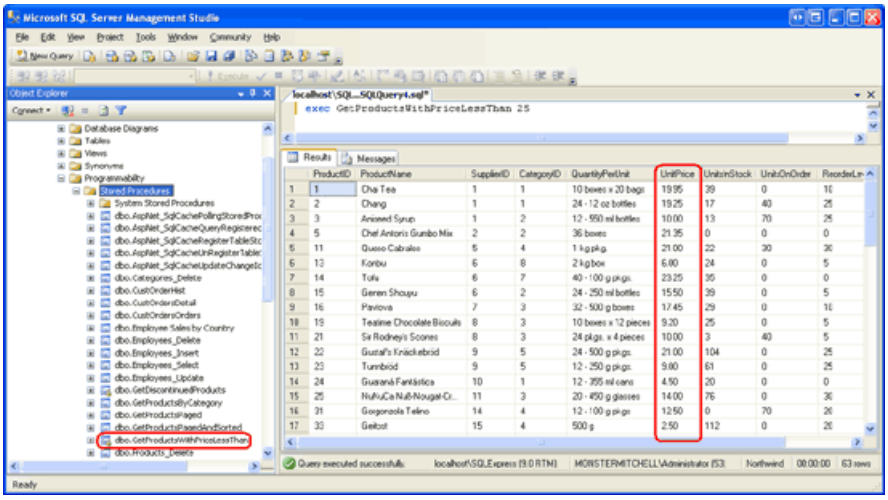


Figure 14: Products Under $25 are Displayed (Click to view full-size image)

Step 6: Calling the Managed Stored Procedure from the Data Access Layer

At this point we have added the `GetDiscontinuedProducts` and `GetProductsWithPriceLessThan` managed stored procedures to the `ManagedDatabaseConstructs` project and have registered them with the Northwind SQL Server database. We also invoked these managed stored procedures from SQL Server Management Studio (see Figure s 13 and 14). In order for our ASP.NET application to use these managed stored procedures, however, we need to add them to the Data Access and Business Logic Layers in the architecture. In this step we will add two new methods to the `ProductsTableAdapter` in the `NorthwindWithSprocs` Typed DataSet, which was initially created in the Creating New Stored Procedures for the Typed DataSet s TableAdapters tutorial. In Step 7 we will add corresponding methods to the BLL.

Open the `NorthwindWithSprocs` Typed DataSet in Visual Studio and start by adding a new method to the `ProductsTableAdapter` named `GetDiscontinuedProducts`. To add a new method to a TableAdapter, right-click on the TableAdapter s name in the Designer and choose the Add Query option from the context menu.

> **NOTE**
>
> Since we moved the Northwind database from the `App_Data` folder to the SQL Server 2005 Express Edition database instance, it is imperative that the corresponding connection string in Web.config be updated to reflect this change. In Step 2 we discussed updating the `NORTHWNDConnectionString` value in `Web.config`. If you forgot to make this update, then you will see the error message Failed to add query. Unable to find connection `NORTHWNDConnectionString` for object `Web.config` in a dialog box when attempting to add a new method to the TableAdapter. To resolve this error, click OK and then go to `Web.config` and update the `NORTHWNDConnectionString` value as discussed in Step 2. Then try re-adding the method to the TableAdapter. This time it should work without error.

Adding a new method launches the TableAdapter Query Configuration wizard, which we have used many times in past tutorials. The first step asks us to specify how the TableAdapter should access the database: through an ad-hoc SQL statement or via a new or existing stored procedure. Since we have already created and registered the `GetDiscontinuedProducts` managed stored procedure with the database, choose the Use existing stored procedure option and hit Next.

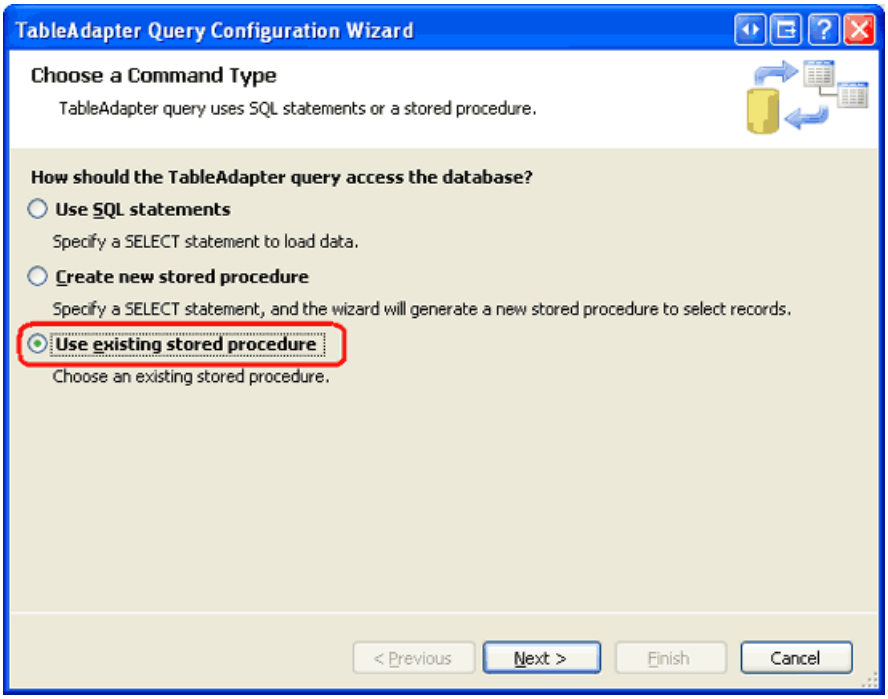


Figure 15: Choose the Use existing stored procedure Option (Click to view full-size image)

The next screen prompts us for the stored procedure the method will invoke. Choose the `GetDiscontinuedProducts` managed stored procedure from the drop-down list and hit Next.

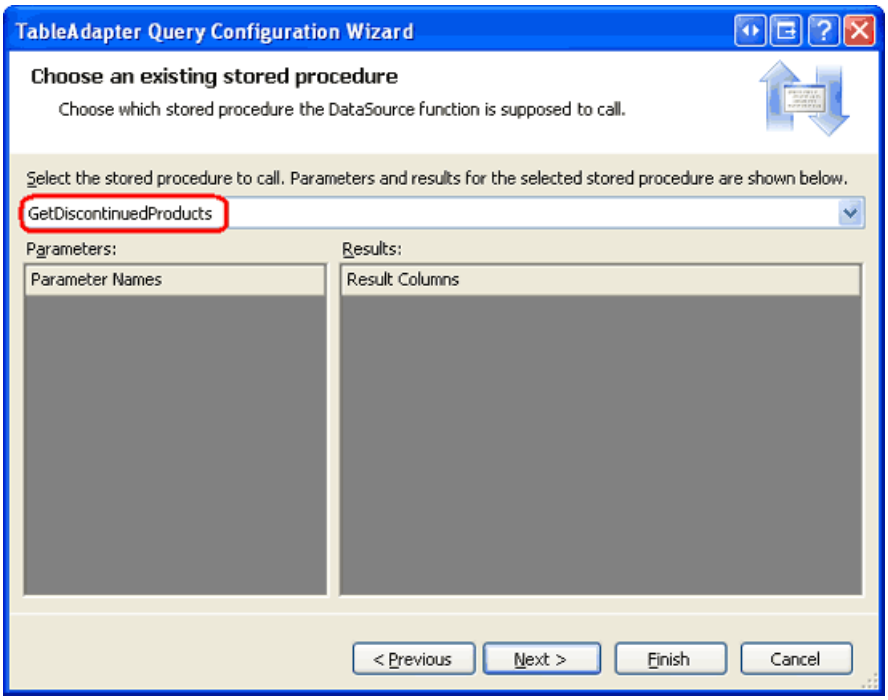


Figure 16: Select the `GetDiscontinuedProducts` Managed Stored Procedure (Click to view full-size image)

We are then asked to specify whether the stored procedure returns rows, a single value, or nothing. Since `GetDiscontinuedProducts` returns the set of discontinued product rows, choose the first option (Tabular data) and click Next.

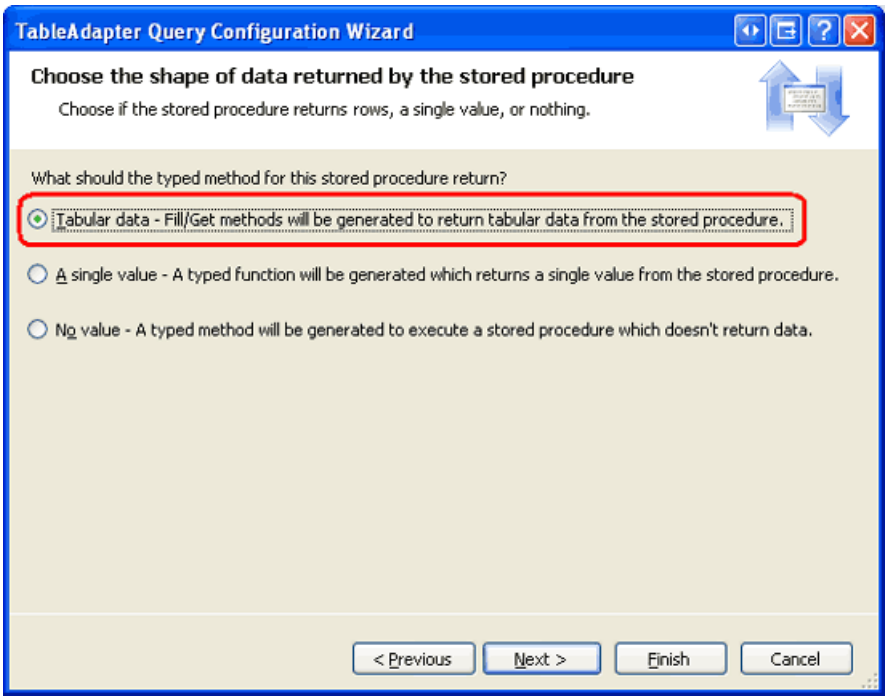


Figure 17: Select the Tabular Data Option (Click to view full-size image)

The final wizard screen allows us to specify the data access patterns used and the names of the resulting methods. Leave both checkboxes checked and name the methods `FillByDiscontinued` and `GetDiscontinuedProducts`. Click Finish to complete the wizard.

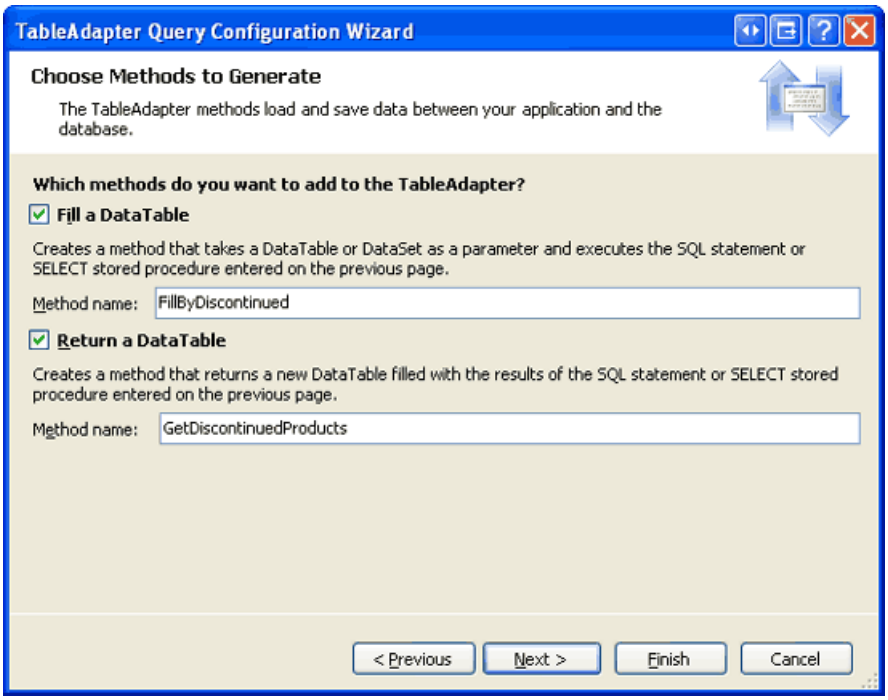


Figure 18: Name the Methods `FillByDiscontinued` and `GetDiscontinuedProducts` (Click to view full-size image)

Repeat these steps to create methods named `FillByPriceLessThan` and `GetProductsWithPriceLessThan` in the `ProductsTableAdapter` for the `GetProductsWithPriceLessThan` managed stored procedure.

Figure 19 shows a screenshot of the DataSet Designer after adding the methods to the `ProductsTableAdapter` for the `GetDiscontinuedProducts` and `GetProductsWithPriceLessThan` managed stored procedures.

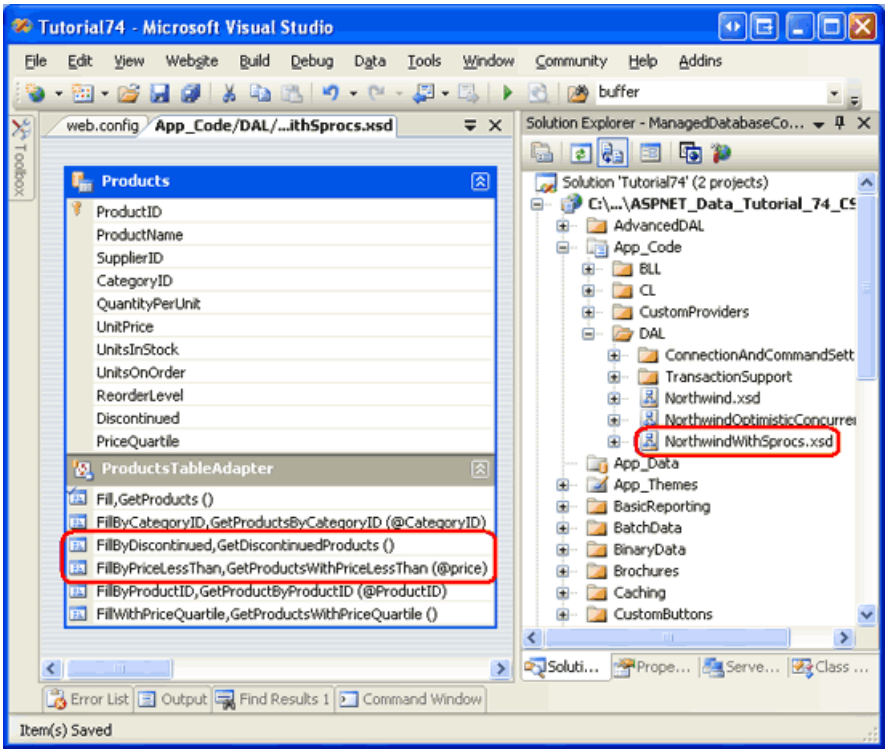


Figure 19: The `ProductsTableAdapter` Includes the New Methods Added in this Step (Click to view full-size image)

Step 7: Adding Corresponding Methods to the Business Logic Layer

Now that we have updated the Data Access Layer to include methods for calling the managed stored procedures added in Steps 4 and 5, we need to add corresponding methods to the Business Logic Layer. Add the following two methods to the `ProductsBLLWithSprocs` class:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Select, false)]
public NorthwindWithSprocs.ProductsDataTable GetDiscontinuedProducts()
{
    return Adapter.GetDiscontinuedProducts();
}
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Select, false)]
public NorthwindWithSprocs.ProductsDataTable
    GetProductsWithPriceLessThan(decimal priceLessThan)
{
    return Adapter.GetProductsWithPriceLessThan(priceLessThan);
}
```

Both methods simply call the corresponding DAL method and return the `ProductsDataTable` instance. The `DataObjectMethodAttribute` markup above each method causes these methods to be included in the drop-down list in the SELECT tab of the ObjectDataSource s Configure Data Source wizard.

Step 8: Invoking the Managed Stored Procedures from the Presentation Layer

With the Business Logic and Data Access Layers augmented to include support for calling the `GetDiscontinuedProducts` and `GetProductsWithPriceLessThan` managed stored procedures, we can now display these stored procedures results through an ASP.NET page.

Open the `ManagedFunctionsAndSprocs.aspx` page in the `AdvancedDAL` folder and, from the Toolbox, drag a GridView onto the Designer. Set the GridView s `ID` property to `DiscontinuedProducts` and, from its smart tag, bind it to a new ObjectDataSource named `DiscontinuedProductsDataSource`. Configure the ObjectDataSource to pull its data from the `ProductsBLLWithSprocs` class s `GetDiscontinuedProducts` method.

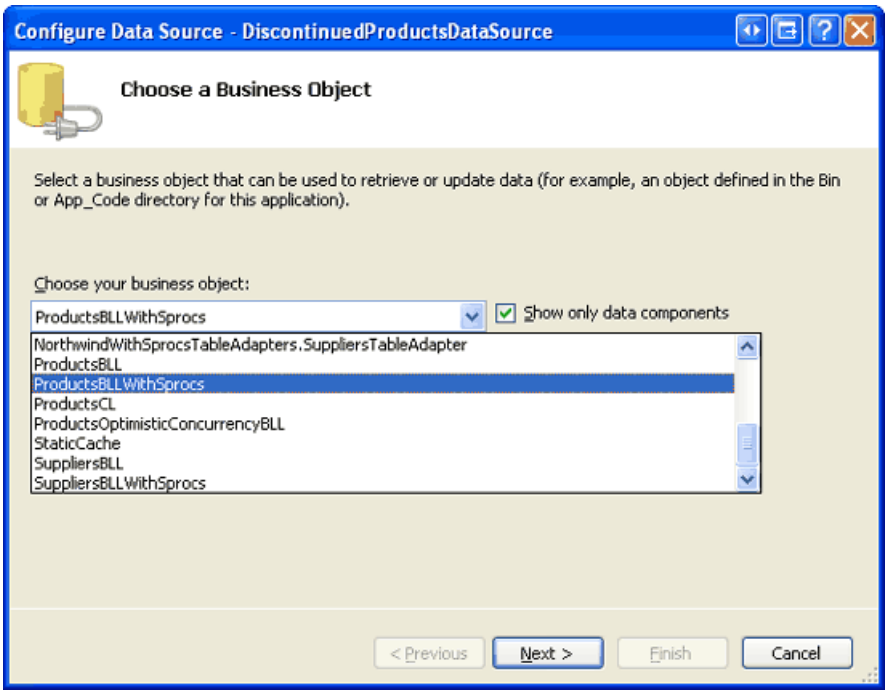


Figure 20: Configure the ObjectDataSource to Use the `ProductsBLLWithSprocs` Class (Click to view full-size image)

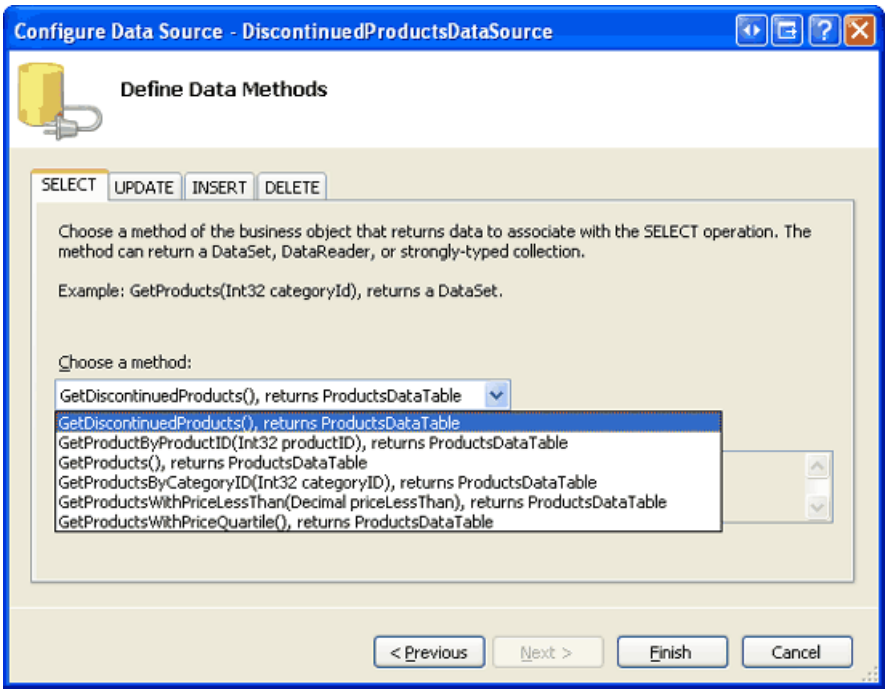


Figure 21: Choose the `GetDiscontinuedProducts` Method from the Drop-Down List in the SELECT Tab (Click to view full-size image)

Since this grid will be used to just display product information, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) and then click Finish.

Upon completing the wizard, Visual Studio will automatically add a BoundField or CheckBoxField for each data field in the `ProductsDataTable`. Take a moment to remove all of these fields except for `ProductName` and `Discontinued`, at which point your GridView and ObjectDataSource s declarative markup should look similar to the following:

```
<asp:GridView ID="DiscontinuedProducts" runat="server"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="DiscontinuedProductsDataSource">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="ProductName"
            SortExpression="ProductName" />
        <asp:CheckBoxField DataField="Discontinued"
            HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="DiscontinuedProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetDiscontinuedProducts" TypeName="ProductsBLLWithSprocs">
</asp:ObjectDataSource>
```

Take a moment to view this page through a browser. When the page is visited, the ObjectDataSource calls the `ProductsBLLWithSprocs` class s `GetDiscontinuedProducts` method. As we saw in Step 7, this method calls down to the DAL s `ProductsDataTable` class s `GetDiscontinuedProducts` method, which invokes the `GetDiscontinuedProducts` stored procedure. This stored procedure is a managed stored procedure and executes the code we created in Step 3, returning the discontinued products.

The results returned by the managed stored procedure are packaged up into a `ProductsDataTable` by the DAL and then returned to the BLL, which then returns them to the Presentation Layer where they are bound to the GridView and displayed. As expected, the grid lists those products that have been discontinued.

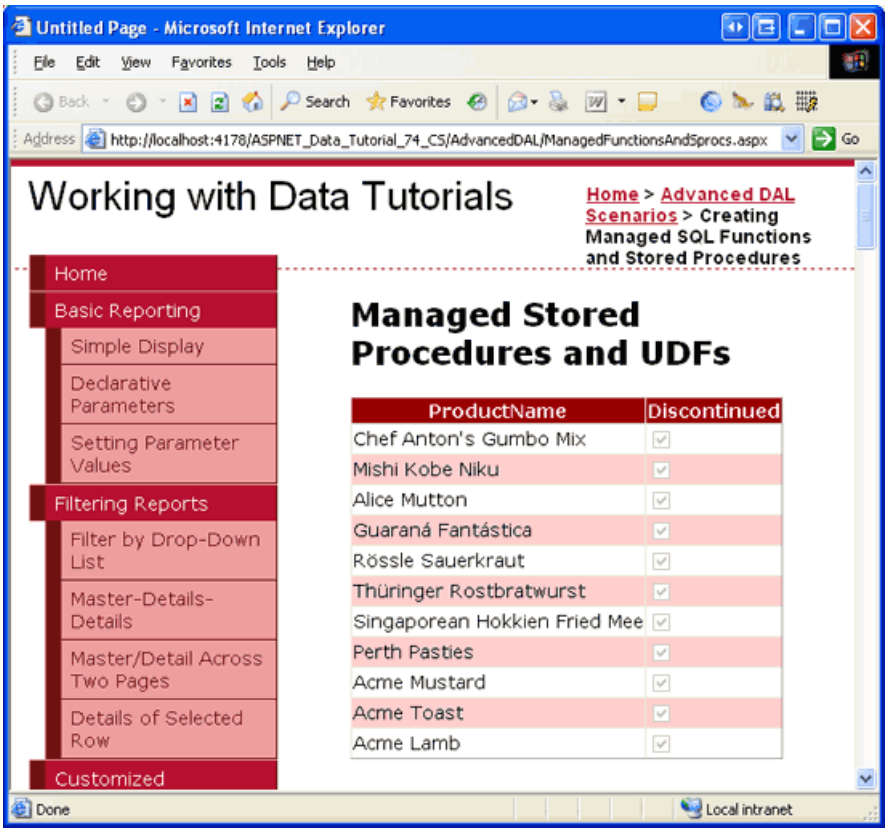


Figure 22: The Discontinued Products are Listed (Click to view full-size image)

For further practice, add a TextBox and another GridView to the page. Have this GridView display the products less than the amount entered into the TextBox by calling the `ProductsBLLWithSprocs` class s `GetProductsWithPriceLessThan` method.

Step 9: Creating and Calling T-SQL UDFs

User-Defined Functions, or UDFs, are database objects the closely mimic the semantics of functions in programming languages. Like a function in C#, UDFs can include a variable number of input parameters and return a value of a particular type. A UDF can return either scalar data - a string, an integer, and so forth - or tabular data. Let s take a quick look at both types of UDFs, starting with a UDF that returns a scalar data type.

The following UDF calculates the estimated value of the inventory for a particular product. It does so by taking in three input parameters - the `UnitPrice`, `UnitsInStock`, and `Discontinued` values for a particular product - and returns a value of type `money`. It computes the estimated value of the inventory by multiplying the `UnitPrice` by the `UnitsInStock`. For discontinued items, this value is halved.

```
CREATE FUNCTION udf_ComputeInventoryValue
(
    @UnitPrice money,
    @UnitsInStock smallint,
    @Discontinued bit
)
RETURNS money
AS
BEGIN
    DECLARE @Value decimal
    SET @Value = ISNULL(@UnitPrice, 0) * ISNULL(@UnitsInStock, 0)
    IF @Discontinued = 1
        SET @Value = @Value * 0.5

    RETURN @Value
END
```

Once this UDF has been added to the database, it can be found through Management Studio by expanding the Programmability folder, then Functions, and then Scalar-value Functions. It can be used in a `SELECT` query like so:

```
SELECT ProductID, ProductName, dbo.udf_ComputeInventoryValue
    (UnitPrice, UnitsInStock, Discontinued) as InventoryValue
FROM Products
ORDER BY InventoryValue DESC
```

I have added the `udf_ComputeInventoryValue` UDF to the Northwind database; Figure 23 shows the output of the above `SELECT` query when viewed through Management Studio. Also note that the UDF is listed under the Scalar-value Functions folder in the Object Explorer.

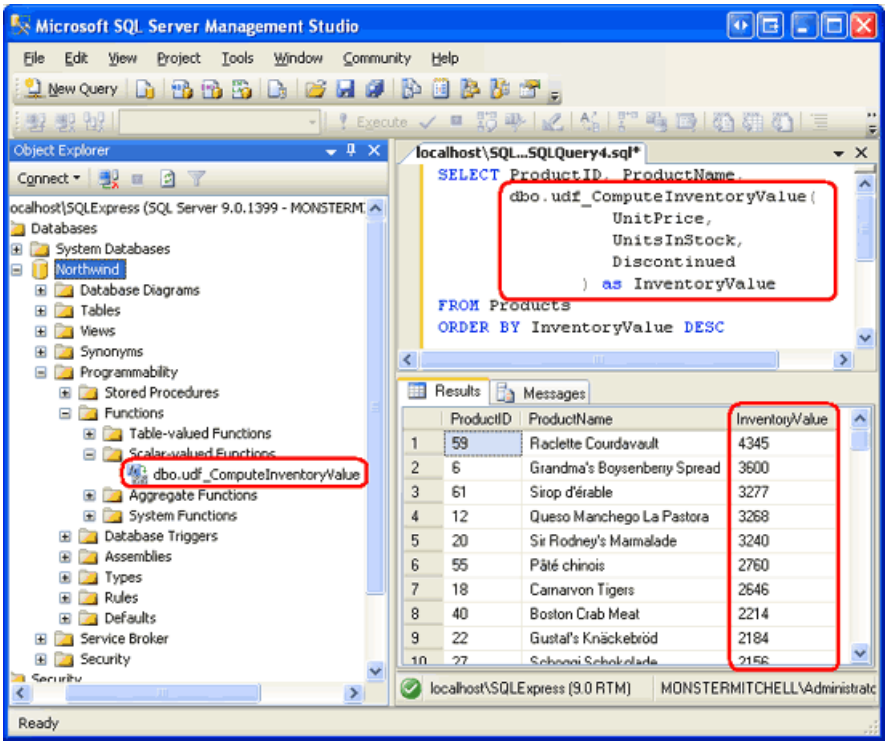


Figure 23: Each Product s Inventory Values is Listed (Click to view full-size image)

UDFs can also return tabular data. For example, we can create a UDF that returns products that belong to a particular category:

```
CREATE FUNCTION dbo.udf_GetProductsByCategoryID
(
    @CategoryID int
)
RETURNS TABLE
AS
RETURN
(
    SELECT ProductID, ProductName, SupplierID, CategoryID,
           QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
           ReorderLevel, Discontinued
    FROM Products
    WHERE CategoryID = @CategoryID
)
```

The `udf_GetProductsByCategoryID` UDF accepts a `@CategoryID` input parameter and returns the results of the specified `SELECT` query. Once created, this UDF can be referenced in the `FROM` (or `JOIN`) clause of a `SELECT` query. The following example would return the `ProductID`, `ProductName`, and `CategoryID` values for each of the beverages.

```
SELECT ProductID, ProductName, CategoryID
FROM dbo.udf_GetProductsByCategoryID(1)
```

I have added the `udf_GetProductsByCategoryID` UDF to the Northwind database; Figure 24 shows the output of the above `SELECT` query when viewed through Management Studio. UDFs that return tabular data can be found in the Object Explorer s Table-value Functions folder.

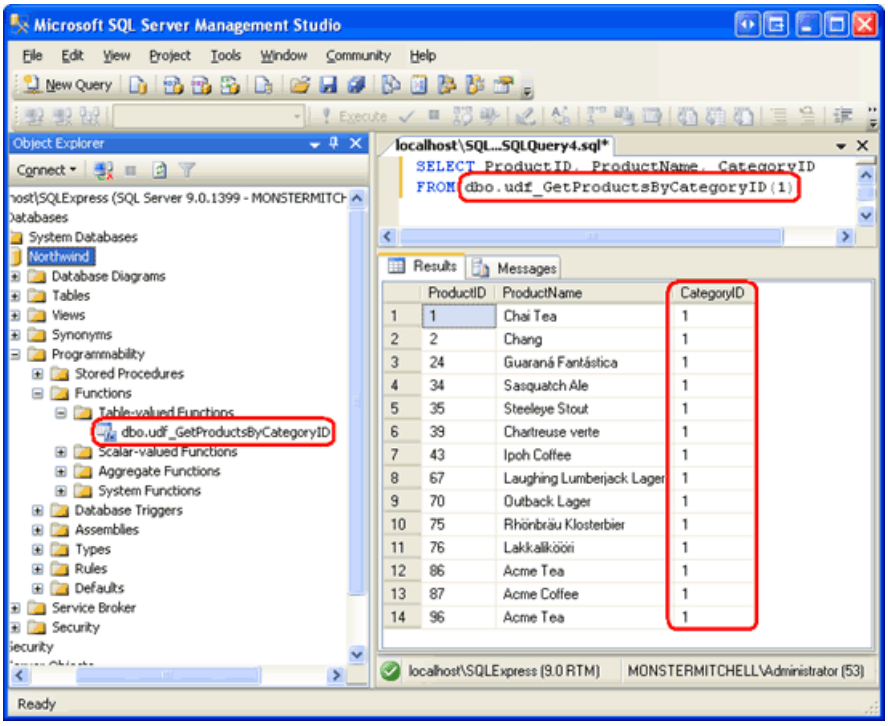


Figure 24: The `ProductID`, `ProductName`, and `CategoryID` are Listed for Each Beverage (Click to view full-size image)

> **NOTE**
>
> For more information on creating and using UDFs, check out Intro to User-Defined Functions. Also check out Advantages and Drawbacks of User-Defined Functions.

Step 10: Creating a Managed UDF

The `udf_ComputeInventoryValue` and `udf_GetProductsByCategoryID` UDFs created in the above examples are T-SQL database objects. SQL Server 2005 also supports managed UDFs, which can be added to the `ManagedDatabaseConstructs` project just like the managed stored procedures from Steps 3 and 5. For this step, let s implement the `udf_ComputeInventoryValue` UDF in managed code.

To add a managed UDF to the `ManagedDatabaseConstructs` project, right-click on the project name in Solution Explorer and choose to Add a New Item. Select the User-Defined Template from the Add New Item dialog box and name the new UDF file `udf_ComputeInventoryValue_Managed.cs`.

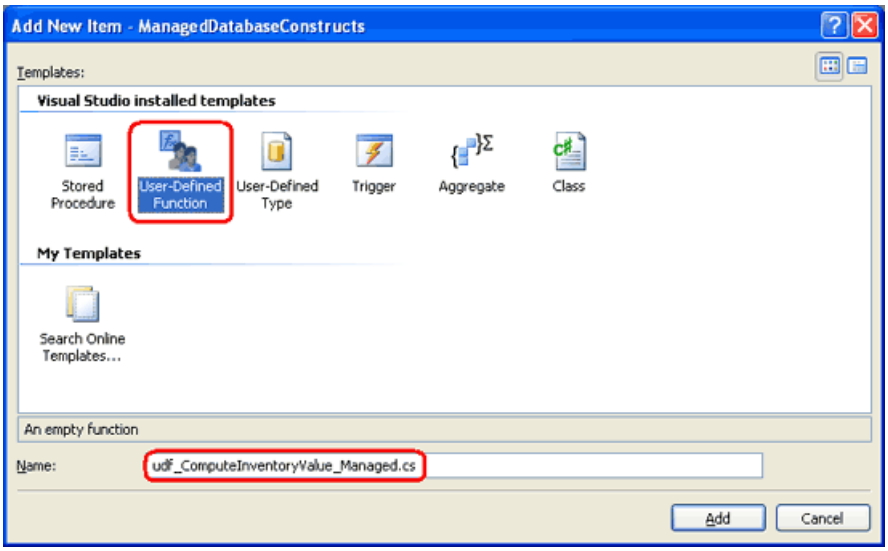


Figure 25: Add a New Managed UDF to the `ManagedDatabaseConstructs` Project (Click to view full-size image)

The User-Defined Function template creates a `partial` class named `UserDefinedFunctions` with a method whose name is the same as the class file s name (`udf_ComputeInventoryValue_Managed` , in this instance). This method is decorated using the `SqlFunction` attribute, which flags the method as a managed UDF.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
public partial class UserDefinedFunctions
{
    [Microsoft.SqlServer.Server.SqlFunction]
    public static SqlString udf_ComputeInventoryValue_Managed()
    {
        // Put your code here
        return new SqlString("Hello");
    }
};
```

The `udf_ComputeInventoryValue` method currently returns a `SqlString` object and does not accept any input parameters. We need to update the method definition so that it accepts three input parameters - `UnitPrice` , `UnitsInStock` , and `Discontinued` - and returns a `SqlMoney` object. The logic for calculating the inventory value is identical to that in the T-SQL `udf_ComputeInventoryValue` UDF.

```
[Microsoft.SqlServer.Server.SqlFunction]
public static SqlMoney udf_ComputeInventoryValue_Managed
    (SqlMoney UnitPrice, SqlInt16 UnitsInStock, SqlBoolean Discontinued)
{
    SqlMoney inventoryValue = 0;
    if (!UnitPrice.IsNull && !UnitsInStock.IsNull)
    {
        inventoryValue = UnitPrice * UnitsInStock;
        if (Discontinued == true)
            inventoryValue = inventoryValue * new SqlMoney(0.5);
    }
    return inventoryValue;
}
```

Note that the UDF method s input parameters are of their corresponding SQL types: `SqlMoney` for the `UnitPrice` field, `SqlInt16` for `UnitsInStock` , and `SqlBoolean` for `Discontinued` . These data types reflect the types defined in the `Products` table: the `UnitPrice` column is of type `money` , the `UnitsInStock` column of type `smallint` , and the

`Discontinued` column of type `bit`.

The code starts by creating a `SqlMoney` instance named `inventoryValue` that is assigned a value of 0. The `Products` table allows for database `NULL` values in the `UnitsInPrice` and `UnitsInStock` columns. Therefore, we need to first check to see if these values contain `NULL` s, which we do through the `SqlMoney` object s `IsNull` property. If both `UnitPrice` and `UnitsInStock` contain non- `NULL` values, then we compute the `inventoryValue` to be the product of the two. Then, if `Discontinued` is true, then we halve the value.

> **NOTE**
> The `SqlMoney` object only allows two `SqlMoney` instances to be multiplied together. It does not allow a `SqlMoney` instance to be multiplied by a literal floating-point number. Therefore, to halve `inventoryValue` we multiply it by a new `SqlMoney` instance that has the value 0.5.

Step 11: Deploying the Managed UDF

Now that the managed UDF has been created, we are ready to deploy it to the Northwind database. As we saw in Step 4, the managed objects in a SQL Server Project are deployed by right-clicking on the project name in the Solution Explorer and choosing the Deploy option from the context menu.

Once you have deployed the project, return to SQL Server Management Studio and refresh the Scalar-valued Functions folder. You should now see two entries:

- `dbo.udf_ComputeInventoryValue` - the T-SQL UDF created in Step 9, and
- `dbo.udf ComputeInventoryValue_Managed` - the managed UDF created in Step 10 that was just deployed.

To test this managed UDF, execute the following query from within Management Studio:

```
SELECT ProductID, ProductName,
       dbo.udf_ComputeInventoryValue_Managed(
                 UnitPrice,
                 UnitsInStock,
                 Discontinued
              ) as InventoryValue
FROM Products
ORDER BY InventoryValue DESC
```

This command uses the managed `udf ComputeInventoryValue_Managed` UDF instead of the T-SQL `udf_ComputeInventoryValue` UDF, but the output is the same. Refer back to Figure 23 to see a screenshot of the UDF s output.

Step 12: Debugging the Managed Database Objects

In the Debugging Stored Procedures tutorial we discussed the three options for debugging SQL Server through Visual Studio: Direct Database Debugging, Application Debugging, and Debugging from a SQL Server Project. Managed database objects cannot be debugged via Direct Database Debugging, but can be debugged from a client application and directly from the SQL Server Project. In order for debugging to work, however, the SQL Server 2005 database must allow SQL/CLR debugging. Recall that when we first created the `ManagedDatabaseConstructs` project Visual Studio asked us whether we wanted to enable SQL/CLR debugging (see Figure 6 in Step 2). This setting can be modified by right-clicking on the database from the Server Explorer window.

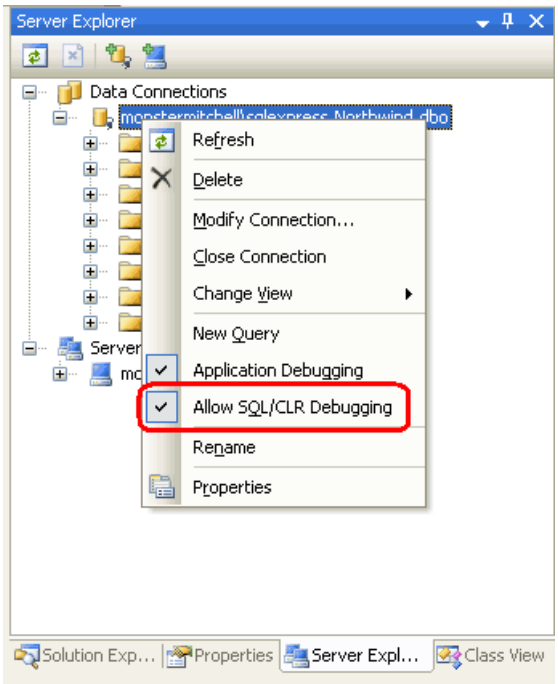


Figure 26: Ensure that the Database Allows SQL/CLR Debugging

Imagine that we wanted to debug the `GetProductsWithPriceLessThan` managed stored procedure. We would start by setting a breakpoint within the code of the `GetProductsWithPriceLessThan` method.

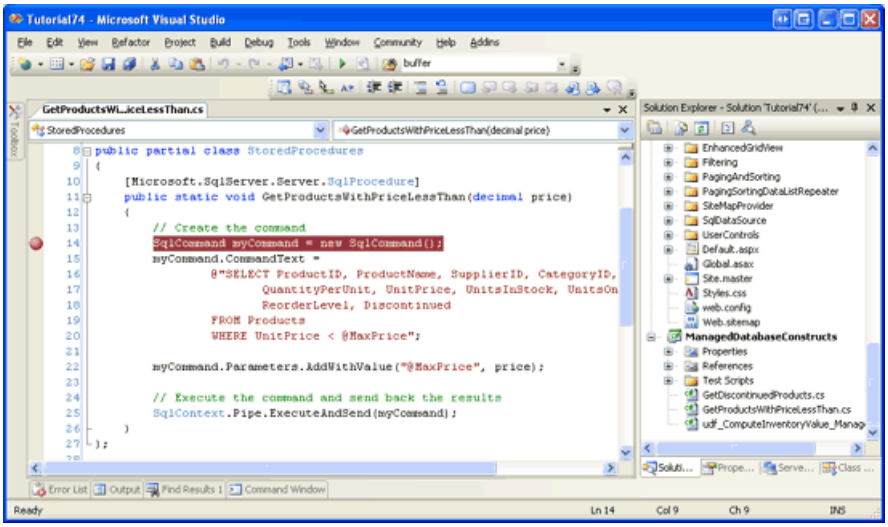


Figure 27: Set a Breakpoint in the `GetProductsWithPriceLessThan` Method (Click to view full-size image)

Let s first look at debugging the managed database objects from the SQL Server Project. Since our Solution includes two projects - the `ManagedDatabaseConstructs` SQL Server Project along with our website - in order to debug from the SQL Server Project we need to instruct Visual Studio to launch the `ManagedDatabaseConstructs` SQL Server Project when we start debugging. Right-click the `ManagedDatabaseConstructs` project in Solution Explorer and choose the Set as StartUp Project option from the context menu.

When the `ManagedDatabaseConstructs` project is launched from the debugger it executes the SQL statements in the `Test.sql` file, which is located in the `Test Scripts` folder. For example, to test the `GetProductsWithPriceLessThan` managed stored procedure, replace the existing `Test.sql` file content with the following statement, which invokes the `GetProductsWithPriceLessThan` managed stored procedure passing in the `@CategoryID` value of 14.95:

```
exec GetProductsWithPriceLessThan 14.95
```

Once you ve entered the above script into `Test.sql`, start debugging by going to the Debug menu and choosing

Start Debugging or by hitting F5 or the green play icon in the Toolbar. This will build the projects within the Solution, deploy the managed database objects to the Northwind database, and then execute the `Test.sql` script. At this point, the breakpoint will be hit and we can step through the `GetProductsWithPriceLessThan` method, examine the values of the input parameters, and so on.

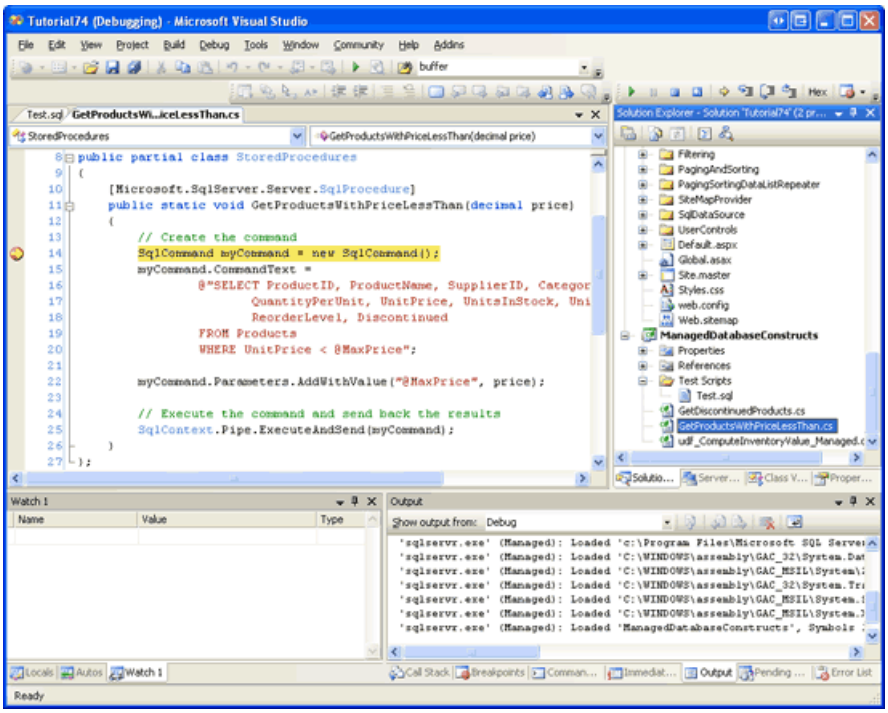


Figure 28: The Breakpoint in the `GetProductsWithPriceLessThan` Method Was Hit (Click to view full-size image)

In order for a SQL database object to be debugged through a client application, it is imperative that the database be configured to support application debugging. Right-click on the database in Server Explorer and ensure that the Application Debugging option is checked. Furthermore, we need to configure the ASP.NET application to integrate with the SQL Debugger and to disable connection pooling. These steps were discussed in detail in Step 2 of the Debugging Stored Procedures tutorial.

Once you have configured the ASP.NET application and database, set the ASP.NET website as the startup project and start debugging. If you visit a page that calls one of the managed objects that has a breakpoint, the application will halt and control will be turned over to the debugger, where you can step through the code as shown in Figure 28.

Step 13: Manually Compiling and Deploying Managed Database Objects

SQL Server Projects make it easy to create, compile, and deploy managed database objects. Unfortunately, SQL Server Projects are only available in the Professional and Team Systems editions of Visual Studio. If you are using Visual Web Developer or the Standard Edition of Visual Studio and want to use managed database objects, you will need to manually create and deploy them. This involves four steps:

1. Create a file that contains the source code for the managed database object,
2. Compile the object into an assembly,
3. Register the assembly with the SQL Server 2005 database, and
4. Create a database object in SQL Server that points to the appropriate method in the assembly.

To illustrate these tasks, let s create a new managed stored procedure that returns those products whose `UnitPrice` is greater than a specified value. Create a new file on your computer named `GetProductsWithPriceGreaterThan.cs` and enter the following code into the file (you can use Visual Studio, Notepad, or any text editor to accomplish this):

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void GetProductsWithPriceGreaterThan(SqlMoney price)
    {
        // Create the command
        SqlCommand myCommand = new SqlCommand();
        myCommand.CommandText =
            @"SELECT ProductID, ProductName, SupplierID, CategoryID,
                     QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
                     ReorderLevel, Discontinued
              FROM Products
              WHERE UnitPrice > @MinPrice";
        myCommand.Parameters.AddWithValue("@MinPrice", price);
        // Execute the command and send back the results
        SqlContext.Pipe.ExecuteAndSend(myCommand);
    }
};
```

This code is nearly identical to that of the `GetProductsWithPriceLessThan` method created in Step 5. The only differences are the method names, the `WHERE` clause, and the parameter name used in the query. Back in the `GetProductsWithPriceLessThan` method, the `WHERE` clause read: `WHERE UnitPrice < @MaxPrice`. Here, in `GetProductsWithPriceGreaterThan`, we use: `WHERE UnitPrice > @MinPrice` .

We now need to compile this class into an assembly. From the command line, navigate to the directory where you saved the `GetProductsWithPriceGreaterThan.cs` file and use the C# compiler (`csc.exe`) to compile the class file into an assembly:

```
csc.exe /t:library /out:ManuallyCreatedDBObjects.dll GetProductsWithPriceGreaterThan.cs
```

If the folder containing `csc.exe` in not in the system s `PATH`, you will have to fully reference its path, `%WINDOWS%\Microsoft.NET\Framework\version\`, like so:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\csc.exe /t:library /out:ManuallyCreatedDBObjects.dll
GetProductsWithPriceGreaterThan.cs
```

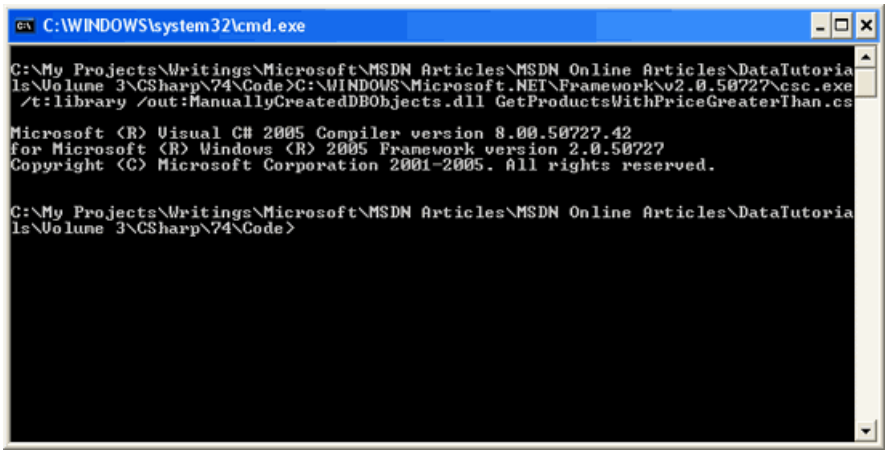


Figure 29: Compile `GetProductsWithPriceGreaterThan.cs` Into an Assembly (Click to view full-size image)

The `/t` flag specifies that the C# class file should be compiled into a DLL (rather than an executable). The `/out`

flag specifies the name of the resulting assembly.

> **NOTE**
>
> Rather than compiling the `GetProductsWithPriceGreaterThan.cs` class file from the command line you could alternatively use Visual C# Express Edition or create a separate Class Library project in Visual Studio Standard Edition. S ren Jacob Lauritsen has kindly provided such a Visual C# Express Edition project with code for the `GetProductsWithPriceGreaterThan` stored procedure and the two managed stored procedures and UDF created in Steps 3, 5, and 10. S ren s project also includes the T-SQL commands needed to add the corresponding database objects.

With the code compiled into an assembly, we are ready to register the assembly within the SQL Server 2005 database. This can be performed through T-SQL, using the command `CREATE ASSEMBLY`, or through SQL Server Management Studio. Let s focus on using Management Studio.

From Management Studio, expand the Programmability folder in the Northwind database. One of its subfolder is Assemblies. To manually add a new Assembly to the database, right-click on the Assemblies folder and choose New Assembly from the context menu. This displays the New Assembly dialog box (see Figure 30). Click on the Browse button, select the `ManuallyCreatedDBObjects.dll` assembly we just compiled, and then click OK to add the Assembly to the database. You should not see the `ManuallyCreatedDBObjects.dll` assembly in the Object Explorer.

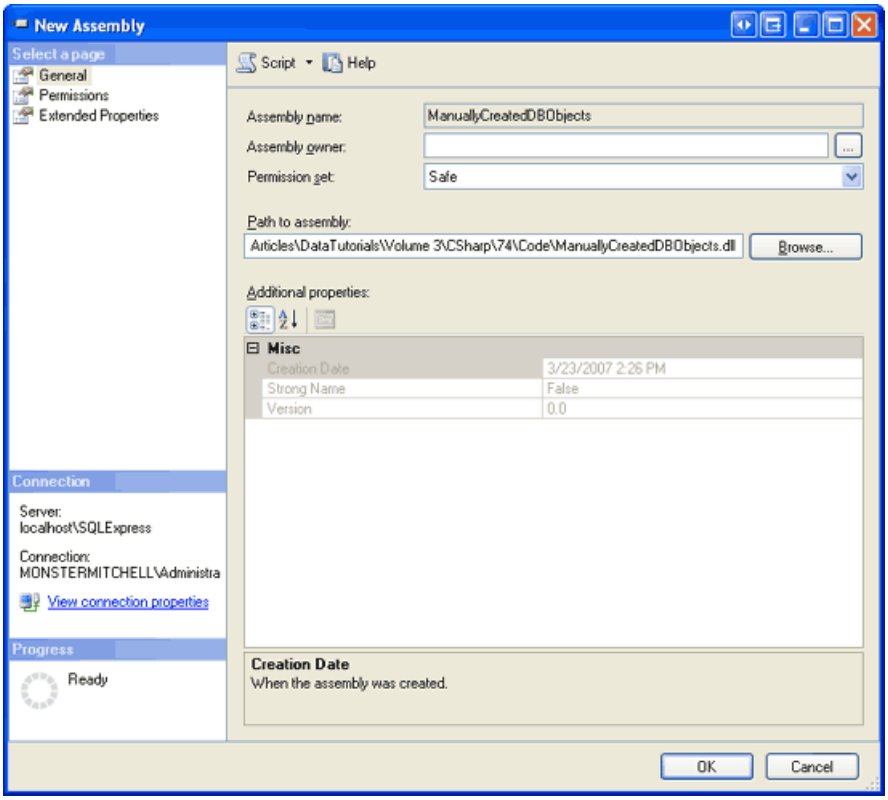


Figure 30: Add the `ManuallyCreatedDBObjects.dll` Assembly to the Database (Click to view full-size image)

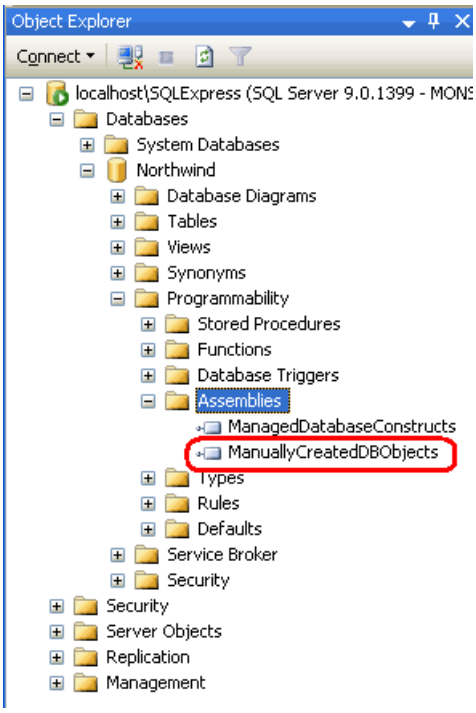


Figure 31: The `ManuallyCreatedDBObjects.dll` is Listed in the Object Explorer

While we have added the assembly to the Northwind database, we have yet to associate a stored procedure with the `GetProductsWithPriceGreaterThan` method in the assembly. To accomplish this, open a new query window and execute the following script:

```
CREATE PROCEDURE [dbo].[GetProductsWithPriceGreaterThan]
(
    @price money
)
WITH EXECUTE AS CALLER
AS
EXTERNAL NAME [ManuallyCreatedDBObjects].[StoredProcedures].[GetProductsWithPriceGreaterThan]
GO
```

This creates a new stored procedure in the Northwind database named `GetProductsWithPriceGreaterThan` and associates it with the managed method `GetProductsWithPriceGreaterThan` (which is in the class `StoredProcedures`, which is in the assembly `ManuallyCreatedDBObjects`).

After executing the above script, refresh the Stored Procedures folder in the Object Explorer. You should see a new stored procedure entry - `GetProductsWithPriceGreaterThan` - which has a lock icon next to it. To test this stored procedure, enter and execute the following script in the query window:

```
exec GetProductsWithPriceGreaterThan 24.95
```

As Figure 32 shows, the above command displays information for those products with a `UnitPrice` greater than $24.95.

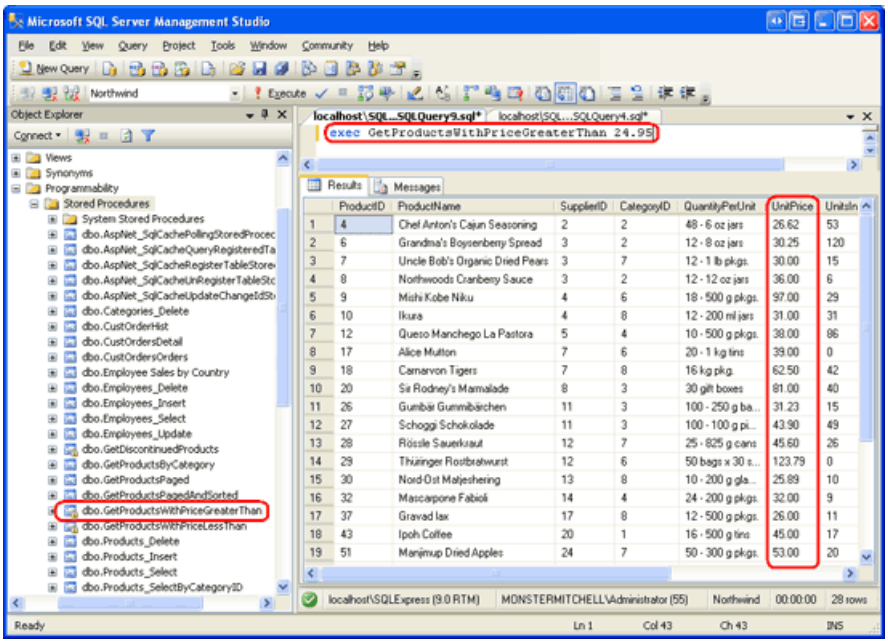


Figure 32: The `ManuallyCreatedDBObjects.dll` is Listed in the Object Explorer (Click to view full-size image)

Summary

Microsoft SQL Server 2005 provides integration with the Common Language Runtime (CLR), which allows database objects to be created using managed code. Previously, these database objects could only be created using T-SQL, but now we can create these objects using .NET programming languages like C#. In this tutorial we created two managed stored procedures and a managed User-Defined Function.

Visual Studio s SQL Server Project type facilitates creating, compiling, and deploying managed database objects. Moreover, it offers rich debugging support. However, SQL Server Project types are only available in the Professional and Team Systems editions of Visual Studio. For those using Visual Web Developer or the Standard Edition of Visual Studio, the creation, compilation, and deployment steps must be performed manually, as we saw in Step 13.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Advantages and Drawbacks of User-Defined Functions
- Creating SQL Server 2005 Objects in Managed Code
- Creating Triggers Using Managed Code in SQL Server 2005
- How To: Create and Run a CLR SQL Server Stored Procedure
- How To: Create and Run a CLR SQL Server User-Defined Function
- How To: Edit the `Test.sql` Script to Run SQL Objects
- Intro to User Defined Functions
- Managed Code and SQL Server 2005 (Video)
- Transact-SQL Reference
- Walkthrough: Creating a Stored Procedure in Managed Code

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest

book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was S ren Jacob Lauritsen. In addition to reviewing this article, S ren also created the Visual C# Express Edition project included in this article s download for manually compiling the managed database objects. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Creating New Stored Procedures for the Typed DataSet's TableAdapters (VB)

2/20/2020 • 27 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> In earlier tutorials we have created SQL statements in our code and passed the statements to the database to be executed. An alternative approach is to use stored procedures, where the SQL statements are pre-defined at the database. In this tutorial we learn how to have the TableAdapter Wizard generate new stored procedures for us.

Introduction

The Data Access Layer (DAL) for these tutorials uses Typed DataSets. As discussed in the Creating a Data Access Layer tutorial, Typed DataSets consist of strongly-typed DataTables and TableAdapters. The DataTables represent the logical entities in the system while the TableAdapters interface with the underlying database to perform the data access work. This includes populating the DataTables with data, executing queries that return scalar data, and inserting, updating, and deleting records from the database.

The SQL commands executed by the TableAdapters can be either ad-hoc SQL statements, such as `SELECT columnList FROM TableName`, or stored procedures. The TableAdapters in our architecture use ad-hoc SQL statements. Many developers and database administrators, however, prefer stored procedures over ad-hoc SQL statements for security, maintainability, and updatability reasons. Others ardently prefer ad-hoc SQL statements for their flexibility. In my own work I favor stored procedures over ad-hoc SQL statements, but chose to use ad-hoc SQL statements to simplify the earlier tutorials.

When defining a TableAdapter or adding new methods, the TableAdapter s wizard makes it just as easy to create new stored procedures or use existing stored procedures as it does to use ad-hoc SQL statements. In this tutorial we'll examine how to have the TableAdapter s wizard auto-generate stored procedures. In the next tutorial we will look at how to configure the TableAdapter s methods to use existing or manually-created stored procedures.

> **NOTE**
>
> See Rob Howard's blog entry Don t Use Stored Procedures Yet? and Frans Bouma s blog entry Stored Procedures are Bad, M Kay? for a lively debate on the pros and cons of stored procedures and ad-hoc SQL.

Stored Procedure Basics

Functions are a construct common to all programming languages. A function is a collection of statements that are executed when the function is called. Functions can accept input parameters and may optionally return a value. *Stored procedures* are database constructs that share many similarities with functions in programming languages. A stored procedure is made up of a set of T-SQL statements that are executed when the stored procedure is called. A stored procedure may accept zero to many input parameters and can return scalar values, output parameters, or, most commonly, result sets from `SELECT` queries.

> **NOTE**
> Stored procedures are oftentimes referred to as sprocs or SPs .

Stored procedures are created using the `CREATE PROCEDURE` T-SQL statement. For example, the following T-SQL script creates a stored procedure named `GetProductsByCategoryID` that takes in a single parameter named `@CategoryID` and returns the `ProductID`, `ProductName`, `UnitPrice`, and `Discontinued` fields of those columns in the `Products` table that have a matching `CategoryID` value:

```
CREATE PROCEDURE GetProductsByCategoryID
(
    @CategoryID int
)
AS
SELECT ProductID, ProductName, UnitPrice, Discontinued
FROM Products
WHERE CategoryID = @CategoryID
```

Once this stored procedure has been created, it can be called using the following syntax:

```
EXEC GetProductsByCategory categoryID
```

> **NOTE**
> In the next tutorial we will examine creating stored procedures through the Visual Studio IDE. For this tutorial, however, we are going to let the TableAdapter wizard automatically generate the stored procedures for us.

In addition to simply returning data, stored procedures are often used to perform multiple database commands within the scope of a single transaction. A stored procedure named `DeleteCategory`, for example, might take in a `@CategoryID` parameter and perform two `DELETE` statements: first, one to delete the related products and a second one deleting the specified category. Multiple statements within a stored procedure are *not* automatically wrapped within a transaction. Additional T-SQL commands need to be issued to ensure the stored procedure s multiple commands are treated as an atomic operation. We'll see how to wrap a stored procedure s commands within the scope of a transaction in the subsequent tutorial.

When using stored procedures within an architecture, the Data Access Layer s methods invoke a particular stored procedure rather than issuing an ad-hoc SQL statement. This centralizes the location of the SQL statements executed (on the database) rather than having it defined within the application s architecture. This centralization arguably makes it easier to find, analyze, and tune the queries and provides a much clearer picture as to where and how the database is being used.

For more information on stored procedure fundamentals, consult the resources in the Further Reading section at the end of this tutorial.

Step 1: Creating the Advanced Data Access Layer Scenarios Web Pages

Before we start our discussion on creating a DAL using stored procedures, let s first take a moment to create the ASP.NET pages in our website project that we will need for this and the next several tutorials. Start by adding a new folder named `AdvancedDAL`. Next, add the following ASP.NET pages to that folder, making sure to associate each page with the `Site.master` master page:

- `Default.aspx`
- `NewSprocs.aspx`
- `ExistingSprocs.aspx`
- `JOINs.aspx`
- `AddingColumns.aspx`
- `ComputedColumns.aspx`
- `EncryptingConfigSections.aspx`
- `ManagedFunctionsAndSprocs.aspx`

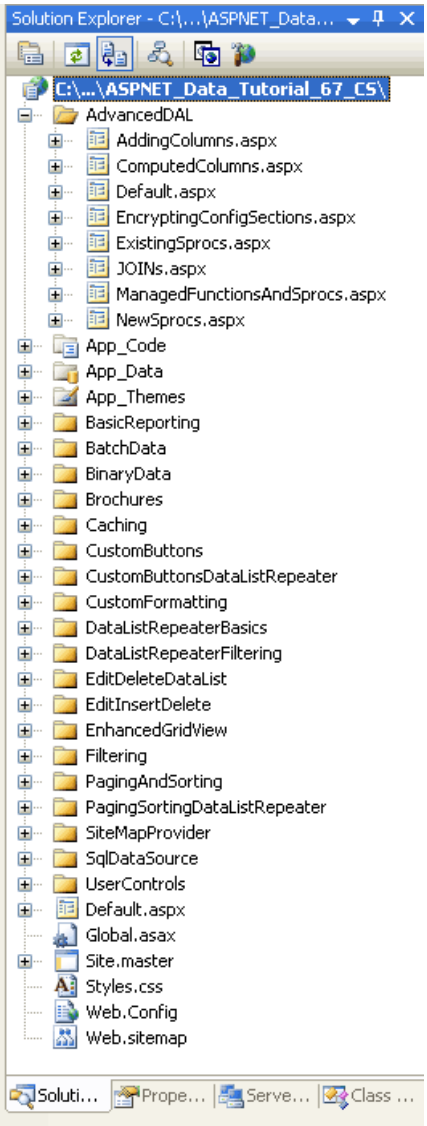


Figure 1: Add the ASP.NET Pages for the Advanced Data Access Layer Scenarios Tutorials

Like in the other folders, `Default.aspx` in the `AdvancedDAL` folder will list the tutorials in its section. Recall that the `SectionLevelTutorialListing.ascx` User Control provides this functionality. Therefore, add this User Control to `Default.aspx` by dragging it from the Solution Explorer onto the page s Design view.

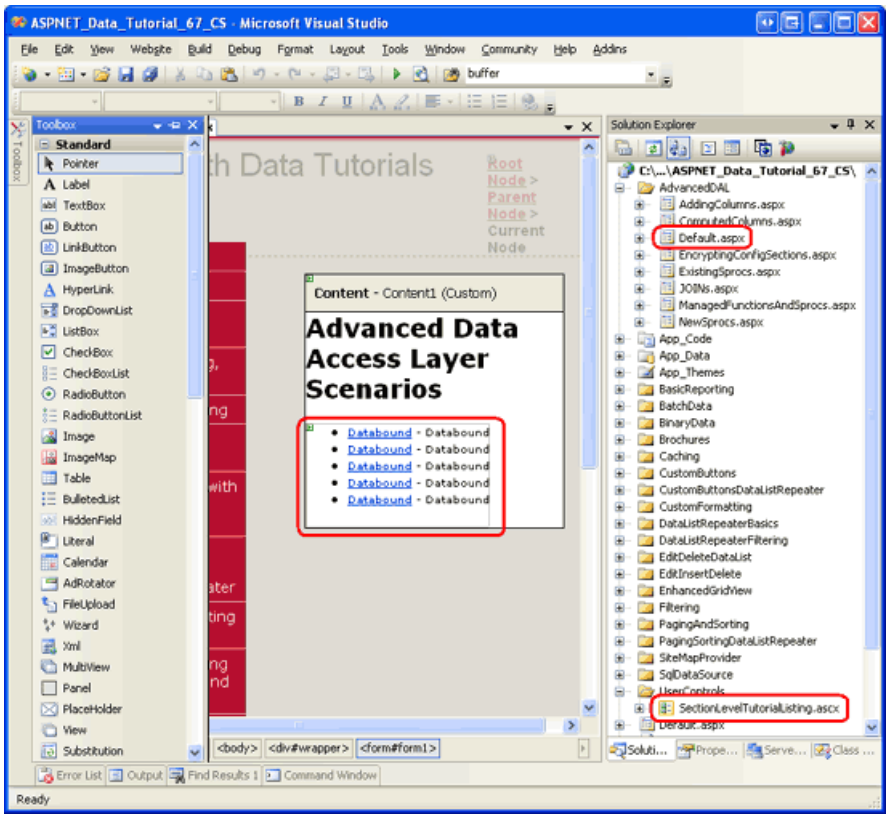


Figure 2: Add the `SectionLevelTutorialListing.ascx` User Control to `Default.aspx` (Click to view full-size image)

Lastly, add these pages as entries to the `Web.sitemap` file. Specifically, add the following markup after the Working with Batched Data `<siteMapNode>`:

```
<siteMapNode url="~/AdvancedDAL/Default.aspx"
    title="Advanced DAL Scenarios"
    description="Explore a number of advanced Data Access Layer scenarios.">

    <siteMapNode url="~/AdvancedDAL/NewSprocs.aspx"
        title="Creating New Stored Procedures for TableAdapters"
        description="Learn how to have the TableAdapter wizard automatically
            create and use stored procedures." />
    <siteMapNode url="~/AdvancedDAL/ExistingSprocs.aspx"
        title="Using Existing Stored Procedures for TableAdapters"
        description="See how to plug existing stored procedures into a
            TableAdapter." />
    <siteMapNode url="~/AdvancedDAL/JOINs.aspx"
        title="Returning Data Using JOINs"
        description="Learn how to augment your DataTables to work with data
            returned from multiple tables via a JOIN query." />
    <siteMapNode url="~/AdvancedDAL/AddingColumns.aspx"
        title="Adding DataColumns to a DataTable"
        description="Master adding new columns to an existing DataTable." />
    <siteMapNode url="~/AdvancedDAL/ComputedColumns.aspx"
        title="Working with Computed Columns"
        description="Explore how to work with computed columns when using
            Typed DataSets." />
    <siteMapNode url="~/AdvancedDAL/EncryptingConfigSections.aspx"
        title="Protected Connection Strings in Web.config"
        description="Protect your connection string information in
            Web.config using encryption." />
    <siteMapNode url="~/AdvancedDAL/ManagedFunctionsAndSprocs.aspx"
        title="Creating Managed SQL Functions and Stored Procedures"
        description="See how to create SQL functions and stored procedures
            using managed code." />
</siteMapNode>
```

After updating `Web.sitemap`, take a moment to view the tutorials website through a browser. The menu on the left now includes items for the advanced DAL scenarios tutorials.

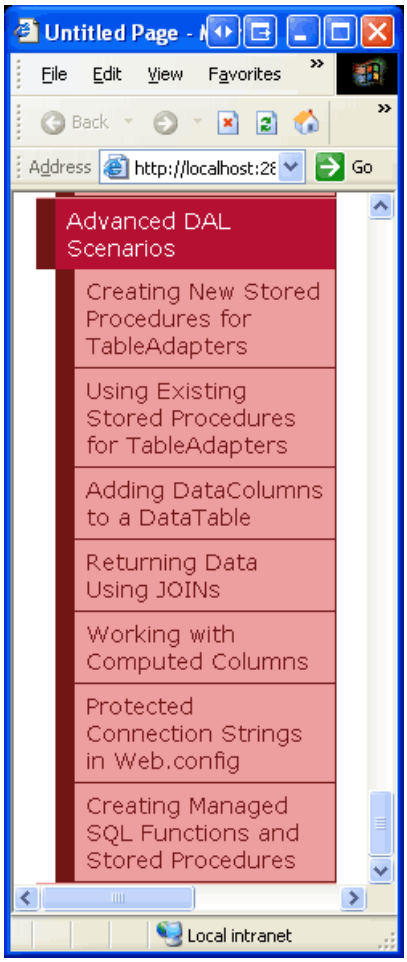


Figure 3: The Site Map Now Includes Entries for the Advanced DAL Scenarios Tutorials

Step 2: Configuring a TableAdapter to Create New Stored Procedures

To demonstrate creating a Data Access Layer that uses stored procedures instead of ad-hoc SQL statements, let s create a new Typed DataSet in the `~/App_Code/DAL` folder named `NorthwindWithSprocs.xsd`. Since we have stepped through this process in detail in previous tutorials, we will proceed quickly through the steps here. If you get stuck or need further step-by-step instructions in creating and configuring a Typed DataSet, refer back to the Creating a Data Access Layer tutorial.

Add a new DataSet to the project by right-clicking on the `DAL` folder, choosing Add New Item, and selecting the DataSet template as shown in Figure 4.

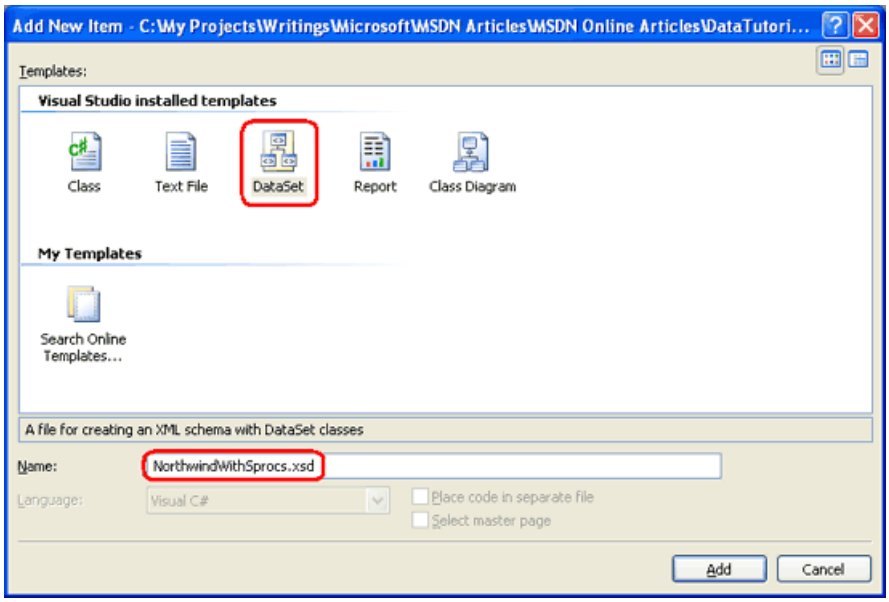


Figure 4: Add a New Typed DataSet to the Project Named `NorthwindWithSprocs.xsd` (Click to view full-size image)

This will create the new Typed DataSet, open its Designer, create a new TableAdapter, and launch the TableAdapter Configuration Wizard. The TableAdapter Configuration Wizard s first step asks us to select the database to work with. The connection string to the Northwind database should be listed in the drop-down list. Select this and click Next.

From this next screen we can choose how the TableAdapter should access the database. In previous tutorials, we selected the first option, Use SQL statements. For this tutorial, select the second option, Create new stored procedures, and click Next.

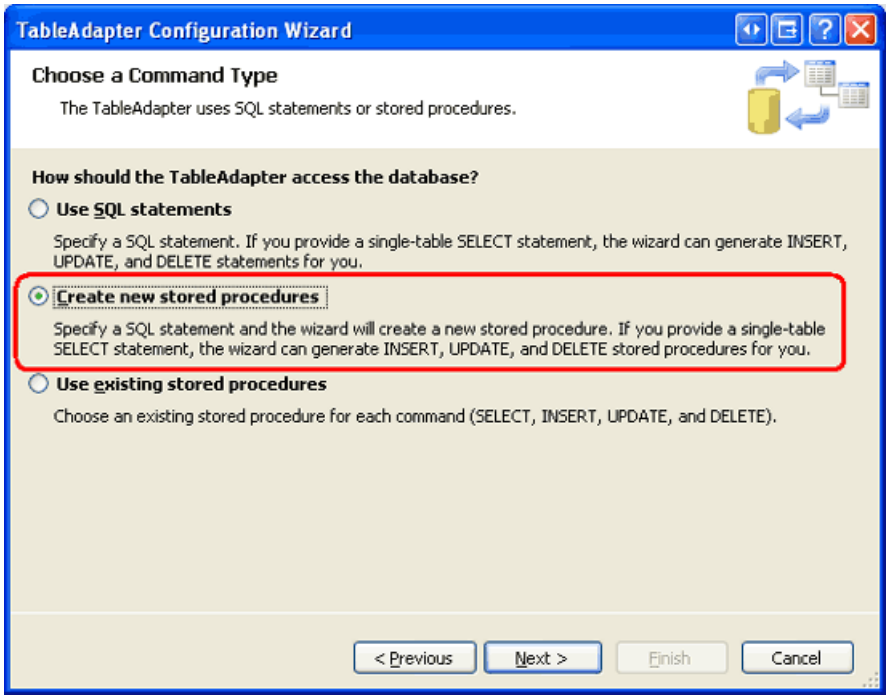


Figure 5: Instruct the TableAdapter to Create New Stored Procedures (Click to view full-size image)

Just like with using ad-hoc SQL statements, in the following step we are asked to provide the `SELECT` statement for the TableAdapter s main query. But instead of using the `SELECT` statement entered here to perform an ad-hoc query directly, the TableAdapter s wizard will create a stored procedure that contains this `SELECT` query.

Use the following `SELECT` query for this TableAdapter:

```
SELECT ProductID, ProductName, SupplierID, CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued
FROM Products
```

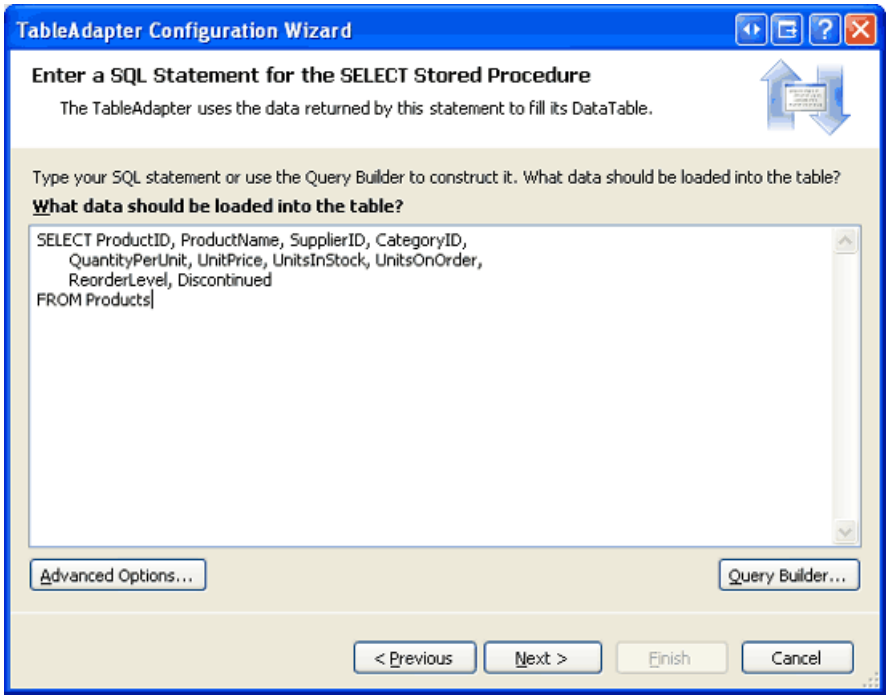


Figure 6: Enter the `SELECT` Query (Click to view full-size image)

> **NOTE**
>
> The query above differs slightly from the main query of the `ProductsTableAdapter` in the `Northwind` Typed DataSet. Recall that the `ProductsTableAdapter` in the `Northwind` Typed DataSet includes two correlated subqueries to bring back the category name and company name for each product s category and supplier. In the upcoming Updating the TableAdapter to Use JOINs tutorial we will look at adding this related data to this TableAdapter.

Take a moment to click the Advanced Options button. From here we can specify whether the wizard should also generate insert, update, and delete statements for the TableAdapter, whether to use optimistic concurrency, and whether the data table should be refreshed after inserts and updates. The Generate Insert, Update and Delete statements option is checked by default. Leave it checked. For this tutorial, leave the Use optimistic concurrency options unchecked.

When having the stored procedures automatically created by the TableAdapter wizard, it appears that the Refresh the data table option is ignored. Regardless of whether this checkbox is checked, the resulting insert and update stored procedures retrieve the just-inserted or just-updated record, as we will see in Step 3.

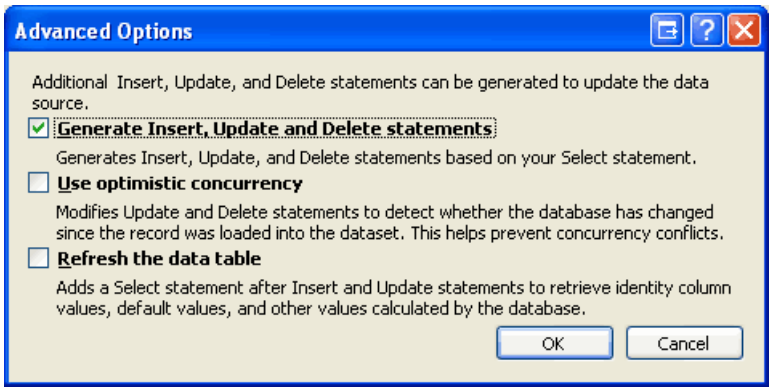


Figure 7: Leave the Generate Insert, Update and Delete statements Option Checked

> **NOTE**
>
> If the Use optimistic concurrency option is checked, the wizard will add additional conditions to the `WHERE` clause that prevent data from being updated if there were changes in other fields. Refer back to the Implementing Optimistic Concurrency tutorial for more information on using the TableAdapter s built-in optimistic concurrency control feature.

After entering the `SELECT` query and confirming that the Generate Insert, Update and Delete statements option is checked, click Next. This next screen, shown in Figure 8, prompts for the names of the stored procedures the wizard will create for selecting, inserting, updating, and deleting data. Change these stored procedures names to `Products_Select`, `Products_Insert`, `Products_Update`, and `Products_Delete`.

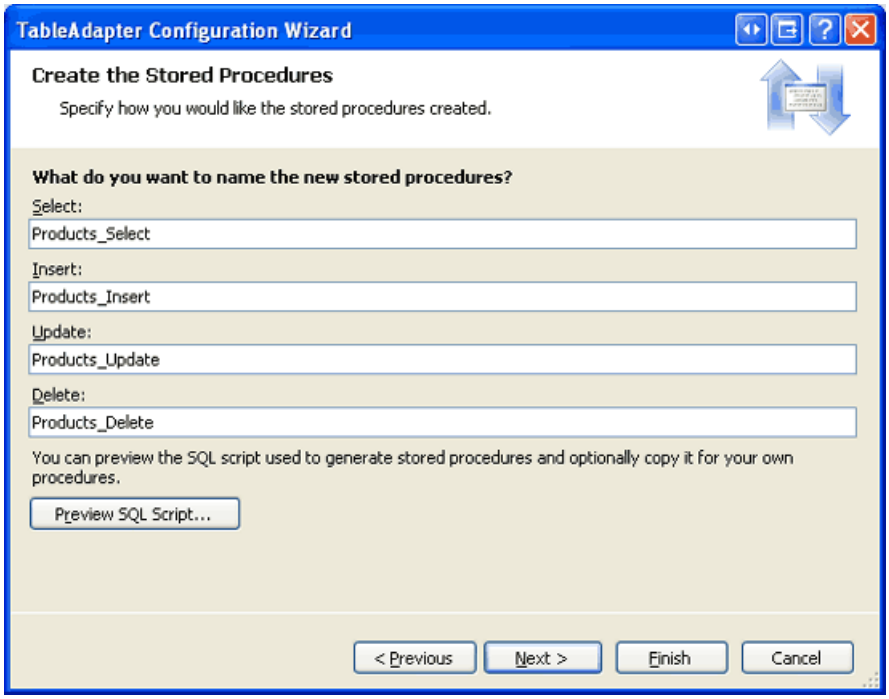


Figure 8: Rename the Stored Procedures (Click to view full-size image)

To see the T-SQL the TableAdapter wizard will use to create the four stored procedures, click the Preview SQL Script button. From the Preview SQL Script dialog box you may save the script to a file or copy it to the clipboard.

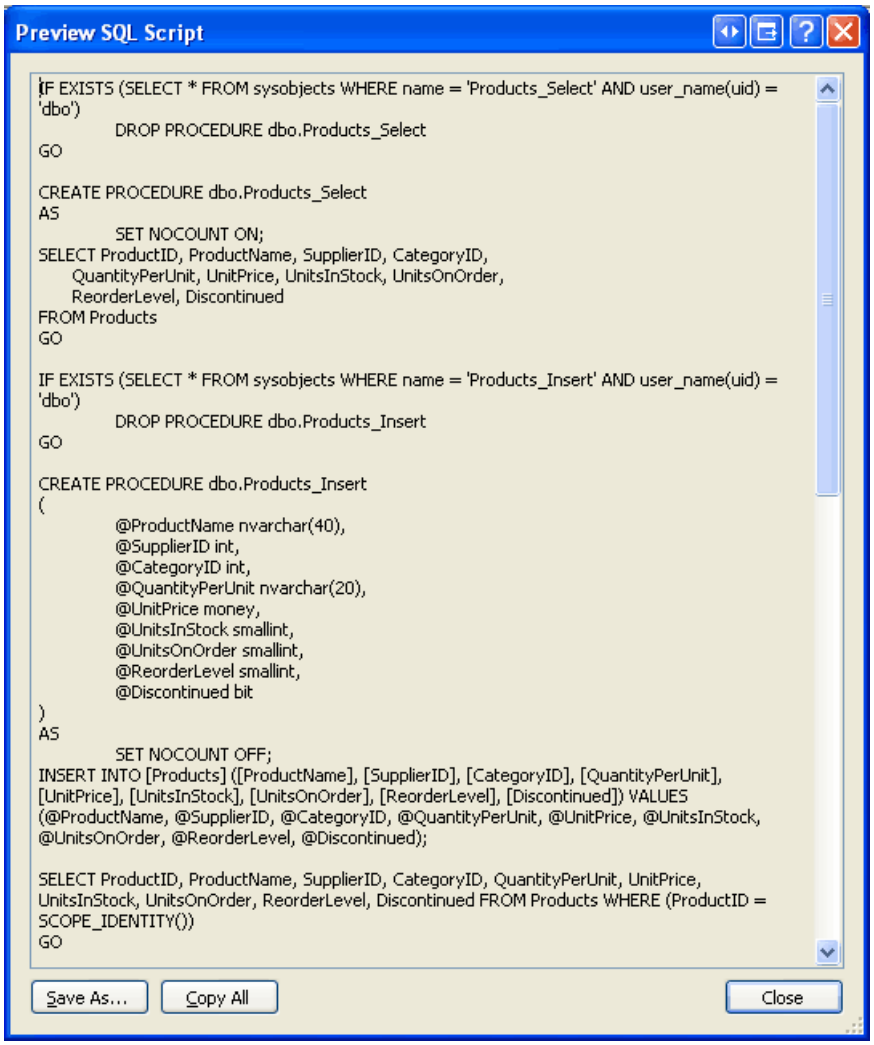


Figure 9: Preview the SQL Script Used to Generate the Stored Procedures

After naming the stored procedures, click Next to name the TableAdapter s corresponding methods. Just like when using ad-hoc SQL statements, we can create methods that fill an existing DataTable or return a new one. We can also specify whether the TableAdapter should include the DB-Direct pattern for inserting, updating, and deleting records. Leave all three checkboxes checked, but rename the Return a DataTable method to `GetProducts` (as shown in Figure 10).

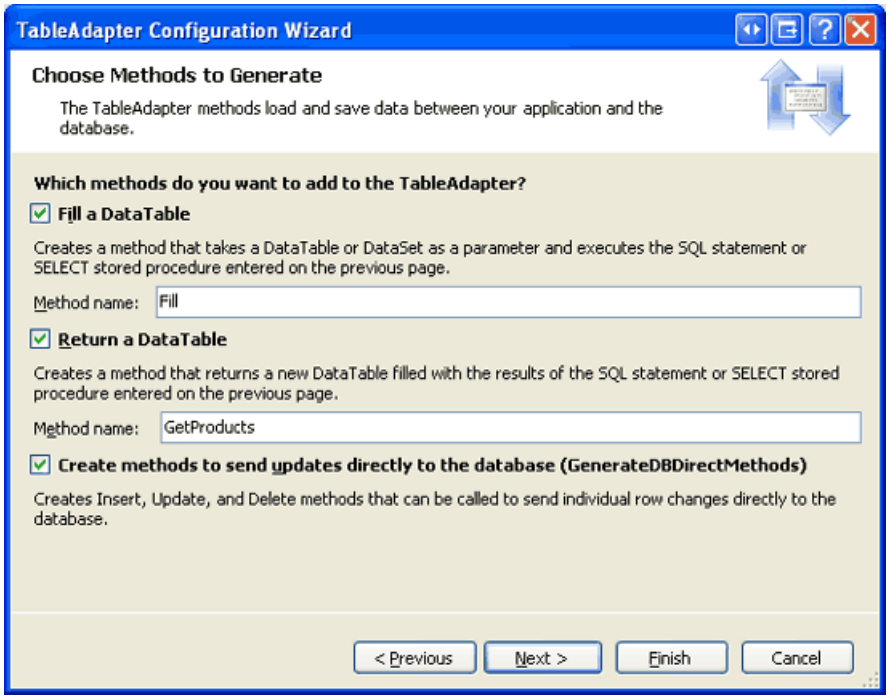


Figure 10: Name the Methods `Fill` and `GetProducts` (Click to view full-size image)

Click Next to see a summary of the steps the wizard will perform. Complete the wizard by clicking the Finish button. Once the wizard completes, you will be returned to the DataSet s Designer, which should now include the `ProductsDataTable`.

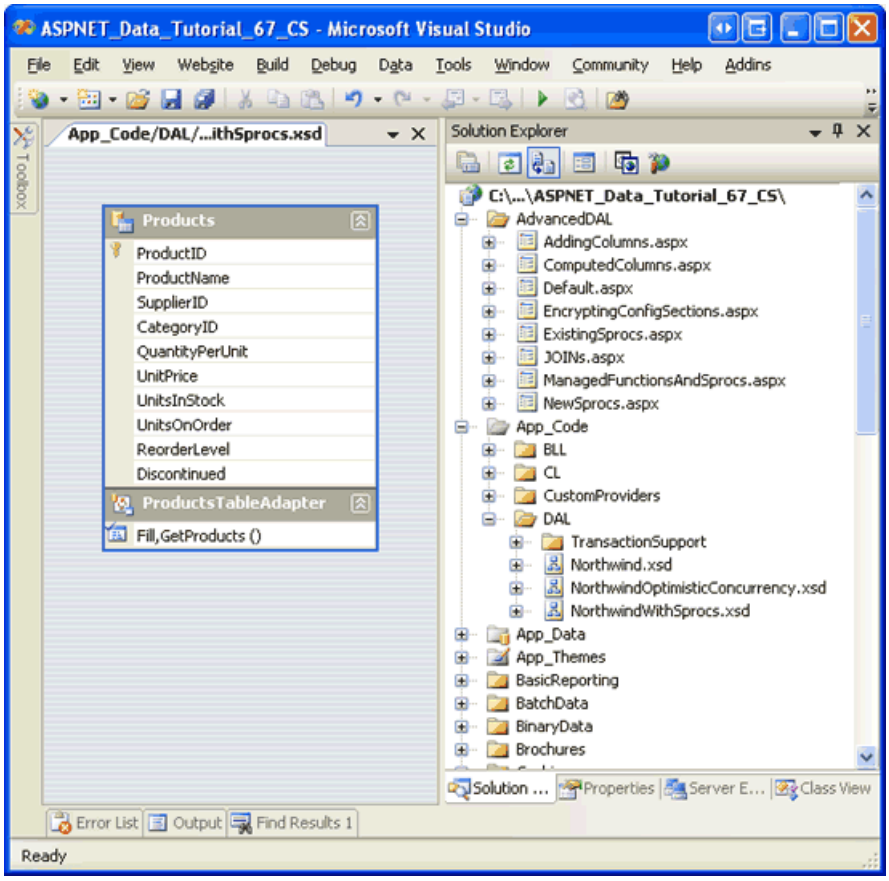


Figure 11: The DataSet s Designer Shows the Newly Added `ProductsDataTable` (Click to view full-size image)

Step 3: Examining the Newly Created Stored Procedures

The TableAdapter wizard used in Step 2 automatically created the stored procedures for selecting, inserting, updating, and deleting data. These stored procedures can be viewed or modified through Visual Studio by going to the Server Explorer and drilling down into the database s Stored Procedures folder. As Figure 12 shows, the Northwind database contains four new stored procedures: `Products_Delete`, `Products_Insert`, `Products_Select`, and `Products_Update`.

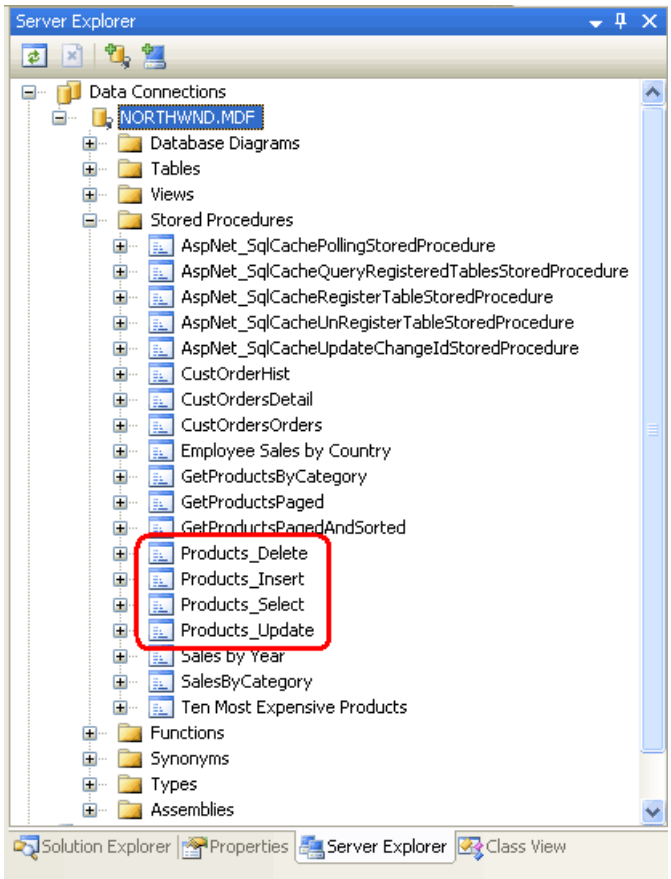


Figure 12: The Four Stored Procedures Created in Step 2 Can Be Found in the Database s Stored Procedures Folder

> **NOTE**
>
> If you do not see the Server Explorer, go to the View menu and choose the Server Explorer option. If you do not see the product-related stored procedures added from Step 2, try right-clicking on the Stored Procedures folder and choosing Refresh.

To view or modify a stored procedure, double-click its name in the Server Explorer or, alternatively, right-click on the stored procedure and choose Open. Figure 13 shows the `Products_Delete` stored procedure, when opened.

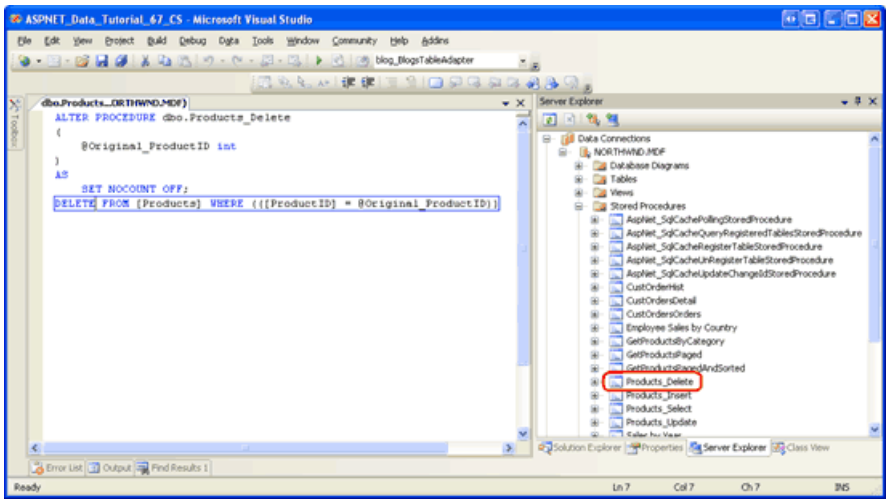


Figure 13: Stored Procedures Can Be Opened and Modified From Within Visual Studio (Click to view full-size image)

The contents of both the `Products_Delete` and `Products_Select` stored procedures are quite straightforward. The `Products_Insert` and `Products_Update` stored procedures, on the other hand, warrant a closer inspection as they

both perform a `SELECT` statement after their `INSERT` and `UPDATE` statements. For example, the following SQL makes up the `Products_Insert` stored procedure:

```
ALTER PROCEDURE dbo.Products_Insert
(
    @ProductName nvarchar(40),
    @SupplierID int,
    @CategoryID int,
    @QuantityPerUnit nvarchar(20),
    @UnitPrice money,
    @UnitsInStock smallint,
    @UnitsOnOrder smallint,
    @ReorderLevel smallint,
    @Discontinued bit
)
AS
    SET NOCOUNT OFF;
INSERT INTO [Products] ([ProductName], [SupplierID], [CategoryID], [QuantityPerUnit],
    [UnitPrice], [UnitsInStock], [UnitsOnOrder], [ReorderLevel], [Discontinued])
VALUES (@ProductName, @SupplierID, @CategoryID, @QuantityPerUnit, @UnitPrice,
    @UnitsInStock, @UnitsOnOrder, @ReorderLevel, @Discontinued);

SELECT ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit, UnitPrice,
    UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued
FROM Products
WHERE (ProductID = SCOPE_IDENTITY())
```

The stored procedure accepts as input parameters the `Products` columns that were returned by the `SELECT` query specified in the TableAdapter s wizard and these values are used in an `INSERT` statement. Following the `INSERT` statement, a `SELECT` query is used to return the `Products` column values (including the `ProductID`) of the newly added record. This refresh capability is useful when adding a new record using the Batch Update pattern as it automatically updates the newly added `ProductRow` instances `ProductID` properties with the auto-incremented values assigned by the database.

The following code illustrates this feature. It contains a `ProductsTableAdapter` and `ProductsDataTable` created for the `NorthwindWithSprocs` Typed DataSet. A new product is added to the database by creating a `ProductsRow` instance, supplying its values, and calling the TableAdapter s `Update` method, passing in the `ProductsDataTable`. Internally, the TableAdapter s `Update` method enumerates the `ProductsRow` instances in the passed-in DataTable (in this example there is only one - the one we just added), and performs the appropriate insert, update, or delete command. In this case, the `Products_Insert` stored procedure is executed, which adds a new record to the `Products` table and returns the details of the newly-added record. The `ProductsRow` instance s `ProductID` value is then updated. After the `Update` method has completed, we can access the newly-added record s `ProductID` value through the `ProductsRow` s `ProductID` property.

```
' Create the ProductsTableAdapter and ProductsDataTable
Dim productsAPI As New NorthwindWithSprocsTableAdapters.ProductsTableAdapter
Dim products As New NorthwindWithSprocs.ProductsDataTable
' Create a new ProductsRow instance and set its properties
Dim product As NorthwindWithSprocs.ProductsRow = products.NewProductsRow()
product.ProductName = "New Product"
product.CategoryID = 1  ' Beverages
product.Discontinued = False
' Add the ProductsRow instance to the DataTable
products.AddProductsRow(product)
' Update the DataTable using the Batch Update pattern
productsAPI.Update(products)
' At this point, we can determine the value of the newly-added record's ProductID
Dim newlyAddedProductIDValue as Integer = product.ProductID
```

The `Products_Update` stored procedure similarly includes a `SELECT` statement after its `UPDATE` statement.

```
ALTER PROCEDURE dbo.Products_Update
(
    @ProductName nvarchar(40),
    @SupplierID int,
    @CategoryID int,
    @QuantityPerUnit nvarchar(20),
    @UnitPrice money,
    @UnitsInStock smallint,
    @UnitsOnOrder smallint,
    @ReorderLevel smallint,
    @Discontinued bit,
    @Original_ProductID int,
    @ProductID int
)
AS
    SET NOCOUNT OFF;
UPDATE [Products]
SET [ProductName] = @ProductName, [SupplierID] = @SupplierID,
    [CategoryID] = @CategoryID, [QuantityPerUnit] = @QuantityPerUnit,
    [UnitPrice] = @UnitPrice, [UnitsInStock] = @UnitsInStock,
    [UnitsOnOrder] = @UnitsOnOrder, [ReorderLevel] = @ReorderLevel,
    [Discontinued] = @Discontinued
WHERE (([ProductID] = @Original_ProductID));

SELECT ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit,
    UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued
FROM Products
WHERE (ProductID = @ProductID)
```

Note that this stored procedure includes two input parameters for `ProductID`: `@Original_ProductID` and `@ProductID`. This functionality allows for scenarios where the primary key might be changed. For example, in an employee database, each employee record might use the employee s social security number as their primary key. In order to change an existing employee s social security number, both the new social security number and the original one must be supplied. For the `Products` table, such functionality is not needed because the `ProductID` column is an `IDENTITY` column and should never be changed. In fact, the `UPDATE` statement in the `Products_Update` stored procedure doesn t include the `ProductID` column in its column list. So, while `@Original_ProductID` is used in the `UPDATE` statement s `WHERE` clause, it is superfluous for the `Products` table and could be replaced by the `@ProductID` parameter. When modifying a stored procedure s parameters it is important that the TableAdapter method(s) that use that stored procedure are also updated.

Step 4: Modifying a Stored Procedure s Parameters and Updating the TableAdapter

Since the `@Original_ProductID` parameter is superfluous, let s remove it from the `Products_Update` stored procedure altogether. Open the `Products_Update` stored procedure, delete the `@Original_ProductID` parameter, and, in the `WHERE` clause of the `UPDATE` statement, change the parameter name used from `@Original_ProductID` to `@ProductID`. After making these changes, the T-SQL within the stored procedure should look like the following:

```
ALTER PROCEDURE dbo.Products_Update
(
    @ProductName nvarchar(40),
    @SupplierID int,
    @CategoryID int,
    @QuantityPerUnit nvarchar(20),
    @UnitPrice money,
    @UnitsInStock smallint,
    @UnitsOnOrder smallint,
    @ReorderLevel smallint,
    @Discontinued bit,
    @ProductID int
)
AS
    SET NOCOUNT OFF;
UPDATE [Products] SET [ProductName] = @ProductName, [SupplierID] = @SupplierID,
    [CategoryID] = @CategoryID, [QuantityPerUnit] = @QuantityPerUnit,
    [UnitPrice] = @UnitPrice, [UnitsInStock] = @UnitsInStock,
    [UnitsOnOrder] = @UnitsOnOrder, [ReorderLevel] = @ReorderLevel,
    [Discontinued] = @Discontinued
WHERE (([ProductID] = @ProductID));

SELECT ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit,
    UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued
FROM Products
WHERE (ProductID = @ProductID)
```

To save these changes to the database, click the Save icon in the toolbar or hit Ctrl+S. At this point, the `Products_Update` stored procedure does not expect an `@Original_ProductID` input parameter, but the TableAdapter is configured to pass such a parameter. You can see the parameters the TableAdapter will send to the `Products_Update` stored procedure by selecting the TableAdapter in the DataSet Designer, going to the Properties window, and clicking the ellipses in the `UpdateCommand` s `Parameters` collection. This brings up the Parameters Collection Editor dialog box shown in Figure 14.

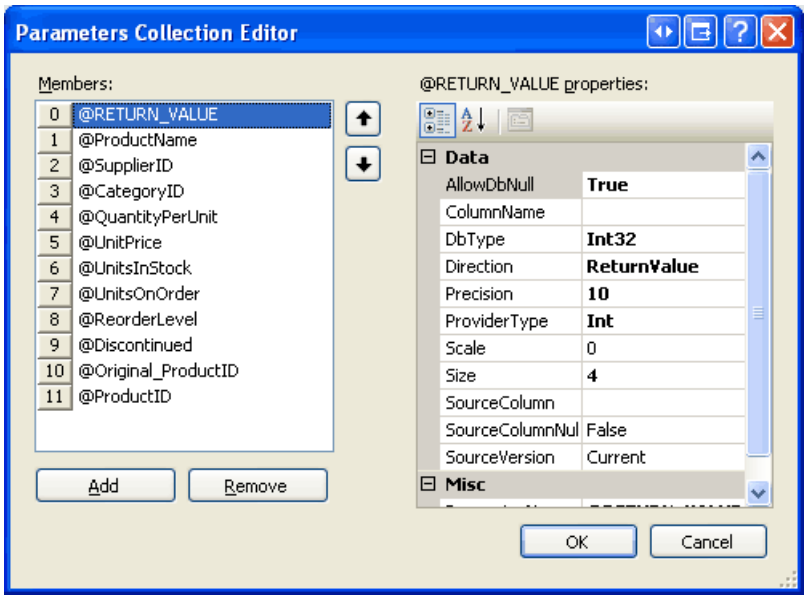


Figure 14: The Parameters Collection Editor Lists the Parameters Used Passed to the `Products_Update` Stored Procedure

You can remove this parameter from here by simply selecting the `@Original_ProductID` parameter from the list of members and clicking the Remove button.

Alternatively, you can refresh the parameters used for all of the methods by right-clicking on the TableAdapter in the Designer and choosing Configure. This will bring up the TableAdapter Configuration wizard, listing the stored procedures used for selecting, inserting, updating, and deleting, along with the parameters the stored procedures

expect to receive. If you click on the Update drop-down list you can see the `Products_Update` stored procedures expected input parameters, which now no longer includes `@Original_ProductID` (see Figure 15). Simply click Finish to automatically update the parameter collection used by the TableAdapter.

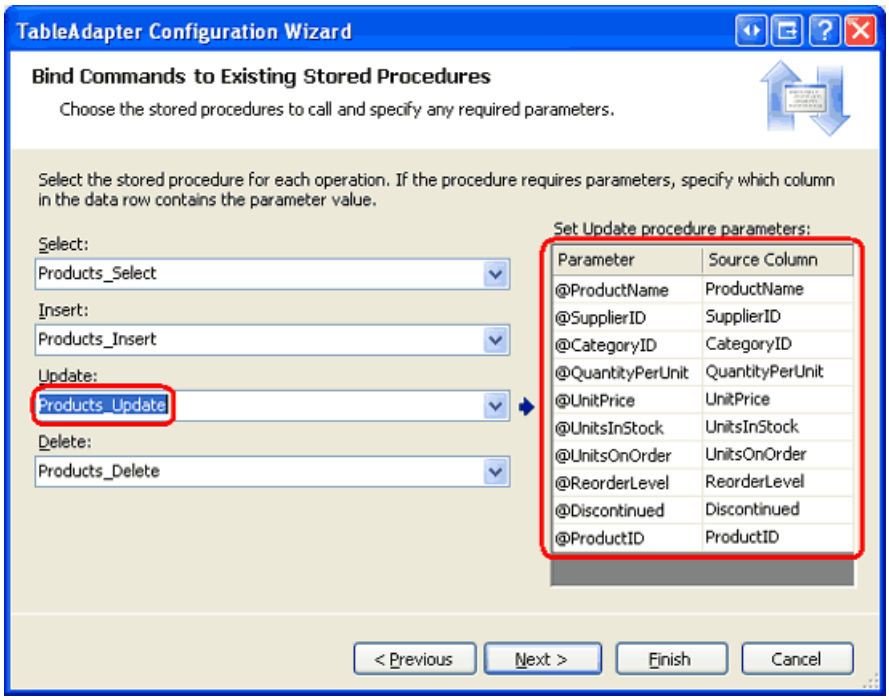


Figure 15: You Can Alternatively Use the TableAdapter s Configuration Wizard to Refresh Its Methods Parameter Collections (Click to view full-size image)

Step 5: Adding Additional TableAdapter Methods

As Step 2 illustrated, when creating a new TableAdapter it is easy to have the corresponding stored procedures automatically generated. The same is true when adding additional methods to a TableAdapter. To illustrate this, let s add a `GetProductByProductID(productID)` method to the `ProductsTableAdapter` created in Step 2. This method will take as input a `ProductID` value and return details about the specified product.

Start by right-clicking on the TableAdapter and choosing Add Query from the context menu.

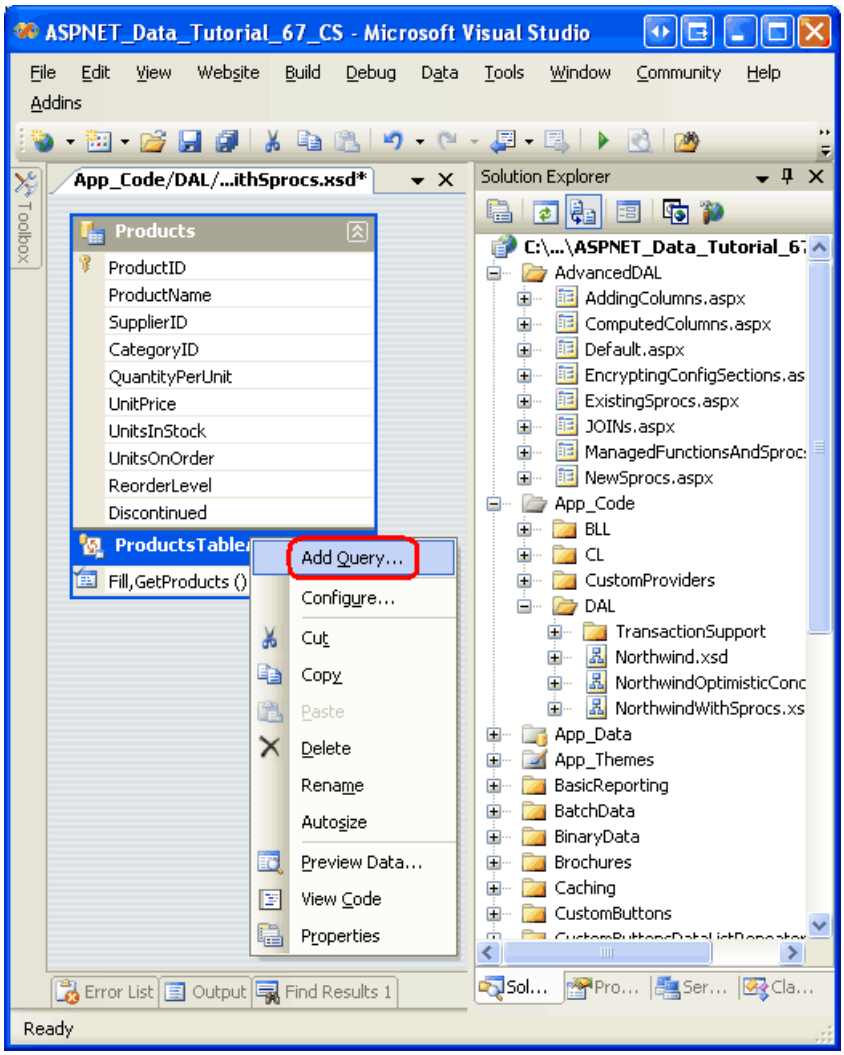


Figure 16: Add a New Query to the TableAdapter

This will start the TableAdapter Query Configuration wizard, which first prompts for how the TableAdapter should access the database. To have a new stored procedure created, choose the Create a new stored procedure option and click Next.

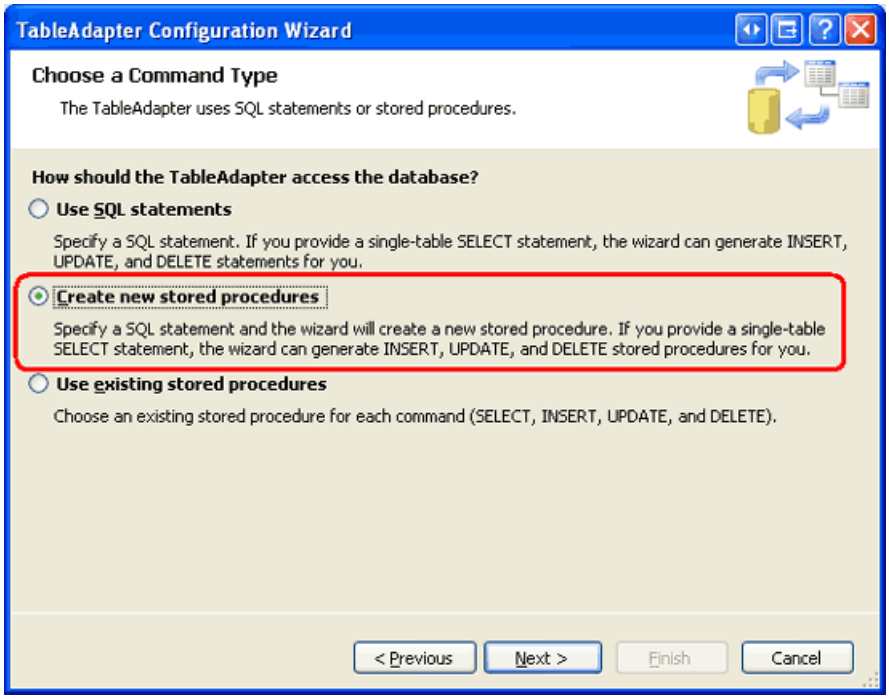


Figure 17: Choose the Create a new stored procedure Option (Click to view full-size image)

The next screen asks us to identify the type of query to execute, whether it will return a set of rows or a single scalar value, or perform an `UPDATE`, `INSERT`, or `DELETE` statement. Since the `GetProductByProductID(productID)` method will return a row, leave the SELECT which returns row option selected and hit Next.

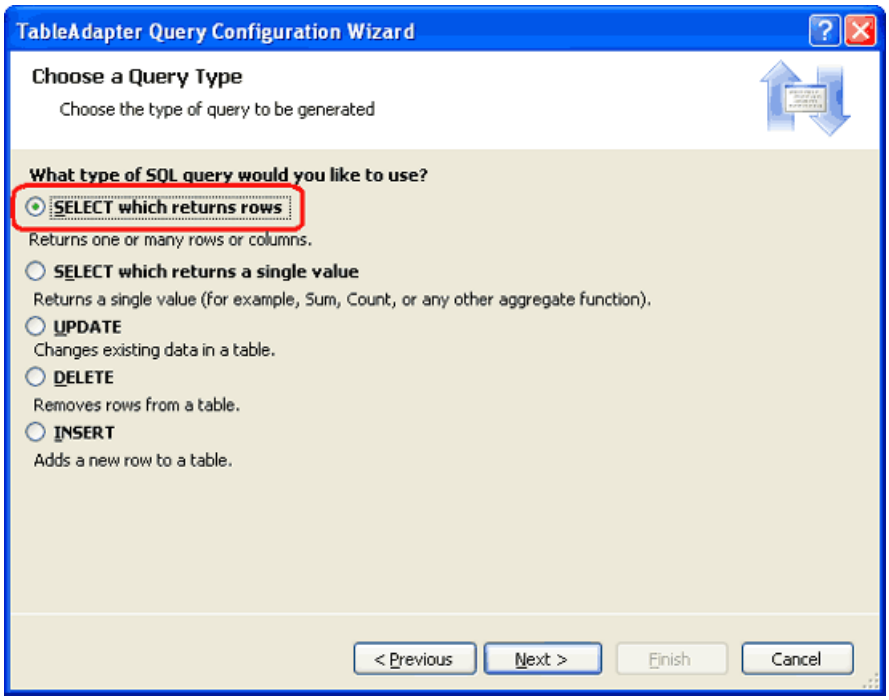


Figure 18: Choose the SELECT which returns row Option (Click to view full-size image)

The next screen displays the TableAdapter s main query, which just lists the name of the stored procedure (`dbo.Products_Select`). Replace the stored procedure name with the following `SELECT` statement, which returns all of the product fields for a specified product:

```
SELECT ProductID, ProductName, SupplierID, CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued
FROM Products
WHERE ProductID = @ProductID
```

Figure 19: Replace the Stored Procedure Name with a `SELECT` Query (Click to view full-size image)

The subsequent screen asks you to name the stored procedure that will be created. Enter the name `Products_SelectByProductID` and click Next.

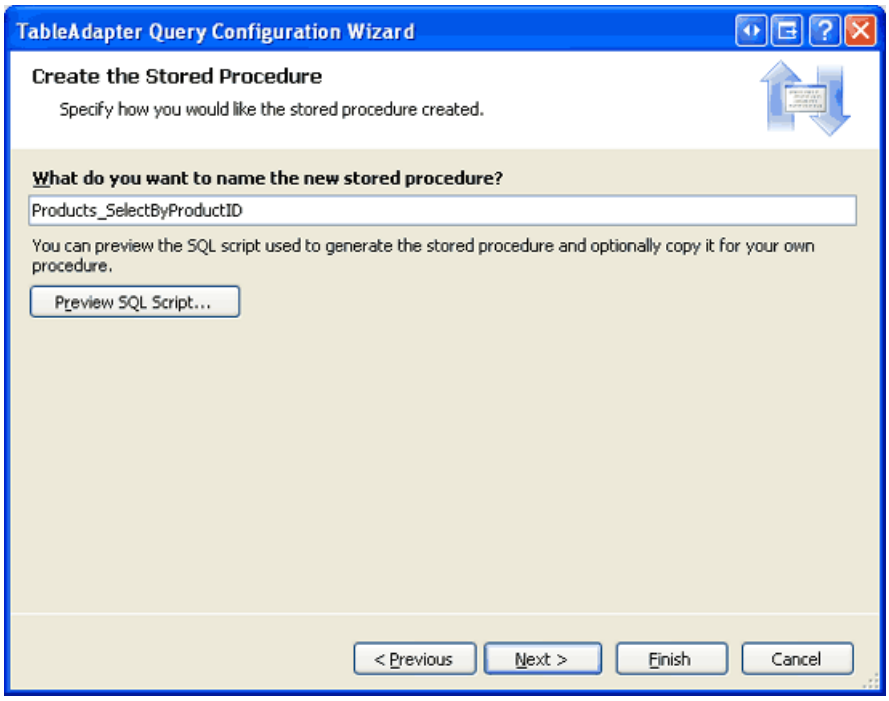


Figure 20: Name the New Stored Procedure `Products_SelectByProductID` (Click to view full-size image)

The final step of the wizard allows us to change the method names generated as well as indicate whether to use the Fill a DataTable pattern, Return a DataTable pattern, or both. For this method, leave both options checked, but rename the methods to `FillByProductID` and `GetProductByProductID`. Click Next to view a summary of the steps the wizard will perform and then click Finish to complete the wizard.

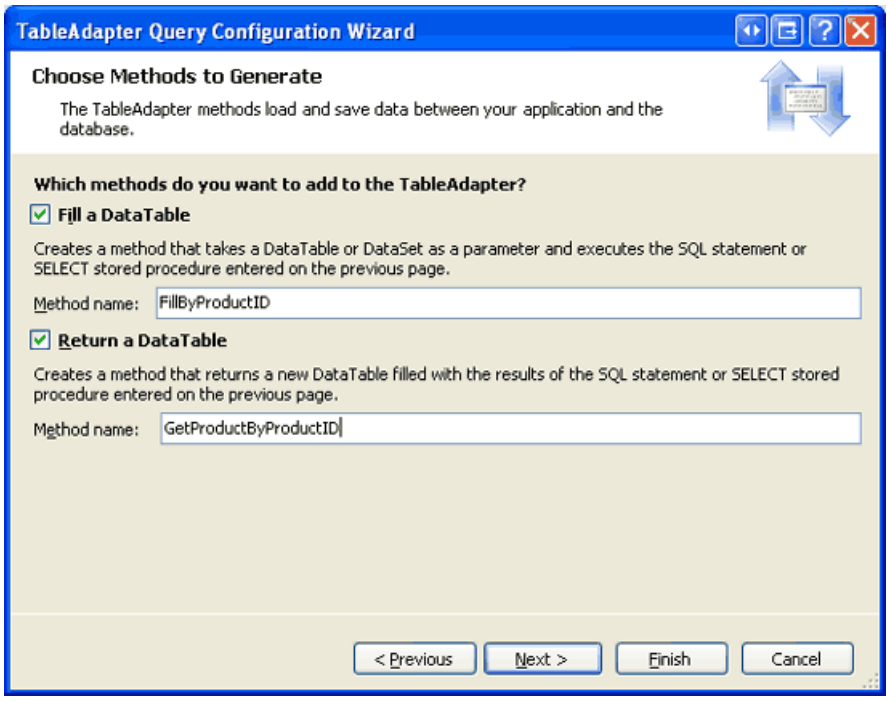


Figure 21: Rename the TableAdapter s Methods to `FillByProductID` and `GetProductByProductID` (Click to view full-size image)

After completing the wizard, the TableAdapter has a new method available, `GetProductByProductID(productID)` that, when invoked, will execute the `Products_SelectByProductID` stored procedure that was just created. Take a moment to view this new stored procedure from the Server Explorer by drilling into the Stored Procedures folder and opening `Products_SelectByProductID` (if you do not see it, right-click on the Stored Procedures folder and

choose Refresh).

Note that the `SelectByProductID` stored procedure takes `@ProductID` as an input parameter and executes the `SELECT` statement that we entered in the wizard.

```
ALTER PROCEDURE dbo.Products_SelectByProductID
(
    @ProductID int
)
AS
    SET NOCOUNT ON;
SELECT ProductID, ProductName, SupplierID, CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued
FROM Products
WHERE ProductID = @ProductID
```

Step 6: Creating a Business Logic Layer Class

Throughout the tutorial series we have strived to maintain a layered architecture in which the Presentation Layer made all of its calls to the Business Logic Layer (BLL). In order to adhere to this design decision, we first need to create a BLL class for the new Typed DataSet before we can access product data from the Presentation Layer.

Create a new class file named `ProductsBLLWithSprocs.vb` in the `~/App_Code/BLL` folder and add to it the following code:

```
Imports NorthwindWithSprocsTableAdapters
<System.ComponentModel.DataObject()> _
Public Class ProductsBLLWithSprocs
    Private _productsAdapter As ProductsTableAdapter = Nothing
    Protected ReadOnly Property Adapter() As ProductsTableAdapter
        Get
            If _productsAdapter Is Nothing Then
                _productsAdapter = New ProductsTableAdapter()
            End If
            Return _productsAdapter
        End Get
    End Property
    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Select, True)> _
    Public Function GetProducts() As NorthwindWithSprocs.ProductsDataTable
        Return Adapter.GetProducts()
    End Function
    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Select, False)> _
    Public Function GetProductByProductID(ByVal productID As Integer) _
        As NorthwindWithSprocs.ProductsDataTable
        Return Adapter.GetProductByProductID(productID)
    End Function
    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Insert, True)> _
    Public Function AddProduct _
        (ByVal productName As String, ByVal supplierID As Nullable(Of Integer), _
         ByVal categoryID As Nullable(Of Integer), ByVal quantityPerUnit As String, _
         ByVal unitPrice As Nullable(Of Decimal), _
         ByVal unitsInStock As Nullable(Of Short), _
         ByVal unitsOnOrder As Nullable(Of Short), _
         ByVal reorderLevel As Nullable(Of Short), _
         ByVal discontinued As Boolean) _
         As Boolean

        ' Create a new ProductRow instance
        Dim products As New NorthwindWithSprocs.ProductsDataTable()
        Dim product As NorthwindWithSprocs.ProductsRow = products.NewProductsRow()
```

```
        Dim product As NorthwindWithSprocs.ProductsRow = products.NewProductsRow()
        product.ProductName = productName
        If Not supplierID.HasValue Then
            product.SetSupplierIDNull()
        Else
            product.SupplierID = supplierID.Value
        End If
        If Not categoryID.HasValue Then
            product.SetCategoryIDNull()
        Else
            product.CategoryID = categoryID.Value
        End If
        If quantityPerUnit Is Nothing Then
            product.SetQuantityPerUnitNull()
        Else
            product.QuantityPerUnit = quantityPerUnit
        End If
        If Not unitPrice.HasValue Then
            product.SetUnitPriceNull()
        Else
            product.UnitPrice = unitPrice.Value
        End If
        If Not unitsInStock.HasValue Then
            product.SetUnitsInStockNull()
        Else
            product.UnitsInStock = unitsInStock.Value
        End If
        If Not unitsOnOrder.HasValue Then
            product.SetUnitsOnOrderNull()
        Else
            product.UnitsOnOrder = unitsOnOrder.Value
        End If
        If Not reorderLevel.HasValue Then
            product.SetReorderLevelNull()
        Else
            product.ReorderLevel = reorderLevel.Value
        End If
        product.Discontinued = discontinued
        ' Add the new product
        products.AddProductsRow(product)
        Dim rowsAffected As Integer = Adapter.Update(products)
        ' Return true if precisely one row was inserted, otherwise false
        Return rowsAffected = 1
    End Function
    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Update, True)> _
    Public Function UpdateProduct
        (ByVal productName As String, ByVal supplierID As Nullable(Of Integer), _
         ByVal categoryID As Nullable(Of Integer), ByVal quantityPerUnit As String, _
         ByVal unitPrice As Nullable(Of Decimal), _
         ByVal unitsInStock As Nullable(Of Short), _
         ByVal unitsOnOrder As Nullable(Of Short), _
         ByVal reorderLevel As Nullable(Of Short), _
         ByVal discontinued As Boolean, ByVal productID As Integer) _
         As Boolean

        Dim products As NorthwindWithSprocs.ProductsDataTable = _
            Adapter.GetProductByProductID(productID)
        If products.Count = 0 Then
            ' no matching record found, return false
            Return False
        End If
        Dim product As NorthwindWithSprocs.ProductsRow = products(0)
        product.ProductName = productName
        If Not supplierID.HasValue Then
            product.SetSupplierIDNull()
        Else
            product.SupplierID = supplierID.Value
        End If
```

```
        If Not categoryID.HasValue Then
            product.SetCategoryIDNull()
        Else
            product.CategoryID = categoryID.Value
        End If
        If quantityPerUnit Is Nothing Then
            product.SetQuantityPerUnitNull()
        Else
            product.QuantityPerUnit = quantityPerUnit
        End If
        If Not unitPrice.HasValue Then
            product.SetUnitPriceNull()
        Else
            product.UnitPrice = unitPrice.Value
        End If
        If Not unitsInStock.HasValue Then
            product.SetUnitsInStockNull()
        Else
            product.UnitsInStock = unitsInStock.Value
        End If
        If Not unitsOnOrder.HasValue Then
            product.SetUnitsOnOrderNull()
        Else
            product.UnitsOnOrder = unitsOnOrder.Value
        End If
        If Not reorderLevel.HasValue Then
            product.SetReorderLevelNull()
        Else
            product.ReorderLevel = reorderLevel.Value
        End If
        product.Discontinued = discontinued
        ' Update the product record
        Dim rowsAffected As Integer = Adapter.Update(product)
        ' Return true if precisely one row was updated, otherwise false
        Return rowsAffected = 1
    End Function
    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Delete, True)> _
    Public Function DeleteProduct(ByVal productID As Integer) As Boolean
        Dim rowsAffected As Integer = Adapter.Delete(productID)
        ' Return true if precisely one row was deleted, otherwise false
        Return rowsAffected = 1
    End Function
End Class
```

This class mimics the `ProductsBLL` class semantics from earlier tutorials, but uses the `ProductsTableAdapter` and `ProductsDataTable` objects from the `NorthwindWithSprocs` DataSet. For example, rather than having a `Imports NorthwindTableAdapters` statement at the start of the class file as `ProductsBLL` does, the `ProductsBLLWithSprocs` class uses `Imports NorthwindWithSprocsTableAdapters`. Likewise, the `ProductsDataTable` and `ProductsRow` objects used in this class are prefixed with the `NorthwindWithSprocs` namespace. The `ProductsBLLWithSprocs` class provides two data access methods, `GetProducts` and `GetProductByProductID`, and methods to add, update, and delete a single product instance.

Step 7: Working with the `NorthwindWithSprocs` DataSet from the Presentation Layer

At this point we have created a DAL that uses stored procedures to access and modify the underlying database data. We have also built a rudimentary BLL with methods to retrieve all products or a particular product along with methods for adding, updating, and deleting products. To round off this tutorial, let s create an ASP.NET page that uses the BLL s `ProductsBLLWithSprocs` class for displaying, updating, and deleting records.

Open the `NewSprocs.aspx` page in the `AdvancedDAL` folder and drag a GridView from the Toolbox onto the

Designer, naming it `Products`. From the GridView s smart tag choose to bind it to a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to use the `ProductsBLLWithSprocs` class, as shown in Figure 22.

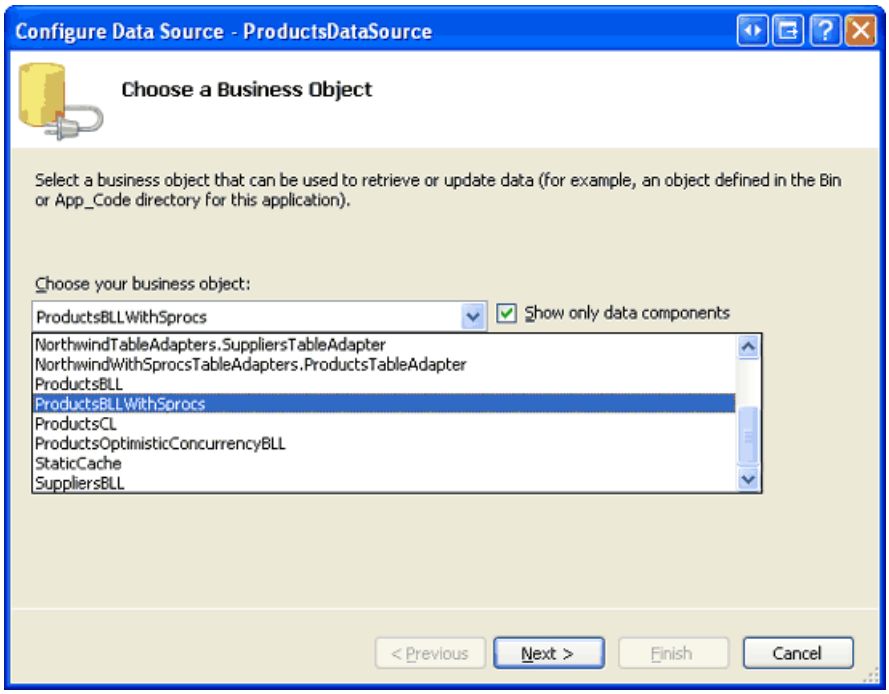


Figure 22: Configure the ObjectDataSource to Use the `ProductsBLLWithSprocs` Class (Click to view full-size image)

The drop-down list in the SELECT tab has two options, `GetProducts` and `GetProductByProductID`. Since we want to display all products in the GridView, choose the `GetProducts` method. The drop-down lists in the UPDATE, INSERT, and DELETE tabs each only have one method. Ensure that each of these drop-down lists has its appropriate method selected and then click Finish.

After the ObjectDataSource wizard has completed, Visual Studio will add BoundFields and a CheckBoxField to the GridView for the product data fields. Turn on the GridView s built-in editing and deleting features by checking the Enable Editing and Enable Deleting options present in the smart tag.

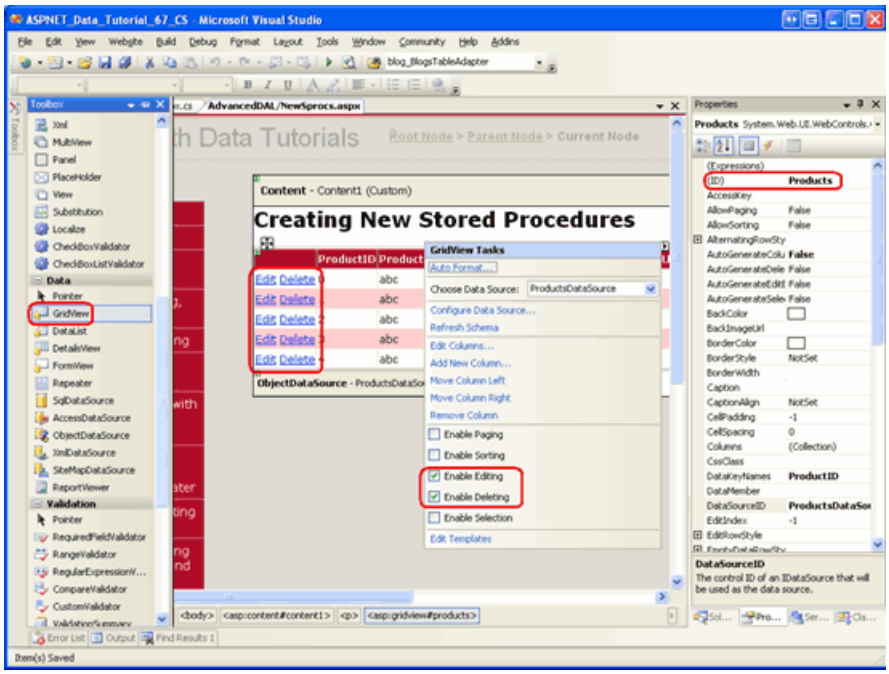


Figure 23: The Page Contains a GridView with Editing and Deleting Support Enabled (Click to view full-size image)

As we ve discussed in previous tutorials, at the completion of the ObjectDataSource s wizard, Visual Studio sets the `OldValuesParameterFormatString` property to original_{0}. This needs to be reverted to its default value of {0} in order for the data modification features to work properly given the parameters expected by the methods in our BLL. Therefore, be sure to set the `OldValuesParameterFormatString` property to {0} or remove the property altogether from the declarative syntax.

After completing the Configure Data Source wizard, turning on editing and deleting support in the GridView, and returning the ObjectDataSource s `OldValuesParameterFormatString` property to its default value, your page s declarative markup should look similar to the following:

```
<asp:GridView ID="Products" runat="server" AutoGenerateColumns="False"
    DataKeyNames="ProductID" DataSourceID="ProductsDataSource">
    <Columns>
        <asp:CommandField ShowDeleteButton="True" ShowEditButton="True" />
        <asp:BoundField DataField="ProductID" HeaderText="ProductID"
            InsertVisible="False" ReadOnly="True"
            SortExpression="ProductID" />
        <asp:BoundField DataField="ProductName" HeaderText="ProductName"
            SortExpression="ProductName" />
        <asp:BoundField DataField="SupplierID" HeaderText="SupplierID"
            SortExpression="SupplierID" />
        <asp:BoundField DataField="CategoryID" HeaderText="CategoryID"
            SortExpression="CategoryID" />
        <asp:BoundField DataField="QuantityPerUnit" HeaderText="QuantityPerUnit"
            SortExpression="QuantityPerUnit" />
        <asp:BoundField DataField="UnitPrice" HeaderText="UnitPrice"
            SortExpression="UnitPrice" />
        <asp:BoundField DataField="UnitsInStock" HeaderText="UnitsInStock"
            SortExpression="UnitsInStock" />
        <asp:BoundField DataField="UnitsOnOrder" HeaderText="UnitsOnOrder"
            SortExpression="UnitsOnOrder" />
        <asp:BoundField DataField="ReorderLevel" HeaderText="ReorderLevel"
            SortExpression="ReorderLevel" />
        <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    DeleteMethod="DeleteProduct" InsertMethod="AddProduct"
    SelectMethod="GetProducts" TypeName="ProductsBLLWithSprocs"
    UpdateMethod="UpdateProduct">
    <DeleteParameters>
        <asp:Parameter Name="productID" Type="Int32" />
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="unitsInStock" Type="Int16" />
        <asp:Parameter Name="unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="reorderLevel" Type="Int16" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
        <asp:Parameter Name="productID" Type="Int32" />
    </UpdateParameters>
    <InsertParameters>
        <asp:Parameter Name="productName" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
        <asp:Parameter Name="categoryID" Type="Int32" />
        <asp:Parameter Name="quantityPerUnit" Type="String" />
        <asp:Parameter Name="unitPrice" Type="Decimal" />
        <asp:Parameter Name="unitsInStock" Type="Int16" />
        <asp:Parameter Name="unitsOnOrder" Type="Int16" />
        <asp:Parameter Name="reorderLevel" Type="Int16" />
        <asp:Parameter Name="discontinued" Type="Boolean" />
    </InsertParameters>
</asp:ObjectDataSource>
```

At this point we could tidy up the GridView by customizing the editing interface to include validation, having the `CategoryID` and `SupplierID` columns render as DropDownLists, and so on. We could also add a client-side confirmation to the Delete button, and I encourage you to take the time to implement these enhancements. Since these topics have been covered in previous tutorials, however, we will not cover them again here.

Regardless of whether you enhance the GridView or not, test out the page s core features in a browser. As Figure

24 shows, the page lists the products in a GridView that provides per-row editing and deleting capabilities.

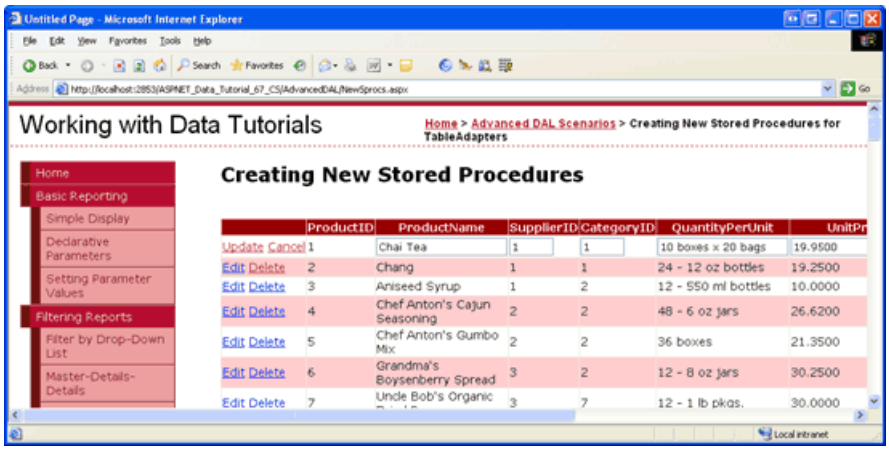


Figure 24: The Products Can Be Viewed, Edited, and Deleted from the GridView (Click to view full-size image)

Summary

The TableAdapters in a Typed DataSet can access data from the database using ad-hoc SQL statements or through stored procedures. When working with stored procedures, either existing stored procedures can be used or the TableAdapter wizard can be instructed to create new stored procedures based on a `SELECT` query. In this tutorial we explored how to have the stored procedures automatically created for us.

While having the stored procedures auto-generated helps save time, there are certain cases where the stored procedure created by the wizard doesn t align with what we would have created on our own. One example is the `Products_Update` stored procedure, which expected both `@Original_ProductID` and `@ProductID` input parameters even though the `@Original_ProductID` parameter was superfluous.

In many scenarios, the stored procedures may already have been created, or we may want to build them manually so as to have a finer degree of control over the stored procedure s commands. In either case, we would want to instruct the TableAdapter to use existing stored procedures for its methods. We shall see how to accomplish this in the next tutorial.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Creating and Maintaining Stored Procedures
- Retrieving Scalar Data from a Stored Procedure
- SQL Server Stored Procedure Basics
- Stored Procedures: An Overview
- Writing a Stored Procedure

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Hilton Geisenow.

Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Using Existing Stored Procedures for the Typed DataSet's TableAdapters (VB)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> In the previous tutorial we learned how to use the TableAdapter Wizard to generate new stored procedures. In this tutorial we learn how the same TableAdapter Wizard can work with existing stored procedures. We also learn how to manually add new stored procedures to our database.

Introduction

In the preceding tutorial we saw how the Typed DataSet s TableAdapters could be configured to use stored procedures to access data rather than ad-hoc SQL statements. In particular, we examined how to have the TableAdapter wizard automatically create these stored procedures. When porting a legacy application to ASP.NET 2.0 or when building an ASP.NET 2.0 website around an existing data model, chances are that the database already contains the stored procedures we need. Alternatively, you may prefer to create your stored procedures by hand or through some tool other than the TableAdapter wizard that auto-generates your stored procedures.

In this tutorial we will look at how to configure the TableAdapter to use existing stored procedures. Since the Northwind database only has a small set of built-in stored procedures, we will also look at the steps needed to manually add new stored procedures to the database through the Visual Studio environment. Let s get started!

> **NOTE**
>
> In the Wrapping Database Modifications within a Transaction tutorial we added methods to the TableAdapter to support transactions (`BeginTransaction` , `CommitTransaction` , and so on). Alternatively, transactions can be managed entirely within a stored procedure, which requires no modifications to the Data Access Layer code. In this tutorial we'll explore the T-SQL commands used to execute a stored procedure s statements within the scope of a transaction.

Step 1: Adding Stored Procedures to the Northwind Database

Visual Studio makes it easy to add new stored procedures to a database. Let s add a new stored procedure to the Northwind database that returns all columns from the `Products` table for those that have a particular `CategoryID` value. From the Server Explorer window, expand the Northwind database so that its folders - Database Diagrams, Tables, Views, and so forth - are displayed. As we saw in the preceding tutorial, the Stored Procedures folder contains the database s existing stored procedures. To add a new stored procedure, simply right-click the Stored Procedures folder and choose the Add New Stored Procedure option from the context menu.

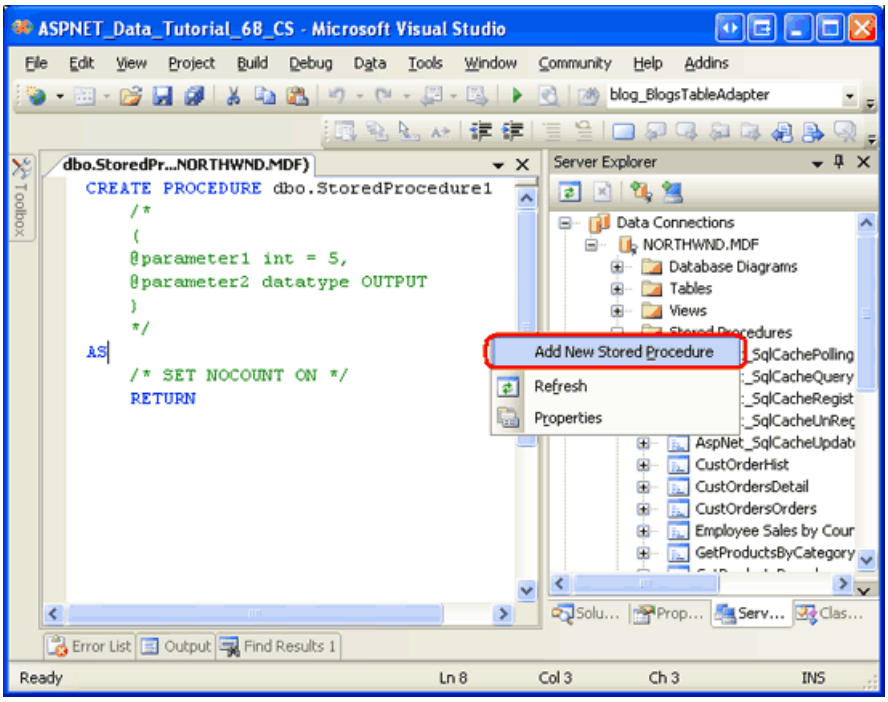


Figure 1: Right-Click the Stored Procedures Folder and Add a New Stored Procedure (Click to view full-size image)

As Figure 1 shows, selecting the Add New Stored Procedure option opens a script window in Visual Studio with the outline of the SQL script needed to create the stored procedure. It is our job to flesh out this script and execute it, at which point the stored procedure will be added to the database.

Enter the following script:

```
CREATE PROCEDURE dbo.Products_SelectByCategoryID
(
    @CategoryID int
)
AS
SELECT ProductID, ProductName, SupplierID, CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued
FROM Products
WHERE CategoryID = @CategoryID
```

This script, when executed, will add a new stored procedure to the Northwind database named `Products_SelectByCategoryID`. This stored procedure accepts a single input parameter (`@CategoryID`, of type `int`) and it returns all of the fields for those products with a matching `CategoryID` value.

To execute this `CREATE PROCEDURE` script and add the stored procedure to the database, click the Save icon in the toolbar or hit Ctrl+S. After doing so, the Stored Procedures folder refreshes, showing the newly created stored procedure. Also, the script in the window will change subtlety from `CREATE PROCEDURE dbo.Products_SelectProductByCategoryID` to `ALTER PROCEDURE dbo.Products_SelectProductByCategoryID`. `CREATE PROCEDURE` adds a new stored procedure to the database, while `ALTER PROCEDURE` updates an existing one. Since the start of the script has changed to `ALTER PROCEDURE`, changing the stored procedures input parameters or SQL statements and clicking the Save icon will update the stored procedure with these changes.

Figure 2 shows Visual Studio after the `Products_SelectByCategoryID` stored procedure has been saved.

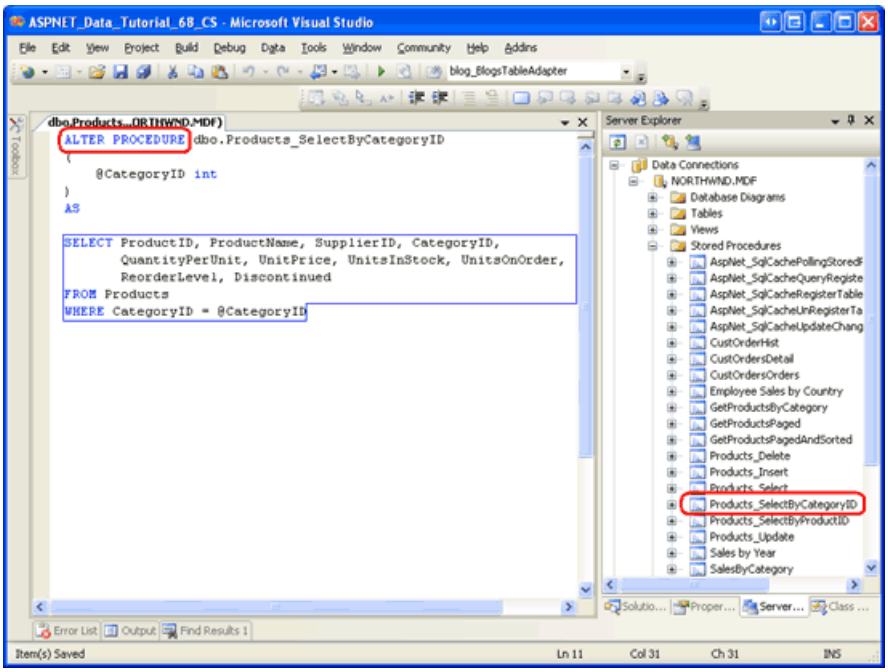


Figure 2: The Stored Procedure `Products_SelectByCategoryID` Has Been Added to the Database (Click to view full-size image)

Step 2: Configuring the TableAdapter to Use an Existing Stored Procedure

Now that the `Products_SelectByCategoryID` stored procedure has been added to the database, we can configure our Data Access Layer to use this stored procedure when one of its methods is invoked. In particular, we will add a `GetProductsByCategoryID(<_i22_>categoryID)<!--_i22_-->` method to the `ProductsTableAdapter` in the `NorthwindWithSprocs` Typed DataSet that calls the `Products_SelectByCategoryID` stored procedure we just created.

Start by opening the `NorthwindWithSprocs` DataSet. Right-click on the `ProductsTableAdapter` and choose Add Query to launch the TableAdapter Query Configuration wizard. In the preceding tutorial we opted to have the TableAdapter create a new stored procedure for us. For this tutorial, however, we want to wire the new TableAdapter method to the existing `Products_SelectByCategoryID` stored procedure. Therefore, choose the Use existing stored procedure option from the wizard s first step and then click Next.

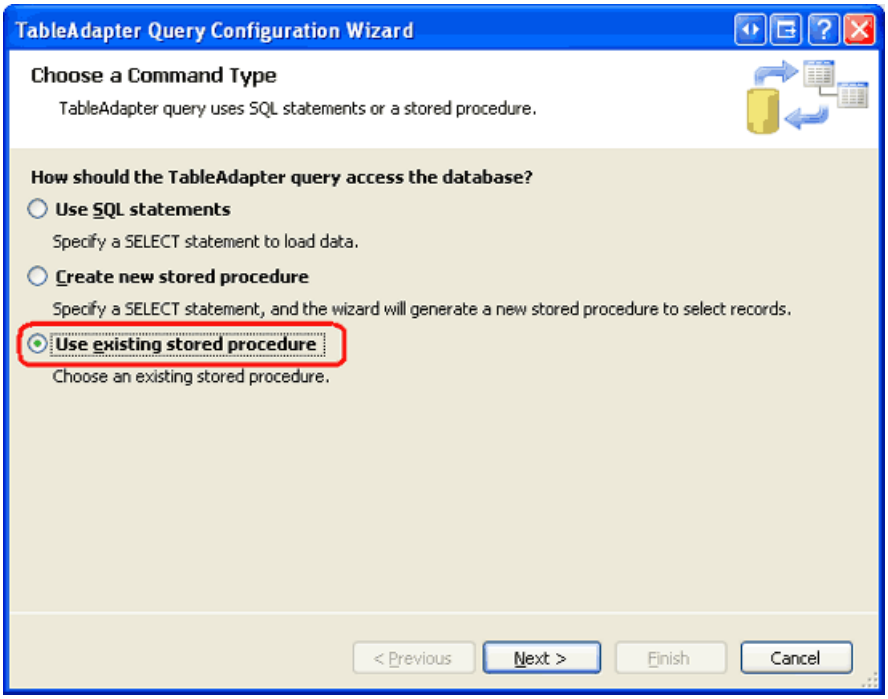


Figure 3: Choose the Use existing stored procedure Option (Click to view full-size image)

The following screen provides a drop-down list populated with the database s stored procedures. Selecting a stored procedure lists its input parameters on the left and the data fields returned (if any) on the right. Choose the `Products_SelectByCategoryID` stored procedure from the list and click Next.

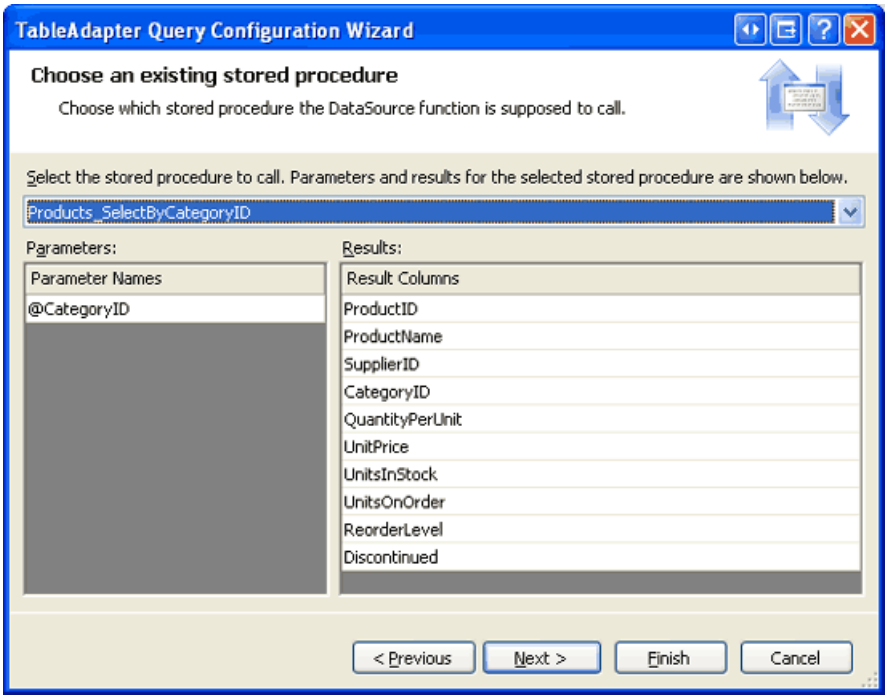


Figure 4: Pick the `Products_SelectByCategoryID` Stored Procedure (Click to view full-size image)

The next screen asks us what kind of data is returned by the stored procedure and our answer here determines the type returned by the TableAdapter s method. For example, if we indicate that tabular data is returned, the method will return a `ProductsDataTable` instance populated with the records returned by the stored procedure. In contrast, if we indicate that this stored procedure returns a single value the TableAdapter will return an `Object` that is assigned the value in the first column of the first record returned by the stored procedure.

Since the `Products_SelectByCategoryID` stored procedure returns all products that belong to a particular category, choose the first answer - Tabular data - and click Next.

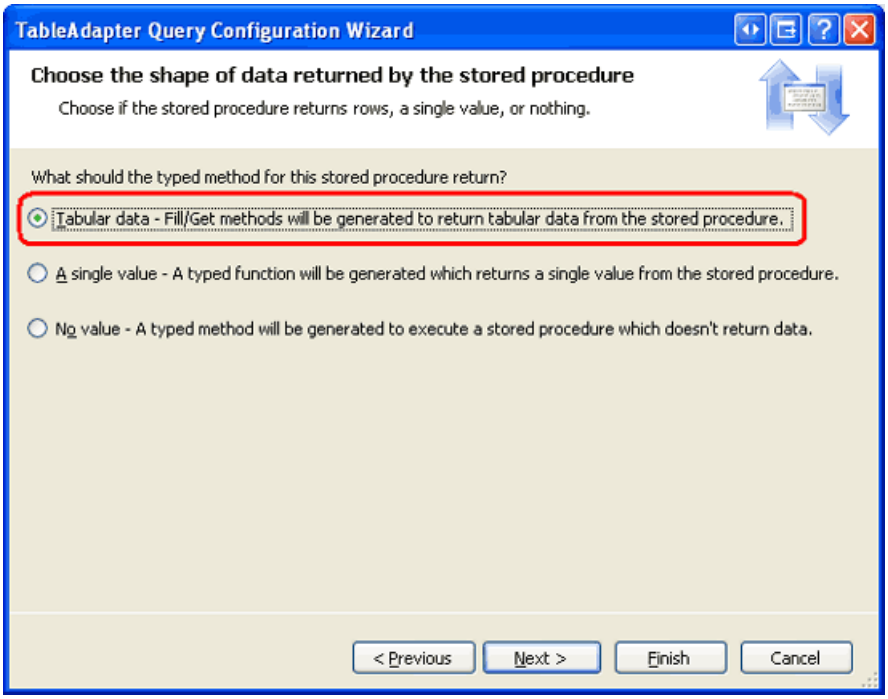


Figure 5: Indicate that the Stored Procedure Returns Tabular Data (Click to view full-size image)

All that remains is to indicate what method patterns to use followed by the names for these methods. Leave both the Fill a DataTable and Return a DataTable options checked, but rename the methods to `FillByCategoryID` and `GetProductsByCategoryID`. Then click Next to review a summary of the tasks the wizard will perform. If everything looks correct, click Finish.

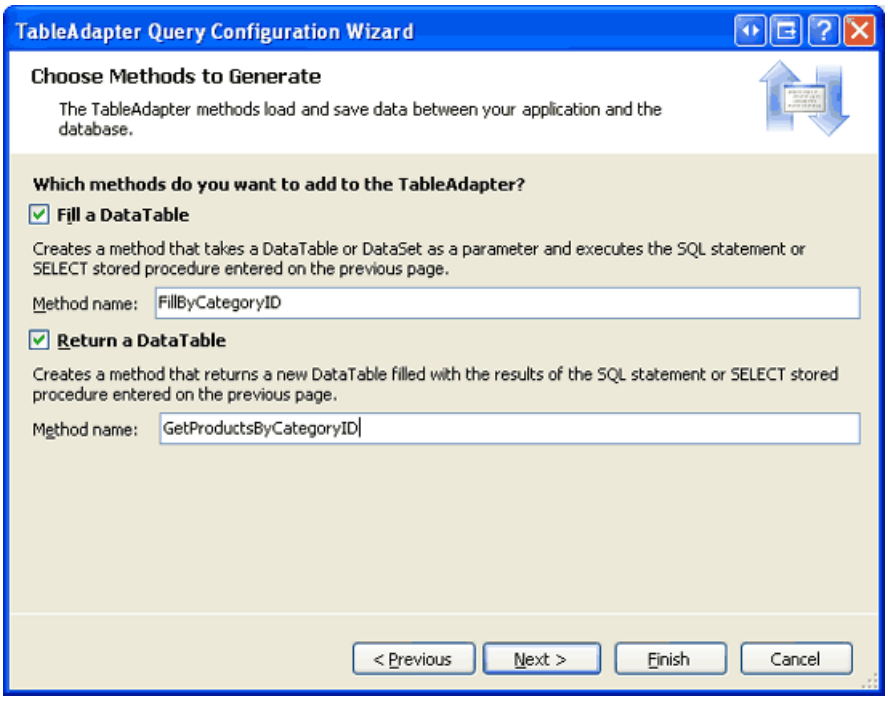


Figure 6: Name the Methods `FillByCategoryID` and `GetProductsByCategoryID` (Click to view full-size image)

> **NOTE**
>
> The TableAdapter methods we just created, `FillByCategoryID` and `GetProductsByCategoryID`, expect an input parameter of type `Integer`. This input parameter value is passed into the stored procedure via its `@CategoryID` parameter. If you modify the `Products_SelectByCategory` stored procedure s parameters, you'll need to also update the parameters for these TableAdapter methods. As discussed in the previous tutorial, this can be done in one of two ways: by manually adding or removing parameters from the parameters collection or by rerunning the TableAdapter wizard.

Step 3: Adding a `GetProductsByCategoryID(categoryID)` Method to the BLL

With the `GetProductsByCategoryID` DAL method complete, the next step is to provide access to this method in the Business Logic Layer. Open the `ProductsBLLWithSprocs` class file and add the following method:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetProductsByCategoryID(ByVal categoryID As Integer) _
    As NorthwindWithSprocs.ProductsDataTable
    Return Adapter.GetProductsByCategoryID(categoryID)
End Function
```

This BLL method simply returns the `ProductsDataTable` returned from the `ProductsTableAdapter` s `GetProductsByCategoryID` method. The `DataObjectMethodAttribute` attribute provides metadata used by the ObjectDataSource s Configure Data Source wizard. In particular, this method will appear in the SELECT tab s drop-down list.

Step 4: Displaying Products by Category

To test the newly added `Products_SelectByCategoryID` stored procedure and the corresponding DAL and BLL

methods, let s create an ASP.NET page that contains a DropDownList and a GridView. The DropDownList will list all of the categories in the database while the GridView will display the products belonging to the selected category.

> **NOTE**
>
> We ve created master/detail interfaces using DropDownLists in previous tutorials. For a more in-depth look at implementing such a master/detail report, refer to the Master/Detail Filtering With a DropDownList tutorial.

Open the `ExistingSprocs.aspx` page in the `AdvancedDAL` folder and drag a DropDownList from the Toolbox onto the Designer. Set the DropDownList s `ID` property to `Categories` and its `AutoPostBack` property to `True`. Next, from its smart tag, bind the DropDownList to a new ObjectDataSource named `CategoriesDataSource`. Configure the ObjectDataSource so that it retrieves its data from the `CategoriesBLL` class s `GetCategories` method. Set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) .

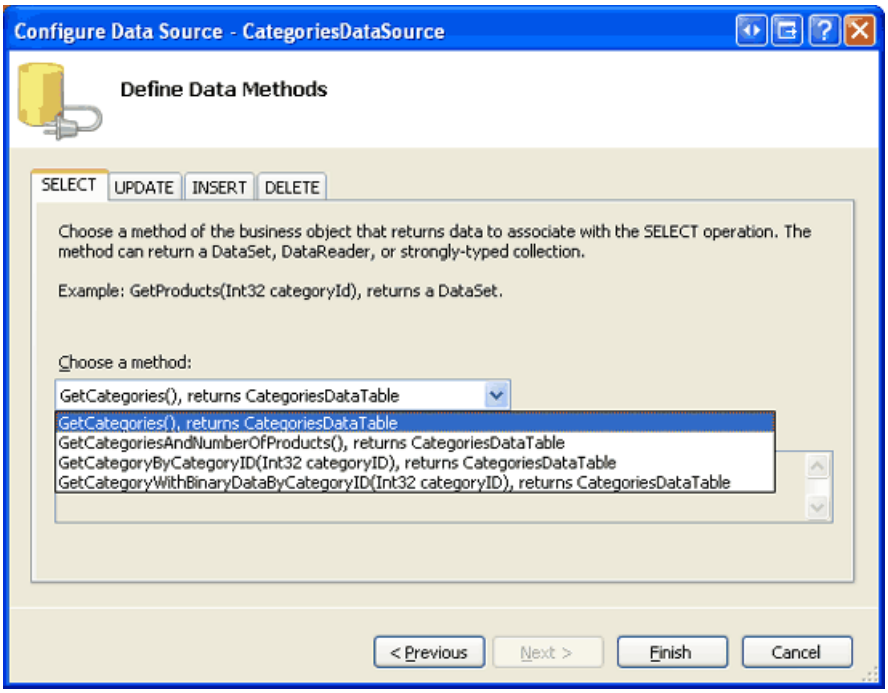


Figure 7: Retrieve Data from the `CategoriesBLL` Class s `GetCategories` Method (Click to view full-size image)

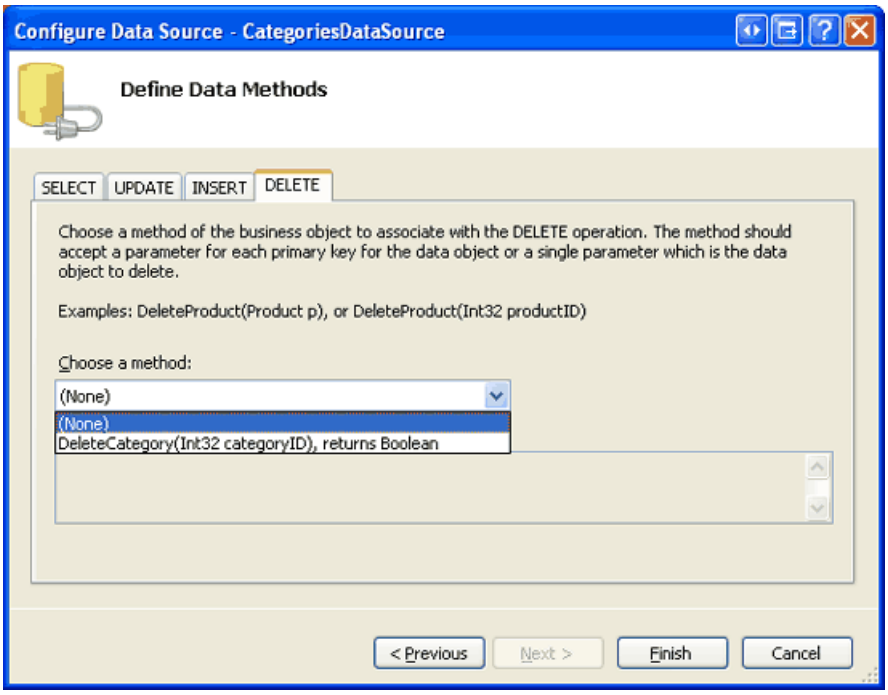


Figure 8: Set the Drop-Down Lists in the UPDATE, INSERT, and DELETE Tabs to (None) (Click to view full-size image)

After completing the ObjectDataSource wizard, configure the DropDownList to display the `CategoryName` data field and to use the `CategoryID` field as the `Value` for each `ListItem`.

At this point, the DropDownList and ObjectDataSource s declarative markup should similar to the following:

```
<asp:DropDownList ID="Categories" runat="server" AutoPostBack="True"
    DataSourceID="CategoriesDataSource" DataTextField="CategoryName"
    DataValueField="CategoryID">
</asp:DropDownList>
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCategories" TypeName="CategoriesBLL">
</asp:ObjectDataSource>
```

Next, drag a GridView onto the Designer, placing it beneath the DropDownList. Set the GridView s `ID` to `ProductsByCategory` and, from its smart tag, bind it to a new ObjectDataSource named `ProductsByCategoryDataSource`. Configure the `ProductsByCategoryDataSource` ObjectDataSource to use the `ProductsBLLWithSprocs` class, having it retrieve its data using the `GetProductsByCategoryID(categoryID)` method. Since this GridView will only be used to display data, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) and click Next.

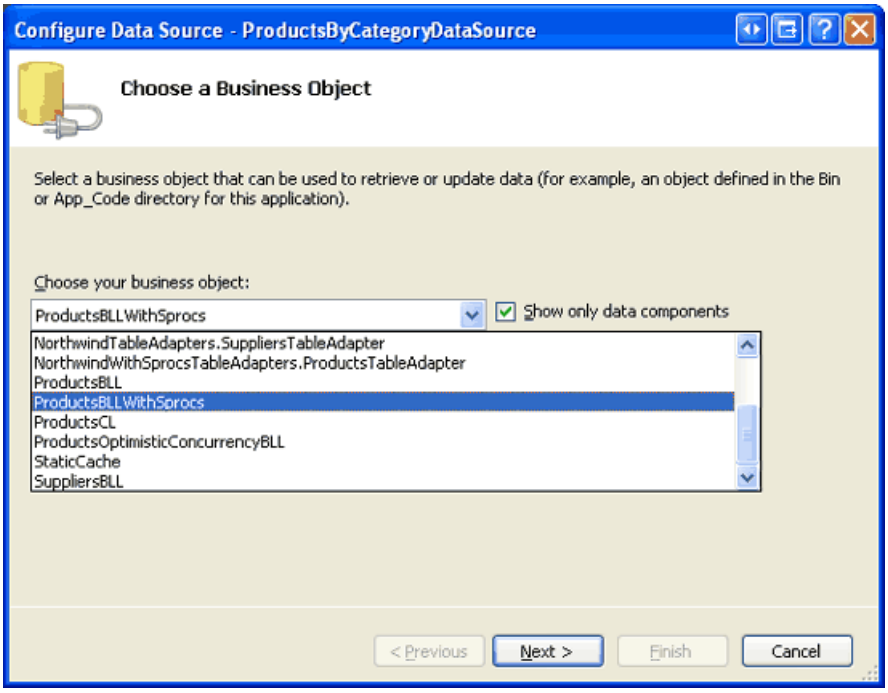


Figure 9: Configure the ObjectDataSource to Use the `ProductsBLLWithSprocs` Class (Click to view full-size image)

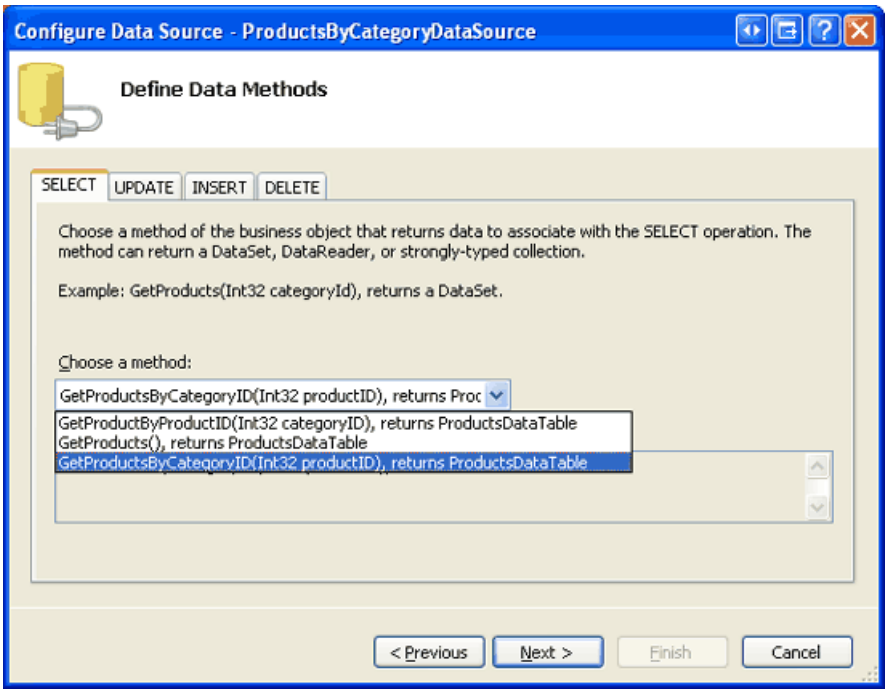


Figure 10: Retrieve Data from the `GetProductsByCategoryID(categoryID)` Method (Click to view full-size image)

The method chosen in the SELECT tab expects a parameter, so the final step of the wizard prompts us for the parameter s source. Set the Parameter source drop-down list to Control and choose the `Categories` control from the ControlID drop-down list. Click Finish to complete the wizard.

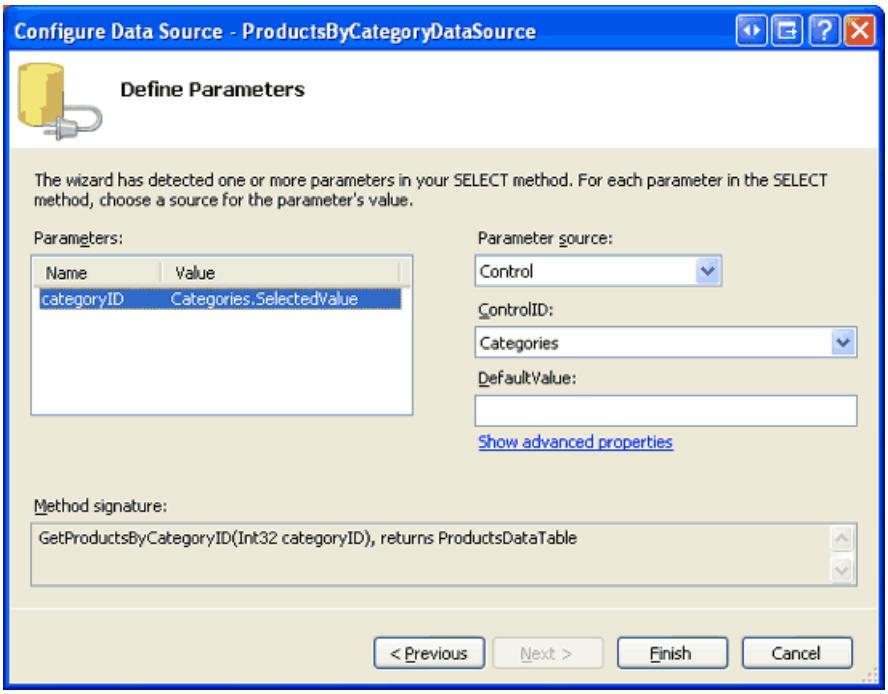


Figure 11: Use the `Categories` DropDownList as the Source of the `categoryID` Parameter (Click to view full-size image)

Upon completing the ObjectDataSource wizard, Visual Studio will add BoundFields and a CheckBoxField for each of the product data fields. Feel free to customize these fields as you see fit.

Visit the page through a browser. When visiting the page the Beverages category is selected and the corresponding products listed in the grid. Changing the drop-down list to an alternative category, as Figure 12 shows, causes a postback and reloads the grid with the products of the newly selected category.

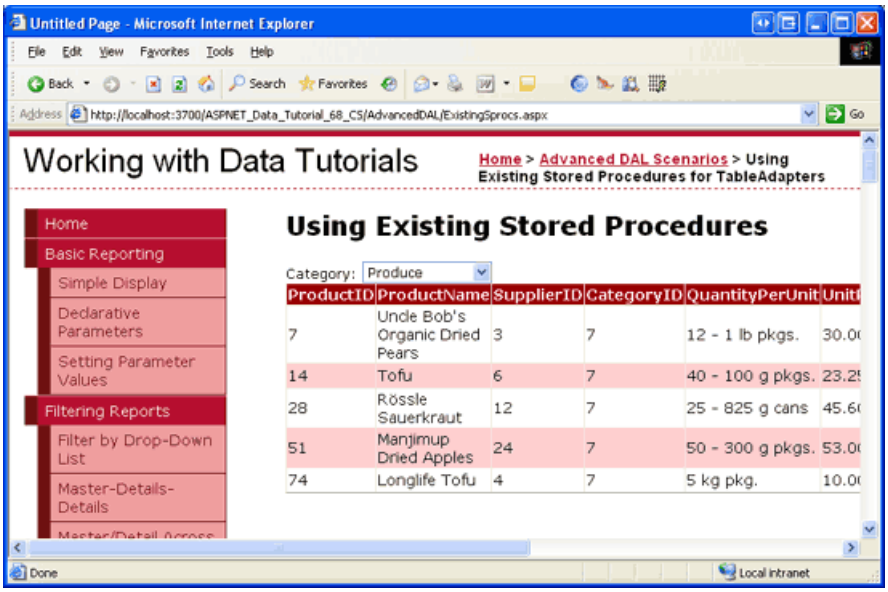


Figure 12: The Products in the Produce Category are Displayed (Click to view full-size image)

Step 5: Wrapping a Stored Procedure s Statements Within the Scope of a Transaction

In the Wrapping Database Modifications within a Transaction tutorial we discussed techniques for performing a series of database modification statements within the scope of a transaction. Recall that the modifications performed under the umbrella of a transaction either all succeed or all fail, guaranteeing atomicity. Techniques for using transactions include:

- Using the classes in the `System.Transactions` namespace,
- Having the Data Access Layer use ADO.NET classes like `SqlTransaction`, and
- Adding the T-SQL transaction commands directly within the stored procedure

The *Wrapping Database Modifications within a Transaction* tutorial used the ADO.NET classes in the DAL. The remainder of this tutorial examines how to manage a transaction using T-SQL commands from within a stored procedure.

The three key SQL commands for manually starting, committing, and rolling back a transaction are `BEGIN TRANSACTION`, `COMMIT TRANSACTION`, and `ROLLBACK TRANSACTION`, respectively. Like with the ADO.NET approach, when using transactions from within a stored procedure we need to apply the following pattern:

1. Indicate the start of a transaction.
2. Execute the SQL statements that comprise the transaction.
3. If there is an error in any one of the statements from Step 2, rollback the transaction.
4. If all of the statements from Step 2 complete without error, commit the transaction.

This pattern can be implemented in T-SQL syntax using the following template:

```
BEGIN TRY
  BEGIN TRANSACTION -- Start the transaction
  ... Perform the SQL statements that makeup the transaction ...
  -- If we reach here, success!
  COMMIT TRANSACTION
END TRY
BEGIN CATCH
  -- Whoops, there was an error
  ROLLBACK TRANSACTION
  -- Raise an error with the
  -- details of the exception
  DECLARE @ErrMsg nvarchar(4000),
          @ErrSeverity int
  SELECT @ErrMsg = ERROR_MESSAGE(),
         @ErrSeverity = ERROR_SEVERITY()

  RAISERROR(@ErrMsg, @ErrSeverity, 1)
END CATCH
```

The template starts by defining a `TRY...CATCH` block, a construct new to SQL Server 2005. Like with `Try...Catch` blocks in Visual Basic, the SQL `TRY...CATCH` block executes the statements in the `TRY` block. If any statement raises an error, control is immediately transferred to the `CATCH` block.

If there are no errors executing the SQL statements that makeup the transaction, the `COMMIT TRANSACTION` statement commits the changes and completes the transaction. If, however, one of the statements results in an error, the `ROLLBACK TRANSACTION` in the `CATCH` block returns the database to its state prior to the start of the transaction. The stored procedure also raises an error using the RAISERROR command, which causes a `SqlException` to be raised in the application.

> **NOTE**
> Since the `TRY...CATCH` block is new to SQL Server 2005, the above template will not work if you are using older versions of Microsoft SQL Server. If you are not using SQL Server 2005, consult Managing Transactions in SQL Server Stored Procedures for a template that will work with other versions of SQL Server.

Let s look at a concrete example. A foreign key constraint exists between the `Categories` and `Products` tables, meaning that each `CategoryID` field in the `Products` table must map to a `CategoryID` value in the `Categories` table. Any action that would violate this constraint, such as attempting to delete a category that has associated products, results in a foreign key constraint violation. To verify this, revisit the Updating and Deleting Existing Binary Data example in the Working with Binary Data section (`~/BinaryData/UpdatingAndDeleting.aspx`). This page lists each category in the system along with Edit and Delete buttons (see Figure 13), but if you attempt to delete a category that has associated products - such as Beverages - the delete fails due to a foreign key constraint violation (see Figure 14).

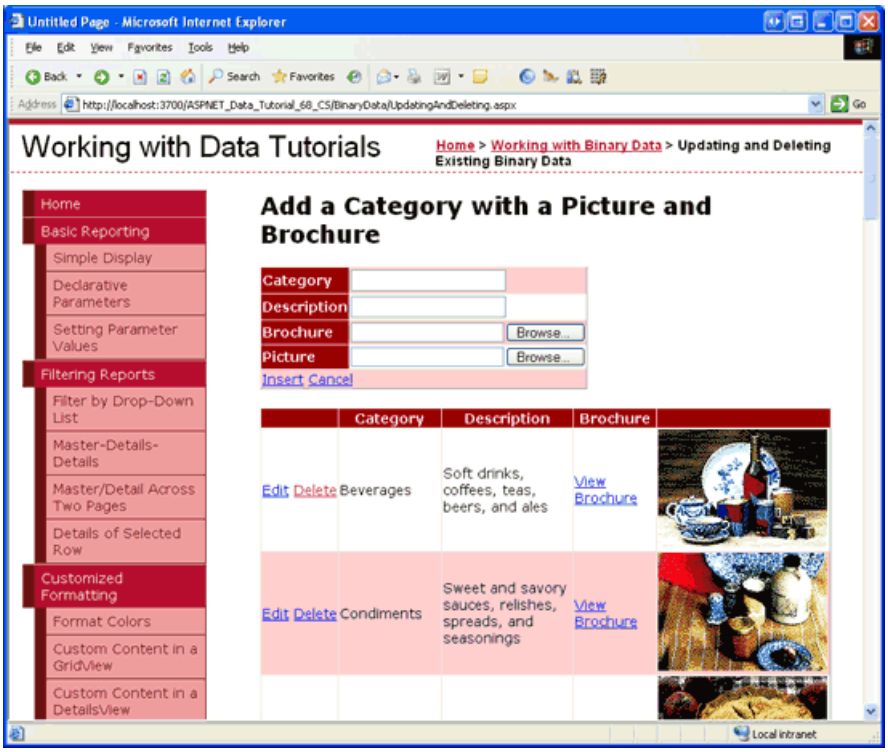


Figure 13: Each Category is Displayed in a GridView with Edit and Delete Buttons (Click to view full-size image)

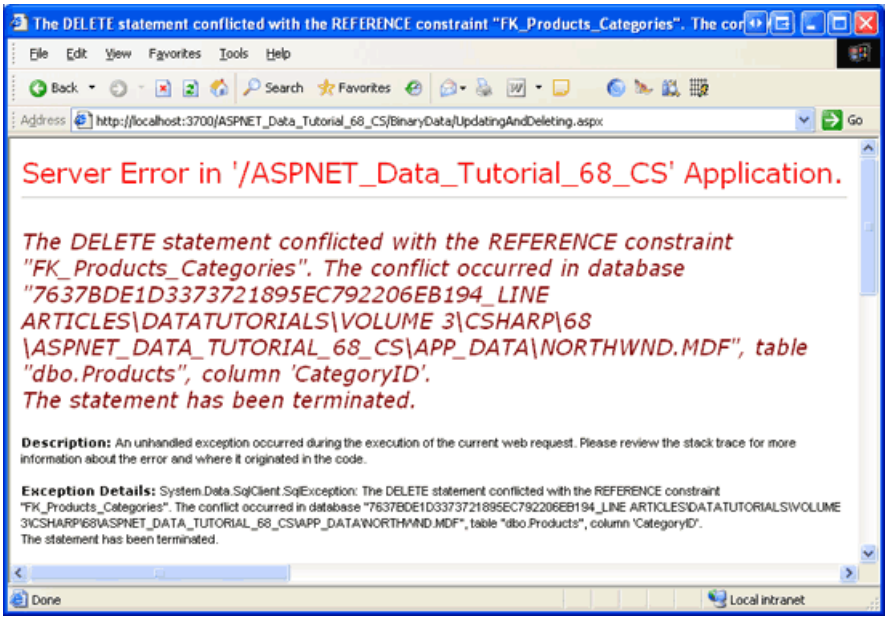


Figure 14: You Cannot Delete a Category that has Existing Products (Click to view full-size image)

Imagine, though, that we want to allow categories to be deleted regardless of whether they have associated products. Should a category with products be deleted, imagine that we want to also delete its existing products (although another option would be to simply set its products `CategoryID` values to `NULL`). This functionality could be implemented through the cascade rules of the foreign key constraint. Alternatively, we could create a stored procedure that accepts a `@CategoryID` input parameter and, when invoked, explicitly deletes all of the associated products and then the specified category.

Our first attempt at such a stored procedure might look like the following:

```
CREATE PROCEDURE dbo.Categories_Delete
(
    @CategoryID int
)
AS
-- First, delete the associated products...
DELETE FROM Products
WHERE CategoryID = @CategoryID
-- Now delete the category
DELETE FROM Categories
WHERE CategoryID = @CategoryID
```

While this will definitely delete the associated products and category, it does not do so under the umbrella of a transaction. Imagine that there is some other foreign key constraint on `Categories` that would prohibit the deletion of a particular `@CategoryID` value. The problem is that in such a case all of the products will be deleted before we attempt to delete the category. The net result is that for such a category, this stored procedure would remove all of its products while the category remained since it still has related records in some other table.

If the stored procedure were wrapped within the scope of a transaction, however, the deletes to the `Products` table would be rolled back in the face of a failed delete on `Categories`. The following stored procedure script uses a transaction to assure atomicity between the two `DELETE` statements:

```
CREATE PROCEDURE dbo.Categories_Delete
(
    @CategoryID int
)
AS
BEGIN TRY
  BEGIN TRANSACTION -- Start the transaction
  -- First, delete the associated products...
  DELETE FROM Products
  WHERE CategoryID = @CategoryID
  -- Now delete the category
  DELETE FROM Categories
  WHERE CategoryID = @CategoryID
  -- If we reach here, success!
  COMMIT TRANSACTION
END TRY
BEGIN CATCH
  -- Whoops, there was an error
  ROLLBACK TRANSACTION
  -- Raise an error with the
  -- details of the exception
  DECLARE @ErrMsg nvarchar(4000),
          @ErrSeverity int
  SELECT @ErrMsg = ERROR_MESSAGE(),
         @ErrSeverity = ERROR_SEVERITY()

  RAISERROR(@ErrMsg, @ErrSeverity, 1)
END CATCH
```

Take a moment to add the `Categories_Delete` stored procedure to the Northwind database. Refer back to Step 1 for instructions on adding stored procedures to a database.

Step 6: Updating the `CategoriesTableAdapter`

While we ve added the `Categories_Delete` stored procedure to the database, the DAL is currently configured to use ad-hoc SQL statements to perform the delete. We need to update the `CategoriesTableAdapter` and instruct it to use the `Categories_Delete` stored procedure instead.

> **NOTE**
>
> Earlier in this tutorial we were working with the `NorthwindWithSprocs` DataSet. But that DataSet only has a single entity, `ProductsDataTable`, and we need to work with categories. Therefore, for the remainder of this tutorial when I talk about the Data Access Layer I m referring to the `Northwind` DataSet, the one that we first created in the Creating a Data Access Layer tutorial.

Open the Northwind DataSet, select the `CategoriesTableAdapter`, and go to the Properties window. The Properties window lists the `InsertCommand`, `UpdateCommand`, `DeleteCommand`, and `SelectCommand` used by the TableAdapter, as well as its name and connection information. Expand the `DeleteCommand` property to see its details. As Figure 15 shows, the `DeleteCommand` s `CommandType` property is set to Text, which instructs it to send the text in the `CommandText` property as an ad-hoc SQL query.

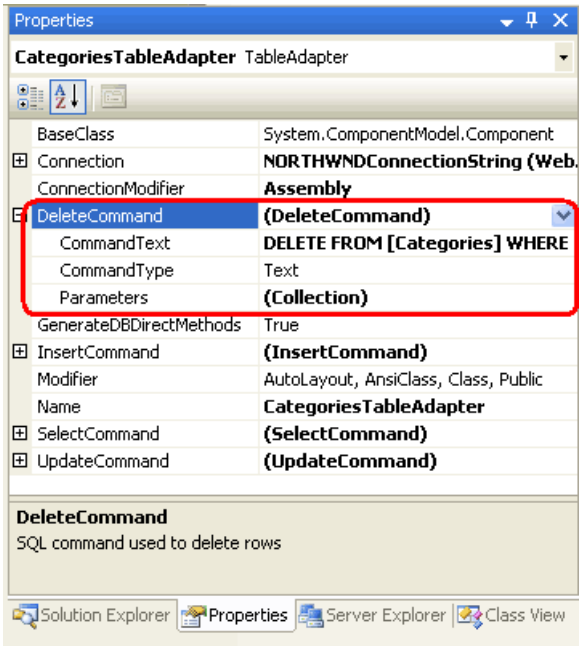


Figure 15: Select the `CategoriesTableAdapter` in the Designer to View Its Properties in the Properties Window

To change these settings, select the (DeleteCommand) text in the Properties window and choose (New) from the drop-down list. This will clear out the settings for the `CommandText`, `CommandType`, and `Parameters` properties. Next, set the `CommandType` property to `StoredProcedure` and then type in the name of the stored procedure for the `CommandText` (`dbo.Categories_Delete`). If you make sure to enter the properties in this order - first the `CommandType` and then the `CommandText` - Visual Studio will automatically populate the Parameters collection. If you do not enter these properties in this order, you will have to manually add the parameters through the Parameters Collection Editor. In either case, it s prudent to click on the ellipses in the Parameters property to bring up the Parameters Collection Editor to verify that the correct parameter settings changes have been made (see Figure 16). If you do not see any parameters in the dialog box, add the `@CategoryID` parameter manually (you do not need to add the `@RETURN_VALUE` parameter).

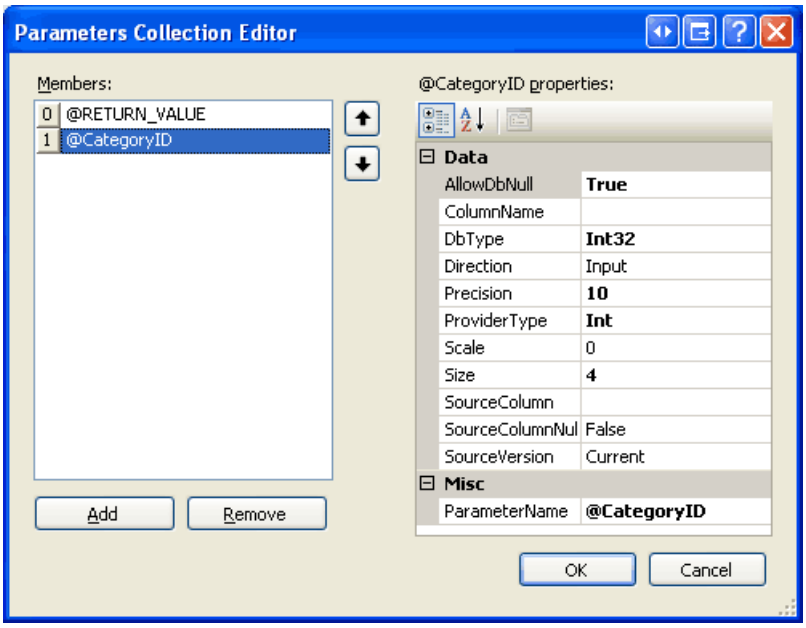


Figure 16: Ensure That the Parameters Settings are Correct

Once the DAL has been updated, deleting a category will automatically delete all of its associated products and do so under the umbrella of a transaction. To verify this, return to the Updating and Deleting Existing Binary Data page and click the Delete button for one of the categories. With one single click of the mouse, the category and all of its associated products will be deleted.

> **NOTE**
>
> Before testing the `Categories_Delete` stored procedure, which will delete a number of products along with the selected category, it might be prudent to make a backup copy of your database. If you are using the `NORTHWND.MDF` database in `App_Data`, simply close Visual Studio and copy the MDF and LDF files in `App_Data` to some other folder. After testing the functionality, you can restore the database by closing Visual Studio and replacing the current MDF and LDF files in `App_Data` with the backup copies.

Summary

While the TableAdapter s wizard will automatically generate stored procedures for us, there are times when we might already have such stored procedures created or want to create them manually or with other tools instead. To accommodate such scenarios, the TableAdapter can also be configured to point to an existing stored procedure. In this tutorial we looked at how to manually add stored procedures to a database through the Visual Studio environment and how to wire the TableAdapter s methods to these stored procedures. We also examined the T-SQL commands and script pattern used for starting, committing, and rolling back transactions from within a stored procedure.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Hilton

Geisenow, S ren Jacob Lauritsen, and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Updating the TableAdapter to Use JOINs (VB)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

When working with a database it is common to request data that is spread across multiple tables. To retrieve data from two different tables we can use either a correlated subquery or a JOIN operation. In this tutorial we compare correlated subqueries and the JOIN syntax before looking at how to create a TableAdapter that includes a JOIN in its main query.

Introduction

With relational databases the data we are interested in working with is often spread across multiple tables. For example, when displaying product information we likely want to list each product s corresponding category and supplier s names. The `Products` table has `CategoryID` and `SupplierID` values, but the actual category and supplier names are in the `Categories` and `Suppliers` tables, respectively.

To retrieve information from another, related table, we can either use *correlated subqueries* or `JOIN` *s*. A correlated subquery is a nested `SELECT` query that references columns in the outer query. For example, in the Creating a Data Access Layer tutorial we used two correlated subqueries in the `ProductsTableAdapter` s main query to return the category and supplier names for each product. A `JOIN` is a SQL construct that merges related rows from two different tables. We used a `JOIN` in the Querying Data with the SqlDataSource Control tutorial to display category information alongside each product.

The reason we have abstained from using `JOIN` s with the TableAdapters is because of limitations in the TableAdapter s wizard to auto-generate corresponding `INSERT`, `UPDATE`, and `DELETE` statements. More specifically, if the TableAdapter s main query contains any `JOIN` s, the TableAdapter cannot auto-create the ad-hoc SQL statements or stored procedures for its `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties.

In this tutorial we will briefly compare and contrast correlated subqueries and `JOIN` s before exploring how to create a TableAdapter that includes `JOIN` s in its main query.

Comparing and Contrasting Correlated Subqueries and `JOIN` s

Recall that the `ProductsTableAdapter` created in the first tutorial in the `Northwind` DataSet uses correlated subqueries to bring back each product s corresponding category and supplier name. The `ProductsTableAdapter` s main query is shown below.

```
SELECT ProductID, ProductName, SupplierID, CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued,
       (SELECT CategoryName FROM Categories WHERE Categories.CategoryID =
            Products.CategoryID) as CategoryName,
       (SELECT CompanyName FROM Suppliers WHERE Suppliers.SupplierID =
            Products.SupplierID) as SupplierName
FROM Products
```

The two correlated subqueries -

`(SELECT CategoryName FROM Categories WHERE Categories.CategoryID = Products.CategoryID)` and

`(SELECT CompanyName FROM Suppliers WHERE Suppliers.SupplierID = Products.SupplierID)` - are `SELECT` queries that return a single value per product as an additional column in the outer `SELECT` statement s column list.

Alternatively, a `JOIN` can be used to return each product s supplier and category name. The following query returns the same output as the above one, but uses `JOIN` s in place of subqueries:

```
SELECT ProductID, ProductName, Products.SupplierID, Products.CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued,
       Categories.CategoryName,
       Suppliers.CompanyName as SupplierName
FROM Products
    LEFT JOIN Categories ON
        Categories.CategoryID = Products.CategoryID
    LEFT JOIN Suppliers ON
        Suppliers.SupplierID = Products.SupplierID
```

A `JOIN` merges the records from one table with records from another table based on some criteria. In the above query, for example, the `LEFT JOIN Categories ON Categories.CategoryID = Products.CategoryID` instructs SQL Server to merge each product record with the category record whose `CategoryID` value matches the product s `CategoryID` value. The merged result allows us to work with the corresponding category fields for each product (such as `CategoryName`).

NOTE

`JOIN` s are commonly used when querying data from relational databases. If you are new to the `JOIN` syntax or need to brush up a bit on its usage, I d recommend the SQL Join tutorial at W3 Schools. Also worth reading are the `JOIN` Fundamentals and Subquery Fundamentals sections of the SQL Books Online.

Since `JOIN` s and correlated subqueries can both be used to retrieve related data from other tables, many developers are left scratching their heads and wondering which approach to use. All of the SQL gurus I ve talked to have said roughly the same thing, that it doesn t really matter performance-wise as SQL Server will produce roughly identical execution plans. Their advice, then, is to use the technique that you and your team are most comfortable with. It merits noting that after imparting this advice these experts immediately express their preference of `JOIN` s over correlated subqueries.

When building a Data Access Layer using Typed DataSets, the tools work better when using subqueries. In particular, the TableAdapter s wizard will not auto-generate corresponding `INSERT`, `UPDATE`, and `DELETE` statements if the main query contains any `JOIN` s, but will auto-generate these statements when correlated subqueries are used.

To explore this shortcoming, create a temporary Typed DataSet in the `~/App_Code/DAL` folder. During the TableAdapter Configuration wizard, choose to use ad-hoc SQL statements and enter the following `SELECT` query (see Figure 1):

```
SELECT ProductID, ProductName, Products.SupplierID, Products.CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued,
       Categories.CategoryName,
       Suppliers.CompanyName as SupplierName
FROM Products
    LEFT JOIN Categories ON
        Categories.CategoryID = Products.CategoryID
    LEFT JOIN Suppliers ON
        Suppliers.SupplierID = Products.SupplierID
```

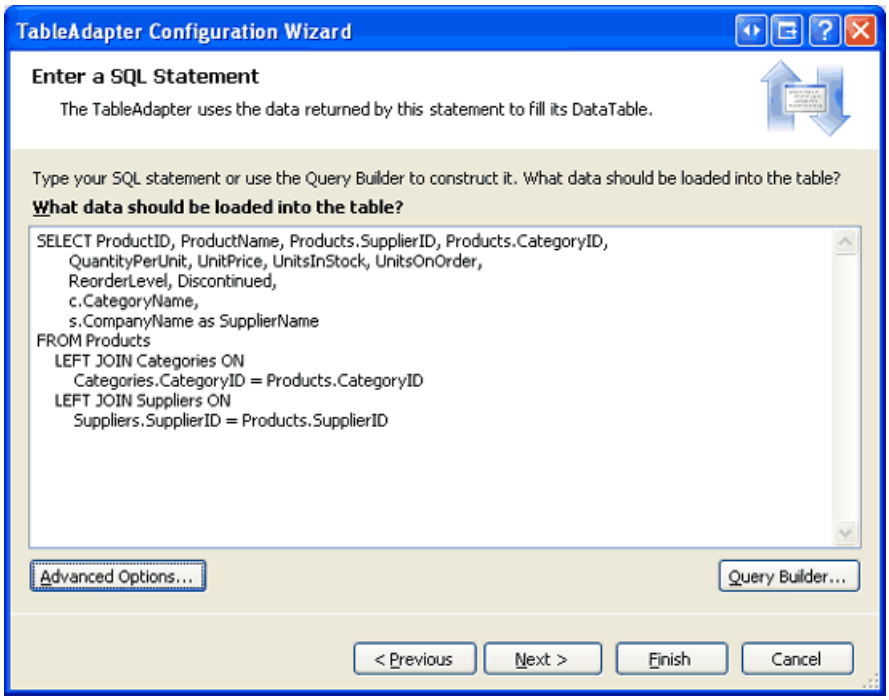


Figure 1: Enter a Main Query that Contains `JOIN` s (Click to view full-size image)

By default, the TableAdapter will automatically create `INSERT`, `UPDATE`, and `DELETE` statements based on the main query. If you click the Advanced button you can see that this feature is enabled. Despite this setting, the TableAdapter will not be able to create the `INSERT`, `UPDATE`, and `DELETE` statements because the main query contains a `JOIN`.

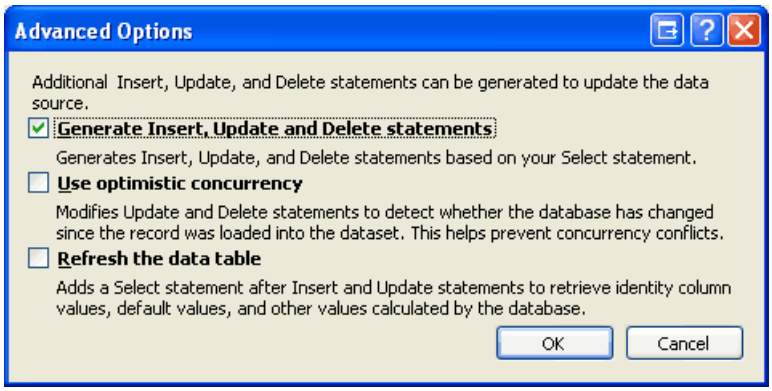


Figure 2: Enter a Main Query that Contains `JOIN` s

Click Finish to complete the wizard. At this point your DataSet s Designer will include a single TableAdapter with a DataTable with columns for each of the fields returned in the `SELECT` query s column list. This includes the `CategoryName` and `SupplierName`, as Figure 3 shows.

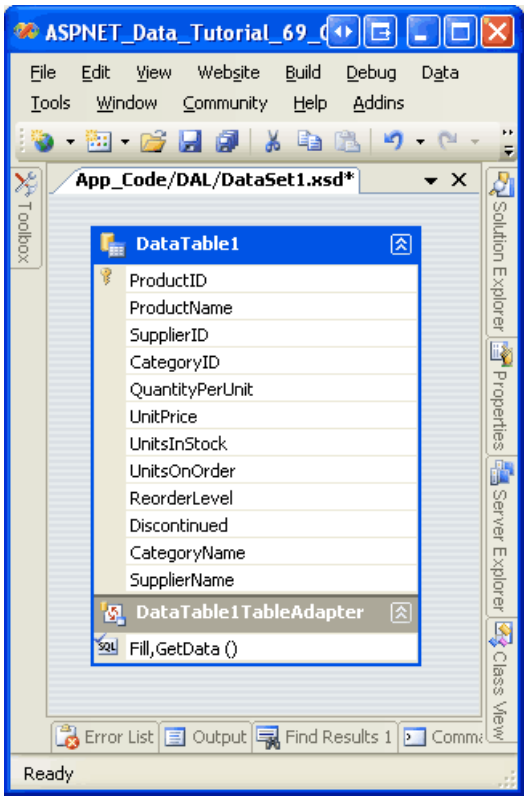


Figure 3: The DataTable Includes a Column for Each Field Returned in the Column List

While the DataTable has the appropriate columns, the TableAdapter lacks values for its `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties. To confirm this, click on the TableAdapter in the Designer and then go to the Properties window. There you will see that the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties are set to (None) .

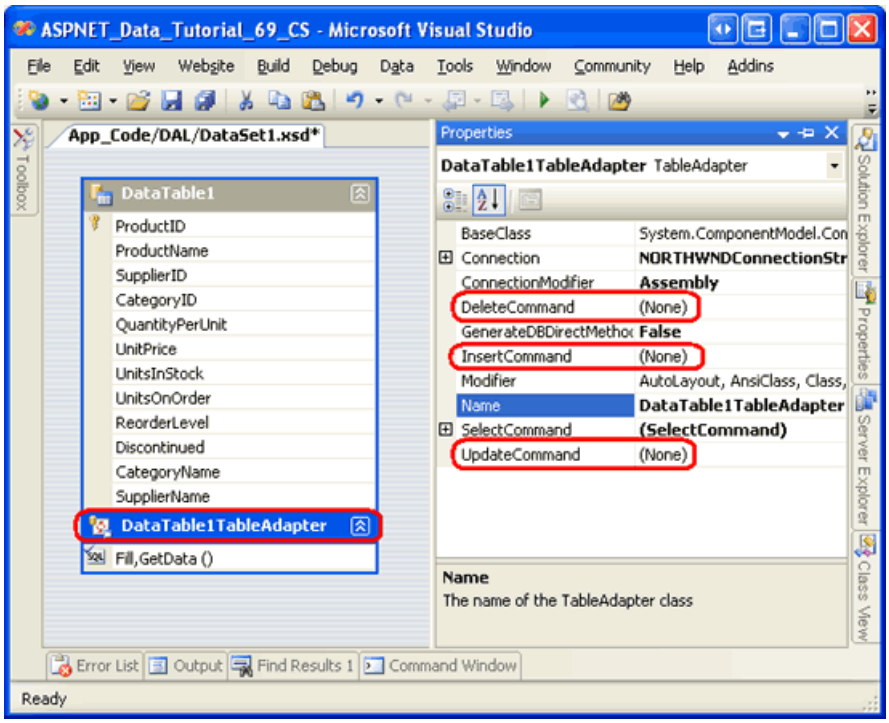


Figure 4: The `InsertCommand`, `UpdateCommand`, and `DeleteCommand` Properties are Set to (None) (Click to view full-size image)

To work around this shortcoming, we can manually provide the SQL statements and parameters for the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties via the Properties window. Alternatively, we could start by configuring the TableAdapter s main query to *not* include any `JOIN` s. This will allow the `INSERT`, `UPDATE`, and `DELETE` statements to be auto-generated for us. After completing the wizard, we could then manually

update the TableAdapter s `SelectCommand` from the Properties window so that it includes the `JOIN` syntax.

While this approach works, it is very brittle when using ad-hoc SQL queries because any time the TableAdapter s main query is re-configured through the wizard, the auto-generated `INSERT`, `UPDATE`, and `DELETE` statements are recreated. That means all of the customizations we later made would be lost if we right-clicked on the TableAdapter, chose Configure from the context menu, and completed the wizard again.

The brittleness of the TableAdapter s auto-generated `INSERT`, `UPDATE`, and `DELETE` statements is, fortunately, limited to ad-hoc SQL statements. If your TableAdapter uses stored procedures, you can customize the `SelectCommand`, `InsertCommand`, `UpdateCommand`, or `DeleteCommand` stored procedures and re-run the TableAdapter Configuration wizard without having to fear that the stored procedures will be modified.

Over the next several steps we will create a TableAdapter that, initially, uses a main query that omits any `JOIN` s so that the corresponding insert, update, and delete stored procedures will be auto-generated. We will then update the `SelectCommand` so that uses a `JOIN` that returns additional columns from related tables. Finally, we'll create a corresponding Business Logic Layer class and demonstrate using the TableAdapter in an ASP.NET web page.

Step 1: Creating the TableAdapter Using a Simplified Main Query

For this tutorial we will add a TableAdapter and strongly-typed DataTable for the `Employees` table in the `NorthwindWithSprocs` DataSet. The `Employees` table contains a `ReportsTo` field that specified the `EmployeeID` of the employee s manager. For example, employee Anne Dodsworth has a `ReportTo` value of 5, which is the `EmployeeID` of Steven Buchanan. Consequently, Anne reports to Steven, her manager. Along with reporting each employee s `ReportsTo` value, we might also want to retrieve the name of their manager. This can be accomplished using a `JOIN`. But using a `JOIN` when initially creating the TableAdapter precludes the wizard from automatically generating the corresponding insert, update, and delete capabilities. Therefore, we will start by creating a TableAdapter whose main query does not contain any `JOIN` s. Then, in Step 2, we will update the main query stored procedure to retrieve the manager s name via a `JOIN`.

Start by opening the `NorthwindWithSprocs` DataSet in the `~/App_Code/DAL` folder. Right-click on the Designer, select the Add option from the context menu, and pick the TableAdapter menu item. This will launch the TableAdapter Configuration wizard. As Figure 5 depicts, have the wizard create new stored procedures and click Next. For a refresher on creating new stored procedures from the TableAdapter s wizard, consult the Creating New Stored Procedures for the Typed DataSet s TableAdapters tutorial.

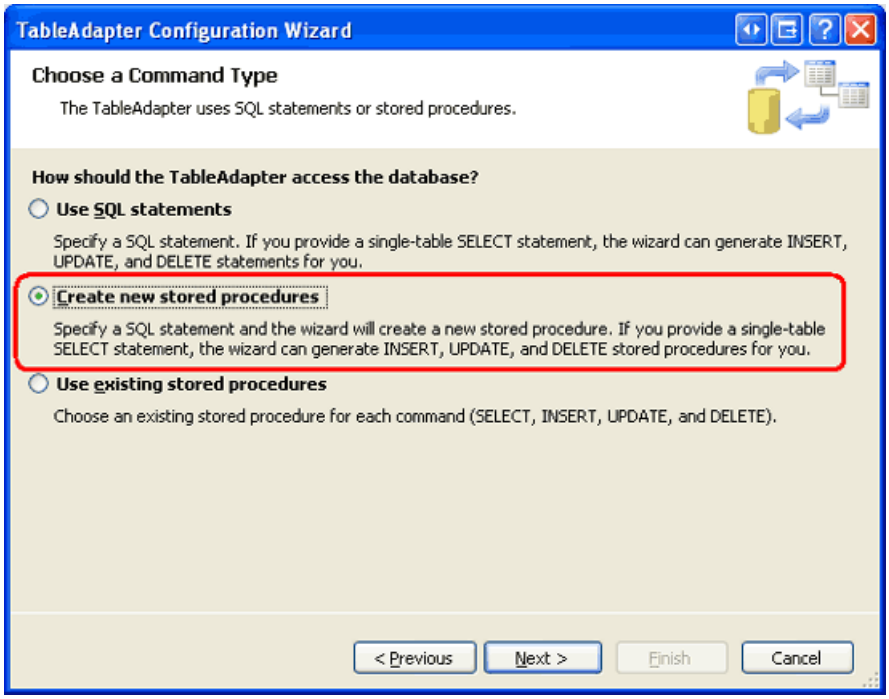


Figure 5: Select the Create new stored procedures Option (Click to view full-size image)

Use the following `SELECT` statement for the TableAdapter s main query:

```
SELECT EmployeeID, LastName, FirstName, Title, HireDate, ReportsTo, Country
FROM Employees
```

Since this query does not include any `JOIN` s, the TableAdapter wizard will automatically create stored procedures with corresponding `INSERT`, `UPDATE`, and `DELETE` statements, as well as a stored procedure for executing the main query.

The following step allows us to name the TableAdapter s stored procedures. Use the names `Employees_Select`, `Employees_Insert`, `Employees_Update`, and `Employees_Delete`, as shown in Figure 6.

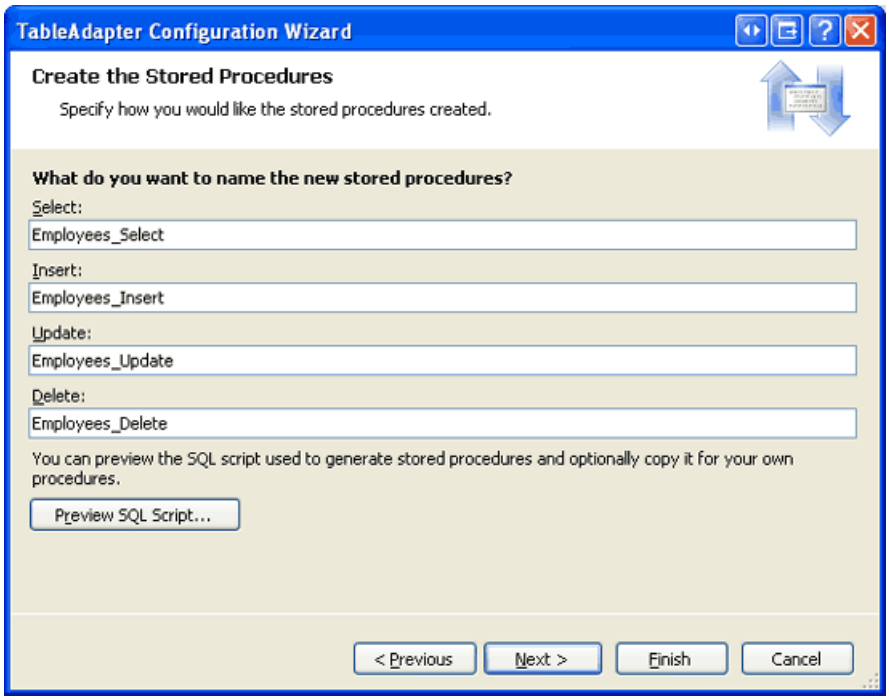


Figure 6: Name the TableAdapter s Stored Procedures (Click to view full-size image)

The final step prompts us to name the TableAdapter s methods. Use `Fill` and `GetEmployees` as the method names. Also be sure to leave the Create methods to send updates directly to the database (GenerateDBDirectMethods) checkbox checked.

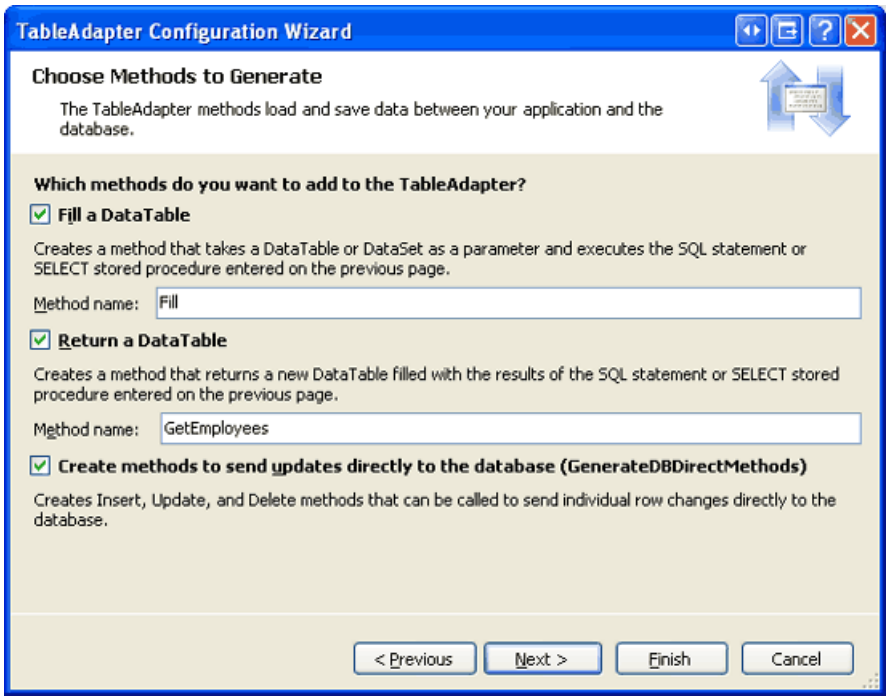


Figure 7: Name the TableAdapter s Methods `Fill` and `GetEmployees` (Click to view full-size image)

After completing the wizard, take a moment to examine the stored procedures in the database. You should see four new ones: `Employees_Select`, `Employees_Insert`, `Employees_Update`, and `Employees_Delete`. Next, inspect the `EmployeesDataTable` and `EmployeesTableAdapter` just created. The DataTable contains a column for each field returned by the main query. Click on the TableAdapter and then go to the Properties window. There you will see that the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties are correctly configured to call the corresponding stored procedures.

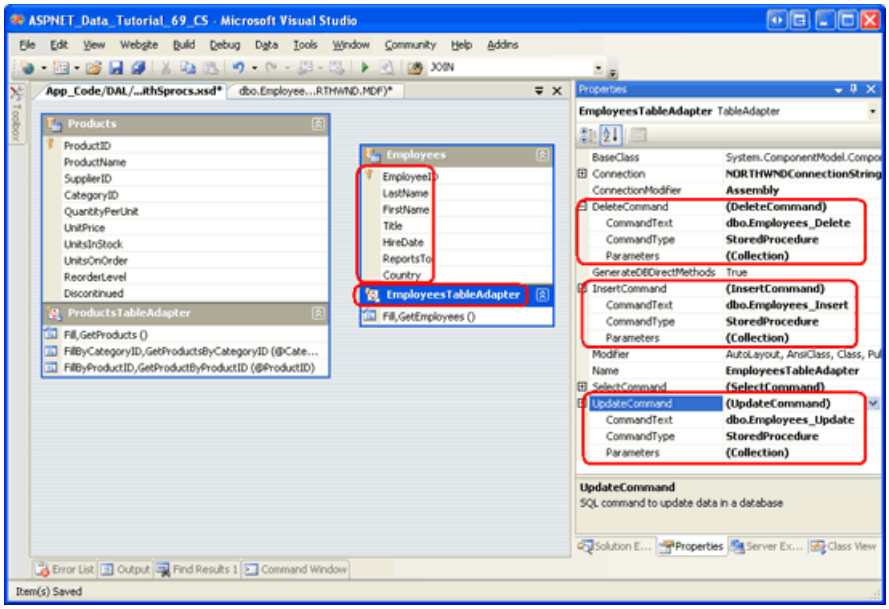


Figure 8: The TableAdapter Includes Insert, Update, and Delete Capabilities (Click to view full-size image)

With the insert, update, and delete stored procedures automatically created and the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties correctly configured, we are ready to customize the `SelectCommand` s stored procedure to return additional information about each employee s manager. Specifically, we need to update the `Employees_Select` stored procedure to use a `JOIN` and return the manager s `FirstName` and `LastName` values. After the stored procedure has been updated, we will need to update the DataTable so that it includes these additional columns. We'll tackle these two tasks in Steps 2 and 3.

Step 2: Customizing the Stored Procedure to Include a `JOIN`

Start by going to the Server Explorer, drilling down into the Northwind database s Stored Procedures folder, and opening the `Employees_Select` stored procedure. If you do not see this stored procedure, right-click on the Stored Procedures folder and choose Refresh. Update the stored procedure so that it uses a `LEFT JOIN` to return the manager s first and last name:

```
SELECT Employees.EmployeeID, Employees.LastName,
       Employees.FirstName, Employees.Title,
       Employees.HireDate, Employees.ReportsTo,
       Employees.Country,
       Manager.FirstName as ManagerFirstName,
       Manager.LastName as ManagerLastName
FROM Employees
    LEFT JOIN Employees AS Manager ON
        Employees.ReportsTo = Manager.EmployeeID
```

After updating the `SELECT` statement, save the changes by going to the File menu and choosing Save `Employees_Select`. Alternatively, you can click the Save icon in the toolbar or hit Ctrl+S. After saving your changes, right-click on the `Employees_Select` stored procedure in the Server Explorer and choose Execute. This will run the stored procedure and show its results in the Output window (see Figure 9).

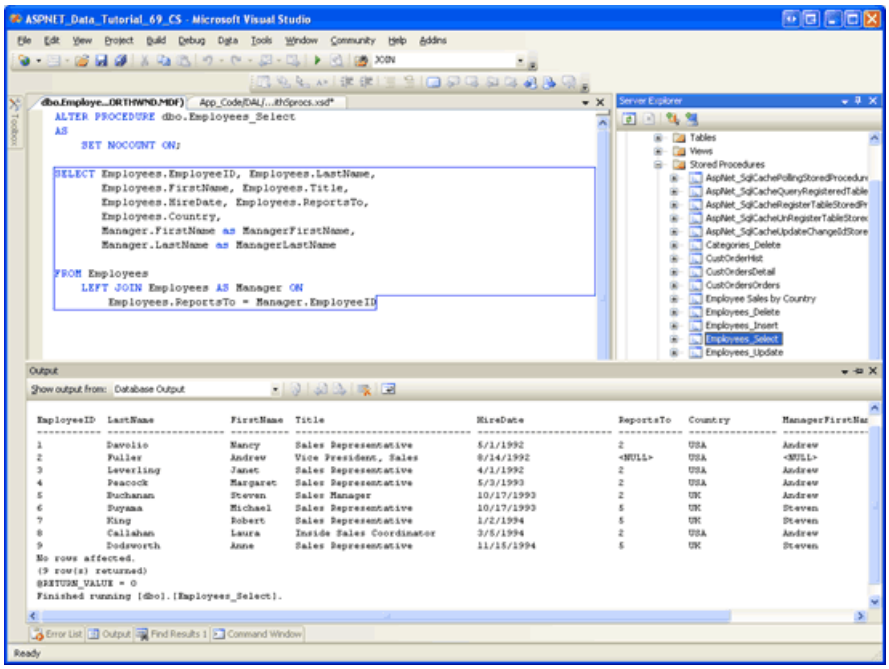


Figure 9: The Stored Procedures Results are Displayed in the Output Window (Click to view full-size image)

Step 3: Updating the DataTable s Columns

At this point, the `Employees_Select` stored procedure returns `ManagerFirstName` and `ManagerLastName` values, but the `EmployeesDataTable` is missing these columns. These missing columns can be added to the DataTable in one of two ways:

- **Manually** - right-click on the DataTable in the DataSet Designer and, from the Add menu, choose Column. You can then name the column and set its properties accordingly.
- **Automatically** - the TableAdapter Configuration wizard will update the DataTable s columns to reflect the fields returned by the `SelectCommand` stored procedure. When using ad-hoc SQL statements, the wizard will also remove the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties since the `SelectCommand` now contains a `JOIN`. But when using stored procedures, these command properties remain intact.

We have explored manually adding DataTable columns in previous tutorials, including Master/Detail Using a Bulleted List of Master Records with a Details DataList and Uploading Files, and we will look at this process again in more detail in our next tutorial. For this tutorial, however, let s use the automatic approach via the TableAdapter Configuration wizard.

Start by right-clicking on the `EmployeesTableAdapter` and selecting Configure from the context menu. This brings up the TableAdapter Configuration wizard, which lists the stored procedures used for selecting, inserting, updating, and deleting, along with their return values and parameters (if any). Figure 10 shows this wizard. Here we can see that the `Employees_Select` stored procedure now returns the `ManagerFirstName` and `ManagerLastName` fields.

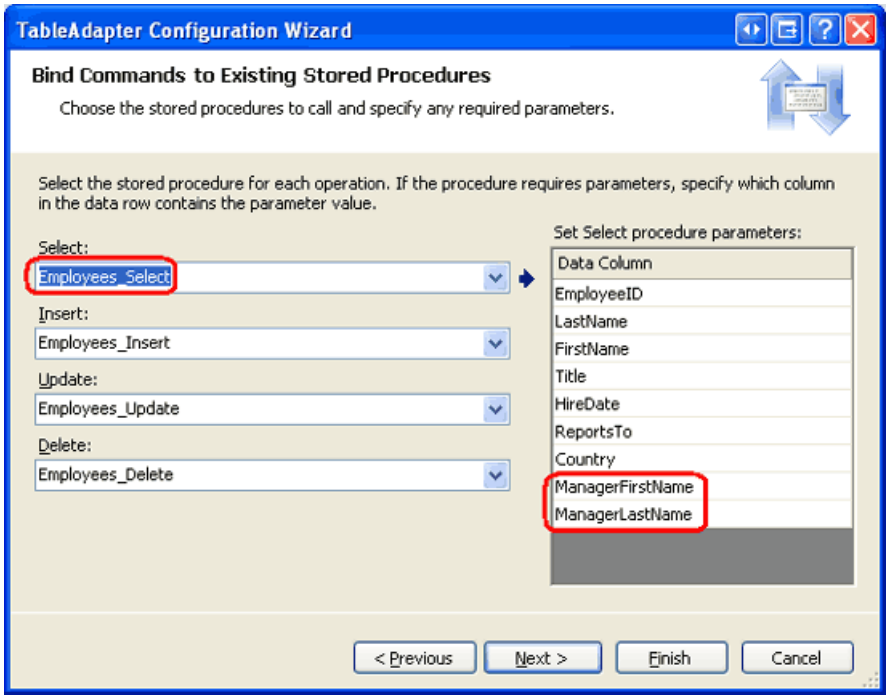


Figure 10: The Wizard Shows the Updated Column List for the `Employees_Select` Stored Procedure (Click to view full-size image)

Complete the wizard by clicking Finish. Upon returning to the DataSet Designer, the `EmployeesDataTable` includes two additional columns: `ManagerFirstName` and `ManagerLastName`.

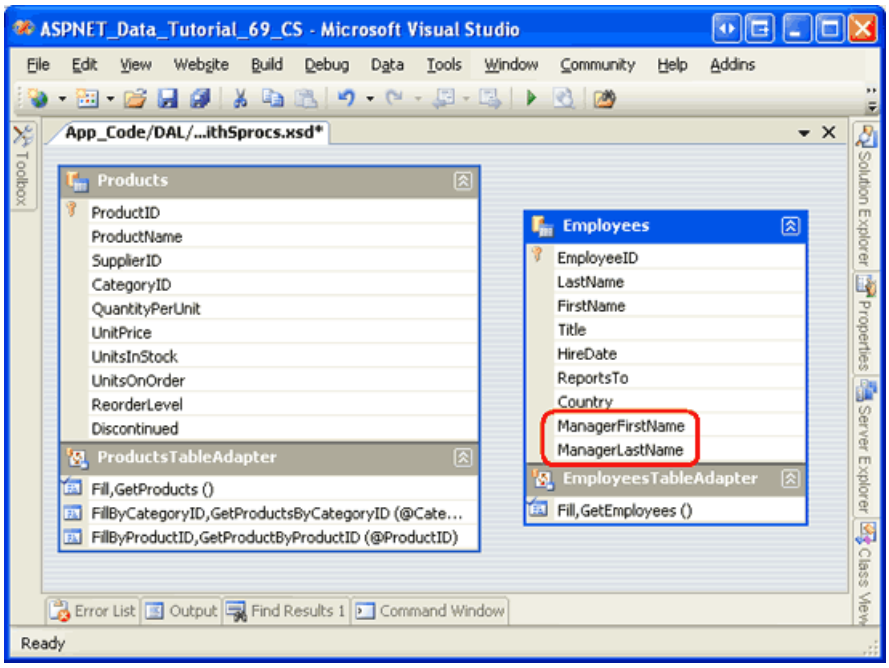


Figure 11: The `EmployeesDataTable` Contains Two New Columns (Click to view full-size image)

To illustrate that the updated `Employees_Select` stored procedure is in effect and that the insert, update, and delete capabilities of the TableAdapter are still functional, let s create a web page that allows users to view and delete employees. Before we create such a page, however, we need to first create a new class in the Business Logic Layer for working with employees from the `NorthwindWithSprocs` DataSet. In Step 4, we will create an `EmployeesBLLWithSprocs` class. In Step 5, we will use this class from an ASP.NET page.

Step 4: Implementing the Business Logic Layer

Create a new class file in the `~/App_Code/BLL` folder named `EmployeesBLLWithSprocs.vb`. This class mimics the semantics of the existing `EmployeesBLL` class, only this new one provides fewer methods and uses the `NorthwindWithSprocs` DataSet (instead of the `Northwind` DataSet). Add the following code to the `EmployeesBLLWithSprocs` class.

```
Imports NorthwindWithSprocsTableAdapters
<System.ComponentModel.DataObject()> _
Public Class EmployeesBLLWithSprocs
    Private _employeesAdapter As EmployeesTableAdapter = Nothing
    Protected ReadOnly Property Adapter() As EmployeesTableAdapter
        Get
            If _employeesAdapter Is Nothing Then
                _employeesAdapter = New EmployeesTableAdapter()
            End If
            Return _employeesAdapter
        End Get
    End Property
    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Select, True)> _
    Public Function GetEmployees() As NorthwindWithSprocs.EmployeesDataTable
        Return Adapter.GetEmployees()
    End Function
    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Delete, True)> _
    Public Function DeleteEmployee(ByVal employeeID As Integer) As Boolean
        Dim rowsAffected = Adapter.Delete(employeeID)
        'Return true if precisely one row was deleted, otherwise false
        Return rowsAffected = 1
    End Function
End Class
```

The `EmployeesBLLWithSprocs` class s `Adapter` property returns an instance of the `NorthwindWithSprocs` DataSet s `EmployeesTableAdapter`. This is used by the class s `GetEmployees` and `DeleteEmployee` methods. The `GetEmployees` method calls the `EmployeesTableAdapter` s corresponding `GetEmployees` method, which invokes the `Employees_Select` stored procedure and populates its results in an `EmployeeDataTable`. The `DeleteEmployee` method similarly calls the `EmployeesTableAdapter` s `Delete` method, which invokes the `Employees_Delete` stored procedure.

Step 5: Working with the Data in the Presentation Layer

With the `EmployeesBLLWithSprocs` class complete, we re ready to work with employee data through an ASP.NET page. Open the `JOINs.aspx` page in the `AdvancedDAL` folder and drag a GridView from the Toolbox onto the Designer, setting its `ID` property to `Employees`. Next, from the GridView s smart tag, bind the grid to a new ObjectDataSource control named `EmployeesDataSource`.

Configure the ObjectDataSource to use the `EmployeesBLLWithSprocs` class and, from the SELECT and DELETE tabs, ensure that the `GetEmployees` and `DeleteEmployee` methods are selected from the drop-down lists. Click Finish to complete the ObjectDataSource s configuration.

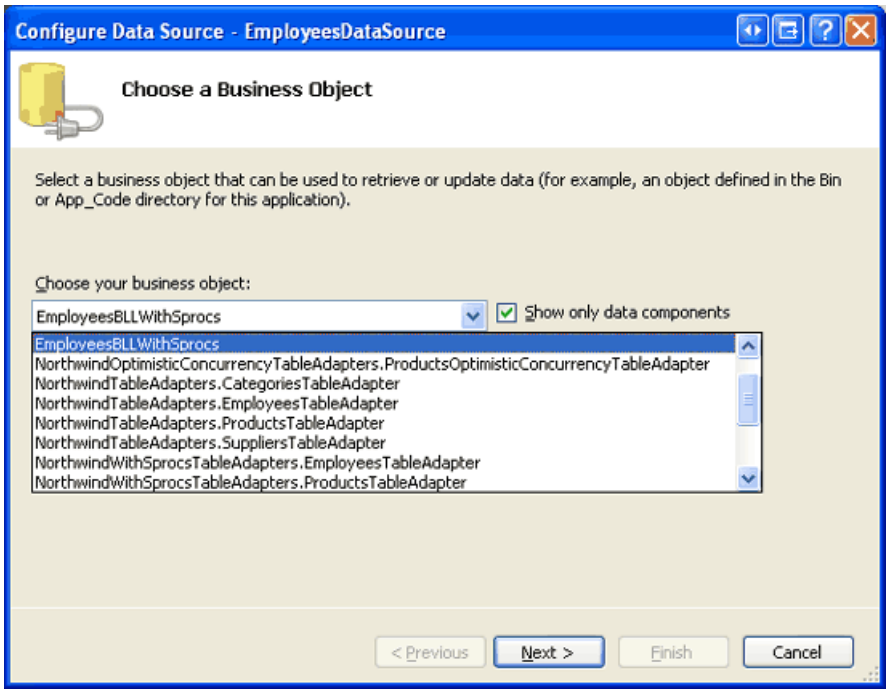


Figure 12: Configure the ObjectDataSource to Use the `EmployeesBLLWithSprocs` Class (Click to view full-size image)

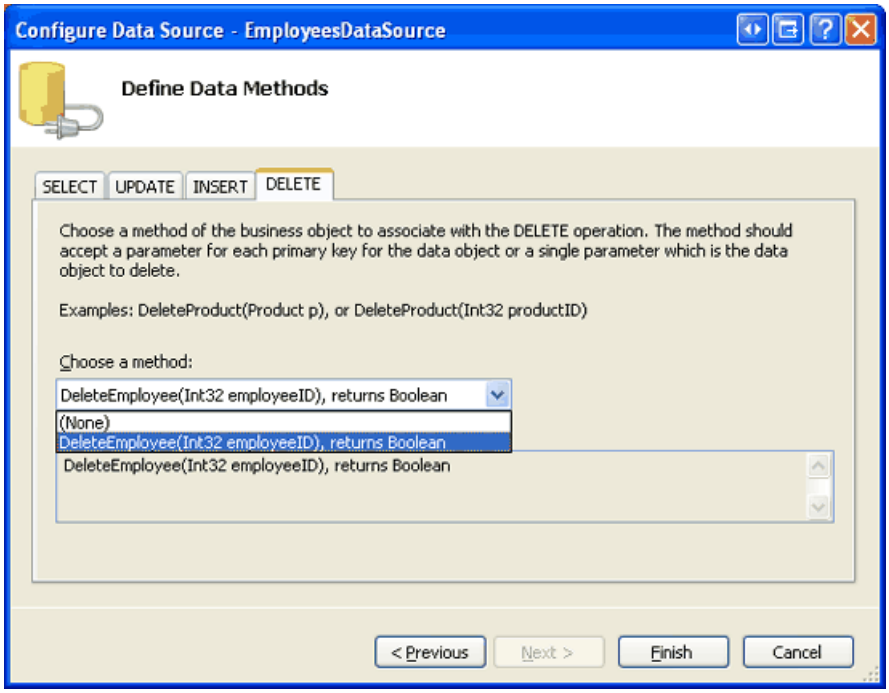


Figure 13: Have the ObjectDataSource Use the `GetEmployees` and `DeleteEmployee` Methods (Click to view full-size image)

Visual Studio will add a BoundField to the GridView for each of the `EmployeesDataTable` s columns. Remove all of these BoundFields except for `Title`, `LastName`, `FirstName`, `ManagerFirstName`, and `ManagerLastName` and rename the `HeaderText` properties for the last four BoundFields to Last Name, First Name, Manager s First Name, and Manager s Last Name, respectively.

To allow users to delete employees from this page we need to do two things. First, instruct the GridView to provide deleting capabilities by checking the Enable Deleting option from its smart tag. Second, change the ObjectDataSource s `OldValuesParameterFormatString` property from the value set by the ObjectDataSource wizard (`original_{0}`) to its default value (`{0}`). After making these changes, your GridView and ObjectDataSource s declarative markup should look similar to the following:

```
<asp:GridView ID="Employees" runat="server" AutoGenerateColumns="False"
    DataKeyNames="EmployeeID" DataSourceID="EmployeesDataSource">
    <Columns>
        <asp:CommandField ShowDeleteButton="True" />
        <asp:BoundField DataField="Title"
            HeaderText="Title"
            SortExpression="Title" />
        <asp:BoundField DataField="LastName"
            HeaderText="Last Name"
            SortExpression="LastName" />
        <asp:BoundField DataField="FirstName"
            HeaderText="First Name"
            SortExpression="FirstName" />
        <asp:BoundField DataField="ManagerFirstName"
            HeaderText="Manager's First Name"
            SortExpression="ManagerFirstName" />
        <asp:BoundField DataField="ManagerLastName"
            HeaderText="Manager's Last Name"
            SortExpression="ManagerLastName" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="EmployeesDataSource" runat="server"
    DeleteMethod="DeleteEmployee" OldValuesParameterFormatString="{0}"
    SelectMethod="GetEmployees" TypeName="EmployeesBLLWithSprocs">
    <DeleteParameters>
        <asp:Parameter Name="employeeID" Type="Int32" />
    </DeleteParameters>
</asp:ObjectDataSource>
```

Test out the page by visiting it through a browser. As Figure 14 shows, the page will list each employee and his or her manager s name (assuming they have one).

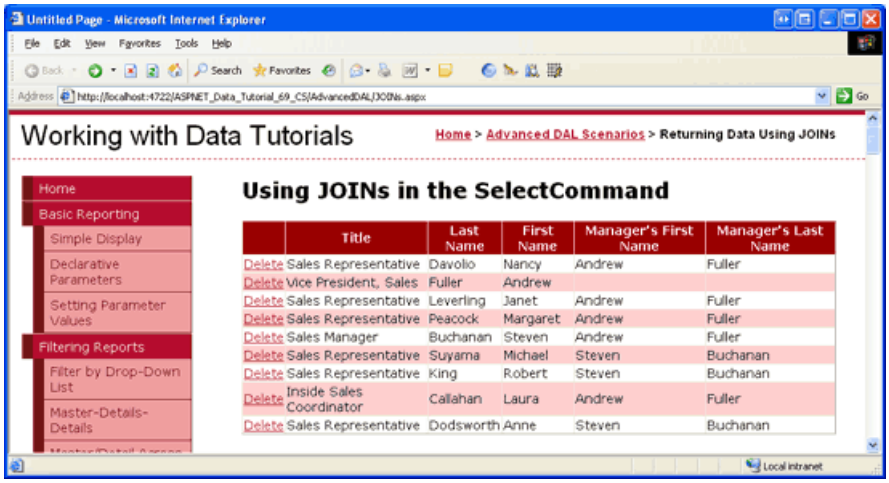


Figure 14: The `JOIN` in the `Employees_Select` Stored Procedure Returns the Manager s Name (Click to view full-size image)

Clicking the Delete button starts the deleting workflow, which culminates in the execution of the `Employees_Delete` stored procedure. However, the attempted `DELETE` statement in the stored procedure fails because of a foreign key constraint violation (see Figure 15). Specifically, each employee has one or more records in the `Orders` table, causing the delete to fail.

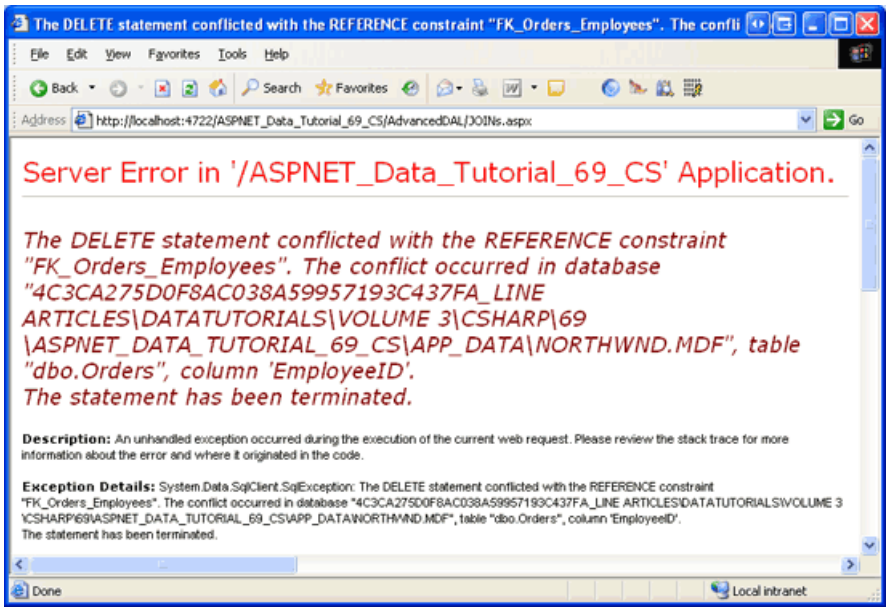


Figure 15: Deleting an Employee That has Corresponding Orders Results in a Foreign Key Constraint Violation (Click to view full-size image)

To allow an employee to be deleted you could:

- Update the foreign key constraint to cascade deletes,
- Manually delete the records from the `Orders` table for the employee(s) you want to delete, or
- Update the `Employees_Delete` stored procedure to first delete the related records from the `Orders` table before deleting the `Employees` record. We discussed this technique in the Using Existing Stored Procedures for the Typed DataSet s TableAdapters tutorial.

I leave this as an exercise for the reader.

Summary

When working with relational databases, it is common for queries to pull their data from multiple, related tables. Correlated subqueries and `JOIN` s provide two different techniques for accessing data from related tables in a query. In previous tutorials we most commonly made use of correlated subqueries because the TableAdapter cannot auto-generate `INSERT`, `UPDATE`, and `DELETE` statements for queries involving `JOIN` s. While these values can be provided manually, when using ad-hoc SQL statements any customizations will be overwritten when the TableAdapter Configuration wizard is completed.

Fortunately, TableAdapters created using stored procedures do not suffer from the same brittleness as those created using ad-hoc SQL statements. Therefore, it is feasible to create a TableAdapter whose main query uses a `JOIN` when using stored procedures. In this tutorial we saw how to create such a TableAdapter. We started by using a `JOIN`-less `SELECT` query for the TableAdapter s main query so that the corresponding insert, update, and delete stored procedures would be auto-created. With the TableAdapter s initial configuration complete, we augmented the `SelectCommand` stored procedure to use a `JOIN` and re-ran the TableAdapter Configuration wizard to update the `EmployeesDataTable` s columns.

Re-running the TableAdapter Configuration wizard automatically updated the `EmployeesDataTable` columns to reflect the data fields returned by the `Employees_Select` stored procedure. Alternatively, we could have added these columns manually to the DataTable. We will explore manually adding columns to the DataTable in the next tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Hilton Geisenow, David Suru, and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Adding Additional DataTable Columns (VB)

2/20/2020 • 14 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

When using the TableAdapter Wizard to create a Typed DataSet, the corresponding DataTable contains the columns returned by the main database query. But there are occasions when the DataTable needs to include additional columns. In this tutorial we learn why stored procedures are recommended when we need additional DataTable columns.

Introduction

When adding a TableAdapter to a Typed DataSet, the corresponding DataTable s schema is determined by the TableAdapter s main query. For example, if the main query returns data fields *A*, *B*, and *C*, the DataTable will have three corresponding columns named *A*, *B*, and *C*. In addition to its main query, a TableAdapter can include additional queries that return, perhaps, a subset of the data based on some parameter. For instance, in addition to the `ProductsTableAdapter` s main query, which returns information about all products, it also contains methods like `GetProductsByCategoryID(categoryID)` and `GetProductByProductID(productID)`, which return specific product information based on a supplied parameter.

The model of having the DataTable s schema reflect the TableAdapter s main query works well if all of the TableAdapter s methods return the same or fewer data fields than those specified in the main query. If a TableAdapter method needs to return additional data fields, then we should expand the DataTable s schema accordingly. In the Master/Detail Using a Bulleted List of Master Records with a Details DataList tutorial we added a method to the `CategoriesTableAdapter` that returned the `CategoryID`, `CategoryName`, and `Description` data fields defined in the main query plus `NumberOfProducts`, an additional data field that reported the number of products associated with each category. We manually added a new column to the `CategoriesDataTable` in order to capture the `NumberOfProducts` data field value from this new method.

As discussed in the Uploading Files tutorial, great care must be taken with TableAdapters that use ad-hoc SQL statements and have methods whose data fields do not precisely match the main query. If the TableAdapter Configuration wizard is re-run, it will update all of the TableAdapter s methods so that their data field list matches the main query. Consequently, any methods with customized column lists will revert to the main query s column list and not return the expected data. This issue does not arise when using stored procedures.

In this tutorial we will look at how to extend a DataTable s schema to include additional columns. Due to the brittleness of the TableAdapter when using ad-hoc SQL statements, in this tutorial we will use stored procedures. Refer to the Creating New Stored Procedures for the Typed DataSet s TableAdapters and Using Existing Stored Procedures for the Typed DataSet s TableAdapters tutorials for more information on configuring a TableAdapter to use stored procedures.

Step 1: Adding a `PriceQuartile` Column to the `ProductsDataTable`

In the *Creating New Stored Procedures for the Typed DataSet s TableAdapters* tutorial we created a Typed DataSet named `NorthwindWithSprocs`. This DataSet currently contains two DataTables: `ProductsDataTable` and `EmployeesDataTable`. The `ProductsTableAdapter` has the following three methods:

- `GetProducts` - the main query, which returns all records from the `Products` table

- `GetProductsByCategoryID(categoryID)` - returns all products with the specified *categoryID*.
- `GetProductByProductID(productID)` - returns the particular product with the specified *productID*.

The main query and the two additional methods all return the same set of data fields, namely all of the columns from the `Products` table. There are no correlated subqueries or `JOIN` s pulling related data from the `Categories` or `Suppliers` tables. Therefore, the `ProductsDataTable` has a corresponding column for each field in the `Products` table.

For this tutorial, let s add a method to the `ProductsTableAdapter` named `GetProductsWithPriceQuartile` that returns all of the products. In addition to the standard product data fields, `GetProductsWithPriceQuartile` will also include a `PriceQuartile` data field that indicates under which quartile the product s price falls. For example, those products whose prices are in the most expensive 25% will have a `PriceQuartile` value of 1, while those whose prices fall in the bottom 25% will have a value of 4. Before we worry about creating the stored procedure to return this information, however, we first need to update the `ProductsDataTable` to include a column to hold the `PriceQuartile` results when the `GetProductsWithPriceQuartile` method is used.

Open the `NorthwindWithSprocs` DataSet and right-click on the `ProductsDataTable`. Choose Add from the context-menu and then pick Column.

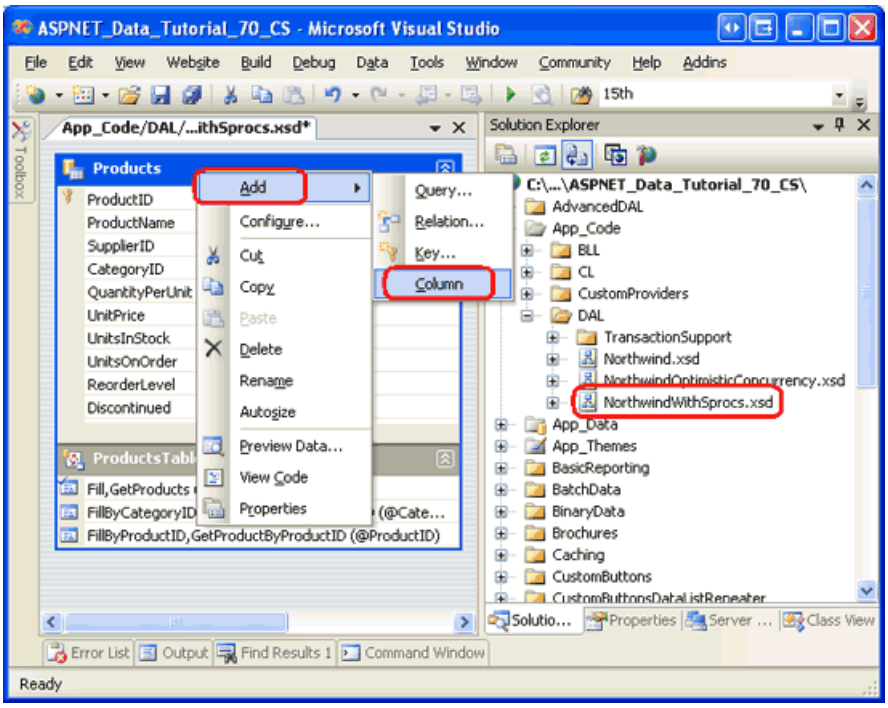


Figure 1: Add a New Column to the `ProductsDataTable` (Click to view full-size image)

This will add a new column to the DataTable named Column1 of type `System.String`. We need to update this column s name to PriceQuartile and its type to `System.Int32` since it will be used to hold a number between 1 and 4. Select the newly-added column in the `ProductsDataTable` and, from the Properties window, set the `Name` property to PriceQuartile and the `DataType` property to `System.Int32`.

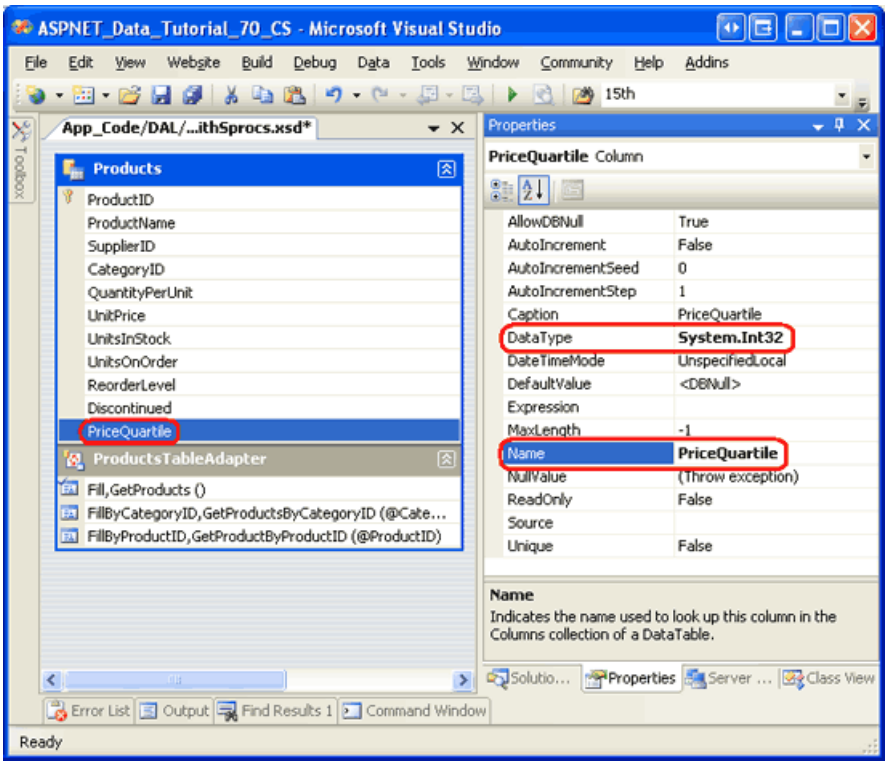


Figure 2: Set the New Column s `Name` and `DataType` Properties (Click to view full-size image)

As Figure 2 shows, there are additional properties that can be set, such as whether the values in the column must be unique, if the column is an auto-increment column, whether or not database `NULL` values are allowed, and so on. Leave these values set to their defaults.

Step 2: Creating the `GetProductsWithPriceQuartile` Method

Now that the `ProductsDataTable` has been updated to include the `PriceQuartile` column, we are ready to create the `GetProductsWithPriceQuartile` method. Start by right-clicking on the TableAdapter and choosing Add Query from the context-menu. This brings up the TableAdapter Query Configuration wizard, which first prompts us as to whether we want to use ad-hoc SQL statements or a new or existing stored procedure. Since we don t yet have a stored procedure that returns the price quartile data, let s allow the TableAdapter to create this stored procedure for us. Select the Create new stored procedure option and click Next.

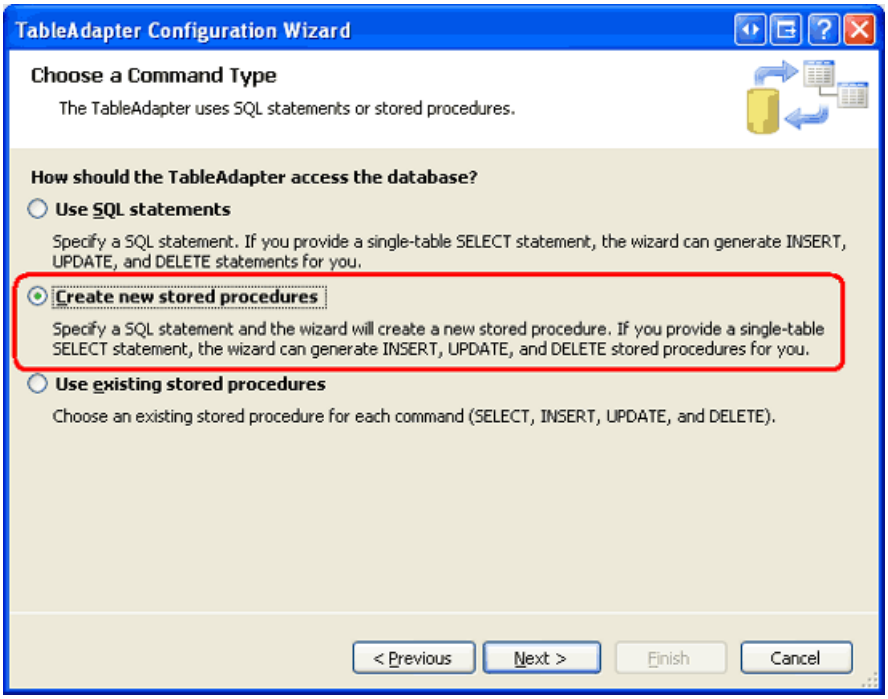


Figure 3: Instruct the TableAdapter Wizard to Create the Stored Procedure For Us (Click to view full-size image)

In the subsequent screen, shown in Figure 4, the wizard asks us what type of query to add. Since the `GetProductsWithPriceQuartile` method will return all columns and records from the `Products` table, select the SELECT which returns rows option and click Next.

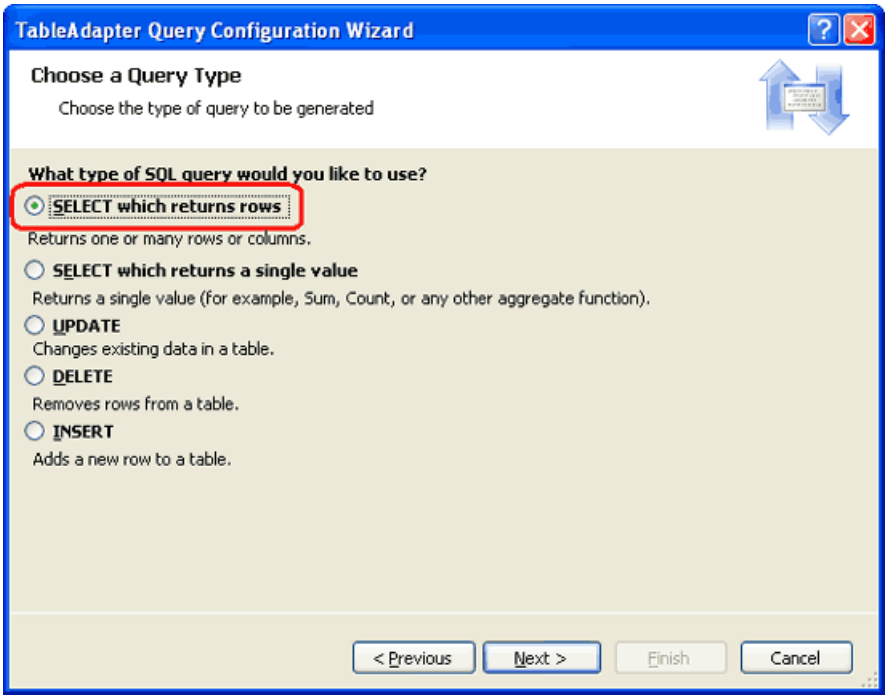


Figure 4: Our Query will be a `SELECT` Statement that Returns Multiple Rows (Click to view full-size image)

Next we are prompted for the `SELECT` query. Enter the following query into the wizard:

```
SELECT ProductID, ProductName, SupplierID, CategoryID,
       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
       ReorderLevel, Discontinued,
       NTILE(4) OVER (ORDER BY UnitPrice DESC) as PriceQuartile
FROM Products
```

The above query uses SQL Server 2005 s new `NTILE` function to divide the results into four groups where the groups are determined by the `UnitPrice` values sorted in descending order.

Unfortunately, the Query Builder does not know how to parse the `OVER` keyword and will display an error when parsing the above query. Therefore, enter the above query directly in the textbox in the wizard without using the Query Builder.

> **NOTE**
> For more information on NTILE and SQL Server 2005 s other ranking functions, see Returning Ranked Results with Microsoft SQL Server 2005 and the Ranking Functions section from the SQL Server 2005 Books Online.

After entering the `SELECT` query and clicking Next, the wizard asks us to provide a name for the stored procedure it will create. Name the new stored procedure `Products_SelectWithPriceQuartile` and click Next.

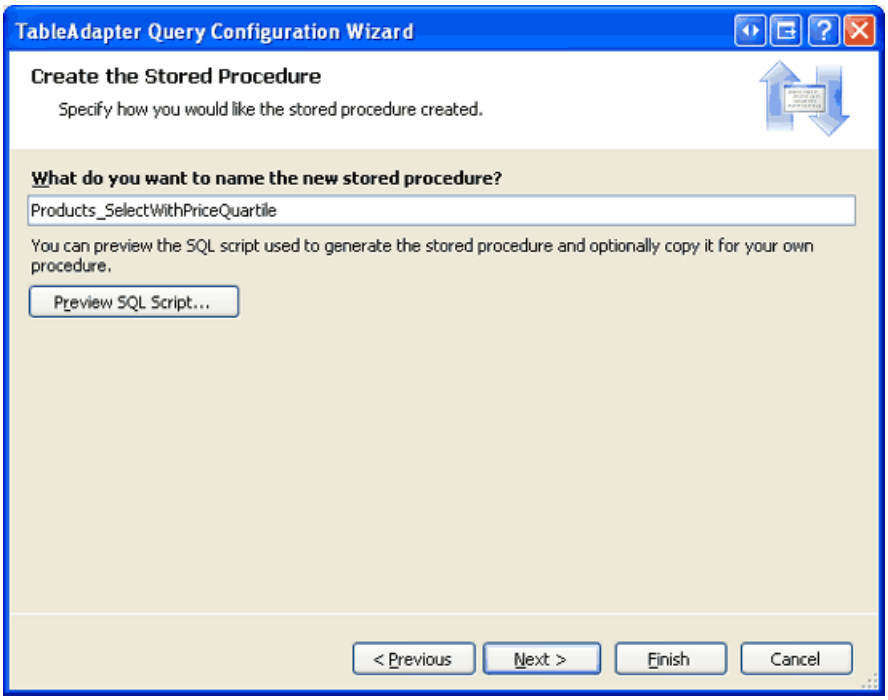


Figure 5: Name the Stored Procedure `Products_SelectWithPriceQuartile` (Click to view full-size image)

Lastly, we are prompted to name the TableAdapter methods. Leave both the Fill a DataTable and Return a DataTable checkboxes checked and name the methods `FillWithPriceQuartile` and `GetProductsWithPriceQuartile`.

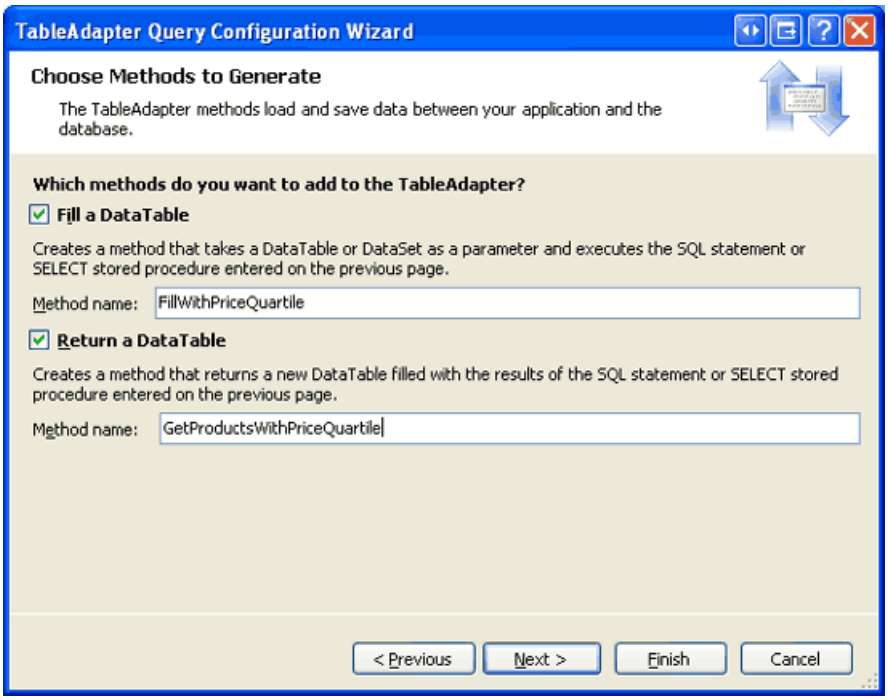


Figure 6: Name the TableAdapter s Methods and Click Finish (Click to view full-size image)

With the `SELECT` query specified and the stored procedure and TableAdapter methods named, click Finish to complete the wizard. At this point you may get a warning or two from the wizard saying that The `OVER` SQL construct or statement is not supported. These warnings can be ignored.

After completing the wizard, the TableAdapter should include the `FillWithPriceQuartile` and `GetProductsWithPriceQuartile` methods and the database should include a stored procedure named `Products_SelectWithPriceQuartile`. Take a moment to verify that the TableAdapter does indeed contain this new method and that the stored procedure has been correctly added to the database. When checking the database, if you do not see the stored procedure try right-clicking on the Stored Procedures folder and choosing Refresh.

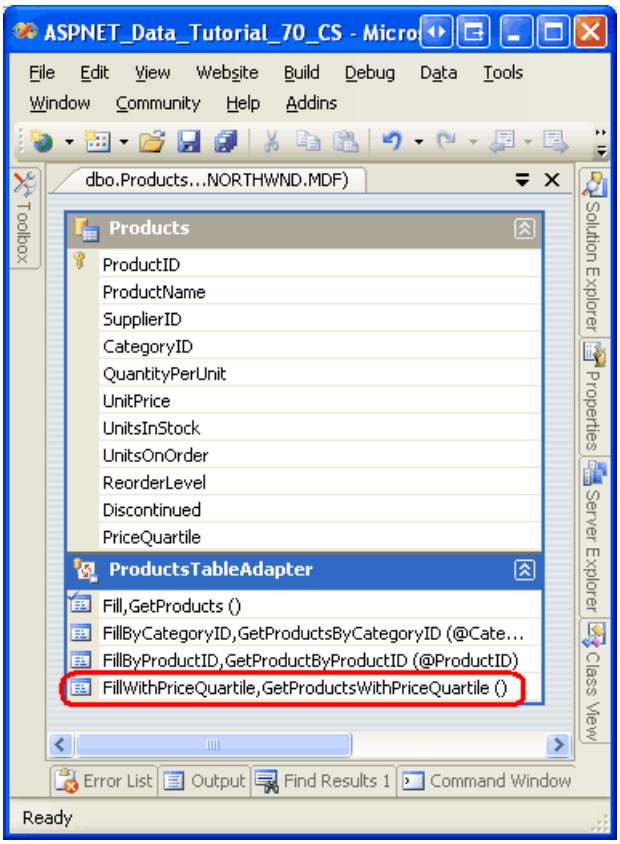


Figure 7: Verify that a New Method Has Been Added to the TableAdapter

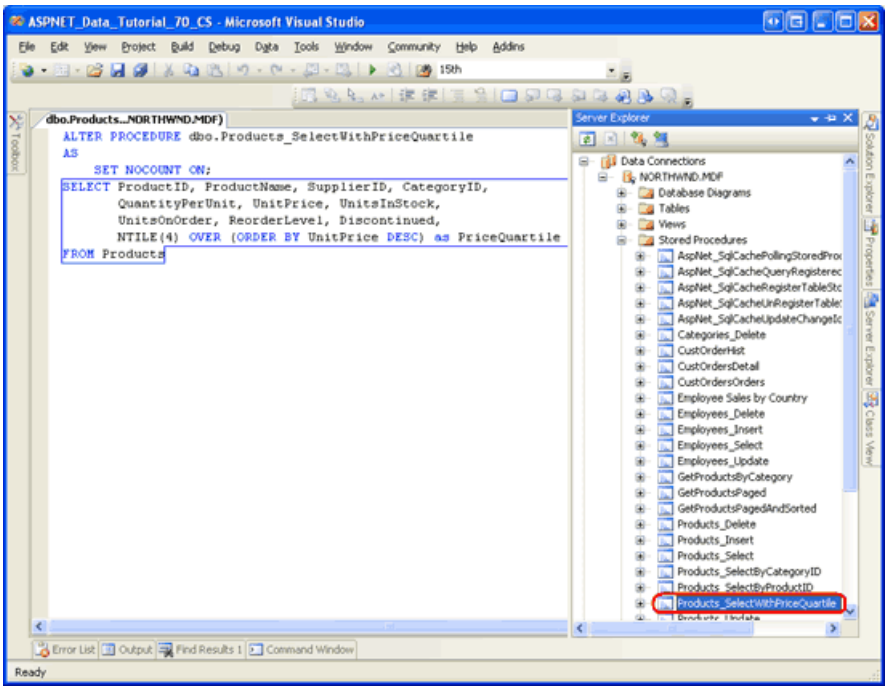


Figure 8: Ensure that the Database Contains the `Products_SelectWithPriceQuartile` Stored Procedure (Click to view full-size image)

NOTE

One of the benefits of using stored procedures instead of ad-hoc SQL statements is that re-running the TableAdapter Configuration wizard will not modify the stored procedures column lists. Verify this by right-clicking on the TableAdapter, choosing the Configure option from the context-menu to start the wizard, and then clicking Finish to complete it. Next, go to the database and view the `Products_SelectWithPriceQuartile` stored procedure. Note that its column list has not been modified. Had we been using ad-hoc SQL statements, re-running the TableAdapter Configuration wizard would have reverted this query s column list to match the main query column list, thereby removing the NTILE statement from the query used by the `GetProductsWithPriceQuartile` method.

When the Data Access Layer s `GetProductsWithPriceQuartile` method is invoked, the TableAdapter executes the `Products_SelectWithPriceQuartile` stored procedure and adds a row to the `ProductsDataTable` for each returned record. The data fields returned by the stored procedure are mapped to the `ProductsDataTable` s columns. Since there is a `PriceQuartile` data field returned from the stored procedure, its value is assigned to the `ProductsDataTable` s `PriceQuartile` column.

For those TableAdapter methods whose queries do not return a `PriceQuartile` data field, the `PriceQuartile` column s value is the value specified by its `DefaultValue` property. As Figure 2 shows, this value is set to `DBNull`, the default. If you would prefer a different default value, simply set the `DefaultValue` property accordingly. Just make sure that the `DefaultValue` value is valid given the column s `DataType` (i.e., `System.Int32` for the `PriceQuartile` column).

At this point we have performed the necessary steps for adding an additional column to a DataTable. To verify that this additional column works as expected, let s create an ASP.NET page that displays each product s name, price, and price quartile. Before we do that, though, we first need to update the Business Logic Layer to include a method that calls down to the DAL s `GetProductsWithPriceQuartile` method. We will update the BLL next, in Step 3, and then create the ASP.NET page in Step 4.

Step 3: Augmenting the Business Logic Layer

Before we use the new `GetProductsWithPriceQuartile` method from the Presentation Layer, we should first add a corresponding method to the BLL. Open the `ProductsBLLWithSprocs` class file and add the following code:

```
<System.ComponentModel.DataObjectMethodAttribute_
    (System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetProductsWithPriceQuartile() As NorthwindWithSprocs.ProductsDataTable
    Return Adapter.GetProductsWithPriceQuartile()
End Function
```

Like the other data retrieval methods in `ProductsBLLWithSprocs`, the `GetProductsWithPriceQuartile` method simply calls the DAL s corresponding `GetProductsWithPriceQuartile` method and returns its results.

Step 4: Displaying the Price Quartile Information in an ASP.NET Web Page

With the BLL addition complete we re ready to create an ASP.NET page that shows the price quartile for each product. Open the `AddingColumns.aspx` page in the `AdvancedDAL` folder and drag a GridView from the Toolbox onto the Designer, setting its `ID` property to `Products`. From the GridView s smart tag, bind it to a new ObjectDataSource named `ProductsDataSource`. Configure the ObjectDataSource to use the `ProductsBLLWithSprocs` class s `GetProductsWithPriceQuartile` method. Since this will be a read-only grid, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) .

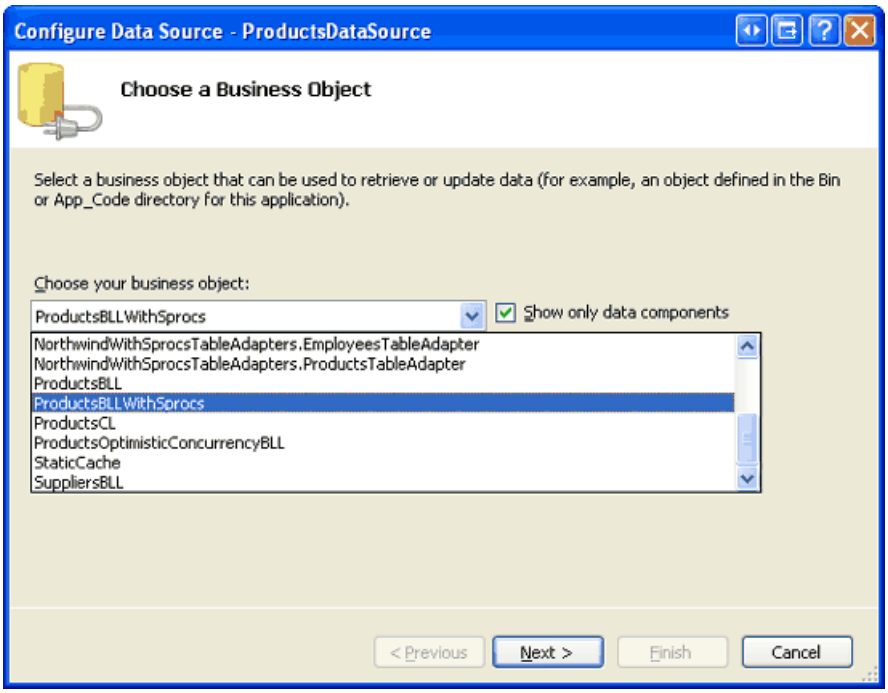


Figure 9: Configure the ObjectDataSource to Use the `ProductsBLLWithSprocs` Class (Click to view full-size image)

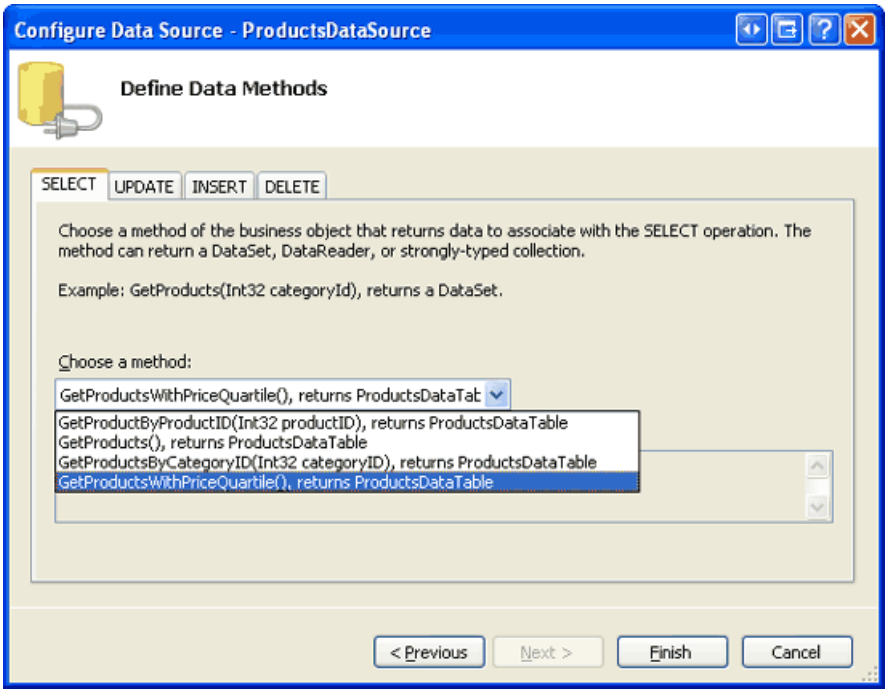


Figure 10: Retrieve Product Information from the `GetProductsWithPriceQuartile` Method (Click to view full-size image)

After completing the Configure Data Source wizard, Visual Studio will automatically add a BoundField or CheckBoxField to the GridView for each of the data fields returned by the method. One of these data fields is `PriceQuartile`, which is the column we added to the `ProductsDataTable` in Step 1.

Edit the GridView s fields, removing all but the `ProductName`, `UnitPrice`, and `PriceQuartile` BoundFields. Configure the `UnitPrice` BoundField to format its value as a currency and have the `UnitPrice` and `PriceQuartile` BoundFields right- and center-aligned, respectively. Finally, update the remaining BoundFields `HeaderText` properties to Product, Price, and Price Quartile, respectively. Also, check the Enable Sorting checkbox from the GridView s smart tag.

After these modifications, the GridView and ObjectDataSource s declarative markup should look like the following:

```
<asp:GridView ID="Products" runat="server" AllowSorting="True"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="ProductsDataSource">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="Product"
            SortExpression="ProductName" />
        <asp:BoundField DataField="UnitPrice" DataFormatString="{0:c}"
            HeaderText="Price" HtmlEncode="False"
            SortExpression="UnitPrice">
            <ItemStyle HorizontalAlign="Right" />
        </asp:BoundField>
        <asp:BoundField DataField="PriceQuartile" HeaderText="Price Quartile"
            SortExpression="PriceQuartile">
            <ItemStyle HorizontalAlign="Center" />
        </asp:BoundField>
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="ProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetProductsWithPriceQuartile"
    TypeName="ProductsBLLWithSprocs">
</asp:ObjectDataSource>
```

Figure 11 shows this page when visited through a browser. Note that, initially, the products are ordered by their price in descending order with each product assigned an appropriate `PriceQuartile` value. Of course this data can be sorted by other criteria with the Price Quartile column value still reflecting the product s ranking with respect to price (see Figure 12).

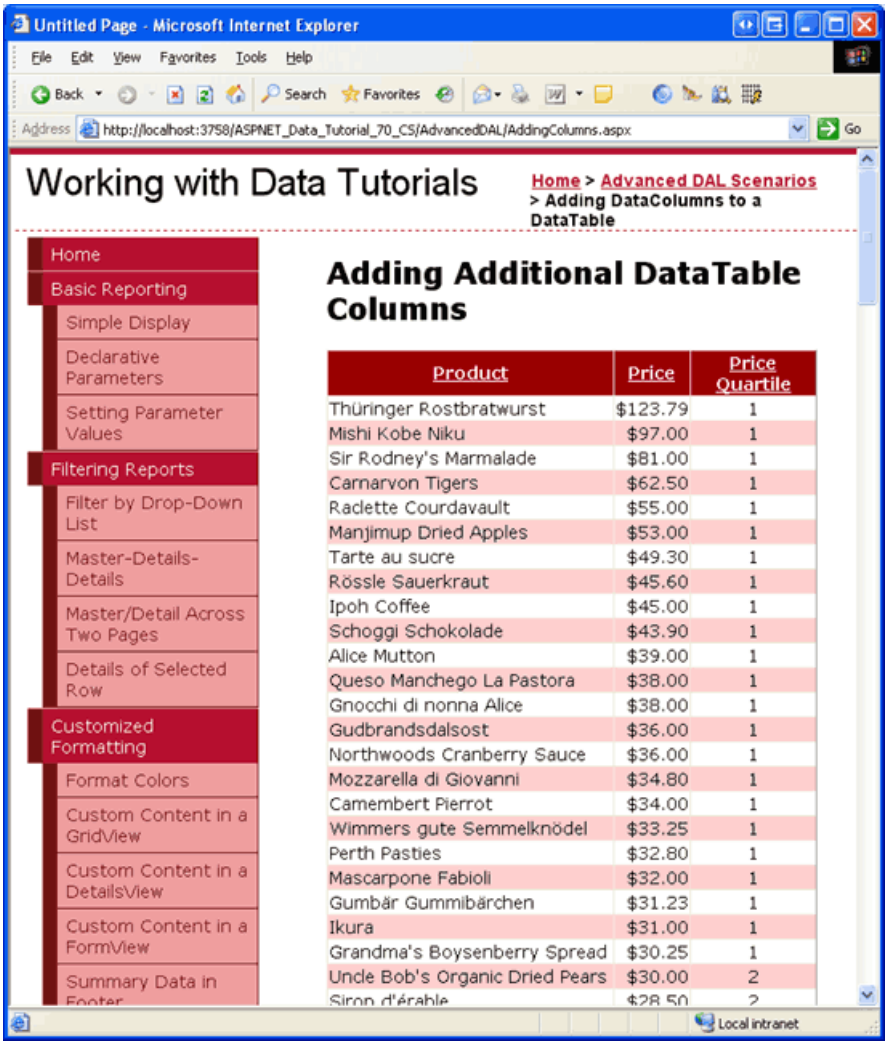


Figure 11: The Products are Ordered by their Prices (Click to view full-size image)

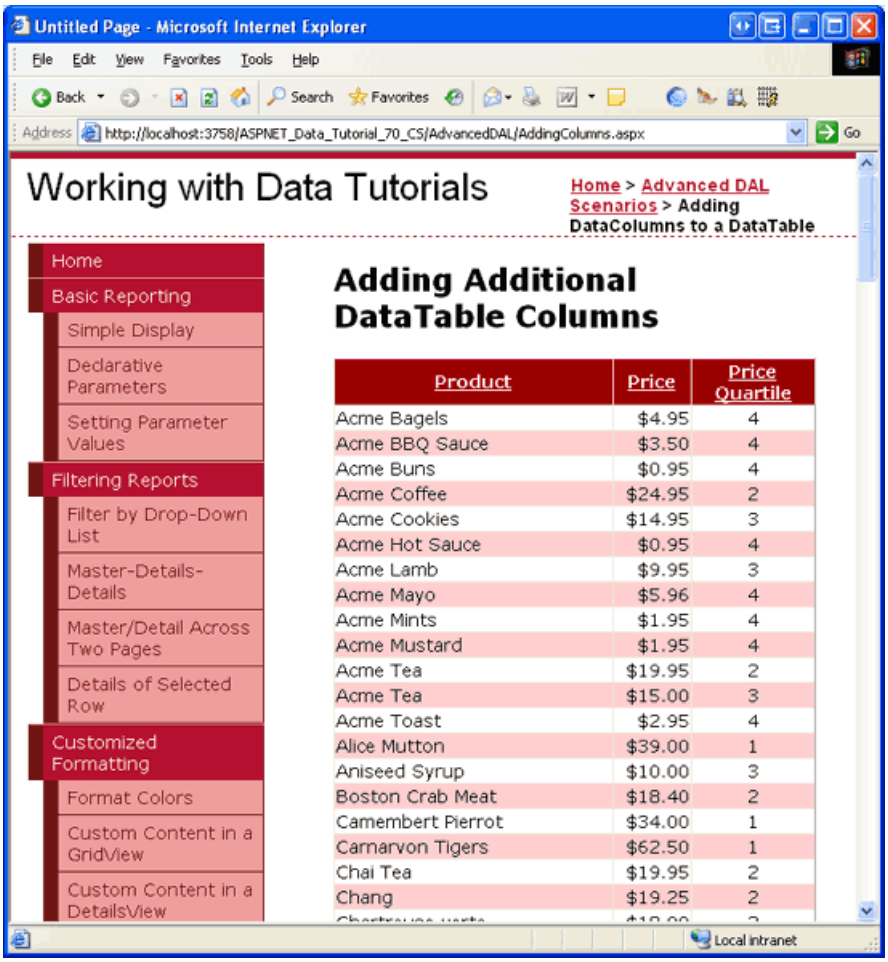


Figure 12: The Products are Ordered by their Names (Click to view full-size image)

> **NOTE**
>
> With a few lines of code we could augment the GridView so that it colored the product rows based on their `PriceQuartile` value. We might color those products in the first quartile a light green, those in the second quartile a light yellow, and so forth. I encourage you to take a moment to add this functionality. If you need a refresher on formatting a GridView, consult the Custom Formatting Based Upon Data tutorial.

An Alternative Approach - Creating Another TableAdapter

As we saw in this tutorial, when adding a method to a TableAdapter that returns data fields other than those spelled out by the main query, we can add corresponding columns to the DataTable. Such an approach, however, works well only if there are a small number of methods in the TableAdapter that return different data fields and if those alternate data fields do not vary too much from the main query.

Rather than adding columns to the DataTable, you can instead add another TableAdapter to the DataSet that contains the methods from the first TableAdapter that return different data fields. For this tutorial, rather than add the `PriceQuartile` column to the `ProductsDataTable` (where it is only used by the `GetProductsWithPriceQuartile` method), we could have added an additional TableAdapter to the DataSet named `ProductsWithPriceQuartileTableAdapter` that used the `Products_SelectWithPriceQuartile` stored procedure as its main query. ASP.NET pages that needed to get product information with the price quartile would use the `ProductsWithPriceQuartileTableAdapter`, while those that did not could continue to use the `ProductsTableAdapter`.

By adding a new TableAdapter, the DataTables remain untarnished and their columns precisely mirror the data fields returned by their TableAdapter s methods. However, additional TableAdapters can introduce repetitive tasks and functionality. For example, if those ASP.NET pages that displayed the `PriceQuartile` column also needed to provide insert, update, and delete support, the `ProductsWithPriceQuartileTableAdapter` would need to have its

`InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties properly configured. While these properties would mirror the `ProductsTableAdapter` s, this configuration introduces an extra step. Moreover, there are now two ways to update, delete, or add a product to the database - through the `ProductsTableAdapter` and `ProductsWithPriceQuartileTableAdapter` classes.

The download for this tutorial includes a `ProductsWithPriceQuartileTableAdapter` class in the `NorthwindWithSprocs` DataSet that illustrates this alternative approach.

Summary

In most scenarios, all of the methods in a TableAdapter will return the same set of data fields, but there are times when a particular method or two may need to return an additional field. For example, in the Master/Detail Using a Bulleted List of Master Records with a Details DataList tutorial we added a method to the `CategoriesTableAdapter` that, in addition to the main query s data fields, returned a `NumberOfProducts` field that reported the number of products associated with each category. In this tutorial we looked at adding a method in the `ProductsTableAdapter` that returned a `PriceQuartile` field in addition to the main query s data fields. To capture additional data fields returned by the TableAdapter s methods we need to add corresponding columns to the DataTable.

If you plan on manually adding columns to the DataTable, it is recommended that the TableAdapter use stored procedures. If the TableAdapter uses ad-hoc SQL statements, any time the TableAdapter Configuration wizard is run all of the methods data field lists revert to the data fields returned by the main query. This problem does not extend to stored procedures, which is why they are recommended and were used in this tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Randy Schmidt, Jacky Goor, Bernadette Leigh, and Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Working with Computed Columns (VB)

2/20/2020 • 18 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

When creating a database table, Microsoft SQL Server allows you to define a computed column whose value is calculated from an expression that usually references other values in the same database record. Such values are read-only at the database, which requires special considerations when working with TableAdapters. In this tutorial we learn how to meet the challenges posed by computed columns.

Introduction

Microsoft SQL Server allows for *computed columns*, which are columns whose values are calculated from an expression that usually references the values from other columns in the same table. As an example, a time tracking data model might have a table named `ServiceLog` with columns including `ServicePerformed`, `EmployeeID`, `Rate`, and `Duration`, among others. While the amount due per service item (being the rate multiplied by the duration) could be calculated through a web page or other programmatic interface, it might be handy to include a column in the `ServiceLog` table named `AmountDue` that reported this information. This column could be created as a normal column, but it would need to be updated anytime the `Rate` or `Duration` column values changed. A better approach would be to make the `AmountDue` column a computed column using the expression `Rate * Duration`. Doing so would cause SQL Server to automatically calculate the `AmountDue` column value whenever it was referenced in a query.

Since a computed column s value is determined by an expression, such columns are read-only and therefore cannot have values assigned to them in `INSERT` or `UPDATE` statements. However, when computed columns are part of the main query for a TableAdapter that uses ad-hoc SQL statements, they are automatically included in the auto-generated `INSERT` and `UPDATE` statements. Consequently, the TableAdapter s `INSERT` and `UPDATE` queries and `InsertCommand` and `UpdateCommand` properties must be updated to remove references to any computed columns.

One challenge of using computed columns with a TableAdapter that uses ad-hoc SQL statements is that the TableAdapter s `INSERT` and `UPDATE` queries are automatically regenerated any time the TableAdapter Configuration wizard is completed. Therefore, the computed columns manually removed from the `INSERT` and `UPDATE` queries will reappear if the wizard is re-run. Although TableAdapters that use stored procedures don t suffer from this brittleness, they do have their own quirks that we will address in Step 3.

In this tutorial we will add a computed column to the `Suppliers` table in the Northwind database and then create a corresponding TableAdapter to work with this table and its computed column. We will have our TableAdapter use stored procedures instead of ad-hoc SQL statements so that our customizations aren't lost when the TableAdapter Configuration wizard is used.

Let s get started!

Step 1: Adding a Computed Column to the `Suppliers` Table

The Northwind database does not have any computed columns so we will need to add one ourselves. For this tutorial let s add a computed column to the `Suppliers` table called `FullContactName` that returns the contact s name, title, and the company they work for in the following format: `ContactName` (`ContactTitle` , `CompanyName`).

This computed column might be used in reports when displaying information about suppliers.

Start by opening the `Suppliers` table definition by right-clicking on the `Suppliers` table in the Server Explorer and choosing Open Table Definition from the context-menu. This will display the columns of the table and their properties, such as their data type, whether they allow `NULL` s, and so forth. To add a computed column, start by typing in the name of the column into the table definition. Next, enter its expression into the (Formula) textbox under the Computed Column Specification section in the Column Properties window (see Figure 1). Name the computed column `FullContactName` and use the following expression:

```
ContactName + ' (' + CASE WHEN ContactTitle IS NOT NULL THEN
    ContactTitle + ', ' ELSE '' END + CompanyName + ')'
```

Note that strings can be concatenated in SQL using the `+` operator. The `CASE` statement can be used like a conditional in a traditional programming language. In the above expression the `CASE` statement can be read as: If `ContactTitle` is not `NULL` then output the `ContactTitle` value concatenated with a comma, otherwise emit nothing. For more on the usefulness of the `CASE` statement, see The Power of SQL `CASE` Statements.

> **NOTE**
>
> Instead of using a `CASE` statement here, we could have alternatively used `ISNULL(ContactTitle, '')`. `ISNULL(checkExpression, replacementValue)` returns *checkExpression* if it is non-NULL, otherwise it returns *replacementValue*. While either `ISNULL` or `CASE` will work in this instance, there are more intricate scenarios where the flexibility of the `CASE` statement cannot be matched by `ISNULL`.

After adding this computed column your screen should look like the screen shot in Figure 1.

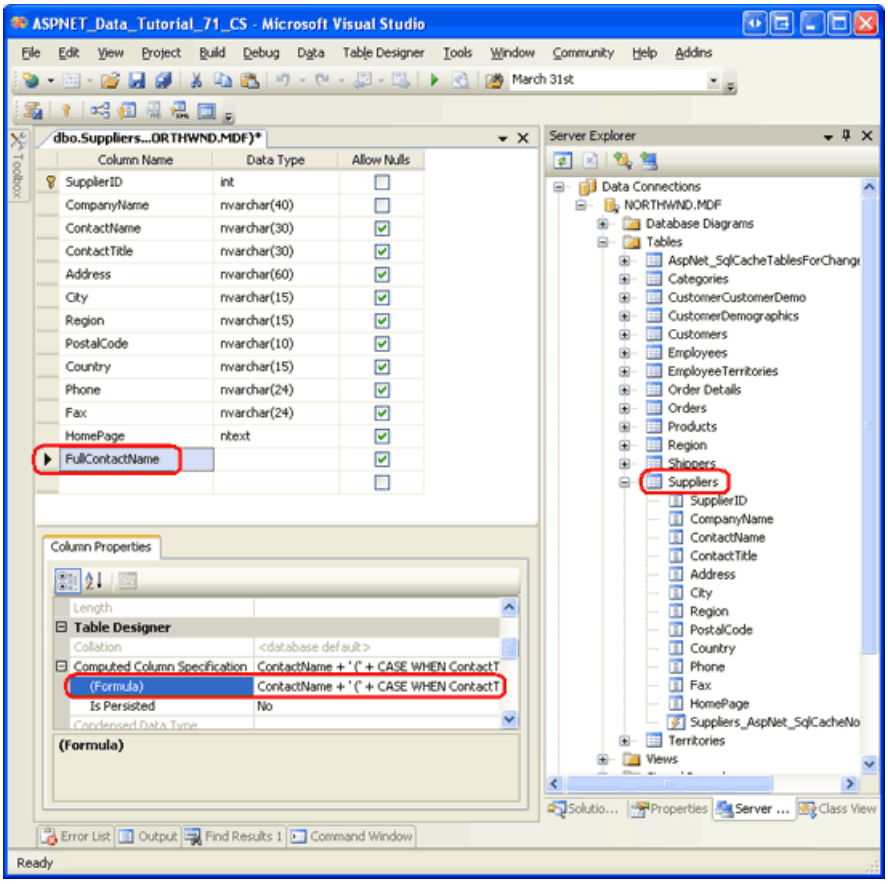


Figure 1: Add a Computed Column Named `FullContactName` to the `Suppliers` Table (Click to view full-size image)

After naming the computed column and entering its expression, save the changes to the table by clicking the Save

icon in the toolbar, by hitting Ctrl+S, or by going to the File menu and choosing Save `Suppliers`.

Saving the table should refresh the Server Explorer, including the just-added column in the `Suppliers` table s column list. Furthermore, the expression entered into the (Formula) textbox will automatically adjust to an equivalent expression that strips unnecessary whitespace, surrounds column names with brackets (`[]`), and includes parentheses to more explicitly show the order of operations:

```
((([ContactName]+' (')+case when [ContactTitle] IS NOT NULL
    then [ContactTitle]+', ' else '' end)+[CompanyName])+')')
```

For more information on computed columns in Microsoft SQL Server, refer to the technical documentation. Also check out the How to: Specify Computed Columns for a step-by-step walkthrough of creating computed columns.

> **NOTE**
> By default, computed columns are not physically stored in the table but are instead recalculated each time they are referenced in a query. By checking the Is Persisted checkbox, however, you can instruct SQL Server to physically store the computed column in the table. Doing so allows an index to be created on the computed column, which can improve the performance of queries that use the computed column value in their `WHERE` clauses. See Creating Indexes on Computed Columns for more information.

Step 2: Viewing the Computed Column s Values

Before we start work on the Data Access Layer, let s take a minute to view the `FullContactName` values. From the Server Explorer, right-click on the `Suppliers` table name and choose New Query from the context-menu. This will bring up a Query window that prompts us to choose what tables to include in the query. Add the `Suppliers` table and click Close. Next, check the `CompanyName`, `ContactName`, `ContactTitle`, and `FullContactName` columns from the Suppliers table. Finally, click the red exclamation point icon in the Toolbar to execute the query and view the results.

As Figure 2 shows, the results include `FullContactName`, which lists the `CompanyName`, `ContactName`, and `ContactTitle` columns using the format `ContactName` (`ContactTitle`, `CompanyName`) .

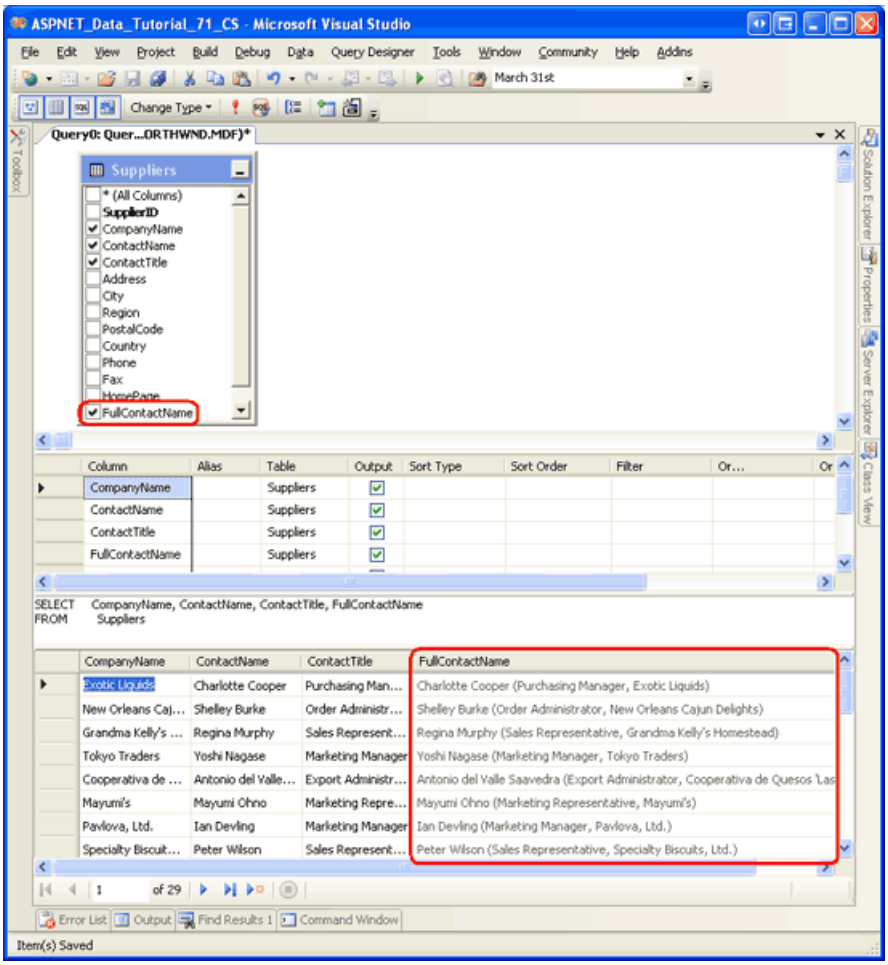


Figure 2: The `FullContactName` Uses the Format `ContactName` (`ContactTitle`, `CompanyName`) (Click to view full-size image)

Step 3: Adding the `SuppliersTableAdapter` to the Data Access Layer

In order to work with the supplier information in our application we need to first create a TableAdapter and DataTable in our DAL. Ideally, this would be accomplished using the same straightforward steps examined in earlier tutorials. However, working with computed columns introduces a few wrinkles that merit discussion.

If you are using a TableAdapter that uses ad-hoc SQL statements, you can simply include the computed column in the TableAdapter s main query via the TableAdapter Configuration wizard. This, however, will auto-generate `INSERT` and `UPDATE` statements that include the computed column. If you attempt to execute one of these methods, a `SqlException` with the message The column *ColumnName* cannot be modified because it is either a computed column or is the result of a UNION operator will be thrown. While the `INSERT` and `UPDATE` statement can be manually adjusted through the TableAdapter s `InsertCommand` and `UpdateCommand` properties, these customizations will be lost whenever the TableAdapter Configuration wizard is re-run.

Due to the brittleness of TableAdapters that use ad-hoc SQL statements, it is recommended that we use stored procedures when working with computed columns. If you are using existing stored procedures, simply configure the TableAdapter as discussed in the Using Existing Stored Procedures for the Typed DataSet s TableAdapters tutorial. If you have the TableAdapter wizard create the stored procedures for you, however, it is important to initially omit any computed columns from the main query. If you include a computed column in the main query, the TableAdapter Configuration wizard will inform you, upon completion, that it cannot create the corresponding stored procedures. In short, we need to initially configure the TableAdapter using a computed column-free main query and then manually update the corresponding stored procedure and the TableAdapter s `SelectCommand` to include the computed column. This approach is similar to the one used in the Updating the TableAdapter to Use `JOIN` *s* tutorial.

For this tutorial, let s add a new TableAdapter and have it automatically create the stored procedures for us. Consequently, we will need to initially omit the `FullContactName` computed column from the main query.

Start by opening the `NorthwindWithSprocs` DataSet in the `~/App_Code/DAL` folder. Right-click in the Designer and, from the context-menu, choose to add a new TableAdapter. This will launch the TableAdapter Configuration wizard. Specify the database to query data from (`NORTHWNDConnectionString` from `Web.config`) and click Next. Since we have not yet created any stored procedures for querying or modifying the `Suppliers` table, select the Create new stored procedures option so that the wizard will create them for us and click Next.

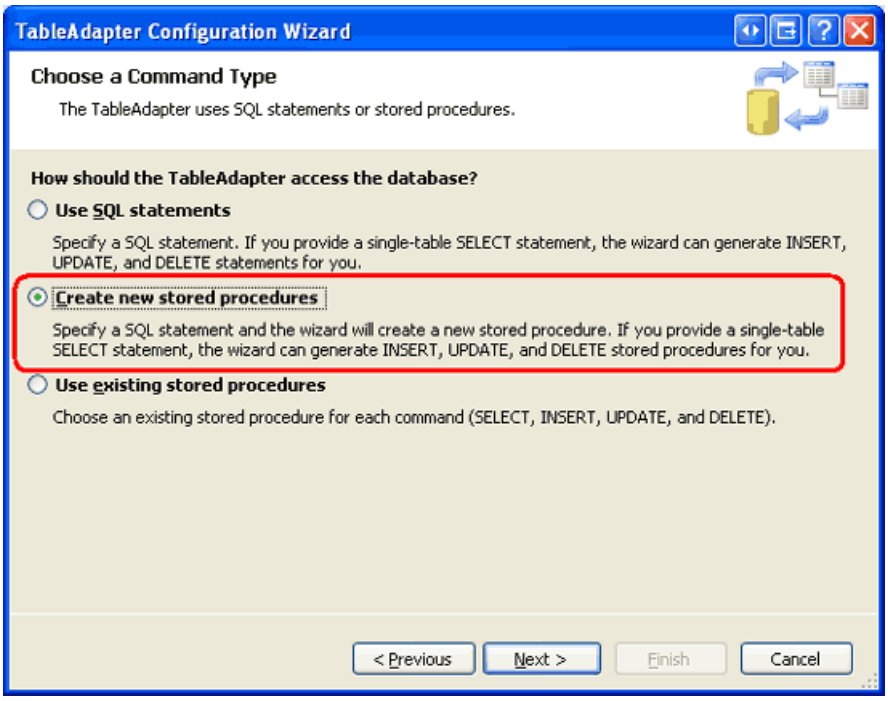


Figure 3: Choose the Create new stored procedures Option (Click to view full-size image)

The subsequent step prompts us for the main query. Enter the following query, which returns the `SupplierID` , `CompanyName` , `ContactName` , and `ContactTitle` columns for each supplier. Note that this query purposefully omits the computed column (`FullContactName`); we will update the corresponding stored procedure to include this column in Step 4.

```
SELECT SupplierID, CompanyName, ContactName, ContactTitle
FROM Suppliers
```

After entering the main query and clicking Next, the wizard allows us to name the four stored procedures it will generate. Name these stored procedures `Suppliers_Select` , `Suppliers_Insert` , `Suppliers_Update` , and `Suppliers_Delete` , as Figure 4 illustrates.

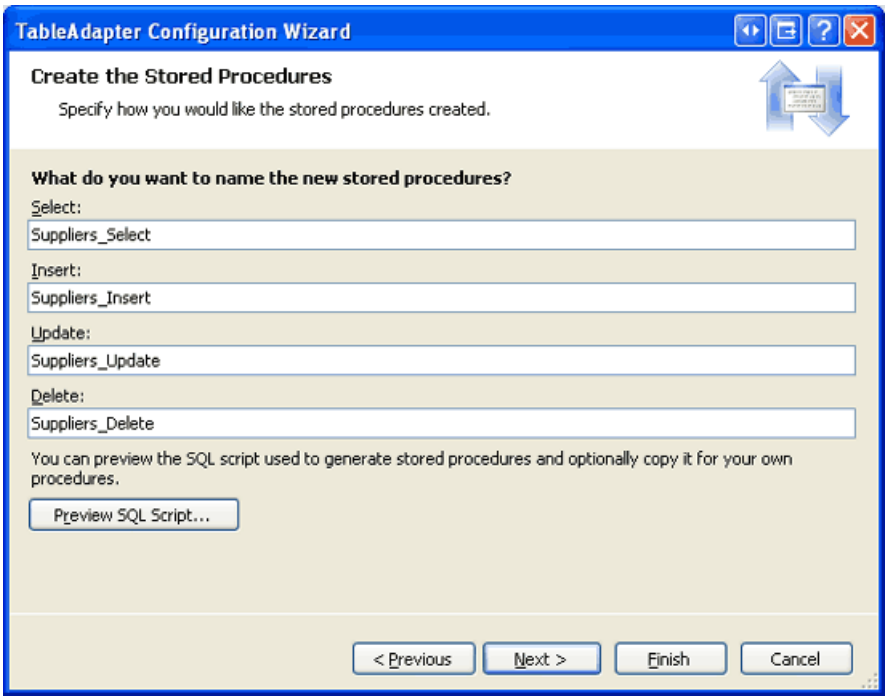


Figure 4: Customize the Names of the Auto-Generated Stored Procedures (Click to view full-size image)

The next wizard step allows us to name the TableAdapter s methods and specify the patterns used to access and update data. Leave all three checkboxes checked, but rename the `GetData` method to `GetSuppliers`. Click Finish to complete the wizard.

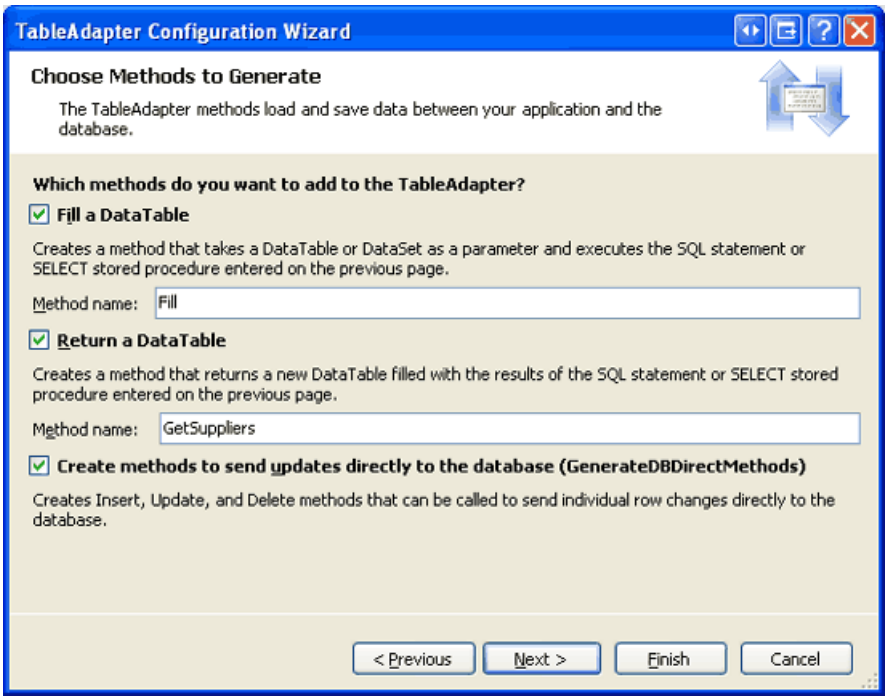


Figure 5: Rename the `GetData` Method to `GetSuppliers` (Click to view full-size image)

Upon clicking Finish, the wizard will create the four stored procedures and add the TableAdapter and corresponding DataTable to the Typed DataSet.

Step 4: Including the Computed Column in the TableAdapter s Main Query

We now need to update the TableAdapter and DataTable created in Step 3 to include the `FullContactName` computed column. This involves two steps:

1. Updating the `Suppliers_Select` stored procedure to return the `FullContactName` computed column, and
2. Updating the DataTable to include a corresponding `FullContactName` column.

Start by navigating to the Server Explorer and drilling down into the Stored Procedures folder. Open the `Suppliers_Select` stored procedure and update the `SELECT` query to include the `FullContactName` computed column:

```
SELECT SupplierID, CompanyName, ContactName, ContactTitle, FullContactName
FROM Suppliers
```

Save the changes to the stored procedure by clicking the Save icon in the Toolbar, by hitting Ctrl+S, or by choosing the Save `Suppliers_Select` option from the File menu.

Next, return to the DataSet Designer, right-click on the `SuppliersTableAdapter`, and choose Configure from the context-menu. Note that the `Suppliers_Select` column now includes the `FullContactName` column in its Data Columns collection.

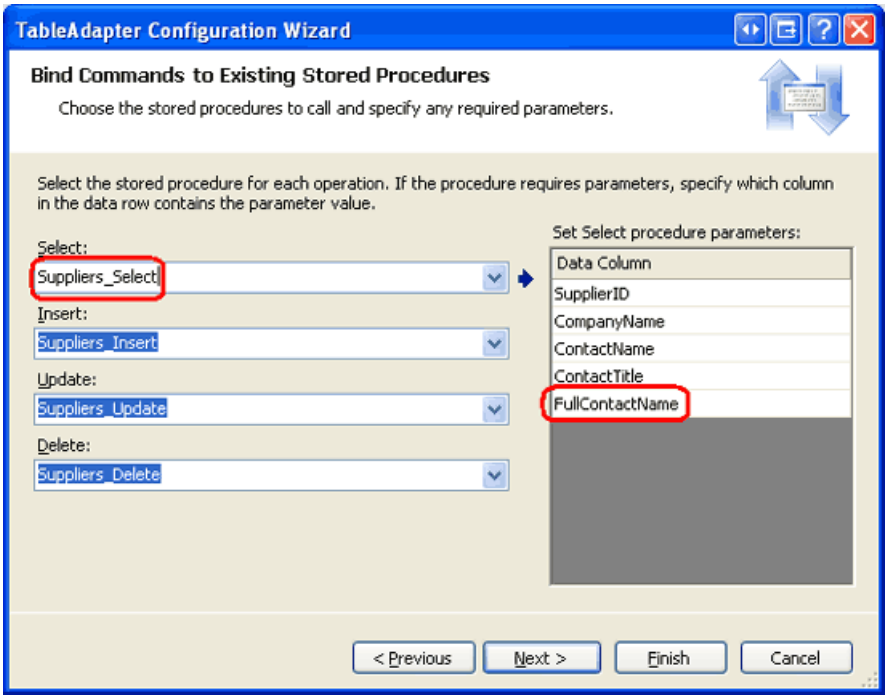


Figure 6: Run the TableAdapter s Configuration Wizard to Update the DataTable s Columns (Click to view full-size image)

Click Finish to complete the wizard. This will automatically add a corresponding column to the `SuppliersDataTable`. The TableAdapter wizard is smart enough to detect that the `FullContactName` column is a computed column and therefore read-only. Consequently, it sets the column s `ReadOnly` property to `true`. To verify this, select the column from the `SuppliersDataTable` and then go to the Properties window (see Figure 7). Note that the `FullContactName` column s `DataType` and `MaxLength` properties are also set accordingly.

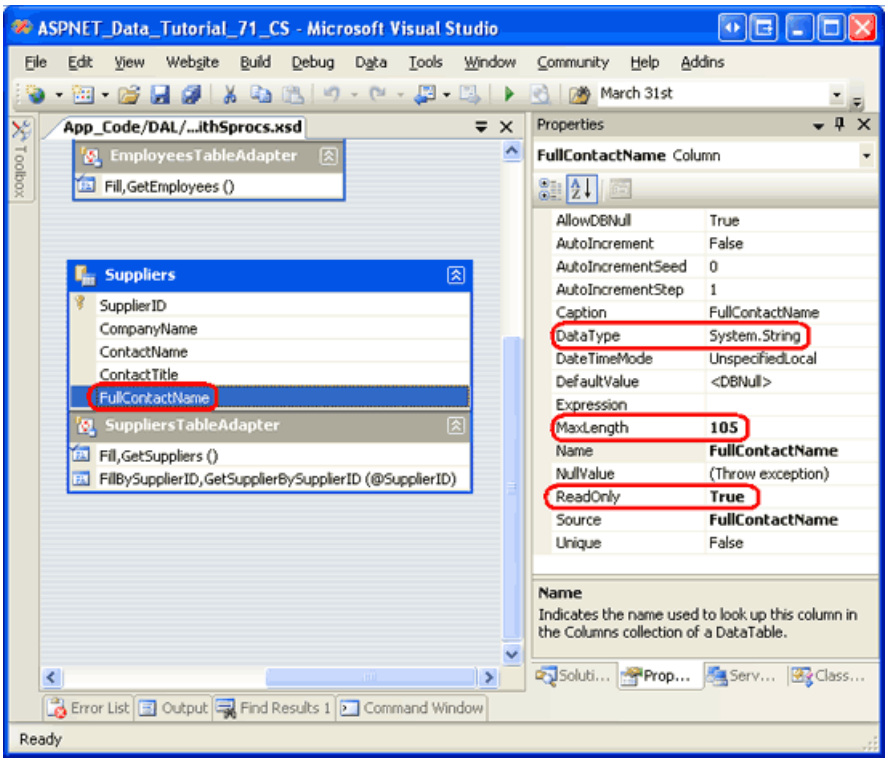


Figure 7: The `FullContactName` Column is Marked as Read-Only (Click to view full-size image)

Step 5: Adding a `GetSupplierBySupplierID` Method to the TableAdapter

For this tutorial we will create an ASP.NET page that displays the suppliers in an updateable grid. In past tutorials we have updated a single record from the Business Logic Layer by retrieving that particular record from the DAL as a strongly-typed DataTable, updating its properties, and then sending the updated DataTable back to the DAL to propagate the changes to the database. To accomplish this first step - retrieving the record being updated from the DAL - we need to first add a `GetSupplierBySupplierID(supplierID)` method to the DAL.

Right-click on the `SuppliersTableAdapter` in the DataSet Design and choose the Add Query option from the context-menu. As we did in Step 3, let the wizard generate a new stored procedure for us by selecting the Create new stored procedure option (refer back to Figure 3 for a screenshot of this wizard step). Since this method will return a record with multiple columns, indicate that we want to use a SQL query that is a SELECT which returns rows and click Next.

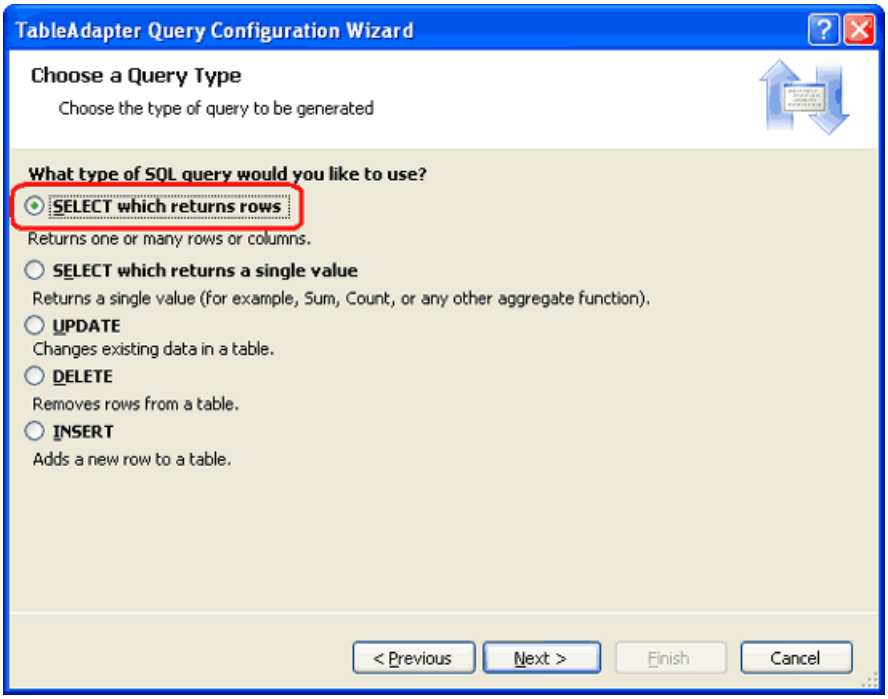


Figure 8: Choose the SELECT which returns rows Option (Click to view full-size image)

The subsequent step prompts us for the query to use for this method. Enter the following, which returns the same data fields as the main query but for a particular supplier.

```
SELECT SupplierID, CompanyName, ContactName, ContactTitle, FullContactName
FROM Suppliers
WHERE SupplierID = @SupplierID
```

The next screen asks us to name the stored procedure that will be auto-generated. Name this stored procedure `Suppliers_SelectBySupplierID` and click Next.

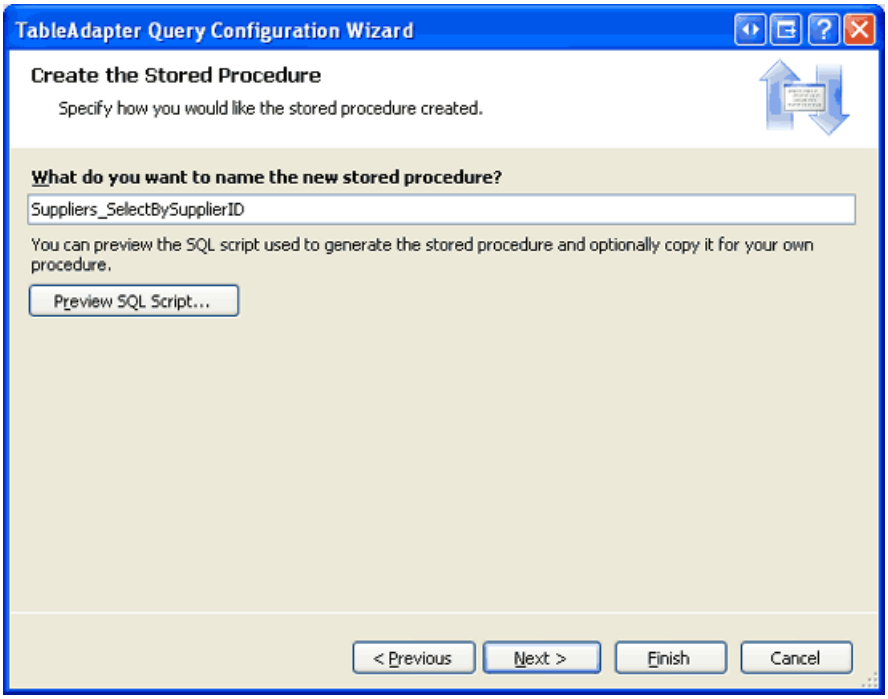


Figure 9: Name the Stored Procedure `Suppliers_SelectBySupplierID` (Click to view full-size image)

Lastly, the wizard prompts us for the data access patterns and method names to use for the TableAdapter. Leave both checkboxes checked, but rename the `FillBy` and `GetDataBy` methods to `FillBySupplierID` and `GetSupplierBySupplierID`, respectively.

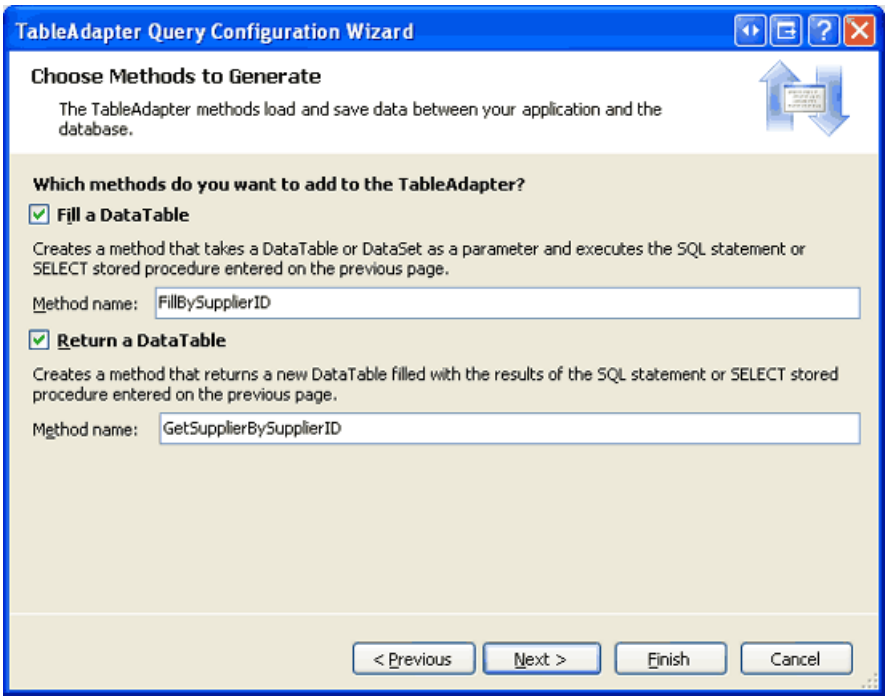


Figure 10: Name the TableAdapter Methods `FillBySupplierID` and `GetSupplierBySupplierID` (Click to view full-size image)

Click Finish to complete the wizard.

Step 6: Creating the Business Logic Layer

Before we create an ASP.NET page that uses the computed column created in Step 1, we first need to add the corresponding methods in the BLL. Our ASP.NET page, which we will create in Step 7, will allow users to view and edit suppliers. Therefore, we need our BLL to provide, at minimum, a method to get all of the suppliers and another to update a particular supplier.

Create a new class file named `SuppliersBLLWithSprocs` in the `~/App_Code/BLL` folder and add the following code:

```
Imports NorthwindWithSprocsTableAdapters
<System.ComponentModel.DataObject()> _
Public Class SuppliersBLLWithSprocs
    Private _suppliersAdapter As SuppliersTableAdapter = Nothing
    Protected ReadOnly Property Adapter() As SuppliersTableAdapter
        Get
            If _suppliersAdapter Is Nothing Then
                _suppliersAdapter = New SuppliersTableAdapter()
            End If
            Return _suppliersAdapter
        End Get
    End Property
    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Select, True)> _
    Public Function GetSuppliers() As NorthwindWithSprocs.SuppliersDataTable
        Return Adapter.GetSuppliers()
    End Function
    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Update, True)> _
    Public Function UpdateSupplier(companyName As String, contactName As String, _
        contactTitle As String, supplierID As Integer) As Boolean
        Dim suppliers As NorthwindWithSprocs.SuppliersDataTable = _
            Adapter.GetSupplierBySupplierID(supplierID)
        If suppliers.Count = 0 Then
            ' no matching record found, return false
            Return False
        End If
        Dim supplier As NorthwindWithSprocs.SuppliersRow = suppliers(0)
        supplier.CompanyName = companyName
        If contactName Is Nothing Then
            supplier.SetContactNameNull()
        Else
            supplier.ContactName = contactName
        End If
        If contactTitle Is Nothing Then
            supplier.SetContactTitleNull()
        Else
            supplier.ContactTitle = contactTitle
        End If
        ' Update the product record
        Dim rowsAffected As Integer = Adapter.Update(supplier)
        ' Return true if precisely one row was updated, otherwise false
        Return rowsAffected = 1
    End Function
End Class
```

Like the other BLL classes, `SuppliersBLLWithSprocs` has a `Protected` `Adapter` property that returns an instance of the `SuppliersTableAdapter` class along with two `Public` methods: `GetSuppliers` and `UpdateSupplier`. The `GetSuppliers` method calls and returns the `SuppliersDataTable` returned by the corresponding `GetSupplier` method in the Data Access Layer. The `UpdateSupplier` method retrieves information about the particular supplier being updated via a call to the DAL s `GetSupplierBySupplierID(supplierID)` method. It then updates the `CategoryName`, `ContactName`, and `ContactTitle` properties and commits these changes to the database by calling the Data Access Layer s `Update` method, passing in the modified `SuppliersRow` object.

NOTE

Except for `SupplierID` and `CompanyName`, all columns in the Suppliers table allow `NULL` values. Therefore, if the passed-in `contactName` or `contactTitle` parameters are `Nothing` we need to set the corresponding `ContactName` and `ContactTitle` properties to a `NULL` database value using the `SetContactNameNull` and `SetContactTitleNull` methods, respectively.

Step 7: Working with the Computed Column from the Presentation Layer

With the computed column added to the `Suppliers` table and the DAL and BLL updated accordingly, we are ready to build an ASP.NET page that works with the `FullContactName` computed column. Start by opening the `ComputedColumns.aspx` page in the `AdvancedDAL` folder and drag a GridView from the Toolbox onto the Designer. Set the GridView s `ID` property to `Suppliers` and, from its smart tag, bind it to a new ObjectDataSource named `SuppliersDataSource`. Configure the ObjectDataSource to use the `SuppliersBLLWithSprocs` class we added back in Step 6 and click Next.

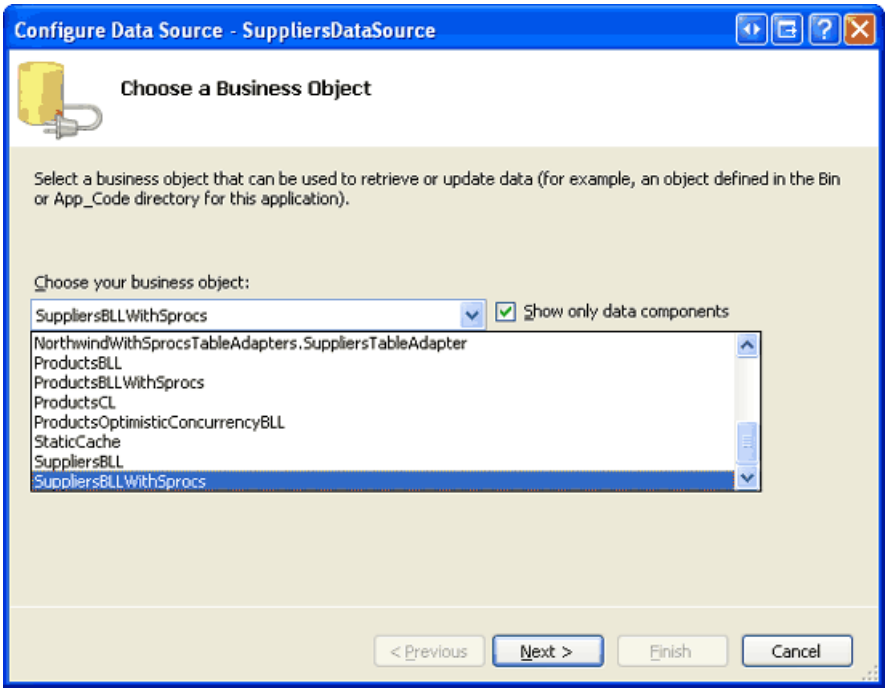


Figure 11: Configure the ObjectDataSource to Use the `SuppliersBLLWithSprocs` Class (Click to view full-size image)

There are only two methods defined in the `SuppliersBLLWithSprocs` class: `GetSuppliers` and `UpdateSupplier`. Ensure that these two methods are specified in the SELECT and UPDATE tabs, respectively, and click Finish to complete the configuration of the ObjectDataSource.

Upon completion of the Data Source Configuration wizard, Visual Studio will add a BoundField for each of the data fields returned. Remove the `SupplierID` BoundField and change the `HeaderText` properties of the `CompanyName`, `ContactName`, `ContactTitle`, and `FullContactName` BoundFields to Company, Contact Name, Title, and Full Contact Name, respectively. From the smart tag, check the Enable Editing checkbox to turn on the GridView s built-in editing capabilities.

In addition to adding BoundFields to the GridView, completion of the Data Source Wizard also causes Visual Studio to set the ObjectDataSource s `OldValuesParameterFormatString` property to original_{0}. Revert this setting back to its default value, {0} .

After making these edits to the GridView and ObjectDataSource, their declarative markup should look similar to the following:

```
<asp:GridView ID="Suppliers" runat="server" AutoGenerateColumns="False"
    DataKeyNames="SupplierID" DataSourceID="SuppliersDataSource">
    <Columns>
        <asp:CommandField ShowEditButton="True" />
        <asp:BoundField DataField="CompanyName"
            HeaderText="Company"
            SortExpression="CompanyName" />
        <asp:BoundField DataField="ContactName"
            HeaderText="Contact Name"
            SortExpression="ContactName" />
        <asp:BoundField DataField="ContactTitle"
            HeaderText="Title"
            SortExpression="ContactTitle" />
        <asp:BoundField DataField="FullContactName"
            HeaderText="Full Contact Name"
            SortExpression="FullContactName"
            ReadOnly="True" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="SuppliersDataSource" runat="server"
    SelectMethod="GetSuppliers" TypeName="SuppliersBLLWithSprocs"
        UpdateMethod="UpdateSupplier">
    <UpdateParameters>
        <asp:Parameter Name="companyName" Type="String" />
        <asp:Parameter Name="contactName" Type="String" />
        <asp:Parameter Name="contactTitle" Type="String" />
        <asp:Parameter Name="supplierID" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

Next, visit this page through a browser. As Figure 12 shows, each supplier is listed in a grid that includes the `FullContactName` column, whose value is simply the concatenation of the other three columns formatted as `ContactName` (`ContactTitle` , `CompanyName`) .

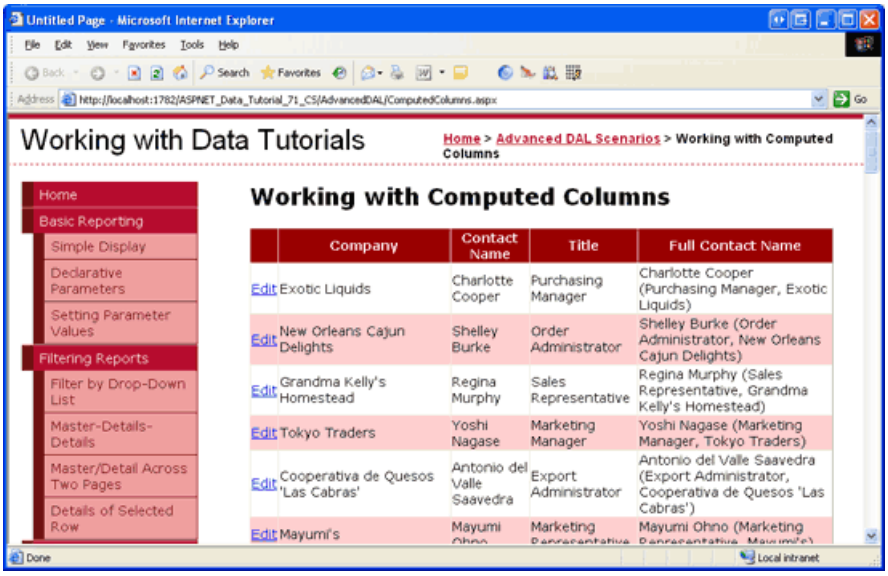


Figure 12: Each Supplier is Listed in the Grid (Click to view full-size image)

Clicking the Edit button for a particular supplier causes a postback and has that row rendered in its editing interface (see Figure 13). The first three columns render in their default editing interface - a TextBox control whose `Text` property is set to the value of the data field. The `FullContactName` column, however, remains as text. When the BoundFields were added to the GridView at the completion of the Data Source Configuration wizard, the `FullContactName` BoundField s `ReadOnly` property was set to `True` because the corresponding `FullContactName` column in the `SuppliersDataTable` has its `ReadOnly` property set to `True` . As noted in Step 4, the `FullContactName` s `ReadOnly` property was set to `True` because the TableAdapter detected that the column was a

computed column.

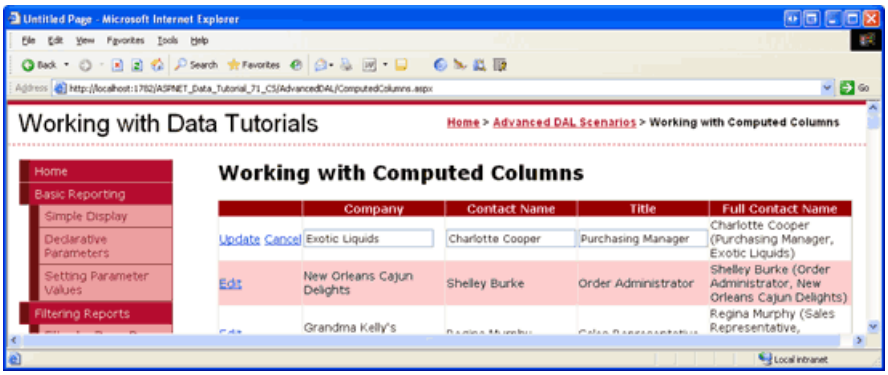


Figure 13: The `FullContactName` Column is Not Editable (Click to view full-size image)

Go ahead and update the value of one or more of the editable columns and click Update. Note how the `FullContactName` s value is automatically updated to reflect the change.

> **NOTE**
>
> The GridView currently uses BoundFields for the editable fields, resulting in the default editing interface. Since the `CompanyName` field is required, it should be converted into a TemplateField that includes a RequiredFieldValidator. I leave this as an exercise for the interested reader. Consult the Adding Validation Controls to the Editing and Inserting Interfaces tutorial for step-by-step instructions on converting a BoundField to a TemplateField and adding validation controls.

Summary

When defining the schema for a table, Microsoft SQL Server allows the inclusion of computed columns. These are columns whose values are calculated from an expression that usually references the values from other columns in the same record. Since the values for computed columns are based on an expression, they are read-only and cannot be assigned a value in an `INSERT` or `UPDATE` statement. This introduces challenges when using a computed column in the main query of a TableAdapter that tries to automatically generate corresponding `INSERT` , `UPDATE` , and `DELETE` statements.

In this tutorial we discussed techniques for circumventing the challenges posed by computed columns. In particular, we used stored procedures in our TableAdapter to overcome the brittleness inherent in TableAdapters that use ad-hoc SQL statements. When having the TableAdapter wizard create new stored procedures, it is important that we have the main query initially omit any computed columns because their presence prevents the data modification stored procedures from being generated. After the TableAdapter has been initially configured, its `SelectCommand` stored procedure can be retooled to include any computed columns.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Hilton Geisenow and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Configuring the Data Access Layer's Connection- and Command-Level Settings (VB)

2/20/2020 • 12 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> The TableAdapters within a Typed DataSet automatically take care of connecting to the database, issuing commands, and populating a DataTable with the results. There are occasions however when we want to take care of these details ourselves, and in this tutorial we learn how to access the database connection- and command-level settings in the TableAdapter.

Introduction

Throughout the tutorial series we have used Typed DataSets to implement the Data Access Layer and business objects of our layered architecture. As discussed in the first tutorial, the Typed DataSet s DataTables serve as repositories of data whereas the TableAdapters act as wrappers to communicate with the database to retrieve and modify the underlying data. The TableAdapters encapsulate the complexity involved in working with the database and saves us from having to write code to connect to the database, issue a command, or populate the results into a DataTable.

There are times, however, when we need to burrow into the depths of the TableAdapter and write code that works directly with the ADO.NET objects. In the Wrapping Database Modifications within a Transaction tutorial, for example, we added methods to the TableAdapter for beginning, committing, and rolling back ADO.NET transactions. These methods used an internal, manually-created `SqlTransaction` object that was assigned to the TableAdapter s `SqlCommand` objects.

In this tutorial we will examine how to access the database connection- and command-level settings in the TableAdapter. In particular, we will add functionality to the `ProductsTableAdapter` that enables access to the underlying connection string and command timeout settings.

Working with Data Using ADO.NET

The Microsoft .NET Framework contains a plethora of classes designed specifically to work with data. These classes, found within the `System.Data` namespace, are referred to as the *ADO.NET* classes. Some of the classes under the ADO.NET umbrella are tied to a particular *data provider*. You can think of a data provider as a communication channel that allows information to flow between the ADO.NET classes and the underlying data store. There are generalized providers, like OleDb and ODBC, as well as providers that are specially designed for a particular database system. For example, while it is possible to connect to a Microsoft SQL Server database using the OleDb provider, the SqlClient provider is much more efficient as it was designed and optimized specifically for SQL Server.

When programmatically accessing data, the following pattern is commonly used:

1. Establish a connection to the database.
2. Issue a command.
3. For `SELECT` queries, work with the resulting records.

There are separate ADO.NET classes for performing each of these steps. To connect to a database using the

SqlClient provider, for example, use the `SqlConnection` class. To issue an `INSERT`, `UPDATE`, `DELETE`, or `SELECT` command to the database, use the `SqlCommand` class.

Except for the Wrapping Database Modifications within a Transaction tutorial, we have not had to write any low-level ADO.NET code ourselves because the TableAdapters auto-generated code includes the functionality needed to connect to the database, issue commands, retrieve data, and populate that data into DataTables. However, there may be times when we need to customize these low-level settings. Over the next few steps we will examine how to tap into the ADO.NET objects used internally by the TableAdapters.

Step 1: Examining with the Connection Property

Each TableAdapter class has a `Connection` property that specifies database connection information. This property s data type and `ConnectionString` value are determined by the selections made in the TableAdapter Configuration wizard. Recall that when we first add a TableAdapter to a Typed DataSet this wizard asks us for the database source (see Figure 1). The drop-down list in this first step includes those databases specified in the configuration file as well as any other databases in the Server Explorer s Data Connections. If the database we want to use does not exist in the drop-down list, a new database connection can be specified by clicking the New Connection button and providing the needed connection information.

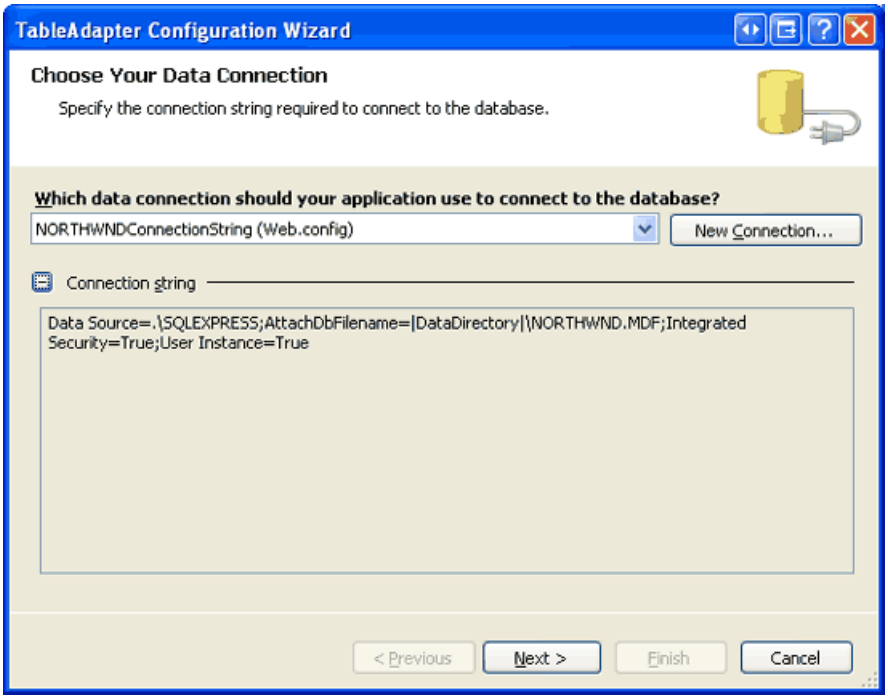


Figure 1: The First Step of the TableAdapter Configuration Wizard (Click to view full-size image)

Let s take a moment to inspect the code for the TableAdapter s `Connection` property. As noted in the Creating a Data Access Layer tutorial, we can view the auto-generated TableAdapter code by going to the Class View window, drilling down to the appropriate class, and then double-clicking the member name.

Navigate to the Class View window by going to the View menu and choosing Class View (or by typing Ctrl+Shift+C). From the top half of the Class View window, drill down to the `NorthwindTableAdapters` namespace and select the `ProductsTableAdapter` class. This will display the `ProductsTableAdapter` s members in the bottom half of the Class View, as shown in Figure 2. Double-click the `Connection` property to see its code.

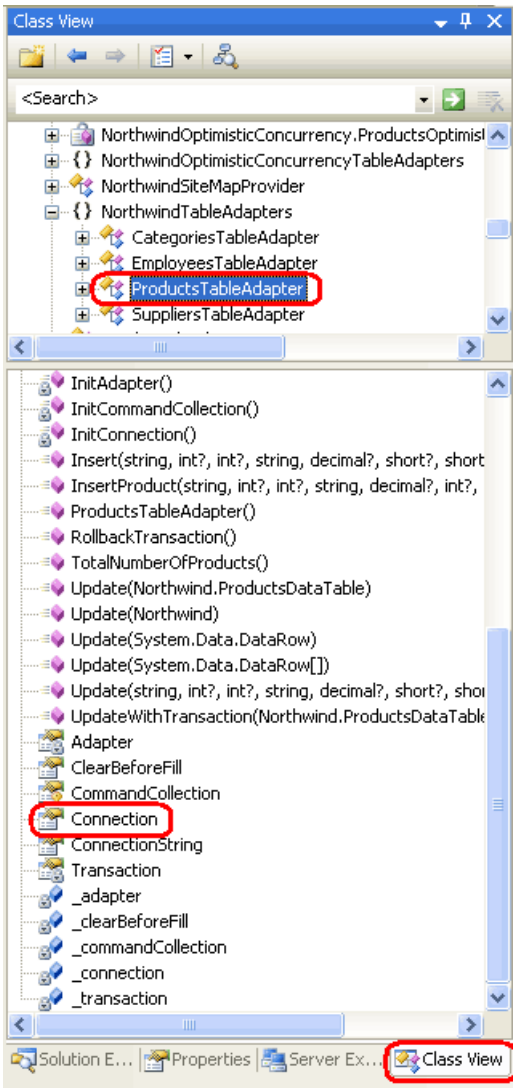


Figure 2: Double-Click the Connection Property in the Class View to View Its Auto-Generated Code

The TableAdapter s `Connection` property and other connection-related code follows:

```
Private _connection As System.Data.SqlClient.SqlConnection
Private Sub InitConnection()
    Me._connection = New System.Data.SqlClient.SqlConnection
    Me._connection.ConnectionString = _
        ConfigurationManager.ConnectionStrings("NORTHWNDConnectionString").ConnectionString
End Sub
Friend Property Connection() As System.Data.SqlClient.SqlConnection
    Get
        If (Me._connection Is Nothing) Then
            Me.InitConnection
        End If
        Return Me._connection
    End Get
    Set
        Me._connection = value
        If (Not (Me.Adapter.InsertCommand) Is Nothing) Then
            Me.Adapter.InsertCommand.Connection = value
        End If
        If (Not (Me.Adapter.DeleteCommand) Is Nothing) Then
            Me.Adapter.DeleteCommand.Connection = value
        End If
        If (Not (Me.Adapter.UpdateCommand) Is Nothing) Then
            Me.Adapter.UpdateCommand.Connection = value
        End If
        Dim i As Integer = 0
        Do While (i < Me.CommandCollection.Length)
            If (Not (Me.CommandCollection(i)) Is Nothing) Then
                CType(Me.CommandCollection(i), _
                    System.Data.SqlClient.SqlCommand).Connection = value
            End If
            i = (i + 1)
        Loop
    End Set
End Property
```

When the TableAdapter class is instantiated, the member variable `_connection` is equal to `Nothing`. When the `Connection` property is accessed, it first checks to see if the `_connection` member variable has been instantiated. If it has not, the `InitConnection` method is invoked, which instantiates `_connection` and sets its `ConnectionString` property to the connection string value specified from the TableAdapter Configuration wizard s first step.

The `Connection` property can also be assigned to a `SqlConnection` object. Doing so associates the new `SqlConnection` object with each of the TableAdapter s `SqlCommand` objects.

Step 2: Exposing Connection-Level Settings

The connection information should remain encapsulated within the TableAdapter and not be accessible to other layers in the application architecture. However, there may be scenarios when the TableAdapter s connection-level information needs to be accessible or customizable for a query, user, or ASP.NET page.

Let s extend the `ProductsTableAdapter` in the `Northwind` DataSet to include a `ConnectionString` property that can be used by the Business Logic Layer to read or change the connection string used by the TableAdapter.

> **NOTE**
>
> A *connection string* is a string that specifies database connection information, such as the provider to use, the location of the database, authentication credentials, and other database-related settings. For a list of connection string patterns used by a variety of data stores and providers, see ConnectionStrings.com.

As discussed in the Creating a Data Access Layer tutorial, the Typed DataSet s auto-generated classes can be extended through the use of partial classes. First, create a new subfolder in the project named

`ConnectionAndCommandSettings` underneath the `~/App_Code/DAL` folder.

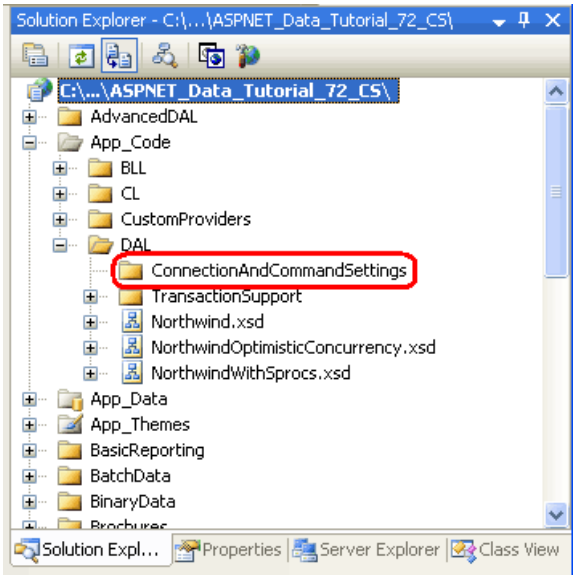


Figure 3: Add a Subfolder Named `ConnectionAndCommandSettings`

Add a new class file named `ProductsTableAdapter.ConnectionAndCommandSettings.vb` and enter the following code:

```
Namespace NorthwindTableAdapters
    Partial Public Class ProductsTableAdapter
        Public Property ConnectionString() As String
            Get
                Return Me.Connection.ConnectionString
            End Get
            Set(ByVal value As String)
                Me.Connection.ConnectionString = value
            End Set
        End Property
    End Class
End Namespace
```

This partial class adds a `Public` property named `ConnectionString` to the `ProductsTableAdapter` class that allows any layer to read or update the connection string for the TableAdapter s underlying connection.

With this partial class created (and saved), open the `ProductsBLL` class. Go to one of the existing methods and type in `Adapter` and then hit the period key to bring up IntelliSense. You should see the new `ConnectionString` property available in IntelliSense, meaning that you can programmatically read or adjust this value from the BLL.

Exposing the Entire Connection Object

This partial class exposes just one property of the underlying connection object: `ConnectionString`. If you want to make the entire connection object available beyond the confines of the TableAdapter, you can alternatively change the `Connection` property s protection level. The auto-generated code we examined in Step 1 showed that the TableAdapter s `Connection` property is marked as `Friend`, meaning that it can only be accessed by classes in the same assembly. This can be changed, however, via the TableAdapter s `ConnectionModifier` property.

Open the `Northwind` DataSet, click on the `ProductsTableAdapter` in the Designer, and navigate to the Properties window. There you will see the `ConnectionModifier` set to its default value, `Assembly`. To make the `Connection` property available outside of the Typed DataSet s assembly, change the `ConnectionModifier` property to `Public`.

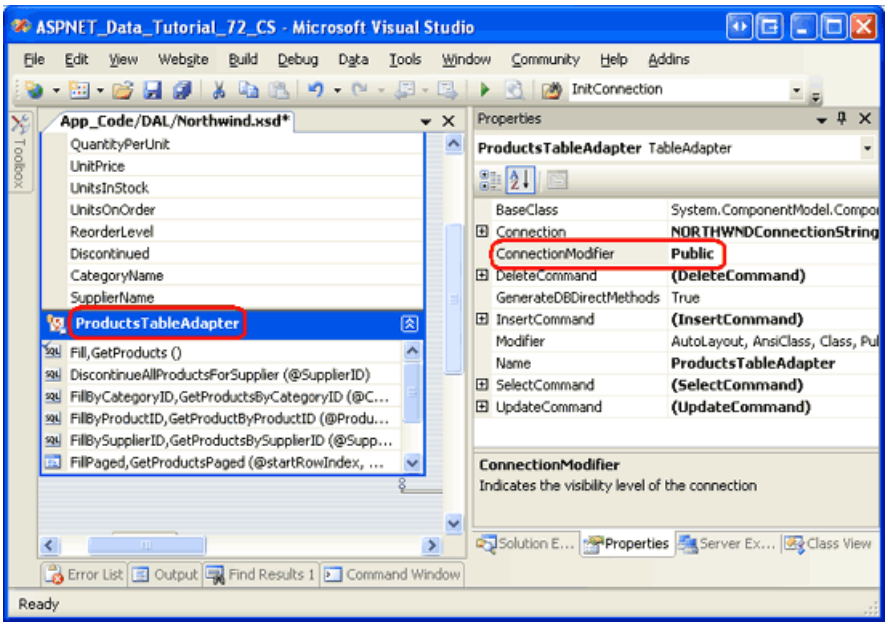


Figure 4: The `Connection` Property s Accessibility Level Can Be Configured via the `ConnectionModifier` Property (Click to view full-size image)

Save the DataSet and then return to the `ProductsBLL` class. As before, go to one of the existing methods and type in `Adapter` and then hit the period key to bring up IntelliSense. The list should include a `Connection` property, meaning that you can now programmatically read or assign any connection-level settings from the BLL.

Step 3: Examining the Command-Related Properties

A TableAdapter consists of a main query that, by default, has auto-generated `INSERT`, `UPDATE`, and `DELETE` statements. This main query s `INSERT`, `UPDATE`, and `DELETE` statements are implemented in the TableAdapter s code as an ADO.NET data adapter object via the `Adapter` property. Like with its `Connection` property, the `Adapter` property s data type is determined by the data provider used. Since these tutorials use the SqlClient provider, the `Adapter` property is of type `SqlDataAdapter`.

The TableAdapter s `Adapter` property has three properties of type `SqlCommand` that it uses to issue the `INSERT`, `UPDATE`, and `DELETE` statements:

- `InsertCommand`
- `UpdateCommand`
- `DeleteCommand`

A `SqlCommand` object is responsible for sending a particular query to the database and has properties like: `CommandText`, which contains the ad-hoc SQL statement or stored procedure to execute; and `Parameters`, which is a collection of `SqlParameter` objects. As we saw back in the Creating a Data Access Layer tutorial, these command objects can be customized through the Properties window.

In addition to its main query, the TableAdapter can include a variable number of methods that, when invoked, dispatch a specified command to the database. The main query s command object and the command objects for all additional methods are stored in the TableAdapter s `CommandCollection` property.

Let s take a moment to look at the code generated by the `ProductsTableAdapter` in the `Northwind` DataSet for these two properties and their supporting member variables and helper methods:

```
Private WithEvents _adapter As System.Data.SqlClient.SqlDataAdapter
Private Sub InitAdapter()
    Me._adapter = New System.Data.SqlClient.SqlDataAdapter

    ... Code that creates the InsertCommand, UpdateCommand, ...
    ... and DeleteCommand instances - omitted for brevity ...
End Sub
Private ReadOnly Property Adapter() As System.Data.SqlClient.SqlDataAdapter
    Get
        If (Me._adapter Is Nothing) Then
            Me.InitAdapter
        End If
        Return Me._adapter
    End Get
End Property
Private _commandCollection() As System.Data.SqlClient.SqlCommand
Private Sub InitCommandCollection()
    Me._commandCollection = New System.Data.SqlClient.SqlCommand(8) {}
    ... Code that creates the command objects for the main query and the ...
    ... ProductsTableAdapter►s other eight methods - omitted for brevity ...
End Sub
Protected ReadOnly Property CommandCollection() As System.Data.SqlClient.SqlCommand()
    Get
        If (Me._commandCollection Is Nothing) Then
            Me.InitCommandCollection
        End If
        Return Me._commandCollection
    End Get
End Property
```

The code for the `Adapter` and `CommandCollection` properties closely mimics that of the `Connection` property. There are member variables that hold the objects used by the properties. The properties `Get` accessors start by checking to see if the corresponding member variable is `Nothing`. If so, an initialization method is called which creates an instance of the member variable and assigns the core command-related properties.

Step 4: Exposing Command-Level Settings

Ideally, the command-level information should remain encapsulated within the Data Access Layer. Should this information be needed in other layers of the architecture, however, it can be exposed through a partial class, just like with the connection-level settings.

Since the TableAdapter only has a single `Connection` property, the code for exposing connection-level settings is fairly straightforward. Things are a bit more complicated when modifying command-level settings because the TableAdapter can have multiple command objects - an `InsertCommand`, `UpdateCommand`, and `DeleteCommand`, along with a variable number of command objects in the `CommandCollection` property. When updating command-level settings, these settings will need to be propagated to all of the command objects.

For example, imagine that there were certain queries in the TableAdapter that took an extraordinary long time to execute. When using the TableAdapter to execute one of those queries, we might want to increase the command object s `CommandTimeout` property. This property specifies the number of seconds to wait for the command to execute and defaults to 30.

To allow the `CommandTimeout` property to be adjusted by the BLL, add the following `Public` method to the `ProductsDataTable` using the partial class file created in Step 2 (`ProductsTableAdapter.ConnectionAndCommandSettings.vb`):

```
Public Sub SetCommandTimeout(ByVal timeout As Integer)
    If Me.Adapter.InsertCommand IsNot Nothing Then
        Me.Adapter.InsertCommand.CommandTimeout = timeout
    End If
    If Me.Adapter.DeleteCommand IsNot Nothing Then
        Me.Adapter.DeleteCommand.CommandTimeout = timeout
    End If
    If Me.Adapter.UpdateCommand IsNot Nothing Then
        Me.Adapter.UpdateCommand.CommandTimeout = timeout
    End If
    For i As Integer = 0 To Me.CommandCollection.Length - 1
        If Me.CommandCollection(i) IsNot Nothing Then
            Me.CommandCollection(i).CommandTimeout = timeout
        End If
    Next
End Sub
```

This method could be invoked from the BLL or Presentation Layer to set the command timeout for all commands issues by that TableAdapter instance.

NOTE

The `Adapter` and `CommandCollection` properties are marked as `Private`, meaning they can only be accessed from code within the TableAdapter. Unlike the `Connection` property, these access modifiers are not configurable. Therefore, if you need to expose command-level properties to other layers in the architecture you must use the partial class approach discussed above to provide a `Public` method or property that reads or writes to the `Private` command objects.

Summary

The TableAdapters within a Typed DataSet serve to encapsulate data access details and complexity. Using TableAdapters, we do not have to worry about writing ADO.NET code to connect to the database, issue a command, or populate the results into a DataTable. It is all handled automatically for us.

However, there may be times when we need to customize the low-level ADO.NET specifics, such as changing the connection string or the default connection or command timeout values. The TableAdapter has auto-generated `Connection`, `Adapter`, and `CommandCollection` properties, but these are either `Friend` or `Private`, by default. This internal information can be exposed by extending the TableAdapter using partial classes to include `Public` methods or properties. Alternatively, the TableAdapter s `Connection` property access modifier can be configured through the TableAdapter s `ConnectionModifier` property.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Burnadette Leigh, S ren Jacob Lauritsen, Teresa Murphy, and Hilton Geisenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Protecting Connection Strings and Other Configuration Information (VB)

2/20/2020 • 16 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

An ASP.NET application typically stores configuration information in a Web.config file. Some of this information is sensitive and warrants protection. By default this file will not be served to a Web site visitor, but an administrator or a hacker may gain access to the Web server's file system and view the contents of the file. In this tutorial we learn that ASP.NET 2.0 allows us to protect sensitive information by encrypting sections of the Web.config file.

Introduction

Configuration information for ASP.NET applications is commonly stored in an XML file named `Web.config`. Over the course of these tutorials we have updated the `Web.config` a handful of times. When creating the `Northwind` Typed DataSet in the first tutorial, for example, connection string information was automatically added to `Web.config` in the `<connectionStrings>` section. Later, in the Master Pages and Site Navigation tutorial, we manually updated `Web.config`, adding a `<pages>` element indicating that all of the ASP.NET pages in our project should use the `DataWebControls` Theme.

Since `Web.config` may contain sensitive data such as connection strings, it is important that the contents of `Web.config` be kept safe and hidden from unauthorized viewers. By default, any HTTP request to a file with the `.config` extension is handled by the ASP.NET engine, which returns the *This type of page is not served* message shown in Figure 1. This means that visitors cannot view your `Web.config` file s contents by simply entering http://www.YourServer.com/Web.config into their browser s Address bar.

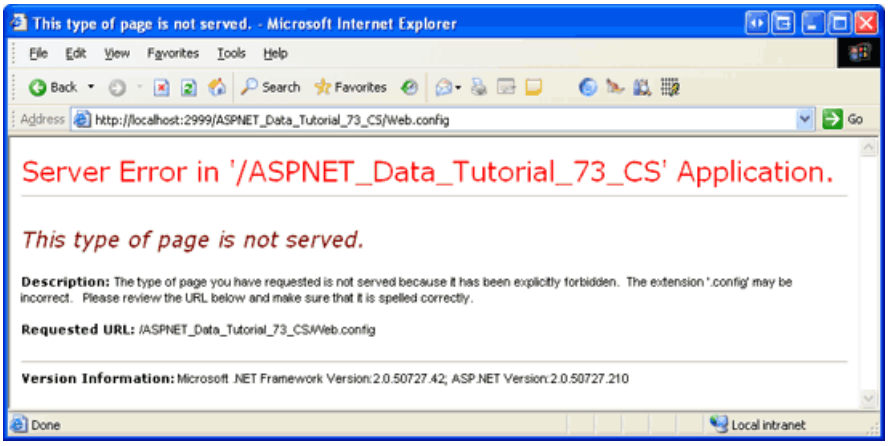


Figure 1: Visiting `Web.config` Through a Browser Returns a This type of page is not served Message (Click to view full-size image)

But what if an attacker is able to find some other exploit that allows her to view your `Web.config` file s contents? What could an attacker do with this information, and what steps can be taken to further protect the sensitive information within `Web.config`? Fortunately, most sections in `Web.config` do not contain sensitive information. What harm can an attacker perpetrate if they know the name of the default Theme used by your ASP.NET pages?

Certain `Web.config` sections, however, contain sensitive information that may include connection strings, user

names, passwords, server names, encryption keys, and so forth. This information is typically found in the following `Web.config` sections:

- `<appSettings>`
- `<connectionStrings>`
- `<identity>`
- `<sessionState>`

In this tutorial we will look at techniques for protecting such sensitive configuration information. As we will see, the .NET Framework version 2.0 includes a protected configurations system that makes programmatically encrypting and decrypting selected configuration sections a breeze.

> **NOTE**
> This tutorial concludes with a look at Microsoft s recommendations for connecting to a database from an ASP.NET application. In addition to encrypting your connection strings, you can help harden your system by ensuring that you are connecting to the database in a secure fashion.

Step 1: Exploring ASP.NET 2.0 s Protected Configuration Options

ASP.NET 2.0 includes a protected configuration system for encrypting and decrypting configuration information. This includes methods in the .NET Framework that can be used to programmatically encrypt or decrypt configuration information. The protected configuration system uses the provider model, which allows developers to choose what cryptographic implementation is used.

The .NET Framework ships with two protected configuration providers:

- `RSAProtectedConfigurationProvider` - uses the asymmetric RSA algorithm for encryption and decryption.
- `DPAPIProtectedConfigurationProvider` - uses the Windows Data Protection API (DPAPI) for encryption and decryption.

Since the protected configuration system implements the provider design pattern, it is possible to create your own protected configuration provider and plug it into your application. See Implementing a Protected Configuration Provider for more information on this process.

The RSA and DPAPI providers use keys for their encryption and decryption routines, and these keys can be stored at the machine- or user-level. Machine-level keys are ideal for scenarios where the web application runs on its own dedicated server or if there are multiple applications on a server that need to share encrypted information. User-level keys are a more secure option in shared hosting environments where other applications on the same server should not be able to decrypt your application s protected configuration sections.

In this tutorial our examples will use the DPAPI provider and machine-level keys. Specifically, we will look at encrypting the `<connectionStrings>` section in `Web.config`, although the protected configuration system can be used to encrypt most any `Web.config` section. For information on using user-level keys or using the RSA provider, consult the resources in the Further Readings section at the end of this tutorial.

NOTE

The `RSAProtectedConfigurationProvider` and `DPAPIProtectedConfigurationProvider` providers are registered in the `machine.config` file with the provider names `RsaProtectedConfigurationProvider` and `DataProtectionConfigurationProvider`, respectively. When encrypting or decrypting configuration information we will need to supply the appropriate provider name (`RsaProtectedConfigurationProvider` or `DataProtectionConfigurationProvider`) rather than the actual type name (`RSAProtectedConfigurationProvider` and `DPAPIProtectedConfigurationProvider`). You can find the `machine.config` file in the `$WINDOWS$\Microsoft.NET\Framework\version\CONFIG` folder.

Step 2: Programmatically Encrypting and Decrypting Configuration Sections

With a few lines of code we can encrypt or decrypt a particular configuration section using a specified provider. The code, as we will see shortly, simply needs to programmatically reference the appropriate configuration section, call its `ProtectSection` or `UnprotectSection` method, and then call the `Save` method to persist the changes. Moreover, the .NET Framework includes a helpful command line utility that can encrypt and decrypt configuration information. We will explore this command line utility in Step 3.

To illustrate programmatically protecting configuration information, let s create an ASP.NET page that includes buttons for encrypting and decrypting the `<connectionStrings>` section in `Web.config`.

Start by opening the `EncryptingConfigSections.aspx` page in the `AdvancedDAL` folder. Drag a TextBox control from the Toolbox onto the Designer, setting its `ID` property to `WebConfigContents`, its `TextMode` property to `MultiLine`, and its `Width` and `Rows` properties to 95% and 15, respectively. This TextBox control will display the contents of `Web.config` allowing us to quickly see if the contents are encrypted or not. Of course, in a real application you would never want to display the contents of `Web.config`.

Beneath the TextBox, add two Button controls named `EncryptConnStrings` and `DecryptConnStrings`. Set their Text properties to Encrypt Connection Strings and Decrypt Connection Strings .

At this point your screen should look similar to Figure 2.

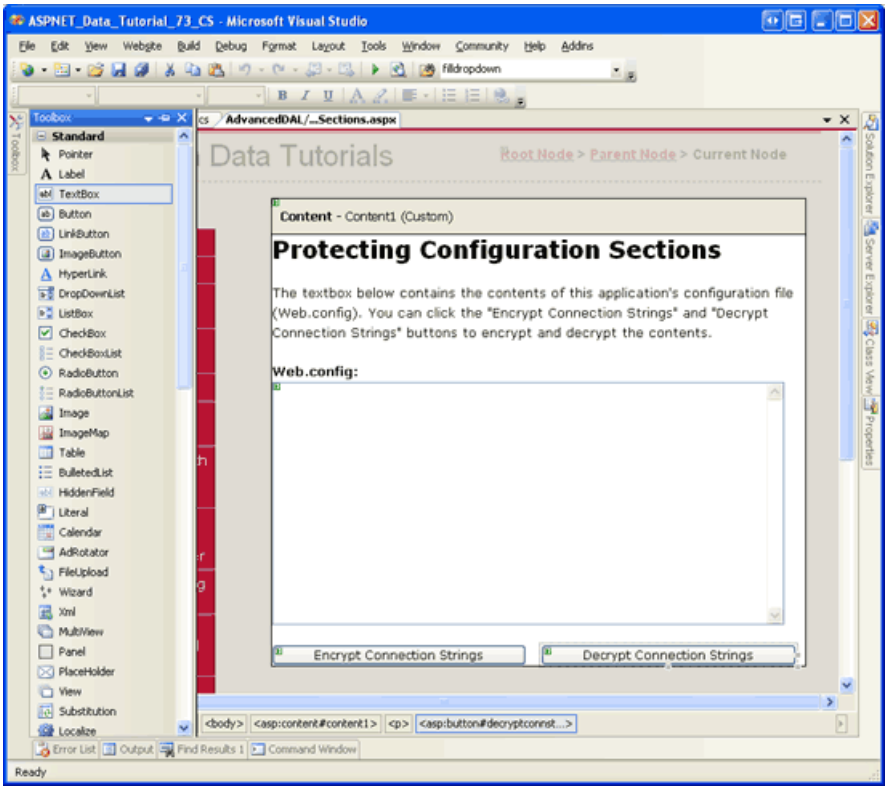


Figure 2: Add a TextBox and Two Button Web Controls to the Page (Click to view full-size image)

Next, we need to write code that loads and displays the contents of `Web.config` in the `WebConfigContents` TextBox when the page is first loaded. Add the following code to the page s code-behind class. This code adds a method named `DisplayWebConfig` and calls it from the `Page_Load` event handler when `Page.IsPostBack` is `False`:

```
Protected Sub Page_Load(sender As Object, e As EventArgs) Handles Me.Load
    'On the first page visit, call DisplayWebConfig method
    If Not Page.IsPostBack Then
        DisplayWebConfig()
    End If
End Sub
Private Sub DisplayWebConfig()
    'Reads in the contents of Web.config and displays them in the TextBox
    Dim webConfigStream As StreamReader = _
        File.OpenText(Path.Combine(Request.PhysicalApplicationPath, "Web.config"))
    Dim configContents As String = webConfigStream.ReadToEnd()
    webConfigStream.Close()
    WebConfigContents.Text = configContents
End Sub
```

The `DisplayWebConfig` method uses the `File` class to open the application s `Web.config` file, the `StreamReader` class to read its contents into a string, and the `Path` class to generate the physical path to the `Web.config` file. These three classes are all found in the `System.IO` namespace. Consequently, you will need to add a `Imports``System.IO` statement to the top of the code-behind class or, alternatively, prefix these class names with `System.IO.`

Next, we need to add event handlers for the two Button controls `Click` events and add the necessary code to encrypt and decrypt the `<connectionStrings>` section using a machine-level key with the DPAPI provider. From the Designer, double-click each of the Buttons to add a `Click` event handler in the code-behind class and then add the following code:

```
Protected Sub EncryptConnStrings_Click(sender As Object, e As EventArgs) _
    Handles EncryptConnStrings.Click
    'Get configuration information about Web.config
    Dim config As Configuration = _
        WebConfigurationManager.OpenWebConfiguration(Request.ApplicationPath)
    ' Let's work with the <connectionStrings> section
    Dim connectionStrings As ConfigurationSection = _
        config.GetSection("connectionStrings")
    If connectionStrings IsNot Nothing Then
        ' Only encrypt the section if it is not already protected
        If Not connectionStrings.SectionInformation.IsProtected Then
            ' Encrypt the <connectionStrings> section using the
            ' DataProtectionConfigurationProvider provider
            connectionStrings.SectionInformation.ProtectSection( _
                "DataProtectionConfigurationProvider")
            config.Save()
            ' Refresh the Web.config display
            DisplayWebConfig()
        End If
    End If
End Sub
Protected Sub DecryptConnStrings_Click(sender As Object, e As EventArgs) _
    Handles DecryptConnStrings.Click
    ' Get configuration information about Web.config
    Dim config As Configuration = _
        WebConfigurationManager.OpenWebConfiguration(Request.ApplicationPath)
    ' Let's work with the <connectionStrings> section
    Dim connectionStrings As ConfigurationSection = _
        config.GetSection("connectionStrings")
    If connectionStrings IsNot Nothing Then
        ' Only decrypt the section if it is protected
        If connectionStrings.SectionInformation.IsProtected Then
            ' Decrypt the <connectionStrings> section
            connectionStrings.SectionInformation.UnprotectSection()
            config.Save()
            ' Refresh the Web.config display
            DisplayWebConfig()
        End If
    End If
End Sub
```

The code used in the two event handlers is nearly identical. They both start by getting information about the current application s `Web.config` file via the `WebConfigurationManager` class s `OpenWebConfiguration` method. This method returns the web configuration file for the specified virtual path. Next, the `Web.config` file s `<connectionStrings>` section is accessed via the `Configuration` class s `GetSection(sectionName)` method, which returns a `ConfigurationSection` object.

The `ConfigurationSection` object includes a `SectionInformation` property that provides additional information and functionality regarding the configuration section. As the code above shows, we can determine whether the configuration section is encrypted by checking the `SectionInformation` property s `IsProtected` property. Moreover, the section can be encrypted or decrypted via the `SectionInformation` property s `ProtectSection(provider)` and `UnprotectSection` methods.

The `ProtectSection(provider)` method accepts as input a string specifying the name of the protected configuration provider to use when encrypting. In the `EncryptConnString` Button s event handler we pass DataProtectionConfigurationProvider into the `ProtectSection(provider)` method so that the DPAPI provider is used. The `UnprotectSection` method can determine the provider that was used to encrypt the configuration section and therefore does not require any input parameters.

After calling the `ProtectSection(provider)` or `UnprotectSection` method, you must call the `Configuration` object s `Save` method to persist the changes. Once the configuration information has been encrypted or decrypted and

the changes saved, we call `DisplayWebConfig` to load the updated `Web.config` contents into the TextBox control.

Once you have entered the above code, test it by visiting the `EncryptingConfigSections.aspx` page through a browser. You should initially see a page that lists the contents of `Web.config` with the `<connectionStrings>` section displayed in plain-text (see Figure 3).

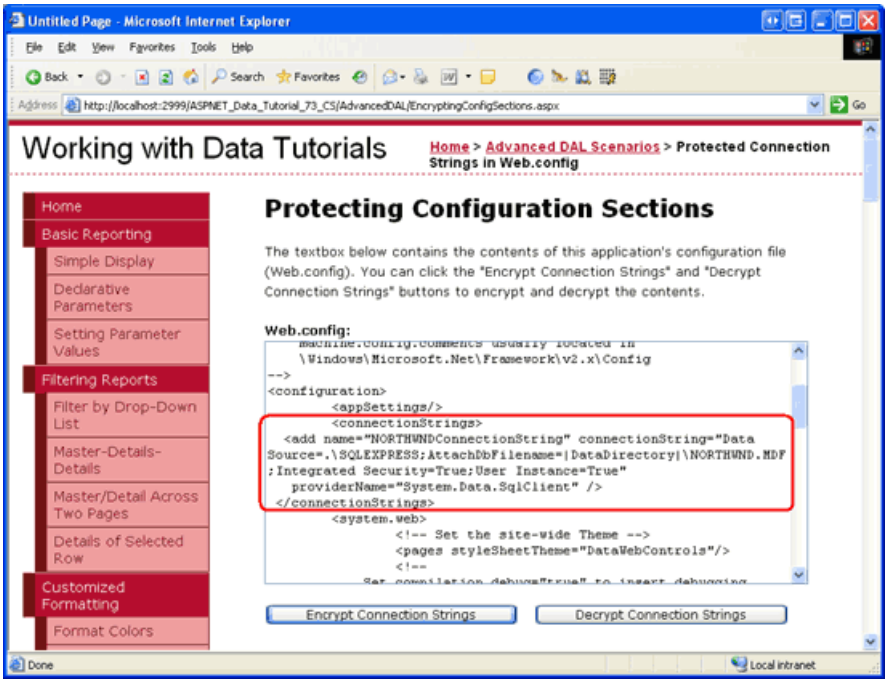


Figure 3: Add a TextBox and Two Button Web Controls to the Page (Click to view full-size image)

Now click the Encrypt Connection Strings button. If request validation is enabled, the markup posted back from the `WebConfigContents` TextBox will produce an `HttpRequestValidationException`, which displays the message, A potentially dangerous `Request.Form` value was detected from the client. Request validation, which is enabled by default in ASP.NET 2.0, prohibits postbacks that include un-encoded HTML and is designed to help prevent script-injection attacks. This check can be disabled at the page- or application-level. To turn it off for this page, set the `ValidateRequest` setting to `False` in the `@Page` directive. The `@Page` directive is found at the top of the page s declarative markup.

```
<%@ Page ValidateRequest="False" ... %>
```

For more information on request validation, its purpose, how to disable it at the page- and application-level, as well as how to HTML encode markup, see Request Validation - Preventing Script Attacks.

After disabling request validation for the page, try clicking the Encrypt Connection Strings button again. On postback, the configuration file will be accessed and its `<connectionStrings>` section encrypted using the DPAPI provider. The TextBox is then updated to display the new `Web.config` content. As Figure 4 shows, the `<connectionStrings>` information is now encrypted.

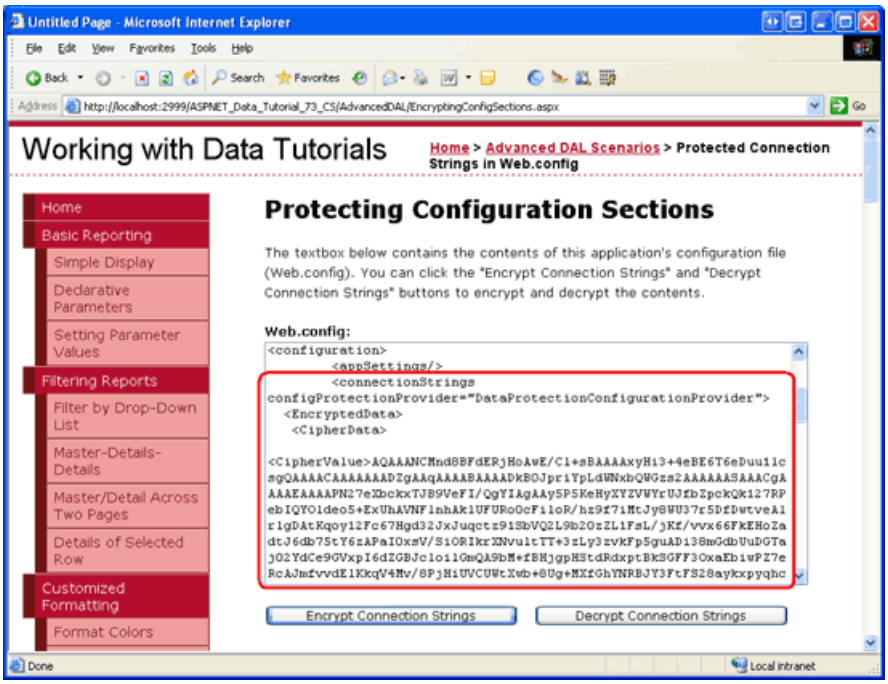


Figure 4: Clicking the Encrypt Connection Strings Button Encrypts the `<connectionString>` Section (Click to view full-size image)

The encrypted `<connectionStrings>` section generated on my computer follows, although some of the content in the `<CipherData>` element has been removed for brevity:

```
<connectionStrings
    configProtectionProvider="DataProtectionConfigurationProvider">
  <EncryptedData>
    <CipherData>
      <CipherValue>AQAAANCMnd8BFdERjHoAwE/...zChw==</CipherValue>
    </CipherData>
  </EncryptedData>
</connectionStrings>
```

NOTE

The `<connectionStrings>` element specifies the provider used to perform the encryption (`DataProtectionConfigurationProvider`). This information is used by the `UnprotectSection` method when the Decrypt Connection Strings button is clicked.

When the connection string information is accessed from `Web.config` - either by code we write, from a SqlDataSource control, or the auto-generated code from the TableAdapters in our Typed DataSets - it is automatically decrypted. In short, we do not need to add any extra code or logic to decrypt the encrypted `<connectionString>` section. To demonstrate this, visit one of the earlier tutorials at this time, such as the Simple Display tutorial from the Basic Reporting section (`~/BasicReporting/SimpleDisplay.aspx`). As Figure 5 shows, the tutorial works exactly as we would expect it, indicating that the encrypted connection string information is being automatically decrypted by the ASP.NET page.

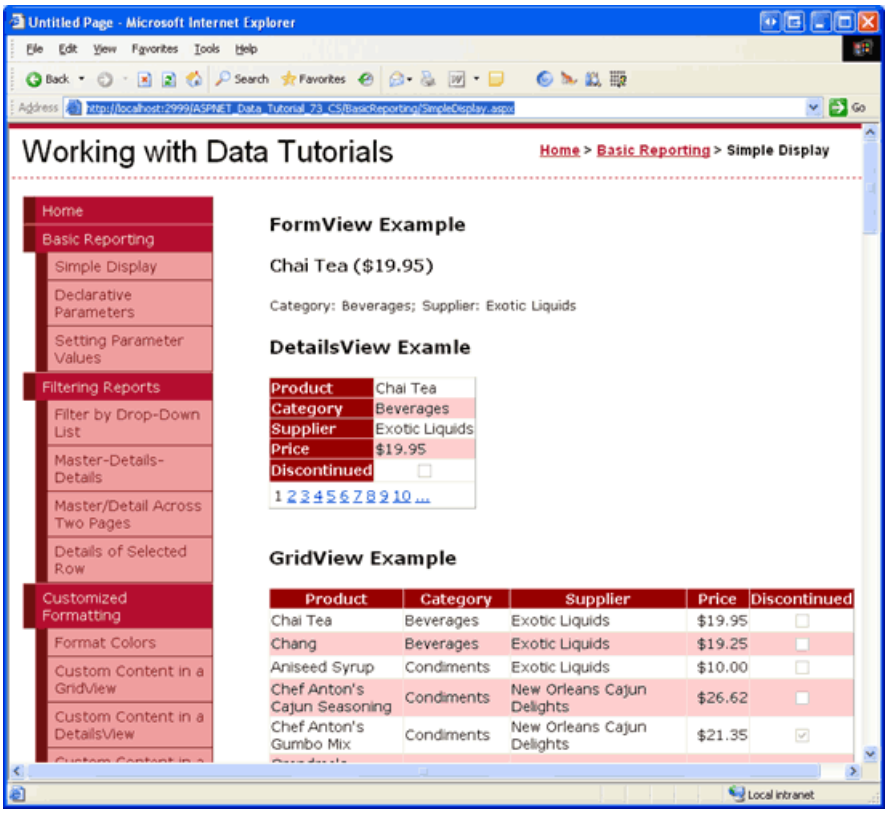


Figure 5: The Data Access Layer Automatically Decrypts the Connection String Information (Click to view full-size image)

To revert the `<connectionStrings>` section back to its plain-text representation, click the Decrypt Connection Strings button. On postback you should see the connection strings in `Web.config` in plain-text. At this point your screen should look like it did when first visiting this page (see in Figure 3).

Step 3: Encrypting Configuration Sections Using `aspnet_regiis.exe`

The .NET Framework includes a variety of command line tools in the `$WINDOWS$\Microsoft.NET\Framework\version\` folder. In the Using SQL Cache Dependencies tutorial, for instance, we looked at using the `aspnet_regsql.exe` command line tool to add the infrastructure necessary for SQL cache dependencies. Another useful command line tool in this folder is the ASP.NET IIS Registration tool (`aspnet_regiis.exe`). As its name implies, the ASP.NET IIS Registration tool is primarily used to register an ASP.NET 2.0 application with Microsoft s professional-grade Web server, IIS. In addition to its IIS-related features, the ASP.NET IIS Registration tool can also be used to encrypt or decrypt specified configuration sections in `Web.config`.

The following statement shows the general syntax used to encrypt a configuration section with the `aspnet_regiis.exe` command line tool:

```
aspnet_regiis.exe -pef section physical_directory -prov provider
```

section is the configuration section to encrypt (like connectionStrings), the *physical_directory* is the full, physical path to the web application s root directory, and *provider* is the name of the protected configuration provider to use (such as DataProtectionConfigurationProvider). Alternatively, if the web application is registered in IIS you can enter the virtual path instead of the physical path using the following syntax:

```
aspnet_regiis.exe -pe section -app virtual_directory -prov provider
```

The following `aspnet_regiis.exe` example encrypts the `<connectionStrings>` section using the DPAPI provider with a machine-level key:

```
aspnet_regiis.exe -pef
"connectionStrings" "C:\Websites\ASPNET_Data_Tutorial_73_VB"
-prov "DataProtectionConfigurationProvider"
```

Similarly, the `aspnet_regiis.exe` command line tool can be used to decrypt configuration sections. Instead of using the `-pef` switch, use `-pdf` (or instead of `-pe`, use `-pd`). Also, note that the provider name is not necessary when decrypting.

```
aspnet_regiis.exe -pdf section physical_directory
  -- or --
aspnet_regiis.exe -pd section -app virtual_directory
```

NOTE

Since we are using the DPAPI provider, which uses keys specific to the computer, you must run `aspnet_regiis.exe` from the same machine from which the web pages are being served. For example, if you run this command line program from your local development machine and then upload the encrypted Web.config file to the production server, the production server will not be able to decrypt the connection string information since it was encrypted using keys specific to your development machine. The RSA provider does not have this limitation as it is possible to export the RSA keys to another machine.

Understanding Database Authentication Options

Before any application can issue `SELECT`, `INSERT`, `UPDATE`, or `DELETE` queries to a Microsoft SQL Server database, the database first must identify the requestor. This process is known as *authentication* and SQL Server provides two methods of authentication:

- **Windows Authentication** - the process under which the application is running is used to communicate with the database. When running an ASP.NET application through Visual Studio 2005 s ASP.NET Development Server, the ASP.NET application assumes the identity of the currently logged on user. For ASP.NET applications on Microsoft Internet Information Server (IIS), ASP.NET applications usually assume the identity of `domainName``\MachineName` or `domainName``\NETWORK SERVICE`, although this can be customized.
- **SQL Authentication** - a user ID and password values are supplied as credentials for authentication. With SQL authentication, the user ID and password are provided in the connection string.

Windows authentication is preferred over SQL authentication because it is more secure. With Windows authentication the connection string is free from a username and password and if the web server and database server reside on two different machines, the credentials are not sent over the network in plain-text. With SQL authentication, however, the authentication credentials are hard-coded in the connection string and are transmitted from the web server to the database server in plain-text.

These tutorials have used Windows authentication. You can tell what authentication mode is being used by inspecting the connection string. The connection string in `Web.config` for our tutorials has been:

```
Data Source=.\SQLEXPRESS; AttachDbFilename=|DataDirectory|\NORTHWND.MDF; Integrated Security=True; User
Instance=True
```

The Integrated Security=True and lack of a username and password indicate that Windows authentication is being used. In some connection strings the term Trusted Connection=Yes or Integrated Security=SSPI is used instead of Integrated Security=True, but all three indicate the use of Windows authentication.

The following example shows a connection string that uses SQL authentication. Note the credentials embedded within the connection string:

```
Server=serverName; Database=Northwind; uid=userID; pwd=password
```

Imagine that an attacker is able to view your application s `Web.config` file. If you use SQL authentication to connect to a database that is accessible over the Internet, the attacker can use this connection string to connect to your database through SQL Management Studio or from ASP.NET pages on their own website. To help mitigate this threat, encrypt the connection string information in `Web.config` using the protected configuration system.

> **NOTE**
> For more information on the different types of authentication available in SQL Server, see Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication. For further connection string examples illustrating the differences between Windows and SQL authentication syntax, refer to ConnectionStrings.com.

Summary

By default, files with a `.config` extension in an ASP.NET application cannot be accessed through a browser. These types of files are not returned because they may contain sensitive information, such as database connection strings, usernames and passwords, and so on. The protected configuration system in .NET 2.0 helps further protect sensitive information by allowing specified configuration sections to be encrypted. There are two built-in protected configuration providers: one that uses the RSA algorithm and one that uses the Windows Data Protection API (DPAPI).

In this tutorial we looked at how to encrypt and decrypt configuration settings using the DPAPI provider. This can be accomplished both programmatically, as we saw in Step 2, as well as through the `aspnet_regiis.exe` command line tool, which was covered in Step 3. For more information on using user-level keys or using the RSA provider instead, see the resources in the Further Reading section.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Building Secure ASP.NET Application: Authentication, Authorization, and Secure Communication
- Encrypting Configuration Information in ASP.NET 2.0 Applications
- Encrypting `Web.config` Values in ASP.NET 2.0
- How To: Encrypt Configuration Sections in ASP.NET 2.0 Using DPAPI
- How To: Encrypt Configuration Sections in ASP.NET 2.0 Using RSA
- The Configuration API in .NET 2.0
- Windows Data Protection

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Teresa Murphy and Randy Schmidt. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Debugging Stored Procedures (VB)

2/20/2020 • 14 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Visual Studio Professional and Team System editions allow you to set breakpoints and step in to stored procedures within SQL Server, making debugging stored procedures as easy as debugging application code. This tutorial demonstrates direct database debugging and application debugging of stored procedures.

Introduction

Visual Studio provides a rich debugging experience. With a few keystrokes or clicks of the mouse, it s possible to use breakpoints to stop execution of a program and examine its state and control flow. Along with debugging application code, Visual Studio offers support for debugging stored procedures from within SQL Server. Just like breakpoints can be set within the code of an ASP.NET code-behind class or Business Logic Layer class, so too can they be placed within stored procedures.

In this tutorial we will look at stepping into stored procedures from the Server Explorer within Visual Studio as well as how to set breakpoints that are hit when the stored procedure is called from the running ASP.NET application.

> **NOTE**
>
> Unfortunately, stored procedures can only be stepped into and debugged through the Professional and Team Systems versions of Visual Studio. If you are using Visual Web Developer or the standard version of Visual Studio, you are welcome to read along as we walk through the steps necessary to debug stored procedures, but you will not be able to replicate these steps on your machine.

SQL Server Debugging Concepts

Microsoft SQL Server 2005 was designed to provide integration with the Common Language Runtime (CLR), which is the runtime used by all .NET assemblies. Consequently, SQL Server 2005 supports managed database objects. That is, you can create database objects like stored procedures and User-Defined Functions (UDFs) as methods in a Visual Basic class. This enables these stored procedures and UDFs to utilize functionality in the .NET Framework and from your own custom classes. Of course, SQL Server 2005 also provides support for T-SQL database objects.

SQL Server 2005 offers debugging support for both T-SQL and managed database objects. However, these objects can only be debugged through Visual Studio 2005 Professional and Team Systems editions. In this tutorial we will examine debugging T-SQL database objects. The subsequent tutorial looks at debugging managed database objects.

The Overview of T-SQL and CLR Debugging in SQL Server 2005 blog entry from the SQL Server 2005 CLR Integration team highlights the three ways to debug SQL Server 2005 objects from Visual Studio:

- **Direct Database Debugging (DDD)** - from Server Explorer we can step into any T-SQL database object, such as stored procedures and UDFs. We will examine DDD in Step 1.
- **Application Debugging** - we can set breakpoints within a database object and then run our ASP.NET application. When the database object is executed, the breakpoint will be hit and control turned over to the

debugger. Note that with application debugging we cannot step into a database object from application code. We must explicitly set breakpoints in those stored procedures or UDFs where we want the debugger to stop. Application debugging is examined starting in Step 2.

- **Debugging from a SQL Server Project** - Visual Studio Professional and Team Systems editions include a SQL Server Project type that is commonly used to create managed database objects. We will examine using SQL Server Projects and debugging their contents in the next tutorial.

Visual Studio can debug stored procedures on local and remote SQL Server instances. A local SQL Server instance is one that is installed on the same machine as Visual Studio. If the SQL Server database you are using is not located on your development machine, then it is considered a remote instance. For these tutorials we have been using local SQL Server instances. Debugging stored procedures on a remote SQL server instance requires more configuration steps than when debugging stored procedures on a local instance.

If you are using a local SQL Server instance, you can start with Step 1 and work through this tutorial to the end. If you are using a remote SQL Server instance, however, you will first need to ensure that when debugging you are logged to your development machine with a Windows user account that has a SQL Server login on the remote instance. Moreover, both this database login and the database login used to connect to the database from the running ASP.NET application must be members of the `sysadmin` role. See the Debugging T-SQL Database Objects on Remote Instances section at the end of this tutorial for more information on configuring Visual Studio and SQL Server to debug a remote instance.

Finally, understand that debugging support for T-SQL database objects is not as feature rich as debugging support for .NET applications. For example, breakpoint conditions and filters are not supported, only a subset of the debugging windows are available, you cannot use Edit and Continue, the Immediate window is rendered useless, and so forth. See Limitations on Debugger Commands and Features for more information.

Step 1: Directly Stepping Into a Stored Procedure

Visual Studio makes it easy to directly debug a database object. Let s look at how to use the Direct Database Debugging (DDD) feature to step into the `Products_SelectByCategoryID` stored procedure in the Northwind database. As its name implies, `Products_SelectByCategoryID` returns product information for a particular category; it was created in the Using Existing Stored Procedures for the Typed DataSet s TableAdapters tutorial. Start by going to the Server Explorer and expand the Northwind database node. Next, drill down into the Stored Procedures folder, right-click on the `Products_SelectByCategoryID` stored procedure, and choose the Step Into Stored Procedure option from the context menu. This will start the debugger.

Since the `Products_SelectByCategoryID` stored procedure expects a `@CategoryID` input parameter, we are asked to provide this value. Enter 1, which will return information about the beverages.

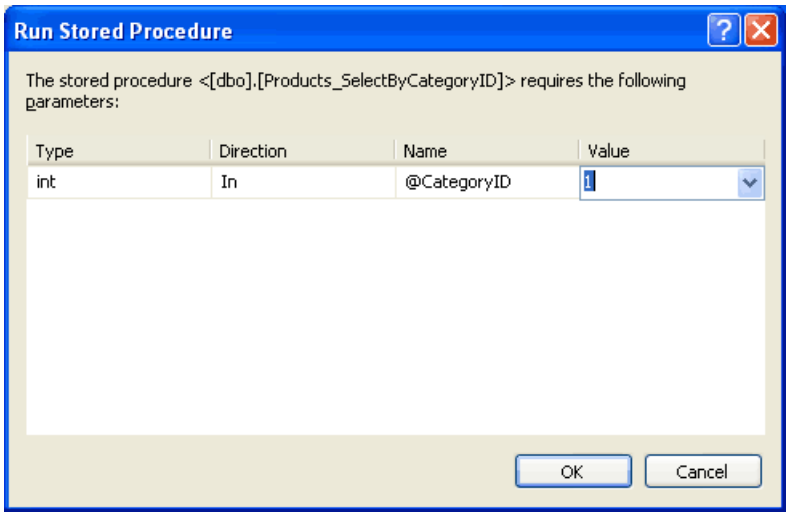


Figure 1: Use the Value 1 for the `@CategoryID` Parameter

After supplying the value for the `@CategoryID` parameter, the stored procedure is executed. Rather than running to completion, though, the debugger halts execution on the first statement. Note the yellow arrow in the margin, indicating the current location in the stored procedure. You can view and edit parameter values through the Watch window or by hovering over the parameter name in the stored procedure.

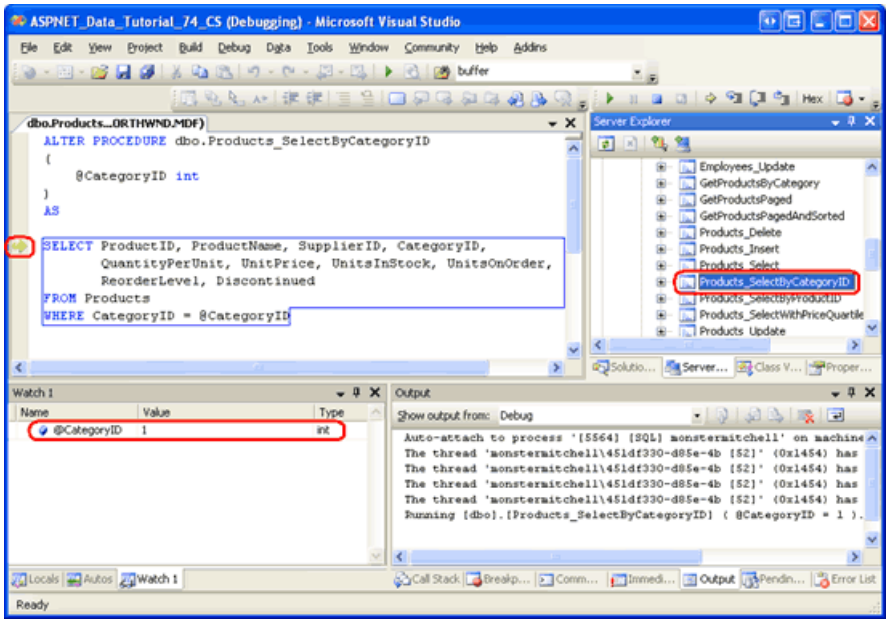


Figure 2: The Debugger has Halted on the First Statement of the Stored Procedure (Click to view full-size image)

To step through the stored procedure one statement at a time, click the Step Over button in the Toolbar or hit the F10 key. The `Products_SelectByCategoryID` stored procedure contains a single `SELECT` statement, so hitting F10 will step over the single statement and complete the execution of the stored procedure. After the stored procedure has completed, its output will appear in the Output window and the debugger will terminate.

> **NOTE**
>
> T-SQL debugging occurs at the statement level; you cannot step into a `SELECT` statement.

Step 2: Configuring the Website for Application Debugging

While debugging a stored procedure directly from the Server Explorer is handy, in many scenarios we are more interested in debugging the stored procedure when it is called from our ASP.NET application. We can add breakpoints to a stored procedure from within Visual Studio and then start debugging the ASP.NET application. When a stored procedure with breakpoints is invoked from the application, execution will halt at the breakpoint and we can view and change the stored procedure s parameter values and step through its statements, just like we did in Step 1.

Before we can start debugging stored procedures called from the application, we need to instruct the ASP.NET web application to integrate with the SQL Server debugger. Start by right-clicking on the website name in the Solution Explorer (`ASPNET_Data_Tutorial_74_VB`). Choose the Property Pages option from the context menu, select the Start Options item on the left, and check the SQL Server checkbox in the Debuggers section (see Figure 3).

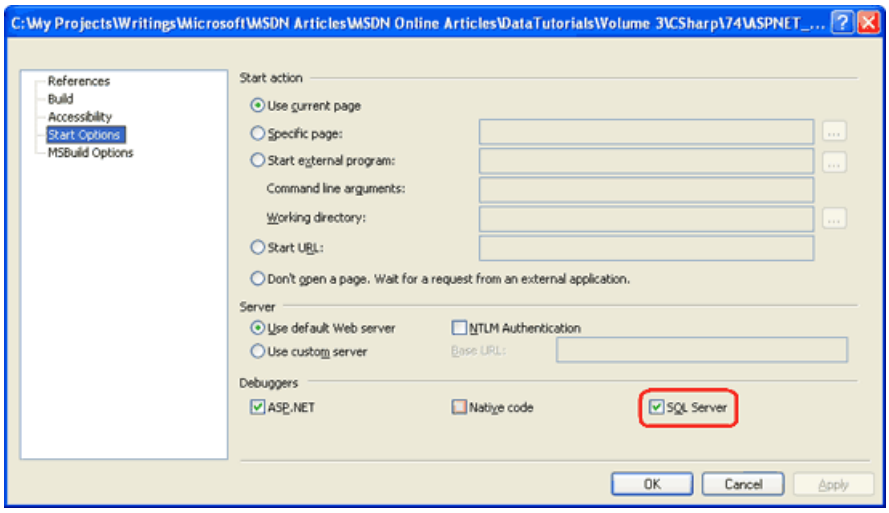


Figure 3: Check the SQL Server Checkbox in the Application s Property Pages (Click to view full-size image)

Additionally, we need to update the database connection string used by the application so that connection pooling is disabled. When a connection to a database is closed, the corresponding `SqlConnection` object is placed in a pool of available connections. When establishing a connection to a database, an available connection object can be retrieved from this pool rather than having to create and establish a new connection. This pooling of connection objects is a performance enhancement and is enabled by default. However, when debugging we want to turn off connection pooling because the debugging infrastructure is not correctly reestablished when working with a connection that was taken from the pool.

To disabled connection pooling, update the `NORTHWNDConnectionString` in `Web.config` so that it includes the setting `Pooling=false` .

```
<connectionStrings>
    <add name="NORTHWNDConnectionString" connectionString=
        "Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\NORTHWND.MDF;
            Integrated Security=True;User Instance=True;Pooling=false"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

NOTE

Once you have finished debugging SQL Server through the ASP.NET application be sure to reinstate connection pooling by removing the `Pooling` setting from the connection string (or by setting it to `Pooling=true`).

At this point the ASP.NET application has been configured to allow Visual Studio to debug SQL Server database objects when invoked through the web application. All that remains now is to add a breakpoint to a stored procedure and start debugging!

Step 3: Adding a Breakpoint and Debugging

Open the `Products_SelectByCategoryID` stored procedure and set a breakpoint at the start of the `SELECT` statement by clicking in the margin at the appropriate place or by placing your cursor at the start of the `SELECT` statement and hitting F9. As Figure 4 illustrates, the breakpoint appears as a red circle in the margin.

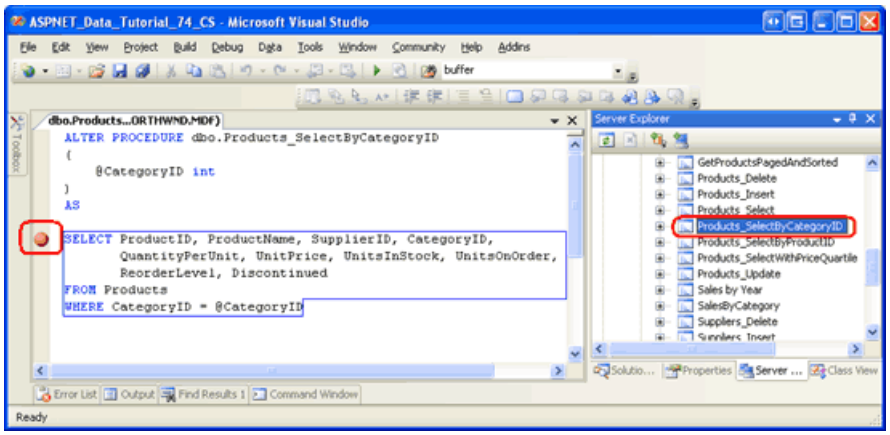


Figure 4: Set a Breakpoint in the `Products_SelectByCategoryID` Stored Procedure (Click to view full-size image)

In order for a SQL database object to be debugged through a client application, it is imperative that the database be configured to support application debugging. When you first set a breakpoint, this setting should automatically be switched on, but it is prudent to double-check. Right-click on the `NORTHWND.MDF` node in the Server Explorer. The context menu should include a checked menu item named Application Debugging .

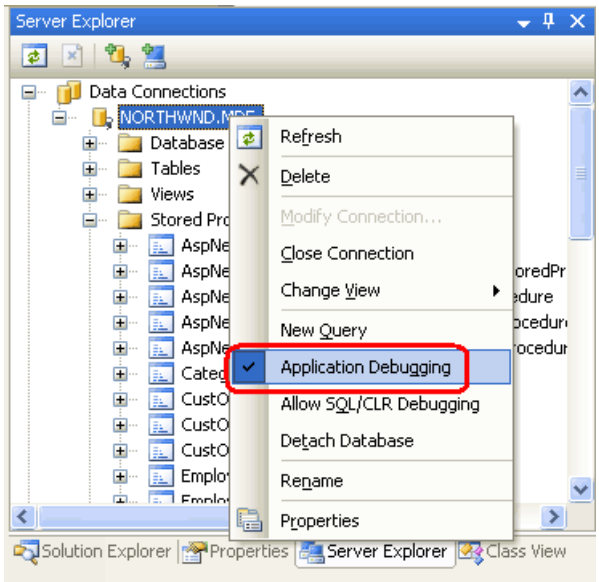


Figure 5: Ensure that the Application Debugging Option is Enabled

With the breakpoint set and the Application Debugging option enabled, we are ready to debug the stored procedure when called from the ASP.NET application. Start the debugger by going to the Debug menu and choosing Start Debugging, by hitting F5, or by clicking the green play icon in the Toolbar. This will start the debugger and launch the website.

The `Products_SelectByCategoryID` stored procedure was created in the Using Existing Stored Procedures for the Typed DataSet s TableAdapters tutorial. Its corresponding web page (`~/AdvancedDAL/ExistingSprocs.aspx`) contains a GridView that displays the results returned by this stored procedure. Visit this page through the browser. Upon reaching the page, the breakpoint in the `Products_SelectByCategoryID` stored procedure will be hit and control returned to Visual Studio. Just like in Step 1, you can step through the stored procedure s statements and view and modify the parameter values.

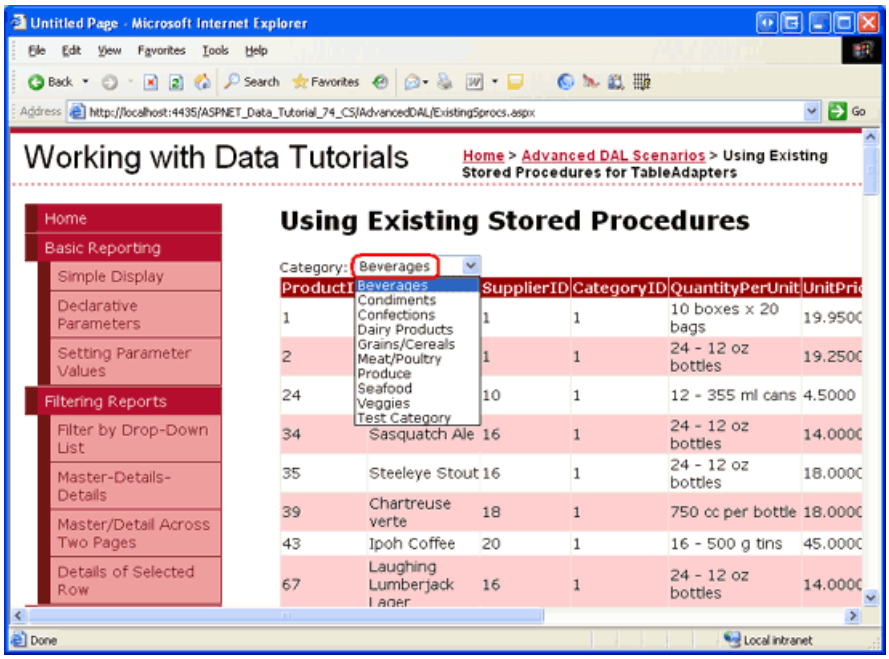


Figure 6: The `ExistingSprocs.aspx` Page Initially Displays the Beverages (Click to view full-size image)

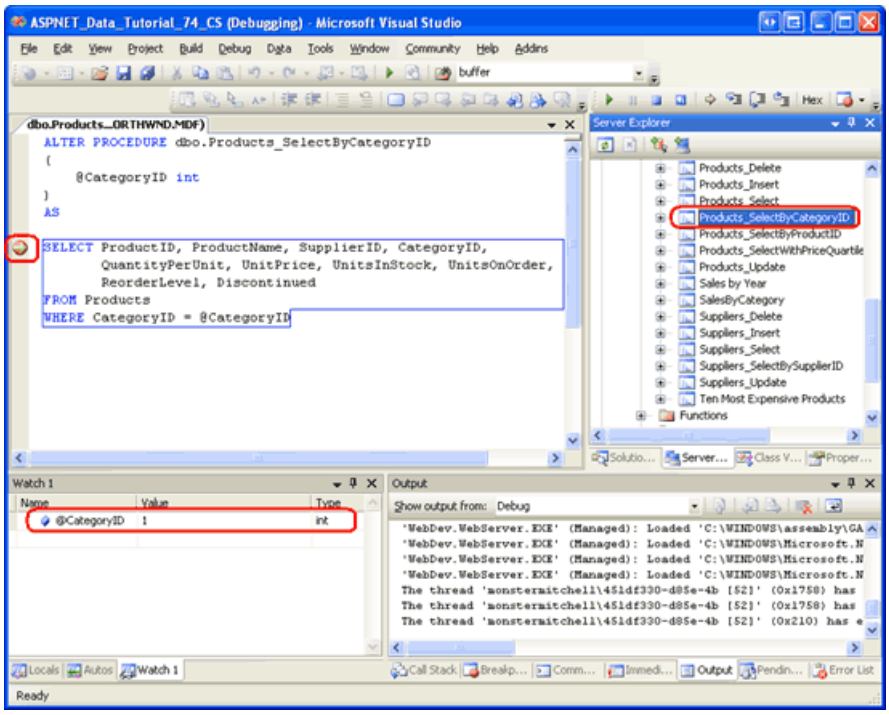


Figure 7: The Stored Procedure s Breakpoint has been Reached (Click to view full-size image)

As the Watch window in Figure 7 shows, the value of the `@CategoryID` parameter is 1. This is because the `ExistingSprocs.aspx` page initially displays products in the beverages category, which has a `CategoryID` value of 1. Choose a different category from the drop-down list. Doing so causes a postback and re-executes the `Products_SelectByCategoryID` stored procedure. The breakpoint is hit again, but this time the `@CategoryID` parameter s value reflects the selected drop-down list item s `CategoryID`.

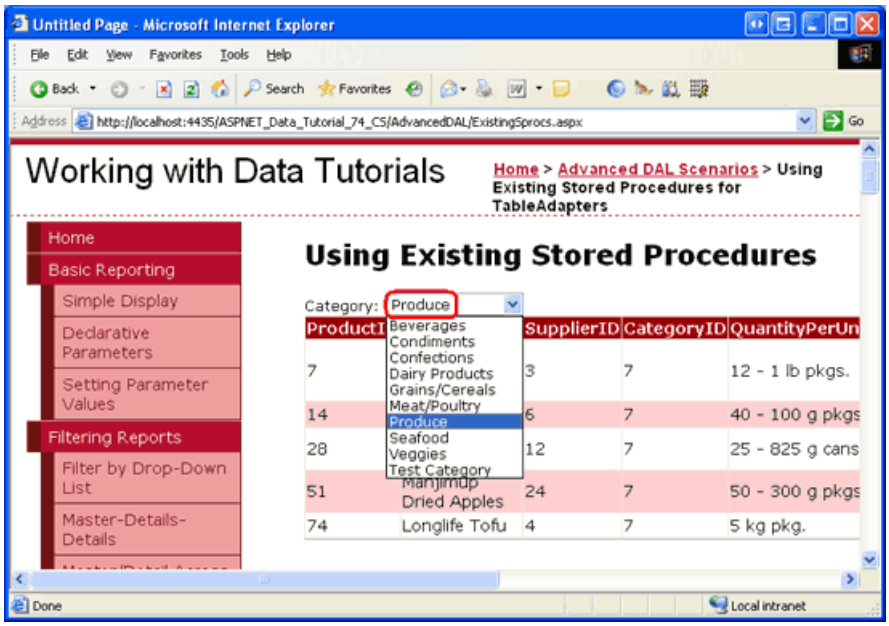


Figure 8: Choose a Different Category from the Drop-Down List (Click to view full-size image)

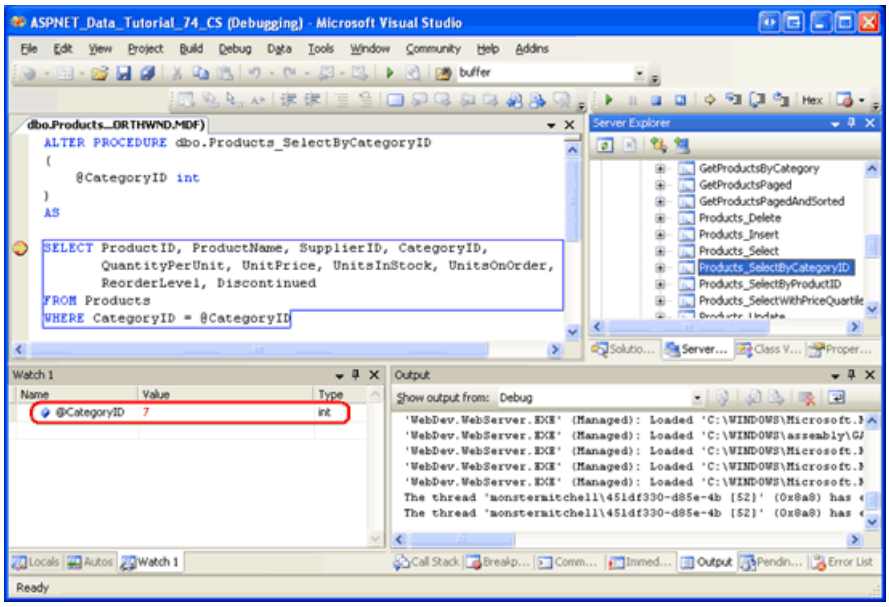


Figure 9: The `@CategoryID` Parameter Reflects the Category Selected from the Web Page (Click to view full-size image)

> **NOTE**
>
> If the breakpoint in the `Products_SelectByCategoryID` stored procedure is not hit when visiting the `ExistingSprocs.aspx` page, make sure that the SQL Server checkbox has been checked in the Debuggers section of the ASP.NET application s Properties Page, that connection pooling has been disabled, and that the database s Application Debugging option is enabled. If you re still having problems, restart Visual Studio and try again.

Debugging T-SQL Database Objects on Remote Instances

Debugging database objects through Visual Studio is fairly straightforward when the SQL Server database instance is on the same machine as Visual Studio. However, if SQL Server and Visual Studio reside on different machines then some careful configuration is required to get everything working properly. There are two core tasks we are faced with:

- Ensure that the login used to connect to the database via ADO.NET belongs to the `sysadmin` role.
- Ensure that the Windows user account used by Visual Studio on the development machine is a valid SQL

Server login account that belongs to the `sysadmin` role.

The first step is relatively straightforward. First, identify the user account used to connect to the database from the ASP.NET application and then, from SQL Server Management Studio, add that login account to the `sysadmin` role.

The second task requires that the Windows user account you use to debug the application be a valid login on the remote database. However, chances are the Windows account you logged on to your workstation with is not a valid login on SQL Server. Rather than adding your particular login account to SQL Server, a better choice would be to designate some Windows user account as the SQL Server debugging account. Then, to debug the database objects of a remote SQL Server instance, you would run Visual Studio using that Windows login account s credentials.

An example should help clarify things. Imagine that there is a Windows account named `SQLDebug` within the Windows domain. This account would need to be added to the remote SQL Server instance as a valid login and as a member of the `sysadmin` role. Then, to debug the remote SQL Server instance from Visual Studio, we would need to run Visual Studio as the `SQLDebug` user. This could be done by logging out of our workstation, logging back in as `SQLDebug`, and then launching Visual Studio, but a simpler approach would be to login to our workstation using our own credentials and then use `runas.exe` to launch Visual Studio as the `SQLDebug` user. `runas.exe` allows a particular application to be executed under the guise of a different user account. To launch Visual Studio as `SQLDebug`, you could enter the following statement at the command line:

```
runas.exe /user:SQLDebug "%PROGRAMFILES%\Microsoft Visual Studio 8\Common7\IDE\devenv.exe"
```

For a more detailed explanation on this process, see William R. Vaughn s *Hitchhiker s Guide to Visual Studio and SQL Server, Seventh Edition* as well as How To: Set SQL Server Permissions for Debugging.

> **NOTE**
> If your development machine is running Windows XP Service Pack 2 you will need to configure the Internet Connection Firewall to allow remote debugging. The How To: Enable SQL Server 2005 Debugging article notes that this involves two steps: (a) On the Visual Studio host machine, you must add `Devenv.exe` to the Exceptions list and open the TCP 135 port; and (b) On the remote (SQL) machine, you must open the TCP 135 port and add `sqlservr.exe` to the Exceptions list. If your domain policy requires network communication to be done through IPSec, you must open the UDP 4500 and UDP 500 ports.

Summary

In addition to providing debugging support for .NET application code, Visual Studio also provides a variety of debugging options for SQL Server 2005. In this tutorial we looked at two of those options: Direct Database Debugging and application debugging. To directly debug a T-SQL database object, locate the object through the Server Explorer then right-click on it and choose Step Into. This starts the debugger and halts on the first statement in the database object, at which point you can step through the object s statements and view and modify parameter values. In Step 1 we used this approach to step into the `Products_SelectByCategoryID` stored procedure.

Application debugging allows breakpoints to be set directly within the database objects. When a database object with breakpoints is invoked from a client application (such as an ASP.NET web application), the program halts as the debugger takes over. Application debugging is useful because it more clearly shows what application action causes a particular database object to be invoked. However, it requires a bit more configuration and setup than Direct Database Debugging.

Database objects can also be debugged through SQL Server Projects. We will look at using SQL Server Projects

and how to use them to create and debug managed database objects in the next tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

PREVIOUS NEXT

Creating Stored Procedures and User-Defined Functions with Managed Code (VB)

9/16/2020 • 36 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

Microsoft SQL Server 2005 integrates with the .NET Common Language Runtime to allow developers to create database objects through managed code. This tutorial shows how to create managed stored procedures and managed user-defined functions with your Visual Basic or C# code. We also see how these editions of Visual Studio allow you to debug such managed database objects.

Introduction

Databases like Microsoft s SQL Server 2005 use the Transact-Structured Query Language (T-SQL) for inserting, modifying, and retrieving data. Most database systems include constructs for grouping a series of SQL statements that can then be executed as a single, reusable unit. Stored procedures are one example. Another is *User-Defined Functions*(UDFs), a construct that we will examine in greater detail in Step 9.

At its core, SQL is designed for working with sets of data. The `SELECT`, `UPDATE`, and `DELETE` statements inherently apply to all records in the corresponding table and are only limited by their `WHERE` clauses. Yet there are many language features designed for working with one record at a time and for manipulating scalar data. `CURSOR` s allow for a set of records to be looped through one at a time. String manipulation functions like `LEFT`, `CHARINDEX`, and `PATINDEX` work with scalar data. SQL also includes control flow statements like `IF` and `WHILE`.

Prior to Microsoft SQL Server 2005, stored procedures and UDFs could only be defined as a collection of T-SQL statements. SQL Server 2005, however, was designed to provide integration with the Common Language Runtime (CLR), which is the runtime used by all .NET assemblies. Consequently, the stored procedures and UDFs in a SQL Server 2005 database can be created using managed code. That is, you can create a stored procedure or UDF as a method in a Visual Basic class. This enables these stored procedures and UDFs to utilize functionality in the .NET Framework and from your own custom classes.

In this tutorial we will examine how to create managed stored procedures and User-Defined Functions and how to integrate them into our Northwind database. Let s get started!

> **NOTE**
> Managed database objects offer some advantages over their SQL counterparts. Language richness and familiarity and the ability to reuse existing code and logic are the main advantages. But managed database objects are likely to be less efficient when working with sets of data that do not involve much procedural logic. For a more thorough discussion on the advantages of using managed code versus T-SQL, check out the Advantages of Using Managed Code to Create Database Objects.

Step 1: Moving the Northwind Database Out of `App_Data`

All of our tutorials thus far have used a Microsoft SQL Server 2005 Express Edition database file in the web application s `App_Data` folder. Placing the database in `App_Data` simplified distributing and running these tutorials as all of the files were located within one directory and required no additional configuration steps to test the

tutorial.

For this tutorial, however, let s move the Northwind database out of `App_Data` and explicitly register it with the SQL Server 2005 Express Edition database instance. While we can perform the steps for this tutorial with the database in the `App_Data` folder, a number of the steps are made much simpler by explicitly registering the database with the SQL Server 2005 Express Edition database instance.

The download for this tutorial has the two database files - `NORTHWND.MDF` and `NORTHWND_log.LDF` - placed in a folder named `DataFiles`. If you are following along with your own implementation of the tutorials, close Visual Studio and move the `NORTHWND.MDF` and `NORTHWND_log.LDF` files from the website s `App_Data` folder to a folder outside of the website. Once the database files have been moved to another folder we need to register the Northwind database with the SQL Server 2005 Express Edition database instance. This can be done from SQL Server Management Studio. If you have a non-Express Edition of SQL Server 2005 installed on your computer then you likely already have Management Studio installed. If you only have SQL Server 2005 Express Edition on your computer then take a moment to download and install Microsoft SQL Server Management Studio Express.

Launch SQL Server Management Studio. As Figure 1 shows, Management Studio starts by asking what server to connect to. Enter localhost\SQLExpress for the server name, choose Windows Authentication in the Authentication drop-down list, and click Connect.

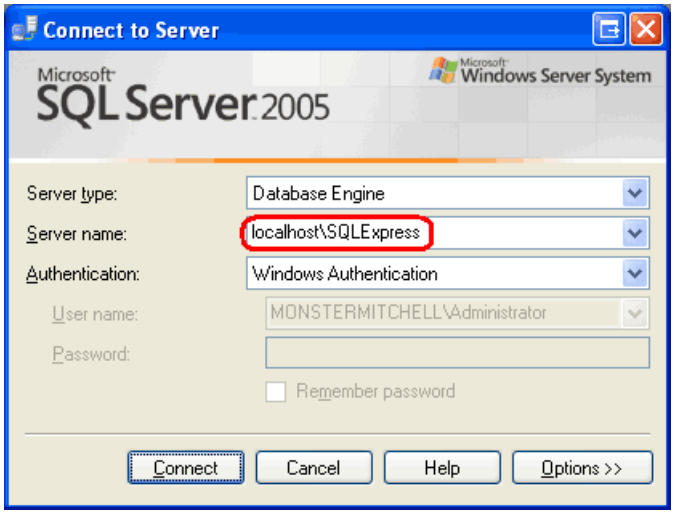


Figure 1: Connect to the Appropriate Database Instance

Once you ve connected, the Object Explorer window will list information about the SQL Server 2005 Express Edition database instance, including its databases, security information, management options, and so forth.

We need to attach the Northwind database in the `DataFiles` folder (or wherever you may have moved it) to the SQL Server 2005 Express Edition database instance. Right-click on the Databases folder and choose the Attach option from the context menu. This will bring up the Attach Databases dialog box. Click the Add button, drill down to the appropriate `NORTHWND.MDF` file, and click OK. At this point your screen should look similar to Figure 2.

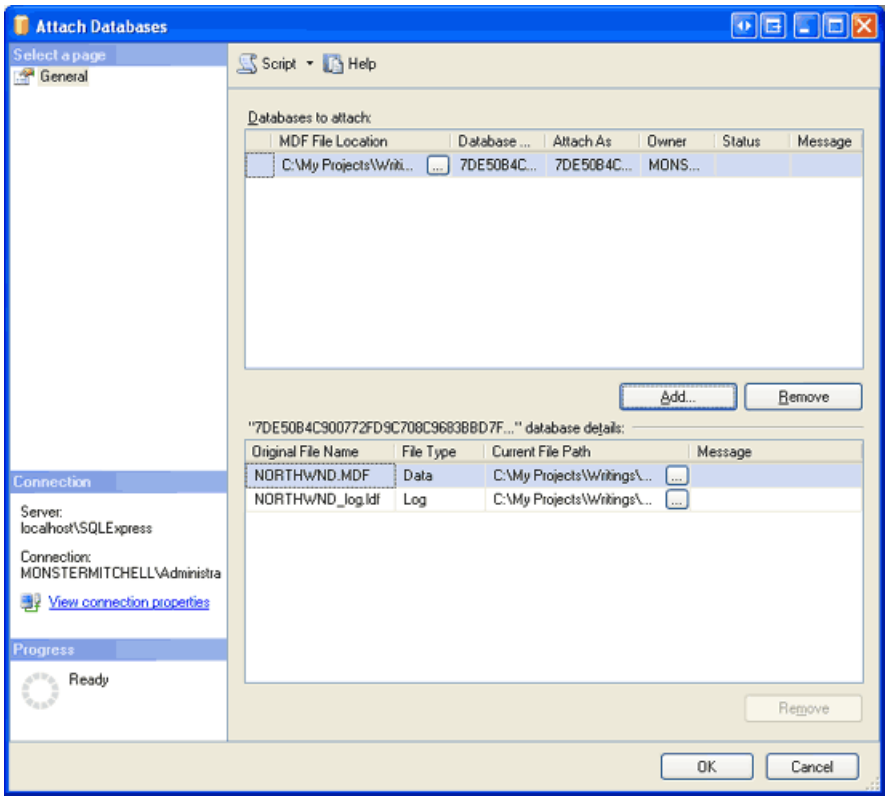


Figure 2: Connect to the Appropriate Database Instance (Click to view full-size image)

> **NOTE**
>
> When connecting to the SQL Server 2005 Express Edition instance through Management Studio the Attach Databases dialog box does not allow you to drill down into user profile directories, such as My Documents. Therefore, make sure to place the `NORTHWND.MDF` and `NORTHWND_log.LDF` files in a non-user profile directory.

Click the OK button to attach the database. The Attach Databases dialog box will close and the Object Explorer should now list the just-attached database. Chances are the Northwind database has a name like

```
9FE54661B32FDD967F51D71D0D5145CC_LINE ARTICLES\DATATUTORIALS\VOLUME
3\CSHARP\73\ASPNET_DATA_TUTORIAL_75_CS\APP_DATA\NORTHWND.MDF
```

. Rename the database to Northwind by right-clicking on the database and choosing Rename.

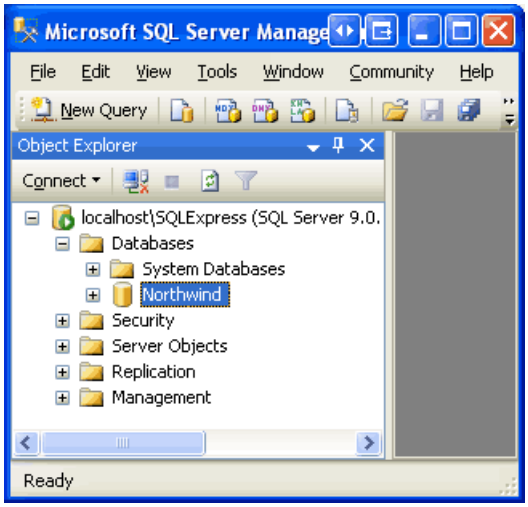


Figure 3: Rename the Database to Northwind

Step 2: Creating a New Solution and SQL Server Project in Visual Studio

To create managed stored procedures or UDFs in SQL Server 2005 we will write the stored procedure and UDF logic as Visual Basic code in a class. Once the code has been written, we will need to compile this class into an assembly (a `.dll` file), register the assembly with the SQL Server database, and then create a stored procedure or UDF object in the database that points to the corresponding method in the assembly. These steps can all be performed manually. We can create the code in any text editor, compile it from the command line using the Visual Basic compiler (`vbc.exe`), register it with the database using the `CREATE ASSEMBLY` command or from Management Studio, and add the stored procedure or UDF object through similar means. Fortunately, the Professional and Team Systems versions of Visual Studio include a SQL Server Project type that automates these tasks. In this tutorial we will walk through using the SQL Server Project type to create a managed stored procedure and UDF.

> **NOTE**
>
> If you are using Visual Web Developer or the Standard edition of Visual Studio, then you will have to use the manual approach instead. Step 13 provides detailed instructions for performing these steps manually. I encourage you to read Steps 2 through 12 before reading Step 13 since these steps include important SQL Server configuration instructions that must be applied regardless of what version of Visual Studio you are using.

Start by opening Visual Studio. From the File menu, choose New Project to display the New Project dialog box (see Figure 4). Drill down to the Database project type and then, from the Templates listed on the right, choose to create a new SQL Server Project. I have chosen to name this project `ManagedDatabaseConstructs` and placed it within a Solution named `Tutorial75`.

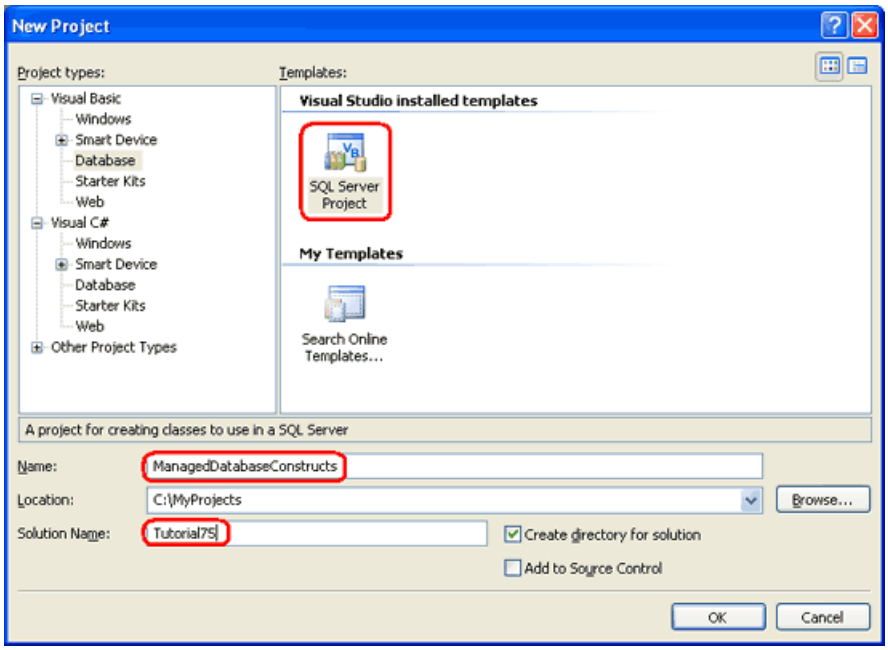


Figure 4: Create a New SQL Server Project (Click to view full-size image)

Click the OK button in the New Project dialog box to create the Solution and SQL Server Project.

A SQL Server Project is tied to a particular database. Consequently, after creating the new SQL Server Project we are immediately asked to specify this information. Figure 5 shows the New Database Reference dialog box that has been filled out to point to the Northwind database we registered in the SQL Server 2005 Express Edition database instance back in Step 1.

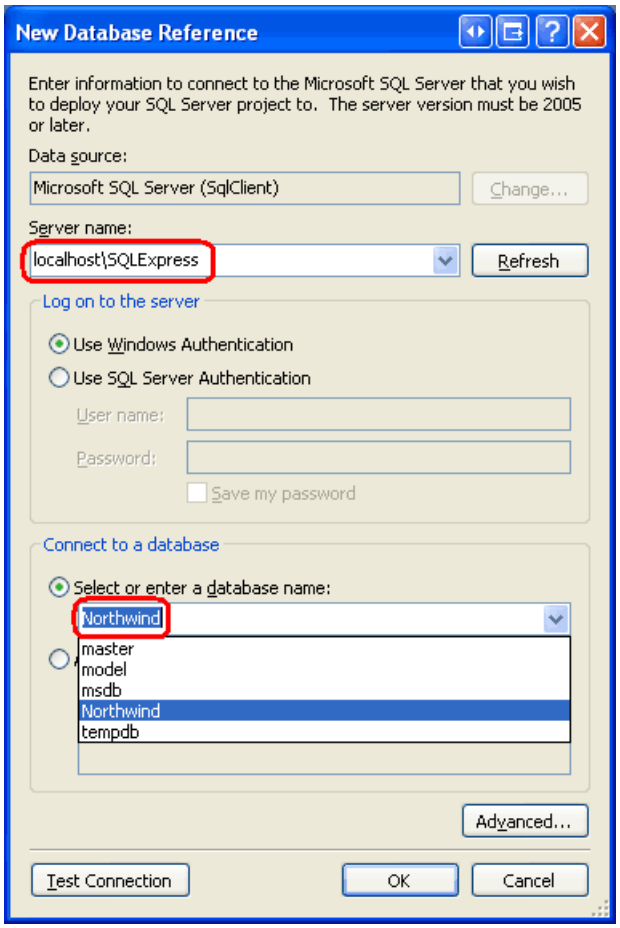


Figure 5: Associate the SQL Server Project with the Northwind Database

In order to debug the managed stored procedures and UDFs we will create within this project, we need to enable SQL/CLR debugging support for the connection. Whenever associating a SQL Server Project with a new database (as we did in Figure 5), Visual Studio asks us if we want to enable SQL/CLR debugging on the connection (see Figure 6). Click Yes.

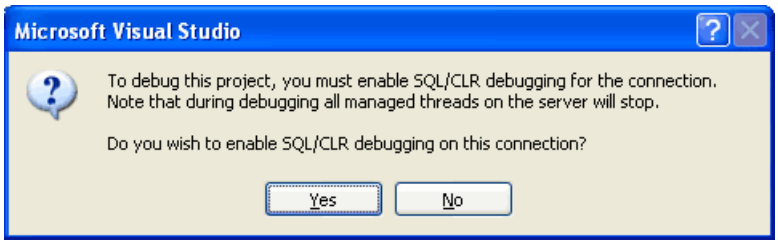


Figure 6: Enable SQL/CLR Debugging

At this point the new SQL Server Project has been added to the Solution. It contains a folder named `Test Scripts` with a file named `Test.sql`, which is used for debugging the managed database objects created in the project. We will look at debugging in Step 12.

We can now add new managed stored procedures and UDFs to this project, but before we do let s first include our existing web application in the Solution. From the File menu select the Add option and choose Existing Web Site. Browse to the appropriate website folder and click OK. As Figure 7 shows, this will update the Solution to include two projects: the website and the `ManagedDatabaseConstructs` SQL Server Project.

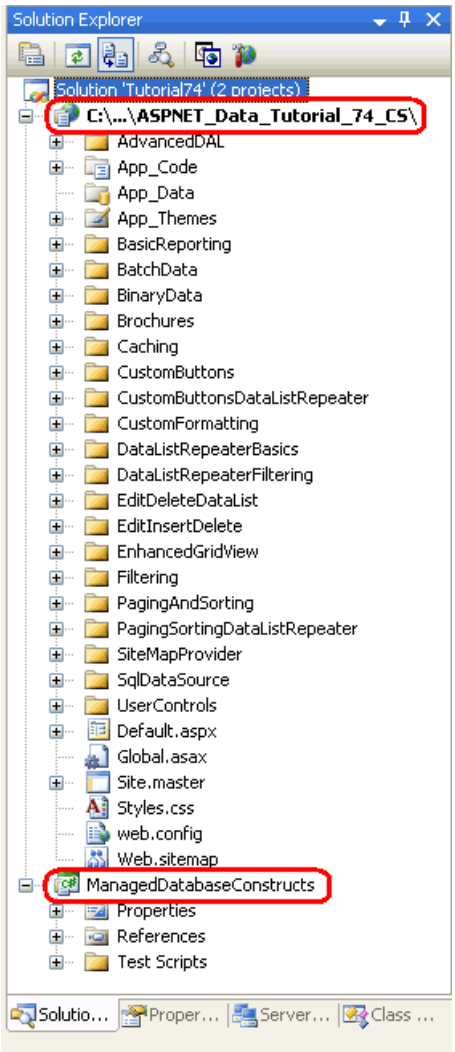


Figure 7: The Solution Explorer Now Includes Two Projects

The `NORTHWNDConnectionString` value in `Web.config` currently references the `NORTHWND.MDF` file in the `App_Data` folder. Since we removed this database from `App_Data` and explicitly registered it in the SQL Server 2005 Express Edition database instance, we need to correspondingly update the `NORTHWNDConnectionString` value. Open the `Web.config` file in the website and change the `NORTHWNDConnectionString` value so that the connection string reads: `Data Source=localhost\SQLExpress;Initial Catalog=Northwind;Integrated Security=True`. After this change, your `<connectionStrings>` section in `Web.config` should look similar to the following:

```
<connectionStrings>
    <add name="NORTHWNDConnectionString" connectionString=
        "Data Source=localhost\SQLExpress;Initial Catalog=Northwind;
            Integrated Security=True;Pooling=false"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

NOTE

As discussed in the preceding tutorial, when debugging a SQL Server object from a client application, such as an ASP.NET website, we need to disable connection pooling. The connection string shown above disables connection pooling (`Pooling=false`). If you do not plan on debugging the managed stored procedures and UDFs from the ASP.NET website, enable connection pooling.

Step 3: Creating a Managed Stored Procedure

To add a managed stored procedure to the Northwind database we first need to create the stored procedure as a method in the SQL Server Project. From the Solution Explorer, right-click on the `ManagedDatabaseConstructs` project name and choose to add a new item. This will display the Add New Item dialog box, which lists the types of managed database objects that can be added to the project. As Figure 8 shows, this includes stored procedures and User-Defined Functions, among others.

Let s start by adding a stored procedure that simply returns all of the products that have been discontinued. Name the new stored procedure file `GetDiscontinuedProducts.vb`.

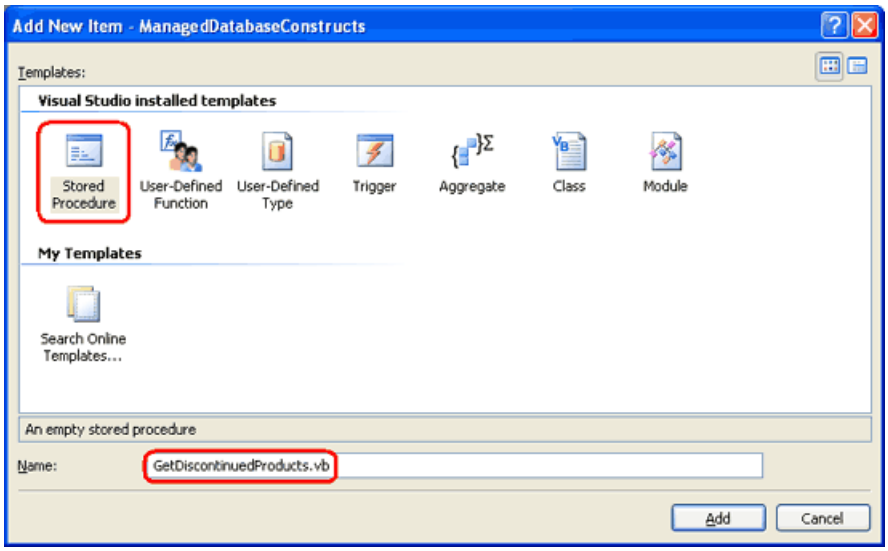


Figure 8: Add a New Stored Procedure Named `GetDiscontinuedProducts.vb` (Click to view full-size image)

This will create a new Visual Basic class file with the following content:

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Partial Public Class StoredProcedures
    <Microsoft.SqlServer.Server.SqlProcedure()> _
    Public Shared Sub  GetDiscontinuedProducts ()
        ' Add your code here
    End Sub
End Class
```

Note that the stored procedure is implemented as a `Shared` method within a `Partial` class file named `StoredProcedures`. Moreover, the `GetDiscontinuedProducts` method is decorated with the `SqlProcedure` attribute, which marks the method as a stored procedure.

The following code creates a `SqlCommand` object and sets its `CommandText` to a `SELECT` query that returns all of the columns from the `Products` table for products whose `Discontinued` field equals 1. It then executes the command and sends the results back to the client application. Add this code to the `GetDiscontinuedProducts` method.

```
' Create the command
Dim myCommand As New SqlCommand()
myCommand.CommandText = _
    "SELECT ProductID, ProductName, SupplierID, CategoryID, " & _
    "       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder, " & _
    "       ReorderLevel, Discontinued " & _
    "FROM Products " & _
    "WHERE Discontinued = 1"
' Execute the command and send back the results
SqlContext.Pipe.ExecuteAndSend(myCommand)
```

All managed database objects have access to a `SqlContext` object that represents the context of the caller. The `SqlContext` provides access to a `SqlPipe` object via its `Pipe` property. This `SqlPipe` object is used to ferry information between the SQL Server database and the calling application. As its name implies, the `ExecuteAndSend` method executes a passed-in `SqlCommand` object and sends the results back to the client application.

> **NOTE**
> Managed database objects are best suited for stored procedures and UDFs that use procedural logic rather than set-based logic. Procedural logic involves working with sets of data on a row-by-row basis or working with scalar data. The `GetDiscontinuedProducts` method we just created, however, involves no procedural logic. Therefore, it would ideally be implemented as a T-SQL stored procedure. It is implemented as a managed stored procedure to demonstrate the steps necessary for creating and deploying managed stored procedures.

Step 4: Deploying the Managed Stored Procedure

With this code complete, we are ready to deploy it to the Northwind database. Deploying a SQL Server Project compiles the code into an assembly, registers the assembly with the database, and creates the corresponding objects in the database, linking them to the appropriate methods in the assembly. The exact set of tasks performed by the Deploy option is more precisely spelled out in Step 13. Right-click on the `ManagedDatabaseConstructs` project name in the Solution Explorer and choose the Deploy option. However, deployment fails with the following error: Incorrect syntax near 'EXTERNAL'. You may need to set the compatibility level of the current database to a higher value to enable this feature. See help for the stored procedure `sp_dbcmptlevel`.

This error message occurs when attempting to register the assembly with the Northwind database. In order to register an assembly with a SQL Server 2005 database, the database s compatibility level must be set to 90. By default, new SQL Server 2005 databases have a compatibility level of 90. However, databases created using Microsoft SQL Server 2000 have a default compatibility level of 80. Since the Northwind database was initially a Microsoft SQL Server 2000 database, its compatibility level is currently set to 80 and therefore needs to be increased to 90 in order to register managed database objects.

To update the database s compatibility level, open a New Query window in Management Studio and enter:

```
exec sp_dbcmptlevel 'Northwind', 90
```

Click the Execute icon in the Toolbar to run the above query.

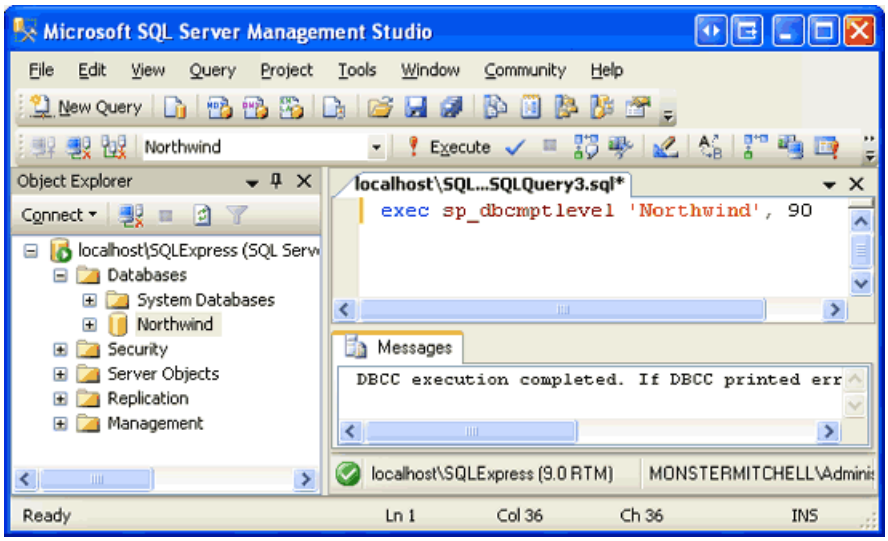


Figure 9: Update the Northwind Database s Compatibility Level (Click to view full-size image)

After updating the compatibility level, redeploy the SQL Server Project. This time the deployment should complete without error.

Return to SQL Server Management Studio, right-click on the Northwind database in the Object Explorer, and choose Refresh. Next, drill down into the Programmability folder and then expand the Assemblies folder. As Figure 10 shows, the Northwind database now includes the assembly generated by the `ManagedDatabaseConstructs` project.

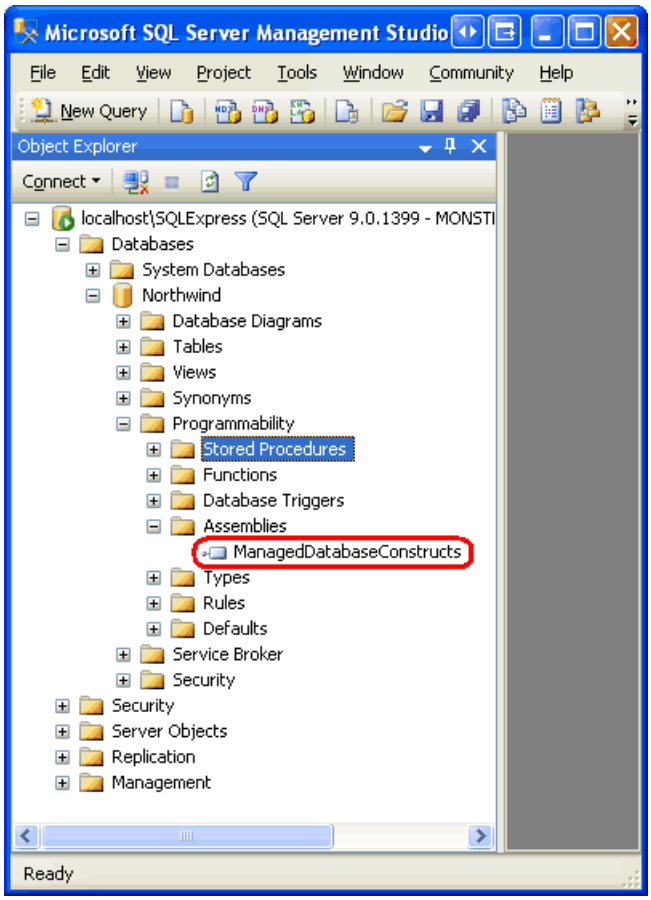


Figure 10: The `ManagedDatabaseConstructs` Assembly is Now Registered with the Northwind Database

Also expand the Stored Procedures folder. There you will see a stored procedure named `GetDiscontinuedProducts`. This stored procedure was created by the deployment process and points to the `GetDiscontinuedProducts` method in the `ManagedDatabaseConstructs` assembly. When the `GetDiscontinuedProducts` stored procedure is executed, it, in turn, executes the `GetDiscontinuedProducts` method. Since this is a managed stored procedure it cannot be edited

through Management Studio (hence the lock icon next to the stored procedure name).

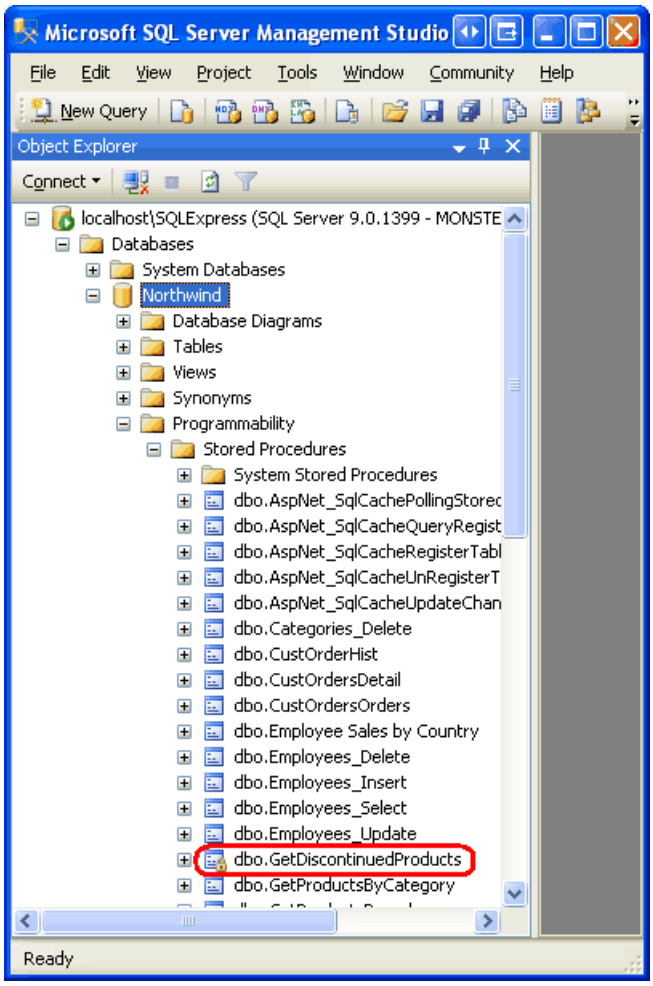


Figure 11: The `GetDiscontinuedProducts` Stored Procedure is Listed in the Stored Procedures Folder

There is still one more hurdle we have to overcome before we can call the managed stored procedure: the database is configured to prevent execution of managed code. Verify this by opening a new query window and executing the `GetDiscontinuedProducts` stored procedure. You will receive the following error message: Execution of user code in the .NET Framework is disabled. Enable 'clr enabled configuration option.

To examine the Northwind database s configuration information, enter and execute the command `exec sp_configure` in the query window. This shows that the clr enabled setting is currently set to 0.

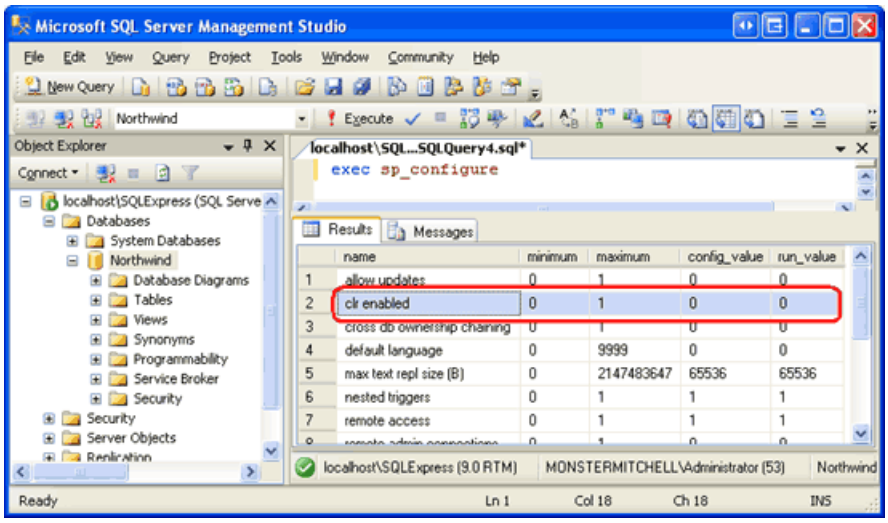


Figure 12: The clr enabled Setting is Currently Set to 0 (Click to view full-size image)

Note that each configuration setting in Figure 12 has four values listed with it: the minimum and maximum values

and the config and run values. To update the config value for the clr enabled setting, execute the following command:

```
exec sp_configure 'clr enabled', 1
```

If you re-run the `exec sp_configure` you will see that the above statement updated the clr enabled setting s config value to 1, but that the run value is still set to 0. For this configuration change to take affect we need to execute the `RECONFIGURE` command, which will set the run value to the current config value. Simply enter `RECONFIGURE` in the query window and click the Execute icon in the Toolbar. If you run `exec sp_configure` now you should see a value of 1 for the clr enabled setting s config and run values.

With the clr enabled configuration complete, we are ready to run the managed `GetDiscontinuedProducts` stored procedure. In the query window enter and execute the command `exec` `GetDiscontinuedProducts`. Invoking the stored procedure causes the corresponding managed code in the `GetDiscontinuedProducts` method to execute. This code issues a `SELECT` query to return all products that are discontinued and returns this data to the calling application, which is SQL Server Management Studio in this instance. Management Studio receives these results and displays them in the Results window.

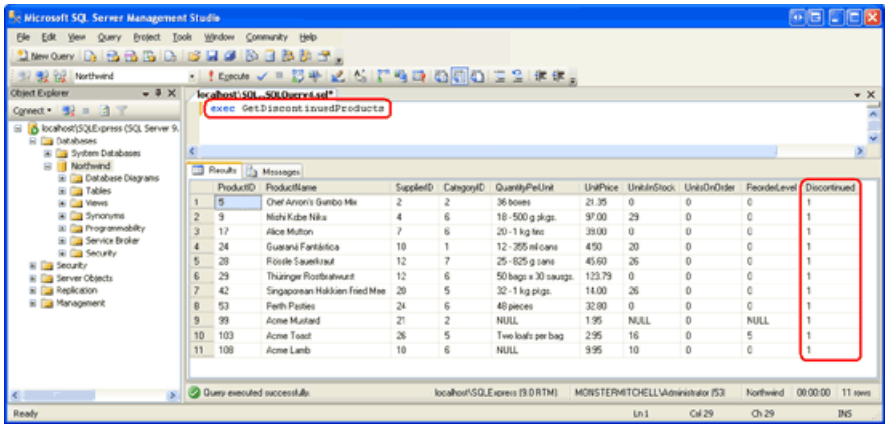


Figure 13: The `GetDiscontinuedProducts` Stored Procedure Returns All Discontinued Products (Click to view full-size image)

Step 5: Creating Managed Stored Procedures that Accept Input Parameters

Many of the queries and stored procedures we have created throughout these tutorials have used *parameters*. For example, in the Creating New Stored Procedures for the Typed DataSet s TableAdapters tutorial we created a stored procedure named `GetProductsByCategoryID` that accepted an input parameter named `@CategoryID`. The stored procedure then returned all products whose `CategoryID` field matched the value of the supplied `@CategoryID` parameter.

To create a managed stored procedure that accepts input parameters, simply specify those parameters in the method s definition. To illustrate this, let s add another managed stored procedure to the `ManagedDatabaseConstructs` project named `GetProductsWithPriceLessThan`. This managed stored procedure will accept an input parameter specifying a price and will return all products whose `UnitPrice` field is less than the parameter s value.

To add a new stored procedure to the project, right-click on the `ManagedDatabaseConstructs` project name and choose to add a new stored procedure. Name the file `GetProductsWithPriceLessThan.vb`. As we saw in Step 3, this will create a new Visual Basic class file with a method named `GetProductsWithPriceLessThan` placed within the `Partial` class `StoredProcedures`.

Update the `GetProductsWithPriceLessThan` method s definition so that it accepts a `SqlMoney` input parameter

named `price` and write the code to execute and return the query results:

```
<Microsoft.SqlServer.Server.SqlProcedure()> _
Public Shared Sub GetProductsWithPriceLessThan(ByVal price As SqlMoney)
    'Create the command
    Dim myCommand As New SqlCommand()
    myCommand.CommandText = _
        "SELECT ProductID, ProductName, SupplierID, CategoryID, " & _
        "       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder, " & _
        "       ReorderLevel, Discontinued " & _
        "FROM Products " & _
        "WHERE UnitPrice < @MaxPrice"
    myCommand.Parameters.AddWithValue("@MaxPrice", price)
    ' Execute the command and send back the results
    SqlContext.Pipe.ExecuteAndSend(myCommand)
End Sub
```

The `GetProductsWithPriceLessThan` method s definition and code closely resembles the definition and code of the `GetDiscontinuedProducts` method created in Step 3. The only differences are that the `GetProductsWithPriceLessThan` method accepts as input parameter (`price`), the `SqlCommand` s query includes a parameter (`@MaxPrice`), and a parameter is added to the `SqlCommand` s `Parameters` collection is and assigned the value of the `price` variable.

After adding this code, redeploy the SQL Server Project. Next, return to SQL Server Management Studio and Refresh the Stored Procedures folder. You should see a new entry, `GetProductsWithPriceLessThan`. From a query window, enter and execute the command `exec GetProductsWithPriceLessThan 25`, which will list all products less than $25, as Figure 14 shows.

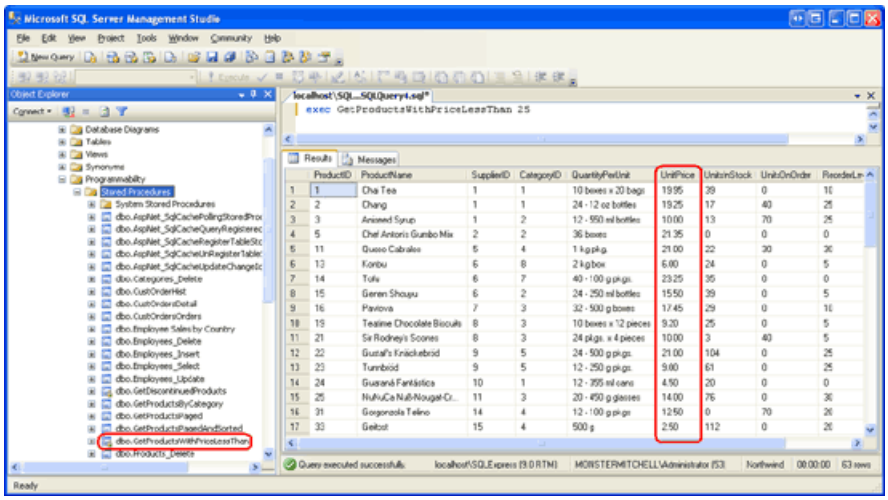


Figure 14: Products Under $25 are Displayed (Click to view full-size image)

Step 6: Calling the Managed Stored Procedure from the Data Access Layer

At this point we have added the `GetDiscontinuedProducts` and `GetProductsWithPriceLessThan` managed stored procedures to the `ManagedDatabaseConstructs` project and have registered them with the Northwind SQL Server database. We also invoked these managed stored procedures from SQL Server Management Studio (see Figures 13 and 14). In order for our ASP.NET application to use these managed stored procedures, however, we need to add them to the Data Access and Business Logic Layers in the architecture. In this step we will add two new methods to the `ProductsTableAdapter` in the `NorthwindWithSprocs` Typed DataSet, which was initially created in the Creating New Stored Procedures for the Typed DataSet s TableAdapters tutorial. In Step 7 we will add corresponding methods to the BLL.

Open the `NorthwindWithSprocs` Typed DataSet in Visual Studio and start by adding a new method to the `ProductsTableAdapter` named `GetDiscontinuedProducts`. To add a new method to a TableAdapter, right-click on the TableAdapter s name in the Designer and choose the Add Query option from the context menu.

> **NOTE**
>
> Since we moved the Northwind database from the `App_Data` folder to the SQL Server 2005 Express Edition database instance, it is imperative that the corresponding connection string in Web.config be updated to reflect this change. In Step 2 we discussed updating the `NORTHWNDConnectionString` value in `Web.config`. If you forgot to make this update, then you will see the error message Failed to add query. Unable to find connection `NORTHWNDConnectionString` for object `Web.config` in a dialog box when attempting to add a new method to the TableAdapter. To resolve this error, click OK and then go to `Web.config` and update the `NORTHWNDConnectionString` value as discussed in Step 2. Then try re-adding the method to the TableAdapter. This time it should work without error.

Adding a new method launches the TableAdapter Query Configuration wizard, which we have used many times in past tutorials. The first step asks us to specify how the TableAdapter should access the database: through an ad-hoc SQL statement or via a new or existing stored procedure. Since we have already created and registered the `GetDiscontinuedProducts` managed stored procedure with the database, choose the Use existing stored procedure option and hit Next.

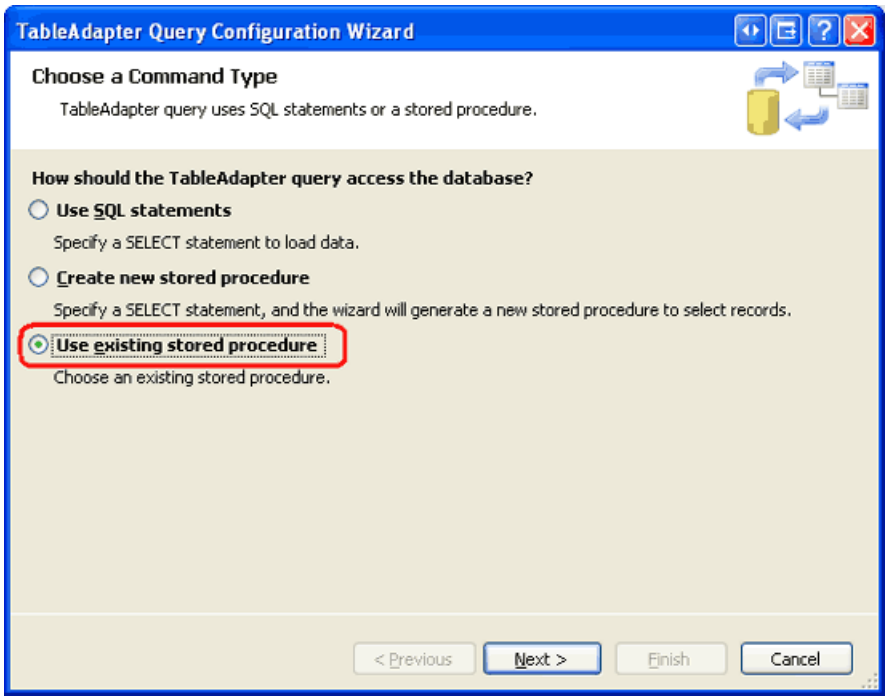


Figure 15: Choose the Use existing stored procedure Option (Click to view full-size image)

The next screen prompts us for the stored procedure the method will invoke. Choose the `GetDiscontinuedProducts` managed stored procedure from the drop-down list and hit Next.

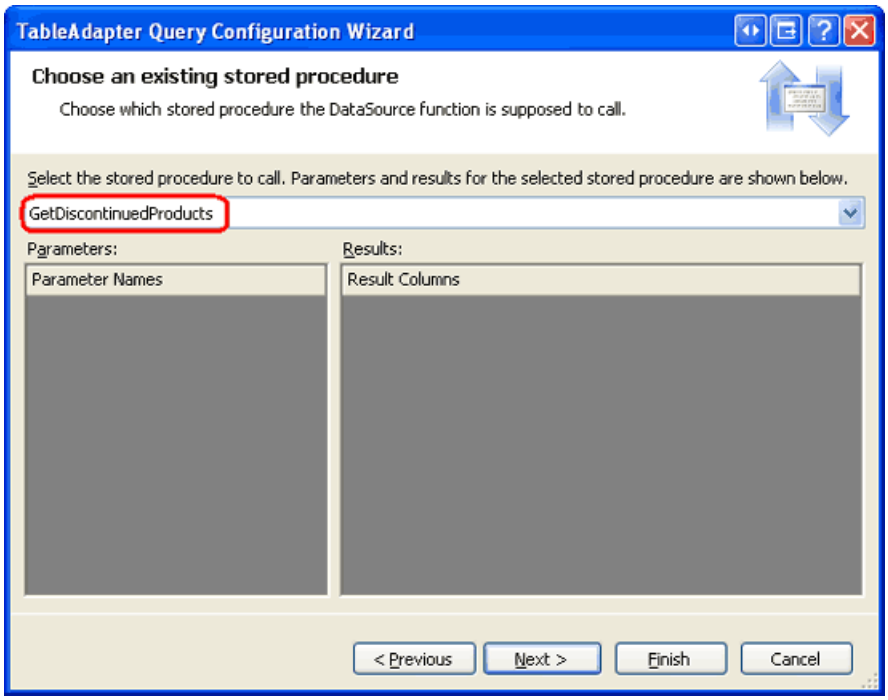


Figure 16: Select the `GetDiscontinuedProducts` Managed Stored Procedure (Click to view full-size image)

We are then asked to specify whether the stored procedure returns rows, a single value, or nothing. Since `GetDiscontinuedProducts` returns the set of discontinued product rows, choose the first option (Tabular data) and click Next.

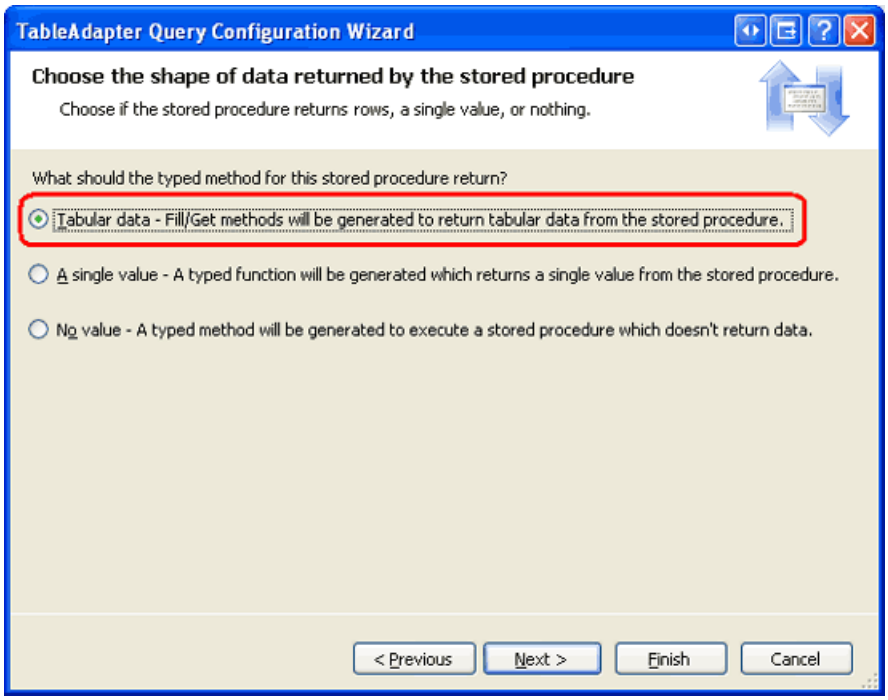


Figure 17: Select the Tabular Data Option (Click to view full-size image)

The final wizard screen allows us to specify the data access patterns used and the names of the resulting methods. Leave both checkboxes checked and name the methods `FillByDiscontinued` and `GetDiscontinuedProducts`. Click Finish to complete the wizard.

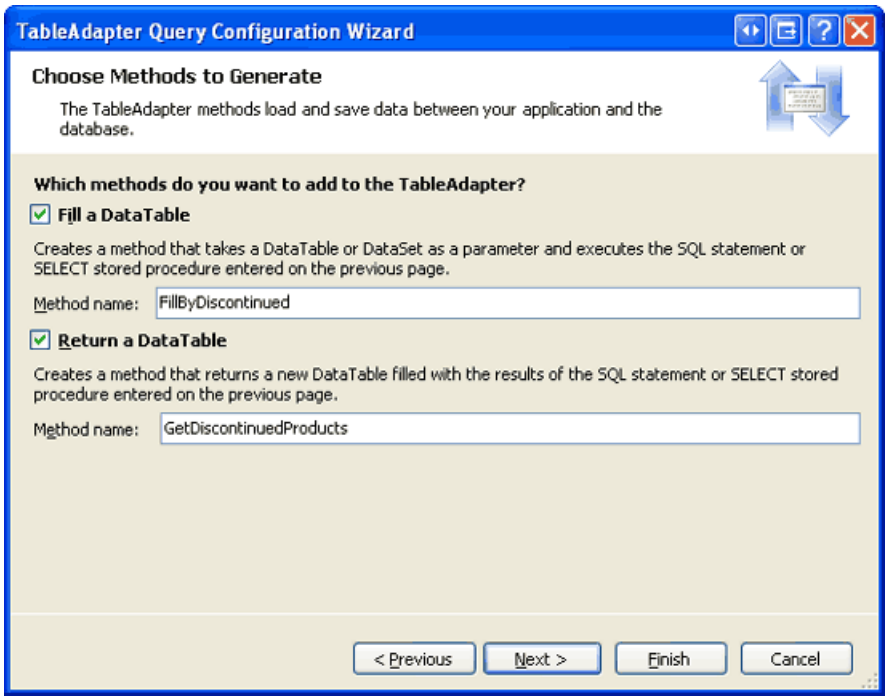


Figure 18: Name the Methods `FillByDiscontinued` and `GetDiscontinuedProducts` (Click to view full-size image)

Repeat these steps to create methods named `FillByPriceLessThan` and `GetProductsWithPriceLessThan` in the `ProductsTableAdapter` for the `GetProductsWithPriceLessThan` managed stored procedure.

Figure 19 shows a screenshot of the DataSet Designer after adding the methods to the `ProductsTableAdapter` for the `GetDiscontinuedProducts` and `GetProductsWithPriceLessThan` managed stored procedures.

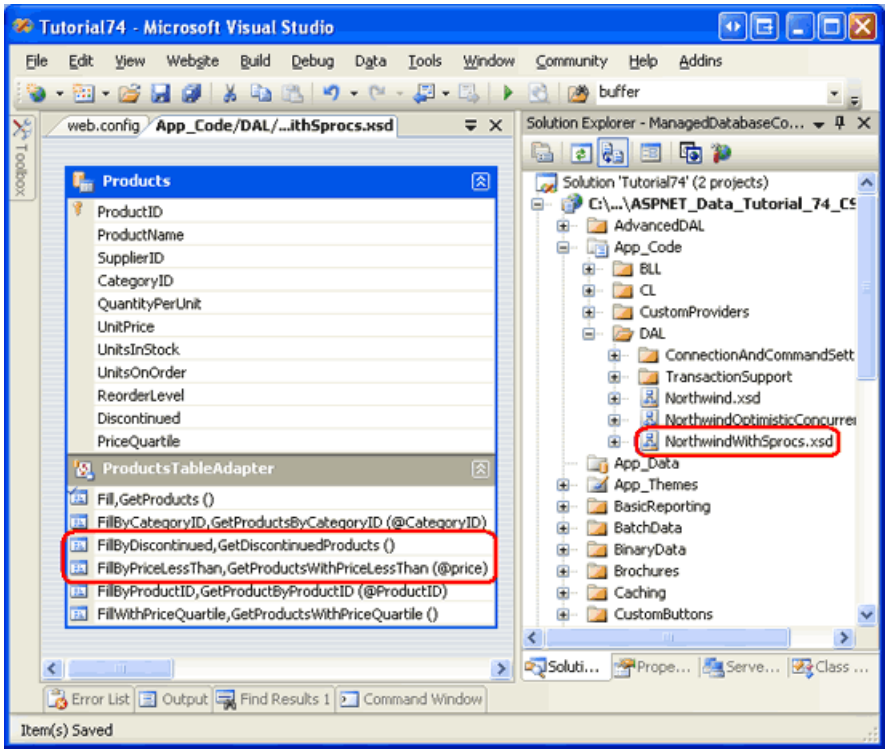


Figure 19: The `ProductsTableAdapter` Includes the New Methods Added in this Step (Click to view full-size image)

Step 7: Adding Corresponding Methods to the Business Logic Layer

Now that we have updated the Data Access Layer to include methods for calling the managed stored procedures added in Steps 4 and 5, we need to add corresponding methods to the Business Logic Layer. Add the following two methods to the `ProductsBLLWithSprocs` class:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetDiscontinuedProducts() As NorthwindWithSprocs.ProductsDataTable
    Return Adapter.GetDiscontinuedProducts()
End Function
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetProductsWithPriceLessThan(ByVal priceLessThan As Decimal) _
    As NorthwindWithSprocs.ProductsDataTable
    Return Adapter.GetProductsWithPriceLessThan(priceLessThan)
End Function
```

Both methods simply call the corresponding DAL method and return the `ProductsDataTable` instance. The `DataObjectMethodAttribute` markup above each method causes these methods to be included in the drop-down list in the SELECT tab of the ObjectDataSource s Configure Data Source wizard.

Step 8: Invoking the Managed Stored Procedures from the Presentation Layer

With the Business Logic and Data Access Layers augmented to include support for calling the `GetDiscontinuedProducts` and `GetProductsWithPriceLessThan` managed stored procedures, we can now display these stored procedures results through an ASP.NET page.

Open the `ManagedFunctionsAndSprocs.aspx` page in the `AdvancedDAL` folder and, from the Toolbox, drag a GridView onto the Designer. Set the GridView s `ID` property to `DiscontinuedProducts` and, from its smart tag, bind it to a new ObjectDataSource named `DiscontinuedProductsDataSource`. Configure the ObjectDataSource to pull its data from the `ProductsBLLWithSprocs` class s `GetDiscontinuedProducts` method.

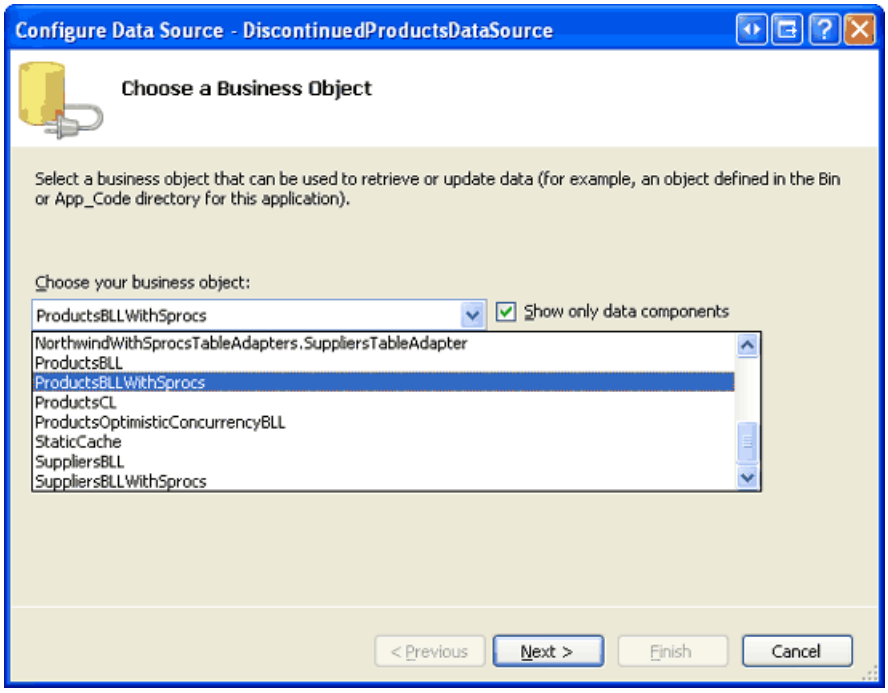


Figure 20: Configure the ObjectDataSource to Use the `ProductsBLLWithSprocs` Class (Click to view full-size image)

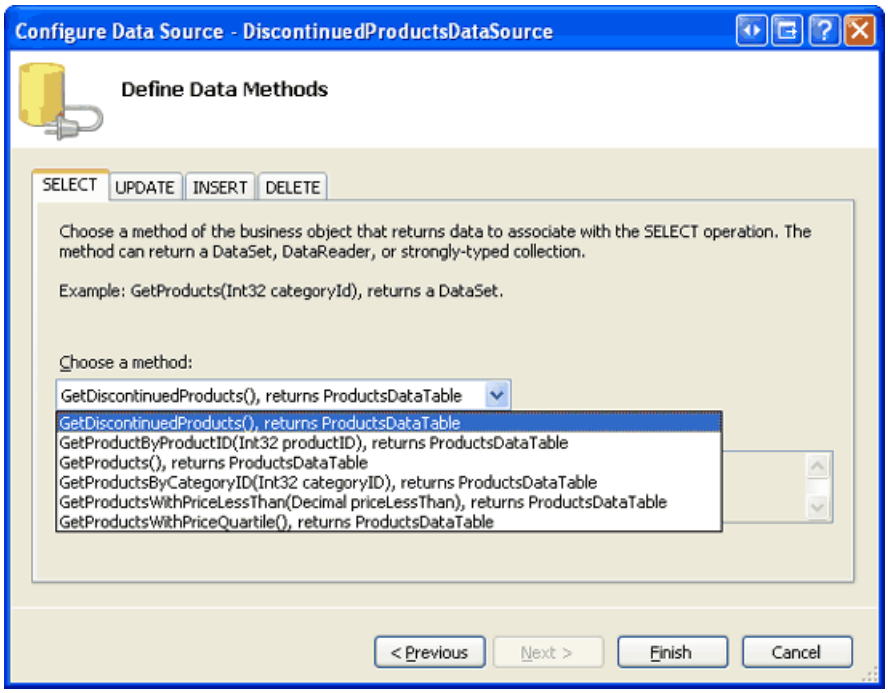


Figure 21: Choose the `GetDiscontinuedProducts` Method from the Drop-Down List in the SELECT Tab (Click to view full-size image)

Since this grid will be used to just display product information, set the drop-down lists in the UPDATE, INSERT, and DELETE tabs to (None) and then click Finish.

Upon completing the wizard, Visual Studio will automatically add a BoundField or CheckBoxField for each data field in the `ProductsDataTable`. Take a moment to remove all of these fields except for `ProductName` and `Discontinued`, at which point your GridView and ObjectDataSource s declarative markup should look similar to the following:

```
<asp:GridView ID="DiscontinuedProducts" runat="server"
    AutoGenerateColumns="False" DataKeyNames="ProductID"
    DataSourceID="DiscontinuedProductsDataSource">
    <Columns>
        <asp:BoundField DataField="ProductName" HeaderText="ProductName"
            SortExpression="ProductName" />
        <asp:CheckBoxField DataField="Discontinued"
            HeaderText="Discontinued"
            SortExpression="Discontinued" />
    </Columns>
</asp:GridView>
<asp:ObjectDataSource ID="DiscontinuedProductsDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetDiscontinuedProducts" TypeName="ProductsBLLWithSprocs">
</asp:ObjectDataSource>
```

Take a moment to view this page through a browser. When the page is visited, the ObjectDataSource calls the `ProductsBLLWithSprocs` class s `GetDiscontinuedProducts` method. As we saw in Step 7, this method calls down to the DAL s `ProductsDataTable` class s `GetDiscontinuedProducts` method, which invokes the `GetDiscontinuedProducts` stored procedure. This stored procedure is a managed stored procedure and executes the code we created in Step 3, returning the discontinued products.

The results returned by the managed stored procedure are packaged up into a `ProductsDataTable` by the DAL and then returned to the BLL, which then returns them to the Presentation Layer where they are bound to the GridView and displayed. As expected, the grid lists those products that have been discontinued.

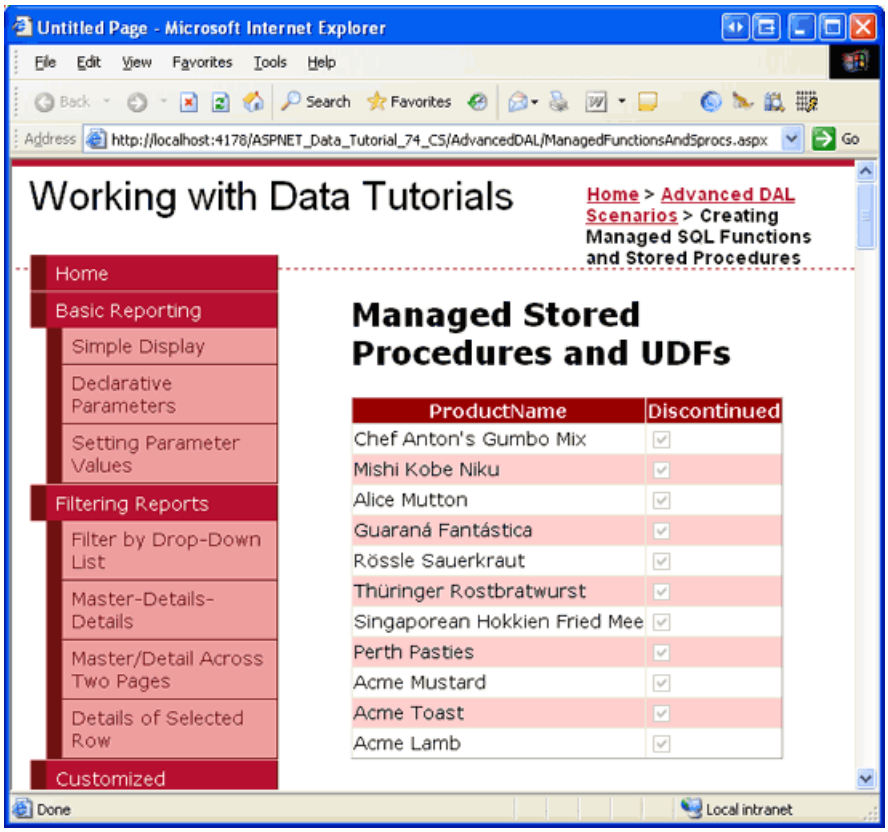


Figure 22: The Discontinued Products are Listed (Click to view full-size image)

For further practice, add a TextBox and another GridView to the page. Have this GridView display the products less than the amount entered into the TextBox by calling the `ProductsBLLWithSprocs` class s `GetProductsWithPriceLessThan` method.

Step 9: Creating and Calling T-SQL UDFs

User-Defined Functions, or UDFs, are database objects the closely mimic the semantics of functions in programming languages. Like a function in Visual Basic, UDFs can include a variable number of input parameters and return a value of a particular type. A UDF can return either scalar data - a string, an integer, and so forth - or tabular data. Let s take a quick look at both types of UDFs, starting with a UDF that returns a scalar data type.

The following UDF calculates the estimated value of the inventory for a particular product. It does so by taking in three input parameters - the `UnitPrice`, `UnitsInStock`, and `Discontinued` values for a particular product - and returns a value of type `money`. It computes the estimated value of the inventory by multiplying the `UnitPrice` by the `UnitsInStock`. For discontinued items, this value is halved.

```
CREATE FUNCTION udf_ComputeInventoryValue
(
    @UnitPrice money,
    @UnitsInStock smallint,
    @Discontinued bit
)
RETURNS money
AS
BEGIN
    DECLARE @Value decimal
    SET @Value = ISNULL(@UnitPrice, 0) * ISNULL(@UnitsInStock, 0)
    IF @Discontinued = 1
        SET @Value = @Value * 0.5

    RETURN @Value
END
```

Once this UDF has been added to the database, it can be found through Management Studio by expanding the Programmability folder, then Functions, and then Scalar-value Functions. It can be used in a `SELECT` query like so:

```
SELECT ProductID, ProductName, dbo.udf_ComputeInventoryValue
    (UnitPrice, UnitsInStock, Discontinued) as InventoryValue
FROM Products
ORDER BY InventoryValue DESC
```

I have added the `udf_ComputeInventoryValue` UDF to the Northwind database; Figure 23 shows the output of the above `SELECT` query when viewed through Management Studio. Also note that the UDF is listed under the Scalar-value Functions folder in the Object Explorer.

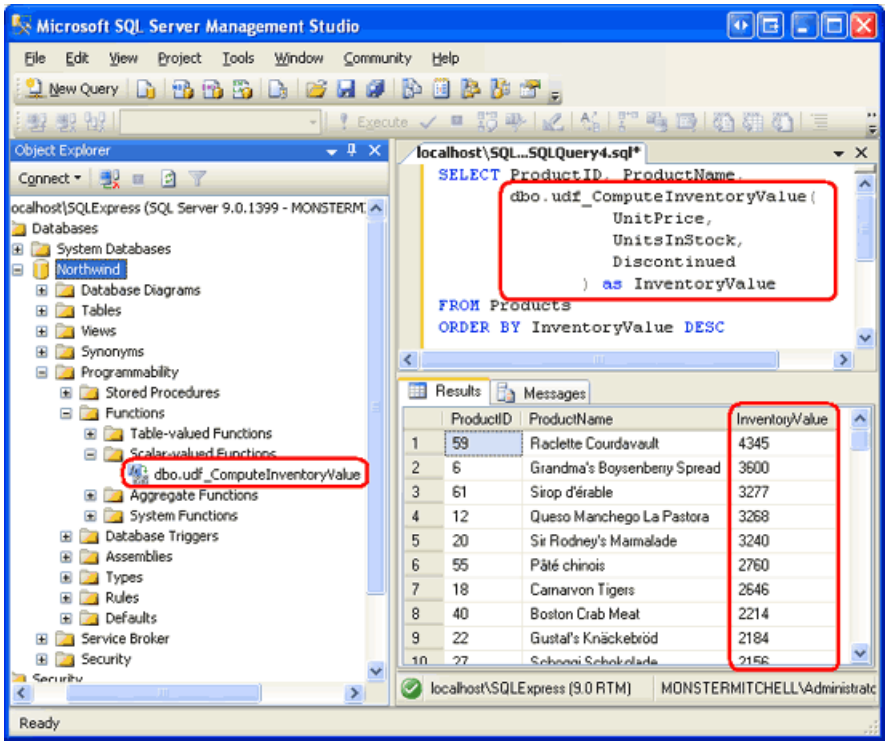


Figure 23: Each Product s Inventory Values is Listed (Click to view full-size image)

UDFs can also return tabular data. For example, we can create a UDF that returns products that belong to a particular category:

```
CREATE FUNCTION dbo.udf_GetProductsByCategoryID
(
    @CategoryID int
)
RETURNS TABLE
AS
RETURN
(
    SELECT ProductID, ProductName, SupplierID, CategoryID,
           QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder,
           ReorderLevel, Discontinued
    FROM Products
    WHERE CategoryID = @CategoryID
)
```

The `udf_GetProductsByCategoryID` UDF accepts a `@CategoryID` input parameter and returns the results of the specified `SELECT` query. Once created, this UDF can be referenced in the `FROM` (or `JOIN`) clause of a `SELECT` query. The following example would return the `ProductID`, `ProductName`, and `CategoryID` values for each of the beverages.

```
SELECT ProductID, ProductName, CategoryID
FROM dbo.udf_GetProductsByCategoryID(1)
```

I have added the `udf_GetProductsByCategoryID` UDF to the Northwind database; Figure 24 shows the output of the above `SELECT` query when viewed through Management Studio. UDFs that return tabular data can be found in the Object Explorer s Table-value Functions folder.

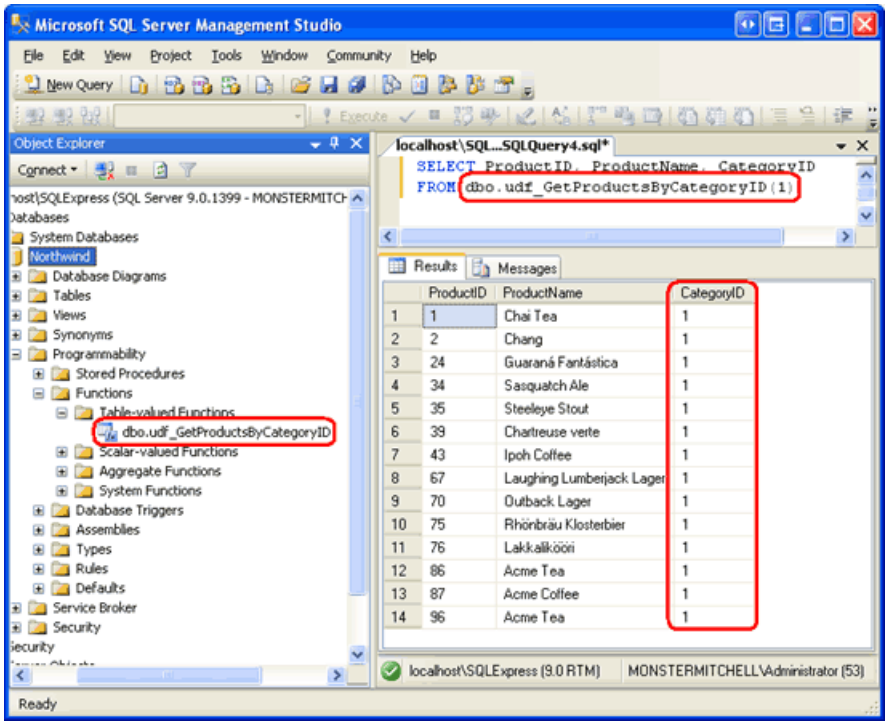


Figure 24: The `ProductID`, `ProductName`, and `CategoryID` are Listed for Each Beverage (Click to view full-size image)

> **NOTE**
>
> For more information on creating and using UDFs, check out Intro to User-Defined Functions. Also check out Advantages and Drawbacks of User-Defined Functions.

Step 10: Creating a Managed UDF

The `udf_ComputeInventoryValue` and `udf_GetProductsByCategoryID` UDFs created in the above examples are T-SQL database objects. SQL Server 2005 also supports managed UDFs, which can be added to the `ManagedDatabaseConstructs` project just like the managed stored procedures from Steps 3 and 5. For this step, let s implement the `udf_ComputeInventoryValue` UDF in managed code.

To add a managed UDF to the `ManagedDatabaseConstructs` project, right-click on the project name in Solution Explorer and choose to Add a New Item. Select the User-Defined Template from the Add New Item dialog box and name the new UDF file `udf_ComputeInventoryValue_Managed.vb`.

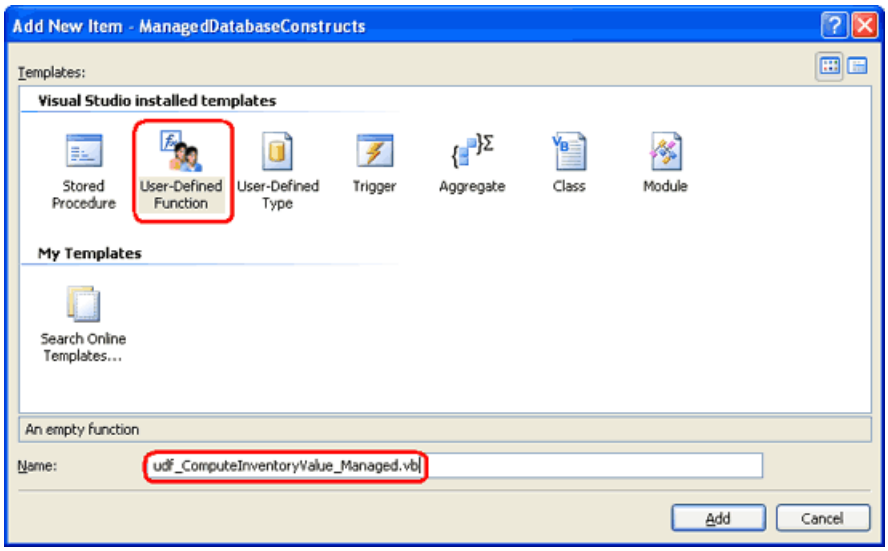


Figure 25: Add a New Managed UDF to the `ManagedDatabaseConstructs` Project (Click to view full-size image)

The User-Defined Function template creates a `Partial` class named `UserDefinedFunctions` with a method whose name is the same as the class file s name (`udf_ComputeInventoryValue_Managed` , in this instance). This method is decorated using the `SqlFunction` attribute, which flags the method as a managed UDF.

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Partial Public Class UserDefinedFunctions
    <Microsoft.SqlServer.Server.SqlFunction()> _
    Public Shared Function udf_ComputeInventoryValue_Managed() As SqlString
        ' Add your code here
        Return New SqlString("Hello")
    End Function
End Class
```

The `udf_ComputeInventoryValue` method currently returns a `SqlString` object and does not accept any input parameters. We need to update the method definition so that it accepts three input parameters - `UnitPrice` , `UnitsInStock` , and `Discontinued` - and returns a `SqlMoney` object. The logic for calculating the inventory value is identical to that in the T-SQL `udf_ComputeInventoryValue` UDF.

```
<Microsoft.SqlServer.Server.SqlFunction()> _
Public Shared Function udf_ComputeInventoryValue_Managed _
    (UnitPrice As SqlMoney, UnitsInStock As SqlInt16, Discontinued As SqlBoolean) _
    As SqlMoney
    Dim inventoryValue As SqlMoney = 0
    If Not UnitPrice.IsNull AndAlso Not UnitsInStock.IsNull Then
        inventoryValue = UnitPrice * UnitsInStock
        If Discontinued = True Then
            inventoryValue = inventoryValue * New SqlMoney(0.5)
        End If
    End If
    Return inventoryValue
End Function
```

Note that the UDF method s input parameters are of their corresponding SQL types: `SqlMoney` for the `UnitPrice` field, `SqlInt16` for `UnitsInStock` , and `SqlBoolean` for `Discontinued` . These data types reflect the types defined in the `Products` table: the `UnitPrice` column is of type `money` , the `UnitsInStock` column of type `smallint` , and the `Discontinued` column of type `bit` .

The code starts by creating a `SqlMoney` instance named `inventoryValue` that is assigned a value of 0. The `Products` table allows for database `NULL` values in the `UnitsInPrice` and `UnitsInStock` columns. Therefore, we need to first check to see if these values contain `NULL` s, which we do through the `SqlMoney` object s `IsNull` property. If both `UnitPrice` and `UnitsInStock` contain non- `NULL` values, then we compute the `inventoryValue` to be the product of the two. Then, if `Discontinued` is true, then we halve the value.

> **NOTE**
>
> The `SqlMoney` object only allows two `SqlMoney` instances to be multiplied together. It does not allow a `SqlMoney` instance to be multiplied by a literal floating-point number. Therefore, to halve `inventoryValue` we multiply it by a new `SqlMoney` instance that has the value 0.5.

Step 11: Deploying the Managed UDF

Now that that the managed UDF has been created, we are ready to deploy it to the Northwind database. As we saw in Step 4, the managed objects in a SQL Server Project are deployed by right-clicking on the project name in the Solution Explorer and choosing the Deploy option from the context menu.

Once you have deployed the project, return to SQL Server Management Studio and refresh the Scalar-valued Functions folder. You should now see two entries:

- `dbo.udf_ComputeInventoryValue` - the T-SQL UDF created in Step 9, and
- `dbo.udf ComputeInventoryValue_Managed` - the managed UDF created in Step 10 that was just deployed.

To test this managed UDF, execute the following query from within Management Studio:

```
SELECT ProductID, ProductName,
       dbo.udf_ComputeInventoryValue_Managed(
                 UnitPrice,
                 UnitsInStock,
                 Discontinued
              ) as InventoryValue
FROM Products
ORDER BY InventoryValue DESC
```

This command uses the managed `udf ComputeInventoryValue_Managed` UDF instead of the T-SQL `udf_ComputeInventoryValue` UDF, but the output is the same. Refer back to Figure 23 to see a screenshot of the UDF s output.

Step 12: Debugging the Managed Database Objects

In the Debugging Stored Procedures tutorial we discussed the three options for debugging SQL Server through Visual Studio: Direct Database Debugging, Application Debugging, and Debugging from a SQL Server Project. Managed database objects cannot be debugged via Direct Database Debugging, but can be debugged from a client application and directly from the SQL Server Project. In order for debugging to work, however, the SQL Server 2005 database must allow SQL/CLR debugging. Recall that when we first created the `ManagedDatabaseConstructs` project Visual Studio asked us whether we wanted to enable SQL/CLR debugging (see Figure 6 in Step 2). This setting can be modified by right-clicking on the database from the Server Explorer window.

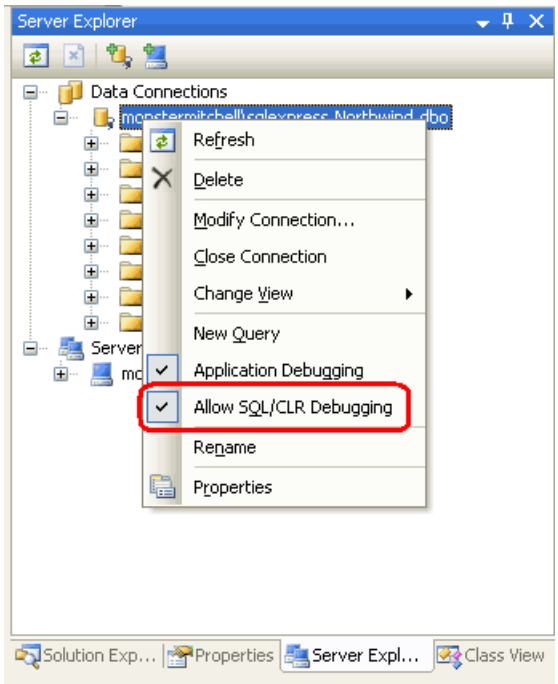


Figure 26: Ensure that the Database Allows SQL/CLR Debugging

Imagine that we wanted to debug the `GetProductsWithPriceLessThan` managed stored procedure. We would start by setting a breakpoint within the code of the `GetProductsWithPriceLessThan` method.

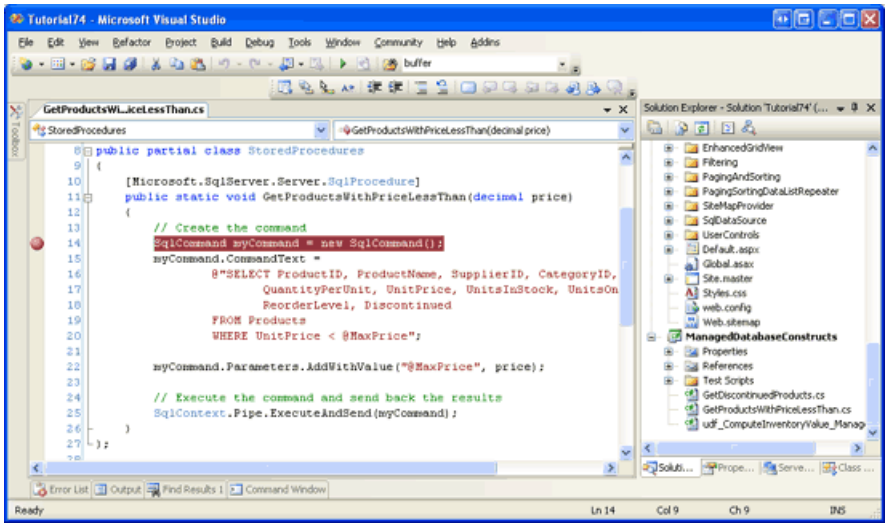


Figure 27: Set a Breakpoint in the `GetProductsWithPriceLessThan` Method (Click to view full-size image)

Let s first look at debugging the managed database objects from the SQL Server Project. Since our Solution includes two projects - the `ManagedDatabaseConstructs` SQL Server Project along with our website - in order to debug from the SQL Server Project we need to instruct Visual Studio to launch the `ManagedDatabaseConstructs` SQL Server Project when we start debugging. Right-click the `ManagedDatabaseConstructs` project in Solution Explorer and choose the Set as StartUp Project option from the context menu.

When the `ManagedDatabaseConstructs` project is launched from the debugger it executes the SQL statements in the `Test.sql` file, which is located in the `Test Scripts` folder. For example, to test the `GetProductsWithPriceLessThan` managed stored procedure, replace the existing `Test.sql` file content with the following statement, which invokes the `GetProductsWithPriceLessThan` managed stored procedure passing in the `@CategoryID` value of 14.95:

```
exec GetProductsWithPriceLessThan 14.95
```

Once you ve entered the above script into `Test.sql`, start debugging by going to the Debug menu and choosing

Start Debugging or by hitting F5 or the green play icon in the Toolbar. This will build the projects within the Solution, deploy the managed database objects to the Northwind database, and then execute the `Test.sql` script. At this point, the breakpoint will be hit and we can step through the `GetProductsWithPriceLessThan` method, examine the values of the input parameters, and so on.

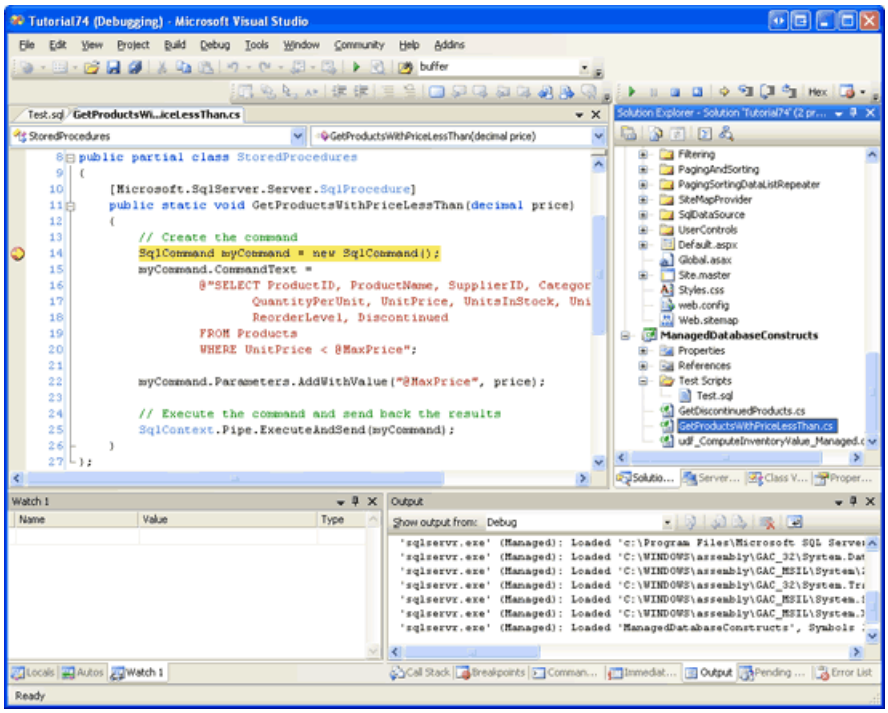


Figure 28: The Breakpoint in the `GetProductsWithPriceLessThan` Method Was Hit (Click to view full-size image)

In order for a SQL database object to be debugged through a client application, it is imperative that the database be configured to support application debugging. Right-click on the database in Server Explorer and ensure that the Application Debugging option is checked. Furthermore, we need to configure the ASP.NET application to integrate with the SQL Debugger and to disable connection pooling. These steps were discussed in detail in Step 2 of the Debugging Stored Procedures tutorial.

Once you have configured the ASP.NET application and database, set the ASP.NET website as the startup project and start debugging. If you visit a page that calls one of the managed objects that has a breakpoint, the application will halt and control will be turned over to the debugger, where you can step through the code as shown in Figure 28.

Step 13: Manually Compiling and Deploying Managed Database Objects

SQL Server Projects make it easy to create, compile, and deploy managed database objects. Unfortunately, SQL Server Projects are only available in the Professional and Team Systems editions of Visual Studio. If you are using Visual Web Developer or the Standard Edition of Visual Studio and want to use managed database objects, you will need to manually create and deploy them. This involves four steps:

1. Create a file that contains the source code for the managed database object,
2. Compile the object into an assembly,
3. Register the assembly with the SQL Server 2005 database, and
4. Create a database object in SQL Server that points to the appropriate method in the assembly.

To illustrate these tasks, let s create a new managed stored procedure that returns those products whose `UnitPrice` is greater than a specified value. Create a new file on your computer named `GetProductsWithPriceGreaterThan.vb` and enter the following code into the file (you can use Visual Studio, Notepad, or any text editor to accomplish this):

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Partial Public Class StoredProcedures
    <Microsoft.SqlServer.Server.SqlProcedure()> _
    Public Shared Sub GetProductsWithPriceGreaterThan(ByVal price As SqlMoney)
        'Create the command
        Dim myCommand As New SqlCommand()
        myCommand.CommandText = _
            "SELECT ProductID, ProductName, SupplierID, CategoryID, " & _
            "       QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder, " & _
            "       ReorderLevel, Discontinued " & _
            "FROM Products " & _
            "WHERE UnitPrice > @MinPrice"
        myCommand.Parameters.AddWithValue("@MinPrice", price)
        ' Execute the command and send back the results
        SqlContext.Pipe.ExecuteAndSend(myCommand)
    End Sub
End Class
```

This code is nearly identical to that of the `GetProductsWithPriceLessThan` method created in Step 5. The only differences are the method names, the `WHERE` clause, and the parameter name used in the query. Back in the `GetProductsWithPriceLessThan` method, the `WHERE` clause read: `WHERE UnitPrice < @MaxPrice`. Here, in `GetProductsWithPriceGreaterThan`, we use: `WHERE UnitPrice > @MinPrice`.

We now need to compile this class into an assembly. From the command line, navigate to the directory where you saved the `GetProductsWithPriceGreaterThan.vb` file and use the C# compiler (`csc.exe`) to compile the class file into an assembly:

```
vbc.exe /t:library /out:ManuallyCreatedDBObjects.dll GetProductsWithPriceGreaterThan.vb
```

If the folder containing v `bc.exe` in not in the system s `PATH`, you will have to fully reference its path, `%WINDOWS%\Microsoft.NET\Framework\version\`, like so:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\vbc.exe /t:library /out:ManuallyCreatedDBObjects.dll
GetProductsWithPriceGreaterThan.vb
```

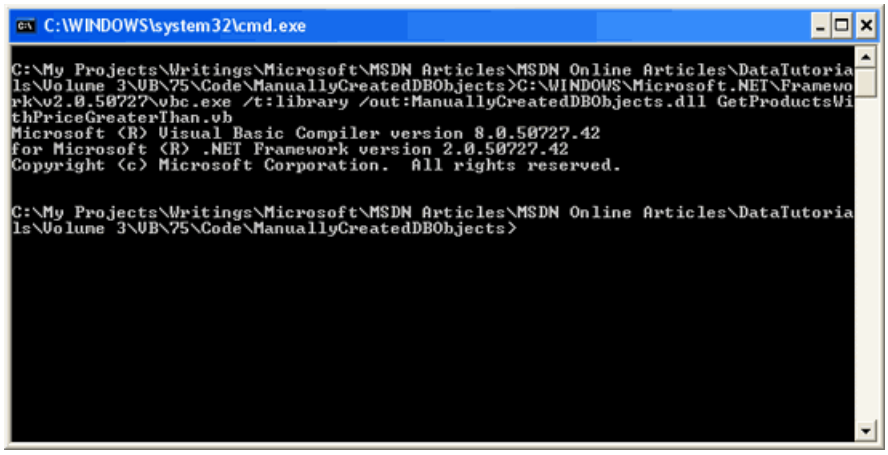


Figure 29: Compile `GetProductsWithPriceGreaterThan.vb` Into an Assembly (Click to view full-size image)

The `/t` flag specifies that the Visual Basic class file should be compiled into a DLL (rather than an executable). The `/out` flag specifies the name of the resulting assembly.

NOTE

Rather than compiling the `GetProductsWithPriceGreaterThan.vb` class file from the command line you could alternatively use Visual Basic Express Edition or create a separate Class Library project in Visual Studio Standard Edition. S ren Jacob Lauritsen has kindly provided such a Visual Basic Express Edition project with code for the `GetProductsWithPriceGreaterThan` stored procedure and the two managed stored procedures and UDF created in Steps 3, 5, and 10. S ren s project also includes the T-SQL commands needed to add the corresponding database objects.

With the code compiled into an assembly, we are ready to register the assembly within the SQL Server 2005 database. This can be performed through T-SQL, using the command `CREATE ASSEMBLY`, or through SQL Server Management Studio. Let s focus on using Management Studio.

From Management Studio, expand the Programmability folder in the Northwind database. One of its subfolder is Assemblies. To manually add a new Assembly to the database, right-click on the Assemblies folder and choose New Assembly from the context menu. This displays the New Assembly dialog box (see Figure 30). Click on the Browse button, select the `ManuallyCreatedDBObjects.dll` assembly we just compiled, and then click OK to add the Assembly to the database. You should not see the `ManuallyCreatedDBObjects.dll` assembly in the Object Explorer.

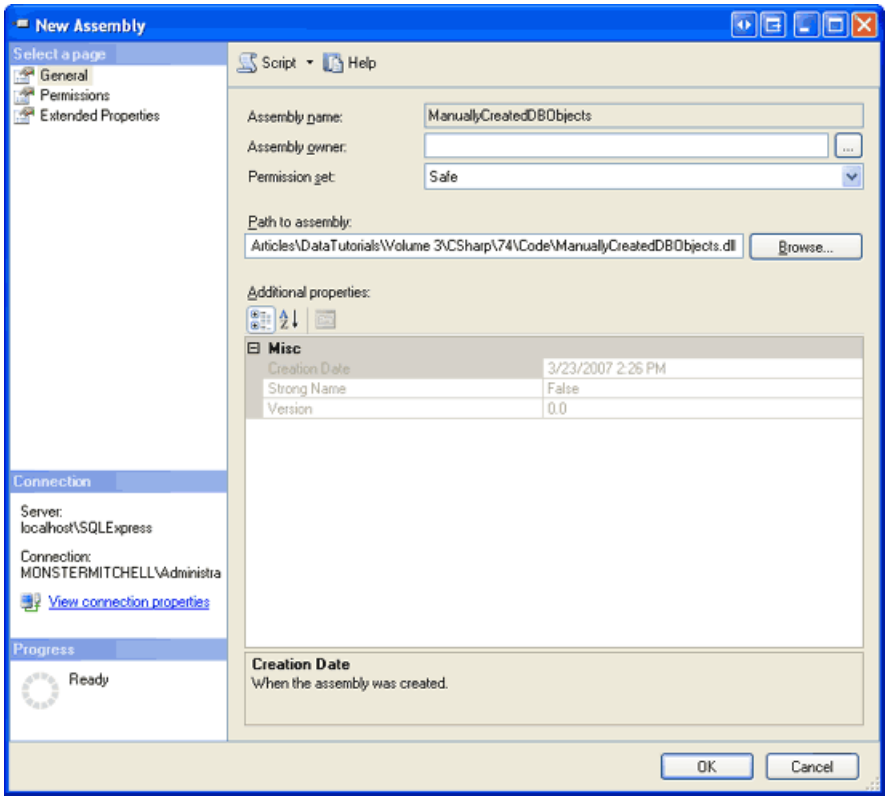


Figure 30: Add the `ManuallyCreatedDBObjects.dll` Assembly to the Database (Click to view full-size image)

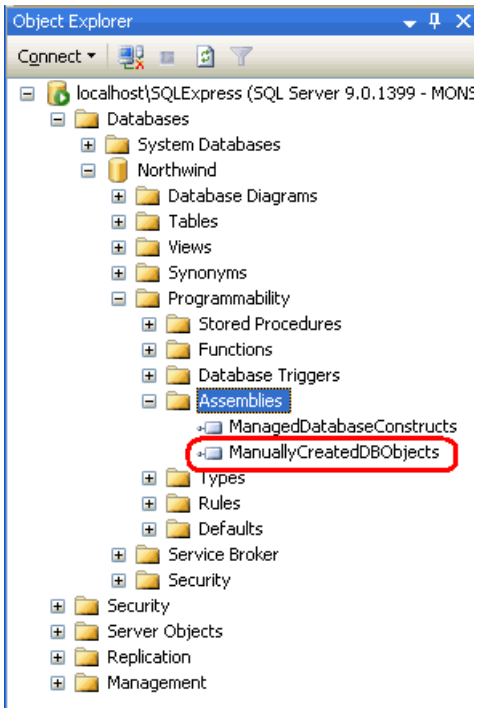


Figure 31: The `ManuallyCreatedDBObjects.dll` is Listed in the Object Explorer

While we have added the assembly to the Northwind database, we have yet to associate a stored procedure with the `GetProductsWithPriceGreaterThan` method in the assembly. To accomplish this, open a new query window and execute the following script:

```
CREATE PROCEDURE [dbo].[GetProductsWithPriceGreaterThan]
(
    @price money
)
WITH EXECUTE AS CALLER
AS
EXTERNAL NAME [ManuallyCreatedDBObjects].[StoredProcedures].[GetProductsWithPriceGreaterThan]
GO
```

This creates a new stored procedure in the Northwind database named `GetProductsWithPriceGreaterThan` and associates it with the managed method `GetProductsWithPriceGreaterThan` (which is in the class `StoredProcedures`, which is in the assembly `ManuallyCreatedDBObjects`).

After executing the above script, refresh the Stored Procedures folder in the Object Explorer. You should see a new stored procedure entry - `GetProductsWithPriceGreaterThan` - which has a lock icon next to it. To test this stored procedure, enter and execute the following script in the query window:

```
exec GetProductsWithPriceGreaterThan 24.95
```

As Figure 32 shows, the above command displays information for those products with a `UnitPrice` greater than $24.95.

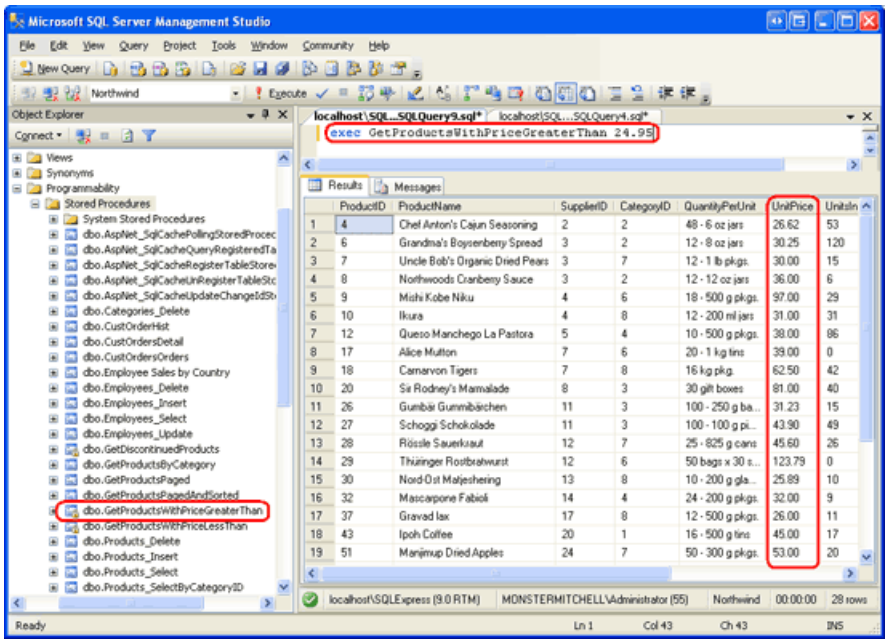


Figure 32: The `ManuallyCreatedDBObjects.dll` is Listed in the Object Explorer (Click to view full-size image)

Summary

Microsoft SQL Server 2005 provides integration with the Common Language Runtime (CLR), which allows database objects to be created using managed code. Previously, these database objects could only be created using T-SQL, but now we can create these objects using .NET programming languages like Visual Basic. In this tutorial we created two managed stored procedures and a managed User-Defined Function.

Visual Studio s SQL Server Project type facilitates creating, compiling, and deploying managed database objects. Moreover, it offers rich debugging support. However, SQL Server Project types are only available in the Professional and Team Systems editions of Visual Studio. For those using Visual Web Developer or the Standard Edition of Visual Studio, the creation, compilation, and deployment steps must be performed manually, as we saw in Step 13.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Advantages and Drawbacks of User-Defined Functions
- Creating SQL Server 2005 Objects in Managed Code
- Creating Triggers Using Managed Code in SQL Server 2005
- How To: Create and Run a CLR SQL Server Stored Procedure
- How To: Create and Run a CLR SQL Server User-Defined Function
- How To: Edit the `Test.sql` Script to Run SQL Objects
- Intro to User Defined Functions
- Managed Code and SQL Server 2005 (Video)
- Transact-SQL Reference
- Walkthrough: Creating a Stored Procedure in Managed Code

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest

book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was S ren Jacob Lauritsen. In addition to reviewing this article, S ren also created the Visual C# Express Edition project included in this article s download for manually compiling the managed database objects. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

Older Versions - Security

2/20/2020 • 2 minutes to read • Edit Online

Security, authentication, and authorization topics for ASP.NET 4 and earlier Web Forms applications.

- ASP.NET 3.5 - Introduction to Security
  - Security Basics and ASP.NET Support (C#)
  - Overview of Forms Authentication (C#)
  - Forms Authentication Configuration and Advanced Topics (C#)
  - Security Basics and ASP.NET Support (VB)
  - Overview of Forms Authentication (VB)
  - Forms Authentication Configuration and Advanced Topics (VB)
- ASP.NET 3.5 - Membership
  - Creating the Membership Schema in SQL Server (C#)
  - Creating User Accounts (C#)
  - Validating User Credentials Against the Membership User Store (C#)
  - User-Based Authorization (C#)
  - Storing Additional User Information (C#)
  - Creating the Membership Schema in SQL Server (VB)
  - Creating User Accounts (VB)
  - Validating User Credentials Against the Membership User Store (VB)
  - User-Based Authorization (VB)
  - Storing Additional User Information (VB)
- ASP.NET 3.5 - Roles
  - Creating and Managing Roles (C#)
  - Assigning Roles to Users (C#)
  - Role-Based Authorization (C#)
  - Creating and Managing Roles (VB)
  - Assigning Roles to Users (VB)
  - Role-Based Authorization (VB)
- ASP.NET 3.5 - Security Administration
  - Building an Interface to Select One User Account from Many (C#)
  - Recovering and Changing Passwords (C#)
  - Unlocking and Approving User Accounts (C#)
  - Building an Interface to Select One User Account from Many (VB)
  - Recovering and Changing Passwords (VB)
  - Unlocking and Approving User Accounts (VB)

ASP.NET 3.5 - Introduction to Security

2/20/2020 • 2 minutes to read • Edit Online

This series of tutorials explores techniques for authenticating users using a login form (forms authentication) and authorizing access to individual pages in an ASP.NET application.

- Security Basics and ASP.NET Support (C#)
- Overview of Forms Authentication (C#)
- Forms Authentication Configuration and Advanced Topics (C#)
- Security Basics and ASP.NET Support (VB)
- Overview of Forms Authentication (VB)
- Forms Authentication Configuration and Advanced Topics (VB)

Security Basics and ASP.NET Support (C#)

2/20/2020 • 16 minutes to read • Edit Online

by Scott Mitchell

Download PDF

This is the first tutorial in a series of tutorials that will explore techniques for authenticating visitors through a web form, authorizing access to particular pages and functionality, and managing user accounts in an ASP.NET application.

Introduction

What is the one thing forums, eCommerce sites, online email websites, portal websites, and social network sites all have in common? They all offer *user accounts*. Sites that offer user accounts must provide a number of services. At a minimum, new visitors need to be able to create an account and returning visitors must be able to log in. Such web applications can make decisions based on the logged in user: some pages or actions might be restricted to only logged in users, or to a certain subset of users; other pages might show information specific to the logged in user, or might show more or less information, depending on what user is viewing the page.

This is the first tutorial in a series of tutorials that will explore techniques for authenticating visitors through a web form, authorizing access to particular pages and functionality, and managing user accounts in an ASP.NET application. Over the course of these tutorials we will examine how to:

- Identify and log users in to a website
- Use ASP.NET's Membership framework to manage user accounts
- Create, update, and delete user accounts
- Limit access to a web page, directory, or specific functionality based on the logged in user
- Use ASP.NET's Roles framework to associate user accounts with roles
- Manage user roles
- Limit access to a web page, directory, or specific functionality based on the logged in user's role
- Customize and extend ASP.NET's security Web controls

These tutorials are geared to be concise and provide step-by-step instructions with plenty of screen shots to walk you through the process visually. Each tutorial is available in C# and Visual Basic versions and includes a download of the complete code used. (This first tutorial focuses on security concepts from a high-level viewpoint and therefore does not contain any associated code.)

In this tutorial we will discuss important security concepts and what facilities are available in ASP.NET to assist in implementing forms authentication, authorization, user accounts, and roles. Let's get started!

> **NOTE**
>
> Security is an important aspect of any application that spans physical, technological, and policy decisions and requires a high degree of planning and domain knowledge. This tutorial series is not intended as a guide for developing secure web applications. Rather, it focuses specifically on forms authentication, authorization, user accounts, and roles. While some security concepts revolving around these issues are discussed in this series, others are left unexplored.

Authentication, Authorization, User Accounts, and Roles

Authentication, authorization, user accounts, and roles are four terms that will be used very often throughout this tutorial series, so I'd like to take a quick moment to define these terms within the context of web security. In a client-server model, such as the Internet, there are many scenarios in which the server needs to identify the client making the request. *Authentication* is the process of ascertaining the client's identity. A client who has been successfully identified is said to be *authenticated*. An unidentified client is said to be *unauthenticated* or *anonymous*.

Secure authentication systems involve at least one of the following three facets: something you know, something you have, or something you are. Most web applications rely on something the client knows, such as a password or a PIN. The information used to identify a user - her username and password, for example - are referred to as *credentials*. This tutorial series focuses on *forms authentication*, which is an authentication model where users log in to the site by providing their credentials in a web page form. We have all experienced this type of authentication before. Go to any eCommerce site. When you are ready to check out you are asked to log in by entering your username and password into textboxes on a web page.

In addition to identifying clients, a server may need to limit what resources or functionalities are accessible depending on the client making the request. *Authorization* is the process of determining whether a particular user has the authority to access a specific resource or functionality.

A *user account* is a store for persisting information about a particular user. User accounts must minimally include information that uniquely identifies the user, such as the user's login name and password. Along with this essential information, user accounts may include things like: the user's email address; the date and time the account was created; the date and time they last logged in; first and last name; phone number; and mailing address. When using forms authentication, user account information is typically stored in a relational database like Microsoft SQL Server.

Web applications that support user accounts may optionally group users into *roles*. A role is simply a label that is applied to a user and provides an abstraction for defining authorization rules and page-level functionality. For example, a website might include an Administrator role with authorization rules that prohibit anyone but an Administrator to access a particular set of web pages. Moreover, a variety of pages that are accessible to all users (including non-Administrators) might display additional data or offer extra functionality when visited by users in the Administrators role. Using roles, we can define these authorization rules on a role-by-role basis rather than user-by-user.

Authenticating Users in an ASP.NET Application

When a user enters a URL into their browser's address window or clicks on a link, the browser makes a Hypertext Transfer Protocol (HTTP) request to the web server for the specified content, be it an ASP.NET page, an image, a JavaScript file, or any other type of content. The web server is tasked with returning the requested content. In doing so, it must determine a number of things about the request, including who made the request and whether the identity is authorized to retrieve the requested content.

By default, browsers send HTTP requests that lack any sort of identification information. But if the browser does include authentication information then the web server starts the authentication workflow, which attempts to identify the client making the request. The steps of the authentication workflow depend on the type of authentication being used by the web application. ASP.NET supports three types of authentication: Windows, Passport, and forms. This tutorial series focuses on forms authentication, but let's take a minute to compare and contrast Windows authentication user stores and workflow.

Authentication via Windows Authentication

The Windows authentication workflow uses one of the following authentication techniques:

- Basic authentication

- Digest authentication
- Windows Integrated Authentication

All three techniques work in roughly the same way: when an unauthorized, anonymous request arrives, the web server sends back an HTTP response that indicates that authorization is required to continue. The browser then displays a modal dialog box that prompts the user for their username and password (see Figure 1). This information is then sent back to the web server via an HTTP header.

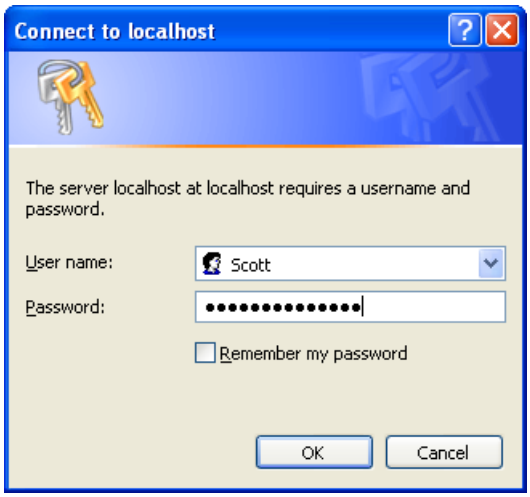


Figure 1: A Modal Dialog Box Prompts the User for His Credentials

The supplied credentials are validated against the web server's Windows User Store. This means that each authenticated user in your web application must have a Windows account in your organization. This is commonplace in intranet scenarios. In fact, when using Windows Integrated Authentication in an intranet setting, the browser automatically provides the web server with the credentials used to log on to the network, thereby suppressing the dialog box shown in Figure 1. While Windows authentication is great for intranet applications, it is usually unfeasible for Internet applications since you do not want to create Windows accounts for each and every user who signs up at your site.

Authentication via Forms Authentication

Forms authentication, on the other hand, is ideal for Internet web applications. Recall that forms authentication identifies the user by prompting them to enter their credentials through a web form. Consequently, when a user attempts to access an unauthorized resource, they are automatically redirected to the login page where they can enter their credentials. The submitted credentials are then validated against a custom user store - usually a database.

After verifying the submitted credentials, a *forms authentication ticket* is created for the user. This ticket indicates that the user has been authenticated and includes identifying information, such as the username. The forms authentication ticket is (typically) stored as a cookie on the client computer. Therefore, subsequent visits to the website include the forms authentication ticket in the HTTP request, thereby enabling the web application to identify the user once they have logged in.

Figure 2 illustrates the forms authentication workflow from a high-level vantage point. Notice how the authentication and authorization pieces in ASP.NET act as two separate entities. The forms authentication system identifies the user (or reports that they are anonymous). The authorization system is what determines whether the user has access to the requested resource. If the user is unauthorized (as they are in Figure 2 when attempting to anonymously visit ProtectedPage.aspx), the authorization system reports that the user is denied, causing the forms authentication system to automatically redirect the user to the login page.

Once the user has successfully logged in, subsequent HTTP requests include the forms authentication ticket. The forms authentication system merely identifies the user - it is the authorization system that determines whether the user can access the requested resource.

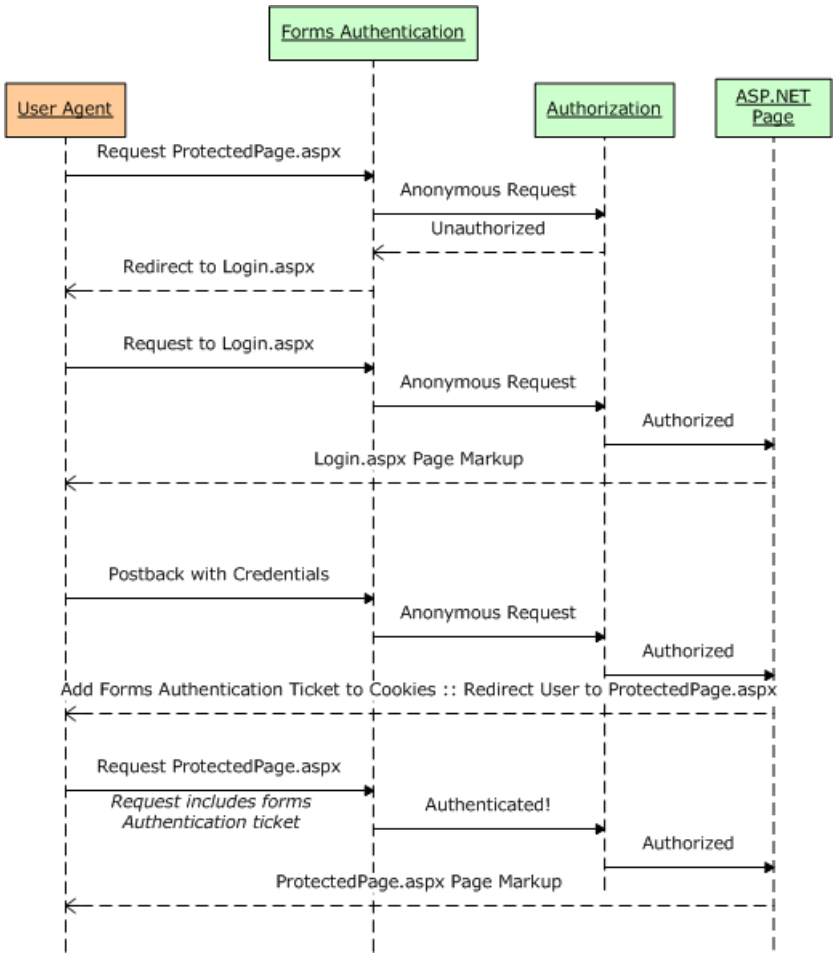


Figure 2: The Forms Authentication Workflow

We will dig into forms authentication in much greater detail in the next two tutorials,An Overview of Forms Authentication and Forms Authentication Configuration and Advanced Topics. For more on ASP.NET's authentication options, see ASP.NET Authentication.

Limiting Access to Web Pages, Directories, and Page Functionality

ASP.NET includes two ways to determine whether a particular user has authority to access a specific file or directory:

- **File authorization** - since ASP.NET pages and web services are implemented as files that reside on the web server's file system, access to these files can be specified through Access Control Lists (ACLs). File authorization is most commonly used with Windows authentication because ACLs are permissions that apply to Windows accounts. When using forms authentication, all operating system- and file system-level requests are executed by the same Windows account, regardless of the user visiting the site.
- **URL authorization**- with URL authorization, the page developer specifies authorization rules in Web.config. These authorization rules specify what users or roles are allowed to access or are denied from accessing certain pages or directories in the application.

File authorization and URL authorization define authorization rules for accessing a particular ASP.NET page or for all ASP.NET pages in a particular directory. Using these techniques we can instruct ASP.NET to deny requests to a particular page for a particular user, or allow access to a set of users and deny access to everyone else. What about scenarios where all of the users can access the page, but the page's functionality depends on the user? For example, many sites that support user accounts have pages that display different content or data for authenticated users versus anonymous users. An anonymous user might see a link to log in to the site, whereas an authenticated user would instead see a message like, Welcome back, *Username* along with a link to log out. Another example: when viewing an item at an auction site you see different information depending on whether you are a bidder or the one auctioning the item.

Such page-level adjustments can be accomplished declaratively or programmatically. To show different content for anonymous than authenticated users, simply drag a LoginView control onto your page and enter the appropriate content into its AnonymousTemplate and LoggedInTemplate templates. Alternatively, you can programmatically determine whether the current request is authenticated, who the user is, and what roles they belong to (if any). You can use this information to then show or hide columns in a grid or Panels on the page.

This series includes three tutorials that focus on authorization. ***User-Based Authorization***examines how to limit access to a page or pages in a directory for specific user accounts; ***Role-Based Authorization*** looks at supplying authorization rules at the role level; lastly, the ***Displaying Content Based on the Currently Logged In User*** tutorial explores modifying a particular page's content and functionality based on the user visiting the page. For more on ASP.NET's authorization options, see ASP.NET Authorization.

User Accounts and Roles

ASP.NET's forms authentication provides an infrastructure for users to log in to a site and have their authenticated state remembered across page visits. And URL authorization offers a framework for limiting access to specific files or folders in an ASP.NET application. Neither feature, however, supplies a means for storing user account information or managing roles.

Prior to ASP.NET 2.0, developers were responsible for creating their own user and role stores. They were also on the hook for designing the user interfaces and writing the code for essential user account-related pages like the login page and the page to create a new account, among others. Without any built-in user account framework in ASP.NET, each developer implementing user accounts had to arrive at his own design decisions on questions like, How do I store passwords or other sensitive information? and What guidelines should I impose regarding password length and strength?

Today, implementing user accounts in an ASP.NET application is much simpler thanks to the *Membership framework* and the built-in Login Web controls. The Membership framework is a handful of classes in the System.Web.Security namespace that provide functionality for performing essential user account-related tasks. The key class in the Membership framework is the Membership class, which has methods like:

- CreateUser
- DeleteUser
- GetAllUsers
- GetUser
- UpdateUser
- ValidateUser

The Membership framework uses the provider model, which cleanly separates the Membership framework's API from its implementation. This enables developers to use a common API, but empowers them to use an implementation that meets their application's custom needs. In short, the Membership class defines the essential functionality of the framework (the methods, properties, and events), but does not actually supply any implementation details. Instead, the methods of the Membership class invoke the configured provider, which is what performs the actual work. For example, when the Membership class's CreateUser method is invoked, the Membership class doesn't know the details of the user store. It doesn't know if users are being maintained in a database or in an XML file or in some other store. The Membership class examines the web application's configuration to determine what provider to delegate the call to, and that provider class is responsible for actually creating the new user account in the appropriate user store. This interaction is illustrated in Figure 3.

Microsoft ships two Membership provider classes in the .NET Framework:

- ActiveDirectoryMembershipProvider - implements the Membership API in Active Directory and Active Directory Application Mode (ADAM) servers.
- SqlMembershipProvider - implements the Membership API in a SQL Server database.

This tutorial series focuses exclusively on the SqlMembershipProvider.

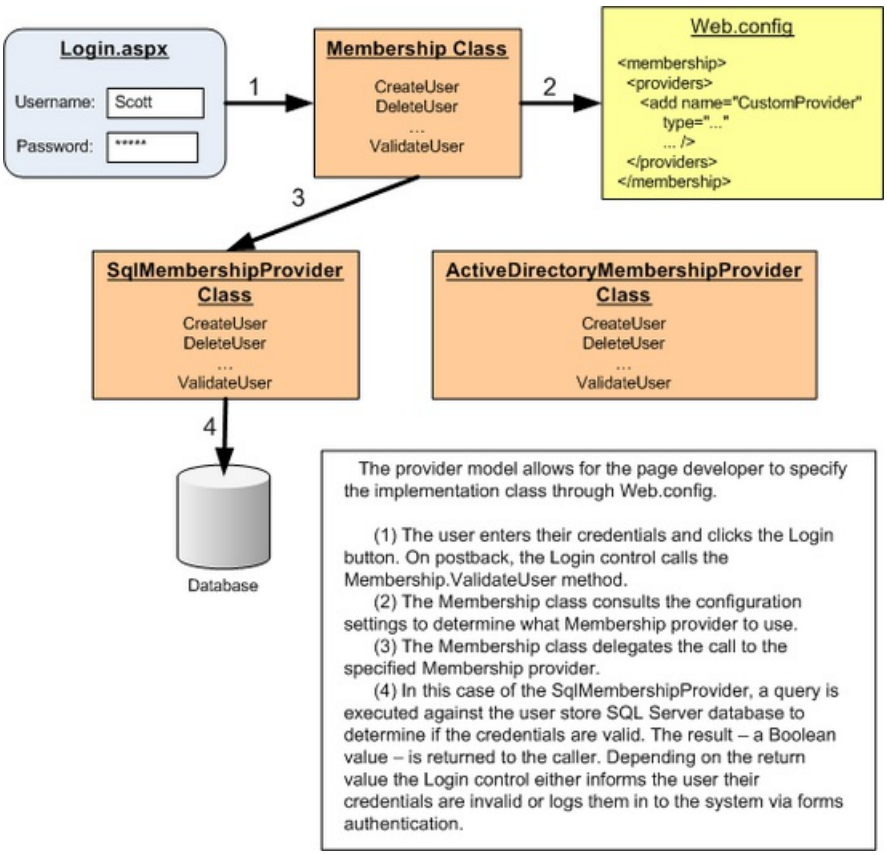


Figure 03: The Provider Model Enables Different Implementations to be Seamlessly Plugged Into the Framework (Click to view full-size image)

The benefit of the provider model is that alternative implementations can be developed by Microsoft, third-party vendors, or individual developers and seamlessly plugged into the Membership framework. For example, Microsoft has released a Membership provider for Microsoft Access databases. For more information on the Membership providers, refer to the Provider Toolkit, which includes a walkthrough of the Membership providers, sample custom providers, over 100 pages of documentation on the provider model, and the complete source code for the built-in Membership providers (namely, ActiveDirectoryMembershipProvider and SqlMembershipProvider).

ASP.NET 2.0 also introduced the Roles framework. Like the Membership framework, the Roles framework is built atop the provider model. Its API is exposed via the Roles class and the .NET Framework ships with three provider classes:

- AuthorizationStoreRoleProvider - manages role information in an authorization-manager policy store, such as Active Directory or ADAM.
- SqlRoleProvider - implements roles in a SQL Server database.
- WindowsTokenRoleProvider - associates role information based on the visitor's Windows group. This method is typically used with Windows authentication.

This tutorial series focuses exclusively on the SqlRoleProvider.

Since the provider model includes a single forward-facing API (the Membership and Roles classes), it is possible to build functionality around that API without having to worry about the implementation details - those are handled by the providers selected by the page developer. This unified API allows for Microsoft and third-party vendors to build Web controls that interface with the Membership and Roles frameworks.

ASP.NET ships with a number of Login Web controls for implementing common user account user interfaces. For example, the Login control prompts a user for their credentials, validates them, and then logs them in via forms authentication. The LoginView control offers templates for displaying different markup to anonymous users versus authenticated users, or different markup based on the user's role. And the CreateUserWizard control provides a step-by-step user interface for creating a new user account.

Underneath the covers the various Login controls interact with the Membership and Roles frameworks. Most Login controls can be implemented without having to write a single line of code. We will examine these controls in greater detail in future tutorials, including techniques for extending and customizing their functionality.

Summary

All web applications that support user accounts require similar features, such as: the ability for users to log in and have their log in status remembered across page visits; a web page for new visitors to create an account; and the ability to the page developer to specify what resource, data, and functionality are available to what users or roles. The tasks of authenticating and authorizing users and of managing user accounts and roles is remarkably easy to accomplish in ASP.NET applications thanks to forms authentication, URL authorization, and the Membership and Roles frameworks.

Over the course of the next several tutorials we will examine these aspects by building a working web application from the ground up in a step-by-step fashion. In the next two tutorial we will explore forms authentication in detail. We will see the forms authentication workflow in action, dissect the forms authentication ticket, discuss security concerns, and see how to configure the forms authentication system - all while building a web application that allows visitors to log in and log out.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET 2.0 Membership, Roles, Forms Authentication, and Security Resources
- ASP.NET 2.0 Security Guidelines
- ASP.NET Authentication
- ASP.NET Authorization
- ASP.NET Login Controls Overview
- Examining ASP.NET 2.0's Membership, Roles, and Profile
- How Do I: Secure My Site Using Membership and Roles? (Video)
- Introduction to Membership
- MSDN Security Developer Center
- Professional ASP.NET 2.0 Security, Membership, and Role Management (ISBN: 978-0-7645-9698-8)
- Provider Toolkit

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was This

tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial include Alicja Maziarz, John Suru, and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

An Overview of Forms Authentication (C#)

2/20/2020 • 36 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> In this tutorial we will turn from mere discussion to implementation; in particular, we will look at implementing forms authentication. The web application we start constructing in this tutorial will continue to be built upon in subsequent tutorials, as we move from simple forms authentication to membership and roles.
>
> Please see this video for more information on this topic: Using Basic Forms Authentication in ASP.NET.

Introduction

In the preceding tutorial we discussed the various authentication, authorization, and user account options provided by ASP.NET. In this tutorial we will turn from mere discussion to implementation; in particular, we will look at implementing forms authentication. The web application we start constructing in this tutorial will continue to be built upon in subsequent tutorials, as we move from simple forms authentication to membership and roles.

This tutorial begins with an in-depth look at the forms authentication workflow, a topic we touched upon in the previous tutorial. Following that, we will create an ASP.NET website through which to demo the concepts of forms authentication. Next, we will configure the site to use forms authentication, create a simple login page, and see how to determine, in code, whether a user is authenticated and, if so, the username they logged in with.

Understanding the forms authentication workflow, enabling it in a web application, and creating the login and logoff pages are all vital steps in building an ASP.NET application that supports user accounts and authenticates users through a web page. Because of this – and because these tutorials build upon one another - I would encourage you to work through this tutorial in full before moving on to the next one even if you already have had experience configuring forms authentication in past projects.

Understanding the Forms Authentication Workflow

When the ASP.NET runtime processes a request for an ASP.NET resource, such as an ASP.NET page or ASP.NET Web service, the request raises a number of events during its lifecycle. There are events raised at the very beginning and very end of the request, ones raised when the request is being authenticated and authorized, an event raised in the case of an unhandled exception, and so forth. To see a complete listing of the events, refer to the HttpApplication object's events.

HTTP Modules are managed classes whose code is executed in response to a particular event in the request lifecycle. ASP.NET ships with a number of HTTP Modules that perform essential tasks behind the scenes. Two built-in HTTP Modules that are especially relevant to our discussion are:

- `FormsAuthenticationModule` – authenticates the user by inspecting the forms authentication ticket, which is typically included in the user's cookies collection. If no forms authentication ticket is present, the user is anonymous.
- `UrlAuthorizationModule` – determines whether or not the current user is authorized to access the requested URL. This module determines the authority by consulting the authorization rules specified in the application's configuration files. ASP.NET also includes the `FileAuthorizationModule` that determines authority by

consulting the requested file(s) ACLs.

The `FormsAuthenticationModule` attempts to authenticate the user prior to the `UrlAuthorizationModule` (and `FileAuthorizationModule`) executing. If the user making the request is not authorized to access the requested resource, the authorization module terminates the request and returns an HTTP 401 Unauthorized status. In Windows authentication scenarios, the HTTP 401 status is returned to the browser. This status code causes the browser to prompt the user for their credentials via a modal dialog box. With forms authentication, however, the HTTP 401 Unauthorized status is never sent to the browser because the FormsAuthenticationModule detects this status and modifies it to redirect the user to the login page instead (via an HTTP 302 Redirect status).

The login page's responsibility is to determine if the user's credentials are valid and, if so, to create a forms authentication ticket and redirect the user back to the page they were attempting to visit. The authentication ticket is included in subsequent requests to the pages on the website, which the `FormsAuthenticationModule` uses to identify the user.

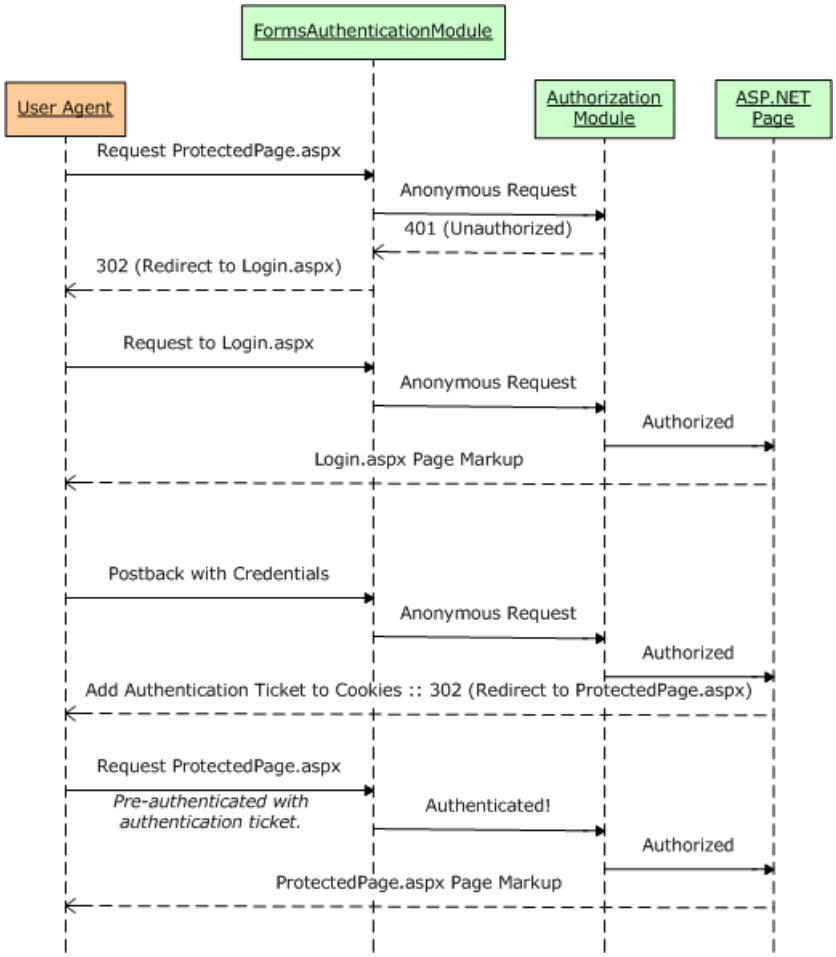


Figure 1: The Forms Authentication Workflow

Remembering the Authentication Ticket Across Page Visits

After logging in, the forms authentication ticket must be sent back to the web server on each request so that the user remains logged in as they browse the site. This is typically accomplished by placing the authentication ticket in the user's cookies collection. Cookies are small text files that reside on the user's computer and are transmitted in the HTTP headers on each request to the website that created the cookie. Therefore, once the forms authentication ticket has been created and stored in the browser's cookies, each subsequent visit to that site sends the authentication ticket along with the request, thereby identifying the user.

One aspect of cookies is their expiration, which is the date and time at which the browser discards the cookie. When the forms authentication cookie expires, the user can no longer be authenticated and therefore become anonymous. When a user is visiting from a public terminal, chances are they want their authentication ticket to expire when they close their browser. When visiting from home, however, that same user might want the

authentication ticket to be remembered across browser restarts so that they do not have to re-log in each time they visit the site. This decision is often made by the user in the form of a "Remember me" checkbox on the login page. In Step 3 we will examine how to implement a "Remember me" checkbox in the login page. The following tutorial addresses the authentication ticket timeout settings in detail.

> **NOTE**
> It is possible that the user agent used to log on to the website may not support cookies. In such a case, ASP.NET can use cookieless forms authentication tickets. In this mode, the authentication ticket is encoded into the URL. We will look at when cookieless authentication tickets are used and how they are created and managed in the next tutorial.

The Scope of Forms Authentication

The `FormsAuthenticationModule` is managed code that is part of the ASP.NET runtime. Prior to version 7 of Microsoft's Internet Information Services (IIS) web server, there was a distinct barrier between IIS's HTTP pipeline and the ASP.NET runtime's pipeline. In short, in IIS 6 and earlier, the `FormsAuthenticationModule` only executes when a request is delegated from IIS to the ASP.NET runtime. By default, IIS processes static content itself – like HTML pages and CSS and image files – and only hands off requests to the ASP.NET runtime when a page with an extension of .aspx, .asmx, or .ashx is requested.

IIS 7, however, allows for integrated IIS and ASP.NET pipelines. With a few configuration settings you can setup IIS 7 to invoke the FormsAuthenticationModule for *all* requests. Furthermore, with IIS 7 you can define URL authorization rules for files of any type. For more information, see Changes Between IIS6 and IIS7 Security, Your Web Platform Security, and Understanding IIS7 URL Authorization.

Long story short, in versions prior to IIS 7, you can only use forms authentication to protect resources handled by the ASP.NET runtime. Likewise, URL authorization rules are only applied to resources handled by the ASP.NET runtime. But with IIS 7 it is possible to integrate the FormsAuthenticationModule and UrlAuthorizationModule into IIS's HTTP pipeline, thereby extending this functionality to all requests.

Step 1: Creating an ASP.NET Website for this Tutorial Series

In order to reach the widest possible audience, the ASP.NET website we will be building throughout this series will be created with Microsoft's free version of Visual Studio 2008, Visual Web Developer 2008. We will implement the `SqlMembershipProvider` user store in a Microsoft SQL Server 2005 Express Edition database. If you are using Visual Studio 2005 or a different edition of Visual Studio 2008 or SQL Server, don't worry - the steps will be nearly identical and any non-trivial differences will be pointed out.

> **NOTE**
> The demo web application used in each tutorial is available as a download. This downloadable application was created with Visual Web Developer 2008 targeted for the .NET Framework version 3.5. Since the application is targeted for .NET 3.5, its Web.config file includes additional, 3.5-specific configuration elements. Long story short, if you have yet to install .NET 3.5 on your computer then the downloadable web application will not work without first removing the 3.5-specific markup from Web.config.

Before we can configure forms authentication, we first need an ASP.NET website. Start by creating a new file system-based ASP.NET website. To accomplish this, launch Visual Web Developer and then go to the File menu and choose New Web Site, displaying the New Web Site dialog box. Choose the ASP.NET Web Site template, set the Location drop-down list to File System, choose a folder to place the web site, and set the language to C#. This will create a new web site with a Default.aspx ASP.NET page, an App_Data folder, and a Web.config file.

NOTE

Visual Studio supports two modes of project management: Web Site Projects and Web Application Projects. Web Site Projects lack a project file, whereas Web Application Projects mimic the project architecture in Visual Studio .NET 2002/2003 – they include a project file and compile the project's source code into a single assembly, which is placed in the /bin folder. Visual Studio 2005 initially only supported Web Site Projects, although the Web Application Project model was reintroduced with Service Pack 1; Visual Studio 2008 offers both project models. The Visual Web Developer 2005 and 2008 editions, however, only support Web Site Projects. I will be using the Web Site Project model. If you are using a non-Express edition and want to use the Web Application Project model instead, feel free to do so but be aware that there may be some discrepancies between what you see on your screen and the steps you must take versus the screen shots shown and instructions provided in these tutorials.

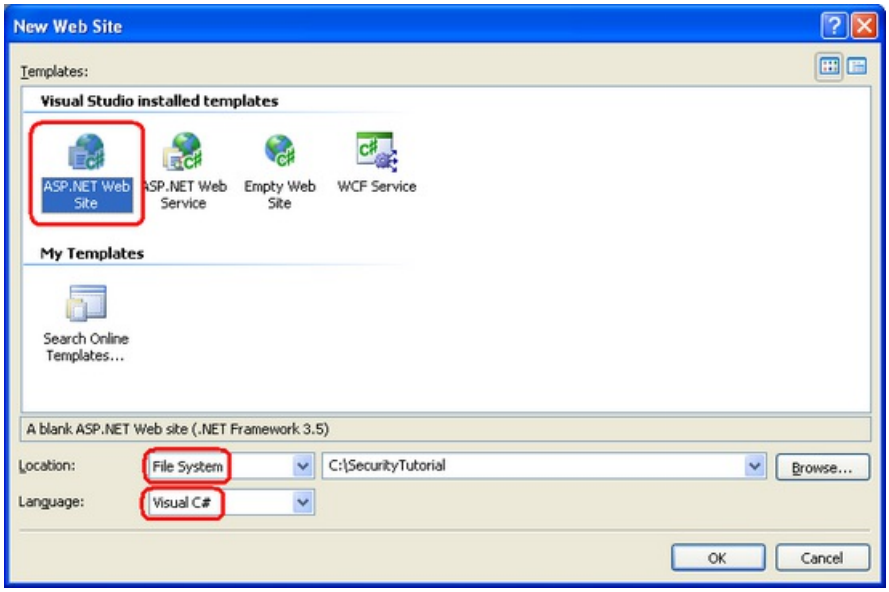


Figure 2: Create a New File System-Based Web Site (Click to view full-size image)

Adding a Master Page

Next, add a new Master Page to the site in the root directory named Site.master. Master pages enable a page developer to define a site-wide template that can be applied to ASP.NET pages. The main benefit of master pages is that the site's overall appearance can be defined in a single location, thereby making it easy to update or tweak the site's layout.

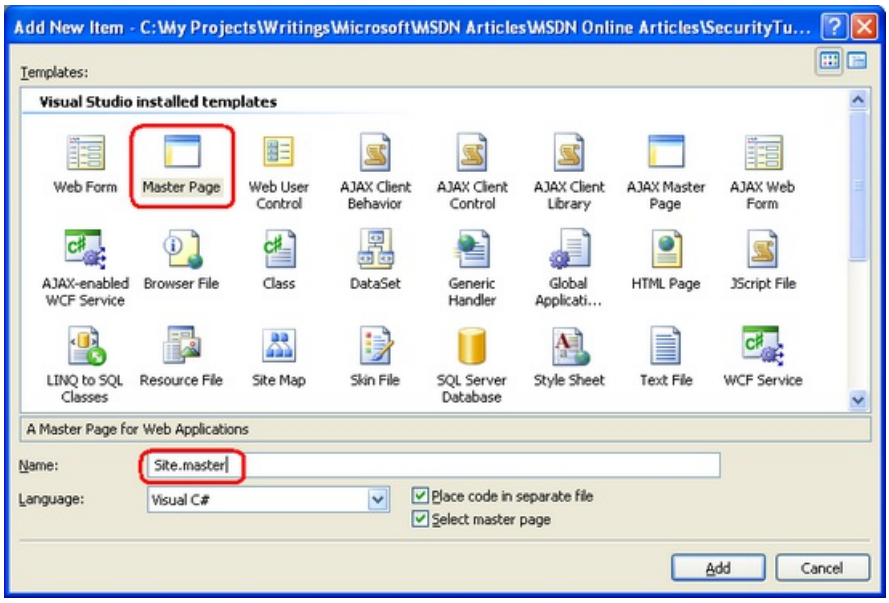


Figure 3: Add a Master Page Named Site.master to the Website (Click to view full-size image)

Define the site-wide page layout here in the master page. You can use the Design view and add whatever Layout or Web controls you need, or you can manually add the markup by hand in the Source view. I structured my master page's layout to mimic the layout used in my *Working with Data in ASP.NET 2.0* tutorial series (see Figure 4). The master page uses cascading style sheets for positioning and styles with the CSS settings defined in the file Style.css (which is included in this tutorial's associated download). While you cannot tell from the markup shown below, the CSS rules are defined such that the navigation 's content is absolutely positioned so that it appears on the left and has a fixed width of 200 pixels.

```
<%@ Master Language="C#" AutoEventWireup="true" CodeFile="Site.master.cs" Inherits="Site" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Forms Authentication, Authorization, and User Accounts</title>
    <link href="Styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <div id="wrapper">
        <form id="form1" runat="server">

            <div id="header">
                <span class="title">User Account Tutorials</span>
            </div>

            <div id="content">
                <asp:contentplaceholder id="MainContent" runat="server">
                  <!-- Page-specific content will go here... -->
                </asp:contentplaceholder>
            </div>

            <div id="navigation">
                TODO: Menu will go here...
            </div>
        </form>
    </div>
</body>
</html>
```

A master page defines both the static page layout and the regions that can be edited by the ASP.NET pages that use the master page. These content editable regions are indicated by the `ContentPlaceHolder` control, which can be seen within the content . Our master page has a single `ContentPlaceHolder` (MainContent), but master page's may have multiple ContentPlaceHolders.

With the markup entered above, switching to the Design view shows the master page's layout. Any ASP.NET pages that use this master page will have this uniform layout, with the ability to specify the markup for the `MainContent` region.

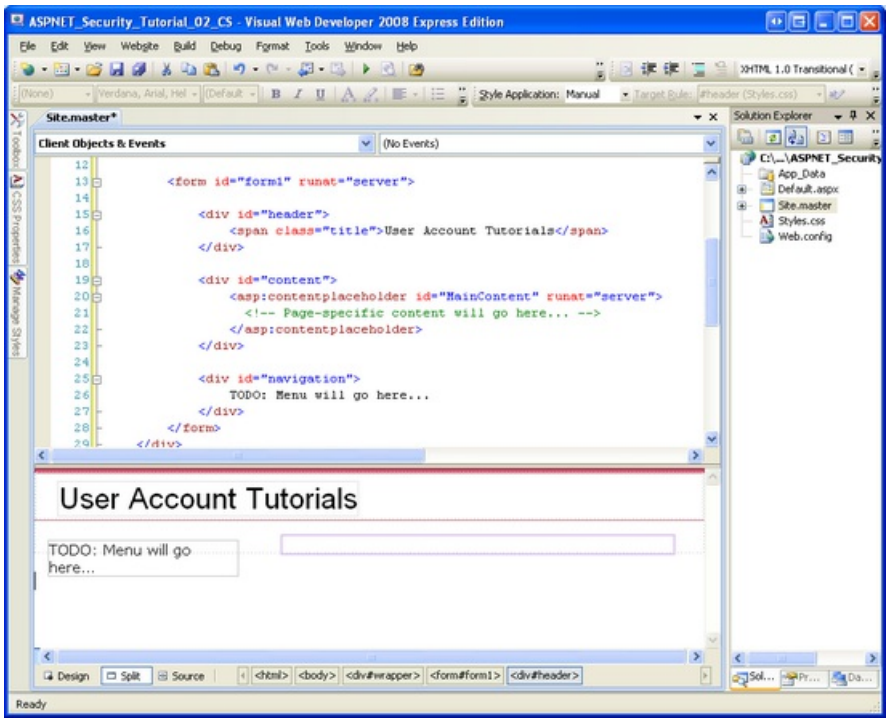


Figure 4: The Master Page, When Viewed Through the Design View (Click to view full-size image)

Creating Content Pages

At this point we have a Default.aspx page in our website, but it does not use the master page we just created. While it is possible to manipulate the declarative markup of a web page to use a master page, if the page doesn't contain any content yet it is easier to just delete the page and re-add it to the project, specifying the master page to use. Therefore, start by deleting Default.aspx from the project.

Next, right-click on the project name in the Solution Explorer and choose to add a new Web Form named Default.aspx. This time, check the "Select master page" checkbox and choose the Site.master master page from the list.

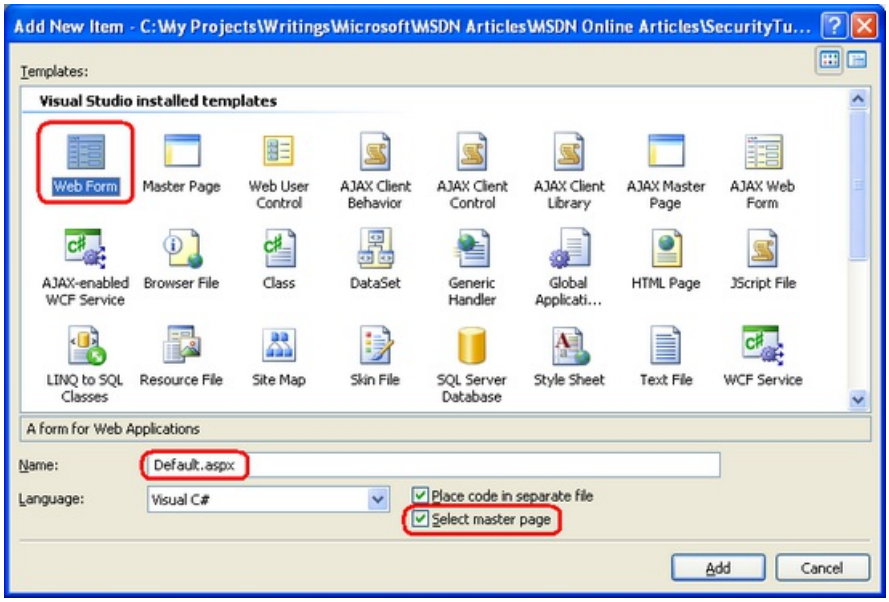


Figure 5: Add a New Default.aspx Page Choosing to Select a Master Page (Click to view full-size image)

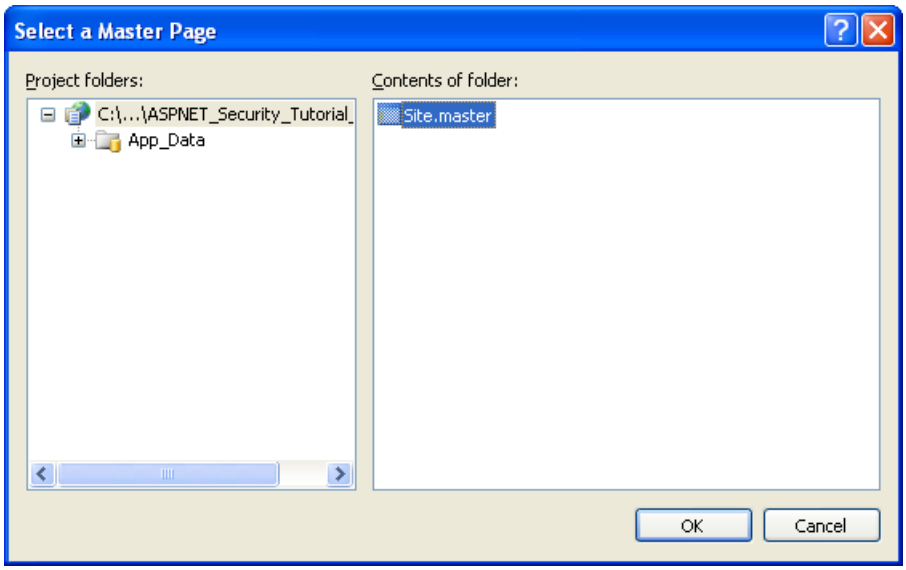


Figure 6: Use the Site.master Master Page

> **NOTE**
> If you are using the Web Application Project Model the Add New Item dialog box does not include a "Select master page" checkbox. Instead, you need to add an item of type "Web Content Form." After choosing the "Web Content Form" option and clicking Add, Visual Studio will display the same Select a Master dialog box shown in Figure 6.

The new Default.aspx page's declarative markup includes just a @Page directive specifying the path to the master page file and a Content control for the master page's MainContent ContentPlaceHolder.

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" Runat="Server">
</asp:Content>
```

For now, leave Default.aspx empty. We will return to it later in this tutorial to add content.

> **NOTE**
> Our master page includes a section for a menu or some other navigation interface. We will create such an interface in a future tutorial.

Step 2: Enabling Forms Authentication

With the ASP.NET website created, our next task is to enable forms authentication. The application's authentication configuration is specified through the `<authentication>` element in Web.config. The `<authentication>` element contains a single attribute named mode that specifies the authentication model used by the application. This attribute can have one of the following four values:

- **Windows** – as discussed in the preceding tutorial, when an application uses Windows authentication it is the web server's responsibility to authenticate the visitor, and this is usually done through Basic, Digest, or Integrated Windows authentication.
- **Forms**– users are authenticated via a form on a web page.
- **Passport**– users are authenticated using Microsoft's Passport Network.
- **None**– no authentication model is used; all visitors are anonymous.

By default, ASP.NET applications use Windows authentication. To change the authentication type to forms

authentication, then, we need to modify the `<authentication>` element's mode attribute to Forms.

If your project does not yet contain a Web.config file, add one now by right-clicking on the project name in the Solution Explorer, choosing Add New Item, and then adding a Web Configuration file.

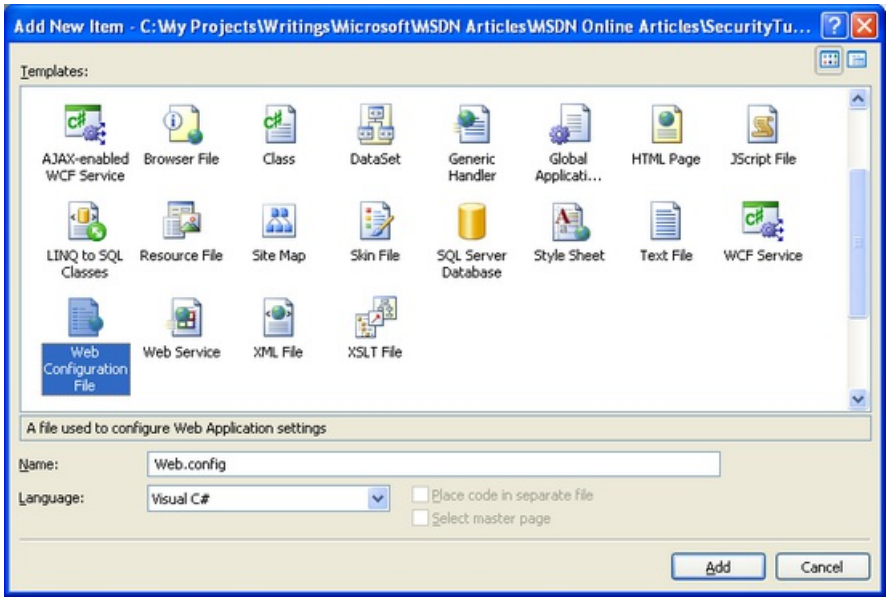


Figure 7: If Your Project Does Not Yet Include Web.config, Add It Now (Click to view full-size image)

Next, locate the `<authentication>` element and update it to use forms authentication. After this change, your Web.config file's markup should look similar to the following:

```
<configuration>
    <system.web>
        ... Unrelated configuration settings and comments removed for brevity ...
        <!--
            The <authentication> section enables configuration
            of the security authentication mode used by
            ASP.NET to identify an incoming user.
        -->
        <authentication mode="Forms" />
    </system.web>
</configuration>
```

> **NOTE**
> Since Web.config is an XML file, casing is important. Make sure that you set the mode attribute to Forms, with a capital "F". If you use a different casing, such as "forms", you'll receive a configuration error when visiting the site through a browser.

The `<authentication>` element may optionally include a `<forms>` child element that contains forms authentication-specific settings. For now, let's just use the default forms authentication settings. We will explore the `<forms>` child element in more detail in the next tutorial.

Step 3: Building the Login Page

In order to support forms authentication our website needs a login page. As discussed in the "Understanding the Forms Authentication Workflow" section, the `FormsAuthenticationModule` will automatically redirect the user to the login page if they attempt to access a page that they are not authorized to view. There are also ASP.NET Web controls that will display a link to the login page to anonymous users. This begs the question, "What is the URL of the login page?"

By default, the forms authentication system expects the login page to be named Login.aspx and placed in the root directory of the web application. If you want to use a different login page URL, you can do so by specifying it in Web.config. We will see how to do this in the subsequent tutorial.

The login page has three responsibilities:

1. Provide an interface that allows the visitor to enter their credentials.
2. Determine if the submitted credentials are valid.
3. "Log in" the user by creating the forms authentication ticket.

Creating the Login Page's User Interface

Let's get started with the first task. Add a new ASP.NET page to the site's root directory named Login.aspx and associate it with the Site.master master page.

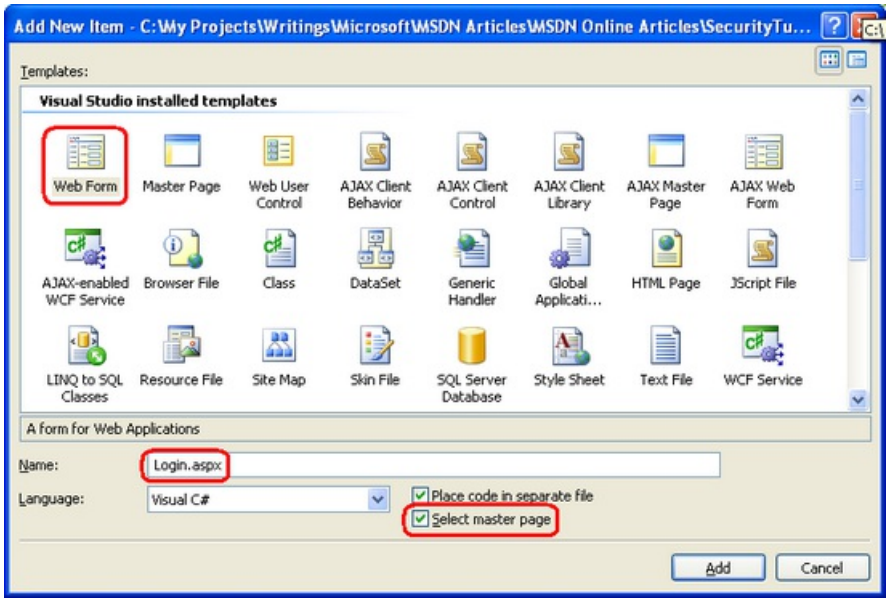


Figure 8: Add a New ASP.NET Page Named Login.aspx (Click to view full-size image)

The typical login page interface consists of two textboxes – one for the user's name, one for their password – and a button to submit the form. Websites oftentimes include a "Remember me" checkbox that, if checked, persists the resulting authentication ticket across browser restarts.

Add two TextBoxes to Login.aspx and set their `ID` properties to UserName and Password, respectively. Also set Password's `TextMode` property to Password. Next, add a CheckBox control, setting its `ID` property to RememberMe and its `Text` property to "Remember Me". Following that, add a Button named LoginButton whose `Text` property is set to "Login". And finally, add a Label Web control and set its `ID` property to InvalidCredentialsMessage, its `Text` property to "Your username or password is invalid. Please try again.", its `ForeColor` property to Red, and its `Visible` property to False.

At this point your screen should look similar to the screen shot in Figure 9, and your page's declarative syntax should like the following:

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true" CodeFile="Login.aspx.cs"
Inherits="Login" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" Runat="Server">
    <h1>
        Login</h1>
    <p>
        Username:
        <asp:TextBox ID="UserName" runat="server"></asp:TextBox></p>
    <p>
        Password:
        <asp:TextBox ID="Password" runat="server" TextMode="Password"></asp:TextBox></p>
    <p>
        <asp:CheckBox ID="RememberMe" runat="server" Text="Remember Me" /> </p>
    <p>
        <asp:Button ID="LoginButton" runat="server" Text="Login" OnClick="LoginButton_Click" /> </p>
    <p>
        <asp:Label ID="InvalidCredentialsMessage" runat="server" ForeColor="Red" Text="Your username or
password is invalid. Please try again."
            Visible="False"></asp:Label> </p>
</asp:Content>
```

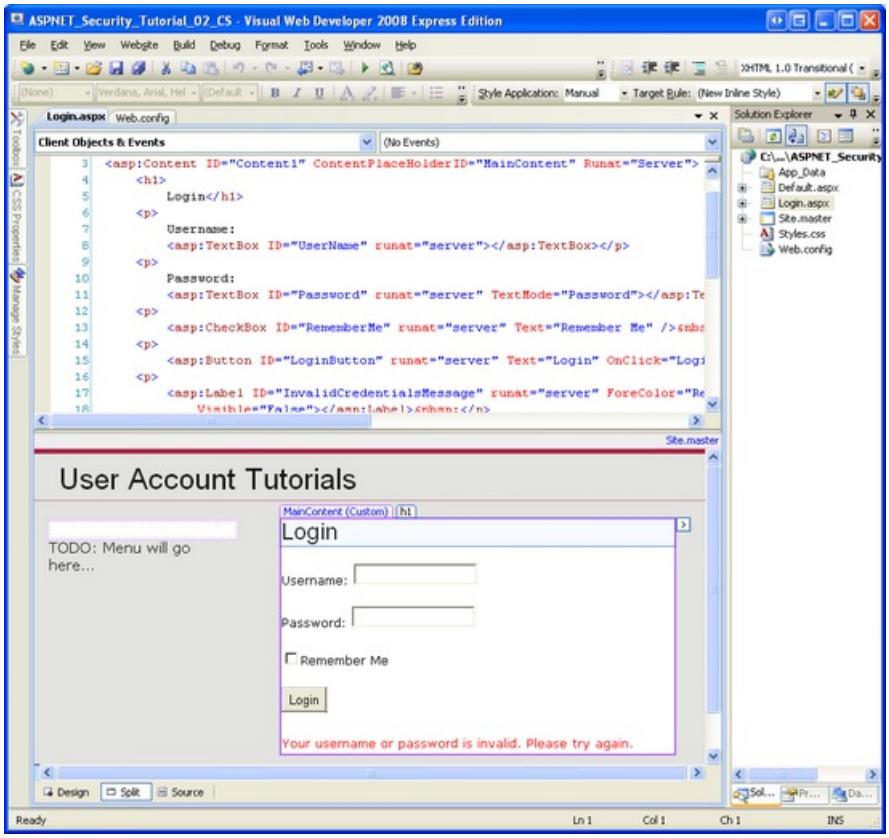


Figure 9: The Login Page Contains Two TextBoxes, a CheckBox, a Button, and a Label (Click to view full-size image)

Finally, create an event handler for the LoginButton's Click event. From the Designer, simply double-click the Button control to create this event handler.

Determining If the Supplied Credentials are Valid

We now need to implement task 2 in the Button's Click event handler – determining whether the supplied credentials are valid. In order to do this there needs to be a user store that holds all of the users' credentials so that we can determine if the supplied credentials match up with any known credentials.

Prior to ASP.NET 2.0, developers were responsible for implementing both their own user stores and writing the code to validate the supplied credentials against the store. Most developers would implement the user store in a database, creating a table named Users with columns like UserName, Password, Email, LastLoginDate, and so

forth. This table, then, would have one record per user account. Verifying a user's supplied credentials would involve querying the database for a matching username and then ensuring that the password in the database corresponded to the supplied password.

With ASP.NET 2.0, developers should use one of the Membership providers to manage the user store. In this tutorial series we will be using the SqlMembershipProvider, which uses a SQL Server database for the user store. When using the SqlMembershipProvider we need to implement a specific database schema that includes the tables, views, and stored procedures expected by the provider. We will examine how to implement this schema in the ***Creating the Membership Schema in SQL Server*** tutorial. With the Membership provider in place, validating the user's credentials is as simple as calling the Membership class's ValidateUser(*username*, *password*) method, which returns a Boolean value indicating whether the validity of the *username* and *password* combination. Seeing as we have not yet implemented the SqlMembershipProvider's user store, we cannot use the Membership class's ValidateUser method at this time.

Rather than take the time to build our own custom Users database table (which would be obsolete once we implemented the SqlMembershipProvider), let's instead hard-code the valid credentials within the login page itself. In the LoginButton's Click event handler, add the following code:

```
protected void LoginButton_Click(object sender, EventArgs e)
{
    // Three valid username/password pairs: Scott/password, Jisun/password, and Sam/password.
    string[] users = { "Scott", "Jisun", "Sam" };
    string[] passwords = { "password", "password", "password" };
    for (int i = 0; i < users.Length; i++)
    {
        bool validUsername = (string.Compare(UserName.Text, users[i], true) == 0);
        bool validPassword = (string.Compare(Password.Text, passwords[i], false) == 0);
        if (validUsername && validPassword)
        {
            // TODO: Log in the user...
            // TODO: Redirect them to the appropriate page
        }
    }
    // If we reach here, the user's credentials were invalid
    InvalidCredentialsMessage.Visible = true;
}
```

As you can see, there are three valid user accounts – Scott, Jisun, and Sam – and all three have the same password ("password"). The code loops through the users and passwords arrays looking for a valid username and password match. If both the username and password are valid, we need to login the user and then redirect them to the appropriate page. If the credentials are invalid, then we display the InvalidCredentialsMessage Label.

When a user enters valid credentials, I mentioned that they are then redirected to the "appropriate page." What is the appropriate page, though? Recall that when a user visits a page they are not authorized to view, the FormsAuthenticationModule automatically redirects them to the login page. In doing so, it includes the requested URL in the querystring via the ReturnUrl parameter. That is, if a user attempted to visit ProtectedPage.aspx, and they were not authorized to do so, the FormsAuthenticationModule would redirect them to:

Login.aspx?ReturnUrl=ProtectedPage.aspx

Upon successfully logging in, the user should be redirected back to ProtectedPage.aspx. Alternatively, users may visit the login page on their own volition. In that case, after logging in the user they should be sent to the root folder's Default.aspx page.

Logging In the User

Assuming that the supplied credentials are valid, we need to create a forms authentication ticket, thereby logging in the user to the site. The FormsAuthentication class in the System.Web.Security namespace provides

assorted methods for logging in and logging out users via the forms authentication system. While there are several methods in the FormsAuthentication class, the three we are interested in at this juncture are:

- GetAuthCookie(*username, persistCookie*) – creates a forms authentication ticket for the supplied name *username*. Next, this method creates and returns an HttpCookie object that holds the contents of the authentication ticket. If *persistCookie* is true, a persistent cookie is created.
- SetAuthCookie(*username, persistCookie*) – calls the GetAuthCookie(*username*, *persistCookie*) method to generate the forms authentication cookie. This method then adds the cookie returned by GetAuthCookie to the Cookies collection (assuming cookies-based forms authentication is being used; otherwise, this method calls an internal class that handles the cookieless ticket logic).
- RedirectFromLoginPage(*username, persistCookie*) – this method calls SetAuthCookie(*username*, *persistCookie*), and then redirects the user to the appropriate page.

GetAuthCookie is handy when you need to modify the authentication ticket before writing the cookie out to the Cookies collection. SetAuthCookie is useful if you want to create the forms authentication ticket and add it to the Cookies collection, but do not want to redirect the user to the appropriate page. Perhaps you want to keep them on the login page or send them to some alternate page.

Since we want to log in the user and redirect them to the appropriate page, let's use RedirectFromLoginPage. Update the LoginButton's Click event handler, replacing the two commented TODO lines with the following line of code:

FormsAuthentication.RedirectFromLoginPage(UserName.Text, RememberMe.Checked);

When creating the forms authentication ticket we use the UserName TextBox's Text property for the forms authentication ticket *username* parameter, and the checked state of the RememberMe CheckBox for the *persistCookie* parameter.

To test the login page, visit it in a browser. Start by entering invalid credentials, such as a username of "Nope" and a password of "wrong". Upon clicking the Login button a postback will occur and the InvalidCredentialsMessage Label will be displayed.

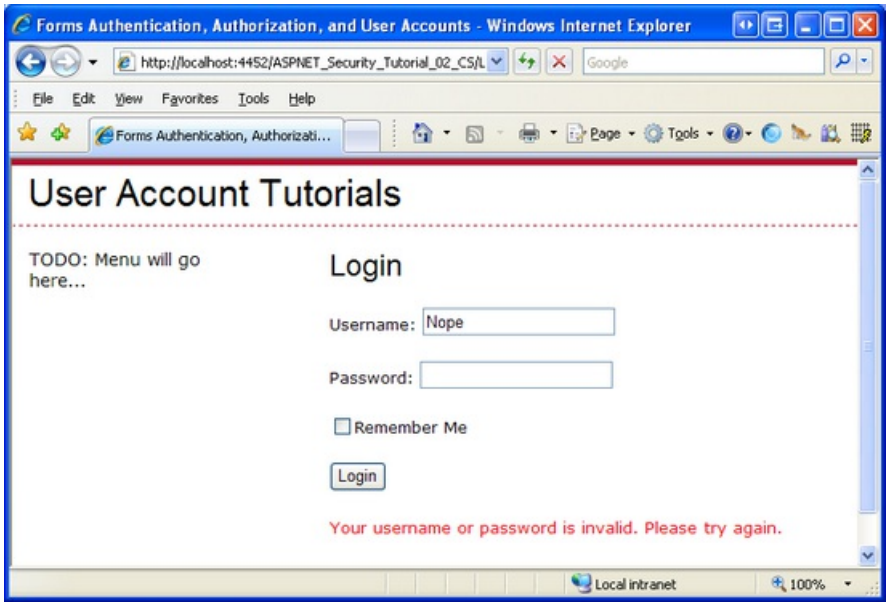


Figure 10: The InvalidCredentialsMessage Label is Displayed When Entering Invalid Credentials (Click to view full-size image)

Next, enter valid credentials and click the Login button. This time when the postback occurs a forms authentication ticket is created and you are automatically redirected back to Default.aspx. At this point you have logged in to the website, although there are no visual cues to indicate that you are currently logged in. In Step 4 we will see how to programmatically determine whether a user is logged in or not as well as how to identify the

user visiting the page.

Step 5 examines techniques for logging a user out of the website.

Securing the Login Page

When the user enters her credentials and submits the login page form, the credentials – including her password – are transmitted over the Internet to the web server in *plain text*. That means any hacker sniffing the network traffic can see the username and password. To prevent this, it is essential to encrypt the network traffic by using Secure Socket Layers (SSL). This will ensure that the credentials (as well as the entire page's HTML markup) are encrypted from the moment they leave the browser until they are received by the web server.

Unless your website contains sensitive information, you will only need to use SSL on the login page and on other pages where the user's password would otherwise be sent over the wire in plain text. You do not need to worry about securing the forms authentication ticket since, by default, it is both encrypted and digitally signed (to prevent tampering). A more thorough discussion on forms authentication ticket security is presented in the following tutorial.

> **NOTE**
>
> Many financial and medical websites are configured to use SSL on *all* pages accessible to authenticated users. If you are building such a website you can configure the forms authentication system so that the forms authentication ticket is only transmitted over a secure connection. We will look at the various forms authentication configuration options in the next tutorial, *Forms Authentication Configuration and Advanced Topics*.

Step 4: Detecting Authenticated Visitors and Determining Their Identity

At this point we have enabled forms authentication and created a rudimentary login page, but we have yet to examine how we can determine whether a user is authenticated or anonymous. In certain scenarios we may wish to display different data or information depending on whether an authenticated or anonymous user is visiting the page. Moreover, we oftentimes need to know the identity of the authenticated user.

Let's augment the existing Default.aspx page to illustrate these techniques. In Default.aspx add two Panel controls, one named AuthenticatedMessagePanel and another named AnonymousMessagePanel. Add a Label control named WelcomeBackMessage in the first Panel. In the second Panel add a HyperLink control, set its Text property to "Log In" and its NavigateUrl property to "~/Login.aspx". At this point the declarative markup for Default.aspx should look similar to the following:

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" Runat="Server">
    <asp:Panel runat="server" ID="AuthenticatedMessagePanel">
        <asp:Label runat="server" ID="WelcomeBackMessage"></asp:Label>
    </asp:Panel>

    <asp:Panel runat="Server" ID="AnonymousMessagePanel">
        <asp:HyperLink runat="server" ID="lnkLogin" Text="Log In" NavigateUrl="~/Login.aspx">
</asp:HyperLink>
    </asp:Panel>
</asp:Content>
```

As you have probably guessed by now, the idea here is to display just the AuthenticatedMessagePanel to authenticated visitors and just the AnonymousMessagePanel to anonymous visitors. To accomplish this we need to set these Panels' Visible properties depending on whether the user is logged in or not.

The Request.IsAuthenticated property returns a Boolean value indicating whether the request has been

authenticated. Enter the following code into the Page_Load event handler code:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Request.IsAuthenticated)
    {
        WelcomeBackMessage.Text = "Welcome back!";

        AuthenticatedMessagePanel.Visible = true;
        AnonymousMessagePanel.Visible = false;
    }
    else
    {
        AuthenticatedMessagePanel.Visible = false;
        AnonymousMessagePanel.Visible = true;
    }
}
```

With this code in place, visit Default.aspx through a browser. Assuming that you have yet to log in, you will see a link to the login page (see Figure 11). Click this link and log in to the site. As we saw in Step 3, after entering your credentials you will be returned to Default.aspx, but this time the page shows the "Welcome back!" message (see Figure 12).

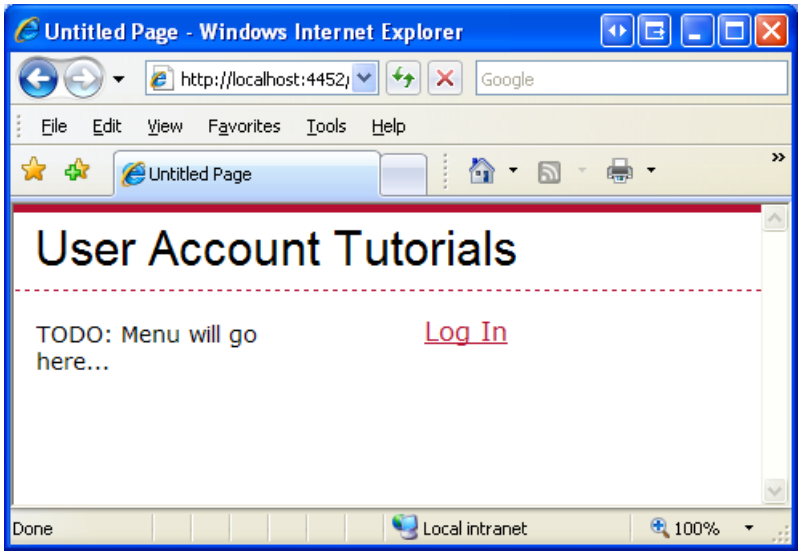


Figure 11: When Visiting Anonymously, a Log In Link is Displayed

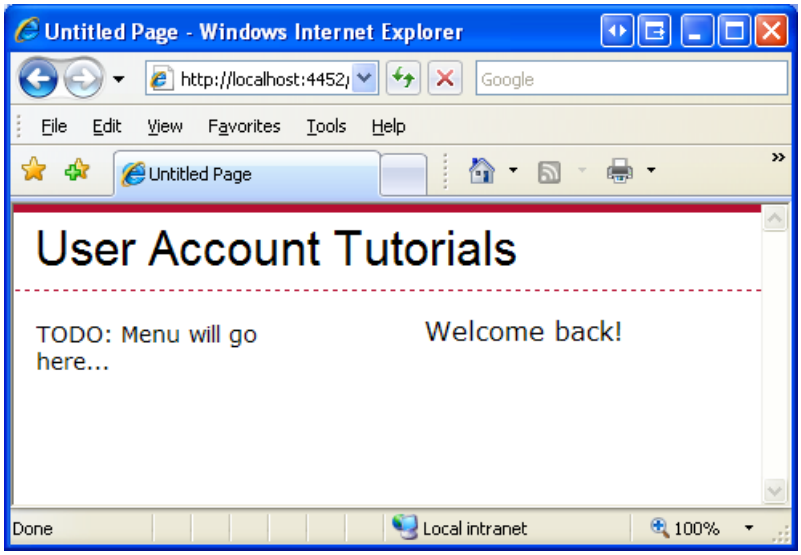


Figure 12: Authenticated Users are Shown the "Welcome back!" Message

We can determine the currently logged on user's identity via the HttpContext object's User property. The HttpContext object represents information about the current request, and is the home for such common ASP.NET objects as Response, Request, and Session, among others. The User property represents the security context of the current HTTP request and implements the IPrincipal interface.

The User property is set by the FormsAuthenticationModule. Specifically, when the FormsAuthenticationModule finds a forms authentication ticket in the incoming request, it creates a new GenericPrincipal object and assigns it to the User property.

Principal objects (like GenericPrincipal) provide information on the user's identity and the roles to which they belong. The IPrincipal interface defines two members:

- IsInRole(*roleName*) – a method that returns a Boolean value indicating if the principal belongs to the specified role.
- Identity – a property that returns an object that implements the IIdentity interface. The IIdentity interface defines three properties: AuthenticationType, IsAuthenticated, and Name.

We can determine the name of the current visitor using the following code:

string currentUsersName = User.Identity.Name;

When using forms authentication, a FormsIdentity object is created for the GenericPrincipal's Identity property. The FormsIdentity class always returns the string "Forms" for its AuthenticationType property and true for its IsAuthenticated property. The Name property returns the username specified when creating the forms authentication ticket. In addition to these three properties, FormsIdentity includes access to the underlying authentication ticket via its Ticket property. The Ticket property returns an object of type FormsAuthenticationTicket, which has properties like Expiration, IsPersistent, IssueDate, Name, and so on.

The important point to take away here is that the *username* parameter specified in the FormsAuthentication.GetAuthCookie(*username*, *persistCookie*), FormsAuthentication.SetAuthCookie(*username*, *persistCookie*), and FormsAuthentication.RedirectFromLoginPage(*username*, *persistCookie*) methods is the same value returned by User.Identity.Name. Moreover, the authentication ticket created by these methods is available by casting User.Identity to a FormsIdentity object and then accessing the Ticket property:

```
FormsIdentity ident = User.Identity as FormsIdentity;
FormsAuthenticationTicket authTicket = ident.Ticket;
```

Let's provide a more personalized message in Default.aspx. Update the Page_Load event handler so that the WelcomeBackMessage Label's Text property is assigned the string "Welcome back, *username*!"

WelcomeBackMessage.Text = "Welcome back, " + User.Identity.Name + "!";

Figure 13 shows the effect of this modification (when logging in as user Scott).

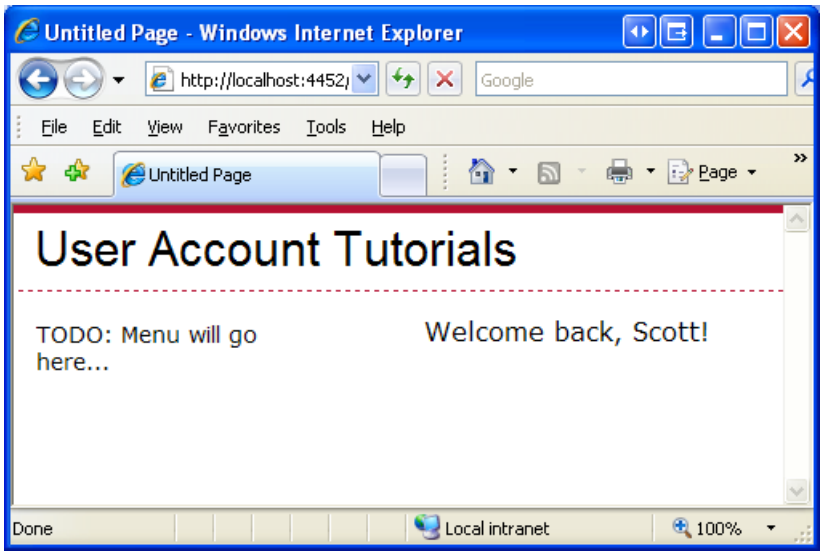


Figure 13: The Welcome Message Includes the Currently Logged In User's Name

Using the LoginView and LoginName Controls

Displaying different content to authenticated and anonymous users is a common requirement; so is displaying the name of the currently logged on user. For that reason, ASP.NET includes two Web controls that provide the same functionality shown in Figure 13, but without the need to write a single line of code.

The LoginView control is a template-based Web control that makes it easy to display different data to authenticated and anonymous users. The LoginView includes two predefined templates:

- AnonymousTemplate – any markup added to this template is only displayed to anonymous visitors.
- LoggedInTemplate – this template's markup is shown only to authenticated users.

Let's add the LoginView control to our site's master page, Site.master. Rather than adding just the LoginView control, though, let's add both a new ContentPlaceHolder control and then put the LoginView control within that new ContentPlaceHolder. The rationale for this decision will become apparent shortly.

> **NOTE**
> In addition to the AnonymousTemplate and LoggedInTemplate, the LoginView control can include role-specific templates. Role-specific templates show markup only to those users that belong to a specified role. We will examine the role-based features of the LoginView control in a future tutorial.

Start by adding a ContentPlaceHolder named LoginContent into the master page within the navigation element. You can simply drag a ContentPlaceHolder control from the Toolbox onto the Source view, placing the resulting markup right above the "TODO: Menu will go here..." text.

```
<div id="navigation">
    <asp:ContentPlaceHolder ID="LoginContent" runat="server">
    </asp:ContentPlaceHolder>

    TODO: Menu will go here...
</div>
```

Next, add a LoginView control within the LoginContent ContentPlaceHolder. Content placed into the master page's ContentPlaceHolder controls are considered *default content* for the ContentPlaceHolder. That is, ASP.NET pages that use this master page can specify their own content for each ContentPlaceHolder or use the master page's default content.

The LoginView and other login-related controls are located in the Toolbox's Login tab.

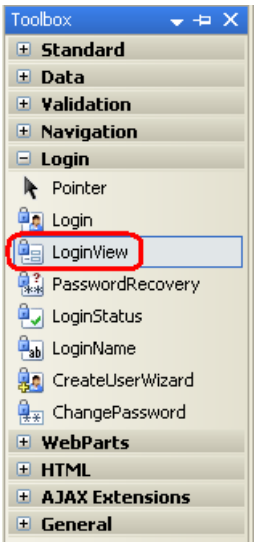


Figure 14: The LoginView Control in the Toolbox

Next, add two <br /> elements immediately after the LoginView control, but still within the ContentPlaceHolder. At this point, the navigation element's markup should look like the following:

```
<div id="navigation">
    <asp:ContentPlaceHolder ID="LoginContent" runat="server">
        <asp:LoginView ID="LoginView1" runat="server">
        </asp:LoginView>
        <br /><br />
    </asp:ContentPlaceHolder>

    TODO: Menu will go here...
</div>
```

The LoginView's templates can be defined from the Designer or the declarative markup. From Visual Studio's Designer, expand the LoginView's smart tag, which lists the configured templates in a drop-down list. Type in the text "Hello, stranger" into the AnonymousTemplate; next, add a HyperLink control and set its Text and NavigateUrl properties to "Log In" and "~/Login.aspx", respectively.

After configuring the AnonymousTemplate, switch to the LoggedInTemplate and enter the text, "Welcome back, ". Then drag a LoginName control from the Toolbox into the LoggedInTemplate, placing it immediately after the "Welcome back, " text. The LoginName control, as its name implies, displays the name of the currently logged in user. Internally, the LoginName control simply outputs the User.Identity.Name property

After making these additions to the LoginView's templates, the markup should look similar to the following:

```
<div id="navigation">
    <asp:ContentPlaceHolder ID="LoginContent" runat="server">
        <asp:LoginView ID="LoginView1" runat="server">
            <LoggedInTemplate>
                Welcome back,
                <asp:LoginName ID="LoginName1" runat="server" />.
            </LoggedInTemplate>
            <AnonymousTemplate>
                Hello, stranger.
                <asp:HyperLink ID="lnkLogin" runat="server" NavigateUrl="~/Login.aspx">Log
In</asp:HyperLink>
            </AnonymousTemplate>
        </asp:LoginView>

        <br /><br />
    </asp:ContentPlaceHolder>

    TODO: Menu will go here...
</div>
```

With this addition to the Site.master master page, each page in our website will display a different message depending on whether the user is authenticated. Figure 15 shows the Default.aspx page when visited through a browser by user Jisun. The "Welcome back, Jisun" message is repeated twice: once in the master page's navigation section on the left (via the LoginView control we just added) and once in the Default.aspx's content area (via Panel controls and programmatic logic).

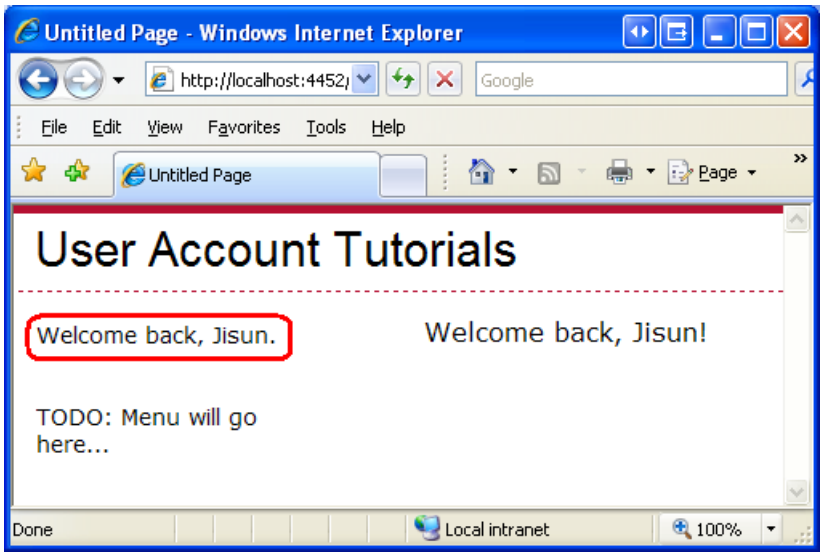


Figure 15: The LoginView Control Displays "Welcome back, Jisun."

Because we added the LoginView to the master page, it can appear in every page on our site. However, there may be web pages where we don't want to show this message. One such page is the login page, since a link to the login page seems out of place there. Since we placed the LoginView control in a ContentPlaceHolder in the master page, we can override this default markup in our content page. Open Login.aspx and go to the Designer. Since we have not explicitly defined a Content control in Login.aspx for the LoginContent ContentPlaceHolder in the master page, the login page will show the master page's default markup for this ContentPlaceHolder. You can see this through the Designer – the LoginContent ContentPlaceHolder shows the default markup (the LoginView control).

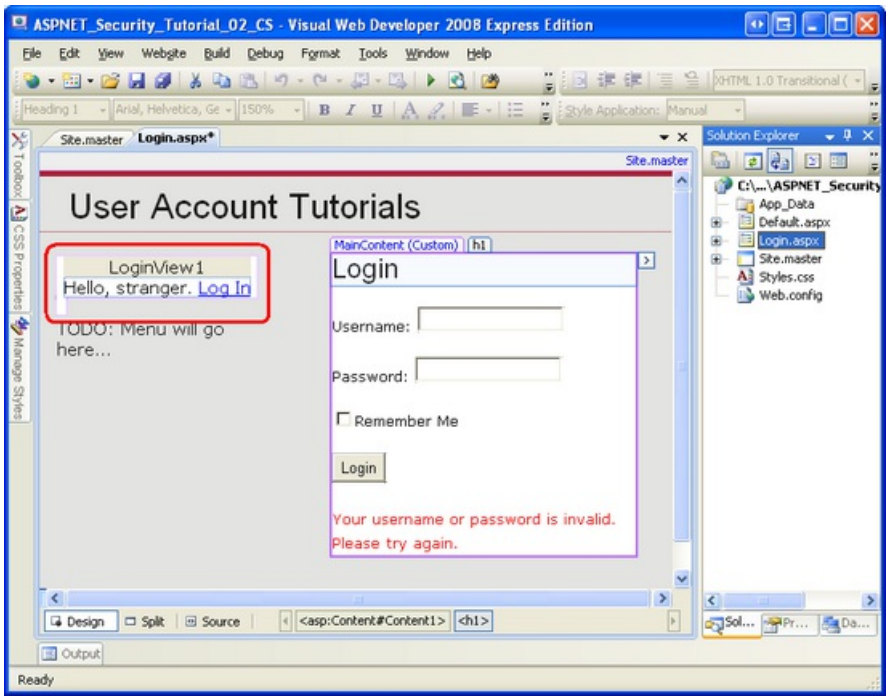


Figure 16: The Login Page Shows the Default Content for the Master Page's LoginContent ContentPlaceHolder (Click to view full-size image)

To override the default markup for the LoginContent ContentPlaceHolder, simply right-click on the region in the Designer and choose the Create Custom Content option from the context menu. (When using Visual Studio 2008 the ContentPlaceHolder includes a smart tag that, when selected, offers the same option.) This adds a new Content control to the page's markup and thereby allows us to define custom content for this page. You could add a custom message here, such as "Please log in...", but let's just leave this blank.

> **NOTE**
>
> In Visual Studio 2005, creating custom content creates an empty Content control in the ASP.NET page. In Visual Studio 2008, however, creating custom content copies the master page's default content into the newly created Content control. If you are using Visual Studio 2008, then, after creating the new Content control make sure to clear out the content copied over from the master page.

Figure 17 shows the Login.aspx page when visited from a browser after making this change. Note that there is no "Hello, stranger" or "Welcome back, *username*" message in the left navigation as there is when visiting Default.aspx.

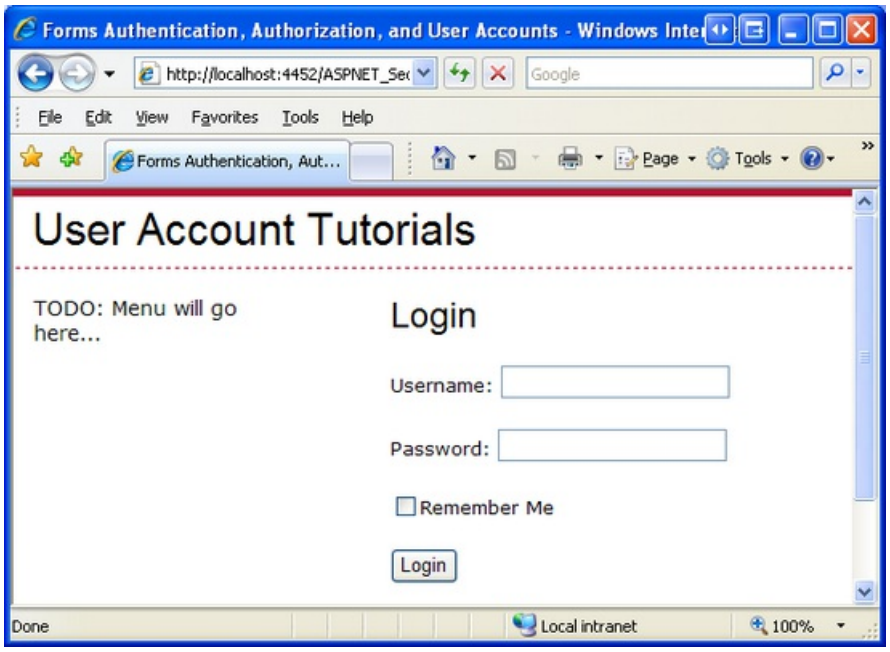


Figure 17: The Login Page Hides the Default LoginContent ContentPlaceHolder's Markup (Click to view full-size image)

Step 5: Logging Out

In Step 3 we looked at building a login page to log a user in to the site, but we have yet to see how to log a user out. In addition to methods for logging a user in, the FormsAuthentication class also provides a SignOut method. The SignOut method simply destroys the forms authentication ticket, thereby logging the user out of the site.

Offering a log out link is such a common feature that ASP.NET includes a control specifically designed to log a user out. The LoginStatus control displays either a "Login" LinkButton or a "Logout" LinkButton, depending on the user's authentication status. A "Login" LinkButton is rendered for anonymous users, whereas a "Logout" LinkButton is displayed to authenticated users. The text for the "Login" and "Logout" LinkButtons can be configured via the LoginStatus's LoginText and LogoutText properties.

Clicking the "Login" LinkButton causes a postback, from which a redirect is issued to the login page. Clicking the "Logout" LinkButton causes the LoginStatus control to invoke the FormsAuthentication.SignOff method and then redirects the user to a page. The page the logged off user is redirected to depends on the LogoutAction property, which can be assigned to one of the three following values:

- Refresh – the default; redirects the user to the page they were just visiting. If the page they were just visiting does not allow anonymous users, then the FormsAuthenticationModule will automatically redirect the user to the login page.

You may be curious as to why a redirect is performed here. If the user wants to remain on the same page, why the need for the explicit redirect? The reason is because when the "Logoff" LinkButton is clicked, the user still has the forms authentication ticket in their cookies collection. Consequently, the postback request is an authenticated request. The LoginStatus control calls the SignOut method, but that happens after the FormsAuthenticationModule has authenticated the user. Therefore, an explicit redirect causes the browser to re-request the page. By the time the browser re-requests the page, the forms authentication ticket has been removed and therefore the incoming request is anonymous.

- Redirect – the user is redirected to the URL specified by the LoginStatus's LogoutPageUrl property.
- RedirectToLoginPage – the user is redirected to the login page.

Let's add a LoginStatus control to the master page and configure it to use the Redirect option to send the user to a page that displays a message confirming that they have been signed out. Start by creating a page in the root

directory named Logout.aspx. Don't forget to associate this page with the Site.master master page. Next, enter a message in the page's markup explaining to the user that they have been logged out.

Next, return to the Site.master master page and add a LoginStatus control beneath the LoginView in the LoginContent ContentPlaceHolder. Set the LoginStatus control's LogoutAction property to Redirect and its LogoutPageUrl property to "~/Logout.aspx".

```
<div id="navigation">
    <asp:ContentPlaceHolder ID="LoginContent" runat="server">
        <asp:LoginView ID="LoginView1" runat="server">
            <LoggedInTemplate>
                Welcome back,
                <asp:LoginName ID="LoginName1" runat="server" />.
            </LoggedInTemplate>
            <AnonymousTemplate>
                Hello, stranger.
                <asp:HyperLink ID="lnkLogin" runat="server" NavigateUrl="~/Login.aspx">Log
In</asp:HyperLink>
            </AnonymousTemplate>
        </asp:LoginView>
        <br />
        <asp:LoginStatus ID="LoginStatus1" runat="server" LogoutAction="Redirect"
LogoutPageUrl="~/Logout.aspx" />

        <br /><br />
    </asp:ContentPlaceHolder>

    TODO: Menu will go here...
</div>
```

Since the LoginStatus is outside of the LoginView control, it will appear for both anonymous and authenticated users, but that's OK because the LoginStatus will correctly display a "Login" or "Logout" LinkButton. With the addition of the LoginStatus control, the "Log In" HyperLink in the AnonymousTemplate is superfluous, so remove it.

Figure 18 shows Default.aspx when Jisun visits. Note that the left column displays the message, "Welcome back, Jisun" along with a link to log out. Clicking the log out LinkButton causes a postback, signs Jisun out of the system, and then redirects her to Logout.aspx. As Figure 19 shows, by the time Jisun reaches Logout.aspx she has already been signed out and is therefore anonymous. Consequently, the left column shows the text "Welcome, stranger" and a link to the login page.

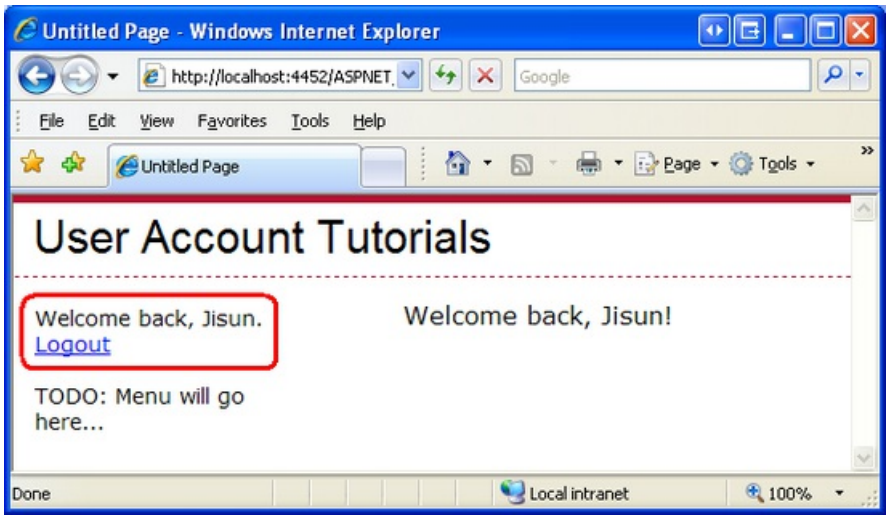


Figure 18: Default.aspx Shows "Welcome Back, Jisun" Along with a "Logout" LinkButton (Click to view full-size image)

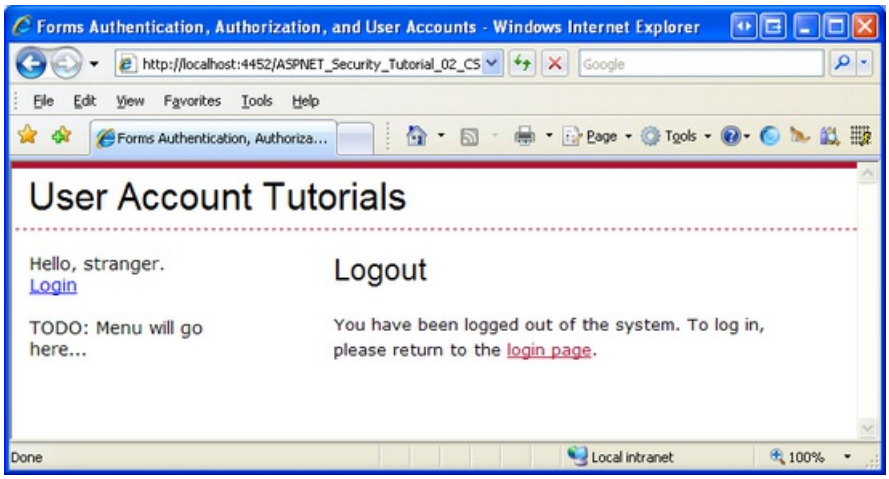


Figure 19: Logout.aspx Shows "Welcome, stranger" Along with a "Login" LinkButton (Click to view full-size image)

> **NOTE**
>
> I encourage you to customize the Logout.aspx page to hide the master page's LoginContent ContentPlaceHolder (like we did for Login.aspx in Step 4). The reason is because the "Login" LinkButton rendered by the LoginStatus control (the one beneath "Hello, stranger") sends the user to the login page passing the current URL in the ReturnUrl querystring parameter. In short, if a user who has logged out clicks this LoginStatus's "Login" LinkButton, and then logs in, they will be redirected back to Logout.aspx, which could easily confuse the user.

Summary

In this tutorial we started with an examination of the forms authentication workflow and then turned to implementing forms authentication in an ASP.NET application. Forms authentication is powered by the FormsAuthenticationModule, which has two responsibilities: identifying users based on their forms authentication ticket, and redirecting unauthorized users to the login page.

The .NET Framework's FormsAuthentication class includes methods for creating, inspecting, and removing forms authentication tickets. The Request.IsAuthenticated property and User object provide additional programmatic support for determining whether a request is authenticated and information about the user's identity. There are also the LoginView, LoginStatus, and LoginName Web controls, which give developers a quick, code-free way for performing many common login-related tasks. We will examine these and other login-related Web controls in greater detail in future tutorials.

This tutorial provided a cursory overview of forms authentication. We did not examine the assorted configuration options, look at how cookieless forms authentication tickets work, or explore how ASP.NET protects the contents of the forms authentication ticket. We will discuss these topics and more in the next tutorial.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Changes Between IIS6 and IIS7 Security
- Login ASP.NET Controls
- Professional ASP.NET 2.0 Security, Membership, and Role Management (ISBN: 978-0-7645-9698-8)
- The `<authentication>` Element
- The `<forms>` Element for `<authentication>`

Video Training on Topics Contained in this Tutorial

- Using Basic Forms Authentication in ASP.NET

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial include Alicja Maziarz, John Suru, and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Forms Authentication Configuration and Advanced Topics (C#)

2/20/2020 • 38 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

In this tutorial we will examine the various forms authentication settings and see how to modify them through the forms element. This will entail a detailed look at customizing the forms authentication ticket's timeout value, using a login page with a custom URL (like SignIn.aspx instead of Login.aspx), and cookieless forms authentication tickets.

Introduction

In the previous tutorial we looked at the steps necessary for implementing forms authentication in an ASP.NET application, from specifying configuration settings in Web.config to creating a log in page to displaying different content for authenticated and anonymous users. Recall that we configured the website to use forms authentication by setting the mode attribute of the <authentication> element to Forms. The <authentication> element may optionally include a <forms> child element, through which an assortment of forms authentication settings may be specified.

In this tutorial we will examine the various forms authentication settings and see how to modify them through the <forms> element. This will entail a detailed look at customizing the forms authentication ticket's timeout value, using a login page with a custom URL (like SignIn.aspx instead of Login.aspx), and cookieless forms authentication tickets. We will also examine the makeup of the forms authentication ticket more closely and see the precautions ASP.NET takes to ensure that the ticket's data is secure from inspection and tampering. Finally, we will look at how to store extra user data in the forms authentication ticket and how to model this data through a custom principal object.

Step 1: Examining the <forms> Configuration Settings

The forms authentication system in ASP.NET offers a number of configuration settings that can be customized on an application-by-application basis. This includes settings like: the lifetime of the forms authentication ticket; what sort of protection is applied to the ticket; under what conditions cookieless authentication tickets are used; the path to the login page; and other information. To modify the default values, add a <forms> element as a child of the <authentication> element, specifying those property values you want to customize as XML attributes like so:

```
<authentication mode="Forms">

  <forms

 propertyName1="value1"

 propertyName2="value2"

  ...

 propertyNameN="valueN"

  />

</authentication>
```

Table 1 summarizes the properties that can be customized through the <forms> element. Since Web.config is an XML file, the attribute names in the left column are case-sensitive.

ATTRIBUTE	DESCRIPTION
cookieless	This attribute specifies under what conditions the authentication ticket is stored in a cookie versus being embedded in the URL. Allowable values are: UseCookies; UseUri; AutoDetect; and UseDeviceProfile (the default). Step 2 examines this setting in more detail.
defaultUrl	Indicates the URL that users are redirected to after signing in from the login page if there is no RedirectUrl value specified in the querystring. The default value is default.aspx.
domain	When using cookie-based authentication tickets, this setting specifies the cookie's domain value. The default value is an empty string, which causes the browser to use the domain from which it was issued (such as www.yourdomain.com). In this case, the cookie will **not** be sent when making requests to subdomains, such as admin.yourdomain.com. If you want the cookie to be passed to all subdomains you need to customize the domain attribute setting it to yourdomain.com.
enableCrossAppRedirects	A Boolean value indicating whether authenticated users are remembered when redirected to URLs in other web applications on the same server. The default is false.
loginUrl	The URL of the login page. The default value is login.aspx.
name	When using cookie-based authentication tickets, the name of the cookie. The default is .ASPXAUTH.
path	When using cookie-based authentication tickets, this setting specifies the cookie's path attribute. The path attribute enables a developer to limit the scope of a cookie to a particular directory hierarchy. The default value is /, which informs the browser to send the authentication ticket cookie to any request made to the domain.

ATTRIBUTE	DESCRIPTION
protection	Indicates what techniques are used to protect the forms authentication ticket. The allowable values are: All (the default); Encryption; None; and Validation. These settings are discussed in detail in Step 3.
requireSSL	A Boolean value that indicates whether an SSL connection is required to transmit the authentication cookie. The default value is false.
slidingExpiration	A Boolean value that indicates whether the authentication cookie's timeout is reset each time the user visits the site during a single session. The default value is true. The authentication ticket timeout policy is discussed in more detail in the Specifying the Ticket's Timeout Value section.
timeout	Specifies the time, in minutes, after which the authentication ticket cookie expires. The default value is 30. The authentication ticket timeout policy is discussed in more detail in the Specifying the Ticket's Timeout Value section.

Table 1: A Summary of the <forms> Element's Attributes

In ASP.NET 2.0 and beyond, the default forms authentication values are hard-coded in the FormsAuthenticationConfiguration class in the .NET Framework. Any modifications must be applied on an application-by-application basis in the Web.config file. This differs from ASP.NET 1.x, where the default forms authentication values were stored in the machine.config file (and could therefore be modified via editing machine.config). While on the topic of ASP.NET 1.x, it is worthwhile to mention that a number of the forms authentication system settings have different default values in ASP.NET 2.0 and beyond than in ASP.NET 1.x. If you are migrating your application from an ASP.NET 1.x environment, it is important to be aware of these differences. Consult the <forms> element technical documentation for a list of the differences.

> **NOTE**
> Several forms authentication settings, such as the timeout, domain, and path, specify details for the resulting forms authentication ticket cookie. For more information on cookies, how they work, and their various properties, read this Cookies tutorial.

Specifying the Ticket's Timeout Value

The forms authentication ticket is a token that represents an identity. With cookie-based authentication tickets, this token is held in the form of a cookie and sent to the web server on each request. Possession of the token, in essence, declares, I am *username*, I have already logged in, and is used so that a user's identity can be remembered across page visits.

The forms authentication ticket not only includes the user's identity, but also contains information to help ensure the integrity and security of the token. After all, we don't want a nefarious user to be able to create a counterfeit token, or to modify a legit token in some underhanded way.

One such bit of information included in the ticket is an *expiry*, which is the date and time the ticket is no longer valid. Each time the FormsAuthenticationModule inspects an authentication ticket, it ensures that the ticket's expiry has not yet passed. If it has, it disregards the ticket and identifies the user as being anonymous. This safeguard helps protect against replay attacks. Without an expiry, if a hacker was able to get her hands on a user's valid authentication ticket - perhaps by gaining physical access to their computer and rooting through their cookies - they could send a request to the server with this stolen authentication ticket and gain entry. While

the expiry doesn't prevent this scenario, it does limit the window during which such an attack can succeed.

> **NOTE**
>
> Step 3 details additional techniques used by the forms authentication system to protect the authentication ticket.

When creating the authentication ticket, the forms authentication system determines its expiry by consulting the timeout setting. As noted in Table 1, the timeout setting defaults to 30 minutes, meaning that when the forms authentication ticket is created its expiry is set to a date and time 30 minutes in the future.

The expiry defines an absolute time in the future when the forms authentication ticket expires. But usually developers want to implement a sliding expiry, one that is reset every time the user revisits the site. This behavior is determined by the slidingExpiration settings. If set to true (the default), each time the FormsAuthenticationModule authenticates a user, it updates the ticket's expiry. If set to false, the expiry is not updated on each request, thereby causing the ticket to expire exactly timeout number of minutes past when the ticket was first created.

> **NOTE**
>
> The expiry stored in the authentication ticket is an absolute date and time value, like August 2, 2008 11:34 AM. Moreover, the date and time are relative to the web server's local time. This design decision can have some interesting side effects around Daylight Saving Time (DST), which is when clocks in the United States are moved ahead one hour (assuming the web server is hosted in a locale where Daylight Saving Time is observed). Consider what would happen for an ASP.NET website with a 30 minute expiry near the time that DST begins (which is at 2:00 AM). Imagine a visitor signs on to the site on March 11, 2008 at 1:55 AM. This would generate a forms authentication ticket that expires at March 11, 2008 at 2:25 AM (30 minutes in the future). However, once 2:00 AM rolls around, the clock jumps to 3:00 AM because of DST. When the user loads a new page six minutes after signing in (at 3:01 AM), the FormsAuthenticationModule notes that the ticket has expired and redirects the user to the login page. For a more thorough discussion on this and other authentication ticket timeout oddities, as well as workarounds, pick up a copy of Stefan Schackow's *Professional ASP.NET 2.0 Security, Membership, and Role Management* (ISBN: 978-0-7645-9698-8).

Figure 1 illustrates the workflow when slidingExpiration is set to false and timeout is set to 30. Note that the authentication ticket generated at login contains the expiration date, and this value is not updated on subsequent requests. If the FormsAuthenticationModule finds that the ticket has expired, it discards it and treats the request as anonymous.

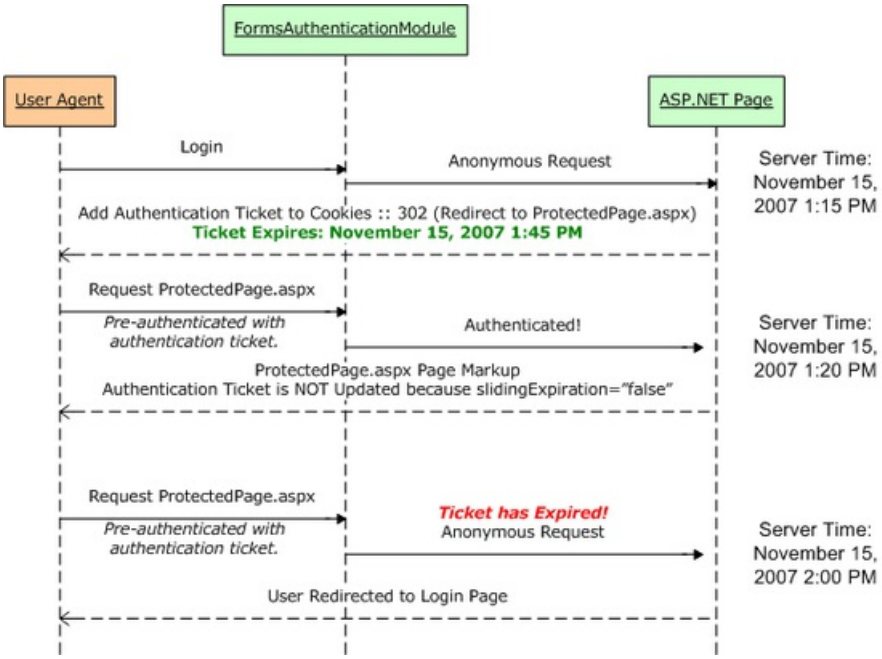


Figure 01: A Graphical Representation of the Forms Authentication Ticket's Expiry When slidingExpiration is false(Click to view full-size image)

Figure 2 shows the workflow when slidingExpiration is set to true and timeout is set to 30. When an authenticated request is received (with a non-expired ticket) its expiry is updated to timeout number of minutes in the future.

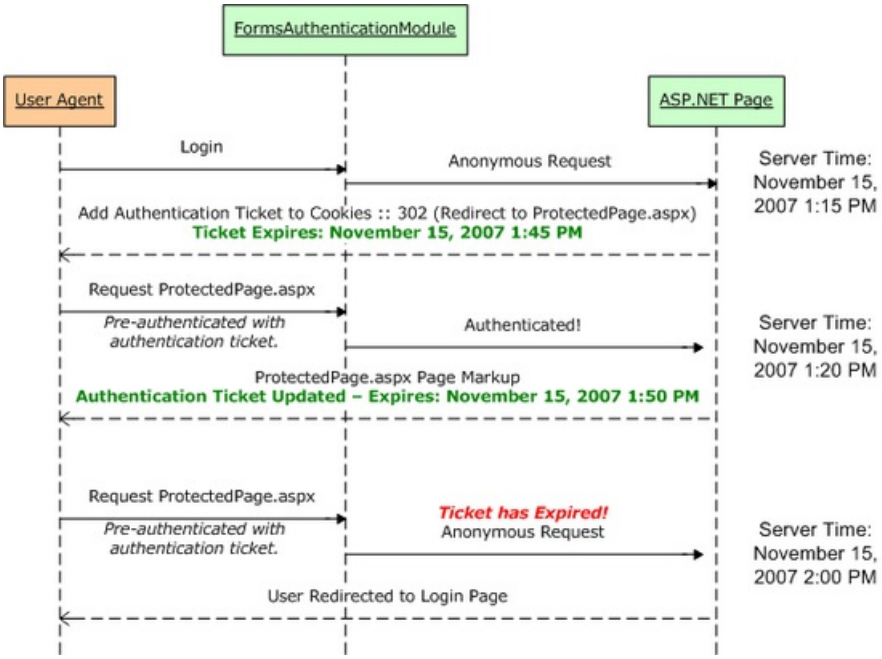


Figure 02: A Graphical Representation of the Forms Authentication Ticket's When slidingExpiration is true (Click to view full-size image)

When using cookie-based authentication tickets (the default), this discussion becomes a little more confusing because cookies can also have their own expiries specified. A cookie's expiry (or lack thereof) instructs the browser when the cookie should be destroyed. If the cookie lacks an expiry, it is destroyed when the browser shuts down. If an expiry is present, however, the cookie remains stored on the user's computer until the date and time specified in the expiry has passed. When a cookie is destroyed by the browser, it is no longer sent to the web server. Therefore, the destruction of a cookie is analogous to the user logging out of the site.

> **NOTE**
>
> Of course, a user may proactively remove any cookies stored on their computer. In Internet Explorer 7, you would go to Tools, Options, and click the Delete button in the Browsing history section. From there, click the Delete cookies button.

The forms authentication system creates session-based or expiry-based cookies depending on the value passed in to the *persistCookie* parameter. Recall that the FormsAuthentication class's GetAuthCookie, SetAuthCookie, and RedirectFromLoginPage methods take in two input parameters: *username* and *persistCookie*. The login page we created in the preceding tutorial included a Remember me CheckBox, which determined whether a persistent cookie was created. Persistent cookies are expiry-based; non-persistent cookies are session-based.

The timeout and slidingExpiration concepts already discussed apply the same to both session- and expiry-based cookies. There is only one minor difference in execution: when using expiry-based cookies with slidingTimeout set to true, the cookie's expiry is only updated when more than half of the specified time has elapsed.

Let's update our website's authentication ticket timeout policies so that tickets timeout after one hour (60 minutes), using a sliding expiration. To effect this change, update the Web.config file, adding a <forms> element to the <authentication> element with the following markup:

```
<authentication mode="Forms">

  <forms

  slidingExpiration="true"

  timeout="60"

  />

</authentication>
```

Using an Login Page URL Other than Login.aspx

Since the FormsAuthenticationModule automatically redirects unauthorized users to the login page, it needs to know the login page's URL. This URL is specified by the loginUrl attribute in the <forms> element and defaults to login.aspx. If you are porting over an existing website, you may already have a login page with a different URL, one that has already been bookmarked and indexed by search engines. Rather than renaming your existing login page to login.aspx and breaking links and users' bookmarks, you can instead modify the loginUrl attribute to point to your login page.

For example, if your login page was named SignIn.aspx and was located in the Users directory, you could point the loginUrl configuration setting to ~/Users/SignIn.aspx like so:

```
<authentication mode="Forms">

  <forms

  loginUrl="~/Users/SignIn.aspx"

  />

</authentication>
```

Since our current application already has a login page named Login.aspx, there's no need to specify a custom value in the <forms> element.

Step 2: Using Cookieless Forms Authentication Tickets

By default the forms authentication system determines whether to store its authentication tickets in the cookies collection or embed them in the URL based on the user agent visiting the site. All mainstream desktop browsers like Internet Explorer, Firefox, Opera, and Safari, support cookies, but not all mobile devices do.

The cookie policy used by the forms authentication system depends on the cookieless setting in the <forms> element, which can be assigned one of four values:

- UseCookies - specifies that cookie-based authentication tickets will always be used.
- UseUri - indicates that cookie-based authentication tickets will never be used.
- AutoDetect - if the device profile does not support cookies, cookie-based authentication tickets are not used; if the device profile supports cookies, a probing mechanism is used to determine if cookies are enabled.
- UseDeviceProfile - the default; uses cookie-based authentication tickets only if the device profile supports cookies. No probing mechanism is used.

The AutoDetect and UseDeviceProfile settings rely on a *device profile* in ascertaining whether to use cookie-based or cookieless authentication tickets. ASP.NET maintains a database of various devices and their capabilities, such as whether they support cookies, what version of JavaScript they support, and so on. Each time a device requests a web page from a web server it sends along a *user-agent* HTTP header that identifies the device type.

ASP.NET automatically matches the supplied user-agent string with the corresponding profile specified in its database.

> **NOTE**
> This database of device capabilities is stored in a number of XML files that adhere to the Browser Definition File schema. The default device profile files are located in %WINDIR%\Microsoft.Net\Framework\v2.0.50727\CONFIG\Browsers. You can also add custom files to your application's App_Browsers folder. For more information, see How To: Detect Browser Types in ASP.NET Web Pages.

Because the default setting is UseDeviceProfile, cookieless forms authentication tickets will be used when the site is visited by a device whose profile reports that it does not support cookies.

Encoding the Authentication Ticket in the URL

Cookies are a natural medium for including information from the browser in each request to a particular website, which is why the default forms authentication settings use cookies if the visiting device supports them. If cookies are not supported, an alternate means for passing the authentication ticket from the client to the server must be employed. A common workaround used in cookieless environments is to encode the cookie data in the URL.

The best way to see how such information can be embedded within the URL is to force the site to use cookieless authentication tickets. This can be accomplished by setting the cookieless configuration setting to UseUri:

```
<authentication mode="Forms">

  <forms

  cookieless="UseUri"

  slidingExpiration="true"

  timeout="60"

  />

</authentication>
```

Once you have made this change, visit the site through a browser. When visiting as an anonymous user, the URLs will look exactly like they did before. For example, when visiting Default.aspx page my browser's address bar shows the following URL:

```
http://localhost:2448/ASPNET\_Security\_Tutorial\_03\_CS/default.aspx
```

However, upon logging in, the forms authentication ticket is embedded into the URL. For example, after visiting the login page and logging in as Sam, I am returned to the Default.aspx page, but the URL this time is:

```
http://localhost:2448/ASPNET\_Security\_Tutorial\_03\_CS/(F(jaIOIDTJxIr12xYS-VVgkqKCVAuIoW30Bu0diWi6flQC-
FyMaLXJfow\_Vd9GZkB2Cv-rfezq0gKadKX0YPZCkA2))/default.aspx
```

The forms authentication ticket has been embedded within the URL. The string (F(jaIOIDTJxIr12xYS-VVgkqKCVAuIoW30Bu0diWi6flQC-FyMaLXJfow_Vd9GZkB2Cv-rfezq0gKadKX0YPZCkA2) represents the hex-encoded authentication ticket information, and is the same data that is usually stored within a cookie.

In order for cookieless authentication tickets to work, the system must encode all URLs on the page to include the authentication ticket data, otherwise the authentication ticket will be lost when the user clicks on a link. Thankfully, this embedding logic is performed automatically. To demonstrate this functionality, open the Default.aspx page and add a HyperLink control, setting its Text and NavigateUrl properties to Test Link and SomePage.aspx, respectively. It doesn't matter that there really isn't a page in our project named SomePage.aspx.

Save the changes to Default.aspx and then visit it through a browser. Log on to the site so that the forms authentication ticket is embedded in the URL. Next, from Default.aspx, click the Test Link link. What happened? If no page named SomePage.aspx exists, then a 404 error occurred, but that's not what's important here. Instead, focus on the Address bar in your browser. Note that it includes the forms authentication ticket in the URL!

```
http://localhost:2448/ASPNET\_Security\_Tutorial\_03\_CS/(F(jaIOIDTJxIr12xYS-VVgkqKCVAuIoW30Bu0diWi6flQC-
FyMaLXJfow\_Vd9GZkB2Cv-rfezq0gKadKX0YPZCkA2))/SomePage.aspx
```

The URL SomePage.aspx in the link was automatically converted into a URL that included the authentication ticket - we didn't have to write a lick of code! The form authentication ticket will automatically be embedded in the URL for any hyperlinks that do not start with `http://` or `/`. It doesn't matter if the hyperlink appears in a call to Response.Redirect, in a HyperLink control, or in an anchor HTML element (i.e., `<a href="...">...</a>`). As long as the URL isn't something like `http://www.someserver.com/SomePage.aspx` or `/SomePage.aspx`, the forms authentication ticket will be embedded for us.

> **NOTE**
> Cookieless forms authentication tickets adhere to the same timeout policies as cookie-based authentication tickets. However, cookieless authentication tickets are more prone to replay attacks since the authentication ticket is embedded directly in the URL. Imagine a user who visits a website, logs in, and then pastes the URL in an email to a colleague. If the colleague clicks on that link before the expiry is reached, they will be logged in as the user who sent the email!

Step 3: Securing the Authentication Ticket

The forms authentication ticket is transmitted over the wire either in a cookie or embedded directly within the URL. In addition to identity information, the authentication ticket can also include user data (as we will see in Step 4). Consequently, it is important that the ticket's data is encrypted from prying eyes and (even more importantly) that the forms authentication system can guarantee that the ticket was not tampered with.

To ensure the privacy of the ticket's data, the forms authentication system can encrypt the ticket data. Failure to encrypt the ticket data sends potentially sensitive information over the wire in plain-text.

To guarantee a ticket's authenticity, the forms authentication system must *validate* the ticket. Validation is the act of ensuring that a particular piece of data has not been modified, and is accomplished via a *message authentication code (MAC)*. In a nutshell, the MAC is a small piece of information that identifies the data that needs to be validated (in this case, the ticket). If the data represented by the MAC is modified, then the MAC and the data will not match up. Moreover, it is computationally hard for a hacker to both modify the data and generate his own MAC to correspond with the modified data.

When creating (or modifying) a ticket, the forms authentication system creates a MAC and attaches it to the ticket's data. When a subsequent request arrives, the forms authentication system compares the MAC and ticket data to validate the authenticity of the ticket data. Figure 3 illustrates this workflow graphically.

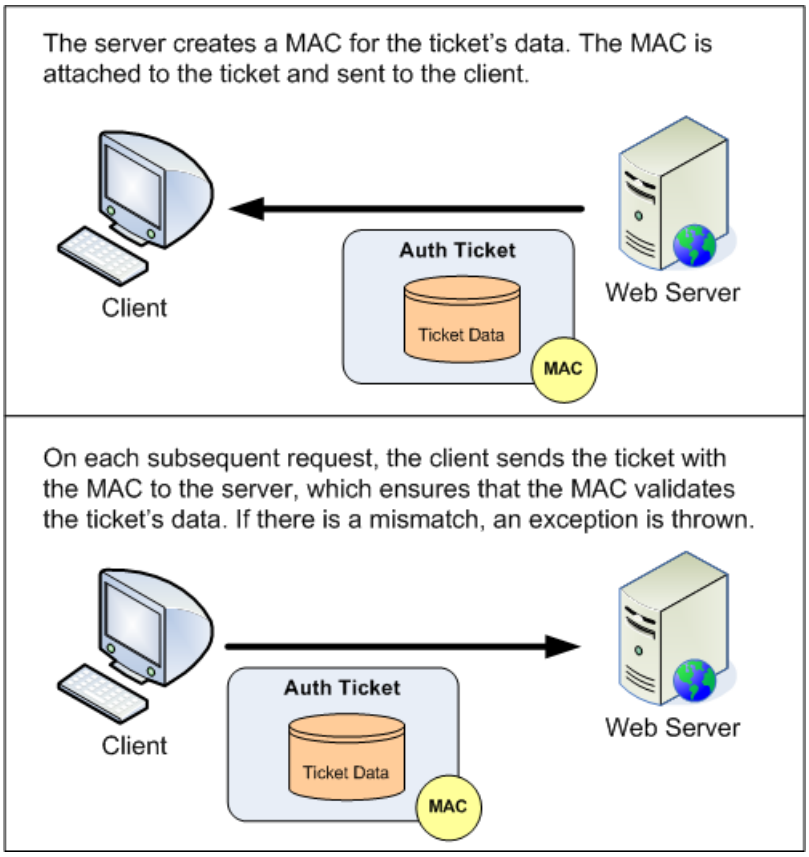


Figure 03: The Ticket's Authenticity is Ensured through a MAC (Click to view full-size image)

What security measures are applied to the authentication ticket depends on the protection setting in the <forms> element. The protection setting may be assigned to one of the following three values:

- All - the ticket is both encrypted and digitally signed (the default).
- Encryption - only encryption is applied - no MAC is generated.
- None - the ticket is neither encrypted nor digitally signed.
- Validation - a MAC is generated, but the ticket data is sent over the wire in plain-text.

Microsoft strongly recommends using the All setting.

Setting the Validation and Decryption Keys

The encryption and hashing algorithms used by the forms authentication system to encrypt and validate the authentication ticket are customizable through the <machineKey> element in Web.config. Table 2 outlines the <machineKey> element's attributes and their possible values.

ATTRIBUTE	DESCRIPTION
decryption	Indicates the algorithm used for encryption. This attribute can have one of the following four values: - Auto - the default; determines the algorithm based on the length of the decryptionKey attribute. - AES - uses the Advanced Encryption Standard (AES) algorithm. - DES - uses the Data Encryption Standard (DES) This algorithm is considered computationally weak and should not be used. - 3DES - uses the Triple DES algorithm, which works by applying the DES algorithm three times.

ATTRIBUTE	DESCRIPTION
decryptionKey	The secret key used by the encryption algorithm. This value must either be a hexadecimal string of the appropriate length (based on the value in decryption), AutoGenerate, or either value appended with,IsolateApps. Adding IsolateApps instructs ASP.NET to use a unique value for each application. The default is AutoGenerate,IsolateApps.
validation	Indicates the algorithm used for validation. This attribute can have one of the following four values: - AES - uses the Advanced Encryption Standard (AES) algorithm. - MD5 - uses the Message-Digest 5 (MD5) algorithm. - SHA1 - uses the SHA1 algorithm (the default). - 3DES - uses the Triple DES algorithm.
validationKey	The secret key used by the validation algorithm. This value must either be a hexadecimal string of the appropriate length (based on the value in validation), AutoGenerate, or either value appended with,IsolateApps. Adding IsolateApps instructs ASP.NET to use a unique value for each application. The default is AutoGenerate,IsolateApps.

Table 2: The <machineKey> Element Attributes

A thorough discussion of these encryption and validation options, and the pros and cons of the various algorithms, is beyond the scope of this tutorial. For an in-depth look at these issues, including guidance on what encryption and validation algorithms to use, what key lengths to use, and how best to generate these keys, refer to *Professional ASP.NET 2.0 Security, Membership, and Role Management*.

By default, the keys used for encryption and validation are generated automatically for each application, and these keys are stored in the Local Security Authority (LSA). In short, the default settings guarantee unique keys on a web server-by-web server and application-by-application basis. Consequently, this default behavior will not work for the two following scenarios:

- **Web Farms** - in a web farm scenario, a single web application is hosted on multiple web servers for purposes of scalability and redundancy. Each incoming request is dispatched to a server in the farm, meaning that over the lifetime of a user's session, different servers may be used to handle his various requests. Consequently, each server must use the same encryption and validation keys so that the forms authentication ticket created, encrypted, and validated on one server can be decrypted and validated on a different server in the farm.
- **Cross Application Ticket Sharing** - a single web server may host multiple ASP.NET applications. If you need for these different applications to share a single forms authentication ticket, it is imperative that their encryption and validation keys match up.

When working in a web farm setting or sharing authentication tickets across applications on the same server, you will need to configure the <machineKey> element in the affected applications so that their decryptionKey and validationKey values match up.

While neither of the above scenarios applies to our sample application, we can still specify explicit decryptionKey and validationKey values and define the algorithms to be used. Add a <machineKey> setting to the Web.config file:

```
<configuration>

  <system.web>

  ... Some markup was removed for brevity ...

  <machineKey

  decryption="AES"

  validation="SHA1"

  decryptionKey="1513F567EE75F7FB5AC0AC4D79E1D9F25430E3E2F1BCDD3370BCFC4EFC97A541"

  validationKey="32CBA563F26041EE5B5FE9581076C40618DCC1218F5F447634EDE8624508A129"

  />

  </system.web>

</configuration>
```

For more information check out How To: Configure MachineKey in ASP.NET 2.0.

> **NOTE**
> The decryptionKey and validationKey values were taken from Steve Gibson's Perfect Passwords web page, which generates 64 random hexadecimal characters on each page visit. To lessen the likelihood of these keys making their way into your production applications, you are encouraged to replace the above keys with randomly generated ones from the Perfect Passwords page.

Step 4: Storing Additional User Data in the Ticket

Many web applications display information about or base the page's display on the currently logged on user. For example, a web page might show the user's name and the date she last logged on in the upper corner of every page. The forms authentication ticket stores the currently logged on user's username, but when any other information is needed, the page must go to the user store - typically a database - to lookup the information not stored in the authentication ticket.

With a little bit of code we can store additional user information in the forms authentication ticket. Such data can be expressed through the FormsAuthenticationTicket class's UserData property. This is a useful place to put small amounts of information about the user that is commonly needed. The value specified in the UserData property is included as part of the authentication ticket cookie and, like the other ticket fields, is encrypted and validated based on the forms authentication system's configuration. By default, UserData is an empty string.

In order to store user data in the authentication ticket, we need to write a bit of code in the login page that grabs the user-specific information and stores it in the ticket. Since UserData is a property of type string, the data stored in it must be properly serialized as a string. For example, imagine that our user store included each user's date of birth and the name of their employer, and we wanted to store these two property values in the authentication ticket. We could serialize these values into a string by concatenating the user's date of birth's string with a pipe (|), followed by the employer name. For a user born on August 15, 1974 that works for Northwind Traders, we would assign the UserData property the string: 1974-08-15|Northwind Traders .

Whenever we need to access the data stored in the ticket, we can do so by grabbing the current request's FormsAuthenticationTicket and deserializing the UserData property. In the case of the date of birth and employer name example, we would split the UserData string into two substrings based on the delimiter (|).

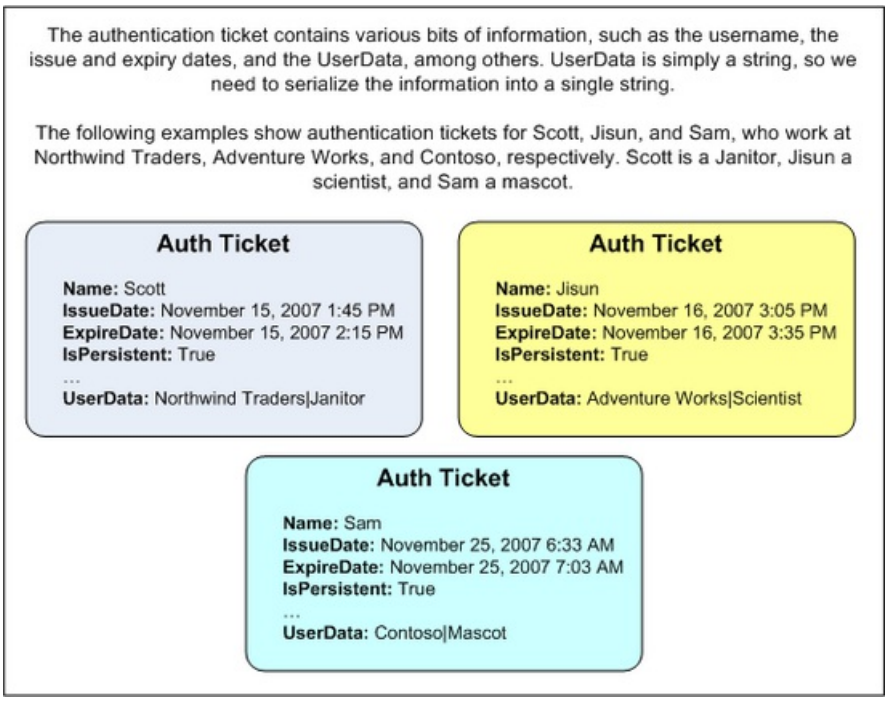


Figure 04: Additional User Information Can Be Stored in the Authentication Ticket (Click to view full-size image)

Writing Information to UserData

Unfortunately, adding user-specific information to a forms authentication ticket is not as straightforward as one might expect. The UserData property of the FormsAuthenticationTicket class is read-only and can only be specified through the FormsAuthenticationTicket class constructor. When specifying the UserData property in the constructor, we also need to provide the ticket's other values: the username, the issue date, the expiration, and so on. When we created the login page in the preceding tutorial, this was all handled for us by the FormsAuthentication class. When adding UserData to the FormsAuthenticationTicket, we will need to write code to replicate much of the functionality already provided by the FormsAuthentication class.

Let's explore the necessary code for working with UserData by updating the Login.aspx page to record additional information about the user to the authentication ticket. Pretend that our user store contains information about the company the user works for and their title, and that we want to capture this information in the authentication ticket. Update the Login.aspx page's LoginButton Click event handler so that the code looks like the following:

```
protected void LoginButton_Click(object sender, EventArgs e)

{

  // Three valid username/password pairs: Scott/password, Jisun/password, and Sam/password.

  string[] users = { "Scott", "Jisun", "Sam" };

  string[] passwords = { "password", "password", "password" };

  string[] companyName = { "Northwind Traders", "Adventure Works", "Contoso" };

  string[] titleAtCompany = { "Janitor", "Scientist", "Mascot" };

  for (int i = 0; i < users.Length; i++)

  {

  bool validUsername = (string.Compare(UserName.Text, users[i], true) == 0);

  bool validPassword = (string.Compare(Password.Text, passwords[i], false) == 0);

  if (validUsername && validPassword)

  {

  // Query the user store to get this user's User Data

  string userDataString = string.Concat(companyName[i], "|", titleAtCompany[i]);

  // Create the cookie that contains the forms authentication ticket

  HttpCookie authCookie = FormsAuthentication.GetAuthCookie(UserName.Text, RememberMe.Checked);

  // Get the FormsAuthenticationTicket out of the encrypted cookie

  FormsAuthenticationTicket ticket = FormsAuthentication.Decrypt(authCookie.Value);

  // Create a new FormsAuthenticationTicket that includes our custom User Data

  FormsAuthenticationTicket newTicket = new FormsAuthenticationTicket(ticket.Version, ticket.Name,
ticket.IssueDate, ticket.Expiration, ticket.IsPersistent, userDataString);

  // Update the authCookie's Value to use the encrypted version of newTicket

  authCookie.Value = FormsAuthentication.Encrypt(newTicket);

  // Manually add the authCookie to the Cookies collection

  Response.Cookies.Add(authCookie);

  // Determine redirect URL and send user there

  string redirUrl = FormsAuthentication.GetRedirectUrl(UserName.Text, RememberMe.Checked);

  Response.Redirect(redirUrl);

  }

  }

  // If we reach here, the user's credentials were invalid

  InvalidCredentialsMessage.Visible = true;

}
```

Let's step through this code one line at a time. The method starts by defining four string arrays: users, passwords, companyName, and titleAtCompany. These arrays hold the usernames, passwords, company names, and titles for the user accounts in the system, of which there are three: Scott, Jisun, and Sam. In a real application, these values would be queried from the user store, not hard-coded in the page's source code.

In the previous tutorial, if the supplied credentials were valid we simply called FormsAuthentication.RedirectFromLoginPage(UserName.Text, RememberMe.Checked), which performed the following steps:

1. Created the forms authentication ticket
2. Wrote the ticket to the appropriate store. For cookies-based authentication tickets, the browser's cookies collection is used; for cookieless authentication tickets, the ticket data is serialized into the URL
3. Redirected the user to the appropriate page

These steps are replicated in the code above. First, the string we will eventually store in the UserData property is formed by combining the company name and the title, delimiting the two values with a pipe character (|).

string userDataString = string.Concat(companyName[i], "|", titleAtCompany[i]);

Next, the FormsAuthentication.GetAuthCookie method is invoked, which creates the authentication ticket, encrypts and validates it according to the configuration settings, and places it in an HttpCookie object.

HttpCookie authCookie = FormsAuthentication.GetAuthCookie(UserName.Text, RememberMe.Checked);

In order to work with the FormAuthenticationTicket embedded within the cookie, we need to call the FormAuthentication class's Decrypt method, passing in the cookie value.

FormsAuthenticationTicket ticket = FormsAuthentication.Decrypt(authCookie.Value);

We then create a *new* FormsAuthenticationTicket instance based on the existing FormsAuthenticationTicket's values. However, this new ticket includes the user-specific information (userDataString).

FormsAuthenticationTicket newTicket = new FormsAuthenticationTicket(ticket.Version, ticket.Name, ticket.IssueDate, ticket.Expiration, ticket.IsPersistent, userDataString);

We then encrypt (and validate) the new FormsAuthenticationTicket instance by calling the Encrypt method, and put this encrypted (and validated) data back into authCookie.

authCookie.Value = FormsAuthentication.Encrypt(newTicket);

Finally, authCookie is added to the Response.Cookies collection and the GetRedirectUrl method is called to determine the appropriate page to send the user.

```
Response.Cookies.Add(authCookie);

string redirUrl = FormsAuthentication.GetRedirectUrl(UserName.Text, RememberMe.Checked);

Response.Redirect(redirUrl);
```

All of this code is needed because the UserData property is read-only and the FormsAuthentication class does not provide any methods for specifying UserData information in its GetAuthCookie, SetAuthCookie, or RedirectFromLoginPage methods.

> **NOTE**
> The code we just examined stores user-specific information in a cookies-based authentication ticket. The classes responsible for serializing the forms authentication ticket to the URL are internal to the .NET Framework. Long story short, you cannot store user data in a cookieless forms authentication ticket.

Accessing the UserData Information

At this point each user's company name and title is stored in the forms authentication ticket's UserData property when they log in. This information can be accessed from the authentication ticket on any page without requiring a trip to the user store. To illustrate how this information can be retrieved from the UserData property, let's update Default.aspx so that its welcome message includes not only the user's name, but also the company they work for and their title.

Currently, Default.aspx contains an AuthenticatedMessagePanel Panel with a Label control named WelcomeBackMessage. This Panel is only displayed to authenticated users. Update the code in Default.aspx's Page_Load event handler so that it looks like the following:

```
protected void Page_Load(object sender, EventArgs e)

{

  if (Request.IsAuthenticated)

  {

  WelcomeBackMessage.Text = "Welcome back, " + User.Identity.Name + "!";

  // Get User Data from FormsAuthenticationTicket and show it in WelcomeBackMessage

  FormsIdentity ident = User.Identity as FormsIdentity;

  if (ident != null)

  {

  FormsAuthenticationTicket ticket = ident.Ticket;

  string userDataString = ticket.UserData;

  // Split on the |

  string[] userDataPieces = userDataString.Split("|".ToCharArray());

  string companyName = userDataPieces[0];

  string titleAtCompany = userDataPieces[1];

  WelcomeBackMessage.Text += string.Format(" You are the {0} of {1}.", titleAtCompany, companyName);

  }

  AuthenticatedMessagePanel.Visible = true;

  AnonymousMessagePanel.Visible = false;

  }

  else

  {

  AuthenticatedMessagePanel.Visible = false;

  AnonymousMessagePanel.Visible = true;

  }

}
```

If Request.IsAuthenticated is true, then the WelcomeBackMessage's Text property is first set to Welcome back,

username. Then, the User.Identity property is cast to a FormsIdentity object so that we can access the underlying FormsAuthenticationTicket. Once we have the FormsAuthenticationTicket, we deserialize the UserData property into the company name and title. This is accomplished by splitting the string on the pipe character. The company name and title are then displayed in the WelcomeBackMessage Label.

Figure 5 shows a screenshot of this display in action. Logging in as Scott displays a welcome back message that includes Scott's company and title.

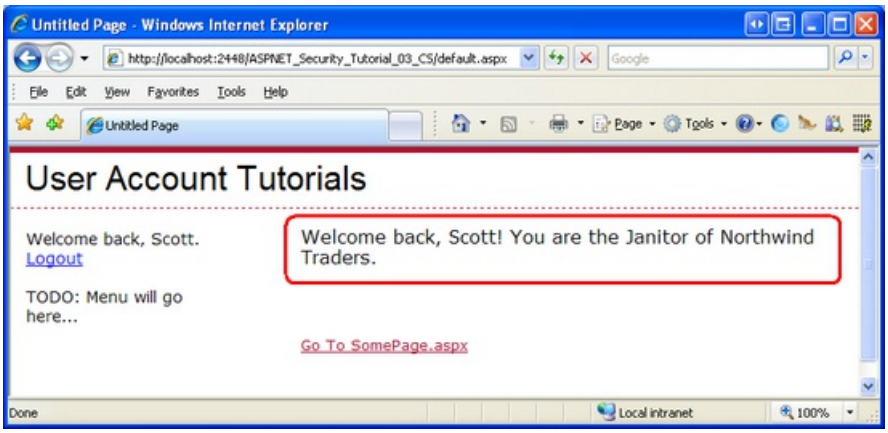


Figure 05: The Currently Logged On User's Company and Title are Displayed (Click to view full-size image)

> **NOTE**
>
> The authentication ticket's UserData property serves as a cache for the user store. Like any cache, it needs to be updated when the underlying data is modified. For example, if there is a web page from which users can update their profile, the fields cached in the UserData property must be refreshed to reflect the changes made by the user.

Step 5: Using a Custom Principal

On each incoming request the FormsAuthenticationModule attempts to authenticate the user. If a non-expired authentication ticket is present, the FormsAuthenticationModule assigns the HttpContext.User property to a new GenericPrincipal object. This GenericPrincipal object has an Identity of type FormsIdentity, which includes a reference to the forms authentication ticket. The GenericPrincipal class contains the bare minimum functionality needed by a class that implements IPrincipal - it just has an Identity property and an IsInRole method.

The principal object has two responsibilities: to indicate what roles the user belongs to and to provide identity information. This is accomplished through the IPrincipal interface's IsInRole(*roleName*) method and Identity property, respectively. The GenericPrincipal class allows for a string array of role names to be specified via its constructor; its IsInRole(*roleName*) method merely checks to see if the passed in *roleName* exists within the string array. When the FormsAuthenticationModule creates the GenericPrincipal, it passes in an empty string array to the GenericPrincipal's constructor. Consequently, any call to IsInRole will always return false.

The GenericPrincipal class meets the needs for most forms based authentication scenarios where roles are not used. For those situations where the default role handling is insufficient or when you need to associate a custom IIdentity object with the user, you can create a custom IPrincipal object during the authentication workflow and assign it to the HttpContext.User property.

> **NOTE**
>
> As we will see in future tutorials, when ASP.NET's Roles framework is enabled it creates a custom principal object of type RolePrincipal and overwrites the forms authentication-created GenericPrincipal object. It does this in order to customize the principal's IsInRole method to interface with the Roles framework's API.

Since we have not concerned ourselves with roles yet, the only reason we would have for creating a custom principal at this juncture would be to associate a custom IIdentity object to the principal. In Step 4 we looked at storing additional user information in the authentication ticket's UserData property, in particular the user's company name and their title. However, the UserData information is only accessible through the authentication ticket and only then as a serialized string, meaning that anytime we want to view the user information stored in the ticket we need to parse the UserData property.

We can improve the developer experience by creating a class that implements IIdentity and includes CompanyName and Title properties. That way, a developer can access the currently logged on user's company name and title directly through the CompanyName and Title properties without needed to know how to parse the UserData property.

Creating the Custom Identity and Principal Classes

For this tutorial, let's create the custom principal and identity objects in the App_Code folder. Start by adding an App_Code folder to your project - right-click on the project name in Solution Explorer, select the Add ASP.NET Folder option, and choose App_Code. The App_Code folder is a special ASP.NET folder that holds class files specific to the website.

> **NOTE**
>
> The App_Code folder should only be used when managing your project through the Website Project Model. If you are using the Web Application Project Model, create a standard folder and add the classes to that. For example, you could add a new folder named Classes, and place your code there.

Next, add two new class files to the App_Code folder, one named CustomIdentity.cs and one named CustomPrincipal.cs.

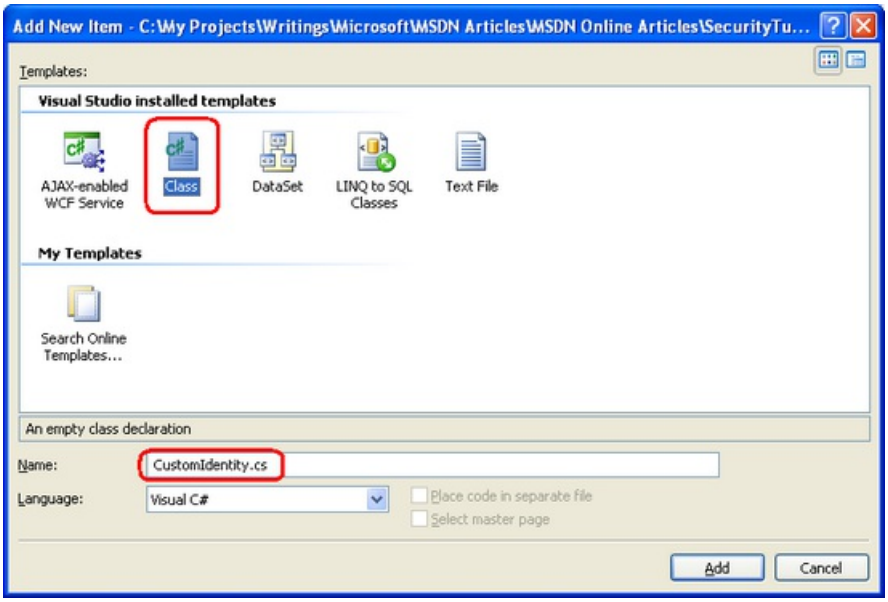


Figure 06: Add the CustomIdentity and CustomPrincipal Classes to Your Project (Click to view full-size image)

The CustomIdentity class is responsible for implementing the IIdentity interface, which defines the AuthenticationType, IsAuthenticated, and Name properties. In addition to those required properties, we are interested in exposing the underlying forms authentication ticket as well as properties for the user's company name and title. Enter the following code into the CustomIdentity class.

```
using System;

using System.Data;

using System.Configuration;
```

```
using System.Web;

using System.Web.Security;

using System.Web.UI;

using System.Web.UI.WebControls;

using System.Web.UI.WebControls.WebParts;

using System.Web.UI.HtmlControls;

public class CustomIdentity : System.Security.Principal.IIdentity

{

  private FormsAuthenticationTicket _ticket;

  public CustomIdentity(FormsAuthenticationTicket ticket)

  {

  _ticket = ticket;

  }

  public string AuthenticationType

  {

  get { return "Custom"; }

  }

  public bool IsAuthenticated

  {

  get { return true; }

  }

  public string Name

  {

  get { return _ticket.Name; }

  }

  public FormsAuthenticationTicket Ticket

  {

  get { return _ticket; }

  }

  public string CompanyName

  {

  get

  {

  string[] userDataPieces = _ticket.UserData.Split("|".ToCharArray());
```

```
    return userDataPieces[0];

    }

    }

    public string Title

    {

    get

    {

    string[] userDataPieces = _ticket.UserData.Split("|".ToCharArray());

    return userDataPieces[1];

    }

    }

  }
```

Note that the class includes a FormsAuthenticationTicket member variable (_ticket) and that this ticket information must be supplied through the constructor. This ticket data is used in returning the identity's Name; its UserData property is parsed to return the values for the CompanyName and Title properties.

Next, create the CustomPrincipal class. Since we are not concerned with roles at this juncture, the CustomPrincipal class's constructor accepts only a CustomIdentity object; its IsInRole method always returns false.

```
using System;

using System.Data;

using System.Configuration;

using System.Web;

using System.Web.Security;

using System.Web.UI;

using System.Web.UI.WebControls;

using System.Web.UI.WebControls.WebParts;

using System.Web.UI.HtmlControls;

public class CustomPrincipal : System.Security.Principal.IPrincipal

{

  private CustomIdentity _identity;

  public CustomPrincipal(CustomIdentity identity)

  {

  _identity = identity;

  }

  public System.Security.Principal.IIdentity Identity

  {

  get { return _identity; }

  }

  public bool IsInRole(string role)

  {

  return false;

  }

}
```

Assigning a CustomPrincipal Object to the Incoming Request's Security Context

We now have a class that extends the default IIdentity specification to include CompanyName and Title properties, as well as a custom principal class that uses the custom identity. We are ready to step into the ASP.NET pipeline and assign our custom principal object to the incoming request's security context.

The ASP.NET pipeline takes an incoming request and processes it through a number of steps. At each step, a particular event is raised, making it possible for developers to tap into the ASP.NET pipeline and modify the request at certain points in its lifecycle. The FormsAuthenticationModule, for example, waits for ASP.NET to raise the AuthenticateRequest event, at which point it inspects the incoming request for an authentication ticket. If an authentication ticket is found, a GenericPrincipal object is created and assigned to the HttpContext.User property.

After the AuthenticateRequest event, the ASP.NET pipeline raises the PostAuthenticateRequest event, which is where we can replace the GenericPrincipal object created by the FormsAuthenticationModule with an instance of

our CustomPrincipal object. Figure 7 depicts this workflow.

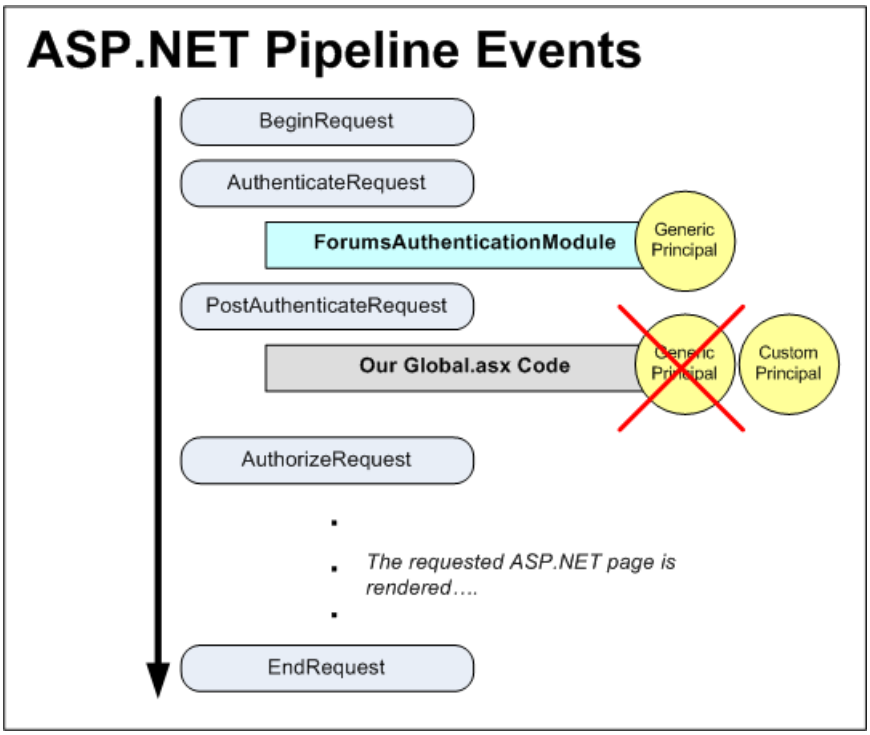


Figure 07: The GenericPrincipal is Replaced by a CustomPrincipal in the PostAuthenticationRequest Event (Click to view full-size image)

In order to execute code in response to an ASP.NET pipeline event, we can either create the appropriate event handler in Global.asax or create our own HTTP Module. For this tutorial let's create the event handler in Global.asax. Start by adding Global.asax to your website. Right-click on the project name in Solution Explorer and add an item of type Global Application Class named Global.asax.

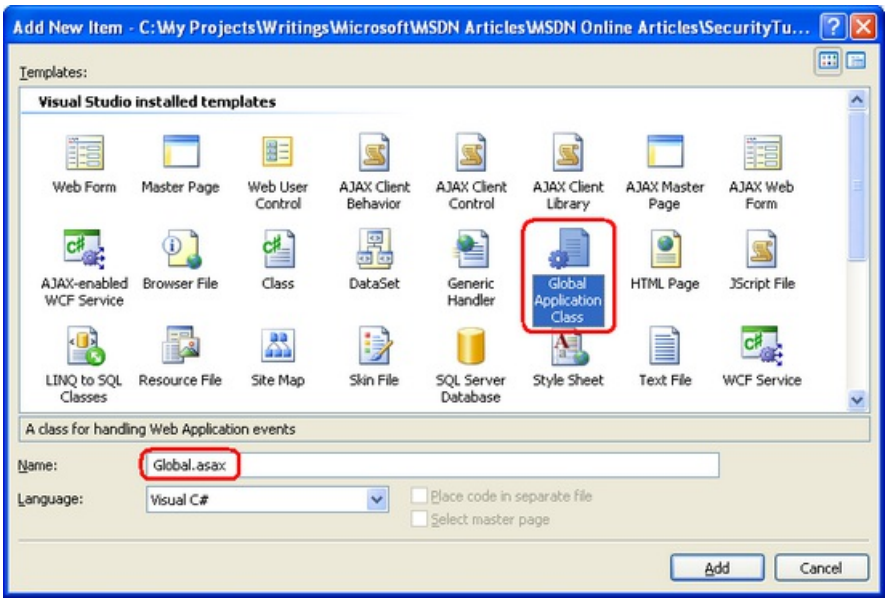


Figure 08: Add a Global.asax File to Your Website (Click to view full-size image)

The default Global.asax template includes event handlers for a number of the ASP.NET pipeline events, including the Start, End and Error event, among others. Feel free to remove these event handlers, as we do not need them for this application. The event we are interested in is PostAuthenticateRequest. Update your Global.asax file so its markup looks similar to the following:

```
<%@ Application Language="C#" %>

<%@ Import Namespace="System.Security.Principal" %>

<%@ Import Namespace="System.Threading" %>

<script runat="server">

  void Application_OnPostAuthenticateRequest(object sender, EventArgs e)

  {

  // Get a reference to the current User

  IPrincipal usr = HttpContext.Current.User;

  // If we are dealing with an authenticated forms authentication request

  if (usr.Identity.IsAuthenticated && usr.Identity.AuthenticationType == "Forms")

  {

  FormsIdentity fIdent = usr.Identity as FormsIdentity;

  // Create a CustomIdentity based on the FormsAuthenticationTicket

  CustomIdentity ci = new CustomIdentity(fIdent.Ticket);

  // Create the CustomPrincipal

  CustomPrincipal p = new CustomPrincipal(ci);

  // Attach the CustomPrincipal to HttpContext.User and Thread.CurrentPrincipal

  HttpContext.Current.User = p;

  Thread.CurrentPrincipal = p;

  }

  }

</script>
```

The Application_OnPostAuthenticateRequest method executes each time the ASP.NET runtime raises the PostAuthenticateRequest event, which happens once on each incoming page request. The event handler starts by checking to see if the user is authenticated and was authenticated via forms authentication. If so, a new CustomIdentity object is created and passed the current request's authentication ticket in its constructor. Following that, a CustomPrincipal object is created and passed the just-created CustomIdentity object in its constructor. Finally, the current request's security context is assigned to the newly created CustomPrincipal object.

Note that the last step - associating the CustomPrincipal object with the request's security context - assigns the principal to two properties: HttpContext.User and Thread.CurrentPrincipal. These two assignments are necessary because of the way security contexts are handled in ASP.NET. The .NET Framework associates a security context with each running thread; this information is available as an IPrincipal object through the Thread object's CurrentPrincipal property. What is a confusing is that ASP.NET has its own security context information (HttpContext.User).

In certain scenarios, the Thread.CurrentPrincipal property is examined when determining the security context; in other scenarios, HttpContext.User is used. For example, there are security features in .NET that allow developers to declaratively state what users or roles can instantiate a class or invoke specific methods (see Adding

Authorization Rules to Business and Data Layers Using PrincipalPermissionAttributes). Underneath the covers, these declarative techniques determine the security context via the Thread.CurrentPrincipal property.

In other scenarios, the HttpContext.User property is used. For example, in the previous tutorial we used this property to display the currently logged on user's username. Clearly, then, it is imperative that the security context information in the Thread.CurrentPrincipal and HttpContext.User properties match up.

The ASP.NET runtime automatically syncs these property values for us. However, this synchronization occurs after the AuthenticateRequest event, but *before* the PostAuthenticateRequest event. Consequently, when adding a custom principal in the PostAuthenticateRequest event we need to be certain to manually assign the Thread.CurrentPrincipal or else Thread.CurrentPrincipal and HttpContext.User will be out of sync. See Context.User vs. Thread.CurrentPrincipal for a more detailed discussion on this issue.

Accessing the CompanyName and Title Properties

Whenever a request arrives and is dispatched to the ASP.NET engine, the Application_OnPostAuthenticateRequest event handler in Global.asax will fire. If the request has been successfully authenticated by the FormsAuthenticationModule, the event handler will create a new CustomPrincipal object with a CustomIdentity object based on the forms authentication ticket. With this logic in place, accessing information about the currently logged on user's company name and title is incredibly straightforward.

Return to the Page_Load event handler in Default.aspx, where in Step 4 we wrote code to retrieve the form authentication ticket and parse the UserData property in order to display the user's company name and title. With the CustomPrincipal and CustomIdentity objects in use now, there's no need to parse the values out of the ticket's UserData property. Instead, simply get a reference to the CustomIdentity object and use its CompanyName and Title properties:

```
CustomIdentity ident = User.Identity as CustomIdentity;

if (ident != null)

  WelcomeBackMessage.Text += string.Format(" You are the {0} of {1}.", ident.Title, ident.CompanyName);
```

Summary

In this tutorial we examined how to customize the forms authentication system's settings via Web.config. We looked at how the authentication ticket's expiration is handled and how the encryption and validation safeguards are used to protect the ticket from inspection and modification. Finally, we discussed using the authentication ticket's UserData property to store additional user information in the ticket itself, and how to use custom principal and identity objects to expose this information in a more developer-friendly manner.

This tutorial concludes our examination of forms authentication in ASP.NET. The next tutorial starts our journey into the Membership framework.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Dissecting Forms Authentication
- Explained: Forms Authentication in ASP.NET 2.0
- How To: Protect Forms Authentication in ASP.NET 2.0
- Professional ASP.NET 2.0 Security, Membership, and Role Management (ISBN: 978-0-7645-9698-8)
- Securing Login Controls
- The <authentication> Element

- The <forms> Element for <authentication>
- The <machineKey> Element
- Understanding the Forms Authentication Ticket and Cookie

Video Training on Topics Contained in this Tutorial

- How to Change the Forms Authentication Properties
- How to Setup and Use Cookie-less Authentication in an ASP.NET Application
- ASP Forms Login Relocation
- Forms Login Custom Key Configuration
- Add Custom Data to the Authentication Method
- Use Custom Principal Objects

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Alicja Maziarz. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Security Basics and ASP.NET Support (VB)

2/20/2020 • 16 minutes to read • Edit Online

by Scott Mitchell

Download PDF

This is the first tutorial in a series of tutorials that will explore techniques for authenticating visitors through a web form, authorizing access to particular pages and functionality, and managing user accounts in an ASP.NET application.

Introduction

What is the one thing forums, eCommerce sites, online email websites, portal websites, and social network sites all have in common? They all offer *user accounts*. Sites that offer user accounts must provide a number of services. At a minimum, new visitors need to be able to create an account and returning visitors must be able to log in. Such web applications can make decisions based on the logged in user: some pages or actions might be restricted to only logged in users, or to a certain subset of users; other pages might show information specific to the logged in user, or might show more or less information, depending on what user is viewing the page.

This is the first tutorial in a series of tutorials that will explore techniques for authenticating visitors through a web form, authorizing access to particular pages and functionality, and managing user accounts in an ASP.NET application. Over the course of these tutorials we will examine how to:

- Identify and log users in to a website
- Use ASP.NET's Membership framework to manage user accounts
- Create, update, and delete user accounts
- Limit access to a web page, directory, or specific functionality based on the logged in user
- Use ASP.NET's Roles framework to associate user accounts with roles
- Manage user roles
- Limit access to a web page, directory, or specific functionality based on the logged in user's role
- Customize and extend ASP.NET's security Web controls

These tutorials are geared to be concise and provide step-by-step instructions with plenty of screen shots to walk you through the process visually. Each tutorial is available in C# and Visual Basic versions and includes a download of the complete code used. (This first tutorial focuses on security concepts from a high-level viewpoint and therefore does not contain any associated code.)

In this tutorial we will discuss important security concepts and what facilities are available in ASP.NET to assist in implementing forms authentication, authorization, user accounts, and roles. Let's get started!

> **NOTE**
>
> Security is an important aspect of any application that spans physical, technological, and policy decisions and requires a high degree of planning and domain knowledge. This tutorial series is not intended as a guide for developing secure web applications. Rather, it focuses specifically on forms authentication, authorization, user accounts, and roles. While some security concepts revolving around these issues are discussed in this series, others are left unexplored.

Authentication, Authorization, User Accounts, and Roles

Authentication, authorization, user accounts, and roles are four terms that will be used very often throughout this tutorial series, so I'd like to take a quick moment to define these terms within the context of web security. In a client-server model, such as the Internet, there are many scenarios in which the server needs to identify the client making the request. *Authentication* is the process of ascertaining the client's identity. A client who has been successfully identified is said to be *authenticated*. An unidentified client is said to be *unauthenticated* or *anonymous*.

Secure authentication systems involve at least one of the following three facets: something you know, something you have, or something you are. Most web applications rely on something the client knows, such as a password or a PIN. The information used to identify a user - her username and password, for example - are referred to as *credentials*. This tutorial series focuses on *forms authentication*, which is an authentication model where users log in to the site by providing their credentials in a web page form. We have all experienced this type of authentication before. Go to any eCommerce site. When you are ready to check out you are asked to log in by entering your username and password into textboxes on a web page.

In addition to identifying clients, a server may need to limit what resources or functionalities are accessible depending on the client making the request. *Authorization* is the process of determining whether a particular user has the authority to access a specific resource or functionality.

A *user account* is a store for persisting information about a particular user. User accounts must minimally include information that uniquely identifies the user, such as the user's login name and password. Along with this essential information, user accounts may include things like: the user's email address; the date and time the account was created; the date and time they last logged in; first and last name; phone number; and mailing address. When using forms authentication, user account information is typically stored in a relational database like Microsoft SQL Server.

Web applications that support user accounts may optionally group users into *roles*. A role is simply a label that is applied to a user and provides an abstraction for defining authorization rules and page-level functionality. For example, a website might include an Administrator role with authorization rules that prohibit anyone but an Administrator to access a particular set of web pages. Moreover, a variety of pages that are accessible to all users (including non-Administrators) might display additional data or offer extra functionality when visited by users in the Administrators role. Using roles, we can define these authorization rules on a role-by-role basis rather than user-by-user.

Authenticating Users in an ASP.NET Application

When a user enters a URL into their browser's address window or clicks on a link, the browser makes a Hypertext Transfer Protocol (HTTP) request to the web server for the specified content, be it an ASP.NET page, an image, a JavaScript file, or any other type of content. The web server is tasked with returning the requested content. In doing so, it must determine a number of things about the request, including who made the request and whether the identity is authorized to retrieve the requested content.

By default, browsers send HTTP requests that lack any sort of identification information. But if the browser does include authentication information then the web server starts the authentication workflow, which attempts to identify the client making the request. The steps of the authentication workflow depend on the type of authentication being used by the web application. ASP.NET supports three types of authentication: Windows, Passport, and forms. This tutorial series focuses on forms authentication, but let's take a minute to compare and contrast Windows authentication user stores and workflow.

Authentication via Windows Authentication

The Windows authentication workflow uses one of the following authentication techniques:

- Basic authentication
- Digest authentication
- Windows Integrated Authentication

All three techniques work in roughly the same way: when an unauthorized, anonymous request arrives, the web server sends back an HTTP response that indicates that authorization is required to continue. The browser then displays a modal dialog box that prompts the user for their username and password (see Figure 1). This information is then sent back to the web server via an HTTP header.

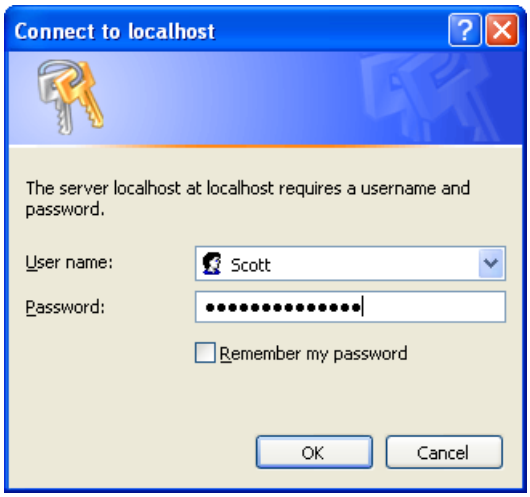


Figure 1: A Modal Dialog Box Prompts the User for His Credentials

The supplied credentials are validated against the web server's Windows User Store. This means that each authenticated user in your web application must have a Windows account in your organization. This is commonplace in intranet scenarios. In fact, when using Windows Integrated Authentication in an intranet setting, the browser automatically provides the web server with the credentials used to log on to the network, thereby suppressing the dialog box shown in Figure 1. While Windows authentication is great for intranet applications, it is usually unfeasible for Internet applications since you do not want to create Windows accounts for each and every user who signs up at your site.

Authentication via Forms Authentication

Forms authentication, on the other hand, is ideal for Internet web applications. Recall that forms authentication identifies the user by prompting them to enter their credentials through a web form. Consequently, when a user attempts to access an unauthorized resource, they are automatically redirected to the login page where they can enter their credentials. The submitted credentials are then validated against a custom user store - usually a database.

After verifying the submitted credentials, a *forms authentication ticket* is created for the user. This ticket indicates that the user has been authenticated and includes identifying information, such as the username. The forms authentication ticket is (typically) stored as a cookie on the client computer. Therefore, subsequent visits to the website include the forms authentication ticket in the HTTP request, thereby enabling the web application to identify the user once they have logged in.

Figure 2 illustrates the forms authentication workflow from a high-level vantage point. Notice how the authentication and authorization pieces in ASP.NET act as two separate entities. The forms authentication system identifies the user (or reports that they are anonymous). The authorization system is what determines whether the user has access to the requested resource. If the user is unauthorized (as they are in Figure 2 when attempting to anonymously visit ProtectedPage.aspx), the authorization system reports that the user is denied, causing the forms authentication system to automatically redirect the user to the login page.

Once the user has successfully logged in, subsequent HTTP requests include the forms authentication ticket. The forms authentication system merely identifies the user - it is the authorization system that determines whether the user can access the requested resource.

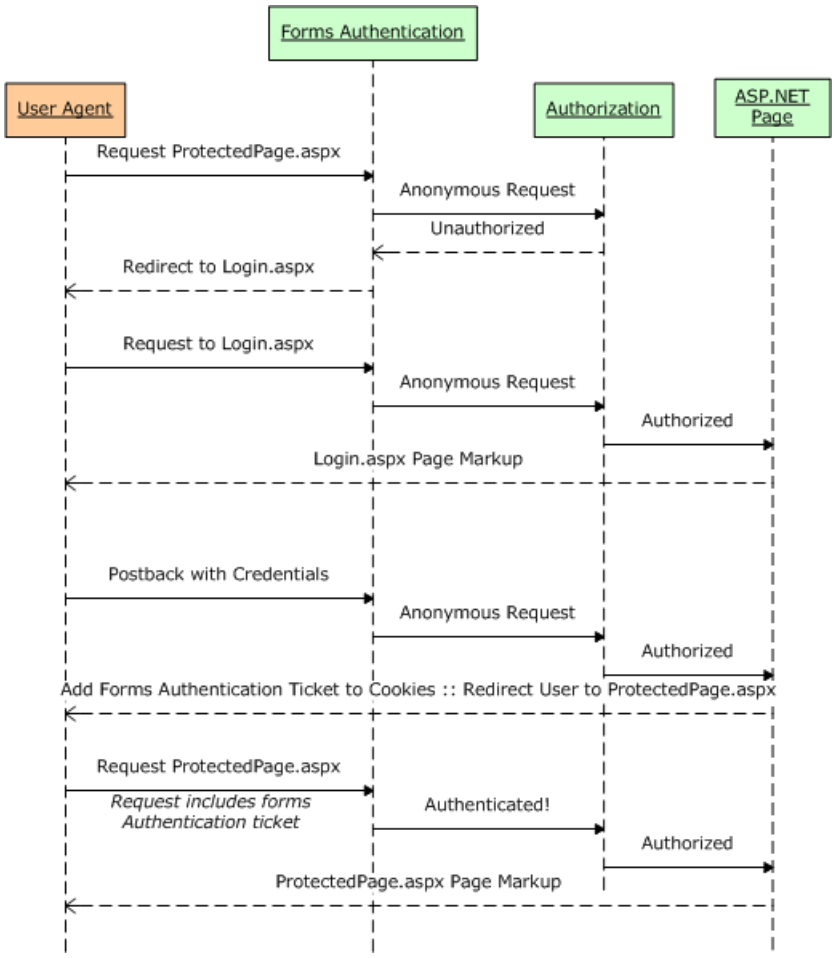


Figure 2: The Forms Authentication Workflow

We will dig into forms authentication in much greater detail in the next two tutorials,An Overview of Forms Authentication and Forms Authentication Configuration and Advanced Topics. For more on ASP.NET's authentication options, see ASP.NET Authentication.

Limiting Access to Web Pages, Directories, and Page Functionality

ASP.NET includes two ways to determine whether a particular user has authority to access a specific file or directory:

- **File authorization** - since ASP.NET pages and web services are implemented as files that reside on the web server's file system, access to these files can be specified through Access Control Lists (ACLs). File authorization is most commonly used with Windows authentication because ACLs are permissions that apply to Windows accounts. When using forms authentication, all operating system- and file system-level requests are executed by the same Windows account, regardless of the user visiting the site.
- **URL authorization**- with URL authorization, the page developer specifies authorization rules in Web.config. These authorization rules specify what users or roles are allowed to access or are denied from accessing certain pages or directories in the application.

File authorization and URL authorization define authorization rules for accessing a particular ASP.NET page or for all ASP.NET pages in a particular directory. Using these techniques we can instruct ASP.NET to deny requests to a particular page for a particular user, or allow access to a set of users and deny access to everyone else. What about scenarios where all of the users can access the page, but the page's functionality depends on the user? For example, many sites that support user accounts have pages that display different content or data for authenticated users versus anonymous users. An anonymous user might see a link to log in to the site, whereas an authenticated user would instead see a message like, Welcome back, *Username* along with a link to log out. Another example: when viewing an item at an auction site you see different information depending on whether you are a bidder or the one auctioning the item.

Such page-level adjustments can be accomplished declaratively or programmatically. To show different content for anonymous than authenticated users, simply drag a LoginView control onto your page and enter the appropriate content into its AnonymousTemplate and LoggedInTemplate templates. Alternatively, you can programmatically determine whether the current request is authenticated, who the user is, and what roles they belong to (if any). You can use this information to then show or hide columns in a grid or Panels on the page.

This series includes three tutorials that focus on authorization. ***User-Based Authorization***examines how to limit access to a page or pages in a directory for specific user accounts; ***Role-Based Authorization*** looks at supplying authorization rules at the role level; lastly, the ***Displaying Content Based on the Currently Logged In User*** tutorial explores modifying a particular page's content and functionality based on the user visiting the page. For more on ASP.NET's authorization options, see ASP.NET Authorization.

User Accounts and Roles

ASP.NET's forms authentication provides an infrastructure for users to log in to a site and have their authenticated state remembered across page visits. And URL authorization offers a framework for limiting access to specific files or folders in an ASP.NET application. Neither feature, however, supplies a means for storing user account information or managing roles.

Prior to ASP.NET 2.0, developers were responsible for creating their own user and role stores. They were also on the hook for designing the user interfaces and writing the code for essential user account-related pages like the login page and the page to create a new account, among others. Without any built-in user account framework in ASP.NET, each developer implementing user accounts had to arrive at his own design decisions on questions like, How do I store passwords or other sensitive information? and What guidelines should I impose regarding password length and strength?

Today, implementing user accounts in an ASP.NET application is much simpler thanks to the *Membership framework* and the built-in Login Web controls. The Membership framework is a handful of classes in the System.Web.Security namespace that provide functionality for performing essential user account-related tasks. The key class in the Membership framework is the Membership class, which has methods like:

- CreateUser
- DeleteUser
- GetAllUsers
- GetUser
- UpdateUser
- ValidateUser

The Membership framework uses the provider model, which cleanly separates the Membership framework's API from its implementation. This enables developers to use a common API, but empowers them to use an implementation that meets their application's custom needs. In short, the Membership class defines the essential functionality of the framework (the methods, properties, and events), but does not actually supply any implementation details. Instead, the methods of the Membership class invoke the configured provider, which is what performs the actual work. For example, when the Membership class's CreateUser method is invoked, the Membership class doesn't know the details of the user store. It doesn't know if users are being maintained in a database or in an XML file or in some other store. The Membership class examines the web application's configuration to determine what provider to delegate the call to, and that provider class is responsible for actually creating the new user account in the appropriate user store. This interaction is illustrated in Figure 3.

Microsoft ships two Membership provider classes in the .NET Framework:

- ActiveDirectoryMembershipProvider - implements the Membership API in Active Directory and Active Directory Application Mode (ADAM) servers.
- SqlMembershipProvider - implements the Membership API in a SQL Server database.

This tutorial series focuses exclusively on the SqlMembershipProvider.

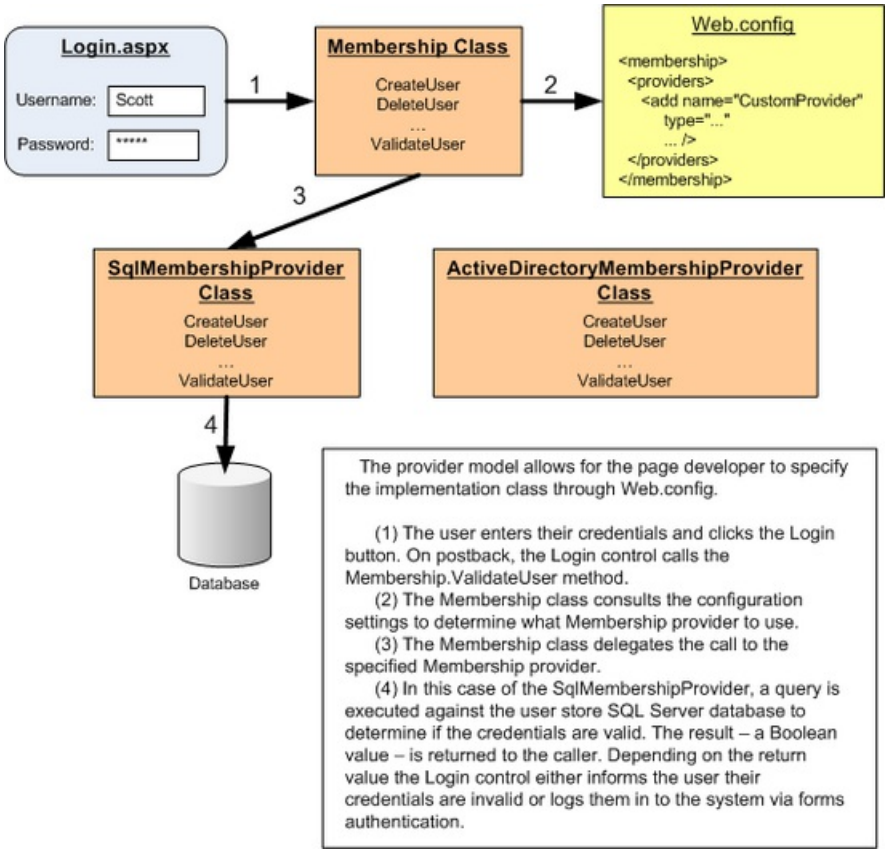


Figure 03: The Provider Model Enables Different Implementations to be Seamlessly Plugged Into the Framework (Click to view full-size image)

The benefit of the provider model is that alternative implementations can be developed by Microsoft, third-party vendors, or individual developers and seamlessly plugged into the Membership framework. For example, Microsoft has released a Membership provider for Microsoft Access databases. For more information on the Membership providers, refer to the Provider Toolkit, which includes a walkthrough of the Membership providers, sample custom providers, over 100 pages of documentation on the provider model, and the complete source code for the built-in Membership providers (namely, ActiveDirectoryMembershipProvider and SqlMembershipProvider).

ASP.NET 2.0 also introduced the Roles framework. Like the Membership framework, the Roles framework is built atop the provider model. Its API is exposed via the Roles class and the .NET Framework ships with three provider classes:

- AuthorizationStoreRoleProvider - manages role information in an authorization-manager policy store, such as Active Directory or ADAM.
- SqlRoleProvider - implements roles in a SQL Server database.
- WindowsTokenRoleProvider - associates role information based on the visitor's Windows group. This method is typically used with Windows authentication.

This tutorial series focuses exclusively on the SqlRoleProvider.

Since the provider model includes a single forward-facing API (the Membership and Roles classes), it is possible to build functionality around that API without having to worry about the implementation details - those are handled by the providers selected by the page developer. This unified API allows for Microsoft and third-party vendors to build Web controls that interface with the Membership and Roles frameworks. ASP.NET ships with a number of Login Web controls for implementing common user account user interfaces. For example, the Login control prompts a user for their credentials, validates them, and then logs them in via forms authentication. The LoginView control offers templates for displaying different markup to anonymous users versus authenticated

users, or different markup based on the user's role. And the CreateUserWizard control provides a step-by-step user interface for creating a new user account.

Underneath the covers the various Login controls interact with the Membership and Roles frameworks. Most Login controls can be implemented without having to write a single line of code. We will examine these controls in greater detail in future tutorials, including techniques for extending and customizing their functionality.

Summary

All web applications that support user accounts require similar features, such as: the ability for users to log in and have their log in status remembered across page visits; a web page for new visitors to create an account; and the ability to the page developer to specify what resource, data, and functionality are available to what users or roles. The tasks of authenticating and authorizing users and of managing user accounts and roles is remarkably easy to accomplish in ASP.NET applications thanks to forms authentication, URL authorization, and the Membership and Roles frameworks.

Over the course of the next several tutorials we will examine these aspects by building a working web application from the ground up in a step-by-step fashion. In the next two tutorial we will explore forms authentication in detail. We will see the forms authentication workflow in action, dissect the forms authentication ticket, discuss security concerns, and see how to configure the forms authentication system - all while building a web application that allows visitors to log in and log out.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET 2.0 Membership, Roles, Forms Authentication, and Security Resources
- ASP.NET 2.0 Security Guidelines
- ASP.NET Authentication
- ASP.NET Authorization
- ASP.NET Login Controls Overview
- Examining ASP.NET 2.0's Membership, Roles, and Profile
- How Do I: Secure My Site Using Membership and Roles? (Video)
- Introduction to Membership
- MSDN Security Developer Center
- Professional ASP.NET 2.0 Security, Membership, and Role Management (ISBN: 978-0-7645-9698-8)
- Provider Toolkit

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial include Alicja Maziarz, John Suru, and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

An Overview of Forms Authentication (VB)

2/20/2020 • 37 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> In this tutorial we will turn from mere discussion to implementation; in particular, we will look at implementing forms authentication. The web application we start constructing in this tutorial will continue to be built upon in subsequent tutorials, as we move from simple forms authentication to membership and roles.
>
> Please see this video for more information on this topic: Using Basic Forms Authentication in ASP.NET.

Introduction

In the preceding tutorial we discussed the various authentication, authorization, and user account options provided by ASP.NET. In this tutorial we will turn from mere discussion to implementation; in particular, we will look at implementing forms authentication. The web application we start constructing in this tutorial will continue to be built upon in subsequent tutorials, as we move from simple forms authentication to membership and roles.

This tutorial begins with an in-depth look at the forms authentication workflow, a topic we touched upon in the previous tutorial. Following that, we will create an ASP.NET website through which to demo the concepts of forms authentication. Next, we will configure the site to use forms authentication, create a simple login page, and see how to determine, in code, whether a user is authenticated and, if so, the username they logged in with.

Understanding the forms authentication workflow, enabling it in a web application, and creating the login and logoff pages are all vital steps in building an ASP.NET application that supports user accounts and authenticates users through a web page. Because of this - and because these tutorials build upon one another - I would encourage you to work through this tutorial in full before moving on to the next one even if you already have had experience configuring forms authentication in past projects.

Understanding the Forms Authentication Workflow

When the ASP.NET runtime processes a request for an ASP.NET resource, such as an ASP.NET page or ASP.NET Web service, the request raises a number of events during its lifecycle. There are events raised at the very beginning and very end of the request, ones raised when the request is being authenticated and authorized, an event raised in the case of an unhandled exception, and so forth. To see a complete listing of the events, refer to the HttpApplication object's events.

HTTP Modules are managed classes whose code is executed in response to a particular event in the request lifecycle. ASP.NET ships with a number of HTTP Modules that perform essential tasks behind the scenes. Two built-in HTTP Modules that are especially relevant to our discussion are:

- **FormsAuthenticationModule** - authenticates the user by inspecting the forms authentication ticket, which is typically included in the user's cookies collection. If no forms authentication ticket is present, the user is anonymous.
- **UrlAuthorizationModule** - determines whether or not the current user is authorized to access the requested URL. This module determines the authority by consulting the authorization rules specified in the application's configuration files. ASP.NET also includes the FileAuthorizationModule that determines authority

by consulting the requested file(s) ACLs.

The FormsAuthenticationModule attempts to authenticate the user prior to the UrlAuthorizationModule (and FileAuthorizationModule) executing. If the user making the request is not authorized to access the requested resource, the authorization module terminates the request and returns an HTTP 401 Unauthorized status. In Windows authentication scenarios, the HTTP 401 status is returned to the browser. This status code causes the browser to prompt the user for their credentials via a modal dialog box. With forms authentication, however, the HTTP 401 Unauthorized status is never sent to the browser because the FormsAuthenticationModule detects this status and modifies it to redirect the user to the login page instead (via an HTTP 302 Redirect status).

The login page's responsibility is to determine if the user's credentials are valid and, if so, to create a forms authentication ticket and redirect the user back to the page they were attempting to visit. The authentication ticket is included in subsequent requests to the pages on the website, which the FormsAuthenticationModule uses to identify the user.

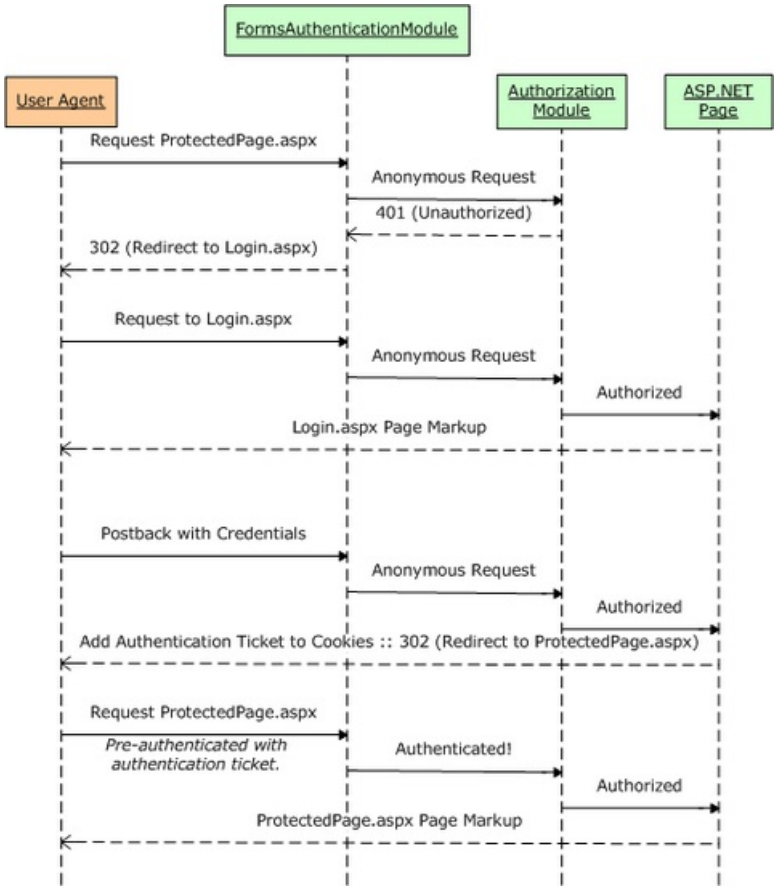


Figure 01: The Forms Authentication Workflow(Click to view full-size image)

Remembering the Authentication Ticket Across Page Visits

After logging in, the forms authentication ticket must be sent back to the web server on each request so that the user remains logged in as they browse the site. This is typically accomplished by placing the authentication ticket in the user's cookies collection. Cookies are small text files that reside on the user's computer and are transmitted in the HTTP headers on each request to the website that created the cookie. Therefore, once the forms authentication ticket has been created and stored in the browser's cookies, each subsequent visit to that site sends the authentication ticket along with the request, thereby identifying the user.

NOTE

The demo web application used in each tutorial is available as a download. This downloadable application was created with Visual Web Developer 2008 targeted for the .NET Framework version 3.5. Since the application is targeted for .NET 3.5, its Web.config file includes additional, 3.5-specific configuration elements. Long story short, if you have yet to install .NET 3.5 on your computer then the downloadable web application will not work without first removing the 3.5-specific markup from Web.config.

One aspect of cookies is their expiration, which is the date and time at which the browser discards the cookie. When the forms authentication cookie expires, the user can no longer be authenticated and therefore become anonymous. When a user is visiting from a public terminal, chances are they want their authentication ticket to expire when they close their browser. When visiting from home, however, that same user might want the authentication ticket to be remembered across browser restarts so that they do not have to re-log in each time they visit the site. This decision is often made by the user in the form of a Remember me checkbox on the login page. In Step 3 we will examine how to implement a Remember me checkbox in the login page. The following tutorial addresses the authentication ticket timeout settings in detail.

NOTE

It is possible that the user agent used to log on to the website may not support cookies. In such a case, ASP.NET can use cookieless forms authentication tickets. In this mode, the authentication ticket is encoded into the URL. We will look at when cookieless authentication tickets are used and how they are created and managed in the next tutorial.

The Scope of Forms Authentication

The FormsAuthenticationModule is managed code that is part of the ASP.NET runtime. Prior to version 7 of Microsoft's Internet Information Services (IIS) web server, there was a distinct barrier between IIS's HTTP pipeline and the ASP.NET runtime's pipeline. In short, in IIS 6 and earlier, the FormsAuthenticationModule only executes when a request is delegated from IIS to the ASP.NET runtime. By default, IIS processes static content itself - like HTML pages and CSS and image files - and only hands off requests to the ASP.NET runtime when a page with an extension of .aspx, .asmx, or .ashx is requested.

IIS 7, however, allows for integrated IIS and ASP.NET pipelines. With a few configuration settings you can setup IIS 7 to invoke the FormsAuthenticationModule for *all* requests. Furthermore, with IIS 7 you can define URL authorization rules for files of any type. For more information, see Changes Between IIS6 and IIS7 Security, Your Web Platform Security, and Understanding IIS7 URL Authorization.

Long story short, in versions prior to IIS 7, you can only use forms authentication to protect resources handled by the ASP.NET runtime. Likewise, URL authorization rules are only applied to resources handled by the ASP.NET runtime. But with IIS 7 it is possible to integrate the FormsAuthenticationModule and UrlAuthorizationModule into IIS's HTTP pipeline, thereby extending this functionality to all requests.

Step 1: Creating an ASP.NET Website for this Tutorial Series

In order to reach the widest possible audience, the ASP.NET website we will be building throughout this series will be created with Microsoft's free version of Visual Studio 2008, Visual Web Developer 2008. We will implement the SqlMembershipProvider user store in a Microsoft SQL Server 2005 Express Edition database. If you are using Visual Studio 2005 or a different edition of Visual Studio 2008 or SQL Server, don't worry - the steps will be nearly identical and any non-trivial differences will be pointed out.

Before we can configure forms authentication, we first need an ASP.NET website. Start by creating a new file system-based ASP.NET website. To accomplish this, launch Visual Web Developer and then go to the File menu and choose New Web Site, displaying the New Web Site dialog box. Choose the ASP.NET Web Site template, set the Location drop-down list to File System, choose a folder to place the web site, and set the language to VB.

This will create a new web site with a Default.aspx ASP.NET page, an App_Data folder, and a Web.config file.

> **NOTE**
>
> Visual Studio supports two modes of project management: Web Site Projects and Web Application Projects. Web Site Projects lack a project file, whereas Web Application Projects mimic the project architecture in Visual Studio .NET 2002/2003 - they include a project file and compile the project's source code into a single assembly, which is placed in the /bin folder. Visual Studio 2005 initially only supported Web Site Projects, although the Web Application Project model was reintroduced with Service Pack 1; Visual Studio 2008 offers both project models. The Visual Web Developer 2005 and 2008 editions, however, only support Web Site Projects. I will be using the Web Site Project model. If you are using a non-Express edition and want to use the Web Application Project model instead, feel free to do so but be aware that there may be some discrepancies between what you see on your screen and the steps you must take versus the screen shots shown and instructions provided in these tutorials.

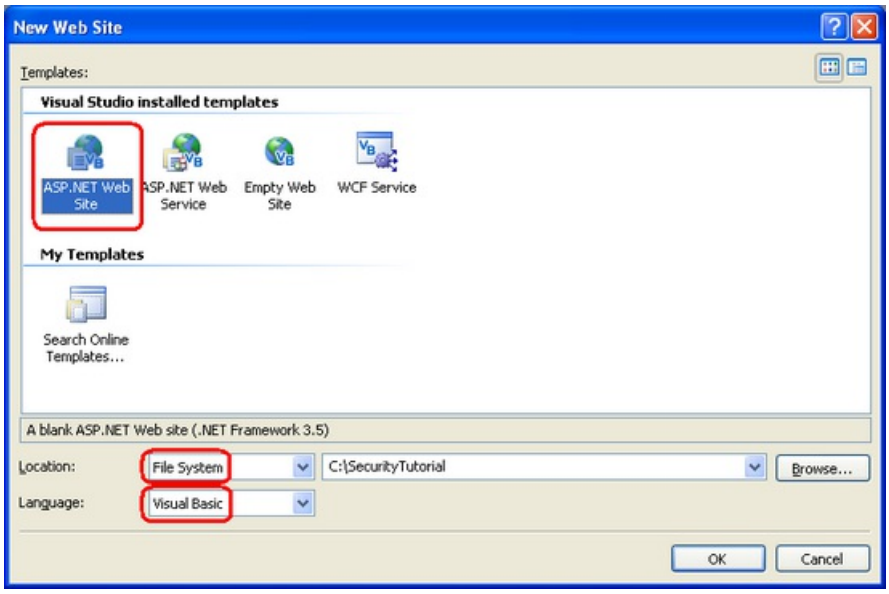


Figure 02: Create a New File System-Based Web Site(Click to view full-size image)

Adding a Master Page

Next, add a new Master Page to the site in the root directory named Site.master. Master pages enable a page developer to define a site-wide template that can be applied to ASP.NET pages. The main benefit of master pages is that the site's overall appearance can be defined in a single location, thereby making it easy to update or tweak the site's layout.

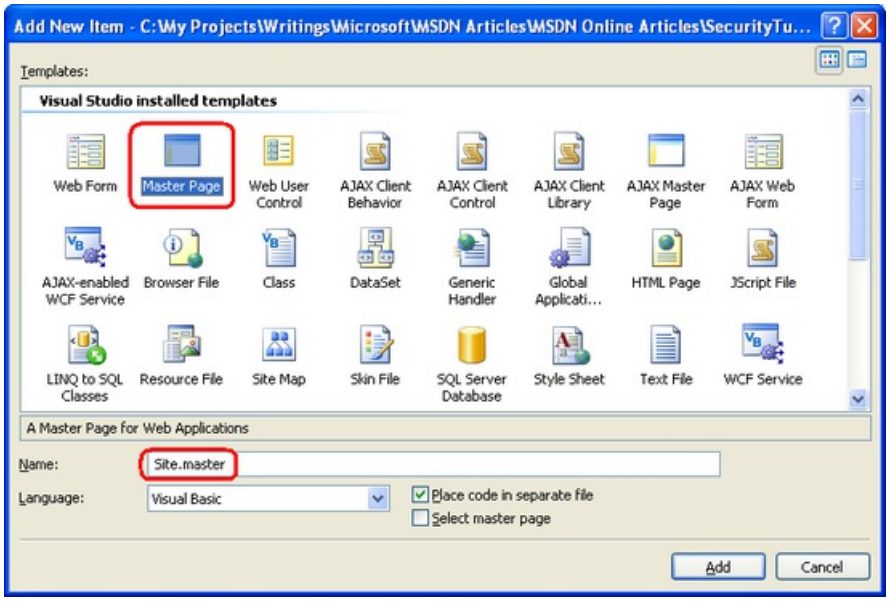


Figure 03: Add a Master Page Named Site.master to the Website(Click to view full-size image)

Define the site-wide page layout here in the master page. You can use the Design view and add whatever Layout or Web controls you need, or you can manually add the markup by hand in the Source view. I structured my master page's layout to mimic the layout used in my *Working with Data in ASP.NET 2.0* tutorial series (see Figure 4). The master page uses cascading style sheets for positioning and styles with the CSS settings defined in the file Style.css (which is included in this tutorial's associated download). While you cannot tell from the markup shown below, the CSS rules are defined such that the navigation 's content is absolutely positioned so that it appears on the left and has a fixed width of 200 pixels.

```
<%@ Master Language="VB" CodeFile="Site.master.vb" Inherits="Site" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
 <head id="Head1" runat="server">
 <title>Forms Authentication, Authorization, and User Accounts</title>
 <link href="Styles.css" rel="stylesheet" type="text/css" />
 </head>
 <body>
 <div id="wrapper">
 <form id="form1" runat="server">
 <div id="header">
 <span class="title">User Account Tutorials</span>
 </div>
 <div id="content">
 <asp:contentplaceholder id="MainContent" runat="server">
 <!-- Page-specific content will go here... -->
 </asp:contentplaceholder>
 </div>
 <div id="navigation">
 TODO: Menu will go here...
 </div>
 </form>
 </div>
 </body>
</html>
```

A master page defines both the static page layout and the regions that can be edited by the ASP.NET pages that use the master page. These content editable regions are indicated by the ContentPlaceHolder control, which can be seen within the content . Our master page has a single ContentPlaceHolder (MainContent), but master page's may have multiple ContentPlaceHolders.

With the markup entered above, switching to the Design view shows the master page's layout. Any ASP.NET pages that use this master page will have this uniform layout, with the ability to specify the markup for the MainContent region.

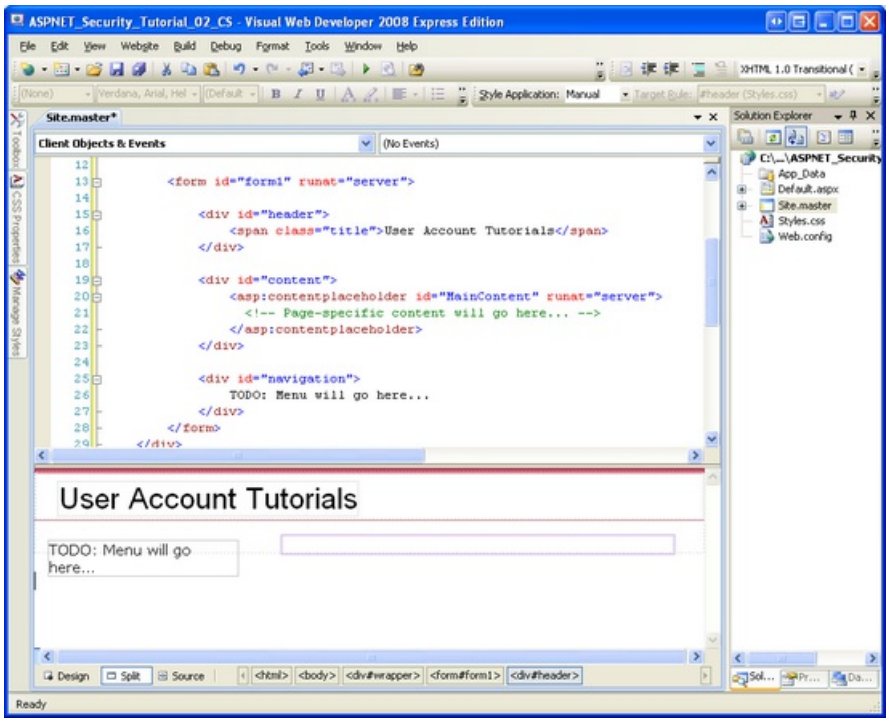


Figure 04:The Master Page, When Viewed Through the Design View(Click to view full-size image)

Creating Content Pages

At this point we have a Default.aspx page in our website, but it does not use the master page we just created. While it is possible to manipulate the declarative markup of a web page to use a master page, if the page doesn't contain any content yet it is easier to just delete the page and re-add it to the project, specifying the master page to use. Therefore, start by deleting Default.aspx from the project.

Next, right-click on the project name in the Solution Explorer and choose to add a new Web Form named Default.aspx. This time, check the Select master page checkbox and choose the Site.master master page from the list.

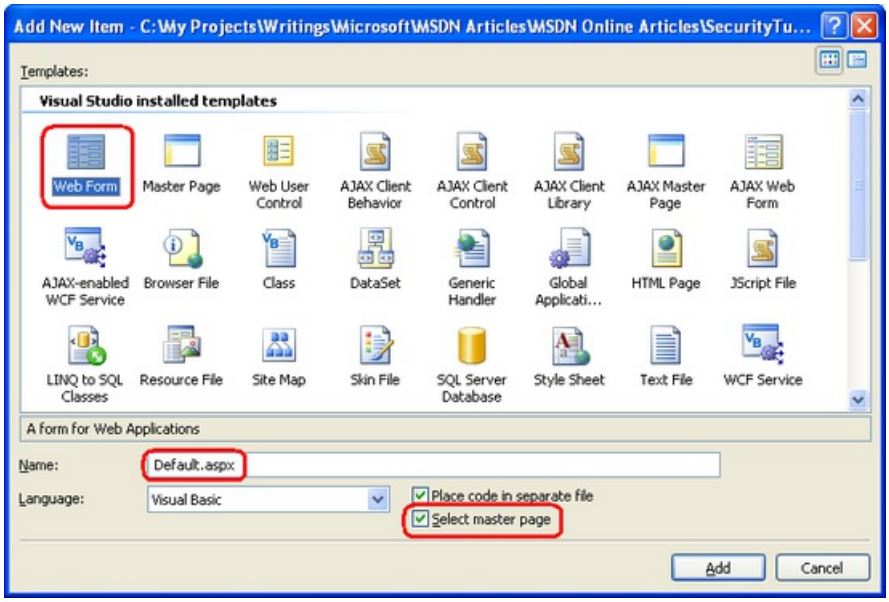


Figure 05: Add a New Default.aspx Page Choosing to Select a Master Page(Click to view full-size image)

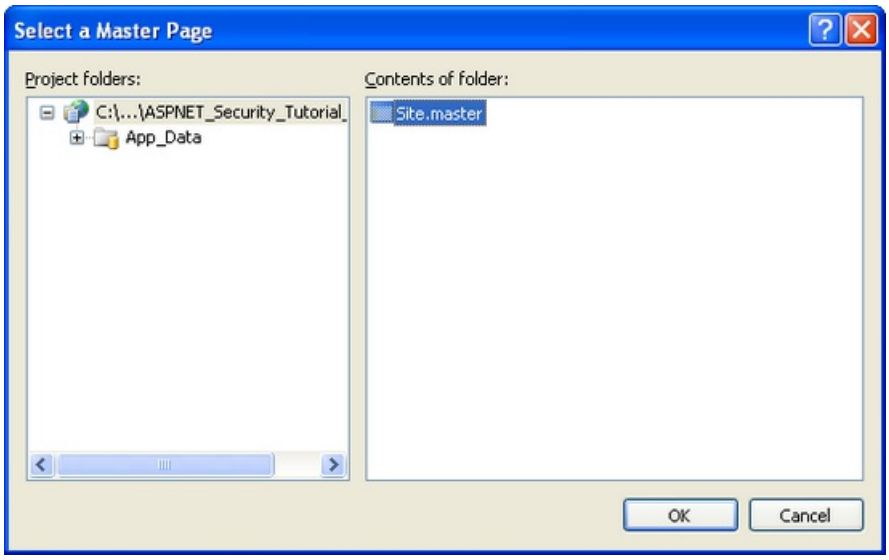


Figure 06: Use the Site.master Master Page(Click to view full-size image)

> **NOTE**
> If you are using the Web Application Project Model the Add New Item dialog box does not include a Select master page checkbox. Instead, you need to add an item of type Web Content Form. After choosing the Web Content Form option and clicking Add, Visual Studio will display the same Select a Master dialog box shown in Figure 6.

The new Default.aspx page's declarative markup includes just a @Page directive specifying the path to the master page file and a Content control for the master page's MainContent ContentPlaceHolder.

```
<%@ Page Language="VB" MasterPageFile="~/Site.master" AutoEventWireup="false" CodeFile="Default.aspx.vb"
Inherits="_Default" Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" Runat="Server">
</asp:Content>
```

For now, leave Default.aspx empty. We will return to it later in this tutorial to add content.

> **NOTE**
> Our master page includes a section for a menu or some other navigation interface. We will create such an interface in a future tutorial.

Step 2: Enabling Forms Authentication

With the ASP.NET website created, our next task is to enable forms authentication. The application's authentication configuration is specified through the <authentication> element in Web.config. The <authentication> element contains a single attribute named mode that specifies the authentication model used by the application. This attribute can have one of the following four values:

- **Windows** - as discussed in the preceding tutorial, when an application uses Windows authentication it is the web server's responsibility to authenticate the visitor, and this is usually done through Basic, Digest, or Integrated Windows authentication.
- **Forms**- users are authenticated via a form on a web page.
- **Passport**- users are authenticated using Microsoft's Passport Network.
- **None**- no authentication model is used; all visitors are anonymous.

By default, ASP.NET applications use Windows authentication. To change the authentication type to forms

authentication, then, we need to modify the <authentication> element's mode attribute to Forms.

If your project does not yet contain a Web.config file, add one now by right-clicking on the project name in the Solution Explorer, choosing Add New Item, and then adding a Web Configuration file.

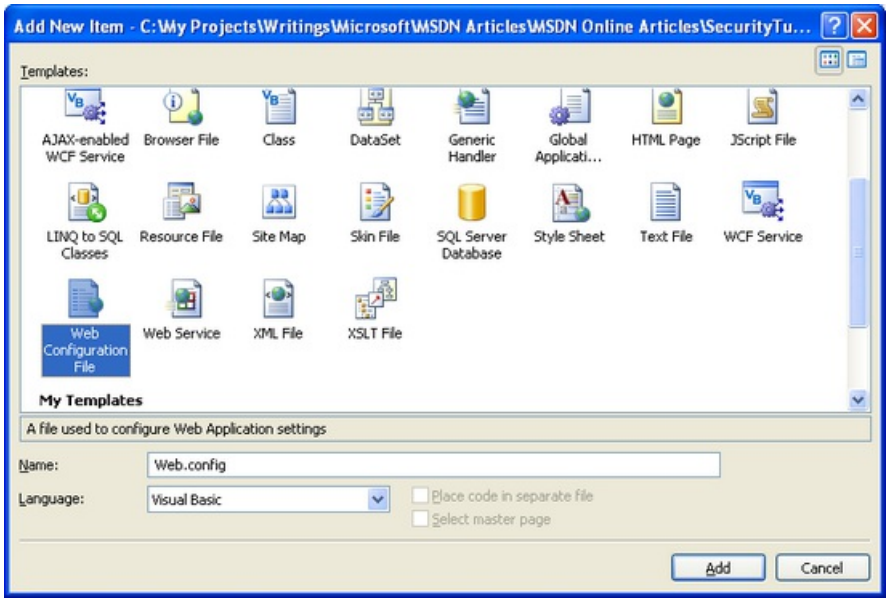


Figure 07: If Your Project Does Not Yet Include Web.config, Add It Now(Click to view full-size image)

Next, locate the <authentication> element and update it to use forms authentication. After this change, your Web.config file's markup should look similar to the following:

```
<configuration>
 <system.web>
 ... Unrelated configuration settings and comments removed for brevity ...
 <!--
 The <authentication> section enables configuration
 of the security authentication mode used by
 ASP.NET to identify an incoming user.
 -->
 <authentication mode="Forms" />
 </system.web>
</configuration>
```

> **NOTE**
> Since Web.config is an XML file, casing is important. Make sure that you set the mode attribute to Forms, with a capital F. If you use a different casing, such as forms, you'll receive a configuration error when visiting the site through a browser.

The <authentication> element may optionally include a <forms> child element that contains forms authentication-specific settings. For now, let's just use the default forms authentication settings. We will explore the <forms> child element in more detail in the next tutorial.

Step 3: Building the Login Page

In order to support forms authentication our website needs a login page. As discussed in the Understanding the Forms Authentication Workflow section, the FormsAuthenticationModule will automatically redirect the user to the login page if they attempt to access a page that they are not authorized to view. There are also ASP.NET Web controls that will display a link to the login page to anonymous users. This begs the question, What is the URL of the login page?

By default, the forms authentication system expects the login page to be named Login.aspx and placed in the

root directory of the web application. If you want to use a different login page URL, you can do so by specifying it in Web.config. We will see how to do this in the subsequent tutorial.

The login page has three responsibilities:

1. Provide an interface that allows the visitor to enter their credentials.
2. Determine if the submitted credentials are valid.
3. Log in the user by creating the forms authentication ticket.

Creating the Login Page's User Interface

Let's get started with the first task. Add a new ASP.NET page to the site's root directory named Login.aspx and associate it with the Site.master master page.

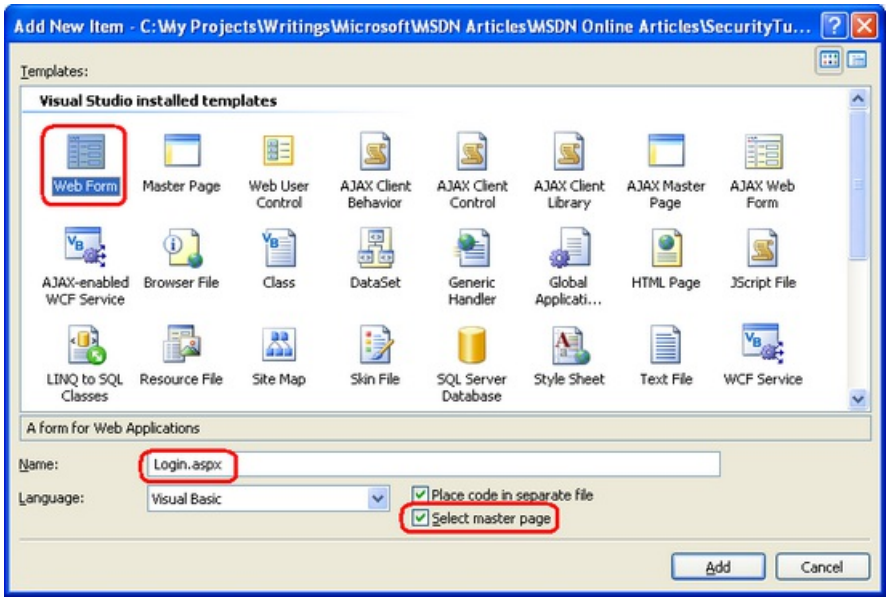


Figure 08: Add a New ASP.NET Page Named Login.aspx(Click to view full-size image)

The typical login page interface consists of two textboxes - one for the user's name, one for their password - and a button to submit the form. Websites oftentimes include a Remember me checkbox that, if checked, persists the resulting authentication ticket across browser restarts.

Add two TextBoxes to Login.aspx and set their ID properties to UserName and Password, respectively. Also set Password's TextMode property to Password. Next, add a CheckBox control, setting its ID property to RememberMe and its Text property to Remember Me. Following that, add a Button named LoginButton whose Text property is set to Login. And finally, add a Label Web control and set its ID property to InvalidCredentialsMessage, its Text property to Your username or password is invalid. Please try again., its ForeColor property to Red, and its Visible property to False.

At this point your screen should look similar to the screen shot in Figure 9, and your page's declarative syntax should like the following:

```
<%@ Page Language="VB" MasterPageFile="~/Site.master" AutoEventWireup="false" CodeFile="Login.aspx.vb"
Inherits="Login" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" Runat="Server">
 <h1>
 Login
 </h1>
 <p>
 Username: <asp:TextBox ID="UserName" runat="server"></asp:TextBox>
 </p>
 <p>
 Password: <asp:TextBox ID="Password" runat="server" TextMode="Password"></asp:TextBox>
 </p>
 <p>
 <asp:CheckBox ID="RememberMe" runat="server" Text="Remember Me" />
 </p>
 <p>
 <asp:Button ID="LoginButton" runat="server" Text="Login" OnClick="LoginButton_Click" />
 </p>
 <p>
 <asp:Label ID="InvalidCredentialsMessage" runat="server" ForeColor="Red" Text="Your username or password
is invalid. Please try again."
 Visible="False"></asp:Label>
 </p>
</asp:Content>
```

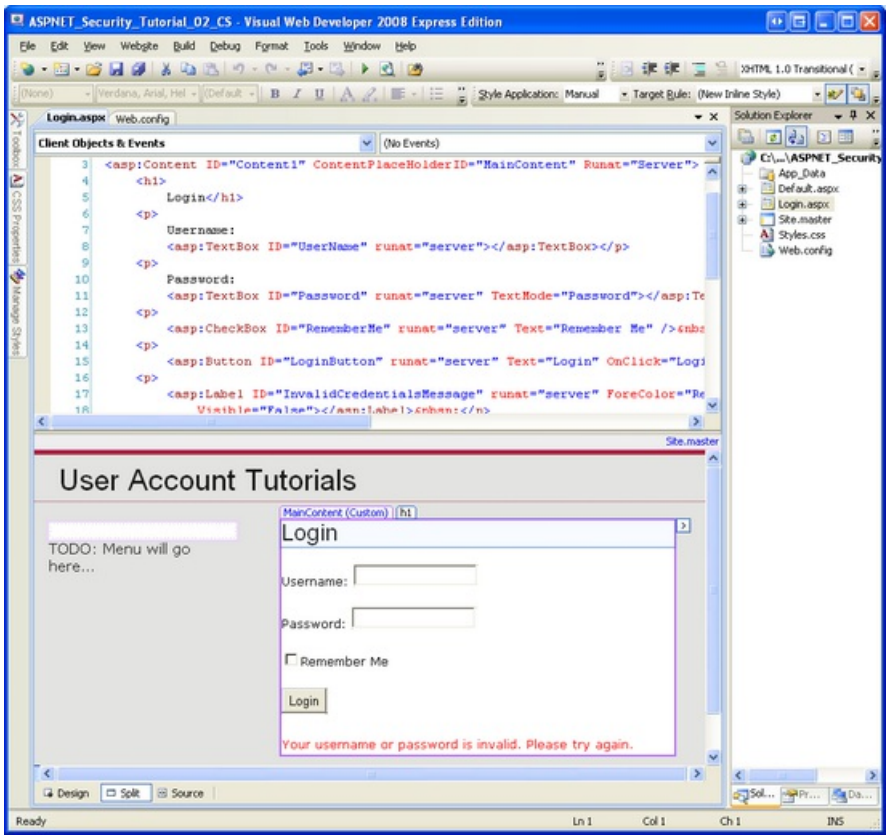


Figure 09: The Login Page Contains Two TextBoxes, a CheckBox, a Button, and a Label(Click to view full-size image)

Finally, create an event handler for the LoginButton's Click event. From the Designer, simply double-click the Button control to create this event handler.

Determining If the Supplied Credentials are Valid

We now need to implement task 2 in the Button's Click event handler - determining whether the supplied credentials are valid. In order to do this there needs to be a user store that holds all of the users' credentials so that we can determine if the supplied credentials match up with any known credentials.

Prior to ASP.NET 2.0, developers were responsible for implementing both their own user stores and writing the code to validate the supplied credentials against the store. Most developers would implement the user store in a database, creating a table named Users with columns like UserName, Password, Email, LastLoginDate, and so forth. This table, then, would have one record per user account. Verifying a user's supplied credentials would involve querying the database for a matching username and then ensuring that the password in the database corresponded to the supplied password.

With ASP.NET 2.0, developers should use one of the Membership providers to manage the user store. In this tutorial series we will be using the SqlMembershipProvider, which uses a SQL Server database for the user store. When using the SqlMembershipProvider we need to implement a specific database schema that includes the tables, views, and stored procedures expected by the provider. We will examine how to implement this schema in the *Creating the Membership Schema in SQL Server* tutorial. With the Membership provider in place, validating the user's credentials is as simple as calling the Membership class's ValidateUser(*username*, *password*) method, which returns a Boolean value indicating whether the validity of the *username* and *password* combination. Seeing as we have not yet implemented the SqlMembershipProvider's user store, we cannot use the Membership class's ValidateUser method at this time.

Rather than take the time to build our own custom Users database table (which would be obsolete once we implemented the SqlMembershipProvider), let's instead hard-code the valid credentials within the login page itself. In the LoginButton's Click event handler, add the following code:

```
Protected Sub LoginButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
LoginButton.Click
 ' Three valid username/password pairs: Scott/password, Jisun/password, and Sam/password.
 Dim users() As String = {"Scott", "Jisun", "Sam"}
 Dim passwords() As String = {"password", "password", "password"}
 For i As Integer = 0 To users.Length - 1
 Dim validUsername As Boolean = (String.Compare(UserName.Text, users(i), True) = 0)
 Dim validPassword As Boolean = (String.Compare(Password.Text, passwords(i), False) = 0)
 If validUsername AndAlso validPassword Then
 ' TODO: Log in the user...
 ' TODO: Redirect them to the appropriate page
 End If
 Next
 ' If we reach here, the user's credentials were invalid
 InvalidCredentialsMessage.Visible = True
End Sub
```

As you can see, there are three valid user accounts - Scott, Jisun, and Sam - and all three have the same password (password). The code loops through the users and passwords arrays looking for a valid username and password match. If both the username and password are valid, we need to login the user and then redirect them to the appropriate page. If the credentials are invalid, then we display the InvalidCredentialsMessage Label.

When a user enters valid credentials, I mentioned that they are then redirected to the appropriate page. What is the appropriate page, though? Recall that when a user visits a page they are not authorized to view, the FormsAuthenticationModule automatically redirects them to the login page. In doing so, it includes the requested URL in the querystring via the ReturnUrl parameter. That is, if a user attempted to visit ProtectedPage.aspx, and they were not authorized to do so, the FormsAuthenticationModule would redirect them to:

Login.aspx?ReturnUrl=ProtectedPage.aspx

Upon successfully logging in, the user should be redirected back to ProtectedPage.aspx. Alternatively, users may visit the login page on their own volition. In that case, after logging in the user they should be sent to the root folder's Default.aspx page.

Logging In the User

Assuming that the supplied credentials are valid, we need to create a forms authentication ticket, thereby

logging in the user to the site. The FormsAuthentication class in the System.Web.Security namespace provides assorted methods for logging in and logging out users via the forms authentication system. While there are several methods in the FormsAuthentication class, the three we are interested in at this juncture are:

- GetAuthCookie(*username*, *persistCookie*) - creates a forms authentication ticket for the supplied name *username*. Next, this method creates and returns an HttpCookie object that holds the contents of the authentication ticket. If *persistCookie* is True, a persistent cookie is created.
- SetAuthCookie(*username*, *persistCookie*) - calls the GetAuthCookie(*username*, *persistCookie*) method to generate the forms authentication cookie. This method then adds the cookie returned by GetAuthCookie to the Cookies collection (assuming cookies-based forms authentication is being used; otherwise, this method calls an internal class that handles the cookieless ticket logic).
- RedirectFromLoginPage(*username*, *persistCookie*) - this method calls SetAuthCookie(*username*, *persistCookie*), and then redirects the user to the appropriate page.

GetAuthCookie is handy when you need to modify the authentication ticket before writing the cookie out to the Cookies collection. SetAuthCookie is useful if you want to create the forms authentication ticket and add it to the Cookies collection, but do not want to redirect the user to the appropriate page. Perhaps you want to keep them on the login page or send them to some alternate page.

Since we want to log in the user and redirect them to the appropriate page, let's use RedirectFromLoginPage. Update the LoginButton's Click event handler, replacing the two commented TODO lines with the following line of code:

```
FormsAuthentication.RedirectFromLoginPage(UserName.Text, RememberMe.Checked)
```

When creating the forms authentication ticket we use the UserName TextBox's Text property for the forms authentication ticket *username* parameter, and the checked state of the RememberMe CheckBox for the *persistCookie* parameter.

To test the login page, visit it in a browser. Start by entering invalid credentials, such as a username of Nope and a password of wrong. Upon clicking the Login button a postback will occur and the InvalidCredentialsMessage Label will be displayed.

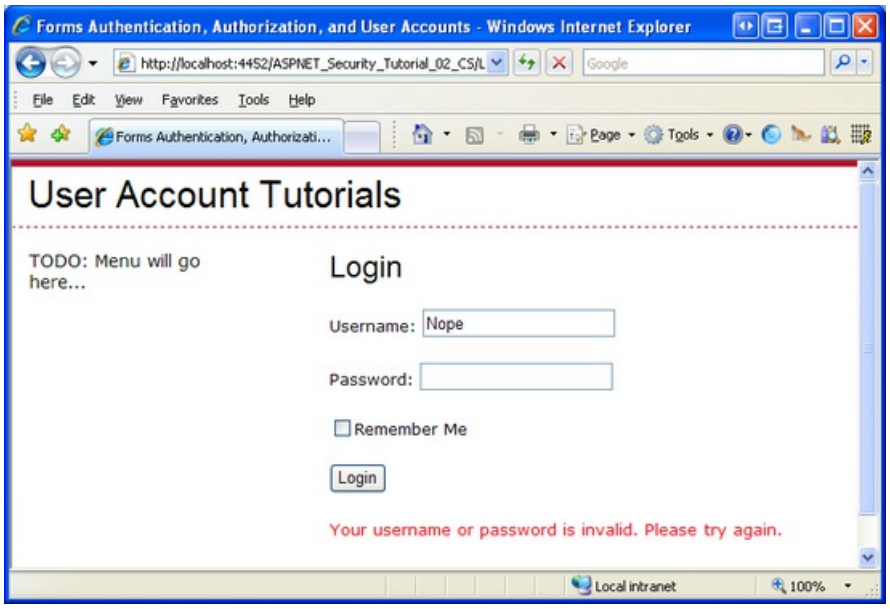


Figure 10: The InvalidCredentialsMessage Label is Displayed When Entering Invalid Credentials(Click to view full-size image)

Next, enter valid credentials and click the Login button. This time when the postback occurs a forms authentication ticket is created and you are automatically redirected back to Default.aspx. At this point you have logged in to the website, although there are no visual cues to indicate that you are currently logged in. In Step 4

we will see how to programmatically determine whether a user is logged in or not as well as how to identify the user visiting the page.

Step 5 examines techniques for logging a user out of the website.

Securing the Login Page

When the user enters her credentials and submits the login page form, the credentials - including her password - are transmitted over the Internet to the web server in *plain text*. That means any hacker sniffing the network traffic can see the username and password. To prevent this, it is essential to encrypt the network traffic by using Secure Socket Layers (SSL). This will ensure that the credentials (as well as the entire page's HTML markup) are encrypted from the moment they leave the browser until they are received by the web server.

Unless your website contains sensitive information, you will only need to use SSL on the login page and on other pages where the user's password would otherwise be sent over the wire in plain text. You do not need to worry about securing the forms authentication ticket since, by default, it is both encrypted and digitally signed (to prevent tampering). A more thorough discussion on forms authentication ticket security is presented in the following tutorial.

> **NOTE**
>
> Many financial and medical websites are configured to use SSL on *all* pages accessible to authenticated users. If you are building such a website you can configure the forms authentication system so that the forms authentication ticket is only transmitted over a secure connection. We will look at the various forms authentication configuration options in the next tutorial, *Forms Authentication Configuration and Advanced Topics*.

Step 4: Detecting Authenticated Visitors and Determining Their Identity

At this point we have enabled forms authentication and created a rudimentary login page, but we have yet to examine how we can determine whether a user is authenticated or anonymous. In certain scenarios we may wish to display different data or information depending on whether an authenticated or anonymous user is visiting the page. Moreover, we oftentimes need to know the identity of the authenticated user.

Let's augment the existing Default.aspx page to illustrate these techniques. In Default.aspx add two Panel controls, one named AuthenticatedMessagePanel and another named AnonymousMessagePanel. Add a Label control named WelcomeBackMessage in the first Panel. In the second Panel add a HyperLink control, set its Text property to Log In and its NavigateUrl property to ~/Login.aspx. At this point the declarative markup for Default.aspx should look similar to the following:

```
<%@ Page Language="VB" MasterPageFile="~/Site.master" AutoEventWireup="false" CodeFile="Default.aspx.vb"
Inherits="_Default" Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" Runat="Server">
 <asp:Panel runat="server" ID="AuthenticatedMessagePanel">
 <asp:Label runat="server" ID="WelcomeBackMessage"></asp:Label>
 </asp:Panel>
 <asp:Panel runat="Server" ID="AnonymousMessagePanel">
 <asp:HyperLink runat="server" ID="lnkLogin" Text="Log In" NavigateUrl="~/Login.aspx"></asp:HyperLink>
 </asp:Panel>
</asp:Content>
```

As you have probably guessed by now, the idea here is to display just the AuthenticatedMessagePanel to authenticated visitors and just the AnonymousMessagePanel to anonymous visitors. To accomplish this we need to set these Panels' Visible properties depending on whether the user is logged in or not.

The Request.IsAuthenticated property returns a Boolean value indicating whether the request has been

authenticated. Enter the following code into the Page_Load event handler code:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
 If Request.IsAuthenticated Then
 WelcomeBackMessage.Text = "Welcome back!"
 AuthenticatedMessagePanel.Visible = True
 AnonymousMessagePanel.Visible = False
 Else
 AuthenticatedMessagePanel.Visible = False
 AnonymousMessagePanel.Visible = True
 End If
End Sub
```

With this code in place, visit Default.aspx through a browser. Assuming that you have yet to log in, you will see a link to the login page (see Figure 11). Click this link and log in to the site. As we saw in Step 3, after entering your credentials you will be returned to Default.aspx, but this time the page shows the Welcome back! message (see Figure 12).

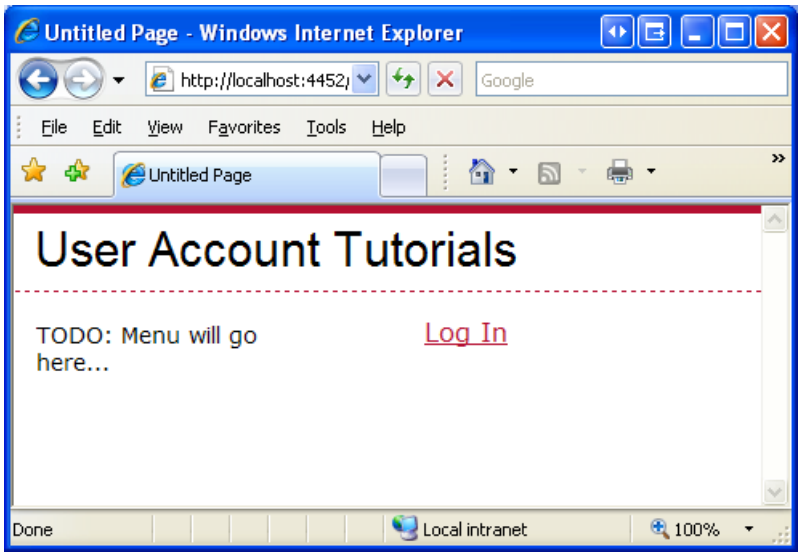


Figure 11: When Visiting Anonymously, a Log In Link is Displayed(Click to view full-size image)

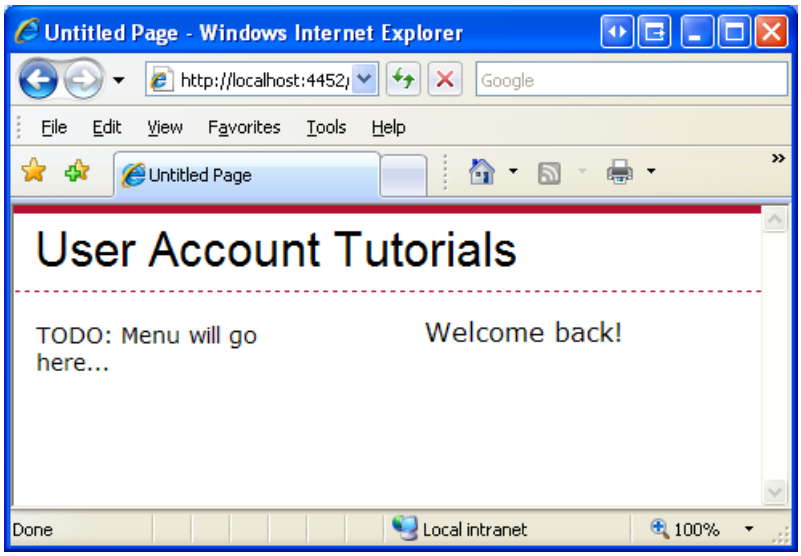


Figure 12: Authenticated Users are Shown the Welcome back! Message(Click to view full-size image)

We can determine the currently logged on user's identity via the HttpContext object's User property. The HttpContext object represents information about the current request, and is the home for such common ASP.NET objects as Response, Request, and Session, among others. The User property represents the security context of the current HTTP request and implements the IPrincipal interface.

The User property is set by the FormsAuthenticationModule. Specifically, when the FormsAuthenticationModule finds a forms authentication ticket in the incoming request, it creates a new GenericPrincipal object and assigns it to the User property.

Principal objects (like GenericPrincipal) provide information on the user's identity and the roles to which they belong. The IPrincipal interface defines two members:

- IsInRole(*roleName*) - a method that returns a Boolean value indicating if the principal belongs to the specified role.
- Identity - a property that returns an object that implements the IIdentity interface. The IIdentity interface defines three properties: AuthenticationType, IsAuthenticated, and Name.

We can determine the name of the current visitor using the following code:

```
Dim currentUsersName As String = User.Identity.Name
```

When using forms authentication, a FormsIdentity object is created for the GenericPrincipal's Identity property. The FormsIdentity class always returns the string Forms for its AuthenticationType property and True for its IsAuthenticated property. The Name property returns the username specified when creating the forms authentication ticket. In addition to these three properties, FormsIdentity includes access to the underlying authentication ticket via its Ticket property. The Ticket property returns an object of type FormsAuthenticationTicket, which has properties like Expiration, IsPersistent, IssueDate, Name, and so on.

The important point to take away here is that the *username* parameter specified in the FormsAuthentication.GetAuthCookie(*username*, *persistCookie*), FormsAuthentication.SetAuthCookie(*username*, *persistCookie*), and FormsAuthentication.RedirectFromLoginPage(*username*, *persistCookie*) methods is the same value returned by User.Identity.Name. Moreover, the authentication ticket created by these methods is available by casting User.Identity to a FormsIdentity object and then accessing the Ticket property:

```
Dim ident As FormsIdentity = CType(User.Identity, FormsIdentity)

Dim authTicket As FormsAuthenticationTicket = ident.Ticket
```

Let's provide a more personalized message in Default.aspx. Update the Page_Load event handler so that the WelcomeBackMessage Label's Text property is assigned the string Welcome back, *username*!

```
WelcomeBackMessage.Text = "Welcome back, " & User.Identity.Name & "!"
```

Figure 13 shows the effect of this modification (when logging in as user Scott).

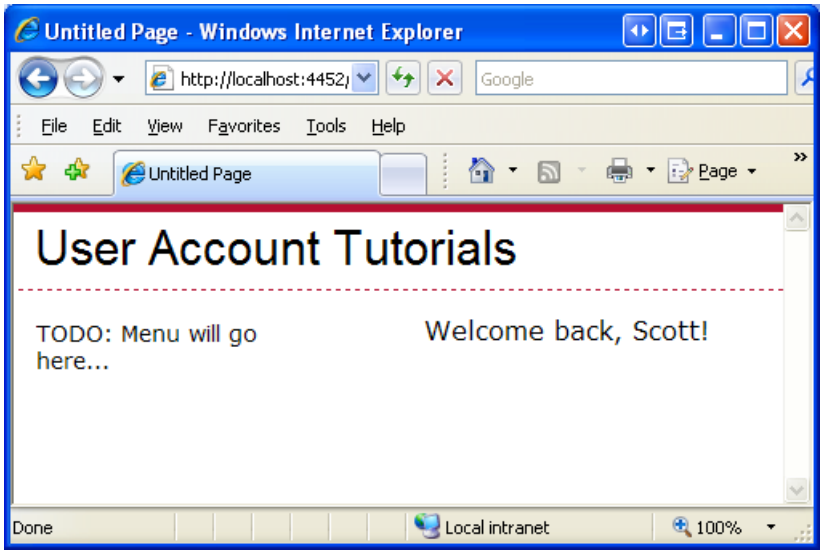


Figure 13: The Welcome Message Includes the Currently Logged In User's Name(Click to view full-size image)

Using the LoginView and LoginName Controls

Displaying different content to authenticated and anonymous users is a common requirement; so is displaying the name of the currently logged on user. For that reason, ASP.NET includes two Web controls that provide the same functionality shown in Figure 13, but without the need to write a single line of code.

The LoginView control is a template-based Web control that makes it easy to display different data to authenticated and anonymous users. The LoginView includes two predefined templates:

- AnonymousTemplate - any markup added to this template is only displayed to anonymous visitors.
- LoggedInTemplate - this template's markup is shown only to authenticated users.

Let's add the LoginView control to our site's master page, Site.master. Rather than adding just the LoginView control, though, let's add both a new ContentPlaceHolder control and then put the LoginView control within that new ContentPlaceHolder. The rationale for this decision will become apparent shortly.

> **NOTE**
>
> In addition to the AnonymousTemplate and LoggedInTemplate, the LoginView control can include role-specific templates. Role-specific templates show markup only to those users that belong to a specified role. We will examine the role-based features of the LoginView control in a future tutorial.

Start by adding a ContentPlaceHolder named LoginContent into the master page within the navigation element. You can simply drag a ContentPlaceHolder control from the Toolbox onto the Source view, placing the resulting markup right above the TODO: Menu will go here text.

```
<div id="navigation">
 <asp:ContentPlaceHolder ID="LoginContent" runat="server">
 </asp:ContentPlaceHolder>
 TODO: Menu will go here...
</div>
```

Next, add a LoginView control within the LoginContent ContentPlaceHolder. Content placed into the master page's ContentPlaceHolder controls are considered *default content* for the ContentPlaceHolder. That is, ASP.NET pages that use this master page can specify their own content for each ContentPlaceHolder or use the master page's default content.

The LoginView and other login-related controls are located in the Toolbox's Login tab.

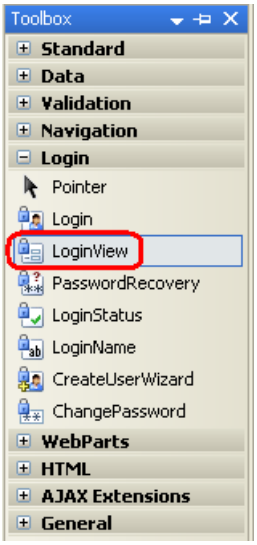


Figure 14: The LoginView Control in the Toolbox(Click to view full-size image)

Next, add two <br /> elements immediately after the LoginView control, but still within the ContentPlaceHolder. At this point, the navigation element's markup should look like the following:

```
<div id="navigation">
 <asp:ContentPlaceHolder ID="LoginContent" runat="server">
 <asp:LoginView ID="LoginView1" runat="server">
 </asp:LoginView>
 <br /><br />
 </asp:ContentPlaceHolder>
 TODO: Menu will go here...
</div>
```

The LoginView's templates can be defined from the Designer or the declarative markup. From Visual Studio's Designer, expand the LoginView's smart tag, which lists the configured templates in a drop-down list. Type in the text Hello, stranger into the AnonymousTemplate; next, add a HyperLink control and set its Text and NavigateUrl properties to Log In and ~/Login.aspx, respectively.

After configuring the AnonymousTemplate, switch to the LoggedInTemplate and enter the text, "Welcome back, ". Then drag a LoginName control from the Toolbox into the LoggedInTemplate, placing it immediately after the "Welcome back, " text. The LoginName control, as its name implies, displays the name of the currently logged in user. Internally, the LoginName control simply outputs the User.Identity.Name property

After making these additions to the LoginView's templates, the markup should look similar to the following:

```
<div id="navigation">
 <asp:ContentPlaceHolder ID="LoginContent" runat="server">
 <asp:LoginView ID="LoginView1" runat="server">
 <LoggedInTemplate>
 Welcome back, <asp:LoginName ID="LoginName1" runat="server" />.
 </LoggedInTemplate>
 <AnonymousTemplate>
 Hello, stranger. <asp:HyperLink ID="lnkLogin" runat="server" NavigateUrl="~/Login.aspx">Log
In</asp:HyperLink>
 </AnonymousTemplate>
 </asp:LoginView>
 <br /><br />
 </asp:ContentPlaceHolder>
 TODO: Menu will go here...
</div>
```

With this addition to the Site.master master page, each page in our website will display a different message depending on whether the user is authenticated. Figure 15 shows the Default.aspx page when visited through a browser by user Jisun. The Welcome back, Jisun message is repeated twice: once in the master page's navigation section on the left (via the LoginView control we just added) and once in the Default.aspx's content area (via Panel controls and programmatic logic).

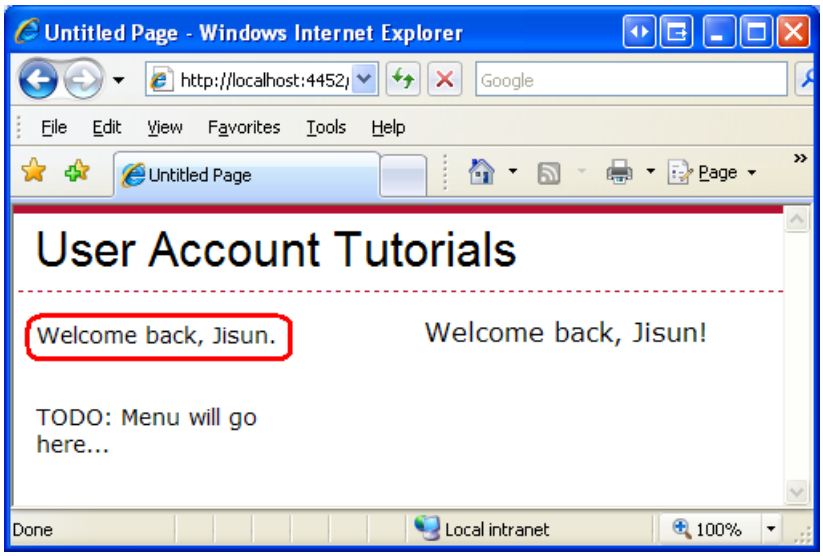


Figure 15: The LoginView Control Displays Welcome back, Jisun. (Click to view full-size image)

Because we added the LoginView to the master page, it can appear in every page on our site. However, there may be web pages where we don't want to show this message. One such page is the login page, since a link to the login page seems out of place there. Since we placed the LoginView control in a ContentPlaceHolder in the master page, we can override this default markup in our content page. Open Login.aspx and go to the Designer. Since we have not explicitly defined a Content control in Login.aspx for the LoginContent ContentPlaceHolder in the master page, the login page will show the master page's default markup for this ContentPlaceHolder. You can see this through the Designer - the LoginContent ContentPlaceHolder shows the default markup (the LoginView control).

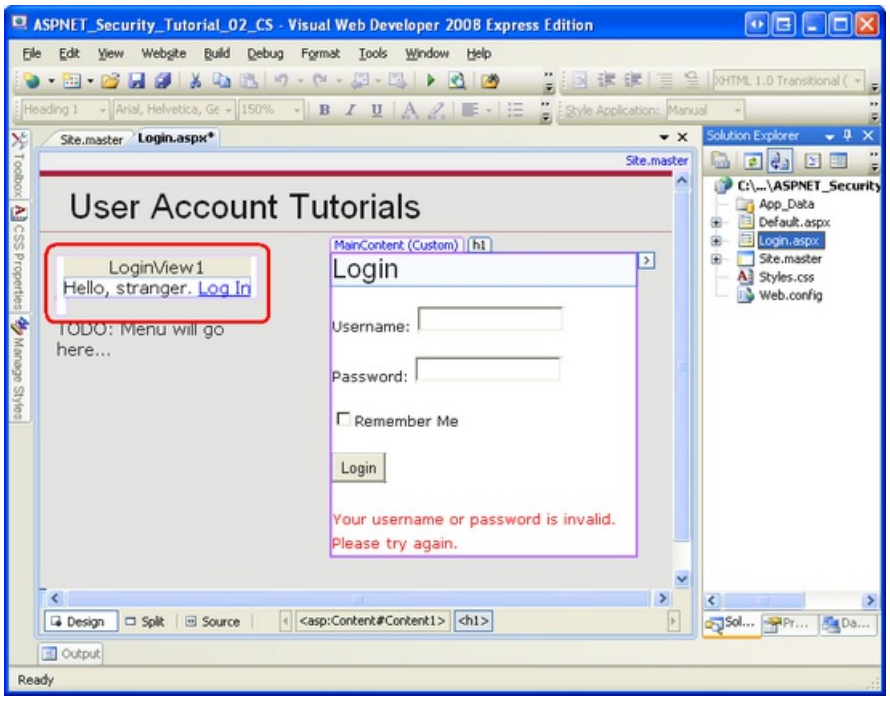


Figure 16: The Login Page Shows the Default Content for the Master Page's LoginContent ContentPlaceHolder(Click to view full-size image)

To override the default markup for the LoginContent ContentPlaceHolder, simply right-click on the region in the Designer and choose the Create Custom Content option from the context menu. (When using Visual Studio 2008 the ContentPlaceHolder includes a smart tag that, when selected, offers the same option.) This adds a new Content control to the page's markup and thereby allows us to define custom content for this page. You could add a custom message here, such as Please log in, but let's just leave this blank.

> **NOTE**
>
> In Visual Studio 2005, creating custom content creates an empty Content control in the ASP.NET page. In Visual Studio 2008, however, creating custom content copies the master page's default content into the newly created Content control. If you are using Visual Studio 2008, then, after creating the new Content control make sure to clear out the content copied over from the master page.

Figure 17 shows the Login.aspx page when visited from a browser after making this change. Note that there is no Hello, stranger or Welcome back, *username* message in the left navigation as there is when visiting Default.aspx.

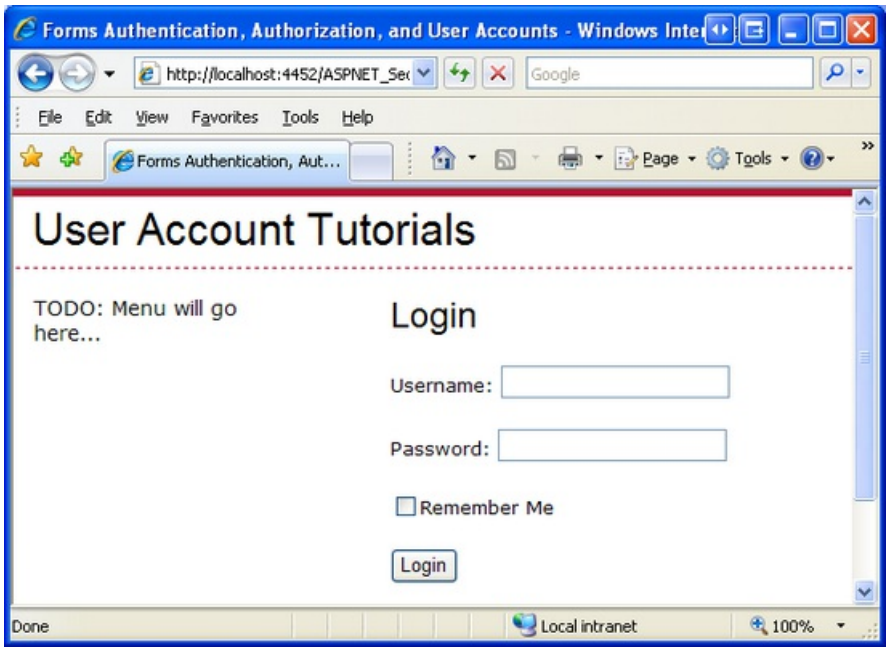


Figure 17: The Login Page Hides the Default LoginContent ContentPlaceHolder's Markup(Click to view full-size image)

Step 5: Logging Out

In Step 3 we looked at building a login page to log a user in to the site, but we have yet to see how to log a user out. In addition to methods for logging a user in, the FormsAuthentication class also provides a SignOut method. The SignOut method simply destroys the forms authentication ticket, thereby logging the user out of the site.

Offering a log out link is such a common feature that ASP.NET includes a control specifically designed to log a user out. The LoginStatus control displays either a Login LinkButton or a Logout LinkButton, depending on the user's authentication status. A Login LinkButton is rendered for anonymous users, whereas a Logout LinkButton is displayed to authenticated users. The text for the Login and Logout LinkButtons can be configured via the LoginStatus's LoginText and LogoutText properties.

Clicking the Login LinkButton causes a postback, from which a redirect is issued to the login page. Clicking the Logout LinkButton causes the LoginStatus control to invoke the FormsAuthentication.SignOff method and then redirects the user to a page. The page the logged off user is redirected to depends on the LogoutAction property, which can be assigned to one of the three following values:

- Refresh - the default; redirects the user to the page they were just visiting. If the page they were just visiting does not allow anonymous users, then the FormsAuthenticationModule will automatically redirect the user to the login page.

You may be curious as to why a redirect is performed here. If the user wants to remain on the same page, why the need for the explicit redirect? The reason is because when the Logoff LinkButton is clicked, the user still has the forms authentication ticket in their cookies collection. Consequently, the postback request is an authenticated request. The LoginStatus control calls the SignOut method, but that happens after the FormsAuthenticationModule has authenticated the user. Therefore, an explicit redirect causes the browser to re-request the page. By the time the browser re-requests the page, the forms authentication ticket has been removed and therefore the incoming request is anonymous.

- Redirect - the user is redirected to the URL specified by the LoginStatus's LogoutPageUrl property.
- RedirectToLoginPage - the user is redirected to the login page.

Let's add a LoginStatus control to the master page and configure it to use the Redirect option to send the user to a page that displays a message confirming that they have been signed out. Start by creating a page in the root

directory named Logout.aspx. Don't forget to associate this page with the Site.master master page. Next, enter a message in the page's markup explaining to the user that they have been logged out.

Next, return to the Site.master master page and add a LoginStatus control beneath the LoginView in the LoginContent ContentPlaceHolder. Set the LoginStatus control's LogoutAction property to Redirect and its LogoutPageUrl property to ~/Logout.aspx .

```
<div id="navigation">
 <asp:ContentPlaceHolder ID="LoginContent" runat="server">
 <asp:LoginView ID="LoginView1" runat="server">
 <LoggedInTemplate>
 Welcome back, <asp:LoginName ID="LoginName1" runat="server" />.
 </LoggedInTemplate>
 <AnonymousTemplate>
 Hello, stranger. <asp:HyperLink ID="lnkLogin" runat="server" NavigateUrl="~/Login.aspx">Log
In</asp:HyperLink>
 </AnonymousTemplate>
 </asp:LoginView>
 <br />
 <asp:LoginStatus ID="LoginStatus1" runat="server" LogoutAction="Redirect" LogoutPageUrl="~/Logout.aspx" />
 <br /><br />
 </asp:ContentPlaceHolder>
 TODO: Menu will go here...
</div>
```

Since the LoginStatus is outside of the LoginView control, it will appear for both anonymous and authenticated users, but that's OK because the LoginStatus will correctly display a Login or Logout LinkButton. With the addition of the LoginStatus control, the Log In HyperLink in the AnonymousTemplate is superfluous, so remove it.

Figure 18 shows Default.aspx when Jisun visits. Note that the left column displays the message, Welcome back, Jisun along with a link to log out. Clicking the log out LinkButton causes a postback, signs Jisun out of the system, and then redirects her to Logout.aspx. As Figure 19 shows, by the time Jisun reaches Logout.aspx she has already been signed out and is therefore anonymous. Consequently, the left column shows the text Welcome, stranger and a link to the login page.

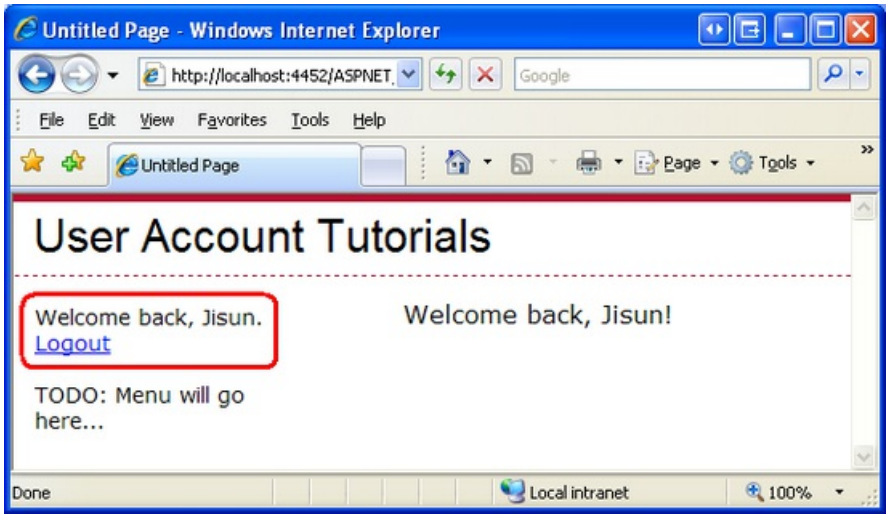


Figure 18: Default.aspx Shows Welcome Back, Jisun Along with a Logout LinkButton(Click to view full-size image)

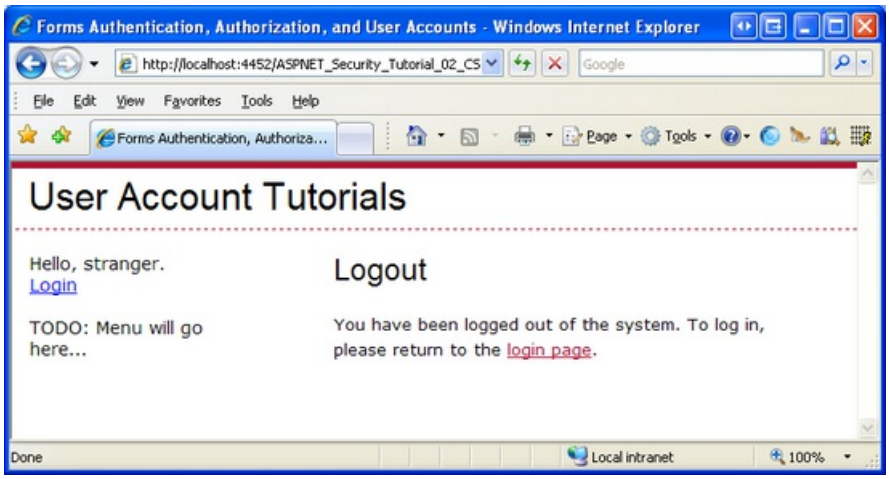


Figure 19: Logout.aspx Shows Welcome, stranger Along with a Login LinkButton(Click to view full-size image)

> **NOTE**
>
> I encourage you to customize the Logout.aspx page to hide the master page's LoginContent ContentPlaceHolder (like we did for Login.aspx in Step 4). The reason is because the Login LinkButton rendered by the LoginStatus control (the one beneath Hello, stranger) sends the user to the login page passing the current URL in the ReturnUrl querystring parameter. In short, if a user who has logged out clicks this LoginStatus's Login LinkButton, and then logs in, they will be redirected back to Logout.aspx, which could easily confuse the user.

Summary

In this tutorial we started with an examination of the forms authentication workflow and then turned to implementing forms authentication in an ASP.NET application. Forms authentication is powered by the FormsAuthenticationModule, which has two responsibilities: identifying users based on their forms authentication ticket, and redirecting unauthorized users to the login page.

The .NET Framework's FormsAuthentication class includes methods for creating, inspecting, and removing forms authentication tickets. The Request.IsAuthenticated property and User object provide additional programmatic support for determining whether a request is authenticated and information about the user's identity. There are also the LoginView, LoginStatus, and LoginName Web controls, which give developers a quick, code-free way for performing many common login-related tasks. We will examine these and other login-related Web controls in greater detail in future tutorials.

This tutorial provided a cursory overview of forms authentication. We did not examine the assorted configuration options, look at how cookieless forms authentication tickets work, or explore how ASP.NET protects the contents of the forms authentication ticket. We will discuss these topics and more in the next tutorial.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Changes Between IIS6 and IIS7 Security
- Login ASP.NET Controls
- Professional ASP.NET 2.0 Security, Membership, and Role Management (ISBN: 978-0-7645-9698-8)
- The <authentication> Element
- The <forms> Element for <authentication>

Video Training on Topics Contained in this Tutorial

- Using Basic Forms Authentication in ASP.NET

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial include Alicja Maziarz, John Suru, and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4guysfromrolla.com.

PREVIOUS NEXT

Forms Authentication Configuration and Advanced Topics (VB)

2/20/2020 • 39 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

In this tutorial we will examine the various forms authentication settings and see how to modify them through the forms element. This will entail a detailed look at customizing the forms authentication ticket's timeout value, using a login page with a custom URL (like SignIn.aspx instead of Login.aspx), and cookieless forms authentication tickets.

Introduction

In the previous tutorial we looked at the steps necessary for implementing forms authentication in an ASP.NET application, from specifying configuration settings in Web.config to creating a log in page to displaying different content for authenticated and anonymous users. Recall that we configured the website to use forms authentication by setting the mode attribute of the <authentication> element to Forms. The <authentication> element may optionally include a <forms> child element, through which an assortment of forms authentication settings may be specified.

In this tutorial we will examine the various forms authentication settings and see how to modify them through the <forms> element. This will entail a detailed look at customizing the forms authentication ticket's timeout value, using a login page with a custom URL (like SignIn.aspx instead of Login.aspx), and cookieless forms authentication tickets. We will also examine the makeup of the forms authentication ticket more closely and see the precautions ASP.NET takes to ensure that the ticket's data is secure from inspection and tampering. Finally, we will look at how to store extra user data in the forms authentication ticket and how to model this data through a custom principal object.

Step 1: Examining the <forms> Configuration Settings

The forms authentication system in ASP.NET offers a number of configuration settings that can be customized on an application-by-application basis. This includes settings like: the lifetime of the forms authentication ticket; what sort of protection is applied to the ticket; under what conditions cookieless authentication tickets are used; the path to the login page; and other information. To modify the default values, add a <forms> element as a child of the <authentication> element, specifying those property values you want to customize as XML attributes like so:

```
<authentication mode="Forms">

 <forms

 propertyName1="value1"

 propertyName2="value2"

 ...

 propertyNameN="valueN"

 />

</authentication>
```

Table 1 summarizes the properties that can be customized through the <forms> element. Since Web.config is an XML file, the attribute names in the left column are case-sensitive.

ATTRIBUTE	DESCRIPTION
cookieless	This attribute specifies under what conditions the authentication ticket is stored in a cookie versus being embedded in the URL. Allowable values are: UseCookies; UseUri; AutoDetect; and UseDeviceProfile (the default). Step 2 examines this setting in more detail.
defaultUrl	Indicates the URL that users are redirected to after signing in from the login page if there is no RedirectUrl value specified in the querystring. The default value is default.aspx.
domain	When using cookie-based authentication tickets, this setting specifies the cookie s domain value. The default value is an empty string, which causes the browser to use the domain from which it was issued (such as www.yourdomain.com). In this case, the cookie will **not** be sent when making requests to subdomains, such as admin.yourdomain.com. If you want the cookie to be passed to all subdomains you need to customize the domain attribute setting it to yourdomain.com.
enableCrossAppRedirects	A Boolean value indicating whether authenticated users are remembered when redirected to URLs in other web applications on the same server. The default is false.
loginUrl	The URL of the login page. The default value is login.aspx.
name	When using cookie-based authentication tickets, the name of the cookie. The default is .ASPXAUTH.
path	When using cookie-based authentication tickets, this setting specifies the cookie s path attribute. The path attribute enables a developer to limit the scope of a cookie to a particular directory hierarchy. The default value is /, which informs the browser to send the authentication ticket cookie to any request made to the domain.

ATTRIBUTE	DESCRIPTION
protection	Indicates what techniques are used to protect the forms authentication ticket. The allowable values are: All (the default); Encryption; None; and Validation. These settings are discussed in detail in Step 3.
requireSSL	A Boolean value that indicates whether an SSL connection is required to transmit the authentication cookie. The default value is false.
slidingExpiration	A Boolean value that indicates whether the authentication cookie s timeout is reset each time the user visits the site during a single session. The default value is true. The authentication ticket timeout policy is discussed in more detail in the Specifying the Ticket s Timeout Value section.
timeout	Specifies the time, in minutes, after which the authentication ticket cookie expires. The default value is 30. The authentication ticket timeout policy is discussed in more detail in the Specifying the Ticket s Timeout Value section.

Table 1: A Summary of the <forms> Element's Attributes

In ASP.NET 2.0 and beyond, the default forms authentication values are hard-coded in the FormsAuthenticationConfiguration class in the .NET Framework. Any modifications must be applied on an application-by-application basis in the Web.config file. This differs from ASP.NET 1.x, where the default forms authentication values were stored in the machine.config file (and could therefore be modified via editing machine.config). While on the topic of ASP.NET 1.x, it is worthwhile to mention that a number of the forms authentication system settings have different default values in ASP.NET 2.0 and beyond than in ASP.NET 1.x. If you are migrating your application from an ASP.NET 1.x environment, it is important to be aware of these differences. Consult the <forms> element technical documentation for a list of the differences.

> **NOTE**
> Several forms authentication settings, such as the timeout, domain, and path, specify details for the resulting forms authentication ticket cookie. For more information on cookies, how they work, and their various properties, read this Cookies tutorial.

Specifying the Ticket's Timeout Value

The forms authentication ticket is a token that represents an identity. With cookie-based authentication tickets, this token is held in the form of a cookie and sent to the web server on each request. Possession of the token, in essence, declares, I am *username*, I have already logged in, and is used so that a user's identity can be remembered across page visits.

The forms authentication ticket not only includes the user's identity, but also contains information to help ensure the integrity and security of the token. After all, we don't want a nefarious user to be able to create a counterfeit token, or to modify a legit token in some underhanded way.

One such bit of information included in the ticket is an *expiry*, which is the date and time the ticket is no longer valid. Each time the FormsAuthenticationModule inspects an authentication ticket, it ensures that the ticket's expiry has not yet passed. If it has, it disregards the ticket and identifies the user as being anonymous. This safeguard helps protect against replay attacks. Without an expiry, if a hacker was able to get her hands on a user's valid authentication ticket - perhaps by gaining physical access to their computer and rooting through their cookies - they could send a request to the server with this stolen authentication ticket and gain entry.

While the expiry doesn't prevent this scenario, it does limit the window during which such an attack can succeed.

> **NOTE**
>
> Step 3 details additional techniques used by the forms authentication system to protect the authentication ticket.

When creating the authentication ticket, the forms authentication system determines its expiry by consulting the timeout setting. As noted in Table 1, the timeout setting defaults to 30 minutes, meaning that when the forms authentication ticket is created its expiry is set to a date and time 30 minutes in the future.

The expiry defines an absolute time in the future when the forms authentication ticket expires. But usually developers want to implement a sliding expiry, one that is reset every time the user revisits the site. This behavior is determined by the slidingExpiration settings. If set to true (the default), each time the FormsAuthenticationModule authenticates a user, it updates the ticket's expiry. If set to false, the expiry is not updated on each request, thereby causing the ticket to expire exactly timeout number of minutes past when the ticket was first created.

> **NOTE**
>
> The expiry stored in the authentication ticket is an absolute date and time value, like August 2, 2008 11:34 AM. Moreover, the date and time are relative to the web server's local time. This design decision can have some interesting side effects around Daylight Saving Time (DST), which is when clocks in the United States are moved ahead one hour (assuming the web server is hosted in a locale where Daylight Saving Time is observed). Consider what would happen for an ASP.NET website with a 30 minute expiry near the time that DST begins (which is at 2:00 AM). Imagine a visitor signs on to the site on March 11, 2008 at 1:55 AM. This would generate a forms authentication ticket that expires at March 11, 2008 at 2:25 AM (30 minutes in the future). However, once 2:00 AM rolls around, the clock jumps to 3:00 AM because of DST. When the user loads a new page six minutes after signing in (at 3:01 AM), the FormsAuthenticationModule notes that the ticket has expired and redirects the user to the login page. For a more thorough discussion on this and other authentication ticket timeout oddities, as well as workarounds, pick up a copy of Stefan Schackow's *Professional ASP.NET 2.0 Security, Membership, and Role Management* (ISBN: 978-0-7645-9698-8).

Figure 1 illustrates the workflow when slidingExpiration is set to false and timeout is set to 30. Note that the authentication ticket generated at login contains the expiration date, and this value is not updated on subsequent requests. If the FormsAuthenticationModule finds that the ticket has expired, it discards it and treats the request as anonymous.

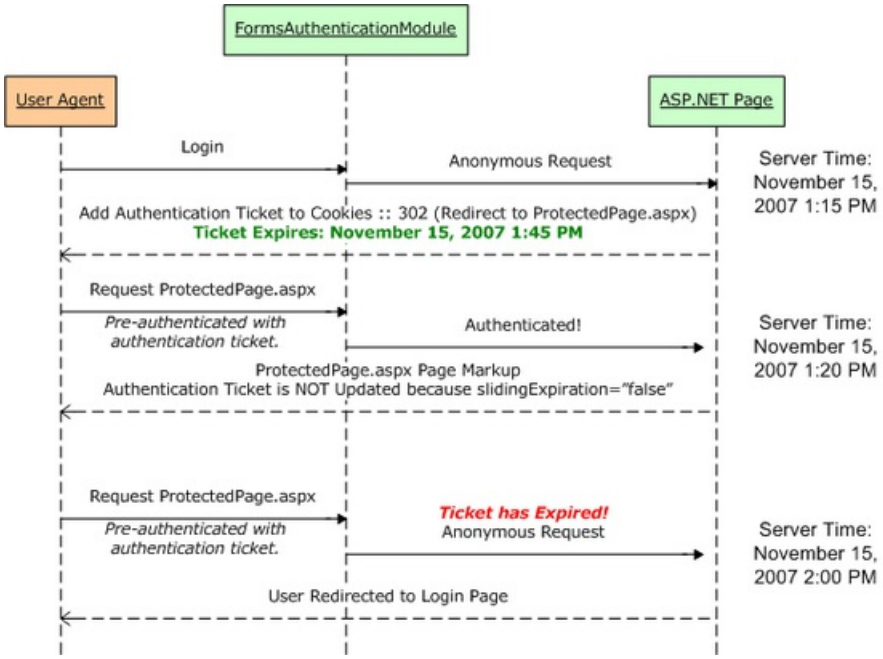


Figure 01: A Graphical Representation of the Forms Authentication Ticket's Expiry When slidingExpiration is false (Click to view full-size image)

Figure 2 shows the workflow when slidingExpiration is set to true and timeout is set to 30. When an authenticated request is received (with a non-expired ticket) its expiry is updated to timeout number of minutes in the future.

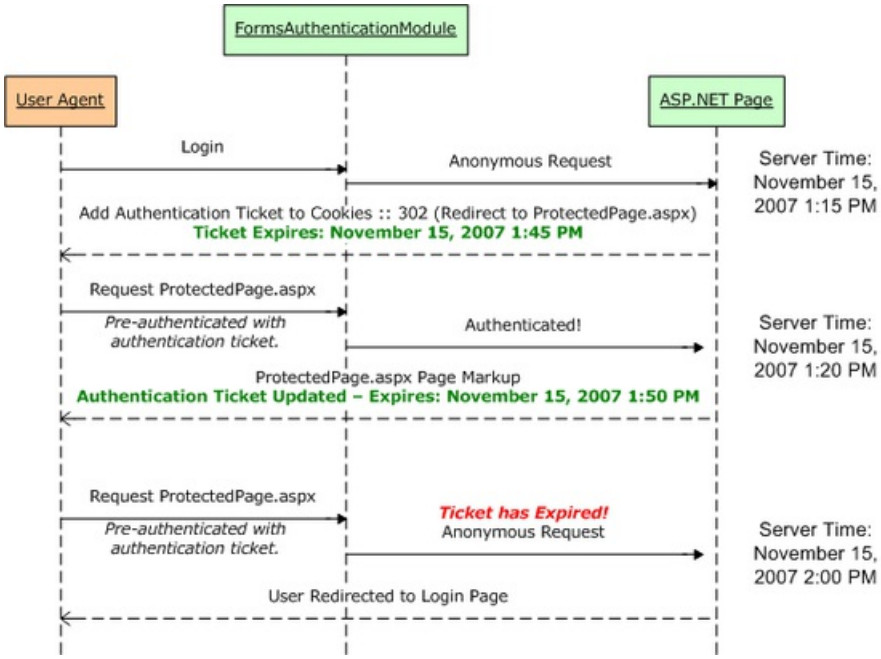


Figure 02: A Graphical Representation of the Forms Authentication Ticket's When slidingExpiration is true (Click to view full-size image)

When using cookie-based authentication tickets (the default), this discussion becomes a little more confusing because cookies can also have their own expiries specified. A cookie's expiry (or lack thereof) instructs the browser when the cookie should be destroyed. If the cookie lacks an expiry, it is destroyed when the browser shuts down. If an expiry is present, however, the cookie remains stored on the user's computer until the date and time specified in the expiry has passed. When a cookie is destroyed by the browser, it is no longer sent to the web server. Therefore, the destruction of a cookie is analogous to the user logging out of the site.

> **NOTE**
>
> Of course, a user may proactively remove any cookies stored on their computer. In Internet Explorer 7, you would go to Tools, Options, and click the Delete button in the Browsing history section. From there, click the Delete cookies button.

The forms authentication system creates session-based or expiry-based cookies depending on the value passed in to the *persistCookie* parameter. Recall that the FormsAuthentication class's GetAuthCookie, SetAuthCookie, and RedirectFromLoginPage methods take in two input parameters: *username* and *persistCookie*. The login page we created in the preceding tutorial included a Remember me CheckBox, which determined whether a persistent cookie was created. Persistent cookies are expiry-based; non-persistent cookies are session-based.

The timeout and slidingExpiration concepts already discussed apply the same to both session- and expiry-based cookies. There is only one minor difference in execution: when using expiry-based cookies with slidingTimeout set to true, the cookie's expiry is only updated when more than half of the specified time has elapsed.

Let's update our website's authentication ticket timeout policies so that tickets timeout after one hour (60 minutes), using a sliding expiration. To effect this change, update the Web.config file, adding a <forms> element to the <authentication> element with the following markup:

```
<authentication mode="Forms">

 <forms

 slidingExpiration="true"

 timeout="60"

 />

</authentication>
```

Using an Login Page URL Other than Login.aspx

Since the FormsAuthenticationModule automatically redirects unauthorized users to the login page, it needs to know the login page's URL. This URL is specified by the loginUrl attribute in the <forms> element and defaults to login.aspx. If you are porting over an existing website, you may already have a login page with a different URL, one that has already been bookmarked and indexed by search engines. Rather than renaming your existing login page to login.aspx and breaking links and users' bookmarks, you can instead modify the loginUrl attribute to point to your login page.

For example, if your login page was named SignIn.aspx and was located in the Users directory, you could point the loginUrl configuration setting to ~/Users/SignIn.aspx like so:

```
<authentication mode="Forms">

 <forms

 loginUrl="~/Users/SignIn.aspx"

 />

</authentication>
```

Since our current application already has a login page named Login.aspx, there's no need to specify a custom value in the <forms> element.

Step 2: Using Cookieless Forms Authentication Tickets

By default the forms authentication system determines whether to store its authentication tickets in the cookies collection or embed them in the URL based on the user agent visiting the site. All mainstream desktop browsers like Internet Explorer, Firefox, Opera, and Safari, support cookies, but not all mobile devices do.

The cookie policy used by the forms authentication system depends on the cookieless setting in the <forms> element, which can be assigned one of four values:

- UseCookies - specifies that cookie-based authentication tickets will always be used.
- UseUri - indicates that cookie-based authentication tickets will never be used.
- AutoDetect - if the device profile does not support cookies, cookie-based authentication tickets are not used; if the device profile supports cookies, a probing mechanism is used to determine if cookies are enabled.
- UseDeviceProfile - the default; uses cookie-based authentication tickets only if the device profile supports cookies. No probing mechanism is used.

The AutoDetect and UseDeviceProfile settings rely on a *device profile* in ascertaining whether to use cookie-based or cookieless authentication tickets. ASP.NET maintains a database of various devices and their capabilities, such as whether they support cookies, what version of JavaScript they support, and so on. Each time a device requests a web page from a web server it sends along a *user-agent* HTTP header that identifies

the device type. ASP.NET automatically matches the supplied user-agent string with the corresponding profile specified in its database.

> **NOTE**
> This database of device capabilities is stored in a number of XML files that adhere to the Browser Definition File schema. The default device profile files are located in %WINDIR%\Microsoft.Net\Framework\v2.0.50727\CONFIG\Browsers. You can also add custom files to your application's App_Browsers folder. For more information, see How To: Detect Browser Types in ASP.NET Web Pages.

Because the default setting is UseDeviceProfile, cookieless forms authentication tickets will be used when the site is visited by a device whose profile reports that it does not support cookies.

Encoding the Authentication Ticket in the URL

Cookies are a natural medium for including information from the browser in each request to a particular website, which is why the default forms authentication settings use cookies if the visiting device supports them. If cookies are not supported, an alternate means for passing the authentication ticket from the client to the server must be employed. A common workaround used in cookieless environments is to encode the cookie data in the URL.

The best way to see how such information can be embedded within the URL is to force the site to use cookieless authentication tickets. This can be accomplished by setting the cookieless configuration setting to UseUri:

```
<authentication mode="Forms">

 <forms

 cookieless="UseUri"

 slidingExpiration="true"

 timeout="60"

 />

</authentication>
```

Once you have made this change, visit the site through a browser. When visiting as an anonymous user, the URLs will look exactly like they did before. For example, when visiting Default.aspx page my browser's address bar shows the following URL:

```
http://localhost:2448/ASPNET\_Security\_Tutorial\_03\_CS/default.aspx
```

However, upon logging in, the forms authentication ticket is embedded into the URL. For example, after visiting the login page and logging in as Sam, I am returned to the Default.aspx page, but the URL this time is:

```
http://localhost:2448/ASPNET\_Security\_Tutorial\_03\_CS/(F(jaIOIDTJxIr12xYS-VVgkqKCVAuIoW30Bu0diWi6flQC-
FyMaLXJfow\_Vd9GZkB2Cv-rfezq0gKadKX0YPZCkA2))/default.aspx
```

The forms authentication ticket has been embedded within the URL. The string (F(jaIOIDTJxIr12xYS-VVgkqKCVAuIoW30Bu0diWi6flQC-FyMaLXJfow_Vd9GZkB2Cv-rfezq0gKadKX0YPZCkA2) represents the hex-encoded authentication ticket information, and is the same data that is usually stored within a cookie.

In order for cookieless authentication tickets to work, the system must encode all URLs on the page to include the authentication ticket data, otherwise the authentication ticket will be lost when the user clicks on a link. Thankfully, this embedding logic is performed automatically. To demonstrate this functionality, open the Default.aspx page and add a HyperLink control, setting its Text and NavigateUrl properties to Test Link and SomePage.aspx, respectively. It doesn't matter that there really isn't a page in our project named SomePage.aspx.

Save the changes to Default.aspx and then visit it through a browser. Log on to the site so that the forms authentication ticket is embedded in the URL. Next, from Default.aspx, click the Test Link link. What happened? If no page named SomePage.aspx exists, then a 404 error occurred, but that's not what's important here. Instead, focus on the Address bar in your browser. Note that it includes the forms authentication ticket in the URL!

```
http://localhost:2448/ASPNET\_Security\_Tutorial\_03\_CS/(F(jaIOIDTJxIr12xYS-VVgkqKCVAuIoW30Bu0diWi6flQC-
FyMaLXJfow\_Vd9GZkB2Cv-rfezq0gKadKX0YPZCkA2))/SomePage.aspx
```

The URL SomePage.aspx in the link was automatically converted into a URL that included the authentication ticket - we didn't have to write a lick of code! The form authentication ticket will automatically be embedded in the URL for any hyperlinks that do not start with http:// or /. It doesn't matter if the hyperlink appears in a call to Response.Redirect, in a HyperLink control, or in an anchor HTML element (i.e., ...). As long as the URL isn't something like http://www.someserver.com/SomePage.aspx or /SomePage.aspx, the forms authentication ticket will be embedded for us.

> **NOTE**
>
> Cookieless forms authentication tickets adhere to the same timeout policies as cookie-based authentication tickets. However, cookieless authentication tickets are more prone to replay attacks since the authentication ticket is embedded directly in the URL. Imagine a user who visits a website, logs in, and then pastes the URL in an email to a colleague. If the colleague clicks on that link before the expiry is reached, they will be logged in as the user who sent the email!

Step 3: Securing the Authentication Ticket

The forms authentication ticket is transmitted over the wire either in a cookie or embedded directly within the URL. In addition to identity information, the authentication ticket can also include user data (as we will see in Step 4). Consequently, it is important that the ticket's data is encrypted from prying eyes and (even more importantly) that the forms authentication system can guarantee that the ticket was not tampered with.

To ensure the privacy of the ticket's data, the forms authentication system can encrypt the ticket data. Failure to encrypt the ticket data sends potentially sensitive information over the wire in plain-text.

To guarantee a ticket's authenticity, the forms authentication system must *validate* the ticket. Validation is the act of ensuring that a particular piece of data has not been modified, and is accomplished via a *message authentication code (MAC)*. In a nutshell, the MAC is a small piece of information that identifies the data that needs to be validated (in this case, the ticket). If the data represented by the MAC is modified, then the MAC and the data will not match up. Moreover, it is computationally hard for a hacker to both modify the data and generate his own MAC to correspond with the modified data.

When creating (or modifying) a ticket, the forms authentication system creates a MAC and attaches it to the ticket's data. When a subsequent request arrives, the forms authentication system compares the MAC and ticket data to validate the authenticity of the ticket data. Figure 3 illustrates this workflow graphically.

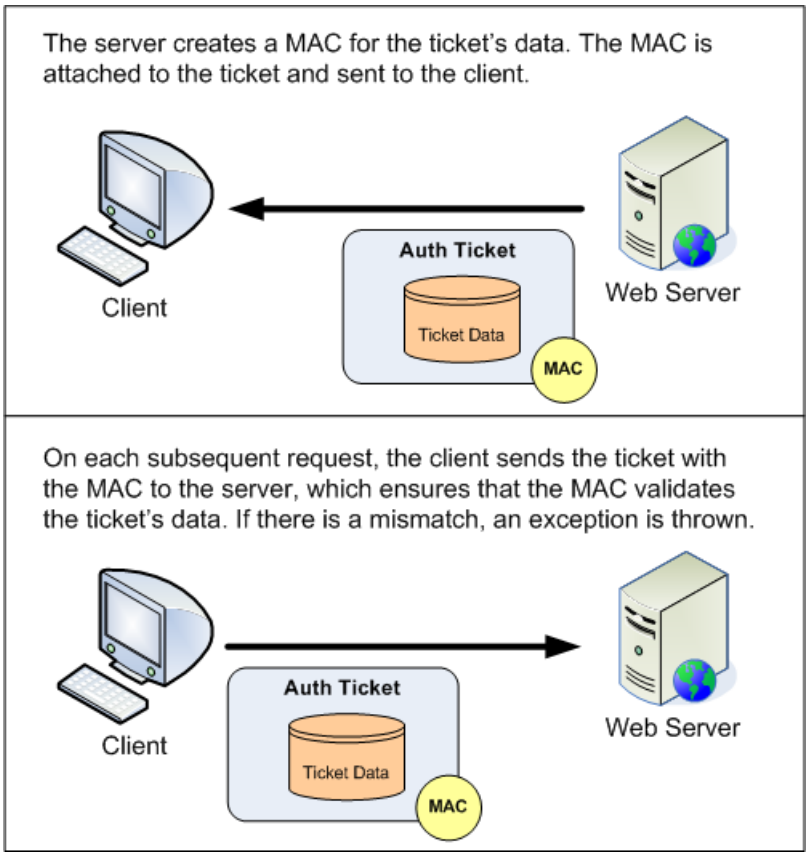


Figure 03: The Ticket's Authenticity is Ensured through a MAC (Click to view full-size image)

What security measures are applied to the authentication ticket depends on the protection setting in the <forms> element. The protection setting may be assigned to one of the following three values:

- All - the ticket is both encrypted and digitally signed (the default).
- Encryption - only encryption is applied - no MAC is generated.
- None - the ticket is neither encrypted nor digitally signed.
- Validation - a MAC is generated, but the ticket data is sent over the wire in plain-text.

Microsoft strongly recommends using the All setting.

Setting the Validation and Decryption Keys

The encryption and hashing algorithms used by the forms authentication system to encrypt and validate the authentication ticket are customizable through the <machineKey> element in Web.config. Table 2 outlines the <machineKey> element's attributes and their possible values.

ATTRIBUTE	DESCRIPTION
decryption	Indicates the algorithm used for encryption. This attribute can have one of the following four values: - Auto - the default; determines the algorithm based on the length of the decryptionKey attribute. - AES - uses the Advanced Encryption Standard (AES) algorithm. - DES - uses the Data Encryption Standard (DES) This algorithm is considered computationally weak and should not be used. - 3DES - uses the Triple DES algorithm, which works by applying the DES algorithm three times.

ATTRIBUTE	DESCRIPTION
decryptionKey	The secret key used by the encryption algorithm. This value must either be a hexadecimal string of the appropriate length (based on the value in decryption), AutoGenerate, or either value appended with,IsolateApps. Adding IsolateApps instructs ASP.NET to use a unique value for each application. The default is AutoGenerate,IsolateApps.
validation	Indicates the algorithm used for validation. This attribute can have one of the following four values: - AES - uses the Advanced Encryption Standard (AES) algorithm. - MD5 - uses the Message-Digest 5 (MD5) algorithm. - SHA1 - uses the SHA1 algorithm (the default). - 3DES - uses the Triple DES algorithm.
validationKey	The secret key used by the validation algorithm. This value must either be a hexadecimal string of the appropriate length (based on the value in validation), AutoGenerate, or either value appended with,IsolateApps. Adding IsolateApps instructs ASP.NET to use a unique value for each application. The default is AutoGenerate,IsolateApps.

Table 2: The <machineKey> Element Attributes

A thorough discussion of these encryption and validation options, and the pros and cons of the various algorithms, is beyond the scope of this tutorial. For an in-depth look at these issues, including guidance on what encryption and validation algorithms to use, what key lengths to use, and how best to generate these keys, refer to *Professional ASP.NET 2.0 Security, Membership, and Role Management*.

By default, the keys used for encryption and validation are generated automatically for each application, and these keys are stored in the Local Security Authority (LSA). In short, the default settings guarantee unique keys on a web server-by-web server and application-by-application basis. Consequently, this default behavior will not work for the two following scenarios:

- **Web Farms** - in a web farm scenario, a single web application is hosted on multiple web servers for purposes of scalability and redundancy. Each incoming request is dispatched to a server in the farm, meaning that over the lifetime of a user's session, different servers may be used to handle his various requests. Consequently, each server must use the same encryption and validation keys so that the forms authentication ticket created, encrypted, and validated on one server can be decrypted and validated on a different server in the farm.
- **Cross Application Ticket Sharing** - a single web server may host multiple ASP.NET applications. If you need for these different applications to share a single forms authentication ticket, it is imperative that their encryption and validation keys match up.

When working in a web farm setting or sharing authentication tickets across applications on the same server, you will need to configure the <machineKey> element in the affected applications so that their decryptionKey and validationKey values match up.

While neither of the above scenarios applies to our sample application, we can still specify explicit decryptionKey and validationKey values and define the algorithms to be used. Add a <machineKey> setting to the Web.config file:

```
<configuration>

 <system.web>

 ... Some markup was removed for brevity ...

 <machineKey

 decryption="AES"

 validation="SHA1"

 decryptionKey="1513F567EE75F7FB5AC0AC4D79E1D9F25430E3E2F1BCDD3370BCFC4EFC97A541"

 validationKey="32CBA563F26041EE5B5FE9581076C40618DCC1218F5F447634EDE8624508A129"

 />

 </system.web>

</configuration>
```

For more information check out How To: Configure MachineKey in ASP.NET 2.0.

> **NOTE**
>
> The decryptionKey and validationKey values were taken from Steve Gibson's Perfect Passwords web page, which generates 64 random hexadecimal characters on each page visit. To lessen the likelihood of these keys making their way into your production applications, you are encouraged to replace the above keys with randomly generated ones from the Perfect Passwords page.

Step 4: Storing Additional User Data in the Ticket

Many web applications display information about or base the page's display on the currently logged on user. For example, a web page might show the user's name and the date she last logged on in the upper corner of every page. The forms authentication ticket stores the currently logged on user's username, but when any other information is needed, the page must go to the user store - typically a database - to lookup the information not stored in the authentication ticket.

With a little bit of code we can store additional user information in the forms authentication ticket. Such data can be expressed through the FormsAuthenticationTicket class's UserData property. This is a useful place to put small amounts of information about the user that is commonly needed. The value specified in the UserData property is included as part of the authentication ticket cookie and, like the other ticket fields, is encrypted and validated based on the forms authentication system's configuration. By default, UserData is an empty string.

In order to store user data in the authentication ticket, we need to write a bit of code in the login page that grabs the user-specific information and stores it in the ticket. Since UserData is a property of type String, the data stored in it must be properly serialized as a string. For example, imagine that our user store included each user's date of birth and the name of their employer, and we wanted to store these two property values in the authentication ticket. We could serialize these values into a string by concatenating the user's date of birth's string with a pipe (|), followed by the employer name. For a user born on August 15, 1974 that works for Northwind Traders, we would assign the UserData property the string: 1974-08-15|Northwind Traders .

Whenever we need to access the data stored in the ticket, we can do so by grabbing the current request's FormsAuthenticationTicket and deserializing the UserData property. In the case of the date of birth and employer name example, we would split the UserData string into two substrings based on the delimiter (|).

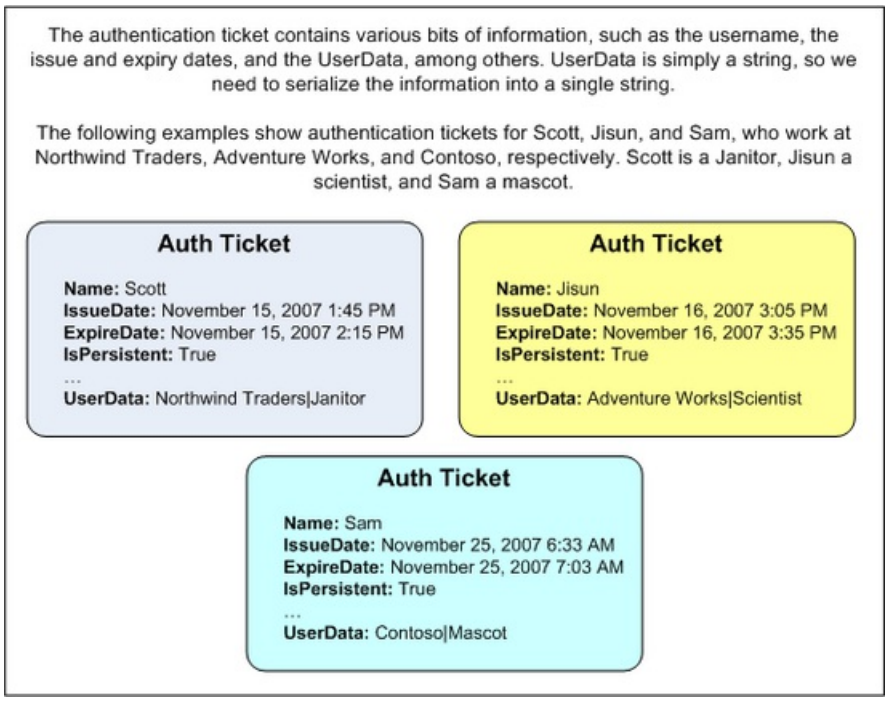


Figure 04: Additional User Information Can Be Stored in the Authentication Ticket (Click to view full-size image)

Writing Information to UserData

Unfortunately, adding user-specific information to a forms authentication ticket is not as straightforward as one might expect. The UserData property of the FormsAuthenticationTicket class is read-only and can only be specified through the FormsAuthenticationTicket class constructor. When specifying the UserData property in the constructor, we also need to provide the ticket's other values: the username, the issue date, the expiration, and so on. When we created the login page in the preceding tutorial, this was all handled for us by the FormsAuthentication class. When adding UserData to the FormsAuthenticationTicket, we will need to write code to replicate much of the functionality already provided by the FormsAuthentication class.

Let's explore the necessary code for working with UserData by updating the Login.aspx page to record additional information about the user to the authentication ticket. Pretend that our user store contains information about the company the user works for and their title, and that we want to capture this information in the authentication ticket. Update the Login.aspx page's LoginButton Click event handler so that the code looks like the following:

```
Protected Sub LoginButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
LoginButton.Click

 ' Three valid username/password pairs: Scott/password, Jisun/password, and Sam/password.

 Dim users() As String = {"Scott", "Jisun", "Sam"}

 Dim passwords() As String = {"password", "password", "password"}

 Dim companyName() As String = {"Northwind Traders", "Adventure Works", "Contoso"}

 Dim titleAtCompany() As String = {"Janitor", "Scientist", "Mascot"}

 For i As Integer = 0 To users.Length - 1

 Dim validUsername As Boolean = (String.Compare(UserName.Text, users(i), True) = 0)

 Dim validPassword As Boolean = (String.Compare(Password.Text, passwords(i), False) = 0)

 If validUsername AndAlso validPassword Then

 ' Query the user store to get this user's User Data

 Dim userDataString As String = String.Concat(companyName(i), "|", titleAtCompany(i))

 ' Create the cookie that contains the forms authentication ticket

 Dim authCookie As HttpCookie = FormsAuthentication.GetAuthCookie(UserName.Text, RememberMe.Checked)

 ' Get the FormsAuthenticationTicket out of the encrypted cookie

 Dim ticket As FormsAuthenticationTicket = FormsAuthentication.Decrypt(authCookie.Value)

 ' Create a new FormsAuthenticationTicket that includes our custom User Data

 Dim newTicket As FormsAuthenticationTicket = New FormsAuthenticationTicket(ticket.Version, ticket.Name,
ticket.IssueDate, ticket.Expiration, ticket.IsPersistent, userDataString)

 ' Update the authCookie's Value to use the encrypted version of newTicket

 authCookie.Value = FormsAuthentication.Encrypt(newTicket)

 ' Manually add the authCookie to the Cookies collection

 Response.Cookies.Add(authCookie)

 ' Determine redirect URL and send user there

 Dim redirUrl As String = FormsAuthentication.GetRedirectUrl(UserName.Text, RememberMe.Checked)

 Response.Redirect(redirUrl)

 End If

 Next

 ' If we reach here, the user's credentials were invalid

 InvalidCredentialsMessage.Visible = True

End Sub
```

Let's step through this code one line at a time. The method starts by defining four string arrays: users, passwords, companyName, and titleAtCompany. These arrays hold the usernames, passwords, company names, and titles for the user accounts in the system, of which there are three: Scott, Jisun, and Sam. In a real

application, these values would be queried from the user store, not hard-coded in the page's source code.

In the previous tutorial, if the supplied credentials were valid we simply called FormsAuthentication.RedirectFromLoginPage(UserName.Text, RememberMe.Checked), which performed the following steps:

1. Created the forms authentication ticket
2. Wrote the ticket to the appropriate store. For cookies-based authentication tickets, the browser's cookies collection is used; for cookieless authentication tickets, the ticket data is serialized into the URL
3. Redirected the user to the appropriate page

These steps are replicated in the code above. First, the string we will eventually store in the UserData property is formed by combining the company name and the title, delimiting the two values with a pipe character (|).

Dim userDataString As String = String.Concat(companyName(i), "|", titleAtCompany(i))

Next, the FormsAuthentication.GetAuthCookie method is invoked, which creates the authentication ticket, encrypts and validates it according to the configuration settings, and places it in an HttpCookie object.

Dim authCookie As HttpCookie = FormsAuthentication.GetAuthCookie(UserName.Text, RememberMe.Checked)

In order to work with the FormAuthenticationTicket embedded within the cookie, we need to call the FormAuthentication class's Decrypt method, passing in the cookie value.

Dim ticket As FormsAuthenticationTicket = FormsAuthentication.Decrypt(authCookie.Value)

We then create a *new* FormsAuthenticationTicket instance based on the existing FormsAuthenticationTicket's values. However, this new ticket includes the user-specific information (userDataString).

Dim newTicket As FormsAuthenticationTicket = New FormsAuthenticationTicket(ticket.Version, ticket.Name, ticket.IssueDate, ticket.Expiration, ticket.IsPersistent, userDataString)

We then encrypt (and validate) the new FormsAuthenticationTicket instance by calling the Encrypt method, and put this encrypted (and validated) data back into authCookie.

authCookie.Value = FormsAuthentication.Encrypt(newTicket)

Finally, authCookie is added to the Response.Cookies collection and the GetRedirectUrl method is called to determine the appropriate page to send the user.

```
Response.Cookies.Add(authCookie)

Dim redirUrl As String = FormsAuthentication.GetRedirectUrl(UserName.Text, RememberMe.Checked)

Response.Redirect(redirUrl)
```

All of this code is needed because the UserData property is read-only and the FormsAuthentication class does not provide any methods for specifying UserData information in its GetAuthCookie, SetAuthCookie, or RedirectFromLoginPage methods.

> **NOTE**
> The code we just examined stores user-specific information in a cookies-based authentication ticket. The classes responsible for serializing the forms authentication ticket to the URL are internal to the .NET Framework. Long story short, you cannot store user data in a cookieless forms authentication ticket.

Accessing the UserData Information

At this point each user's company name and title is stored in the forms authentication ticket's UserData property

when they log in. This information can be accessed from the authentication ticket on any page without requiring a trip to the user store. To illustrate how this information can be retrieved from the UserData property, let's update Default.aspx so that its welcome message includes not only the user's name, but also the company they work for and their title.

Currently, Default.aspx contains an AuthenticatedMessagePanel Panel with a Label control named WelcomeBackMessage. This Panel is only displayed to authenticated users. Update the code in Default.aspx's Page_Load event handler so that it looks like the following:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load

 If Request.IsAuthenticated Then

 WelcomeBackMessage.Text = "Welcome back, " & User.Identity.Name & "!"

 ' Get User Data from FormsAuthenticationTicket and show it in WelcomeBackMessage

 Dim ident As FormsIdentity = CType(User.Identity, FormsIdentity)

 If ident IsNot Nothing Then

 Dim ticket As FormsAuthenticationTicket = ident.Ticket

 Dim userDataString As String = ticket.UserData

 ' Split on the |

 Dim userDataPieces() As String = userDataString.Split("|".ToCharArray())

 Dim companyName As String = userDataPieces(0)

 Dim titleAtCompany As String = userDataPieces(1)

 WelcomeBackMessage.Text &= String.Format(" You are the {0} of {1}.", titleAtCompany, companyName)

 End If

 AuthenticatedMessagePanel.Visible = True

 AnonymousMessagePanel.Visible = False

 Else

 AuthenticatedMessagePanel.Visible = False

 AnonymousMessagePanel.Visible = True

 End If

End Sub
```

If Request.IsAuthenticated is True, then the WelcomeBackMessage's Text property is first set to Welcome back, *username*. Then, the User.Identity property is cast to a FormsIdentity object so that we can access the underlying FormsAuthenticationTicket. Once we have the FormsAuthenticationTicket, we deserialize the UserData property into the company name and title. This is accomplished by splitting the string on the pipe character. The company name and title are then displayed in the WelcomeBackMessage Label.

Figure 5 shows a screenshot of this display in action. Logging in as Scott displays a welcome back message that includes Scott's company and title.

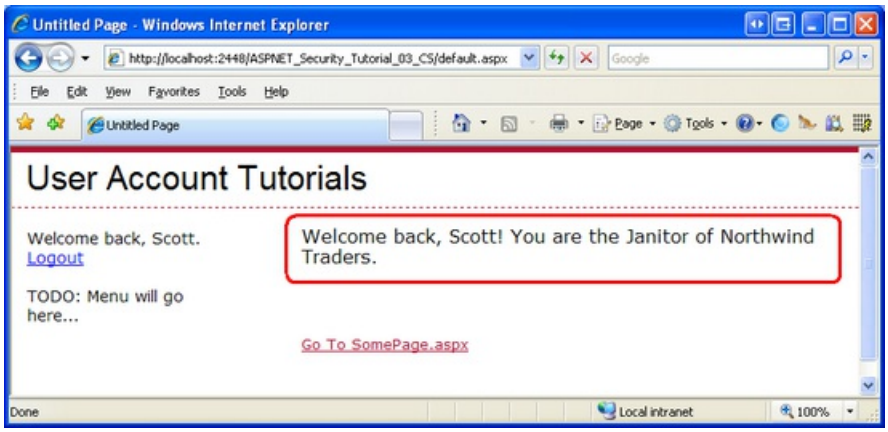


Figure 05: The Currently Logged On User's Company and Title are Displayed (Click to view full-size image)

> **NOTE**
>
> The authentication ticket's UserData property serves as a cache for the user store. Like any cache, it needs to be updated when the underlying data is modified. For example, if there is a web page from which users can update their profile, the fields cached in the UserData property must be refreshed to reflect the changes made by the user.

Step 5: Using a Custom Principal

On each incoming request the FormsAuthenticationModule attempts to authenticate the user. If a non-expired authentication ticket is present, the FormsAuthenticationModule assigns the HttpContext.User property to a new GenericPrincipal object. This GenericPrincipal object has an Identity of type FormsIdentity, which includes a reference to the forms authentication ticket. The GenericPrincipal class contains the bare minimum functionality needed by a class that implements IPrincipal - it just has an Identity property and an IsInRole method.

The principal object has two responsibilities: to indicate what roles the user belongs to and to provide identity information. This is accomplished through the IPrincipal interface's IsInRole(*roleName*) method and Identity property, respectively. The GenericPrincipal class allows for a string array of role names to be specified via its constructor; its IsInRole(*roleName*) method merely checks to see if the passed in *roleName* exists within the string array. When the FormsAuthenticationModule creates the GenericPrincipal, it passes in an empty string array to the GenericPrincipal's constructor. Consequently, any call to IsInRole will always return False.

The GenericPrincipal class meets the needs for most forms based authentication scenarios where roles are not used. For those situations where the default role handling is insufficient or when you need to associate a custom IIdentity object with the user, you can create a custom IPrincipal object during the authentication workflow and assign it to the HttpContext.User property.

> **NOTE**
>
> As we will see in future tutorials, when ASP.NET's Roles framework is enabled it creates a custom principal object of type RolePrincipal and overwrites the forms authentication-created GenericPrincipal object. It does this in order to customize the principal's IsInRole method to interface with the Roles framework's API.

Since we have not concerned ourselves with roles yet, the only reason we would have for creating a custom principal at this juncture would be to associate a custom IIdentity object to the principal. In Step 4 we looked at storing additional user information in the authentication ticket's UserData property, in particular the user's company name and their title. However, the UserData information is only accessible through the authentication ticket and only then as a serialized string, meaning that anytime we want to view the user information stored in the ticket we need to parse the UserData property.

We can improve the developer experience by creating a class that implements IIdentity and includes

CompanyName and Title properties. That way, a developer can access the currently logged on user's company name and title directly through the CompanyName and Title properties without needed to know how to parse the UserData property.

Creating the Custom Identity and Principal Classes

For this tutorial, let's create the custom principal and identity objects in the App_Code folder. Start by adding an App_Code folder to your project - right-click on the project name in Solution Explorer, select the Add ASP.NET Folder option, and choose App_Code. The App_Code folder is a special ASP.NET folder that holds class files specific to the website.

> **NOTE**
>
> The App_Code folder should only be used when managing your project through the Website Project Model. If you are using the Web Application Project Model, create a standard folder and add the classes to that. For example, you could add a new folder named Classes, and place your code there.

Next, add two new class files to the App_Code folder, one named CustomIdentity.vb and one named CustomPrincipal.vb.

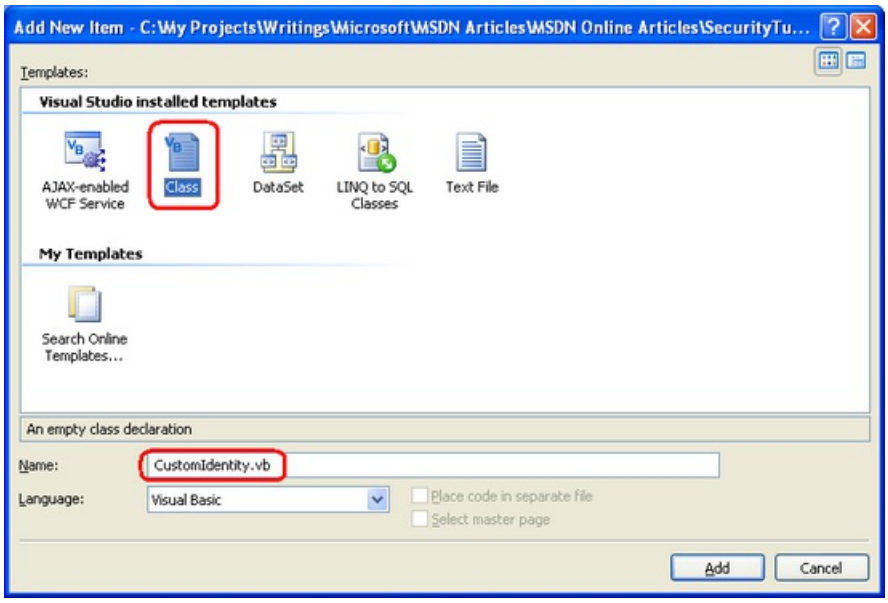


Figure 06: Add the CustomIdentity and CustomPrincipal Classes to Your Project (Click to view full-size image)

The CustomIdentity class is responsible for implementing the IIdentity interface, which defines the AuthenticationType, IsAuthenticated, and Name properties. In addition to those required properties, we are interested in exposing the underlying forms authentication ticket as well as properties for the user's company name and title. Enter the following code into the CustomIdentity class.

```
Imports Microsoft.VisualBasic

Public Class CustomIdentity

 Implements System.Security.Principal.IIdentity

 Private _ticket As FormsAuthenticationTicket

 Public Sub New(ByVal ticket As FormsAuthenticationTicket)

 _ticket = ticket

 End Sub

 Public ReadOnly Property AuthenticationType() As String Implements
```

```
System.Security.Principal.IIdentity.AuthenticationType

 Get

 Return "Custom"

 End Get

 End Property

 Public ReadOnly Property IsAuthenticated() As Boolean Implements
System.Security.Principal.IIdentity.IsAuthenticated

 Get

 Return True

 End Get

 End Property

 Public ReadOnly Property Name() As String Implements System.Security.Principal.IIdentity.Name

 Get

 Return _ticket.Name

 End Get

 End Property

 Public ReadOnly Property Ticket() As FormsAuthenticationTicket

 Get

 Return _ticket

 End Get

 End Property

 Public ReadOnly Property CompanyName() As String

 Get

 Dim userDataPieces() As String = _ticket.UserData.Split("|".ToCharArray())

 Return userDataPieces(0)

 End Get

 End Property

 Public ReadOnly Property Title() As String

 Get

 Dim userDataPieces() As String = _ticket.UserData.Split("|".ToCharArray())

 Return userDataPieces(1)

 End Get

 End Property

End Class
```

Note that the class includes a FormsAuthenticationTicket member variable (_ticket) and that this ticket

information must be supplied through the constructor. This ticket data is used in returning the identity's Name; its UserData property is parsed to return the values for the CompanyName and Title properties.

Next, create the CustomPrincipal class. Since we are not concerned with roles at this juncture, the CustomPrincipal class's constructor accepts only a CustomIdentity object; its IsInRole method always returns False.

```
Imports Microsoft.VisualBasic

Public Class CustomPrincipal

 Implements System.Security.Principal.IPrincipal

 Private _identity As CustomIdentity

 Public Sub New(ByVal identity As CustomIdentity)

 _identity = identity

 End Sub

 Public ReadOnly Property Identity() As System.Security.Principal.IIdentity Implements
System.Security.Principal.IPrincipal.Identity

 Get

 Return _identity

 End Get

 End Property

 Public Function IsInRole(ByVal role As String) As Boolean Implements
System.Security.Principal.IPrincipal.IsInRole

 Return False

 End Function

End Class
```

Assigning a CustomPrincipal Object to the Incoming Request's Security Context

We now have a class that extends the default IIdentity specification to include CompanyName and Title properties, as well as a custom principal class that uses the custom identity. We are ready to step into the ASP.NET pipeline and assign our custom principal object to the incoming request's security context.

The ASP.NET pipeline takes an incoming request and processes it through a number of steps. At each step, a particular event is raised, making it possible for developers to tap into the ASP.NET pipeline and modify the request at certain points in its lifecycle. The FormsAuthenticationModule, for example, waits for ASP.NET to raise the AuthenticateRequest event, at which point it inspects the incoming request for an authentication ticket. If an authentication ticket is found, a GenericPrincipal object is created and assigned to the HttpContext.User property.

After the AuthenticateRequest event, the ASP.NET pipeline raises the PostAuthenticateRequest event, which is where we can replace the GenericPrincipal object created by the FormsAuthenticationModule with an instance of our CustomPrincipal object. Figure 7 depicts this workflow.

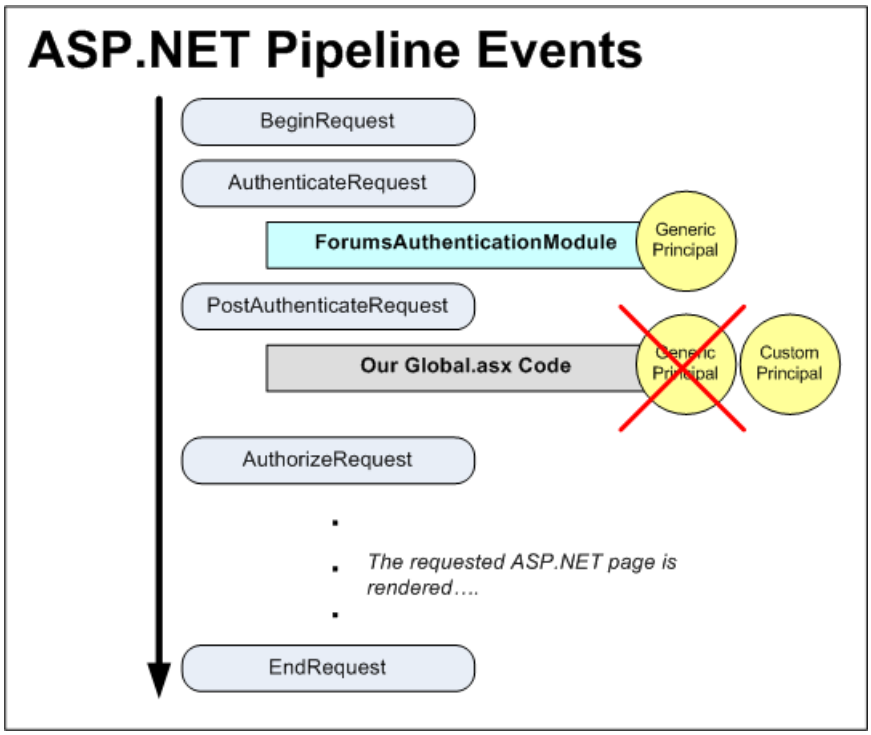


Figure 07: The GenericPrincipal is Replaced by a CustomPrincipal in the PostAuthenticationRequest Event (Click to view full-size image)

In order to execute code in response to an ASP.NET pipeline event, we can either create the appropriate event handler in Global.asax or create our own HTTP Module. For this tutorial let's create the event handler in Global.asax. Start by adding Global.asax to your website. Right-click on the project name in Solution Explorer and add an item of type Global Application Class named Global.asax.

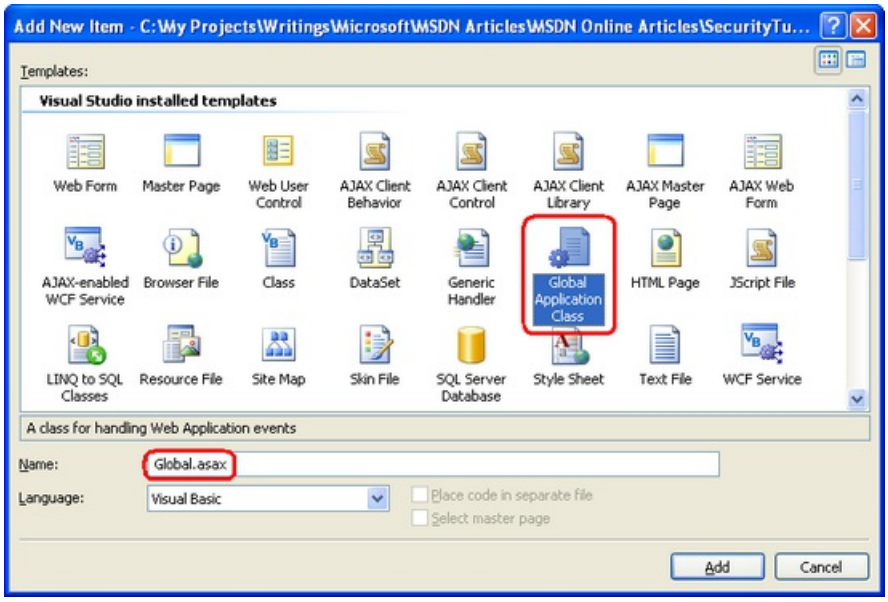


Figure 08: Add a Global.asax File to Your Website (Click to view full-size image)

The default Global.asax template includes event handlers for a number of the ASP.NET pipeline events, including the Start, End and Error event, among others. Feel free to remove these event handlers, as we do not need them for this application. The event we are interested in is PostAuthenticateRequest. Update your Global.asax file so its markup looks similar to the following:

```
<%@ Application Language="VB" %>

<%@ Import Namespace="System.Security.Principal" %>

<%@ Import Namespace="System.Threading" %>

<script runat="server">

 Sub Application_OnPostAuthenticateRequest(ByVal sender As Object, ByVal e As EventArgs)

 ' Get a reference to the current User

 Dim usr As IPrincipal = HttpContext.Current.User

 ' If we are dealing with an authenticated forms authentication request

 If usr.Identity.IsAuthenticated AndAlso usr.Identity.AuthenticationType = "Forms" Then

 Dim fIdent As FormsIdentity = CType(usr.Identity, FormsIdentity)

 ' Create a CustomIdentity based on the FormsAuthenticationTicket

 Dim ci As New CustomIdentity(fIdent.Ticket)

 ' Create the CustomPrincipal

 Dim p As New CustomPrincipal(ci)

 ' Attach the CustomPrincipal to HttpContext.User and Thread.CurrentPrincipal

 HttpContext.Current.User = p

 Thread.CurrentPrincipal = p

 End If

 End Sub

</script>
```

The Application_OnPostAuthenticateRequest method executes each time the ASP.NET runtime raises the PostAuthenticateRequest event, which happens once on each incoming page request. The event handler starts by checking to see if the user is authenticated and was authenticated via forms authentication. If so, a new CustomIdentity object is created and passed the current request's authentication ticket in its constructor. Following that, a CustomPrincipal object is created and passed the just-created CustomIdentity object in its constructor. Finally, the current request's security context is assigned to the newly created CustomPrincipal object.

Note that the last step - associating the CustomPrincipal object with the request's security context - assigns the principal to two properties: HttpContext.User and Thread.CurrentPrincipal. These two assignments are necessary because of the way security contexts are handled in ASP.NET. The .NET Framework associates a security context with each running thread; this information is available as an IPrincipal object through the Thread object's CurrentPrincipal property. What is a confusing is that ASP.NET has its own security context information (HttpContext.User).

In certain scenarios, the Thread.CurrentPrincipal property is examined when determining the security context; in other scenarios, HttpContext.User is used. For example, there are security features in .NET that allow developers to declaratively state what users or roles can instantiate a class or invoke specific methods (see Adding Authorization Rules to Business and Data Layers Using PrincipalPermissionAttributes). Underneath the covers, these declarative techniques determine the security context via the Thread.CurrentPrincipal property.

In other scenarios, the HttpContext.User property is used. For example, in the previous tutorial we used this

property to display the currently logged on user's username. Clearly, then, it is imperative that the security context information in the Thread.CurrentPrincipal and HttpContext.User properties match up.

The ASP.NET runtime automatically syncs these property values for us. However, this synchronization occurs after the AuthenticateRequest event, but *before* the PostAuthenticateRequest event. Consequently, when adding a custom principal in the PostAuthenticateRequest event we need to be certain to manually assign the Thread.CurrentPrincipal or else Thread.CurrentPrincipal and HttpContext.User will be out of sync. See Context.User vs. Thread.CurrentPrincipal for a more detailed discussion on this issue.

Accessing the CompanyName and Title Properties

Whenever a request arrives and is dispatched to the ASP.NET engine, the Application_OnPostAuthenticateRequest event handler in Global.asax will fire. If the request has been successfully authenticated by the FormsAuthenticationModule, the event handler will create a new CustomPrincipal object with a CustomIdentity object based on the forms authentication ticket. With this logic in place, accessing information about the currently logged on user's company name and title is incredibly straightforward.

Return to the Page_Load event handler in Default.aspx, where in Step 4 we wrote code to retrieve the form authentication ticket and parse the UserData property in order to display the user's company name and title. With the CustomPrincipal and CustomIdentity objects in use now, there's no need to parse the values out of the ticket's UserData property. Instead, simply get a reference to the CustomIdentity object and use its CompanyName and Title properties:

```
Dim ident As CustomIdentity = CType(User.Identity, CustomIdentity)

If ident IsNot Nothing Then

 WelcomeBackMessage.Text &= String.Format(" You are the {0} of {1}.", ident.Title, ident.CompanyName)

End If
```

Summary

In this tutorial we examined how to customize the forms authentication system's settings via Web.config. We looked at how the authentication ticket's expiration is handled and how the encryption and validation safeguards are used to protect the ticket from inspection and modification. Finally, we discussed using the authentication ticket's UserData property to store additional user information in the ticket itself, and how to use custom principal and identity objects to expose this information in a more developer-friendly manner.

This tutorial concludes our examination of forms authentication in ASP.NET. The next tutorial starts our journey into the Membership framework.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Dissecting Forms Authentication
- Explained: Forms Authentication in ASP.NET 2.0
- How To: Protect Forms Authentication in ASP.NET 2.0
- Professional ASP.NET 2.0 Security, Membership, and Role Management (ISBN: 978-0-7645-9698-8)
- Securing Login Controls
- The <authentication> Element
- The <forms> Element for <authentication>

- The <machineKey> Element
- Understanding the Forms Authentication Ticket and Cookie

Video Training on Topics Contained in this Tutorial

- How to Change the Forms Authentication Properties
- How to Setup and Use Cookie-less Authentication in an ASP.NET Application
- ASP Forms Login Relocation
- Forms Login Custom Key Configuration
- Add Custom Data to the Authentication Method
- Use Custom Principal Objects

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Alicja Maziarz. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

ASP.NET 3.5 - Membership

2/20/2020 • 2 minutes to read • Edit Online

This tutorial series examines techniques for using the built-in membership (security) system to create user accounts and configure authorization rules.

- Creating the Membership Schema in SQL Server (C#)
- Creating User Accounts (C#)
- Validating User Credentials Against the Membership User Store (C#)
- User-Based Authorization (C#)
- Storing Additional User Information (C#)
- Creating the Membership Schema in SQL Server (VB)
- Creating User Accounts (VB)
- Validating User Credentials Against the Membership User Store (VB)
- User-Based Authorization (VB)
- Storing Additional User Information (VB)

Creating the Membership Schema in SQL Server (C#)

9/16/2020 • 27 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

This tutorial starts by examining techniques for adding the necessary schema to the database in order to use the SqlMembershipProvider. Following that, we will examine the key tables in the schema and discuss their purpose and importance. This tutorial ends with a look at how to tell an ASP.NET application which provider the Membership framework should use.

Introduction

The previous two tutorials examined using forms authentication to identify website visitors. The forms authentication framework makes it easy for developers to log a user into a website and to remember them across page visits through the use of authentication tickets. The `FormsAuthentication` class includes methods for generating the ticket and adding it to the visitor's cookies. The `FormsAuthenticationModule` examines all incoming requests and, for those with a valid authentication ticket, creates and associates a `GenericPrincipal` and a `FormsIdentity` object with the current request. Forms authentication is merely a mechanism for granting an authentication ticket to a visitor when logging in and, on subsequent requests, parsing that ticket to determine the user's identity. For a web application to support user accounts, we still need to implement a user store and add functionality to validate credentials, register new users, and the myriad of other user account-related tasks.

Prior to ASP.NET 2.0, developers were on the hook for implementing all of these user account-related tasks. Fortunately the ASP.NET team recognized this shortcoming and introduced the Membership framework with ASP.NET 2.0. The Membership framework is a set of classes in the .NET Framework that provide a programmatic interface for accomplishing core user account-related tasks. This framework is built atop the provider model, which allows developers to plug a customized implementation into a standardized API.

As discussed in the *Security Basics and ASP.NET Support* tutorial, the .NET Framework ships with two built-in Membership providers: `ActiveDirectoryMembershipProvider` and `SqlMembershipProvider`. As its name implies, the `SqlMembershipProvider` uses a Microsoft SQL Server database as the user store. In order to use this provider in an application, we need to tell the provider what database to use as the store. As you might imagine, the `SqlMembershipProvider` expects the user store database to have certain database tables, views, and stored procedures. We need to add this expected schema to the selected database.

This tutorial starts by examining techniques for adding the necessary schema to the database in order to use the `SqlMembershipProvider`. Following that, we will examine the key tables in the schema and discuss their purpose and importance. This tutorial ends with a look at how to tell an ASP.NET application which provider the Membership framework should use.

Let's get started!

Step 1: Deciding Where to Place the User Store

An ASP.NET application's data is commonly stored in a number of tables in a database. When implementing the `SqlMembershipProvider` database schema we must decide whether to place the Membership schema in the same database as the application data or in an alternate database.

I recommend locating the Membership schema in the same database as the application data for the following reasons:

- **Maintainability** ' an application whose data is encapsulated in one database is easier to understand, maintain, and deploy than an application that has two separate databases.
- **Relational Integrity** ' by locating the Membership-related tables in the same database as the application tables it is possible to establish foreign key constraints between the primary keys in the Membership-related tables and related application tables.

Decoupling the user store and application data into separate databases only makes sense if you have multiple applications that each use separate databases, but need to share a common user store.

Creating a Database

The application we have been building since the second tutorial has not yet needed a database. We need one now, however, for the user store. Let's create one and then add to it the schema required by the `SqlMembershipProvider` provider (see Step 2).

> **NOTE**
>
> Throughout this tutorial series we will be using a Microsoft SQL Server 2005 Express Edition database to store our application tables and the `SqlMembershipProvider` schema. This decision was made for two reasons: first, due to its cost - free - the Express Edition is the most readably accessible version of SQL Server 2005; second, SQL Server 2005 Express Edition databases can be placed directly in the web application's `App_Data` folder, making it a cinch to package the database and web application together in one ZIP file and to redeploy it without any special setup instructions or configuration options. If you'd prefer to follow along using a non-Express Edition version of SQL Server, feel free. The steps are virtually identical. The `SqlMembershipProvider` schema will work with any version of Microsoft SQL Server 2000 and up.

From the Solution Explorer, right-click on the `App_Data` folder and choose to Add New Item. (If you do not see an `App_Data` folder in your project, right-click on the project in Solution Explorer, select Add ASP.NET Folder, and pick `App_Data`.) From the Add New Item dialog box, choose to add a new SQL Database named `SecurityTutorials.mdf`. In this tutorial we will add the `SqlMembershipProvider` schema to this database; in subsequent tutorials we will create additional tables to capture our application data.

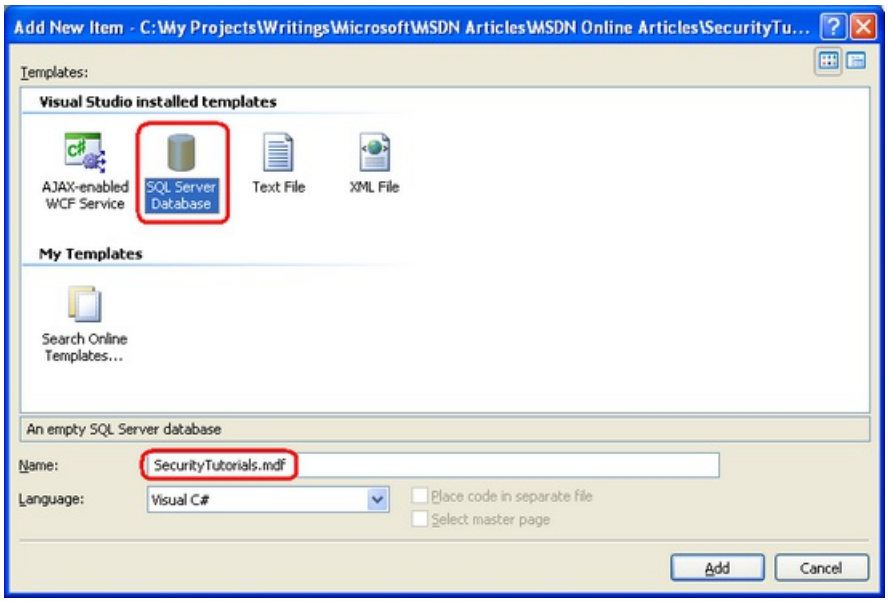


Figure 1: Add a New SQL Database Named `SecurityTutorials.mdf` Database to the `App_Data` Folder (Click to view full-size image)

Adding a database to the `App_Data` folder automatically includes it in the Database Explorer view. (In the non-

Express Edition version of Visual Studio, the Database Explorer is called the Server Explorer.) Go to the Database Explorer and expand the just-added `SecurityTutorials` database. If you do not see the Database Explorer on screen, go to the View menu and choose Database Explorer, or hit Ctrl+Alt+S. As Figure 2 shows, the `SecurityTutorials` database is empty - it contains no tables, no views, and no stored procedures.

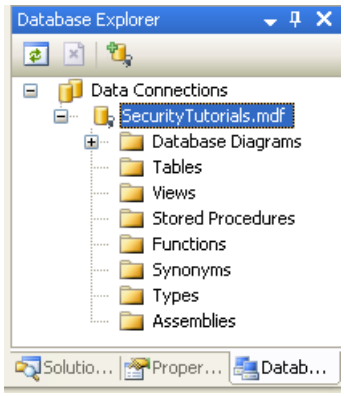


Figure 2: The `SecurityTutorials` Database is Currently Empty (Click to view full-size image)

Step 2: Adding the `SqlMembershipProvider` Schema to the Database

The `SqlMembershipProvider` requires a particular set of tables, views, and stored procedures to be installed in the user store database. These requisite database objects can be added using the `aspnet_regsql.exe` tool. This file is located in the `%WINDIR%\Microsoft.Net\Framework\v2.0.50727\` folder.

> **NOTE**
>
> The `aspnet_regsql.exe` tool offers both command line functionality and a graphical user interface. The graphical interface is more user friendly and is what we will examine in this tutorial. The command line interface is useful when the addition of the `SqlMembershipProvider` schema needs to be automated, such as in build scripts or automated testing scenarios.

The `aspnet_regsql.exe` tool is used to add or remove *ASP.NET application services* to a specified SQL Server database. The ASP.NET application services encompass the schemas for the `SqlMembershipProvider` and `SqlRoleProvider`, along with the schemas for the SQL-based providers for other ASP.NET 2.0 frameworks. We need to provide two bits of information to the `aspnet_regsql.exe` tool:

- Whether we want to add or remove application services, and
- The database from which to add or remove the application services schema

In prompting for the database to use, the `aspnet_regsql.exe` tool asks us to provide the name of the server the database resides on, the security credentials for connecting to the database, and the database name. If you are using the non-Express Edition of SQL Server, you should already know this information, as it is the same information you must provide through a connection string when working with the database through an ASP.NET web page. Determining the server and database name when using a SQL Server 2005 Express Edition database in the `App_Data` folder, however, is a bit more involved.

The following section examines a straightforward way for specifying the server and database name for a SQL Server 2005 Express Edition database in the `App_Data` folder. If you are not using SQL Server 2005 Express Edition feel free to skip ahead to the Installing the Application Services section.

Determining the Server and Database Name for a SQL Server 2005 Express Edition Database in the `App_Data` Folder

In order to use the `aspnet_regsql.exe` tool we need to know the server and database names. The server name is `localhost\InstanceName`. Most likely, the *InstanceName* is `SQLExpress`. However, if you installed SQL Server

2005 Express Edition manually (that is, you did not install it automatically while installing Visual Studio), then it is possible that you selected a different instance name.

The database name is a bit trickier to determine. Databases in the `App_Data` folder typically have a database name that includes a globally unique identifier along with the path to the database file. We need to determine this database name in order to add the application services schema through `aspnet_regsql.exe`.

The easiest way to ascertain the database name is to examine it through SQL Server Management Studio. SQL Server Management Studio provides a graphical interface for managing SQL Server 2005 databases, but it does not ship with the Express Edition of SQL Server 2005. The good news is that you can download the free Express Edition of SQL Server Management Studio.

> **NOTE**
>
> If you also have a non-Express Edition version of SQL Server 2005 installed on your desktop then the full version of Management Studio is likely installed. You can use the full version to determine the database name, following the same steps as outlined below for the Express Edition.

Start by closing Visual Studio to ensure that any locks imposed by Visual Studio on the database file are closed. Next, launch SQL Server Management Studio and connect to the `localhost\InstanceName` database for SQL Server 2005 Express Edition. As noted earlier, chances are the instance name is `SQLExpress`. For the Authentication option, select Windows Authentication.

Figure 3: Connect to the SQL Server 2005 Express Edition Instance (Click to view full-size image)

After connecting to the SQL Server 2005 Express Edition instance, Management Studio displays folders for the Databases, the Security settings, the Server Objects, and so on. If you expand the Databases tab you will see that the `SecurityTutorials.mdf` database is *not* registered in the database instance - we need to attach the database first.

Right-click on the Databases folder and choose Attach from the context menu. This will display the Attach Databases dialog box. From here, click the Add button, browse to the `SecurityTutorials.mdf` database, and click OK. Figure 4 shows the Attach Databases dialog box after the `SecurityTutorials.mdf` database has been selected. Figure 5 shows Management Studio's Object Explorer after the database has been successfully attached.

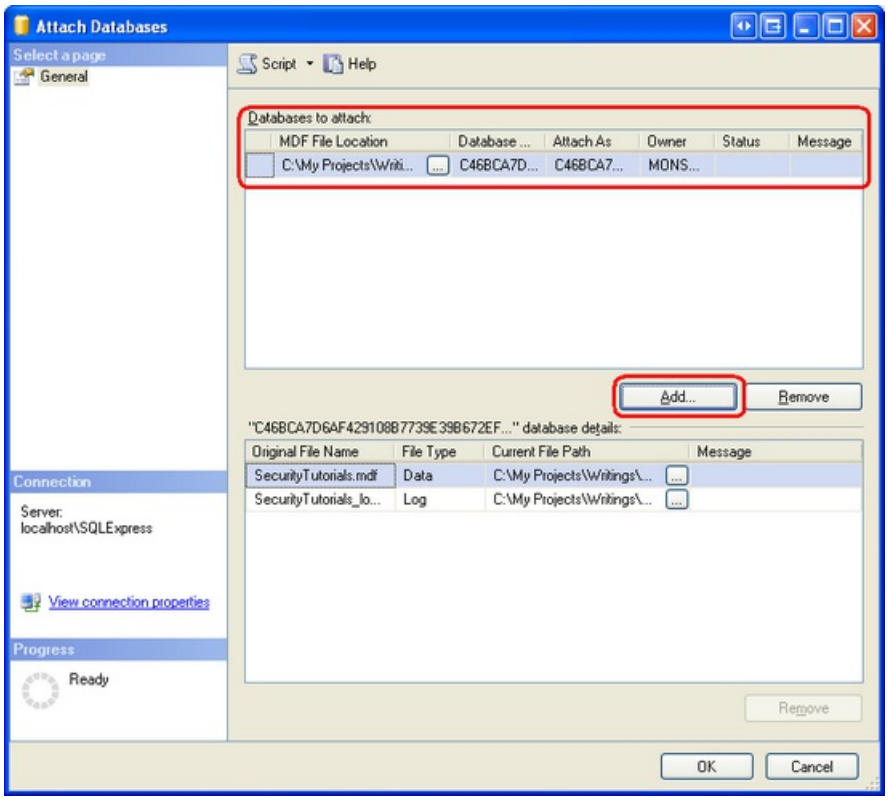


Figure 4: Attach the `SecurityTutorials.mdf` Database (Click to view full-size image)

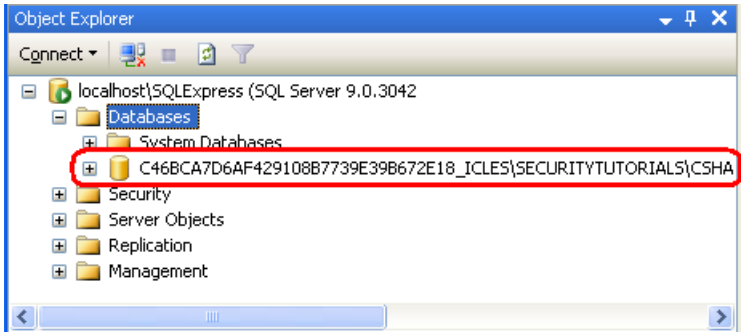


Figure 5: The `SecurityTutorials.mdf` Database Appears in the Databases Folder (Click to view full-size image)

As Figure 5 shows, the `SecurityTutorials.mdf` database has a rather abstruse name. Let's change it to a more memorable (and easier to type) name. Right-click on the database, choose Rename from the context menu, and rename it `SecurityTutorialsDatabase`. This does not change the filename, just the name the database uses to identify itself to SQL Server.

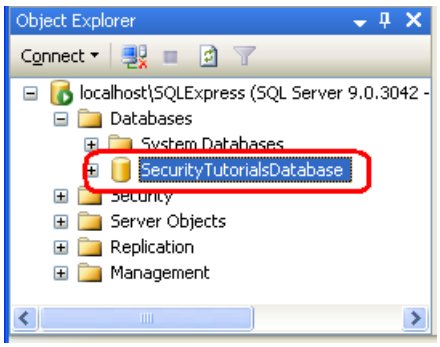


Figure 6: Rename the Database to `SecurityTutorialsDatabase` (Click to view full-size image)

At this point we know the server and database names for the `SecurityTutorials.mdf` database file: `localhost\InstanceName` and `SecurityTutorialsDatabase`, respectively. We are now ready to install the application services through the `aspnet_regsql.exe` tool.

Installing the Application Services

To launch the `aspnet_regsql.exe` tool, go to the start menu and choose Run. Enter `%WINDIR%\Microsoft.Net\Framework\v2.0.50727\aspnet_regsql.exe` into the textbox and click OK. Alternatively, you can use Windows Explorer to drill down to the appropriate folder and double-click the `aspnet_regsql.exe` file. Either approach will net the same results.

Running the `aspnet_regsql.exe` tool without any command line arguments launches the ASP.NET SQL Server Setup Wizard graphical user interface. The wizard makes it easy to add or remove the ASP.NET application services on a specified database. The first screen of the wizard, shown in Figure 7, describes the purpose of the tool.

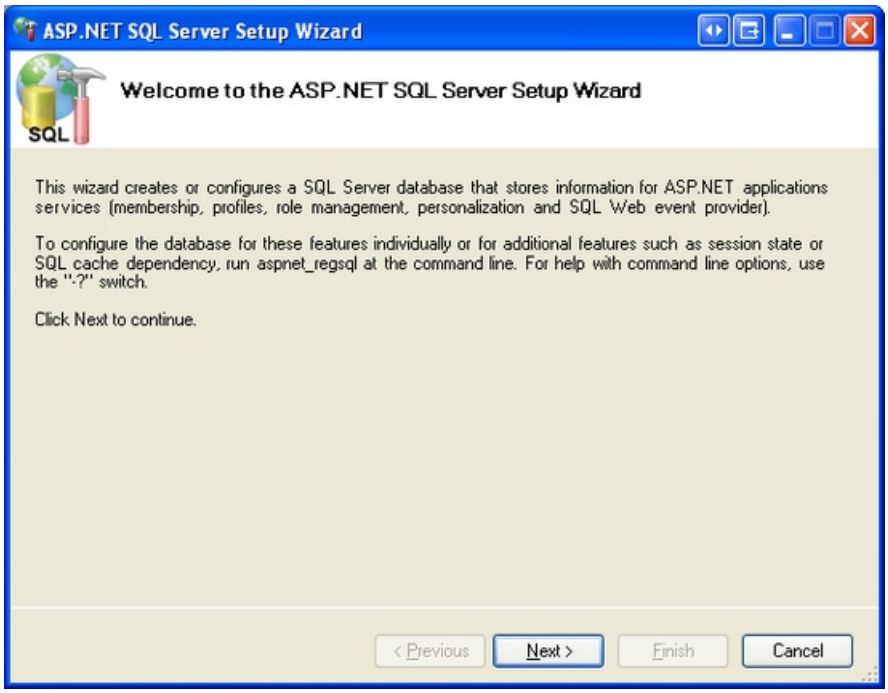


Figure 7: Use the ASP.NET SQL Server Setup Wizard Makes to Add the Membership Schema (Click to view full-size image)

The second step in the wizard asks us whether we want to add the application services or remove them. Since we want to add the tables, views, and stored procedures necessary for the `SqlMembershipProvider`, choose the Configure SQL Server for application services option. Later, if you want to remove this schema from your database, re-run this wizard, but instead choose the Remove application services information from an existing database option.

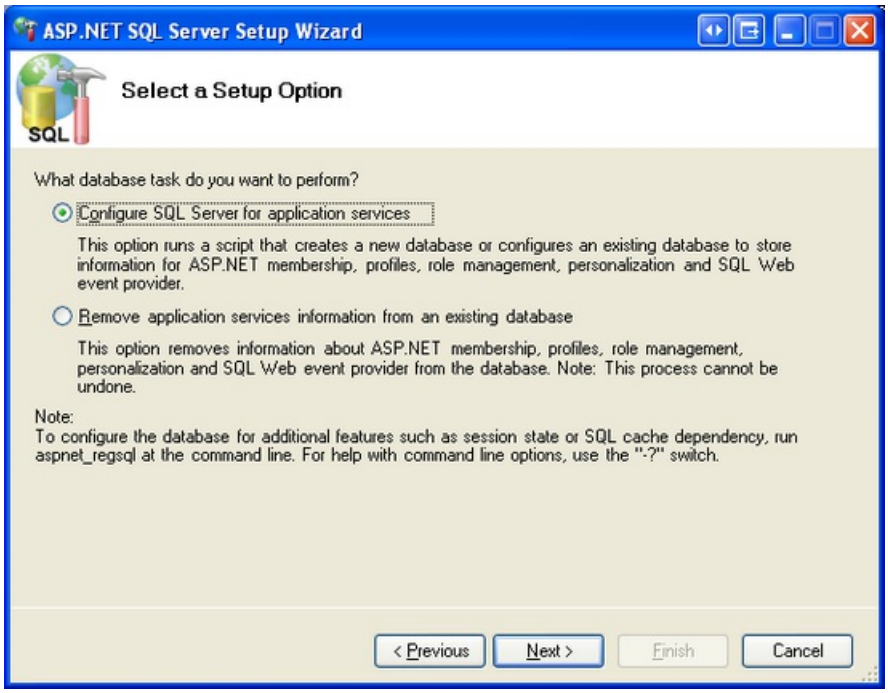


Figure 8: Choose the Configure SQL Server for Application Services Option (Click to view full-size image)

The third step prompts for the database information: the server name, authentication information, and the database name. If you have been following along with this tutorial and have added the `SecurityTutorials.mdf` database to `App_Data`, attached it to `localhost\InstanceName`, and renamed it to `SecurityTutorialsDatabase`, then use the following values:

- Server: `localhost\InstanceName`
- Windows authentication
- Database: `SecurityTutorialsDatabase`

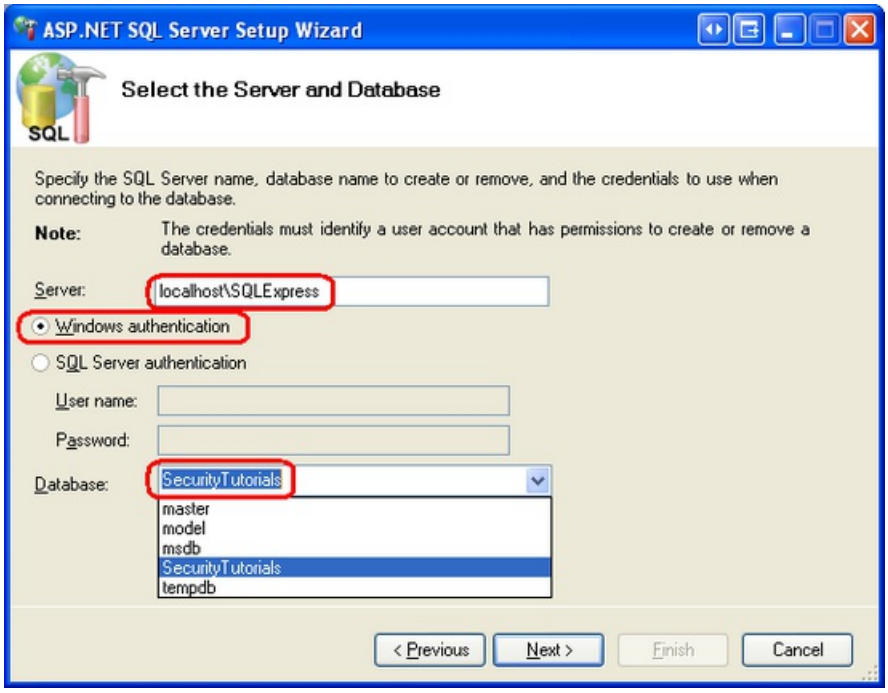


Figure 9: Enter the Database Information (Click to view full-size image)

After entering the database information, click Next. The final step summarizes the steps that will be taken. Click Next to install the application services and then Finish to complete the wizard.

> **NOTE**
>
> If you used Management Studio to attach the database and rename the database file, be sure to detach the database and close Management Studio before reopening Visual Studio. To detach the `SecurityTutorialsDatabase` database, right-click on the database name and, from the Tasks menu, choose Detach.

Upon completion of the wizard, return to Visual Studio and navigate to the Database Explorer. Expand the Tables folder. You should see a series of tables whose names start with the prefix `aspnet_`. Likewise, a variety of views and stored procedures can be found under the Views and Stored Procedures folders. These database objects make up the application services schema. We will examine the membership- and role-specific database objects in Step 3.

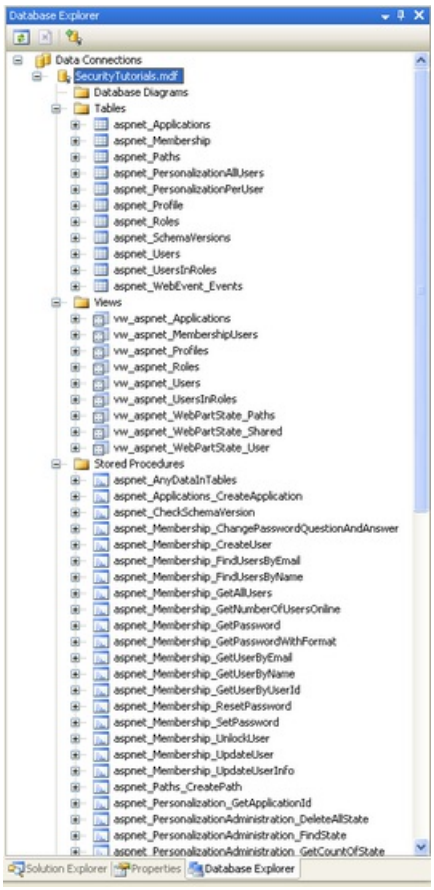


Figure 10: A Variety of Tables, Views, and Stored Procedures Have Been Added to the Database (Click to view full-size image)

> **NOTE**
>
> The `aspnet_regsql.exe` tool's graphical user interface installs the entire application services schema. But when executing `aspnet_regsql.exe` from the command-line you can specify what particular application services components to install (or remove). Therefore, if you want to add just the tables, views, and stored procedures necessary for the `SqlMembershipProvider` and `SqlRoleProvider` providers, run `aspnet_regsql.exe` from the command-line. Alternatively, you can manually run the appropriate subset of T-SQL create scripts used by `aspnet_regsql.exe`. These scripts are located in the `WINDIR%\Microsoft.Net\Framework\v2.0.50727\` folder with names like `InstallCommon.sql`, `InstallMembership.sql`, `InstallRoles.sql`, `InstallProfile.sql`, `InstallSqlState.sql`, and so on.

At this point we have created the database objects needed by the `SqlMembershipProvider`. However, we still need to instruct the Membership framework that it should use the `SqlMembershipProvider` (versus, say, the `ActiveDirectoryMembershipProvider`) and that the `SqlMembershipProvider` should use the `SecurityTutorials` database. We'll look at how to specify what provider to use and how to customize the selected provider's

settings in Step 4. But first, let's take a deeper look at the database objects that were just created.

Step 3: A Look at the Schema's Core Tables

When working with the Membership and Roles frameworks in an ASP.NET application, the implementation details are encapsulated by the provider. In future tutorials we will interface with these frameworks via the .NET Framework's `Membership` and `Roles` classes. When using these high-level APIs we do not need to concern ourselves with the low-level details, like what queries are executed or what tables are modified by the `SqlMembershipProvider` and `SqlRoleProvider`.

Given this, we could confidently use the Membership and Roles frameworks without having explored the database schema created in Step 2. However, when creating the tables to store application data we may need to create entities that relate to users or roles. It helps to have a familiarity with the `SqlMembershipProvider` and `SqlRoleProvider` schemas when establishing foreign key constraints between the application data tables and those tables created in Step 2. Moreover, in certain rare circumstances we may need to interface with the user and role stores directly at the database level (instead of through the `Membership` or `Roles` classes).

Partitioning the User Store Into Applications

The Membership and Roles frameworks are designed such that a single user and role store can be shared among many different applications. An ASP.NET application that uses the Membership or Roles frameworks must specify what application partition to use. In short, multiple web applications can use the same user and role stores. Figure 11 depicts user and role stores that are partitioned into three applications: HRSite, CustomerSite, and SalesSite. These three web applications each have their own unique users and roles, yet they all physically store their user account and role information in the same database tables.

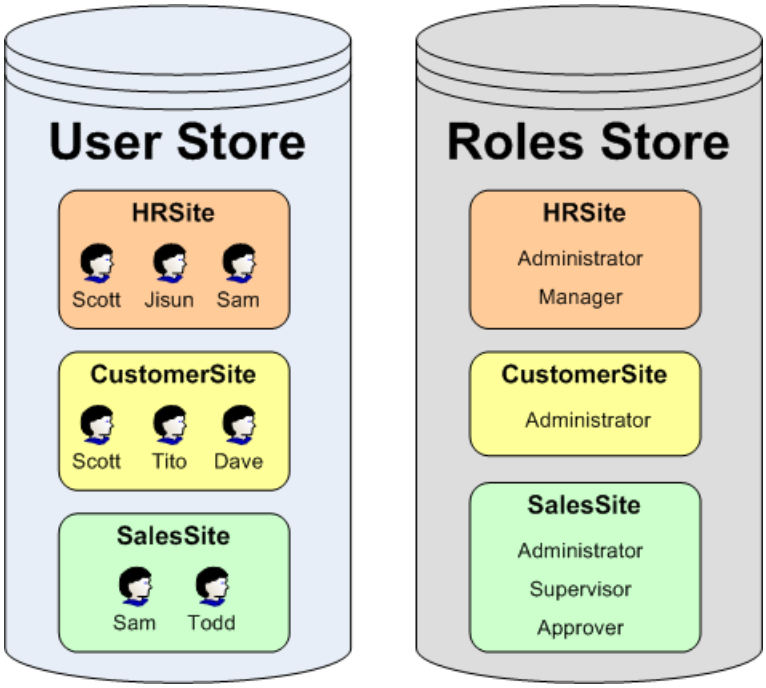


Figure 11: User Accounts May Be Partitioned Across Multiple Applications (Click to view full-size image)

The `aspnet_Applications` table is what defines these partitions. Each application that uses the database to store user account information is represented by a row in this table. The `aspnet_Applications` table has four columns: `ApplicationId`, `ApplicationName`, `LoweredApplicationName`, and `Description`. `ApplicationId` is of type `uniqueidentifier` and is the table's primary key; `ApplicationName` provides a unique human-friendly name for each application.

The other Membership- and Role-related tables link back to the `ApplicationId` field in `aspnet_Applications`. For example, the `aspnet_Users` table, which contains a record for each user account, has an `ApplicationId` foreign key field; ditto for the `aspnet_Roles` table. The `ApplicationId` field in these tables specifies the application

partition the user account or role belongs to.

Storing User Account Information

User account information is housed in two tables: `aspnet_Users` and `aspnet_Membership`. The `aspnet_Users` table contains fields that hold the essential user account information. The three most pertinent columns are:

- `UserId`
- `UserName`
- `ApplicationId`

`UserId` is the primary key (and of type `uniqueidentifier`). `UserName` is of type `nvarchar(256)` and, along with the password, makes up the user's credentials. (A user's password is stored in the `aspnet_Membership` table.) `ApplicationId` links the user account to a particular application in `aspnet_Applications`. There is a composite `UNIQUE` constraint on the `UserName` and `ApplicationId` columns. This ensures that in a given application each UserName is unique, yet it allows for the same `UserName` to be used in different applications.

The `aspnet_Membership` table includes additional user account information, like the user's password, email address, the last login date and time, and so forth. There is a one-to-one correspondence between records in the `aspnet_Users` and `aspnet_Membership` tables. This relationship is ensured by the `UserId` field in `aspnet_Membership`, which serves as the table's primary key. Like the `aspnet_Users` table, `aspnet_Membership` includes an `ApplicationId` field that ties this information to a particular application partition.

Securing Passwords

Password information is stored in the `aspnet_Membership` table. The `SqlMembershipProvider` allows for passwords to be stored in the database using one of the following three techniques:

- **Clear** - the password is stored in the database as plain-text. I strongly discourage using this option. If the database is compromised - be it by a hacker who finds a back door or a disgruntled employee who has database access - every single user's credentials are there for the taking.
- **Hashed** - passwords are hashed using a one-way hash algorithm and a randomly generated salt value. This hashed value (along with the salt) is stored in the database.
- **Encrypted** - an encrypted version of the password is stored in the database.

The password storage technique used depends on the `SqlMembershipProvider` settings specified in `Web.config`. We will look at customizing the `SqlMembershipProvider` settings in Step 4. The default behavior is to store the hash of the password.

The columns responsible for storing the password are `Password`, `PasswordFormat`, and `PasswordSalt`. `PasswordFormat` is a field of type `int` whose value indicates the technique used for storing the password: 0 for Clear; 1 for Hashed; 2 for Encrypted. `PasswordSalt` is assigned a randomly generated string regardless of the password storage technique used; the value of `PasswordSalt` is only used when computing the hash of the password. Finally, the `Password` column contains the actual password data, be it the plain-text password, the hash of the password, or the encrypted password.

Table 1 illustrates what these three columns might look like for the various storage techniques when storing the password MySecret! .

STORAGE TECHNIQUE<_O3A_P />	PASSWORD<_O3A_P />	PASSWORDFORMAT<_O3A_P />	PASSWORDSALT<_O3A_P />
Clear	MySecret!	0	tTnkPlesqissc2y2SMEygA==
Hashed	2oXm6sZHWbTHFgjgkGQsc2Ec9ZM=	1	wFgjUfhdUFOCKQiI61vtiQ==

STORAGE TECHNIQUE<_O3A_P />	PASSWORD<_O3A_P />	PASSWORDFORMAT<_O3A_P />	PASSWORDSALT<_O3A_P />
Encrypted	62RZgDvhxykkqsMchZ0Yly7HS6onhpaoCYaRxV8g0F4CW56OXUU3e7Inza9j9BKp	2	LSRzhGS/aa/oqAXGLHJNBw==

Table 1: Example Values for the Password-Related Fields When Storing the Password MySecret!

> **NOTE**
> The particular encryption or hashing algorithm used by the `SqlMembershipProvider` is determined by the settings in the `<machineKey>` element. We discussed this configuration element in Step 3 of the *Forms Authentication Configuration and Advanced Topics* tutorial.

Storing Roles and Role Associations

The Roles framework allows developers to define a set of roles and specify what users belong to what roles. This information is captured in the database through two tables: `aspnet_Roles` and `aspnet_UsersInRoles`. Each record in the `aspnet_Roles` table represents a role for a particular application. Much like the `aspnet_Users` table, the `aspnet_Roles` table has three columns pertinent to our discussion:

- `RoleId`
- `RoleName`
- `ApplicationId`

`RoleId` is the primary key (and of type `uniqueidentifier`). `RoleName` is of type `nvarchar(256)`. And `ApplicationId` links the user account to a particular application in `aspnet_Applications`. There is a composite `UNIQUE` constraint on the `RoleName` and `ApplicationId` columns, ensuring that in a given application each role name is unique.

The `aspnet_UsersInRoles` table serves as a mapping between users and roles. There are only two columns - `UserId` and `RoleId` - and together they make up a composite primary key.

Step 4: Specifying the Provider and Customizing Its Settings

All of the frameworks that support the provider model - such as the Membership and Roles frameworks - lack implementation details themselves and instead delegate that responsibility to a provider class. In the case of the Membership framework, the `Membership` class defines the API for managing user accounts, but it does not interact directly with any user store. Rather, the `Membership` class's methods hand off the request to the configured provider - we will be using the `SqlMembershipProvider`. When we invoke one of the methods in the `Membership` class, how does the Membership framework know to delegate the call to the `SqlMembershipProvider`?

The `Membership` class has a `Providers` property that contains a reference to all of the registered provider classes available for use by the Membership framework. Each registered provider has an associated name and type. The name offers a human-friendly way to reference a particular provider in the `Providers` collection, while the type identifies the provider class. Moreover, each registered provider may include configuration settings. Configuration settings for the Membership framework include `passwordFormat` and `requiresUniqueEmail`, among many others. See Table 2 for a complete list of configuration settings used by the `SqlMembershipProvider`.

The `Providers` property's contents are specified through the web application's configuration settings. By default, all web applications have a provider named `AspNetSqlMembershipProvider` of type `SqlMembershipProvider`. This default Membership provider is registered in `machine.config` (located at

`%WINDIR%\Microsoft.Net\Framework\v2.0.50727\CONFIG)`:

```
<membership>
 <providers>
 <add name="AspNetSqlMembershipProvider"
 type="System.Web.Security.SqlMembershipProvider, System.Web, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a"
 connectionStringName="LocalSqlServer"
 enablePasswordRetrieval="false"
 enablePasswordReset="true"
 requiresQuestionAndAnswer="true"
 applicationName="/"
 requiresUniqueEmail="false"
 passwordFormat="Hashed"
 maxInvalidPasswordAttempts="5"
 minRequiredPasswordLength="7"
 minRequiredNonalphanumericCharacters="1"
 passwordAttemptWindow="10"
 passwordStrengthRegularExpression=""/>
 </providers>
</membership>
```

As the markup above shows, the `<membership>` element defines the configuration settings for the Membership framework while the `<providers>` child element specifies the registered providers. Providers may be added or removed using the `<add>` or `<remove>` elements; use the `<clear>` element to remove all currently registered providers. As the markup above shows, `machine.config` adds a provider named `AspNetSqlMembershipProvider` of type `SqlMembershipProvider`.

In addition to the `name` and `type` attributes, the `<add>` element contains attributes that define the values for various configuring settings. Table 2 lists the available `SqlMembershipProvider`-specific configuration settings, along with a description of each.

NOTE

Any default values noted in Table 2 refer to the default values defined in the `SqlMembershipProvider` class. Note that not all of the configuration settings in `AspNetSqlMembershipProvider` correspond to the default values of the `SqlMembershipProvider` class. For example, if not specified in a Membership provider, the `requiresUniqueEmail` setting defaults to true. However, the `AspNetSqlMembershipProvider` overrides this default value by explicitly specifying a value of `false`.

SETTING<_O3A_P />	DESCRIPTION<_O3A_P />
`ApplicationName`	Recall that the Membership framework allows for a single user store to be partitioned across multiple applications. This setting indicates the name of the application partition used by the Membership provider. If this value is not explicitly specified, it is set, at runtime, to the value of the application's virtual root path.
`commandTimeout`	Specifies the SQL command timeout value (in seconds). The default value is 30.
`connectionStringName`	The name of the connection string in the `<connectionStrings>` element to use to connect to the user store database. This value is required.

SETTING<_O3A_P />	DESCRIPTION<_O3A_P />
`description`	Provides a human-friendly description of the registered provider.
`enablePasswordRetrieval`	Specifies whether users may retrieve their forgotten password. The default value is `false`.
`enablePasswordReset`	Indicates whether users are allowed to reset their password. Defaults to `true`.
`maxInvalidPasswordAttempts`	The maximum number of unsuccessful login attempts that may occur for a given user during the specified `passwordAttemptWindow` before the user is locked out. The default value is 5.
`minRequiredNonalphanumericCharacters`	The minimum number of non-alphanumeric characters that must appear in a user's password. This value must be between 0 and 128; the default is 1.
`minRequiredPasswordLength`	The minimum number of characters required in a password. This value must be between 0 and 128; the default is 7.
`name`	The name of the registered provider. This value is required.
`passwordAttemptWindow`	The number of minutes during which failed login attempts are tracked. If a user supplies invalid login credentials `maxInvalidPasswordAttempts` times within this specified window, they are locked out. The default value is 10.
`PasswordFormat`	The password storage format: `Clear`, `Hashed`, or `Encrypted`. The default is `Hashed`.
`passwordStrengthRegularExpression`	If provided, this regular expression is used to evaluate the strength of the user's selected password when creating a new account or when changing their password. The default value is an empty string.
`requiresQuestionAndAnswer`	Specifies whether a user must answer his security question when retrieving or resetting his password. The default value is `true`.
`requiresUniqueEmail`	Indicates whether all user accounts in a given application partition must have a unique email address. The default value is `true`.
`type`	Specifies the type of the provider. This value is required.

Table 2: Membership and `SqlMembershipProvider` Configuration Settings

In addition to `AspNetSqlMembershipProvider`, other Membership providers may be registered on an application-by-application basis by adding similar markup to the `Web.config` file.

> **NOTE**
> The Roles framework works in much the same way: there is a default registered role provider in `machine.config` and the registered providers may be customized on an application-by-application basis in `Web.config`. We will examine the Roles framework and its configuration markup in detail in a future tutorial.

Customizing the `SqlMembershipProvider` Settings

The default `SqlMembershipProvider` (`AspNetSqlMembershipProvider`) has its `connectionStringName` attribute set to `LocalSqlServer`. Like the `AspNetSqlMembershipProvider` provider, the connection string name `LocalSqlServer` is defined in `machine.config`.

```
<connectionStrings>
 <add name="LocalSqlServer"
 connectionString="data source=.\SQLEXPRESS;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User Instance=true"
 providerName="System.Data.SqlClient"/>
</connectionStrings>
```

As you can see, this connection string defines a SQL 2005 Express Edition database located at |DataDirectory|aspnetdb.mdf'. The string |DataDirectory| is translated at runtime to point to the `~/App_Data/` directory, so the database path |DataDirectory|aspnetdb.mdf" translates to `~/App_Data` / `aspnet.mdf`.

If we did not specify any Membership provider information in our application's `Web.config` file, the application uses the default registered Membership provider, `AspNetSqlMembershipProvider`. If the `~/App_Data/aspnet.mdf` database does not exist, the ASP.NET runtime will automatically create it and add the application services schema. However, we don't want to use the `aspnet.mdf` database; rather, we want to use the `SecurityTutorials.mdf` database we created in Step 2. This modification can be accomplished in one of two ways:

- **Specify a value for the `LocalSqlServer` connection string name in `Web.config`.** By overwriting the `LocalSqlServer` connection string name value in `Web.config`, we can use the default registered Membership provider (`AspNetSqlMembershipProvider`) and have it correctly work with the `SecurityTutorials.mdf` database. This approach is fine if you are content with the configuration settings specified by `AspNetSqlMembershipProvider`. For more information on this technique, see Scott Guthrie's blog post, Configuring ASP.NET 2.0 Application Services to Use SQL Server 2000 or SQL Server 2005.
- **Add a new registered provider of type `SqlMembershipProvider` and configure its `connectionStringName` setting to point to the `SecurityTutorials.mdf` database.** This approach is useful in scenarios where you want to customize other configuration properties in addition to the database connection string. In my own projects I always use this approach because of its flexibility and readability.

Before we can add a new registered provider that references the `SecurityTutorials.mdf` database, we first need to add an appropriate connection string value in the `<connectionStrings>` section in `Web.config`. The following markup adds a new connection string named `SecurityTutorialsConnectionString` that references the SQL Server 2005 Express Edition `SecurityTutorials.mdf` database in the `App_Data` folder.

```
<configuration>
 <connectionStrings>
 <add name="SecurityTutorialsConnectionString"
 connectionString="Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\SecurityTutorials.mdf;Integrated Security=True;User
Instance=True"
 providerName="System.Data.SqlClient"/>
 </connectionStrings>
 <system.web>
 ... Configuration markup  removed for brevity ...
 </system.web>
</configuration>
```

> **NOTE**
> If you are using an alternate database file, update the connection string as needed. For more information on forming the correct connection string, refer to ConnectionStrings.com.

Next, add the following Membership configuration markup to the `Web.config` file. This markup registers a new provider named `SecurityTutorialsSqlMembershipProvider`.

```
<configuration>
 <connectionStrings>
 <add name="SecurityTutorialsConnectionString"
 connectionString="Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\SecurityTutorials.mdf;Integrated Security=True;User
Instance=True"
 providerName="System.Data.SqlClient"/>
 </connectionStrings>
 <system.web>
 <membership defaultProvider="SecurityTutorialsSqlMembershipProvider">
 <providers>
 <!--Add a customized SqlMembershipProvider -->
 <add name="SecurityTutorialsSqlMembershipProvider"
 type="System.Web.Security.SqlMembershipProvider"
 connectionStringName="SecurityTutorialsConnectionString"
 enablePasswordRetrieval="false"
 enablePasswordReset="true"
 requiresQuestionAndAnswer="true"
 applicationName="SecurityTutorials"
 requiresUniqueEmail="true"
 passwordFormat="Hashed"
 maxInvalidPasswordAttempts="5"
 minRequiredPasswordLength="7"
 minRequiredNonalphanumericCharacters="1"
 passwordAttemptWindow="10"
 passwordStrengthRegularExpression=""/>
 </providers>
 </membership>
 ... Configuration markup removed for brevity ...
 </system.web>
</configuration>
```

In addition to registering the `SecurityTutorialsSqlMembershipProvider` provider, the above markup defines the `SecurityTutorialsSqlMembershipProvider` as the default provider (via the `defaultProvider` attribute in the `<membership>` element). Recall that the Membership framework can have multiple registered providers. Since `AspNetSqlMembershipProvider` is registered as the first provider in `machine.config`, it serves as the default provider unless we indicate otherwise.

Currently, our application has two registered providers: `AspNetSqlMembershipProvider` and

`SecurityTutorialsSqlMembershipProvider`. However, before registering the `SecurityTutorialsSqlMembershipProvider` provider we could have cleared out all previously registered providers by adding a `<clear />` element immediately before our `<add>` element. This would clear out the `AspNetSqlMembershipProvider` from the list of registered providers, meaning that the `SecurityTutorialsSqlMembershipProvider` would be the only registered Membership provider. If we used this approach, then we would not need to mark the `SecurityTutorialsSqlMembershipProvider` as the default provider, since it would be the only registered Membership provider. For more information on using `<clear />`, see Using `<clear />` When Adding Providers.

Note that the `SecurityTutorialsSqlMembershipProvider`'s `connectionStringName` setting references the just-added `SecurityTutorialsConnectionString` connection string name, and that its `applicationName` setting has been set to a value of SecurityTutorials. Additionally, the `requiresUniqueEmail` setting has been set to `true`. All other configuration options are identical to the values in `AspNetSqlMembershipProvider`. Feel free to make any configuration modifications here, if you wish. For example, you could tighten the password strength by requiring two non-alphanumeric characters instead of one, or by increasing the password length to eight characters instead of seven.

> **NOTE**
>
> Recall that the Membership framework allows for a single user store to be partitioned across multiple applications. The Membership provider's `applicationName` setting indicates what application the provider uses when working with the user store. It is important that you explicitly set a value for the `applicationName` configuration setting because if the `applicationName` is not explicitly set, it is assigned to the web application's virtual root path at runtime. This works fine as long as the application's virtual root path doesn't change, but if you move the application to a different path, the `applicationName` setting will change too. When this happens, the Membership provider will start working with a different application partition than was previously used. User accounts created prior to the move will reside in a different application partition and those users will no longer be able to log into the site. For a more in-depth discussion on this matter, see Always Set the `applicationName` Property When Configuring ASP.NET 2.0 Membership and Other Providers.

Summary

At this point we have a database with the configured application services (`SecurityTutorials.mdf`) and have configured our web application so that the Membership framework uses the `SecurityTutorialsSqlMembershipProvider` provider we just registered. This registered provider is of type `SqlMembershipProvider` and has its `connectionStringName` set to the appropriate connection string (`SecurityTutorialsConnectionString`) and its `applicationName` value explicitly set.

We are now ready to use the Membership framework from our application. In the next tutorial we will examine how to create new user accounts. Following that we will explore authenticating users, performing user-based authorization, and storing additional user-related information in the database.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Always Set the `applicationName` Property When Configuring ASP.NET 2.0 Membership and Other Providers
- Configuring ASP.NET 2.0 Application Services to Use SQL Server 2000 or SQL Server 2005
- Download SQL Server Management Studio Express Edition
- Examining ASP.NET 2.0's Membership, Roles, and Profile
- The `<add>` Element for Providers for Membership
- The `<membership>` Element
- The `<providers>` Element for Membership

- Using `<clear />` When Adding Providers
- Working Directly with the `SqlMembershipProvider`

Video Training on Topics Contained in this Tutorial

- Understanding ASP.NET Memberships
- Configuring SQL To Work with Membership Schemas
- Changing Membership Settings in the Default Membership Schema

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Alicja Maziarz. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

NEXT

Creating User Accounts (C#)

2/20/2020 • 31 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

In this tutorial we will explore using the Membership framework (via the SqlMembershipProvider) to create new user accounts. We will see how to create new users programmatically and through ASP.NET's built-in CreateUserWizard control.

Introduction

In the preceding tutorial we installed the application services schema in a database, which added the tables, views, and stored procedures needed by the `SqlMembershipProvider` and `SqlRoleProvider`. This created the infrastructure we will need for the remainder of the tutorials in this series. In this tutorial we will explore using the Membership framework (via the `SqlMembershipProvider`) to create new user accounts. We will see how to create new users programmatically and through ASP.NET's built-in CreateUserWizard control.

In addition to learning how to create new user accounts, we will also need to update the demo website we first created in the *An Overview of Forms Authentication* tutorial and then enhanced in the *Forms Authentication Configuration and Advanced Topics* tutorial. Our demo web application has a login page that validates users' credentials against hard-coded username/password pairs. Moreover, `Global.asax` includes code that creates custom `IPrincipal` and `IIdentity` objects for authenticated users. We will update the login page to validate users' credentials against the Membership framework and remove the custom principal and identity logic.

Let's get started!

The Forms Authentication and Membership Checklist

Before we begin working with the Membership framework, let's take a moment to review the important steps we have taken to reach this point. When using the Membership framework with the `SqlMembershipProvider` in a forms-based authentication scenario, the following steps need to be performed prior to implementing Membership functionality in your web application:

1. **Enable forms-based authentication.** As we discussed in *An Overview of Forms Authentication*, forms authentication is enabled by editing `Web.config` and setting the `<authentication>` element's `mode` attribute to `Forms`. With forms authentication enabled, each incoming request is examined for a *forms authentication ticket*, which, if present, identifies the requestor.
2. **Add the application services schema to the appropriate database.** When using the `SqlMembershipProvider` we need to install the application services schema to a database. Usually this schema is added to the same database that holds the application's data model. The *Creating the Membership Schema in SQL Server* tutorial looked at using the `aspnet_regsql.exe` tool to accomplish this.
3. **Customize the Web Application's Settings to reference the database from step 2.** The *Creating the Membership Schema in SQL Server* tutorial showed two ways to configure the web application so that the `SqlMembershipProvider` would use the database selected in step 2: by modifying the `LocalSqlServer` connection string name; or by adding a new registered provider to the list of Membership framework providers and customizing that new provider to use the database from step 2.

When building a web application that uses the `SqlMembershipProvider` and forms-based authentication, you will

need to perform these three steps before using the `Membership` class or the ASP.NET Login Web controls. Since we already performed these steps in previous tutorials, we are ready to start using the Membership framework!

Step 1: Adding New ASP.NET Pages

In this tutorial and the next three we will be examining various Membership-related functions and capabilities. We will need a series of ASP.NET pages to implement the topics examined throughout these tutorials. Let's create those pages and then a site map file `(Web.sitemap)`.

Start by creating a new folder in the project named `Membership`. Next, add five new ASP.NET pages to the `Membership` folder, linking each page with the `Site.master` master page. Name the pages:

- `CreatingUserAccounts.aspx`
- `UserBasedAuthorization.aspx`
- `EnhancedCreateUserWizard.aspx`
- `AdditionalUserInfo.aspx`
- `Guestbook.aspx`

At this point your project's Solution Explorer should look similar to the screen shot shown in Figure 1.

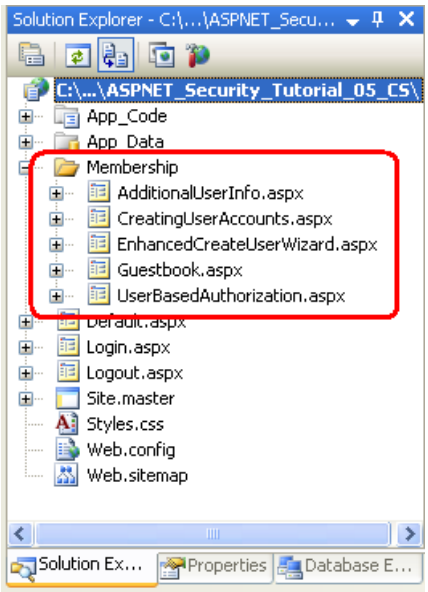


Figure 1: Five New Pages Have Been Added to the `Membership` Folder (Click to view full-size image)

Each page should, at this point, have the two Content controls, one for each of the master page's ContentPlaceHolders: `MainContent` and `LoginContent`.

```
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent"

Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="LoginContent"
Runat="Server">
</asp:Content>
```

Recall that the `LoginContent` ContentPlaceHolder's default markup displays a link to log on or log off the site, depending on whether the user is authenticated. The presence of the `Content2` Content control, however, overrides the master page's default markup. As we discussed in *An Overview of Forms Authentication* tutorial, this is useful in pages where we do not want to display login-related options in the left column.

For these five pages, however, we want to show the master page's default markup for the `LoginContent`

ContentPlaceHolder. Therefore, remove the declarative markup for the `Content2` Content control. After doing so, each of the five page's markup should contain just one Content control.

Step 2: Creating the Site Map

All but the most trivial websites need to implement some form of a navigational user interface. The navigation user interface may be a simple list of links to the various sections of the site. Alternatively, these links may be arranged into menus or tree views. As page developers, creating the navigational user interface is only half of the story. We also need some means to define the site's logical structure in a maintainable and updateable fashion. As new pages are added or existing pages removed, we want to be able to update a single source – the site map – and have those modifications reflected across the site's navigational user interface.

These two tasks – defining the site map and implementing a navigational user interface based on the site map – are easy to accomplish thanks to the Site Map framework and the Navigation Web controls added in ASP.NET version 2.0. The Site Map framework allows for a developer to define a site map and then to access it through a programmatic API (the `SiteMap` class). The built-in Navigation Web controls include a Menu control, the TreeView control, and the SiteMapPath control.

Like the Membership and Roles frameworks, the Site Map framework is built atop the provider model. The job of the Site Map provider class is to generate the in-memory structure used by the `SiteMap` class from a persistent data store, such as an XML file or a database table. The .NET Framework ships with a default Site Map provider that reads the site map data from an XML file (`XmlSiteMapProvider`), and this is the provider we will be using in this tutorial. For some alternate Site Map provider implementations, refer to the Further Readings section at the end of this tutorial.

The default Site Map provider expects a properly-formatted XML file named `Web.sitemap` to exist the root directory. Since we are using this default provider, we need to add such a file and define the site map's structure in the appropriate XML format. To add the file, right-click on the project name in Solution Explorer and choose Add New Item. From the dialog box, opt to add a file of type Site Map named `Web.sitemap`.

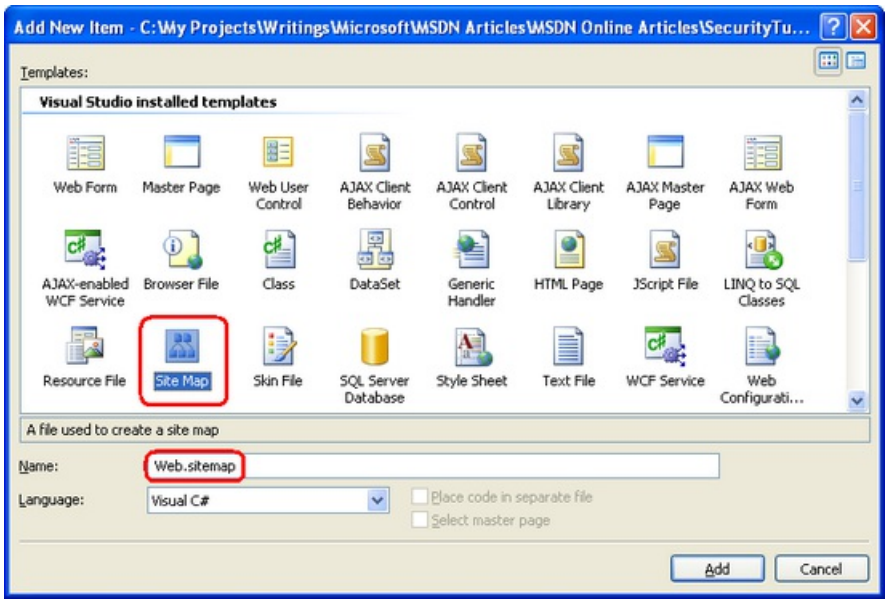


Figure 2: Add a File Named `Web.sitemap` to the Project's Root Directory (Click to view full-size image)

The XML site map file defines the website's structure as a hierarchy. This hierarchical relationship is modeled in the XML file via the ancestry of the `<siteMapNode>` elements. The `Web.sitemap` must start with a `<siteMap>` parent node that has precisely one `<siteMapNode>` child. This top-level `<siteMapNode>` element represents the root of the hierarchy, and may have an arbitrary number of descendent nodes. Each `<siteMapNode>` element must include a `title` attribute and may optionally include `url` and `description` attributes, among others; each non-empty `url` attribute must be unique.

Enter the following XML into the `Web.sitemap` file:

```
<?xml version="1.0" encoding="utf-8" ?>

<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
     <siteMapNode url="~/Default.aspx" title="Home">
          <siteMapNode title="Membership">
               <siteMapNode url="~/Membership/CreatingUserAccounts.aspx" title="Creating User Accounts" />

               <siteMapNode url="~/Membership/UserBasedAuthorization.aspx" title="User-Based Authorization"
/>
               <siteMapNode url="~/Membership/Guestbook.aspx" title="Storing Additional User Information" />
          </siteMapNode>

     </siteMapNode>
</siteMap>
```

The above site map markup defines the hierarchy shown in Figure 3.

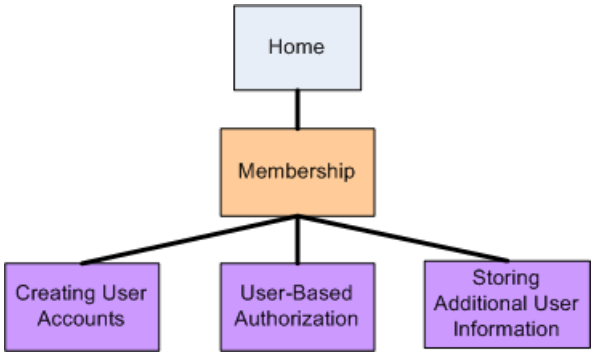


Figure 3: The Site Map Represents a Hierarchical Navigational Structure (Click to view full-size image)

Step 3: Updating the Master Page to Include a Navigational User Interface

ASP.NET includes a number of navigation-related Web controls for designing a user interface. These include the Menu, TreeView, and the SiteMapPath controls. The Menu and TreeView controls render the site map structure in a menu or a tree, respectively, whereas the SiteMapPath displays a breadcrumb that shows the current node being visited as well as its ancestors. The site map data can be bound to other data Web controls using the SiteMapDataSource and can be accessed programmatically via the `SiteMap` class.

Since a thorough discussion of the Site Map framework and the Navigation controls is beyond the scope of this tutorial series, rather than spend time crafting our own navigational user interface let's instead borrow the one used in my *Working with Data in ASP.NET 2.0* tutorial series, which uses a Repeater control to display a two-deep bulleted list of navigation links, as shown in Figure 4.

Adding a Two-Level List of Links in the Left Column

To create this interface, add the following declarative markup to the `Site.master` master page's left column where the text "TODO: Menu will go here..." currently resides.

```
<ul>
     <li>

          <asp:HyperLink runat="server" ID="lnkHome" NavigateUrl="~/Default.aspx">Home</asp:HyperLink>
     </li>
     <asp:Repeater runat="server" ID="menu" DataSourceID="SiteMapDataSource1">

          <ItemTemplate>
               <li>
                    <asp:HyperLink ID="lnkMenuItem" runat="server"
                         NavigateUrl='<%# Eval("Url") %>'><%# Eval("Title") %></asp:HyperLink>

                    <asp:Repeater ID="submenu" runat="server" DataSource="<%#
                         ((SiteMapNode) Container.DataItem).ChildNodes %>">
                         <HeaderTemplate>
                              <ul>
                         </HeaderTemplate>
                         <ItemTemplate>

                              <li>
                                   <asp:HyperLink ID="lnkMenuItem" runat="server" NavigateUrl='<%#
                                        Eval("Url") %>'><%# Eval("Title") %></asp:HyperLink>

                              </li>
                         </ItemTemplate>
                         <FooterTemplate>
                              </ul>
                         </FooterTemplate>
                    </asp:Repeater>
               </li>
          </ItemTemplate>
     </asp:Repeater>

</ul>

<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" ShowStartingNode="false" />
```

The above markup binds a Repeater control named `menu` to a SiteMapDataSource, which returns the site map hierarchy defined in `Web.sitemap`. Since the SiteMapDataSource control's `ShowStartingNode` property is set to False it starts returning the site map's hierarchy starting with the descendents of the "Home" node. The Repeater displays each of these nodes (currently just "Membership") in an `<li>` element. Another, inner Repeater then displays the current node's children in a nested unordered list.

Figure 4 shows the above markup's rendered output with the site map structure we created in Step 2. The Repeater renders vanilla unordered list markup; the cascading style sheet rules defined in `Styles.css` are responsible for the aesthetically-pleasing layout. For a more detailed description of how the above markup works, refer to the Master Pages and Site Navigation tutorial.

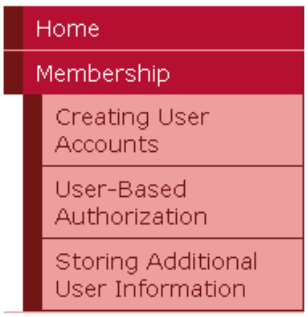


Figure 4: The Navigational User Interface is Rendered Using Nested Unordered Lists (Click to view full-size image)

Adding Breadcrumb Navigation

In addition to the list of links in the left column, let's also have each page display a breadcrumb. A breadcrumb is a navigational user interface element that quickly shows users their current position within the site hierarchy. The SiteMapPath control uses the Site Map framework to determine the current page's location in the site map and then displays a breadcrumb based on this information.

Specifically, add a `<span>` element to the master page's header `<div>` element, and set the new `<span>` element's `class` attribute to "breadcrumb". (The `Styles.css` class contains a rule for a "breadcrumb" class.) Next, add a SiteMapPath to this new `<span>` element.

```
<div id="header">
     <span class="title">User Account Tutorials</span><br />
     <span class="breadcrumb">
          <asp:SiteMapPath ID="SiteMapPath1" runat="server">

          </asp:SiteMapPath>
     </span>
</div>
```

Figure 5 shows the output of the SiteMapPath when visiting `~/Membership/CreatingUserAccounts.aspx`.

Home > Membership > Creating User Accounts

Figure 5: The Breadcrumb Displays the Current Page and its Ancestors in the Site Map (Click to view full-size image)

Step 4: Removing the Custom Principal and Identity Logic

In the *Forms Authentication Configuration and Advanced Topics* tutorial we saw how to associate custom principal and identity objects to the authenticated user. We accomplished this by creating an event handler in `Global.asax` for the application's `PostAuthenticateRequest` event, which fires after the `FormsAuthenticationModule` has authenticated the user. In this event handler we replaced the `GenericPrincipal` and `FormsIdentity` objects added by the `FormsAuthenticationModule` with the `CustomPrincipal` and `CustomIdentity` objects we created in that tutorial.

While custom principal and identity objects are useful in certain scenarios, in most cases the `GenericPrincipal` and `FormsIdentity` objects are sufficient. Consequently, I think it would be worthwhile to return to the default behavior. Make this change by either removing or commenting out the `PostAuthenticateRequest` event handler or by deleting the `Global.asax` file entirely.

Step 5: Programmatically Creating a New User

To create a new user account through the Membership framework use the `Membership` class's `CreateUser` method. This method has input parameters for the username, password, and other user-related fields. On invocation, it delegates the creation of the new user account to the configured Membership provider and then returns a `MembershipUser` object representing the just-created user account.

The `CreateUser` method has four overloads, each accepting a different number of input parameters:

- `CreateUser(username, password)`
- `CreateUser(username, password, email)`
- `CreateUser(username, password, email, passwordQuestion, passwordAnswer, isApproved, MembershipCreateStatus)`
- `CreateUser(username, password, email, passwordQuestion, passwordAnswer, isApproved, providerUserKey, MembershipCreateStatus)`

These four overloads differ on the amount of information that is collected. The first overload, for example,

requires just the username and password for the new user account, whereas the second one also requires the user's email address.

These overloads exist because the information needed to create a new user account depends on the Membership provider's configuration settings. In the *Creating the Membership Schema in SQL Server* tutorial we examined specifying Membership provider configuration settings in `Web.config`. Table 2 included a complete list of the configuration settings.

One such Membership provider configuration setting that impacts what `CreateUser` overloads may be used is the `requiresQuestionAndAnswer` setting. If `requiresQuestionAndAnswer` is set to `true` (the default), then when creating a new user account we must specify a security question and answer. This information is later used if the user needs to reset or change their password. Specifically, at that time they are shown the security question and they must enter the correct answer in order to reset or change their password. Consequently, if the `requiresQuestionAndAnswer` is set to `true` then calling either of the first two `CreateUser` overloads results in an exception because the security question and answer are missing. Since our application is currently configured to require a security question and answer, we will need to use one of the latter two overloads when creating user's programmatically.

To illustrate using the `CreateUser` method, let's create a user interface where we prompt the user for their name, password, email, and an answer to a pre-defined security question. Open the `CreatingUserAccounts.aspx` page in the `Membership` folder and add the following Web controls to the Content control:

- A TextBox named `Username`
- A TextBox named `Password`, whose `TextMode` property is set to `Password`
- A TextBox named `Email`
- A Label named `SecurityQuestion` with its `Text` property cleared out
- A TextBox named `SecurityAnswer`
- A Button named `CreateAccountButton` whose Text property is set to "Create the User Account"
- A Label control named `CreateAccountResults` with its `Text` property cleared out

At this point your screen should look similar to the screen shot shown in Figure 6.

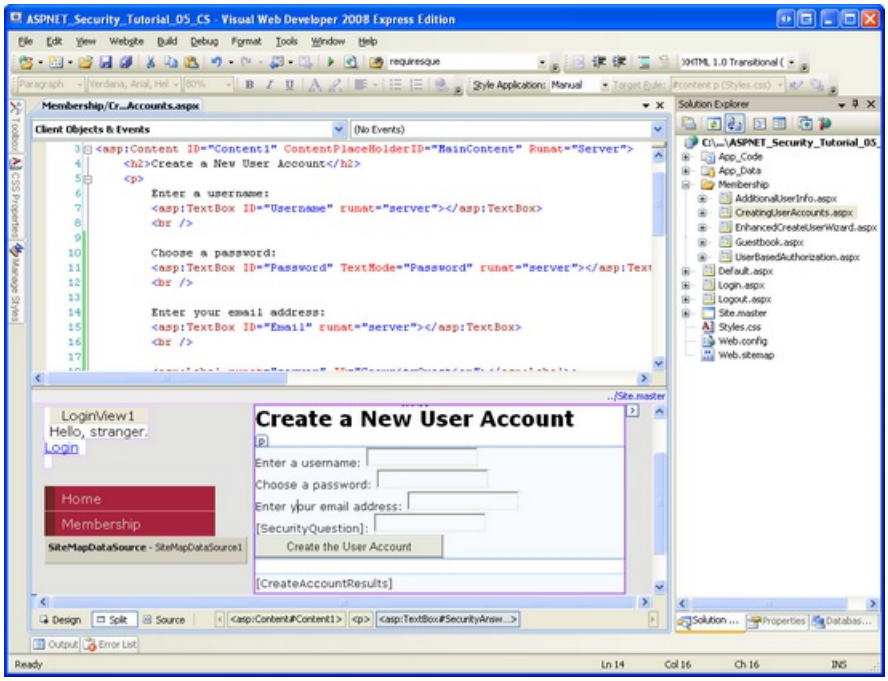


Figure 6: Add the Various Web Controls to the `CreatingUserAccounts.aspx` Page (Click to view full-size image)

The `SecurityQuestion` Label and `SecurityAnswer` TextBox are intended to display a pre-defined security question and collect the user's answer. Note that both the security question and answer are stored on a user-by-user

basis, so it is possible to allow each user to define their own security question. However, for this example I have decided to use a universal security question, namely: "What is your favorite color?"

To implement this pre-defined security question, add a constant to the page's code-behind class named `passwordQuestion`, assigning it the security question. Then, in the `Page_Load` event handler, assign this constant to the `SecurityQuestion` Label's `Text` property:

```
const string passwordQuestion = "What is your favorite color";

protected void Page_Load(object sender, EventArgs e)
{
     if (!Page.IsPostBack)
          SecurityQuestion.Text = passwordQuestion;
}
```

Next, create an event handler for the `CreateAccountButton`'s `Click` event and add the following code:

```
protected void CreateAccountButton_Click(object sender, EventArgs e)
{
     MembershipCreateStatus createStatus;
     MembershipUser newUser = Membership.CreateUser(Username.Text, Password.Text, Email.Text,
passwordQuestion, SecurityAnswer.Text, true, out createStatus);
     switch (createStatus)
     {
          case MembershipCreateStatus.Success:
               CreateAccountResults.Text = "The user account was successfully created!";
               break;
          case MembershipCreateStatus.DuplicateUserName:
               CreateAccountResults.Text = "There already exists a user with this username.";
               break;

          case MembershipCreateStatus.DuplicateEmail:
               CreateAccountResults.Text = "There already exists a user with this email address.";
               break;
          case MembershipCreateStatus.InvalidEmail:
               CreateAccountResults.Text = "There email address you provided in invalid.";
               break;
          case MembershipCreateStatus.InvalidAnswer:
               CreateAccountResults.Text = "There security answer was invalid.";
               break;
          case MembershipCreateStatus.InvalidPassword:
               CreateAccountResults.Text = "The password you provided is invalid. It must be seven
characters long and have at least one non-alphanumeric character.";

               break;
          default:
               CreateAccountResults.Text = "There was an unknown error; the user account was NOT created.";
               break;
     }
}
```

The `Click` event handler starts by defining a variable named `createStatus` of type `MembershipCreateStatus`. `MembershipCreateStatus` is an enumeration that indicates the status of the `CreateUser` operation. For example, if the user account is created successfully, the resulting `MembershipCreateStatus` instance will be set to a value of `Success`; on the other hand, if the operation fails because there already exists a user with the same username, it will be set to a value of `DuplicateUserName`. In the `CreateUser` overload we use, we need to pass a `MembershipCreateStatus` instance into the method as an `out` parameter. This parameter is set to the appropriate value within the `CreateUser` method, and we can examine its value after the method call to determine whether the user account was successfully created.

After calling `CreateUser`, passing in `createStatus`, a `switch` statement is used to output an appropriate

message depending on the value assigned to `createStatus`. Figures 7 shows the output when a new user has successfully been created. Figures 8 and 9 show the output when the user account is not created. In Figure 8, the visitor entered a five-letter password, which does not meet the password strength requirements spelled out in the Membership provider's configuration settings. In Figure 9, the visitor is attempting to create a user account with an existing username (the one created in Figure 7).

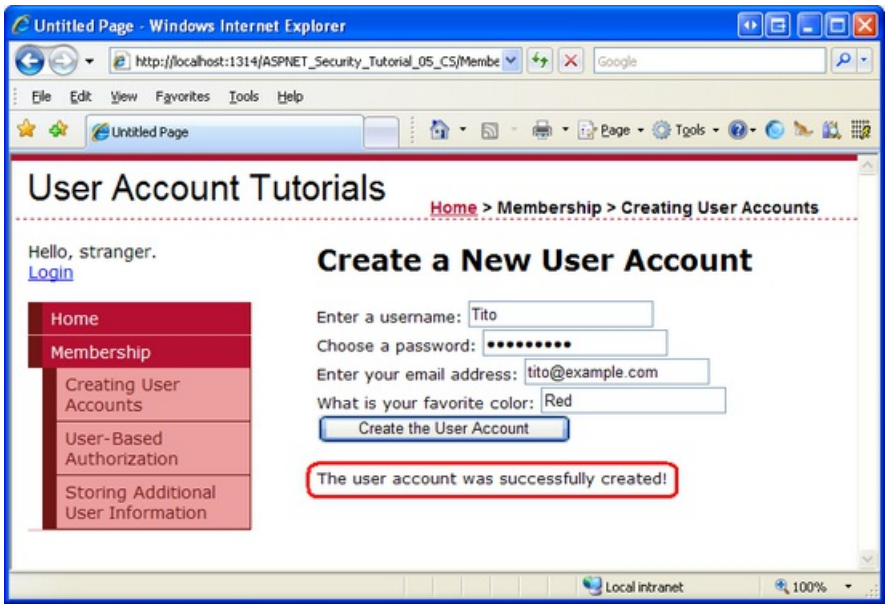


Figure 7: A New User Account is Successfully Created (Click to view full-size image)

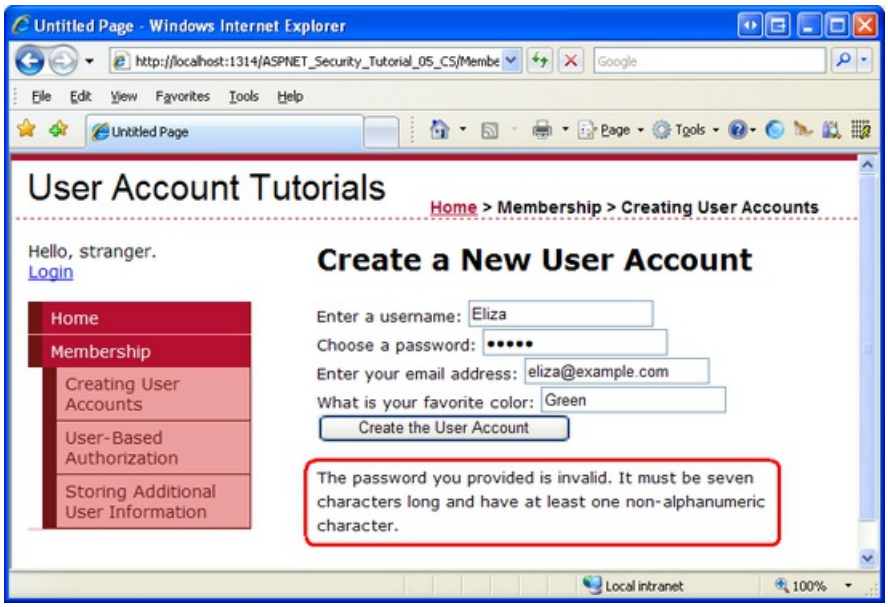


Figure 8: The User Account is Not Created Because the Supplied Password is Too Weak (Click to view full-size image)

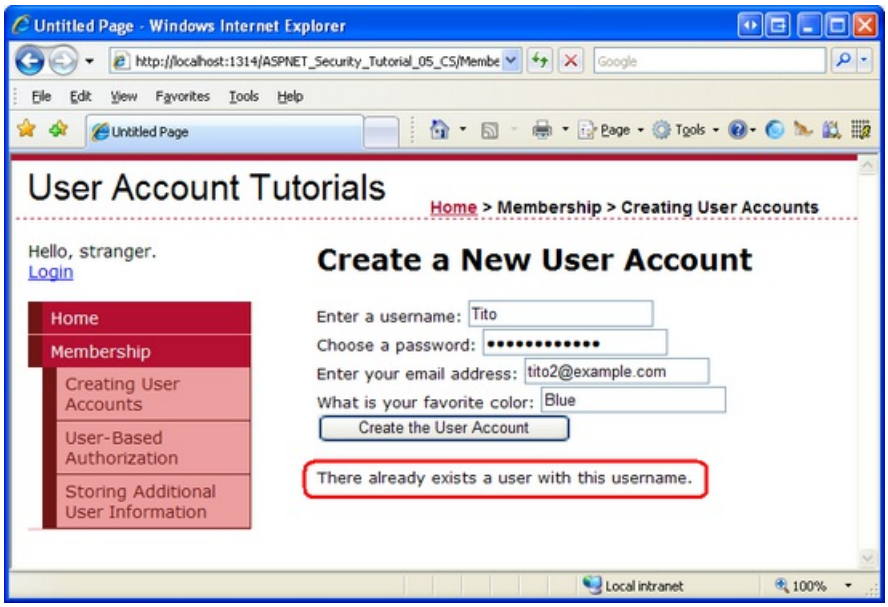


Figure 9: The User Account is Not Created Because the Username is Already in Use (Click to view full-size image)

> **NOTE**
>
> You may be wondering how to determine success or failure when using one of the first two `CreateUser` method overloads, neither of which has a parameter of type `MembershipCreateStatus`. These first two overloads throw a `MembershipCreateUserException` exception in the face of a failure, which includes a `StatusCode` property of type `MembershipCreateStatus`.

After creating a few user accounts, verify that the accounts have been created by listing the contents of the `aspnet_Users` and `aspnet_Membership` tables in the `SecurityTutorials.mdf` database. As Figure 10 shows, I have added two users via the `CreatingUserAccounts.aspx` page: Tito and Bruce.

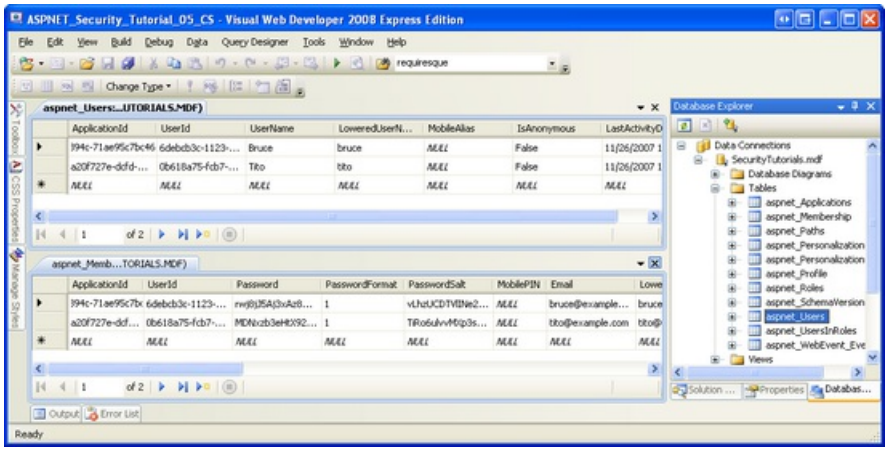


Figure 10: There are Two Users in the Membership User Store: Tito and Bruce (Click to view full-size image)

While the Membership user store now includes Bruce and Tito's account information, we have yet to implement functionality that allows for Bruce or Tito to log on to the site. Currently, `Login.aspx` validates the user's credentials against a hard-coded set of username/password pairs – it does *not* validate the supplied credentials against the Membership framework. For now seeing the new user accounts in the `aspnet_Users` and `aspnet_Membership` tables will have to suffice. In the next tutorial, *Validating User Credentials Against the Membership User Store*, we will update the login page to validate against the Membership store.

> **NOTE**
> If you do not see any users in your `SecurityTutorials.mdf` database, it may be because your web application is using the default Membership provider, `AspNetSqlMembershipProvider`, which uses the `ASPNETDB.mdf` database as its user store. To determine if this is the problem, click the Refresh button in the Solution Explorer. If a database named `ASPNETDB.mdf` has been added to the `App_Data` folder, this is the problem. Return to Step 4 of the *Creating the Membership Schema in SQL Server* tutorial for instructions on how to properly configure the Membership provider.

In most create user account scenarios, the visitor is presented with some interface to enter their username, password, email, and other essential information, at which point a new account is created. In this step we looked at building such an interface by hand and then saw how to use the `Membership.CreateUser` method to programmatically add the new user account based on the user's inputs. Our code, however, just created the new user account. It did not perform any follow up actions, like logging in the user to the site under the just-created user account, or sending a confirmation email to the user. These additional steps would require additional code in the Button's `Click` event handler.

ASP.NET ships with the CreateUserWizard control, which is designed to handle the user account creation process, from rendering a user interface for creating new user accounts, to creating the account in the Membership framework and performing post-account creation tasks, such as sending a confirmation email and logging the just-created user into the site. Using the CreateUserWizard control is as simple as dragging the CreateUserWizard control from the Toolbox onto a page, and then setting a few properties. In most cases, you won't need to write a single line of code. We will explore this nifty control in detail in Step 6.

If new user accounts are only created through a typical Create Account web page, it is unlikely that you will ever need to write code that uses the `CreateUser` method, as the CreateUserWizard control will likely meet your needs. However, the `CreateUser` method is handy in scenarios where you need a highly customized Create Account user experience or when you need to programmatically create new user accounts through an alternative interface. For example, you might have a page that allows a user to upload an XML file that contains user information from some other application. The page might parses the contents of the uploaded XML file and create a new account for each user represented in the XML by calling the `CreateUser` method.

Step 6: Creating a New User with the CreateUserWizard Control

ASP.NET ships with a number of Login Web controls. These controls aid in many common user account- and login-related scenarios. The CreateUserWizard control is one such control that is designed to present a user interface for adding a new user account to the Membership framework.

Like many of the other Login-related Web controls, the CreateUserWizard can be used without writing a single line of code. It intuitively provides a user interface based on the Membership provider's configuration settings and internally calls the `Membership` class's `CreateUser` method after the user enters the necessary information and clicks the "Create User" button. The CreateUserWizard control is extremely customizable. There are a host of events that fire during various stages of the account creation process. We can create event handlers, as needed, to inject custom logic into the account creation workflow. Furthermore, the CreateUserWizard's appearance is very flexible. There are a number of properties that define the appearance of the default interface; if necessary, the control can be converted into a template or additional user registration "steps" may be added.

Let's start with a look at using the CreateUserWizard control's default interface and behavior. We'll then explore how to customize the appearance via the control's properties and events.

Examining the CreateUserWizard's Default Interface and Behavior

Return to the `CreatingUserAccounts.aspx` page in the `Membership` folder, switch to the Design or Split mode, and then add a CreateUserWizard control to the top of the page. The CreateUserWizard control is filed under the Toolbox's Login controls section. After adding the control, set its `ID` property to `RegisterUser`. As the screen

shot in Figure 11 shows, the CreateUserWizard renders an interface with textboxes for the new user's username, password, email address, and security question and answer.

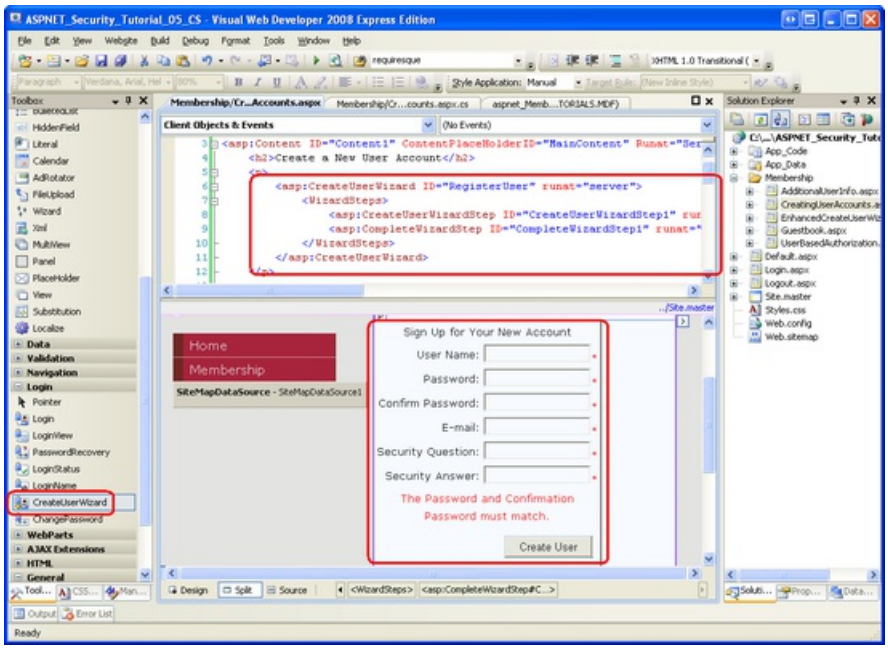


Figure 11: The CreateUserWizard Control Renders a Generic Create User Interface (Click to view full-size image)

Let's take a moment to compare the default user interface generated by the CreateUserWizard control with the interface we created in Step 5. For starters, the CreateUserWizard control allows the visitor to specify both the security question and answer, whereas our manually-created interface used a pre-defined security question. The CreateUserWizard control's interface also includes validation controls, whereas we had yet to implement validation on our interface's form fields. And the CreateUserWizard control interface includes a "Confirm Password" textbox (along with a CompareValidator to ensure that the text entered the "Password" and "Compare Password" textboxes are equal).

What's interesting is that the CreateUserWizard control consults the Membership provider's configuration settings when rendering its user interface. For example, the security question and answer textboxes are only displayed if `requiresQuestionAndAnswer` is set to True. Likewise, CreateUserWizard automatically adds a RegularExpressionValidator control to ensure that the password strength requirements are met, and sets its `ErrorMessage` and `ValidationExpression` properties based on the `minRequiredPasswordLength`, `minRequiredNonalphanumericCharacters`, and `passwordStrengthRegularExpression` configuration settings.

The CreateUserWizard control, as its name implies, is derived from the Wizard control. Wizard controls are designed to provide an interface for completing multi-step tasks. A Wizard control may have an arbitrary number of `WizardSteps`, each of which is a template that defines the HTML and Web controls for that step. The Wizard control initially displays the first `WizardStep`, along with navigation controls that permit the user to proceed from one step to the next, or to return to previous steps.

As the declarative markup in Figure 11 shows, the CreateUserWizard control's default interface includes two `WizardSteps:`

- `CreateUserWizardStep` – renders the interface to collect information for creating the new user account. This is the step shown in Figure 11.
- `CompleteWizardStep` – renders a message indicating that the account has successfully been created.

The CreateUserWizard's appearance and behavior can be modified by converting either of these steps to templates, or by adding your own `WizardSteps`. We will look at adding a `WizardStep` to the registration interface in the *Storing Additional User Information* tutorial.

Let's see the CreateUserWizard control in action. Visit the `CreatingUserAccounts.aspx` page through a browser. Start by entering some invalid values into the CreateUserWizard's interface. Try entering a password that doesn't conform to the password strength requirements, or leaving the "User Name" textbox empty. The CreateUserWizard will display an appropriate error message. Figure 12 shows the output when attempting to create a user with an insufficiently strong password.

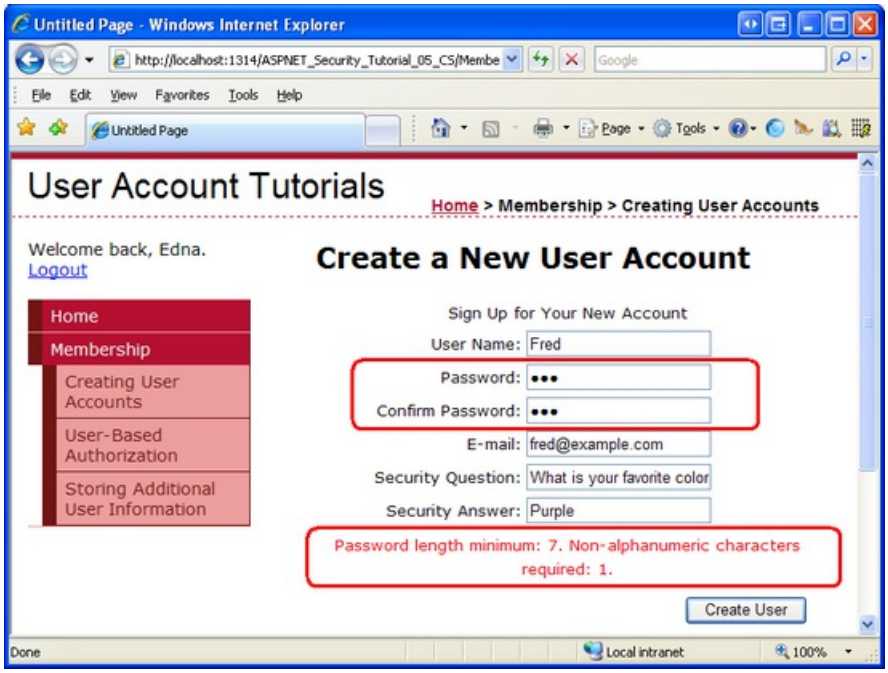


Figure 12: The CreateUserWizard Automatically Injects Validation Controls (Click to view full-size image)

Next, enter appropriate values into the CreateUserWizard and click the "Create User" button. Assuming the required fields have been entered and the password's strength is sufficient, the CreateUserWizard will create a new user account through the Membership framework and then display the `CompleteWizardStep`'s interface (see Figure 13). Behind the scenes, the CreateUserWizard calls the `Membership.CreateUser` method, just like we did in Step 5.

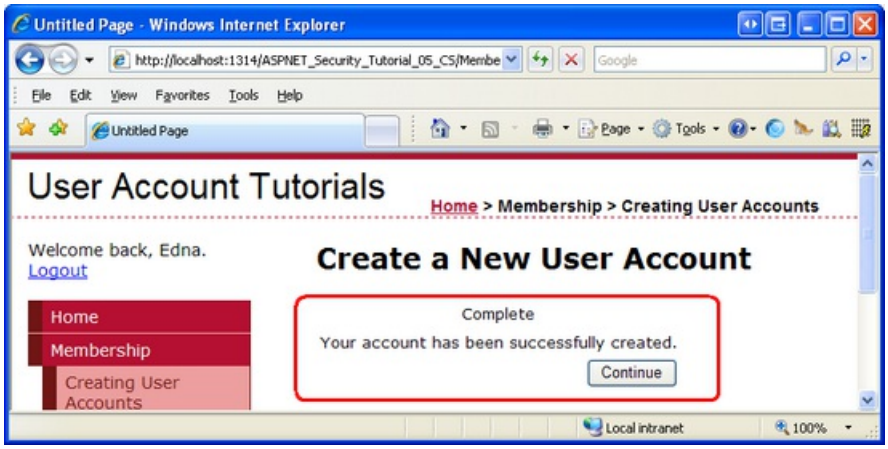


Figure 13: A New User Account has Successfully Been Created (Click to view full-size image)

> **NOTE**
>
> As Figure 13 shows, the `CompleteWizardStep`'s interface includes a Continue button. However, at this point clicking it just performs a postback, leaving the visitor on the same page. In the "Customizing the CreateUserWizard's Appearance and Behavior Through Its Properties" section we will look at how you can have this button send the visitor to `Default.aspx` (or some other page).

After creating a new user account, return to Visual Studio and examine the `aspnet_Users` and `aspnet_Membership`

tables like we did in Figure 10 to verify that the account was successfully created.

Customizing the CreateUserWizard's Behavior and Appearance Through Its Properties

The CreateUserWizard can be customized in a variety of ways, through properties, `WizardSteps`, and event handlers. In this section we will look at how to customize the control's appearance through its properties; the next section looks at extending the control's behavior through event handlers.

Virtually all of the text displayed in the CreateUserWizard control's default user interface can be customized through its plethora of properties. For example, the "User Name", "Password", "Confirm Password", "E-mail", "Security Question", and "Security Answer" labels displayed to the left of the textboxes can be customized by the `UserNameLabelText`, `PasswordLabelText`, `ConfirmPasswordLabelText`, `EmailLabelText`, `QuestionLabelText`, and `AnswerLabelText` properties, respectively. Likewise, there are properties for specifying the text for the "Create User" and "Continue" buttons in the `CreateUserWizardStep` and `CompleteWizardStep`, as well as if these buttons are rendered as Buttons, LinkButtons, or ImageButtons.

The colors, borders, fonts, and other visual elements are configurable through a host of style properties. The CreateUserWizard control itself has the common Web control style properties – `BackColor`, `BorderStyle`, `CssClass`, `Font`, and so on – and there are a number of style properties for defining the appearance for particular sections of the CreateUserWizard's interface. The `TextBoxStyle` property, for instance, defines the styles for the textboxes in the `CreateUserWizardStep`, while the `TitleTextStyle` property defines the style for the title ("Sign Up for Your New Account").

In addition to the appearance-related properties, there are a number of properties that affect the CreateUserWizard control's behavior. The `DisplayCancelButton` property, if set to True, displays a Cancel button next to the "Create User" button (the default value is False). If you display the Cancel button, be sure to also set the `CancelDestinationPageUrl` property, which specifies the page the user is sent to after clicking Cancel. As noted in the previous section, the Continue button in the `CompleteWizardStep`'s interface causes a postback, but leaves the visitor on the same page. To send the visitor to some other page after clicking the Continue button, simply specify the URL in the `ContinueDestinationPageUrl` property.

Let's update the `RegisterUser` CreateUserWizard control to show a Cancel button and to send the visitor to `Default.aspx` when the Cancel or Continue buttons are clicked. To accomplish this, set the `DisplayCancelButton` property to True and both the `CancelDestinationPageUrl` and `ContinueDestinationPageUrl` properties to "~/Default.aspx". Figure 14 shows the updated CreateUserWizard when viewed through a browser.

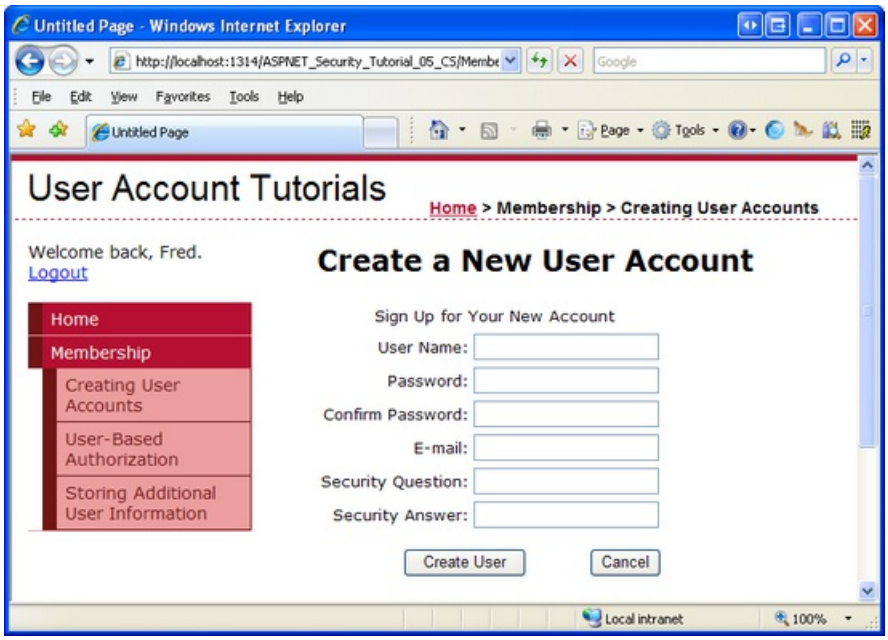


Figure 14: The `CreateUserWizardStep` Includes a Cancel Button (Click to view full-size image)

When a visitor enters a username, password, email address, and security question and answer and clicks "Create

User," a new user account is created and the visitor is logged in as that newly created user. Assuming that the person visiting the page is creating a new account for themselves, this is likely the desired behavior. However, you may want to allow Administrators to add new user accounts. In doing so, the user account would be created, but the Administrator would remain logged in as an Administrator (and not as the newly created account). This behavior can be modified through the Boolean `LoginCreatedUser` property.

User accounts in the Membership framework contain an approved flag; users who are not approved are unable to log into the site. By default, a newly created account is marked as approved, allowing the user to log into the site immediately. It is possible, however, to have new user accounts marked as unapproved. Perhaps you want an Administrator to manually approve new users before they can log in; or maybe you want to verify that the email address entered at signup is valid before permitting a user to log on. Whatever the case may be, you can have the newly created user account marked as unapproved by setting the CreateUserWizard control's `DisableCreatedUser` property to True (the default is False).

Other behavior-related properties of note include `AutoGeneratePassword` and `MailDefinition`. If the `AutoGeneratePassword` property is set to True, the `CreateUserWizardStep` does not display the "Password" and "Confirm Password" textboxes; instead, the newly-created user's password is automatically generated using the `Membership` class's `GeneratePassword` method. The `GeneratePassword` method constructs a password of a specified length and with a sufficient number of non-alphanumeric characters to satisfy the configured password strength requirements.

The `MailDefinition` property is useful if you want to send an email to the email address specified during the account creation process. The `MailDefinition` property includes a series of subproperties for defining information about the constructed email message. These subproperties include options like `Subject`, `Priority`, `IsBodyHtml`, `From`, `CC`, and `BodyFileName`. The `BodyFileName` property points to a text or HTML file that contains the body for the email message. The body supports two pre-defined placeholders: `<%UserName%>` and `<%Password%>`. These placeholders, if present in the `BodyFileName` file, will be replaced with the just-created user's name and password.

> **NOTE**
>
> The `CreateUserWizard` control's `MailDefinition` property just specifies details about the email message that is sent when a new account is created. It does not include any details on how the email message is actually sent (that is, whether an SMTP server or mail drop directory is used, any authentication information, and so on). These low-level details need to be defined in the `<system.net>` section in `Web.config`. For more information on these configuration settings and on sending email from ASP.NET 2.0 in general, refer to the FAQs at SystemNetMail.com and my article, Sending Email in ASP.NET 2.0.

Extending the CreateUserWizard's Behavior Using Event Handlers

The CreateUserWizard control raises a number of events during its workflow. For example, after a visitor enters their username, password, and other pertinent information and clicks the "Create User" button, the CreateUserWizard control raises its `CreatingUser` event. If there is a problem during the create process, the `CreateUserError` event is fired; however, if the user is successfully created, then the `CreatedUser` event is raised. There are additional CreateUserWizard control events that get raised, but these are the three most germane ones.

In certain scenarios we may want to tap into the CreateUserWizard workflow, which we can do by creating an event handler for the appropriate event. To illustrate this, let's enhance the `RegisterUser` CreateUserWizard control to include some custom validation on the username and password. In particular, let's enhance our CreateUserWizard so that usernames cannot contain leading or trailing spaces and the username cannot appear anywhere in the password. In short, we want to prevent someone from creating a username like "Scott ", or having a username/password combination like "Scott" and "Scott.1234".

To accomplish this we will create an event handler for the `CreatingUser` event to perform our extra validation

checks. If the supplied data is not valid we need to cancel the creation process. We also need to add a Label Web control to the page to display a message explaining that the username or password is invalid. Start by adding a Label control beneath the CreateUserWizard control, setting its `ID` property to `InvalidUserNameOrPasswordMessage` and its `ForeColor` property to `Red`. Clear out its `Text` property and set its `EnableViewState` and `Visible` properties to False.

```
<asp:Label runat="server" id="InvalidUserNameOrPasswordMessage"
     Visible="false" ForeColor="Red" EnableViewState="false">

</asp:Label>
```

Next, create an event handler for the CreateUserWizard control's `CreatingUser` event. To create an event handler, select the control in the Designer and then go to the Properties window. From there, click on the lightning bolt icon and then double-click the appropriate event to create the event handler.

Add the following code to the `CreatingUser` event handler:

```
protected void RegisterUser_CreatingUser(object sender, LoginCancelEventArgs e)
{
     string trimmedUserName = RegisterUser.UserName.Trim();
     if (RegisterUser.UserName.Length != trimmedUserName.Length)
     {
          // Show the error message
          InvalidUserNameOrPasswordMessage.Text = "The username cannot contain leading or trailing spaces.";
          InvalidUserNameOrPasswordMessage.Visible = true;

          // Cancel the create user workflow
          e.Cancel = true;
     }
     else
     {
          // Username is valid, make sure that the password does not contain the username

          if (RegisterUser.Password.IndexOf(RegisterUser.UserName, StringComparison.OrdinalIgnoreCase) >= 0)
          {
               // Show the error message
               InvalidUserNameOrPasswordMessage.Text = "The username may not appear anywhere in the
password.";
               InvalidUserNameOrPasswordMessage.Visible = true;
               // Cancel the create user workflow
               e.Cancel = true;
          }
     }
}
```

Note that the username and password entered into the CreateUserWizard control are available through its `UserName` and `Password` properties, respectively. We use these properties in the above event handler to determine whether the supplied username contains leading or trailing spaces and whether the username is found within the password. If either of these conditions is met, an error message is displayed in the `InvalidUserNameOrPasswordMessage` Label and the event handler's `e.Cancel` property is set to `true`. If `e.Cancel` is set to `true`, the CreateUserWizard short-circuits its workflow, effectively cancelling the user account creation process.

Figure 15 shows a screen shot of `CreatingUserAccounts.aspx` when the user enters a username with leading spaces.

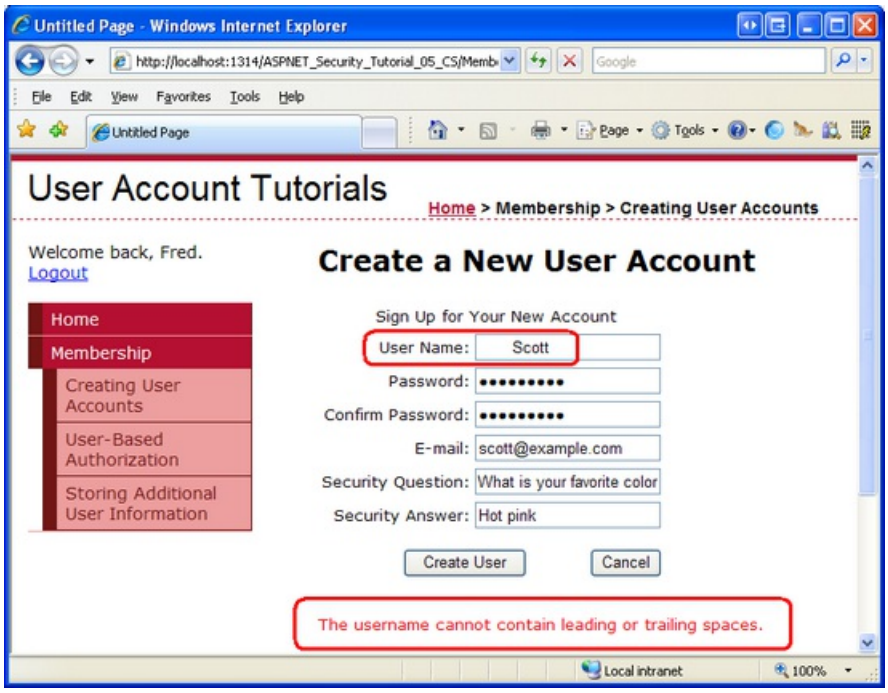


Figure 15: Usernames with Leading or Trailing Spaces are not Permitted (Click to view full-size image)

> **NOTE**
>
> We will see an example of using the CreateUserWizard control's `CreatedUser` event in the *Storing Additional User Information* tutorial.

Summary

The `Membership` class's `CreateUser` method creates a new user account in the Membership framework. It does so by delegating the call to the configured Membership provider. In the case of the `SqlMembershipProvider`, the `CreateUser` method adds a record to the `aspnet_Users` and `aspnet_Membership` database tables.

While new user accounts can be created programmatically (as we saw in Step 5), a faster and easier approach is to use the CreateUserWizard control. This control renders a multi-step user interface for collecting user information and creating a new user in the Membership framework. Underneath the covers, this control uses the same `Membership.CreateUser` method as examined in Step 5, but the control creates the user interface, validation controls, and responds to user account creation errors without having to write a lick of code.

At this point we have the functionality in place to create new user accounts. However, the login page still is validating against those hard-coded credentials we specified back in the second tutorial. In the next tutorial we will update `Login.aspx` to validate the user's supplied credentials against the Membership framework.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- `CreateUser` Technical Documentation
- CreateUserWizard Control Overview
- Creating a File System-Based Site Map Provider
- Creating a Step-by-Step User Interface with the ASP.NET 2.0 Wizard Control
- Examining ASP.NET 2.0's Site Navigation
- Master Pages and Site Navigation
- The SQL Site Map Provider You've Been Waiting For

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Validating User Credentials Against the Membership User Store (C#)

9/16/2020 • 25 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

In this tutorial we will examine how to validate a user's credentials against the Membership user store using both programmatic means and the Login control. We will also look at how to customize the login control's appearance and behavior.

Introduction

In the preceding tutorial we looked at how to create a new user account in the Membership framework. We first looked at programmatically creating user accounts via the `Membership` class's `CreateUser` method, and then examined using the CreateUserWizard Web control. However, the login page currently validates the supplied credentials against a hard-coded list of username and password pairs. We need to update the login page's logic so that it validates credentials against the Membership framework's user store.

Much like with creating user accounts, credentials can be validated programmatically or declaratively. The Membership API includes a method for programmatically validating a user's credentials against the user store. And ASP.NET ships with the Login Web control, which renders a user interface with textboxes for the username and password and a button to log in.

In this tutorial we will examine how to validate a user's credentials against the Membership user store using both programmatic means and the Login control. We will also look at how to customize the login control's appearance and behavior. Let's get started!

Step 1: Validating Credentials Against the Membership User Store

For web sites that use forms authentication, a user logs on to the website by visiting a login page and entering their credentials. These credentials are then compared against the user store. If they are valid, then the user is granted a forms authentication ticket, which is a security token that indicates the identity and authenticity of the visitor.

To validate a user against the Membership framework, use the `Membership` class's `ValidateUser` method. The `ValidateUser` method takes in two input parameters - *`username`* and *`password`* - and returns a Boolean value indicating whether the credentials were valid. Like with the `CreateUser` method we examined in the previous tutorial, the `ValidateUser` method delegates the actual validation to the configured Membership provider.

The `SqlMembershipProvider` validates the supplied credentials by obtaining the specified user's password via the `aspnet_Membership_GetPasswordWithFormat` stored procedure. Recall that the `SqlMembershipProvider` stores users' passwords using one of three formats: clear, encrypted, or hashed. The `aspnet_Membership_GetPasswordWithFormat` stored procedure returns the password in its raw format. For encrypted or hashed passwords, the `SqlMembershipProvider` transforms the *`password`* value passed into the `ValidateUser` method into its equivalent encrypted or hashed state and then compares it with what was returned from the database. If the password stored in the database matches the formatted password entered by the user, the credentials are valid.

Let's update our login page (~/`Login.aspx`) so that it validates the supplied credentials against the Membership

framework user store. We created this login page back in the *An Overview of Forms Authentication* tutorial, creating an interface with two TextBoxes for the username and password, a Remember Me checkbox, and a Login button (see Figure 1). The code validates the entered credentials against a hard-coded list of username and password pairs (Scott/password, Jisun/password, and Sam/password). In the *Forms Authentication Configuration and Advanced Topics* tutorial we updated the login page's code to store additional information in the forms authentication ticket's `UserData` property.

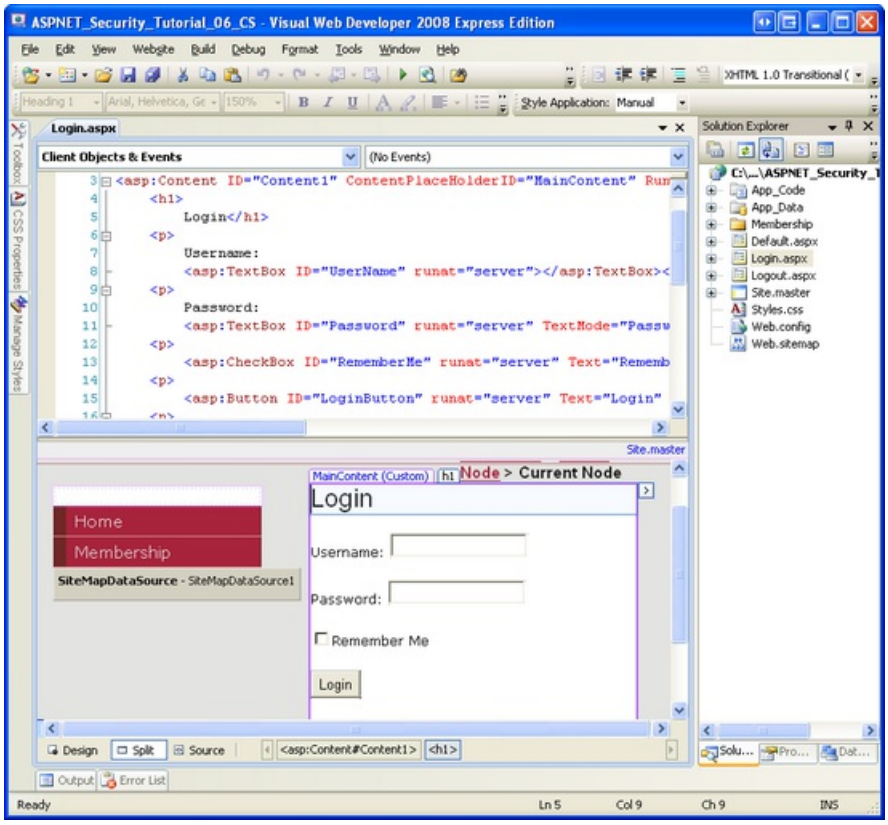


Figure 1: The Login Page's Interface Includes Two TextBoxes, a CheckBoxList, and a Button (Click to view full-size image)

The login page's user interface can remain unchanged, but we need to replace the Login button's `Click` event handler with code that validates the user against the Membership framework user store. Update the event handler so that its code appears as follows:

```
protected void LoginButton_Click(object sender, EventArgs e)
{
    // Validate the user against the Membership framework user store
    if (Membership.ValidateUser(UserName.Text, Password.Text))
    {
        // Log the user into the site
        FormsAuthentication.RedirectFromLoginPage(UserName.Text, RememberMe.Checked);
    }
    // If we reach here, the user's credentials were invalid
    InvalidCredentialsMessage.Visible = true;
}
```

This code is remarkably simple. We start by calling the `Membership.ValidateUser` method, passing in the supplied username and password. If that method returns true, then the user is signed into the site via the `FormsAuthentication` class's RedirectFromLoginPage method. (As we discussed in the *An Overview of Forms Authentication* tutorial, the `FormsAuthentication.RedirectFromLoginPage` creates the forms authentication ticket and then redirects the user to the appropriate page.) If the credentials are invalid, however, the `InvalidCredentialsMessage` Label is displayed, informing the user that their username or password was incorrect.

That's all there is to it!

To test that the login page works as expected, attempt to login with one of the user accounts you created in the preceding tutorial. Or, if you have not yet created an account, go ahead and create one from the `~/Membership/CreatingUserAccounts.aspx` page.

> **NOTE**
> When the user enters her credentials and submits the login page form, the credentials, including her password, are transmitted over the Internet to the web server in *plain text*. That means any hacker sniffing the network traffic can see the username and password. To prevent this, it is essential to encrypt the network traffic by using Secure Socket Layers (SSL). This will ensure that the credentials (as well as the entire page's HTML markup) are encrypted from the moment they leave the browser until they are received by the web server.

How the Membership Framework Handles Invalid Login Attempts

When a visitor reaches the login page and submits their credentials, their browser makes an HTTP request to the login page. If the credentials are valid, the HTTP response includes the authentication ticket in a cookie. Therefore, a hacker attempting to break into your site could create a program that exhaustively sends HTTP requests to the login page with a valid username and a guess at the password. If the password guess is correct, the login page will return the authentication ticket cookie, at which point the program knows it has stumbled upon a valid username/password pair. Through brute force, such a program might be able to stumble upon a user's password, especially if the password is weak.

To prevent such brute force attacks, the Membership framework locks out a user if there are a certain number of unsuccessful login attempts within a certain period of time. The exact parameters are configurable through the following two Membership provider configuration settings:

- `maxInvalidPasswordAttempts` - specifies how many invalid password attempts are allowed for the user within the time period before the account is locked out. The default value is 5.
- `passwordAttemptWindow` - indicates the time period in minutes during which the specified number of invalid login attempts will cause the account to be locked out. The default value is 10.

If a user has been locked out, she cannot login until an administrator unlocks her account. When a user is locked out, the `ValidateUser` method will *always* return `false`, even if valid credentials are supplied. While this behavior lessens the likelihood that a hacker will break into your site through brute force methods, it can end up locking out a valid user who has simply forgotten her password or accidentally has the Caps Lock on or is having a bad typing day.

Unfortunately, there is no built-in tool for unlocking a user account. In order to unlock an account, you can modify the database directly - change the `IsLockedOut` field in the `aspnet_Membership` table for the appropriate user account - or create a web-based interface that lists locked out accounts with options to unlock them. We will examine creating administrative interfaces for accomplishing common user account- and role-related tasks in a future tutorial.

> **NOTE**
> One downside of the `ValidateUser` method is that when the supplied credentials are invalid, it does not provide any explanation as to why. The credentials may be invalid because there is no matching username/password pair in the user store, or because the user has not yet been approved, or because the user has been locked out. In Step 4 we will see how to show a more detailed message to the user when their login attempt fails.

Step 2: Collecting Credentials through the Login Web Control

The Login Web control renders a default user interface very similar to the one we created back in the *An Overview of Forms Authentication* tutorial. Using the Login control saves us the work of having to create the

interface to collect the visitor's credentials. Moreover, the Login control automatically signs in the user (assuming the submitted credentials are valid), thereby saving us from having to write any code.

Let's update `Login.aspx`, replacing the manually created interface and code with a Login control. Start by removing the existing markup and code in `Login.aspx`. You may delete it outright, or simply comment it out. To comment out declarative markup, surround it with the `<%--` and `--%>` delimiters. You can enter these delimiters manually, or, as Figure 2 shows, you may select the text to comment out and then click the Comment out the selected lines icon in the Toolbar. Similarly, you can use the Comment out the selected lines icon to comment out the selected code in the code-behind class.

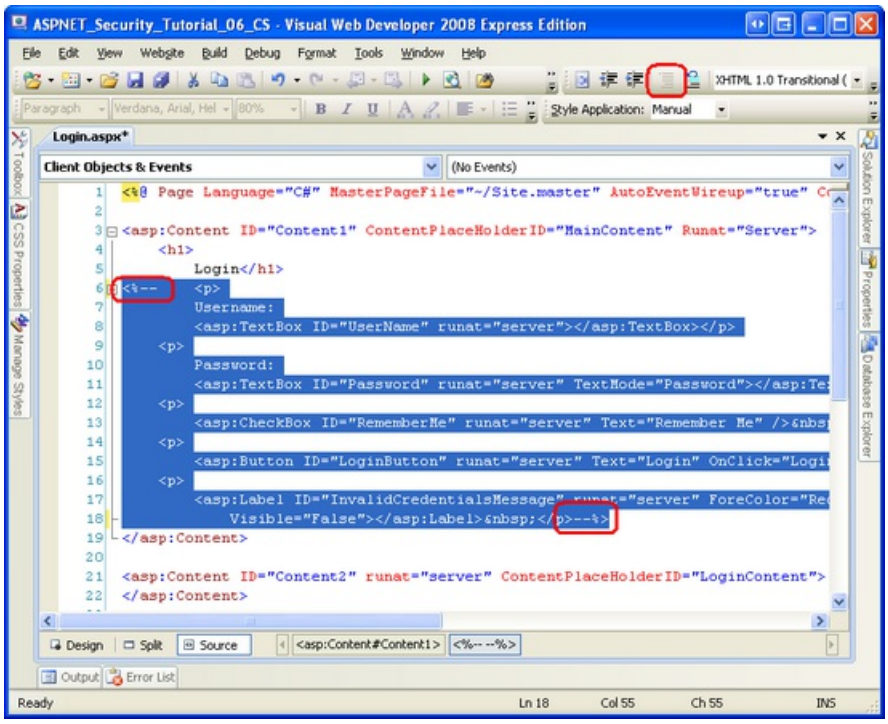


Figure 2: Comment Out the Existing Declarative Markup and Source Code in `Login.aspx` (Click to view full-size image)

> **NOTE**
>
> The Comment out the selected lines icon is not available when viewing the declarative markup in Visual Studio 2005. If you are not using Visual Studio 2008 you will need to manually add the `<%--` and `--%>` delimiters.

Next, drag a Login control from the Toolbox on to the page and set its `ID` property to `myLogin`. At this point your screen should look similar to Figure 3. Note that the Login control's default interface includes TextBox controls for the username and password, a Remember me next time CheckBox, and a Log In Button. There are also `RequiredFieldValidator` controls for the two TextBoxes.

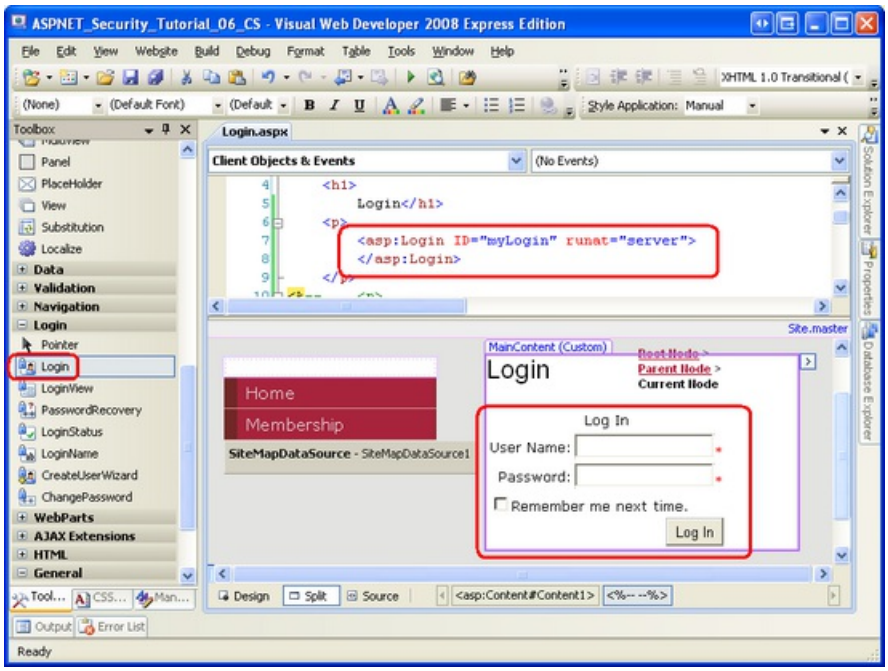


Figure 3: Add a Login Control to the Page (Click to view full-size image)

And we're done! When the Login control's Log In button is clicked, a postback will occur and the Login control will call the `Membership.ValidateUser` method, passing in the entered username and password. If the credentials are invalid, the Login control displays a message stating such. If, however, the credentials are valid, then the Login control creates the forms authentication ticket and redirects the user to the appropriate page.

The Login control uses four factors to determine the appropriate page to redirect the user to upon a successful login:

- Whether the Login control is on the login page as defined by `loginUrl` setting in the forms authentication configuration; this setting's default value is `Login.aspx`
- The presence of a `ReturnUrl` querystring parameter
- The value of the Login control's `DestinationUrl` property
- The `defaultUrl` value specified in the forms authentication configuration settings; this setting's default value is `Default.aspx`

Figure 4 depicts the how the Login control uses these four parameters to arrive at its appropriate page decision.

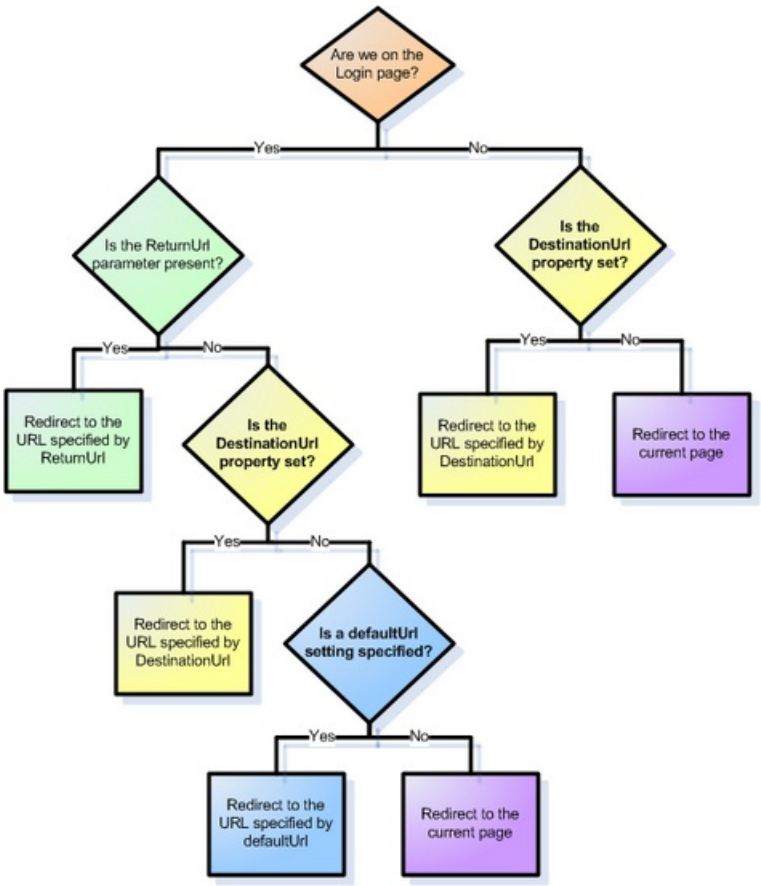


Figure 4: Add a Login Control to the Page (Click to view full-size image)

Take a moment to test out the Login control by visiting the site through a browser and logging in as an existing user in the Membership framework.

The Login control's rendered interface is highly configurable. There are a number of properties that influence its appearance; what's more, the Login control can be converted into a template for precise control over the layout the user interface elements. The remainder of this step examines how to customize the appearance and layout.

Customizing the Login Control's Appearance

The Login control's default property settings render a user interface with a title (Log In), TextBox and Label controls for the username and password inputs, a Remember me next time CheckBox, and a Log In button. The appearances of these elements are all configurable through the Login control's numerous properties. Furthermore, additional user interface elements - such as a link to a page to create a new user account - can be added by setting a property or two.

Let's spend a few moments to gussy up the appearance of our Login control. Since the `Login.aspx` page already has text at the top of the page that says Login, the Login control's title is superfluous. Therefore, clear out the `TitleText` property value in order to remove the Login control's title.

The User Name: and Password: Labels to the left of the two TextBox controls can be customized through the `UserNameLabelText` and `PasswordLabelText` properties, respectively. Let's change the User Name: Label to read Username:. The Label and TextBox styles are configurable through the `LabelStyle` and `TextBoxStyle` properties, respectively.

The Remember me next time CheckBox's Text property can be set through the Login control's `RememberMeText property`, and its default checked state is configurable through the `RememberMeSet property` (which defaults to False). Go ahead and set the `RememberMeSet` property to True so that the Remember me next time CheckBox will be checked by default.

The Login control offers two properties for adjusting the layout of its user interface controls. The `TextLayout property` indicates whether the Username: and Password: Labels appear to the left of their

corresponding TextBoxes (the default), or above them. The `Orientation property` indicates whether the username and password inputs are situated vertically (one above the other) or horizontally. I am going to leave these two properties set to their defaults, but I encourage you to try setting these two properties to their non-default values to see the resulting effect.

> **NOTE**
>
> In the next section, Configuring the Login Control's Layout, we will look at using templates to define the precise layout of the Layout control's user interface elements.

Wrap up the Login control's property settings by setting the `CreateUserText` and `CreateUserUrl` properties to Not registered yet? Create an account! and `~/Membership/CreatingUserAccounts.aspx`, respectively. This adds a hyperlink to the Login control's interface pointing to the page we created in the preceding tutorial. The Login control's `HelpPageText` and `HelpPageUrl` properties and `PasswordRecoveryText` and `PasswordRecoveryUrl` properties work in the same manner, rendering links to a help page and a password recovery page.

After making these property changes, your Login control's declarative markup and appearance should look similar to that shown in Figure 5.

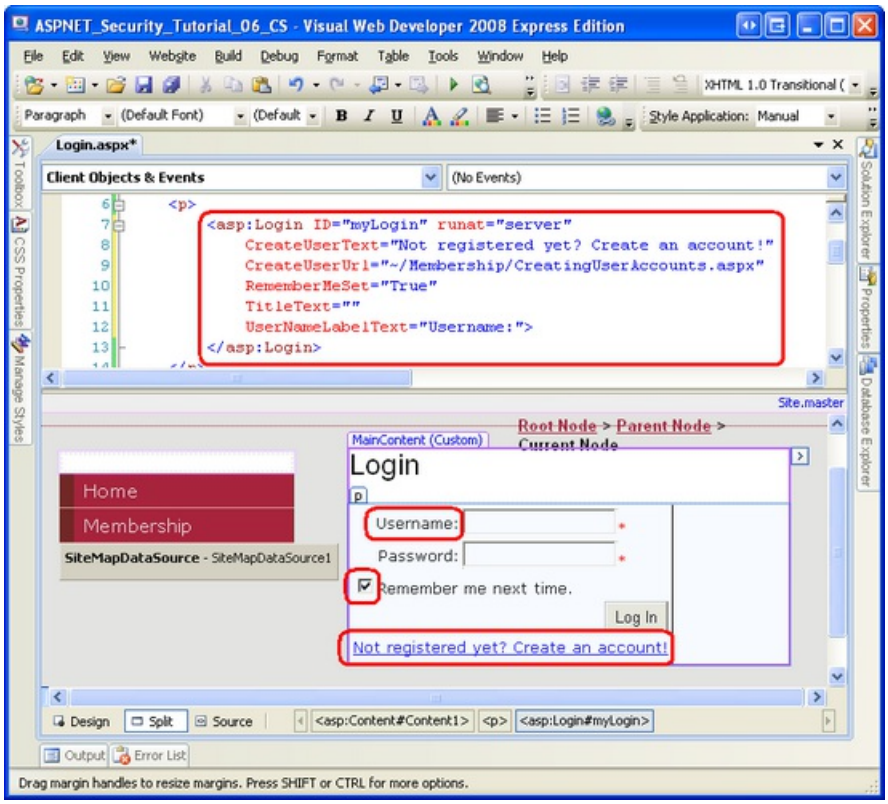


Figure 5: The Login Control's Properties' Values Dictate Its Appearance (Click to view full-size image)

Configuring the Login Control's Layout

The Login Web control's default user interface lays out the interface in an HTML `<table>`. But what if we need finer control over the rendered output? Maybe we want to replace the `<table>` with a series of `<div>` tags. Or what if our application requires additional credentials for authentication? Many financial websites, for instance, require users to supply not only a username and password, but also a Personal Identification Number (PIN), or other identifying information. Whatever the reasons may be, it is possible to convert the Login control into a template, from which we can explicitly define the interface's declarative markup.

We need to do two things in order to update the Login control to collect additional credentials:

1. Update the Login control's interface to include Web control(s) to collect the additional credentials.
2. Override the Login control's internal authentication logic so that a user is only authenticated if their username

and password is valid and their additional credentials are valid, too.

To accomplish the first task, we need to convert the Login control into a template and add the necessary Web controls. As for the second task, the Login control's authentication logic can be superseded by creating an event handler for the control's `Authenticate` event.

Let's update the Login control so that it prompts users for their username, password, and email address and only authenticates the user if the email address supplied matches their email address on file. We first need to convert the Login control's interface to a template. From the Login control's Smart Tag, choose the Convert to template option.

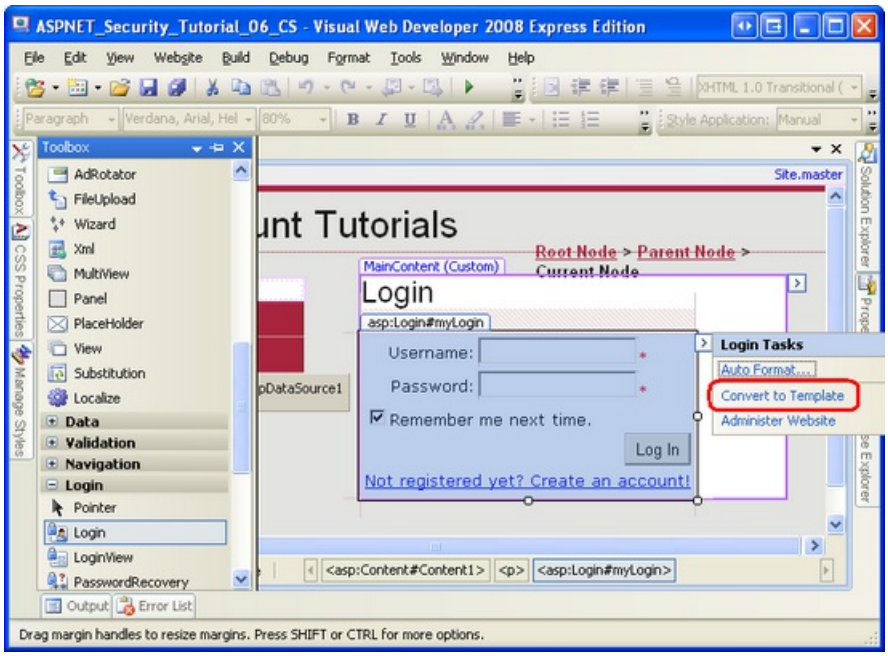


Figure 6: Convert the Login Control to a Template (Click to view full-size image)

> **NOTE**
>
> To revert the Login control to its pre-template version, click the Reset link from the control's Smart Tag.

Converting the Login control to a template adds a `LayoutTemplate` to the control's declarative markup with HTML elements and Web controls defining the user interface. As Figure 7 shows, converting the control to a template removes a number of properties from the Properties window, such as `TitleText`, `CreateUserUrl`, and so forth, since these property values are ignored when using a template.

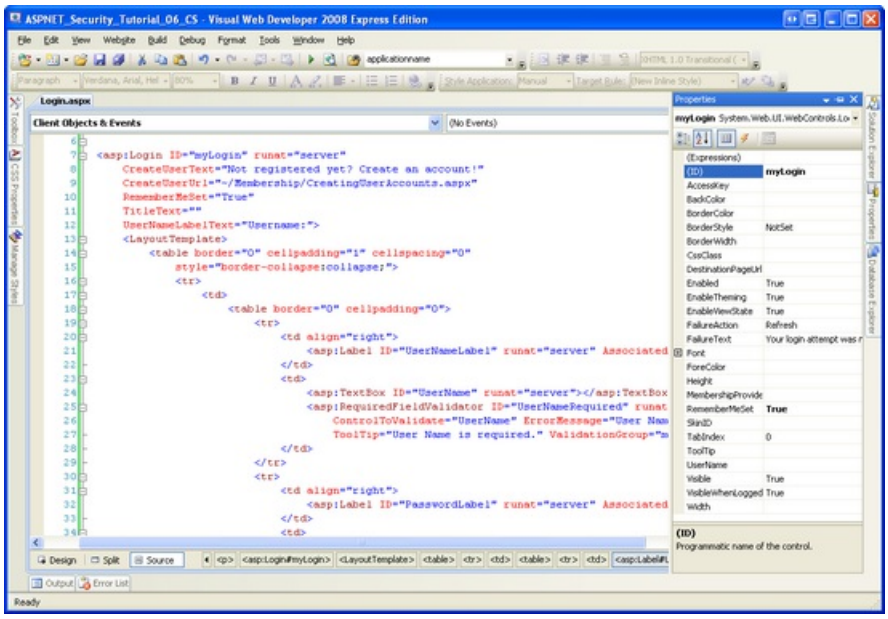


Figure 7: Fewer Properties are Available When the Login Control is Converted to a Template (Click to view full-size image)

The HTML markup in the `LayoutTemplate` may be modified as needed. Likewise, feel free to add any new Web controls to the template. However, it is important that Login control's core Web controls remain in the template and keep their assigned `ID` values. In particular, do not remove or rename the `UserName` or `Password` TextBoxes, the `RememberMe` CheckBox, the `LoginButton` Button, the `FailureText` Label, or the `RequiredFieldValidator` controls.

To collect the visitor's email address, we need to add a TextBox to the template. Add the following declarative markup between the table row (`<tr>`) that contains the `Password` TextBox and the table row that holds the Remember me next time CheckBox:

```
<tr>
 <td align="right">
 <asp:Label ID="EmailLabel" runat="server" AssociatedControlID="Email">Email:</asp:Label>
 </td>
 <td>
 <asp:TextBox ID="Email" runat="server"></asp:TextBox>
 <asp:RequiredFieldValidator ID="EmailRequired" runat="server"
 ControlToValidate="Email" ErrorMessage="Email is required."
 ToolTip="Email is required." ValidationGroup="myLogin">*</asp:RequiredFieldValidator>
 </td>
</tr>
```

After adding the `Email` TextBox, visit the page through a browser. As Figure 8 shows, the Login control's user interface now includes a third textbox.

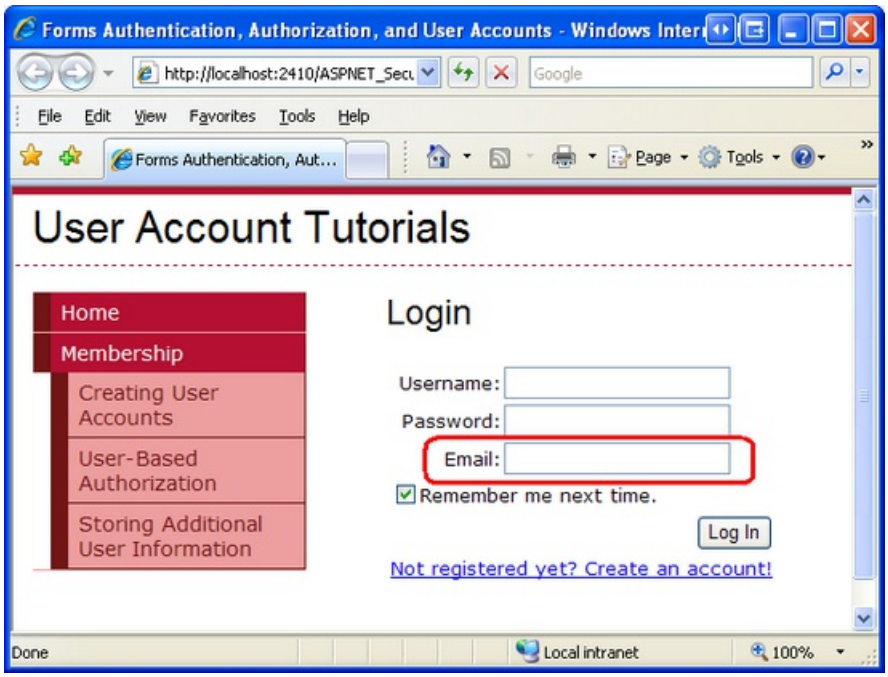


Figure 8: The Login Control Now Includes a Textbox for the User's Email Address (Click to view full-size image)

At this point, the Login control is still using the `Membership.ValidateUser` method to validate the supplied credentials. Correspondingly, the value entered into the `Email` TextBox has no bearing on whether the user can log in. In Step 3 we will look at how to override the Login control's authentication logic so that the credentials are only considered valid if the username and password are valid and the supplied email address matches up with the email address on file.

Step 3: Modifying the Login Control's Authentication Logic

When a visitor supplies her credentials and clicks the Log In button, a postback ensues and the Login control progresses through its authentication workflow. The workflow starts by raising the `LoggingIn` event. Any event handlers associated with this event may cancel the log in operation by setting the `e.Cancel` property to `true`.

If the log in operation is not cancelled, the workflow progresses by raising the `Authenticate` event. If there is an event handler for the `Authenticate` event, then it is responsible for determining whether the supplied credentials are valid or not. If no event handler is specified, the Login control uses the `Membership.ValidateUser` method to determine the validity of the credentials.

If the supplied credentials are valid, then the forms authentication ticket is created, the `LoggedIn` event is raised, and the user is redirected to the appropriate page. If, however, the credentials are deemed invalid, then the `LoginError` event is raised and a message is displayed informing the user that their credentials were invalid. By default, on failure the Login control simply sets its `FailureText` Label control's Text property to a failure message (Your login attempt was not successful. Please try again). However, if the Login control's `FailureAction` property is set to `RedirectToLoginPage`, then the Login control issues a `Response.Redirect` to the login page appending the querystring parameter `loginfailure=1` (which causes the Login control to display the failure message).

Figure 9 offers a flow chart of the authentication workflow.

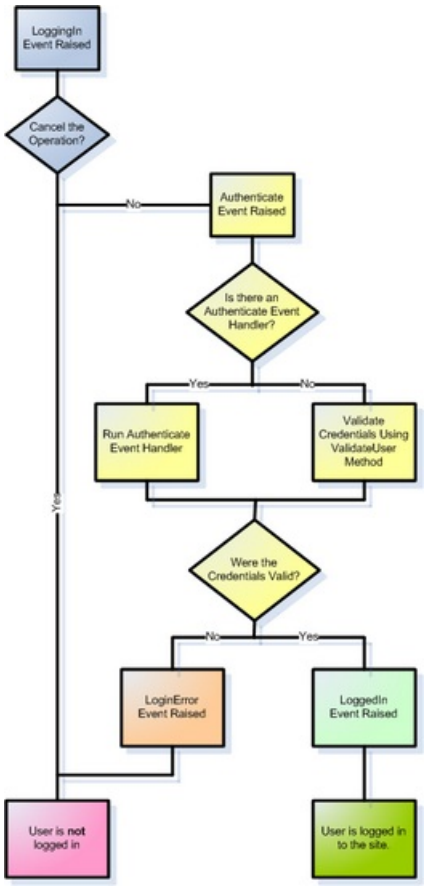


Figure 9: The Login Control's Authentication Workflow (Click to view full-size image)

> **NOTE**
> If you are wondering when you would use the `FailureAction`'s `RedirectToLogin` page option, consider the following scenario. Right now our `Site.master` master page currently has the text, Hello, stranger displayed in the left column when visited by an anonymous user, but imagine that we wanted to replace that text with a Login control. This would allow an anonymous user to log in from any page on the site, instead of requiring them to visit the login page directly. However, if a user was unable to log in via the Login control rendered by the master page, it might make sense to redirect them to the login page (`Login.aspx`) because that page likely includes additional instructions, links, and other help - such as links to create a new account or retrieve a lost password - that were not added to the master page.

Creating the `Authenticate` Event Handler

In order to plug in our custom authentication logic, we need to create an event handler for the Login control's `Authenticate` event. Creating an event handler for the `Authenticate` event will generate the following event handler definition:

```
protected void myLogin_Authenticate(object sender, AuthenticateEventArgs e)
{
}
```

As you can see, the `Authenticate` event handler is passed an object of type `AuthenticateEventArgs` as its second input parameter. The `AuthenticateEventArgs` class contains a Boolean property named `Authenticated` that is used to specify whether the supplied credentials are valid. Our task, then, is to write code here that determines whether the supplied credentials are valid or not, and to set the `e.Authenticate` property accordingly.

Determining and Validating the Supplied Credentials

Use the Login control's `UserName` and `Password` properties to determine the username and password credentials entered by the user. In order to determine the values entered into any additional Web controls (such as the

`Email` TextBox we added in the previous step), use `LoginControlID` `.FindControl` ("`controlID`") to obtain a programmatic reference to the Web control in the template whose `ID` property is equal to `controlID`. For example, to get a reference to the `Email` TextBox, use the following code:

```
TextBox EmailTextBox = myLogin.FindControl("Email") as TextBox;
```

In order to validate the user's credentials we need to do two things:

1. Ensure that the provided username and password are valid
2. Ensure that email address entered matches the email address on file for the user attempting to log in

To accomplish the first check we can simply use the `Membership.ValidateUser` method like we saw in Step 1. For the second check, we need to determine the user's email address so that we can compare it to the email address they entered into the TextBox control. To get information about a particular user, use the `Membership` class's `GetUser` method.

The `GetUser` method has a number of overloads. If used without passing in any parameters, it returns information about the currently logged in user. To get information about a particular user, call `GetUser` passing in their username. Either way, `GetUser` returns a `MembershipUser` object, which has properties like `UserName`, `Email`, `IsApproved`, `IsOnline`, and so on.

The following code implements these two checks. If both pass, then `e.Authenticate` is set to `true`, otherwise it is assigned `false`.

```
protected void myLogin_Authenticate(object sender, AuthenticateEventArgs e)
{
    // Get the email address entered
    TextBox EmailTextBox = myLogin.FindControl("Email") as TextBox;
    string email = EmailTextBox.Text.Trim();

    // Verify that the username/password pair is valid
    if (Membership.ValidateUser(myLogin.UserName, myLogin.Password))
    {
        // Username/password are valid, check email
        MembershipUser usrInfo = Membership.GetUser(myLogin.UserName);
        if (usrInfo != null && string.Compare(usrInfo.Email, email, true) == 0)
        {
            // Email matches, the credentials are valid
            e.Authenticated = true;
        }
        else
        {
            // Email address is invalid...
            e.Authenticated = false;
        }
    }
    else
    {
        // Username/password are not valid...
        e.Authenticated = false;
    }
}
```

With this code in place, attempt to log in as a valid user, entering the correct username, password, and email address. Try it again, but this time purposefully use an incorrect email address (see Figure 10). Finally, try it a third time using a non-existent username. In the first case you should be successfully logged on to the site, but in the last two cases you should see the Login control's invalid credentials message.

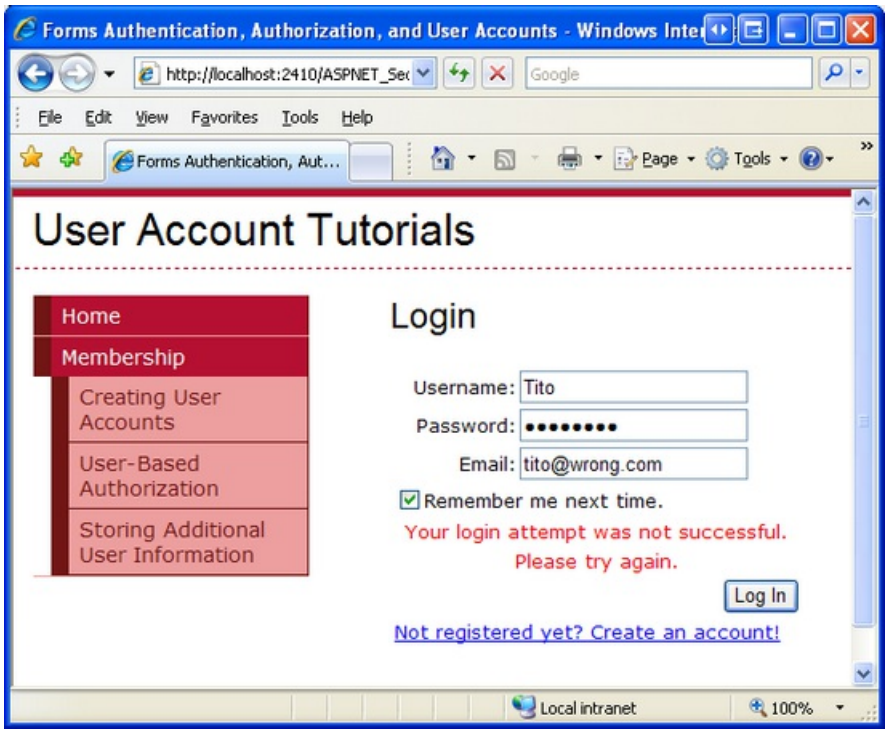


Figure 10: Tito Cannot Log In When Supplying an Incorrect Email Address (Click to view full-size image)

> **NOTE**
>
> As discussed in the How the Membership Framework Handles Invalid Login Attempts section in Step 1, when the `Membership.ValidateUser` method is called and passed invalid credentials, it keeps track of the invalid login attempt and locks out the user if they exceed a certain threshold of invalid attempts within a specified time window. Since our custom authentication logic calls the `ValidateUser` method, an incorrect password for a valid username will increment the invalid login attempt counter, but this counter is not incremented in the case where the username and password are valid, but the email address is incorrect. Chances are, this behavior is suitable, since it's unlikely that a hacker will know the username and password, but have to use brute force techniques to determine the user's email address.

Step 4: Improving the Login Control's Invalid Credentials Message

When a user attempts to log on with invalid credentials, the Login control displays a message explaining that the login attempt was unsuccessful. In particular, the control displays the message specified by its `FailureText` property, which has a default value of Your login attempt was not successful. Please try again.

Recall that there are many reasons why a user's credentials may be invalid:

- The username may not exist
- The username exists, but the password is invalid
- The username and password are valid, but the user is not yet approved
- The username and password are valid, but the user is locked out (most likely because they exceeded the number of invalid login attempts within the specified time frame)

And there may be other reasons when using custom authentication logic. For example, with the code we wrote in Step 3, the username and password may be valid, but the email address may be incorrect.

Regardless of why the credentials are invalid, the Login control displays the same error message. This lack of feedback can be confusing for a user whose account has not yet been approved or who has been locked out. With a little bit of work, though, we can have the Login control display a more appropriate message.

Whenever a user attempts to login with invalid credentials the Login control raises its `LoginError` event. Go ahead and create an event handler for this event, and add the following code:

```
protected void myLogin_LoginError(object sender, EventArgs e)
{
    // Determine why the user could not login...
    myLogin.FailureText = "Your login attempt was not successful. Please try again.";

    // Does there exist a User account for this user?
    MembershipUser usrInfo = Membership.GetUser(myLogin.UserName);
    if (usrInfo != null)
    {
        // Is this user locked out?
        if (usrInfo.IsLockedOut)
        {
            myLogin.FailureText = "Your account has been locked out because of too many invalid login
attempts. Please contact the administrator to have your account unlocked.";
        }
        else if (!usrInfo.IsApproved)
        {
            myLogin.FailureText = "Your account has not yet been approved. You cannot login until an
administrator has approved your account.";
        }
    }
}
```

The above code starts by setting the Login control's `FailureText` property to the default value (Your login attempt was not successful. Please try again). It then checks to see if the username supplied maps to an existing user account. If so, it consults the resulting `MembershipUser` object's `IsLockedOut` and `IsApproved` properties to determine if the account has been locked out or has not yet been approved. In either case, the `FailureText` property is updated to a corresponding value.

To test this code, purposefully attempt to log in as an existing user, but use an incorrect password. Do this five times in a row within a 10 minute time frame and the account will be locked out. As Figure 11 shows, subsequent login attempts will always fail (even with the correct password), but will now display the more descriptive Your account has been locked out because of too many invalid login attempts. Please contact the administrator to have your account unlocked message.

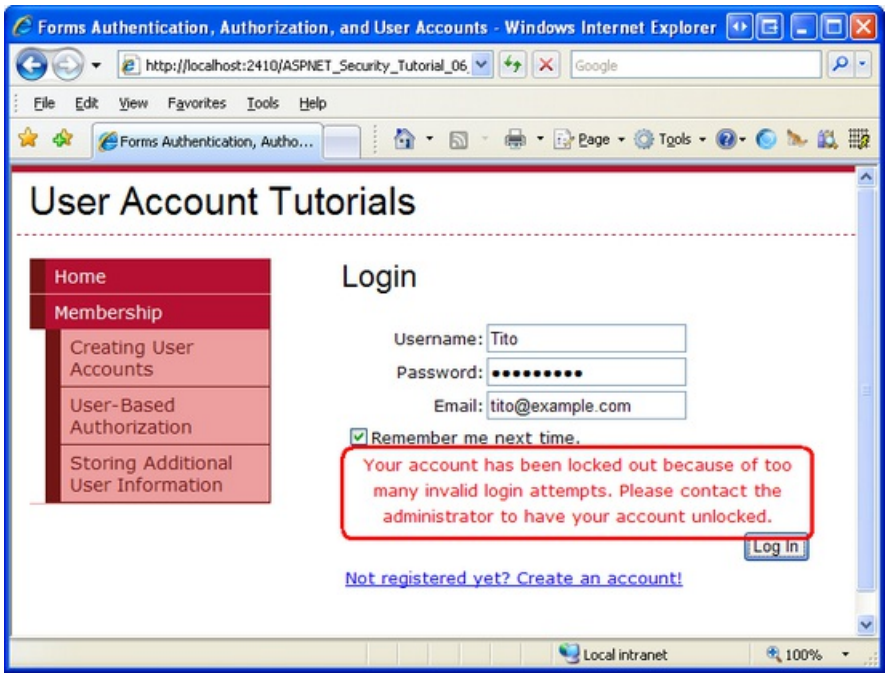


Figure 11: Tito Performed Too Many Invalid Login Attempts and Has Been Locked Out (Click to view full-size image)

Summary

Prior this tutorial, our login page validated the supplied credentials against a hard-coded list of username/password pairs. In this tutorial, we updated the page to validate credentials against the Membership framework. In Step 1 we looked at using the `Membership.ValidateUser` method programmatically. In Step 2 we replaced our manually created user interface and code with the Login control.

The Login control renders a standard login user interface and automatically validates the user's credentials against the Membership framework. Moreover, in the event of valid credentials, the Login control signs the user in via forms authentication. In short, a fully functional login user experience is available by simply dragging the Login control onto a page, no extra declarative markup or code necessary. What's more, the Login control is highly customizable, allowing for a fine degree of control over both the rendered user interface and authentication logic.

At this point visitors to our website can create a new user account and log into the site, but we have yet to look at restricting access to pages based on the authenticated user. Currently, any user, authenticated or anonymous, can view any page on our site. Along with controlling access to our site's pages on a user-by-user basis, we might have certain pages whose functionality depends on the user. The next tutorial looks at how to limit access and in-page functionality based on the logged in user.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Displaying Custom Messages to Locked Out and Non-Approved Users
- Examining ASP.NET 2.0's Membership, Roles, and Profile
- How To: Create an ASP.NET Login Page
- Login Control Technical Documentation
- Using the Login Controls

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Teresa Murphy and Michael Olivero. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

User-Based Authorization (C#)

2/20/2020 • 33 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> In this tutorial we will look at limiting access to pages and restricting page-level functionality through a variety of techniques.

Introduction

Most web applications that offer user accounts do so in part to restrict certain visitors from accessing certain pages within the site. In most online messageboard sites, for example, all users - anonymous and authenticated - are able to view the messageboard's posts, but only authenticated users can visit the web page to create a new post. And there may be administrative pages that are only accessible to a particular user (or a particular set of users). Moreover, page-level functionality can differ on a user-by-user basis. When viewing a list of posts, authenticated users are shown an interface for rating each post, whereas this interface is not available to anonymous visitors.

ASP.NET makes it easy to define user-based authorization rules. With just a bit of markup in `Web.config`, specific web pages or entire directories can be locked down so that they are only accessible to a specified subset of users. Page-level functionality can be turned on or off based on the currently logged in user through programmatic and declarative means.

In this tutorial we will look at limiting access to pages and restricting page-level functionality through a variety of techniques. Let's get started!

A Look at the URL Authorization Workflow

As discussed in the *An Overview of Forms Authentication* tutorial, when the ASP.NET runtime processes a request for an ASP.NET resource the request raises a number of events during its lifecycle. *HTTP Modules* are managed classes whose code is executed in response to a particular event in the request lifecycle. ASP.NET ships with a number of HTTP Modules that perform essential tasks behind the scenes.

One such HTTP Module is `FormsAuthenticationModule`. As discussed in previous tutorials, the primary function of the `FormsAuthenticationModule` is to determine the identity of the current request. This is accomplished by inspecting the forms authentication ticket, which is either located in a cookie or embedded within the URL. This identification takes place during the `AuthenticateRequest` event.

Another important HTTP Module is the `UrlAuthorizationModule`, which is raised in response to the `AuthorizeRequest` event (which happens after the `AuthenticateRequest` event). The `UrlAuthorizationModule` examines configuration markup in `Web.config` to determine whether the current identity has authority to visit the specified page. This process is referred to as *URL authorization*.

We'll examine the syntax for the URL authorization rules in Step 1, but first let's look at what the `UrlAuthorizationModule` does depending on whether the request is authorized or not. If the `UrlAuthorizationModule` determines that the request is authorized, then it does nothing, and the request continues through its lifecycle. However, if the request is *not* authorized, then the `UrlAuthorizationModule` aborts the lifecycle and instructs the `Response` object to return an HTTP 401 Unauthorized status. When using forms authentication this HTTP 401 status is never returned to the client because if the `FormsAuthenticationModule`

detects an HTTP 401 status is modifies it to an HTTP 302 Redirect to the login page.

Figure 1 illustrates the workflow of the ASP.NET pipeline, the `FormsAuthenticationModule`, and the `UrlAuthorizationModule` when an unauthorized request arrives. In particular, Figure 1 shows a request by an anonymous visitor for `ProtectedPage.aspx`, which is a page that denies access to anonymous users. Since the visitor is anonymous, the `UrlAuthorizationModule` aborts the request and returns an HTTP 401 Unauthorized status. The `FormsAuthenticationModule` then converts the 401 status into a 302 Redirect to login page. After the user is authenticated via the login page, he is redirected to `ProtectedPage.aspx`. This time the `FormsAuthenticationModule` identifies the user based on his authentication ticket. Now that the visitor is authenticated, the `UrlAuthorizationModule` permits access to the page.

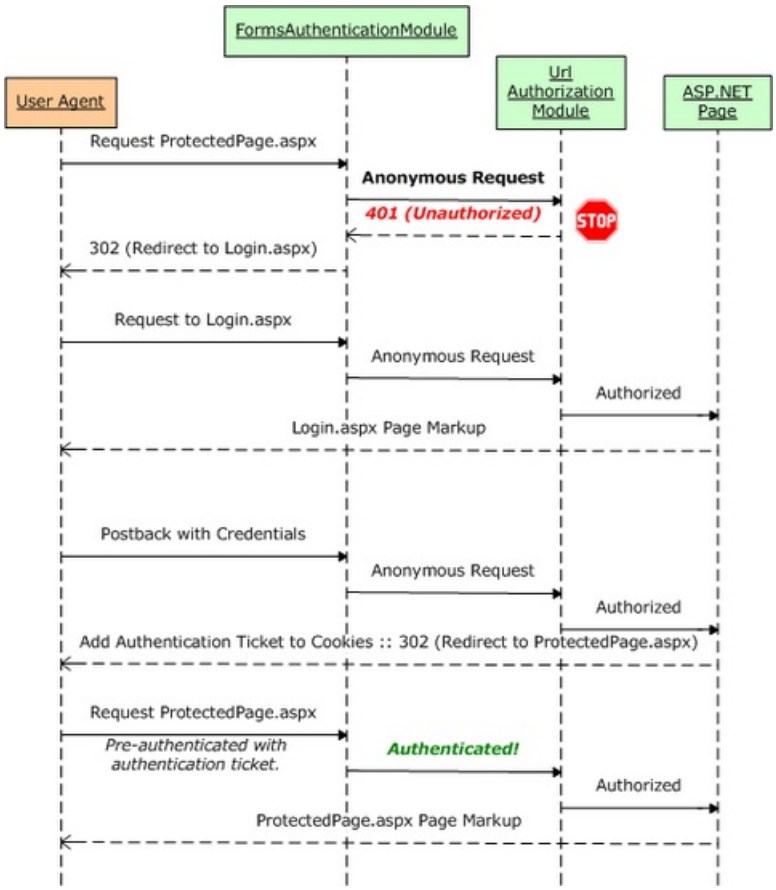


Figure 1: The Forms Authentication and URL Authorization Workflow (Click to view full-size image)

Figure 1 depicts the interaction that occurs when an anonymous visitor attempts to access a resource that is not available to anonymous users. In such a case, the anonymous visitor is redirected to the login page with the page she attempted to visit specified in the querystring. Once the user has successfully logged on, she will be automatically redirected back to the resource she was initially attempting to view.

When the unauthorized request is made by an anonymous user, this workflow is straightforward and is easy for the visitor to understand what has happened and why. But keep in mind that the `FormsAuthenticationModule` will redirect *any* unauthorized user to the login page, even if the request is made by an authenticated user. This can result in a confusing user experience if an authenticated user attempts to visit a page for which she lacks authority.

Imagine that our website had its URL authorization rules configured such that the ASP.NET page `OnlyTito.aspx` was accessibly only to Tito. Now, imagine that Sam visits the site, logs on, and then attempts to visit `OnlyTito.aspx`. The `UrlAuthorizationModule` will halt the request lifecycle and return an HTTP 401 Unauthorized status, which the `FormsAuthenticationModule` will detect and then redirect Sam to the login page. Since Sam has already logged in, though, she may wonder why she has been sent back to the login page. She might reason that her login credentials were lost somehow, or that she entered invalid credentials. If Sam reenters her credentials from the login page she will be logged on (again) and redirected to `OnlyTito.aspx`. The `UrlAuthorizationModule`

will detect that Sam cannot visit this page and she will be returned to the login page.

Figure 2 depicts this confusing workflow.

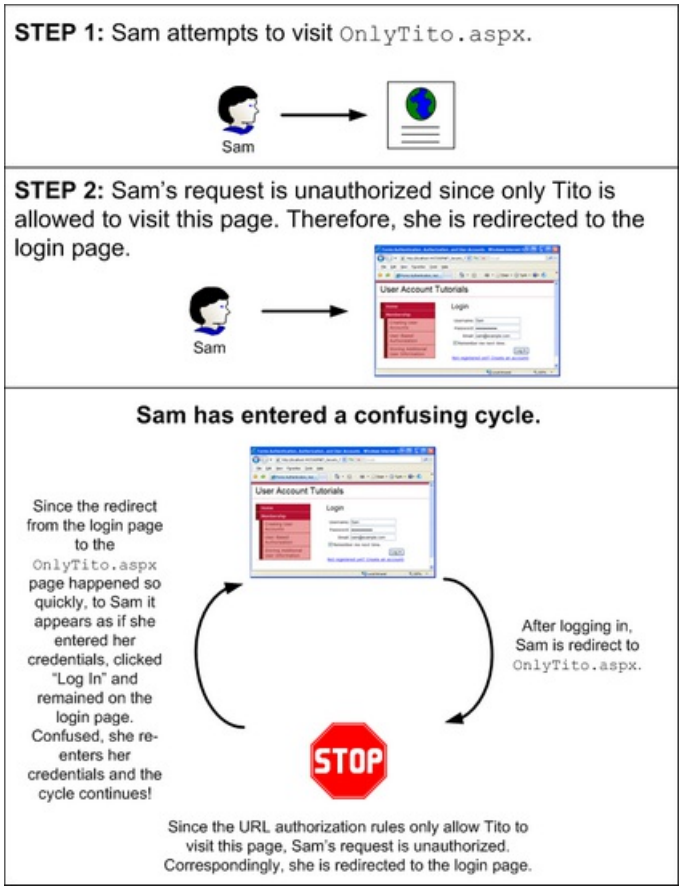


Figure 2: The Default Workflow Can Lead to a Confusing Cycle (Click to view full-size image)

The workflow illustrated in Figure 2 can quickly befuddle even the most computer savvy visitor. We will look at ways to prevent this confusing cycle in Step 2.

> **NOTE**
>
> ASP.NET uses two mechanisms to determine whether the current user can access a particular web page: URL authorization and file authorization. File authorization is implemented by the `FileAuthorizationModule`, which determines authority by consulting the requested file(s) ACLs. File authorization is most commonly used with Windows authentication because ACLs are permissions that apply to Windows accounts. When using forms authentication, all operating system- and file system-level requests are executed by the same Windows account, regardless of the user visiting the site. Since this tutorial series focuses on forms authentication, we will not be discussing file authorization.

The Scope of URL Authorization

The `UrlAuthorizationModule` is managed code that is part of the ASP.NET runtime. Prior to version 7 of Microsoft's Internet Information Services (IIS) web server, there was a distinct barrier between IIS's HTTP pipeline and the ASP.NET runtime's pipeline. In short, in IIS 6 and earlier, ASP.NET's `UrlAuthorizationModule` only executes when a request is delegated from IIS to the ASP.NET runtime. By default, IIS processes static content itself - like HTML pages and CSS, JavaScript, and image files - and only hands off requests to the ASP.NET runtime when a page with an extension of `.aspx`, `.asmx`, or `.ashx` is requested.

IIS 7, however, allows for integrated IIS and ASP.NET pipelines. With a few configuration settings you can setup IIS 7 to invoke the `UrlAuthorizationModule` for *all* requests, meaning that URL authorization rules can be defined for files of any type. Additionally, IIS 7 includes its own URL authorization engine. For more information on ASP.NET integration and IIS 7's native URL authorization functionality, see Understanding IIS7 URL Authorization. For a more in-depth look at ASP.NET and IIS 7 integration, pick up a copy of Shahram Khosravi's book,

Professional IIS 7 and ASP.NET Integrated Programming (ISBN: 978-0470152539).

In a nutshell, in versions prior to IIS 7, URL authorization rules are only applied to resources handled by the ASP.NET runtime. But with IIS 7 it is possible to use IIS's native URL authorization feature or to integrate ASP.NET's `UrlAuthorizationModule` into IIS's HTTP pipeline, thereby extending this functionality to all requests.

> **NOTE**
> There are some subtle yet important differences in how ASP.NET's `UrlAuthorizationModule` and IIS 7's URL authorization feature process the authorization rules. This tutorial does not examine IIS 7's URL authorization functionality or the differences in how it parses authorization rules compared to the `UrlAuthorizationModule`. For more information on these topics, refer to the IIS 7 documentation on MSDN or at www.iis.net.

Step 1: Defining URL Authorization Rules in `Web.config`

The `UrlAuthorizationModule` determines whether to grant or deny access to a requested resource for a particular identity based on the URL authorization rules defined in the application's configuration. The authorization rules are spelled out in the `<authorization>` element in the form of `<allow>` and `<deny>` child elements. Each `<allow>` and `<deny>` child element can specify:

- A particular user
- A comma-delimited list of users
- All anonymous users, denoted by a question mark (?)
- All users, denoted by an asterisk (*)

The following markup illustrates how to use the URL authorization rules to allow users Tito and Scott and deny all others:

```
<authorization>
 <allow users="Tito, Scott" />
 <deny users="*" />
</authorization>
```

The `<allow>` element defines what users are permitted - Tito and Scott - while the `<deny>` element instructs that *all* users are denied.

> **NOTE**
> The `<allow>` and `<deny>` elements can also specify authorization rules for roles. We will examine role-based authorization in a future tutorial.

The following setting grants access to anyone other than Sam (including anonymous visitors):

```
<authorization>
 <deny users="Sam" />
</authorization>
```

To allow only authenticated users, use the following configuration, which denies access to all anonymous users:

```
<authorization>
 <deny users="?" />
</authorization>
```

The authorization rules are defined within the `<system.web>` element in `Web.config` and apply to all of the ASP.NET resources in the web application. Oftentimes, an application has different authorization rules for different sections. For example, at an eCommerce site, all visitors may peruse the products, see product reviews, search the catalog, and so on. However, only authenticated users may reach the checkout or the pages to manage one's shipping history. Moreover, there may be portions of the site that are only accessible by select users, such as site administrators.

ASP.NET makes it easy to define different authorization rules for different files and folders in the site. The authorization rules specified in the root folder's `Web.config` file apply to all ASP.NET resources in the site. However, these default authorization settings can be overridden for a particular folder by adding a `Web.config` with an `<authorization>` section.

Let's update our website so that only authenticated users can visit the ASP.NET pages in the `Membership` folder. To accomplish this we need to add a `Web.config` file to the `Membership` folder and set its authorization settings to deny anonymous users. Right-click the `Membership` folder in the Solution Explorer, choose the Add New Item menu from the context menu, and add a new Web Configuration File named `Web.config`.

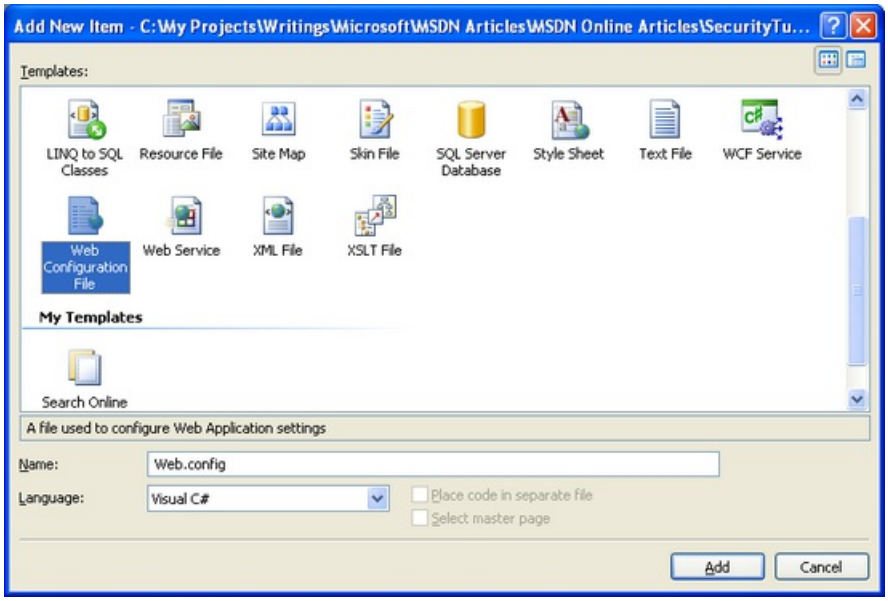


Figure 3: Add a `Web.config` File to the `Membership` Folder (Click to view full-size image)

At this point your project should contain two `Web.config` files: one in the root directory and one in the `Membership` folder.

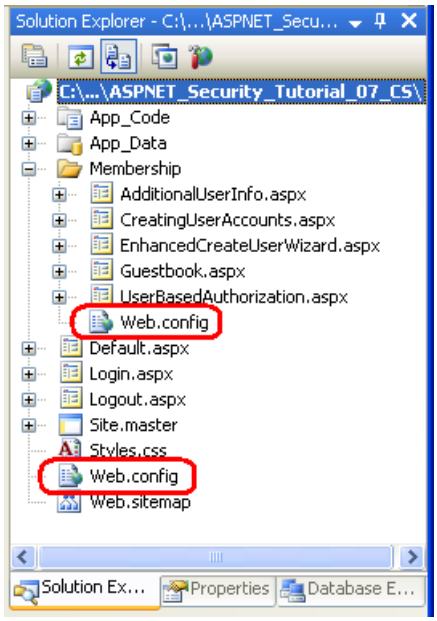


Figure 4: Your Application Should Now Contain Two `Web.config` Files (Click to view full-size image)

Update the configuration file in the `Membership` folder so that it prohibits access to anonymous users.

```
<?xml version="1.0"?>
<configuration>
 <system.web>
 <authorization>
 <deny users="?" />
 </authorization>
 </system.web>
</configuration>
```

That's all there is to it!

To test out this change, visit the homepage in a browser and make sure you are logged out. Since the default behavior of an ASP.NET application is to allow all visitors, and since we didn't make any authorization modifications to the root directory's `Web.config` file, we are able to visit the files in the root directory as an anonymous visitor.

Click on the Creating User Accounts link found in the left column. This will take you to the `~/Membership/CreatingUserAccounts.aspx`. Since the `Web.config` file in the `Membership` folder defines authorization rules to prohibit anonymous access, the `UrlAuthorizationModule` aborts the request and returns an HTTP 401 Unauthorized status. The `FormsAuthenticationModule` modifies this to a 302 Redirect status, sending us to the login page. Note that the page we were attempting to access (`CreatingUserAccounts.aspx`) is passed to the login page via the `ReturnUrl` querystring parameter.

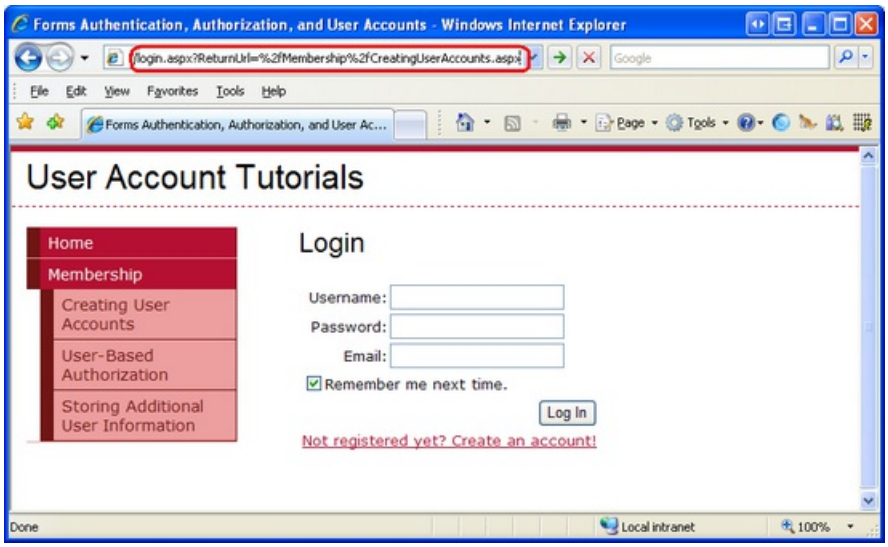


Figure 5: Since the URL Authorization Rules Prohibit Anonymous Access, We are Redirected to the Login Page (Click to view full-size image)

Upon successfully logging in, we are redirected to the `CreatingUserAccounts.aspx` page. This time the `UrlAuthorizationModule` permits access to the page because we are no longer anonymous.

Applying URL Authorization Rules to a Specific Location

The authorization settings defined in the `<system.web>` section of `Web.config` apply to all of the ASP.NET resources in that directory and its subdirectories (until otherwise overridden by another `Web.config` file). In some cases, though, we may want all ASP.NET resources in a given directory to have a particular authorization configuration except for one or two specific pages. This can be achieved by adding a `<location>` element in `Web.config`, pointing it to the file whose authorization rules differ, and defining its unique authorization rules therein.

To illustrate using the `<location>` element to override the configuration settings for a specific resource, let's customize the authorization settings so that only Tito can visit `CreatingUserAccounts.aspx`. To accomplish this, add a `<location>` element to the `Membership` folder's `Web.config` file and update its markup so that it looks like the following:

```
<?xml version="1.0"?>
<configuration>
 <system.web>
 <authorization>
 <deny users="?" />
 </authorization>
 </system.web>

 <location path="CreatingUserAccounts.aspx">
 <system.web>
 <authorization>
 <allow users="Tito" />
 <deny users="*" />
 </authorization>
 </system.web>
 </location>
</configuration>
```

The `<authorization>` element in `<system.web>` defines the default URL authorization rules for ASP.NET resources in the `Membership` folder and its subfolders. The `<location>` element allows us to override these rules for a particular resource. In the above markup the `<location>` element references the `CreatingUserAccounts.aspx` page and specifies its authorization rules such as to allow Tito, but deny everyone else.

To test out this authorization change, start by visiting the website as an anonymous user. If you attempt to visit any page in the `Membership` folder, such as `UserBasedAuthorization.aspx`, the `UrlAuthorizationModule` will deny the request and you will be redirected to the login page. After logging in as, say, Scott, you can visit any page in the `Membership` folder *except* for `CreatingUserAccounts.aspx`. Attempting to visit `CreatingUserAccounts.aspx` logged on as anyone but Tito will result in an unauthorized access attempt, redirecting you back to the login page.

> **NOTE**
> The `<location>` element must appear outside of the configuration's `<system.web>` element. You need to use a separate `<location>` element for each resource whose authorization settings you want to override.

A Look at How the `UrlAuthorizationModule` Uses the Authorization Rules to Grant or Deny Access

The `UrlAuthorizationModule` determines whether to authorize a particular identity for a particular URL by analyzing the URL authorization rules one at a time, starting from the first one and working its way down. As soon as a match is found, the user is granted or denied access, depending on if the match was found in an `<allow>` or `<deny>` element. **If no match is found, the user is granted access.** Consequently, if you want to restrict access, it is imperative that you use a `<deny>` element as the last element in the URL authorization configuration. **If you omit a `<deny>` element, all users will be granted access.**

To better understand the process used by the `UrlAuthorizationModule` to determine authority, consider the example URL authorization rules we looked at earlier in this step. The first rule is an `<allow>` element that allows access to Tito and Scott. The second rules is a `<deny>` element that denies access to everyone. If an anonymous user visits, the `UrlAuthorizationModule` starts by asking, Is anonymous either Scott or Tito? The answer, obviously, is No, so it proceeds to the second rule. Is anonymous in the set of everybody? Since the answer here is Yes, the `<deny>` rule is put in effect and the visitor is redirected to the login page. Similarly, if Jisun is visiting, the `UrlAuthorizationModule` starts by asking, Is Jisun either Scott or Tito? Since she is not, the

`UrlAuthorizationModule` proceeds to the second question, Is Jisun in the set of everybody? She is, so she, too, is denied access. Finally, if Tito visits, the first question posed by the `UrlAuthorizationModule` is an affirmative answer, so Tito is granted access.

Since the `UrlAuthorizationModule` processes the authorization rules from the top down, stopping at any match, it is important to have the more specific rules come before the less specific ones. That is, to define authorization rules that forbid Jisun and anonymous users, but allow all other authenticated users, you would start with the most specific rule - the one impacting Jisun - and then proceed to the less specific rules - those allowing all other authenticated users, but denying all anonymous users. The following URL authorization rules implements this policy by first denying Jisun, and then denying any anonymous user. Any authenticated user other than Jisun will be granted access because neither of these `<deny>` statements will match.

```
<authorization>
 <deny users="Jisun" />
 <deny users="?" />
</authorization>
```

Step 2: Fixing the Workflow for Unauthorized, Authenticated Users

As we discussed earlier in this tutorial in the A Look at the URL Authorization Workflow section, anytime an unauthorized request transpires, the `UrlAuthorizationModule` aborts the request and returns an HTTP 401 Unauthorized status. This 401 status is modified by the `FormsAuthenticationModule` into a 302 Redirect status that sends the user to the login page. This workflow occurs on any unauthorized request, even if the user is authenticated.

Returning an authenticated user to the login page is likely to confuse them since they have already logged into the system. With a little bit of work we can improve this workflow by redirecting authenticated users who make unauthorized requests to a page that explains that they have attempted to access a restricted page.

Start by creating a new ASP.NET page in the web application's root folder named `UnauthorizedAccess.aspx`; don't forget to associate this page with the `Site.master` master page. After creating this page, remove the Content control that references the `LoginContent` ContentPlaceHolder so that the master page's default content will be displayed. Next, add a message that explains the situation, namely that the user attempted to access a protected resource. After adding such a message, the `UnauthorizedAccess.aspx` page's declarative markup should look similar to the following:

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true"
CodeFile="UnauthorizedAccess.aspx.cs" Inherits="UnauthorizedAccess"
Title="Untitled Page" %>

<asp:Content ID="Content1" ContentPlaceHolderID="MainContent"
Runat="Server">
 <h2>Unauthorized Access</h2>
 <p>
 You have attempted to access a page that you are not authorized to view.
 </p>
 <p>
 If you have any questions, please contact the site administrator.
 </p>
</asp:Content>
```

We now need to alter the workflow so that if an unauthorized request is performed by an authenticated user they are sent to the `UnauthorizedAccess.aspx` page instead of the login page. The logic that redirects unauthorized requests to the login page is buried within a private method of the `FormsAuthenticationModule` class, so we cannot customize this behavior. What we can do, however, is add our own logic to the login page

that redirects the user to `UnauthorizedAccess.aspx`, if needed.

When the `FormsAuthenticationModule` redirects an unauthorized visitor to the login page it appends the requested, unauthorized URL to the querystring with the name `ReturnUrl`. For example, if an unauthorized user attempted to visit `OnlyTito.aspx`, the `FormsAuthenticationModule` would redirect them to `Login.aspx?ReturnUrl=OnlyTito.aspx`. Therefore, if the login page is reached by an authenticated user with a querystring that includes the `ReturnUrl` parameter, then we know that this unauthenticated user just attempted to visit a page she is not authorized to view. In such a case, we want to redirect her to `UnauthorizedAccess.aspx`.

To accomplish this, add the following code to the login page's `Page_Load` event handler:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        if (Request.IsAuthenticated && !string.IsNullOrEmpty(Request.QueryString["ReturnUrl"]))
        // This is an unauthorized, authenticated request...
        Response.Redirect("~/UnauthorizedAccess.aspx");
    }
}
```

The above code redirects authenticated, unauthorized users to the `UnauthorizedAccess.aspx` page. To see this logic in action, visit the site as an anonymous visitor and click on the Creating User Accounts link in the left column. This will take you to the `~/Membership/CreatingUserAccounts.aspx` page, which in Step 1 we configured to only permit access to Tito. Since anonymous users are prohibited, the `FormsAuthenticationModule` redirects us back to the login page.

At this point we are anonymous, so `Request.IsAuthenticated` returns `false` and we are not redirected to `UnauthorizedAccess.aspx`. Instead, the login page is displayed. Log in as a user other than Tito, such as Bruce. After entering the appropriate credentials, the login page redirects us back to `~/Membership/CreatingUserAccounts.aspx`. However, since this page is only accessible to Tito, we are unauthorized to view it and are promptly returned to the login page. This time, however, `Request.IsAuthenticated` returns `true` (and the `ReturnUrl` querystring parameter exists), so we are redirected to the `UnauthorizedAccess.aspx` page.

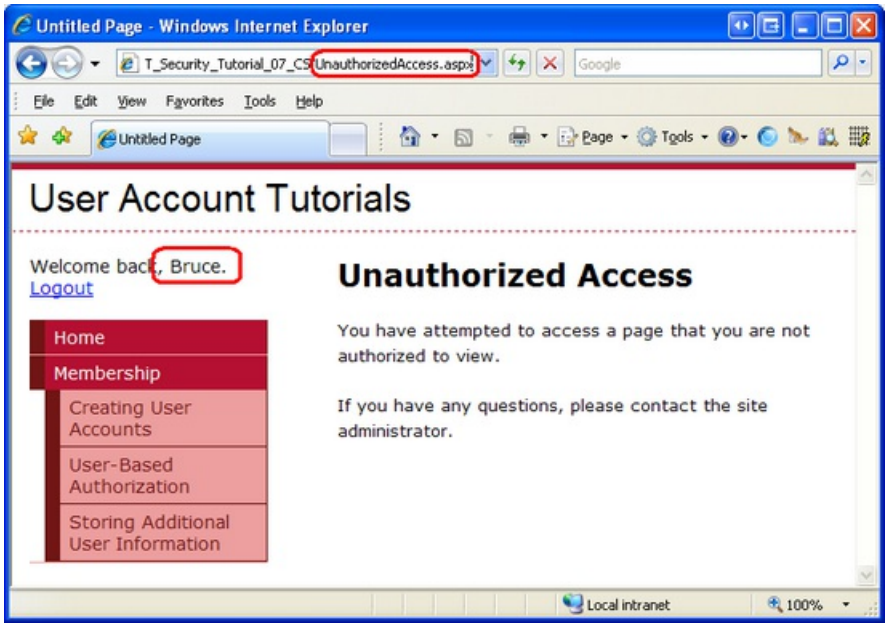


Figure 6: Authenticated, Unauthorized Users are Redirected to `UnauthorizedAccess.aspx` (Click to view full-size image)

This customized workflow presents a more sensible and straightforward user experience by short circuiting the

cycle depicted in Figure 2.

Step 3: Limiting Functionality Based on the Currently Logged In User

URL authorization makes it easy to specify coarse authorization rules. As we saw in Step 1, with URL authorization we can succinctly state what identities are permitted and which ones are denied from viewing a particular page or all pages in a folder. In certain scenarios, however, we may want to allow all users to visit a page, but limit the page's functionality based on the user visiting it.

Consider the case of an eCommerce website that allows authenticated visitors to review their products. When an anonymous user visits a product's page, they would see just the product information and would not be given the opportunity to leave a review. However, an authenticated user visiting the same page would see the reviewing interface. If the authenticated user had not yet reviewed this product, the interface would enable them to submit a review; otherwise it would show them their previously-submitted review. To take this scenario a step further, the product page might show additional information and offer extended features for those users that work for the eCommerce company. For example, the product page might list the inventory in stock and include options to edit the product's price and description when visited by an employee.

Such fine grain authorization rules can be implemented either declaratively or programmatically (or through some combination of the two). In the next section we will see how to implement fine grain authorization via the LoginView control. Following that, we will explore programmatic techniques. Before we can look at applying fine grain authorization rules, however, we first need to create a page whose functionality depends on the user visiting it.

Let's create a page that lists the files in a particular directory within a GridView. Along with listing each file's name, size, and other information, the GridView will include two columns of LinkButtons: one titled View and one titled Delete. If the View LinkButton is clicked, the contents of the selected file will be displayed; if the Delete LinkButton is clicked, the file will be deleted. Let's initially create this page such that its view and delete functionality is available to all users. In the Using the LoginView Control and Programmatically Limiting Functionality sections we will see how to enable or disable these features based on the user visiting the page.

> **NOTE**
>
> The ASP.NET page we are about to build uses a GridView control to display a list of files. Since this tutorial series focuses on forms authentication, authorization, user accounts, and roles, I do not want to spend too much time discussing the inner workings of the GridView control. While this tutorial provides specific step-by-step instructions for setting up this page, it does not delve into the details of why certain choices were made, or what effect particular properties have on the rendered output. For a thorough examination of the GridView control, consult my *Working with Data in ASP.NET 2.0* tutorial series.

Start by opening the `UserBasedAuthorization.aspx` file in the `Membership` folder and adding a GridView control to the page named `FilesGrid`. From the GridView's Smart Tag, click the Edit Columns link to launch the Fields dialog box. From here, uncheck the Auto-generate fields checkbox in the lower left corner. Next, add a Select button, a Delete button, and two BoundFields from the upper left corner (the Select and Delete buttons can be found under the CommandField type). Set the Select button's `SelectText` property to View and the first BoundField's `HeaderText` and `DataField` properties to Name. Set the second BoundField's `HeaderText` property to Size in Bytes, its `DataField` property to Length, its `DataFormatString` property to {0:N0} and its `HtmlEncode` property to False.

After configuring the GridView's columns, click OK to close the Fields dialog box. From the Properties window, set the GridView's `DataKeyNames` property to `FullName`. At this point the GridView's declarative markup should look like the following:

```
<asp:GridView ID="FilesGrid" DataKeyNames="FullName" runat="server" AutoGenerateColumns="False">
 <Columns>
 <asp:CommandField SelectText="View" ShowSelectButton="True"/>
 <asp:CommandField ShowDeleteButton="True" />
 <asp:BoundField DataField="Name" HeaderText="Name" />
 <asp:BoundField DataField="Length" DataFormatString="{0:N0}"
 HeaderText="Size in Bytes" HtmlEncode="False" />
 </Columns>
</asp:GridView>
```

With the GridView's markup created, we're ready to write the code that will retrieve the files in a particular directory and bind them to the GridView. Add the following code to the page's `Page_Load` event handler:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        string appPath = Request.PhysicalApplicationPath;
        DirectoryInfo dirInfo = new DirectoryInfo(appPath);

        FileInfo[] files = dirInfo.GetFiles();

        FilesGrid.DataSource = files;
        FilesGrid.DataBind();
    }
}
```

The above code uses the `DirectoryInfo` class to obtain a list of the files in the application's root folder. The `GetFiles()` method returns all of the files in the directory as an array of `FileInfo` objects, which is then bound to the GridView. The `FileInfo` object has an assortment of properties, such as `Name`, `Length`, and `IsReadOnly`, among others. As you can see from its declarative markup, the GridView displays just the `Name` and `Length` properties.

NOTE

The `DirectoryInfo` and `FileInfo` classes are found in the `System.IO` namespace. Therefore, you will either need to preface these class names with their namespace names or have the namespace imported into the class file (via `using System.IO`).

Take a moment to visit this page through a browser. It will display the list of files residing in the application's root directory. Clicking any of the View or Delete LinkButtons will cause a postback, but no action will occur because we've yet to create the necessary event handlers.

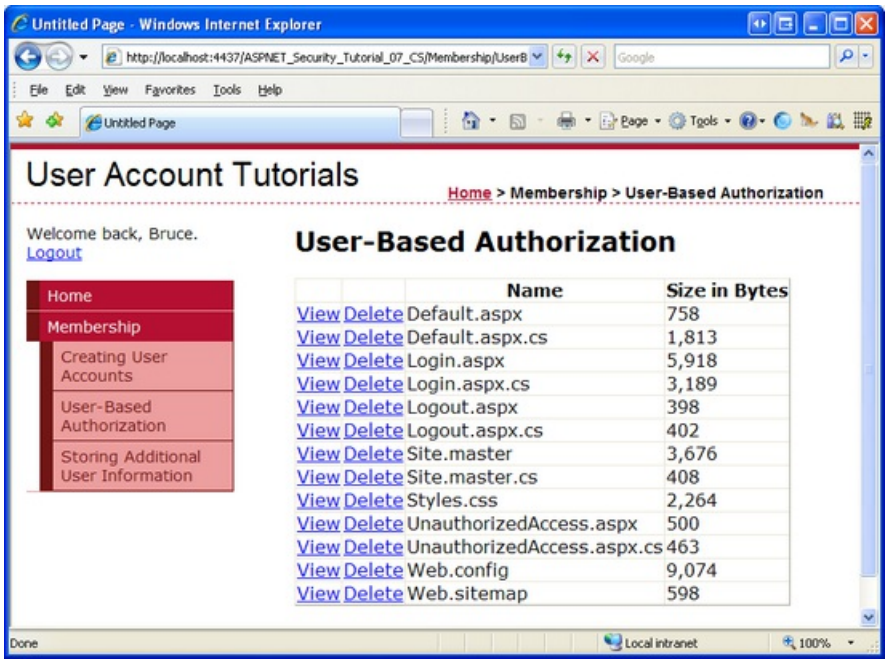


Figure 7: The GridView Lists the Files in the Web Application's Root Directory (Click to view full-size image)

We need a means to display the contents of the selected file. Return to Visual Studio and add a TextBox named `FileContents` above the GridView. Set its `TextMode` property to `MultiLine` and its `Columns` and `Rows` properties to 95% and 10, respectively.

```
<asp:TextBox ID="FileContents" runat="server" Rows="10"
TextMode="MultiLine" Width="95%"></asp:TextBox>
```

Next, create an event handler for the GridView's `SelectedIndexChanged` event and add the following code:

```
protected void FilesGrid_SelectedIndexChanged(object sender, EventArgs e)
{
    // Open the file and display it
    string fullFileName = FilesGrid.SelectedValue.ToString();
    string contents = File.ReadAllText(fullFileName);
    FileContents.Text = contents;
}
```

This code uses the GridView's `SelectedValue` property to determine the full file name of the selected file. Internally, the `DataKeys` collection is referenced in order to obtain the `SelectedValue`, so it is imperative that you set the GridView's `DataKeyNames` property to Name, as described earlier in this step. The `File` class is used to read the selected file's contents into a string, which is then assigned to the `FileContents` TextBox's `Text` property, thereby displaying the contents of the selected file on the page.

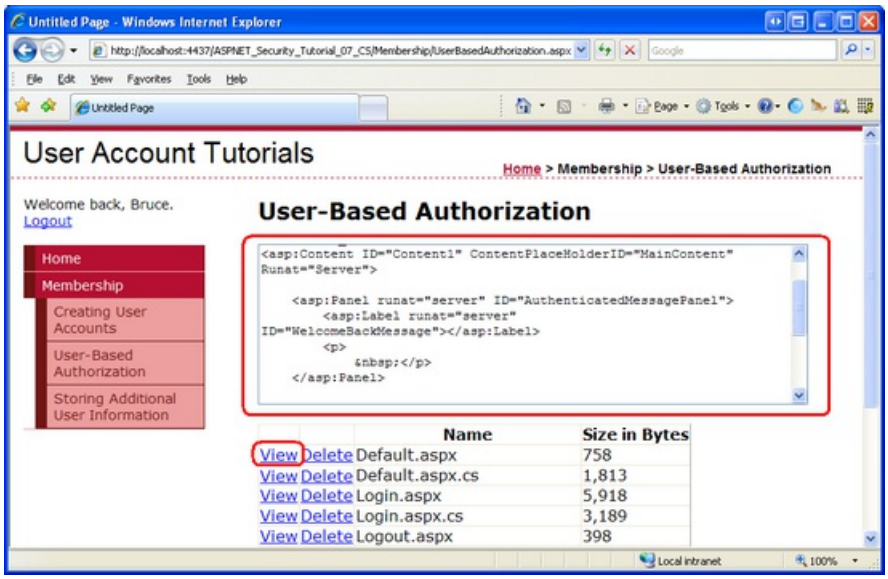


Figure 8: The Selected File's Contents are Displayed in the TextBox (Click to view full-size image)

> **NOTE**
>
> If you view the contents of a file that contains HTML markup, and then attempt to view or delete a file, you will receive an `HttpRequestValidationException` error. This occurs because on postback the TextBox's contents are sent back to the web server. By default, ASP.NET raises an `HttpRequestValidationException` error whenever potentially dangerous postback content, such as HTML markup, is detected. To disable this error from occurring, turn off request validation for the page by adding `ValidateRequest="false"` to the `@Page` directive. For more information on the benefits of request validation as well as what precautions you should take when disabling it, read Request Validation - Preventing Script Attacks.

Finally, add an event handler with the following code for the GridView's `RowDeleting` event:

```
protected void FilesGrid_RowDeleting(object sender, GridViewDeleteEventArgs e)
{
    string fullFileName = FilesGrid.DataKeys[e.RowIndex].Value.ToString();
    FileContents.Text = string.Format("You have opted to delete {0}.", fullFileName);

    // To actually delete the file, uncomment the following line
    // File.Delete(fullFileName);
}
```

The code simply displays the full name of the file to delete in the `FileContents` TextBox *without* actually deleting the file.

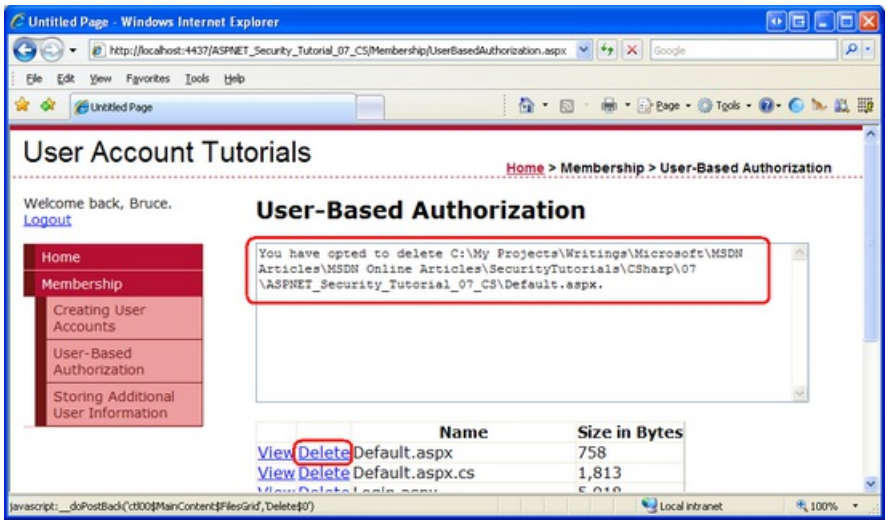


Figure 9: Clicking the Delete Button Does Not Actually Delete the File (Click to view full-size image)

In Step 1 we configured the URL authorization rules to prohibit anonymous users from viewing the pages in the `Membership` folder. In order to better exhibit fine grain authentication, let's allow anonymous users to visit the `UserBasedAuthorization.aspx` page, but with limited functionality. To open this page up to be accessed by all users, add the following `<location>` element to the `Web.config` file in the `Membership` folder:

```
<location path="UserBasedAuthorization.aspx">
 <system.web>
 <authorization>
 <allow users="*" />
 </authorization>
 </system.web>
</location>
```

After adding this `<location>` element, test the new URL authorization rules by logging out of the site. As an anonymous user you should be permitted to visit the `UserBasedAuthorization.aspx` page.

Currently, any authenticated or anonymous user can visit the `UserBasedAuthorization.aspx` page and view or delete files. Let's make it so that only authenticated users can view contents of a file and only Tito can delete a file. Such fine grain authorization rules can be applied declaratively, programmatically, or through a combination of both methods. Let's use the declarative approach to limit who can view the contents of a file; we'll use the programmatic approach to limit who can delete a file.

Using the LoginView Control

As we've seen in past tutorials, the LoginView control is useful for displaying different interfaces for authenticated and anonymous users, and offers an easy way to hide functionality that is not accessible to anonymous users. Since anonymous users cannot view or delete files, we only need to show the `FileContents` TextBox when the page is visited by an authenticated user. To achieve this, add a LoginView control to the page, name it `LoginViewForFileContentsTextBox`, and move the `FileContents` TextBox's declarative markup into the LoginView control's `LoggedInTemplate`.

```
<asp:LoginView ID=" LoginViewForFileContentsTextBox " runat="server">
 <LoggedInTemplate>
 <p>
 <asp:TextBox ID="FileContents" runat="server" Rows="10"
 TextMode="MultiLine" Width="95%"></asp:TextBox>
 </p>
 </LoggedInTemplate>
</asp:LoginView>
```

The Web controls in the LoginView's templates are no longer directly accessible from the code-behind class. For example, the `FilesGrid` GridView's `SelectedIndexChanged` and `RowDeleting` event handlers currently reference the `FileContents` TextBox control with code like:

```
FileContents.Text = text;
```

However, this code is no longer valid. By moving the `FileContents` TextBox into the `LoggedInTemplate` the TextBox cannot be directly accessed. Instead, we must use the `FindControl("controlId")` method to programmatically reference the control. Update the `FilesGrid` event handlers to reference the TextBox like so:

```
TextBox FileContentsTextBox = LoginViewForFileContentsTextBox.FindControl("FileContents") as TextBox;
FileContentsTextBox.Text = text;
```

After moving the TextBox to the LoginView's `LoggedInTemplate` and updating the page's code to reference the TextBox using the `FindControl("controlId")` pattern, visit the page as an anonymous user. As Figure 10 shows, the `FileContents` TextBox is not displayed. However, the View LinkButton is still displayed.

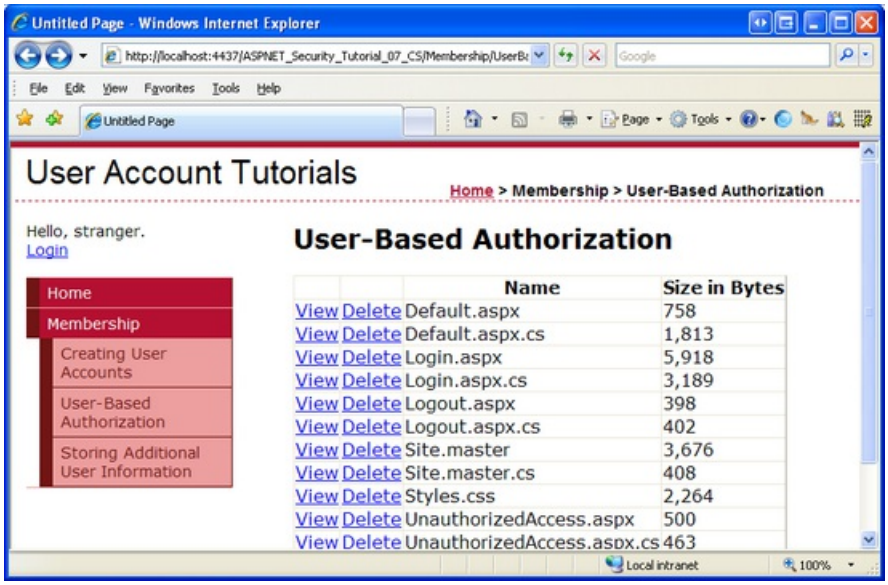


Figure 10: The LoginView Control Only Renders the `FileContents` TextBox for Authenticated Users (Click to view full-size image)

One way to hide the View button for anonymous users is to convert the GridView field into a TemplateField. This will generate a template that contains the declarative markup for the View LinkButton. We can then add a LoginView control to the TemplateField and place the LinkButton within the LoginView's `LoggedInTemplate`, thereby hiding the View button from anonymous visitors. To accomplish this, click on the Edit Columns link from the GridView's Smart Tag to launch the Fields dialog box. Next, select the Select button from the list in the lower left corner and then click the Convert this field to a TemplateField link. Doing so will modify the field's declarative markup from:

```
<asp:CommandField SelectText="View" ShowSelectButton="True"/>
```

To:

```
<asp:TemplateField ShowHeader="False">
 <ItemTemplate>
 <asp:LinkButton ID="LinkButton1" runat="server" CausesValidation="False"
 CommandName="Select" Text="View"></asp:LinkButton>
 </ItemTemplate>
</asp:TemplateField>
```

At this point, we can add a LoginView to the TemplateField. The following markup displays the View LinkButton only for authenticated users.

```
<asp:TemplateField ShowHeader="False">
 <ItemTemplate>
 <asp:LoginView ID="LoginView1" runat="server">
 <LoggedInTemplate>
 <asp:LinkButton ID="LinkButton1" runat="server" CausesValidation="False"
 CommandName="Select" Text="View"></asp:LinkButton>
 </LoggedInTemplate>
 </asp:LoginView>
 </ItemTemplate>
</asp:TemplateField>
```

As Figure 11 shows, the end result is not that pretty as the View column is still displayed even though the View LinkButtons within the column are hidden. We will look at how to hide the entire GridView column (and not just the LinkButton) in the next section.

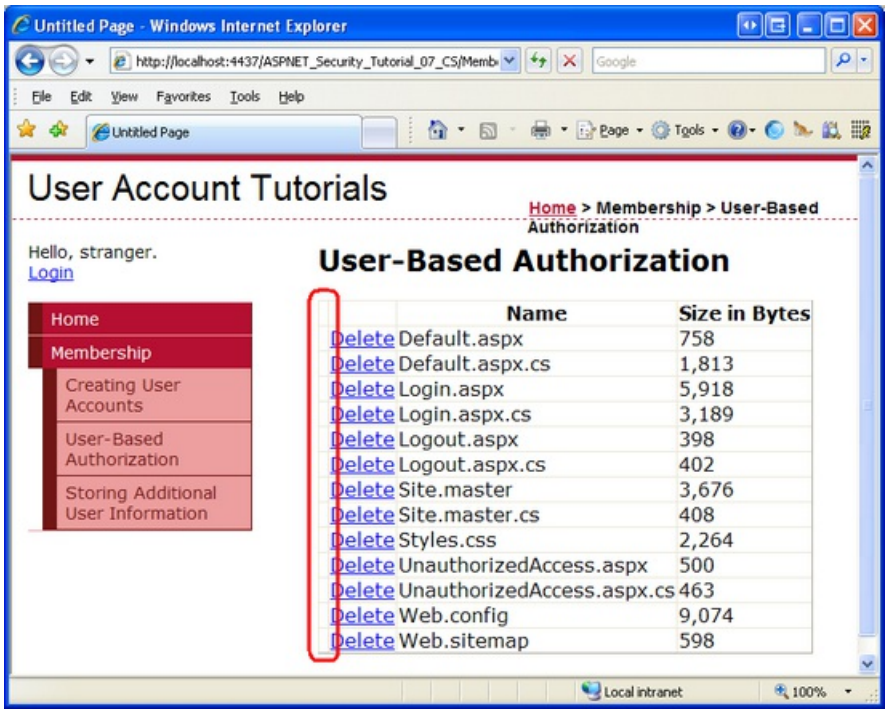


Figure 11: The LoginView Control Hides the View LinkButtons for Anonymous Visitors (Click to view full-size image)

Programmatically Limiting Functionality

In some circumstances, declarative techniques are insufficient for limiting functionality to a page. For example, the availability of certain page functionality may be dependent on criteria beyond whether the user visiting the page is anonymous or authenticated. In such cases, the various user interface elements can be displayed or hidden through programmatic means.

In order to limit the functionality programmatically, we need to perform two tasks:

1. Determine whether the user visiting the page can access the functionality, and
2. Programmatically modify the user interface based on whether the user has access to the functionality in question.

To demonstrate the application of these two tasks, let's only allow Tito to delete files from the GridView. Our first task, then, is to determine whether it is Tito visiting the page. Once that has been determined, we need to hide (or show) the GridView's Delete column. The GridView's columns are accessible through its `Columns` property; a column is only rendered if its `Visible` property is set to `true` (the default).

Add the following code to the `Page_Load` event handler prior to binding the data to the GridView:

```
// Is this Tito visiting the page?
string userName = User.Identity.Name;
if (string.Compare(userName, "Tito", true) == 0)
    // This is Tito, SHOW the Delete column
    FilesGrid.Columns[1].Visible = true;
else
    // This is NOT Tito, HIDE the Delete column
    FilesGrid.Columns[1].Visible = false;
```

As we discussed in the *An Overview of Forms Authentication* tutorial, `User.Identity.Name` returns the identity's name. This corresponds to the username entered in the Login control. If it is Tito visiting the page, the GridView's

second column's `Visible` property is set to `true`; otherwise, it is set to `false`. The net result is that when someone other than Tito visits the page, either another authenticated user or an anonymous user, the Delete column is not rendered (see Figure 12); however, when Tito visits the page, the Delete column is present (see Figure 13).

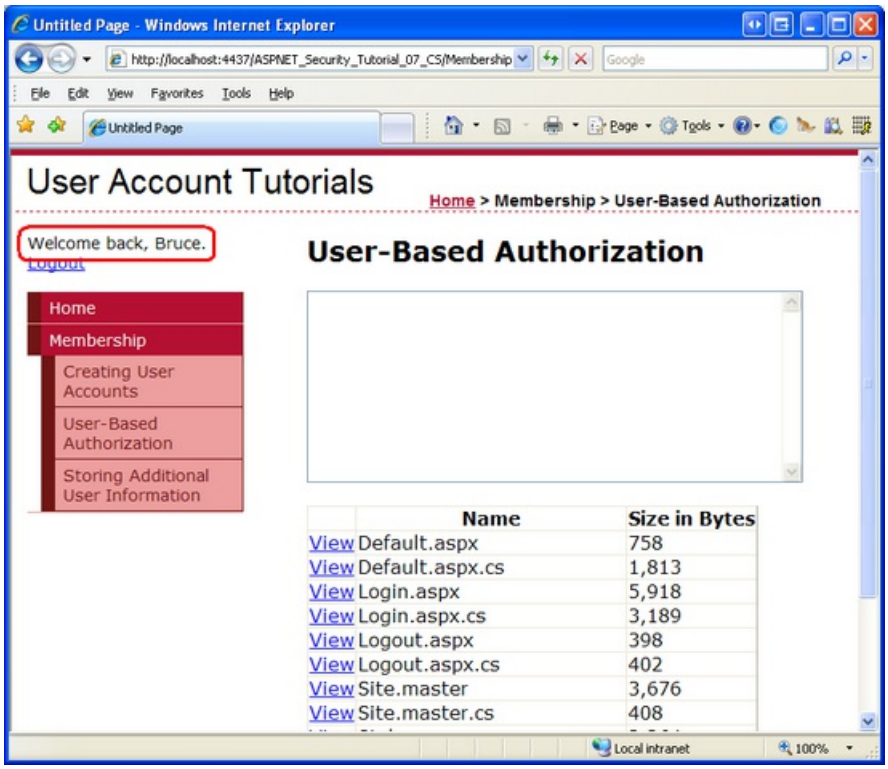


Figure 12: The Delete Column is Not Rendered When Visited By Someone Other Than Tito (Such as Bruce) (Click to view full-size image)

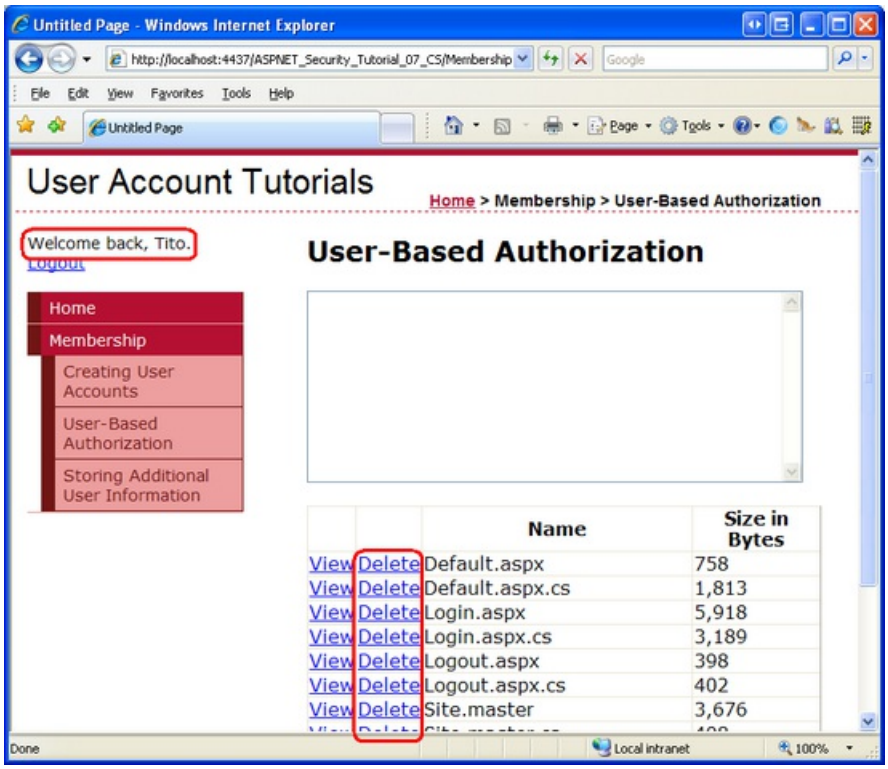


Figure 13: The Delete Column is Rendered for Tito (Click to view full-size image)

Step 4: Applying Authorization Rules to Classes and Methods

In Step 3 we disallowed anonymous users from viewing a file's contents and prohibited all users but Tito from deleting files. This was accomplished by hiding the associated user interface elements for unauthorized visitors

through declarative and programmatic techniques. For our simple example, properly hiding the user interface elements was straightforward, but what about more complex sites where there may be many different ways to perform the same functionality? In limiting that functionality to unauthorized users, what happens if we forget to hide or disable all of the applicable user interface elements?

An easy way to ensure that a particular piece of functionality cannot be accessed by an unauthorized user is to decorate that class or method with the `PrincipalPermission` attribute. When the .NET runtime uses a class or executes one of its methods, it checks to ensure that the current security context has permission to use the class or execute the method. The `PrincipalPermission` attribute provides a mechanism through which we can define these rules.

Let's demonstrate using the `PrincipalPermission` attribute on the GridView's `SelectedIndexChanged` and `RowDeleting` event handlers to prohibit execution by anonymous users and users other than Tito, respectively. All we need to do is add the appropriate attribute atop each function definition:

```
[PrincipalPermission(SecurityAction.Demand, Authenticated=true)]
protected void FilesGrid_SelectedIndexChanged(object sender, EventArgs e)
{
    ...
}

[PrincipalPermission(SecurityAction.Demand, Name="Tito")]
protected void FilesGrid_RowDeleting(object sender, GridViewDeleteEventArgs e)
{
    ...
}
```

The attribute for the `SelectedIndexChanged` event handler dictates that only authenticated users can execute the event handler, where as the attribute on the `RowDeleting` event handler limits the execution to Tito.

If, somehow, a user other than Tito attempts to execute the `RowDeleting` event handler or a non-authenticated user attempts to execute the `SelectedIndexChanged` event handler, the .NET runtime will raise a `SecurityException`.

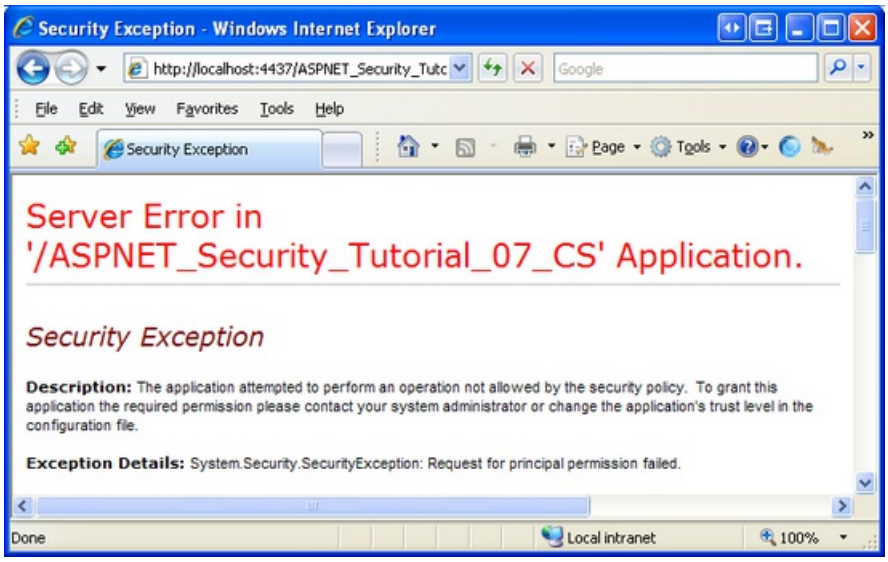


Figure 14: If the Security Context is not Authorized to Execute the Method, a `SecurityException` is Thrown (Click to view full-size image)

> **NOTE**
> To allow multiple security contexts to access a class or method, decorate the class or method with a `PrincipalPermission` attribute for each security context. That is, to allow both Tito and Bruce to execute the `RowDeleting` event handler, add *two* `PrincipalPermission` attributes:

```
[PrincipalPermission(SecurityAction.Demand, Name="Tito")]

[PrincipalPermission(SecurityAction.Demand, Name="Bruce")]
```

In addition to ASP.NET pages, many applications also have an architecture that includes various layers, such as Business Logic and Data Access Layers. These layers are typically implemented as Class Libraries and offer classes and methods for performing business logic- and data-related functionality. The `PrincipalPermission` attribute is useful for applying authorization rules to these layers.

For more information on using the `PrincipalPermission` attribute to define authorization rules on classes and methods, refer to Scott Guthrie's blog entry Adding Authorization Rules to Business and Data Layers Using `PrincipalPermissionAttributes`.

Summary

In this tutorial we looked at how to apply user-based authorization rules. We started with a look at ASP.NET's URL authorization framework. On each request, the ASP.NET engine's `UrlAuthorizationModule` inspects the URL authorization rules defined in the application's configuration to determine whether the identity is authorized to access the requested resource. In short, URL authorization makes it easy to specify authorization rules for a specific page or for all pages in a particular directory.

The URL authorization framework applies authorization rules on a page-by-page basis. With URL authorization, either the requesting identity is authorized to access a particular resource or not. Many scenarios, however, call for more fine grain authorization rules. Rather than defining who is permitted to access a page, we may need to let everyone access a page, but to show different data or offer different functionality depending on the user visiting the page. Page-level authorization usually involves hiding specific user interface elements in order to prevent unauthorized users from accessing prohibited functionality. Additionally, it is possible to use attributes to restrict access to classes and execution of its methods for certain users.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Adding Authorization Rules to Business and Data Layers Using `PrincipalPermissionAttributes`
- ASP.NET Authorization
- Changes Between IIS6 and IIS7 Security
- Configuring Specific Files and Subdirectories
- Limiting Data Modification Functionality Based on the User
- LoginView Control QuickStarts
- Understanding IIS7 URL Authorization
- `UrlAuthorizationModule` Technical Documentation
- Working with Data in ASP.NET 2.0

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working

with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Storing Additional User Information (C#)

2/20/2020 • 46 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> In this tutorial we will answer this question by building a very rudimentary guestbook application. In doing so, we will look at different options for modeling user information in a database, and then see how to associate this data with the user accounts created by the Membership framework.

Introduction

ASP.NET's Membership framework offers a flexible interface for managing users. The Membership API includes methods for validating credentials, retrieving information about the currently logged on user, creating a new user account, and deleting a user account, among others. Each user account in the Membership framework contains only the properties needed for validating credentials and performing essential user account-related tasks. This is evidenced by the methods and properties of the `MembershipUser` class, which models a user account in the Membership framework. This class has properties like `UserName`, `Email`, and `IsLockedOut`, and methods like `GetPassword` and `UnlockUser`.

Oftentimes, applications need to store additional user information not included in the Membership framework. For example, an online retailer might need to let each user store her shipping and billing addresses, payment information, delivery preferences, and contact phone number. Furthermore, each order in the system is associated with a particular user account.

The `MembershipUser` class does not include properties like `PhoneNumber` or `DeliveryPreferences` or `PastOrders`. So how do we track the user information needed by the application and have it integrate with the Membership framework? In this tutorial we will answer this question by building a very rudimentary guestbook application. In doing so, we will look at different options for modeling user information in a database, and then see how to associate this data with the user accounts created by the Membership framework. Let's get started!

Step 1: Creating the Guestbook Application's Data Model

There are a variety of techniques that can be employed to capture user information in a database and associate it with the user accounts created by the Membership framework. In order to illustrate these techniques, we will need to augment the tutorial web application so that it captures some sort of user-related data. (Currently, the application's data model contains only the application services tables needed by the `SqlMembershipProvider`.)

Let's create a very simple guestbook application where an authenticated user can leave a comment. In addition to storing guestbook comments, let's allow each user to store his home town, homepage, and signature. If provided, the user's home town, homepage, and signature will appear on each message he has left in the guestbook.

Adding the `GuestbookComments` Table

In order to capture the guestbook comments, we need to create a database table named `GuestbookComments` that has columns like `CommentId`, `Subject`, `Body`, and `CommentDate`. We also need to have each record in the `GuestbookComments` table reference the user who left the comment.

To add this table to our database, go to the Database Explorer in Visual Studio and drill down into the `SecurityTutorials` database. Right-click on the Tables folder and choose Add New Table. This brings up an interface that allows us to define the columns for the new table.

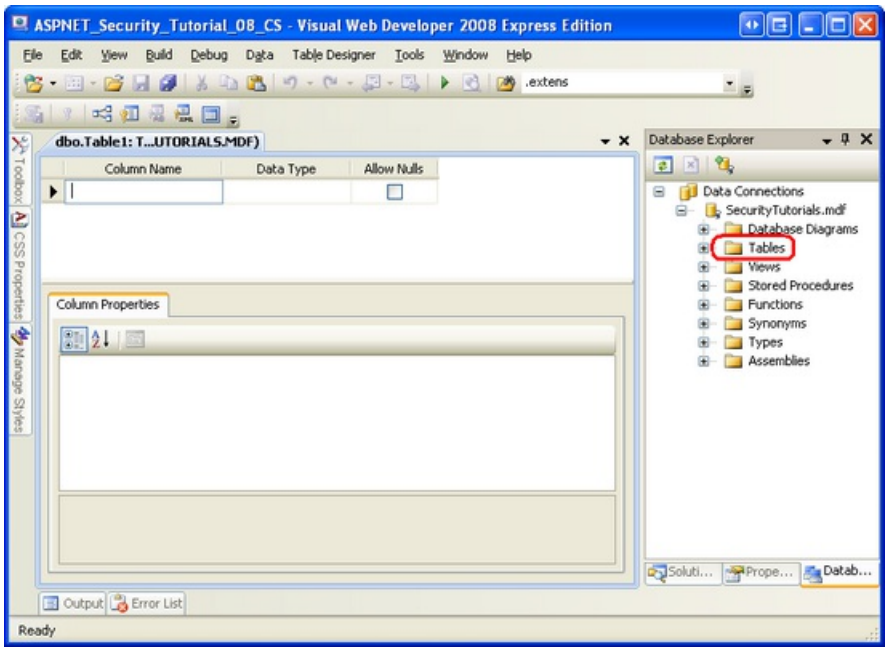


Figure 1: Add a New Table to the `SecurityTutorials` Database (Click to view full-size image)

Next, define the `GuestbookComments`'s columns. Start by adding a column named `CommentId` of type `uniqueidentifier`. This column will uniquely identify each comment in the guestbook, so disallow `NULL`s and mark it as the table's primary key. Rather than providing a value for the `CommentId` field on each `INSERT`, we can indicate that a new `uniqueidentifier` value should be automatically generated for this field on `INSERT` by setting the column's default value to `NEWID()`. After adding this first field, marking it as the primary key, and settings its default value, your screen should look similar to the screen shot shown in Figure 2.

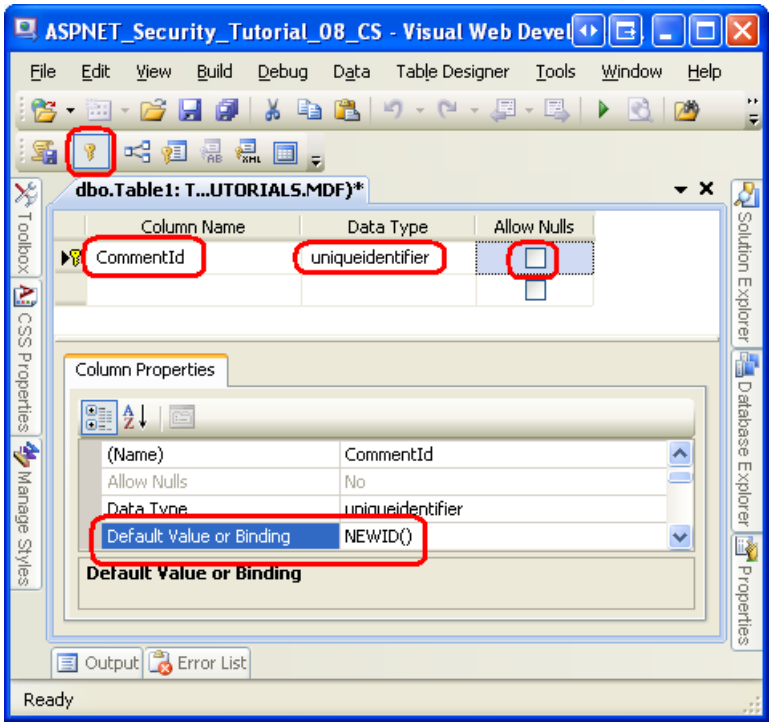


Figure 2: Add a Primary Column Named `CommentId` (Click to view full-size image)

Next, add a column named `Subject` of type `nvarchar(50)` and a column named `Body` of type `nvarchar(MAX)`, disallowing `NULL`s in both columns. Following that, add a column named `CommentDate` of type `datetime`. Disallow `NULL`s and set the `CommentDate` column's default value to `getdate()`.

All that remains is to add a column that associates a user account with each guestbook comment. One option would be to add a column named `UserName` of type `nvarchar(256)`. This is a suitable choice when using a Membership provider other than the `SqlMembershipProvider`. But when using the `SqlMembershipProvider`, as we

are in this tutorial series, the `UserName` column in the `aspnet_Users` table is not guaranteed to be unique. The `aspnet_Users` table's primary key is `UserId` and is of type `uniqueidentifier`. Therefore, the `GuestbookComments` table needs a column named `UserId` of type `uniqueidentifier` (disallowing `NULL` values). Go ahead and add this column.

> **NOTE**
>
> As we discussed in the *Creating the Membership Schema in SQL Server* tutorial, the Membership framework is designed to enable multiple web applications with different user accounts to share the same user store. It does this by partitioning user accounts into different applications. And while each username is guaranteed to be unique within an application, the same username may be used in different applications using the same user store. There is a composite `UNIQUE` constraint in the `aspnet_Users` table on the `UserName` and `ApplicationId` fields, but not one on just the `UserName` field. Consequently, it is possible for the aspnet_Users table to have two (or more) records with the same `UserName` value. There is, however, a `UNIQUE` constraint on the `aspnet_Users` table's `UserId` field (since it is the primary key). A `UNIQUE` constraint is important because without it we cannot establish a foreign key constraint between the `GuestbookComments` and `aspnet_Users` tables.

After adding the `UserId` column, save the table by clicking on the Save icon in the Toolbar. Name the new table `GuestbookComments`.

We have one last issue to attend to with the `GuestbookComments` table: we need to create a foreign key constraint between the `GuestbookComments.UserId` column and the `aspnet_Users.UserId` column. To achieve this, click the Relationship icon in the Toolbar to launch the Foreign Key Relationships dialog box. (Alternatively, you can launch this dialog box by going to the Table Designer menu and choosing Relationships.)

Click the Add button in the lower left corner of the Foreign Key Relationships dialog box. This will add a new foreign key constraint, although we still need to define the tables that participate in the relationship.

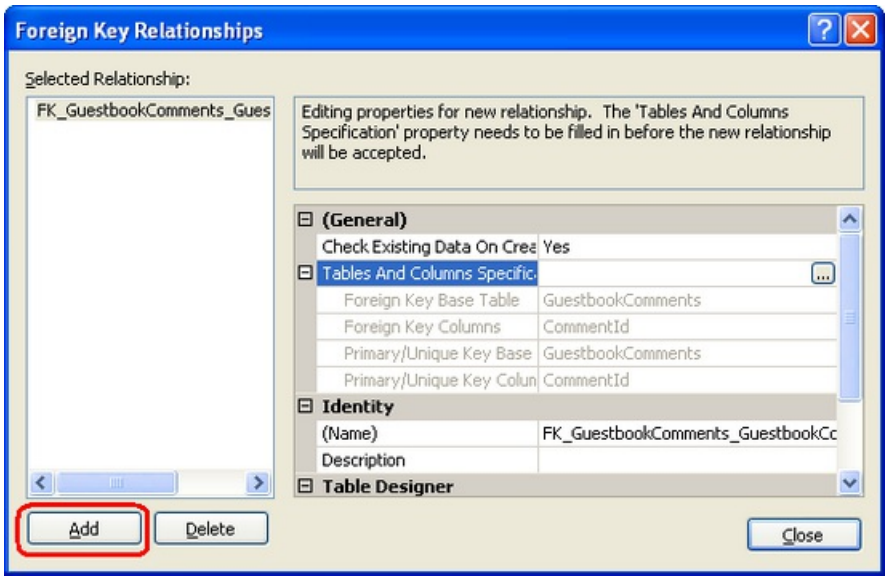


Figure 3: Use the Foreign Key Relationships Dialog Box to Manage a Table's Foreign Key Constraints (Click to view full-size image)

Next, click the ellipses icon in the "Table and Columns Specifications" row on the right. This will launch the Tables and Columns dialog box, from which we can specify the primary key table and column and the foreign key column from the `GuestbookComments` table. In particular, select `aspnet_Users` and `UserId` as the primary key table and column, and `UserId` from the `GuestbookComments` table as the foreign key column (see Figure 4). After defining the primary and foreign key tables and columns, click OK to return to the Foreign Key Relationships dialog box.

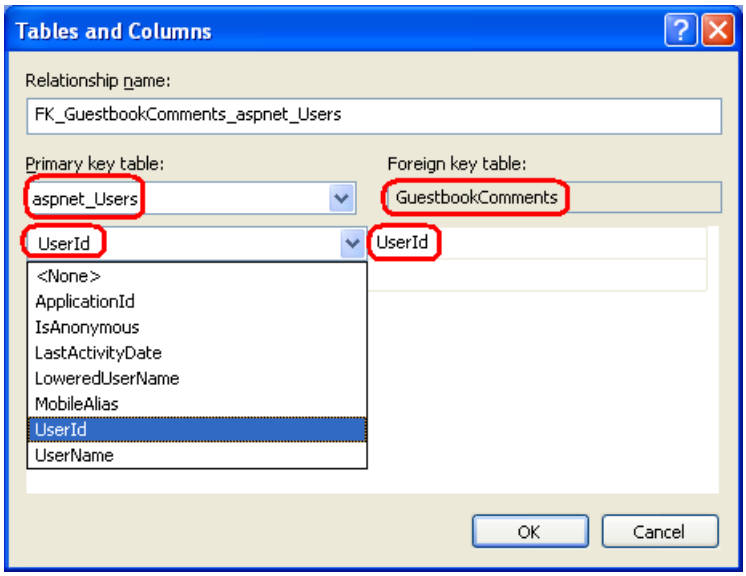


Figure 4: Establish a Foreign Key Constraint Between the `aspnet_Users` and `GuesbookComments` Tables (Click to view full-size image)

At this point the foreign key constraint has been established. The presence of this constraint ensures relational integrity between the two tables by guaranteeing that there will never be a guestbook entry referring to a non-existent user account. By default, a foreign key constraint will disallow a parent record to be deleted if there are corresponding child records. That is, if a user makes one or more guestbook comments, and then we attempt to delete that user account, the delete will fail unless his guestbook comments are deleted first.

Foreign key constraints can be configured to automatically delete the associated child records when a parent record is deleted. In other words, we can setup this foreign key constraint so that a user's guestbook entries are automatically deleted when her user account is deleted. To accomplish this, expand the "INSERT And UPDATE Specification" section and set the "Delete Rule" property to Cascade.

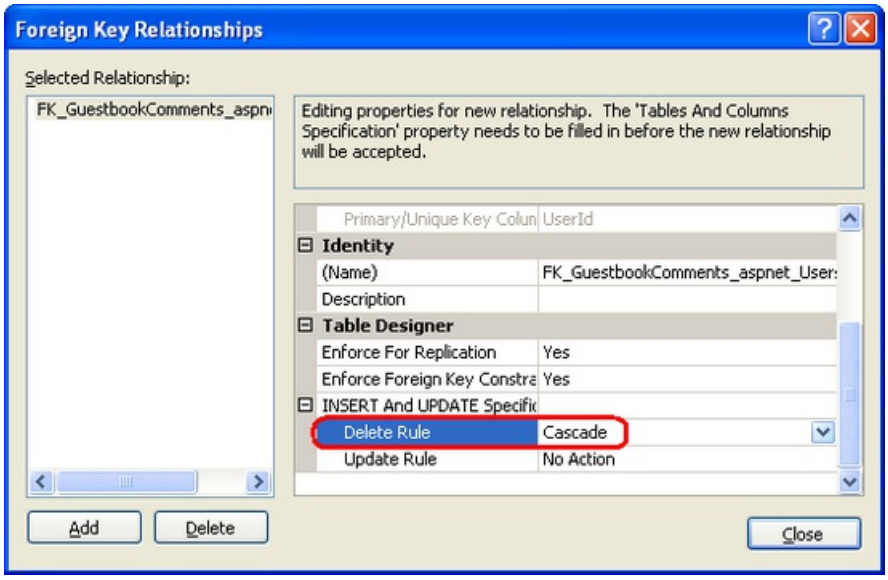


Figure 5: Configure the Foreign Key Constraint to Cascade Deletes (Click to view full-size image)

To save the foreign key constraint, click the Close button to exit out of the Foreign Key Relationships. Then click the Save icon in the Toolbar to save the table and the this relationship.

Storing the User's Home Town, Homepage, and Signature

The `GuestbookComments` table illustrates how to store information that shares a one-to-many relationship with user accounts. Since each user account may have an arbitrary number of associated comments, this relationship is modeled by creating a table to hold the set of comments that includes a column that links back each comment to a particular user. When using the `SqlMembershipProvider`, this link is best established by creating a column

named `UserId` of type `uniqueidentifier` and a foreign key constraint between this column and `aspnet_Users.UserId`.

We now need to associate three columns with each user account to store the user's home town, homepage, and signature, which will appear in his guestbook comments. There are number of different ways to accomplish this:

- **Add new columns to the `aspnet_Users` or `aspnet_Membership` tables.** I would not recommend this approach because it modifies the schema used by the `SqlMembershipProvider`. This decision may come back to haunt you down the road. For example, what if a future version of ASP.NET uses a different `SqlMembershipProvider` schema. Microsoft may include a tool to migrate the ASP.NET 2.0 `SqlMembershipProvider` data to the new schema, but if you have modified the ASP.NET 2.0 `SqlMembershipProvider` schema, such a conversion may not be possible.

- **Use ASP.NET's Profile framework, defining a profile property for the home town, homepage, and signature.** ASP.NET includes a Profile framework that is designed to store additional user-specific data. Like the Membership framework, the Profile framework is built atop the provider model. The .NET Framework ships with a `SqlProfileProvider` sthat stores profile data in a SQL Server database. In fact, our database already has the table used by the `SqlProfileProvider` (`aspnet_Profile`), as it was added when we added the application services back in the *Creating the Membership Schema in SQL Server* tutorial. The main benefit of the Profile framework is that it allows for developers to define the profile properties in `Web.config` – no code needs to be written to serialize the profile data to and from the underlying data store. In short, it is incredibly easy to define a set of profile properties and to work with them in code. However, the Profile system leaves a lot to be desired when it comes to versioning, so if you have an application where you expect new user-specific properties to be added at a later time, or existing ones to be removed or modified, then the Profile framework may not be the best option. Moreover, the `SqlProfileProvider` stores the profile properties in a highly denormalized fashion, making it next to impossible to run queries directly against the profile data (such as, how many users have a home town of New York).
  For more information on the Profile framework, consult the "Further Readings" section at the end of this tutorial.

- **Add these three columns to a new table in the database and establish a one-to-one relationship between this table and `aspnet_Users`.** This approach involves a bit more work than with the Profile framework, but offers maximum flexibility in how the additional user properties are modeled in the database. This is the option we will use in this tutorial.

We will create a new table called `UserProfiles` to save the home town, homepage, and signature for each user. Right-click on the Tables folder in the Database Explorer window and choose to create a new table. Name the first column `UserId` and set its type to `uniqueidentifier`. Disallow `NULL` values and mark the column as a primary key. Next, add columns named: `HomeTown` of type `nvarchar(50)`; `HomepageUrl` of type `nvarchar(100)`; and Signature of type `nvarchar(500)`. Each of these three columns can accept a `NULL` value.

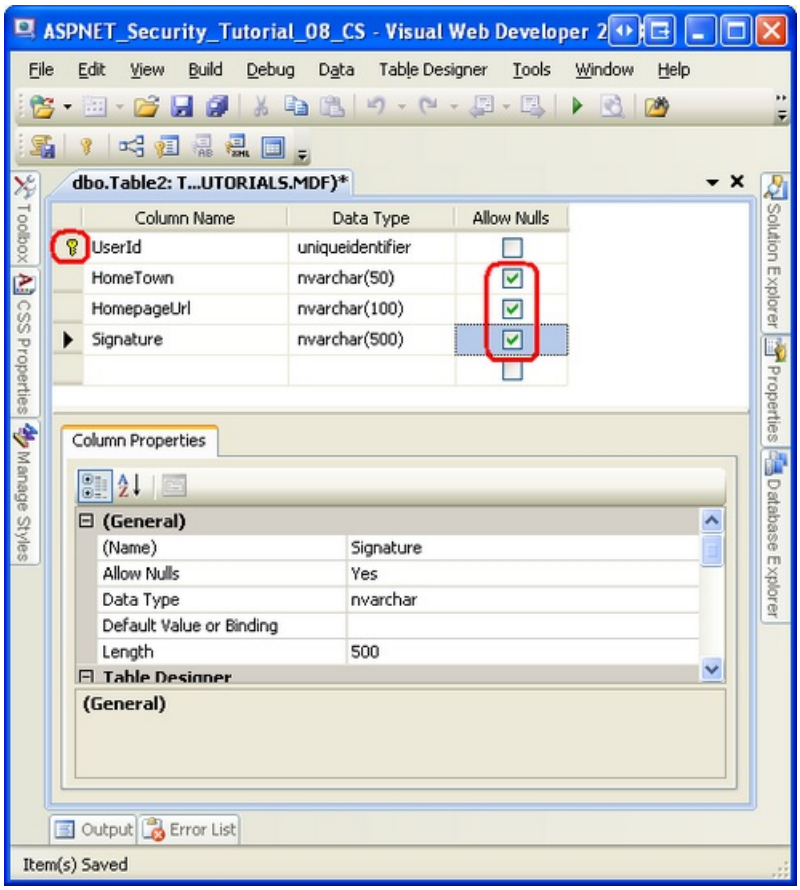


Figure 6: Create the `UserProfiles` Table (Click to view full-size image)

Save the table and name it `UserProfiles`. Lastly, establish a foreign key constraint between the `UserProfiles` table's `UserId` field and the `aspnet_Users.UserId` field. As we did with the foreign key constraint between the `GuestbookComments` and `aspnet_Users` tables, have this constraint cascade deletes. Since the `UserId` field in `UserProfiles` is the primary key, this ensures that there will be no more than one record in the `UserProfiles` table for each user account. This type of relationship is referred to as one-to-one.

Now that we have the data model created, we are ready to use it. In Steps 2 and 3 we will look at how the currently logged on user can view and edit their home town, homepage, and signature information. In Step 4 we will create the interface for authenticated users to submit new comments to the guestbook and view the existing ones.

Step 2: Displaying the User's Home Town, Homepage, and Signature

There are a variety of ways to allow the currently logged on user to view and edit his home town, homepage, and signature information. We could manually create the user interface with TextBox and Label controls or we could use one of the data Web controls, such as the DetailsView control. To perform the database `SELECT` and `UPDATE` statements we could write ADO.NET code in our page's code-behind class or, alternatively, employ a declarative approach with the SqlDataSource. Ideally our application would contain a tiered architecture, which we could either invoke programmatically from the page's code-behind class or declaratively via the ObjectDataSource control.

Since this tutorial series focuses on forms authentication, authorization, user accounts, and roles, there will not be a thorough discussion of these different data access options or why a tiered architecture is preferred over executing SQL statements directly from the ASP.NET page. I am going to walk through using a DetailsView and SqlDataSource – the quickest and easiest option – but the concepts discussed can certainly be applied to alternative Web controls and data access logic. For more information on working with data in ASP.NET, refer to my *Working with Data in ASP.NET 2.0* tutorial series.

Open the `AdditionalUserInfo.aspx` page in the `Membership` folder and add a DetailsView control to the page,

setting its `ID` property to `UserProfile` and clearing out its `Width` and `Height` properties. Expand the DetailsView's Smart Tag and choose to bind it to a new data source control. This will launch the DataSource Configuration Wizard (see Figure 7). The first step asks you to specify the data source type. Since we are going to connect directly to the `SecurityTutorials` database, choose the Database icon, specifying the `ID` as `UserProfileDataSource`.

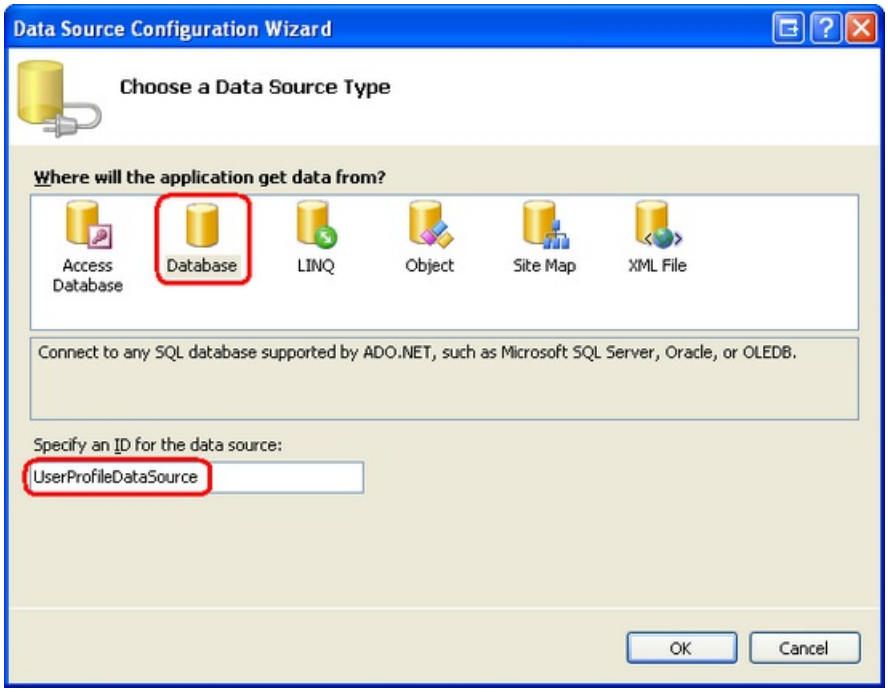


Figure 7: Add a New SqlDataSource Control Named `UserProfileDataSource` (Click to view full-size image)

The next screen prompts for the database to use. We have already defined a connection string in `Web.config` for the `SecurityTutorials` database. This connection string name – `SecurityTutorialsConnectionString` – should be in the drop-down list. Select this option and click Next.

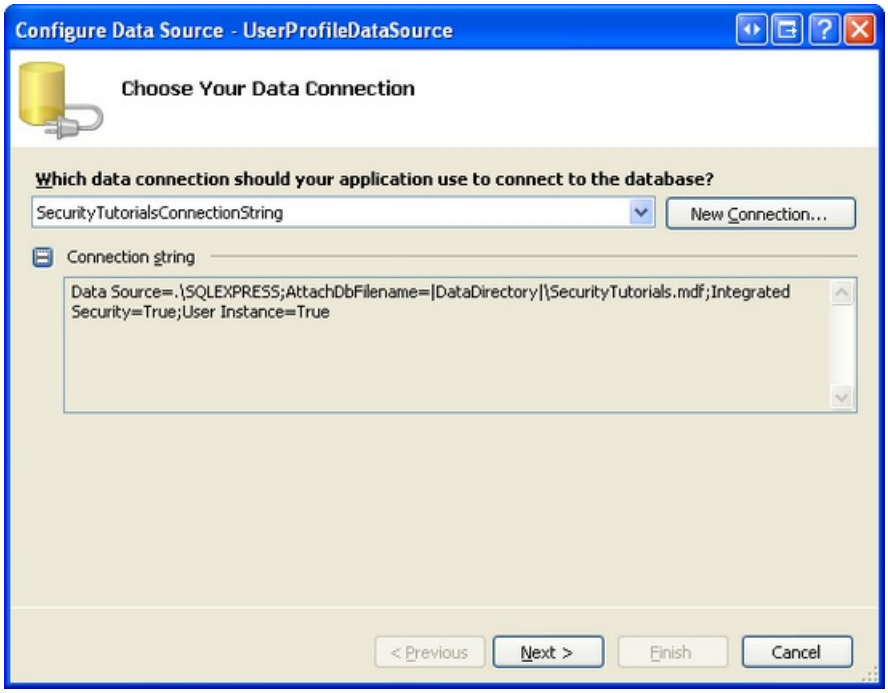


Figure 8: Choose `SecurityTutorialsConnectionString` from the Drop-Down List (Click to view full-size image)

The subsequent screen asks us to specify the table and columns to query. Choose the `UserProfiles` table from the drop-down list and check all of the columns.

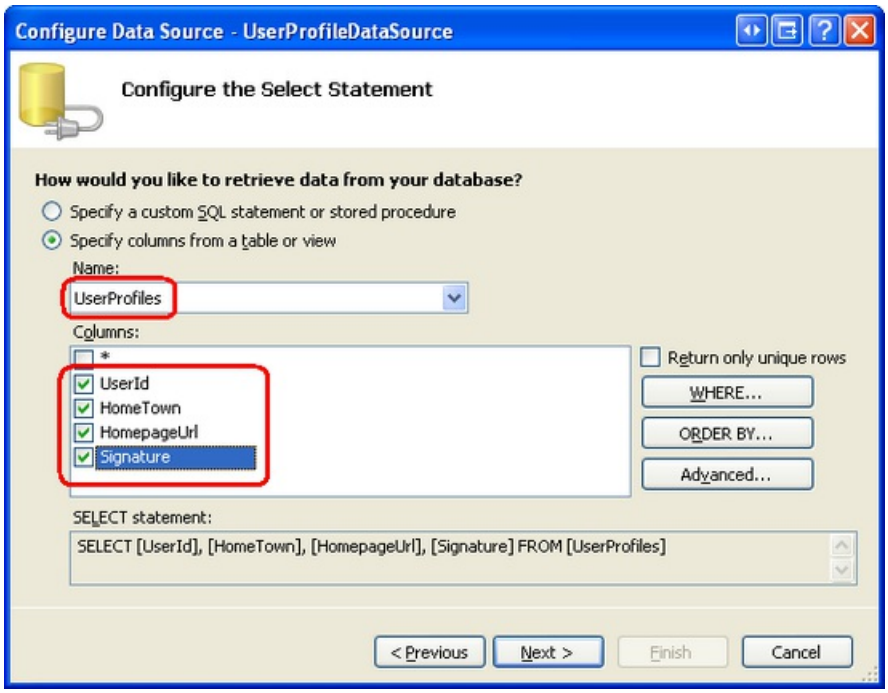


Figure 9: Bring Back All of the Columns from the `UserProfiles` Table (Click to view full-size image)

The current query in Figure 9 returns *all* of the records in `UserProfiles`, but we are only interested in the currently logged on user's record. To add a `WHERE` clause, click the `WHERE` button to bring up the Add `WHERE` Clause dialog box (see Figure 10). Here you can select the column to filter on, the operator, and the source of the filter parameter. Select `UserId` as the column and "=" as the Operator.

Unfortunately there is no built-in parameter source to return the currently logged on user's `UserId` value. We will need to grab this value programmatically. Therefore, set the Source drop-down list to "None," click the Add button to add the parameter, and then click OK.

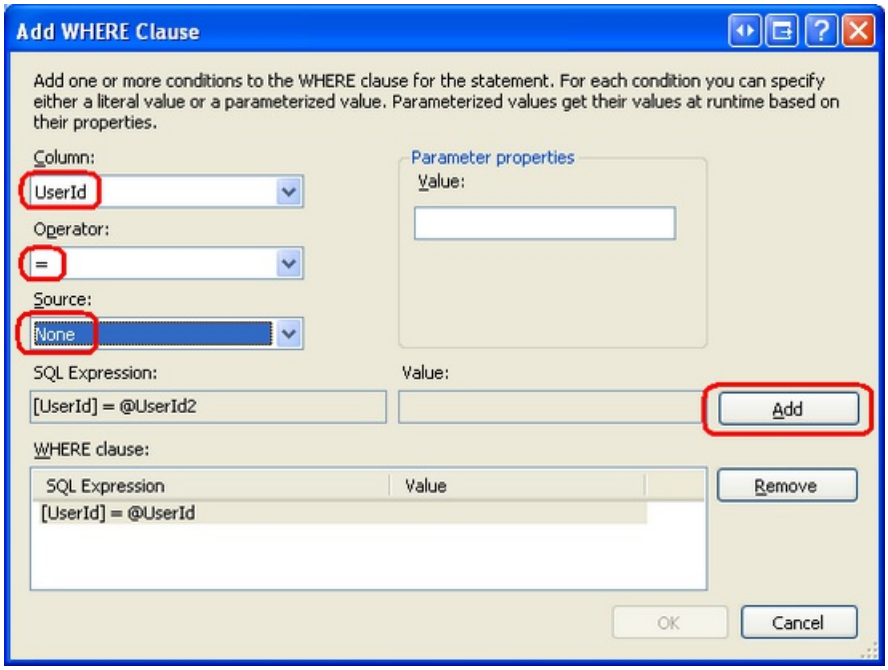


Figure 10: Add a Filter Parameter on the `UserId` Column (Click to view full-size image)

After clicking OK you will be returned to the screen shown in Figure 9. This time, however, the SQL query at the bottom of the screen should include a `WHERE` clause. Click Next to move on to the "Test Query" screen. Here you can run the query and see the results. Click Finish to complete the wizard.

Upon completing the DataSource Configuration Wizard, Visual Studio creates the SqlDataSource control based on the settings specified in the wizard. Moreover, it manually adds BoundFields to the DetailsView for each

column returned by the SqlDataSource's `SelectCommand`. There's no need to show the `UserId` field in the DetailsView, since the user does not need to know this value. You can remove this field directly from the DetailsView control's declarative markup or by clicking the "Edit Fields" link from its Smart Tag.

At this point your page's declarative markup should look similar to the following:

```
<asp:DetailsView ID="UserProfile" runat="server"
     AutoGenerateRows="False" DataKeyNames="UserId"
     DataSourceID="UserProfileDataSource">
     <Fields>
          <asp:BoundField DataField="HomeTown" HeaderText="HomeTown"
               SortExpression="HomeTown" />
          <asp:BoundField DataField="HomepageUrl" HeaderText="HomepageUrl"
               SortExpression="HomepageUrl" />
          <asp:BoundField DataField="Signature" HeaderText="Signature"
               SortExpression="Signature" />
     </Fields>
</asp:DetailsView>
<asp:SqlDataSource ID="UserProfileDataSource" runat="server"
          ConnectionString="<%$ ConnectionStrings:SecurityTutorialsConnectionString %>"
          SelectCommand="SELECT [UserId], [HomeTown], [HomepageUrl], [Signature] FROM
          [UserProfiles] WHERE ([UserId] = @UserId)">
     <SelectParameters>
          <asp:Parameter Name="UserId" Type="Object" />
     </SelectParameters>
</asp:SqlDataSource>
```

We need to programmatically set the SqlDataSource control's `UserId` parameter to the currently logged in user's `UserId` before the data is selected. This can be accomplished by creating an event handler for the SqlDataSource's `Selecting` event and adding the following code there:

```
protected void UserProfileDataSource_Selecting(object sender,
          SqlDataSourceSelectingEventArgs e)
{
     // Get a reference to the currently logged on user
     MembershipUser currentUser = Membership.GetUser();

     // Determine the currently logged on user's UserId value
     Guid currentUserId = (Guid)currentUser.ProviderUserKey;

     // Assign the currently logged on user's UserId to the @UserId parameter
     e.Command.Parameters["@UserId"].Value = currentUserId;
}
```

The above code starts by obtaining a reference to the currently logged on user by calling the `Membership` class's `GetUser` method. This returns a `MembershipUser` object, whose `ProviderUserKey` property contains the `UserId`. The `UserId` value is then assigned to the SqlDataSource's `@UserId` parameter.

NOTE

The `Membership.GetUser()` method returns information about the currently logged on user. If an anonymous user is visiting the page, it will return a value of `null`. In such a case, this will lead to a `NullReferenceException` on the following line of code when attempting to read the `ProviderUserKey` property. Of course, we don't have to worry about `Membership.GetUser()` returning a `null` value in the `AdditionalUserInfo.aspx` page because we configured URL authorization in a previous tutorial so that only authenticated users could access the ASP.NET resources in this folder. If you need to access information about the currently logged on user in a page where anonymous access is permitted, make sure to check that a non-`null MembershipUser` object is returned from the `GetUser()` method before referencing its properties.

If you visit the `AdditionalUserInfo.aspx` page through a browser you will see a blank page because we have yet to add any rows to the `UserProfiles` table. In Step 6 we will look at how to customize the CreateUserWizard control to automatically add a new row to the `UserProfiles` table when a new user account is created. For now, however, we will need to manually create a record in the table.

Navigate to the Database Explorer in Visual Studio and expand the Tables folder. Right-click on the `aspnet_Users` table and choose "Show Table Data" to see the records in the table; do the same thing for the `UserProfiles` table. Figure 11 shows these results when tiled vertically. In my database there are currently `aspnet_Users` records for Bruce, Fred, and Tito, but no records in the `UserProfiles` table.

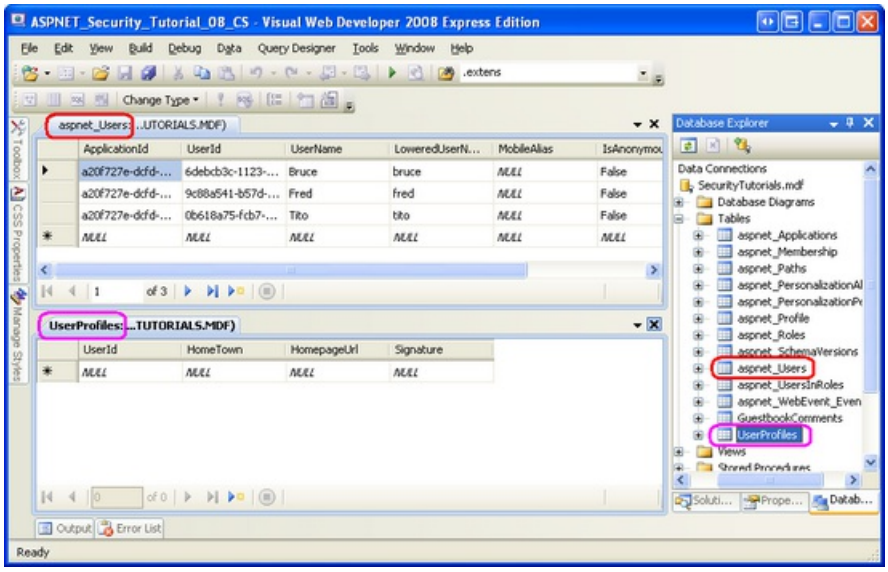


Figure 11: The Contents of the `aspnet_Users` and `UserProfiles` Tables are Displayed (Click to view full-size image)

Add a new record to the `UserProfiles` table by manually typing in values for the `HomeTown`, `HomepageUrl`, and `Signature` fields. The easiest way to get a valid `UserId` value in the new `UserProfiles` record is to select the `UserId` field from a particular user account in the `aspnet_Users` table and copy and paste it into the `UserId` field in `UserProfiles`. Figure 12 shows the `UserProfiles` table after a new record has been added for Bruce.

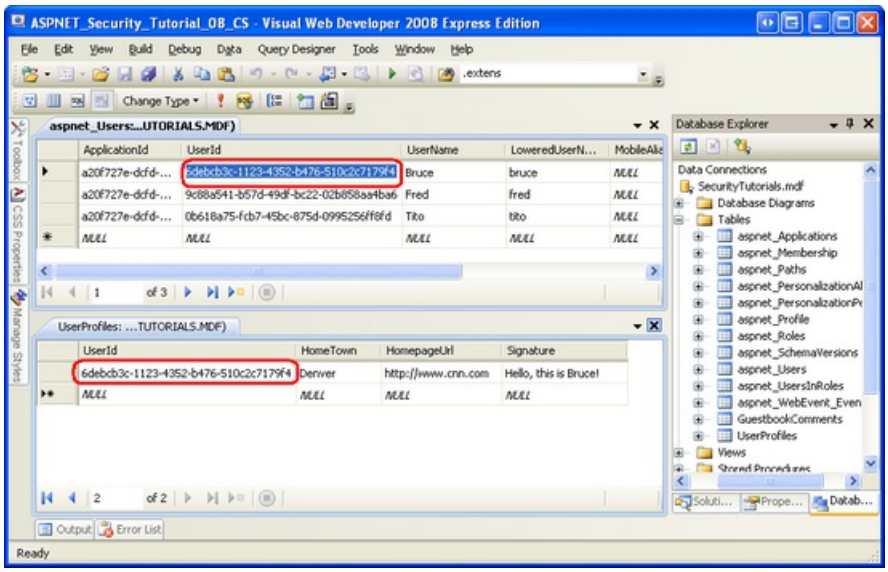


Figure 12: A Record was Added to `UserProfiles` for Bruce (Click to view full-size image)

Return to the `AdditionalUserInfo.aspx` page, logged in as Bruce. As Figure 13 shows, Bruce's settings are displayed.

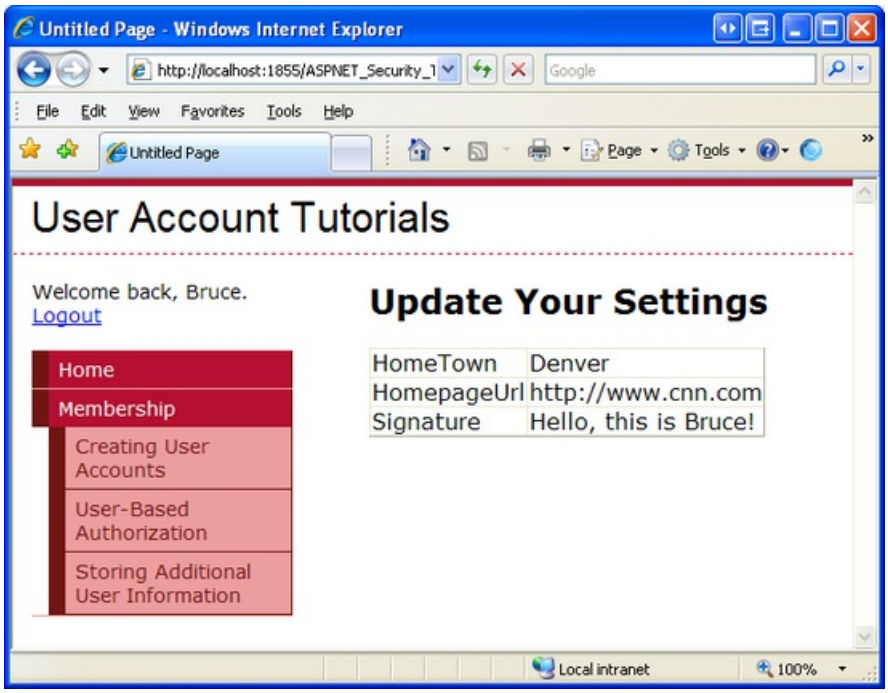


Figure 13: The Currently Visiting User is Shown His Settings (Click to view full-size image)

> **NOTE**
>
> Go ahead and manually add records in the `UserProfiles` table for each Membership user. In Step 6 we will look at how to customize the CreateUserWizard control to automatically add a new row to the `UserProfiles` table when a new user account is created.

Step 3: Allowing the User to Edit His Home Town, Homepage, and Signature

At this point the currently logged in user can view their home town, homepage, and signature setting, but they cannot yet modify them. Let's update the DetailsView control so that the data can be edited.

The first thing we need to do is add an `UpdateCommand` for the SqlDataSource, specifying the `UPDATE` statement to execute and its corresponding parameters. Select the SqlDataSource and, from the Properties window, click on the ellipses next to the UpdateQuery property to bring up the Command and Parameter Editor dialog box. Enter the following `UPDATE` statement into the textbox:

```
UPDATE UserProfiles SET
     HomeTown = @HomeTown,
     HomepageUrl = @HomepageUrl,
     Signature = @Signature
WHERE UserId = @UserId
```

Next, click the "Refresh Parameters" button, which will create a parameter in the SqlDataSource control's `UpdateParameters` collection for each of the parameters in the `UPDATE` statement. Leave the source for all of the parameters set to None and click the OK button to complete the dialog box.

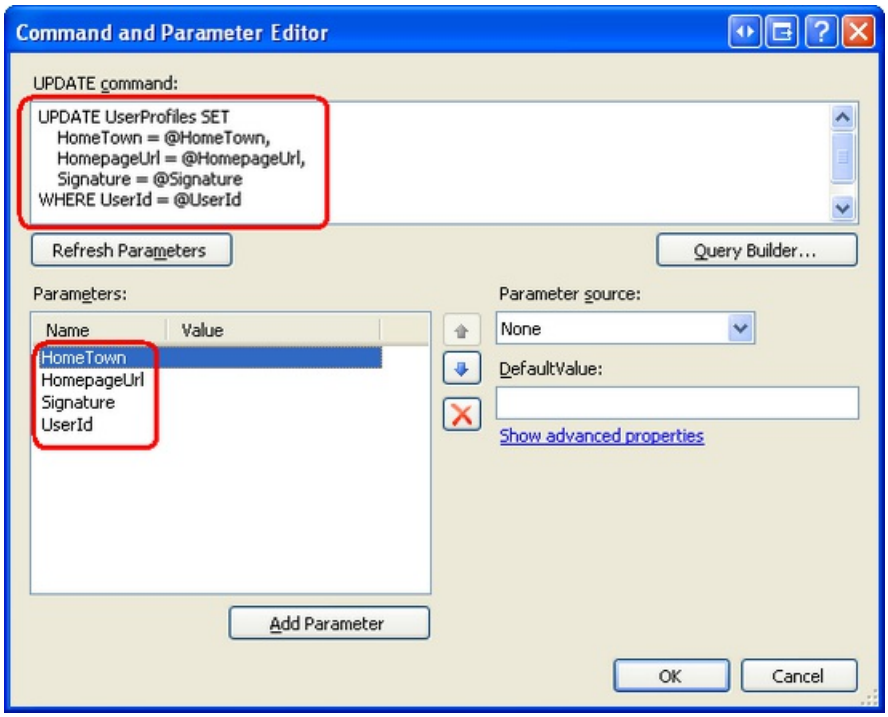


Figure 14: Specify the SqlDataSource's `UpdateCommand` and `UpdateParameters` (Click to view full-size image)

Due to the additions we made to the SqlDataSource control, the DetailsView control can now support editing. From the DetailsView's Smart Tag, check the "Enable Editing" checkbox. This adds a CommandField to the control's `Fields` collection with its `ShowEditButton` property set to True. This renders an Edit button when the DetailsView is displayed in read-only mode and Update and Cancel buttons when displayed in edit mode. Rather than requiring the user to click Edit, though, we can have the DetailsView render in an "always editable" state by setting the DetailsView control's `DefaultMode` property to `Edit`.

With these changes, your DetailsView control's declarative markup should look similar to the following:

```
<asp:DetailsView ID="UserProfile" runat="server"
          AutoGenerateRows="False" DataKeyNames="UserId"
          DataSourceID="UserProfileDataSource" DefaultMode="Edit">
     <Fields>
          <asp:BoundField DataField="HomeTown" HeaderText="HomeTown"
               SortExpression="HomeTown" />
          <asp:BoundField DataField="HomepageUrl" HeaderText="HomepageUrl"
               SortExpression="HomepageUrl" />
          <asp:BoundField DataField="Signature" HeaderText="Signature"
               SortExpression="Signature" />
          <asp:CommandField ShowEditButton="True" />
     </Fields>
</asp:DetailsView>
```

Note the addition of the CommandField and the `DefaultMode` property.

Go ahead and test this page through a browser. When visiting with a user that has a corresponding record in `UserProfiles`, the user's settings are displayed in an editable interface.

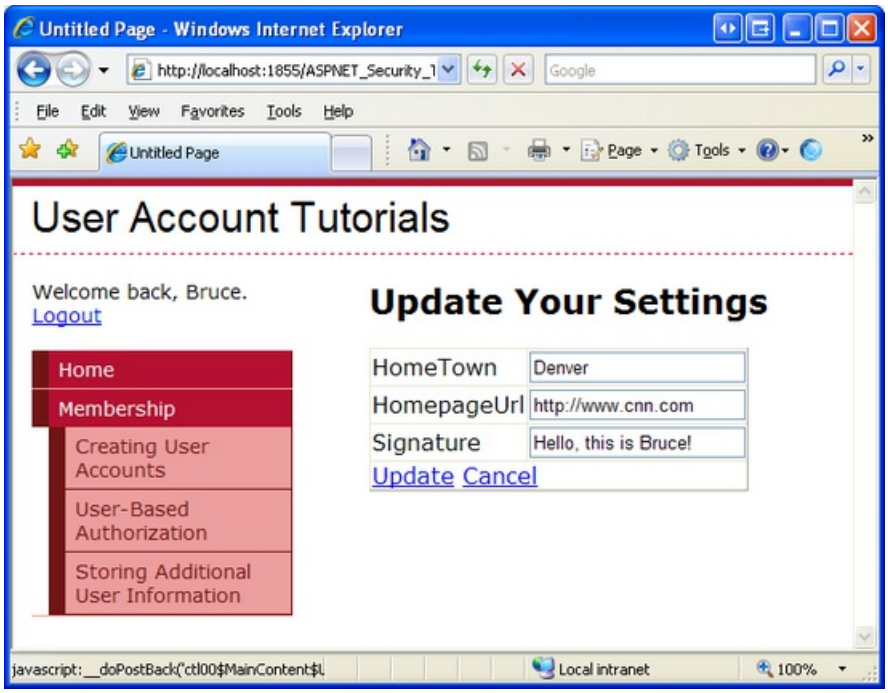


Figure 15: The DetailsView Renders an Editable Interface (Click to view full-size image)

Try changing the values and clicking the Update button. It appears as if nothing happens. There is a postback and the values are saved to the database, but there's no visual feedback that the save occurred.

To remedy this, return to Visual Studio and add a Label control above the DetailsView. Set its `ID` to `SettingsUpdatedMessage`, its `Text` property to "Your settings have been updated," and its `Visible` and `EnableViewState` properties to `false`.

```
<asp:Label ID="SettingsUpdatedMessage" runat="server"
     Text="Your settings have been updated."
     EnableViewState="false"
     Visible="false"></asp:Label>
```

We need to display the `SettingsUpdatedMessage` Label whenever the DetailsView is updated. To accomplish this, create an event handler for the DetailsView's `ItemUpdated` event and add the following code:

```
protected void UserProfile_ItemUpdated(object sender, DetailsViewUpdatedEventArgs e)
{
     SettingsUpdatedMessage.Visible = true;
}
```

Return to the `AdditionalUserInfo.aspx` page through a browser and update the data. This time, a helpful status message is displayed.

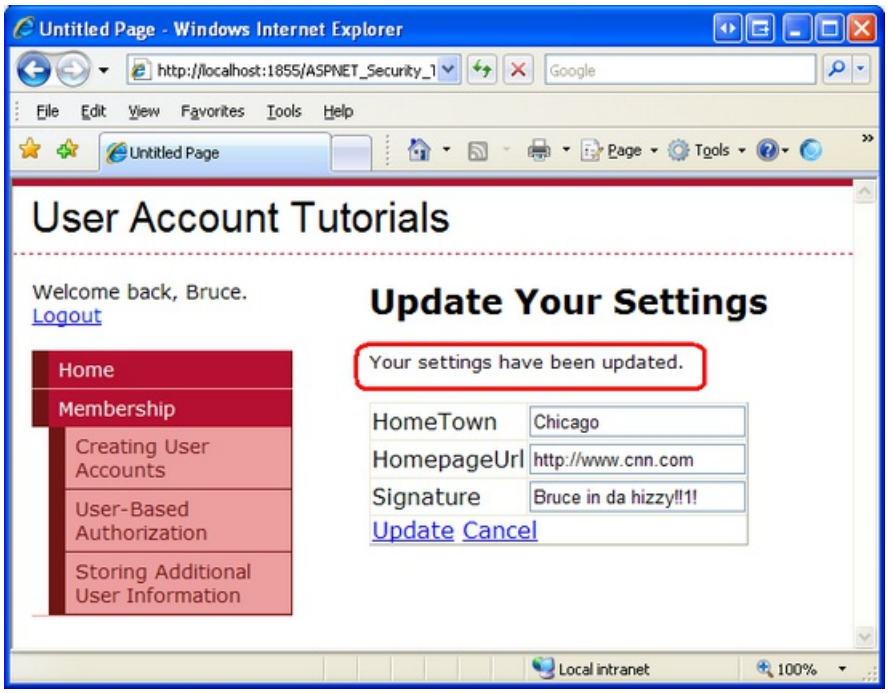


Figure 16: A Short Message is Displayed When the Settings are Updated (Click to view full-size image)

> **NOTE**
>
> The DetailsView control's editing interface leaves a lot to be desired. It uses standard-sized textboxes, but the Signature field should probably be a multi-line textbox. A RegularExpressionValidator should be used to ensure that the homepage URL, if entered, starts with "http://" or "https://". Moreover, since the DetailsView control has its `DefaultMode` property set to `Edit`, the Cancel button does not do anything. It should either be removed or, when clicked, redirect the user to some other page (such as `~/Default.aspx`). I leave these enhancements as an exercise for the reader.

Adding a Link to the `AdditionalUserInfo.aspx` Page in the Master Page

Currently, the website does not provide any links to the `AdditionalUserInfo.aspx` page. The only way to reach it is to enter the page's URL directly into the browser's Address bar. Let's add a link to this page in the `Site.master` master page.

Recall that the master page contains a LoginView Web control in its `LoginContent` ContentPlaceHolder that displays different markup for authenticated and anonymous visitors. Update the LoginView control's `LoggedInTemplate` to include a link to the `AdditionalUserInfo.aspx` page. After making these changes the LoginView control's declarative markup should look similar to the following:

```
<asp:LoginView ID="LoginView1" runat="server">
     <LoggedInTemplate>
          Welcome back,
          <asp:LoginName ID="LoginName1" runat="server" />.
          <br />
          <asp:HyperLink ID="lnkUpdateSettings" runat="server"
               NavigateUrl="~/Membership/AdditionalUserInfo.aspx">
               Update Your Settings</asp:HyperLink>
     </LoggedInTemplate>
     <AnonymousTemplate>
          Hello, stranger.
     </AnonymousTemplate>
</asp:LoginView>
```

Note the addition of the `lnkUpdateSettings` HyperLink control to the `LoggedInTemplate`. With this link in place, authenticated users can quickly jump to the page to view and modify their home town, homepage, and signature settings.

Step 4: Adding New Guestbook Comments

The `Guestbook.aspx` page is where authenticated users can view the guestbook and leave a comment. Let's start with creating the interface to add new guestbook comments.

Open the `Guestbook.aspx` page in Visual Studio and construct a user interface consisting of two TextBox controls, one for the new comment's subject and one for its body. Set the first TextBox control's `ID` property to `Subject` and its `Columns` property to 40; set second's `ID` to `Body`, its `TextMode` to `MultiLine`, and its `Width` and `Rows` properties to "95%" and 8, respectively. To complete the user interface, add a Button Web control named `PostCommentButton` and set its `Text` property to "Post Your Comment".

Since each guestbook comment requires a subject and body, add a RequiredFieldValidator for each of the TextBoxes. Set the `ValidationGroup` property of these controls to "EnterComment"; likewise, set the `PostCommentButton` control's `ValidationGroup` property to "EnterComment". For more information on ASP.NET's validation controls, check out Form Validation in ASP.NET, Dissecting the Validation Controls in ASP.NET 2.0, and the Validation Server Controls Tutorial on W3Schools.

After crafting the user interface your page's declarative markup should look something like the following:

```
<h3>Leave a Comment</h3>
<p>
     <b>Subject:</b>
     <asp:RequiredFieldValidator ID="SubjectReqValidator" runat="server"
          ErrorMessage="You must provide a value for Subject"
          ControlToValidate="Subject" ValidationGroup="EnterComment">
     </asp:RequiredFieldValidator><br/>
     <asp:TextBox ID="Subject" Columns="40" runat="server"></asp:TextBox>
</p>
<p>
     <b>Body:</b>
     <asp:RequiredFieldValidator ID="BodyReqValidator" runat="server"
          ControlToValidate="Body"
          ErrorMessage="You must provide a value for Body" ValidationGroup="EnterComment">
     </asp:RequiredFieldValidator><br/>
     <asp:TextBox ID="Body" TextMode="MultiLine" Width="95%"
          Rows="8" runat="server"></asp:TextBox>
</p>
<p>
     <asp:Button ID="PostCommentButton" runat="server"
          Text="Post Your Comment"
          ValidationGroup="EnterComment" />
</p>
```

With the user interface complete, our next task is to insert a new record into the `GuestbookComments` table when the `PostCommentButton` is clicked. This can be accomplished in a number of ways: we can write ADO.NET code in the Button's `Click` event handler; we can add a SqlDataSource control to the page, configure its `InsertCommand`, and then call its `Insert` method from the `Click` event handler; or we could build a middle tier that was responsible for inserting new guestbook comments, and invoke this functionality from the `Click` event handler. Since we looked at using a SqlDataSource in Step 3, let's use ADO.NET code here.

> **NOTE**
>
> The ADO.NET classes used to programmatically access data from a Microsoft SQL Server database are located in the `System.Data.SqlClient` namespace. You may need to import this namespace into your page's code-behind class (i.e., `using System.Data.SqlClient;`).

Create an event handler for the `PostCommentButton`'s `Click` event and add the following code:

```
protected void PostCommentButton_Click(object sender, EventArgs e)
{
     if (!Page.IsValid)
          return;

     // Determine the currently logged on user's UserId
     MembershipUser currentUser = Membership.GetUser();
     Guid currentUserId = (Guid)currentUser.ProviderUserKey;

     // Insert a new record into GuestbookComments
     string connectionString =
          ConfigurationManager.ConnectionStrings["SecurityTutorialsConnectionString"].ConnectionString;
     string insertSql = "INSERT INTO GuestbookComments(Subject, Body, UserId) VALUES(@Subject,
               @Body, @UserId)";

     using (SqlConnection myConnection = new SqlConnection(connectionString))
     {
          myConnection.Open();
          SqlCommand myCommand = new SqlCommand(insertSql, myConnection);
          myCommand.Parameters.AddWithValue("@Subject", Subject.Text.Trim());
          myCommand.Parameters.AddWithValue("@Body", Body.Text.Trim());
          myCommand.Parameters.AddWithValue("@UserId", currentUserId);
          myCommand.ExecuteNonQuery();
          myConnection.Close();
     }

     // "Reset" the Subject and Body TextBoxes
     Subject.Text = string.Empty;
     Body.Text = string.Empty;
}
```

The `Click` event handler starts by checking that the user-supplied data is valid. If it is not, the event handler exits before inserting a record. Assuming the supplied data is valid, the currently logged on user's `UserId` value is retrieved and stored in the `currentUserId` local variable. This value is needed because we must supply a `UserId` value when inserting a record into `GuestbookComments`.

Following that, the connection string for the `SecurityTutorials` database is retrieved from `Web.config` and the `INSERT` SQL statement is specified. A `SqlConnection` object is then created and opened. Next, a `SqlCommand` object is constructed and the values for the parameters used in the `INSERT` query are assigned. The `INSERT` statement is then executed and the connection closed. At the end of the event handler, the `Subject` and `Body` TextBoxes' `Text` properties are cleared out so that the user's values are not persisted across the postback.

Go ahead and test out this page in a browser. Since this page is in the `Membership` folder it is not accessible to anonymous visitors. Therefore, you will need to first log on (if you have not already). Enter a value into the `Subject` and `Body` TextBoxes and click the `PostCommentButton` button. This will cause a new record to be added to `GuestbookComments`. On postback, the subject and body you provided are wiped from the TextBoxes.

After clicking the `PostCommentButton` button there is no visual feedback that the comment was added to the guestbook. We still need to update this page to display the existing guestbook comments, which we will do in Step 5. Once we accomplish that, the just-added comment will appear in the list of comments, providing adequate visual feedback. For now, confirm that your guestbook comment was saved by examining the contents of the `GuestbookComments` table.

Figure 17 shows the contents of the `GuestbookComments` table after two comments have been left.

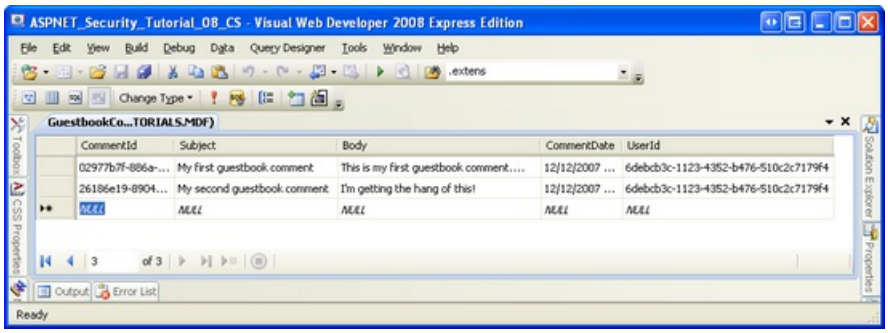


Figure 17: You Can See the Guestbook Comments in the `GuestbookComments` Table (Click to view full-size image)

> **NOTE**
>
> If a user attempts to insert a guestbook comment that contains potentially dangerous markup – such as HTML – ASP.NET will throw an `HttpRequestValidationException`. To learn more about this exception, why it's thrown, and how to permit users to submit potentially dangerous values, consult the Request Validation Whitepaper.

Step 5: Listing the Existing Guestbook Comments

In addition to leaving comments, a user visiting the `Guestbook.aspx` page should also be able to view the guestbook's existing comments. To accomplish this, add a ListView control named `CommentList` to the bottom of the page.

> **NOTE**
>
> The ListView control is new to ASP.NET version 3.5. It is designed to display a list of items in a very customizable and flexible layout, yet still offer built-in editing, inserting, deleting, paging, and sorting functionality like the GridView. If you are using ASP.NET 2.0, you will need to use the DataList or Repeater control instead. For more information on using the ListView, see Scott Guthrie's blog entry, The asp:ListView Control, and my article, Displaying Data with the ListView Control.

Open the ListView's Smart Tag and, from the Choose Data Source drop-down list, bind the control to a new data source. As we saw in Step 2, this will launch the Data Source Configuration Wizard. Select the Database icon, name the resulting SqlDataSource `CommentsDataSource`, and click OK. Next, select the `SecurityTutorialsConnectionString` connection string from the drop-down list and click Next.

At this point in Step 2 we specified the data to query by picking the `UserProfiles` table from the drop-down list and selecting the columns to return (refer back to Figure 9). This time, however, we want to craft a SQL statement that pulls back not only the records from `GuestbookComments`, but also the commenter's home town, homepage, signature, and username. Therefore, select the "Specify a custom SQL statement or stored procedure" radio button and click Next.

This will bring up the "Define Custom Statements or Stored Procedures" screen. Click the Query Builder button to graphically build the query. The Query Builder starts by prompting us to specify the tables we want to query from. Select the `GuestbookComments`, `UserProfiles`, and `aspnet_Users` tables and click OK. This will add all three tables to the design surface. Since there are foreign key constraints amongst the `GuestbookComments`, `UserProfiles`, and `aspnet_Users` tables, the Query Builder automatically `JOIN`s these tables.

All that remains is to specify the columns to return. From the `GuestbookComments` table select the `Subject`, `Body`, and `CommentDate` columns; return the `HomeTown`, `HomepageUrl`, and `Signature` columns from the `UserProfiles` table; and return `UserName` from `aspnet_Users`. Also, add "`ORDER BY CommentDate DESC`" to the end of the `SELECT` query so that the most recent posts are returned first. After making these selections, your Query Builder interface should look similar to the screen shot in Figure 18.

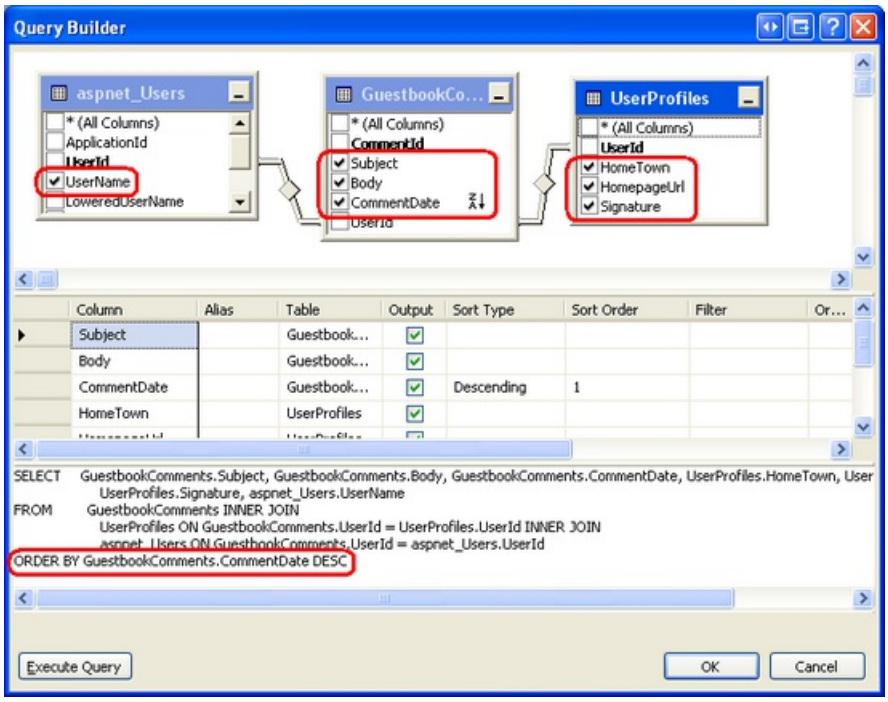


Figure 18: The Constructed Query `JOIN` s the `GuestbookComments`, `UserProfiles`, and `aspnet_Users` Tables (Click to view full-size image)

Click OK to close the Query Builder window and return to the "Define Custom Statements or Stored Procedures" screen. Click Next to advance to the "Test Query" screen, where you may view the query results by clicking the Test Query button. When you're ready, click Finish to complete the Configure Data Source wizard.

When we completed the Configure Data Source wizard in Step 2, the associated DetailsView control's `Fields` collection was updated to include a BoundField for each column returned by the `SelectCommand`. The ListView, however, remains unchanged; we still need to define its layout. The ListView's layout can be constructed manually through its declarative markup or from the "Configure ListView" option in its Smart Tag. I usually prefer defining the markup by hand, but use whatever method is most natural to you.

I ended up using the following `LayoutTemplate`, `ItemTemplate`, and `ItemSeparatorTemplate` for my ListView control:

```
<asp:ListView ID="CommentList" runat="server" DataSourceID="CommentsDataSource">
     <LayoutTemplate>
          <span ID="itemPlaceholder" runat="server" />
          <p>
               <asp:DataPager ID="DataPager1" runat="server">
                    <Fields>
                         <asp:NextPreviousPagerField ButtonType="Button"
                              ShowFirstPageButton="True"
                              ShowLastPageButton="True" />
                    </Fields>
               </asp:DataPager>
          </p>
     </LayoutTemplate>
     <ItemTemplate>
          <h4><asp:Label ID="SubjectLabel" runat="server"
               Text='<%# Eval("Subject") %>' /></h4>
          <asp:Label ID="BodyLabel" runat="server"
               Text='<%# Eval("Body").ToString().Replace(Environment.NewLine, "<br />") %>' />
          <p>
               ---<br />
               <asp:Label ID="SignatureLabel" Font-Italic="true" runat="server"
                    Text='<%# Eval("Signature") %>' />
               <br />
               <br />
               My Home Town:
               <asp:Label ID="HomeTownLabel" runat="server"
                    Text='<%# Eval("HomeTown") %>' />
               <br />
               My Homepage:
               <asp:HyperLink ID="HomepageUrlLink" runat="server"
                    NavigateUrl='<%# Eval("HomepageUrl") %>'
                    Text='<%# Eval("HomepageUrl") %>' />
          </p>
          <p align="center">
               Posted by
               <asp:Label ID="UserNameLabel" runat="server"
                    Text='<%# Eval("UserName") %>' /> on
               <asp:Label ID="CommentDateLabel" runat="server"
                    Text='<%# Eval("CommentDate") %>' />
          </p>
     </ItemTemplate>
     <ItemSeparatorTemplate>
          <hr />
     </ItemSeparatorTemplate>
</asp:ListView>
```

The `LayoutTemplate` defines the markup emitted by the control, while the `ItemTemplate` renders each item returned by the SqlDataSource. The `ItemTemplate`'s resulting markup is placed in the `LayoutTemplate`'s `itemPlaceholder` control. In addition to the `itemPlaceholder`, the `LayoutTemplate` includes a DataPager control, which limits the ListView to showing just 10 guestbook comments per page (the default) and renders a paging interface.

My `ItemTemplate` displays each guestbook comment's subject in an `<h4>` element with the body situated below the subject. Note that syntax used for displaying the body takes the data returned by the `Eval("Body")` databinding statement, converts it to a string, and replaces line breaks with the `<br />` element. This conversion is needed in order to show the line breaks entered when submitting the comment since whitespace is ignored by HTML. The user's signature is displayed beneath the body in italics, followed by the user's home town, a link to his homepage, the date and time the comment was made, and the username of the person who left the comment.

Take a moment to view the page through a browser. You should see the comments that you added to the guestbook in Step 5 displayed here.

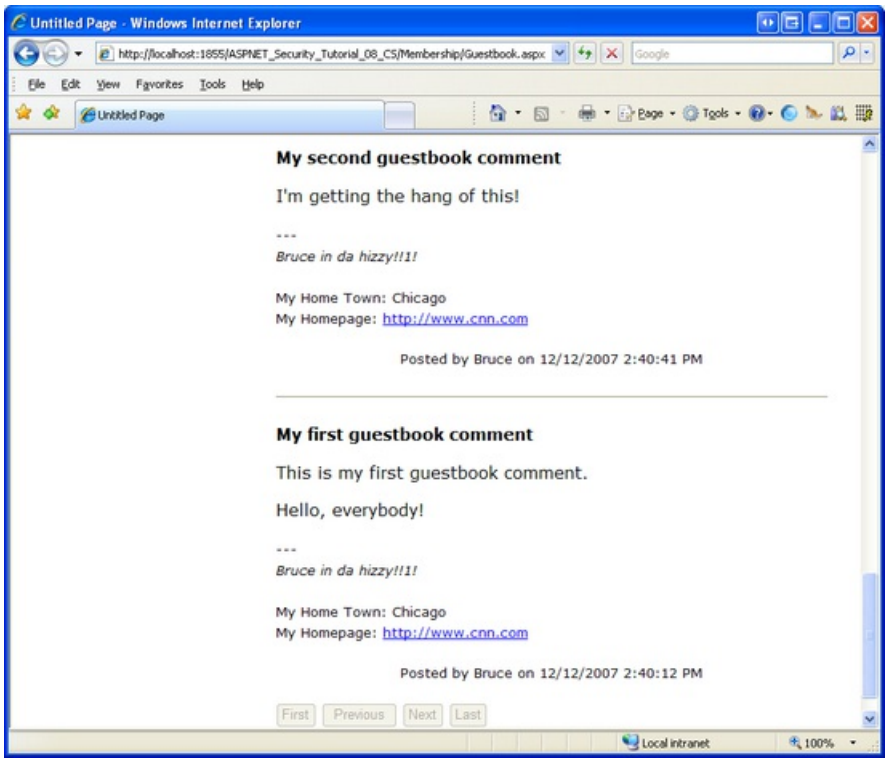


Figure 19: `Guestbook.aspx` Now Displays the Guestbook's Comments (Click to view full-size image)

Try adding a new comment to the guestbook. Upon clicking the `PostCommentButton` button the page posts back and the comment is added to the database, but the ListView control is not updated to show the new comment. This can be fixed by either:

- Updating the `PostCommentButton` button's `Click` event handler so that it invokes the ListView control's `DataBind()` method after inserting the new comment into the database, or
- Setting the ListView control's `EnableViewState` property to `false`. This approach works because by disabling the control's view state, it must rebind to the underlying data on every postback.

The tutorial website downloadable from this tutorial illustrates both techniques. The ListView control's `EnableViewState` property to `false` and the code needed to programmatically rebind the data to the ListView is present in the `Click` event handler, but is commented out.

> **NOTE**
>
> Currently the `AdditionalUserInfo.aspx` page allows the user to view and edit their home town, homepage, and signature settings. It might be nice to update `AdditionalUserInfo.aspx` to display the logged in user's guestbook comments. That is, in addition to examining and modifying her information, a user can visit the `AdditionalUserInfo.aspx` page to see what guestbook comments she's made in the past. I leave this as an exercise for the interested reader.

Step 6: Customizing the CreateUserWizard Control to Include an Interface for the Home Town, Homepage, and Signature

The `SELECT` query used by the `Guestbook.aspx` page uses an `INNER JOIN` to combine the related records amongst the `GuestbookComments`, `UserProfiles`, and `aspnet_Users` tables. If a user that has no record in `UserProfiles` makes a guestbook comment, the comment won't be displayed in the ListView because the `INNER JOIN` only returns `GuestbookComments` records when there are matching records in `UserProfiles` and `aspnet_Users`. And as we saw in Step 3, if a user does not have a record in `UserProfiles` she cannot view or edit her settings in the `AdditionalUserInfo.aspx` page.

Needless to say, due to our design decisions it is important that every user account in the Membership system have a matching record in the `UserProfiles` table. What we want is for a corresponding record to be added to `UserProfiles` whenever a new Membership user account is created through the CreateUserWizard.

As discussed in the *Creating User Accounts* tutorial, after the new Membership user account is created the CreateUserWizard control raises its `CreatedUser` event. We can create an event handler for this event, get the UserId for the just-created user, and then insert a record into the `UserProfiles` table with default values for the `HomeTown`, `HomepageUrl`, and `Signature` columns. What's more, it is possible to prompt the user for these values by customizing the CreateUserWizard control's interface to include additional TextBoxes.

Let's first look at how to add a new row to the `UserProfiles` table in the `CreatedUser` event handler with default values. Following that, we will see how to customize the CreateUserWizard control's user interface to include additional form fields to collect the new user's home town, homepage, and signature.

Adding a Default Row to `UserProfiles`

In the *Creating User Accounts* tutorial we added a CreateUserWizard control to the `CreatingUserAccounts.aspx` page in the `Membership` folder. In order to have the CreateUserWizard control add a record to `UserProfiles` table upon user account creation, we need to update the CreateUserWizard control's functionality. Rather than making these changes to the `CreatingUserAccounts.aspx` page, let's instead add a new CreateUserWizard control to `EnhancedCreateUserWizard.aspx` page and make the modifications for this tutorial there.

Open the `EnhancedCreateUserWizard.aspx` page in Visual Studio and drag a CreateUserWizard control from the Toolbox onto the page. Set the CreateUserWizard control's `ID` property to `NewUserWizard`. As we discussed in the *Creating User Accounts* tutorial, the CreateUserWizard's default user interface prompts the visitor for the necessary information. Once this information has been supplied, the control internally creates a new user account in the Membership framework, all without us having to write a single line of code.

The CreateUserWizard control raises a number of events during its workflow. After a visitor supplies the request information and submits the form, the CreateUserWizard control initially fires its `CreatingUser` event. If there is a problem during the create process, the `CreateUserError` event is fired; however, if the user is successfully created, then the `CreatedUser` event is raised. In the *Creating User Accounts* tutorial we created an event handler for the `CreatingUser` event to ensure that the supplied username did not contain any leading or trailing spaces, and that the username did not appear anywhere in the password.

In order to add a row in the `UserProfiles` table for the just-created user, we need to create an event handler for the `CreatedUser` event. By the time the `CreatedUser` event has fired, the user account has already been created in the Membership framework, enabling us to retrieve the account's UserId value.

Create an event handler for the `NewUserWizard`'s `CreatedUser` event and add the following code:

```
protected void NewUserWizard_CreatedUser(object sender, EventArgs e)
{
     // Get the UserId of the just-added user
     MembershipUser newUser = Membership.GetUser(NewUserWizard.UserName);
     Guid newUserId = (Guid)newUser.ProviderUserKey;

     // Insert a new record into UserProfiles
     string connectionString =
          ConfigurationManager.ConnectionStrings["SecurityTutorialsConnectionString"].ConnectionString;
     string insertSql = "INSERT INTO UserProfiles(UserId, HomeTown, HomepageUrl,
          Signature) VALUES(@UserId, @HomeTown, @HomepageUrl, @Signature)";

     using (SqlConnection myConnection = new SqlConnection(connectionString))
     {
          myConnection.Open();
          SqlCommand myCommand = new SqlCommand(insertSql, myConnection);
          myCommand.Parameters.AddWithValue("@UserId", newUserId);
          myCommand.Parameters.AddWithValue("@HomeTown", DBNull.Value);
          myCommand.Parameters.AddWithValue("@HomepageUrl", DBNull.Value);
          myCommand.Parameters.AddWithValue("@Signature", DBNull.Value);
          myCommand.ExecuteNonQuery();
          myConnection.Close();
     }
}
```

The above code beings by retrieving the UserId of the just-added user account. This is accomplished by using the `Membership.GetUser(username)` method to return information about a particular user, and then using the `ProviderUserKey` property to retrieve their UserId. The username entered by the user in the CreateUserWizard control is available via its `UserName` property.

Next, the connection string is retrieved from `Web.config` and the `INSERT` statement is specified. The necessary ADO.NET objects are instantiated and the command executed. The code assigns a `DBNull` instance to the `@HomeTown`, `@HomepageUrl`, and `@Signature` parameters, which has the effect of inserting database `NULL` values for the `HomeTown`, `HomepageUrl`, and `Signature` fields.

Visit the `EnhancedCreateUserWizard.aspx` page through a browser and create a new user account. After doing so, return to Visual Studio and examine the contents of the `aspnet_Users` and `UserProfiles` tables (like we did back in Figure 12). You should see the new user account in `aspnet_Users` and a corresponding `UserProfiles` row (with `NULL` values for `HomeTown`, `HomepageUrl`, and `Signature`).

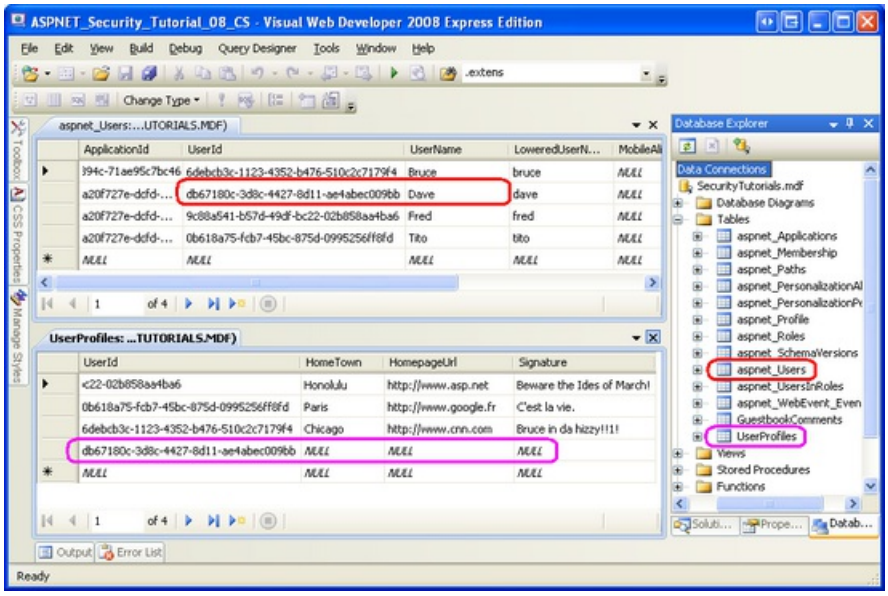


Figure 20: A New User Account and `UserProfiles` Record Have Been Added (Click to view full-size image)

After the visitor has supplied his new account information and clicked the "Create User" button, the user account is created and a row added to the `UserProfiles` table. The CreateUserWizard then displays its `CompleteWizardStep`, which displays a success message and a Continue button. Clicking the Continue button causes a postback, but no action is taken, leaving the user stuck on the `EnhancedCreateUserWizard.aspx` page.

We can specify a URL to send the user to when the Continue button is clicked via the CreateUserWizard control's `ContinueDestinationPageUrl` property. Set the `ContinueDestinationPageUrl` property to "~/Membership/AdditionalUserInfo.aspx". This takes the new user to `AdditionalUserInfo.aspx`, where they can view and update their settings.

Customizing the CreateUserWizard's Interface to Prompt for the New User's Home Town, Homepage, and Signature

The CreateUserWizard control's default interface is sufficient for simple account creation scenarios where only core user account information like username, password, and email need be collected. But what if we wanted to prompt the visitor to enter her home town, homepage, and signature while creating her account? It is possible to customize the CreateUserWizard control's interface to collect additional information at signup, and this information may be used in the `CreatedUser` event handler to insert additional records into the underlying database.

The CreateUserWizard control extends the ASP.NET Wizard control, which is a control that allows a page developer to define a series of ordered `WizardSteps`. The Wizard control renders the active step and provides a navigation interface that allows the visitor to move through these steps. The Wizard control is ideal for breaking down a long task into several short steps. For more information on the Wizard control, see Creating a Step-by-Step User Interface with the ASP.NET 2.0 Wizard Control.

The CreateUserWizard control's default markup defines two `WizardSteps`: `CreateUserWizardStep` and `CompleteWizardStep`.

```
<asp:CreateUserWizard ID="NewUserWizard" runat="server"
     ContinueDestinationPageUrl="~/Membership/AdditionalUserInfo.aspx">
     <WizardSteps>
          <asp:CreateUserWizardStep ID="CreateUserWizardStep1" runat="server">
          </asp:CreateUserWizardStep>
          <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">
          </asp:CompleteWizardStep>
     </WizardSteps>
</asp:CreateUserWizard>
```

The first `WizardStep`, `CreateUserWizardStep`, renders the interface that prompts for the username, password, email, and so on. After the visitor supplies this information and clicks "Create User", she is shown the `CompleteWizardStep`, which shows the success message and a Continue button.

To customize the CreateUserWizard control's interface to include additional form fields, we can:

- **Create one or more new `WizardStep`s to contain the additional user interface elements**. To add a new `WizardStep` to the CreateUserWizard, click the "Add/Remove `WizardSteps`" link from its Smart Tag to launch the `WizardStep` Collection Editor. From there you can add, remove, or reorder the steps in the wizard. This is the approach we will use for this tutorial.
- **Convert the `CreateUserWizardStep` into an editable `WizardStep`**. This replaces the `CreateUserWizardStep` with an equivalent `WizardStep` whose markup defines a user interface that matches the `CreateUserWizardStep`'s. By converting the `CreateUserWizardStep` into a `WizardStep` we can reposition the controls or add additional user interface elements to this step. To convert the `CreateUserWizardStep` or `CompleteWizardStep` into an editable `WizardStep`, click the "Customize Create User Step" or "Customize Complete Step" link from the control's Smart Tag.

- Use some combination of the above two options.

One important thing to keep in mind is that the CreateUserWizard control executes its user account creation process when the "Create User" button is clicked from within its `CreateUserWizardStep`. It doesn't matter if there are additional `WizardStep` s after the `CreateUserWizardStep` or not.

When adding a custom `WizardStep` to the CreateUserWizard control to collect additional user input, the custom `WizardStep` can be placed before or after the `CreateUserWizardStep`. If it comes before the `CreateUserWizardStep` then the additional user input collected from the custom `WizardStep` is available for the `CreatedUser` event handler. However, if the custom `WizardStep` comes after `CreateUserWizardStep` then by the time the custom `WizardStep` is displayed the new user account has already been created and the `CreatedUser` event has already fired.

Figure 21 shows the workflow when the added `WizardStep` precedes the `CreateUserWizardStep`. Since the additional user information has been collected by the time the `CreatedUser` event fires, all we have to do is update the `CreatedUser` event handler to retrieve these inputs and use those for the `INSERT` statement's parameter values (rather than `DBNull.Value`).

Figure 21: The CreateUserWizard Workflow When an Additional `WizardStep` Precedes the `CreateUserWizardStep` (Click to view full-size image)

If the custom `WizardStep` is placed *after* the `CreateUserWizardStep`, however, the create user account process occurs before the user has had a chance to enter her home town, homepage, or signature. In such a case, this additional information needs to be inserted into the database after the user account has been created, as Figure 22 illustrates.

Figure 22: The CreateUserWizard Workflow When an Additional `WizardStep` Comes After the `CreateUserWizardStep` (Click to view full-size image)

The workflow shown in Figure 22 waits to insert a record into the `UserProfiles` table until after Step 2 completes. If the visitor closes her browser after step 1, however, we will have reached a state where a user account was created, but no record was added to `UserProfiles`. One workaround is to have a record with `NULL` or default values inserted into `UserProfiles` in the `CreatedUser` event handler (which fires after step 1), and then update this record after step 2 completes. This ensures that a `UserProfiles` record will be added for the user account even if the user quits the registration process midway through.

For this tutorial let's create a new `WizardStep` that occurs after the `CreateUserWizardStep` but before the `CompleteWizardStep`. Let's first get the WizardStep in place and configured and then we'll look at the code.

From the CreateUserWizard control's Smart Tag, select the "Add/Remove `WizardStep` s", which brings up the `WizardStep` Collection Editor dialog. Add a new `WizardStep`, setting its `ID` to `UserSettings`, its `Title` to "Your Settings" and its `StepType` to `Step`. Then position it so that it comes after the `CreateUserWizardStep` ("Sign Up for Your New Account") and before the `CompleteWizardStep` ("Complete"), as shown in Figure 23.

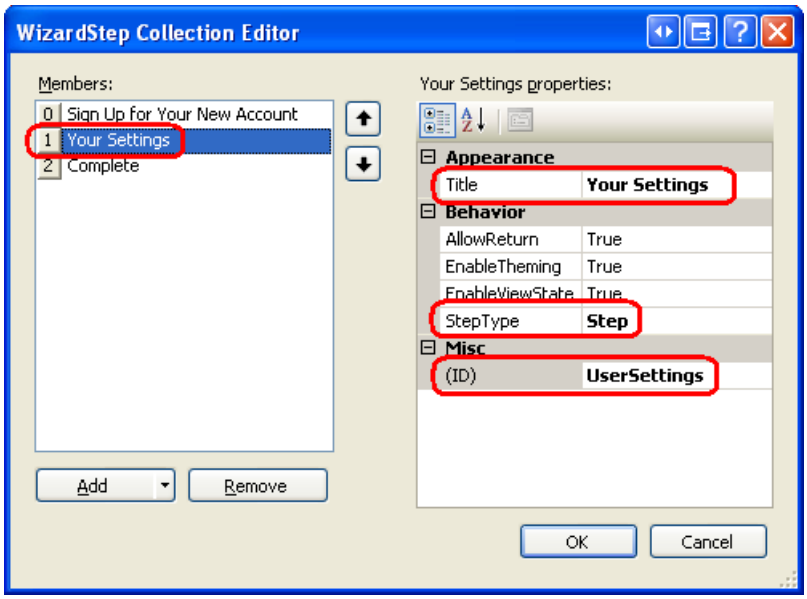


Figure 23: Add a New `WizardStep` to the CreateUserWizard Control (Click to view full-size image)

Click OK to close the `WizardStep` Collection Editor dialog. The new `WizardStep` is evidenced by the CreateUserWizard control's updated declarative markup:

```
<asp:CreateUserWizard ID="NewUserWizard" runat="server"
     ContinueDestinationPageUrl="~/Membership/AdditionalUserInfo.aspx">
     <WizardSteps>
          <asp:CreateUserWizardStep ID="CreateUserWizardStep1" runat="server">
          </asp:CreateUserWizardStep>
          <asp:WizardStep runat="server" ID="UserSettings" StepType="Step"
               Title="Your Settings">
          </asp:WizardStep>
          <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">
          </asp:CompleteWizardStep>
     </WizardSteps>
</asp:CreateUserWizard>
```

Note the new `<asp:WizardStep>` element. We need to add the user interface to collect the new user's home town, homepage, and signature here. You can enter this content in the declarative syntax or through the Designer. To use the Designer, select the "Your Settings" step from the drop-down list in the Smart Tag to see the step in the Designer.

> **NOTE**
>
> Selecting a step through the Smart Tag's drop-down list updates the CreateUserWizard control's `ActiveStepIndex` property, which specifies the index of the starting step. Therefore, if you use this drop-down list to edit the "Your Settings" step in the Designer, be sure to set it back to "Sign Up for Your New Account" so that this step is shown when users first visit the `EnhancedCreateUserWizard.aspx` page.

Create a user interface within the "Your Settings" step that contains three TextBox controls named `HomeTown`, `HomepageUrl`, and `Signature`. After constructing this interface, the CreateUserWizard's declarative markup should look similar to the following:

```
<asp:CreateUserWizard ID="NewUserWizard" runat="server"
     ContinueDestinationPageUrl="~/Membership/AdditionalUserInfo.aspx">
     <WizardSteps>
          <asp:CreateUserWizardStep ID="CreateUserWizardStep1" runat="server">
          </asp:CreateUserWizardStep>
          <asp:WizardStep runat="server" ID="UserSettings" StepType="Step"
               Title="Your Settings">
               <p>
                    <b>Home Town:</b><br />
                    <asp:TextBox ID="HomeTown" runat="server"></asp:TextBox>
               </p>
               <p>
                    <b>Homepage URL:</b><br />
                    <asp:TextBox ID="HomepageUrl" Columns="40" runat="server"></asp:TextBox>
               </p>
               <p>
                    <b>Signature:</b><br />
                    <asp:TextBox ID="Signature" TextMode="MultiLine" Width="95%"
                         Rows="5" runat="server"></asp:TextBox>
               </p>
          </asp:WizardStep>
          <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">
          </asp:CompleteWizardStep>
     </WizardSteps>
</asp:CreateUserWizard>
```

Go ahead and visit this page through a browser and create a new user account, specifying values for the home town, homepage, and signature. After completing the `CreateUserWizardStep` the user account is created in the Membership framework and the `CreatedUser` event handler runs, which adds a new row to `UserProfiles`, but with a database `NULL` value for `HomeTown`, `HomepageUrl`, and `Signature`. The values entered for the home town, homepage, and signature are never used. The net result is a new user account with a `UserProfiles` record whose `HomeTown`, `HomepageUrl`, and `Signature` fields have yet to be specified.

We need to execute code after the "Your Settings" step that takes the home town, honepage, and signature values entered by the user and updates the appropriate `UserProfiles` record. Each time the user moves between steps in a Wizard control, the Wizard's `ActiveStepChanged` event fires. We can create an event handler for this event and update the `UserProfiles` table when the "Your Settings" step has completed.

Add an event handler for the CreateUserWizard's `ActiveStepChanged` event and add the following code:

```
protected void NewUserWizard_ActiveStepChanged(object sender, EventArgs e)
{
     // Have we JUST reached the Complete step?
     if (NewUserWizard.ActiveStep.Title == "Complete")
     {
          WizardStep UserSettings = NewUserWizard.FindControl("UserSettings") as
          WizardStep;

          // Programmatically reference the TextBox controls
          TextBox HomeTown = UserSettings.FindControl("HomeTown") as TextBox;
          TextBox HomepageUrl = UserSettings.FindControl("HomepageUrl") as TextBox;
          TextBox Signature = UserSettings.FindControl("Signature") as TextBox;

          // Update the UserProfiles record for this user
          // Get the UserId of the just-added user
          MembershipUser newUser = Membership.GetUser(NewUserWizard.UserName);
          Guid newUserId = (Guid)newUser.ProviderUserKey;

          // Insert a new record into UserProfiles
          string connectionString =
               ConfigurationManager.ConnectionStrings["SecurityTutorialsConnectionString"].ConnectionString;
          string updateSql = "UPDATE UserProfiles SET HomeTown = @HomeTown, HomepageUrl
               = @HomepageUrl, Signature = @Signature WHERE UserId = @UserId";

          using (SqlConnection myConnection = new SqlConnection(connectionString))
          {
               myConnection.Open();
               SqlCommand myCommand = new SqlCommand(updateSql, myConnection);
               myCommand.Parameters.AddWithValue("@HomeTown", HomeTown.Text.Trim());
               myCommand.Parameters.AddWithValue("@HomepageUrl", HomepageUrl.Text.Trim());
               myCommand.Parameters.AddWithValue("@Signature", Signature.Text.Trim());
               myCommand.Parameters.AddWithValue("@UserId", newUserId);
               myCommand.ExecuteNonQuery();
               myConnection.Close();
          }
     }
}
```

The above code starts by determining if we have just reached the "Complete" step. Since the "Complete" step occurs immediately after the "Your Settings" step, then when the visitor reaches "Complete" step that means she just finished the "Your Settings" step.

In such a case, we need to programmatically reference the TextBox controls within the `UserSettings WizardStep`. This is accomplished by first using the `FindControl` method to programmatically referencing the `UserSettings WizardStep`, and then again to reference the TextBoxes from within the `WizardStep`. Once the TextBoxes have been referenced, we're ready to execute the `UPDATE` statement. The `UPDATE` statement has the same number of parameters as the `INSERT` statement in the `CreatedUser` event handler, but here we use the home town, homepage, and signature values supplied by the user.

With this event handler in place, visit the `EnhancedCreateUserWizard.aspx` page through a browser and create a new user account specifying values for the home town, homepage, and signature. After creating the new account you should be redirected to the `AdditionalUserInfo.aspx` page, where the just-entered home town, homepage, and signature information is displayed.

NOTE

Our website currently has two pages from which a visitor can create a new account: `CreatingUserAccounts.aspx` and `EnhancedCreateUserWizard.aspx`. The website's sitemap and login page point to the `CreatingUserAccounts.aspx` page, but the `CreatingUserAccounts.aspx` page does not prompt the user for their home town, homepage, and signature information and does not add a corresponding row to `UserProfiles`. Therefore, either update the `CreatingUserAccounts.aspx` page so that it offers this functionality or update the sitemap and login page to reference `EnhancedCreateUserWizard.aspx` instead of `CreatingUserAccounts.aspx`. If you choose the latter option, be sure to update the `Membership` folder's `Web.config` file so as to allow anonymous users access to the `EnhancedCreateUserWizard.aspx` page.

Summary

In this tutorial we looked at techniques for modeling data that is related to user accounts within the Membership framework. In particular, we looked at modeling entities that share a one-to-many relationship with user accounts as well as data that shares a one-to-one relationship. Furthermore, we saw how this related information could be displayed, inserted, and updated, with some examples using the SqlDataSource control and others using ADO.NET code.

This tutorial completes our look at user accounts. Starting with the next tutorial we will turn our attention to roles. Over the next several tutorials we will look at the Roles framework, see how to create new roles, how to assign roles to users, how to determine what roles a user belongs to, and how to apply role-based authorization.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Accessing and Updating Data in ASP.NET 2.0
- ASP.NET 2.0 Wizard Control
- Creating a Step-by-Step User Interface with the ASP.NET 2.0 Wizard Control
- Creating Custom DataSource Control Parameters
- Customizing the CreateUserWizard Control
- DetailsView Control QuickStarts
- Displaying Data with the ListView Control
- Dissecting the Validation Controls in ASP.NET 2.0
- Editing Insert and Deleting Data
- Form Validation in ASP.NET
- Gathering Custom User Registration Information
- Profiles in ASP.NET 2.0
- The asp:ListView Control
- User Profiles QuickStart

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

Creating the Membership Schema in SQL Server (VB)

9/16/2020 • 27 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

This tutorial starts by examining techniques for adding the necessary schema to the database in order to use the SqlMembershipProvider. Following that, we will examine the key tables in the schema and discuss their purpose and importance. This tutorial ends with a look at how to tell an ASP.NET application which provider the Membership framework should use.

Introduction

The previous two tutorials examined using forms authentication to identify website visitors. The forms authentication framework makes it easy for developers to log a user into a website and to remember them across page visits through the use of authentication tickets. The `FormsAuthentication` class includes methods for generating the ticket and adding it to the visitor's cookies. The `FormsAuthenticationModule` examines all incoming requests and, for those with a valid authentication ticket, creates and associates a `GenericPrincipal` and a `FormsIdentity` object with the current request. Forms authentication is merely a mechanism for granting an authentication ticket to a visitor when logging in and, on subsequent requests, parsing that ticket to determine the user's identity. For a web application to support user accounts, we still need to implement a user store and add functionality to validate credentials, register new users, and the myriad of other user account-related tasks.

Prior to ASP.NET 2.0, developers were on the hook for implementing all of these user account-related tasks. Fortunately the ASP.NET team recognized this shortcoming and introduced the Membership framework with ASP.NET 2.0. The Membership framework is a set of classes in the .NET Framework that provide a programmatic interface for accomplishing core user account-related tasks. This framework is built atop the provider model, which allows developers to plug a customized implementation into a standardized API.

As discussed in the *Security Basics and ASP.NET Support* tutorial, the .NET Framework ships with two built-in Membership providers: `ActiveDirectoryMembershipProvider` and `SqlMembershipProvider`. As its name implies, the `SqlMembershipProvider` uses a Microsoft SQL Server database as the user store. In order to use this provider in an application, we need to tell the provider what database to use as the store. As you might imagine, the `SqlMembershipProvider` expects the user store database to have certain database tables, views, and stored procedures. We need to add this expected schema to the selected database.

This tutorial starts by examining techniques for adding the necessary schema to the database in order to use the `SqlMembershipProvider`. Following that, we will examine the key tables in the schema and discuss their purpose and importance. This tutorial ends with a look at how to tell an ASP.NET application which provider the Membership framework should use.

Let's get started!

Step 1: Deciding Where to Place the User Store

An ASP.NET application's data is commonly stored in a number of tables in a database. When implementing the `SqlMembershipProvider` database schema we must decide whether to place the Membership schema in the

same database as the application data or in an alternate database.

I recommend locating the Membership schema in the same database as the application data for the following reasons:

- **Maintainability** an application whose data is encapsulated in one database is easier to understand, maintain, and deploy than an application that has two separate databases.
- **Relational Integrity** by locating the Membership-related tables in the same database as the application tables it is possible to establish foreign key constraints between the primary keys in the Membership-related tables and related application tables.

Decoupling the user store and application data into separate databases only makes sense if you have multiple applications that each use separate databases, but need to share a common user store.

Creating a Database

The application we have been building since the second tutorial has not yet needed a database. We need one now, however, for the user store. Let's create one and then add to it the schema required by the `SqlMembershipProvider` provider (see Step 2).

> **NOTE**
>
> Throughout this tutorial series we will be using a Microsoft SQL Server 2005 Express Edition database to store our application tables and the `SqlMembershipProvider` schema. This decision was made for two reasons: first, due to its cost - free - the Express Edition is the most readably accessible version of SQL Server 2005; second, SQL Server 2005 Express Edition databases can be placed directly in the web application's `App_Data` folder, making it a cinch to package the database and web application together in one ZIP file and to redeploy it without any special setup instructions or configuration options. If you'd prefer to follow along using a non-Express Edition version of SQL Server, feel free. The steps are virtually identical. The `SqlMembershipProvider` schema will work with any version of Microsoft SQL Server 2000 and up.

From the Solution Explorer, right-click on the `App_Data` folder and choose to Add New Item. (If you do not see an `App_Data` folder in your project, right-click on the project in Solution Explorer, select Add ASP.NET Folder, and pick `App_Data`.) From the Add New Item dialog box, choose to add a new SQL Database named `SecurityTutorials.mdf`. In this tutorial we will add the `SqlMembershipProvider` schema to this database; in subsequent tutorials we will create additional tables to capture our application data.

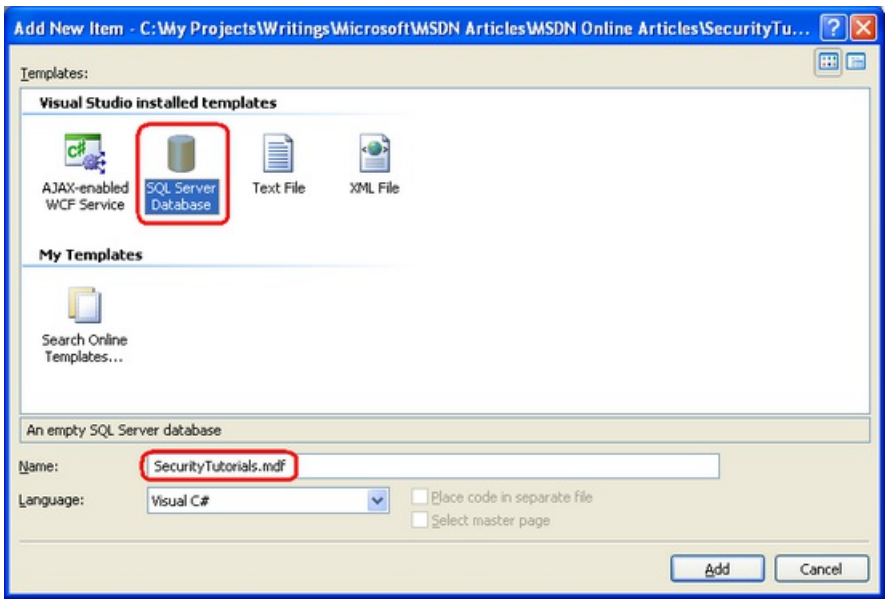


Figure 1: Add a New SQL Database Named `SecurityTutorials.mdf` Database to the `App_Data` Folder (Click to view full-size image)

Adding a database to the `App_Data` folder automatically includes it in the Database Explorer view. (In the non-Express Edition version of Visual Studio, the Database Explorer is called the Server Explorer.) Go to the Database Explorer and expand the just-added `SecurityTutorials` database. If you do not see the Database Explorer on screen, go to the View menu and choose Database Explorer, or hit Ctrl+Alt+S. As Figure 2 shows, the `SecurityTutorials` database is empty - it contains no tables, no views, and no stored procedures.

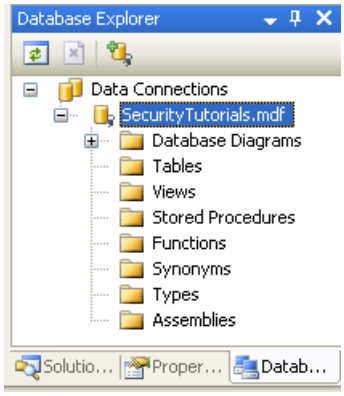


Figure 2: The `SecurityTutorials` Database is Currently Empty (Click to view full-size image)

Step 2: Adding the `SqlMembershipProvider` Schema to the Database

The `SqlMembershipProvider` requires a particular set of tables, views, and stored procedures to be installed in the user store database. These requisite database objects can be added using the `aspnet_regsql.exe` tool. This file is located in the `%WINDIR%\Microsoft.Net\Framework\v2.0.50727\` folder.

> **NOTE**
>
> The `aspnet_regsql.exe` tool offers both command line functionality and a graphical user interface. The graphical interface is more user friendly and is what we will examine in this tutorial. The command line interface is useful when the addition of the `SqlMembershipProvider` schema needs to be automated, such as in build scripts or automated testing scenarios.

The `aspnet_regsql.exe` tool is used to add or remove *ASP.NET application services* to a specified SQL Server database. The ASP.NET application services encompass the schemas for the `SqlMembershipProvider` and `SqlRoleProvider`, along with the schemas for the SQL-based providers for other ASP.NET 2.0 frameworks. We need to provide two bits of information to the `aspnet_regsql.exe` tool:

- Whether we want to add or remove application services, and
- The database from which to add or remove the application services schema

In prompting for the database to use, the `aspnet_regsql.exe` tool asks us to provide the name of the server the database resides on, the security credentials for connecting to the database, and the database name. If you are using the non-Express Edition of SQL Server, you should already know this information, as it is the same information you must provide through a connection string when working with the database through an ASP.NET web page. Determining the server and database name when using a SQL Server 2005 Express Edition database in the `App_Data` folder, however, is a bit more involved.

The following section examines a straightforward way for specifying the server and database name for a SQL Server 2005 Express Edition database in the `App_Data` folder. If you are not using SQL Server 2005 Express Edition feel free to skip ahead to the Installing the Application Services section.

Determining the Server and Database Name for a SQL Server 2005 Express Edition Database in the `App_Data` Folder

In order to use the `aspnet_regsql.exe` tool we need to know the server and database names. The server name

is `localhost\InstanceName`. Most likely, the *InstanceName* is `SQLExpress`. However, if you installed SQL Server 2005 Express Edition manually (that is, you did not install it automatically while installing Visual Studio), then it is possible that you selected a different instance name.

The database name is a bit trickier to determine. Databases in the `App_Data` folder typically have a database name that includes a globally unique identifier along with the path to the database file. We need to determine this database name in order to add the application services schema through `aspnet_regsql.exe`.

The easiest way to ascertain the database name is to examine it through SQL Server Management Studio. SQL Server Management Studio provides a graphical interface for managing SQL Server 2005 databases, but it does not ship with the Express Edition of SQL Server 2005. The good news is that you can download the free Express Edition of SQL Server Management Studio.

> **NOTE**
> If you also have a non-Express Edition version of SQL Server 2005 installed on your desktop then the full version of Management Studio is likely installed. You can use the full version to determine the database name, following the same steps as outlined below for the Express Edition.

Start by closing Visual Studio to ensure that any locks imposed by Visual Studio on the database file are closed. Next, launch SQL Server Management Studio and connect to the `localhost\InstanceName` database for SQL Server 2005 Express Edition. As noted earlier, chances are the instance name is `SQLExpress`. For the Authentication option, select Windows Authentication.

Figure 3: Connect to the SQL Server 2005 Express Edition Instance (Click to view full-size image)

After connecting to the SQL Server 2005 Express Edition instance, Management Studio displays folders for the Databases, the Security settings, the Server Objects, and so on. If you expand the Databases tab you will see that the `SecurityTutorials.mdf` database is *not* registered in the database instance - we need to attach the database first.

Right-click on the Databases folder and choose Attach from the context menu. This will display the Attach Databases dialog box. From here, click the Add button, browse to the `SecurityTutorials.mdf` database, and click OK. Figure 4 shows the Attach Databases dialog box after the `SecurityTutorials.mdf` database has been selected. Figure 5 shows Management Studio's Object Explorer after the database has been successfully attached.

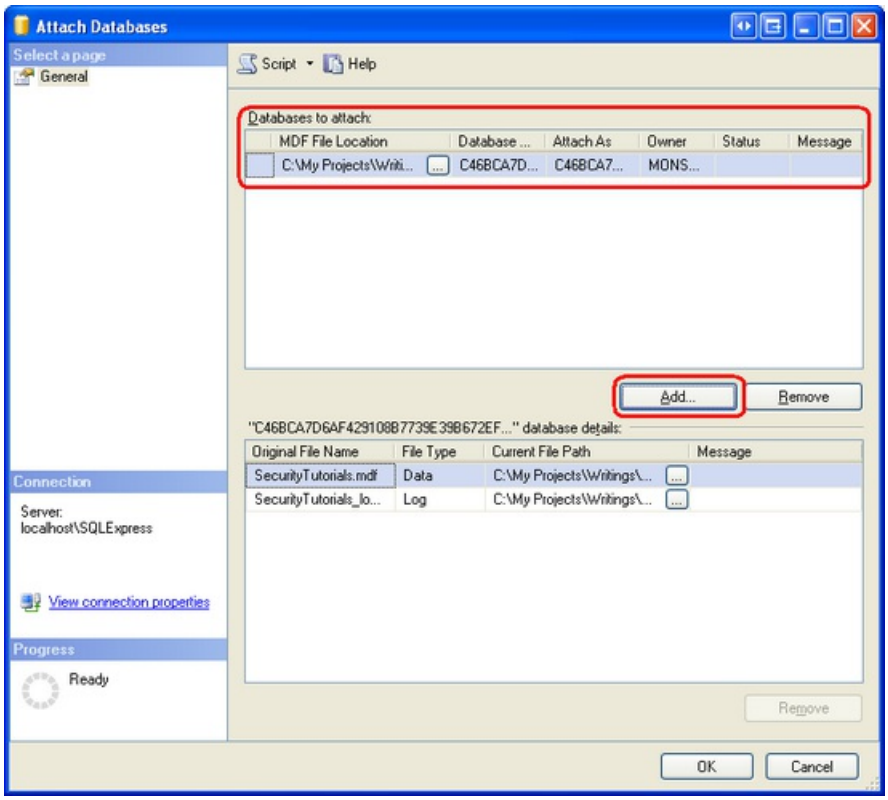


Figure 4: Attach the `SecurityTutorials.mdf` Database (Click to view full-size image)

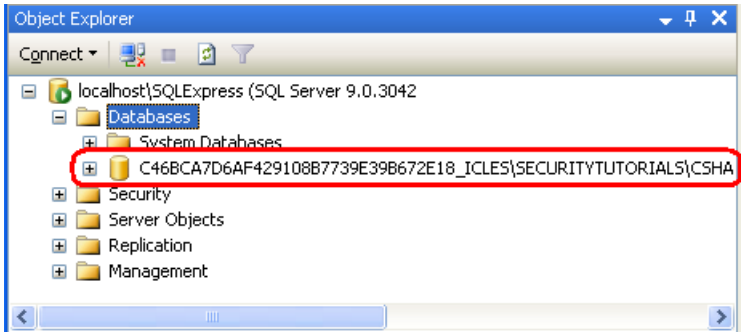


Figure 5: The `SecurityTutorials.mdf` Database Appears in the Databases Folder (Click to view full-size image)

As Figure 5 shows, the `SecurityTutorials.mdf` database has a rather abstruse name. Let's change it to a more memorable (and easier to type) name. Right-click on the database, choose Rename from the context menu, and rename it `SecurityTutorialsDatabase`. This does not change the filename, just the name the database uses to identify itself to SQL Server.

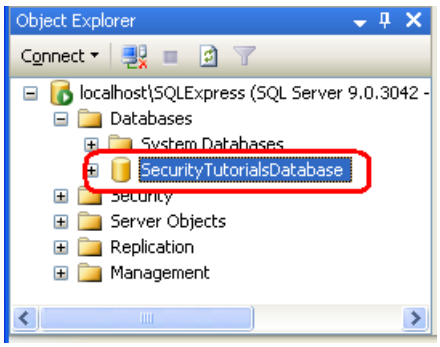


Figure 6: Rename the Database to `SecurityTutorialsDatabase` (Click to view full-size image)

At this point we know the server and database names for the `SecurityTutorials.mdf` database file: `localhost\InstanceName` and `SecurityTutorialsDatabase`, respectively. We are now ready to install the application services through the `aspnet_regsql.exe` tool.

Installing the Application Services

To launch the `aspnet_regsql.exe` tool, go to the start menu and choose Run. Enter `%WINDIR%\Microsoft.Net\Framework\v2.0.50727\aspnet_regsql.exe` into the textbox and click OK. Alternatively, you can use Windows Explorer to drill down to the appropriate folder and double-click the `aspnet_regsql.exe` file. Either approach will net the same results.

Running the `aspnet_regsql.exe` tool without any command line arguments launches the ASP.NET SQL Server Setup Wizard graphical user interface. The wizard makes it easy to add or remove the ASP.NET application services on a specified database. The first screen of the wizard, shown in Figure 7, describes the purpose of the tool.

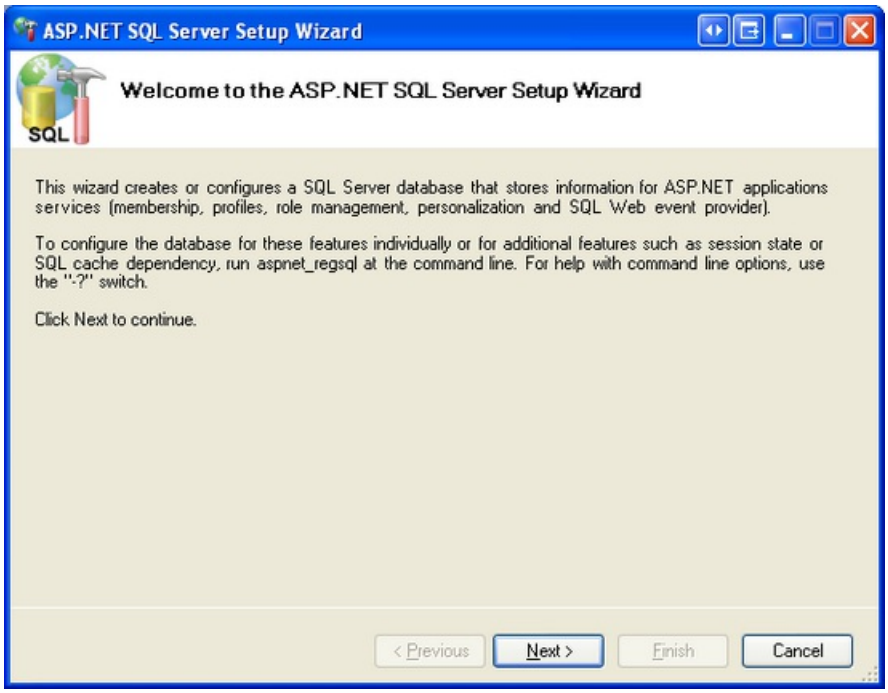


Figure 7: Use the ASP.NET SQL Server Setup Wizard Makes to Add the Membership Schema (Click to view full-size image)

The second step in the wizard asks us whether we want to add the application services or remove them. Since we want to add the tables, views, and stored procedures necessary for the `SqlMembershipProvider`, choose the Configure SQL Server for application services option. Later, if you want to remove this schema from your database, re-run this wizard, but instead choose the Remove application services information from an existing database option.

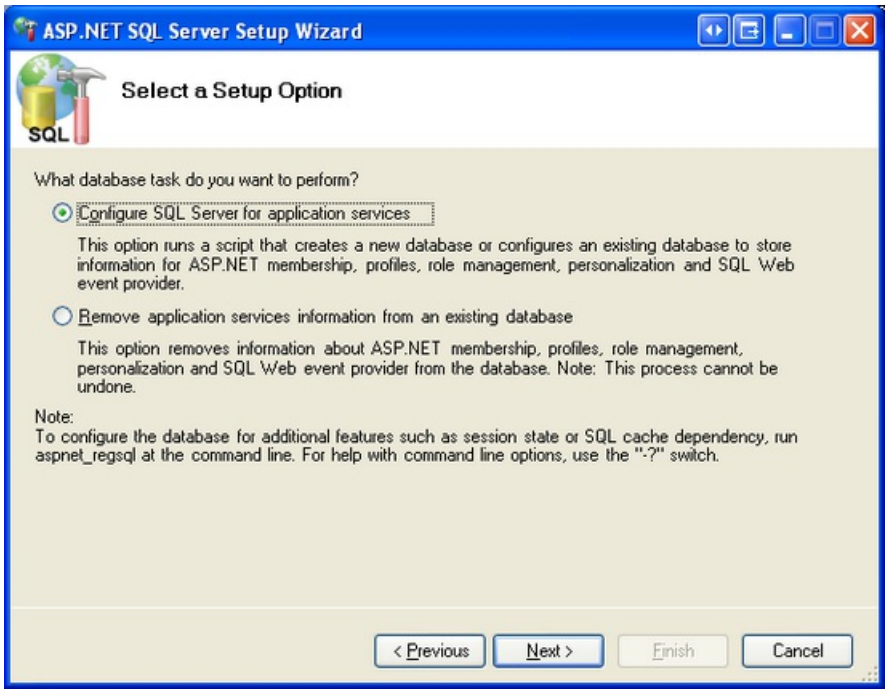


Figure 8: Choose the Configure SQL Server for Application Services Option (Click to view full-size image)

The third step prompts for the database information: the server name, authentication information, and the database name. If you have been following along with this tutorial and have added the `SecurityTutorials.mdf` database to `App_Data`, attached it to `localhost\InstanceName`, and renamed it to `SecurityTutorialsDatabase`, then use the following values:

- Server: `localhost\InstanceName`
- Windows authentication
- Database: `SecurityTutorialsDatabase`

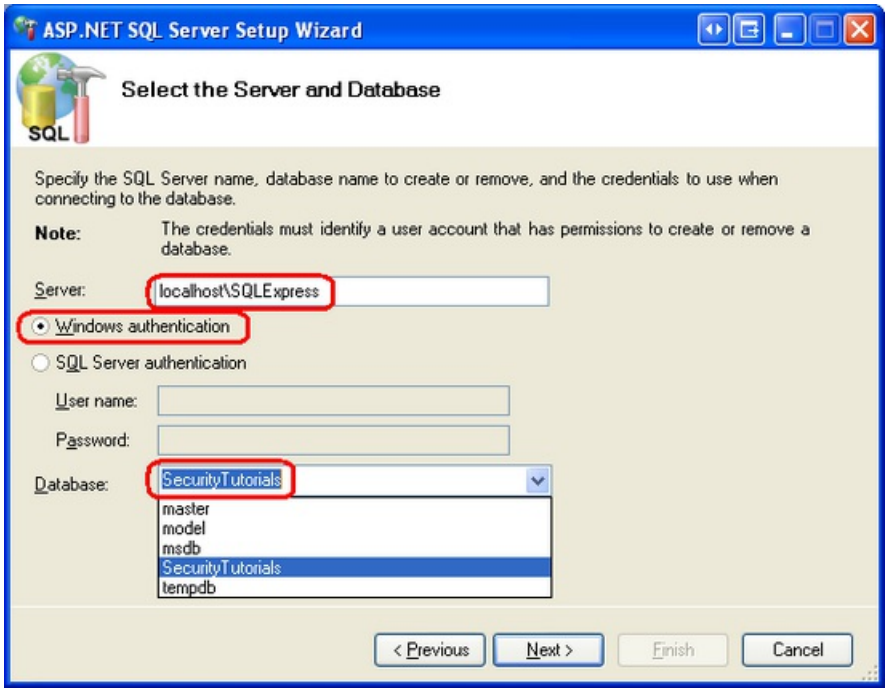


Figure 9: Enter the Database Information (Click to view full-size image)

After entering the database information, click Next. The final step summarizes the steps that will be taken. Click Next to install the application services and then Finish to complete the wizard.

> **NOTE**
> If you used Management Studio to attach the database and rename the database file, be sure to detach the database and close Management Studio before reopening Visual Studio. To detach the `SecurityTutorialsDatabase` database, right-click on the database name and, from the Tasks menu, choose Detach.

Upon completion of the wizard, return to Visual Studio and navigate to the Database Explorer. Expand the Tables folder. You should see a series of tables whose names start with the prefix `aspnet_`. Likewise, a variety of views and stored procedures can be found under the Views and Stored Procedures folders. These database objects make up the application services schema. We will examine the membership- and role-specific database objects in Step 3.

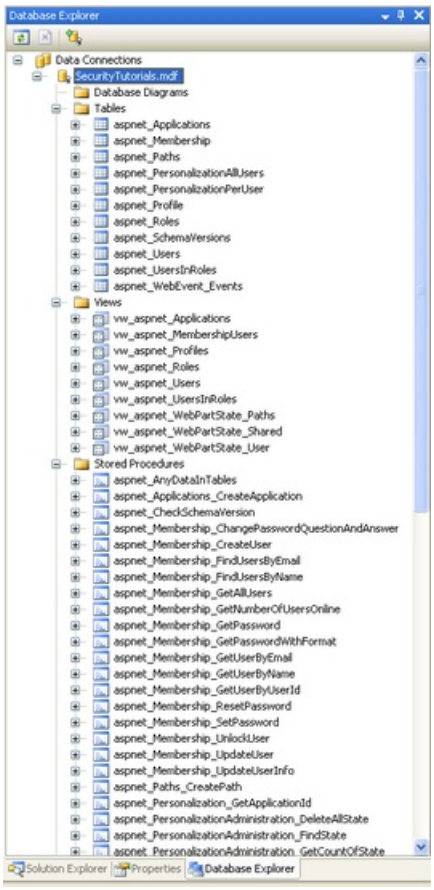


Figure 10: A Variety of Tables, Views, and Stored Procedures Have Been Added to the Database (Click to view full-size image)

> **NOTE**
> The `aspnet_regsql.exe` tool's graphical user interface installs the entire application services schema. But when executing `aspnet_regsql.exe` from the command-line you can specify what particular application services components to install (or remove). Therefore, if you want to add just the tables, views, and stored procedures necessary for the `SqlMembershipProvider` and `SqlRoleProvider` providers, run `aspnet_regsql.exe` from the command-line. Alternatively, you can manually run the appropriate subset of T-SQL create scripts used by `aspnet_regsql.exe`. These scripts are located in the `WINDIR%\Microsoft.Net\Framework\v2.0.50727\` folder with names like `InstallCommon.sql`, `InstallMembership.sql`, `InstallRoles.sql`, `InstallProfile.sql`, `InstallSqlState.sql`, and so on.

At this point we have created the database objects needed by the `SqlMembershipProvider`. However, we still need to instruct the Membership framework that it should use the `SqlMembershipProvider` (versus, say, the `ActiveDirectoryMembershipProvider`) and that the `SqlMembershipProvider` should use the `SecurityTutorials` database. We'll look at how to specify what provider to use and how to customize the selected provider's

settings in Step 4. But first, let's take a deeper look at the database objects that were just created.

Step 3: A Look at the Schema's Core Tables

When working with the Membership and Roles frameworks in an ASP.NET application, the implementation details are encapsulated by the provider. In future tutorials we will interface with these frameworks via the .NET Framework's `Membership` and `Roles` classes. When using these high-level APIs we do not need to concern ourselves with the low-level details, like what queries are executed or what tables are modified by the `SqlMembershipProvider` and `SqlRoleProvider`.

Given this, we could confidently use the Membership and Roles frameworks without having explored the database schema created in Step 2. However, when creating the tables to store application data we may need to create entities that relate to users or roles. It helps to have a familiarity with the `SqlMembershipProvider` and `SqlRoleProvider` schemas when establishing foreign key constraints between the application data tables and those tables created in Step 2. Moreover, in certain rare circumstances we may need to interface with the user and role stores directly at the database level (instead of through the `Membership` or `Roles` classes).

Partitioning the User Store Into Applications

The Membership and Roles frameworks are designed such that a single user and role store can be shared among many different applications. An ASP.NET application that uses the Membership or Roles frameworks must specify what application partition to use. In short, multiple web applications can use the same user and role stores. Figure 11 depicts user and role stores that are partitioned into three applications: HRSite, CustomerSite, and SalesSite. These three web applications each have their own unique users and roles, yet they all physically store their user account and role information in the same database tables.

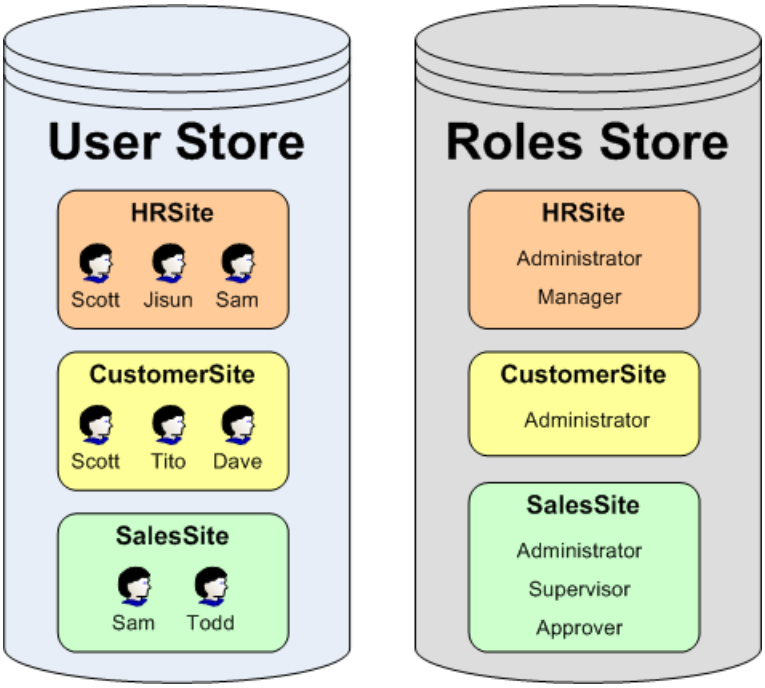


Figure 11: User Accounts May Be Partitioned Across Multiple Applications (Click to view full-size image)

The `aspnet_Applications` table is what defines these partitions. Each application that uses the database to store user account information is represented by a row in this table. The `aspnet_Applications` table has four columns: `ApplicationId`, `ApplicationName`, `LoweredApplicationName`, and `Description`. `ApplicationId` is of type `uniqueidentifier` and is the table's primary key; `ApplicationName` provides a unique human-friendly name for each application.

The other Membership- and Role-related tables link back to the `ApplicationId` field in `aspnet_Applications`. For example, the `aspnet_Users` table, which contains a record for each user account, has an `ApplicationId` foreign key field; ditto for the `aspnet_Roles` table. The `ApplicationId` field in these tables specifies the

application partition the user account or role belongs to.

Storing User Account Information

User account information is housed in two tables: `aspnet_Users` and `aspnet_Membership`. The `aspnet_Users` table contains fields that hold the essential user account information. The three most pertinent columns are:

- `UserId`
- `UserName`
- `ApplicationId`

`UserId` is the primary key (and of type `uniqueidentifier`). `UserName` is of type `nvarchar(256)` and, along with the password, makes up the user's credentials. (A user's password is stored in the `aspnet_Membership` table.) `ApplicationId` links the user account to a particular application in `aspnet_Applications`. There is a composite `UNIQUE` constraint on the `UserName` and `ApplicationId` columns. This ensures that in a given application each UserName is unique, yet it allows for the same `UserName` to be used in different applications.

The `aspnet_Membership` table includes additional user account information, like the user's password, email address, the last login date and time, and so forth. There is a one-to-one correspondence between records in the `aspnet_Users` and `aspnet_Membership` tables. This relationship is ensured by the `UserId` field in `aspnet_Membership`, which serves as the table's primary key. Like the `aspnet_Users` table, `aspnet_Membership` includes an `ApplicationId` field that ties this information to a particular application partition.

Securing Passwords

Password information is stored in the `aspnet_Membership` table. The `SqlMembershipProvider` allows for passwords to be stored in the database using one of the following three techniques:

- **Clear** - the password is stored in the database as plain-text. I strongly discourage using this option. If the database is compromised - be it by a hacker who finds a back door or a disgruntled employee who has database access - every single user's credentials are there for the taking.
- **Hashed** - passwords are hashed using a one-way hash algorithm and a randomly generated salt value. This hashed value (along with the salt) is stored in the database.
- **Encrypted** - an encrypted version of the password is stored in the database.

The password storage technique used depends on the `SqlMembershipProvider` settings specified in `Web.config`. We will look at customizing the `SqlMembershipProvider` settings in Step 4. The default behavior is to store the hash of the password.

The columns responsible for storing the password are `Password`, `PasswordFormat`, and `PasswordSalt`. `PasswordFormat` is a field of type `int` whose value indicates the technique used for storing the password: 0 for Clear; 1 for Hashed; 2 for Encrypted. `PasswordSalt` is assigned a randomly generated string regardless of the password storage technique used; the value of `PasswordSalt` is only used when computing the hash of the password. Finally, the `Password` column contains the actual password data, be it the plain-text password, the hash of the password, or the encrypted password.

Table 1 illustrates what these three columns might look like for the various storage techniques when storing the password MySecret! .

STORAGE TECHNIQUE<_O3A_P />	PASSWORD<_O3A_P />	PASSWORDFORMAT<_O3A_P />	PASSWORDSALT<_O3A_P />
Clear	MySecret!	0	tTnkPlesqissc2y2SMEygA==
Hashed	2oXm6sZHWbTHFgjgkGQsc2Ec9ZM=	1	wFgjUfhdUFOCKQiI61vtiQ==

STORAGE TECHNIQUE<_O3A_P />	PASSWORD<_O3A_P />	PASSWORDFORMAT<_O3A_P />	PASSWORDSALT<_O3A_P />
Encrypted	62RZgDvhxykkqsMchZ0Yly7HS6onhpaoCYaRxV8g0F4CW56OXUU3e7Inza9j9BKp	2	LSRzhGS/aa/oqAXGLHJNBw==

Table 1: Example Values for the Password-Related Fields When Storing the Password MySecret!

> **NOTE**
>
> The particular encryption or hashing algorithm used by the `SqlMembershipProvider` is determined by the settings in the `<machineKey>` element. We discussed this configuration element in Step 3 of the *Forms Authentication Configuration and Advanced Topics* tutorial.

Storing Roles and Role Associations

The Roles framework allows developers to define a set of roles and specify what users belong to what roles. This information is captured in the database through two tables: `aspnet_Roles` and `aspnet_UsersInRoles`. Each record in the `aspnet_Roles` table represents a role for a particular application. Much like the `aspnet_Users` table, the `aspnet_Roles` table has three columns pertinent to our discussion:

- `RoleId`
- `RoleName`
- `ApplicationId`

`RoleId` is the primary key (and of type `uniqueidentifier`). `RoleName` is of type `nvarchar(256)`. And `ApplicationId` links the user account to a particular application in `aspnet_Applications`. There is a composite `UNIQUE` constraint on the `RoleName` and `ApplicationId` columns, ensuring that in a given application each role name is unique.

The `aspnet_UsersInRoles` table serves as a mapping between users and roles. There are only two columns - `UserId` and `RoleId` - and together they make up a composite primary key.

Step 4: Specifying the Provider and Customizing Its Settings

All of the frameworks that support the provider model - such as the Membership and Roles frameworks - lack implementation details themselves and instead delegate that responsibility to a provider class. In the case of the Membership framework, the `Membership` class defines the API for managing user accounts, but it does not interact directly with any user store. Rather, the `Membership` class's methods hand off the request to the configured provider - we will be using the `SqlMembershipProvider`. When we invoke one of the methods in the `Membership` class, how does the Membership framework know to delegate the call to the `SqlMembershipProvider`?

The `Membership` class has a `Providers` property that contains a reference to all of the registered provider classes available for use by the Membership framework. Each registered provider has an associated name and type. The name offers a human-friendly way to reference a particular provider in the `Providers` collection, while the type identifies the provider class. Moreover, each registered provider may include configuration settings. Configuration settings for the Membership framework include `PasswordFormat` and `requiresUniqueEmail`, among many others. See Table 2 for a complete list of configuration settings used by the `SqlMembershipProvider`.

The `Providers` property's contents are specified through the web application's configuration settings. By default, all web applications have a provider named `AspNetSqlMembershipProvider` of type `SqlMembershipProvider`. This default Membership provider is registered in `machine.config` (located at `%WINDIR%\Microsoft.Net\Framework\v2.0.50727\CONFIG`):

```
<membership>
 <providers>
 <add name="AspNetSqlMembershipProvider"
 type="System.Web.Security.SqlMembershipProvider, System.Web, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a"
 connectionStringName="LocalSqlServer"
 enablePasswordRetrieval="false"
 enablePasswordReset="true"
 requiresQuestionAndAnswer="true"
 applicationName="/"
 requiresUniqueEmail="false"
 passwordFormat="Hashed"
 maxInvalidPasswordAttempts="5"
 minRequiredPasswordLength="7"
 minRequiredNonalphanumericCharacters="1"
 passwordAttemptWindow="10"
 passwordStrengthRegularExpression=""/>
 </providers>
</membership>
```

As the markup above shows, the `<membership>` element defines the configuration settings for the Membership framework while the `<providers>` child element specifies the registered providers. Providers may be added or removed using the `<add>` or `<remove>` elements; use the `<clear>` element to remove all currently registered providers. As the markup above shows, `machine.config` adds a provider named `AspNetSqlMembershipProvider` of type `SqlMembershipProvider`.

In addition to the `name` and `type` attributes, the `<add>` element contains attributes that define the values for various configuring settings. Table 2 lists the available `SqlMembershipProvider`-specific configuration settings, along with a description of each.

NOTE

Any default values noted in Table 2 refer to the default values defined in the `SqlMembershipProvider` class. Note that not all of the configuration settings in `AspNetSqlMembershipProvider` correspond to the default values of the `SqlMembershipProvider` class. For example, if not specified in a Membership provider, the `requiresUniqueEmail` setting defaults to true. However, the `AspNetSqlMembershipProvider` overrides this default value by explicitly specifying a value of `false`.

SETTING<_O3A_P />	DESCRIPTION<_O3A_P />
`ApplicationName`	Recall that the Membership framework allows for a single user store to be partitioned across multiple applications. This setting indicates the name of the application partition used by the Membership provider. If this value is not explicitly specified, it is set, at runtime, to the value of the application's virtual root path.
`commandTimeout`	Specifies the SQL command timeout value (in seconds). The default value is 30.

SETTING<_O3A_P />	DESCRIPTION<_O3A_P />
`connectionStringName`	The name of the connection string in the `<connectionStrings>` element to use to connect to the user store database. This value is required.
`description`	Provides a human-friendly description of the registered provider.
`enablePasswordRetrieval`	Specifies whether users may retrieve their forgotten password. The default value is `false`.
`enablePasswordReset`	Indicates whether users are allowed to reset their password. Defaults to `true`.
`maxInvalidPasswordAttempts`	The maximum number of unsuccessful login attempts that may occur for a given user during the specified `passwordAttemptWindow` before the user is locked out. The default value is 5.
`minRequiredNonalphanumericCharacters`	The minimum number of non-alphanumeric characters that must appear in a user's password. This value must be between 0 and 128; the default is 1.
`minRequiredPasswordLength`	The minimum number of characters required in a password. This value must be between 0 and 128; the default is 7.
`name`	The name of the registered provider. This value is required.
`passwordAttemptWindow`	The number of minutes during which failed login attempts are tracked. If a user supplies invalid login credentials `maxInvalidPasswordAttempts` times within this specified window, they are locked out. The default value is 10.
`PasswordFormat`	The password storage format: `Clear`, `Hashed`, or `Encrypted`. The default is `Hashed`.
`passwordStrengthRegularExpression`	If provided, this regular expression is used to evaluate the strength of the user's selected password when creating a new account or when changing their password. The default value is an empty string.
`requiresQuestionAndAnswer`	Specifies whether a user must answer his security question when retrieving or resetting his password. The default value is `true`.
`requiresUniqueEmail`	Indicates whether all user accounts in a given application partition must have a unique email address. The default value is `true`.
`type`	Specifies the type of the provider. This value is required.

Table 2: Membership and `SqlMembershipProvider` Configuration Settings

In addition to `AspNetSqlMembershipProvider`, other Membership providers may be registered on an application-by-application basis by adding similar markup to the `Web.config` file.

> **NOTE**
> The Roles framework works in much the same way: there is a default registered role provider in `machine.config` and the registered providers may be customized on an application-by-application basis in `Web.config`. We will examine the Roles framework and its configuration markup in detail in a future tutorial.

Customizing the `SqlMembershipProvider` Settings

The default `SqlMembershipProvider` (`AspNetSqlMembershipProvider`) has its `connectionStringName` attribute set to `LocalSqlServer`. Like the `AspNetSqlMembershipProvider` provider, the connection string name `LocalSqlServer` is defined in `machine.config`.

```
<connectionStrings>
 <add name="LocalSqlServer"
 connectionString="data source=.\SQLEXPRESS;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User Instance=true"
 providerName="System.Data.SqlClient"/>
</connectionStrings>
```

As you can see, this connection string defines a SQL 2005 Express Edition database located at |DataDirectory|aspnetdb.mdf. The string |DataDirectory| is translated at runtime to point to the `~/App_Data/` directory, so the database path |DataDirectory|aspnetdb.mdf translates to `~/App_Data` / `aspnet.mdf`.

If we did not specify any Membership provider information in our application's `Web.config` file, the application uses the default registered Membership provider, `AspNetSqlMembershipProvider`. If the `~/App_Data/aspnet.mdf` database does not exist, the ASP.NET runtime will automatically create it and add the application services schema. However, we don't want to use the `aspnet.mdf` database; rather, we want to use the `SecurityTutorials.mdf` database we created in Step 2. This modification can be accomplished in one of two ways:

- **Specify a value for the `LocalSqlServer` connection string name in `Web.config`.** By overwriting the `LocalSqlServer` connection string name value in `Web.config`, we can use the default registered Membership provider (`AspNetSqlMembershipProvider`) and have it correctly work with the `SecurityTutorials.mdf` database. This approach is fine if you are content with the configuration settings specified by `AspNetSqlMembershipProvider`. For more information on this technique, see Scott Guthrie's blog post, Configuring ASP.NET 2.0 Application Services to Use SQL Server 2000 or SQL Server 2005.
- **Add a new registered provider of type `SqlMembershipProvider` and configure its `connectionStringName` setting to point to the `SecurityTutorials.mdf` database.** This approach is useful in scenarios where you want to customize other configuration properties in addition to the database connection string. In my own projects I always use this approach because of its flexibility and readability.

Before we can add a new registered provider that references the `SecurityTutorials.mdf` database, we first need to add an appropriate connection string value in the `<connectionStrings>` section in `Web.config`. The following markup adds a new connection string named `SecurityTutorialsConnectionString` that references the SQL Server 2005 Express Edition `SecurityTutorials.mdf` database in the `App_Data` folder.

```
<configuration>
 <connectionStrings>
 <add name="SecurityTutorialsConnectionString"
 connectionString="Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\SecurityTutorials.mdf;Integrated Security=True;User
Instance=True"
 providerName="System.Data.SqlClient"/>
 </connectionStrings>
 <system.web>
 ... Configuration markup removed for brevity ...  </system.web>
</configuration>
```

> **NOTE**
> If you are using an alternate database file, update the connection string as needed. For more information on forming the correct connection string, refer to ConnectionStrings.com.

Next, add the following Membership configuration markup to the `Web.config` file. This markup registers a new provider named `SecurityTutorialsSqlMembershipProvider`.

```
<configuration>
 <connectionStrings>
 <add name="SecurityTutorialsConnectionString"
 connectionString="Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\SecurityTutorials.mdf;Integrated Security=True;User
Instance=True"
 providerName="System.Data.SqlClient"/>
 </connectionStrings>
 <system.web>
 <membership defaultProvider="SecurityTutorialsSqlMembershipProvider">
 <providers>
 <!-- Add a customized SqlMembershipProvider -->
 <add name="SecurityTutorialsSqlMembershipProvider"
 type="System.Web.Security.SqlMembershipProvider"
 connectionStringName="SecurityTutorialsConnectionString"
 enablePasswordRetrieval="false"
 enablePasswordReset="true"
 requiresQuestionAndAnswer="true"
 applicationName="SecurityTutorials"
 requiresUniqueEmail="true"
 passwordFormat="Hashed"
 maxInvalidPasswordAttempts="5"
 minRequiredPasswordLength="7"
 minRequiredNonalphanumericCharacters="1"
 passwordAttemptWindow="10"
 passwordStrengthRegularExpression=""/>
 </providers>
 </membership>
 ... Configuration markup removed for brevity ...
 </system.web>
</configuration>
```

In addition to registering the `SecurityTutorialsSqlMembershipProvider` provider, the above markup defines the `SecurityTutorialsSqlMembershipProvider` as the default provider (via the `defaultProvider` attribute in the `<membership>` element). Recall that the Membership framework can have multiple registered providers. Since `AspNetSqlMembershipProvider` is registered as the first provider in `machine.config`, it serves as the default provider unless we indicate otherwise.

Currently, our application has two registered providers: `AspNetSqlMembershipProvider` and `SecurityTutorialsSqlMembershipProvider`. However, before registering the

`SecurityTutorialsSqlMembershipProvider` provider we could have cleared out all previously registered providers by adding a `<clear />` element immediately before our `<add>` element. This would clear out the `AspNetSqlMembershipProvider` from the list of registered providers, meaning that the `SecurityTutorialsSqlMembershipProvider` would be the only registered Membership provider. If we used this approach, then we would not need to mark the `SecurityTutorialsSqlMembershipProvider` as the default provider, since it would be the only registered Membership provider. For more information on using `<clear />`, see Using `<clear />` When Adding providers.

Note that the `SecurityTutorialsSqlMembershipProvider`'s `connectionStringName` setting references the just-added `SecurityTutorialsConnectionString` connection string name, and that its `applicationName` setting has been set to a value of SecurityTutorials. Additionally, the `requiresUniqueEmail` setting has been set to `true`. All other configuration options are identical to the values in `AspNetSqlMembershipProvider`. Feel free to make any configuration modifications here, if you wish. For example, you could tighten the password strength by requiring two non-alphanumeric characters instead of one, or by increasing the password length to eight characters instead of seven.

> **NOTE**
>
> Recall that the Membership framework allows for a single user store to be partitioned across multiple applications. The Membership provider's `applicationName` setting indicates what application the provider uses when working with the user store. It is important that you explicitly set a value for the `applicationName` configuration setting because if the `applicationName` is not explicitly set, it is assigned to the web application's virtual root path at runtime. This works fine as long as the application's virtual root path doesn't change, but if you move the application to a different path, the `applicationName` setting will change too. When this happens, the Membership provider will start working with a different application partition than was previously used. User accounts created prior to the move will reside in a different application partition and those users will no longer be able to log into the site. For a more in-depth discussion on this matter, see Always Set the `applicationName` Property When Configuring ASP.NET 2.0 Membership and Other providers.

Summary

At this point we have a database with the configured application services (`SecurityTutorials.mdf`) and have configured our web application so that the Membership framework uses the `SecurityTutorialsSqlMembershipProvider` provider we just registered. This registered provider is of type `SqlMembershipProvider` and has its `connectionStringName` set to the appropriate connection string (`SecurityTutorialsConnectionString`) and its `applicationName` value explicitly set.

We are now ready to use the Membership framework from our application. In the next tutorial we will examine how to create new user accounts. Following that we will explore authenticating users, performing user-based authorization, and storing additional user-related information in the database.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Always Set the `applicationName` Property When Configuring ASP.NET 2.0 Membership and Other providers
- Configuring ASP.NET 2.0 Application Services to Use SQL Server 2000 or SQL Server 2005
- Download SQL Server Management Studio Express Edition
- Examining ASP.NET 2.0s Membership, Roles, and Profile
- The `<add>` Element for providers for Membership
- The `<membership>` Element
- The `<providers>` Element for Membership

- Using `<clear />` When Adding providers
- Working Directly with the `SqlMembershipProvider`

Video Training on Topics Contained in this Tutorial

- Understanding ASP.NET Memberships
- Configuring SQL To Work with Membership Schemas
- Changing Membership Settings in the Default Membership Schema

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Alicja Maziarz. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Creating User Accounts (VB)

2/20/2020 • 31 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

In this tutorial we will explore using the Membership framework (via the SqlMembershipProvider) to create new user accounts. We will see how to create new users programmatically and through ASP.NET's built-in CreateUserWizard control.

Introduction

In the preceding tutorial we installed the application services schema in a database, which added the tables, views, and stored procedures needed by the `SqlMembershipProvider` and `SqlRoleProvider`. This created the infrastructure we will need for the remainder of the tutorials in this series. In this tutorial we will explore using the Membership framework (via the `SqlMembershipProvider`) to create new user accounts. We will see how to create new users programmatically and through ASP.NET's built-in CreateUserWizard control.

In addition to learning how to create new user accounts, we will also need to update the demo website we first created in the *An Overview of Forms Authentication* tutorial and then enhanced in the *Forms Authentication Configuration and Advanced Topics* tutorial. Our demo web application has a login page that validates users' credentials against hard-coded username/password pairs. Moreover, `Global.asax` includes code that creates custom `IPrincipal` and `IIdentity` objects for authenticated users. We will update the login page to validate users' credentials against the Membership framework and remove the custom principal and identity logic.

Let's get started!

The Forms Authentication and Membership Checklist

Before we begin working with the Membership framework, let's take a moment to review the important steps we have taken to reach this point. When using the Membership framework with the `SqlMembershipProvider` in a forms-based authentication scenario, the following steps need to be performed prior to implementing Membership functionality in your web application:

1. **Enable forms-based authentication.** As we discussed in *An Overview of Forms Authentication*, forms authentication is enabled by editing `Web.config` and setting the `<authentication>` element's `mode` attribute to `Forms`. With forms authentication enabled, each incoming request is examined for a *forms authentication ticket*, which, if present, identifies the requestor.
2. **Add the application services schema to the appropriate database.** When using the `SqlMembershipProvider` we need to install the application services schema to a database. Usually this schema is added to the same database that holds the application's data model. The *Creating the Membership Schema in SQL Server* tutorial looked at using the `aspnet_regsql.exe` tool to accomplish this.
3. **Customize the Web Application's Settings to reference the database from step 2.** The *Creating the Membership Schema in SQL Server* tutorial showed two ways to configure the web application so that the `SqlMembershipProvider` would use the database selected in step 2: by modifying the `LocalSqlServer` connection string name; or by adding a new registered provider to the list of Membership framework providers and customizing that new provider to use the database from step 2.

When building a web application that uses the `SqlMembershipProvider` and forms-based authentication, you will

need to perform these three steps before using the `Membership` class or the ASP.NET Login Web controls. Since we already performed these steps in previous tutorials, we are ready to start using the Membership framework!

Step 1: Adding New ASP.NET Pages

In this tutorial and the next three we will be examining various Membership-related functions and capabilities. We will need a series of ASP.NET pages to implement the topics examined throughout these tutorials. Let's create those pages and then a site map file `(Web.sitemap)`.

Start by creating a new folder in the project named `Membership`. Next, add five new ASP.NET pages to the `Membership` folder, linking each page with the `Site.master` master page. Name the pages:

- `CreatingUserAccounts.aspx`
- `UserBasedAuthorization.aspx`
- `EnhancedCreateUserWizard.aspx`
- `AdditionalUserInfo.aspx`
- `Guestbook.aspx`

At this point your project's Solution Explorer should look similar to the screen shot shown in Figure 1.

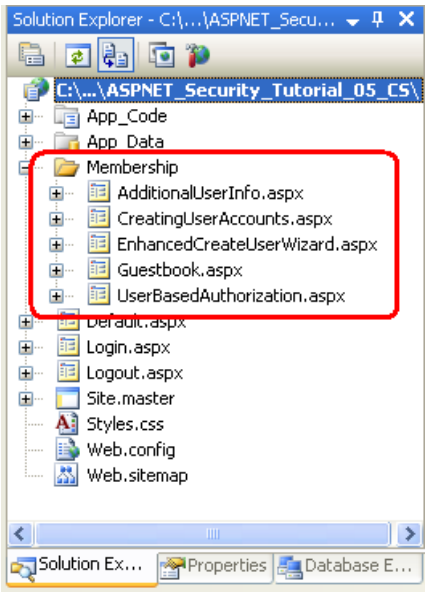


Figure 1: Five New Pages Have Been Added to the `Membership` Folder (Click to view full-size image)

Each page should, at this point, have the two Content controls, one for each of the master page's ContentPlaceHolders: `MainContent` and `LoginContent`.

```
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent"
Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="LoginContent"
Runat="Server">
</asp:Content>
```

Recall that the `LoginContent` ContentPlaceHolder's default markup displays a link to log on or log off the site, depending on whether the user is authenticated. The presence of the `Content2` Content control, however, overrides the master page's default markup. As we discussed in *An Overview of Forms Authentication* tutorial, this is useful in pages where we do not want to display login-related options in the left column.

For these five pages, however, we want to show the master page's default markup for the `LoginContent` ContentPlaceHolder. Therefore, remove the declarative markup for the `Content2` Content control. After doing so,

each of the five page's markup should contain just one Content control.

Step 2: Creating the Site Map

All but the most trivial websites need to implement some form of a navigational user interface. The navigation user interface may be a simple list of links to the various sections of the site. Alternatively, these links may be arranged into menus or tree views. As page developers, creating the navigational user interface is only half of the story. We also need some means to define the site's logical structure in a maintainable and updateable fashion. As new pages are added or existing pages removed, we want to be able to update a single source - the site map - and have those modifications reflected across the site's navigational user interface.

These two tasks - defining the site map and implementing a navigational user interface based on the site map - are easy to accomplish thanks to the Site Map framework and the Navigation Web controls added in ASP.NET version 2.0. The Site Map framework allows for a developer to define a site map and then to access it through a programmatic API (the `SiteMap` class). The built-in Navigation Web controls include a Menu control, the TreeView control, and the SiteMapPath control.

Like the Membership and Roles frameworks, the Site Map framework is built atop the provider model. The job of the Site Map provider class is to generate the in-memory structure used by the `SiteMap` class from a persistent data store, such as an XML file or a database table. The .NET Framework ships with a default Site Map provider that reads the site map data from an XML file (`XmlSiteMapProvider`), and this is the provider we will be using in this tutorial. For some alternate Site Map provider implementations, refer to the Further Readings section at the end of this tutorial.

The default Site Map provider expects a properly-formatted XML file named `Web.sitemap` to exist the root directory. Since we are using this default provider, we need to add such a file and define the site map's structure in the appropriate XML format. To add the file, right-click on the project name in Solution Explorer and choose Add New Item. From the dialog box, opt to add a file of type Site Map named `Web.sitemap`.

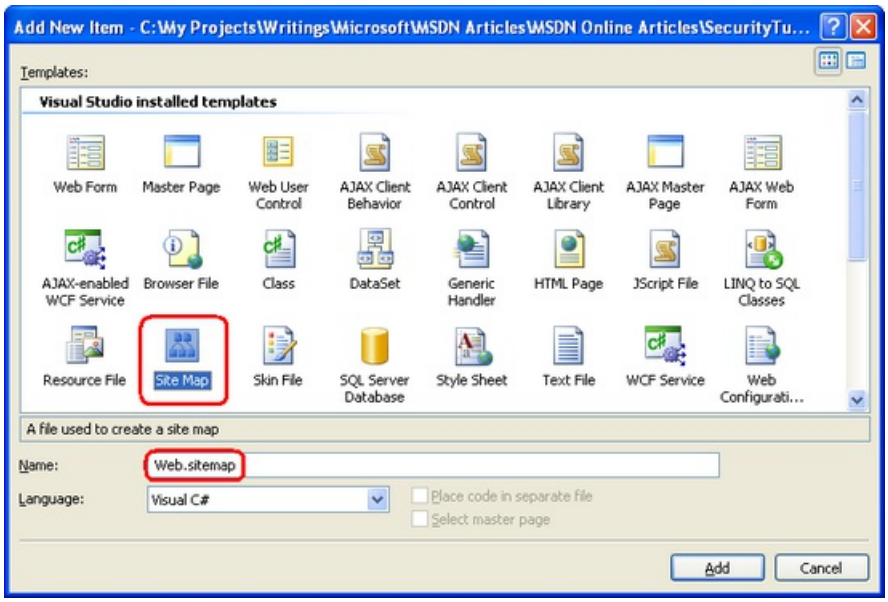


Figure 2: Add a File Named `Web.sitemap` to the Project's Root Directory (Click to view full-size image)

The XML site map file defines the website's structure as a hierarchy. This hierarchical relationship is modeled in the XML file via the ancestry of the `<siteMapNode>` elements. The `Web.sitemap` must start with a `<siteMap>` parent node that has precisely one `<siteMapNode>` child. This top-level `<siteMapNode>` element represents the root of the hierarchy, and may have an arbitrary number of descendent nodes. Each `<siteMapNode>` element must include a `title` attribute and may optionally include `url` and `description` attributes, among others; each non-empty `url` attribute must be unique.

Enter the following XML into the `Web.sitemap` file:

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0">
 <siteMapNode url="~/Default.aspx" title="Home">
 <siteMapNode title="Membership">
 <siteMapNode url="~/Membership/CreatingUserAccounts.aspx" title="Creating User Accounts" />
 <siteMapNode url="~/Membership/UserBasedAuthorization.aspx" title="User-Based Authorization" />
 <siteMapNode url="~/Membership/Guestbook.aspx" title="Storing Additional User Information" />
 </siteMapNode>
 </siteMapNode>
</siteMap>
```

The above site map markup defines the hierarchy shown in Figure 3.

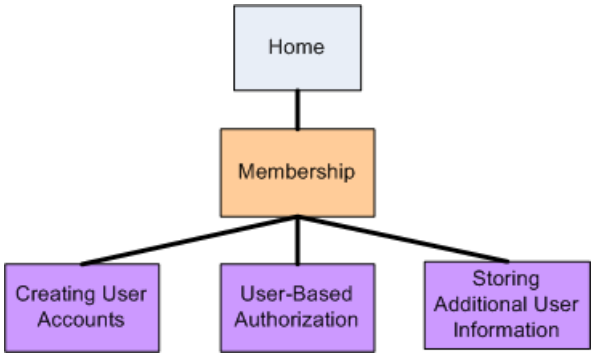


Figure 3: The Site Map Represents a Hierarchical Navigational Structure (Click to view full-size image)

Step 3: Updating the Master Page to Include a Navigational User Interface

ASP.NET includes a number of navigation-related Web controls for designing a user interface. These include the Menu, TreeView, and the SiteMapPath controls. The Menu and TreeView controls render the site map structure in a menu or a tree, respectively, whereas the SiteMapPath displays a breadcrumb that shows the current node being visited as well as its ancestors. The site map data can be bound to other data Web controls using the SiteMapDataSource and can be accessed programmatically via the `SiteMap` class.

Since a thorough discussion of the Site Map framework and the Navigation controls is beyond the scope of this tutorial series, rather than spend time crafting our own navigational user interface let's instead borrow the one used in my *Working with Data in ASP.NET 2.0* tutorial series, which uses a Repeater control to display a two-deep bulleted list of navigation links, as shown in Figure 4.

Adding a Two-Level List of Links in the Left Column

To create this interface, add the following declarative markup to the `Site.master` master page's left column where the text TODO: Menu will go here... currently resides.

```
<ul>
 <li>
 <asp:HyperLink runat="server" ID="lnkHome" NavigateUrl="~/Default.aspx">Home</asp:HyperLink>
 </li>
 <asp:Repeater runat="server" ID="menu" DataSourceID="SiteMapDataSource1">
 <ItemTemplate>
 <li>
 <asp:HyperLink ID="lnkMenuItem" runat="server"
 NavigateUrl='<%# Eval("Url") %>'><%# Eval("Title") %></asp:HyperLink>
 <asp:Repeater ID="submenu" runat="server" DataSource="<%#
 CType(Container.DataItem, SiteMapNode).ChildNodes %>">
 <HeaderTemplate>
 <ul>
 </HeaderTemplate>
 <ItemTemplate>
 <li>
 <asp:HyperLink ID="lnkMenuItem"
 runat="server" NavigateUrl='<%#  Eval("Url") %>'><%# Eval("Title") %></asp:HyperLink>
 </li>
 </ItemTemplate>
 <FooterTemplate>
 </ul>
 </FooterTemplate>
 </asp:Repeater>
 </li>
 </ItemTemplate>
 </asp:Repeater>
</ul>
<asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server" ShowStartingNode="false"/>
```

The above markup binds a Repeater control named `menu` to a SiteMapDataSource, which returns the site map hierarchy defined in `Web.sitemap`. Since the SiteMapDataSource control's `ShowStartingNode` property is set to False it starts returning the site map's hierarchy starting with the descendents of the Home node. The Repeater displays each of these nodes (currently just Membership) in an `<li>` element. Another, inner Repeater then displays the current node's children in a nested unordered list.

Figure 4 shows the above markup's rendered output with the site map structure we created in Step 2. The Repeater renders vanilla unordered list markup; the cascading style sheet rules defined in `Styles.css` are responsible for the aesthetically-pleasing layout. For a more detailed description of how the above markup works, refer to the Master Pages and Site Navigation tutorial.

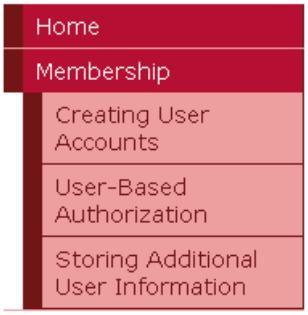


Figure 4: The Navigational User Interface is Rendered Using Nested Unordered Lists (Click to view full-size image)

Adding Breadcrumb Navigation

In addition to the list of links in the left column, let's also have each page display a breadcrumb. A breadcrumb is a navigational user interface element that quickly shows users their current position within the site hierarchy. The SiteMapPath control uses the Site Map framework to determine the current page's location in the site map and then displays a breadcrumb based on this information.

Specifically, add a `<span>` element to the master page's header `<div>` element, and set the new `<span>` element's `class` attribute to breadcrumb. (The `Styles.css` class contains a rule for a breadcrumb class.) Next, add a SiteMapPath to this new `<span>` element.

```
<div id="header">
 <span class="title">User Account Tutorials</span><br />
 <span class="breadcrumb">
 <asp:SiteMapPath ID="SiteMapPath1" runat="server">
 </asp:SiteMapPath>
 </span>
</div>
```

Figure 5 shows the output of the SiteMapPath when visiting `~/Membership/CreatingUserAccounts.aspx`.

Home > Membership > Creating User Accounts

Figure 5: The Breadcrumb Displays the Current Page and its Ancestors in the Site Map (Click to view full-size image)

Step 4: Removing the Custom Principal and Identity Logic

In the *Forms Authentication Configuration and Advanced Topics* tutorial we saw how to associate custom principal and identity objects to the authenticated user. We accomplished this by creating an event handler in `Global.asax` for the application's `PostAuthenticateRequest` event, which fires after the `FormsAuthenticationModule` has authenticated the user. In this event handler we replaced the `GenericPrincipal` and `FormsIdentity` objects added by the `FormsAuthenticationModule` with the `CustomPrincipal` and `CustomIdentity` objects we created in that tutorial.

While custom principal and identity objects are useful in certain scenarios, in most cases the `GenericPrincipal` and `FormsIdentity` objects are sufficient. Consequently, I think it would be worthwhile to return to the default behavior. Make this change by either removing or commenting out the `PostAuthenticateRequest` event handler or by deleting the `Global.asax` file entirely.

> **NOTE**
> Once you've commented or removed the code in `Global.asax`, you will need to comment out the code in `Default.aspx's` code-behind class that casts the `User.Identity` property to a `CustomIdentity` instance.

Step 5: Programmatically Creating a New User

To create a new user account through the Membership framework use the `Membership` class's `CreateUser` method. This method has input parameters for the username, password, and other user-related fields. On invocation, it delegates the creation of the new user account to the configured Membership provider and then returns a `MembershipUser` object representing the just-created user account.

The `CreateUser` method has four overloads, each accepting a different number of input parameters:

- `CreateUser(username, password)`
- `CreateUser(username, password, email)`
- `CreateUser(username, password, email, passwordQuestion, passwordAnswer, isApproved, MembershipCreateStatus)`
- `CreateUser(username, password, email, passwordQuestion, passwordAnswer, isApproved, providerUserKey, MembershipCreateStatus)`

These four overloads differ on the amount of information that is collected. The first overload, for example,

requires just the username and password for the new user account, whereas the second one also requires the user's email address.

These overloads exist because the information needed to create a new user account depends on the Membership provider's configuration settings. In the *Creating the Membership Schema in SQL Server* tutorial we examined specifying Membership provider configuration settings in `Web.config`. Table 2 included a complete list of the configuration settings.

One such Membership provider configuration setting that impacts what `CreateUser` overloads may be used is the `requiresQuestionAndAnswer` setting. If `requiresQuestionAndAnswer` is set to `true` (the default), then when creating a new user account we must specify a security question and answer. This information is later used if the user needs to reset or change their password. Specifically, at that time they are shown the security question and they must enter the correct answer in order to reset or change their password. Consequently, if the `requiresQuestionAndAnswer` is set to `true` then calling either of the first two `CreateUser` overloads results in an exception because the security question and answer are missing. Since our application is currently configured to require a security question and answer, we will need to use one of the latter two overloads when creating user's programmatically.

To illustrate using the `CreateUser` method, let's create a user interface where we prompt the user for their name, password, email, and an answer to a pre-defined security question. Open the `CreatingUserAccounts.aspx` page in the `Membership` folder and add the following Web controls to the Content control:

- A TextBox named `Username`
- A TextBox named `Password`, whose `TextMode` property is set to `Password`
- A TextBox named `Email`
- A Label named `SecurityQuestion` with its `Text` property cleared out
- A TextBox named `SecurityAnswer`
- A Button named `CreateAccountButton` whose `Text` property is set to Create the User Account
- A Label control named `CreateAccountResults` with its `Text` property cleared out

At this point your screen should look similar to the screen shot shown in Figure 6.

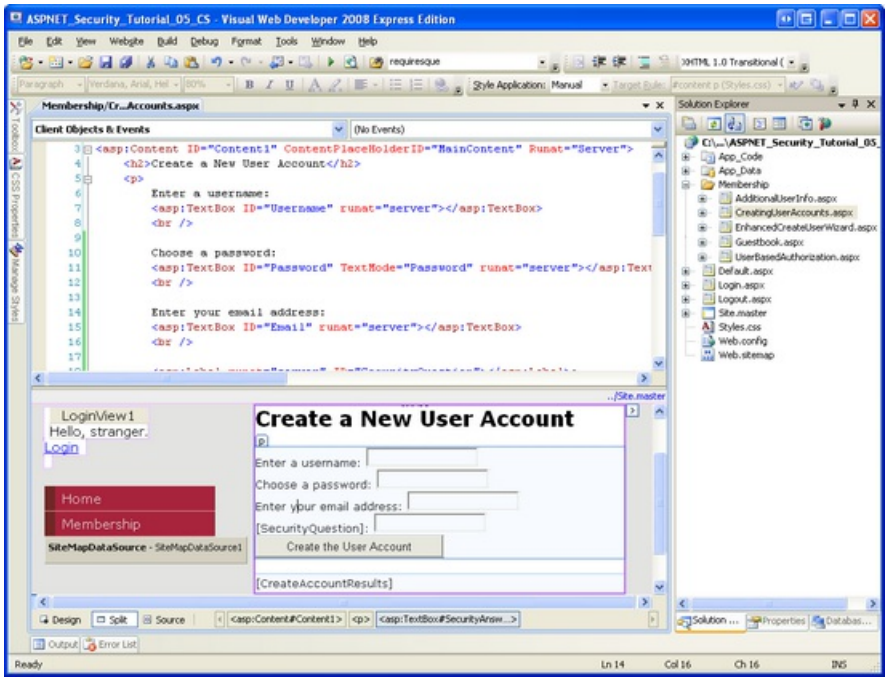


Figure 6: Add the Various Web Controls to the `CreatingUserAccounts.aspx Page` (Click to view full-size image)

The `SecurityQuestion` Label and `SecurityAnswer` TextBox are intended to display a pre-defined security question and collect the user's answer. Note that both the security question and answer are stored on a user-by-user

basis, so it is possible to allow each user to define their own security question. However, for this example I have decided to use a universal security question, namely: What is your favorite color?

To implement this pre-defined security question, add a constant to the page's code-behind class named `passwordQuestion`, assigning it the security question. Then, in the `Page_Load` event handler, assign this constant to the `SecurityQuestion` Label's `Text` property:

```
Const passwordQuestion As String = "What is your favorite color"
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
 If Not Page.IsPostBack Then
 SecurityQuestion.Text = passwordQuestion
 End If
End Sub
```

Next, create an event handler for the `CreateAccountButton'` s `Click` event and add the following code:

```
Protected Sub CreateAccountButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
CreateAccountButton.Click
 Dim createStatus As MembershipCreateStatus
 Dim newUser As MembershipUser = _
 Membership.CreateUser(Username.Text, Password.Text, _
 Email.Text, passwordQuestion, _
 SecurityAnswer.Text, True, _
 createStatus)
 Select Case createStatus
 Case MembershipCreateStatus.Success
 CreateAccountResults.Text = "The user account was successfully created!"
 Case MembershipCreateStatus.DuplicateUserName
 CreateAccountResults.Text = "There already exists a user with this username."
 Case MembershipCreateStatus.DuplicateEmail
 CreateAccountResults.Text = "There already exists a user with this email address."
 Case MembershipCreateStatus.InvalidEmail
 CreateAccountResults.Text = "There email address you provided in invalid."
 Case MembershipCreateStatus.InvalidAnswer
 CreateAccountResults.Text = "There security answer was invalid."
 Case MembershipCreateStatus.InvalidPassword
 CreateAccountResults.Text = "The password you provided is invalid. It must be seven characters long and
have at least one non-alphanumeric character."
 Case Else
 CreateAccountResults.Text = "There was an unknown error; the user account was NOT created."
 End Select
End Sub
```

The `Click` event handler starts by defining a variable named `createStatus` of type `MembershipCreateStatus`. `MembershipCreateStatus` is an enumeration that indicates the status of the `CreateUser` operation. For example, if the user account is created successfully, the resulting `MembershipCreateStatus` instance will be set to a value of `Success;` on the other hand, if the operation fails because there already exists a user with the same username, it will be set to a value of `DuplicateUserName`. In the `CreateUser` overload we use, we need to pass a `MembershipCreateStatus` instance into the method. This parameter is set to the appropriate value within the `CreateUser` method, and we can examine its value after the method call to determine whether the user account was successfully created.

After calling `CreateUser`, passing in `createStatus`, a `Select Case` statement is used to output an appropriate message depending on the value assigned to `createStatus`. Figures 7 shows the output when a new user has successfully been created. Figures 8 and 9 show the output when the user account is not created. In Figure 8, the visitor entered a five-letter password, which does not meet the password strength requirements spelled out in the Membership provider's configuration settings. In Figure 9, the visitor is attempting to create a user account with an existing username (the one created in Figure 7).

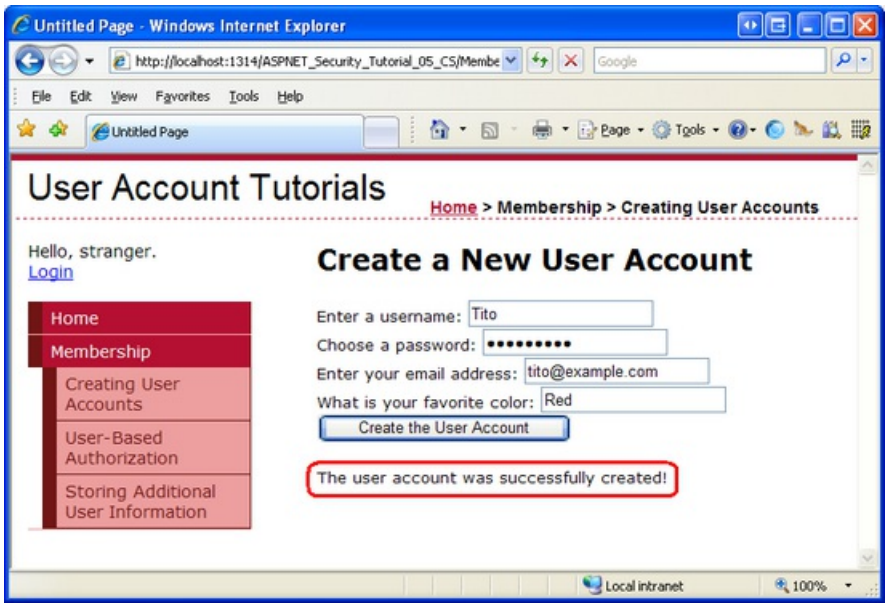


Figure 7: A New User Account is Successfully Created (Click to view full-size image)

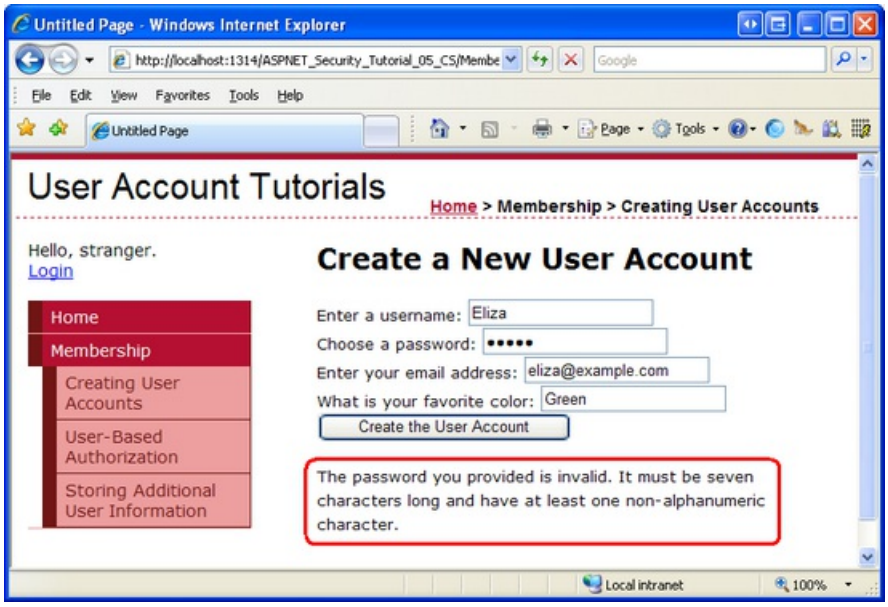


Figure 8: The User Account is Not Created Because the Supplied Password is Too Weak (Click to view full-size image)

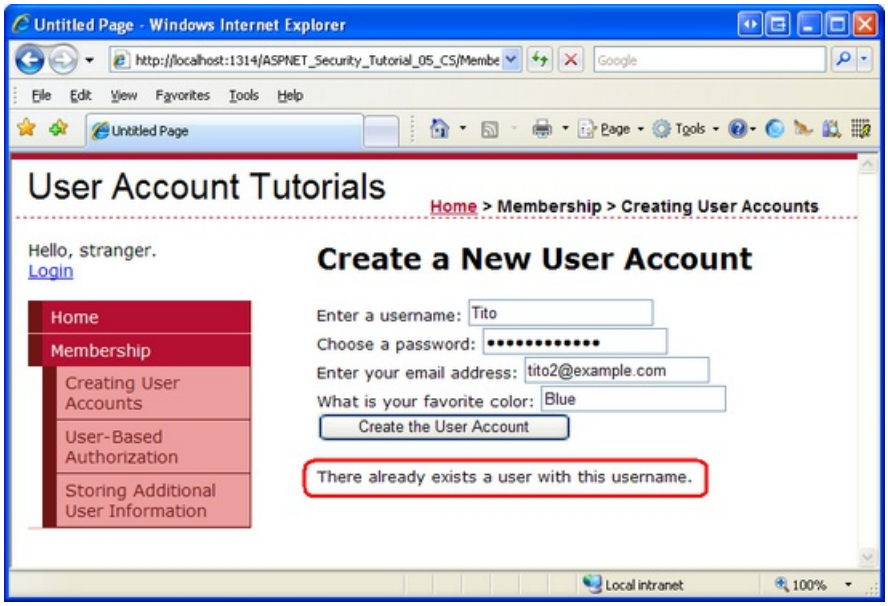


Figure 9: The User Account is Not Created Because the Username is Already in Use (Click to view full-size

image)

> **NOTE**
>
> You may be wondering how to determine success or failure when using one of the first two `CreateUser` method overloads, neither of which has a parameter of type `MembershipCreateStatus`. These first two overloads throw a `MembershipCreateUserException` exception in the face of a failure, which includes a `StatusCode` property of type `MembershipCreateStatus`.

After creating a few user accounts, verify that the accounts have been created by listing the contents of the `aspnet_Users` and `aspnet_Membership` tables in the `SecurityTutorials.mdf` database. As Figure 10 shows, I have added two users via the `CreatingUserAccounts.aspx` page: Tito and Bruce.

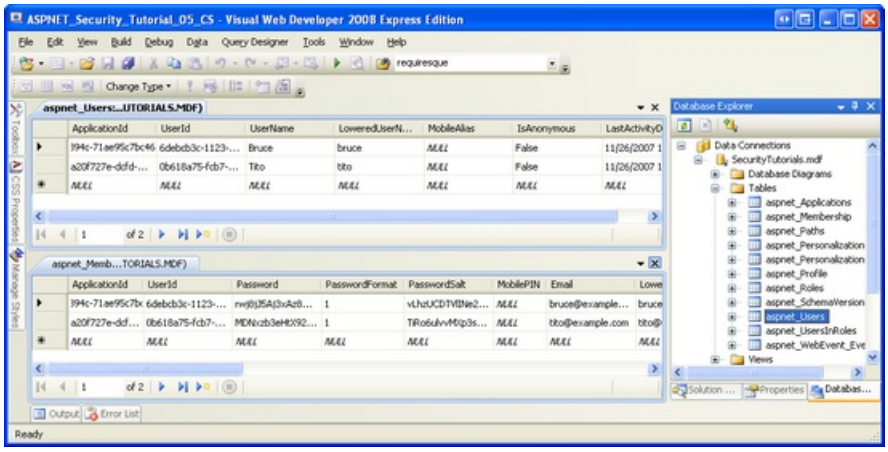


Figure 10: There are Two Users in the Membership User Store: Tito and Bruce (Click to view full-size image)

While the Membership user store now includes Bruce and Tito's account information, we have yet to implement functionality that allows for Bruce or Tito to log on to the site. Currently, `Login.aspx` validates the user's credentials against a hard-coded set of username/password pairs - it does *not* validate the supplied credentials against the Membership framework. For now seeing the new user accounts in the `aspnet_Users` and `aspnet_Membership` tables will have to suffice. In the next tutorial, *Validating User Credentials Against the Membership User Store*, we will update the login page to validate against the Membership store.

> **NOTE**
>
> If you do not see any users in your `SecurityTutorials.mdf` database, it may be because your web application is using the default Membership provider, `AspNetSqlMembershipProvider`, which uses the `ASPNETDB.mdf` database as its user store. To determine if this is the problem, click the Refresh button in the Solution Explorer. If a database named `ASPNETDB.mdf` has been added to the `App_Data` folder, this is the problem. Return to Step 4 of the *Creating the Membership Schema in SQL Server* tutorial for instructions on how to properly configure the Membership provider.

In most create user account scenarios, the visitor is presented with some interface to enter their username, password, email, and other essential information, at which point a new account is created. In this step we looked at building such an interface by hand and then saw how to use the `Membership.CreateUser` method to programmatically add the new user account based on the user's inputs. Our code, however, just created the new user account. It did not perform any follow up actions, like logging in the user to the site under the just-created user account, or sending a confirmation email to the user. These additional steps would require additional code in the Button's `Click` event handler.

ASP.NET ships with the CreateUserWizard control, which is designed to handle the user account creation process, from rendering a user interface for creating new user accounts, to creating the account in the Membership framework and performing post-account creation tasks, such as sending a confirmation email and logging the

just-created user into the site. Using the CreateUserWizard control is as simple as dragging the CreateUserWizard control from the Toolbox onto a page, and then setting a few properties. In most cases, you won't need to write a single line of code. We will explore this nifty control in detail in Step 6.

If new user accounts are only created through a typical Create Account web page, it is unlikely that you will ever need to write code that uses the `CreateUser` method, as the CreateUserWizard control will likely meet your needs. However, the `CreateUser` method is handy in scenarios where you need a highly customized Create Account user experience or when you need to programmatically create new user accounts through an alternative interface. For example, you might have a page that allows a user to upload an XML file that contains user information from some other application. The page might parses the contents of the uploaded XML file and create a new account for each user represented in the XML by calling the `CreateUser` method.

Step 6: Creating a New User with the CreateUserWizard Control

ASP.NET ships with a number of Login Web controls. These controls aid in many common user account- and login-related scenarios. The CreateUserWizard control is one such control that is designed to present a user interface for adding a new user account to the Membership framework.

Like many of the other Login-related Web controls, the CreateUserWizard can be used without writing a single line of code. It intuitively provides a user interface based on the Membership provider's configuration settings and internally calls the `Membership` class's `CreateUser` method after the user enters the necessary information and clicks the Create User button. The CreateUserWizard control is extremely customizable. There are a host of events that fire during various stages of the account creation process. We can create event handlers, as needed, to inject custom logic into the account creation workflow. Furthermore, the CreateUserWizard's appearance is very flexible. There are a number of properties that define the appearance of the default interface; if necessary, the control can be converted into a template or additional user registration steps may be added.

Let's start with a look at using the CreateUserWizard control's default interface and behavior. We'll then explore how to customize the appearance via the control's properties and events.

Examining the CreateUserWizard's Default Interface and Behavior

Return to the `CreatingUserAccounts.aspx` page in the `Membership` folder, switch to the Design or Split mode, and then add a CreateUserWizard control to the top of the page. The CreateUserWizard control is filed under the Toolbox's Login controls section. After adding the control, set its `ID` property to `RegisterUser`. As the screen shot in Figure 11 shows, the CreateUserWizard renders an interface with textboxes for the new user's username, password, email address, and security question and answer.

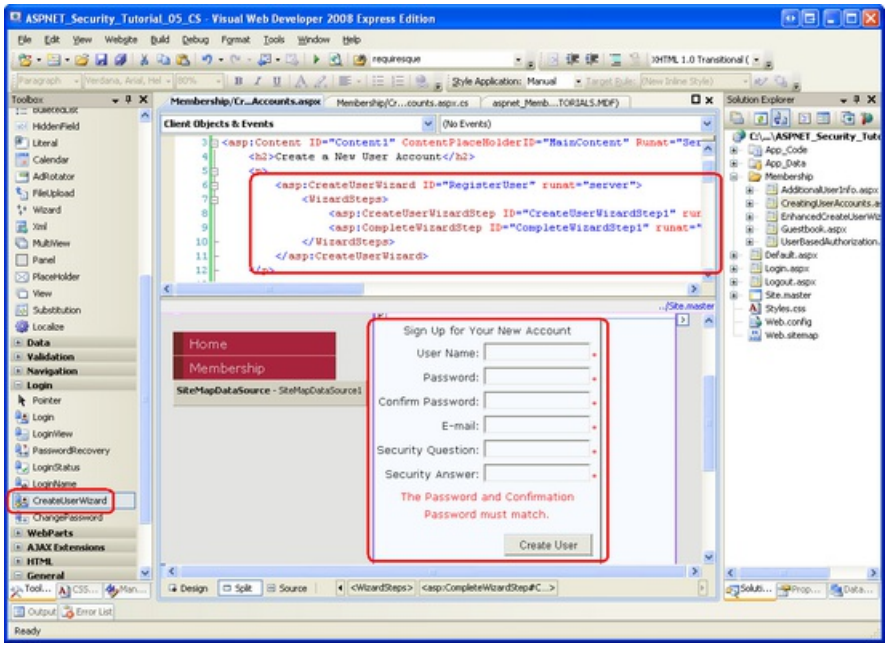


Figure 11: The CreateUserWizard Control Renders a Generic Create User Interface (Click to view full-size image)

Let's take a moment to compare the default user interface generated by the CreateUserWizard control with the interface we created in Step 5. For starters, the CreateUserWizard control allows the visitor to specify both the security question and answer, whereas our manually-created interface used a pre-defined security question. The CreateUserWizard control's interface also includes validation controls, whereas we had yet to implement validation on our interface's form fields. And the CreateUserWizard control interface includes a Confirm Password textbox (along with a CompareValidator to ensure that the text entered the Password and Compare Password textboxes are equal).

What's interesting is that the CreateUserWizard control consults the Membership provider's configuration settings when rendering its user interface. For example, the security question and answer textboxes are only displayed if `requiresQuestionAndAnswer` is set to True. Likewise, CreateUserWizard automatically adds a RegularExpressionValidator control to ensure that the password strength requirements are met, and sets its `ErrorMessage` and `ValidationExpression` properties based on the `minRequiredPasswordLength`, `minRequiredNonalphanumericCharacters`, and `passwordStrengthRegularExpression` configuration settings.

The CreateUserWizard control, as its name implies, is derived from the Wizard control. Wizard controls are designed to provide an interface for completing multi-step tasks. A Wizard control may have an arbitrary number of `WizardSteps`, each of which is a template that defines the HTML and Web controls for that step. The Wizard control initially displays the first `WizardStep`, along with navigation controls that permit the user to proceed from one step to the next, or to return to previous steps.

As the declarative markup in Figure 11 shows, the CreateUserWizard control's default interface includes two `WizardStep` s:

- `CreateUserWizardStep` ? renders the interface to collect information for creating the new user account. This is the step shown in Figure 11.
- `CompleteWizardStep` ? renders a message indicating that the account has successfully been created.

The CreateUserWizard's appearance and behavior can be modified by converting either of these steps to templates, or by adding your own `WizardStep` s. We will look at adding a `WizardStep` to the registration interface in the *Storing Additional User Information* tutorial.

Let's see the CreateUserWizard control in action. Visit the `CreatingUserAccounts.aspx` page through a browser. Start by entering some invalid values into the CreateUserWizard's interface. Try entering a password that doesn't conform to the password strength requirements, or leaving the User Name textbox empty. The CreateUserWizard will display an appropriate error message. Figure 12 shows the output when attempting to create a user with an insufficiently strong password.

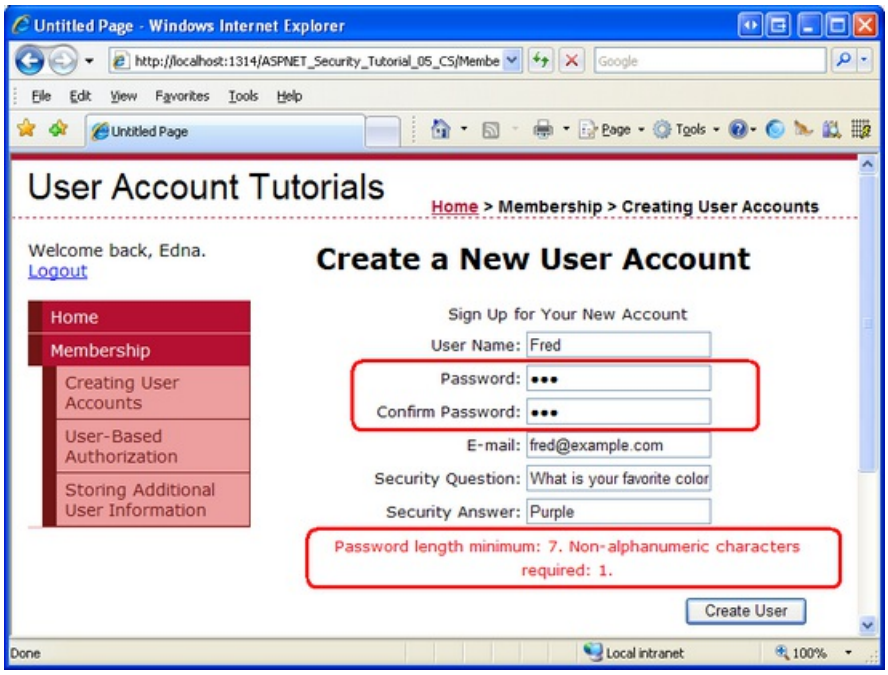


Figure 12: The CreateUserWizard Automatically Injects Validation Controls (Click to view full-size image)

Next, enter appropriate values into the CreateUserWizard and click the Create User button. Assuming the required fields have been entered and the password's strength is sufficient, the CreateUserWizard will create a new user account through the Membership framework and then display the `CompleteWizardStep`'s interface (see Figure 13). Behind the scenes, the CreateUserWizard calls the `Membership.CreateUser` method, just like we did in Step 5.

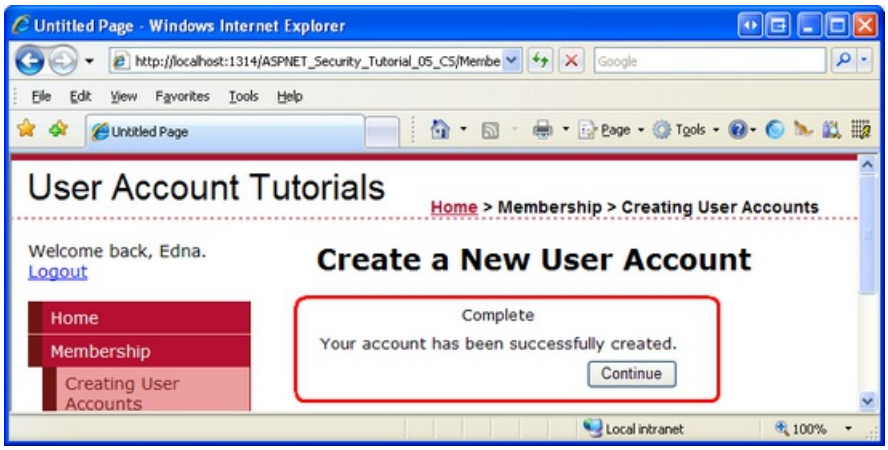


Figure 13: A New User Account has Successfully Been Created (Click to view full-size image)

> **NOTE**
>
> As Figure 13 shows, the `CompleteWizardStep`'s interface includes a Continue button. However, at this point clicking it just performs a postback, leaving the visitor on the same page. In the Customizing the CreateUserWizard's Appearance and Behavior Through Its Properties section we will look at how you can have this button send the visitor to `Default.aspx` (or some other page).

After creating a new user account, return to Visual Studio and examine the `aspnet_Users` and `aspnet_Membership` tables like we did in Figure 10 to verify that the account was successfully created.

Customizing the CreateUserWizard's Behavior and Appearance Through Its Properties

The CreateUserWizard can be customized in a variety of ways, through properties, `WizardStep` s, and event handlers. In this section we will look at how to customize the control's appearance through its properties; the next section looks at extending the control's behavior through event handlers.

Virtually all of the text displayed in the CreateUserWizard control's default user interface can be customized through its plethora of properties. For example, the User Name, Password, Confirm Password, E-mail, Security Question, and Security Answer labels displayed to the left of the textboxes can be customized by the `UserNameLabelText`, `PasswordLabelText`, `ConfirmPasswordLabelText`, `EmailLabelText`, `QuestionLabelText`, and `AnswerLabelText` properties, respectively. Likewise, there are properties for specifying the text for the Create User and Continue buttons in the `CreateUserWizardStep` and `CompleteWizardStep`, as well as if these buttons are rendered as Buttons, LinkButtons, or ImageButtons.

The colors, borders, fonts, and other visual elements are configurable through a host of style properties. The CreateUserWizard control itself has the common Web control style properties - `BackColor`, `BorderStyle`, `CssClass`, `Font`, and so on - and there are a number of style properties for defining the appearance for particular sections of the CreateUserWizard's interface. The `TextBoxStyle` property, for instance, defines the styles for the textboxes in the `CreateUserWizardStep`, while the `TitleTextStyle` property defines the style for the title (Sign Up for Your New Account).

In addition to the appearance-related properties, there are a number of properties that affect the CreateUserWizard control's behavior. The `DisplayCancelButton` property, if set to True, displays a Cancel button next to the Create User button (the default value is False). If you display the Cancel button, be sure to also set the `CancelDestinationPageUrl` property, which specifies the page the user is sent to after clicking Cancel. As noted in the previous section, the Continue button in the `CompleteWizardStep`'s interface causes a postback, but leaves the visitor on the same page. To send the visitor to some other page after clicking the Continue button, simply specify the URL in the `ContinueDestinationPageUrl` property.

Let's update the `RegisterUser` CreateUserWizard control to show a Cancel button and to send the visitor to `Default.aspx` when the Cancel or Continue buttons are clicked. To accomplish this, set the `DisplayCancelButton` property to True and both the `CancelDestinationPageUrl` and `ContinueDestinationPageUrl` properties to ~/Default.aspx. Figure 14 shows the updated CreateUserWizard when viewed through a browser.

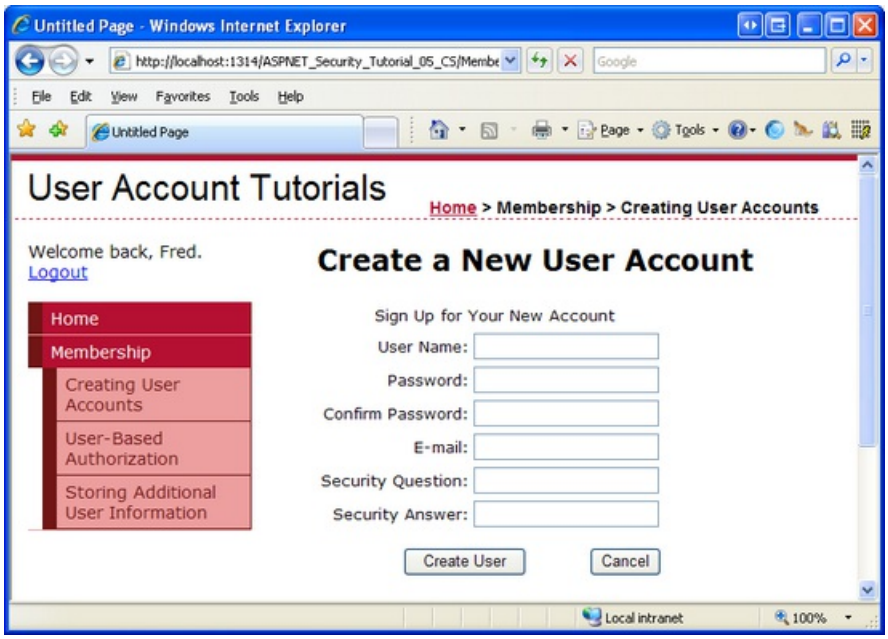


Figure 14: The `CreateUserWizardStep` Includes a Cancel Button (Click to view full-size image)

When a visitor enters a username, password, email address, and security question and answer and clicks Create User, a new user account is created and the visitor is logged in as that newly created user. Assuming that the person visiting the page is creating a new account for themselves, this is likely the desired behavior. However, you may want to allow Administrators to add new user accounts. In doing so, the user account would be created, but the Administrator would remain logged in as an Administrator (and not as the newly created account). This behavior can be modified through the Boolean `LoginCreatedUser` property.

User accounts in the Membership framework contain an approved flag; users who are not approved are unable to log into the site. By default, a newly created account is marked as approved, allowing the user to log into the site immediately. It is possible, however, to have new user accounts marked as unapproved. Perhaps you want an Administrator to manually approve new users before they can log in; or maybe you want to verify that the email address entered at signup is valid before permitting a user to log on. Whatever the case may be, you can have the newly created user account marked as unapproved by setting the CreateUserWizard control's `DisableCreatedUser` property to True (the default is False).

Other behavior-related properties of note include `AutoGeneratePassword` and `MailDefinition`. If the `AutoGeneratePassword` property is set to True, the `CreateUserWizardStep` does not display the Password and Confirm Password textboxes; instead, the newly-created user's password is automatically generated using the `Membership` class's `GeneratePassword` method. The `GeneratePassword` method constructs a password of a specified length and with a sufficient number of non-alphanumeric characters to satisfy the configured password strength requirements.

The `MailDefinition` property is useful if you want to send an email to the email address specified during the account creation process. The `MailDefinition` property includes a series of subproperties for defining information about the constructed email message. These subproperties include options like `Subject`, `Priority`, `IsBodyHtml`, `From`, `CC`, and `BodyFileName`. The `BodyFileName` property points to a text or HTML file that contains the body for the email message. The body supports two pre-defined placeholders: `<%UserName%>` and `<%Password%>`. These placeholders, if present in the `BodyFileName` file, will be replaced with the just-created user's name and password.

> **NOTE**
>
> The `CreateUserWizard` control's `MailDefinition` property just specifies details about the email message that is sent when a new account is created. It does not include any details on how the email message is actually sent (that is, whether an SMTP server or mail drop directory is used, any authentication information, and so on). These low-level details need to be defined in the `<system.net>` section in `Web.config`. For more information on these configuration settings and on sending email from ASP.NET 2.0 in general, refer to the FAQs at SystemNetMail.com and my article, Sending Email in ASP.NET 2.0.

Extending the CreateUserWizard's Behavior Using Event Handlers

The CreateUserWizard control raises a number of events during its workflow. For example, after a visitor enters their username, password, and other pertinent information and clicks the Create User button, the CreateUserWizard control raises its `CreatingUser` event. If there is a problem during the create process, the `CreateUserError` event is fired; however, if the user is successfully created, then the `CreatedUser` event is raised. There are additional CreateUserWizard control events that get raised, but these are the three most germane ones.

In certain scenarios we may want to tap into the CreateUserWizard workflow, which we can do by creating an event handler for the appropriate event. To illustrate this, let's enhance the `RegisterUser` CreateUserWizard control to include some custom validation on the username and password. In particular, let's enhance our CreateUserWizard so that usernames cannot contain leading or trailing spaces and the username cannot appear anywhere in the password. In short, we want to prevent someone from creating a username like "Scott ", or having a username/password combination like Scott and Scott.1234 .

To accomplish this we will create an event handler for the `CreatingUser` event to perform our extra validation checks. If the supplied data is not valid we need to cancel the creation process. We also need to add a Label Web control to the page to display a message explaining that the username or password is invalid. Start by adding a Label control beneath the CreateUserWizard control, setting its `ID` property to `InvalidUserNameOrPasswordMessage` and its `ForeColor` property to `Red`. Clear out its `Text` property and set its `EnableViewState` and `Visible` properties to False.

```
<asp:Label runat="server"" id="InvalidUserNameOrPasswordMessage"
 Visible="false" ForeColor="Red" EnableViewState="false">
</asp:Label>
```

Next, create an event handler for the CreateUserWizard control's `CreatingUser` event. To create an event handler, select the control in the Designer and then go to the Properties window. From there, click on the lightning bolt icon and then double-click the appropriate event to create the event handler.

Add the following code to the `CreatingUser` event handler:

```
Protected Sub RegisterUser_CreatingUser(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.LoginCancelEventArgs) Handles RegisterUser.CreatingUser
 Dim trimmedUserName As String = RegisterUser.UserName.Trim()
 If RegisterUser.UserName.Length <> trimmedUserName.Length Then
 ' Show the error message
 InvalidUserNameOrPasswordMessage.Text = "The username cannot contain leading or trailing spaces."
 InvalidUserNameOrPasswordMessage.Visible = True
 ' Cancel the create user workflow
 e.Cancel = True
 Else
 ' Username is valid, make sure that the password does not contain the username
 If RegisterUser.Password.IndexOf(RegisterUser.UserName, StringComparison.OrdinalIgnoreCase) >= 0 Then
 ' Show the error message
 InvalidUserNameOrPasswordMessage.Text = "The username may not appear anywhere in the password."
 InvalidUserNameOrPasswordMessage.Visible = True
 ' Cancel the create user workflow
 e.Cancel = True
 End If
 End If
End Sub
```

Note that the username and password entered into the CreateUserWizard control are available through its `UserName` and `Password` properties, respectively. We use these properties in the above event handler to determine whether the supplied username contains leading or trailing spaces and whether the username is found within the password. If either of these conditions is met, an error message is displayed in the `InvalidUserNameOrPasswordMessage` Label and the event handler's `e.Cancel` property is set to `True`. If `e.Cancel` is set to `True`, the CreateUserWizard short-circuits its workflow, effectively cancelling the user account creation process.

Figure 15 shows a screen shot of `CreatingUserAccounts.aspx` when the user enters a username with leading spaces.

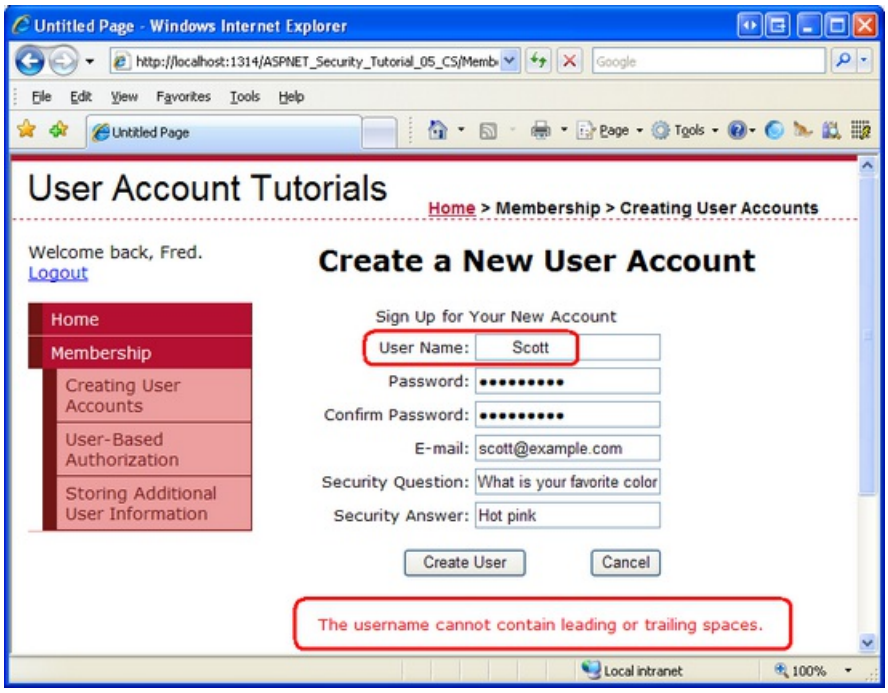


Figure 15: Usernames with Leading or Trailing Spaces are not Permitted (Click to view full-size image)

> **NOTE**
>
> We will see an example of using the CreateUserWizard control's `CreatedUser` event in the *Storing Additional User Information* tutorial.

Summary

The `Membership` class's `CreateUser` method creates a new user account in the Membership framework. It does so by delegating the call to the configured Membership provider. In the case of the `SqlMembershipProvider`, the `CreateUser` method adds a record to the `aspnet_Users` and `aspnet_Membership` database tables.

While new user accounts can be created programmatically (as we saw in Step 5), a faster and easier approach is to use the CreateUserWizard control. This control renders a multi-step user interface for collecting user information and creating a new user in the Membership framework. Underneath the covers, this control uses the same `Membership.CreateUser` method as examined in Step 5, but the control creates the user interface, validation controls, and responds to user account creation errors without having to write a lick of code.

At this point we have the functionality in place to create new user accounts. However, the login page still is validating against those hard-coded credentials we specified back in the second tutorial. In the next tutorial we will update `Login.aspx` to validate the user's supplied credentials against the Membership framework.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- `CreateUser` Technical Documentation
- CreateUserWizard Control Overview
- Creating a File System-Based Site Map Provider
- Creating a Step-by-Step User Interface with the ASP.NET 2.0 Wizard Control
- Examining ASP.NET 2.0's Site Navigation
- Master Pages and Site Navigation
- The SQL Site Map Provider You've Been Waiting For

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Validating User Credentials Against the Membership User Store (VB)

2/20/2020 • 25 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> In this tutorial we will examine how to validate a user's credentials against the Membership user store using both programmatic means and the Login control. We will also look at how to customize the login control's appearance and behavior.

Introduction

In the preceding tutorial we looked at how to create a new user account in the Membership framework. We first looked at programmatically creating user accounts via the `Membership` class's `CreateUser` method, and then examined using the CreateUserWizard Web control. However, the login page currently validates the supplied credentials against a hard-coded list of username and password pairs. We need to update the login page's logic so that it validates credentials against the Membership framework's user store.

Much like with creating user accounts, credentials can be validated programmatically or declaratively. The Membership API includes a method for programmatically validating a user's credentials against the user store. And ASP.NET ships with the Login Web control, which renders a user interface with textboxes for the username and password and a button to log in.

In this tutorial we will examine how to validate a user's credentials against the Membership user store using both programmatic means and the Login control. We will also look at how to customize the login control's appearance and behavior. Let's get started!

Step 1: Validating Credentials Against the Membership User Store

For web sites that use forms authentication, a user logs on to the website by visiting a login page and entering their credentials. These credentials are then compared against the user store. If they are valid, then the user is granted a forms authentication ticket, which is a security token that indicates the identity and authenticity of the visitor.

To validate a user against the Membership framework, use the `Membership` class's `ValidateUser` method. The `ValidateUser` method takes in two input parameters - `username` and `password` - and returns a Boolean value indicating whether the credentials were valid. Like with the `CreateUser` method we examined in the previous tutorial, the `ValidateUser` method delegates the actual validation to the configured Membership provider.

The `SqlMembershipProvider` validates the supplied credentials by obtaining the specified user's password via the `aspnet_Membership_GetPasswordWithFormat` stored procedure. Recall that the `SqlMembershipProvider` stores users' passwords using one of three formats: clear, encrypted, or hashed. The `aspnet_Membership_GetPasswordWithFormat` stored procedure returns the password in its raw format. For encrypted or hashed passwords, the `SqlMembershipProvider` transforms the `password` value passed into the `ValidateUser` method into its equivalent encrypted or hashed state and then compares it with what was returned from the database. If the password stored in the database matches the formatted password entered by the user, the credentials are valid.

Let's update our login page (~/`Login.aspx`) so that it validates the supplied credentials against the Membership

framework user store. We created this login page back in the *An Overview of Forms Authentication* tutorial, creating an interface with two TextBoxes for the username and password, a Remember Me checkbox, and a Login button (see Figure 1). The code validates the entered credentials against a hard-coded list of username and password pairs (Scott/password, Jisun/password, and Sam/password). In the *Forms Authentication Configuration and Advanced Topics* tutorial we updated the login page's code to store additional information in the forms authentication ticket's `UserData` property.

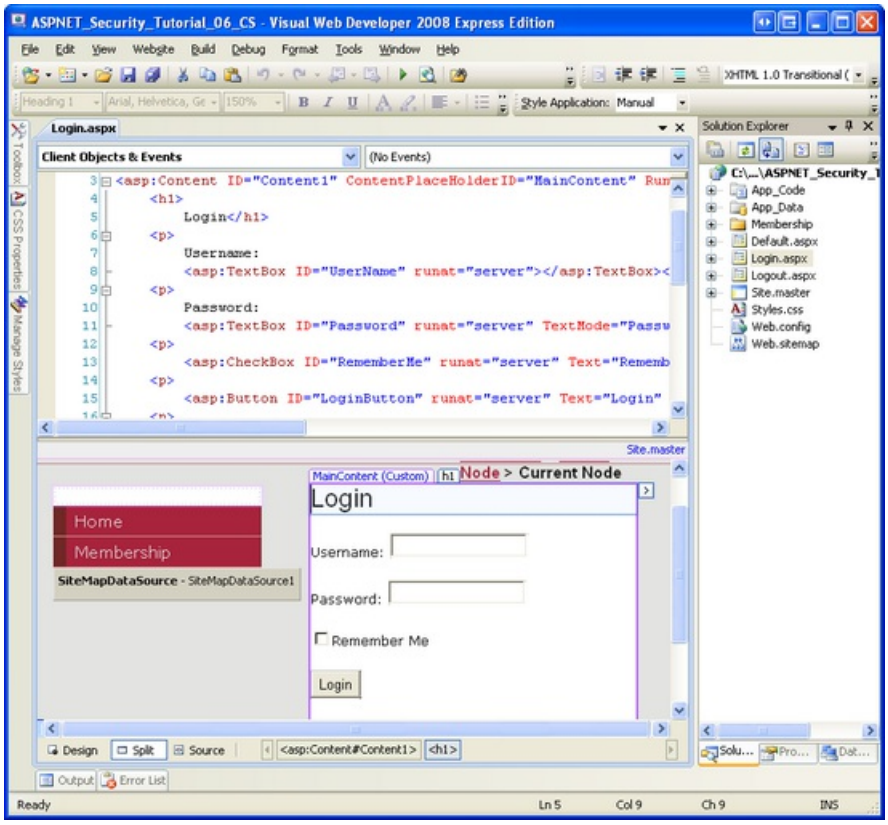


Figure 1: The Login Page's Interface Includes Two TextBoxes, a CheckBoxList, and a Button (Click to view full-size image)

The login page's user interface can remain unchanged, but we need to replace the Login button's `Click` event handler with code that validates the user against the Membership framework user store. Update the event handler so that its code appears as follows:

```
Protected Sub LoginButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
LoginButton.Click

 ' Validate the user against the Membership framework user store
 If Membership.ValidateUser(UserName.Text, Password.Text) Then
 ' Log the user into the site
 FormsAuthentication.RedirectFromLoginPage(UserName.Text, RememberMe.Checked)
 End If

 ' If we reach here, the user's credentials were invalid
 InvalidCredentialsMessage.Visible = True
End Sub
```

This code is remarkably simple. We start by calling the `Membership.ValidateUser` method, passing in the supplied username and password. If that method returns True, then the user is signed into the site via the `FormsAuthentication` class's RedirectFromLoginPage method. (As we discussed in the *An Overview of Forms Authentication* tutorial, the `FormsAuthentication.RedirectFromLoginPage` creates the forms authentication ticket and then redirects the user to the appropriate page.) If the credentials are invalid, however, the `InvalidCredentialsMessage` Label is displayed, informing the user that their username or password was incorrect.

That's all there is to it!

To test that the login page works as expected, attempt to login with one of the user accounts you created in the preceding tutorial. Or, if you have not yet created an account, go ahead and create one from the `~/Membership/CreatingUserAccounts.aspx` page.

> **NOTE**
>
> When the user enters her credentials and submits the login page form, the credentials, including her password, are transmitted over the Internet to the web server in *plain text*. That means any hacker sniffing the network traffic can see the username and password. To prevent this, it is essential to encrypt the network traffic by using Secure Socket Layers (SSL). This will ensure that the credentials (as well as the entire page's HTML markup) are encrypted from the moment they leave the browser until they are received by the web server.

How the Membership Framework Handles Invalid Login Attempts

When a visitor reaches the login page and submits their credentials, their browser makes an HTTP request to the login page. If the credentials are valid, the HTTP response includes the authentication ticket in a cookie. Therefore, a hacker attempting to break into your site could create a program that exhaustively sends HTTP requests to the login page with a valid username and a guess at the password. If the password guess is correct, the login page will return the authentication ticket cookie, at which point the program knows it has stumbled upon a valid username/password pair. Through brute force, such a program might be able to stumble upon a user's password, especially if the password is weak.

To prevent such brute force attacks, the Membership framework locks out a user if there are a certain number of unsuccessful login attempts within a certain period of time. The exact parameters are configurable through the following two Membership provider configuration settings:

- `maxInvalidPasswordAttempts` - specifies how many invalid password attempts are allowed for the user within the time period before the account is locked out. The default value is 5.
- `passwordAttemptWindow` - indicates the time period in minutes during which the specified number of invalid login attempts will cause the account to be locked out. The default value is 10.

If a user has been locked out, she cannot login until an administrator unlocks her account. When a user is locked out, the `ValidateUser` method will *always* return `False`, even if valid credentials are supplied. While this behavior lessens the likelihood that a hacker will break into your site through brute force methods, it can end up locking out a valid user who has simply forgotten her password or accidentally has the Caps Lock on or is having a bad typing day.

Unfortunately, there is no built-in tool for unlocking a user account. In order to unlock an account, you can modify the database directly - change the `IsLockedOut` field in the `aspnet_Membership` table for the appropriate user account - or create a web-based interface that lists locked out accounts with options to unlock them. We will examine creating administrative interfaces for accomplishing common user account- and role-related tasks in a future tutorial.

> **NOTE**
>
> One downside of the `ValidateUser` method is that when the supplied credentials are invalid, it does not provide any explanation as to why. The credentials may be invalid because there is no matching username/password pair in the user store, or because the user has not yet been approved, or because the user has been locked out. In Step 4 we will see how to show a more detailed message to the user when their login attempt fails.

Step 2: Collecting Credentials through the Login Web Control

The Login Web control renders a default user interface very similar to the one we created back in the *An*

Overview of Forms Authentication tutorial. Using the Login control saves us the work of having to create the interface to collect the visitor's credentials. Moreover, the Login control automatically signs in the user (assuming the submitted credentials are valid), thereby saving us from having to write any code.

Let's update `Login.aspx`, replacing the manually created interface and code with a Login control. Start by removing the existing markup and code in `Login.aspx`. You may delete it outright, or simply comment it out. To comment out declarative markup, surround it with the `<%--` and `--%>` delimiters. You can enter these delimiters manually, or, as Figure 2 shows, you may select the text to comment out and then click the Comment out the selected lines icon in the Toolbar. Similarly, you can use the Comment out the selected lines icon to comment out the selected code in the code-behind class.

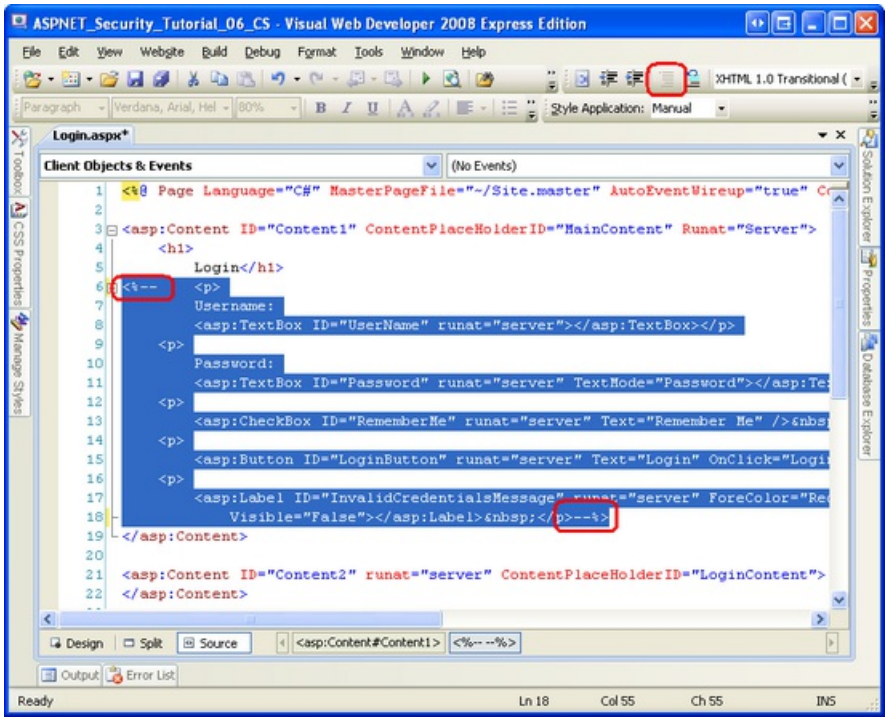


Figure 2: Comment Out the Existing Declarative Markup and Source Code in Login.aspx (Click to view full-size image)

> **NOTE**
>
> The Comment out the selected lines icon is not available when viewing the declarative markup in Visual Studio 2005. If you are not using Visual Studio 2008 you will need to manually add the `<%--` and `--%>` delimiters.

Next, drag a Login control from the Toolbox on to the page and set its `ID` property to `myLogin`. At this point your screen should look similar to Figure 3. Note that the Login control's default interface includes TextBox controls for the username and password, a Remember me next time CheckBox, and a Log In Button. There are also `RequiredFieldValidator` controls for the two TextBoxes.

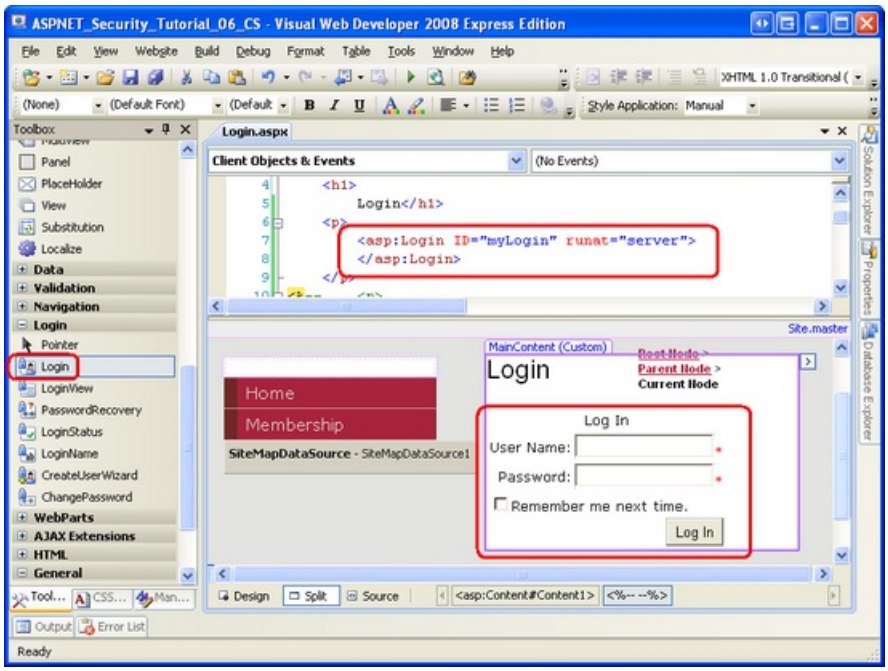


Figure 3: Add a Login Control to the Page (Click to view full-size image)

And we're done! When the Login control's Log In button is clicked, a postback will occur and the Login control will call the `Membership.ValidateUser` method, passing in the entered username and password. If the credentials are invalid, the Login control displays a message stating such. If, however, the credentials are valid, then the Login control creates the forms authentication ticket and redirects the user to the appropriate page.

The Login control uses four factors to determine the appropriate page to redirect the user to upon a successful login:

- Whether the Login control is on the login page as defined by `loginUrl` setting in the forms authentication configuration; this setting's default value is `Login.aspx`
- The presence of a `ReturnUrl` querystring parameter
- The value of the Login control's `DestinationUrl` property
- The `defaultUrl` value specified in the forms authentication configuration settings; this setting's default value is Default.aspx

Figure 4 depicts the how the Login control uses these four parameters to arrive at its appropriate page decision.

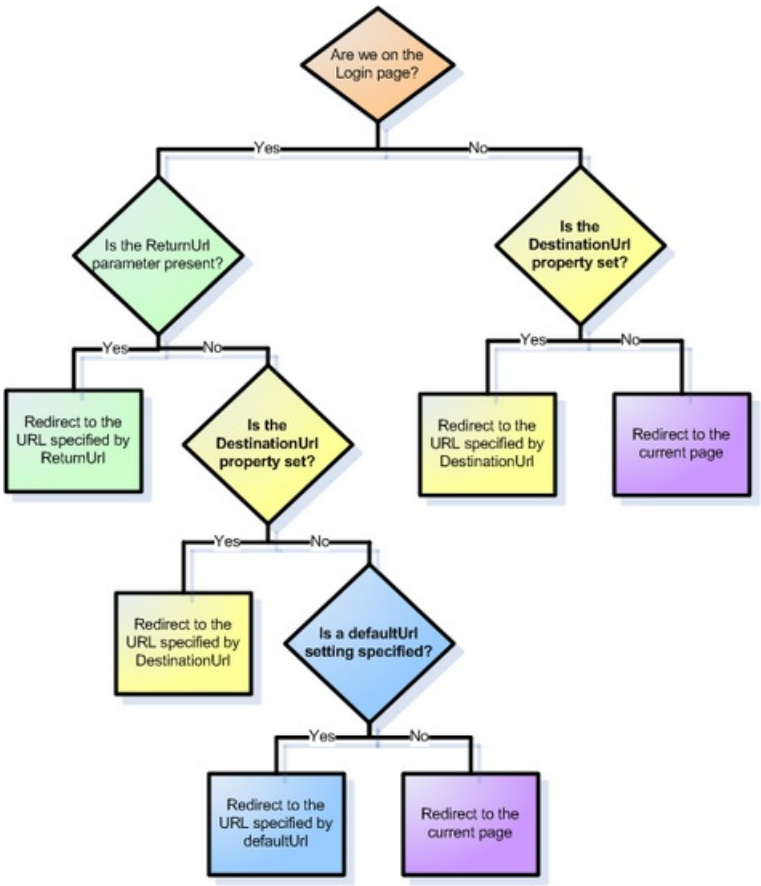


Figure 4: Add a Login Control to the Page (Click to view full-size image)

Take a moment to test out the Login control by visiting the site through a browser and logging in as an existing user in the Membership framework.

The Login control's rendered interface is highly configurable. There are a number of properties that influence its appearance; what's more, the Login control can be converted into a template for precise control over the layout the user interface elements. The remainder of this step examines how to customize the appearance and layout.

Customizing the Login Control's Appearance

The Login control's default property settings render a user interface with a title (Log In), TextBox and Label controls for the username and password inputs, a Remember me next time CheckBox, and a Log In button. The appearances of these elements are all configurable through the Login control's numerous properties. Furthermore, additional user interface elements - such as a link to a page to create a new user account - can be added by setting a property or two.

Let's spend a few moments to gussy up the appearance of our Login control. Since the `Login.aspx` page already has text at the top of the page that says Login, the Login control's title is superfluous. Therefore, clear out the `TitleText` property value in order to remove the Login control's title.

The User Name: and Password: Labels to the left of the two TextBox controls can be customized through the `UserNameLabelText` and `PasswordLabelText` properties, respectively. Let's change the User Name: Label to read Username:. The Label and TextBox styles are configurable through the `LabelStyle` and `TextBoxStyle` properties, respectively.

The Remember me next time CheckBox's Text property can be set through the Login control's `RememberMeText` property, and its default checked state is configurable through the `RememberMeSet` property (which defaults to False). Go ahead and set the `RememberMeSet` property to True so that the Remember me next time CheckBox will be checked by default.

The Login control offers two properties for adjusting the layout of its user interface controls. The `TextLayout` property indicates whether the Username: and Password: Labels appear to the left of their corresponding

TextBoxes (the default), or above them. The `Orientation` property indicates whether the username and password inputs are situated vertically (one above the other) or horizontally. I am going to leave these two properties set to their defaults, but I encourage you to try setting these two properties to their non-default values to see the resulting effect.

> **NOTE**
>
> In the next section, Configuring the Login Control's Layout, we will look at using templates to define the precise layout of the Layout control's user interface elements.

Wrap up the Login control's property settings by setting the `CreateUserText` and `CreateUserUrl` properties to Not registered yet? Create an account! and `~/Membership/CreatingUserAccounts.aspx`, respectively. This adds a hyperlink to the Login control's interface pointing to the page we created in the preceding tutorial. The Login control's `HelpPageText` and `HelpPageUrl` properties and `PasswordRecoveryText` and `PasswordRecoveryUrl` properties work in the same manner, rendering links to a help page and a password recovery page.

After making these property changes, your Login control's declarative markup and appearance should look similar to that shown in Figure 5.

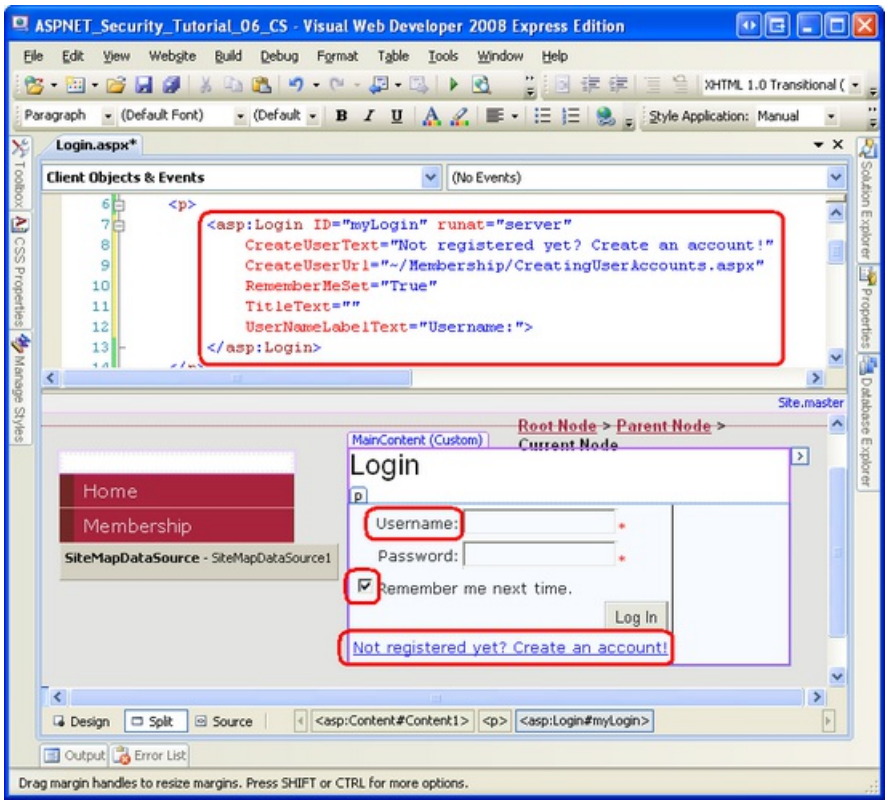


Figure 5: The Login Control's Properties' Values Dictate Its Appearance (Click to view full-size image)

Configuring the Login Control's Layout

The Login Web control's default user interface lays out the interface in an HTML `<table>`. But what if we need finer control over the rendered output? Maybe we want to replace the `<table>` with a series of `<div>` tags. Or what if our application requires additional credentials for authentication? Many financial websites, for instance, require users to supply not only a username and password, but also a Personal Identification Number (PIN), or other identifying information. Whatever the reasons may be, it is possible to convert the Login control into a template, from which we can explicitly define the interface's declarative markup.

We need to do two things in order to update the Login control to collect additional credentials:

1. Update the Login control's interface to include Web control(s) to collect the additional credentials.
2. Override the Login control's internal authentication logic so that a user is only authenticated if their username

and password is valid and their additional credentials are valid, too.

To accomplish the first task, we need to convert the Login control into a template and add the necessary Web controls. As for the second task, the Login control's authentication logic can be superseded by creating an event handler for the control's `Authenticate` event.

Let's update the Login control so that it prompts users for their username, password, and email address and only authenticates the user if the email address supplied matches their email address on file. We first need to convert the Login control's interface to a template. From the Login control's Smart Tag, choose the Convert to template option.

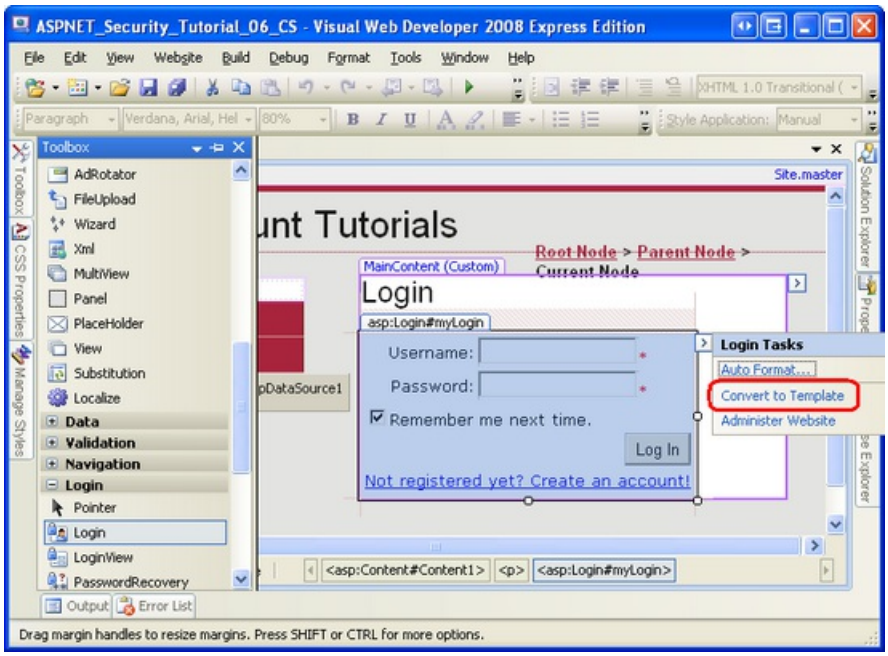


Figure 6: Convert the Login Control to a Template (Click to view full-size image)

> **NOTE**
>
> To revert the Login control to its pre-template version, click the Reset link from the control's Smart Tag.

Converting the Login control to a template adds a `LayoutTemplate` to the control's declarative markup with HTML elements and Web controls defining the user interface. As Figure 7 shows, converting the control to a template removes a number of properties from the Properties window, such as `TitleText`, `CreateUserUrl`, and so forth, since these property values are ignored when using a template.

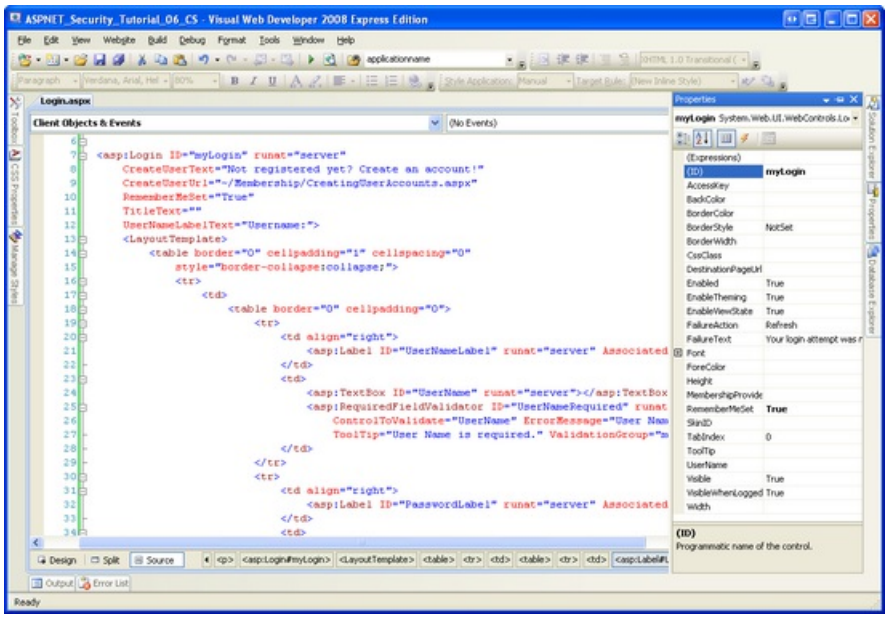


Figure 7: Fewer Properties are Available When the Login Control is Converted to a Template (Click to view full-size image)

The HTML markup in the `LayoutTemplate` may be modified as needed. Likewise, feel free to add any new Web controls to the template. However, it is important that Login control's core Web controls remain in the template and keep their assigned `ID` values. In particular, do not remove or rename the `UserName` or `Password` TextBoxes, the `RememberMe` CheckBox, the `LoginButton` Button, the `FailureText` Label, or the `RequiredFieldValidator` controls.

To collect the visitor's email address, we need to add a TextBox to the template. Add the following declarative markup between the table row (`<tr>`) that contains the `Password` TextBox and the table row that holds the Remember me next time CheckBox:

```
<tr>
 <td align="right">
 <asp:Label ID="EmailLabel" runat="server" AssociatedControlID="Email">Email:</asp:Label>
 </td>
 <td>
 <asp:TextBox ID="Email" runat="server"></asp:TextBox>
 <asp:RequiredFieldValidator ID="EmailRequired" runat="server"
 ControlToValidate="Email" ErrorMessage="Email is required."
 ToolTip="Email is required." ValidationGroup="myLogin">*</asp:RequiredFieldValidator>
 </td>
</tr>
```

After adding the `Email` TextBox, visit the page through a browser. As Figure 8 shows, the Login control's user interface now includes a third textbox.

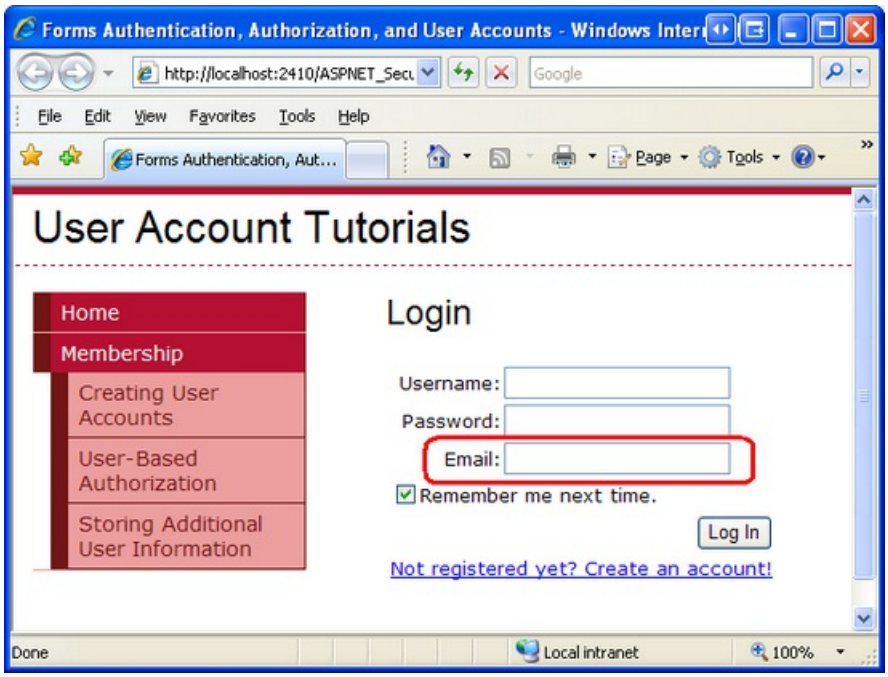


Figure 8: The Login Control Now Includes a Textbox for the User's Email Address (Click to view full-size image)

At this point, the Login control is still using the `Membership.ValidateUser` method to validate the supplied credentials. Correspondingly, the value entered into the `Email` TextBox has no bearing on whether the user can log in. In Step 3 we will look at how to override the Login control's authentication logic so that the credentials are only considered valid if the username and password are valid and the supplied email address matches up with the email address on file.

Step 3: Modifying the Login Control's Authentication Logic

When a visitor supplies her credentials and clicks the Log In button, a postback ensues and the Login control progresses through its authentication workflow. The workflow starts by raising the `LoggingIn` event. Any event handlers associated with this event may cancel the log in operation by setting the `e.Cancel` property to `True`.

If the log in operation is not cancelled, the workflow progresses by raising the `Authenticate` event. If there is an event handler for the `Authenticate` event, then it is responsible for determining whether the supplied credentials are valid or not. If no event handler is specified, the Login control uses the `Membership.ValidateUser` method to determine the validity of the credentials.

If the supplied credentials are valid, then the forms authentication ticket is created, the `LoggedIn` event is raised, and the user is redirected to the appropriate page. If, however, the credentials are deemed invalid, then the `LoginError` event is raised and a message is displayed informing the user that their credentials were invalid. By default, on failure the Login control simply sets its `FailureText` Label control's Text property to a failure message (Your login attempt was not successful. Please try again). However, if the Login control's `FailureAction` property is set to `RedirectToLoginPage`, then the Login control issues a `Response.Redirect` to the login page appending the querystring parameter `loginfailure=1` (which causes the Login control to display the failure message).

Figure 9 offers a flow chart of the authentication workflow.

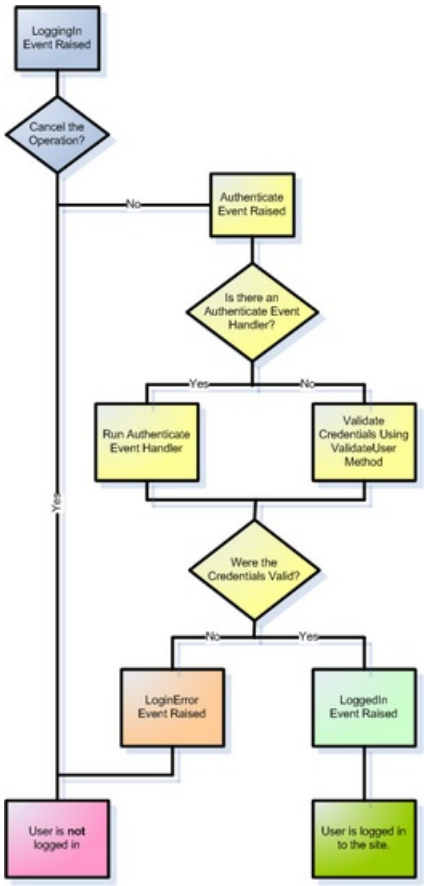


Figure 9: The Login Control's Authentication Workflow (Click to view full-size image)

> **NOTE**
> If you are wondering when you would use the `FailureAction`'s `RedirectToLogin` page option, consider the following scenario. Right now our `Site.master` master page currently has the text, Hello, stranger displayed in the left column when visited by an anonymous user, but imagine that we wanted to replace that text with a Login control. This would allow an anonymous user to log in from any page on the site, instead of requiring them to visit the login page directly. However, if a user was unable to log in via the Login control rendered by the master page, it might make sense to redirect them to the login page (`Login.aspx`) because that page likely includes additional instructions, links, and other help - such as links to create a new account or retrieve a lost password - that were not added to the master page.

Creating the `Authenticate` Event Handler

In order to plug in our custom authentication logic, we need to create an event handler for the Login control's `Authenticate` event. Creating an event handler for the `Authenticate` event will generate the following event handler definition:

```
Protected Sub myLogin_Authenticate(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.AuthenticateEventArgs) Handles myLogin.Authenticate

End Sub
```

As you can see, the `Authenticate` event handler is passed an object of type `AuthenticateEventArgs` as its second input parameter. The `AuthenticateEventArgs` class contains a Boolean property named `Authenticated` that is used to specify whether the supplied credentials are valid. Our task, then, is to write code here that determines whether the supplied credentials are valid or not, and to set the `e.Authenticate` property accordingly.

Determining and Validating the Supplied Credentials

Use the Login control's `UserName` and `Password` properties to determine the username and password credentials

entered by the user. In order to determine the values entered into any additional Web controls (such as the `Email` TextBox we added in the previous step), use `LoginControlID.FindControl` ("*`controlID`*") to obtain a programmatic reference to the Web control in the template whose `ID` property is equal to *`controlID`*. For example, to get a reference to the `Email` TextBox, use the following code:

```
Dim EmailTextBox As TextBox = CType(myLogin.FindControl("Email"), TextBox)
```

In order to validate the user's credentials we need to do two things:

1. Ensure that the provided username and password are valid
2. Ensure that email address entered matches the email address on file for the user attempting to log in

To accomplish the first check we can simply use the `Membership.ValidateUser` method like we saw in Step 1. For the second check, we need to determine the user's email address so that we can compare it to the email address they entered into the TextBox control. To get information about a particular user, use the `Membership` class's `GetUser` method.

The `GetUser` method has a number of overloads. If used without passing in any parameters, it returns information about the currently logged in user. To get information about a particular user, call `GetUser` passing in their username. Either way, `GetUser` returns a `MembershipUser` object, which has properties like `UserName`, `Email`, `IsApproved`, `IsOnline`, and so on.

The following code implements these two checks. If both pass, then `e.Authenticate` is set to `True`, otherwise it is assigned `False`.

```
Protected Sub myLogin_Authenticate(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.AuthenticateEventArgs) Handles myLogin.Authenticate

 ' Get the email address entered
 Dim EmailTextBox As TextBox = CType(myLogin.FindControl("Email"), TextBox)
 Dim email As String = EmailTextBox.Text.Trim()

 ' Verify that the username/password pair is valid
 If Membership.ValidateUser(myLogin.UserName, myLogin.Password) Then

 ' Username/password are valid, check email
 Dim usrInfo As MembershipUser = Membership.GetUser(myLogin.UserName)

 If usrInfo IsNot Nothing AndAlso String.Compare(usrInfo.Email, email, True) = 0 Then
 ' Email matches, the credentials are valid
 e.Authenticated = True
 Else
 ' Email address is invalid...
 e.Authenticated = False
 End If
 Else
 ' Username/password are not valid...
 e.Authenticated = False
 End If
End Sub
```

With this code in place, attempt to log in as a valid user, entering the correct username, password, and email address. Try it again, but this time purposefully use an incorrect email address (see Figure 10). Finally, try it a third time using a non-existent username. In the first case you should be successfully logged on to the site, but in the last two cases you should see the Login control's invalid credentials message.

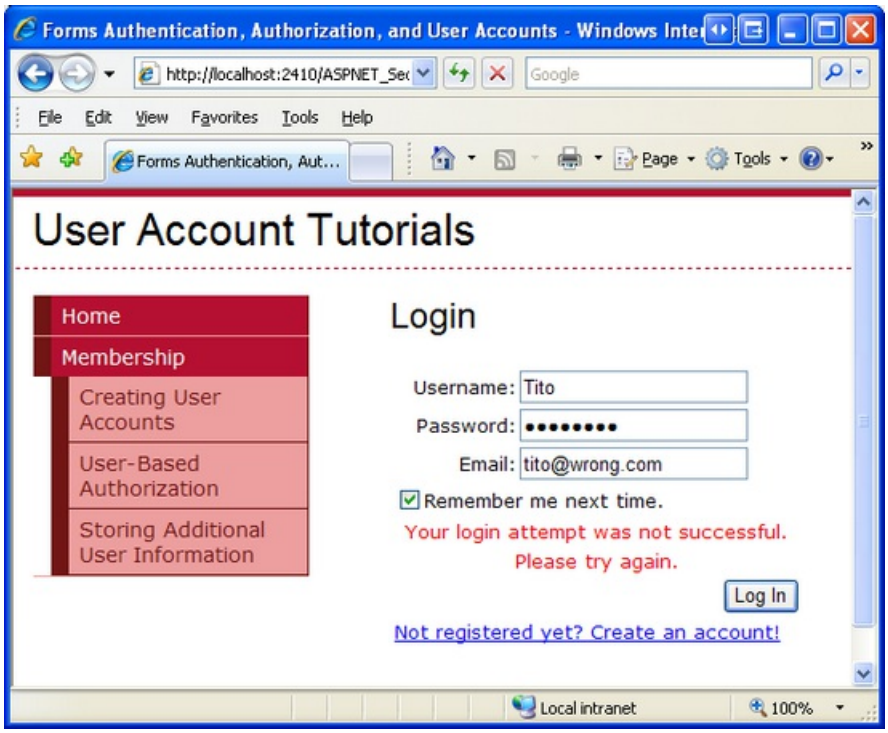


Figure 10: Tito Cannot Log In When Supplying an Incorrect Email Address (Click to view full-size image)

> **NOTE**
>
> As discussed in the How the Membership Framework Handles Invalid Login Attempts section in Step 1, when the `Membership.ValidateUser` method is called and passed invalid credentials, it keeps track of the invalid login attempt and locks out the user if they exceed a certain threshold of invalid attempts within a specified time window. Since our custom authentication logic calls the `ValidateUser` method, an incorrect password for a valid username will increment the invalid login attempt counter, but this counter is not incremented in the case where the username and password are valid, but the email address is incorrect. Chances are, this behavior is suitable, since it's unlikely that a hacker will know the username and password, but have to use brute force techniques to determine the user's email address.

Step 4: Improving the Login Control's Invalid Credentials Message

When a user attempts to log on with invalid credentials, the Login control displays a message explaining that the login attempt was unsuccessful. In particular, the control displays the message specified by its `FailureText` property, which has a default value of Your login attempt was not successful. Please try again.

Recall that there are many reasons why a user's credentials may be invalid:

- The username may not exist
- The username exists, but the password is invalid
- The username and password are valid, but the user is not yet approved
- The username and password are valid, but the user is locked out (most likely because they exceeded the number of invalid login attempts within the specified time frame)

And there may be other reasons when using custom authentication logic. For example, with the code we wrote in Step 3, the username and password may be valid, but the email address may be incorrect.

Regardless of why the credentials are invalid, the Login control displays the same error message. This lack of feedback can be confusing for a user whose account has not yet been approved or who has been locked out. With a little bit of work, though, we can have the Login control display a more appropriate message.

Whenever a user attempts to login with invalid credentials the Login control raises its `LoginError` event. Go ahead and create an event handler for this event, and add the following code:

```
Protected Sub myLogin_LoginError(ByVal sender As Object, ByVal e As System.EventArgs) Handles
myLogin.LoginError

 ' Determine why the user could not login...
 myLogin.FailureText = "Your login attempt was not successful. Please try again."

 ' Does there exist a User account for this user?
 Dim usrInfo As MembershipUser = Membership.GetUser(myLogin.UserName)

 If usrInfo IsNot Nothing Then
 ' Is this user locked out?
 If usrInfo.IsLockedOut Then
 myLogin.FailureText = "Your account has been locked out because of too many invalid login attempts. Please
contact the administrator to have your account unlocked."
 ElseIf Not usrInfo.IsApproved Then
 myLogin.FailureText = "Your account has not yet been approved. You cannot login until an administrator has
approved your account."
 End If
 End If
End Sub
```

The above code starts by setting the Login control's `FailureText` property to the default value (Your login attempt was not successful. Please try again). It then checks to see if the username supplied maps to an existing user account. If so, it consults the resulting `MembershipUser` object's `IsLockedOut` and `IsApproved` properties to determine if the account has been locked out or has not yet been approved. In either case, the `FailureText` property is updated to a corresponding value.

To test this code, purposefully attempt to log in as an existing user, but use an incorrect password. Do this five times in a row within a 10 minute time frame and the account will be locked out. As Figure 11 shows, subsequent login attempts will always fail (even with the correct password), but will now display the more descriptive Your account has been locked out because of too many invalid login attempts. Please contact the administrator to have your account unlocked message.

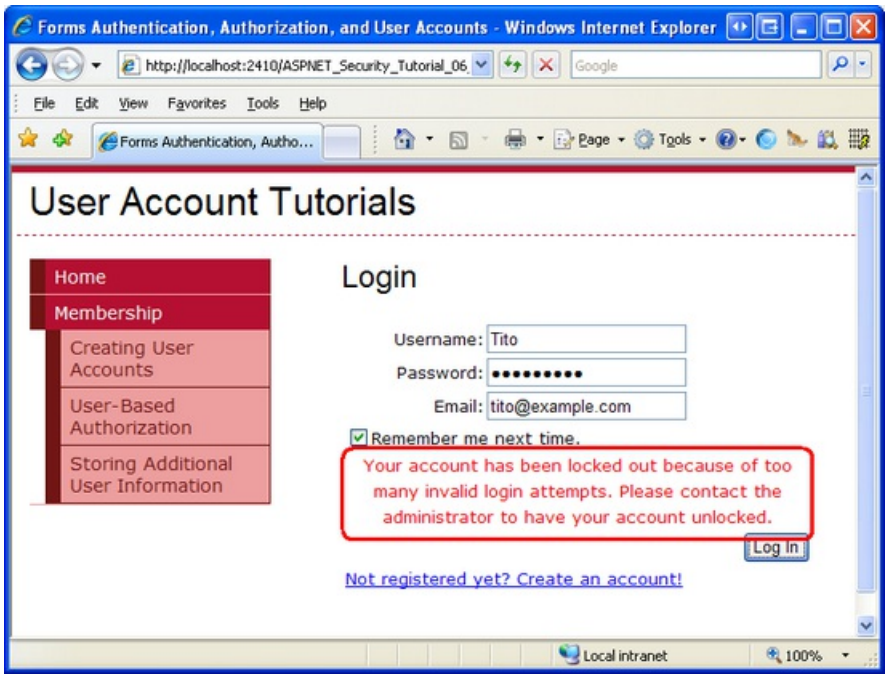


Figure 11: Tito Performed Too Many Invalid Login Attempts and Has Been Locked Out (Click to view full-size image)

Summary

Prior this tutorial, our login page validated the supplied credentials against a hard-coded list of

username/password pairs. In this tutorial, we updated the page to validate credentials against the Membership framework. In Step 1 we looked at using the `Membership.ValidateUser` method programmatically. In Step 2 we replaced our manually created user interface and code with the Login control.

The Login control renders a standard login user interface and automatically validates the user's credentials against the Membership framework. Moreover, in the event of valid credentials, the Login control signs the user in via forms authentication. In short, a fully functional login user experience is available by simply dragging the Login control onto a page, no extra declarative markup or code necessary. What's more, the Login control is highly customizable, allowing for a fine degree of control over both the rendered user interface and authentication logic.

At this point visitors to our website can create a new user account and log into the site, but we have yet to look at restricting access to pages based on the authenticated user. Currently, any user, authenticated or anonymous, can view any page on our site. Along with controlling access to our site's pages on a user-by-user basis, we might have certain pages whose functionality depends on the user. The next tutorial looks at how to limit access and in-page functionality based on the logged in user.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Displaying Custom Messages to Locked Out and Non-Approved Users
- Examining ASP.NET 2.0's Membership, Roles, and Profile
- How To: Create an ASP.NET Login Page
- Login Control Technical Documentation
- Using the Login Controls

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Teresa Murphy and Michael Olivero. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

User-Based Authorization (VB)

2/20/2020 • 33 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> In this tutorial we will look at limiting access to pages and restricting page-level functionality through a variety of techniques.

Introduction

Most web applications that offer user accounts do so in part to restrict certain visitors from accessing certain pages within the site. In most online messageboard sites, for example, all users - anonymous and authenticated - are able to view the messageboard's posts, but only authenticated users can visit the web page to create a new post. And there may be administrative pages that are only accessible to a particular user (or a particular set of users). Moreover, page-level functionality can differ on a user-by-user basis. When viewing a list of posts, authenticated users are shown an interface for rating each post, whereas this interface is not available to anonymous visitors.

ASP.NET makes it easy to define user-based authorization rules. With just a bit of markup in `Web.config`, specific web pages or entire directories can be locked down so that they are only accessible to a specified subset of users. Page-level functionality can be turned on or off based on the currently logged in user through programmatic and declarative means.

In this tutorial we will look at limiting access to pages and restricting page-level functionality through a variety of techniques. Let's get started!

A Look at the URL Authorization Workflow

As discussed in the *An Overview of Forms Authentication* tutorial, when the ASP.NET runtime processes a request for an ASP.NET resource the request raises a number of events during its lifecycle. *HTTP Modules* are managed classes whose code is executed in response to a particular event in the request lifecycle. ASP.NET ships with a number of HTTP Modules that perform essential tasks behind the scenes.

One such HTTP Module is `FormsAuthenticationModule`. As discussed in previous tutorials, the primary function of the `FormsAuthenticationModule` is to determine the identity of the current request. This is accomplished by inspecting the forms authentication ticket, which is either located in a cookie or embedded within the URL. This identification takes place during the `AuthenticateRequest` event.

Another important HTTP Module is the `UrlAuthorizationModule`, which is raised in response to the `AuthorizeRequest` event (which happens after the `AuthenticateRequest` event). The `UrlAuthorizationModule` examines configuration markup in `Web.config` to determine whether the current identity has authority to visit the specified page. This process is referred to as *URL authorization*.

We'll examine the syntax for the URL authorization rules in Step 1, but first let's look at what the `UrlAuthorizationModule` does depending on whether the request is authorized or not. If the `UrlAuthorizationModule` determines that the request is authorized, then it does nothing, and the request continues through its lifecycle. However, if the request is *not* authorized, then the `UrlAuthorizationModule` aborts the lifecycle and instructs the `Response` object to return an HTTP 401 Unauthorized status. When using forms authentication this HTTP 401 status is never returned to the client because if the `FormsAuthenticationModule`

detects an HTTP 401 status is modifies it to an HTTP 302 Redirect to the login page.

Figure 1 illustrates the workflow of the ASP.NET pipeline, the `FormsAuthenticationModule`, and the `UrlAuthorizationModule` when an unauthorized request arrives. In particular, Figure 1 shows a request by an anonymous visitor for `ProtectedPage.aspx`, which is a page that denies access to anonymous users. Since the visitor is anonymous, the `UrlAuthorizationModule` aborts the request and returns an HTTP 401 Unauthorized status. The `FormsAuthenticationModule` then converts the 401 status into a 302 Redirect to login page. After the user is authenticated via the login page, he is redirected to `ProtectedPage.aspx`. This time the `FormsAuthenticationModule` identifies the user based on his authentication ticket. Now that the visitor is authenticated, the `UrlAuthorizationModule` permits access to the page.

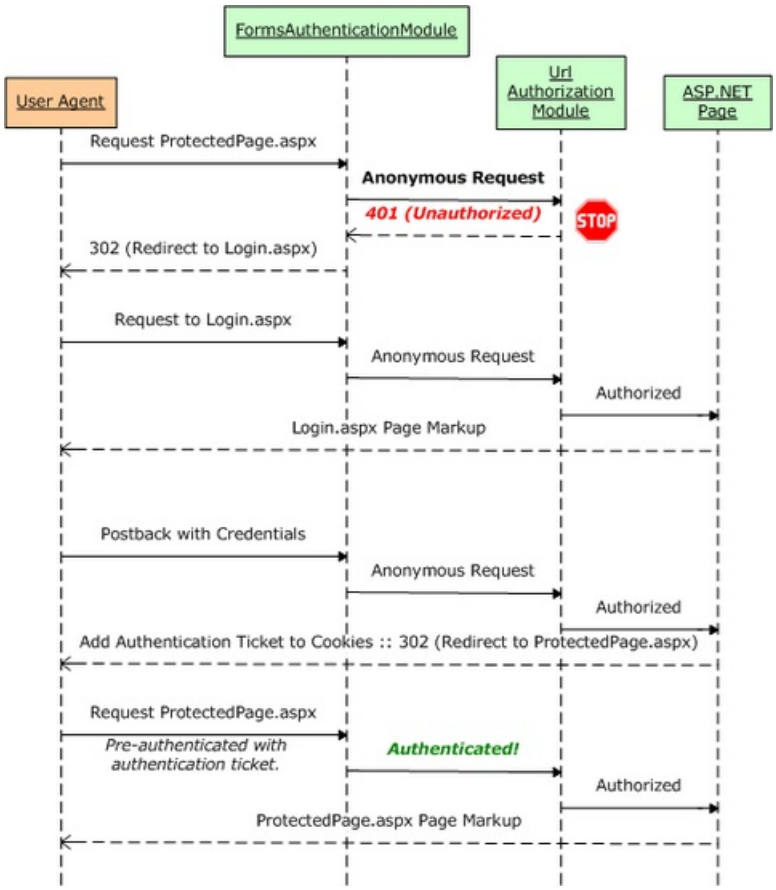


Figure 1: The Forms Authentication and URL Authorization Workflow (Click to view full-size image)

Figure 1 depicts the interaction that occurs when an anonymous visitor attempts to access a resource that is not available to anonymous users. In such a case, the anonymous visitor is redirected to the login page with the page she attempted to visit specified in the querystring. Once the user has successfully logged on, she will be automatically redirected back to the resource she was initially attempting to view.

When the unauthorized request is made by an anonymous user, this workflow is straightforward and is easy for the visitor to understand what has happened and why. But keep in mind that the `FormsAuthenticationModule` will redirect *any* unauthorized user to the login page, even if the request is made by an authenticated user. This can result in a confusing user experience if an authenticated user attempts to visit a page for which she lacks authority.

Imagine that our website had its URL authorization rules configured such that the ASP.NET page `OnlyTito.aspx` was accessibly only to Tito. Now, imagine that Sam visits the site, logs on, and then attempts to visit `OnlyTito.aspx`. The `UrlAuthorizationModule` will halt the request lifecycle and return an HTTP 401 Unauthorized status, which the `FormsAuthenticationModule` will detect and then redirect Sam to the login page. Since Sam has already logged in, though, she may wonder why she has been sent back to the login page. She might reason that her login credentials were lost somehow, or that she entered invalid credentials. If Sam reenters her credentials from the login page she will be logged on (again) and redirected to `OnlyTito.aspx`. The `UrlAuthorizationModule`

will detect that Sam cannot visit this page and she will be returned to the login page.

Figure 2 depicts this confusing workflow.

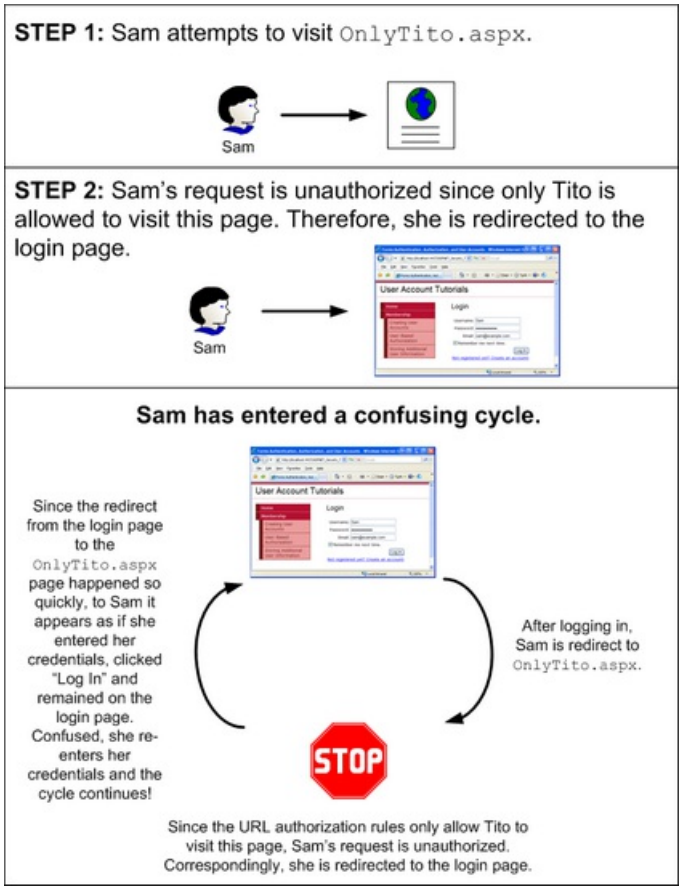


Figure 2: The Default Workflow Can Lead to a Confusing Cycle (Click to view full-size image)

The workflow illustrated in Figure 2 can quickly befuddle even the most computer savvy visitor. We will look at ways to prevent this confusing cycle in Step 2.

> **NOTE**
>
> ASP.NET uses two mechanisms to determine whether the current user can access a particular web page: URL authorization and file authorization. File authorization is implemented by the `FileAuthorizationModule`, which determines authority by consulting the requested file(s) ACLs. File authorization is most commonly used with Windows authentication because ACLs are permissions that apply to Windows accounts. When using forms authentication, all operating system- and file system-level requests are executed by the same Windows account, regardless of the user visiting the site. Since this tutorial series focuses on forms authentication, we will not be discussing file authorization.

The Scope of URL Authorization

The `UrlAuthorizationModule` is managed code that is part of the ASP.NET runtime. Prior to version 7 of Microsoft's Internet Information Services (IIS) web server, there was a distinct barrier between IIS's HTTP pipeline and the ASP.NET runtime's pipeline. In short, in IIS 6 and earlier, ASP.NET's `UrlAuthorizationModule` only executes when a request is delegated from IIS to the ASP.NET runtime. By default, IIS processes static content itself - like HTML pages and CSS, JavaScript, and image files - and only hands off requests to the ASP.NET runtime when a page with an extension of `.aspx`, `.asmx`, or `.ashx` is requested.

IIS 7, however, allows for integrated IIS and ASP.NET pipelines. With a few configuration settings you can setup IIS 7 to invoke the `UrlAuthorizationModule` for *all* requests, meaning that URL authorization rules can be defined for files of any type. Additionally, IIS 7 includes its own URL authorization engine. For more information on ASP.NET integration and IIS 7's native URL authorization functionality, see Understanding IIS7 URL Authorization. For a more in-depth look at ASP.NET and IIS 7 integration, pick up a copy of Shahram Khosravi's book,

Professional IIS 7 and ASP.NET Integrated Programming (ISBN: 978-0470152539).

In a nutshell, in versions prior to IIS 7, URL authorization rules are only applied to resources handled by the ASP.NET runtime. But with IIS 7 it is possible to use IIS's native URL authorization feature or to integrate ASP.NET's `UrlAuthorizationModule` into IIS's HTTP pipeline, thereby extending this functionality to all requests.

> **NOTE**
> There are some subtle yet important differences in how ASP.NET's `UrlAuthorizationModule` and IIS 7's URL authorization feature process the authorization rules. This tutorial does not examine IIS 7's URL authorization functionality or the differences in how it parses authorization rules compared to the `UrlAuthorizationModule`. For more information on these topics, refer to the IIS 7 documentation on MSDN or at www.iis.net.

Step 1: Defining URL Authorization Rules in `Web.config`

The `UrlAuthorizationModule` determines whether to grant or deny access to a requested resource for a particular identity based on the URL authorization rules defined in the application's configuration. The authorization rules are spelled out in the `<authorization>` element in the form of `<allow>` and `<deny>` child elements. Each `<allow>` and `<deny>` child element can specify:

- A particular user
- A comma-delimited list of users
- All anonymous users, denoted by a question mark (?)
- All users, denoted by an asterisk (*)

The following markup illustrates how to use the URL authorization rules to allow users Tito and Scott and deny all others:

```
<authorization>
 <allow users="Tito, Scott" />
 <deny users="*" />
</authorization>
```

The `<allow>` element defines what users are permitted - Tito and Scott - while the `<deny>` element instructs that *all* users are denied.

> **NOTE**
> The `<allow>` and `<deny>` elements can also specify authorization rules for roles. We will examine role-based authorization in a future tutorial.

The following setting grants access to anyone other than Sam (including anonymous visitors):

```
<authorization>
 <deny users="Sam" />
</authorization>
```

To allow only authenticated users, use the following configuration, which denies access to all anonymous users:

```
<authorization>
 <deny users="?" />
</authorization>
```

The authorization rules are defined within the `<system.web>` element in `Web.config` and apply to all of the ASP.NET resources in the web application. Oftentimes, an application has different authorization rules for different sections. For example, at an eCommerce site, all visitors may peruse the products, see product reviews, search the catalog, and so on. However, only authenticated users may reach the checkout or the pages to manage one's shipping history. Moreover, there may be portions of the site that are only accessible by select users, such as site administrators.

ASP.NET makes it easy to define different authorization rules for different files and folders in the site. The authorization rules specified in the root folder's `Web.config` file apply to all ASP.NET resources in the site. However, these default authorization settings can be overridden for a particular folder by adding a `Web.config` with an `<authorization>` section.

Let's update our website so that only authenticated users can visit the ASP.NET pages in the `Membership` folder. To accomplish this we need to add a `Web.config` file to the `Membership` folder and set its authorization settings to deny anonymous users. Right-click the `Membership` folder in the Solution Explorer, choose the Add New Item menu from the context menu, and add a new Web Configuration File named `Web.config`.

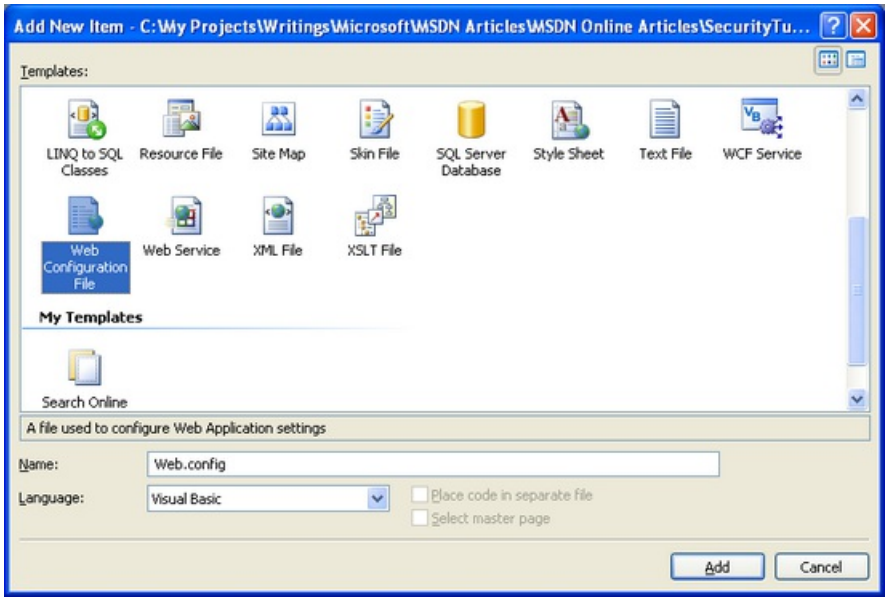


Figure 3: Add a `Web.config` File to the `Membership` Folder (Click to view full-size image)

At this point your project should contain two `Web.config` files: one in the root directory and one in the `Membership` folder.

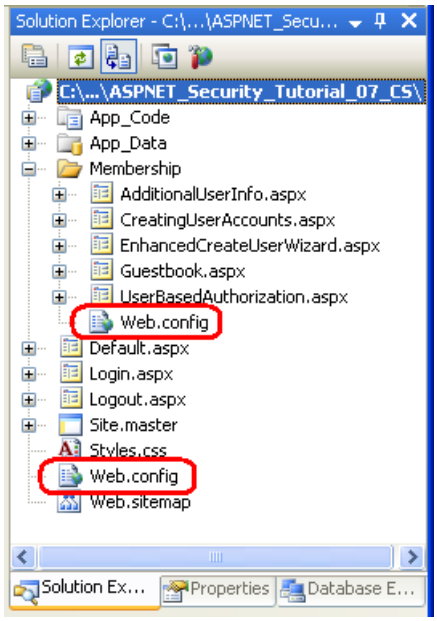


Figure 4: Your Application Should Now Contain Two `Web.config` Files (Click to view full-size image)

Update the configuration file in the `Membership` folder so that it prohibits access to anonymous users.

```
<?xml version="1.0"?>
<configuration>
 <system.web>
 <authorization>
 <deny users="?" />
 </authorization>
 </system.web>
</configuration>
```

That's all there is to it!

To test out this change, visit the homepage in a browser and make sure you are logged out. Since the default behavior of an ASP.NET application is to allow all visitors, and since we didn't make any authorization modifications to the root directory's `Web.config` file, we are able to visit the files in the root directory as an anonymous visitor.

Click on the Creating User Accounts link found in the left column. This will take you to the `~/Membership/CreatingUserAccounts.aspx`. Since the `Web.config` file in the `Membership` folder defines authorization rules to prohibit anonymous access, the `UrlAuthorizationModule` aborts the request and returns an HTTP 401 Unauthorized status. The `FormsAuthenticationModule` modifies this to a 302 Redirect status, sending us to the login page. Note that the page we were attempting to access (`CreatingUserAccounts.aspx`) is passed to the login page via the `ReturnUrl` querystring parameter.

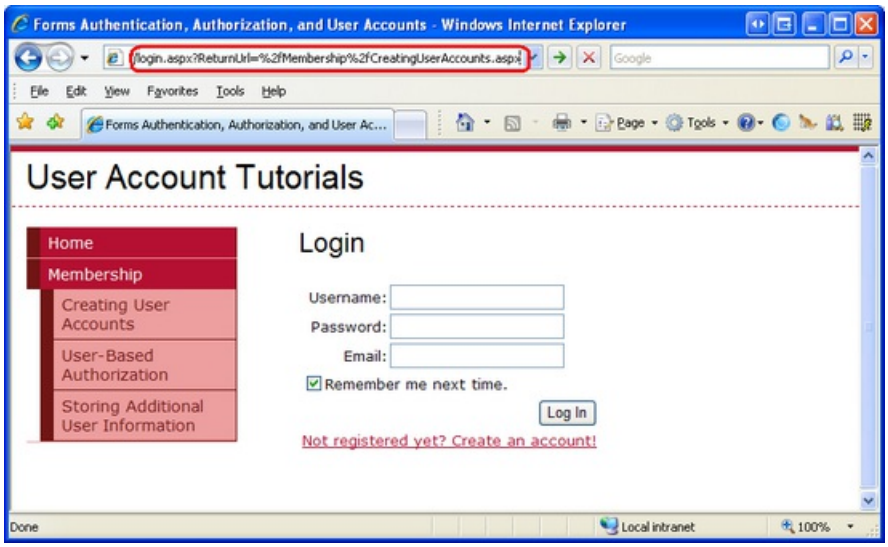


Figure 5: Since the URL Authorization Rules Prohibit Anonymous Access, We are Redirected to the Login Page (Click to view full-size image)

Upon successfully logging in, we are redirected to the `CreatingUserAccounts.aspx` page. This time the `UrlAuthorizationModule` permits access to the page because we are no longer anonymous.

Applying URL Authorization Rules to a Specific Location

The authorization settings defined in the `<system.web>` section of `Web.config` apply to all of the ASP.NET resources in that directory and its subdirectories (until otherwise overridden by another `Web.config` file). In some cases, though, we may want all ASP.NET resources in a given directory to have a particular authorization configuration except for one or two specific pages. This can be achieved by adding a `<location>` element in `Web.config`, pointing it to the file whose authorization rules differ, and defining its unique authorization rules therein.

To illustrate using the `<location>` element to override the configuration settings for a specific resource, let's customize the authorization settings so that only Tito can visit `CreatingUserAccounts.aspx`. To accomplish this, add a `<location>` element to the `Membership` folder's `Web.config` file and update its markup so that it looks like the following:

```
<?xml version="1.0"?>
<configuration>
 <system.web>
 <authorization>
 <deny users="?" />
 </authorization>
 </system.web>

 <location path="CreatingUserAccounts.aspx">
 <system.web>
 <authorization>
 <allow users="Tito" />
 <deny users="*" />
 </authorization>
 </system.web>
 </location>
</configuration>
```

The `<authorization>` element in `<system.web>` defines the default URL authorization rules for ASP.NET resources in the `Membership` folder and its subfolders. The `<location>` element allows us to override these rules for a particular resource. In the above markup the `<location>` element references the `CreatingUserAccounts.aspx` page and specifies its authorization rules such as to allow Tito, but deny everyone else.

To test out this authorization change, start by visiting the website as an anonymous user. If you attempt to visit any page in the `Membership` folder, such as `UserBasedAuthorization.aspx`, the `UrlAuthorizationModule` will deny the request and you will be redirected to the login page. After logging in as, say, Scott, you can visit any page in the `Membership` folder *except* for `CreatingUserAccounts.aspx`. Attempting to visit `CreatingUserAccounts.aspx` logged on as anyone but Tito will result in an unauthorized access attempt, redirecting you back to the login page.

> **NOTE**
> The `<location>` element must appear outside of the configuration's `<system.web>` element. You need to use a separate `<location>` element for each resource whose authorization settings you want to override.

A Look at How the `UrlAuthorizationModule` Uses the Authorization Rules to Grant or Deny Access

The `UrlAuthorizationModule` determines whether to authorize a particular identity for a particular URL by analyzing the URL authorization rules one at a time, starting from the first one and working its way down. As soon as a match is found, the user is granted or denied access, depending on if the match was found in an `<allow>` or `<deny>` element. **If no match is found, the user is granted access.** Consequently, if you want to restrict access, it is imperative that you use a `<deny>` element as the last element in the URL authorization configuration. **If you omit a `<deny>` element, all users will be granted access.**

To better understand the process used by the `UrlAuthorizationModule` to determine authority, consider the example URL authorization rules we looked at earlier in this step. The first rule is an `<allow>` element that allows access to Tito and Scott. The second rules is a `<deny>` element that denies access to everyone. If an anonymous user visits, the `UrlAuthorizationModule` starts by asking, Is anonymous either Scott or Tito? The answer, obviously, is No, so it proceeds to the second rule. Is anonymous in the set of everybody? Since the answer here is Yes, the `<deny>` rule is put in effect and the visitor is redirected to the login page. Similarly, if Jisun is visiting, the `UrlAuthorizationModule` starts by asking, Is Jisun either Scott or Tito? Since she is not, the

`UrlAuthorizationModule` proceeds to the second question, Is Jisun in the set of everybody? She is, so she, too, is denied access. Finally, if Tito visits, the first question posed by the `UrlAuthorizationModule` is an affirmative answer, so Tito is granted access.

Since the `UrlAuthorizationModule` processes the authorization rules from the top down, stopping at any match, it is important to have the more specific rules come before the less specific ones. That is, to define authorization rules that forbid Jisun and anonymous users, but allow all other authenticated users, you would start with the most specific rule - the one impacting Jisun - and then proceed to the less specific rules - those allowing all other authenticated users, but denying all anonymous users. The following URL authorization rules implements this policy by first denying Jisun, and then denying any anonymous user. Any authenticated user other than Jisun will be granted access because neither of these `<deny>` statements will match.

```
<authorization>
 <deny users="Jisun" />
 <deny users="?" />
</authorization>
```

Step 2: Fixing the Workflow for Unauthorized, Authenticated Users

As we discussed earlier in this tutorial in the A Look at the URL Authorization Workflow section, anytime an unauthorized request transpires, the `UrlAuthorizationModule` aborts the request and returns an HTTP 401 Unauthorized status. This 401 status is modified by the `FormsAuthenticationModule` into a 302 Redirect status that sends the user to the login page. This workflow occurs on any unauthorized request, even if the user is authenticated.

Returning an authenticated user to the login page is likely to confuse them since they have already logged into the system. With a little bit of work we can improve this workflow by redirecting authenticated users who make unauthorized requests to a page that explains that they have attempted to access a restricted page.

Start by creating a new ASP.NET page in the web application's root folder named `UnauthorizedAccess.aspx`; don't forget to associate this page with the `Site.master` master page. After creating this page, remove the Content control that references the `LoginContent` ContentPlaceHolder so that the master page's default content will be displayed. Next, add a message that explains the situation, namely that the user attempted to access a protected resource. After adding such a message, the `UnauthorizedAccess.aspx` page's declarative markup should look similar to the following:

```
<%@ Page Language="VB" MasterPageFile="~/Site.master" AutoEventWireup="false"
CodeFile="UnauthorizedAccess.aspx.cs" Inherits="UnauthorizedAccess"
Title="Untitled Page" %>

<asp:Content ID="Content1" ContentPlaceHolderID="MainContent"
Runat="Server">
 <h2>Unauthorized Access</h2>
 <p>
 You have attempted to access a page that you are not authorized to view.
 </p>
 <p>
 If you have any questions, please contact the site administrator.
 </p>
</asp:Content>
```

We now need to alter the workflow so that if an unauthorized request is performed by an authenticated user they are sent to the `UnauthorizedAccess.aspx` page instead of the login page. The logic that redirects unauthorized requests to the login page is buried within a private method of the `FormsAuthenticationModule` class, so we cannot customize this behavior. What we can do, however, is add our own logic to the login page

that redirects the user to `UnauthorizedAccess.aspx`, if needed.

When the `FormsAuthenticationModule` redirects an unauthorized visitor to the login page it appends the requested, unauthorized URL to the querystring with the name `ReturnUrl`. For example, if an unauthorized user attempted to visit `OnlyTito.aspx`, the `FormsAuthenticationModule` would redirect them to `Login.aspx?ReturnUrl=OnlyTito.aspx`. Therefore, if the login page is reached by an authenticated user with a querystring that includes the `ReturnUrl` parameter, then we know that this unauthenticated user just attempted to visit a page she is not authorized to view. In such a case, we want to redirect her to `UnauthorizedAccess.aspx`.

To accomplish this, add the following code to the login page's `Page_Load` event handler:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
 If Not Page.IsPostBack Then
 If Request.IsAuthenticated AndAlso Not String.IsNullOrEmpty(Request.QueryString("ReturnUrl")) Then
 ' This is an unauthorized, authenticated request...
 Response.Redirect("~/UnauthorizedAccess.aspx")
 End If
 End If
End Sub
```

The above code redirects authenticated, unauthorized users to the `UnauthorizedAccess.aspx` page. To see this logic in action, visit the site as an anonymous visitor and click on the Creating User Accounts link in the left column. This will take you to the `~/Membership/CreatingUserAccounts.aspx` page, which in Step 1 we configured to only permit access to Tito. Since anonymous users are prohibited, the `FormsAuthenticationModule` redirects us back to the login page.

At this point we are anonymous, so `Request.IsAuthenticated` returns `False` and we are not redirected to `UnauthorizedAccess.aspx`. Instead, the login page is displayed. Log in as a user other than Tito, such as Bruce. After entering the appropriate credentials, the login page redirects us back to `~/Membership/CreatingUserAccounts.aspx`. However, since this page is only accessible to Tito, we are unauthorized to view it and are promptly returned to the login page. This time, however, `Request.IsAuthenticated` returns `True` (and the `ReturnUrl` querystring parameter exists), so we are redirected to the `UnauthorizedAccess.aspx` page.

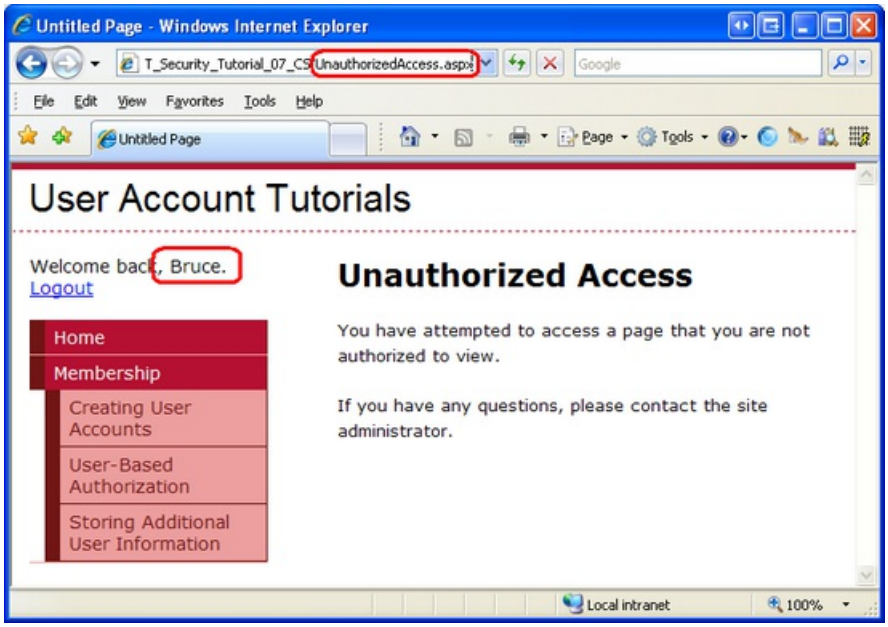


Figure 6: Authenticated, Unauthorized Users are Redirected to `UnauthorizedAccess.aspx` (Click to view full-size image)

This customized workflow presents a more sensible and straightforward user experience by short circuiting the cycle depicted in Figure 2.

Step 3: Limiting Functionality Based on the Currently Logged In User

URL authorization makes it easy to specify coarse authorization rules. As we saw in Step 1, with URL authorization we can succinctly state what identities are permitted and which ones are denied from viewing a particular page or all pages in a folder. In certain scenarios, however, we may want to allow all users to visit a page, but limit the page's functionality based on the user visiting it.

Consider the case of an eCommerce website that allows authenticated visitors to review their products. When an anonymous user visits a product's page, they would see just the product information and would not be given the opportunity to leave a review. However, an authenticated user visiting the same page would see the reviewing interface. If the authenticated user had not yet reviewed this product, the interface would enable them to submit a review; otherwise it would show them their previously-submitted review. To take this scenario a step further, the product page might show additional information and offer extended features for those users that work for the eCommerce company. For example, the product page might list the inventory in stock and include options to edit the product's price and description when visited by an employee.

Such fine grain authorization rules can be implemented either declaratively or programmatically (or through some combination of the two). In the next section we will see how to implement fine grain authorization via the LoginView control. Following that, we will explore programmatic techniques. Before we can look at applying fine grain authorization rules, however, we first need to create a page whose functionality depends on the user visiting it.

Let's create a page that lists the files in a particular directory within a GridView. Along with listing each file's name, size, and other information, the GridView will include two columns of LinkButtons: one titled View and one titled Delete. If the View LinkButton is clicked, the contents of the selected file will be displayed; if the Delete LinkButton is clicked, the file will be deleted. Let's initially create this page such that its view and delete functionality is available to all users. In the Using the LoginView Control and Programmatically Limiting Functionality sections we will see how to enable or disable these features based on the user visiting the page.

> **NOTE**
>
> The ASP.NET page we are about to build uses a GridView control to display a list of files. Since this tutorial series focuses on forms authentication, authorization, user accounts, and roles, I do not want to spend too much time discussing the inner workings of the GridView control. While this tutorial provides specific step-by-step instructions for setting up this page, it does not delve into the details of why certain choices were made, or what effect particular properties have on the rendered output. For a thorough examination of the GridView control, consult my *Working with Data in ASP.NET 2.0* tutorial series.

Start by opening the `UserBasedAuthorization.aspx` file in the `Membership` folder and adding a GridView control to the page named `FilesGrid`. From the GridView's Smart Tag, click the Edit Columns link to launch the Fields dialog box. From here, uncheck the Auto-generate fields checkbox in the lower left corner. Next, add a Select button, a Delete button, and two BoundFields from the upper left corner (the Select and Delete buttons can be found under the CommandField type). Set the Select button's `SelectText` property to View and the first BoundField's `HeaderText` and `DataField` properties to Name. Set the second BoundField's `HeaderText` property to Size in Bytes, its `DataField` property to Length, its `DataFormatString` property to {0:N0} and its `HtmlEncode` property to False.

After configuring the GridView's columns, click OK to close the Fields dialog box. From the Properties window, set the GridView's `DataKeyNames` property to `FullName`. At this point the GridView's declarative markup should look like the following:

```
<asp:GridView ID="FilesGrid" DataKeyNames="FullName" runat="server" AutoGenerateColumns="False">
 <Columns>
 <asp:CommandField SelectText="View" ShowSelectButton="True"/>
 <asp:CommandField ShowDeleteButton="True" />
 <asp:BoundField DataField="Name" HeaderText="Name" />
 <asp:BoundField DataField="Length" DataFormatString="{0:N0}"
 HeaderText="Size in Bytes" HtmlEncode="False" />
 </Columns>
</asp:GridView>
```

With the GridView's markup created, we're ready to write the code that will retrieve the files in a particular directory and bind them to the GridView. Add the following code to the page's `Page_Load` event handler:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
 If Not Page.IsPostBack Then
 Dim appPath As String = Request.PhysicalApplicationPath
 Dim dirInfo As New DirectoryInfo(appPath)

 Dim files() As FileInfo = dirInfo.GetFiles()

 FilesGrid.DataSource = files
 FilesGrid.DataBind()
 End If
End Sub
```

The above code uses the `DirectoryInfo` class to obtain a list of the files in the application's root folder. The `GetFiles()` method returns all of the files in the directory as an array of `FileInfo` objects, which is then bound to the GridView. The `FileInfo` object has an assortment of properties, such as `Name`, `Length`, and `IsReadOnly`, among others. As you can see from its declarative markup, the GridView displays just the `Name` and `Length` properties.

> **NOTE**
> The `DirectoryInfo` and `FileInfo` classes are found in the `System.IO` namespace. Therefore, you will either need to preface these class names with their namespace names or have the namespace imported into the class file (via `Imports System.IO`).

Take a moment to visit this page through a browser. It will display the list of files residing in the application's root directory. Clicking any of the View or Delete LinkButtons will cause a postback, but no action will occur because we've yet to create the necessary event handlers.

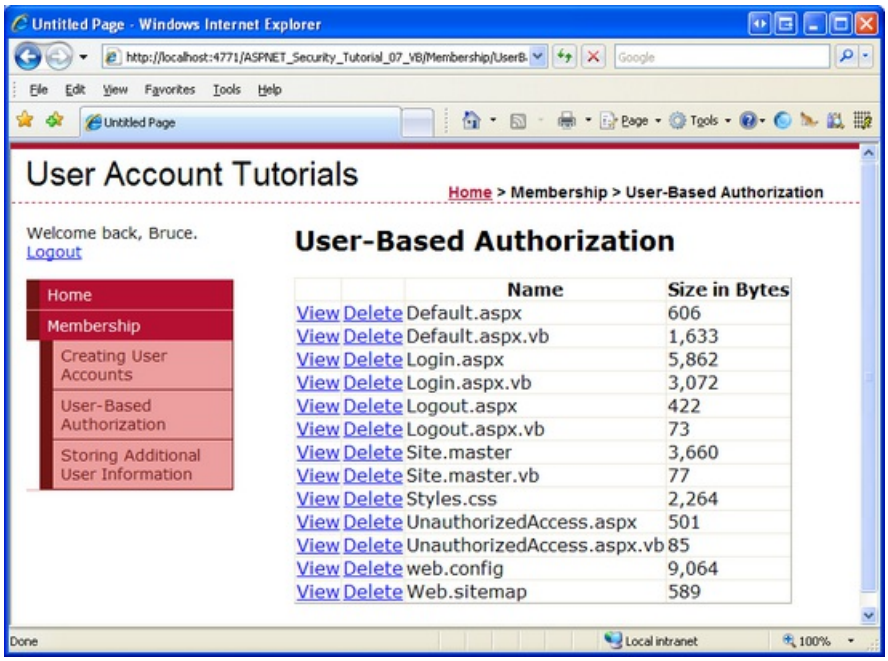


Figure 7: The GridView Lists the Files in the Web Application's Root Directory (Click to view full-size image)

We need a means to display the contents of the selected file. Return to Visual Studio and add a TextBox named `FileContents` above the GridView. Set its `TextMode` property to `MultiLine` and its `Columns` and `Rows` properties to 95% and 10, respectively.

```
<asp:TextBox ID="FileContents" runat="server" Rows="10"
TextMode="MultiLine" Width="95%"></asp:TextBox>
```

Next, create an event handler for the GridView's `SelectedIndexChanged` event and add the following code:

```
Protected Sub FilesGrid_SelectedIndexChanged(ByVal sender As Object, ByVal e As System.EventArgs) Handles
FilesGrid.SelectedIndexChanged
 ' Open the file and display it
 Dim fullFileName As String = FilesGrid.SelectedValue.ToString()
 Dim contents As String = File.ReadAllText(fullFileName)
 FileContents.Text = contents
End Sub
```

This code uses the GridView's `SelectedValue` property to determine the full file name of the selected file. Internally, the `DataKeys` collection is referenced in order to obtain the `SelectedValue`, so it is imperative that you set the GridView's `DataKeyNames` property to Name, as described earlier in this step. The `File` class is used to read the selected file's contents into a string, which is then assigned to the `FileContents` TextBox's `Text` property, thereby displaying the contents of the selected file on the page.

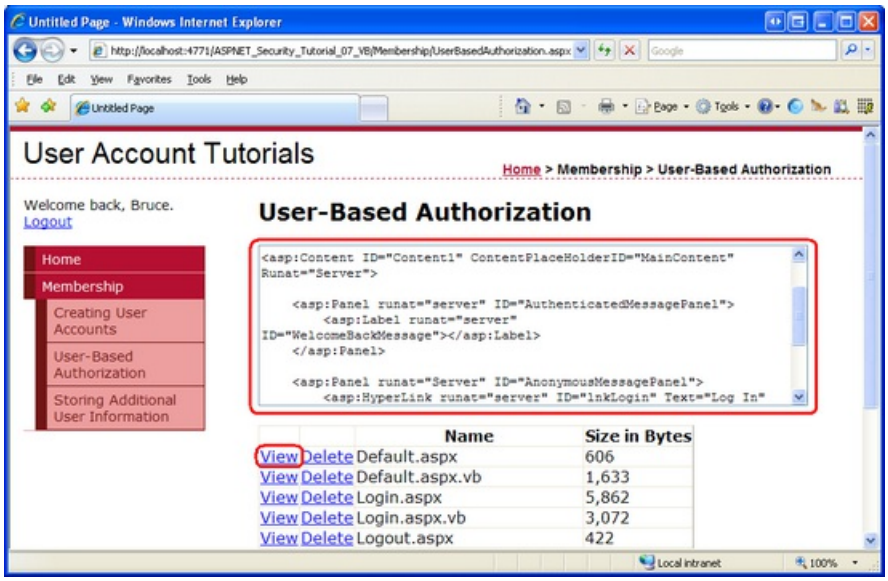


Figure 8: The Selected File's Contents are Displayed in the TextBox (Click to view full-size image)

> **NOTE**
>
> If you view the contents of a file that contains HTML markup, and then attempt to view or delete a file, you will receive an `HttpRequestValidationException` error. This occurs because on postback the TextBox's contents are sent back to the web server. By default, ASP.NET raises an `HttpRequestValidationException` error whenever potentially dangerous postback content, such as HTML markup, is detected. To disable this error from occurring, turn off request validation for the page by adding `ValidateRequest="false"` to the `@Page` directive. For more information on the benefits of request validation as well as what precautions you should take when disabling it, read Request Validation - Preventing Script Attacks.

Finally, add an event handler with the following code for the GridView's `RowDeleting` event:

```
Protected Sub FilesGrid_RowDeleting(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.GridViewDeleteEventArgs)Handles FilesGrid.RowDeleting
 Dim fullFileName As String = FilesGrid.DataKeys(e.RowIndex).Value.ToString()
 FileContents.Text = String.Format("You have opted to delete {0}.", fullFileName)

 ' To actually delete the file, uncomment the following line
 ' File.Delete(fullFileName)
End Sub
```

The code simply displays the full name of the file to delete in the `FileContents` TextBox *without* actually deleting the file.

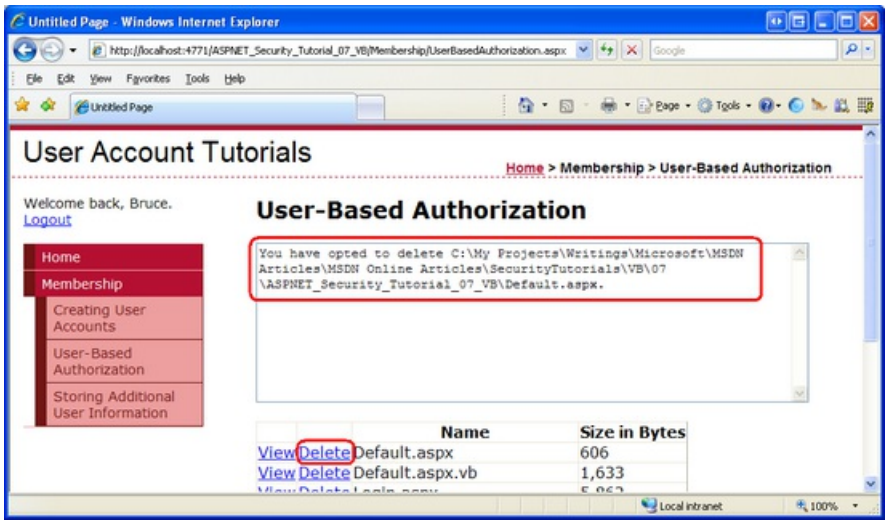


Figure 9: Clicking the Delete Button Does Not Actually Delete the File (Click to view full-size image)

In Step 1 we configured the URL authorization rules to prohibit anonymous users from viewing the pages in the `Membership` folder. In order to better exhibit fine grain authentication, let's allow anonymous users to visit the `UserBasedAuthorization.aspx` page, but with limited functionality. To open this page up to be accessed by all users, add the following `<location>` element to the `Web.config` file in the `Membership` folder:

```
<location path="UserBasedAuthorization.aspx">
 <system.web>
 <authorization>
 <allow users="*" />
 </authorization>
 </system.web>
</location>
```

After adding this `<location>` element, test the new URL authorization rules by logging out of the site. As an anonymous user you should be permitted to visit the `UserBasedAuthorization.aspx` page.

Currently, any authenticated or anonymous user can visit the `UserBasedAuthorization.aspx` page and view or delete files. Let's make it so that only authenticated users can view contents of a file and only Tito can delete a file. Such fine grain authorization rules can be applied declaratively, programmatically, or through a combination of both methods. Let's use the declarative approach to limit who can view the contents of a file; we'll use the programmatic approach to limit who can delete a file.

Using the LoginView Control

As we've seen in past tutorials, the LoginView control is useful for displaying different interfaces for authenticated and anonymous users, and offers an easy way to hide functionality that is not accessible to anonymous users. Since anonymous users cannot view or delete files, we only need to show the `FileContents` TextBox when the page is visited by an authenticated user. To achieve this, add a LoginView control to the page, name it `LoginViewForFileContentsTextBox`, and move the `FileContents` TextBox's declarative markup into the LoginView control's `LoggedInTemplate`.

```
<asp:LoginView ID=" LoginViewForFileContentsTextBox " runat="server">
 <LoggedInTemplate>
 <p>
 <asp:TextBox ID="FileContents" runat="server" Rows="10"
 TextMode="MultiLine" Width="95%"></asp:TextBox>
 </p>
 </LoggedInTemplate>
</asp:LoginView>
```

The Web controls in the LoginView's templates are no longer directly accessible from the code-behind class. For example, the `FilesGrid` GridView's `SelectedIndexChanged` and `RowDeleting` event handlers currently reference the `FileContents` TextBox control with code like:

```
FileContents.Text = text
```

However, this code is no longer valid. By moving the `FileContents` TextBox into the `LoggedInTemplate` the TextBox cannot be directly accessed. Instead, we must use the `FindControl("controlId")` method to programmatically reference the control. Update the `FilesGrid` event handlers to reference the TextBox like so:

```
Dim FileContentsTextBox As TextBox =
CType(LoginViewForFileContentsTextBox.FindControl("FileContents"),TextBox)
FileContentsTextBox.Text = text
```

After moving the TextBox to the LoginView's `LoggedInTemplate` and updating the page's code to reference the TextBox using the `FindControl("controlId")` pattern, visit the page as an anonymous user. As Figure 10 shows, the `FileContents` TextBox is not displayed. However, the View LinkButton is still displayed.

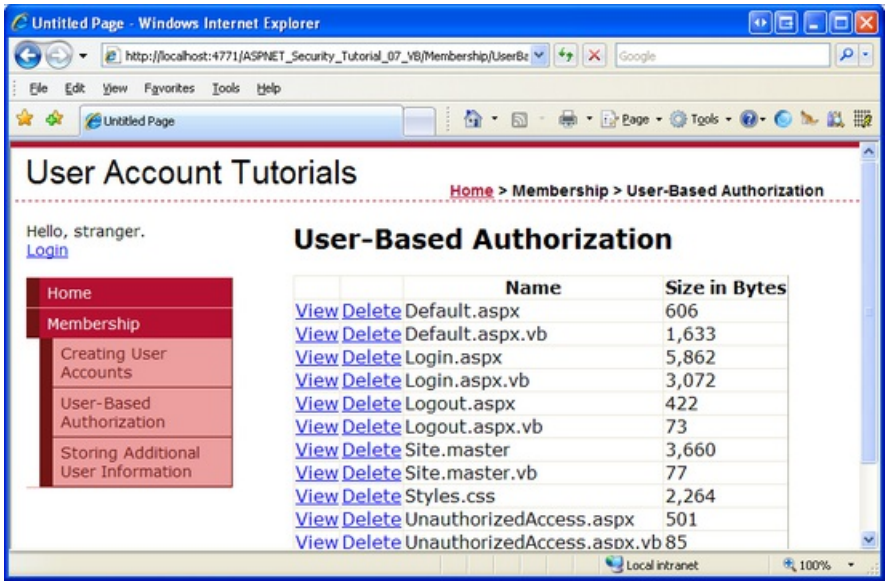


Figure 10: The LoginView Control Only Renders the `FileContents` TextBox for Authenticated Users (Click to view full-size image)

One way to hide the View button for anonymous users is to convert the GridView field into a TemplateField. This will generate a template that contains the declarative markup for the View LinkButton. We can then add a LoginView control to the TemplateField and place the LinkButton within the LoginView's `LoggedInTemplate`, thereby hiding the View button from anonymous visitors. To accomplish this, click on the Edit Columns link from the GridView's Smart Tag to launch the Fields dialog box. Next, select the Select button from the list in the lower left corner and then click the Convert this field to a TemplateField link. Doing so will modify the field's declarative markup from:

```
<asp:CommandField SelectText="View" ShowSelectButton="True"/>
```

To:

```
<asp:TemplateField ShowHeader="False">
 <ItemTemplate>
 <asp:LinkButton ID="LinkButton1" runat="server" CausesValidation="False"
 CommandName="Select" Text="View"></asp:LinkButton>
 </ItemTemplate>
</asp:TemplateField>
```

At this point, we can add a LoginView to the TemplateField. The following markup displays the View LinkButton only for authenticated users.

```
<asp:TemplateField ShowHeader="False">
 <ItemTemplate>
 <asp:LoginView ID="LoginView1" runat="server">
 <LoggedInTemplate>
 <asp:LinkButton ID="LinkButton1" runat="server" CausesValidation="False"
 CommandName="Select" Text="View"></asp:LinkButton>
 </LoggedInTemplate>
 </asp:LoginView>
 </ItemTemplate>
</asp:TemplateField>
```

As Figure 11 shows, the end result is not that pretty as the View column is still displayed even though the View LinkButtons within the column are hidden. We will look at how to hide the entire GridView column (and not just the LinkButton) in the next section.

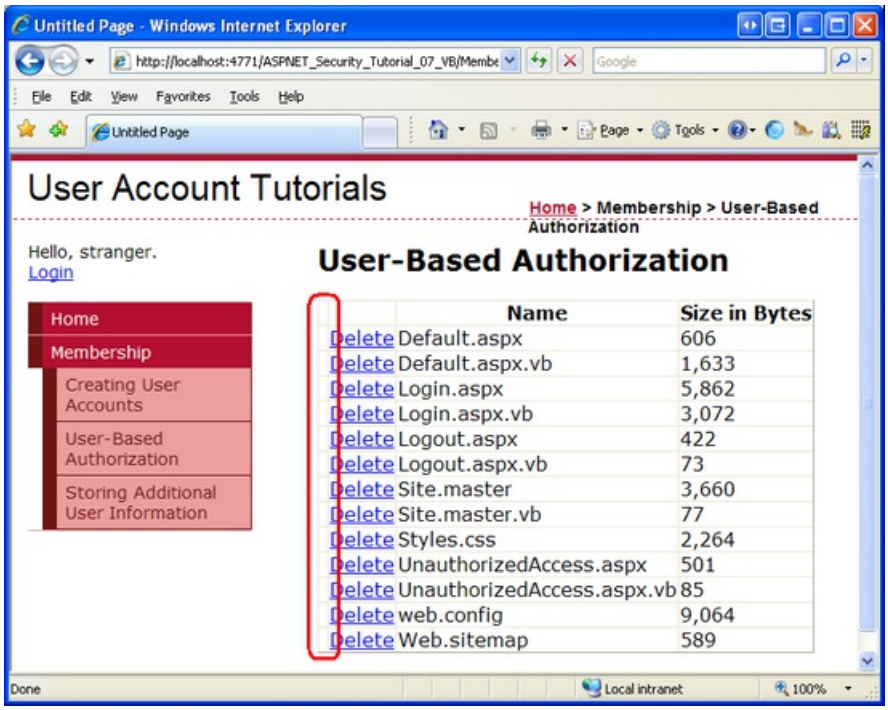


Figure 11: The LoginView Control Hides the View LinkButtons for Anonymous Visitors (Click to view full-size image)

Programmatically Limiting Functionality

In some circumstances, declarative techniques are insufficient for limiting functionality to a page. For example, the availability of certain page functionality may be dependent on criteria beyond whether the user visiting the page is anonymous or authenticated. In such cases, the various user interface elements can be displayed or hidden through programmatic means.

In order to limit the functionality programmatically, we need to perform two tasks:

1. Determine whether the user visiting the page can access the functionality, and
2. Programmatically modify the user interface based on whether the user has access to the functionality in question.

To demonstrate the application of these two tasks, let's only allow Tito to delete files from the GridView. Our first task, then, is to determine whether it is Tito visiting the page. Once that has been determined, we need to hide (or show) the GridView's Delete column. The GridView's columns are accessible through its `Columns` property; a column is only rendered if its `Visible` property is set to `True` (the default).

Add the following code to the `Page_Load` event handler prior to binding the data to the GridView:

```
' Is this Tito visiting the page?
Dim userName As String = User.Identity.Name
If String.Compare(userName, "Tito", True) = 0 Then
 ' This is Tito, SHOW the Delete column
 FilesGrid.Columns(1).Visible = True
Else
 ' This is NOT Tito, HIDE the Delete column
 FilesGrid.Columns(1).Visible = False
End If
```

As we discussed in the *An Overview of Forms Authentication* tutorial, `User.Identity.Name` returns the identity's

name. This corresponds to the username entered in the Login control. If it is Tito visiting the page, the GridView's second column's `Visible` property is set to `True`; otherwise, it is set to `False`. The net result is that when someone other than Tito visits the page, either another authenticated user or an anonymous user, the Delete column is not rendered (see Figure 12); however, when Tito visits the page, the Delete column is present (see Figure 13).

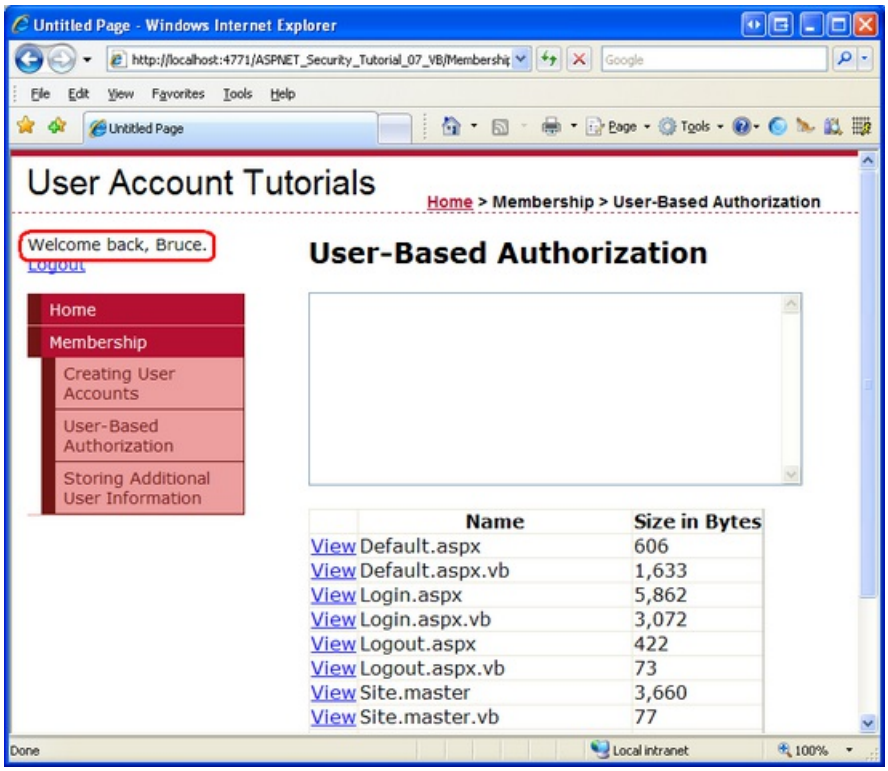


Figure 12: The Delete Column is Not Rendered When Visited By Someone Other Than Tito (Such as Bruce) (Click to view full-size image)

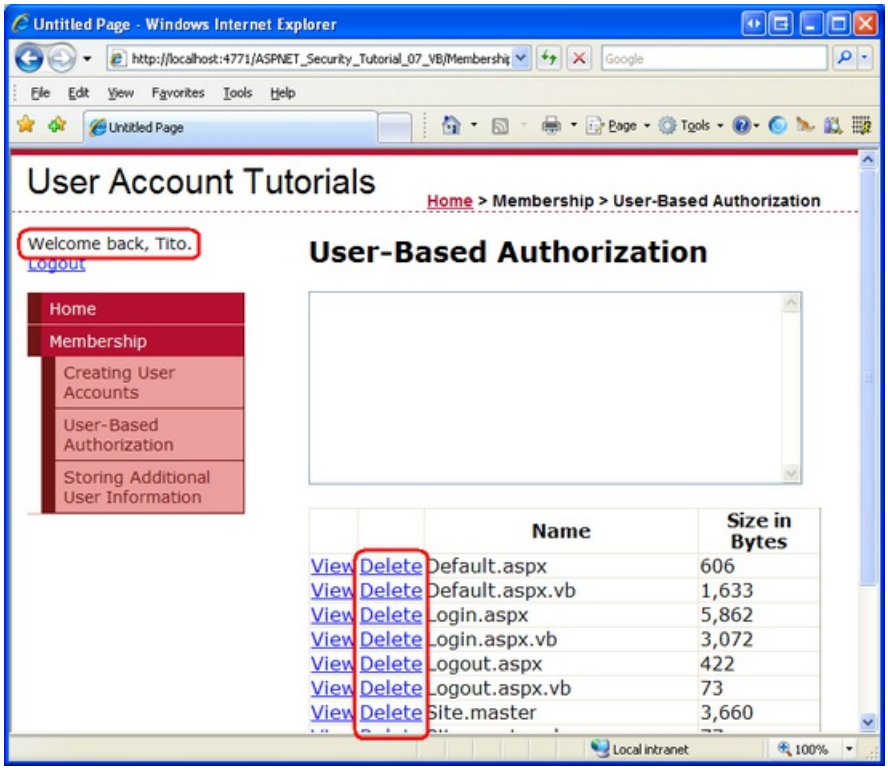


Figure 13: The Delete Column is Rendered for Tito (Click to view full-size image)

Step 4: Applying Authorization Rules to Classes and Methods

In Step 3 we disallowed anonymous users from viewing a file's contents and prohibited all users but Tito from deleting files. This was accomplished by hiding the associated user interface elements for unauthorized visitors through declarative and programmatic techniques. For our simple example, properly hiding the user interface elements was straightforward, but what about more complex sites where there may be many different ways to perform the same functionality? In limiting that functionality to unauthorized users, what happens if we forget to hide or disable all of the applicable user interface elements?

An easy way to ensure that a particular piece of functionality cannot be accessed by an unauthorized user is to decorate that class or method with the `PrincipalPermission` attribute. When the .NET runtime uses a class or executes one of its methods, it checks to ensure that the current security context has permission to use the class or execute the method. The `PrincipalPermission` attribute provides a mechanism through which we can define these rules.

Let's demonstrate using the `PrincipalPermission` attribute on the GridView's `SelectedIndexChanged` and `RowDeleting` event handlers to prohibit execution by anonymous users and users other than Tito, respectively. All we need to do is add the appropriate attribute atop each function definition:

```
<PrincipalPermission(SecurityAction.Demand, Authenticated:=True)> _
Protected Sub FilesGrid_SelectedIndexChanged(ByVal sender As Object, ByVal e As System.EventArgs) Handles
FilesGrid.SelectedIndexChanged
 ...
End Sub

<PrincipalPermission(SecurityAction.Demand, Name:="Tito")> _
Protected Sub FilesGrid_RowDeleting(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.GridViewDeleteEventArgs) Handles FilesGrid.RowDeleting
 ...
End Sub
```

The attribute for the `SelectedIndexChanged` event handler dictates that only authenticated users can execute the event handler, where as the attribute on the `RowDeleting` event handler limits the execution to Tito.

> **NOTE**
>
> An attribute can be applied to a class, method, property, or event. When adding an attribute, it must be part of the class, method, property, or event declaration statement. Since Visual Basic uses line breaks as statement delimiters, attributes must either appear on the same line as the declaration or directly above it with a line continuation character (the underscore). In the code snippet above, the line continuation character is used to place the attribute on one line and the method declaration on another.

If, somehow, a user other than Tito attempts to execute the `RowDeleting` event handler or a non-authenticated user attempts to execute the `SelectedIndexChanged` event handler, the .NET runtime will raise a `SecurityException`.

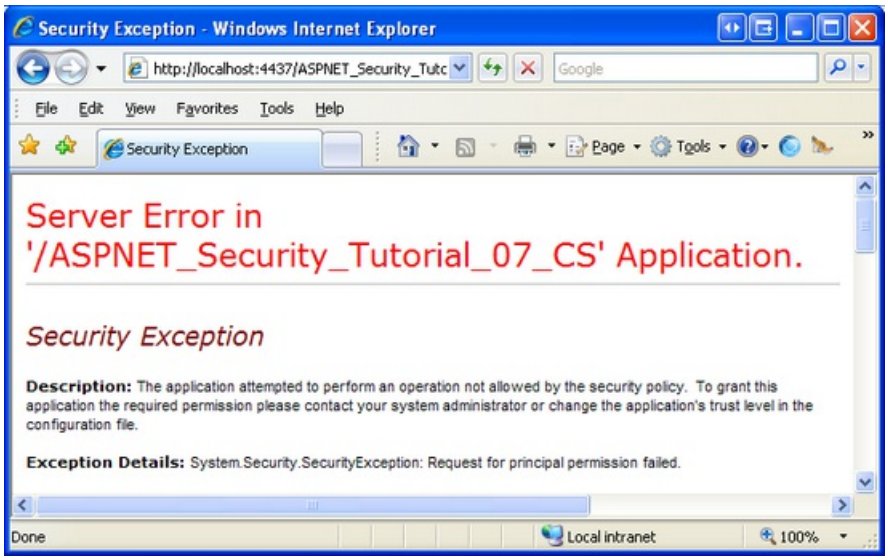


Figure 14: If the Security Context is not Authorized to Execute the Method, a `SecurityException` is Thrown (Click to view full-size image)

> **NOTE**
>
> To allow multiple security contexts to access a class or method, decorate the class or method with a `PrincipalPermission` attribute for each security context. That is, to allow both Tito and Bruce to execute the `RowDeleting` event handler, add *two* `PrincipalPermission` attributes:

```
<PrincipalPermission(SecurityAction.Demand, Name:="Tito")> _

<PrincipalPermission(SecurityAction.Demand, Name:="Bruce")> _
```

In addition to ASP.NET pages, many applications also have an architecture that includes various layers, such as Business Logic and Data Access Layers. These layers are typically implemented as Class Libraries and offer classes and methods for performing business logic- and data-related functionality. The `PrincipalPermission` attribute is useful for applying authorization rules to these layers.

For more information on using the `PrincipalPermission` attribute to define authorization rules on classes and methods, refer to Scott Guthrie's blog entry Adding Authorization Rules to Business and Data Layers Using `PrincipalPermissionAttributes`.

Summary

In this tutorial we looked at how to apply user-based authorization rules. We started with a look at ASP.NET's URL authorization framework. On each request, the ASP.NET engine's `UrlAuthorizationModule` inspects the URL authorization rules defined in the application's configuration to determine whether the identity is authorized to access the requested resource. In short, URL authorization makes it easy to specify authorization rules for a specific page or for all pages in a particular directory.

The URL authorization framework applies authorization rules on a page-by-page basis. With URL authorization, either the requesting identity is authorized to access a particular resource or not. Many scenarios, however, call for more fine grain authorization rules. Rather than defining who is permitted to access a page, we may need to let everyone access a page, but to show different data or offer different functionality depending on the user visiting the page. Page-level authorization usually involves hiding specific user interface elements in order to prevent unauthorized users from accessing prohibited functionality. Additionally, it is possible to use attributes to restrict access to classes and execution of its methods for certain users.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Adding Authorization Rules to Business and Data Layers Using `PrincipalPermissionAttributes`
- ASP.NET Authorization
- Changes Between IIS6 and IIS7 Security
- Configuring Specific Files and Subdirectories
- Limiting Data Modification Functionality Based on the User
- LoginView Control QuickStarts
- Understanding IIS7 URL Authorization
- `UrlAuthorizationModule` Technical Documentation
- Working with Data in ASP.NET 2.0

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS NEXT

Storing Additional User Information (VB)

2/20/2020 • 46 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> In this tutorial we will answer this question by building a very rudimentary guestbook application. In doing so, we will look at different options for modeling user information in a database, and then see how to associate this data with the user accounts created by the Membership framework.

Introduction

ASP.NET's Membership framework offers a flexible interface for managing users. The Membership API includes methods for validating credentials, retrieving information about the currently logged on user, creating a new user account, and deleting a user account, among others. Each user account in the Membership framework contains only the properties needed for validating credentials and performing essential user account-related tasks. This is evidenced by the methods and properties of the `MembershipUser` class, which models a user account in the Membership framework. This class has properties like `UserName`, `Email`, and `IsLockedOut`, and methods like `GetPassword` and `UnlockUser`.

Oftentimes, applications need to store additional user information not included in the Membership framework. For example, an online retailer might need to let each user store her shipping and billing addresses, payment information, delivery preferences, and contact phone number. Furthermore, each order in the system is associated with a particular user account.

The `MembershipUser` class does not include properties like `PhoneNumber` or `DeliveryPreferences` or `PastOrders`. So how do we track the user information needed by the application and have it integrate with the Membership framework? In this tutorial we will answer this question by building a very rudimentary guestbook application. In doing so, we will look at different options for modeling user information in a database, and then see how to associate this data with the user accounts created by the Membership framework. Let's get started!

Step 1: Creating the Guestbook Application's Data Model

There are a variety of techniques that can be employed to capture user information in a database and associate it with the user accounts created by the Membership framework. In order to illustrate these techniques, we will need to augment the tutorial web application so that it captures some sort of user-related data. (Currently, the application's data model contains only the application services tables needed by the `SqlMembershipProvider`.)

Let's create a very simple guestbook application where an authenticated user can leave a comment. In addition to storing guestbook comments, let's allow each user to store his home town, homepage, and signature. If provided, the user's home town, homepage, and signature will appear on each message he has left in the guestbook.

Adding the `GuestbookComments` Table

In order to capture the guestbook comments, we need to create a database table named `GuestbookComments` that has columns like `CommentId`, `Subject`, `Body`, and `CommentDate`. We also need to have each record in the `GuestbookComments` table reference the user who left the comment.

To add this table to our database, go to the Database Explorer in Visual Studio and drill down into the `SecurityTutorials` database. Right-click on the Tables folder and choose Add New Table. This brings up an interface that allows us to define the columns for the new table.

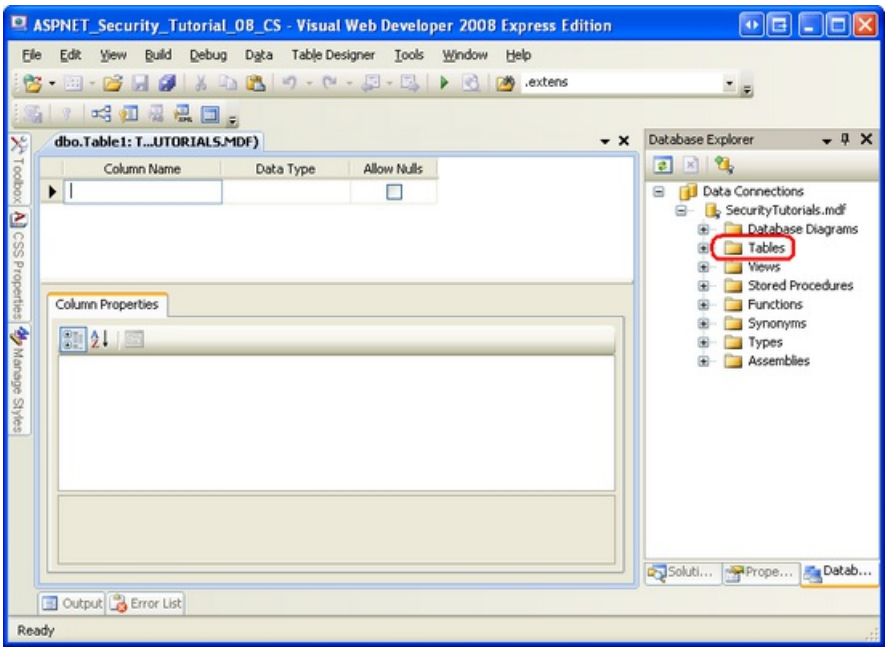


Figure 1: Add a New Table to the `SecurityTutorials` Database (Click to view full-size image)

Next, define the `GuestbookComments`'s columns. Start by adding a column named `CommentId` of type `uniqueidentifier`. This column will uniquely identify each comment in the guestbook, so disallow `NULL`s and mark it as the table's primary key. Rather than providing a value for the `CommentId` field on each `INSERT`, we can indicate that a new `uniqueidentifier` value should be automatically generated for this field on `INSERT` by setting the column's default value to `NEWID()`. After adding this first field, marking it as the primary key, and settings its default value, your screen should look similar to the screen shot shown in Figure 2.

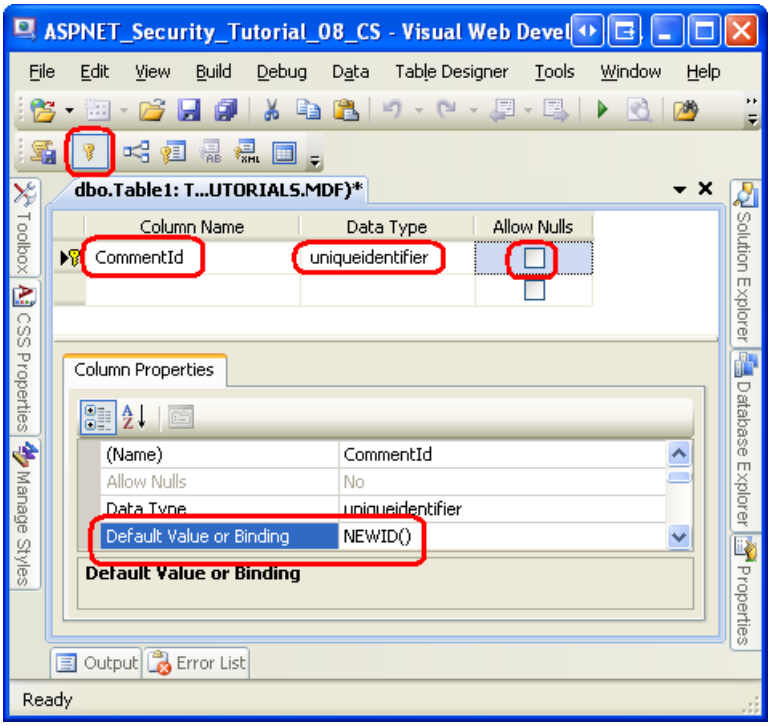


Figure 2: Add a Primary Column Named `CommentId` (Click to view full-size image)

Next, add a column named `Subject` of type `nvarchar(50)` and a column named `Body` of type `nvarchar(MAX)`, disallowing `NULL`s in both columns. Following that, add a column named `CommentDate` of type `datetime`. Disallow `NULL`s and set the `CommentDate` column's default value to `getdate()`.

All that remains is to add a column that associates a user account with each guestbook comment. One option would be to add a column named `UserName` of type `nvarchar(256)`. This is a suitable choice when using a Membership provider other than the `SqlMembershipProvider`. But when using the `SqlMembershipProvider`, as we

are in this tutorial series, the `UserName` column in the `aspnet_Users` table is not guaranteed to be unique. The `aspnet_Users` table's primary key is `UserId` and is of type `uniqueidentifier`. Therefore, the `GuestbookComments` table needs a column named `UserId` of type `uniqueidentifier` (disallowing `NULL` values). Go ahead and add this column.

> **NOTE**
>
> As we discussed in the *Creating the Membership Schema in SQL Server* tutorial, the Membership framework is designed to enable multiple web applications with different user accounts to share the same user store. It does this by partitioning user accounts into different applications. And while each username is guaranteed to be unique within an application, the same username may be used in different applications using the same user store. There is a composite `UNIQUE` constraint in the `aspnet_Users` table on the `UserName` and `ApplicationId` fields, but not one on just the `UserName` field. Consequently, it is possible for the aspnet_Users table to have two (or more) records with the same `UserName` value. There is, however, a `UNIQUE` constraint on the `aspnet_Users` table's `UserId` field (since it is the primary key). A `UNIQUE` constraint is important because without it we cannot establish a foreign key constraint between the `GuestbookComments` and `aspnet_Users` tables.

After adding the `UserId` column, save the table by clicking on the Save icon in the Toolbar. Name the new table `GuestbookComments`.

We have one last issue to attend to with the `GuestbookComments` table: we need to create a foreign key constraint between the `GuestbookComments.UserId` column and the `aspnet_Users.UserId` column. To achieve this, click the Relationship icon in the Toolbar to launch the Foreign Key Relationships dialog box. (Alternatively, you can launch this dialog box by going to the Table Designer menu and choosing Relationships.)

Click the Add button in the lower left corner of the Foreign Key Relationships dialog box. This will add a new foreign key constraint, although we still need to define the tables that participate in the relationship.

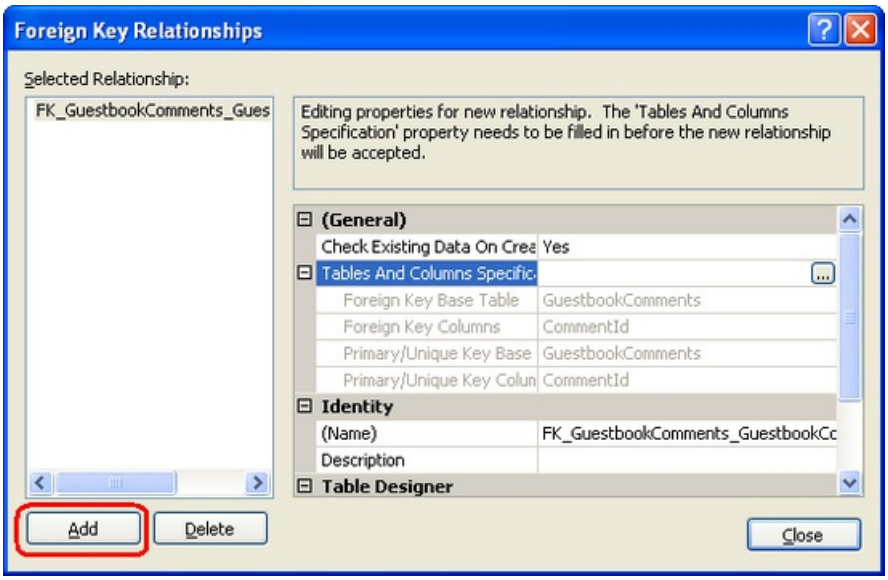


Figure 3: Use the Foreign Key Relationships Dialog Box to Manage a Table's Foreign Key Constraints (Click to view full-size image)

Next, click the ellipses icon in the "Table and Columns Specifications" row on the right. This will launch the Tables and Columns dialog box, from which we can specify the primary key table and column and the foreign key column from the `GuestbookComments` table. In particular, select `aspnet_Users` and `UserId` as the primary key table and column, and `UserId` from the `GuestbookComments` table as the foreign key column (see Figure 4). After defining the primary and foreign key tables and columns, click OK to return to the Foreign Key Relationships dialog box.

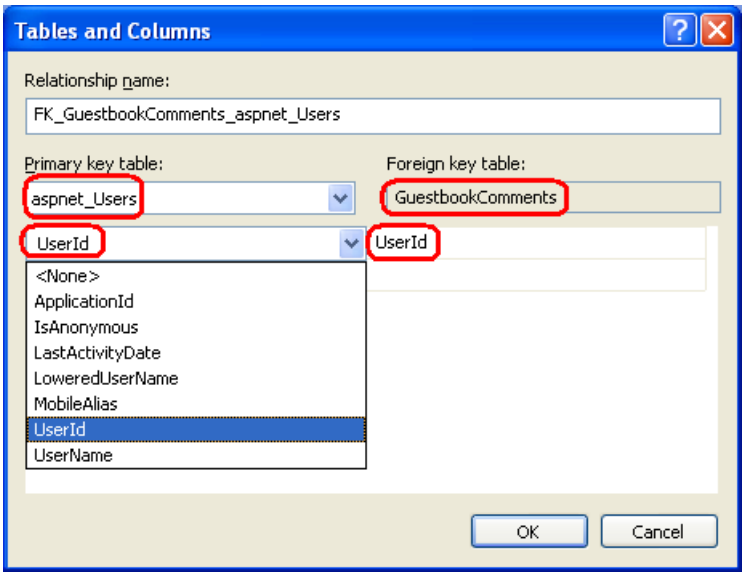


Figure 4: Establish a Foreign Key Constraint Between the `aspnet_Users` and `GuesbookComments` Tables (Click to view full-size image)

At this point the foreign key constraint has been established. The presence of this constraint ensures relational integrity between the two tables by guaranteeing that there will never be a guestbook entry referring to a non-existent user account. By default, a foreign key constraint will disallow a parent record to be deleted if there are corresponding child records. That is, if a user makes one or more guestbook comments, and then we attempt to delete that user account, the delete will fail unless his guestbook comments are deleted first.

Foreign key constraints can be configured to automatically delete the associated child records when a parent record is deleted. In other words, we can setup this foreign key constraint so that a user's guestbook entries are automatically deleted when her user account is deleted. To accomplish this, expand the "INSERT And UPDATE Specification" section and set the "Delete Rule" property to Cascade.

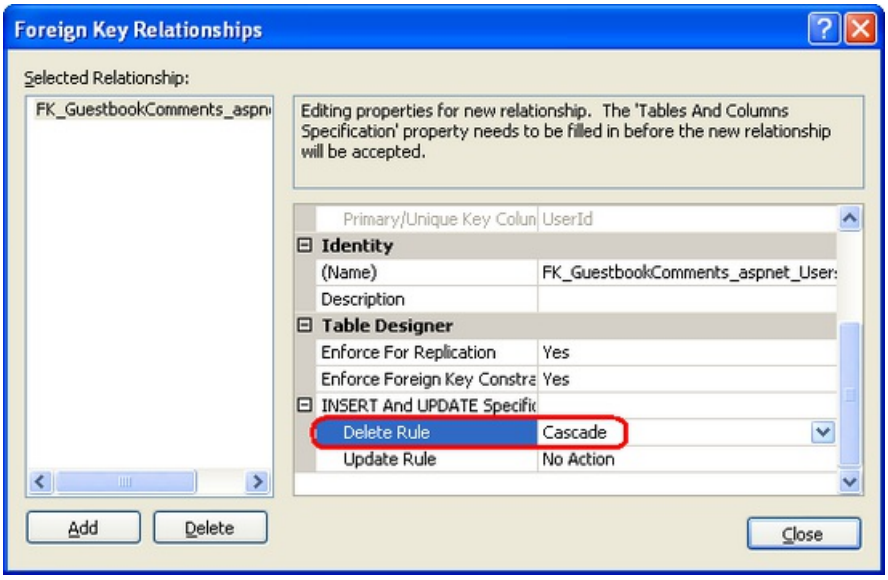


Figure 5: Configure the Foreign Key Constraint to Cascade Deletes (Click to view full-size image)

To save the foreign key constraint, click the Close button to exit out of the Foreign Key Relationships. Then click the Save icon in the Toolbar to save the table and the this relationship.

Storing the User's Home Town, Homepage, and Signature

The `GuestbookComments` table illustrates how to store information that shares a one-to-many relationship with user accounts. Since each user account may have an arbitrary number of associated comments, this relationship is modeled by creating a table to hold the set of comments that includes a column that links back each comment to a particular user. When using the `SqlMembershipProvider`, this link is best established by creating a column

named `UserId` of type `uniqueidentifier` and a foreign key constraint between this column and `aspnet_Users.UserId`.

We now need to associate three columns with each user account to store the user's home town, homepage, and signature, which will appear in his guestbook comments. There are number of different ways to accomplish this:

- **Add new columns to the `aspnet_Users` or `aspnet_Membership` tables.** I would not recommend this approach because it modifies the schema used by the `SqlMembershipProvider`. This decision may come back to haunt you down the road. For example, what if a future version of ASP.NET uses a different `SqlMembershipProvider` schema. Microsoft may include a tool to migrate the ASP.NET 2.0 `SqlMembershipProvider` data to the new schema, but if you have modified the ASP.NET 2.0 `SqlMembershipProvider` schema, such a conversion may not be possible.

- **Use ASP.NET's Profile framework, defining a profile property for the home town, homepage, and signature.** ASP.NET includes a Profile framework that is designed to store additional user-specific data. Like the Membership framework, the Profile framework is built atop the provider model. The .NET Framework ships with a `SqlProfileProvider` that stores profile data in a SQL Server database. In fact, our database already has the table used by the `SqlProfileProvider` (`aspnet_Profile`), as it was added when we added the application services back in the *Creating the Membership Schema in SQL Server* tutorial. The main benefit of the Profile framework is that it allows for developers to define the profile properties in `Web.config` – no code needs to be written to serialize the profile data to and from the underlying data store. In short, it is incredibly easy to define a set of profile properties and to work with them in code. However, the Profile system leaves a lot to be desired when it comes to versioning, so if you have an application where you expect new user-specific properties to be added at a later time, or existing ones to be removed or modified, then the Profile framework may not be the best option. Moreover, the `SqlProfileProvider` stores the profile properties in a highly denormalized fashion, making it next to impossible to run queries directly against the profile data (such as, how many users have a home town of New York).
  For more information on the Profile framework, consult the "Further Readings" section at the end of this tutorial.

- **Add these three columns to a new table in the database and establish a one-to-one relationship between this table and `aspnet_Users`.** This approach involves a bit more work than with the Profile framework, but offers maximum flexibility in how the additional user properties are modeled in the database. This is the option we will use in this tutorial.

We will create a new table called `UserProfiles` to save the home town, homepage, and signature for each user. Right-click on the Tables folder in the Database Explorer window and choose to create a new table. Name the first column `UserId` and set its type to `uniqueidentifier`. Disallow `NULL` values and mark the column as a primary key. Next, add columns named: `HomeTown` of type `nvarchar(50)`; `HomepageUrl` of type `nvarchar(100)`; and Signature of type `nvarchar(500)`. Each of these three columns can accept a `NULL` value.

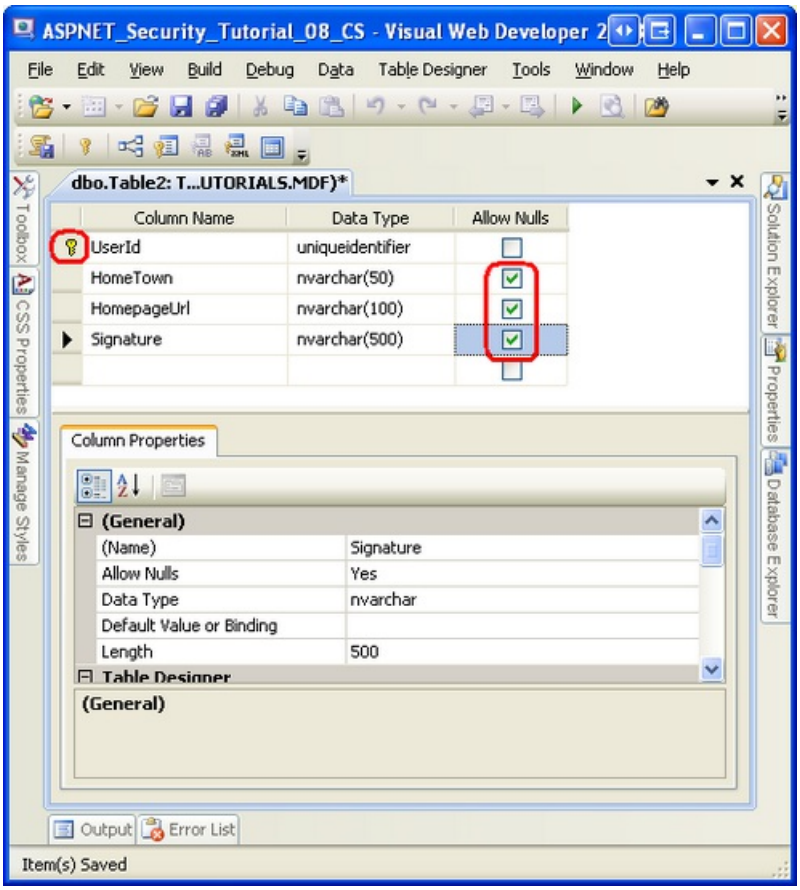


Figure 6: Create the `UserProfiles` Table (Click to view full-size image)

Save the table and name it `UserProfiles`. Lastly, establish a foreign key constraint between the `UserProfiles` table's `UserId` field and the `aspnet_Users.UserId` field. As we did with the foreign key constraint between the `GuestbookComments` and `aspnet_Users` tables, have this constraint cascade deletes. Since the `UserId` field in `UserProfiles` is the primary key, this ensures that there will be no more than one record in the `UserProfiles` table for each user account. This type of relationship is referred to as one-to-one.

Now that we have the data model created, we are ready to use it. In Steps 2 and 3 we will look at how the currently logged on user can view and edit their home town, homepage, and signature information. In Step 4 we will create the interface for authenticated users to submit new comments to the guestbook and view the existing ones.

Step 2: Displaying the User's Home Town, Homepage, and Signature

There are a variety of ways to allow the currently logged on user to view and edit his home town, homepage, and signature information. We could manually create the user interface with TextBox and Label controls or we could use one of the data Web controls, such as the DetailsView control. To perform the database `SELECT` and `UPDATE` statements we could write ADO.NET code in our page's code-behind class or, alternatively, employ a declarative approach with the SqlDataSource. Ideally our application would contain a tiered architecture, which we could either invoke programmatically from the page's code-behind class or declaratively via the ObjectDataSource control.

Since this tutorial series focuses on forms authentication, authorization, user accounts, and roles, there will not be a thorough discussion of these different data access options or why a tiered architecture is preferred over executing SQL statements directly from the ASP.NET page. I am going to walk through using a DetailsView and SqlDataSource – the quickest and easiest option – but the concepts discussed can certainly be applied to alternative Web controls and data access logic. For more information on working with data in ASP.NET, refer to my *Working with Data in ASP.NET 2.0* tutorial series.

Open the `AdditionalUserInfo.aspx` page in the `Membership` folder and add a DetailsView control to the page,

setting its ID property to `UserProfile` and clearing out its `Width` and `Height` properties. Expand the DetailsView's Smart Tag and choose to bind it to a new data source control. This will launch the DataSource Configuration Wizard (see Figure 7). The first step asks you to specify the data source type. Since we are going to connect directly to the `SecurityTutorials` database, choose the Database icon, specifying the `ID` as `UserProfileDataSource`.

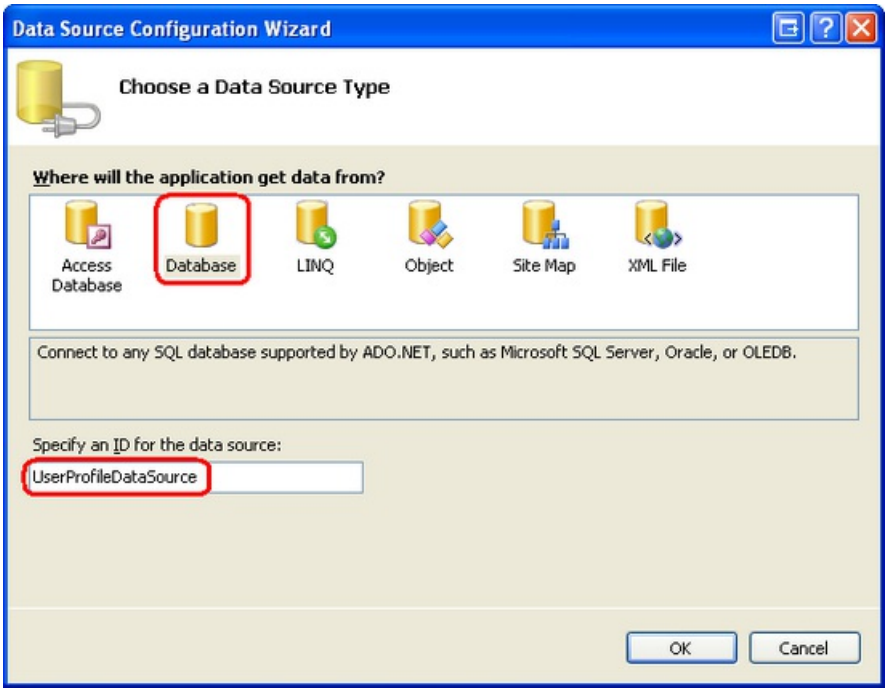


Figure 7: Add a New SqlDataSource Control Named `UserProfileDataSource` (Click to view full-size image)

The next screen prompts for the database to use. We have already defined a connection string in `Web.config` for the `SecurityTutorials` database. This connection string name – `SecurityTutorialsConnectionString` – should be in the drop-down list. Select this option and click Next.

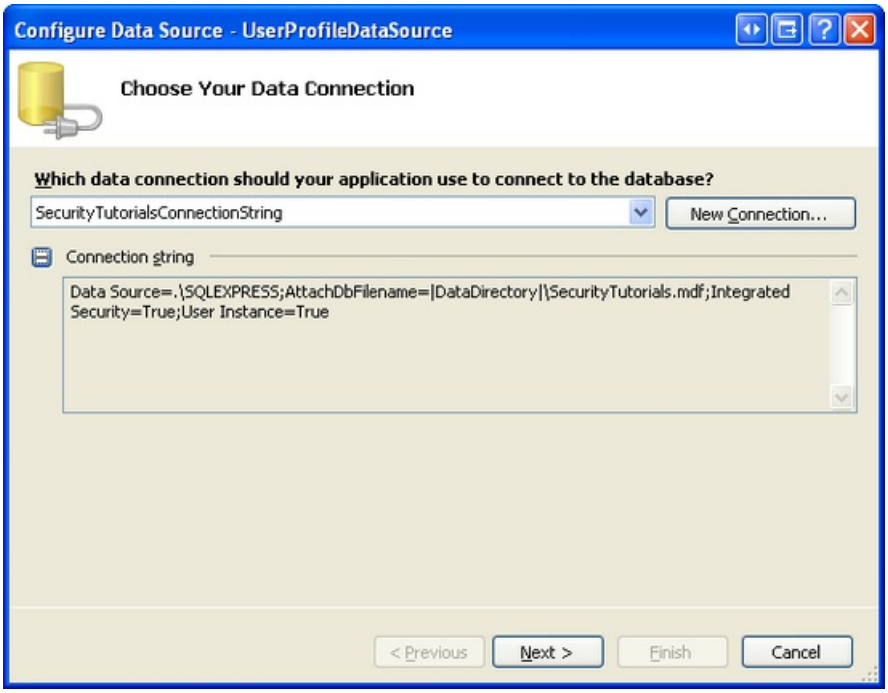


Figure 8: Choose `SecurityTutorialsConnectionString` from the Drop-Down List (Click to view full-size image)

The subsequent screen asks us to specify the table and columns to query. Choose the `UserProfiles` table from the drop-down list and check all of the columns.

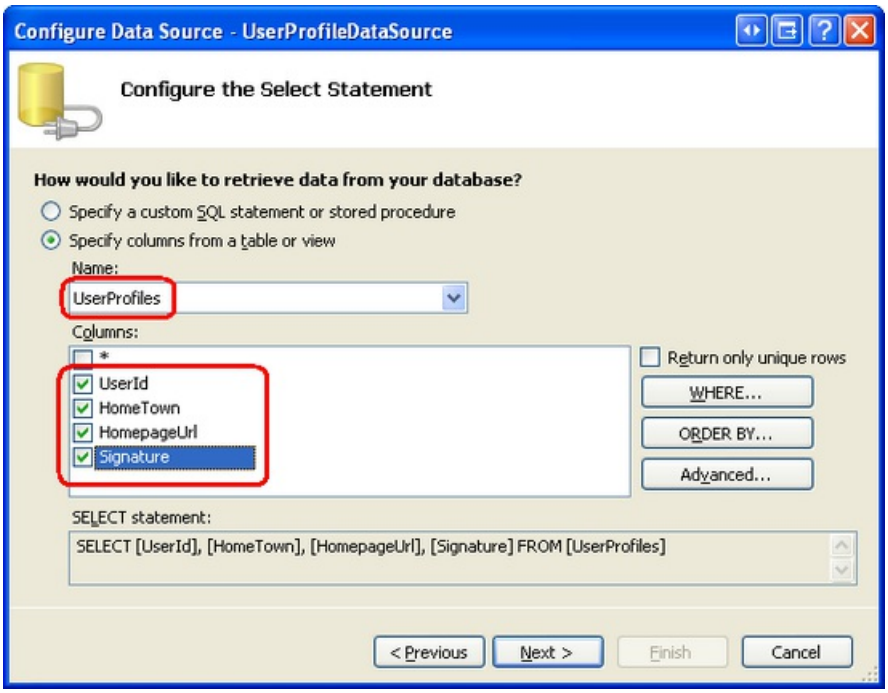


Figure 9: Bring Back All of the Columns from the `UserProfiles` Table (Click to view full-size image)

The current query in Figure 9 returns *all* of the records in `UserProfiles`, but we are only interested in the currently logged on user's record. To add a `WHERE` clause, click the `WHERE` button to bring up the Add `WHERE` Clause dialog box (see Figure 10). Here you can select the column to filter on, the operator, and the source of the filter parameter. Select `UserId` as the column and "=" as the Operator.

Unfortunately there is no built-in parameter source to return the currently logged on user's `UserId` value. We will need to grab this value programmatically. Therefore, set the Source drop-down list to "None," click the Add button to add the parameter, and then click OK.

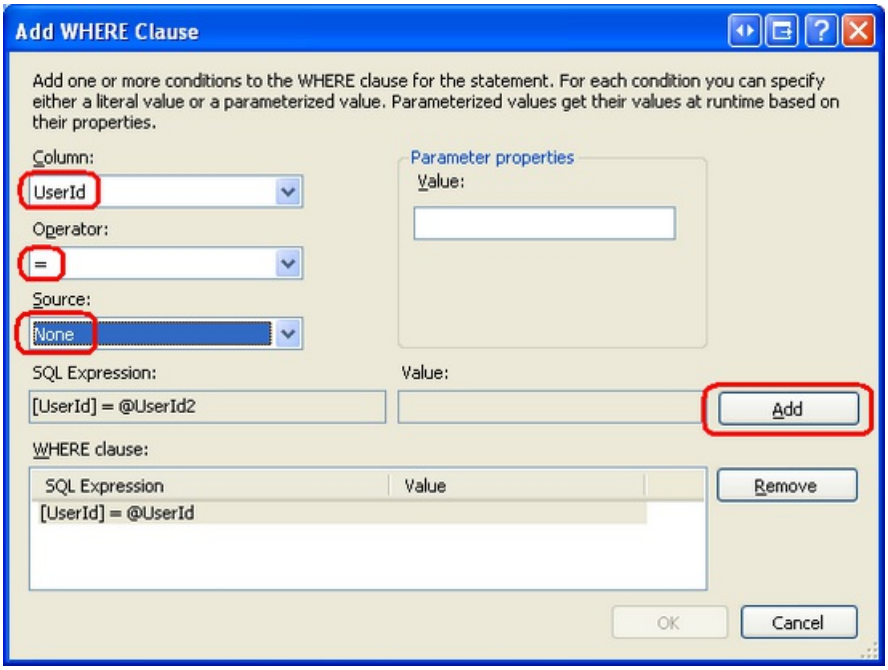


Figure 10: Add a Filter Parameter on the `UserId` Column (Click to view full-size image)

After clicking OK you will be returned to the screen shown in Figure 9. This time, however, the SQL query at the bottom of the screen should include a `WHERE` clause. Click Next to move on to the "Test Query" screen. Here you can run the query and see the results. Click Finish to complete the wizard.

Upon completing the DataSource Configuration Wizard, Visual Studio creates the SqlDataSource control based on the settings specified in the wizard. Moreover, it manually adds BoundFields to the DetailsView for each

column returned by the SqlDataSource's `SelectCommand`. There's no need to show the `UserId` field in the DetailsView, since the user does not need to know this value. You can remove this field directly from the DetailsView control's declarative markup or by clicking the "Edit Fields" link from its Smart Tag.

At this point your page's declarative markup should look similar to the following:

```
<asp:DetailsView ID="UserProfile" runat="server"
          AutoGenerateRows="False" DataKeyNames="UserId"

          DataSourceID="UserProfileDataSource">
     <Fields>
          <asp:BoundField DataField="HomeTown" HeaderText="HomeTown"
               SortExpression="HomeTown" />
          <asp:BoundField DataField="HomepageUrl" HeaderText="HomepageUrl"

               SortExpression="HomepageUrl" />
          <asp:BoundField DataField="Signature" HeaderText="Signature"
               SortExpression="Signature" />
     </Fields>

</asp:DetailsView>
<asp:SqlDataSource ID="UserProfileDataSource" runat="server"
          ConnectionString="<%$ ConnectionStrings:SecurityTutorialsConnectionString %>"
          SelectCommand="SELECT [UserId], [HomeTown], [HomepageUrl], [Signature] FROM
          [UserProfiles] WHERE ([UserId] = @UserId)">
     <SelectParameters>

          <asp:Parameter Name="UserId" Type="Object" />
     </SelectParameters>
</asp:SqlDataSource>
```

We need to programmatically set the SqlDataSource control's `UserId` parameter to the currently logged in user's `UserId` before the data is selected. This can be accomplished by creating an event handler for the SqlDataSource's `Selecting` event and adding the following code there:

```
Protected Sub UserProfileDataSource_Selecting(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.SqlDataSourceSelectingEventArgs) Handles UserProfileDataSource.Selecting
     ' Get a reference to the currently logged on user
     Dim currentUser As MembershipUser = Membership.GetUser()

     ' Determine the currently logged on user's UserId value
     Dim currentUserId As Guid = CType(currentUser.ProviderUserKey, Guid)

     ' Assign the currently logged on user's UserId to the @UserId parameter
     e.Command.Parameters("@UserId").Value = currentUserId
End Sub
```

The above code starts by obtaining a reference to the currently logged on user by calling the `Membership` class's `GetUser` method. This returns a `MembershipUser` object, whose `ProviderUserKey` property contains the `UserId`. The `UserId` value is then assigned to the SqlDataSource's `@UserId` parameter.

> **NOTE**
>
> The `Membership.GetUser()` method returns information about the currently logged on user. If an anonymous user is visiting the page, it will return a value of `Nothing`. In such a case, this will lead to a `NullReferenceException` on the following line of code when attempting to read the `ProviderUserKey` property. Of course, we don't have to worry about `Membership.GetUser()` returning Nothing in the `AdditionalUserInfo.aspx` page because we configured URL authorization in a previous tutorial so that only authenticated users could access the ASP.NET resources in this folder. If you need to access information about the currently logged on user in a page where anonymous access is permitted, make sure to check that the `MembershipUser` object returned from the `GetUser()` method is not Nothing before referencing its properties.

If you visit the `AdditionalUserInfo.aspx` page through a browser you will see a blank page because we have yet to add any rows to the `UserProfiles` table. In Step 6 we will look at how to customize the CreateUserWizard control to automatically add a new row to the `UserProfiles` table when a new user account is created. For now, however, we will need to manually create a record in the table.

Navigate to the Database Explorer in Visual Studio and expand the Tables folder. Right-click on the `aspnet_Users` table and choose "Show Table Data" to see the records in the table; do the same thing for the `UserProfiles` table. Figure 11 shows these results when tiled vertically. In my database there are currently `aspnet_Users` records for Bruce, Fred, and Tito, but no records in the `UserProfiles` table.

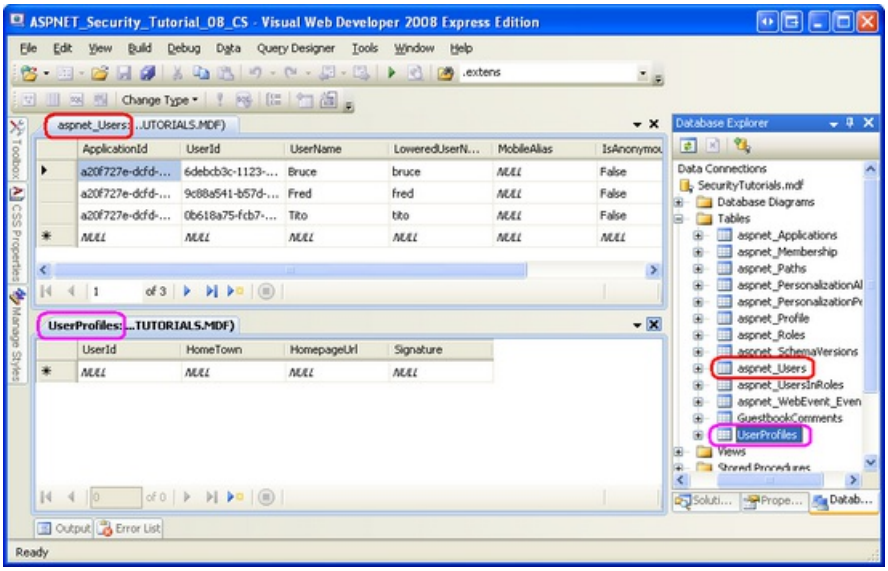


Figure 11: The Contents of the `aspnet_Users` and `UserProfiles` Tables are Displayed (Click to view full-size image)

Add a new record to the `UserProfiles` table by manually typing in values for the `HomeTown`, `HomepageUrl`, and `Signature` fields. The easiest way to get a valid `UserId` value in the new `UserProfiles` record is to select the `UserId` field from a particular user account in the `aspnet_Users` table and copy and paste it into the `UserId` field in `UserProfiles`. Figure 12 shows the `UserProfiles` table after a new record has been added for Bruce.

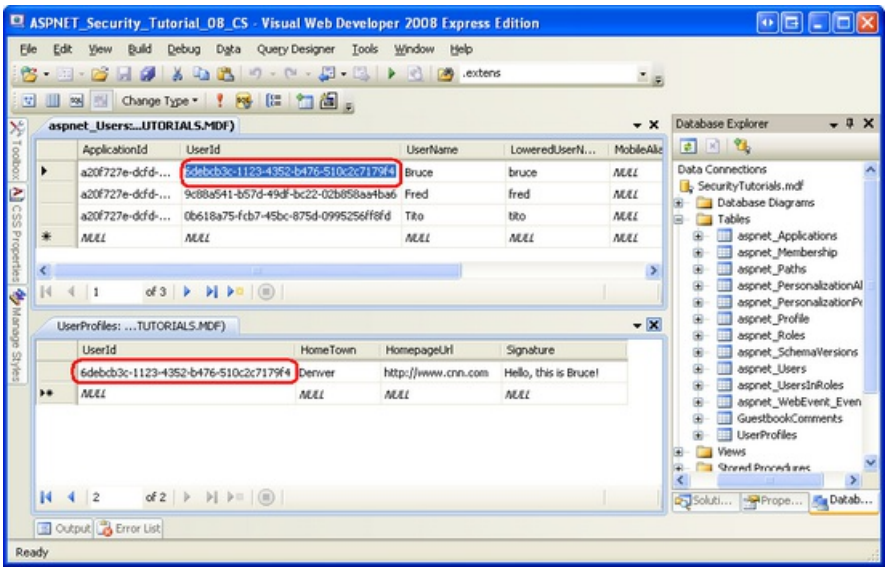


Figure 12: A Record was Added to `UserProfiles` for Bruce (Click to view full-size image)

Return to the `AdditionalUserInfo.aspx page`, logged in as Bruce. As Figure 13 shows, Bruce's settings are displayed.

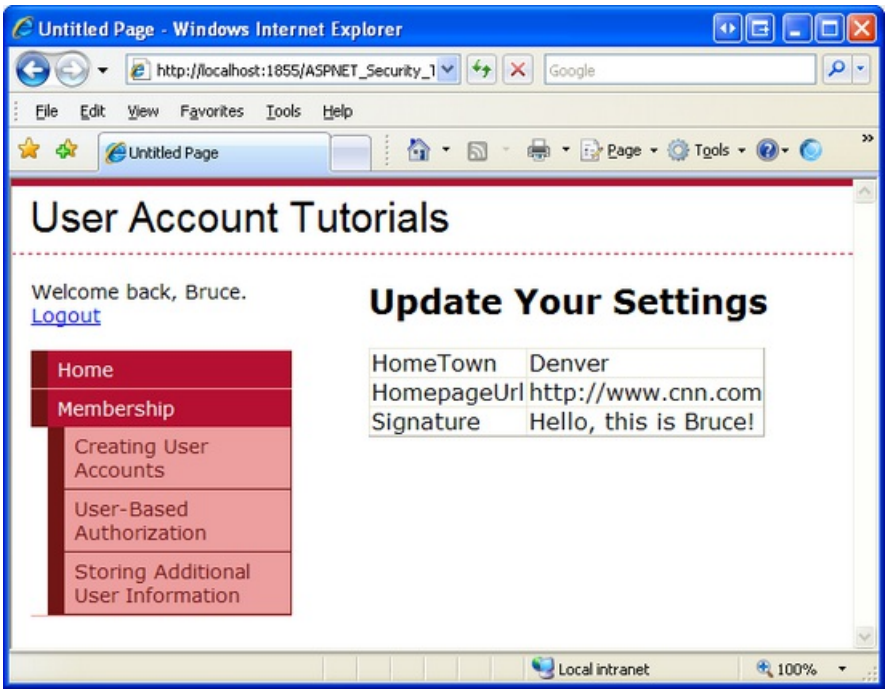


Figure 13: The Currently Visiting User is Shown His Settings (Click to view full-size image)

> **NOTE**
>
> Go ahead and manually add records in the `UserProfiles` table for each Membership user. In Step 6 we will look at how to customize the CreateUserWizard control to automatically add a new row to the `UserProfiles` table when a new user account is created.

Step 3: Allowing the User to Edit His Home Town, Homepage, and Signature

At this point the currently logged in user can view their home town, homepage, and signature setting, but they cannot yet modify them. Let's update the DetailsView control so that the data can be edited.

The first thing we need to do is add an `UpdateCommand` for the SqlDataSource, specifying the `UPDATE` statement to

execute and its corresponding parameters. Select the SqlDataSource and, from the Properties window, click on the ellipses next to the UpdateQuery property to bring up the Command and Parameter Editor dialog box. Enter the following `UPDATE` statement into the textbox:

```
UPDATE UserProfiles SET
     HomeTown = @HomeTown,
     HomepageUrl = @HomepageUrl,
     Signature = @Signature
WHERE UserId = @UserId
```

Next, click the "Refresh Parameters" button, which will create a parameter in the SqlDataSource control's `UpdateParameters` collection for each of the parameters in the `UPDATE` statement. Leave the source for all of the parameters set to None and click the OK button to complete the dialog box.

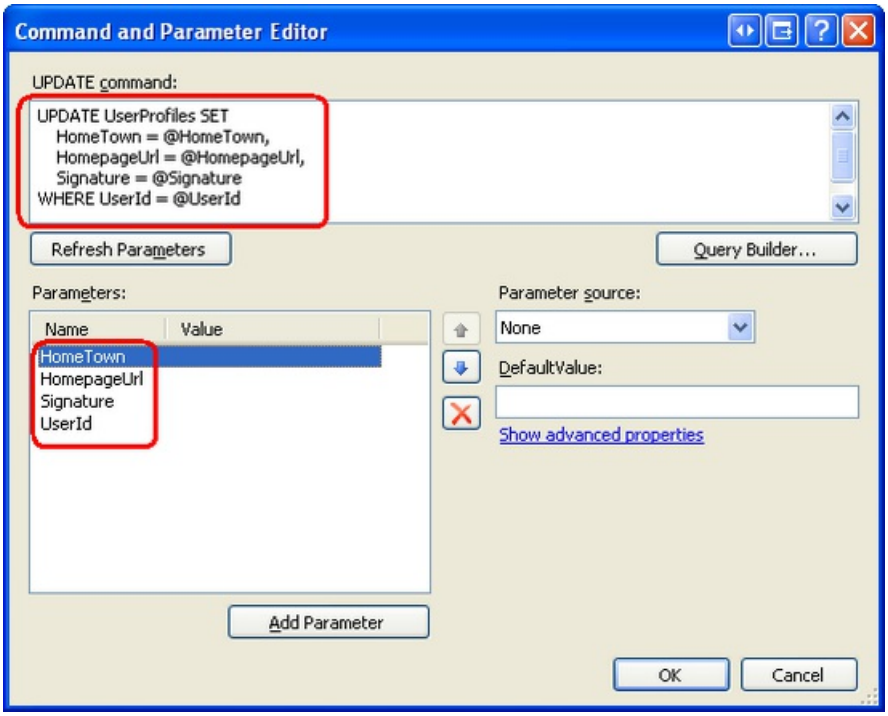


Figure 14: Specify the SqlDataSource's `UpdateCommand` and `UpdateParameters` (Click to view full-size image)

Due to the additions we made to the SqlDataSource control, the DetailsView control can now support editing. From the DetailsView's Smart Tag, check the "Enable Editing" checkbox. This adds a CommandField to the control's `Fields` collection with its `ShowEditButton` property set to True. This renders an Edit button when the DetailsView is displayed in read-only mode and Update and Cancel buttons when displayed in edit mode. Rather than requiring the user to click Edit, though, we can have the DetailsView render in an "always editable" state by setting the DetailsView control's `DefaultMode` property to `Edit`.

With these changes, your DetailsView control's declarative markup should look similar to the following:

```
<asp:DetailsView ID="UserProfile" runat="server"

          AutoGenerateRows="False" DataKeyNames="UserId"
          DataSourceID="UserProfileDataSource" DefaultMode="Edit">
     <Fields>
          <asp:BoundField DataField="HomeTown" HeaderText="HomeTown"

               SortExpression="HomeTown" />
          <asp:BoundField DataField="HomepageUrl" HeaderText="HomepageUrl"
               SortExpression="HomepageUrl" />
          <asp:BoundField DataField="Signature" HeaderText="Signature"

               SortExpression="Signature" />
          <asp:CommandField ShowEditButton="True" />
     </Fields>
</asp:DetailsView>
```

Note the addition of the CommandField and the `DefaultMode` property.

Go ahead and test this page through a browser. When visiting with a user that has a corresponding record in `UserProfiles`, the user's settings are displayed in an editable interface.

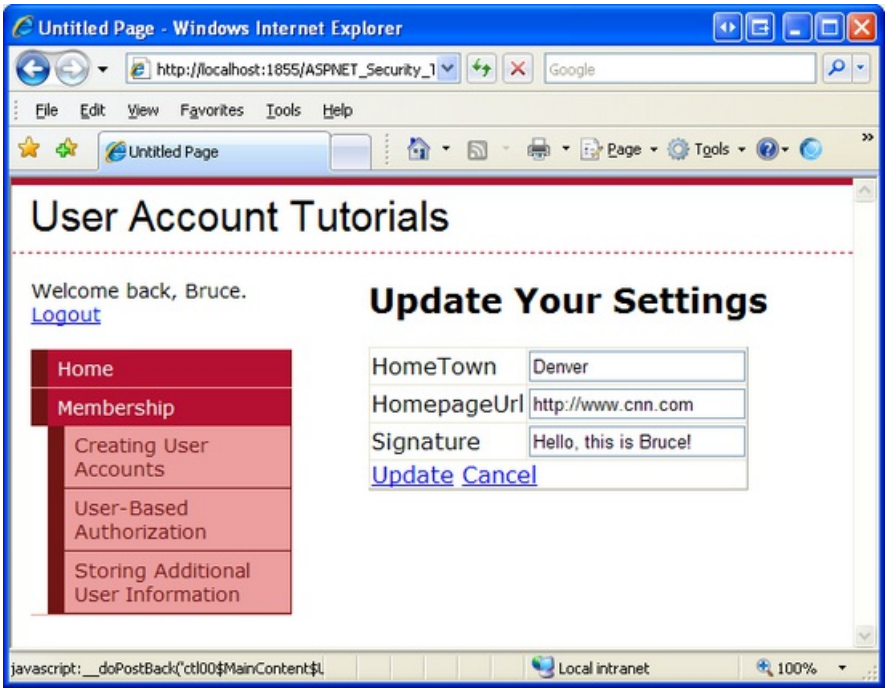


Figure 15: The DetailsView Renders an Editable Interface (Click to view full-size image)

Try changing the values and clicking the Update button. It appears as if nothing happens. There is a postback and the values are saved to the database, but there's no visual feedback that the save occurred.

To remedy this, return to Visual Studio and add a Label control above the DetailsView. Set its `ID` to `SettingsUpdatedMessage`, its `Text` property to "Your settings have been updated," and its `Visible` and `EnableViewState` properties to `False`.

```
<asp:Label ID="SettingsUpdatedMessage" runat="server"
     Text="Your settings have been updated."
     EnableViewState="false"
     Visible="false">
</asp:Label>
```

We need to display the `SettingsUpdatedMessage` Label whenever the DetailsView is updated. To accomplish this, create an event handler for the DetailsView's `ItemUpdated` event and add the following code:

```
Protected Sub UserProfile_ItemUpdated(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.DetailsViewUpdatedEventArgs) Handles UserProfile.ItemUpdated
     SettingsUpdatedMessage.Visible = True
End Sub
```

Return to the `AdditionalUserInfo.aspx` page through a browser and update the data. This time, a helpful status message is displayed.

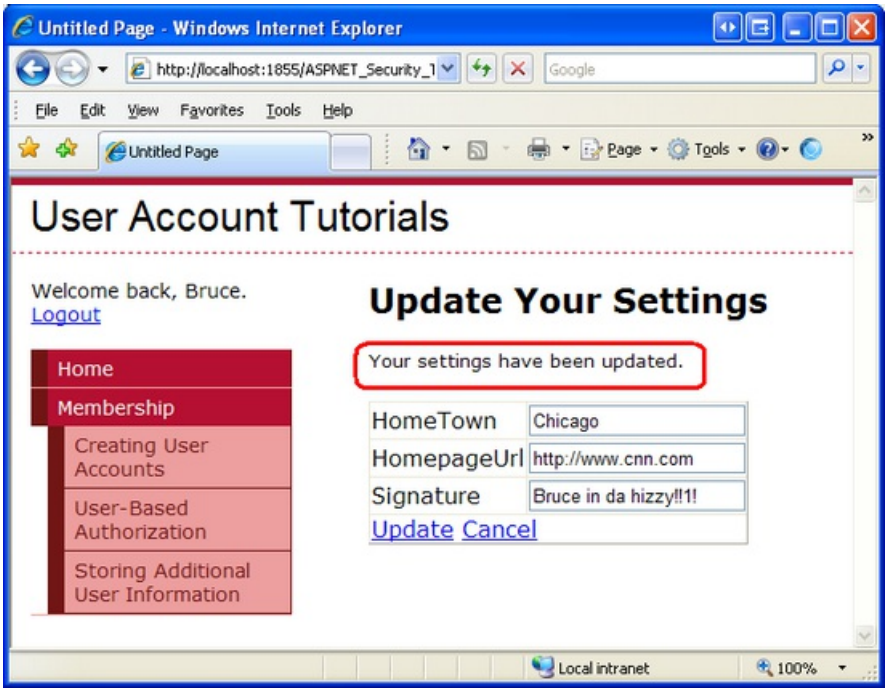


Figure 16: A Short Message is Displayed When the Settings are Updated (Click to view full-size image)

> **NOTE**
>
> The DetailsView control's editing interface leaves a lot to be desired. It uses standard-sized textboxes, but the Signature field should probably be a multi-line textbox. A RegularExpressionValidator should be used to ensure that the homepage URL, if entered, starts with "http://" or "https://". Moreover, since the DetailsView control has its `DefaultMode` property set to `Edit`, the Cancel button does not do anything. It should either be removed or, when clicked, redirect the user to some other page (such as `~/Default.aspx`). I leave these enhancements as an exercise for the reader.

Adding a Link to the `AdditionalUserInfo.aspx` Page in the Master Page

Currently, the website does not provide any links to the `AdditionalUserInfo.aspx` page. The only way to reach it is to enter the page's URL directly into the browser's Address bar. Let's add a link to this page in the `Site.master` master page.

Recall that the master page contains a LoginView Web control in its `LoginContent` ContentPlaceHolder that displays different markup for authenticated and anonymous visitors. Update the LoginView control's `LoggedInTemplate` to include a link to the `AdditionalUserInfo.aspx` page. After making these changes the LoginView control's declarative markup should look similar to the following:

```
<asp:LoginView ID="LoginView1" runat="server">
     <LoggedInTemplate>
          Welcome back,
          <asp:LoginName ID="LoginName1" runat="server" />.

          <br />
          <asp:HyperLink ID="lnkUpdateSettings" runat="server"
               NavigateUrl="~/Membership/AdditionalUserInfo.aspx">
               Update Your Settings</asp:HyperLink>
     </LoggedInTemplate>
     <AnonymousTemplate>

          Hello, stranger.
     </AnonymousTemplate>
</asp:LoginView>
```

Note the addition of the `lnkUpdateSettings` HyperLink control to the `LoggedInTemplate`. With this link in place, authenticated users can quickly jump to the page to view and modify their home town, homepage, and signature settings.

Step 4: Adding New Guestbook Comments

The `Guestbook.aspx` page is where authenticated users can view the guestbook and leave a comment. Let's start with creating the interface to add new guestbook comments.

Open the `Guestbook.aspx` page in Visual Studio and construct a user interface consisting of two TextBox controls, one for the new comment's subject and one for its body. Set the first TextBox control's `ID` property to `Subject` and its `Columns` property to 40; set second's `ID` to `Body`, its `TextMode` to `MultiLine`, and its `Width` and `Rows` properties to "95%" and 8, respectively. To complete the user interface, add a Button Web control named `PostCommentButton` and set its `Text` property to "Post Your Comment".

Since each guestbook comment requires a subject and body, add a RequiredFieldValidator for each of the TextBoxes. Set the `ValidationGroup` property of these controls to "EnterComment"; likewise, set the `PostCommentButton` control's `ValidationGroup` property to "EnterComment". For more information on ASP.NET's validation controls, check out Form Validation in ASP.NET, Dissecting the Validation Controls in ASP.NET 2.0, and the Validation Server Controls Tutorial on W3Schools.

After crafting the user interface your page's declarative markup should look something like the following:

```
<h3>Leave a Comment</h3>
<p>
     <b>Subject:</b>
     <asp:RequiredFieldValidator ID="SubjectReqValidator" runat="server"

          ErrorMessage="You must provide a value for Subject"
          ControlToValidate="Subject" ValidationGroup="EnterComment">
     </asp:RequiredFieldValidator><br />
     <asp:TextBox ID="Subject" Columns="40" runat="server"></asp:TextBox>

</p>
<p>
     <b>Body:</b>
     <asp:RequiredFieldValidator ID="BodyReqValidator" runat="server"
          ControlToValidate="Body"

          ErrorMessage="You must provide a value for Body" ValidationGroup="EnterComment">
     </asp:RequiredFieldValidator><br />
     <asp:TextBox ID="Body" TextMode="MultiLine" Width="95%"

          Rows="8" runat="server"></asp:TextBox>
</p>
<p>
     <asp:Button ID="PostCommentButton" runat="server"

          Text="Post Your Comment"
          ValidationGroup="EnterComment" />
</p>
```

With the user interface complete, our next task is to insert a new record into the `GuestbookComments` table when the `PostCommentButton` is clicked. This can be accomplished in a number of ways: we can write ADO.NET code in the Button's `Click` event handler; we can add a SqlDataSource control to the page, configure its `InsertCommand`, and then call its `Insert` method from the `Click` event handler; or we could build a middle tier that was responsible for inserting new guestbook comments, and invoke this functionality from the `Click` event handler. Since we looked at using a SqlDataSource in Step 3, let's use ADO.NET code here.

> **NOTE**
>
> The ADO.NET classes used to programmatically access data from a Microsoft SQL Server database are located in the `System.Data.SqlClient` namespace. You may need to import this namespace into your page's code-behind class (i.e., `Imports System.Data.SqlClient`).

Create an event handler for the `PostCommentButton`'s `Click` event and add the following code:

```
Protected Sub PostCommentButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
PostCommentButton.Click
     If Not Page.IsValid Then Exit Sub

     ' Determine the currently logged on user's UserId
     Dim currentUser As MembershipUser = Membership.GetUser()
     Dim currentUserId As Guid = CType(currentUser.ProviderUserKey, Guid)

     ' Insert a new record into GuestbookComments
     Dim connectionString As String =
          ConfigurationManager.ConnectionStrings("SecurityTutorialsConnectionString").ConnectionString
     Dim insertSql As String = "INSERT INTO GuestbookComments(Subject, Body, UserId)
          VALUES(@Subject, @Body, @UserId)"

     Using myConnection As New SqlConnection(connectionString)

          myConnection.Open()
          Dim myCommand As New SqlCommand(insertSql, myConnection)
          myCommand.Parameters.AddWithValue("@Subject", Subject.Text.Trim())
          myCommand.Parameters.AddWithValue("@Body", Body.Text.Trim())
          myCommand.Parameters.AddWithValue("@UserId", currentUserId)
          myCommand.ExecuteNonQuery()
          myConnection.Close()
     End Using

     ' "Reset" the Subject and Body TextBoxes

     Subject.Text = String.Empty
     Body.Text = String.Empty
End Sub
```

The `Click` event handler starts by checking that the user-supplied data is valid. If it is not, the event handler exits before inserting a record. Assuming the supplied data is valid, the currently logged on user's `UserId` value is retrieved and stored in the `currentUserId` local variable. This value is needed because we must supply a `UserId` value when inserting a record into `GuestbookComments`.

Following that, the connection string for the `SecurityTutorials` database is retrieved from `Web.config` and the `INSERT` SQL statement is specified. A `SqlConnection` object is then created and opened. Next, a `SqlCommand` object is constructed and the values for the parameters used in the `INSERT` query are assigned. The `INSERT` statement is then executed and the connection closed. At the end of the event handler, the `Subject` and `Body` TextBoxes' `Text` properties are cleared out so that the user's values are not persisted across the postback.

Go ahead and test out this page in a browser. Since this page is in the `Membership` folder it is not accessible to anonymous visitors. Therefore, you will need to first log on (if you have not already). Enter a value into the `Subject` and `Body` TextBoxes and click the `PostCommentButton` button. This will cause a new record to be added to `GuestbookComments`. On postback, the subject and body you provided are wiped from the TextBoxes.

After clicking the `PostCommentButton` button there is no visual feedback that the comment was added to the guestbook. We still need to update this page to display the existing guestbook comments, which we will do in Step 5. Once we accomplish that, the just-added comment will appear in the list of comments, providing adequate visual feedback. For now, confirm that your guestbook comment was saved by examining the contents of the `GuestbookComments` table.

Figure 17 shows the contents of the `GuestbookComments` table after two comments have been left.

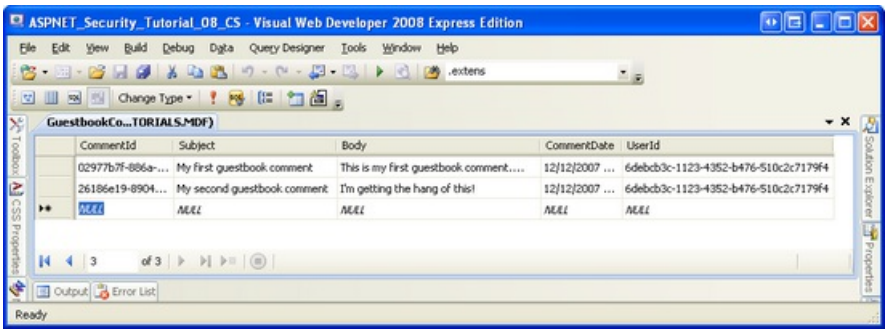


Figure 17: You Can See the Guestbook Comments in the `GuestbookComments` Table (Click to view full-size image)

> **NOTE**
>
> If a user attempts to insert a guestbook comment that contains potentially dangerous markup – such as HTML – ASP.NET will throw an `HttpRequestValidationException`. To learn more about this exception, why it's thrown, and how to permit users to submit potentially dangerous values, consult the Request Validation Whitepaper.

Step 5: Listing the Existing Guestbook Comments

In addition to leaving comments, a user visiting the `Guestbook.aspx` page should also be able to view the guestbook's existing comments. To accomplish this, add a ListView control named `CommentList` to the bottom of the page.

> **NOTE**
>
> The ListView control is new to ASP.NET version 3.5. It is designed to display a list of items in a very customizable and flexible layout, yet still offer built-in editing, inserting, deleting, paging, and sorting functionality like the GridView. If you are using ASP.NET 2.0, you will need to use the DataList or Repeater control instead. For more information on using the ListView, see Scott Guthrie's blog entry, The asp:ListView Control, and my article, Displaying Data with the ListView Control.

Open the ListView's Smart Tag and, from the Choose Data Source drop-down list, bind the control to a new data source. As we saw in Step 2, this will launch the Data Source Configuration Wizard. Select the Database icon, name the resulting SqlDataSource `CommentsDataSource`, and click OK. Next, select the `SecurityTutorialsConnectionString` connection string from the drop-down list and click Next.

At this point in Step 2 we specified the data to query by picking the `UserProfiles` table from the drop-down list and selecting the columns to return (refer back to Figure 9). This time, however, we want to craft a SQL statement that pulls back not only the records from `GuestbookComments`, but also the commenter's home town, homepage, signature, and username. Therefore, select the "Specify a custom SQL statement or stored procedure" radio button and click Next.

This will bring up the "Define Custom Statements or Stored Procedures" screen. Click the Query Builder button to graphically build the query. The Query Builder starts by prompting us to specify the tables we want to query from. Select the `GuestbookComments`, `UserProfiles`, and `aspnet_Users` tables and click OK. This will add all three tables to the design surface. Since there are foreign key constraints amongst the `GuestbookComments`, `UserProfiles`, and `aspnet_Users` tables, the Query Builder automatically `JOIN` s these tables.

All that remains is to specify the columns to return. From the `GuestbookComments` table select the `Subject`, `Body`, and `CommentDate` columns; return the `HomeTown`, `HomepageUrl`, and `Signature` columns from the `UserProfiles` table; and return `UserName` from `aspnet_Users`. Also, add "`ORDER BY CommentDate DESC`" to the end of the `SELECT` query so that the most recent posts are returned first. After making these selections, your Query Builder interface should look similar to the screen shot in Figure 18.

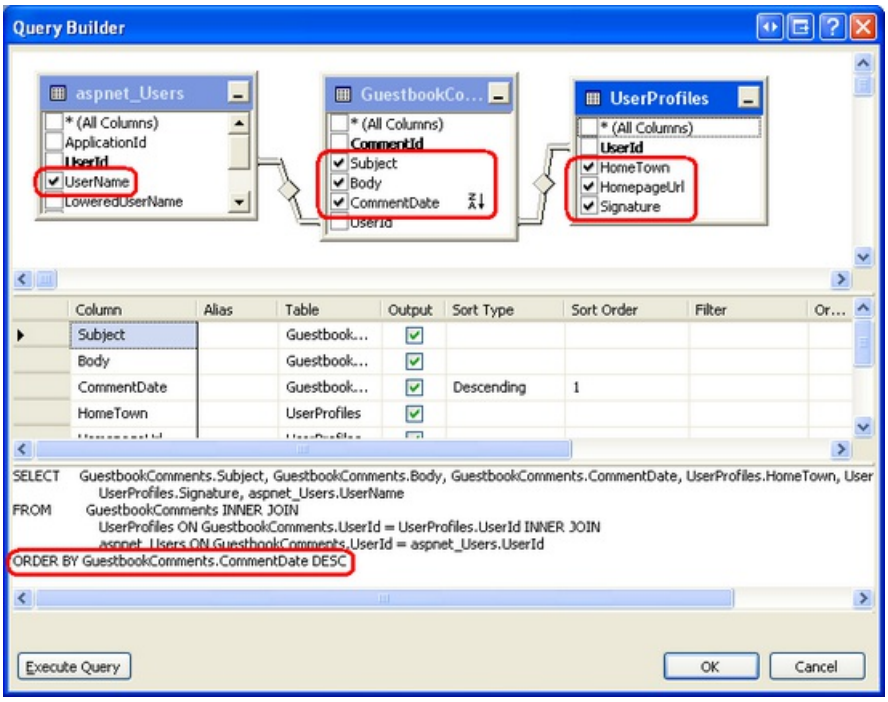


Figure 18: The Constructed Query `JOIN` s the `GuestbookComments`, `UserProfiles`, and `aspnet_Users` Tables (Click to view full-size image)

Click OK to close the Query Builder window and return to the "Define Custom Statements or Stored Procedures" screen. Click Next to advance to the "Test Query" screen, where you may view the query results by clicking the Test Query button. When you're ready, click Finish to complete the Configure Data Source wizard.

When we completed the Configure Data Source wizard in Step 2, the associated DetailsView control's `Fields` collection was updated to include a BoundField for each column returned by the `SelectCommand`. The ListView, however, remains unchanged; we still need to define its layout. The ListView's layout can be constructed manually through its declarative markup or from the "Configure ListView" option in its Smart Tag. I usually prefer defining the markup by hand, but use whatever method is most natural to you.

I ended up using the following `LayoutTemplate`, `ItemTemplate`, and `ItemSeparatorTemplate` for my ListView control:

```
<asp:ListView ID="CommentList" runat="server" DataSourceID="CommentsDataSource">

     <LayoutTemplate>
          <span ID="itemPlaceholder" runat="server" />
          <p>
               <asp:DataPager ID="DataPager1" runat="server">

                    <Fields>
                         <asp:NextPreviousPagerField ButtonType="Button"
                              ShowFirstPageButton="True"
                              ShowLastPageButton="True" />
                    </Fields>
               </asp:DataPager>

          </p>
     </LayoutTemplate>
     <ItemTemplate>
          <h4>
               <asp:Label ID="SubjectLabel" runat="server"
                    Text='<%# Eval("Subject") %>' />

          </h4>
          <asp:Label ID="BodyLabel" runat="server"
               Text='<%# Eval("Body").ToString().Replace(Environment.NewLine, "<br />") %>' />
          <p>

          ---<br />
          <asp:Label ID="SignatureLabel" Font-Italic="true" runat="server"
               Text='<%# Eval("Signature") %>' />

          <br />
          <br />
          My Home Town:
          <asp:Label ID="HomeTownLabel" runat="server"
               Text='<%# Eval("HomeTown") %>' />

          <br />
          My Homepage:
          <asp:HyperLink ID="HomepageUrlLink" runat="server"
               NavigateUrl='<%# Eval("HomepageUrl") %>'
               Text='<%# Eval("HomepageUrl") %>' />

          </p>
          <p align="center">
          Posted by
          <asp:Label ID="UserNameLabel" runat="server"
               Text='<%# Eval("UserName") %>' />

          on
          <asp:Label ID="CommentDateLabel" runat="server"
               Text='<%# Eval("CommentDate") %>' />
          </p>
     </ItemTemplate>

     <ItemSeparatorTemplate>
          <hr />
     </ItemSeparatorTemplate>
</asp:ListView>
```

The `LayoutTemplate` defines the markup emitted by the control, while the `ItemTemplate` renders each item returned by the SqlDataSource. The `ItemTemplate`'s resulting markup is placed in the `LayoutTemplate`'s `itemPlaceholder` control. In addition to the `itemPlaceholder`, the `LayoutTemplate` includes a DataPager control, which limits the ListView to showing just 10 guestbook comments per page (the default) and renders a paging interface.

My `ItemTemplate` displays each guestbook comment's subject in an `<h4>` element with the body situated below the subject. Note that syntax used for displaying the body takes the data returned by the `Eval("Body")` databinding statement, converts it to a string, and replaces line breaks with the `<br />` element. This conversion is needed in order to show the line breaks entered when submitting the comment since whitespace is ignored by HTML. The user's signature is displayed beneath the body in italics, followed by the user's home town, a link to his homepage, the date and time the comment was made, and the username of the person who left the comment.

Take a moment to view the page through a browser. You should see the comments that you added to the guestbook in Step 5 displayed here.

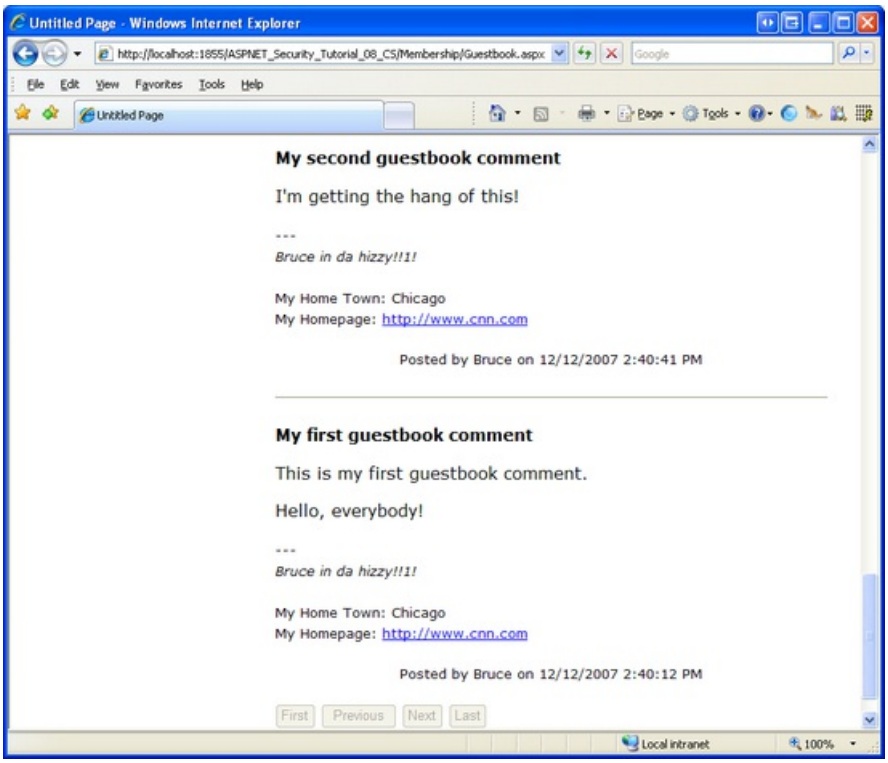


Figure 19: `Guestbook.aspx` Now Displays the Guestbook's Comments (Click to view full-size image)

Try adding a new comment to the guestbook. Upon clicking the `PostCommentButton` button the page posts back and the comment is added to the database, but the ListView control is not updated to show the new comment. This can be fixed by either:

- Updating the `PostCommentButton` button's `Click` event handler so that it invokes the ListView control's `DataBind()` method after inserting the new comment into the database, or
- Setting the ListView control's `EnableViewState` property to `False`. This approach works because by disabling the control's view state, it must rebind to the underlying data on every postback.

The tutorial website downloadable from this tutorial illustrates both techniques. The ListView control's `EnableViewState` property to `False` and the code needed to programmatically rebind the data to the ListView is present in the `Click` event handler, but is commented out.

> **NOTE**
>
> Currently the `AdditionalUserInfo.aspx` page allows the user to view and edit their home town, homepage, and signature settings. It might be nice to update `AdditionalUserInfo.aspx` to display the logged in user's guestbook comments. That is, in addition to examining and modifying her information, a user can visit the `AdditionalUserInfo.aspx` page to see what guestbook comments she's made in the past. I leave this as an exercise for the interested reader.

Step 6: Customizing the CreateUserWizard Control to Include an Interface for the Home Town, Homepage, and Signature

The `SELECT` query used by the `Guestbook.aspx` page uses an `INNER JOIN` to combine the related records amongst the `GuestbookComments`, `UserProfiles`, and `aspnet_Users` tables. If a user that has no record in `UserProfiles` makes a guestbook comment, the comment won't be displayed in the ListView because the `INNER JOIN` only returns `GuestbookComments` records when there are matching records in `UserProfiles` and `aspnet_Users`. And as we saw in Step 3, if a user does not have a record in `UserProfiles` she cannot view or edit her settings in the `AdditionalUserInfo.aspx` page.

Needless to say, due to our design decisions it is important that every user account in the Membership system have a matching record in the `UserProfiles` table. What we want is for a corresponding record to be added to `UserProfiles` whenever a new Membership user account is created through the CreateUserWizard.

As discussed in the *Creating User Accounts* tutorial, after the new Membership user account is created the CreateUserWizard control raises its `CreatedUser` event. We can create an event handler for this event, get the UserId for the just-created user, and then insert a record into the `UserProfiles` table with default values for the `HomeTown`, `HomepageUrl`, and `Signature` columns. What's more, it is possible to prompt the user for these values by customizing the CreateUserWizard control's interface to include additional TextBoxes.

Let's first look at how to add a new row to the `UserProfiles` table in the `CreatedUser` event handler with default values. Following that, we will see how to customize the CreateUserWizard control's user interface to include additional form fields to collect the new user's home town, homepage, and signature.

Adding a Default Row to `UserProfiles`

In the *Creating User Accounts* tutorial we added a CreateUserWizard control to the `CreatingUserAccounts.aspx` page in the `Membership` folder. In order to have the CreateUserWizard control add a record to `UserProfiles` table upon user account creation, we need to update the CreateUserWizard control's functionality. Rather than making these changes to the `CreatingUserAccounts.aspx` page, let's instead add a new CreateUserWizard control to `EnhancedCreateUserWizard.aspx` page and make the modifications for this tutorial there.

Open the `EnhancedCreateUserWizard.aspx` page in Visual Studio and drag a CreateUserWizard control from the Toolbox onto the page. Set the CreateUserWizard control's `ID` property to `NewUserWizard`. As we discussed in the *Creating User Accounts* tutorial, the CreateUserWizard's default user interface prompts the visitor for the necessary information. Once this information has been supplied, the control internally creates a new user account in the Membership framework, all without us having to write a single line of code.

The CreateUserWizard control raises a number of events during its workflow. After a visitor supplies the request information and submits the form, the CreateUserWizard control initially fires its `CreatingUser` event. If there is a problem during the create process, the `CreateUserError` event is fired; however, if the user is successfully created, then the `CreatedUser` event is raised. In the *Creating User Accounts* tutorial we created an event handler for the `CreatingUser` event to ensure that the supplied username did not contain any leading or trailing spaces, and that the username did not appear anywhere in the password.

In order to add a row in the `UserProfiles` table for the just-created user, we need to create an event handler for the `CreatedUser` event. By the time the `CreatedUser` event has fired, the user account has already been created in the Membership framework, enabling us to retrieve the account's UserId value.

Create an event handler for the `NewUserWizard`'s `CreatedUser` event and add the following code:

```
Protected Sub NewUserWizard_CreatedUser(ByVal sender As Object, ByVal e As System.EventArgs) Handles
NewUserWizard.CreatedUser
     ' Get the UserId of the just-added user
     Dim newUser As MembershipUser = Membership.GetUser(NewUserWizard.UserName)
     Dim newUserId As Guid = CType(newUser.ProviderUserKey, Guid)

     ' Insert a new record into UserProfiles
     Dim connectionString As String =

          ConfigurationManager.ConnectionStrings("SecurityTutorialsConnectionString").ConnectionString
     Dim insertSql As String = "INSERT INTO UserProfiles(UserId, HomeTown, HomepageUrl,
          Signature) VALUES(@UserId, @HomeTown, @HomepageUrl, @Signature)"

     Using myConnection As New SqlConnection(connectionString)
          myConnection.Open()
          Dim myCommand As New SqlCommand(insertSql, myConnection)
          myCommand.Parameters.AddWithValue("@UserId", newUserId)
          myCommand.Parameters.AddWithValue("@HomeTown", DBNull.Value)

          myCommand.Parameters.AddWithValue("@HomepageUrl", DBNull.Value)
          myCommand.Parameters.AddWithValue("@Signature", DBNull.Value)
          myCommand.ExecuteNonQuery()
          myConnection.Close()
     End Using
End Sub
```

The above code beings by retrieving the UserId of the just-added user account. This is accomplished by using the `Membership.GetUser(username)` method to return information about a particular user, and then using the `ProviderUserKey` property to retrieve their UserId. The username entered by the user in the CreateUserWizard control is available via its `UserName` property.

Next, the connection string is retrieved from `Web.config` and the `INSERT` statement is specified. The necessary ADO.NET objects are instantiated and the command executed. The code assigns a `DBNull` instance to the `@HomeTown`, `@HomepageUrl`, and `@Signature` parameters, which has the effect of inserting database `NULL` values for the `HomeTown`, `HomepageUrl`, and `Signature` fields.

Visit the `EnhancedCreateUserWizard.aspx` page through a browser and create a new user account. After doing so, return to Visual Studio and examine the contents of the `aspnet_Users` and `UserProfiles` tables (like we did back in Figure 12). You should see the new user account in `aspnet_Users` and a corresponding `UserProfiles` row (with `NULL` values for `HomeTown`, `HomepageUrl`, and `Signature`).

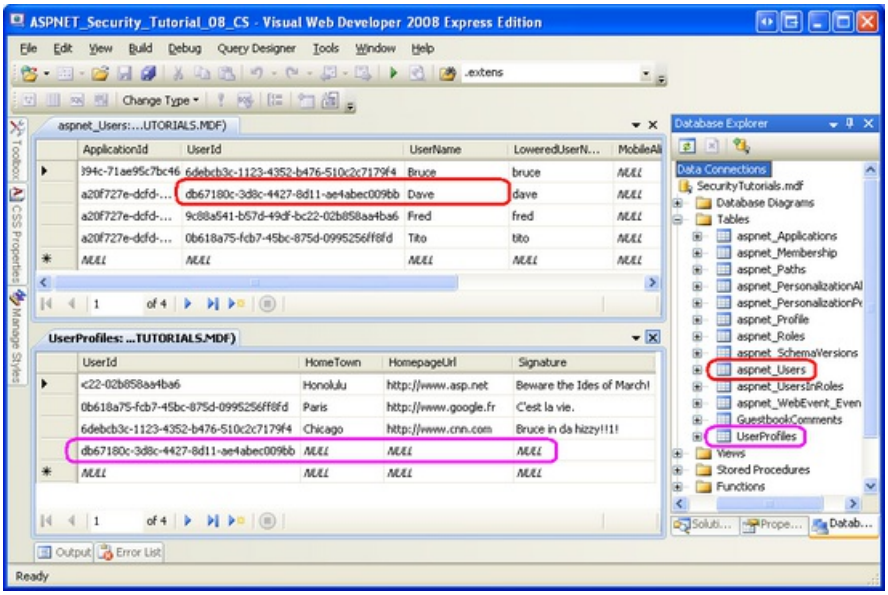


Figure 20: A New User Account and `UserProfiles` Record Have Been Added (Click to view full-size image)

After the visitor has supplied his new account information and clicked the "Create User" button, the user account is created and a row added to the `UserProfiles` table. The CreateUserWizard then displays its `CompleteWizardStep`, which displays a success message and a Continue button. Clicking the Continue button causes a postback, but no action is taken, leaving the user stuck on the `EnhancedCreateUserWizard.aspx` page.

We can specify a URL to send the user to when the Continue button is clicked via the CreateUserWizard control's `ContinueDestinationPageUrl` property. Set the `ContinueDestinationPageUrl` property to "~/Membership/AdditionalUserInfo.aspx". This takes the new user to `AdditionalUserInfo.aspx`, where they can view and update their settings.

Customizing the CreateUserWizard's Interface to Prompt for the New User's Home Town, Homepage, and Signature

The CreateUserWizard control's default interface is sufficient for simple account creation scenarios where only core user account information like username, password, and email need be collected. But what if we wanted to prompt the visitor to enter her home town, homepage, and signature while creating her account? It is possible to customize the CreateUserWizard control's interface to collect additional information at signup, and this information may be used in the `CreatedUser` event handler to insert additional records into the underlying database.

The CreateUserWizard control extends the ASP.NET Wizard control, which is a control that allows a page developer to define a series of ordered `WizardSteps`. The Wizard control renders the active step and provides a navigation interface that allows the visitor to move through these steps. The Wizard control is ideal for breaking down a long task into several short steps. For more information on the Wizard control, see Creating a Step-by-Step User Interface with the ASP.NET 2.0 Wizard Control.

The CreateUserWizard control's default markup defines two `WizardSteps`: `CreateUserWizardStep` and `CompleteWizardStep`.

```
<asp:CreateUserWizard ID="NewUserWizard" runat="server"
          ContinueDestinationPageUrl="~/Membership/AdditionalUserInfo.aspx">
     <WizardSteps>
          <asp:CreateUserWizardStep ID="CreateUserWizardStep1" runat="server">

          </asp:CreateUserWizardStep>
          <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">
          </asp:CompleteWizardStep>
     </WizardSteps>
</asp:CreateUserWizard>
```

The first `WizardStep`, `CreateUserWizardStep`, renders the interface that prompts for the username, password, email, and so on. After the visitor supplies this information and clicks "Create User", she is shown the `CompleteWizardStep`, which shows the success message and a Continue button.

To customize the CreateUserWizard control's interface to include additional form fields, we can:

- **Create one or more new `WizardStep`s to contain the additional user interface elements**. To add a new `WizardStep` to the CreateUserWizard, click the "Add/Remove `WizardStep` s" link from its Smart Tag to launch the `WizardStep` Collection Editor. From there you can add, remove, or reorder the steps in the wizard. This is the approach we will use for this tutorial.
- **Convert the `CreateUserWizardStep` into an editable `WizardStep`**. This replaces the `CreateUserWizardStep` with an equivalent `WizardStep` whose markup defines a user interface that matches the `CreateUserWizardStep`'s. By converting the `CreateUserWizardStep` into a `WizardStep` we can reposition the controls or add additional user interface elements to this step. To convert the `CreateUserWizardStep` or `CompleteWizardStep` into an editable `WizardStep`, click the "Customize Create User Step" or "Customize Complete Step" link from the control's Smart Tag.

- Use some combination of the above two options.

One important thing to keep in mind is that the CreateUserWizard control executes its user account creation process when the "Create User" button is clicked from within its `CreateUserWizardStep`. It doesn't matter if there are additional `WizardStep` s after the `CreateUserWizardStep` or not.

When adding a custom `WizardStep` to the CreateUserWizard control to collect additional user input, the custom `WizardStep` can be placed before or after the `CreateUserWizardStep`. If it comes before the `CreateUserWizardStep` then the additional user input collected from the custom `WizardStep` is available for the `CreatedUser` event handler. However, if the custom `WizardStep` comes after `CreateUserWizardStep` then by the time the custom `WizardStep` is displayed the new user account has already been created and the `CreatedUser` event has already fired.

Figure 21 shows the workflow when the added `WizardStep` precedes the `CreateUserWizardStep`. Since the additional user information has been collected by the time the `CreatedUser` event fires, all we have to do is update the `CreatedUser` event handler to retrieve these inputs and use those for the `INSERT` statement's parameter values (rather than `DBNull.Value`).

Figure 21: The CreateUserWizard Workflow When an Additional `WizardStep` Precedes the `CreateUserWizardStep` (Click to view full-size image)

If the custom `WizardStep` is placed *after* the `CreateUserWizardStep`, however, the create user account process occurs before the user has had a chance to enter her home town, homepage, or signature. In such a case, this additional information needs to be inserted into the database after the user account has been created, as Figure 22 illustrates.

Figure 22: The CreateUserWizard Workflow When an Additional `WizardStep` Comes After the `CreateUserWizardStep` (Click to view full-size image)

The workflow shown in Figure 22 waits to insert a record into the `UserProfiles` table until after Step 2 completes. If the visitor closes her browser after step 1, however, we will have reached a state where a user account was created, but no record was added to `UserProfiles`. One workaround is to have a record with `NULL` or default values inserted into `UserProfiles` in the `CreatedUser` event handler (which fires after step 1), and then update this record after step 2 completes. This ensures that a `UserProfiles` record will be added for the user account even if the user quits the registration process midway through.

For this tutorial let's create a new `WizardStep` that occurs after the `CreateUserWizardStep` but before the `CompleteWizardStep`. Let's first get the WizardStep in place and configured and then we'll look at the code.

From the CreateUserWizard control's Smart Tag, select the "Add/Remove `WizardStep` s", which brings up the `WizardStep` Collection Editor dialog. Add a new `WizardStep`, setting its `ID` to `UserSettings`, its `Title` to "Your Settings" and its `StepType` to `Step`. Then position it so that it comes after the `CreateUserWizardStep` ("Sign Up for Your New Account") and before the `CompleteWizardStep` ("Complete"), as shown in Figure 23.

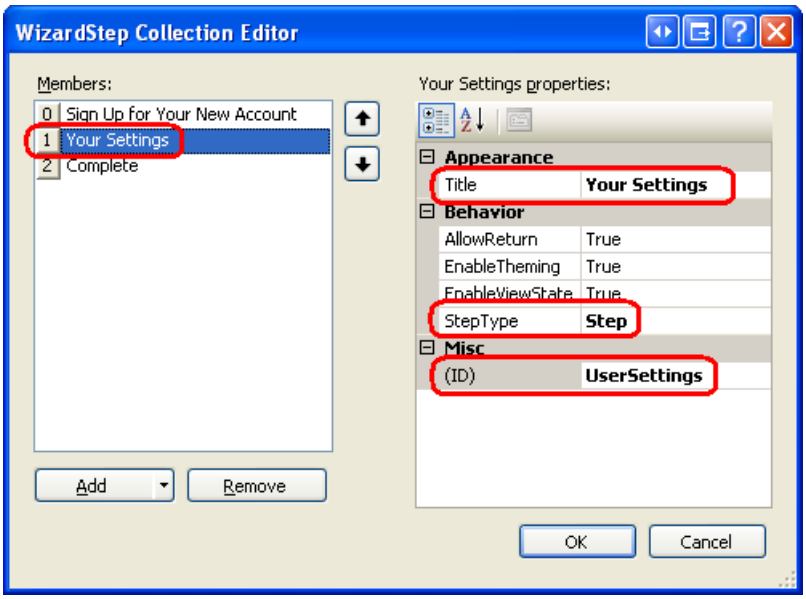


Figure 23: Add a New `WizardStep` to the CreateUserWizard Control (Click to view full-size image)

Click OK to close the `WizardStep` Collection Editor dialog. The new `WizardStep` is evidenced by the CreateUserWizard control's updated declarative markup:

```
<asp:CreateUserWizard ID="NewUserWizard" runat="server"

          ContinueDestinationPageUrl="~/Membership/AdditionalUserInfo.aspx">
     <WizardSteps>
          <asp:CreateUserWizardStep ID="CreateUserWizardStep1" runat="server">
          </asp:CreateUserWizardStep>
          <asp:WizardStep runat="server" ID="UserSettings" StepType="Step"

               Title="Your Settings">
          </asp:WizardStep>
          <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">
          </asp:CompleteWizardStep>
     </WizardSteps>
</asp:CreateUserWizard>
```

Note the new `<asp:WizardStep>` element. We need to add the user interface to collect the new user's home town, homepage, and signature here. You can enter this content in the declarative syntax or through the Designer. To use the Designer, select the "Your Settings" step from the drop-down list in the Smart Tag to see the step in the Designer.

> **NOTE**
>
> Selecting a step through the Smart Tag's drop-down list updates the CreateUserWizard control's `ActiveStepIndex` property, which specifies the index of the starting step. Therefore, if you use this drop-down list to edit the "Your Settings" step in the Designer, be sure to set it back to "Sign Up for Your New Account" so that this step is shown when users first visit the `EnhancedCreateUserWizard.aspx` page.

Create a user interface within the "Your Settings" step that contains three TextBox controls named `HomeTown`, `HomepageUrl`, and `Signature`. After constructing this interface, the CreateUserWizard's declarative markup should look similar to the following:

```
<asp:CreateUserWizard ID="NewUserWizard" runat="server"

          ContinueDestinationPageUrl="~/Membership/AdditionalUserInfo.aspx">
     <WizardSteps>
          <asp:CreateUserWizardStep ID="CreateUserWizardStep1" runat="server">
          </asp:CreateUserWizardStep>
          <asp:WizardStep runat="server" ID="UserSettings" StepType="Step"

                    Title="Your Settings">
               <p>
                    <b>Home Town:</b><br />
                    <asp:TextBox ID="HomeTown" runat="server"></asp:TextBox>

               </p>
               <p>
                    <b>Homepage URL:</b><br />
                    <asp:TextBox ID="HomepageUrl" Columns="40" runat="server"></asp:TextBox>

               </p>
               <p>
                    <b>Signature:</b><br />
                    <asp:TextBox ID="Signature" TextMode="MultiLine" Width="95%"

                         Rows="5" runat="server"></asp:TextBox>
               </p>
          </asp:WizardStep>
          <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">

          </asp:CompleteWizardStep>
     </WizardSteps>
</asp:CreateUserWizard>
```

Go ahead and visit this page through a browser and create a new user account, specifying values for the home town, homepage, and signature. After completing the `CreateUserWizardStep` the user account is created in the Membership framework and the `CreatedUser` event handler runs, which adds a new row to `UserProfiles`, but with a database `NULL` value for `HomeTown`, `HomepageUrl`, and `Signature`. The values entered for the home town, homepage, and signature are never used. The net result is a new user account with a `UserProfiles` record whose `HomeTown`, `HomepageUrl`, and `Signature` fields have yet to be specified.

We need to execute code after the "Your Settings" step that takes the home town, honepage, and signature values entered by the user and updates the appropriate `UserProfiles` record. Each time the user moves between steps in a Wizard control, the Wizard's `ActiveStepChanged` event fires. We can create an event handler for this event and update the `UserProfiles` table when the "Your Settings" step has completed.

Add an event handler for the CreateUserWizard's `ActiveStepChanged` event and add the following code:

```
Protected Sub NewUserWizard_ActiveStepChanged(ByVal sender As Object, ByVal e As System.EventArgs) Handles
NewUserWizard.ActiveStepChanged
     ' Have we JUST reached the Complete step?
     If NewUserWizard.ActiveStep.Title = "Complete" Then
          Dim UserSettings As WizardStep = CType(NewUserWizard.FindControl("UserSettings"),WizardStep)

          ' Programmatically reference the TextBox controls
          Dim HomeTown As TextBox = CType(UserSettings.FindControl("HomeTown"), TextBox)
          Dim HomepageUrl As TextBox = CType(UserSettings.FindControl("HomepageUrl"), TextBox)
          Dim Signature As TextBox = CType(UserSettings.FindControl("Signature"), TextBox)

          ' Update the UserProfiles record for this user
          ' Get the UserId of the just-added user
          Dim newUser As MembershipUser = Membership.GetUser(NewUserWizard.UserName)
          Dim newUserId As Guid = CType(newUser.ProviderUserKey, Guid)

          ' Insert a new record into UserProfiles
          Dim connectionString As String =
ConfigurationManager.ConnectionStrings("SecurityTutorialsConnectionString").ConnectionString

          Dim updateSql As String = "UPDATE UserProfiles SET HomeTown = @HomeTown, HomepageUrl
               = @HomepageUrl, Signature = @Signature WHERE UserId = @UserId"

          Using myConnection As New SqlConnection(connectionString)
               myConnection.Open()
               Dim myCommand As New SqlCommand(updateSql, myConnection)
               myCommand.Parameters.AddWithValue("@HomeTown", HomeTown.Text.Trim())
               myCommand.Parameters.AddWithValue("@HomepageUrl", HomepageUrl.Text.Trim())
               myCommand.Parameters.AddWithValue("@Signature", Signature.Text.Trim())

               myCommand.Parameters.AddWithValue("@UserId", newUserId)
               myCommand.ExecuteNonQuery()
               myConnection.Close()
          End Using
     End If
End Sub
```

The above code starts by determining if we have just reached the "Complete" step. Since the "Complete" step occurs immediately after the "Your Settings" step, then when the visitor reaches "Complete" step that means she just finished the "Your Settings" step.

In such a case, we need to programmatically reference the TextBox controls within the `UserSettings WizardStep`. This is accomplished by first using the `FindControl` method to programmatically referencing the `UserSettings WizardStep`, and then again to reference the TextBoxes from within the `WizardStep`. Once the TextBoxes have been referenced, we're ready to execute the `UPDATE` statement. The `UPDATE` statement has the same number of parameters as the `INSERT` statement in the `CreatedUser` event handler, but here we use the home town, homepage, and signature values supplied by the user.

With this event handler in place, visit the `EnhancedCreateUserWizard.aspx` page through a browser and create a new user account specifying values for the home town, homepage, and signature. After creating the new account you should be redirected to the `AdditionalUserInfo.aspx` page, where the just-entered home town, homepage, and signature information is displayed.

NOTE

Our website currently has two pages from which a visitor can create a new account: `CreatingUserAccounts.aspx` and `EnhancedCreateUserWizard.aspx`. The website's sitemap and login page point to the `CreatingUserAccounts.aspx` page, but the `CreatingUserAccounts.aspx` page does not prompt the user for their home town, homepage, and signature information and does not add a corresponding row to `UserProfiles`. Therefore, either update the `CreatingUserAccounts.aspx` page so that it offers this functionality or update the sitemap and login page to reference `EnhancedCreateUserWizard.aspx` instead of `CreatingUserAccounts.aspx`. If you choose the latter option, be sure to update the `Membership` folder's `Web.config` file so as to allow anonymous users access to the `EnhancedCreateUserWizard.aspx` page.

Summary

In this tutorial we looked at techniques for modeling data that is related to user accounts within the Membership framework. In particular, we looked at modeling entities that share a one-to-many relationship with user accounts as well as data that shares a one-to-one relationship. Furthermore, we saw how this related information could be displayed, inserted, and updated, with some examples using the SqlDataSource control and others using ADO.NET code.

This tutorial completes our look at user accounts. Starting with the next tutorial we will turn our attention to roles. Over the next several tutorials we will look at the Roles framework, see how to create new roles, how to assign roles to users, how to determine what roles a user belongs to, and how to apply role-based authorization.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Accessing and Updating Data in ASP.NET 2.0
- ASP.NET 2.0 Wizard Control
- Creating a Step-by-Step User Interface with the ASP.NET 2.0 Wizard Control
- Creating Custom DataSource Control Parameters
- Customizing the CreateUserWizard Control
- DetailsView Control QuickStarts
- Displaying Data with the ListView Control
- Dissecting the Validation Controls in ASP.NET 2.0
- Editing Insert and Deleting Data
- Form Validation in ASP.NET
- Gathering Custom User Registration Information
- Profiles in ASP.NET 2.0
- The asp:ListView Control
- User Profiles QuickStart

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.

PREVIOUS

ASP.NET 3.5 - Roles

2/20/2020 • 2 minutes to read • Edit Online

This tutorial series describes how to create and delete roles, how to assign users to roles, and how to authorize by role.

- Creating and Managing Roles (C#)
- Assigning Roles to Users (C#)
- Role-Based Authorization (C#)
- Creating and Managing Roles (VB)
- Assigning Roles to Users (VB)
- Role-Based Authorization (VB)

Creating and Managing Roles (C#)

2/20/2020 • 20 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

This tutorial examines the steps necessary for configuring the Roles framework. Following that, we will build web pages to create and delete roles.

Introduction

In the *User-Based Authorization* tutorial we looked at using URL authorization to restrict certain users from a set of pages and explored declarative and programmatic techniques for adjusting an ASP.NET page's functionality based on the visiting user. Granting permission for page access or functionality on a user-by-user basis, however, can become a maintenance nightmare in scenarios where there are many user accounts or when users' privileges change often. Any time a user gains or loses authorization to perform a particular task, the administrator needs to update the appropriate URL authorization rules, declarative markup, and code.

It usually helps to classify users into groups or *roles* and then to apply permissions on a role-by-role basis. For example, most web applications have a certain set of pages or tasks that are reserved only for administrative users. Using the techniques learned in the *User-Based Authorization* tutorial, we would add the appropriate URL authorization rules, declarative markup, and code to allow the specified user accounts to perform administrative tasks. But if a new administrator was added or if an existing administrator needed to have her administration rights revoked, we would have to return and update the configuration files and web pages. With roles, however, we could create a role called Administrators and assign those trusted users to the Administrators role. Next, we would add the appropriate URL authorization rules, declarative markup, and code to allow the Administrators role to perform the various administrative tasks. With this infrastructure in place, adding new administrators to the site or removing existing ones is as simple as including or removing the user from the Administrators role. No configuration, declarative markup, or code changes are necessary.

ASP.NET offers a Roles framework for defining roles and associating them with user accounts. With the Roles framework we can create and delete roles, add users to or remove users from a role, determine the set of users that belong to a particular role, and tell whether a user belongs to a particular role. Once the Roles framework has been configured, we can limit access to pages on a role-by-role basis through URL authorization rules and show or hide additional information or functionality on a page based on the currently logged on user's roles.

This tutorial examines the steps necessary for configuring the Roles framework. Following that, we will build web pages to create and delete roles. In the *Assigning Roles to Users* tutorial we will look at how to add and remove users from roles. And in the *Role-Based Authorization* tutorial we will see how to limit access to pages on a role-by-role basis along with how to adjust page functionality depending on the visiting user's role. Let's get started!

Step 1: Adding New ASP.NET Pages

In this tutorial and the next two we will be examining various roles-related functions and capabilities. We will need a series of ASP.NET pages to implement the topics examined throughout these tutorials. Let's create these pages and update the site map.

Start by creating a new folder in the project named `Roles`. Next, add four new ASP.NET pages to the `Roles` folder, linking each page with the `Site.master` master page. Name the pages:

- `ManageRoles.aspx`
- `UsersAndRoles.aspx`
- `CreateUserWizardWithRoles.aspx`
- `RoleBasedAuthorization.aspx`

At this point your project's Solution Explorer should look similar to the screen shot shown in Figure 1.

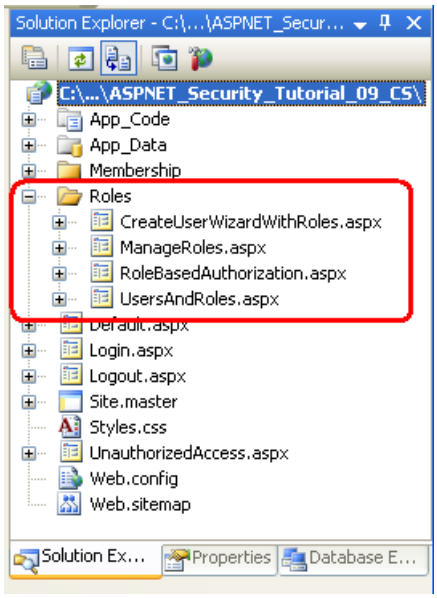


Figure 1: Four New Pages Have Been Added to the `Roles` Folder (Click to view full-size image)

Each page should, at this point, have the two Content controls, one for each of the master page's ContentPlaceHolders: `MainContent` and `LoginContent`.

```
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent"Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="LoginContent"Runat="Server">
</asp:Content>
```

Recall that the `LoginContent` ContentPlaceHolder's default markup displays a link to log on or log off the site, depending on whether the user is authenticated. The presence of the `Content2` Content control in the ASP.NET page, however, overrides the master page's default markup. As we discussed in *An Overview of Forms Authentication* tutorial, overriding the default markup is useful in pages where we do not want to display login-related options in the left column.

For these four pages, however, we want to show the master page's default markup for the `LoginContent` ContentPlaceHolder. Therefore, remove the declarative markup for the `Content2` Content control. After doing so, each of the four page's markup should contain just one Content control.

Finally, let's update the site map (`Web.sitemap`) to include these new web pages. Add the following XML after the `<siteMapNode>` we added for the Membership tutorials.

```
<siteMapNode title="Roles">
 <siteMapNode url="~/Roles/ManageRoles.aspx" title="Manage Roles"/>
 <siteMapNode url="~/Roles/UsersAndRoles.aspx" title="Users and Roles" />
 <siteMapNode url="~/Roles/CreateUserWizardWithRoles.aspx" title="Create Account (with Roles)" />
 <siteMapNode url="~/Roles/RoleBasedAuthorization.aspx" title="Role-Based Authorization" />
</siteMapNode>
```

With the site map updated, visit the site through a browser. As Figure 2 shows, the navigation on the left now includes items for the Roles tutorials.

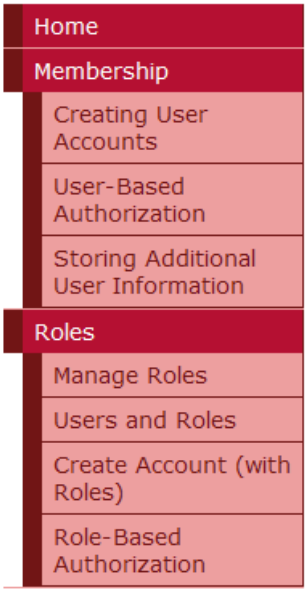


Figure 2: Four New Pages Have Been Added to the `Roles` Folder (Click to view full-size image)

Step 2: Specifying and Configuring the Roles Framework Provider

Like the Membership framework, the Roles framework is built atop the provider model. As discussed in the *Security Basics and ASP.NET Support* tutorial, the .NET Framework ships with three built-in Roles providers: `AuthorizationStoreRoleProvider`, `WindowsTokenRoleProvider`, and `SqlRoleProvider`. This tutorial series focuses on the `SqlRoleProvider`, which uses a Microsoft SQL Server database as the role store.

Underneath the covers the Roles framework and `SqlRoleProvider` work just like the Membership framework and `SqlMembershipProvider`. The .NET Framework contains a `Roles` class that serves as the API to the Roles framework. The `Roles` class has static methods like `CreateRole`, `DeleteRole`, `GetAllRoles`, `AddUserToRole`, `IsUserInRole`, and so forth. When one of these methods is invoked, the `Roles` class delegates the call to the configured provider. The `SqlRoleProvider` works with the role-specific tables (`aspnet_Roles` and `aspnet_UsersInRoles`) in response.

In order to use the `SqlRoleProvider` provider in our application, we need to specify what database to use as the store. The `SqlRoleProvider` expects the specified role store to have certain database tables, views, and stored procedures. These requisite database objects can be added using the `aspnet_regsql.exe` tool. At this point we already have a database with the schema necessary for the `SqlRoleProvider`. Back in the *Creating the Membership Schema in SQL Server* tutorial we created a database named `SecurityTutorials.mdf` and used `aspnet_regsql.exe` to add the application services, which included the database objects required by the `SqlRoleProvider`. Therefore we just need to tell the Roles framework to enable role support and to use the `SqlRoleProvider` with the `SecurityTutorials.mdf` database as the role store.

The Roles framework is configured via the `<roleManager>` element in the application's `Web.config` file. By default, role support is disabled. To enable it, you must set the `<roleManager>` element's `enabled` attribute to `true` like so:

```
<?xml version="1.0"?>
<configuration>
 <system.web>
 ... Additional configuration markup removed for brevity ...

 <roleManager enabled="true" />
 <system.web>
</configuration>
```

By default, all web applications have a Roles provider named `AspNetSqlRoleProvider` of type `SqlRoleProvider`. This default provider is registered in `machine.config` (located at `%WINDIR%\Microsoft.Net\Framework\v2.0.50727\CONFIG`):

```
<roleManager>
 <providers>
 <add name="AspNetSqlRoleProvider"
 connectionStringName="LocalSqlServer"
 applicationName="/"
 type="System.Web.Security.SqlRoleProvider,
 System.Web, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"/>
 </providers>
</roleManager>
```

The provider's `connectionStringName` attribute specifies the role store that is used. The `AspNetSqlRoleProvider` provider sets this attribute to `LocalSqlServer`, which is also defined in `machine.config` and points, by default, to a SQL Server 2005 Express Edition database in the `App_Data` folder named `aspnet.mdf`.

Consequently, if we simply enable the Roles framework without specifying any provider information in our application's `Web.config` file, the application uses the default registered Roles provider, `AspNetSqlRoleProvider`. If the `~/App_Data/aspnet.mdf` database does not exist, the ASP.NET runtime will automatically create it and add the application services schema. However, we don't want to use the `aspnet.mdf` database; rather, we want to use the `SecurityTutorials.mdf` database that we have already created and added the application services schema to. This modification can be accomplished in one of two ways:

- **Specify a value for the `LocalSqlServer` connection string name in `Web.config`.** By overwriting the `LocalSqlServer` connection string name value in `Web.config`, we can use the default registered Roles provider (`AspNetSqlRoleProvider`) and have it correctly work with the `SecurityTutorials.mdf` database. For more information on this technique, see Scott Guthrie's blog post, Configuring ASP.NET 2.0 Application Services to Use SQL Server 2000 or SQL Server 2005.
- **Add a new registered provider of type `SqlRoleProvider` and configure its `connectionStringName` setting to point to the `SecurityTutorials.mdf` database.** This is the approach I recommended and used in the *Creating the Membership Schema in SQL Server* tutorial, and it is the approach I will use in this tutorial as well.

Add the following Roles configuration markup to the `Web.config` file. This markup registers a new provider named `SecurityTutorialsSqlRoleProvider`.

```
<?xml version="1.0"?>
<configuration>
 <connectionStrings>
 <add name="SecurityTutorialsConnectionString"
 connectionString="..."/>
 </connectionStrings>

 <system.web>
 ... Additional configuration markup removed for brevity ...

 <roleManager enabled="true" defaultProvider="SecurityTutorialsSqlRoleProvider">
 <providers>
 <add name="SecurityTutorialsSqlRoleProvider"
 type="System.Web.Security.SqlRoleProvider"
 applicationName="SecurityTutorials"
 connectionStringName="SecurityTutorialsConnectionString" />
 </providers>
 </roleManager>
 <system.web>
</configuration>
```

The above markup defines the `SecurityTutorialsSqlRoleProvider` as the default provider (via the `defaultProvider` attribute in the `<roleManager>` element). It also sets the `SecurityTutorialsSqlRoleProvider`'s `applicationName` setting to `SecurityTutorials`, which is the same `applicationName` setting used by the Membership provider (`SecurityTutorialsSqlMembershipProvider`). While not shown here, the `<add>` element for the `SqlRoleProvider` may also contain a `commandTimeout` attribute to specify the database timeout duration, in seconds. The default value is 30.

With this configuration markup in place, we are ready to start using role functionality within our application.

> **NOTE**
> The above configuration markup illustrates using the < `roleManager` > element's `enabled` and `defaultProvider` attributes. There are a number of other attributes that affect how the Roles framework associates role information on a user-by-user basis. We will examine these settings in the *Role-Based Authorization* tutorial.

Step 3: Examining the Roles API

The Roles framework's functionality is exposed via the `Roles` class, which contains thirteen static methods for performing role-based operations. When we look at creating and deleting roles in Steps 4 and 6 we will use the `CreateRole` and `DeleteRole` methods, which add or remove a role from the system.

To get a list of all of the roles in the system, use the `GetAllRoles` method (see Step 5). The `RoleExists` method returns a Boolean value indicating whether a specified role exists.

In the next tutorial we will examine how to associate users with roles. The `Roles` class's `AddUserToRole`, `AddUserToRoles`, `AddUsersToRole`, and `AddUsersToRoles` methods add one or more users to one or more roles. To remove users from roles, use the `RemoveUserFromRole`, `RemoveUserFromRoles`, `RemoveUsersFromRole`, or `RemoveUsersFromRoles` methods.

In the *Role-Based Authorization* tutorial we will look at ways to programmatically show or hide functionality based on the currently logged in user's role. To accomplish this, we can use the `Role` class's `FindUsersInRole`, `GetRolesForUser`, `GetUsersInRole`, or `IsUserInRole` methods.

> **NOTE**
> Keep in mind that any time one of these methods is invoked, the `Roles` class delegates the call to the configured provider. In our case, this means that the call is being sent to the `SqlRoleProvider`. The `SqlRoleProvider` then performs the appropriate database operation based on the invoked method. For example, the code `Roles.CreateRole("Administrators")` results in the `SqlRoleProvider` executing the `aspnet_Roles_CreateRole` stored procedure, which inserts a new record into the `aspnet_Roles` table named Administrators .

The remainder of this tutorial looks at using the `Roles` class's `CreateRole`, `GetAllRoles`, and `DeleteRole` methods to manage the roles in the system.

Step 4: Creating New Roles

Roles offer a way to arbitrarily group users, and most commonly this grouping is used for a more convenient way to apply authorization rules. But in order to use roles as an authorization mechanism we first need to define what roles exist in the application. Unfortunately, ASP.NET does not include a CreateRoleWizard control. In order to add new roles we need to create a suitable user interface and invoke the Roles API ourselves. The good news is that this is very easy to accomplish.

NOTE

While there is no CreateRoleWizard Web control, there is the ASP.NET Web Site Administration Tool, which is a local ASP.NET application designed to assist with viewing and managing your web application's configuration. However, I am not a big fan of the ASP.NET Web Site Administration Tool for two reasons. First, it is a bit buggy and the user experience leaves a lot to be desired. Second, the ASP.NET Web Site Administration Tool is designed to only work locally, meaning that you will have to build your own role management web pages if you need to manage roles on a live site remotely. For these two reasons, this tutorial and the next will focus on building the necessary role management tools in a web page rather than relying on the ASP.NET Web Site Administration Tool.

Open the `ManageRoles.aspx` page in the `Roles` folder and add a TextBox and a Button Web control to the page. Set the TextBox control's `ID` property to `RoleName` and the Button's `ID` and `Text` properties to `CreateRoleButton` and Create Role, respectively. At this point, your page's declarative markup should look similar to the following:

```
<b>Create a New Role: </b>
<asp:TextBox ID="RoleName" runat="server"></asp:TextBox>
<br />
<asp:Button ID="CreateRoleButton" runat="server" Text="Create Role" />
```

Next, double-click the `CreateRoleButton` Button control in the Designer to create a `Click` event handler and then add the following code:

```
protected void CreateRoleButton_Click(object sender, EventArgs e)
{
    string newRoleName = RoleName.Text.Trim();

    if (!Roles.RoleExists(newRoleName))
        // Create the role
        Roles.CreateRole(newRoleName);

    RoleName.Text = string.Empty;
}
```

The above code starts by assigning the trimmed role name entered in the `RoleName` TextBox to the `newRoleName` variable. Next, the `Roles` class's `RoleExists` method is called to determine if the role `newRoleName` already exists in the system. If the role does not exist, it is created via a call to the `CreateRole` method. If the `CreateRole` method is passed a role name that already exists in the system, a `ProviderException` exception is thrown. This is why the code first checks to ensure that the role does not already exist in the system before calling `CreateRole`. The `Click` event handler concludes by clearing out the `RoleName` TextBox's `Text` property.

NOTE

You may be wondering what will happen if the user doesn't enter any value into the `RoleName` TextBox. If the value passed into the `CreateRole` method is `null` or an empty string, an exception is raised. Likewise, if the role name contains a comma an exception is raised. Consequently, the page should contain validation controls to ensure that the user enters a role and that it does not contain any commas. I leave as an exercise for the reader.

Let's create a role named Administrators. Visit the `ManageRoles.aspx` page through a browser, type in Administrators into the textbox (see Figure 3), and then click the Create Role button.

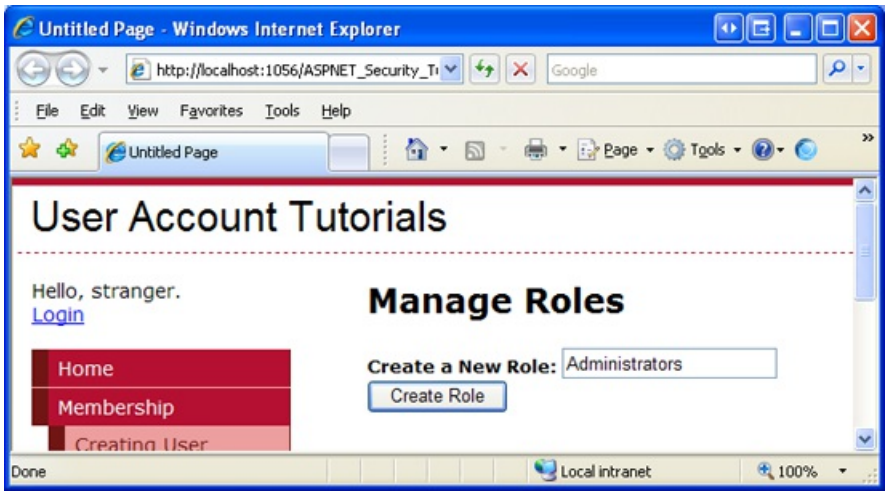


Figure 3: Create an Administrators Role (Click to view full-size image)

What happens? A postback occurs, but there's no visual cue that the role has actually been added to the system. We will update this page in Step 5 to include visual feedback. For now, however, you can verify that the role was created by going to the `SecurityTutorials.mdf` database and displaying the data from the `aspnet_Roles` table. As Figure 4 shows, the `aspnet_Roles` table contains a record for the just-added Administrators roles.

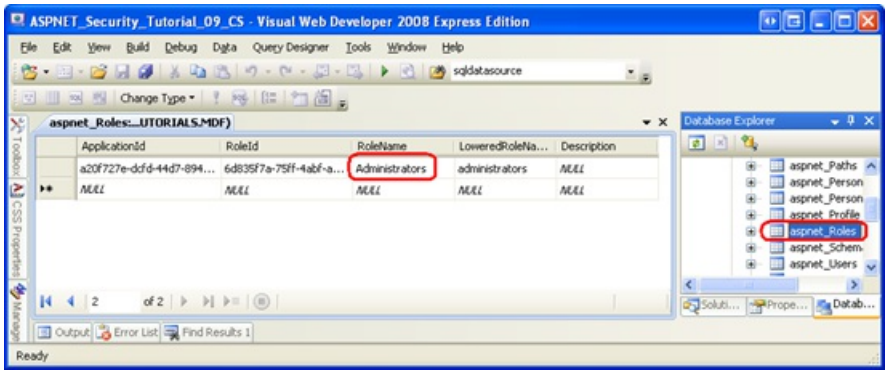


Figure 4: The `aspnet_Roles` Table has a Row for the Administrators (Click to view full-size image)

Step 5: Displaying the Roles in the System

Let's augment the `ManageRoles.aspx` page to include a list of the current roles in the system. To accomplish this, add a GridView control to the page and set its `ID` property to `RoleList`. Next, add a method to the page's code-behind class named `DisplayRolesInGrid` using the following code:

```
private void DisplayRolesInGrid()
{
    RoleList.DataSource = Roles.GetAllRoles();
    RoleList.DataBind();
}
```

The `Roles` class's `GetAllRoles` method returns all of the roles in the system as an array of strings. This string array is then bound to the GridView. In order to bind the list of roles to the GridView when the page is first loaded, we need to call the `DisplayRolesInGrid` method from the page's `Page_Load` event handler. The following code calls this method when the page is first visited, but not on subsequent postbacks.

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    DisplayRolesInGrid();
}
```

With this code in place, visit the page through a browser. As Figure 5 shows, you should see a grid with a single column labeled Item. The grid includes a row for the Administrators role we added in Step 4.

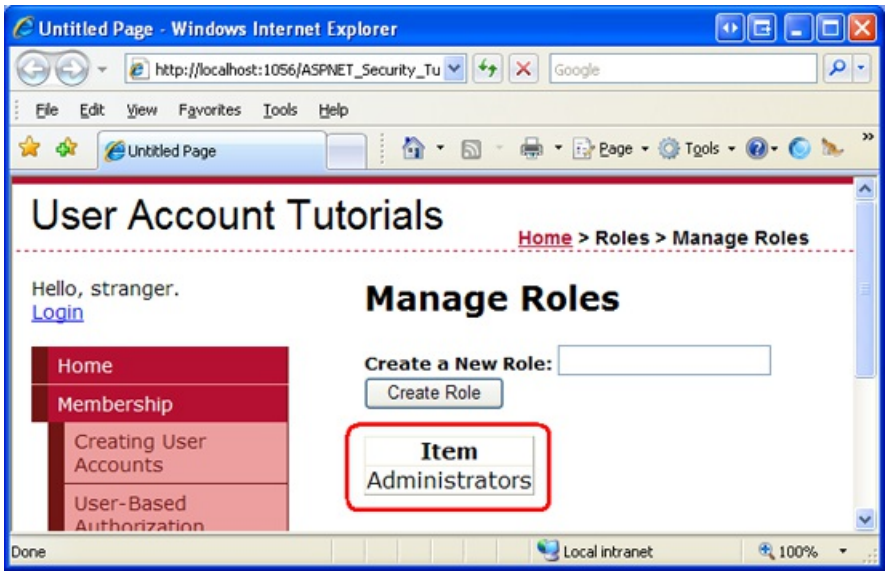


Figure 5: The GridView Displays the Roles in a Single Column (Click to view full-size image)

The GridView displays a lone column labeled Item because the GridView's `AutoGenerateColumns` property is set to True (the default), which causes the GridView to automatically create a column for each property in its `DataSource`. An array has a single property that represents the elements in the array, hence the single column in the GridView.

When displaying data with a GridView, I prefer to explicitly define my columns rather than have them implicitly generated by the GridView. By explicitly defining the columns it is much easier to format the data, rearrange the columns, and perform other common tasks. Therefore, let's update the GridView's declarative markup so that its columns are explicitly defined.

Start by setting the GridView's `AutoGenerateColumns` property to False. Next, add a TemplateField to the grid, set its `HeaderText` property to Roles, and configure its `ItemTemplate` so that it displays the contents of the array. To accomplish this, add a Label Web control named `RoleNameLabel` to the `ItemTemplate` and bind its `Text` property to `Container.DataItem`.

These properties and the `ItemTemplate`'s contents can be set declaratively or through the GridView's Fields dialog box and Edit Templates interface. To reach the Fields dialog box, click the Edit Columns link in the GridView's Smart Tag. Next, uncheck the Auto-generate fields checkbox to set the `AutoGenerateColumns` property to False, and add a TemplateField to the GridView, setting its `HeaderText` property to Role. To define the `ItemTemplate`'s contents, choose the Edit Templates option from the GridView's Smart Tag. Drag a Label Web control onto the `ItemTemplate`, set its `ID` property to `RoleNameLabel`, and configure its databinding settings so that its `Text` property is bound to `Container.DataItem`.

Regardless of what approach you use, the GridView's resulting declarative markup should look similar to the following when you are done.

```
<asp:GridView ID="RoleList" runat="server" AutoGenerateColumns="false">
 <Columns>
 <asp:TemplateField HeaderText="Role">
 <ItemTemplate>
 <asp:Label runat="server" ID="RoleNameLabel" Text='<%# Container.DataItem %>' />
 </ItemTemplate>
 </asp:TemplateField>
 </Columns>
</asp:GridView>
```

NOTE

The array's contents are displayed using the databinding syntax `<%# Container.DataItem %>`. A thorough description of why this syntax is used when displaying the contents of an array bound to the GridView is beyond the scope of this tutorial. For more information on this matter, refer to Binding a Scalar Array to a Data Web Control.

Currently, the `RoleList` GridView is only bound to the list of roles when the page is first visited. We need to refresh the grid whenever a new role is added. To accomplish this, update the `CreateRoleButton` Button's `Click` event handler so that it calls the `DisplayRolesInGrid` method if a new role is created.

```
protected void CreateRoleButton_Click(object sender, EventArgs e)
{
    string newRoleName = RoleName.Text.Trim();

    if (!Roles.RoleExists(newRoleName))
    {
        // Create the role
        Roles.CreateRole(newRoleName);

        // Refresh the RoleList Grid
        DisplayRolesInGrid();
    }

    RoleName.Text = string.Empty;
}
```

Now when the user adds a new role the `RoleList` GridView shows the just-added role on postback, providing visual feedback that the role was successfully created. To illustrate this, visit the `ManageRoles.aspx` page through a browser and add a role named Supervisors. Upon clicking the Create Role button, a postback will ensue and the grid will update to include Administrators as well as the new role, Supervisors.

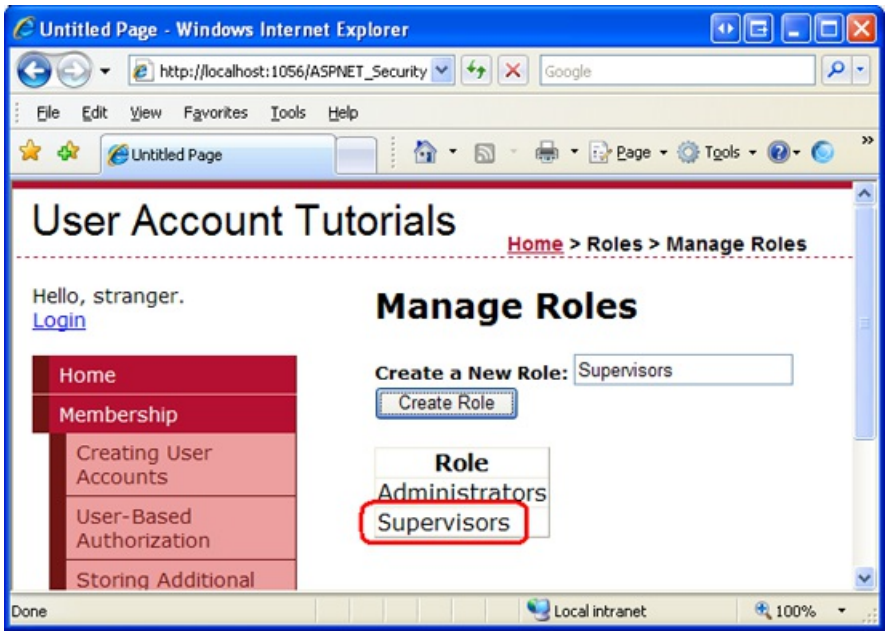


Figure 6: The Supervisors Role has Been Added (Click to view full-size image)

Step 6: Deleting Roles

At this point a user can create a new role and view all existing roles from the `ManageRoles.aspx` page. Let's allow users to also delete roles. The `Roles.DeleteRole` method has two overloads:

- `DeleteRole(roleName)` - deletes the role *roleName*. An exception is thrown if the role contains one or more members.
- `DeleteRole(roleName, throwOnPopulatedRole)` - deletes the role *roleName*. If *throwOnPopulateRole* is `true`, then an exception is thrown if the role contains one or more members. If *throwOnPopulateRole* is `false`, then the role is deleted whether it contains any members or not. Internally, the `DeleteRole(roleName)` method calls `DeleteRole(roleName, true)`.

The `DeleteRole` method will also throw an exception if *roleName* is `null` or an empty string or if *roleName* contains a comma. If *roleName* does not exist in the system, `DeleteRole` fails silently, without raising an exception.

Let's augment the GridView in `ManageRoles.aspx` to include a Delete button that, when clicked, deletes the selected role. Start by adding a Delete button to the GridView by going to the Fields dialog box and adding a Delete button, which is located under the CommandField option. Make the Delete button the far left column and set its `DeleteText` property to Delete Role .

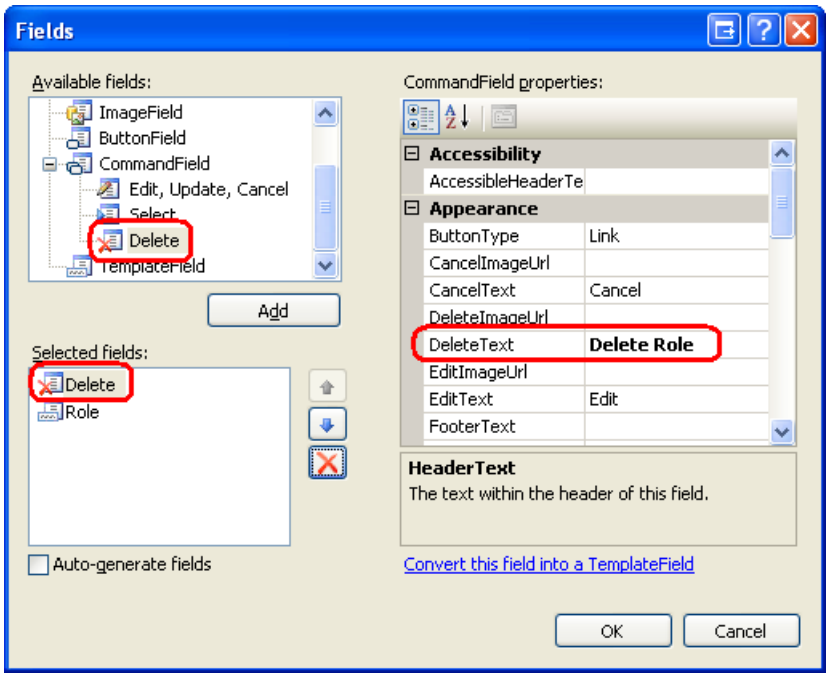


Figure 7: Add a Delete Button to the `RoleList` GridView (Click to view full-size image)

After adding the Delete button, your GridView's declarative markup should look similar to the following:

```
<asp:GridView ID="RoleList" runat="server" AutoGenerateColumns="False">
 <Columns>
 <asp:CommandField DeleteText="Delete Role" ShowDeleteButton="True"/>
 <asp:TemplateField HeaderText="Role">
 <ItemTemplate>
 <asp:Label runat="server" ID="RoleNameLabel" Text='<%# Container.DataItem %>' />
 </ItemTemplate>
 </asp:TemplateField>
 </Columns>
</asp:GridView>
```

Next, create an event handler for the GridView's `RowDeleting` event. This is the event that is raised on postback when the Delete Role button is clicked. Add the following code to the event handler.

```
protected void RoleList_RowDeleting(object sender, GridViewDeleteEventArgs e)
{
    // Get the RoleNameLabel
    Label RoleNameLabel = RoleList.Rows[e.RowIndex].FindControl("RoleNameLabel") as Label;

    // Delete the role
    Roles.DeleteRole(RoleNameLabel.Text, false);

    // Rebind the data to the RoleList grid
    DisplayRolesInGrid();
}
```

The code starts by programmatically referencing the `RoleNameLabel` Web control in the row whose Delete Role button was clicked. The `Roles.DeleteRole` method is then invoked, passing in the `Text` of the `RoleNameLabel` and `false`, thereby deleting the role regardless of whether there are any users associated with the role. Finally, the `RoleList` GridView is refreshed so that the just-deleted role no longer appears in the grid.

NOTE

The Delete Role button does not require any sort of confirmation from the user before deleting the role. One of the easiest ways to confirm an action is through a client-side confirm dialog box. For more information on this technique, see Adding Client-Side Confirmation When Deleting.

Summary

Many web applications have certain authorization rules or page-level functionality that is only available to certain classes of users. For example, there may be a set of web pages that only administrators can access. Rather than defining these authorization rules on a user-by-user basis, oftentimes it is more useful to define the rules based on a role. That is, rather than explicitly allowing users Scott and Jisun to access the administrative web pages, a more maintainable approach is to permit members of the Administrators role to access these pages, and then to denote Scott and Jisun as users belonging to the Administrators role.

The Roles framework makes it easy to create and manage roles. In this tutorial we examined how to configure the Roles framework to use the `SqlRoleProvider`, which uses a Microsoft SQL Server database as the role store. We also created a web page that lists the existing roles in the system and allows for new roles to be created and existing ones to be deleted. In subsequent tutorials we will see how to assign users to roles and how to apply role-based authorization.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Examining ASP.NET 2.0's Membership, Roles, and Profile
- How To: Use Role Manager in ASP.NET 2.0
- Role Providers
- Rolling Your Own Website Administration Tool
- Technical documentation for the `<roleManager>` Element
- Using the Membership and Role Manager APIs

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial include Alicja Maziarz, Suchi Banerjee, and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

NEXT

Assigning Roles to Users (C#)

2/20/2020 • 28 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

In this tutorial we will build two ASP.NET pages to assist with managing what users belong to what roles. The first page will include facilities to see what users belong to a given role, what roles a particular user belongs to, and the ability to assign or remove a particular user from a particular role. In the second page we will augment the CreateUserWizard control so that it includes a step to specify what roles the newly created user belongs to. This is useful in scenarios where an administrator is able to create new user accounts.

Introduction

The previous tutorial examined the Roles framework and the `SqlRoleProvider`; we saw how to use the `Roles` class to create, retrieve, and delete roles. In addition to creating and deleting roles, we need to be able to assign or remove users from a role. Unfortunately, ASP.NET does not ship with any Web controls for managing what users belong to what roles. Instead, we must create our own ASP.NET pages to manage these associations. The good news is that adding and removing users to roles is quite easy. The `Roles` class contains a number of methods for adding one or more users to one or more roles.

In this tutorial we will build two ASP.NET pages to assist with managing what users belong to what roles. The first page will include facilities to see what users belong to a given role, what roles a particular user belongs to, and the ability to assign or remove a particular user from a particular role. In the second page we will augment the CreateUserWizard control so that it includes a step to specify what roles the newly created user belongs to. This is useful in scenarios where an administrator is able to create new user accounts.

Let's get started!

Listing What Users Belong To What Roles

The first order of business for this tutorial is to create a web page from which users can be assigned to roles. Before we concern ourselves with how to assign users to roles, let's first concentrate on how to determine what users belong to what roles. There are two ways to display this information: "by role" or "by user." We could allow the visitor to select a role and then show them all of the users that belong to the role (the "by role" display), or we could prompt the visitor to select a user and then show them the roles assigned to that user (the "by user" display).

The "by role" view is useful in circumstances where the visitor wants to know the set of users that belong to a particular role; the "by user" view is ideal when the visitor needs to know a particular user's role(s). Let's have our page include both "by role" and "by user" interfaces.

We will start with creating the "by user" interface. This interface will consist of a drop-down list and a list of checkboxes. The drop-down list will be populated with the set of users in the system; the checkboxes will enumerate the roles. Selecting a user from the drop-down list will check those roles the user belongs to. The person visiting the page can then check or uncheck the checkboxes to add or remove the selected user from the corresponding roles.

> **NOTE**
> Using a drop-down list to list the user accounts is not an ideal choice for websites where there may be hundreds of user accounts. A drop-down list is designed to allow a user to pick one item from a relatively short list of options. It quickly becomes unwieldy as the number of list items grows. If you are building a website that will have potentially large numbers of user accounts, you may want to consider using an alternative user interface, such as a pageable GridView or a filterable interface that lists prompts the visitor to choose a letter and then only shows those users whose username starts with the selected letter.

Step 1: Building the "By User" User Interface

Open the `UsersAndRoles.aspx` page. At the top of the page, add a Label Web control named `ActionStatus` and clear out its `Text` property. We will use this Label to provide feedback on the actions performed, displaying messages like, "User Tito has been added to the Administrators role," or "User Jisun has been removed from the Supervisors role." In order to make these messages stand out, set the Label's `CssClass` property to "Important".

```
<p align="center">

     <asp:Label ID="ActionStatus" runat="server" CssClass="Important"></asp:Label>
</p>
```

Next, add the following CSS class definition to the `Styles.css` stylesheet:

```
.Important
{
     font-size: large;
     color: Red;
}
```

This CSS definition instructs the browser to display the Label using a large, red font. Figure 1 shows this effect through the Visual Studio Designer.

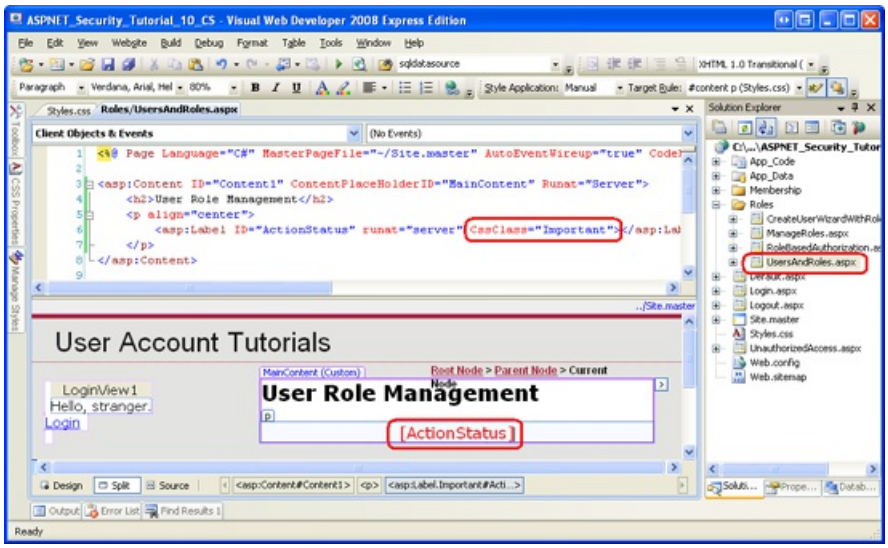


Figure 1: The Label's `CssClass` Property Results in a Large, Red Font (Click to view full-size image)

Next, add a DropDownList to the page, set its `ID` property to `UserList`, and set its `AutoPostBack` property to True. We will use this DropDownList to list all of the users in the system. This DropDownList will be bound to a collection of MembershipUser objects. Because we want the DropDownList to display the UserName property of the MembershipUser object (and use it as the value of the list items), set the DropDownList's `DataTextField` and `DataValueField` properties to "UserName".

Underneath the DropDownList, add a Repeater named `UsersRoleList`. This Repeater will list all of the roles in the system as a series of checkboxes. Define the Repeater's `ItemTemplate` using the following declarative markup:

```
<asp:Repeater ID="UsersRoleList" runat="server">
     <ItemTemplate>
          <asp:CheckBox runat="server" ID="RoleCheckBox" AutoPostBack="true"

               Text='<%# Container.DataItem %>' />
          <br />
     </ItemTemplate>
</asp:Repeater>
```

The `ItemTemplate` markup includes a single CheckBox Web control named `RoleCheckBox`. The CheckBox's `AutoPostBack` property is set to True and the `Text` property is bound to `Container.DataItem`. The reason the databinding syntax is simply `Container.DataItem` is because the Roles framework returns the list of role names as a string array, and it is this string array that we will be binding to the Repeater. A thorough description of why this syntax is used to display the contents of an array bound to a data Web control is beyond the scope of this tutorial. For more information on this matter, refer to Binding a Scalar Array to a Data Web Control.

At this point your "by user" interface's declarative markup should look similar to the following:

```
<h3>Manage Roles By User</h3>

<p>
     <b>Select a User:</b>
     <asp:DropDownList ID="UserList" runat="server" AutoPostBack="True"
          DataTextField="UserName" DataValueField="UserName">

     </asp:DropDownList>
</p>
<p>
     <asp:Repeater ID="UsersRoleList" runat="server">
          <ItemTemplate>
               <asp:CheckBox runat="server" ID="RoleCheckBox" AutoPostBack="true"

                    Text='<%# Container.DataItem %>' />
               <br />
          </ItemTemplate>
     </asp:Repeater>
</p>
```

We are now ready to write the code to bind the set of user accounts to the DropDownList and the set of roles to the Repeater. In the page's code-behind class, add a method named `BindUsersToUserList` and another named `BindRolesList`, using the following code:

```
private void BindUsersToUserList()
{
     // Get all of the user accounts
     MembershipUserCollection users = Membership.GetAllUsers();
     UserList.DataSource = users;
     UserList.DataBind();
}

private void BindRolesToList()
{
     // Get all of the roles
     string[] roles = Roles.GetAllRoles();
     UsersRoleList.DataSource = roles;
     UsersRoleList.DataBind();
}
```

The `BindUsersToUserList` method retrieves all of the user accounts in the system via the `Membership.GetAllUsers` method. This returns a `MembershipUserCollection` object, which is a collection of `MembershipUser` instances. This collection is then bound to the `UserList` DropDownList. The `MembershipUser` instances that makeup the collection contain a variety of properties, like `UserName`, `Email`, `CreationDate`, and `IsOnline`. In order to instruct the DropDownList to display the value of the `UserName` property, ensure that the `UserList` DropDownList's `DataTextField` and `DataValueField` properties have been set to "UserName".

> **NOTE**
>
> The `Membership.GetAllUsers` method has two overloads: one that accepts no input parameters and returns all of the users, and one that takes in integer values for the page index and page size, and returns only the specified subset of the users. When there are large amounts of user accounts being displayed in a pageable user interface element, the second overload can be used to more efficiently page through the users since it returns just the precise subset of user accounts rather than all of them.

The `BindRolesToList` method starts by calling the `Roles` class's `GetAllRoles` method, which returns a string array containing the roles in the system. This string array is then bound to the Repeater.

Finally, we need to call these two methods when the page is first loaded. Add the following code to the `Page_Load` event handler:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        // Bind the users and roles
        BindUsersToUserList();
        BindRolesToList();
    }
}
```

With this code in place, take a moment to visit the page through a browser; your screen should look similar to Figure 2. All of the user accounts are populated in the drop-down list and, underneath that, each role appears as a checkbox. Because we set the `AutoPostBack` properties of the DropDownList and CheckBoxes to True, changing the selected user or checking or unchecking a role causes a postback. No action is performed, however, because we have yet to write code to handle these actions. We'll tackle these tasks in the next two sections.

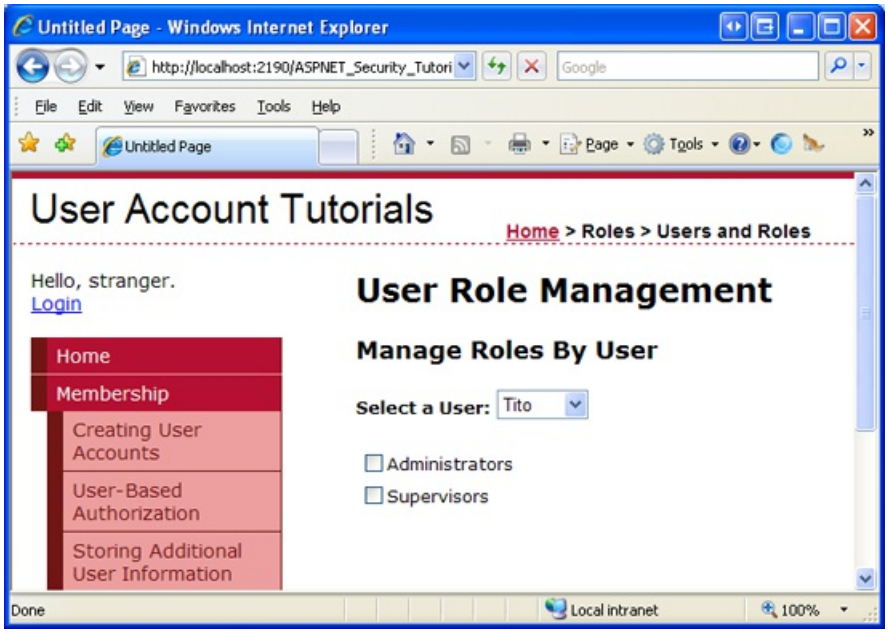


Figure 2: The Page Displays the Users and Roles (Click to view full-size image)

Checking the Roles the Selected User Belongs To

When the page is first loaded, or whenever the visitor selects a new user from the drop-down list, we need to update the `UsersRoleList`'s checkboxes so that a given role checkbox is checked only if the selected user belongs to that role. To accomplish this, create a method named `CheckRolesForSelectedUser` with the following code:

```
private void CheckRolesForSelectedUser()
{
     // Determine what roles the selected user belongs to
     string selectedUserName = UserList.SelectedValue;
     string[] selectedUsersRoles = Roles.GetRolesForUser(selectedUserName);

     // Loop through the Repeater's Items and check or uncheck the checkbox as needed

     foreach (RepeaterItem ri in UsersRoleList.Items)
     {
          // Programmatically reference the CheckBox
          CheckBox RoleCheckBox = ri.FindControl("RoleCheckBox") as CheckBox;
          // See if RoleCheckBox.Text is in selectedUsersRoles
          if (selectedUsersRoles.Contains<string>(RoleCheckBox.Text))
               RoleCheckBox.Checked = true;
          else
               RoleCheckBox.Checked = false;
     }
}
```

The above code starts by determining who the selected user is. It then uses the Roles class's `GetRolesForUser(userName)` method to return the specified user's set of roles as a string array. Next, the Repeater's items are enumerated and each item's `RoleCheckBox` CheckBox is programmatically referenced. The CheckBox is checked only if the role it corresponds to is contained within the `selectedUsersRoles` string array.

> **NOTE**
>
> The `selectedUserRoles.Contains<string>(...)` syntax will not compile if you are using ASP.NET version 2.0. The `Contains<string>` method is part of the LINQ library, which is new to ASP.NET 3.5. If you are still using ASP.NET version 2.0, use the `Array.IndexOf<string>` method instead.

The `CheckRolesForSelectedUser` method needs to be called in two cases: when the page is first loaded and whenever the `UserList` DropDownList's selected index is changed. Therefore, call this method from the `Page_Load` event handler (after the calls to `BindUsersToUserList` and `BindRolesToList`). Also, create an event handler for the DropDownList's `SelectedIndexChanged` event and call this method from there.

```
protected void Page_Load(object sender, EventArgs e)
{
     if (!Page.IsPostBack)
     {

          // Bind the users and roles
          BindUsersToUserList();
          BindRolesToList();
          // Check the selected user's roles
          CheckRolesForSelectedUser();
     }
}

...

protected void UserList_SelectedIndexChanged(object sender, EventArgs e)
{
     CheckRolesForSelectedUser();
}
```

With this code in place, you can test the page through the browser. However, since the `UsersAndRoles.aspx` page currently lacks the ability to assign users to roles, no users have roles. We will create the interface for assigning users to roles in a moment, so you can either take my word that this code works and verify that it does so later, or you can manually add users to roles by inserting records into the `aspnet_UsersInRoles` table in order to test this functionality now.

Assigning and Removing Users from Roles

When the visitor checks or unchecks a CheckBox in the `UsersRoleList` Repeater we need to add or remove the selected user from the corresponding role. The CheckBox's `AutoPostBack` property is currently set to True, which causes a postback anytime a CheckBox in the Repeater is checked or unchecked. In short, we need to create an event handler for the CheckBox's `CheckChanged` event. Since the CheckBox is in a Repeater control, we need to manually add the event handler plumbing. Start by adding the event handler to the code-behind class as a `protected` method, like so:

```
protected void RoleCheckBox_CheckChanged(object sender, EventArgs e)
{

}
```

We will return to write the code for this event handler in a moment. But first let's complete the event handling plumbing. From the CheckBox within the Repeater's `ItemTemplate`, add `OnCheckedChanged="RoleCheckBox_CheckChanged"`. This syntax wires the `RoleCheckBox_CheckChanged` event handler to the `RoleCheckBox`'s `CheckedChanged` event.

```
<asp:CheckBox runat="server" ID="RoleCheckBox"
     AutoPostBack="true"
     Text='<%# Container.DataItem %>'
     OnCheckedChanged="RoleCheckBox_CheckChanged" />
```

Our final task is to complete the `RoleCheckBox_CheckChanged` event handler. We need to start by referencing the CheckBox control that raised the event because this CheckBox instance tells us what role was checked or unchecked via its `Text` and `Checked` properties. Using this information along with the UserName of the selected user, we add or remove the user from the role via the `Roles` class's `AddUserToRole` or `RemoveUserFromRole` method.

```
protected void RoleCheckBox_CheckChanged(object sender, EventArgs e)
{
     // Reference the CheckBox that raised this event
     CheckBox RoleCheckBox = sender as CheckBox;

     // Get the currently selected user and role
     string selectedUserName = UserList.SelectedValue;

     string roleName = RoleCheckBox.Text;

     // Determine if we need to add or remove the user from this role
     if (RoleCheckBox.Checked)
     {
          // Add the user to the role
          Roles.AddUserToRole(selectedUserName, roleName);
          // Display a status message
          ActionStatus.Text = string.Format("User {0} was added to role {1}.", selectedUserName, roleName);
     }
     else
     {
          // Remove the user from the role
          Roles.RemoveUserFromRole(selectedUserName, roleName);
          // Display a status message
          ActionStatus.Text = string.Format("User {0} was removed from role {1}.", selectedUserName,
roleName);

     }
}
```

The above code starts by programmatically referencing the CheckBox that raised the event, which is available via the `sender` input parameter. If the CheckBox is checked, the selected user is added to the specified role, otherwise they are removed from the role. In either case, the `ActionStatus` Label displays a message summarizing the action just performed.

Take a moment to test out this page through a browser. Select user Tito and then add Tito to both the Administrators and Supervisors roles.

Figure 3: Tito Has Been Added to the Administrators and Supervisors Roles (Click to view full-size image)

Next, select user Bruce from the drop-down list. There is a postback and the Repeater's CheckBoxes are updated via the `CheckRolesForSelectedUser`. Since Bruce does not yet belong to any roles, the two checkboxes are unchecked. Next, add Bruce to the Supervisors role.

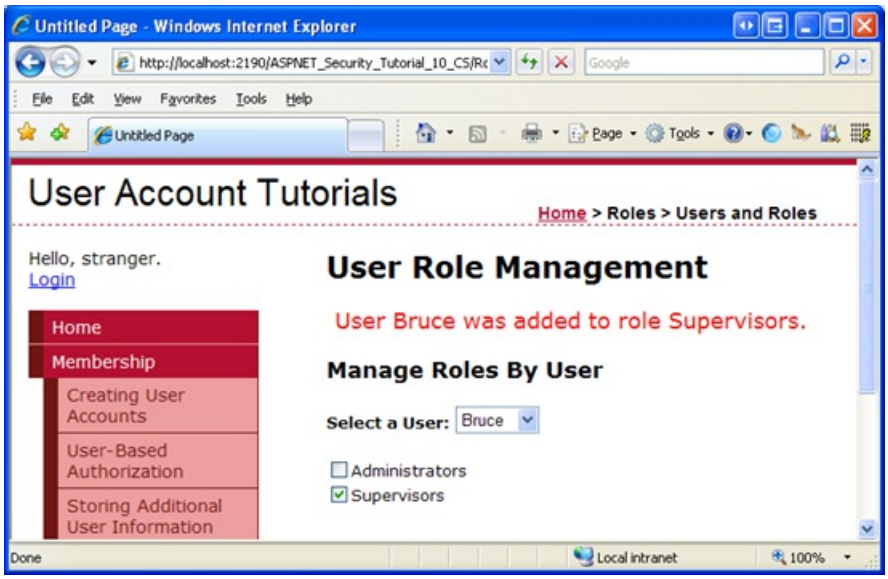


Figure 4: Bruce Has Been Added to the Supervisors Role (Click to view full-size image)

To further verify the functionality of the `CheckRolesForSelectedUser` method, select a user other than Tito or Bruce. Note how the checkboxes are automatically unchecked, denoting that they do not belong to any roles. Return to Tito. Both the Administrators and Supervisors checkboxes should be checked.

Step 2: Building the "By Roles" User Interface

At this point we have completed the "by users" interface and are ready to start tackling the "by roles" interface. The "by roles" interface prompts the user to select a role from a drop-down list and then displays the set of users that belong to that role in a GridView.

Add another DropDownList control to the `UsersAndRoles.aspx` page. Place this one beneath the Repeater control, name it `RoleList`, and set its `AutoPostBack` property to True. Underneath that, add a GridView and name it `RolesUserList`. This GridView will list the users that belong to the selected role. Set the GridView's `AutoGenerateColumns` property to False, add a TemplateField to the grid's `Columns` collection, and set its `HeaderText` property to "Users". Define the TemplateField's `ItemTemplate` so that it displays the value of the databinding expression `Container.DataItem` in the `Text` property of a Label named `UserNameLabel`.

After adding and configuring the GridView, your "by role" interface's declarative markup should look similar to the following:

```
<h3>Manage Users By Role</h3>
<p>
     <b>Select a Role:</b>

     <asp:DropDownList ID="RoleList" runat="server" AutoPostBack="true"></asp:DropDownList>
</p>
<p>      <asp:GridView ID="RolesUserList" runat="server" AutoGenerateColumns="false"

          EmptyDataText="No users belong to this role.">
          <Columns>
               <asp:TemplateField HeaderText="Users">
                    <ItemTemplate>
                         <asp:Label runat="server" id="UserNameLabel"
                              Text='<%# Container.DataItem %>'></asp:Label>

                    </ItemTemplate>
               </asp:TemplateField>
          </Columns>
     </asp:GridView> </p>
```

We need to populate the `RoleList` DropDownList with the set of roles in the system. To accomplish this, update the `BindRolesToList` method so that is binds the string array returned by the `Roles.GetAllRoles` method to the `RolesList` DropDownList (as well as the `UsersRoleList` Repeater).

```
private void BindRolesToList()
{
     // Get all of the roles

     string[] roles = Roles.GetAllRoles();
     UsersRoleList.DataSource = roles;
     UsersRoleList.DataBind();

     RoleList.DataSource = roles;
     RoleList.DataBind();
}
```

The last two lines in the `BindRolesToList` method have been added to bind the set of roles to the `RoleList` DropDownList control. Figure 5 shows the end result when viewed through a browser – a drop-down list populated with the system's roles.

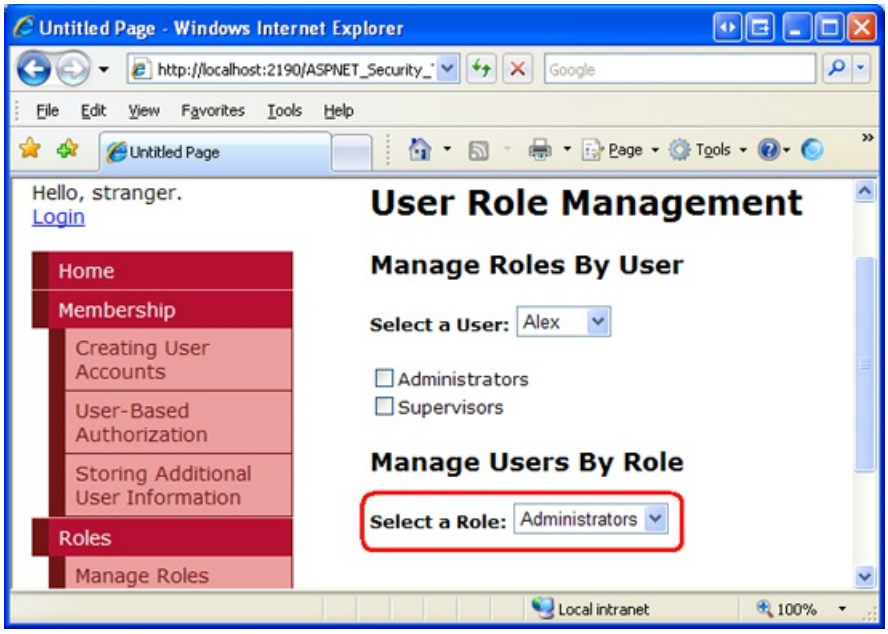


Figure 5: The Roles are Displayed in the `RoleList` DropDownList (Click to view full-size image)

Displaying the Users That Belong To the Selected Role

When the page is first loaded, or when a new role is selected from the `RoleList` DropDownList, we need to display the list of users that belong to that role in the GridView. Create a method named `DisplayUsersBelongingToRole` using the following code:

```
private void DisplayUsersBelongingToRole()
{
     // Get the selected role
     string selectedRoleName = RoleList.SelectedValue;

     // Get the list of usernames that belong to the role
     string[] usersBelongingToRole = Roles.GetUsersInRole(selectedRoleName);

     // Bind the list of users to the GridView
     RolesUserList.DataSource = usersBelongingToRole;
     RolesUserList.DataBind();
}
```

This method starts by getting the selected role from the `RoleList` DropDownList. It then uses the `Roles.GetUsersInRole(roleName)` method to retrieve a string array of the UserNames of the users that belong to that role. This array is then bound to the `RolesUserList` GridView.

This method needs to be called in two circumstances: when the page is initially loaded and when the selected role in the `RoleList` DropDownList changes. Therefore, update the `Page_Load` event handler so that this method is invoked after the call to `CheckRolesForSelectedUser`. Next, create an event handler for the `RoleList`'s `SelectedIndexChanged` event, and call this method from there, too.

```
protected void Page_Load(object sender, EventArgs e)
{
     if (!Page.IsPostBack)
     {
          // Bind the users and roles
          BindUsersToUserList();
          BindRolesToList();

          // Check the selected user's roles
          CheckRolesForSelectedUser();

          // Display those users belonging to the currently selected role
          DisplayUsersBelongingToRole();
     }
}

...

protected void RoleList_SelectedIndexChanged(object sender, EventArgs e)
{
     DisplayUsersBelongingToRole();
}
```

With this code in place, the `RolesUserList` GridView should display those users that belong to the selected role. As Figure 6 shows, the Supervisors role consists of two members: Bruce and Tito.

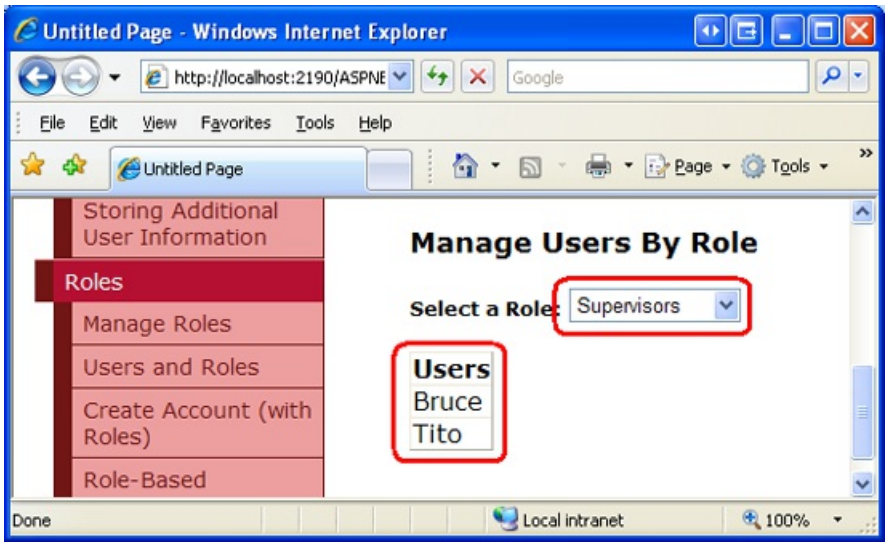


Figure 6: The GridView Lists Those Users That Belong to the Selected Role (Click to view full-size image)

Removing Users from the Selected Role

Let's augment the `RolesUserList` GridView so that it includes a column of "Remove" buttons. Clicking the "Remove" button for a particular user will remove them from that role.

Start by adding a Delete button field to the GridView. Make this field appear as the left most filed and change its `DeleteText` property from "Delete" (the default) to "Remove".

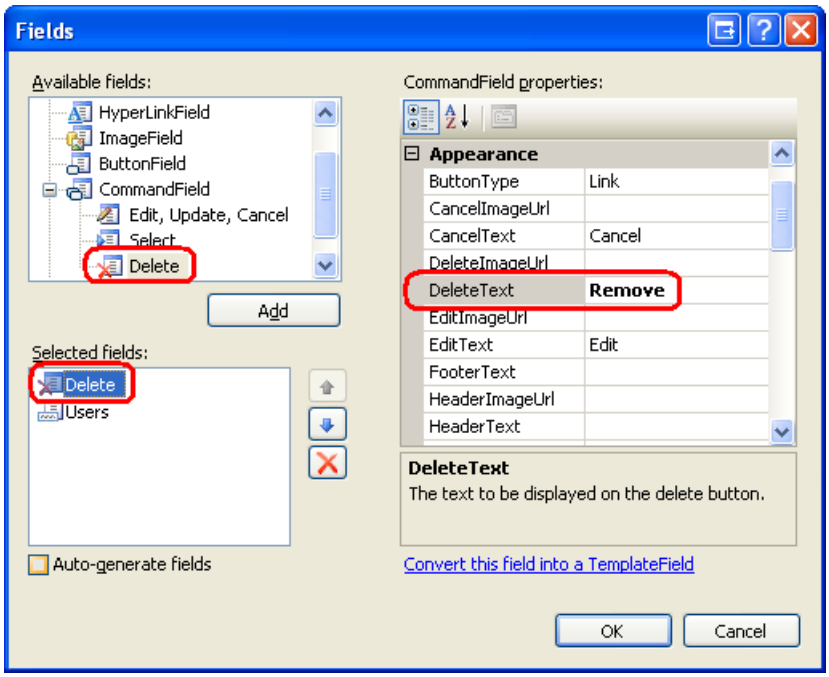


Figure 7: Add the "Remove" Button to the GridView (Click to view full-size image)

When the "Remove" button is clicked a postback ensues and the GridView's `RowDeleting` event is raised. We need to create an event handler for this event and write code that removes the user from the selected role. Create the event handler and then add the following code:

```
protected void RolesUserList_RowDeleting(object sender, GridViewDeleteEventArgs e)
{
     // Get the selected role
     string selectedRoleName = RoleList.SelectedValue;

     // Reference the UserNameLabel
     Label UserNameLabel = RolesUserList.Rows[e.RowIndex].FindControl("UserNameLabel") as Label;

     // Remove the user from the role
     Roles.RemoveUserFromRole(UserNameLabel.Text, selectedRoleName);

     // Refresh the GridView
     DisplayUsersBelongingToRole();

     // Display a status message
     ActionStatus.Text = string.Format("User {0} was removed from role {1}.", UserNameLabel.Text,
selectedRoleName);
}
```

The code starts by determining the selected role name. It then programmatically references the `UserNameLabel` control from the row whose "Remove" button was clicked in order to determine the UserName of the user to remove. The user is then removed from the role via a call to the `Roles.RemoveUserFromRole` method. The `RolesUserList` GridView is then refreshed and a message is displayed via the `ActionStatus` Label control.

> **NOTE**
>
> The "Remove" button does not require any sort of confirmation from the user before removing the user from the role. I invite you to add some level of user confirmation. One of the easiest ways to confirm an action is through a client-side confirm dialog box. For more information on this technique, see Adding Client-Side Confirmation When Deleting.

Figure 8 shows the page after user Tito has been removed from the Supervisors group.

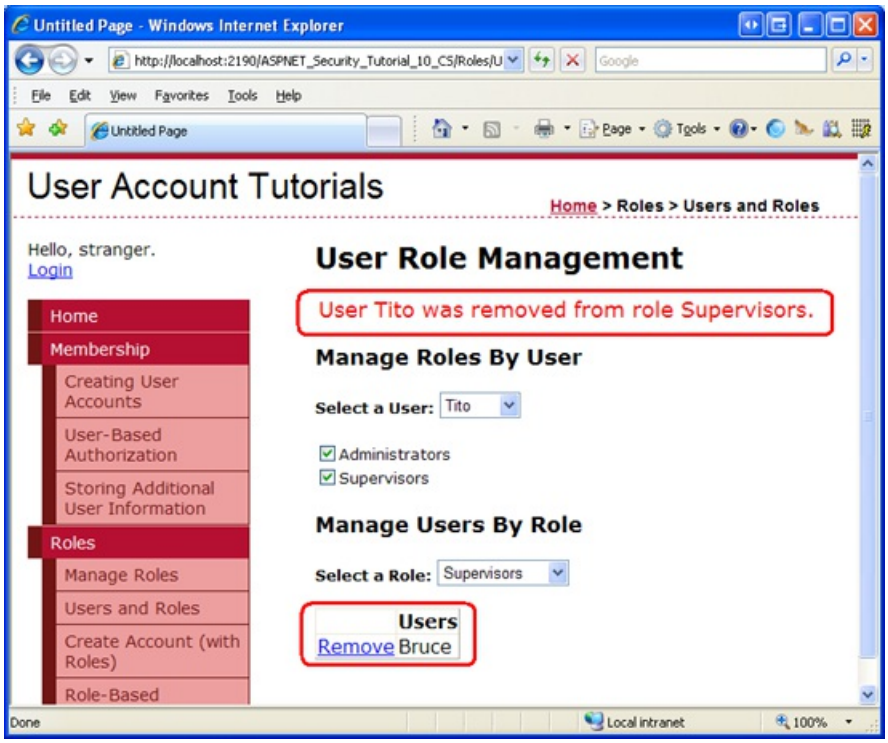


Figure 8: Alas, Tito is No Longer a Supervisor (Click to view full-size image)

Adding New Users to the Selected Role

Along with removing users from the selected role, the visitor to this page should also be able to add a user to the selected role. The best interface for adding a user to the selected role depends on the number of user accounts you expect to have. If your website will house just a few dozen user accounts or less, you could use a DropDownList here. If there might be thousands of user accounts, you would want to include a user interface that permits the visitor to page through the accounts, search for a particular account, or filter the user accounts in some other fashion.

For this page let's use a very simple interface that works regardless of the number of user accounts in the system. Namely, we will use a TextBox, prompting the visitor to type in the username of the user she wants to add to the selected role. If no user with that name exists, or if the user is already a member of the role, we'll display a message in `ActionStatus` Label. But if the user exists and is not a member of the role, we'll add them to the role and refresh the grid.

Add a TextBox and Button beneath the GridView. Set the TextBox's `ID` to `UserNameToAddToRole` and set the Button's `ID` and `Text` properties to `AddUserToRoleButton` and "Add User to Role", respectively.

```
<p>
    <b>UserName:</b>
    <asp:TextBox ID="UserNameToAddToRole" runat="server"></asp:TextBox>
    <br />
    <asp:Button ID="AddUserToRoleButton" runat="server" Text="Add User to Role" />

</p>
```

Next, create a `Click` event handler for the `AddUserToRoleButton` and add the following code:

```
protected void AddUserToRoleButton_Click(object sender, EventArgs e)
{
     // Get the selected role and username

     string selectedRoleName = RoleList.SelectedValue;
     string userNameToAddToRole = UserNameToAddToRole.Text;

     // Make sure that a value was entered
     if (userNameToAddToRole.Trim().Length == 0)
     {
          ActionStatus.Text = "You must enter a username in the textbox.";
          return;
     }

     // Make sure that the user exists in the system
     MembershipUser userInfo = Membership.GetUser(userNameToAddToRole);
     if (userInfo == null)
     {
          ActionStatus.Text = string.Format("The user {0} does not exist in the system.",
userNameToAddToRole);

          return;
     }

     // Make sure that the user doesn't already belong to this role
     if (Roles.IsUserInRole(userNameToAddToRole, selectedRoleName))
     {
          ActionStatus.Text = string.Format("User {0} already is a member of role {1}.", userNameToAddToRole,
selectedRoleName);
          return;
     }

     // If we reach here, we need to add the user to the role
     Roles.AddUserToRole(userNameToAddToRole, selectedRoleName);

     // Clear out the TextBox
     UserNameToAddToRole.Text = string.Empty;

     // Refresh the GridView
     DisplayUsersBelongingToRole();

     // Display a status message

     ActionStatus.Text = string.Format("User {0} was added to role {1}.", userNameToAddToRole,
selectedRoleName); }
```

The majority of the code in the `Click` event handler performs various validation checks. It ensures that the visitor supplied a username in the `UserNameToAddToRole` TextBox, that the user exists in the system, and that they don't already belong to the selected role. If any of these checks fails, an appropriate message is displayed in `ActionStatus` and the event handler is exited. If all of the checks pass, the user is added to the role via the `Roles.AddUserToRole` method. Following that, the TextBox's `Text` property is cleared out, the GridView is refreshed, and the `ActionStatus` Label displays a message indicating that the specified user was successfully added to the selected role.

NOTE

To ensure that the specified user does not already belong to the selected role, we use the `Roles.IsUserInRole(userName, roleName)` method, which returns a Boolean value indicating whether *userName* is a member of *roleName*. We will use this method again in the next tutorial when we look at role-based authorization.

Visit the page through a browser and select the Supervisors role from the `RoleList` DropDownList. Try entering

an invalid username – you should see a message explaining that the user does not exist in the system.

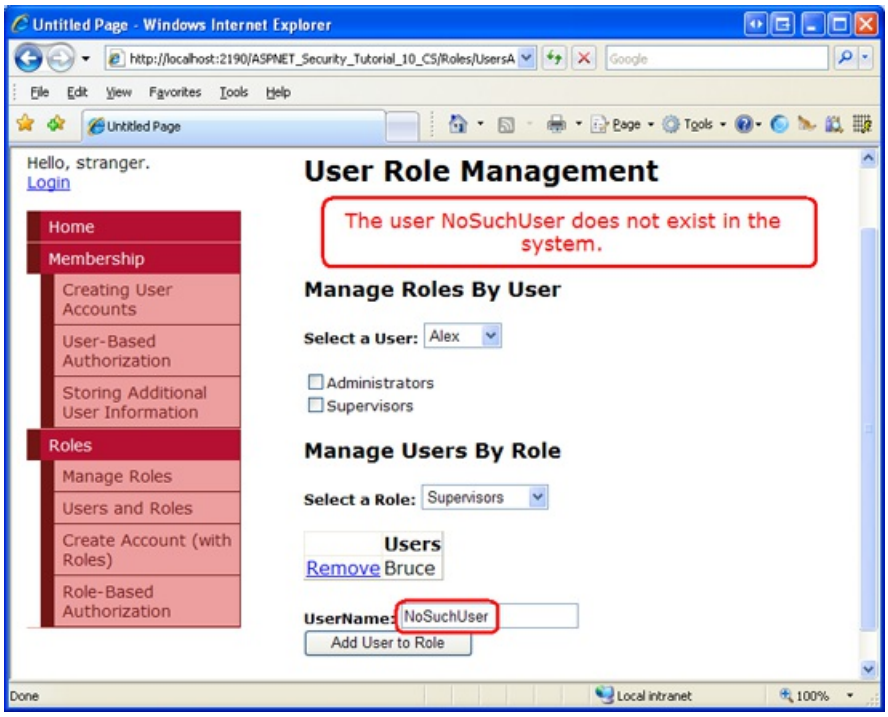


Figure 9: You Cannot Add a Non-Existent User to a Role (Click to view full-size image)

Now try adding a valid user. Go ahead and re-add Tito to the Supervisors role.

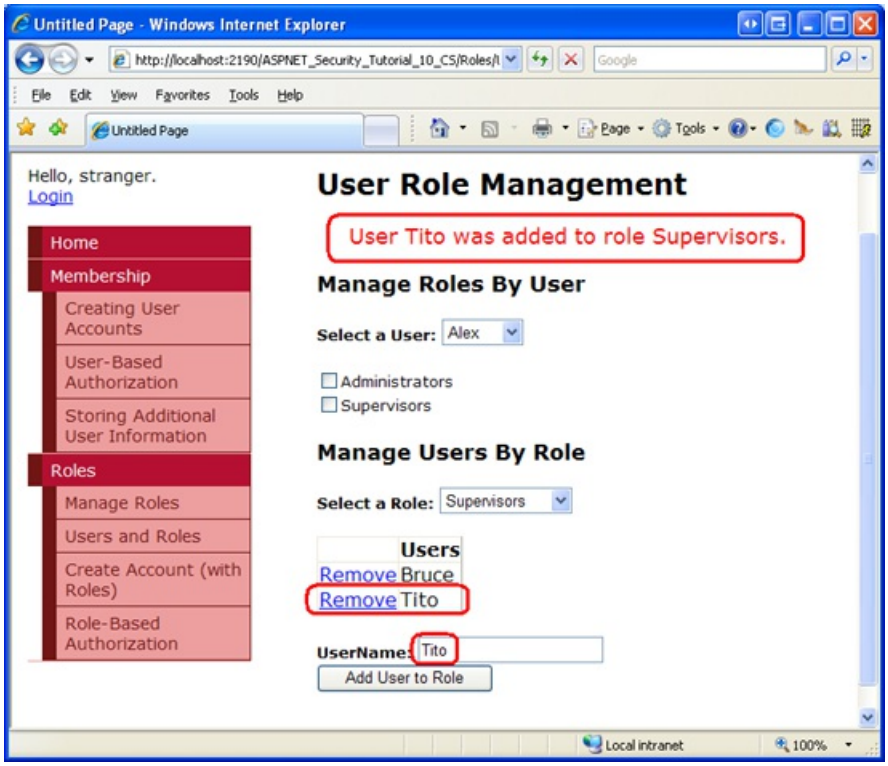


Figure 10: Tito Is Once Again a Supervisor! (Click to view full-size image)

Step 3: Cross-Updating the "By User" and "By Role" Interfaces

The `UsersAndRoles.aspx` page offers two distinct interfaces for managing users and roles. Currently, these two interfaces act independently of one another so it is possible that a change made in one interface will not be reflected immediately in the other. For example, imagine that the visitor to the page selects the Supervisors role from the `RoleList` DropDownList, which lists Bruce and Tito as its members. Next, the visitor selects Tito from the `UserList` DropDownList, which checks the Administrators and Supervisors checkboxes in the `UsersRoleList`

Repeater. If the visitor then unchecks the Supervisor role from the Repeater, Tito is removed from the Supervisors role, but this modification is not reflected in the "by role" interface. The GridView will still show Tito as being a member of the Supervisors role.

To fix this we need to refresh the GridView whenever a role is checked or unchecked from the `UsersRoleList` Repeater. Likewise, we need to refresh the Repeater whenever a user is removed or added to a role from the "by role" interface.

The Repeater in the "by user" interface is refreshed by calling the `CheckRolesForSelectedUser` method. The "by role" interface can be modified in the `RolesUserList` GridView's `RowDeleting` event handler and the `AddUserToRoleButton` Button's `Click` event handler. Therefore, we need to call the `CheckRolesForSelectedUser` method from each of these methods.

```
protected void RolesUserList_RowDeleting(object sender, GridViewDeleteEventArgs e)
{
     ... Code removed for brevity ...

     // Refresh the "by user" interface
     CheckRolesForSelectedUser();
}

protected void AddUserToRoleButton_Click(object sender, EventArgs e)
{
     ... Code removed for brevity ...


     // Refresh the "by user" interface
     CheckRolesForSelectedUser();
}
```

Similarly, the GridView in the "by role" interface is refreshed by calling the `DisplayUsersBelongingToRole` method and the "by user" interface is modified through the `RoleCheckBox_CheckChanged` event handler. Therefore, we need to call the `DisplayUsersBelongingToRole` method from this event handler.

```
protected void RoleCheckBox_CheckChanged(object sender, EventArgs e)
{
     ... Code removed for brevity...

     // Refresh the "by role" interface
     DisplayUsersBelongingToRole();
}
```

With these minor code changes, the "by user" and "by role" interfaces now correctly cross-update. To verify this, visit the page through a browser and select Tito and Supervisors from the `UserList` and `RoleList` DropDownLists, respectively. Note that as you uncheck the Supervisors role for Tito from the Repeater in the "by user" interface, Tito is automatically removed from the GridView in the "by role" interface. Adding Tito back to the Supervisors role from the "by role" interface automatically re-checks the Supervisors checkbox in the "by user" interface.

Step 4: Customizing the CreateUserWizard to Include a "Specify Roles" Step

In the *Creating User Accounts* tutorial we saw how to use the CreateUserWizard Web control to provide an interface for creating a new user account. The CreateUserWizard control can be used in one of two ways:

- As a means for visitors to create their own user account on the site, and
- As a means for administrators to create new accounts

In the first use case, a visitor comes to the site and fills out the CreateUserWizard, entering their information in order to register on the site. In the second case, an administrator creates a new account for another person.

When an account is being created by an administrator for some other person, it might be helpful to allow the administrator to specify what roles the new user account belongs to. In the *Storing Additional User Information* tutorial we saw how to customize the CreateUserWizard by adding additional `WizardSteps`. Let's look at how to add an additional step to the CreateUserWizard in order to specify the new user's roles.

Open the `CreateUserWizardWithRoles.aspx` page and add a CreateUserWizard control named `RegisterUserWithRoles`. Set the control's `ContinueDestinationPageUrl` property to "~/Default.aspx". Because the idea here is that an administrator will be using this CreateUserWizard control to create new user accounts, set the control's `LoginCreatedUser` property to False. This `LoginCreatedUser` property specifies whether the visitor is automatically logged on as the just-created user, and it defaults to True. We set it to False because when an administrator creates a new account we want to keep him signed in as himself.

Next, select the "Add/Remove `WizardSteps`..." option from the CreateUserWizard's Smart Tag and add a new `WizardStep`, setting its `ID` to `SpecifyRolesStep`. Move the `SpecifyRolesStep WizardStep` so that it comes after the "Sign Up for Your New Account" step, but before the "Complete" step. Set the `WizardStep`'s `Title` property to "Specify Roles", its `StepType` property to `Step`, and its `AllowReturn` property to False.

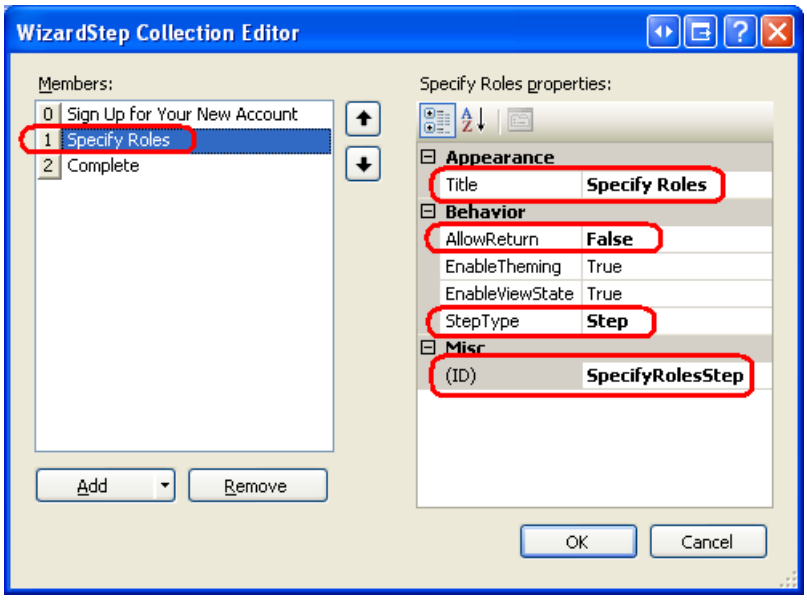


Figure 11: Add the "Specify Roles" `WizardStep` to the CreateUserWizard (Click to view full-size image)

After this change your CreateUserWizard's declarative markup should look like the following:

```
<asp:CreateUserWizard ID="RegisterUserWithRoles" runat="server"
     ContinueDestinationPageUrl="~/Default.aspx" LoginCreatedUser="False">

     <WizardSteps>
          <asp:CreateUserWizardStep ID="CreateUserWizardStep1" runat="server">
          </asp:CreateUserWizardStep>
          <asp:WizardStep ID="SpecifyRolesStep" runat="server" StepType="Step"

               Title="Specify Roles" AllowReturn="False">
          </asp:WizardStep>
          <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">
          </asp:CompleteWizardStep>
     </WizardSteps>

</asp:CreateUserWizard>
```

In the "Specify Roles" `WizardStep`, add a CheckBoxList named `RoleList`. This CheckBoxList will list the available

roles, enabling the person visiting the page to check what roles the newly created user belongs to.

We are left with two coding tasks: first we must populate the `RoleList` CheckBoxList with the roles in the system; second, we need to add the created user to the selected roles when the user moves from the "Specify Roles" step to the "Complete" step. We can accomplish the first task in the `Page_Load` event handler. The following code programmatically references the `RoleList` CheckBox on the first visit to the page and binds the roles in the system to it.

```
protected void Page_Load(object sender, EventArgs e)
{
     if (!Page.IsPostBack)
     {
          // Reference the SpecifyRolesStep WizardStep
          WizardStep SpecifyRolesStep = RegisterUserWithRoles.FindControl("SpecifyRolesStep") as WizardStep;

          // Reference the RoleList CheckBoxList
          CheckBoxList RoleList = SpecifyRolesStep.FindControl("RoleList") as CheckBoxList;

          // Bind the set of roles to RoleList
          RoleList.DataSource = Roles.GetAllRoles();
          RoleList.DataBind();
     }
}
```

The above code should look familiar. In the *Storing Additional User Information* tutorial we used two `FindControl` statements to reference a Web control from within a custom `WizardStep`. And the code that binds the roles to the CheckBoxList was taken from earlier in this tutorial.

In order to perform the second programming task we need to know when the "Specify Roles" step has been completed. Recall that the CreateUserWizard has an `ActiveStepChanged` event, which fires each time the visitor navigates from one step to another. Here we can determine if the user has reached the "Complete" step; if so, we need to add the user to the selected roles.

Create an event handler for the `ActiveStepChanged` event and add the following code:

```
protected void RegisterUserWithRoles_ActiveStepChanged(object sender, EventArgs e)
{
     // Have we JUST reached the Complete step?
     if (RegisterUserWithRoles.ActiveStep.Title == "Complete")
     {
          // Reference the SpecifyRolesStep WizardStep
          WizardStep SpecifyRolesStep = RegisterUserWithRoles.FindControl("SpecifyRolesStep") as WizardStep;

          // Reference the RoleList CheckBoxList
          CheckBoxList RoleList = SpecifyRolesStep.FindControl("RoleList") as CheckBoxList;

          // Add the checked roles to the just-added user
          foreach (ListItem li in RoleList.Items)

          {
               if (li.Selected)
                    Roles.AddUserToRole(RegisterUserWithRoles.UserName, li.Text);
          }
     }
}
```

If the user has just reached the "Completed" step, the event handler enumerates the items of the `RoleList` CheckBoxList and the just-created user is assigned to the selected roles.

Visit this page through a browser. The first step in the CreateUserWizard is the standard "Sign Up for Your New Account" step, which prompts for the new user's username, password, email, and other key information. Enter the

information to create a new user named Wanda.

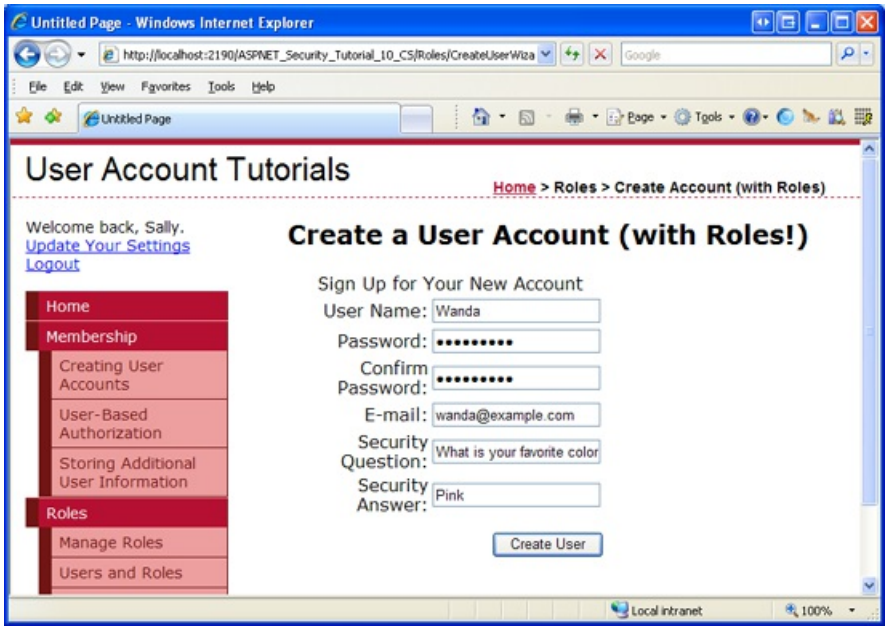


Figure 12: Create a New User Named Wanda (Click to view full-size image)

Click the "Create User" button. The CreateUserWizard internally calls the `Membership.CreateUser` method, creating the new user account, and then progresses to the next step, "Specify Roles." Here the system roles are listed. Check the Supervisors checkbox and click Next.

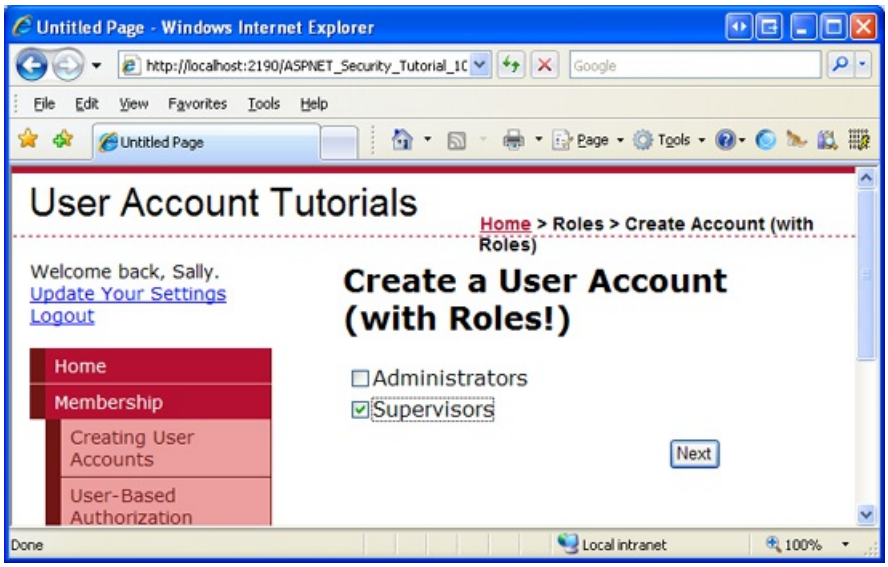


Figure 13: Make Wanda a Member of the Supervisors Role (Click to view full-size image)

Clicking Next causes a postback and updates the `ActiveStep` to the "Complete" step. In the `ActiveStepChanged` event handler, the recently-created user account is assigned to the Supervisors role. To verify this, return to the `UsersAndRoles.aspx` page and select Supervisors from the `RoleList` DropDownList. As Figure 14 shows, the Supervisors are now made up of three users: Bruce, Tito, and Wanda.

Figure 14: Bruce, Tito, and Wanda are All Supervisors (Click to view full-size image)

Summary

The Roles framework offers methods for retrieving information about a particular user's roles and methods for determining what users belong to a specified role. Furthermore, there are a number of methods for adding and removing one or more users to one or more roles. In this tutorial we focused on just two of these methods: `AddUserToRole` and `RemoveUserFromRole`. There are additional variants designed to add multiple users to a single role and to assign multiple roles to a single user.

This tutorial also included a look at extending the CreateUserWizard control to include a `WizardStep` to specify the newly-created user's roles. Such a step could help an administrator streamline the process of creating user accounts for new users.

At this point we have seen how to create and delete roles and how to add and remove users from roles. But we have yet to look at applying role-based authorization. In the following tutorial we will look at defining URL authorization rules on a role-by-role basis, as well as how to limit page-level functionality based on the currently logged in user's roles.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Web Site Administration Tool Overview
- Examining ASP.NET's Membership, Roles, and Profile
- Rolling Your Own Website Administration Tool

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Role-Based Authorization (C#)

2/20/2020 • 34 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

This tutorial starts with a look at how the Roles framework associates a user's roles with his security context. It then examines how to apply role-based URL authorization rules. Following that, we will look at using declarative and programmatic means for altering the data displayed and the functionality offered by an ASP.NET page.

Introduction

In the *User-Based Authorization* tutorial we saw how to use URL authorization to specify what users could visit a particular set of pages. With just a little bit of markup in `Web.config`, we could instruct ASP.NET to allow only authenticated users to visit a page. Or we could dictate that only users Tito and Bob were allowed, or indicate that all authenticated users except for Sam were permitted.

In addition to URL authorization, we also looked at declarative and programmatic techniques for controlling the data displayed and the functionality offered by a page based on the user visiting. In particular, we created a page that listed the contents of the current directory. Anyone could visit this page, but only authenticated users could view the files' contents and only Tito could delete the files.

Applying authorization rules on a user-by-user basis can grow into a bookkeeping nightmare. A more maintainable approach is to use role-based authorization. The good news is that the tools at our disposal for applying authorization rules work equally well with roles as they do for user accounts. URL authorization rules can specify roles instead of users. The LoginView control, which renders different output for authenticated and anonymous users, can be configured to display different content based on the logged in user's roles. And the Roles API includes methods for determining the logged in user's roles.

This tutorial starts with a look at how the Roles framework associates a user's roles with his security context. It then examines how to apply role-based URL authorization rules. Following that, we will look at using declarative and programmatic means for altering the data displayed and the functionality offered by an ASP.NET page. Let's get started!

Understanding How Roles are Associated with a User's Security Context

Whenever a request enters the ASP.NET pipeline it is associated with a security context, which includes information identifying the requestor. When using forms authentication, an authentication ticket is used as an identity token. As we discussed in the *An Overview of Forms Authentication* and *Forms Authentication Configuration and Advanced Topics* tutorials, the `FormsAuthenticationModule` is responsible for determining the identity of the requestor, which it does during the `AuthenticateRequest` event.

If a valid, non-expired authentication ticket is found, the `FormsAuthenticationModule` decodes it to ascertain the requestor's identity. It creates a new `GenericPrincipal` object and assigns this to the `HttpContext.User` object. The purpose of a principal, like `GenericPrincipal`, is to identify the authenticated user's name and what roles she belong to. This purpose is evident by the fact that all principal objects have an `Identity` property and an `IsInRole(roleName)` method. The `FormsAuthenticationModule`, however, is not interested in recording role

information and the `GenericPrincipal` object it creates does not specify any roles.

If the Roles framework is enabled, the `RoleManagerModule` HTTP Module steps in after the `FormsAuthenticationModule` and identifies the authenticated user's roles during the `PostAuthenticateRequest` event, which fires after the `AuthenticateRequest` event. If the request is from an authenticated user, the `RoleManagerModule` overwrites the `GenericPrincipal` object created by the `FormsAuthenticationModule` and replaces it with a `RolePrincipal` object. The `RolePrincipal` class uses the Roles API to determine what roles the user belongs to.

Figure 1 depicts the ASP.NET pipeline workflow when using forms authentication and the Roles framework. The `FormsAuthenticationModule` executes first, identifies the user via her authentication ticket, and creates a new `GenericPrincipal` object. Next, the `RoleManagerModule` steps in and overwrites the `GenericPrincipal` object with a `RolePrincipal` object.

If an anonymous user visits the site, neither the `FormsAuthenticationModule` nor the `RoleManagerModule` creates a principal object.

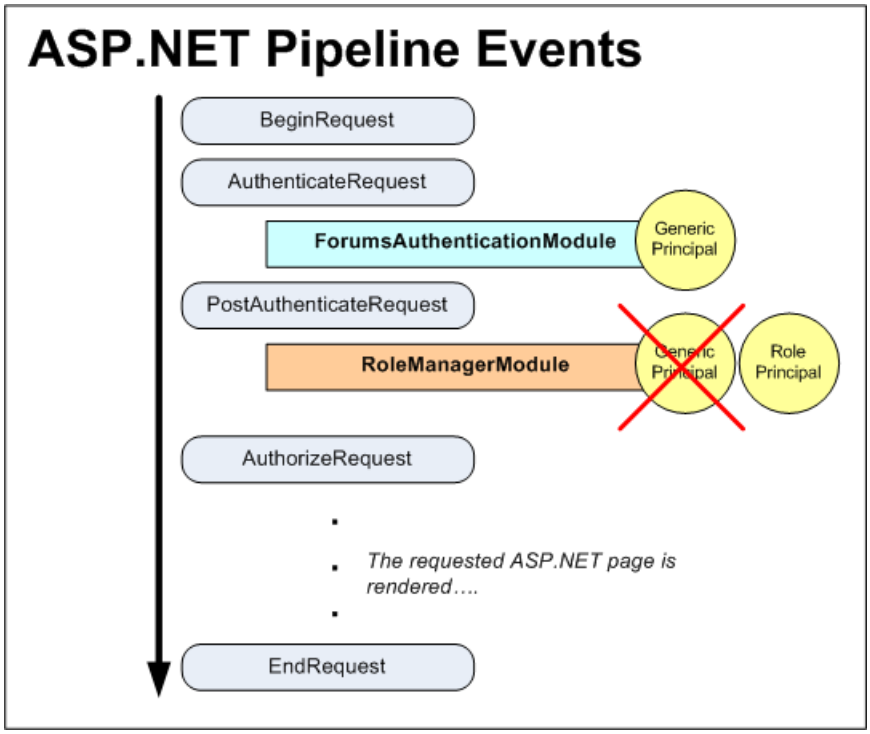


Figure 1: The ASP.NET Pipeline Events for an Authenticated User When Using Forms Authentication and the Roles Framework (Click to view full-size image)

Caching Role Information in a Cookie

The `RolePrincipal` object's `IsInRole(roleName)` method calls `Roles.GetRolesForUser` to get the roles for the user in order to determine whether the user is a member of *roleName*. When using the `SqlRoleProvider`, this results in a query to the role store database. When using role-based URL authorization rules the `RolePrincipal`'s `IsInRole` method will be called on every request to a page that is protected by the role-based URL authorization rules. Rather than have to lookup the role information in the database on every request, the Roles framework includes an option to cache the user's roles in a cookie.

If the Roles framework is configured to cache the user's roles in a cookie, the `RoleManagerModule` creates the cookie during the ASP.NET pipeline's `EndRequest` event. This cookie is used in subsequent requests in the `PostAuthenticateRequest`, which is when the `RolePrincipal` object is created. If the cookie is valid and has not expired, the data in the cookie is parsed and used to populate the user's roles, thereby saving the `RolePrincipal` from having to make a call to the `Roles` class to determine the user's roles. Figure 2 depicts this workflow.

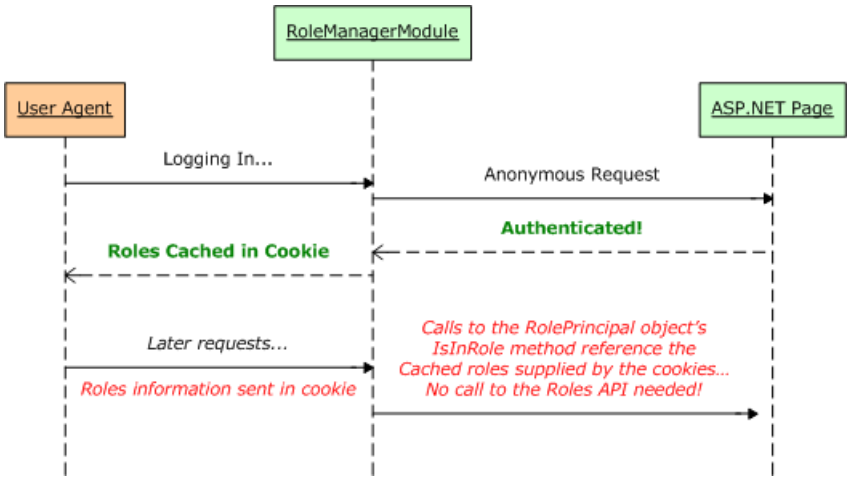


Figure 2: The User's Role Information Can Be Stored in a Cookie to Improve Performance (Click to view full-size image)

By default, the role cache cookie mechanism is disabled. It can be enabled through the `<roleManager>` configuration markup in `Web.config`. We discussed using the `<roleManager>` element to specify Role providers in the *Creating and Managing Roles* tutorial, so you should already have this element in your application's `Web.config` file. The role cache cookie settings are specified as attributes of the `<roleManager>` element, and are summarized in Table 1.

> **NOTE**
>
> The configuration settings listed in Table 1 specify the properties of the resulting role cache cookie. For more information on cookies, how they work, and their various properties, read this Cookies tutorial.

PROPERTY	DESCRIPTION
`cacheRolesInCookie`	A Boolean value that indicates whether cookie caching is used. Defaults to `false`.
`cookieName`	The name of the role cache cookie. The default value is ".ASPXROLES".
`cookiePath`	The path for the roles name cookie. The path attribute enables a developer to limit the scope of a cookie to a particular directory hierarchy. The default value is "/", which informs the browser to send the authentication ticket cookie to any request made to the domain.
`cookieProtection`	Indicates what techniques are used to protect the role cache cookie. The allowable values are: `All` (the default); `Encryption`; `None`; and `Validation`. Refer back to Step 3 of the *Forms Authentication Configuration and Advanced Topics* tutorial for more information on these protection levels.
`cookieRequireSSL`	A Boolean value that indicates whether an SSL connection is required to transmit the authentication cookie. The default value is `false`.

PROPERTY	DESCRIPTION
`cookieSlidingExpiration`	A Boolean value that indicates whether the cookie's timeout is reset each time the user visits the site during a single session. The default value is `false`. This value is only pertinent when `createPersistentCookie` is set to `true`.
`cookieTimeout`	Specifies the time, in minutes, after which the authentication ticket cookie expires. The default value is `30`. This value is only pertinent when `createPersistentCookie` is set to `true`.
`createPersistentCookie`	A Boolean value that specifies whether the role cache cookie is a session cookie or persistent cookie. If `false` (the default), a session cookie is used, which is deleted when the browser is closed. If `true`, a persistent cookie is used; it expires `cookieTimeout` number of minutes after it has been created or after the previous visit, depending on the value of `cookieSlidingExpiration`.
`domain`	Specifies the cookie's domain value. The default value is an empty string, which causes the browser to use the domain from which it was issued (such as www.yourdomain.com). In this case, the cookie will **not** be sent when making requests to subdomains, such as admin.yourdomain.com. If you want the cookie to be passed to all subdomains you need to customize the `domain` attribute, setting it to "yourdomain.com".
`maxCachedResults`	Specifies the maximum number of role names that are cached in the cookie. The default is 25. The `RoleManagerModule` does not create a cookie for users that belong to more than `maxCachedResults` roles. Consequently, the `RolePrincipal` object's `IsInRole` method will use the `Roles` class to determine the user's roles. The reason `maxCachedResults` exists is because many user agents do not permit cookies larger than 4,096 bytes. So this cap is meant to reduce the likelihood of exceeding this size limitation. If you have extremely long role names, you may want to consider specifying a smaller `maxCachedResults` value; contrariwise, if you have extremely short role names, you can probably increase this value.

Table 1: The Role Cache Cookie Configuration Options

Let's configure our application to use non-persistent role cache cookies. To accomplish this, update the `<roleManager>` element in `Web.config` to include the following cookie-related attributes:

```
<roleManager enabled="true"
          defaultProvider="SecurityTutorialsSqlRoleProvider"
          cacheRolesInCookie="true"
          createPersistentCookie="false"
          cookieProtection="All">

     <providers>
     ...
     </providers>
</roleManager>
```

I updated the `<roleManager>` element by adding three attributes: `cacheRolesInCookie`, `createPersistentCookie`, and `cookieProtection`. By setting `cacheRolesInCookie` to `true`, the `RoleManagerModule` will now automatically cache the user's roles in a cookie rather than having to lookup the user's role information on each request. I explicitly set the `createPersistentCookie` and `cookieProtection` attributes to `false` and `All`, respectively. Technically, I didn't need to specify values for these attributes since I just assigned them to their default values, but I put them here to make it explicitly clear that I am not using persistent cookies and that the cookie is both encrypted and validated.

That's all there is to it! Henceforth, the Roles framework will cache the users' roles in cookies. If the user's browser does not support cookies, or if their cookies are deleted or lost, somehow, it's no big deal – the `RolePrincipal` object will simply use the `Roles` class in the case that no cookie (or an invalid or expired one) is available.

> **NOTE**
>
> Microsoft's Patterns & Practices group discourages using persistent role cache cookies. Since possession of the role cache cookie is sufficient to prove role membership, if a hacker can somehow gain access to a valid user's cookie he can impersonate that user. The likelihood of this happening increases if the cookie is persisted on the user's browser. For more information on this security recommendation, as well as other security concerns, refer to the Security Question List for ASP.NET 2.0.

Step 1: Defining Role-Based URL Authorization Rules

As discussed in the *User-Based Authorization* tutorial, URL authorization offers a means to restrict access to a set of pages on a user-by-user or role-by-role basis. The URL authorization rules are spelled out in `Web.config` using the `<authorization>` element with `<allow>` and `<deny>` child elements. In addition to the user-related authorization rules discussed in previous tutorials, each `<allow>` and `<deny>` child element can also include:

- A particular role
- A comma-delimited list of roles

For example, the URL authorization rules grant access to those users in the Administrators and Supervisors roles, but deny access to all others:

```
<authorization>
    <allow roles="Administrators, Supervisors" />
    <deny users="*" />
</authorization>
```

The `<allow>` element in the above markup states that the Administrators and Supervisors roles are allowed; the `<deny>` element instructs that *all* users are denied.

Let's configure our application so that the `ManageRoles.aspx`, `UsersAndRoles.aspx`, and `CreateUserWizardWithRoles.aspx` pages are only accessible to those users in the Administrators role, while the `RoleBasedAuthorization.aspx` page remains accessible to all visitors.

To accomplish this, start by adding a `Web.config` file to the `Roles` folder.

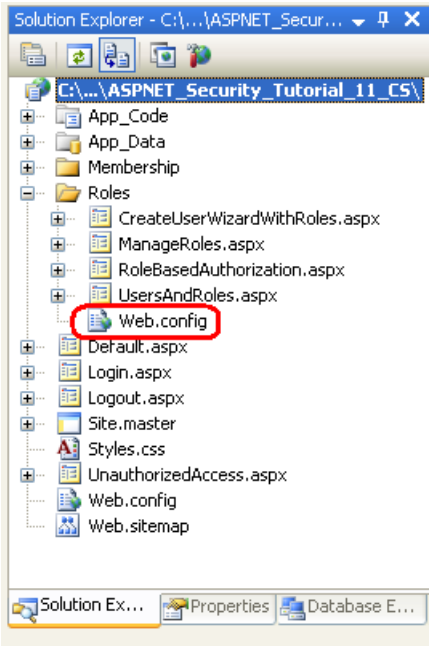


Figure 3: Add a `Web.config` File to the `Roles` directory (Click to view full-size image)

Next, add the following configuration markup to `Web.config`:

```
<?xml version="1.0"?>

<configuration>
     <system.web>
          <authorization>
               <allow roles="Administrators" />
               <deny users="*"/>
          </authorization>

     </system.web>

     <!-- Allow all users to visit RoleBasedAuthorization.aspx -->
     <location path="RoleBasedAuthorization.aspx">
          <system.web>
               <authorization>
                    <allow users="*" />

               </authorization>
          </system.web>
     </location>
</configuration>
```

The `<authorization>` element in the `<system.web>` section indicates that only users in the Administrators role may access the ASP.NET resources in the `Roles` directory. The `<location>` element defines an alternate set of URL authorization rules for the `RoleBasedAuthorization.aspx` page, allowing all users to visit the page.

After saving your changes to `Web.config`, log in as a user that is not in the Administrators role and then try to visit one of the protected pages. The `UrlAuthorizationModule` will detect that you do not have permission to visit the requested resource; consequently, the `FormsAuthenticationModule` will redirect you to the login page. The login page will then redirect you to the `UnauthorizedAccess.aspx` page (see Figure 4). This final redirect from the login page to `UnauthorizedAccess.aspx` occurs because of code we added to the login page in Step 2 of the *User-Based Authorization* tutorial. In particular, the login page automatically redirects any authenticated user to `UnauthorizedAccess.aspx` if the querystring contains a `ReturnUrl` parameter, as this parameter indicates that the user arrived at the login page after attempting to view a page he was not authorized to view.

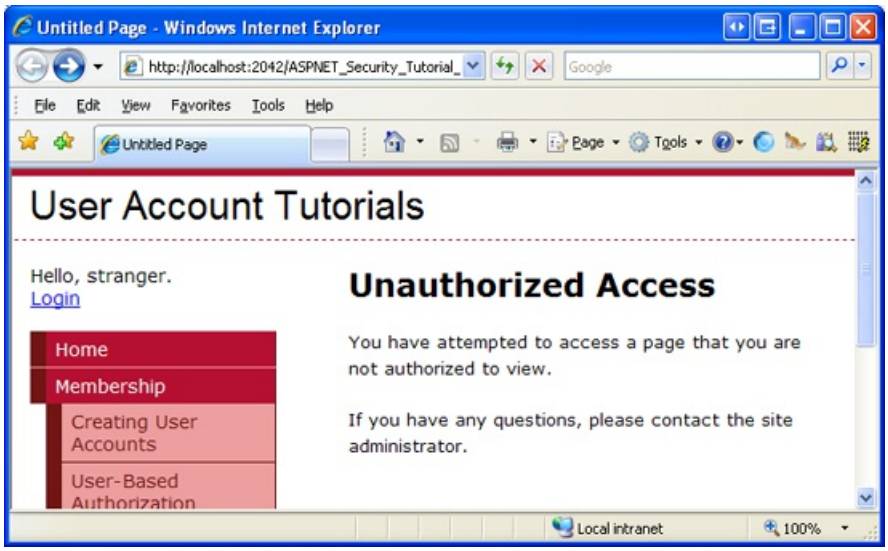


Figure 4: Only Users in the Administrators Role Can View the Protected Pages (Click to view full-size image)

Log off and then log in as a user that is in the Administrators role. Now you should be able to view the three protected pages.

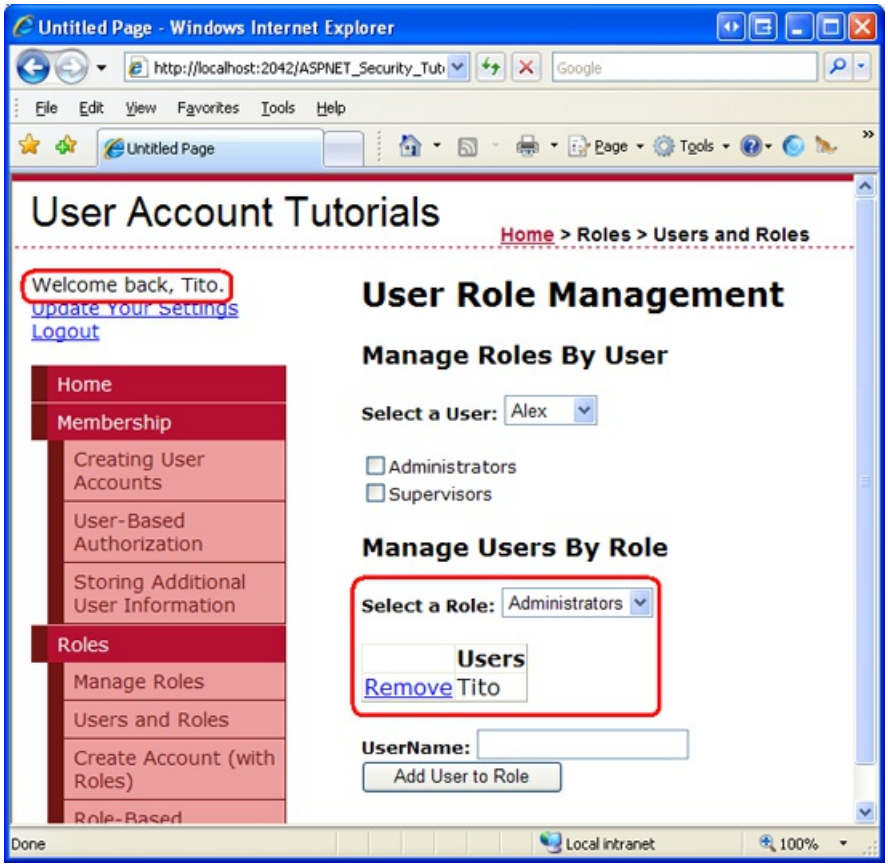


Figure 5: Tito Can Visit the `UsersAndRoles.aspx` Page Because He is in the Administrators Role (Click to view full-size image)

NOTE

When specifying URL authorization rules – for roles or users – it is important to keep in mind that the rules are analyzed one at a time, from the top down. As soon as a match is found, the user is granted or denied access, depending on if the match was found in an `<allow>` or `<deny>` element. **If no match is found, the user is granted access.** Consequently, if you want to restrict access to one or more user accounts, it is imperative that you use a `<deny>` element as the last element in the URL authorization configuration. **If your URL authorization rules do not include a `<deny>` element, all users will be granted access.** For a more thorough discussion on how the URL authorization rules are analyzed, refer back to the "A Look at How the `UrlAuthorizationModule` Uses the Authorization Rules to Grant or Deny Access" section of the *User-Based Authorization* tutorial.

Step 2: Limiting Functionality Based On the Currently Logged In User's Roles

URL authorization makes it easy to specify coarse authorization rules that state what identities are permitted and which ones are denied from viewing a particular page (or all pages in a folder and its subfolders). However, in certain cases we may want to allow all users to visit a page, but limit the page's functionality based on the visiting user's roles. This may entail showing or hiding data based on the user's role, or offering additional functionality to users that belong to a particular role.

Such fine grain role-based authorization rules can be implemented either declaratively or programmatically (or through some combination of the two). In the next section we will see how to implement declarative fine grain authorization via the LoginView control. Following that, we will explore programmatic techniques. Before we can look at applying fine grain authorization rules, however, we first need to create a page whose functionality depends on the role of the user visiting it.

Let's create a page that lists all of the user accounts in the system in a GridView. The GridView will include each user's username, email address, last login date, and comments about the user. In addition to displaying each user's information, the GridView will include edit and delete capabilities. We will initially create this page with the edit and delete functionality available to all users. In the "Using the LoginView Control" and "Programmatically Limiting Functionality" sections we will see how to enable or disable these features based on the visiting user's role.

NOTE

The ASP.NET page we are about to build uses a GridView control to display the user accounts. Since this tutorial series focuses on forms authentication, authorization, user accounts, and roles, I do not want to spend too much time discussing the inner workings of the GridView control. While this tutorial provides specific step-by-step instructions for setting up this page, it does not delve into the details of why certain choices were made, or what effect particular properties have on the rendered output. For a thorough examination of the GridView control, check out my *Working with Data in ASP.NET 2.0* tutorial series.

Start by opening the `RoleBasedAuthorization.aspx` page in the `Roles` folder. Drag a GridView from the page onto the Designer and set its `ID` to `UserGrid`. In a moment we will write code that calls the `Membership.GetAllUsers` method and binds the resulting `MembershipUserCollection` object to the GridView. The `MembershipUserCollection` contains a `MembershipUser` object for each user account in the system; `MembershipUser` objects have properties like `UserName`, `Email`, `LastLoginDate`, and so forth.

Before we write the code that binds the user accounts to the grid, let's first define the GridView's fields. From the GridView's Smart Tag, click the "Edit Columns" link to launch the Fields dialog box (see Figure 6). From here, uncheck the "Auto-generate fields" checkbox in the lower left corner. Since we want this GridView to include editing and deleting capabilities, add a CommandField and set its `ShowEditButton` and `ShowDeleteButton`

properties to True. Next, add four fields for displaying the `UserName`, `Email`, `LastLoginDate`, and `Comment` properties. Use a BoundField for the two read-only properties (`UserName` and `LastLoginDate`) and TemplateFields for the two editable fields (`Email` and `Comment`).

Have the first BoundField display the `UserName` property; set its `HeaderText` and `DataField` properties to "UserName". This field will not be editable, so set its `ReadOnly` property to True. Configure the `LastLoginDate` BoundField by setting its `HeaderText` to "Last Login" and its `DataField` to "LastLoginDate". Let's format the output of this BoundField so that just the date is displayed (instead of the date and time). To accomplish this, set this BoundField's `HtmlEncode` property to False and its `DataFormatString` property to "{0:d}". Also set the `ReadOnly` property to True.

Set the `HeaderText` properties of the two TemplateFields to "Email" and "Comment".

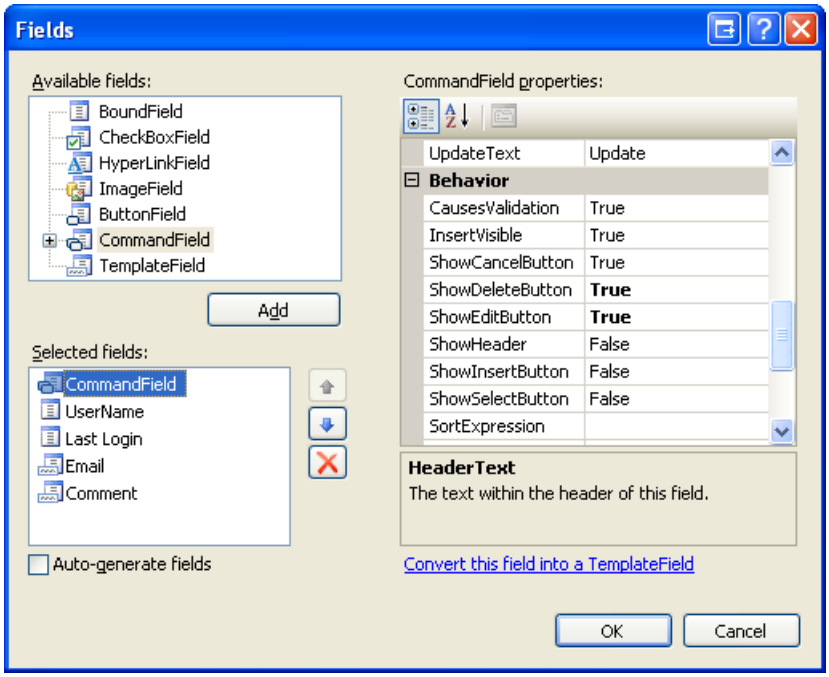


Figure 6: The GridView's Fields Can Be Configured Through the Fields Dialog Box (Click to view full-size image)

We now need to define the `ItemTemplate` and `EditItemTemplate` for the "Email" and "Comment" TemplateFields. Add a Label Web control to each of the `ItemTemplate` s and bind their `Text` properties to the `Email` and `Comment` properties, respectively.

For the "Email" TemplateField, add a TextBox named `Email` to its `EditItemTemplate` and bind its `Text` property to the `Email` property using two-way databinding. Add a RequiredFieldValidator and RegularExpressionValidator to the `EditItemTemplate` to ensure that a visitor editing the Email property has entered a valid email address. For the "Comment" TemplateField, add a multi-line TextBox named `Comment` to its `EditItemTemplate`. Set the TextBox's `Columns` and `Rows` properties to 40 and 4, respectively, and then bind its `Text` property to the `Comment` property using two-way databinding.

After configuring these TemplateFields, their declarative markup should look similar to the following:

```
<asp:TemplateField HeaderText="Email">
     <ItemTemplate>
          <asp:Label runat="server" ID="Label1" Text='<%# Eval("Email") %>'></asp:Label>

     </ItemTemplate>
     <EditItemTemplate>
          <asp:TextBox runat="server" ID="Email" Text='<%# Bind("Email") %>'></asp:TextBox>

          <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
               ControlToValidate="Email" Display="Dynamic"
               ErrorMessage="You must provide an email address."
               SetFocusOnError="True">*</asp:RequiredFieldValidator>

          <asp:RegularExpressionValidator ID="RegularExpressionValidator1" runat="server"
               ControlToValidate="Email" Display="Dynamic"
               ErrorMessage="The email address you have entered is not valid. Please fix
               this and try again."
               SetFocusOnError="True"

               ValidationExpression="\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*">*
          </asp:RegularExpressionValidator>
     </EditItemTemplate>
</asp:TemplateField>

<asp:TemplateField HeaderText="Comment">
     <ItemTemplate>
          <asp:Label runat="server" ID="Label2" Text='<%# Eval("Comment") %>'></asp:Label>

     </ItemTemplate>
     <EditItemTemplate>
          <asp:TextBox runat="server" ID="Comment" TextMode="MultiLine"
               Columns="40" Rows="4" Text='<%# Bind("Comment") %>'>

          </asp:TextBox>
     </EditItemTemplate>
</asp:TemplateField>
```

When editing or deleting a user account we will need to know that user's `UserName` property value. Set the GridView's `DataKeyNames` property to "UserName" so that this information is available through the GridView's `DataKeys` collection.

Finally, add a ValidationSummary control to the page and set its `ShowMessageBox` property to True and its `ShowSummary` property to False. With these settings, the ValidationSummary will display a client-side alert if the user attempts to edit a user account with a missing or invalid email address.

```
<asp:ValidationSummary ID="ValidationSummary1"
               runat="server"
               ShowMessageBox="True"
               ShowSummary="False" />
```

We have now completed this page's declarative markup. Our next task is to bind the set of user accounts to the GridView. Add a method named `BindUserGrid` to the `RoleBasedAuthorization.aspx` page's code-behind class that binds the `MembershipUserCollection` returned by `Membership.GetAllUsers` to the `UserGrid` GridView. Call this method from the `Page_Load` event handler on the first page visit.

```
protected void Page_Load(object sender, EventArgs e)
{
     if (!Page.IsPostBack)
          BindUserGrid();
}

private void BindUserGrid()
{
     MembershipUserCollection allUsers = Membership.GetAllUsers();
     UserGrid.DataSource = allUsers;
     UserGrid.DataBind();
}
```

With this code in place, visit the page through a browser. As Figure 7 shows, you should see a GridView listing information about each user account in the system.

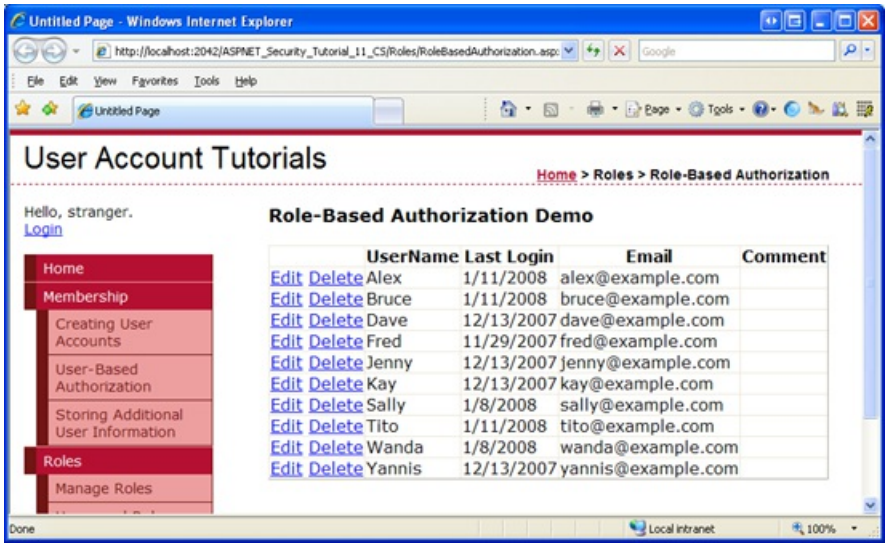


Figure 7: The `UserGrid` GridView Lists Information About Each User in the System (Click to view full-size image)

> **NOTE**
>
> The `UserGrid` GridView lists all of the users in a non-paged interface. This simple grid interface is not suitable for scenarios where there are several dozen or more users. One option is to configure the GridView to enable paging. The `Membership.GetAllUsers` method has two overloads: one that accepts no input parameters and returns all of the users and one that takes in integer values for the page index and page size, and returns only the specified subset of the users. The second overload can be used to more efficiently page through the users since it returns just the precise subset of user accounts rather than *all* of them. If you have thousands of user accounts, you might want to consider a filter-based interface, one that only shows those users whose UserName begins with a selected character, for instance. The `Membership.FindUsersByName method` is ideal for building a filter-based user interface. We will look at building such an interface in a future tutorial.

The GridView control offers built-in editing and deleting support when the control is bound to a properly configured data source control, such as the SqlDataSource or ObjectDataSource. The `UserGrid` GridView, however, has its data programmatically bound; therefore, we must write code to perform these two tasks. In particular, we will need to create event handlers for the GridView's `RowEditing`, `RowCancelingEdit`, `RowUpdating`, and `RowDeleting` events, which are fired when a visitor clicks the GridView's Edit, Cancel, Update, or Delete buttons.

Start by creating the event handlers for the GridView's `RowEditing`, `RowCancelingEdit`, and `RowUpdating` events and then add the following code:

```
protected void UserGrid_RowEditing(object sender, GridViewEditEventArgs e)
{
     // Set the grid's EditIndex and rebind the data

     UserGrid.EditIndex = e.NewEditIndex;
     BindUserGrid();
}

protected void UserGrid_RowCancelingEdit(object sender, GridViewCancelEditEventArgs e)
{
     // Revert the grid's EditIndex to -1 and rebind the data
     UserGrid.EditIndex = -1;
     BindUserGrid();
}

protected void UserGrid_RowUpdating(object sender, GridViewUpdateEventArgs e)
{
     // Exit if the page is not valid
     if (!Page.IsValid)
          return;

     // Determine the username of the user we are editing
     string UserName = UserGrid.DataKeys[e.RowIndex].Value.ToString();

     // Read in the entered information and update the user
     TextBox EmailTextBox = UserGrid.Rows[e.RowIndex].FindControl("Email") as TextBox;
     TextBox CommentTextBox = UserGrid.Rows[e.RowIndex].FindControl("Comment") as TextBox;

     // Return information about the user
     MembershipUser UserInfo = Membership.GetUser(UserName);

     // Update the User account information
     UserInfo.Email = EmailTextBox.Text.Trim();
     UserInfo.Comment = CommentTextBox.Text.Trim();

     Membership.UpdateUser(UserInfo);

     // Revert the grid's EditIndex to -1 and rebind the data
     UserGrid.EditIndex = -1;
     BindUserGrid();
}
```

The `RowEditing` and `RowCancelingEdit` event handlers simply set the GridView's `EditIndex` property and then rebind the list of user accounts to the grid. The interesting stuff happens in the `RowUpdating` event handler. This event handler starts by ensuring that the data is valid and then grabs the `UserName` value of the edited user account from the `DataKeys` collection. The `Email` and `Comment` TextBoxes in the two TemplateFields' `EditItemTemplate` s are then programmatically referenced. Their `Text` properties contain the edited email address and comment.

In order to update a user account through the Membership API we need to first get the user's information, which we do via a call to `Membership.GetUser(userName)`. The returned `MembershipUser` object's `Email` and `Comment` properties are then updated with the values entered into the two TextBoxes from the editing interface. Finally, these modifications are saved with a call to `Membership.UpdateUser`. The `RowUpdating` event handler completes by reverting the GridView to its pre-editing interface.

Next, create the `RowDeleting` event handler and then add the following code:

```
protected void UserGrid_RowDeleting(object sender, GridViewDeleteEventArgs e)
{
     // Determine the username of the user we are editing
     string UserName = UserGrid.DataKeys[e.RowIndex].Value.ToString();

     // Delete the user
     Membership.DeleteUser(UserName);

     // Revert the grid's EditIndex to -1 and rebind the data
     UserGrid.EditIndex = -1;
     BindUserGrid();
}
```

The above event handler starts by grabbing the `UserName` value from the GridView's `DataKeys` collection; this `UserName` value is then passed into the Membership class's `DeleteUser` method. The `DeleteUser` method deletes the user account from the system, including related membership data (such as what roles this user belongs to). After deleting the user, the grid's `EditIndex` is set to -1 (in case the user clicked Delete while another row was in edit mode) and the `BindUserGrid` method is called.

> **NOTE**
> The Delete button does not require any sort of confirmation from the user before deleting the user account. I encourage you to add some form of user confirmation to lessen the chance of an account being accidentally deleted. One of the easiest ways to confirm an action is through a client-side confirm dialog box. For more information on this technique, see Adding Client-Side Confirmation When Deleting.

Verify that this page functions as expected. You should be able to edit any user's email address and comment, as well as delete any user account. Since the `RoleBasedAuthorization.aspx` page is accessible to all users, any user – even anonymous visitors – can visit this page and edit and delete user accounts! Let's update this page so that only users in the Supervisors and Administrators roles can edit a user's email address and comment, and only Administrators can delete a user account.

The "Using the LoginView Control" section looks at using the LoginView control to show instructions specific to the user's role. If a person in the Administrators role visits this page, we will show instructions on how to edit and delete users. If a user in the Supervisors role reaches this page, we will show instructions on editing users. And if the visitor is anonymous or is not in either the Supervisors or Administrators role, we will display a message explaining that they cannot edit or delete user account information. In the "Programmatically Limiting Functionality" section we will write code that programmatically shows or hides the Edit and Delete buttons based on the user's role.

Using the LoginView Control

As we've seen in past tutorials, the LoginView control is useful for displaying different interfaces for authenticated and anonymous users, but the LoginView control can also be used to display different markup based on the user's roles. Let's use a LoginView control to display different instructions based on the visiting user's role.

Start by adding a LoginView above the `UserGrid` GridView. As we've discussed earlier, the LoginView control has two built-in templates: `AnonymousTemplate` and `LoggedInTemplate`. Enter a brief message in both of these templates that informs the user that they cannot edit or delete any user information.

```
<asp:LoginView ID="LoginView1" runat="server">
     <LoggedInTemplate>
          You are not a member of the Supervisors or Administrators roles. Therefore you
          cannot edit or delete any user information.
     </LoggedInTemplate>
     <AnonymousTemplate>
          You are not logged into the system. Therefore you cannot edit or delete any user

          information.
     </AnonymousTemplate>
</asp:LoginView>
```

In addition to the `AnonymousTemplate` and `LoggedInTemplate`, the LoginView control can include *RoleGroups*, which are role-specific templates. Each RoleGroup contains a single property, `Roles`, which specifies what roles the RoleGroup applies to. The `Roles` property can be set to a single role (like "Administrators") or to a comma-delimited list of roles (like "Administrators, Supervisors").

To manage the RoleGroups, click the "Edit RoleGroups" link from the control's Smart Tag to bring up the RoleGroup Collection Editor. Add two new RoleGroups. Set the first RoleGroup's `Roles` property to "Administrators" and the second's to "Supervisors".

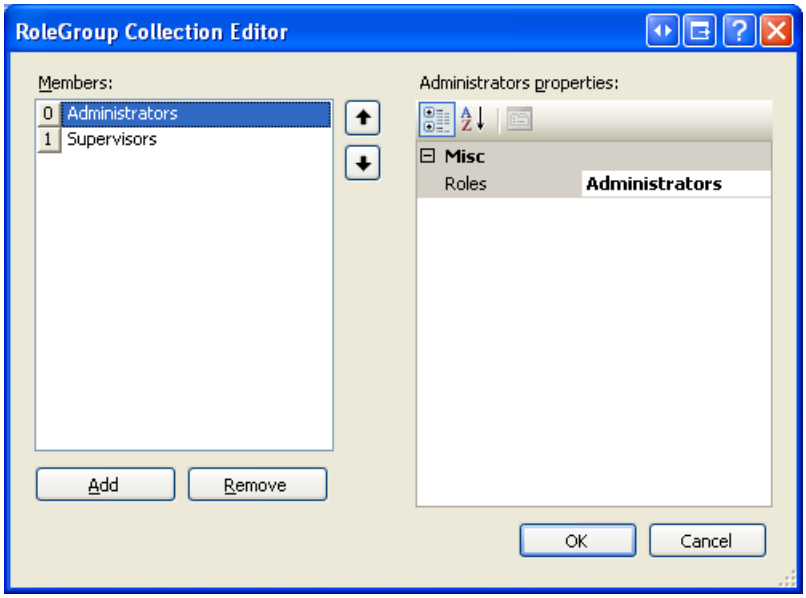


Figure 8: Manage the LoginView's Role-Specific Templates Through the RoleGroup Collection Editor (Click to view full-size image)

Click OK to close the RoleGroup Collection Editor; this updates the LoginView's declarative markup to include a `<RoleGroups>` section with an `<asp:RoleGroup>` child element for each RoleGroup defined in the RoleGroup Collection Editor. Furthermore, the "Views" drop-down list in the LoginView's Smart Tag - which initially listed just the `AnonymousTemplate` and `LoggedInTemplate` – now includes the added RoleGroups as well.

Edit the RoleGroups so that users in the Supervisors role are displayed instructions on how to edit user accounts, while users in the Administrators role are shown instructions for editing and deleting. After making these changes, your LoginView's declarative markup should look similar to the following.

```
<asp:LoginView ID="LoginView1" runat="server">
     <RoleGroups>
          <asp:RoleGroup Roles="Administrators">
               <ContentTemplate>
                    As an Administrator, you may edit and delete user accounts.
                    Remember: With great power comes great responsibility!

               </ContentTemplate>
          </asp:RoleGroup>
          <asp:RoleGroup Roles="Supervisors">
               <ContentTemplate>
                    As a Supervisor, you may edit users&#39; Email and Comment information.
                    Simply click the Edit button, make your changes, and then click Update.
               </ContentTemplate>
          </asp:RoleGroup>
     </RoleGroups>

     <LoggedInTemplate>
          You are not a member of the Supervisors or Administrators roles.
          Therefore you cannot edit or delete any user information.
     </LoggedInTemplate>
     <AnonymousTemplate>
          You are not logged into the system. Therefore you cannot edit or delete any user
          information.
     </AnonymousTemplate>
</asp:LoginView>
```

After making these changes, save the page and then visit it through a browser. First visit the page as an anonymous user. You should be shown the message, "You are not logged into the system. Therefore you cannot edit or delete any user information." Then log in as an authenticated user, but one that is neither in the Supervisors nor Administrators role. This time you should see the message, "You are not a member of the Supervisors or Administrators roles. Therefore you cannot edit or delete any user information."

Next, log in as a user who is a member of the Supervisors role. This time you should see the Supervisors role-specific message (see Figure 9). And if you log in as a user in the Administrators role you should see the Administrators role-specific message (see Figure 10).

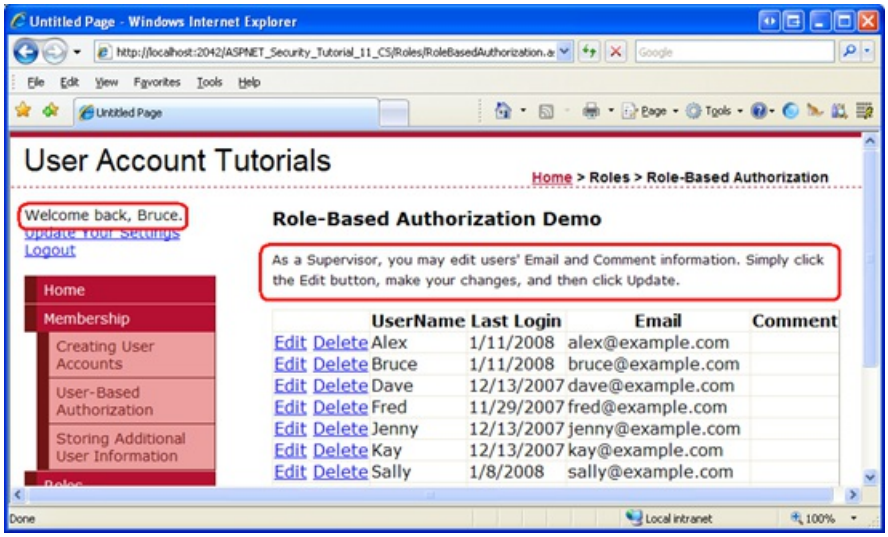


Figure 9: Bruce is Shown the Supervisors Role-Specific Message (Click to view full-size image)

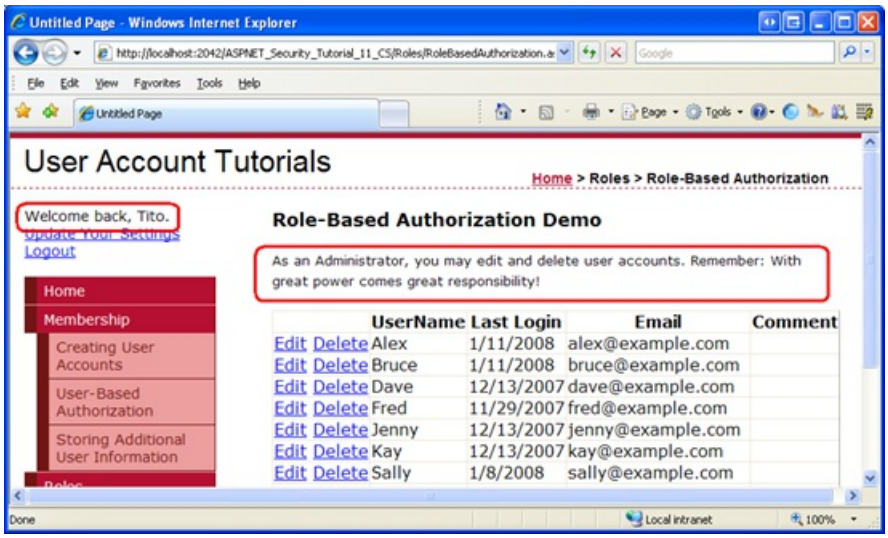


Figure 10: Tito is Shown the Administrators Role-Specific Message (Click to view full-size image)

As the screen shots in Figures 9 and 10 show, the LoginView only renders one template, even if multiple templates apply. Bruce and Tito are both logged in users, yet the LoginView renders only the matching RoleGroup and not the `LoggedInTemplate`. Moreover, Tito belongs to both the Administrators and Supervisors roles, yet the LoginView control renders the Administrators role-specific template instead of the Supervisors one.

Figure 11 illustrates the workflow used by the LoginView control to determine what template to render. Note that if there is more than one RoleGroup specified, the LoginView template renders the *first* RoleGroup that matches. In other words, if we had placed the Supervisors RoleGroup as the first RoleGroup and the Administrators as the second, then when Tito visited this page he would see the Supervisors message.

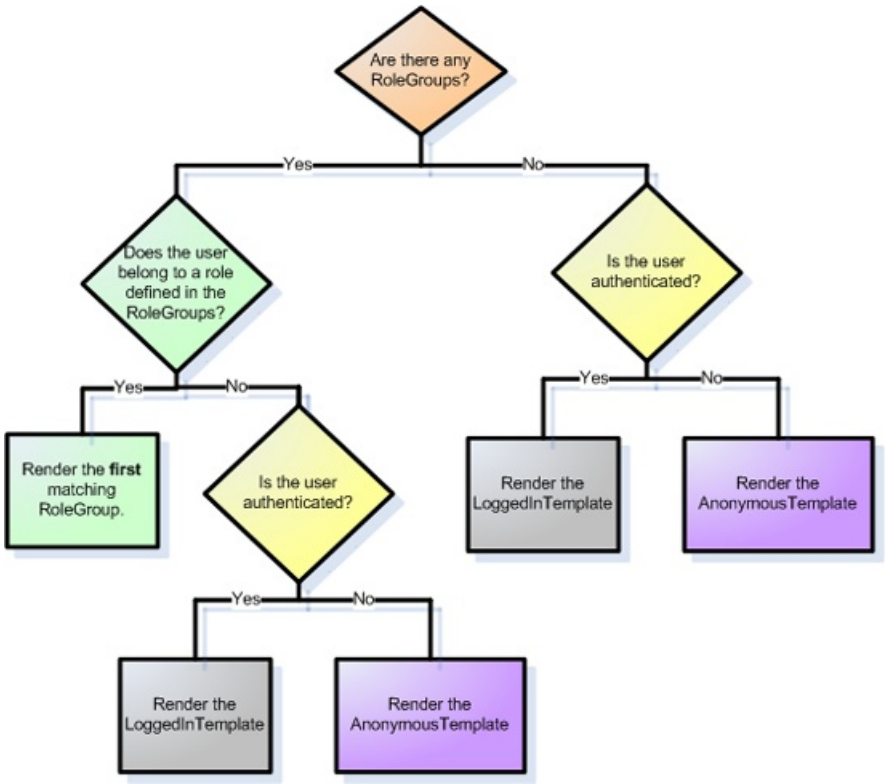


Figure 11: The LoginView Control's Workflow for Determining What Template to Render (Click to view full-size image)

Programmatically Limiting Functionality

While the LoginView control displays different instructions based on the role of the user visiting the page, the Edit and Cancel buttons remain visible to all. We need to programmatically hide the Edit and Delete buttons for

anonymous visitors and users who are in neither the Supervisors nor Administrators role. We need to hide the Delete button for everyone who is not an Administrator. In order to accomplish this we will write a bit of code that programmatically references the CommandField's Edit and Delete LinkButtons and sets their `Visible` properties to `false`, if necessary.

The easiest way to programmatically reference controls in a CommandField is to first convert it to a template. To accomplish this, click the "Edit Columns" link from the GridView's Smart Tag, select the CommandField from the list of current fields, and click the "Convert this field into a TemplateField" link. This turns the CommandField into a TemplateField with an `ItemTemplate` and `EditItemTemplate`. The `ItemTemplate` contains the Edit and Delete LinkButtons while the `EditItemTemplate` houses the Update and Cancel LinkButtons.

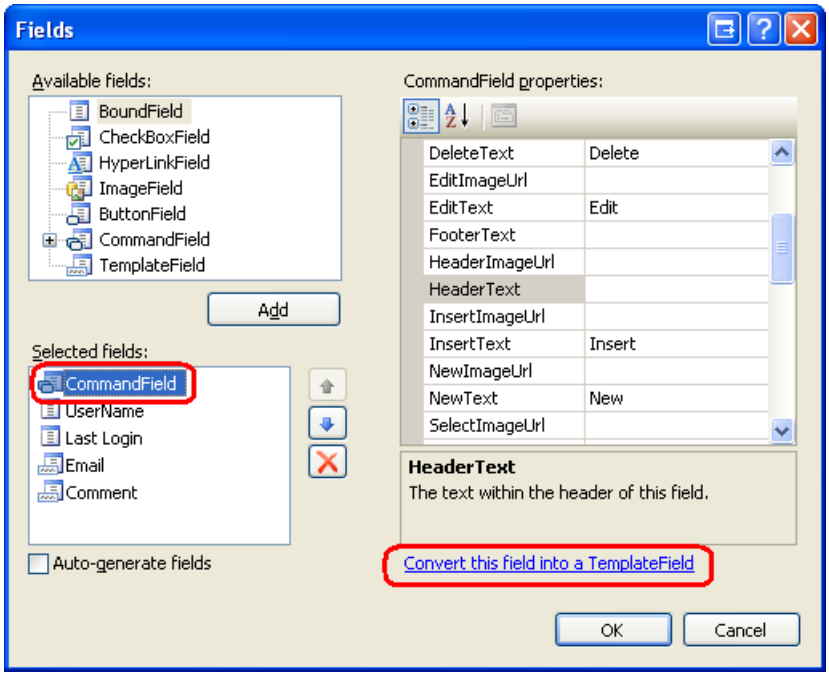


Figure 12: Convert the CommandField Into a TemplateField (Click to view full-size image)

Update the Edit and Delete LinkButtons in the `ItemTemplate`, setting their `ID` properties to values of `EditButton` and `DeleteButton`, respectively.

```
<asp:TemplateField ShowHeader="False">
     <EditItemTemplate>
          <asp:LinkButton ID="LinkButton1" runat="server" CausesValidation="True"
               CommandName="Update" Text="Update"></asp:LinkButton>

           <asp:LinkButton ID="LinkButton2" runat="server" CausesValidation="False"
                CommandName="Cancel" Text="Cancel"></asp:LinkButton>

     </EditItemTemplate>
     <ItemTemplate>
          <asp:LinkButton ID="EditButton" runat="server" CausesValidation="False"
               CommandName="Edit" Text="Edit"></asp:LinkButton>

           <asp:LinkButton ID="DeleteButton" runat="server" CausesValidation="False"
               CommandName="Delete" Text="Delete"></asp:LinkButton>

     </ItemTemplate>
</asp:TemplateField>
```

Whenever data is bound to the GridView, the GridView enumerates the records in its `DataSource` property and generates a corresponding `GridViewRow` object. As each `GridViewRow` object is created, the `RowCreated` event is fired. In order to hide the Edit and Delete buttons for unauthorized users, we need to create an event handler for this event and programmatically reference the Edit and Delete LinkButtons, setting their `Visible` properties

accordingly.

Create an event handler the `RowCreated` event and then add the following code:

```
protected void UserGrid_RowCreated(object sender, GridViewRowEventArgs e)
{
     if (e.Row.RowType == DataControlRowType.DataRow && e.Row.RowIndex != UserGrid.EditIndex)
     {
          // Programmatically reference the Edit and Delete LinkButtons
          LinkButton EditButton = e.Row.FindControl("EditButton") as LinkButton;

          LinkButton DeleteButton = e.Row.FindControl("DeleteButton") as LinkButton;

          EditButton.Visible = (User.IsInRole("Administrators") || User.IsInRole("Supervisors"));
          DeleteButton.Visible = User.IsInRole("Administrators");
     }
}
```

Keep in mind that the `RowCreated` event fires for *all* of the GridView rows, including the header, the footer, the pager interface, and so forth. We only want to programmatically reference the Edit and Delete LinkButtons if we are dealing with a data row not in edit mode (since the row in edit mode has Update and Cancel buttons instead of Edit and Delete). This check is handled by the `if` statement.

If we are dealing with a data row that is not in edit mode, the Edit and Delete LinkButtons are referenced and their `Visible` properties are set based on the Boolean values returned by the `User` object's `IsInRole(roleName)` method. The User object references the principal created by the `RoleManagerModule`; consequently, the `IsInRole(roleName)` method uses the Roles API to determine whether the current visitor belongs to *roleName*.

> **NOTE**
> We could have used the Roles class directly, replacing the call to `User.IsInRole(roleName)` with a call to the `Roles.IsUserInRole(roleName)` method. I decided to use the principal object's `IsInRole(roleName)` method in this example because it is more efficient than using the Roles API directly. Earlier in this tutorial we configured the role manager to cache the user's roles in a cookie. This cached cookie data is only utilized when the principal's `IsInRole(roleName)` method is called; direct calls to the Roles API always involve a trip to the role store. Even if roles are not cached in a cookie, calling the principal object's `IsInRole(roleName)` method is usually more efficient because when it is called for the first time during a request it caches the results. The Roles API, on the other hand, does not perform any caching. Because the `RowCreated` event is fired once for every row in the GridView, using `User.IsInRole(roleName)` involves just one trip to the role store whereas `Roles.IsUserInRole(roleName)` requires *N* trips, where *N* is the number of user accounts displayed in the grid.

The Edit button's `Visible` property is set to `true` if the user visiting this page is in the Administrators or Supervisors role; otherwise it is set to `false`. The Delete button's `Visible` property is set to `true` only if the user is in the Administrators role.

Test this page through a browser. If you visit the page as an anonymous visitor or as a user that is neither a Supervisor nor an Administrator, the CommandField is empty; it still exists, but as a thin sliver without the Edit or Delete buttons.

> **NOTE**
> It is possible to hide the CommandField altogether when a non-Supervisor and non-Administrator is visiting the page. I leave this as an exercise for the reader.

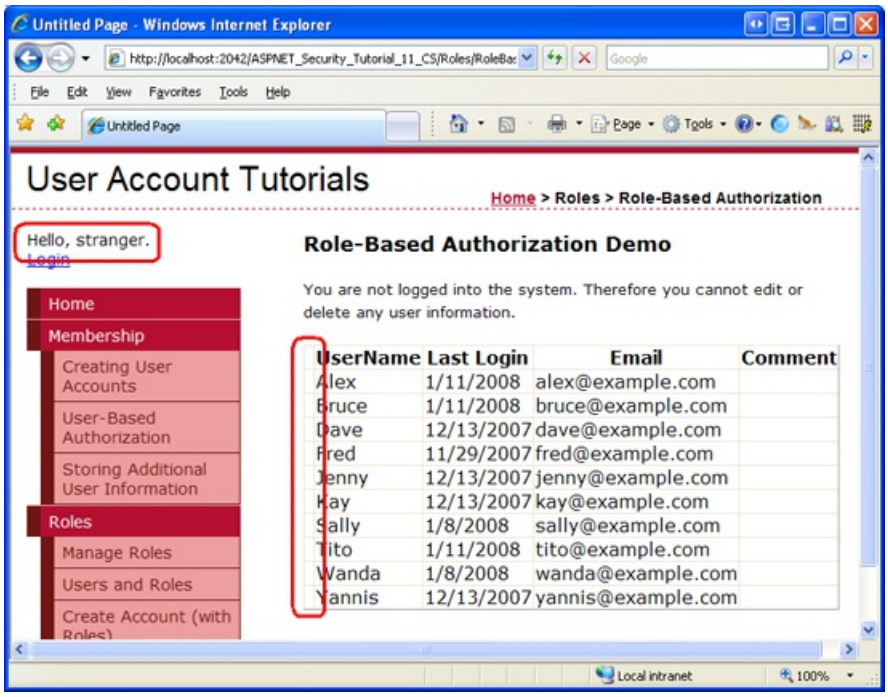


Figure 13: The Edit and Delete Buttons are Hidden for Non-Supervisors and Non-Administrators (Click to view full-size image)

If a user that belongs to the Supervisors role (but not to the Administrators role) visits, he sees only the Edit button.

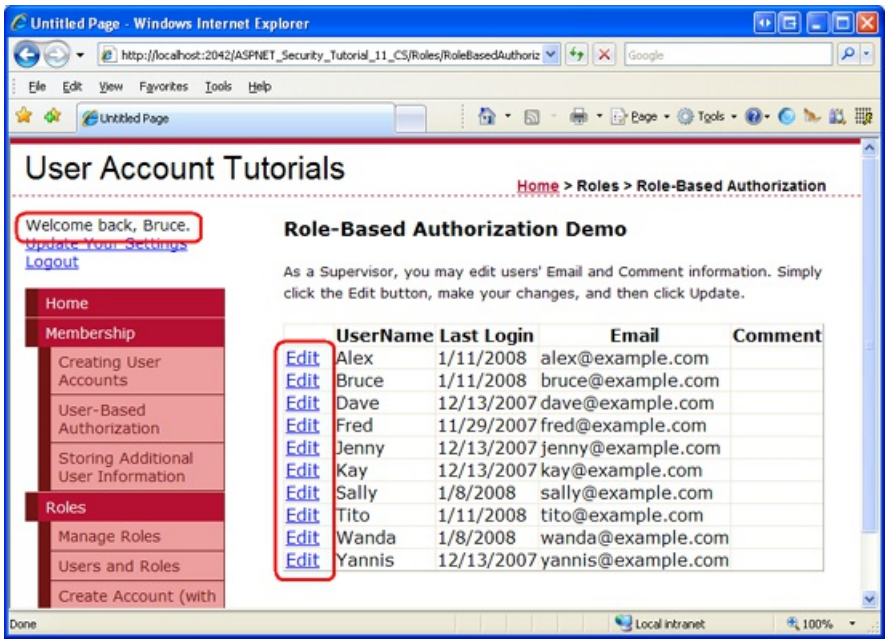


Figure 14: While the Edit Button is Available for Supervisors, the Delete Button is Hidden (Click to view full-size image)

And if an Administrator visits, she has access to both the Edit and Delete buttons.

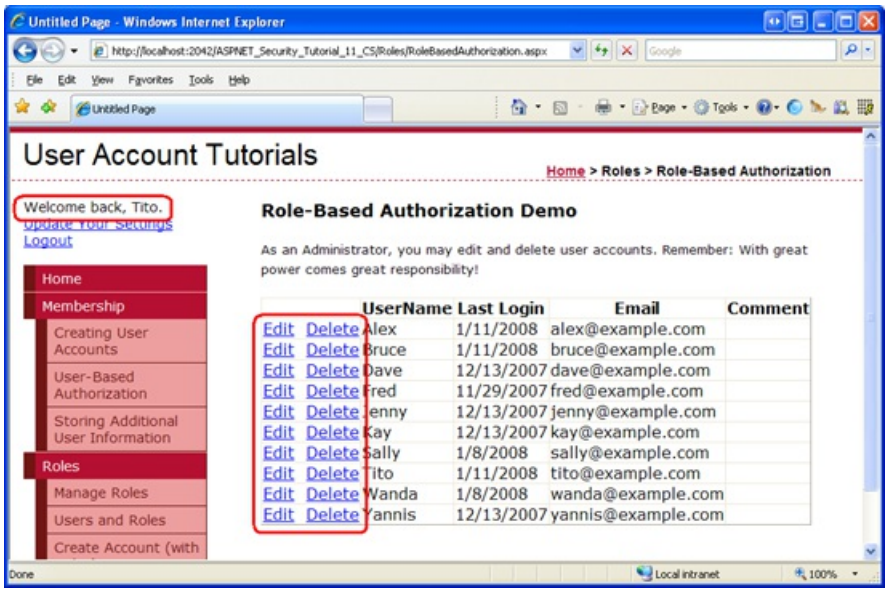


Figure 15: The Edit and Delete Buttons are Available Only for Administrators (Click to view full-size image)

Step 3: Applying Role-Based Authorization Rules to Classes and Methods

In Step 2 we limited edit capabilities to users in the Supervisors and Administrators roles and delete capabilities to Administrators only. This was accomplished by hiding the associated user interface elements for unauthorized users through programmatic techniques. Such measures do not guarantee that an unauthorized user will be unable to perform a privileged action. There may be user interface elements that are added later or that we forgot to hide for unauthorized users. Or a hacker may discover some other way to get the ASP.NET page to execute the desired method.

An easy way to ensure that a particular piece of functionality cannot be accessed by an unauthorized user is to decorate that class or method with the `PrincipalPermission` attribute. When the .NET runtime uses a class or executes one of its methods, it checks to ensure that the current security context has permission. The `PrincipalPermission` attribute provides a mechanism through which we can define these rules.

We looked at using the `PrincipalPermission` attribute back in the *User-Based Authorization* tutorial. Specifically, we saw how to decorate the GridView's `SelectedIndexChanged` and `RowDeleting` event handler so that they could only be executed by authenticated users and Tito, respectively. The `PrincipalPermission` attribute works just as well with roles.

Let's demonstrate using the `PrincipalPermission` attribute on the GridView's `RowUpdating` and `RowDeleting` event handlers to prohibit execution for non-authorized users. All we need to do is add the appropriate attribute atop each function definition:

```
[PrincipalPermission(SecurityAction.Demand, Role = "Administrators")]
[PrincipalPermission(SecurityAction.Demand, Role = "Supervisors")]
protected void UserGrid_RowUpdating(object sender, GridViewUpdateEventArgs e)
{
    ...
}

[PrincipalPermission(SecurityAction.Demand, Role = "Administrators")]
protected void UserGrid_RowDeleting(object sender, GridViewDeleteEventArgs e)
{
    ...
}
```

The attribute for the `RowUpdating` event handler dictates that only users in the Administrators or Supervisors

roles can execute the event handler, where as the attribute on the `RowDeleting` event handler limits the execution to users in the Administrators role.

> **NOTE**
> The `PrincipalPermission` attribute is represented as a class in the `System.Security.Permissions` namespace. Be sure to add a `using System.Security.Permissions` statement at the top of your code-behind class file to import this namespace.

If, somehow, a non-Administrator attempts to execute the `RowDeleting` event handler or if a non-Supervisor or non-Administrator attempts to execute the `RowUpdating` event handler, the .NET runtime will raise a `SecurityException`.

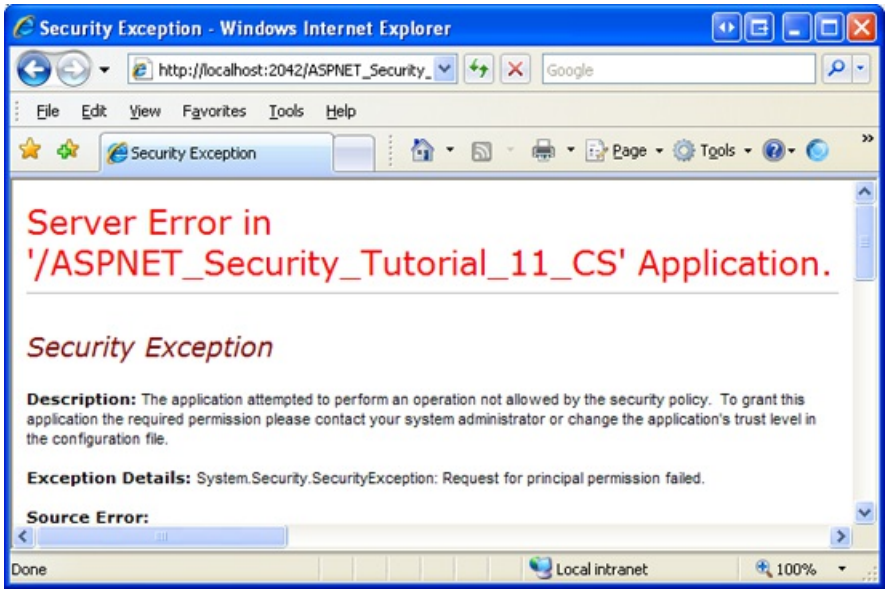


Figure 16: If the Security Context is not Authorized to Execute the Method, a `SecurityException` is Thrown (Click to view full-size image)

In addition to ASP.NET pages, many applications also have an architecture that includes various layers, such as Business Logic and Data Access Layers. These layers are typically implemented as Class Libraries and offer classes and methods for performing business logic- and data-related functionality. The `PrincipalPermission` attribute is useful for applying authorization rules to these layers as well.

For more information on using the `PrincipalPermission` attribute to define authorization rules on classes and methods, refer to Scott Guthrie's blog entry Adding Authorization Rules to Business and Data Layers Using `PrincipalPermissionAttributes`.

Summary

In this tutorial we looked at how to specify coarse and fine grain authorization rules based on the user's roles. ASP.NET's URL authorization feature allows a page developer to specify what identities are allowed or denied access to what pages. As we saw back in the *User-Based Authorization* tutorial, URL authorization rules can be applied on a user-by-user basis. They can also be applied on a role-by-role basis, as we saw in Step 1 of this tutorial.

Fine grain authorization rules may be applied declaratively or programmatically. In Step 2 we looked at using the LoginView control's RoleGroups feature to render different output based on the visiting user's roles. We also looked at ways to programmatically determine if a user belongs to a specific role and how to adjust the page's functionality accordingly.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Adding Authorization Rules to Business and Data Layers Using `PrincipalPermissionAttributes`
- Examining ASP.NET 2.0's Membership, Roles, and Profile: Working with Roles
- Security Question List for ASP.NET 2.0
- Technical documentation for the `<roleManager>` Element

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial include Suchi Banerjee and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Creating and Managing Roles (VB)

2/20/2020 • 20 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

This tutorial examines the steps necessary for configuring the Roles framework. Following that, we will build web pages to create and delete roles.

Introduction

In the *User-Based Authorization* tutorial we looked at using URL authorization to restrict certain users from a set of pages and explored declarative and programmatic techniques for adjusting an ASP.NET page's functionality based on the visiting user. Granting permission for page access or functionality on a user-by-user basis, however, can become a maintenance nightmare in scenarios where there are many user accounts or when users' privileges change often. Any time a user gains or loses authorization to perform a particular task, the administrator needs to update the appropriate URL authorization rules, declarative markup, and code.

It usually helps to classify users into groups or *roles* and then to apply permissions on a role-by-role basis. For example, most web applications have a certain set of pages or tasks that are reserved only for administrative users. Using the techniques learned in the *User-Based Authorization* tutorial, we would add the appropriate URL authorization rules, declarative markup, and code to allow the specified user accounts to perform administrative tasks. But if a new administrator was added or if an existing administrator needed to have her administration rights revoked, we would have to return and update the configuration files and web pages. With roles, however, we could create a role called Administrators and assign those trusted users to the Administrators role. Next, we would add the appropriate URL authorization rules, declarative markup, and code to allow the Administrators role to perform the various administrative tasks. With this infrastructure in place, adding new administrators to the site or removing existing ones is as simple as including or removing the user from the Administrators role. No configuration, declarative markup, or code changes are necessary.

ASP.NET offers a Roles framework for defining roles and associating them with user accounts. With the Roles framework we can create and delete roles, add users to or remove users from a role, determine the set of users that belong to a particular role, and tell whether a user belongs to a particular role. Once the Roles framework has been configured, we can limit access to pages on a role-by-role basis through URL authorization rules and show or hide additional information or functionality on a page based on the currently logged on user's roles.

This tutorial examines the steps necessary for configuring the Roles framework. Following that, we will build web pages to create and delete roles. In the *Assigning Roles to Users* tutorial we will look at how to add and remove users from roles. And in the *Role-Based Authorization* tutorial we will see how to limit access to pages on a role-by-role basis along with how to adjust page functionality depending on the visiting user's role. Let's get started!

Step 1: Adding New ASP.NET Pages

In this tutorial and the next two we will be examining various roles-related functions and capabilities. We will need a series of ASP.NET pages to implement the topics examined throughout these tutorials. Let's create these pages and update the site map.

Start by creating a new folder in the project named `Roles`. Next, add four new ASP.NET pages to the `Roles` folder, linking each page with the `Site.master` master page. Name the pages:

- `ManageRoles.aspx`
- `UsersAndRoles.aspx`
- `CreateUserWizardWithRoles.aspx`
- `RoleBasedAuthorization.aspx`

At this point your project's Solution Explorer should look similar to the screen shot shown in Figure 1.

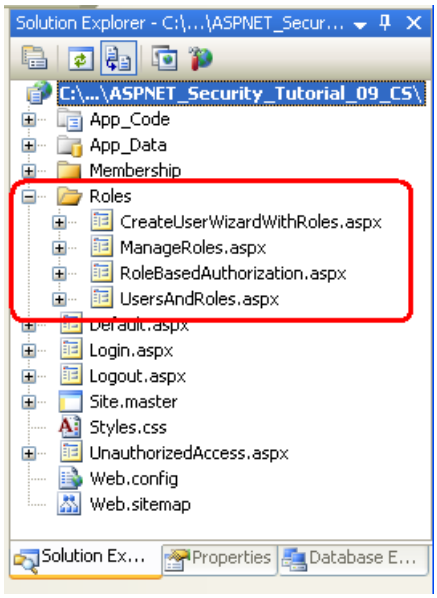


Figure 1: Four New Pages Have Been Added to the `Roles` Folder (Click to view full-size image)

Each page should, at this point, have the two Content controls, one for each of the master page's ContentPlaceHolders: `MainContent` and `LoginContent`.

```
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="LoginContent" Runat="Server">
</asp:Content>
```

Recall that the `LoginContent` ContentPlaceHolder's default markup displays a link to log on or log off the site, depending on whether the user is authenticated. The presence of the `Content2` Content control in the ASP.NET page, however, overrides the master page's default markup. As we discussed in *An Overview of Forms Authentication* tutorial, overriding the default markup is useful in pages where we do not want to display login-related options in the left column.

For these four pages, however, we want to show the master page's default markup for the `LoginContent` ContentPlaceHolder. Therefore, remove the declarative markup for the `Content2` Content control. After doing so, each of the four page's markup should contain just one Content control.

Finally, let's update the site map (`Web.sitemap`) to include these new web pages. Add the following XML after the `<siteMapNode>` we added for the Membership tutorials.

```
<siteMapNode title="Roles">
 <siteMapNode url="~/Roles/ManageRoles.aspx" title="Manage Roles" />
 <siteMapNode url="~/Roles/UsersAndRoles.aspx" title="Users and Roles" />
 <siteMapNode url="~/Roles/CreateUserWizardWithRoles.aspx" title="Create Account (with Roles)"/>
 <siteMapNode url="~/Roles/RoleBasedAuthorization.aspx" title="Role-Based Authorization" />
</siteMapNode>
```

With the site map updated, visit the site through a browser. As Figure 2 shows, the navigation on the left now includes items for the Roles tutorials.

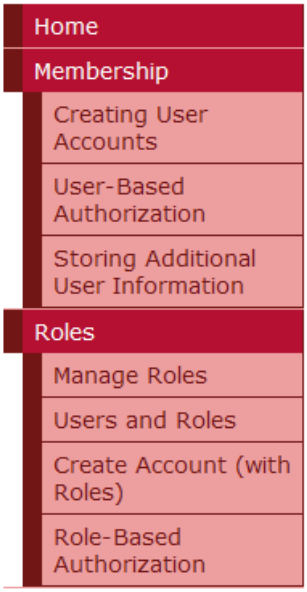


Figure 2: Four New Pages Have Been Added to the `Roles` Folder (Click to view full-size image)

Step 2: Specifying and Configuring the Roles Framework Provider

Like the Membership framework, the Roles framework is built atop the provider model. As discussed in the *Security Basics and ASP.NET Support* tutorial, the .NET Framework ships with three built-in Roles providers: `AuthorizationStoreRoleProvider`, `WindowsTokenRoleProvider`, and `SqlRoleProvider`. This tutorial series focuses on the `SqlRoleProvider`, which uses a Microsoft SQL Server database as the role store.

Underneath the covers the Roles framework and `SqlRoleProvider` work just like the Membership framework and `SqlMembershipProvider`. The .NET Framework contains a `Roles` class that serves as the API to the Roles framework. The `Roles` class has shared methods like `CreateRole`, `DeleteRole`, `GetAllRoles`, `AddUserToRole`, `IsUserInRole`, and so forth. When one of these methods is invoked, the `Roles` class delegates the call to the configured provider. The `SqlRoleProvider` works with the role-specific tables (`aspnet_Roles` and `aspnet_UsersInRoles`) in response.

In order to use the `SqlRoleProvider` provider in our application, we need to specify what database to use as the store. The `SqlRoleProvider` expects the specified role store to have certain database tables, views, and stored procedures. These requisite database objects can be added using the `aspnet_regsql.exe` tool. At this point we already have a database with the schema necessary for the `SqlRoleProvider`. Back in the *Creating the Membership Schema in SQL Server* tutorial we created a database named `SecurityTutorials.mdf` and used `aspnet_regsql.exe` to add the application services, which included the database objects required by the `SqlRoleProvider`. Therefore we just need to tell the Roles framework to enable role support and to use the `SqlRoleProvider` with the `SecurityTutorials.mdf` database as the role store.

The Roles framework is configured via the `<roleManager>` element in the application's `Web.config` file. By default, role support is disabled. To enable it, you must set the `<roleManager>` element's `enabled` attribute to `true` like so:

```
<?xml version="1.0"?>
<configuration>
 <system.web>
 ... Additional configuration markup removed for brevity ...

 <roleManager enabled="true" />
 <system.web>
</configuration>
```

By default, all web applications have a Roles provider named `AspNetSqlRoleProvider` of type `SqlRoleProvider`. This default provider is registered in `machine.config` (located at `%WINDIR%\Microsoft.Net\Framework\v2.0.50727\CONFIG`):

```
<roleManager>
 <providers>
 <add name="AspNetSqlRoleProvider"
 connectionStringName="LocalSqlServer"
 applicationName="/"
 type="System.Web.Security.SqlRoleProvider,
 System.Web, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"/>
 </providers>
</roleManager>
```

The provider's `connectionStringName` attribute specifies the role store that is used. The `AspNetSqlRoleProvider` provider sets this attribute to `LocalSqlServer`, which is also defined in `machine.config` and points, by default, to a SQL Server 2005 Express Edition database in the `App_Data` folder named `aspnet.mdf`.

Consequently, if we simply enable the Roles framework without specifying any provider information in our application's `Web.config` file, the application uses the default registered Roles provider, `AspNetSqlRoleProvider`. If the `~/App_Data/aspnet.mdf` database does not exist, the ASP.NET runtime will automatically create it and add the application services schema. However, we don't want to use the `aspnet.mdf` database; rather, we want to use the `SecurityTutorials.mdf` database that we have already created and added the application services schema to. This modification can be accomplished in one of two ways:

- **Specify a value for the `LocalSqlServer` connection string name in `Web.config`.** By overwriting the `LocalSqlServer` connection string name value in `Web.config`, we can use the default registered Roles provider (`AspNetSqlRoleProvider`) and have it correctly work with the `SecurityTutorials.mdf` database. For more information on this technique, see Scott Guthrie's blog post, Configuring ASP.NET 2.0 Application Services to Use SQL Server 2000 or SQL Server 2005.
- **Add a new registered provider of type `SqlRoleProvider` and configure its `connectionStringName` setting to point to the `SecurityTutorials.mdf` database.** This is the approach I recommended and used in the *Creating the Membership Schema in SQL Server* tutorial, and it is the approach I will use in this tutorial as well.

Add the following Roles configuration markup to the `Web.config` file. This markup registers a new provider named `SecurityTutorialsSqlRoleProvider.`

```
<?xml version="1.0"?>
<configuration>
 <connectionStrings>
 <add name="SecurityTutorialsConnectionString"
 connectionString="..."/>
 </connectionStrings>

 <system.web>
 ... Additional configuration markup removed for brevity ...

    <roleManager enabled="true" defaultProvider="SecurityTutorialsSqlRoleProvider">
      <providers>
 <add name="SecurityTutorialsSqlRoleProvider"
 type="System.Web.Security.SqlRoleProvider"
 applicationName="SecurityTutorials"
 connectionStringName="SecurityTutorialsConnectionString"/>
 </providers>
 </roleManager>
 <system.web>
</configuration>
```

The above markup defines the `SecurityTutorialsSqlRoleProvider` as the default provider (via the `defaultProvider` attribute in the `<roleManager>` element). It also sets the `SecurityTutorialsSqlRoleProvider`'s `applicationName` setting to `SecurityTutorials`, which is the same `applicationName` setting used by the Membership provider (`SecurityTutorialsSqlMembershipProvider`). While not shown here, the `<add>` element for the `SqlRoleProvider` may also contain a `commandTimeout` attribute to specify the database timeout duration, in seconds. The default value is 30.

With this configuration markup in place, we are ready to start using role functionality within our application.

> **NOTE**
> The above configuration markup illustrates using the `<roleManager>` element's `enabled` and `defaultProvider` attributes. There are a number of other attributes that affect how the Roles framework associates role information on a user-by-user basis. We will examine these settings in the *Role-Based Authorization* tutorial.

Step 3: Examining the Roles API

The Roles framework's functionality is exposed via the `Roles` class, which contains thirteen shared methods for performing role-based operations. When we look at creating and deleting roles in Steps 4 and 6 we will use the `CreateRole` and `DeleteRole` methods, which add or remove a role from the system.

To get a list of all of the roles in the system, use the `GetAllRoles` method (see Step 5). The `RoleExists` method returns a Boolean value indicating whether a specified role exists.

In the next tutorial we will examine how to associate users with roles. The `Roles` class's `AddUserToRole`, `AddUserToRoles`, `AddUsersToRole`, and `AddUsersToRoles` methods add one or more users to one or more roles. To remove users from roles, use the `RemoveUserFromRole`, `RemoveUserFromRoles`, `RemoveUsersFromRole`, or `RemoveUsersFromRoles` methods.

In the *Role-Based Authorization* tutorial we will look at ways to programmatically show or hide functionality based on the currently logged in user's role. To accomplish this, we can use the Role class's `FindUsersInRole`, `GetRolesForUser`, `GetUsersInRole`, or `IsUserInRole` methods.

NOTE

Keep in mind that any time one of these methods is invoked, the `Roles` class delegates the call to the configured provider. In our case, this means that the call is being sent to the `SqlRoleProvider`. The `SqlRoleProvider` then performs the appropriate database operation based on the invoked method. For example, the code `Roles.CreateRole("Administrators")` results in the `SqlRoleProvider` executing the `aspnet_Roles_CreateRole` stored procedure, which inserts a new record into the `aspnet_Roles` table named Administrators .

The remainder of this tutorial looks at using the `Roles` class's `CreateRole`, `GetAllRoles`, and `DeleteRole` methods to manage the roles in the system.

Step 4: Creating New Roles

Roles offer a way to arbitrarily group users, and most commonly this grouping is used for a more convenient way to apply authorization rules. But in order to use roles as an authorization mechanism we first need to define what roles exist in the application. Unfortunately, ASP.NET does not include a CreateRoleWizard control. In order to add new roles we need to create a suitable user interface and invoke the Roles API ourselves. The good news is that this is very easy to accomplish.

NOTE

While there is no CreateRoleWizard Web control, there is the ASP.NET Web Site Administration Tool, which is a local ASP.NET application designed to assist with viewing and managing your web application's configuration. However, I am not a big fan of the ASP.NET Web Site Administration Tool for two reasons. First, it is a bit buggy and the user experience leaves a lot to be desired. Second, the ASP.NET Web Site Administration Tool is designed to only work locally, meaning that you will have to build your own role management web pages if you need to manage roles on a live site remotely. For these two reasons, this tutorial and the next will focus on building the necessary role management tools in a web page rather than relying on the ASP.NET Web Site Administration Tool.

Open the `ManageRoles.aspx` page in the `Roles` folder and add a TextBox and a Button Web control to the page. Set the TextBox control's `ID` property to `RoleName` and the Button's `ID` and `Text` properties to `CreateRoleButton` and Create Role, respectively. At this point, your page's declarative markup should look similar to the following:

```
<b>Create a New Role:</b>
<asp:TextBox ID="RoleName" runat="server"></asp:TextBox>
<br />
<asp:Button ID="CreateRoleButton" runat="server" Text="Create Role" />
```

Next, double-click the `CreateRoleButton` Button control in the Designer to create a `Click` event handler and then add the following code:

```
Protected Sub CreateRoleButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
CreateRoleButton.Click
 Dim newRoleName As String = RoleName.Text.Trim()

 If Not Roles.RoleExists(newRoleName) Then
 ' Create the role
 Roles.CreateRole(newRoleName)
 End If

 RoleName.Text= String.Empty
End Sub
```

The above code starts by assigning the trimmed role name entered in the `RoleName` TextBox to the `newRoleName` variable. Next, the `Roles` class's `RoleExists` method is called to determine if the role `newRoleName` already exists in the system. If the role does not exist, it is created via a call to the `CreateRole` method. If the `CreateRole` method is passed a role name that already exists in the system, a `ProviderException` exception is thrown. This is why the code first checks to ensure that the role does not already exist in the system before calling `CreateRole`. The `Click` event handler concludes by clearing out the `RoleName` TextBox's `Text` property.

> **NOTE**
>
> You may be wondering what will happen if the user doesn't enter any value into the `RoleName` TextBox. If the value passed into the `CreateRole` method is `Nothing` or an empty string, an exception is raised. Likewise, if the role name contains a comma an exception is raised. Consequently, the page should contain validation controls to ensure that the user enters a role and that it does not contain any commas. I leave as an exercise for the reader.

Let's create a role named Administrators. Visit the `ManageRoles.aspx` page through a browser, type in Administrators into the textbox (see Figure 3), and then click the Create Role button.

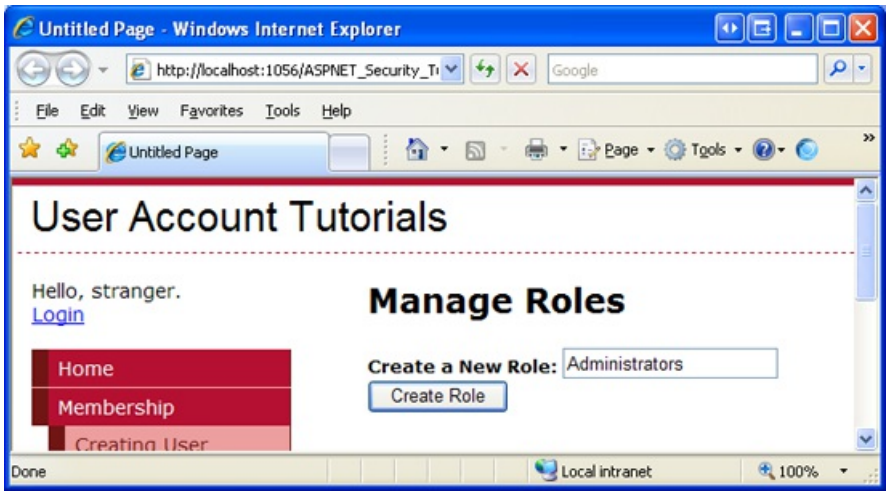


Figure 3: Create an Administrators Role (Click to view full-size image)

What happens? A postback occurs, but there's no visual cue that the role has actually been added to the system. We will update this page in Step 5 to include visual feedback. For now, however, you can verify that the role was created by going to the `SecurityTutorials.mdf` database and displaying the data from the `aspnet_Roles` table. As Figure 4 shows, the `aspnet_Roles` table contains a record for the just-added Administrators roles.

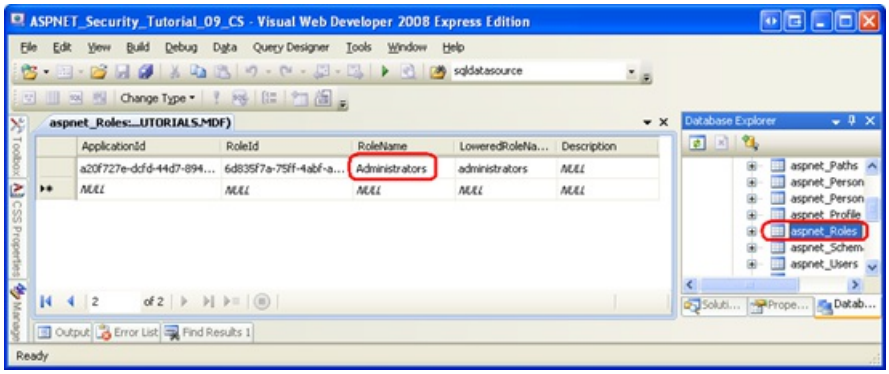


Figure 4: The `aspnet_Roles` Table has a Row for the Administrators (Click to view full-size image)

Step 5: Displaying the Roles in the System

Let's augment the `ManageRoles.aspx` page to include a list of the current roles in the system. To accomplish this, add a GridView control to the page and set its `ID` property to `RoleList`. Next, add a method to the page's code-

behind class named `DisplayRolesInGrid` using the following code:

```
Private Sub DisplayRolesInGrid()
 RoleList.DataSource = Roles.GetAllRoles()
 RoleList.DataBind()
End Sub
```

The `Roles` class's `GetAllRoles` method returns all of the roles in the system as an array of strings. This string array is then bound to the GridView. In order to bind the list of roles to the GridView when the page is first loaded, we need to call the `DisplayRolesInGrid` method from the page's `Page_Load` event handler. The following code calls this method when the page is first visited, but not on subsequent postbacks.

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
 If Not Page.IsPostBack Then
 DisplayRolesInGrid()
 End If
End Sub
```

With this code in place, visit the page through a browser. As Figure 5 shows, you should see a grid with a single column labeled Item. The grid includes a row for the Administrators role we added in Step 4.

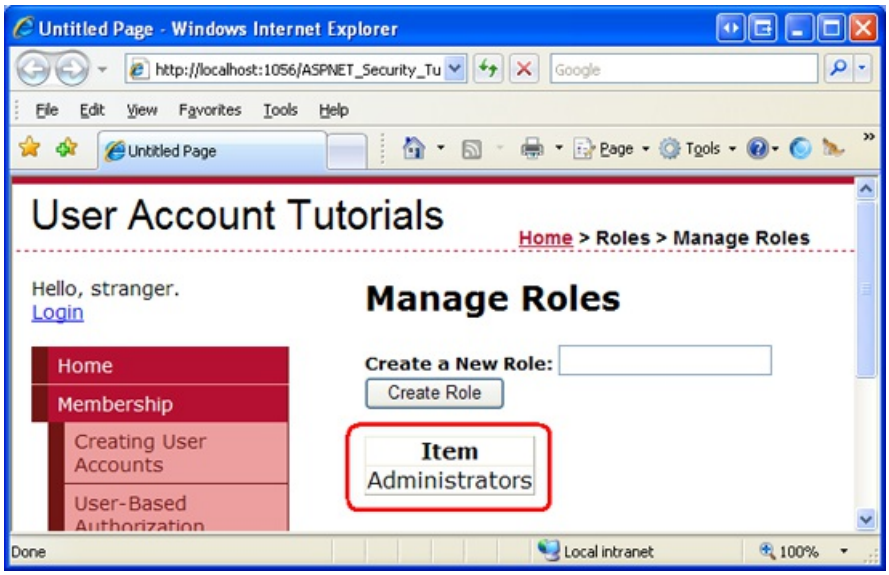


Figure 5: The GridView Displays the Roles in a Single Column (Click to view full-size image)

The GridView displays a lone column labeled Item because the GridView's `AutoGenerateColumns` property is set to True (the default), which causes the GridView to automatically create a column for each property in its `DataSource`. An array has a single property that represents the elements in the array, hence the single column in the GridView.

When displaying data with a GridView, I prefer to explicitly define my columns rather than have them implicitly generated by the GridView. By explicitly defining the columns it is much easier to format the data, rearrange the columns, and perform other common tasks. Therefore, let's update the GridView's declarative markup so that its columns are explicitly defined.

Start by setting the GridView's `AutoGenerateColumns` property to False. Next, add a TemplateField to the grid, set its `HeaderText` property to Roles, and configure its `ItemTemplate` so that it displays the contents of the array. To accomplish this, add a Label Web control named `RoleNameLabel` to the `ItemTemplate` and bind its `Text` property to `Container.DataItem.`

These properties and the `ItemTemplate`'s contents can be set declaratively or through the GridView's Fields dialog box and Edit Templates interface. To reach the Fields dialog box, click the Edit Columns link in the GridView's

Smart Tag. Next, uncheck the Auto-generate fields checkbox to set the `AutoGenerateColumns` property to False, and add a TemplateField to the GridView, setting its `HeaderText` property to Role. To define the `ItemTemplate`'s contents, choose the Edit Templates option from the GridView's Smart Tag. Drag a Label Web control onto the `ItemTemplate`, set its `ID` property to `RoleNameLabel`, and configure its databinding settings so that its `Text` property is bound to `Container.DataItem`.

Regardless of what approach you use, the GridView's resulting declarative markup should look similar to the following when you are done.

```
<asp:GridView ID="RoleList" runat="server" AutoGenerateColumns="false">
 <Columns>
 <asp:TemplateField HeaderText="Role">
 <ItemTemplate>
 <asp:Label runat="server" ID="RoleNameLabel" Text='<%# Container.DataItem %>' />
 </ItemTemplate>
 </asp:TemplateField>
 </Columns>
</asp:GridView>
```

NOTE

The array's contents are displayed using the databinding syntax `<%# Container.DataItem %>`. A thorough description of why this syntax is used when displaying the contents of an array bound to the GridView is beyond the scope of this tutorial. For more information on this matter, refer to Binding a Scalar Array to a Data Web Control.

Currently, the `RoleList` GridView is only bound to the list of roles when the page is first visited. We need to refresh the grid whenever a new role is added. To accomplish this, update the `CreateRoleButton` Button's `Click` event handler so that it calls the `DisplayRolesInGrid` method if a new role is created.

```
Protected Sub CreateRoleButton_Click(ByVal sender As Object,ByVal e As System.EventArgs) Handles
CreateRoleButton.Click
 Dim newRoleName As String = RoleName.Text.Trim()

 If Not Roles.RoleExists(newRoleName) Then
 ' Create the role
 Roles.CreateRole(newRoleName)

 ' Refresh the RoleList Grid
 DisplayRolesInGrid()
 End If

 RoleName.Text = String.Empty
End Sub
```

Now when the user adds a new role the `RoleList` GridView shows the just-added role on postback, providing visual feedback that the role was successfully created. To illustrate this, visit the `ManageRoles.aspx` page through a browser and add a role named Supervisors. Upon clicking the Create Role button, a postback will ensue and the grid will update to include Administrators as well as the new role, Supervisors.

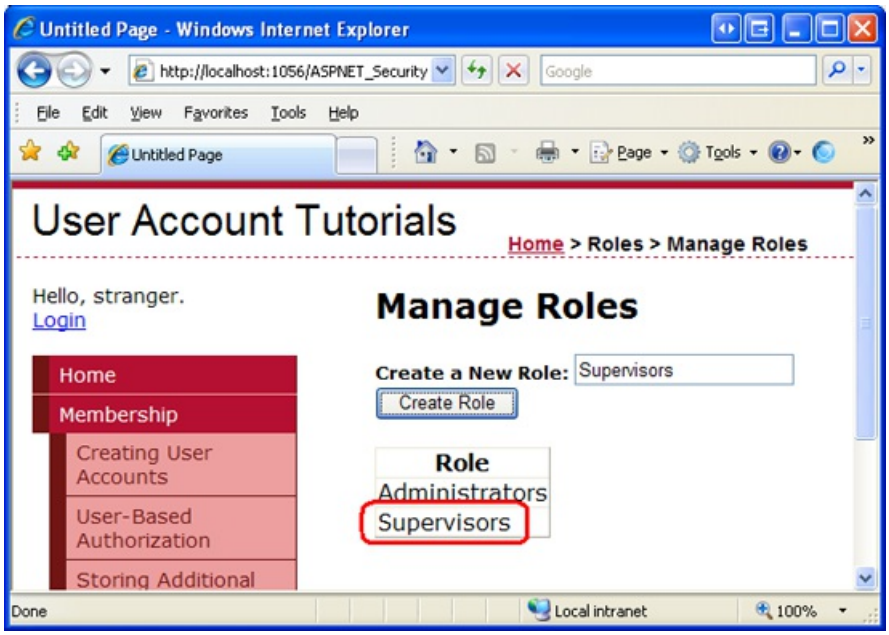


Figure 6: The Supervisors Role has Been Added (Click to view full-size image)

Step 6: Deleting Roles

At this point a user can create a new role and view all existing roles from the `ManageRoles.aspx` page. Let's allow users to also delete roles. The `Roles.DeleteRole` method has two overloads:

- `DeleteRole(roleName)` - deletes the role *roleName*. An exception is thrown if the role contains one or more members.
- `DeleteRole(roleName, throwOnPopulatedRole)` - deletes the role *roleName*. If *throwOnPopulateRole* is `True`, then an exception is thrown if the role contains one or more members. If *throwOnPopulateRole* is `False`, then the role is deleted whether it contains any members or not. Internally, the `DeleteRole(roleName)` method calls `DeleteRole(roleName, True)`.

The `DeleteRole` method will also throw an exception if *roleName* is `Nothing` or an empty string or if *roleName* contains a comma. If *roleName* does not exist in the system, `DeleteRole` fails silently, without raising an exception.

Let's augment the GridView in `ManageRoles.aspx` to include a Delete button that, when clicked, deletes the selected role. Start by adding a Delete button to the GridView by going to the Fields dialog box and adding a Delete button, which is located under the CommandField option. Make the Delete button the far left column and set its `DeleteText` property to Delete Role .

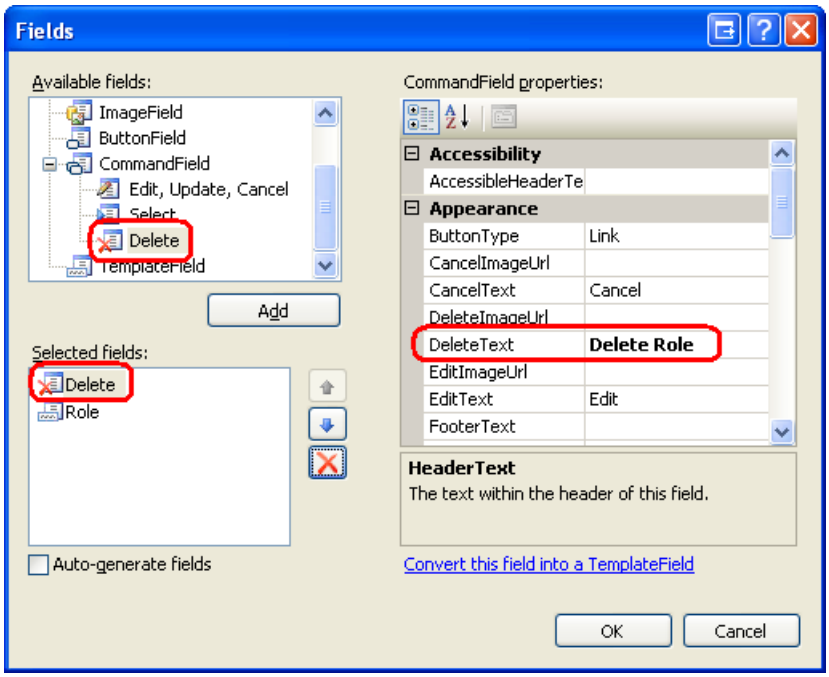


Figure 7: Add a Delete Button to the `RoleList` GridView (Click to view full-size image)

After adding the Delete button, your GridView's declarative markup should look similar to the following:

```
<asp:GridView ID="RoleList" runat="server" AutoGenerateColumns="False">
 <Columns>
 <asp:CommandField DeleteText="Delete Role" ShowDeleteButton="True" />
 <asp:TemplateField HeaderText="Role">
 <ItemTemplate>
 <asp:Label runat="server" ID="RoleNameLabel" Text='<%# Container.DataItem %>' />
 </ItemTemplate>
 </asp:TemplateField>
 </Columns>
</asp:GridView>
```

Next, create an event handler for the GridView's `RowDeleting` event. This is the event that is raised on postback when the Delete Role button is clicked. Add the following code to the event handler.

```
Protected Sub RoleList_RowDeleting(ByVal sender As Object,ByVal e As
System.Web.UI.WebControls.GridViewDeleteEventArgs) Handles RoleList.RowDeleting
 ' Get the RoleNameLabel
 Dim RoleNameLabel As Label = CType(RoleList.Rows(e.RowIndex).FindControl("RoleNameLabel"),Label)

 ' Delete the role
 Roles.DeleteRole(RoleNameLabel.Text,False)

 ' Rebind the data to the RoleList grid
 DisplayRolesInGrid()
End Sub
```

The code starts by programmatically referencing the `RoleNameLabel` Web control in the row whose Delete Role button was clicked. The `Roles.DeleteRole` method is then invoked, passing in the `Text` of the `RoleNameLabel` and `False`, thereby deleting the role regardless of whether there are any users associated with the role. Finally, the `RoleList` GridView is refreshed so that the just-deleted role no longer appears in the grid.

NOTE

The Delete Role button does not require any sort of confirmation from the user before deleting the role. One of the easiest ways to confirm an action is through a client-side confirm dialog box. For more information on this technique, see Adding Client-Side Confirmation When Deleting.

Summary

Many web applications have certain authorization rules or page-level functionality that is only available to certain classes of users. For example, there may be a set of web pages that only administrators can access. Rather than defining these authorization rules on a user-by-user basis, oftentimes it is more useful to define the rules based on a role. That is, rather than explicitly allowing users Scott and Jisun to access the administrative web pages, a more maintainable approach is to permit members of the Administrators role to access these pages, and then to denote Scott and Jisun as users belonging to the Administrators role.

The Roles framework makes it easy to create and manage roles. In this tutorial we examined how to configure the Roles framework to use the `SqlRoleProvider`, which uses a Microsoft SQL Server database as the role store. We also created a web page that lists the existing roles in the system and allows for new roles to be created and existing ones to be deleted. In subsequent tutorials we will see how to assign users to roles and how to apply role-based authorization.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Examining ASP.NET 2.0's Membership, Roles, and Profile
- How To: Use Role Manager in ASP.NET 2.0
- Role Providers
- Rolling Your Own Website Administration Tool
- Technical documentation for the `<roleManager>` Element
- Using the Membership and Role Manager APIs

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial include Alicja Maziarz, Suchi Banerjee, and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

Assigning Roles to Users (VB)

2/20/2020 • 29 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

In this tutorial we will build two ASP.NET pages to assist with managing what users belong to what roles. The first page will include facilities to see what users belong to a given role, what roles a particular user belongs to, and the ability to assign or remove a particular user from a particular role. In the second page we will augment the CreateUserWizard control so that it includes a step to specify what roles the newly created user belongs to. This is useful in scenarios where an administrator is able to create new user accounts.

Introduction

The previous tutorial examined the Roles framework and the `SqlRoleProvider`; we saw how to use the `Roles` class to create, retrieve, and delete roles. In addition to creating and deleting roles, we need to be able to assign or remove users from a role. Unfortunately, ASP.NET does not ship with any Web controls for managing what users belong to what roles. Instead, we must create our own ASP.NET pages to manage these associations. The good news is that adding and removing users to roles is quite easy. The `Roles` class contains a number of methods for adding one or more users to one or more roles.

In this tutorial we will build two ASP.NET pages to assist with managing what users belong to what roles. The first page will include facilities to see what users belong to a given role, what roles a particular user belongs to, and the ability to assign or remove a particular user from a particular role. In the second page we will augment the CreateUserWizard control so that it includes a step to specify what roles the newly created user belongs to. This is useful in scenarios where an administrator is able to create new user accounts.

Let's get started!

Listing What Users Belong To What Roles

The first order of business for this tutorial is to create a web page from which users can be assigned to roles. Before we concern ourselves with how to assign users to roles, let's first concentrate on how to determine what users belong to what roles. There are two ways to display this information: "by role" or "by user." We could allow the visitor to select a role and then show them all of the users that belong to the role (the "by role" display), or we could prompt the visitor to select a user and then show them the roles assigned to that user (the "by user" display).

The "by role" view is useful in circumstances where the visitor wants to know the set of users that belong to a particular role; the "by user" view is ideal when the visitor needs to know a particular user's role(s). Let's have our page include both "by role" and "by user" interfaces.

We will start with creating the "by user" interface. This interface will consist of a drop-down list and a list of checkboxes. The drop-down list will be populated with the set of users in the system; the checkboxes will enumerate the roles. Selecting a user from the drop-down list will check those roles the user belongs to. The person visiting the page can then check or uncheck the checkboxes to add or remove the selected user from the corresponding roles.

> **NOTE**
> Using a drop-down list to list the user accounts is not an ideal choice for websites where there may be hundreds of user accounts. A drop-down list is designed to allow a user to pick one item from a relatively short list of options. It quickly becomes unwieldy as the number of list items grows. If you are building a website that will have potentially large numbers of user accounts, you may want to consider using an alternative user interface, such as a pageable GridView or a filterable interface that lists prompts the visitor to choose a letter and then only shows those users whose username starts with the selected letter.

Step 1: Building the "By User" User Interface

Open the `UsersAndRoles.aspx` page. At the top of the page, add a Label Web control named `ActionStatus` and clear out its `Text` property. We will use this Label to provide feedback on the actions performed, displaying messages like, "User Tito has been added to the Administrators role," or "User Jisun has been removed from the Supervisors role." In order to make these messages stand out, set the Label's `CssClass` property to "Important".

```
<p align="center">
     <asp:Label ID="ActionStatus" runat="server" CssClass="Important"> </asp:Label>
</p>
```

Next, add the following CSS class definition to the `Styles.css` stylesheet:

```
.Important
{
     font-size: large;
     color: Red;
}
```

This CSS definition instructs the browser to display the Label using a large, red font. Figure 1 shows this effect through the Visual Studio Designer.

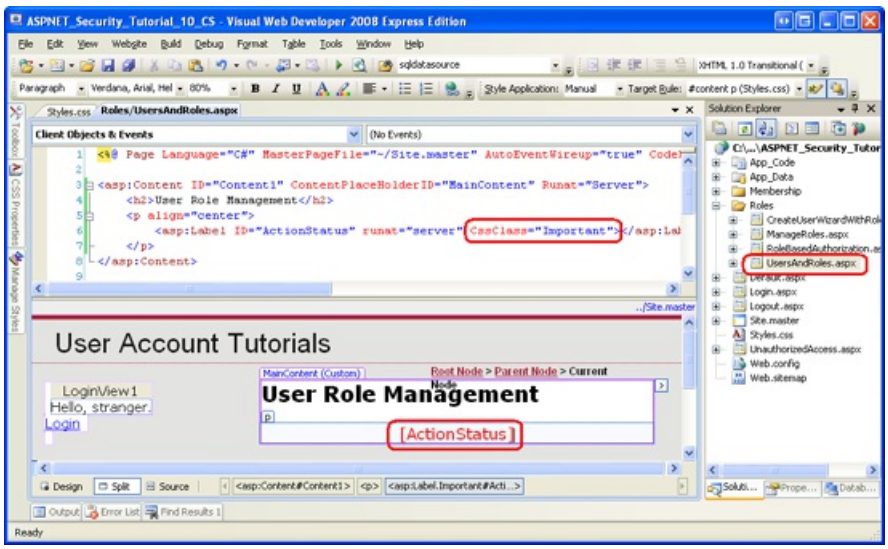


Figure 1: The Label's `CssClass` Property Results in a Large, Red Font (Click to view full-size image)

Next, add a DropDownList to the page, set its `ID` property to `UserList`, and set its `AutoPostBack` property to True. We will use this DropDownList to list all of the users in the system. This DropDownList will be bound to a collection of MembershipUser objects. Because we want the DropDownList to display the UserName property of the MembershipUser object (and use it as the value of the list items), set the DropDownList's `DataTextField` and `DataValueField` properties to "UserName".

Underneath the DropDownList, add a Repeater named `UsersRoleList`. This Repeater will list all of the roles in the system as a series of checkboxes. Define the Repeater's `ItemTemplate` using the following declarative markup:

```
<asp:Repeater ID="UsersRoleList" runat="server">
     <ItemTemplate>
          <asp:CheckBox runat="server" ID="RoleCheckBox" AutoPostBack="true"
               Text='<%# Container.DataItem %>' />
          <br />
     </ItemTemplate>
</asp:Repeater>
```

The `ItemTemplate` markup includes a single CheckBox Web control named `RoleCheckBox`. The CheckBox's `AutoPostBack` property is set to True and the `Text` property is bound to `Container.DataItem`. The reason the databinding syntax is simply `Container.DataItem` is because the Roles framework returns the list of role names as a string array, and it is this string array that we will be binding to the Repeater. A thorough description of why this syntax is used to display the contents of an array bound to a data Web control is beyond the scope of this tutorial. For more information on this matter, refer to Binding a Scalar Array to a Data Web Control.

At this point your "by user" interface's declarative markup should look similar to the following:

```
<h3>Manage Roles By User</h3>
<p>
     <b>Select a User:</b>
     <asp:DropDownList ID="UserList" runat="server" AutoPostBack="True"
          DataTextField="UserName" DataValueField="UserName">
     </asp:DropDownList>
</p>
<p>
     <asp:Repeater ID="UsersRoleList" runat="server">
          <ItemTemplate>
               <asp:CheckBox runat="server" ID="RoleCheckBox" AutoPostBack="true"
                    Text='<%# Container.DataItem %>' />
               <br />
          </ItemTemplate>
     </asp:Repeater>
</p>
```

We are now ready to write the code to bind the set of user accounts to the DropDownList and the set of roles to the Repeater. In the page's code-behind class, add a method named `BindUsersToUserList` and another named `BindRolesList`, using the following code:

```
Private Sub BindUsersToUserList()
     ' Get all of the user accounts
     Dim users As MembershipUserCollection = Membership.GetAllUsers()
     UserList.DataSource = users
     UserList.DataBind()
End Sub

Private Sub BindRolesToList()
     ' Get all of the roles
     Dim roleNames() As String = Roles.GetAllRoles()
     UsersRoleList.DataSource = roleNames
     UsersRoleList.DataBind()
End Sub
```

The `BindUsersToUserList` method retrieves all of the user accounts in the system via the `Membership.GetAllUsers` method. This returns a `MembershipUserCollection` object, which is a collection of `MembershipUser` instances. This collection is then bound to the `UserList` DropDownList. The `MembershipUser` instances that makeup the collection contain a variety of properties, like `UserName`, `Email`, `CreationDate`, and `IsOnline`. In order to instruct

the DropDownList to display the value of the `UserName` property, ensure that the `UserList` DropDownList's `DataTextField` and `DataValueField` properties have been set to "UserName".

> **NOTE**
>
> The `Membership.GetAllUsers` method has two overloads: one that accepts no input parameters and returns all of the users, and one that takes in integer values for the page index and page size, and returns only the specified subset of the users. When there are large amounts of user accounts being displayed in a pageable user interface element, the second overload can be used to more efficiently page through the users since it returns just the precise subset of user accounts rather than all of them.

The `BindRolesToList` method starts by calling the `Roles` class's `GetAllRoles` method, which returns a string array containing the roles in the system. This string array is then bound to the Repeater.

Finally, we need to call these two methods when the page is first loaded. Add the following code to the `Page_Load` event handler:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
     If Not Page.IsPostBack Then
          ' Bind the users and roles
          BindUsersToUserList()
          BindRolesToList()
     End If
End Sub
```

With this code in place, take a moment to visit the page through a browser; your screen should look similar to Figure 2. All of the user accounts are populated in the drop-down list and, underneath that, each role appears as a checkbox. Because we set the `AutoPostBack` properties of the DropDownList and CheckBoxes to True, changing the selected user or checking or unchecking a role causes a postback. No action is performed, however, because we have yet to write code to handle these actions. We'll tackle these tasks in the next two sections.

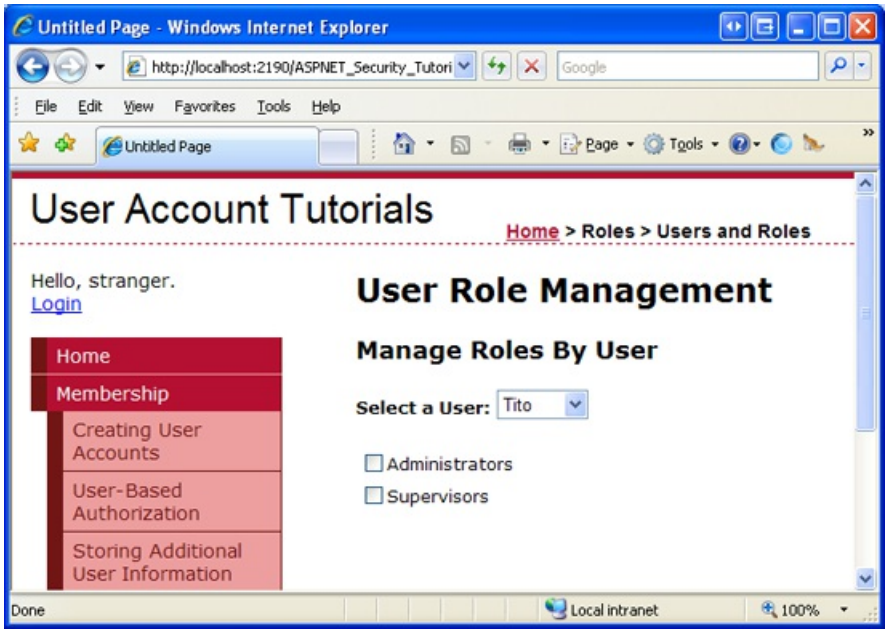


Figure 2: The Page Displays the Users and Roles (Click to view full-size image)

Checking the Roles the Selected User Belongs To

When the page is first loaded, or whenever the visitor selects a new user from the drop-down list, we need to update the `UsersRoleList`'s checkboxes so that a given role checkbox is checked only if the selected user belongs to that role. To accomplish this, create a method named `CheckRolesForSelectedUser` with the following code:

```
Private Sub CheckRolesForSelectedUser()
     ' Determine what roles the selected user belongs to
     Dim selectedUserName As String = UserList.SelectedValue
     Dim selectedUsersRoles() As String = Roles.GetRolesForUser(selectedUserName)

     ' Loop through the Repeater's Items and check or uncheck the checkbox as needed
     For Each ri As RepeaterItem In UsersRoleList.Items
          ' Programmatically reference the CheckBox
          Dim RoleCheckBox As CheckBox = CType(ri.FindControl("RoleCheckBox"), CheckBox)
          ' See if RoleCheckBox.Text is in selectedUsersRoles
          If Linq.Enumerable.Contains(Of String)(selectedUsersRoles, RoleCheckBox.Text) Then
               RoleCheckBox.Checked = True
          Else
               RoleCheckBox.Checked = False
          End If
     Next
End Sub
```

The above code starts by determining who the selected user is. It then uses the Roles class's `GetRolesForUser(userName)` method to return the specified user's set of roles as a string array. Next, the Repeater's items are enumerated and each item's `RoleCheckBox` CheckBox is programmatically referenced. The CheckBox is checked only if the role it corresponds to is contained within the `selectedUsersRoles` string array.

> **NOTE**
> The `Linq.Enumerable.Contains(Of String)(...)` syntax will not compile if you are using ASP.NET version 2.0. The `Contains(Of String)` method is part of the LINQ library, which is new to ASP.NET 3.5. If you are still using ASP.NET version 2.0, use the `Array.IndexOf(Of String)` method instead.

The `CheckRolesForSelectedUser` method needs to be called in two cases: when the page is first loaded and whenever the `UserList` DropDownList's selected index is changed. Therefore, call this method from the `Page_Load` event handler (after the calls to `BindUsersToUserList` and `BindRolesToList`). Also, create an event handler for the DropDownList's `SelectedIndexChanged` event and call this method from there.

```
Protected Sub Page_Load(ByVal sender As Object,ByVal e As System.EventArgs) Handles Me.Load
     If Not Page.IsPostBack Then
          ' Bind the users and roles
          BindUsersToUserList()
          BindRolesToList()
          ' Check the selected user's roles
          CheckRolesForSelectedUser()
     End If
End Sub

...

Protected Sub UserList_SelectedIndexChanged(ByVal sender As Object,ByVal e As System.EventArgs) Handles
UserList.SelectedIndexChanged
     CheckRolesForSelectedUser()
End Sub
```

With this code in place, you can test the page through the browser. However, since the `UsersAndRoles.aspx` page currently lacks the ability to assign users to roles, no users have roles. We will create the interface for assigning users to roles in a moment, so you can either take my word that this code works and verify that it does so later, or you can manually add users to roles by inserting records into the `aspnet_UsersInRoles` table in order to test this functionality now.

Assigning and Removing Users from Roles

When the visitor checks or unchecks a CheckBox in the `UsersRoleList` Repeater we need to add or remove the selected user from the corresponding role. The CheckBox's `AutoPostBack` property is currently set to True, which causes a postback anytime a CheckBox in the Repeater is checked or unchecked. In short, we need to create an event handler for the CheckBox's `CheckChanged` event. Since the CheckBox is in a Repeater control, we need to manually add the event handler plumbing. Start by adding the event handler to the code-behind class as a `Protected` method, like so:

```
Protected Sub RoleCheckBox_CheckChanged(ByVal sender As Object, ByVal e As EventArgs)

End Sub
```

We will return to write the code for this event handler in a moment. But first let's complete the event handling plumbing. From the CheckBox within the Repeater's `ItemTemplate`, add `OnCheckedChanged="RoleCheckBox_CheckChanged"`. This syntax wires the `RoleCheckBox_CheckChanged` event handler to the `RoleCheckBox`'s `CheckedChanged` event.

```
<asp:CheckBox runat="server" ID="RoleCheckBox"
     AutoPostBack="true"
     Text='<%# Container.DataItem %>'
     OnCheckedChanged="RoleCheckBox_CheckChanged" />
```

Our final task is to complete the `RoleCheckBox_CheckChanged` event handler. We need to start by referencing the CheckBox control that raised the event because this CheckBox instance tells us what role was checked or unchecked via its `Text` and `Checked` properties. Using this information along with the UserName of the selected user, we add or remove the user from the role via the `Roles` class's `AddUserToRole` or `RemoveUserFromRole` method.

```
Protected Sub RoleCheckBox_CheckChanged(ByVal sender As Object, ByVal e As EventArgs)
     'Reference the CheckBox that raised this event
     Dim RoleCheckBox As CheckBox = CType(sender, CheckBox)

     ' Get the currently selected user and role
     Dim selectedUserName As String = UserList.SelectedValue
     Dim roleName As String = RoleCheckBox.Text

     ' Determine if we need to add or remove the user from this role
     If RoleCheckBox.Checked Then
          ' Add the user to the role
          Roles.AddUserToRole(selectedUserName, roleName)
          ' Display a status message
          ActionStatus.Text = String.Format("User {0} was added to role {1}.", selectedUserName,roleName)
     Else
          ' Remove the user from the role
          Roles.RemoveUserFromRole(selectedUserName, roleName)
          ' Display a status message
          ActionStatus.Text = String.Format("User {0} was removed from role {1}.", selectedUserName,roleName)
     End If
End Sub
```

The above code starts by programmatically referencing the CheckBox that raised the event, which is available via the `sender` input parameter. If the CheckBox is checked, the selected user is added to the specified role, otherwise they are removed from the role. In either case, the `ActionStatus` Label displays a message summarizing the action just performed.

Take a moment to test out this page through a browser. Select user Tito and then add Tito to both the Administrators and Supervisors roles.

Figure 3: Tito Has Been Added to the Administrators and Supervisors Roles (Click to view full-size image)

Next, select user Bruce from the drop-down list. There is a postback and the Repeater's CheckBoxes are updated via the `CheckRolesForSelectedUser`. Since Bruce does not yet belong to any roles, the two checkboxes are unchecked. Next, add Bruce to the Supervisors role.

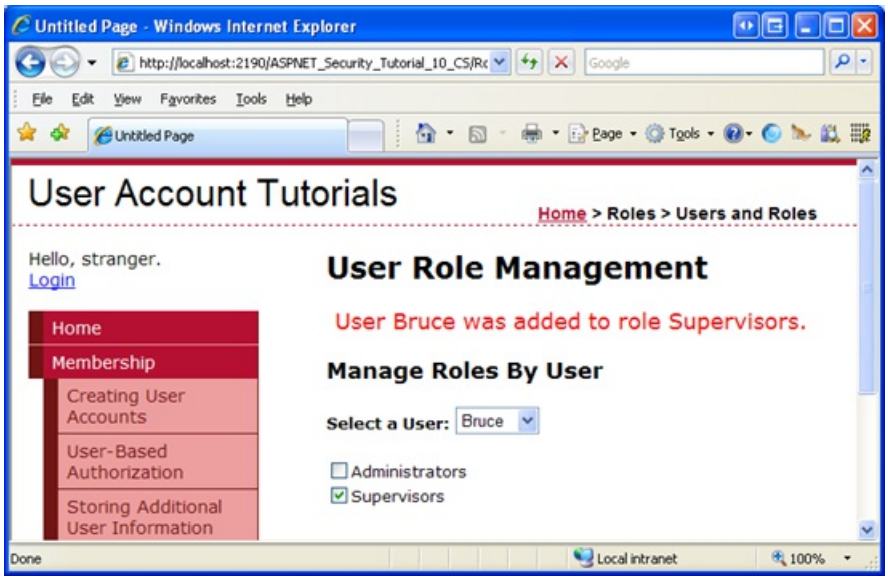


Figure 4: Bruce Has Been Added to the Supervisors Role (Click to view full-size image)

To further verify the functionality of the `CheckRolesForSelectedUser` method, select a user other than Tito or Bruce. Note how the checkboxes are automatically unchecked, denoting that they do not belong to any roles. Return to Tito. Both the Administrators and Supervisors checkboxes should be checked.

Step 2: Building the "By Roles" User Interface

At this point we have completed the "by users" interface and are ready to start tackling the "by roles" interface. The "by roles" interface prompts the user to select a role from a drop-down list and then displays the set of users that belong to that role in a GridView.

Add another DropDownList control to the `UsersAndRoles.aspx page`. Place this one beneath the Repeater control, name it `RoleList`, and set its `AutoPostBack` property to True. Underneath that, add a GridView and name it `RolesUserList`. This GridView will list the users that belong to the selected role. Set the GridView's `AutoGenerateColumns` property to False, add a TemplateField to the grid's `Columns` collection, and set its `HeaderText` property to "Users". Define the TemplateField's `ItemTemplate` so that it displays the value of the

databinding expression `Container.DataItem` in the `Text` property of a Label named `UserNameLabel`.

After adding and configuring the GridView, your "by role" interface's declarative markup should look similar to the following:

```
<h3>Manage Users By Role</h3>
<p>
     <b>Select a Role:</b>
     <asp:DropDownList ID="RoleList" runat="server" AutoPostBack="true"></asp:DropDownList>
</p>
<p>
     <asp:GridView ID="RolesUserList" runat="server" AutoGenerateColumns="false"
          EmptyDataText="No users belong to this role.">
          <Columns>
               <asp:TemplateField HeaderText="Users">
                    <ItemTemplate>
                         <asp:Label runat="server" id="UserNameLabel"
                              Text='<%# Container.DataItem %>'></asp:Label>
                    </ItemTemplate>
               </asp:TemplateField>
          </Columns>
     </asp:GridView>
</p>
```

We need to populate the `RoleList` DropDownList with the set of roles in the system. To accomplish this, update the `BindRolesToList` method so that is binds the string array returned by the `Roles.GetAllRoles` method to the `RolesList` DropDownList (as well as the `UsersRoleList` Repeater).

```
Private Sub BindRolesToList()
     ' Get all of the roles
     Dim roleNames() As String = Roles.GetAllRoles()
     UsersRoleList.DataSource = roleNames
     UsersRoleList.DataBind()

     RoleList.DataSource = roleNames
     RoleList.DataBind()
End Sub
```

The last two lines in the `BindRolesToList` method have been added to bind the set of roles to the `RoleList` DropDownList control. Figure 5 shows the end result when viewed through a browser – a drop-down list populated with the system's roles.

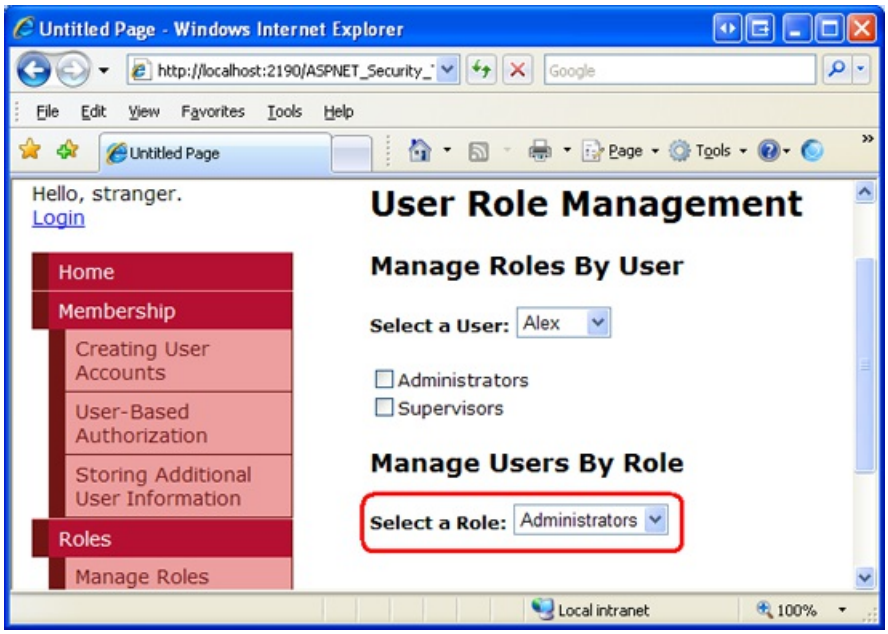


Figure 5: The Roles are Displayed in the `RoleList` DropDownList (Click to view full-size image)

Displaying the Users That Belong To the Selected Role

When the page is first loaded, or when a new role is selected from the `RoleList` DropDownList, we need to display the list of users that belong to that role in the GridView. Create a method named `DisplayUsersBelongingToRole` using the following code:

```
Private Sub DisplayUsersBelongingToRole()
     ' Get the selected role
     Dim selectedRoleName As String = RoleList.SelectedValue

     ' Get the list of usernames that belong to the role
     Dim usersBelongingToRole() As String = Roles.GetUsersInRole(selectedRoleName)

     ' Bind the list of users to the GridView
     RolesUserList.DataSource = usersBelongingToRole
     RolesUserList.DataBind()
End Sub
```

This method starts by getting the selected role from the `RoleList` DropDownList. It then uses the `Roles.GetUsersInRole(roleName)` method to retrieve a string array of the UserNames of the users that belong to that role. This array is then bound to the `RolesUserList` GridView.

This method needs to be called in two circumstances: when the page is initially loaded and when the selected role in the `RoleList` DropDownList changes. Therefore, update the `Page_Load` event handler so that this method is invoked after the call to `CheckRolesForSelectedUser`. Next, create an event handler for the `RoleList`'s `SelectedIndexChanged` event, and call this method from there, too.

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
     If Not Page.IsPostBack Then
          ' Bind the users and roles
          BindUsersToUserList()
          BindRolesToList()

          ' Check the selected user's roles
          CheckRolesForSelectedUser()

          'Display those users belonging to the currently selected role
          DisplayUsersBelongingToRole()
     End If
End Sub

...

Protected Sub RoleList_SelectedIndexChanged(ByVal sender As Object, ByVal e As System.EventArgs) Handles
RoleList.SelectedIndexChanged
     DisplayUsersBelongingToRole()
End Sub
```

With this code in place, the `RolesUserList` GridView should display those users that belong to the selected role. As Figure 6 shows, the Supervisors role consists of two members: Bruce and Tito.

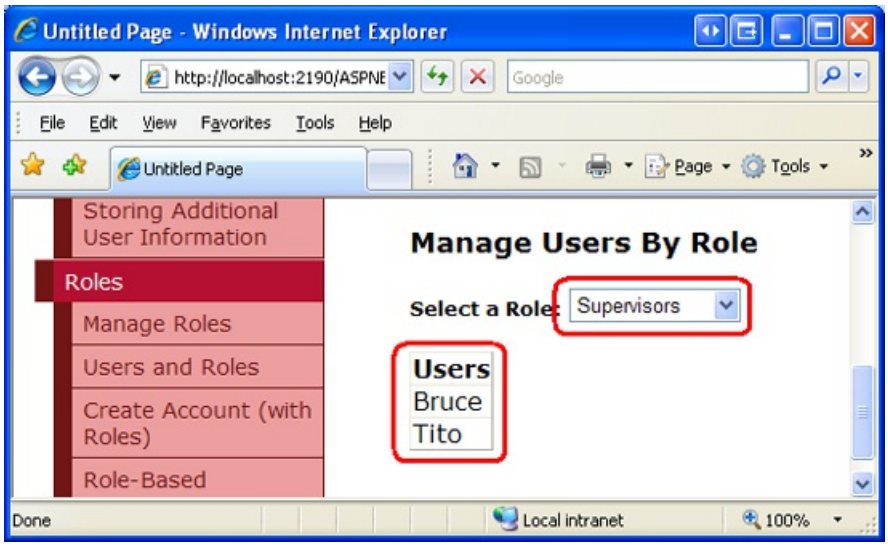


Figure 6: The GridView Lists Those Users That Belong to the Selected Role (Click to view full-size image)

Removing Users from the Selected Role

Let's augment the `RolesUserList` GridView so that it includes a column of "Remove" buttons. Clicking the "Remove" button for a particular user will remove them from that role.

Start by adding a Delete button field to the GridView. Make this field appear as the left most filed and change its `DeleteText` property from "Delete" (the default) to "Remove".

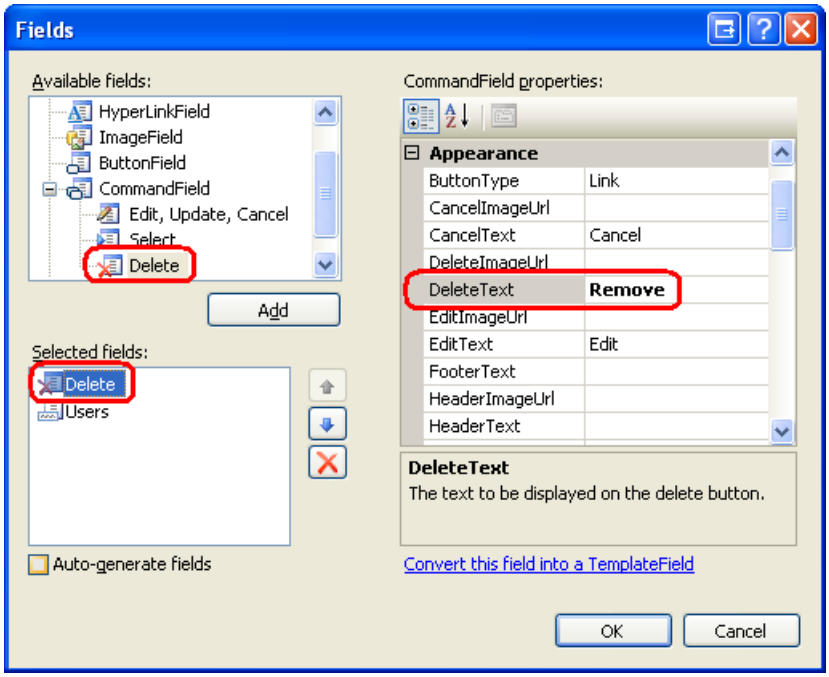


Figure 7: Add the "Remove" Button to the GridView (Click to view full-size image)

When the "Remove" button is clicked a postback ensues and the GridView's `RowDeleting` event is raised. We need to create an event handler for this event and write code that removes the user from the selected role. Create the event handler and then add the following code:

```
Protected Sub RolesUserList_RowDeleting(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.GridViewDeleteEventArgs) Handles RolesUserList.RowDeleting
     ' Get the selected role
     Dim selectedRoleName As String = RoleList.SelectedValue

     ' Reference the UserNameLabel
     Dim UserNameLabel As Label = CType(RolesUserList.Rows(e.RowIndex).FindControl("UserNameLabel"),Label)

     ' Remove the user from the role
     Roles.RemoveUserFromRole(UserNameLabel.Text, selectedRoleName)

     ' Refresh the GridView
     DisplayUsersBelongingToRole()

     ' Display a status message
     ActionStatus.Text = String.Format("User {0} was removed from role {1}.",
UserNameLabel.Text,selectedRoleName)
End Sub
```

The code starts by determining the selected role name. It then programmatically references the `UserNameLabel` control from the row whose "Remove" button was clicked in order to determine the UserName of the user to remove. The user is then removed from the role via a call to the `Roles.RemoveUserFromRole` method. The `RolesUserList` GridView is then refreshed and a message is displayed via the `ActionStatus` Label control.

> **NOTE**
>
> The "Remove" button does not require any sort of confirmation from the user before removing the user from the role. I invite you to add some level of user confirmation. One of the easiest ways to confirm an action is through a client-side confirm dialog box. For more information on this technique, see Adding Client-Side Confirmation When Deleting.

Figure 8 shows the page after user Tito has been removed from the Supervisors group.

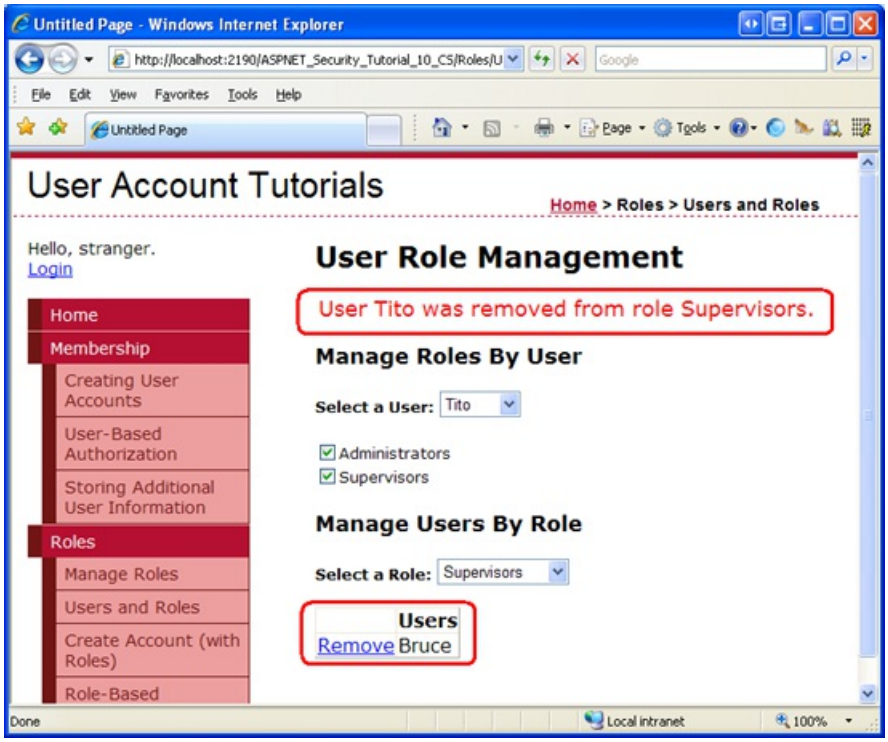


Figure 8: Alas, Tito is No Longer a Supervisor (Click to view full-size image)

Adding New Users to the Selected Role

Along with removing users from the selected role, the visitor to this page should also be able to add a user to the selected role. The best interface for adding a user to the selected role depends on the number of user accounts

you expect to have. If your website will house just a few dozen user accounts or less, you could use a DropDownList here. If there might be thousands of user accounts, you would want to include a user interface that permits the visitor to page through the accounts, search for a particular account, or filter the user accounts in some other fashion.

For this page let's use a very simple interface that works regardless of the number of user accounts in the system. Namely, we will use a TextBox, prompting the visitor to type in the username of the user she wants to add to the selected role. If no user with that name exists, or if the user is already a member of the role, we'll display a message in `ActionStatus` Label. But if the user exists and is not a member of the role, we'll add them to the role and refresh the grid.

Add a TextBox and Button beneath the GridView. Set the TextBox's `ID` to `UserNameToAddToRole` and set the Button's `ID` and `Text` properties to `AddUserToRoleButton` and "Add User to Role", respectively.

```
<p>
     <b>UserName:</b>
     <asp:TextBox ID="UserNameToAddToRole" runat="server"></asp:TextBox>
     <br />
     <asp:Button ID="AddUserToRoleButton" runat="server" Text="Add User to Role" />
</p>
```

Next, create a `Click` event handler for the `AddUserToRoleButton` and add the following code:

```
Protected Sub AddUserToRoleButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
AddUserToRoleButton.Click
     ' Get the selected role and username
     Dim selectedRoleName As String = RoleList.SelectedValue
     Dim userToAddToRole As String = UserNameToAddToRole.Text

     ' Make sure that a value was entered
     If userToAddToRole.Trim().Length = 0 Then
          ActionStatus.Text = "You must enter a username in the textbox."
          Exit Sub
     End If

     ' Make sure that the user exists in the system
     Dim userInfo As MembershipUser = Membership.GetUser(userToAddToRole)
     If userInfo Is Nothing Then
          ActionStatus.Text = String.Format("The user {0} does not exist in the system.",userNameToAddToRole)
          Exit Sub
     End If

     ' Make sure that the user doesn't already belong to this role
     If Roles.IsUserInRole(userToAddToRole, selectedRoleName) Then
          ActionStatus.Text = String.Format("User {0} already is a member of role {1}.",
UserNameToAddToRole,selectedRoleName)
          Exit Sub
     End If

     ' If we reach here, we need to add the user to the role
     Roles.AddUserToRole(userToAddToRole, selectedRoleName)

     ' Clear out the TextBox
     userNameToAddToRole.Text = String.Empty

     ' Refresh the GridView
     DisplayUsersBelongingToRole()

     ' Display a status message
     ActionStatus.Text = String.Format("User {0} was added to role {1}.",
UserNameToAddToRole,selectedRoleName)
End Sub
```

The majority of the code in the `Click` event handler performs various validation checks. It ensures that the visitor supplied a username in the `UserNameToAddToRole` TextBox, that the user exists in the system, and that they don't already belong to the selected role. If any of these checks fails, an appropriate message is displayed in `ActionStatus` and the event handler is exited. If all of the checks pass, the user is added to the role via the `Roles.AddUserToRole` method. Following that, the TextBox's `Text` property is cleared out, the GridView is refreshed, and the `ActionStatus` Label displays a message indicating that the specified user was successfully added to the selected role.

NOTE

To ensure that the specified user does not already belong to the selected role, we use the `Roles.IsUserInRole(userName, roleName)` method, which returns a Boolean value indicating whether *userName* is a member of *roleName*. We will use this method again in the next tutorial when we look at role-based authorization.

Visit the page through a browser and select the Supervisors role from the `RoleList` DropDownList. Try entering an invalid username – you should see a message explaining that the user does not exist in the system.

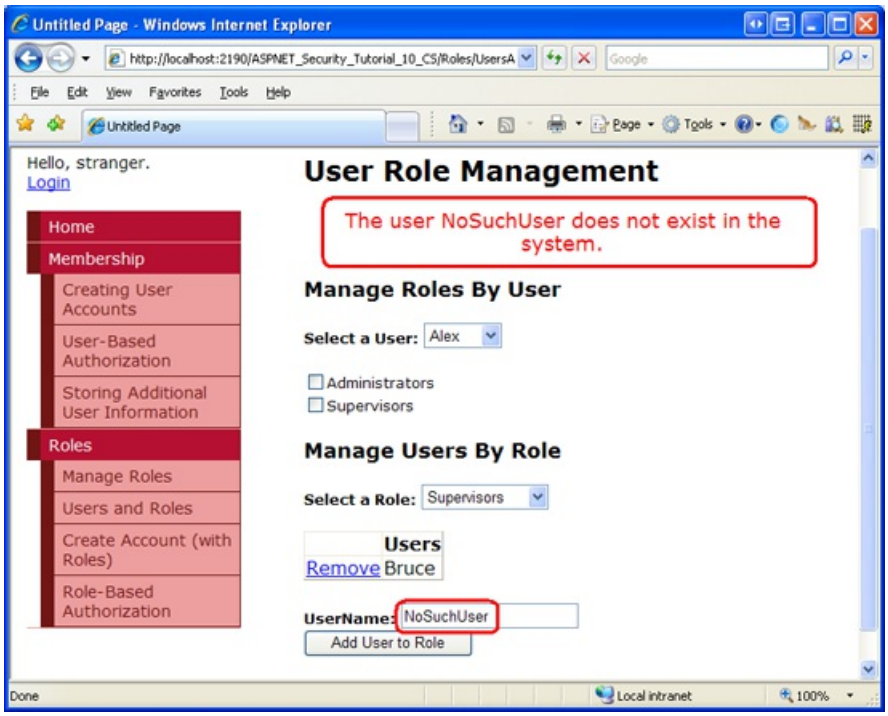


Figure 9: You Cannot Add a Non-Existent User to a Role (Click to view full-size image)

Now try adding a valid user. Go ahead and re-add Tito to the Supervisors role.

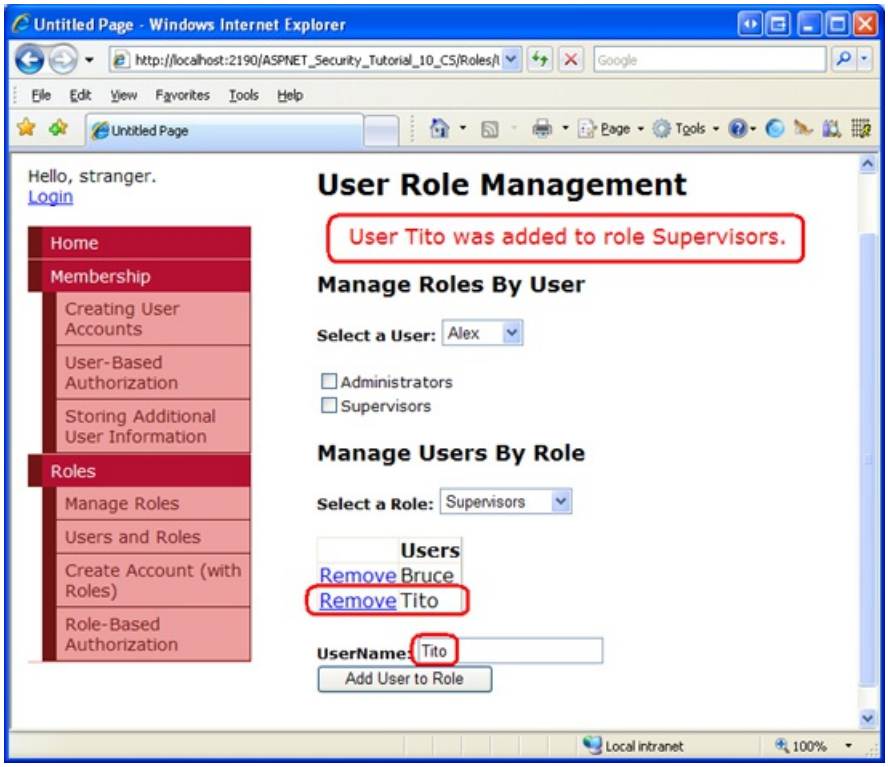


Figure 10: Tito Is Once Again a Supervisor! (Click to view full-size image)

Step 3: Cross-Updating the "By User" and "By Role" Interfaces

The `UsersAndRoles.aspx` page offers two distinct interfaces for managing users and roles. Currently, these two interfaces act independently of one another so it is possible that a change made in one interface will not be reflected immediately in the other. For example, imagine that the visitor to the page selects the Supervisors role from the `RoleList` DropDownList, which lists Bruce and Tito as its members. Next, the visitor selects Tito from the `UserList` DropDownList, which checks the Administrators and Supervisors checkboxes in the `UsersRoleList` Repeater. If the visitor then unchecks the Supervisor role from the Repeater, Tito is removed from the Supervisors role, but this modification is not reflected in the "by role" interface. The GridView will still show Tito as being a

member of the Supervisors role.

To fix this we need to refresh the GridView whenever a role is checked or unchecked from the `UsersRoleList` Repeater. Likewise, we need to refresh the Repeater whenever a user is removed or added to a role from the "by role" interface.

The Repeater in the "by user" interface is refreshed by calling the `CheckRolesForSelectedUser` method. The "by role" interface can be modified in the `RolesUserList` GridView's `RowDeleting` event handler and the `AddUserToRoleButton` Button's `Click` event handler. Therefore, we need to call the `CheckRolesForSelectedUser` method from each of these methods.

```
Protected Sub RolesUserList_RowDeleting(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.GridViewDeleteEventArgs) Handles RolesUserList.RowDeleting
     ... Code removed for brevity ...

     ' Refresh the "by user" interface
     CheckRolesForSelectedUser()
End Sub

Protected Sub AddUserToRoleButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
AddUserToRoleButton.Click
     ... Code removed for brevity ...

     ' Refresh the "by user" interface
     CheckRolesForSelectedUser()
End Sub
```

Similarly, the GridView in the "by role" interface is refreshed by calling the `DisplayUsersBelongingToRole` method and the "by user" interface is modified through the `RoleCheckBox_CheckChanged` event handler. Therefore, we need to call the `DisplayUsersBelongingToRole` method from this event handler.

```
Protected Sub RoleCheckBox_CheckChanged(ByVal sender As Object, ByVal e As EventArgs)
     ... Code removed for brevity ...

     ' Refresh the "by role" interface
     DisplayUsersBelongingToRole()
End Sub
```

With these minor code changes, the "by user" and "by role" interfaces now correctly cross-update. To verify this, visit the page through a browser and select Tito and Supervisors from the `UserList` and `RoleList` DropDownLists, respectively. Note that as you uncheck the Supervisors role for Tito from the Repeater in the "by user" interface, Tito is automatically removed from the GridView in the "by role" interface. Adding Tito back to the Supervisors role from the "by role" interface automatically re-checks the Supervisors checkbox in the "by user" interface.

Step 4: Customizing the CreateUserWizard to Include a "Specify Roles" Step

In the *Creating User Accounts* tutorial we saw how to use the CreateUserWizard Web control to provide an interface for creating a new user account. The CreateUserWizard control can be used in one of two ways:

- As a means for visitors to create their own user account on the site, and
- As a means for administrators to create new accounts

In the first use case, a visitor comes to the site and fills out the CreateUserWizard, entering their information in order to register on the site. In the second case, an administrator creates a new account for another person.

When an account is being created by an administrator for some other person, it might be helpful to allow the administrator to specify what roles the new user account belongs to. In the *Storing Additional User Information* tutorial we saw how to customize the CreateUserWizard by adding additional `WizardSteps`. Let's look at how to add an additional step to the CreateUserWizard in order to specify the new user's roles.

Open the `CreateUserWizardWithRoles.aspx` page and add a CreateUserWizard control named `RegisterUserWithRoles`. Set the control's `ContinueDestinationPageUrl` property to "~/Default.aspx". Because the idea here is that an administrator will be using this CreateUserWizard control to create new user accounts, set the control's `LoginCreatedUser` property to False. This `LoginCreatedUser` property specifies whether the visitor is automatically logged on as the just-created user, and it defaults to True. We set it to False because when an administrator creates a new account we want to keep him signed in as himself.

Next, select the "Add/Remove `WizardSteps`..." option from the CreateUserWizard's Smart Tag and add a new `WizardStep`, setting its `ID` to `SpecifyRolesStep`. Move the `SpecifyRolesStep WizardStep` so that it comes after the "Sign Up for Your New Account" step, but before the "Complete" step. Set the `WizardStep`'s `Title` property to "Specify Roles", its `StepType` property to `Step`, and its `AllowReturn` property to False.

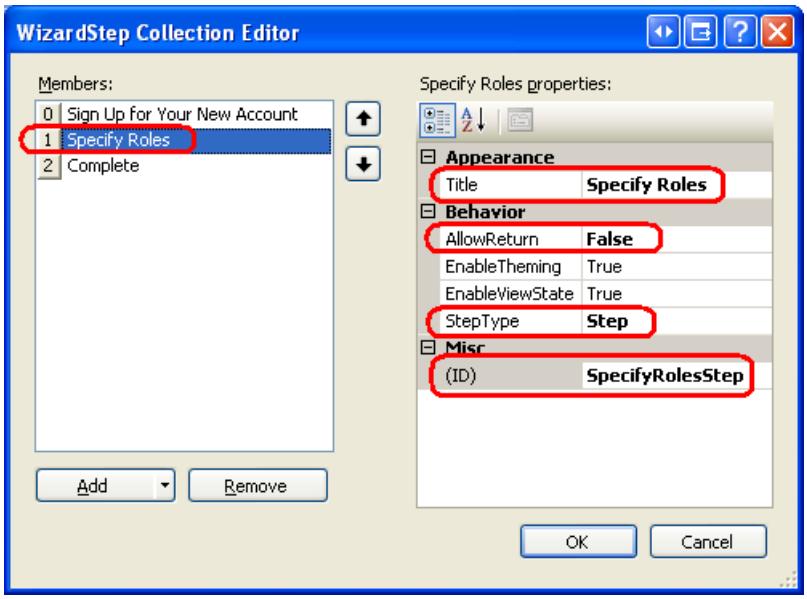


Figure 11: Add the "Specify Roles" `WizardStep` to the CreateUserWizard (Click to view full-size image)

After this change your CreateUserWizard's declarative markup should look like the following:

```
<asp:CreateUserWizard ID="RegisterUserWithRoles" runat="server"
     ContinueDestinationPageUrl="~/Default.aspx" LoginCreatedUser="False">
     <WizardSteps>
          <asp:CreateUserWizardStep ID="CreateUserWizardStep1" runat="server">
          </asp:CreateUserWizardStep>
          <asp:WizardStep ID="SpecifyRolesStep" runat="server" StepType="Step"
               Title="Specify Roles" AllowReturn="False">
          </asp:WizardStep>
          <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">
          </asp:CompleteWizardStep>
     </WizardSteps>
</asp:CreateUserWizard>
```

In the "Specify Roles" `WizardStep`, add a CheckBoxList named `RoleList.` This CheckBoxList will list the available roles, enabling the person visiting the page to check what roles the newly created user belongs to.

We are left with two coding tasks: first we must populate the `RoleList` CheckBoxList with the roles in the system; second, we need to add the created user to the selected roles when the user moves from the "Specify Roles" step to the "Complete" step. We can accomplish the first task in the `Page_Load` event handler. The following code

programmatically references the `RoleList` CheckBox on the first visit to the page and binds the roles in the system to it.

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
     If Not Page.IsPostBack Then
          ' Reference the SpecifyRolesStep WizardStep
          Dim SpecifyRolesStep As WizardStep =
CType(RegisterUserWithRoles.FindControl("SpecifyRolesStep"),WizardStep)

          ' Reference the RoleList CheckBoxList
          Dim RoleList As CheckBoxList = CType(SpecifyRolesStep.FindControl("RoleList"), CheckBoxList)

          ' Bind the set of roles to RoleList
          RoleList.DataSource = Roles.GetAllRoles()
          RoleList.DataBind()
     End If
End Sub
```

The above code should look familiar. In the *Storing Additional User Information* tutorial we used two `FindControl` statements to reference a Web control from within a custom `WizardStep`. And the code that binds the roles to the CheckBoxList was taken from earlier in this tutorial.

In order to perform the second programming task we need to know when the "Specify Roles" step has been completed. Recall that the CreateUserWizard has an `ActiveStepChanged` event, which fires each time the visitor navigates from one step to another. Here we can determine if the user has reached the "Complete" step; if so, we need to add the user to the selected roles.

Create an event handler for the `ActiveStepChanged` event and add the following code:

```
Protected Sub RegisterUserWithRoles_ActiveStepChanged(ByVal sender As Object, ByVal e As System.EventArgs)
Handles RegisterUserWithRoles.ActiveStepChanged
     'Have we JUST reached the Complete step?
     If RegisterUserWithRoles.ActiveStep.Title = "Complete" Then
          ' Reference the SpecifyRolesStep WizardStep
          Dim SpecifyRolesStep As WizardStep =
CType(RegisterUserWithRoles.FindControl("SpecifyRolesStep"),WizardStep)

          ' Reference the RoleList CheckBoxList
          Dim RoleList As CheckBoxList = CType(SpecifyRolesStep.FindControl("RoleList"), CheckBoxList)

          ' Add the checked roles to the just-added user
          For Each li As ListItem In RoleList.Items
               If li.Selected Then
                    Roles.AddUserToRole(RegisterUserWithRoles.UserName, li.Text)
               End If
          Next
     End If
End Sub
```

If the user has just reached the "Completed" step, the event handler enumerates the items of the `RoleList` CheckBoxList and the just-created user is assigned to the selected roles.

Visit this page through a browser. The first step in the CreateUserWizard is the standard "Sign Up for Your New Account" step, which prompts for the new user's username, password, email, and other key information. Enter the information to create a new user named Wanda.

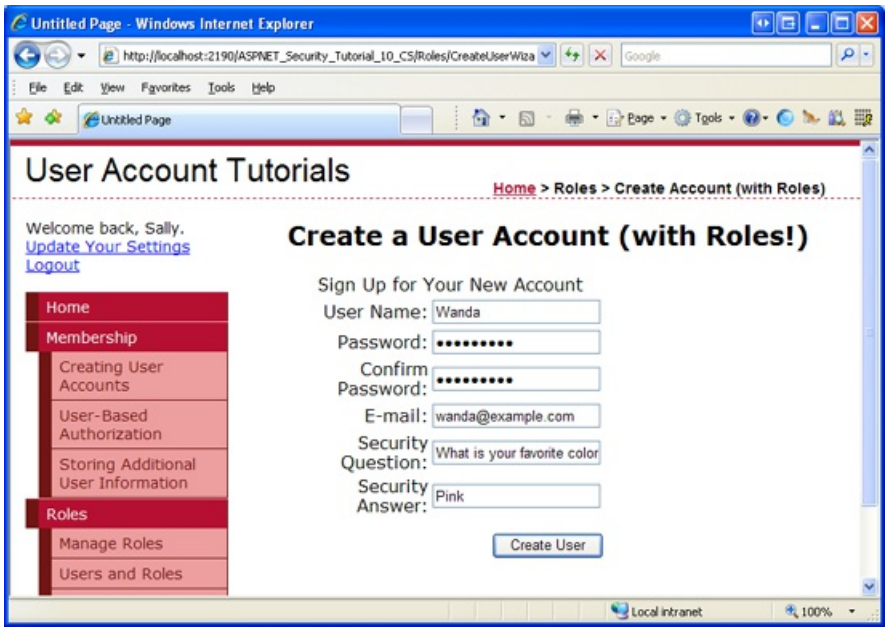


Figure 12: Create a New User Named Wanda (Click to view full-size image)

Click the "Create User" button. The CreateUserWizard internally calls the `Membership.CreateUser` method, creating the new user account, and then progresses to the next step, "Specify Roles." Here the system roles are listed. Check the Supervisors checkbox and click Next.

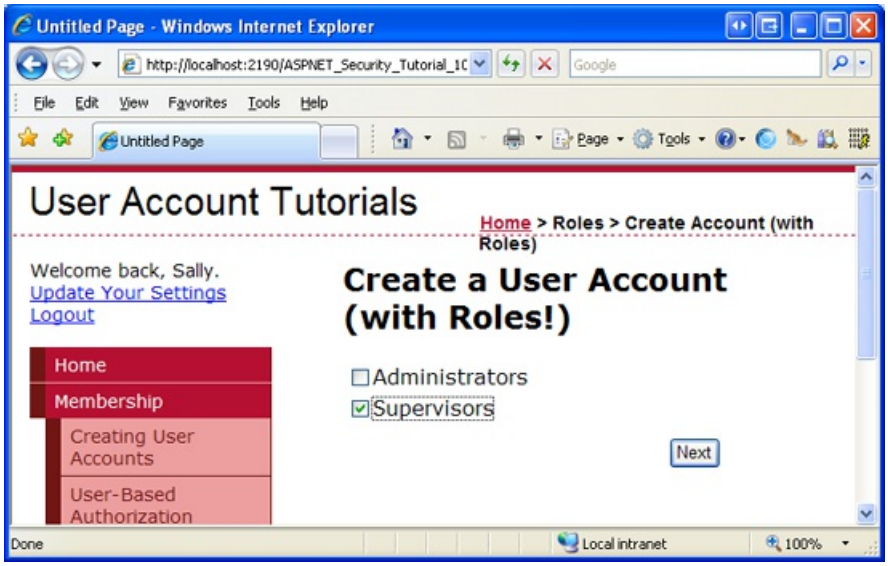


Figure 13: Make Wanda a Member of the Supervisors Role (Click to view full-size image)

Clicking Next causes a postback and updates the `ActiveStep` to the "Complete" step. In the `ActiveStepChanged` event handler, the recently-created user account is assigned to the Supervisors role. To verify this, return to the `UsersAndRoles.aspx` page and select Supervisors from the `RoleList` DropDownList. As Figure 14 shows, the Supervisors are now made up of three users: Bruce, Tito, and Wanda.

Figure 14: Bruce, Tito, and Wanda are All Supervisors (Click to view full-size image)

Summary

The Roles framework offers methods for retrieving information about a particular user's roles and methods for determining what users belong to a specified role. Furthermore, there are a number of methods for adding and removing one or more users to one or more roles. In this tutorial we focused on just two of these methods: `AddUserToRole` and `RemoveUserFromRole`. There are additional variants designed to add multiple users to a single role and to assign multiple roles to a single user.

This tutorial also included a look at extending the CreateUserWizard control to include a `WizardStep` to specify the newly-created user's roles. Such a step could help an administrator streamline the process of creating user accounts for new users.

At this point we have seen how to create and delete roles and how to add and remove users from roles. But we have yet to look at applying role-based authorization. In the following tutorial we will look at defining URL authorization rules on a role-by-role basis, as well as how to limit page-level functionality based on the currently logged in user's roles.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ASP.NET Web Site Administration Tool Overview
- Examining ASP.NET's Membership, Roles, and Profile
- Rolling Your Own Website Administration Tool

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Role-Based Authorization (VB)

2/20/2020 • 34 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

This tutorial starts with a look at how the Roles framework associates a user's roles with his security context. It then examines how to apply role-based URL authorization rules. Following that, we will look at using declarative and programmatic means for altering the data displayed and the functionality offered by an ASP.NET page.

Introduction

In the *User-Based Authorization* tutorial we saw how to use URL authorization to specify what users could visit a particular set of pages. With just a little bit of markup in `Web.config`, we could instruct ASP.NET to allow only authenticated users to visit a page. Or we could dictate that only users Tito and Bob were allowed, or indicate that all authenticated users except for Sam were permitted.

In addition to URL authorization, we also looked at declarative and programmatic techniques for controlling the data displayed and the functionality offered by a page based on the user visiting. In particular, we created a page that listed the contents of the current directory. Anyone could visit this page, but only authenticated users could view the files' contents and only Tito could delete the files.

Applying authorization rules on a user-by-user basis can grow into a bookkeeping nightmare. A more maintainable approach is to use role-based authorization. The good news is that the tools at our disposal for applying authorization rules work equally well with roles as they do for user accounts. URL authorization rules can specify roles instead of users. The LoginView control, which renders different output for authenticated and anonymous users, can be configured to display different content based on the logged in user's roles. And the Roles API includes methods for determining the logged in user's roles.

This tutorial starts with a look at how the Roles framework associates a user's roles with his security context. It then examines how to apply role-based URL authorization rules. Following that, we will look at using declarative and programmatic means for altering the data displayed and the functionality offered by an ASP.NET page. Let's get started!

Understanding How Roles are Associated with a User's Security Context

Whenever a request enters the ASP.NET pipeline it is associated with a security context, which includes information identifying the requestor. When using forms authentication, an authentication ticket is used as an identity token. As we discussed in the *An Overview of Forms Authentication* and *Forms Authentication Configuration and Advanced Topics* tutorials, the `FormsAuthenticationModule` is responsible for determining the identity of the requestor, which it does during the `AuthenticateRequest` event.

If a valid, non-expired authentication ticket is found, the `FormsAuthenticationModule` decodes it to ascertain the requestor's identity. It creates a new `GenericPrincipal` object and assigns this to the `HttpContext.User` object. The purpose of a principal, like `GenericPrincipal`, is to identify the authenticated user's name and what roles she belong to. This purpose is evident by the fact that all principal objects have an `Identity` property and an `IsInRole(roleName)` method. The `FormsAuthenticationModule`, however, is not interested in recording role

information and the `GenericPrincipal` object it creates does not specify any roles.

If the Roles framework is enabled, the `RoleManagerModule` HTTP Module steps in after the `FormsAuthenticationModule` and identifies the authenticated user's roles during the `PostAuthenticateRequest` event, which fires after the `AuthenticateRequest` event. If the request is from an authenticated user, the `RoleManagerModule` overwrites the `GenericPrincipal` object created by the `FormsAuthenticationModule` and replaces it with a `RolePrincipal` object. The `RolePrincipal` class uses the Roles API to determine what roles the user belongs to.

Figure 1 depicts the ASP.NET pipeline workflow when using forms authentication and the Roles framework. The `FormsAuthenticationModule` executes first, identifies the user via her authentication ticket, and creates a new `GenericPrincipal` object. Next, the `RoleManagerModule` steps in and overwrites the `GenericPrincipal` object with a `RolePrincipal` object.

If an anonymous user visits the site, neither the `FormsAuthenticationModule` nor the `RoleManagerModule` creates a principal object.

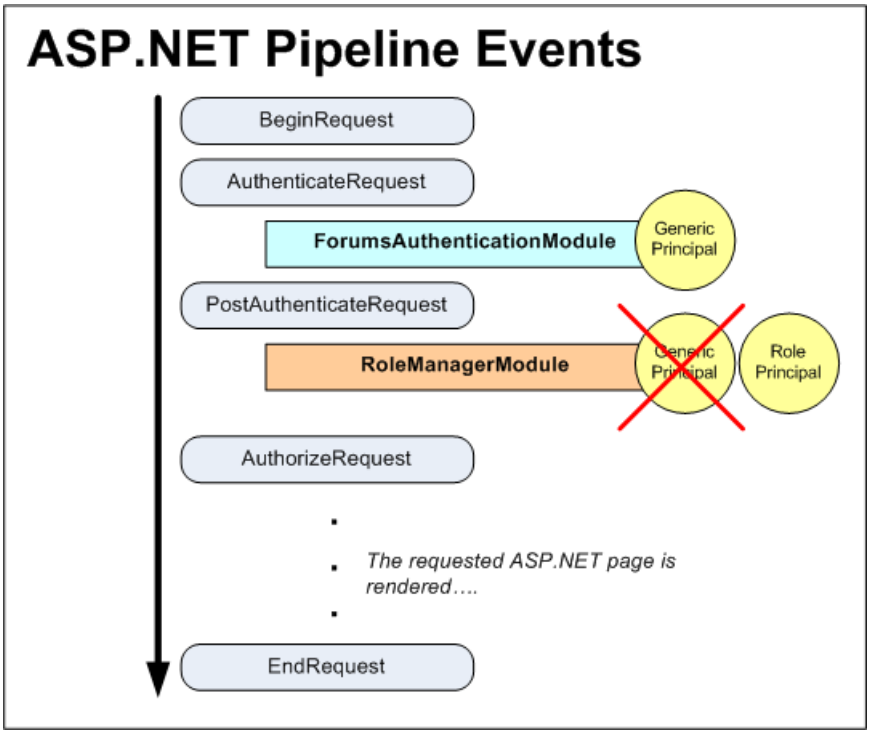


Figure 1: The ASP.NET Pipeline Events for an Authenticated User When Using Forms Authentication and the Roles Framework (Click to view full-size image)

Caching Role Information in a Cookie

The `RolePrincipal` object's `IsInRole(roleName)` method calls `Roles`.`GetRolesForUser` to get the roles for the user in order to determine whether the user is a member of *roleName*. When using the `SqlRoleProvider`, this results in a query to the role store database. When using role-based URL authorization rules the `RolePrincipal`'s `IsInRole` method will be called on every request to a page that is protected by the role-based URL authorization rules. Rather than have to lookup the role information in the database on every request, the `Roles` framework includes an option to cache the user's roles in a cookie.

If the Roles framework is configured to cache the user's roles in a cookie, the `RoleManagerModule` creates the cookie during the ASP.NET pipeline's `EndRequest` event. This cookie is used in subsequent requests in the `PostAuthenticateRequest`, which is when the `RolePrincipal` object is created. If the cookie is valid and has not expired, the data in the cookie is parsed and used to populate the user's roles, thereby saving the `RolePrincipal` from having to make a call to the `Roles` class to determine the user's roles. Figure 2 depicts this workflow.

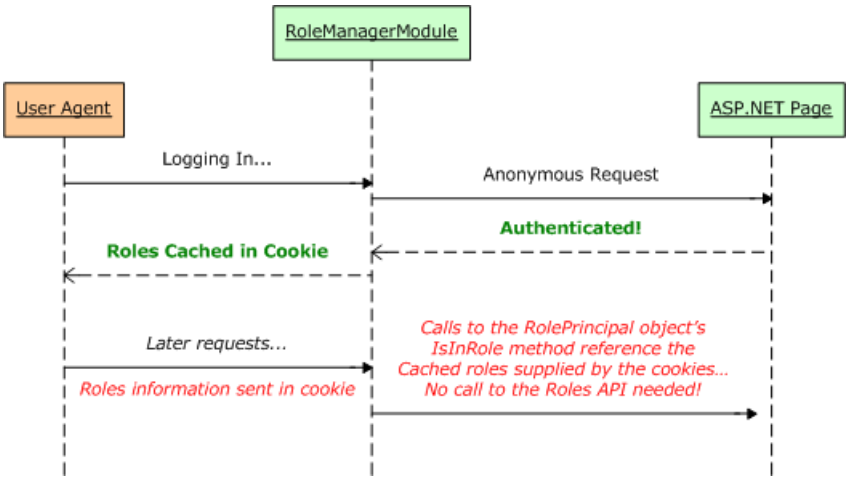


Figure 2: The User's Role Information Can Be Stored in a Cookie to Improve Performance (Click to view full-size image)

By default, the role cache cookie mechanism is disabled. It can be enabled through the `<roleManager>`; configuration markup in `Web.config`. We discussed using the `<roleManager>` element to specify Role providers in the *Creating and Managing Roles* tutorial, so you should already have this element in your application's `Web.config` file. The role cache cookie settings are specified as attributes of the `<roleManager>`; element, and are summarized in Table 1.

> **NOTE**
>
> The configuration settings listed in Table 1 specify the properties of the resulting role cache cookie. For more information on cookies, how they work, and their various properties, read this Cookies tutorial.

PROPERTY	DESCRIPTION
`cacheRolesInCookie`	A Boolean value that indicates whether cookie caching is used. Defaults to `false`.
`cookieName`	The name of the role cache cookie. The default value is ".ASPXROLES".
`cookiePath`	The path for the roles name cookie. The path attribute enables a developer to limit the scope of a cookie to a particular directory hierarchy. The default value is "/", which informs the browser to send the authentication ticket cookie to any request made to the domain.
`cookieProtection`	Indicates what techniques are used to protect the role cache cookie. The allowable values are: `All` (the default); `Encryption`; `None`; and `Validation`. Refer back to Step 3 of the *Forms Authentication Configuration and Advanced Topics* tutorial for more information on these protection levels.
`cookieRequireSSL`	A Boolean value that indicates whether an SSL connection is required to transmit the authentication cookie. The default value is `false`.

PROPERTY	DESCRIPTION
`cookieSlidingExpiration`	A Boolean value that indicates whether the cookie's timeout is reset each time the user visits the site during a single session. The default value is `false`. This value is only pertinent when `createPersistentCookie` is set to `true`.
`cookieTimeout`	Specifies the time, in minutes, after which the authentication ticket cookie expires. The default value is `30`. This value is only pertinent when `createPersistentCookie` is set to `true`.
`createPersistentCookie`	A Boolean value that specifies whether the role cache cookie is a session cookie or persistent cookie. If `false` (the default), a session cookie is used, which is deleted when the browser is closed. If `true`, a persistent cookie is used; it expires `cookieTimeout` number of minutes after it has been created or after the previous visit, depending on the value of `cookieSlidingExpiration`.
`domain`	Specifies the cookie's domain value. The default value is an empty string, which causes the browser to use the domain from which it was issued (such as www.yourdomain.com). In this case, the cookie will **not** be sent when making requests to subdomains, such as admin.yourdomain.com. If you want the cookie to be passed to all subdomains you need to customize the `domain` attribute, setting it to "yourdomain.com".
`maxCachedResults`	Specifies the maximum number of role names that are cached in the cookie. The default is 25. The `RoleManagerModule` does not create a cookie for users that belong to more than `maxCachedResults` roles. Consequently, the `RolePrincipal` object's `IsInRole` method will use the `Roles` class to determine the user's roles. The reason `maxCachedResults` exists is because many user agents do not permit cookies larger than 4,096 bytes. So this cap is meant to reduce the likelihood of exceeding this size limitation. If you have extremely long role names, you may want to consider specifying a smaller `maxCachedResults` value; contrariwise, if you have extremely short role names, you can probably increase this value.

Table 1: The Role Cache Cookie Configuration Options

Let's configure our application to use non-persistent role cache cookies. To accomplish this, update the `<roleManager>` element in `Web.config` to include the following cookie-related attributes:

```
<roleManager enabled="true"
          defaultProvider="SecurityTutorialsSqlRoleProvider"
          cacheRolesInCookie="true"
          createPersistentCookie="false"
          cookieProtection="All">

     <providers>
     ...
     </providers>
</roleManager>
```

I updated the `<roleManager>` element by adding three attributes: `cacheRolesInCookie`, `createPersistentCookie`, and `cookieProtection`. By setting `cacheRolesInCookie` to `true`, the `RoleManagerModule` will now automatically cache the user's roles in a cookie rather than having to lookup the user's role information on each request. I explicitly set the `createPersistentCookie` and `cookieProtection` attributes to `false` and `All`, respectively. Technically, I didn't need to specify values for these attributes since I just assigned them to their default values, but I put them here to make it explicitly clear that I am not using persistent cookies and that the cookie is both encrypted and validated.

That's all there is to it! Henceforth, the Roles framework will cache the users' roles in cookies. If the user's browser does not support cookies, or if their cookies are deleted or lost, somehow, it's no big deal – the `RolePrincipal` object will simply use the `Roles` class in the case that no cookie (or an invalid or expired one) is available.

> **NOTE**
> Microsoft's Patterns & Practices group discourages using persistent role cache cookies. Since possession of the role cache cookie is sufficient to prove role membership, if a hacker can somehow gain access to a valid user's cookie he can impersonate that user. The likelihood of this happening increases if the cookie is persisted on the user's browser. For more information on this security recommendation, as well as other security concerns, refer to the Security Question List for ASP.NET 2.0.

Step 1: Defining Role-Based URL Authorization Rules

As discussed in the *User-Based Authorization* tutorial, URL authorization offers a means to restrict access to a set of pages on a user-by-user or role-by-role basis. The URL authorization rules are spelled out in `Web.config` using the `<authorization>` element with `<allow>` and `<deny>` child elements. In addition to the user-related authorization rules discussed in previous tutorials, each `<allow>` and `<deny>` child element can also include:

- A particular role
- A comma-delimited list of roles

For example, the URL authorization rules grant access to those users in the Administrators and Supervisors roles, but deny access to all others:

```
<authorization>

    <allow roles="Administrators, Supervisors" />
    <deny users="*" />
</authorization>
```

The `<allow>` element in the above markup states that the Administrators and Supervisors roles are allowed; the `<deny>` element instructs that *all* users are denied.

Let's configure our application so that the `ManageRoles.aspx`, `UsersAndRoles.aspx`, and `CreateUserWizardWithRoles.aspx` pages are only accessible to those users in the Administrators role, while the `RoleBasedAuthorization.aspx` page remains accessible to all visitors.

To accomplish this, start by adding a `Web.config` file to the `Roles` folder.

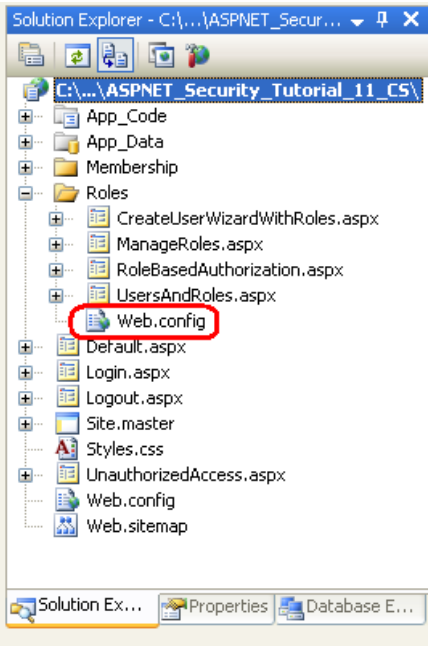


Figure 3: Add a `Web.config` File to the `Roles` directory (Click to view full-size image)

Next, add the following configuration markup to `Web.config`:

```
<?xml version="1.0"?>

<configuration>
     <system.web>
          <authorization>
               <allow roles="Administrators" />
               <deny users="*"/>
          </authorization>

     </system.web>

     <!-- Allow all users to visit RoleBasedAuthorization.aspx -->
     <location path="RoleBasedAuthorization.aspx">
          <system.web>
               <authorization>
                    <allow users="*" />

               </authorization>
          </system.web>
     </location>
</configuration>
```

The `<authorization>` element in the `<system.web>` section indicates that only users in the Administrators role may access the ASP.NET resources in the `Roles` directory. The `<location>` element defines an alternate set of URL authorization rules for the `RoleBasedAuthorization.aspx` page, allowing all users to visit the page.

After saving your changes to `Web.config`, log in as a user that is not in the Administrators role and then try to visit one of the protected pages. The `UrlAuthorizationModule` will detect that you do not have permission to visit the requested resource; consequently, the `FormsAuthenticationModule` will redirect you to the login page. The login page will then redirect you to the `UnauthorizedAccess.aspx` page (see Figure 4). This final redirect from the login page to `UnauthorizedAccess.aspx` occurs because of code we added to the login page in Step 2 of the *User-Based Authorization* tutorial. In particular, the login page automatically redirects any authenticated user to `UnauthorizedAccess.aspx` if the querystring contains a `ReturnUrl` parameter, as this parameter indicates that the user arrived at the login page after attempting to view a page he was not authorized to view.

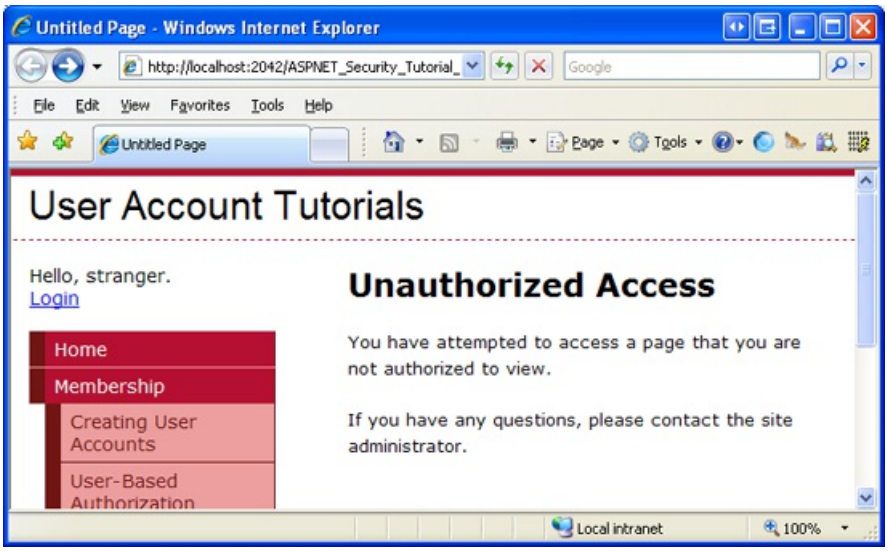


Figure 4: Only Users in the Administrators Role Can View the Protected Pages (Click to view full-size image)

Log off and then log in as a user that is in the Administrators role. Now you should be able to view the three protected pages.

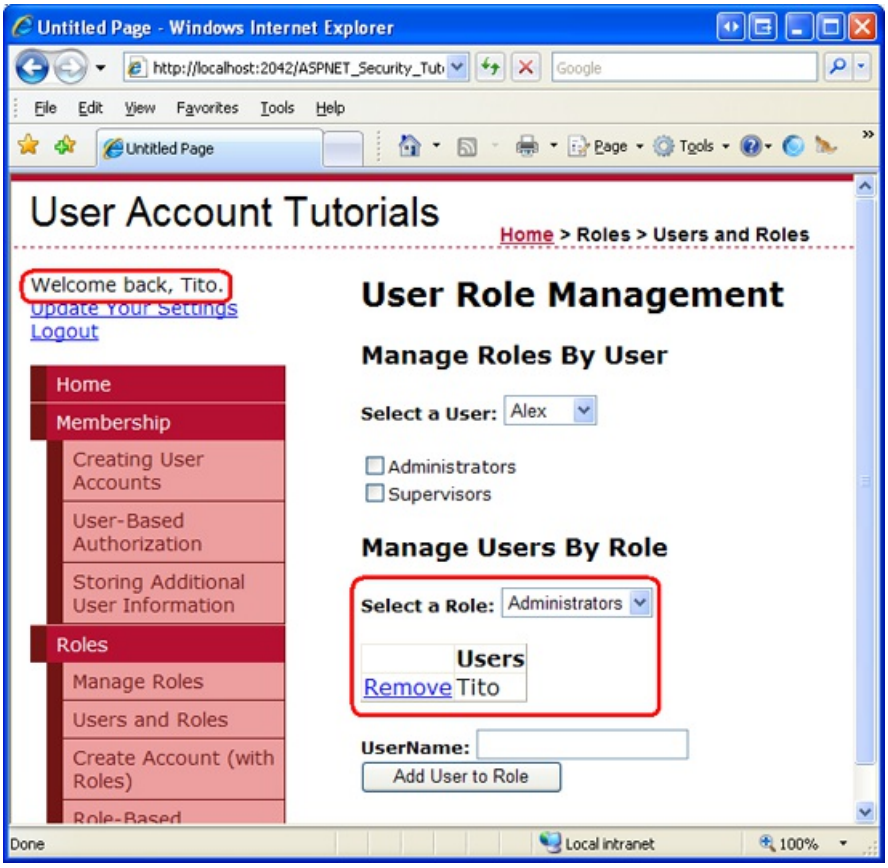


Figure 5: Tito Can Visit the `UsersAndRoles.aspx` Page Because He is in the Administrators Role (Click to view full-size image)

NOTE

When specifying URL authorization rules – for roles or users – it is important to keep in mind that the rules are analyzed one at a time, from the top down. As soon as a match is found, the user is granted or denied access, depending on if the match was found in an `<allow>` or `<deny>` element. **If no match is found, the user is granted access.** Consequently, if you want to restrict access to one or more user accounts, it is imperative that you use a `<deny>` element as the last element in the URL authorization configuration. **If your URL authorization rules do not include a `<deny>` element, all users will be granted access.** For a more thorough discussion on how the URL authorization rules are analyzed, refer back to the "A Look at How the `UrlAuthorizationModule` Uses the Authorization Rules to Grant or Deny Access" section of the *User-Based Authorization* tutorial.

Step 2: Limiting Functionality Based On the Currently Logged In User's Roles

URL authorization makes it easy to specify coarse authorization rules that state what identities are permitted and which ones are denied from viewing a particular page (or all pages in a folder and its subfolders). However, in certain cases we may want to allow all users to visit a page, but limit the page's functionality based on the visiting user's roles. This may entail showing or hiding data based on the user's role, or offering additional functionality to users that belong to a particular role.

Such fine grain role-based authorization rules can be implemented either declaratively or programmatically (or through some combination of the two). In the next section we will see how to implement declarative fine grain authorization via the LoginView control. Following that, we will explore programmatic techniques. Before we can look at applying fine grain authorization rules, however, we first need to create a page whose functionality depends on the role of the user visiting it.

Let's create a page that lists all of the user accounts in the system in a GridView. The GridView will include each user's username, email address, last login date, and comments about the user. In addition to displaying each user's information, the GridView will include edit and delete capabilities. We will initially create this page with the edit and delete functionality available to all users. In the "Using the LoginView Control" and "Programmatically Limiting Functionality" sections we will see how to enable or disable these features based on the visiting user's role.

NOTE

The ASP.NET page we are about to build uses a GridView control to display the user accounts. Since this tutorial series focuses on forms authentication, authorization, user accounts, and roles, I do not want to spend too much time discussing the inner workings of the GridView control. While this tutorial provides specific step-by-step instructions for setting up this page, it does not delve into the details of why certain choices were made, or what effect particular properties have on the rendered output. For a thorough examination of the GridView control, check out my *Working with Data in ASP.NET 2.0* tutorial series.

Start by opening the `RoleBasedAuthorization.aspx` page in the `Roles` folder. Drag a GridView from the page onto the Designer and set its `ID` to `UserGrid`. In a moment we will write code that calls the `Membership`. `GetAllUsers` method and binds the resulting `MembershipUserCollection` object to the GridView. The `MembershipUserCollection` contains a `MembershipUser` object for each user account in the system; `MembershipUser` objects have properties like `UserName`, `Email`, `LastLoginDate` and so forth.

Before we write the code that binds the user accounts to the grid, let's first define the GridView's fields. From the GridView's Smart Tag, click the "Edit Columns" link to launch the Fields dialog box (see Figure 6). From here, uncheck the "Auto-generate fields" checkbox in the lower left corner. Since we want this GridView to include editing and deleting capabilities, add a CommandField and set its `ShowEditButton` and `ShowDeleteButton`

properties to True. Next, add four fields for displaying the `UserName`, `Email`, `LastLoginDate`, and `Comment` properties. Use a BoundField for the two read-only properties (`UserName` and `LastLoginDate`) and TemplateFields for the two editable fields (`Email` and `Comment`).

Have the first BoundField display the `UserName` property; set its `HeaderText` and `DataField` properties to "UserName". This field will not be editable, so set its `ReadOnly` property to True. Configure the `LastLoginDate` BoundField by setting its `HeaderText` to "Last Login" and its `DataField` to "LastLoginDate". Let's format the output of this BoundField so that just the date is displayed (instead of the date and time). To accomplish this, set this BoundField's `HtmlEncode` property to False and its `DataFormatString` property to "{0:d}". Also set the `ReadOnly` property to True.

Set the `HeaderText` properties of the two TemplateFields to "Email" and "Comment".

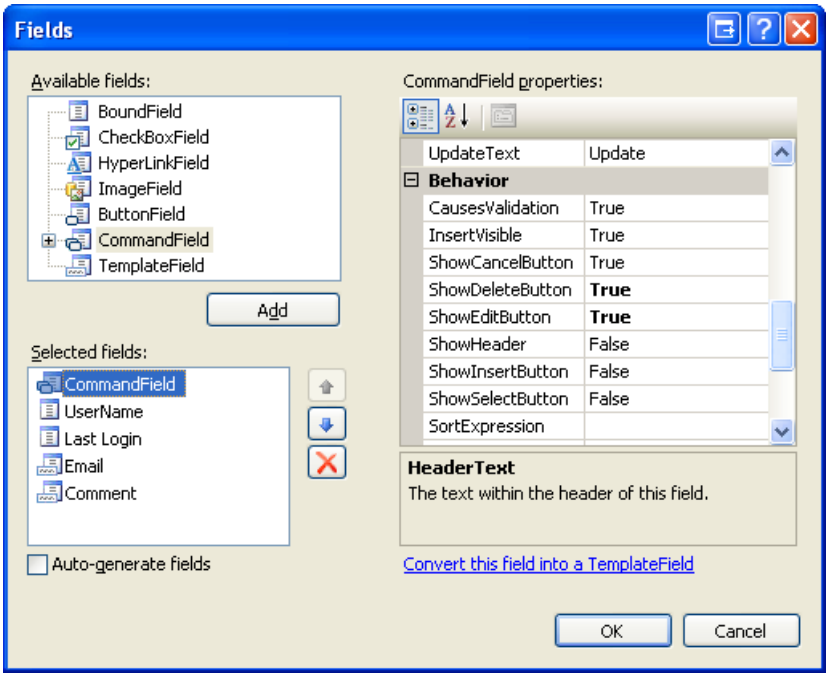


Figure 6: The GridView's Fields Can Be Configured Through the Fields Dialog Box (Click to view full-size image)

We now need to define the `ItemTemplate` and `EditItemTemplate` for the "Email" and "Comment" TemplateFields. Add a Label Web control to each of the `ItemTemplates` and bind their `Text` properties to the `Email` and `Comment` properties, respectively.

For the "Email" TemplateField, add a TextBox named `Email` to its `EditItemTemplate` and bind its `Text` property to the `Email` property using two-way databinding. Add a RequiredFieldValidator and RegularExpressionValidator to the `EditItemTemplate` to ensure that a visitor editing the Email property has entered a valid email address. For the "Comment" TemplateField, add a multi-line TextBox named `Comment` to its `EditItemTemplate`. Set the TextBox's `Columns` and `Rows` properties to 40 and 4, respectively, and then bind its `Text` property to the `Comment` property using two-way databinding.

After configuring these TemplateFields, their declarative markup should look similar to the following:

```
<asp:TemplateField HeaderText="Email">
     <ItemTemplate>
          <asp:Label runat="server" ID="Label1" Text='<%# Eval("Email")%>'></asp:Label>

     </ItemTemplate>
     <EditItemTemplate>
          <asp:TextBox runat="server" ID="Email" Text='<%# Bind("Email")%>'></asp:TextBox>

          <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
               ControlToValidate="Email" Display="Dynamic"
               ErrorMessage="You must provide an email address."
               SetFocusOnError="True">*</asp:RequiredFieldValidator>

          <asp:RegularExpressionValidator ID="RegularExpressionValidator1" runat="server"
               ControlToValidate="Email" Display="Dynamic"
               ErrorMessage="The email address you have entered is not valid. Please fix
               this and try again."
               SetFocusOnError="True"

               ValidationExpression="\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*">*
          </asp:RegularExpressionValidator>
     </EditItemTemplate>
</asp:TemplateField>

<asp:TemplateField HeaderText="Comment">
     <ItemTemplate>
          <asp:Label runat="server" ID="Label2" Text='<%# Eval("Comment")%>'></asp:Label>

     </ItemTemplate>
     <EditItemTemplate>
          <asp:TextBox runat="server" ID="Comment" TextMode="MultiLine"
               Columns="40" Rows="4" Text='<%# Bind("Comment")%>'>

          </asp:TextBox>
     </EditItemTemplate>
</asp:TemplateField>
```

When editing or deleting a user account we will need to know that user's `UserName` property value. Set the GridView's `DataKeyNames` property to "UserName" so that this information is available through the GridView's `DataKeys` collection.

Finally, add a ValidationSummary control to the page and set its `ShowMessageBox` property to True and its `ShowSummary` property to False. With these settings, the ValidationSummary will display a client-side alert if the user attempts to edit a user account with a missing or invalid email address.

```
<asp:ValidationSummary ID="ValidationSummary1"
               runat="server"
               ShowMessageBox="True"
               ShowSummary="False" />
```

We have now completed this page's declarative markup. Our next task is to bind the set of user accounts to the GridView. Add a method named `BindUserGrid` to the `RoleBasedAuthorization.aspx` page's code-behind class that binds the `MembershipUserCollection` returned by `Membership.GetAllUsers` to the `UserGrid` GridView. Call this method from the `Page_Load` event handler on the first page visit.

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
     If Not Page.IsPostBack Then
          BindUserGrid()
     End If
End Sub

Private Sub BindUserGrid()
     Dim allUsers As MembershipUserCollection = Membership.GetAllUsers()
     UserGrid.DataSource = allUsers
     UserGrid.DataBind()
End Sub
```

With this code in place, visit the page through a browser. As Figure 7 shows, you should see a GridView listing information about each user account in the system.

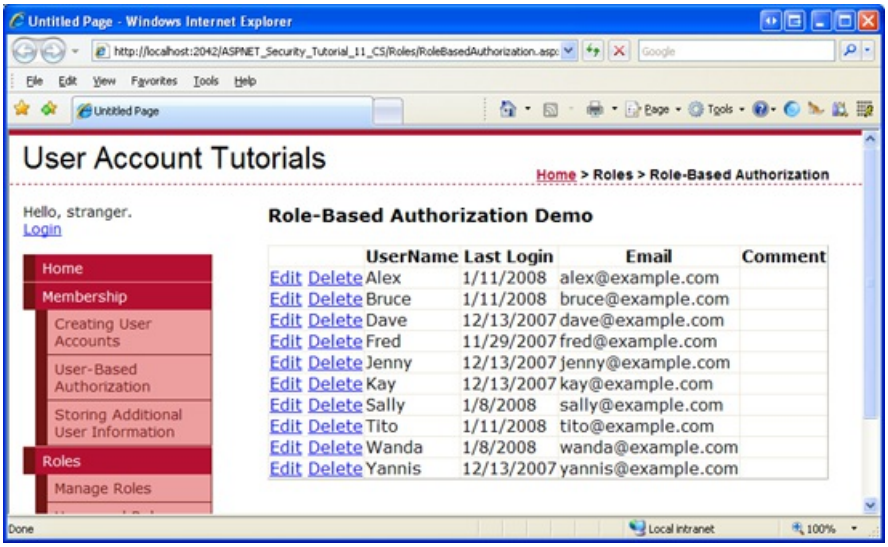


Figure 7: The `UserGrid` GridView Lists Information About Each User in the System (Click to view full-size image)

> **NOTE**
>
> The `UserGrid` GridView lists all of the users in a non-paged interface. This simple grid interface is not suitable for scenarios where there are several dozen or more users. One option is to configure the GridView to enable paging. The `Membership.GetAllUsers` method has two overloads: one that accepts no input parameters and returns all of the users and one that takes in integer values for the page index and page size, and returns only the specified subset of the users. The second overload can be used to more efficiently page through the users since it returns just the precise subset of user accounts rather than *all* of them. If you have thousands of user accounts, you might want to consider a filter-based interface, one that only shows those users whose UserName begins with a selected character, for instance. The `Membership.FindUsersByName` method is ideal for building a filter-based user interface. We will look at building such an interface in a future tutorial.

The GridView control offers built-in editing and deleting support when the control is bound to a properly configured data source control, such as the SqlDataSource or ObjectDataSource. The `UserGrid` GridView, however, has its data programmatically bound; therefore, we must write code to perform these two tasks. In particular, we will need to create event handlers for the GridView's `RowEditing`, `RowCancelingEdit`, `RowUpdating`, and `RowDeleting` events, which are fired when a visitor clicks the GridView's Edit, Cancel, Update, or Delete buttons.

Start by creating the event handlers for the GridView's `RowEditing`, `RowCancelingEdit`, and `RowUpdating` events and then add the following code:

```
Protected Sub UserGrid_RowEditing(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.GridViewEditEventArgs) Handles UserGrid.RowEditing
     ' Set the grid's EditIndex and rebind the data

     UserGrid.EditIndex = e.NewEditIndex
     BindUserGrid()
End Sub

Protected Sub UserGrid_RowCancelingEdit(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.GridViewCancelEditEventArgs) Handles UserGrid.RowCancelingEdit
     ' Revert the grid's EditIndex to -1 and rebind the data
     UserGrid.EditIndex = -1
     BindUserGrid()
End Sub

Protected Sub UserGrid_RowUpdating(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.GridViewUpdateEventArgs) Handles UserGrid.RowUpdating
     ' Exit if the page is not valid
     If Not Page.IsValid Then
          Exit Sub
     End If

     ' Determine the username of the user we are editing
     Dim UserName As String = UserGrid.DataKeys(e.RowIndex).Value.ToString()

     ' Read in the entered information and update the user
     Dim EmailTextBox As TextBox = CType(UserGrid.Rows(e.RowIndex).FindControl("Email"),TextBox)
     Dim CommentTextBox As TextBox= CType(UserGrid.Rows(e.RowIndex).FindControl("Comment"),TextBox)

     ' Return information about the user
     Dim UserInfo As MembershipUser = Membership.GetUser(UserName)

     ' Update the User account information
     UserInfo.Email = EmailTextBox.Text.Trim()
     UserInfo.Comment = CommentTextBox.Text.Trim()

     Membership.UpdateUser(UserInfo)

     ' Revert the grid's EditIndex to -1 and rebind the data
     UserGrid.EditIndex = -1
     BindUserGrid()
End Sub
```

The `RowEditing` and `RowCancelingEdit` event handlers simply set the GridView's `EditIndex` property and then rebind the list of user accounts to the grid. The interesting stuff happens in the `RowUpdating` event handler. This event handler starts by ensuring that the data is valid and then grabs the `UserName` value of the edited user account from the `DataKeys` collection. The `Email` and `Comment` TextBoxes in the two TemplateFields' `EditItemTemplate` s are then programmatically referenced. Their `Text` properties contain the edited email address and comment.

In order to update a user account through the Membership API we need to first get the user's information, which we do via a call to `Membership.GetUser(userName)`. The returned `MembershipUser` object's `Email` and `Comment` properties are then updated with the values entered into the two TextBoxes from the editing interface. Finally, these modifications are saved with a call to `Membership.UpdateUser`. The `RowUpdating` event handler completes by reverting the GridView to its pre-editing interface.

Next, create the `RowDeleting` RowDeleting event handler and then add the following code:

```
Protected Sub UserGrid_RowDeleting(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.GridViewDeleteEventArgs) Handles UserGrid.RowDeleting

     ' Determine the username of the user we are editing
     Dim UserName As String = UserGrid.DataKeys(e.RowIndex).Value.ToString()

     ' Delete the user
     Membership.DeleteUser(UserName)

     ' Revert the grid's EditIndex to -1 and rebind the data
     UserGrid.EditIndex = -1
     BindUserGrid()
End Sub
```

The above event handler starts by grabbing the `UserName` value from the GridView's `DataKeys` collection; this `UserName` value is then passed into the Membership class's `DeleteUser` method. The `DeleteUser` method deletes the user account from the system, including related membership data (such as what roles this user belongs to). After deleting the user, the grid's `EditIndex` is set to -1 (in case the user clicked Delete while another row was in edit mode) and the `BindUserGrid` method is called.

> **NOTE**
>
> The Delete button does not require any sort of confirmation from the user before deleting the user account. I encourage you to add some form of user confirmation to lessen the chance of an account being accidentally deleted. One of the easiest ways to confirm an action is through a client-side confirm dialog box. For more information on this technique, see Adding Client-Side Confirmation When Deleting.

Verify that this page functions as expected. You should be able to edit any user's email address and comment, as well as delete any user account. Since the `RoleBasedAuthorization.aspx` page is accessible to all users, any user – even anonymous visitors – can visit this page and edit and delete user accounts! Let's update this page so that only users in the Supervisors and Administrators roles can edit a user's email address and comment, and only Administrators can delete a user account.

The "Using the LoginView Control" section looks at using the LoginView control to show instructions specific to the user's role. If a person in the Administrators role visits this page, we will show instructions on how to edit and delete users. If a user in the Supervisors role reaches this page, we will show instructions on editing users. And if the visitor is anonymous or is not in either the Supervisors or Administrators role, we will display a message explaining that they cannot edit or delete user account information. In the "Programmatically Limiting Functionality" section we will write code that programmatically shows or hides the Edit and Delete buttons based on the user's role.

Using the LoginView Control

As we've seen in past tutorials, the LoginView control is useful for displaying different interfaces for authenticated and anonymous users, but the LoginView control can also be used to display different markup based on the user's roles. Let's use a LoginView control to display different instructions based on the visiting user's role.

Start by adding a LoginView above the `UserGrid` GridView. As we've discussed earlier, the LoginView control has two built-in templates: `AnonymousTemplate` and `LoggedInTemplate`. Enter a brief message in both of these templates that informs the user that they cannot edit or delete any user information.

```
<asp:LoginView ID="LoginView1" runat="server">
     <LoggedInTemplate>
          You are not a member of the Supervisors or Administrators roles. Therefore you
           cannot edit or delete any user information.
     </LoggedInTemplate>
     <AnonymousTemplate>

          You are not logged into the system. Therefore you cannot edit or delete any user
           information.
     </AnonymousTemplate>
</asp:LoginView>
```

In addition to the `AnonymousTemplate` and `LoggedInTemplate`, the LoginView control can include *RoleGroups*, which are role-specific templates. Each RoleGroup contains a single property, `Roles`, which specifies what roles the RoleGroup applies to. The `Roles` property can be set to a single role (like "Administrators") or to a comma-delimited list of roles (like "Administrators, Supervisors").

To manage the RoleGroups, click the "Edit RoleGroups" link from the control's Smart Tag to bring up the RoleGroup Collection Editor. Add two new RoleGroups. Set the first RoleGroup's `Roles` property to "Administrators" and the second's to "Supervisors".

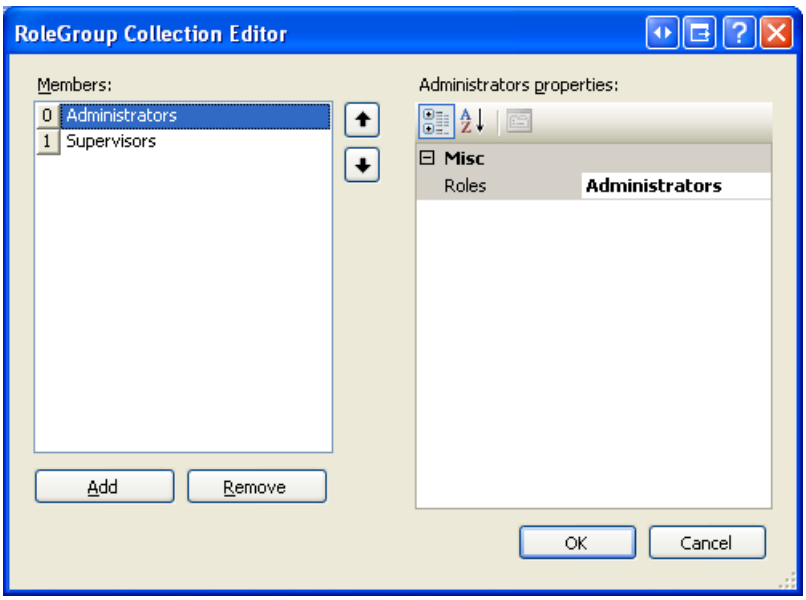


Figure 8: Manage the LoginView's Role-Specific Templates Through the RoleGroup Collection Editor (Click to view full-size image)

Click OK to close the RoleGroup Collection Editor; this updates the LoginView's declarative markup to include a `<RoleGroups>` section with an `<asp:RoleGroup>` child element for each RoleGroup defined in the RoleGroup Collection Editor. Furthermore, the "Views" drop-down list in the LoginView's Smart Tag - which initially listed just the `AnonymousTemplate` and `LoggedInTemplate` – now includes the added RoleGroups as well.

Edit the RoleGroups so that users in the Supervisors role are displayed instructions on how to edit user accounts, while users in the Administrators role are shown instructions for editing and deleting. After making these changes, your LoginView's declarative markup should look similar to the following.

```
<asp:LoginView ID="LoginView1" runat="server">
     <RoleGroups>
          <asp:RoleGroup Roles="Administrators">

               <ContentTemplate>
                    As an Administrator, you may edit and delete user accounts.
                    Remember: With great power comes great responsibility!
               </ContentTemplate>
          </asp:RoleGroup>
          <asp:RoleGroup Roles="Supervisors">
               <ContentTemplate>
                    As a Supervisor, you may edit users&#39; Email and Comment information.
                    Simply click the Edit button, make your changes, and then click Update.
               </ContentTemplate>

          </asp:RoleGroup>
     </RoleGroups>
     <LoggedInTemplate>
          You are not a member of the Supervisors or Administrators roles.
          Therefore you cannot edit or delete any user information.
     </LoggedInTemplate>
     </AnonymousTemplate>
          You are not logged into the system.
          Therefore you cannot edit or delete any user information.
     </AnonymousTemplate>
</asp:LoginView>
```

After making these changes, save the page and then visit it through a browser. First visit the page as an anonymous user. You should be shown the message, "You are not logged into the system. Therefore you cannot edit or delete any user information." Then log in as an authenticated user, but one that is neither in the Supervisors nor Administrators role. This time you should see the message, "You are not a member of the Supervisors or Administrators roles. Therefore you cannot edit or delete any user information."

Next, log in as a user who is a member of the Supervisors role. This time you should see the Supervisors role-specific message (see Figure 9). And if you log in as a user in the Administrators role you should see the Administrators role-specific message (see Figure 10).

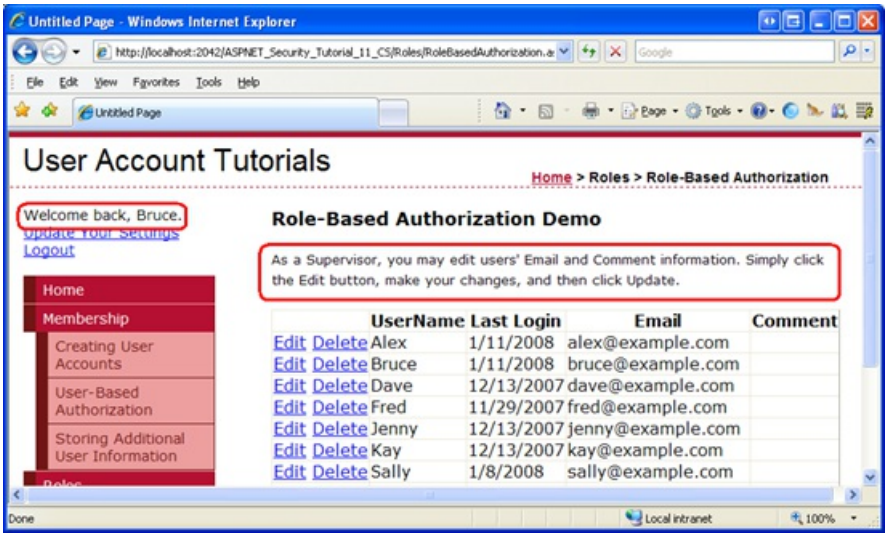


Figure 9: Bruce is Shown the Supervisors Role-Specific Message (Click to view full-size image)

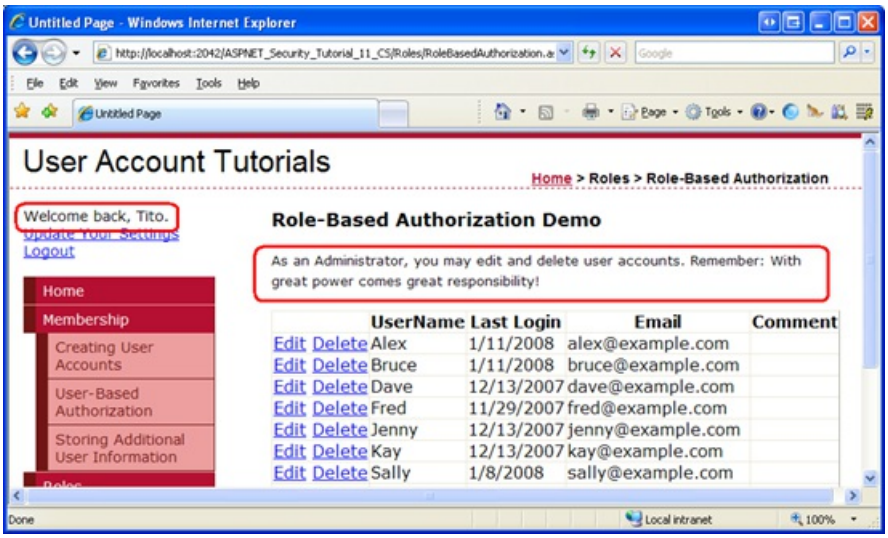


Figure 10: Tito is Shown the Administrators Role-Specific Message (Click to view full-size image)

As the screen shots in Figures 9 and 10 show, the LoginView only renders one template, even if multiple templates apply. Bruce and Tito are both logged in users, yet the LoginView renders only the matching RoleGroup and not the `LoggedInTemplate`. Moreover, Tito belongs to both the Administrators and Supervisors roles, yet the LoginView control renders the Administrators role-specific template instead of the Supervisors one.

Figure 11 illustrates the workflow used by the LoginView control to determine what template to render. Note that if there is more than one RoleGroup specified, the LoginView template renders the *first* RoleGroup that matches. In other words, if we had placed the Supervisors RoleGroup as the first RoleGroup and the Administrators as the second, then when Tito visited this page he would see the Supervisors message.

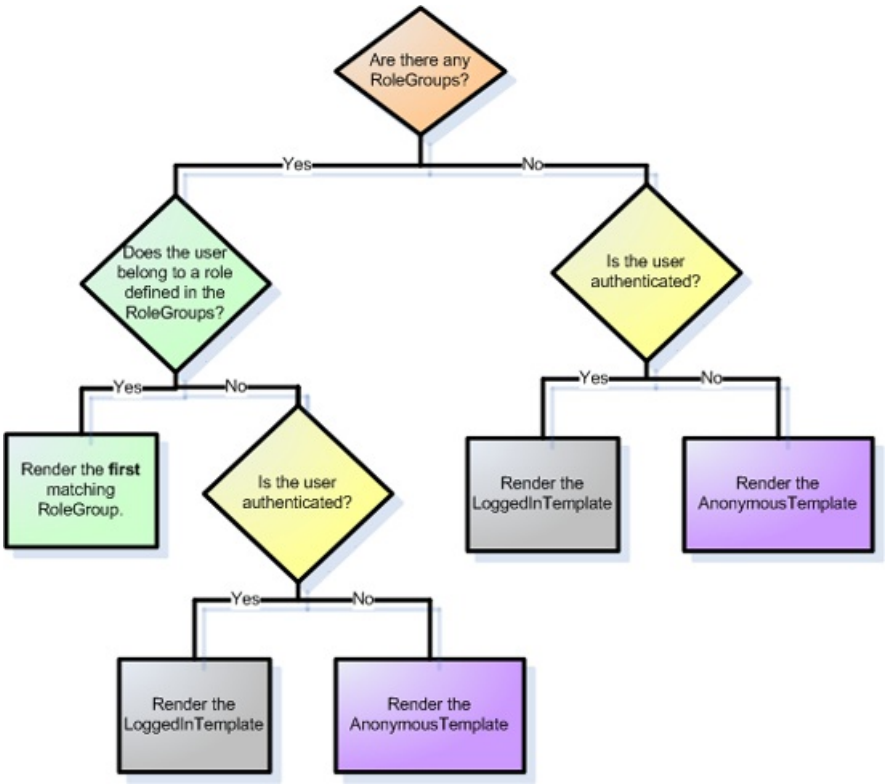


Figure 11: The LoginView Control's Workflow for Determining What Template to Render (Click to view full-size image)

Programmatically Limiting Functionality

While the LoginView control displays different instructions based on the role of the user visiting the page, the Edit and Cancel buttons remain visible to all. We need to programmatically hide the Edit and Delete buttons for

anonymous visitors and users who are in neither the Supervisors nor Administrators role. We need to hide the Delete button for everyone who is not an Administrator. In order to accomplish this we will write a bit of code that programmatically references the CommandField's Edit and Delete LinkButtons and sets their `Visible` properties to `False`, if necessary.

The easiest way to programmatically reference controls in a CommandField is to first convert it to a template. To accomplish this, click the "Edit Columns" link from the GridView's Smart Tag, select the CommandField from the list of current fields, and click the "Convert this field into a TemplateField" link. This turns the CommandField into a TemplateField with an `ItemTemplate` and `EditItemTemplate`. The `ItemTemplate` contains the Edit and Delete LinkButtons while the `EditItemTemplate` houses the Update and Cancel LinkButtons.

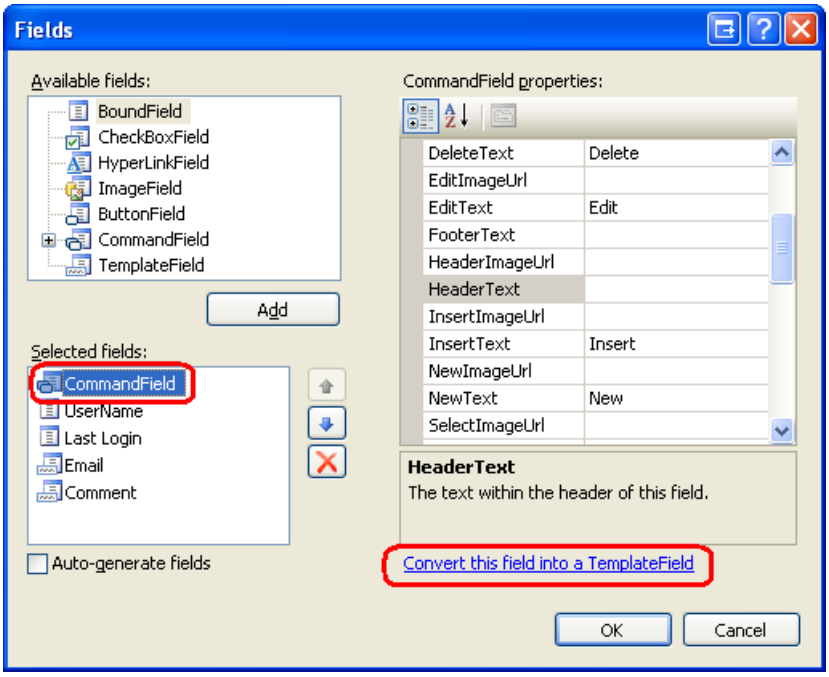


Figure 12: Convert the CommandField Into a TemplateField (Click to view full-size image)

Update the Edit and Delete LinkButtons in the `ItemTemplate`, setting their `ID` properties to values of `EditButton` and `DeleteButton`, respectively.

```
<asp:TemplateField ShowHeader="False">
     <EditItemTemplate>
          <asp:LinkButton ID="LinkButton1" runat="server" CausesValidation="True"
               CommandName="Update" Text="Update"></asp:LinkButton>

           <asp:LinkButton ID="LinkButton2" runat="server" CausesValidation="False"
               CommandName="Cancel" Text="Cancel"></asp:LinkButton>

     </EditItemTemplate>
     <ItemTemplate>
          <asp:LinkButton ID="EditButton" runat="server" CausesValidation="False"
               CommandName="Edit" Text="Edit"></asp:LinkButton>

           <asp:LinkButton ID="DeleteButton" runat="server" CausesValidation="False"
               CommandName="Delete" Text="Delete"></asp:LinkButton>

     </ItemTemplate>
</asp:TemplateField>
```

Whenever data is bound to the GridView, the GridView enumerates the records in its `DataSource` property and generates a corresponding `GridViewRow` object. As each `GridViewRow` object is created, the `RowCreated` event is fired. In order to hide the Edit and Delete buttons for unauthorized users, we need to create an event handler for this event and programmatically reference the Edit and Delete LinkButtons, setting their `Visible` properties

accordingly.

Create an event handler the `RowCreated` event and then add the following code:

```
Protected Sub UserGrid_RowCreated(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.GridViewRowEventArgs) Handles UserGrid.RowCreated
     If e.Row.RowType = DataControlRowType.DataRow AndAlso e.Row.RowIndex <> UserGrid.EditIndex Then
          ' Programmatically reference the Edit and Delete LinkButtons
          Dim EditButton As LinkButton = CType(e.Row.FindControl("EditButton"), LinkButton)

          Dim DeleteButton As LinkButton = CType(e.Row.FindControl("DeleteButton"), LinkButton)

          EditButton.Visible = (User.IsInRole("Administrators") OrElse User.IsInRole("Supervisors"))
          DeleteButton.Visible = User.IsInRole("Administrators")
     End If
End Sub
```

Keep in mind that the `RowCreated` event fires for *all* of the GridView rows, including the header, the footer, the pager interface, and so forth. We only want to programmatically reference the Edit and Delete LinkButtons if we are dealing with a data row not in edit mode (since the row in edit mode has Update and Cancel buttons instead of Edit and Delete). This check is handled by the `If` statement.

If we are dealing with a data row that is not in edit mode, the Edit and Delete LinkButtons are referenced and their `Visible` properties are set based on the Boolean values returned by the `User` object's `IsInRole(roleName)` method. The `User` object references the principal created by the `RoleManagerModule`; consequently, the `IsInRole(roleName)` method uses the Roles API to determine whether the current visitor belongs to *roleName*.

> **NOTE**
>
> We could have used the Roles class directly, replacing the call to `User.IsInRole(roleName)` with a call to the `Roles.IsUserInRole(roleName)` method. I decided to use the principal object's `IsInRole(roleName)` method in this example because it is more efficient than using the Roles API directly. Earlier in this tutorial we configured the role manager to cache the user's roles in a cookie. This cached cookie data is only utilized when the principal's `IsInRole(roleName)` method is called; direct calls to the Roles API always involve a trip to the role store. Even if roles are not cached in a cookie, calling the principal object's `IsInRole(roleName)` method is usually more efficient because when it is called for the first time during a request it caches the results. The Roles API, on the other hand, does not perform any caching. Because the `RowCreated` event is fired once for every row in the GridView, using `User.IsInRole(roleName)` involves just one trip to the role store whereas `Roles.IsUserInRole(roleName)` requires *N* trips, where *N* is the number of user accounts displayed in the grid.

The Edit button's `Visible` property is set to `True` if the user visiting this page is in the Administrators or Supervisors role; otherwise it is set to `False`. The Delete button's `Visible` property is set to `True` only if the user is in the Administrators role.

Test this page through a browser. If you visit the page as an anonymous visitor or as a user that is neither a Supervisor nor an Administrator, the CommandField is empty; it still exists, but as a thin sliver without the Edit or Delete buttons.

> **NOTE**
>
> It is possible to hide the CommandField altogether when a non-Supervisor and non-Administrator is visiting the page. I leave this as an exercise for the reader.

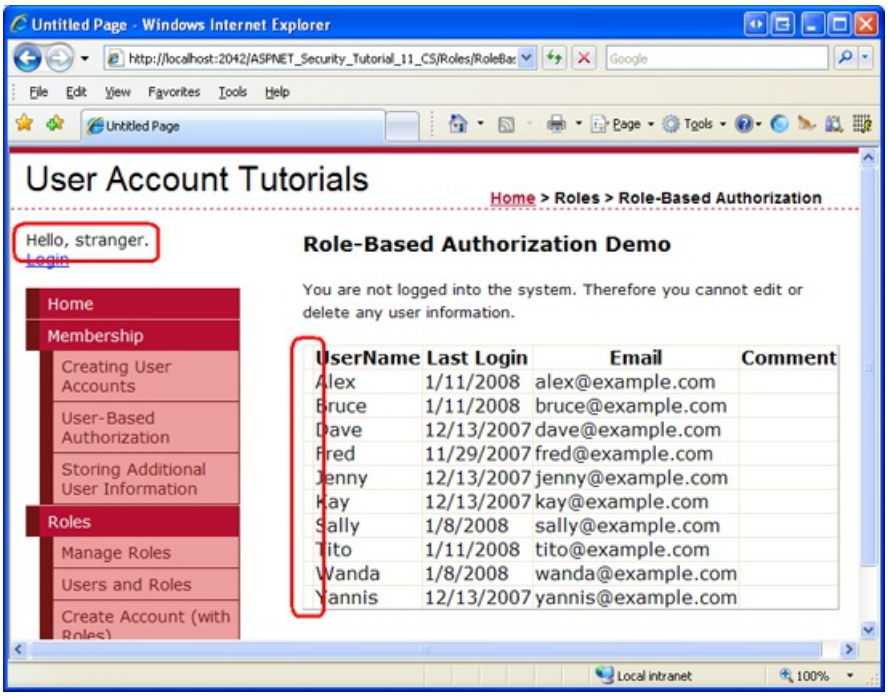


Figure 13: The Edit and Delete Buttons are Hidden for Non-Supervisors and Non-Administrators (Click to view full-size image)

If a user that belongs to the Supervisors role (but not to the Administrators role) visits, he sees only the Edit button.

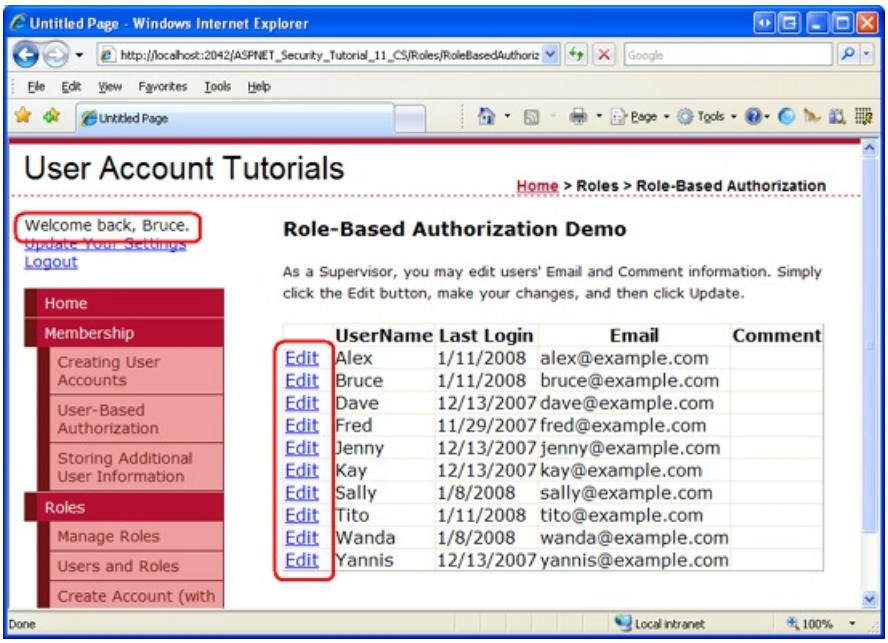


Figure 14: While the Edit Button is Available for Supervisors, the Delete Button is Hidden (Click to view full-size image)

And if an Administrator visits, she has access to both the Edit and Delete buttons.

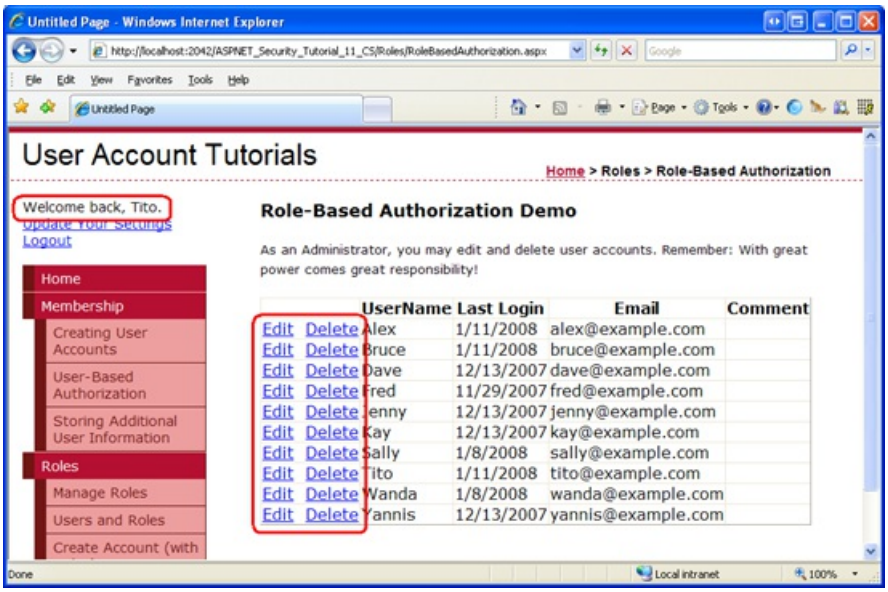


Figure 15: The Edit and Delete Buttons are Available Only for Administrators (Click to view full-size image)

Step 3: Applying Role-Based Authorization Rules to Classes and Methods

In Step 2 we limited edit capabilities to users in the Supervisors and Administrators roles and delete capabilities to Administrators only. This was accomplished by hiding the associated user interface elements for unauthorized users through programmatic techniques. Such measures do not guarantee that an unauthorized user will be unable to perform a privileged action. There may be user interface elements that are added later or that we forgot to hide for unauthorized users. Or a hacker may discover some other way to get the ASP.NET page to execute the desired method.

An easy way to ensure that a particular piece of functionality cannot be accessed by an unauthorized user is to decorate that class or method with the `PrincipalPermission` attribute. When the .NET runtime uses a class or executes one of its methods, it checks to ensure that the current security context has permission. The `PrincipalPermission` attribute provides a mechanism through which we can define these rules.

We looked at using the `PrincipalPermission` attribute back in the *User-Based Authorization* tutorial. Specifically, we saw how to decorate the GridView's `SelectedIndexChanged` and `RowDeleting` event handler so that they could only be executed by authenticated users and Tito, respectively. The `PrincipalPermission` attribute works just as well with roles.

Let's demonstrate using the `PrincipalPermission` attribute on the GridView's `RowUpdating` and `RowDeleting` event handlers to prohibit execution for non-authorized users. All we need to do is add the appropriate attribute atop each function definition:

```
<PrincipalPermission(SecurityAction.Demand, Role:="Administrators")>_
<PrincipalPermission(SecurityAction.Demand, Role:="Supervisors")>_
Protected Sub UserGrid_RowUpdating(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.GridViewUpdateEventArgs) Handles UserGrid.RowUpdating
     ...
End Sub

<PrincipalPermission(SecurityAction.Demand, Role:="Administrators")>_
Protected Sub UserGrid_RowDeleting(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.GridViewDeleteEventArgs) Handles UserGrid.RowDeleting
     ...
End Sub
```

The attribute for the `RowUpdating` event handler dictates that only users in the Administrators or Supervisors

roles can execute the event handler, where as the attribute on the `RowDeleting` event handler limits the execution to users in the Administrators role.

> **NOTE**
> The `PrincipalPermission` attribute is represented as a class in the `System.Security.Permissions` namespace. Be sure to add a `Imports System.Security.Permissions` statement at the top of your code-behind class file to import this namespace.

If, somehow, a non-Administrator attempts to execute the `RowDeleting` event handler or if a non-Supervisor or non-Administrator attempts to execute the `RowUpdating` event handler, the .NET runtime will raise a `SecurityException`.

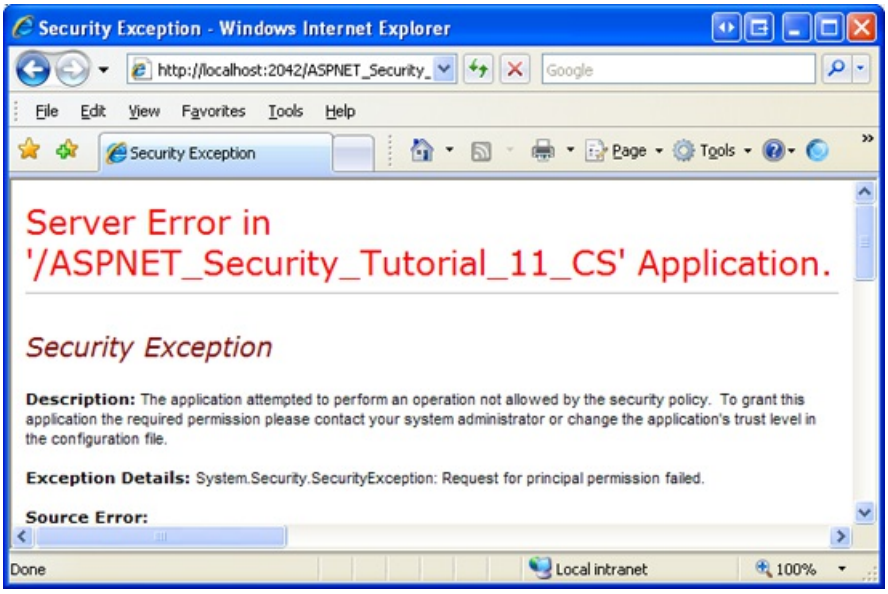


Figure 16: If the Security Context is not Authorized to Execute the Method, a `SecurityException` is Thrown (Click to view full-size image)

In addition to ASP.NET pages, many applications also have an architecture that includes various layers, such as Business Logic and Data Access Layers. These layers are typically implemented as Class Libraries and offer classes and methods for performing business logic- and data-related functionality. The `PrincipalPermission` attribute is useful for applying authorization rules to these layers as well.

For more information on using the `PrincipalPermission` attribute to define authorization rules on classes and methods, refer to Scott Guthrie's blog entry Adding Authorization Rules to Business and Data Layers Using `PrincipalPermissionAttributes`.

Summary

In this tutorial we looked at how to specify coarse and fine grain authorization rules based on the user's roles. ASP.NET's URL authorization feature allows a page developer to specify what identities are allowed or denied access to what pages. As we saw back in the *User-Based Authorization* tutorial, URL authorization rules can be applied on a user-by-user basis. They can also be applied on a role-by-role basis, as we saw in Step 1 of this tutorial.

Fine grain authorization rules may be applied declaratively or programmatically. In Step 2 we looked at using the LoginView control's RoleGroups feature to render different output based on the visiting user's roles. We also looked at ways to programmatically determine if a user belongs to a specific role and how to adjust the page's functionality accordingly.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Adding Authorization Rules to Business and Data Layers Using `PrincipalPermissionAttributes`
- Examining ASP.NET 2.0's Membership, Roles, and Profile: Working with Roles
- Security Question List for ASP.NET 2.0
- Technical documentation for the `<roleManager>` Element

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial include Suchi Banerjee and Teresa Murphy. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS

ASP.NET 3.5 - Security Administration

2/20/2020 • 2 minutes to read • Edit Online

This tutorial series shows you one way to create an administration page that lets you manage users and roles.

- Building an Interface to Select One User Account from Many (C#)
- Recovering and Changing Passwords (C#)
- Unlocking and Approving User Accounts (C#)
- Building an Interface to Select One User Account from Many (VB)
- Recovering and Changing Passwords (VB)
- Unlocking and Approving User Accounts (VB)

Building an Interface to Select One User Account from Many (C#)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

In this tutorial we will build a user interface with a paged, filterable grid. In particular, our user interface will consist of a series of LinkButtons for filtering the results based on the starting letter of the username, and a GridView control to show the matching users. We'll start by listing all of the user accounts in a GridView. Then, in Step 3, we will add the filter LinkButtons. Step 4 looks at paging the filtered results. The interface constructed in Steps 2 through 4 will be used in the subsequent tutorials to perform administrative tasks for a particular user account.

Introduction

In the *Assigning Roles to Users* tutorial, we created a rudimentary interface for an administrator to select a user and manage her roles. Specifically, the interface presented the administrator with a drop-down list of all of the users. Such an interface is suitable when there are but a dozen or so user accounts, but is unwieldy for sites with hundreds or thousands of accounts. A paged, filterable grid is more suitable user interface for websites with large user bases.

In this tutorial we will build such a user interface. In particular, our user interface will consist of a series of LinkButtons for filtering the results based on the starting letter of the username, and a GridView control to show the matching users. We'll start by listing all of the user accounts in a GridView. Then, in Step 3, we will add the filter LinkButtons. Step 4 looks at paging the filtered results. The interface constructed in Steps 2 through 4 will be used in the subsequent tutorials to perform administrative tasks for a particular user account.

Let's get started!

Step 1: Adding New ASP.NET Pages

In this tutorial and the next two we will be examining various administration-related functions and capabilities. We will need a series of ASP.NET pages to implement the topics examined throughout these tutorials. Let's create these pages and update the site map.

Start by creating a new folder in the project named `Administration`. Next, add two new ASP.NET pages to the folder, linking each page with the `Site.master` master page. Name the pages:

- `ManageUsers.aspx`
- `UserInformation.aspx`

Also add two pages to the website's root directory: `ChangePassword.aspx` and `RecoverPassword.aspx`.

These four pages should, at this point, have two Content controls, one for each of the master page's ContentPlaceHolders: `MainContent` and `LoginContent`.

```
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="LoginContent" Runat="Server">
</asp:Content>
```

We want to show the master page's default markup for the `LoginContent` ContentPlaceHolder for these pages. Therefore, remove the declarative markup for the `Content2` Content control. After doing so, the pages' markup should contain just one Content control.

The ASP.NET pages in the `Administration` folder are intended solely for administrative users. We added an Administrators role to the system in the *Creating and Managing Roles* tutorial; restrict access to these two pages to this role. To accomplish this, add a `Web.config` file to the `Administration` folder and configure its `<authorization>` element to admit users in the Administrators role and to deny all others.

```
<?xml version="1.0"?>
<configuration>
 <system.web>
 <authorization>
 <allow roles="Administrators" />
 <deny users="*"/>
 </authorization>
 </system.web>
</configuration>
```

At this point your project's Solution Explorer should look similar to the screen shot shown in Figure 1.

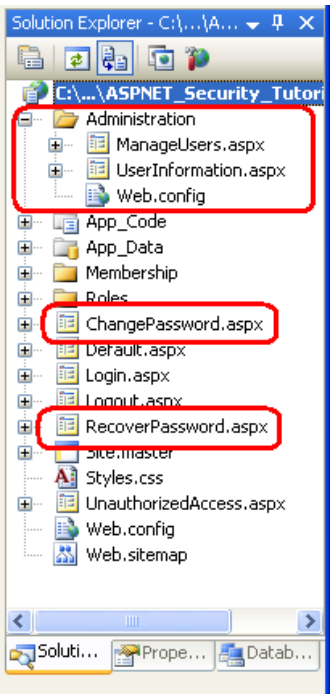


Figure 1: Four New Pages and a `Web.config` File Have Been Added to the Website (Click to view full-size image)

Finally, update the site map (`Web.sitemap`) to include an entry to the `ManageUsers.aspx` page. Add the following XML after the `<siteMapNode>` we added for the Roles tutorials.

```
<siteMapNode title="User Administration" url="~/Administration/ManageUsers.aspx"/>
```

With the site map updated, visit the site through a browser. As Figure 2 shows, the navigation on the left now includes items for the Administration tutorials.

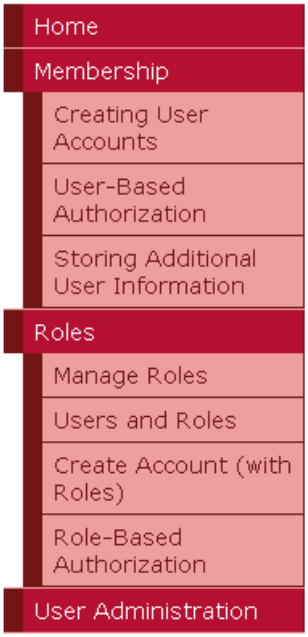


Figure 2: The Site Map Includes a Node Titled User Administration (Click to view full-size image)

Step 2: Listing All User Accounts in a GridView

Our end goal for this tutorial is to create a paged, filterable grid through which an administrator can select a user account to manage. Let's start by listing *all* users in a GridView. Once this is complete, we will add the filtering and paging interfaces and functionality.

Open the `ManageUsers.aspx` page in the `Administration` folder and add a GridView, setting its `ID` to `UserAccounts`. In a moment, we will write code to bind the set of user accounts to the GridView using the `Membership` class's `GetAllUsers` method. As discussed in earlier tutorials, the GetAllUsers method returns a `MembershipUserCollection` object, which is a collection of `MembershipUser` objects. Each `MembershipUser` in the collection includes properties like `UserName`, `Email`, `IsApproved`, and so on.

In order to display the desired user account information in the GridView, set the GridView's `AutoGenerateColumns` property to False and add BoundFields for the `UserName`, `Email`, and `Comment` properties and CheckBoxFields for the `IsApproved`, `IsLockedOut`, and `IsOnline` properties. This configuration can be applied through the control's declarative markup or via the Fields dialog box. Figure 3 shows a screen shot of the Fields dialog box after the Auto-generate fields checkbox has been unchecked and the BoundFields and CheckBoxFields have been added and configured.

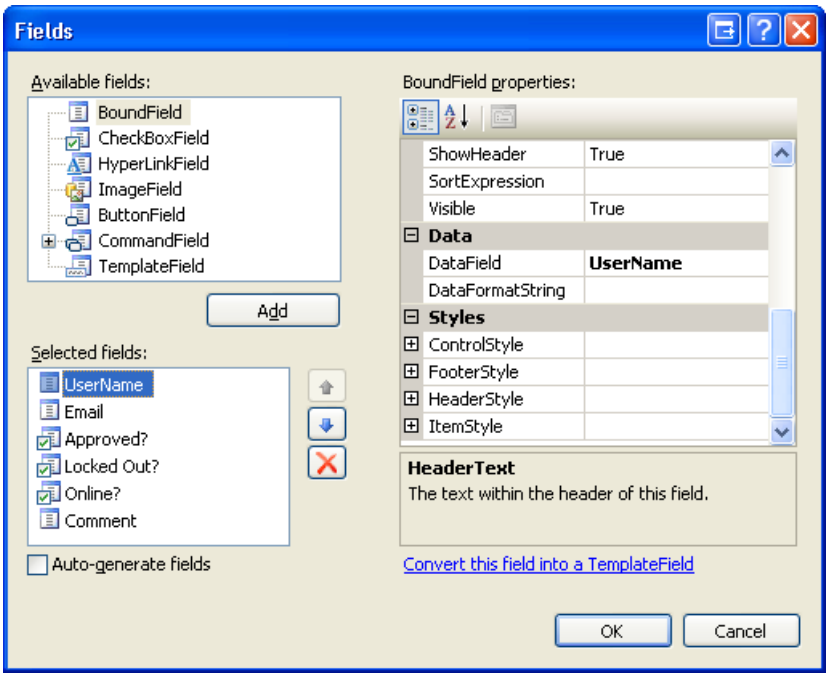


Figure 3: Add Three BoundFields and Three CheckBoxFields to the GridView (Click to view full-size image)

After configuring your GridView, ensure that its declarative markup resembles the following:

```
<asp:GridView ID="UserAccounts" runat="server" AutoGenerateColumns="False">
 <Columns>
 <asp:BoundField DataField="UserName" HeaderText="UserName"/>
 <asp:BoundField DataField="Email" HeaderText="Email" />
 <asp:CheckBoxField DataField="IsApproved" HeaderText="Approved?"/>
 <asp:CheckBoxField DataField="IsLockedOut" HeaderText="Locked Out?" />
 <asp:CheckBoxField DataField="IsOnline" HeaderText="Online?"/>
 <asp:BoundField DataField="Comment" HeaderText="Comment"/>
 </Columns>
</asp:GridView>
```

Next, we need to write code that binds the user accounts to the GridView. Create a method named `BindUserAccounts` to perform this task and then call it from the `Page_Load` event handler on the first page visit.

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    BindUserAccounts();
}

private void BindUserAccounts()
{
    UserAccounts.DataSource = Membership.GetAllUsers();
    UserAccounts.DataBind();
}
```

Take a moment to test the page through a browser. As Figure 4 shows, the `UserAccounts` GridView lists the username, email address, and other pertinent account information for all users in the system.

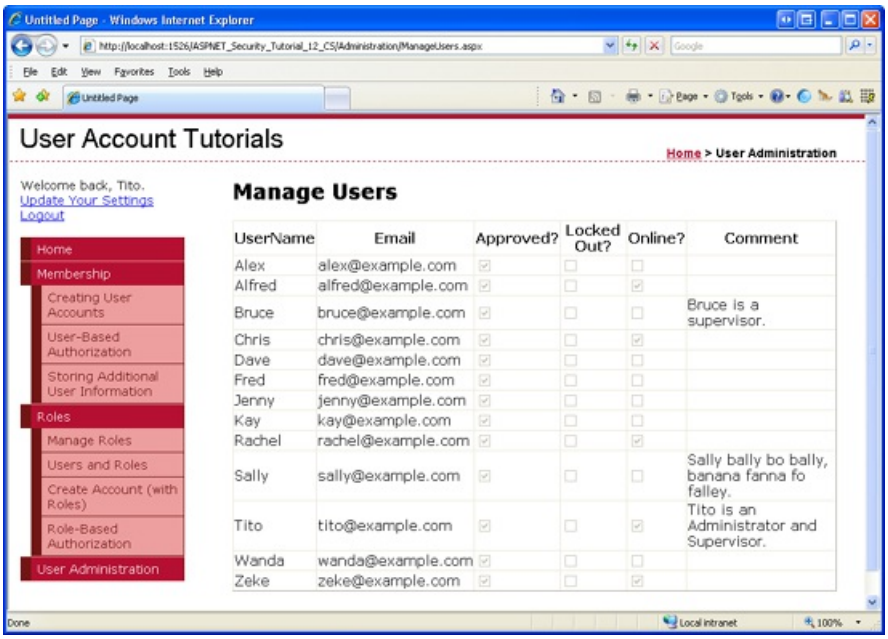


Figure 4: The User Accounts are Listed in the GridView (Click to view full-size image)

Step 3: Filtering the Results by the First Letter of the Username

Currently the `UserAccounts` GridView shows *all* of the user accounts. For websites with hundreds or thousands of user accounts, it is imperative that user be able to quickly pare down the displayed accounts. This can be accomplished by adding filtering LinkButtons to the page. Let's add 27 LinkButtons to the page: one titled All along with one LinkButton for each letter of the alphabet. If a visitor clicks the All LinkButton, the GridView will show all of the users. If they click a particular letter, only those users whose username starts with the selected letter will be displayed.

Our first task is to add the 27 LinkButton controls. One option would be to create the 27 LinkButtons declaratively, one at a time. A more flexible approach is to use a Repeater control with an `ItemTemplate` that renders a LinkButton and then binds the filtering options to the Repeater as a `string` array.

Start by adding a Repeater control to the page above the `UserAccounts` GridView. Set the Repeater's `ID` property to `FilteringUI`. Configure the Repeater's templates so that its `ItemTemplate` renders a LinkButton whose `Text` and `CommandName` properties are bound to the current array element. As we saw in the *Assigning Roles to Users* tutorial, this can be accomplished using the `Container.DataItem` databinding syntax. Use the Repeater's `SeparatorTemplate` to display a vertical line between each link.

```
<asp:Repeater ID="FilteringUI" runat="server">
 <ItemTemplate>
 <asp:LinkButton runat="server" ID="lnkFilter"
 Text='<%# Container.DataItem %>'
 CommandName='<%# Container.DataItem %>'></asp:LinkButton>
 </ItemTemplate>
 <SeparatorTemplate>|</SeparatorTemplate>
</asp:Repeater>
```

To populate this Repeater with the desired filtering options, create a method named `BindFilteringUI`. Be sure to call this method from the `Page_Load` event handler on the first page load.

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        BindUserAccounts();
        BindFilteringUI();
    }
}

private void BindFilteringUI()
{
    string[] filterOptions = { "All", "A", "B", "C","D", "E", "F", "G", "H", "I","J", "K", "L", "M", "N",
"O","P", "Q", "R", "S", "T", "U","V", "W", "X", "Y", "Z" };
    FilteringUI.DataSource = filterOptions;
    FilteringUI.DataBind();
}
```

This method specifies the filtering options as elements in the `string` array `filterOptions`. For each element in the array, the Repeater will render a LinkButton with its `Text` and `CommandName` properties assigned to the value of the array element.

Figure 5 shows the `ManageUsers.aspx` page when viewed through a browser.

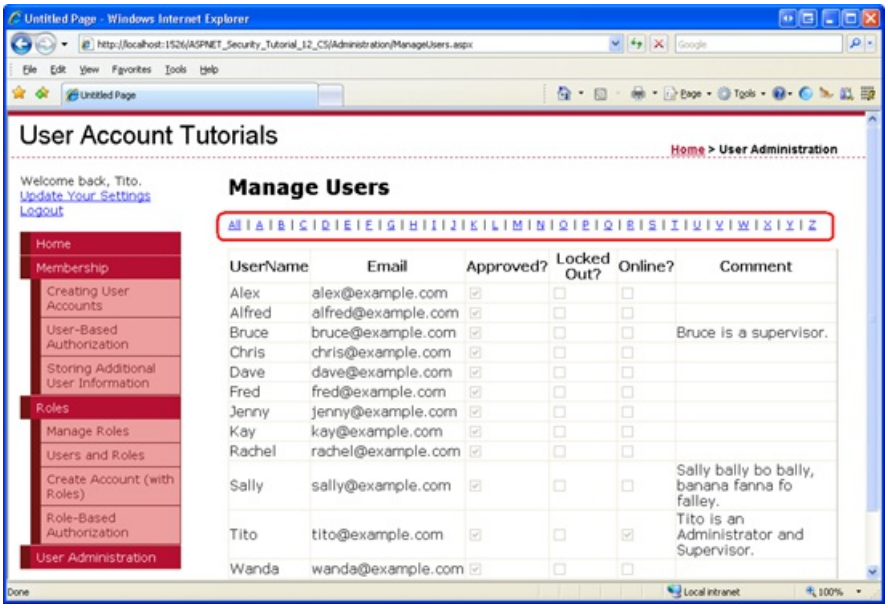


Figure 5: The Repeater Lists 27 Filtering LinkButtons (Click to view full-size image)

> **NOTE**
>
> Usernames may start with any character, including numbers and punctuation. In order to view these accounts, the administrator will have to use the All LinkButton option. Alternatively, you could add a LinkButton to return all user accounts that start with a number. I leave this as an exercise for the reader.

Clicking any of the filtering LinkButtons causes a postback and raises the Repeater's `ItemCommand` event, but there's no change in the grid because we've yet to write any code to filter the results. The `Membership` class includes a `FindUsersByName` method that returns those user accounts whose username matches a specified search pattern. We can use this method to retrieve just those user accounts whose usernames start with the letter specified by the `CommandName` of the filtered LinkButton that was clicked.

Start by updating the `ManageUser.aspx` page's code-behind class so that it includes a property named `UsernameToMatch`. This property persists the username filter string across postbacks:

```
private string UsernameToMatch
{
 get
 {
 object o = ViewState["UsernameToMatch"];
 if (o == null)
 return string.Empty;
 else
 return (string)o;
 }
 set
 {
 ViewState["UsernameToMatch"] = value;
 }
}
```

The `UsernameToMatch` property stores its value it is assigned into the `ViewState` collection using the key UsernameToMatch. When this property's value is read, it checks to see if a value exists in the `ViewState` collection; if not, it returns the default value, an empty string. The `UsernameToMatch` property exhibits a common pattern, namely persisting a value to view state so that any changes to the property are persisted across postbacks. For more information on this pattern, read Understanding ASP.NET View State.

Next, update the `BindUserAccounts` method so that instead of calling `Membership.GetAllUsers`, it calls `Membership.FindUsersByName`, passing in the value of the `UsernameToMatch` property appended with the SQL wildcard character, %.

```
private void BindUserAccounts()
{
    UserAccounts.DataSource = Membership.FindUsersByName(this.UsernameToMatch + "%");
    UserAccounts.DataBind();
}
```

To display just those users whose username starts with the letter A, set the `UsernameToMatch` property to A and then call `BindUserAccounts`. This would result in a call to `Membership.FindUsersByName("A%")`, which will return all users whose username starts with A. Likewise, to return *all* users, assign an empty string to the `UsernameToMatch` property so that the `BindUserAccounts` method will invoke `Membership.FindUsersByName("%")`, thereby returning all user accounts.

Create an event handler for the Repeater's `ItemCommand` event. This event is raised whenever one of the filter LinkButtons is clicked; it is passed the clicked LinkButton's `CommandName` value through the `RepeaterCommandEventArgs` object. We need to assign the appropriate value to the `UsernameToMatch` property and then call the `BindUserAccounts` method. If the `CommandName` is All, assign an empty string to `UsernameToMatch` so that all user accounts are displayed. Otherwise, assign the `CommandName` value to `UsernameToMatch`.

```
protected void FilteringUI_ItemCommand(object source, RepeaterCommandEventArgs e)
{
    if (e.CommandName == "All")
        this.UsernameToMatch = string.Empty;
    else
        this.UsernameToMatch e.CommandName;
    BindUserAccounts();
}
```

With this code in place, test out the filtering functionality. When the page is first visited, all user accounts are displayed (refer back to Figure 5). Clicking the A LinkButton causes a postback and filters the results, displaying only those user accounts that start with A .

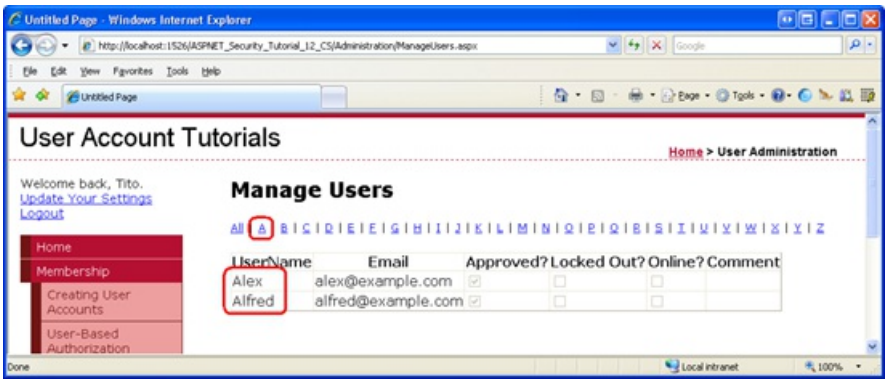


Figure 6: Use the Filtering LinkButtons to Display those Users Whose Username Starts with a Certain Letter (Click to view full-size image)

Step 4: Updating the GridView to Use Paging

The GridView shown in Figures 5 and 6 lists all of the records returned from the `FindUsersByName` method. If there are hundreds or thousands of user accounts this can lead to information overload when viewing all of the accounts (as is the case when clicking the All LinkButton or when initially visiting the page). To help present the user accounts in more manageable chunks, let's configure the GridView to display 10 user accounts at a time.

The GridView control offers two types of paging:

- **Default paging** - easy to implement, but inefficient. In a nutshell, with default paging the GridView expects *all* of the records from its data source. It then only displays the appropriate page of records.
- **Custom paging** - requires more work to implement, but is more efficient than default paging because with custom paging the data source returns only the precise set of records to display.

The performance difference between default and custom paging can be quite substantial when paging through thousands of records. Because we are building this interface assuming that there may be hundreds or thousands of user accounts, let's use custom paging.

> **NOTE**
>
> For a more thorough discussion on the differences between default and custom paging, as well as the challenges involved in implementing custom paging, refer to Efficiently Paging Through Large Amounts of Data. For some analysis of the performance difference between default and custom paging, see Custom Paging in ASP.NET with SQL Server 2005.

To implement custom paging we first need some mechanism by which to retrieve the precise subset of records being displayed by the GridView. The good news is that the `Membership` class's `FindUsersByName` method has an overload that allows us to specify the page index and page size, and returns only those user accounts that fall within that range of records.

In particular, this overload has the following signature: `FindUsersByName(usernameToMatch, pageIndex, pageSize, totalRecords)`.

The *pageIndex* parameter specifies the page of user accounts to return; *pageSize* indicates how many records to display per page. The *totalRecords* parameter is an `out` parameter that returns the number of total user accounts in the user store.

> **NOTE**
>
> The data returned by `FindUsersByName` is sorted by username; the sort criteria cannot be customized.

The GridView can be configured to utilize custom paging, but only when bound to an ObjectDataSource control.

For the ObjectDataSource control to implement custom paging, it requires two methods: one that is passed a start row index and the maximum number of records to display, and returns the precise subset of records that fall within that span; and a method that returns the total number of records being paged through. The `FindUsersByName` overload accepts a page index and page size, and returns the total number of records through an `out` parameter. So there is an interface mismatch here.

One option would be to create a proxy class that exposes the interface the ObjectDataSource expects, and then internally calls the `FindUsersByName` method. Another option - and the one we'll use for this article - is to create our own paging interface and use that instead of the GridView's built-in paging interface.

Creating a First, Previous, Next, Last Paging Interface

Let's build a paging interface with First, Previous, Next, and Last LinkButtons. The First LinkButton, when clicked, will take the user to the first page of data, whereas Previous will return him to the previous page. Likewise, Next and Last will move the user to the next and last page, respectively. Add the four LinkButton controls beneath the `UserAccounts` GridView.

```
<p>
 <asp:LinkButton ID="lnkFirst" runat="server"> First</asp:LinkButton> |
 <asp:LinkButton ID="lnkPrev" runat="server">  Prev</asp:LinkButton>|
 <asp:LinkButton ID="lnkNext" runat="server">Next  </asp:LinkButton>|
 <asp:LinkButton ID="lnkLast" runat="server">Last  </asp:LinkButton>
</p>
```

Next, create an event handler for each of the LinkButton's `Click` events.

Figure 7 shows the four LinkButtons when viewed through the Visual Web Developer Design view.

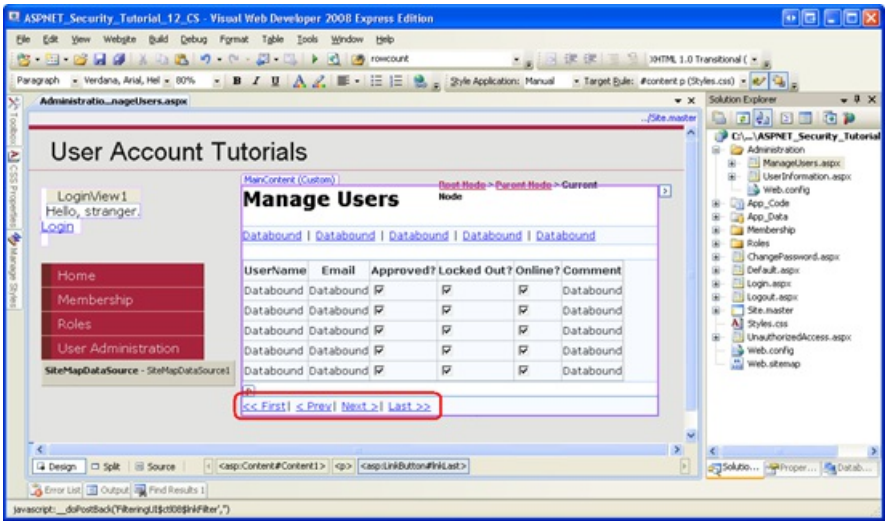


Figure 7: Add First, Previous, Next, and Last LinkButtons Beneath the GridView (Click to view full-size image)

Keeping Track of the Current Page Index

When a user first visits the `ManageUsers.aspx` page or clicks one of the filtering buttons, we want to display the first page of data in the GridView. When the user clicks one of the navigation LinkButtons, however, we need to update the page index. To maintain the page index and the number of records to display per page, add the following two properties to the page's code-behind class:

```
private int PageIndex
{
 get
 {
 object o = ViewState["PageIndex"];
 if (o == null)
 return 0;
 else
 return (int)o;
 }
 set
 {
 ViewState["PageIndex"] = value;
 }
}

private int PageSize
{
 get
 {
 return 10;
 }
}
```

Like the `UsernameToMatch` property, the `PageIndex` property persists its value to view state. The read-only `PageSize` property returns a hard-coded value, 10. I invite the interested reader to update this property to use the same pattern as `PageIndex`, and then to augment the `ManageUsers.aspx` page such that the person visiting the page can specify how many user accounts to display per page.

Retrieving Just the Current Page's Records, Updating the Page Index, and Enabling and Disabling the Paging Interface LinkButtons

With the paging interface in place and the `PageIndex` and `PageSize` properties added, we are ready to update the `BindUserAccounts` method so that it uses the appropriate `FindUsersByName` overload. Additionally, we need to have this method enable or disable the paging interface depending on what page is being displayed. When viewing the first page of data, the First and Previous links should be disabled; Next and Last should be disabled when viewing the last page.

Update the `BindUserAccounts` method with the following code:

```
private void BindUserAccounts()
{
 int totalRecords;
 UserAccounts.DataSource = Membership.FindUsersByName(this.UsernameToMatch + "%",this.PageIndex,
this.PageSize, out totalRecords);
 UserAccounts.DataBind();

 // Enable/disable the paging interface
 bool visitingFirstPage = (this.PageIndex == 0);
 lnkFirst.Enabled = !visitingFirstPage;
 lnkPrev.Enabled = !visitingFirstPage;

 int lastPageIndex = (totalRecords - 1) / this.PageSize;
 bool visitingLastPage = (this.PageIndex >= lastPageIndex);
 lnkNext.Enabled = !visitingLastPage;
 lnkLast.Enabled = !visitingLastPage;
}
```

Note that the total number of records being paged through is determined by the last parameter of the `FindUsersByName` method. This is an `out` parameter, so we need to first declare a variable to hold this value (`totalRecords`) and then prefix it with the `out` keyword.

After the specified page of user accounts are returned, the four LinkButtons are either enabled or disabled, depending on whether the first or last page of data is being viewed.

The last step is to write the code for the four LinkButtons' `Click` event handlers. These event handlers need to update the `PageIndex` property and then rebind the data to the GridView via a call to `BindUserAccounts`. The First, Previous, and Next event handlers are very simple. The `Click` event handler for the Last LinkButton, however, is a bit more complex because we need to determine how many records are being displayed in order to determine the last page index.

```
protected void lnkFirst_Click(object sender, EventArgs e)
{
 this.PageIndex = 0;
 BindUserAccounts();
}

protected void lnkPrev_Click(object sender, EventArgs e)
{
 this.PageIndex -= 1;
 BindUserAccounts();
}

protected void lnkNext_Click(object sender, EventArgs e)
{
 this.PageIndex += 1;
 BindUserAccounts();
}

protected void lnkLast_Click(object sender, EventArgs e)
{
 // Determine the total number of records
 int totalRecords;
 Membership.FindUsersByName(this.UsernameToMatch + "%", this.PageIndex,this.PageSize, out totalRecords);
 // Navigate to the last page index
 this.PageIndex = (totalRecords - 1) / this.PageSize;
 BindUserAccounts();
}
```

Figures 8 and 9 show the custom paging interface in action. Figure 8 shows the `ManageUsers.aspx` page when viewing the first page of data for all user accounts. Note that only 10 of the 13 accounts are displayed. Clicking the Next or Last link causes a postback, updates the `PageIndex` to 1, and binds the second page of user accounts to the grid (see Figure 9).

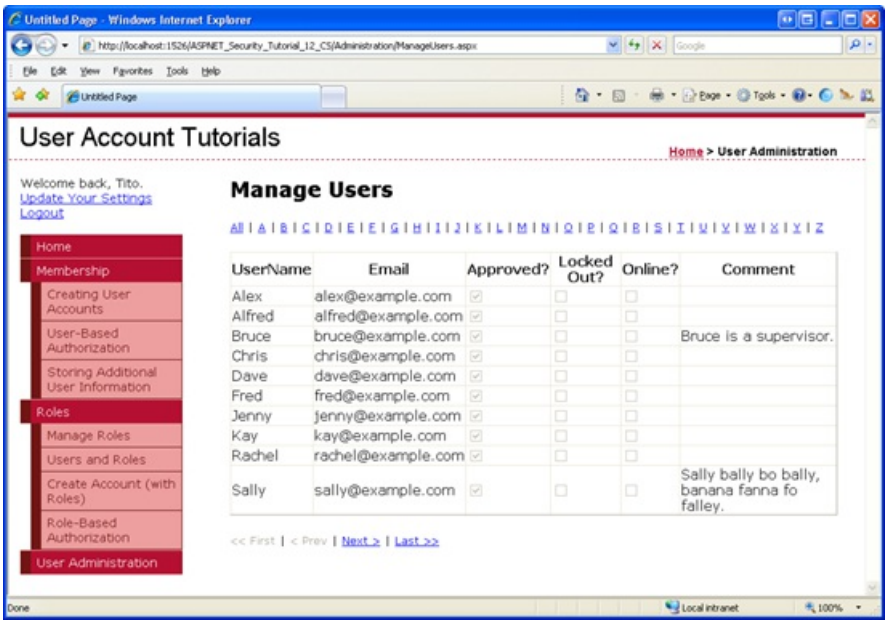


Figure 8: The First 10 User Accounts are Displayed (Click to view full-size image)

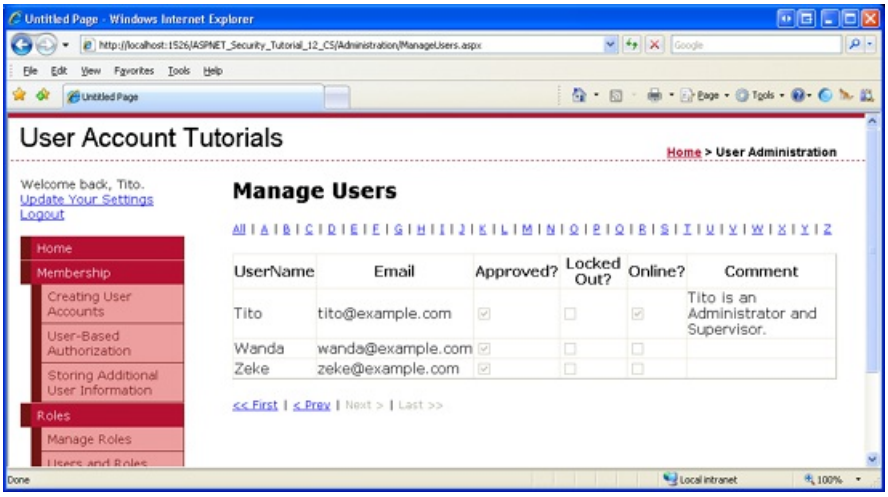


Figure 9: Clicking the Next Link Displays the Second Page of User Accounts (Click to view full-size image)

Summary

Administrators often need to select a user from the list of accounts. In previous tutorials we looked at using a drop-down list populated with the users, but this approach does not scale well. In this tutorial we explored a better alternative: a filterable interface whose results are displayed in a paged GridView. With this user interface, administrators can quickly and efficiently locate and select one user account among thousands.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Custom Paging in ASP.NET with SQL Server 2005
- Efficiently Paging Through Large Amounts of Data
- Rolling Your Own Website Administration Tool

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Alicja Maziarz. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

NEXT

Recovering and Changing Passwords (C#)

2/20/2020 • 20 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

ASP.NET includes two Web controls for assisting with recovering and changing passwords. The PasswordRecovery control enables a visitor to recover his lost password. The ChangePassword control allows the user to update his password. Like the other Login-related Web controls we've seen throughout this tutorial series, the PasswordRecovery and ChangePassword controls work with the Membership framework behind the scenes to reset or modify users' passwords.

Introduction

Between the websites for my bank, utility company, phone company, email accounts, and personalized web portals, I, like most people, have dozens of different passwords to remember. With so many credentials to memorize these days, it's not uncommon for people to forget their password. To account for this, websites that offer user accounts need to include a way for a user to recover his password. This process typically involves generating a new, random password and emailing it to the user's email address on file. After receiving their new password most users return to the site and change their password from the randomly generated one to a more memorable one.

ASP.NET includes two Web controls for assisting with recovering and changing passwords. The PasswordRecovery control enables a visitor to recover his lost password. The ChangePassword control allows the user to update his password. Like the other Login-related Web controls we've seen throughout this tutorial series, the PasswordRecovery and ChangePassword controls work with the Membership framework behind the scenes to reset or modify users' passwords.

In this tutorial we will examine using these two controls. We will also see how to programmatically change and reset a user's password via the `MembershipUser` class's `ChangePassword` and `ResetPassword` methods.

Step 1: Helping Users Recover Lost Passwords

All websites that support user accounts need to provide users with some mechanism for recovering their forgotten passwords. The good news is that implementing such functionality in ASP.NET is a breeze thanks to the PasswordRecovery Web control. The PasswordRecovery control renders an interface that prompts the user for their username and, if needed, the answer to their security question. It then emails the user their password.

> **NOTE**
>
> Because email messages are transmitted over the wire in plain-text there are security risks involved with sending a user's password via email.

The PasswordRecovery control consists of three views:

- **UserName** - prompts the visitor for their username. This is the initial view.
- **Question**- displays the user's username and security question as text, along with a TextBox for the user to enter the answer to his security question.
- **Success**- displays a message informing the user that his password has been emailed.

The views displayed and actions performed by the PasswordRecovery control depend upon the following Membership configuration settings:

- `RequiresQuestionAndAnswer`
- `EnablePasswordRetrieval`
- `EnablePasswordReset`

The Membership framework's `RequiresQuestionAndAnswer` setting indicates whether users must specify a security question and answer when registering for an account. As we discussed in the *Creating User Accounts* tutorial, if `RequiresQuestionAndAnswer` is True (the default) then the CreateUserWizard's interface includes TextBox controls for the new user's security question and answer; if `RequiresQuestionAndAnswer` is False, no such information is collected. Similarly, if `RequiresQuestionAndAnswer` is True, then the PasswordRecovery control displays the Question view after the user has entered their username; the password is recovered only if the user enters the correct security answer. If `RequiresQuestionAndAnswer` is False, however, the PasswordRecovery control moves straight from the UserName view to the Success view.

After the user has provided his username - or his username and security answer, if `RequiresQuestionAndAnswer` is True - the PasswordRecovery emails the user his password. If the `EnablePasswordRetrieval` option is set to True, then the user is emailed their current password. If it is set to False and `EnablePasswordReset` is set to True, then the PasswordRecovery control generates a new, random password for the user, and emails this new password to them. If both `EnablePasswordRetrieval` and `EnablePasswordReset` are False, the PasswordRecovery control throws an exception.

> **NOTE**
> Recall that the `SqlMembershipProvider` stores users' passwords in one of three formats: Clear, Hashed (the default), or Encrypted. The storage mechanism used depends on the Membership configuration settings; the demo application uses the Hashed password format. When using the Hashed password format the `EnablePasswordRetrieval` option must be set to False because the system cannot determine the user's actual password from the hashed version stored in the database.

Figure 1 illustrates how the PasswordRecovery's interface and behavior is influenced by the Membership configuration.

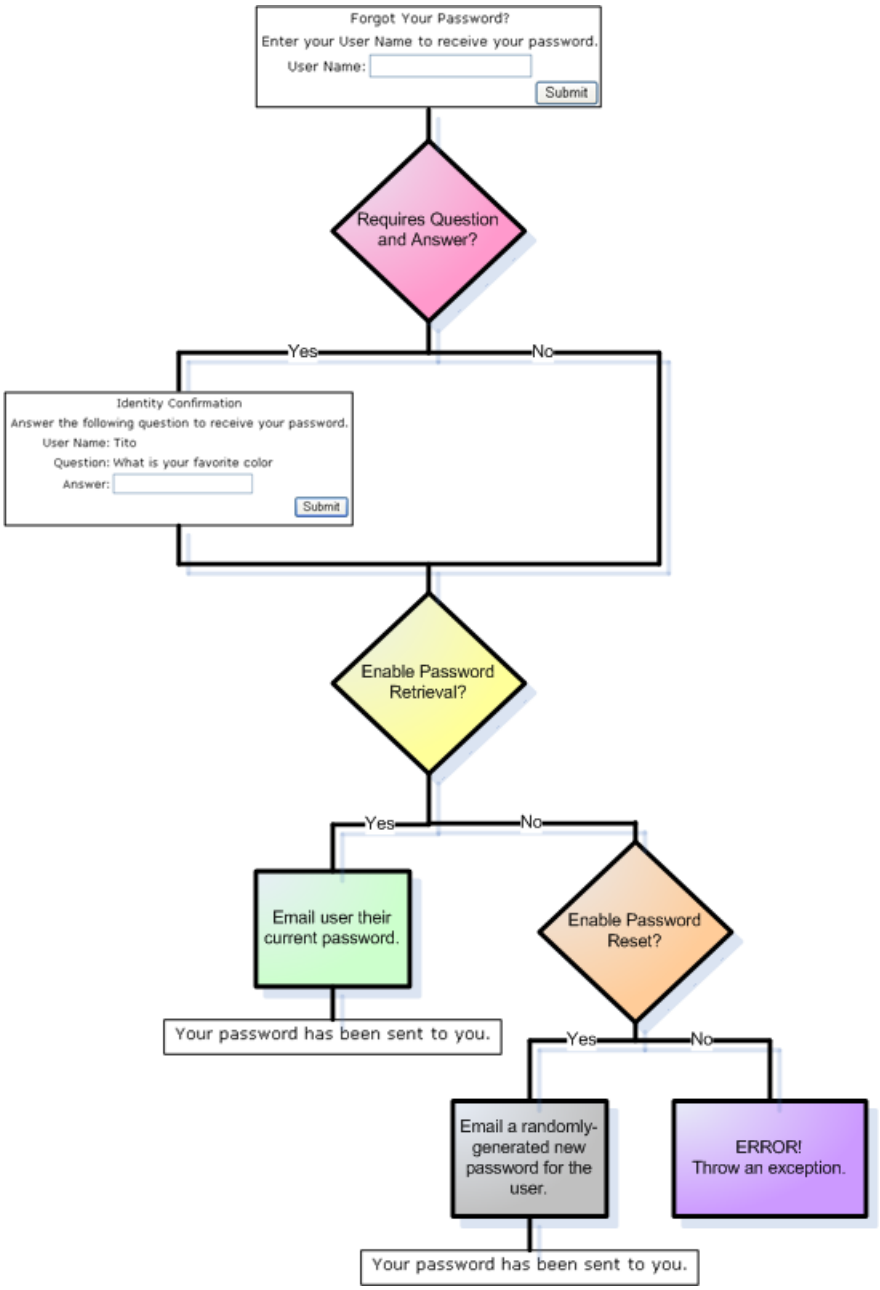


Figure 1: The `RequiresQuestionAndAnswer`, `EnablePasswordRetrieval`, and `EnablePasswordReset` Influence the PasswordRecovery control's Appearance and Behavior (Click to view full-size image)

> **NOTE**
>
> In the *Creating the Membership Schema in SQL Server* tutorial we configured the Membership provider by setting `RequiresQuestionAndAnswer` to True, `EnablePasswordRetrieval` to False, and `EnablePasswordReset` to True.

Using the PasswordRecovery Control

Let's look at using the PasswordRecovery control in an ASP.NET page. Open `RecoverPassword.aspx` and drag and drop a PasswordRecovery control from the Toolbox onto the Designer; set its `ID` to `RecoverPwd`. Like the Login and CreateUserWizard Web controls, the PasswordRecovery control's views render a rich composite interface that includes Labels, TextBoxes, Buttons, and validation controls. You can customize the appearance of the views through the control's style properties or by converting the views into templates. I leave this as an exercise for the interested reader.

When a user visits this page she will enter her username and click the Submit button. Because we have set the `RequiresQuestionAndAnswer` property to True in our Membership configuration settings, the PasswordRecovery control will then display the Question view. After the user enters her correct security answer and clicks Submit,

the PasswordRecovery control will update the user's password to a randomly-generated one, and email this password to the email address on file. All of this was possible without us having to write a single line of code!

Before you test this page, there's one final piece of configuration to tend to: we need to specify the mail delivery settings in `Web.config`. The PasswordRecovery control relies on these settings for sending the email.

The mail delivery configuration is specified through the `<system.net>` element's `<mailSettings>` element. Use the `<smtp>` element to indicate the delivery method and the default From address. The following markup configures mail settings to use a network SMTP server named `smtp.example.com` on port 25 and with username/password credentials of username and password.

> **NOTE**
>
> `<system.net>` is a child element of the root `<configuration>` element and a sibling of `<system.web>`. Therefore, do not put the `<system.net>` element within the `<system.web>` element; instead, put it at the same level.

```
<configuration>
 ...
 <system.net>
 <mailSettings>
 <smtp deliveryMethod="Network" from="youraddress@example.com">
 <network
 host="smtp.example.com"
 userName="username"
 password="password"
 port="25" />
 </smtp>
 </mailSettings>
 </system.net>
</configuration>
```

In addition to using an SMTP server on the network, you can alternatively specify a pickup directory where email messages to be sent should be deposited.

Once you have configured the SMTP settings, visit the `RecoverPassword.aspx` page through a browser. First try entering a username that doesn't exist in the user store. As Figure 2 shows, the PasswordRecovery control displays a message indicating that the user information could not be accessed. The text of the message can be customized through the control's `UserNameFailureText` property.

Figure 2: An Error Message is Displayed if an Invalid Username is Entered (Click to view full-size image)

Now enter a username. Use the username of an account in the system with an email address that you can access and whose security answer you know. After entering the username and clicking Submit, the PasswordRecovery control displays its Question view. As with the UserName view, if you enter an incorrect answer the PasswordRecovery control displays an error message (see Figure 3). Use the `QuestionFailureText` property to customize this error message.

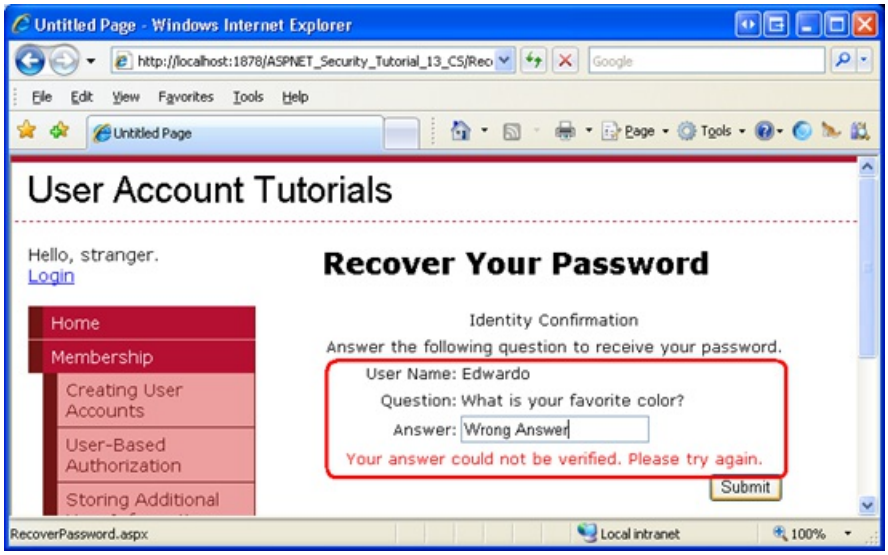


Figure 3: An Error Message is Displayed if the User Enters an Invalid Security Answer (Click to view full-size image)

Finally, enter the correct security answer and click Submit. Behind the scenes, the PasswordRecovery control generates a random password, assigns it to the user account, sends an email informing the user of their new password (see Figure 4), and then displays the Success view.

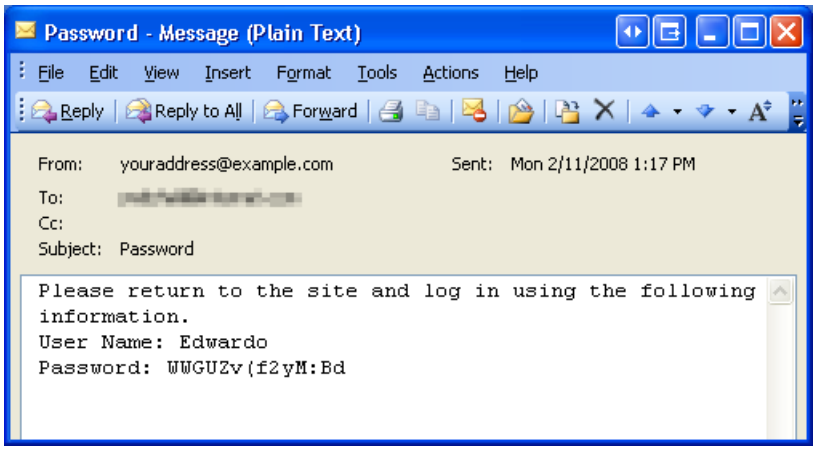


Figure 4: The User is Sent an Email with His New Password (Click to view full-size image)

Customizing the Email

The default email sent by the PasswordRecovery control is rather dull (see Figure 4). The message is sent from the account specified in the `<smtp>` element's `from` attribute with the subject Password and the plain-text body:

Please return to the site and log in using the following information.

User Name: *username*

Password: *password*

This message can be customized programmatically through an event handler for the PasswordRecovery control's `SendingMail` event, or declaratively through the `MailDefinition` property. Let's explore both of these options.

The `SendingMail` event is fired right before the email message is sent and is our last chance to programmatically

adjust the email message. When this event is raised, the event handler is passed an object of type `MailMessageEventArgs`, whose `Message` property contains a reference to the email about to be sent.

Create an event handler for the `SendingMail` event and add the following code, which programmatically adds `webmaster@example.com` to the CC list.

```
protected void RecoverPwd_SendingMail(object sender, MailMessageEventArgs e)
{
    e.Message.CC.Add("webmaster@example.com");
}
```

The email message can also be configured through declarative means. The PasswordRecovery's `MailDefinition` property is an object of type `MailDefinition`. The `MailDefinition` class offers a host of email-related properties, including `From`, `CC`, `Priority`, `Subject`, `IsBodyHtml`, `BodyFileName`, and others. For starters, set the `Subject` property to something more descriptive than the one used by default (Password), such as Your password has been reset...

To customize the body of the email message we need to create a separate email template file that contains the body's contents. Start by creating a new folder in the website named `EmailTemplates`. Next, add a new text file to this folder named `PasswordRecovery.txt` and add the following content:

```
Your password has been reset, <%UserName%>!

According to our records, you have requested that your password be reset. Your new
password is: <%Password%>

If you have any questions or trouble logging on please contact a site administrator.

Thank you!
```

Note the use of the placeholders `<%UserName%>` and `<%Password%>`. The PasswordRecovery control automatically replaces these two placeholders with the user's username and recovered password prior to sending the email.

Lastly, point the `MailDefinition`'s `BodyFileName` property to the email template we just created (`~/EmailTemplates/PasswordRecovery.txt`).

After making these changes revisit the `RecoverPassword.aspx` page and enter your username and security answer. You receive should an email that looks like the one in Figure 5. Note that `webmaster@example.com` has been CC'd and that the subject and body have been updated.

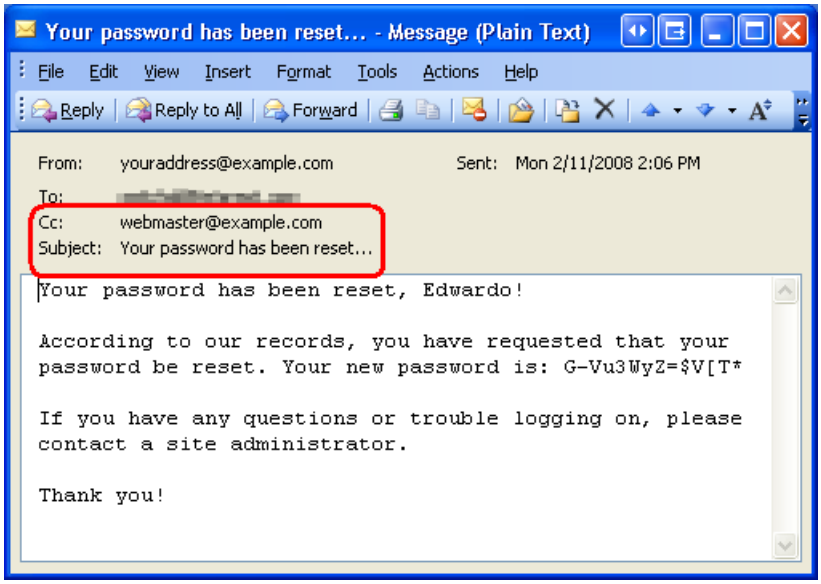


Figure 5: The Subject, Body, and CC List Have Been Updated (Click to view full-size image)

To send an HTML-formatted email set `IsBodyHtml` to True (the default is False) and update the email template to include HTML.

The `MailDefinition` property is not unique to the PasswordRecovery class. As we will see in Step 2, the ChangePassword control also offers a `MailDefinition` property. Moreover, the CreateUserWizard control includes such a property that you can configure to automatically send a welcome email message to new users.

> **NOTE**
> Currently there are no links in the left-hand navigation for reaching the `RecoverPassword.aspx` page. A user would only be interested in visiting this page if she was unable to successfully log on to the site. Therefore, update the `Login.aspx` page to include a link to the `RecoverPassword.aspx` page.

Programmatically Resetting a User's Password

When resetting a user's password the PasswordRecovery control calls the `MembershipUser` object's `ResetPassword` method. This method has two overloads:

- `ResetPassword` - resets a user's password. Use this overload if `RequiresQuestionAndAnswer` is False.
- `ResetPassword(securityAnswer)` - resets a user's password only if the supplied *securityAnswer* is correct. Use this overload if `RequiresQuestionAndAnswer` is True.

Both overloads return the new, randomly-generated password.

Like with the other methods in the Membership framework, the `ResetPassword` method delegates to the configured provider. The `SqlMembershipProvider` invokes the `aspnet_Membership_ResetPassword` stored procedure, passing in the user's username, the new password, and the supplied password answer, among other fields. The stored procedure ensures that the password answer matches and then updates the user's password.

A couple of low-level implementation notes:

- A locked out user cannot reset her password. However, an unapproved user may. We will discuss the locked out and approved states in more detail in the *Unlocking and Approving User* Accounts tutorial.
- If the password answer is incorrect, the user's failed password answer attempt count is incremented. If a specified number of invalid security answer attempts occur within a specified time window, the user is locked out.

A Word on How the Random Passwords are Generated

The randomly-generated passwords shown in the email messages in Figures 4 and 5 are created by the Membership class's `GeneratePassword` method. This method accepts two integer input paramters - *length* and *numberOfNonAlphanumericCharacters* - and returns a string at least *length* characters long with at least *numberOfNonAlphanumericCharacters* number of non-alphanumeric characters. When this method is called from within the Membership classes or Login-related Web controls, the values for these two parameters are determined by the Membership configuration's `MinRequiredPasswordLength` and `MinRequiredNonalphanumericCharacters` properties, which we set to 7 and 1, respectively.

The `GeneratePassword` method uses a cryptographically strong random number generator to ensure that there is no bias in what random characters are selected. Furthermore, `GeneratePassword` is `public`, meaning that you can use it directly from your ASP.NET application if you need to generate random strings or passwords.

> **NOTE**
>
> The `SqlMembershipProvider` class always generates a random password at least 14 characters long, so if `MinRequiredPasswordLength` is less than 14 then its value is ignored.

Step 2: Changing Passwords

The randomly-generated passwords are difficult to remember. Consider the password shown in Figure 4: `WWGUZv(f2yM:Bd`. Try committing that to memory! Needless to say, after a user is sent a randomly-generated password of this sort, she'll want to change the password to something more memorable.

Use the ChangePassword control to create an interface for a user to change her password. Much like the PasswordRecovery control, the ChangePassword control consists of two views: Change Password and Success. The Change Password view prompts the user for their old and new passwords. Upon supplying the correct old password and a new password that meets the minimum length and non-alphanumeric character requirements, the ChangePassword control updates the user's password and displays the Success view.

> **NOTE**
>
> The ChangePassword control modifies the user's password by invoking the `MembershipUser` object's `ChangePassword` method. The ChangePassword method accepts two `string` input parameters - *oldPassword* and *newPassword*- and updates the user's account with the *newPassword*, assuming the supplied *oldPassword* is correct.

Open the `ChangePassword.aspx` page and add a ChangePassword control to the page, naming it `ChangePwd`. At this point, the Design view should show the Change Password view (see Figure 6). Like with the PasswordRecovery control, you can toggle between the views via the control's Smart Tag. Furthermore, these views' appearances are customizable through the assorted style properties or by converting them to a template.

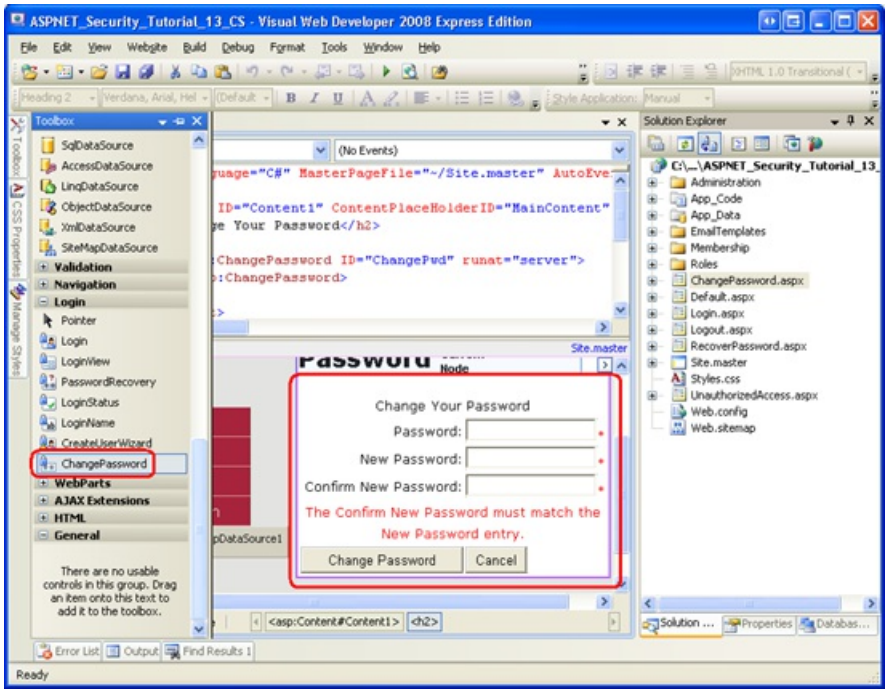


Figure 6: Add a ChangePassword Control to the Page (Click to view full-size image)

The ChangePassword control can update the currently logged in user's password *or* the password of another, specified user. As Figure 6 shows, the default Change Password view renders just three TextBox controls: one for the old password and two for the new password. This default interface is used to update the currently logged on user's password.

To use the ChangePassword control to update another user's password, set the control's `DisplayUserName` property to True. Doing so adds a fourth TextBox to the page, prompting for the username of the user whose password to change.

Setting `DisplayUserName` to True is useful if you want to let a logged out user change her password without having to log in. Personally, I think there's nothing wrong with requiring a user to login before allowing her to change her password. Therefore, leave `DisplayUserName` set to False (its default). In making this decision, however, we essentially are barring anonymous users from reaching this page. Update the site's URL authorization rules so as to deny anonymous users from visiting `ChangePassword.aspx`. If you need to refresh your memory on the URL authorization rule syntax, refer back to the *User-Based Authorization* tutorial.

> **NOTE**
>
> It may seem that the `DisplayUserName` property is useful for allowing administrators to change other users' passwords. However, even when `DisplayUserName` is set to True the correct old password must be known and entered. We will talk about techniques for allowing administrators to change users' passwords in Step 3.

Visit the `ChangePassword.aspx` page through a browser and change your password. Note that an error message is displayed if you enter a new password that fails to meet the password length and non-alphanumeric character requirements specified in the Membership configuration (see Figure 7).

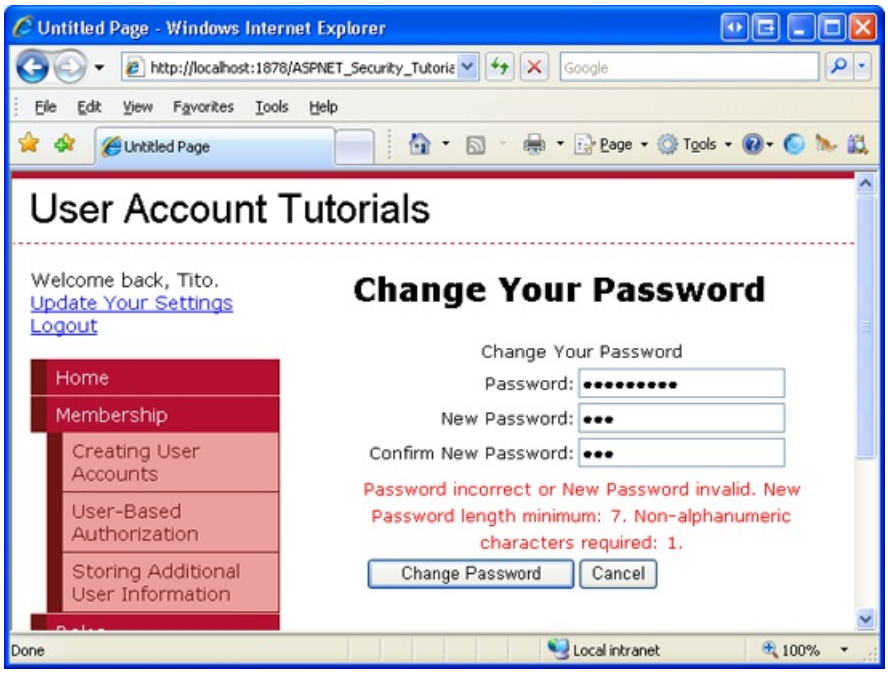


Figure 7: Add a ChangePassword Control to the Page (Click to view full-size image)

Upon entering the correct old password and a valid new password, the logged on user's password is changed and the Success view displayed.

Sending a Confirmation Email

By default, the ChangePassword control does not send an email message to the user whose password was just updated. If you'd like to send an email, simply configure the control's `MailDefinition` property. Let's configure the ChangePassword control so that the user is sent an HTML-formatted email that contains their new password.

Start by creating a new file in the `EmailTemplates` folder named `ChangePassword.htm`. Add the following markup:

```
<html>
 <body>
 <h2>Your Password Has Been Changed!</h2>
 <p>
 This email confirms that your password has been changed.
 </p>
 <p>
 To log on to the site, use the following credentials:
 </p>
 <table>
 <tr>
 <td>
 <b>Username:</b>
 </td>
 <td>
 <%UserName%>
 </td>
 </tr>
 <tr>
 <td>
 <b>Password:</b>
 </td>
 <td>
 <%Password%>
 </td>
 </tr>
 </table>
 <p>
 If you have any questions or encounter any problems logging in,
 please contact a site administrator.
 </p>
 </body>
</html>
```

Next, set the ChangePassword control's `MailDefinition` property's `BodyFileName`, `IsBodyHtml`, and `Subject` properties to ~/EmailTemplates/ChangePassword.htm, True, and Your password has changed!, respectively.

After making these changes, revisit the page and change your password again. This time, the ChangePassword control sends a customized, HTML-formatted email to the user's email address on file (see Figure 8).

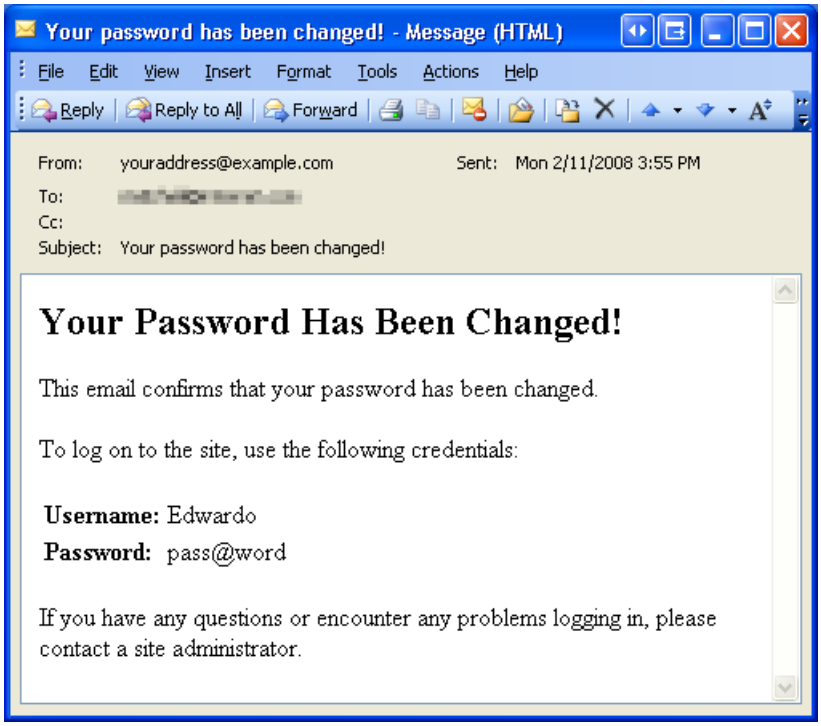


Figure 8: An Email Message Informs the User That Their Password has Changed (Click to view full-size image)

Step 3: Allowing Administrators to Change Users' Passwords

A common feature in applications that support user accounts is the ability for an administrative user to change other users' passwords. Sometimes this functionality is required because the system lacks the ability for users to change their own passwords. In such a case, the only way for a user to recover their forgotten password would be for the administrator to assign them a new password. With the PasswordRecovery and ChangePassword controls, however, administrative users need not busy themselves with changing users' passwords, as users are capable of doing this themselves.

But what if your client insists that administrative users should be able to change other users' passwords? Unfortunately, adding this functionality can be a bit of work. To change a user's password, both the old and new password must be supplied to the `MembershipUser` object's `ChangePassword` method, but an administrator shouldn't have to know a user's password in order to modify it.

One workaround is to first reset the user's password and then change it to the new password using code like the following:

```
MembershipUser usr = Membership.GetUser(username);
string resetPwd = usr.ResetPassword();
usr.ChangePassword(resetPwd, newPassword);
```

This code starts by retrieving information about *username*, which is the user whose password the administrator wishes to change. Next, the `ResetPassword` method is invoked, which assigns and the user a new, random password. This randomly-generated password is returned by the method and stored in the variable `resetPwd`. Now that we know the user's password, we can change it via a call to `ChangePassword`.

The problem is that this code only works if the Membership system configuration is set such that `RequiresQuestionAndAnswer` is False. If `RequiresQuestionAndAnswer` is True, as it is with our application, then the `ResetPassword` method needs to be passed the security answer, otherwise it will throw an exception.

If the Membership framework is configured to require a security question and answer, and yet your client insists that administrators be able to change users' passwords, you have three options:

- Throw your hands in the air and tell your client that this is just one thing that cannot be done.
- Set `RequiresQuestionAndAnswer` to False. This results in a less secure application. Imagine that a nefarious user has gained access to another user's email inbox. Perhaps the compromised user has left their desk to go to lunch and didn't lock their workstation, or maybe they accessed their email from a public terminal and didn't sign out. In either case, the nefarious user can visit the `RecoverPassword.aspx` page and enter the user's username. The system will then email the recovered password without prompting for the security answer.
- Bypass the abstraction layer created by the Membership framework and work directly with the SQL Server database. The Membership schema includes a stored procedure named `aspnet_Membership_SetPassword` that sets a user's password and does not require the security answer or old password in order to accomplish its task.

None of these options are particularly appealing, but that's how the life of a developer goes sometimes.

I went ahead and implemented the third approach, writing code that bypasses the `Membership` and `MembershipUser` classes and operates directly against the `SecurityTutorials` database.

NOTE

By working directly with the database, the encapsulation provided by the Membership framework is shattered. This decision ties us to the `SqlMembershipProvider`, making our code less portable. Furthermore, this code may not work as expected in future versions of ASP.NET if the Membership schema changes. This approach is a workaround and, like most workarounds, is not an example of best practices.

The code has some unattractive bits and is quite lengthy. Therefore, I don't want to clutter this tutorial with an in-depth examination of it. If you are interested in learning more, download the code for this tutorial and visit the `~/Administration/ManageUsers.aspx` page. This page, which we created in the preceding tutorial, lists each user. I've updated the GridView to include a link to the `UserInformation.aspx` page, passing the selected user's username through the querystring. The `UserInformation.aspx` page displays information about the selected user and TextBoxes for changing their password (see Figure 9).

After entering the new password, confirming it in the second TextBox, and clicking the Update User Button, a postback ensues and the `aspnet_Membership_SetPassword` stored procedure is invoked, updating the user's password. I encourage those readers interested in this functionality to become more familiar with the code and try extending the functionality to include sending an email to the user whose password was changed.

Figure 9: An Administrator May Change a User's Password (Click to view full-size image)

NOTE

The `UserInformation.aspx` page currently only works if the Membership framework is configured to store passwords in Clear or Hashed format. It lacks the code to encrypt the new password, although you are invited to add this functionality. The way I recommend adding the necessary code is to use a decompiler like Reflector to examine the source code for methods in the .NET Framework; start by examining the `SqlMembershipProvider` class's `ChangePassword` method. This is the technique I used to write the code for creating a hash of the password.

Summary

ASP.NET offers two controls to help users manage their password. The PasswordRecovery control is useful for those who have forgotten their passwords. Depending on the Membership framework's configuration, the user is either emailed their existing password or a new, randomly-generated password. The ChangePassword control enables a user to update their password.

Like the Login and CreateUserWizard controls, the PasswordRecovery and ChangePassword controls render a rich

user interface without having to write a lick of declarative markup or line of code. If the default user interface doesn't meet your needs, you can customize it through a variety of style properties. Alternatively, the controls' interfaces may be converted into templates, for an even finer degree of control. Behind the scenes these controls use the Membership API, invoking the `MembershipUser` object's `ResetPassword` and `ChangePassword` methods.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ChangePassword Control QuickStarts
- PasswordRecovery Control QuickStarts
- Sending Email in ASP.NET
- `System.Net.Mail` FAQs

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial include Michael Emmings and Suchi Banerjee. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Unlocking and Approving User Accounts (C#)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> This tutorial shows how to build a web page for administrators to manage users' locked out and approved statuses. We will also see how to approve new users only after they have verified their email address.

Introduction

Along with a username, password, and email, each user account has two status fields that dictate whether the user can log into the site: locked out and approved. A user is automatically locked out if they provide invalid credentials a specified number of times within a specified number of minutes (the default settings lock a user out after 5 invalid login attempts within 10 minutes). The approved status is useful in scenarios where some action must transpire before a new user is able to log on to the site. For example, a user might need to first verify their email address or be approved by an administrator before being able to login.

Because a locked out or unapproved user cannot login, it's only natural to wonder how these statuses can be reset. ASP.NET does not include any built-in functionality or Web controls for managing users' locked out and approved statuses, in part because these decisions need to be handled on a site-by-site basis. Some sites might automatically approve all new user accounts (the default behavior). Others have an administrator approve new accounts or do not approve users until they visit a link sent to the email address provided when they signed up. Likewise, some sites may lock out users until an administrator resets their status, while other sites send an email to the locked out user with a URL they can visit to unlock their account.

This tutorial shows how to build a web page for administrators to manage users' locked out and approved statuses. We will also see how to approve new users only after they have verified their email address.

Step 1: Managing Users' Locked Out and Approved Statuses

In the *Building an Interface to Select One User Account from Many* tutorial we constructed a page that listed each user account in a paged, filtered GridView. The grid lists each user's name and email, their approved and locked out statuses, whether they're currently online, and any comments about the user. To manage users' approved and locked out statuses, we could make this grid editable. To change a user's approved status, the administrator would first locate the user account and then edit the corresponding GridView row, checking or unchecking the approved checkbox. Alternatively, we could manage the approved and locked out statuses through a separate ASP.NET page.

For this tutorial let's use two ASP.NET pages: `ManageUsers.aspx` and `UserInformation.aspx`. The idea here is that `ManageUsers.aspx` lists the user accounts in the system, while `UserInformation.aspx` enables the administrator to manage the approved and locked out statuses for a specific user. Our first order of business is to augment the GridView in `ManageUsers.aspx` to include a HyperLinkField, which renders as a column of links. We want each link to point to `UserInformation.aspx?user=UserName`, where *UserName* is the name of the user to edit.

> **NOTE**
>
> If you downloaded the code for the *Recovering and Changing Passwords* tutorial you may have noticed that the `ManageUsers.aspx` page already contains a set of "Manage" links and the `UserInformation.aspx` page provides an interface for changing the selected user's password. I decided not to replicate that functionality in the code associated with this tutorial because it worked by circumventing the Membership API and operating directly with the SQL Server database to change a user's password. This tutorial starts from scratch with the `UserInformation.aspx` page.

Adding "Manage" Links to the `UserAccounts` GridView

Open the `ManageUsers.aspx` page and add a HyperLinkField to the `UserAccounts` GridView. Set the HyperLinkField's `Text` property to "Manage" and its `DataNavigateUrlFields` and `DataNavigateUrlFormatString` properties to `UserName` and "UserInformation.aspx?user={0}", respectively. These settings configure the HyperLinkField such that all of the hyperlinks display the text "Manage", but each link passes in the appropriate *UserName* value into the querystring.

After adding the HyperLinkField to the GridView, take a moment to view the `ManageUsers.aspx` page through a browser. As Figure 1 shows, each GridView row now includes a "Manage" link. The "Manage" link for Bruce points to `UserInformation.aspx?user=Bruce`, whereas the "Manage" link for Dave points to `UserInformation.aspx?user=Dave`.

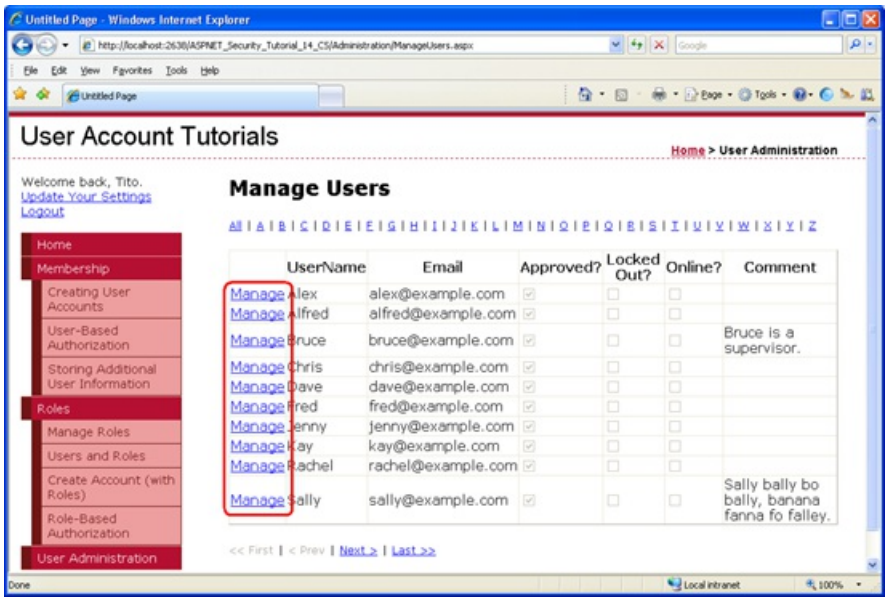


Figure 1: The HyperLinkField Adds a "Manage" Link for Each User Account (Click to view full-size image)

We will create the user interface and code for the `UserInformation.aspx` page in a moment, but first let's talk about how to programmatically change a user's locked out and approved statuses. The `MembershipUser` class has `IsLockedOut` and `IsApproved` properties. The `IsLockedOut` property is read-only. There is no mechanism to programmatically lock out a user; to unlock a user, use the `MembershipUser` class's `UnlockUser` method. The `IsApproved` property is readable and writeable. To save any changes to this property, we need to call the `Membership` class's `UpdateUser` method, passing in the modified `MembershipUser` object.

Because the `IsApproved` property is readable and writeable, a CheckBox control is probably the best user interface element for configuring this property. However, a CheckBox will not work for the `IsLockedOut` property because an administrator cannot lock out a user, she may only unlock a user. A suitable user interface for the `IsLockedOut` property is a Button that, when clicked, unlocks the user account. This Button should only be enabled if the user is locked out.

Creating the `UserInformation.aspx` Page

We are now ready to implement the user interface in `UserInformation.aspx`. Open this page and add the

following Web controls:

- A HyperLink control that, when clicked, returns the administrator to the `ManageUsers.aspx` page.
- A Label Web control for displaying the selected user's name. Set this Label's `ID` to `UserNameLabel` and clear out its `Text` property.
- A CheckBox control named `IsApproved`. Set its `AutoPostBack` property to `true`.
- A Label control for displaying the user's last locked out date. Name this Label `LastLockedOutDateLabel` and clear out its `Text` property.
- A Button for unlocking the user. Name this Button `UnlockUserButton` and set its `Text` property to "Unlock User".
- A Label control for displaying status messages, such as, "The user's approved status has been updated." Name this control `StatusMessage`, clear out its `Text` property, and set its `CssClass` property to `Important`. (The `Important` CSS class is defined in the `Styles.css` stylesheet file; it displays the corresponding text in a large, red font.)

After adding these controls, the Design view in Visual Studio should look similar to the screen shot in Figure 2.

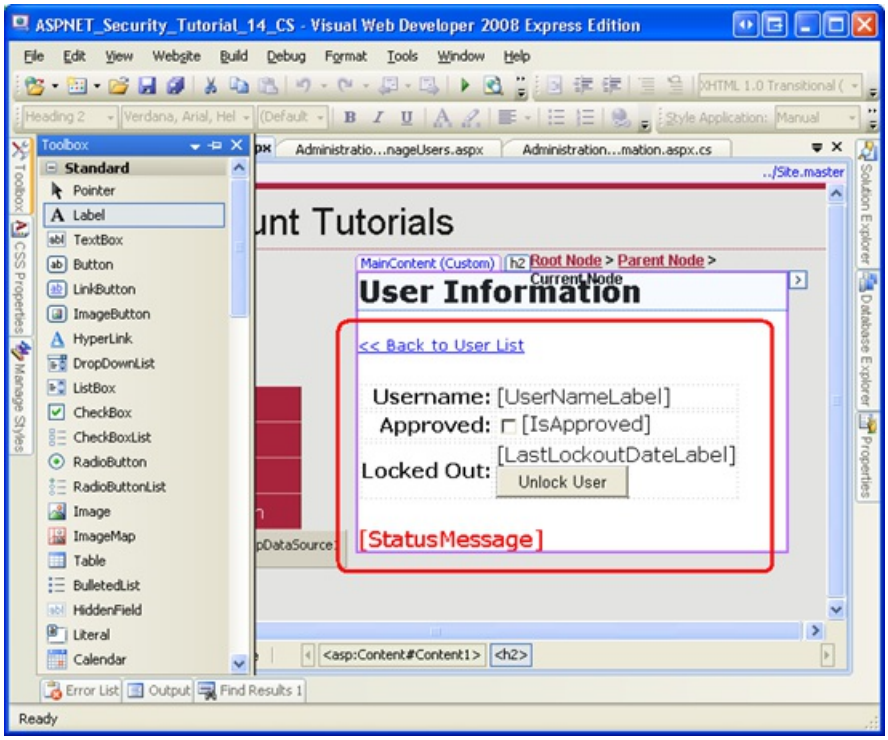


Figure 2: Create the User Interface for `UserInformation.aspx` (Click to view full-size image)

With the user interface complete, our next task is to set the `IsApproved` CheckBox and other controls based on the selected user's information. Create an event handler for the page's `Load` event and add the following code:

```
protected void Page_Load(object sender, EventArgs e)
{
     if (!Page.IsPostBack)
     {

          // If querystring value is missing, send the user to ManageUsers.aspx
          string userName = Request.QueryString["user"];
          if (string.IsNullOrEmpty(userName))
               Response.Redirect("ManageUsers.aspx");

          // Get information about this user
          MembershipUser usr = Membership.GetUser(userName);
          if (usr == null)
               Response.Redirect("ManageUsers.aspx");

          UserNameLabel.Text = usr.UserName;
          IsApproved.Checked = usr.IsApproved;
          if (usr.LastLockoutDate.Year < 2000)

               LastLockoutDateLabel.Text = string.Empty;
          else
               LastLockoutDateLabel.Text = usr.LastLockoutDate.ToShortDateString();

          UnlockUserButton.Enabled = usr.IsLockedOut;
     }
}
```

The above code starts by ensuring that this is the first visit to the page and not a subsequent postback. It then reads the username passed through the `user` querystring field and retrieves information about that user account via the `Membership.GetUser(username)` method. If no username was supplied through the querystring, or if the specified user could not be found, the administrator is sent back to the `ManageUsers.aspx` page.

The `MembershipUser` object's `UserName` value is then displayed in the `UserNameLabel` and the `IsApproved` CheckBox is checked based on the `IsApproved` property value.

The `MembershipUser` object's `LastLockoutDate` property returns a `DateTime` value indicating when the user was last locked out. If the user has never been locked out, the value returned depends on the Membership provider. When a new account is created, the `SqlMembershipProvider` sets the `aspnet_Membership` table's `LastLockoutDate` field to `1754-01-01 12:00:00 AM`. The above code displays an empty string in the `LastLockoutDateLabel` if the `LastLockoutDate` property occurs before year 2000; otherwise, the date portion of the `LastLockoutDate` property is displayed in the Label. The `UnlockUserButton'` s `Enabled` property is set to the user's locked out status, meaning that this Button will only be enabled if the user is locked out.

Take a moment to test the `UserInformation.aspx` page through a browser. You will, of course, need to start at `ManageUsers.aspx` and select a user account to manage. Upon arriving at `UserInformation.aspx`, note that the `IsApproved` CheckBox is only checked if the user is approved. If the user has ever been locked out, their last locked out date is displayed. The Unlock User button is enabled only if the user is currently locked out. Checking or unchecking the `IsApproved` CheckBox or clicking the Unlock User button causes a postback, but no modifications are made to the user account because we've yet to create event handlers for these events.

Return to Visual Studio and create event handlers for the `IsApproved` CheckBox's `CheckedChanged` event and the `UnlockUser` Button's `Click` event. In the `CheckedChanged` event handler, set the user's `IsApproved` property to the `Checked` property of the CheckBox and then save the changes via a call to `Membership.UpdateUser`. In the `Click` event handler, simply call the `MembershipUser` object's `UnlockUser` method. In both event handlers, display a suitable message in the `StatusMessage` Label.

```
protected void IsApproved_CheckedChanged(object sender, EventArgs e)
{
     // Toggle the user's approved status
     string userName = Request.QueryString["user"];
     MembershipUser usr = Membership.GetUser(userName);
     usr.IsApproved = IsApproved.Checked;
     Membership.UpdateUser(usr);
     StatusMessage.Text = "The user's approved status has been updated.";
}

protected void UnlockUserButton_Click(object sender, EventArgs e)
{
     // Unlock the user account
     string userName = Request.QueryString["user"];
     MembershipUser usr = Membership.GetUser(userName);

     usr.UnlockUser();
     UnlockUserButton.Enabled = false;
     StatusMessage.Text = "The user account has been unlocked.";
}
```

Testing the `UserInformation.aspx` Page

With these event handlers in place, revisit the page and unapproved a user. As Figure 3 shows, you should see a brief message on the page indicating that the user's `IsApproved` property was successfully modified.

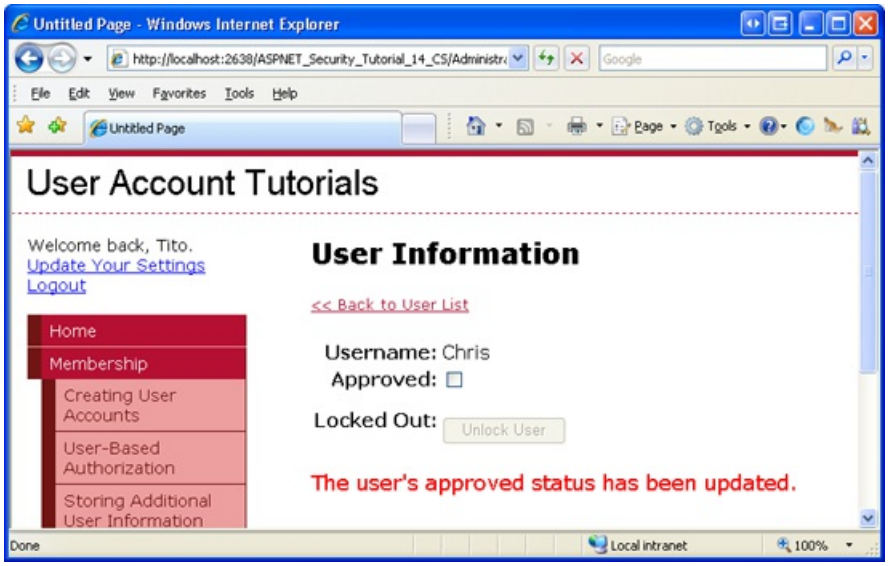


Figure 3: Chris has been Unapproved (Click to view full-size image)

Next, logout and try to login as the user whose account was just unapproved. Because the user is not approved, they cannot login. By default, the Login control displays the same message if the user cannot login, regardless of the reason. But in the *Validating User Credentials Against the Membership User Store* tutorial we looked at enhancing the Login control to display a more appropriate message. As Figure 4 shows, Chris is shown a message explaining that he cannot login because his account is not yet approved.

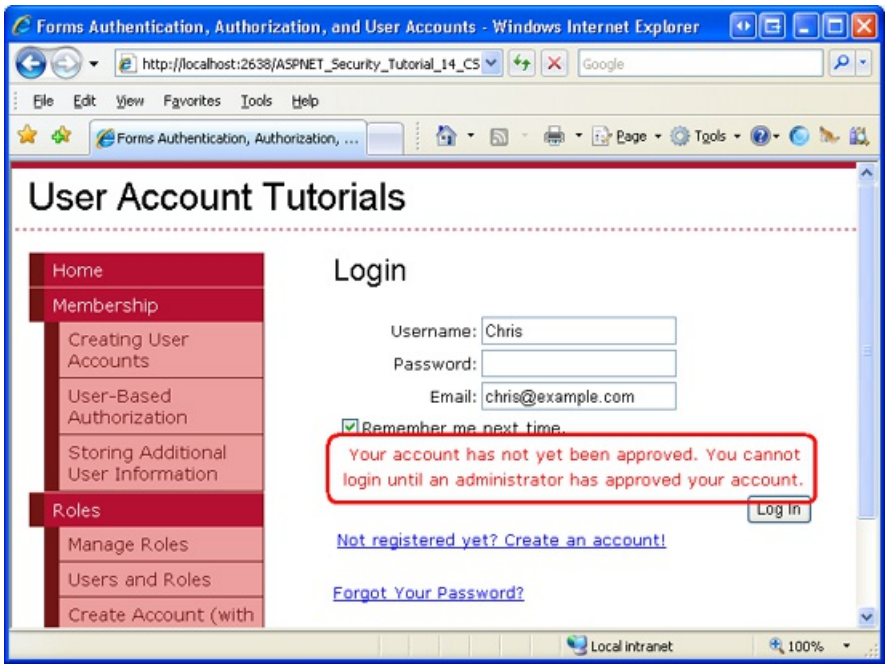


Figure 4: Chris Cannot Login Because His Account is Unapproved (Click to view full-size image)

To test the locked out functionality, attempt to login as an approved user, but use an incorrect password. Repeat this process the necessary number of times until the user's account has been locked out. The Login control was also updated to show a custom message if attempting to login from a locked out account. You know that an account has been locked out once you start seeing the following message at the login page: "Your account has been locked out because of too many invalid login attempts. Please contact the administrator to have your account unlocked."

Return to the `ManageUsers.aspx` page and click the Manage link for the locked out user. As Figure 5 shows, you should see a value in the `LastLockedOutDateLabel` the Unlock User button should be enabled. Click the Unlock User button to unlock the user account. Once you have unlocked the user, they will be able to login again.

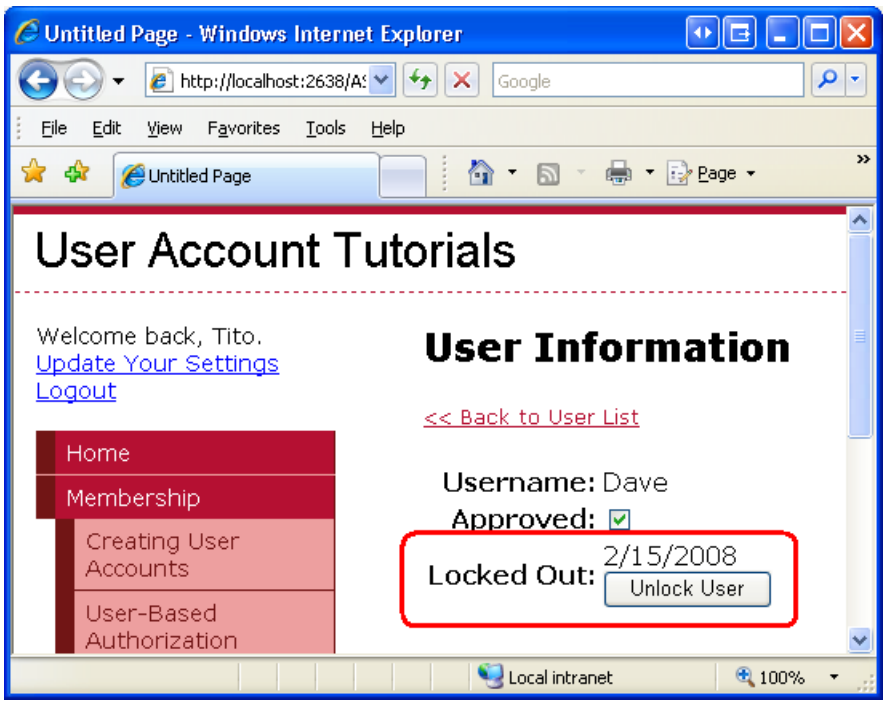


Figure 5: Dave Has Been Locked Out of the System (Click to view full-size image)

Step 2: Specifying New Users' Approved Status

The approved status is useful in scenarios where you want some action to be performed before a new user is able

to login and access the user-specific features of the site. For example, you may be running a private website where all pages, except for the login and signup pages, are accessible only to authenticated users. But what happens if a stranger reaches your website, finds the signup page, and creates an account? To prevent this from happening you could move the signup page to an `Administration` folder, and require that an administrator manually create each account. Alternatively, you could allow anyone to signup, but prohibit site access until an administrator approves the user account.

By default, the CreateUserWizard control approves new accounts. You can configure this behavior using the control's `DisableCreatedUser` property. Set this property to `true` to not approve new user accounts.

> **NOTE**
> By default the CreateUserWizard control automatically logs on the new user account. This behavior is dictated by the control's `LoginCreatedUser` property. Because unapproved users cannot login to the site, when `DisableCreatedUser` is `true` the new user account is not logged into the site, regardless of the value of the `LoginCreatedUser` property.

If you are programmatically creating new user accounts via the `Membership.CreateUser` method, to create an unapproved user account use one of the overloads that accept the new user's `IsApproved` property value as an input parameter.

Step 3: Approving Users By Verifying their Email Address

Many websites that support user accounts do not approve new users until they verify the email address they supplied when registering. This verification process is commonly used to thwart bots, spammers, and other ne'er-do-wells as it requires a unique, verified email address and adds an extra step in the signup process. With this model, when a new user signs up they are sent an email message that includes a link to a verification page. By visiting the link the user has proven that they received the email and, therefore, that the email address provided is valid. The verification page is responsible for approving the user. This may happen automatically, thereby approving any user who reaches this page, or only after the user provides some additional information, such as a CAPTCHA.

To accommodate this workflow, we need to first update the account creation page so that new users are unapproved. Open the `EnhancedCreateUserWizard.aspx` page in the `Membership` folder and set the CreateUserWizard control's `DisableCreatedUser` property to `true`.

Next, we need to configure the CreateUserWizard control to send an email to the new user with instructions on how to verify their account. In particular, we will include a link in the email to the `Verification.aspx` page (which we've yet to create), passing in the new user's `UserId` through the querystring. The `Verification.aspx` page will lookup the specified user and mark them approved.

Sending a Verification Email to New Users

To send an email from the CreateUserWizard control, configure its `MailDefinition` property appropriately. As discussed in the previous tutorial, the ChangePassword and PasswordRecovery controls include a `MailDefinition` property that works in the same manner as the CreateUserWizard control's.

> **NOTE**
> To use the `MailDefinition` property you need to specify mail delivery options in `Web.config`. For more information, refer to Sending Email in ASP.NET.

Start by creating a new email template named `CreateUserWizard.txt` in the `EmailTemplates` folder. Use the following text for the template:

```
Hello <%UserName%>! Welcome aboard.

Your new account is almost ready, but before you can login you must first visit:
<%VerificationUrl%>

Once you have visited the verification URL you will be redirected to the login page.

If you have any problems or questions, please reply to this email.

Thanks!
```

Set the `MailDefinition'` s `BodyFileName` property to "~/EmailTemplates/CreateUserWizard.txt" and its `Subject` property to "Welcome to My Website! Please activate your account."

Note that the `CreateUserWizard.txt` email template includes a `<%VerificationUrl%>` placeholder. This is where the URL for the `Verification.aspx` page will be placed. The CreateUserWizard automatically replaces the `<%UserName%>` and `<%Password%>` placeholders with the new account's username and password, but there is no built-in `<%VerificationUrl%>` placeholder. We need to manually replace it with the appropriate verification URL.

To accomplish this, create an event handler for the CreateUserWizard's `SendingMail` event and add the following code:

```
protected void NewUserWizard_SendingMail(object sender, MailMessageEventArgs e)
{
     // Get the UserId of the just-added user
     MembershipUser newUser = Membership.GetUser(NewUserWizard.UserName);

     Guid newUserId = (Guid)newUser.ProviderUserKey;

     // Determine the full verification URL (i.e., http://yoursite.com/Verification.aspx?ID=...)
     string urlBase = Request.Url.GetLeftPart(UriPartial.Authority) +
          Request.ApplicationPath;

     string verifyUrl = "/Verification.aspx?ID=" + newUserId.ToString();
     string fullUrl = urlBase + verifyUrl;

     // Replace <%VerificationUrl%> with the appropriate URL and querystring
     e.Message.Body = e.Message.Body.Replace("<%VerificationUrl%>", fullUrl);
}
```

The `SendingMail` event fires after the `CreatedUser` event, meaning that by the time the above event handler executes the new user account has already been created. We can access the new user's `UserId` value by calling the `Membership.GetUser` method, passing in the `UserName` entered into the CreateUserWizard control. Next, the verification URL is formed. The statement `Request.Url.GetLeftPart(UriPartial.Authority)` returns the `http://yourserver.com` portion of the URL; `Request.ApplicationPath` returns path where the application is rooted. The verification URL is then defined as `Verification.aspx?ID=userId`. These two strings are then concatenated to form the complete URL. Finally, the email message body (`e.Message.Body`) has all occurrences of `<%VerificationUrl%>` replaced with the full URL.

The net effect is that new users are unapproved, meaning that they cannot log into the site. Furthermore, they are automatically sent an email with a link to the verification URL (see Figure 6).

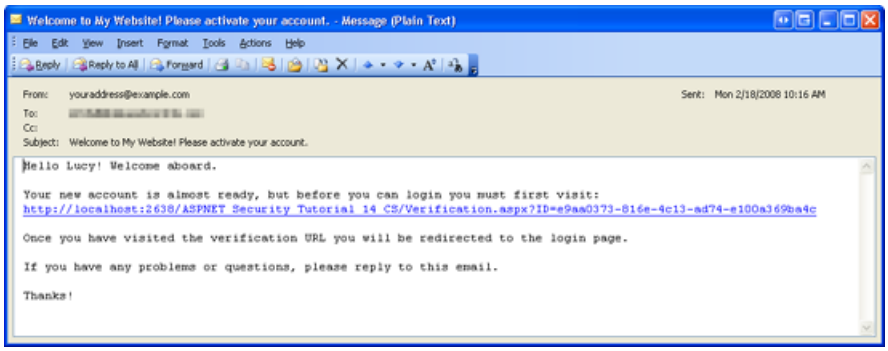


Figure 6: The New User Receives an Email with a Link to the Verification URL (Click to view full-size image)

> **NOTE**
>
> The CreateUserWizard control's default CreateUserWizard step displays a message informing the user their account has been created and displays a Continue button. Clicking this takes the user to the URL specified by the control's `ContinueDestinationPageUrl` property. The CreateUserWizard in `EnhancedCreateUserWizard.aspx` is configured to send new users to the `~/Membership/AdditionalUserInfo.aspx`, which prompts the user for their hometown, homepage URL, and signature. Because this information can only be added by logged on users, it makes sense to update this property to send users back to the site's homepage (`~/Default.aspx`). Moreover, the `EnhancedCreateUserWizard.aspx` page or the CreateUserWizard step should be augmented to inform the user that they have been sent a verification email and their account won't be activated until they follow the instructions in this email. I leave these modifications as an exercise for the reader.

Creating the Verification Page

Our final task is to create the `Verification.aspx` page. Add this page to the root folder, associating it with the `Site.master` master page. As we've done with most of the previous content pages added to the site, remove the Content control that references the `LoginContent` ContentPlaceHolder so that the content page uses the master page's default content.

Add a Label Web control to the `Verification.aspx` page, set its `ID` to `StatusMessage` and clear out its text property. Next, create the `Page_Load` event handler and add the following code:

```
protected void Page_Load(object sender, EventArgs e)
{
     if (string.IsNullOrEmpty(Request.QueryString["ID"]))
          StatusMessage.Text = "The UserId was not included in the querystring...";
     else
     {
          Guid userId;
          try
          {
               userId = new Guid(Request.QueryString["ID"]);
          }

          catch
          {
               StatusMessage.Text = "The UserId passed into the querystring is not in the
                    proper format...";
               return;
          }

          MembershipUser usr = Membership.GetUser(userId);
          if (usr == null)
               StatusMessage.Text = "User account could not be found...";
          else
          {
               // Approve the user
               usr.IsApproved = true;

               Membership.UpdateUser(usr);
               StatusMessage.Text = "Your account has been approved.
                    Please <a href=\"Login.aspx\">login</a> to the site.";
          }
     }
}
```

The bulk of the above code verifies that the `UserId` supplied through the querystring exists, that it is a valid `Guid` value, and that it references an existing user account. If all of these checks pass, the user account is approved; otherwise, a suitable status message is displayed.

Figure 7 shows the `Verification.aspx` page when visited through a browser.

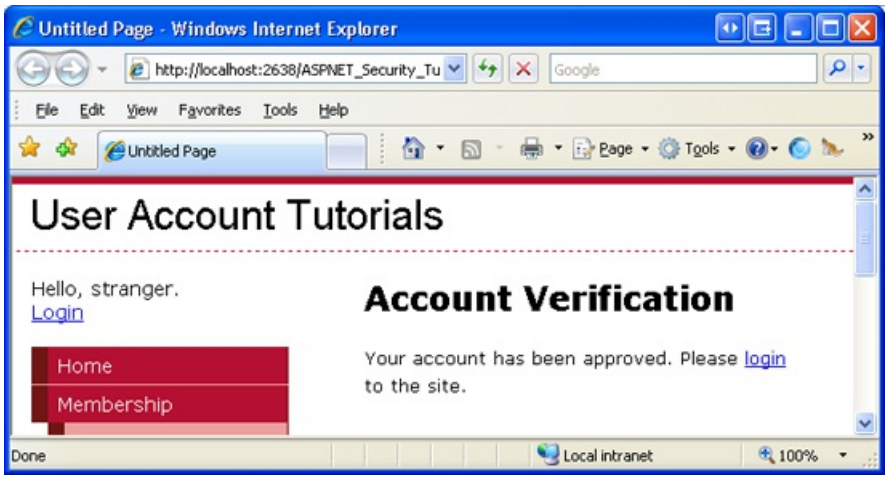


Figure 7: The New User's Account is Now Approved (Click to view full-size image)

Summary

All Membership user accounts have two statuses that determine whether the user can log into the site: `IsLockedOut` and `IsApproved`. Both of these properties must be `true` for the user to login.

The user's locked out status is used as a security measure to reduce the likelihood of a hacker breaking into a site

through brute force methods. Specifically, a user is locked out if there are a certain number of invalid login attempts within a certain window of time. These bounds are configurable through the Membership provider settings in `Web.config`.

The approved status is commonly used as a means to prohibit new users from logging in until some action has transpired. Perhaps the site requires that new accounts first be approved by the administrator or, as we saw in Step 3, by verifying their email address.

Happy Programming!

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Building an Interface to Select One User Account from Many (VB)

9/16/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

In this tutorial we will build a user interface with a paged, filterable grid. In particular, our user interface will consist of a series of LinkButtons for filtering the results based on the starting letter of the username, and a GridView control to show the matching users. We'll start by listing all of the user accounts in a GridView. Then, in Step 3, we will add the filter LinkButtons. Step 4 looks at paging the filtered results. The interface constructed in Steps 2 through 4 will be used in the subsequent tutorials to perform administrative tasks for a particular user account.

Introduction

In the *Assigning Roles to Users* tutorial, we created a rudimentary interface for an administrator to select a user and manage her roles. Specifically, the interface presented the administrator with a drop-down list of all of the users. Such an interface is suitable when there are but a dozen or so user accounts, but is unwieldy for sites with hundreds or thousands of accounts. A paged, filterable grid is more suitable user interface for websites with large user bases.

In this tutorial we will build such a user interface. In particular, our user interface will consist of a series of LinkButtons for filtering the results based on the starting letter of the username, and a GridView control to show the matching users. We'll start by listing all of the user accounts in a GridView. Then, in Step 3, we will add the filter LinkButtons. Step 4 looks at paging the filtered results. The interface constructed in Steps 2 through 4 will be used in the subsequent tutorials to perform administrative tasks for a particular user account.

Let's get started!

Step 1: Adding New ASP.NET Pages

In this tutorial and the next two we will be examining various administration-related functions and capabilities. We will need a series of ASP.NET pages to implement the topics examined throughout these tutorials. Let's create these pages and update the site map.

Start by creating a new folder in the project named `Administration`. Next, add two new ASP.NET pages to the folder, linking each page with the `Site.master` master page. Name the pages:

- `ManageUsers.aspx`
- `UserInformation.aspx`

Also add two pages to the website's root directory: `ChangePassword.aspx` and `RecoverPassword.aspx`.

These four pages should, at this point, have two Content controls, one for each of the master page's ContentPlaceHolders: `MainContent` and `LoginContent`.

```
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="LoginContent" Runat="Server">
</asp:Content>
```

We want to show the master page's default markup for the `LoginContent` ContentPlaceHolder for these pages. Therefore, remove the declarative markup for the `Content2` Content control. After doing so, the pages' markup should contain just one Content control.

The ASP.NET pages in the `Administration` folder are intended solely for administrative users. We added an Administrators role to the system in the *Creating and Managing Roles* tutorial; restrict access to these two pages to this role. To accomplish this, add a `Web.config` file to the `Administration` folder and configure its `<authorization>` element to admit users in the Administrators role and to deny all others.

```
<?xml version="1.0"?>
<configuration>
 <system.web>
 <authorization>
 <allow roles="Administrators" />
 <deny users="*"/>
 </authorization>
 </system.web>
</configuration>
```

At this point your project's Solution Explorer should look similar to the screen shot shown in Figure 1.

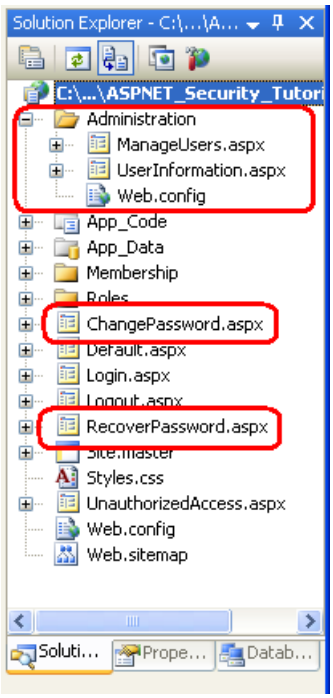


Figure 1: Four New Pages and a `Web.config` File Have Been Added to the Website (Click to view full-size image)

Finally, update the site map (`Web.sitemap`) to include an entry to the `ManageUsers.aspx` page. Add the following XML after the `<siteMapNode>` we added for the Roles tutorials.

```
<siteMapNode title="User Administration" url="~/Administration/ManageUsers.aspx"/>
```

With the site map updated, visit the site through a browser. As Figure 2 shows, the navigation on the left now includes items for the Administration tutorials.

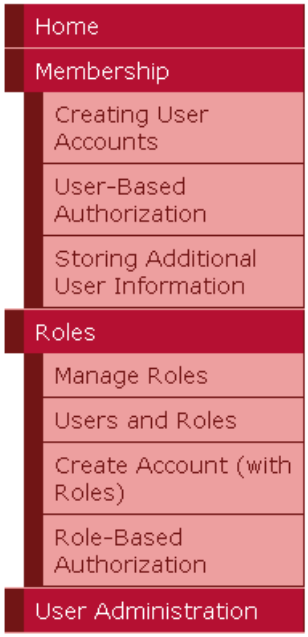


Figure 2: The Site Map Includes a Node Titled User Administration (Click to view full-size image)

Step 2: Listing All User Accounts in a GridView

Our end goal for this tutorial is to create a paged, filterable grid through which an administrator can select a user account to manage. Let's start by listing *all* users in a GridView. Once this is complete, we will add the filtering and paging interfaces and functionality.

Open the `ManageUsers.aspx` page in the `Administration` folder and add a GridView, setting its `ID` to `UserAccounts` In a moment, we will write code to bind the set of user accounts to the GridView using the `Membership` class's `GetAllUsers` method. As discussed in earlier tutorials, the `GetAllUsers` method returns a `MembershipUserCollection` object, which is a collection of `MembershipUser` objects. Each `MembershipUser` in the collection includes properties like `UserName`, `Email`, `IsApproved`, and so on.

In order to display the desired user account information in the GridView, set the GridView's `AutoGenerateColumns` property to False and add BoundFields for the `UserName`, `Email`, and `Comment` properties and CheckBoxFields for the `IsApproved`, `IsLockedOut`, and `IsOnline` properties. This configuration can be applied through the control's declarative markup or via the Fields dialog box. Figure 3 shows a screen shot of the Fields dialog box after the Auto-generate fields checkbox has been unchecked and the BoundFields and CheckBoxFields have been added and configured.

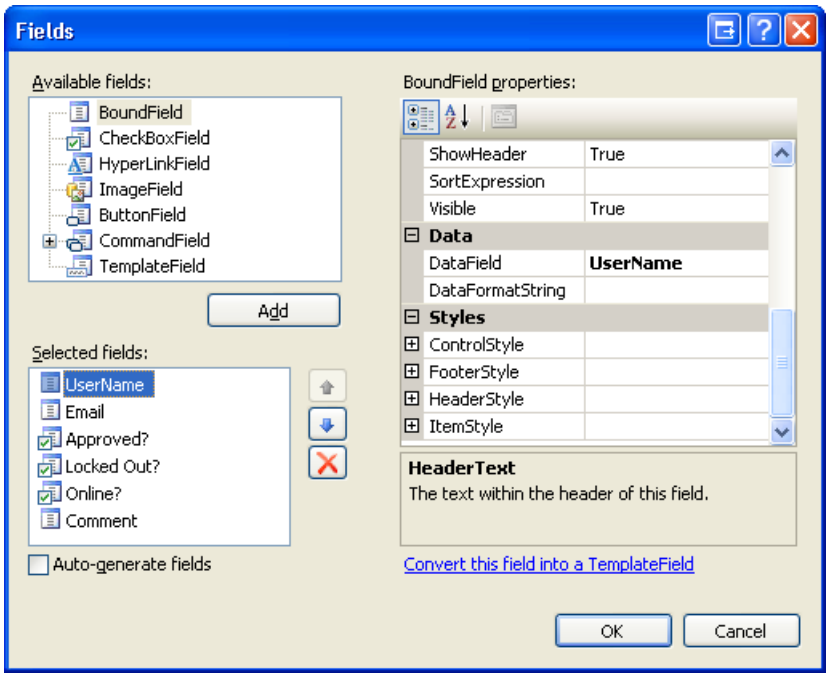


Figure 3: Add Three BoundFields and Three CheckBoxFields to the GridView (Click to view full-size image)

After configuring your GridView, ensure that its declarative markup resembles the following:

```
<asp:GridView ID="UserAccounts" runat="server" AutoGenerateColumns="False">
 <Columns>
 <asp:BoundField DataField="UserName" HeaderText="UserName" />
 <asp:BoundField DataField="Email" HeaderText="Email" />
 <asp:CheckBoxField DataField="IsApproved" HeaderText="Approved?" />
 <asp:CheckBoxField DataField="IsLockedOut" HeaderText="Locked Out?" />
 <asp:CheckBoxField DataField="IsOnline" HeaderText="Online?" />
 <asp:BoundField DataField="Comment" HeaderText="Comment" />
 </Columns>
</asp:GridView>
```

Next, we need to write code that binds the user accounts to the GridView. Create a method named `BindUserAccounts` to perform this task and then call it from the `Page_Load` event handler on the first page visit.

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
 If Not Page.IsPostBack Then
 BindUserAccounts()
 End If
End Sub

Private Sub BindUserAccounts()
 UserAccounts.DataSource = Membership.GetAllUsers()
 UserAccounts.DataBind()
End Sub
```

Take a moment to test the page through a browser. As Figure 4 shows, the `UserAccounts` GridView lists the username, email address, and other pertinent account information for all users in the system.

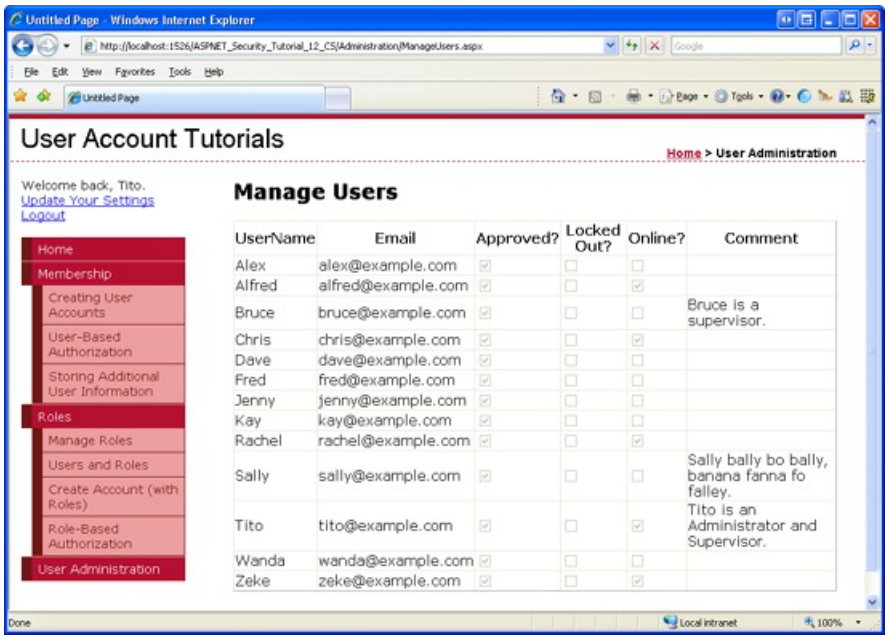


Figure 4: The User Accounts are Listed in the GridView (Click to view full-size image)

Step 3: Filtering the Results by the First Letter of the Username

Currently the `UserAccounts` GridView shows *all* of the user accounts. For websites with hundreds or thousands of user accounts, it is imperative that user be able to quickly pare down the displayed accounts. This can be accomplished by adding filtering LinkButtons to the page. Let's add 27 LinkButtons to the page: one titled All along with one LinkButton for each letter of the alphabet. If a visitor clicks the All LinkButton, the GridView will show all of the users. If they click a particular letter, only those users whose username starts with the selected letter will be displayed.

Our first task is to add the 27 LinkButton controls. One option would be to create the 27 LinkButtons declaratively, one at a time. A more flexible approach is to use a Repeater control with an `ItemTemplate` that renders a LinkButton and then binds the filtering options to the Repeater as a `String` array.

Start by adding a Repeater control to the page above the `UserAccounts` GridView. Set the Repeater's `ID` property to `FilteringUI` Configure the Repeater's templates so that its `ItemTemplate` renders a LinkButton whose `Text` and `CommandName` properties are bound to the current array element. As we saw in the *Assigning Roles to Users* tutorial, this can be accomplished using the `Container.DataItem` databinding syntax. Use the Repeater's `SeparatorTemplate` to display a vertical line between each link.

```
<asp:Repeater ID="FilteringUI" runat="server">
 <ItemTemplate>
 <asp:LinkButton runat="server" ID="lnkFilter"
 Text='<%# Container.DataItem %>'
 CommandName='<%# Container.DataItem %>'></asp:LinkButton>
 </ItemTemplate>
 <SeparatorTemplate>|</SeparatorTemplate>
</asp:Repeater>
```

To populate this Repeater with the desired filtering options, create a method named `BindFilteringUI`. Be sure to call this method from the `Page_Load` event handler on the first page load.

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
 If Not Page.IsPostBack Then
 BindUserAccounts()
 BindFilteringUI()
 End If
End Sub

Private Sub BindFilteringUI()
 Dim filterOptions() As String = {"All", "A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M",
"N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"}
 FilteringUI.DataSource = filterOptions
 FilteringUI.DataBind()
End Sub
```

This method specifies the filtering options as elements in the `String` array `filterOptions` For each element in the array, the Repeater will render a LinkButton with its `Text` and `CommandName` properties assigned to the value of the array element.

Figure 5 shows the `ManageUsers.aspx` page when viewed through a browser.

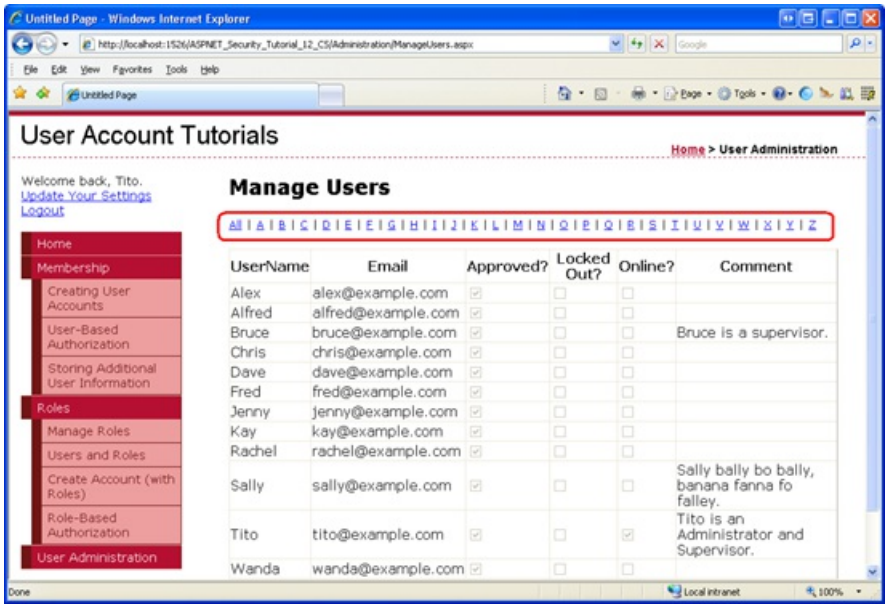


Figure 5: The Repeater Lists 27 Filtering LinkButtons (Click to view full-size image)

> **NOTE**
>
> Usernames may start with any character, including numbers and punctuation. In order to view these accounts, the administrator will have to use the All LinkButton option. Alternatively, you could add a LinkButton to return all user accounts that start with a number. I leave this as an exercise for the reader.

Clicking any of the filtering LinkButtons causes a postback and raises the Repeater's `ItemCommand` event, but there's no change in the grid because we've yet to write any code to filter the results. The `Membership` class includes a `FindUsersByName` method that returns those user accounts whose username matches a specified search pattern. We can use this method to retrieve just those user accounts whose usernames start with the letter specified by the `CommandName` of the filtered LinkButton that was clicked.

Start by updating the `ManageUser.aspx` page's code-behind class so that it includes a property named `UsernameToMatch` This property persists the username filter string across postbacks:

```
Private Property UsernameToMatch() As String
 Get
 Dim o As Object = ViewState("UsernameToMatch")
 If o Is Nothing Then
 Return String.Empty
 Else
 Return o.ToString()
 End If
 End Get
 Set(ByVal Value As String)
 ViewState("UsernameToMatch") = Value
 End Set
End Property
```

The `UsernameToMatch` property stores its value it is assigned into the `ViewState` collection using the key 'UsernameToMatch'. When this property's value is read, it checks to see if a value exists in the `ViewState` collection; if not, it returns the default value, an empty string. The `UsernameToMatch` property exhibits a common pattern, namely persisting a value to view state so that any changes to the property are persisted across postbacks. For more information on this pattern, read Understanding ASP.NET View State.

Next, update the `BindUserAccounts` method so that instead of calling `Membership.GetAllUsers`, it calls `Membership.FindUsersByName`, passing in the value of the `UsernameToMatch` property appended with the SQL wildcard character, %.

```
Private Sub BindUserAccounts()
 UserAccounts.DataSource = Membership.FindUsersByName(Me.UsernameToMatch &"%")
 UserAccounts.DataBind()
End Sub
```

To display just those users whose username starts with the letter A, set the `UsernameToMatch` property to A and then call `BindUserAccounts` This would result in a call to `Membership.FindUsersByName("A%")`, which will return all users whose username starts with A. Likewise, to return *all* users, assign an empty string to the `UsernameToMatch` property so that the `BindUserAccounts` method will invoke `Membership.FindUsersByName("%")`, thereby returning all user accounts.

Create an event handler for the Repeater's `ItemCommand` event. This event is raised whenever one of the filter LinkButtons is clicked; it is passed the clicked LinkButton's `CommandName` value through the `RepeaterCommandEventArgs` object. We need to assign the appropriate value to the `UsernameToMatch` property and then call the `BindUserAccounts` method. If the `CommandName` is All, assign an empty string to `UsernameToMatch` so that all user accounts are displayed. Otherwise, assign the `CommandName` value to `UsernameToMatch`

```
Protected Sub FilteringUI_ItemCommand(ByVal source As Object, ByVal e As
System.Web.UI.WebControls.RepeaterCommandEventArgs) Handles FilteringUI.ItemCommand
 If e.CommandName = "All" Then
 Me.UsernameToMatch = String.Empty
 Else
 Me.UsernameToMatch = e.CommandName
 End If

 BindUserAccounts()
End Sub
```

With this code in place, test out the filtering functionality. When the page is first visited, all user accounts are displayed (refer back to Figure 5). Clicking the A LinkButton causes a postback and filters the results, displaying only those user accounts that start with A .

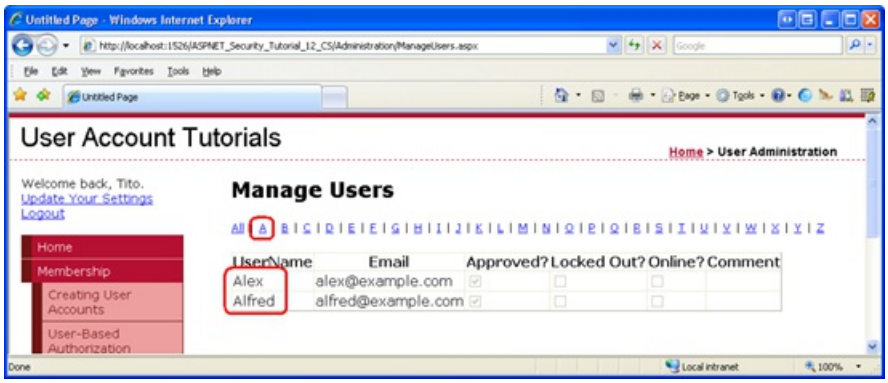


Figure 6: Use the Filtering LinkButtons to Display those Users Whose Username Starts with a Certain Letter (Click to view full-size image)

Step 4: Updating the GridView to Use Paging

The GridView shown in Figures 5 and 6 lists all of the records returned from the `FindUsersByName` method. If there are hundreds or thousands of user accounts this can lead to information overload when viewing all of the accounts (as is the case when clicking the All LinkButton or when initially visiting the page). To help present the user accounts in more manageable chunks, let's configure the GridView to display 10 user accounts at a time.

The GridView control offers two types of paging:

- **Default paging** - easy to implement, but inefficient. In a nutshell, with default paging the GridView expects *all* of the records from its data source. It then only displays the appropriate page of records.
- **Custom paging** - requires more work to implement, but is more efficient than default paging because with custom paging the data source returns only the precise set of records to display.

The performance difference between default and custom paging can be quite substantial when paging through thousands of records. Because we are building this interface assuming that there may be hundreds or thousands of user accounts, let's use custom paging.

> **NOTE**
> For a more thorough discussion on the differences between default and custom paging, as well as the challenges involved in implementing custom paging, refer to Efficiently Paging Through Large Amounts of Data. For some analysis of the performance difference between default and custom paging, see Custom Paging in ASP.NET with SQL Server 2005.

To implement custom paging we first need some mechanism by which to retrieve the precise subset of records being displayed by the GridView. The good news is that the `Membership` class's `FindUsersByName` method has an overload that allows us to specify the page index and page size, and returns only those user accounts that fall within that range of records.

In particular, this overload has the following signature:
`FindUsersByName(usernameToMatch, pageIndex, pageSize, totalRecords)`.

The *pageIndex* parameter specifies the page of user accounts to return; *pageSize* indicates how many records to display per page. The *totalRecords* parameter is a `ByRef` parameter that returns the number of total user accounts in the user store.

> **NOTE**
> The data returned by `FindUsersByName` is sorted by username; the sort criteria cannot be customized.

The GridView can be configured to utilize custom paging, but only when bound to an ObjectDataSource control.

For the ObjectDataSource control to implement custom paging, it requires two methods: one that is passed a start row index and the maximum number of records to display, and returns the precise subset of records that fall within that span; and a method that returns the total number of records being paged through. The `FindUsersByName` overload accepts a page index and page size, and returns the total number of records through a `ByRef` parameter. So there is an interface mismatch here.

One option would be to create a proxy class that exposes the interface the ObjectDataSource expects, and then internally calls the `FindUsersByName` method. Another option - and the one we'll use for this article - is to create our own paging interface and use that instead of the GridView's built-in paging interface.

Creating a First, Previous, Next, Last Paging Interface

Let's build a paging interface with First, Previous, Next, and Last LinkButtons. The First LinkButton, when clicked, will take the user to the first page of data, whereas Previous will return him to the previous page. Likewise, Next and Last will move the user to the next and last page, respectively. Add the four LinkButton controls beneath the `UserAccounts` GridView.

```
<p>
 <asp:LinkButton ID="lnkFirst" runat="server">  First</asp:LinkButton> |
 <asp:LinkButton ID="lnkPrev" runat="server">  Prev</asp:LinkButton> |
 <asp:LinkButton ID="lnkNext" runat="server">Next  </asp:LinkButton> |
 <asp:LinkButton ID="lnkLast" runat="server">Last  </asp:LinkButton>
</p>
```

Next, create an event handler for each of the LinkButton's `Click` events.

Figure 7 shows the four LinkButtons when viewed through the Visual Web Developer Design view.

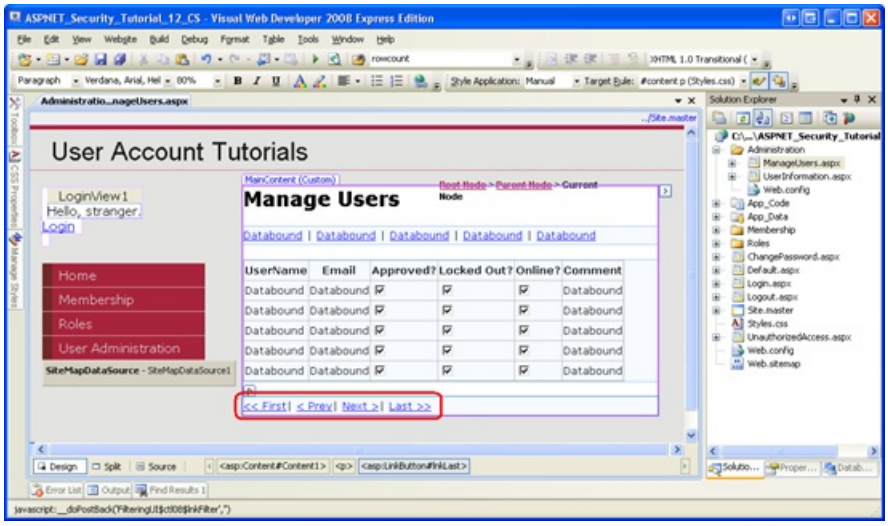


Figure 7: Add First, Previous, Next, and Last LinkButtons Beneath the GridView (Click to view full-size image)

Keeping Track of the Current Page Index

When a user first visits the `ManageUsers.aspx` page or clicks one of the filtering buttons, we want to display the first page of data in the GridView. When the user clicks one of the navigation LinkButtons, however, we need to update the page index. To maintain the page index and the number of records to display per page, add the following two properties to the page's code-behind class:

```
Private Property PageIndex() As Integer
 Get
 Dim o As Object = ViewState("PageIndex")
 If o Is Nothing Then
 Return 0
 Else
 Return Convert.ToInt32(o)
 End If
 End Get
 Set(ByVal Value As Integer)
 ViewState("PageIndex") = Value
 End Set
End Property

Private ReadOnly Property PageSize() As Integer
 Get
 Return 10
 End Get
End Property
```

Like the `UsernameToMatch` property, the `PageIndex` property persists its value to view state. The read-only `PageSize` property returns a hard-coded value, 10. I invite the interested reader to update this property to use the same pattern as `PageIndex`, and then to augment the `ManageUsers.aspx` page such that the person visiting the page can specify how many user accounts to display per page.

Retrieving Just the Current Page's Records, Updating the Page Index, and Enabling and Disabling the Paging Interface LinkButtons

With the paging interface in place and the `PageIndex` and `PageSize` properties added, we are ready to update the `BindUserAccounts` method so that it uses the appropriate `FindUsersByName` overload. Additionally, we need to have this method enable or disable the paging interface depending on what page is being displayed. When viewing the first page of data, the First and Previous links should be disabled; Next and Last should be disabled when viewing the last page.

Update the `BindUserAccounts` method with the following code:

```
Private Sub BindUserAccounts()
 Dim totalRecords As Integer
 UserAccounts.DataSource = Membership.FindUsersByName(Me.UsernameToMatch + "%", Me.PageIndex, Me.PageSize,
totalRecords)
 UserAccounts.DataBind()

 ' Enable/disable the paging interface
 Dim visitingFirstPage As Boolean = (Me.PageIndex = 0)
 lnkFirst.Enabled = Not visitingFirstPage
 lnkPrev.Enabled = Not visitingFirstPage

 Dim lastPageIndex As Integer = (totalRecords - 1) / Me.PageSize
 Dim visitingLastPage As Boolean = (Me.PageIndex >= lastPageIndex)
 lnkNext.Enabled = Not visitingLastPage
 lnkLast.Enabled = Not visitingLastPage
End Sub
```

Note that the total number of records being paged through is determined by the last parameter of the `FindUsersByName` method. After the specified page of user accounts are returned, the four LinkButtons are either enabled or disabled, depending on whether the first or last page of data is being viewed.

The last step is to write the code for the four LinkButtons' `Click` event handlers. These event handlers need to update the `PageIndex` property and then rebind the data to the GridView via a call to `BindUserAccounts` The First, Previous, and Next event handlers are very simple. The `Click` event handler for the Last LinkButton, however, is a bit more complex because we need to determine how many records are being displayed in order to determine

the last page index.

```
Protected Sub lnkFirst_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles lnkFirst.Click
 Me.PageIndex = 0
 BindUserAccounts()
End Sub

Protected Sub lnkPrev_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles lnkPrev.Click
 Me.PageIndex -= 1
 BindUserAccounts()
End Sub

Protected Sub lnkNext_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles lnkNext.Click
 Me.PageIndex += 1
 BindUserAccounts()
End Sub

Protected Sub lnkLast_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles lnkLast.Click
 ' Determine the total number of records
 Dim totalRecords As Integer
 Membership.FindUsersByName(Me.UsernameToMatch + "%", Me.PageIndex, Me.PageSize, totalRecords)
 ' Navigate to the last page index
 Me.PageIndex = (totalRecords - 1) / Me.PageSize
 BindUserAccounts()
End Sub
```

Figures 8 and 9 show the custom paging interface in action. Figure 8 shows the `ManageUsers.aspx` page when viewing the first page of data for all user accounts. Note that only 10 of the 13 accounts are displayed. Clicking the Next or Last link causes a postback, updates the `PageIndex` to 1, and binds the second page of user accounts to the grid (see Figure 9).

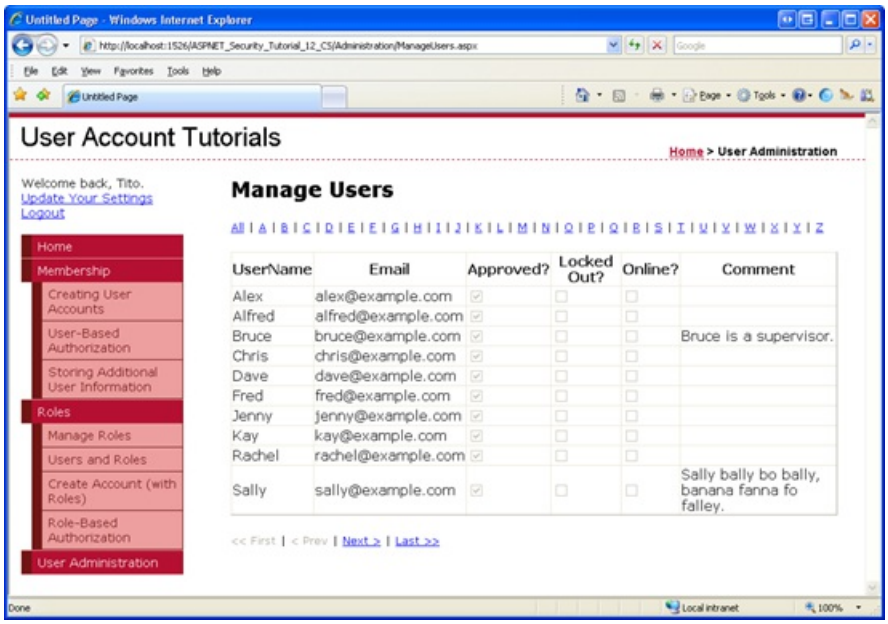


Figure 8: The First 10 User Accounts are Displayed (Click to view full-size image)

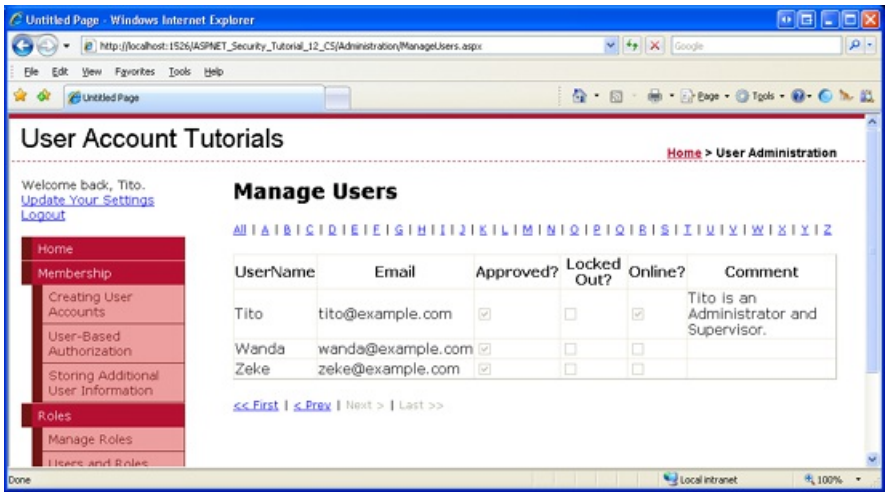


Figure 9: Clicking the Next Link Displays the Second Page of User Accounts (Click to view full-size image)

Summary

Administrators often need to select a user from the list of accounts. In previous tutorials we looked at using a drop-down list populated with the users, but this approach does not scale well. In this tutorial we explored a better alternative: a filterable interface whose results are displayed in a paged GridView. With this user interface, administrators can quickly and efficiently locate and select one user account among thousands.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- Custom Paging in ASP.NET with SQL Server 2005
- Efficiently Paging Through Large Amounts of Data
- Rolling Your Own Website Administration Tool

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewer for this tutorial was Alicja Maziarz. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at

Recovering and Changing Passwords (VB)

2/20/2020 • 20 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

ASP.NET includes two Web controls for assisting with recovering and changing passwords. The PasswordRecovery control enables a visitor to recover his lost password. The ChangePassword control allows the user to update his password. Like the other Login-related Web controls we've seen throughout this tutorial series, the PasswordRecovery and ChangePassword controls work with the Membership framework behind the scenes to reset or modify users' passwords.

Introduction

Between the websites for my bank, utility company, phone company, email accounts, and personalized web portals, I, like most people, have dozens of different passwords to remember. With so many credentials to memorize these days, it's not uncommon for people to forget their password. To account for this, websites that offer user accounts need to include a way for a user to recover his password. This process typically involves generating a new, random password and emailing it to the user's email address on file. After receiving their new password most users return to the site and change their password from the randomly generated one to a more memorable one.

ASP.NET includes two Web controls for assisting with recovering and changing passwords. The PasswordRecovery control enables a visitor to recover his lost password. The ChangePassword control allows the user to update his password. Like the other Login-related Web controls we've seen throughout this tutorial series, the PasswordRecovery and ChangePassword controls work with the Membership framework behind the scenes to reset or modify users' passwords.

In this tutorial we will examine using these two controls. We will also see how to programmatically change and reset a user's password via the `MembershipUser` class's `ChangePassword` and `ResetPassword` methods.

Step 1: Helping Users Recover Lost Passwords

All websites that support user accounts need to provide users with some mechanism for recovering their forgotten passwords. The good news is that implementing such functionality in ASP.NET is a breeze thanks to the PasswordRecovery Web control. The PasswordRecovery control renders an interface that prompts the user for their username and, if needed, the answer to their security question. It then emails the user their password.

> **NOTE**
>
> Because email messages are transmitted over the wire in plain-text there are security risks involved with sending a user's password via email.

The PasswordRecovery control consists of three views:

- **UserName** - prompts the visitor for their username. This is the initial view.
- **Question**- displays the user's username and security question as text, along with a TextBox for the user to enter the answer to his security question.
- **Success**- displays a message informing the user that his password has been emailed.

The views displayed and actions performed by the PasswordRecovery control depend upon the following Membership configuration settings:

- `RequiresQuestionAndAnswer`
- `EnablePasswordRetrieval`
- `EnablePasswordReset`

The Membership framework's `RequiresQuestionAndAnswer` setting indicates whether users must specify a security question and answer when registering for an account. As we discussed in the *Creating User Accounts* tutorial, if `RequiresQuestionAndAnswer` is True (the default) then the CreateUserWizard's interface includes TextBox controls for the new user's security question and answer; if `RequiresQuestionAndAnswer` is False, no such information is collected. Similarly, if `RequiresQuestionAndAnswer` is True, then the PasswordRecovery control displays the Question view after the user has entered their username; the password is recovered only if the user enters the correct security answer. If `RequiresQuestionAndAnswer` is False, however, the PasswordRecovery control moves straight from the UserName view to the Success view.

After the user has provided his username - or his username and security answer, if `RequiresQuestionAndAnswer` is True - the PasswordRecovery emails the user his password. If the `EnablePasswordRetrieval` option is set to True, then the user is emailed their current password. If it is set to False and `EnablePasswordReset` is set to True, then the PasswordRecovery control generates a new, random password for the user, and emails this new password to them. If both `EnablePasswordRetrieval` and `EnablePasswordReset` are False, the PasswordRecovery control throws an exception.

> **NOTE**
> Recall that the `SqlMembershipProvider` stores users' passwords in one of three formats: Clear, Hashed (the default), or Encrypted. The storage mechanism used depends on the Membership configuration settings; the demo application uses the Hashed password format. When using the Hashed password format the `EnablePasswordRetrieval` option must be set to False because the system cannot determine the user's actual password from the hashed version stored in the database.

Figure 1 illustrates how the PasswordRecovery's interface and behavior is influenced by the Membership configuration.

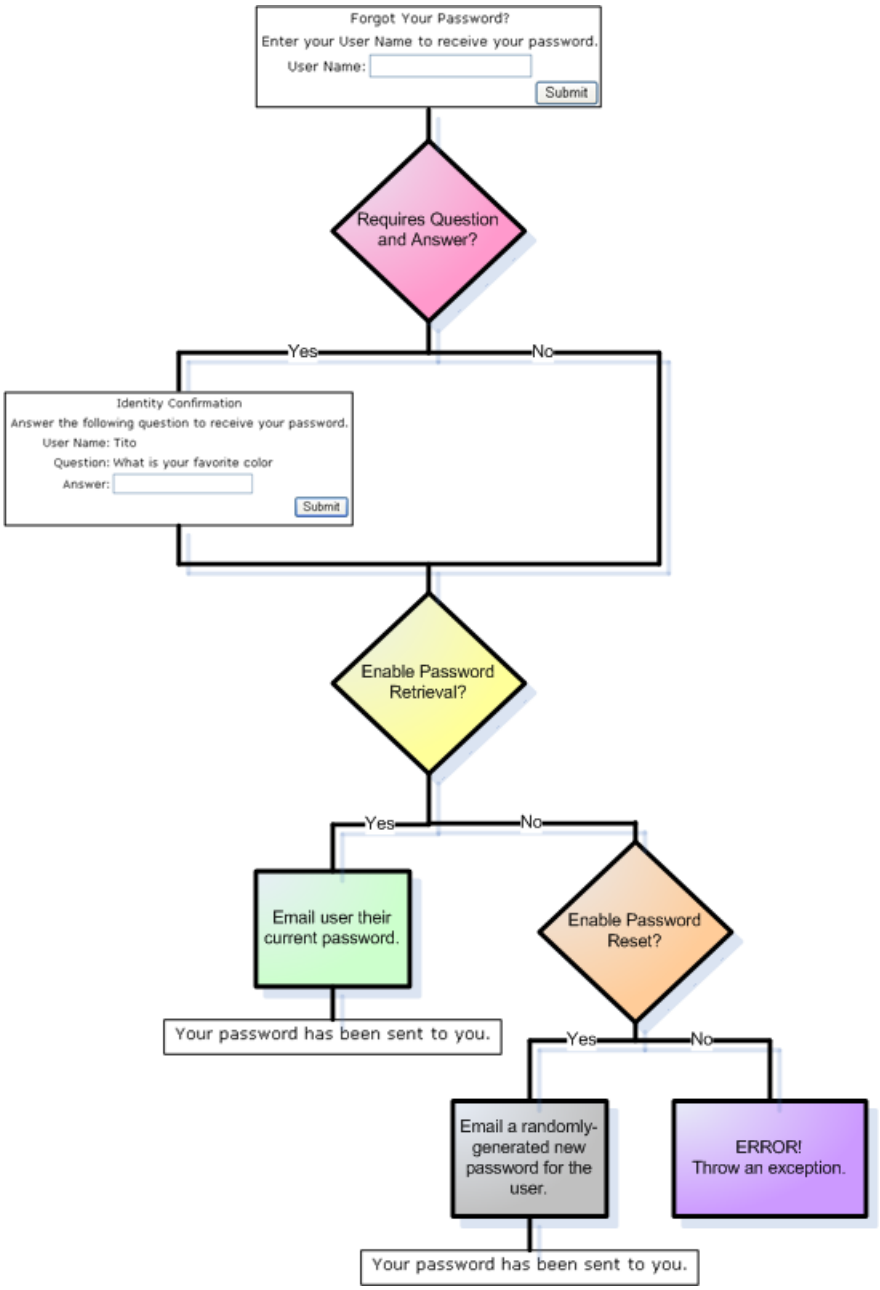


Figure 1: The `RequiresQuestionAndAnswer`, `EnablePasswordRetrieval`, and `EnablePasswordReset` Influence the PasswordRecovery control's Appearance and Behavior (Click to view full-size image)

> **NOTE**
>
> In the *Creating the Membership Schema in SQL Server* tutorial we configured the Membership provider by setting `RequiresQuestionAndAnswer` to True, `EnablePasswordRetrieval` to False, and `EnablePasswordReset` to True.

Using the PasswordRecovery Control

Let's look at using the PasswordRecovery control in an ASP.NET page. Open `RecoverPassword.aspx` and drag and drop a PasswordRecovery control from the Toolbox onto the Designer; set its `ID` to `RecoverPwd`. Like the Login and CreateUserWizard Web controls, the PasswordRecovery control's views render a rich composite interface that includes Labels, TextBoxes, Buttons, and validation controls. You can customize the appearance of the views through the control's style properties or by converting the views into templates. I leave this as an exercise for the interested reader.

When a user visits this page she will enter her username and click the Submit button. Because we have set the `RequiresQuestionAndAnswer` property to True in our Membership configuration settings, the PasswordRecovery control will then display the Question view. After the user enters her correct security answer and clicks Submit,

the PasswordRecovery control will update the user's password to a randomly-generated one, and email this password to the email address on file. All of this was possible without us having to write a single line of code!

Before you test this page, there's one final piece of configuration to tend to: we need to specify the mail delivery settings in `Web.config`. The PasswordRecovery control relies on these settings for sending the email.

The mail delivery configuration is specified through the `<system.net>` element's `<mailSettings>` element. Use the `<smtp>` element to indicate the delivery method and the default From address. The following markup configures mail settings to use a network SMTP server named `smtp.example.com` on port 25 and with username/password credentials of username and password.

> **NOTE**
>
> `<system.net>` is a child element of the root `<configuration>` element and a sibling of `<system.web>`. Therefore, do not put the `<system.net>` element within the `<system.web>` element; instead, put it at the same level.

```
<configuration>
 ...
 <system.net>
 <mailSettings>
 <smtp deliveryMethod="Network" from="youraddress@example.com">
 <network
 host="smtp.example.com"
 userName="username"
 password="password"
 port="25" />
 </smtp>
 </mailSettings>
 </system.net>
</configuration>
```

In addition to using an SMTP server on the network, you can alternatively specify a pickup directory where email messages to be sent should be deposited.

Once you have configured the SMTP settings, visit the `RecoverPassword.aspx` page through a browser. First try entering a username that doesn't exist in the user store. As Figure 2 shows, the PasswordRecovery control displays a message indicating that the user information could not be accessed. The text of the message can be customized through the control's `UserNameFailureText` property.

Figure 2: An Error Message is Displayed if an Invalid Username is Entered (Click to view full-size image)

Now enter a username. Use the username of an account in the system with an email address that you can access and whose security answer you know. After entering the username and clicking Submit, the PasswordRecovery control displays its Question view. As with the UserName view, if you enter an incorrect answer the PasswordRecovery control displays an error message (see Figure 3). Use the `QuestionFailureText` property to customize this error message.

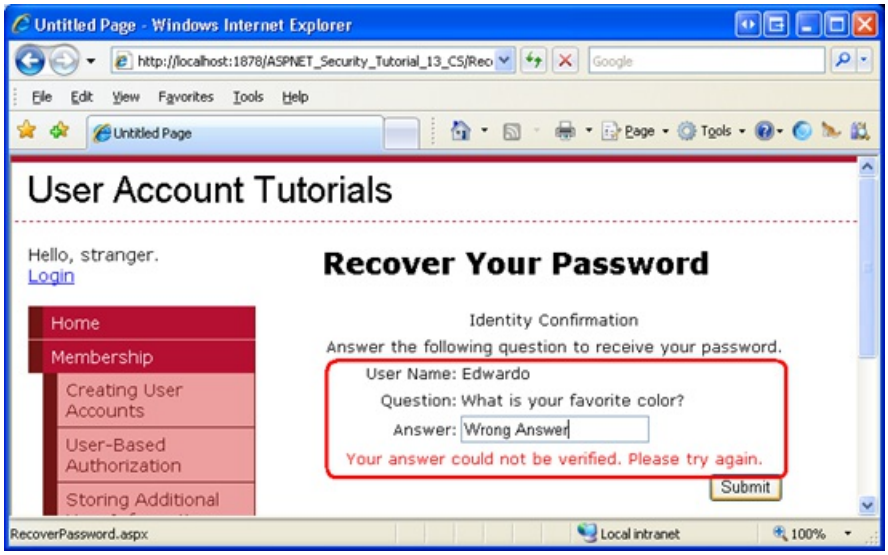


Figure 3: An Error Message is Displayed if the User Enters an Invalid Security Answer (Click to view full-size image)

Finally, enter the correct security answer and click Submit. Behind the scenes, the PasswordRecovery control generates a random password, assigns it to the user account, sends an email informing the user of their new password (see Figure 4), and then displays the Success view.

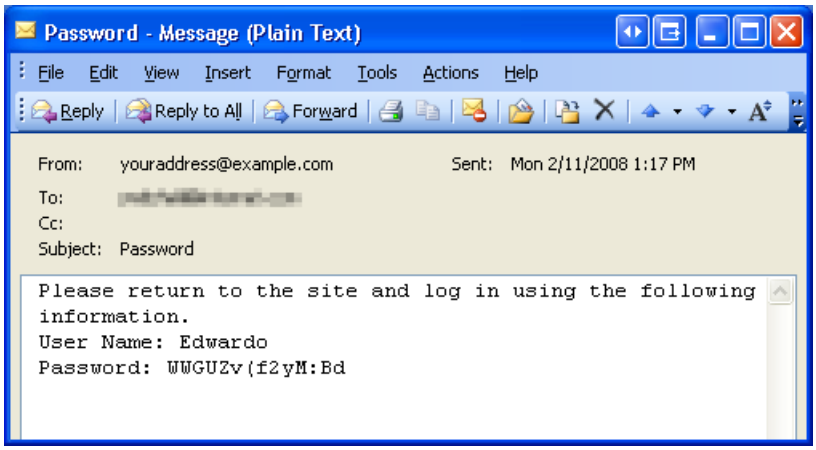


Figure 4: The User is Sent an Email with His New Password (Click to view full-size image)

Customizing the Email

The default email sent by the PasswordRecovery control is rather dull (see Figure 4). The message is sent from the account specified in the `<smtp>` element's `from` attribute with the subject Password and the plain-text body:

Please return to the site and log in using the following information.

User Name: *username*

Password: *password*

This message can be customized programmatically through an event handler for the PasswordRecovery control's `SendingMail` event, or declaratively through the `MailDefinition` property. Let's explore both of these options.

The `SendingMail` event is fired right before the email message is sent and is our last chance to programmatically

adjust the email message. When this event is raised, the event handler is passed an object of type `MailMessageEventArgs`, whose `Message` property contains a reference to the email about to be sent.

Create an event handler for the `SendingMail` event and add the following code, which programmatically adds `webmaster@example.com` to the CC list.

```
Protected Sub RecoverPwd_SendingMail(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.MailMessageEventArgs)Handles RecoverPwd.SendingMail
 e.Message.CC.Add("webmaster@example.com")
End Sub
```

The email message can also be configured through declarative means. The PasswordRecovery's `MailDefinition` property is an object of type `MailDefinition`. The `MailDefinition` class offers a host of email-related properties, including `From`, `CC`, `Priority`, `Subject`, `IsBodyHtml`, `BodyFileName`, and others. For starters, set the `Subject` property to something more descriptive than the one used by default (Password), such as Your password has been reset...

To customize the body of the email message we need to create a separate email template file that contains the body's contents. Start by creating a new folder in the website named `EmailTemplates`. Next, add a new text file to this folder named `PasswordRecovery.txt` and add the following content:

```
Your password has been reset, <%UserName%>!

According to our records, you have requested that your password be reset. Your new
password is: <%Password%>

If you have any questions or trouble logging on please contact a site administrator.

Thank you!
```

Note the use of the placeholders `<%UserName%>` and `<%Password%>`. The PasswordRecovery control automatically replaces these two placeholders with the user's username and recovered password prior to sending the email.

Lastly, point the `MailDefinition`'s `BodyFileName` property to the email template we just created (`~/EmailTemplates/PasswordRecovery.txt`).

After making these changes revisit the `RecoverPassword.aspx` page and enter your username and security answer. You receive should an email that looks like the one in Figure 5. Note that `webmaster@example.com` has been CC'd and that the subject and body have been updated.

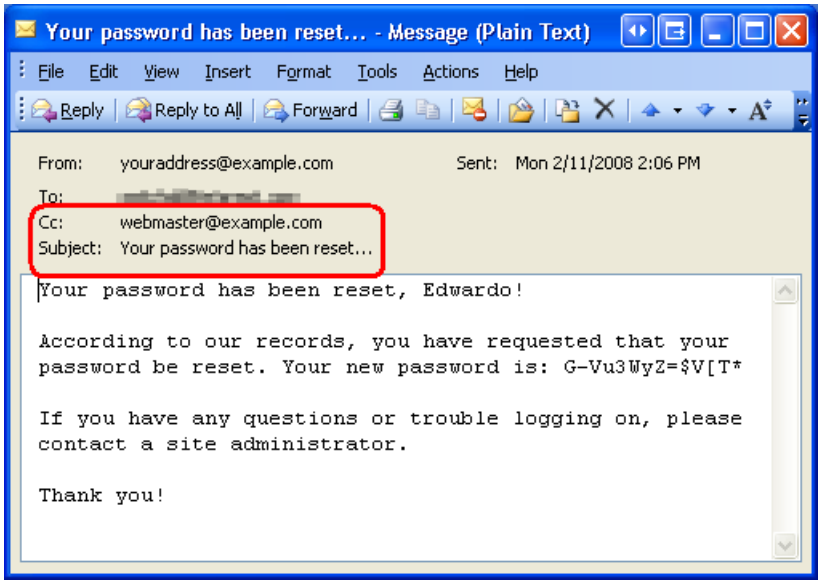


Figure 5: The Subject, Body, and CC List Have Been Updated (Click to view full-size image)

To send an HTML-formatted email set `IsBodyHtml` to True (the default is False) and update the email template to include HTML.

The `MailDefinition` property is not unique to the PasswordRecovery class. As we will see in Step 2, the ChangePassword control also offers a `MailDefinition` property. Moreover, the CreateUserWizard control includes such a property that you can configure to automatically send a welcome email message to new users.

> **NOTE**
> Currently there are no links in the left-hand navigation for reaching the `RecoverPassword.aspx` page. A user would only be interested in visiting this page if she was unable to successfully log on to the site. Therefore, update the `Login.aspx` page to include a link to the `RecoverPassword.aspx` page.

Programmatically Resetting a User's Password

When resetting a user's password the PasswordRecovery control calls the `MembershipUser` object's `ResetPassword` method. This method has two overloads:

- `ResetPassword` - resets a user's password. Use this overload if `RequiresQuestionAndAnswer` is False.
- `ResetPassword(securityAnswer)` - resets a user's password only if the supplied *securityAnswer* is correct. Use this overload if `RequiresQuestionAndAnswer` is True.

Both overloads return the new, randomly-generated password.

Like with the other methods in the Membership framework, the `ResetPassword` method delegates to the configured provider. The `SqlMembershipProvider` invokes the `aspnet_Membership_ResetPassword` stored procedure, passing in the user's username, the new password, and the supplied password answer, among other fields. The stored procedure ensures that the password answer matches and then updates the user's password.

A couple of low-level implementation notes:

- A locked out user cannot reset her password. However, an unapproved user may. We will discuss the locked out and approved states in more detail in the *Unlocking and Approving User* Accounts tutorial.
- If the password answer is incorrect, the user's failed password answer attempt count is incremented. If a specified number of invalid security answer attempts occur within a specified time window, the user is locked out.

A Word on How the Random Passwords are Generated

The randomly-generated passwords shown in the email messages in Figures 4 and 5 are created by the Membership class's `GeneratePassword` method. This method accepts two integer input paramters - *length* and *numberOfNonAlphanumericCharacters* - and returns a string at least *length* characters long with at least *numberOfNonAlphanumericCharacters* number of non-alphanumeric characters. When this method is called from within the Membership classes or Login-related Web controls, the values for these two parameters are determined by the Membership configuration's `MinRequiredPasswordLength` and `MinRequiredNonalphanumericCharacters` properties, which we set to 7 and 1, respectively.

The `GeneratePassword` method uses a cryptographically strong random number generator to ensure that there is no bias in what random characters are selected. Furthermore, `GeneratePassword` is `Public`, meaning that you can use it directly from your ASP.NET application if you need to generate random strings or passwords.

NOTE

The `SqlMembershipProvider` class always generates a random password at least 14 characters long, so if `MinRequiredPasswordLength` is less than 14 then its value is ignored.

Step 2: Changing Passwords

The randomly-generated passwords are difficult to remember. Consider the password shown in Figure 4: `WWGUZv(f2yM:Bd`. Try committing that to memory! Needless to say, after a user is sent a randomly-generated password of this sort, she'll want to change the password to something more memorable.

Use the ChangePassword control to create an interface for a user to change her password. Much like the PasswordRecovery control, the ChangePassword control consists of two views: Change Password and Success. The Change Password view prompts the user for their old and new passwords. Upon supplying the correct old password and a new password that meets the minimum length and non-alphanumeric character requirements, the ChangePassword control updates the user's password and displays the Success view.

NOTE

The ChangePassword control modifies the user's password by invoking the `MembershipUser` object's `ChangePassword` method. The ChangePassword method accepts two `String` input parameters - *oldPassword* and *newPassword*- and updates the user's account with the *newPassword*, assuming the supplied *oldPassword* is correct.

Open the `ChangePassword.aspx` page and add a ChangePassword control to the page, naming it `ChangePwd`. At this point, the Design view should show the Change Password view (see Figure 6). Like with the PasswordRecovery control, you can toggle between the views via the control's Smart Tag. Furthermore, these views' appearances are customizable through the assorted style properties or by converting them to a template.

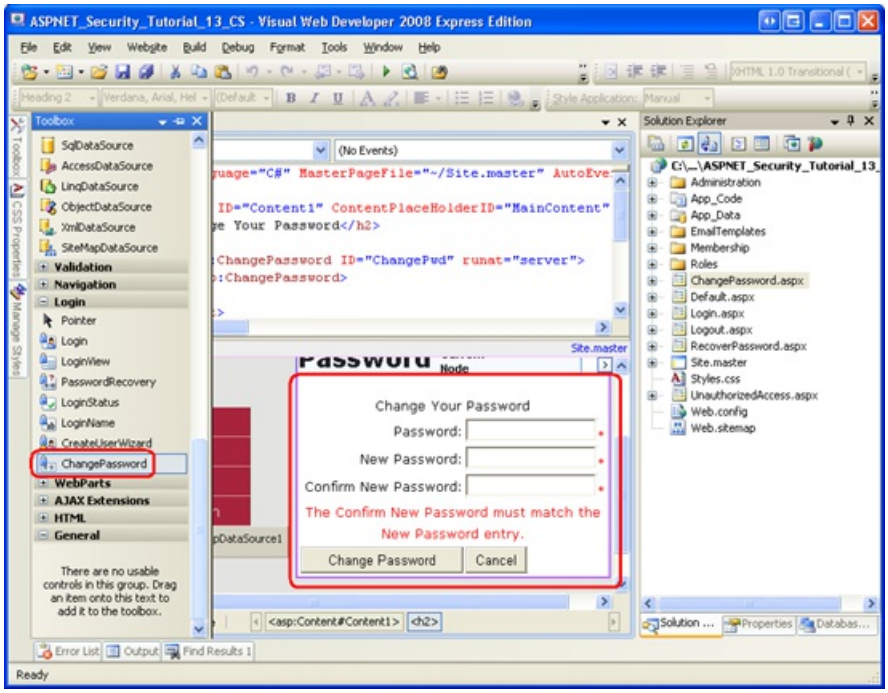


Figure 6: Add a ChangePassword Control to the Page (Click to view full-size image)

The ChangePassword control can update the currently logged in user's password *or* the password of another, specified user. As Figure 6 shows, the default Change Password view renders just three TextBox controls: one for the old password and two for the new password. This default interface is used to update the currently logged on user's password.

To use the ChangePassword control to update another user's password, set the control's `DisplayUserName` property to True. Doing so adds a fourth TextBox to the page, prompting for the username of the user whose password to change.

Setting `DisplayUserName` to True is useful if you want to let a logged out user change her password without having to log in. Personally, I think there's nothing wrong with requiring a user to login before allowing her to change her password. Therefore, leave `DisplayUserName` set to False (its default). In making this decision, however, we essentially are barring anonymous users from reaching this page. Update the site's URL authorization rules so as to deny anonymous users from visiting `ChangePassword.aspx`. If you need to refresh your memory on the URL authorization rule syntax, refer back to the *User-Based Authorization* tutorial.

> **NOTE**
>
> It may seem that the `DisplayUserName` property is useful for allowing administrators to change other users' passwords. However, even when `DisplayUserName` is set to True the correct old password must be known and entered. We will talk about techniques for allowing administrators to change users' passwords in Step 3.

Visit the `ChangePassword.aspx` page through a browser and change your password. Note that an error message is displayed if you enter a new password that fails to meet the password length and non-alphanumeric character requirements specified in the Membership configuration (see Figure 7).

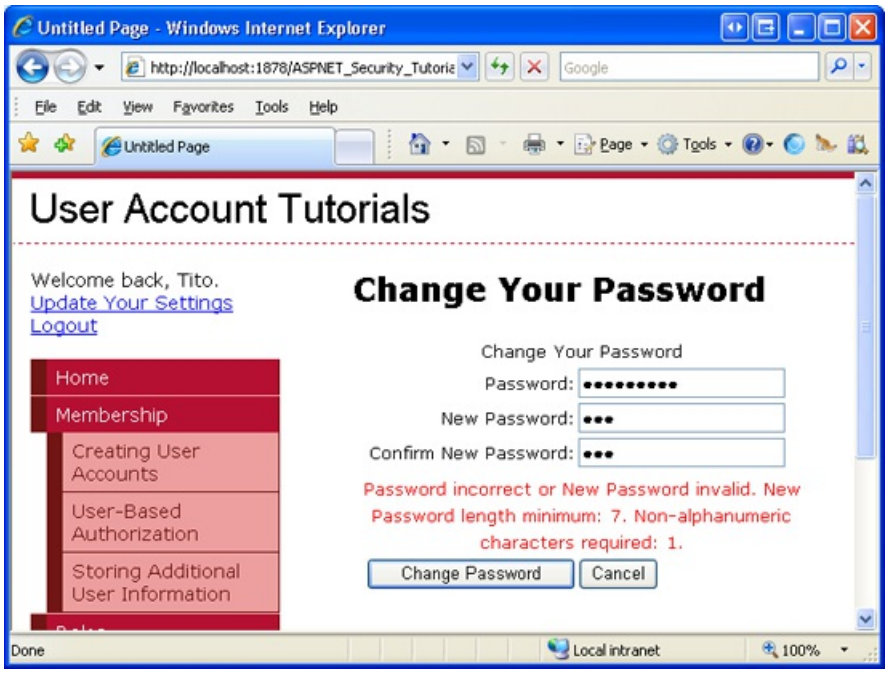


Figure 7: Add a ChangePassword Control to the Page (Click to view full-size image)

Upon entering the correct old password and a valid new password, the logged on user's password is changed and the Success view displayed.

Sending a Confirmation Email

By default, the ChangePassword control does not send an email message to the user whose password was just updated. If you'd like to send an email, simply configure the control's `MailDefinition` property. Let's configure the ChangePassword control so that the user is sent an HTML-formatted email that contains their new password.

Start by creating a new file in the `EmailTemplates` folder named `ChangePassword.htm`. Add the following markup:

```
<html>
 <body>
 <h2>Your Password Has Been Changed!</h2>
 <p>
 This email confirms that your password has been changed.
 </p>
 <p>
 To log on to the site, use the following credentials:
 </p>
 <table>
 <tr>
 <td>
 <b>Username:</b>
 </td>
 <td>
 <%UserName%>
 </td>
 </tr>
 <tr>
 <td>
 <b>Password:</b>
 </td>
 <td>
 <%Password%>
 </td>
 </tr>
 </table>
 <p>
 If you have any questions or encounter any problems logging in,
 please contact a site administrator.
 </p>
 </body>
</html>
```

Next, set the ChangePassword control's `MailDefinition` property's `BodyFileName`, `IsBodyHtml`, and `Subject` properties to ~/EmailTemplates/ChangePassword.htm, True, and Your password has changed!, respectively.

After making these changes, revisit the page and change your password again. This time, the ChangePassword control sends a customized, HTML-formatted email to the user's email address on file (see Figure 8).

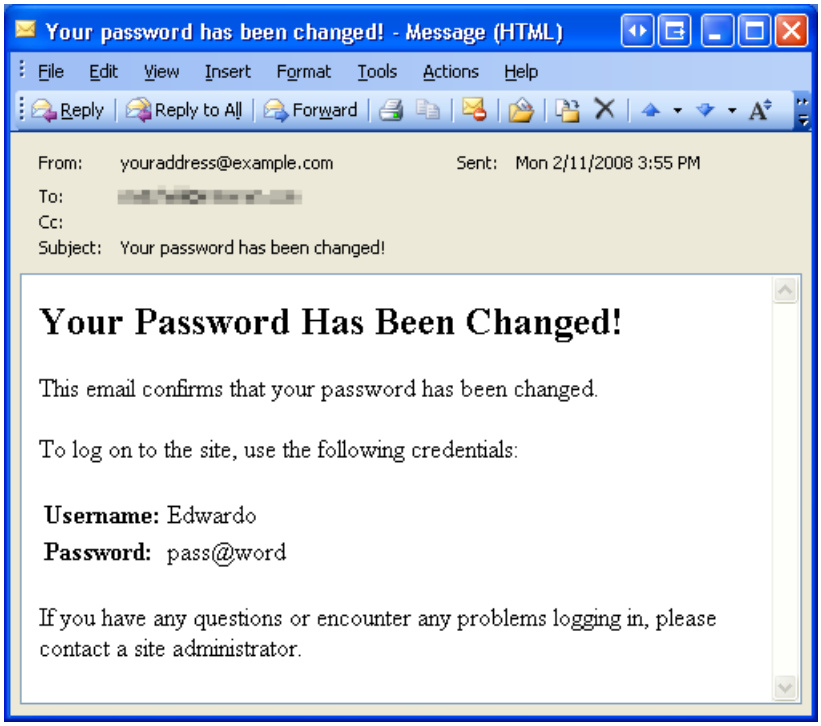


Figure 8: An Email Message Informs the User That Their Password has Changed (Click to view full-size image)

Step 3: Allowing Administrators to Change Users' Passwords

A common feature in applications that support user accounts is the ability for an administrative user to change other users' passwords. Sometimes this functionality is required because the system lacks the ability for users to change their own passwords. In such a case, the only way for a user to recover their forgotten password would be for the administrator to assign them a new password. With the PasswordRecovery and ChangePassword controls, however, administrative users need not busy themselves with changing users' passwords, as users are capable of doing this themselves.

But what if your client insists that administrative users should be able to change other users' passwords? Unfortunately, adding this functionality can be a bit of work. To change a user's password, both the old and new password must be supplied to the `MembershipUser` object's `ChangePassword` method, but an administrator shouldn't have to know a user's password in order to modify it.

One workaround is to first reset the user's password and then change it to the new password using code like the following:

```
Dim usr As MembershipUser = Membership.GetUser(username)
Dim resetPwd As String = usr.ResetPassword()
usr.ChangePassword(resetPwd, newPassword)
```

This code starts by retrieving information about *username*, which is the user whose password the administrator wishes to change. Next, the `ResetPassword` method is invoked, which assigns and the user a new, random password. This randomly-generated password is returned by the method and stored in the variable `resetPwd`. Now that we know the user's password, we can change it via a call to `ChangePassword`.

The problem is that this code only works if the Membership system configuration is set such that `RequiresQuestionAndAnswer` is False. If `RequiresQuestionAndAnswer` is True, as it is with our application, then the `ResetPassword` method needs to be passed the security answer, otherwise it will throw an exception.

If the Membership framework is configured to require a security question and answer, and yet your client insists that administrators be able to change users' passwords, you have three options:

- Throw your hands in the air and tell your client that this is just one thing that cannot be done.
- Set `RequiresQuestionAndAnswer` to False. This results in a less secure application. Imagine that a nefarious user has gained access to another user's email inbox. Perhaps the compromised user has left their desk to go to lunch and didn't lock their workstation, or maybe they accessed their email from a public terminal and didn't sign out. In either case, the nefarious user can visit the `RecoverPassword.aspx` page and enter the user's username. The system will then email the recovered password without prompting for the security answer.
- Bypass the abstraction layer created by the Membership framework and work directly with the SQL Server database. The Membership schema includes a stored procedure named `aspnet_Membership_SetPassword` that sets a user's password and does not require the security answer or old password in order to accomplish its task.

None of these options are particularly appealing, but that's how the life of a developer goes sometimes.

I went ahead and implemented the third approach, writing code that bypasses the `Membership` and `MembershipUser` classes and operates directly against the `SecurityTutorials` database.

NOTE

By working directly with the database, the encapsulation provided by the Membership framework is shattered. This decision ties us to the `SqlMembershipProvider`, making our code less portable. Furthermore, this code may not work as expected in future versions of ASP.NET if the Membership schema changes. This approach is a workaround and, like most workarounds, is not an example of best practices.

The code has some unattractive bits and is quite lengthy. Therefore, I don't want to clutter this tutorial with an in-depth examination of it. If you are interested in learning more, download the code for this tutorial and visit the `~/Administration/ManageUsers.aspx` page. This page, which we created in the preceding tutorial, lists each user. I've updated the GridView to include a link to the `UserInformation.aspx` page, passing the selected user's username through the querystring. The `UserInformation.aspx` page displays information about the selected user and TextBoxes for changing their password (see Figure 9).

After entering the new password, confirming it in the second TextBox, and clicking the Update User Button, a postback ensues and the `aspnet_Membership_SetPassword` stored procedure is invoked, updating the user's password. I encourage those readers interested in this functionality to become more familiar with the code and try extending the functionality to include sending an email to the user whose password was changed.

Figure 9: An Administrator May Change a User's Password (Click to view full-size image)

NOTE

The `UserInformation.aspx` page currently only works if the Membership framework is configured to store passwords in Clear or Hashed format. It lacks the code to encrypt the new password, although you are invited to add this functionality. The way I recommend adding the necessary code is to use a decompiler like Reflector to examine the source code for methods in the .NET Framework; start by examining the `SqlMembershipProvider` class's `ChangePassword` method. This is the technique I used to write the code for creating a hash of the password.

Summary

ASP.NET offers two controls to help users manage their password. The PasswordRecovery control is useful for those who have forgotten their passwords. Depending on the Membership framework's configuration, the user is either emailed their existing password or a new, randomly-generated password. The ChangePassword control enables a user to update their password.

Like the Login and CreateUserWizard controls, the PasswordRecovery and ChangePassword controls render a rich

user interface without having to write a lick of declarative markup or line of code. If the default user interface doesn't meet your needs, you can customize it through a variety of style properties. Alternatively, the controls' interfaces may be converted into templates, for an even finer degree of control. Behind the scenes these controls use the Membership API, invoking the `MembershipUser` object's `ResetPassword` and `ChangePassword` methods.

Happy Programming!

Further Reading

For more information on the topics discussed in this tutorial, refer to the following resources:

- ChangePassword Control QuickStarts
- PasswordRecovery Control QuickStarts
- Sending Email in ASP.NET
- `System.Net.Mail` FAQs

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial include Michael Emmings and Suchi Banerjee. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS NEXT

Unlocking and Approving User Accounts (VB)

2/20/2020 • 17 minutes to read • Edit Online

by Scott Mitchell

Download Code or Download PDF

> This tutorial shows how to build a web page for administrators to manage users' locked out and approved statuses. We will also see how to approve new users only after they have verified their email address.

Introduction

Along with a username, password, and email, each user account has two status fields that dictate whether the user can log into the site: locked out and approved. A user is automatically locked out if they provide invalid credentials a specified number of times within a specified number of minutes (the default settings lock a user out after 5 invalid login attempts within 10 minutes). The approved status is useful in scenarios where some action must transpire before a new user is able to log on to the site. For example, a user might need to first verify their email address or be approved by an administrator before being able to login.

Because a locked out or unapproved user cannot login, it's only natural to wonder how these statuses can be reset. ASP.NET does not include any built-in functionality or Web controls for managing users' locked out and approved statuses, in part because these decisions need to be handled on a site-by-site basis. Some sites might automatically approve all new user accounts (the default behavior). Others have an administrator approve new accounts or do not approve users until they visit a link sent to the email address provided when they signed up. Likewise, some sites may lock out users until an administrator resets their status, while other sites send an email to the locked out user with a URL they can visit to unlock their account.

This tutorial shows how to build a web page for administrators to manage users' locked out and approved statuses. We will also see how to approve new users only after they have verified their email address.

Step 1: Managing Users' Locked Out and Approved Statuses

In the *Building an Interface to Select One User Account from Many* tutorial we constructed a page that listed each user account in a paged, filtered GridView. The grid lists each user's name and email, their approved and locked out statuses, whether they're currently online, and any comments about the user. To manage users' approved and locked out statuses, we could make this grid editable. To change a user's approved status, the administrator would first locate the user account and then edit the corresponding GridView row, checking or unchecking the approved checkbox. Alternatively, we could manage the approved and locked out statuses through a separate ASP.NET page.

For this tutorial let's use two ASP.NET pages: `ManageUsers.aspx` and `UserInformation.aspx`. The idea here is that `ManageUsers.aspx` lists the user accounts in the system, while `UserInformation.aspx` enables the administrator to manage the approved and locked out statuses for a specific user. Our first order of business is to augment the GridView in `ManageUsers.aspx` to include a HyperLinkField, which renders as a column of links. We want each link to point to `UserInformation.aspx?user=UserName`, where *UserName* is the name of the user to edit.

> **NOTE**
>
> If you downloaded the code for the *Recovering and Changing Passwords* tutorial you may have noticed that the `ManageUsers.aspx` page already contains a set of "Manage" links and the `UserInformation.aspx` page provides an interface for changing the selected user's password. I decided not to replicate that functionality in the code associated with this tutorial because it worked by circumventing the Membership API and operating directly with the SQL Server database to change a user's password. This tutorial starts from scratch with the `UserInformation.aspx` page.

Adding "Manage" Links to the `UserAccounts` GridView

Open the `ManageUsers.aspx` page and add a HyperLinkField to the `UserAccounts` GridView. Set the HyperLinkField's `Text` property to "Manage" and its `DataNavigateUrlFields` and `DataNavigateUrlFormatString` properties to `UserName` and "UserInformation.aspx?user={0}", respectively. These settings configure the HyperLinkField such that all of the hyperlinks display the text "Manage", but each link passes in the appropriate *UserName* value into the querystring.

After adding the HyperLinkField to the GridView, take a moment to view the `ManageUsers.aspx` page through a browser. As Figure 1 shows, each GridView row now includes a "Manage" link. The "Manage" link for Bruce points to `UserInformation.aspx?user=Bruce`, whereas the "Manage" link for Dave points to `UserInformation.aspx?user=Dave`.

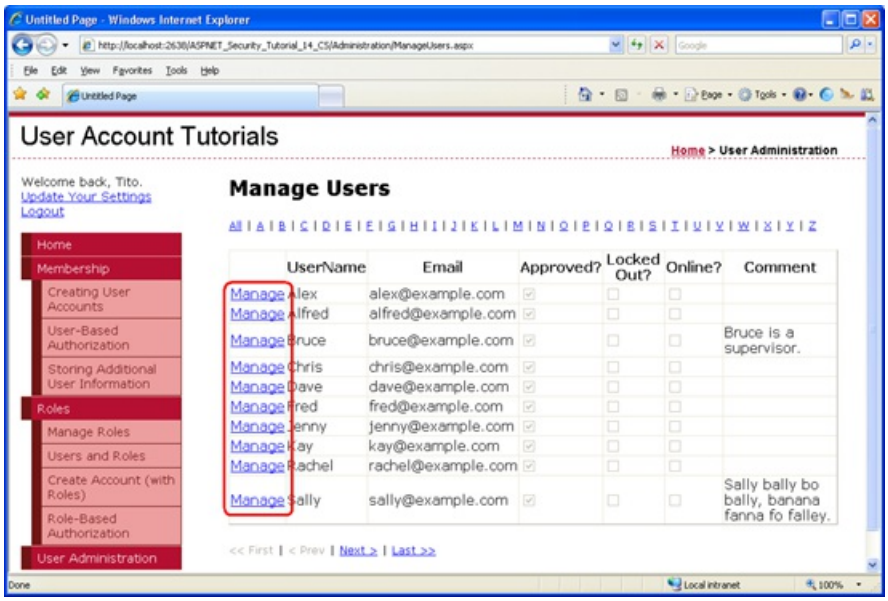


Figure 1: The HyperLinkField Adds a "Manage" Link for Each User Account (Click to view full-size image)

We will create the user interface and code for the `UserInformation.aspx` page in a moment, but first let's talk about how to programmatically change a user's locked out and approved statuses. The `MembershipUser` class has `IsLockedOut` and `IsApproved` properties. The `IsLockedOut` property is read-only. There is no mechanism to programmatically lock out a user; to unlock a user, use the `MembershipUser` class's `UnlockUser` method. The `IsApproved` property is readable and writeable. To save any changes to this property, we need to call the `Membership` class's `UpdateUser` method, passing in the modified `MembershipUser` object.

Because the `IsApproved` property is readable and writeable, a CheckBox control is probably the best user interface element for configuring this property. However, a CheckBox will not work for the `IsLockedOut` property because an administrator cannot lock out a user, she may only unlock a user. A suitable user interface for the `IsLockedOut` property is a Button that, when clicked, unlocks the user account. This Button should only be enabled if the user is locked out.

Creating the `UserInformation.aspx` Page

We are now ready to implement the user interface in `UserInformation.aspx`. Open this page and add the

following Web controls:

- A HyperLink control that, when clicked, returns the administrator to the `ManageUsers.aspx` page.
- A Label Web control for displaying the selected user's name. Set this Label's `ID` to `UserNameLabel` and clear out its Text property.
- A CheckBox control named `IsApproved`. Set its `AutoPostBack` property to `True`.
- A Label control for displaying the user's last locked out date. Name this Label `LastLockedOutDateLabel` and clear out its `Text` property.
- A Button for unlocking the user. Name this Button `UnlockUserButton` and set its `Text` property to "Unlock User".
- A Label control for displaying status messages, such as, "The user's approved status has been updated." Name this control `StatusMessage`, clear out its `Text` property, and set its `CssClass` property to `Important`. (The `Important` CSS class is defined in the `Styles.css` stylesheet file; it displays the corresponding text in a large, red font.)

After adding these controls, the Design view in Visual Studio should look similar to the screen shot in Figure 2.

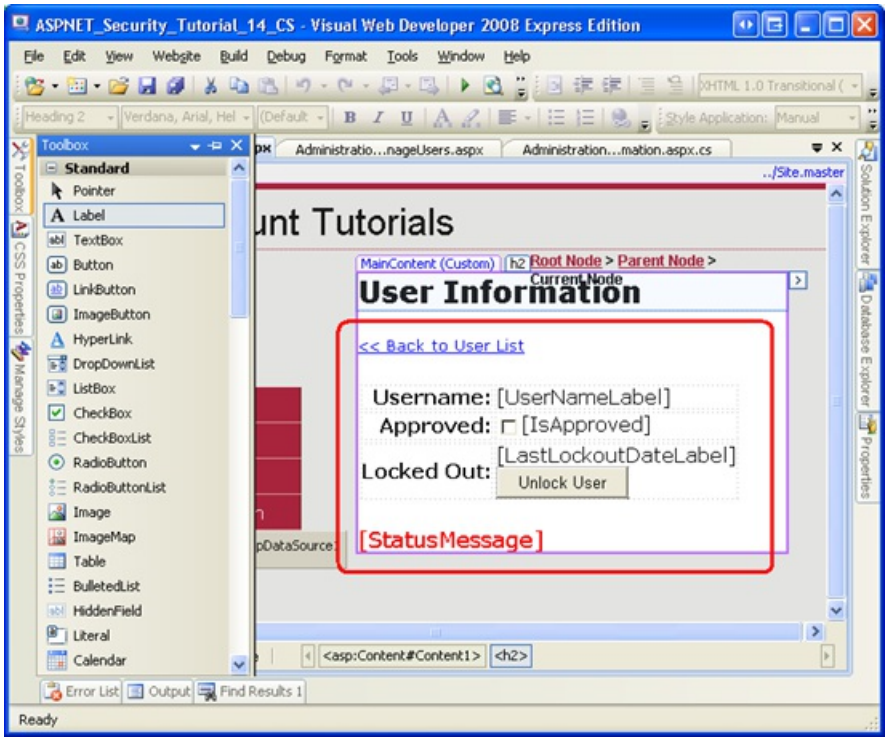


Figure 2: Create the User Interface for `UserInformation.aspx` (Click to view full-size image)

With the user interface complete, our next task is to set the `IsApproved` CheckBox and other controls based on the selected user's information. Create an event handler for the page's `Load` event and add the following code:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
     If Not Page.IsPostBack Then
          ' If querystring value is missing, send the user to ManageUsers.aspx
          Dim userName As String = Request.QueryString("user")
          If String.IsNullOrEmpty(userName) Then
               Response.Redirect("ManageUsers.aspx")
          End If

          ' Get information about this user
          Dim usr As MembershipUser = Membership.GetUser(userName)
          If usr Is Nothing Then
               Response.Redirect("ManageUsers.aspx")

          End If

          UserNameLabel.Text = usr.UserName
          IsApproved.Checked = usr.IsApproved

          If usr.LastLockoutDate.Year < 2000 Then
               LastLockoutDateLabel.Text = String.Empty
          Else
               LastLockoutDateLabel.Text = usr.LastLockoutDate.ToShortDateString()
               UnlockUserButton.Enabled = usr.IsLockedOut
          End If
     End If
End Sub
```

The above code starts by ensuring that this is the first visit to the page and not a subsequent postback. It then reads the username passed through the `user` querystring field and retrieves information about that user account via the `Membership.GetUser(username)` method. If no username was supplied through the querystring, or if the specified user could not be found, the administrator is sent back to the `ManageUsers.aspx` page.

The `MembershipUser` object's `UserName` value is then displayed in the `UserNameLabel` and the `IsApproved` CheckBox is checked based on the `IsApproved` property value.

The `MembershipUser` object's `LastLockoutDate` property returns a `DateTime` value indicating when the user was last locked out. If the user has never been locked out, the value returned depends on the Membership provider. When a new account is created, the `SqlMembershipProvider` sets the `aspnet_Membership` table's `LastLockoutDate` field to `1754-01-01 12:00:00 AM`. The above code displays an empty string in the `LastLockoutDateLabel` if the `LastLockoutDate` property occurs before year 2000; otherwise, the date portion of the `LastLockoutDate` property is displayed in the Label. The `UnlockUserButton`'s `Enabled` property is set to the user's locked out status, meaning that this Button will only be enabled if the user is locked out.

Take a moment to test the `UserInformation.aspx` page through a browser. You will, of course, need to start at `ManageUsers.aspx` and select a user account to manage. Upon arriving at `UserInformation.aspx`, note that the `IsApproved` CheckBox is only checked if the user is approved. If the user has ever been locked out, their last locked out date is displayed. The Unlock User button is enabled only if the user is currently locked out. Checking or unchecking the `IsApproved` CheckBox or clicking the Unlock User button causes a postback, but no modifications are made to the user account because we've yet to create event handlers for these events.

Return to Visual Studio and create event handlers for the `IsApproved` CheckBox's `CheckedChanged` event and the `UnlockUser` Button's `Click` event. In the `CheckedChanged` event handler, set the user's `IsApproved` property to the `Checked` property of the CheckBox and then save the changes via a call to `Membership.UpdateUser`. In the `Click` event handler, simply call the `MembershipUser` object's `UnlockUser` method. In both event handlers, display a suitable message in the `StatusMessage` Label.

```
Protected Sub IsApproved_CheckedChanged(ByVal sender As Object, ByVal e As System.EventArgs) Handles
IsApproved.CheckedChanged
     'Toggle the user's approved status
     Dim userName As String = Request.QueryString("user")
     Dim usr As MembershipUser = Membership.GetUser(userName)
     usr.IsApproved = IsApproved.Checked
     Membership.UpdateUser(usr)

     StatusMessage.Text = "The user's approved status has been updated."

End Sub

Protected Sub UnlockUserButton_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
UnlockUserButton.Click
     'Unlock the user account
     Dim userName As String = Request.QueryString("user")
     Dim usr As MembershipUser = Membership.GetUser(userName)
     usr.UnlockUser()
     UnlockUserButton.Enabled = False

     StatusMessage.Text = "The user account has been unlocked."
End Sub
```

Testing the `UserInformation.aspx` Page

With these event handlers in place, revisit the page and unapproved a user. As Figure 3 shows, you should see a brief message on the page indicating that the user's `IsApproved` property was successfully modified.

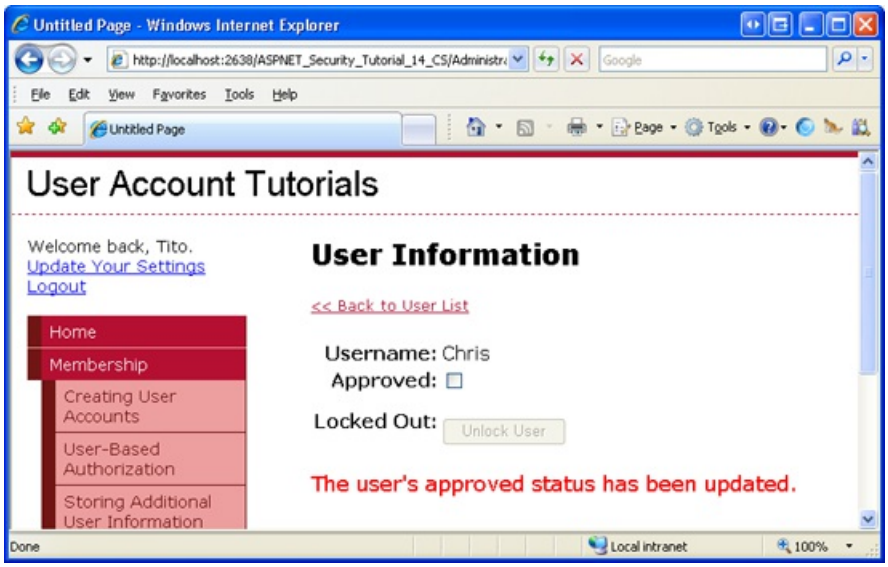


Figure 3: Chris has been Unapproved (Click to view full-size image)

Next, logout and try to login as the user whose account was just unapproved. Because the user is not approved, they cannot login. By default, the Login control displays the same message if the user cannot login, regardless of the reason. But in the *Validating User Credentials Against the Membership User Store* tutorial we looked at enhancing the Login control to display a more appropriate message. As Figure 4 shows, Chris is shown a message explaining that he cannot login because his account is not yet approved.

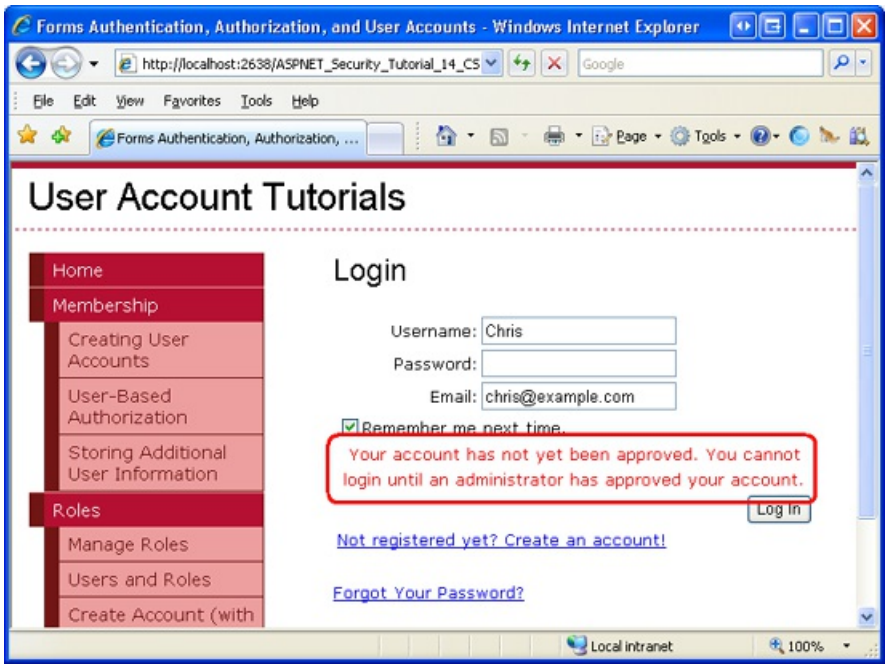


Figure 4: Chris Cannot Login Because His Account is Unapproved (Click to view full-size image)

To test the locked out functionality, attempt to login as an approved user, but use an incorrect password. Repeat this process the necessary number of times until the user's account has been locked out. The Login control was also updated to show a custom message if attempting to login from a locked out account. You know that an account has been locked out once you start seeing the following message at the login page: "Your account has been locked out because of too many invalid login attempts. Please contact the administrator to have your account unlocked."

Return to the `ManageUsers.aspx` page and click the Manage link for the locked out user. As Figure 5 shows, you should see a value in the `LastLockedOutDateLabel` the Unlock User button should be enabled. Click the Unlock User button to unlock the user account. Once you have unlocked the user, they will be able to login again.

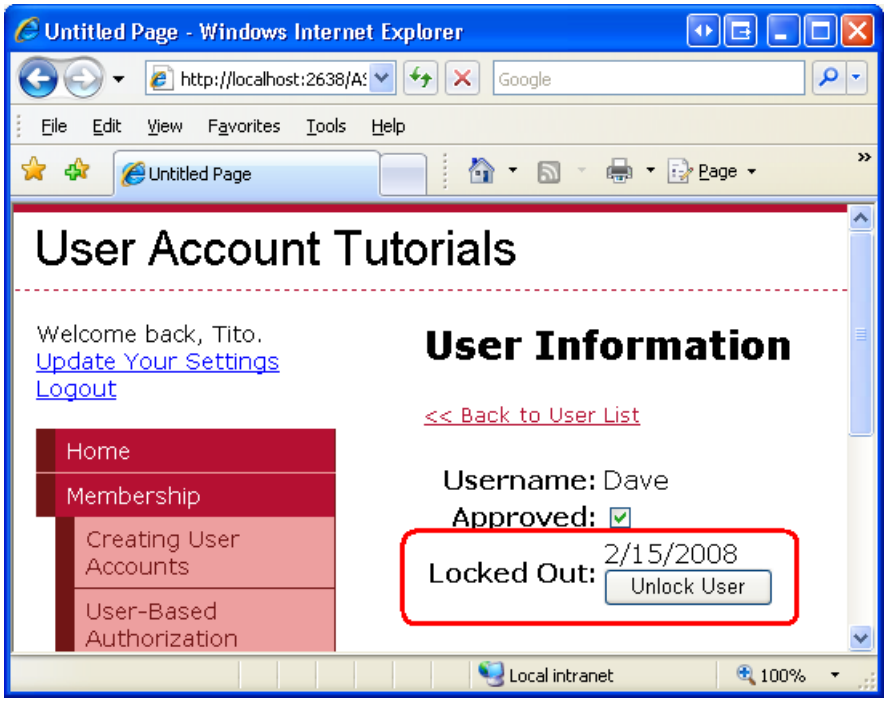


Figure 5: Dave Has Been Locked Out of the System (Click to view full-size image)

Step 2: Specifying New Users' Approved Status

The approved status is useful in scenarios where you want some action to be performed before a new user is able

to login and access the user-specific features of the site. For example, you may be running a private website where all pages, except for the login and signup pages, are accessible only to authenticated users. But what happens if a stranger reaches your website, finds the signup page, and creates an account? To prevent this from happening you could move the signup page to an `Administration` folder, and require that an administrator manually create each account. Alternatively, you could allow anyone to signup, but prohibit site access until an administrator approves the user account.

By default, the CreateUserWizard control approves new accounts. You can configure this behavior using the control's `DisableCreatedUser` property. Set this property to `True` to not approve new user accounts.

> **NOTE**
> By default the CreateUserWizard control automatically logs on the new user account. This behavior is dictated by the control's `LoginCreatedUser` property. Because unapproved users cannot login to the site, when `DisableCreatedUser` is `True` the new user account is not logged into the site, regardless of the value of the `LoginCreatedUser` property.

If you are programmatically creating new user accounts via the `Membership.CreateUser` method, to create an unapproved user account use one of the overloads that accept the new user's `IsApproved` property value as an input parameter.

Step 3: Approving Users By Verifying their Email Address

Many websites that support user accounts do not approve new users until they verify the email address they supplied when registering. This verification process is commonly used to thwart bots, spammers, and other ne'er-do-wells as it requires a unique, verified email address and adds an extra step in the signup process. With this model, when a new user signs up they are sent an email message that includes a link to a verification page. By visiting the link the user has proven that they received the email and, therefore, that the email address provided is valid. The verification page is responsible for approving the user. This may happen automatically, thereby approving any user who reaches this page, or only after the user provides some additional information, such as a CAPTCHA.

To accommodate this workflow, we need to first update the account creation page so that new users are unapproved. Open the `EnhancedCreateUserWizard.aspx` page in the `Membership` folder and set the CreateUserWizard control's `DisableCreatedUser` property to `True`.

Next, we need to configure the CreateUserWizard control to send an email to the new user with instructions on how to verify their account. In particular, we will include a link in the email to the `Verification.aspx` page (which we've yet to create), passing in the new user's `UserId` through the querystring. The `Verification.aspx` page will lookup the specified user and mark them approved.

Sending a Verification Email to New Users

To send an email from the CreateUserWizard control, configure its `MailDefinition` property appropriately. As discussed in the previous tutorial, the ChangePassword and PasswordRecovery controls include a `MailDefinition` property that works in the same manner as the CreateUserWizard control's.

> **NOTE**
> To use the `MailDefinition` property you need to specify mail delivery options in `Web.config`. For more information, refer to Sending Email in ASP.NET.

Start by creating a new email template named `CreateUserWizard.txt` in the `EmailTemplates` folder. Use the following text for the template:

```
Hello <%UserName%>! Welcome aboard.

Your new account is almost ready, but before you can login you must first visit:
<%VerificationUrl%>

Once you have visited the verification URL you will be redirected to the login page.

If you have any problems or questions, please reply to this email.

Thanks!
```

Set the `MailDefinition`'s `BodyFileName` property to "~/EmailTemplates/CreateUserWizard.txt" and its `Subject` property to "Welcome to My Website! Please activate your account."

Note that the `CreateUserWizard.txt` email template includes a `<%VerificationUrl%>` placeholder. This is where the URL for the `Verification.aspx` page will be placed. The CreateUserWizard automatically replaces the `<%UserName%>` and `<%Password%>` placeholders with the new account's username and password, but there is no built-in `<%VerificationUrl%>` placeholder. We need to manually replace it with the appropriate verification URL.

To accomplish this, create an event handler for the CreateUserWizard's `SendingMail` event and add the following code:

```
Protected Sub NewUserWizard_SendingMail(ByVal sender As Object, ByVal e As
System.Web.UI.WebControls.MailMessageEventArgs) Handles NewUserWizard.SendingMail
     'Get the UserId of the just-added user
     Dim newUser As MembershipUser = Membership.GetUser(NewUserWizard.UserName)
     Dim newUserId As Guid = CType(newUser.ProviderUserKey, Guid)

     ' Determine the full verification URL (i.e., http://yoursite.com/Verification.aspx?ID=...)
     Dim urlBase As String = Request.Url.GetLeftPart(UriPartial.Authority) & Request.ApplicationPath
     Dim verifyUrl As String = "/Verification.aspx?ID=" + newUserId.ToString()
     Dim fullUrl As String = urlBase & verifyUrl

     ' Replace <%VerificationUrl%> with the appropriate URL and querystring

     e.Message.Body = e.Message.Body.Replace("<%VerificationUrl%>", fullUrl)
End Sub
```

The `SendingMail` event fires after the `CreatedUser` event, meaning that by the time the above event handler executes the new user account has already been created. We can access the new user's `UserId` value by calling the `Membership.GetUser` method, passing in the `UserName` entered into the CreateUserWizard control. Next, the verification URL is formed. The statement `Request.Url.GetLeftPart(UriPartial.Authority)` returns the `http://yourserver.com` portion of the URL; `Request.ApplicationPath` returns path where the application is rooted. The verification URL is then defined as `Verification.aspx?ID=userId`. These two strings are then concatenated to form the complete URL. Finally, the email message body (`e.Message.Body`) has all occurrences of `<%VerificationUrl%>` replaced with the full URL.

The net effect is that new users are unapproved, meaning that they cannot log into the site. Furthermore, they are automatically sent an email with a link to the verification URL (see Figure 6).

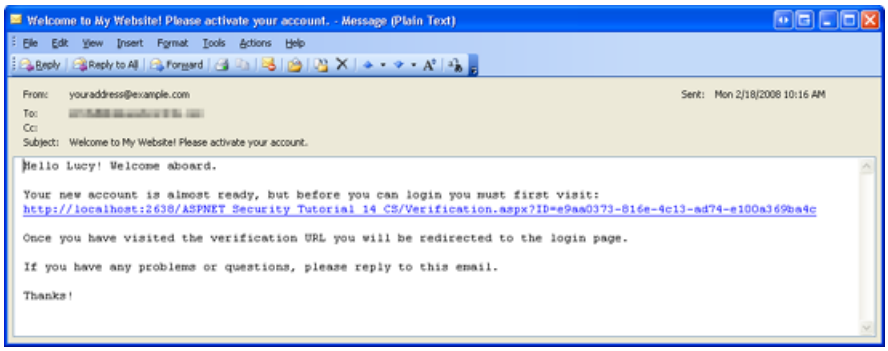


Figure 6: The New User Receives an Email with a Link to the Verification URL (Click to view full-size image)

> **NOTE**
>
> The CreateUserWizard control's default CreateUserWizard step displays a message informing the user their account has been created and displays a Continue button. Clicking this takes the user to the URL specified by the control's `ContinueDestinationPageUrl` property. The CreateUserWizard in `EnhancedCreateUserWizard.aspx` is configured to send new users to the `~/Membership/AdditionalUserInfo.aspx`, which prompts the user for their hometown, homepage URL, and signature. Because this information can only be added by logged on users, it makes sense to update this property to send users back to the site's homepage (`~/Default.aspx`). Moreover, the `EnhancedCreateUserWizard.aspx` page or the CreateUserWizard step should be augmented to inform the user that they have been sent a verification email and their account won't be activated until they follow the instructions in this email. I leave these modifications as an exercise for the reader.

Creating the Verification Page

Our final task is to create the `Verification.aspx` page. Add this page to the root folder, associating it with the `Site.master` master page. As we've done with most of the previous content pages added to the site, remove the Content control that references the `LoginContent` ContentPlaceHolder so that the content page uses the master page's default content.

Add a Label Web control to the `Verification.aspx` page, set its `ID` to `StatusMessage` and clear out its text property. Next, create the `Page_Load` event handler and add the following code:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
     If String.IsNullOrEmpty(Request.QueryString("ID")) Then
          StatusMessage.Text = "The UserId was not included in the querystring..."
     Else
          Dim userId As Guid
          Try
               userId = New Guid(Request.QueryString("ID"))
          Catch
               StatusMessage.Text = "The UserId passed into the querystring is not in the proper format..."

               Exit Sub
          End Try

          Dim usr As MembershipUser = Membership.GetUser(userId)
          If usr Is Nothing Then
               StatusMessage.Text = "User account could not be found..."
          Else
               ' Approve the user
               usr.IsApproved = True
               Membership.UpdateUser(usr)
               StatusMessage.Text = "Your account has been approved. Please <a href=""Login.aspx"">login</a>
to the site."

          End If
     End If
End Sub
```

The bulk of the above code verifies that the UserId supplied through the querystring exists, that it is a valid `Guid` value, and that it references an existing user account. If all of these checks pass, the user account is approved; otherwise, a suitable status message is displayed.

Figure 7 shows the `Verification.aspx` page when visited through a browser.

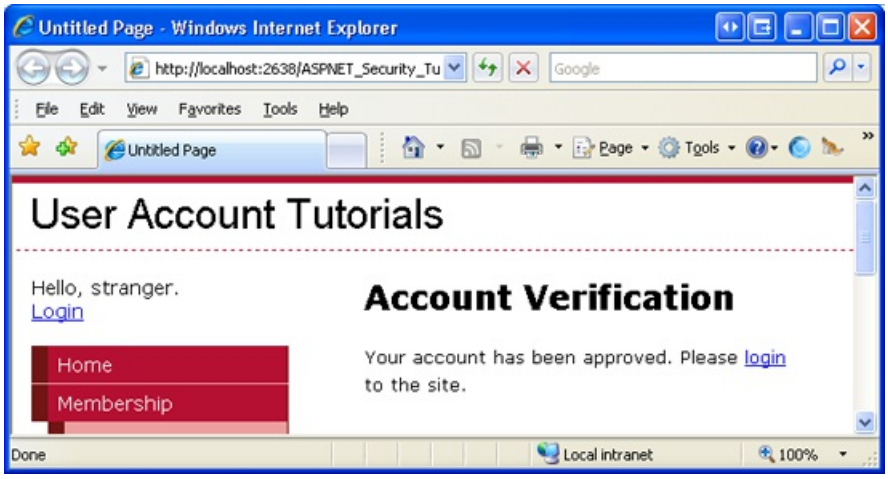


Figure 7: The New User's Account is Now Approved (Click to view full-size image)

Summary

All Membership user accounts have two statuses that determine whether the user can log into the site: `IsLockedOut` and `IsApproved`. Both of these properties must be `True` for the user to login.

The user's locked out status is used as a security measure to reduce the likelihood of a hacker breaking into a site through brute force methods. Specifically, a user is locked out if there are a certain number of invalid login attempts within a certain window of time. These bounds are configurable through the Membership provider settings in `Web.config`.

The approved status is commonly used as a means to prohibit new users from logging in until some action has

transpired. Perhaps the site requires that new accounts first be approved by the administrator or, as we saw in Step 3, by verifying their email address.

Happy Programming!

About the Author

Scott Mitchell, author of multiple ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. Scott can be reached at mitchell@4guysfromrolla.com or via his blog at http://ScottOnWriting.NET.

Special Thanks To...

This tutorial series was reviewed by many helpful reviewers. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com

PREVIOUS

Older Versions - ASP.NET 2.0

2/20/2020 • 2 minutes to read • Edit Online

How to migrate ASP.NET 1.x applications to ASP.NET 2.0.

- Improvements in Visual Studio 2005
- ASP.NET 2.0 Page Model
- Server Controls
- Master Pages
- Data Source Controls
- Data Bound Controls
- Caching
- Membership
- Configuration and Instrumentation
- Profiles, Themes, and Web Parts

Improvements in Visual Studio 2005

4/10/2020 • 21 minutes to read • Edit Online

by Microsoft

> Visual Studio 2005 provides Web application developers with a long list of improvements and enhancements to Web projects.

Visual Studio 2005 provides Web application developers with a long list of improvements and enhancements to Web projects. As powerful as Visual Studio .NET 2002 and 2003 are, there were many complaints in the way that Web projects were handled. Visual Studio 2005 adds a significant number of new features in order to address these complaints. For those who prefer the way that Visual Studio .NET 2003 handled compilation of Web applications, see Web Application Projects.

In this module, well cover improvements in Web project creation, management, and development. In a later module, well cover improvements in building Web projects and deploying them.

FrontPage Server Extensions

Visual Studio .NET 2002 and 2003 required FrontPage Server Extensions on the box in order to create or build Web projects. Developers did have a choice between two different access modes (FrontPage Server Extensions or File access mode), both used FrontPage Server Extensions to perform tasks such as setting the application root in IIS, etc.

Visual Studio 2005 removes the reliance on FrontPage Server Extensions for local projects. Visual Studio 2005 now accesses the IIS metabase directly instead of using the FrontPage Server Extensions. Visual Studio 2005 also adds support for FTP which allows for remote project access without requiring FrontPage Server Extensions.

For those developers who want to use FrontPage Server Extensions in their projects, the option is still available. However, based upon strong feedback from the ASP.NET developer community, it is not a requirement.

> **NOTE**
>
> FrontPage Server Extensions are still required for remote project creation, opening, etc.

ASP.NET Development Server

Visual Studio 2005 ships with a new Web server called ASP.NET Development Server. (This Web server was previously known as Cassini.)

There are several benefits of the ASP.NET Development Server.

- It is now possible for non-Administrators to develop and debug against a Web server.
- The ASP.NET Development Server dynamically maps virtual directories to any location in the file system allowing for flexible project locations.
- Users on Windows XP Professional who are already using IIS will now be able to create new Web applications that will not affect the file or folder structure of their Default Web Site in IIS.

No special configuration is required to take advantage of the ASP.NET Development Server. When a Web project that is hosted on the file system is debugged or browsed, Visual Studio 2005 will automatically start an instance of the ASP.NET Development Server on a random port to service the request.

More information will be covered on the ASP.NET Development Server later in this module.

Improved File Management

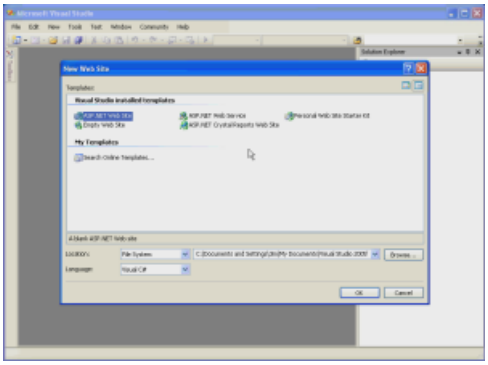
In Visual Studio 2002 and 2003, a project file (.vbproj for VB.NET and .csproj for C#) stored information on all files in the Web application. The Solution Explorer display is based upon the file information in the project file. Because of this, the Solution Explorer would often display inaccurate information in cases where external editors were used. Visual Studio 2002 and 2003 would often overwrite file changes or not display the most recent version of files.

Visual Studio 2005 does away with the project file. Instead, it reads the file and folder information directly from the disk, resulting in an accurate display of the files in your project. Because the References folder in Visual Studio 2002 and 2003 does not represent an actual folder in your Web application, Visual Studio 2005 also removes the References folder from Solution Explorer. To access the references for your project in Visual Studio 2005, you should use the Property pages for the project.

Creating Web Projects

Web developers have many new options available for project creation in Visual Studio 2005. Web sites can now be created anywhere in the file system and can then be debugged or browsed using the new ASP.NET Development Server. Developers can also create new Web sites using FTP.

Click here to view a video walkthrough of creating Web projects in Visual Studio 2005.



Open Full-Screen Video

File System Projects

As you saw in the video walkthrough, you can choose to create Web sites on the file system either on the local machine or on a remote location via a file share. Web sites that are created on the file system are browsed and debugged using the ASP.NET Development Server.

> **NOTE**
>
> The ASP.NET Development Server may cause some confusion for customers. If a Web project is created on the file system in IISs directory structure (i.e. c:/inetpub/wwwroot), the Web site will still be browsed via the ASP.NET Development Server when launched from within Visual Studio 2005. Therefore, any IIS configuration (i.e. authentication methods) is not applicable.

The default web project also removes a lot of the overhead by only includes a Default.aspx page, default.cs file, and an App/_Data folder. The web.config and special folders (i.e. app/_code) are added as they are needed. Your web project only includes the files and folders that you need.

HTTP Projects

HTTP projects can either be projects that are created on a local IIS Web site or on a remote Web site. The default project location is `http://localhost`. If you click the Browse button, there are two HTTP options: Local IIS and

Remote Site. The main difference in these two options is the method in which the web site information is displayed in the Choose Location dialog and in how the files are copied to the Web server.

The Local IIS option reads the site information from the metabase on the local machine and files are copied using the file system. The Remote Site option uses the FrontPage Server Extensions and the site information and files are copied using HTTP and FrontPage Server Extensions RPC calls.

> **NOTE**
>
> The vs###/_tmp.htm file and get/_aspx/_ver.aspx are no longer used to determine version information.

The default HTTP option is Local IIS. This option reads the IIS Metabase to determine which sites are available and the location in which to create content. You can select a different folder or virtual directory by selecting it in the tree view. You can also create a new virtual directory, mark folders as applications, as well as delete existing virtual directories from this dialog box.

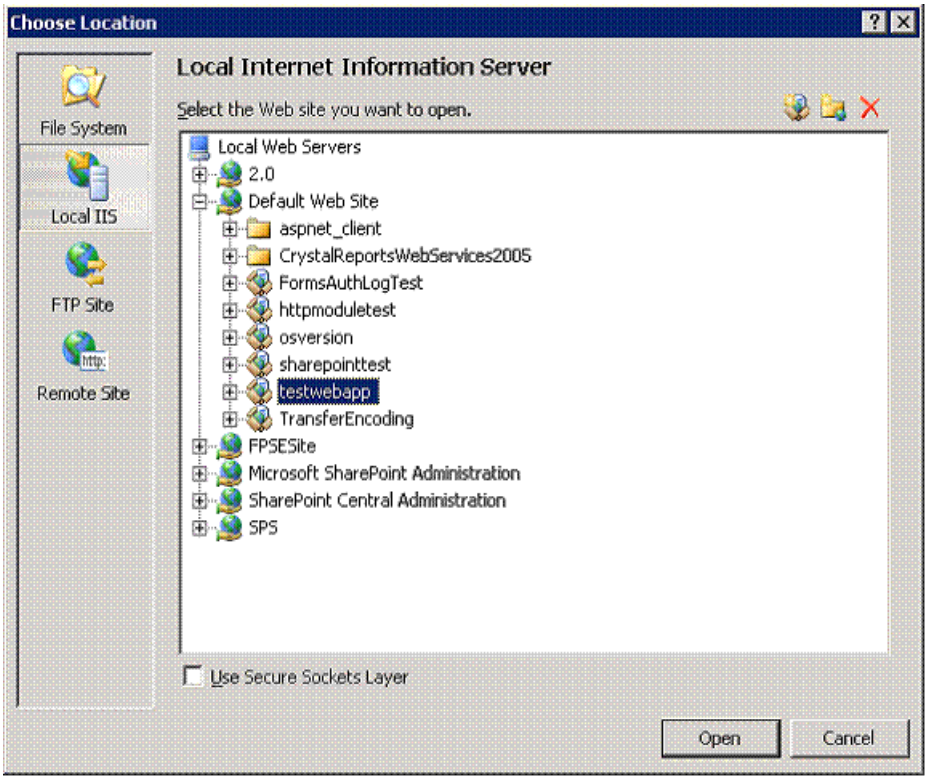


Figure 1: The Choose Location Dialog

Unlike in earlier versions of Visual Studio, if you check the **Use Secure Sockets Layer** checkbox and the SSL certificate does not match the URL you are browsing, you will be presented with a Security Alert dialog asking you if you would like to proceed. Using Visual Studio .NET 2003, if the certificate was not a matching one, creating the project would fail.

Figure 2: Security Alert Regarding SSL Certificate

Note on Host Headers

If you are creating a Web application on a site bound to a specific IP, you will need to ensure that a host header is configured. Otherwise, Visual Studio will create the site at `http://localhost`, but the IP address will not resolve correctly when the site is browsed or debugged from within the IDE.

If you select the Remote Site option, the dialog changes to allow you to enter the destination URL for the new Web site. This URL must be on a server that has the FrontPage Server Extensions enabled. If you want to work with your local Web server using the FrontPage Server Extensions, you can use the Remote Site option and specify a local URL.

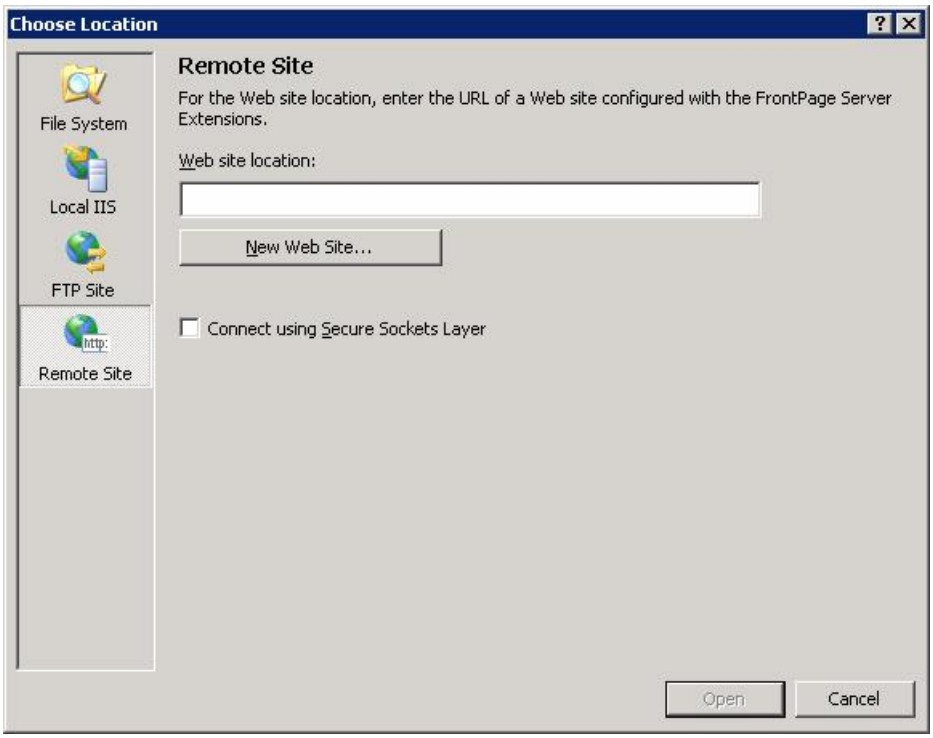


Figure 3: Creating a Web Site on a Remote Server

When creating an application on a remote site via SSL, if the SSL certificate does not match, the confirmation dialog is slightly different than the dialog displayed when using the Local IIS option.

Figure 4: The Remote Site Security Alert

FTP

Visual Studio 2005 introduces the option to create Web sites via FTP. When you use this option, the IDE creates the files locally in the users temp folder and then uses FTP to move the files to the FTP location.

> **NOTE**
>
> The temp folder location is c:/Documents and Settings/<User>/Local Settings/Temp/VWDWebCache/<Server>/_<application name>

When using the FTP option, you will be presented with a Choose Location dialog. You enter the required FTP connection information into this dialog as shown below.

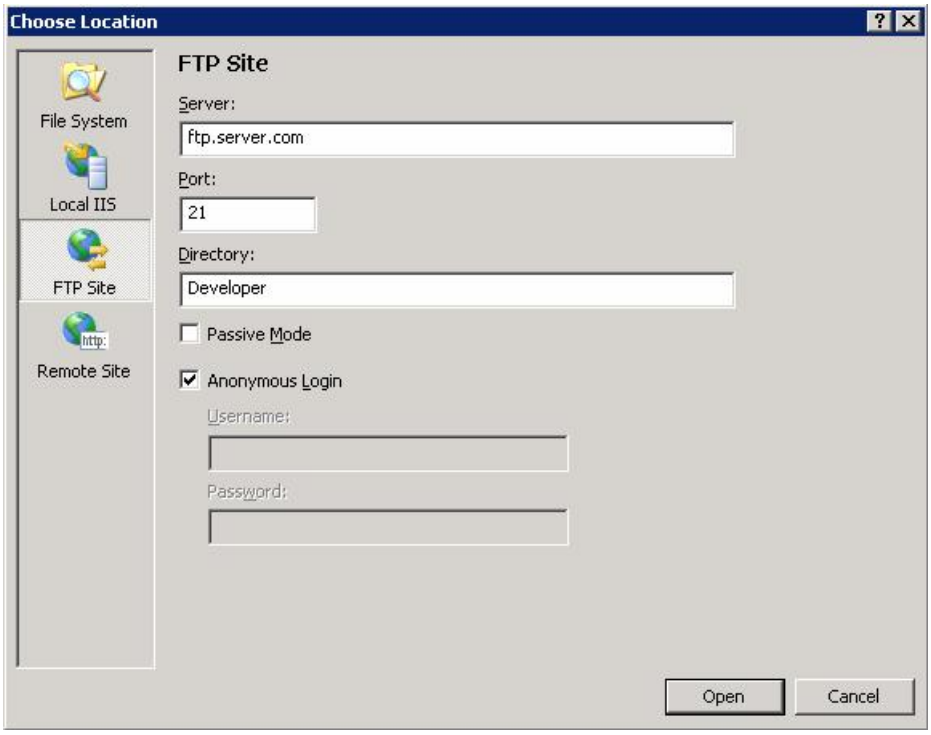


Figure 5: The Choose Location Dialog for FTP

Lab: Setup FTP site and create a project

The following steps configure the FTP site so that a user has a location that only they can upload to via FTP.

Install the FTP Service

1. Open Add Remove Programs, select Add/Remove Windows Components
2. Select Internet Information Services (Application Server on Windows 2003) and click **Details**.
3. Check **File Transfer Protocol (FTP) Service** and click **OK**.

4. Click **Next** to install the FTP service.

Create a New Folder for Content

1. In Windows Explorer, create a new folder called **User1** inside of c:/inetpub/wwwroot.

Configure folders and permissions on folders.

1. Open the Internet Information Services snap-in from Administrative Tools. You will now have an FTP Sites folder under the computer name node.
2. Expand **FTP Sites**.
3. Right-click the **Default FTP Site**, select **New**, then **Virtual Directory**, then click **Next**.
4. Enter **User1** for the virtual directory name and click **Next**.
5. Enter **c:/inetpub/wwwroot/User1** for the path and click **Next**.
6. Click **Next** and then **Finish** to complete the wizard.
7. Right-click the **User1** virtual directory under Default FTP Site and select **Properties**.
8. Check the **Write** checkbox and click **OK** to close the dialog.
9. Right-click **Default FTP Site** and select **Properties**.
10. On the **Security Accounts** tab, uncheck **Allow Anonymous Connections**.
11. Click **Yes** in the dialog asking if you want to continue.
12. Click **OK** to close the dialog.
13. Expand the **Default Web Site** under the **Web Sites** node.
14. Right-click the **User1** directory and select **Properties**
15. In the **Application Settings** section, click **Create** to mark the folder as an application.
16. Click **OK** to close the dialog.
17. Close the Internet Information Services snap-in.

Create web project

1. Open Visual Studio 2005.
2. From the **File** menu, select **New Web Site**.
3. In the **Location** dropdown, select **FTP**.
4. Click **Browse**.
5. Enter **localhost** in the **Server** textbox.
6. Enter **User1** in the Directory textbox.
7. Click **Open**. The FTP location will be entered into the New Web Site dialog.
8. Click **OK**.
9. Uncheck **Anonymous log on** in the FTP Log On dialog, enter your credentials, and click **OK**.
10. What is the URL for the project? (The URL for the project will be displayed in Solution Explorer.)
11. From the **Build** menu, select **Build Web Site** or **Build Solution**.
12. Right-click on Default.aspx in Solution Explorer and select **View in Browser**.
13. In the Web Site URL Required dialog, enter `http://localhost/user1` for the URL and click **OK**.

> **NOTE**
>
> If you get a error indicating an inability to load the type /_Default, make sure that you are running ASP.NET 2.0 on your Web site and not an earlier version. You can do that from the ASP.NET tab in Internet Information Services.

Opening Web Projects

Opening Web projects is similar to creating projects. The following sections call out areas to keep an eye out for while working within the IDE. It also covers working with Web projects using HTTP and FTP.

To open a Web project, select Open Web Site from the File menu. You will be prompted with the same Choose Location dialog covered previously and you have the same four options available to you: File System, Local IIS, FTP, and Remote Site.

File System

As indicated previously in this module, Visual Studio no longer uses a project file. Therefore, if you choose to open a Web site from the file system, you actually have the option of choosing any folder that you wish, even if the folder you choose was not created as a Web project initially in Visual Studio. For example, you can choose to open the My Documents folder as a Web site and Visual Studio will happily open it and display your files as shown below.

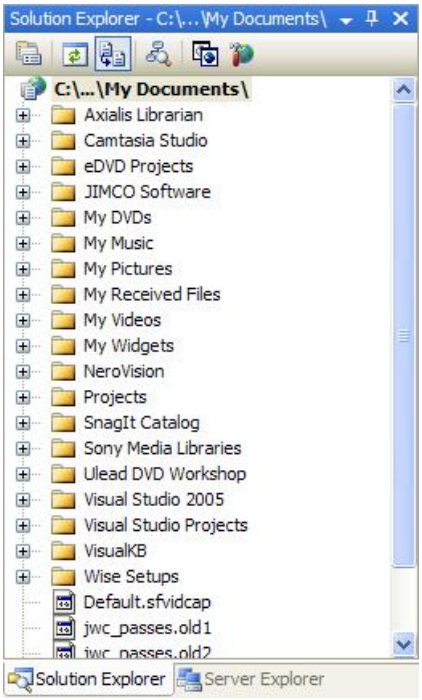


Figure 6: *My Documents* Opened As a Web Site

Because Visual Studio only creates additional files and folders when necessary, no additional files or folders are added to the location you open. A side-effect of this architecture is that it prevents you from nesting Web sites on the file system. For example, consider the following directory structure.

Web project at C:/MyWebSite

Another web project at C:/MyWebSite/Nested

When you open the Web site at c:/MyWebSite, the Nested folder will appear as a sub-folder of that application.

HTTP

When opening Web sites via HTTP, settings are read either from the IIS metabase (Local IIS) or using FrontPage Server Extensions (Remote Site.) If there are nested web applications, these are displayed as well with an icon identifying them as an application. If you are familiar with working with web applications in FrontPage, the behavior in Visual Studio 2005 is similar.

Even though Visual Studio will display an icon for applications that are nested beneath the application that is currently opened within the IDE, it will not allow you to expand them to see their content. You can, however, double-click on them to open them. When you do, you will be presented with a dialog prompting you to either open the web application (and replace the currently open solution) or add the Web application to your current solution.

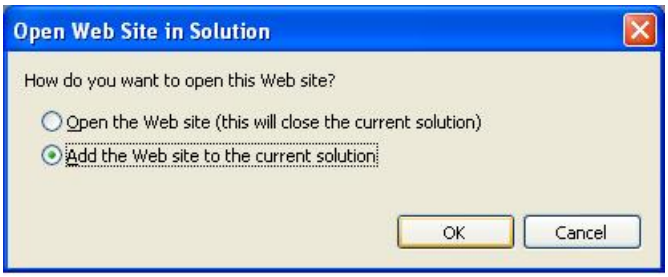


Figure 7: Double-clicking a nested application icon presents you with this dialog

FTP Site

When you open a site via FTP, the files are all copied locally to your temp folder. The full path for the local storage location is displayed in the Properties pane for the project and is created using the following format.

C:/Documents and Settings/<User>/Local Settings/Temp/VWDWebCache/<Server>/_<application name>

When using FTP, Visual Studio will need to specify the base URL for your project so that you can browse it as shown below. If you do not specify a base URL, Visual Studio will ask you for it the first time you attempt to browse a page in the Web site.

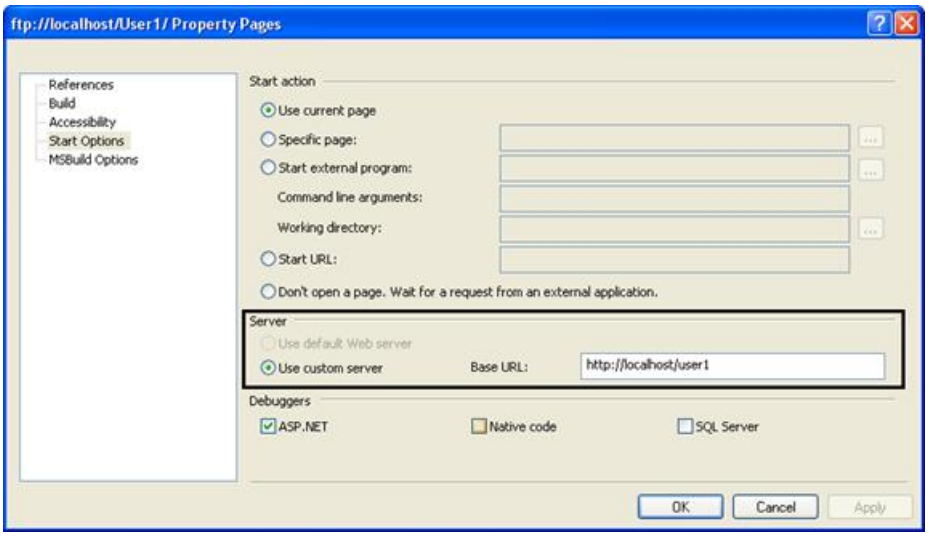


Figure 8: Specifying a Base URL for FTP Sites

Improvements in Compilation

Working with Web applications in Visual Studio 2005 is noticeably faster than previous versions. This is due in no small part to the changes in compilation architecture.

In Visual Studio 2002 and 2003, Web applications were compiled into one primary assembly residing in the /bin folder. In Visual Studio 2005, an App/_Code folder was added. Classes and other non-UI code are added to the App/_Code folder. When Visual Studio builds the project, all files in the App/_Code folder are compiled into a single App/_Code.dll file. The result of this change is that subsequent builds are much faster than in previous versions.

> **NOTE**
>
> The MSBuild command line utility can also be used to build ASP.NET Web applications. That tool will be covered in module 9.

Another compilation enhancement is the new Build Page option on the Build menu. This feature allows a developer to rebuild only the current page (along with, of course, and dependencies) so that changes can be compiled more

quickly. Because C# does not offer background compilation for purposes of updating IntelliSense, etc., they will benefit immensely from this feature because it will allow for IntelliSense to be updated quickly by simply rebuilding a single page.

The Build properties for a project allow you to configure the type of build that occurs before the startup page is executed. Developers can choose to only build the current page so that Visual Studio can start debugging applications more quickly after code changes.

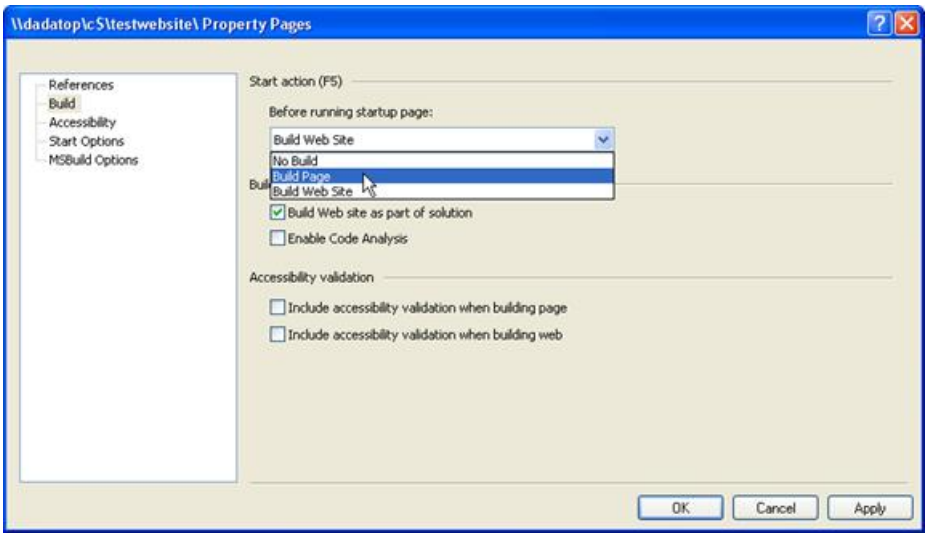
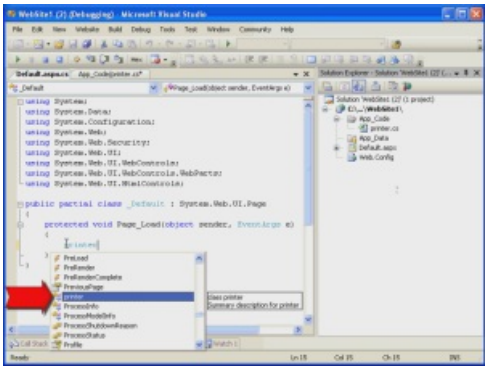


Figure 9: The Build Page Start Action

Another great enhancement to Visual Studio and the ASP.NET architecture is in the area of edit and continue. In Visual Studio 2005, developers can start debugging a project and make code changes on the project without detaching the debugger. In fact, you can literally start debugging a project, add a new class, add code to that class, add code to your page that creates a new instance of that class and execute a method of the class, all without detaching the debugger. Executing the new code is literally as easy as refreshing the browser!

Click here to see a video walkthrough of the edit and continue feature in Visual Studio 2005.

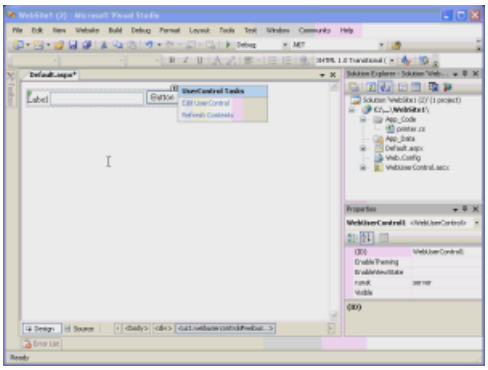


Open Full-Screen Video

The robust edit and continue functionality in ASP.NET 2.0 and Visual Studio 2005 is due to an architectural change for ASP.NET applications. In ASP.NET 1.x, applications created in Visual Studio 2002/2003 were compiled into a primary assembly that was stored in the /bin folder. All classes, pages, etc. for the application were compiled into that one DLL. Then at runtime, ASP.NET would compile all of the controls, markup, and ASP.NET code within pages and copy those DLLs into the ASP.NET temporary folder.

In Visual Studio 2005 using ASP.NET 2.0, the two compilation models outline above (one for Visual Studio and one for ASP.NET at runtime) have been merged into one common compilation model. That means that all compilation issues are now caught during the development stage instead of at runtime. It also allows for designer and IntelliSense support for features such as user controls and master pages.

Click here to see a video walkthrough of designer support for user controls.



Open Full-Screen Video

> **NOTE**
>
> When a user control is removed from a page, the @Register directive remains in the markup and should be removed manually in order to avoid parser errors if the user control is deleted from the Web site.

Another improvement in the Visual Studio compilation model is the Publish Web Site feature. Because the Publish feature precompiles the Web site, developers can enjoy the added performance of not having to compile anything on demand. It also precompiles all source code in the App/_Code folder into a DLL so that no source code has to be deployed.

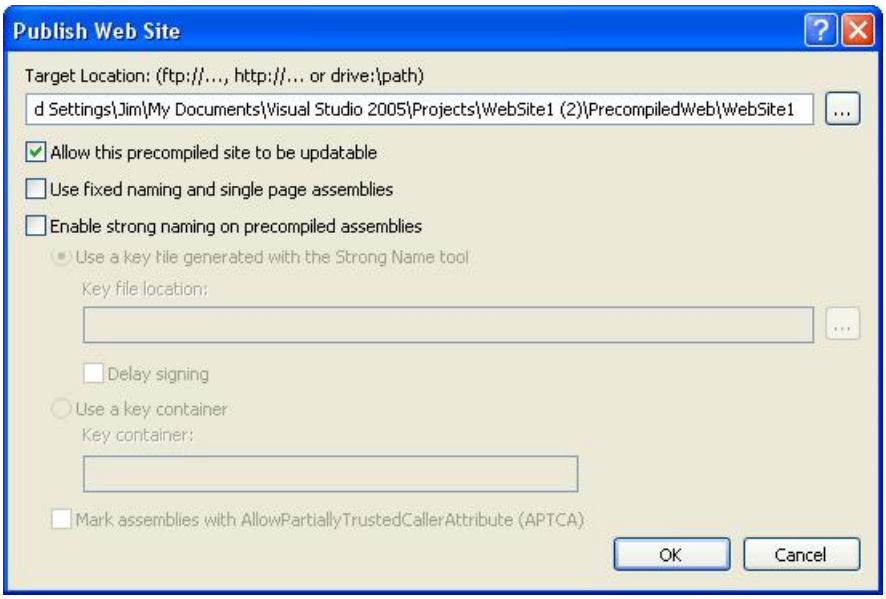


Figure 10: The Publish Web Site Dialog

> **NOTE**
>
> The aspnet/_compile.exe utility can also be used to pre-compile an ASP.NET Web application. That tool will be covered in module 9.

When you Publish a Web site, the precompiled files are stored in the Temporary ASP.NET Files folder as shown below. Files with a *.compiled* file extension are XML files that define dependencies for particular DLLs. Any Webform or user controls are compiled into random DLLs that begin with *App/Web/*.

If you leave the *Allow this precompiled site to be updatable* checkbox checked, markup inside of your Webforms and user controls will not be pre-compiled into a DLL allowing you to make changes after deployment. If you would prefer to lock down the markup so that changes to the deployed content are not allowed, uncheck this box.

The *Use fixed naming and single page assemblies* checkbox allows you to disable batch compilation so that each

page is compiled into a fixed-named assembly. Leaving this box unchecked allows you to take advantage of batch compilation.

The *Enable strong naming on precompiled assemblies* checkbox allows you to strong-name your precompiled assemblies.

> **NOTE**
>
> In ASP.NET 1.x, strong-named assemblies had to be installed into the Global Assembly Cache (GAC). In ASP.NET 2.0, you are not required to install strong-named assemblies into the GAC.

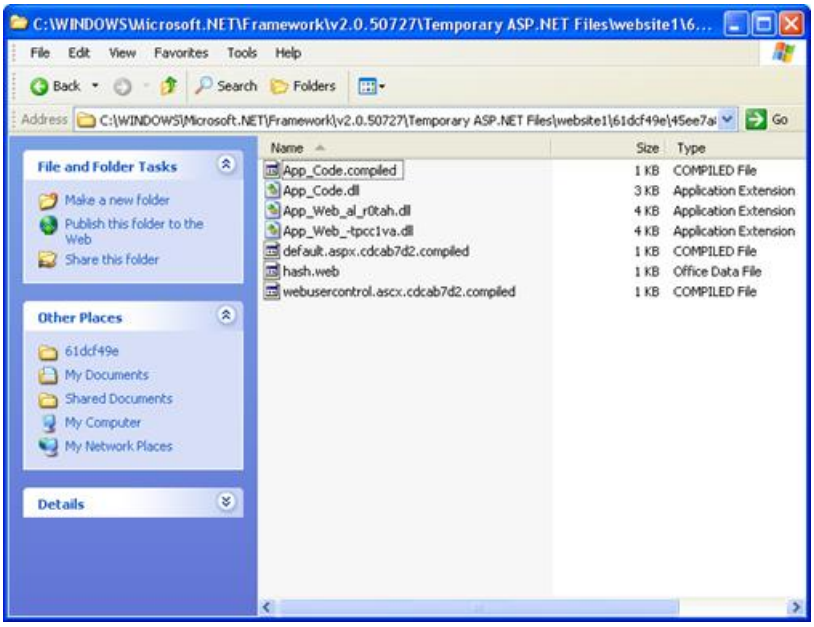


Figure 11: An ASP.NET Applications Pre-Compiled Files

> **NOTE**
>
> In the application above, there was no web.config file. If there had been, it would have been called *PrecompiledApp.config* after the Publish Web site process.

Improvements in Deployment

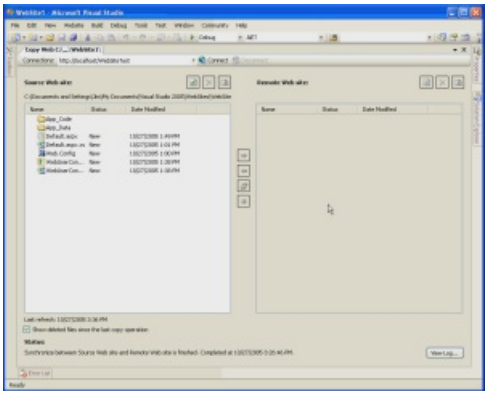
As with Visual Studio 2002 and 2003, Visual Studio 2005 offers a Copy Project feature. However, the feature has been beefed up in Visual Studio 2005 and is now called Copy Web Site.

The Copy Web Site dialog is split into a left frame and a right frame. The left frame is called the Source Web Site and the right frame is called the Remote Web Site. One thing that may confuse some developers is that the site displayed in the right frame is not necessarily a remote site. It could be a site on the local file system or on the local instance of IIS. Additionally, the site displayed in the left frame is not necessarily the source Web site because the dialog allows you to publish from the remote Web site *to* the source Web site.

If you are copying a project to a remote Web site, that site must have the FrontPage Server Extensions installed on it. If it does not, you will need to connect using FTP. On the other hand, if you are copying a project to the local IIS instance, FrontPage Server Extensions are not required.

> **NOTE**
> If you try to create a new Web site on the local IIS instance and the FrontPage 2002 Server Extensions are installed, you will get an error message telling you that creating Web sites is not supported on a SharePoint server. In that case, you have the option of installing the FrontPage 2000 Server Extensions or removing the FrontPage Server Extensions.

Click here for a video walkthrough of the Copy Web Site feature.



Open Full-Screen Video

Improvements in Debugging

There are four key improvements in debugging in Visual Studio 2005.

- Debugging locally as a non-administrator is possible out of the box.
- The Debug attribute for the Compilation element is now false by default.
- Remote debugging setup and configuration is easier than before.
- You can now debug a Web site opened via an FTP location.

Debugging as a Non-Administrator

The addition of the ASP.NET Development Server allows non-administrators to easily debug ASP.NET applications right out of the box. When an ASP.NET application running on the local file system is debugged, Visual Studio launches the ASP.NET Development Server under the context of the logged-on user. That user can then debug that application without any additional configuration.

Debug is False by Default

In ASP.NET 1.x, the *debug* attribute in the *compilation* element of the web.config file was set to *true* by default. It has always been recommended that developers set this attribute to *false* before deploying an application to production, but because most developers don't fully understand the consequences of leaving the debug attribute set to true, they simply left it as-is.

The most severe problem with having the debug attribute set to true is that it disables ASP.NETs batch compilation model. Therefore, each page is compiled into a separate DLL. If a Web application consists of thousands of pages (not unheard of by any means), that means several thousand small DLLs will be created by that application. While these DLLs are small in size, they are not loaded into any particular location in memory. Therefore, they cause fragmentation in system memory and can contribute to OutOfMemoryException occurrences.

In ASP.NET 2.0, the debug attribute is set to false by default. As you have already seen, when a developer debugs an ASP.NET application in Visual Studio 2005, they are prompted to add a web.config file with debugging enabled. Doing so incurs the same drawbacks that were present in ASP.NET 1.x, but now the developer is clearly warned that the attribute should be reset to false before moving the application to production.

Remote Debugging Setup and Configuration

In Visual Studio 2002/2003, remote debugging relied on the Machine Debug Manager (mdm.exe) and the vs7jit.exe process. Because of that, troubleshooting remote debugging problems was often a black box for customers and it was often not much better for PSS.

Visual Studio 2005 removes the reliance on the mdm.exe and vs7jit.exe processes. Instead, it now uses the Remote Debug Monitor service (msvsmon.exe.)

The requirement for debugging in Visual Studio 2005 remotely is quite simple. You need to run msvsmon.exe on the remote server prior to debugging. You can install the Remote Debug Monitor from the Visual Studio CD or you can simply run msvsmon.exe from a share without installing anything at all on the Web server.

When you run msvsmon.exe, it is likely that it will complain about ports being blocked for remote debugging. Fortunately, you can easily unblock the ports from right within the warning dialog as shown below.

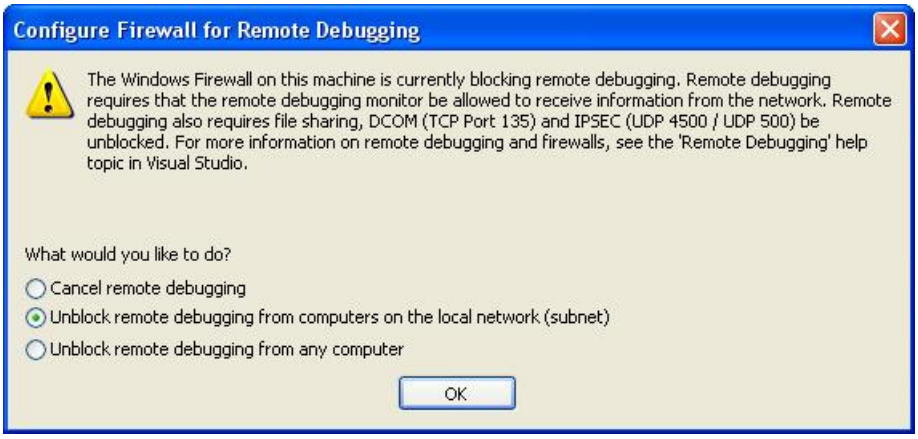


Figure 12: Notification that Windows Firewall is Blocking Remote Debugging

Once you have unblocked the ports necessary for debugging, you will see the Remote Debugging Monitor as shown below. From this interface, you can monitor connections and change debugging permissions easily.

Figure 13: The Remote Debugging Monitor

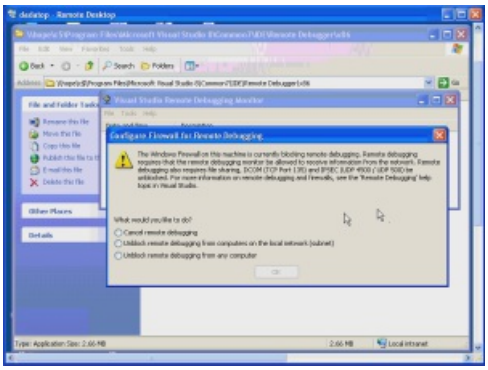
It is also possible to remotely debug a Web application opened via FTP. The steps are the same as those previously covered. However, you will need to specify a base URL for browsing the FTP project as outlined earlier in this module.

Lab 2

Remote Debugging with Visual Studio 2005

This lab will walk you through remote debugging with Visual Studio 2005.

Click here for a video walkthrough of this lab.



Open Full-Screen Video

This lab requires you to have two machines, one running Visual Studio 2005 and the other running IIS 5 or greater.

1. Open Visual Studio 2005 and create a new Web site on the remote server.

> **NOTE**
>
> You can create the Web site on a remote IIS instance or via FTP.

1. From the remote Web server, locate msvsmon.exe on the development machine using a UNC path and execute it.
   The default location for msvsmon.exe is //server/c$/Program Files/Microsoft Visual Studio 8/Common7/IDE/Remote Debugger/x86.
2. If prompted to unblock ports for remote debugging, do so.
3. From the development machine, open the code-behind for Default.aspx and set a breakpoint in the Page/_Load method.
4. Start debugging from the development machine.

You should hit the breakpoint as expected.

ASP.NET Development Server

As we've already discussed, Visual Studio 2005 ships with a Web server called the ASP.NET Development Server. (The ASP.NET Development Server is sometimes referred to as Cassini.) This Web server is a convenient means to browse and debug Web applications running on the file system.

The ASP.NET Development Server is a restricted Web server. It does not allow remote connections, it does not allow any requests from any user other than the user who started the Web server. It also does not have the capability of serving ASP pages. Only ASP.NET resources and HTML resources (including images, CSS files, etc.) are served.

The ASP.NET Development Server can be launched via the command line by running the WebDev.WebServer.exe file located at c:/Windows/Microsoft.NET/Framework/v2.0.////*. The following dialog displays the parameters that are available.

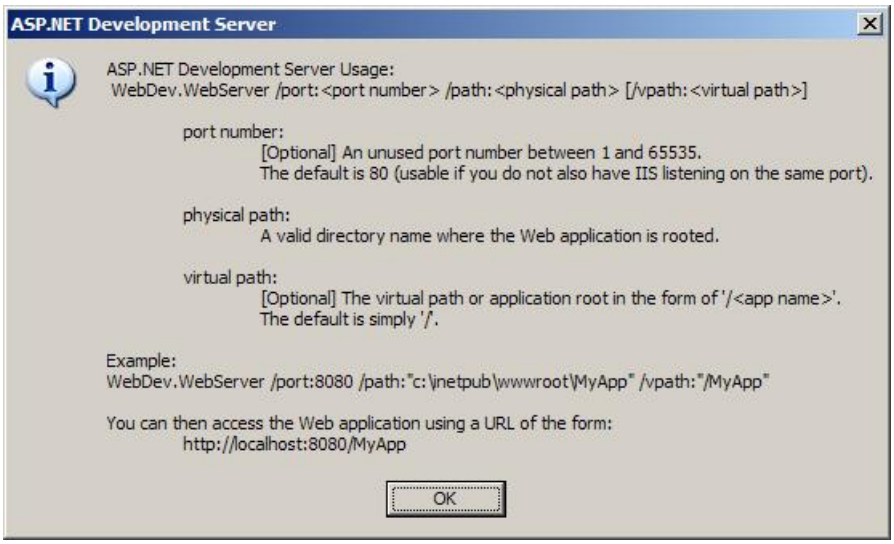


Figure 14

NOTE

The ASP.NET Development Server is not supported when launched explicitly via the command line.

The ASP.NET 2.0 Page Model

4/10/2020 • 26 minutes to read • Edit Online

by Microsoft

> In ASP.NET 1.x, developers had a choice between an inline code model and a code-behind code model. Code-behind could be implemented using either the Src attribute or the CodeBehind attribute of the @Page directive. In ASP.NET 2.0, developers still have a choice between inline code and code-behind, but there have been significant enhancements to the code-behind model.

In ASP.NET 1.x, developers had a choice between an inline code model and a code-behind code model. Code-behind could be implemented using either the Src attribute or the CodeBehind attribute of the @Page directive. In ASP.NET 2.0, developers still have a choice between inline code and code-behind, but there have been significant enhancements to the code-behind model.

Improvements in the Code-Behind Model

In order to fully understand the changes in the code-behind model in ASP.NET 2.0, its best to quickly review the model as it existed in ASP.NET 1.x.

The Code-Behind Model in ASP.NET 1.x

In ASP.NET 1.x, the code-behind model consisted of an ASPX file (the Webform) and a code-behind file containing programming code. The two files were connected using the @Page directive in the ASPX file. Each control on the ASPX page had a corresponding declaration in the code-behind file as an instance variable. The code-behind file also contained code for event binding and generated code necessary for the Visual Studio designer. This model worked fairly well, but because every ASP.NET element in the ASPX page required corresponding code in the code-behind file, there was no true separation of code and content. For example, if a designer added a new server control to an ASPX file outside of the Visual Studio IDE, the application would break due to the absence of a declaration for that control in the code-behind file.

The Code-Behind Model in ASP.NET 2.0

ASP.NET 2.0 greatly improves upon this model. In ASP.NET 2.0, code-behind is implemented using the new *partial classes* provided in ASP.NET 2.0. The code-behind class in ASP.NET 2.0 is defined as a partial class meaning that it contains only part of the class definition. The remaining part of the class definition is dynamically generated by ASP.NET 2.0 using the ASPX page at runtime or when the Web site is precompiled. The link between the code-behind file and the ASPX page is still established using the @ Page directive. However, instead of a CodeBehind or Src attribute, ASP.NET 2.0 now uses the CodeFile attribute. The Inherits attribute is also used to specify the class name for the page.

A typical @ Page directive might look like this:

```
<%@Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>
```

A typical class definition in an ASP.NET 2.0 code-behind file might look like this:

```
public partial class _Default : System.Web.UI.Page
```

> **NOTE**
> C# and Visual Basic are the only managed languages that currently support partial classes. Therefore, developers using J# will not be able to use the code-behind model in ASP.NET 2.0.

The new model enhances the code-behind model because developers will now have code files that contain only the code that they have created. It also provides for a true separation of code and content because there are no instance variable declarations in the code-behind file.

> **NOTE**
> Because the partial class for the ASPX page is where event binding takes place, Visual Basic developers can realize a slight performance increase by using the Handles keyword in code-behind to bind events. C# has no equivalent keyword.

New @ Page Directive Attributes

ASP.NET 2.0 adds many new attributes to the @ Page directive. The following attributes are new in ASP.NET 2.0.

Async

The Async attribute allows you to configure page to be executed asynchronously. Well cover asynchronous pages later in this module.

AsyncTimeout

Specified the timeout for asynchronous pages. The default is 45 seconds.

CodeFile

The CodeFile attribute is the replacement for the CodeBehind attribute in Visual Studio 2002/2003.

CodeFileBaseClass

The CodeFileBaseClass attribute is used in cases where you want multiple pages to derive from a single base class. Because of the implementation of partial classes in ASP.NET, without this attribute, a base class that uses shared common fields to reference controls declared in an ASPX page would not work properly because ASP.NETs compilation engine will automatically create new members based on controls in the page. Therefore, if you want a common base class for two or more pages in ASP.NET, you will need to define specify your base class in the CodeFileBaseClass attribute and then derive each pages class from that base class. The CodeFile attribute is also required when this attribute is used.

CompilationMode

This attribute allows you to set the CompilationMode property of the ASPX page. The CompilationMode property is an enumeration containing the values **Always**, **Auto**, and **Never**. The default is **Always**. The **Auto** setting will prevent ASP.NET from dynamically compiling the page if possible. Excluding pages from dynamic compilation increases performance. However, if a page that is excluded contains that code that must be compiled, an error will be thrown when the page is browsed.

EnableEventValidation

This attribute specifies whether or not postback and callback events are validated. When this is enabled, arguments to postback or callback events are checked to ensure that they originated from the server control that originally

rendered them.

EnableTheming

This attribute specifies whether or not ASP.NET themes are used on a page. The default is **false**. ASP.NET themes are covered in Module 10.

LinePragmas

This attribute specifies whether line pragmas should be added during compilation. Line pragmas are options used by debuggers to mark specific sections of code.

MaintainScrollPositionOnPostback

This attribute specifies whether or not JavaScript is injected into the page in order to maintain scroll position between postbacks. This attribute is **false** by default.

When this attribute is **true**, ASP.NET will add a <script> block on postback that looks like this:

```
<script src="/website/WebResource.axd?d=jBAvpwrdOM_V_Xzeox989A2 &t=632653133849531250" type="text/javascript">
</script>
```

Note that the src for this script block is WebResource.axd. This resource is not a physical path. When this script is requested, ASP.NET dynamically builds the script.

MasterPageFile

This attribute specifies the master page file for the current page. The path can be relative or absolute. Master pages are covered in Module 4.

StyleSheetTheme

This attribute allows you to override user-interface appearance properties defined by an ASP.NET 2.0 theme. Themes are covered in Module 10.

Theme

Specifies the theme for the page. If a value is not specified for the StyleSheetTheme attribute, the Theme attribute overrides all styles applied to controls on the page.

Title

Sets the title for the page. The value specified here will appear in the <title> element of the rendered page.

ViewStateEncryptionMode

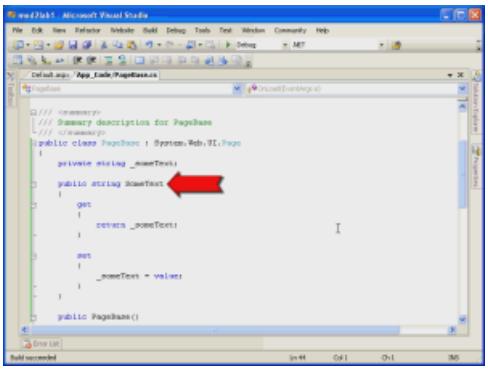
Sets the value for the ViewStateEncryptionMode enumeration. The available values are **Always**, **Auto**, and **Never**. The default value is **Auto**. When this attribute is set to a value of **Auto**, viewstate is encrypted is a control requests it by calling the **RegisterRequiresViewStateEncryption** method.

Setting Public Property Values via the @ Page Directive

Another new capability of the @ Page directive in ASP.NET 2.0 is the ability to set the initial value of public properties of a base class. Suppose, for example, that you have a public property called **SomeText** in your base class and you d like it to be initialized to **Hello** when a page is loaded. You can accomplish this by simply setting the value in the @ Page directive like so:

```
<%@Page Language="C#" SomeText="Hello!" Inherits="PageBase" %>
```

The **SomeText** attribute of the @ Page directive sets the initial value of the SomeText property in the base class to *Hello!*. The video below is a walkthrough of setting the initial value of a public property in a base class using the @ Page directive.



Open Full-Screen Video

New Public Properties of the Page Class

The following public properties are new in ASP.NET 2.0.

AppRelativeTemplateSourceDirectory

Returns the application-relative path to the page or control. For example, for a page located at http://app/folder/page.aspx, the property returns ~/folder/.

AppRelativeVirtualPath

Returns the relative virtual directory path to the page or control. For example for a page located at http://app/folder/page.aspx, the property returns ~/folder/page.aspx.

AsyncTimeout

Gets or sets the timeout used for asynchronous page handling. (Asynchronous pages will be covered later in this module.)

ClientQueryString

A read-only property that returns the query string portion of the requested URL. This value is URL encoded. You can use the UrlDecode method of the HttpServerUtility class to decode it.

ClientScript

This property returns a ClientScriptManager object that can be used to manage ASP.NETs emission of client-side script. (The ClientScriptManager class is covered later in this module.)

EnableEventValidation

This property controls whether or not event validation is enabled for postback and callback events. When enabled, arguments to postback or callback events are verified to ensure that they originated from the server control that originally rendered them.

EnableTheming

This property gets or sets a Boolean that specifies whether or not an ASP.NET 2.0 theme applies to the page.

Form

This property returns the HTML form on the ASPX page as an HtmlForm object.

This property returns a reference to an HtmlHead object that contains the page header. You can use the returned HtmlHead object to get/set style sheets, Meta tags, etc.

IdSeparator

This read-only property gets the character that is used to separate control identifiers when ASP.NET is building a unique ID for controls on a page. It is not intended to be used directly from your code.

IsAsync

This property allows for asynchronous pages. Asynchronous pages are discussed later in this module.

IsCallback

This read-only property returns **true** if the page is the result of a call back. Call backs are discussed later in this module.

IsCrossPagePostBack

This read-only property returns **true** if the page is part of a cross-page postback. Cross-page postbacks are covered later in this module.

Items

Returns a reference to an IDictionary instance that contains all objects stored in the pages context. You can add items to this IDictionary object and they will be available to you throughout the lifetime of the context.

MaintainScrollPositionOnPostBack

This property controls whether or not ASP.NET emits JavaScript that maintains the pages scroll position in the browser after a postback occurs. (Details of this property were discussed earlier in this module.)

Master

This read-only property returns a reference to the MasterPage instance for a page to which a master page has been applied.

MasterPageFile

Gets or sets the master page filename for the page. This property can only be set in the PreInit method.

MaxPageStateFieldLength

This property gets or sets the maximum length for the pages state in bytes. If the property is set to a positive number, the pages view state will be broken up into multiple hidden fields so that it doesnt exceed the number of bytes specified. If the property is a negative number, the view state will not be broken into chunks.

PageAdapter

Returns a reference to the PageAdapter object that modifies the page for the requesting browser.

PreviousPage

Returns a reference to the previous page in cases of a Server.Transfer or a cross-page postback.

SkinID

Specifies the ASP.NET 2.0 skin to apply to the page.

StyleSheetTheme

This property gets or sets the style sheet that is applied to a page.

TemplateControl

Returns a reference to the containing control for the page.

Theme

Gets or sets the name of the ASP.NET 2.0 theme applied to the page. This value must be set prior to the PreInit method.

Title

This property gets or sets the title for the page as obtained from the pages header.

ViewStateEncryptionMode

Gets or sets the ViewStateEncryptionMode of the page. See a detailed discussion of this property earlier in this module.

New Protected Properties of the Page Class

The following are the new protected properties of the Page class in ASP.NET 2.0.

Adapter

Returns a reference to the ControlAdapter that renders the page on the device that requested it.

AsyncMode

This property indicates whether or not the page is processed asynchronously. It is intended for use by the runtime and not directly in code.

ClientIDSeparator

This property returns the character used as a separator when creating unique client IDs for controls. It is intended for use by the runtime and not directly in code.

PageStatePersister

This property returns the PageStatePersister object for the page. This property is primarily used by ASP.NET control developers.

UniqueFilePathSuffix

This property returns a unique suffix that is appended to the file path for caching browsers. The default value is __ufps= and a 6-digit number.

New Public Methods for the Page Class

The following public methods are new to the Page class in ASP.NET 2.0.

AddOnPreRenderCompleteAsync

This method registers event handler delegates for asynchronous page execution. Asynchronous pages are discussed later in this module.

ApplyStyleSheetSkin

Applies the properties in a pages style sheet to the page.

ExecuteRegisteredAsyncTasks

This method beings an asynchronous task.

GetValidators

Returns a collection of validators for the specified validation group or the default validation group if none is specified.

RegisterAsyncTask

This method registers a new async task. Asynchronous pages are covered later in this module.

RegisterRequiresControlState

This method tells ASP.NET that the pages control state must be persisted.

RegisterRequiresViewStateEncryption

This method tells ASP.NET that the pages viewstate requires encryption.

ResolveClientUrl

Returns a relative URL that can be used for client requests for images, etc.

SetFocus

This method will set the focus to the control that is specified when the page is initially loaded.

UnregisterRequiresControlState

This method will unregister the control that is passed to it as no longer requiring control state persistence.

Changes to the Page Lifecycle

The page lifecycle in ASP.NET 2.0 hasn't changed dramatically, but there are some new methods that you should be aware of. The ASP.NET 2.0 page lifecycle is outlined below.

PreInit (New in ASP.NET 2.0)

The PreInit event is the earliest stage in the lifecycle that a developer can access. The addition of this event makes it possible to programmatically change ASP.NET 2.0 themes, master pages, access properties for an ASP.NET 2.0 profile, etc. If you are in a postback state, its important to realize that Viewstate has not yet been applied to controls at this point in the lifecycle. Therefore, if a developer changes a property of a control at this stage, it will likely be overwritten later in the pages lifecycle.

Init

The Init event has not changed from ASP.NET 1.x. This is where you would want to read or initialize properties of controls on your page. At this stage, master pages, themes, etc. are already applied to the page.

InitComplete (New in 2.0)

The InitComplete event is called at the end of the pages initialization stage. At this point in the lifecycle, you can access controls on the page, but their state has not yet been populated.

PreLoad (New in 2.0)

This event is called after all postback data has been applied and just prior to Page_Load.

Load

The Load event has not changed from ASP.NET 1.x.

LoadComplete (New in 2.0)

The LoadComplete event is the last event in the pages load stage. At this stage, all postback and viewstate data has been applied to the page.

PreRender

If you would like for viewstate to be properly maintained for controls that are added to the page dynamically, the PreRender event is the last opportunity to add them.

PreRenderComplete (New in 2.0)

At the PreRenderComplete stage, all controls have been added to the page and the page is ready to be rendered. The PreRenderComplete event is the last event raised before the pages viewstate is saved.

SaveStateComplete (New in 2.0)

The SaveStateComplete event is called immediately after all page viewstate and control state has been saved. This is the last event before the page is actually rendered to the browser.

Render

The Render method has not changed since ASP.NET 1.x. This is where the HtmlTextWriter is initialized and the page is rendered to the browser.

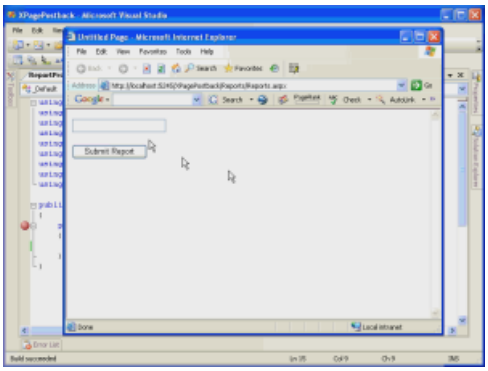
Cross-Page Postback in ASP.NET 2.0

In ASP.NET 1.x, postbacks were required to post to the same page. Cross-page postbacks were not allowed. ASP.NET 2.0 adds the ability to post back to a different page via the IButtonControl interface. Any control that implements the new IButtonControl interface (Button, LinkButton, and ImageButton in addition to third-party custom controls) can take advantage of this new functionality via the use of the PostBackUrl attribute. The following code shows a Button control that posts back to a second page.

```
<asp:Button ID="SubmitReport" PostBackUrl="~/Default.aspx" runat="server" Text="Submit Report" />
```

When the page is posted back, the Page that initiates the postback is accessible via the PreviousPage property on the second page. This functionality is implemented via the new WebForm_DoPostBackWithOptions client-side function that ASP.NET 2.0 renders to the page when a control posts back to a different page. This JavaScript function is provided by the new WebResource.axd handler which emits script to the client.

The video below is a walkthrough of a cross-page postback.



Open Full-Screen Video

More Details on Cross-Page Postbacks

Viewstate

You may have asked yourself already about what happens to the viewstate from the first page in a cross-page postback scenario. After all, any control that does not implement IPostBackDataHandler will persist its state via viewstate, so to have access to the properties of that control on the second page of a cross-page postback, you must have access to the viewstate for the page. ASP.NET 2.0 takes care of this scenario using a new hidden field in the second page called __PREVIOUSPAGE. The __PREVIOUSPAGE form field contains the viewstate for the first page so that you can have access to the properties of all controls in the second page.

Circumventing FindControl

In the video walkthrough of a cross-page postback, I used the FindControl method to get a reference to the TextBox control on the first page. That method works well for that purpose, but FindControl is expensive and it requires writing additional code. Fortunately, ASP.NET 2.0 provides an alternative to FindControl for this purpose that will work in many scenarios. The PreviousPageType directive allows you to have a strongly-typed reference to the previous page by using either the TypeName or the VirtualPath attribute. The TypeName attribute allows you to specify the type of the previous page while the VirtualPath attribute allows you to refer to the previous page using a virtual path. After you've set the PreviousPageType directive, you must then expose the controls, etc. to which you want to allow access using public properties.

Lab 1 Cross-Page Postback

In this lab, you will create an application that uses the new cross-page postback functionality of ASP.NET 2.0.

1. Open Visual Studio 2005 and create a new ASP.NET Web site.

2. Add a new Webform called page2.aspx.
3. Open the Default.aspx in Design view and add a Button control and a TextBox control.
   a. Give the Button control an ID of **SubmitButton** and the TextBox control an ID of **UserName**.
   b. Set the PostBackUrl property of the Button to page2.aspx.
4. Open page2.aspx in Source view.
5. Add a @ PreviousPageType directive as shown below:
6. Add the following code to the Page_Load of page2.aspx's code-behind:

```
Response.Write(PreviousPage.UserName.Text);
```

7. Build the project by clicking on Build on the Build menu.
8. Add the following code to the code-behind for Default.aspx:

```
public TextBox txtUserName {
    get { return this.UserName; }
}
```

9. Change the Page_Load in page2.aspx to the following:

```
Response.Write(PreviousPage.txtUserName.Text);
```

10. Build the project.
11. Run the project.
12. Enter your name in the TextBox and click the button.
13. What is the result?

Asynchronous Pages in ASP.NET 2.0

Many contention problems in ASP.NET are caused by latency of external calls (such as Web service or database calls), file IO latency, etc. When a request is made against an ASP.NET application, ASP.NET uses one of its worker threads to service that request. That request owns that thread until the request is complete and the response has been sent. ASP.NET 2.0 seeks to resolve latency issues with these types of issues by adding the capability to execute pages asynchronously. That means that a worker thread can start the request and then hand off additional execution to another thread, thereby returning to the available thread pool quickly. When the file IO, database call, etc. has completed, a new thread is obtained from the thread pool to finish the request.

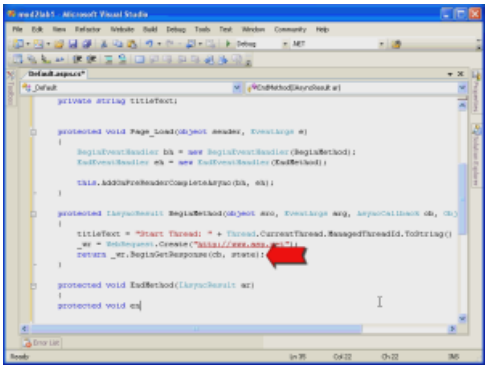
The first step in making a page execute asynchronously is to set the **Async** attribute of the page directive like so:

```
<%@ Page Async="true" %>
```

This attribute tells ASP.NET to implement the IHttpAsyncHandler for the page.

The next step is to call the AddOnPreRenderCompleteAsync method at a point in the lifecycle of the page prior to PreRender. (This method is typically called in Page_Load.) The AddOnPreRenderCompleteAsync method takes two parameters; a BeginEventHandler and an EndEventHandler. The BeginEventHandler returns an IAsyncResult which is then passed as a parameter to the EndEventHandler.

The video below is a walkthrough of an asynchronous page request.



Open Full-Screen Video

> **NOTE**
>
> An async page does not render to the browser until the EndEventHandler has completed. No doubt but that some developers will think of async requests as being similar to async callbacks. It's important to realize that they are not. The benefit to asynchronous requests is that the first worker thread can be returned to the thread pool to service new requests, thereby reducing contention due to being IO bound, etc.

Script Callbacks in ASP.NET 2.0

Web developers have always looked for ways to prevent the flickering associated with a callback. In ASP.NET 1.x, SmartNavigation was the most common method for avoiding flickering, but SmartNavigation caused problems for some developers because of the complexity of its implementation on the client. ASP.NET 2.0 addresses this issue with script callbacks. Script callbacks utilize XMLHttp to make requests against the Web server via JavaScript. The XMLHttp request returns XML data that can then be manipulated via the browser's DOM. XMLHttp code is hidden from the user by the new WebResource.axd handler.

There are several steps that are necessary in order to configure a script callback in ASP.NET 2.0.

Step 1 : Implement the ICallbackEventHandler Interface

In order for ASP.NET to recognize your page as participating in a script callback, you must implement the ICallbackEventHandler interface. You can do this in your code-behind file like so:

```
public partial class _Default : System.Web.UI.Page, ICallbackEventHandler
```

You can also do this using the @ Implements directive like so:

```
<%@ Implements Interface="System.Web.UI.ICallbackEventHandler" %>
```

You would typically use the @ Implements directive when using inline ASP.NET code.

Step 2 : Call GetCallbackEventReference

As mentioned previously, the XMLHttp call is encapsulated in the WebResource.axd handler. When your page is rendered, ASP.NET will add a call to WebForm_DoCallback, a client script that is provided by WebResource.axd. The WebForm_DoCallback function replaces the __doPostBack function for a callback. Remember that __doPostBack programmatically submits the form on the page. In a callback scenario, you want to prevent a postback, so __doPostBack will not suffice.

> **NOTE**
> __doPostBack is still rendered to the page in a client script callback scenario. However, it's not used for the callback.

The arguments for the WebForm_DoCallback client-side function are provided via the server-side function GetCallbackEventReference which would normally be called in Page_Load. A typical call to GetCallbackEventReference might look like this:

```
// Set up the JavaScript callback string cbRef = cm.GetCallbackEventReference(this,
"document.getElementById('ddlCompany').value", "ShowCompanyName", "null", true);
```

> **NOTE**
> In this case, cm is an instance of ClientScriptManager. The ClientScriptManager class will be covered later in this module.

There are several overloaded versions of GetCallbackEventReference. In this case, the arguments are as follows:

`this`

A reference to the control where GetCallbackEventReference is being called. In this case, it's the page itself.

```
document.getElementById('ddlCompany').value
```

A string argument that will be passed from the client-side code to the server-side event. In this case, Im passing the value of a dropdown called ddlCompany.

`ShowCompanyName`

The name of the client-side function that will accept the return value (as string) from the server-side callback event. This function will only be called when the server-side callback is successful. Therefore, for the sake of robustness, it is generally recommended to use the overloaded version of GetCallbackEventReference that takes an additional string argument specifying the name of a client-side function to execute in the event of an error.

`null`

A string representing a client-side function that it initiated before the callback to the server. In this case, there is no such script, so the argument is null.

`true`

A Boolean specifying whether or not to conduct the callback asynchronously.

The call to WebForm_DoCallback on the client will pass these arguments. Therefore, when this page is rendered on the client, that code will look like so:

```
WebForm_DoCallback('__Page',document.getElementById('ddlCompany').value, ShowCompanyName,null,null,true)
```

Notice that the signature of the function on the client is a bit different. The client-side function passes 5 strings and a Boolean. The additional string (which is null in the above example) contains the client-side function that will handle any errors from the server-side callback.

Step 3 : Hook the Client-Side Control Event

Notice that the return value of GetCallbackEventReference above was assigned to a string variable. That string is

used to hook a client-side event for the control that initiates the callback. In this example, the callback is initiated by a dropdown on the page, so I want to hook the *OnChange* event.

To hook the client-side event, simply add a handler to the client-side markup as follows:

```
// Hook the JavaScript function to the onchange event of the dropdown ddlCompany.Attributes["onchange"] =
String.Format("javascript:{0}", cbRef);
```

Recall that *cbRef* is the return value from the call to GetCallbackEventReference. It contains the call to WebForm_DoCallback that was shown above.

Step 4 : Register the Client-Side Script

Recall that the call to GetCallbackEventReference specified that a client-side script called **ShowCompanyName** would be executed when the server-side callback succeeds. That script needs to be added to the page using a ClientScriptManager instance. (The ClientScriptManager class will be discussed later in this module.) You do that like so:

```
System.Text.StringBuilder clientScript = new System.Text.StringBuilder(""); ClientScriptManager cm =
Page.ClientScript; // Create the client script clientScript.Append("function ShowCompanyName(companyName)");
clientScript.Append("{"); clientScript.Append("document.getElementById('CoClicked').innerHTML = \"You chose \"
+ companyName + \".\";"); clientScript.Append("}"); cm.RegisterClientScriptBlock(this.GetType(), "showCo",
clientScript.ToString(), true);
```

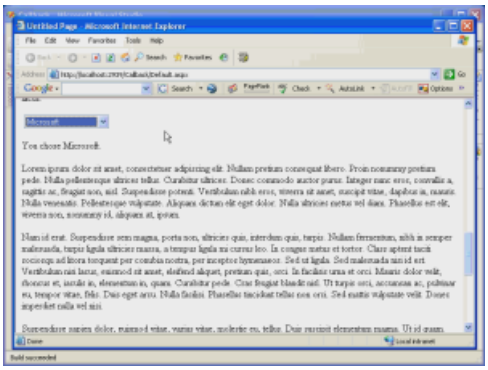
Step 5 : Call the Methods of the ICallbackEventHandler Interface

The ICallbackEventHandler contains two methods that you need to implement in your code. They are **RaiseCallbackEvent** and **GetCallbackEvent**.

RaiseCallbackEvent takes a string as an argument and returns nothing. The string argument is passed from the client-side call to WebForm_DoCallback. In this case, that value is the *value* attribute of the dropdown called ddlCompany. Your server-side code should be placed in the RaiseCallbackEvent method. For example, if your callback is making a WebRequest against an external resource, that code should be placed in RaiseCallbackEvent.

GetCallbackEvent is responsible for processing the return of the callback to the client. It takes no arguments and returns a string. The string that it returns will be passed as an argument to the client-side function, in this case *ShowCompanyName*.

Once you have completed the above steps, you are ready to perform a script callback in ASP.NET 2.0.



Open Full-Screen Video

Script callbacks in ASP.NET are supported in any browser that supports making XMLHttp calls. That includes all of the modern browsers in use today. Internet Explorer uses the XMLHttp ActiveX object while other modern browsers (including the upcoming IE 7) use an intrinsic XMLHttp object. To programmatically determine if a

browser supports callbacks, you can use the **Request.Browser.SupportCallback** property. This property will return **true** if the requesting client supports script callbacks.

Working with Client Script in ASP.NET 2.0

Client scripts in ASP.NET 2.0 are managed via the use of the ClientScriptManager class. The ClientScriptManager class keeps track of client scripts using a type and a name. This prevents the same script from being programmatically inserted on a page more than once.

> **NOTE**
> After a script has been successfully registered on a page, any subsequent attempt to register the same script will simply result in the script not being registered a second time. No duplicate scripts are added and no exception occurs. To avoid unnecessary computation, there are methods that you can use to determine if a script is already registered so that you do not attempt to register it more than once.

The methods of the ClientScriptManager should be familiar to all current ASP.NET developers:

RegisterClientScriptBlock

This method adds a script to the top of the rendered page. This is useful for adding functions that will be explicitly called on the client.

There are two overloaded versions of this method. Three of four arguments are common among them. They are:

`type (string)`

The ***type*** argument identifies a type for the script. It is generally a good idea to use the page's type (this.GetType()) for the type.

`key (string)`

The ***key*** argument is a user-defined key for the script. This should be unique for each script. If you attempt to add a script with the same key and type of an already added script, it will not be added.

`script (string)`

The ***script*** argument is a string containing the actual script to add. It's recommended that you use a StringBuilder to create the script and then use the ToString() method on the StringBuilder to assign the ***script*** argument.

If you use the overloaded RegisterClientScriptBlock that only takes three arguments, you must include script elements (<script> and </script>) in your script.

You may choose to use the overload of RegisterClientScriptBlock that takes a fourth argument. The fourth argument is a Boolean that specifies whether or not ASP.NET should add script elements for you. If this argument is **true**, your script should not include the script elements explicitly.

Use the IsClientScriptBlockRegistered method to determine if a script has already been registered. This allows you to avoid an attempt to re-register a script that has already been registered.

RegisterClientScriptInclude (New in 2.0)

The RegisterClientScriptInclude tag creates a script block that links to an external script file. It has two overloads. One takes a key and a URL. The second adds a third argument specifying the type.

For example, the following code generates a script block that links to jsfunctions.js in the root of the scripts folder of the application:

```
ClientScriptManager cm = Page.ClientScript; if(!cm.IsClientScriptIncludeRegistered("jsfunc")) {
cm.RegisterClientScriptInclude(this.GetType(), "jsfunc", "/scripts/jsfunctions.js"); }
```

This code produces the following code in the rendered page:

```
<script src="/scripts/jsfunctions.js" type="text/javascript"></script>
```

> **NOTE**
> The script block is rendered at the bottom of the page.

Use the IsClientScriptIncludeRegistered method to determine if a script has already been registered. This allows you to avoid an attempt to re-register a script.

RegisterStartupScript

The RegisterStartupScript method takes the same arguments as the RegisterClientScriptBlock method. A script registered with RegisterStartupScript executes after the page loads but before the OnLoad client-side event. In 1.X, scripts registered with RegisterStartupScript were placed just before the closing </form> tag while scripts registered with RegisterClientScriptBlock were placed immediately after the opening <form> tag. In ASP.NET 2.0, both are placed immediately before the closing </form> tag.

> **NOTE**
> If you register a function with RegisterStartupScript, that function will not execute until you explicitly call it in client-side code.

Use the IsStartupScriptRegistered method to determine if a script has already been registered and avoid an attempt to re-register a script.

Other ClientScriptManager Methods

Here are some of the other useful methods of the ClientScriptManager class.

GETCALLBACKEVENTREFERENCE	SEE SCRIPT CALLBACKS EARLIER IN THIS MODULE.
GetPostBackClientHyperlink	Gets a JavaScript reference (javascript:<call>) that can be used to post back from a client-side event.
GetPostBackEventReference	Gets a string that can be used to initiate a post back from the client.
GetWebResourceUrl	Returns a URL to a resource that is embedded in an assembly. Must be used in conjunction with **RegisterClientScriptResource**.
RegisterClientScriptResource	Registers a Web resource with the page. These are resources embedded in an assembly and handled by the new WebResource.axd handler.
RegisterHiddenField	Registers a hidden form field with the page.

GETCALLBACKEVENTREFERENCE	SEE SCRIPT CALLBACKS EARLIER IN THIS MODULE.
RegisterOnSubmitStatement	Registers client-side code that executes when the HTML form is submitted.

Server Controls

4/10/2020 • 16 minutes to read • Edit Online

by Microsoft

> ASP.NET 2.0 enhances server controls in many ways. In this module, we'll cover some of the architectural changes to the way ASP.NET 2.0 and Visual Studio 2005 deals with server controls.

ASP.NET 2.0 enhances server controls in many ways. In this module, we'll cover some of the architectural changes to the way ASP.NET 2.0 and Visual Studio 2005 deals with server controls.

View state

The primary change in view state in ASP.NET 2.0 is a dramatic reduction in size. Consider a page with only a Calendar control on it. Here is the view state in ASP.NET 1.1.


```
dDwtMTg1NDkwMjc0Nzt0PDtsPGk8MT47PjtsPHQ8O2w8aTwxPjs
+O2w8dDxAMDxwPHA8bDxTRDs+O2w8bDxTeXN0ZW0uRGF0ZVRpbWUsIG1
zY29ybGliLCBWZXJzaW9uPTEuMC41MDAwLjAsIEN1bHR1cmU9bmV1dHJ
hbCwgUHVibGljS2V5VG9rZW49Yjc3YTVjNTYxOTM0ZTA4OTwyMDA1LTE
xLTA4Pjs+Oz4+Oz47Ozs7Ozs7Pjs7Pjs+Pjs+Pjs+lkX2YWqfACtP
/VWr8G03pob/+tU=
```


Now here's the view state on an identical page in ASP.NET 2.0.


```
/wEPDwULLTEzNjg5MjAxMzgPZBYCAgMPZBYCAgEPPCsAC
gEADxYCHgJTRBYBBgDAEX8OsscIZGRkllfArINjlhvzQX7Xfign2q6HK5E=
```


That's a pretty significant change, and considering that view state is carried back and forth over the wire, this change can give developers a significant performance increase. The reduction in size of view state is largely due to the way that we handle it internally. Remember that view state is a Base64 encoded string. To better understand the change in view state in ASP.NET 2.0, let's have a look at the decoded values from the examples above.

Here is the 1.1 view state decoded:

```
t<-1854902747;t<;l<i<1>;>;l<t<;l<
i<1>;>;l<t<@0<p<p<l<SD;>;l<l<
System.DateTime, mscorlib, Version=1.0.5000.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089<2005-11-08>;>;>>;
>;;;;;;;;;;>;;>;>>;>>;>Eaj
```

This may look a bit like gibberish, but there is a pattern here. In ASP.NET 1.x, we used single characters to identify data-types and delimited values using the <> characters. The "t" in the view state sample above represents a Triplet. The Triplet contains a pair of ArrayLists (the "l" represents an ArrayList.) One of those ArrayLists contains an Int32 ("i") with a value of 1 and the other contains another Triplet. The Triplet contains a pair of ArrayLists, etc. The important thing to remember is that we use Triplets that contain pairs, we identify the data-types via a letter, and we use the < and > characters as delimiters.

In ASP.NET 2.0, the decoded view state looks a bit different.

```
-1368920138 d
 d


 SD
 dddWc A ('ك
```

You should notice a huge change in the appearance of the decoded view state. This change has several architectural underpinnings. View state in ASP.NET 1.x used the LosFormatter to serialize data. In 2.0, we use the new ObjectStateFormatter class. This class was specifically designed to aid in the serialization and deserialization of view state and control state. (Control state will be covered in the next section.) There are many benefits gained by changing the method by which serialization and deserialization take place. One of the most dramatic is the fact that unlike the LosFormatter which uses a TextWriter, the ObjectStateFormatter uses a BinaryWriter. This allows ASP.NET 2.0 to store view state a series of bytes instead of strings. Take, for example, an integer. In ASP.NET 1.1, an integer required 4 bytes of view state. In ASP.NET 2.0, that same integer only requires 1 byte. Other enhancements were made to decrease the amount of view state that is stored. DateTime values, for example, are now stored using a TickCount instead of a string.

As if all of that weren't enough, special attention was paid to the fact that one of the greatest consumers of view state in 1.x was the DataGrid and similar controls. A major drawback of controls such as the DataGrid where view state is concerned is that it often contains large amounts of repeated information. In ASP.NET 1.x, that repeated information was simply stored over and over again resulting in a bloated view state. In ASP.NET 2.0, we use the new IndexedString class to store such data. If a string repeats, we just store the token for the IndexedString and the index within a running table of IndexedString objects.

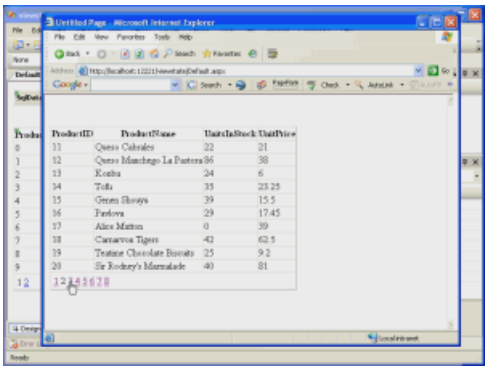
Control State

One of the major gripes that developers had with view state was the size that it added to the HTTP payload. As previously mentioned, one of the greatest consumers of view state is the DataGrid control. To avoid the huge amounts of view state generated by a DataGrid, many developers simply disabled view state for that control. Unfortunately, that solution wasn't always a good one. View state in ASP.NET 1.x contains not only data necessary for the correct functionality of the control. It also contains information concerning the state of the control's UI. This means that if you want to allow for pagination on a DataGrid, you must enable view state even if you don't need all of the UI information that view state contains. It's an all-or-nothing scenario.

In ASP.NET 2.0, control state solves that problem nicely via the introduction of control state. Control state contains the data that is absolutely necessary for the proper functionality of a control. Unlike view state, control state cannot be disabled. Therefore, it is important that the data being stored in control state is carefully controlled.

> **NOTE**
>
> Control state is persisted along with the view state in the __VIEWSTATE hidden form field.

This video is a walkthrough of view state and control state.



Open Full-Screen Video

In order for a server control to read and write to control state, you must take three steps.

Step 1: Call the RegisterRequiresControlState Method

The RegisterRequiresControlState method informs ASP.NET that a control needs to persist control state. It takes one argument of type Control which is the control that is being registered.

It is important to note that registration does not persist from request to request. Therefore, this method must be called on every request if a control is to persist control state. It is recommended that the method be called in OnInit.

```
protected override void OnInit(EventArgs e) { Page.RegisterRequiresControlState(this); base.OnInit(e); }
```

Step 2: Override SaveControlState

The SaveControlState method saves control state changes for a control since the last post back. It returns an object representing the control's state.

Step 3: Override LoadControlState

The LoadControlState method loads saved state into a control. The method takes one argument of type Object which holds the saved state for the control.

Full XHTML Compliance

Any Web developer knows the importance of standards in Web applications. In order to maintain a standards-based development environment, ASP.NET 2.0 is fully XHTML compliant. Therefore, all tags are rendered according to XHTML standards in browsers that support HTML 4.0 or greater.

The DOCTYPE definition in ASP.NET 1.1 was as follows:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN
```

In ASP.NET 2.0, the default DOCTYPE definition is as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

If you choose, you can alter the default XHTML compliance via the xhtmlConformance node in the configuration file. For example, the following node in the web.config file will change XHTML compliance to XHTML 1.0 Strict:

```
<xhtmlConformance mode="Strict" />
```

If you choose, you can also configure ASP.NET to use the legacy configuration used in ASP.NET 1.x as follows:

```
<xhtmlConformance mode="Legacy" />
```

Adaptive Rendering Using Adapters

In ASP.NET 1.x, the configuration file contained a <browserCaps> section that populated a HttpBrowserCapabilities object. This object allowed a developer to determine what device is making a particular request and render code appropriately. In ASP.NET 2.0, the model has improved and now uses the new ControlAdapter class. The ControlAdapter class overrides events in the control's lifecycle and controls the rendering of controls based upon the user agent's capabilities. The capabilities of a specific user agent are defined by a browser definition file (a file with a .browser file extension) stored in the c:\windows\microsoft.net\framework\v2.0.****\CONFIG\Browsers folder.

> **NOTE**
> The ControlAdapter class is an abstract class.

Much like the <browserCaps> section in 1.x, the browser definition file uses a Regular Expression to parse the user agent string in order to identify the requesting browser. It them defines particular capabilities for that user agent. The ControlAdapter renders the control via the Render method. Therefore, if you override the Render method, you should not call Render on the base class. Doing so may cause rendering to occur twice, once for the adapter and once for the control itself.

Developing a Custom Adapter

You can develop your own custom adapter by inheriting from ControlAdapter. Additionally, you can inherit from the abstract class PageAdapter in cases where an adapter is needed for a page. Mapping of controls to your custom adapter is accomplished via the <controlAdapters> element in the browser definition file. For example, the following XML from a browser definition file maps the Menu control to the MenuAdapter class:

```
<controlAdapters> <adapter controlType="System.Web.UI.WebControls.Menu"
adapterType="System.Web.UI.WebControls.Adapters.MenuAdapter" /> </controlAdapters>
```

Using this model, it becomes quite easy for a control developer to target a particular device or browser. It's also quite simple for a developer to have complete control over how pages render on every device.

Per-Device Rendering

Server control properties in ASP.NET 2.0 can be specified per-device using a browser-specific prefix. For example, the code below will change the Text of a label depending upon which device is being used to browse the page.

```
<asp:Label ID="lblBrowser" runat="server" Text="You are browsing from an unknown device." ie:Text="You are
browsing from Internet Explorer." mozilla:Text="You are browsing from Firefox."> </asp:Label>
```

When the page containing this label is browsed from Internet Explorer, the label will display text saying "You are browsing from Internet Explorer." When the page is browsed from Firefox, the label will display the text "You are browsing from Firefox." When the page is browsed from any other device, it will display "You are browsing from

an unknown device." Any property can be specified using this special syntax.

Setting Focus

ASP.NET 1.x developers frequently asked about how to set initial focus on a particular control. For example, on a login page, it's useful to have the User ID textbox get the focus when the page first loads. In ASP.NET 1.x, doing this required writing some client-side script. Even though such a script is a trivial task, it's no longer necessary in ASP.NET 2.0 thanks to the SetFocus method. The SetFocus method takes one argument indicating the control that should receive focus. This argument can either be the client ID of the control as a string or the name of the Server control as a Control object. For example, to set the initial focus to a TextBox control called txtUserID when the page first loads, add the following code to Page_Load:

```
if (!IsPostBack) {
    SetFocus(txtUserID);
}
```

-- or

```
if (!IsPostBack) {
    SetFocus(txtUserID.ClientID);
}
```

ASP.NET 2.0 uses the Webresource.axd handler (discussed previously) to render a client-side function that sets the focus. The name of the client-side function is WebForm_AutoFocus as shown here:

```
<script type="text/javascript"> <!-- WebForm_AutoFocus('txtUserID'); // --> </script>
```

Alternatively, you can use the Focus method for a control to set the initial focus to that control. The Focus method derives from the Control class and is available to all ASP.NET 2.0 controls. It is also possible to set focus to a particular control when a validation error occurs. That will be covered in a later module.

New Server Controls in ASP.NET 2.0

The following are new server controls in ASP.NET 2.0. We will go into more detail on some of them in later modules.

ImageMap Control

The ImageMap control allows you to add hotspots to an image that can initiate a post back or navigate to a URL. There are three types of hotspots available; CircleHotSpot, RectangleHotSpot, and PolygonHotSpot. Hotspots are added via a collection editor in Visual Studio or programmatically in code. There is no user-interface available for drawing hotspots on an image. The coordinates and size or radius of the hotspot must be specified declaratively. There is also no visual representation of a hotspot in the designer. If a hotspot is configured to navigate to a URL, the URL is specified via the NavigateUrl property of the hotspot. In the case of a post back hotspot, the PostBackValue property allows you to pass a string in the post back that can be retrieved in server-side code.

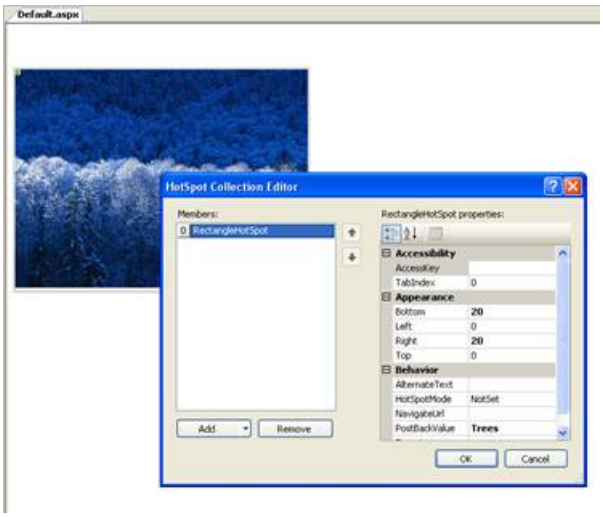


Figure 1: HotSpot Collection Editor in Visual Studio

BulletedList Control

The BulletedList control is a bulleted list that can easily be data bound. The list can be ordered (numbered) or unordered via the BulletStyle property. Each item in the list is represented by a ListItem object.

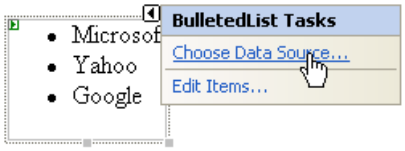


Figure 2: BulletedList Control in Visual Studio

HiddenField Control

The HiddenField control adds a hidden form field to your page, the value of which is available in server-side code. The value of a hidden form field is generally expected to remain unchanged between post backs. However, it is possible for a malicious user to change the value prior to post back. If this happens, the HiddenField control will raise the ValueChanged event. If you have sensitive information in the HiddenField control and you want to ensure that it remains unchanged, you should handle the ValueChanged event in your code.

FileUpload Control

The FileUpload control in ASP.NET 2.0 makes it possible to upload files to a Web server via an ASP.NET page. This control is quite similar to the ASP.NET 1.x HtmlInputFile class with a few exceptions. In ASP.NET 1.x, it was recommended that the PostedFile property be checked for null in order to determine if you had a good file. The FileUpload control in ASP.NET 2.0 adds a new HasFile property that you can use for the same purpose and it's a bit more efficient.

The PostedFile property is still available for access to an HttpPostedFile object, but some of the functionality of the HttpPostedFile is now available intrinsically with the FileUpload control. For example, to save an uploaded file in ASP.NET 1.x, you call the SaveAs method on the HttpPostedFile object. Using the FileUpload control in ASP.NET 2.0, you would call the SaveAs method on the FileUpload control itself.

Another significant change in the 2.0 behavior (and likely the most significant change) is that it is no longer necessary to load an entire uploaded file into memory before saving it. In 1.x, any file that was uploaded is saved entirely into memory prior to being written to disk. This architecture prevents the upload of large files.

In ASP.NET 2.0, the requestLengthDiskThreshold attribute of the httpRuntime element allows you to configure how many Kilobytes are held in a buffer in memory prior to being written to disk.

IMPORTANT: MSDN documentation (and documentation elsewhere) specifies that this value is in bytes (not Kilobytes) and that the default is 256. The value is actually specified in Kilobytes and the default value is 80. By having a default value of 80K, we ensure that the buffer does not end up on the large object heap.

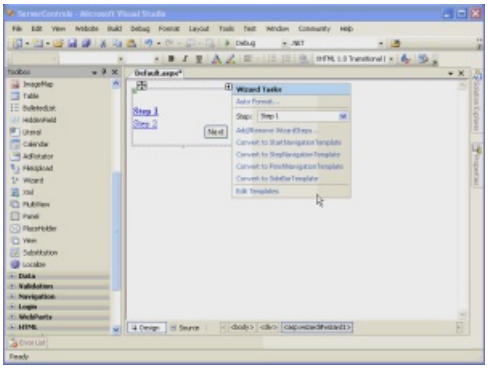
Wizard Control

It is fairly common to encounter ASP.NET developers struggling with attempting to gather information in a series of "pages" using panels or by transferring from page to page. More often than not, the endeavor is a frustrating one and is time consuming. The new Wizard control solves the problems by allowing for linear and non-linear steps in a wizard interface that users are familiar with. The Wizard control presents input forms in a series of steps. Each step is of a particular type specified by the StepType property of the control. The available step types are as follows:

STEP TYPE	EXPLANATION
Auto	The wizard automatically determines the type of step based upon its position within the step hierarchy.
Start	The first step, often used to present an introductory statement.
Step	A normal step.
Finish	The final step, usually used to present a button to finish the wizard.
Complete	Presents a message communicating success or failure.

> **NOTE**
>
> The Wizard control keeps track of its state using ASP.NET control state. Therefore, the EnableViewState property can be set to false without any detriment.

This video is a walkthrough of the Wizard control.



Open Full-Screen Video

Localize Control

The Localize control is similar to a Literal control. However, the Localize control has a **Mode** property that controls how markup that is added to it is rendered. The Mode property supports the following values:

MODE	EXPLANATION
Transform	Markup is transformed according to the protocol of the browser making the request.
PassThrough	Markup is rendered as-is.
Encode	Markup that is added to the control is encoded using HtmlEncode.

MultiView and View Controls

The MultiView control acts as a container for View controls, and the View control acts as a container (much like a Panel control) for other controls. Each view in a MultiView control is represented by a single View control. The first View control in the MultiView is view 0, the second is view 1, etc. You can switch views by specifying the ActiveViewIndex of the MultiView control.

Substitution Control

The Substitution control is used in conjunction with ASP.NET caching. In cases where you want to take advantage of caching, but you have portions of a page that must be updated on each request (in other words, portions of a page that are exempt from caching), the Substitution component provides a great solution. The control doesn't actually render any output on its own. Instead, it is bound to a method in server-side code. When the page is requested, the method is called and the returned markup is rendered in place of the substitution control.

The method to which the Substitution control is bound is specified via the **MethodName** property. That method must meet the following criteria:

- It must be a static (shared in VB) method.
- It accepts one parameter of type HttpContext.
- It returns a string representing the markup that should replace the control on the page.

The Substitution control does not have the ability to modify any other control on the page, but it does have access to the current HttpContext via its parameter.

GridView Control

The GridView control is the replacement for the DataGrid control. This control will be covered in more detail in a later module.

DetailsView Control

The DetailsView control allows you to display a single record from a data source and to edit or delete it. It is covered in more detail in a later module.

FormView Control

The FormView control is used to display a single record from a datasource in a configurable interface. It is covered in more detail in a later module.

AccessDataSource Control

The AccessDataSource control is used to data bind an Access database. It is covered in more detail in a later module.

ObjectDataSource Control

The ObjectDataSource control is used to support a three-tier architecture so that controls can be data-bound to a middle-tier business object as opposed to a two-tiered model where controls are bound directly to the data source. It will be discussed in more detail in a later module.

XmlDataSource Control

The XmlDataSource control is used to data bind to an XML data source. It is covered in more detail in a later module.

SiteMapDataSource Control

The SiteMapDataSource control provides data binding for site navigation controls based on a site map. It will be discussed in more detail in a later module.

SiteMapPath Control

The SiteMapPath control displays a series of navigation links commonly referred to as breadcrumbs. It is covered in more detail in a later module.

Menu Control

The Menu control displays dynamic menus using DHTML. It is covered in more detail in a later module.

TreeView Control

The TreeView control is used to display a hierarchical tree view of data. It is covered in more detail in a later module.

Login Control

The Login control provides for a mechanism to log into a Web site. It is covered in more detail in a later module.

LoginView Control

The LoginView control allows for the display of different templates based upon a user's login status. It is covered in more detail in a later module.

PasswordRecovery Control

The PasswordRecovery control is used to retrieve forgotten passwords by users of an ASP.NET application. It is covered in more detail in a later module.

LoginStatus

The LoginStatus control displays a user's login status. It is covered in more detail in a later module.

LoginName

The LoginName control displays a user's username after being logged into an ASP.NET application. It is covered in more detail in a later module.

CreateUserWizard

The CreateUserWizard is a configurable wizard which gives users the ability to create an ASP.NET Membership account for use in an ASP.NET application. It is covered in more detail in a later module.

ChangePassword

The ChangePassword control allows users to change their password for an ASP.NET application. It is covered in more detail in a later module.

Various WebParts

ASP.NET 2.0 ships with various Web Parts. These will be covered in detail in a later module.

Master Pages

4/10/2020 • 10 minutes to read • Edit Online

by Microsoft

One of the key components to a successful Web site is a consistent look and feel. In ASP.NET 1.x, developers used user controls to replicate common page elements across a Web application. While that is certainly a workable solution, using user controls does have some drawbacks. For example, a change in position of a user control requires a change to multiple pages across a site. User controls are also not rendered in Design view after being inserted on a page.

One of the key components to a successful Web site is a consistent look and feel. In ASP.NET 1.x, developers used user controls to replicate common page elements across a Web application. While that is certainly a workable solution, using user controls does have some drawbacks. For example, a change in position of a user control requires a change to multiple pages across a site. User controls are also not rendered in Design view after being inserted on a page.

ASP.NET 2.0 introduces Master pages as a way of maintaining a consistent look and feel, and as you'll soon see, Master pages represent a significant improvement over the user control method.

Why Master Pages?

You may be wondering why master pages were needed in ASP.NET 2.0. After all, Web site developers are already using user controls in ASP.NET 1.x to share content areas between pages. There are actually several reasons why user controls are a less-than-optimal solution for creating a common layout.

User controls don't actually define page layout. Instead, they define the layout and functionality for a portion of a page. The distinction between these two is important because it makes manageability of a user control solution much more difficult. For example, when you want to change the position of a user control on your page, you must edit the actual page on which the user control appears. Thats fine if you have only a few pages, but in large sites, it quickly becomes a site management nightmare!

Another drawback of using user controls for defining a common layout is rooted in the architecture of ASP.NET itself. If any public member of a user control is changed, it requires you to recompile all of the pages that use the user control. In turn, ASP.NET will then re-JIT your pages when they are first accessed. This, once again, produces a non-scalable architecture and a site management problem for larger sites.

Both of these problems (and many more) are nicely addressed by master pages in ASP.NET 2.0.

How Master Pages Work

A master page is analogous to a template for other pages. Page elements that should be shared among other pages (i.e. menus, borders, etc.) are added to the master page. When new pages are added to the site, you can associate them with a master page. A page that is associated with a master page is called a **content page**. By default, a content page takes on the appearance from the master page. However, when you create a master page, you can define portions of the page that the content page can replace with its own content. These portions are defined using a new control introduced in ASP.NET 2.0; the **ContentPlaceHolder** control.

A master page can contain any number of ContentPlaceHolder controls (or none at all.) On the content page, the content from the ContentPlaceHolder controls appears inside of **Content** controls, another new control in ASP.NET 2.0. By default, the content pages Content controls are empty so that you can provide your own content. If you

want to use the content from the master page inside of the Content controls, you can do so as you will see later in this module. The Content control is mapped to the ContentPlaceHolder control via the ContentPlaceHolderID attribute of the Content control. The code below maps a Content control to a ContentPlaceHolder control called mainBody on a master page.

```
<asp:Content ID="Content1" ContentPlaceHolderID="mainBody" Runat="Server">
```

NOTE

You will often hear people describe master pages as being a base class for other pages. Thats actually not true. The relationship between master pages and content pages is not one of inheritance.

Figure 1 shows a master page and an associated content page as they appear in Visual Studio 2005. You can see the ContentPlaceHolder control in the master page and the corresponding Content control in the content page. Notice that the master pages content that is outside of the ContentPlaceHolder is visible but grayed out in the content page. Only the content inside of the ContentPlaceHolder can be supplanted by the content page. All other content that comes from the master page is immutable.

Figure 1: A master page and its associated content page

Creating a Master Page

To create a new master page:

1. Open Visual Studio 2005 and create a new Web site.
2. Click File, New, File.
3. Choose Master File from the Add New Item dialog as shown in **figure 2**.
4. Click Add.

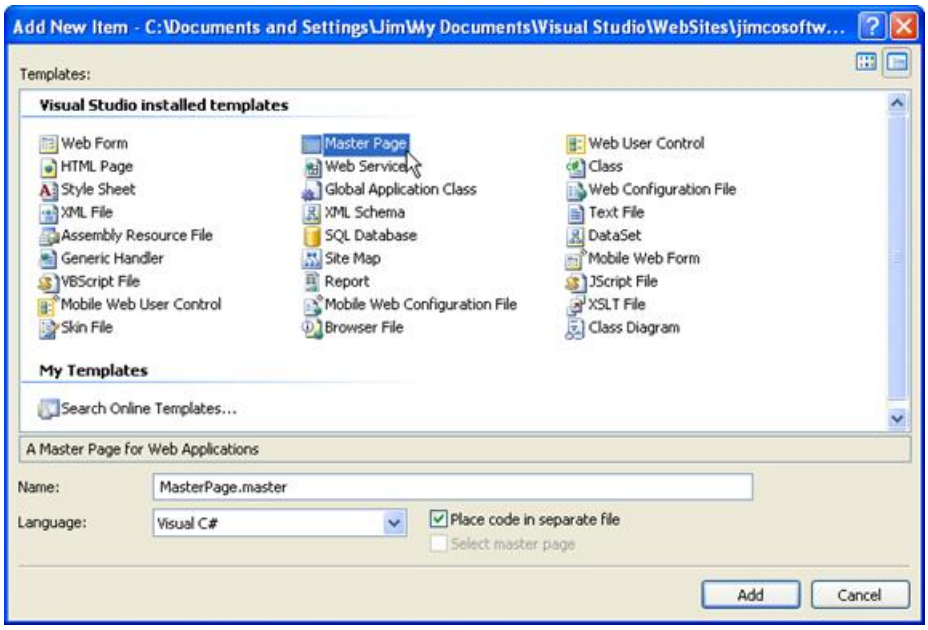


Figure 2: Creating a New Master Page

Notice that the file extension for a master page is *.master*. This is one of the ways that a master page differs from an ordinary page. The other primary difference is that in lieu of a @Page directive, the master page contains a @Master directive. Switch to Source View for the master page you've just created and review the code.

A new master page will have one ContentPlaceHolder control by default. In most cases, it makes more sense to create the common page elements first and then insert ContentPlaceHolder controls where custom content is desired. In those cases, developers will want to delete the default ContentPlaceHolder control and insert new ones as the page is developed. ContentPlaceHolder controls are not resizable despite the fact that they do display sizing handles. The ContentPlaceHolder control sizes automatically based upon the content that it contains with one exception; if you place a ContentPlaceHolder control inside of a block element such as a table cell, it will size according to the size of the element.

Lab 1 Working with Master Pages

In this lab, you will create a new master page and define three ContentPlaceHolder controls. You will then create a new Content page and replace the content from at least one of the ContentPlaceHolder controls.

1. Create a master page and insert ContentPlaceHolder controls.

    a. Create a new master page as described above.
    b. Delete the default ContentPlaceHolder control.
    c. Select the ContentPlaceHolder control by clicking the shaded top border of the control and then delete it by hitting the DEL key on your keyboard.
    d. Insert a new table using the *Header and side* template as shown in figure 3. Change the width and height to 90% each so that the entire table is visible in the designer.

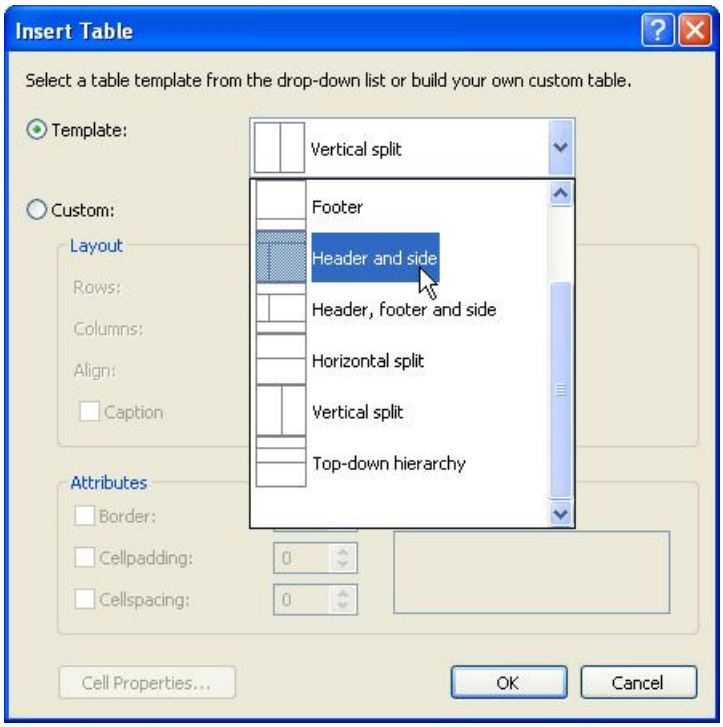


Figure 3

1. Place the cursor into each cell of the table and set the *valign* property to *top*.
2. From the Toolbox, insert a ContentPlaceHolder control in the top cell of the table (the header cell.)
3. When you insert this ContentPlaceHolder control, you will notice that the row height will take up almost the entire page as shown in figure 4. Don't be concerned about that at this point.

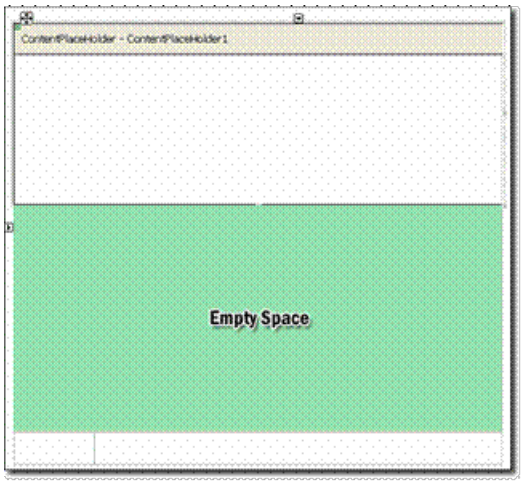


Figure 4: The empty space is in the same cell as the ContentPlaceHolder

1. Place a ContentPlaceHolder control in the other two cells. Once the other ContentPlaceHolder controls have been inserted, the size of the table cells should be as you would expect. The page should now look like the page shown in **figure 5**.

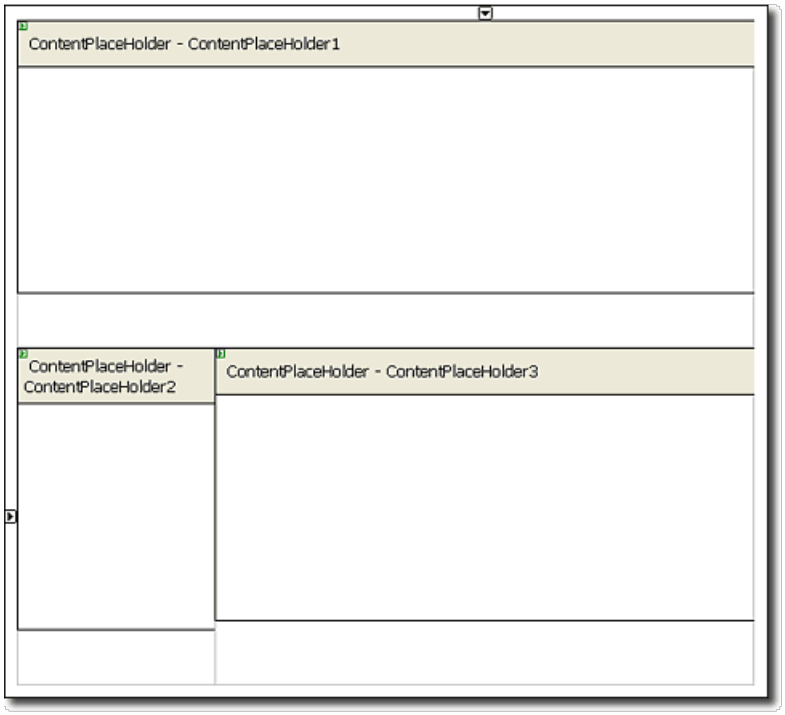


Figure 5: The Master with all ContentPlaceHolder controls. Notice that the cell height for the header cell is now what it should be

1. Enter some text of your choice into each of the three ContentPlaceHolder controls.
2. Save the master page as exercise1.master.
3. Create a new Web Form and associate it with the exercise1.master master page.
4. Select File, New, File in Visual Studio 2005.
5. Select **Web Form** in the Add New Item dialog.
6. Make sure that the Select master page checkbox is checked as shown in figure 6.

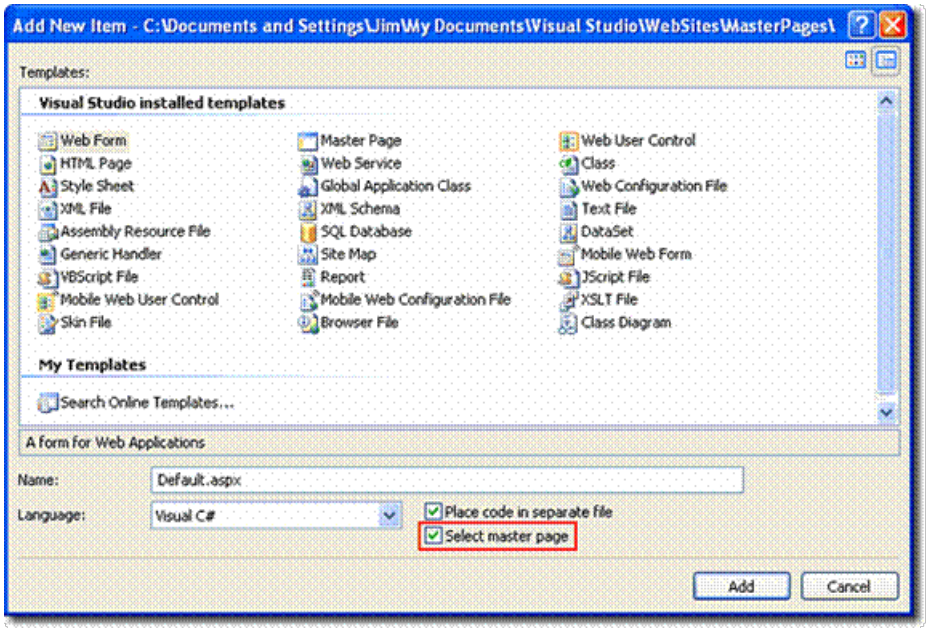


Figure 6: Adding a new Content Page

1. Click Add.
2. Select exercise1.master in the Select a master page dialog as shown in figure 7.
3. Click OK to add the new content page.

The new content page appears in Visual Studio with one Content control for each ContentPlaceHolder control on the master page. By default, the Content controls are empty so that you can add your own content. If you'd like for

them to use the content from the ContentPlaceHolder control on the master page, simply click on the smart tag symbol (the small black arrow in the upper-right corner of the control) and choose *Default to Masters Content* from the smart tag as shown in **figure 8**. When you do so, the menu item changes to *Create Custom Content*. Clicking it at that point removes the content from the master page allowing you to define custom content for that particular Content control.

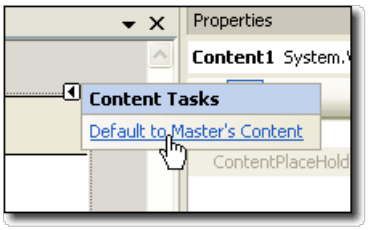


Figure 7: Setting a Content Control to Default to the Master Pages Content

Connecting Master Page and Content Pages

The association between a master page and a content page can be configured in one of four different ways:

- The **MasterPageFile** attribute of the @Page directive
- Setting the **Page.MasterPageFile** property in code.
- The **<pages>** element in the applications configuration file (web.config in the root folder of the application)
- The **<pages>** element in a subfolders configuration file (web.config in a subfolder)

MasterPageFile Attribute

The MasterPageFile attribute makes it easy to apply a master page to a particular ASP.NET page. It is also the method used to apply the master page when you check the **Select Master Page** checkbox as you did in Exercise 1.

Setting Page.MasterPageFile in Code

By setting the MasterPageFile property in code, you can apply a particular master page to your content at runtime. This is useful in cases where you may need to apply a specific master page based upon a users role or some other criteria. The MasterPageFile property must be set in the PreInit method. If it is set after the PreInit method, an InvalidOperationException will be thrown. The page on which this property is being set must also have a Content control as the top-level control for the page. Otherwise an HttpException will be thrown when the MasterPageFile property is set.

Using the <pages> Element

You can configure a master page for your pages by setting the masterPageFile attribute in the <pages> element of the web.config file. When using this method, keep in mind that web.config files lower in the application structure can override this setting. Any MasterPageFile attribute set in a @Page directive will also override this setting. Using the <pages> element makes it simple to create a *master* master page that can be overridden if necessary in particular folders or files.

Properties in Master Pages

A master page can expose properties by simply making those properties public within the master page. For example, the following code defines a property called SomeProperty:

```
private string _SomeProperty; public string SomeProperty { get { return _SomeProperty; } set { _SomeProperty =
value; } }
```

To access the SomeProperty property from the Content page, you will need to use the Master property like so:

```
void Page_Load() { Master.SomeProperty = "Master Page Property"; }
```

Nesting Master Pages

Master pages are the perfect solution for ensuring a common look and feel across a large Web application. However, it's not uncommon to have certain parts of a large site share a common interface while other parts share a different interface. To address that need, multiple master pages are the perfect solution. However, that still doesn't address the fact that a large application may have certain components (such as a menu, for example) that are shared among all pages and other components that are shared only among certain sections of the site. For that type of situation, nested master pages fill the need nicely. As you've seen, a normal master page consists of a master page and a content page. In a nested master page situation, there are two master pages; a parent master and a child master. The child master page is also a content page and its master is the parent master page.

Here is the code for a typical master page:

```
<%@ Master Language="C#" AutoEventWireup="true" CodeFile="parent.master.cs" Inherits="parent" %> <!DOCTYPE
html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd"> <html xmlns="http://www.w3.org/1999/xhtml" > <head runat="server"> <title>Typical Master
Page</title> </head> <body> <form id="form1" runat="server"> <div> <asp:contentplaceholder
id="MainMenuContent" runat="server" /> </div> </form> </body> </html>
```

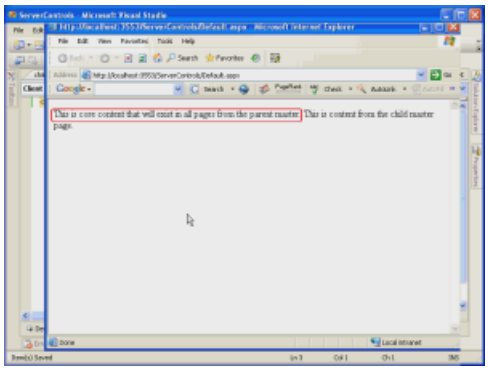
In a nested master scenario, this would be the parent master. Another master page would use this page as its master page, and that code would look like this:

```
<%@ Master Language="C#" MasterPageFile="~/parent.master" AutoEventWireup="true" CodeFile="child.master.cs"
Inherits="child" %> <asp:Content ID="Content1" ContentPlaceHolderID="MainMenuContent" Runat="Server">
<span>From the Child Master.</span> <asp:contentplaceholder id="ChildPlaceHolder1" runat="server" />
</asp:Content>
```

Note that in this scenario, the child master is also a content page for the parent master. All of the child master's content appears inside of a Content control that gets its content from the parent's ContentPlaceHolder control.

> **NOTE**
> Designer support is not available for nested master pages. When you are developing using nested masters, you will need to use source view.

This video shows a walkthrough of using nested master pages.



Open Full-Screen Video

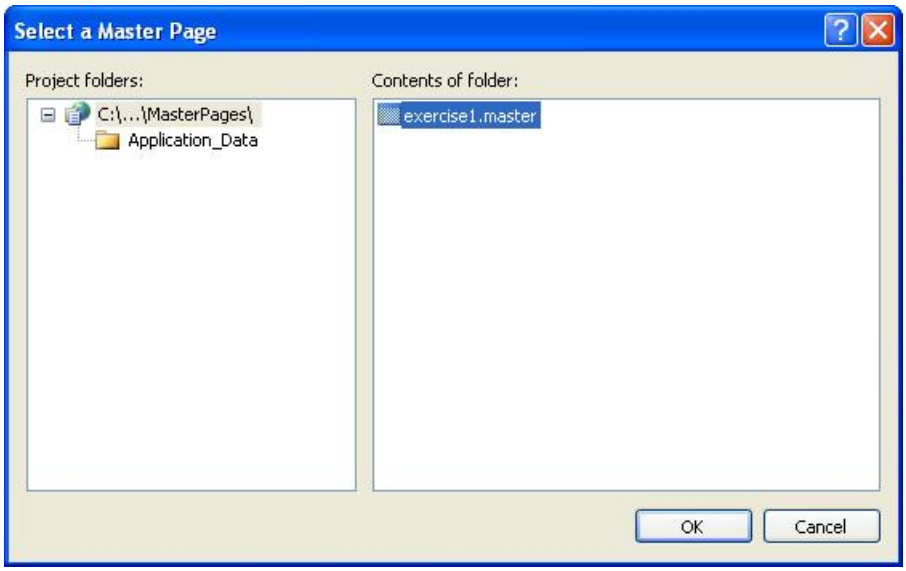


Figure 8: Selecting a Master Page

Data Source Controls

4/10/2020 • 22 minutes to read • Edit Online

by Microsoft

> The DataGrid control in ASP.NET 1.x marked a great improvement in data access in Web applications. However, it wasn't as user-friendly as it could have been. It still required a considerable amount of code to obtain much useful functionality from it. Such is the model in all data access endeavors in 1.x.

The DataGrid control in ASP.NET 1.x marked a great improvement in data access in Web applications. However, it wasn't as user-friendly as it could have been. It still required a considerable amount of code to obtain much useful functionality from it. Such is the model in all data access endeavors in 1.x.

ASP.NET 2.0 addresses this with in part with data source controls. The data source controls in ASP.NET 2.0 provide developers with a declarative model for retrieving data, displaying data, and editing data. The purpose of data source controls is to provide a consistent representation of data to data-bound controls regardless of the source of those data. At the heart of the data source controls in ASP.NET 2.0 is the DataSourceControl abstract class. The DataSourceControl class provides a base implementation of the IDataSource interface and the IListSource interface, the latter of which allows you to assign the data source control as the DataSource of a data-bound control (via the new DataSourceId property discussed later) and expose the data therein as a list. Each list of data from a data source control is exposed as a DataSourceView object. Access to the DataSourceView instances is provided by the IDataSource interface. For example, the GetViewNames method returns an ICollection that allows you to enumerate the DataSourceViews associated with a particular data source control, and the GetView method allows you to access a particular DataSourceView instance by name.

Data source controls have no user-interface. They are implemented as server controls so that they can support declarative syntax and so that they have access to page state if desired. Data source controls do not render any HTML markup to the client.

> **NOTE**
>
> As you'll see later, there are also caching benefits obtained by using data source controls.

Storing Connection Strings

Before we get into looking at how to configure data source controls, we should cover a new capability in ASP.NET 2.0 concerning connection strings. ASP.NET 2.0 introduces a new section in the configuration file that allows you to easily store connection strings that can be read dynamically at runtime. The <connectionStrings> section makes it easy to store connection strings.

The snippet below adds a new connection string.

```
<connectionStrings> <add name="Northwind" connectionString="Data Source=localhost; Integrated
Security=SSPI;Initial Catalog=Northwind;" providerName="System.Data.SqlClient" /> </connectionStrings>
```

> **NOTE**
> Just as with the <appSettings> section, the <connectionStrings> section appears outside of the <system.web> section in the configuration file.

To use this connection string, you can use the following syntax when setting the ConnectionString attribute of a server control.

```
ConnectionString="<%$ ConnectionStrings:Northwind%>"
```

The <connectionStrings> section can also be encrypted so that sensitive information is not exposed. That ability will be covered in a later module.

Caching Data Sources

Each DataSourceControl provides four properties for configuring caching; EnableCaching, CacheDuration, CacheExpirationPolicy, and CacheKeyDependency.

EnableCaching

EnableCaching is a Boolean property that determines whether or not caching is enabled for the data source control.

CacheDuration Property

The CacheDuration property sets the number of seconds that the cache remains valid. Setting this property to **0** causes the cache to remain valid until explicitly invalidated.

CacheExpirationPolicy Property

The CacheExpirationPolicy property can be set to either **Absolute** or **Sliding**. Setting it to Absolute means that the maximum amount of time that the data will be cached is the number of seconds specified by the CacheDuration property. By setting it to Sliding, the expiration time is reset when each operation is performed.

CacheKeyDependency Property

If a string value is specified for the CacheKeyDependency property, ASP.NET will set up a new cache dependency based on that string. This allows you to explicitly invalidate the cache by simply changing or removing the CacheKeyDependency.

Important: If impersonation is enabled and access to the data source and/or content of data are based upon client identity, it is recommended that caching be disabled by setting EnableCaching to False. If caching is enabled in this scenario and a user other than the user who originally requested the data issues a request, authorization to the data source is not enforced. The data will simply be served from cache.

The SqlDataSource Control

The SqlDataSource control allows a developer to access data stored in any relational database that supports ADO.NET. It can use the System.Data.SqlClient provider to access a SQL Server database, the System.Data.OleDb provider, the System.Data.Odbc provider, or the System.Data.OracleClient provider to access Oracle. Therefore, the SqlDataSource is certainly not only used for accessing data in a SQL Server database.

In order to use the SqlDataSource, you simply provide a value for the ConnectionString property and specify a

SQL command or stored procedure. The SqlDataSource control takes care of working with the underlying ADO.NET architecture. It opens the connection, queries the data source or executes the stored procedure, returns the data, and then closes the connection for you.

> **NOTE**
> Because the DataSourceControl class automatically closes the connection for you, it should reduce the number of customer calls generated by leaking database connections.

The code snippet below binds a DropDownList control to a SqlDataSource control using the connection string that is stored in the configuration file as shown above.

```
<asp:SqlDataSource id="SqlDataSource1" runat="server" DataSourceMode="DataReader" ConnectionString="<%$
ConnectionStrings:Northwind%>" SelectCommand="SELECT EmployeeID, LastName FROM Employees">
</asp:SqlDataSource><asp:DropDownList id="ListBox1" runat="server" DataTextField="LastName"
DataValueField="EmployeeID" DataSourceID="SqlDataSource1"> </asp:DropDownList>
```

As illustrated above, the DataSourceMode property of the SqlDataSource specifies the mode for the data source. In the example above, the DataSourceMode is set to DataReader. In that case, the SqlDataSource will return an IDataReader object using a forward-only and read-only cursor. The specified type of object that is returned is controlled by the provider that is used. In this case, I'm using the System.Data.SqlClient provider as specified in the <connectionStrings> section of the web.config file. Therefore, the object that is returned will be of type SqlDataReader. By specifying a DataSourceMode value of DataSet, the data can be stored in a DataSet on the server. This mode allows you to add features such as sorting, paging, etc. If I had been data-binding the SqlDataSource to a GridView control, I would have chosen the DataSet mode. However, in the case of a DropDownList, the DataReader mode is the correct choice.

> **NOTE**
> When caching a SqlDataSource or an AccessDataSource, the DataSourceMode property must be set to DataSet. An exception will occur if you enable caching with a DataSourceMode of DataReader.

SqlDataSource Properties

The following are some of the properties of the SqlDataSource control.

CancelSelectOnNullParameter

A Boolean value that specifies whether a select command is canceled if one of the parameters is null. True by default.

ConflictDetection

In a situation where multiple users may be updating a data source at the same time, the ConflictDetection property determines the behavior of the SqlDataSource control. This property evaluates to one of the values of the ConflictOptions enumeration. Those values are **CompareAllValues** and **OverwriteChanges**. If set to OverwriteChanges, the last person to write data to the data source overwrites any previous changes. However, if the ConflictDetection property is set to CompareAllValues, parameters get created for the columns returned by the SelectCommand and parameters are also created to hold the original values in each of those columns allowing the SqlDataSource to determine whether or not the values have changed since the SelectCommand was executed.

DeleteCommand

Sets or gets the SQL string used when deleting rows from the database. This can either be a SQL query or a stored procedure name.

DeleteCommandType

Sets or gets the type of delete command, either a SQL query (Text) or a stored procedure (StoredProcedure).

DeleteParameters

Returns the parameters that are used by the DeleteCommand of the SqlDataSourceView object associated with the SqlDataSource control.

OldValuesParameterFormatString

This property is used to specify the format of the original value parameters in cases where the ConflictDetection property is set to CompareAllValues. The default is {0} which means that original value parameters will take the same name as the original parameter. In other words, if the field name is EmployeeID, the original value parameter would be @EmployeeID.

SelectCommand

Sets or gets the SQL string that is used to retrieve data from the database. This can either be a SQL query or a stored procedure name.

SelectCommandType

Sets or gets the type of select command, either a SQL query (Text) or a stored procedure (StoredProcedure).

SelectParameters

Returns the parameters that are used by the SelectCommand of the SqlDataSourceView object associated with the SqlDataSource control.

SortParameterName

Gets or sets the name of a stored procedure parameter that is used when sorting data retrieved by the data source control. Valid only when SelectCommandType is set to StoredProcedure.

SqlCacheDependency

A semi-colon delimited string specifying the databases and tables used in a SQL Server cache dependency. (SQL cache dependencies will be discussed in a later module.)

UpdateCommand

Sets or gets the SQL string that is used when updating data in the database. This can either be a SQL query or a stored procedure name.

UpdateCommandType

Sets or gets the type of update command, either a SQL query (Text) or a stored procedure (StoredProcedure).

UpdateParameters

Returns the parameters that are used by the UpdateCommand of the SqlDataSourceView object associated with the SqlDataSource control.

The AccessDataSource Control

The AccessDataSource control derives from the SqlDataSource class and is used to data-bind to a Microsoft Access database. The ConnectionString property for the AccessDataSource control is a read-only property. Instead of using the ConnectionString property, the DataFile property is used to point to the Access Database as shown below.

```
<asp:AccessDataSource id="AccessDataSource1" runat="server" DataFile="~/App_Data/Northwind.mdb">
</asp:AccessDataSource>
```

The AccessDataSource will always set the ProviderName of the base SqlDataSource to System.Data.OleDb and

connects to the database using the Microsoft.Jet.OLEDB.4.0 OLE DB provider. You cannot use the AccessDataSource control to connect to a password-protected Access database. If you have to connect to a password protected database, you should use the SqlDataSource control.

> **NOTE**
> Access databases stored within the Web site should be placed in the App_Data directory. ASP.NET does not allow files in this directory to be browsed. You will need to grant the process account Read and Write permissions to the App_Data directory when using Access databases.

The XmlDataSource Control

The XmlDataSource is used to data-bind XML data to data-bound controls. You can bind to an XML file using the DataFile property or you can bind to an XML string using the Data property. The XmlDataSource exposes XML attributes as bindable fields. In cases where you need to bind to values that are not represented as attributes, you will need to use an XSL transform. You can also use XPath expressions to filter XML data.

Consider the following XML file:

```
<?xml version="1.0" encoding="utf-8" ?> <People> <Person FirstName="Jake" LastName="Stone"> <Address>
<Street>345 Maple St.</Street> <City>Redmond</City> <Region>WA</Region> <ZipCode>01434</ZipCode> </Address>
<Job> <Title>CEO</Title> <Description>Develops company strategies.</Description> </Job> </Person> <Person
FirstName="Jacob" LastName="Ladder"> <Address> <Street>123 Elm St.</Street> <City>Seattle</City>
<Region>WA</Region> <ZipCode>11223</ZipCode> </Address> <Job> <Title>Attorney</Title> <Description>Reviews
legal issues.</Description> </Job> </Person> <Person FirstName="Angela" LastName="Hound"> <Address> <Street>34
Palm Avenue</Street> <City>Renton</City> <Region>WA</Region> <ZipCode>63910</ZipCode> </Address> <Job>
<Title>IT Director</Title> <Description>In charge of corporate network.</Description> </Job> </Person>
</People>
```

Notice that the XmlDataSource uses an XPath property of *People/Person* in order to filter on just the <Person> nodes. The DropDownList then data-binds to the LastName attribute using the DataTextField property.

While the XmlDataSource control is primarily used to data-bind to read-only XML data, it is possible to edit the XML data file. Note that in such cases, automatic insertion, updating, and deletion of information in the XML file does not happen automatically as it does with other data source controls. Instead, you will have to write code to manually edit the data using the following methods of the XmlDataSource control.

GetXmlDocument

Retrieves an XmlDocument object containing the XML code retrieved by the XmlDataSource.

Save

Saves the in-memory XmlDocument back to the data source.

It's important to realize that the Save method will only work when the following two conditions are met:

1. The XmlDataSource is using the DataFile property to bind to an XML file instead of the Data property to bind to in-memory XML data.
2. No transformation is specified via the Transform or TransformFile property.

Note also that the Save method can yield unexpected results when called by multiple users concurrently.

The ObjectDataSource Control

The data source controls that we have covered up to this point are excellent choices for two-tier applications where the data source control communicates directly to the data store. However, many real-world applications are multi-tier applications where a data source control might need to communicate to a business object which, in turn,

communicates with the data layer. In these situations, the ObjectDataSource fills the bill nicely. The ObjectDataSource works in conjunction with a source object. The ObjectDataSource control will create an instance of the source object, call the specified method, and dispose of the object instance all within the scope of a single request, if your object has instance methods instead of static methods (Shared in Visual Basic). Therefore, your object must be stateless. That is, your object should acquire and release all required resources within the span of a single request. You can control how the source object is created by handling the ObjectCreating event of the ObjectDataSource control. You can create an instance of the source object, and then set the ObjectInstance property of the ObjectDataSourceEventArgs class to that instance. The ObjectDataSource control will use the instance that is created in the ObjectCreating event instead of creating an instance on its own.

If the source object for an ObjectDataSource control exposes public static methods (Shared in Visual Basic) that can be called to retrieve and modify data, an ObjectDataSource control will call those methods directly. If an ObjectDataSource control must create an instance of the source object in order to make method calls, the object must include a public constructor that takes no parameters. The ObjectDataSource control will call this constructor when it creates a new instance of the source object.

If the source object does not contain a public constructor without parameters, you can create an instance of the source object that will be used by the ObjectDataSource control in the ObjectCreating event.

Specifying Object Methods

The source object for an ObjectDataSource control can contain any number of methods that are used to select, insert, update, or delete data. These methods are called by the ObjectDataSource control based on the name of the method, as identified by using either the SelectMethod, InsertMethod, UpdateMethod, or DeleteMethod property of the ObjectDataSource control. The source object can also include an optional SelectCount method, which is identified by the ObjectDataSource control using the SelectCountMethod property, that returns the count of the total number of objects at the data source. The ObjectDataSource control will call the SelectCount method after a Select method has been called to retrieve the total number of records at the data source for use when paging.

Lab Using Data Source Controls

Exercise 1 - Displaying Data with the SqlDataSource Control

The following exercise uses the SqlDataSource control to connect to the Northwind database. It assumes that you have access to the Northwind database on a SQL Server 2000 instance.

1. Create a new ASP.NET Web site.
2. Add a new web.config file.
   a. Right-click on the project in Solution Explorer and click Add New Item.
   b. Choose Web Configuration File from the list of templates and click Add.
3. Edit the <connectionStrings> section as follows:

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%$ConnectionStrings:Northwind%>"
    SelectCommand="SELECT * FROM Products">
</asp:SqlDataSource>
```

4. Switch to Code view and add a ConnectionString attribute and a SelectCommand attribute to the <asp:SqlDataSource> control as follows:

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%$ConnectionStrings:Northwind%>"
    SelectCommand="SELECT * FROM Products">
</asp:SqlDataSource>
```

5. From Design view, add a new GridView control.
6. From the Choose Data Source dropdown in the GridView Tasks menu, choose SqlDataSource1.
7. Right-click on Default.aspx and choose View in Browser from the menu. Click Yes when prompted to save.
8. The GridView displays the data from the Products table.

Exercise 2 - Editing Data with the SqlDataSource Control

The following exercise demonstrates how to data bind a DropDownList control using the declarative syntax and allows you to edit the data presented in the DropDownList control.

1. In Design view, delete the GridView control from Default.aspx.

   Important: Leave the SqlDataSource control on the page.

2. Add a DropDownList control to Default.aspx.
3. Switch to Source view.
4. Add a DataSourceId, DataTextField, and DataValueField attribute to the <asp:DropDownList> control as follows:

```
<asp:DropDownList ID="ddlProducts" runat="server"
     DataSourceId="SqlDataSource1" DataTextField="ProductName"
     DataValueField="ProductID">
</asp:DropDownList>
```

5. Save Default.aspx and view it in the browser. Note that the DropDownList contains all of the products from the Northwind database.
6. Close the browser.
7. In Source view of Default.aspx, add a new TextBox control below the DropDownList control. Change the ID property of the TextBox to txtProductName.
8. Under the TextBox control, add a new Button control. Change the ID property of the Button to btnUpdate and the Text property to **Update Product Name**.
9. In Source view of Default.aspx, add an UpdateCommand property and two new UpdateParameters to the SqlDataSource tag as follows:

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%$ConnectionStrings:Northwind%>"
    SelectCommand="SELECT * FROM Products"
    UpdateCommand="UPDATE Products SET ProductName=@ProductName WHERE ProductID=@ProductID">
      <UpdateParameters>
      <asp:ControlParameter Name="ProductName"
        ControlID="txtProductName" PropertyName="Text" />
      <asp:ControlParameter Name="ProductID"
        ControlID="ddlProducts" PropertyName="SelectedValue" />
</asp:SqlDataSource>
```

> **NOTE**
> Note that there are two update parameters (ProductName and ProductID) added in this code. These parameters are mapped to the Text property of the txtProductName TextBox and the SelectedValue property of the ddlProducts DropDownList.

10. Switch to Design view and double-click on the Button control to add an event handler.
11. Add the following code to the btnUpdate_Click code:

```
SqlDataSource1.Update();
```

12. Right-click on Default.aspx and choose to view it in the browser. Click Yes when prompted to save all changes.
13. ASP.NET 2.0 partial classes allow for compilation at runtime. It is not necessary to build an application in order to see code changes take effect.
14. Select a product from the DropDownList.
15. Enter a new name for the selected product in the TextBox and then click the Update button.
16. The product name is updated in the database.

Exercise 3 Using the ObjectDataSource Control

This exercise will demonstrate how to use the ObjectDataSource control and a source object to interact with the Northwind database.

1. Right-click on the project in Solution Explorer and click on Add New Item.
2. Select Web Form in the templates list. Change the name to object.aspx and click Add.
3. Right-click on the project in Solution Explorer and click on Add New Item.
4. Select Class in the templates list. Change the name of the class to NorthwindData.cs and click Add.
5. Click Yes when prompted to add the class to the App_Code folder.
6. Add the following code to the NorthwindData.cs file:

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Data.SqlClient;
public class NorthwindData {
    private string _connectionString;
    public NorthwindData() {
        Initialize();
    }

    private void Initialize() {
        if (ConfigurationManager.ConnectionStrings["Northwind"] == null ||
            ConfigurationManager.ConnectionStrings["Northwind"].ConnectionString.Trim() == "") {
                throw new Exception("A connection string named 'Northwind' with " +
```

```
                "a valid connection string must exist in the <connectionStrings> " +
                "configuration section for the application.");
        }
        _connectionString = ConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
    }

    public DataTable GetAllEmployees(string sortColumns, int startRecord, int maxRecords) {
        VerifySortColumns(sortColumns);
        string sqlCmd = "SELECT EmployeeID, LastName, FirstName, Address, " +
            "City, Region, PostalCode FROM Employees ";
        if (sortColumns.Trim() == "")
            sqlCmd += "ORDER BY EmployeeID";
        else
            sqlCmd += "ORDER BY " + sortColumns;

        SqlConnection conn = new SqlConnection(_connectionString);
        SqlDataAdapter da = new SqlDataAdapter(sqlCmd, conn);
        DataSet ds = new DataSet();
        try {
            conn.Open();
            da.Fill(ds, startRecord, maxRecords, "Employees");
        } catch (SqlException e) {
            // Handle exception.
        } finally {
            conn.Close();
        }
        return ds.Tables["Employees"];
    }

    public int SelectCount() {
        SqlConnection conn = new SqlConnection(_connectionString);
        SqlCommand cmd = new SqlCommand("SELECT COUNT(*) FROM Employees", conn);
        int result = 0;

        try {
            conn.Open();
            result = (int)cmd.ExecuteScalar();
        } catch (SqlException e) {
            // Handle exception.
        } finally {
            conn.Close();
        }
        return result;
    }

    //////////
    // Verify that only valid columns are specified in the sort expression to
    // avoid a SQL Injection attack.
    private void VerifySortColumns(string sortColumns) {
        if (sortColumns.ToLowerInvariant().EndsWith(" desc"))
            sortColumns = sortColumns.Substring(0, sortColumns.Length - 5);
        string[] columnNames = sortColumns.Split(',');
        foreach (string columnName in columnNames) {
            switch (columnName.Trim().ToLowerInvariant()) {
                case "employeeid":
                    break;
                case "lastname":
                    break;
                case "firstname":
                    break;
                case "":
                    break;
                default:
                    throw new ArgumentException("SortColumns contains an " +
                        "invalid column name.");
                    break;
            }
        }
    }
```

```
    }

    // Select an employee.
    public DataTable GetEmployee(int EmployeeID) {
        SqlConnection conn = new SqlConnection(_connectionString);
        SqlDataAdapter da =
            new SqlDataAdapter("SELECT EmployeeID, LastName, FirstName, " +
            "Address, City, Region, PostalCode " +
            " FROM Employees WHERE EmployeeID = @EmployeeID", conn);
        da.SelectCommand.Parameters.Add("@EmployeeID", SqlDbType.Int).Value = EmployeeID;
        DataSet ds = new DataSet();
        try {
            conn.Open();
            da.Fill(ds, "Employees");
        } catch (SqlException e) {
            // Handle exception.
        } finally {
            conn.Close();
        }

        return ds.Tables["Employees"];
    }

    // Delete the Employee by ID.
    public int DeleteEmployee(int EmployeeID) {
         SqlConnection conn = new SqlConnection(_connectionString);
         SqlCommand cmd = new SqlCommand("DELETE FROM Employees WHERE " +
             "EmployeeID = @EmployeeID", conn);
         cmd.Parameters.Add("@EmployeeID", SqlDbType.Int).Value = EmployeeID;
         int result = 0;
         try {
             conn.Open();
             result = cmd.ExecuteNonQuery();
         } catch (SqlException e) {
             // Handle exception.
         } finally {
             conn.Close();
         }

         return result;
     }

     // Update the Employee by original ID.
     public int UpdateEmployee(int EmployeeID, string LastName, string FirstName,
         string Address, string City, string Region,
         string PostalCode) {
         if (String.IsNullOrEmpty(FirstName))
             throw new ArgumentException("FirstName cannot be null or an empty string.");
         if (String.IsNullOrEmpty(LastName))
             throw new ArgumentException("LastName cannot be null or an empty string.");
         if (Address == null) { Address = String.Empty; }
         if (City == null) { City = String.Empty; }
         if (Region == null) { Region = String.Empty; }
         if (PostalCode == null) { PostalCode = String.Empty; }

         SqlConnection conn = new SqlConnection(_connectionString);
         SqlCommand cmd = new SqlCommand("UPDATE Employees " +
             " SET FirstName=@FirstName, " +
             "LastName=@LastName, " +
             "Address=@Address, City=@City, " +
             "Region=@Region, " +
             "PostalCode=@PostalCode " +
             "WHERE EmployeeID=@EmployeeID", conn);
         cmd.Parameters.Add("@FirstName", SqlDbType.VarChar, 10).Value = FirstName;
         cmd.Parameters.Add("@LastName", SqlDbType.VarChar, 20).Value = LastName;
         cmd.Parameters.Add("@Address", SqlDbType.VarChar, 60).Value = Address;
         cmd.Parameters.Add("@City", SqlDbType.VarChar, 15).Value = City;
         cmd.Parameters.Add("@Region", SqlDbType.VarChar, 15).Value = Region;
         cmd.Parameters.Add("@PostalCode", SqlDbType.VarChar, 10).Value = PostalCode;
         cmd.Parameters.Add("@EmployeeID", SqlDbType.Int).Value = EmployeeID;
```

```
        cmd.Parameters.Add("@EmployeeID", SqlDbType.Int).Value = _EmployeeID;

        int result = 0;
        try {
            conn.Open();
            result = cmd.ExecuteNonQuery();
        } catch (SqlException e) {
            // Handle exception.
        } finally {
            conn.Close();
        }

        return result;
    }

    // Insert an Employee.
    public int InsertEmployee(string LastName, string FirstName,
        string Address, string City, string Region,
        string PostalCode) {
        if (String.IsNullOrEmpty(FirstName))
            throw new ArgumentException("FirstName cannot be null or an empty string.");
        if (String.IsNullOrEmpty(LastName))
            throw new ArgumentException("LastName cannot be null or an empty string.");
        if (Address == null) { Address = String.Empty; }
        if (City == null) { City = String.Empty; }
        if (Region == null) { Region = String.Empty; }
        if (PostalCode == null) { PostalCode = String.Empty; }

        SqlConnection conn = new SqlConnection(_connectionString);
        SqlCommand cmd = new SqlCommand("INSERT INTO Employees " +
            " (FirstName, LastName, Address, " +
            " City, Region, PostalCode) " +
            " Values(@FirstName, @LastName, " +
            "@Address, @City, @Region, @PostalCode); " +
            "SELECT @EmployeeID = SCOPE_IDENTITY()", conn);

        cmd.Parameters.Add("@FirstName", SqlDbType.VarChar, 10).Value = FirstName;
        cmd.Parameters.Add("@LastName", SqlDbType.VarChar, 20).Value = LastName;
        cmd.Parameters.Add("@Address", SqlDbType.VarChar, 60).Value = Address;
        cmd.Parameters.Add("@City", SqlDbType.VarChar, 15).Value = City;
        cmd.Parameters.Add("@Region", SqlDbType.VarChar, 15).Value = Region;
        cmd.Parameters.Add("@PostalCode", SqlDbType.VarChar, 10).Value = PostalCode;
        SqlParameter p = cmd.Parameters.Add("@EmployeeID", SqlDbType.Int);
            p.Direction = ParameterDirection.Output;
        int newEmployeeID = 0;
        try {
            conn.Open();
            cmd.ExecuteNonQuery();
            newEmployeeID = (int)p.Value;
        } catch (SqlException e) {
            // Handle exception.
        } finally {
            conn.Close();
        }

        return newEmployeeID;
    }

    //
    // Methods that support Optimistic Concurrency checks.
    //
    // Delete the Employee by ID.
    public int DeleteEmployee(int original_EmployeeID, string original_LastName,
        string original_FirstName, string original_Address,
        string original_City, string original_Region,
        string original_PostalCode) {

        if (String.IsNullOrEmpty(original_FirstName))
            throw new ArgumentException("FirstName cannot be null or an empty string.");
        if (String.IsNullOrEmpty(original_LastName))
```

```
        if (String.IsNullOrEmpty(original_LastName))
            throw new ArgumentException("LastName cannot be null or an empty string.");
        if (original_Address == null) { original_Address = String.Empty; }
        if (original_City == null) { original_City = String.Empty; }
        if (original_Region == null) { original_Region = String.Empty; }
        if (original_PostalCode == null) { original_PostalCode = String.Empty; }
        string sqlCmd = "DELETE FROM Employees WHERE EmployeeID = " + @original_EmployeeID

        SqlConnection conn = new SqlConnection(_connectionString);
        SqlCommand cmd = new SqlCommand(sqlCmd, conn);
        cmd.Parameters.Add("@original_EmployeeID",
            SqlDbType.Int).Value = original_EmployeeID;
        cmd.Parameters.Add("@original_FirstName",
            SqlDbType.VarChar, 10).Value = original_FirstName;
        cmd.Parameters.Add("@original_LastName",
            SqlDbType.VarChar, 20).Value = original_LastName;
        cmd.Parameters.Add("@original_Address",
            SqlDbType.VarChar, 60).Value = original_Address;
        cmd.Parameters.Add("@original_City",
            SqlDbType.VarChar, 15).Value = original_City;
        cmd.Parameters.Add("@original_Region",
            SqlDbType.VarChar, 15).Value = original_Region;
        cmd.Parameters.Add("@original_PostalCode",
            SqlDbType.VarChar, 10).Value = original_PostalCode;

        int result = 0;
        try {
            conn.Open();
            result = cmd.ExecuteNonQuery();
        } catch (SqlException e) {
            // Handle exception.
        } finally {
            conn.Close();
        }

        return result;
    }

    // Update the Employee by original ID.
    public int UpdateEmployee(string LastName, string FirstName,
        string Address, string City, string Region,
        string PostalCode, int original_EmployeeID,
        string original_LastName, string original_FirstName,
        string original_Address, string original_City,
        string original_Region, string original_PostalCode) {

        if (String.IsNullOrEmpty(FirstName))
            throw new ArgumentException("FirstName cannot be null or an empty string.");
        if (String.IsNullOrEmpty(LastName))
            throw new ArgumentException("LastName cannot be null or an empty string.");
        if (Address == null) { Address = String.Empty; }
        if (City == null) { City = String.Empty; }
        if (Region == null) { Region = String.Empty; }
        if (PostalCode == null) { PostalCode = String.Empty; }
        if (original_Address == null) { original_Address = String.Empty; }
        if (original_City == null) { original_City = String.Empty; }
        if (original_Region == null) { original_Region = String.Empty; }
        if (original_PostalCode == null) { original_PostalCode = String.Empty; }

        string sqlCmd = "UPDATE Employees " +
            " SET FirstName = @FirstName, LastName = @LastName, " +
            " Address = @Address, City = @City, Region = @Region, " +
            " PostalCode = @PostalCode " +
            " WHERE EmployeeID = @original_EmployeeID";

        SqlConnection conn = new SqlConnection(_connectionString);
        SqlCommand cmd = new SqlCommand(sqlCmd, conn);
        cmd.Parameters.Add("@FirstName", SqlDbType.VarChar, 10).Value = FirstName;
        cmd.Parameters.Add("@LastName", SqlDbType.VarChar, 20).Value = LastName;
```

```
        cmd.Parameters.Add("@Address", SqlDbType.VarChar, 60).Value = Address;
        cmd.Parameters.Add("@City", SqlDbType.VarChar, 15).Value = City;
        cmd.Parameters.Add("@Region", SqlDbType.VarChar, 15).Value = Region;
        cmd.Parameters.Add("@PostalCode", SqlDbType.VarChar, 10).Value = PostalCode;
        cmd.Parameters.Add("@original_EmployeeID",
            SqlDbType.Int).Value = original_EmployeeID;
        cmd.Parameters.Add("@original_FirstName",
            SqlDbType.VarChar, 10).Value = original_FirstName;
        cmd.Parameters.Add("@original_LastName",
            SqlDbType.VarChar, 20).Value = original_LastName;
        cmd.Parameters.Add("@original_Address",
            SqlDbType.VarChar, 60).Value = original_Address;
        cmd.Parameters.Add("@original_City",
            SqlDbType.VarChar, 15).Value = original_City;
        cmd.Parameters.Add("@original_Region",
            SqlDbType.VarChar, 15).Value = original_Region;
        cmd.Parameters.Add("@original_PostalCode",
            SqlDbType.VarChar, 10).Value = original_PostalCode;

        int result = 0;

        try {
            conn.Open();
            result = cmd.ExecuteNonQuery();
        } catch (SqlException e) {
            // Handle exception.
        } finally {
            conn.Close();
        }
        return result;
    }
}
```

7. Add the following code to the Source view of object.aspx:

```
<%@ Page language="C#" %>
<script RunAt="server">
void EmployeesDetailsView_ItemInserted(Object sender, DetailsViewInsertedEventArgs e) {
    EmployeesGridView.DataBind();
}

void EmployeesDetailsView_ItemUpdated(Object sender, DetailsViewUpdatedEventArgs e) {
    EmployeesGridView.DataBind();
}

void EmployeesDetailsView_ItemDeleted(Object sender, DetailsViewDeletedEventArgs e) {
    EmployeesGridView.DataBind();
}
void EmployeesGridView_OnSelectedIndexChanged(object sender, EventArgs e) {
    EmployeeDetailsObjectDataSource.SelectParameters["EmployeeID"].DefaultValue =
        EmployeesGridView.SelectedDataKey.Value.ToString();
    EmployeesDetailsView.DataBind();
}
void EmployeeDetailsObjectDataSource_OnInserted(object sender,
    ObjectDataSourceStatusEventArgs e) {

    EmployeeDetailsObjectDataSource.SelectParameters["EmployeeID"].DefaultValue =
        e.ReturnValue.ToString();
    EmployeesDetailsView.DataBind();
}
void EmployeeDetailsObjectDataSource_OnUpdated(object sender,
    ObjectDataSourceStatusEventArgs e) {

    if ((int)e.ReturnValue == 0)
        Msg.Text = "Employee was not updated. Please try again.";
}
void EmployeeDetailsObjectDataSource_OnDeleted(object sender,
```

```
    ObjectDataSourceStatusEventArgs e) {

    if ((int)e.ReturnValue == 0)
        Msg.Text = "Employee was not deleted. Please try again.";
}
void Page_Load() {
    Msg.Text = "";
}
</script>
<html>
  <body>
    <form id="Form1" runat="server">
      <h3>ObjectDataSource Example</h3>
      <asp:Label id="Msg" runat="server" ForeColor="Red" />
      <asp:ObjectDataSource
          ID="EmployeesObjectDataSource"
          runat="server"
          TypeName="NorthwindData"
          SortParameterName="SortColumns"
          EnablePaging="true"
          SelectCountMethod="SelectCount"
          StartRowIndexParameterName="StartRecord"
          MaximumRowsParameterName="MaxRecords"
          SelectMethod="GetAllEmployees" >
      </asp:ObjectDataSource>
      <asp:ObjectDataSource
          ID="EmployeeDetailsObjectDataSource"
          runat="server"
          TypeName="NorthwindData"
          ConflictDetection="CompareAllValues"
          OldValuesParameterFormatString="{0}"
          SelectMethod="GetEmployee"
          InsertMethod="InsertEmployee"
          UpdateMethod="UpdateEmployee"
          DeleteMethod="DeleteEmployee"
          OnInserted="EmployeeDetailsObjectDataSource_OnInserted"
          OnUpdated="EmployeeDetailsObjectDataSource_OnUpdated"
          OnDeleted="EmployeeDetailsObjectDataSource_OnDeleted">
          <SelectParameters>
              <asp:Parameter Name="EmployeeID" Type="Int32" />
          </SelectParameters>
      </asp:ObjectDataSource>
      <table cellspacing="10">
        <tr>
          <td valign="top">
            <asp:GridView ID="EmployeesGridView"
                DataSourceID="EmployeesObjectDataSource"
                AutoGenerateColumns="false"
                AllowSorting="true"
                AllowPaging="true"
                PageSize="5"
                DataKeyNames="EmployeeID"
                OnSelectedIndexChanged="EmployeesGridView_OnSelectedIndexChanged"
                RunAt="server">
                <HeaderStyle backcolor="lightblue" forecolor="black"/>
                <Columns>
                <asp:ButtonField Text="Details..."
                HeaderText="Show Details"
                CommandName="Select"/>

                <asp:BoundField DataField="EmployeeID" HeaderText="Employee ID"
                SortExpression="EmployeeID" />
                <asp:BoundField DataField="FirstName" HeaderText="First Name"
                SortExpression="FirstName" />
                <asp:BoundField DataField="LastName" HeaderText="Last Name"
                SortExpression="LastName, FirstName" />
                </Columns>
            </asp:GridView>
          </td>
```

```
            <td valign="top">
              <asp:DetailsView ID="EmployeesDetailsView"
                  DataSourceID="EmployeeDetailsObjectDataSource"
                  AutoGenerateRows="false"
                  EmptyDataText="No records."
                  DataKeyNames="EmployeeID"
                  Gridlines="Both"
                  AutoGenerateInsertButton="true"
                  AutoGenerateEditButton="true"
                  AutoGenerateDeleteButton="true"
                  OnItemInserted="EmployeesDetailsView_ItemInserted"
                  OnItemUpdated="EmployeesDetailsView_ItemUpdated"
                  OnItemDeleted="EmployeesDetailsView_ItemDeleted"
                  RunAt="server">
                  <HeaderStyle backcolor="Navy" forecolor="White"/>
                  <RowStyle backcolor="White"/>
                  <AlternatingRowStyle backcolor="LightGray"/>
                  <EditRowStyle backcolor="LightCyan"/>
                  <Fields>
                      <asp:BoundField DataField="EmployeeID" HeaderText="Employee ID"
                          InsertVisible="False" ReadOnly="true"/>
                      <asp:BoundField DataField="FirstName" HeaderText="First Name"/>
                      <asp:BoundField DataField="LastName" HeaderText="Last Name"/>
                      <asp:BoundField DataField="Address" HeaderText="Address"/>
                      <asp:BoundField DataField="City" HeaderText="City"/>
                      <asp:BoundField DataField="Region" HeaderText="Region"/>
                      <asp:BoundField DataField="PostalCode" HeaderText="Postal Code"/>
                  </Fields>
                </asp:DetailsView>
              </td>
            </tr>
          </table>
        </form>
      </body>
    </html>
```

8. Save all files and browse object.aspx.
9. Interact with the interface by viewing details, editing employees, adding employees, and deleting employees.

Data Bound Controls

4/10/2020 • 52 minutes to read • Edit Online

by Microsoft

Most ASP.NET applications rely on some degree of data presentation from a back-end data source. Data-bound controls have been a pivotal part of interacting with data in dynamic Web applications. ASP.NET 2.0 introduces some substantial improvements to data-bound controls, including a new BaseDataBoundControl class and declarative syntax.

Most ASP.NET applications rely on some degree of data presentation from a back-end data source. Data-bound controls have been a pivotal part of interacting with data in dynamic Web applications. ASP.NET 2.0 introduces some substantial improvements to data-bound controls, including a new BaseDataBoundControl class and declarative syntax.

The BaseDataBoundControl acts as the base class for the DataBoundControl class and the HierarchicalDataBoundControl class. In this module, we will discuss the following classes that derive from DataBoundControl:

- AdRotator
- List controls
- GridView
- FormView
- DetailsView

We will also discuss the following classes that derive from HierarchicalDataBoundControl class:

- TreeView
- Menu
- SiteMapPath

DataBoundControl Class

The DataBoundControl class is an abstract class (marked MustInherit in VB) used to interact with tabular or list-style data. The following controls are some of the controls that derive from DataBoundControl.

AdRotator

The AdRotator control allows you to display a graphic banner on a Web page that is linked to a specific URL. The graphic that is displayed is rotated using properties for the control. The frequency of a particular ad displaying on a page can be configured using the **Impressions** property and ads can be filtered using keyword filtering.

AdRotator controls use either an XML file or a table in a database for data. The following attributes are used in XML files to configure the AdRotator control.

ImageUrl

The URL of an image to display for the ad.

NavigateUrl

The URL that the user should be taken to when the ad is clicked. This should be URL encoded.

AlternateText

The alternate text that is displayed in a tooltip and read by screen readers. Also displays when the image specified by ImageUrl is not available.

Keyword

Defines a keyword that can be used when using keyword filtering. If specified, only those ads with a keyword matching the keyword filter will be displayed.

Impressions

A weighting number that determines how often a particular ad is likely to appear. It is relative to the impression of other ads in the same file. The maximum value of the collective impressions for all ads in an XML file is 2,048,000,000 1.

Height

The height of the ad in pixels.

Width

The width of the ad in pixels.

> **NOTE**
>
> The Height and Width attributes override the height and width for the AdRotator control itself.

A typical XML file might look like the following:

```
<?xml version="1.0" encoding="utf-8" ?> <Advertisements xmlns="http://schemas.microsoft.com/AspNet/AdRotator-Schedule-File"> <Ad> <ImageUrl>~/images/Contoso_ad.gif</ImageUrl> <NavigateUrl>http://www.contoso-ltd.com</NavigateUrl> <AlternateText>Ad for Contoso, Ltd. Web site</AlternateText> <Impressions>100</Impressions> </Ad> <Ad> <ImageUrl>~/images/Aspnet_ad.gif</ImageUrl> <NavigateUrl>http://www.asp.net</NavigateUrl> <AlternateText>Ad for ASP.NET Web site</AlternateText> <Impressions>50</Impressions> </Ad> </Advertisements>
```

In the above example, the ad for Contoso is twice as likely to appear as the ad for the ASP.NET Web site because of the value for the Impressions attribute.

To display ads from the above XML file, add an AdRotator control to a page and set the **AdvertisementFile** property to point to the XML file as shown below:

```
<asp:AdRotator ID="AdRotator1" runat="server" AdvertisementFile="App_Data/Ads.xml" />
```

If you choose to use a database table as the data source for your AdRotator control, you will first need to set up a database using the following schema:

COLUMN NAME	DATA TYPE	DESCRIPTION
ID	int	Primary key. This column can have any name.
ImageUrl	nvarchar(*length*)	The relative or absolute URL of the image to display for the ad.
NavigateUrl	nvarchar(*length*)	The target URL for the ad. If you do not provide a value, the ad is not a hyperlink.

COLUMN NAME	DATA TYPE	DESCRIPTION
AlternateText	nvarchar(*length*)	The text displayed if the image cannot be found. In some browsers, the text is displayed as a ToolTip. Alternate text is also used for accessibility so that users who cannot see the graphic can hear its description read out loud.
Keyword	nvarchar(*length*)	A category for the ad on which the page can filter.
Impressions	int(4)	A number that indicates the likelihood of how often the ad is displayed. The larger the number, the more often the ad will be displayed. The total of all impressions values in the XML file may not exceed 2,048,000,000 - 1.
Width	int(4)	The width of the image in pixels.
Height	int(4)	The height of the image in pixels.

In cases where you already have a database with a different schema, you can use the **AlternateTextField**, **ImageUrlField**, and **NavigateUrlField** properties to map the AdRotator attributes to your existing database. To display the data from the database in the AdRotator control, add a data source control to the page, configure the connection string for the data source control to point to your database, and set the AdRotator control's **DataSourceID** property to the ID of the data source control. In cases where you have a need to configure AdRotator ads programmatically, use the AdCreated event. The AdCreated event takes two parameters; one an object, and the other an instance of AdCreatedEventArgs. The AdCreatedEventArgs is a reference to the ad that is being created.

The following code snippet sets the ImageUrl, NavigateUrl, and AlternateText for an ad programmatically:

```
protected void AdRotator1_AdCreated(object sender, System.Web.UI.WebControls.AdCreatedEventArgs e) {
e.ImageUrl = "images/contoso_ad.gif"; e.NavigateUrl = "http://www.contoso-ltd.com/"; e.AlternateText = "Ad for
Contoso, Ltd Web site"; }
```

List Controls

List controls include the ListBox, DropDownList, CheckBoxList, RadioButtonList, and BulletedList. Each of these controls can be data bound to a data source. They use one field in the data source as the display text and can optionally use a second field as the value of an item. Items can also be added statically at design-time, and you can mix static items and dynamic items added from a data source.

To data bind a list control, add a data source control to the page. Specify a SELECT command for the data source control and then set the DataSourceID property of the list control to the ID of the data source control. Use the **DataTextField** and **DataValueField** properties to define the display text and the value for the control. Additionally, you can use the **DataTextFormatString** property to control the appearance of the display text as follows:

EXPRESSION	DESCRIPTION

EXPRESSION	DESCRIPTION
Price: {0:C}	For numeric/decimal data. Displays the literal "Price:" followed by numbers in currency format. The currency format depends on the culture setting specified in the culture attribute on the **Page** directive or in the Web.config file.
{0:D4}	For integer data. Cannot be used with decimal numbers. Integers are displayed in a zero-padded field that is four characters wide.
{0:N2}%	For numeric data. Displays the number with 2-decimal place precision followed by the literal "%".
{0:000.0}	For numeric/decimal data. Numbers are rounded to one decimal place. Numbers less than three digits are zero-padded.
{0:D}	For date/time data. Displays long date format ("Thursday, August 06, 1996"). Date format depends on the culture setting of the page or the Web.config file.
{0:d}	For date/time data. Displays short date format ("12/31/99").
{0:yy-MM-dd}	For date/time data. Displays date in numeric year-month-day format (96-08-06)

GridView

The GridView control allows for tabular data display and editing using a declarative approach and is the successor to the DataGrid control. The following features are available in the GridView control.

- Binding to data source controls, such as SqlDataSource.
- Built-in sorting capabilities.
- Built-in updating and deleting capabilities.
- Built-in paging capabilities.
- Built-in row selection capabilities.
- Programmatic access to the GridView object model to dynamically set properties, handle events, and so on.
- Multiple key fields.
- Multiple data fields for the hyperlink columns.
- Customizable appearance through themes and styles.

Column Fields

Each column in the GridView control is represented by a DataControlField object. By default, the AutoGenerateColumns property is set to **true**, which creates an AutoGeneratedField object for each field in the data source. Each field is then rendered as a column in the GridView control in the order that each field appears in the data source. You can also manually control which column fields appear in the **GridView** control by setting the **AutoGenerateColumns** property to **false** and then defining your own column field collection. Different column field types determine the behavior of the columns in the control.

The following table lists the different column field types that can be used.

COLUMN FIELD TYPE	DESCRIPTION
BoundField	Displays the value of a field in a data source. This is the default column type of the GridView control.
ButtonField	Displays a command button for each item in the GridView control. This allows you to create a column of custom button controls, such as the Add or the Remove button.
CheckBoxField	Displays a check box for each item in the GridView control. This column field type is commonly used to display fields with a Boolean value.
CommandField	Displays predefined command buttons to perform selecting, editing, or deleting operations.
HyperLinkField	Displays the value of a field in a data source as a hyperlink. This column field type allows you to bind a second field to the hyperlink's URL.
ImageField	Displays an image for each item in the GridView control.
TemplateField	Displays user-defined content for each item in the GridView control according to a specified template. This column field type allows you to create a custom column field.

To define a column field collection declaratively, first add opening and closing **<Columns>** tags between the opening and closing tags of the GridView control. Next, list the column fields that you want to include between the opening and closing **<Columns>** tags. The columns specified are added to the Columns collection in the order listed. The **Columns** collection stores all the column fields in the control and allows you to programmatically manage the column fields in the GridView control.

Explicitly declared column fields can be displayed in combination with automatically generated column fields. When both are used, explicitly declared column fields are rendered first, followed by the automatically generated column fields.

Binding to Data

The GridView control can be bound to a data source control (such as **SqlDataSource**, **ObjectDataSource**, and so on), as well as any data source that implements the System.Collections.IEnumerable interface (such as System.Data.DataView, System.Collections.ArrayList, or System.Collections.Hashtable). Use one of the following methods to bind the GridView control to the appropriate data source type:

- To bind to a data source control, set the DataSourceID property of the GridView control to the ID value of the data source control. The GridView control automatically binds to the specified data source control and can take advantage of the data source control's capabilities to perform sorting, updating, deleting, and paging functionality. This is the preferred method to bind to data.
- To bind to a data source that implements the System.Collections.IEnumerable interface, programmatically set the DataSource property of the GridView control to the data source and then call the DataBind method. When using this method, the GridView control does not provide built-in sorting, updating, deleting, and paging functionality. You need to provide this functionality yourself.

Data Operations

The GridView control provides many built-in capabilities that allow the user to sort, update, delete, select, and page

through items in the control. When the GridView control is bound to a data source control, the GridView control can take advantage of the data source control's capabilities and provide automatic sorting, updating, and deleting functionality.

> **NOTE**
> The GridView control can provide support for sorting, updating, and deleting with other types of data sources; however, you will need to provide an appropriate event handler with the implementation for these operations.

Sorting allows the user to sort the items in the GridView control with respect to a specific column by clicking on the column's header. To enable sorting, set the AllowSorting property to **true**.

The automatic updating, deleting, and selection functionalities are enabled when a button in a **ButtonField** or **TemplateField** column field, with a command name of "Edit", "Delete", and "Select", respectively, is clicked. The GridView control can automatically add a **CommandField** column field with an Edit, Delete, or Select button if the AutoGenerateEditButton, AutoGenerateDeleteButton, or AutoGenerateSelectButton property is set to **true**, respectively.

> **NOTE**
> Inserting records into the data source is not directly supported by the GridView control. However, it is possible to insert records by using the GridView control in conjunction with the DetailsView or FormView control.

Instead of displaying all the records in the data source at the same time, the GridView control can automatically break the records up into pages. To enable paging, set the AllowPaging property to **true**.

Customizing the User Interface

You can customize the appearance of the GridView control by setting the style properties for the different parts of the control. The following table lists the different style properties.

STYLE PROPERTY	DESCRIPTION
AlternatingRowStyle	The style settings for the alternating data rows in the GridView control. When this property is set, the data rows are displayed alternating between the RowStyle settings and the **AlternatingRowStyle** settings.
EditRowStyle	The style settings for the row being edited in the GridView control.
EmptyDataRowStyle	The style settings for the empty data row displayed in the GridView control when the data source does not contain any records.
FooterStyle	The style settings for the footer row of the GridView control.
HeaderStyle	The style settings for the header row of the GridView control.
PagerStyle	The style settings for the pager row of the GridView control.

STYLE PROPERTY	DESCRIPTION
RowStyle	The style settings for the data rows in the GridView control. When the **AlternatingRowStyle** property is also set, the data rows are displayed alternating between the **RowStyle** settings and the **AlternatingRowStyle** settings.
SelectedRowStyle	The style settings for the selected row in the GridView control.

You can also show or hide different parts of the control. The following table lists the properties that control which parts are shown or hidden.

PROPERTY	DESCRIPTION
ShowFooter	Shows or hides the footer section of the GridView control.
ShowHeader	Shows or hides the header section of the GridView control.

Events

The GridView control provides several events that you can program against. This allows you to run a custom routine whenever an event occurs. The following table lists the events supported by the GridView control.

EVENT	DESCRIPTION
PageIndexChanged	Occurs when one of the pager buttons is clicked, but after the GridView control handles the paging operation. This event is commonly used when you need to perform a task after the user navigates to a different page in the control.
PageIndexChanging	Occurs when one of the pager buttons is clicked, but before the GridView control handles the paging operation. This event is often used to cancel the paging operation.
RowCancelingEdit	Occurs when a row's Cancel button is clicked, but before the GridView control exits edit mode. This event is often used to stop the canceling operation.
RowCommand	Occurs when a button is clicked in the GridView control. This event is often used to perform a task when a button is clicked in the control.
RowCreated	Occurs when a new row is created in the GridView control. This event is often used to modify the contents of a row when the row is created.
RowDataBound	Occurs when a data row is bound to data in the GridView control. This event is often used to modify the contents of a row when the row is bound to data.
RowDeleted	Occurs when a row's Delete button is clicked, but after the GridView control deletes the record from the data source. This event is often used to check the results of the delete operation.

EVENT	DESCRIPTION
RowDeleting	Occurs when a row's Delete button is clicked, but before the GridView control deletes the record from the data source. This event is often used to cancel the deleting operation.
RowEditing	Occurs when a row's Edit button is clicked, but before the GridView control enters edit mode. This event is often used to cancel the editing operation.
RowUpdated	Occurs when a row's Update button is clicked, but after the GridView control updates the row. This event is often used to check the results of the update operation.
RowUpdating	Occurs when a row's Update button is clicked, but before the GridView control updates the row. This event is often used to cancel the updating operation.
SelectedIndexChanged	Occurs when a row's Select button is clicked, but after the GridView control handles the select operation. This event is often used to perform a task after a row is selected in the control.
SelectedIndexChanging	Occurs when a row's Select button is clicked, but before the GridView control handles the select operation. This event is often used to cancel the selection operation.
Sorted	Occurs when the hyperlink to sort a column is clicked, but after the GridView control handles the sort operation. This event is commonly used to perform a task after the user clicks on a hyperlink to sort a column.
Sorting	Occurs when the hyperlink to sort a column is clicked, but before the GridView control handles the sort operation. This event is often used to cancel the sorting operation or to perform a custom sorting routine.

FormView

The FormView control is used to display a single record from a data source. It is similar to the DetailsView control, except it displays user-defined templates instead of row fields. Creating your own templates gives you greater flexibility in controlling how the data is displayed. The FormView control supports the following features:

- Binding to data source controls, such as SqlDataSource and ObjectDataSource.
- Built-in inserting capabilities.
- Built-in updating and deleting capabilities.
- Built-in paging capabilities.
- Programmatic access to the FormView object model to dynamically set properties, handle events, and so on.
- Customizable appearance through user-defined templates, themes, and styles.

Templates

For the FormView control to display content, you need to create templates for the different parts of the control. Most templates are optional; however, you must create a template for the mode in which the control is configured. For example, a FormView control that supports inserting records must have an insert item template defined. The following table lists the different templates that you can create.

TEMPLATE TYPE	DESCRIPTION
EditItemTemplate	Defines the content for the data row when the FormView control is in edit mode. This template usually contains input controls and command buttons with which the user can edit an existing record.
EmptyDataTemplate	Defines the content for the empty data row displayed when the FormView control is bound to a data source that does not contain any records. This template usually contains content to alert the user that the data source does not contain any records.
FooterTemplate	Defines the content for the footer row. This template usually contains any additional content you would like to display in the footer row. As an alternative, you can simply specify text to display in the footer row by setting the FooterText property.
HeaderTemplate	Defines the content for the header row. This template usually contains any additional content you would like to display in the header row. As an alternative, you can simply specify text to display in the header row by setting the HeaderText property.
ItemTemplate	Defines the content for the data row when the FormView control is in read-only mode. This template usually contains content to display the values of an existing record.
InsertItemTemplate	Defines the content for the data row when the FormView control is in insert mode. This template usually contains input controls and command buttons with which the user can add a new record.
PagerTemplate	Defines the content for the pager row displayed when the paging feature is enabled (when the AllowPaging property is set to **true**). This template usually contains controls with which the user can navigate to another record.

Input controls in the edit item template and insert item template can be bound to the fields of a data source by using a two-way binding expression. This allows the FormView control to automatically extract the values of the input control for an update or insert operation. Two-way binding expressions also allow input controls in an edit item template to automatically display the original field values.

Binding to Data

The FormView control can be bound to a data source control (such as **SqlDataSource**, AccessDataSource, **ObjectDataSource** and so on), or to any data source that implements the System.Collections.IEnumerable interface (such as System.Data.DataView, System.Collections.ArrayList, and System.Collections.Hashtable). Use one of the following methods to bind the FormView control to the appropriate data source type:

- To bind to a data source control, set the DataSourceID property of the FormView control to the ID value of the data source control. The FormView control automatically binds to the specified data source control and can take advantage of the data source control's capabilities to perform inserting, updating, deleting, and paging functionality. This is the preferred method to bind to data.
- To bind to a data source that implements the **System.Collections.IEnumerable** interface, programmatically set the DataSource property of the FormView control to the data source and then call the DataBind method.

When using this method, the FormView control does not provide built-in inserting, updating, deleting, and paging functionality. You need to provide this functionality by using the appropriate event.

Data Operations

The FormView control provides many built-in capabilities that allow the user to update, delete, insert, and page through items in the control. When the FormView control is bound to a data source control, the FormView control can take advantage of the data source control's capabilities and provide automatic updating, deleting, inserting, and paging functionality. The FormView control can provide support for update, delete, insert, and paging operations with other types of data sources; however, you must provide an appropriate event handler with the implementation for these operations.

Because the FormView control uses templates, it does not provide a way to automatically generate command buttons to perform updating, deleting, or inserting operations. You must manually include these command buttons in the appropriate template. The FormView control recognizes certain buttons that have their **CommandName** properties set to specific values. The following table lists the command buttons that the FormView control recognizes.

BUTTON	COMMANDNAME VALUE	DESCRIPTION
Cancel	"Cancel"	Used in updating or inserting operations to cancel the operation and to discard the values entered by the user. The FormView control then returns to the mode specified by the DefaultMode property.
Delete	"Delete"	Used in deleting operations to delete the displayed record from the data source. Raises the ItemDeleting and ItemDeleted events.
Edit	"Edit"	Used in updating operations to put the FormView control in edit mode. The content specified in the **EditItemTemplate** property is displayed for the data row.
Insert	"Insert"	Used in inserting operations to attempt to insert a new record in the data source using the values provided by the user. Raises the ItemInserting and ItemInserted events.
New	"New"	Used in inserting operations to put the FormView control in insert mode. The content specified in the **InsertItemTemplate** property is displayed for the data row.

BUTTON	COMMANDNAME VALUE	DESCRIPTION
Page	"Page"	Used in paging operations to represent a button in the pager row that performs paging. To specify the paging operation, set the **CommandArgument** property of the button to "Next", "Prev", "First", "Last", or the index of the page to which to navigate. Raises the PageIndexChanging and PageIndexChanged events.
Update	"Update"	Used in updating operations to attempt to update the displayed record in the data source with the values provided by the user. Raises the ItemUpdating and ItemUpdated events.

Unlike the Delete button (which deletes the displayed record immediately), when the Edit or New button is clicked, the FormView control goes into edit or insert mode respectively. In edit mode, the content contained in the **EditItemTemplate** property is displayed for the current data item. Typically, the edit item template is defined such that the Edit button is replaced with an Update and a Cancel button. Input controls that are appropriate for the field's data type (such as a TextBox or a CheckBox control) are also usually displayed with a field's value for the user to modify. Clicking the Update button updates the record in the data source, while clicking the Cancel button abandons any changes.

Likewise, the content contained in the **InsertItemTemplate** property is displayed for the data item when the control is in insert mode. The insert item template is typically defined such that the New button is replaced with an Insert and a Cancel button, and empty input controls are displayed for the user to enter the values for the new record. Clicking the Insert button inserts the record in the data source, while clicking the Cancel button abandons any changes.

The FormView control provides a paging feature, which allows the user to navigate to other records in the data source. When enabled, a pager row is displayed in the FormView control that contains the page navigation controls. To enable paging, set the **AllowPaging** property to **true**. You can customize the pager row by setting the properties of objects contained in the PagerStyle and the PagerSettings property. Instead of using the built-in pager row UI, you can create your own UI by using the **PagerTemplate** property.

Customizing the User Interface

You can customize the appearance of the FormView control by setting the style properties for the different parts of the control. The following table lists the different style properties.

STYLE PROPERTY	DESCRIPTION
EditRowStyle	The style settings for the data row when the FormView control is in edit mode.
EmptyDataRowStyle	The style settings for the empty data row displayed in the FormView control when the data source does not contain any records.
FooterStyle	The style settings for the footer row of the FormView control.
HeaderStyle	The style settings for the header row of the FormView control.

STYLE PROPERTY	DESCRIPTION
InsertRowStyle	The style settings for the data row when the FormView control is in insert mode.
PagerStyle	The style settings for the pager row displayed in the FormView control when the paging feature is enabled.
RowStyle	The style settings for the data row when the FormView control is in read-only mode.

Events

The FormView control provides several events that you can program against. This allows you to run a custom routine whenever an event occurs. The following table lists the events supported by the FormView control.

EVENT	DESCRIPTION
ItemCommand	Occurs when a button within a FormView control is clicked. This event is often used to perform a task when a button is clicked in the control.
ItemCreated	Occurs after all FormViewRow objects are created in the FormView control. This event is often used to modify the values of a record before it is displayed.
ItemDeleted	Occurs when a Delete button (a button with its **CommandName** property set to "Delete") is clicked, but after the FormView control deletes the record from the data source. This event is often used to check the results of the delete operation.
ItemDeleting	Occurs when a Delete button is clicked, but before the FormView control deletes the record from the data source. This event is often used to cancel the delete operation.
ItemInserted	Occurs when an Insert button (a button with its **CommandName** property set to "Insert") is clicked, but after the FormView control inserts the record. This event is often used to check the results of the insert operation.
ItemInserting	Occurs when an Insert button is clicked, but before the FormView control inserts the record. This event is often used to cancel the insert operation.
ItemUpdated	Occurs when an Update button (a button with its **CommandName** property set to "Update") is clicked, but after the FormView control updates the row. This event is often used to check the results of the update operation.
ItemUpdating	Occurs when an Update button is clicked, but before the FormView control updates the record. This event is often used to cancel the update operation.

EVENT	DESCRIPTION
ModeChanged	Occurs after the FormView control changes modes (to edit, insert, or read-only mode). This event is often used to perform a task when the FormView control changes modes.
ModeChanging	Occurs before the FormView control changes modes (to edit, insert, or read-only mode). This event is often used to cancel a mode change.
PageIndexChanged	Occurs when one of the pager buttons is clicked, but after the FormView control handles the paging operation. This event is commonly used when you need to perform a task after the user navigates to a different record in the control.
PageIndexChanging	Occurs when one of the pager buttons is clicked, but before the FormView control handles the paging operation. This event is often used to cancel the paging operation.

DetailsView

The DetailsView control is used to display a single record from a data source in a table, where each field of the record is displayed in a row of the table. It can be used in combination with a GridView control for master-detail scenarios. The DetailsView control supports the following features:

- Binding to data source controls, such as SqlDataSource.
- Built-in inserting capabilities.
- Built-in updating and deleting capabilities.
- Built-in paging capabilities.
- Programmatic access to the DetailsView object model to dynamically set properties, handle events, and so on.
- Customizable appearance through themes and styles.

Row Fields

Each data row in the DetailsView control is created by declaring a field control. Different row field types determine the behavior of the rows in the control. Field controls derive from DataControlField. The following table lists the different row field types that can be used.

COLUMN FIELD TYPE	DESCRIPTION
BoundField	Displays the value of a field in a data source as text.
ButtonField	Displays a command button in the DetailsView control. This allows you to display a row with a custom button control, such as an Add or a Remove button.
CheckBoxField	Displays a check box in the DetailsView control. This row field type is commonly used to display fields with a Boolean value.
CommandField	Displays built-in command buttons to perform edit, insert, or delete operations in the DetailsView control.

COLUMN FIELD TYPE	DESCRIPTION
HyperLinkField	Displays the value of a field in a data source as a hyperlink. This row field type allows you to bind a second field to the hyperlink's URL.
ImageField	Displays an image in the DetailsView control.
TemplateField	Displays user-defined content for a row in the DetailsView control according to a specified template. This row field type allows you to create a custom row field.

By default, the AutoGenerateRows property is set to **true**, which automatically generates a bound row field object for each field of a bindable type in the data source. Valid bindable types are String, DateTime, Decimal, Guid, and the set of primitive types. Each field is then displayed in a row as text, in the order in which each field appears in the data source.

Automatically generating the rows provides a quick and easy way to display every field in the record. However, to make use of the DetailsView control's advanced capabilities you must explicitly declare the row fields to include in the DetailsView control. To declare the row fields, first set the **AutoGenerateRows** property to **false**. Next, add opening and closing **<Fields>** tags between the opening and closing tags of the DetailsView control. Finally, list the row fields that you want to include between the opening and closing **<Fields>** tags. The row fields specified are added to the Fields collection in the order listed. The **Fields** collection allows you to programmatically manage the row fields in the DetailsView control.

> **NOTE**
> Automatically generated row fields are not added to the Fields collection.

Binding to Data

The DetailsView control can be bound to a data source control, such as **SqlDataSource** or AccessDataSource, or to any data source that implements the System.Collections.IEnumerable interface, such as System.Data.DataView, System.Collections.ArrayList and System.Collections.Hashtable.

Use one of the following methods to bind the DetailsView control to the appropriate data source type:

- To bind to a data source control, set the DataSourceID property of the DetailsView control to the ID value of the data source control. The DetailsView control automatically binds to the specified data source control. This is the preferred method to bind to data.
- To bind to a data source that implements the **System.Collections.IEnumerable** interface, programmatically set the DataSource property of the DetailsView control to the data source and then call the DataBind method.

Security

This control can be used to display user input, which might include malicious client script. Check any information that is sent from a client for executable script, SQL statements, or other code before displaying it in your application. ASP.NET provides an input request validation feature to block script and HTML in user input.

Data Operations

The DetailsView control provides built-in capabilities that allow the user to update, delete, insert, and page through items in the control. When the DetailsView control is bound to a data source control, the DetailsView control can take advantage of the data source control's capabilities and provide automatic updating, deleting, inserting, and

paging functionality.

The DetailsView control can provide support for update, delete, insert, and paging operations with other types of data sources; however, you must provide the implementation for these operations in an appropriate event handler.

The DetailsView control can automatically add a **CommandField** row field with an Edit, Delete, or New button by setting the AutoGenerateEditButton, AutoGenerateDeleteButton, or AutoGenerateInsertButton properties to **true**, respectively. Unlike the Delete button (which deletes the selected record immediately), when the Edit or New button is clicked, the DetailsView control goes into edit or insert mode, respectively. In edit mode, the Edit button is replaced with an Update and a Cancel button. Input controls that are appropriate for the field's data type (such as a TextBox or a CheckBox control) are displayed with a field's value for the user to modify. Clicking the Update button updates the record in the data source, while clicking the Cancel button abandons any changes. Likewise, in insert mode, the New button is replaced with an Insert and a Cancel button, and empty input controls are displayed for the user to enter the values for the new record.

The DetailsView control provides a paging feature, which allows the user to navigate to other records in the data source. When enabled, page navigation controls are displayed in a pager row. To enable paging, set the AllowPaging property to **true**. The pager row can be customized using the PagerStyle and PagerSettings properties.

Customizing the User Interface

You can customize the appearance of the DetailsView control by setting the style properties for the different parts of the control. The following table lists the different style properties.

STYLE PROPERTY	DESCRIPTION
AlternatingRowStyle	The style settings for the alternating data rows in the DetailsView control. When this property is set, the data rows are displayed alternating between the RowStyle settings and the **AlternatingRowStyle** settings.
CommandRowStyle	The style settings for the row containing the built-in command buttons in the DetailsView control.
EditRowStyle	The style settings for the data rows when the DetailsView control is in edit mode.
EmptyDataRowStyle	The style settings for the empty data row displayed in the DetailsView control when the data source does not contain any records.
FooterStyle	The style settings for the footer row of the DetailsView control.
HeaderStyle	The style settings for the header row of the DetailsView control.
InsertRowStyle	The style settings for the data rows when the DetailsView control is in insert mode.
PagerStyle	The style settings for the pager row of the DetailsView control.

STYLE PROPERTY	DESCRIPTION
RowStyle	The style settings for the data rows in the DetailsView control. When the **AlternatingRowStyle** property is also set, the data rows are displayed alternating between the **RowStyle** settings and the **AlternatingRowStyle** settings.
FieldHeaderStyle	The style settings for the header column of the DetailsView control.

Events

The DetailsView control provides several events that you can program against. This allows you to run a custom routine whenever an event occurs. The following table lists the events supported by the DetailsView control. The DetailsView control also inherits these events from its base classes: DataBinding, DataBound, Disposed, Init, Load, PreRender, and Render.

EVENT	DESCRIPTION
ItemCommand	Occurs when a button is clicked in the DetailsView control.
ItemCreated	Occurs after all DetailsViewRow objects are created in the DetailsView control. This event is often used to modify the values of a record before it is displayed.
ItemDeleted	Occurs when a Delete button is clicked, but after the DetailsView control deletes the record from the data source. This event is often used to check the results of the delete operation.
ItemDeleting	Occurs when a Delete button is clicked, but before the DetailsView control deletes the record from the data source. This event is often used to cancel the delete operation.
ItemInserted	Occurs when an Insert button is clicked, but after the DetailsView control inserts the record. This event is often used to check the results of the insert operation.
ItemInserting	Occurs when an Insert button is clicked, but before the DetailsView control inserts the record. This event is often used to cancel the insert operation.
ItemUpdated	Occurs when an Update button is clicked, but after the DetailsView control updates the row. This event is often used to check the results of the update operation.
ItemUpdating	Occurs when an Update button is clicked, but before the DetailsView control updates the record. This event is often used to cancel the update operation.
ModeChanged	Occurs after the DetailsView control changes modes (edit, insert, or read-only mode). This event is often used to perform a task when the DetailsView control changes modes.

EVENT	DESCRIPTION
ModeChanging	Occurs before the DetailsView control changes modes (edit, insert, or read-only mode). This event is often used to cancel a mode change.
PageIndexChanged	Occurs when one of the pager buttons is clicked, but after the DetailsView control handles the paging operation. This event is commonly used when you need to perform a task after the user navigates to a different record in the control.
PageIndexChanging	Occurs when one of the pager buttons is clicked, but before the DetailsView control handles the paging operation. This event is often used to cancel the paging operation.

The Menu Control

The Menu control in ASP.NET 2.0 is designed to be a full-featured navigation system. It can be databound easily to hierarchical data sources such as the SiteMapDataSource.

A Menu controls structure can be defined declaratively or dynamically and consists of a single root node and any number of sub-nodes. The following code declaratively defines a menu for the Menu control.

```
<asp:menu id="NavigationMenu" disappearafter="2000" staticdisplaylevels="2" staticsubmenuindent="10"
orientation="Vertical" font-names="Arial" target="_blank" runat="server"> <staticmenuitemstyle
backcolor="LightSteelBlue" forecolor="Black"/> <statichoverstyle backcolor="LightSkyBlue"/>
<dynamicmenuitemstyle backcolor="Black" forecolor="Silver"/> <dynamichoverstyle backcolor="LightSkyBlue"
forecolor="Black"/> <items> <asp:menuitem navigateurl="Home.aspx" text="Home" tooltip="Home"> <asp:menuitem
navigateurl="Music.aspx" text="Music" tooltip="Music"> <asp:menuitem navigateurl="Classical.aspx"
text="Classical" tooltip="Classical"/> <asp:menuitem navigateurl="Rock.aspx" text="Rock" tooltip="Rock"/>
<asp:menuitem navigateurl="Jazz.aspx" text="Jazz" tooltip="Jazz"/> </asp:menuitem> <asp:menuitem
navigateurl="Movies.aspx" text="Movies" tooltip="Movies"> <asp:menuitem navigateurl="Action.aspx"
text="Action" tooltip="Action"/> <asp:menuitem navigateurl="Drama.aspx" text="Drama" tooltip="Drama"/>
<asp:menuitem navigateurl="Musical.aspx" text="Musical" tooltip="Musical"/> </asp:menuitem> </asp:menuitem>
</items> </asp:menu>
```

In the above example, the Home.aspx node is the root node. All other nodes are nested within the root node at various levels.

There are two types of menus that the Menu control can render; static menus and dynamic menus. Static menus consist of menu items that are always visible. Dynamic menus consist of menu items that are only visible when the user hovers over them with the mouse. Customers may often confuse static menus with menus defined declaratively and dynamic menus with menus that are databound at runtime. In fact, dynamic and static menus are unrelated to the method of population. The terms *static* and *dynamic* refer only to whether or not the menu is statically displayed by default or only displayed when the user takes some action.

The **StaticDisplayLevels** property is used to configure how many levels of the menu are static and therefore displayed by default. In the above example, setting the **StaticDisplayLevels** property to a value of 2 would cause the menu to statically display the Home node, the Music node, and the Movies node. All other nodes would be dynamically displayed when the user hovers over the parent node.

The **MaximumDynamicDisplayLevels** property configures the maximum number of dynamic levels the menu is capable of displaying. Any dynamic menus at a level higher than the value specified by the **MaximumDynamicDisplayLevels** property are discarded.

> **NOTE**
> It is almost certain that you might encounter situations where menus don't appear to render due to the MaximumDynamicDisplayLevels property. In those cases, ensure that the property is set sufficiently to allow for the display of the customers menus.

Data Binding the Menu Control

The Menu control can be bound to any hierarchical data source such as the SiteMapDataSource or the XMLDataSource. The SiteMapDataSource is the most commonly used method for data binding to a Menu control because it feeds off of the Web.sitemap file and its schema provides a known API to the Menu control. The listing below shows a simple Web.sitemap file.

```
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" > <siteMapNode url="~/default.aspx"
description="Our Home Page" title="Home"> <siteMapNode url="~/products/default.aspx" title="Products"
description="Our Products"> <siteMapNode url="~/products/winprods.aspx" title="Windows Products"
description="Windows Products" /> <siteMapNode url="~/products/webprods.aspx" title="Web Products"
description="Web Products" /> </siteMapNode> <siteMapNode url="~/services/default.aspx" title="Services"
description="Our Services"> <siteMapNode url="~/services/consulting.aspx" title="Consulting Services"
description="Consulting Services" /> <siteMapNode url="~/services/develop.aspx" title="Development Services"
description="Development Services" /> </siteMapNode> </siteMapNode> </siteMap>
```

Notice that there is only one root siteMapNode element, in this case, the Home element. Several attributes can be configured for each siteMapNode. The most commonly used attributes are:

- **url** Specifies the URL to display when a user clicks the menu item. If this attribute is not present, the node will simply post back when clicked.
- **title** Specifies the text that is displayed on the menu item.
- **description** Used as documentation for the node. Also displays as a tool tip when the mouse is hovered over the node.
- **siteMapFile** Allows for nested sitemaps. This attribute must point to a well-formed ASP.NET sitemap file.
- **roles** Allows for the appearance of a node to be controlled by ASP.NET security trimming.

Note that while these attributes are all optional, the behavior of the menu may not be what is expected if they are not specified. For example, if the *url* attribute is specified but the *description* attribute is not, the node will not be visible and there will be no way to navigate to the URL specified.

Controlling a Menus Operation

There are several properties that affect the operation of an ASP.NET Menu control; the **Orientation** property, the **DisappearAfter** property, the **StaticItemFormatString** property, and the **StaticPopoutImageUrl** property are just a few of these.

- The **Orientation** can be set to either *horizontal* or *vertical* and controls whether static menu items are laid out horizontally in a row or vertically and stacked upon one another. This property does not affect dynamic menus.
- The **DisappearAfter** property configures how long a dynamic menu should remain visible after the mouse has been moved away from it. The value is specified in milliseconds and defaults to 500. Setting this property to a value of -1 will cause the menu to never disappear automatically. In that case, the menu will only disappear when the user clicks outside of the menu.
- The **StaticItemFormatString** property makes it easy to maintain consistent verbiage in your menu system. When specifying this property, *{0}* should be entered in place of the description that appears in the data source. For example, in order to have the menu item from exercise 1 say Visit Our Products Page, etc., you would specify Visit Our {0} Page for the StaticItemFormatString. At runtime, ASP.NET will replace any occurrence of {0}

with the correct description for the menu item.

- The **StaticPopoutImageUrl** property specifies the image that is used to indicate that a particular menu node has child nodes that can be accessed by hovering over it. Dynamic menus will continue to use the default image.

Templated Menu Controls

The Menu control is a templated control and allows for two different ItemTemplates; the StaticItemTemplate and the DynamicItemTemplate. Using these templates, you can easily add server controls or user controls to your menus.

To edit the templates in Visual Studio .NET, click the Smart Tag button on the menu and either choose Edit Templates. You can then choose between editing the StaticItemTemplate or the DynamicItemTemplate.

Any controls added to the StaticItemTemplate will appear in the static menu when the page loads. Any controls added to the DynamicItemTemplate will appear on all pop-up menus.

Menu Events

The Menu control has two events that are unique to it; the **MenuItemClicked** and the **MenuItemDatabound** event.

The MenuItemClicked event is raised when a menu item is clicked. The MenuItemDatabound event is raised when a menu item is databound. The **MenuEventArgs** that is passed to the event handler provides access to the menu item via the Item property.

Controlling a Menus Appearance

You can also affect the appearance of a menu control using one or more of the many styles available to format menus. Among these are **StaticMenuStyle**, **DynamicMenuStyle**, **DynamicMenuItemStyle**, **DynamicSelectedStyle**, and **DynamicHoverStyle**. These properties are configured using a standard HTML Style string. For example, the following will affect the style for dynamic menus.

```
<DynamicMenuStyle BorderStyle="Outset" BorderWidth="1px"
    BackColor="#D6D3CE" />
```

> **NOTE**
> If you are using any of the Hover styles, you will need to add a <head> element in to the page with the *runat* element set to *server*.

Menu controls also support the use of ASP.NET 2.0 themes.

The TreeView Control

The TreeView control displays data in a tree-like structure. As with the Menu control, it can be easily data bound to any hierarchical data source such as the SiteMapDataSource.

The first question that customers are likely to ask about the TreeView control in ASP.NET 2.0 is whether or not it is related to the TreeView IE WebControl that was available for ASP.NET 1.x. It is not. The ASP.NET 2.0 TreeView control was written from the ground up and offers significant improvement over the IE TreeView WebControl that was previously available.

I won't go into detail on how to bind a TreeView control to a site map as it is performed in exactly the same way as

the Menu control. However, the TreeView control has some distinct differences in the way that it operates.

By default, a TreeView control appears fully expanded. To change the level of expansion upon initial load, modify the **ExpandDepth** property of the control. This is particularly important in cases where the TreeView is databound upon expanding particular nodes.

DataBinding the TreeView Control

Unlike the Menu control, the TreeView lends itself well to handling large amounts of data. Therefore, in addition to databinding to a SiteMapDataSource or XMLDataSource, the TreeView is often data bound to a DataSet or other relational data. In cases where the TreeView control is bound to large amounts of data, it is best to bind only to data that is actually visible in the control. You can then data bind to additional data as TreeView nodes are expanded.

In these cases, the **PopulateOnDemand** property of the TreeView should be set to *true*. You will then need to provide an implementation for the **TreeNodePopulate** method.

Data Binding Without PostBack

Notice that when you expand a node in the previous example for the first time, the page posts back and refreshes. Thats not a problem in this example, but you can imagine that it might be in a production environment with a large amount of data. A better scenario would be one in which the TreeView would still dynamically populate its nodes, but without a post back to the server.

By setting the **PopulateNodesFromClient** and the **PopulateOnDemand** properties to true, the ASP.NET TreeView control will dynamically populate nodes without a post back. When the parent node is expanded, an XMLHttp request is made from the client and the OnTreeNodePopulate event is fired. The server responds with an XML data island which is then used to data bind the child nodes.

ASP.NET dynamically creates the client code that implements this functionality. The <script> tags that contain the script are generated pointing to an AXD file. For example, the listing below shows the script links for the script code that generates the XMLHttp request.

```
<script src="/WebSite1/WebResource.axd?d=QQG9vSdBi4_k2y9AeP_iww2&
    amp;t=632514255396426531"
    type="text/javascript"></script>
```

If you browse the AXD file above in your browser and open it, you will see the code that implements the XMLHttp request. This method prevents customers from modifying the script file.

Controlling the Operation of the TreeView Control

The TreeView control has several properties that affect the operation of the control. The most obvious properties are the **ShowCheckBoxes**, **ShowExpandCollapse**, and **ShowLines**.

The **ShowCheckBoxes** property affects whether or not nodes display a check box when rendered. The valid values for this property are **None**, **Root**, **Parent**, **Leaf**, and **All**. These affect the TreeView control as follows:

PROPERTY VALUE	EFFECT
None	Checkboxes are not displayed on any nodes. This is the default setting.
Root	A checkbox is only displayed on the root node.

PROPERTY VALUE	EFFECT
Parent	A checkbox is only displayed on those nodes that have child nodes. Those child nodes can be parent nodes or leaf nodes.
Leaf	A checkbox is displayed only on those nodes that have no child nodes.
All	A checkbox is displayed on all nodes.

When checkboxes are being used, the **CheckedNodes** property will return a collection of TreeView nodes that are checked upon postback.

The **ShowExpandCollapse** property controls the appearance of the expand/collapse image next to root and parent nodes. If this property is set to **false**, TreeView nodes are rendered as hyperlinks and are expanded/collapsed by clicking on the link.

The **ShowLines** property controls whether or not lines are displayed connecting parent nodes to child nodes. When **false** (the default), no lines are displayed. When **true**, the TreeView control will use lines images in the folder specified by the **LineImagesFolder** property.

To customize the appearance of TreeView lines, Visual Studio .NET 2005 includes a Line Designer tool. You can access this tool using the Smart Tag button on the TreeView control as below.

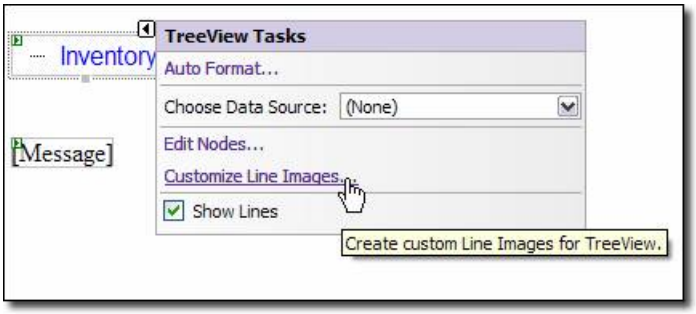


Figure 1

When select the **Customize Line Images** menu option, the Line Designer tool will be launched allowing you to configure the appearance of TreeView lines.

TreeView Events

The TreeView control has the following unique events:

- SelectedNodeChanged Occurs when a node is selected based upon the **SelectAction** property.
- TreeNodeCheckChanged Occurs when a nodes checkboxs state is changed.
- TreeNodeExpanded Occurs when a node is expanded based upon the **SelectAction** property.
- TreeNodeCollapsed Occurs when a node is collapsed.
- TreeNodeDataBound Occurs when a node is data bound.
- TreeNodePopulate Occurs when a node is populated.

The **SelectAction** property allows you to configure which event is fired when a node is selected. The SelectAction property provides the following actions:

- TreeNodeSelectAction.Expand Raises TreeNodeExpanded when the node is selected.
- TreeNodeSelectAction.None Raises no event when the node is selected.
- TreeNodeSelectAction.Select Raises the SelectedNodeChanged event when the node is selected.

- TreeNodeSelectAction.SelectExpand Raises both the SelectedNodeChanged event and the TreeNodeExpanded event when the node is selected.

Controlling Appearance with Styles

The TreeView control provides many properties for controlling the appearance of the control with styles. The following properties are available.

PROPERTY NAME	CONTROLS
HoverNodeStyle	Controls the style of nodes when the mouse is hovered over them.
LeafNodeStyle	Controls the style of leaf nodes.
NodeStyle	Controls the style for all nodes. Specific node styles (such as LeafNodeStyle) override this style.
ParentNodeStyle	Controls the style for all parent nodes.
RootNodeStyle	Controls the style for the root node.
SelectedNodeStyle	Controls the style for the selected node.

Each of these properties is read-only. However, they will each return a **TreeNodeStyle** object, and the properties of that object can be modified using the *property-subproperty* format. For example, to set the **ForeColor** property of the **SelectedNodeStyle**, you would use the following syntax:

```
<asp:TreeView id=sampleTreeView
    SelectedNodeStyle-ForeColor=Blue
    runat=server>
```

Notice that the tag above is not closed. That is because when using the declarative syntax shown here, you would include the TreeViews nodes in the HTML code as well.

The style properties can also be specified in code using the *property.subproperty* format. For example, to set the **ForeColor** property of the **RootNodeStyle** in code, you would use the following syntax:

```
treeView1.RootNodeStyle.ForeColor = System.Drawing.Color.Red;
```

> **NOTE**
> For a comprehensive list of the different style properties, see the MSDN documentation on the TreeNodeStyle object.

The SiteMapPath Control

The SiteMapPath control provides a bread crumb navigation control for ASP.NET developers. Like the other navigation controls, it can be easily data bound to hierarchical data sources such as the SiteMapDataSource or XmlDataSource.

A SiteMapPath control is made up of SiteMapNodeItem objects. There are three types of nodes; the Root node, Parent nodes, and the Current node. The root node is the node at the top of the hierarchical structure. The current node represents the current page. All other nodes are parent nodes.

Controlling the Operation of the SiteMapPath Control

The properties that control the operation of the SiteMapPath control are as follows:

PROPERTY	DESCRIPTION OF PROPERTY
ParentLevelsDisplayed	Controls how many parent nodes are displayed. The default is -1 which imposes no restriction on the number of parent nodes displayed.
PathDirection	Controls the direction of the SiteMapPath. Valid values are RootToCurrent (default) and CurrentToRoot.
PathSeparator	A String that controls the character that separates nodes in a SiteMapPath control. The default value is :.
RenderCurrentNodeAsLink	A Boolean value that controls whether or not the current node is rendered as a link. The default is False.
SkipLinkText	Assists with accessibility when the page is viewed by screen readers. This property allows screen readers to skip the SiteMapPath control. To disable this feature, set the property to String.Empty.

Templated SiteMapPath Controls

The SiteMapControl is a templated control, and as such, you can define different templates for use in displaying the control. To edit templates in a SiteMapPath control, click the Smart Tag button on the control and choose Edit Templates from the menu. This displays the SiteMapTasks menu as shown below where you can choose between the different templates available.

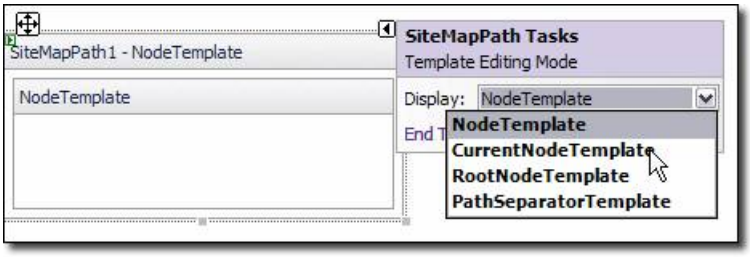


Figure 2

The **NodeTemplate** template refers to any node in the SiteMapPath. If the node is a root node or the current node and a **RootNodeTemplate** or **CurrentNodeTemplate** is configured, the NodeTemplate is overridden.

SiteMapPath Events

The SiteMapPath control has two events that are not derived from the Control class; the **ItemCreated** event and the **ItemDataBound** event. The ItemCreated event is raised when a SiteMapPath item is created. ItemDataBound is raised when the DataBind method is called during the data binding of a SiteMapPath node. A **SiteMapNodeItemEventArgs** object provides access to the specific SiteMapNodeItem via the Item property.

Controlling Appearance with Styles

The following styles are available for formatting of a SiteMapPath control.

PROPERTY NAME	CONTROLS
CurrentNodeStyle	Controls the style of the text for the current node.
RootNodeStyle	Controls the style of the text for the root node.
NodeStyle	Controls the style of the text for all nodes assuming that a CurrentNodeStyle or RootNodeStyle doesnt apply.

The NodeStyle property is overridden by either the CurrentNodeStyle or the RootNodeStyle. Each of these properties is read-only and returns a **Style** object. To affect the appearance of a node using one of these properties, you will need to set the properties of the Style object that is returned. For example, the code below changes the forecolor property of the current node.

```
<asp:SiteMapPath runat="server" ID="SiteMapPath1"
  CurrentNodeStyle-ForeColor="Orange"/>
```

The property can also be applied programmatically as follows:

```
this.SiteMapPath1.CurrentNodeStyle.ForeColor =
    System.Drawing.Color.Orange;
```

NOTE

If a template is applied, the style will not be applied.

Lab 1: Configuring an ASP.NET Menu Control

1. Create a new Web site.
2. Add a Site Map file by selecting File, New, File and choosing Site Map from the list of file templates.
3. Open the site map (Web.sitemap by default) and modify it so that it looks like the listing below. The pages to which you are linking in the site map file do not really exist, but that will not be an issue for this exercise.

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
    <siteMapNode url="~/default.aspx">
        <siteMapNode url="~/products/default.aspx"
          title="Products" description="Our Products">
            <siteMapNode url="~/products/winprods.aspx"
              title="Windows Products" description="Windows Products" />
            <siteMapNode url="~/products/webprods.aspx"
              title="Web Products" description="Web Products" />
        </siteMapNode>
        <siteMapNode url="~/services/default.aspx"
          title="Services" description="Our Services">
            <siteMapNode url="~/services/consulting.aspx"
              title="Consulting Services"
              description="Consulting Services" />
            <siteMapNode url="~/services/develop.aspx"
              title="Development Services"
              description="Development Services" />
        </siteMapNode>
    </siteMapNode>
</siteMap>
```

4. Open the default Web form in Design view.
5. From the Navigation section of the Toolbox, add a new Menu control to the page.
6. From the Data section of the Toolbox, add a new SiteMapDataSource. The SiteMapDataSource will automatically use the Web.sitemap file in your site. (The Web.sitemap file *must* be in the root folder of the site.)
7. Click the Menu control and then click the Smart Tag button to display the Menu Tasks dialog.
8. In the Choose Data Source dropdown, select SiteMapDataSource1.
9. Click the AutoFormat link and choose a format for the Menu.
10. In the Properties pane, set the **StaticDisplayLevels** property to 2. The Menu control should now display the Home, Products, and Services node in the Designer.
11. Browse the page in your browser to use the menu. (Because the pages you ve added to the site map don't actually exist, you will see an error when you try and browse to them.)

Experiment with changing the StaticDisplayLevels and the MaximumDynamicDisplayLevels properties and see how they affect how the menu is rendered.

Lab 2: Dynamically Binding a TreeView Control

This exercise assumes that you have SQL Server running locally and that the Northwind database is present on the SQL Server instance. If these conditions are not met, please change the connection string in the sample. Note that you may also need to specify SQL Server authentication instead of a trusted connection.

1. Create a new Web site.
2. Switch to Code view for Default.aspx and replace all code with the code listed below.

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<script runat="server">
    void PopulateNode(Object sender, TreeNodeEventArgs e) {
        // Call the appropriate method to populate a node at a particular level.
        switch (e.Node.Depth) {
            case 0:
                // Populate the first-level nodes.
                PopulateCategories(e.Node);
                break;
            case 1:
                // Populate the second-level nodes.
                PopulateProducts(e.Node);
                break;
            default:
                // Do nothing.
                break;
        }

    }
    void PopulateCategories(TreeNode node) {

        // Query for the product categories. These are the values
        // for the second-level nodes.
        DataSet ResultSet = RunQuery("Select CategoryID, CategoryName From Categories");
        // Create the second-level nodes.
        if (ResultSet.Tables.Count > 0) {

            // Iterate through and create a new node for each row in the query results.
```

```
            // Notice that the query results are stored in the table of the DataSet.
            foreach (DataRow row in ResultSet.Tables[0].Rows) {

                // Create the new node. Notice that the CategoryId is stored in the Value property
                // of the node. This will make querying for items in a specific category easier when
                // the third-level nodes are created.
                TreeNode newNode = new TreeNode();
                newNode.Text = row["CategoryName"].ToString();
                newNode.Value = row["CategoryID"].ToString();
                // Set the PopulateOnDemand property to true so that the child nodes can be
                // dynamically populated.
                newNode.PopulateOnDemand = true;

                // Set additional properties for the node.
                newNode.SelectAction = TreeNodeSelectAction.Expand;

                // Add the new node to the ChildNodes collection of the parent node.
                node.ChildNodes.Add(newNode);

            }

        }

    }
    void PopulateProducts(TreeNode node) {
        // Query for the products of the current category. These are the values
        // for the third-level nodes.
        DataSet ResultSet = RunQuery("Select ProductName From Products Where CategoryID=" +
node.Value);
        // Create the third-level nodes.
        if (ResultSet.Tables.Count > 0) {

            // Iterate through and create a new node for each row in the query results.
            // Notice that the query results are stored in the table of the DataSet.
            foreach (DataRow row in ResultSet.Tables[0].Rows) {

                // Create the new node.
                TreeNode NewNode = new TreeNode(row["ProductName"].ToString());

                // Set the PopulateOnDemand property to false, because these are leaf nodes and
                // do not need to be populated.
                NewNode.PopulateOnDemand = false;

                // Set additional properties for the node.
                NewNode.SelectAction = TreeNodeSelectAction.None;

                // Add the new node to the ChildNodes collection of the parent node.
                node.ChildNodes.Add(NewNode);

            }

        }
    }
    DataSet RunQuery(String QueryString) {
        // Declare the connection string. This example uses Microsoft SQL Server
        // and connects to the Northwind sample database.
        String ConnectionString = "server=localhost;database=NorthWind;Integrated Security=SSPI";
        SqlConnection DBConnection = new SqlConnection(ConnectionString);
        SqlDataAdapter DBAdapter;
        DataSet ResultsDataSet = new DataSet();
        try {
            // Run the query and create a DataSet.
            DBAdapter = new SqlDataAdapter(QueryString, DBConnection);
            DBAdapter.Fill(ResultsDataSet);
            // Close the database connection.
            DBConnection.Close();
        } catch (Exception ex) {
            // Close the database connection if it is still open.
            if (DBConnection.State == ConnectionState.Open) {
```

```
                DBConnection.Close();
            }

            Message.Text = "Unable to connect to the database.";
        }
        return ResultsDataSet;
    }
</script>

<html>
<body>
    <form id="Form1" runat="server">
        <h3>
            TreeView PopulateNodesFromClient Example</h3>
        <asp:TreeView ID="LinksTreeView"
          Font-Name="Arial" ForeColor="Blue" EnableClientScript="true"
          PopulateNodesFromClient="false"
          OnTreeNodePopulate="PopulateNode" runat="server"
            ExpandDepth="0">
            <Nodes>
                <asp:TreeNode Text="Inventory" SelectAction="Expand"
                              PopulateOnDemand="True" Value="Inventory" />
            </Nodes>
        </asp:TreeView>
        <br>
        <br>
        <asp:Label ID="Message" runat="server" />
    </form>
</body>
</html>
```

3. Save the page as treeview.aspx.
4. Browse the page.
5. When the page is first displayed, view the source of the page in your browser. Note that only the visible nodes were sent to the client.
6. Click the plus sign next to any node.
7. View source on the page again. Notice that the newly displayed nodes are now present.

Lab 3: Details View and Editing Data Using a GridView and DetailsView

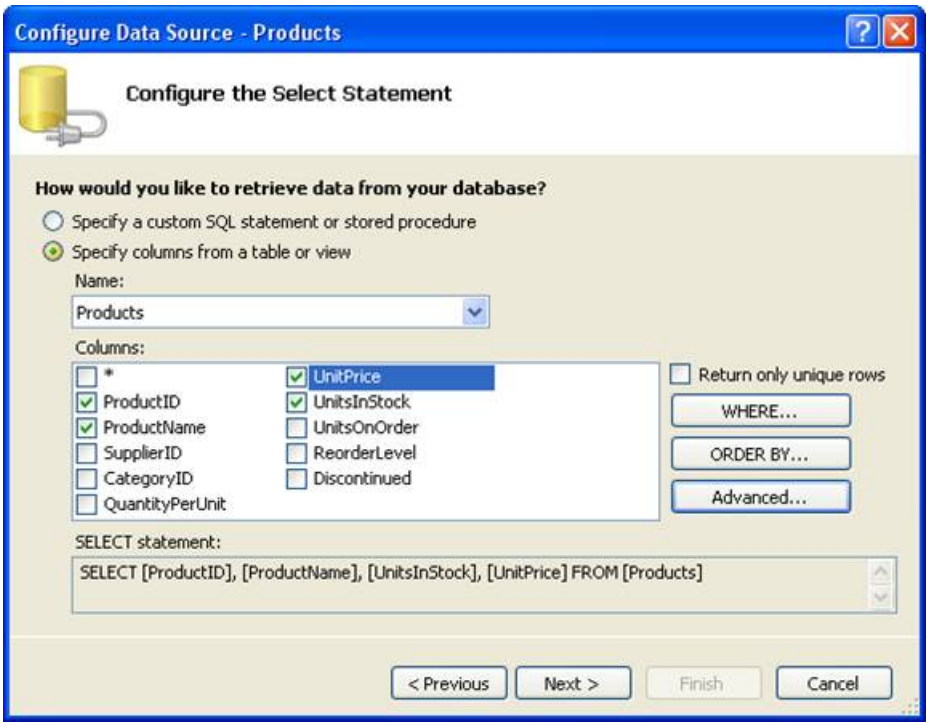
1. Create a new Web site.
2. Add a new web.config to the Web site.
3. Add a connection string to the web.config file as shown below:

```
<connectionStrings>
    <add name="Northwind"
         providerName="System.Data.SqlClient"
         connectionString="Data Source=localhost;Integrated Security=SSPI;
         Initial Catalog=Northwind;"/>
</connectionStrings>
```

> **NOTE**
>
> You may need to change the connection string based upon your environment.

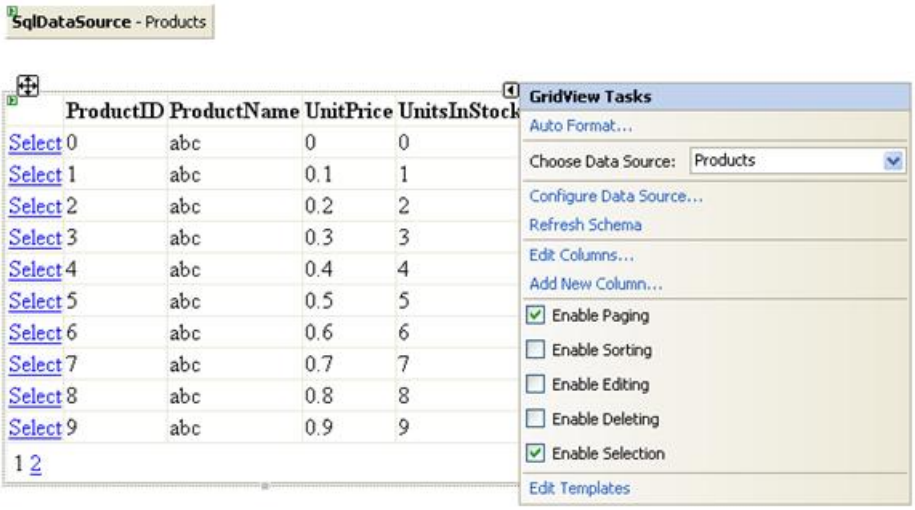
4. Save and close the web.config file.

5. Open Default.aspx and add a new SqlDataSource control.
6. Change the ID of the SqlDataSource control to **Products**.
7. In the **SqlDataSource Tasks** menu, click **Configure Data Source**.
8. Select **Northwind** in the connection dropdown and click Next.
9. Select **Products** from the **Name** dropdown and check the **ProductID**, **ProductName**, **UnitPrice**, and **UnitsInStock** checkboxes as shown below.



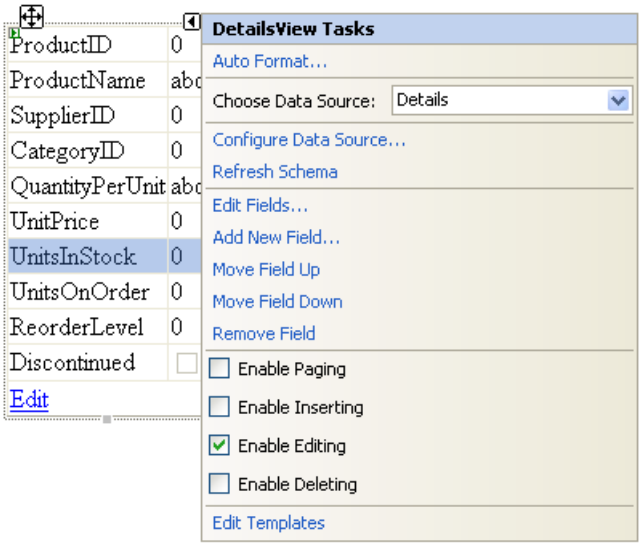

```
**Figure 3**
```

10. Click **Next**.
11. Click **Finish**.
12. Switch to Source view and examine the code that was generated. Notice the **SelectCommand**, **DeleteCommand**, **InsertCommand**, and **UpdateCommand** that were added to the SqlDataSource control. Also notice the parameters that were added.
13. Switch to Design view and add a new GridView control to the page.
14. Select **Products** from the **Choose Data Source** dropdown.
15. Check **Enable Paging** and **Enable Selection** as shown below.




```
**Figure 4**
```

16. Click the **Edit Columns** link and make sure that **Auto-generate fields** is checked.
17. Click **OK**.
18. With the GridView control selected, click the button next to the **DataKeyNames** property in the Properties pane.
19. Select **ProductID** from the **Available data fields** list and click the > button to add it.
20. Click OK.
21. Add a new SqlDataSource control to the page.
22. Change the ID of the SqlDataSource control to **Details**.
23. From the SqlDataSource Tasks menu, choose **Configure Data Source**.
24. Choose **Northwind** from the dropdown and click **Next**.
25. Select **Products** from the **Name** dropdown and check the *** checkbox in the Columns listbox.**
26. Click the **WHERE** button.
27. Select **ProductID** from the **Column** dropdown.
28. Select = in the Operator dropdown.
29. Select **Control** from the **Source** dropdown.
30. Select **GridView1** from the **Control ID** dropdown.
31. Click the **Add** button to add the WHERE clause.
32. Click **OK**.
33. Click the **Advanced** button and check the **Generate INSERT, UPDATE, and DELETE statements** checkbox.
34. Click **OK**.
35. Click **Next** and click **Finish**.
36. Add a DetailsView control to the page.
37. In the **Choose Data Source** dropdown, choose **Details**.
38. Check the **Enable Editing** checkbox as shown below.




```
**Figure 5**
```

39. Save the page and browse Default.aspx.
40. Click the **Select** link next to different records to see the DetailsView update automatically.
41. Click the **Edit** link in the DetailsView control.
42. Make a change to the record and click **Update**.

Caching

4/10/2020 • 18 minutes to read • Edit Online

by Microsoft

> An understanding of caching is important for a well-performing ASP.NET application. ASP.NET 1.x offered three different options for caching; output caching, fragment caching, and the cache API.

An understanding of caching is important for a well-performing ASP.NET application. ASP.NET 1.x offered three different options for caching; output caching, fragment caching, and the cache API. ASP.NET 2.0 offers all three of these methods, but it adds some significant additional features. There are several new cache dependencies and developers now have the option to create custom cache dependencies as well. The configuration of caching has also been improved significantly in ASP.NET 2.0.

New Features

Cache Profiles

Cache profiles allow developers to define specific cache settings that can then be applied to individual pages. For example, if you have some pages that should be expired from cache after 12 hours, you can easily create a cache profile that can be applied to those pages. To add a new cache profile, use the <outputCacheSettings> section in the configuration file. For example, below is the configuration of a cache profile called *twoday* that configures a cache duration of 12 hours.

```
<outputCacheSettings>
    <outputCacheProfiles>
        <add name="TwoDay" duration="43200" />
    </outputCacheProfiles>
</outputCacheSettings>
```

To apply this cache profile to a particular page, use the CacheProfile attribute of the @ OutputCache directive as shown below:

```
<%@ OutputCache CacheProfile="TwoDay" %>
```

Custom Cache Dependencies

ASP.NET 1.x developers cried out for custom cache dependencies. In ASP.NET 1.x, the CacheDependency class was sealed which prevented developers from deriving their own classes from it. In ASP.NET 2.0, that limitation is removed and developers are free to develop their own custom cache dependencies. The CacheDependency class allows for the creation of a custom cache dependency based on files, directories, or cache keys.

For example, the code below creates a new custom cache dependency based on a file called stuff.xml located in the root of the Web application:

```
System.Web.Caching.CacheDependency dep = new
    System.Web.Caching.CacheDependency(Server.MapPath("stuff.xml"));
Response.AddCacheDependency(dep);
Cache.Insert("key", "value");
```

In this scenario, when the stuff.xml file changes, the cached item is invalidated.

It is also possible to create a custom cache dependency using cache keys. Using this method, the removal of the cache key will invalidate the cached data. The following example illustrates this:

```
// insert a value into cache that will serve
// as the cache key
Cache["CacheKey"] = "something";

// create an array of cache keys
string[] keys = new String[1];
keys[0] = "CacheKey";

CacheDependency dep = new CacheDependency(null, keys);

// insert an item into cache with a dependency on
// the above CacheDependency
Cache.Insert("Key", "Value", dep);
```

To invalidate the item that was inserted above, simply remove the item that was inserted into cache to act as the cache key.

```
// Remove the cache item that serves as the cache key
Cache.Remove("CacheKey");
```

Note that the key of the item that acts as the cache key must be the same as the value added to the array of cache keys.

Polling-Based SQL Cache Dependencies(Also called Table-Based Dependencies)

SQL Server 7 and 2000 use the polling-based model for SQL cache dependencies. The polling-based model uses a trigger on a database table that is triggered when data in the table change. That trigger updates a **changeId** field in the notification table that ASP.NET checks periodically. If the **changeId** field has been updated, ASP.NET knows that the data have changed and it invalidates the cached data.

> **NOTE**
>
> SQL Server 2005 can also use the polling-based model, but because the polling-based model is not the most efficient model, it is advisable to use a query-based model (discussed later) with SQL Server 2005.

In order for a SQL cache dependency using the polling-based model to work correctly, the tables must have notifications enabled. This can be accomplished programmatically using the SqlCacheDependencyAdmin class or by using the aspnet_regsql.exe utility.

The following command line registers the Products table in the Northwind database located on a SQL Server instance named *dbase* for SQL cache dependency.

```
aspnet_regsql -S dbase -ed -d Northwind -E -et -t Products
```

The following is an explanation of the command line switches used in the above command:

COMMAND LINE SWITCH	PURPOSE
-S *server*	Specifies the server name.
-ed	Specifies that the database should be enabled for SQL cache dependency.
-d *database_name*	Specifies the database name that should be enabled for SQL cache dependency.
-E	Specifies that aspnet_regsql should use Windows authentication when connecting to the database.
-et	Specifies that we are enabling a database table for SQL cache dependency.
-t *table_name*	Specifies the name of the database table to enable for SQL cache dependency.

> **NOTE**
> There are other switches available for aspnet_regsql.exe. For a complete list, run aspnet_regsql.exe -? from a command line.

When this command runs the following changes are made to the SQL Server database:

- An **AspNet_SqlCacheTablesForChangeNotification** table is added. This table contains one row for each table in the database for which a SQL cache dependency has been enabled.
- The following stored procedures are created inside of the database:

ASPNET_SQLCACHEPOLLINGSTOREDPROCEDURE	QUERIES THE ASPNET_SQLCACHETABLESFORCHANGENOTIFICATION TABLE AND RETURNS ALL TABLES THAT ARE ENABLED FOR SQL CACHE DEPENDENCY AND THE VALUE OF CHANGEID FOR EACH TABLE. THIS STORED PROC IS USED FOR POLLING TO DETERMINE IF DATA HAVE CHANGED.
AspNet_SqlCacheQueryRegisteredTablesStoredProcedure	Returns all of the tables enabled for SQL cache dependency by querying the AspNet_SqlCacheTablesForChangeNotification table and returns all tables enabled for SQL cache dependency.
AspNet_SqlCacheRegisterTableStoredProcedure	Registers a table for SQL cache dependency by adding the necessary entry in the notification table and adds the trigger.
AspNet_SqlCacheUnRegisterTableStoredProcedure	Unregisters a table for SQL cache dependency by removing the entry in the notification table and removes the trigger.
AspNet_SqlCacheUpdateChangeIdStoredProcedure	Updates the notification table by incrementing the changeId for the changed table. ASP.NET uses this value to determine if the data have changed. As indicated below, this stored proc is executed by the trigger created when the table is enabled.

- A SQL Server trigger called ***table_name*_AspNet_SqlCacheNotification_Trigger** is created for the table. This trigger executes the AspNet_SqlCacheUpdateChangeIdStoredProcedure when an INSERT, UPDATE, or DELETE is performed on the table.

- A SQL Server role called **aspnet_ChangeNotification_ReceiveNotificationsOnlyAccess** is added to the database.

The **aspnet_ChangeNotification_ReceiveNotificationsOnlyAccess** SQL Server role has EXEC permissions to the AspNet_SqlCachePollingStoredProcedure. In order for the polling model to work correctly, you must add your process account to the aspnet_ChangeNotification_ReceiveNotificationsOnlyAccess role. The aspnet_regsql.exe tool will not do this for you.

Configuring Polling-Based SQL Cache Dependencies

There are several steps that are required for configuring polling-based SQL cache dependencies. The first step is to enable the database and the table as discussed above. Once that step is complete, the rest of the configuration is as follows:

- Configuring the ASP.NET configuration file.
- Configuring the SqlCacheDependency

Configuring the ASP.NET Configuration File

In addition to adding a connection string as discussed in a previous module, you must also configure a <cache> element with a <sqlCacheDependency> element as shown below:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="Pubs"
    connectionString="Data Source=(local);
      Initial Catalog=pubs;Integrated Security=true;"
    providerName="System.Data.SqlClient" />
  </connectionStrings>
  <system.web>
    <caching>
      <sqlCacheDependency enabled = "true" pollTime = "60000" >
        <databases>
          <add name="pubs" connectionStringName = "pubs" pollTime = "9000000" />
        </databases>
      </sqlCacheDependency>
    </caching>
  </system.web>
</configuration>
```

This configuration enables a SQL cache dependency on the *pubs* database. Note that the pollTime attribute in the <sqlCacheDependency> element defaults to 60000 milliseconds or 1 minute. (This value cannot be less than 500 milliseconds.) In this example, the <add> element adds a new database and overrides the pollTime, setting it to 9000000 milliseconds.

Configuring the SqlCacheDependency

The next step is to configure the SqlCacheDependency. The easiest way to accomplish that is to specify the value for the SqlDependency attribute in the @ Outcache directive as follows:

```
<%@ OutputCache duration="60"
    VaryByParam="none" SqlDependency="pubs:authors" %>
```

In the above @ OutputCache directive, a SQL cache dependency is configured for the *authors* table in the *pubs* database. Multiple dependencies can be configured by separating them with a semi-colon like so:

```
<%@ OutputCache duration="60"
    VaryByParam="none"
    SqlDependency="database_name:table_name;database_name:table_name" %>
```

Another method of configuring the SqlCacheDependency is to do so programmatically. The following code creates a new SQL cache dependency on the *authors* table in the *pubs* database.

```
SqlCacheDependency dep = new SqlCacheDependency("pubs", "authors");
```

One of the benefits of programmatically defining the SQL cache dependency is that you can handle any exceptions that might occur. For example, if you attempt to define a SQL cache dependency for a database that has not been enabled for notification, a **DatabaseNotEnabledForNotificationException** exception will be thrown. In that case, you can attempt to enable the database for notifications by calling the **SqlCacheDependencyAdmin.EnableNotifications** method and passing it the database name.

Likewise, if you attempt to define a SQL cache dependency for a table that has not been enabled for notification, a **TableNotEnabledForNotificationException** will be thrown. You can then call the **SqlCacheDependencyAdmin.EnableTableForNotifications** method passing it the database name and table name.

The following code sample illustrates how to properly configure exception handling when configuring a SQL cache dependency.

```
try {
    SqlCacheDependency SqlDep = new
    SqlCacheDependency("pubs", "authors");
} catch (DatabaseNotEnabledForNotificationException exDBDis) {
    try {
        SqlCacheDependencyAdmin.EnableNotifications("pubs");
    } catch (UnauthorizedAccessException exPerm) {
        Response.Redirect("ErrorPage.htm");
    }
} catch (TableNotEnabledForNotificationException exTabDis) {
    try {
        SqlCacheDependencyAdmin.EnableTableForNotifications("pubs",
        "authors");
    } catch (System.Data.SqlClient.SqlException exc) {
        Response.Redirect("ErrorPage.htm");
    }
} finally {
    Cache.Insert("SqlSource", Source1, SqlDep);
}
```

More Information: https://msdn.microsoft.com/library/t9x04ed2.aspx

Query-Based SQL Cache Dependencies (SQL Server 2005 Only)

When using SQL Server 2005 for SQL cache dependency, the polling-based model is not necessary. When used with SQL Server 2005, SQL cache dependencies communicate directly via SQL connections to the SQL Server instance (no further configuration is necessary) using SQL Server 2005 query notifications.

The simplest way to enable query-based notification is to do so declaratively by setting the **SqlCacheDependency** attribute of the data source object to **CommandNotification** and setting the **EnableCaching** attribute to **true**. Using this method, no code is required. If the result of a command executed against the data source changes, it will invalidate the cache data.

The following example configures a data source control for SQL cache dependency:

```
<asp:SqlDataSource ID="ProductList" runat="server"
    ConnectionString="<%$ ConnectionStrings:Northwind %>"
    EnableCaching="true"
    SqlCacheDependency="CommandNotification"
    SelectCommand="SELECT * FROM [Products]" />
```

In this case, if the query specified in the **SelectCommand** returns a different result than it did originally, the results that are cached are invalidated.

You can also specify that all of your data sources be enabled for SQL cache dependencies by setting the **SqlDependency** attribute of the **@ OutputCache** directive to **CommandNotification**. The example below illustrates this.

```
<%@ OutputCache SqlDependency="CommandNotification"
    duration="60" VaryByParam="none" %>
```

> **NOTE**
>
> For more information on query notifications in SQL Server 2005, see the SQL Server Books Online.

Another method of configuring a query-based SQL cache dependency is to do so programmatically using the SqlCacheDependency class. The following code sample illustrates how this is accomplished.

```
string sql = "SELECT ProductName, ProductID FROM Products";
SqlConnection conn = new
SqlConnection(ConfigurationManager.ConnectionStrings["Northwind"].ConnectionString);
SqlCommand cmd = new SqlCommand(sql, conn);
SqlCacheDependency dep = new SqlCacheDependency(cmd);
Response.AddCacheDependency(dep);
```

More Information: https://msdn.microsoft.com/library/default.asp?url=/library/enus/dnvs05/html/querynotification.asp

Post-Cache Substitution

Caching a page can dramatically increase the performance of a Web application. However, in some cases you need most of the page to be cached and some fragments within the page to be dynamic. For example, if you create a page of news stories that is entirely static for set periods of time, you can set the entire page to be cached. If you wanted to include a rotating ad banner that changed on every page request, then the part of the page containing the advertisement needs to be dynamic. To allow you to cache a page but substitute some content dynamically, you can use ASP.NET post-cache substitution. With post-cache substitution, the entire page is output cached with specific parts marked as exempt from caching. In the example of the ad banners, the AdRotator control allows you to take advantage of post-cache substitution so that ads dynamically created for each user and for each page refresh.

There are three ways to implement post-cache substitution:

- Declaratively, using the Substitution control.
- Programmatically, using the Substitution control API.
- Implicitly, using the AdRotator control.

Substitution Control

The ASP.NET Substitution control specifies a section of a cached page that is created dynamically rather than cached. You place a Substitution control at the location on the page where you want the dynamic content to

appear. At run time, the Substitution control calls a method that you specify with the MethodName property. The method must return a string, which then replaces the content of the Substitution control. The method must be a static method on the containing Page or UserControl control. Using the substitution control causes client-side cacheability to be changed to server cacheability, so that the page will not be cached on the client. This ensures that future requests to the page call the method again to generate dynamic content.

Substitution API

To create dynamic content for a cached page programmatically, you can call the WriteSubstitution method in your page code, passing it the name of a method as a parameter. The method that handles the creation of the dynamic content takes a single HttpContext parameter and returns a string. The return string is the content that will be substituted at the given location. An advantage of calling the WriteSubstitution method instead of using the Substitution control declaratively is that you can call a method of any arbitrary object rather than calling a static method of the Page or the UserControl object.

Calling the WriteSubstitution method causes client-side cacheability to be changed to server cacheability, so that the page will not be cached on the client. This ensures that future requests to the page call the method again to generate dynamic content.

AdRotator Control

The AdRotator server control implements support for post-cache substitution internally. If you place an AdRotator control on your page, it will render unique advertisements on each request, regardless of whether the parent page is cached. As a result, a page that includes an AdRotator control is only cached server-side.

ControlCachePolicy Class

The ControlCachePolicy class allows for the programmatic control of fragment caching using user controls. ASP.NET embeds user controls within a BasePartialCachingControl instance. The BasePartialCachingControl class represents a user control that has output caching enabled.

When you access the BasePartialCachingControl.CachePolicy property of a PartialCachingControl control, you will always receive a valid ControlCachePolicy object. However, if you access the UserControl.CachePolicy property of a UserControl control, you receive a valid ControlCachePolicy object only if the user control is already wrapped by a BasePartialCachingControl control. If it is not wrapped, the ControlCachePolicy object returned by the property will throw exceptions when you attempt to manipulate it because it does not have an associated BasePartialCachingControl. To determine whether a UserControl instance supports caching without generating exceptions, inspect the SupportsCaching property.

Using the ControlCachePolicy class is one of several ways you can enable output caching. The following list describes methods you can use to enable output caching:

- Use the @ OutputCache directive to enable output caching in declarative scenarios.
- Use the PartialCachingAttribute attribute to enable caching for a user control in a code-behind file.
- Use the ControlCachePolicy class to specify cache settings in programmatic scenarios in which you are working with BasePartialCachingControl instances that have been cache-enabled using one of the previous methods and dynamically loaded using the System.Web.UI.TemplateControl.LoadControl method.

A ControlCachePolicy instance can be successfully manipulated only between the Init and PreRender stages of the control life cycle. If you modify a ControlCachePolicy object after the PreRender phase, ASP.NET throws an exception because any changes made after the control is rendered cannot actually affect cache settings (a control is cached during the Render stage). Finally, a user control instance (and therefore its ControlCachePolicy object) is only available for programmatic manipulation when it is actually rendered.

Changes to Caching Configuration - The <caching> Element

There are several changes to caching configuration in ASP.NET 2.0. The <caching> element is new in ASP.NET 2.0

and allows you to make caching configuration changes in the configuration file. The following attributes are available.

ELEMENT	DESCRIPTION
cache	Optional element. Defines global application cache settings.
outputCache	Optional element. Specifies application-wide output-cache settings.
outputCacheSettings	Optional element. Specifies output-cache settings that can be applied to pages in the application.
sqlCacheDependency	Optional element. Configures the SQL cache dependencies for an ASP.NET application.

The <cache> Element

The following attributes are available in the <cache> element:

ATTRIBUTE	DESCRIPTION
disableMemoryCollection	Optional **Boolean** attribute. Gets or sets a value indicating whether the cache memory collection that occurs when the machine is under memory pressure is disabled.
disableExpiration	Optional **Boolean** attribute. Gets or sets a value indicating whether cache expiration is disabled. When disabled, cached items do not expire and background scavenging of expired cache items does not occur.
privateBytesLimit	Optional **Int64** attribute. Gets or sets a value indicating the maximum size of an application's private bytes before the cache starts flushing expired items and attempting to reclaim memory. This limit includes both memory used by the cache as well as normal memory overhead from the running application. A setting of zero indicates that ASP.NET will use its own heuristics for determining when to start reclaiming memory.
percentagePhysicalMemoryUsedLimit	Optional **Int32** attribute. Gets or sets a value indicating the maximum percentage of a machine's physical memory that can be consumed by an application before the cache starts flushing expired items and attempting to reclaim memory This memory usage includes both memory used by the cache as well as the normal memory usage of the running application. A setting of zero indicates that ASP.NET will use its own heuristics for determining when to start reclaiming memory.
privateBytesPollTime	Optional **TimeSpan** attribute. Gets or sets a value indicating the time interval between polling for the application's private bytes memory usage.

The <outputCache> Element

The following attributes are available for the <outputCache> element.

ATTRIBUTE	DESCRIPTION
enableOutputCache	Optional **Boolean** attribute. Enables/disables the page output cache. If disabled, no pages are cached regardless of the programmatic or declarative settings. Default value is **true**.
enableFragmentCache	Optional **Boolean** attribute. Enables/disables the application fragment cache. If disabled, no pages are cached regardless of the @ OutputCache directive or caching profile used. Includes a cache-control header indicating that upstream proxy servers as well as browser clients should not attempt to cache page output. Default value is **false**.
sendCacheControlHeader	Optional **Boolean** attribute. Gets or sets a value indicating whether the **cache-control:private** header is sent by the output cache module by default. Default value is **false**.
omitVaryStar	Optional **Boolean** attribute. Enables/disables sending an Http "**Vary: ** *" header in the response. With the default setting of false, a* "***Vary: *" header is sent for output cached pages. When the Vary header is sent, it allows for different versions to be cached based upon what is specified in the Vary header. For example, *Vary:User-Agents* will store different versions of a page based upon the user agent issuing the request. Default value is **false.**

The <outputCacheSettings> Element

The <outputCacheSettings> element allows for the creation of cache profiles as previously described. The only child element for the <outputCacheSettings> element is the <outputCacheProfiles> element for configuring cache profiles.

The <sqlCacheDependency> Element

The following attributes are available for the <sqlCacheDependency> element.

ATTRIBUTE	DESCRIPTION
enabled	Required **Boolean** attribute. Indicates whether or not changes are being polled for.
pollTime	Optional **Int32** attribute. Sets the frequency with which the SqlCacheDependency polls the database table for changes. This value corresponds to the number of milliseconds between successive pollings. It cannot be set to less than 500 milliseconds. Default value is 1 minute.

More Information

There is some additional information that you should be aware of regarding cache configuration.

- If the worker process private bytes limit is not set, the cache will use one of the following limits:
  - x86 2GB: 800MB or 60% of physical RAM, whichever is less
  - x86 3GB: 1800MB or 60% of physical RAM, whichever is less
  - x64: 1 terabyte or 60% of physical RAM, whichever is less
- If both the worker process private bytes limit and <cache privateBytesLimit/> are set, the cache will use the minimum of the two.

- Just like in 1.x, we drop cache entries and call GC.Collect for two reasons:
  - We are very close to the private bytes limit
  - The available memory is near or less than 10%
- You can effectively disable trim and cache for low available memory conditions by setting <cache percentagePhysicalMemoryUseLimit/> to 100.
- Unlike 1.x, 2.0 will suspend the trim and collect calls if the last GC.Collect did not reduce private bytes or the size of the managed heaps by more than 1% of the (cache) memory limit.

Lab1: Custom Cache Dependencies

1. Create a new Web site.
2. Add a new XML file called cache.xml and save it to the root of the Web application.
3. Add the following code to the Page_Load method in the code-behind of default.aspx:

```
System.Web.Caching.CacheDependency dep = new
    System.Web.Caching.CacheDependency(Server.MapPath("cache.xml"));
Response.AddCacheDependency(dep);
Cache.Insert("time", DateTime.Now.ToString());
Response.Write(Cache["time"]);
```

4. Add the following to the top of default.aspx in source view:

```
<%@ OutputCache Duration="240" VaryByParam="None" %>
```

5. Browse Default.aspx. What does the time say?
6. Refresh the browser. What does the time say?
7. Open cache.xml and add the following code:

```
<anElement></anElement>
```

8. Save cache.xml.
9. Refresh your browser. What does the time say?
10. Explain why the time updated instead of displaying the previously cached values:

Lab 2: Using Polling-Based Cache Dependencies

This lab uses the project you created in the previous module that allows for editing of data in the Northwind database via a GridView and DetailsView control.

1. Open the project in Visual Studio 2005.
2. Run the aspnet_regsql utility against the Northwind database to enable the database and the Products table. Use the following command from a Visual Studio Command Prompt:

```
aspnet_regsql -S server -ed -d Northwind -E -et -t Products
```

3. Add the following to your web.config file:

```
<caching>
    <sqlCacheDependency enabled = "true" pollTime = "60000" >
        <databases>
            <add name="Northwind" connectionStringName = "Northwind" pollTime = "9000000" />
        </databases>
    </sqlCacheDependency>
</caching>
```

4. Add a new webform called showdata.aspx.
5. Add the following @ outputcache directive to the showdata.aspx page:

```
<%@ OutputCache SqlDependency="Northwind:Products" Duration="480" VaryByParam="None"%>
```

6. Add the following code to the Page_Load of showdata.aspx:

```
Response.Write(DateTime.Now.ToString() + "<br><br>");
```

7. Add a new SqlDataSource control to showdata.aspx and configure it to use the Northwind database connection. Click Next.
8. Select the ProductName and ProductID checkboxes and click Next.
9. Click Finish.
10. Add a new GridView to the showdata.aspx page.
11. Choose SqlDataSource1 from the dropdown.
12. Save and browse showdata.aspx. Make note of the time displayed.

Membership

4/10/2020 • 15 minutes to read • Edit Online

by Microsoft

> ASP.NET Membership builds on the success of the Forms authentication model from ASP.NET 1.x. ASP.NET Forms authentication provides a convenient way to incorporate a login form into your ASP.NET application and validate users against a database or other data store.

ASP.NET Membership builds on the success of the Forms authentication model from ASP.NET 1.x. ASP.NET Forms authentication provides a convenient way to incorporate a login form into your ASP.NET application and validate users against a database or other data store. The members of the FormsAuthentication class make it possible to handle cookies for authentication, check for a valid login, log a user out etc. However, implementing Forms authentication in an ASP.NET 1.x application can require a fair amount of code.

Membership in ASP.NET 2.0 is a major advancement over using Forms authentication alone. (Membership is most robust when coupled with Forms authentication, but using Forms authentication is not a requirement.) As you'll soon see, you can use ASP.NET Membership and the login controls in ASP.NET 2.0 to implement a powerful membership system without writing much code at all.

Implementing Membership in ASP.NET 2.0

Membership is implemented by following four steps. Keep in mind that there are many sub-steps that are involved as well as optional configuration that can be implemented as well. These steps are meant to illustrate the big picture of configuring membership.

1. Create your membership database (if SQL Server is used as the membership store.)
2. Specify the membership options in your applications configuration files. (Membership is enabled by default.)
3. Determine the type of membership store you want to use. Options are:
   - Microsoft SQL Server (version 7.0 or later)
   - Active Directory Store
   - Custom membership provider
4. Configure the application for ASP.NET Forms authentication. Once again, Membership is designed to take advantage of Forms authentication, but using Forms authentication is not a requirement.
5. Define user accounts for membership and configure roles if desired.

Creating the Membership Database

If you're using SQL Server 7.0 or later as your membership store, you can use the aspnet_regsql utility (available most easily from the Visual Studio .NET 2005 Command Prompt) to configure your database. The aspnet_regsql utility can be used as a command prompt tool or via a GUI wizard. The wizard method is the easiest way to configure your database. To access the wizard, simply run the following command:

```
aspnet_regsql W
```

Once you run that command, you will be presented with the ASP.NET SQL Server Setup Wizard as shown below.

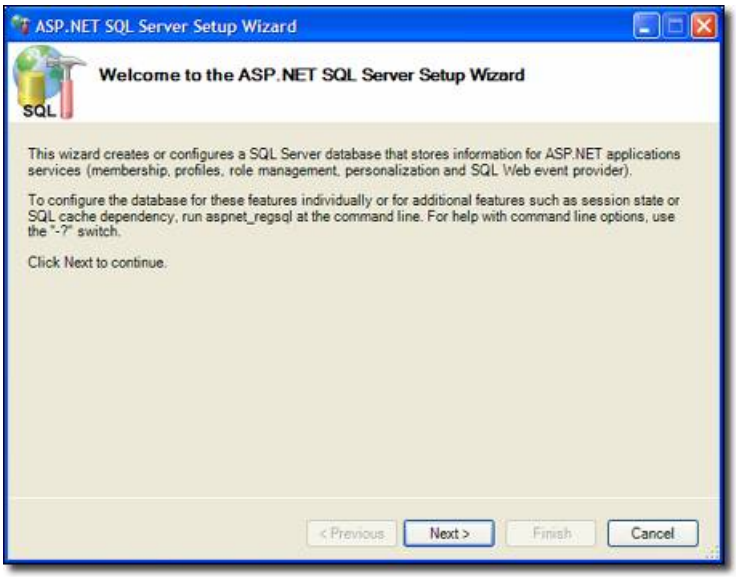


Figure 1

The ASP.NET SQL Server Setup Wizard creates the Web site in the instance you specify in the wizard. However, ASP.NET will use the connection string in the machine.config file to connect to your database. By default, this connection string will point to a SQL Server 2005 instance, so if you are using a SQL Server 2000 or SQL Server 7.0 instance, you will need to modify the connection string in the machine.config file. That connection string can be located here:

```
<configuration>
    <connectionStrings>
      <add name="LocalSqlServer"
         connectionString="data source=(local);
         Integrated Security=SSPI;Initial Catalog=aspnetdb;"
         providerName="System.Data.SqlClient" />
    </connectionStrings>
</configuration>
```

Unfortunately, if you don't modify the connection string, ASP.NET will not give you a descriptive error. It will just continue to complain saying that you haven't created the database. In the case above, I have modified the connection string to point to my local SQL Server 2000 instance.

Specifying Configuration and Adding Users and Roles

The next step in configuring Membership is to add the necessary information to the web.config file of the application. In ASP.NET 1.x, modifying the web.config file was sometimes difficult because of the use of lowerCamelCase and the lack of Intellisense. Visual Studio .NET 2005 makes the task much easier with Intellisense for configuration files, but ASP.NET 2.0 goes one step further by providing a Web interface for editing configuration files.

You can launch the Web interface by clicking the ASP.NET Configuration button on the Solution Explorer toolbar as shown below. You can also launch the Web interface via pop-ups that are displayed when Login controls are inserted.

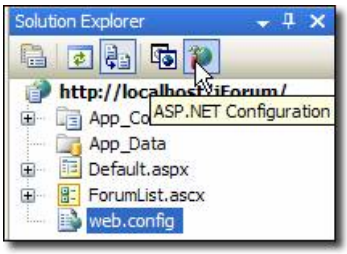


Figure 2

This launches the ASP.NET Web Site Administration Tool shown below. The ASP.NET Web Site Administration is a four-tab interface that makes it easy to manage application settings. The following tabs are available:

- **Home**
- **Security** Configure users, roles, and access.
- **Application** Configure application settings.
- **Provider** Configure and test your applications membership provider.

The Web Site Administration Tool allows you to easily create new users, create new roles, and to manage users and roles. This ability is not available in the Windows interface. The Windows interface allows you to easily define authorization settings and to add, delete, and manage providers, capabilities that are not in the Web Site Administration Tool.

To launch the Windows interface, open the Internet Information Services snap-in, right-click on your application, and choose Properties. Click the ASP.NET tab and then click the Edit Configuration button. (The application must be running under ASP.NET 2.0 for the Edit Configuration button to be enabled. You can configure the ASP.NET version in the ASP.NET dialog as well.) The ASP.NET Configuration Settings dialog is displayed as shown below.

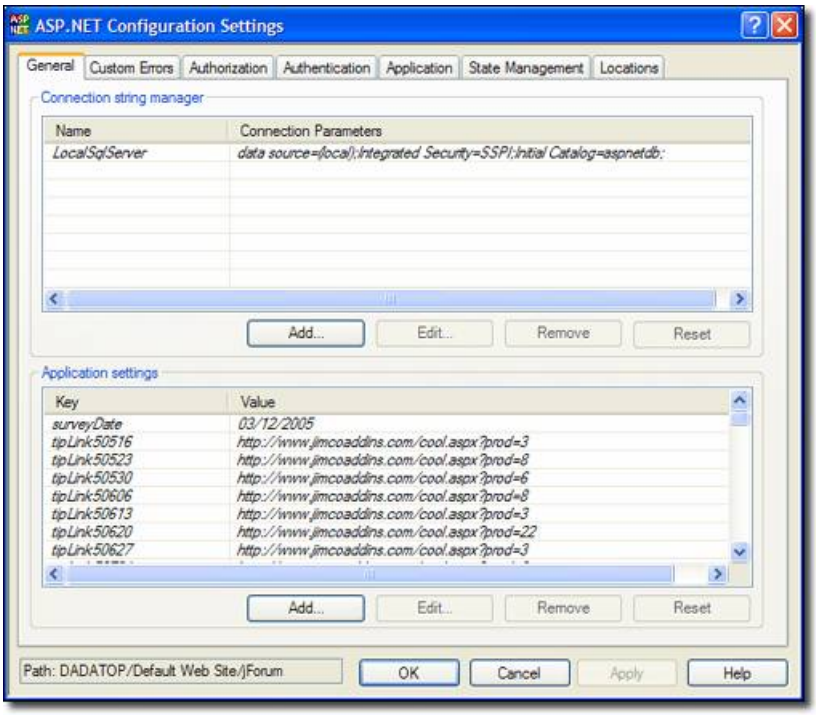


Figure 3

On the General tab, connection strings and application settings are listed. Any settings in italics are defined in a parent configuration file (either the machine.config or a web.config at a higher level) and settings not in italics are from the applications configuration file. If a setting is added, removed, or edited at the application level, ASP.NET will add, remove, or modify the setting at the application levels web.config instead of removing the setting from the configuration file from which it is inherited.

The Authentication tab is shown below. This is where you will configure your membership settings. Forms authentication settings, membership providers, and role providers can be configured here.

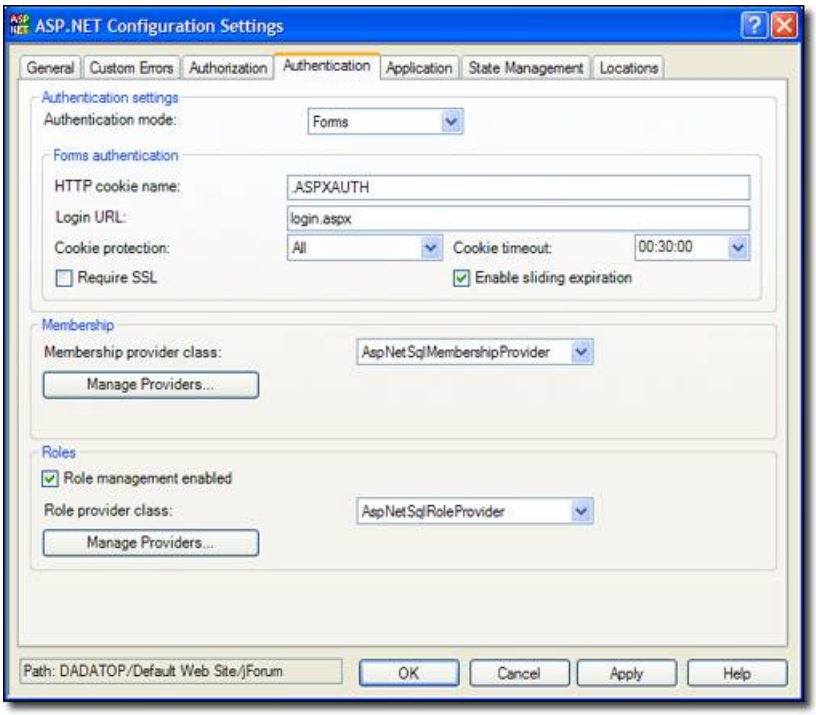


Figure 4

Implementing Membership in Your Application

The easiest way to implement ASP.NET 2.0 membership in your application is to use the provided Logon controls. This method allows you to implement the basics of ASP.NET 2.0 membership without writing any code at all.

The following Logon controls are available in ASP.NET 2.0:

Login Control

The Login control provides an interface for someone to log into your membership system. It provides you with a username and password textbox and a login button. Many other common features such as a link to register for people who have not yet done so, a checkbox that allows the user to automatically login on subsequent visits, a link for a password reminder, etc. All features of the Login control are customizable via the properties of the control.

In ASP.NET 1.x, developers had to write a fair amount of code to do a lookup when using Forms authentication. With ASP.NET 2.0 membership, you can validate users without writing any code at all. ASP.NET will automatically do the look-up of the user for you. (If you are using the Login control without using ASP.NET membership, you can use the **OnAuthenticate** method to validate the user.)

LoginView Control

The LoginView control is a templated control that provides two templates by default; the AnonymousTemplate and the LoggedInTemplate. The template that is displayed is determined by whether or not the user is logged into your membership system. This control is typically used to display a Login control when a user has not yet logged in and a LoginStatus control and/or other login controls when the user has logged in. If you are using role management in your ASP.NET application, the LoginView control can display a specific template based upon the users role. (More on ASP.NET role management will be covered later.)

PasswordRecovery Control

The PasswordRecovery control allows users to receive an email with his or her current password or reset his or her password. Clear text and encrypted passwords can be recovered and emailed to users. If the password is hashed, it

cannot be recovered. Instead the user will be required to perform a password reset.

LoginStatus Control

The LoginStatus control is used to display a login indicator to users who are not logged in and a logout indicator to users who are currently logged in. The Request.IsAuthenticated property is used to determine which indicator to display. The indicator displayed by the LoginStatus control can be text (implemented via the **LoginText** and **LogoutText** properties) or images (implemented via the **LoginImageUrl** and **LogoutImageUrl** properties.)

When a user logs out via the LoginStatus control, he or she is redirected to the URL specified by the **LogoutPageUrl** property. If that property is not set, the current page is refreshed. Since the site is likely protected by Forms authentication, the refresh of the current page will redirect the user to the login page for the site.

LoginName Control

The LoginName control displays the username of the user currently logged into the site.

CreateUserWizard Control

The CreateUserWizard control provides users with a convenient way to register for your membership system. You can add steps (implemented as a collection of WizardSteps) via the interface shown below.

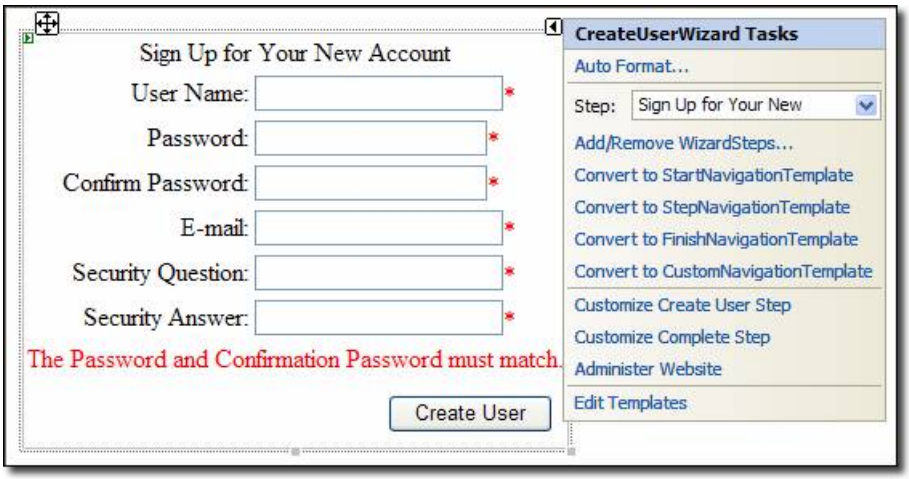


Figure 5

The CreateUserWizard is a templated control that derives from the Wizard class and provides the following templates:

- **HeaderTemplate** This template controls the appearance of the header of the wizard.
- **SidebarTemplate** This template controls the appearance of the sidebar of the wizard.
- **StartNavigationTemplate** This template controls the appearance of the navigation are of the wizard at the start step.
- **StepNavigationTemplate** This template controls the appearance of the navigation area when not in the start or finish step.
- **FinishNavigationTemplate** This template controls the appearance of the navigation area when on the finish step.

Additionally, for each step that you add to the Wizard, ASP.NET will create a custom template that contains both a ContentTemplate and a CustomNavigationTemplate for that step. For full details on customizing the CreateUserWizard, see the VS.NET 2005 documentation:

ChangePassword Control

The ChangePassword control allows users to change his or her password. If the DisplayUserName property is true (it is false by default), the user can change his or her password when they are not logged in. If the user *is* already logged in and the DisplayUserName property is true, the user will be able to change the password of another user that is not logged in providing they know the user ID of that user.

Keep in mind that if you want users to be able to change passwords without having to log in, you will need to ensure that the page on which the ChangePassword control is displayed allows anonymous access. Obviously, users will have to provide their old password in order to change their password.

Role Management

Role management allows you to assign users to a particular role and then restrict access to certain file or folders based on that role. Role management also provides an API so that you can programmatically determine someones role or determine all users in a particular role and respond accordingly.

Role management is not a requirement in ASP.NET membership, nor is membership a requirement in order to use role management. However, the two supplement each other nicely and it is likely that developers will use them in conjunction with each other.

To enable role management in your application, make the following change in your web.config file:

```
<roleManager enabled="true" cacheRolesInCookie="true" cookieProtection="All" />
```

When the **cacheRolesInCookie** attribute is set to true, ASP.NET caches a users role membership in a cookie on the client. This allows role lookups to occur without calls into the RoleProvider. When using this attribute, developers are encouraged to ensure that the **cookieProtection** attribute is set to All. (This is the default setting.) This ensures that the cookie data are encrypted and helps to ensure that the cookies contents haven't been altered. Roles can be added using the Web Site Administration Tool. It allows you to easily define roles, configure access to parts of the site based on those roles, and assign users to roles.

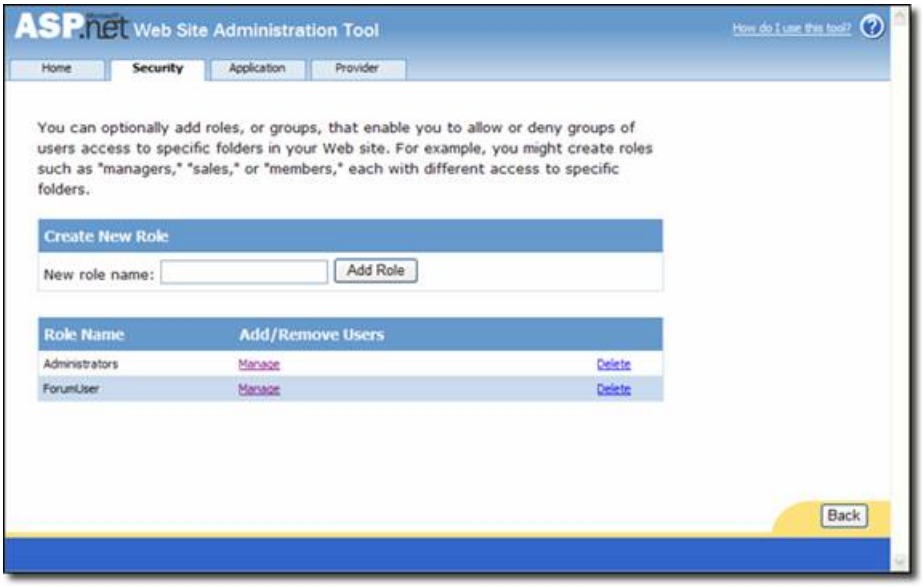


Figure 6

As shown above, new roles can be added by simply entering the name of the role and then clicking Add Role. Existing roles can be managed or deleted by clicking the appropriate link in the list of existing roles.

When you manage a role, you can add or remove users as shown below.

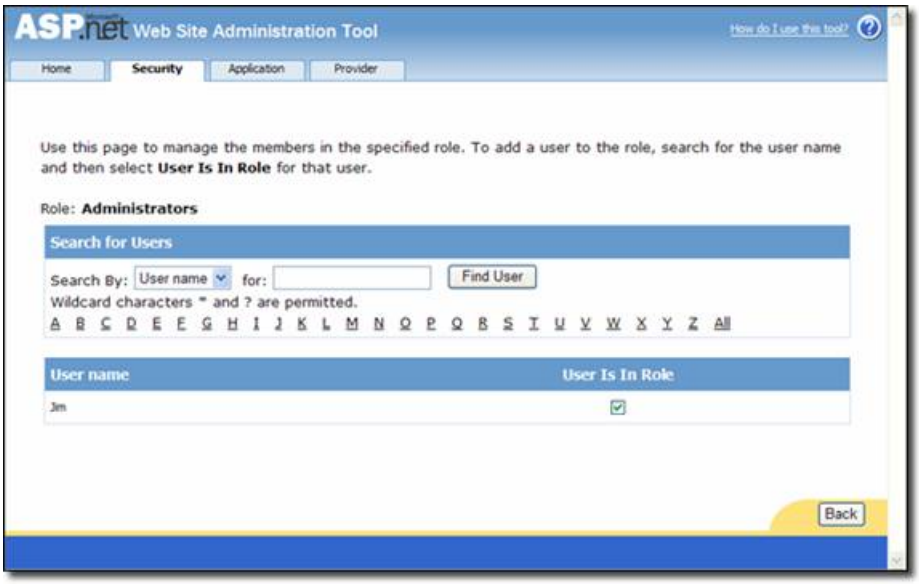


Figure 7

By checking the User Is In Role checkbox, you can easily add a user to a specific role. ASP.NET will automatically update your membership database with the appropriate entries. You will also want to configure access rules for your application. ASP.NET 1.x developers are familiar with doing this via the <authorization> element in the web.config file, and that option is still available in ASP.NET 2.0. However, its easier to manage access rules using the Web Site Administration Tool as shown below.

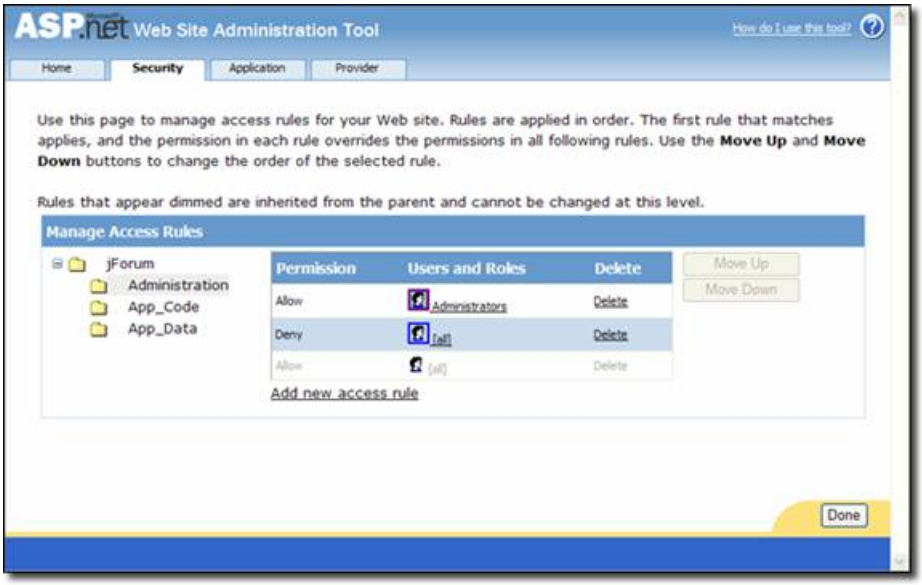


Figure 8

In this case, the Administration folder is highlighted (its difficult to see because the tool highlights it in light gray) and the Administrators role has been granted access. All other users are denied. You can click on the head icon to select a rule and then use the Move Up and Move Down buttons to arrange the rules. As with the ASP.NET <authorization> element, rules are processed in the order in which they appear. In other words, if the order of rules in the shot above were reversed, no one would have access to the Administration folder because the first rule that ASP.NET would encounter would be the rule that denies everyone to the folder.

ASP.NET 2.0 adds a web.config file to the folder for which you are specifying an access rule. Access rules can be edited via the configuration file or via the Web Site Administration Tool. In other words, the Web Site Administration Tool is simply an interface through which the configuration file can be edited in a user-friendly environment.

Using Roles in Code

The API for role management has not changed since version 1.x. The **IsInRole** method is used to determine if a user is in a particular role.

```
if (User.IsInRole(Administrators)) {
    btnManageSite.Visible = true;
}
```

ASP.NET also creates a RolePrincipal instance as a member of the current context. The RolePrincipal object can be used to obtain all of the roles to which the user belongs as follows:

```
string[] userRoles = ((RolePrincipal)User).GetRoles();
```

Using RoleGroups with the LoginView Control

Now that you have an understanding of role management and membership, lets discuss briefly how the LoginView control takes advantage of this capability in ASP.NET 2.0. As previously discussed, the LoginView control is a templated control that contains two templates by default; the AnonymousTemplate and the LoggedInTemplate. Within the LoginView Tasks dialog is a link (shown below) that allows you to edit RoleGroups.

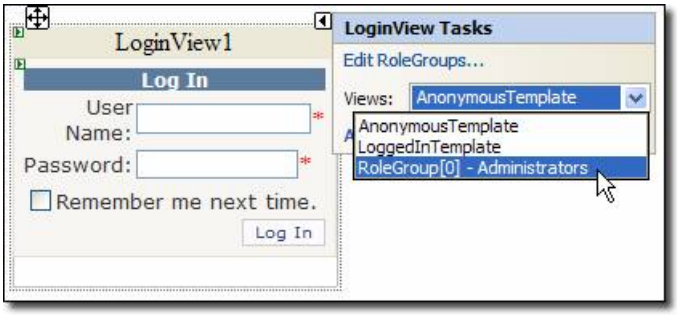


Figure 9

Each RoleGroup object contains an array of strings that defines which roles that RoleGroup applies to. To add a new RoleGroup to the LoginView control, click the Edit RoleGroups link. In the image above, you can see that I have added a new RoleGroup for Administrators. By selecting that RoleGroup (RoleGroup[0]) from the Views dropdown, I can configure a template that will only be displayed to members of the Administrators role. In the image below, I have added a new RoleGroup that applies to members of the Sales role and the Distribution role. This adds a second RoleGroup to the Views dropdown in the LoginView Tasks dialog and anything added to that template will be visible by any user in either the Sales or Distribution role.

Figure 10

Overriding the Existing Membership Provider

There are a couple of ways that you can extend the functionality of ASP.NET membership. First of all, you can obviously change the existing functionality of the SqlMembershipProvider class by inheriting from it and overriding its methods. For example, if you want to implement your own functionality when users are created, you can create your own class that inherits from SqlMembershipProvider as follows:

```
public class jForumMembershipProvider : SqlMembershipProvider {
    public jForumMembershipProvider() {

    }
    public override MembershipUser CreateUser(
    string username,
    string password,
    string email,
    string passwordQuestion,
    string passwordAnswer,
    bool isApproved,
    object providerUserKey,
    out MembershipCreateStatus status) {
        // your own implementation
        return base.CreateUser(
        username,
        password,
        email,
        passwordQuestion,
        passwordAnswer,
        isApproved,
        providerUserKey,
        out status);
    }
}
```

If, on the other hand, you want to create your own provider (to store your membership information in an Access database, for example), you can create your own provider.

Creating Your Own Membership Provider

To create your own membership provider, you will first need to create a class that inherits from the MembershipProvider class. If you are using VB.NET, Visual Studio 2005 will add the stubs for all of the methods

that you need to override. If you are using C#, its up to you to add the stubs.

You will need to override the following:

- ApplicationName property
- ChangePassword function
- ChangePasswordQuestionAndAnswer function
- CreateUser function
- DeleteUser function
- EnablePasswordReset property
- EnablePasswordRetrieval property
- FindUsersByEmail function
- FindUsersByName function
- GetAllUsers function
- GetNumberOfUsersOnline function
- GetPassword function
- GetUser function
- GetUserNameByEmail function
- MaxInvalidPasswordAttempts property
- MinRequiredNonAlphanumericCharacters property
- MinRequiredPasswordLength property
- PasswordAttemptWindow property
- PasswordFormat property
- PasswordStrengthRegularExpression property
- RequiresQuestionAndAnswer property
- RequiresUniqueEmail property
- ResetPassword function
- Unlock user function
- UpdateUser function
- ValidateUser function

Thats quite a list to implement as a C# developer. You may find it easier to create the class in VB.NET without any implementation and then use .NET Reflector or a similar tool to convert the code to C#.

The connection string and other properties should be set to their defaults in the Initialize method. (The Initialize method is fired when the provider is loaded at runtime.) The second parameter to the Initialize method is of type System.Collections.Specialized.NameValueCollection and is a reference to the <add> element that is associated with your custom provider in the web.config file. That entry looks like the following:

```
<system.web>
  <authentication mode="Forms"/>
  <membership
  defaultProvider="jForumCustomMembershipProvider" >
    <providers>
      <add name="jForumCustomMembershipProvider"
      type="jForumCustomMembershipProvider"
      requiresQuestionAndAnswer="true"
      connectionString="Provider=Microsoft.Jet.
        OLEDB.4.0;Data Source=C:\jForumCustomMembershipProvider\
        App_Data\Members.mdb;Persist Security
        Info=False"
        />
    </providers>
  </membership>
</system.web>
```

Here is an example of the Initialize method.

```
public override void Initialize(string name,
    System.Collections.Specialized.NameValueCollection config) {

    if (config["requiresQuestionAndAnswer"])
        _requiresQuestionAndAnswer = true;
    _connString = config["connectionString"];
    base.Initialize(name, config);
}
```

In order to validate the user when they submit your login form, you will need to use the ValidateUser method. This method fires when the user clicks the login button in the Login control. You will place your code that does the user lookup in this method.

As you can see, writing your own membership provider is not difficult and allows you to extend this powerful functionality of ASP.NET 2.0.

Configuration and Instrumentation

4/10/2020 • 30 minutes to read • Edit Online

by Microsoft

> There are major changes in configuration and instrumentation in ASP.NET 2.0. The new ASP.NET configuration API allows for configuration changes to be made programmatically. In addition, many new configuration settings exist allow for new configurations and instrumentation.

There are major changes in configuration and instrumentation in ASP.NET 2.0. The new ASP.NET configuration API allows for configuration changes to be made programmatically. In addition, many new configuration settings exist allow for new configurations and instrumentation.

In this module, we will discuss ASP.NET the configuration API as it relates to reading from and writing to ASP.NET configuration files, and we will also cover ASP.NET instrumentation. We will also cover the new features available in ASP.NET tracing.

ASP.NET Configuration API

The ASP.NET configuration API allows you to develop, deploy, and manage application configuration data by using a single programming interface. You can use the configuration API to develop and modify complete ASP.NET configurations programmatically without directly editing the XML in the configuration files. In addition, you can use the configuration API in console applications and scripts that you develop, in Web-based management tools, and in Microsoft Management Console (MMC) snap-ins.

The following two configuration-management tools use the configuration API and are included with the .NET Framework version 2.0:

- The ASP.NET MMC snap-in, which uses the configuration API to simplify administrative tasks, providing an integrated view of local configuration data from all levels of the configuration hierarchy.
- The Web Site Administration Tool, which allows you to manage configuration settings for local and remote applications, including hosted sites.

The ASP.NET configuration API comprises a set of ASP.NET management objects that you can use to configure Web sites and applications programmatically. Management objects are implemented as a .NET Framework class library. The configuration API programming model helps ensure code consistency and reliability by enforcing data types at compile time. To make it easier to manage application configurations, the configuration API allows you to view data that is merged from all points in the configuration hierarchy as a single collection, instead of viewing the data as separate collections from different configuration files. Additionally, the configuration API enables you to manipulate entire application configurations without directly editing the XML in the configuration files. Finally, the API simplifies configuration tasks by supporting administrative tools, such as the Web Site Administration Tool. The configuration API simplifies deployment by supporting the creation of configuration files on a computer and running configuration scripts across multiple computers.

> **NOTE**
>
> The configuration API does not support the creation of IIS applications.

Working with Local and Remote Configuration Settings

A Configuration object represents the merged view of the configuration settings that apply to a specific physical entity, such as a computer, or to a logical entity, such as an application or a Web site. The specified logical entity can exist on the local computer or on a remote server. When no configuration file exists for a specified entity, the Configuration object represents the default configuration settings as defined by the Machine.config file.

You can get a Configuration object by using one of the open configuration methods from the following classes:

1. The ConfigurationManager class, if your entity is a client application.
2. The WebConfigurationManager class, if your entity is a Web application.

These methods will return a Configuration object, which in turn provides the required methods and properties to handle the underlying configuration files. You can access these files for reading or writing.

Reading

You use the GetSection or GetSectionGroup method to read configuration information. The user or process that reads must have Read permissions on all of the configuration files in the hierarchy.

> **NOTE**
>
> If you use a static GetSection method that takes a path parameter, the path parameter must refer to the application in which the code is running. Otherwise, the parameter is ignored and the configuration information for the currently running application is returned.

Writing

You use one of the Save methods to write configuration information. The user or process that writes must have Write permissions on the configuration file and directory at the current configuration hierarchy level, as well as Read permissions on all of the configuration files in the hierarchy.

To generate a configuration file that represents the inherited configuration settings for a specified entity, use one of the following save-configuration methods:

1. The Save method to create a new configuration file.
2. The SaveAs method to generate a new configuration file at another location.

Configuration Classes and Namespaces

Many configuration classes and methods are similar to each other. The following table describes the most commonly used configuration classes and namespaces.

CONFIGURATION CLASS OR NAMESPACE	DESCRIPTION
System.Configuration namespace	Contains the main configuration classes for all .NET Framework applications. Section handler classes are used to obtain configuration data for a section from methods, such as GetSection and GetSectionGroup. These two methods are non-static.
System.Configuration.Configuration class	Represents a set of configuration data for a computer, application, Web directory, or other resource. This class contains useful methods, such as GetSection and GetSectionGroup, for updating configuration settings and obtaining references to sections and section groups. This class is used as a return type for methods that obtain design-time configuration data, such as the methods of the WebConfigurationManager and ConfigurationManager classes.

CONFIGURATION CLASS OR NAMESPACE	DESCRIPTION
System.Web.Configuration namespace	Contains the section handler classes for the ASP.NET configuration sections defined at ASP.NET Configuration Settings. Section handler classes are used to obtain configuration data for a section from methods, such as GetSection and GetSectionGroup.
System.Web.Configuration.WebConfigurationManager class	Provides useful methods for obtaining references to run-time and design-time configuration settings. These methods use the System.Configuration.Configuration class as a return type. You can use the static GetSection method of this class or the non-static GetSection method of the System.Configuration.ConfigurationManager class interchangeably. For Web application configurations, the System.Web.Configuration.WebConfigurationManager class is recommended instead of the System.Configuration.ConfigurationManager class.
System.Configuration.Provider namespace	Provides a way to customize and extend the configuration provider. This is the base class for all provider classes in the configuration system.
System.Web.Management namespace	Contains classes and interfaces for managing and monitoring the health of Web applications. Strictly speaking, this namespace is not considered part of the configuration API. For example, tracing and event firing is accomplished by the classes in this namespace.
System.Management.Instrumentation namespace	Provides the classes necessary for the instrumentation of applications to expose their management information and events through Windows Management Instrumentation (WMI) to potential consumers. ASP.NET health monitoring uses WMI to deliver events. Strictly speaking, this namespace is not considered part of the configuration API.

Reading from ASP.NET Configuration Files

The WebConfigurationManager class is the core class for reading from ASP.NET configuration files. There are essentially three steps to reading ASP.NET configuration files:

1. Get a Configuration object using the OpenWebConfiguration method.
2. Get a reference to the desired section in the configuration file.
3. Read the desired information from the configuration file.

The Configuration object represents does not represent a particular configuration file. Instead, it represents a merged view of the configuration of a computer, application, or Web site. The following code sample instantiates a Configuration object representing the configuration of a Web application called *ProductInfo*.

```
Configuration config = WebConfigurationManager.OpenWebConfiguration("/ProductInfo);
```

NOTE

Note that if the /ProductInfo path doesn't exist, the above code will return the default configuration as specified in the machine.config file.

Once you have the Configuration object, you can then use the GetSection or GetSectionGroup method to drill into the configuration settings. The following example gets a reference to the impersonation settings for the above ProductInfo application:

```
Configuration config =
    WebConfigurationManager.OpenWebConfiguration("/ProductInfo);
IdentitySection section =
   (IdentitySection)config.GetSection("system.web/identity");
```

Writing to ASP.NET Configuration Files

As in reading from configuration files, the WebConfigurationManager class is the core for writing to Asp.NET configuration files. There are also three steps to writing to ASP.NET configuration files.

1. Get a Configuration object using the OpenWebConfiguration method.
2. Get a reference to the desired section in the configuration file.
3. Write the desired information from the configuration file using the Save or SaveAs method.

The following code changes the **debug** attribute of the <compilation> element to false:

```
System.Configuration.Configuration updateWebConfig =
    System.Web.Configuration.WebConfigurationManager.OpenWebConfiguration("/webApp");

System.Web.Configuration.CompilationSection compilation =
    updateWebConfig.GetSection("system.web/compilation")
    as System.Web.Configuration.CompilationSection;

compilation.Debug = false;

if (!compilation.SectionInformation.IsLocked) {
    updateWebConfig.Save();
    Response.Write("Save Success!");
} else {
    Response.Write("Save Failed!");
}
```

When this code is executed, the **debug** attribute of the <compilation> element will be set to false for the *webApp* application's web.config file.

System.Web.Management Namespace

The System.Web.Management namespace provides the classes and interfaces for managing and monitoring the health of ASP.NET applications.

Logging is accomplished by defining a rule that associates events with a provider. The rule defines the type of events that are sent to the provider. The following base events are available for you to log:

WEBBASEEVENT	THE BASE EVENT CLASS FOR ALL EVENTS. CONTAINS THE REQUIRED PROPERTIES FOR ALL EVENTS SUCH AS EVENT CODE, EVENT DETAIL CODE, THE DATE AND TIME THE EVENT WAS RAISED, SEQUENCE NUMBER, THE EVENT MESSAGE, AND EVENT DETAILS.
WebManagementEvent	The base event class for management events, such as application lifetime, request, error, and audit events.
WebHeartbeatEvent	The event generated by the application in regular intervals to capture useful runtime state information.

WEBBASEEVENT	THE BASE EVENT CLASS FOR ALL EVENTS. CONTAINS THE REQUIRED PROPERTIES FOR ALL EVENTS SUCH AS EVENT CODE, EVENT DETAIL CODE, THE DATE AND TIME THE EVENT WAS RAISED, SEQUENCE NUMBER, THE EVENT MESSAGE, AND EVENT DETAILS.
WebAuditEvent	The base class for security audit events, which are used to mark conditions such as authorization failure, decryption failure, *etc.*
WebRequestEvent	The base class for all informational request events.
WebBaseErrorEvent	The base class for all events indicating error conditions.

The types of providers available allow you to send event output to Event Viewer, SQL Server, Windows Management Instrumentation (WMI), and email. The pre-configured providers and event mappings reduce the amount of work necessary to get event output logged.

ASP.NET 2.0 uses the Event Log provider out-of-the-box to log events based on application domains starting and stopping, as well as logging any unhandled exceptions. This helps to cover some of the basic scenarios. For example, let's say that your application throws an exception, but the user doesn't save the error and you can't reproduce it. With the default Event Log rule, you would be able to gather the exception and stack information to get a better idea of what kind of error occurred. Another example applies if your application is losing session state. In that case, you can look in the Event Log to determine whether the application domain is recycling, and why the application domain stopped in the first place.

Also, the health monitoring system is extensible. For example, you can define custom Web events, fire them within your application, and then define a rule to send the event information to a provider such as your email. This allows you to easily tie your instrumentation to the health monitoring providers. As another example, you could fire an event each time an order is processed and set up a rule that sends each event to the SQL Server database. You could also fire an event when a user fails to log on multiple times in a row, and set up the event to use the email-based providers.

The configuration for the default providers and events is stored in the global Web.config file. The global Web.config file stores all the Web-based settings that were stored in the Machine.config file in ASP.NET 1x. The global Web.config file is located in the following directory:

```
%windir%\Microsoft.Net\Framework\v2.0.*\config\Web.config
```

The <healthMonitoring> section of the global Web.config file provides default configuration settings. You can override these setting or configure your own settings by implementing the <healthMonitoring> section in the Web.config file for your application.

The <healthMonitoring> section of the global Web.config file contains the following items:

PROVIDERS	CONTAINS PROVIDERS SET UP FOR THE EVENT VIEWER, WMI, AND SQL SERVER.
eventMappings	Contains mappings for the various WebBase classes. You can extend this list if you generate your own event class. Generating your own event class gives you finer granularity over the providers you send information to. For example, you could configure unhandled exceptions to be sent to SQL Server, while sending your own custom events to email.
rules	Links the eventMappings to the provider.

PROVIDERS	CONTAINS PROVIDERS SET UP FOR THE EVENT VIEWER, WMI, AND SQL SERVER.
buffering	Used with SQL Server and email providers to determine how often to flush the events to the provider.

Below is a code example from the global Web.config file.

```
<healthMonitoring>
  <!-- Event Log Provider being added. -->
  <providers>
    <add name="EventLogProvider"
         type="System.Web.Management.EventLogWebEventProvider,
               System.Web,Version=2.0.0.0,Culture=neutral,
               PublicKeyToken=b03f5f7f11d50a3a" />
  </providers>
  <!-- Event mapping provides a friendly name to the
events based on the WebBaseErrorEvent class. -->
  <eventMappings>
    <add name="All Errors"
         type="System.Web.Management.WebBaseErrorEvent,
               System.Web,Version=2.0.0.0,Culture=neutral,
               PublicKeyToken=b03f5f7f11d50a3a"
          startEventCode="0" endEventCode="2147483647" />
  </eventMappings>
  <!-- Rule tying the "All Errors" event mapping to the EventLog Provider. -->
  <rules>
    <add name="All Errors Default" eventName="All Errors"
         provider="EventLogProvider"
         profile="Default" minInstances="1"
         maxLimit="Infinite" minInterval="00:01:00" custom="" />
  </rules>
</healthMonitoring>
```

How to store events to Event Viewer

As mentioned earlier, the provider for logging events in the Event Viewer is configured for you in the global Web.config file. By default, all events based on **WebBaseErrorEvent** and **WebFailureAuditEvent** are logged. You can add additional rules to log additional information to the Event Log. For example, if you wanted to log all events (*i.e.*, every event based on **WebBaseEvent**), you could add the following rule to your Web.config file:

```
<healthMonitoring>
    <rules>
        <add name="All Events" eventName="All Events"
             provider="EventLogProvider" profile="Critical" />
    </rules>
</healthMonitoring>
```

This rule would link the **All Events** event map to the Event Log provider. Both eventMapping and the provider are included in the global Web.config file.

How to store events to SQL Server

This method uses the **ASPNETDB** database, which is generated by the Aspnet_regsql.exe tool. The default provider uses the LocalSqlServer connection string, which uses either a file-based database in the App_data folder or the local SQLExpress instance of SQL Server. Both the LocalSqlServer connection string and the SqlProvider are configured in the global Web.config file.

The LocalSqlServer connection string in the global Web.config file looks like this:

```
<connectionStrings>
    <add name="LocalSqlServer"
         connectionString="data source=.\SQLEXPRESS;
         Integrated Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;
         User Instance=true"
         providerName="System.Data.SqlClient" />
</connectionStrings>
```

If you want to use another SQL Server instance, you'll need to use the Aspnet_regsql.exe tool, which can be found in the %windir%\Microsoft.Net\Framework\v2.0.* folder. Use the Aspnet_regsql.exe tool to generate a custom **ASPNETDB** database on the SQL Server instance, then add the connection string to your applications configuration file, and then add a provider by using the new connection string. Once you have the **ASPNETDB** database created, you'll need to set a rule to link an eventMapping to the sqlProvider.

Whether you use the default SqlProvider or configure your own provider, you'll need to add a rule linking the provider with an event map. The following rule links the new provider that you created above to the **All Events** event map. This rule will log all the events based on **WebBaseEvent** and send them to the MySqlWebEventProvider that will use the MYASPNETDB connection string. The following code adds a rule to link the provider with an event map:

```
<healthMonitoring>
    <rules>
        <add name="All Events" eventName="All Events"
        provider="MySqlWebEventProvider" profile="Critical"/>
    </rules>
</healthMonitoring>
```

If you wanted to only send errors to SQL Server, you could add the following rule:

```
<add name="All Errors"
    eventName="All Errors"
    provider="MySqlWebEventProvider"
    profile="Critical"/>
```

How to forward events to WMI

You can also forward the events to WMI. The WMI provider is configured for you in the global Web.config file by default.

The following code example adds a rule to forward the events to WMI:

```
<providers>
    <add name="WmiWebEventProvider"
      type="System.Web.Management.WmiWebEventProvider,System.Web,
         Version=2.0.0.0,Culture=neutral,
         PublicKeyToken=b03f5f7f11d50a3a" />
</providers>
```

You will need to add a rule to associate an eventMapping to the provider, and also a WMI listener application to listen for the events. The following code example adds a rule to link the WMI provider to the **All Events** event map:

```
<rules>
    <add name="All Events"
      eventName="All Events" provider="WmiWebEventProvider"
        profile="Critical" />
</rules>
```

How to forward events to email

You can also forward events to email. Be careful about which event rules you map to your email provider, as you can unintentionally send yourself a lot of information that may be better suited for SQL Server or the Event Log. There are two email providers; SimpleMailWebEventProvider and TemplatedMailWebEventProvider. Each has the same configuration attributes, with the exception of the "template" and "detailedTemplateErrors" attributes, both of which are only available on the TemplatedMailWebEventProvider.

> **NOTE**
> Neither of these email providers is configured for you. You'll need to add them to your Web.config file.

The main difference between these two email providers is that SimpleMailWebEventProvider sends emails in a generic template that cannot be modified. The sample Web.config file adds this email provider to the list of configured providers by using the following rule:

```
<add name="mySimple-mailWebEventProvider"
     type="System.Web.Management.Simple-mailWebEventProvider"
     to="e-mail@foo.com" from="e-mail@foo.com"
     maxMessagesPerNotification="1" maxEventsPerMessage="10"
     buffer="true" bufferMode="Critical Notification"
     subjectPrefix="Web Events"/>
```

The following rule is also added to tie the email provider to the **All Events** event map:

```
<add name="All Events" eventName="All Events"
    provider="mySimple-mailWebEventProvider" profile="Critical"/>
```

ASP.NET 2.0 Tracing

There are three major enhancements to tracing in ASP.NET 2.0.

1. Integrated tracing functionality
2. Programmatic access to trace messages
3. Improved application-level tracing

Integrated Tracing Functionality

You can now route messages emitted by the System.Diagnostics.Trace class to ASP.NET tracing output, and route messages emitted by ASP.NET tracing to System.Diagnostics.Trace. You can also forward ASP.NET instrumentation events to System.Diagnostics.Trace. This functionality is provided by the new **writeToDiagnosticsTrace** attribute of the <trace> element. When this Boolean value is true, ASP.NET Trace messages are forwarded to the System.Diagnostics tracing infrastructure for use by any listeners that are registered to display Trace messages.

Programmatic Access to Trace Messages

ASP.NET 2.0 allows for programmatic access to all trace messages via the **TraceContextRecord** class and the **TraceRecords** collection. The most efficient way of accessing trace messages is to register a **TraceContextEventHandler** delegate (also new in ASP.NET 2.0) to handle the new **TraceFinished** event. You can then loop through the trace messages as you wish.

The following code sample illustrates this:

```
void Page_Load(object sender, EventArgs e) {
    // Register a handler for the TraceFinished event.
    Trace.TraceFinished += new
        TraceContextEventHandler(this.OnTraceFinished);
    // Write a trace message.
    Trace.Write("Web Forms Infrastructure Methods",
      "USERMESSAGE: Page_Load complete.");
}

// A TraceContextEventHandler for the TraceFinished event.
void OnTraceFinished(object sender, TraceContextEventArgs e) {
    TraceContextRecord r = null;
    // Iterate through the collection of trace records and write
    // them to the response stream.
    foreach (object o in e.TraceRecords) {
        r = (TraceContextRecord)o;
        Response.Write(String.Format("trace message: {0} <BR>",
        r.Message));
    }
}
```

In the above example, I loop through the TraceRecords collection and then write each message to the Response stream.

Improved Application-Level Tracing

Application-level tracing is improved via the introduction of the new **mostRecent** attribute of the <trace> element. This attribute specifies whether the most recent application-level tracing output is displayed and older trace data beyond the limits that are indicated by the requestLimit are discarded. If false, trace data are displayed for requests until the requestLimit attribute is reached.

ASP.NET Command Line Tools

There are several command-line tools to aid in configuration of ASP.NET. ASP.NET developers should be familiar with the aspnet_regiis.exe tool. ASP.NET 2.0 provides three other command-line tools to aid in configuration.

The following command-line tools are available:

TOOL	USE
aspnet_regiis.exe	Allows for registration of ASP.NET with IIS. There are two versions of this tools that ship with ASP.NET 2.0, one for 32-bit systems (in the Framework folder) and one for 64-bit systems (in the Framework64 folder.) The 64-bit version will not be installed on a 32-bit OS.
aspnet_regsql.exe	The ASP.NET SQL Server Registration tool is used to create a Microsoft SQL Server database for use by the SQL Server providers in ASP.NET, or to add or remove options from an existing database. The Aspnet_regsql.exe file is located in the [drive:]\WINDOWS\Microsoft.NET\Framework\versionNumber folder on your Web server.

TOOL	USE
aspnet_regbrowsers.exe	The ASP.NET Browser Registration tool parses and compiles all system-wide browser definitions into an assembly and installs the assembly into the global assembly cache. The tool uses the browser definition files (.BROWSER files) from the .NET Framework Browsers subdirectory. The tool can be found in the %SystemRoot%\Microsoft.NET\Framework\version\ directory.
aspnet_compiler.exe	The ASP.NET Compilation tool enables you to compile an ASP.NET Web application, either in place or for deployment to a target location such as a production server. In-place compilation helps application performance because end users do not encounter a delay on the first request to the application while the application is compiled.

Because the aspnet_regiis.exe tool is not new to ASP.NET 2.0, we will not discuss it here.

ASP.NET SQL Server Registration Tool - aspnet_regsql.exe

You can set several types of options using the ASP.NET SQL Server Registration tool. You can specify a SQL connection, specify which ASP.NET application services use SQL Server to manage information, indicate which database or table is used for SQL cache dependency, and add or remove support for using SQL Server to store procedures and session state.

Several ASP.NET application services rely on a provider to manage storing and retrieving data from a data source. Each provider is specific to the data source. ASP.NET includes a SQL Server provider for the following ASP.NET features:

- Membership (the SqlMembershipProvider class).
- Role Management (the SqlRoleProvider class).
- Profile (the SqlProfileProvider class).
- Web Parts Personalization (the SqlPersonalizationProvider class).
- Web Events (the SqlWebEventProvider class).

When you install ASP.NET, the Machine.config file for your server includes configuration elements that specify SQL Server providers for each of the ASP.NET features that rely on a provider. These providers are configured, by default, to connect to a local user instance of SQL Server Express 2005. If you change the default connection string used by the providers, then before you can use any of the ASP.NET features configured in the machine configuration, you must install the SQL Server database and the database elements for your chosen feature using Aspnet_regsql.exe. If the database that you specify with the SQL registration tool does not already exist (aspnetdb will be the default database if one is not specified on the command line), then the current user must have rights to create databases in SQL Server as well as to create schema elements within a database.

SQL Cache Dependency

An advanced feature of ASP.NET output caching is SQL cache dependency. SQL cache dependency supports two different modes of operation: one that uses an ASP.NET implementation of table polling and a second mode that uses the query notification features of SQL Server 2005. The SQL registration tool can be used to configure the table-polling mode of operation.

Session State

By default, session state values and information are stored in memory within the ASP.NET process. Alternatively, you can store session data in a SQL Server database, where it can be shared by multiple Web servers. If the database that you specify for session state with the SQL registration tool does not already exist, then the current

user must have rights to create databases in SQL Server as well as to create schema elements within a database. If the database does exist, then the current user must have rights to create schema elements in the existing database.

To install the session state database on SQL Server, run the Aspnet_regsql.exe tool and supply the following information with the command:

- The name of the SQL Server instance, using the **-S** option.
- The logon credentials for an account that has permission to create a database on a computer running SQL Server. Use the **-E** option to use the currently logged-on user, or use the **-U** option to specify a user ID along with the **-P** option to specify a password.
- The **-ssadd** command-line option to add the session state database.

By default, you cannot use the Aspnet_regsql.exe tool to install the session state database on a computer running SQL Server 2005 Express Edition.

The ASP.NET Browser Registration Tool - aspnet_regbrowsers.exe

In ASP.NET version 1.1, the Machine.config file contained a section called <browserCaps>. This section contained a series of XML entries that defined the configurations for various browsers based on a regular expression. For ASP.NET version 2.0, a new .BROWSER file defines the parameters of a particular browser using XML entries. You add information on a new browser by adding a new .BROWSER file to the folder located at %SystemRoot%\Microsoft.NET\Framework\version\CONFIG\Browsers on your system.

Because an application is not reading a .config file every time it requires browser information, you can create a new .BROWSER file and run Aspnet_regbrowsers.exe to add the required changes to the assembly. This allows the server to access the new browser information immediately so you do not have to shut down any of your applications to pick up the information. An application can access browser capabilities through the Browser property of the current HttpRequest.

The following options are available when running aspnet_regbrowser.exe:

OPTION	DESCRIPTION
-?	Displays the Aspnet_regbbrowsers.exe Help text in the command window.
-i	Creates the runtime browser capabilities assembly and installs it in the global assembly cache.
-u	Uninstalls the runtime browser capabilities assembly from the global assembly cache.

The ASP.NET Compilation Tool - aspnet_compiler.exe

The ASP.NET Compilation tool can be used in two general ways: for in-place compilation and compilation for deployment, where a target output directory is specified.

Compiling an Application in Place

The ASP.NET Compilation tool can compile an application in place, that is, it mimics the behavior of making multiple requests to the application, thus causing regular compilation. Users of a pre-compiled site will not experience a delay caused by compiling the page on first request.

When you precompile a site in place, the following items apply:

- The site retains its files and directory structure.
- You must have compilers for all programming languages used by the site on the server.
- If any file fails compilation, the entire site fails compilation.

You can also recompile an application in place after adding new source files to it. The tool compiles only the new or changed files unless you include the **-c** option.

> **NOTE**
>
> Compilation of an application that contains a nested application does not compile the nested application. The nested application must be compiled separately.

Compiling an Application for Deployment

You compile an application for deployment (compilation to a target location) by specifying the targetDir parameter. The targetDir can be the final location for the Web application, or the compiled application can be further deployed. Using the **-u** option compiles the application in such a way that you can make changes to certain files in the compiled application without recompiling it. Aspnet_compiler.exe makes a distinction between static and dynamic file types, and handles them differently when creating the resulting application.

- Static file types are those that do not have an associated compiler or build provider, such as files whose named have extensions such as .css, .gif, .htm, .html, .jpg, .js and so on. These files are simply copied to the target location, with their relative places in the directory structure preserved.
- Dynamic file types are those that have an associated compiler or build provider, including files with ASP.NET-specific file name extensions such as .asax, .ascx, .ashx, .aspx, .browser, .master, and so on. The ASP.NET Compilation tool generates assemblies from these files. If the **-u** option is omitted, the tool also creates files with the file name extension .COMPILED that map the original source files to their assembly. To ensure that the directory structure of the application source is preserved, the tool generates placeholder files in the corresponding locations in the target application.

You must use the **-u** option to indicate that the content of the compiled application can be modified. Otherwise, subsequent modifications are ignored or cause run-time errors.

The following table describes how the ASP.NET Compilation tool handles different file types when the **-u** option is included.

FILE TYPE	COMPILER ACTION
.ascx, .aspx, .master	These files are split into markup and source code, which includes both code-behind files and any code that is enclosed in <script runat="server"> elements. Source code is compiled into assemblies, with names that are derived from a hashing algorithm, and the assemblies are placed in the Bin directory. Any inline code, that is, code enclosed between the **<%** and **%>** brackets, is included with markup and not compiled. New files with the same name as the source files are created to contain the markup and placed in the corresponding output directories.
.ashx, .asmx	These files are not compiled and are moved to the output directories as is and not compiled. If you wish to have the handler code compiled, place the code into source code files in the App_Code directory.
.cs, .vb, .jsl, .cpp (not including code-behind files for the file types listed earlier)	These files are compiled and included as a resource in assemblies that reference them. Source files are not copied to the output directory. If a code file is not referenced, it is not compiled.
Custom file types	These files are not compiled. These files are copied to the corresponding output directories.

FILE TYPE	COMPILER ACTION
Source code files in the App_Code subdirectory	These files are compiled into assemblies and placed in the Bin directory.
.resx and .resource files in the App_GlobalResources subdirectory	These files are compiled into assemblies and placed in the Bin directory. No App_GlobalResources subdirectory is created under the main output directory, and no .resx or .resources files located in the source directory are copied to the output directories.
.resx and .resource files in the App_LocalResources subdirectory	These files are not compiled and are copied to the corresponding output directories.
.skin files in the App_Themes subdirectory	The .skin files and static theme files are not compiled and are copied to the corresponding output directories.
.browser Web.config Static file types Assemblies already present in the Bin directory	These files are copied as is to the output directories.

The following table describes how the ASP.NET Compilation tool handles different file types when the **-u** option is omitted.

FILE TYPE	COMPILER ACTION
.aspx, .asmx, .ashx, .master	These files are split into markup and source code, which includes both code-behind files and any code that is enclosed in <script runat="server"> elements. Source code is compiled into assemblies, with names that are derived from a hashing algorithm. The resulting assemblies are placed in the Bin directory. Any inline code, that is, code enclosed between the **<%** and **%>** brackets, is included with markup and not compiled. The compiler creates new files to contain the markup with the same name as the source files. These resulting files are placed in the Bin directory. The compiler also creates files with the same name as the source files but with the extension .COMPILED that contain mapping information. The .COMPILED files are placed in the output directories corresponding to the original location of the source files.
.ascx	These files are split into markup and source code. Source code is compiled into assemblies and placed in the Bin directory, with names that are derived from a hashing algorithm. No markup files are generated.
.cs, .vb, .jsl, .cpp (not including code-behind files for the file types listed earlier)	Source code that is referenced by the assemblies generated from .ascx, .ashx, or .aspx files is compiled into assemblies and placed in the Bin directory. No source files are copied.
Custom file types	These files are compiled like dynamic files. Depending on the type of file they are based on, the compiler can place mapping files in the output directories.
Files in the App_Code subdirectory	Source code files in this subdirectory are compiled into assemblies and placed in the Bin directory.

FILE TYPE	COMPILER ACTION
Files in the App_GlobalResources subdirectory	These files are compiled into assemblies and placed in the Bin directory. No App_GlobalResources subdirectory is created under the main output directory. If the configuration file specifies appliesTo="All", .resx and .resources files are copied to the output directories. They are not copied if they are referenced by a BuildProvider.
.resx and .resource files in the App_LocalResources subdirectory	These files are compiled into assemblies with unique names and placed in the Bin directory. No .resx or .resource files are copied to the output directories.
.skin files in the App_Themes subdirectory	Themes are compiled into assemblies and placed in the Bin directory. Stub files are created for .skin files and placed in the corresponding output directory. Static files (such as .css) are copied to the output directories.
.browser Web.config Static file types Assemblies already present in the Bin directory	These files are copied as is to the output directory.

Fixed Assembly Names

Some scenarios, such as deploying a Web application using the MSI Windows Installer, require the use of consistent file names and contents, as well as consistent directory structures to identify assemblies or configuration settings for updates. In those cases, you can use the **-fixednames** option to specify that the ASP.NET Compilation tool should compile an assembly for each source file instead of using the where multiple pages are compiled into assemblies. This can lead to a large number of assemblies, so if you are concerned with scalability you should use this option with caution.

Strong-Name Compilation

The **-aptca**, **-delaysign**, **-keycontainer** and **-keyfile** options are provided so that you can use Aspnet_compiler.exe to create strongly named assemblies without using the Strong Name Tool (Sn.exe) separately. These options correspond, respectively, to **AllowPartiallyTrustedCallersAttribute**, **AssemblyDelaySignAttribute**, **AssemblyKeyNameAttribute**, and **AssemblyKeyFileAttribute**.

Discussion of these attributes is outside of the scope of this course.

Labs

Each of the following labs builds on the previous labs. You will need to do them in order.

Lab 1: Using the Configuration API

1. Create a new Web site called *mod9lab*.
2. Add a new Web Configuration File to the site.
3. Add the following to the web.config file:

```
<authorization>
    <deny users="?"/>
</authorization>

<identity impersonate="true"/>
```

This will ensure that you have permission to save changes to the web.config file.

1. Add a new Label control to Default.aspx and change the ID to **lblDebugStatus**.

2. Add a new Button control to Default.aspx.
3. Change the Button control's ID to **btnToggleDebug** and the Text to **Toggle Debug Status**.
4. Open the Code View for the code-behind file of Default.aspx and add a **using** statement for **System.Web.Configuration** as follows:

```
using System.Web.Configuration;
```

1. Add two private variables to the class and a Page_Init method as shown below:

```
public partial class _Default : System.Web.UI.Page {
    private bool _debugStatus;
    private CompilationSection compilation;
    private Configuration config;
    protected void Page_Init(object sender, EventArgs e) {
        config = WebConfigurationManager.OpenWebConfiguration("/mod9lab");
        compilation =
            (CompilationSection)config.GetSection("system.web/compilation");
        _debugStatus = compilation.Debug;
    }
}
```

1. Add the following code to Page_Load:

```
lblDebugStatus.Text = "Debug set to: " + _debugStatus.ToString();
```

1. Save and browse default.aspx. Notice that the Label control displays the current debug status.
2. Double-click on the Button control in the designer and add the following code to the Click event for the Button control:

```
compilation.Debug = !_debugStatus;
config.Save();
lblDebugStatus.Text = "Debug set to: " + compilation.Debug.ToString();
```

1. Save and browse default.aspx and click the button.
2. Open the web.config file after each button click and observe the **debug** attribute in the <compilation> section.

Lab 2: Logging Application Restarts

In this lab, you will create code that will allow you to toggle the logging of application shutdowns, startups, and recompilations in the Event Viewer.

1. Add a DropDownList to default.aspx and change the ID to ddlLogAppEvents.
2. Set the **AutoPostBack** property for the DropDownList to **true**.
3. Add three items to the Items collection for the DropDownList. Make the **Text** for the first item *Select Value* and the value -1. Make the **Text** and **Value** of the second item **True** and the **Text** and **Value** of the third item **False**.
4. Add a new Label to default.aspx. Change the ID to **lblLogAppEvents**.
5. Open the code-behind view for default.aspx and add a new declaration for a variable of type HealthMonitoringSection as shown below:

```
public partial class _Default : System.Web.UI.Page {
    private bool _debugStatus;
    private CompilationSection compilation;
    private Configuration config;

    // new variable below
    private HealthMonitoringSection health;
}
```

1. Add the following code to the existing code in Page_Init:

```
health = (HealthMonitoringSection)config.GetSection("system.web/healthMonitoring");
```

1. Double-click on the DropDownList and add the following code to the SelectedIndexChanged event:

```
if (ddlLogAppEvents.SelectedValue != "-1") {
    if (Convert.ToBoolean(ddlLogAppEvents.SelectedValue)) {
        RuleSettings appRules = new
        RuleSettings("AppRestartEvents",
        "Application Lifetime Events",
        "EventLogProvider");
        health.Rules.Add(appRules);
        config.Save();
    } else {
        health.Rules.Remove("AppRestartEvents");
        config.Save();
    }
}
```

1. Browse default.aspx.
2. Set the dropdown to **False**.
3. Clear the Application log in the Event Viewer.
4. Click the button to change the Debug attribute for the application.
5. Refresh the Application log in the Event Viewer.
   a. Were any events logged?
   b. Why or why not?
6. Set the dropdown to **True**.
7. Click the button to toggle the Debug attribute for the application.
8. Refresh the Application login the Event Viewer.
   a. Were any events logged?
   b. What was the reason for the app shutdown?
9. Experiment with turning on and off logging and look at the changes made to the web.config file.

More Information:

ASP.NET 2.0's Provider model allows you to create your own providers for not only application instrumentation, but for many other uses as well such as Membership, Profiles, etc. For detailed information on writing a custom provider to log application events to a text file, visit this link.

Profiles, Themes, and Web Parts

4/10/2020 • 42 minutes to read • Edit Online

by Microsoft

> There are major changes in configuration and instrumentation in ASP.NET 2.0. The new ASP.NET configuration API allows for configuration changes to be made programmatically. In addition, many new configuration settings exist allow for new configurations and instrumentation.

ASP.NET 2.0 represents a substantial improvement in the area of personalized Web sites. In addition to the membership features we've already covered, ASP.NET profiles, themes, and Web parts significantly enhance personalization in Web sites.

ASP.NET Profiles

ASP.NET profiles are similar to sessions. The difference is that a profile is persistent whereas a session is lost when the browser is closed. Another big difference between sessions and profiles is that profiles are strongly typed, therefore providing you with IntelliSense during the development process.

A profile is defined in either the machines configuration file or the web.config file for the application. (You cannot define a profile in a sub-folders web.config file.) The code below defines a profile to store the Web site visitors first and last name.

```
<profile>
    <properties>
        <add name="FirstName" />
        <add name="LastName" />
    </properties>
</profile>
```

The default data type for a profile property is System.String. In the above example, no data type was specified. Therefore the FirstName and LastName properties are both of type String. As previously mentioned, profile properties are strongly typed. The code below adds a new property for age that is of type Int32.

```
<profile>
    <properties>
        <add name="FirstName" />
        <add name="LastName" />
        <add name="Age" type="Int32"/>
    </properties>
</profile>
```

Profiles are generally used with ASP.NET Forms authentication. When used in combination with Forms authentication, each user has a separate profile associated with their user ID. However, it is also possible to allow the use of profiles in an anonymous application using the <anonymousIdentification> element in the configuration file along with the **allowAnonymous** attribute as shown below:

```
<anonymousIdentification enabled="true" />
<profile>
    <properties>
        <add name="FirstName" allowAnonymous="true" />
        <add name="LastName" allowAnonymous="true" />
    </properties>
</profile>
```

When an anonymous user browses the site, ASP.NET creates an instance of **ProfileCommon** for the user. This profile uses a unique ID stored in a cookie on the browser to identify the user as a unique visitor. In this way, you can store profile information for users who are browsing anonymously.

Profile Groups

It is possible to group properties of profiles. By grouping properties, it is possible to simulate multiple profiles for a specific application.

The following configuration configures a FirstName and LastName property for two groups; Buyers and Prospects.

```
<profile>
    <properties>
        <group name="Buyers">
            <add name="FirstName" />
            <add name="Lastname" />
            <add name="NumberOfPurchases" type="Int32" />
        </group>
        <group name="Prospects">
            <add name="FirstName" />
            <add name="Lastname" />
        </group>
    </properties>
</profile>
```

It is then possible to set properties on a particular group as follows:

```
Profile.Buyers.NumberOfPurchases += 1;
```

Storing Complex Objects

So far, the examples we've covered have stored simple data types in a profile. It is also possible to store complex data types in a profile by specifying the method of serialization using the **serializeAs** attribute as follows:

```
<add name="PurchaseInvoice"
     type="PurchaseInvoice"
     serializeAs="Binary"
/>
```

In this case, the type is PurchaseInvoice. The PurchaseInvoice class needs to be marked as serializable and can contain any number of properties. For example, if PurchaseInvoice has a property called **NumItemsPurchased**, you can refer to that property in code as follows:

```
Profile.PurchaseInvoice.NumItemsPurchased
```

Profile Inheritance

It is possible to create a profile for use in multiple applications. By creating a profile class that derives from ProfileBase, you can reuse a profile in several applications by using the **inherits** attribute as shown below:

```
<profile inherits="PurchasingProfile" />
```

In this case, the class **PurchasingProfile** would look like so:

```
using System;
using System.Web.Profile;
public class PurchasingProfile : ProfileBase {
    private string _ProductName;
    private Int32 _ProductID;
    public string ProductName {
        get { return _ProductName; }
        set { _ProductName = value; }
    }
    public Int32 ProductID {
        get { return _ProductID; }
        set { _ProductID = value; }
    }
}
```

Profile Providers

ASP.NET profiles use the provider model. The default provider stores the information in a SQL Server Express database in the App_Data folder of the Web application using the SqlProfileProvider provider. If the database doesn't exist, ASP.NET will automatically create it when the profile attempts to store information.

In some cases, however, you may want to develop your own profile provider. The ASP.NET profile feature enables you to easily use different providers.

You create a custom profile provider when:

- You need to store profile information in a data source, such as in a FoxPro database or in an Oracle database, that is not supported by the profile providers included with the .NET Framework.
- You need to manage profile information using a database schema that is different from the database schema used by the providers included with the .NET Framework. A common example is that you want to integrate profile information with user data in an existing SQL Server database.

Required Classes

To implement a profile provider, you create a class that inherits the System.Web.Profile.ProfileProvider abstract class. The **ProfileProvider** abstract class in turn inherits the System.Configuration.SettingsProvider abstract class, which inherits the System.Configuration.Provider.ProviderBase abstract class. Because of this inheritance chain, in addition to the required members of the **ProfileProvider** class, you must implement the required members of the **SettingsProvider** and **ProviderBase** classes.

The following tables describe the properties and methods that you must implement from the **ProviderBase**, **SettingsProvider**, and **ProfileProvider** abstract classes.

ProviderBase Members

MEMBER	DESCRIPTION

MEMBER	DESCRIPTION
Initialize method	Takes as input the name of the provider instance and a NameValueCollection of configuration settings. Used to set options and property values for the provider instance, including implementation-specific values and options specified in the machine configuration or Web.config file.

SettingsProvider Members

MEMBER	DESCRIPTION
ApplicationName property	The application name that is stored with each profile. The profile provider uses the application name to store profile information separately for each application. This enables multiple ASP.NET applications to use the same data source without a conflict if the same user name is created in different applications. Alternatively, multiple ASP.NET applications can share a profile data source by specifying the same application name.
GetPropertyValues method	Takes as input a SettingsContext and a SettingsPropertyCollection object. The **SettingsContext** provides information about the user. You can use the information as a primary key to retrieve profile property information for the user. Use the **SettingsContext** object to get the user name and whether the user is authenticated or anonymous. The **SettingsPropertyCollection** contains a collection of SettingsProperty objects. Each **SettingsProperty** object provides the name and type of the property as well as additional information such as the default value for the property and whether the property is read-only. The **GetPropertyValues** method populates a SettingsPropertyValueCollection with SettingsPropertyValue objects based on the **SettingsProperty** objects provided as input. The values from the data source for the specified user are assigned to the PropertyValue properties for each **SettingsPropertyValue** object and the entire collection is returned. Calling the method also updates the LastActivityDate value for the specified user profile to the current date and time.
SetPropertyValues method	Takes as input a **SettingsContext** and a **SettingsPropertyValueCollection** object. The **SettingsContext** provides information about the user. You can use the information as a primary key to retrieve profile property information for the user. Use the **SettingsContext** object to get the user name and whether the user is authenticated or anonymous. The **SettingsPropertyValueCollection** contains a collection of **SettingsPropertyValue** objects. Each **SettingsPropertyValue** object provides the name, type, and value of the property as well as additional information such as the default value for the property and whether the property is read-only. The **SetPropertyValues** method updates the profile property values in the data source for the specified user. Calling the method also updates the **LastActivityDate** and LastUpdatedDate values for the specified user profile to the current date and time.

ProfileProvider Members

MEMBER	DESCRIPTION
DeleteProfiles method	Takes as input a string array of user names and deletes from the data source all profile information and property values for the specified names where the application name matches the **ApplicationName** property value. If your data source supports transactions, it is recommended that you include all delete operations in a transaction and that you roll back the transaction and throw an exception if any delete operation fails.
DeleteProfiles method	Takes as input a collection of ProfileInfo objects and deletes from the data source all profile information and property values for each profile where the application name matches the **ApplicationName** property value. If your data source supports transactions, it is recommended that you include all delete operations in a transaction and roll back the transaction and throw an exception if any delete operation fails.
DeleteInactiveProfiles method	Takes as input a ProfileAuthenticationOption value and a DateTime object and deletes from the data source all profile information and property values where the last activity date is less than or equal to the specified date and time and where the application name matches the **ApplicationName** property value. The **ProfileAuthenticationOption** parameter specifies whether only anonymous profiles, only authenticated profiles, or all profiles are to be deleted. If your data source supports transactions, it is recommended that you include all delete operations in a transaction and roll back the transaction and throw an exception if any delete operation fails.
GetAllProfiles method	Takes as input a **ProfileAuthenticationOption** value, an integer that specifies the page index, an integer that specifies the page size, and a reference to an integer that will be set to the total count of profiles. Returns a ProfileInfoCollection that contains **ProfileInfo** objects for all profiles in the data source where the application name matches the **ApplicationName** property value. The **ProfileAuthenticationOption** parameter specifies whether only anonymous profiles, only authenticated profiles, or all profiles are to be returned. The results returned by the **GetAllProfiles** method are constrained by the page index and page size values. The page size value specifies the maximum number of **ProfileInfo** objects to return in the **ProfileInfoCollection**. The page index value specifies which page of results to return, where 1 identifies the first page. The parameter for total records is an out parameter (you can use **ByRef** in Visual Basic) that is set to the total number of profiles. For example, if the data store contains 13 profiles for the application and the page index value is 2 with a page size of 5, the **ProfileInfoCollection** returned contains the sixth through the tenth profiles. The total records value is set to 13 when the method returns.

MEMBER	DESCRIPTION
GetAllInactiveProfiles method	Takes as input a **ProfileAuthenticationOption** value, a **DateTime** object, an integer that specifies the page index, an integer that specifies the page size, and a reference to an integer that will be set to the total count of profiles. Returns a **ProfileInfoCollection** that contains **ProfileInfo** objects for all profiles in the data source where the last activity date is less than or equal to the specified **DateTime** and where the application name matches the **ApplicationName** property value. The **ProfileAuthenticationOption** parameter specifies whether only anonymous profiles, only authenticated profiles, or all profiles are to be returned. The results returned by the **GetAllInactiveProfiles** method are constrained by the page index and page size values. The page size value specifies the maximum number of **ProfileInfo** objects to return in the **ProfileInfoCollection**. The page index value specifies which page of results to return, where 1 identifies the first page. The parameter for total records is an out parameter (you can use **ByRef** in Visual Basic) that is set to the total number of profiles. For example, if the data store contains 13 profiles for the application and the page index value is 2 with a page size of 5, the **ProfileInfoCollection** returned contains the sixth through the tenth profiles. The total records value is set to 13 when the method returns.
FindProfilesByUserName method	Takes as input a **ProfileAuthenticationOption** value, a string containing a user name, an integer that specifies the page index, an integer that specifies the page size, and a reference to an integer that will be set to the total count of profiles. Returns a **ProfileInfoCollection** that contains **ProfileInfo** objects for all profiles in the data source where the user name matches the specified user name and where the application name matches the **ApplicationName** property value. The **ProfileAuthenticationOption** parameter specifies whether only anonymous profiles, only authenticated profiles, or all profiles are to be returned. If your data source supports additional search capabilities, such as wildcard characters, you can provide more extensive search capabilities for user names. The results returned by the **FindProfilesByUserName** method are constrained by the page index and page size values. The page size value specifies the maximum number of **ProfileInfo** objects to return in the **ProfileInfoCollection**. The page index value specifies which page of results to return, where 1 identifies the first page. The parameter for total records is an out parameter (you can use **ByRef** in Visual Basic) that is set to the total number of profiles. For example, if the data store contains 13 profiles for the application and the page index value is 2 with a page size of 5, the **ProfileInfoCollection** returned contains the sixth through the tenth profiles. The total records value is set to 13 when the method returns.

MEMBER	DESCRIPTION
FindInactiveProfilesByUserName method	Takes as input a **ProfileAuthenticationOption** value, a string containing a user name, a **DateTime** object, an integer that specifies the page index, an integer that specifies the page size, and a reference to an integer that will be set to the total count of profiles. Returns a **ProfileInfoCollection** that contains **ProfileInfo** objects for all profiles in the data source where the user name matches the specified user name, where the last activity date is less than or equal to the specified **DateTime**, and where the application name matches the **ApplicationName** property value. The **ProfileAuthenticationOption** parameter specifies whether only anonymous profiles, only authenticated profiles, or all profiles are to be returned. If your data source supports additional search capabilities, such as wildcard characters, you can provide more extensive search capabilities for user names. The results returned by the **FindInactiveProfilesByUserName** method are constrained by the page index and page size values. The page size value specifies the maximum number of **ProfileInfo** objects to return in the **ProfileInfoCollection**. The page index value specifies which page of results to return, where 1 identifies the first page. The parameter for total records is an out parameter (you can use **ByRef** in Visual Basic) that is set to the total number of profiles. For example, if the data store contains 13 profiles for the application and the page index value is 2 with a page size of 5, the **ProfileInfoCollection** returned contains the sixth through the tenth profiles. The total records value is set to 13 when the method returns.
GetNumberOfInActiveProfiles method	Takes as input a **ProfileAuthenticationOption** value and a **DateTime** object and returns a count of all profiles in the data source where the last activity date is less than or equal to the specified **DateTime** and where the application name matches the **ApplicationName** property value. The **ProfileAuthenticationOption** parameter specifies whether only anonymous profiles, only authenticated profiles, or all profiles are to be counted.

ApplicationName

Because profile providers store profile information separately for each application, you must ensure that your data schema includes the application name and that queries and updates also include the application name. For example, the following command is used to retrieve a property value from a database based on the user name and whether the profile is anonymous, and ensures that the **ApplicationName** value is included in the query.

```
SELECT Property
FROM PropertyTable
WHERE Username = 'user1'
AND IsAnonymous = False
AND ApplicationName = 'MyApplication'
```

ASP.NET Themes

What are ASP.NET 2.0 Themes?

One of the most important aspects of a Web application is a consistent look and feel across the site. ASP.NET 1.x developers usually use Cascading Style Sheets (CSS) to implement a consistent look and feel. ASP.NET 2.0 themes

significantly improve upon CSS because they give the ASP.NET developer the ability to define the appearance of ASP.NET server controls as well as HTML elements. ASP.NET themes can be applied to individual controls, a specific Web page, or an entire Web application. Themes use a combination of CSS files, an optional skin file, and an optional Images directory if images are needed. The skin file controls the visual appearance of ASP.NET server controls.

Where are Themes Stored?

The location where themes are stored differs based upon their scope. Themes that can be applied to any application are stored in the following folder:

`C:\WINDOWS\Microsoft.NET\Framework\v2.x.xxxxx\ASP.NETClientFiles\Themes\<Theme_Name>`

A theme that is specific to a particular application is stored in an `App\_Themes\<Theme\_Name>` directory in the root of the Web site.

> **NOTE**
> A skin file should only modify server control properties that affect appearance.

A global theme is a theme that can be applied to any application or Web site running on the Web server. These themes are stored by default in the ASP.NETClientfiles\Themes directory that is inside of the v2.x.xxxxx directory. Alternatively, you can move the theme files into the aspnet_client/system_web/[version]/Themes/[theme_name] folder in the root of your Web site.

Application-specific themes can only be applied to the application in which the files reside. These files are stored in the `App\_Themes/<theme\_name>` directory in the root of the Web site.

The Components of a Theme

A theme is made up of one or more CSS files, an optional skin file, and an optional Images folder. The CSS files can be any name you wish (i.e. default.css or theme.css, etc.) and must be in the root of the themes folder. The CSS files are used to define ordinary CSS classes and attributes for specific selectors. To apply one of the CSS classes to a page element, the **CSSClass** property is used.

The skin file is an XML file that contains property definitions for ASP.NET server controls. The code listed below is an example skin file.

```
<asp:TextBox runat="server"
    BackColor="#FFC080"
    BorderColor="Black"
    BorderStyle="Solid"
    BorderWidth="1px"
    Font-Names="Tahoma, Verdana, Arial"
    Font-Size="Smaller" />

<asp:Button runat="server"
    BackColor="#C04000"
    BorderColor="Maroon"
    BorderStyle="Solid"
    BorderWidth="2px"
    Font-Names="Tahoma,Verdana,Arial"
    Font-Size="Smaller"
    ForeColor="#FFFFC0" />
```

Figure 1 below shows a small ASP.NET page browsed without a theme applied. **Figure 2** shows the same file with a theme applied. The background color and text color are configured via a CSS file. The appearance of the

button and textbox are configured using the skin file listed above.

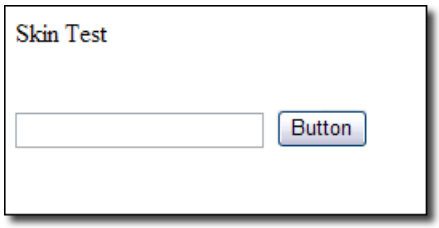


Figure 1: No Theme

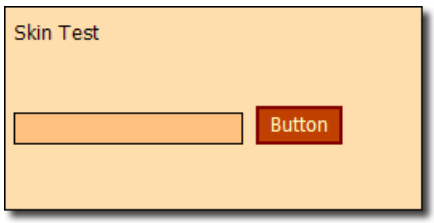


Figure 2: Theme Applied

The skin file listed above defines a default skin for all TextBox controls and Button controls. That means that every TextBox control and Button control inserted on a page will take on this appearance. You can also define a skin that can be applied to specific instances of these controls using the **SkinID** property of the control.

The code below defines a skin for a Button control. Only Button controls with a **SkinID** property of **goButton** will take on the appearance of the skin.

```
<asp:Button runat="server"
    BackColor="#C04000"
    BorderColor="Maroon"
    BorderStyle="Solid"
    BorderWidth="2px"
    Font-Names="Tahoma,Verdana,Arial"
    Font-Size="Smaller"
    ForeColor="#FFFFC0"
    Text=go
    SkinID=goButton
    Width="95px" />
```

You can only have one default skin per server control type. If you require additional skins, you should use the SkinID property.

Applying Themes to Pages

A theme can be applied using any one of the following methods:

- In the <pages> element of the web.config file
- In the @Page directive of a page
- Programmatically

Applying a Theme in the Configuration File

To apply a theme in the applications configuration file, use the following syntax:

```
<system.web>
    <pages theme="CoolTheme" />
    ...
</system.web>
```

The theme name specified here must match the name of the themes folder. This folder can exist either in any one of the locations mentioned earlier in this course. If you attempt to apply a theme that doesn't exist, a configuration error will occur.

Applying a Theme in the Page Directive

You can also apply a theme in the @ Page directive. This method allows you to use a theme for a specific page.

To apply a theme in the @Page directive, use the following syntax:

```
<%@ Page Language="C#" Theme=CoolTheme CodeFile="Default.aspx.cs" ... %>
```

Once again, the theme specified here must match the theme folder as mentioned previously. If you attempt to apply a theme that doesn't exist, a build failure will occur. Visual Studio will also highlight the attribute and notify you that no such theme exists.

Applying a Theme Programmatically

To apply a theme programmatically, you must specify the **Theme** property for the page in the **Page_PreInit** method.

To apply a theme programmatically, use the following syntax:

```
Page.Theme = CoolTheme;
```

It is necessary to apply the theme in the PreInit method due to the page lifecycle. If you apply it after that point, the pages theme will have already been applied by the runtime and a change at that point is too late in the lifecycle. If you apply a theme that doesn't exist, an **HttpException** occurs. When a theme is applied programmatically, a build warning will occur if any server controls have a SkinID property specified. This warning is intended to inform you that no theme is declaratively applied and it can be ignored.

Exercise 1 : Applying a Theme

In this exercise, you will apply an ASP.NET theme to a Web site.

> **IMPORTANT**
> If you are using Microsoft Word to enter information into a skin file, make sure that you are not replacing regular quotes with smart quotes. Smart quotes will cause problems with skin files.

1. Create a new ASP.NET Web site.
2. Right-click on the project in Solution Explorer and choose Add New Item.
3. Choose Web Configuration File from the list of files and click Add.
4. Right-click on the project in Solution Explorer and choose Add New Item.
5. Choose Skin File and click Add.

6. Click Yes when asked if you'd like to place the file inside of the App_Themes folder.
7. Right-click on the SkinFile folder inside of the App_Themes folder in Solution Explorer and choose Add New Item.
8. Choose Style Sheet from the list of files and click Add. You now have all of the files necessary to implement your new theme. However, Visual Studio has named your themes folder SkinFile. Right-click on that folder and change the name to CoolTheme.
9. Open the SkinFile.skin file and add the following code the end of the file:

```
<asp:TextBox runat="server"
    BackColor="#FFC080"
    BorderColor="Black"
    BorderStyle="Solid"
    BorderWidth="1px"
    Font-Names="Tahoma, Verdana, Arial"
    Font-Size="Smaller"
/>

<asp:Button runat="server"
    BackColor="#C04000"
    BorderColor="Maroon"
    BorderStyle="Solid"
    BorderWidth="2px"
    Font-Names="Tahoma,Verdana,Arial"
    Font-Size="Smaller"
    ForeColor="#FFFFC0"
/>

<asp:Button runat="server"
    BackColor="#C04000"
    BorderColor="Maroon"
    BorderStyle="Solid"
    BorderWidth="2px"
    Font-Names="Tahoma,Verdana,Arial"
    Font-Size="Smaller"
    ForeColor="#FFFFC0"
    Text="go"
    SkinID="goButton"
    Width="95px"
/>
```

10. Save the SkinFile.skin file.
11. Open the StyleSheet.css.
12. Replace all of the text in it with the following:

```
body {
    background-color: #FFDEAD;
}
```

13. Save the StyleSheet.css file.
14. Open the Default.aspx page.
15. Add a TextBox control and a Button control.
16. Save the page. Now browse the Default.aspx page. It should display as a normal Web form.
17. Open the web.config file.

18. Add the following directly underneath the opening `<system.web>` tag:

```
<pages theme="CoolTheme" />
```

19. Save the web.config file. Now browse the Default.aspx page. It should display with the theme applied.
20. If it's not already open, open the Default.aspx page in Visual Studio.
21. Select the Button.
22. Change the **SkinID** property to goButton. Notice that Visual Studio provides a dropdown with valid SkinID values for a Button control.
23. Save the page. Now preview the page in your browser again. The Button should now say "go" and should be wider in appearance.

Using the **SkinID** property, you can easily configure different skins for different instances of a particular type of server control.

The StyleSheetTheme Property

So far, we've talked only about applying themes using the Theme property. When using the Theme property, the skin file will override any declarative settings for server controls. For example, in exercise 1, you specified a SkinID of "goButton" for the Button control and that changed the Button's text to "go". You may have noticed that the Text property of the Button in the designer was set to "Button", but the theme overrode that. The theme will always override any property settings in the designer.

If you'd like to be able to override the properties defined in the theme's skin file with properties specified in the designer, you can use the **StyleSheetTheme** property instead of the Theme property. The StyleSheetTheme property is the same as the Theme property except that it does not override all explicit property settings like the Theme property does.

To see this in action, open the web.config file from the project in exercise 1 and change the `<pages>` element to the following:

```
<pages styleSheetTheme="CoolTheme" />
```

Now browse the Default.aspx page and you'll see that the Button control has a Text property of "Button" again. That's because the explicit property setting in the designer is overriding the Text property set by the goButton SkinID.

Overriding Themes

A global theme can be overridden by applying a theme by the same name in the App_Themes folder of the application. However, the theme is not applied in a true override scenario. If the runtime encounters theme files in the App_Themes folder, it will apply the theme using those files and will ignore the global theme.

The StyleSheetTheme property is overridable and can be overridden in code as follows:

```
const String THEME_NAME = "CoolTheme";
public override string StyleSheetTheme {
    get { return THEME_NAME; }
    set { Page.StyleSheetTheme = THEME_NAME; }
}
```

Web Parts

ASP.NET Web Parts is an integrated set of controls for creating Web sites that enable end users to modify the content, appearance, and behavior of Web pages directly from a browser. The modifications can be applied to all users on the site or to individual users. When users modify pages and controls, the settings can be saved to retain a user's personal preferences across future browser sessions, a feature called personalization. These Web Parts capabilities mean that developers can empower end users to personalize a Web application dynamically, without developer or administrator intervention.

Using the Web Parts control set, you as a developer can enable end users to:

- Personalize page content. Users can add new Web Parts controls to a page, remove them, hide them, or minimize them like ordinary windows.
- Personalize page layout. Users can drag a Web Parts control to a different zone on a page, or change its appearance, properties, and behavior.
- Export and import controls. Users can import or export Web Parts control settings for use in other pages or sites, retaining the properties, appearance, and even the data in the controls. This reduces data entry and configuration demands on end users.
- Create connections. Users can establish connections between controls so that, for example, a chart control could display a graph for the data in a stock ticker control. Users could personalize not only the connection itself, but the appearance and details of how the chart control displays the data.
- Manage and personalize site-level settings. Authorized users can configure site-level settings, determine who can access a site or page, set role-based access to controls, and so on. For example, a user in an administrative role could set a Web Parts control to be shared by all users, and prevent users who are not administrators from personalizing the shared control.

You will typically work with Web Parts in one of three ways: creating pages that use Web Parts controls, creating individual Web Parts controls, or creating complete, personalizable Web applications, such as a portal.

Page Development

Page developers can use visual design tools such as Microsoft Visual Studio 2005 to create pages that use Web Parts. One advantage in using a tool such as Visual Studio is that the Web Parts control set provides features for drag-and-drop creation and configuration of Web Parts controls in a visual designer. For example, you can use the designer to drag a Web Parts zone, or a Web Parts editor control, onto the design surface, and then configure the control right in the designer using the UI provided by the Web Parts control set. This can speed development of Web Parts applications and reduce the amount of code you have to write.

Control Development

You can use any existing ASP.NET control as a Web Parts control, including standard Web server controls, custom server controls, and user controls. For maximum programmatic control of your environment, you can also create custom Web Parts controls that derive from the WebPart class. For individual Web Parts control development, you will typically either create a user control and use it as a Web Parts control, or develop a custom Web Parts control.

As an example of developing a custom Web Parts control, you could create a control to provide any of the features provided by other ASP.NET server controls that might be useful to package as a personalizable Web Parts control: calendars, lists, financial information, news, calculators, rich text controls for updating content, editable grids that connect to databases, charts that dynamically update their displays, or weather and travel information. If you provide a visual designer with your control, then any page developer using Visual Studio can simply drag your control into a Web Parts zone and configure it at design time without having to write additional code.

Personalization is the foundation of the Web Parts feature. It enables users to modify--or personalize--the layout, appearance, and behavior of Web Parts controls on a page. The personalized settings are long-lived: they are

persisted not just during the current browser session (as with view state), but also in long-term storage, so that a user's settings are saved for future browser sessions as well. Personalization is enabled by default for Web Parts pages.

The UI structural components rely on personalization and provide the core structure and services needed by all Web Parts controls. One UI structural component required on every Web Parts page is the WebPartManager control. Although never visible, this control has the critical task of coordinating all Web Parts controls on a page. For example, it tracks all the individual Web Parts controls. It manages Web Parts zones (regions that contain Web Parts controls on a page), and which controls are in which zones. It also tracks and controls the different display modes a page can be in, such as browse,connect, edit, or catalog mode, and whether personalization changes apply to all users or to individual users. Finally, it initiates and tracks connections and communication between Web Parts controls.

The second kind of UI structural component is the zone. Zones act as layout managers on a Web Parts page. They contain and organize controls that derive from the Part class (part controls), and provide the ability to do modular page layout in either horizontal or vertical orientation. Zones also offer common and consistent UI elements (such as header and footer style, title, border style, action buttons, and so on) for each control they contain; these common elements are known as the chrome of a control. Several specialized types of zones are used in the different display modes and with various controls. The different types of zones are described in the Web Parts Essential Controls section below.

The Web Parts UI controls, all of which derive from the **Part** class, comprise the primary UI on a Web Parts page. The Web Parts control set is flexible and inclusive in the options it gives you for creating part controls. In addition to creating your own custom Web Parts controls, you can also use existing ASP.NET server controls, user controls, or custom server controls as Web Parts controls. The essential controls that are most commonly used for creating Web Parts pages are described in the next section.

Web Parts Essential Controls

The Web Parts control set is extensive, but some controls are essential either because they are required for Web Parts to work, or because they are the controls most frequently used on Web Parts pages. As you begin using Web Parts and creating basic Web Parts pages, it is helpful to be familiar with the essential Web Parts controls described in the following table.

WEB PARTS CONTROL	DESCRIPTION
WebPartManager	Manages all Web Parts controls on a page. One (and only one) **WebPartManager** control is required for every Web Parts page.
CatalogZone	Contains CatalogPart controls. Use this zone to create a catalog of Web Parts controls from which users can select controls to add to a page.
EditorZone	Contains EditorPart controls. Use this zone to enable users to edit and personalize Web Parts controls on a page.
WebPartZone	Contains and provides overall layout for the WebPart controls that compose the main UI of a page. Use this zone whenever you create pages with Web Parts controls. Pages can contain one or more zones.
ConnectionsZone	Contains WebPartConnection controls, and provides a UI for managing connections.

WEB PARTS CONTROL	DESCRIPTION
WebPart (GenericWebPart)	Renders the primary UI; most Web Parts UI controls fall into this category. For maximum programmatic control, you can create custom Web Parts controls that derive from the base **WebPart** control. You can also use existing server controls, user controls, or custom controls as Web Parts controls. Whenever any of these controls are placed in a zone, the **WebPartManager** control automatically wraps them with **GenericWebPart** controls at run time so that you can use them with Web Parts functionality.
CatalogPart	Contains a list of available Web Parts controls that users can add to the page.
WebPartConnection	Creates a connection between two Web Parts controls on a page. The connection defines one of the Web Parts controls as a provider (of data), and the other as a consumer.
EditorPart	Serves as the base class for the specialized editor controls.
EditorPart controls (AppearanceEditorPart, LayoutEditorPart, BehaviorEditorPart, and PropertyGridEditorPart)	Allow users to personalize various aspects of Web Parts UI controls on a page

Lab: Create a Web Part Page

In this lab, you will create a Web part page that will persist information via ASP.NET profiles.

Creating a Simple Page with Web Parts

In this part of the walkthrough, you create a page that uses Web Parts controls to show static content. The first step in working with Web Parts is to create a page with two required structural elements. First, a Web parts page needs a WebPartManager control to track and coordinate all Web Parts controls. Second, a Web Parts page needs one or more zones, which are composite controls that contain WebPart or other server controls and occupy a specified region of a page.

> **NOTE**
> You do not need to do anything to enable Web Parts personalization; it is enabled by default for the Web Parts control set. When you first run a Web Parts page on a site, ASP.NET sets up a default personalization provider to store user personalization settings. For more information about personalization, see Web Parts Personalization Overview.

To create a page for containing Web Parts controls

1. Close the default page and add a new page to the site named WebPartsDemo.aspx.
2. Switch to **Design** view.
3. From the **View** menu, make sure that the **Non-Visual Controls** and **Details** options are selected so you can see layout tags and controls that do not have a UI.
4. Place the insertion point before the `<div>` tags on the design surface, and press ENTER to add a new line. Position the insertion point before the new line character, click the **Block Format** drop-down list control on the menu, and select the **Heading 1** option. In the heading, add the text **Web Parts Demonstration Page**.
5. From the **WebParts** tab of the Toolbox, drag a **WebPartManager** control onto the page, positioning it just after the new line character and before the `<div>` tags.

The **WebPartManager** control does not render any output, so it appears as a grey box on the designer surface.

6. Position the insertion point within the `<div>` tags.

7. In the **Layout** menu, click **Insert Table**, and create a new table that has one row and three columns. Click the **Cell Properties** button, select **top** from the **Vertical align** drop-down list, click **OK**, and click **OK** again to create the table.

8. Drag a WebPartZone control into the left table column. Right-click the **WebPartZone** control, choose **Properties**, and set the following properties:

   ID: SidebarZone

   HeaderText: Sidebar

9. Drag a second **WebPartZone** control into the middle table column and set the following properties:

   ID: MainZone

   HeaderText: Main

10. Save the file.

Your page now has two distinct zones that you can control separately. However, neither zone has any content, so creating content is the next step. For this walkthrough, you work with Web Parts controls that display only static content.

The layout of a Web Parts zone is specified by a <zonetemplate> element. Inside the zone template, you can add any ASP.NET control, whether it is a custom Web Parts control, a user control, or an existing server control. Notice that here you are using the Label control, and to that you are simply adding static text. When you place a regular server control in a **WebPartZone** zone, ASP.NET treats the control as a Web Parts control at run time, which enables Web Parts features on the control.

To create content for the main zone

1. In **Design** view, drag a **Label** control from the **Standard** tab of the Toolbox into the contents area of the zone whose **ID** property is set to MainZone.

2. Switch to **Source** view. Notice that a <zonetemplate> element was added to wrap the **Label** control in the MainZone.

3. Add an attribute named **title** to the <asp:label> element, and set its value to Content. Remove the Text="Label" attribute from the <asp:label> element. Between the opening and closing tags of the <asp:label> element, add some text such as **Welcome to my Home Page** within a pair of <h2> element tags. Your code should look as follows.

```
<asp:webpartzone id="MainZone" runat="server" headertext="Main">
    <zonetemplate>
        <asp:label id="Label1" runat="server" title="Content">
            <h2>Welcome to My Home Page</h2>
        </asp:label>
    </zonetemplate>
</asp:webpartzone>
```

4. Save the file.

Next, create a user control that can also be added to the page as a Web Parts control.

To create a user control

1. Add a new Web user control to your site to serve as a search control. Deselect the option to **Place source code in a separate file**. Add it in the same directory as the WebPartsDemo.aspx page, and name it SearchUserControl.ascx.

> **NOTE**
>
> The user control for this walkthrough does not implement actual search functionality; it is used only to demonstrate Web Parts features.

2. Switch to **Design** view. From the **Standard** tab of the Toolbox, drag a TextBox control onto the page.
3. Place the insertion point after the text box you just added, and press ENTER to add a new line.
4. Drag a Button control onto the page on the new line below the text box you just added.
5. Switch to **Source** view. Ensure that the source code for the user control looks like the following example.

```
<%@ control language="C#"
    classname="SearchUserControl" %>
<asp:textbox runat="server"
  id=" TextBox1"></asp:textbox>
<br />
<asp:button runat="server"
  id=" Button1" text="Search" />
```

6. Save and close the file.

Now you can add Web Parts controls to the Sidebar zone. You are adding two controls to the Sidebar zone, one containing a list of links and another that is the user control you created in the previous procedure. The links are added as a standard **Label** server control, similar to the way you created the static text for the Main zone. However, although the individual server controls contained in the user control could be contained directly in the zone (like the label control), in this case they are not. Instead, they are part of the user control you created in the previous procedure. This demonstrates a common way to package whatever controls and extra functionality you want in a user control, and then reference that control in a zone as a Web Parts control.

At run time, the Web Parts control set wraps both controls with GenericWebPart controls. When a **GenericWebPart** control wraps a Web server control, the generic part control is the parent control, and you can access the server control through the parent control's ChildControl property. This use of generic part controls enables standard Web server controls to have the same basic behavior and attributes as Web Parts controls that derive from the **WebPart** class.

To add Web Parts controls to the sidebar zone

1. Open the WebPartsDemo.aspx page.
2. Switch to **Design** view.
3. Drag the user control page you created, SearchUserControl.ascx, from **Solution Explorer** into the zone whose **ID** property is set to SidebarZone, and drop it there.
4. Save the WebPartsDemo.aspx page.
5. Switch to **Source** view.
6. Inside the <asp:webpartzone> element for the SidebarZone, just above the reference to your user control, add an <asp:label> element with contained links, as shown in the following example. Also, add a **Title** attribute to the user control tag, with a value of **Search**, as shown.

```
<asp:WebPartZone id="SidebarZone" runat="server"
                 headertext="Sidebar">
    <zonetemplate>
        <asp:label runat="server" id="linksPart" title="My Links">
            <a href="http://www.asp.net">ASP.NET site</a>
            <br />
            <a href="http://www.gotdotnet.com">GotDotNet</a>
            <br />
            <a href="http://www.contoso.com">Contoso.com</a>
            <br />
        </asp:label>
        <uc1:SearchUserControl id="searchPart"
          runat="server" title="Search" />
    </zonetemplate>
</asp:WebPartZone>
```

7. Save and close the file.

Now you can test your page by browsing to it in your browser. The page displays the two zones. The following screen shot shows the page.

Web Parts Demo page with two zones

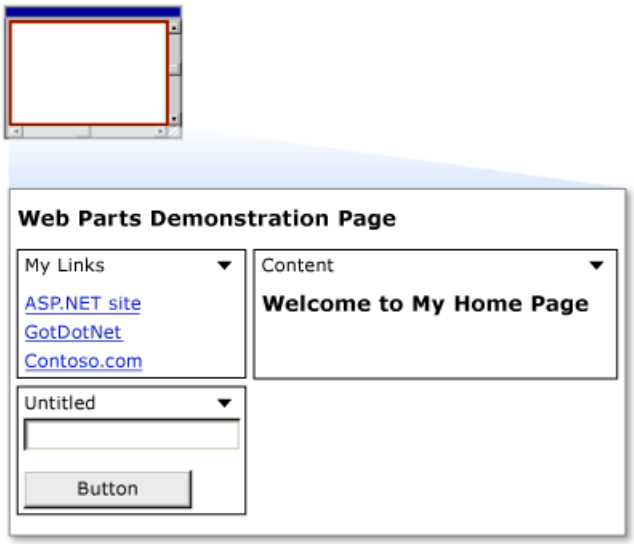


Figure 3: Web Parts VS Walkthrough 1 Screenshot

In the title bar of each control is a downward arrow that provides access to a verbs menu of available actions you can perform on a control. Click the verbs menu for one of the controls, then click the **Minimize** verb and note that the control is minimized. From the verbs menu, click **Restore**, and the control returns to its normal size.

Enabling Users to Edit Pages and Change Layout

Web Parts provides the capability for users to change the layout of Web Parts controls by dragging them from one zone to another. In addition to allowing users to move **WebPart** controls from one zone to another, you can allow users to edit various characteristics of the controls, including their appearance, layout, and behavior. The Web Parts control set provides basic editing functionality for **WebPart** controls. Although you will not do so in this walkthrough, you can also create custom editor controls that allow users to edit the features of **WebPart** controls. As with changing the location of a **WebPart** control, editing a control's properties relies on ASP.NET personalization to save the changes that users make.

In this part of the walkthrough, you add the ability for users to edit the basic characteristics of any **WebPart** control on the page. To enable these features, you add another custom user control to the page, along with an <asp:editorzone> element and two editing controls.

To create a user control that enables changing page layout

1. In Visual Studio, on the **File** menu, select the **New** submenu, and click the **File** option.
2. In the **Add New Item** dialog, select **Web User Control**. Name the new file DisplayModeMenu.ascx. Deselect the option to **Place source code in separate file**.
3. Click Add to create the new control.
4. Switch to **Source** view.
5. Remove all the existing code in the new file, and paste in the following code. This user control code uses features of the Web Parts control set that enable a page to change its view or display mode, and also enables you to change the physical appearance and layout of the page while you are in certain display modes.

```
<%@ Control Language="C#" ClassName="DisplayModeMenuCS" %>

<script runat="server">

    // Use a field to reference the current WebPartManager control.
    WebPartManager _manager;
    void Page_Init(object sender, EventArgs e) {
        Page.InitComplete += new EventHandler(InitComplete);
    }
    void InitComplete(object sender, System.EventArgs e) {
        _manager = WebPartManager.GetCurrentWebPartManager(Page);
        String browseModeName = WebPartManager.BrowseDisplayMode.Name;
        // Fill the drop-down list with the names of supported display modes.
        foreach (WebPartDisplayMode mode in
        _manager.SupportedDisplayModes) {
            String modeName = mode.Name;
            // Make sure a mode is enabled before adding it.
            if (mode.IsEnabled(_manager)) {
                ListItem item = new ListItem(modeName, modeName);
                DisplayModeDropdown.Items.Add(item);
            }
        }
        // If Shared scope is allowed for this user, display the
        // scope-switching UI and select the appropriate radio
        // button for the current user scope.
        if (_manager.Personalization.CanEnterSharedScope) {
            Panel2.Visible = true;
            if (_manager.Personalization.Scope ==
            PersonalizationScope.User)
                RadioButton1.Checked = true;
            else
                RadioButton2.Checked = true;
        }
    }

    // Change the page to the selected display mode.
    void DisplayModeDropdown_SelectedIndexChanged(object sender,
        EventArgs e) {
        String selectedMode = DisplayModeDropdown.SelectedValue;
        WebPartDisplayMode mode =
            _manager.SupportedDisplayModes[selectedMode];
        if (mode != null)
            _manager.DisplayMode = mode;
    }
    // Set the selected item equal to the current display mode.
    void Page_PreRender(object sender, EventArgs e) {
        ListItemCollection items = DisplayModeDropdown.Items;
        int selectedIndex =
        items.IndexOf(items.FindByText(_manager.DisplayMode.Name));
        DisplayModeDropdown.SelectedIndex = selectedIndex;
    }
    // Reset all of a user's personalization data for the page.
```

```
    protected void LinkButton1_Click(object sender, EventArgs e) {
        _manager.Personalization.ResetPersonalizationState();
    }
    // If not in User personalization scope, toggle into it.
    protected void RadioButton1_CheckedChanged(object sender, EventArgs e) {
        if (_manager.Personalization.Scope == PersonalizationScope.Shared)
            _manager.Personalization.ToggleScope();
    }

    // If not in Shared scope, and if user has permission, toggle
    // the scope.
    protected void RadioButton2_CheckedChanged(object sender,
    EventArgs e) {
        if (_manager.Personalization.CanEnterSharedScope &&
            _manager.Personalization.Scope == PersonalizationScope.User)
            _manager.Personalization.ToggleScope();
    }
</script>

<div>
    <asp:Panel ID="Panel1" runat="server"
      BorderWidth="1" Width="230" BackColor="lightgray"
        Font-Names="Verdana, Arial, Sans Serif">
        <asp:Label ID="Label1" runat="server"
          Text=" Display Mode" Font-Bold="true"
            Font-Size="8" Width="120" />
        <asp:DropDownList ID="DisplayModeDropdown"
          runat="server" AutoPostBack="true" Width="120"
            OnSelectedIndexChanged="DisplayModeDropdown_SelectedIndexChanged" />
        <asp:LinkButton ID="LinkButton1" runat="server"
             Text="Reset User State"
             ToolTip="Reset the current user's personalization data for the page."
             Font-Size="8" OnClick="LinkButton1_Click" />
        <asp:Panel ID="Panel2" runat="server"
            GroupingText="Personalization Scope" Font-Bold="true"
            Font-Size="8" Visible="false">
            <asp:RadioButton ID="RadioButton1" runat="server"
                Text="User" AutoPostBack="true"
                GroupName="Scope"
                OnCheckedChanged="RadioButton1_CheckedChanged" />
            <asp:RadioButton ID="RadioButton2" runat="server"
                Text="Shared" AutoPostBack="true"
                GroupName="Scope"
                OnCheckedChanged="RadioButton2_CheckedChanged" />
        </asp:Panel>
    </asp:Panel>
</div>
```

6. Save the file by clicking the save icon on the toolbar, or by selecting **Save** on the **File** menu.

To enable users to change the layout

1. Open the WebPartsDemo.aspx page, and switch to **Design** view.

2. Position the insertion point in the **Design** view just after the **WebPartManager** control that you added earlier. Add a hard return after the text so that there is a blank line after the **WebPartManager** control. Place the insertion point on the blank line.

3. Drag the user control you just created (the file is named DisplayModeMenu.ascx) into the WebPartsDemo.aspx page and drop it on the blank line.

4. Drag an EditorZone control from the **WebParts** section of the Toolbox to the remaining open table cell in the WebPartsDemo.aspx page.

5. From the **WebParts** section of the Toolbox, drag an AppearanceEditorPart control and a LayoutEditorPart control into the **EditorZone** control.

6. Switch to **Source** view. The resulting code in the table cell should look similar to the following code.

```
<td valign="top">
    <asp:EditorZone ID="EditorZone1" runat="server">
        <ZoneTemplate>
            <asp:AppearanceEditorPart ID="AppearanceEditorPart1"
              runat="server" />
            <asp:LayoutEditorPart ID="LayoutEditorPart1"
              runat="server" />
        </ZoneTemplate>
    </asp:EditorZone>
</td>
```

7. Save the WebPartsDemo.aspx file. You have created a user control that allows you to change display modes and change page layout, and you have referenced the control on the primary Web page.

You can now test the capability to edit pages and change layout.

To test layout changes

1. Load the page in a browser.
2. Click the **Display Mode** drop-down menu, and select **Edit**. The zone titles are displayed.
3. Drag the **My Links** control by its title bar from the Sidebar zone to the bottom of the Main zone. Your page should look like the following screen shot.

Web Parts Demo page with My Links control moved

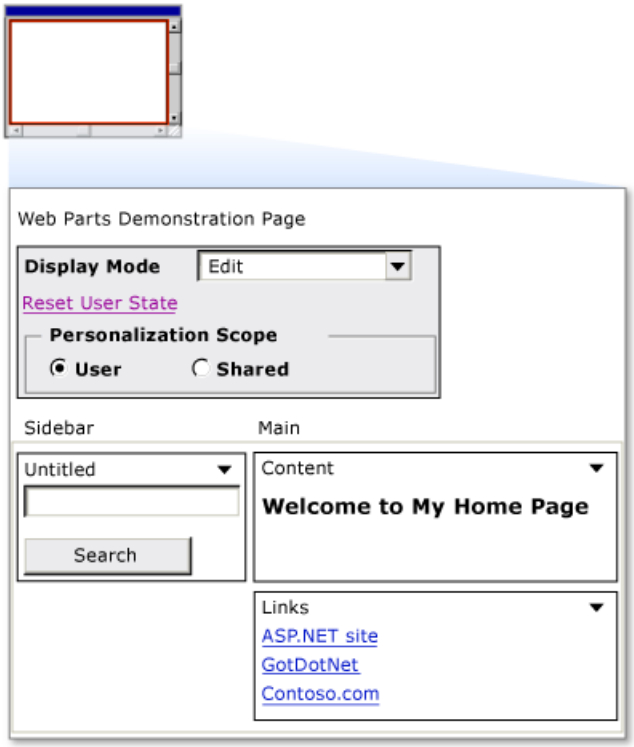


Figure 4: Web Parts VS Walkthrough 2 Screenshot

1. Click the **Display Mode** drop-down menu, and select **Browse**. The page is refreshed, the zone names disappear, and the **My Links** control remains where you positioned it.

2. To demonstrate that personalization is working, close the browser, and then load the page again. The changes you made are saved for future browser sessions.

3. From the **Display Mode** menu, select **Edit**.

   Each control on the page is now displayed with a downward arrow in its title bar, which contains the verbs drop-down menu.

4. Click the arrow to display the verbs menu on the **My Links** control. Click the **Edit** verb.

   The **EditorZone** control appears, displaying the EditorPart controls you added.

5. In the **Appearance** section of the edit control, change the **Title** to My Favorites, use the **Chrome Type** drop-down list to select **Title Only**, and then click **Apply**. The following screen shot shows the page in edit mode.

Web Parts Demo page in Edit mode

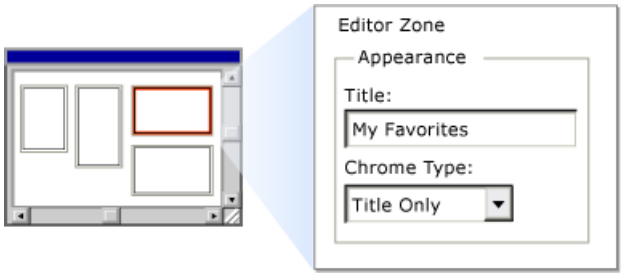


Figure 5: Web Parts VS Walkthrough 3 Screenshot

1. Click the **Display Mode** menu, and select **Browse** to return to browse mode.
2. The control now has an updated title and no border, as shown in the following screen shot.

Edited Web Parts Demo page

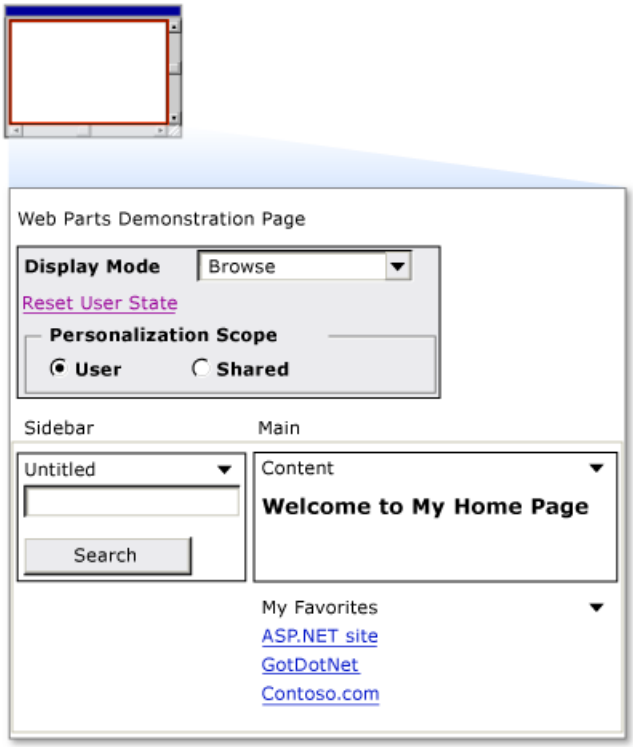


Figure 4: Web Parts VS Walkthrough 4 Screenshot

Adding Web Parts at Run Time

You can also allow users to add Web Parts controls to their page at run time. To do so, configure the page with a Web Parts catalog, which contains a list of Web Parts controls that you want to make available to users.

To allow users to add Web Parts at run time

1. Open the WebPartsDemo.aspx page, and switch to **Design** view.

2. From the **WebParts** tab of the Toolbox, drag a CatalogZone control into the right column of the table, beneath the **EditorZone** control.

Both controls can be in the same table cell because they will not be displayed at the same time.

3. In the Properties pane, assign the string **Add Web Parts** to the HeaderText property of the **CatalogZone** control.

4. From the **WebParts** section of the Toolbox, drag a DeclarativeCatalogPart control into the content area of the **CatalogZone** control.

5. Click the arrow in the upper right corner of the **DeclarativeCatalogPart** control to expose its Tasks menu, and then select **Edit Templates**.

6. From the **Standard** section of the Toolbox, drag a **FileUpload** control and a **Calendar** control into the **WebPartsTemplate** section of the **DeclarativeCatalogPart** control.

7. Switch to **Source** view. Inspect the source code of the <asp:catalogzone> element. Notice that the **DeclarativeCatalogPart** control contains a <webpartstemplate> element with the two enclosed server controls that you will be able to add to your page from the catalog.

8. Add a **Title** property to each of the controls you added to the catalog, using the string value shown for each title in the code example below. Even though the title is not a property you can normally set on these two server controls at design time, when a user adds these controls to a **WebPartZone** zone from the catalog at run time, they are each wrapped with a **GenericWebPart** control. This enables them to act as Web Parts controls, so they will be able to display titles.

   The code for the two controls contained in the **DeclarativeCatalogPart** control should look as follows.

```
<asp:DeclarativeCatalogPart ID="DeclarativeCatalogPart1" runat="server">
    <WebPartsTemplate>
        <asp:Calendar ID="Calendar1" runat="server" title="My Calendar" />
        <asp:FileUpload ID="FileUpload1" runat="server" title="Upload Files" />
    </WebPartsTemplate>
</asp:DeclarativeCatalogPart>
```

9. Save the page.

You can now test the catalog.

To test the Web Parts catalog

1. Load the page in a browser.

2. Click the **Display Mode** drop-down menu, and select **Catalog**.

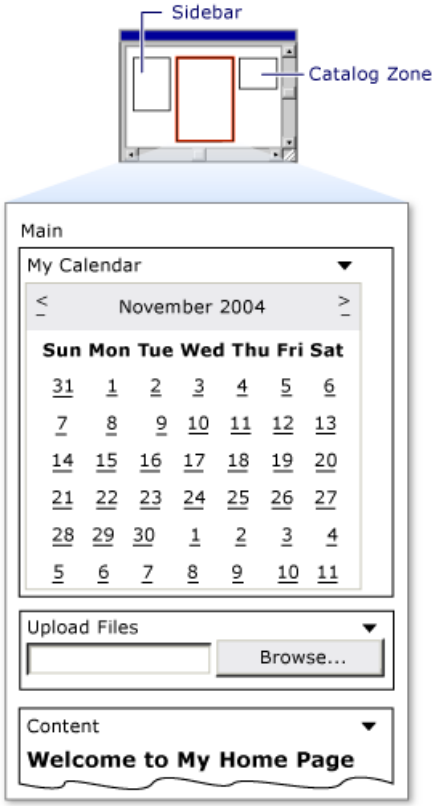
   The catalog titled **Add Web Parts** is displayed.

3. Drag the **My Favorites** control from the Main zone back to the top of the Sidebar zone, and drop it there.

4. In the **Add Web Parts** catalog, select both check boxes, and then select **Main** from the drop-down list that contains the available zones.

5. Click **Add** in the catalog. The controls are added to the Main zone. If you want, you can add multiple instances of controls from the catalog to your page.

   The following screen shot shows the page with the file upload control and the calendar in the Main zone.




```
**Figure 5**: Controls added to Main zone from the catalog
```

6. Click the **Display Mode** drop-down menu, and select **Browse**. The catalog disappears and the page is refreshed.
7. Close the browser. Load the page again. The changes you made persist.

Videos

2/20/2020 • 12 minutes to read • Edit Online

- How Do I?
  - Learn the Tips and Tricks of Experts
  - Create a Multi-Lingual Site with Localization
  - Make use of Caching
  - Make Greater use of Caching
  - Utilize Web Parts and Personalization
  - Customize my Site with Profiles and Themes
  - Secure my Site using Membership and Roles
  - Create a "Contact Us" Page
  - Create a Full-Featured Customer Login Portal
  - Use Master Pages and Site Navigation
  - Create Data-Driven Web Sites
  - Implement an HttpModule
  - Set Up the SQL Membership Provider
  - Implement URL Rewriting
  - Utilize the CSS Friendly Control Adapters
  - Handle Application Events using an HttpModule
  - Use Cascading Style Sheets for Web Page Layout
  - Create a Custom Membership Provider
  - Create a Custom Profile Provider
  - Implement Site Navigation in ASP.NET
  - Use Validation Controls in ASP.NET
  - Create a Custom User Control in ASP.NET
  - Implement Health Monitoring for an ASP.NET Application
  - Implement Tracing in an ASP.NET Web Site
  - Event Handlers in ASP.NET Web Pages and Controls
  - Implement a Custom Health Monitoring Event
  - Use Control State to Persist Information for a Custom Web Server Control
  - Read, Write, and Delete Cookies in ASP.NET
  - Save and Load View State Information for a Custom Web Server Control
  - Create and Use Application Level Events in ASP.NET
  - Simple File Uploads in ASP.NET
  - Multiple File Uploads in ASP.NET 2
  - Multiple File Uploads in ASP.NET 1
  - File Uploads with an AJAX Style Interface
  - Work with Master Pages Declaratively and Programmatically
  - Handle Events in Master and Content Pages
  - Use the ObjectDataSource with a Data Access Class and Business Object
  - Create User Selectable Themes for a Web Site
  - Share Data Between Client JavaScript and Server Code Using a Hidden Field

- Add JavaScript to ASP.NET Page
- Add JavaScript Event Handlers to ASP.NET Server Controls
- Create and Call a Simple Web Service in ASP.NET
- Deploy a Web Site Copy Web Site Tool
- Create and Use a Custom Configuration Section in an ASP.NET Configuration File
- Configure an ASP.NET Web Site for a .NET Framework Version
- Use the ASP.NET IIS Registration Tool to Configure ASP.NET
- Extend and Customize an ASP.NET Server Control for a Specific Purpose
- Use the Ability to Create Groups with the ListView Control for Different Data
- Make HTML elements accessible from server side code
- Use Roles to Segment Functionality in an ASP.NET Web Site
- Work with Nested Master Pages to Create Standard Content Layouts
- Detect Browser Capabilities in ASP.NET Web Pages
- Map an ASP.NET Server Control to the Adaptor Used to Render It
- Pass Information From One Page to Another Using a Query String
- Use the aspnet_merge.exe Utility to Merge Assemblies
- Implement a Cross Page PostBack in ASP.NET
- Add Controls to an ASP.NET Web Page Programmatically
- Use a Hidden Field to Store and Manipulate Client-Side Information
- Use the Fixednames Option with the Compiler Utility for Maximum Flexibility
- Implement Command Buttons for Individual Items
- Use Skins with CSS for a Flexible and Maintainable ASP.NET Web Site
- Determine Whether to Use a Web Site or a Web Application Project
- Write Web Events to a SQL Server Database SqlWebEventProvider
- Use ASP.NET to Send Email from a Web Site
- Configure Email Notification for Health Monitoring on an ASP.NET Web Site
- Use a Visual Studio 2008 Web Deployment Project
- Send Templated Emails for Health Monitoring Events in ASP.NET
- Send Email Asynchronously with ASP.NET
- Embed an Image in an Email with ASP.NET
- Implement Error Handling when Sending Email with ASP.NET
- Create a Custom Provider for Logging Health Monitoring Events
- Create a Reusable Component for Sending Email to a Distribution List
- Precompile an ASP.NET Website
- Use the FtpWebRequest Class to Create an FTP Connection to a FTP Server
- Export Data to a Comma Delimited (CSV) File for an Application Like Excel
- Use the Reponse.Filter Property to Replace HTML in an ASP.NET Page
- Use OutputCache to Cache an ASP.NET Page
- Cache Portions of an ASP.NET Page
- Control the Caching of an ASP.NET Page Based Upon Custom Information
- Cache an ASP.NET Page Based Upon Information in the HTTP Header
- Synchronize Data in Secondary User Control(s) Based Upon User Selections
- Use the ASP.NET Cache Object to Cache Application Information
- Persist the State of a User Control During a Postback
- Add Client Script Event Handlers Controls on an ASP.NET Page
- Understand and Defend Against Script Injection Attacks in ASP.NET

- ASP.NET Data Access Technologies, an Intro to LINQ
- Create and Use Classes in JavaScript
- Create an Efficient and Standardized Approach for Implementing CSS Styles
- What is ASP.NET
- Choosing the Right Programming Model
- Build Your First ASP.NET Application with ASP.NET Web Forms
- Build Your First ASP.NET Application with ASP.NET MVC
- Best Learn ASP.NET Web Forms
- 5 Minute Intro to ASP.NET Web Forms
- Install ASP.NET and Tools

- Tailspin Spyworks
  - Tailspin Spyworks - Creating and Popular Products Control
  - Tailspin Spyworks - Implementing and Also Purchased Control
  - Tailspin Spyworks - Intro UI and EDM
  - Tailspin Spyworks - Directory Organization
  - Tailspin Spyworks - Category Menu
  - Tailspin Spyworks - Display the Product List
  - Tailspin Spyworks - Display Per Product Details
  - Tailspin Spyworks - Adding Items to the Shopping Cart
  - Tailspin Spyworks - Display Shopping Cart
  - Tailspin Spyworks - Update the Shopping Cart
  - Tailspin Spyworks - Migrate the Shopping Cart
  - Tailspin Spyworks - Final Check Out
  - Tailspin Spyworks - Adding User Product Reviews
  - Tailspin Spyworks - Displaying User Reviews
- Building 3.5 Applications
  - Intro to Visual Web Developer
  - IntelliSense
  - Intro to Web Forms
  - Page Layout
  - Page Lifecycle Events
  - Intro to ASP.NET Controls
  - Submit and Postback
  - Application Level Objects
  - Session Level Objects
  - Debugging
  - CSS
  - MasterPages
  - Intro to SOAP Based Web Services with Visual Web Developer 2008
  - AJAX Style Services with Microsoft Visual Web Developer 2008
  - Validation
  - Login Controls
- Authentication
  - Using Basic Forms Authentication in ASP.NET
  - Change the Forms Authentication Properties

- Setup and Use Cookie-less Authentication in an ASP.NET Application
- ASP Forms Login Relocation
- Forms Login Custom Key Configuration
- Add Custom Data to the Authentication Method
- Use Custom Principal Objects
- Understanding ASP.NET Memberships
- Configuring SQL To Work with Membership Schemas
- Changing Membership Settings in the Default Membership Schema
- Creating User Accounts with the Create User Wizard
- Creating User Accounts Programmatically
- Validating Users Manually
- Validating Users with the Login Control
- Adding Users to Your Membership System
- Logging Users Into Your Membership System
- Implement the Registration Verification Pattern
- Simple Web Service Authentication
- Creating Inactive Users
- SQL Injection Defense

- SQL 2005
  - What is a Database
  - Understanding Database Tables and Records
  - More about Column Data Types and Other Properties
  - Designing Relational Database Tables
  - Manipulating Database Data
  - More Structured Query Language
  - Understanding Security and Network Connectivity
  - Connecting your Web Application to SQL Server 2005 Express Edition
  - Using SQL Server Management Studio
  - Getting Started with Reporting Services
  - Building and Customizing Reports in Business Intelligence Development Studio
  - Creating and Using Stored Procedures
  - Enabling Full-Text Search in your Text Data
- Visual Studio 2005
  - Intro to Unit Testing with Team System
  - Intro to Testing Web Applications with Team System
  - Intro to Load Testing Web Applications with Team System
  - Intro to Manual Testing with Team System
  - Intro to Managing and Running Tests with Team System
  - Measuring the Business Value of AJAX
  - Code Coverage of Automated Tests
  - Custom Extraction Rules and Coded Web Tests
  - Effects of Caching
  - Load Test Agent
  - Effects of ViewState
  - Integrate Defect Tracking with Testing
  - Create My Own Bug Work Item

- Write Code More Quickly with Unit Tests
- Practice Test-Driven Development
- Load Test a Web Application
- Tune Web Application Performance with Profiling
- Set Up Distributed Load Testing for High Volume Tests
- Enforce Coding Standards with Code Analysis
- Use Generic Tests
- Publish and Analyze Test Results
- Discover Application Changes Prior to Deployment
- Implement Continuous Integration with Team Foundation
- Automate Testing using Team Build
- Deploy a Web Application during a Team Build
- Run Unit Tests against a Deployed Database
- Enable Code Coverage and Profiling in Production Applications
- Web Deployment Projects
- Web Application Projects & Web Deployment Projects

- Migrating
  - [Intro to ASP.NET 2.0:] ASP.NET 2.0 Fundamentals
  - [Intro to ASP.NET 2.0:] User Interface Elements
  - [Migrating from] Classic ASP to ASP.NET
  - Intro to ASP.NET for JSP Developers: Welcome to ASP.NET 2.0
  - Intro to ASP.NET for JSP Developers: Building Applications
  - Intro to ASP.NET for ColdFusion Developers: Adding ASP.NET to Your Repertoire
  - Intro to ASP.NET for ColdFusion Developers: Building an ASP.NET Application
  - Interop between PHP and the Windows Platform
- Building 2.0 Applications
  - [Lesson 1:] Getting Started with Visual Web Developer Express
  - [Lesson 2:] Creating a Web Forms User Interface
  - [Lesson 3:] Understanding More About Events and Postback
  - [Lesson 4:] Understanding Web Application State
  - [Lesson 5:] Debugging and Tracing Your Website
  - [Lesson 6:] Working with Stylesheets and Master Pages
  - [Lesson 7:] Databinding to User Interface Controls
  - [Lesson 8:] Working with the GridView and FormView
  - Watch ASP.NET Development in Action
- General IIS Tips
  - Developing and Deploying In a Shared Hosting
  - Working with IIS7 Delegated Admin
  - Feature Specific Delegated Management
  - Troubleshooting Production ASP.NET Apps
  - Creating a Site with IIS7 Manager
  - Installing FTP7
  - Bit Rate Throttling
  - IIS7 Playlists
- General VS 2010 Tips

- Code Optimized Profile
- Code Search View Hierarchy
- IntelliSense Smart Lists
- Multi-Monitor Support
- New Web Project Template
- New Multi-Targeting
- Websites Instead of Web Projects
- Snippets IntelliSense

- Ajax Control Toolkit
  - Get Started with the ASP.NET AJAX Control Toolkit
  - ASP.NET AJAX CascadingDropDown Control Extender
  - ASP.NET AJAX TextBoxWatermark Control Extender
  - ASP.NET AJAX Popup Control Extender
  - ASP.NET AJAX ModalPopup Extender Control
  - ASP.NET AJAX AlwaysVisible Control Extender
  - ASP.NET AJAX Accordion Control
  - ASP.NET AJAX Collapsable Panel Extender
  - ASP.NET AJAX Draggable Panel Extender
  - ASP.NET AJAX DynamicPopulate Extender
  - ASP.NET AJAX FilteredTextbox Extender
  - ASP.NET AJAX HoverMenu Extender
  - ASP.NET AJAX ToggleButton Extender
  - ASP.NET AJAX DropShadow Extender
  - ASP.NET AJAX PasswordStrength Extender
  - Get Started with the ASP.NET AJAX Animation Extender Control
  - ASP.NET AJAX ConfirmButton Extender
  - ASP.NET AJAX Slider Control
  - ASP.NET AJAX AutoComplete Control
  - Configure the ASP.NET AJAX Calendar Control
  - ASP.NET AJAX DropDown Control
  - ASP.NET AJAX MaskedEdit Controls
  - ASP.NET AJAX MutuallyExclusive Checkbox Extender
  - ASP.NET AJAX NoBot Control
  - ASP.NET AJAX ListSearch Extender
  - PagingBulletedList Extender Control
  - NumericUpDown Extender Control
  - ASP.NET AJAX ValidatorCallout Extender
  - ASP.NET AJAX ResizableControl Extender
  - ASP.NET AJAX Tabs Control
  - ASP.NET AJAX SlideShow Extender
  - ASP.NET AJAX UpdatePanelAnimation Extender
  - AJAX Toolkit Reorder Control
  - Utilize the AJAX Rating Control in the ASP.NET Toolkit
  - Control Extenders
  - Color Picker
  - Combo Box

- Editor Control
- Editor Control Custom
- Create a New Custom Extender

- ASP.NET AJAX
  - Get Started with ASP.NET AJAX
  - Implement Dynamic Partial-Page Updates with ASP.NET AJAX
  - Make Client-Side Network Callbacks with ASP.NET AJAX
  - Add ASP.NET AJAX Features to an Existing Web Application
  - ASP.NET AJAX Enable an Existing Web Service
  - ASP.NET AJAX Client Library Controls
  - Use an ASP.NET AJAX ScriptManagerProxy
  - ASP.NET AJAX RoundedCorners Extender
  - ASP.NET AJAX Timer Control
  - Implement the Predictive Fetch Pattern for AJAX
  - Implement the AJAX Paging Pattern
  - Implement the AJAX Incremental Page Display Pattern
  - Implement the Incremental Page Display Pattern using HTTP GET and POST
  - ASP.NET AJAX UpdateProgress Control
  - ASP.NET AJAX History Control
  - Implement the AJAX After Processing Pattern
  - Update Multiple Regions of a Page with ASP.NET AJAX
  - Choose Between Methods of AJAX Page Updates
  - Use Other JavaScript User Interface Libraries with ASP.NET AJAX
  - Use the ASP.NET AJAX Profile Services
  - Debug ASP.NET AJAX Applications Using Visual Studio 2005
  - Build a Custom ASP.NET AJAX Server Control
  - Use JavaScript to Refresh an ASP.NET AJAX UpdatePanel
  - Determine Whether an Asynchronous Postback has Occurred
  - Use the Conditional UpdateMode of the UpdatePanel
  - Implement the Persistent Communications Pattern with the UpdatePanel
  - Localize an ASP.NET AJAX Application
  - Implement the Persistent Communications Pattern using Web Services
  - Trigger an UpdatePanel Refresh from a DropDownList Control
  - Create an ASP.NET AJAX Extender from Scratch
  - Build Custom Server Controls that Work With or Without ASP.NET AJAX
  - Associate AJAX Client Behavior with an ASP.NET Server Control
  - Retrieve Values From Server Side AJAX Controls
  - Two Simple Techniques for Triggering Updates to Update Panels
  - Use ASP.NET AJAX Cascading Drop Down Control to Access a Database
  - Implement Infinite Data Patterns in AJAX
  - Basic ASP.NET Authentication in an AJAX Enabled Application
  - Dynamically Change CSS ASP.NET AJAX UpdatePanel
  - Dynamically Add Controls to a Web Page
  - Set Up Your Development Environment for ASP.NET 3.5
  - Set Up Your Development Environment for ASP.NET 2.0
  - Customize Error Handling for the ASP.NET AJAX UpdatePanel

- Use ASP.NET AJAX Client Templates
- Data Access
  - ADO.NET Data Services
    - Data Access Layers in ASP.NET Applications
    - Manually Bind a Dataset to a Datagrid
    - Work with Datasets and Filters from an ASP Application
  - How Do I with LINQ
    - LINQ to SQL: Overview
    - LINQ to SQL: Data Model
    - LINQ to SQL: Querying the Database
    - LINQ to SQL: Updating the Database
    - LINQ to SQL: LinqDataSource
    - LINQ to SQL: Custom LinqDataSource
    - LINQ to SQL: Using Stored Procedures
    - LINQ to SQL: Updating with Stored Procedures
    - LINQ to SQL: Executing Arbitrary SQL
  - LINQ Videos from the VB Team
    - Get Started with LINQ
    - Perform Group and Aggregate Queries
    - Upgrade Visual Basic Projects to Enable LINQ
    - Get Started with LINQ to XML
    - Enable XML IntelliSense and Use XML Namespaces
    - Create XML Documents from SQL Data
    - Create Excel Spreadsheets using LINQ to XML
- ASP.NET Dynamic Data
  - Your First Scaffold and What is Dynamic Data
  - Enable Inline GridView Editing
  - Change how my Fields render
  - Handle Business Logic Exceptions
  - Make Custom Pages
  - Display Unknown datatypes
  - Use a DynamicControl in ListView and DetailsView Controls
  - Getting Started with Dynamic Data
  - Begin Editing the Templates in ASP.NET Dynamic Data Applications
  - Begin Modifying Dynamic Data Applications with URL Routing
  - Enable In-Line Editing in ASP.NET Dynamic Data Applications
  - Enable Table Specific Routing in Dynamic Data Applications
  - Use Attribute Validation in ASP.NET Dynamic Data Applications
  - Implement Custom Field Validation with Imperative Logic in VB or C#
  - Remove Columns From Your DynamicData Data Grids
  - Create Table Specific Custom Forms in an ASP.NET Dynamic Data Application
  - ASP.NET Dynamic Data Custom Form Formatting
- ASP.NET 3.5
  - How Do I

    - Create a Master Page in Visual Studio 2008
    - Create Nested Master Page in Visual Studio 2008
    - Cascading Style Sheets in Visual Studio 2008
    - Working with Visual Studio 2008 .NET Framework
    - Adding Elements to a CSS File and Create New CSS on the Fly
    - Advance Cascading Style Sheet Features and Management
    - Converting a .NET 2.0 Windows Forms Application to .NET 3.5
    - Get Started with the Entity Framework
    - Use the New Entity Data Source
    - Serialize a Graph with the Entity Framework
    - Use MSBuild to Automate the ASP.NET Compiler and Merge Utilities
  - ASP.NET Ajax
    - ASP.NET AJAX Support in Visual Studio 2008
    - Adding AJAX Functionality to an Existing ASP.NET Page
    - Creating and Using an AJAX-enabled Web Service in a Web Site
    - ASP.NET AJAX: A demonstration of ASP.NET AJAX
    - ADO.NET Data Services with ASP.NET AJAX Support
    - Intro to ASP.NET Ajax History
    - Using Script Combining to improve AJAX performance
  - Data Controls
    - ListView Control
    - DataPager Control
  - Visual Studio Designer
    - Visual Studio 2008 and Nested Masterpages
  - Visual Studio Editor
    - New Designer Support in Visual Studio 2008
    - JavaScript Intellisense Support in Visual Studio 2008
    - JavaScript Debugging in Visual Studio 2008
    - Multi Targeting Support in Visual Studio 2008
    - IntelliSense for Jscript and ASP.NET AJAX
    - Quick Tour of the Visual Studio 2008 Integrated Development Environment
    - Creating and Modifying a CSS File
- jQuery
  - Make Ajax Calls Using jQuery
- .NET 4
  - Ajax
    - Imperative JavaScript Syntax for Microsoft Client Side Controls
    - ScriptLoader
    - JQuery Syntax for Microsoft Ajax
    - AJAX Data Templates
  - Chart
    - Chart Control
    - Introducing the New Chart Control in Visual Studio 2010

- Core
  - Auto Start
  - Clean Web.Config Files
  - Predictable Client IDs
  - HtmlEncoder Utility Method
- Data
  - Persistent GridView Row Selection
- Markup Control
  - New Rendering Option for Check Box Lists and Radio Button Lists
  - Table Free Templated Controls
  - Tableless Menu Control
  - Hidden Field Divs
  - Disabled Control Styling
- Performance
  - Selective View State
  - Easy State Compression
  - Use the ViewStateMode Property for Managing ViewState
- Routing
  - Permanent Redirect
  - Imperative WebForms Routing
  - Declarative WebForms Routing
  - Outbound WebForms Routing
  - Use Routing with ASP.NET Web Forms
  - Work with URLs in ASP.NET Routing
- SEO
  - Dynamic Metadata

- ASP.NET 4.5
  - Strongly Typed Data Controls
  - Model Binding Part 1 - Selecting Data
  - Model Binding Part 2 - Filtering
  - Model Binding Part 3 - Updating
  - ASP.NET 4.5 Web Forms Model Binding
  - ASP.NET 4.5 Web Forms Strong Typed Data Controls
  - Bundling and Minification
  - Getting Started with the Next Version of ASP.NET
  - ASP.NET and Web Tools 2012.2

How Do I?

2/20/2020 • 3 minutes to read • Edit Online

How to perform a variety of common tasks.

- Learn the Tips and Tricks of Experts
- Create a Multi-Lingual Site with Localization
- Make use of Caching
- Make Greater use of Caching
- Utilize Web Parts and Personalization
- Customize my Site with Profiles and Themes
- Secure my Site using Membership and Roles
- Create a "Contact Us" Page
- Create a Full-Featured Customer Login Portal
- Use Master Pages and Site Navigation
- Create Data-Driven Web Sites
- Implement an HttpModule
- Set Up the SQL Membership Provider
- Implement URL Rewriting
- Utilize the CSS Friendly Control Adapters
- Handle Application Events using an HttpModule
- Use Cascading Style Sheets for Web Page Layout
- Create a Custom Membership Provider
- Create a Custom Profile Provider
- Implement Site Navigation in ASP.NET
- Use Validation Controls in ASP.NET
- Create a Custom User Control in ASP.NET
- Implement Health Monitoring for an ASP.NET Application
- Implement Tracing in an ASP.NET Web Site
- Event Handlers in ASP.NET Web Pages and Controls
- Implement a Custom Health Monitoring Event
- Use Control State to Persist Information for a Custom Web Server Control
- Read, Write, and Delete Cookies in ASP.NET
- Save and Load View State Information for a Custom Web Server Control
- Create and Use Application Level Events in ASP.NET
- Simple File Uploads in ASP.NET
- Multiple File Uploads in ASP.NET 2
- Multiple File Uploads in ASP.NET 1
- File Uploads with an AJAX Style Interface
- Work with Master Pages Declaratively and Programmatically
- Handle Events in Master and Content Pages
- Use the ObjectDataSource with a Data Access Class and Business Object
- Create User Selectable Themes for a Web Site
- Share Data Between Client JavaScript and Server Code Using a Hidden Field

- Add JavaScript to ASP.NET Page
- Add JavaScript Event Handlers to ASP.NET Server Controls
- Create and Call a Simple Web Service in ASP.NET
- Deploy a Web Site Copy Web Site Tool
- Create and Use a Custom Configuration Section in an ASP.NET Configuration File
- Configure an ASP.NET Web Site for a .NET Framework Version
- Use the ASP.NET IIS Registration Tool to Configure ASP.NET
- Extend and Customize an ASP.NET Server Control for a Specific Purpose
- Use the Ability to Create Groups with the ListView Control for Different Data
- Make HTML elements accessible from server side code
- Use Roles to Segment Functionality in an ASP.NET Web Site
- Work with Nested Master Pages to Create Standard Content Layouts
- Detect Browser Capabilities in ASP.NET Web Pages
- Map an ASP.NET Server Control to the Adaptor Used to Render It
- Pass Information From One Page to Another Using a Query String
- Use the aspnet_merge.exe Utility to Merge Assemblies
- Implement a Cross Page PostBack in ASP.NET
- Add Controls to an ASP.NET Web Page Programmatically
- Use a Hidden Field to Store and Manipulate Client-Side Information
- Use the Fixednames Option with the Compiler Utility for Maximum Flexibility
- Implement Command Buttons for Individual Items
- Use Skins with CSS for a Flexible and Maintainable ASP.NET Web Site
- Determine Whether to Use a Web Site or a Web Application Project
- Write Web Events to a SQL Server Database SqlWebEventProvider
- Use ASP.NET to Send Email from a Web Site
- Configure Email Notification for Health Monitoring on an ASP.NET Web Site
- Use a Visual Studio 2008 Web Deployment Project
- Send Templated Emails for Health Monitoring Events in ASP.NET
- Send Email Asynchronously with ASP.NET
- Embed an Image in an Email with ASP.NET
- Implement Error Handling when Sending Email with ASP.NET
- Create a Custom Provider for Logging Health Monitoring Events
- Create a Reusable Component for Sending Email to a Distribution List
- Precompile an ASP.NET Website
- Use the FtpWebRequest Class to Create an FTP Connection to a FTP Server
- Export Data to a Comma Delimited (CSV) File for an Application Like Excel
- Use the Reponse.Filter Property to Replace HTML in an ASP.NET Page
- Use OutputCache to Cache an ASP.NET Page
- Cache Portions of an ASP.NET Page
- Control the Caching of an ASP.NET Page Based Upon Custom Information
- Cache an ASP.NET Page Based Upon Information in the HTTP Header
- Synchronize Data in Secondary User Control(s) Based Upon User Selections
- Use the ASP.NET Cache Object to Cache Application Information
- Persist the State of a User Control During a Postback
- Add Client Script Event Handlers Controls on an ASP.NET Page
- Understand and Defend Against Script Injection Attacks in ASP.NET

- ASP.NET Data Access Technologies, an Intro to LINQ
- Create and Use Classes in JavaScript
- Create an Efficient and Standardized Approach for Implementing CSS Styles
- What is ASP.NET
- Choosing the Right Programming Model
- Build Your First ASP.NET Application with ASP.NET Web Forms
- Build Your First ASP.NET Application with ASP.NET MVC
- Best Learn ASP.NET Web Forms
- 5 Minute Intro to ASP.NET Web Forms
- Install ASP.NET and Tools

[How Do I:] Learn the Tips and Tricks of Experts?

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

View tips and tricks including encrypting sensitive information stored in the web.config file, posting content to a different page, simplifying complex URLs using URL remapping, and much, much more.

Watch video (18 minutes)

[How Do I:] Create a Multi-Lingual Site with Localization?

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Learn how to create culture-aware and locale-specific web content with no additional code. Use the Resource Editor to create page-level and application-level resources.

Watch video (13 minutes)

[How Do I:] Make use of Caching?

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Watch a demonstration of Page Output Caching followed by an overview of the new Database Caching support for SQL Server 2000 and SQL Server 2005.

Watch video (11 minutes)

[How Do I:] Make Greater use of Caching?

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Watch a demonstration of Partial-Page Output Caching, followed by an overview of the Cache API that demonstrates how to efficiently bind a data presentation control to an XML file.

Watch video (15 minutes)

[How Do I:] Utilize Web Parts and Personalization?

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Learn how to use the new Web Parts and Personalization features to build customizable web pages whose content and layout can be modified by the user.

Watch video (13 minutes)

[How Do I:] Customize my Site with Profiles and Themes?

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Learn how to use Profiles to collect per-user configuration information and apply Themes to extend a consistent look-and-feel throughout the Web site.

Watch video (12 minutes)

[How Do I:] Secure my Site using Membership and Roles?

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Learn how to secure a web site using the new Membership and Roles features of ASP.NET 2.0. Topics include user registration, password recovery, and restricting access to content with roles.

Watch video (17 minutes)

[How Do I:] Create a "Contact Us" Page?

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Build a web page to collect information from the user with the Wizard control, validate that information, and use the System.Net.Mail classes to send a confirmation email message.

Watch video (13 minutes)

[How Do I:] Create a Full-Featured Customer Login Portal?

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Follow along through the creation of a fictitious customer portal to learn about master pages, themes, membership, roles, site navigation, user profiles, and the Web Site Administration Tool.

Watch video (27 minutes)

[How Do I:] Use Master Pages and Site Navigation?

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Create a web site with a consistent layout and design across all its pages using ASP.NET 2.0 MasterPages. See how easy it is to add navigation to a web site using the new TreeView and SiteMapPath controls.

Watch video (13 minutes)

[How Do I:] Create Data-Driven Web Sites?

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

See how simple it is to create data-driven web sites using ASP.NET 2.0, Visual Web Developer 2005 Express Edition, and SQL Server 2005 Express Edition. Learn how to create a database, add its data, and display that data on a web page.

Watch video (13 minutes)

[How Do I:] Implement an HttpModule?

2/20/2020 • 2 minutes to read • Edit Online

by Scott Golightly

In this video Scott Golightly shows how to implement a simple HttpModule that uses the BeginRequest and EndRequest events to store some data about the request and later display that data to the user. Scott also shows how to register the module in the Web.config file.

Watch video (13 minutes)

[How Do I:] Set Up the SQL Membership Provider

2/20/2020 • 2 minutes to read • Edit Online

by Patrick Hynds

Join Patrick Hynds as he demonstrates how to set up the SQL Membership Provider to add Forms authentication to your ASP.NET 2.0 Web site. There are a few tips and tricks involved, such as using the aspnet_regsql.exe utility, which are covered in the video.

Watch video (9 minutes)

[How Do I:] Implement URL Rewriting?

2/20/2020 • 2 minutes to read • Edit Online

by Scott Golightly

In this video Scott Golightly shows how to create an ASP.NET HttpModule to "rewrite" the URL when a request for a web page comes in. You may want to rewrite URLs to create friendly URLs or to direct an old URL to a new URL. We will look at the code needed to implement URL rewriting and also how to handle page post back events.

Watch video (22 minutes)

[How Do I:] Utilize the CSS Friendly Control Adapters?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

The CSS Friendly Control Adapters allow you to alter the HTML that is rendered by a standard ASP.NET 2.0 server control. In this video we learn that the standard ASP.NET Menu control renders an HTML `<table>` element, but with the CSS Friendly Control Adapters the same control will render an HTML `<ul>` list element, improving the menu's accessibility and presentation.

Watch video (26 minutes)

[How Do I:] Handle Application Events using an HttpModule?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

Learn how to create an HttpModule that handles application events during the processing of a page request. The sample module will write information to a log file at various points in the page lifecycle. We also see how the HttpModule is configured for use in a sample web site.

Watch video (19 minutes)

[How Do I:] Use Cascading Style Sheets for Web Page Layout?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video we learn how to use `<div>` elements in combination with cascading style sheets (CSS) to create robust and higher performance web pages. The `<div>` based approach will be compared to the `<table>` based layout approach used by most web sites over recent years. Then, several CSS-based approaches will be shown, ranging from absolute positioning to floating elements. We also learn the benefits of separating structural HTML from design with the use of CSS.

Watch video (29 minutes)

[How Do I:] Create a Custom Membership Provider?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

A Membership Provider allows a web application to store and retrieve membership data for a user, and the standard ASP.NET Membership Provider uses pre-defined SQL Server tables. In this video we learn how to create a custom membership provider that uses custom SQL Server tables separate from the pre-defined tables used by the standard provider. The principles covered in the video will apply to creating custom membership providers using other databases such as Access or Oracle. After learning to create the custom membership provider, we learn how to configure the provider in a web application, and see a demonstration of creating a new user and authenticating the new user with the custom membership provider.

Watch video (29 minutes)

[How Do I:] Create a Custom Profile Provider?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video we learn how to create a custom profile provider to store and retrieve data associated with a user profile in SQL Server tables separate from the standard ASP.NET membership provider's aspnet_Profile table. The principles covered in this video will apply to creating profile providers for other databases like Access and Oracle. After learning how to create the custom profile provider we learn how to install the provider in a web site and see a demonstration of loading and saving profile data.

Watch video (32 minutes)

[How Do I:] Implement Site Navigation in ASP.NET?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video we learn how to define a sitemap and use it with the Menu, TreeView, and Sitemap server controls. We see how to use a sitemap on a master page for site-wide menus and how we can have single or multiple sitemaps on a page. In addition, we are shown how to integrate a sitemap with ASP.NET roles by using "security trimming" to dynamically show or hide menu options for users with different access levels.

Watch video (29 minutes)

[How Do I:] Use Validation Controls in ASP.NET?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video learn how to use the validation controls available in ASP.NET to implement validation on your web pages. All the major validation controls such as RequiredFieldValidator and RegularExpressionValidator will be discussed. In addition, learn how to use the CustomValidator when one of the standard validators does not meet your need. Both server and client side validation options are shown. See how to do single validation on a UI control as well as multiple validations.

Watch video (31 minutes)

[How Do I:] Create a Custom User Control in ASP.NET?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will demonstrate how to create a custom user control in ASP.NET that can be used throughout a web site or across many web sites. Start out by learning the basics of creating a user control that will be used in multiple pages including validation, controlling formatting using style elements, and loading/saving data for the user interface elements. Throughout the video considerations for building user controls that can be easily maintained and used across multiple pages are discussed.

Watch video (27 minutes)

[How Do I:] Implement Health Monitoring for an ASP.NET Application?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to implement health monitoring to get detailed run-time information for an ASP.NET application. Learn the powerful and flexible configuration options for health monitoring including health monitoring events, aka web events, providers for storing data, rules for mapping events to providers, profiles for commonly used settings, and buffermode to optimize health monitoring for best performance. Then see a sample web application in action with health monitoring enabled writing events to the Windows event log.

Watch video (21 minutes)

[How Do I:] Implement Tracing in an ASP.NET Web Site?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show you how to implement tracing in an ASP.NET web site to monitor or measure the performance and to diagnose errors. Learn how to add tracing to an entire web site as well as individual pages. See how to add custom trace messages in pages and business objects to aid in assessing site performance and in diagnosing errors. Learn the differences between tracing in an ASP.NET page and in using System.Diagnostics.Trace for tracing information in business objects. Writing trace information to log files and viewing trace information in the trace viewer are also discussed.

Watch video (30 minutes)

[How Do I:] Event Handlers in ASP.NET Web Pages and Controls

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to create event handlers for ASP.NET web pages and web server controls. First, learn how to create page level events for the standard page events provided by the .NET Framework. See how to automatically bind the events to a page using the AutoEventWireup attribute as well as programmatically bind the events. Next, learn how to create and bind event handlers for the standard web server control events. Also learn how to connect multiple events for different controls to the same event handler. Finally, see the usage of the AutoPostbackTrue property for web server controls.

Watch video (32 minutes)

[How Do I:] Implement a Custom Health Monitoring Event

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to create a custom health monitoring event that extends a standard health monitoring events in ASP.NET. The custom provides additional event information and formats it for a health monitoring event provider. A custom HTTP module is then created to raise the custom health monitoring event explicitly since the .NET Framework only raises the standard health monitoring events. In addition, the custom health monitoring event is raised explicitly in the page behind code for an .aspx page.

Watch video (26 minutes)

[How Do I:] Use Control State to Persist Information for a Custom Web Server Control

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to store information related to a custom web server control in the control state which is independent of the ViewState information for the control. Unlike ViewState the control state can not be turned off and is therefore a more reliable way to store important information for a control. A discussion of what types of information are appropriate to store in control state is also included.

Watch video (13 minutes)

Read, Write, and Delete Cookies in ASP.NET

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to read, write, and delete cookies in an ASP.NET application. First, learn the basic operations for manipulating cookies. Then learn approaches to storing multiple values in a single cookie using name-value pairs called subkeys. Next, learn how to limit the scope of cookies to a particular application domain or folder. See how to modify values and expiration dates in existing cookies and how to delete an existing cookie. In addition, several considerations for using cookies are also discussed.

Watch video (25 minutes)

[How Do I:] Save and Load View State Information for a Custom Web Server Control

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to persist and restore information in the ViewState of a custom user control. First, learn how to persist simple values containing .NET primitive types such as String, Int32, Boolean, etc. Then see how to persist complex types such as Class which contain properties storing information. The sequence of processing View State information during the lifecycle of an ASP.NET page is also discussed.

Watch video (28 minutes)

[How Do I:] Create and Use Application Level Events in ASP.NET

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to create an Application Class File (Global.asax) that contains application level event handlers such as Application_Start and Application_BeginRequest. Then the sequence of application level events in the application lifecycle will be discussed as a basis for determining where certain types of operations would be best performed during the application lifecycle.

Watch video (12 minutes)

[How Do I:] Simple File Uploads in ASP.NET

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video I will demonstrate, using ASP.NET, uploading a single file from the client machine to your server. This will be the first in a series of uploading files videos.

Watch video (17 minutes)

[How Do I:] Multiple File Uploads in ASP.NET 2

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video I will demonstrate uploading a dynamically determined number of files to your ASP.NET application.

Watch video (16 minutes)

[How Do I:] Multiple File Uploads in ASP.NET 1

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video I will elaborate on uploading multiple files to an ASP.NET application.

Watch video (15 minutes)

[How Do I:] File Uploads with an AJAX Style Interface

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video I will demonstrate, using ASP.NET, an AJAX style of uploading files from the client machine to your server. I say AJAX style because there is actually a problem using AJAX to upload files. The video will demonstrate what I mean.

Watch video (27 minutes)

[How Do I:] Work with Master Pages Declaratively and Programmatically

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show several aspects of defining and using master pages in an ASP.NET web site. First, see how to create master pages declaratively using the ASP.NET master page model and learn what information is best suited to a master page compared to an individual content page. Next, see how to programmatically assign master pages on-the-fly in an individual content page. Also learn how to create a base page class which assigns a master page programmatically for all pages that inherit from the base class. In addition, see how to set the master page in the web.config file. Lastly, see how to access elements in a master page from an individual content page including some best practices for structuring access to master page elements.

Watch video (29 minutes)

[How Do I:] Handle Events in Master and Content Pages

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how events are handled in master and individual content pages. Even though the end result of a master and individual content page is a single ASP.NET page rendered in the user's browser the event handling in each type of page is handled separately. If a control is located in the master page the corresponding event handler is located in the master page by default. The same is true for controls in the content page. See two approaches to having the events for a control located in the master page handled in the code behind of the content page. This allows "communication" between the master page and content page.

Watch video (23 minutes)

[How Do I:] Use the ObjectDataSource with a Data Access Class and Business Object

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to use the ObjectDataSource as the intermediate layer between data bound user interface elements and a custom data access class for a business object. Uses of the ObjectDataSource in the GUI Design mode and declaratively in the source view of an .aspx page are discussed. In addition, the considerations for design of the data access class and business object for use with the ObjectDataSource are covered.

Watch video (32 minutes)

[How Do I:] Create User Selectable Themes for a Web Site

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to create multiple themes for a web site and then allow the user to select which theme is applied to the site. See how to create a simple user interface that allows the user to select each of the themes with a user control. In addition, learn how to set a theme for a content page from a user control on a master page.

Watch video (19 minutes)

[How Do I:] Share Data Between Client JavaScript and Server Code Using a Hidden Field

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to share data in between JavaScript and server-side code by using a hidden field in an ASP.NET page. Next, learn how to use the RegisterHiddenField method in server-side code to programmatically create a hidden field on the client. In addition, see how to programmatically create a HiddenField server control for sharing data on an ASP.NET page. Finally, use the RegisterArrayDeclaration method to create a container in server code for storing multiple values that is rendered to the page where it can be accessed by JavaScript.

Watch video (21 minutes)

[How Do I:] Add JavaScript to An ASP.NET Page

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to include JavaScript to an ASP.NET page for use on the client page to provide a richer and more robust user experience. First, see how to include JavaScript directly within an ASP.NET page and reference the script from within HTML page elements. Next, see how to abstract the JavaScript to an external JavaScript file which offers the advantage of being cached by the browser and can be used by multiple pages. Lastly, learn how to add JavaScript to a page dynamically at runtime which has the advantage of including script for a specific situation or where information such as a control ID is not known until runtime. As part of the dynamic script sample see how to add JavaScript that can be automatically executed when the page loads as well as the click event for button controls (Button, LinkButton, and ImageButton).

Watch video (24 minutes)

[How Do I:] Add JavaScript Event Handlers to ASP.NET Server Controls

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to interface JavaScript methods with ASP.NET server controls to provide client side functionality for the server controls. First, see how to add client event handlers declaratively by extending ASP.NET server controls with custom attributes. Next, see how to add attributes for mapping client methods to ASP.NET server controls programmatically which offers the advantage of being done at runtime. Finally, learn how to reference the ID of a server control, which is rendered dynamically by ASP.NET, in JavaScript code.

Watch video (18 minutes)

[How Do I:] Create and Call a Simple Web Service in ASP.NET

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to create a web service in ASP.NET and then call the service from another ASP.NET web site. First, learn how to create an .asmx component which represents a web service in an ASP.NET web site. When creating the web service learn how to control several aspects of the web service functionality with web service attributes. Next, see how to test the web service in the Visual Studio web service test page. Then learn how ASP.NET allows many .NET data types to be returned from a web service. Create a business class library project and then use a class in that project as a return value from a web service by serializing the custom business object instance. Finally, see how to consume a web service from an ASP.NET web site.

Watch video (33 minutes)

[How Do I:] Deploy a Web Site Using the Copy Web Site Tool

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to deploy a web site using the Copy Web Site Tool in Visual Studio. First see how to connect to a remote web site and deploy an entire site to the target location. Next, learn how to synchronize an entire source and target web site. In addition, see how to copy and synchronize individual or groups of files including how the Copy Web Site tool provides status indicators of differences between files. As part of this process see how the tool provides options on a file-by-file basis for decision making when there are differences in status between two or more files.

Watch video (17 minutes)

[How Do I:] Create and Use a Custom Configuration Section in an ASP.NET Configuration File

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to extend the standard set of ASP.NET configuration settings with your own XML configuration elements. First, see how to create a configuration section handler which is required to interpret and process the information in custom XML configuration elements. During this learn how to use declarative model for defining the behavior of a configuration section handler. Next, see how to add a custom section handler and custom elements to an ASP.NET configuration file. Finally, see how to programmatically access the custom configuration data.

Watch video (19 minutes)

[How Do I:] Configure an ASP.NET Web Site for a .NET Framework Version

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to configure an ASP.NET web site to use a specific version of ASP.NET/.NET Framework. First see how to identify what version of the .NET Framework a web site is associated with by viewing the script map in Internet Information Server (IIS). See how a script map is associated with the root in IIS and then with individual web sites. Next learn about the ASP.NET IIS Registration tool (aspnet_regiis.exe) used to configure IIS or an ASP.NET web site to use a specific version of the .NET Framework. Using this information see how a single web site is configured to use a different version of the .NET Framework than is the default to IIS. Finally, see two different web sites under the same IIS where one is running under .NET Framework 2.0 and the other under 1.1.

Watch video (12 minutes)

How Do I Use the ASP.NET IIS Registration Tool to Configure ASP.NET

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to use the ASP.NET IIS Registration Tool to configure ASP.NET. First, see how file extensions are mapped to ASP.NET in the IIS Manager on a global and web-site-by-web site basis. These script maps provide the basis for configuring individual ASP.NET applications to run under a specific version of the .NET Framework. Next, see how aspnet_regiis.exe, the IIS Registration Tool command line utility, is specific to each version of the .NET Framework and how it can be used for a variety of tasks like determining what versions of ASP.NET are installed on a computer and what ASP.NET applications are mapped to that version. Then, learn how install and uninstall ASP.NET on a computer using the registration tool. Also learn how to use the registration tool to install, uninstall, or update multiple ASP.NET applications for a specific version of the .NET Framework. For example, see how to install ASP.NET 2.0 and update script ma

Watch video (27 minutes)

[How Do I:] Extend and Customize an ASP.NET Server Control for a Specific Purpose

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to extend a standard ASP.NET server control and customize it for a specific purpose. Specialized controls provide an convenient way to implement standardized user interface elements for multiple web sites for individuals or teams of developers. In this example, see how to extend the DropDownList control to create a special purpose year selection control. Learn how to add properties for custom attributes that control the behavior of the range of years that can be displayed. Next, see how those custom attributes can be set in declarative syntax like standard attributes of a control. See how several additional properties can then be added to provide additional functionality for controlling the list behavior. Finally, see how an extended ASP.NET server control can be moved to a separate assembly so that it can be used across multiple web sites.

Watch video (26 minutes)

[How Do I:] Use the Ability to Create Groups with the ListView Control for Different Data

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to group items when displaying data in a ListView control. First, see the basics of item layout in the ListView control with the LayoutTemplate and the ItemTemplate. Then, learn how the GroupTemplate relates to the LayoutTemplate and ItemTemplate within a ListView to establish a group of data. See how to use floating DIV elements to create a flexible tabular layout since any of the templates used with the ListView can contain any HTML. This provides significant flexibility in creating groupings of data using the ListView control. Finally, see some options for enhancing the look by manipulating the CSS styles used in the ListView templates and how to use the GroupSeparatorTemplate as well as the EmptyItemTemplate.

Watch video (23 minutes)

[How Do I:] Make HTML elements accessible from server side code

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to make standard HTML elements in an .aspx page accessible from server side code so they can be used in page processing like a standard ASP.NET server control. First, see how the addition of a runat="server" attribute to a standard HTML control like a allows it to be referenced in server side code with full Intellisense. Next, see several examples of how you might manipulate HTML elements on a page with this technique. Lastly, see how this technique can be used in conjunction with the new ListView control in .NET Framework Version 3.5 to control the layout of a list.

Watch video (21 minutes)

[How Do I:] Use Roles to Segment Functionality in an ASP.NET Web Site

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to maximize the use of roles in an ASP.NET web site after setting up ASP.NET Membership. First, learn how to setup roles using the built-in role manager in Visual Studio .NET. See how to define roles based upon the business needs associated with the web site. Once the roles are setup learn how to structure your web site from a folder and page hierarchy perspective to most efficiently accommodate the roles and business needs. As part of this see how to use the roles attribute in defining authorization in the web.config file associated with each folder. Also, see how to use the <location> element to apply authorization settings to a specific location such as a page.

Watch video (32 minutes)

[How Do I:] Work with Nested Master Pages to Create Standard Content Layouts

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to use nested master pages to create individual master pages that represent different standard content layouts for a web site. First, see how several major commercial web sites use a standard set of content layouts. Next, see how to nest a master page within another master page, and use the design time support in Visual Studio 2008. Then, learn the considerations for establishing a "page architecture" which represents the major types of content layout used in a sample web site. Once that definition is complete see how to structure the nested master pages so developers can then select a master page, resulting in a standardized and consistent display of content for a web site.

Watch video (30 minutes)

[How Do I:] Detect Browser Capabilities in ASP.NET Web Pages

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to determine what capabilities a user's browser has when viewing pages in an ASP.NET web site. First, learn how to access an instance of the HttpBrowserCapabilities object from the ASP.NET Request object associated with the web page request. Next, see the different information available in the HttpBrowserCapabilities object such as the type of browser and whether the user's browser supports JavaScript and Cookies. Then see an example of how to user this information about the user's browser and determine if a crawler is accessing the web site so it can be redirected to a crawler specific web page.

Watch video (14 minutes)

[How Do I:] Map an ASP.NET Server Control to the Adaptor Used to Render It

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to use a control adaptor to provide different renderings for an ASP.NET server control without actually changing the control itself. In this video, an ASP.NET BulletList control will be adapted to display each list item horizontally using DIV elements instead of the traditional UL elements. First, see how to create a class that inherits WebControlAdaptor and then implements the code to render the new list format. Next, learn how to map the new control adaptor to the ASP.NET server control in the .browser definition file. Then see how to use the new control adaptor on pages in a web site. Finally, learn how a control adaptor can be associated with either all browsers or specific types of browsers.

Watch video (23 minutes)

[How Do I:] Pass Information From One Page to Another Using a Query String

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how information can be passed from one page to another using a query string. First, see how to construct a query string including variable names and values from information on a data entry form. Next, see how to access the query string information when the page is posted including how multiple query string variables are handled. As part of the server side processing of query string information, see how to implement validation for existence of query string variables and if they contain the expected type of data. Finally, learn how to pass special characters including a space and an ampersand in a query string by encoding the string.

Watch video (25 minutes)

[How Do I:] Use the aspnet_merge.exe Utility to Merge Assemblies

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to use the aspnet_merge.exe utility to combine assemblies that have been compiled using the aspnet_compiler.exe utility. First, view the output of the ASP.NET compilation process to learn what assemblies are created. Next, learn the mechanics of using the aspnet_merge.exe utility to merge assemblies from the compilation of a web site, first seeing how to control the compiled assembly names with the –prefix option. Then see how to merge all the assemblies for a compiled web site into a single assembly. Lastly, learn how to merge just the assemblies related to the content (aspx, ascs, etc.) into a single assembly, leaving the top level assemblies such as app_code.dll representing the /App_Code folder. During each option the advantages and disadvantages of that option are discussed.

Watch video (28 minutes)

[How Do I:] Implement a Cross Page PostBack in ASP.NET

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to create a Cross Page PostBack in ASP.NET, where a control on a source page raises a PostBack event to a different target page. First, learn how to configure a Button control to PostBack to a different target page. Then, learn how to get information from controls on the source page with the PreviousPage property and the FindControl method. Then, learn how to use public properties on the source page to expose data to the target page, and learn how to use the @PreviousPageType directive on the target page to access these properties. Finally, learn the difference between a Cross Page PostBack and the Server.Transfer method.

Watch video (29 minutes)

[How Do I:] Add Controls to an ASP.NET Web Page Programmatically

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to create an ASP.NET server control programmatically at runtime and add it to a page. First, learn the basic concept of adding a control to a page programmatically. Then, the concept of a "container" control is examined which is necessary for adding a new control programmatically at runtime. Next, learn considerations for nesting programmatically added controls, and how creating separate methods can compartmentalize the process to make it more maintainable. In addition, learn considerations for managing view state when adding controls dynamically.

Watch video (33 minutes)

[How Do I:] Use a Hidden Field to Store and Manipulate Client-Side Information

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to use a hidden field as one approach to simple state management on an ASP.NET page and for communicating client side processing information. First, see how to add a hidden field to a page using an ASP.NET HiddenField server control and retrieve the value of the hidden field in server-side code after a post-back. Next, learn how to manipulate the value of the hidden field in client-side script, and then retrieve the modified value in server-side code. Finally, see an example of how to use several hidden fields to store information from client-side processing that can be posted back for server processing.

Watch video (24 minutes)

[How Do I:] Use the Fixednames Option with the Compiler Utility for Maximum Flexibility

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to use the fixednames option with the aspnet_compiler.exe utility to produce a consistent file names in compilation output. First, see how the compilation output of the aspnet_compiler.exe utility generates output files for an ASP.NET web site that represent the different components including pages, themes, and the folders in the site. See how the components are named and what they contain in relation to the original web site. Then, see how to use the fixednames option to standardize the output file names based upon the source files upon which they are based. Next, learn about considerations for using each of the compilation options when deploying updates to compiled web site, focusing on which components are most likely to be updated.

Watch video (23 minutes)

[How Do I:] Implement Command Buttons for Individual Items

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will show how to handle events associated with a button control in each item of a data-bound control like a ListView control. First, see how these data-bound controls have standard button event handlers for standard operations like Edit or Delete. Next, learn how to add a button which performs a custom action and how to handle events for this custom button action. Finally, see how to pass data to the custom event handler for identification of the item where the button click originated.

Watch video (34 minutes)

[How Do I:] Use Skins with CSS for a Flexible and Maintainable ASP.NET Web Site

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to create skins for ASP.NET server controls and then incorporate CSS styles to control the appearance of the skinned controls. First, learn how to create a skin for an ASP.NET server control such as a Label that controls the appearance of labels throughout a web site. Next, see how to abstract the definition of the visual appearance for a skin to CSS styles. During this process learn the advantages of having the styling information for a skin in the CSS styles. Then see how to create multiple skins and corresponding styles for a single ASP.NET control such as a Label that can be used to standardize the appearance of certain types of labels throughout a web site. Finally, learn the advantages of applying a CSS style to multiple skins and the ability to apply it to a generic HTML element.

Watch video (29 minutes)

[How Do I:] Determine Whether to Use a Web Site or a Web Application Project

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels will contrast the differences between an ASP.NET web site project and a web application project. Learn the considerations for using one type of project or the other. First, learn about the major attributes of a web application project including its folder basis. Next, see the major attributes of a web site project and its traditional Visual Studio project based nature which provides more granularity and flexibility including adding classes, dealing with namespaces, and referencing user controls or pages programmatically. The compilation model for each type of web site is then discussed along with alternatives that include the use of the aspnet_compiler.exe and aspnet_merge.exe utilities and the Visual Studio 2008 web deployment project.

Watch video (29 minutes)

[How Do I:] Write Web Events to a SQL Server Database Using the SqlWebEventProvider

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to use the ASP.NET health monitoring SqlWebEventProvider to log errors in a web site to a SQL Server database. First, learn the role of the provider and events in ASP.NET health monitoring. Next, see how to configure a SQL Server database with the necessary objects for recording health monitoring events using the aspnet_regiis utility, the same utility used to configure ASP.NET Membership. Then, learn how to configure health monitoring in the web.config file to record events in an ASP.NET web site to the newly created SQL Server database. As part of this configuration, see how the root web.config file in the .NET Framework 2.0 has both providers and events defined that can be leveraged when configuring your health monitoring. These fundamentals are a base upon which custom events can be created that log your own specific information to a SQL Server database in an ASP.NET web site.

Watch video (31 minutes)

[How Do I:] Use ASP.NET to Send Email from a Web Site

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to create and send an email in an ASP.NET web site. First, learn about the core classes in the System.Net.Mail namespace for sending mail in an ASP.NET web site. See how to create an email message using the MailMessage class including how to use the MailAddress class for the From address and how to include one or more To addresses. Next, see how email is sent using the SmtpClient class. Finally, learn about the additional options for configuring the email message such as CC, BCC, Priority, and adding attachments.

Watch video (25 minutes)

[How Do I:] Configure Email Notification for Health Monitoring on an ASP.NET Web Site

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to configure email notification for health monitoring in an ASP.NET web site. First, see how to configure the sending of email in an ASP.NET web site through the use of the <emailSettings> element in the web.config file. Next, learn how to add the SimpleMailWebEventProvider which sends emails for health monitoring events as a provider. Then see the standard health monitoring events that can be used with email notification by examining the machine level health monitoring configuration. After reviewing the available events see a rule implemented that maps the "All Events" to the email provider. Upon starting the web site several emails are then sent and examined when they are received in Outlook. Finally, some basic principles of which events might be mapped to the health monitoring email provider are discussed.

Watch video (25 minutes)

[How Do I:] Use a Visual Studio 2008 Web Deployment Project

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to use the Visual Studio 2008 Web Deployment add-on to create deployment projects with a finer degree of control over how a web site is compiled and deployed. First, learn where to download and install the Web Deployment Project add-on for Visual Studio 2008. Then, create a web site project and an associated web deployment project. Learn the available configuration options for a web deployment project. See a brief review of the aspnet_compiler and aspnet_merge command line utilities which are integral to the web deployment project. Next, learn how to create multiple configurations for a web deployment project, e.g., one for Debug and one for Release modes. Then create a web application project and a corresponding web deployment project. Finally, see how the options and process for the web application project are the same as those for the web site project.

Watch video (26 minutes)

[How Do I:] Send Templated Emails for Health Monitoring Events in ASP.NET

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to use the TemplatedEmailWebEventProvider to send emails when health monitoring events occur that utilize a template for the email content. First, see how to configure the <provider> and <rules> elements in the web.config file to implement the use of templated email and associate a health monitoring event with the templated email provider. Once the templated provider is configured see how to create the email template using as standard .aspx page. Learn what information is available in the MailEventNotificaitonInfo class that is passed by the TemplatedEmailWebEventProvider to the template .aspx page. See how it can be used to include whatever information is appropriate in the email content. Finally, view the test web site which sends emails in response to health monitoring events. Then view the actual emails received that contain the health monitoring event information based upon the template.

Watch video (25 minutes)

[How Do I:] Send Email Asynchronously with ASP.NET

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video, Chris Pels shows how to use the System.Net.Mail classes in ASP.NET to send an asynchronous email message. First, see how to configure a web site to send email using the <mailSettings> element in the web.config file. Next, create a simple user interface for entering email information. Then learn how to create use the MailMessage class to create an email message in the code behind for the page. As part of that process create an event handler for the asynchronous callback following the sending of the email. In the event handler see how to use the instance of the AsynchCompletedEventArgs class which provides information about the email sending process. Finally, send a test email asynchronously, following the steps in the debug mode, and view the actual email received from the process.

Watch video (18 minutes)

[How Do I:] Embed an Image in an Email with ASP.NET

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

Chris Pels shows how to embed an image in an email with ASP.NET. He creates a web form (with fields for To, From, Subject, and Body), uses the AlternateView class to create text and HTML versions of an email, stores an image in a LinkedResource class instance, embeds it in the HTML AlternateView. He then adds both versions to the MailMessage object and sends the email twice, first with HTML-receiving capabilities enabled and then as text-only. Both email messages appear in Outlook. The HTML version shows the embedded image; the text version does not.

Watch video (19 minutes)

[How Do I:] Implement Error Handling when Sending Email with ASP.NET

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

Chris Pels shows how to implement error handling when sending an email with ASP.NET. He creates an ASP.NET web page to send email, shows how to configure <mailSettings> in the web.config file, describes the System.Net.Mail class and how it's used to create and send email messages. He then adds error handling using System.Net.Mail exception classes, which provide information about errors that can occur when sending email, and reviews the SmtpStatusCode enumeration, which provides a list of possible outcomes when sending an email with the SmtpClient. Finally, he sends a test email that raises an exception and reviews the error handling information in the Visual Studio debugger.

Watch video (24 minutes)

[How Do I:] Create a Custom Provider for Logging Health Monitoring Events

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

Chris Pels shows how to create a custom provider for the ASP.NET heath monitoring system, which records health monitoring events to a text-based log file. Learn how to create the custom provider class, which inherits from the BufferedWebEventProvider class, understand the methods and properties used to create the provider, and create a web site that uses it to write log files.

Watch video (26 minutes)

[How Do I:] Create a Reusable Component for Sending Email to a Distribution List

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to create a component that can be used on multiple web pages and web sites that sends emails to a list of recipients. First, the ASP.NET web site will be configured to send email using the <mailSettings> in the web.config file. Then a class and several methods are created to read a list of recipients from a data source (DB, XML, etc.) and send an email message to each of the recipients using the System.Net.Mail classes. As part of this process exception handling is included. In addition, a user interface is created to allow the user to specify items such as the From address, Subject, add an attachment, etc.

Watch video (35 minutes)

[How Do I:] Precompile an ASP.NET Website

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

Chris Pels shows how to precompile an ASP.NET website, which increases first-run performance and strengthens application security.

Watch video (21 minutes)

[How Do I:] Use the FtpWebRequest Class to Create an FTP Connection to a FTP Server

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to use the FtpWebRequest class to create an FTP connection and transfer a file. First, create a simple .aspx form to select a file to transfer to the FTP server. Next, see the essential methods and properties of the FtpWebRequest class that need to be configured in order to establish an FTP connection. Then, see how to get the local file contents so the file can be transferred using a StreamReader and a StreamWriter to write it to the FtpWebRequest. Finally, a FtpWebResponse is obtained to determine the outcome of the transfer. This approach can be used to programmatically transfer files for a web application.

Watch video (21 minutes)

[How Do I:] Export Data to a Comma Delimited (CSV) File for an Application Like Excel

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to take data from a database or other source and export it to a comma delimited file that can be used in an application like Excel. First, a set of data is created as a DataTable object. Next, the Response for the current web page request is cleared and the header and content type are configured to be a csv file. Then the actual data is added to the response stream by first writing the column headers for the csv file followed by the data values. This approach can be useful when users require an export of data so it can be manipulated locally in a program like Excel.

Watch video (19 minutes)

[How Do I:] Use the Reponse.Filter Property to Replace HTML in an ASP.NET Page

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to use the Reponse.Filter property to intercept and alter the HTML being sent to a page. First, a sample page is created with some simple text. Then, a custom Stream class is created which serves as the replacement stream for the current stream being sent to the user's browser. In that custom stream class the contents of the page are retrieved from the stream, altered, and then written out to the response stream. In this custom Stream class the Write method is customized to replace the HTML in the base Response stream, thereby altering what is sent to the user's browser. Finally, the new stream class is assigned to the Response.Filter property in the Page_Load event, thereby, providing the mechanism for altering the page content.

Watch video (13 minutes)

[How Do I:] Use OutputCache to Cache an ASP.NET Page

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to utilize the built-in caching of ASP.NET to store a page in the cache so it does not have to be re-rendered. First, the simplest form of page caching is demonstrated where the entire page is cached for a specified period of time. Next, a more granular approach is taken where the values of QueryString variables are used as the basis for caching pages. Finally, see how to have the type of browser be the basis for caching a page.

Watch video (14 minutes)

[How Do I:] Cache Portions of an ASP.NET Page

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to cache one or more portions of an ASP.NET page by implementing the OutputCache for user controls on the page. First, an ASP.NET page is created and cached using the OutputCache directive. Several user controls are then created and cached for different durations. The relationship between page cached duration and control cache duration is reviewed. The different criteria for caching controls are reviewed including an example of VaryByControl where nested user controls can control caching of their parent controls.

Watch video (19 minutes)

[How Do I:] Control the Caching of an ASP.NET Page Based Upon Custom Information

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to control the criteria for caching an ASP.NET page based upon custom information. A sample page is created and then the OutputCache directive is used with the VaryByCustom attribute which contains a custom value. Next, the GetVaryCustomByString() method is overridden in the global.asax module which provides the handling of the custom attribute. In that method a string is returned that uniquely identifies the cached version of the page. Finally, there is a discussion about how caching using a custom value can be used in several ways for a web site.

Watch video (12 minutes)

[How Do I:] Cache an ASP.NET Page Based Upon Information in the HTTP Header

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to keep a page in the ASP.NET output cache based upon information in the page's HTTP header. First, the potential HTTP header values are reviewed. Then, a sample page is created and then the OutputCache directive is used with the VaryByHeader attribute which contains a value "accept-language", an HTTP header, to control caching based upon the language of the user's browser. The sample page is viewed in IE which is set to English and then in FireFox which is set to use French. Finally, the option to move the cache definition to a CacheProfile in the web.config file is discussed.

Watch video (12 minutes)

[How Do I]: Synchronize Data in Secondary User Control(s) Based Upon User Selections

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to create two DropDownList controls where selecting an item in the first list populates the items in the second list. First, an ASP.NET page is created with two DropDownList controls, one for customer and the other for states in which the customer does business. The customer list is then populated in the code behind. Next, in the SelectedIndexChanged event of the customer list the items in the state list are populated based upon the selection. Finally, there is a discussion of how this approach can be applied to other types of controls and data.

Watch video (22 minutes)

[How Do I]: Use the ASP.NET Cache Object to Cache Application Information

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to use the ASP.NET Cache object to store application information. A sample page is created with a simple user interface and code is added that populates the grid with data from an XML disk file. Then the code is revised to cache the data in the application Cache object after it is initially retrieved and read from the cache for subsequent requests. Finally, methods are added to make use of absolute and sliding time based dependency for the cached object.

Watch video (22 minutes)

[How Do I]: Persist the State of a User Control During a Postback

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to persist the state of one or more objects in a user control. First, a user control is created that represents the ability for a user to specify filter criteria for a search. In addition, a companion Filter class is created to store the filter information. Several user interface elements are added to the filter control along with some methods and properties to store the current filter information in the Filter class instance. Next, the user control persistence is implemented using the RegisterRequiresControlState method and associated Save/Restore methods. These methods store the instance of the filter class and its data during page postbacks. Finally, there is a discussion of how to store multiple objects in control state implementation.

Watch video (23 minutes)

How Do I: Add Client Script Event Handlers Controls on an ASP.NET Page?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to add client script events to controls on an ASP.NET page. First, several ASP.NET controls are added to the page and an event handler is added for standard server-side processing of the Click event for a button. Next, client side JavaScript event handlers are added for Mouseover and Mouseout client side events that call JavaScript functions on the page to change a button background color. Next, key presses are captured for a textbox by adding a client side event handler. Then the OnClientClick attribute is shown as the way to handle client site click events for a button in addition to server side processing. Finally, several client side event handlers are defined by adding them programmatically in the server Page_Load event.

Watch video (12 minutes)

How Do I Understand and Defend Against Script Injection Attacks in ASP.NET

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Cross Site Scripting (XSS) is perhaps the most troublesome security issue for web developers to tackle these days. In this video, Microsoft's Joe Stagner provides some examples of how dangerous XSS can really be and how ASP.NET helps you easily defend against XSS in your Web Applications.

Watch video (40 minutes)

ASP.NET Data Access Technologies, an Introduction to LINQ

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

LINQ is the ASP.NET developers preferred Data Access Mechanism. This video is an introduction to programming with LINQ and compares simple data access between ADO.NET and LINQ from a developer experience perspective.

Watch video (37 minutes)

How Do I: Create and Use Classes in JavaScript?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to create classes in JavaScript with functionality similar to other programming languages with object oriented capabilities. First a sample web site project is created with an external JavaScript file. A function() is created and the difference between using the function as a "function" and as a class constructor with the "new" keyword is discussed. Then public properties are added to the class constructor as well as private variables. Next a separate function is created and assigned as a method to the class. Alternatively, the same method is shown as a function defined in the class constructor. Finally, a private class method is created and used within the class constructor.

Watch video (18 minutes)

How Do I: Create an Efficient and Standardized Approach for Implementing CSS Styles?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to organize and implement CSS styles that provide a standardized look and feel to a web site. In addition, the styles are organized to provide maximum flexibility and reusability. A set of styles is created that focuses on the visual appearance of the site. First, a sample web site is created with a CSS file for a sample page. The first groups of styles created are oriented towards providing containers for content throughout the site. Next, a set of styles are added that focus on displaying the "state" of user interface elements. Finally, a set of styles for user interaction cues are created. Each of the styles is used with HTML elements on the sample page. There is also a discussion about using the same implementation approach for styles that control element layout on the site.

Watch video (27 minutes)

What is ASP.NET?

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Microsoft's Scott Hanselman gives a brief overview of ASP.NET.

Watch video (3 minutes)

Choosing the Right Programming Model

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Microsoft's Scott Hanselman gives you the information you need to decide if ASP.NET Web Forms, ASP.NET MVC, or ASP.NET Web Pages is right for you.

Watch video (5 minutes)

Build Your First ASP.NET Application with ASP.NET Web Forms

2/20/2020 • 2 minutes to read • Edit Online

by Laurence Moroney

Microsoft's Laurence Moroney shows you how to build your first ASP.NET application with ASP.NET Web Forms.

Watch video (11 minutes)

Build Your First ASP.NET Application with ASP.NET MVC

2/20/2020 • 2 minutes to read • Edit Online

by Laurence Moroney

Microsoft's Laurence Moroney shows you how to build your first ASP.NET application with ASP.NET MVC.

Watch video (18 minutes)

How to Best Learn ASP.NET Web Forms

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Microsoft's Joe Stagner shows you what you need to learn how to use ASP.NET Web Forms to develop your web applications.

Watch video (3 minutes)

Intro to ASP.NET Web Forms

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Scott Hanselman presents a 5 minute introduction to ASP.NET Web Forms.

Watch video (5 minutes)

How to Install ASP.NET and Tools

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Microsoft's Scott Hanselman shows you how to download and install ASP.NET and tools so you can start developing your web applications.

Watch video (3 minutes)

Tailspin Spyworks

2/20/2020 • 2 minutes to read • Edit Online

How to use new features in ASP.NET 4 to build an online store, including shopping, checkout, and administration.

- Tailspin Spyworks - Creating and Popular Products Control
- Tailspin Spyworks - Implementing and Also Purchased Control
- Tailspin Spyworks - Intro UI and EDM
- Tailspin Spyworks - Directory Organization
- Tailspin Spyworks - Category Menu
- Tailspin Spyworks - Display the Product List
- Tailspin Spyworks - Display Per Product Details
- Tailspin Spyworks - Adding Items to the Shopping Cart
- Tailspin Spyworks - Display Shopping Cart
- Tailspin Spyworks - Update the Shopping Cart
- Tailspin Spyworks - Migrate the Shopping Cart
- Tailspin Spyworks - Final Check Out
- Tailspin Spyworks - Adding User Product Reviews
- Tailspin Spyworks - Displaying User Reviews

Tailspin Spyworks - Creating and Using the Popular Products Control

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner creates and uses the Popular Products control in the Tailspin Spyworks sample Web Forms application.

Watch video (23 minutes)

NEXT

Tailspin Spyworks - Implementing and Using the Also Purchased Control

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner implements and uses the Also Purchased control in the Tailspin Spyworks sample Web Forms applicaton.

Watch video (30 minutes)

PREVIOUS NEXT

Tailspin Spyworks - Intro UI and EDM

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner kicks off the Tailspin Spyworks sample Web Forms application by implementing the default template.

Watch video (12 minutes)

PREVIOUS NEXT

Tailspin Spyworks - Directory Organization

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner modifies the default directory structure for the Tailspin Spyworks sample Web Forms application.

Watch video (4 minutes)

PREVIOUS NEXT

Tailspin Spyworks - Category Menu

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner builds the category menu for the Tailspin Spyworks sample Web Forms application.

Watch video (7 minutes)

PREVIOUS NEXT

Tailspin Spyworks - Display the Product List

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner shows how to display the product list in the Tailspin Spyworks sample Web Forms application.

Watch video (9 minutes)

PREVIOUS NEXT

Tailspin Spyworks - Display Per Product Details

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner displays the product details in the Tailspin Spyworks sample Web Forms application.

Watch video (6 minutes)

PREVIOUS NEXT

Tailspin Spyworks - Adding Items to the Shopping Cart

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner builds the Add to Cart function for the Tailspin Spyworks sample Web Forms application.

Watch video (24 minutes)

PREVIOUS NEXT

Tailspin Spyworks - Display Shopping Cart

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner shows how to display the shopping cart in the Tailspin Spyworks sample Web Forms application.

Watch video (15 minutes)

PREVIOUS NEXT

Tailspin Spyworks - Update the Shopping Cart

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner updates the shopping cart for the Tailspin Spyworks sample Web Forms application.

Watch video (41 minutes)

PREVIOUS NEXT

Tailspin Spyworks - Migrate the Shopping Cart

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner shows how to convert the shopping cart of an anonymous user to an authenticated one before final check out for the Tailspin Spyworks sample Web Forms application.

Watch video (23 minutes)

PREVIOUS NEXT

Tailspin Spyworks - Final Check Out

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner places the customer's order during the final check out process for the Tailspin Spyworks sample Web Forms application.

Watch video (37 minutes)

PREVIOUS NEXT

Tailspin Spyworks - Adding User Product Reviews

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner adds reviews from customers to product details page in the Tailspin Spyworks sample Web Forms application.

Watch video (41 minutes)

PREVIOUS NEXT

Tailspin Spyworks - Displaying User Reviews

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner displays reviews from customers in the Tailspin Spyworks sample Web Forms application.

Watch video (13 minutes)

PREVIOUS

Building 3.5 Applications

2/20/2020 • 2 minutes to read • Edit Online

The basics of creating ASP.NET Web Forms applications.

- Intro to Visual Web Developer
- IntelliSense
- Intro to Web Forms
- Page Layout
- Page Lifecycle Events
- Intro to ASP.NET Controls
- Submit and Postback
- Application Level Objects
- Session Level Objects
- Debugging
- CSS
- MasterPages
- Intro to SOAP Based Web Services with Visual Web Developer 2008
- AJAX Style Services with Microsoft Visual Web Developer 2008
- Validation
- Login Controls

Intro to Visual Web Developer

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Microsoft Visual Web Developer is the FREE version of Visual Studio for developing ASP.NET applications. This video will show how to get and install it and take you through a general tour of the IDE and its features.

Watch video (39 minutes)

IntelliSense

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

IntelliSense is a valuable Visual Studio feature for both learning .NET programming as well as accelerating your coding efficiency. This video will introduce you to the value of IntelliSense in Visual Web Developer.

Watch video (14 minutes)

Intro to Web Forms

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

ASP.NET Web Forms is the default User Interface mechanism for ASP.NET applications.. This video will demonstrate the basic of developing with Web Forms.

Watch video (23 minutes)

Page Layout

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Tables or StyleSheets? Relative Positioning or Absolute? This video will demonstrate the features of Visual Web developer that support the developing with your page layout preference.

Watch video (8 minutes)

Page Lifecycle Events

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

ASP.NET provides a very rich eventing mechanism with hooks into all the important chronological instances in the ASP.NET page lifecycle. This video will enumerate many of the common lifecycle events, their firing sequence and the kinds of things that you might use each event for.

Watch video (12 minutes)

Intro to ASP.NET Controls

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Controls driven development can really increase your development productivity. This video provides a basic introduction to the standard ASP.NET controls.

Watch video (8 minutes)

Submit and Postback

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video explains what goes on "under the covers" when your ASP.NET application's browser and server interact.

Watch video (19 minutes)

Application Level Objects

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video will introduce Application Level Events and Objects and explain the concept of "Scope" as it pertains to them.

Watch video (13 minutes)

Session Level Objects

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video will introduce Session Level Events and Objects and compare their use with their Application and Page level equivalents.

Watch video (10 minutes)

Debugging

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Description: Visual Web Developer offers a rich debugging experience for beginners and expert developer alike. In this video Joe will tour you through the VWD debugging experience.

Watch video (37 minutes)

CSS

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Cascading Style Sheets are integral to modern web development and Visual Web Developer 2008 includes a collection of powerful CSS building facilities to help. In this video learn about VWD's CSS Editor, Builder, Manager, and IntelliSense.

Watch video (24 minutes)

MasterPages

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

ASP.NET MasterPages are a key part of building agile web applications. This video will demonstrate the basics of using master pages and nested MasterPages in your ASP.NET applications.

Watch video (16 minutes)

An Introduction to SOAP Based Web Services with Visual Web Developer 2008

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video shows you how quick and easy it can be to start developing distributed applications based on SOAP / XML web services.

Watch video (26 minutes)

AJAX Style Services with Microsoft Visual Web Developer 2008

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

AJAX means different things to different people. This video shows you the basics of writing AJAX style services (and client side consumers) in Visual Web Developer 2008.

Watch video (16 minutes)

Validation

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video Joe will demonstrate the use of the FormValidation controls in ASP.NET Using Microsoft Visual Web Developer.

Watch video (16 minutes)

Login Controls

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video Joe will introduce you to ASP.NET Membership by showing you how to use some of the Login Controls in ASP.NET.

Watch video (10 minutes)

Authentication

2/20/2020 • 2 minutes to read • Edit Online

How to implement and manage security (authentication) in an ASP.NET application.

- Using Basic Forms Authentication in ASP.NET
- Change the Forms Authentication Properties
- Setup and Use Cookie-less Authentication in an ASP.NET Application
- ASP Forms Login Relocation
- Forms Login Custom Key Configuration
- Add Custom Data to the Authentication Method
- Use Custom Principal Objects
- Understanding ASP.NET Memberships
- Configuring SQL To Work with Membership Schemas
- Changing Membership Settings in the Default Membership Schema
- Creating User Accounts with the Create User Wizard
- Creating User Accounts Programmatically
- Validating Users Manually
- Validating Users with the Login Control
- Adding Users to Your Membership System
- Logging Users Into Your Membership System
- Implement the Registration Verification Pattern
- Simple Web Service Authentication
- Creating Inactive Users
- SQL Injection Defense

Using Basic Forms Authentication in ASP.NET

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner walks through the process to utilize basic forms based authentication programmatically with ASP.NET. It is suggested that viewers read this before viewing. For more information this demo relates to the tutorial located here.

Watch video (23 minutes)

NEXT

How to Change the Forms Authentication Properties

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner utilizes the previous code example to demonstrate how to change the authentication period for the ASP.NET application. For more information on this demo, a tutorial is located here.

Watch video (7 minutes)

PREVIOUS NEXT

How to Setup and Use Cookie-less Authentication in an ASP.NET Application

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner utilizes the previous code example to demonstrate how to use cookie-less authentication. For more information on this demo, a tutorial is located here.

Watch video (4 minutes)

PREVIOUS NEXT

ASP Forms Login Relocation

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner uses the previous code example to continue customizing the ASP.NET authentication by relocating and renaming the login functionality of the ASP.NET application. For more information on this demo, a tutorial is located here.

Watch video (4 minutes)

PREVIOUS NEXT

Forms Login Custom Key Configuration

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner will examine the machine key setting in your ASP.NET application. For more information on this demo, a tutorial is located here.

Watch video (7 minutes)

PREVIOUS NEXT

Add Custom Data to the Authentication Method

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner will continue the exploration of ASP.NET authentication by configuring the authentication method to add custom data to the authentication ticket that is used by ASP.NET. For more information on this demo, a tutorial is located here.

Watch video (14 minutes)

PREVIOUS NEXT

Use Custom Principal Objects

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner demonstrates how to implement custom identity in principal objects in order to store user authentication data. For more information on this demo, a tutorial is located here.

Watch video (22 minutes)

PREVIOUS NEXT

Understanding ASP.NET Memberships

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video Joe Stagner looks at working with the built-in providers, available to ASP.NET developers, for membership. For additional information about this video, read the Creating the Membership Schema in SQL Server security tutorial.

Watch video (22 minutes)

PREVIOUS NEXT

Configuring SQL To Work with Membership Schemas

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video Joe Stagner demonstrates how to configure a SQL instance to work with a specific ASP Services Membership Profile; as well as configure the application to specifically use this SQL instance and not the default settings in the machine.cgf file. For additional information about this video, read the Creating the Membership Schema in SQL Server security tutorial.

Watch video (18 minutes)

PREVIOUS NEXT

Changing Membership Settings in the Default Membership Schema

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video Joe Stagner looks at how to change the default settings of the membership schema to conform to the specific needs of an custom application. For additional information about this video, read the Creating the Membership Schema in SQL Server security tutorial.

Watch video (9 minutes)

PREVIOUS NEXT

Creating User Accounts with the Create User Wizard

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video, Joe Stagner shows how to create users with the Create User Wizard that comes with the Login Control Collection in ASP.NET and Visual Studio. For additional information about this video, read the "Creating User Accounts" security tutorial (C#, VB).

Watch video (9 minutes)

PREVIOUS NEXT

Creating User Accounts Programmatically

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video, Joe Stagner demonstrates how to create a custom create user web form programmatically while not using the Create User wizard. For additional information about this video, read the "Creating User Accounts" security tutorial (C#, VB).

Watch video (19 minutes)

PREVIOUS NEXT

Validating Users Manually

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video Joe Stagner shows you how to create your own web form and process the authentication request manually. For additional information about this video, read the "Validating User Credentials Against the Membership User Store" security tutorial (C#, VB).

Watch video (13 minutes)

PREVIOUS NEXT

Validating Users with the Login Control

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video, Joe Stagner explores an alternative method of user authentication rather then creating a custom form and calling the API to authenticate a user. He uses the Login Control that ships with ASP.NET 2 and above. For additional information about this video, read the "Validating User Credentials Against the Membership User Store" security tutorial (C#, VB).

Watch video (5 minutes)

PREVIOUS NEXT

Adding Users to Your Membership System

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video, Joe Stagner shows you how to add new users to the membership repository.

Watch video (47 minutes)

PREVIOUS NEXT

Logging Users Into Your Membership System

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video, Joe Stagner shows how to log our users into the ASP.NET membership system.

Watch video (19 minutes)

PREVIOUS NEXT

Implement the Registration Verification Pattern

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video, Joe Stagner extends the registration process in an ASP.NET application by adding an email verification step.

Watch video (35 minutes)

PREVIOUS NEXT

Simple Web Service Authentication

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video, Joe Stagner demonstrates different mechanisms that can be added to an ASP.NET application to restrict unauthorized access to ASMX web services.

Watch video (54 minutes)

PREVIOUS NEXT

Creating Inactive Users

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video Joe Stagner walks through creating inactive users in the ASP.NET membership system, and then using an administrative interface to activate the users one at a time.

Watch video (23 minutes)

PREVIOUS NEXT

SQL Injection Defense

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

A SQL Injection vulnerability in your application can ruin your whole day. In this video, Microsoft's Joe Stagner explains how SQL Injection attacks can happen, what a bad guy can do with them, and how to protect your ASP.NET application from SQL Injection vulnerabilities.

Watch video (25 minutes)

PREVIOUS

SQL 2005

2/20/2020 • 2 minutes to read • Edit Online

How to work with SQL Server Express to add database capabilities to your ASP.NET applications.

- What is a Database
- Understanding Database Tables and Records
- More about Column Data Types and Other Properties
- Designing Relational Database Tables
- Manipulating Database Data
- More Structured Query Language
- Understanding Security and Network Connectivity
- Connecting your Web Application to SQL Server 2005 Express Edition
- Using SQL Server Management Studio
- Getting Started with Reporting Services
- Building and Customizing Reports in Business Intelligence Development Studio
- Creating and Using Stored Procedures
- Enabling Full-Text Search in your Text Data

What is a Database?

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

This introduction covers the questions and terminology SQL Server beginners need to create and manage databases, such as: What is a SQL Server database? How is a SQL Server database different from an Excel spreadsheet? What do SQL Server databases do in my website or software applications? What's SQL Server Express' relation to Visual Studio Express? What do terms like "ADO.NET" mean?

Watch video (28 minutes)

NEXT

Understanding Database Tables and Records

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

SQL Server beginners will learn about tables and definitions of data types, properties, keys, etc. in this second video. Find out how table rows, columns, and fields interrelate and whether columns can be empty.

Watch video (24 minutes)

PREVIOUS NEXT

More about Column Data Types and Other Properties

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

SQL Server beginners will explore data integrity, learn which data types to use for what, and avoid common data type pitfalls.

Watch video (21 minutes)

PREVIOUS NEXT

Designing Relational Database Tables

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Why are relational databases the most popular? SQL Server beginners will learn to differentiate between primary and foreign keys, design relational database tables, and learn about normalization rules and de-normalization.

Watch video (34 minutes)

PREVIOUS NEXT

Manipulating Database Data

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

SQL Server beginners will learn about ADO.NET objects and execution of SQL statement from ADO.NET. Explore SQL and learn about insert, update, and delete statements.

Watch video (40 minutes)

PREVIOUS NEXT

More Structured Query Language

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

SQL Server beginners will discover more SQL statements like sub-select and learn stored procedures and their counterparts in C# and VB methods.

Watch video (23 minutes)

PREVIOUS NEXT

Understanding Security and Network Connectivity

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

SQL Server beginners will learn to differentiate between SQL Server database files and SQL Server processes. Explore scenarios for connecting to SQL Server Express databases, enabling and disabling user instances, login permissions, and other security precautions.

Watch video (40 minutes)

PREVIOUS NEXT

Connecting your Web Application to SQL Server 2005 Express Edition

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

SQL Server beginners will walkthrough the entire workflow of creating a database driven Web application and use visual tools to quickly create an administration application.

Watch video (65 minutes)

PREVIOUS NEXT

Using SQL Server Management Studio

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

SQL Server Management Studio Express will typically be your main interface into SQL Server Express. This lesson demonstrates the essential techniques and skills you'll need to successfully connect to database instances, navigate the query window and scripting database objects, attaching, detaching and backing up databases, and much more.

Watch video (40 minutes)

PREVIOUS NEXT

Getting Started with Reporting Services

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

SQL Server Reporting Services allows you to design and deploy nicely formatted and interactive reports representative of your data. Suitable for print or online distribution, these reports and it allows you to get professional results very quickly. This video demonstrates how to set up Reporting Services and the Business Intelligence Development Studio to build reports using the wizards and designers, and deploy them to your Report Server.

Watch video (32 minutes)

PREVIOUS NEXT

Building and Customizing Reports in Business Intelligence Development Studio

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Building on the previous lesson, we dive into building reports from scratch and modifying the reports that were created using the wizards and designers. We also demonstrate how to embed your reports into your ASP.NET Web Form applications.

Watch video (44 minutes)

PREVIOUS NEXT

Creating and Using Stored Procedures

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

This video expands on the topic of Stored Procedures which was introduced several lessons ago. In this video we explain the basic syntax to create and update stored procedures, using input and output parameters, return values, declaring variables and adding other T-SQL programming elements into you stored procedures, how to execute your stored procedures from the query window, and finally the Visual Basic and C# syntax required to utilize stored procedures in your own applications.

Watch video (42 minutes)

PREVIOUS NEXT

Enabling Full-Text Search in your Text Data

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Full-Text search allows you to save large portions of text - even Microsoft Word documents, or other file formats - into your database and perform complex queries based on this textual data while expecting great performance. This lesson demonstrates how to set up Full-Text search paying particular attention to the nuances of setting it up within SQL Server Express. Then we move to creating full-text indexes, configuring the index population settings, and performing queries that demonstrate the power of the CONTAINS predicate and its many, many variations.

Watch video (38 minutes)

PREVIOUS

Visual Studio 2005

2/20/2020 • 2 minutes to read • Edit Online

How to work with Visual Studio 2005, Team System, and add-ins to perform testing, deployment, and other tasks.

- Intro to Unit Testing with Team System
- Intro to Testing Web Applications with Team System
- Intro to Load Testing Web Applications with Team System
- Intro to Manual Testing with Team System
- Intro to Managing and Running Tests with Team System
- Measuring the Business Value of AJAX
- Code Coverage of Automated Tests
- Custom Extraction Rules and Coded Web Tests
- Effects of Caching
- Load Test Agent
- Effects of ViewState
- Integrate Defect Tracking with Testing
- Create My Own Bug Work Item
- Write Code More Quickly with Unit Tests
- Practice Test-Driven Development
- Load Test a Web Application
- Tune Web Application Performance with Profiling
- Set Up Distributed Load Testing for High Volume Tests
- Enforce Coding Standards with Code Analysis
- Use Generic Tests
- Publish and Analyze Test Results
- Discover Application Changes Prior to Deployment
- Implement Continuous Integration with Team Foundation
- Automate Testing using Team Build
- Deploy a Web Application during a Team Build
- Run Unit Tests against a Deployed Database
- Enable Code Coverage and Profiling in Production Applications
- Web Deployment Projects
- Web Application Projects & Web Deployment Projects

Introduction to Unit Testing with Team System

2/20/2020 • 2 minutes to read • Edit Online

by Eric Lee

In this video we see how the unit testing features of Visual Studio Team System integrate unit testing directly into the Visual Studio IDE.

Watch video (10 minutes)

NEXT

Introduction to Testing Web Applications with Team System

2/20/2020 • 2 minutes to read • Edit Online

by Eric Lee

In this video we see how you can automate testing of your web applications using the web testing features of Visual Studio Team System.

Watch video (9 minutes)

PREVIOUS NEXT

Introduction to Load Testing Web Applications with Team System

2/20/2020 • 2 minutes to read • Edit Online

by Eric Lee

In this video we see how you can load test your web applications using the load testing features of Visual Studio Team System.

Watch video (9 minutes)

PREVIOUS NEXT

Introduction to Manual Testing with Team System

2/20/2020 • 2 minutes to read • Edit Online

by Eric Lee

In this video we see how you can create and run your manual test cases for your applications by using the manual testing features of Visual Studio Team System.

Watch video (8 minutes)

PREVIOUS NEXT

Introduction to Managing and Running Tests with Team System

2/20/2020 • 2 minutes to read • Edit Online

by Eric Lee

In this video we see how you can manage all or your various test cases using the test management capabilities of Visual Studio Team System. We will also see how tests are run and configured.

Watch video (8 minutes)

PREVIOUS NEXT

Measuring the Business Value of AJAX

2/20/2020 • 2 minutes to read • Edit Online

by Eric Lee

ASP.NET AJAX is well known for its ability to improve the user experience of Web applications. With Visual Studio Team System you can also measure the business value of adding ASP.NET AJAX to your Web applications. Specifically you can use the Load and Web Testing capabilities of Visual Studio Team System to explore how the difference in HTTP data transmission size affects the performance and scalability characteristics of the Web application.

Watch video (10 minutes)

PREVIOUS NEXT

Code Coverage of Automated Tests

2/20/2020 • 2 minutes to read • Edit Online

by Eric Lee

This video demonstrates how to use code coverage as a metric for determining how effective our testing is of an ASP.NET Web application. After a test has completed, Visual Studio Team System reports on the percentage of code that was executed and indicates executed code with a green background and unexecuted code with a red background, making it easy to see the code that was covered by the test. You will also learn how you can use C# or Visual Basic to extend the automated test.

Watch video (6 minutes)

PREVIOUS NEXT

Custom Extraction Rules and Coded Web Tests

2/20/2020 • 2 minutes to read • Edit Online

by Eric Lee

Visual Studio Team System makes it easy to test a Web application by recording a user's interaction with the application and then repeatedly replaying the recorded interaction. But users do not typically perform the same actions again and again. In this video we see how to extend and customize our automated Web tests to more accurately reflect the way a user would interact with the application. We learn how to create custom extraction rules to pull data from a Web control that the default extraction rules may not support, and how to add C# or Visual Basic-coded Web tests to add functionality such as looping, logic, and accessing external data.

Watch video (12 minutes)

PREVIOUS NEXT

The Effects of Caching

2/20/2020 • 2 minutes to read • Edit Online

by Eric Lee

Nearly every Web application today needs to access data from a database, but this common operation can have a real impact on the overall performance of the application. This video illustrates how Visual Studio Team System can help you identify whether your application is making redundant calls to the database, and then demonstrates how you can employ ASP.NET 2.0 DataSource caching functionality to eliminate the redundant calls.

Watch video (15 minutes)

PREVIOUS NEXT

Using the Load Test Agent

2/20/2020 • 2 minutes to read • Edit Online

by Eric Lee

In this video we examine how to incorporate the Load Test Agent in to the testing that we do with Visual Studio Team System. A Load Test Agent is part of a "rig" that can be deployed to separate machines to enable load, stress, and unit testing from another operating system or machine configuration. Each rig can work with several agents and can be deployed to virtual or remote machines, allowing for comprehensive testing with Visual Studio Team System.

Watch video (5 minutes)

PREVIOUS NEXT

The Effects of ViewState

2/20/2020 • 2 minutes to read • Edit Online

by Eric Lee

The ASP.NET framework provides the ViewState mechanism to automatically maintain application state by carrying data forward and backward between browser requests. In this video we see how some ASP.NET controls such as the DropDownList and TextBox maintain state based on postback form values rather than ViewState, allowing the smart developer to disable unnecessary ViewState. Using Visual Studio Team System we can see how this dramatically reduces the size of request and response data, and improves the performance and scalability of the application.

Watch video (10 minutes)

PREVIOUS NEXT

[How Do I:] Integrate Defect Tracking with Testing?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Menegay

In this video we take a look at the integration points between the Testing Framework of Visual Studio Team System and the Work Item Tracking System.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Create My Own Bug Work Item?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Menegay

This video demonstrates how to create your own Bug Work Item definition for use in Visual Studio Team System. We see how a Work Item is defined by an XML file that we can manually edit in order to add, remove or modify the fields, workflows, and forms associated with a particular type of Work Item.

Watch video (11 minutes)

PREVIOUS NEXT

[How Do I:] Write Code More Quickly with Unit Tests?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Menegay

In this video we learn how Unit Testing can increase your productivity by easing the debugging process. We see how Visual Studio Team System can generate a Unit Test to automate execution of code that is being debugged, which saves you the time it takes to fire up your Web application and perform any steps needed to manually execute the same code. The result of Unit Testing is that you can write more robust code, more quickly.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Practice Test-Driven Development?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Menegay

Visual Studio Team System provides the tools necessary to practice Test-Driven Development, and this video we learn how Team System's testing functionality allows you to go from code to test and from test to code. We see how you may write a test and then use Visual Studio Team System to generate the target code, which adheres to the TDD ideology of writing the test first and the code second.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Load Test a Web Application?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Menegay

This video walks you through the process of recording a Web Test and then replaying it as a Load Test that simulates multiple users. We see how Visual Studio Team System allows you to set warning and critical thresholds of server resources utilization, which defines whether the Web application passes or fails the load test.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Tune Web Application Performance with Profiling?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Menegay

In this video we learn that Visual Studio Team System provides two methods of profiling a Web application: Sampling and Instrumentation. Instrumentation injects markers in the underlying code so that the developer can view full details of the application's execution, including call trees and object allocation, to help tune the performance of the application.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Set Up Distributed Load Testing for High Volume Tests?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Menegay

Visual Studio Team System allows you to set up distibuted load testing by creating a test "rig" that comprises a Controller on one remote machine and multiple Agents on the same or additional remote machines. This video walks you through installing the Controller and the Agents, and then configurating your test cases to run using the remote Controller. As you will see, Visual Studio Team System makes it very simple to set up distributed load testing for high-volume automated testing.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Enforce Coding Standards with Code Analysis?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Menegay

With Visual Studio Team System you have fine control over the code analysis rules that apply to class library projects, web site projects, and source code control. In this video we learn how to specify the errors and warnings that will apply during project builds and code check ins, and we see how to suppress a particular rule from applying to a specific class member. With this configurability you can enforce coding standards to improve the quality of your code, while ensuring inapplicable rules never get in your way.

Watch video (6 minutes)

PREVIOUS NEXT

[How Do I:] Use Generic Tests?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Menegay

A generic test is an existing program that is wrapped and made available as a test in Visual Studio Team System. In this video we see a demonstration of how a wrapped shell program accepts an input value from our test environment and how the program's output value is used to determine whether the test passed or failed.

Watch video (5 minutes)

PREVIOUS NEXT

[How Do I:] Publish and Analyze Test Results?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Menegay

Visual Studio Team System enables you to chart changes in a project's code quality as the project progresses. This video provides instructions on how to publish test results to the Team Foundation Server project database and associate the results with a build number. We see that after the results are published we can use the Report Manager to view a graphical representation of changes in our build test results.

Watch video (7 minutes)

PREVIOUS NEXT

[How Do I:] Discover Application Changes Prior to Deployment?

2/20/2020 • 2 minutes to read • Edit Online

by Paul Hacker

Learn how to use the Source Control Explorer from Visual Studio Team System to discover the changes that have been made to your Web application, and to ensure that only the necessary changes get deployed to the live Web site. We also look at a free utility that allows you to merge changes together to simplify deployment.

Watch video (5 minutes)

PREVIOUS NEXT

[How Do I:] Implement Continuous Integration with Team Foundation?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Menegay

There are a number of continuous integration tools that work on top of Team Foundation Server, and in this video we look at the free Norton TeamCI tool available from Team System Rocks. TeamCI periodically checks whether updated files have been checked in to Team Foundation Server and, if so, will build the solution and run any automated tests.

Watch video (6 minutes)

PREVIOUS NEXT

[How Do I:] Automate Testing using Team Build?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Menegay

In this video we learn how to use Visual Studio Team System to define the automated tests that will run during a team build, and how we can drill down in to the results of those tests. We also learn that the Test Tools Build Task provides an alternative way of specifying the tests to run during a team build.

Watch video (6 minutes)

PREVIOUS NEXT

[How Do I:] Deploy a Web Application during a Team Build?

2/20/2020 • 2 minutes to read • Edit Online

by Paul Hacker

This video demonstrates how to create a new team build and how to deploy your Web application as part of the build process. We see that including the deployment step is as simple as adding an X-copy command to the build script.

Watch video (5 minutes)

PREVIOUS NEXT

[How Do I:] Run Unit Tests against a Deployed Database?

2/20/2020 • 2 minutes to read • Edit Online

by Paul Hacker

If your ASP.NET application is configured to deploy a database along with the Web site, how can you verify that a valid database instance has been deployed? In this video we learn how to run unit tests against the deployed database, asserting the existence of a value that is known to be consistent across database instances.

Watch video (3 minutes)

PREVIOUS NEXT

[How Do I:] Enable Code Coverage and Profiling in Production Applications?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Menegay

Learn how to set up instrumentation for both code coverage and profiling in a deployed ASP.NET application, rather than an application running from within Visual Studio. This video follows the steps set out in the Profiling ASP.NET in Instrumentation mode from the Command Line TechNote on the Microsoft MSDN Web site.

Watch video (9 minutes)

PREVIOUS NEXT

Web Deployment Projects

2/20/2020 • 2 minutes to read • Edit Online

by Bradley Bartz

Bradley Bartz and Richard Ersek show how the Web Deployment Projects provide additional functionality for building and deploying web site applications created in Visual Studio 2005.

Watch video (65 minutes)

PREVIOUS NEXT

Web Application Projects & Web Deployment Projects

2/20/2020 • 2 minutes to read • Edit Online

by Bradley Bartz

Bradley Bartz shows how the Web Application Projects and Web Deployment Projects for Visual Studio 2005 make developing and deploying ASP.NET projects similar to the Visual Studio .NET 2003 model.

Watch video (66 minutes)

PREVIOUS

Migrating

2/20/2020 • 2 minutes to read • Edit Online

Migrating to ASP.NET

- [Intro to ASP.NET 2.0:] ASP.NET 2.0 Fundamentals
- [Intro to ASP.NET 2.0:] User Interface Elements
- [Migrating from] Classic ASP to ASP.NET
- Intro to ASP.NET for JSP Developers: Welcome to ASP.NET 2.0
- Intro to ASP.NET for JSP Developers: Building Applications
- Intro to ASP.NET for ColdFusion Developers: Adding ASP.NET to Your Repertoire
- Intro to ASP.NET for ColdFusion Developers: Building an ASP.NET Application
- Interop between PHP and the Windows Platform

[Intro to ASP.NET 2.0:] ASP.NET 2.0 Fundamentals

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Developers new to ASP.NET will want to attend this webcast designed to familiarize you with its essential features.

Watch video (70 minutes)

NEXT

[Intro to ASP.NET 2.0:] User Interface Elements

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

This webcast looks at the features of ASP.NET 2.0 that simplify the construction of user interfaces, including master pages, themes, and navigation controls.

Watch video (63 minutes)

PREVIOUS NEXT

[Migrating from] Classic ASP to ASP.NET

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Are you planning to migrate from classic ASP to Microsoft ASP.NET 2.0? This webcast series walks you through the important architectural design considerations to keep in mind when you plan your migration strategy.

Watch video (78 minutes)

PREVIOUS NEXT

Intro to ASP.NET for JSP Developers: Welcome to ASP.NET 2.0

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

This session focuses on the basic background necessary to understand ASP.NET and how ASP.NET integrates with the .NET Framework and Microsoft IIS.

Watch video (61 minutes)

PREVIOUS NEXT

Intro to ASP.NET for JSP Developers: Building Applications

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

This webcast looks at both the similarities and differences between building a Web application in ASP.NET 2.0 and building one using Java.

Watch video (62 minutes)

PREVIOUS NEXT

Intro to ASP.NET for ColdFusion Developers: Adding ASP.NET to Your Repertoire

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

This sessions shows a simple yet real world code comparison between ColdFusion and ASP.NET.

Watch video (49 minutes)

PREVIOUS NEXT

Introduction to ASP.NET for ColdFusion Developers: Building an ASP.NET Application

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

This webcast shows how to build an ASP.NET application starting from a blank form, to give the experienced ColdFusion developer a clear idea of exactly what is involved.

Watch video (77 minutes)

PREVIOUS NEXT

Interop between PHP and the Windows Platform

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

In this webcast, you will learn about building and running PHP applications on the Microsoft Windows Server operating systems.

Watch video (64 minutes)

PREVIOUS

Building 2.0 Applications

2/20/2020 • 2 minutes to read • Edit Online

Building ASP.NET 2.0 Applications

- [Lesson 1:] Getting Started with Visual Web Developer Express
- [Lesson 2:] Creating a Web Forms User Interface
- [Lesson 3:] Understanding More About Events and Postback
- [Lesson 4:] Understanding Web Application State
- [Lesson 5:] Debugging and Tracing Your Website
- [Lesson 6:] Working with Stylesheets and Master Pages
- [Lesson 7:] Databinding to User Interface Controls
- [Lesson 8:] Working with the GridView and FormView
- Watch ASP.NET Development in Action

[Lesson 1:] Getting Started with Visual Web Developer Express

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

In this lesson, you will begin to get familiar with Visual Web Developer Express Edition, and look at the workflow involved in using the program to build dynamic ASP.NET Web sites.

Watch video (28 minutes)

NEXT

[Lesson 2:] Creating a Web Forms User Interface

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

In this lesson, you will explore the various ASP.NET2.0 server controls available in Visual Web Developer Express Edition. In addition, you will begin to get familiar with the windows in Visual Web Developer.

Watch video (49 minutes)

PREVIOUS NEXT

[Lesson 3:] Understanding More About Events and Postback

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

In this lesson, you will learn how events can be used in your code, and how to react when those events occur.

Watch video (17 minutes)

PREVIOUS NEXT

[Lesson 4:] Understanding Web Application State

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

In this lesson, you will see how to maintain state in ASP.NET, including the various objects you can use to store information for your ASP.NET 2.0 applications.

Watch video (40 minutes)

PREVIOUS NEXT

[Lesson 5:] Debugging and Tracing Your Website

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

In this lesson, you will learn how to debug and monitor your ASP.NET 2.0 Web applications.

Watch video (29 minutes)

PREVIOUS NEXT

[Lesson 6:] Working with Stylesheets and Master Pages

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

In this lesson, you will begin to learn how style sheets and MasterPages can be used to maintain a consistent appearance for all pages in a web application.

Watch video (24 minutes)

PREVIOUS NEXT

[Lesson 7:] Databinding to User Interface Controls

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

In this lesson, you will learn how to display data in your ASP.NET2.0 web applications through some of the new databound controls.

Watch video (22 minutes)

PREVIOUS NEXT

[Lesson 8:] Working with the GridView and FormView

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

In this lesson, you will learn about the GridView and FormView controls and how you can use some of the features of these controls to make editing and displaying of database content easier.

Watch video (30 minutes)

PREVIOUS NEXT

Watch ASP.NET Development in Action

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Watch how easy it is to build a "to do" task list application from scratch using ASP.NET 2.0, ASP.NET AJAX, SQL Server Express, and Visual Web Developer. Microsoft's Joe Stagner starts by creating a fresh database and the code to access its data. We then see how to define a ToDo List page that displays and updates content in the database. Finally we learn how to add AJAX functionality to the page to improve the user experience.

Watch video (20 minutes)

PREVIOUS

General IIS Tips

2/20/2020 • 2 minutes to read • Edit Online

How to configure IIS to work with ASP.NET applications.

- Developing and Deploying In a Shared Hosting
- Working with IIS7 Delegated Admin
- Feature Specific Delegated Management
- Troubleshooting Production ASP.NET Apps
- Creating a Site with IIS7 Manager
- Installing FTP7
- Bit Rate Throttling
- IIS7 Playlists

Developing and Deploying In a Shared Hosting

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video introduces developing ASP.NET applications in a shared hosting environment by demonstrating "Live Editing" and "Remote Publication" features in Visual Web Developer 2008.

Watch video (46 minutes)

NEXT

Working with IIS7 Delegated Admin

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video introduces the exciting new remote management functionality built for the IIS7 manager. It illustrates both server and client configuration as well as the two authorization options.

Watch video (22 minutes)

PREVIOUS NEXT

Feature Specific Delegated Management

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video demonstrates how to work with remotely administering SPECFIC web site features with IIS7 remote / delegated administration.

Watch video (8 minutes)

PREVIOUS NEXT

Troubleshooting Production ASP.NET Apps

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video provides you with a quick inventory of tools to troubleshoot your production ASP.NET applications.

Watch video (30 minutes)

PREVIOUS NEXT

Creating a Site with IIS7 Manager

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video demonstrates the provisioning process for new web sites with IIS7.

Watch video (8 minutes)

PREVIOUS NEXT

Installing FTP7

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

FTP7 is new and powerful. This video shows you how to get it up and running!

Watch video (8 minutes)

PREVIOUS NEXT

Bit Rate Throttling

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video will teach how to install and configure a great new IIS7 feature – Bit Rate Throttling. With BRT you can serve media files or big downloads without ruining your sites UI performance.

Watch video (6 minutes)

PREVIOUS NEXT

IIS7 Playlists

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

If you're hosting media files on your web site, play lists will let you define sequences for your media and optionally prevent users from skipping parts of the content (like ads or disclaimers).

Watch video (8 minutes)

PREVIOUS

General VS 2010 Tips

2/20/2020 • 2 minutes to read • Edit Online

Features of Visual Studio 2010, including those specifically for working with ASP.NET.

- Code Optimized Profile
- Code Search View Hierarchy
- IntelliSense Smart Lists
- Multi-Monitor Support
- New Web Project Template
- New Multi-Targeting
- Websites Instead of Web Projects
- Snippets IntelliSense

Visual Studio 2010 "Quick Hit" - Code Optimized Profile

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn how to use the "Web Development (Code Optimized)" profile option in Visual Studio 2010 that hides the designer tab and focuses on a source-only editing experience.

Watch video (3 minutes)

NEXT

Visual Studio 2010 "Quick Hit" - Code Search View Hierarchy

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about Visual Studio's code search view hierarchy.

Watch video (5 minutes)

PREVIOUS NEXT

Visual Studio 2010 "Quick Hit" - IntelliSense Smart Lists

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about list filtering in Visual Studio's IntelliSense.

Watch video (3 minutes)

PREVIOUS NEXT

Visual Studio 2010 "Quick Hit" - Multi-Monitor Support

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about Visual Studio's support of multiple monitor scenarios to improve your use of screen real-estate, and optimize your overall development workflow.

Watch video (5 minutes)

PREVIOUS NEXT

Visual Studio 2010 "Quick Hit" - New Web Project Template

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about Visual Studio's new web project starter template.

Watch video (2 minutes)

PREVIOUS NEXT

Visual Studio 2010 "Quick Hit" - New Multi-Targeting

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn how to use Visual Studio's enhanced multi-targeting support which allows you to use and target multiple versions of .NET.

Watch video (6 minutes)

PREVIOUS NEXT

Visual Studio 2010 "Quick Hit" - Websites Instead of Web Projects

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about Visual Studio IDE Start Page changes which affect how you create or open Websites.

Watch video (3 minutes)

PREVIOUS NEXT

Visual Studio 2010 "Quick Hit" - Snippets IntelliSense

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about Visual Studio's snippet support of HTML, ASP.NET markup and JavaScript.

Watch video (4 minutes)

PREVIOUS

Ajax Control Toolkit

2/20/2020 • 2 minutes to read • Edit Online

How to create very responsive ASP.NET Web pages using the Ajax Control Toolkit.

- Get Started with the ASP.NET AJAX Control Toolkit
- ASP.NET AJAX CascadingDropDown Control Extender
- ASP.NET AJAX TextBoxWatermark Control Extender
- ASP.NET AJAX Popup Control Extender
- ASP.NET AJAX ModalPopup Extender Control
- ASP.NET AJAX AlwaysVisible Control Extender
- ASP.NET AJAX Accordion Control
- ASP.NET AJAX Collapsable Panel Extender
- ASP.NET AJAX Draggable Panel Extender
- ASP.NET AJAX DynamicPopulate Extender
- ASP.NET AJAX FilteredTextbox Extender
- ASP.NET AJAX HoverMenu Extender
- ASP.NET AJAX ToggleButton Extender
- ASP.NET AJAX DropShadow Extender
- ASP.NET AJAX PasswordStrength Extender
- Get Started with the ASP.NET AJAX Animation Extender Control
- ASP.NET AJAX ConfirmButton Extender
- ASP.NET AJAX Slider Control
- ASP.NET AJAX AutoComplete Control
- Configure the ASP.NET AJAX Calendar Control
- ASP.NET AJAX DropDown Control
- ASP.NET AJAX MaskedEdit Controls
- ASP.NET AJAX MutuallyExclusive Checkbox Extender
- ASP.NET AJAX NoBot Control
- ASP.NET AJAX ListSearch Extender
- PagingBulletedList Extender Control
- NumericUpDown Extender Control
- ASP.NET AJAX ValidatorCallout Extender
- ASP.NET AJAX ResizableControl Extender
- ASP.NET AJAX Tabs Control
- ASP.NET AJAX SlideShow Extender
- ASP.NET AJAX UpdatePanelAnimation Extender
- AJAX Toolkit Reorder Control
- Utilize the AJAX Rating Control in the ASP.NET Toolkit
- Control Extenders
- Color Picker
- Combo Box
- Editor Control
- Editor Control Custom

- Create a New Custom Extender

[How Do I:] Get Started with the ASP.NET AJAX Control Toolkit?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Learn how to extend your ASP.NET AJAX applications using the ASP.NET AJAX Control Toolkit. This video starts with the very basics, including downloading and installing the toolkit, and adding a few of its controls to a simple ASP.NET AJAX-enabled web page.

Watch video (11 minutes)

NEXT

[How Do I:] Use the ASP.NET AJAX CascadingDropDown Control Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video demonstrates how ASP.NET's standard DropDownList can become AJAX-enabled using the CascadingDropDown control extender from the ASP.NET AJAX Control Toolkit. You will also see how easy it is for a web form to draw its data from an XML Web Service.

Watch video (27 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX TextBoxWatermark Control Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn how to use the TextBoxWatermark control extender from the ASP.NET AJAX Control Toolkit, which adds new watermark properties to all standard TextBox controls on an ASP.NET page.

Watch video (6 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX Popup Control Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video demonstrates the use of the Popup control extender that ships with the ASP.NET AJAX Control Toolkit. Learn how to extend a TextBox control so that, when it receives focus, a popup dialog will appear with a list of options.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX ModalPopup Extender Control?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Learn how to use the ASP.NET AJAX ModalPopup extender control that comes with the ASP.NET AJAX Control Toolkit. In this video the ModalPopup extender is used to pop open a standard ASP.NET Panel control as a modal dialog box.

Watch video (10 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX AlwaysVisible Control Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

See how easy it is to anchor a page element to a position on screen, where it remains always visible no matter how far the user may scroll the document. The floating page element may be anything from static text to an AJAX-based, context-aware help box..

Watch video (6 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX Accordion Control?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Learn how the ASP.NET AJAX Accordion control makes efficient use of screen real-estate by enabling the smooth animated transition between different content panes.

Watch video (9 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX Collapsable Panel Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn how to use the ASP.NET AJAX Collapsable Panel Extender to allow the user to dynamically show or hide content on a web page.

Watch video (11 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX Draggable Panel Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

With the ASP.NET AJAX Draggable Panel Extender you can provide a content panel with a header area that allows the user to move the panel around the web page.

Watch video (6 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX DynamicPopulate Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Watch a demonstration of using the ASP.NET AJAX DynamicPopulate extender to dynamically populate an area of a web page with the results of an asynchronous call to a web method.

Watch video (11 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX FilteredTextbox Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Learn how the handy ASP.NET AJAX FilteredTextbox extender allows the developer to define which characters the user may enter in a text box control.

Watch video (5 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX HoverMenu Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Discover how to use the HoverMenu extender control to create a context-sensitive menu that appears whenever the user hovers the mouse over an element of a web page.

Watch video (7 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX ToggleButton Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Demonstrates using the ToggleButton extender control to modify the behavior and appearance of the standard ASP.NET CheckBox control to enhance the user interface.

Watch video (7 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX DropShadow Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Discover how to improve the presentation of ASP.NET sites by using the AJAX DropShadow extender control to add opaque, rounded-corner drop shadows to areas of your pages.

Watch video (2 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX PasswordStrength Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Learn how the PasswordStrength extender control provides immediate and dynamic feedback to users about the strength of their chosen passwords. The feedback can be a text notice, a graphical bar indicator, or a combination of both.

Watch video (15 minutes)

PREVIOUS NEXT

[How Do I:] Get Started with the ASP.NET AJAX Animation Extender Control?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video introduces the Animation extender control from the ASP.NET AJAX Control Toolkit, which provides developers with a fully-extensible framework for adding animation effects to an ASP.NET Web page.

Watch video (20 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX ConfirmButton Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Learn how the ConfirmButton extender in the ASP.NET AJAX Control Toolkit provides a simple way to add a confirmation dialog to the click event of a Button, LinkButton, or ImageButton control. This provides the user with the option to cancel an impending action.

Watch video (3 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX Slider Control?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

The Slider control from the ASP.NET AJAX Control Toolkit provides a simple and elegant way for the user to select a numeric value. This video shows how to add a Slider control to your Web page and how to configure its orientation, range of values, and size of increments.

Watch video (10 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX AutoComplete Control

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

The AutoComplete extender control from the ASP.NET AJAX Control Toolkit provides real-time suggestions to the user as he or she types in a text box on the Web page, where the list of suggestions is obtained by an asynchronous call to a Web service on the server.

Watch video (7 minutes)

PREVIOUS NEXT

[How Do I:] Configure the ASP.NET AJAX Calendar Control?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video we see a demonstration of the variety of options available to configure the appearance and the behavior of the Calendar extender control from the ASP.NET AJAX Control Toolkit.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX DropDown Control?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

With the DropDown control extender from the ASP.NET AJAX Control Toolkit, you can present the user with a drop-down list of options that trigger a partial-page update elsewhere on the Web page. The option selected by the user is passed to the server, allowing your server-side code to process the selection before completing the partial-page update.

Watch video (11 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX MaskedEdit Controls?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

The ASP.NET AJAX Control Toolkit provides the MaskedEdit control and the MaskedEditValidator control which work together to prompt, style, and validate the data that the user enters in a text box. This allows the Web page to provide the user with immediate visual feedback on whether the data entered is of a valid type and format.

Watch video (17 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX MutuallyExclusive Checkbox Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

The MutuallyExclusive control from the ASP.NET AJAX Control Toolkit enables the developer to configure a number of check boxes as mutually exclusive, which ensures the user may only check a single option. While a radio-button list provides similar behavior, the need for a check box to be clicked by the user makes the MutuallyExclusive control ideal for receiving confirmation from the user, such as accepting or not accepting terms of use.

Watch video (10 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX NoBot Control?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

The NoBot control from the ASP.NET AJAX Control Toolkit helps prevent certain types of unauthorized access to your Web applications. In this video we see how the NoBot control provides a number of built-in guards against automated access, including checking the number and frequency of form submissions. We also see how the control allows the developer to define a custom challenge, such as a JavaScript test that will succeed only if the page is viewed in a browser.

Watch video (21 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX ListSearch Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video demonstrates the use of the ListSearch extender control from the ASP.NET AJAX Control Toolkit. The ListSearch extender enhances DropDownList and ListBox controls so that a user can select an item in the list by simply typing its first few characters. This enhancement improves the user experience by providing an intuitive way of selecting an item from a long list.

Watch video (11 minutes)

PREVIOUS NEXT

[How Do I:] Use the PagingBulletedList Extender Control?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video demonstrates the use of the PagingBulletedList extender control from the ASP.NET AJAX Control Toolkit. We learn how the PagingBulletedList extender adds client-side paging and sorting to the standard ASP.NET BulletedList control, which greatly improves the user experience when presenting an extensive list of items. We also see how we can define custom JavaScript and CSS to fully manage the user experience.

Watch video (25 minutes)

PREVIOUS NEXT

[How Do I:] Use the NumericUpDown Extender Control?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video we examine the NumericUpDown extender control from the ASP.NET AJAX Control Toolkit. We see how the NumericUpDown extender adds "up" and "down" buttons to a standard ASP.NET TextBox control. The buttons allow the user to increment or decrement a number, cycle through a list of known values, or call a Web Service to retrieve the next value. In this way the developer can fully control the values the user may submit to a Web application.

Watch video (21 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX ValidatorCallout Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

The standard ASP.NET validation controls work on both the server and the client. Server-side validation adds to the security of your Web application, while client-side validation adds to the user friendliness of your Web application. This video demonstrates how the ValidatorCallout control from the ASP.NET AJAX Control Toolkit extends the standard validation controls to enhance the user experience by neatly positioning validation messages in callout boxes.

Watch video (11 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX ResizableControl Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

The ResizableControl extender from the ASP.NET AJAX Control Toolkit allows the developer to specify a region of the Web page that the user may resize dynamically. This video demonstrates how to configure the look and feel of the resizable region, and how to run client-side script upon resizing.

Watch video (30 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX Tabs Control?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video demonstrates the basics of the Tabs control from the ASP.NET AJAX Control Toolkit, with which you can create a set of tabs to organize content on a Web page. The selected tab is maintained across post backs, making the Tabs control ideal for form-based administration pages.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX SlideShow Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video demonstrates how the SlideShow extender from the ASP.NET AJAX Control Toolkit turns the standard ASP.NET Image control in to a fully-functional slide show. We also see how an ASP.NET Web service can return not only text and XML data but complete AJAX objects to be consumed by the ASP.NET AJAX Web application. Joe also comments on the benefits of storing images in the file system rather than the database.

Watch video (19 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX UpdatePanelAnimation Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

By adding animation to a partial-page update we can inform the user when an update is occuring, and when it has finished. The UpdatePanelAnimation extender adds animation effects to the ASP.NET AJAX UpdatePanel control, making use of the rich animation framework of the core ASP.NET AJAX Extensions. This video shows how you can define the sequence of effects for the UpdatePanelAnimation, and how you can fine-tune the animation by using ScriptAction definitions.

Watch video (20 minutes)

PREVIOUS NEXT

[How Do I:] The AJAX Toolkit Reorder Control

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video I will demonstrate the use of the reorder control that ships with the AJAX toolkit. The reorder control lets you present to your users a list of data items that can be reordered as the user prefers.

Watch video (21 minutes)

PREVIOUS NEXT

Utilize the AJAX Rating Control in the ASP.NET Toolkit

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Joe Stagner demonstrates how to use the Rating Tool that ships with ASP.NET AJAX Toolkit.

Watch video (20 minutes)

PREVIOUS NEXT

Control Extenders

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Watch this video to understand the difference between Controls and AJAX Control Extenders and how to use those extenders in your applications.

Watch video (3 minutes)

PREVIOUS NEXT

Color Picker

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video will demonstrate how to use the new Color Picker control from the Microsoft AJAX Control Toolkit.

Watch video (9 minutes)

PREVIOUS NEXT

Combo Box

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video will demonstrate how to use the new Combo Box control from the Microsoft AJAX Control Toolkit.

Watch video (9 minutes)

PREVIOUS NEXT

Editor Control

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video will demonstrate how to use the new Editor WYSIWYG Editor control from the Microsoft AJAX Control Toolkit.

Watch video (4 minutes)

PREVIOUS NEXT

Editor Control Custom

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video Joe demonstrates how to create a new class, derived from the AJAX Control Toolkit's Editor control and implementing a "simple" user.

Watch video (8 minutes)

PREVIOUS NEXT

Create a New Custom Extender

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video Joe demonstrates how to use the AJAX Control Toolkit to create a new custom AJAX Control Extender.

Watch video (26 minutes)

PREVIOUS

ASP.NET AJAX

2/20/2020 • 2 minutes to read • Edit Online

Ways to work with ASP.NET AJAX to create highly responsive pages, even if you're not a JavaScript expert.

- Get Started with ASP.NET AJAX
- Implement Dynamic Partial-Page Updates with ASP.NET AJAX
- Make Client-Side Network Callbacks with ASP.NET AJAX
- Add ASP.NET AJAX Features to an Existing Web Application
- ASP.NET AJAX Enable an Existing Web Service
- ASP.NET AJAX Client Library Controls
- Use an ASP.NET AJAX ScriptManagerProxy
- ASP.NET AJAX RoundedCorners Extender
- ASP.NET AJAX Timer Control
- Implement the Predictive Fetch Pattern for AJAX
- Implement the AJAX Paging Pattern
- Implement the AJAX Incremental Page Display Pattern
- Implement the Incremental Page Display Pattern using HTTP GET and POST
- ASP.NET AJAX UpdateProgress Control
- ASP.NET AJAX History Control
- Implement the AJAX After Processing Pattern
- Update Multiple Regions of a Page with ASP.NET AJAX
- Choose Between Methods of AJAX Page Updates
- Use Other JavaScript User Interface Libraries with ASP.NET AJAX
- Use the ASP.NET AJAX Profile Services
- Debug ASP.NET AJAX Applications Using Visual Studio 2005
- Build a Custom ASP.NET AJAX Server Control
- Use JavaScript to Refresh an ASP.NET AJAX UpdatePanel
- Determine Whether an Asynchronous Postback has Occurred
- Use the Conditional UpdateMode of the UpdatePanel
- Implement the Persistent Communications Pattern with the UpdatePanel
- Localize an ASP.NET AJAX Application
- Implement the Persistent Communications Pattern using Web Services
- Trigger an UpdatePanel Refresh from a DropDownList Control
- Create an ASP.NET AJAX Extender from Scratch
- Build Custom Server Controls that Work With or Without ASP.NET AJAX
- Associate AJAX Client Behavior with an ASP.NET Server Control
- Retrieve Values From Server Side AJAX Controls
- Two Simple Techniques for Triggering Updates to Update Panels
- Use ASP.NET AJAX Cascading Drop Down Control to Access a Database
- Implement Infinite Data Patterns in AJAX
- Basic ASP.NET Authentication in an AJAX Enabled Application
- Dynamically Change CSS ASP.NET AJAX UpdatePanel
- Dynamically Add Controls to a Web Page

- Set Up Your Development Environment for ASP.NET 3.5
- Set Up Your Development Environment for ASP.NET 2.0
- Customize Error Handling for the ASP.NET AJAX UpdatePanel
- Use ASP.NET AJAX Client Templates

[How Do I:] Get Started with ASP.NET AJAX?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Note that ASP.NET AJAX is included as part of .NET 3.5, so the installation part of this video is only relevant if you are using the .NET Framework Version 2.0.

Watch video (7 minutes)

NEXT

[How Do I:] Implement Dynamic Partial-Page Updates with ASP.NET AJAX?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn how to add dynamic partial-page updates to your ASP.NET application using Microsoft ASP.NET AJAX. In particular, you will see how to use the TimerControl to manage the frequency of the AJAX callbacks.

Watch video (12 minutes)

PREVIOUS NEXT

[How Do I:] Make Client-Side Network Callbacks with ASP.NET AJAX?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video we see how Microsoft ASP.NET AJAX helps web developers make network callbacks directly from client-side script code. We see also how an ASP.NET AJAX-enabled web service generates the JavaScript needed to call the web service.

Watch video (15 minutes)

PREVIOUS NEXT

[How Do I:] Add ASP.NET AJAX Features to an Existing Web Application?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Learn how easily you add Microsoft ASP.NET AJAX functionality to an existing ASP.NET application. This video shows that there is no need to change your server controls or data access code - just add and configure a few ASP.NET AJAX controls. Simple!

Watch video (7 minutes)

PREVIOUS NEXT

[How Do I:] ASP.NET AJAX Enable an Existing Web Service?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video demonstrates how simple it is to add Microsoft ASP.NET AJAX functionality to an existing ASP.NET web service. This enables the web service to generate a JavaScript proxy that the client browser can use to call the web service.

Watch video (23 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX Client Library Controls?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Demonstrates the client-side programming experience using scripts and controls from the ASP.NET AJAX client library. The client controls add actions, behaviors, and data-bindings to the client-side user interface experience.

Watch video (10 minutes)

PREVIOUS NEXT

[How Do I:] Use an ASP.NET AJAX ScriptManagerProxy?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Learn how a ScriptManagerProxy enables a content page to pass references to the ScriptManager placed on its ASP.NET master page, allowing each content page to define its own AJAX behavior.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX RoundedCorners Extender?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video we see how the RoundedCorner extender control can give your web pages a softer look and feel by adding attractive round corners to content areas.

Watch video (2 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX Timer Control?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

The AJAX Timer control enables a portion of an ASP.NET web page to be dynamically updated at a regular interval, rather than needing the user to perform an action such as clicking on a button.

Watch video (6 minutes)

PREVIOUS NEXT

[How Do I:] Implement the Predictive Fetch Pattern for AJAX?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Demonstrates an implementation of the Predictive Fetch pattern for AJAX, where the AJAX engine uses the current state of the user's interaction to predict which set of content will be needed next, and then fetches that data in advance from the server. This provides for a more responsive user experience.

Watch video (24 minutes)

PREVIOUS NEXT

[How Do I:] Implement the AJAX Paging Pattern?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video we see how to use ASP.NET AJAX to implement the paging pattern, which enables the ASP.NET application to present a large amount of data as being separated into smaller "pages" of data, and allow the user to move forward and backward among the pages.

Watch video (27 minutes)

PREVIOUS NEXT

[How Do I:] Implement the AJAX Incremental Page Display Pattern?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Traditionally when a Web page is requested, the user must wait while the entire page is retrieved from the server after which the complete page appears suddenly in the browser. This video demonstrates how the AJAX Incremental Page Display Pattern improves the user experience by quickly and smoothly fetching and displaying portions of the Web page.

Watch video (11 minutes)

PREVIOUS NEXT

[How Do I:] Implement the Incremental Page Display Pattern using HTTP GET and POST?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video takes another look at the AJAX Incremental Page Display Pattern in which the Web page is retrieved and displayed incrementally. In this video the data is retrieved from the server using HTTP GET and POST requests, rather than the earlier demonstration of using Web service calls.

Watch video (19 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX UpdateProgress Control?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In an AJAX application there is often a noticeable pause between the time the browser makes a network request and the time the browser receives a response. To let the user know that the network conversation is taking place, the ASP.NET AJAX UpdateProgress control provides immediate feedback by displaying an animated image, 'loading' text, or other HTML that disappears once the network response is received.

Watch video (6 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX History Control?

2/20/2020 • 2 minutes to read • Edit Online

by Scott Golightly

One problem faced by a typical AJAX application is that a partial page update is not added to the history of the Web browser. This means that the browser's Back button does not move back one AJAX step, but moves back one entire document, which is unlikely to be what the user expects. The ASP.NET AJAX History control allows the developer to insert history points so that the user may click the browser's Back and Forward buttons to move between AJAX steps.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Implement the AJAX After Processing Pattern?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

The ASP.NET AJAX framework allows your server-side code to hook in and respond to various client-side events. In this video we learn how to implement the After Processing Pattern, which allows server-side code to execute wrap-up logic once the user has finished with the Web page.

Watch video (5 minutes)

PREVIOUS NEXT

[How Do I:] Update Multiple Regions of a Page with ASP.NET AJAX?

2/20/2020 • 2 minutes to read • Edit Online

by Scott Golightly

In this video we learn that the ASP.NET AJAX UpdatePanel control provides two properties that you may use to update multiple regions of a Web page in response to a single client-side event. The UpdateMode property may be set to Always, in which case the panel will always update during a post back, or set to Conditional, in which case the panel will only update in response to specific events. The Triggers collection property allows you to specify those events that will cause the panel to update. Used together these properties allow multiple UpdatePanel controls to update in response to the same event trigger.

Watch video (7 minutes)

PREVIOUS NEXT

[How Do I:] Choose Between Methods of AJAX Page Updates?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video Joe Stagner compares the two primary methods of performing AJAX-style page updates in an ASP.NET application. The first method is to use an UpdatePanel, where no additional code needs to be written on either the client side or the server side. The benefit of using the UpdatePanel is that everything works automatically. The penalty is that at the client it requires a lot of data to be included in the AJAX request and response, and at the server it requires a full page lifecycle to be executed. The second method is to use network callbacks, where additional code needs to be written on both the client side and the server side. The benefit of using network callbacks is that at the client it requires very little data to be included in the AJAX request and response, and at the server it requires only the called service method to be executed. The penality is the time and effort it takes to write the necessary code. Joe concludes the video by discussing what you should consider when choosing between the two primary methods of AJAX-style page updates. (This video uses the code from the How Do I Get Started with ASP.NET AJAX video and the How Do I Make Client-Side Network Callbacks with ASP.NET AJAX video.)

Watch video (11 minutes)

PREVIOUS NEXT

[How Do I:] Use Other JavaScript User Interface Libraries with ASP.NET AJAX?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Microsoft is part of the OpenAjax alliance and the Microsoft AJAX Library is designed to work seamlessly with third-party JavaScript user interface libraries. This video provides a demonstration of interoperability between ASP.NET AJAX, Prototype, and script.aculo.us.

Watch video (13 minutes)

PREVIOUS NEXT

[How Do I:] Use the ASP.NET AJAX Profile Services?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

ASP.NET 2.0 introduced Profile properties with which you can store details for each user of your ASP.NET Web application. ASP.NET AJAX Profile Services allow your client-side Web interface to retrieve a user's Profile properties from the server-side Web application. This video demonstrates how a sample AJAX-enabled travel site can store the travel request information of a user so that the details can be retrieved when he or she next visits the site.

Watch video (11 minutes)

PREVIOUS NEXT

[How Do I:] Debug ASP.NET AJAX Applications Using Visual Studio 2005?

2/20/2020 • 2 minutes to read • Edit Online

by Scott Golightly

In this video we learn how to use Visual Studio 2005 to debug ASP.NET AJAX applications. We are shown how the ScriptManager is used to access the JavaScript created by the ASP.NET AJAX server controls, and we also see how to use the Sys.Debug class to insert a trace message and a breakpoint.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Build a Custom ASP.NET AJAX Server Control?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

Learn how to create a custom Web server control with ASP.NET AJAX functionality using the AJAX extensions in the Microsoft AJAX Library. This video walks you through both the server-side .NET code and the client-side JavaScript code, and provides an explanation of the 'prototype' concept. The custom ASP.NET AJAX control can then be used in an .aspx page like any other server control.

Watch video (20 minutes)

PREVIOUS NEXT

[How Do I:] Use JavaScript to Refresh an ASP.NET AJAX UpdatePanel?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video we learn two different ways of using JavaScript logic to refresh the content of an ASP.NET AJAX UpdatePanel control. The first way is to add a Button-type server control to the UpdatePanel and use JavaScript to trigger an asynchronous postback and 'pretend' that the postback was caused by the server control inside the UpdatePanel. The second way does not require any Button-type server control to be added to the UpdatePanel, but still allows us to use JavaScript logic to refresh the content of the UpdatePanel. (The second technique was suggested by Dave Ward and you can read more on his weblog entry, " Easily refresh an UpdatePanel, using JavaScript ".)

Watch video (10 minutes)

PREVIOUS NEXT

[How Do I:] Determine Whether an Asynchronous Postback has Occurred?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

The Page_Load method of an ASP.NET Web form will execute during either a standard full-page postback or an asychnronous partial-page postback. In this video we learn how we can use the ASP.NET AJAX ScriptManager to determine whether an UpdatePanel postback has occurred, which allows us to conditionally execute code.

Watch video (5 minutes)

PREVIOUS NEXT

[How Do I:] Use the Conditional UpdateMode of the UpdatePanel?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

The ASP.NET AJAX UpdatePanel includes an UpdateMode property that may be set to 'Always' or 'Conditional'. The default is Always, in which case the UpdatePanel will always update its content during an asynchronous postback. In this video we learn how we can set the UpdateMode to Conditional, in which case the UpdatePanel will only update its content when our server-side code calls its Update method. This allows you to use conditional logic in your C# or Visual Basic code to determine whether the UpdatePanel will update its content during the current asynchronous postback.

Watch video (13 minutes)

PREVIOUS NEXT

[How Do I:] Implement the Persistent Communications Pattern with the UpdatePanel?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In a traditional Web site the browser and the server do not maintain an ongoing communication, but communicate only in response to the user performing an action. In a modern Web site where the page becomes an application container, it can be advantageous for the browser and the server to maintain an ongoing communication so that page updates can occur without the user performing an action. This is known as the Persistent Communications Pattern for AJAX. ASP.NET AJAX provides two main ways for Web developers to implement the Persistent Communications Pattern. This video demonstrates the simple way, which is to use the ASP.NET AJAX UpdatePanel as the basis of the implementation. In a later video we will learn how to implement the same pattern without the use of the ASP.NET AJAX UpdatePanel.

Watch video (12 minutes)

PREVIOUS NEXT

[How Do I:] Localize an ASP.NET AJAX Application?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video we start with an overview of the localization capabilities of the standard ASP.NET framework, and then look at the localization features of the ASP.NET AJAX Extensions. We learn how we can localize text, date, and number information on the client based upon the browser's language settings. We also learn how our AJAX application can make use of static and embedded resources back on the server.

Watch video (28 minutes)

PREVIOUS NEXT

[How Do I:] Implement the Persistent Communications Pattern using Web Services?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In a traditional Web site the browser and the server do not maintain an ongoing communication, but communicate only in response to the user performing an action. In a modern Web site where the page becomes an application container, it can be advantageous for the browser and the server to maintain an ongoing communication so that page updates can occur without the user performing an action. This is known as the Persistent Communications Pattern for AJAX. ASP.NET AJAX provides two main ways for Web developers to implement the Persistent Communications Pattern. In an earlier video we saw how to use the ASP.NET AJAX UpdatePanel as the basis of the implementation. In this video we learn how to implement the same pattern using a JavaScrpt call to a Web service, which removes the need for an ASP.NET AJAX UpdatePanel.

Watch video (16 minutes)

PREVIOUS NEXT

[How Do I:] Trigger an UpdatePanel Refresh from a DropDownList Control?

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In most of our videos on the ASP.NET AJAX UpdatePanel, we have used a Button control to cause an UpdatePanel to refresh its content. Yet we may use any event raised by any other ASP.NET server control. This video uses the SelectedIndexChanged event of the DropDownList control as a trigger for refreshing an UpdatePanel control. We see also how we can dynamically change the stylesheet class associated with the controls contained in the UpdatePanel control.

Watch video (9 minutes)

PREVIOUS NEXT

[How Do I:] Create an ASP.NET AJAX Extender from Scratch?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

Learn how to create an ASP.NET AJAX extender for a standard ASP.NET server control. We are shown how to add server-side properties and client-side JavaScript to extend the behavior of the standard TextBox control, though the same approach can be applied to any other server control. For further help in creating an ASP.NET AJAX extender, please refer to the Creating a New Extender walkthrough.

Watch video (15 minutes)

PREVIOUS NEXT

[How Do I:] Build Custom Server Controls that Work With or Without ASP.NET AJAX?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

Do you have an idea for a custom server control that should be able to work without ASP.NET AJAX, yet could take advantage of ASP.NET AJAX if it is available? In this video we learn how to build a custom server control that does not reference the ASP.NET AJAX assembly, but uses Reflection to "play nice" with the ASP.NET AJAX UpdatePanel.

Watch video (19 minutes)

PREVIOUS NEXT

[How Do I:] Associate AJAX Client Behavior with an ASP.NET Server Control?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

Learn how to create a client behavior based upon Microsoft AJAX that can be associated with an ASP.NET server control on an .aspx page. The client behavior extends the functionality of the standard ASP.NET control through association rather than by creating a new custom control. Therefore, the behavior can be reused across multiple instances or types of ASP.NET controls depending upon its implementation.

Watch video (18 minutes)

PREVIOUS NEXT

[How Do I:] Retrieve Values From Server Side AJAX Controls

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video I will demonstrate how to retrieve values from server side AJAX controls like the ones found in the AJAX toolkit.

Watch video (8 minutes)

PREVIOUS NEXT

Two Simple Techniques for Triggering Updates to Update Panels

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video I will demonstrate two simple update techniques. First, using the selected index changed event from the drop down list control as an event trigger for updating update panels. And two, dynamically changing the associated style sheet class with controls contained within the AJAX update panel.

Watch video (9 minutes)

PREVIOUS NEXT

Use ASP.NET AJAX Cascading Drop Down Control to Access a Database

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video demonstrates how ASP.NET's standard DropDownList can become AJAX-enabled using the CascadingDropDown control extender from the ASP.NET AJAX Control Toolkit. You will also see how easy it is for a web form to draw data from a Database.

Watch video (32 minutes)

PREVIOUS NEXT

Implement Infinite Data Patterns in AJAX

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video I will show you how to implement what I refer to as the infinite data pattern for AJAX.

Watch video (18 minutes)

PREVIOUS NEXT

Basic ASP.NET Authentication in an AJAX Enabled Application

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video I will demonstrate authentication in your AJAX enable web application using basic ASP.NET forms based authentication.

Watch video (31 minutes)

PREVIOUS NEXT

How To Dynamically Change CSS Using the ASP.NET AJAX UpdatePanel

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video illustrates how to dynamically modify the style of your page using CSS and the AJAX UpdatePanel.

Watch video (11 minutes)

PREVIOUS NEXT

How To Dynamically Add Controls to a Web Page

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video I will demonstrate a technique to dynamically add an indeterminate number of controls to your web page.

Watch video (31 minutes)

PREVIOUS NEXT

Set Up Your Development Environment for ASP.NET 3.5

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you'll learn how to set up your Development Environment for developing ASP.NET AJAX Applications if you are targeting ASP.NET 3.5.

Watch video (8 minutes)

PREVIOUS NEXT

Set Up Your Development Environment for ASP.NET 2.0

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you'll learn how to set up your Development Environment for developing ASP.NET AJAX Applications if you are targeting ASP.NET 2.0.

Watch video (15 minutes)

PREVIOUS NEXT

How Do I: Customize Error Handling for the ASP.NET AJAX UpdatePanel

2/20/2020 • 2 minutes to read • Edit Online

by Rob Windsor

When an error occurs during the processing of a postback triggered by an ASP.NET AJAX UpdatePanel, the client-side PageRequestManager will show an alert (.NET 2.0 or 3.0) or just re-throw the exception (.NET 3.5 or later). In either case, the experience for the end user is less that optimal. In this video, Rob Windsor will demonstrate how to address this issue by customizing the handling of errors associated with the UpdatePanel

Watch video (17 minutes)

PREVIOUS NEXT

How Do I: Use ASP.NET AJAX Client Templates

2/20/2020 • 2 minutes to read • Edit Online

by Rob Windsor

There are two key steps involved in building AJAX interactions, calling services asynchronously and updating the page with the resulting data. ASP.NET AJAX has always made the first part easy but it hasn't provided much support for the second. To address this issue, Microsoft has added client-side templates and data binding and in this video, Rob Windsor will demonstrate how to use these new features in your AJAX applications.

Watch video (33 minutes)

PREVIOUS

Data Access

2/20/2020 • 2 minutes to read • Edit Online

How to create a Data Access Layer (DAL).

- ADO.NET Data Services
  - Data Access Layers in ASP.NET Applications
  - Manually Bind a Dataset to a Datagrid
  - Work with Datasets and Filters from an ASP Application
- How Do I with LINQ
  - LINQ to SQL: Overview
  - LINQ to SQL: Data Model
  - LINQ to SQL: Querying the Database
  - LINQ to SQL: Updating the Database
  - LINQ to SQL: LinqDataSource
  - LINQ to SQL: Custom LinqDataSource
  - LINQ to SQL: Using Stored Procedures
  - LINQ to SQL: Updating with Stored Procedures
  - LINQ to SQL: Executing Arbitrary SQL
- LINQ Videos from the VB Team
  - Get Started with LINQ
  - Perform Group and Aggregate Queries
  - Upgrade Visual Basic Projects to Enable LINQ
  - Get Started with LINQ to XML
  - Enable XML IntelliSense and Use XML Namespaces
  - Create XML Documents from SQL Data
  - Create Excel Spreadsheets using LINQ to XML

ADO.NET Data Services

2/20/2020 • 2 minutes to read • Edit Online

These videos show you how to create a data access layer using a typed data set for a simple ASP.NET application.

- Data Access Layers in ASP.NET Applications
- Manually Bind a Dataset to a Datagrid
- Work with Datasets and Filters from an ASP Application

Data Access Layers in ASP.NET Applications

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video, part 1 of 3, Joe Stagner creates a data access layer using a typed data set for a simple ASP.NET application. For additional information about this video, read the Creating a Data Access Layer data access tutorial.

Watch video (11 minutes)

NEXT

How to Manually Bind a Dataset to a Datagrid

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video, part 2 of 3, Joe Stagner uses the sample ASP application created in part one, to demonstrate how to manually bind a dataset to a datagrid. For additional information about this video, read the Creating a Data Access Layer data access tutorial.

Watch video (7 minutes)

PREVIOUS NEXT

How to Work with Datasets and Filters from an ASP Application

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video, part 3 of 3, Joe Stagner demonstrates how to return a filtered set of data from the overall application dataset. For additional information about this video, read the Creating a Data Access Layer data access tutorial.

Watch video (6 minutes)

PREVIOUS

How Do I with LINQ

2/20/2020 • 2 minutes to read • Edit Online

This video series walks you through everything you need to know to become proficient in using LINQ as a web developer.

- LINQ to SQL: Overview
- LINQ to SQL: Data Model
- LINQ to SQL: Querying the Database
- LINQ to SQL: Updating the Database
- LINQ to SQL: LinqDataSource
- LINQ to SQL: Custom LinqDataSource
- LINQ to SQL: Using Stored Procedures
- LINQ to SQL: Updating with Stored Procedures
- LINQ to SQL: Executing Arbitrary SQL

[How Do I:] LINQ to SQL: Overview

2/20/2020 • 2 minutes to read • Edit Online

by Scott Stanfield

New language features for C# and VB support data access directly from code, without writing SQL. This video shows the minimum steps to use LINQ (Language Integrated Query) to query, update, add and delete rows from the Northwind database.

Watch video (9 minutes)

NEXT

[How Do I:] LINQ to SQL: Data Model

2/20/2020 • 2 minutes to read • Edit Online

by Scott Stanfield

LINQ to SQL relies on an object-to-relational model set of classes that map to your database tables. Visual Studio 2008 includes a visual ORM design surface that maps these tables to code.

Watch video (4 minutes)

PREVIOUS NEXT

[How Do I:] LINQ to SQL: Querying the Database

2/20/2020 • 2 minutes to read • Edit Online

by Scott Stanfield

This video shows how to query a database with LINQ, the underlying SQL with a query visualizer, databinding to ASP.NET controls and record paging.

Watch video (6 minutes)

PREVIOUS NEXT

[How Do I:] LINQ to SQL: Updating the Database

2/20/2020 • 2 minutes to read • Edit Online

by Scott Stanfield

In this video I demonstrated how to create, update and delete records through LINQ, simple transactions, updates across relationships, and custom column and row validation.

Watch video (17 minutes)

PREVIOUS NEXT

[How Do I:] LINQ to SQL: LinqDataSource

2/20/2020 • 2 minutes to read • Edit Online

by Scott Stanfield

The new asp:LinqDataSource control .NET 3.5 makes binding to UI elements easy. In fact, no code was written or harmed in the making of this video. Query, update, delete, paging, sorting, all done through LINQ and the LinqDataSource.

Watch video (11 minutes)

PREVIOUS NEXT

[How Do I:] LINQ to SQL: Custom LinqDataSource

2/20/2020 • 2 minutes to read • Edit Online

by Scott Stanfield

The LinqDataSource control can be customized declaritively, as shown in Part 5, or through code by handling the Selecting method.

Watch video (9 minutes)

PREVIOUS NEXT

[How Do I:] LINQ to SQL: Using Stored Procedures

2/20/2020 • 2 minutes to read • Edit Online

by Scott Stanfield

Using existing stored procedures and functions is easy with LINQ. We simply drag the stored procedures onto the O/R mapping surface and call them from the generated Data Context object.

Watch video (9 minutes)

PREVIOUS NEXT

How Do I: LINQ to SQL: Updating with Stored Procedures

2/20/2020 • 2 minutes to read • Edit Online

by Scott Stanfield

Override the generated ORM update/delete/insert logic with your own through stored procedures. The video also shows in detail how to use the SQL Profiler to monitor the changes in the ORM from generated SQL to our custom SQL stored procs. Since we're just writing SQL, the same techniques apply to both C# and VB developers.

Watch video (15 minutes)

PREVIOUS NEXT

How Do I: LINQ to SQL: Executing Arbitrary SQL

2/20/2020 • 2 minutes to read • Edit Online

by Scott Stanfield

This video shows how to execute any arbitrary SQL through the ExecuteCommand syntax and encapsulating the code through the partial DataContext class.

Watch video (4 minutes)

PREVIOUS

LINQ Videos from the VB Team

2/20/2020 • 2 minutes to read • Edit Online

These videos provide over an hour and half of content focused on the new LINQ language features of Visual Basic 9.0 included in Visual Studio 2008.

- Get Started with LINQ
- Perform Group and Aggregate Queries
- Upgrade Visual Basic Projects to Enable LINQ
- Get Started with LINQ to XML
- Enable XML IntelliSense and Use XML Namespaces
- Create XML Documents from SQL Data
- Create Excel Spreadsheets using LINQ to XML

[How Do I:] Get Started with LINQ?

2/20/2020 • 2 minutes to read • Edit Online

by Beth Massi

Visual Basic programmers will learn how to write simple LINQ queries over objects by navigating files on the file system.

Watch video (9 minutes)

NEXT

[How Do I:] Perform Group and Aggregate Queries?

2/20/2020 • 2 minutes to read • Edit Online

by Beth Massi

Visual Basic programmers will learn how to perform group and aggregate queries over business object collections and how to display them in related grids.

Watch video (17 minutes)

PREVIOUS NEXT

[How Do I:] Upgrade Visual Basic Projects to Enable LINQ?

2/20/2020 • 2 minutes to read • Edit Online

by Beth Massi

Visual Basic programmers will learn about the new multi-targeting feature and how to upgrade projects written in Visual Basic 2005 to Visual Basic 2008 Beta 2 in order to enable different LINQ providers.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Get Started with LINQ to XML?

2/20/2020 • 2 minutes to read • Edit Online

by Beth Massi

Visual Basic developers will learn how to work with XML in Visual Basic using XML literals, embedded expressions and XML axis properties to create, query and transform XML documents easily.

Watch video (14 minutes)

PREVIOUS NEXT

[How Do I:] Enable XML IntelliSense and Use XML Namespaces?

2/20/2020 • 2 minutes to read • Edit Online

by Beth Massi

Visual Basic developers will learn how to enable XML IntelliSense using schemas and how XML namespaces are used when querying and creating XML documents.

Watch video (13 minutes)

PREVIOUS NEXT

[How Do I:] Create XML Documents from SQL Data?

2/20/2020 • 2 minutes to read • Edit Online

by Beth Massi

Visual Basic developers will learn how to query their SQL Server database in order to easily create XML and HTML documents directly in the Visual Basic editor.

Watch video (15 minutes)

PREVIOUS NEXT

[How Do I:] Create Excel Spreadsheets using LINQ to XML?

2/20/2020 • 2 minutes to read • Edit Online

by Beth Massi

Visual Basic developers will learn how to query their SQL Server database in order to easily create Excel spreadsheets directly in the Visual Basic editor using LINQ to XML.

Watch video (10 minutes)

PREVIOUS

ASP.NET Dynamic Data

2/20/2020 • 2 minutes to read • Edit Online

How to use ASP.NET Dynamic Data to create scaffolded applications that automatically know how to perform data access.

- Your First Scaffold and What is Dynamic Data
- Enable Inline GridView Editing
- Change how my Fields render
- Handle Business Logic Exceptions
- Make Custom Pages
- Display Unknown datatypes
- Use a DynamicControl in ListView and DetailsView Controls
- Getting Started with Dynamic Data
- Begin Editing the Templates in ASP.NET Dynamic Data Applications
- Begin Modifying Dynamic Data Applications with URL Routing
- Enable In-Line Editing in ASP.NET Dynamic Data Applications
- Enable Table Specific Routing in Dynamic Data Applications
- Use Attribute Validation in ASP.NET Dynamic Data Applications
- Implement Custom Field Validation with Imperative Logic in VB or C#
- Remove Columns From Your DynamicData Data Grids
- Create Table Specific Custom Forms in an ASP.NET Dynamic Data Application
- ASP.NET Dynamic Data Custom Form Formatting

Your First Scaffold and What is Dynamic Data?

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

In this short introduction, you create a complete Dynamic Data application in minutes using scaffolding, and in the process learn about Dynamic Data.

Watch video (6 minutes)

NEXT

How Do I Enable Inline GridView Editing?

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Now that you have created a site with Edit and Details views, how do you enable inline editing in a **GridView** control? This video shows you how, and also touches on routing.

Watch video (6 minutes)

PREVIOUS NEXT

How do I Change how my Fields render?

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

In this video you learn how to change the way that data fields are rendered. You will plug in third-party controls for some fields.

Watch video (13 minutes)

PREVIOUS NEXT

How do I Handle Business Logic Exceptions?

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

This video shows you how to add validation to the data model. You will be able to see how that validation logic is reflected in the UI.

Watch video (5 minutes)

PREVIOUS NEXT

How Do I Make Custom Pages?

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

In this video, you learn how to make a custom page for some tables, while other pages use standard functionality.

Watch video (5 minutes)

PREVIOUS NEXT

How do I Display Unknown datatypes?

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

In this video, we show you how to configure Dynamic Data to handle a totally unknown custom data type that is stored in the database as a binary BLOB.

Watch video (8 minutes)

PREVIOUS NEXT

How Do I Use a DynamicControl in ListView and DetailsView Controls?

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

This video compares the same application written twice, once with Dynamic Data and once without. In the process, you add **DynamicControl** objects to **ListView** and **DetailsView** controls.

Watch video (5 minutes)

PREVIOUS NEXT

Getting Started with Dynamic Data

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video we'll build an entire ASP.NET application with ASP.NET Dynamic Data using the Northwind Data by using Dynamic Data's code generating, scaffolding and templates. The finished application is complete with AJAX Behaviors and a database layer with full CRUD functionality.

Watch video (23 minutes)

PREVIOUS NEXT

Begin Editing the Templates in ASP.NET Dynamic Data Applications

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video will explore the application structure of your generated ASP.NET Dynamic Data Application and get you started on editing the provided templates.

Watch video (8 minutes)

PREVIOUS NEXT

Begin Modifying Dynamic Data Applications with URL Routing

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video introduces ASP.NET Dynamic Data URL routing and shows how to configure an application's URL with routing options.

Watch video (5 minutes)

PREVIOUS NEXT

Enable In-Line Editing in ASP.NET Dynamic Data Applications

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video will demonstrate using URL Routing to enable In-Line editing in your ASP.NET Dynamic Data Application.

Watch video (13 minutes)

PREVIOUS NEXT

How to Enable Table Specific Routing in Dynamic Data Applications

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video learn how to add custom workflow logic to your ASP.NET dynamic data application by using Routing to enable table-specific .aspx file usage.

Watch video (6 minutes)

PREVIOUS NEXT

How to Use Attribute Validation in ASP.NET Dynamic Data Applications

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

This video will introduce modifying the Partial Classes in our Dynamic Data Application's Data Access Layer and get us started using Attributes by focusing on the Validation Attributes. This Data Access layer validation is preferable to validation at the User Interface Layer and is very easy to do with ASP.NET Dynamic Data.

Watch video (11 minutes)

PREVIOUS NEXT

How to Implement Custom Field Validation with Imperative Logic in VB or C#

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video we will build on the declarative field validation we implemented in our previous Dynamic Data Video and add custom programming logic to our validation.

Watch video (9 minutes)

PREVIOUS NEXT

How to Remove Columns From Your DynamicData Data Grids

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you'll see how to take advantage of full column scaffolding but still exclude table columns that you don't want to appear in your user interface.

Watch video (5 minutes)

PREVIOUS NEXT

How to Create Table Specific Custom Forms in an ASP.NET Dynamic Data Application

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Sometimes we need to deviate from the default UI in our ASP.NET Dynamic Data Application. In this video we'll see how we can do that, and how we can turn off auto column generation and construct a web form that displays only the data we specifically select.

Watch video (10 minutes)

PREVIOUS NEXT

ASP.NET Dynamic Data Custom Form Formatting

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will gain some insight on how you can configure the visual aesthetics of a Dynamic Data application down to the very specifics of each field's display.

Watch video (6 minutes)

PREVIOUS

ASP.NET 3.5

2/20/2020 • 2 minutes to read • Edit Online

How to build ASP.NET 3.5 applications, with an emphasis on new features.

- How Do I
  - Create a Master Page in Visual Studio 2008
  - Create Nested Master Page in Visual Studio 2008
  - Cascading Style Sheets in Visual Studio 2008
  - Working with Visual Studio 2008 .NET Framework
  - Adding Elements to a CSS File and Create New CSS on the Fly
  - Advance Cascading Style Sheet Features and Management
  - Converting a .NET 2.0 Windows Forms Application to .NET 3.5
  - Get Started with the Entity Framework
  - Use the New Entity Data Source
  - Serialize a Graph with the Entity Framework
  - Use MSBuild to Automate the ASP.NET Compiler and Merge Utilities
- ASP.NET Ajax
  - ASP.NET AJAX Support in Visual Studio 2008
  - Adding AJAX Functionality to an Existing ASP.NET Page
  - Creating and Using an AJAX-enabled Web Service in a Web Site
  - ASP.NET AJAX: A demonstration of ASP.NET AJAX
  - ADO.NET Data Services with ASP.NET AJAX Support
  - Intro to ASP.NET Ajax History
  - Using Script Combining to improve AJAX performance
- Data Controls
  - ListView Control
  - DataPager Control
- Visual Studio Designer
  - Visual Studio 2008 and Nested Masterpages
- Visual Studio Editor
  - New Designer Support in Visual Studio 2008
  - JavaScript Intellisense Support in Visual Studio 2008
  - JavaScript Debugging in Visual Studio 2008
  - Multi Targeting Support in Visual Studio 2008
  - IntelliSense for Jscript and ASP.NET AJAX
  - Quick Tour of the Visual Studio 2008 Integrated Development Environment
  - Creating and Modifying a CSS File

How Do I

2/20/2020 • 2 minutes to read • Edit Online

This multi-part video series shows you how to take advantage of new features in ASP.NET 3.5 and Visual Studio 2008.

- Create a Master Page in Visual Studio 2008
- Create Nested Master Page in Visual Studio 2008
- Cascading Style Sheets in Visual Studio 2008
- Working with Visual Studio 2008 .NET Framework
- Adding Elements to a CSS File and Create New CSS on the Fly
- Advance Cascading Style Sheet Features and Management
- Converting a .NET 2.0 Windows Forms Application to .NET 3.5
- Get Started with the Entity Framework
- Use the New Entity Data Source
- Serialize a Graph with the Entity Framework
- Use MSBuild to Automate the ASP.NET Compiler and Merge Utilities

[How Do I:] Create a Master Page in Visual Studio 2008

2/20/2020 • 2 minutes to read • Edit Online

by Asli Bilgin

In this video I will demonstrate how to create a simple Master Page using .NET 3.5 in Visual Studio 2008.

Watch video (8 minutes)

NEXT

[How Do I:] Create Nested Master Page in Visual Studio 2008

2/20/2020 • 2 minutes to read • Edit Online

by Asli Bilgin

Using Visual Studio 2008's improved Designer, I will demonstrate how to create a Nested Master Page. Nested Master Pages are useful when you build a hierarchy of templates for your web site.

Watch video (12 minutes)

PREVIOUS NEXT

[How Do I:] Cascading Style Sheets in Visual Studio 2008

2/20/2020 • 2 minutes to read • Edit Online

by Asli Bilgin

In this video we will review the improved tools for creating external Cascading Style Sheets. CSS are vital to any well designed web site and Visual Studio 2008 has provided an arsenal of tools for creating and managing CSS.

Watch video (12 minutes)

PREVIOUS NEXT

[How Do I:] Working with Visual Studio 2008 .NET Framework

2/20/2020 • 2 minutes to read • Edit Online

by Asli Bilgin

In this video I will show you how to use Visual Studio 2008 to work with various versions of the .NET framework. This is called multi-targeting. I will also cover new templates in Visual Studio 2008.

Watch video (4 minutes)

PREVIOUS NEXT

[How Do I:] Adding Elements to a CSS File and Create New CSS on the Fly

2/20/2020 • 2 minutes to read • Edit Online

by Asli Bilgin

In this video we will build on an existing style sheet and enhance it with element ids using the new CSS management tools in Visual Studio 2008.

Watch video (16 minutes)

PREVIOUS NEXT

[How Do I:] Advance Cascading Style Sheet Features and Management

2/20/2020 • 2 minutes to read • Edit Online

by Asli Bilgin

In this video we will build upon the basic Cascading Style Sheet from a previous video. We will advance beyond the basics where we just created elements and now we will create custom classes. We will then learn how to manage classes with the Build Style Sheet Wizard of Visual Studio 2008.

Watch video (9 minutes)

PREVIOUS NEXT

[How Do I:] Converting a .NET 2.0 Windows Forms Application to .NET 3.5

2/20/2020 • 2 minutes to read • Edit Online

by Asli Bilgin

Using the new multi-targeting feature of Visual Studio 2008, I will demonstrate upgrading a project from .NET 2.0 to .NET 3.5. In this video we will take a .NET 2.0 Windows forms application and upgrade it to .NET 3.5.

Watch video (8 minutes)

PREVIOUS NEXT

[How Do I:] Get Started with the Entity Framework

2/20/2020 • 2 minutes to read • Edit Online

by Alex James

In this introductory video, Alex James will show you how to build a simple Entity Data Model in a short amount of time.

Watch video (9 minutes)

PREVIOUS NEXT

[How Do I:] Use the New Entity Data Source

2/20/2020 • 2 minutes to read • Edit Online

by Diego Vega

In this introductory video Diego Vega will show you how to use the Entity Data Source in an ASP.NET Application.

Watch video (11 minutes)

PREVIOUS NEXT

[How Do I:] Serialize a Graph with the Entity Framework

2/20/2020 • 2 minutes to read • Edit Online

by Brian Dawson

In this introductory video Brian Dawson will show you how to serialize a Graph when developing an ASP.NET application with the ADO.NET Entity Framework.

Watch video (4 minutes)

PREVIOUS NEXT

[How Do I:] Use MSBuild to Automate the ASP.NET Compiler and Merge Utilities

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to create an MSBuild project that orchestrates the execution of the aspnet_compiler and aspnet_merge utilities for an ASP.NET web site. First, learn the fundamentals of the compilation process for an ASP.NET web site and the role of the two command line utilities. Next, see how to create an MSBuild project file and the primary elements such as PropertyGroup and Target. Then learn the details of how to execute the aspnet_compiler and aspnet_merge utilities from the MSBuild project. Lastly, several potential expansions of the core concepts are discussed.

Watch video (31 minutes)

PREVIOUS

ASP.NET Ajax

2/20/2020 • 2 minutes to read • Edit Online

This video series explores how to add Ajax functionality to an ASP.NET application.

- ASP.NET AJAX Support in Visual Studio 2008
- Adding AJAX Functionality to an Existing ASP.NET Page
- Creating and Using an AJAX-enabled Web Service in a Web Site
- ASP.NET AJAX: A demonstration of ASP.NET AJAX
- ADO.NET Data Services with ASP.NET AJAX Support
- Intro to ASP.NET Ajax History
- Using Script Combining to improve AJAX performance

ASP.NET AJAX Support in Visual Studio 2008

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

You will get a quick overview of the inclusion of ASP.NET AJAX into Visual Studio 2008 (Orcas) and .NET 3.5. This will be demonstrated by using Visual Studio 2008 and the building of web site application.

Watch video (9 minutes)

NEXT

Adding AJAX Functionality to an Existing ASP.NET Page

2/20/2020 • 2 minutes to read • Edit Online

by Todd Miranda

Using AJAX in ASP.NET pages can increase performance and improve the overall user experience of an application. It is not necessary to rewrite existing pages to add AJAX features. With only a few modifications, an existing ASP.NET page can be made to use AJAX. This video demonstrates how to add AJAX functionality to an existing ASP.NET page with minimal effort.

Watch video (11 minutes)

PREVIOUS NEXT

Creating and Using an AJAX-enabled Web Service in a Web Site

2/20/2020 • 2 minutes to read • Edit Online

by Rob Windsor

Visual Studio 2008 lets you create ASP.NET custom Web services that you can access from client scripts. This walkthrough will demonstrate how to create an AJAX-enabled Web service and use its methods from client script running in separate Web application.

Watch video (12 minutes)

PREVIOUS NEXT

ASP.NET AJAX: A demonstration of ASP.NET AJAX

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

See how easy it is to build from scratch a "To Do List" management application using ASP.NET 2.0, and then enhance the user experience using the ASP.NET AJAX Extensions.

Watch video (20 minutes)

PREVIOUS NEXT

ADO.NET Data Services with ASP.NET AJAX Support

2/20/2020 • 2 minutes to read • Edit Online

by Levi Broderick

Find out how ADO.NET Data Services can be integrated with ASP.NET AJAX support seamlessly. **ADP.NET Data Services technology is available in the ASP.NET 3.5 Extensions Preview (December 2007).**

Watch video (12 minutes)

PREVIOUS NEXT

Introduction to ASP.NET Ajax History

2/20/2020 • 2 minutes to read • Edit Online

by Bertrand Le Roy

Learn how the ASP.NET Extensions Preview allows control over the Browser back button in Ajax. **ASP.NET AJAX Extensions are available in the ASP.NET 3.5 Extensions Preview (December 2007).**

Watch video (14 minutes)

PREVIOUS NEXT

Using Script Combining to improve AJAX performance

2/20/2020 • 2 minutes to read • Edit Online

Betrand Le Roy outlines how to improve the performance of an Ajax application using a new feature called Script Combining, utilizing a simple page with sample code from the Ajax toolkit.

Watch video (4 minutes)

PREVIOUS

Data Controls

2/20/2020 • 2 minutes to read • Edit Online

This set of videos explores how to use new data control features with ASP.NET 3.5

- ListView Control
- DataPager Control

The ListView Control

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

By creating a new website, I will demonstrate the new ListView control. Because the ListView is data drive, we will need a data source. You can use any data source: SQL, object data, link data or XAML. I will use my blog as the data source in this video.

Watch video (10 minutes)

NEXT

The DataPager Control

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Generally the DataPager control is used with the ListView control. The ListView control is data driven. First create a ListView control and provide it some data. Next drop the DataPager control onto the page to enhance the web page functionally. If you have not watched the ListView control How-Do-I video it would be best to review it before using the DataPager control.

Watch video (8 minutes)

PREVIOUS

Visual Studio Designer

2/20/2020 • 2 minutes to read • Edit Online

This video illustrates enhanced features in Visual Studio 2008 for working with master pages.

- Visual Studio 2008 and Nested Masterpages

Visual Studio 2008 and Nested Masterpages

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

I will use a web template from oswd.org to demonstrate the use of Master Pages and how they are supported in Visual Studio 2008. Specifically, I will show the inheritance features by using two Master Pages.

Watch video (13 minutes)

Visual Studio Editor

2/20/2020 • 2 minutes to read • Edit Online

This video series shows you new features of the Visual Studio 2008 editor, including enhanced support for JavaScript and CSS.

- New Designer Support in Visual Studio 2008
- JavaScript Intellisense Support in Visual Studio 2008
- JavaScript Debugging in Visual Studio 2008
- Multi Targeting Support in Visual Studio 2008
- IntelliSense for Jscript and ASP.NET AJAX
- Quick Tour of the Visual Studio 2008 Integrated Development Environment
- Creating and Modifying a CSS File

New Designer Support in Visual Studio 2008

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video I will demonstrate several of the new designer features in Visual Studio 2008.

Watch video (14 minutes)

NEXT

JavaScript Intellisense Support in Visual Studio 2008

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video I will highlight some of JavaScript Intellisense built into Visual Studio 2008. I will show ways of improving the Intellisense JavaScript feature by the way you write your JavaScript code.

Watch video (11 minutes)

PREVIOUS NEXT

JavaScript Debugging in Visual Studio 2008

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Using an existing web application, I will show you how to use Visual Studio 2008 to debug JavaScript in a web application.

Watch video (9 minutes)

PREVIOUS NEXT

Multi Targeting Support in Visual Studio 2008

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

I will illustrate the MultiTarget feature of Visual Studio 2008. In the past, you had to choose the version of Visual Studio to match the targeted .NET version. For example, Visual Studio 2005 worked with .NET 2.5. Now, with Visual Studio 2008 you can choose the version of .NET appropriate for your application.

Watch video (5 minutes)

PREVIOUS NEXT

IntelliSense for Jscript and ASP.NET AJAX

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

This video provides an overview of the new Intellisense features for JavaScript in Visual Studio 2008 including core JavaScript methods, external .js files in .aspx pages, between external .js files, with core Microsoft libraries like MicrosoftAjax.js, client JavaScript behaviors, controls, and libraries, and web services. In addition, the integration of the new XML comments for JavaScript with Intellisense is demonstrated.

Watch video (20 minutes)

PREVIOUS NEXT

Quick Tour of the Visual Studio 2008 Integrated Development Environment

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

Take a tour of the new features and changes to the IDE in Visual Studio 2008. See the updated web page editor, WPF form designer, CSS editor, and more.

Watch video (14 minutes)

PREVIOUS NEXT

Creating and Modifying a CSS File

2/20/2020 • 2 minutes to read • Edit Online

by Todd Miranda

This video introduces the cascading style sheet (CSS) features of Visual Studio 2008. Visual Studio 2008's CSS features make it easy to create style sheets and styles through a property style interface. This removes the developer from having to deal with the implementation details of CSS.

Watch video (15 minutes)

PREVIOUS

jQuery

2/20/2020 • 2 minutes to read • Edit Online

How to use jQuery features in ASP.NET pages.

- Make Ajax Calls Using jQuery

How Do I: Make Ajax Calls Using jQuery?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to make Ajax calls using the jQuery library. A sample web site is created that contains a simple web service which returns a value. Next, the jQuery ajax() method is used to call the web service. A number of the ajax() method options are set including url, dataType, data, etc. In addition, event handlers for both success and error conditions are defined. Another web services is then defined which accepts an argument and also returns a value. A second ajax() call is defined that passes a value to the web service call and processes the return data. Finally, the setup for global options and event handlers for ajax() method calls are discussed.

Watch video (21 minutes)

.NET 4

2/20/2020 • 2 minutes to read • Edit Online

The new ASP.NET features and functionality available in the .NET Framework 4.

- Ajax
  - Imperative JavaScript Syntax for Microsoft Client Side Controls
  - ScriptLoader
  - JQuery Syntax for Microsoft Ajax
  - AJAX Data Templates
- Chart
  - Chart Control
  - Introducing the New Chart Control in Visual Studio 2010
- Core
  - Auto Start
  - Clean Web.Config Files
  - Predictable Client IDs
  - HtmlEncoder Utility Method
- Data
  - Persistent GridView Row Selection
- Markup Control
  - New Rendering Option for Check Box Lists and Radio Button Lists
  - Table Free Templated Controls
  - Tableless Menu Control
  - Hidden Field Divs
  - Disabled Control Styling
- Performance
  - Selective View State
  - Easy State Compression
  - Use the ViewStateMode Property for Managing ViewState
- Routing
  - Permanent Redirect
  - Imperative WebForms Routing
  - Declarative WebForms Routing
  - Outbound WebForms Routing
  - Use Routing with ASP.NET Web Forms
  - Work with URLs in ASP.NET Routing
- SEO
  - Dynamic Metadata

Ajax

2/20/2020 • 2 minutes to read • Edit Online

This video series shows the new features in ASP.NET 4 for JavaScript and jQuery support.

- Imperative JavaScript Syntax for Microsoft Client Side Controls
- ScriptLoader
- JQuery Syntax for Microsoft Ajax
- AJAX Data Templates

ASP.NET 4 "Quick Hit" - Imperative JavaScript Syntax for Microsoft Client Side Controls

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn the imperative JavaScript syntax for Microsoft client side controls.

Watch video (10 minutes)

NEXT

ASP.NET 4 "Quick Hit" - The ScriptLoader

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video will learn about the ScriptLoader.

Watch video (14 minutes)

PREVIOUS NEXT

ASP.NET 4 "Quick Hit" - jQuery Syntax for Microsoft Ajax

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about jQuery syntax for Microsoft Ajax.

Watch video (6 minutes)

PREVIOUS NEXT

ASP.NET 4 "Quick Hit" - AJAX Data Templates

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn how to use client templates to display database records with the DataView control.

Watch video (12 minutes)

PREVIOUS

Chart

2/20/2020 • 2 minutes to read • Edit Online

This set of videos shows how to use the ASP.NET Chart control to create visually compelling data display.

- Chart Control
- Introducing the New Chart Control in Visual Studio 2010

ASP.NET 4 "Quick Hit" - Chart Control

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn how to use the ASP.NET Chart Control to create simple, intuitive, and visually compelling charts.

Watch video (2 minutes)

NEXT

How Do I: Introducing the New Chart Control in Visual Studio 2010

2/20/2020 • 2 minutes to read • Edit Online

by Max Adams

There is a lot of excitement around the imminent release of Microsoft Visual Studio 2010 and the .NET Framework 4.0. Amongst many new features is the ASP.NET Web Charting Control. In this video, Max Adams introduces the control and demonstrates just how easy it actually is to create visually appealing statistical or financial analysis tools in a browser using the Chart control.

Watch video (13 minutes)

PREVIOUS

Core

2/20/2020 • 2 minutes to read • Edit Online

This video series explores core improvements in ASP.NET 4 like auto-start, clean configuration files, and HTML encoding.

- Auto Start
- Clean Web.Config Files
- Predictable Client IDs
- HtmlEncoder Utility Method

ASP.NET 4 "Quick Hit" - Auto Start

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about the new auto-start scalability feature which provides a controlled approach for starting up an application pool, initializing an ASP.NET application, and then accepting HTTP requests.

Watch video (11 minutes)

NEXT

ASP.NET 4 "Quick Hit" - Clean Web.Config Files

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will see how the major configuration elements have been moved to the machine.config file. This allows the Web.config file in ASP.NET 4 applications to either be empty or contain a minimum number of lines.

Watch video (2 minutes)

PREVIOUS NEXT

ASP.NET 4 "Quick Hit" - Predictable Client IDs

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about the ClientIDMode property, which lets you specify more precisely how the client ID is generated for controls.

Watch video (10 minutes)

PREVIOUS NEXT

ASP.NET 4 "Quick Hit" – The HtmlEncoder Utility Method

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about the HtmlEncoder utility method.

Watch video (5 minutes)

PREVIOUS

Data

2/20/2020 • 2 minutes to read • Edit Online

This video shows you new features for ASP.NET 4 data controls.

- Persistent GridView Row Selection

ASP.NET 4 "Quick Hit" – Persistent GridView Row Selection

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about persisted row selection for the GridView control.

Watch video (3 minutes)

Markup Control

2/20/2020 • 2 minutes to read • Edit Online

This video series demonstrates improvements to controls for Web Forms pages in ASP.NET 4.

- New Rendering Option for Check Box Lists and Radio Button Lists
- Table Free Templated Controls
- Tableless Menu Control
- Hidden Field Divs
- Disabled Control Styling

ASP.NET 4 "Quick Hit" – New Rendering Option for Check Box Lists and Radio Button Lists

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about the RepeatLayout rendering option for Check Box and Radio Button lists.

Watch video (4 minutes)

NEXT

ASP.NET 4 "Quick Hit" – Table Free Templated Controls

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about the RenderOuterTable property for templated controls, which allows you to control whether or not the outer table is rendered to the page.

Watch video (5 minutes)

PREVIOUS NEXT

ASP.NET 4 "Quick Hit" – Tableless Menu Control

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will see how the ASP.NET Menu control renders HTML using semantic markup consisting of an unordered list and list elements

Watch video (5 minutes)

PREVIOUS NEXT

ASP.NET 4 "Quick Hit" - Hidden Field Divs

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn how div elements around hidden fields are now styled so they do not interfere with user-created CSS rules.

Watch video (3 minutes)

PREVIOUS NEXT

ASP.NET 4 "Quick Hit" - Disabled Control Styling

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn how to use the controlRenderingCompatibility setting so disabling non-input controls will not render invalid styles.

Watch video (6 minutes)

PREVIOUS

Performance

2/20/2020 • 2 minutes to read • Edit Online

This video series covers new ways to optimize Web Forms pages in ASP.NET 4.

- Selective View State
- Easy State Compression
- Use the ViewStateMode Property for Managing ViewState

ASP.NET 4 "Quick Hit" - Selective View State

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about the ViewStateMode property that lets you disable view state by default and then enable it only for the controls that require it in the page.

Watch video (6 minutes)

NEXT

ASP.NET 4 "Quick Hit" – Easy State Compression

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn how to use the compressionEnabled configuration option to compress (and decompress) serialized session state.

Watch video (2 minutes)

PREVIOUS NEXT

How Do I: Use the ViewStateMode Property for Managing ViewState?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to use the ViewStateMode property when designing ViewState for a web site in ASP.NET. First, the functionality for managing ViewState in ASP.NET prior to ASP.NET 4 is discussed. Next, the ViewStateMode property added in ASP.NET 4 is demonstrated both at the page level and in a sample page with nested child controls. The importance of the hierarchical inheritance of the ViewStateMode property is discussed. Finally, a sample scenario for minimizing ViewState on ASP.NET pages is demonstrated.

Watch video (16 minutes)

PREVIOUS

Routing

2/20/2020 • 2 minutes to read • Edit Online

This video shows you how to create intuitive and SEO-friendly URLs for your Web Forms applications.

- Permanent Redirect
- Imperative WebForms Routing
- Declarative WebForms Routing
- Outbound WebForms Routing
- Use Routing with ASP.NET Web Forms
- Work with URLs in ASP.NET Routing

ASP.NET 4 "Quick Hit" - Permanent Redirect

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn how to use the RedirectPermanent helper method to issue HTTP 301 responses.

Watch video (5 minutes)

NEXT

ASP.NET 4 "Quick Hit" - Imperative WebForms Routing

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn how to use an expression builder to do WebForms routing imperatively.

Watch video (12 minutes)

PREVIOUS NEXT

ASP.NET 4 "Quick Hit" - Declarative WebForms Routing

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn how to do WebForms routing declaratively through markup.

Watch video (14 minutes)

PREVIOUS NEXT

ASP.NET 4 "Quick Hit" - Outbound WebForms Routing

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will see how to use the routing mechanism to dynamically create an outbound URL based on input from the user.

Watch video (6 minutes)

PREVIOUS NEXT

How Do I: Use Routing with ASP.NET Web Forms?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to implement routing for Web Forms in ASP.NET 4. First, the concept of routing a URL is compared to mapping the URL to a physical file in the site. Then, a sample route for a URL is defined in the global.asax file Application_Start event handler. The route contains a parameterized value that the user can enter in the URL. A sample page is then created and the route parameter value is extracted in the Page_Load event handler. Next, a second route is defined that has multiple parameters and routes to the same page as the initial route. The Page_Load event handler is expanded to extract the additional route parameter value and display different information depending upon what values have been passed to the page.

Watch video (15 minutes)

PREVIOUS NEXT

How Do I: Work with URLs in ASP.NET Routing?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to specify URLs in a web site that utilizes ASP.NET routing. First, a web site is created and routing is defined in the Global Application Class (.asax). Next, a sample web page is created and a URL based upon a defined route is added to the page using the standard "hard coded" approach, e.g., "~/Stats/Visitors". Another link is then added to the page which dynamically generates the same URL in markup using the RouteValue method which accepts the route name and parameters. The same URL is then implemented using code rather than markup directly in the page. The original route and physical page location are then changed, resulting in the hard coded link no longer working whereas both dynamically generated links function properly. Finally, the value of dynamically generated links is then discussed.

Watch video (20 minutes)

PREVIOUS

SEO

2/20/2020 • 2 minutes to read • Edit Online

This video describes new features for Search Engine Optimization (SEO) in ASP.NET 4.

- Dynamic Metadata

ASP.NET 4 "Quick Hit" - Dynamic Metadata

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video you will learn about the Page.MetaKeywords and Page.MetaDescription properties, which allow you to set the page meta data dynamically at run time.

Watch video (5 minutes)

ASP.NET 4.5

2/20/2020 • 2 minutes to read • Edit Online

About new Web Forms features introduced in ASP.NET 4.5.

- Strongly Typed Data Controls
- Model Binding Part 1 - Selecting Data
- Model Binding Part 2 - Filtering
- Model Binding Part 3 - Updating
- ASP.NET 4.5 Web Forms Model Binding
- ASP.NET 4.5 Web Forms Strong Typed Data Controls
- Bundling and Minification
- Getting Started with the Next Version of ASP.NET
- ASP.NET and Web Tools 2012.2

ASP.NET vNext Videos: Strongly Typed Data Controls

2/20/2020 • 2 minutes to read • Edit Online

by Damian Edwards

In this video, Damian Edwards demonstrates strongly typed data controls in Visual Studio vNext, which provide IntelliSense and compile time checking.

Watch video (1 minutes)

NEXT

ASP.NET vNext Videos: Model Binding Part 1 - Selecting Data

2/20/2020 • 2 minutes to read • Edit Online

by Damian Edwards

In this video, Damian Edwards begins a look at model binding by demonstrating data selection.

Watch video (1 minutes)

PREVIOUS NEXT

ASP.NET vNext Videos: Model Binding Part 2 - Filtering

2/20/2020 • 2 minutes to read • Edit Online

by Damian Edwards

In this video, Damian Edwards continues the demonstration of model binding by explaining filtering.

Watch video (1 minutes)

PREVIOUS NEXT

ASP.NET vNext Videos: Model Binding Part 3 - Updating

2/20/2020 • 2 minutes to read • Edit Online

by Damian Edwards

In this video, Damian Edwards completes the review of model binding by demonstrating how data is updated.

Watch video (1 minutes)

PREVIOUS NEXT

ASP.NET 4.5 Web Forms Model Binding

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Watch video (7 minutes)

PREVIOUS NEXT

ASP.NET 4.5 Web Forms Strong Typed Data Controls

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Watch video (4 minutes)

PREVIOUS NEXT

ASP.NET vNext Videos: Bundling and Minification

2/20/2020 • 2 minutes to read • Edit Online

by Mads Kristensen

Mads Kristensen demonstrates how the ASP.NET vNext can automatically bundle and modify JavaScript and CSS resources to improve performance.

Watch video (1 minutes)

PREVIOUS NEXT

Getting Started with the Next Version of ASP.NET

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Scott Hanselman introduces you to the upcoming features in the next version of ASP.NET.

Watch video (3 minutes)

PREVIOUS NEXT

ASP.NET and Web Tools 2012.2

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

Meet the new ASP.NET and Web Tools 2012.2 release! This release includes a pile of great new features, including project templates, web tooling, and more. Join Jon Galloway as he shows off all the top new features in under a half an hour!

Watch video (26 minutes)

PREVIOUS

What is Web Forms

2/20/2020 • 11 minutes to read • Edit Online

ASP.NET Web Forms is a part of the ASP.NET web application framework and is included with Visual Studio. It is one of the four programming models you can use to create ASP.NET web applications, the others are ASP.NET MVC, ASP.NET Web Pages, and ASP.NET Single Page Applications.

Web Forms are pages that your users request using their browser. These pages can be written using a combination of HTML, client-script, server controls, and server code. When users request a page, it is compiled and executed on the server by the framework, and then the framework generates the HTML markup that the browser can render. An ASP.NET Web Forms page presents information to the user in any browser or client device.

Using Visual Studio, you can create ASP.NET Web Forms. The Visual Studio Integrated Development Environment (IDE) lets you drag and drop server controls to lay out your Web Forms page. You can then easily set properties, methods, and events for controls on the page or for the page itself. These properties, methods, and events are used to define the web page's behavior, look and feel, and so on. To write server code to handle the logic for the page, you can use a .NET language like Visual Basic or C#.

> **NOTE**
>
> ASP.NET and Visual Studio documentation spans several versions. Topics that highlight features from previous versions may be useful for your current tasks and scenarios using the latest versions.

ASP.NET Web Forms are:

- Based on Microsoft ASP.NET technology, in which code that runs on the server dynamically generates Web page output to the browser or client device.
- Compatible with any browser or mobile device. An ASP.NET Web page automatically renders the correct browser-compliant HTML for features such as styles, layout, and so on.
- Compatible with any language supported by the .NET common language runtime, such as Microsoft Visual Basic and Microsoft Visual C#.
- Built on the Microsoft .NET Framework. This provides all the benefits of the framework, including a managed environment, type safety, and inheritance.
- Flexible because you can add user-created and third party controls to them.

ASP.NET Web Forms offer:

- Separation of HTML and other UI code from application logic.
- A rich suite of server controls for common tasks, including data access.
- Powerful data binding, with great tool support.
- Support for client-side scripting that executes in the browser.
- Support for a variety of other capabilities, including routing, security, performance, internationalization, testing, debugging, error handling and state management.

ASP.NET Web Forms Helps You Overcome Challenges

Web application programming presents challenges that do not typically arise when programming traditional client-based applications. Among the challenges are:

- **Implementing a rich Web user interface** - It can be difficult and tedious to design and implement a user

interface using basic HTML facilities, especially if the page has a complex layout, a large amount of dynamic content, and full-featured user-interactive objects.

- **Separation of client and server** - In a Web application, the client (browser) and server are different programs often running on different computers (and even on different operating systems). Consequently, the two halves of the application share very little information; they can communicate, but typically exchange only small chunks of simple information.
- **Stateless execution** - When a Web server receives a request for a page, it finds the page, processes it, sends it to the browser, and then discards all page information. If the user requests the same page again, the server repeats the entire sequence, reprocessing the page from scratch. Put another way, a server has no memory of pages that it has processed—page are stateless. Therefore, if an application needs to maintain information about a page, its stateless nature can become a problem.
- **Unknown client capabilities** - In many cases, Web applications are accessible to many users using different browsers. Browsers have different capabilities, making it difficult to create an application that will run equally well on all of them.
- **Complications with data access** - Reading from and writing to a data source in traditional Web applications can be complicated and resource-intensive.
- **Complications with scalability** - In many cases Web applications designed with existing methods fail to meet scalability goals due to the lack of compatibility between the various components of the application. This is often a common failure point for applications under a heavy growth cycle.

Meeting these challenges for Web applications can require substantial time and effort. ASP.NET Web Forms and the ASP.NET framework address these challenges in the following ways:

- **Intuitive, consistent object model** - The ASP.NET page framework presents an object model that enables you to think of your forms as a unit, not as separate client and server pieces. In this model, you can program the page in a more intuitive way than in traditional Web applications, including the ability to set properties for page elements and respond to events. In addition, ASP.NET server controls are an abstraction from the physical contents of an HTML page and from the direct interaction between browser and server. In general, you can use server controls the way you might work with controls in a client application and not have to think about how to create the HTML to present and process the controls and their contents.
- **Event-driven programming model** - ASP.NET Web Forms bring to Web applications the familiar model of writing event handlers for events that occur on either the client or server. The ASP.NET page framework abstracts this model in such a way that the underlying mechanism of capturing an event on the client, transmitting it to the server, and calling the appropriate method is all automatic and invisible to you. The result is a clear, easily written code structure that supports event-driven development.
- **Intuitive state management** - The ASP.NET page framework automatically handles the task of maintaining the state of your page and its controls, and it provides you with explicit ways to maintain the state of application-specific information. This is accomplished without heavy use of server resources and can be implemented with or without sending cookies to the browser.
- **Browser-independent applications** - The ASP.NET page framework enables you to create all application logic on the server, eliminating the need to explicitly code for differences in browsers. However, it still enables you to take advantage of browser-specific features by writing client-side code to provide improved performance and a richer client experience.
- **.NET Framework common language runtime support** - The ASP.NET page framework is built on the .NET Framework, so the entire framework is available to any ASP.NET application. Your applications can be written in any language that is compatible that is with the runtime. In addition, data access is simplified using the data access infrastructure provided by the .NET Framework, including ADO.NET.
- **.NET Framework scalable server performance** - The ASP.NET page framework enables you to scale your Web application from one computer with a single processor to a multi-computer Web farm cleanly and without complicated changes to the application's logic.

Features of ASP.NET Web Forms

- **Server Controls**- ASP.NET Web server controls are objects on ASP.NET Web pages that run when the page is requested and that render markup to the browser. Many Web server controls are similar to familiar HTML elements, such as buttons and text boxes. Other controls encompass complex behavior, such as a calendar controls, and controls that you can use to connect to data sources and display data.
- **Master Pages**- ASP.NET master pages allow you to create a consistent layout for the pages in your application. A single master page defines the look and feel and standard behavior that you want for all of the pages (or a group of pages) in your application. You can then create individual content pages that contain the content you want to display. When users request the content pages, they merge with the master page to produce output that combines the layout of the master page with the content from the content page.
- **Working with Data**- ASP.NET provides many options for storing, retrieving, and displaying data. In an ASP.NET Web Forms application, you use data-bound controls to automate the presentation or input of data in web page UI elements such as tables and text boxes and drop-down lists.
- **Membership**- ASP.NET Identity stores your users' credentials in a database created by the application. When your users log in, the application validates their credentials by reading the database. Your project's *Account* folder contains the files that implement the various parts of membership: registering, logging in, changing a password, and authorizing access. Additionally, ASP.NET Web Forms supports OAuth and OpenID. These authentication enhancements allow users to log into your site using existing credentials, from such accounts as Facebook, Twitter, Windows Live, and Google. By default, the template creates a membership database using a default database name on an instance of SQL Server Express LocalDB, the development database server that comes with Visual Studio Express 2013 for Web.
- **Client Script and Client Frameworks**- You can enhance the server-based features of ASP.NET by including client-script functionality in ASP.NET Web Form pages. You can use client script to provide a richer, more responsive user interface to users. You can also use client script to make asynchronous calls to the Web server while a page is running in the browser.
- **Routing**- URL routing allows you to configure an application to accept request URLs that do not map to physical files. A request URL is simply the URL a user enters into their browser to find a page on your web site. You use routing to define URLs that are semantically meaningful to users and that can help with search-engine optimization (SEO).
- **State Management**- ASP.NET Web Forms includes several options that help you preserve data on both a per-page basis and an application-wide basis.
- **Security**- An important part of developing a more secure application is to understand the threats to it. Microsoft has developed a way to categorize threats: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege (STRIDE). In ASP.NET Web Forms, you can add extensibility points and configuration options that enable you to customize various security behaviors in ASP.NET Web Forms.
- **Performance**- Performance can be a key factor in a successful Web site or project. ASP.NET Web Forms allows you to modify performance related to page and server control processing, state management, data access, application configuration and loading, and efficient coding practices.
- **Internationalization**- ASP.NET Web Forms enables you to create web pages that can obtain content and other data based on the preferred language setting for the browser or based on the user's explicit choice of language. Content and other data is referred to as resources and such data can be stored in resource files or other sources. In an ASP.NET Web Forms page, you configure controls to get their property values from resources. At run time, the resource expressions are replaced by resources from the appropriate localized resource file.
- **Debugging and Error Handling**- ASP.NET includes features to help you diagnose problems that might arise in your Web Forms application. Debugging and error handling are well supported within ASP.NET Web Forms so that your applications compile and run effectively.
- **Deployment and Hosting**- Visual Studio, ASP.NET, Azure, and IIS provide tools that help you with the process of deploying and hosting your Web Forms application.

Deciding When to Create a Web Forms Application

You must consider carefully whether to implement a Web application by using either the ASP.NET Web Forms model or another model, such as the ASP.NET MVC framework. The MVC framework does not replace the Web Forms model; you can use either framework for Web applications. Before you decide to use the Web Forms model or the MVC framework for a specific Web site, weigh the advantages of each approach.

Advantages of a Web Forms-Based Web Application

The Web Forms-based framework offers the following advantages:

- It supports an event model that preserves state over HTTP, which benefits line-of-business Web application development. The Web Forms-based application provides dozens of events that are supported in hundreds of server controls.
- It uses a Page Controller pattern that adds functionality to individual pages. For more information, see Page Controller on the MSDN Web site.
- It uses view state or server-based forms, which can make managing state information easier.
- It works well for small teams of Web developers and designers who want to take advantage of the large number of components available for rapid application development.
- In general, it is less complex for application development, because the components (the **Page** class, controls, and so on) are tightly integrated and usually require less code than the MVC model.

Advantages of an MVC-Based Web Application

The ASP.NET MVC framework offers the following advantages:

- It makes it easier to manage complexity by dividing an application into the model, the view, and the controller.
- It does not use view state or server-based forms. This makes the MVC framework ideal for developers who want full control over the behavior of an application.
- It uses a Front Controller pattern that processes Web application requests through a single controller. This enables you to design an application that supports a rich routing infrastructure. For more information, see Front Controller on the MSDN Web site.
- It provides better support for test-driven development (TDD).
- It works well for Web applications that are supported by large teams of developers and Web designers who need a high degree of control over the application behavior.

Web Forms Video Training from Pluralsight

2/20/2020 • 2 minutes to read • Edit Online

Videos

- Introduction to ASP.NET
- ASP.NET Web Forms Features
- Working with Data
- Entity Framework
- Master Pages
- Javascript and Ajax
- Security
- Using ASP.NET Dynamic Data
- Web Deployment

"This course, recorded by Dan Wahlin for Pluralsight, is a great introduction to ASP.NET Web Forms. His thorough coverage begins with an introduction to the .NET Framework, continues with how to install and use Visual Studio, walks through how to build applications (including server controls, data access, and Ajax), and finishes with deployment techniques. Dan's a true ASP.NET expert and a great presenter, so don't miss this free course!"

Scott Hanselman, Principal Program Manager, Web Platform and Azure Tools, Microsoft

Unlock access to 3,000+ dev, IT and creative courses that you can watch anytime, anywhere.

Starting at $ 29/mo.Start Free TrialSubscribe Now

ASP.NET MVC

2/20/2020 • 2 minutes to read • Edit Online

- Guidance
- Videos
- ASP.NET MVC 5
- ASP.NET MVC 4
- ASP.NET MVC 3
- MVC Video Training from Pluralsight

ASP.NET MVC Guidance

2/20/2020 • 2 minutes to read • Edit Online

- Getting Started
- UI, Layouts, and Themes
- Security, Authentication, and Authorization
- Deployment
- Performance
- Internationalization and Advanced Topics
- Releases
- Older Versions - MVC 3 and 4
- Older Versions - MVC 1 and 2

ASP.NET MVC Getting Started

2/20/2020 • 2 minutes to read • Edit Online

The following guidance explains what ASP.NET MVC is and shows how to get started.

- Getting Started with ASP.NET MVC 5
  - Getting Started
  - Adding a Controller
  - Adding a View
  - Adding a Model
  - Creating a Connection String and Working with SQL Server LocalDB
  - Accessing Your Model's Data from a Controller
  - Examining the Edit Methods and Edit View
  - Adding Search
  - Adding a New Field
  - Adding Validation
  - Examining the Details and Delete Methods
- Getting Started with EF 6 using MVC 5
  - Creating an Entity Framework Data Model
  - Implementing Basic CRUD Functionality
  - Sorting, Filtering, and Paging
  - Connection Resiliency and Command Interception
  - Code First Migrations and Deployment
  - Creating a More Complex Data Model
  - Reading Related Data
  - Updating Related Data
  - Async and Stored Procedures
  - Handling Concurrency
  - Implementing Inheritance
  - Advanced Entity Framework Scenarios
- EF Database First with ASP.NET MVC
  - Getting Started with Entity Framework 6 Database First using MVC 5
  - Creating the Web Application and Data Models
  - Generating Views
  - Changing the Database
  - Customizing a View
  - Enhancing Data Validation
- Lifecycle of an ASP.NET MVC 5 Application
- MVC Recommended Resources
- MVC recommended tutorials and articles

Getting Started with ASP.NET MVC 5

9/16/2020 • 2 minutes to read • Edit Online

Introduction to ASP.NET MVC 5

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

This following tutorial series covers ASP.NET MVC: Source located on GitHub

- Getting Started
- Adding a Controller
- Adding a View
- Adding a Model
- Creating a Connection String and Working with SQL Server LocalDB
- Accessing Your Model's Data from a Controller
- Examining the Edit Methods and Edit View
- Adding Search
- Adding a New Field
- Adding Validation
- Examining the Details and Delete Methods

Getting started with ASP.NET MVC 5

2/21/2020 • 3 minutes to read • Edit Online

by Rick Anderson

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

This tutorial teaches you the basics of building an ASP.NET MVC 5 web app using Visual Studio 2017. The final source code for the tutorial is located on GitHub.

This tutorial was written by Scott Guthrie (twitter@scottgu), Scott Hanselman (twitter: @shanselman), and Rick Anderson (@RickAndMSFT)

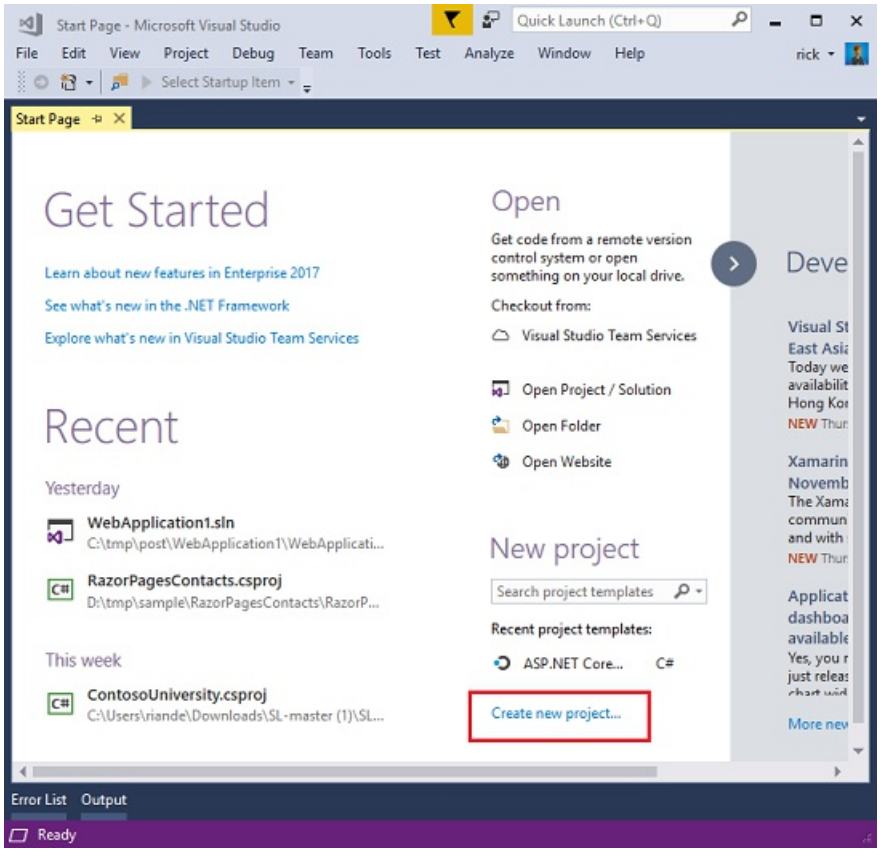
You need an Azure account to deploy this app to Azure:

- You can open an Azure account for free - You get credits you can use to try out paid Azure services, and even after they're used up you can keep the account and use free Azure services.
- You can activate MSDN subscriber benefits - Your MSDN subscription gives you credits every month that you can use for paid Azure services.

Get started

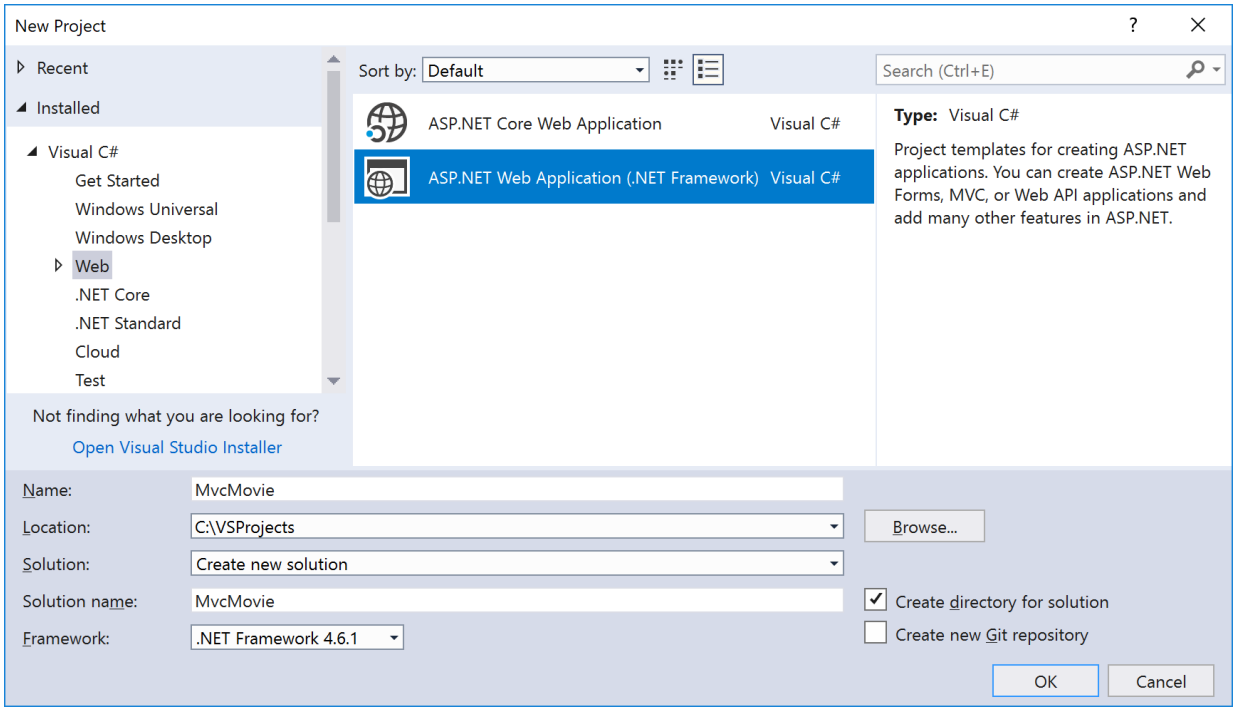
Start by installing Visual Studio 2017. Then, open Visual Studio.

Visual Studio is an IDE, or integrated development environment. Just like you use Microsoft Word to write documents, you'll use an IDE to create applications. In Visual Studio, there's a list along the bottom showing various options available to you. There's also a menu that provides another way to perform tasks in the IDE. For example, instead of selecting **New Project** on the **Start page**, you can use the menu bar and select **File** > **New Project**.

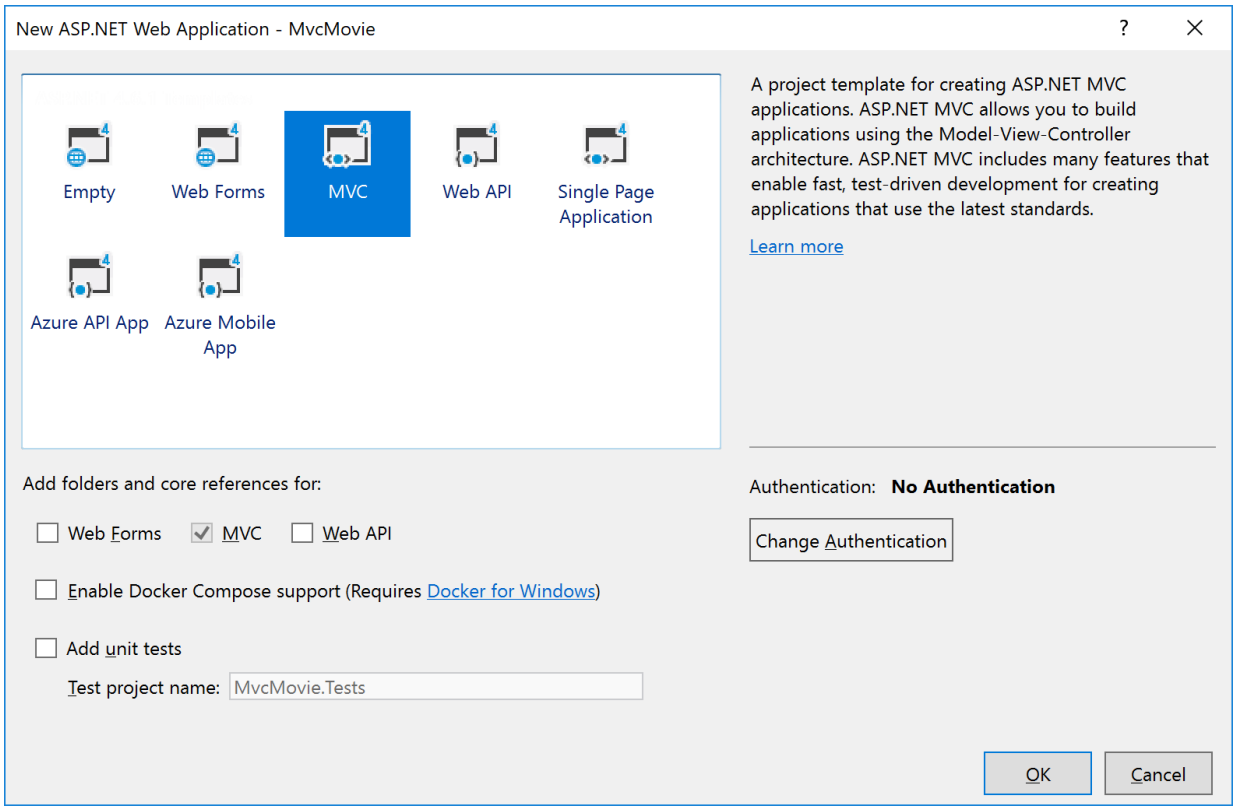


Create your first app

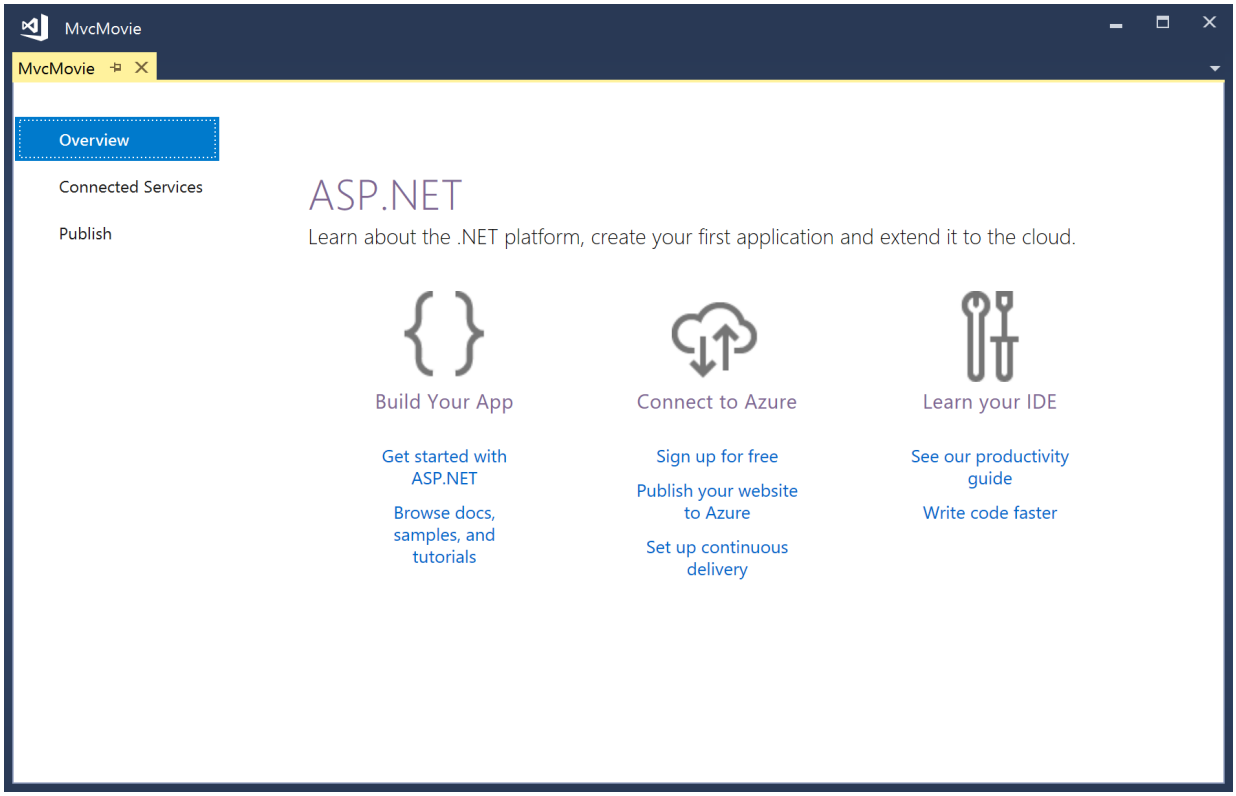
On the **Start page**, select **New Project**. In the **New project** dialog box, select the **Visual C#** category on the left, then **Web**, and then select the **ASP.NET Web Application (.NET Framework)** project template. Name your project "MvcMovie" and then choose **OK**.



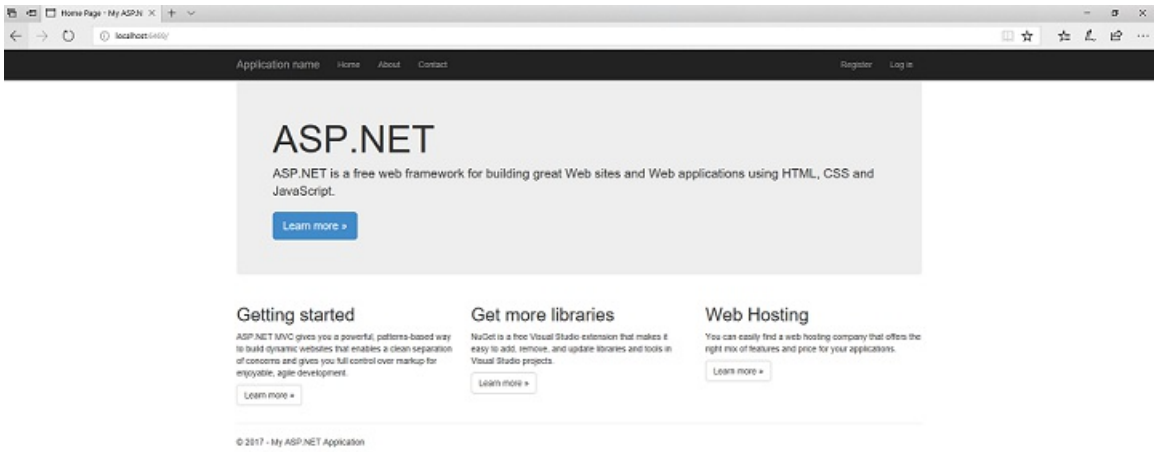
In the **New ASP.NET Web Application** dialog, choose **MVC** and then choose **OK**.



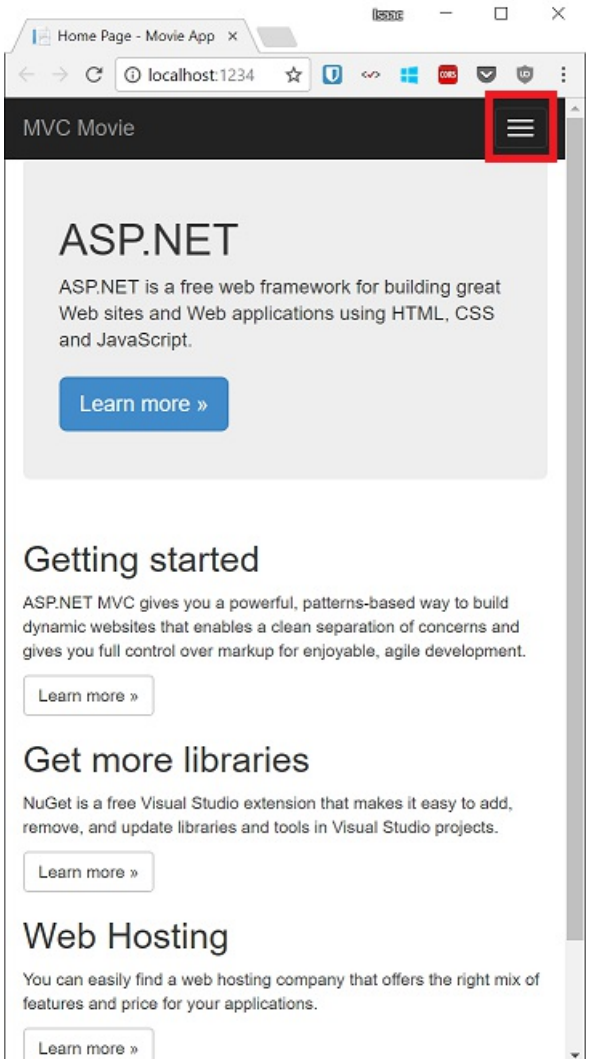
Visual Studio used a default template for the ASP.NET MVC project you just created, so you have a working application right now without doing anything! This is a simple "Hello World!" project, and it's a good place to start your application.



Press **F5** to start debugging. When you press **F5**, Visual Studio starts IIS Express and runs your web app. Visual Studio then launches a browser and opens the application's home page. Notice that the address bar of the browser says `localhost:port#` and not something like `example.com`. That's because `localhost` always points to your own local computer, which in this case is running the application you just built. When Visual Studio runs a web project, a random port is used for the web server. In the image below, the port number is 1234. When you run the application, you'll see a different port number.



Right out of the box this default template gives you `Home`, `Contact`, and `About` pages. The image below doesn't show the **Home**, **About**, and **Contact** links. Depending on the size of your browser window, you might need to click the navigation icon to see these links.



The application also provides support to register and log in. The next step is to change how this application works and learn a little bit about ASP.NET MVC. Close the ASP.NET MVC application and let's change some code.

For a list of current tutorials, see MVC recommended articles.

See this app running on Azure

Would you like to see the finished site running as a live web app? You can deploy a complete version of the app to your Azure account by simply clicking the following button.

Deploy to Azure

You need an Azure account to deploy this solution to Azure. If you don't already have an account, use one of the following options to create one:

- Open an Azure account for free - You get credits you can use to try out paid Azure services, and even after they're used up you can keep the account and use free Azure services.
- Activate Visual Studio subscriber benefits - Your Visual Studio subscription gives you credits every month that you can use for paid Azure services.

NEXT

Adding a Controller

9/16/2020 • 6 minutes to read • Edit Online

by Rick Anderson

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

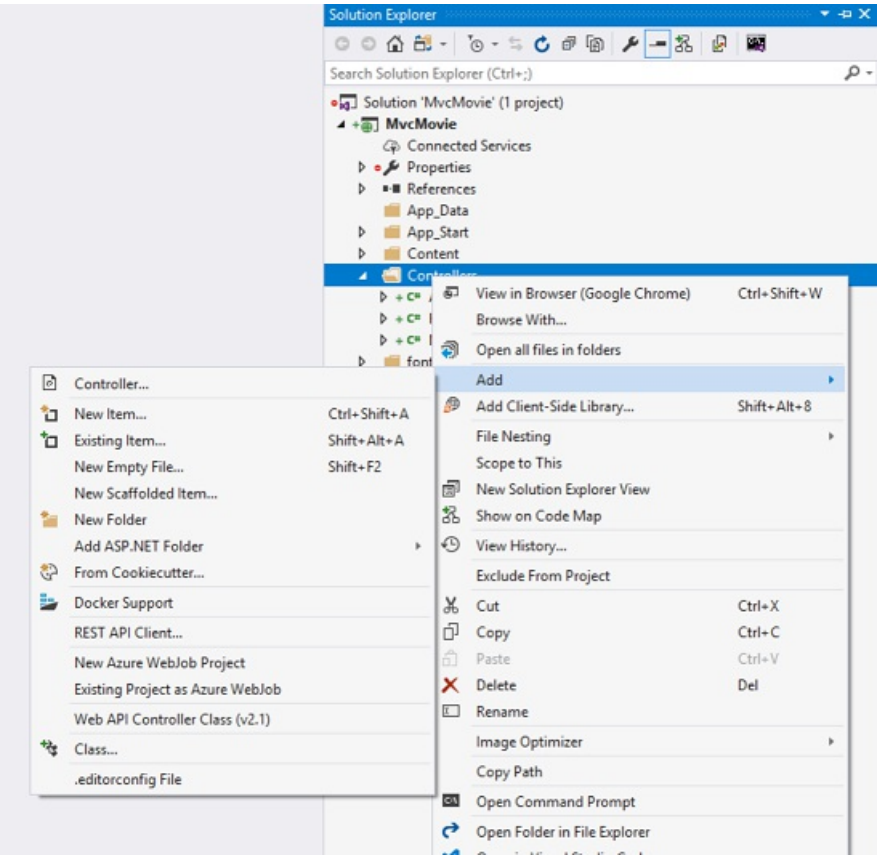
- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

MVC stands for *model-view-controller*. MVC is a pattern for developing applications that are well architected, testable and easy to maintain. MVC-based applications contain:

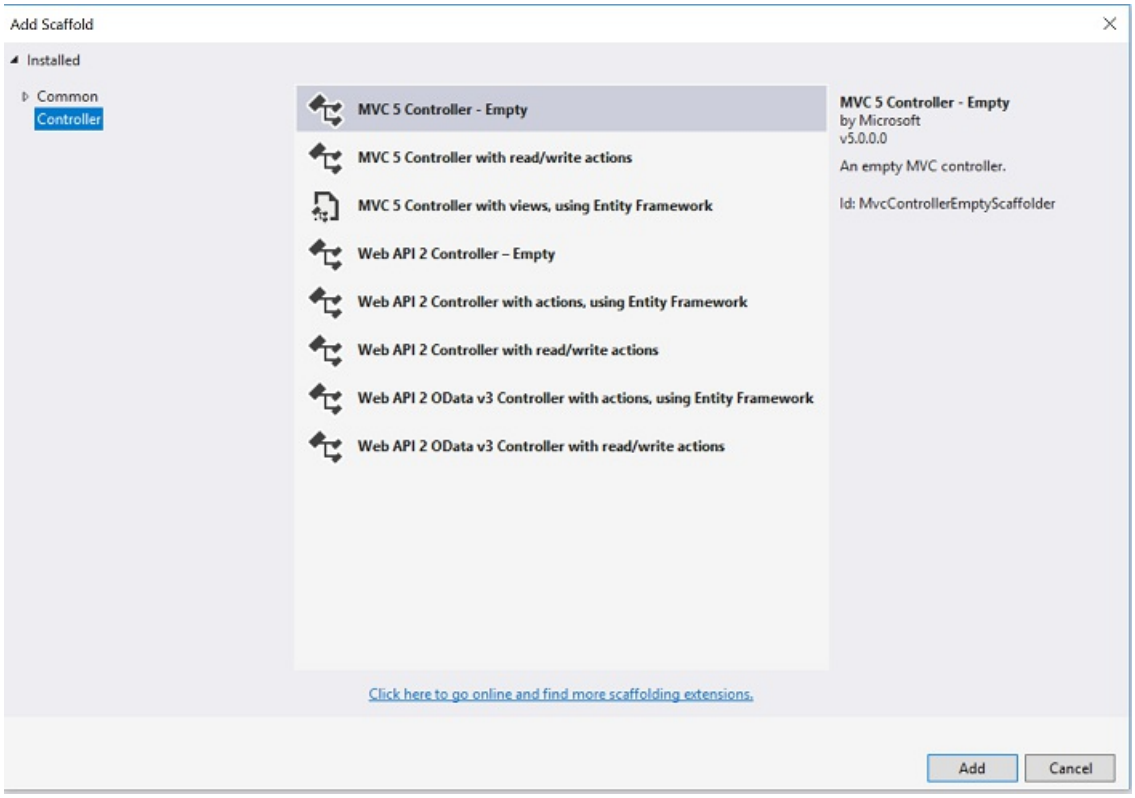
- **M** odels: Classes that represent the data of the application and that use validation logic to enforce business rules for that data.
- **V** iews: Template files that your application uses to dynamically generate HTML responses.
- **C** ontrollers: Classes that handle incoming browser requests, retrieve model data, and then specify view templates that return a response to the browser.

We'll be covering all these concepts in this tutorial series and show you how to use them to build an application.

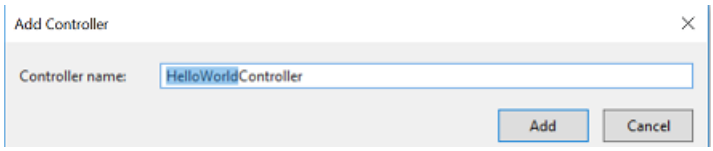
Let's begin by creating a controller class. In **Solution Explorer**, right-click the *Controllers* folder and then click **Add**, then **Controller**.



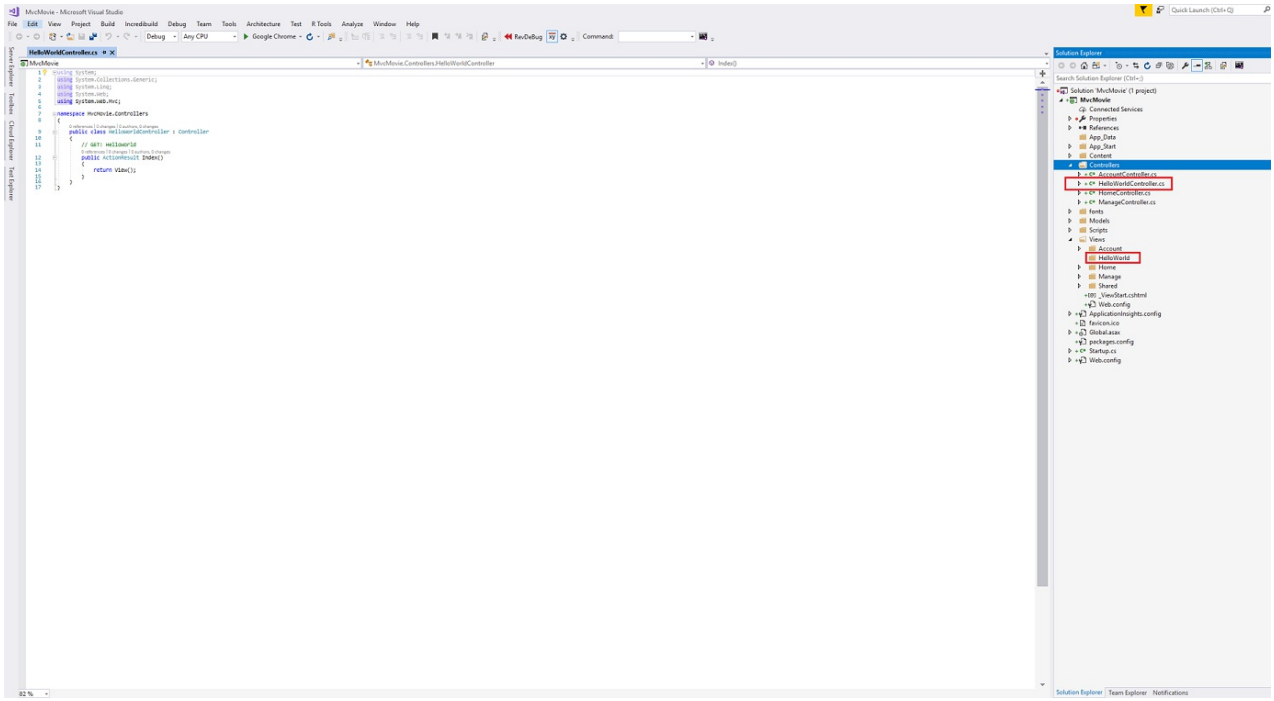
In the **Add Scaffold** dialog box, click **MVC 5 Controller - Empty**, and then click **Add**.



Name your new controller "HelloWorldController" and click **Add**.



Notice in **Solution Explorer** that a new file has been created named *HelloWorldController.cs* and a new folder *Views\HelloWorld*. The controller is open in the IDE.



Replace the contents of the file with the following code.

```
using System.Web;
using System.Web.Mvc;

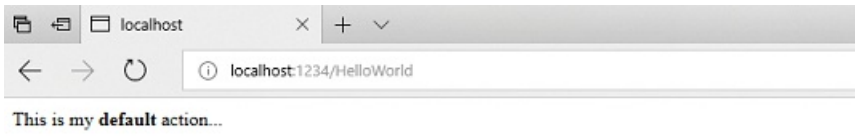
namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my <b>default</b> action...";
        }

        //
        // GET: /HelloWorld/Welcome/

        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

The controller methods will return a string of HTML as an example. The controller is named `HelloWorldController` and the first method is named `Index`. Let's invoke it from a browser. Run the application (press F5 or Ctrl+F5). In the browser, append "HelloWorld" to the path in the address bar. (For example, in the illustration below, it's `http://localhost:1234/HelloWorld.`) The page in the browser will look like the following screenshot. In the method above, the code returned a string directly. You told the system to just return some HTML, and it did!



ASP.NET MVC invokes different controller classes (and different action methods within them) depending on the incoming URL. The default URL routing logic used by ASP.NET MVC uses a format like this to determine what code to invoke:

`/[Controller]/[ActionName]/[Parameters]`

You set the format for routing in the *App_Start/RouteConfig.cs* file.

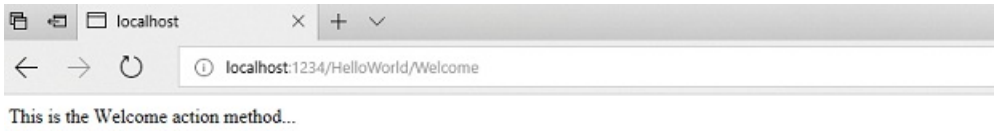
```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    );
}
```

When you run the application and don't supply any URL segments, it defaults to the "Home" controller and the "Index" action method specified in the defaults section of the code above.

The first part of the URL determines the controller class to execute. So */HelloWorld* maps to the `HelloWorldController` class. The second part of the URL determines the action method on the class to execute. So */HelloWorld/Index* would cause the `Index` method of the `HelloWorldController` class to execute. Notice that we only had to browse to */HelloWorld* and the `Index` method was used by default. This is because a method named `Index` is the default method that will be called on a controller if one is not explicitly specified. The third part of the URL segment (`Parameters`) is for route data. We'll see route data later on in this tutorial.

Browse to `http://localhost:xxxx/HelloWorld/Welcome`. The `Welcome` method runs and returns the string "This is the Welcome action method...". The default MVC mapping is `/[Controller]/[ActionName]/[Parameters]`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't used the `[Parameters]` part of the URL yet.



Let's modify the example slightly so that you can pass some parameter information from the URL to the controller (for example, */HelloWorld/Welcome?name=Scott&numtimes=4*). Change your `Welcome` method to include two parameters as shown below. Note that the code uses the C# optional-parameter feature to indicate that the `numTimes` parameter should default to 1 if no value is passed for that parameter.

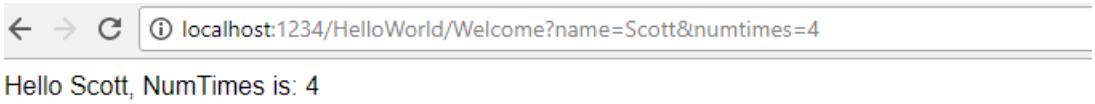
```
public string Welcome(string name, int numTimes = 1) {
     return HttpUtility.HtmlEncode("Hello " + name + ", NumTimes is: " + numTimes);
}
```

NOTE

Security Note: The code above uses HttpUtility.HtmlEncode to protect the application from malicious input (namely JavaScript). For more information see How to: Protect Against Script Exploits in a Web Application by Applying HTML Encoding to Strings.

Run your application and browse to the example URL (`http://localhost:xxxx/HelloWorld/Welcome?name=Scott&numtimes=4`). You can try different values for `name` and `numtimes` in the URL. The ASP.NET MVC model binding system automatically maps the named parameters from

the query string in the address bar to parameters in your method.



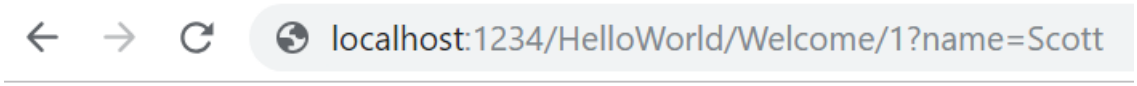
Hello Scott, NumTimes is: 4

In the sample above, the URL segment (`Parameters`) is not used, the `name` and `numTimes` parameters are passed as query strings. The ? (question mark) in the above URL is a separator, and the query strings follow. The & character separates query strings.

Replace the Welcome method with the following code:

```
public string Welcome(string name, int ID = 1)
{
    return HttpUtility.HtmlEncode("Hello " + name + ", ID: " + ID);
}
```

Run the application and enter the following URL: `http://localhost:xxx/HelloWorld/Welcome/1?name=Scott`



Hello Scott, ID: 1

This time the third URL segment matched the route parameter `ID.` The `Welcome` action method contains a parameter (`ID`) that matched the URL specification in the `RegisterRoutes` method.

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    );
}
```

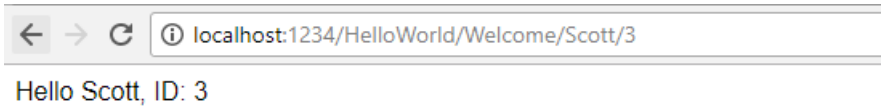
In ASP.NET MVC applications, it's more typical to pass in parameters as route data (like we did with ID above) than passing them as query strings. You could also add a route to pass both the `name` and `numtimes` in parameters as route data in the URL. In the *App_Start\RouteConfig.cs* file, add the "Hello" route:

```
public class RouteConfig
{
   public static void RegisterRoutes(RouteCollection routes)
   {
      routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

      routes.MapRoute(
          name: "Default",
          url: "{controller}/{action}/{id}",
          defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
      );

      routes.MapRoute(
           name: "Hello",
           url: "{controller}/{action}/{name}/{id}"
       );
   }
}
```

Run the application and browse to `/localhost:XXX/HelloWorld/Welcome/Scott/3`.



For many MVC applications, the default route works fine. You'll learn later in this tutorial to pass data using the model binder, and you won't have to modify the default route for that.

In these examples the controller has been doing the "VC" portion of MVC — that is, the view and controller work. The controller is returning HTML directly. Ordinarily you don't want controllers returning HTML directly, since that becomes very cumbersome to code. Instead we'll typically use a separate view template file to help generate the HTML response. Let's look next at how we can do this.

PREVIOUS NEXT

Adding a View

9/16/2020 • 9 minutes to read • Edit Online

by Rick Anderson

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

In this section you're going to modify the `HelloWorldController` class to use view template files to cleanly encapsulate the process of generating HTML responses to a client.

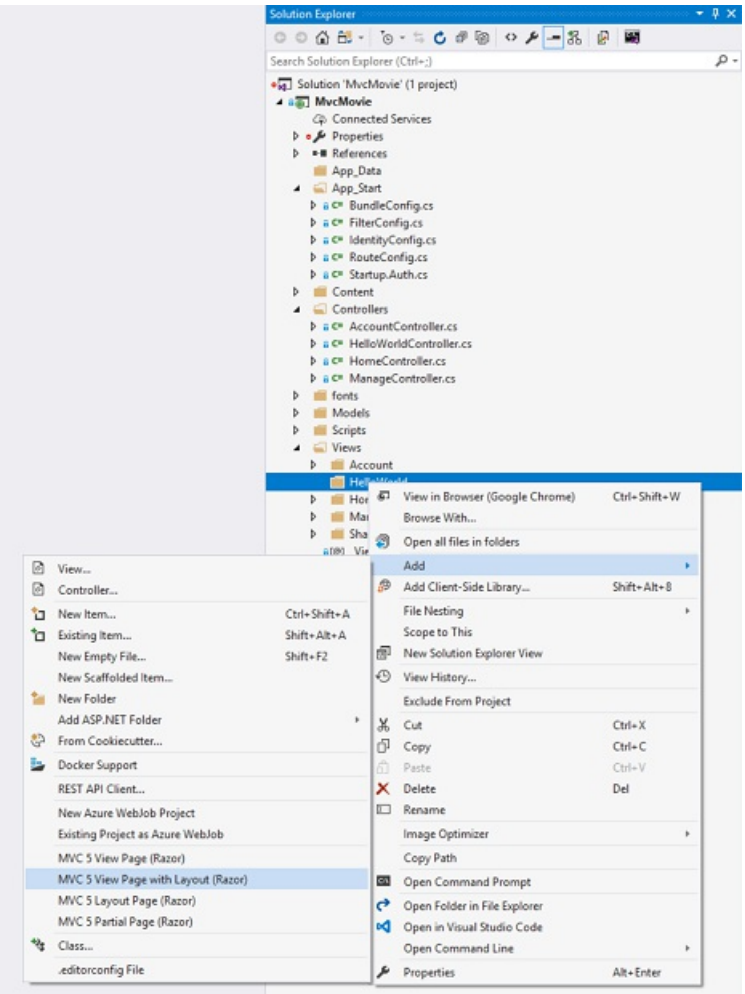
You'll create a view template file using the Razor view engine. Razor-based view templates have a *.cshtml* file extension, and provide an elegant way to create HTML output using C#. Razor minimizes the number of characters and keystrokes required when writing a view template, and enables a fast, fluid coding workflow.

Currently the `Index` method returns a string with a message that is hard-coded in the controller class. Change the `Index` method to call the controllers View method, as shown in the following code:

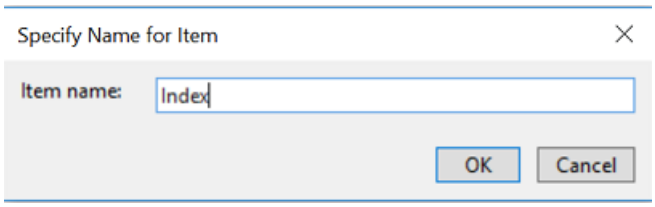
```
public ActionResult Index()
{
    return View();
}
```

The `Index` method above uses a view template to generate an HTML response to the browser. Controller methods (also known as action methods), such as the `Index` method above, generally return an ActionResult (or a class derived from ActionResult), not primitive types like string.

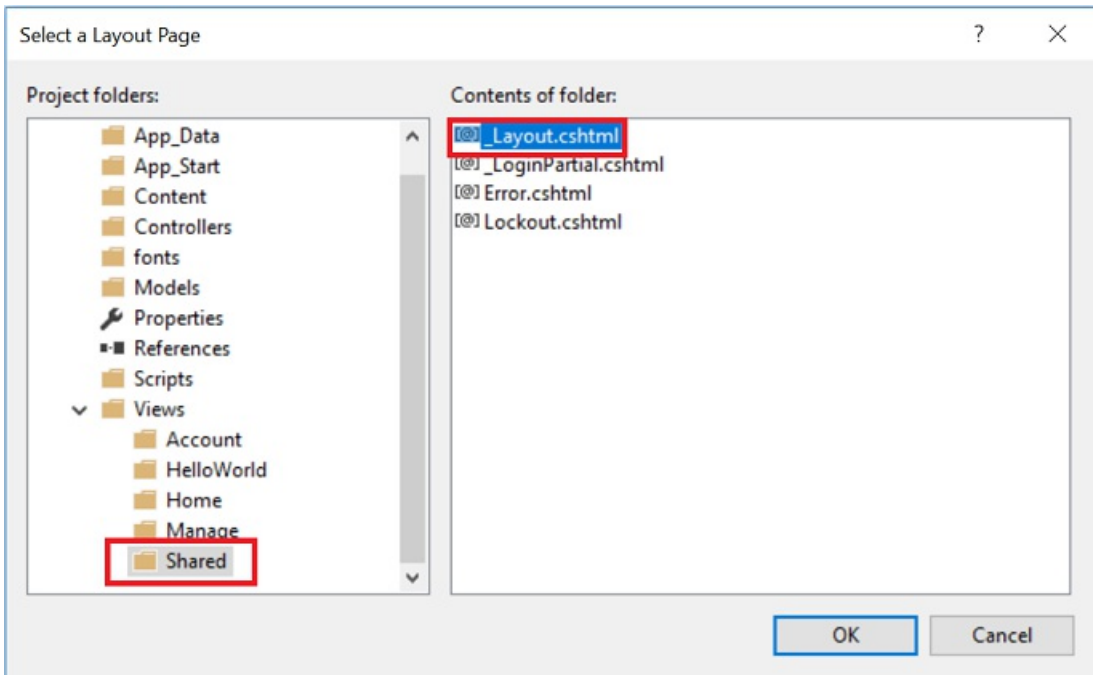
Right click the *Views\HelloWorld* folder and click **Add**, then click **MVC 5 View Page with Layout (Razor)**.



In the **Specify Name for Item** dialog box, enter *Index*, and then click **OK**.

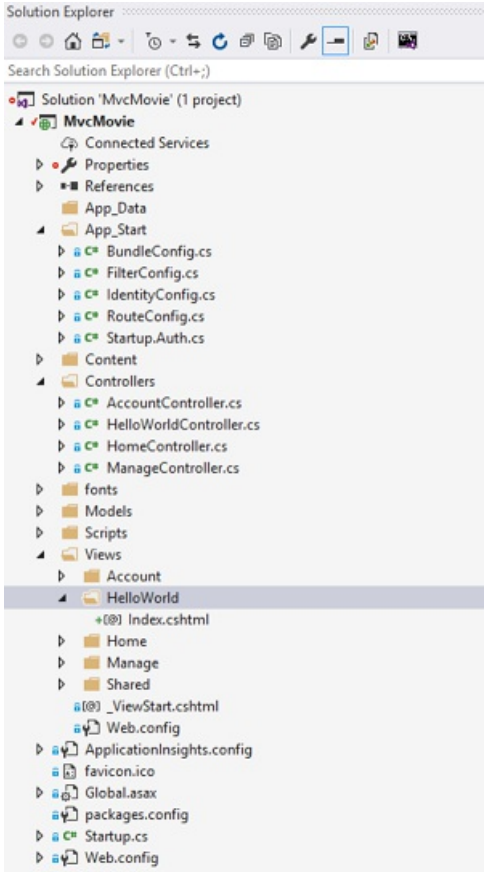


In the **Select a Layout Page** dialog, accept the default **_Layout.cshtml** and click **OK**.



In the dialog above, the *Views\Shared* folder is selected in the left pane. If you had a custom layout file in another folder, you could select it. We'll talk about the layout file later in the tutorial

The *MvcMovie\Views\HelloWorld\Index.cshtml* file is created.



Add the following highlighted markup.

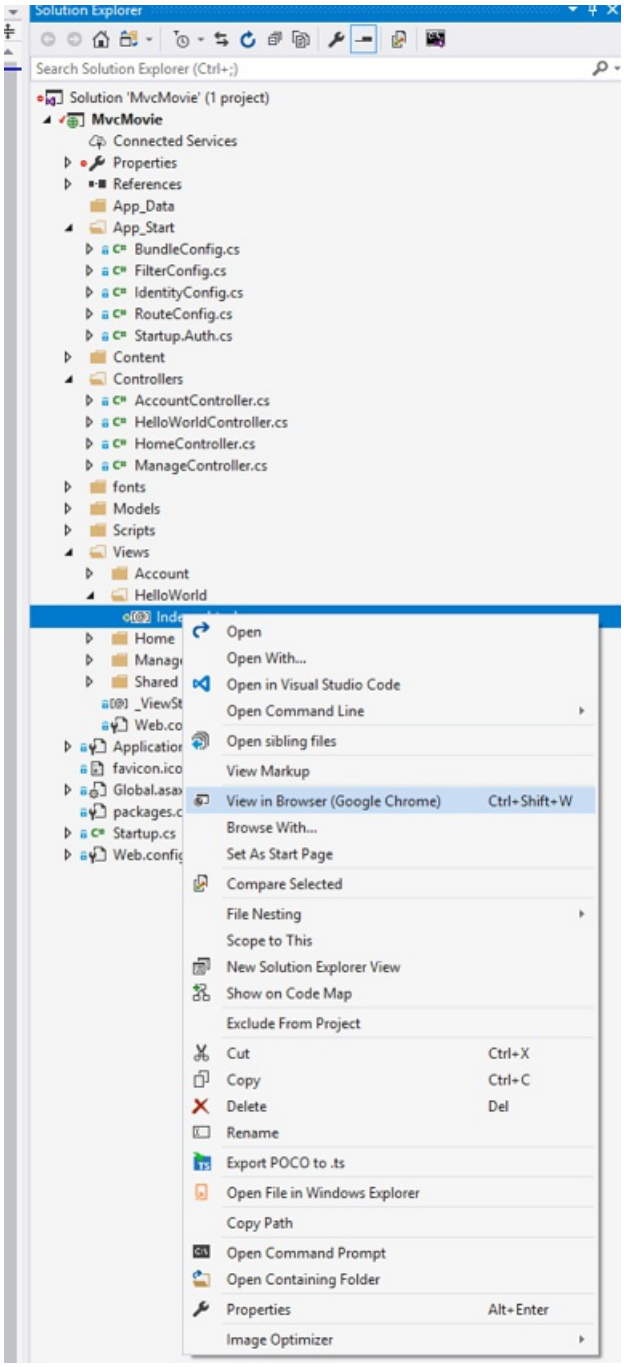
```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

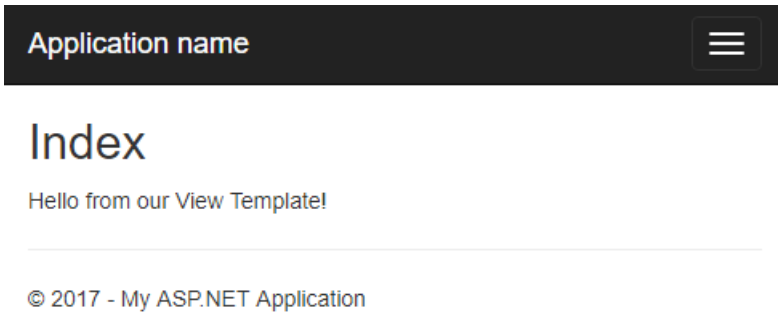
<p>Hello from our View Template!</p>
```

Right click the *Index.cshtml* file and select **View in Browser**.



You can also right click the *Index.cshtml* file and select **View in Page Inspector.** See the Page Inspector tutorial for more information.

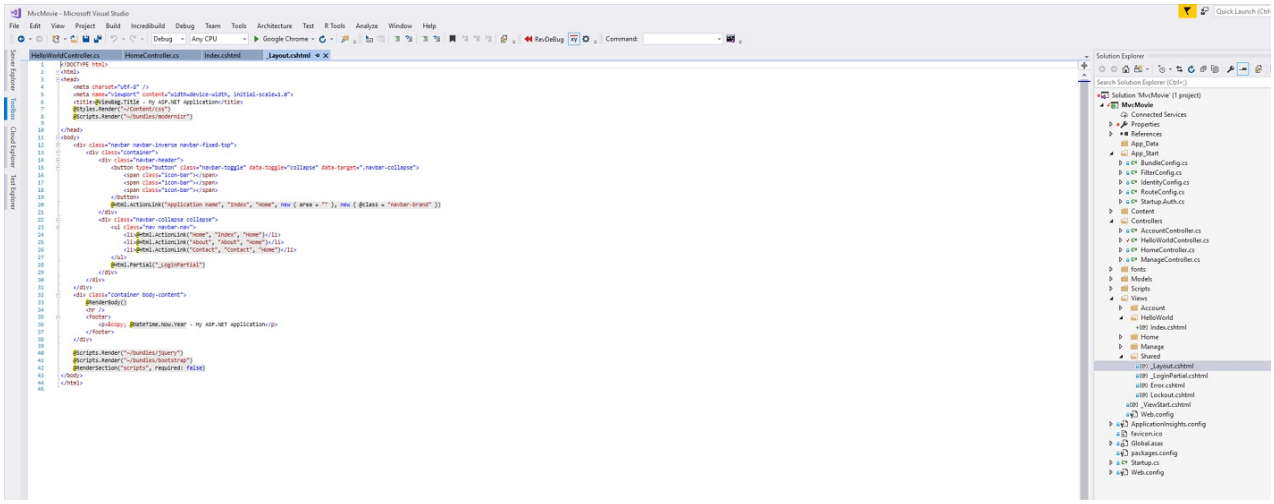
Alternatively, run the application and browse to the `HelloWorld` controller (`http://localhost:xxxx/HelloWorld`). The `Index` method in your controller didn't do much work; it simply ran the statement `return View()`, which specified that the method should use a view template file to render a response to the browser. Because you didn't explicitly specify the name of the view template file to use, ASP.NET MVC defaulted to using the *Index.cshtml* view file in the *\Views\HelloWorld* folder. The image below shows the string "Hello from our View Template!" hard-coded in the view.



Looks pretty good. However, notice that the browser's title bar shows "Index - My ASP.NET Application," and the big link at the top of the page says "Application name." Depending on how small you make your browser window, you might need to click the three bars in the upper right to see the to the **Home**, **About**, **Contact**, **Register** and **Log in** links.

Changing Views and Layout Pages

First, you want to change the "Application name" link at the top of the page. That text is common to every page. It's actually implemented in only one place in the project, even though it appears on every page in the application. Go to the */Views/Shared* folder in **Solution Explorer** and open the *_Layout.cshtml* file. This file is called a *layout page* and it's in the shared folder that all other pages use.



Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, "wrapped" in the layout page. For example, if you select the **About** link, the *Views\Home\About.cshtml* view is rendered inside the `RenderBody` method.

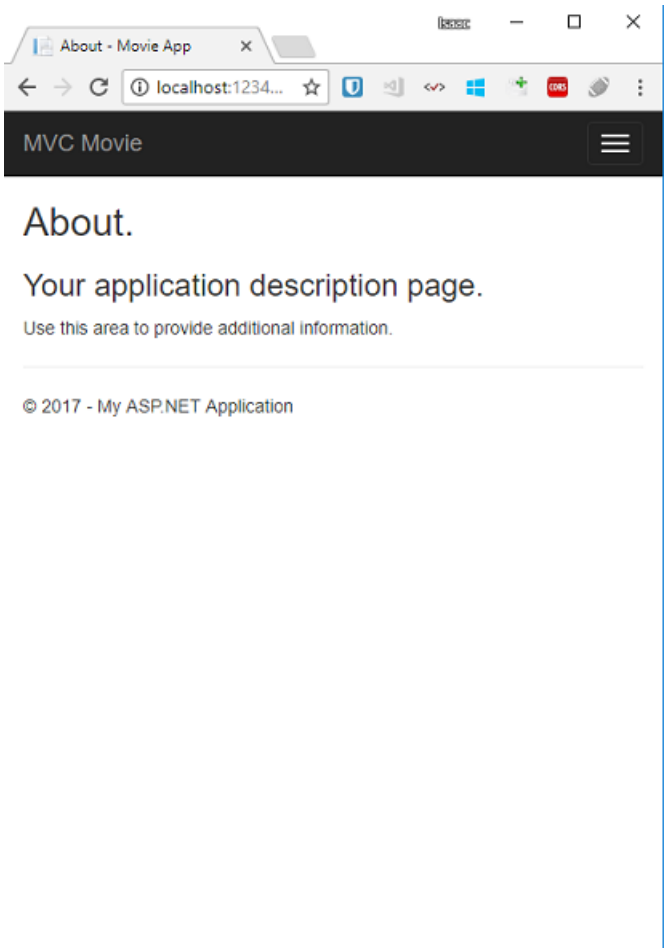
Change the contents of the title element. Change the ActionLink in the layout template from "Application name" to "MVC Movie" and the controller from `Home` to `Movies`. The complete layout file is shown below:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - Movie App</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")

</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("MVC Movie", "Index", "Movies", null, new { @class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>

    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/bootstrap")
    @RenderSection("scripts", required: false)
</body>
</html>
```

Run the application and notice that it now says "MVC Movie ". Click the **About** link, and you see how that page shows "MVC Movie", too. We were able to make the change once in the layout template and have all pages on the site reflect the new title.



When we first created the *Views\HelloWorld\Index.cshtml* file, it contained the following code:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

The Razor code above is explicitly setting the layout page. Examine the *Views_ViewStart.cshtml* file, it contains the exact same Razor markup. The *Views_ViewStart.cshtml* file defines the common layout that all views will use, therefore you can comment out or remove that code from the *Views\HelloWorld\Index.cshtml* file.

```
@*@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}*@

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

You can use the `Layout` property to set a different layout view, or set it to `null` so no layout file will be used.

Now, let's change the title of the Index view.

Open *MvcMovie\Views\HelloWorld\Index.cshtml*. There are two places to make a change: first, the text that appears in the title of the browser, and then in the secondary header (the `<h2>` element). You'll make them slightly different so you can see which bit of code changes which part of the app.

```
@{
    ViewBag.Title = "Movie List";
}

<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

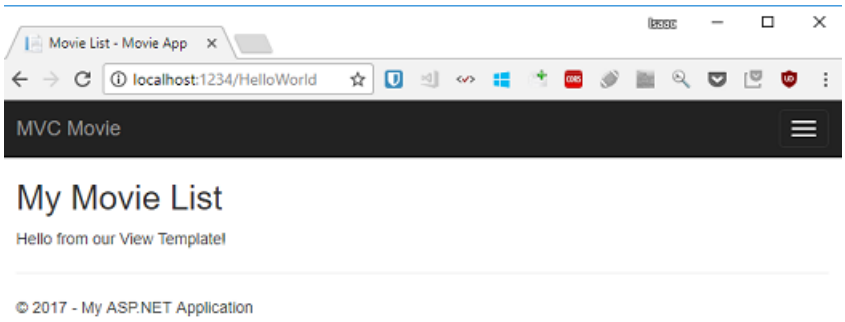
To indicate the HTML title to display, the code above sets a `Title` property of the `ViewBag` object (which is in the *Index.cshtml* view template). Notice that the layout template (*Views\Shared_Layout.cshtml*) uses this value in the `<title>` element as part of the `<head>` section of the HTML that we modified previously.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - Movie App</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
```

Using this `ViewBag` approach, you can easily pass other parameters between your view template and your layout file.

Run the application. Notice that the browser title, the primary heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing cached content. Press Ctrl+F5 in your browser to force the response from the server to be loaded.) The browser title is created with the `ViewBag.Title` we set in the *Index.cshtml* view template and the additional "- Movie App" added in the layout file.

Also notice how the content in the *Index.cshtml* view template was merged with the *_Layout.cshtml* view template and a single HTML response was sent to the browser. Layout templates make it really easy to make changes that apply across all of the pages in your application.



Our little bit of "data" (in this case the "Hello from our View Template!" message) is hard-coded, though. The MVC application has a "V" (view) and you've got a "C" (controller), but no "M" (model) yet. Shortly, we'll walk through how to create a database and retrieve model data from it.

Passing Data from the Controller to the View

Before we go to a database and talk about models, though, let's first talk about passing information from the controller to a view. Controller classes are invoked in response to an incoming URL request. A controller class is

where you write the code that handles the incoming browser requests, retrieves data from a database, and ultimately decides what type of response to send back to the browser. View templates can then be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing whatever data or objects are required in order for a view template to render a response to the browser. A best practice: **A view template should never perform business logic or interact with a database directly**. Instead, a view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep your code clean, testable and more maintainable.

Currently, the `Welcome` action method in the `HelloWorldController` class takes a `name` and a `numTimes` parameter and then outputs the values directly to the browser. Rather than have the controller render this response as a string, let's change the controller to use a view template instead. The view template will generate a dynamic response, which means that you need to pass appropriate bits of data from the controller to the view in order to generate the response. You can do this by having the controller put the dynamic data (parameters) that the view template needs in a `ViewBag` object that the view template can then access.

Return to the *HelloWorldController.cs* file and change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewBag` object. `ViewBag` is a dynamic object, which means you can put whatever you want in to it; the `ViewBag` object has no defined properties until you put something inside it. The ASP.NET MVC model binding system automatically maps the named parameters (`name` and `numTimes`) from the query string in the address bar to parameters in your method. The complete *HelloWorldController.cs* file looks like this:

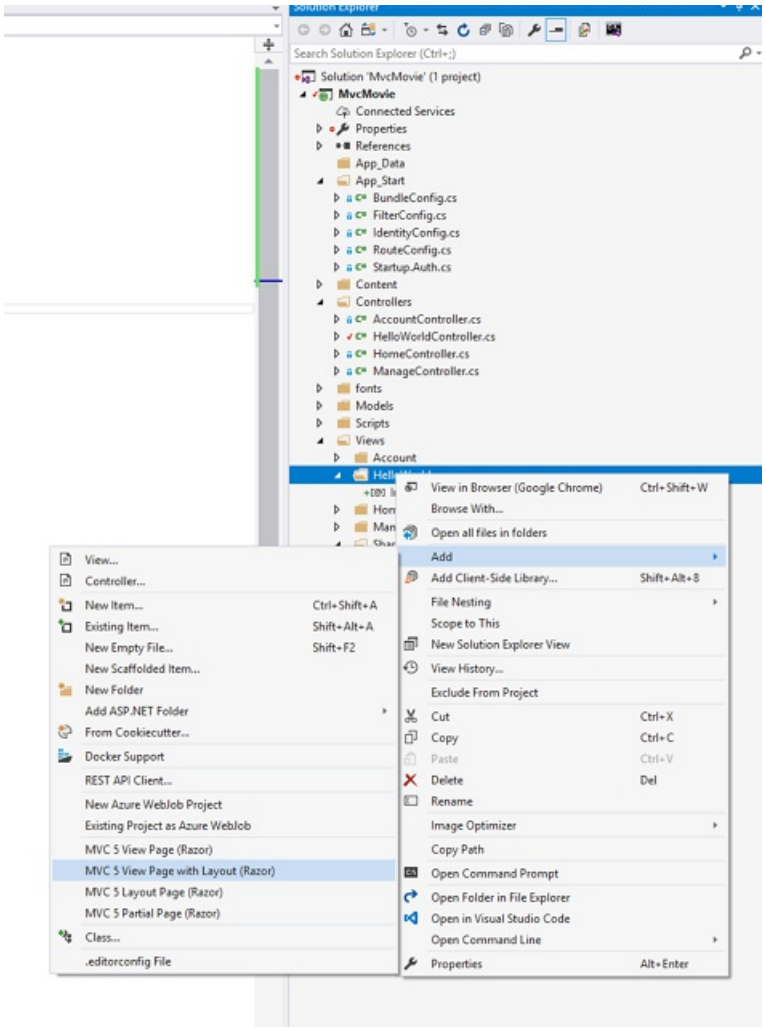
```
using System.Web;
using System.Web.Mvc;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Welcome(string name, int numTimes = 1)
        {
            ViewBag.Message = "Hello " + name;
            ViewBag.NumTimes = numTimes;

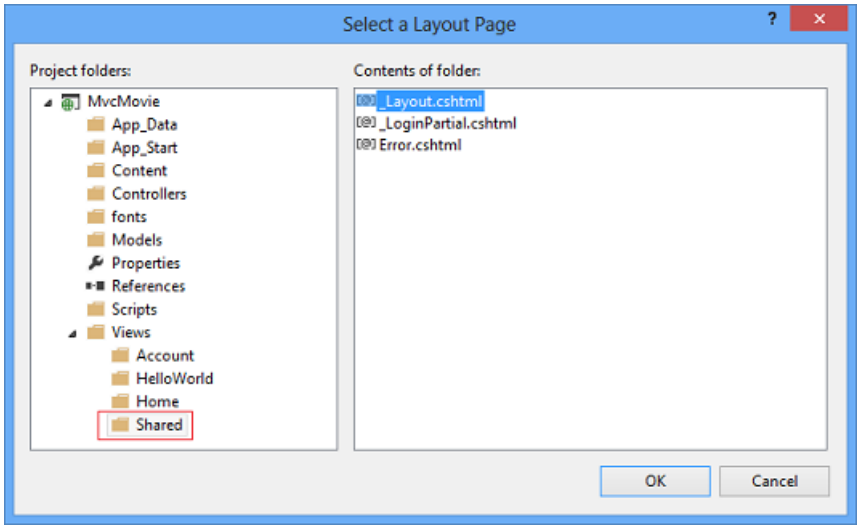
            return View();
        }
    }
}
```

Now the `ViewBag` object contains data that will be passed to the view automatically. Next, you need a Welcome view template! In the **Build** menu, select **Build Solution** (or Ctrl+Shift+B) to make sure the project is compiled. Right click the *Views\HelloWorld* folder and click **Add**, then click **MVC 5 View Page with Layout (Razor)**.



In the **Specify Name for Item** dialog box, enter *Welcome*, and then click **OK**.

In the **Select a Layout Page** dialog, accept the default **_Layout.cshtml** and click **OK**.



The *MvcMovie\Views\HelloWorld\Welcome.cshtml* file is created.

Replace the markup in the *Welcome.cshtml* file. You'll create a loop that says "Hello" as many times as the user says it should. The complete *Welcome.cshtml* file is shown below.

```
@{
    ViewBag.Title = "Welcome";
}

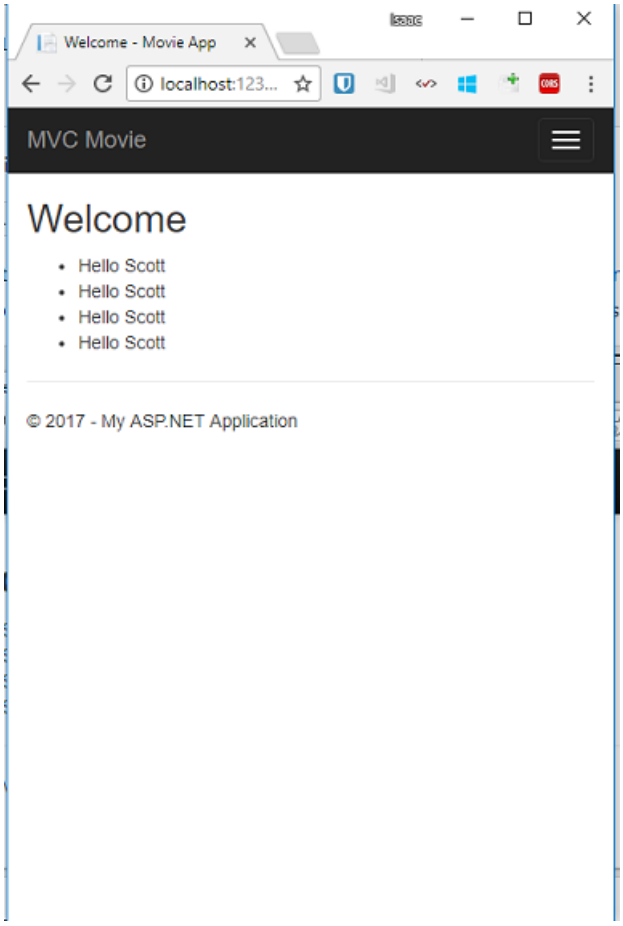
<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < ViewBag.NumTimes; i++)
    {
        <li>@ViewBag.Message</li>
    }
</ul>
```

Run the application and browse to the following URL:

`http://localhost:xx/HelloWorld/Welcome?name=Scott&numtimes=4`

Now data is taken from the URL and passed to the controller using the model binder. The controller packages the data into a `ViewBag` object and passes that object to the view. The view then displays the data as HTML to the user.



In the sample above, we used a `ViewBag` object to pass data from the controller to a view. Later in the tutorial, we will use a view model to pass data from a controller to a view. The view model approach to passing data is generally much preferred over the view bag approach. See the blog entry Dynamic V Strongly Typed Views for more information.

Well, that was a kind of an "M" for model, but not the database kind. Let's take what we've learned and create a database of movies.

PREVIOUS NEXT

Adding a Model

9/16/2020 • 2 minutes to read • Edit Online

by Rick Anderson

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

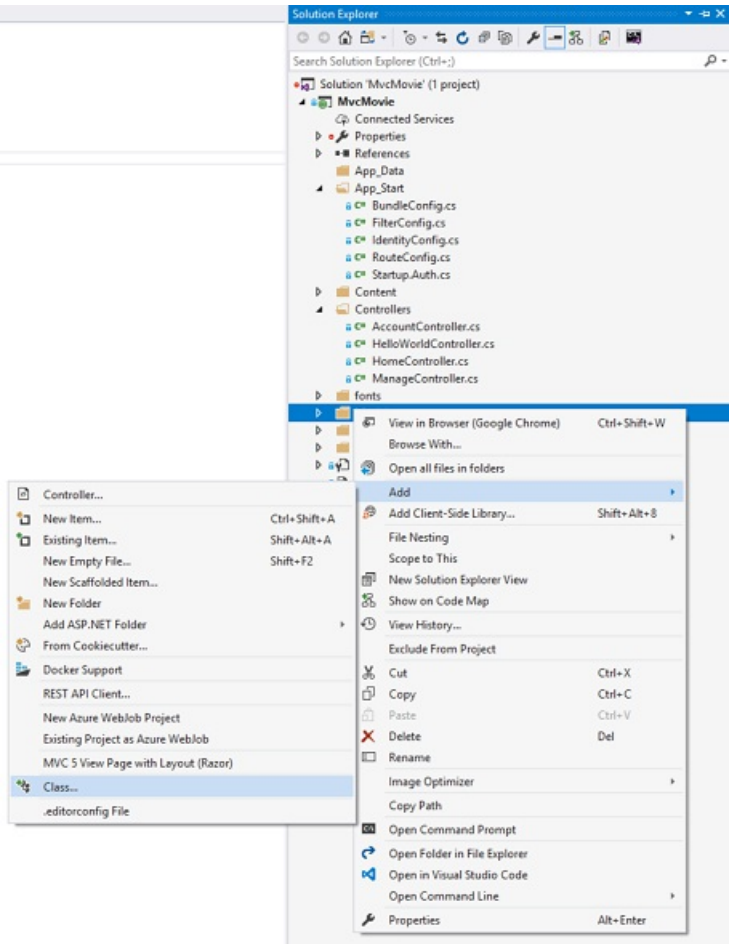
- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

In this section you'll add some classes for managing movies in a database. These classes will be the "model" part of the ASP.NET MVC app.

You'll use a .NET Framework data-access technology known as the Entity Framework to define and work with these model classes. The Entity Framework (often referred to as EF) supports a development paradigm called *Code First*. Code First allows you to create model objects by writing simple classes. (These are also known as POCO classes, from "plain-old CLR objects.") You can then have the database created on the fly from your classes, which enables a very clean and rapid development workflow. If you are required to create the database first, you can still follow this tutorial to learn about MVC and EF app development. You can then follow Tom Fizmakens ASP.NET Scaffolding tutorial, which covers the database first approach.

Adding Model Classes

In **Solution Explorer**, right click the *Models* folder, select **Add**, and then select **Class**.



Enter the *class* name "Movie".

Add the following five properties to the `Movie` class:

```
using System;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

We'll use the `Movie` class to represent movies in a database. Each instance of a `Movie` object will correspond to a row within a database table, and each property of the `Movie` class will map to a column in the table.

Note: In order to use System.Data.Entity, and the related class, you need to install the Entity Framework NuGet Package. Follow the link for further instructions.

In the same file, add the following `MovieDBContext` class:

```
using System;
using System.Data.Entity;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }

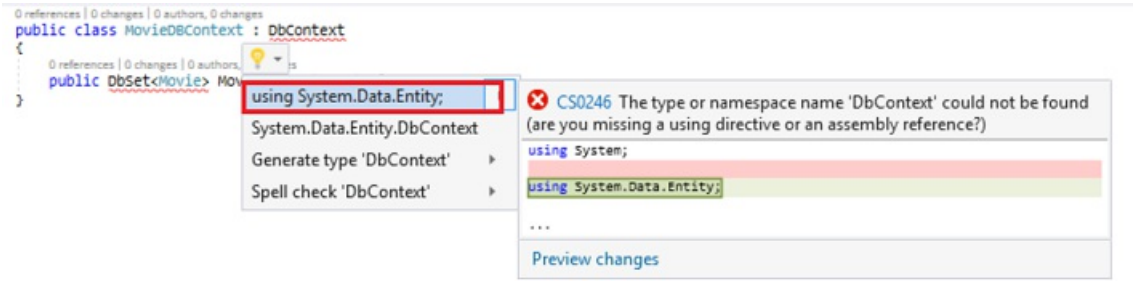
    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

The `MovieDBContext` class represents the Entity Framework movie database context, which handles fetching, storing, and updating `Movie` class instances in a database. The `MovieDBContext` derives from the `DbContext` base class provided by the Entity Framework.

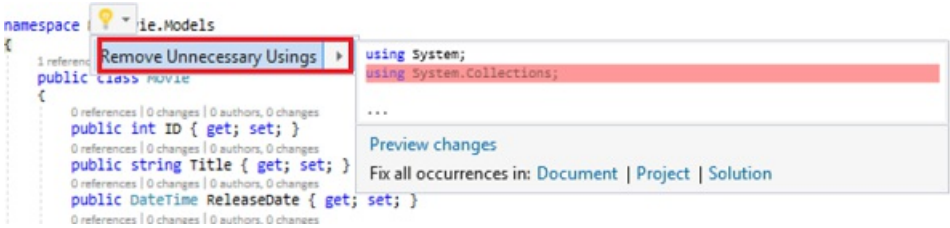
In order to be able to reference `DbContext` and `DbSet`, you need to add the following `using` statement at the top of the file:

```
using System.Data.Entity;
```

You can do this by manually adding the using statement, or you can hover over the red squiggly lines, click `Show potential fixes` and click `using System.Data.Entity;`



Note: Several unused `using` statements have been removed. Visual Studio will show unused dependencies as gray. You can remove unused dependencies by hovering over the gray dependencies, click `Show potential fixes` and click **Remove Unused Usings**.



We've finally added a model (the M in MVC). In the next section you'll work with the database connection string.

Creating a Connection String and Working with SQL Server LocalDB

9/16/2020 • 3 minutes to read • Edit Online

by Rick Anderson

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

Creating a Connection String and Working with SQL Server LocalDB

The `MovieDBContext` class you created handles the task of connecting to the database and mapping `Movie` objects to database records. One question you might ask, though, is how to specify which database it will connect to. You don't actually have to specify which database to use, Entity Framework will default to using LocalDB. In this section we'll explicitly add a connection string in the *Web.config* file of the application.

SQL Server Express LocalDB

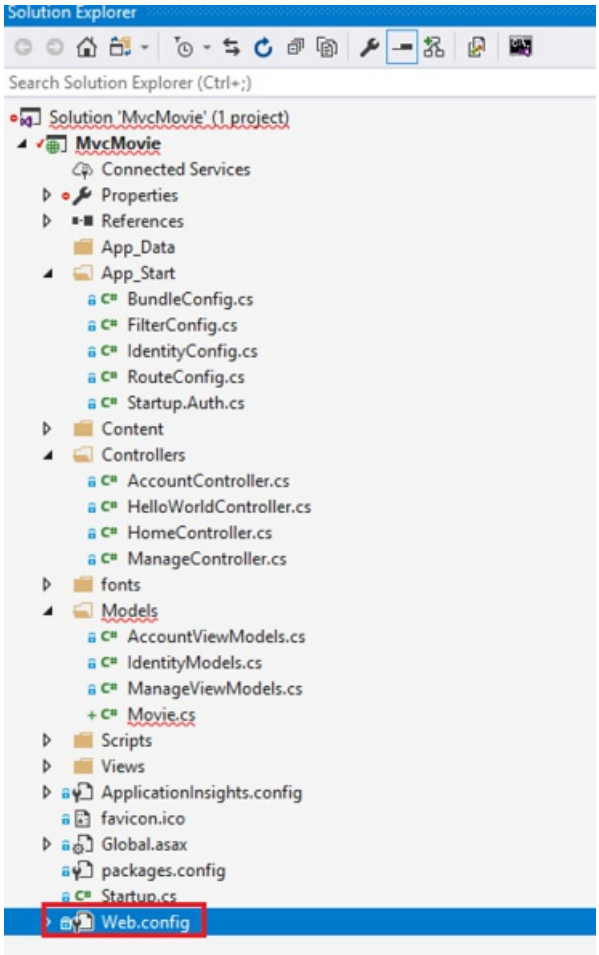
LocalDB is a lightweight version of the SQL Server Express Database Engine that starts on demand and runs in user mode. LocalDB runs in a special execution mode of SQL Server Express that enables you to work with databases as *.mdf* files. Typically, LocalDB database files are kept in the *App_Data* folder of a web project.

SQL Server Express is not recommended for use in production web applications. LocalDB in particular should not be used for production with a web application because it is not designed to work with IIS. However, a LocalDB database can be easily migrated to SQL Server or SQL Azure.

In Visual Studio 2017, LocalDB is installed by default with Visual Studio.

By default, the Entity Framework looks for a connection string named the same as the object context class (`MovieDBContext` for this project). For more information see SQL Server Connection Strings for ASP.NET Web Applications.

Open the application root *Web.config* file shown below. (Not the *Web.config* file in the *Views* folder.)



Find the `<connectionStrings>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  For more information on how to configure your ASP.NET application, please visit
  https://go.microsoft.com/fwlink/?LinkId=301880
  -->
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 --
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramewor
  </configSections>
  <connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=aspnet-MvcMovie-
  </connectionStrings>
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  </appSettings>
  <system.web>
    <authentication mode="None" />
    <compilation debug="true" targetFramework="4.6.1" />
    <httpRuntime targetFramework="4.6.1" />
  </system.web>
```

Add the following connection string to the `<connectionStrings>` element in the *Web.config* file.

```
<add name="MovieDBContext"
   connectionString="Data Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=aspnet-MvcMovie;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|\Movies.mdf"
   providerName="System.Data.SqlClient"
/>
```

The following example shows a portion of the *Web.config* file with the new connection string added:

```
<connectionStrings>
  <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=aspnet-
MvcMovie-fefdc1f0-bd81-4ce9-b712-93a062e01031;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|\aspnet-MvcMovie-fefdc1f0-bd81-4ce9-b712-93a062e01031.mdf"
providerName="System.Data.SqlClient" />
  <add name="MovieDBContext" connectionString="Data Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=aspnet-
MvcMovie;Integrated Security=SSPI;AttachDBFilename=|DataDirectory|\Movies.mdf"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

The two connection strings are very similar. The first connection string is named `DefaultConnection` and is used for the membership database to control who can access the application. The connection string you've added specifies a LocalDB database named *Movie.mdf* located in the *App_Data* folder. We won't use the membership database in this tutorial, for more information on membership, authentication and security, see my tutorial Create an ASP.NET MVC app with auth and SQL DB and deploy to Azure App Service.

The name of the connection string must match the name of the DbContext class.

```
using System;
using System.Data.Entity;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }

    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

You don't actually need to add the `MovieDBContext` connection string. If you don't specify a connection string, Entity Framework will create a LocalDB database in the users directory with the fully qualified name of the DbContext class (in this case `MvcMovie.Models.MovieDBContext`). You can name the database anything you like, as long as it has the *.MDF* suffix. For example, we could name the database *MyFilms.mdf*.

Next, you'll build a new `MoviesController` class that you can use to display the movie data and allow users to create new movie listings.

PREVIOUS NEXT

Accessing Your Model's Data from a Controller

9/16/2020 • 8 minutes to read • Edit Online

by Rick Anderson

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

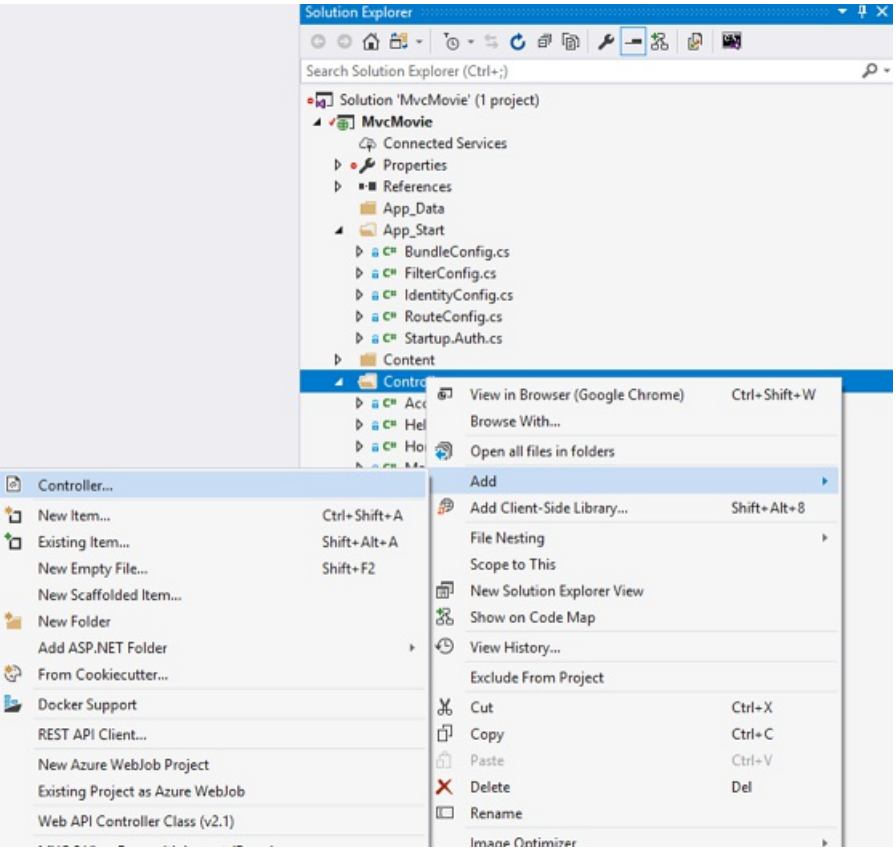
This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

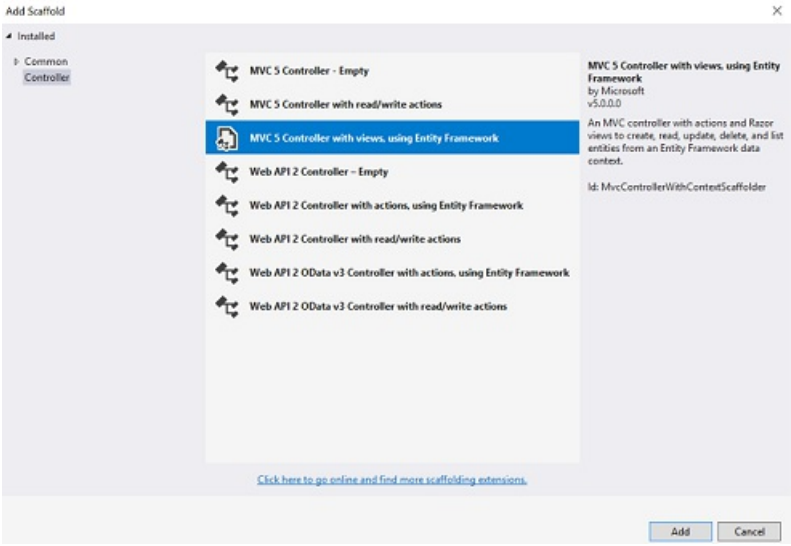
In this section, you'll create a new `MoviesController` class and write code that retrieves the movie data and displays it in the browser using a view template.

Build the application before going on to the next step. If you don't build the application, you'll get an error adding a controller.

In Solution Explorer, right-click the *Controllers* folder and then click **Add**, then **Controller**.

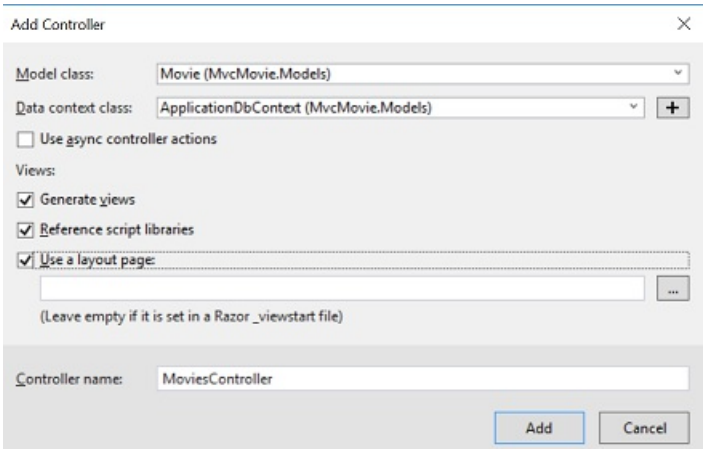


In the **Add Scaffold** dialog box, click **MVC 5 Controller with views, using Entity Framework**, and then click **Add**.



- Select **Movie (MvcMovie.Models)** for the Model class.
- Select **MovieDBContext (MvcMovie.Models)** for the Data context class.
- For the Controller name enter **MoviesController**.

  The image below shows the completed dialog.

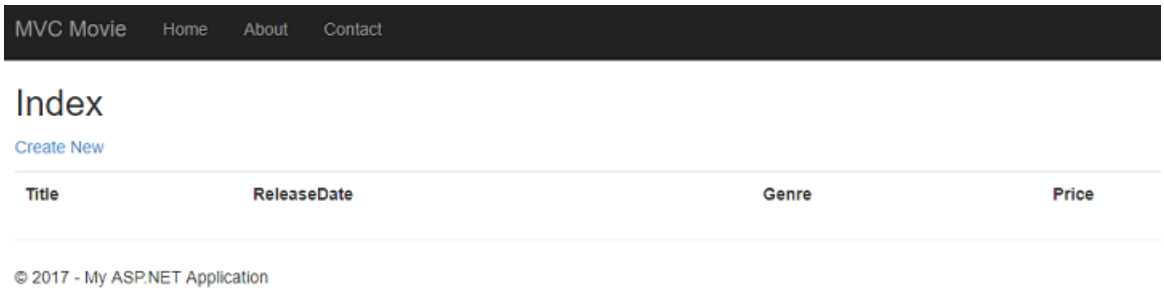


Click **Add**. (If you get an error, you probably didn't build the application before starting adding the controller.) Visual Studio creates the following files and folders:

- *A MoviesController.cs* file in the *Controllers* folder.
- A *Views\Movies* folder.
- *Create.cshtml, Delete.cshtml, Details.cshtml, Edit.cshtml*, and *Index.cshtml* in the new *Views\Movies* folder.

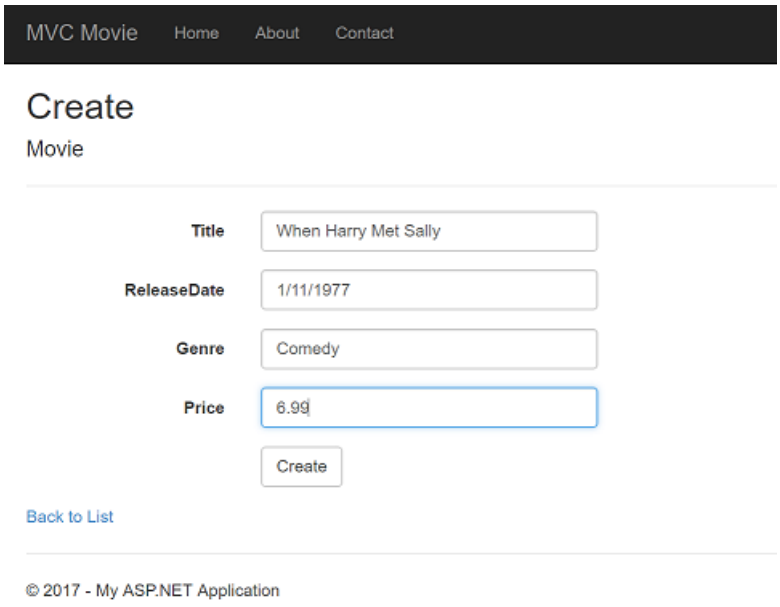
Visual Studio automatically created the CRUD (create, read, update, and delete) action methods and views for you (the automatic creation of CRUD action methods and views is known as scaffolding). You now have a fully functional web application that lets you create, list, edit, and delete movie entries.

Run the application and click on the **MVC Movie** link (or browse to the `Movies` controller by appending */Movies* to the URL in the address bar of your browser). Because the application is relying on the default routing (defined in the *App_Start\RouteConfig.cs* file), the browser request `http://localhost:xxxxx/Movies` is routed to the default `Index` action method of the `Movies` controller. In other words, the browser request `http://localhost:xxxxx/Movies` is effectively the same as the browser request `http://localhost:xxxxx/Movies/Index`. The result is an empty list of movies, because you haven't added any yet.



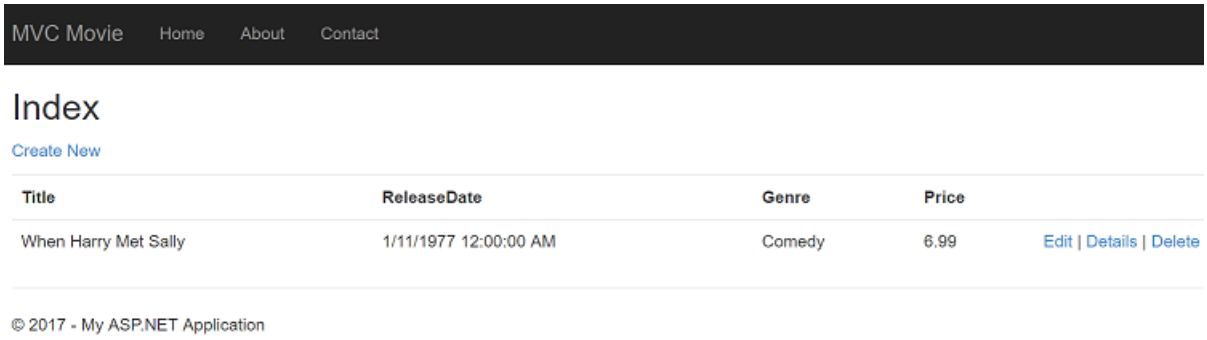
Creating a Movie

Select the **Create New** link. Enter some details about a movie and then click the **Create** button.



> **NOTE**
>
> You may not be able to enter decimal points or commas in the Price field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must include *globalize.js* and your specific *cultures/globalize.cultures.js* file(from https://github.com/jquery/globalize) and JavaScript to use `Globalize.parseFloat`. I'll show how to do this in the next tutorial. For now, just enter whole numbers like 10.

Clicking the **Create** button causes the form to be posted to the server, where the movie information is saved in the database. You're then redirected to the */Movies* URL, where you can see the newly created movie in the listing.



Create a couple more movie entries. Try the **Edit**, **Details**, and **Delete** links, which are all functional.

Examining the Generated Code

Open the *Controllers\MoviesController.cs* file and examine the generated `Index` method. A portion of the movie controller with the `Index` method is shown below.

```
public class MoviesController : Controller
{
    private MovieDBContext db = new MovieDBContext();

    // GET: /Movies/
    public ActionResult Index()
    {
        return View(db.Movies.ToList());
    }
```

A request to the `Movies` controller returns all the entries in the `Movies` table and then passes the results to the `Index` view. The following line from the `MoviesController` class instantiates a movie database context, as described previously. You can use the movie database context to query, edit, and delete movies.

```
private MovieDBContext db = new MovieDBContext();
```

Strongly Typed Models and the @model Keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view template using the `ViewBag` object. The `ViewBag` is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC also provides the ability to pass *strongly* typed objects to a view template. This strongly typed approach enables better compile-time checking of your code and richer IntelliSense in the Visual Studio editor. The scaffolding mechanism in Visual Studio used this approach (that is, passing a *strongly* typed model) with the `MoviesController` class and view templates when it created the methods and views.

In the *Controllers\MoviesController.cs* file examine the generated `Details` method. The `Details` method is shown below.

```
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}
```

The `id` parameter is generally passed as route data, for example `http://localhost:1234/movies/details/1` will set the controller to the movie controller, the action to `details` and the `id` to 1. You could also pass in the id with a query string as follows:

`http://localhost:1234/movies/details?id=1`

If a `Movie` is found, an instance of the `Movie` model is passed to the `Details` view:

```
return View(movie);
```

Examine the contents of the *Views\Movies\Details.cshtml* file:

```
@model MvcMovie.Models.Movie

@{
    ViewBag.Title = "Details";
}

<h2>Details</h2>

<div>
    <h4>Movie</h4>
<hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Title)
        </dt>
         @*Markup omitted for clarity.*@
    </dl>
</div>
<p>
    @Html.ActionLink("Edit", "Edit", new { id = Model.ID }) |
    @Html.ActionLink("Back to List", "Index")
</p>
```

By including a `@model` statement at the top of the view template file, you can specify the type of object that the view expects. When you created the movie controller, Visual Studio automatically included the following `@model` statement at the top of the *Details.cshtml* file:

```
@model MvcMovie.Models.Movie
```

This `@model` directive allows you to access the movie that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Details.cshtml* template, the code passes each movie field to the `DisplayNameFor` and DisplayFor HTML Helpers with the strongly typed `Model` object. The `Create` and `Edit` methods and view templates also pass a movie model object.

Examine the *Index.cshtml* view template and the `Index` method in the *MoviesController.cs* file. Notice how the code creates a `List` object when it calls the `View` helper method in the `Index` action method. The code then passes this `Movies` list from the `Index` action method to the view:

```
public ActionResult Index()
{
    return View(db.Movies.ToList());
}
```

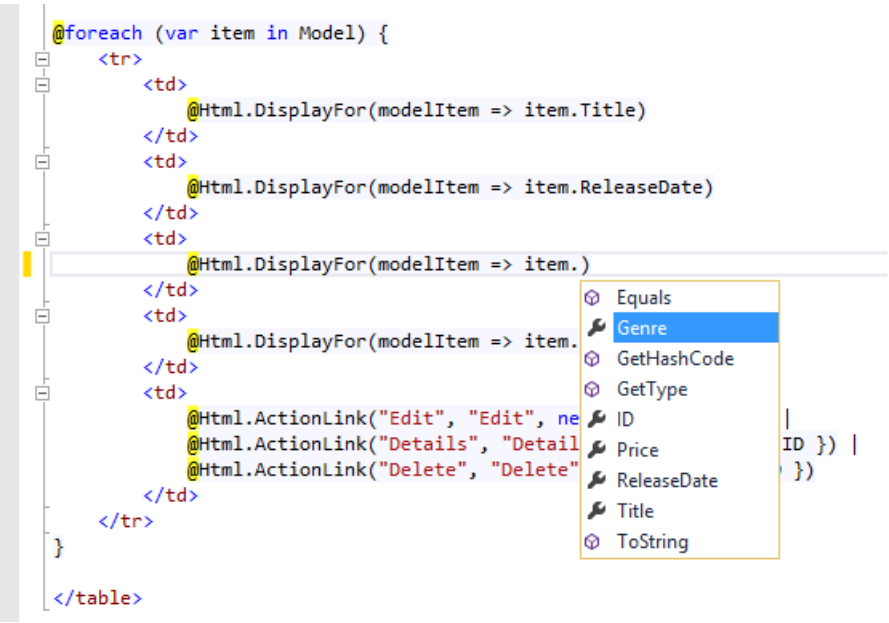
When you created the movie controller, Visual Studio automatically included the following `@model` statement at the top of the *Index.cshtml* file:

```
@model IEnumerable<MvcMovie.Models.Movie>
```

This `@model` directive allows you to access the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Index.cshtml* template, the code loops through the movies by doing a `foreach` statement over the strongly typed `Model` object:

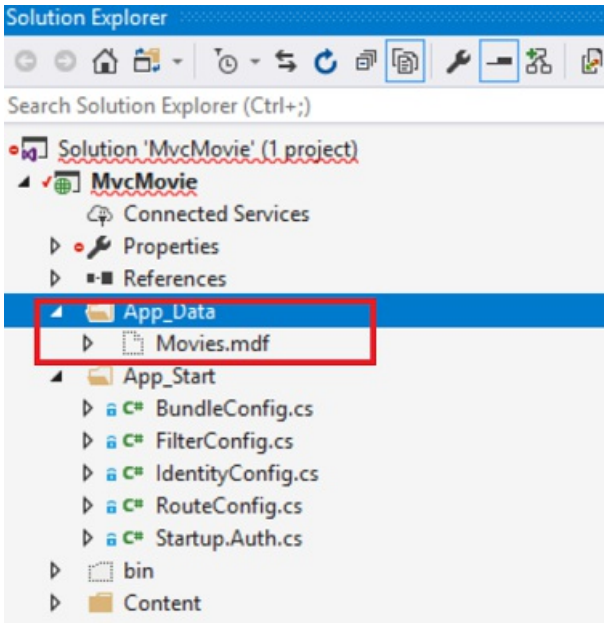
```
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
         <th>
            @Html.DisplayFor(modelItem => item.Rating)
        </th>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID })  |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
    </tr>
}
```

Because the `Model` object is strongly typed (as an `IEnumerable<Movie>` object), each `item` object in the loop is typed as `Movie`. Among other benefits, this means that you get compile-time checking of the code and full IntelliSense support in the code editor:

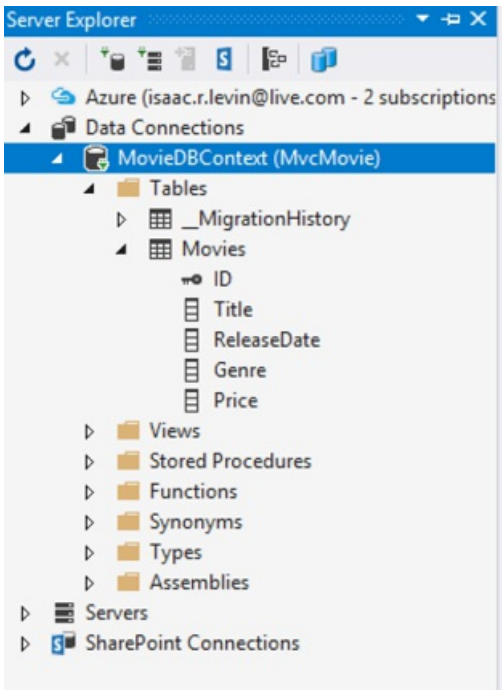


Working with SQL Server LocalDB

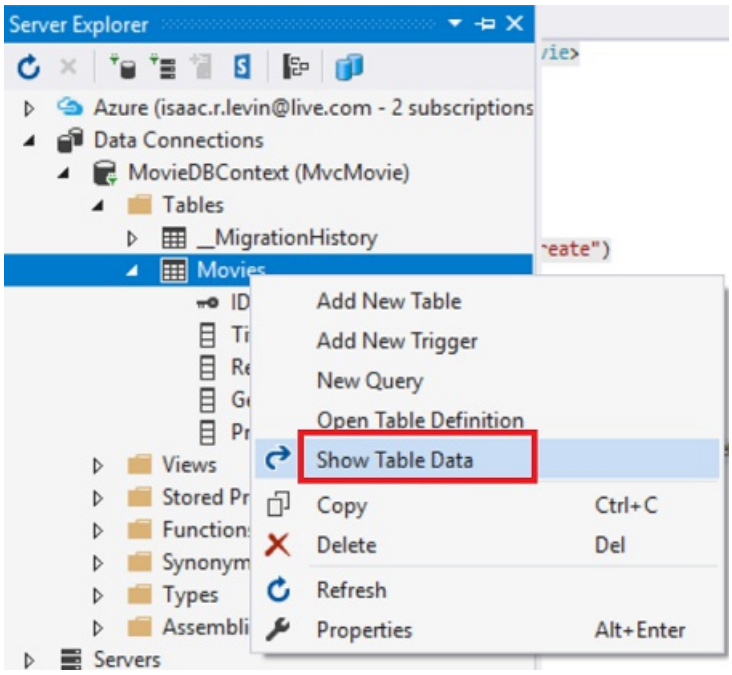
Entity Framework Code First detected that the database connection string that was provided pointed to a `Movies` database that didn't exist yet, so Code First created the database automatically. You can verify that it's been created by looking in the *App_Data* folder. If you don't see the *Movies.mdf* file, click the **Show All Files** button in the **Solution Explorer** toolbar, click the **Refresh** button, and then expand the *App_Data* folder.

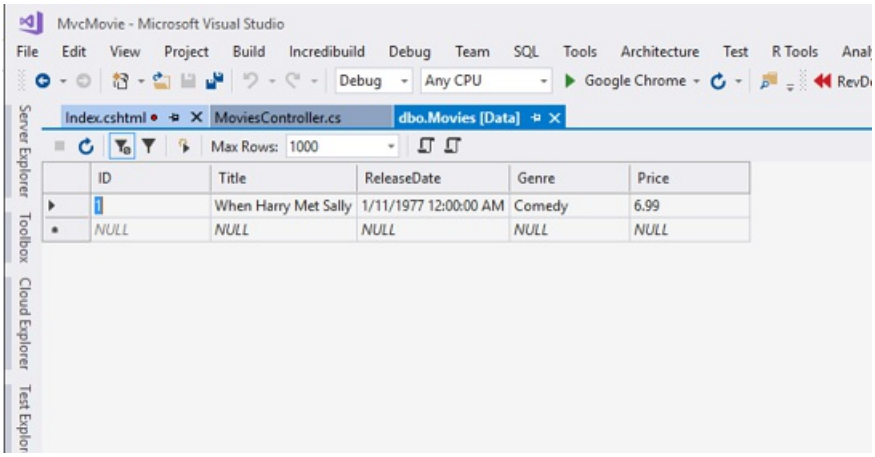


Double-click *Movies.mdf* to open **SERVER EXPLORER**, then expand the **Tables** folder to see the Movies table. Note the key icon next to ID. By default, EF will make a property named ID the primary key. For more information on EF and MVC, see Tom Dykstra's excellent tutorial on MVC and EF.

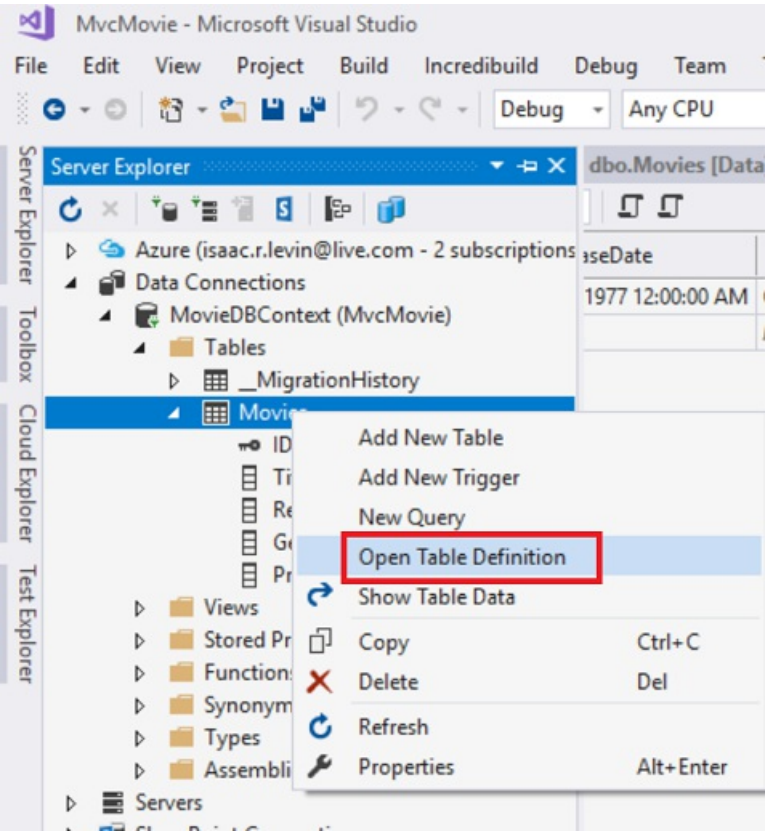


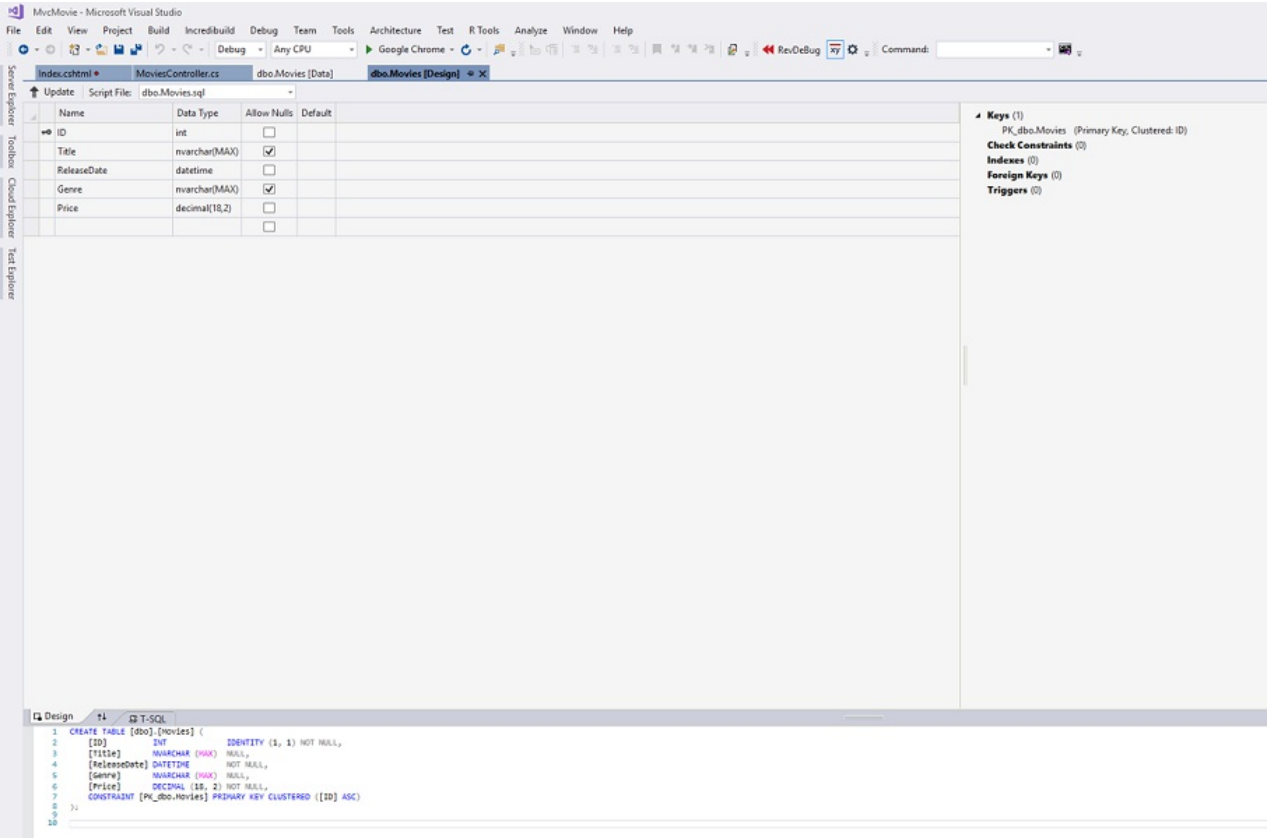
Right-click the `Movies` table and select **Show Table Data** to see the data you created.





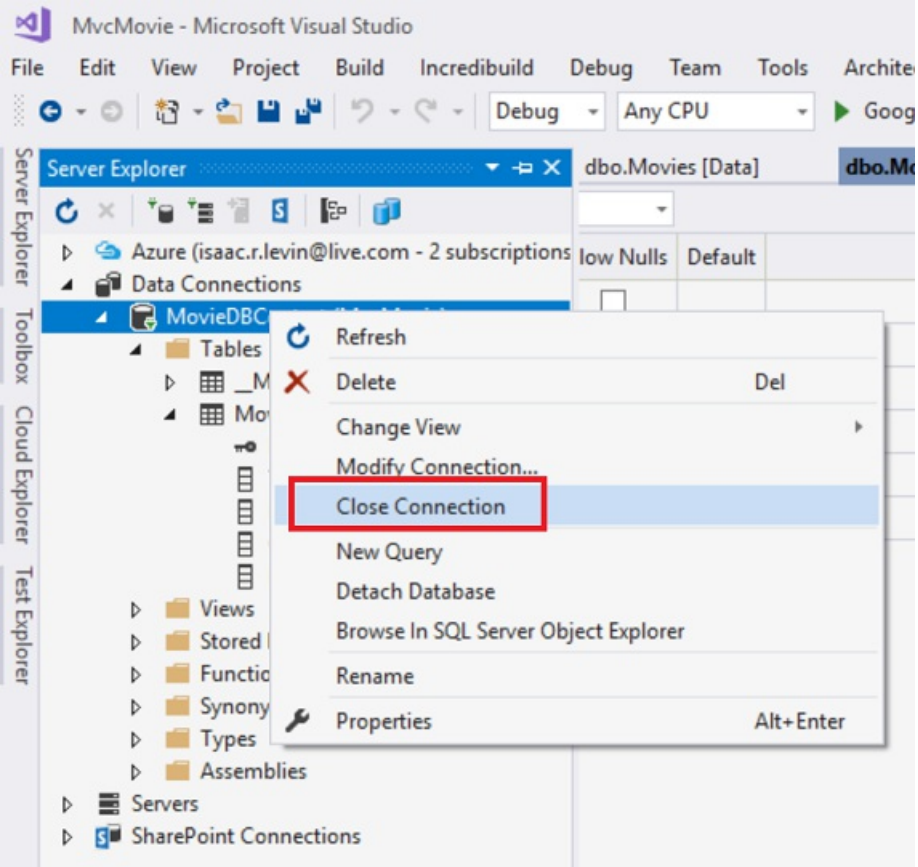
Right-click the `Movies` table and select **Open Table Definition** to see the table structure that Entity Framework Code First created for you.





Notice how the schema of the `Movies` table maps to the `Movie` class you created earlier. Entity Framework Code First automatically created this schema for you based on your `Movie` class.

When you're finished, close the connection by right clicking *MovieDBContext* and selecting **Close Connection**. (If you don't close the connection, you might get an error the next time you run the project).



You now have a database and pages to display, edit, update and delete data. In the next tutorial, we'll examine the rest of the scaffolded code and add a `SearchIndex` method and a `SearchIndex` view that lets you search for movies in this database. For more information on using Entity Framework with MVC, see Creating an Entity Framework Data Model for an ASP.NET MVC Application.

PREVIOUS NEXT

Examining the Edit Methods and Edit View

9/16/2020 • 10 minutes to read • Edit Online

by Rick Anderson

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

In this section, you'll examine the generated `Edit` action methods and views for the movie controller. But first we'll take a short diversion to make the release date look better. Open the *Models\Movie.cs* file and add the highlighted lines shown below:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }

    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

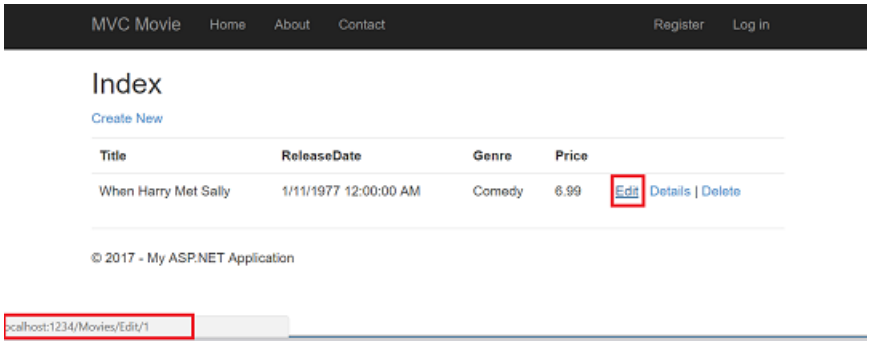
You can also make the date culture specific like this:

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
[DisplayFormat(DataFormatString = "{0:d}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

We'll cover DataAnnotations in the next tutorial. The Display attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The DataType attribute specifies the type of the data, in this case it's a date, so the time information stored in the field is not displayed. The DisplayFormat attribute is

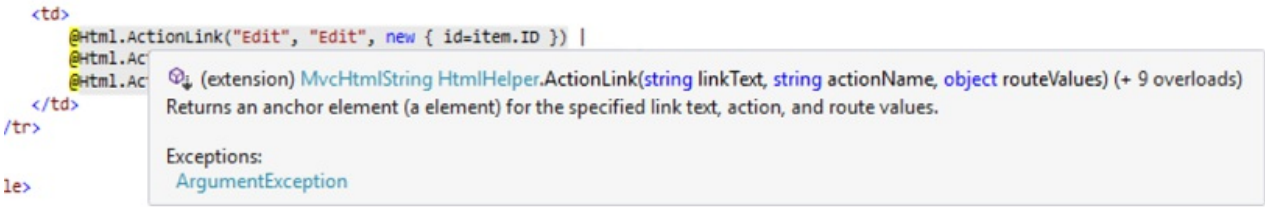
needed for a bug in the Chrome browser that renders date formats incorrectly.

Run the application and browse to the `Movies` controller. Hold the mouse pointer over an **Edit** link to see the URL that it links to.



The **Edit** link was generated by the `Html.ActionLink` method in the *Views\Movies\Index.cshtml* view:

```
@Html.ActionLink("Edit", "Edit", new { id=item.ID })
```



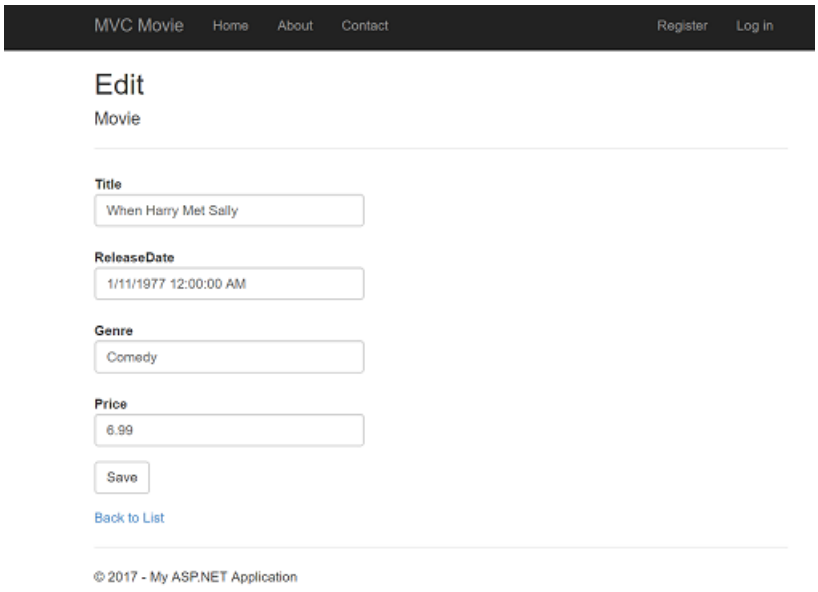
The `Html` object is a helper that's exposed using a property on the System.Web.Mvc.WebViewPage base class. The `ActionLink` method of the helper makes it easy to dynamically generate HTML hyperlinks that link to action methods on controllers. The first argument to the `ActionLink` method is the link text to render (for example, `<a>Edit Me</a>`). The second argument is the name of the action method to invoke (In this case, the `Edit` action). The final argument is an anonymous object that generates the route data (in this case, the ID of 4).

The generated link shown in the previous image is `http://localhost:1234/Movies/Edit/4`. The default route (established in *App_Start\RouteConfig.cs*) takes the URL pattern `{controller}/{action}/{id}`. Therefore, ASP.NET translates `http://localhost:1234/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `ID` equal to 4. Examine the following code from the *App_Start\RouteConfig.cs* file. The MapRoute method is used to route HTTP requests to the correct controller and action method and supply the optional ID parameter. The MapRoute method is also used by the HtmlHelpers such as `ActionLink` to generate URLs given the controller, action method and any route data.

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index",
            id = UrlParameter.Optional }
    );
}
```

You can also pass action method parameters using a query string. For example, the URL `http://localhost:1234/Movies/Edit?ID=3` also passes the parameter `ID` of 3 to the `Edit` action method of the `Movies` controller.



Open the `Movies` controller. The two `Edit` action methods are shown below.

```
// GET: /Movies/Edit/5
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}

// POST: /Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see https://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include="ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Entry(movie).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

Notice the second `Edit` action method is preceded by the `HttpPost` attribute. This attribute specifies that the overload of the `Edit` method can be invoked only for POST requests. You could apply the `HttpGet` attribute to the first edit method, but that's not necessary because it's the default. (We'll refer to action methods that are implicitly assigned the `HttpGet` attribute as `HttpGet` methods.) The Bind attribute is another important security mechanism that keeps hackers from over-posting data to your model. You should only include properties in the bind attribute that you want to change. You can read about overposting and the bind attribute in my overposting security note. In the simple model used in this tutorial, we will be binding all the data in the model. The ValidateAntiForgeryToken attribute is used to prevent forgery of a request and is paired up with `@Html.AntiForgeryToken()` in the edit view file (*Views\Movies\Edit.cshtml*), a portion is shown below:

```
@model MvcMovie.Models.Movie

@{
    ViewBag.Title = "Edit";
}
<h2>Edit</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        @Html.ValidationSummary(true)
        @Html.HiddenFor(model => model.ID)

        <div class="form-group">
            @Html.LabelFor(model => model.Title, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Title)
                @Html.ValidationMessageFor(model => model.Title)
            </div>
        </div>
```

`@Html.AntiForgeryToken()` generates a hidden form anti-forgery token that must match in the `Edit` method of the `Movies` controller. You can read more about Cross-site request forgery (also known as XSRF or CSRF) in my tutorial XSRF/CSRF Prevention in MVC.

The `HttpGet` `Edit` method takes the movie ID parameter, looks up the movie using the Entity Framework `Find` method, and returns the selected movie to the Edit view. If a movie cannot be found, HttpNotFound is returned. When the scaffolding system created the Edit view, it examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The following example shows the Edit view that was generated by the visual studio scaffolding system:

```
@model MvcMovie.Models.Movie

@{
    ViewBag.Title = "Edit";
}
<h2>Edit</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        @Html.ValidationSummary(true)
        @Html.HiddenFor(model => model.ID)

        <div class="form-group">
            @Html.LabelFor(model => model.Title, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Title)
                @Html.ValidationMessageFor(model => model.Title)
            </div>
        </div>
        <div class="form-group">
            @Html.LabelFor(model => model.ReleaseDate, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.ReleaseDate)
                @Html.ValidationMessageFor(model => model.ReleaseDate)
            </div>
        </div>
        @*Genre and Price removed for brevity.*@
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Notice how the view template has a `@model MvcMovie.Models.Movie` statement at the top of the file — this specifies that the view expects the model for the view template to be of type `Movie`.

The scaffolded code uses several *helper methods* to streamline the HTML markup. The `Html.LabelFor` helper displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The `Html.EditorFor` helper renders an HTML `<input>` element. The `Html.ValidationMessageFor` helper displays any validation messages associated with that property.

Run the application and navigate to the */Movies* URL. Click an **Edit** link. In the browser, view the source for the page. The HTML for the form element is shown below.

```
<form action="/movies/Edit/4" method="post">
   <input name="__RequestVerificationToken" type="hidden" value="UxY6bkQyJCXO3Kn5AXg-
6TXxOj6yVBi9tghHaQ5Lq_qwKvcojNXEEfcbn-FGh_0vuw4tS_BRk7QQQHlJp8AP4_X4orVNoQnp2cd8kXhykS01" />  <fieldset
class="form-horizontal">
      <legend>Movie</legend>

      <input data-val="true" data-val-number="The field ID must be a number." data-val-required="The ID field
is required." id="ID" name="ID" type="hidden" value="4" />

      <div class="control-group">
         <label class="control-label" for="Title">Title</label>
         <div class="controls">
            <input class="text-box single-line" id="Title" name="Title" type="text" value="GhostBusters" />
            <span class="field-validation-valid help-inline" data-valmsg-for="Title" data-valmsg-
replace="true"></span>
         </div>
      </div>

      <div class="control-group">
         <label class="control-label" for="ReleaseDate">Release Date</label>
         <div class="controls">
            <input class="text-box single-line" data-val="true" data-val-date="The field Release Date must be
a date." data-val-required="The Release Date field is required." id="ReleaseDate" name="ReleaseDate"
type="date" value="1/1/1984" />
            <span class="field-validation-valid help-inline" data-valmsg-for="ReleaseDate" data-valmsg-
replace="true"></span>
         </div>
      </div>

      <div class="control-group">
         <label class="control-label" for="Genre">Genre</label>
         <div class="controls">
            <input class="text-box single-line" id="Genre" name="Genre" type="text" value="Comedy" />
            <span class="field-validation-valid help-inline" data-valmsg-for="Genre" data-valmsg-
replace="true"></span>
         </div>
      </div>

      <div class="control-group">
         <label class="control-label" for="Price">Price</label>
         <div class="controls">
            <input class="text-box single-line" data-val="true" data-val-number="The field Price must be a
number." data-val-required="The Price field is required." id="Price" name="Price" type="text" value="7.99" />
            <span class="field-validation-valid help-inline" data-valmsg-for="Price" data-valmsg-
replace="true"></span>
         </div>
      </div>

      <div class="form-actions no-color">
         <input type="submit" value="Save" class="btn" />
      </div>
   </fieldset>
</form>
```

The `<input>` elements are in an HTML `<form>` element whose `action` attribute is set to post to the */Movies/Edit* URL. The form data will be posted to the server when the **Save** button is clicked. The second line shows the hidden XSRF token generated by the `@Html.AntiForgeryToken()` call.

Processing the POST Request

The following listing shows the `HttpPost` version of the `Edit` action method.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include="ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Entry(movie).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

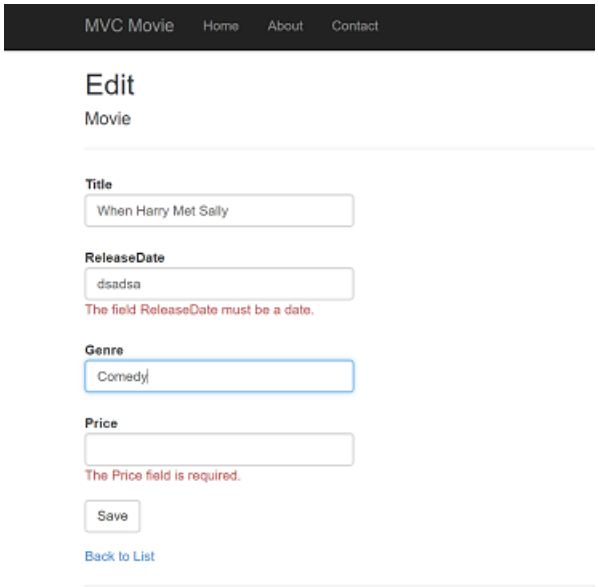
The ValidateAntiForgeryToken attribute validates the XSRF token generated by the `@Html.AntiForgeryToken()` call in the view.

The ASP.NET MVC model binder takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` verifies that the data submitted in the form can be used to modify (edit or update) a `Movie` object. If the data is valid, the movie data is saved to the `Movies` collection of the `db` (`MovieDBContext` instance). The new movie data is saved to the database by calling the `SaveChanges` method of `MovieDBContext`. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which displays the movie collection, including the changes just made.

As soon as the client-side validation determines the value of a field isn't valid, an error message is displayed. If JavaScript is disabled, client side validation is disabled. However, the server detects the posted values aren't valid, and the form values are redisplayed with error messages.

Validation is examined in more detail later in the tutorial.

The `Html.ValidationMessageFor` helpers in the *Edit.cshtml* view template take care of displaying appropriate error messages.



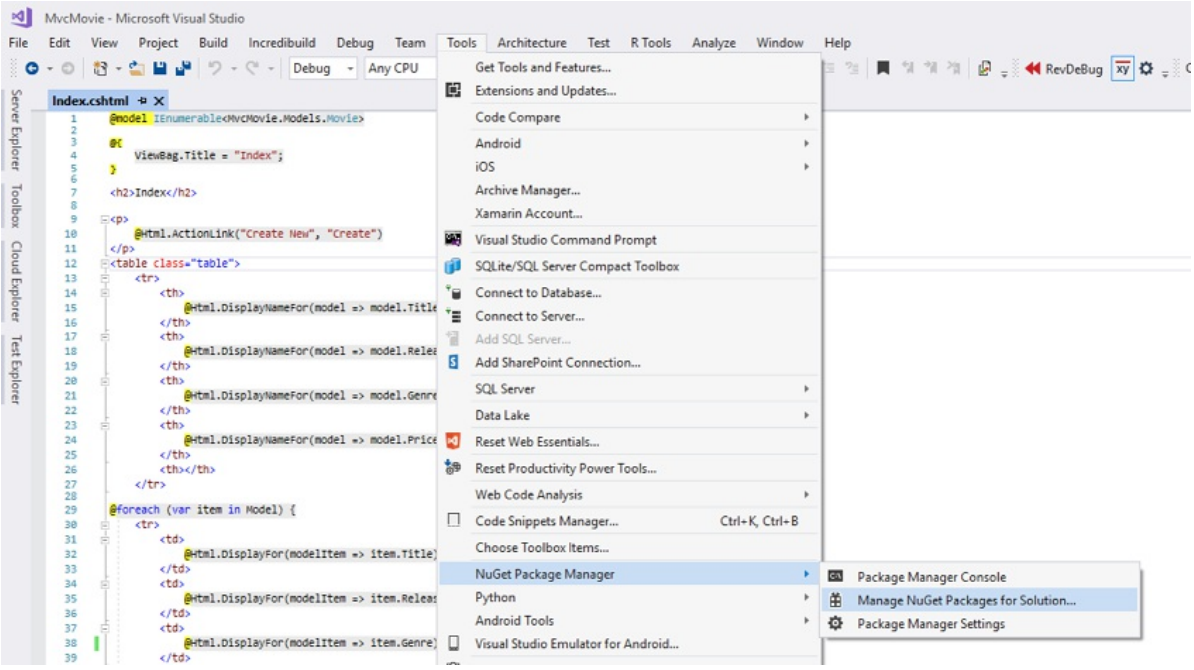
All the `HttpGet` methods follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the model to the view. The `Create` method passes an empty movie object to the Create view. All the methods that create, edit, delete, or otherwise modify data do so in the `HttpPost` overload of the method. Modifying data in an HTTP GET method is a security risk, as described in the blog post entry ASP.NET MVC Tip #46 – Don't use Delete Links because they create Security Holes. Modifying data in a GET method also violates HTTP best practices and the architectural REST pattern, which specifies that GET requests should not change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

jQuery validation for non-English locales

If you are using a US-English computer, you can skip this section and go to the next tutorial. You can download the Globalize version of this tutorial here. For an excellent two part tutorial on Internationalization, see Nadeem's ASP.NET MVC 5 Internationalization.

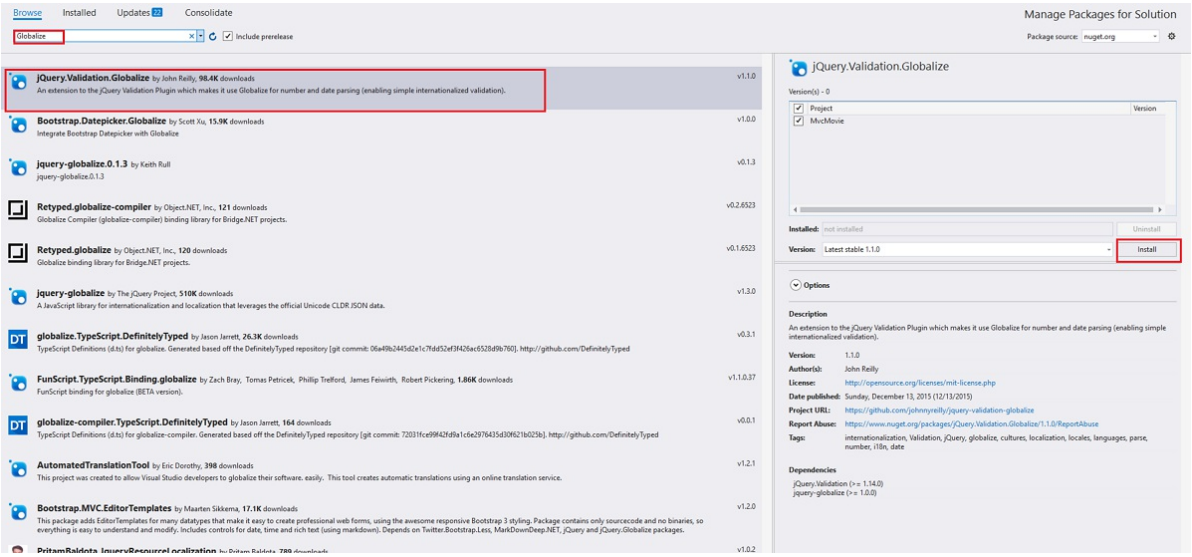
> **NOTE**
>
> to support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must include *globalize.js* and your specific *cultures/globalize.cultures.js* file(from https://github.com/jquery/globalize) and JavaScript to use `Globalize.parseFloat`. You can get the jQuery non-English validation from NuGet. (Don't install Globalize if you are using a English locale.)

1. From the **Tools** menu click **NuGet Package Manager**, and then click **Manage NuGet Packages for Solution**.



2. On the left pane, select **Browse***.*(See the image below.)

3. In the input box, enter *Globalize**.



Choose `jQuery.Validation.Globalize`, choose `MvcMovie` and click **Install**. The *Scripts\jquery.globalize\globalize.js* file will be added to your project. The

Scripts\jquery.globalize\cultures folder will contain many culture JavaScript files. Note, it may take five minutes to install this package.

The following code shows the modifications to the Views\Movies\Edit.cshtml file:

```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")

<script src="~/Scripts/globalize/globalize.js"></script>
<script
src="~/Scripts/globalize/cultures/globalize.culture.@(System.Threading.Thread.CurrentThread.CurrentCulture.Nam
e).js"></script>
<script>
    $.validator.methods.number = function (value, element) {
        return this.optional(element) ||
            !isNaN(Globalize.parseFloat(value));
    }
    $(document).ready(function () {
        Globalize.culture('@(System.Threading.Thread.CurrentThread.CurrentCulture.Name)');
    });
</script>
<script>
    jQuery.extend(jQuery.validator.methods, {
        range: function (value, element, param) {
            //Use the Globalization plugin to parse the value
            var val = Globalize.parseFloat(value);
            return this.optional(element) || (
                val >= param[0] && val <= param[1]);
        }
    });
    $.validator.methods.date = function (value, element) {
        return this.optional(element) ||
            Globalize.parseDate(value) ||
            Globalize.parseDate(value, "yyyy-MM-dd");
    }
</script>
}
```

To avoid repeating this code in every Edit view, you can move it to the layout file. To optimize the script download, see my tutorial Bundling and Minification.

For more information see ASP.NET MVC 3 Internationalization and ASP.NET MVC 3 Internationalization - Part 2 (NerdDinner).

As a temporary fix, if you can't get validation working in your locale, you can force your computer to use US English or you can disable JavaScript in your browser. To force your computer to use US English, you can add the globalization element to the projects root *web.config* file. The following code shows the globalization element with the culture set to United States English.

```
<system.web>
    <globalization culture ="en-US" />
    <!--elements removed for clarity-->
  </system.web>
```

In the next tutorial, we'll implement search functionality.

PREVIOUS NEXT

Search

9/16/2020 • 9 minutes to read • Edit Online

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

Adding a Search Method and Search View

In this section you'll add search capability to the `Index` action method that lets you search movies by genre or name.

Prerequisites

To match this section's screenshots, you need to run the application (F5) and add the following movies to the database.

TITLE	RELEASE DATE	GENRE	PRICE
Ghostbusters	6/8/1984	Comedy	6.99
Ghostbusters II	6/16/1989	Comedy	6.99
Planet of the Apes	3/27/1986	Action	5.99

Updating the Index Form

Start by updating the `Index` action method to the existing `MoviesController` class. Here's the code:

```
public ActionResult Index(string searchString)
{
    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

The first line of the `Index` method creates the following LINQ query to select the movies:

```
var movies = from m in db.Movies
                 select m;
```

The query is defined at this point, but hasn't yet been run against the database.

If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string, using the following code:

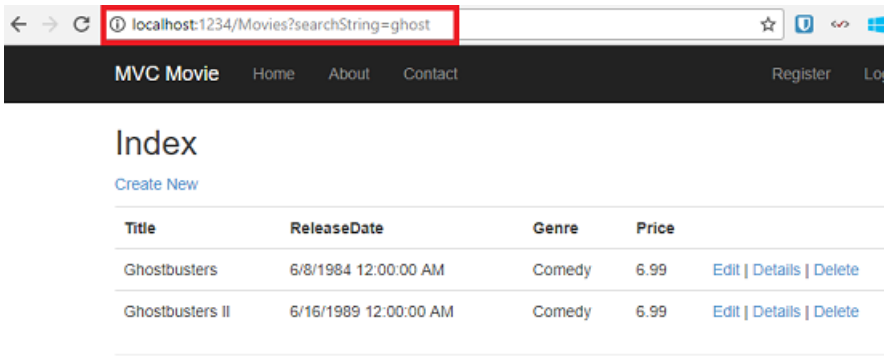
```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

The `s => s.Title` code above is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method used in the above code. LINQ queries are not executed when they are defined or when they are modified by calling a method such as `Where` or `OrderBy` . Instead, query execution is deferred, which means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToList` method is called. In the `Search` sample, the query is executed in the *Index.cshtml* view. For more information about deferred query execution, see Query Execution.

> **NOTE**
> The Contains method is run on the database, not the c# code above. On the database, Contains maps to SQL LIKE, which is case insensitive.

Now you can update the `Index` view that will display the form to the user.

Run the application and navigate to */Movies/Index*. Append a query string such as `?searchString=ghost` to the URL. The filtered movies are displayed.



If you change the signature of the `Index` method to have a parameter named `id`, the `id` parameter will match the `{id}` placeholder for the default routes set in the *App_Start\RouteConfig.cs* file.

```
{controller}/{action}/{id}
```

The original `Index` method looks like this::

```
public ActionResult Index(string searchString)
{
    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

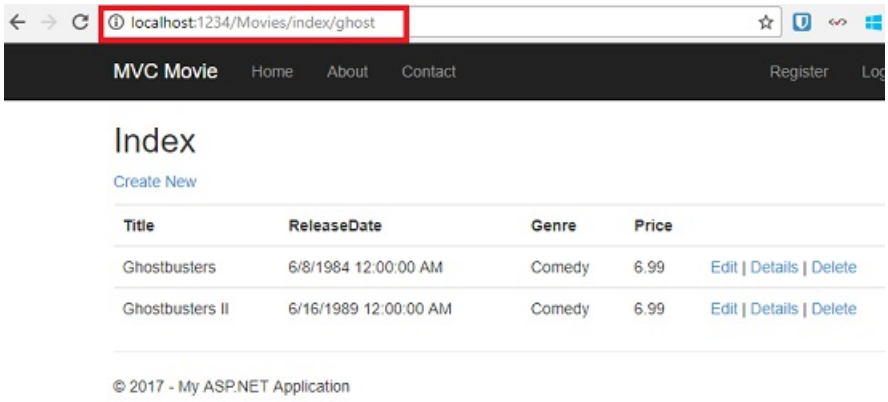
The modified `Index` method would look as follows:

```
public ActionResult Index(string id)
{
    string searchString = id;
    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

You can now pass the search title as route data (a URL segment) instead of as a query string value.



However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI to help them filter movies. If you changed the signature of the `Index` method to test how to pass the route-bound ID parameter, change it back so that your `Index` method takes a string parameter named `searchString`:

```
public ActionResult Index(string searchString)
{
    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

Open the *Views\Movies\Index.cshtml* file, and just after `@Html.ActionLink("Create New", "Create")`, add the form markup highlighted below:

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewBag.Title = "Index";
}

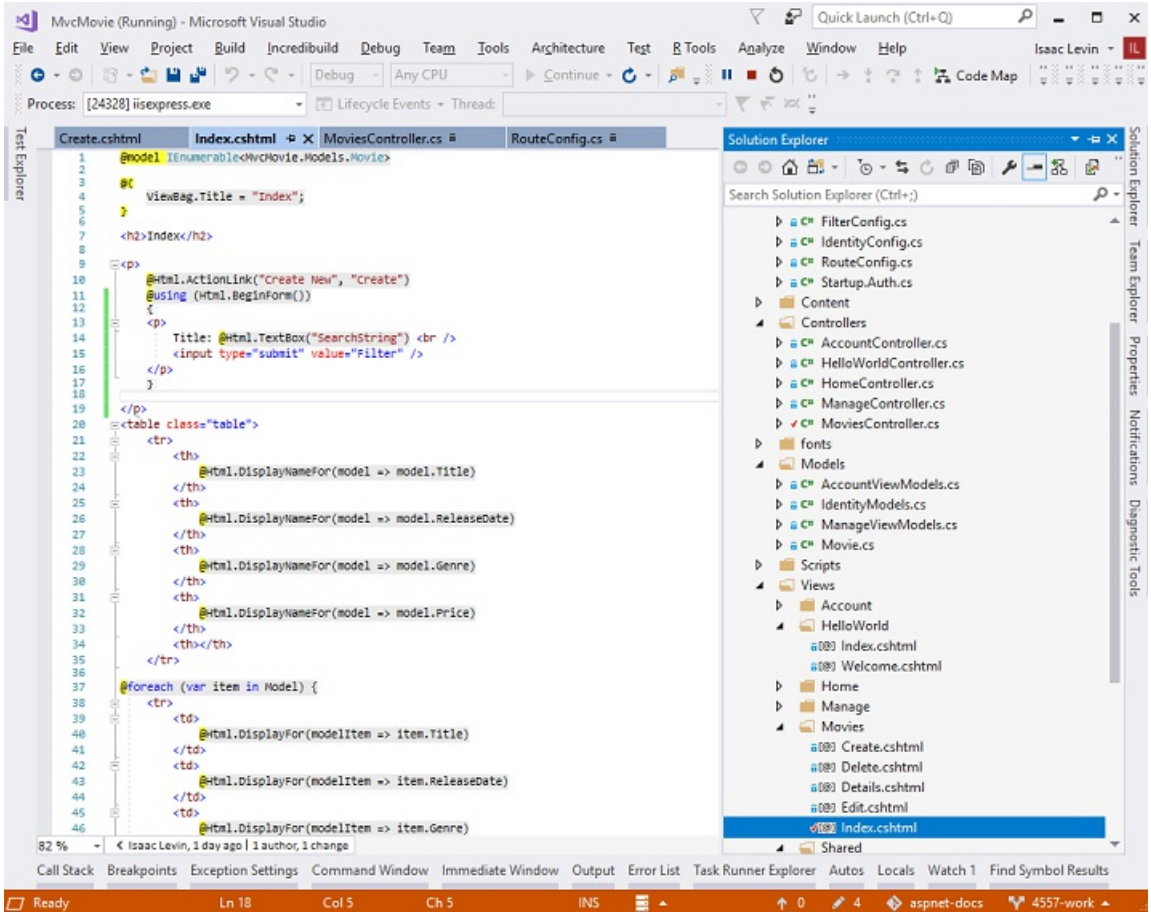
<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")

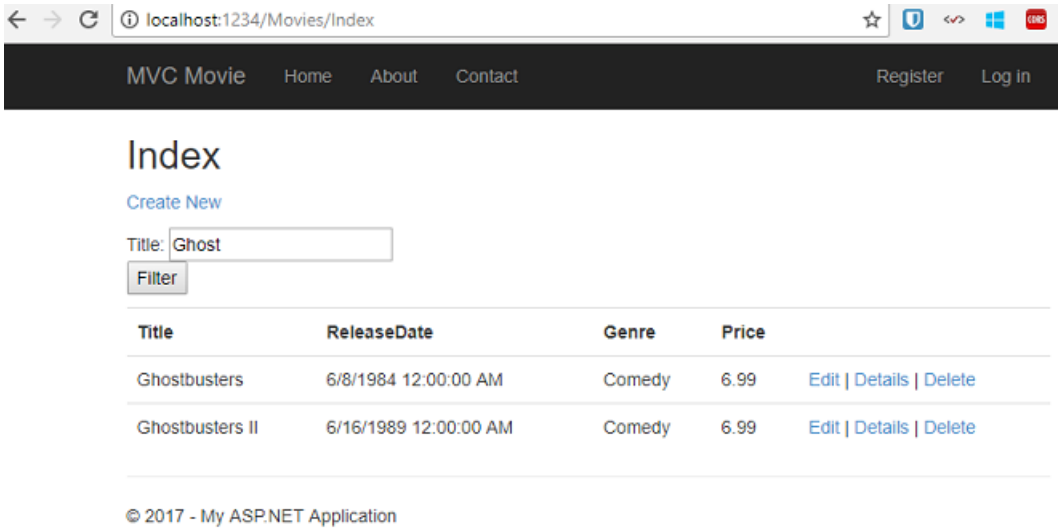
     @using (Html.BeginForm()){
         <p> Title: @Html.TextBox("SearchString") <br />
         <input type="submit" value="Filter" /></p>
        }
</p>
```

The `Html.BeginForm` helper creates an opening `<form>` tag. The `Html.BeginForm` helper causes the form to post to itself when the user submits the form by clicking the **Filter** button.

Visual Studio 2013 has a nice improvement when displaying and editing View files. When you run the application with a view file open, Visual Studio 2013 invokes the correct controller action method to display the view.



With the Index view open in Visual Studio (as shown in the image above), tap Ctr F5 or F5 to run the application and then try searching for a movie.



There's no `HttpPost` overload of the `Index` method. You don't need it, because the method isn't changing the state of the application, just filtering data.

You could add the following `HttpPost Index` method. In that case, the action invoker would match the `HttpPost Index` method, and the `HttpPost Index` method would run as shown in the image below.

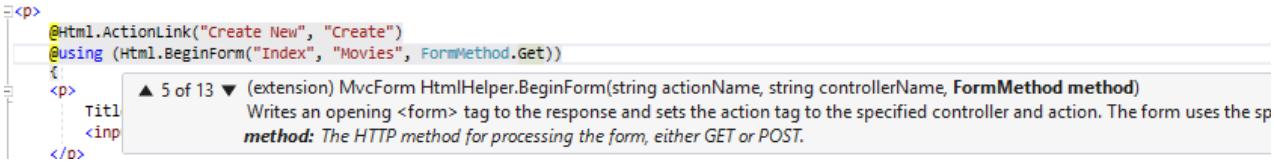
```
[HttpPost]
public string Index(FormCollection fc, string searchString)
{
    return "<h3> From [HttpPost]Index: " + searchString + "</h3>";
}
```

From [HttpPost]Index: ghost

However, even if you add this `HttpPost` version of the `Index` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (localhost:xxxxx/Movies/Index) -- there's no search information in the URL itself. Right now, the search string information is sent to the server as a form field value. This means you can't capture that search information to bookmark or send to friends in a URL.

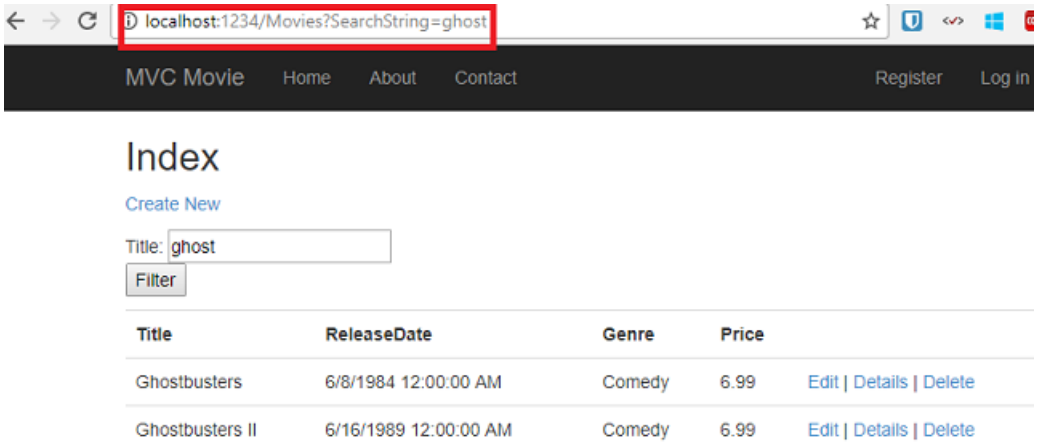
The solution is to use an overload of `BeginForm` that specifies that the POST request should add the search information to the URL and that it should be routed to the `HttpGet` version of the `Index` method. Replace the existing parameterless `BeginForm` method with the following markup:

```
@using (Html.BeginForm("Index","Movies",FormMethod.Get))
```



Now when you submit a search, the URL contains a search query string. Searching will also go to the

`HttpGet Index` action method, even if you have a `HttpPost Index` method.



Adding Search by Genre

If you added the `HttpPost` version of the `Index` method, delete it now.

Next, you'll add a feature to let users search for movies by genre. Replace the `Index` method with the following code:

```
public ActionResult Index(string movieGenre, string searchString)
{
    var GenreLst = new List<string>();

    var GenreQry = from d in db.Movies
                   orderby d.Genre
                   select d.Genre;

    GenreLst.AddRange(GenreQry.Distinct());
    ViewBag.movieGenre = new SelectList(GenreLst);

    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!string.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    return View(movies);
}
```

This version of the `Index` method takes an additional parameter, namely `movieGenre`. The first few lines of code create a `List` object to hold movie genres from the database.

The following code is a LINQ query that retrieves all the genres from the database.

```
var GenreQry = from d in db.Movies
               orderby d.Genre
               select d.Genre;
```

The code uses the `AddRange` method of the generic `List` collection to add all the distinct genres to the list.

(Without the `Distinct` modifier, duplicate genres would be added — for example, comedy would be added twice in our sample). The code then stores the list of genres in the `ViewBag.MovieGenre` object. Storing category data (such a movie genres) as a SelectList object in a `ViewBag`, then accessing the category data in a dropdown list box is a typical approach for MVC applications.

The following code shows how to check the `movieGenre` parameter. If it's not empty, the code further constrains the movies query to limit the selected movies to the specified genre.

```
if (!string.IsNullOrEmpty(movieGenre))
{
    movies = movies.Where(x => x.Genre == movieGenre);
}
```

As stated previously, the query is not run on the database until the movie list is iterated over (which happens in the View, after the `Index` action method returns).

Adding Markup to the Index View to Support Search by Genre

Add an `Html.DropDownList` helper to the *Views\Movies\Index.cshtml* file, just before the `TextBox` helper. The completed markup is shown below:

```
@model IEnumerable<MvcMovie.Models.Movie>
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
<p>
    @Html.ActionLink("Create New", "Create")
    @using (Html.BeginForm("Index", "Movies", FormMethod.Get))
    {
    <p>
        Genre: @Html.DropDownList("movieGenre", "All")
        Title: @Html.TextBox("SearchString")
        <input type="submit" value="Filter" />
    </p>
    }
</p>
<table class="table">
```

In the following code:

```
@Html.DropDownList("movieGenre", "All")
```

The parameter "MovieGenre" provides the key for the `DropDownList` helper to find a `IEnumerable<SelectListItem>` in the `ViewBag`. The `ViewBag` was populated in the action method:

```
public ActionResult Index(string movieGenre, string searchString)
{
    var GenreLst = new List<string>();

    var GenreQry = from d in db.Movies
                   orderby d.Genre
                   select d.Genre;

    GenreLst.AddRange(GenreQry.Distinct());
    ViewBag.movieGenre = new SelectList(GenreLst);

    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!string.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

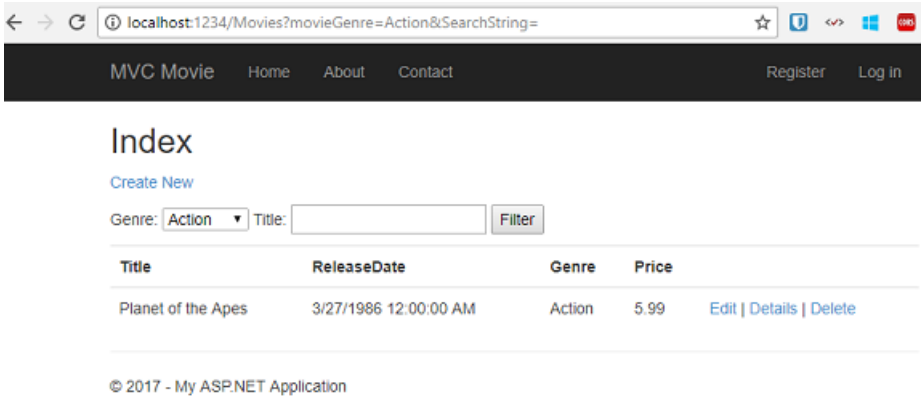
    return View(movies);
}
```

The parameter "All" provides an option label. If you inspect that choice in your browser, you'll see that its "value" attribute is empty. Since our controller only filters `if` the string is not `null` or empty, submitting an empty value for `movieGenre` shows all genres.

You can also set an option to be selected by default. If you wanted "Comedy" as your default option, you would change the code in the Controller like so:

```
ViewBag.movieGenre = new SelectList(GenreLst, "Comedy");
```

Run the application and browse to */Movies/Index*. Try a search by genre, by movie name, and by both criteria.



In this section you created a search action method and view that let users search by movie title and genre. In the next section, you'll look at how to add a property to the `Movie` model and how to add an initializer that will automatically create a test database.

PREVIOUS NEXT

Adding a New Field

9/16/2020 • 11 minutes to read • Edit Online

by Rick Anderson

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

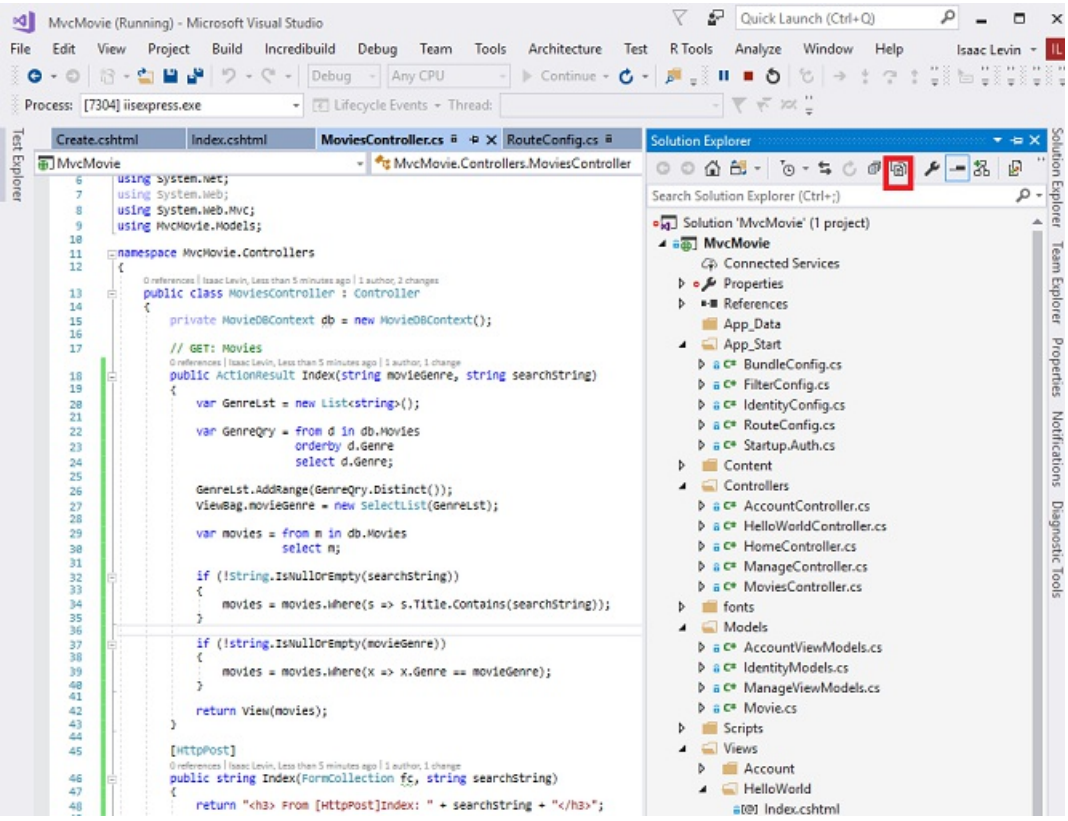
- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

In this section you'll use Entity Framework Code First Migrations to migrate some changes to the model classes so the change is applied to the database.

By default, when you use Entity Framework Code First to automatically create a database, as you did earlier in this tutorial, Code First adds a table to the database to help track whether the schema of the database is in sync with the model classes it was generated from. If they aren't in sync, the Entity Framework throws an error. This makes it easier to track down issues at development time that you might otherwise only find (by obscure errors) at run time.

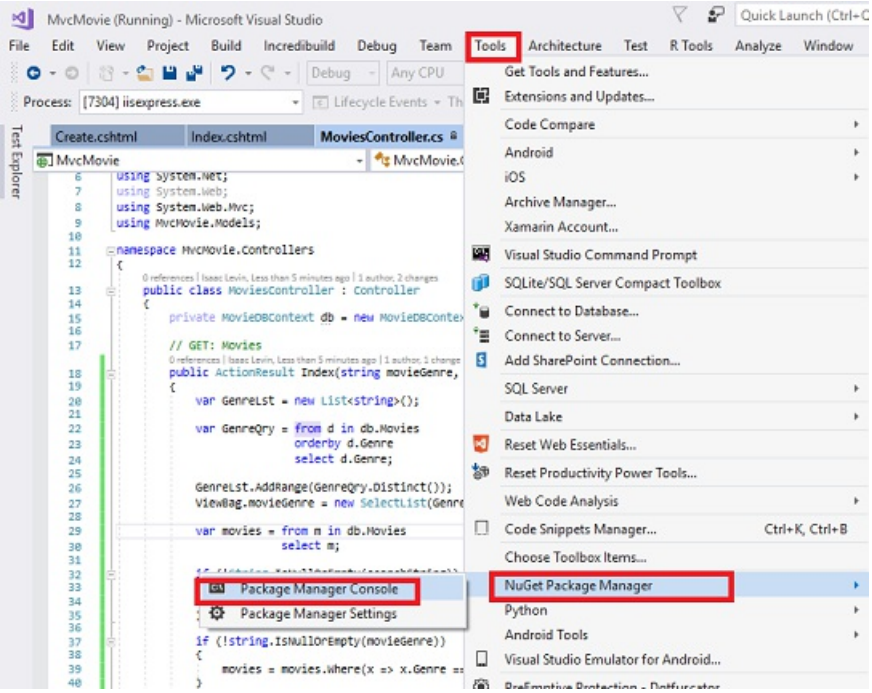
Setting up Code First Migrations for Model Changes

Navigate to Solution Explorer. Right click on the *Movies.mdf* file and select **Delete** to remove the movies database. If you don't see the *Movies.mdf* file, click on the **Show All Files** icon shown below in the red outline.



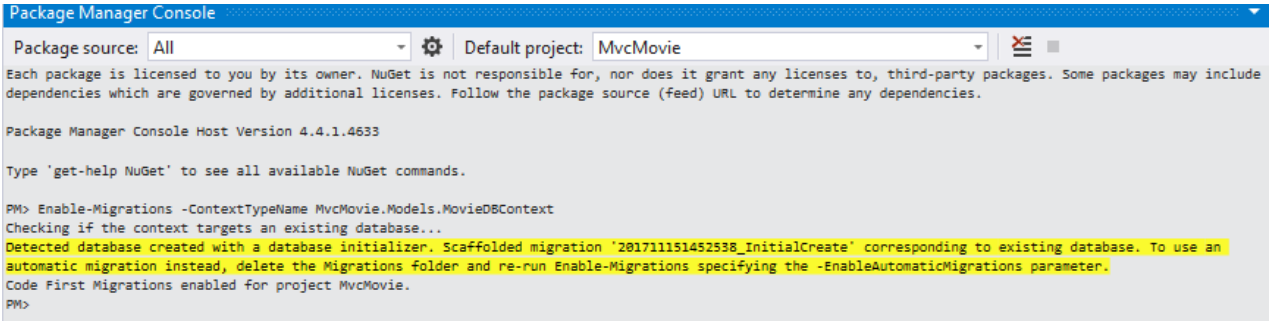
Build the application to make sure there are no errors.

From the **Tools** menu, click **NuGet Package Manager** and then **Package Manager Console**.

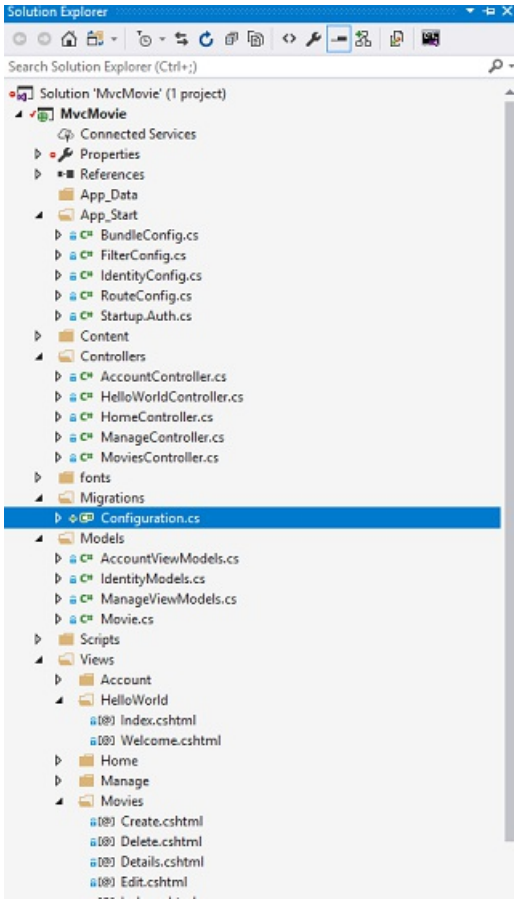


In the **Package Manager Console** window at the `PM>` prompt enter

Enable-Migrations -ContextTypeName MvcMovie.Models.MovieDBContext



The **Enable-Migrations** command (shown above) creates a *Configuration.cs* file in a new *Migrations* folder.



Visual Studio opens the *Configuration.cs* file. Replace the `Seed` method in the *Configuration.cs* file with the following code:

```
protected override void Seed(MvcMovie.Models.MovieDBContext context)
{
    context.Movies.AddOrUpdate( i => i.Title,
        new Movie
        {
            Title = "When Harry Met Sally",
            ReleaseDate = DateTime.Parse("1989-1-11"),
            Genre = "Romantic Comedy",
            Price = 7.99M
        },

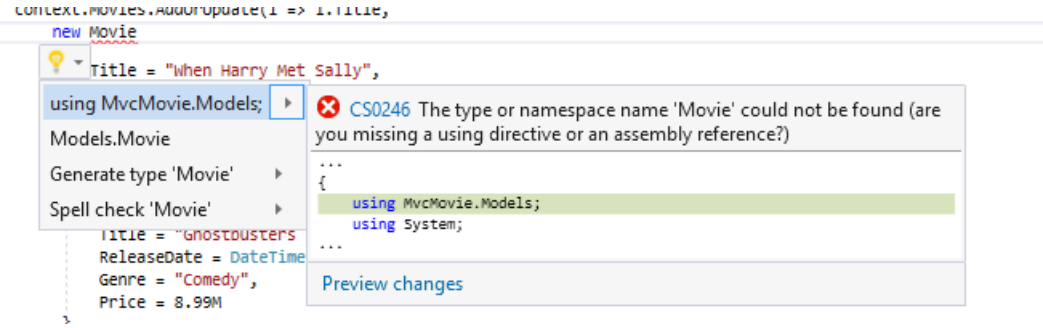
         new Movie
         {
             Title = "Ghostbusters ",
             ReleaseDate = DateTime.Parse("1984-3-13"),
             Genre = "Comedy",
             Price = 8.99M
         },

         new Movie
         {
             Title = "Ghostbusters 2",
             ReleaseDate = DateTime.Parse("1986-2-23"),
             Genre = "Comedy",
             Price = 9.99M
         },

        new Movie
        {
            Title = "Rio Bravo",
            ReleaseDate = DateTime.Parse("1959-4-15"),
            Genre = "Western",
            Price = 3.99M
        }
    );

}
```

Hover over the red squiggly line under `Movie` and click `Show Potential Fixes` and then click **using MvcMovie.Models;**



Doing so adds the following using statement:

```
using MvcMovie.Models;
```

NOTE

Code First Migrations calls the `Seed` method after every migration (that is, calling **update-database** in the Package Manager Console), and this method updates rows that have already been inserted, or inserts them if they don't exist yet.

The AddOrUpdate method in the following code performs an "upsert" operation:

```
context.Movies.AddOrUpdate(i => i.Title,
    new Movie
    {
        Title = "When Harry Met Sally",
        ReleaseDate = DateTime.Parse("1989-1-11"),
        Genre = "Romantic Comedy",
        Rating = "PG",
        Price = 7.99M
    }
```

Because the Seed method runs with every migration, you can't just insert data, because the rows you are trying to add will already be there after the first migration that creates the database. The "upsert" operation prevents errors that would happen if you try to insert a row that already exists, but it overrides any changes to data that you may have made while testing the application. With test data in some tables you might not want that to happen: in some cases when you change data while testing you want your changes to remain after database updates. In that case you want to do a conditional insert operation: insert a row only if it doesn't already exist.

The first parameter passed to the AddOrUpdate method specifies the property to use to check if a row already exists. For the test movie data that you are providing, the `Title` property can be used for this purpose since each title in the list is unique:

```
context.Movies.AddOrUpdate(i => i.Title,
```

This code assumes that titles are unique. If you manually add a duplicate title, you'll get the following exception the next time you perform a migration.

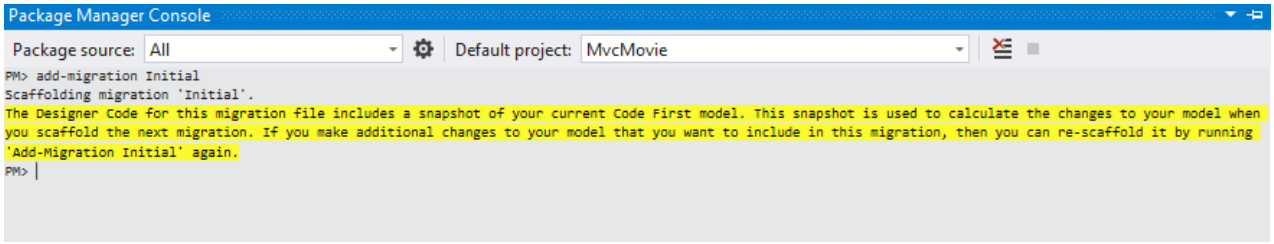
Sequence contains more than one element

For more information about the AddOrUpdate method, see Take care with EF 4.3 AddOrUpdate Method..

Press CTRL-SHIFT-B to build the project.(The following steps will fail if you don't build at this point.)

The next step is to create a `DbMigration` class for the initial migration. This migration creates a new database, that's why you deleted the *movie.mdf* file in a previous step.

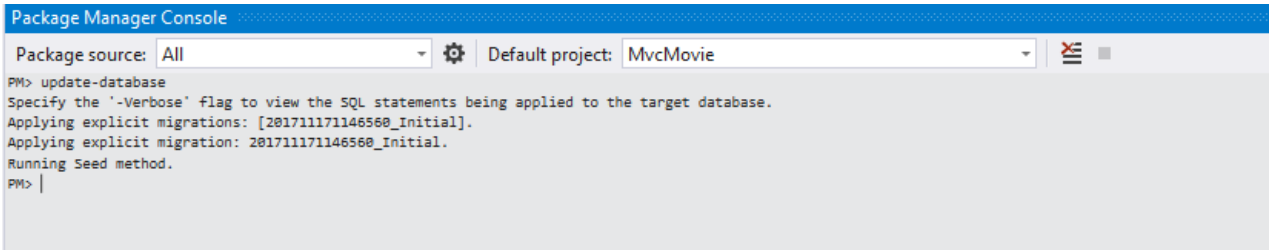
In the **Package Manager Console** window, enter the command `add-migration Initial` to create the initial migration. The name "Initial" is arbitrary and is used to name the migration file created.



Code First Migrations creates another class file in the *Migrations* folder (with the name *{DateStamp}_Initial.cs*), and this class contains code that creates the database schema. The migration filename is pre-fixed with a timestamp to help with ordering. Examine the *{DateStamp}_Initial.cs* file, it contains the instructions to create the `Movies` table for the Movie DB. When you update the database in the instructions below, this *{DateStamp}_Initial.cs* file will run and create the DB schema. Then the **Seed** method will run to populate the DB with test data.

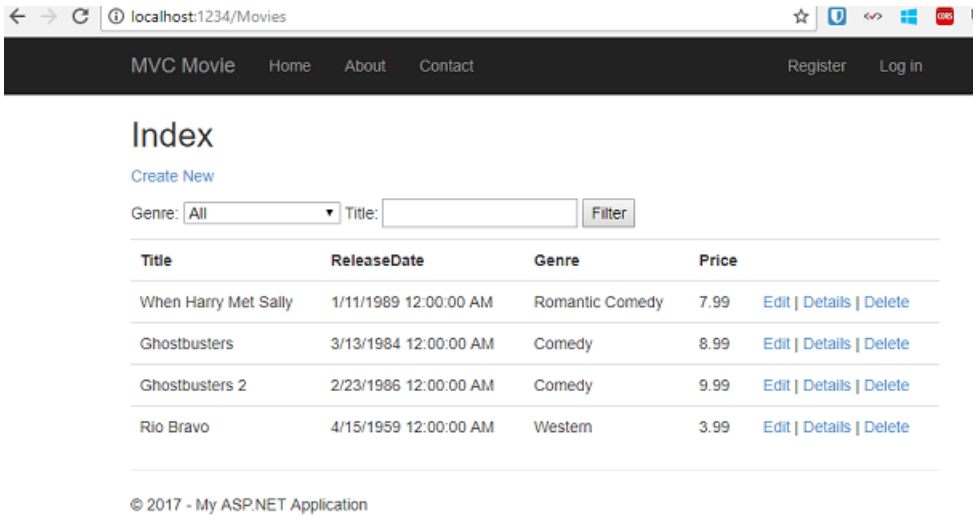
In the **Package Manager Console**, enter the command `update-database` to create the database and run the

`Seed` method.



If you get an error that indicates a table already exists and can't be created, it is probably because you ran the application after you deleted the database and before you executed `update-database`. In that case, delete the *Movies.mdf* file again and retry the `update-database` command. If you still get an error, delete the migrations folder and contents then start with the instructions at the top of this page (that is delete the *Movies.mdf* file then proceed to Enable-Migrations). If you still get an error, open SQL Server Object Explorer and remove the database from the list.

Run the application and navigate to the */Movies* URL. The seed data is displayed.



Adding a Rating Property to the Movie Model

Start by adding a new `Rating` property to the existing `Movie` class. Open the *Models\Movie.cs* file and add the `Rating` property like this one:

```
public string Rating { get; set; }
```

The complete `Movie` class now looks like the following code:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

Build the application (Ctrl+Shift+B).

Because you've added a new field to the `Movie` class, you also need to update the binding *white list* so this new property will be included. Update the `bind` attribute for `Create` and `Edit` action methods to include the `Rating` property:

```
[Bind(Include = "ID,Title,ReleaseDate,Genre,Price,Rating")]
```

You also need to update the view templates in order to display, create and edit the new `Rating` property in the browser view.

Open the *\Views\Movies\Index.cshtml* file and add a `<th>Rating</th>` column heading just after the **Price** column. Then add a `<td>` column near the end of the template to render the `@item.Rating` value. Below is what the updated *Index.cshtml* view template looks like:

```
@model IEnumerable<MvcMovie.Models.Movie>
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
<p>
    @Html.ActionLink("Create New", "Create")
    @using (Html.BeginForm("Index", "Movies", FormMethod.Get))
    {
    <p>
        Genre: @Html.DropDownList("movieGenre", "All")
        Title: @Html.TextBox("SearchString")
        <input type="submit" value="Filter" />
    </p>
    }
</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Genre)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Price)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Rating)
        </th>

        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Rating)
        </td>

        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
    </tr>
}

</table>
```

Next, open the *\Views\Movies\Create.cshtml* file and add the `Rating` field with the following highlighted markup.

This renders a text box so that you can specify a rating when a new movie is created.

```
<div class="form-group">
            @Html.LabelFor(model => model.Price, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Price, new { htmlAttributes = new { @class = "form-control" }
})
                @Html.ValidationMessageFor(model => model.Price, "", new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Rating, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Rating, new { htmlAttributes = new { @class = "form-control" }
})
                @Html.ValidationMessageFor(model => model.Rating, "", new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </div>
    </div>
}

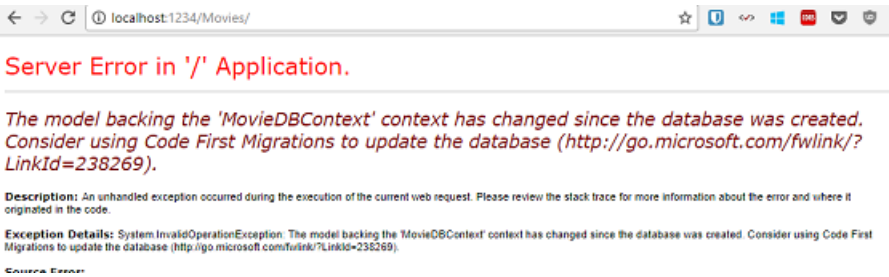
<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

You've now updated the application code to support the new `Rating` property.

Run the application and navigate to the */Movies* URL. When you do this, though, you'll see one of the following errors:

The model backing the 'MovieDBContext' context has changed since the database was created. Consider using Code First Migrations to update the database (https://go.microsoft.com/fwlink/?LinkId=238269).



You're seeing this error because the updated `Movie` model class in the application is now different than the schema of the `Movie` table of the existing database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the development cycle when you are doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you *don't* want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develop an application. For more information on Entity Framework database initializers, see ASP.NET MVC/Entity Framework tutorial.
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

For this tutorial, we'll use Code First Migrations.

Update the Seed method so that it provides a value for the new column. Open Migrations\Configuration.cs file and add a Rating field to each Movie object.

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "PG",
    Price = 7.99M
},
```

Build the solution, and then open the **Package Manager Console** window and enter the following command:

```
add-migration Rating
```

The `add-migration` command tells the migration framework to examine the current movie model with the current movie DB schema and create the necessary code to migrate the DB to the new model. The name *Rating* is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration step.

When this command finishes, Visual Studio opens the class file that defines the new `DbMigration` derived class, and in the `Up` method you can see the code that creates the new column.

```
public partial class AddRatingMig : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.Movies", "Rating", c => c.String());
    }

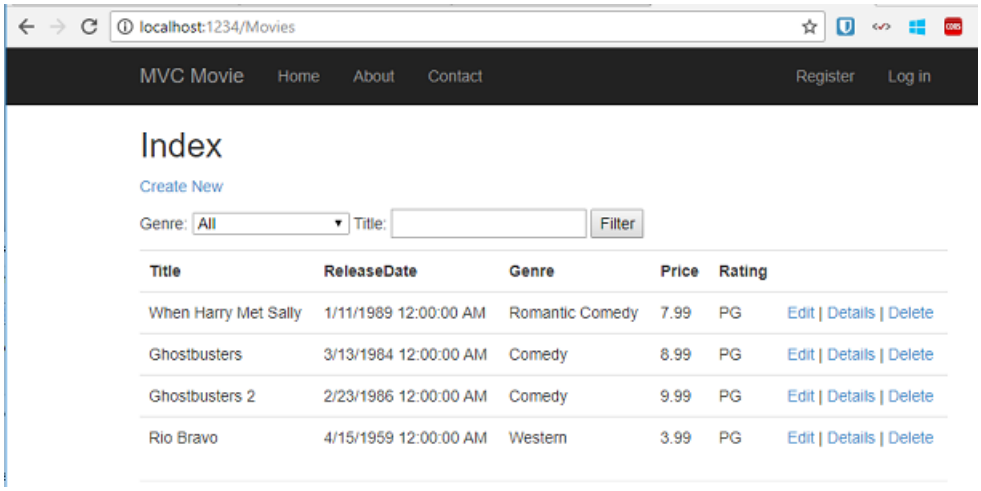
    public override void Down()
    {
        DropColumn("dbo.Movies", "Rating");
    }
}
```

Build the solution, and then enter the `update-database` command in the **Package Manager Console** window.

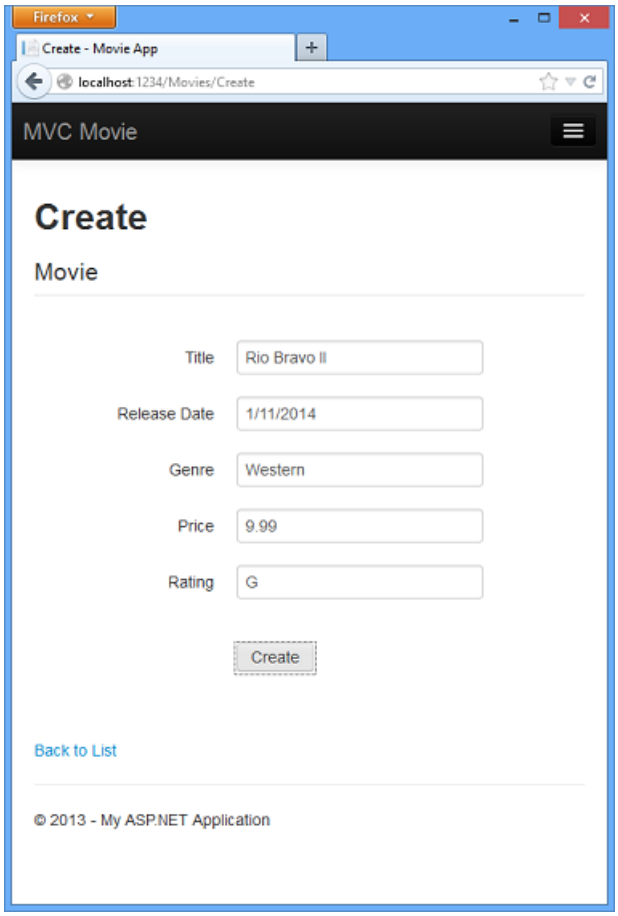
The following image shows the output in the **Package Manager Console** window (The date stamp prepending *Rating* will be different.)

```
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Applying explicit migrations: [201711171332278_Rating].
Applying explicit migration: 201711171332278_Rating.
Running Seed method.
PM>
```

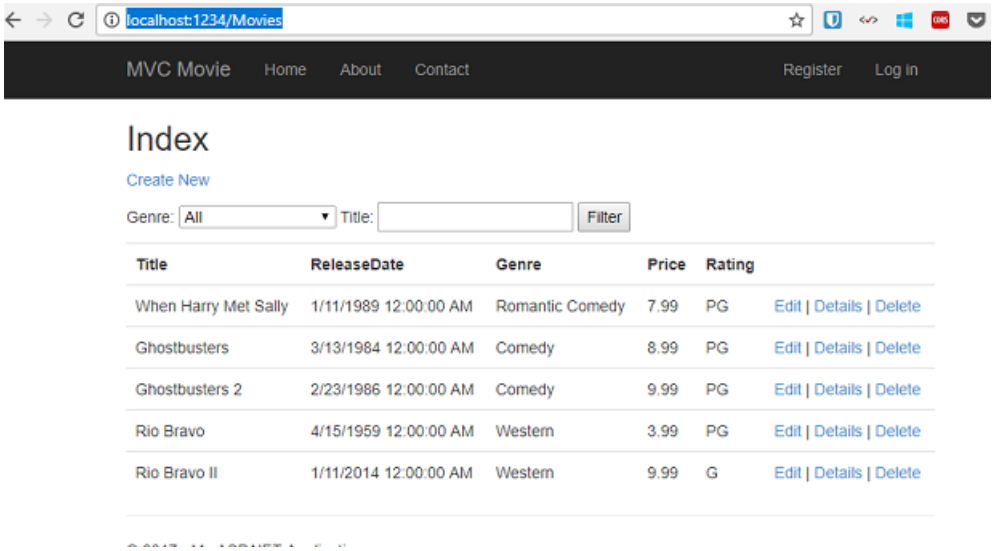
Re-run the application and navigate to the /Movies URL. You can see the new Rating field.



Click the **Create New** link to add a new movie. Note that you can add a rating.



Click **Create**. The new movie, including the rating, now shows up in the movies listing:



Now that the project is using migrations, you won't need to drop the database when you add a new field or otherwise update the schema. In the next section, we'll make more schema changes and use migrations to update the database.

You should also add the `Rating` field to the Edit, Details, and Delete view templates.

You could enter the "update-database" command in the **Package Manager Console** window again and no migration code would run, because the schema matches the model. However, running "update-database" will run the `Seed` method again, and if you changed any of the Seed data, the changes will be lost because the `Seed` method upserts data. You can read more about the `Seed` method in Tom Dykstra's popular ASP.NET MVC/Entity Framework tutorial.

In this section you saw how you can modify model objects and keep the database in sync with the changes. You also learned a way to populate a newly created database with sample data so you can try out scenarios. This was

just a quick introduction to Code First, see Creating an Entity Framework Data Model for an ASP.NET MVC Application for a more complete tutorial on the subject. Next, let's look at how you can add richer validation logic to the model classes and enable some business rules to be enforced.

PREVIOUS NEXT

Adding Validation

9/16/2020 • 13 minutes to read • Edit Online

by Rick Anderson

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

In this section you'll add validation logic to the `Movie` model, and you'll ensure that the validation rules are enforced any time a user attempts to create or edit a movie using the application.

Keeping Things DRY

One of the core design tenets of ASP.NET MVC is DRY ("Don't Repeat Yourself"). ASP.NET MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an application. This reduces the amount of code you need to write and makes the code you do write less error prone and easier to maintain.

The validation support provided by ASP.NET MVC and Entity Framework Code First is a great example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the application.

Let's look at how you can take advantage of this validation support in the movie application.

Adding Validation Rules to the Movie Model

You'll begin by adding some validation logic to the `Movie` class.

Open the *Movie.cs* file. Notice the `System.ComponentModel.DataAnnotations` namespace does not contain `System.Web`. DataAnnotations provides a built-in set of validation attributes that you can apply declaratively to any class or property. (It also contains formatting attributes like DataType that help with formatting and don't provide any validation.)

Now update the `Movie` class to take advantage of the built-in `Required`, `StringLength`, RegularExpression, and `Range` validation attributes. Replace the `Movie` class with the following:

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

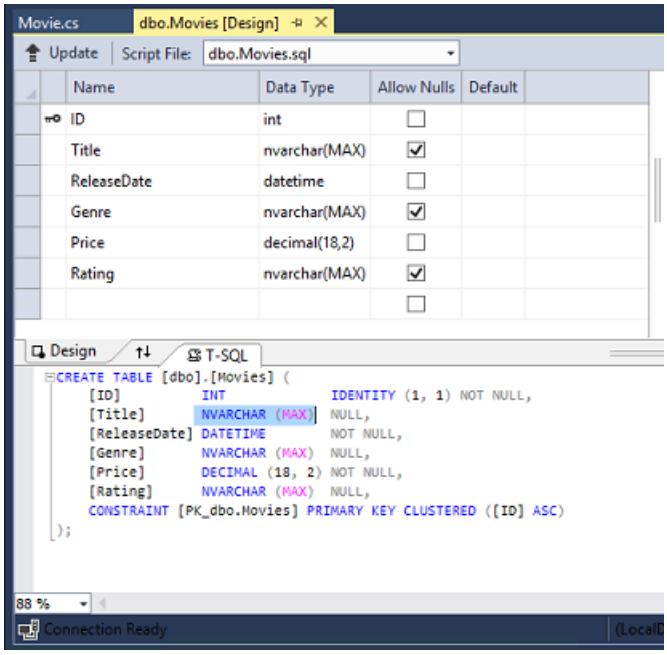
    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z'\s]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z'\s]*$")]
    [StringLength(5)]
    public string Rating { get; set; }
}
```

The `StringLength` attribute sets the maximum length of the string, and it sets this limitation on the database, therefore the database schema will change. Right click on the **Movies** table in **Server explorer** and click **Open Table Definition**:



In the image above, you can see all the string fields are set to NVARCHAR (MAX). We will use migrations to update the schema. Build the solution, and then open the **Package Manager Console** window and enter the following commands:

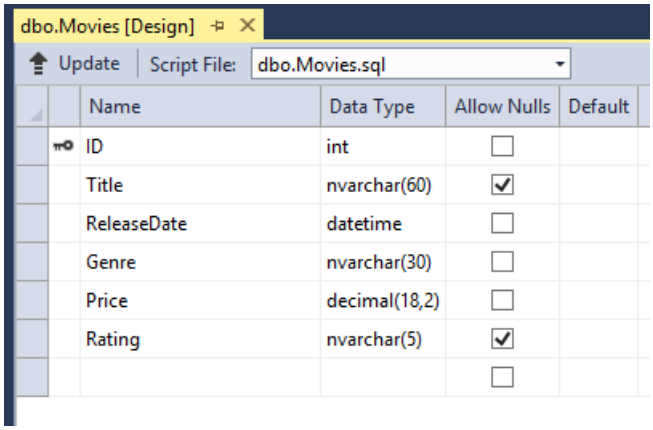
```
add-migration DataAnnotations
update-database
```

When this command finishes, Visual Studio opens the class file that defines the new `DbMigration` derived class with the name specified (`DataAnnotations`), and in the `Up` method you can see the code that updates the schema constraints:

```
public override void Up()
{
    AlterColumn("dbo.Movies", "Title", c => c.String(maxLength: 60));
    AlterColumn("dbo.Movies", "Genre", c => c.String(nullable: false, maxLength: 30));
    AlterColumn("dbo.Movies", "Rating", c => c.String(maxLength: 5));
}
```

The `Genre` field is no longer nullable (that is, you must enter a value). The `Rating` field has a maximum length of 5 and `Title` has a maximum length of 60. The minimum length of 3 on `Title` and the range on `Price` did not create schema changes.

Examine the Movie schema:



The string fields show the new length limits and `Genre` is no longer checked as nullable.

The validation attributes specify behavior that you want to enforce on the model properties they are applied to. The `Required` and `MinimumLength` attributes indicates that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation. The RegularExpression attribute is used to limit what characters can be input. In the code above, `Genre` and `Rating` must use only letters (white space, numbers and special characters are not allowed). The `Range` attribute constrains a value to within a specified range. The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length. Value types (such as `decimal, int, float, DateTime`) are inherently required and don't need the `Required` attribute.

Code First ensures that the validation rules you specify on a model class are enforced before the application saves changes in the database. For example, the code below will throw a DbEntityValidationException exception when the `SaveChanges` method is called, because several required `Movie` property values are missing:

```
MovieDBContext db = new MovieDBContext();
Movie movie = new Movie();
movie.Title = "Gone with the Wind";
db.Movies.Add(movie);
db.SaveChanges();        // <= Will throw server side validation exception
```

The code above throws the following exception:

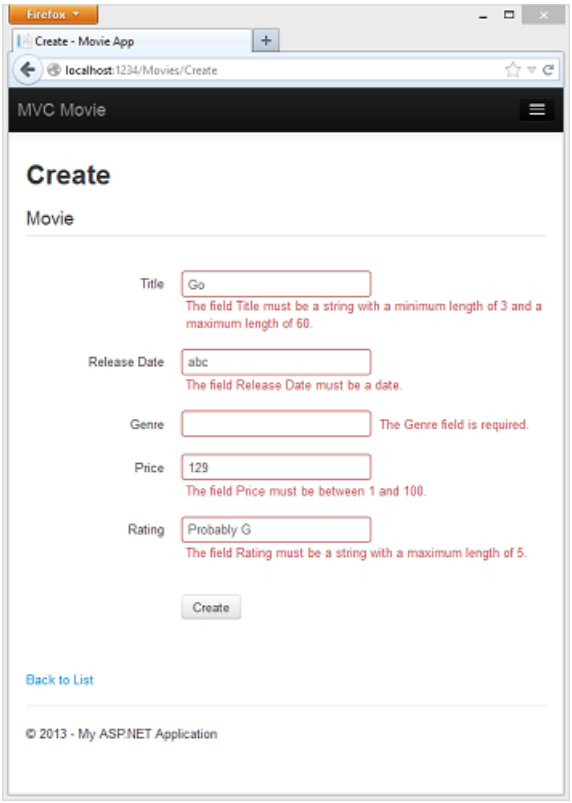
Validation failed for one or more entities. See 'EntityValidationErrors' property for more details.

Having validation rules automatically enforced by the .NET Framework helps make your application more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI in ASP.NET MVC

Run the application and navigate to the */Movies* URL.

Click the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.



> **NOTE**
> to support jQuery validation for non-English locales that use a comma (",") for a decimal point, you must include the NuGet globalize as described previously in this tutorial.

Notice how the form has automatically used a red border color to highlight the text boxes that contain invalid data and has emitted an appropriate validation error message next to each one. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

A real benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the *Create.cshtml* view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the `Movie` model class. Test validation using the `Edit` action method, and the same validation is applied.

The form data is not sent to the server until there are no client side validation errors. You can verify this by putting a break point in the HTTP Post method, by using the fiddler tool, or the IE F12 developer tools.

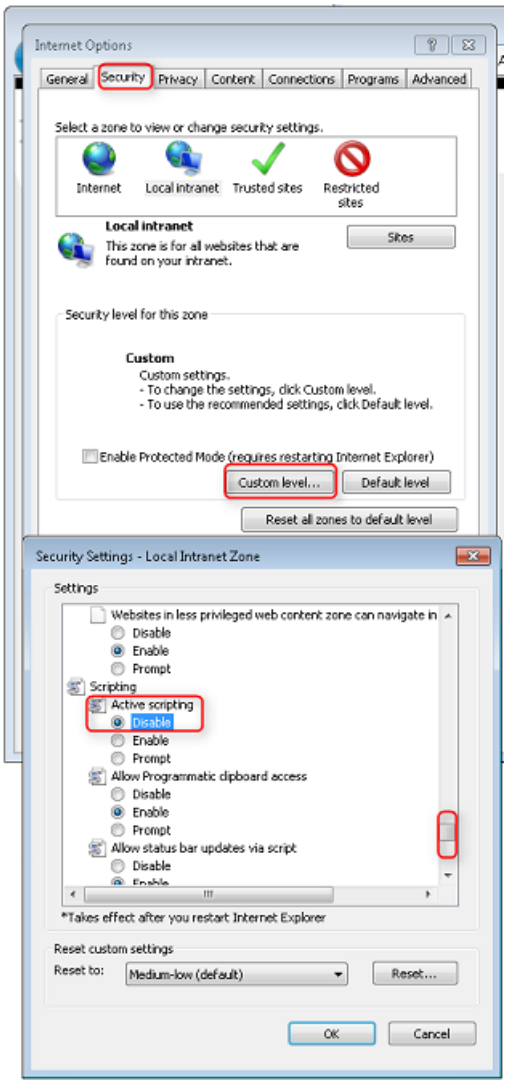
How Validation Occurs in the Create View and Create Action Method

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The next listing shows what the `Create` methods in the `MovieController` class look like. They're unchanged from how you created them earlier in this tutorial.

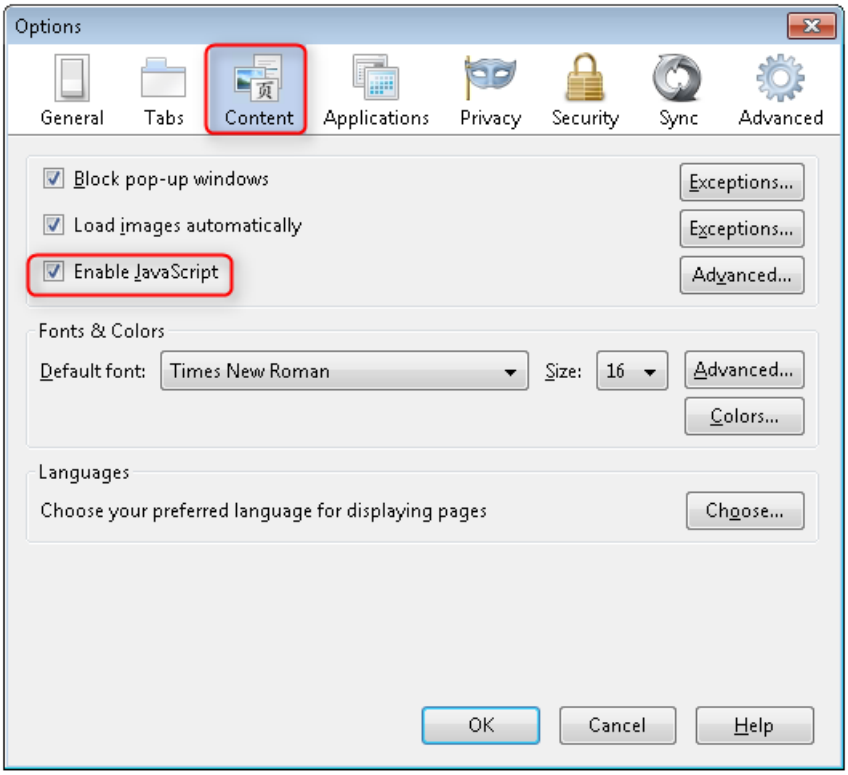
```
public ActionResult Create()
{
    return View();
}
// POST: /Movies/Create
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see https://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "ID,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Movies.Add(movie);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The first (HTTP GET) `Create` action method displays the initial Create form. The second (`[HttpPost]`) version handles the form post. The second `Create` method (the `HttpPost` version) checks `ModelState.IsValid` to see whether the movie has any validation errors. Getting this property evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method redisplays the form. If there are no errors, the method saves the new movie in the database. In our movie example, **the form isn't posted to the server when there are validation errors detected on the client side; the second `Create` method is never called**. If you disable JavaScript in your browser, client validation is disabled, and the HTTP POST `Create` method gets `ModelState.IsValid` to check whether the movie has any validation errors.

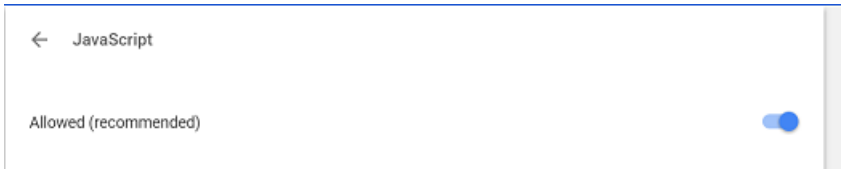
You can set a break point in the `HttpPost Create` method and verify the method is never called, client side validation will not submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript. The following image shows how to disable JavaScript in Internet Explorer.



The following image shows how to disable JavaScript in the FireFox browser.



The following image shows how to disable JavaScript in the Chrome browser.



Below is the *Create.cshtml* view template that you scaffolded earlier in the tutorial. It's used by the action methods shown above both to display the initial form and to redisplay it in the event of an error.

```
@model MvcMovie.Models.Movie
@{
    ViewBag.Title = "Create";
}
<h2>Create</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        @Html.ValidationSummary(true)
        <div class="form-group">
            @Html.LabelFor(model => model.Title, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Title)
                @Html.ValidationMessageFor(model => model.Title)
            </div>
        </div>
        @*Fields removed for brevity.*@




        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </div>
    </div>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Notice how the code uses an `Html.EditorFor` helper to output the `<input>` element for each `Movie` property. Next to this helper is a call to the `Html.ValidationMessageFor` helper method. These two helper methods work with the model object that's passed by the controller to the view (in this case, a `Movie` object). They automatically look for validation attributes specified on the model and display error messages as appropriate.

What's really nice about this approach is that neither the controller nor the `Create` view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the `Movie` class. These same validation rules are automatically applied to the `Edit` view and any other views templates you might create that edit your model.

If you want to change the validation logic later, you can do so in exactly one place by adding validation attributes to the model (in this example, the `movie` class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the *DRY* principle.

Using DataType Attributes

Open the *Movie.cs* file and examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a `DataType` enumeration value to the release date and to the price fields. The following code shows the `ReleaseDate` and `Price` properties with the appropriate `DataType` attribute.

```
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

The DataType attributes only provide hints for the view engine to format the data (and supply attributes such as `<a>` for URL's and `<a href="mailto:EmailAddress.com">` for email. You can use the RegularExpression attribute to validate the format of the data. The DataType attribute is used to specify a data type that is more specific than the database intrinsic type, they are ***not*** validation attributes. In this case we only want to keep track of the date, not the date and time. The DataType Enumeration provides for many data types, such as *Date, Time, PhoneNumber, Currency, EmailAddress* and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for DataType.EmailAddress, and a date selector can be provided for DataType.Date in browsers that support HTML5. The DataType attributes emits HTML 5 data- (pronounced *data dash*) attributes that HTML 5 browsers can understand. The DataType attributes do not provide any validation.

`DataType.Date` does not specify the format of the date that is displayed. By default, the data field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime EnrollmentDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the specified formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you might not want the currency symbol in the text box for editing.)

You can use the DisplayFormat attribute by itself, but it's generally a good idea to use the DataType attribute also. The `DataType` attribute conveys the *semantics* of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.).
- By default, the browser will render data using the correct format based on your locale.
- The DataType attribute can enable MVC to choose the right field template to render the data (the DisplayFormat if used by itself uses the string template). For more information, see Brad Wilson's ASP.NET MVC 2 Templates. (Though written for MVC 2, this article still applies to the current version of ASP.NET MVC.)

If you use the `DataType` attribute with a date field, you have to specify the `DisplayFormat` attribute also in order to ensure that the field renders correctly in Chrome browsers. For more information, see this StackOverflow thread.

NOTE

jQuery validation does not work with the Range attribute and DateTime. For example, the following code will always display a client side validation error, even when the date is in the specified range:

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

You will need to disable jQuery date validation to use the Range attribute with DateTime. It's generally not a good practice to compile hard dates in your models, so using the Range attribute and DateTime is discouraged.

The following code shows combining attributes on one line:

```
public class Movie
{
   public int ID { get; set; }
   [Required,StringLength(60, MinimumLength = 3)]
   public string Title { get; set; }
   [Display(Name = "Release Date"),DataType(DataType.Date)]
   public DateTime ReleaseDate { get; set; }
   [Required]
   public string Genre { get; set; }
   [Range(1, 100),DataType(DataType.Currency)]
   public decimal Price { get; set; }
   [Required,StringLength(5)]
   public string Rating { get; set; }
}
```

In the next part of the series, we'll review the application and make some improvements to the automatically generated `Details` and `Delete` methods.

PREVIOUS NEXT

Examining the Details and Delete Methods

9/16/2020 • 4 minutes to read • Edit Online

by Rick Anderson

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

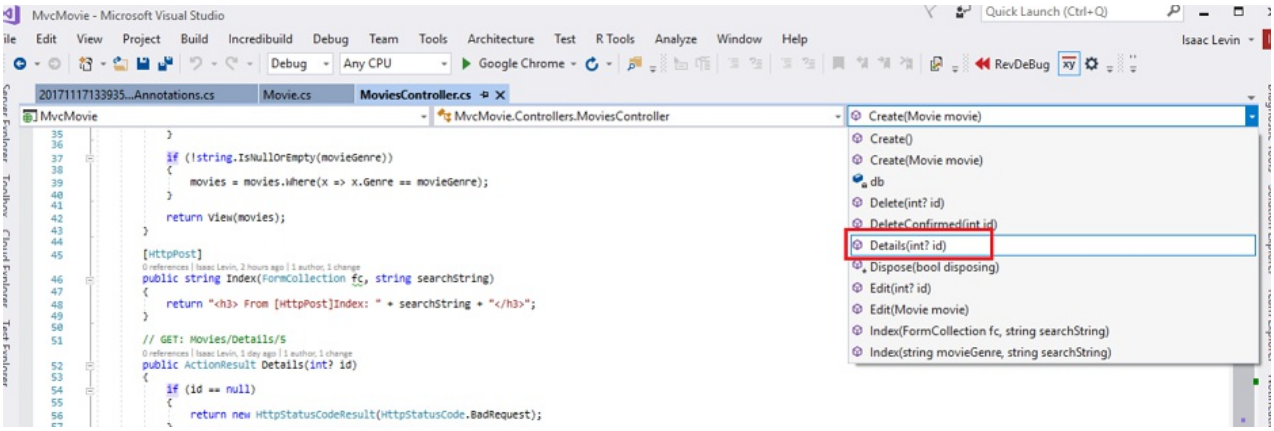
This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

In this part of the tutorial, you'll examine the automatically generated `Details` and `Delete` methods.

Examining the Details and Delete Methods

Open the `Movie` controller and examine the `Details` method.




```
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}
```

The MVC scaffolding engine that created this action method adds a comment showing a HTTP request that invokes the method. In this case it's a `GET` request with three URL segments, the `Movies` controller, the `Details` method and a `ID` value.

Code First makes it easy to search for data using the `Find` method. An important security feature built into the

method is that the code verifies that the `Find` method has found a movie before the code tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from `http://localhost:xxxx/Movies/Details/1` to something like `http://localhost:xxxx/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you did not check for a null movie, a null movie would result in a database error.

Examine the `Delete` and `DeleteConfirmed` methods.

```
// GET: /Movies/Delete/5
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}

// POST: /Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
    Movie movie = db.Movies.Find(id);
    db.Movies.Remove(movie);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

Note that the HTTP GET `Delete` method doesn't delete the specified movie, it returns a view of the movie where you can submit (`HttpPost`) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole. For more information about this, see Stephen Walther's blog entry ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes.

The `HttpPost` method that deletes the data is named `DeleteConfirmed` to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

```
// GET: /Movies/Delete/5
public ActionResult Delete(int? id)

//
// POST: /Movies/Delete/5
[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
```

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two Delete methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

To sort this out, you can do a couple of things. One is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to

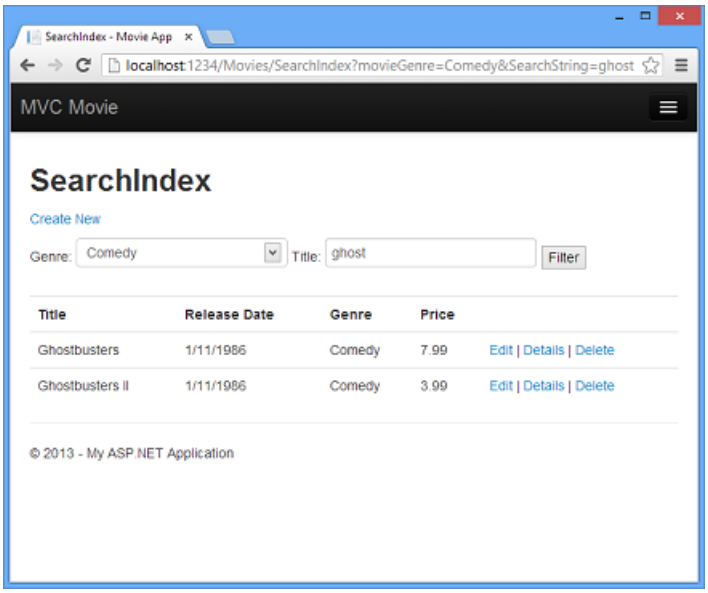
find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. This effectively performs mapping for the routing system so that a URL that includes */Delete/* for a POST request will find the `DeleteConfirmed` method.

Another common way to avoid a problem with methods that have identical names and signatures is to artificially change the signature of the POST method to include an unused parameter. For example, some developers add a parameter type `FormCollection` that is passed to the POST method, and then simply don't use the parameter:

```
public ActionResult Delete(FormCollection fcNotUsed, int id = 0)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    db.Movies.Remove(movie);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

Summary

You now have a complete ASP.NET MVC application that stores data in a local DB database. You can create, read, update, delete, and search for movies.



Next Steps

After you have built and tested a web application, the next step is to make it available to other people to use over the Internet. To do that, you have to deploy it to a web hosting provider. Microsoft offers free web hosting for up to 10 web sites in a free Azure trial account. I suggest you next follow my tutorial Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL Database to Azure. An excellent tutorial is Tom Dykstra's intermediate-level Creating an Entity Framework Data Model for an ASP.NET MVC Application. Stackoverflow and the ASP.NET MVC forums are a great places to ask questions. Follow me on twitter so you can get updates on my latest tutorials.

Feedback is welcome.

— Rick Anderson twitter: @RickAndMSFT
— Scott Hanselman twitter: @shanselman

Getting Started with Entity Framework 6 Code First using MVC 5

2/20/2020 • 2 minutes to read • Edit Online

The basics of using Entity Framework 6 to display and edit data in an ASP.NET MVC 5 application.

- Creating an Entity Framework Data Model
- Implementing Basic CRUD Functionality
- Sorting, Filtering, and Paging
- Connection Resiliency and Command Interception
- Code First Migrations and Deployment
- Creating a More Complex Data Model
- Reading Related Data
- Updating Related Data
- Async and Stored Procedures
- Handling Concurrency
- Implementing Inheritance
- Advanced Entity Framework Scenarios

Tutorial: Get Started with Entity Framework 6 Code First using MVC 5

2/20/2020 • 19 minutes to read • Edit Online

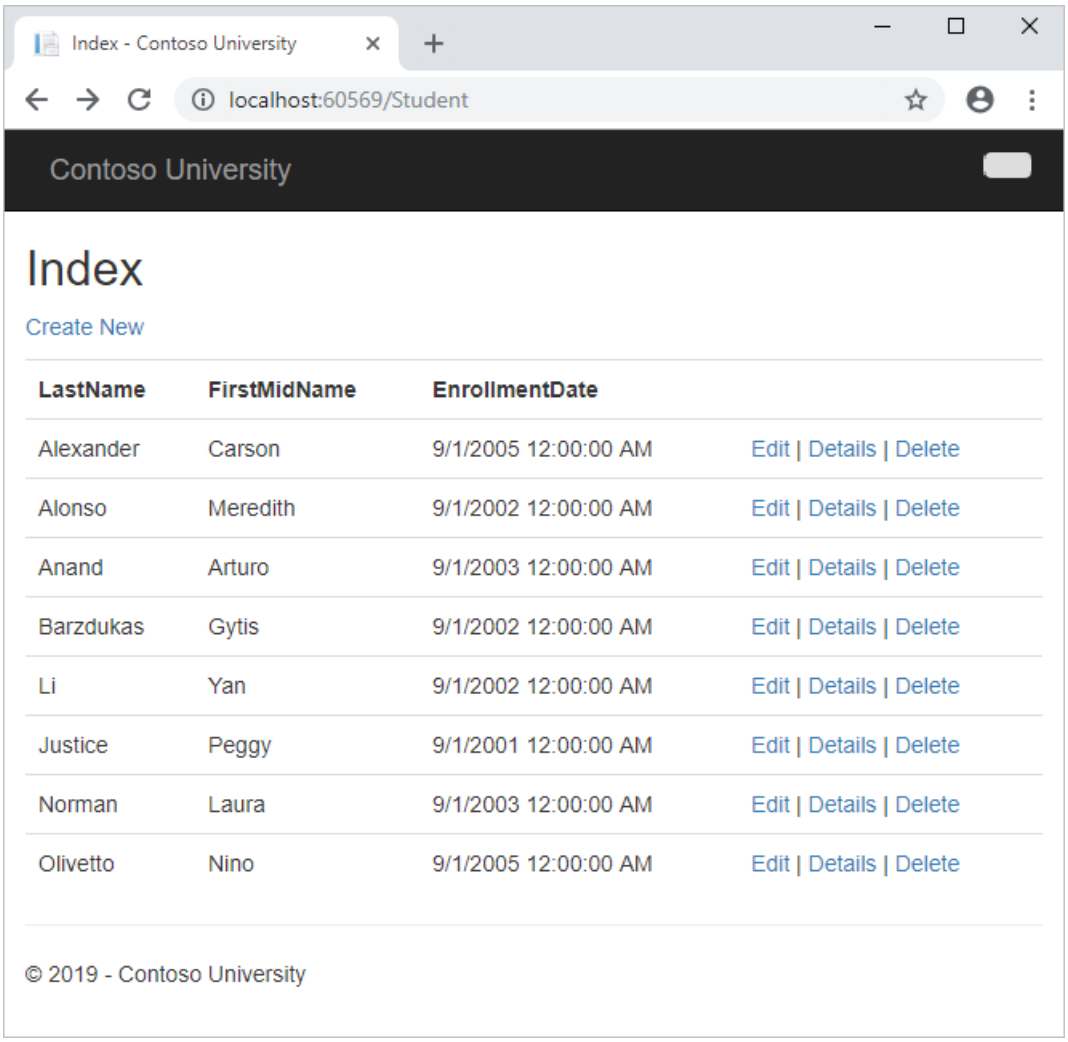
NOTE

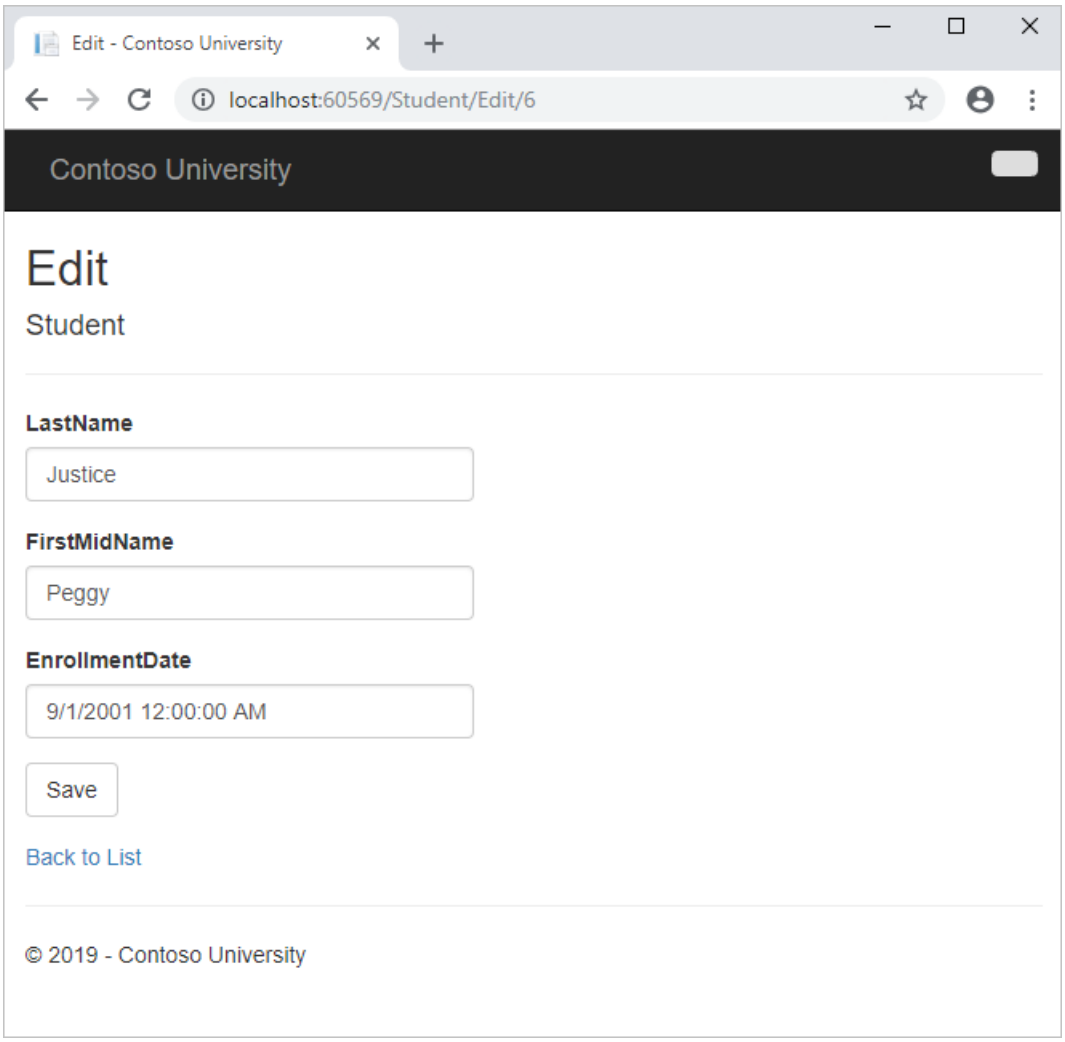
For new development, we recommend ASP.NET Core Razor Pages over ASP.NET MVC controllers and views. For a tutorial series similar to this one using Razor Pages, see Tutorial: Get started with Razor Pages in ASP.NET Core. The new tutorial:

- Is easier to follow.
- Provides more EF Core best practices.
- Uses more efficient queries.
- Is more current with the latest API.
- Covers more features.
- Is the preferred approach for new application development.

In this series of tutorials, you learn how to build an ASP.NET MVC 5 application that uses Entity Framework 6 for data access. This tutorial uses the Code First workflow. For information about how to choose between Code First, Database First, and Model First, see Create a model.

This tutorial series explains how to build the Contoso University sample application. The sample application is a simple university website. With it, you can view and update student, course, and instructor information. Here are two of the screens you create:


Index - Contoso University
localhost:60569/Student
Contoso University
Index
Create New
LastName
FirstMidName
EnrollmentDate
Alexander Carson 9/1/2005 12:00:00 AM Edit | Details | Delete
Alonso Meredith 9/1/2002 12:00:00 AM Edit | Details | Delete
Anand Arturo 9/1/2003 12:00:00 AM Edit | Details | Delete
Barzdukas Gytis 9/1/2002 12:00:00 AM Edit | Details | Delete
Li Yan 9/1/2002 12:00:00 AM Edit | Details | Delete
Justice Peggy 9/1/2001 12:00:00 AM Edit | Details | Delete
Norman Laura 9/1/2003 12:00:00 AM Edit | Details | Delete
Olivetto Nino 9/1/2005 12:00:00 AM Edit | Details | Delete
© 2019 - Contoso University



In this tutorial, you:

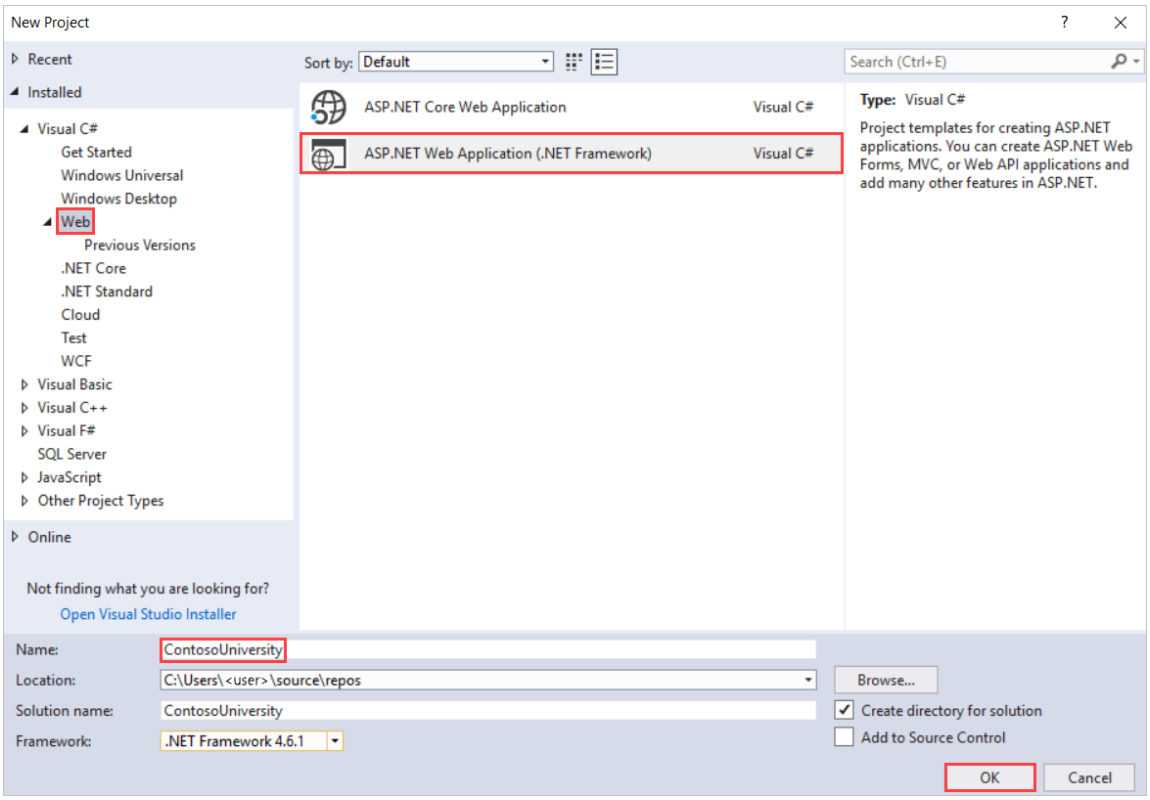
- Create an MVC web app
- Set up the site style
- Install Entity Framework 6
- Create the data model
- Create the database context
- Initialize DB with test data
- Set up EF 6 to use LocalDB
- Create controller and views
- View the database

Prerequisites

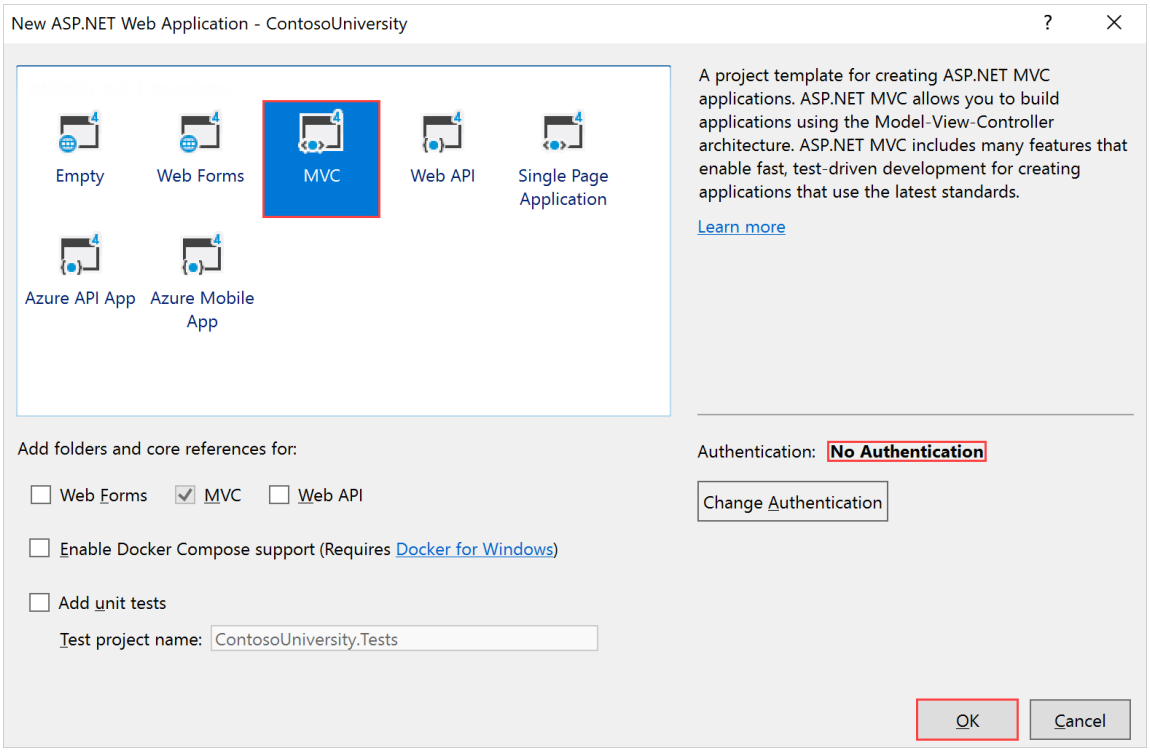
- Visual Studio 2017

Create an MVC web app

1. Open Visual Studio and create a C# web project using the **ASP.NET Web Application (.NET Framework)** template. Name the project *ContosoUniversity* and select **OK**.



2. In **New ASP.NET Web Application - ContosoUniversity**, select **MVC**.



> **NOTE**
>
> By default, the **Authentication** option is set to **No Authentication**. For this tutorial, the web app doesn't require users to sign in. Also, it doesn't restrict access based on who's signed in.

3. Select **OK** to create the project.

Set up the site style

A few simple changes will set up the site menu, layout, and home page.

1. Open *Views\Shared_Layout.cshtml*, and make the following changes:

   - Change each occurrence of "My ASP.NET Application" and "Application name" to "Contoso University".
   - Add menu entries for Students, Courses, Instructors, and Departments, and delete the Contact entry.

   The changes are highlighted in the following code snippet:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - Contoso University</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="navbar-inner">
            <div class="container">
                <button type="button" class="btn btn-navbar" data-toggle="collapse" data-
target=".nav-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Contoso University", "Index", "Home", new { area = "" }, new {
@class = "navbar-brand" })
                <div class="nav-collapse collapse">
                    <ul class="nav">
                        <li>@Html.ActionLink("Home", "Index", "Home")</li>
                        <li>@Html.ActionLink("About", "About", "Home")</li>
                        <li>@Html.ActionLink("Students", "Index", "Student")</li>
                        <li>@Html.ActionLink("Courses", "Index", "Course")</li>
                        <li>@Html.ActionLink("Instructors", "Index", "Instructor")</li>
                        <li>@Html.ActionLink("Departments", "Index", "Department")</li>
                    </ul>
                </div>
            </div>
        </div>
    </div>

    <div class="container">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - Contoso University</p>
        </footer>
    </div>

    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/bootstrap")
    @RenderSection("scripts", required: false)
</body>
</html>
```

2. In *Views\Home\Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET and MVC with text about this application:

```
@{
    ViewBag.Title = "Home Page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>Contoso University is a sample application that
        demonstrates how to use Entity Framework 6 in an
        ASP.NET MVC 5 web application.</p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in the tutorial series on the
ASP.NET site.</p>
        <p><a class="btn btn-default" href="http://www.asp.net/mvc/tutorials/getting-started-with-
ef-using-mvc/">See the tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project.</p>
        <p><a class="btn btn-default"
href="https://webpifeed.blob.core.windows.net/webpifeed/Partners/ASP.NET%20MVC%20Application%20Usin
g%20Entity%20Framework%20Code%20First.zip">Download &raquo;</a></p>
    </div>
</div>
```

3. Press Ctrl+F5 to run the web site. You see the home page with the main menu.

Install Entity Framework 6

1. From the **Tools** menu, choose **NuGet Package Manager**, and then choose **Package Manager Console**.

2. In the **Package Manager Console** window, enter the following command:

```
Install-Package EntityFramework
```

This step is one of a few steps that this tutorial has you do manually, but that could have been done automatically by the ASP.NET MVC scaffolding feature. You're doing them manually so that you can see the steps required to use Entity Framework (EF). You'll use scaffolding later to create the MVC controller and views. An alternative is to let scaffolding automatically install the EF NuGet package, create the database context class, and create the connection string. When you're ready to do it that way, all you have to do is skip those steps and scaffold your MVC controller after you create your entity classes.

Create the data model

Next you'll create entity classes for the Contoso University application. You'll start with the following three entities:

Course <-> **Enrollment** <-> **Student**

ENTITIES	RELATIONSHIP
Course to Enrollment	One-to-many

ENTITIES	RELATIONSHIP
Student to Enrollment	One-to-many

There's a one-to-many relationship between `Student` and `Enrollment` entities, and there's a one-to-many relationship between `Course` and `Enrollment` entities. In other words, a student can be enrolled in any number of courses, and a course can have any number of students enrolled in it.

In the following sections, you'll create a class for each one of these entities.

> **NOTE**
> If you try to compile the project before you finish creating all of these entity classes, you'll get compiler errors.

The Student entity

- In the *Models* folder, create a class file named *Student.cs* by right-clicking on the folder in **Solution Explorer** and choosing **Add** > **Class**. Replace the template code with the following code:

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `ID` property will become the primary key column of the database table that corresponds to this class. By default, Entity Framework interprets a property that's named `ID` or *classname* `ID` as the primary key.

The `Enrollments` property is a *navigation property*. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity will hold all of the `Enrollment` entities that are related to that `Student` entity. In other words, if a given `Student` row in the database has two related `Enrollment` rows (rows that contain that student's primary key value in their `StudentID` foreign key column), that `Student` entity's `Enrollments` navigation property will contain those two `Enrollment` entities.

Navigation properties are typically defined as `virtual` so that they can take advantage of certain Entity Framework functionality such as *lazy loading*. (Lazy loading will be explained later, in the Reading Related Data tutorial later in this series.)

If a navigation property can hold multiple entities (as in many-to-many or one-to-many relationships), its type must be a list in which entries can be added, deleted, and updated, such as `ICollection`.

The Enrollment entity

- In the *Models* folder, create *Enrollment.cs* and replace the existing code with the following code:

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public virtual Course Course { get; set; }
        public virtual Student Student { get; set; }
    }
}
```

The `EnrollmentID` property will be the primary key; this entity uses the *classname* `ID` pattern instead of `ID` by itself as you saw in the `Student` entity. Ordinarily you would choose one pattern and use it throughout your data model. Here, the variation illustrates that you can use either pattern. In a later tutorial, you'll see how using `ID` without `classname` makes it easier to implement inheritance in the data model.

The `Grade` property is an enum. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade — null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property can only hold a single `Student` entity (unlike the `Student.Enrollments` navigation property you saw earlier, which can hold multiple `Enrollment` entities).

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

Entity Framework interprets a property as a foreign key property if it's named *<navigation property name>* *<primary key property name>* (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named the same simply *<primary key property name>* (for example, `CourseID` since the `Course` entity's primary key is `CourseID`).

The Course entity

- In the *Models* folder, create *Course.cs*, replacing the template code with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

We'll say more about the DatabaseGeneratedAttribute attribute in a later tutorial in this series. Basically, this attribute lets you enter the primary key for the course rather than having the database generate it.

Create the database context

The main class that coordinates Entity Framework functionality for a given data model is the *database context* class. You create this class by deriving from the System.Data.Entity.DbContext class. In your code, you specify which entities are included in the data model. You can also customize certain Entity Framework behavior. In this project, the class is named `SchoolContext`.

- To create a folder in the ContosoUniversity project, right-click the project in **Solution Explorer** and click **Add**, and then click **New Folder**. Name the new folder *DAL* (for Data Access Layer). In that folder, create a new class file named *SchoolContext.cs*, and replace the template code with the following code:

```
using ContosoUniversity.Models;
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace ContosoUniversity.DAL
{
    public class SchoolContext : DbContext
    {

        public SchoolContext() : base("SchoolContext")
        {
        }

        public DbSet<Student> Students { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Course> Courses { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
        }
    }
}
```

Specify entity sets

This code creates a DbSet property for each entity set. In Entity Framework terminology, an *entity set* typically corresponds to a database table, and an *entity* corresponds to a row in the table.

> **NOTE**
> You can omit the `DbSet<Enrollment>` and `DbSet<Course>` statements and it would work the same. Entity Framework would include them implicitly because the `Student` entity references the `Enrollment` entity and the `Enrollment` entity references the `Course` entity.

Specify the connection string

The name of the connection string (which you'll add to the Web.config file later) is passed in to the constructor.

```
public SchoolContext() : base("SchoolContext")
{
}
```

You could also pass in the connection string itself instead of the name of one that is stored in the Web.config file. For more information about options for specifying the database to use, see Connection strings and models.

If you don't specify a connection string or the name of one explicitly, Entity Framework assumes that the connection string name is the same as the class name. The default connection string name in this example would then be `SchoolContext`, the same as what you're specifying explicitly.

Specify singular table names

The `modelBuilder.Conventions.Remove` statement in the OnModelCreating method prevents table names from being pluralized. If you didn't do this, the generated tables in the database would be named `Students`, `Courses`, and `Enrollments`. Instead, the table names will be `Student`, `Course`, and `Enrollment`. Developers disagree about whether table names should be pluralized or not. This tutorial uses the singular form, but the important point is that you can select whichever form you prefer by including or omitting this line of code.

Initialize DB with test data

Entity Framework can automatically create (or drop and re-create) a database for you when the application runs. You can specify that this should be done every time your application runs or only when the model is out of sync with the existing database. You can also write a `Seed` method that Entity Framework automatically calls after creating the database in order to populate it with test data.

The default behavior is to create a database only if it doesn't exist (and throw an exception if the model has changed and the database already exists). In this section, you'll specify that the database should be dropped and re-created whenever the model changes. Dropping the database causes the loss of all your data. This is generally okay during development, because the `Seed` method will run when the database is re-created and will re-create your test data. But in production you generally don't want to lose all your data every time you need to change the database schema. Later you'll see how to handle model changes by using Code First Migrations to change the database schema instead of dropping and re-creating the database.

1. In the DAL folder, create a new class file named *SchoolInitializer.cs* and replace the template code with the following code, which causes a database to be created when needed and loads test data into the new database.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using ContosoUniversity.Models;

namespace ContosoUniversity.DAL
{
    public class SchoolInitializer : System.Data.Entity.
DropCreateDatabaseIfModelChanges<SchoolContext>
    {
        protected override void Seed(SchoolContext context)
        {
            var students = new List<Student>
            {
            new
Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2005-09-01")},
            new
Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2002-09-01")},
            new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003-
09-01")},
            new
Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2002-09-01")},
            new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-
01")},
```

```
            new
Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2001-09-01")},
            new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003-
09-01")},
            new
Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-09-01")}
            };

            students.ForEach(s => context.Students.Add(s));
            context.SaveChanges();
            var courses = new List<Course>
            {
            new Course{CourseID=1050,Title="Chemistry",Credits=3,},
            new Course{CourseID=4022,Title="Microeconomics",Credits=3,},
            new Course{CourseID=4041,Title="Macroeconomics",Credits=3,},
            new Course{CourseID=1045,Title="Calculus",Credits=4,},
            new Course{CourseID=3141,Title="Trigonometry",Credits=4,},
            new Course{CourseID=2021,Title="Composition",Credits=3,},
            new Course{CourseID=2042,Title="Literature",Credits=4,}
            };
            courses.ForEach(s => context.Courses.Add(s));
            context.SaveChanges();
            var enrollments = new List<Enrollment>
            {
            new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
            new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
            new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
            new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
            new Enrollment{StudentID=3,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=1050,},
            new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
            new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
            new Enrollment{StudentID=6,CourseID=1045},
            new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };
            enrollments.ForEach(s => context.Enrollments.Add(s));
            context.SaveChanges();
        }
    }
}
```

The `Seed` method takes the database context object as an input parameter, and the code in the method uses that object to add new entities to the database. For each entity type, the code creates a collection of new entities, adds them to the appropriate `DbSet` property, and then saves the changes to the database. It isn't necessary to call the `SaveChanges` method after each group of entities, as is done here, but doing that helps you locate the source of a problem if an exception occurs while the code is writing to the database.

2. To tell Entity Framework to use your initializer class, add an element to the `entityFramework` element in the application *Web.config* file (the one in the root project folder), as shown in the following example:

```
<entityFramework>
  <contexts>
    <context type="ContosoUniversity.DAL.SchoolContext, ContosoUniversity">
      <databaseInitializer type="ContosoUniversity.DAL.SchoolInitializer, ContosoUniversity" />
    </context>
  </contexts>
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
EntityFramework">
    <parameters>
      <parameter value="v11.0" />
    </parameters>
  </defaultConnectionFactory>
  <providers>
    <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
  </providers>
</entityFramework>
```

The `context type` specifies the fully qualified context class name and the assembly it's in, and the `databaseinitializer type` specifies the fully qualified name of the initializer class and the assembly it's in. (When you don't want EF to use the initializer, you can set an attribute on the `context` element: `disableDatabaseInitialization="true"`.) For more information, see Configuration File Settings.

An alternative to setting the initializer in the *Web.config* file is to do it in code by adding a `Database.SetInitializer` statement to the `Application_Start` method in the *Global.asax.cs* file. For more information, see Understanding Database Initializers in Entity Framework Code First.

The application is now set up so that when you access the database for the first time in a given run of the application, Entity Framework compares the database to the model (your `SchoolContext` and entity classes). If there's a difference, the application drops and re-creates the database.

> **NOTE**
> When you deploy an application to a production web server, you must remove or disable code that drops and re-creates the database. You'll do that in a later tutorial in this series.

Set up EF 6 to use LocalDB

LocalDB is a lightweight version of the SQL Server Express database engine. It's easy to install and configure, starts on demand, and runs in user mode. LocalDB runs in a special execution mode of SQL Server Express that enables you to work with databases as *.mdf* files. You can put LocalDB database files in the *App_Data* folder of a web project if you want to be able to copy the database with the project. The user instance feature in SQL Server Express also enables you to work with *.mdf* files, but the user instance feature is deprecated; therefore, LocalDB is recommended for working with *.mdf* files. LocalDB is installed by default with Visual Studio.

Typically, SQL Server Express is not used for production web applications. LocalDB in particular is not recommended for production use with a web application because it's not designed to work with IIS.

- In this tutorial, you'll work with LocalDB. Open the application *Web.config* file and add a `connectionStrings` element preceding the `appSettings` element, as shown in the following example. (Make sure you update the *Web.config* file in the root project folder. There's also a *Web.config* file in the *Views* subfolder that you don't need to update.)

```
<connectionStrings>
    <add name="SchoolContext" connectionString="Data Source=(LocalDb)\MSSQLLocalDB;Initial
Catalog=ContosoUniversity1;Integrated Security=SSPI;" providerName="System.Data.SqlClient"/>
</connectionStrings>
<appSettings>
  <add key="webpages:Version" value="3.0.0.0" />
  <add key="webpages:Enabled" value="false" />
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
```

The connection string you've added specifies that Entity Framework will use a LocalDB database named *ContosoUniversity1.mdf*. (The database doesn't exist yet but EF will create it.) If you want to create the database in your *App_Data* folder, you could add `AttachDBFilename=|DataDirectory|\ContosoUniversity1.mdf` to the connection string. For more information about connection strings, see SQL Server Connection Strings for ASP.NET Web Applications.

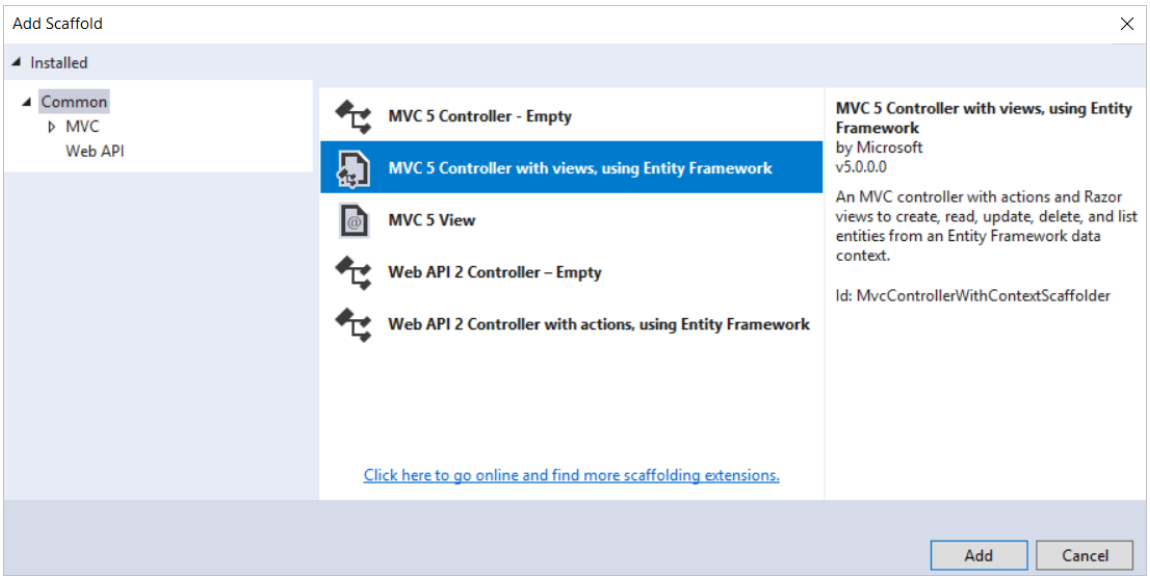
You don't actually need a connection string in the *Web.config* file. If you don't supply a connection string, Entity Framework uses a default connection string based on your context class. For more information, see Code First to a New Database.

Create controller and views

Now you'll create a web page to display data. The process of requesting the data automatically triggers the creation of the database. You'll begin by creating a new controller. But before you do that, build the project to make the model and context classes available to MVC controller scaffolding.

1. Right-click the **Controllers** folder in **Solution Explorer**, select **Add**, and then click **New Scaffolded Item**.

2. In the **Add Scaffold** dialog box, select **MVC 5 Controller with views, using Entity Framework**, and then choose **Add**.



3. In the **Add Controller** dialog box, make the following selections, and then choose **Add**:

   - Model class: **Student (ContosoUniversity.Models)**. (If you don't see this option in the drop-down list, build the project and try again.)
   - Data context class: **SchoolContext (ContosoUniversity.DAL)**.
   - Controller name: **StudentController** (not StudentsController).

- Leave the default values for the other fields.

  When you click **Add**, the scaffolder creates a *StudentController.cs* file and a set of views (*.cshtml* files) that work with the controller. In the future when you create projects that use Entity Framework, you can also take advantage of some additional functionality of the scaffolder: create your first model class, don't create a connection string, and then in the **Add Controller** box specify **New data context** by selecting the **+** button next to **Data context class**. The scaffolder will create your `DbContext` class and your connection string as well as the controller and views.

4. Visual Studio opens the *Controllers\StudentController.cs* file. You see that a class variable has been created that instantiates a database context object:

```
private SchoolContext db = new SchoolContext();
```

The `Index` action method gets a list of students from the *Students* entity set by reading the `Students` property of the database context instance:

```
public ViewResult Index()
{
    return View(db.Students.ToList());
}
```

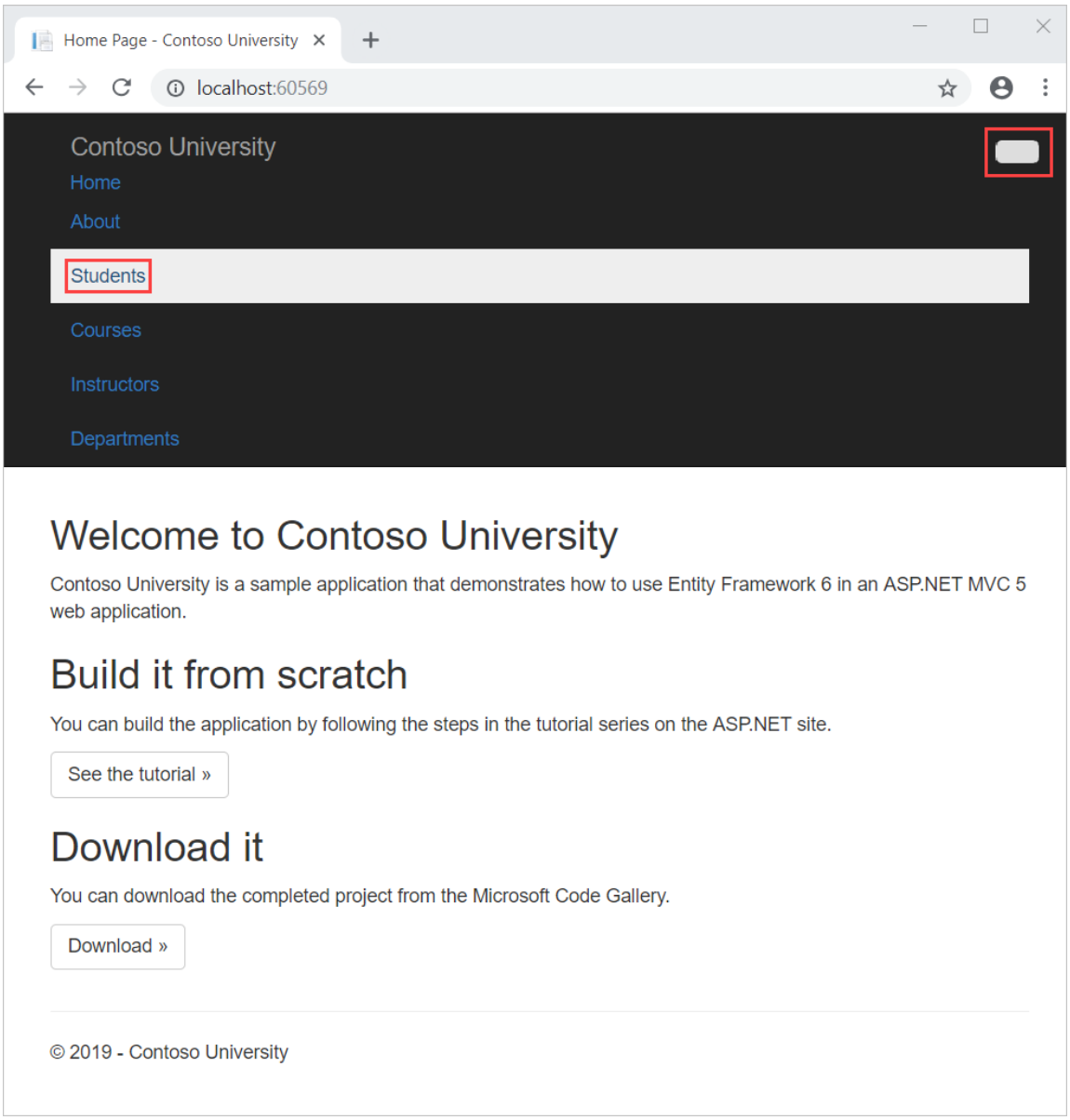
The *Student\Index.cshtml* view displays this list in a table:

```
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.LastName)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.FirstMidName)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.EnrollmentDate)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.LastName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.FirstMidName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.EnrollmentDate)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
    </tr>
}
```

5. Press Ctrl+F5 to run the project. (If you get a "Cannot create Shadow Copy" error, close the browser and try again.)

Click the **Students** tab to see the test data that the `Seed` method inserted. Depending on how narrow your browser window is, you'll see the Student tab link in the top address bar or you'll have to click the upper right corner to see the link.



View the database

When you ran the Students page and the application tried to access the database, EF discovered that there was no database and created one. EF then ran the seed method to populate the database with data.

You can use either **Server Explorer** or **SQL Server Object Explorer** (SSOX) to view the database in Visual Studio. For this tutorial, you'll use **Server Explorer**.

1. Close the browser.
2. In **Server Explorer**, expand **Data Connections** (you may need to select the refresh button first), expand **School Context (ContosoUniversity)**, and then expand **Tables** to see the tables in your new database.
3. Right-click the **Student** table and click **Show Table Data** to see the columns that were created and the rows that were inserted into the table.
4. Close the **Server Explorer** connection.

The *ContosoUniversity1.mdf* and *.ldf* database files are in the *%USERPROFILE%* folder.

Because you're using the `DropCreateDatabaseIfModelChanges` initializer, you could now make a change to the `Student` class, run the application again, and the database would automatically be re-created to match your change. For example, if you add an `EmailAddress` property to the `Student` class, run the Students page again, and then look at the table again, you'll see a new `EmailAddress` column.

Conventions

The amount of code you had to write in order for Entity Framework to be able to create a complete database for you is minimal because of *conventions*, or assumptions that Entity Framework makes. Some of them have already been noted or were used without your being aware of them:

- The pluralized forms of entity class names are used as table names.
- Entity property names are used for column names.
- Entity properties that are named `ID` or *classname* `ID` are recognized as primary key properties.
- A property is interpreted as a foreign key property if it's named *<navigation property name><primary key property name>* (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named the same simply <primary key property name> (for example, `EnrollmentID` since the `Enrollment` entity's primary key is `EnrollmentID`).

You've seen that conventions can be overridden. For example, you specified that table names shouldn't be pluralized, and you'll see later how to explicitly mark a property as a foreign key property.

Get the code

Download Completed Project

Additional resources

For more about EF 6, see these articles:

- ASP.NET Data Access - Recommended Resources
- Code First Conventions
- Creating a More Complex Data Model

Next steps

In this tutorial, you:

- Created an MVC web app
- Set up the site style
- Installed Entity Framework 6
- Created the data model
- Created the database context
- Initialized DB with test data
- Set up EF 6 to use LocalDB
- Created controller and views
- Viewed the database

Advance to the next article to learn how to review and customize the create, read, update, delete (CRUD) code in your controllers and views.

Implement basic CRUD functionality

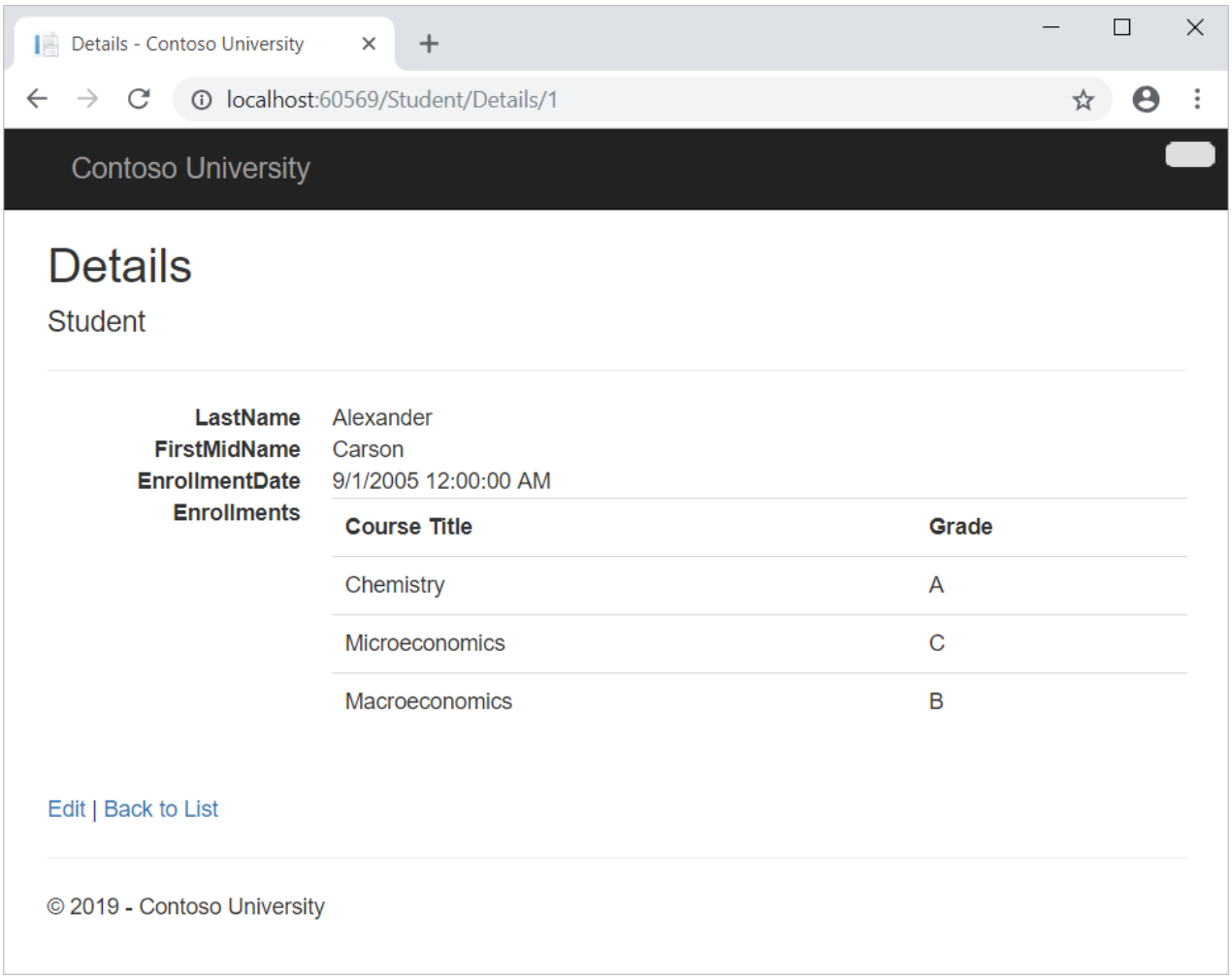
Tutorial: Implement CRUD Functionality with the Entity Framework in ASP.NET MVC

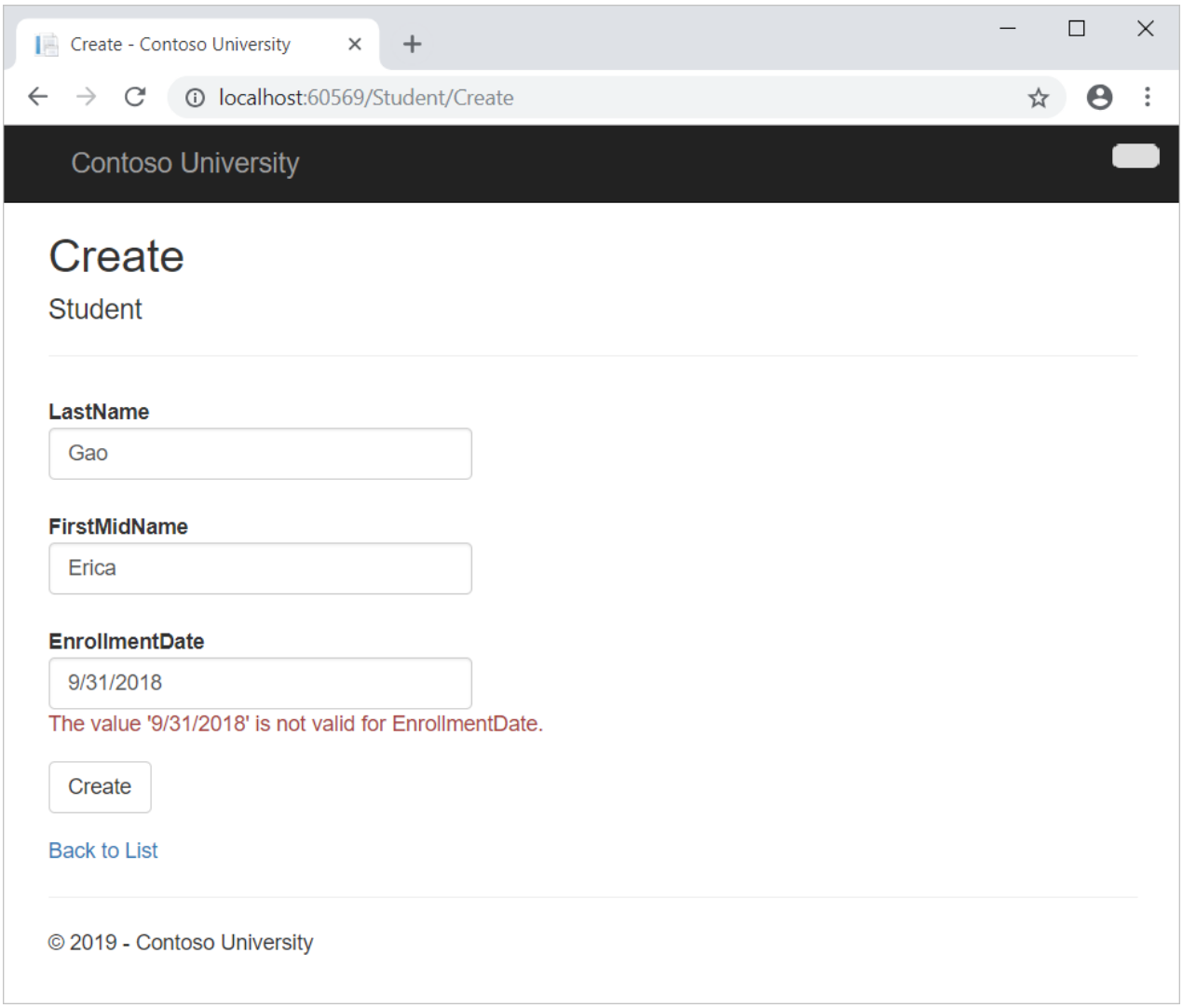
2/20/2020 • 18 minutes to read • Edit Online

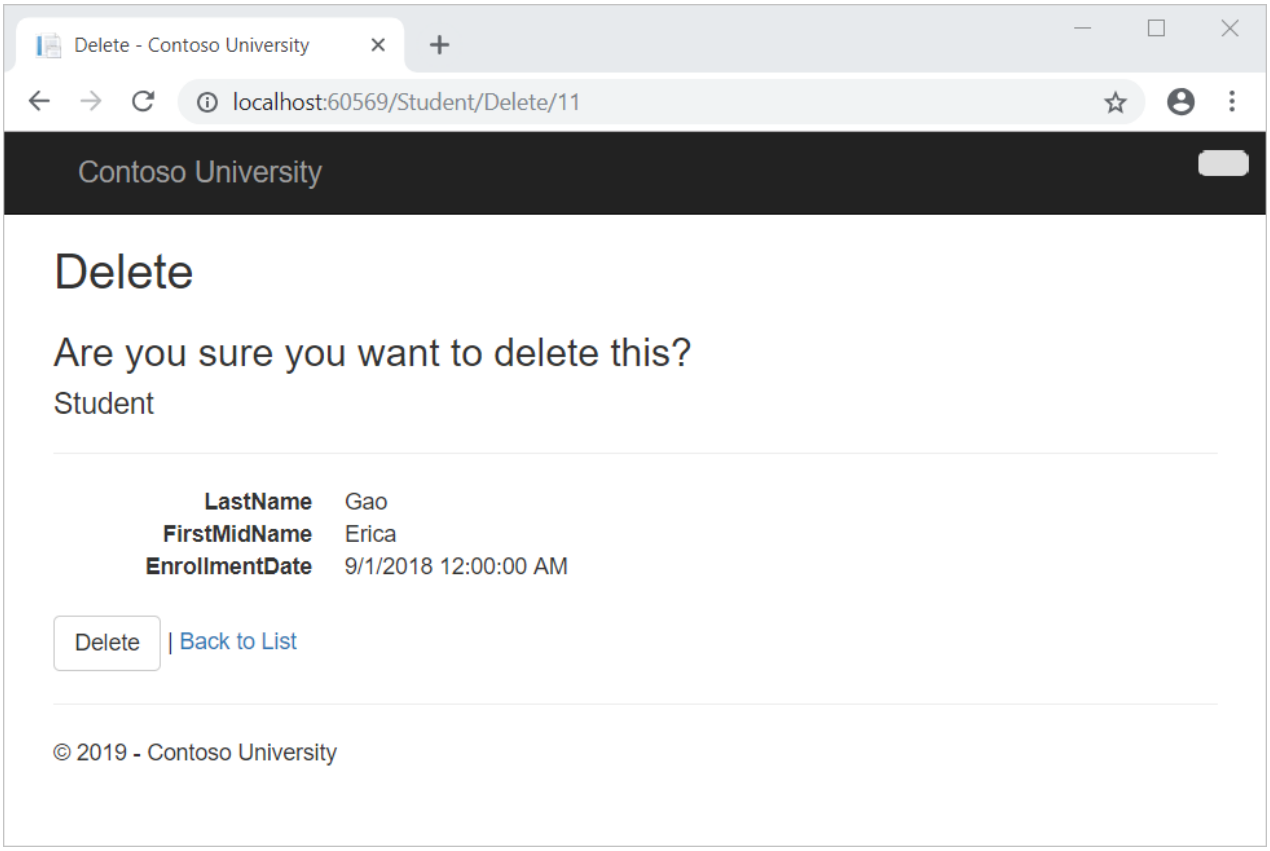
In the previous tutorial, you created an MVC application that stores and displays data using the Entity Framework (EF) 6 and SQL Server LocalDB. In this tutorial, you review and customize the create, read, update, delete (CRUD) code that the MVC scaffolding automatically creates for you in controllers and views.

> **NOTE**
>
> It's a common practice to implement the repository pattern in order to create an abstraction layer between your controller and the data access layer. To keep these tutorials simple and focused on teaching how to use EF 6 itself, they don't use repositories. For info about how to implement repositories, see the ASP.NET Data Access Content Map.

Here are examples of the web pages you create:







In this tutorial, you:

- Create a Details page

- Update the Create page
- Update the HttpPost Edit method
- Update the Delete page
- Close database connections
- Handle transactions

Prerequisites

- Create the Entity Framework Data Model

Create a Details page

The scaffolded code for the Students `Index` page left out the `Enrollments` property, because that property holds a collection. In the `Details` page, you'll display the contents of the collection in an HTML table.

In *Controllers\StudentController.cs*, the action method for the `Details` view uses the Find method to retrieve a single `Student` entity.

```
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Student student = db.Students.Find(id);
    if (student == null)
    {
        return HttpNotFound();
    }
    return View(student);
}
```

The key value is passed to the method as the `id` parameter and comes from *route data* in the **Details** hyperlink on the Index page.

Tip: Route data

Route data is data that the model binder found in a URL segment specified in the routing table. For example, the default route specifies `controller`, `action`, and `id` segments:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

In the following URL, the default route maps `Instructor` as the `controller`, `Index` as the `action` and 1 as the `id`; these are route data values.

`http://localhost:1230/Instructor/Index/1?courseID=2021`

`?courseID=2021` is a query string value. The model binder will also work if you pass the `id` as a query string value:

`http://localhost:1230/Instructor/Index?id=1&CourseID=2021`

The URLs are created by `ActionLink` statements in the Razor view. In the following code, the `id` parameter matches the default route, so `id` is added to the route data.

```
@Html.ActionLink("Select", "Index", new { id = item.PersonID  })
```

In the following code, `courseID` doesn't match a parameter in the default route, so it's added as a query string.

```
@Html.ActionLink("Select", "Index", new { courseID = item.CourseID })
```

To create the Details page

1. Open *Views\Student\Details.cshtml*.

   Each field is displayed using a `DisplayFor` helper, as shown in the following example:

   ```
   <dt>
       @Html.DisplayNameFor(model => model.LastName)
   </dt>
   <dd>
       @Html.DisplayFor(model => model.LastName)
   </dd>
   ```

2. After the `EnrollmentDate` field and immediately before the closing `</dl>` tag, add the highlighted code to display a list of enrollments, as shown in the following example:

   ```
   <dt>
               @Html.DisplayNameFor(model => model.EnrollmentDate)
           </dt>

           <dd>
               @Html.DisplayFor(model => model.EnrollmentDate)
           </dd>
           <dt>
               @Html.DisplayNameFor(model => model.Enrollments)
           </dt>
           <dd>
               <table class="table">
                   <tr>
                       <th>Course Title</th>
                       <th>Grade</th>
                   </tr>
                   @foreach (var item in Model.Enrollments)
                   {
                       <tr>
                           <td>
                               @Html.DisplayFor(modelItem => item.Course.Title)
                           </td>
                           <td>
                               @Html.DisplayFor(modelItem => item.Grade)
                           </td>
                       </tr>
                   }
               </table>
           </dd>
       </dl>
   </div>
   <p>
       @Html.ActionLink("Edit", "Edit", new { id = Model.ID }) |
       @Html.ActionLink("Back to List", "Index")
   </p>
   ```

   If code indentation is wrong after you paste the code, press **Ctrl**+**K**, **Ctrl**+**D** to format it.

   This code loops through the entities in the `Enrollments` navigation property. For each `Enrollment` entity in

the property, it displays the course title and the grade. The course title is retrieved from the `Course` entity that's stored in the `Course` navigation property of the `Enrollments` entity. All of this data is retrieved from the database automatically when it's needed. In other words, you are using lazy loading here. You did not specify *eager loading* for the `Courses` navigation property, so the enrollments were not retrieved in the same query that got the students. Instead, the first time you try to access the `Enrollments` navigation property, a new query is sent to the database to retrieve the data. You can read more about lazy loading and eager loading in the Reading Related Data tutorial later in this series.

3. Open the Details page by starting the program (**Ctrl+F5**), selecting the **Students** tab, and then clicking the **Details** link for Alexander Carson. (If you press **Ctrl+F5** while the *Details.cshtml* file is open, you get an HTTP 400 error. This is because Visual Studio tries to run the Details page, but it wasn't reached from a link that specifies the student to display. If that happens, remove "Student/Details" from the URL and try again, or, close the browser, right-click the project, and click **View** > **View in Browser**.)

   You see the list of courses and grades for the selected student.

4. Close the browser.

Update the Create page

1. In *Controllers\StudentController.cs*, replace the HttpPostAttribute `Create` action method with the following code. This code adds a `try-catch` block and removes `ID` from the BindAttribute attribute for the scaffolded method:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "LastName, FirstMidName, EnrollmentDate")]Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            db.Students.Add(student);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
    }
    catch (DataException /* dex */)
    {
        //Log the error (uncomment dex variable name and add a line here to write a log.
        ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists
see your system administrator.");
    }
    return View(student);
}
```

   This code adds the `Student` entity created by the ASP.NET MVC model binder to the `Students` entity set and then saves the changes to the database. *Model binder* refers to the ASP.NET MVC functionality that makes it easier for you to work with data submitted by a form; a model binder converts posted form values to CLR types and passes them to the action method in parameters. In this case, the model binder instantiates a `Student` entity for you using property values from the `Form` collection.

   You removed `ID` from the Bind attribute because `ID` is the primary key value which SQL Server will set automatically when the row is inserted. Input from the user does not set the `ID` value.

   Security warning - The `ValidateAntiForgeryToken` attribute helps prevent cross-site request forgery attacks. It requires a corresponding `Html.AntiForgeryToken()` statement in the view, which you'll see later.

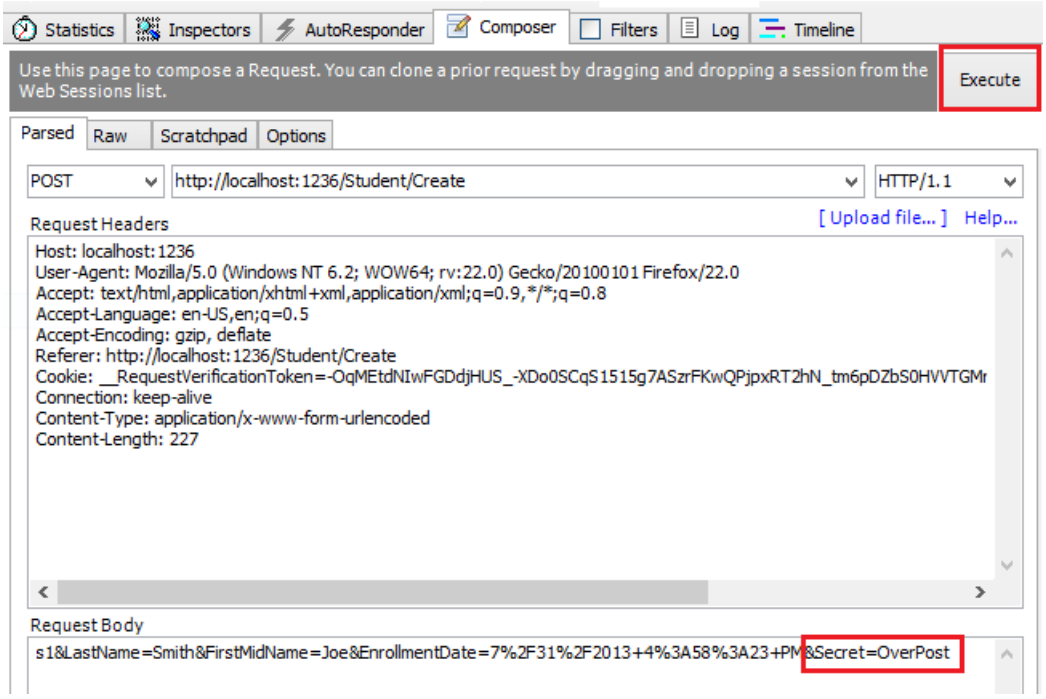
   The `Bind` attribute is one way to protect against *over-posting* in create scenarios. For example, suppose the

`Student` entity includes a `Secret` property that you don't want this web page to set.

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }

    public virtual ICollection<Enrollment> Enrollments { get; set; }
}
```

Even if you don't have a `Secret` field on the web page, a hacker could use a tool such as fiddler, or write some JavaScript, to post a `Secret` form value. Without the BindAttribute attribute limiting the fields that the model binder uses when it creates a `Student` instance, the model binder would pick up that `Secret` form value and use it to create the `Student` entity instance. Then whatever value the hacker specified for the `Secret` form field would be updated in your database. The following image shows the fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" would then be successfully added to the `Secret` property of the inserted row, although you never intended that the web page be able to set that property.

It's best to use the `Include` parameter with the `Bind` attribute to *whitelist* fields. It's also possible to use the `Exclude` parameter to *blacklist* fields you want to exclude. The reason `Include` is more secure is that when you add a new property to the entity, the new field is not automatically protected by an `Exclude` list.

You can prevent overposting in edit scenarios is by reading the entity from the database first and then calling `TryUpdateModel`, passing in an explicit allowed properties list. That is the method used in these tutorials.

An alternative way to prevent overposting that is preferred by many developers is to use view models rather than entity classes with model binding. Include only the properties you want to update in the view model. Once the MVC model binder has finished, copy the view model properties to the entity instance, optionally using a tool such as AutoMapper. Use db.Entry on the entity instance to set its state to Unchanged, and then set Property("PropertyName").IsModified to true on each entity property that is included in the view model. This method works in both edit and create scenarios.

Other than the `Bind` attribute, the `try-catch` block is the only change you've made to the scaffolded code. If an exception that derives from DataException is caught while the changes are being saved, a generic error message is displayed. DataException exceptions are sometimes caused by something external to the application rather than a programming error, so the user is advised to try again. Although not implemented in this sample, a production quality application would log the exception. For more information, see the **Log for insight** section in Monitoring and Telemetry (Building Real-World Cloud Apps with Azure).

The code in *Views\Student\Create.cshtml* is similar to what you saw in *Details.cshtml*, except that `EditorFor` and `ValidationMessageFor` helpers are used for each field instead of `DisplayFor`. Here is the relevant code:

```
<div class="form-group">
    @Html.LabelFor(model => model.LastName, new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.LastName)
        @Html.ValidationMessageFor(model => model.LastName)
    </div>
</div>
```

Create.cshtml also includes `@Html.AntiForgeryToken()`, which works with the `ValidateAntiForgeryToken` attribute in the controller to help prevent cross-site request forgery attacks.

No changes are required in *Create.cshtml*.

2. Run the page by starting the program, selecting the **Students** tab, and then clicking **Create New**.

3. Enter names and an invalid date and click **Create** to see the error message.

   This is server-side validation that you get by default. In a later tutorial, you'll see how to add attributes that generate code for client-side validation. The following highlighted code shows the model validation check in the **Create** method.

```
if (ModelState.IsValid)
{
    db.Students.Add(student);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

4. Change the date to a valid value and click **Create** to see the new student appear in the **Index** page.

5. Close the browser.

Update HttpPost Edit method

1. Replace the HttpPostAttribute `Edit` action method with the following code:

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public ActionResult EditPost(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    var studentToUpdate = db.Students.Find(id);
    if (TryUpdateModel(studentToUpdate, "",
       new string[] { "LastName", "FirstMidName", "EnrollmentDate" }))
    {
        try
        {
            db.SaveChanges();

            return RedirectToAction("Index");
        }
        catch (DataException /* dex */)
        {
            //Log the error (uncomment dex variable name and add a line here to write a log.
            ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem
persists, see your system administrator.");
        }
    }
    return View(studentToUpdate);
}
```

NOTE

In *Controllers\StudentController.cs*, the `HttpGet Edit` method (the one without the `HttpPost` attribute) uses the `Find` method to retrieve the selected `Student` entity, as you saw in the `Details` method. You don't need to change this method.

These changes implement a security best practice to prevent overposting, The scaffolder generated a `Bind` attribute and added the entity created by the model binder to the entity set with a Modified flag. That code is no longer recommended because the `Bind` attribute clears out any pre-existing data in fields not listed in the `Include` parameter. In the future, the MVC controller scaffolder will be updated so that it doesn't generate `Bind` attributes for Edit methods.

The new code reads the existing entity and calls TryUpdateModel to update fields from user input in the posted form data. The Entity Framework's automatic change tracking sets the EntityState.Modified flag on the entity. When the SaveChanges method is called, the Modified flag causes the Entity Framework to create SQL statements to update the database row. Concurrency conflicts are ignored, and all columns of the database row are updated, including those that the user didn't change. (A later tutorial shows how to handle concurrency conflicts, and if you only want individual fields to be updated in the database, you can set the entity to EntityState.Unchanged and set individual fields to EntityState.Modified.)

To prevent overposting, the fields that you want to be updateable by the Edit page are whitelisted in the `TryUpdateModel` parameters. Currently there are no extra fields that you're protecting, but listing the fields that you want the model binder to bind ensures that if you add fields to the data model in the future, they're automatically protected until you explicitly add them here.

As a result of these changes, the method signature of the HttpPost Edit method is the same as the HttpGet edit method; therefore you've renamed the method EditPost.

TIP

Entity States and the Attach and SaveChanges Methods

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database, and this information determines what happens when you call the `SaveChanges` method. For example, when you pass a new entity to the Add method, that entity's state is set to `Added`. Then when you call the SaveChanges method, the database context issues a SQL `INSERT` command.

An entity may be in one of the following states:

- `Added`. The entity does not yet exist in the database. The `SaveChanges` method must issue an `INSERT` statement.
- `Unchanged`. Nothing needs to be done with this entity by the `SaveChanges` method. When you read an entity from the database, the entity starts out with this status.
- `Modified`. Some or all of the entity's property values have been modified. The `SaveChanges` method must issue an `UPDATE` statement.
- `Deleted`. The entity has been marked for deletion. The `SaveChanges` method must issue a `DELETE` statement.
- `Detached`. The entity isn't being tracked by the database context.

In a desktop application, state changes are typically set automatically. In a desktop type of application, you read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to `Modified`. Then when you call `SaveChanges`, the Entity Framework generates a SQL `UPDATE` statement that updates only the actual properties that you changed.

The disconnected nature of web apps doesn't allow for this continuous sequence. The DbContext that reads an entity is disposed after a page is rendered. When the `HttpPost` `Edit` action method is called, a new request is made and you have a new instance of the DbContext, so you have to manually set the entity state to `Modified.` Then when you call `SaveChanges`, the Entity Framework updates all columns of the database row, because the context has no way to know which properties you changed.

If you want the SQL `Update` statement to update only the fields that the user actually changed, you can save the original values in some way (such as hidden fields) so that they are available when the `HttpPost` `Edit` method is called. Then you can create a `Student` entity using the original values, call the `Attach` method with that original version of the entity, update the entity's values to the new values, and then call `SaveChanges.` For more information, see Entity states and SaveChanges and Local Data.

The HTML and Razor code in *Views\Student\Edit.cshtml* is similar to what you saw in *Create.cshtml*, and no changes are required.

2. Run the page by starting the program, selecting the **Students** tab, and then clicking an **Edit** hyperlink.

3. Change some of the data and click **Save**. You see the changed data in the Index page.

4. Close the browser.

Update the Delete page

In *Controllers\StudentController.cs*, the template code for the HttpGetAttribute `Delete` method uses the `Find` method to retrieve the selected `Student` entity, as you saw in the `Details` and `Edit` methods. However, to implement a custom error message when the call to `SaveChanges` fails, you'll add some functionality to this method and its corresponding view.

As you saw for update and create operations, delete operations require two action methods. The method that is called in response to a GET request displays a view that gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST request is created. When that happens, the `HttpPost` `Delete` method is

called and then that method actually performs the delete operation.

You'll add a `try-catch` block to the HttpPostAttribute `Delete` method to handle any errors that might occur when the database is updated. If an error occurs, the HttpPostAttribute `Delete` method calls the HttpGetAttribute `Delete` method, passing it a parameter that indicates that an error has occurred. The HttpGetAttribute `Delete` method then redisplays the confirmation page along with the error message, giving the user an opportunity to cancel or try again.

1. Replace the HttpGetAttribute `Delete` action method with the following code, which manages error reporting:

```
public ActionResult Delete(int? id, bool? saveChangesError=false)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    if (saveChangesError.GetValueOrDefault())
    {
        ViewBag.ErrorMessage = "Delete failed. Try again, and if the problem persists see your system
administrator.";
    }
    Student student = db.Students.Find(id);
    if (student == null)
    {
        return HttpNotFound();
    }
    return View(student);
}
```

   This code accepts an optional parameter that indicates whether the method was called after a failure to save changes. This parameter is `false` when the `HttpGet` `Delete` method is called without a previous failure. When it is called by the `HttpPost` `Delete` method in response to a database update error, the parameter is `true` and an error message is passed to the view.

2. Replace the HttpPostAttribute `Delete` action method (named `DeleteConfirmed`) with the following code, which performs the actual delete operation and catches any database update errors.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete(int id)
{
    try
    {
        Student student = db.Students.Find(id);
        db.Students.Remove(student);
        db.SaveChanges();
    }
    catch (DataException/* dex */)
    {
        //Log the error (uncomment dex variable name and add a line here to write a log.
        return RedirectToAction("Delete", new { id = id, saveChangesError = true });
    }
    return RedirectToAction("Index");
}
```

   This code retrieves the selected entity, then calls the Remove method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL `DELETE` command is generated. You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code named the `HttpPost` `Delete` method `DeleteConfirmed` to give the `HttpPost` method a unique signature. (The CLR requires overloaded methods

to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the `HttpPost` and `HttpGet` delete methods.

If improving performance in a high-volume application is a priority, you could avoid an unnecessary SQL query to retrieve the row by replacing the lines of code that call the `Find` and `Remove` methods with the following code:

```
Student studentToDelete = new Student() { ID = id };
db.Entry(studentToDelete).State = EntityState.Deleted;
```

This code instantiates a `Student` entity using only the primary key value and then sets the entity state to `Deleted`. That's all that the Entity Framework needs in order to delete the entity.

As noted, the `HttpGet` `Delete` method doesn't delete the data. Performing a delete operation in response to a GET request (or for that matter, performing any edit operation, create operation, or any other operation that changes data) creates a security risk. For more information, see ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes on Stephen Walther's blog.

3. In *Views\Student\Delete.cshtml*, add an error message between the `h2` heading and the `h3` heading, as shown in the following example:

```
<h2>Delete</h2>
<p class="error">@ViewBag.ErrorMessage</p>
<h3>Are you sure you want to delete this?</h3>
```

4. Run the page by starting the program, selecting the **Students** tab, and then clicking a **Delete** hyperlink.

5. Choose **Delete** on the page that says **Are you sure you want to delete this?**.

   The Index page displays without the deleted student. (You'll see an example of the error handling code in action in the concurrency tutorial.)

Close database connections

To close database connections and free up the resources they hold as soon as possible, dispose the context instance when you are done with it. That is why the scaffolded code provides a Dispose method at the end of the `StudentController` class in *StudentController.cs*, as shown in the following example:

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        db.Dispose();
    }
    base.Dispose(disposing);
}
```

The base `Controller` class already implements the `IDisposable` interface, so this code simply adds an override to the `Dispose(bool)` method to explicitly dispose the context instance.

Handle transactions

By default the Entity Framework implicitly implements transactions. In scenarios where you make changes to multiple rows or tables and then call `SaveChanges`, the Entity Framework automatically makes sure that either all of your changes succeed or all fail. If some changes are done first and then an error happens, those changes are

automatically rolled back. For scenarios where you need more control—for example, if you want to include operations done outside of Entity Framework in a transaction—see Working with Transactions.

Get the code

Download Completed Project

Additional resources

You now have a complete set of pages that perform simple CRUD operations for `Student` entities. You used MVC helpers to generate UI elements for data fields. For more info about MVC helpers, see Rendering a Form Using HTML Helpers (the article is for MVC 3 but is still relevant for MVC 5).

Links to other EF 6 resources can be found in ASP.NET Data Access - Recommended Resources.

Next steps

In this tutorial, you:

- Created a Details page
- Updated the Create page
- Updated the HttpPost Edit method
- Updated the Delete page
- Closed database connections
- Handled transactions

Advance to the next article to learn how to add sorting, filtering, and paging to the project.

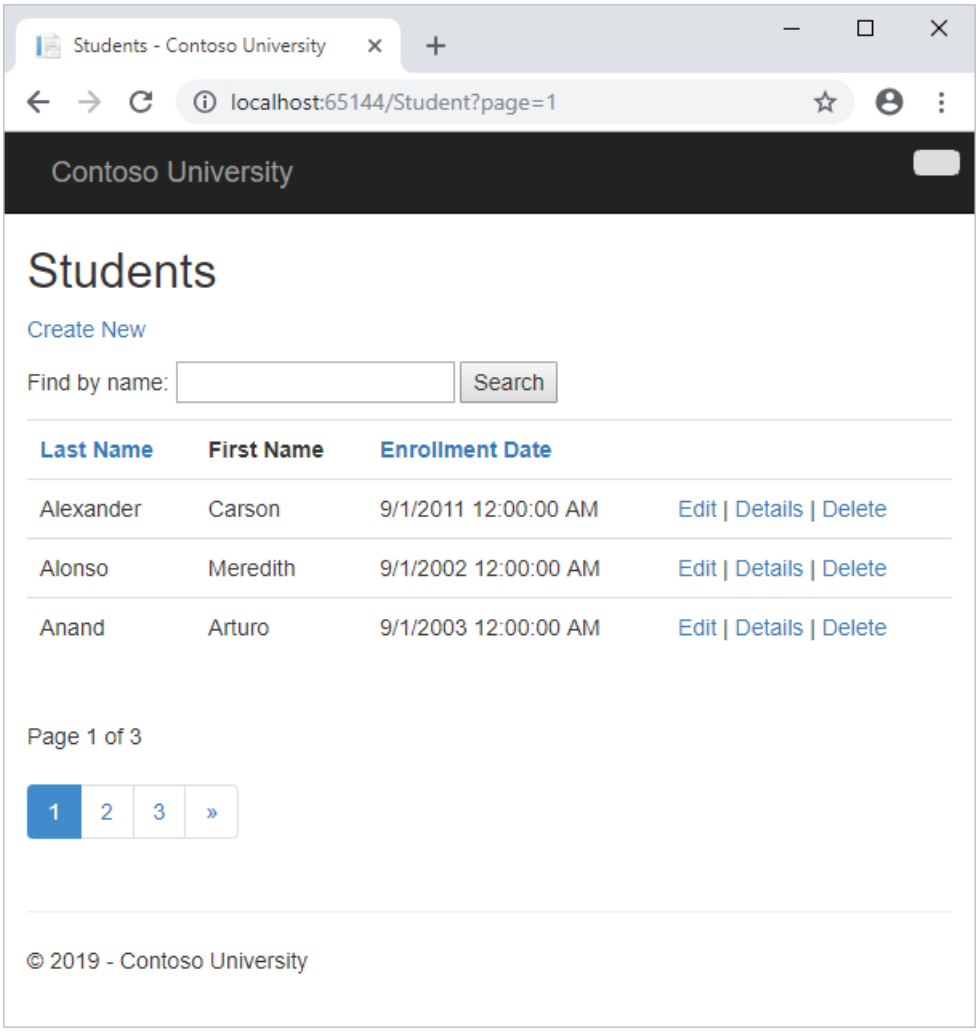
Sorting, Filtering, and Paging

Tutorial: Add sorting, filtering, and paging with the Entity Framework in an ASP.NET MVC application

2/20/2020 • 14 minutes to read • Edit Online

In the previous tutorial, you implemented a set of web pages for basic CRUD operations for `Student` entities. In this tutorial you add sorting, filtering, and paging functionality to the **Students** Index page. You also create a simple grouping page.

The following image shows what the page will look like when you're done. The column headings are links that the user can click to sort by that column. Clicking a column heading repeatedly toggles between ascending and descending sort order.



In this tutorial, you:

- Add column sort links
- Add a Search box
- Add paging
- Create an About page

Prerequisites

- Implementing Basic CRUD Functionality

Add column sort links

To add sorting to the Student Index page, you'll change the `Index` method of the `Student` controller and add code to the `Student` Index view.

Add sorting functionality to the Index method

- In *Controllers\StudentController.cs*, replace the `Index` method with the following code:

```
public ActionResult Index(string sortOrder)
{
   ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
   ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";
   var students = from s in db.Students
                  select s;
   switch (sortOrder)
   {
      case "name_desc":
         students = students.OrderByDescending(s => s.LastName);
         break;
      case "Date":
         students = students.OrderBy(s => s.EnrollmentDate);
         break;
      case "date_desc":
         students = students.OrderByDescending(s => s.EnrollmentDate);
         break;
      default:
         students = students.OrderBy(s => s.LastName);
         break;
   }
   return View(students.ToList());
}
```

This code receives a `sortOrder` parameter from the query string in the URL. The query string value is provided by ASP.NET MVC as a parameter to the action method. The parameter is a string that's either "Name" or "Date", optionally followed by an underscore and the string "desc" to specify descending order. The default sort order is ascending.

The first time the Index page is requested, there's no query string. The students are displayed in ascending order by `LastName`, which is the default as established by the fall-through case in the `switch` statement. When the user clicks a column heading hyperlink, the appropriate `sortOrder` value is provided in the query string.

The two `ViewBag` variables are used so that the view can configure the column heading hyperlinks with the appropriate query string values:

```
ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";
```

These are ternary statements. The first one specifies that if the `sortOrder` parameter is null or empty, `ViewBag.NameSortParm` should be set to "name_desc"; otherwise, it should be set to an empty string. These two statements enable the view to set the column heading hyperlinks as follows:

CURRENT SORT ORDER	LAST NAME HYPERLINK	DATE HYPERLINK
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending

CURRENT SORT ORDER	LAST NAME HYPERLINK	DATE HYPERLINK
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code creates an IQueryable<T> variable before the `switch` statement, modifies it in the `switch` statement, and calls the `ToList` method after the `switch` statement. When you create and modify `IQueryable` variables, no query is sent to the database. The query is not executed until you convert the `IQueryable` object into a collection by calling a method such as `ToList`. Therefore, this code results in a single query that is not executed until the `return View` statement.

As an alternative to writing different LINQ statements for each sort order, you can dynamically create a LINQ statement. For information about dynamic LINQ, see Dynamic LINQ.

Add column heading hyperlinks to the Student index view

1. In *Views\Student\Index.cshtml*, replace the `<tr>` and `<th>` elements for the heading row with the highlighted code:

```
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>
            @Html.ActionLink("Last Name", "Index", new { sortOrder = ViewBag.NameSortParm })
        </th>
        <th>First Name
        </th>
        <th>
            @Html.ActionLink("Enrollment Date", "Index", new { sortOrder = ViewBag.DateSortParm })
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
```

   This code uses the information in the `ViewBag` properties to set up hyperlinks with the appropriate query string values.

2. Run the page and click the **Last Name** and **Enrollment Date** column headings to verify that sorting works.

   After you click the **Last Name** heading, students are displayed in descending last name order.

Add a Search box

To add filtering to the Students index page, you'll add a text box and a submit button to the view and make corresponding changes in the `Index` method. The text box lets you enter a string to search for in the first name and last name fields.

Add filtering functionality to the Index method

- In *Controllers\StudentController.cs*, replace the `Index` method with the following code (the changes are highlighted):

```
public ViewResult Index(string sortOrder, string searchString)
{
    ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in db.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }

    return View(students.ToList());
}
```

The code adds a `searchString` parameter to the `Index` method. The search string value is received from a text box that you'll add to the Index view. It also adds a `where` clause to the LINQ statement that selects only students whose first name or last name contains the search string. The statement that adds the Where clause executes only if there's a value to search for.

> **NOTE**
>
> In many cases you can call the same method either on an Entity Framework entity set or as an extension method on an in-memory collection. The results are normally the same but in some cases may be different.
>
> For example, the .NET Framework implementation of the `Contains` method returns all rows when you pass an empty string to it, but the Entity Framework provider for SQL Server Compact 4.0 returns zero rows for empty strings. Therefore the code in the example (putting the `Where` statement inside an `if` statement) makes sure that you get the same results for all versions of SQL Server. Also, the .NET Framework implementation of the `Contains` method performs a case-sensitive comparison by default, but Entity Framework SQL Server providers perform case-insensitive comparisons by default. Therefore, calling the `ToUpper` method to make the test explicitly case-insensitive ensures that results do not change when you change the code later to use a repository, which will return an `IEnumerable` collection instead of an `IQueryable` object. (When you call the `Contains` method on an `IEnumerable` collection, you get the .NET Framework implementation; when you call it on an `IQueryable` object, you get the database provider implementation.)
>
> Null handling may also be different for different database providers or when you use an `IQueryable` object compared to when you use an `IEnumerable` collection. For example, in some scenarios a `Where` condition such as `table.Column != 0` may not return columns that have `null` as the value. By default, EF generates additional SQL operators to make equality between null values work in the database like it works in memory, but you can set the UseDatabaseNullSemantics flag in EF6 or call the UseRelationalNulls method in EF Core to configure this behavior.

Add a search box to the Student index view

1. In *Views\Student\Index.cshtml*, add the highlighted code immediately before the opening `table` tag in order to create a caption, a text box, and a **Search** button.

```
<p>
    @Html.ActionLink("Create New", "Create")
</p>

@using (Html.BeginForm())
{
    <p>
        Find by name: @Html.TextBox("SearchString")
        <input type="submit" value="Search" /></p>
}

<table>
    <tr>
```

2. Run the page, enter a search string, and click **Search** to verify that filtering is working.

   Notice the URL doesn't contain the "an" search string, which means that if you bookmark this page, you won't get the filtered list when you use the bookmark. This applies also to the column sort links, as they will sort the whole list. You'll change the **Search** button to use query strings for filter criteria later in the tutorial.

Add paging

To add paging to the Students index page, you'll start by installing the **PagedList.Mvc** NuGet package. Then you'll make additional changes in the `Index` method and add paging links to the `Index` view. **PagedList.Mvc** is one of many good paging and sorting packages for ASP.NET MVC, and its use here is intended only as an example, not as a recommendation for it over other options.

Install the PagedList.MVC NuGet package

The NuGet **PagedList.Mvc** package automatically installs the **PagedList** package as a dependency. The **PagedList** package installs a `PagedList` collection type and extension methods for `IQueryable` and `IEnumerable` collections. The extension methods create a single page of data in a `PagedList` collection out of your `IQueryable` or `IEnumerable`, and the `PagedList` collection provides several properties and methods that facilitate paging. The **PagedList.Mvc** package installs a paging helper that displays the paging buttons.

1. From the **Tools** menu, select **NuGet Package Manager** and then **Package Manager Console**.

2. In the **Package Manager Console** window, make sure the **Package source** is **nuget.org** and the **Default project** is **ContosoUniversity**, and then enter the following command:

```
Install-Package PagedList.Mvc
```

3. Build the project.

Add paging functionality to the Index method

1. In *Controllers\StudentController.cs*, add a `using` statement for the `PagedList` namespace:

```
using PagedList;
```

2. Replace the `Index` method with the following code:

```
public ViewResult Index(string sortOrder, string currentFilter, string searchString, int? page)
{
   ViewBag.CurrentSort = sortOrder;
   ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
   ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";

   if (searchString != null)
   {
      page = 1;
   }
   else
   {
      searchString = currentFilter;
   }

   ViewBag.CurrentFilter = searchString;

   var students = from s in db.Students
                  select s;
   if (!String.IsNullOrEmpty(searchString))
   {
      students = students.Where(s => s.LastName.Contains(searchString)
                              || s.FirstMidName.Contains(searchString));
   }
   switch (sortOrder)
   {
      case "name_desc":
         students = students.OrderByDescending(s => s.LastName);
         break;
      case "Date":
         students = students.OrderBy(s => s.EnrollmentDate);
         break;
      case "date_desc":
         students = students.OrderByDescending(s => s.EnrollmentDate);
         break;
      default:  // Name ascending
         students = students.OrderBy(s => s.LastName);
         break;
   }

   int pageSize = 3;
   int pageNumber = (page ?? 1);
   return View(students.ToPagedList(pageNumber, pageSize));
}
```

This code adds a `page` parameter, a current sort order parameter, and a current filter parameter to the method signature:

```
public ActionResult Index(string sortOrder, string currentFilter, string searchString, int? page)
```

The first time the page is displayed, or if the user hasn't clicked a paging or sorting link, all the parameters are null. If a paging link is clicked, the `page` variable contains the page number to display.

A `ViewBag` property provides the view with the current sort order, because this must be included in the paging links in order to keep the sort order the same while paging:

```
ViewBag.CurrentSort = sortOrder;
```

Another property, `ViewBag.CurrentFilter`, provides the view with the current filter string. This value must be included in the paging links in order to maintain the filter settings during paging, and it must be restored to the text box when the page is redisplayed. If the search string is changed during paging, the

page has to be reset to 1, because the new filter can result in different data to display. The search string is changed when a value is entered in the text box and the submit button is pressed. In that case, the `searchString` parameter is not null.

```
if (searchString != null)
{
    page = 1;
}
else
{
    searchString = currentFilter;
}
```

At the end of the method, the `ToPagedList` extension method on the students `IQueryable` object converts the student query to a single page of students in a collection type that supports paging. That single page of students is then passed to the view:

```
int pageSize = 3;
int pageNumber = (page ?? 1);
return View(students.ToPagedList(pageNumber, pageSize));
```

The `ToPagedList` method takes a page number. The two question marks represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type; the expression `(page ?? 1)` means return the value of `page` if it has a value, or return 1 if `page` is null.

Add paging links to the Student index view

1. In *Views\Student\Index.cshtml*, replace the existing code with the following code. The changes are highlighted.

```
@model PagedList.IPagedList<ContosoUniversity.Models.Student>
@using PagedList.Mvc;
<link href="~/Content/PagedList.css" rel="stylesheet" type="text/css" />

@{
    ViewBag.Title = "Students";
}

<h2>Students</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
@using (Html.BeginForm("Index", "Student", FormMethod.Get))
{
    <p>
        Find by name: @Html.TextBox("SearchString", ViewBag.CurrentFilter as string)
        <input type="submit" value="Search" />
    </p>
}
<table class="table">
    <tr>
        <th>
            @Html.ActionLink("Last Name", "Index", new { sortOrder = ViewBag.NameSortParm,
currentFilter=ViewBag.CurrentFilter })
        </th>
        <th>
            First Name
        </th>
        <th>
            @Html.ActionLink("Enrollment Date", "Index", new { sortOrder = ViewBag.DateSortParm,
currentFilter=ViewBag.CurrentFilter })
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.LastName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.FirstMidName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.EnrollmentDate)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
    </tr>
}

</table>
<br />
Page @(Model.PageCount < Model.PageNumber ? 0 : Model.PageNumber) of @Model.PageCount

@Html.PagedListPager(Model, page => Url.Action("Index",
    new { page, sortOrder = ViewBag.CurrentSort, currentFilter = ViewBag.CurrentFilter }))
```

The `@model` statement at the top of the page specifies that the view now gets a `PagedList` object instead of a `List` object.

The `using` statement for `PagedList.Mvc` gives access to the MVC helper for the paging buttons.

The code uses an overload of BeginForm that allows it to specify FormMethod.Get.

```
@using (Html.BeginForm("Index", "Student", FormMethod.Get))
{
    <p>
        Find by name: @Html.TextBox("SearchString", ViewBag.CurrentFilter as string)
        <input type="submit" value="Search" />
    </p>
}
```

The default BeginForm submits form data with a POST, which means that parameters are passed in the HTTP message body and not in the URL as query strings. When you specify HTTP GET, the form data is passed in the URL as query strings, which enables users to bookmark the URL. The W3C guidelines for the use of HTTP GET recommend that you should use GET when the action does not result in an update.

The text box is initialized with the current search string so when you click a new page you can see the current search string.

```
Find by name: @Html.TextBox("SearchString", ViewBag.CurrentFilter as string)
```

The column header links use the query string to pass the current search string to the controller so that the user can sort within filter results:

```
@Html.ActionLink("Last Name", "Index", new { sortOrder=ViewBag.NameSortParm,
currentFilter=ViewBag.CurrentFilter })
```

The current page and total number of pages are displayed.

```
Page @(Model.PageCount < Model.PageNumber ? 0 : Model.PageNumber) of @Model.PageCount
```

If there are no pages to display, "Page 0 of 0" is shown. (In that case the page number is greater than the page count because `Model.PageNumber` is 1, and `Model.PageCount` is 0.)

The paging buttons are displayed by the `PagedListPager` helper:

```
@Html.PagedListPager( Model, page => Url.Action("Index", new { page }) )
```

The `PagedListPager` helper provides a number of options that you can customize, including URLs and styling. For more information, see TroyGoode / PagedList on the GitHub site.

2. Run the page.

   Click the paging links in different sort orders to make sure paging works. Then enter a search string and try paging again to verify that paging also works correctly with sorting and filtering.

Create an About page

For the Contoso University website's About page, you'll display how many students have enrolled for each enrollment date. This requires grouping and simple calculations on the groups. To accomplish this, you'll do the following:

- Create a view model class for the data that you need to pass to the view.
- Modify the `About` method in the `Home` controller.

- Modify the `About` view.

Create the View Model

Create a *ViewModels* folder in the project folder. In that folder, add a class file *EnrollmentDateGroup.cs* and replace the template code with the following code:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.ViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Modify the Home Controller

1. In *HomeController.cs*, add the following `using` statements at the top of the file:

   ```
   using ContosoUniversity.DAL;
   using ContosoUniversity.ViewModels;
   ```

2. Add a class variable for the database context immediately after the opening curly brace for the class:

   ```
   public class HomeController : Controller
   {
       private SchoolContext db = new SchoolContext();
   ```

3. Replace the `About` method with the following code:

   ```
   public ActionResult About()
   {
       IQueryable<EnrollmentDateGroup> data = from student in db.Students
                  group student by student.EnrollmentDate into dateGroup
                  select new EnrollmentDateGroup()
                  {
                      EnrollmentDate = dateGroup.Key,
                      StudentCount = dateGroup.Count()
                  };
       return View(data.ToList());
   }
   ```

   The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

4. Add a `Dispose` method:

   ```
   protected override void Dispose(bool disposing)
   {
       db.Dispose();
       base.Dispose(disposing);
   }
   ```

Modify the About View

1. Replace the code in the *Views\Home\About.cshtml* file with the following code:

```
@model IEnumerable<ContosoUniversity.ViewModels.EnrollmentDateGroup>

@{
    ViewBag.Title = "Student Body Statistics";
}

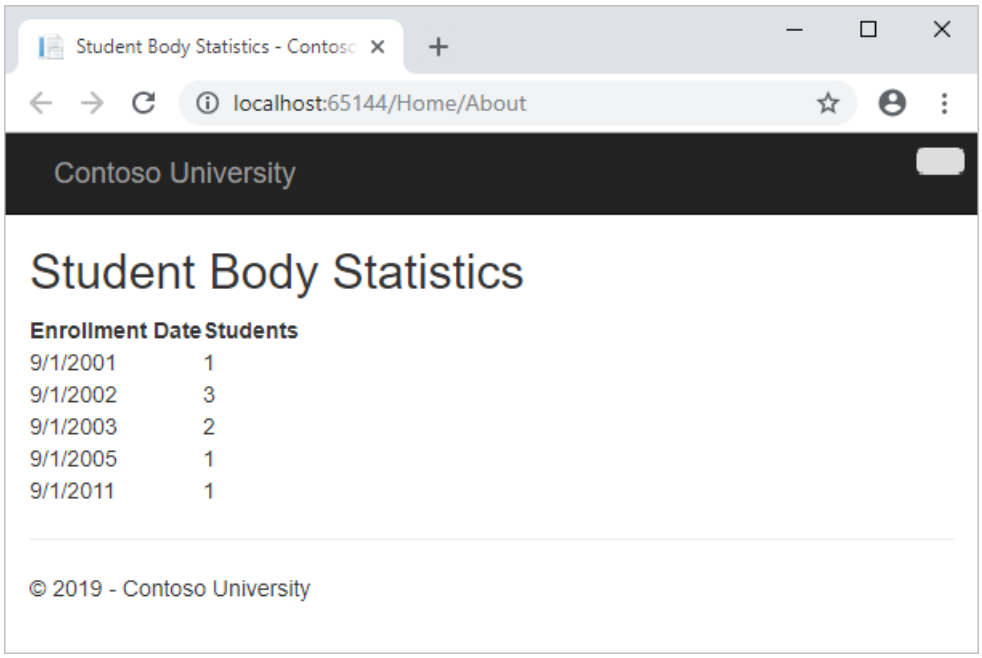
<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.EnrollmentDate)
        </td>
        <td>
            @item.StudentCount
        </td>
    </tr>
}
</table>
```

2. Run the app and click the **About** link.

   The count of students for each enrollment date displays in a table.



Get the code

Download the Completed Project

Additional resources

Links to other Entity Framework resources can be found in ASP.NET Data Access - Recommended Resources.

Next steps

In this tutorial, you:

- Add column sort links
- Add a Search box
- Add paging
- Create an About page

Advance to the next article to learn how to use connection resiliency and command interception.

Connection resiliency and command interception

Tutorial: Use connection resiliency and command interception with Entity Framework in an ASP.NET MVC app

2/20/2020 • 17 minutes to read • Edit Online

So far the application has been running locally in IIS Express on your development computer. To make a real application available for other people to use over the Internet, you have to deploy it to a web hosting provider, and you have to deploy the database to a database server.

In this tutorial you'll learn how to use connection resiliency and command interception. They are two important features of Entity Framework 6 that are especially valuable when you are deploying to the cloud environment: connection resiliency (automatic retries for transient errors) and command interception (catch all SQL queries sent to the database in order to log or change them).

This connection resiliency and command interception tutorial is optional. If you skip this tutorial, a few minor adjustments will have to be made in subsequent tutorials.

In this tutorial, you:

- Enable connection resiliency
- Enable command interception
- Test the new configuration

Prerequisites

- Sorting, Filtering, and Paging

Enable connection resiliency

When you deploy the application to Windows Azure, you'll deploy the database to Windows Azure SQL Database, a cloud database service. Transient connection errors are typically more frequent when you connect to a cloud database service than when your web server and your database server are directly connected together in the same data center. Even if a cloud web server and a cloud database service are hosted in the same data center, there are more network connections between them that can have problems, such as load balancers.

Also a cloud service is typically shared by other users, which means its responsiveness can be affected by them. And your access to the database might be subject to throttling. Throttling means the database service throws exceptions when you try to access it more frequently than is allowed in your Service Level Agreement (SLA).

Many or most connection problems when you're accessing a cloud service are transient, that is, they resolve themselves in a short period of time. So when you try a database operation and get a type of error that is typically transient, you could try the operation again after a short wait, and the operation might be successful. You can provide a much better experience for your users if you handle transient errors by automatically trying again, making most of them invisible to the customer. The connection resiliency feature in Entity Framework 6 automates that process of retrying failed SQL queries.

The connection resiliency feature must be configured appropriately for a particular database service:

- It has to know which exceptions are likely to be transient. You want to retry errors caused by a temporary loss in network connectivity, not errors caused by program bugs, for example.

- It has to wait an appropriate amount of time between retries of a failed operation. You can wait longer between retries for a batch process than you can for an online web page where a user is waiting for a response.
- It has to retry an appropriate number of times before it gives up. You might want to retry more times in a batch process that you would in an online application.

You can configure these settings manually for any database environment supported by an Entity Framework provider, but default values that typically work well for an online application that uses Windows Azure SQL Database have already been configured for you, and those are the settings you'll implement for the Contoso University application.

All you have to do to enable connection resiliency is create a class in your assembly that derives from the DbConfiguration class, and in that class set the SQL Database *execution strategy*, which in EF is another term for *retry policy*.

1. In the DAL folder, add a class file named *SchoolConfiguration.cs*.

2. Replace the template code with the following code:

```
using System.Data.Entity;
using System.Data.Entity.SqlServer;

namespace ContosoUniversity.DAL
{
    public class SchoolConfiguration : DbConfiguration
    {
        public SchoolConfiguration()
        {
            SetExecutionStrategy("System.Data.SqlClient", () => new SqlAzureExecutionStrategy());
        }
    }
}
```

   The Entity Framework automatically runs the code it finds in a class that derives from `DbConfiguration`. You can use the `DbConfiguration` class to do configuration tasks in code that you would otherwise do in the *Web.config* file. For more information, see EntityFramework Code-Based Configuration.

3. In *StudentController.cs*, add a `using` statement for `System.Data.Entity.Infrastructure`.

```
using System.Data.Entity.Infrastructure;
```

4. Change all of the `catch` blocks that catch `DataException` exceptions so that they catch `RetryLimitExceededException` exceptions instead. For example:

```
catch (RetryLimitExceededException /* dex */)
{
    //Log the error (uncomment dex variable name and add a line here to write a log.
    ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists see
your system administrator.");
}
```

   You were using `DataException` to try to identify errors that might be transient in order to give a friendly "try again" message. But now that you've turned on a retry policy, the only errors likely to be transient will already have been tried and failed several times and the actual exception returned will be wrapped in the `RetryLimitExceededException` exception.

For more information, see Entity Framework Connection Resiliency / Retry Logic.

Enable command interception

Now that you've turned on a retry policy, how do you test to verify that it is working as expected? It's not so easy to force a transient error to happen, especially when you're running locally, and it would be especially difficult to integrate actual transient errors into an automated unit test. To test the connection resiliency feature, you need a way to intercept queries that Entity Framework sends to SQL Server and replace the SQL Server response with an exception type that is typically transient.

You can also use query interception in order to implement a best practice for cloud applications: log the latency and success or failure of all calls to external services such as database services. EF6 provides a dedicated logging API that can make it easier to do logging, but in this section of the tutorial you'll learn how to use the Entity Framework's interception feature directly, both for logging and for simulating transient errors.

Create a logging interface and class

A best practice for logging is to do it by using an interface rather than hard-coding calls to System.Diagnostics.Trace or a logging class. That makes it easier to change your logging mechanism later if you ever need to do that. So in this section you'll create the logging interface and a class to implement it./p>

1. Create a folder in the project and name it *Logging*.

2. In the *Logging* folder, create a class file named *ILogger.cs*, and replace the template code with the following code:

```
using System;

namespace ContosoUniversity.Logging
{
    public interface ILogger
    {
        void Information(string message);
        void Information(string fmt, params object[] vars);
        void Information(Exception exception, string fmt, params object[] vars);

        void Warning(string message);
        void Warning(string fmt, params object[] vars);
        void Warning(Exception exception, string fmt, params object[] vars);

        void Error(string message);
        void Error(string fmt, params object[] vars);
        void Error(Exception exception, string fmt, params object[] vars);

        void TraceApi(string componentName, string method, TimeSpan timespan);
        void TraceApi(string componentName, string method, TimeSpan timespan, string properties);
        void TraceApi(string componentName, string method, TimeSpan timespan, string fmt, params
object[] vars);
    }
}
```

   The interface provides three tracing levels to indicate the relative importance of logs, and one designed to provide latency information for external service calls such as database queries. The logging methods have overloads that let you pass in an exception. This is so that exception information including stack trace and inner exceptions is reliably logged by the class that implements the interface, instead of relying on that being done in each logging method call throughout the application.

   The TraceApi methods enable you to track the latency of each call to an external service such as SQL Database.

3. In the *Logging* folder, create a class file named *Logger.cs*, and replace the template code with the following code:

```
using System;
using System.Diagnostics;
using System.Text;

namespace ContosoUniversity.Logging
{
    public class Logger : ILogger
    {
        public void Information(string message)
        {
            Trace.TraceInformation(message);
        }

        public void Information(string fmt, params object[] vars)
        {
            Trace.TraceInformation(fmt, vars);
        }

        public void Information(Exception exception, string fmt, params object[] vars)
        {
            Trace.TraceInformation(FormatExceptionMessage(exception, fmt, vars));
        }

        public void Warning(string message)
        {
            Trace.TraceWarning(message);
        }

        public void Warning(string fmt, params object[] vars)
        {
            Trace.TraceWarning(fmt, vars);
        }

        public void Warning(Exception exception, string fmt, params object[] vars)
        {
            Trace.TraceWarning(FormatExceptionMessage(exception, fmt, vars));
        }

        public void Error(string message)
        {
            Trace.TraceError(message);
        }

        public void Error(string fmt, params object[] vars)
        {
            Trace.TraceError(fmt, vars);
        }

        public void Error(Exception exception, string fmt, params object[] vars)
        {
            Trace.TraceError(FormatExceptionMessage(exception, fmt, vars));
        }

        public void TraceApi(string componentName, string method, TimeSpan timespan)
        {
            TraceApi(componentName, method, timespan, "");
        }

        public void TraceApi(string componentName, string method, TimeSpan timespan, string fmt, params
object[] vars)
        {
            TraceApi(componentName, method, timespan, string.Format(fmt, vars));
        }
        public void TraceApi(string componentName, string method, TimeSpan timespan, string properties)
        {
            string message = String.Concat("Component:", componentName, ";Method:", method,
";Timespan:", timespan.ToString(), ";Properties:", properties);
            Trace.TraceInformation(message);
```

```
        }

        private static string FormatExceptionMessage(Exception exception, string fmt, object[] vars)
        {
            // Simple exception formatting: for a more comprehensive version see
            // https://code.msdn.microsoft.com/windowsazure/Fix-It-app-for-Building-cdd80df4
            var sb = new StringBuilder();
            sb.Append(string.Format(fmt, vars));
            sb.Append(" Exception: ");
            sb.Append(exception.ToString());
            return  sb.ToString();
        }
    }
}
```

The implementation uses System.Diagnostics to do the tracing. This is a built-in feature of .NET which makes it easy to generate and use tracing information. There are many "listeners" you can use with System.Diagnostics tracing, to write logs to files, for example, or to write them to blob storage in Azure. See some of the options, and links to other resources for more information, in Troubleshooting Azure Web Sites in Visual Studio. For this tutorial you'll only look at logs in the Visual Studio **Output** window.

In a production application you might want to consider tracing packages other than System.Diagnostics, and the ILogger interface makes it relatively easy to switch to a different tracing mechanism if you decide to do that.

Create interceptor classes

Next you'll create the classes that the Entity Framework will call into every time it is going to send a query to the database, one to simulate transient errors and one to do logging. These interceptor classes must derive from the `DbCommandInterceptor` class. In them you write method overrides that are automatically called when query is about to be executed. In these methods you can examine or log the query that is being sent to the database, and you can change the query before it's sent to the database or return something to Entity Framework yourself without even passing the query to the database.

1. To create the interceptor class that will log every SQL query that is sent to the database, create a class file named *SchoolInterceptorLogging.cs* in the *DAL* folder, and replace the template code with the following code:

```
using System;
using System.Data.Common;
using System.Data.Entity;
using System.Data.Entity.Infrastructure.Interception;
using System.Data.Entity.SqlServer;
using System.Data.SqlClient;
using System.Diagnostics;
using System.Reflection;
using System.Linq;
using ContosoUniversity.Logging;

namespace ContosoUniversity.DAL
{
    public class SchoolInterceptorLogging : DbCommandInterceptor
    {
        private ILogger _logger = new Logger();
        private readonly Stopwatch _stopwatch = new Stopwatch();

        public override void ScalarExecuting(DbCommand command, DbCommandInterceptionContext<object>
interceptionContext)
        {
            base.ScalarExecuting(command, interceptionContext);
            _stopwatch.Restart();
        }

        public override void ScalarExecuted(DbCommand command, DbCommandInterceptionContext<object>
```

```
        public override void ScalarExecuted(DbCommand command, DbCommandInterceptionContext<object>
interceptionContext)
        {
            _stopwatch.Stop();
            if (interceptionContext.Exception != null)
            {
                _logger.Error(interceptionContext.Exception, "Error executing command: {0}",
command.CommandText);
            }
            else
            {
                _logger.TraceApi("SQL Database", "SchoolInterceptor.ScalarExecuted",
_stopwatch.Elapsed, "Command: {0}: ", command.CommandText);
            }
            base.ScalarExecuted(command, interceptionContext);
        }

        public override void NonQueryExecuting(DbCommand command, DbCommandInterceptionContext<int>
interceptionContext)
        {
            base.NonQueryExecuting(command, interceptionContext);
            _stopwatch.Restart();
        }

        public override void NonQueryExecuted(DbCommand command, DbCommandInterceptionContext<int>
interceptionContext)
        {
            _stopwatch.Stop();
            if (interceptionContext.Exception != null)
            {
                _logger.Error(interceptionContext.Exception, "Error executing command: {0}",
command.CommandText);
            }
            else
            {
                _logger.TraceApi("SQL Database", "SchoolInterceptor.NonQueryExecuted",
_stopwatch.Elapsed, "Command: {0}: ", command.CommandText);
            }
            base.NonQueryExecuted(command, interceptionContext);
        }

        public override void ReaderExecuting(DbCommand command,
DbCommandInterceptionContext<DbDataReader> interceptionContext)
        {
            base.ReaderExecuting(command, interceptionContext);
            _stopwatch.Restart();
        }
        public override void ReaderExecuted(DbCommand command,
DbCommandInterceptionContext<DbDataReader> interceptionContext)
        {
            _stopwatch.Stop();
            if (interceptionContext.Exception != null)
            {
                _logger.Error(interceptionContext.Exception, "Error executing command: {0}",
command.CommandText);
            }
            else
            {
                _logger.TraceApi("SQL Database", "SchoolInterceptor.ReaderExecuted",
_stopwatch.Elapsed, "Command: {0}: ", command.CommandText);
            }
            base.ReaderExecuted(command, interceptionContext);
        }
    }
}
```

For successful queries or commands, this code writes an Information log with latency information. For exceptions, it creates an Error log.

2. To create the interceptor class that will generate dummy transient errors when you enter "Throw" in the **Search** box, create a class file named *SchoolInterceptorTransientErrors.cs* in the *DAL* folder, and replace the template code with the following code:

```
using System;
using System.Data.Common;
using System.Data.Entity;
using System.Data.Entity.Infrastructure.Interception;
using System.Data.Entity.SqlServer;
using System.Data.SqlClient;
using System.Diagnostics;
using System.Reflection;
using System.Linq;
using ContosoUniversity.Logging;

namespace ContosoUniversity.DAL
{
    public class SchoolInterceptorTransientErrors : DbCommandInterceptor
    {
        private int _counter = 0;
        private ILogger _logger = new Logger();

        public override void ReaderExecuting(DbCommand command,
DbCommandInterceptionContext<DbDataReader> interceptionContext)
        {
            bool throwTransientErrors = false;
            if (command.Parameters.Count > 0 && command.Parameters[0].Value.ToString() == "%Throw%")
            {
                throwTransientErrors = true;
                command.Parameters[0].Value = "%an%";
                command.Parameters[1].Value = "%an%";
            }

            if (throwTransientErrors && _counter < 4)
            {
                _logger.Information("Returning transient error for command: {0}", command.CommandText);
                _counter++;
                interceptionContext.Exception = CreateDummySqlException();
            }
        }

        private SqlException CreateDummySqlException()
        {
            // The instance of SQL Server you attempted to connect to does not support encryption
            var sqlErrorNumber = 20;

            var sqlErrorCtor = typeof(SqlError).GetConstructors(BindingFlags.Instance |
BindingFlags.NonPublic).Where(c => c.GetParameters().Count() == 7).Single();
            var sqlError = sqlErrorCtor.Invoke(new object[] { sqlErrorNumber, (byte)0, (byte)0, "", "",
"", 1 });

            var errorCollection = Activator.CreateInstance(typeof(SqlErrorCollection), true);
            var addMethod = typeof(SqlErrorCollection).GetMethod("Add", BindingFlags.Instance |
BindingFlags.NonPublic);
            addMethod.Invoke(errorCollection, new[] { sqlError });

            var sqlExceptionCtor = typeof(SqlException).GetConstructors(BindingFlags.Instance |
BindingFlags.NonPublic).Where(c => c.GetParameters().Count() == 4).Single();
            var sqlException = (SqlException)sqlExceptionCtor.Invoke(new object[] { "Dummy",
errorCollection, null, Guid.NewGuid() });

            return sqlException;
        }
    }
}
```

This code only overrides the `ReaderExecuting` method, which is called for queries that can return multiple rows of data. If you wanted to check connection resiliency for other types of queries, you could also override the `NonQueryExecuting` and `ScalarExecuting` methods, as the logging interceptor does.

When you run the Student page and enter "Throw" as the search string, this code creates a dummy SQL Database exception for error number 20, a type known to be typically transient. Other error numbers currently recognized as transient are 64, 233, 10053, 10054, 10060, 10928, 10929, 40197, 40501, and 40613, but these are subject to change in new versions of SQL Database.

The code returns the exception to Entity Framework instead of running the query and passing back query results. The transient exception is returned four times, and then the code reverts to the normal procedure of passing the query to the database.

Because everything is logged, you'll be able to see that Entity Framework tries to execute the query four times before finally succeeding, and the only difference in the application is that it takes longer to render a page with query results.

The number of times the Entity Framework will retry is configurable; the code specifies four times because that's the default value for the SQL Database execution policy. If you change the execution policy, you'd also change the code here that specifies how many times transient errors are generated. You could also change the code to generate more exceptions so that Entity Framework will throw the `RetryLimitExceededException` exception.

The value you enter in the Search box will be in `command.Parameters[0]` and `command.Parameters[1]` (one is used for the first name and one for the last name). When the value "%Throw%" is found, "Throw" is replaced in those parameters by "an" so that some students will be found and returned.

This is just a convenient way to test connection resiliency based on changing some input to the application UI. You can also write code that generates transient errors for all queries or updates, as explained later in the comments about the *DbInterception.Add* method.

3. In *Global.asax*, add the following `using` statements:

```
using ContosoUniversity.DAL;
using System.Data.Entity.Infrastructure.Interception;
```

4. Add the highlighted lines to the `Application_Start` method:

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
    DbInterception.Add(new SchoolInterceptorTransientErrors());
    DbInterception.Add(new SchoolInterceptorLogging());
}
```

These lines of code are what causes your interceptor code to be run when Entity Framework sends queries to the database. Notice that because you created separate interceptor classes for transient error simulation and logging, you can independently enable and disable them.

You can add interceptors using the `DbInterception.Add` method anywhere in your code; it doesn't have to be in the `Application_Start` method. Another option is to put this code in the DbConfiguration class that you created earlier to configure the execution policy.

```
public class SchoolConfiguration : DbConfiguration
{
    public SchoolConfiguration()
    {
        SetExecutionStrategy("System.Data.SqlClient", () => new SqlAzureExecutionStrategy());
        DbInterception.Add(new SchoolInterceptorTransientErrors());
        DbInterception.Add(new SchoolInterceptorLogging());
    }
}
```

Wherever you put this code, be careful not to execute `DbInterception.Add` for the same interceptor more than once, or you'll get additional interceptor instances. For example, if you add the logging interceptor twice, you'll see two logs for every SQL query.

Interceptors are executed in the order of registration (the order in which the `DbInterception.Add` method is called). The order might matter depending on what you're doing in the interceptor. For example, an interceptor might change the SQL command that it gets in the `CommandText` property. If it does change the SQL command, the next interceptor will get the changed SQL command, not the original SQL command.

You've written the transient error simulation code in a way that lets you cause transient errors by entering a different value in the UI. As an alternative, you could write the interceptor code to always generate the sequence of transient exceptions without checking for a particular parameter value. You could then add the interceptor only when you want to generate transient errors. If you do this, however, don't add the interceptor until after database initialization has completed. In other words, do at least one database operation such as a query on one of your entity sets before you start generating transient errors. The Entity Framework executes several queries during database initialization, and they aren't executed in a transaction, so errors during initialization could cause the context to get into an inconsistent state.

Test the new configuration

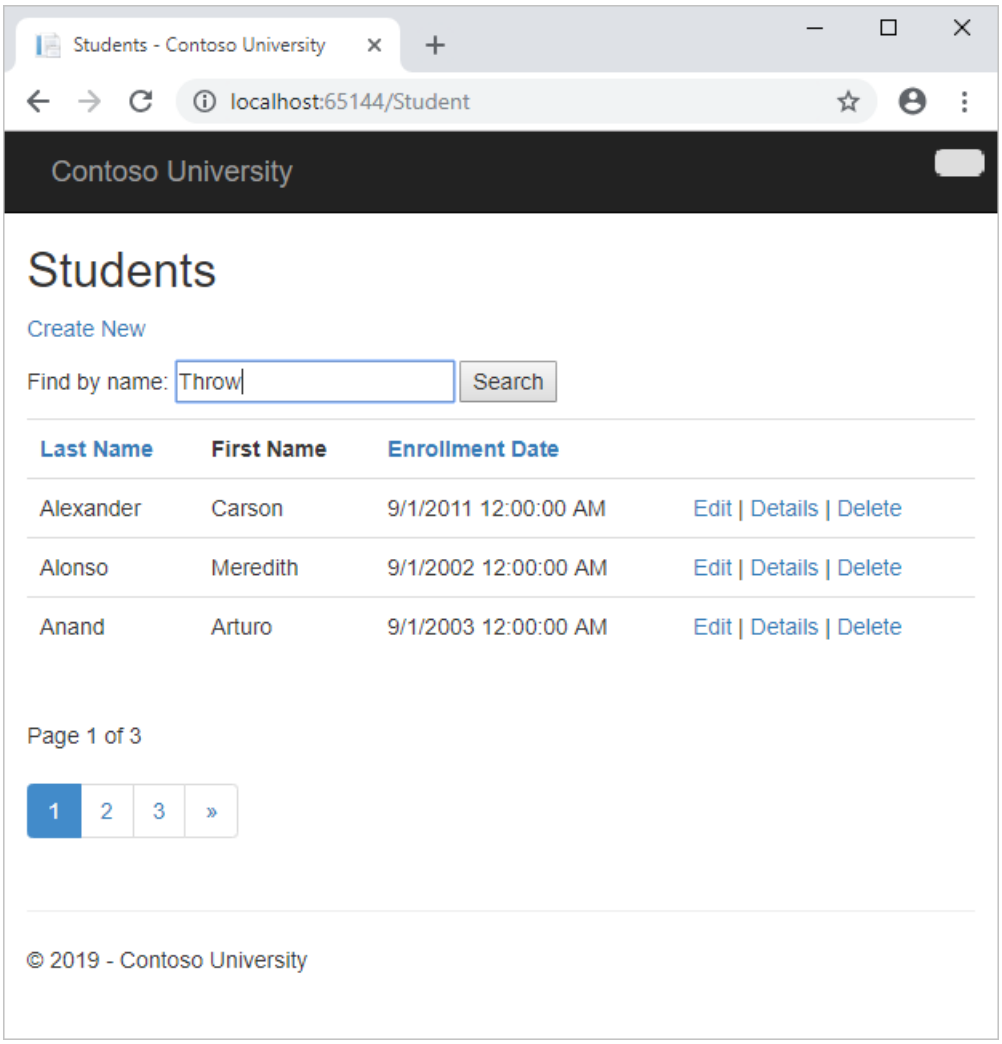
1. Press **F5** to run the application in debug mode, and then click the **Students** tab.

2. Look at the Visual Studio **Output** window to see the tracing output. You might have to scroll up past some JavaScript errors to get to the logs written by your logger.

   Notice that you can see the actual SQL queries sent to the database. You see some initial queries and commands that Entity Framework does to get started, checking the database version and migration history table (you'll learn about migrations in the next tutorial). And you see a query for paging, to find out how many students there are, and finally you see the query that gets the student data.

```
Output
Show output from: Debug
iisexpress.exe Information: 0 : Component:SQL
  Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0276705;Properties
  :Command: select serverproperty('EngineEdition'):
iisexpress.exe Information: 0 : Component:SQL
  Database;Method:SchoolInterceptor.ScalarExecuted;Timespan:00:00:00.0424968;Properties
  :Command: SELECT Count(*) FROM sys.databases WHERE [name]=N'ContosoUniversity2':
iisexpress.exe Information: 0 : Component:SQL
  Database;Method:SchoolInterceptor.ScalarExecuted;Timespan:00:00:00.0450531;Properties
  :Command:
SELECT Count(*)
FROM INFORMATION_SCHEMA.TABLES AS t
WHERE t.TABLE_TYPE = 'BASE TABLE'
    AND (t.TABLE_SCHEMA + '.' + t.TABLE_NAME IN
      ('dbo.Course','dbo.Department','dbo.Instructor','dbo.OfficeAssignment','dbo.Enrol
      lment','dbo.Student','dbo.CourseInstructor')
        OR t.TABLE_NAME = 'EdmMetadata'):
iisexpress.exe Information: 0 : Component:SQL
  Database;Method:SchoolInterceptor.ScalarExecuted;Timespan:00:00:00.0262633;Properties
  :Command: SELECT Count(*) FROM sys.databases WHERE [name]=N'ContosoUniversity2':
iisexpress.exe Information: 0 : Component:SQL
  Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0373571;Properties
  :Command: SELECT
    [GroupBy1].[A1] AS [C1]
    FROM ( SELECT
        COUNT(1) AS [A1]
        FROM [dbo].[__MigrationHistory] AS [Extent1]
    )  AS [GroupBy1]:
iisexpress.exe Information: 0 : Component:SQL
  Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0227459;Properties
  :Command: SELECT
    [GroupBy1].[A1] AS [C1]
    FROM ( SELECT
        COUNT(1) AS [A1]
        FROM [dbo].[__MigrationHistory] AS [Extent1]
        WHERE ([Extent1].[ContextKey] = @p__linq__0) AND (@p__linq__0 IS NOT NULL)
    )  AS [GroupBy1]:
iisexpress.exe Information: 0 : Component:SQL
  Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0298657;Properties
  :Command: SELECT TOP (1)
    [Project1].[C1] AS [C1],
    [Project1].[MigrationId] AS [MigrationId],
    [Project1].[Model] AS [Model]
    FROM ( SELECT
        [Extent1].[MigrationId] AS [MigrationId],
        [Extent1].[Model] AS [Model],
        1 AS [C1]
        FROM [dbo].[__MigrationHistory] AS [Extent1]
        WHERE ([Extent1].[ContextKey] = @p__linq__0) AND (@p__linq__0 IS NOT NULL)
    )  AS [Project1]
    ORDER BY [Project1].[MigrationId] DESC:
iisexpress.exe Information: 0 : Component:SQL
  Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0246607;Properties
  :Command: SELECT
    [GroupBy1].[A1] AS [C1]
    FROM ( SELECT
        COUNT(1) AS [A1]
        FROM [dbo].[Student] AS [Extent1]
    )  AS [GroupBy1]:
'iisexpress.exe' (CLR v4.0.30319: /LM/W3SVC/12/ROOT-1-130329870303366864): Loaded
  'EntityFrameworkDynamicProxies-ContosoUniversity'.
iisexpress.exe Information: 0 : Component:SQL
  Database;Method:SchoolInterceptor.ReaderExecuted;Timespan:00:00:00.0225311;Properties
  :Command: SELECT TOP (3)
    [Extent1].[ID] AS [ID],
    [Extent1].[LastName] AS [LastName],
    [Extent1].[FirstName] AS [FirstName],
    [Extent1].[EnrollmentDate] AS [EnrollmentDate]
    FROM ( SELECT [Extent1].[ID] AS [ID], [Extent1].[LastName] AS [LastName],
      [Extent1].[FirstName] AS [FirstName], [Extent1].[EnrollmentDate] AS
      [EnrollmentDate], row_number() OVER (ORDER BY [Extent1].[LastName] ASC) AS
      [row_number]
        FROM [dbo].[Student] AS [Extent1]
    )  AS [Extent1]
    WHERE [Extent1].[row_number] > 0
    ORDER BY [Extent1].[LastName] ASC:
```

3. In the **Students** page, enter "Throw" as the search string, and click **Search**.



You'll notice that the browser seems to hang for several seconds while Entity Framework is retrying the query several times. The first retry happens very quickly, then the wait before increases before each additional retry. This process of waiting longer before each retry is called *exponential backoff*.

When the page displays, showing students who have "an" in their names, look at the output window, and you'll see that the same query was attempted five times, the first four times returning transient exceptions. For each transient error you'll see the log that you write when generating the transient error in the `SchoolInterceptorTransientErrors` class ("Returning transient error for command...") and you'll see the log written when `SchoolInterceptorLogging` gets the exception.

```
Output
Show output from: Debug
iisexpress.exe Information: 0 : Returning transient error for command: SELECT
    [GroupBy1].[A1] AS [C1]
    FROM ( SELECT
        COUNT(1) AS [A1]
        FROM [dbo].[Student] AS [Extent1]
        WHERE (( CAST(CHARINDEX(UPPER(@p__linq__0), UPPER([Extent1].[LastName])) AS
          int)) > 0) OR (( CAST(CHARINDEX(UPPER(@p__linq__1), UPPER([Extent1].
          [FirstName])) AS int)) > 0)
    )  AS [GroupBy1]
iisexpress.exe Error: 0 : Error executing command: SELECT
    [GroupBy1].[A1] AS [C1]
    FROM ( SELECT
        COUNT(1) AS [A1]
        FROM [dbo].[Student] AS [Extent1]
        WHERE (( CAST(CHARINDEX(UPPER(@p__linq__0), UPPER([Extent1].[LastName])) AS
          int)) > 0) OR (( CAST(CHARINDEX(UPPER(@p__linq__1), UPPER([Extent1].
          [FirstName])) AS int)) > 0)
    )  AS [GroupBy1] Exception: System.Data.SqlClient.SqlException (0x80131904): Dummy
ClientConnectionId:5430f70e-b790-4798-ac6d-d28f91ec3529
iisexpress.exe Information: 0 : Returning transient error for command: SELECT
    [GroupBy1].[A1] AS [C1]
    FROM ( SELECT
        COUNT(1) AS [A1]
        FROM [dbo].[Student] AS [Extent1]
        WHERE (( CAST(CHARINDEX(UPPER(@p__linq__0), UPPER([Extent1].[LastName])) AS
          int)) > 0) OR (( CAST(CHARINDEX(UPPER(@p__linq__1), UPPER([Extent1].
          [FirstName])) AS int)) > 0)
    )  AS [GroupBy1]
iisexpress.exe Error: 0 : Error executing command: SELECT
    [GroupBy1].[A1] AS [C1]
    FROM ( SELECT
        COUNT(1) AS [A1]
        FROM [dbo].[Student] AS [Extent1]
        WHERE (( CAST(CHARINDEX(UPPER(@p__linq__0), UPPER([Extent1].[LastName])) AS
          int)) > 0) OR (( CAST(CHARINDEX(UPPER(@p__linq__1), UPPER([Extent1].
          [FirstName])) AS int)) > 0)
    )  AS [GroupBy1] Exception: System.Data.SqlClient.SqlException (0x80131904): Dummy
ClientConnectionId:bf3d3750-18e6-4e20-9ce7-a31ccd41d74b
```

Since you entered a search string, the query that returns student data is parameterized:

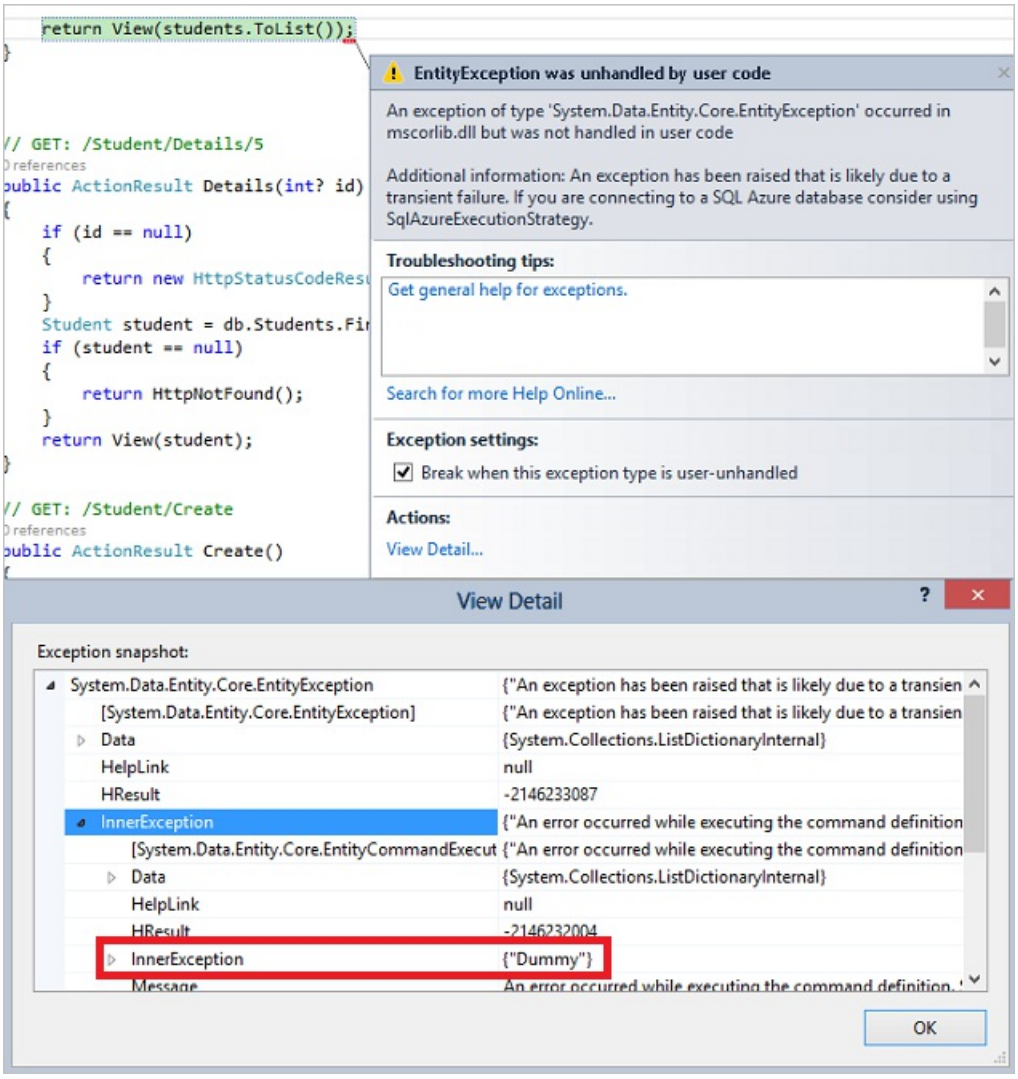
```
SELECT TOP (3)
    [Project1].[ID] AS [ID],
    [Project1].[LastName] AS [LastName],
    [Project1].[FirstMidName] AS [FirstMidName],
    [Project1].[EnrollmentDate] AS [EnrollmentDate]
    FROM ( SELECT [Project1].[ID] AS [ID], [Project1].[LastName] AS [LastName], [Project1].
[FirstMidName] AS [FirstMidName], [Project1].[EnrollmentDate] AS [EnrollmentDate], row_number() OVER
(ORDER BY [Project1].[LastName] ASC) AS [row_number]
        FROM ( SELECT
            [Extent1].[ID] AS [ID],
            [Extent1].[LastName] AS [LastName],
            [Extent1].[FirstMidName] AS [FirstMidName],
            [Extent1].[EnrollmentDate] AS [EnrollmentDate]
            FROM [dbo].[Student] AS [Extent1]
            WHERE ([Extent1].[LastName] LIKE @p__linq__0 ESCAPE N'~') OR ([Extent1].[FirstMidName] LIKE
@p__linq__1 ESCAPE N'~')
        )  AS [Project1]
    )  AS [Project1]
    WHERE [Project1].[row_number] > 0
    ORDER BY [Project1].[LastName] ASC:
```

You're not logging the value of the parameters, but you could do that. If you want to see the parameter values, you can write logging code to get parameter values from the `Parameters` property of the `DbCommand` object that you get in the interceptor methods.

Note that you can't repeat this test unless you stop the application and restart it. If you wanted to be able to test connection resiliency multiple times in a single run of the application, you could write code to reset the error counter in `SchoolInterceptorTransientErrors`.

4. To see the difference the execution strategy (retry policy) makes, comment out the `SetExecutionStrategy` line in *SchoolConfiguration.cs*, run the Students page in debug mode again, and search for "Throw" again.

   This time the debugger stops on the first generated exception immediately when it tries to execute the query the first time.



5. Uncomment the *SetExecutionStrategy* line in *SchoolConfiguration.cs*.

Get the code

Download Completed Project

Additional resources

Links to other Entity Framework resources can be found in ASP.NET Data Access - Recommended Resources.

Next steps

In this tutorial, you:

- Enabled connection resiliency
- Enabled command interception
- Tested the new configuration

Advance to the next article to learn about Code First migrations and Azure deployment.

Code First migrations and Azure deployment

Tutorial: Use EF Migrations in an ASP.NET MVC app and deploy to Azure

9/16/2020 • 20 minutes to read • Edit Online

So far the Contoso University sample web application has been running locally in IIS Express on your development computer. To make a real application available for other people to use over the Internet, you have to deploy it to a web hosting provider. In this tutorial, you enable Code First migrations and deploy the application to the cloud in Azure:

- Enable Code First Migrations. The Migrations feature enables you to change the data model and deploy your changes to production by updating the database schema without having to drop and re-create the database.
- Deploy to Azure. This step is optional; you can continue with the remaining tutorials without having deployed the project.

We recommend that you use a continuous integration process with source control for deployment, but this tutorial does not cover those topics. For more information, see the source control and continuous integration chapters of Building Real-World Cloud Apps with Azure.

In this tutorial, you:

- Enable Code First migrations
- Deploy the app in Azure (optional)

Prerequisites

- Connection Resiliency and Command Interception

Enable Code First migrations

When you develop a new application, your data model changes frequently, and each time the model changes, it gets out of sync with the database. You have configured the Entity Framework to automatically drop and re-create the database each time you change the data model. When you add, remove, or change entity classes or change your `DbContext` class, the next time you run the application it automatically deletes your existing database, creates a new one that matches the model, and seeds it with test data.

This method of keeping the database in sync with the data model works well until you deploy the application to production. When the application is running in production, it is usually storing data that you want to keep, and you don't want to lose everything each time you make a change such as adding a new column. The Code First Migrations feature solves this problem by enabling Code First to update the database schema instead of dropping and re-creating the database. In this tutorial, you'll deploy the application, and to prepare for that you'll enable Migrations.

1. Disable the initializer that you set up earlier by commenting out or deleting the `contexts` element that you added to the application Web.config file.

```
<entityFramework>
  <!--<contexts>
    <context type="ContosoUniversity.DAL.SchoolContext, ContosoUniversity">
      <databaseInitializer type="ContosoUniversity.DAL.SchoolInitializer, ContosoUniversity" />
    </context>
  </contexts>-->
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
EntityFramework">
    <parameters>
      <parameter value="v11.0" />
    </parameters>
  </defaultConnectionFactory>
  <providers>
    <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
  </providers>
</entityFramework>
```

2. Also in the application *Web.config* file, change the name of the database in the connection string to ContosoUniversity2.

```
<connectionStrings>
  <add name="SchoolContext" connectionString="Data Source=(LocalDb)\MSSQLLocalDB;Initial
Catalog=ContosoUniversity2;Integrated Security=SSPI;" providerName="System.Data.SqlClient" />
</connectionStrings>
```

   This change sets up the project so that the first migration creates a new database. This isn't required but you'll see later why it's a good idea.

3. From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.

4. At the `PM>` prompt enter the following commands:

```
enable-migrations
add-migration InitialCreate
```

   The `enable-migrations` command creates a *Migrations* folder in the ContosoUniversity project, and it puts in that folder a *Configuration.cs* file that you can edit to configure Migrations.

   (If you missed the step above that directs you to change the database name, Migrations will find the existing database and automatically do the `add-migration` command. That's okay, it just means you won't run a test of the migrations code before you deploy the database. Later when you run the `update-database` command nothing will happen because the database already exists.)

   Open the *ContosoUniversity\Migrations\Configuration.cs* file. Like the initializer class that you saw earlier, the `Configuration` class includes a `Seed` method.

```
internal sealed class Configuration : DbMigrationsConfiguration<ContosoUniversity.DAL.SchoolContext>
{
    public Configuration()
    {
        AutomaticMigrationsEnabled = false;
    }

    protected override void Seed(ContosoUniversity.DAL.SchoolContext context)
    {
        //  This method will be called after migrating to the latest version.

        //  You can use the DbSet<T>.AddOrUpdate() helper extension method
        //  to avoid creating duplicate seed data. E.g.
        //
        //    context.People.AddOrUpdate(
        //      p => p.FullName,
        //      new Person { FullName = "Andrew Peters" },
        //      new Person { FullName = "Brice Lambson" },
        //      new Person { FullName = "Rowan Miller" }
        //    );
        //
    }
}
```

The purpose of the Seed method is to enable you to insert or update test data after Code First creates or updates the database. The method is called when the database is created and every time the database schema is updated after a data model change.

Set up the Seed method

When you drop and re-create the database for every data model change, you use the initializer class's `Seed` method to insert test data, because after every model change the database is dropped and all the test data is lost. With Code First Migrations, test data is retained after database changes, so including test data in the Seed method is typically not necessary. In fact, you don't want the `Seed` method to insert test data if you'll be using Migrations to deploy the database to production, because the `Seed` method will run in production. In that case, you want the `Seed` method to insert into the database only the data that you need in production. For example, you might want the database to include actual department names in the `Department` table when the application becomes available in production.

For this tutorial, you'll be using Migrations for deployment, but your `Seed` method will insert test data anyway in order to make it easier to see how application functionality works without having to manually insert a lot of data.

1. Replace the contents of the *Configuration.cs* file with the following code, which loads test data into the new database.

```
namespace ContosoUniversity.Migrations
{
    using ContosoUniversity.Models;
    using System;
    using System.Collections.Generic;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    internal sealed class Configuration :
DbMigrationsConfiguration<ContosoUniversity.DAL.SchoolContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
        }

        protected override void Seed(ContosoUniversity.DAL.SchoolContext context)
```

```
        protected override void Seed(ContosoUniversity.DAL.SchoolContext context)
        {
            var students = new List<Student>
            {
                new Student { FirstMidName = "Carson",   LastName = "Alexander",
                    EnrollmentDate = DateTime.Parse("2010-09-01") },
                new Student { FirstMidName = "Meredith", LastName = "Alonso",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Arturo",   LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Gytis",    LastName = "Barzdukas",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Yan",      LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Peggy",    LastName = "Justice",
                    EnrollmentDate = DateTime.Parse("2011-09-01") },
                new Student { FirstMidName = "Laura",    LastName = "Norman",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Nino",     LastName = "Olivetto",
                    EnrollmentDate = DateTime.Parse("2005-08-11") }
            };
            students.ForEach(s => context.Students.AddOrUpdate(p => p.LastName, s));
            context.SaveChanges();

            var courses = new List<Course>
            {
                new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3, },
                new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3, },
                new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3, },
                new Course {CourseID = 1045, Title = "Calculus",       Credits = 4, },
                new Course {CourseID = 3141, Title = "Trigonometry",   Credits = 4, },
                new Course {CourseID = 2021, Title = "Composition",    Credits = 3, },
                new Course {CourseID = 2042, Title = "Literature",     Credits = 4, }
            };
            courses.ForEach(s => context.Courses.AddOrUpdate(p => p.Title, s));
            context.SaveChanges();

            var enrollments = new List<Enrollment>
            {
                new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Alexander").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
                    Grade = Grade.A
                },
                 new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Alexander").ID,
                    CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
                    Grade = Grade.C
                 },
                 new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Alexander").ID,
                    CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
                    Grade = Grade.B
                 },
                 new Enrollment {
                     StudentID = students.Single(s => s.LastName == "Alonso").ID,
                    CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
                    Grade = Grade.B
                 },
                 new Enrollment {
                     StudentID = students.Single(s => s.LastName == "Alonso").ID,
                    CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
                    Grade = Grade.B
                 },
                 new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Alonso").ID,
                    CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
                    Grade = Grade.B
                 },
```

```
                new Enrollment {
                   StudentID = students.Single(s => s.LastName == "Anand").ID,
                   CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
                },
                new Enrollment {
                   StudentID = students.Single(s => s.LastName == "Anand").ID,
                   CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
                   Grade = Grade.B
                },
               new Enrollment {
                   StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
                   CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
                   Grade = Grade.B
                },
                new Enrollment {
                   StudentID = students.Single(s => s.LastName == "Li").ID,
                   CourseID = courses.Single(c => c.Title == "Composition").CourseID,
                   Grade = Grade.B
                },
                new Enrollment {
                   StudentID = students.Single(s => s.LastName == "Justice").ID,
                   CourseID = courses.Single(c => c.Title == "Literature").CourseID,
                   Grade = Grade.B
                }
            };

            foreach (Enrollment e in enrollments)
            {
                var enrollmentInDataBase = context.Enrollments.Where(
                    s =>
                         s.Student.ID == e.StudentID &&
                         s.Course.CourseID == e.CourseID).SingleOrDefault();
                if (enrollmentInDataBase == null)
                {
                    context.Enrollments.Add(e);
                }
            }
            context.SaveChanges();
        }
    }
}
```

The Seed method takes the database context object as an input parameter, and the code in the method uses that object to add new entities to the database. For each entity type, the code creates a collection of new entities, adds them to the appropriate DbSet property, and then saves the changes to the database. It isn't necessary to call the SaveChanges method after each group of entities, as is done here, but doing that helps you locate the source of a problem if an exception occurs while the code is writing to the database.

Some of the statements that insert data use the AddOrUpdate method to perform an "upsert" operation. Because the `Seed` method runs every time you execute the `update-database` command, typically after each migration, you can't just insert data, because the rows you are trying to add will already be there after the first migration that creates the database. The "upsert" operation prevents errors that would happen if you try to insert a row that already exists, but it ***overrides*** any changes to data that you may have made while testing the application. With test data in some tables you might not want that to happen: in some cases when you change data while testing you want your changes to remain after database updates. In that case you want to do a conditional insert operation: insert a row only if it doesn't already exist. The Seed method uses both approaches.

The first parameter passed to the AddOrUpdate method specifies the property to use to check if a row already exists. For the test student data that you are providing, the `LastName` property can be used for this purpose since each last name in the list is unique:

```
context.Students.AddOrUpdate(p => p.LastName, s)
```

This code assumes that last names are unique. If you manually add a student with a duplicate last name, you'll get the following exception the next time you perform a migration:

```
Sequence contains more than one element
```

For information about how to handle redundant data such as two students named "Alexander Carson", see Seeding and Debugging Entity Framework (EF) DBs on Rick Anderson's blog. For more information about the `AddOrUpdate` method, see Take care with EF 4.3 AddOrUpdate Method on Julie Lerman's blog.

The code that creates `Enrollment` entities assumes you have the `ID` value in the entities in the `students` collection, although you didn't set that property in the code that creates the collection.

```
new Enrollment {
    StudentID = students.Single(s => s.LastName == "Alexander").ID,
    CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
    Grade = Grade.A
},
```

You can use the `ID` property here because the `ID` value is set when you call `SaveChanges` for the `students` collection. EF automatically gets the primary key value when it inserts an entity into the database, and it updates the `ID` property of the entity in memory.

The code that adds each `Enrollment` entity to the `Enrollments` entity set doesn't use the `AddOrUpdate` method. It checks if an entity already exists and inserts the entity if it doesn't exist. This approach will preserve changes you make to an enrollment grade by using the application UI. The code loops through each member of the `Enrollment` List and if the enrollment is not found in the database, it adds the enrollment to the database. The first time you update the database, the database will be empty, so it will add each enrollment.

```
foreach (Enrollment e in enrollments)
{
    var enrollmentInDataBase = context.Enrollments.Where(
        s => s.Student.ID == e.Student.ID &&
             s.Course.CourseID == e.Course.CourseID).SingleOrDefault();
    if (enrollmentInDataBase == null)
    {
        context.Enrollments.Add(e);
    }
}
```

2. Build the project.

Execute the first migration

When you executed the `add-migration` command, Migrations generated the code that would create the database from scratch. This code is also in the *Migrations* folder, in the file named *<timestamp>_InitialCreate.cs*. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them.

```
public partial class InitialCreate : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.Course",
            c => new
                {
                    CourseID = c.Int(nullable: false),
                    Title = c.String(),
                    Credits = c.Int(nullable: false),
                })
            .PrimaryKey(t => t.CourseID);

        CreateTable(
            "dbo.Enrollment",
            c => new
                {
                    EnrollmentID = c.Int(nullable: false, identity: true),
                    CourseID = c.Int(nullable: false),
                    StudentID = c.Int(nullable: false),
                    Grade = c.Int(),
                })
            .PrimaryKey(t => t.EnrollmentID)
            .ForeignKey("dbo.Course", t => t.CourseID, cascadeDelete: true)
            .ForeignKey("dbo.Student", t => t.StudentID, cascadeDelete: true)
            .Index(t => t.CourseID)
            .Index(t => t.StudentID);

        CreateTable(
            "dbo.Student",
            c => new
                {
                    ID = c.Int(nullable: false, identity: true),
                    LastName = c.String(),
                    FirstMidName = c.String(),
                    EnrollmentDate = c.DateTime(nullable: false),
                })
            .PrimaryKey(t => t.ID);

    }

    public override void Down()
    {
        DropForeignKey("dbo.Enrollment", "StudentID", "dbo.Student");
        DropForeignKey("dbo.Enrollment", "CourseID", "dbo.Course");
        DropIndex("dbo.Enrollment", new[] { "StudentID" });
        DropIndex("dbo.Enrollment", new[] { "CourseID" });
        DropTable("dbo.Student");
        DropTable("dbo.Enrollment");
        DropTable("dbo.Course");
    }
}
```

Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the `Down` method.

This is the initial migration that was created when you entered the `add-migration InitialCreate` command. The parameter (`InitialCreate` in the example) is used for the file name and can be whatever you want; you typically choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration "AddDepartmentTable".

If you created the initial migration when the database already exists, the database creation code is generated but it doesn't have to run because the database already matches the data model. When you deploy the app to another

environment where the database doesn't exist yet, this code will run to create your database, so it's a good idea to test it first. That's why you changed the name of the database in the connection string earlier—so that migrations can create a new one from scratch.

1. In the **Package Manager Console** window, enter the following command:

   ```
   update-database
   ```

   The `update-database` command runs the `Up` method to create the database and then it runs the `Seed` method to populate the database. The same process will run automatically in production after you deploy the application, as you'll see in the following section.

2. Use **Server Explorer** to inspect the database as you did in the first tutorial, and run the application to verify that everything still works the same as before.

Deploy to Azure

So far the application has been running locally in IIS Express on your development computer. To make it available for other people to use over the Internet, you have to deploy it to a web hosting provider. In this section of the tutorial, you'll deploy it to Azure. This section is optional; you can skip this and continue with the following tutorial, or you can adapt the instructions in this section for a different hosting provider of your choice.

Use Code First migrations to deploy the database

To deploy the database, you'll use Code First Migrations. When you create the publish profile that you use to configure settings for deploying from Visual Studio, you'll select a check box labeled **Update Database**. This setting causes the deployment process to automatically configure the application *Web.config* file on the destination server so that Code First uses the `MigrateDatabaseToLatestVersion` initializer class.

Visual Studio doesn't do anything with the database during the deployment process while it is copying your project to the destination server. When you run the deployed application and it accesses the database for the first time after deployment, Code First checks if the database matches the data model. If there's a mismatch, Code First automatically creates the database (if it doesn't exist yet) or updates the database schema to the latest version (if a database exists but doesn't match the model). If the application implements a Migrations `Seed` method, the method runs after the database is created or the schema is updated.

Your Migrations `Seed` method inserts test data. If you were deploying to a production environment, you would have to change the `Seed` method so that it only inserts data that you want to be inserted into your production database. For example, in your current data model you might want to have real courses but fictional students in the development database. You can write a `Seed` method to load both in development, and then comment out the fictional students before you deploy to production. Or you can write a `Seed` method to load only courses, and enter the fictional students in the test database manually by using the application's UI.

Get an Azure account

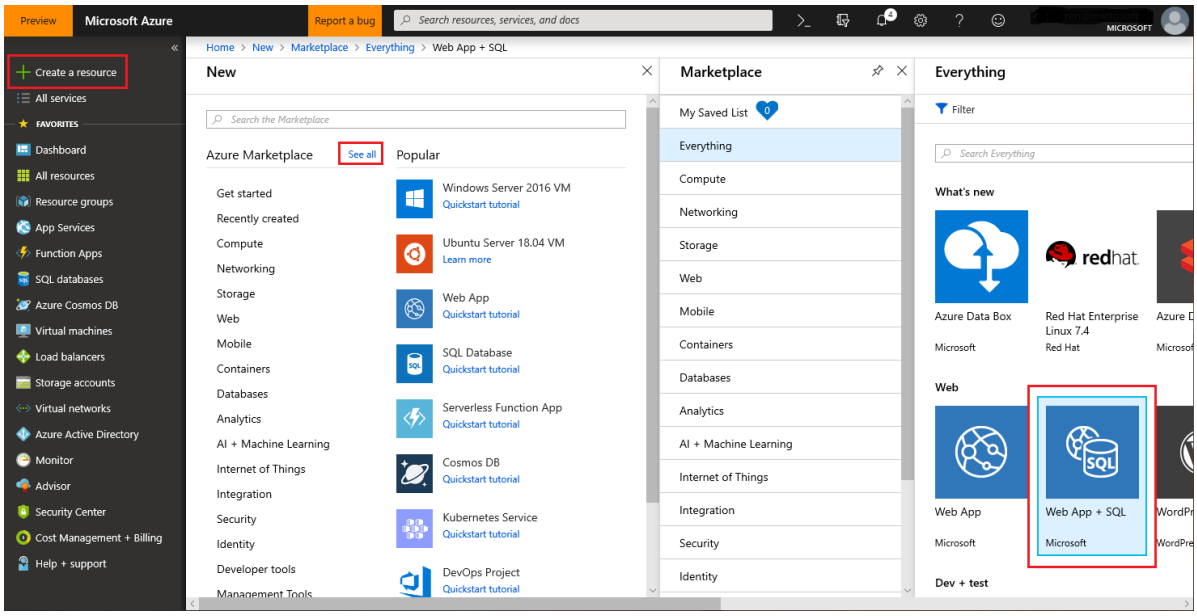
You'll need an Azure account. If you don't already have one, but you do have a Visual Studio subscription, you can activate your subscription benefits. Otherwise, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

Create a web site and a SQL database in Azure

Your web app in Azure will run in a shared hosting environment, which means it runs on virtual machines (VMs) that are shared with other Azure clients. A shared hosting environment is a low-cost way to get started in the cloud. Later, if your web traffic increases, the application can scale to meet the need by running on dedicated VMs. To learn more about Pricing Options for Azure App Service, read App Service pricing.

You'll deploy the database to Azure SQL database. SQL database is a cloud-based relational database service that is built on SQL Server technologies. Tools and applications that work with SQL Server also work with SQL database.

1. In the Azure Management Portal, choose **Create a resource** in the left tab and then choose **See all** on the **New** pane (or *blade*) to see all available resources. Choose **Web App + SQL** in the **Web** section of the **Everything** blade. Finally, choose **Create**.



The form to create a new **New Web App + SQL** resource opens.

2. Enter a string in the **App name** box to use as the unique URL for your application. The complete URL will consist of what you enter here plus the default domain of Azure App Services (.azurewebsites.net). If the **App name** is already taken, the Wizard notifies you with a red *The app name is not available* message. If the **App name** is available, you see a green checkmark.

3. In the **Subscription** box, choose the Azure Subscription in which you want the **App Service** to reside.

4. In the **Resource Group** text box, choose a Resource Group or create a new one. This setting specifies which data center your web site will run in. For more information about Resource Groups, see Resource groups.

5. Create a new **App Service Plan** by clicking the *App Service section*, **Create New**, and fill in **App Service plan** (can be same name as App Service), **Location**, and **Pricing tier** (there is a free option).

6. Click **SQL Database**, and then choose **Create a new database** or select an existing database.

7. In the **Name** box, enter a name for your database.

8. Click the **Target Server** box, and then select **Create a new server**. Alternatively, if you previously created a server, you can select that server from list of available servers.

9. Choose **Pricing tier** section, choose *Free*. If additional resources are needed, the database can be scaled up at any time. To learn more on Azure SQL Pricing, see Azure SQL Database pricing.

10. Modify collation as needed.

11. Enter an administrator **SQL Admin Username** and **SQL Admin Password**.

    - If you selected **New SQL Database server**, define a new name and password that you'll use later when you access the database.
    - If you selected a server that you created previously, enter credentials for that server.

12. Telemetry collection can be enabled for App Service using Application Insights. With little configuration, Application Insights collects valuable event, exception, dependency, request, and trace information. To learn more about Application Insights, see Azure Monitor.

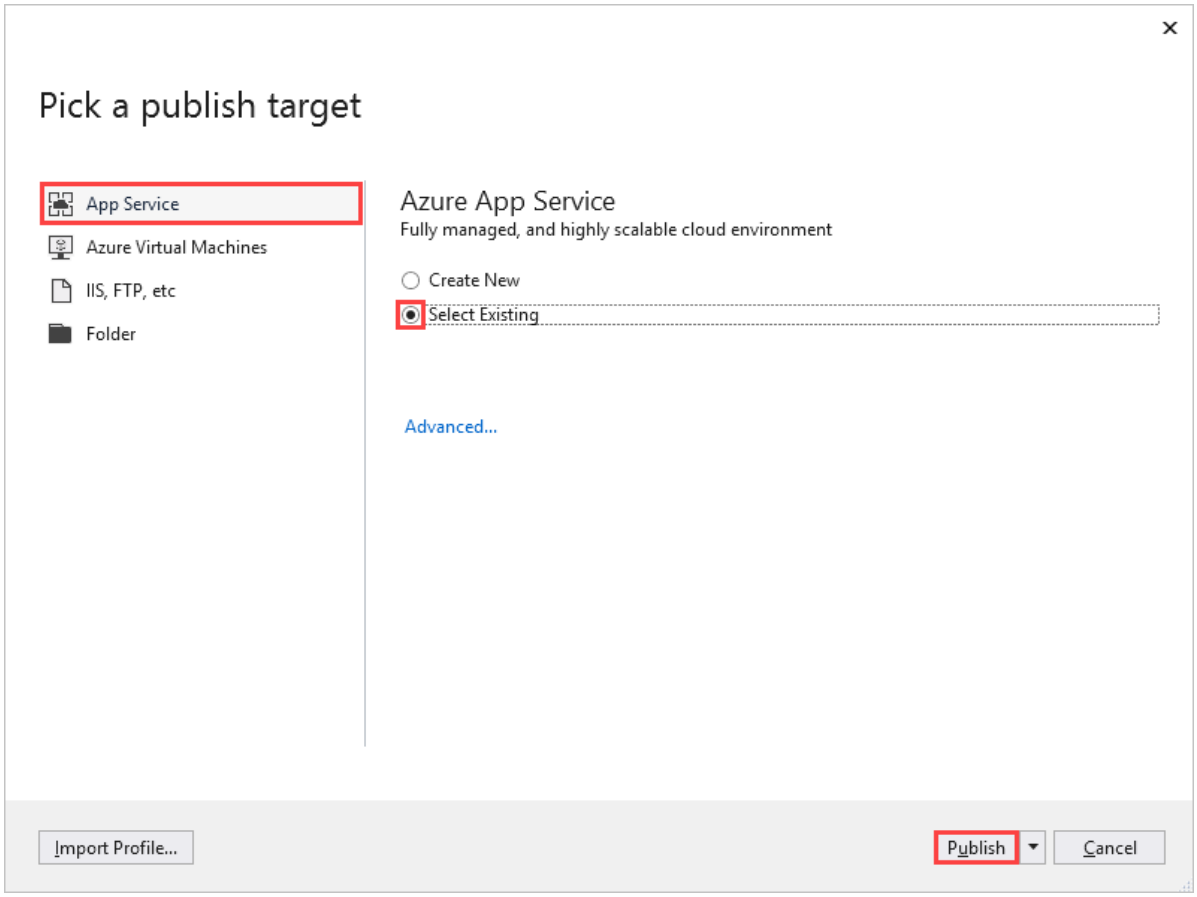
13. Click **Create** at the bottom to indicate that you're finished.

The Management Portal returns to the Dashboard page, and the **Notifications** area at the top of the page shows that the site is being created. After a while (typically less than a minute), there's a notification that the Deployment succeeded. In the navigation bar at the left, the new App Service appears in the **App Services** section and the new SQL database appears in the **SQL databases** section.

Deploy the app to Azure

1. In Visual Studio, right-click the project in **Solution Explorer** and select **Publish** from the context menu.

2. On the **Pick a publish target** page, choose **App Service** and then **Select Existing**, and then choose **Publish**.

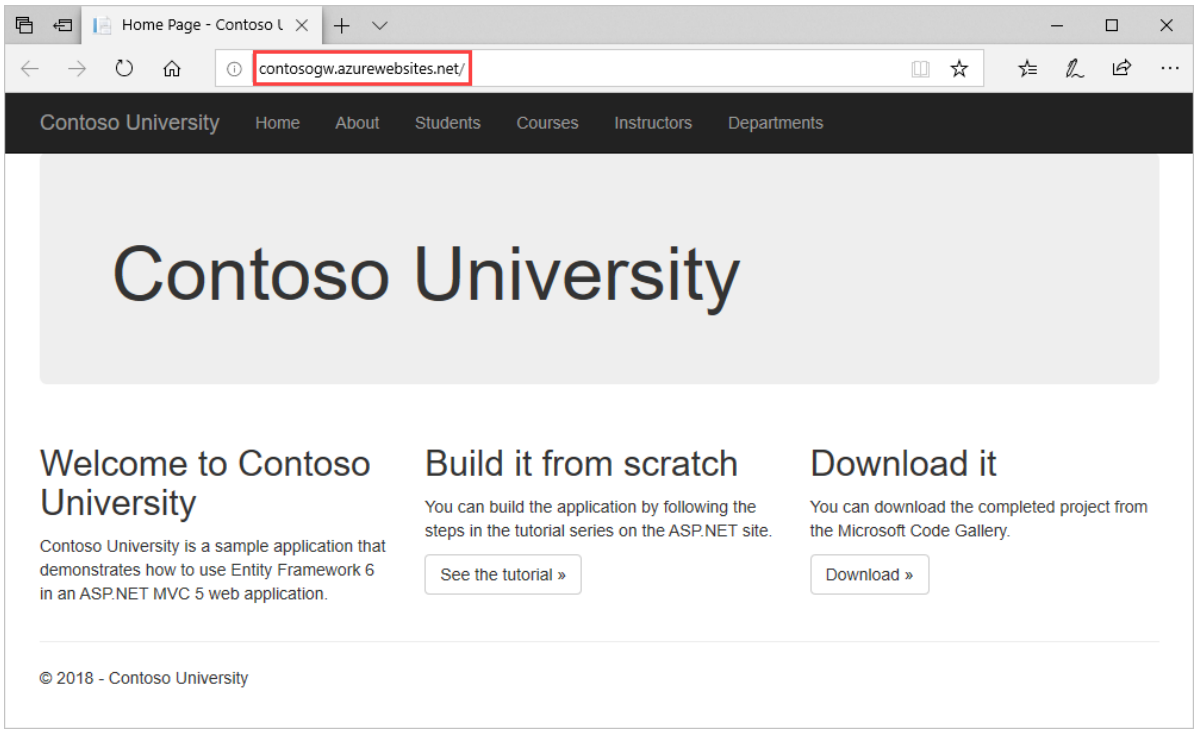


3. If you haven't previously added your Azure subscription in Visual Studio, perform the steps on the screen. These steps enable Visual Studio to connect to your Azure subscription so that the list of **App Services** will include your web site.

4. On the **App Service** page, select the **Subscription** you added the App Service to. Under **View**, select **Resource Group**. Expand the resource group you added the App Service to, and then select the App Service. Choose **OK** to publish the app.

5. The **Output** window shows what deployment actions were taken and reports successful completion of the deployment.

6. Upon successful deployment, the default browser automatically opens to the URL of the deployed web site.



Your app is now running in the cloud.

At this point, the *SchoolContext* database has been created in the Azure SQL database because you selected **Execute Code First Migrations (runs on app start)**. The *Web.config* file in the deployed web site has been changed so that the MigrateDatabaseToLatestVersion initializer runs the first time your code reads or writes data in the database (which happened when you selected the **Students** tab):

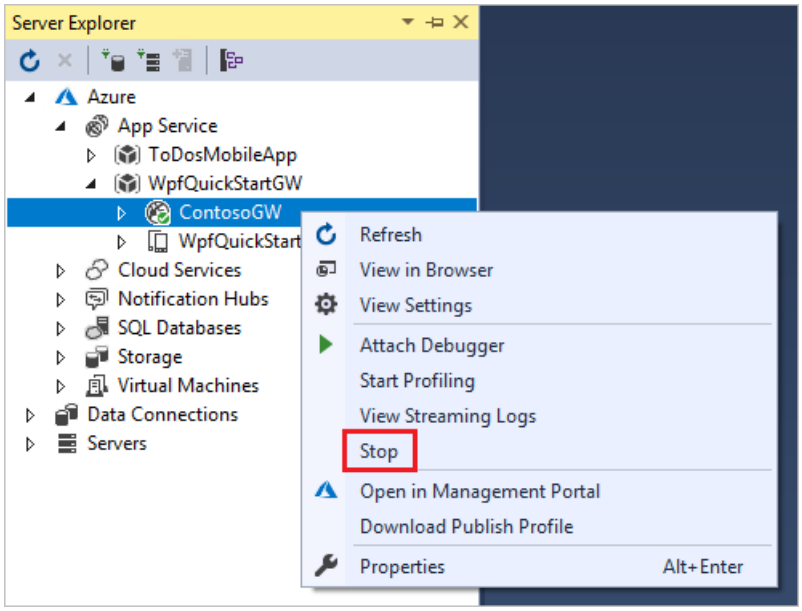
```
</runtime>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFa
      <parameters>
        <parameter value="v11.0" />
      </parameters>
    </defaultConnectionFactory>
    <contexts>
      <context type="ContosoUniversity.Models.SchoolContext, ContosoUniversity">
        <databaseInitializer type="System.Data.Entity.MigrateDatabaseToLatestVersion`2[
                              [ContosoUniversity.Models.SchoolContext, ContosoUniversity],
                              [ContosoUniversity.Migrations.Configuration, ContosoUniversi
                              EntityFramework, PublicKeyToken=b77a5c561934e089">
          <parameters>
            <parameter value="SchoolContext_DatabasePublish"/>
          </parameters>
        </databaseInitializer>
      </context>
    </contexts>
  </entityFramework>
</configuration>
```

The deployment process also created a new connection string *(SchoolContext_DatabasePublish)* for Code First Migrations to use for updating the database schema and seeding the database.

```
<connectionStrings>
    <add name="SchoolContext" connectionString="Data Source=tcp:d015leqjqx.database.winc
    <add name="SchoolContext_DatabasePublish" connectionString="Data Source=tcp:d015leq
</connectionStrings>
```

You can find the deployed version of the Web.config file on your own computer in *ContosoUniversity\obj\Release\Package\PackageTmp\Web.config*. You can access the deployed *Web.config* file itself by using FTP. For instructions, see ASP.NET Web Deployment using Visual Studio: Deploying a Code Update. Follow the instructions that start with "To use an FTP tool, you need three things: the FTP URL, the user name, and the password."

> **NOTE**
> The web app doesn't implement security, so anyone who finds the URL can change the data. For instructions on how to secure the web site, see Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL database to Azure. You can prevent other people from using the site by stopping the service using the Azure Management Portal or **Server Explorer** in Visual Studio.



Advanced migrations scenarios

If you deploy a database by running migrations automatically as shown in this tutorial, and you are deploying to a web site that runs on multiple servers, you could get multiple servers trying to run migrations at the same time. Migrations are atomic, so if two servers try to run the same migration, one will succeed and the other will fail (assuming the operations can't be done twice). In that scenario if you want to avoid those issues, you can call migrations manually and set up your own code so that it only happens once. For more information, see Running and Scripting Migrations from Code on Rowan Miller's blog and Migrate.exe (for executing migrations from the command line).

For information about other migrations scenarios, see Migrations Screencast Series.

Update specific migration

```
update-database -target MigrationName
```

The `update-database -target MigrationName` command runs the targeted migration.

Ignore migration changes to database

```
Add-migration MigrationName -ignoreChanges
```

`ignoreChanges` creates an empty migration with the current model as a snapshot.

Code First initializers

In the deployment section, you saw the MigrateDatabaseToLatestVersion initializer being used. Code First also provides other initializers, including CreateDatabaseIfNotExists (the default), DropCreateDatabaseIfModelChanges (which you used earlier) and DropCreateDatabaseAlways. The `DropCreateAlways` initializer can be useful for setting up conditions for unit tests. You can also write your own initializers, and you can call an initializer explicitly if you

don't want to wait until the application reads from or writes to the database.

For more information about initializers, see Understanding Database Initializers in Entity Framework Code First and chapter 6 of the book Programming Entity Framework: Code First by Julie Lerman and Rowan Miller.

Get the code

Download the Completed Project

Additional resources

Links to other Entity Framework resources can be found in ASP.NET Data Access - Recommended Resources.

Next steps

In this tutorial, you:

- Enabled Code First migrations
- Deployed the app in Azure (optional)

Advance to the next article to learn how to create a more complex data model for an ASP.NET MVC Application.

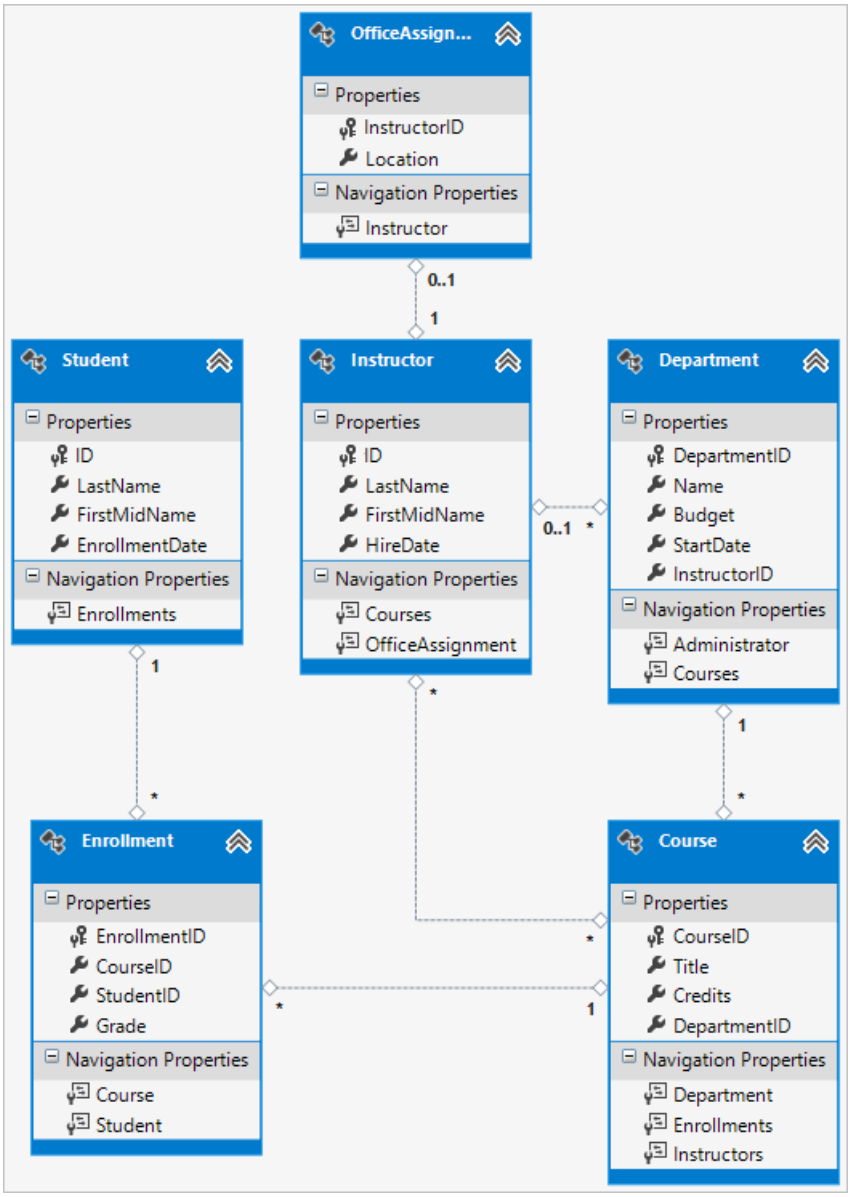
Create a more complex data model

Tutorial: Create a more complex data model for an ASP.NET MVC app

2/20/2020 • 32 minutes to read • Edit Online

In the previous tutorials you worked with a simple data model that was composed of three entities. In this tutorial you add more entities and relationships and you customize the data model by specifying formatting, validation, and database mapping rules. This article shows two ways to customize the data model: by adding attributes to entity classes and by adding code to the database context class.

When you're finished, the entity classes will make up the completed data model that's shown in the following illustration:



In this tutorial, you:

- Customize the data model
- Update Student entity
- Create Instructor entity
- Create OfficeAssignment entity

- Modify the Course entity
- Create the Department entity
- Modify the Enrollment entity
- Add code to database context
- Seed database with test data
- Add a migration
- Update the database

Prerequisites

- Code First migrations and deployment

Customize the data model

In this section you'll see how to customize the data model by using attributes that specify formatting, validation, and database mapping rules. Then in several of the following sections you'll create the complete `School` data model by adding attributes to the classes you already created and creating new classes for the remaining entity types in the model.

The DataType Attribute

For student enrollment dates, all of the web pages currently display the time along with the date, although all you care about for this field is the date. By using data annotation attributes, you can make one code change that will fix the display format in every view that shows the data. To see an example of how to do that, you'll add an attribute to the `EnrollmentDate` property in the `Student` class.

In *Models\Student.cs*, add a `using` statement for the `System.ComponentModel.DataAnnotations` namespace and add `DataType` and `DisplayFormat` attributes to the `EnrollmentDate` property, as shown in the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The DataType attribute is used to specify a data type that is more specific than the database intrinsic type. In this case we only want to keep track of the date, not the date and time. The DataType Enumeration provides for many data types, such as *Date, Time, PhoneNumber, Currency, EmailAddress* and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for DataType.EmailAddress, and a date selector can be provided for DataType.Date in browsers that support HTML5. The DataType attributes emits HTML 5 data- (pronounced *data dash*) attributes that HTML 5 browsers can understand. The DataType attributes do not provide any validation.

`DataType.Date` does not specify the format of the date that is displayed. By default, the data field is displayed

according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

The `ApplyFormatInEditMode` setting specifies that the specified formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you might not want the currency symbol in the text box for editing.)

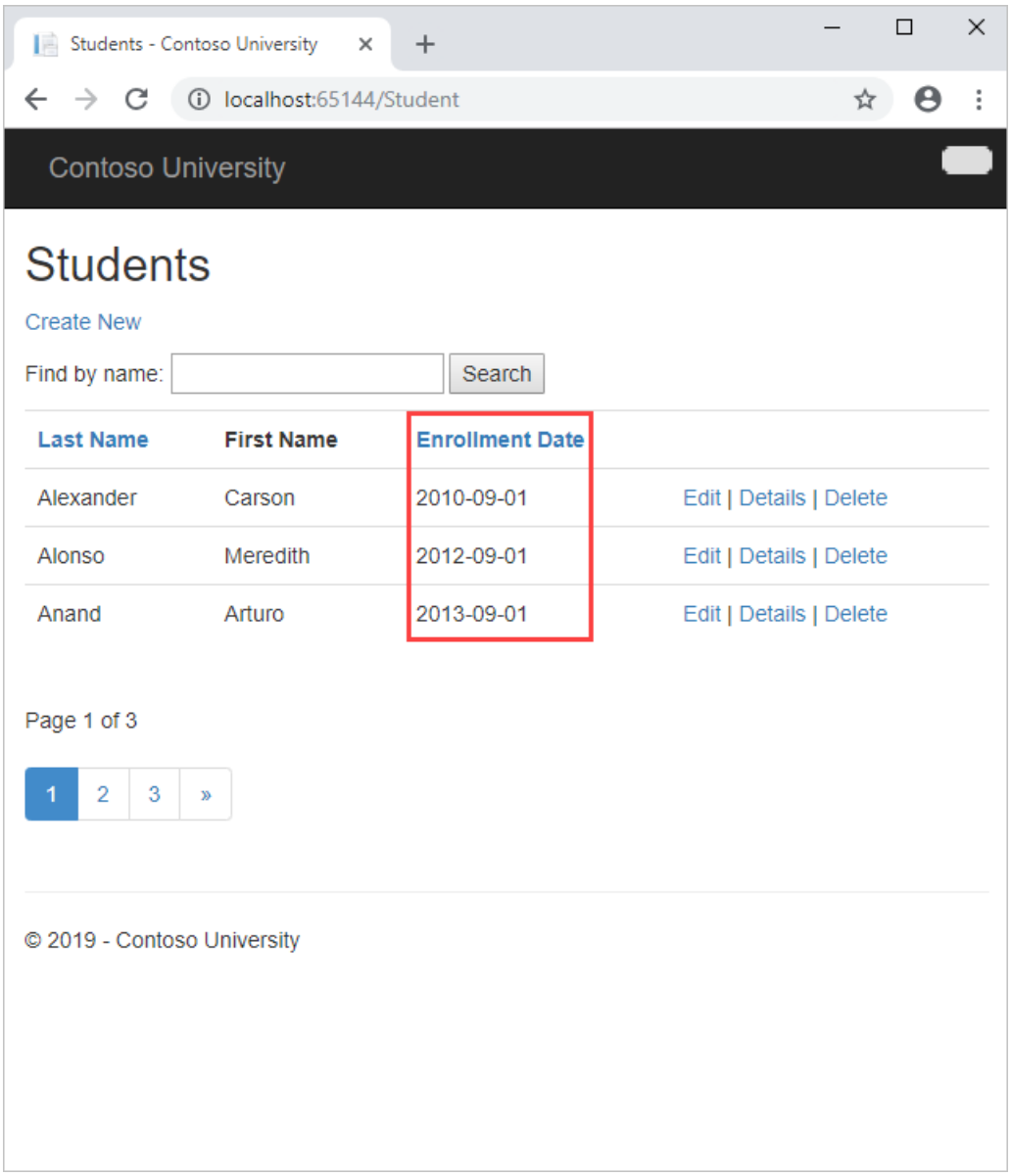
You can use the DisplayFormat attribute by itself, but it's generally a good idea to use the DataType attribute also. The `DataType` attribute conveys the *semantics* of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, some client-side input validation, etc.).
- By default, the browser will render data using the correct format based on your locale.
- The DataType attribute can enable MVC to choose the right field template to render the data (the DisplayFormat uses the string template). For more information, see Brad Wilson's ASP.NET MVC 2 Templates. (Though written for MVC 2, this article still applies to the current version of ASP.NET MVC.)

If you use the `DataType` attribute with a date field, you have to specify the `DisplayFormat` attribute also in order to ensure that the field renders correctly in Chrome browsers. For more information, see this StackOverflow thread.

For more information about how to handle other date formats in MVC, go to MVC 5 Introduction: Examining the Edit Methods and Edit View and search in the page for "internationalization".

Run the Student Index page again and notice that times are no longer displayed for the enrollment dates. The same will be true for any view that uses the `Student` model.



The StringLengthAttribute

You can also specify data validation rules and validation error messages using attributes. The StringLength attribute sets the maximum length in the database and provides client side and server side validation for ASP.NET MVC. You can also specify the minimum string length in this attribute, but the minimum value has no impact on the database schema.

Suppose you want to ensure that users don't enter more than 50 characters for a name. To add this limitation, add StringLength attributes to the `LastName` and `FirstMidName` properties, as shown in the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The StringLength attribute won't prevent a user from entering white space for a name. You can use the RegularExpression attribute to apply restrictions to the input. For example the following code requires the first character to be upper case and the remaining characters to be alphabetical:

```
[RegularExpression(@"^[A-Z]+[a-zA-Z""'\s-]*$")]
```

The MaxLength attribute provides similar functionality to the StringLength attribute but doesn't provide client side validation.

Run the application and click the **Students** tab. You get the following error:

The model backing the 'SchoolContext' context has changed since the database was created. Consider using Code First Migrations to update the database (https://go.microsoft.com/fwlink/?LinkId=238269).

The database model has changed in a way that requires a change in the database schema, and Entity Framework detected that. You'll use migrations to update the schema without losing any data that you added to the database by using the UI. If you changed data that was created by the `Seed` method, that will be changed back to its original state because of the AddOrUpdate method that you're using in the `Seed` method. (AddOrUpdate is equivalent to an "upsert" operation from database terminology.)

In the Package Manager Console (PMC), enter the following commands:

```
add-migration MaxLengthOnNames
update-database
```

The `add-migration` command creates a file named *<timeStamp>_MaxLengthOnNames.cs*. This file contains code in the `Up` method that will update the database to match the current data model. The `update-database` command ran that code.

The timestamp prepended to the migrations file name is used by Entity Framework to order the migrations. You can create multiple migrations before running the `update-database` command, and then all of the migrations are applied in the order in which they were created.

Run the **Create** page, and enter either name longer than 50 characters. When you click **Create**, client side validation shows an error message: *The field LastName must be a string with a maximum length of 50.*

The Column Attribute

You can also use attributes to control how your classes and properties are mapped to the database. Suppose you

had used the name `FirstMidName` for the first-name field because the field might also contain a middle name. But you want the database column to be named `FirstName`, because users who will be writing ad-hoc queries against the database are accustomed to that name. To make this mapping, you can use the `Column` attribute.

The `Column` attribute specifies that when the database is created, the column of the `Student` table that maps to the `FirstMidName` property will be named `FirstName`. In other words, when your code refers to `Student.FirstMidName`, the data will come from or be updated in the `FirstName` column of the `Student` table. If you don't specify column names, they are given the same name as the property name.

In the *Student.cs* file, add a `using` statement for System.ComponentModel.DataAnnotations.Schema and add the column name attribute to the `FirstMidName` property, as shown in the following highlighted code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]

        public DateTime EnrollmentDate { get; set; }

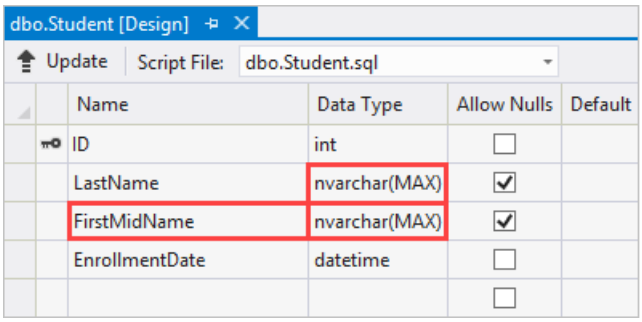
        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

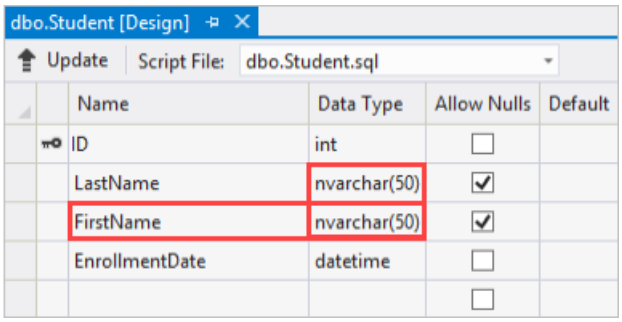
The addition of the Column attribute changes the model backing the SchoolContext, so it won't match the database. Enter the following commands in the PMC to create another migration:

```
add-migration ColumnFirstName
update-database
```

In **Server Explorer**, open the *Student* table designer by double-clicking the *Student* table.

The following image shows the original column name as it was before you applied the first two migrations. In addition to the column name changing from `FirstMidName` to `FirstName`, the two name columns have changed from `MAX` length to 50 characters.





You can also make database mapping changes using the Fluent API, as you'll see later in this tutorial.

> **NOTE**
> If you try to compile before you finish creating all of the entity classes in the following sections, you might get compiler errors.

Update Student entity

In *Models\Student.cs*, replace the code you added earlier with the following code. The changes are highlighted.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The Required Attribute

The Required attribute makes the name properties required fields. The `Required attribute` is not needed for value types such as DateTime, int, double, and float. Value types cannot be assigned a null value, so they are inherently treated as required fields.

The `Required` attribute must be used with `MinimumLength` for the `MinimumLength` to be enforced.

```
[Display(Name = "Last Name")]
[Required]
[StringLength(50, MinimumLength=2)]
public string LastName { get; set; }
```

`MinimumLength` and `Required` allow whitespace to satisfy the validation. Use the `RegularExpression` attribute for full controll over the string.

The Display Attribute

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date" instead of the property name in each instance (which has no space dividing the words).

The FullName Calculated Property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. Therefore it has only a `get` accessor, and no `FullName` column will be generated in the database.

Create Instructor entity

Create *Models\Instructor.cs*, replacing the template code with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public virtual ICollection<Course> Courses { get; set; }
        public virtual OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Notice that several properties are the same in the `Student` and `Instructor` entities. In the Implementing Inheritance tutorial later in this series, you'll refactor this code to eliminate the redundancy.

You can put multiple attributes on one line, so you could also write the instructor class as follows:

```
public class Instructor
{
   public int ID { get; set; }

   [Display(Name = "Last Name"),StringLength(50, MinimumLength=1)]
   public string LastName { get; set; }

   [Column("FirstName"),Display(Name = "First Name"),StringLength(50, MinimumLength=1)]
   public string FirstMidName { get; set; }

   [DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
   public DateTime HireDate { get; set; }

   [Display(Name = "Full Name")]
   public string FullName
   {
      get { return LastName + ", " + FirstMidName; }
   }

   public virtual ICollection<Course> Courses { get; set; }
   public virtual OfficeAssignment OfficeAssignment { get; set; }
}
```

The Courses and OfficeAssignment Navigation Properties

The `Courses` and `OfficeAssignment` properties are navigation properties. As was explained earlier, they are typically defined as virtual so that they can take advantage of an Entity Framework feature called lazy loading. In addition, if a navigation property can hold multiple entities, its type must implement the ICollection<T> Interface. For example IList<T> qualifies but not IEnumerable<T> because `IEnumerable<T>` doesn't implement Add.

An instructor can teach any number of courses, so `Courses` is defined as a collection of `Course` entities.

```
public virtual ICollection<Course> Courses { get; set; }
```

Our business rules state an instructor can only have at most one office, so `OfficeAssignment` is defined as a single `OfficeAssignment` entity (which may be `null` if no office is assigned).

```
public virtual OfficeAssignment OfficeAssignment { get; set; }
```

Create OfficeAssignment entity

Create *Models\OfficeAssignment.cs* with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        [ForeignKey("Instructor")]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public virtual Instructor Instructor { get; set; }
    }
}
```

Build the project, which saves your changes and verifies you haven't made any copy and paste errors the compiler can catch.

The Key Attribute

There's a one-to-zero-or-one relationship between the `Instructor` and the `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to, and therefore its primary key is also its foreign key to the `Instructor` entity. But the Entity Framework can't automatically recognize `InstructorID` as the primary key of this entity because its name doesn't follow the `ID` or *classname* `ID` naming convention. Therefore, the `Key` attribute is used to identify it as the key:

```
[Key]
[ForeignKey("Instructor")]
public int InstructorID { get; set; }
```

You can also use the `Key` attribute if the entity does have its own primary key but you want to name the property something different than `classnameID` or `ID`. By default EF treats the key as non-database-generated because the column is for an identifying relationship.

The ForeignKey Attribute

When there is a one-to-zero-or-one relationship or a one-to-one relationship between two entities (such as between `OfficeAssignment` and `Instructor`), EF can't work out which end of the relationship is the principal and which end is dependent. One-to-one relationships have a reference navigation property in each class to the other class. The ForeignKey Attribute can be applied to the dependent class to establish the relationship. If you omit the ForeignKey Attribute, you get the following error when you try to create the migration:

Unable to determine the principal end of an association between the types 'ContosoUniversity.Models.OfficeAssignment' and 'ContosoUniversity.Models.Instructor'. The principal end of this association must be explicitly configured using either the relationship fluent API or data annotations.

Later in the tutorial you'll see how to configure this relationship with the fluent API.

The Instructor Navigation Property

The `Instructor` entity has a nullable `OfficeAssignment` navigation property (because an instructor might not have an office assignment), and the `OfficeAssignment` entity has a non-nullable `Instructor` navigation property (because an office assignment can't exist without an instructor -- `InstructorID` is non-nullable). When an `Instructor` entity has a related `OfficeAssignment` entity, each entity will have a reference to the other one in its navigation property.

You could put a `[Required]` attribute on the Instructor navigation property to specify that there must be a related instructor, but you don't have to do that because the InstructorID foreign key (which is also the key to this table) is non-nullable.

Modify the Course entity

In *Models\Course.cs*, replace the code you added earlier with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
   public class Course
   {
      [DatabaseGenerated(DatabaseGeneratedOption.None)]
      [Display(Name = "Number")]
      public int CourseID { get; set; }

      [StringLength(50, MinimumLength = 3)]
      public string Title { get; set; }

      [Range(0, 5)]
      public int Credits { get; set; }

      public int DepartmentID { get; set; }

      public virtual Department Department { get; set; }
      public virtual ICollection<Enrollment> Enrollments { get; set; }
      public virtual ICollection<Instructor> Instructors { get; set; }
   }
}
```

The course entity has a foreign key property `DepartmentID` which points to the related `Department` entity and it has a `Department` navigation property. The Entity Framework doesn't require you to add a foreign key property to your data model when you have a navigation property for a related entity. EF automatically creates foreign keys in the database wherever they are needed. But having the foreign key in the data model can make updates simpler and more efficient. For example, when you fetch a course entity to edit, the `Department` entity is null if you don't load it, so when you update the course entity, you would have to first fetch the `Department` entity. When the foreign key property `DepartmentID` is included in the data model, you don't need to fetch the `Department` entity before you update.

The DatabaseGenerated Attribute

The DatabaseGenerated attribute with the None parameter on the `CourseID` property specifies that primary key values are provided by the user rather than generated by the database.

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, the Entity Framework assumes that primary key values are generated by the database. That's what you want in most scenarios. However, for `Course` entities, you'll use a user-specified course number such as a 1000 series for one department, a 2000 series for another department, and so on.

Foreign Key and Navigation Properties

The foreign key properties and navigation properties in the `Course` entity reflect the following relationships:

- A course is assigned to one department, so there's a `DepartmentID` foreign key and a `Department` navigation property for the reasons mentioned above.

```
public int DepartmentID { get; set; }
public virtual Department Department { get; set; }
```

- A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

```
public virtual ICollection<Enrollment> Enrollments { get; set; }
```

- A course may be taught by multiple instructors, so the `Instructors` navigation property is a collection:

```
public virtual ICollection<Instructor> Instructors { get; set; }
```

Create the Department entity

Create *Models\Department.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
   public class Department
   {
      public int DepartmentID { get; set; }

      [StringLength(50, MinimumLength=3)]
      public string Name { get; set; }

      [DataType(DataType.Currency)]
      [Column(TypeName = "money")]
      public decimal Budget { get; set; }

      [DataType(DataType.Date)]
      [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
      [Display(Name = "Start Date")]
      public DateTime StartDate { get; set; }

      public int? InstructorID { get; set; }

      public virtual Instructor Administrator { get; set; }
      public virtual ICollection<Course> Courses { get; set; }
   }
}
```

The Column Attribute

Earlier you used the Column attribute to change column name mapping. In the code for the `Department` entity, the `Column` attribute is being used to change SQL data type mapping so that the column will be defined using the SQL Server money type in the database:

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required, because the Entity Framework usually chooses the appropriate SQL Server data type based on the CLR type that you define for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. But in this case you know that the column will be holding currency amounts, and the money data type is more appropriate for that. For more information about CLR data types and how they match to SQL Server data types, see SqlClient for Entity FrameworkTypes.

Foreign Key and Navigation Properties

The foreign key and navigation properties reflect the following relationships:

- A department may or may not have an administrator, and an administrator is always an instructor. Therefore the `InstructorID` property is included as the foreign key to the `Instructor` entity, and a question mark is added after the `int` type designation to mark the property as nullable.The navigation property is named `Administrator` but holds an `Instructor` entity:

  ```
  public int? InstructorID { get; set; }
  public virtual Instructor Administrator { get; set; }
  ```

- A department may have many courses, so there's a `Courses` navigation property:

  ```
  public virtual ICollection<Course> Courses { get; set; }
  ```

  > **NOTE**
  >
  > By convention, the Entity Framework enables cascade delete for non-nullable foreign keys and for many-to-many relationships. This can result in circular cascade delete rules, which will cause an exception when you try to add a migration. For example, if you didn't define the `Department.InstructorID` property as nullable, you'd get the following exception message: "The referential relationship will result in a cyclical reference that's not allowed." If your business rules required `InstructorID` property to be non-nullable, you would have to use the following fluent API statement to disable cascade delete on the relationship:

```
modelBuilder.Entity().HasRequired(d => d.Administrator).WithMany().WillCascadeOnDelete(false);
```

Modify the Enrollment entity

In *Models\Enrollment.cs*, replace the code you added earlier with the following code

```
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public virtual Course Course { get; set; }
        public virtual Student Student { get; set; }
    }
}
```

Foreign Key and Navigation Properties

The foreign key properties and navigation properties reflect the following relationships:

- An enrollment record is for a single course, so there's a `CourseID` foreign key property and a `Course` navigation property:

  ```
  public int CourseID { get; set; }
  public virtual Course Course { get; set; }
  ```

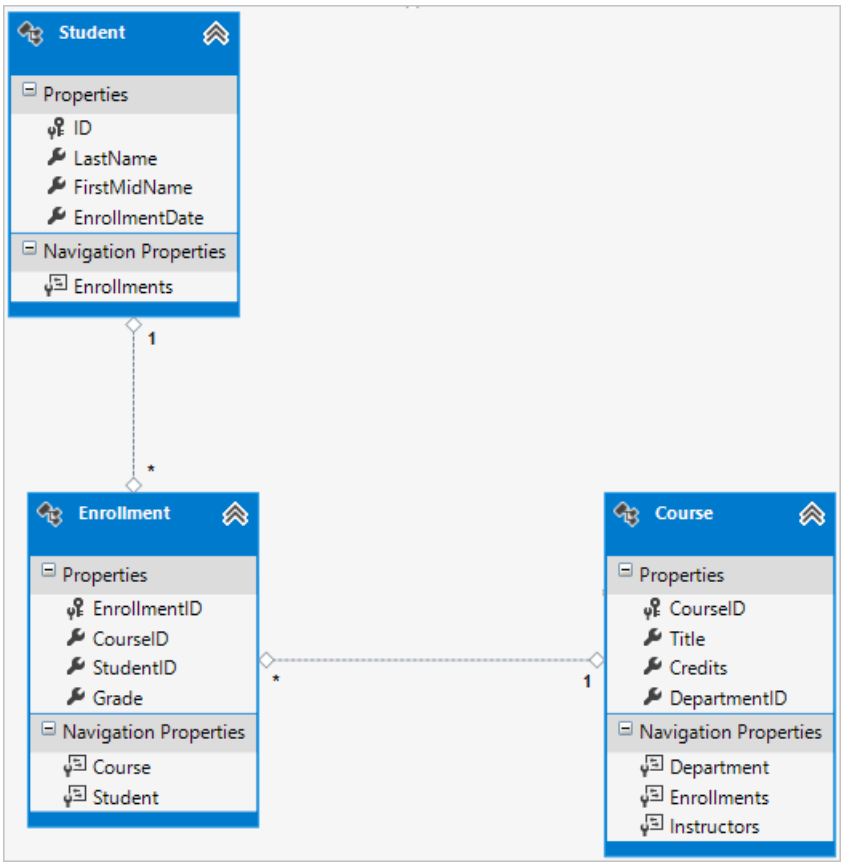
- An enrollment record is for a single student, so there's a `StudentID` foreign key property and a `Student` navigation property:

  ```
  public int StudentID { get; set; }
  public virtual Student Student { get; set; }
  ```

Many-to-Many Relationships

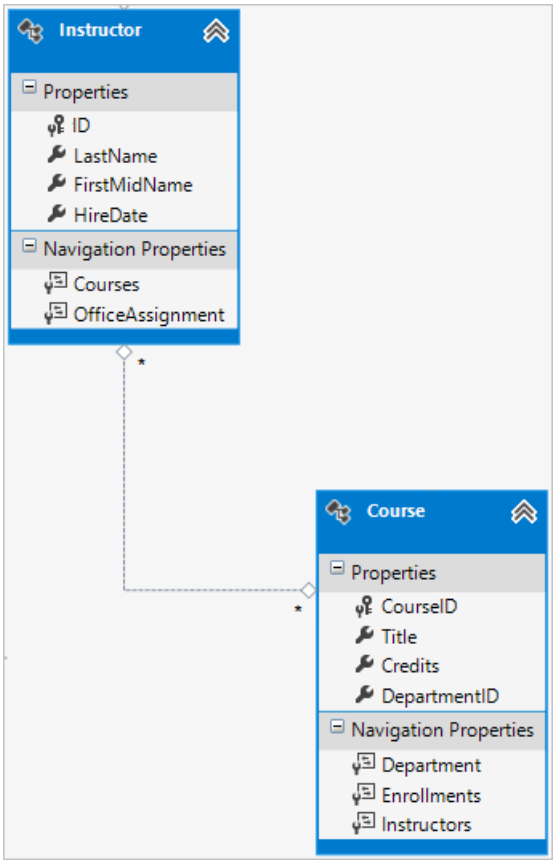
There's a many-to-many relationship between the `Student` and `Course` entities, and the `Enrollment` entity functions as a many-to-many join table *with payload* in the database. This means that the `Enrollment` table contains additional data besides foreign keys for the joined tables (in this case, a primary key and a `Grade` property).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using the Entity Framework Power Tools; creating the diagram isn't part of the tutorial, it's just being used here as an illustration.)

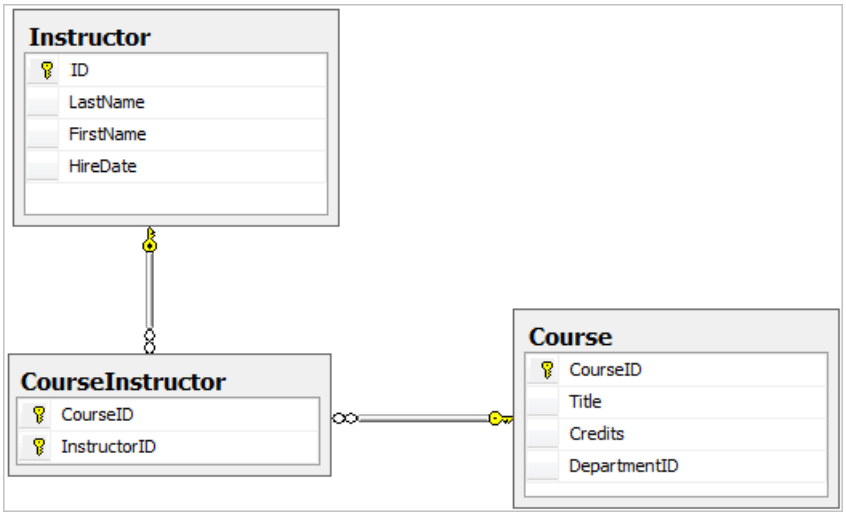


Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two foreign keys `CourseID` and `StudentID`. In that case, it would correspond to a many-to-many join table *without payload* (or a *pure join table*) in the database, and you wouldn't have to create a model class for it at all. The `Instructor` and `Course` entities have that kind of many-to-many relationship, and as you can see, there is no entity class between them:



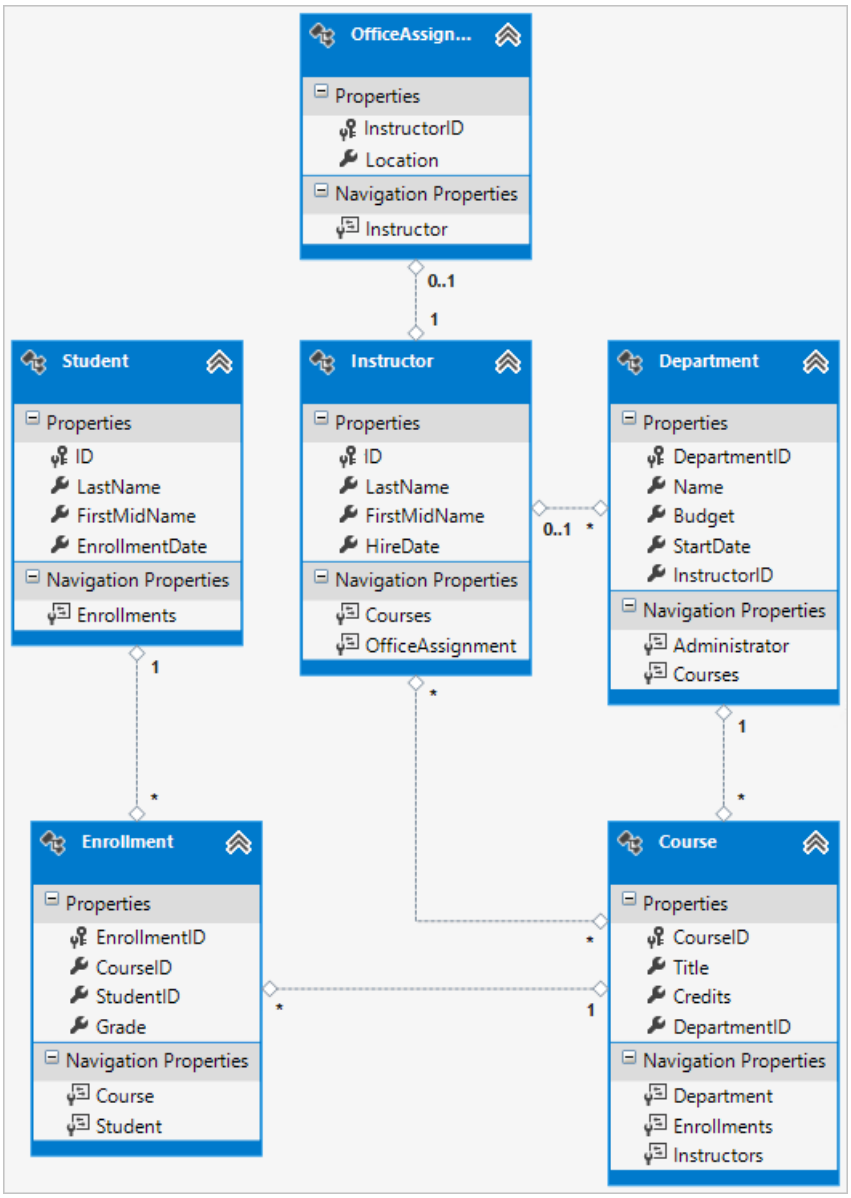
A join table is required in the database, however, as shown in the following database diagram:



The Entity Framework automatically creates the `CourseInstructor` table, and you read and update it indirectly by reading and updating the `Instructor.Courses` and `Course.Instructors` navigation properties.

Entity relationship diagram

The following illustration shows the diagram that the Entity Framework Power Tools create for the completed School model.



Besides the many-to-many relationship lines (* to *) and the one-to-many relationship lines (1 to *), you can see here the one-to-zero-or-one relationship line (1 to 0..1) between the `Instructor` and `OfficeAssignment` entities and the zero-or-one-to-many relationship line (0..1 to *) between the Instructor and Department entities.

Add code to database context

Next you'll add the new entities to the `SchoolContext` class and customize some of the mapping using fluent API calls. The API is "fluent" because it's often used by stringing a series of method calls together into a single statement, as in the following example:

```
modelBuilder.Entity<Course>()
     .HasMany(c => c.Instructors).WithMany(i => i.Courses)
     .Map(t => t.MapLeftKey("CourseID")
         .MapRightKey("InstructorID")
         .ToTable("CourseInstructor"));
```

In this tutorial you'll use the fluent API only for database mapping that you can't do with attributes. However, you can also use the fluent API to specify most of the formatting, validation, and mapping rules that you can do by using attributes. Some attributes such as `MinimumLength` can't be applied with the fluent API. As mentioned previously, `MinimumLength` doesn't change the schema, it only applies a client and server side validation rule

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." You can mix attributes and fluent API if you want, and there are a few customizations that can only be done by using fluent API, but in general the recommended practice is to choose one of these two approaches and use that consistently as much as possible.

To add the new entities to the data model and perform database mapping that you didn't do by using attributes, replace the code in *DAL\SchoolContext.cs* with the following code:

```
using ContosoUniversity.Models;
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace ContosoUniversity.DAL
{
   public class SchoolContext : DbContext
   {
      public DbSet<Course> Courses { get; set; }
      public DbSet<Department> Departments { get; set; }
      public DbSet<Enrollment> Enrollments { get; set; }
      public DbSet<Instructor> Instructors { get; set; }
      public DbSet<Student> Students { get; set; }
      public DbSet<OfficeAssignment> OfficeAssignments { get; set; }

      protected override void OnModelCreating(DbModelBuilder modelBuilder)
      {
         modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

         modelBuilder.Entity<Course>()
             .HasMany(c => c.Instructors).WithMany(i => i.Courses)
             .Map(t => t.MapLeftKey("CourseID")
                 .MapRightKey("InstructorID")
                 .ToTable("CourseInstructor"));
      }
   }
}
```

The new statement in the OnModelCreating method configures the many-to-many join table:

- For the many-to-many relationship between the `Instructor` and `Course` entities, the code specifies the table and column names for the join table. Code First can configure the many-to-many relationship for you without this code, but if you don't call it, you will get default names such as `InstructorInstructorID` for the `InstructorID` column.

```
modelBuilder.Entity<Course>()
    .HasMany(c => c.Instructors).WithMany(i => i.Courses)
    .Map(t => t.MapLeftKey("CourseID")
        .MapRightKey("InstructorID")
        .ToTable("CourseInstructor"));
```

The following code provides an example of how you could have used fluent API instead of attributes to specify the relationship between the `Instructor` and `OfficeAssignment` entities:

```
modelBuilder.Entity<Instructor>()
    .HasOptional(p => p.OfficeAssignment).WithRequired(p => p.Instructor);
```

For information about what "fluent API" statements are doing behind the scenes, see the Fluent API blog post.

Seed database with test data

Replace the code in the *Migrations\Configuration.cs* file with the following code in order to provide seed data for the new entities you've created.

```
namespace ContosoUniversity.Migrations
{
    using ContosoUniversity.Models;
    using ContosoUniversity.DAL;
    using System;
    using System.Collections.Generic;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    internal sealed class Configuration : DbMigrationsConfiguration<SchoolContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
        }

        protected override void Seed(SchoolContext context)
        {
            var students = new List<Student>
            {
                new Student { FirstMidName = "Carson",   LastName = "Alexander",
                    EnrollmentDate = DateTime.Parse("2010-09-01") },
                new Student { FirstMidName = "Meredith", LastName = "Alonso",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Arturo",   LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Gytis",    LastName = "Barzdukas",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Yan",      LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Peggy",    LastName = "Justice",
                    EnrollmentDate = DateTime.Parse("2011-09-01") },
                new Student { FirstMidName = "Laura",    LastName = "Norman",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Nino",     LastName = "Olivetto",
                    EnrollmentDate = DateTime.Parse("2005-09-01") }
```

```
            };

            students.ForEach(s => context.Students.AddOrUpdate(p => p.LastName, s));
            context.SaveChanges();

            var instructors = new List<Instructor>
            {
                new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie",
                    HireDate = DateTime.Parse("1995-03-11") },
                new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",
                    HireDate = DateTime.Parse("2002-07-06") },
                new Instructor { FirstMidName = "Roger",   LastName = "Harui",
                    HireDate = DateTime.Parse("1998-07-01") },
                new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
                    HireDate = DateTime.Parse("2001-01-15") },
                new Instructor { FirstMidName = "Roger",   LastName = "Zheng",
                    HireDate = DateTime.Parse("2004-02-12") }
            };
            instructors.ForEach(s => context.Instructors.AddOrUpdate(p => p.LastName, s));
            context.SaveChanges();

            var departments = new List<Department>
            {
                new Department { Name = "English",     Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Abercrombie").ID },
                new Department { Name = "Mathematics", Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Fakhouri").ID },
                new Department { Name = "Engineering", Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Harui").ID },
                new Department { Name = "Economics",   Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Kapoor").ID }
            };
            departments.ForEach(s => context.Departments.AddOrUpdate(p => p.Name, s));
            context.SaveChanges();

            var courses = new List<Course>
            {
                new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
                  DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID,
                  Instructors = new List<Instructor>()
                },
                new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
                  DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID,
                  Instructors = new List<Instructor>()
                },
                new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
                  DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID,
                  Instructors = new List<Instructor>()
                },
                new Course {CourseID = 1045, Title = "Calculus",       Credits = 4,
                  DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID,
                  Instructors = new List<Instructor>()
                },
                new Course {CourseID = 3141, Title = "Trigonometry",   Credits = 4,
                  DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID,
                  Instructors = new List<Instructor>()
                },
                new Course {CourseID = 2021, Title = "Composition",    Credits = 3,
                  DepartmentID = departments.Single( s => s.Name == "English").DepartmentID,
                  Instructors = new List<Instructor>()
                },
                new Course {CourseID = 2042, Title = "Literature",     Credits = 4,
                  DepartmentID = departments.Single( s => s.Name == "English").DepartmentID,
                  Instructors = new List<Instructor>()
                },
```

```
            };
            courses.ForEach(s => context.Courses.AddOrUpdate(p => p.CourseID, s));
            context.SaveChanges();

            var officeAssignments = new List<OfficeAssignment>
            {
                new OfficeAssignment {
                    InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID,
                    Location = "Smith 17" },
                new OfficeAssignment {
                    InstructorID = instructors.Single( i => i.LastName == "Harui").ID,
                    Location = "Gowan 27" },
                new OfficeAssignment {
                    InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID,
                    Location = "Thompson 304" },
            };
            officeAssignments.ForEach(s => context.OfficeAssignments.AddOrUpdate(p => p.InstructorID, s));
            context.SaveChanges();

            AddOrUpdateInstructor(context, "Chemistry", "Kapoor");
            AddOrUpdateInstructor(context, "Chemistry", "Harui");
            AddOrUpdateInstructor(context, "Microeconomics", "Zheng");
            AddOrUpdateInstructor(context, "Macroeconomics", "Zheng");

            AddOrUpdateInstructor(context, "Calculus", "Fakhouri");
            AddOrUpdateInstructor(context, "Trigonometry", "Harui");
            AddOrUpdateInstructor(context, "Composition", "Abercrombie");
            AddOrUpdateInstructor(context, "Literature", "Abercrombie");

            context.SaveChanges();

            var enrollments = new List<Enrollment>
            {
                new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Alexander").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
                    Grade = Grade.A
                },
                 new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Alexander").ID,
                    CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
                    Grade = Grade.C
                 },
                 new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Alexander").ID,
                    CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
                    Grade = Grade.B
                 },
                 new Enrollment {
                     StudentID = students.Single(s => s.LastName == "Alonso").ID,
                    CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
                    Grade = Grade.B
                 },
                 new Enrollment {
                     StudentID = students.Single(s => s.LastName == "Alonso").ID,
                    CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
                    Grade = Grade.B
                 },
                 new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Alonso").ID,
                    CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
                    Grade = Grade.B
                 },
                 new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Anand").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
                 },
                 new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Anand").ID,
```

```
                    CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
                    Grade = Grade.B
                },
                new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
                    Grade = Grade.B
                },
                new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Li").ID,
                    CourseID = courses.Single(c => c.Title == "Composition").CourseID,
                    Grade = Grade.B
                },
                new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Justice").ID,
                    CourseID = courses.Single(c => c.Title == "Literature").CourseID,
                    Grade = Grade.B
                }
            };

            foreach (Enrollment e in enrollments)
            {
                var enrollmentInDataBase = context.Enrollments.Where(
                    s =>
                         s.Student.ID == e.StudentID &&
                         s.Course.CourseID == e.CourseID).SingleOrDefault();
                if (enrollmentInDataBase == null)
                {
                    context.Enrollments.Add(e);
                }
            }
            context.SaveChanges();
        }

        void AddOrUpdateInstructor(SchoolContext context, string courseTitle, string instructorName)
        {
            var crs = context.Courses.SingleOrDefault(c => c.Title == courseTitle);
            var inst = crs.Instructors.SingleOrDefault(i => i.LastName == instructorName);
            if (inst == null)
                crs.Instructors.Add(context.Instructors.Single(i => i.LastName == instructorName));
        }
    }
}
```

As you saw in the first tutorial, most of this code simply updates or creates new entity objects and loads sample data into properties as required for testing. However, notice how the `Course` entity, which has a many-to-many relationship with the `Instructor` entity, is handled:

```
var courses = new List<Course>
{
    new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
      DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID,
      Instructors = new List<Instructor>()
    },
    ...
};
courses.ForEach(s => context.Courses.AddOrUpdate(p => p.CourseID, s));
context.SaveChanges();
```

When you create a `Course` object, you initialize the `Instructors` navigation property as an empty collection using the code `Instructors = new List<Instructor>()`. This makes it possible to add `Instructor` entities that are related to this `Course` by using the `Instructors.Add` method. If you didn't create an empty list, you wouldn't be able to add these relationships, because the `Instructors` property would be null and wouldn't have an `Add` method. You

could also add the list initialization to the constructor.

Add a migration

From the PMC, enter the `add-migration` command (don't do the `update-database` command yet):

`add-Migration ComplexDataModel`

If you tried to run the `update-database` command at this point (don't do it yet), you would get the following error:

The ALTER TABLE statement conflicted with the FOREIGN KEY constraint "FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.

Sometimes when you execute migrations with existing data, you need to insert stub data into the database to satisfy foreign key constraints, and that's what you have to do now. The generated code in the ComplexDataModel `Up` method adds a non-nullable `DepartmentID` foreign key to the `Course` table. Because there are already rows in the `Course` table when the code runs, the `AddColumn` operation will fail because SQL Server doesn't know what value to put in the column that can't be null. Therefore have to change the code to give the new column a default value, and create a stub department named "Temp" to act as the default department. As a result, existing `Course` rows will all be related to the "Temp" department after the `Up` method runs. You can relate them to the correct departments in the `Seed` method.

Edit the *<timestamp>_ComplexDataModel.cs* file, comment out the line of code that adds the DepartmentID column to the Course table, and add the following highlighted code (the commented line is also highlighted):

```
CreateTable(
        "dbo.CourseInstructor",
        c => new
            {
                CourseID = c.Int(nullable: false),
                InstructorID = c.Int(nullable: false),
            })
        .PrimaryKey(t => new { t.CourseID, t.InstructorID })
        .ForeignKey("dbo.Course", t => t.CourseID, cascadeDelete: true)
        .ForeignKey("dbo.Instructor", t => t.InstructorID, cascadeDelete: true)
        .Index(t => t.CourseID)
        .Index(t => t.InstructorID);

    // Create  a department for course to point to.
    Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00, GETDATE())");
    //  default value for FK points to department created above.
    AddColumn("dbo.Course", "DepartmentID", c => c.Int(nullable: false, defaultValue: 1));
    //AddColumn("dbo.Course", "DepartmentID", c => c.Int(nullable: false));

    AlterColumn("dbo.Course", "Title", c => c.String(maxLength: 50));
```

When the `Seed` method runs, it will insert rows in the `Department` table and it will relate existing `Course` rows to those new `Department` rows. If you haven't added any courses in the UI, you would then no longer need the "Temp" department or the default value on the `Course.DepartmentID` column. To allow for the possibility that someone might have added courses by using the application, you'd also want to update the `Seed` method code to ensure that all `Course` rows (not just the ones inserted by earlier runs of the `Seed` method) have valid `DepartmentID` values before you remove the default value from the column and delete the "Temp" department.

Update the database

After you have finished editing the *<timestamp>_ComplexDataModel.cs* file, enter the `update-database` command in the PMC to execute the migration.

```
update-database
```

NOTE

It's possible to get other errors when migrating data and making schema changes. If you get migration errors you can't resolve, you can either change the database name in the connection string or delete the database. The simplest approach is to rename the database in *Web.config* file. The following example shows the name changed to CU_Test:

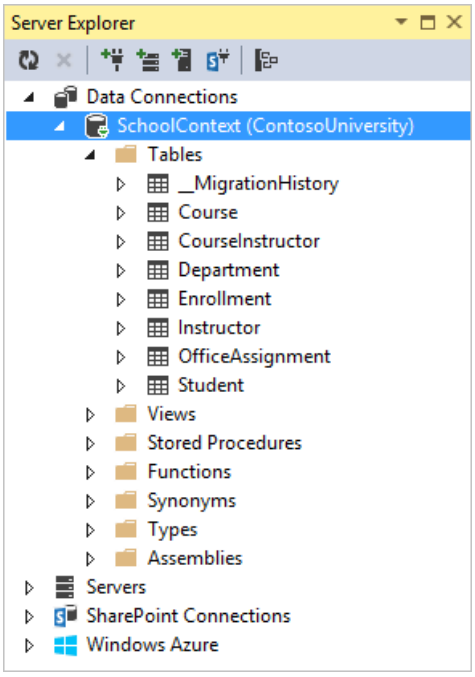
```
<add name="SchoolContext" connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=CU_Test;Integrated Security=SSPI;"
      providerName="System.Data.SqlClient" />
```

With a new database, there is no data to migrate, and the `update-database` command is much more likely to complete without errors. For instructions on how to delete the database, see How to Drop a Database from Visual Studio 2012.

If that fails, another thing you can try is re-initialize the database by entering the following command in the PMC:

```
update-database -TargetMigration:0
```

Open the database in **Server Explorer** as you did earlier, and expand the **Tables** node to see that all of the tables have been created. (If you still have **Server Explorer** open from the earlier time, click the **Refresh** button.)



You didn't create a model class for the `CourseInstructor` table. As explained earlier, this is a join table for the many-to-many relationship between the `Instructor` and `Course` entities.

Right-click the `CourseInstructor` table and select **Show Table Data** to verify that it has data in it as a result of the `Instructor` entities you added to the `Course.Instructors` navigation property.

dbo.CourseInstructor [Data]

Max Rows: 1000

CourseID	InstructorID
2021	1
2042	1
1045	2
1050	3
3141	3
1050	4
4022	5
4041	5
NULL	NULL

Get the code

Download Completed Project

Additional resources

Links to other Entity Framework resources can be found in the ASP.NET Data Access - Recommended Resources.

Next steps

In this tutorial, you:

- Customized the data model
- Updated Student entity
- Created Instructor entity
- Created OfficeAssignment entity
- Modified the Course entity
- Created the Department entity
- Modified the Enrollment entity
- Added code to database context
- Seeded database with test data
- Added a migration
- Updated the database

Advance to the next article to learn how to read and display related data that the Entity Framework loads into navigation properties.

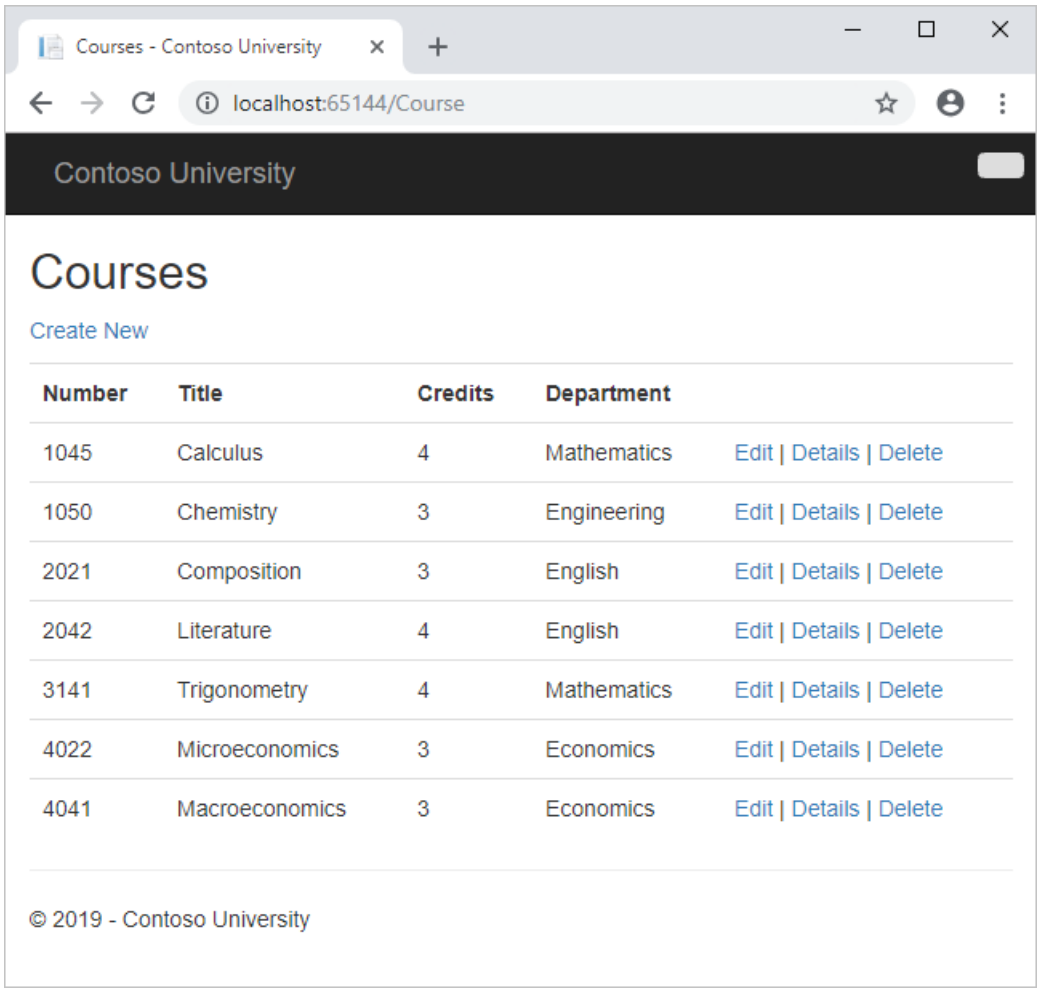
Read related data

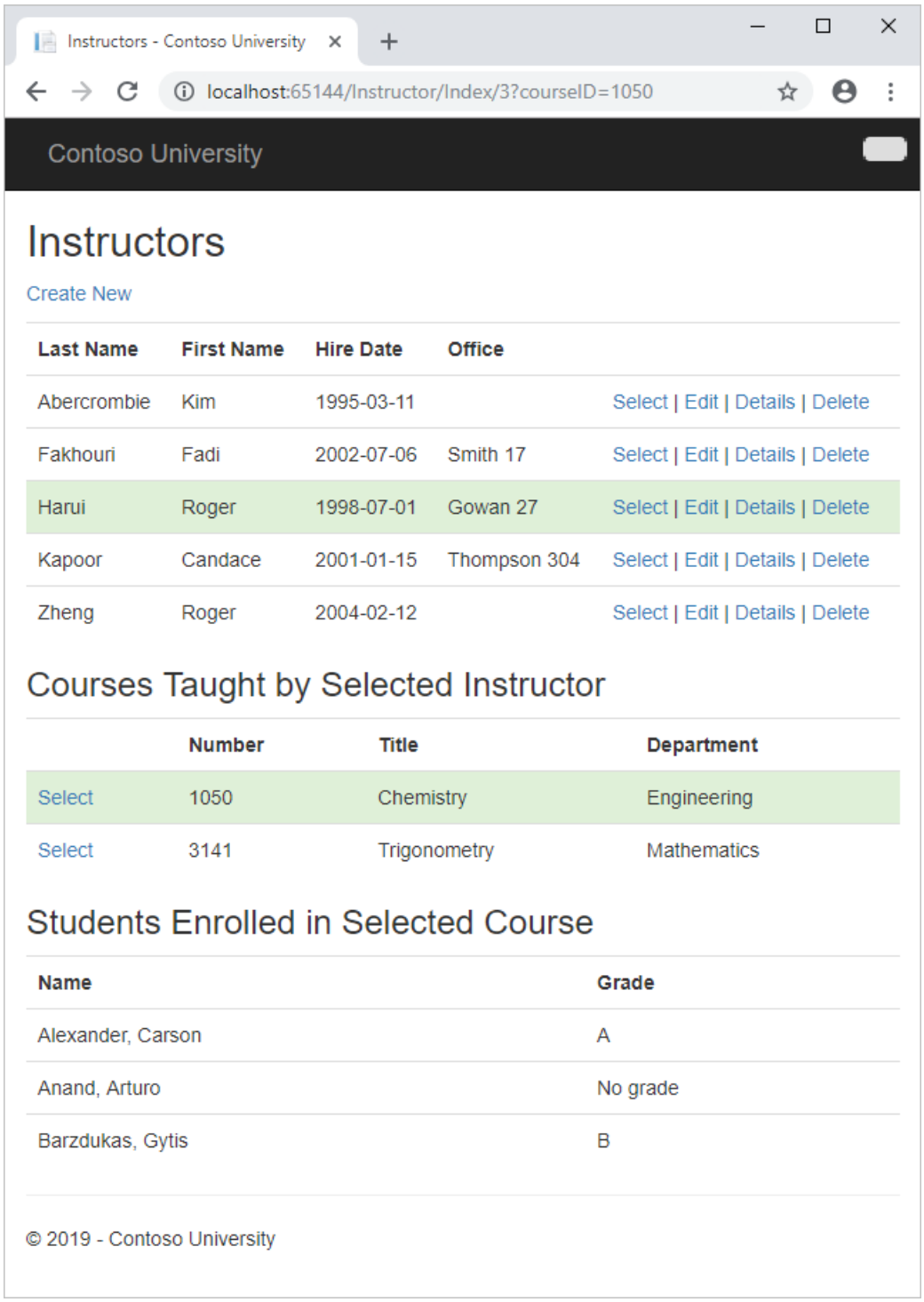
Tutorial: Read related data with EF in an ASP.NET MVC app

2/20/2020 • 16 minutes to read • Edit Online

In the previous tutorial you completed the School data model. In this tutorial you'll read and display related data — that is, data that the Entity Framework loads into navigation properties.

The following illustrations show the pages that you'll work with.





Download Completed Project

> The Contoso University sample web application demonstrates how to create ASP.NET MVC 5 applications using the Entity Framework 6 Code First and Visual Studio. For information about the tutorial series, see the first tutorial in the series.

In this tutorial, you:

- Learn how to load related data
- Create a Courses page
- Create an Instructors page

Prerequisites

- Create a more complex data model

Learn how to load related data

There are several ways that the Entity Framework can load related data into the navigation properties of an entity:

- *Lazy loading*. When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. This results in multiple queries sent to the database — one for the entity itself and one each time that related data for the entity must be retrieved. The `DbContext` class enables lazy loading by default.

```
departments = context.Departments
foreach (Department d in departments)
{
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- *Eager loading*. When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading by using the `Include` method.

```
departments = context.Departments.Include(x => x.Courses)
foreach (Department d in departments)
{
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows and related Course rows

- *Explicit loading*. This is similar to lazy loading, except that you explicitly retrieve the related data in code; it doesn't happen automatically when you access a navigation property. You load related data manually by getting the object state manager entry for an entity and calling the Collection.Load method for collections or the Reference.Load method for properties that hold a single entity. (In the following example, if you wanted to load the Administrator navigation property, you'd replace `Collection(x => x.Courses)` with `Reference(x => x.Administrator)`.) Typically you'd use explicit loading only when you've turned lazy loading off.

```
departments = context.Departments.ToList();
foreach (Department d in departments)
{
    context.Entry(d).Collection(x => x.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

Because they don't immediately retrieve the property values, lazy loading and explicit loading are also both known as *deferred loading*.

Performance considerations

If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, in the above examples, suppose that each department has ten related courses. The eager loading example would result in just a single (join) query and a single round trip to the database. The lazy loading and explicit loading examples would both result in eleven queries and eleven round trips to the database. The extra round trips to the database are especially detrimental to performance when latency is high.

On the other hand, in some scenarios lazy loading is more efficient. Eager loading might cause a very complex join to be generated, which SQL Server can't process efficiently. Or if you need to access an entity's navigation properties only for a subset of a set of the entities you're processing, lazy loading might perform better because

eager loading would retrieve more data than you need. If performance is critical, it's best to test performance both ways in order to make the best choice.

Lazy loading can mask code that causes performance problems. For example, code that doesn't specify eager or explicit loading but processes a high volume of entities and uses several navigation properties in each iteration might be very inefficient (because of many round trips to the database). An application that performs well in development using an on premise SQL server might have performance problems when moved to Azure SQL Database due to the increased latency and lazy loading. Profiling the database queries with a realistic test load will help you determine if lazy loading is appropriate. For more information see Demystifying Entity Framework Strategies: Loading Related Data and Using the Entity Framework to Reduce Network Latency to SQL Azure.

Disable lazy loading before serialization

If you leave lazy loading enabled during serialization, you can end up querying significantly more data than you intended. Serialization generally works by accessing each property on an instance of a type. Property access triggers lazy loading, and those lazy loaded entities are serialized. The serialization process then accesses each property of the lazy-loaded entities, potentially causing even more lazy loading and serialization. To prevent this run-away chain reaction, turn lazy loading off before you serialize an entity.

Serialization can also be complicated by the proxy classes that the Entity Framework uses, as explained in the Advanced Scenarios tutorial.

One way to avoid serialization problems is to serialize data transfer objects (DTOs) instead of entity objects, as shown in the Using Web API with Entity Framework tutorial.

If you don't use DTOs, you can disable lazy loading and avoid proxy issues by disabling proxy creation.

Here are some other ways to disable lazy loading:

- For specific navigation properties, omit the `virtual` keyword when you declare the property.
- For all navigation properties, set `LazyLoadingEnabled` to `false`, put the following code in the constructor of your context class:

```
this.Configuration.LazyLoadingEnabled = false;
```

Create a Courses page

The `Course` entity includes a navigation property that contains the `Department` entity of the department that the course is assigned to. To display the name of the assigned department in a list of courses, you need to get the `Name` property from the `Department` entity that is in the `Course.Department` navigation property.

Create a controller named `CourseController` (not CoursesController) for the `Course` entity type, using the same options for the **MVC 5 Controller with views, using Entity Framework** scaffolder that you did earlier for the `Student` controller:

SETTING	VALUE
Model class	Select **Course (ContosoUniversity.Models)**.
Data context class	Select **SchoolContext (ContosoUniversity.DAL)**.
Controller name	Enter *CourseController*. Again, not *CoursesController* with an *s*. When you selected **Course (ContosoUniversity.Models)**, the **Controller name** value was automatically populated. You have to change the value.

Leave the other default values and add the controller.

Open *Controllers\CourseController.cs* and look at the `Index` method:

```
public ActionResult Index()
{
    var courses = db.Courses.Include(c => c.Department);
    return View(courses.ToList());
}
```

The automatic scaffolding has specified eager loading for the `Department` navigation property by using the `Include` method.

Open *Views\Course\Index.cshtml* and replace the template code with the following code. The changes are highlighted:

```
@model IEnumerable<ContosoUniversity.Models.Course>

@{
    ViewBag.Title = "Courses";
}

<h2>Courses</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.CourseID)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Credits)
        </th>
        <th>
            Department
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.CourseID)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Credits)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Department.Name)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.CourseID }) |
            @Html.ActionLink("Details", "Details", new { id=item.CourseID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.CourseID })
        </td>
    </tr>
}

</table>
```

You've made the following changes to the scaffolded code:

- Changed the heading from **Index** to **Courses**.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they are meaningless to end users. However, in this case the primary key is meaningful and you want to show it.
- Moved the **Department** column to the right side and changed its heading. The scaffolder correctly chose to display the `Name` property from the `Department` entity, but here in the Course page the column heading should be **Department** rather than **Name**.

Notice that for the Department column, the scaffolded code displays the `Name` property of the `Department` entity

that's loaded into the `Department` navigation property:

```
<td>
    @Html.DisplayFor(modelItem => item.Department.Name)
</td>
```

Run the page (select the **Courses** tab on the Contoso University home page) to see the list with department names.

Create an Instructors page

In this section you'll create a controller and view for the `Instructor` entity in order to display the Instructors page. This page reads and displays related data in the following ways:

- The list of instructors displays related data from the `OfficeAssignment` entity. The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. You'll use eager loading for the `OfficeAssignment` entities. As explained earlier, eager loading is typically more efficient when you need the related data for all retrieved rows of the primary table. In this case, you want to display office assignments for all displayed instructors.
- When the user selects an instructor, related `Course` entities are displayed. The `Instructor` and `Course` entities are in a many-to-many relationship. You'll use eager loading for the `Course` entities and their related `Department` entities. In this case, lazy loading might be more efficient because you need courses only for the selected instructor. However, this example shows how to use eager loading for navigation properties within entities that are themselves in navigation properties.
- When the user selects a course, related data from the `Enrollments` entity set is displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship. You'll add explicit loading for `Enrollment` entities and their related `Student` entities. (Explicit loading isn't necessary because lazy loading is enabled, but this shows how to do explicit loading.)

Create a View Model for the Instructor Index View

The Instructors page shows three different tables. Therefore, you'll create a view model that includes three properties, each holding the data for one of the tables.

In the *ViewModels* folder, create *InstructorIndexData.cs* and replace the existing code with the following code:

```
using System.Collections.Generic;
using ContosoUniversity.Models;

namespace ContosoUniversity.ViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Create the Instructor Controller and Views

Create an `InstructorController` (not InstructorsController) controller with EF read/write action:

SETTING	VALUE
Model class	Select **Instructor (ContosoUniversity.Models)**.

SETTING	VALUE
Data context class	Select **SchoolContext (ContosoUniversity.DAL)**.
Controller name	Enter *InstructorController*. Again, not *InstructorsController* with an *s*. When you selected **Course (ContosoUniversity.Models)**, the **Controller name** value was automatically populated. You have to change the value.

Leave the other default values and add the controller.

Open *Controllers\InstructorController.cs* and add a `using` statement for the `ViewModels` namespace:

```
using ContosoUniversity.ViewModels;
```

The scaffolded code in the `Index` method specifies eager loading only for the `OfficeAssignment` navigation property:

```
public ActionResult Index()
{
    var instructors = db.Instructors.Include(i => i.OfficeAssignment);
    return View(instructors.ToList());
}
```

Replace the `Index` method with the following code to load additional related data and put it in the view model:

```
public ActionResult Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.Courses.Select(c => c.Department))
        .OrderBy(i => i.LastName);

    if (id != null)
    {
        ViewBag.InstructorID = id.Value;
        viewModel.Courses = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single().Courses;
    }

    if (courseID != null)
    {
        ViewBag.CourseID = courseID.Value;
        viewModel.Enrollments = viewModel.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }

    return View(viewModel);
}
```

The method accepts optional route data (`id`) and a query string parameter (`courseID`) that provide the ID values of the selected instructor and selected course, and passes all of the required data to the view. The parameters are provided by the **Select** hyperlinks on the page.

The code begins by creating an instance of the view model and putting in it the list of instructors. The code specifies eager loading for the `Instructor.OfficeAssignment` and the `Instructor.Courses` navigation property.

```
var viewModel = new InstructorIndexData();
viewModel.Instructors = db.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.Courses.Select(c => c.Department))
     .OrderBy(i => i.LastName);
```

The second `Include` method loads Courses, and for each Course that is loaded it does eager loading for the `Course.Department` navigation property.

```
.Include(i => i.Courses.Select(c => c.Department))
```

As mentioned previously, eager loading is not required but is done to improve performance. Since the view always requires the `OfficeAssignment` entity, it's more efficient to fetch that in the same query. `Course` entities are required when an instructor is selected in the web page, so eager loading is better than lazy loading only if the page is displayed more often with a course selected than without.

If an instructor ID was selected, the selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is then loaded with the `Course` entities from that instructor's `Courses` navigation property.

```
if (id != null)
{
    ViewBag.InstructorID = id.Value;
    viewModel.Courses = viewModel.Instructors.Where(i => i.ID == id.Value).Single().Courses;
}
```

The `Where` method returns a collection, but in this case the criteria passed to that method result in only a single `Instructor` entity being returned. The `Single` method converts the collection into a single `Instructor` entity, which gives you access to that entity's `Courses` property.

You use the Single method on a collection when you know the collection will have only one item. The `Single` method throws an exception if the collection passed to it is empty or if there's more than one item. An alternative is SingleOrDefault, which returns a default value (`null` in this case) if the collection is empty. However, in this case that would still result in an exception (from trying to find a `Courses` property on a `null` reference), and the exception message would less clearly indicate the cause of the problem. When you call the `Single` method, you can also pass in the `Where` condition instead of calling the `Where` method separately:

```
.Single(i => i.ID == id.Value)
```

Instead of:

```
.Where(I => i.ID == id.Value).Single()
```

Next, if a course was selected, the selected course is retrieved from the list of courses in the view model. Then the view model's `Enrollments` property is loaded with the `Enrollment` entities from that course's `Enrollments` navigation property.

```
if (courseID != null)
{
    ViewBag.CourseID = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

Modify the Instructor Index View

In *Views\Instructor\Index.cshtml*, replace the template code with the following code. The changes are highlighted:

```
@model ContosoUniversity.ViewModels.InstructorIndexData

@{
    ViewBag.Title = "Instructors";
}

<h2>Instructors</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>Last Name</th>
        <th>First Name</th>
        <th>Hire Date</th>
        <th>Office</th>
        <th></th>
    </tr>

    @foreach (var item in Model.Instructors)
    {
        string selectedRow = "";
        if (item.ID == ViewBag.InstructorID)
        {
            selectedRow = "success";
        }
        <tr class="@selectedRow">
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.HireDate)
            </td>
            <td>
                @if (item.OfficeAssignment != null)
                {
                    @item.OfficeAssignment.Location
                }
            </td>
            <td>
                @Html.ActionLink("Select", "Index", new { id = item.ID }) |
                @Html.ActionLink("Edit", "Edit", new { id = item.ID }) |
                @Html.ActionLink("Details", "Details", new { id = item.ID }) |
                @Html.ActionLink("Delete", "Delete", new { id = item.ID })
            </td>
        </tr>
    }

    </table>
```

You've made the following changes to the existing code:

- Changed the model class to `InstructorIndexData`.
- Changed the page title from **Index** to **Instructors**.
- Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` is not null. (Because this is a one-to-zero-or-one relationship, there might not be a related `OfficeAssignment` entity.)

```
<td>
    @if (item.OfficeAssignment != null)
    {
        @item.OfficeAssignment.Location
    }
</td>
```

- Added code that will dynamically add `class="success"` to the `tr` element of the selected instructor. This sets a background color for the selected row using a Bootstrap class.

```
string selectedRow = "";
if (item.InstructorID == ViewBag.InstructorID)
{
    selectedRow = "success";
}
<tr class="@selectedRow" valign="top">
```

- Added a new `ActionLink` labeled **Select** immediately before the other links in each row, which causes the selected instructor ID to be sent to the `Index` method.

Run the application and select the **Instructors** tab. The page displays the `Location` property of related `OfficeAssignment` entities and an empty table cell when there's no related `OfficeAssignment` entity.

In the *Views\Instructor\Index.cshtml* file, after the closing `table` element (at the end of the file), add the following code. This code displays a list of courses related to an instructor when an instructor is selected.

```
@if (Model.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == ViewBag.CourseID)
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.ActionLink("Select", "Index", new { courseID = item.CourseID })
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}
```

This code reads the `Courses` property of the view model to display a list of courses. It also provides a `Select` hyperlink that sends the ID of the selected course to the `Index` action method.

Run the page and select an instructor. Now you see a grid that displays courses assigned to the selected instructor, and for each course you see the name of the assigned department.

After the code block you just added, add the following code. This displays a list of the students who are enrolled in a course when that course is selected.

```
@if (Model.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

This code reads the `Enrollments` property of the view model in order to display a list of students enrolled in the course.

Run the page and select an instructor. Then select a course to see the list of enrolled students and their grades.

Adding Explicit Loading

Open *InstructorController.cs* and look at how the `Index` method gets the list of enrollments for a selected course:

```
if (courseID != null)
{
    ViewBag.CourseID = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

When you retrieved the list of instructors, you specified eager loading for the `Courses` navigation property and for the `Department` property of each course. Then you put the `Courses` collection in the view model, and now you're accessing the `Enrollments` navigation property from one entity in that collection. Because you didn't specify eager loading for the `Course.Enrollments` navigation property, the data from that property is appearing in the page as a result of lazy loading.

If you disabled lazy loading without changing the code in any other way, the `Enrollments` property would be null regardless of how many enrollments the course actually had. In that case, to load the `Enrollments` property, you'd have to specify either eager loading or explicit loading. You've already seen how to do eager loading. In order to see an example of explicit loading, replace the `Index` method with the following code, which explicitly loads the `Enrollments` property. The code changed are highlighted.

```
public ActionResult Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();

    viewModel.Instructors = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.Courses.Select(c => c.Department))
        .OrderBy(i => i.LastName);

    if (id != null)
    {
        ViewBag.InstructorID = id.Value;
        viewModel.Courses = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single().Courses;
    }

    if (courseID != null)
    {
        ViewBag.CourseID = courseID.Value;
        // Lazy loading
        //viewModel.Enrollments = viewModel.Courses.Where(
        //    x => x.CourseID == courseID).Single().Enrollments;
        // Explicit loading
        var selectedCourse = viewModel.Courses.Where(x => x.CourseID == courseID).Single();
        db.Entry(selectedCourse).Collection(x => x.Enrollments).Load();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            db.Entry(enrollment).Reference(x => x.Student).Load();
        }

        viewModel.Enrollments = selectedCourse.Enrollments;
    }

    return View(viewModel);
}
```

After getting the selected `Course` entity, the new code explicitly loads that course's `Enrollments` navigation property:

```
db.Entry(selectedCourse).Collection(x => x.Enrollments).Load();
```

Then it explicitly loads each `Enrollment` entity's related `Student` entity:

```
db.Entry(enrollment).Reference(x => x.Student).Load();
```

Notice that you use the `Collection` method to load a collection property, but for a property that holds just one entity, you use the `Reference` method.

Run the Instructor Index page now and you'll see no difference in what's displayed on the page, although you've changed how the data is retrieved.

Get the code

Download Completed Project

Additional resources

Links to other Entity Framework resources can be found in the ASP.NET Data Access - Recommended Resources.

Next steps

In this tutorial, you:

- Learned how to load related data
- Created a Courses page
- Created an Instructors page

Advance to the next article to learn how to update related data.

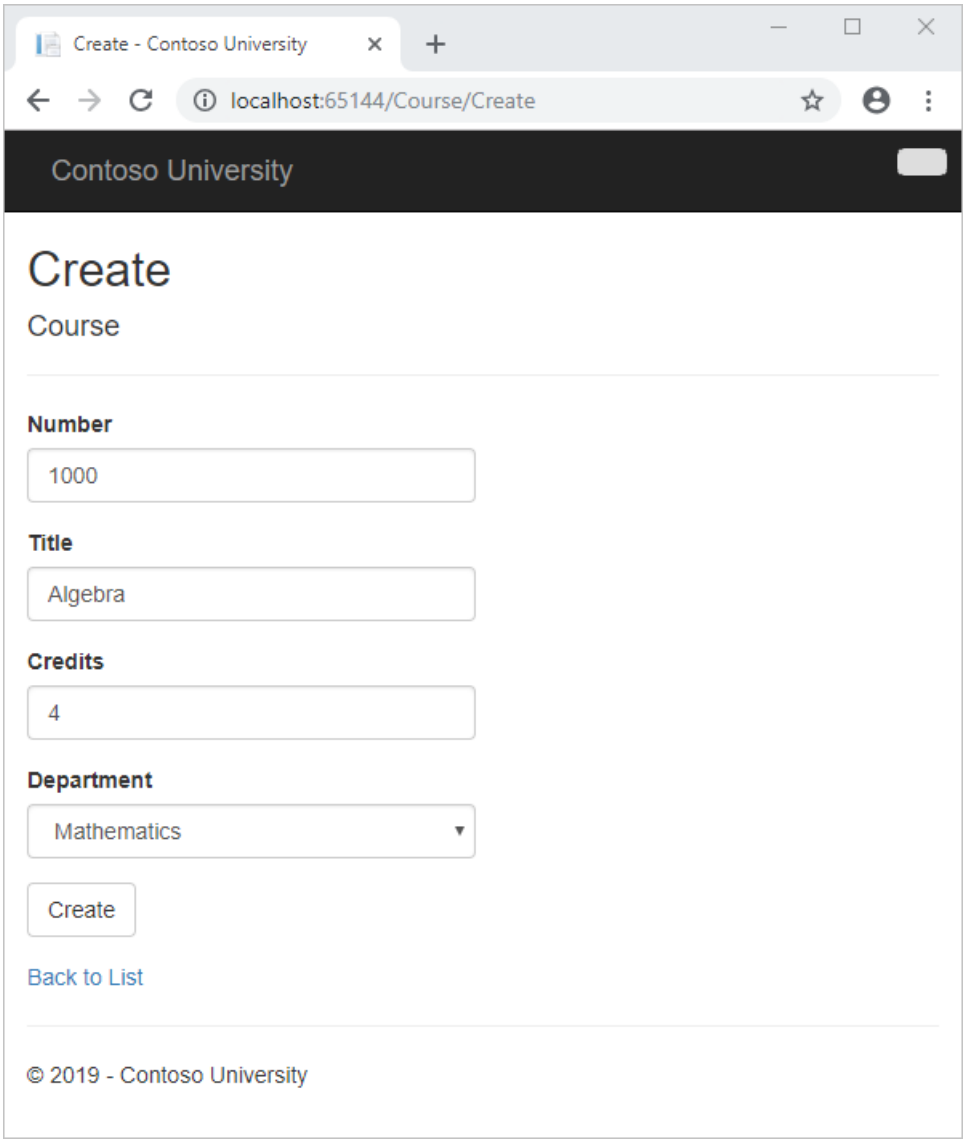
Update related data

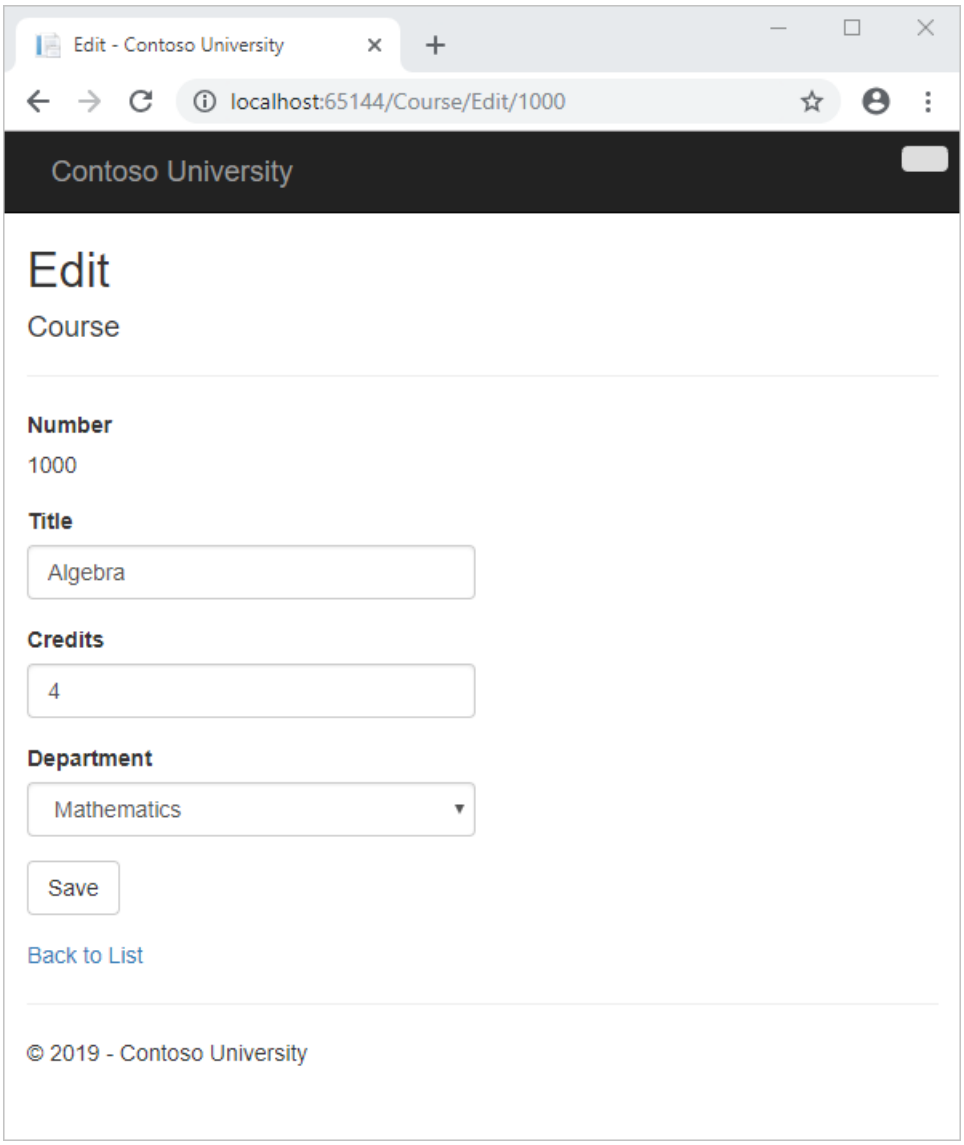
Tutorial: Update related data with EF in an ASP.NET MVC app

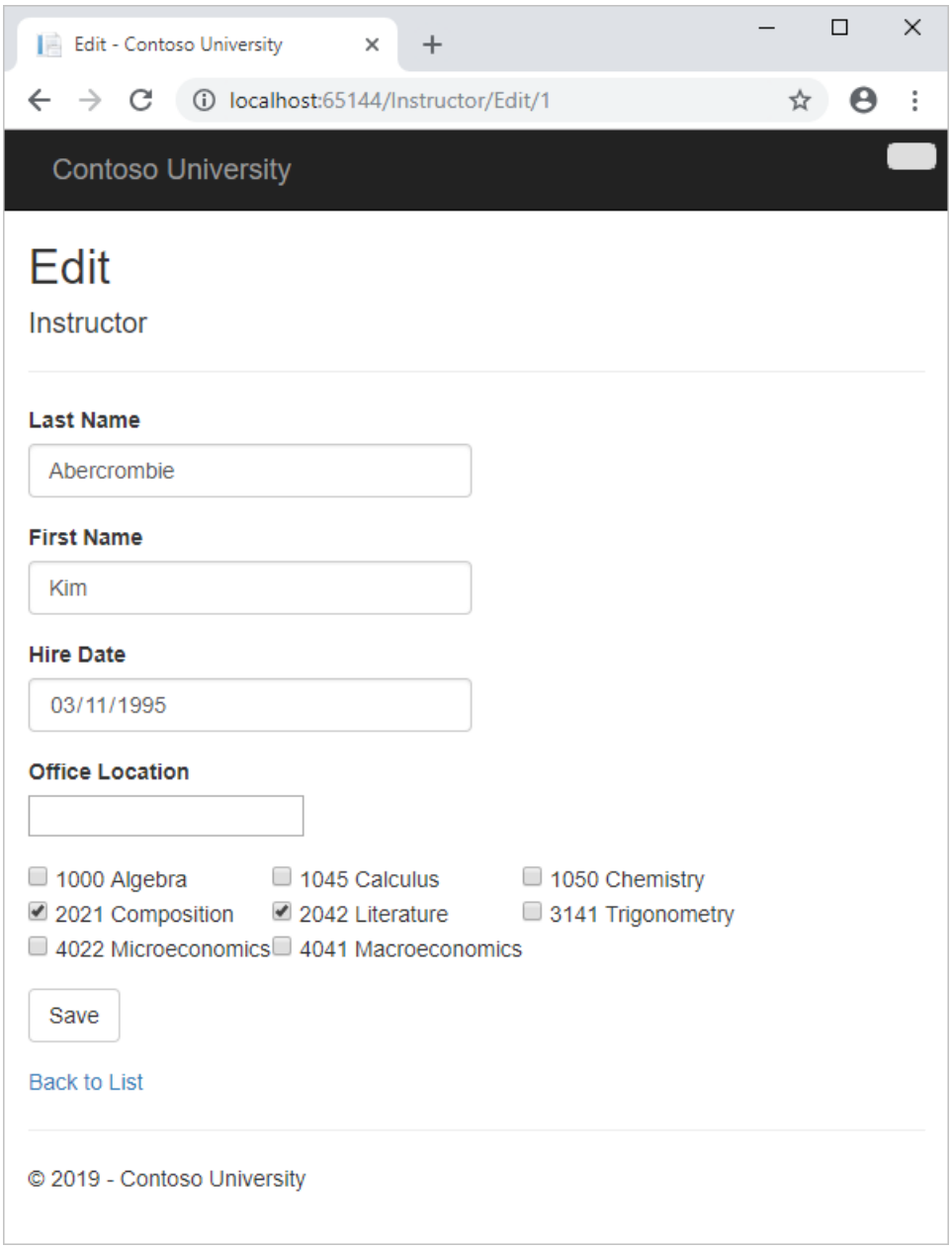
2/20/2020 • 19 minutes to read • Edit Online

In the previous tutorial you displayed related data. In this tutorial you'll update related data. For most relationships, this can be done by updating either foreign key fields or navigation properties. For many-to-many relationships, the Entity Framework doesn't expose the join table directly, so you add and remove entities to and from the appropriate navigation properties.

The following illustrations show some of the pages that you'll work with.




Edit - Contoso University
localhost:65144/Course/Edit/1000
Contoso University
Edit
Course
Number
1000
Title
Algebra
Credits
4
Department
Mathematics
Save
Back to List
© 2019 - Contoso University



In this tutorial, you:

- Customize courses pages
- Add office to instructors page
- Add courses to instructors page
- Update DeleteConfirmed
- Add office location and courses to the Create page

Prerequisites

- Reading Related Data

Customize courses pages

When a new course entity is created, it must have a relationship to an existing department. To facilitate this, the scaffolded code includes controller methods and Create and Edit views that include a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key property, and that's all the Entity Framework needs in order to load the `Department` navigation property with the appropriate `Department` entity. You'll use the scaffolded code, but change it slightly to add error handling and sort the drop-down list.

In *CourseController.cs*, delete the four `Create` and `Edit` methods and replace them with the following code:

```
public ActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "CourseID,Title,Credits,DepartmentID")]Course course)
{
    try
    {
        if (ModelState.IsValid)
        {
            db.Courses.Add(course);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
    }
    catch (RetryLimitExceededException /* dex */)
    {
        //Log the error (uncomment dex variable name and add a line here to write a log.)
        ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists, see your
system administrator.");
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}

public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Course course = db.Courses.Find(id);
    if (course == null)
    {
        return HttpNotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}

[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public ActionResult EditPost(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    var courseToUpdate = db.Courses.Find(id);
    if (TryUpdateModel(courseToUpdate, "",
       new string[] { "Title", "Credits", "DepartmentID" }))
    {
        try
        {
            db.SaveChanges();

            return RedirectToAction("Index");
        }
        catch (RetryLimitExceededException /* dex */)
        {
            //Log the error (uncomment dex variable name and add a line here to write a log.
            ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists, see
```

```
            ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists, see
your system administrator.");
        }
    }
    PopulateDepartmentsDropDownList(courseToUpdate.DepartmentID);
    return View(courseToUpdate);
}

private void PopulateDepartmentsDropDownList(object selectedDepartment = null)
{
    var departmentsQuery = from d in db.Departments
                           orderby d.Name
                           select d;
    ViewBag.DepartmentID = new SelectList(departmentsQuery, "DepartmentID", "Name", selectedDepartment);
}
```

Add the following `using` statement at the beginning of the file:

```
using System.Data.Entity.Infrastructure;
```

The `PopulateDepartmentsDropDownList` method gets a list of all departments sorted by name, creates a `SelectList` collection for a drop-down list, and passes the collection to the view in a `ViewBag` property. The method accepts the optional `selectedDepartment` parameter that allows the calling code to specify the item that will be selected when the drop-down list is rendered. The view will pass the name `DepartmentID` to the DropDownList helper, and the helper then knows to look in the `ViewBag` object for a `SelectList` named `DepartmentID`.

The `HttpGet` `Create` method calls the `PopulateDepartmentsDropDownList` method without setting the selected item, because for a new course the department is not established yet:

```
public ActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

The `HttpGet` `Edit` method sets the selected item, based on the ID of the department that is already assigned to the course being edited:

```
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Course course = db.Courses.Find(id);
    if (course == null)
    {
        return HttpNotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

The `HttpPost` methods for both `Create` and `Edit` also include code that sets the selected item when they redisplay the page after an error:

```
catch (RetryLimitExceededException /* dex */)
{
    //Log the error (uncomment dex variable name and add a line here to write a log.)
    ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists, see your
system administrator.");
}
PopulateDepartmentsDropDownList(course.DepartmentID);
return View(course);
```

This code ensures that when the page is redisplayed to show the error message, whatever department was selected stays selected.

The Course views are already scaffolded with drop-down lists for the department field, but you don't want the DepartmentID caption for this field, so make the following highlighted change to the *Views\Course\Create.cshtml* file to change the caption.

```
@model ContosoUniversity.Models.Course

@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Course</h4>
        <hr />
        @Html.ValidationSummary(true)

        <div class="form-group">
            @Html.LabelFor(model => model.CourseID, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.CourseID)
                @Html.ValidationMessageFor(model => model.CourseID)
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Title, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Title)
                @Html.ValidationMessageFor(model => model.Title)
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Credits, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Credits)
                @Html.ValidationMessageFor(model => model.Credits)
            </div>
        </div>

        <div class="form-group">
            <label class="control-label col-md-2" for="DepartmentID">Department</label>
            <div class="col-md-10">
                @Html.DropDownList("DepartmentID", String.Empty)
                @Html.ValidationMessageFor(model => model.DepartmentID)
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </div>
    </div>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Make the same change in *Views\Course\Edit.cshtml*.

Normally the scaffolder doesn't scaffold a primary key because the key value is generated by the database and can't be changed and isn't a meaningful value to be displayed to users. For Course entities the scaffolder does include an text box for the `CourseID` field because it understands that the `DatabaseGeneratedOption.None` attribute means the user should be able enter the primary key value. But it doesn't understand that because the number is meaningful you want to see it in the other views, so you need to add it manually.

In *Views\Course\Edit.cshtml*, add a course number field before the **Title** field. Because it's the primary key, it's displayed, but it can't be changed.

```
<div class="form-group">
    @Html.LabelFor(model => model.CourseID, new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.DisplayFor(model => model.CourseID)
    </div>
</div>
```

There's already a hidden field (`Html.HiddenFor` helper) for the course number in the Edit view. Adding an *Html.LabelFor* helper doesn't eliminate the need for the hidden field because it doesn't cause the course number to be included in the posted data when the user clicks **Save** on the Edit page.

In *Views\Course\Delete.cshtml* and *Views\Course\Details.cshtml*, change the department name caption from "Name" to "Department" and add a course number field before the **Title** field.

```
<dt>
    Department
</dt>

<dd>
    @Html.DisplayFor(model => model.Department.Name)
</dd>

<dt>
    @Html.DisplayNameFor(model => model.CourseID)
</dt>

<dd>
    @Html.DisplayFor(model => model.CourseID)
</dd>
```

Run the **Create** page (display the Course Index page and click **Create New**) and enter data for a new course:

VALUE	SETTING
Number	Enter *1000*.
Title	Enter *Algebra*.
Credits	Enter *4*.
Department	Select **Mathematics**.

Click **Create**. The Course Index page is displayed with the new course added to the list. The department name in the Index page list comes from the navigation property, showing that the relationship was established correctly.

Run the **Edit** page (display the Course Index page and click **Edit** on a course).

Change data on the page and click **Save**. The Course Index page is displayed with the updated course data.

Add office to instructors page

When you edit an instructor record, you want to be able to update the instructor's office assignment. The `Instructor` entity has a one-to-zero-or-one relationship with the `OfficeAssignment` entity, which means you must handle the following situations:

- If the user clears the office assignment and it originally had a value, you must remove and delete the `OfficeAssignment` entity.
- If the user enters an office assignment value and it originally was empty, you must create a new `OfficeAssignment` entity.
- If the user changes the value of an office assignment, you must change the value in an existing `OfficeAssignment` entity.

Open *InstructorController.cs* and look at the `HttpGet` `Edit` method:

```
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Instructor instructor = db.Instructors.Find(id);
    if (instructor == null)
    {
        return HttpNotFound();
    }
    ViewBag.ID = new SelectList(db.OfficeAssignments, "InstructorID", "Location", instructor.ID);
    return View(instructor);
}
```

The scaffolded code here isn't what you want. It's setting up data for a drop-down list, but you what you need is a text box. Replace this method with the following code:

```
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Instructor instructor = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Where(i => i.ID == id)
        .Single();
    if (instructor == null)
    {
        return HttpNotFound();
    }
    return View(instructor);
}
```

This code drops the `ViewBag` statement and adds eager loading for the associated `OfficeAssignment` entity. You can't perform eager loading with the `Find` method, so the `Where` and `Single` methods are used instead to select the instructor.

Replace the `HttpPost` `Edit` method with the following code. which handles office assignment updates:

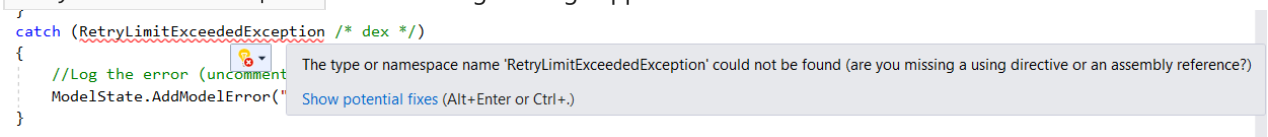
```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public ActionResult EditPost(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    var instructorToUpdate = db.Instructors
       .Include(i => i.OfficeAssignment)
       .Where(i => i.ID == id)
       .Single();

    if (TryUpdateModel(instructorToUpdate, "",
       new string[] { "LastName", "FirstMidName", "HireDate", "OfficeAssignment" }))
    {
       try
       {
          if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment.Location))
          {
             instructorToUpdate.OfficeAssignment = null;
          }

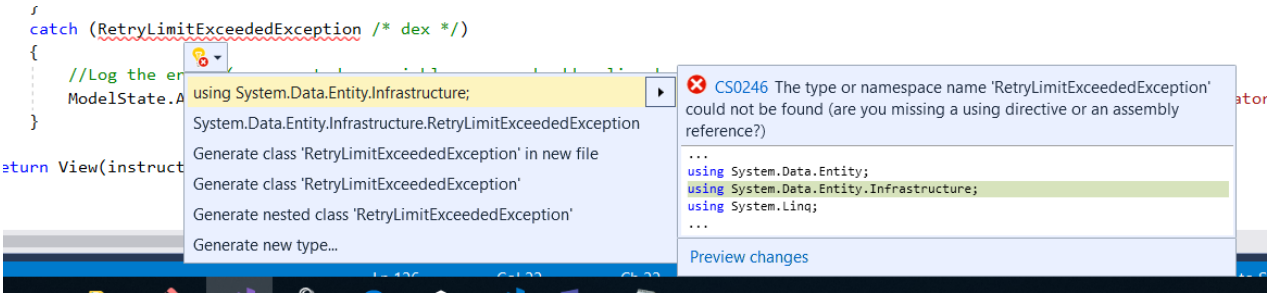
          db.SaveChanges();

          return RedirectToAction("Index");
       }
       catch (RetryLimitExceededException /* dex */)
      {
         //Log the error (uncomment dex variable name and add a line here to write a log.
         ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists, see
your system administrator.");
      }
   }
   return View(instructorToUpdate);
}
```

The reference to `RetryLimitExceededException` requires a `using` statement; to add it - hover your mouse over `RetryLimitExceededException`. The following message appears:



Select **Show potential fixes**, then **using System.Data.Entity.Infrastructure**



The code does the following:

- Changes the method name to `EditPost` because the signature is now the same as the `HttpGet` method (the `ActionName` attribute specifies that the /Edit/ URL is still used).
- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` navigation property. This is the same as what you did in the `HttpGet` `Edit` method.

- Updates the retrieved `Instructor` entity with values from the model binder. The TryUpdateModel overload used enables you to *whitelist* the properties you want to include. This prevents over-posting, as explained in the second tutorial.

```
if (TryUpdateModel(instructorToUpdate, "",
      new string[] { "LastName", "FirstMidName", "HireDate", "OfficeAssignment" }))
```

- If the office location is blank, sets the `Instructor.OfficeAssignment` property to null so that the related row in the `OfficeAssignment` table will be deleted.

```
if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment.Location))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

- Saves the changes to the database.

In *Views\Instructor\Edit.cshtml*, after the `div` elements for the **Hire Date** field, add a new field for editing the office location:

```
<div class="form-group">
    @Html.LabelFor(model => model.OfficeAssignment.Location, new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.OfficeAssignment.Location)
        @Html.ValidationMessageFor(model => model.OfficeAssignment.Location)
    </div>
</div>
```

Run the page (select the **Instructors** tab and then click **Edit** on an instructor). Change the **Office Location** and click **Save**.

Add courses to instructors page

Instructors may teach any number of courses. Now you'll enhance the Instructor Edit page by adding the ability to change course assignments using a group of check boxes.

The relationship between the `Course` and `Instructor` entities is many-to-many, which means you do not have direct access to the foreign key properties which are in the join table. Instead, you add and remove entities to and from the `Instructor.Courses` navigation property.

The UI that enables you to change which courses an instructor is assigned to is a group of check boxes. A check box for every course in the database is displayed, and the ones that the instructor is currently assigned to are selected. The user can select or clear check boxes to change course assignments. If the number of courses were much greater, you would probably want to use a different method of presenting the data in the view, but you'd use the same method of manipulating navigation properties in order to create or delete relationships.

To provide data to the view for the list of check boxes, you'll use a view model class. Create *AssignedCourseData.cs* in the *ViewModels* folder and replace the existing code with the following code:

```
namespace ContosoUniversity.ViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

In *InstructorController.cs*, replace the `HttpGet` `Edit` method with the following code. The changes are highlighted.

```
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Instructor instructor = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.Courses)
        .Where(i => i.ID == id)
        .Single();
    PopulateAssignedCourseData(instructor);
    if (instructor == null)
    {
        return HttpNotFound();
    }
    return View(instructor);
}

private void PopulateAssignedCourseData(Instructor instructor)
{
    var allCourses = db.Courses;
    var instructorCourses = new HashSet<int>(instructor.Courses.Select(c => c.CourseID));
    var viewModel = new List<AssignedCourseData>();
    foreach (var course in allCourses)
    {
        viewModel.Add(new AssignedCourseData
        {
            CourseID = course.CourseID,
            Title = course.Title,
            Assigned = instructorCourses.Contains(course.CourseID)
        });
    }
    ViewBag.Courses = viewModel;
}
```

The code adds eager loading for the `Courses` navigation property and calls the new `PopulateAssignedCourseData` method to provide information for the check box array using the `AssignedCourseData` view model class.

The code in the `PopulateAssignedCourseData` method reads through all `Course` entities in order to load a list of courses using the view model class. For each course, the code checks whether the course exists in the instructor's `Courses` navigation property. To create efficient lookup when checking whether a course is assigned to the instructor, the courses assigned to the instructor are put into a HashSet collection. The `Assigned` property is set to `true` for courses the instructor is assigned. The view will use this property to determine which check boxes must be displayed as selected. Finally, the list is passed to the view in a `ViewBag` property.

Next, add the code that's executed when the user clicks **Save**. Replace the `EditPost` method with the following code, which calls a new method that updates the `Courses` navigation property of the `Instructor` entity. The

changes are highlighted.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(int? id, string[] selectedCourses)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    var instructorToUpdate = db.Instructors
       .Include(i => i.OfficeAssignment)
       .Include(i => i.Courses)
       .Where(i => i.ID == id)
       .Single();

    if (TryUpdateModel(instructorToUpdate, "",
       new string[] { "LastName", "FirstMidName", "HireDate", "OfficeAssignment" }))
    {
        try
        {
            if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }

            UpdateInstructorCourses(selectedCourses, instructorToUpdate);

            db.SaveChanges();

            return RedirectToAction("Index");
        }
        catch (RetryLimitExceededException /* dex */)
        {
            //Log the error (uncomment dex variable name and add a line here to write a log.
            ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists, see
your system administrator.");
        }
    }
    PopulateAssignedCourseData(instructorToUpdate);
    return View(instructorToUpdate);
}
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
   if (selectedCourses == null)
   {
      instructorToUpdate.Courses = new List<Course>();
      return;
   }

   var selectedCoursesHS = new HashSet<string>(selectedCourses);
   var instructorCourses = new HashSet<int>
       (instructorToUpdate.Courses.Select(c => c.CourseID));
   foreach (var course in db.Courses)
   {
      if (selectedCoursesHS.Contains(course.CourseID.ToString()))
      {
         if (!instructorCourses.Contains(course.CourseID))
         {
            instructorToUpdate.Courses.Add(course);
         }
      }
      else
      {
         if (instructorCourses.Contains(course.CourseID))
         {
            instructorToUpdate.Courses.Remove(course);
         }
```

```
        }
    }
}
```

The method signature is now different from the `HttpGet` `Edit` method, so the method name changes from `EditPost` back to `Edit`.

Since the view doesn't have a collection of `Course` entities, the model binder can't automatically update the `Courses` navigation property. Instead of using the model binder to update the `Courses` navigation property, you'll do that in the new `UpdateInstructorCourses` method. Therefore you need to exclude the `Courses` property from model binding. This doesn't require any change to the code that calls TryUpdateModel because you're using the *whitelisting* overload and `Courses` isn't in the include list.

If no check boxes were selected, the code in `UpdateInstructorCourses` initializes the `Courses` navigation property with an empty collection:

```
if (selectedCourses == null)
{
    instructorToUpdate.Courses = new List<Course>();
    return;
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the view. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the check box for a course was selected but the course isn't in the `Instructor.Courses` navigation property, the course is added to the collection in the navigation property.

```
if (selectedCoursesHS.Contains(course.CourseID.ToString()))
{
    if (!instructorCourses.Contains(course.CourseID))
    {
        instructorToUpdate.Courses.Add(course);
    }
}
```

If the check box for a course wasn't selected, but the course is in the `Instructor.Courses` navigation property, the course is removed from the navigation property.

```
else
{
    if (instructorCourses.Contains(course.CourseID))
    {
        instructorToUpdate.Courses.Remove(course);
    }
}
```

In *Views\Instructor\Edit.cshtml*, add a **Courses** field with an array of check boxes by adding the following code immediately after the `div` elements for the `OfficeAssignment` field and before the `div` element for the **Save** button:

```
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;
                    List<ContosoUniversity.ViewModels.AssignedCourseData> courses = ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                               name="selectedCourses"
                               value="@course.CourseID"
                               @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                               @course.CourseID @:  @course.Title
                        @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

After you paste the code, if line breaks and indentation don't look like they do here, manually fix everything so that it looks like what you see here. The indentation doesn't have to be perfect, but the `@</tr><tr>`, `@:<td>`, `@:</td>`, and `@</tr>` lines must each be on a single line as shown or you'll get a runtime error.

This code creates an HTML table that has three columns. In each column is a check box followed by a caption that consists of the course number and title. The check boxes all have the same name ("selectedCourses"), which informs the model binder that they are to be treated as a group. The `value` attribute of each check box is set to the value of `CourseID.` When the page is posted, the model binder passes an array to the controller that consists of the `CourseID` values for only the check boxes which are selected.

When the check boxes are initially rendered, those that are for courses assigned to the instructor have `checked` attributes, which selects them (displays them checked).

After changing course assignments, you'll want to be able to verify the changes when the site returns to the `Index` page. Therefore, you need to add a column to the table in that page. In this case you don't need to use the `ViewBag` object, because the information you want to display is already in the `Courses` navigation property of the `Instructor` entity that you're passing to the page as the model.

In *Views\Instructor\Index.cshtml*, add a **Courses** heading immediately following the **Office** heading, as shown in the following example:

```
<tr>
    <th>Last Name</th>
    <th>First Name</th>
    <th>Hire Date</th>
    <th>Office</th>
    <th>Courses</th>
    <th></th>
</tr>
```

Then add a new detail cell immediately following the office location detail cell:

```
<td>
    @if (item.OfficeAssignment != null)
    {
        @item.OfficeAssignment.Location
    }
</td>
<td>
    @{
        foreach (var course in item.Courses)
        {
            @course.CourseID @:  @course.Title <br />
        }
    }
</td>
<td>
    @Html.ActionLink("Select", "Index", new { id = item.ID }) |
    @Html.ActionLink("Edit", "Edit", new { id = item.ID }) |
    @Html.ActionLink("Details", "Details", new { id = item.ID }) |
    @Html.ActionLink("Delete", "Delete", new { id = item.ID })
</td>
```

Run the **Instructor Index** page to see the courses assigned to each instructor.

Click **Edit** on an instructor to see the Edit page.

Change some course assignments and click **Save**. The changes you make are reflected on the Index page.

Note: The approach taken here to edit instructor course data works well when there is a limited number of courses. For collections that are much larger, a different UI and a different updating method would be required.

Update DeleteConfirmed

In *InstructorController.cs*, delete the `DeleteConfirmed` method and insert the following code in its place.

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
   Instructor instructor = db.Instructors
     .Include(i => i.OfficeAssignment)
     .Where(i => i.ID == id)
     .Single();

   db.Instructors.Remove(instructor);

    var department = db.Departments
        .Where(d => d.InstructorID == id)
        .SingleOrDefault();
    if (department != null)
    {
        department.InstructorID = null;
    }

   db.SaveChanges();
   return RedirectToAction("Index");
}
```

This code makes the following change:

- If the instructor is assigned as administrator of any department, removes the instructor assignment from that department. Without this code, you would get a referential integrity error if you tried to delete an instructor who was assigned as administrator for a department.

This code doesn't handle the scenario of one instructor assigned as administrator for multiple departments. In the last tutorial you'll add code that prevents that scenario from happening.

Add office location and courses to the Create page

In *InstructorController.cs*, delete the `HttpGet` and `HttpPost` `Create` methods, and then add the following code in their place:

```
public ActionResult Create()
{
    var instructor = new Instructor();
    instructor.Courses = new List<Course>();
    PopulateAssignedCourseData(instructor);
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "LastName,FirstMidName,HireDate,OfficeAssignment" )]Instructor
instructor, string[] selectedCourses)
{
    if (selectedCourses != null)
    {
        instructor.Courses = new List<Course>();
        foreach (var course in selectedCourses)
        {
            var courseToAdd = db.Courses.Find(int.Parse(course));
            instructor.Courses.Add(courseToAdd);
        }
    }
    if (ModelState.IsValid)
    {
        db.Instructors.Add(instructor);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}
```

This code is similar to what you saw for the Edit methods except that initially no courses are selected. The `HttpGet` `Create` method calls the `PopulateAssignedCourseData` method not because there might be courses selected but in order to provide an empty collection for the `foreach` loop in the view (otherwise the view code would throw a null reference exception).

The HttpPost Create method adds each selected course to the Courses navigation property before the template code that checks for validation errors and adds the new instructor to the database. Courses are added even if there are model errors so that when there are model errors (for an example, the user keyed an invalid date) so that when the page is redisplayed with an error message, any course selections that were made are automatically restored.

Notice that in order to be able to add courses to the `Courses` navigation property you have to initialize the property as an empty collection:

```
instructor.Courses = new List<Course>();
```

As an alternative to doing this in controller code, you could do it in the Instructor model by changing the property getter to automatically create the collection if it doesn't exist, as shown in the following example:

```
private ICollection<Course> _courses;
public virtual ICollection<Course> Courses
{
    get
    {
        return _courses ?? (_courses = new List<Course>());
    }
    set
    {
        _courses = value;
    }
}
```

If you modify the `Courses` property in this way, you can remove the explicit property initialization code in the controller.

In *Views\Instructor\Create.cshtml*, add an office location text box and course check boxes after the hire date field and before the **Submit** button.

```
<div class="form-group">
    @Html.LabelFor(model => model.OfficeAssignment.Location, new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.OfficeAssignment.Location)
        @Html.ValidationMessageFor(model => model.OfficeAssignment.Location)
    </div>
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;
                    List<ContosoUniversity.ViewModels.AssignedCourseData> courses = ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                               name="selectedCourses"
                               value="@course.CourseID"
                               @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                               @course.CourseID @:  @course.Title
                        @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

After you paste the code, fix line breaks and indentation as you did earlier for the Edit page.

Run the Create page and add an instructor.

Handling transactions

As explained in the Basic CRUD Functionality tutorial, by default the Entity Framework implicitly implements transactions. For scenarios where you need more control -- for example, if you want to include operations done

outside of Entity Framework in a transaction -- see Working with Transactions on MSDN.

Get the code

Download the Completed Project

Additional resources

Links to other Entity Framework resources can be found in ASP.NET Data Access - Recommended Resources.

Next step

In this tutorial, you:

- Customized courses pages
- Added office to instructors page
- Added courses to instructors page
- Updated DeleteConfirmed
- Added office location and courses to the Create page

Advance to the next article to learn how to implement an asynchronous programming model.

Asynchronous programming model

Tutorial: Use async and stored procedures with EF in an ASP.NET MVC App

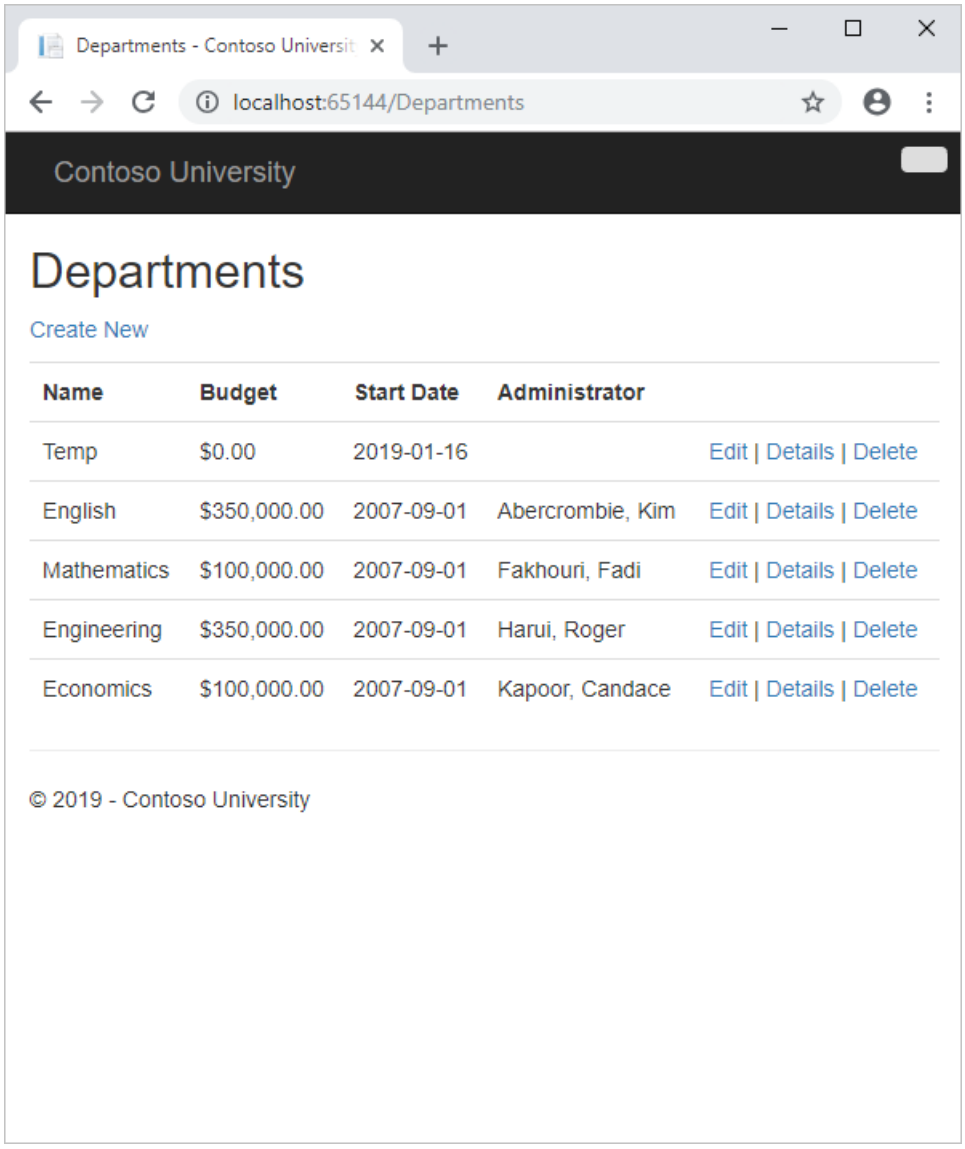
2/20/2020 • 8 minutes to read • Edit Online

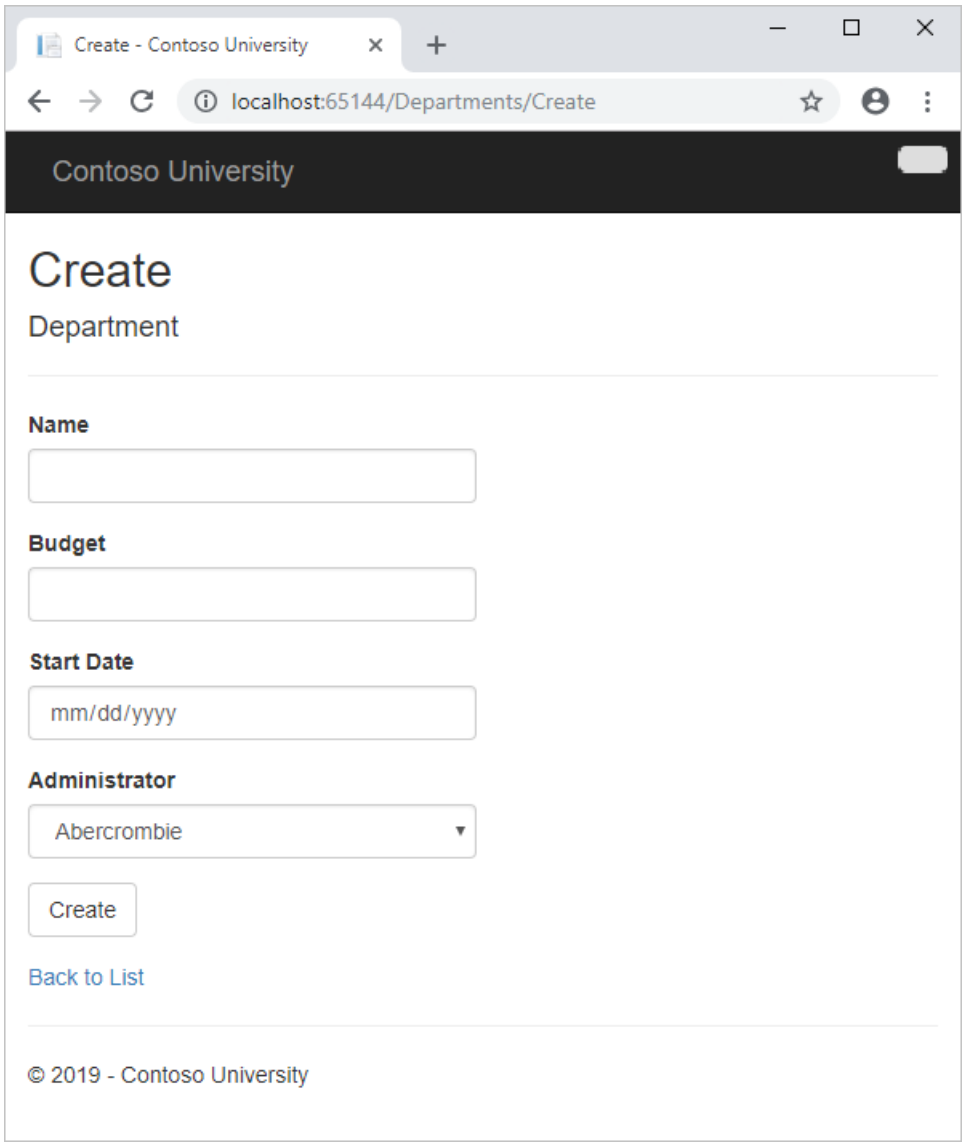
In earlier tutorials you learned how to read and update data using the synchronous programming model. In this tutorial you see how to implement the asynchronous programming model. Asynchronous code can help an application perform better because it makes better use of server resources.

In this tutorial you also see how to use stored procedures for insert, update, and delete operations on an entity.

Finally, you redeploy the application to Azure, along with all of the database changes that you've implemented since the first time you deployed.

The following illustrations show some of the pages that you'll work with.





In this tutorial, you:

- Learn about asynchronous code
- Create a Department controller
- Use stored procedures
- Deploy to Azure

Prerequisites

- Updating Related Data

Why use asynchronous code

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be use more efficiently, and the server is enabled to handle more traffic without delays.

In earlier versions of .NET, writing and testing asynchronous code was complex, error prone, and hard to debug. In .NET 4.5, writing, testing, and debugging asynchronous code is so much easier that you should generally write asynchronous code unless you have a reason not to. Asynchronous code does introduce a small amount of overhead, but for low traffic situations the performance hit is negligible, while for high traffic situations, the

potential performance improvement is substantial.

For more information about asynchronous programming, see Use .NET 4.5's async support to avoid blocking calls.

Create Department controller

Create a Department controller the same way you did the earlier controllers, except this time select the **Use async controller actions** check box.

The following highlights show what was added to the synchronous code for the `Index` method to make it asynchronous:

```
public async Task<ActionResult> Index()
{
    var departments = db.Departments.Include(d => d.Administrator);
    return View(await departments.ToListAsync());
}
```

Four changes were applied to enable the Entity Framework database query to execute asynchronously:

- The method is marked with the `async` keyword, which tells the compiler to generate callbacks for parts of the method body and to automatically create the `Task<ActionResult>` object that is returned.
- The return type was changed from `ActionResult` to `Task<ActionResult>`. The `Task<T>` type represents ongoing work with a result of type `T`.
- The `await` keyword was applied to the web service call. When the compiler sees this keyword, behind the scenes it splits the method into two parts. The first part ends with the operation that is started asynchronously. The second part is put into a callback method that is called when the operation completes.
- The asynchronous version of the `ToList` extension method was called.

Why is the `departments.ToList` statement modified but not the `departments = db.Departments` statement? The reason is that only statements that cause queries or commands to be sent to the database are executed asynchronously. The `departments = db.Departments` statement sets up a query but the query is not executed until the `ToList` method is called. Therefore, only the `ToList` method is executed asynchronously.

In the `Details` method and the `HttpGet` `Edit` and `Delete` methods, the `Find` method is the one that causes a query to be sent to the database, so that's the method that gets executed asynchronously:

```
public async Task<ActionResult> Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Department department = await db.Departments.FindAsync(id);
    if (department == null)
    {
        return HttpNotFound();
    }
    return View(department);
}
```

In the `Create`, `HttpPost Edit`, and `DeleteConfirmed` methods, it is the `SaveChanges` method call that causes a command to be executed, not statements such as `db.Departments.Add(department)` which only cause entities in memory to be modified.

```
public async Task<ActionResult> Create(Department department)
{
    if (ModelState.IsValid)
    {
        db.Departments.Add(department);
    await db.SaveChangesAsync();
        return RedirectToAction("Index");
    }
```

Open *Views\Department\Index.cshtml*, and replace the template code with the following code:

```
@model IEnumerable<ContosoUniversity.Models.Department>
@{
    ViewBag.Title = "Departments";
}
<h2>Departments</h2>
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Name)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Budget)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.StartDate)
        </th>
    <th>
            Administrator
        </th>
        <th></th>
    </tr>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Budget)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.StartDate)
        </td>
    <td>
            @Html.DisplayFor(modelItem => item.Administrator.FullName)
            </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.DepartmentID }) |
            @Html.ActionLink("Details", "Details", new { id=item.DepartmentID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.DepartmentID })
        </td>
    </tr>
}
</table>
```

This code changes the title from Index to Departments, moves the Administrator name to the right, and provides the full name of the administrator.

In the Create, Delete, Details, and Edit views, change the caption for the `InstructorID` field to "Administrator" the same way you changed the department name field to "Department" in the Course views.

In the Create and Edit views use the following code:

```
<label class="control-label col-md-2" for="InstructorID">Administrator</label>
```

In the Delete and Details views use the following code:

```
<dt>
    Administrator
</dt>
```

Run the application, and click the **Departments** tab.

Everything works the same as in the other controllers, but in this controller all of the SQL queries are executing asynchronously.

Some things to be aware of when you are using asynchronous programming with the Entity Framework:

- The async code is not thread safe. In other words, in other words, don't try to do multiple operations in parallel using the same context instance.
- If you want to take advantage of the performance benefits of async code, make sure that any library packages that you're using (such as for paging), also use async if they call any Entity Framework methods that cause queries to be sent to the database.

Use stored procedures

Some developers and DBAs prefer to use stored procedures for database access. In earlier versions of Entity Framework you can retrieve data using a stored procedure by executing a raw SQL query, but you can't instruct EF to use stored procedures for update operations. In EF 6 it's easy to configure Code First to use stored procedures.

1. In *DAL\SchoolContext.cs*, add the highlighted code to the `OnModelCreating` method.

   ```
   protected override void OnModelCreating(DbModelBuilder modelBuilder)
   {
       modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
       modelBuilder.Entity<Course>()
           .HasMany(c => c.Instructors).WithMany(i => i.Courses)
           .Map(t => t.MapLeftKey("CourseID")
               .MapRightKey("InstructorID")
               .ToTable("CourseInstructor"));
       modelBuilder.Entity<Department>().MapToStoredProcedures();
   }
   ```

   This code instructs Entity Framework to use stored procedures for insert, update, and delete operations on the `Department` entity.

2. In Package Manage Console, enter the following command:

   ```
   add-migration DepartmentSP
   ```

   Open *Migrations\<timestamp>_DepartmentSP.cs* to see the code in the `Up` method that creates Insert, Update, and Delete stored procedures:

```
public override void Up()
{
    CreateStoredProcedure(
        "dbo.Department_Insert",
        p => new
            {
                Name = p.String(maxLength: 50),
                Budget = p.Decimal(precision: 19, scale: 4, storeType: "money"),
                StartDate = p.DateTime(),
                InstructorID = p.Int(),
            },
        body:
            @"INSERT [dbo].[Department]([Name], [Budget], [StartDate], [InstructorID])
              VALUES (@Name, @Budget, @StartDate, @InstructorID)

              DECLARE @DepartmentID int
              SELECT @DepartmentID = [DepartmentID]
              FROM [dbo].[Department]
              WHERE @@ROWCOUNT > 0 AND [DepartmentID] = scope_identity()

              SELECT t0.[DepartmentID]
              FROM [dbo].[Department] AS t0
              WHERE @@ROWCOUNT > 0 AND t0.[DepartmentID] = @DepartmentID"
    );

    CreateStoredProcedure(
        "dbo.Department_Update",
        p => new
            {
                DepartmentID = p.Int(),
                Name = p.String(maxLength: 50),
                Budget = p.Decimal(precision: 19, scale: 4, storeType: "money"),
                StartDate = p.DateTime(),
                InstructorID = p.Int(),
            },
        body:
            @"UPDATE [dbo].[Department]
              SET [Name] = @Name, [Budget] = @Budget, [StartDate] = @StartDate, [InstructorID] =
@InstructorID
              WHERE ([DepartmentID] = @DepartmentID)"
    );

    CreateStoredProcedure(
        "dbo.Department_Delete",
        p => new
            {
                DepartmentID = p.Int(),
            },
        body:
            @"DELETE [dbo].[Department]
              WHERE ([DepartmentID] = @DepartmentID)"
    );
}
```

3. In Package Manage Console, enter the following command:

   `update-database`

4. Run the application in debug mode, click the **Departments** tab, and then click **Create New**.

5. Enter data for a new department, and then click **Create**.

6. In Visual Studio, look at the logs in the **Output** window to see that a stored procedure was used to insert the new Department row.

```
Output
Show output from: Debug
    FROM [dbo].[Instructor] AS [Extent1]:
iisexpress.exe Information: 0 : Component:SQL
  Database;Method:SchoolInterceptor.ReaderExecuting;Timespan:00:00:00.0
  000047;Properties:Command: [dbo].[Department_Insert]:
iisexpress.exe Information: 0 : Component:SQL
  Database;Method:SchoolInterceptor.ReaderExecuting;Timespan:00:00:00.0
  000041;Properties:Command: SELECT
    [Extent1].[DepartmentID] AS [DepartmentID],
    [Extent1].[Name] AS [Name],
    [Extent1].[Budget] AS [Budget],
    [Extent1].[StartDate] AS [StartDate],
    [Extent1].[InstructorID] AS [InstructorID],
    [Extent2].[ID] AS [ID],
    [Extent2].[LastName] AS [LastName],
    [Extent2].[FirstName] AS [FirstName],
    [Extent2].[HireDate] AS [HireDate]
    FROM  [dbo].[Department] AS [Extent1]
    LEFT OUTER JOIN [dbo].[Instructor] AS [Extent2] ON [Extent1].
      [InstructorID] = [Extent2].[ID]:
```

Code First creates default stored procedure names. If you are using an existing database, you might need to customize the stored procedure names in order to use stored procedures already defined in the database. For information about how to do that, see Entity Framework Code First Insert/Update/Delete Stored Procedures.

If you want to customize what generated stored procedures do, you can edit the scaffolded code for the migrations `Up` method that creates the stored procedure. That way your changes are reflected whenever that migration is run and will be applied to your production database when migrations runs automatically in production after deployment.

If you want to change an existing stored procedure that was created in a previous migration, you can use the Add-Migration command to generate a blank migration, and then manually write code that calls the AlterStoredProcedure method.

Deploy to Azure

This section requires you to have completed the optional **Deploying the app to Azure** section in the Migrations and Deployment tutorial of this series. If you had migrations errors that you resolved by deleting the database in your local project, skip this section.

1. In Visual Studio, right-click the project in **Solution Explorer** and select **Publish** from the context menu.
2. Click **Publish**.

   Visual Studio deploys the application to Azure, and the application opens in your default browser, running in Azure.
3. Test the application to verify it's working.

   The first time you run a page that accesses the database, the Entity Framework runs all of the migrations `Up` methods required to bring the database up to date with the current data model. You can now use all of the web pages that you added since the last time you deployed, including the Department pages that you added in this tutorial.

Get the code

Download the Completed Project

Additional resources

Links to other Entity Framework resources can be found in the ASP.NET Data Access - Recommended Resources.

Next steps

In this tutorial, you:

- Learned about asynchronous code
- Created a Department controller
- Used stored procedures
- Deployed to Azure

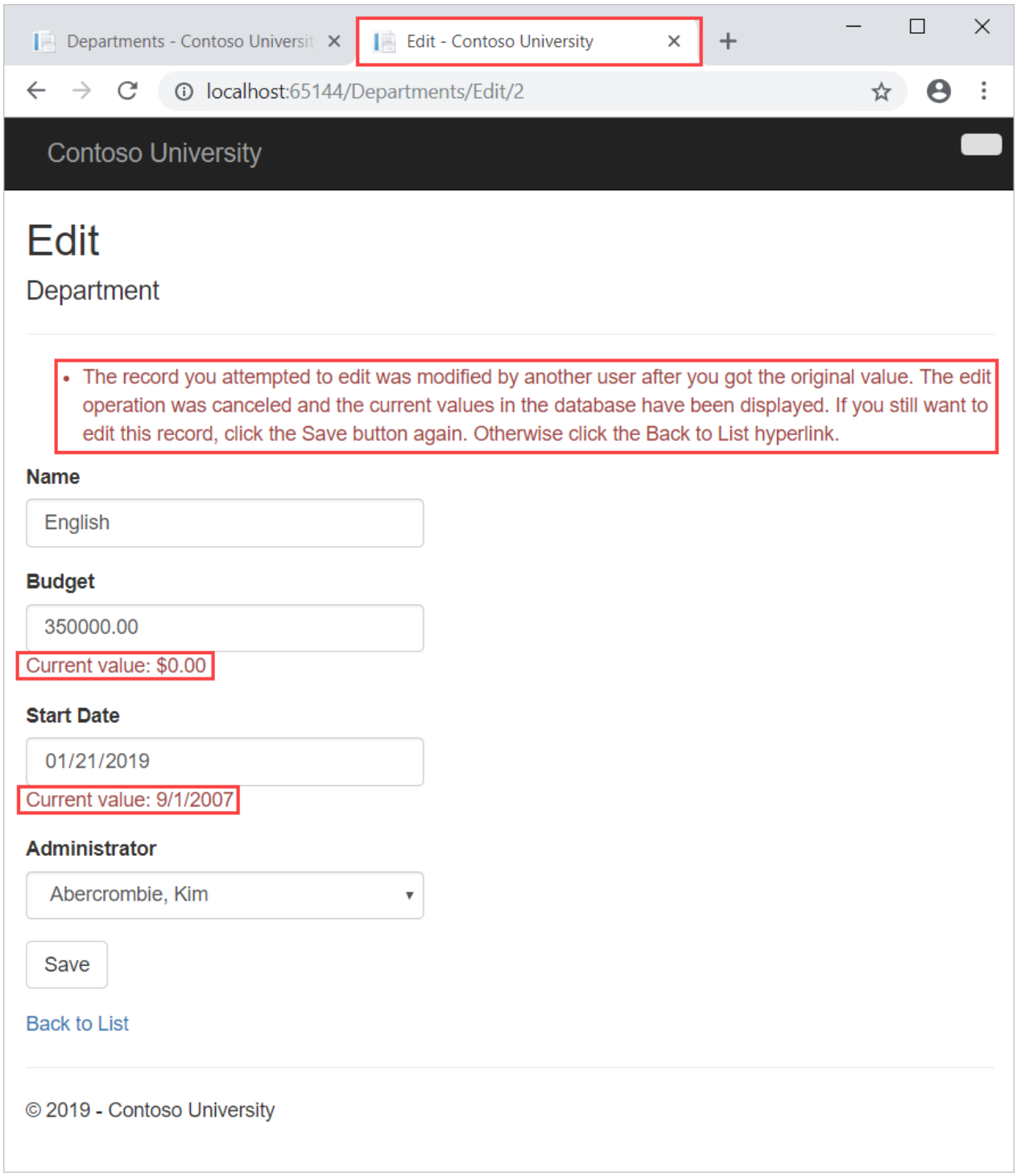
Advance to the next article to learn how to handle conflicts when multiple users update the same entity at the same time.

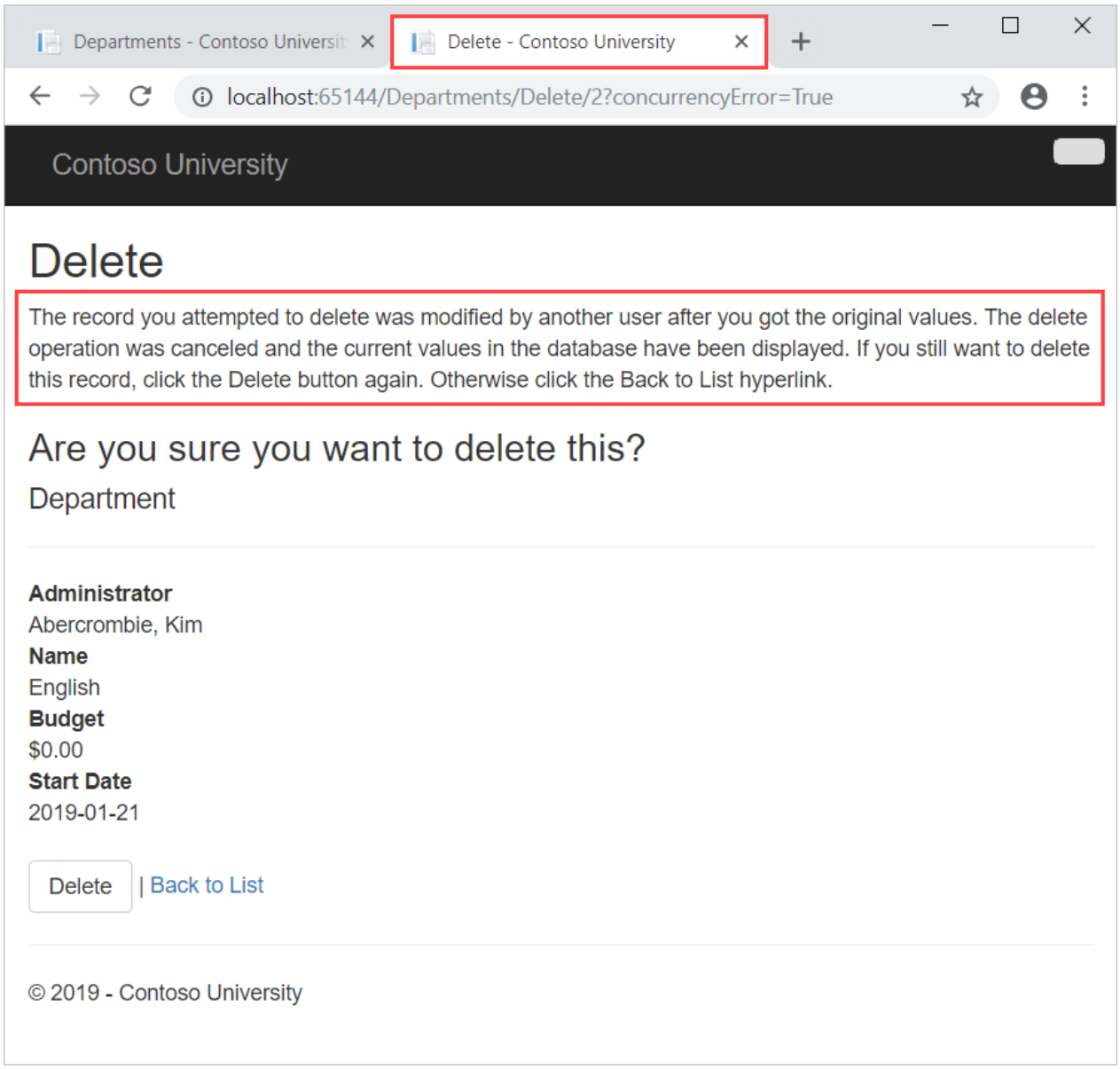
Handling concurrency

Tutorial: Handle Concurrency with EF in an ASP.NET MVC 5 app

2/20/2020 • 16 minutes to read • Edit Online

In earlier tutorials you learned how to update data. This tutorial shows how to use optimistic concurrency to handle conflicts when multiple users update the same entity at the same time. You change the web pages that work with the `Department` entity so that they handle concurrency errors. The following illustrations show the Edit and Delete pages, including some messages that are displayed if a concurrency conflict occurs.





In this tutorial, you:

- Learn about concurrency conflicts
- Add optimistic concurrency
- Modify Department controller
- Test concurrency handling
- Update the Delete page

Prerequisites

- Async and Stored Procedures

Concurrency conflicts

A concurrency conflict occurs when one user displays an entity's data in order to edit it, and then another user updates the same entity's data before the first user's change is written to the database. If you don't enable the detection of such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable: if there are few users, or few updates, or if isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit. In that case, you don't have to configure the application to handle concurrency conflicts.

Pessimistic Concurrency (Locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called *pessimistic concurrency*. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

Managing locks has disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases. For these reasons, not all database management systems support pessimistic concurrency. The Entity Framework provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

The alternative to pessimistic concurrency is *optimistic concurrency*. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, John runs the Departments Edit page, changes the **Budget** amount for the English department from $350,000.00 to $0.00.

Before John clicks **Save**, Jane runs the same page and changes the **Start Date** field from 9/1/2007 to 8/8/2013.

John clicks **Save** first and sees his change when the browser returns to the Index page, then Jane clicks **Save**. What happens next is determined by how you handle concurrency conflicts. Some of the options include the following:

- You can keep track of which property a user has modified and update only the corresponding columns in the database. In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the English department, they'll see both John's and Jane's changes — a start date of 8/8/2013 and a budget of Zero dollars.

  This method of updating can reduce the number of conflicts that could result in data loss, but it can't avoid data loss if competing changes are made to the same property of an entity. Whether the Entity Framework works this way depends on how you implement your update code. It's often not practical in a web application, because it can require that you maintain large amounts of state in order to keep track of all original property values for an entity as well as new values. Maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself (for example, in hidden fields) or in a cookie.

- You can let Jane's change overwrite John's change. The next time someone browses the English department, they'll see 8/8/2013 and the restored $350,000.00 value. This is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) As noted in the introduction to this section, if you don't do any coding for concurrency handling, this will happen automatically.

- You can prevent Jane's change from being updated in the database. Typically, you would display an error message, show her the current state of the data, and allow her to reapply her changes if she still wants to make them. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You'll implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted to what's happening.

Detecting Concurrency Conflicts

You can resolve conflicts by handling OptimisticConcurrencyException exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database and the data model appropriately. Some options for enabling conflict detection include the following:

- In the database table, include a tracking column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the `Where` clause of SQL

`Update` or `Delete` commands.

The data type of the tracking column is typically rowversion. The rowversion value is a sequential number that's incremented each time the row is updated. In an `Update` or `Delete` command, the `Where` clause includes the original value of the tracking column (the original row version). If the row being updated has been changed by another user, the value in the `rowversion` column is different than the original value, so the `Update` or `Delete` statement can't find the row to update because of the `Where` clause. When the Entity Framework finds that no rows have been updated by the `Update` or `Delete` command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

- Configure the Entity Framework to include the original values of every column in the table in the `Where` clause of `Update` and `Delete` commands.

  As in the first option, if anything in the row has changed since the row was first read, the `Where` clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. For database tables that have many columns, this approach can result in very large `Where` clauses, and can require that you maintain large amounts of state. As noted earlier, maintaining large amounts of state can affect application performance. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.

  If you do want to implement this approach to concurrency, you have to mark all non-primary-key properties in the entity you want to track concurrency for by adding the ConcurrencyCheck attribute to them. That change enables the Entity Framework to include all columns in the SQL `WHERE` clause of `UPDATE` statements.

In the remainder of this tutorial you'll add a rowversion tracking property to the `Department` entity, create a controller and views, and test to verify that everything works correctly.

Add optimistic concurrency

In *Models\Department.cs*, add a tracking property named `RowVersion`:

```
public class Department
{
    public int DepartmentID { get; set; }

    [StringLength(50, MinimumLength = 3)]
    public string Name { get; set; }

    [DataType(DataType.Currency)]
    [Column(TypeName = "money")]
    public decimal Budget { get; set; }

    [DataType(DataType.Date)]
    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
    [Display(Name = "Start Date")]
    public DateTime StartDate { get; set; }

    [Display(Name = "Administrator")]
    public int? InstructorID { get; set; }

    [Timestamp]
    public byte[] RowVersion { get; set; }

    public virtual Instructor Administrator { get; set; }
    public virtual ICollection<Course> Courses { get; set; }
}
```

The Timestamp attribute specifies that this column will be included in the `Where` clause of `Update` and `Delete`

commands sent to the database. The attribute is called Timestamp because previous versions of SQL Server used a SQL timestamp data type before the SQL rowversion replaced it. The .Net type for rowversion is a byte array.

If you prefer to use the fluent API, you can use the IsConcurrencyToken method to specify the tracking property, as shown in the following example:

```
modelBuilder.Entity<Department>()
    .Property(p => p.RowVersion).IsConcurrencyToken();
```

By adding a property you changed the database model, so you need to do another migration. In the Package Manager Console (PMC), enter the following commands:

```
Add-Migration RowVersion
Update-Database
```

Modify Department controller

In *Controllers\DepartmentController.cs*, add a `using` statement:

```
using System.Data.Entity.Infrastructure;
```

In the *DepartmentController.cs* file, change all four occurrences of "LastName" to "FullName" so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

```
ViewBag.InstructorID = new SelectList(db.Instructors, "ID", "FullName");
```

Replace the existing code for the `HttpPost` `Edit` method with the following code:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Edit(int? id, byte[] rowVersion)
{
    string[] fieldsToBind = new string[] { "Name", "Budget", "StartDate", "InstructorID", "RowVersion" };

    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    var departmentToUpdate = await db.Departments.FindAsync(id);
    if (departmentToUpdate == null)
    {
        Department deletedDepartment = new Department();
        TryUpdateModel(deletedDepartment, fieldsToBind);
        ModelState.AddModelError(string.Empty,
            "Unable to save changes. The department was deleted by another user.");
        ViewBag.InstructorID = new SelectList(db.Instructors, "ID", "FullName",
deletedDepartment.InstructorID);
        return View(deletedDepartment);
    }

    if (TryUpdateModel(departmentToUpdate, fieldsToBind))
    {
        try
        {
            db.Entry(departmentToUpdate).OriginalValues["RowVersion"] = rowVersion;
            await db.SaveChangesAsync();
```

```
            return RedirectToAction("Index");
        }
        catch (DbUpdateConcurrencyException ex)
        {
            var entry = ex.Entries.Single();
            var clientValues = (Department)entry.Entity;
            var databaseEntry = entry.GetDatabaseValues();
            if (databaseEntry == null)
            {
                ModelState.AddModelError(string.Empty,
                    "Unable to save changes. The department was deleted by another user.");
            }
            else
            {
                var databaseValues = (Department)databaseEntry.ToObject();

                if (databaseValues.Name != clientValues.Name)
                    ModelState.AddModelError("Name", "Current value: "
                        + databaseValues.Name);
                if (databaseValues.Budget != clientValues.Budget)
                    ModelState.AddModelError("Budget", "Current value: "
                        + String.Format("{0:c}", databaseValues.Budget));
                if (databaseValues.StartDate != clientValues.StartDate)
                    ModelState.AddModelError("StartDate", "Current value: "
                        + String.Format("{0:d}", databaseValues.StartDate));
                if (databaseValues.InstructorID != clientValues.InstructorID)
                    ModelState.AddModelError("InstructorID", "Current value: "
                        + db.Instructors.Find(databaseValues.InstructorID).FullName);
                ModelState.AddModelError(string.Empty, "The record you attempted to edit "
                    + "was modified by another user after you got the original value. The "
                    + "edit operation was canceled and the current values in the database "
                    + "have been displayed. If you still want to edit this record, click "
                    + "the Save button again. Otherwise click the Back to List hyperlink.");
                departmentToUpdate.RowVersion = databaseValues.RowVersion;
            }
        }
        catch (RetryLimitExceededException /* dex */)
        {
            //Log the error (uncomment dex variable name and add a line here to write a log.)
            ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists, see
your system administrator.");
        }
    }
    ViewBag.InstructorID = new SelectList(db.Instructors, "ID", "FullName", departmentToUpdate.InstructorID);
    return View(departmentToUpdate);
}
```

If the `FindAsync` method returns null, the department was deleted by another user. The code shown uses the posted form values to create a department entity so that the Edit page can be redisplayed with an error message. As an alternative, you wouldn't have to re-create the department entity if you display only an error message without redisplaying the department fields.

The view stores the original `RowVersion` value in a hidden field, and the method receives it in the `rowVersion` parameter. Before you call `SaveChanges`, you have to put that original `RowVersion` property value in the `OriginalValues` collection for the entity. Then when the Entity Framework creates a SQL `UPDATE` command, that command will include a `WHERE` clause that looks for a row that has the original `RowVersion` value.

If no rows are affected by the `UPDATE` command (no rows have the original `RowVersion` value), the Entity Framework throws a `DbUpdateConcurrencyException` exception, and the code in the `catch` block gets the affected `Department` entity from the exception object.

```
var entry = ex.Entries.Single();
```

This object has the new values entered by the user in its `Entity` property, and you can get the values read from the database by calling the `GetDatabaseValues` method.

```
var clientValues = (Department)entry.Entity;
var databaseEntry = entry.GetDatabaseValues();
```

The `GetDatabaseValues` method returns null if someone has deleted the row from the database; otherwise, you have to cast the returned object to the `Department` class in order to access the `Department` properties. (Because you already checked for deletion, `databaseEntry` would be null only if the department was deleted after `FindAsync` executes and before `SaveChanges` executes.)

```
if (databaseEntry == null)
{
    ModelState.AddModelError(string.Empty,
        "Unable to save changes. The department was deleted by another user.");
}
else
{
    var databaseValues = (Department)databaseEntry.ToObject();
```

Next, the code adds a custom error message for each column that has database values different from what the user entered on the Edit page:

```
if (databaseValues.Name != currentValues.Name)
    ModelState.AddModelError("Name", "Current value: " + databaseValues.Name);
    // ...
```

A longer error message explains what happened and what to do about it:

```
ModelState.AddModelError(string.Empty, "The record you attempted to edit "
    + "was modified by another user after you got the original value. The"
    + "edit operation was canceled and the current values in the database "
    + "have been displayed. If you still want to edit this record, click "
    + "the Save button again. Otherwise click the Back to List hyperlink.");
```

Finally, the code sets the `RowVersion` value of the `Department` object to the new value retrieved from the database. This new `RowVersion` value will be stored in the hidden field when the Edit page is redisplayed, and the next time the user clicks **Save**, only concurrency errors that happen since the redisplay of the Edit page will be caught.

In *Views\Department\Edit.cshtml*, add a hidden field to save the `RowVersion` property value, immediately following the hidden field for the `DepartmentID` property:

```
@model ContosoUniversity.Models.Department

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Department</h4>
        <hr />
        @Html.ValidationSummary(true)
        @Html.HiddenFor(model => model.DepartmentID)
        @Html.HiddenFor(model => model.RowVersion)
```

Test concurrency handling

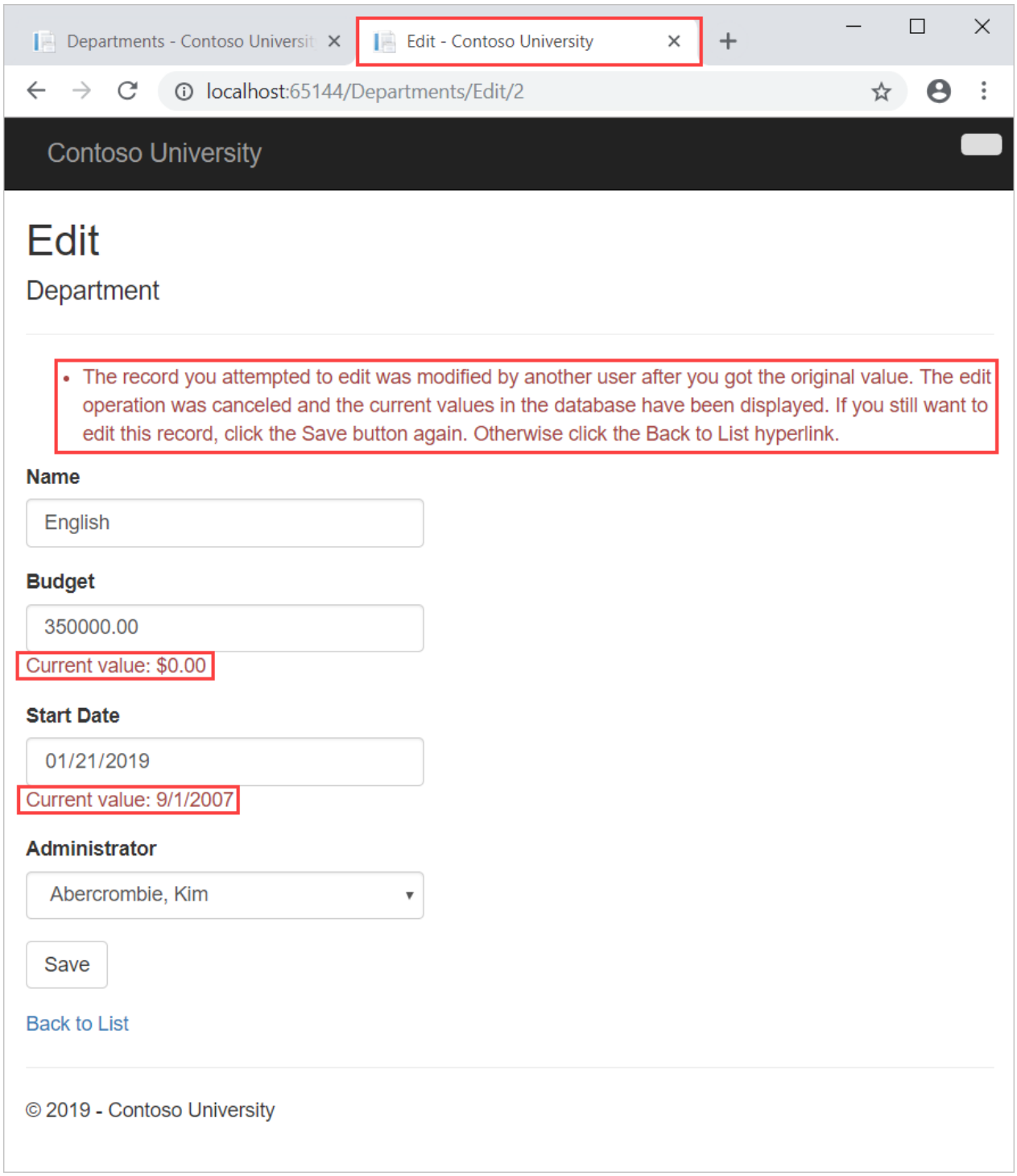
Run the site and click **Departments**.

Right click the **Edit** hyperlink for the English department and select **Open in new tab,** then click the **Edit** hyperlink for the English department. The two tabs display the same information.

Change a field in the first browser tab and click **Save**.

The browser shows the Index page with the changed value.

Change a field in the second browser tab and click **Save**. You see an error message:



Click **Save** again. The value you entered in the second browser tab is saved along with the original value of the data you changed in the first browser. You see the saved values when the Index page appears.

Update the Delete page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the `HttpGet` `Delete` method displays the confirmation view, the view includes the original `RowVersion` value in a hidden field. That value is then available to the `HttpPost` `Delete` method that's called when the user confirms the deletion. When the Entity Framework creates the SQL `DELETE` command, it includes a `WHERE` clause with the original `RowVersion` value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the `HttpGet Delete` method is called with an error flag set to `true` in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case a different error message is displayed.

In *DepartmentController.cs*, replace the `HttpGet` `Delete` method with the following code:

```
public async Task<ActionResult> Delete(int? id, bool? concurrencyError)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Department department = await db.Departments.FindAsync(id);
    if (department == null)
    {
        if (concurrencyError.GetValueOrDefault())
        {
            return RedirectToAction("Index");
        }
        return HttpNotFound();
    }

    if (concurrencyError.GetValueOrDefault())
    {
        ViewBag.ConcurrencyErrorMessage = "The record you attempted to delete "
            + "was modified by another user after you got the original values. "
            + "The delete operation was canceled and the current values in the "
            + "database have been displayed. If you still want to delete this "
            + "record, click the Delete button again. Otherwise "
            + "click the Back to List hyperlink.";
    }

    return View(department);
}
```

The method accepts an optional parameter that indicates whether the page is being redisplayed after a concurrency error. If this flag is `true`, an error message is sent to the view using a `ViewBag` property.

Replace the code in the `HttpPost` `Delete` method (named `DeleteConfirmed`) with the following code:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Delete(Department department)
{
    try
    {
        db.Entry(department).State = EntityState.Deleted;
        await db.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    catch (DbUpdateConcurrencyException)
    {
        return RedirectToAction("Delete", new { concurrencyError = true, id=department.DepartmentID });
    }
    catch (DataException /* dex */)
    {
        //Log the error (uncomment dex variable name after DataException and add a line here to write a log.
        ModelState.AddModelError(string.Empty, "Unable to delete. Try again, and if the problem persists
contact your system administrator.");
        return View(department);
    }
}
```

In the scaffolded code that you just replaced, this method accepted only a record ID:

```
public async Task<ActionResult> DeleteConfirmed(int id)
```

You've changed this parameter to a `Department` entity instance created by the model binder. This gives you access to the `RowVersion` property value in addition to the record key.

```
public async Task<ActionResult> Delete(Department department)
```

You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code named the `HttpPost` `Delete` method `DeleteConfirmed` to give the `HttpPost` method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the `HttpPost` and `HttpGet` delete methods.

If a concurrency error is caught, the code redisplays the Delete confirmation page and provides a flag that indicates it should display a concurrency error message.

In *Views\Department\Delete.cshtml*, replace the scaffolded code with the following code that adds an error message field and hidden fields for the DepartmentID and RowVersion properties. The changes are highlighted.

```
@model ContosoUniversity.Models.Department

@{
    ViewBag.Title = "Delete";
}

<h2>Delete</h2>

<p class="error">@ViewBag.ConcurrencyErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            Administrator
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>

        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>

        <dt>
            @Html.DisplayNameFor(model => model.Budget)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.Budget)
        </dd>

        <dt>
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>

        <dd>
            @Html.DisplayFor(model => model.StartDate)
        </dd>

    </dl>

    @using (Html.BeginForm()) {
        @Html.AntiForgeryToken()
        @Html.HiddenFor(model => model.DepartmentID)
        @Html.HiddenFor(model => model.RowVersion)

        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            @Html.ActionLink("Back to List", "Index")
        </div>
    }
</div>
```

This code adds an error message between the `h2` and `h3` headings:

```
<p class="error">@ViewBag.ConcurrencyErrorMessage</p>
```

It replaces `LastName` with `FullName` in the `Administrator` field:

```
<dt>
  Administrator
</dt>
<dd>
  @Html.DisplayFor(model => model.Administrator.FullName)
</dd>
```

Finally, it adds hidden fields for the `DepartmentID` and `RowVersion` properties after the `Html.BeginForm` statement:

```
@Html.HiddenFor(model => model.DepartmentID)
@Html.HiddenFor(model => model.RowVersion)
```

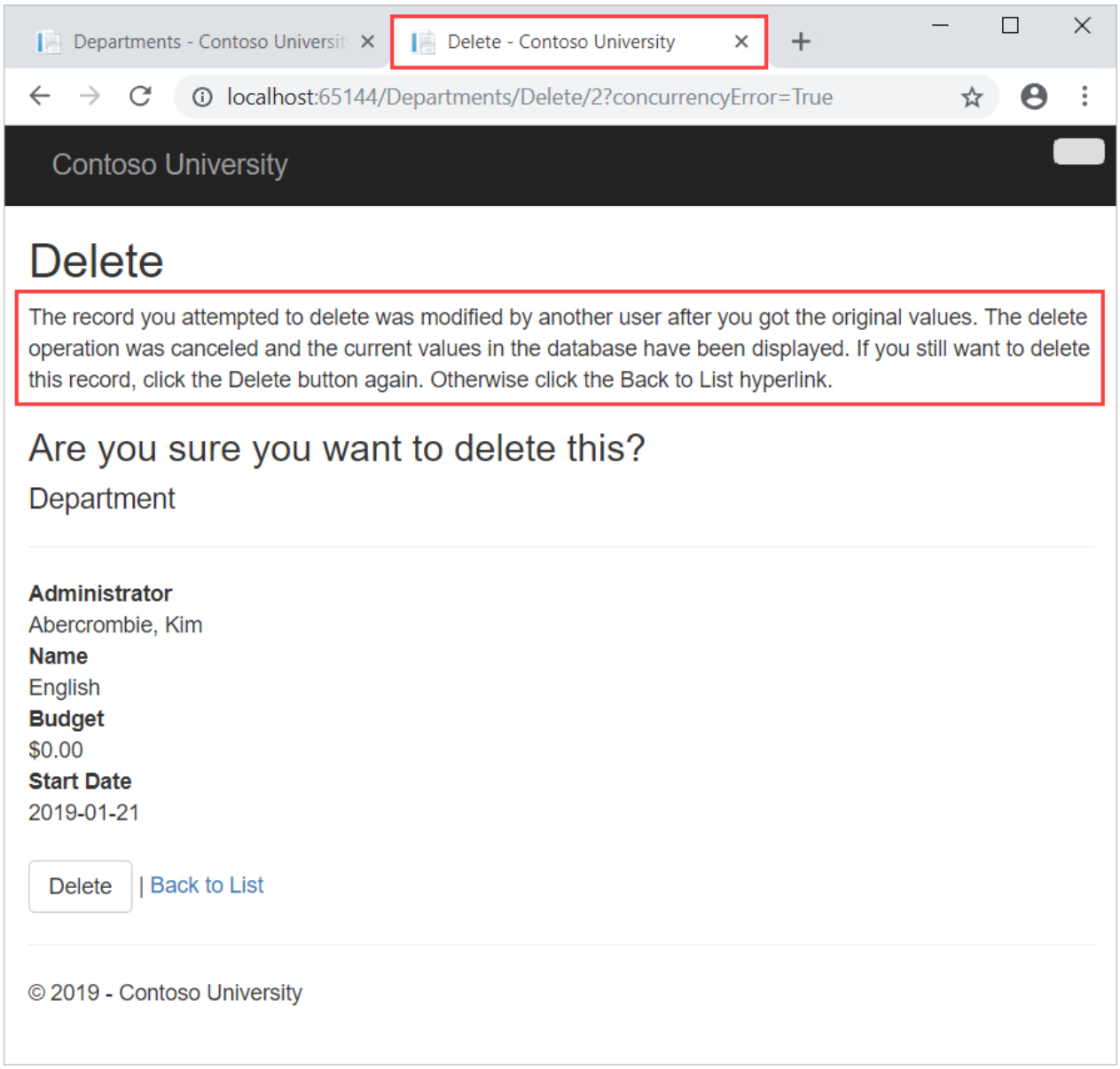
Run the Departments Index page. Right click the **Delete** hyperlink for the English department and select **Open in new tab,** then in the first tab click the **Edit** hyperlink for the English department.

In the first window, change one of the values, and click **Save**.

The Index page confirms the change.

In the second tab, click **Delete**.

You see the concurrency error message, and the Department values are refreshed with what's currently in the database.



If you click **Delete** again, you're redirected to the Index page, which shows that the department has been deleted.

Get the code

Download Completed Project

Additional resources

Links to other Entity Framework resources can be found in the ASP.NET Data Access - Recommended Resources.

For information about other ways to handle various concurrency scenarios, see Optimistic Concurrency Patterns and Working with Property Values on MSDN. The next tutorial shows how to implement table-per-hierarchy inheritance for the `Instructor` and `Student` entities.

Next steps

In this tutorial, you:

- Learned about concurrency conflicts
- Added optimistic concurrency
- Modified Department controller
- Tested concurrency handling

- Updated the Delete page

Advance to the next article to learn how to implement inheritance in the data model.

Implement inheritance in the data model

Tutorial: Implement Inheritance with EF in an ASP.NET MVC 5 app

2/20/2020 • 9 minutes to read • Edit Online

In the previous tutorial you handled concurrency exceptions. This tutorial will show you how to implement inheritance in the data model.

In object-oriented programming, you can use inheritance to facilitate code reuse. In this tutorial, you'll change the `Instructor` and `Student` classes so that they derive from a `Person` base class which contains properties such as `LastName` that are common to both instructors and students. You won't add or change any web pages, but you'll change some of the code and those changes will be automatically reflected in the database.

In this tutorial, you:

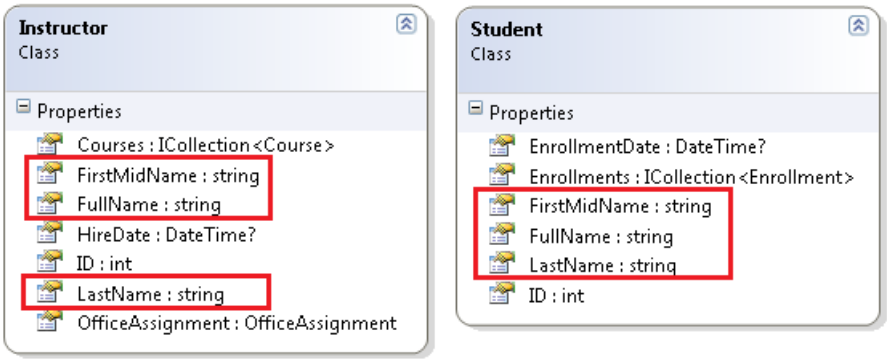
- Learn to map inheritance to database
- Create the Person class
- Update Instructor and Student
- Add Person to the Model
- Create and Update Migrations
- Test the implementation
- Deploy to Azure

Prerequisites

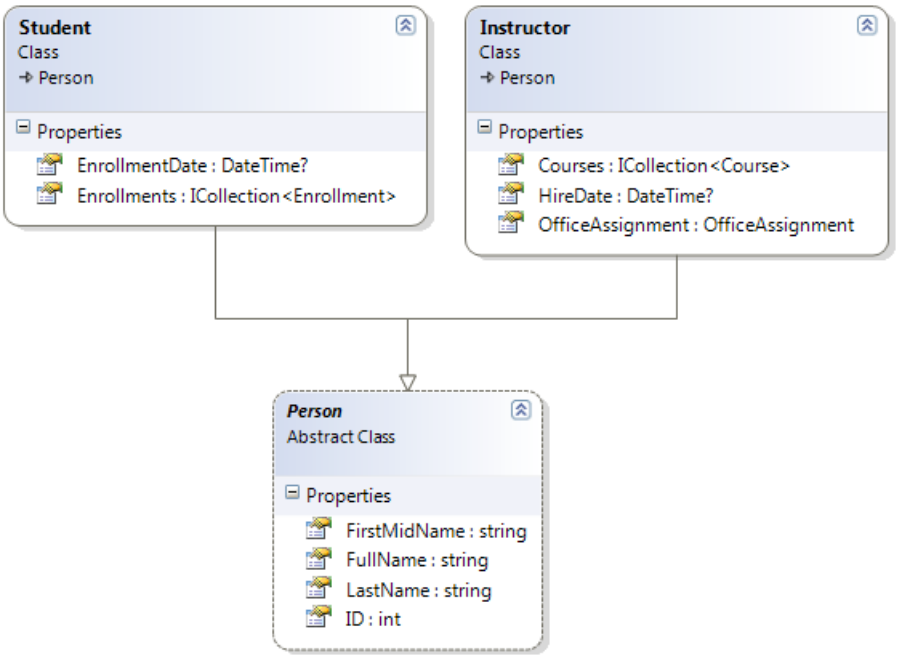
- Handling Concurrency

Map inheritance to database

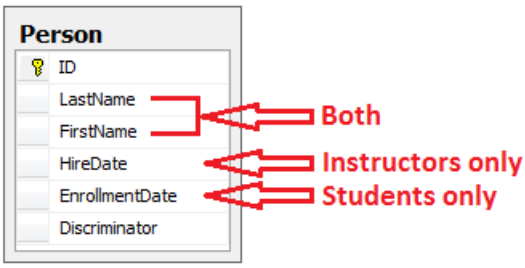
The `Instructor` and `Student` classes in the `School` data model have several properties that are identical:



Suppose you want to eliminate the redundant code for the properties that are shared by the `Instructor` and `Student` entities. Or you want to write a service that can format names without caring whether the name came from an instructor or a student. You could create a `Person` base class which contains only those shared properties, then make the `Instructor` and `Student` entities inherit from that base class, as shown in the following illustration:

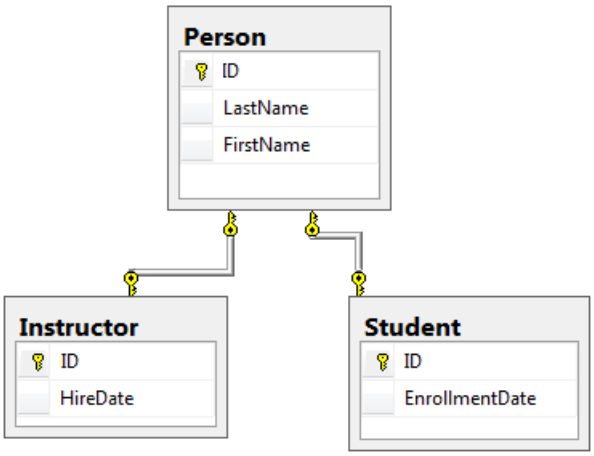


There are several ways this inheritance structure could be represented in the database. You could have a `Person` table that includes information about both students and instructors in a single table. Some of the columns could apply only to instructors (`HireDate`), some only to students (`EnrollmentDate`), some to both (`LastName`, `FirstName`). Typically, you'd have a *discriminator* column to indicate which type each row represents. For example, the discriminator column might have "Instructor" for instructors and "Student" for students.



This pattern of generating an entity inheritance structure from a single database table is called *table-per-hierarchy* (TPH) inheritance.

An alternative is to make the database look more like the inheritance structure. For example, you could have only the name fields in the `Person` table and have separate `Instructor` and `Student` tables with the date fields.



This pattern of making a database table for each entity class is called *table per type* (TPT) inheritance.

Yet another option is to map all non-abstract types to individual tables. All properties of a class, including inherited

properties, map to columns of the corresponding table. This pattern is called Table-per-Concrete Class (TPC) inheritance. If you implemented TPC inheritance for the `Person`, `Student`, and `Instructor` classes as shown earlier, the `Student` and `Instructor` tables would look no different after implementing inheritance than they did before.

TPC and TPH inheritance patterns generally deliver better performance in the Entity Framework than TPT inheritance patterns, because TPT patterns can result in complex join queries.

This tutorial demonstrates how to implement TPH inheritance. TPH is the default inheritance pattern in the Entity Framework, so all you have to do is create a `Person` class, change the `Instructor` and `Student` classes to derive from `Person`, add the new class to the `DbContext`, and create a migration. (For information about how to implement the other inheritance patterns, see Mapping the Table-Per-Type (TPT) Inheritance and Mapping the Table-Per-Concrete Class (TPC) Inheritance in the MSDN Entity Framework documentation.)

Create the Person class

In the *Models* folder, create *Person.cs* and replace the template code with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public abstract class Person
    {
        public int ID { get; set; }

        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }
    }
}
```

Update Instructor and Student

Now update the *Instructor.cs* and *Student.cs* to inherit values from the *Person.sc*.

In *Instructor.cs*, derive the `Instructor` class from the `Person` class and remove the key and name fields. The code will look like the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        public virtual ICollection<Course> Courses { get; set; }
        public virtual OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Make similar changes to *Student.cs*. The `Student` class will look like the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

Add Person to the Model

In *SchoolContext.cs*, add a `DbSet` property for the `Person` entity type:

```
public DbSet<Person> People { get; set; }
```

This is all that the Entity Framework needs in order to configure table-per-hierarchy inheritance. As you'll see, when the database is updated, it will have a `Person` table in place of the `Student` and `Instructor` tables.

Create and Update Migrations

In the Package Manager Console (PMC), enter the following command:

```
Add-Migration Inheritance
```

Run the `Update-Database` command in the PMC. The command will fail at this point because we have existing data that migrations doesn't know how to handle. You get an error message like the following one:

> *Could not drop object 'dbo.Instructor' because it is referenced by a FOREIGN KEY constraint.*

Open *Migrations<timestamp>_Inheritance.cs* and replace the `Up` method with the following code:

```
public override void Up()
{
    // Drop foreign keys and indexes that point to tables we're going to drop.
    DropForeignKey("dbo.Enrollment", "StudentID", "dbo.Student");
    DropIndex("dbo.Enrollment", new[] { "StudentID" });

    RenameTable(name: "dbo.Instructor", newName: "Person");
    AddColumn("dbo.Person", "EnrollmentDate", c => c.DateTime());
    AddColumn("dbo.Person", "Discriminator", c => c.String(nullable: false, maxLength: 128, defaultValue:
"Instructor"));
    AlterColumn("dbo.Person", "HireDate", c => c.DateTime());
    AddColumn("dbo.Person", "OldId", c => c.Int(nullable: true));

    // Copy existing Student data into new Person table.
    Sql("INSERT INTO dbo.Person (LastName, FirstName, HireDate, EnrollmentDate, Discriminator, OldId) SELECT
LastName, FirstName, null AS HireDate, EnrollmentDate, 'Student' AS Discriminator, ID AS OldId FROM
dbo.Student");

    // Fix up existing relationships to match new PK's.
    Sql("UPDATE dbo.Enrollment SET StudentId = (SELECT ID FROM dbo.Person WHERE OldId = Enrollment.StudentId
AND Discriminator = 'Student')");

    // Remove temporary key
    DropColumn("dbo.Person", "OldId");

    DropTable("dbo.Student");

    // Re-create foreign keys and indexes pointing to new table.
    AddForeignKey("dbo.Enrollment", "StudentID", "dbo.Person", "ID", cascadeDelete: true);
    CreateIndex("dbo.Enrollment", "StudentID");
}
```

This code takes care of the following database update tasks:

- Removes foreign key constraints and indexes that point to the Student table.
- Renames the Instructor table as Person and makes changes needed for it to store Student data:
  - Adds nullable EnrollmentDate for students.
  - Adds Discriminator column to indicate whether a row is for a student or an instructor.
  - Makes HireDate nullable since student rows won't have hire dates.
  - Adds a temporary field that will be used to update foreign keys that point to students. When you copy students into the Person table they'll get new primary key values.
- Copies data from the Student table into the Person table. This causes students to get assigned new primary key values.
- Fixes foreign key values that point to students.
- Re-creates foreign key constraints and indexes, now pointing them to the Person table.

(If you had used GUID instead of integer as the primary key type, the student primary key values wouldn't have to change, and several of these steps could have been omitted.)

Run the `update-database` command again.

(In a production system you would make corresponding changes to the Down method in case you ever had to use that to go back to the previous database version. For this tutorial you won't be using the Down method.)

NOTE

It's possible to get other errors when migrating data and making schema changes. If you get migration errors you can't resolve, you can continue with the tutorial by changing the connection string in the *Web.config* file or by deleting the database. The simplest approach is to rename the database in the *Web.config* file. For example, change the database name to ContosoUniversity2 as shown in the following example:

```
<add name="SchoolContext"
    connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=ContosoUniversity2;Integrated
Security=SSPI;"
    providerName="System.Data.SqlClient" />
```

With a new database, there is no data to migrate, and the `update-database` command is much more likely to complete without errors. For instructions on how to delete the database, see How to Drop a Database from Visual Studio 2012. If you take this approach in order to continue with the tutorial, skip the deployment step at the end of this tutorial or deploy to a new site and database. If you deploy an update to the same site you've been deploying to already, EF will get the same error there when it runs migrations automatically. If you want to troubleshoot a migrations error, the best resource is one of the Entity Framework forums or StackOverflow.com.

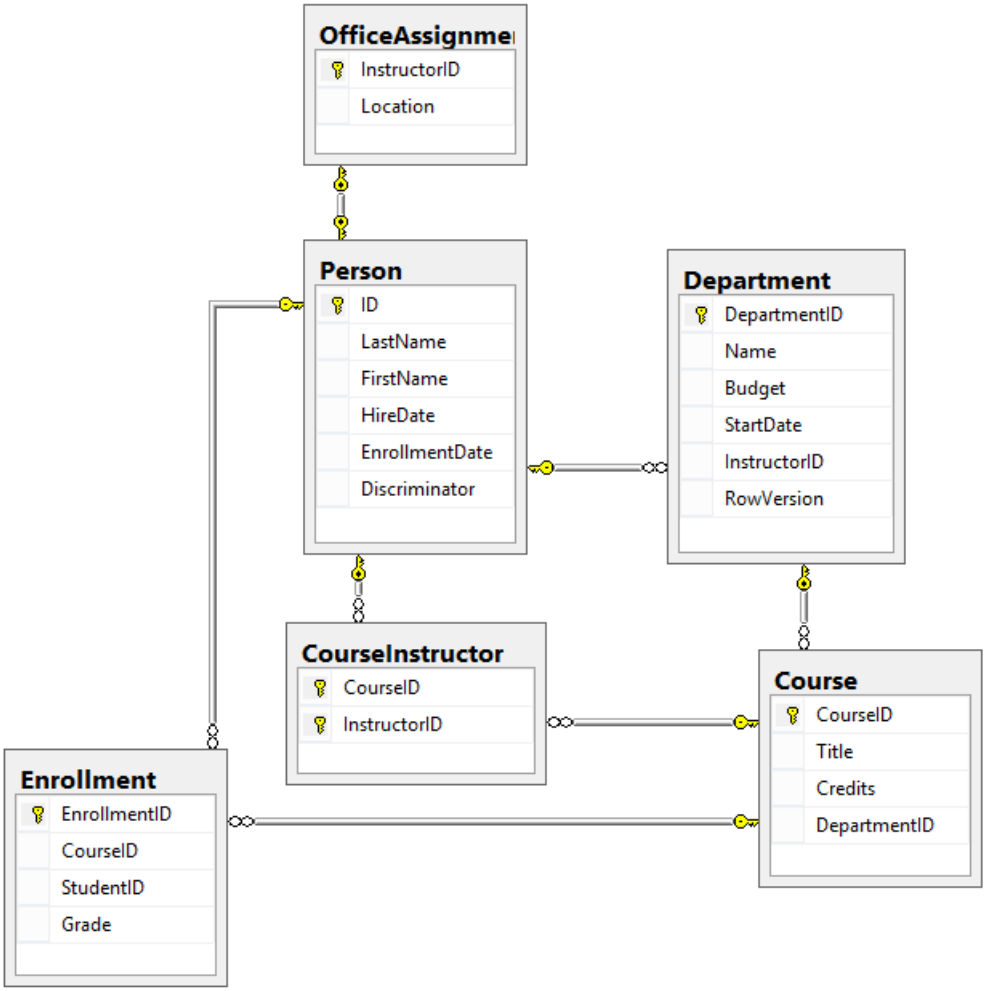
Test the implementation

Run the site and try various pages. Everything works the same as it did before.

In **Server Explorer,** expand **Data Connections\SchoolContext** and then **Tables**, and you see that the **Student** and **Instructor** tables have been replaced by a **Person** table. Expand the **Person** table and you see that it has all of the columns that used to be in the **Student** and **Instructor** tables.

Right-click the Person table, and then click **Show Table Data** to see the discriminator column.

The following diagram illustrates the structure of the new School database:



Deploy to Azure

This section requires you to have completed the optional **Deploying the app to Azure** section in Part 3, Sorting, Filtering, and Paging of this tutorial series. If you had migrations errors that you resolved by deleting the database in your local project, skip this step; or create a new site and database, and deploy to the new environment.

1. In Visual Studio, right-click the project in **Solution Explorer** and select **Publish** from the context menu.
2. Click **Publish**.

   The Web app opens in your default browser.
3. Test the application to verify it's working.

   The first time you run a page that accesses the database, the Entity Framework runs all of the migrations `Up` methods required to bring the database up to date with the current data model.

Get the code

Download Completed Project

Additional resources

Links to other Entity Framework resources can be found in the ASP.NET Data Access - Recommended Resources.

For more information about this and other inheritance structures, see TPT Inheritance Pattern and TPH Inheritance Pattern on MSDN. In the next tutorial you'll see how to handle a variety of relatively advanced Entity Framework scenarios.

Next steps

In this tutorial, you:

- Learned to map inheritance to database
- Created the Person class
- Updated Instructor and Student
- Added Person to the Model
- Created and Update Migrations
- Tested the implementation
- Deployed to Azure

Advance to the next article to learn about topics that are useful to be aware of when you go beyond the basics of developing ASP.NET web applications that use Entity Framework Code First.

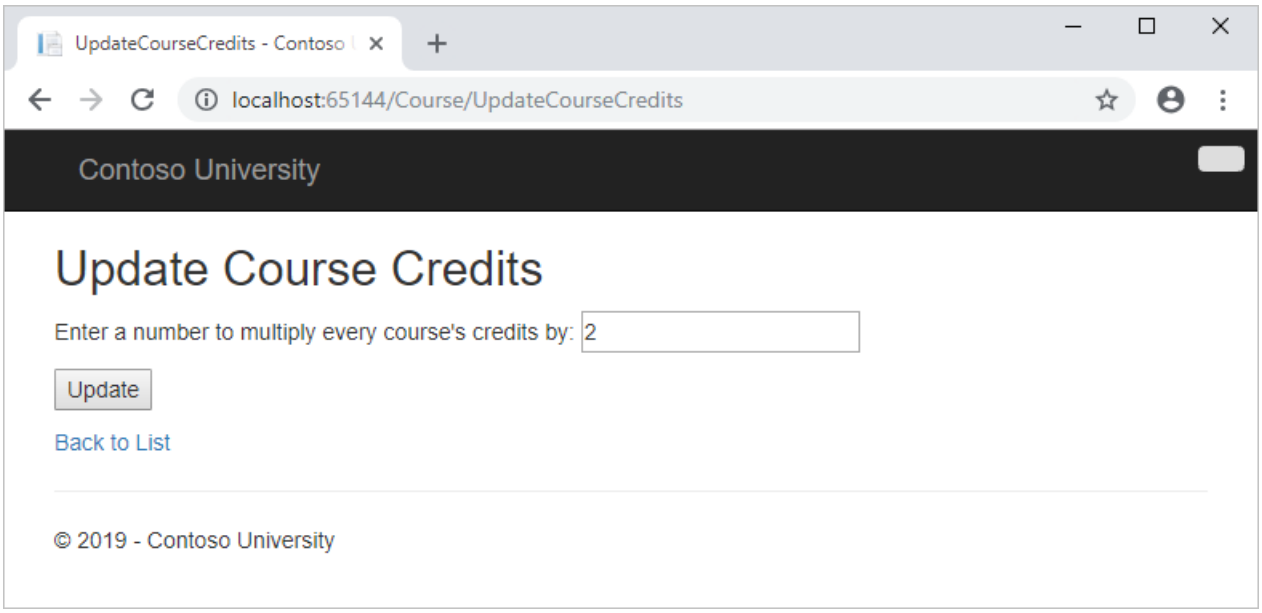
Advanced Entity Framework Scenarios

Tutorial: Learn about advanced EF Scenarios for an MVC 5 Web app

9/16/2020 • 17 minutes to read • Edit Online

In the previous tutorial you implemented table-per-hierarchy inheritance. This tutorial includes introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET web applications that use Entity Framework Code First. The first few sections have step-by-step instructions that walk you through the code and using Visual Studio to complete tasks The sections that follow introduce several topics with brief introductions followed by links to resources for more information.

For most of these topics, you'll work with pages that you already created. To use raw SQL to do bulk updates you'll create a new page that updates the number of credits of all courses in the database:



In this tutorial, you:

- Perform raw SQL queries
- Perform no-tracking queries
- Examine SQL queries sent to database

You also learn about:

- Creating an abstraction layer
- Proxy classes
- Automatic change detection
- Automatic validation
- Entity Framework Power Tools
- Entity Framework source code

Prerequisite

- Implementing Inheritance

Perform raw SQL queries

The Entity Framework Code First API includes methods that enable you to pass SQL commands directly to the database. You have the following options:

- Use the DbSet.SqlQuery method for queries that return entity types. The returned objects must be of the type expected by the `DbSet` object, and they are automatically tracked by the database context unless you turn tracking off. (See the following section about the AsNoTracking method.)
- Use the Database.SqlQuery method for queries that return types that aren't entities. The returned data isn't tracked by the database context, even if you use this method to retrieve entity types.
- Use the Database.ExecuteSqlCommand for non-query commands.

One of the advantages of using the Entity Framework is that it avoids tying your code too closely to a particular method of storing data. It does this by generating SQL queries and commands for you, which also frees you from having to write them yourself. But there are exceptional scenarios when you need to run specific SQL queries that you have manually created, and these methods make it possible for you to handle such exceptions.

As is always true when you execute SQL commands in a web application, you must take precautions to protect your site against SQL injection attacks. One way to do that is to use parameterized queries to make sure that strings submitted by a web page can't be interpreted as SQL commands. In this tutorial you'll use parameterized queries when integrating user input into a query.

Calling a Query that Returns Entities

The DbSet<TEntity> class provides a method that you can use to execute a query that returns an entity of type `TEntity`. To see how this works you'll change the code in the `Details` method of the `Department` controller.

In *DepartmentController.cs*, in the `Details` method, replace the `db.Departments.FindAsync` method call with a `db.Departments.SqlQuery` method call, as shown in the following highlighted code:

```
public async Task<ActionResult> Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }

    // Commenting out original code to show how to use a raw SQL query.
    //Department department = await db.Departments.FindAsync(id);

    // Create and execute raw SQL query.
    string query = "SELECT * FROM Department WHERE DepartmentID = @p0";
    Department department = await db.Departments.SqlQuery(query, id).SingleOrDefaultAsync();

    if (department == null)
    {
        return HttpNotFound();
    }
    return View(department);
}
```

To verify that the new code works correctly, select the **Departments** tab and then **Details** for one of the departments. Make sure all of the data displays as expected.

Calling a Query that Returns Other Types of Objects

Earlier you created a student statistics grid for the About page that showed the number of students for each enrollment date. The code that does this in *HomeController.cs* uses LINQ:

```
var data = from student in db.Students
           group student by student.EnrollmentDate into dateGroup
           select new EnrollmentDateGroup()
           {
               EnrollmentDate = dateGroup.Key,
               StudentCount = dateGroup.Count()
           };
```

Suppose you want to write the code that retrieves this data directly in SQL rather than using LINQ. To do that you need to run a query that returns something other than entity objects, which means you need to use the Database.SqlQuery method.

In *HomeController.cs*, replace the LINQ statement in the `About` method with a SQL statement, as shown in the following highlighted code:

```
public ActionResult About()
{
    // Commenting out LINQ to show how to do the same thing in SQL.
    //IQueryable<EnrollmentDateGroup> = from student in db.Students
    //           group student by student.EnrollmentDate into dateGroup
    //           select new EnrollmentDateGroup()
    //           {
    //               EnrollmentDate = dateGroup.Key,
    //               StudentCount = dateGroup.Count()
    //           };

    // SQL version of the above LINQ code.
    string query = "SELECT EnrollmentDate, COUNT(*) AS StudentCount "
        + "FROM Person "
        + "WHERE Discriminator = 'Student' "
        + "GROUP BY EnrollmentDate";
    IEnumerable<EnrollmentDateGroup> data = db.Database.SqlQuery<EnrollmentDateGroup>(query);

    return View(data.ToList());
}
```

Run the About page. Verify that it displays the same data it did before.

Calling an Update Query

Suppose Contoso University administrators want to be able to perform bulk changes in the database, such as changing the number of credits for every course. If the university has a large number of courses, it would be inefficient to retrieve them all as entities and change them individually. In this section you'll implement a web page that enables the user to specify a factor by which to change the number of credits for all courses, and you'll make the change by executing a SQL `UPDATE` statement.

In *CourseController.cs*, add `UpdateCourseCredits` methods for `HttpGet` and `HttpPost`:

```
public ActionResult UpdateCourseCredits()
{
    return View();
}

[HttpPost]
public ActionResult UpdateCourseCredits(int? multiplier)
{
    if (multiplier != null)
    {
        ViewBag.RowsAffected = db.Database.ExecuteSqlCommand("UPDATE Course SET Credits = Credits * {0}",
multiplier);
    }
    return View();
}
```

When the controller processes an `HttpGet` request, nothing is returned in the `ViewBag.RowsAffected` variable, and the view displays an empty text box and a submit button.

When the **Update** button is clicked, the `HttpPost` method is called, and `multiplier` has the value entered in the text box. The code then executes the SQL that updates courses and returns the number of affected rows to the view in the `ViewBag.RowsAffected` variable. When the view gets a value in that variable, it displays the number of rows updated instead of the text box and submit button.

In *CourseController.cs*, right-click one of the `UpdateCourseCredits` methods, and then click **Add View**. The **Add View** dialog appears. Leave the defaults and select **Add**.

In *Views\Course\UpdateCourseCredits.cshtml*, replace the template code with the following code:

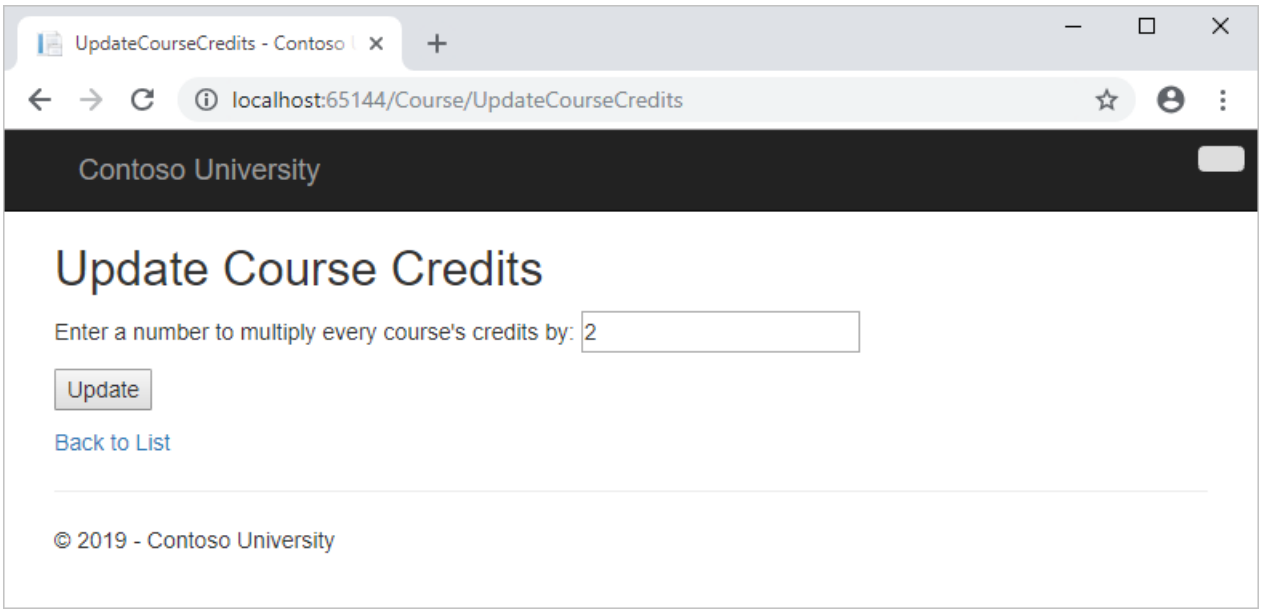
```
@model ContosoUniversity.Models.Course

@{
    ViewBag.Title = "UpdateCourseCredits";
}

<h2>Update Course Credits</h2>

@if (ViewBag.RowsAffected == null)
{
    using (Html.BeginForm())
    {
        <p>
            Enter a number to multiply every course's credits by: @Html.TextBox("multiplier")
        </p>
        <p>
            <input type="submit" value="Update" />
        </p>
    }
}
@if (ViewBag.RowsAffected != null)
{
    <p>
        Number of rows updated: @ViewBag.RowsAffected
    </p>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Run the `UpdateCourseCredits` method by selecting the **Courses** tab, then adding "/UpdateCourseCredits" to the end of the URL in the browser's address bar (for example: `http://localhost:50205/Course/UpdateCourseCredits`). Enter a number in the text box:



Click **Update**. You see the number of rows affected.

Click **Back to List** to see the list of courses with the revised number of credits.

For more information about raw SQL queries, see Raw SQL Queries on MSDN.

No-tracking queries

When a database context retrieves table rows and creates entity objects that represent them, by default it keeps track of whether the entities in memory are in sync with what's in the database. The data in memory acts as a cache and is used when you update an entity. This caching is often unnecessary in a web application because context instances are typically short-lived (a new one is created and disposed for each request) and the context that reads an entity is typically disposed before that entity is used again.

You can disable tracking of entity objects in memory by using the AsNoTracking method. Typical scenarios in which you might want to do that include the following:

- A query retrieves such a large volume of data that turning off tracking might noticeably enhance performance.
- You want to attach an entity in order to update it, but you earlier retrieved the same entity for a different purpose. Because the entity is already being tracked by the database context, you can't attach the entity that you want to change. One way to handle this situation is to use the `AsNoTracking` option with the earlier query.

For an example that demonstrates how to use the AsNoTracking method, see the earlier version of this tutorial. This version of the tutorial doesn't set the Modified flag on a model-binder-created entity in the Edit method, so it doesn't need `AsNoTracking`.

Examine SQL sent to database

Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. In an earlier tutorial you saw how to do that in interceptor code; now you'll see some ways to do it without writing interceptor code. To try this out, you'll look at a simple query and then look at what happens to it as you add options such eager loading, filtering, and sorting.

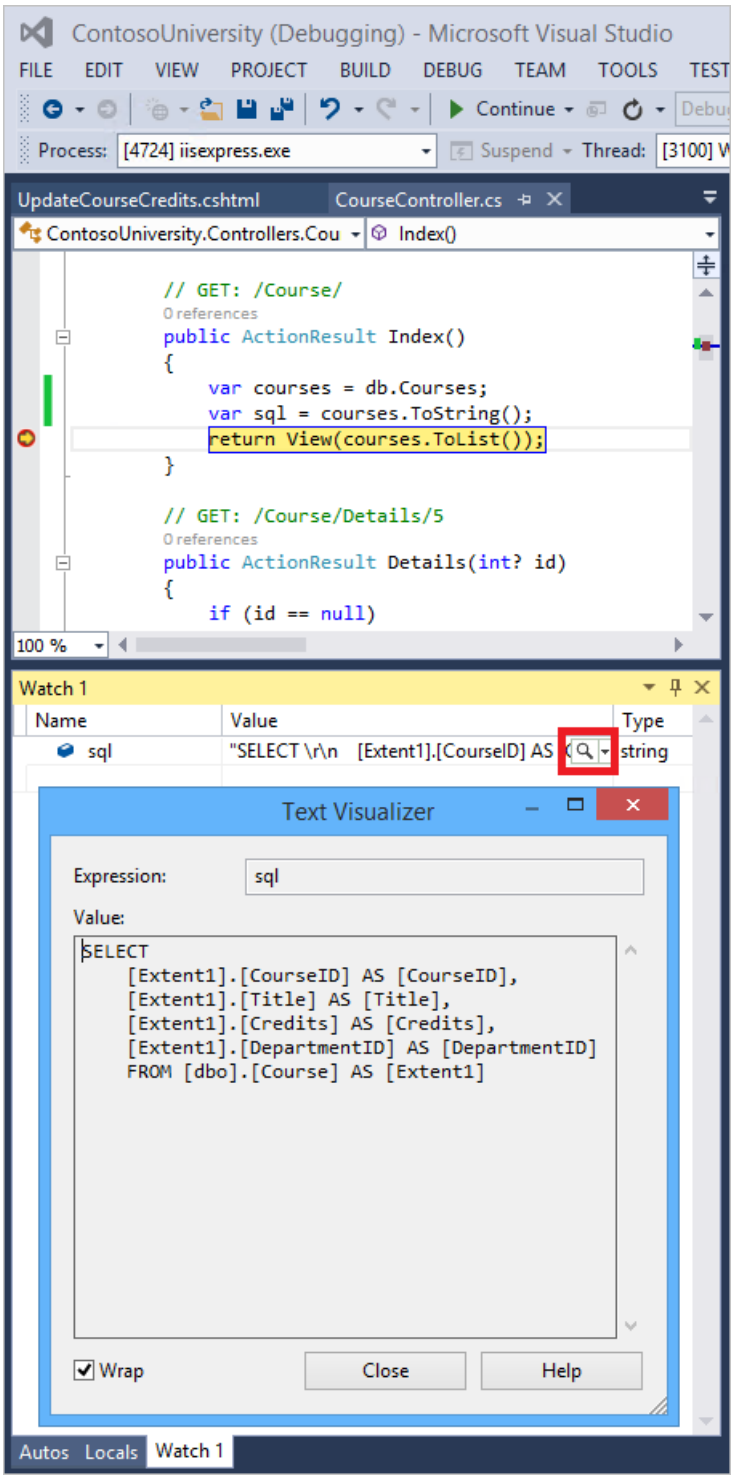
In *Controllers/CourseController*, replace the `Index` method with the following code, in order to temporarily stop eager loading:

```
public ActionResult Index()
{
    var courses = db.Courses;
    var sql = courses.ToString();
    return View(courses.ToList());
}
```

Now set a breakpoint on the `return` statement (F9 with the cursor on that line). Press **F5** to run the project in debug mode, and select the Course Index page. When the code reaches the breakpoint, examine the `sql` variable. You see the query that's sent to SQL Server. It's a simple `Select` statement.

```
{SELECT
[Extent1].[CourseID] AS [CourseID],
[Extent1].[Title] AS [Title],
[Extent1].[Credits] AS [Credits],
[Extent1].[DepartmentID] AS [DepartmentID]
FROM [Course] AS [Extent1]}
```

Click the magnifying glass to see the query in the **Text Visualizer**.



Now you'll add a drop-down list to the Courses Index page so that users can filter for a particular department. You'll sort the courses by title, and you'll specify eager loading for the `Department` navigation property.

In *CourseController.cs*, replace the `Index` method with the following code:

```
public ActionResult Index(int? SelectedDepartment)
{
    var departments = db.Departments.OrderBy(q => q.Name).ToList();
    ViewBag.SelectedDepartment = new SelectList(departments, "DepartmentID", "Name", SelectedDepartment);
    int departmentID = SelectedDepartment.GetValueOrDefault();

    IQueryable<Course> courses = db.Courses
        .Where(c => !SelectedDepartment.HasValue || c.DepartmentID == departmentID)
        .OrderBy(d => d.CourseID)
        .Include(d => d.Department);
    var sql = courses.ToString();
    return View(courses.ToList());
}
```

Restore the breakpoint on the `return` statement.

The method receives the selected value of the drop-down list in the `SelectedDepartment` parameter. If nothing is selected, this parameter will be null.

A `SelectList` collection containing all departments is passed to the view for the drop-down list. The parameters passed to the `SelectList` constructor specify the value field name, the text field name, and the selected item.

For the `Get` method of the `Course` repository, the code specifies a filter expression, a sort order, and eager loading for the `Department` navigation property. The filter expression always returns `true` if nothing is selected in the drop-down list (that is, `SelectedDepartment` is null).

In *Views\Course\Index.cshtml*, immediately before the opening `table` tag, add the following code to create the drop-down list and a submit button:

```
@using (Html.BeginForm())
{
    <p>Select Department: @Html.DropDownList("SelectedDepartment","All")
    <input type="submit" value="Filter" /></p>
}
```

With the breakpoint still set, run the Course Index page. Continue through the first times that the code hits a breakpoint, so that the page is displayed in the browser. Select a department from the drop-down list and click **Filter**.

This time the first breakpoint will be for the departments query for the drop-down list. Skip that and view the `query` variable the next time the code reaches the breakpoint in order to see what the `Course` query now looks like. You'll see something like the following:

```
SELECT
    [Project1].[CourseID] AS [CourseID],
    [Project1].[Title] AS [Title],
    [Project1].[Credits] AS [Credits],
    [Project1].[DepartmentID] AS [DepartmentID],
    [Project1].[DepartmentID1] AS [DepartmentID1],
    [Project1].[Name] AS [Name],
    [Project1].[Budget] AS [Budget],
    [Project1].[StartDate] AS [StartDate],
    [Project1].[InstructorID] AS [InstructorID],
    [Project1].[RowVersion] AS [RowVersion]
    FROM ( SELECT
        [Extent1].[CourseID] AS [CourseID],
        [Extent1].[Title] AS [Title],
        [Extent1].[Credits] AS [Credits],
        [Extent1].[DepartmentID] AS [DepartmentID],
        [Extent2].[DepartmentID] AS [DepartmentID1],
        [Extent2].[Name] AS [Name],
        [Extent2].[Budget] AS [Budget],
        [Extent2].[StartDate] AS [StartDate],
        [Extent2].[InstructorID] AS [InstructorID],
        [Extent2].[RowVersion] AS [RowVersion]
        FROM  [dbo].[Course] AS [Extent1]
        INNER JOIN [dbo].[Department] AS [Extent2] ON [Extent1].[DepartmentID] = [Extent2].[DepartmentID]
        WHERE @p__linq__0 IS NULL OR [Extent1].[DepartmentID] = @p__linq__1
    )  AS [Project1]
    ORDER BY [Project1].[CourseID] ASC
```

You can see that the query is now a `JOIN` query that loads `Department` data along with the `Course` data, and that it includes a `WHERE` clause.

Remove the `var sql = courses.ToString()` line.

Create an abstraction layer

Many developers write code to implement the repository and unit of work patterns as a wrapper around code that works with the Entity Framework. These patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD). However, writing additional code to implement these patterns is not always the best choice for applications that use EF, for several reasons:

- The EF context class itself insulates your code from data-store-specific code.
- The EF context class can act as a unit-of-work class for database updates that you do using EF.
- Features introduced in Entity Framework 6 make it easier to implement TDD without writing repository code.

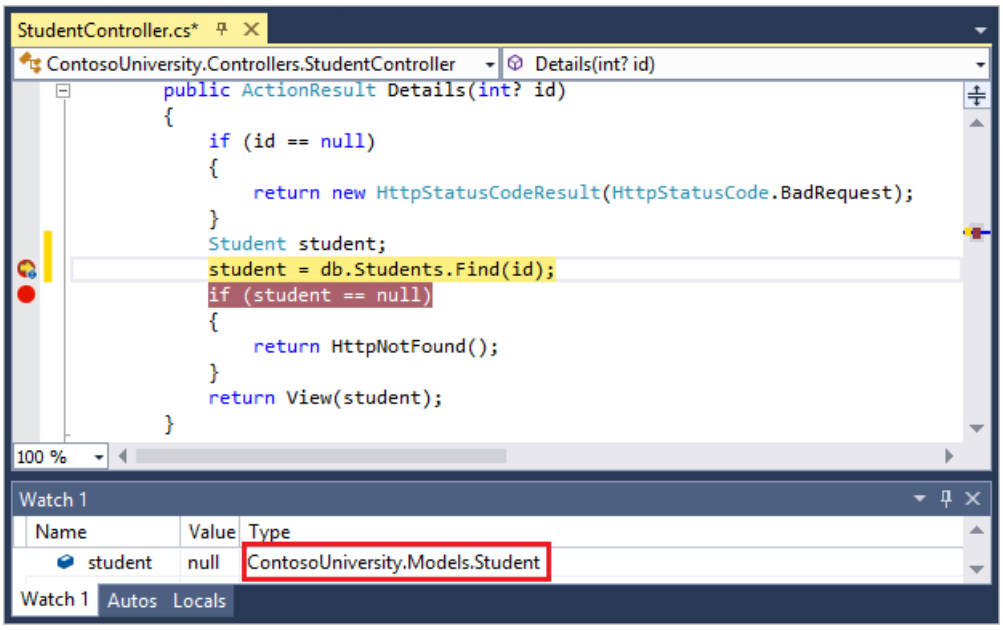
For more information about how to implement the repository and unit of work patterns, see the Entity Framework 5 version of this tutorial series. For information about ways to implement TDD in Entity Framework 6, see the following resources:

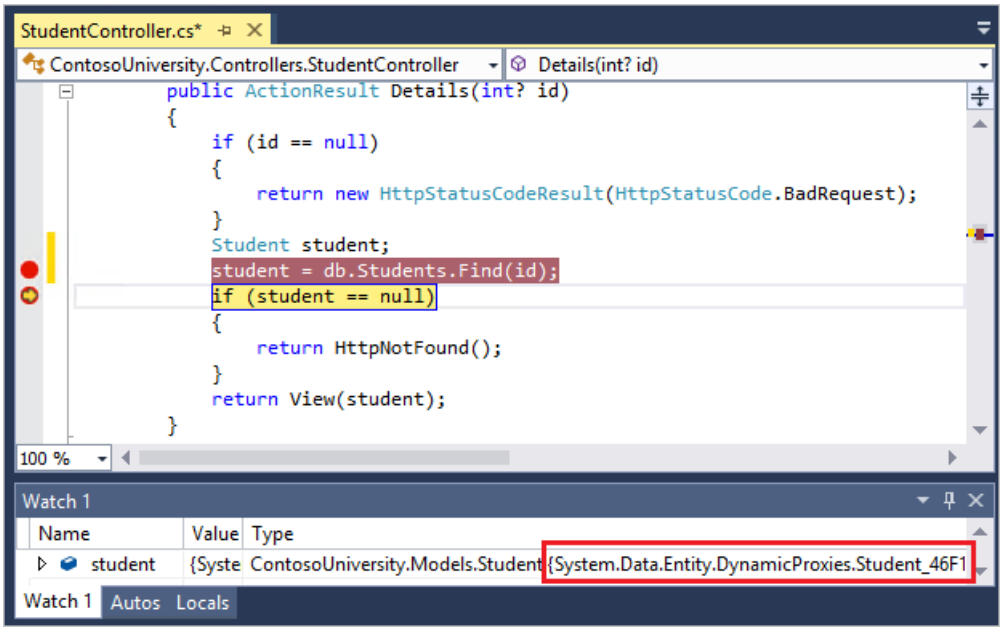
- How EF6 Enables Mocking DbSets more easily
- Testing with a mocking framework
- Testing with your own test doubles

Proxy classes

When the Entity Framework creates entity instances (for example, when you execute a query), it often creates them as instances of a dynamically generated derived type that acts as a proxy for the entity. For example, see the following two debugger images. In the first image, you see that the `student` variable is the expected `Student`

type immediately after you instantiate the entity. In the second image, after EF has been used to read a student entity from the database, you see the proxy class.





This proxy class overrides some virtual properties of the entity to insert hooks for performing actions automatically when the property is accessed. One function this mechanism is used for is lazy loading.

Most of the time you don't need to be aware of this use of proxies, but there are exceptions:

- In some scenarios you might want to prevent the Entity Framework from creating proxy instances. For example, when you're serializing entities you generally want the POCO classes, not the proxy classes. One way to avoid serialization problems is to serialize data transfer objects (DTOs) instead of entity objects, as shown in the Using Web API with Entity Framework tutorial. Another way is to disable proxy creation.
- When you instantiate an entity class using the `new` operator, you don't get a proxy instance. This means you don't get functionality such as lazy loading and automatic change tracking. This is typically okay; you generally don't need lazy loading, because you're creating a new entity that isn't in the database, and you generally don't need change tracking if you're explicitly marking the entity as `Added`. However, if you do need lazy loading and you need change tracking, you can create new entity instances with proxies using the Create method of the `DbSet` class.
- You might want to get an actual entity type from a proxy type. You can use the GetObjectType method of the `ObjectContext` class to get the actual entity type of a proxy type instance.

For more information, see Working with Proxies on MSDN.

Automatic change detection

The Entity Framework determines how an entity has changed (and therefore which updates need to be sent to the database) by comparing the current values of an entity with the original values. The original values are stored when the entity is queried or attached. Some of the methods that cause automatic change detection are the following:

- `DbSet.Find`
- `DbSet.Local`
- `DbSet.Remove`
- `DbSet.Add`
- `DbSet.Attach`
- `DbContext.SaveChanges`
- `DbContext.GetValidationErrors`
- `DbContext.Entry`
- `DbChangeTracker.Entries`

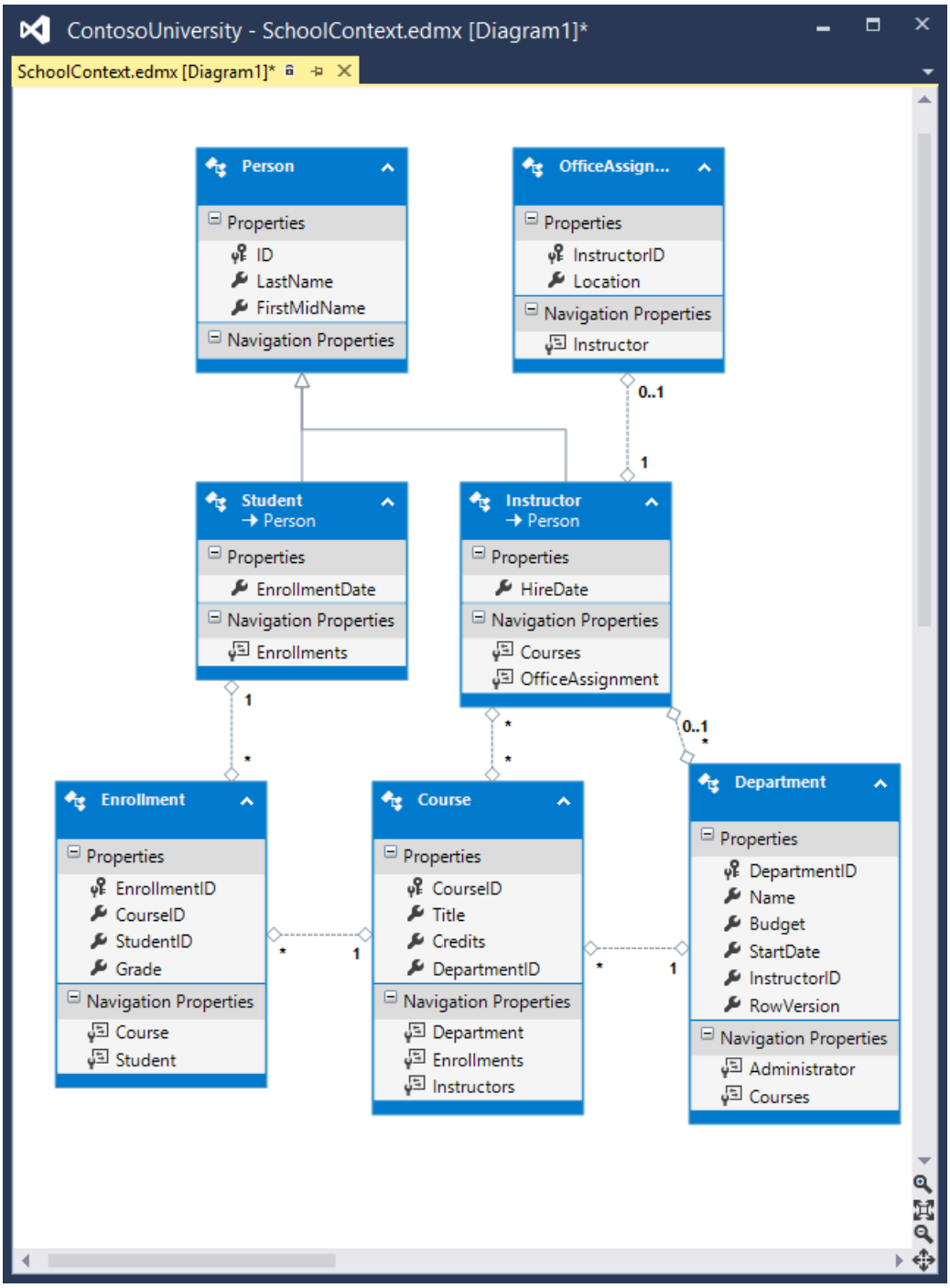
If you're tracking a large number of entities and you call one of these methods many times in a loop, you might get significant performance improvements by temporarily turning off automatic change detection using the AutoDetectChangesEnabled property. For more information, see Automatically Detecting Changes on MSDN.

Automatic validation

When you call the `SaveChanges` method, by default the Entity Framework validates the data in all properties of all changed entities before updating the database. If you've updated a large number of entities and you've already validated the data, this work is unnecessary and you could make the process of saving the changes take less time by temporarily turning off validation. You can do that using the ValidateOnSaveEnabled property. For more information, see Validation on MSDN.

Entity Framework Power Tools

Entity Framework Power Tools is a Visual Studio add-in that was used to create the data model diagrams shown in these tutorials. The tools can also do other function such as generate entity classes based on the tables in an existing database so that you can use the database with Code First. After you install the tools, some additional options appear in context menus. For example, when you right-click your context class in **Solution Explorer**, you see and **Entity Framework** option. This gives you the ability to generate a diagram. When you're using Code First you can't change the data model in the diagram, but you can move things around to make it easier to understand.



Entity Framework source code

The source code for Entity Framework 6 is available at GitHub. You can file bugs, and you can contribute your own enhancements to the EF source code.

Although the source code is open, Entity Framework is fully supported as a Microsoft product. The Microsoft Entity Framework team keeps control over which contributions are accepted and tests all code changes to ensure the quality of each release.

Acknowledgments

- Tom Dykstra wrote the original version of this tutorial, co-authored the EF 5 update, and wrote the EF 6 update. Tom is a senior programming writer on the Microsoft Web Platform and Tools Content Team.
- Rick Anderson (twitter @RickAndMSFT) did most of the work updating the tutorial for EF 5 and MVC 4 and co-authored the EF 6 update. Rick is a senior programming writer for Microsoft focusing on Azure and MVC.
- Rowan Miller and other members of the Entity Framework team assisted with code reviews and helped debug

many issues with migrations that arose while we were updating the tutorial for EF 5 and EF 6.

Troubleshoot common errors

Cannot create/shadow copy

Error Message:

> Cannot create/shadow copy '<filename>' when that file already exists.

Solution

Wait a few seconds and refresh the page.

Update-Database not recognized

Error Message (from the `Update-Database` command in the PMC):

> The term 'Update-Database' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.

Solution

Exit Visual Studio. Reopen project and try again.

Validation failed

Error Message (from the `Update-Database` command in the PMC):

> Validation failed for one or more entities. See 'EntityValidationErrors' property for more details.

Solution

One cause of this problem is validation errors when the `Seed` method runs. See Seeding and Debugging Entity Framework (EF) DBs for tips on debugging the `Seed` method.

HTTP 500.19 error

Error Message:

> HTTP Error 500.19 - Internal Server Error The requested page cannot be accessed because the related configuration data for the page is invalid.

Solution

One way you can get this error is from having multiple copies of the solution, each of them using the same port number. You can usually solve this problem by exiting all instances of Visual Studio, then restarting the project you're working on. If that doesn't work, try changing the port number. Right click on the project file and then click properties. Select the **Web** tab and then change the port number in the **Project Url** text box.

Error locating SQL Server instance

Error Message:

> A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)

Solution

Check the connection string. If you have manually deleted the database, change the name of the database in the construction string.

Get the code

Download Completed Project

Additional resources

For more information about how to work with data using the Entity Framework, see the EF documentation page on MSDN and ASP.NET Data Access - Recommended Resources.

For more information about how to deploy your web application after you've built it, see ASP.NET Web Deployment - Recommended Resources in the MSDN Library.

For information about other topics related to MVC, such as authentication and authorization, see the ASP.NET MVC - Recommended Resources.

Next steps

In this tutorial, you:

- Performed raw SQL queries
- Performed no-tracking queries
- Examined SQL queries sent to the database

You also learned about:

- Creating an abstraction layer
- Proxy classes
- Automatic change detection
- Automatic validation
- Entity Framework Power Tools
- Entity Framework source code

This completes this series of tutorials on using the Entity Framework in an ASP.NET MVC application. If you want to learn about EF Database First, see the DB First tutorial series.

Entity Framework Database First

Getting Started with Entity Framework 6 Database First Using MVC 5

2/20/2020 • 2 minutes to read • Edit Online

This series shows how to use Database First development for creating an MVC 5 application with Entity Framework

- Getting Started with Entity Framework 6 Database First using MVC 5
- Creating the Web Application and Data Models
- Generating Views
- Changing the Database
- Customizing a View
- Enhancing Data Validation

Tutorial: Get started with EF Database First using MVC 5

2/20/2020 • 6 minutes to read • Edit Online

Using MVC, Entity Framework, and ASP.NET Scaffolding, you can create a web application that provides an interface to an existing database. This tutorial series shows you how to automatically generate code that enables users to display, edit, create, and delete data that resides in a database table. The generated code corresponds to the columns in the database table. In the last part of the series, you learn how add data annotations to the data model to specify validation requirements and display formatting. When you're done, you can advance to an Azure article to learn how to deploy a .NET app and SQL database to Azure App Service.

This tutorial shows how to start with an existing database and quickly create a web application that enables users to interact with the data. It uses the Entity Framework 6 and MVC 5 to build the web application. The ASP.NET Scaffolding feature enables you to automatically generate code for displaying, updating, creating and deleting data. Using the publishing tools within Visual Studio, you can easily deploy the site and database to Azure.

This part of the series focuses on creating a database and populating it with data.

This series was written with contributions from Tom Dykstra and Rick Anderson. It was improved based on feedback from users in the comments section.

In this tutorial, you:

- Set up the database

Prerequisites

Visual Studio 2017

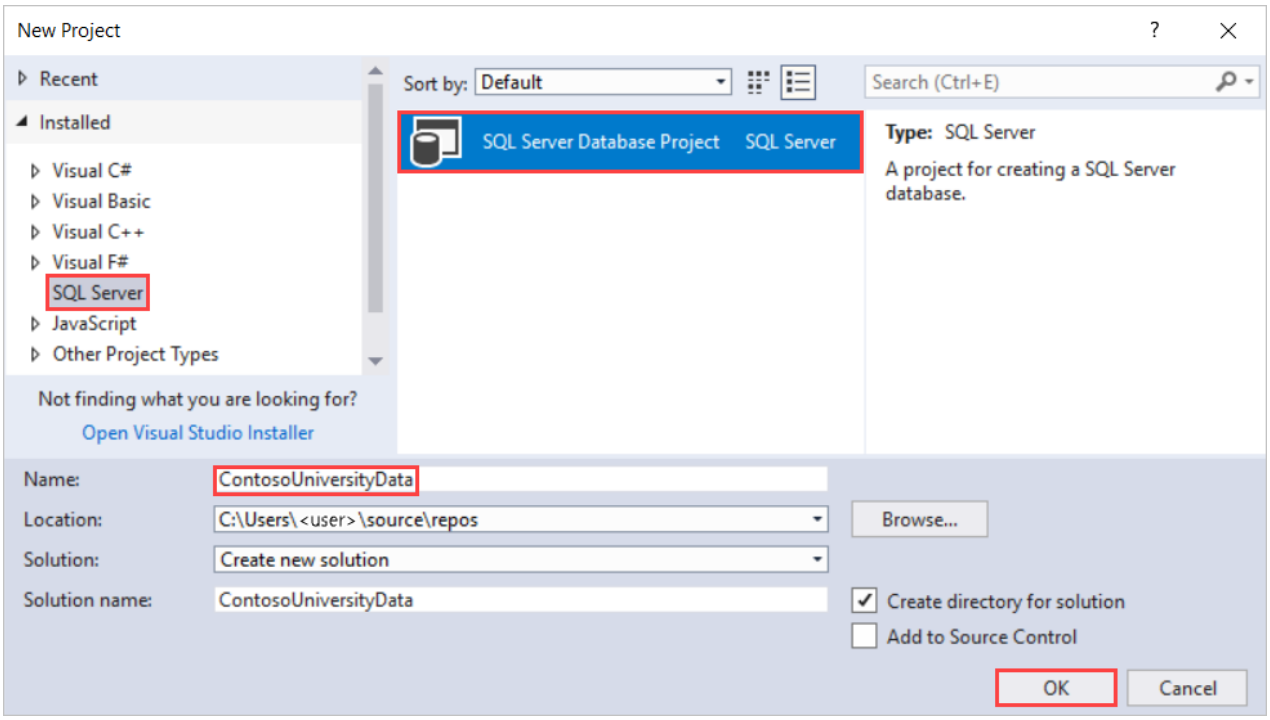
Set up the database

To mimic the environment of having an existing database, you will first create a database with some pre-filled data, and then create your web application that connects to the database.

This tutorial was developed using LocalDB with Visual Studio 2017. You can use an existing database server instead of LocalDB, but depending on your version of Visual Studio and your type of database, all of the data tools in Visual Studio might not be supported. If the tools are not available for your database, you may need to perform some of the database-specific steps within the management suite for your database.

If you have a problem with the database tools in your version of Visual Studio, make sure you have installed the latest version of the database tools. For information about updating or installing the database tools, see Microsoft SQL Server Data Tools.

Launch Visual Studio and create a **SQL Server Database Project**. Name the project **ContosoUniversityData**.



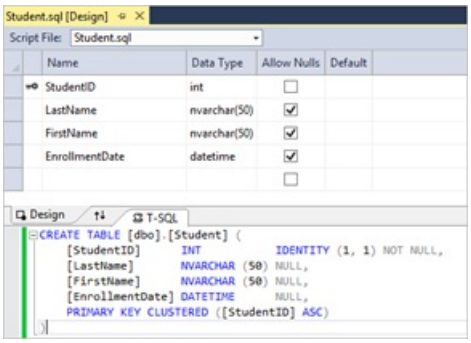
You now have an empty database project. To make sure that you can deploy this database to Azure, you'll set Azure SQL Database as the target platform for the project. Setting the target platform does not actually deploy the database; it only means that the database project will verify that the database design is compatible with the target platform. To set the target platform, open the **Properties** for the project and select **Microsoft Azure SQL Database** for the target platform.

You can create the tables needed for this tutorial by adding SQL scripts that define the tables. Right-click your project and add a new item. Select **Tables and Views** > **Table** and name it *Student*.

In the table file, replace the T-SQL command with the following code to create the table.

```
CREATE TABLE [dbo].[Student] (
    [StudentID]      INT           IDENTITY (1, 1) NOT NULL,
    [LastName]       NVARCHAR (50) NULL,
    [FirstName]      NVARCHAR (50) NULL,
    [EnrollmentDate] DATETIME      NULL,
    PRIMARY KEY CLUSTERED ([StudentID] ASC)
)
```

Notice that the design window automatically synchronizes with the code. You can work with either the code or designer.



Add another table. This time name it Course and use the following T-SQL command.

```
CREATE TABLE [dbo].[Course] (
    [CourseID] INT           IDENTITY (1, 1) NOT NULL,
    [Title]    NVARCHAR (50) NULL,
    [Credits]  INT           NULL,
    PRIMARY KEY CLUSTERED ([CourseID] ASC)
)
```

And, repeat one more time to create a table named Enrollment.

```
CREATE TABLE [dbo].[Enrollment] (
    [EnrollmentID] INT IDENTITY (1, 1) NOT NULL,
    [Grade]        DECIMAL(3, 2) NULL,
    [CourseID]     INT NOT NULL,
    [StudentID]    INT NOT NULL,
    PRIMARY KEY CLUSTERED ([EnrollmentID] ASC),
    CONSTRAINT [FK_dbo.Enrollment_dbo.Course_CourseID] FOREIGN KEY ([CourseID])
        REFERENCES [dbo].[Course] ([CourseID]) ON DELETE CASCADE,
    CONSTRAINT [FK_dbo.Enrollment_dbo.Student_StudentID] FOREIGN KEY ([StudentID])
        REFERENCES [dbo].[Student] ([StudentID]) ON DELETE CASCADE
)
```

You can populate your database with data through a script that is run after the database is deployed. Add a Post-Deployment Script to the project. Right-click your project and add a new item. Select **User Scripts** > **Post-Deployment Script**. You can use the default name.

Add the following T-SQL code to the post-deployment script. This script simply adds data to the database when no matching record is found. It does not overwrite or delete any data you may have entered into the database.

```
MERGE INTO Course AS Target
USING (VALUES
        (1, 'Economics', 3),
        (2, 'Literature', 3),
        (3, 'Chemistry', 4)
)
AS Source (CourseID, Title, Credits)
ON Target.CourseID = Source.CourseID
WHEN NOT MATCHED BY TARGET THEN
INSERT (Title, Credits)
VALUES (Title, Credits);

MERGE INTO Student AS Target
USING (VALUES
        (1, 'Tibbetts', 'Donnie', '2013-09-01'),
        (2, 'Guzman', 'Liza', '2012-01-13'),
(3, 'Catlett', 'Phil', '2011-09-03')
)
AS Source (StudentID, LastName, FirstName, EnrollmentDate)
ON Target.StudentID = Source.StudentID
WHEN NOT MATCHED BY TARGET THEN
INSERT (LastName, FirstName, EnrollmentDate)
VALUES (LastName, FirstName, EnrollmentDate);

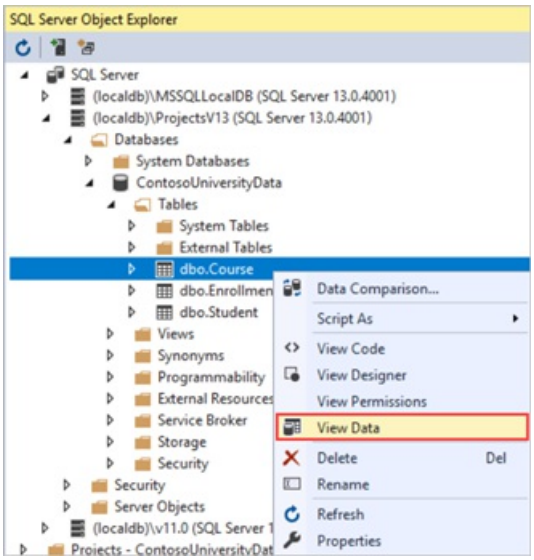
MERGE INTO Enrollment AS Target
USING (VALUES
(1, 2.00, 1, 1),
(2, 3.50, 1, 2),
(3, 4.00, 2, 3),
(4, 1.80, 2, 1),
(5, 3.20, 3, 1),
(6, 4.00, 3, 2)
)
AS Source (EnrollmentID, Grade, CourseID, StudentID)
ON Target.EnrollmentID = Source.EnrollmentID
WHEN NOT MATCHED BY TARGET THEN
INSERT (Grade, CourseID, StudentID)
VALUES (Grade, CourseID, StudentID);
```

It is important to note that the post-deployment script is run every time you deploy your database project. Therefore, you need to carefully consider your requirements when writing this script. In some cases, you may wish to start over from a known set of data every time the project is deployed. In other cases, you may not want to alter the existing data in any way. Based on your requirements, you can decide whether you need a post-deployment script or what you need to include in the script. For more information about populating your database with a post-deployment script, see Including Data in a SQL Server Database Project.

You now have 4 SQL script files but no actual tables. You are ready to deploy your database project to localdb. In Visual Studio, click the Start button (or F5) to build and deploy your database project. Check the **Output** tab to verify that the build and deployment succeeded.

To see that the new database has been created, open **SQL Server Object Explorer** and look for the name of the project in the correct local database server (in this case **(localdb)\ProjectsV13**).

To see that the tables are populated with data, right-click a table, and select **View Data**.



An editable view of the table data is displayed. For example, if you select **Tables** > **dbo.course** > **View Data**, you see a table with three columns (**Course**, **Title**, and **Credits**) and four rows.

Additional resources

For an introductory example of Code First development, see Getting Started with ASP.NET MVC 5. For a more advanced example, see Creating an Entity Framework Data Model for an ASP.NET MVC 4 App.

For guidance on selecting which Entity Framework approach to use, see Entity Framework Development Approaches.

Next steps

In this tutorial, you:

- Set up the database

Advance to the next tutorial to learn how to create the web application and data models.

Create the web application and data models

Tutorial: Create the Web Application and Data Models for EF Database First with ASP.NET MVC

2/20/2020 • 3 minutes to read • Edit Online

Using MVC, Entity Framework, and ASP.NET Scaffolding, you can create a web application that provides an interface to an existing database. This tutorial series shows you how to automatically generate code that enables users to display, edit, create, and delete data that resides in a database table. The generated code corresponds to the columns in the database table.

This tutorial focuses on creating the web application, and generating the data models based on your database tables.

In this tutorial, you:

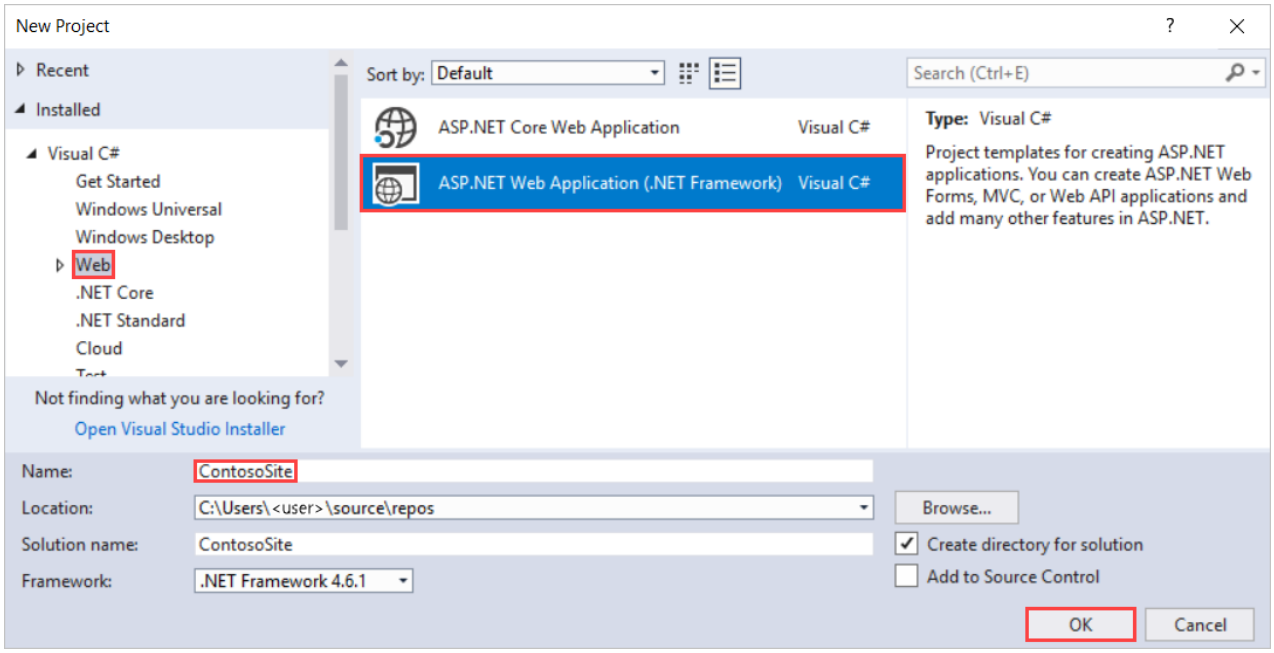
- Create an ASP.NET web app
- Generate the models

Prerequisites

- Getting started with Entity Framework 6 Database First using MVC 5

Create an ASP.NET web app

In either a new solution or the same solution as the database project, create a new project in Visual Studio and select the **ASP.NET Web Application** template. Name the project **ContosoSite**.



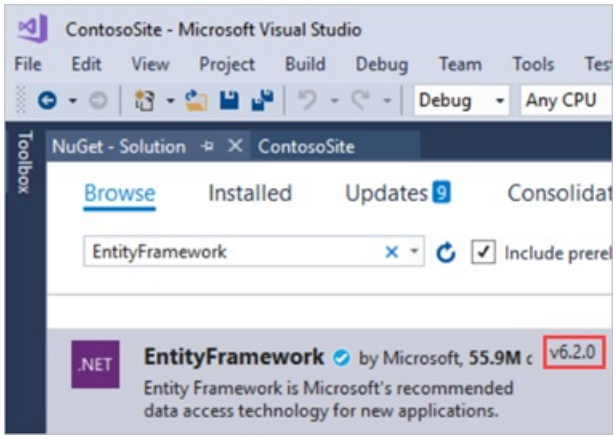
Click **OK**.

In the New ASP.NET Project window, select the **MVC** template. You can clear the **Host in the cloud** option for now because you will deploy the application to the cloud later. Click **OK** to create the application.

The project is created with the default files and folders.

In this tutorial, you will use Entity Framework 6. You can double-check the version of Entity Framework in your

project through the Manage NuGet Packages window. If necessary, update your version of Entity Framework.



Generate the models

You will now create Entity Framework models from the database tables. These models are classes that you will use to work with the data. Each model mirrors a table in the database and contains properties that correspond to the columns in the table.

Right-click the **Models** folder, and select **Add** and **New Item**.

In the Add New Item window, select **Data** in the left pane and **ADO.NET Entity Data Model** from the options in the center pane. Name the new model file **ContosoModel**.

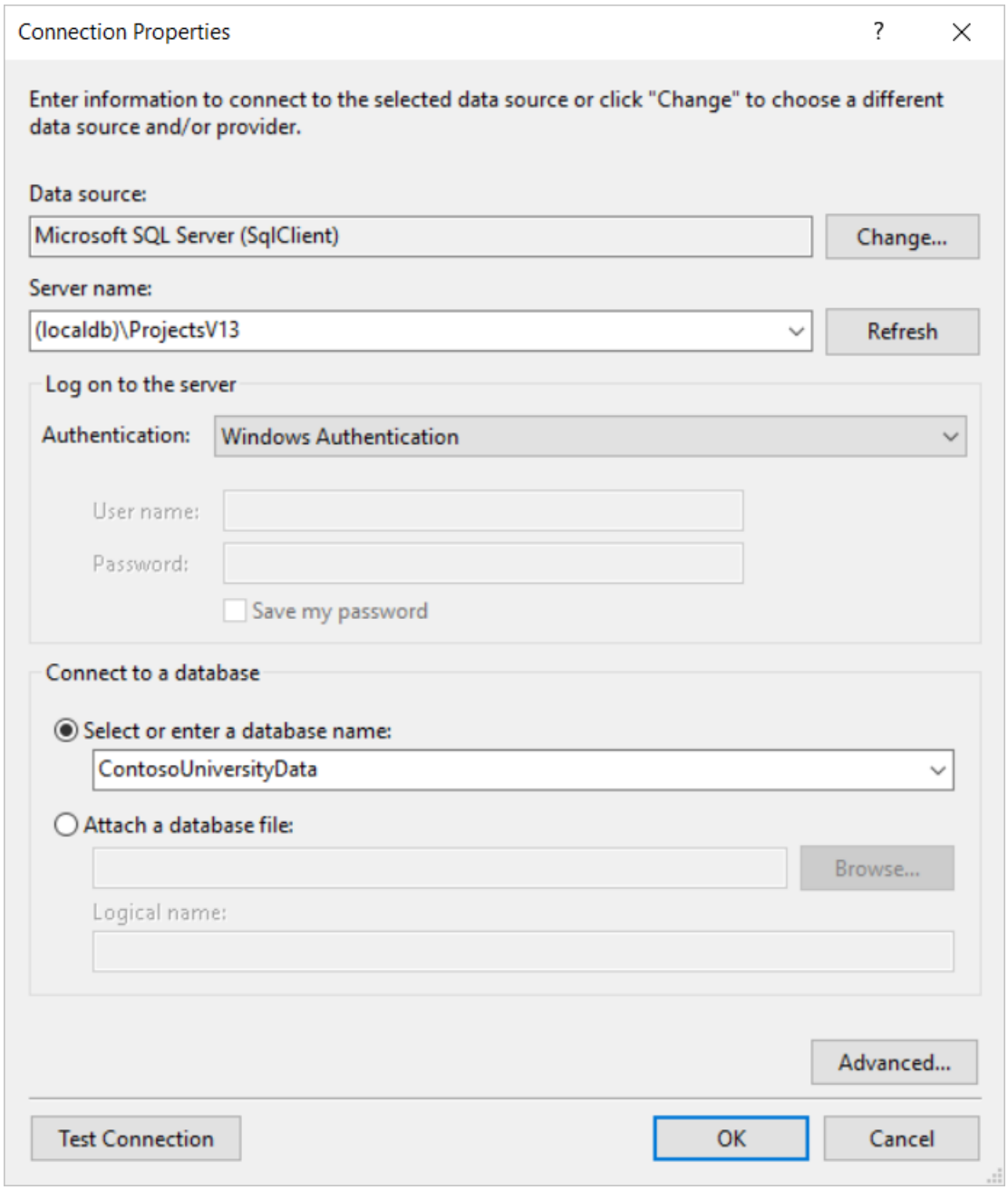
Click **Add**.

In the Entity Data Model Wizard, select **EF Designer from database**.

Click **Next**.

If you have database connections defined within your development environment, you may see one of these connections pre-selected. However, you want to create a new connection to the database you created in the first part of this tutorial. Click the **New Connection** button.

In the Connection Properties window, provide the name of the local server where your database was created (in this case **(localdb)\ProjectsV13**). After providing the server name, select the ContosoUniversityData from the available databases.



Click **OK**.

The correct connection properties are now displayed. You can use the default name for connection in the Web.Config file.

Click **Next**.

Select the latest version of Entity Framework.

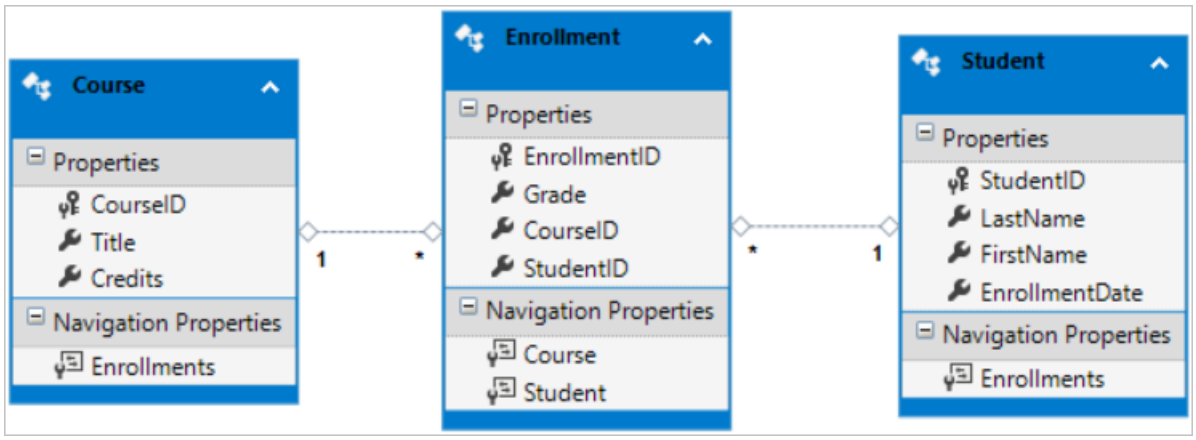
Click **Next**.

Select **Tables** to generate models for all three tables.

Click **Finish**.

If you receive a security warning, select **OK** to continue running the template.

The models are generated from the database tables, and a diagram is displayed that shows the properties and relationships between the tables.



The Models folder now includes many new files related to the models that were generated from the database.

The **ContosoModel.Context.cs** file contains a class that derives from the **DbContext** class, and provides a property for each model class that corresponds to a database table. The **Course.cs**, **Enrollment.cs**, and **Student.cs** files contain the model classes that represent the databases tables. You will use both the context class and the model classes when working with scaffolding.

Before proceeding with this tutorial, build the project. In the next section, you will generate code based on the data models, but that section will not work if the project has not been built.

Next steps

In this tutorial, you:

- Created an ASP.NET web app
- Generated the models

Advance to the next tutorial to learn how to create generate code based on the data models.

Generating views

Tutorial: Generate views for EF Database First with ASP.NET MVC app

2/20/2020 • 4 minutes to read • Edit Online

Using MVC, Entity Framework, and ASP.NET Scaffolding, you can create a web application that provides an interface to an existing database. This tutorial series shows you how to automatically generate code that enables users to display, edit, create, and delete data that resides in a database table. The generated code corresponds to the columns in the database table.

This tutorial focuses on using ASP.NET Scaffolding to generate the controllers and views.

In this tutorial, you:

- Add scaffold
- Add links to new views
- Display student views
- Display enrollment views

Prerequisite

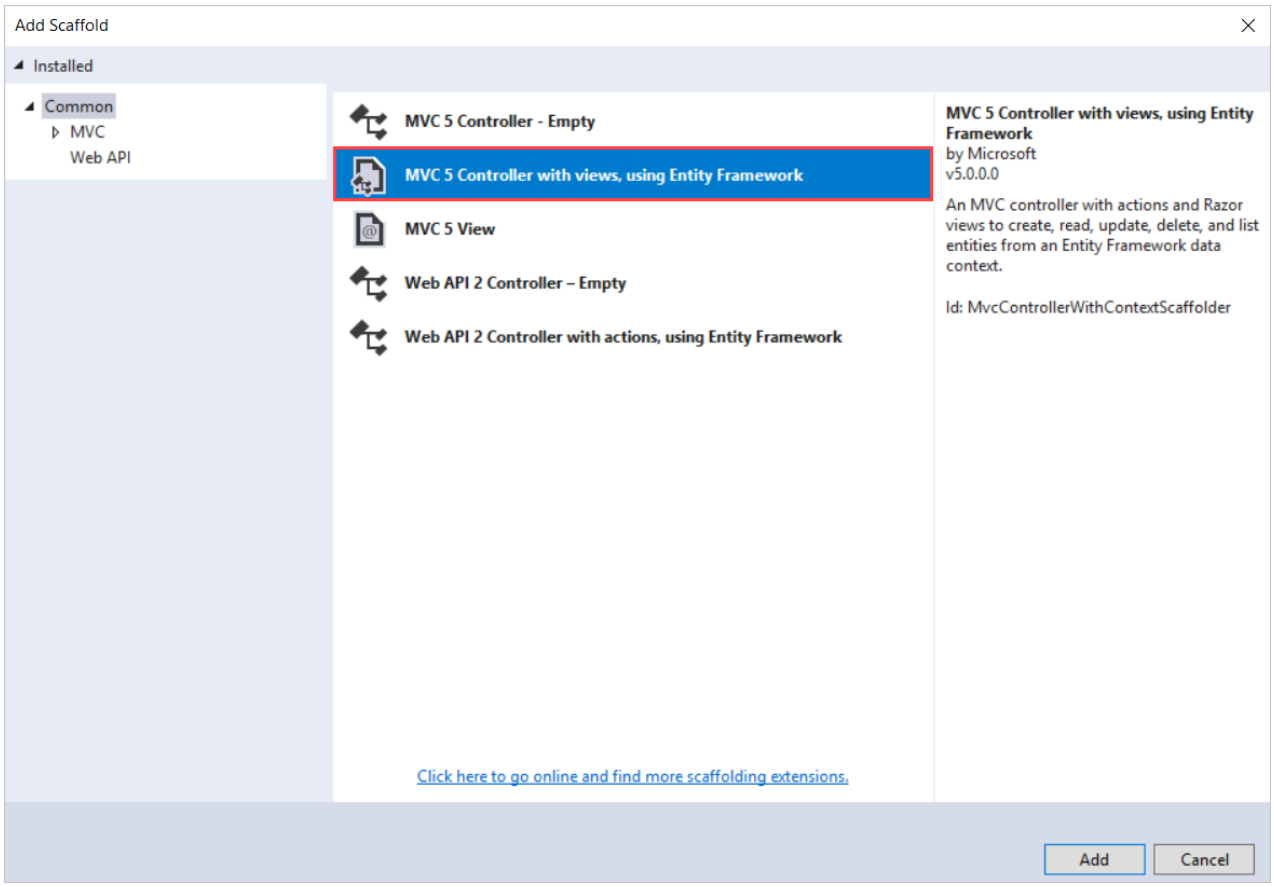
- Create the web application and data models

Add scaffold

You are ready to generate code that will provide standard data operations for the model classes. You add the code by adding a scaffold item. There are many options for the type of scaffolding you can add; in this tutorial, the scaffold will include a controller and views that correspond to the Student and Enrollment models you created in the previous section.

To maintain consistency in your project, you will add the new controller to the existing **Controllers** folder. Right-click the **Controllers** folder, and select **Add** > **New Scaffolded Item**.

Select the **MVC 5 Controller with views, using Entity Framework** option. This option will generate the controller and views for updating, deleting, creating and displaying the data in your model.



Select **Student (ContosoSite.Models)** for the model class and select the **ContosoUniversityDataEntities (ContosoSite.Models)** for the context class. Keep the controller name as **StudentsController**.

Click **Add**.

If you receive an error, it may be because you did not build the project in the previous section. If so, try building the project, and then add the scaffolded item again.

After the code generation process is complete, you will see a new controller and views in your project's **Controllers** and **Views** > **Students** folders.

Perform the same steps again, but add a scaffold for the **Enrollment** class. When finished, you have an **EnrollmentsController.cs** file, and a folder under **Views** named **Enrollments** with the Create, Delete, Details, Edit and Index views.

Add links to new views

To make it easier for you to navigate to your new views, you can add a couple of hyperlinks to the Index views for students and enrollments. Open the file at **Views** > **Home** > *Index.cshtml*, which is the home page for your site. Add the following code below the jumbotron.

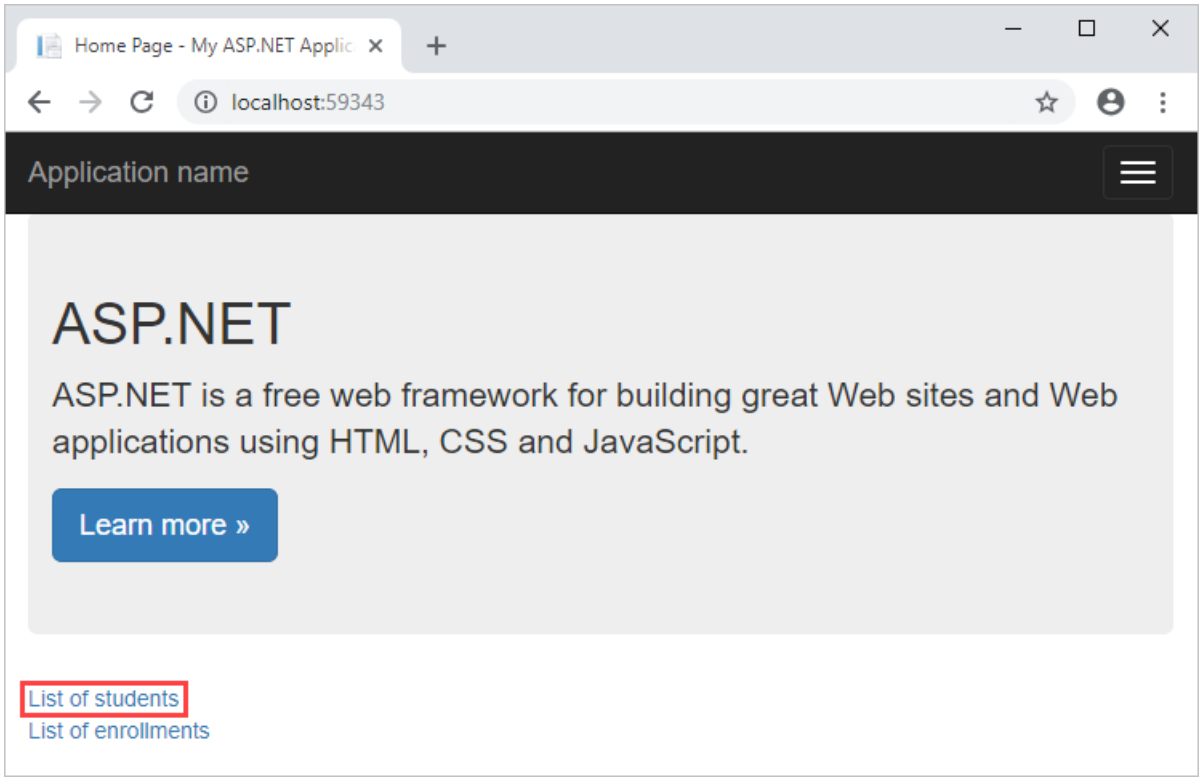
```
<div>
    @Html.ActionLink("List of students", "Index", "Students")<br />
    @Html.ActionLink("List of enrollments", "Index", "Enrollments")
</div>
```

For the ActionLink method, the first parameter is the text to display in the link. The second parameter is the action and the third parameter is the name of the controller. For example, the first link points to the Index action in StudentsController. The actual hyperlink is constructed from these values. The first link ultimately takes users to the **Index.cshtml** file within the **Views/Students** folder.

Display student views

You will verify that the code added to your project correctly displays a list of the students, and enables users to edit, create, or delete the student records in the database.

Right-click the **Views** > **Home** > *Index.cshtml* file, and select **View in Browser**. On the application home page, select **List of students**.



On the **Index** page, notice the list of the students and links to modify this data. Select the **Create New** link and provide some values for a new student. Click **Create**, and notice the new student is added to your list.

Back on the **Index** page, select the **Edit** link, and change some of the values for a student. Click **Save**, and notice the student record has been changed.

Finally, select the **Delete** link and confirm that you want to delete the record by clicking the **Delete** button.

Without writing any code, you have added views that perform common operations on the data in the Student table.

You may have noticed that the text label for a field is based on the database property (such as **LastName**) which is not necessarily what you want to display on the web page. For example, you may prefer the label to be **Last Name**. You will fix this display issue later in the tutorial.

Display enrollment views

Your database includes a one-to-many relationship between the Student and Enrollment tables, and a one-to-many relationship between the Course and Enrollment tables. The views for Enrollment correctly handle these relationships. Navigate to the home page for your site and select the **List of enrollments** link and then the **Create New** link.

The view displays a form for creating a new enrollment record. In particular, notice that the form contains a **CourseID** drop-down list and a **StudentID** drop-down list. Both are populated with values from the related tables.

Furthermore, validation of the provided values is automatically applied based on the data type of the field. **Grade** requires a number, so an error message is displayed if you try to provide an incompatible value: *The field Grade*

must be a number.

You have verified that the automatically-generated views enable users to work with the data in the database. In the next tutorial in this series, you will update the database and make the corresponding changes in the web application.

Next steps

In this tutorial, you:

- Added scaffold
- Added links to new views
- Displayed student views
- Displayed enrollment views

Advance to the next tutorial to learn how to change the database.

Change the database

Tutorial: Change the database for EF Database First with ASP.NET MVC app

2/20/2020 • 2 minutes to read • Edit Online

Using MVC, Entity Framework, and ASP.NET Scaffolding, you can create a web application that provides an interface to an existing database. This tutorial series shows you how to automatically generate code that enables users to display, edit, create, and delete data that resides in a database table. The generated code corresponds to the columns in the database table.

This tutorial focuses on making an update to the database structure and propagating that change throughout the web application.

In this tutorial, you:

- Add a column
- Add the property to the views

Prerequisites

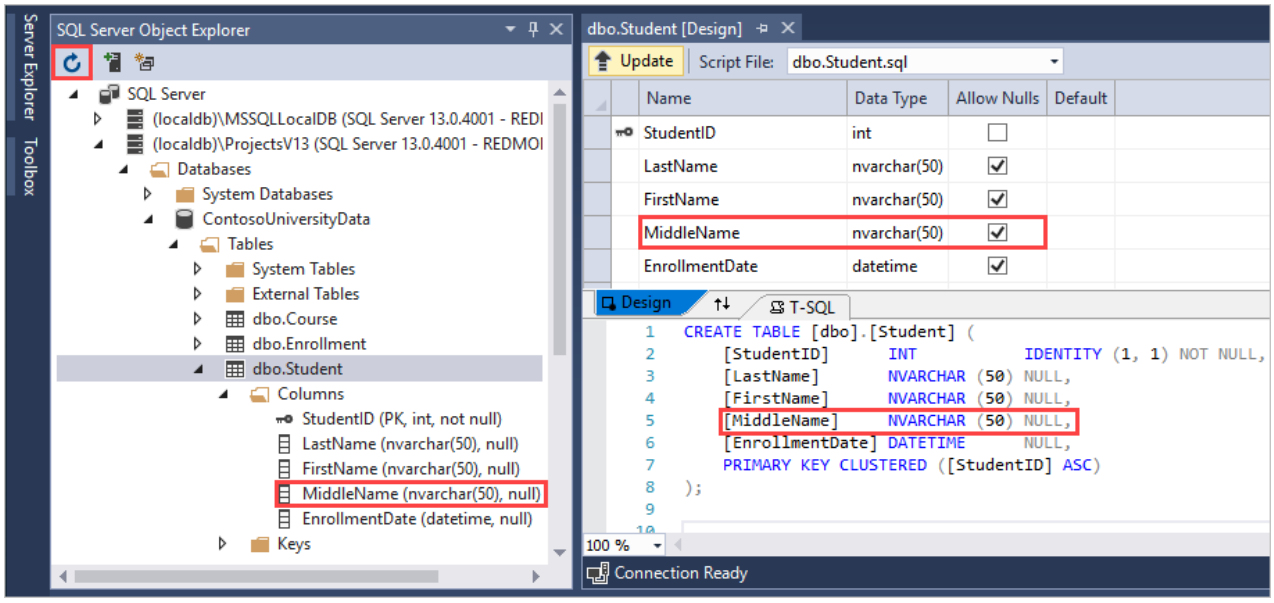
- Generating views

Add a column

If you update the structure of a table in your database, you need to ensure that your change is propagated to the data model, views, and controller.

For this tutorial, you will add a new column to the Student table to record the middle name of the student. To add this column, open the database project, and open the Student.sql file. Through either the designer or the T-SQL code, add a column named **MiddleName** that is an NVARCHAR(50) and allows NULL values.

Deploy this change to your local database by starting your database project (or F5). The new field is added to the table. If you do not see it in the SQL Server Object Explorer, click the Refresh button in the pane.



The new column exists in the database table, but it does not currently exist in the data model class. You must update the model to include your new column. In the **Models** folder, open the **ContosoModel.edmx** file to

display the model diagram. Notice that the Student model does not contain the MiddleName property. Right-click anywhere on the design surface, and select **Update Model from Database**.

In the Update Wizard, select the **Refresh** tab and then select **Tables** > **dbo** > **Student**. Click **Finish**.

After the update process is finished, the database diagram includes the new **MiddleName** property. Save the **ContosoModel.edmx** file. You must save this file for the new property to be propagated to the **Student.cs** class. You have now updated the database and the model.

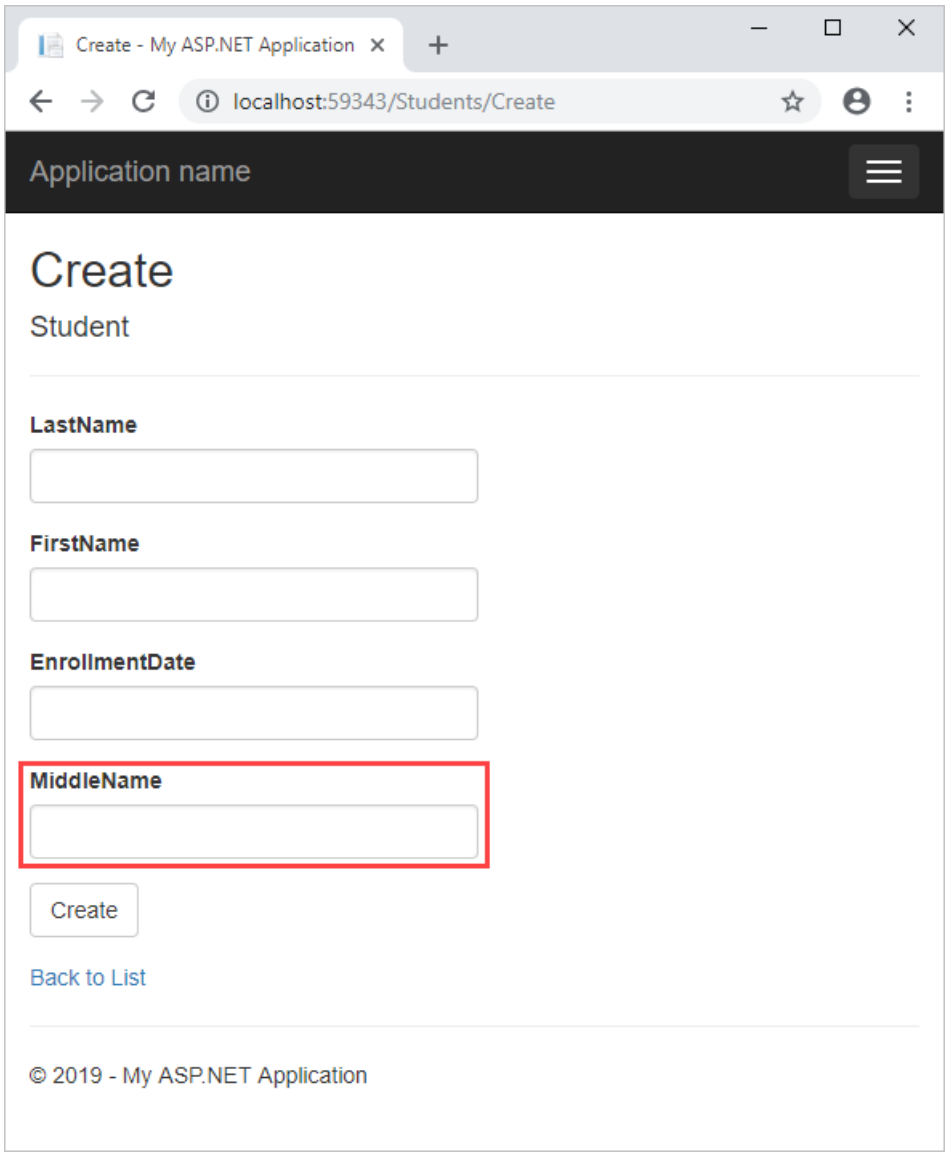
Build the solution.

Add the property to the views

Unfortunately, the views still do not contain the new property. To update the views you have two options - you can either re-generate the views by once again adding scaffolding for the Student class, or you can manually add the new property to your existing views. In this tutorial, you will add the scaffolding again because you have not made any customized changes to the automatically-generated views. You might consider manually adding the property when you have made changes to the views and do not want to lose those changes.

To ensure the views are re-created, delete the **Students** folder under **Views**, and delete the **StudentsController**. Then, right-click the **Controllers** folder and add scaffolding for the **Student** model. Again, name the controller **StudentsController**. Select **Add**.

Build the solution again. The views now contain the MiddleName property.



Next steps

In this tutorial, you:

- Added a column
- Added the property to the views

Advance to the next tutorial to learn how to customize the view for showing details about a student record.

Customize a view

Tutorial: Customize view for EF Database First with ASP.NET MVC app

2/20/2020 • 2 minutes to read • Edit Online

Using MVC, Entity Framework, and ASP.NET Scaffolding, you can create a web application that provides an interface to an existing database. This tutorial series shows you how to automatically generate code that enables users to display, edit, create, and delete data that resides in a database table. The generated code corresponds to the columns in the database table.

This tutorial focuses on changing the automatically-generated views to enhance the presentation.

In this tutorial, you:

- Add courses to the student detail page
- Confirm that the courses are added to the page

Prerequisites

- Change the database

Add courses to student detail

The generated code provides a good starting point for your application but it does not necessarily provide all of the functionality that you need in your application. You can customize the code to meet the particular requirements of your application. Currently, your application does not display the enrolled courses for the selected student. In this section, you will add the enrolled courses for each student to the **Details** view for the student.

Open **Views** > **Students** > *Details.cshtml*. Below the last </dl> tag, but before the closing tag, add the following code.

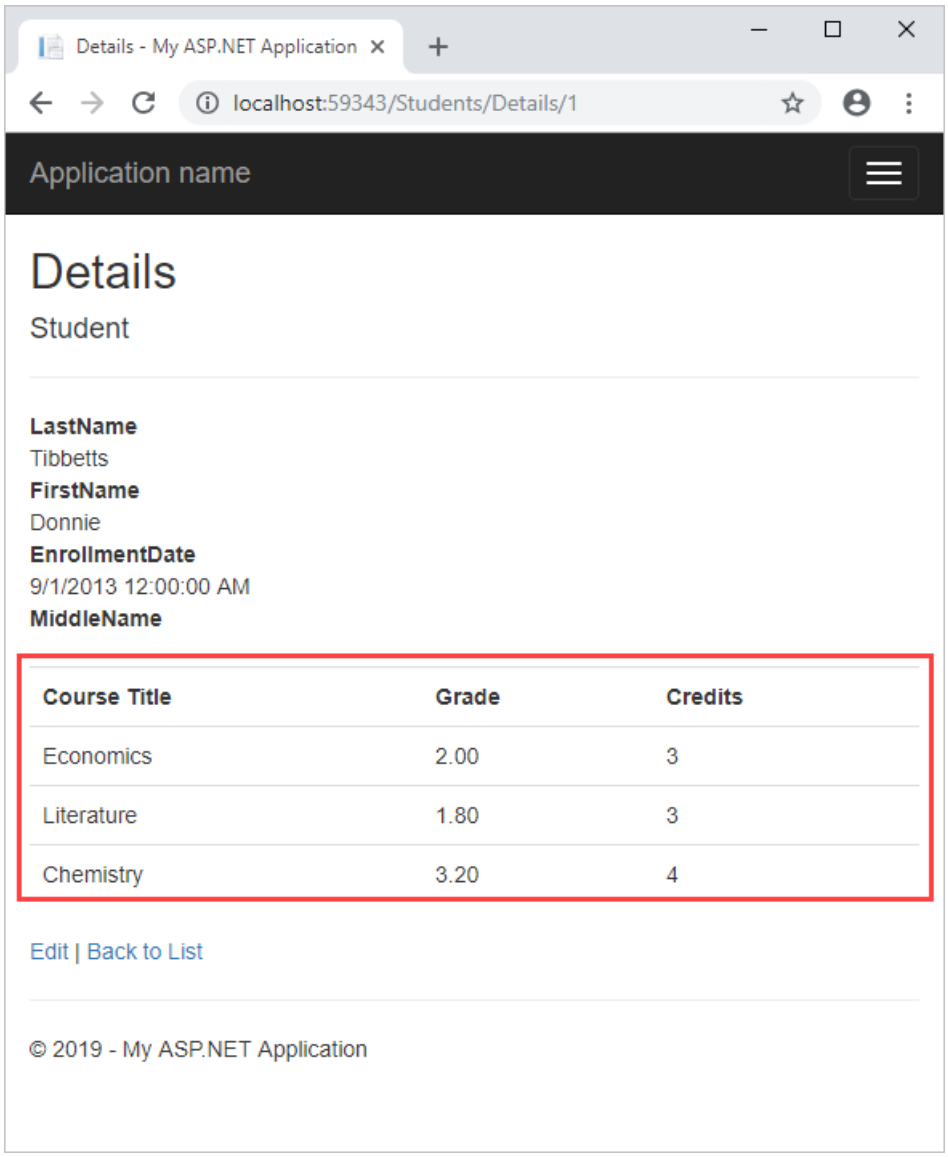
```
<table class="table">
    <tr>
        <th>
            Course Title
        </th>
        <th>
            Grade
        </th>
        <th>
            Credits
        </th>
    </tr>

    @foreach (var item in Model.Enrollments)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Course.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Grade)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Course.Credits)
            </td>
        </tr>
    }
</table>
```

This code creates a table that displays a row for each record in the Enrollment table for the selected student. The **Display** method renders HTML for the object (modelItem) that represents the expression. You use the Display method (rather than simply embedding the property value in the code) to make sure the value is formatted correctly based on its type and the template for that type. In this example, each expression returns a single property from the current record in the loop, and the values are primitive types which are rendered as text.

Confirm courses are added

Run the solution. Click **List of students** and select **Details** for one of the students. You will see the enrolled courses have been included in the view.



Next steps

In this tutorial, you:

- Added courses to the student detail page
- Confirmed that the courses are added to the page

Advance to the next tutorial to learn how to add data annotations to specify validation requirements and display formatting.

Enhance data validation

Tutorial: Enhance data validation for EF Database First with ASP.NET MVC app

2/20/2020 • 5 minutes to read • Edit Online

Using MVC, Entity Framework, and ASP.NET Scaffolding, you can create a web application that provides an interface to an existing database. This tutorial series shows you how to automatically generate code that enables users to display, edit, create, and delete data that resides in a database table. The generated code corresponds to the columns in the database table.

This tutorial focuses on adding data annotations to the data model to specify validation requirements and display formatting. It was improved based on feedback from users in the comments section.

In this tutorial, you:

- Add data annotations
- Add metadata classes

Prerequisites

- Customize a view

Add data annotations

As you saw in an earlier topic, some data validation rules are automatically applied to the user input. For example, you can only provide a number for the Grade property. To specify more data validation rules, you can add data annotations to your model class. These annotations are applied throughout your web application for the specified property. You can also apply formatting attributes that change how the properties are displayed; such as, changing the value used for text labels.

In this tutorial, you will add data annotations to restrict the length of the values provided for the FirstName, LastName, and MiddleName properties. In the database, these values are limited to 50 characters; however, in your web application that character limit is currently not enforced. If a user provides more than 50 characters for one of those values, the page will crash when attempting to save the value to the database. You will also restrict Grade to values between 0 and 4.

Select **Models** > **ContosoModel.edmx** > **ContosoModel.tt** and open the *Student.cs* file. Add the following highlighted code to the class.

```
namespace ContosoSite.Models
{
    using System;
    using System.Collections.Generic;
    using System.ComponentModel.DataAnnotations;

    public partial class Student
    {
        public Student()
        {
            this.Enrollments = new HashSet<Enrollment>();
        }

        public int StudentID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50)]
        public string FirstName { get; set; }
        public Nullable<System.DateTime> EnrollmentDate { get; set; }
        [StringLength(50)]
        public string MiddleName { get; set; }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

Open *Enrollment.cs* and add the following highlighted code.

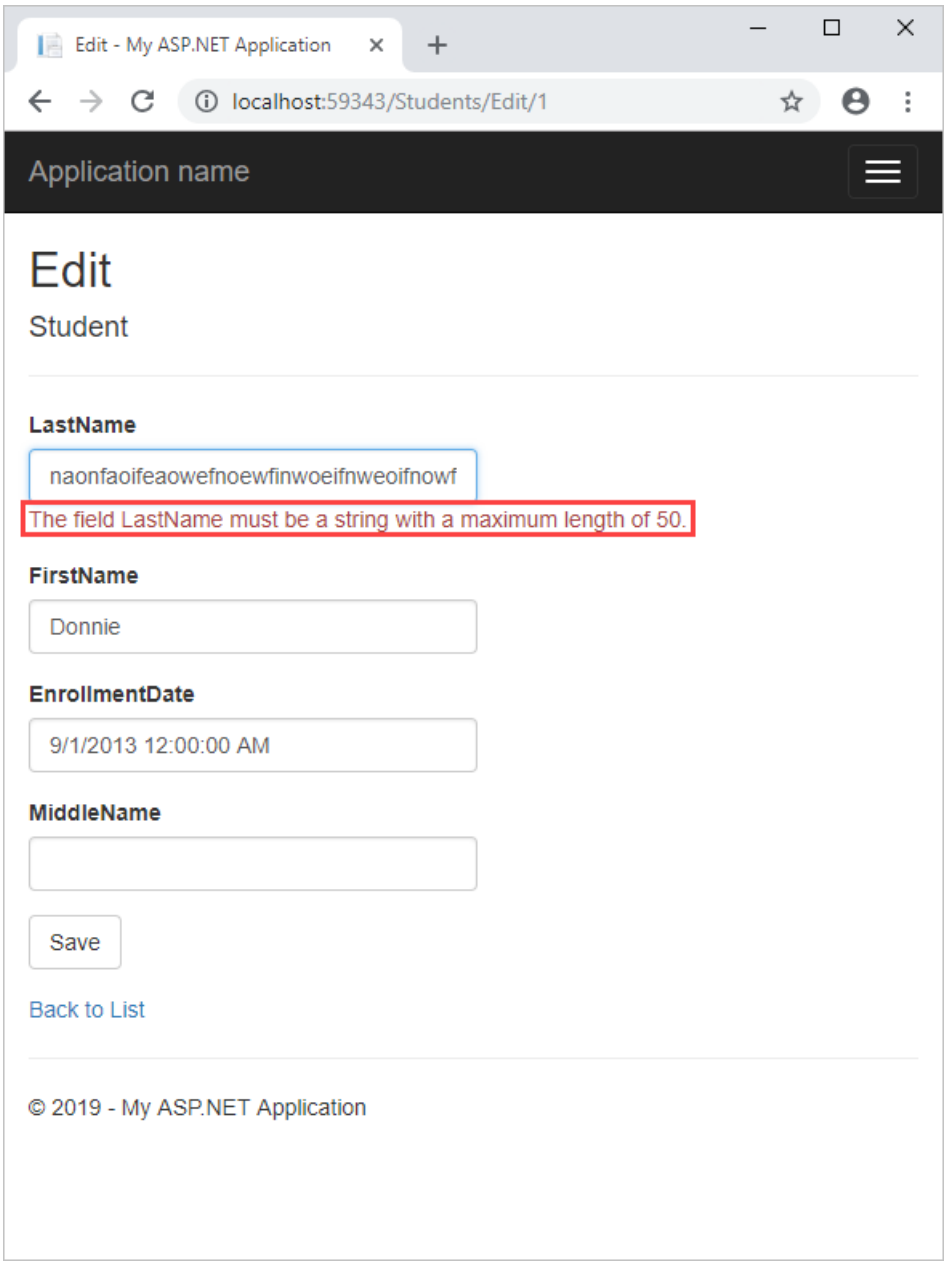
```
namespace ContosoSite.Models
{
    using System;
    using System.Collections.Generic;
    using System.ComponentModel.DataAnnotations;

    public partial class Enrollment
    {
        public int EnrollmentID { get; set; }
        [Range(0, 4)]
        public Nullable<decimal> Grade { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }

        public virtual Course Course { get; set; }
        public virtual Student Student { get; set; }
    }
}
```

Build the solution.

Click **List of students** and select **Edit**. If you attempt to enter more than 50 characters, an error message is displayed.



Go back to the home page. Click **List of enrollments** and select **Edit**. Attempt to provide a grade above 4. You will receive this error: *The field Grade must be between 0 and 4.*

Add metadata classes

Adding the validation attributes directly to the model class works when you do not expect the database to change; however, if your database changes and you need to regenerate the model class, you will lose all of the attributes you had applied to the model class. This approach can be very inefficient and prone to losing important validation rules.

To avoid this problem, you can add a metadata class that contains the attributes. When you associate the model class to the metadata class, those attributes are applied to the model. In this approach, the model class can be regenerated without losing all of the attributes that have been applied to the metadata class.

In the **Models** folder, add a class named *Metadata.cs*.

Replace the code in *Metadata.cs* with the following code.

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoSite.Models
{
    public class StudentMetadata
    {
        [StringLength(50)]
        [Display(Name="Last Name")]
        public string LastName;

        [StringLength(50)]
        [Display(Name="First Name")]
        public string FirstName;

        [StringLength(50)]
        [Display(Name="Middle Name")]
        public string MiddleName;

        [Display(Name = "Enrollment Date")]
        public Nullable<System.DateTime> EnrollmentDate;
    }

    public class EnrollmentMetadata
    {
        [Range(0, 4)]
        public Nullable<decimal> Grade;
    }
}
```

These metadata classes contain all of the validation attributes that you had previously applied to the model classes. The **Display** attribute is used to change the value used for text labels.

Now, you must associate the model classes with the metadata classes.

In the **Models** folder, add a class named *PartialClasses.cs*.

Replace the contents of the file with the following code.

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoSite.Models
{
    [MetadataType(typeof(StudentMetadata))]
    public partial class Student
    {
    }

    [MetadataType(typeof(EnrollmentMetadata))]
    public partial class Enrollment
    {
    }
}
```

Notice that each class is marked as a `partial` class, and each matches the name and namespace as the class that is automatically generated. By applying the metadata attribute to the partial class, you ensure that the data validation attributes will be applied to the automatically-generated class. These attributes will not be lost when you regenerate the model classes because the metadata attribute is applied in partial classes that are not regenerated.

To regenerate the automatically-generated classes, open the *ContosoModel.edmx* file. Once again, right-click on the design surface and select **Update Model from Database**. Even though you have not changed the database,

this process will regenerate the classes. In the **Refresh** tab, select **Tables** and **Finish**.

Save the *ContosoModel.edmx* file to apply the changes.

Open the *Student.cs* file or the *Enrollment.cs* file, and notice that the data validation attributes you applied earlier are no longer in the file. However, run the application, and notice that the validation rules are still applied when you enter data.

Conclusion

This series provided a simple example of how to generate code from an existing database that enables users to edit, update, create and delete data. It used ASP.NET MVC 5, Entity Framework and ASP.NET Scaffolding to create the project.

For an introductory example of Code First development, see Getting Started with ASP.NET MVC 5.

For a more advanced example, see Creating an Entity Framework Data Model for an ASP.NET MVC 4 App. Note that the DbContext API that you use for working with data in Database First is the same as the API you use for working with data in Code First. Even if you intend to use Database First, you can learn how to handle more complex scenarios such as reading and updating related data, handling concurrency conflicts, and so forth from a Code First tutorial. The only difference is in how the database, context class, and entity classes are created.

Additional resources

For a full list of data validation annotations you can apply to properties and classes, see System.ComponentModel.DataAnnotations.

Next steps

In this tutorial, you:

- Added data annotations
- Added metadata classes

To learn how to deploy a web app and SQL database to Azure App Service, see this tutorial:

Deploy a .NET app to Azure App Service

Lifecycle of an ASP.NET MVC 5 Application

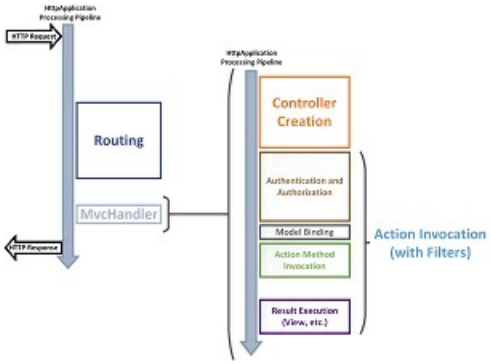
2/20/2020 • 2 minutes to read • Edit Online

by Cephas Lin

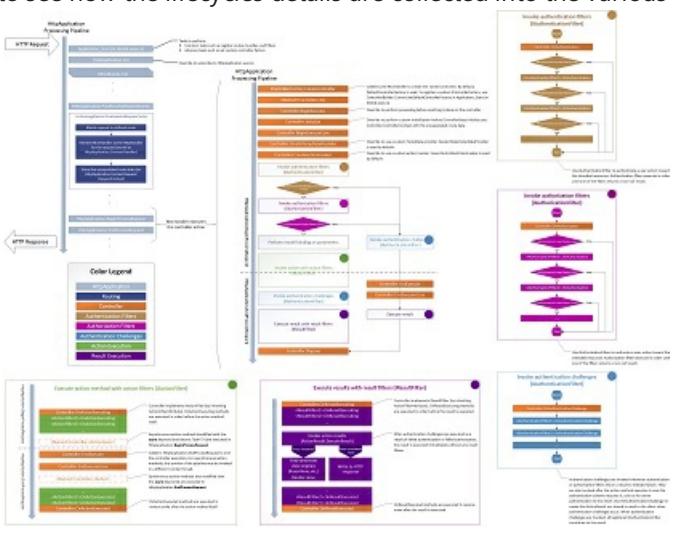
Download PDF Document

Here you can download a PDF document that charts the lifecycle of every ASP.NET MVC 5 application, from receiving the HTTP request to sending the HTTP response back to the client. It is designed both as an educational tool for those who are new to ASP.NET MVC and also as a reference for those who need to drill into specific aspects of the application. The PDF document has the following features:

- Relevant HttpApplication stages to help you understand where MVC integrates into the ASP.NET application lifecycle.
- A high-level view of the MVC application lifecycle, where you can understand the major stages that every MVC application passes through in the request processing pipeline.



- A detail view that shows drills down into the details of the request processing pipeline. You can compare the high-level view and the detail view to see how the lifecycles details are collected into the various stages.



Download PDF to see a larger view.

- Placement and purpose of all overridable methods on the Controller object in the request processing pipeline. You may or may not have the need to override any one method, but it is important for you to understand their role in the application lifecycle so that you can write code at the appropriate life cycle stage for the effect you intend.
- Blown-up diagrams showing how each of the filter types (authentication, authorization, action, and result) is invoked.
- Link to a useful article or blog from each point of interest in the detail view.

Next Steps

Does this document meet your need? We'd appreciate your feedback. If you have any question on the ASP.NET MVC lifecycle in your application, Stackoverflow and the ASP.NET MVC forums are great places to ask. Follow me on twitter so you can get updates on my latest tutorials.

MVC Recommended Resources

2/20/2020 • 12 minutes to read • Edit Online

by Rick Anderson

> This topic provides links to documentation resources about ASP.NET MVC 5. If you know a great blog post, stackoverflow thread or any other link that would be useful, send [me](aspnetue@microsoft.com?subject=Great ASP.NET MVC link) an email with the link. The links provided here are updated periodically. You can find older (but in many cases still relevant) ASP.NET MVC links on the MSDN content map.
>
> A good place to get your questions answered or to find solutions to problems is in the ASP.NET MVC forum or on stackoverflow.

Getting Started with ASP.NET MVC

- Intro to ASP.NET MVC 5
- Introduction to ASP.NET MVC (video by Pluralsight)
- Understanding Model View Controller in Asp.Net MVC (short blog intro).
- ASP.NET MVC Overview
- Free 6 hour online course - Developing ASP.NET MVC 4 Web Applications Jump Start
- Learning MVC - Part 1: Introduction to MVC Architecture and Separation of Concerns
- Poster: Lifecycle of an ASP.NET MVC 5 Application

New Featured Content

- Selecting Cultures for Localization in ASP.NET
- Create an ASP.NET MVC 5 App with Facebook and Google OAuth2 Sign-on
- Understanding ASP.NET MVC Filters and Attributes
- Lifecycle of an ASP.NET MVC 5 Application (PDF)
- What not to do in ASP.NET, and what to do instead
- Releasing SocialGoal - Reference Web App for ASP.NET MVC 5, EF 6 Code First, AutoMapper, Autofac, TDD and DDD
- Building Real-World Cloud Apps with Azure
- Detailed ASP.NET MVC Pipeline

Microsoft Azure

- The World's Greatest Azure Demo
- Videos from //build/ 2014: (Azure and Visual Studio)
  - Deep Dive into Hidden Gems, Tips and Tricks for Azure Web Sites by David Ebbo
  - What's New for ASP.NET and Web in Visual Studio 2013 Update 2 and Beyond by Scott Hanselman, Scott Hunter
  - Avoiding Cloud Fail: Learning from the Mistakes of Azure with Mark Russinovich
  - Deep Dive: Improving Performance in Your ASP.NET App by Levi Broderick
  - What's New for Azure Developers in Visual Studio and Azure SDK Dan Fernandez, Paul Yuknewicz (MSFT)

- Microsoft Azure Storage – What's New, Best Practices and Patterns Jai Haridas, Serdar Ozler
- Tips and Tricks in Visual Studio 2013 by Cathy Sullivan
- Automation Everywhere with New SDKs, Tools, and Services in Azure by Brady Gaster, Joe Levy
- What's New with Azure Web Sites Chris Compy, Nir Mashkowski
- Building Big: Lessons Learned from Azure Customers by Chris Clayton, Mark Simms
- Building a Single Page Application with ASP.NET and AngularJS by David Catuhe, Jon Galloway
- Designing and Implementing Hybrid Cloud Solutions for Architects
- Azure Resource Group Model: Modern Management for Modern Cloud by Gautam Thapar, Nathan Totten
- Building Web APIs for Mobile Apps Using ASP.NET Web API 2.1 by Daniel Roth
- The New Authentication Model for Web, Mobile, and Cloud Applications by Lucas Adams, Stuart Kwan
- Extending Your On-Premises Network into Azure Securely Ganesh Srinivasan
- What's New for Web Developers by Dan Fernandez, Mads Kristensen
- Building Modern Web Applications with Visual Studio and Web Essentials Mads Kristensen
- Using the Cloud-Based Load Testing Service and Application Insights to Find Scale and Performance by Charles Sterling
- Diagnosing Issues with Cloud Applications Hosted in Azure Using Visual Studio by Boris Scholl
- Rapidly Detect Application Outages with Application Insights by Vlad Joanovic
- Azure Web Sites: Architecting Massive-Scale Ready-For-Business Web Apps Yochay Kiriaty
- Azure SQL Database: Under the Hood Tobias Ternstrom

Security

- Create an ASP.NET MVC 5 App with Facebook and Google OAuth2 Sign-on
- What not to do in ASP.NET, and what to do instead
- How to build (and how not to build) a secure "remember me" feature by Troy Hunt
- Video: The New Authentication Model for Web, Mobile, and Cloud Applications by Lucas Adams, Stuart Kwan
- Video: Building Enterprise and SaaS Web Apps and Web APIs using Azure Active Directory for Sign In by Vittorio Bertocci
- ASP.NET Identity 2.0.0
- Understanding ASP.NET MVC Filters and Attributes
- ASP.NET MVC 5 Identity: Extending and Modifying Roles
- Anti-Forgery Tokens using MVC, Web API and AngularJS
- Working With IAuthenticationFilter in ASP.NET MVC 5
- stackoverflow threads on security by Levi
- KnockoutJS and ASP.NET MVC – Alternate techniques and using the Anti Forgery Token
- ASP.NET MVC 5 Authentication Filters
- ASP.NET MVC 5 Authentication Breakdown
- Security In ASP.NET MVC
- Securing your ASP.NET MVC 4 App
- Pluralsight video overview of security practices
- XSRF/CSRF Prevention in ASP.NET MVC and Web Pages
- OWASP Top 10 for .NET developers
- Using OAuth Providers with MVC 4
- Hack-Proofing Your ASP.NET Applications December 2011 MSDN Magazine article by Adam Tuliper
- Securing Your ASP.NET Applications January 2012 MSDN Magazine article by Adam Tuliper
- Preventing Open Redirection Attacks

- How to Create an Intranet Site Using ASP.NET MVC
- ASP.NET MVC Value Provider for encrypted query string
- The HaaHa Show: Microsoft ASP.NET MVC Security with Haack and Hanselman (video)
- Claims-aware user identity in ASP.NET
- Adding custom roles to windows roles in ASP.NET
- Creating a Custom Route Constraint
- Implementing Role Based Menu in ASP.NET MVC 4

Membership, Authentication

- Deploy a Secure ASP.NET MVC application with OAuth, Membership and SQL Database
- SimpleMembership, Membership Providers, Universal Providers and the new ASP.NET 4.5 Web Forms and ASP.NET MVC 4 templates
- Customize the SimpleMembership in ASP.NET MVC 4
- Membership is not the same as Forms Authentication
- Think twice about using MembershipProvider (and SimpleMembership)
- Implementing MVC Two Factor Authentication with Google Authenticator
- new
- ASP.NET Identity and Azure Table Storage

OWIN and Katana

- Getting started with OWIN and Katana
- A primer on external login providers (social logins) with OWIN/Katana authentication middleware

Data Access in ASP.NET MVC

- Getting Started with Entity Framework using ASP.NET MVC
- ASP.NET Data Access Content Map
- Building an ASP.NET MVC4 Application with EF and Web API
- DataList in ASP.NET MVC & KnockoutJS

Web Forms to MVC

- Migrating ASP.NET Web Forms to the MVC Pattern with the ASP.NET Web API
- Matthew Kane's video, Deep Integration of MVC and WebForms
- ASP.NET MVC - Living in a Web Forms World
- Web Forms to MVC
- Integrating ASP.NET Web Forms and ASP.NET MVC
- Adopting ASP.NET MVC enhancements in an Existing Web Forms Project

Performance

- Video: Deep Dive: Improving Performance in Your ASP.NET App by Levi Broderick
- Video: Using the Cloud-Based Load Testing Service and Application Insights to Find Scale and Performance by Charles Sterling
- Using Asynchronous Methods in ASP.NET MVC 4
- Bundling, Minification and Optimization
- Using CDNs and Expires to Improve Web Site Performance
- Async File Uploads in MVC

- Video: High-Performance Web Platform: Real-World Problems and Solutions by Tobin Titus
- Video: Gain Early Warning of Performance Issues and Failures, and Pinpoint the Cause with Application by Bret Grinslade, Vlad Joanovic
- Donut Caching and Donut Hole Caching with Asp.Net MVC 4
- Understanding Caching in Asp.Net MVC with example
- http://webdevchecklist.com/asp.net/performance/
- ASP.NET MVC View Model: Entity Framework & JSON Serialization Nandip Makwana shows a nice tip to increase performance (and lower CPU usage) by eliminating unnecessary serialization of child entities
- Async and Await in ASP.NET by Stephen Cleary
- Debugging and Performance using Glimpse
- Improve perceived performance of ASP.NET MVC websites with asynchronous partial views
- Asynchronous Database Calls With Task-based Asynchronous Programming Model (TAP) in ASP.NET MVC 4
- ASP.NET 4.5 & MVC 4: Revisiting IBundleTransform
- ASP.NET MVC 4: Cookieless domain for bundling and static resources
- Advanced Options of ASP.NET Bundling and Minification
- A harsh reminder about the importance of debug="false"
- ELMAH and MiniProfiler In ASP.NET MVC 4
- ASP.NET Bundling/Minification and Embedded Resources
- Using CDNs and Expires to Improve Web Site Performance
- Bundling and Minifying Inline CSS and JavaScript
- Async in ASP.NET (video)
- Bundling and Optimizing (video)
- Flushing in ASP.NET MVC
- Sorting a ScriptBundle based on /// reference tags
- ASP.NET MVC 4 - Making Asynchronous Calls to External Services from an Action Method

AngularJS, Knockout, KnockoutJS

- Knockout
- Getting Started with KnockoutJS in ASP.NET MVC
- Bootstrapping AngularJS Applications with Server-Side Data from ASP.NET MVC by Marius Schulz
- TypeScripted Knockout in ASP.NET MVC
- DataList in ASP.NET MVC & KnockoutJS
- Cascading drop down with knockout.js & ASP.NET MVC
- Loading KnockoutJS View Models from ASP.NET MVC, for faster page loads
- Master-Details view using Knockout and ASP.NET MVC
- AngularJS – Post data using the $resource Service in an ASP.NET MVC app

jQuery, Ajax, JSON, JavaScript

- Create a JavascriptResult for ASP.NET MVC
- Passing .NET Server-Side Data to JavaScript by Marius Schulz
- Generating External JavaScript Files Using Partial Razor Views by Marius Schulz
- ASP.NET MVC: Show Busy Indicator on Form Submit using JQuery and Ajax
- Using Custom Data Attributes to Store JSON Data in ASP.NET MVC
- MVC 4 links still relevant for MVC 5
- Sending JSON to an Action Method
- Build Rich User Interfaces using jQuery

- Auto populate DataTable columns to the Views in ASP.NET MVC by Sheo Narayan
- Using the HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC
- Using the DropDownList Helper with ASP.NET MVC
- DropDownListFor with ASP.NET MVC
- DropDownList Helper Data Binding in ASP.NET MVC
- Pluralsight video showing how to work with JavaScript and Ajax
- Cascading DropDownList in ASP.NET MVC
- CascadingDropDown jQuery Plugin for ASP.NET MVC
- MVC, JsonResult, DateTime and TimeZone
- MVC- Custom Validation - required DropDownList
- Client Side Progress Bar in ASP.NET MVC using jQuery UI, WebAPI and JsRender
- ASP.NET MVC Ajax using jQuery – Quick Start sample
- Creating Cascading DropDownLists using ASP.NET MVC 4 and jQuery

HTML5

- Using the HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC
- Designing Accessibility with HTML5
- Build Mobile-Friendly HTML5 Forms with ASP.NET MVC 4 and jQuery Mobile
- Integrate HTML5 into Existing ASP.NET Web Forms and MVC Applications

Routing

- Attribute Routing in ASP.NET MVC 5
- Lifecycle of an ASP.NET MVC 5 Application (PDF)
- Using Constraints for Better Routing in MVC
- In depth: The ASP.NET MVC Pipeline
- ASP.NET MVC Routing
- ASP.NET MVC Routing Extensibility
- An Introduction to ASP.NET MVC Extensibility
- ASP.NET MVC Routing
- Strongly typed routing for MVC 4

Globalization, Internationalization and Localization

- ASP.NET MVC 5 Internationalization
- Selecting Cultures for Localization in ASP.NET
- ASP.NET MVC 5 Internationalization · Date and Time by Nadeem Afana
- ASP.NET MVC 5 Internationalization · How to Store Strings in a Database or Xml
- Globalization, Internationalization and Localization in ASP.NET MVC 3, JavaScript and jQuery
- Internationalization
- Localization

Azure and ASP.NET MVC

- Multi-Tier Application Using Storage Tables, Queues, and Blobs
- Windows Azure Web Sites: Quick-and-Easy Hosting as a Service
- Configuration Tips For ASP.NET MVC 4 on a Azure Website
- Episode 85 - Azure, Windows 8, and MVC 4 Demos (video)

- Deploying an ASP.NET Website to Azure in Visual Studio 2012 from a Git Repository directly

Mobile

- ASP.NET MVC 4 Mobile Tutorial
- [Video] Building Mobile Friendly Websites with MVC 4
- Creating High-Performance Mobile Websites
- Creating Mobile-Optimized Views in ASP.NET MVC 4
- Build Mobile-Friendly HTML5 Forms with ASP.NET MVC 4 and jQuery Mobile
- Integrating jQuery Mobile into MVC Web Apps
- A Deeper Dive into jQuery Mobile
- What's New for Mobile Development in ASP.NET MVC 4
- Develop Hybrid Native and Mobile Web Apps
- Making a switchable Desktop and Mobile site with ASP.NET MVC 4 and jQuery Mobile
- MVC 4 Mobile Friendly Web Applications
- ASP.NET MVC 4 Mobile

Single Page Application (SPA)

- A .NET Developer Primer for Single-Page Applications MSDN Magazine
- Single-Page App with AngularJS and ASP.NET MVC 4
- Inside the ASP.NET Single Page Apps Template
- MVC Single Page Application Template Update for ASP.NET and Web Tools 2012.2 RC
- Inside the ASP.NET Single Page Apps Template

Twitter, Facebook

- The new Facebook application template and library for ASP.NET MVC
- Twitter.Bootstrap.MVC4
- MVC 4: facebook twitter login with oAuth

Unit Testing, TDD, Repository Pattern and Unit of Work

- Repository Pattern and Unit of Work with Entity Framework in ASP.NET MVC
- Creating Unit Testable Applications in ASP.NET MVC - A Beginner's Tutorial
- ASP.NET MVC controller dependency injection for beginners
- Writing Unit Tests for an ASP.NET MVC Action Method that handles Ajax Request and Normal Request
- Oak: Bring dynamic goodness and frictionless development to ASP.NET MVC
- Developing an ASP.NET MVC 4 Application for TDD using xUnit and Moq
- Testing ASP.NET MVC Views, from New Project to the Build Server (video)
- Quick Start: Test Driven Development(video)
- Test Driving ASP.NET MVC (video)
- Integration Testing With SpecsFor.Mvc – ASP.NET MVC Testing on Steroids! (video)
- Creating Unit Testable Applications in ASP.NET MVC - A Beginner's Tutorial
- Walkthrough: Using TDD with ASP.NET MVC (MVC 3)
- Releasing SocialGoal - Reference Web App for ASP.NET MVC 5, EF 6 Code First, AutoMapper, Autofac, TDD and DDD

Best Practices

- What not to do in ASP.NET, and what to do instead
- Web Developer Checklist
- Video: Don't do that, do this!
- ASP.NET MVC Solution Best Practices (video)
- SignalR, MVC, Web API (video)
- Best Practices for ASP.NET MVC
- ASP.NET MVC Best Practices (Part 1)
- 12 ASP.NET MVC Best Practices

Deployment

- Web Deployment Content Map for Visual Studio and ASP.NET

Rich text Editors for MVC

- http://www.tinymce.com/
- http://www.componentone.com/SuperProducts/StudioASPNET/
- Rich Text Editing with Infragistics HTML Editor in both jQuery and ASP.NET MVC
- DevExpress ASP.NET MVC RichEdit Control
- Nesting Layouts in MVC by Russell Waite

General

- How to Upgrade an ASP.NET MVC 4 and Web API Project to ASP.NET MVC 5 and Web API 2
- Understanding ASP.NET MVC Filters and Attributes
- Content Negotiation and Web API for the ASP.NET MVC Developer
- Serving WebP images with ASP.NET MVC
- Detailed ASP.NET MVC Pipeline
- Nesting Layouts in MVC by Russell Waite
- Implementing RichTextEditor or WYSIWYG (TinyMCE) in ASP.NET MVC by Sheo Narayan
- Binding views with Anonymous type collection in ASP.NET MVC by Sheo Narayan
- Saving HTML 5 canvas as Image in ASP.NET MVC by Sheo Narayan
- Maintaining image size and position on the page in database using ASP.NET MVC by Sheo Narayan
- Developing a star rating in ASP.NET MVC by Sheo Narayan
- Extension Methods Library useful for MVC by Nadeem Afana
- Upload and download files using ASP.NET MVC
- Get Compile-Time View Errors in ASP.NET MVC
- Using Fluent Validation with ASP.NET MVC
- ASP.NET MVC under the hood part_1 The MvcHandler, the object responsible for handling a request for ASP.NET MVC.
- ASP.NET MVC under the hood part_2
- ASP.NET MVC under the hood part 3 Finding the Controller Type
- ASP.NET MVC under the hood part 4 TempData
- ASP.NET MVC under the hood part 5 The ActionInvoker
- ASP.NET MVC under the hood part 6 How ActionFilters are discovered and how they are applied
- Create Your Own Custom ViewWebPage for ASP.NET MVC
- Tracing in ASP.NET MVC Razor Views
- Rendering ASP.NET MVC Razor Views outside of MVC revisited

- Five Essential Front-End Tools That Should be Used with Modern MVC Projects
- The Features and Foibles of ASP.NET MVC Model Binding
- Controlling Session Behavior in Asp.Net MVC4
- RenderPartial vs RenderAction vs Partial vs Action in MVC Razor
- Resolve Ambiguous Controller Error by routes
- Cookie-based TempData provider
- Sharing a single _ViewStart across areas in ASP.NET MVC
- Filtering in ASP.NET MVC
- Get to Know Action Filters in ASP.NET MVC 3 Using HandleError (video)
- MVC - Custom Filters - Session timeout Handling
- Custom MVC 4 Template Walkthrough
- Browser history
- Why ASP.NET MVC HTML Helper methods take lambda expressions as parameters
- A Web Developer's Look at the MVC4 Request Pipeline
- Understanding Text Encoding in ASP.NET MVC
- Upgrading an MVC4 Visual Studio 2010 Solution to Visual Studio 2012 and .Net 4.5 and EF 5
- ASP.NET MVC DisplayTemplate and EditorTemplates for Entity Framework DbGeography Spatial Types
- ASP.NET MVC - Highlight Current Link
- ASP.NET MVC: Handling Exceptions and 404 Errors
- Finding Bad Controllers
- ASP.NET MVC Tips, Tricks and Hidden Gems (video)
- Rendering an RDLC directly to the Response stream in ASP.NET MVC
- How To Achieve World(-Ready) Domination In ASP.NET MVC 4 (video)
- Deep Integration of MVC and WebForms (video)
- Enterprise Search with MVC 4 (video)
- Skinny controller in ASP.NET MVC 4
- Dynamic Menus in ASP.NET MVC 4 using EF Code First and jQuery
- Making ASP.NET GridView Responsive With jQuery FooTable
- Web Deployment using Visual Studio
- Custom Templates, Data Annotations and UI Hints in ASP.NET MVC

Forum Posts

- image is getting contenttype text/html after outputcache RSS
- How to get the controller and action names or even the route name from an absolutepath?
- Unobtrusive AJAX client side validation Remote method calls to check uniqueness of value

MVC recommended tutorials and articles

9/16/2020 • 2 minutes to read • Edit Online

by Rick Anderson

Getting Started

- Getting Started with ASP.NET MVC 5 This 11 part series is a good place to start.
- Pluralsight ASP.NET MVC 5 Fundamentals (video course)
- Intro to ASP.NET MVC by Jon Galloway and Christopher Harrison
- Lifecycle of an ASP.NET MVC 5 Application PDF document that charts the lifecycle of an ASP.NET MVC 5 app.

Working with data

- Getting Started with EF 6 Code First using MVC 5 Tom Dykstra's award winning series dives deep into EF.

Security

- Create an ASP.NET MVC app with auth and SQL DB and deploy to Azure This popular tutorial walks you through creating a simple app and adding membership and roles.
- Create an ASP.NET MVC 5 App with Facebook, Twitter, LinkedIn and Google OAuth2 Sign-on This tutorial shows you how to build an ASP.NET MVC 5 web application that enables users to log in using OAuth 2.0 with credentials from an external authentication provider, such as Facebook, Twitter, LinkedIn, Microsoft, or Google.
- Create a secure ASP.NET MVC 5 web app with log in, email confirmation and password reset First in a series on Identity, includes code to resend a confirmation link.
- ASP.NET MVC 5 app with SMS and email Two-Factor Authentication Second on Identity series.
- Best practices for deploying passwords and other sensitive data to ASP.NET and Azure App Service
- Two-factor authentication using SMS and email with ASP.NET Identity `isPersistent` and the security cookie, code to require a user to have a validated email account before they can log on, how SignInManager checks for 2FA requirement, and more.
- Account Confirmation and Password Recovery with ASP.NET Identity Provides details on Identity not found in Create a secure ASP.NET MVC 5 web app with log in, email confirmation and password reset such as how to let users reset their forgotten password.

Azure

- Create an ASP.NET web app in Azure Short and simple tutorial for deployment to Azure.
- Create an ASP.NET MVC app with auth and SQL DB and deploy to Azure

Performance and Debugging

- Profile and debug your ASP.NET MVC app with Glimpse

ASP.NET MVC DropDownListFor with SelectListItem

When using the DropDownListFor helper and passing to it the collection of `SelectListItem` from which it is populated, the `DropdownListFor` modifies the passed collection after it is called. `DropdownListFor` changes the `SelectListItems` Selected properties to whatever was selected by the dropdown list. This leads to unexpected

behavior.

The DropDownListFor, DropDownList, EnumDropDownListFor, ListBox, and ListBoxFor update the Selected property of any `IEnumerable<SelectListItem>` passed or found in ViewData.

The workaround is to create separate enumerables, containing distinct `SelectListItem` instances, for each property in the model.

For more information, see

- DropdownListFor modifies the SelectListItem collection passed to it
- GetSelectListWithDefaultValue Modifies IEnumerable selectList

ASP.NET MVC UI, Layouts, and Themes

2/20/2020 • 2 minutes to read • Edit Online

The Razor syntax and how to use it in views (the V in MVC) to display information.

- Using Page Inspector in ASP.NET MVC
- Dynamic v. Strongly Typed Views

Using Page Inspector in ASP.NET MVC

9/16/2020 • 9 minutes to read • Edit Online

by Tim Ammann

Page Inspector in Visual Studio 2012 is a web development tool with an integrated browser. Select any element in the integrated browser, and Page Inspector instantly highlights the element's source and CSS. You can browse any MVC view, quickly find the sources of rendered markup, and use browser tools right within the Visual Studio environment.

Watch the Video

This tutorial shows how to enable Inspection Mode, and then quickly locate and edit markup and CSS within your web project. The tutorial uses an MVC Project, but you can also use Page Inspector for Web Forms and other ASP.NET applications.

The tutorial has the following sections:

- Prerequisites
- Create a Web Application
- Use Page Inspector to Browse to a View
- Enable Inspection Mode
- Use Page Inspector to Make Changes to Markup
- Inspection Mode and the HTML Window
- Preview CSS Changes in the Styles window
- CSS Auto Sync
- Using the CSS Color Picker
- Mapping Dynamic Page Elements to JavaScript

Prerequisites

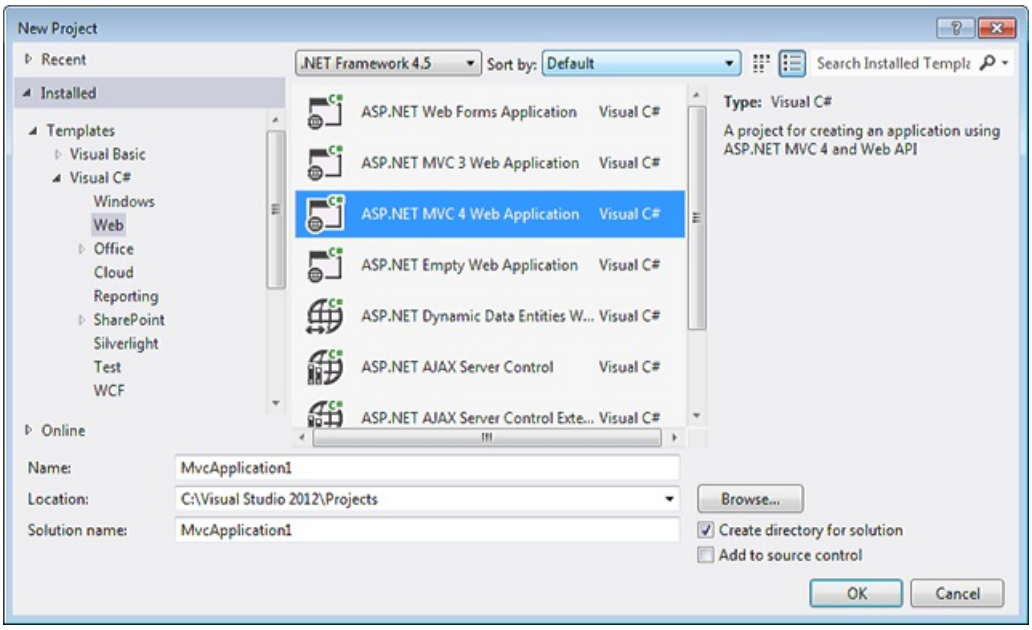
- Visual Studio 2012 or Visual Studio Express 2012 for Web.

> **NOTE**
>
> To get the latest version of Page Inspector, use Web Platform Installer to install the Windows Azure SDK for .NET 2.0.

Page Inspector is bundled with Microsoft Web Developer Tools. The latest version is 1.3. To check which version you have, run Visual Studio and select **About Microsoft Visual Studio** from the **Help** menu.

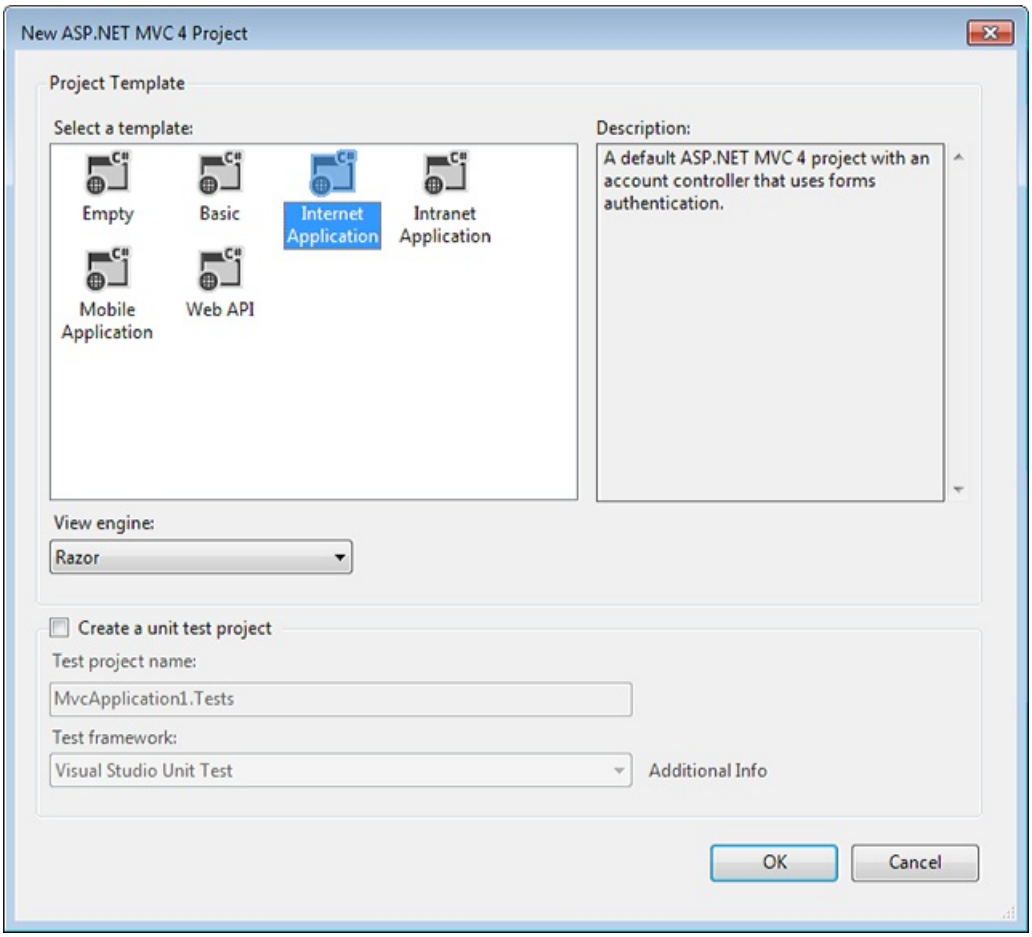
Create a Web Application

First, create a web application that you will use Page Inspector with. In Visual Studio, choose **File** > **New Project**. On the left, expand **Visual C#**, select **Web**, and then select **ASP.NET MVC4 Web Application**.

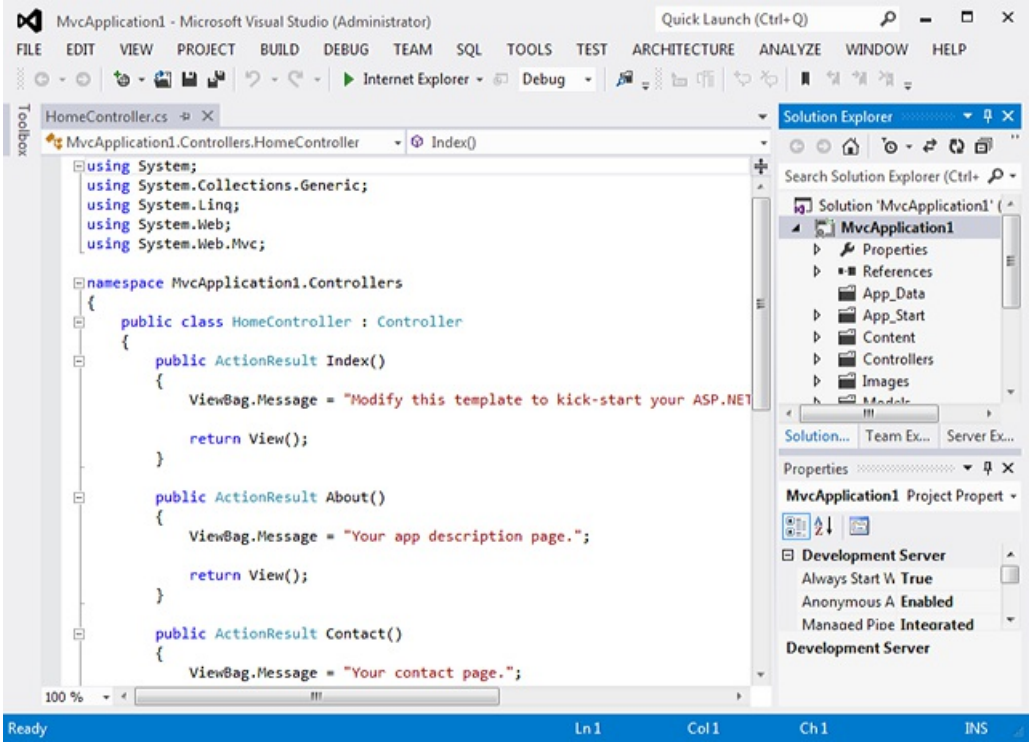


Click **OK**.

In the **New ASP.NET MVC 4 Project** dialog box, select **Internet Application**. Leave **Razor** as the default view engine.



The application opens in **Source** view.

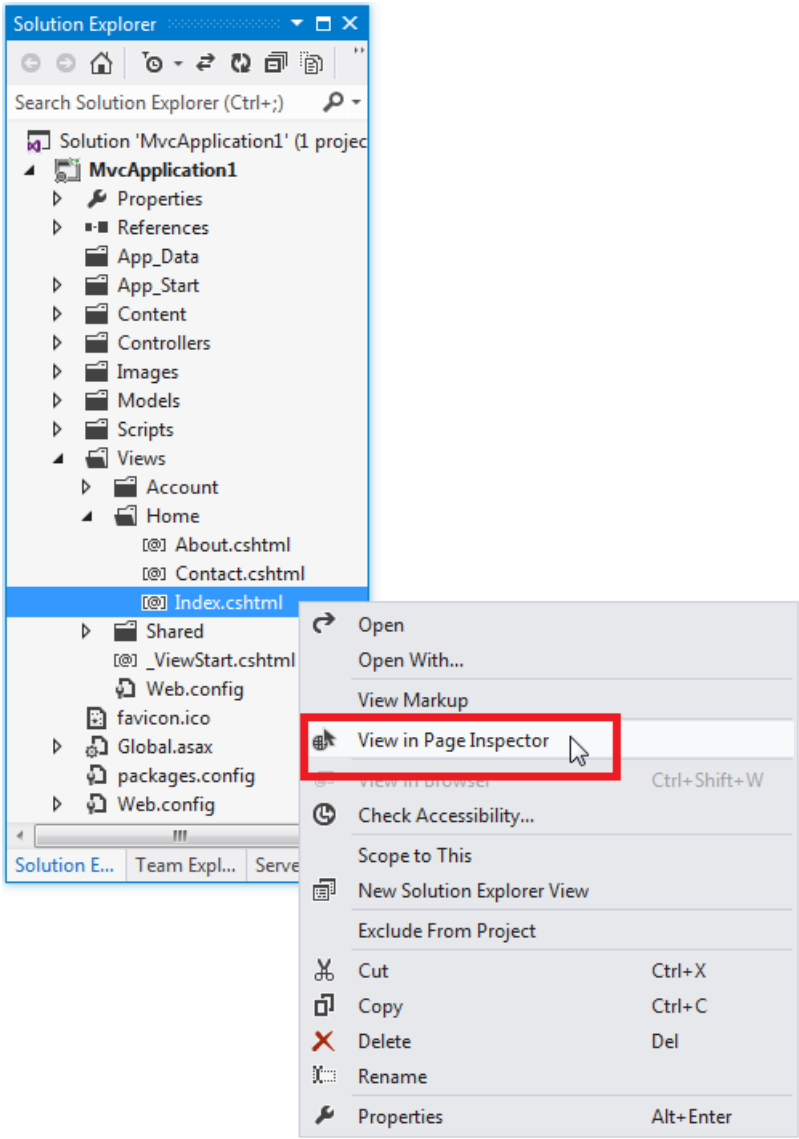


Now that you have an application to work with, you can use Page Inspector to examine and modify it.

Use Page Inspector to Browse to a View

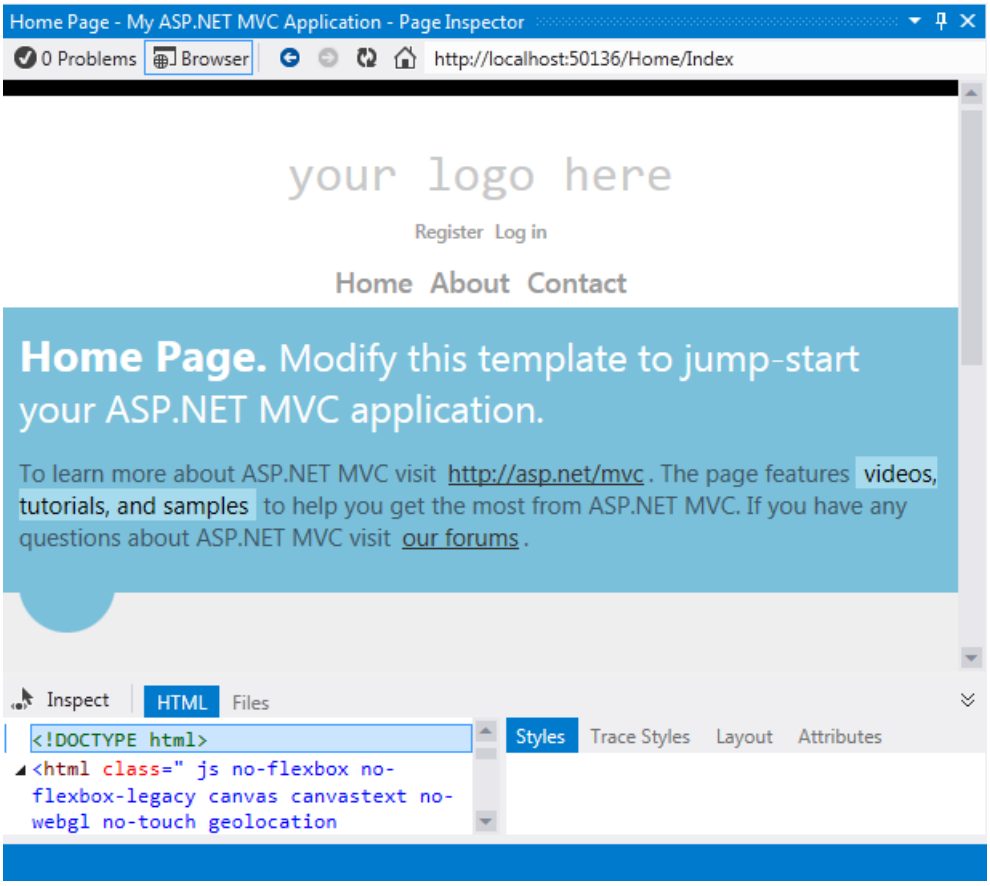
In Visual Studio 2012, you can right-click any view in your project, select **View in Page Inspector**, and Page Inspector will figure out the route and display the page.

In **Solution Explorer**, expand the **Views** folder and then the **Home** folder. Right click the Index.cshtml file and choose **View in Page Inspector**.



By default, Page Inspector is docked as a window on the left side of the Visual Studio environment. If you prefer, you can dock it elsewhere, or undock the window. See How to: Arrange and Dock Windows.

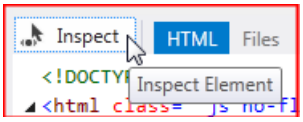
The top pane of the Page Inspector window shows the current page in a browser window. The bottom pane shows the page in HTML markup, along with some tabs that let you inspect different aspects of the page. The bottom pane is similar to the F12 Developer Tools in Internet Explorer.



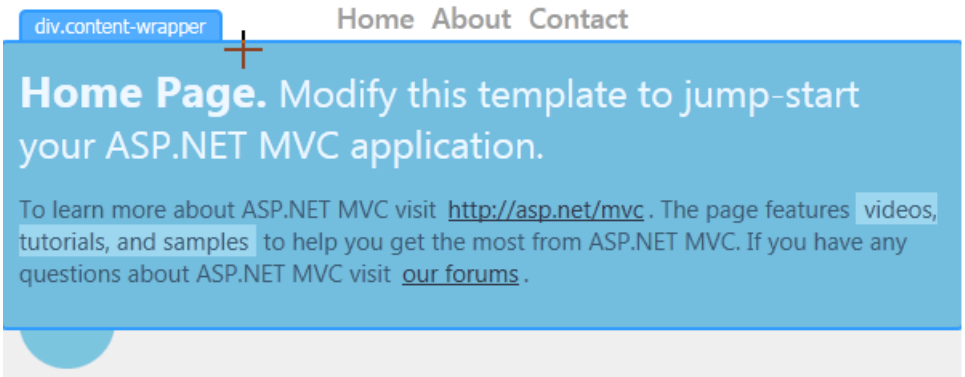
In this tutorial, you will use the **HTML** and **Styles** tabs to navigate quickly and make changes to the application.

EnableInspection Mode

To put Page Inspector into Inspection Mode, click the **Inspect** button. In Inspection Mode, when you hold the mouse pointer over any part of the rendered page, the corresponding source markup or code is highlighted.



Now move your mouse over different parts of the page within Page Inspector. As you do, the mouse pointer changes to a large plus sign, and the element underneath is highlighted:



As you move the mouse pointer, Visual Studio highlights the corresponding Razor syntax in the source file. If the HTML element comes from another source file, Visual Studio automatically opens the file.

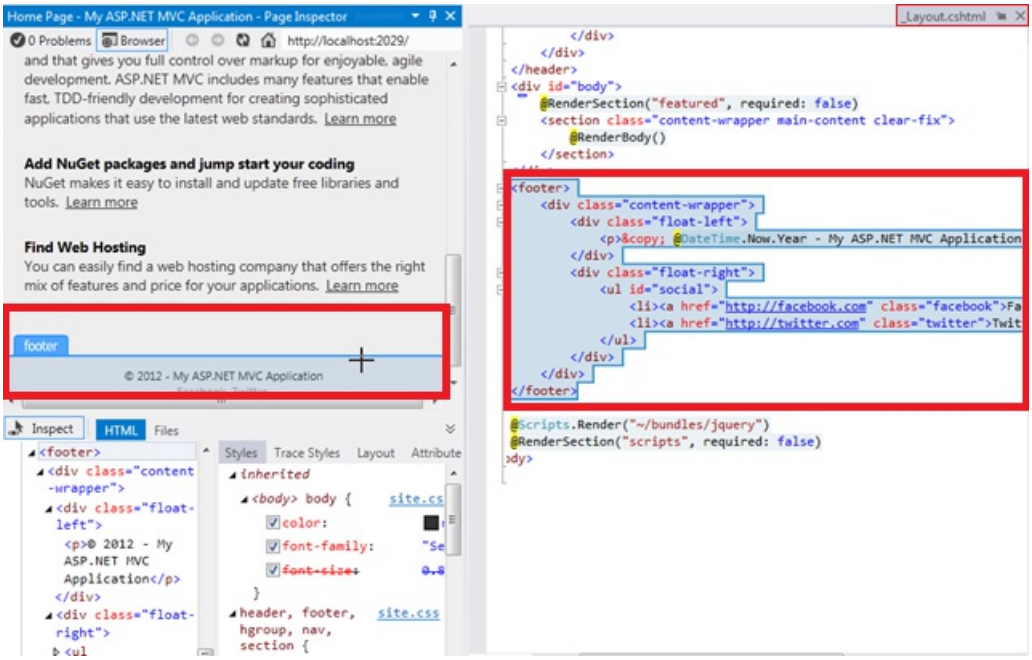
In Page Inspector, the **HTML** tab shows the HTML that was generated from the Razor syntax. As you move the mouse pointer, the HTML elements are highlighted. The **Styles** tab shows the CSS rules for the element.

Use Page Inspector to Make Changes to Markup

Page Inspector lets you find markup whose location might not be obvious. Then you can modify the markup and see the resulting changes.

To see this, click **Inspect** and then scroll to the bottom of the page in the Page Inspector window.

When you move the mouse pointer into the footer area, Page Inspector opens the _Layout.cshtml file and highlights the section of the layout page that you have selected. As you can see, the footer area is defined in the layout file, and not the view itself.



Now move your mouse pointer over the line with the copyright notice. In the _Layout.cshtml page, the corresponding line is highlighted.

```
<p>&copy; @DateTime.Now.Year - My ASP.NET MVC Application</p>
```

Add some text to the end of the line in the _Layout.cshtml file.

© @DateTime.Now.Year - My ASP.NET MVC Application Rocks!

Now, press Ctrl+Alt+Enter or click the Update Bar to see the results in the Page Inspector browser window.

© 2012 - My ASP.NET MVC Application Rocks!

You might have thought that the footer defined in Index.cshtml, but it turned out to be in the _Layout.cshtml, and Page Inspector found it for you.

Inspection Mode and the HTML Window

Next, you will have a quick look at the HTML window and how it maps elements for you.

Click **Inspect** to put Page Inspector in Inspection Mode.

Click the top part of the page that says "Your logo here". You are examining a particular element in more detail, so the display in the browser window no longer changes as you move the mouse pointer.

Now move the mouse pointer to the **HTML** window. As you move the mouse pointer, Page Inspector outlines the element within the **HTML** window and highlights the corresponding element in the browser window.

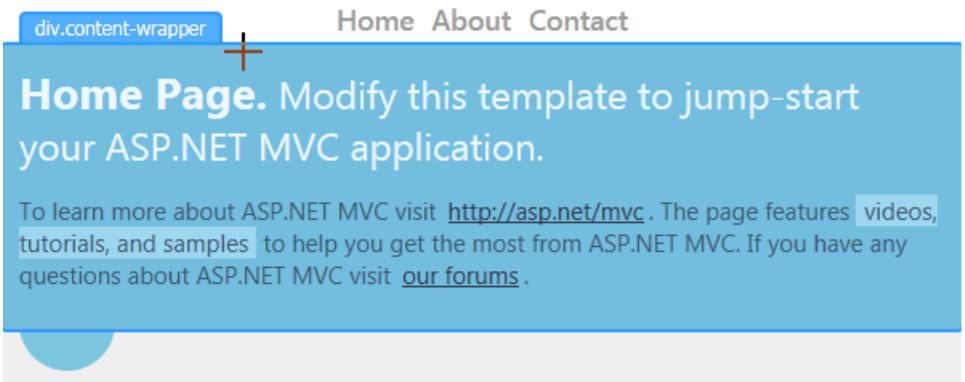
As before, Page Inspector opens the _Layout.cshtml file for you in a temporary tab. Click the _Layout.cshtml temporary tab, and the corresponding markup will be highlighted in the <header> section for you:

Preview CSS Changes in the Styles Window

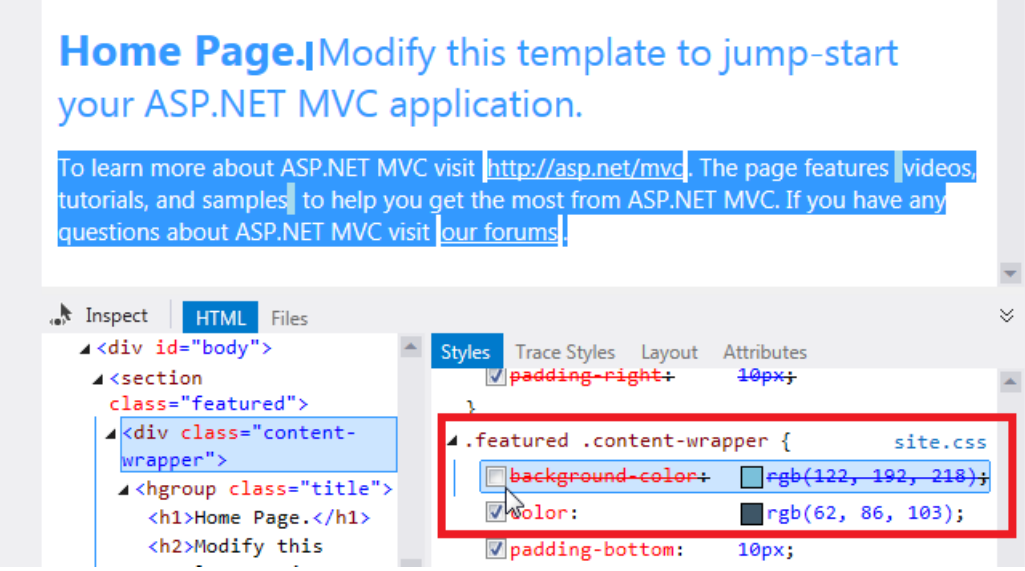
Next, you will use the Page Inspector **Styles** window to preview changes to CSS.

Click **Inspect** to put Page Inspector in Inspection Mode.

In the Page Inspector browser window, move the mouse pointer over the "Home Page" section until the **div.content-wrapper** label appears.

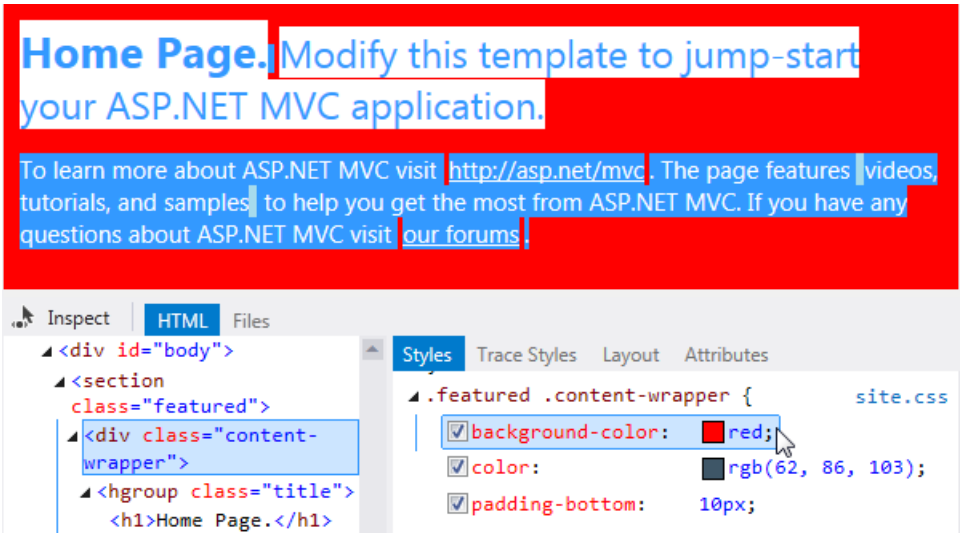


Click within the div.content-wrapper section once, and then move the mouse pointer to the **Styles** window. The **Styles** window shows all of the CSS rules for this element. Scroll down to find the .featured .content-wrapper class selector. Now clear the checkbox for the background-color property.



Notice how the change previews instantly in the Page Inspector browser window.

Select the checkbox again, then double-click the property value and change it to red. The change shows immediately:



The **Styles** window makes it easy to test and preview CSS changes before you commit the changes to the style sheet itself.

CSS Auto Sync

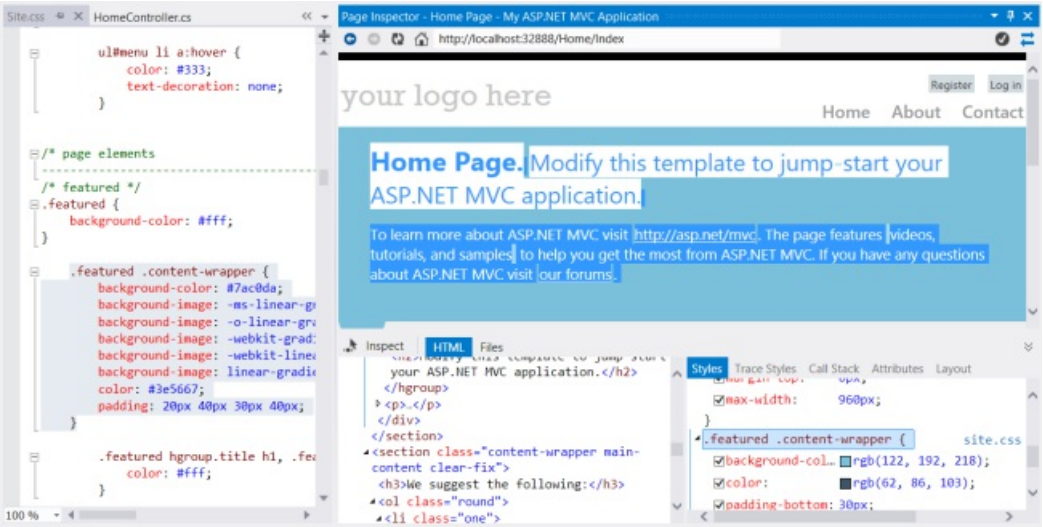
> **NOTE**
>
> This feature requires version 1.3 of Page Inspector.

The CSS Auto-Sync feature allows you to edit a CSS file directly, and see the changes immediately in the Page Inspector browser.

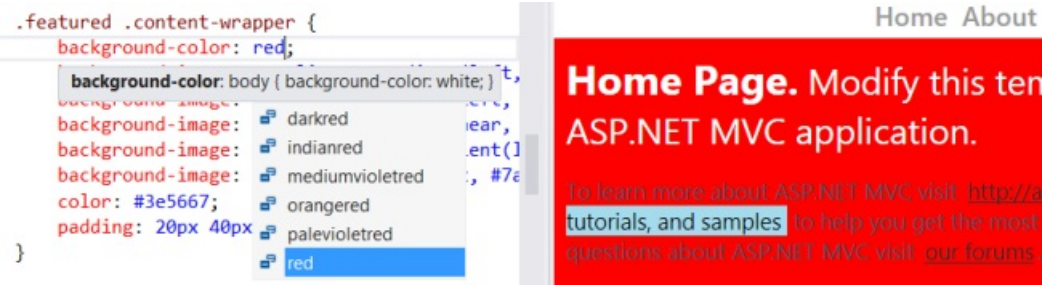
Click **Inspect** to put Page Inspector in Inspection Mode.

In the Page Inspector browser, move the mouse pointer over the "Home Page" section until the **div.content-wrapper** label appears. Click once to select this element.

The **Styles** window shows all of the CSS rules for this element. Scroll down to find the .featured .content-wrapper class selector. Click on ".featured .content-wrapper". Page Inspector opens the CSS file that defines this style (Site.css) and highlights the corresponding CSS style.



Now change the value for `background-color` to "red". The change appears immediately in the Page Inspector browser.

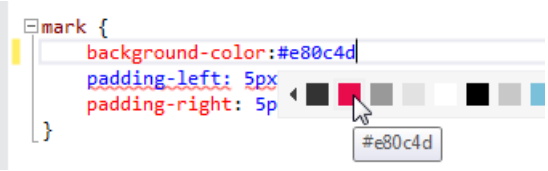


Using the CSS Color Picker

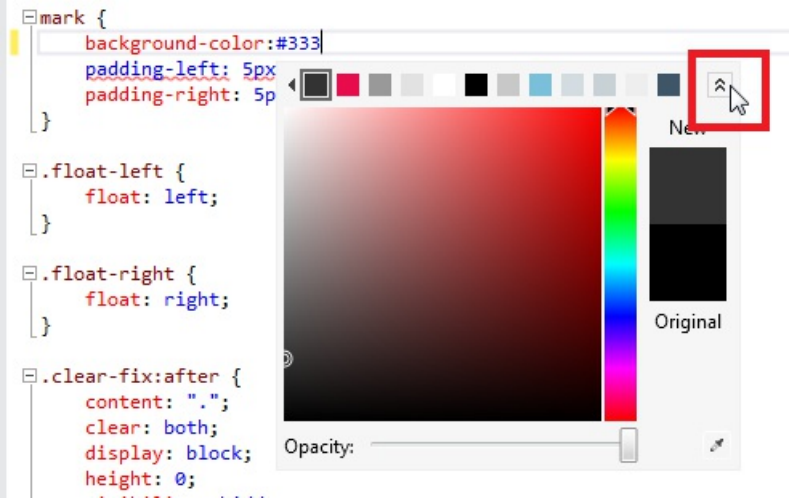
The CSS editor in Visual Studio 2012 has a color picker that makes it easy to choose and insert colors. The color picker includes a standard palette of colors, supports standard color names, hash codes, RGB, RGBA, HSL, and HSLA colors, and maintains a list of the colors you've used most recently in the document.

In the previous section, you changed the value of the `background-color` property. To invoke the color picker, place the insertion point after the property name and type **#** or **rgb(**.

Click on a color to select it, or press the down arrow key and then use the left and right arrow keys to traverse the colors. When you visit a color, the corresponding hex value is previewed:



If the color bar doesn't have the exact color you want, you can use the color picker pop-down. To open it, click the double chevron at the right end of the color bar, or press the Down Arrow once or twice on the keyboard.



Click a color from the vertical bar on the right. This shows a gradient for that color in the main window. Choose a color directly from the vertical bar by pressing Enter, or click any point in the main window to choose with greater precision.

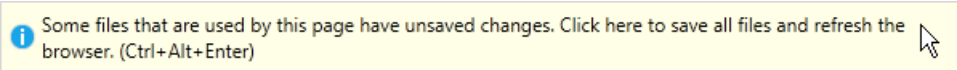
If there is a color on your computer screen that you want to use (it doesn't have to be inside the Visual Studio user interface), you can capture its value by using the eyedropper tool on the lower right.

You also can change the opacity of a color by moving the slider at the bottom of the color picker. Doing so changes color values to RGBA values, because the RGBA format can represent opacity.

After you have chosen a color, press Enter, and then type a semicolon to complete the background-color entry in the *Site.css* file.

The Page Inspector Update Bar

Page Inspector immediately detects the change to the *Site.css* file and displays an alert in an update bar.



To save all your files and refresh the Page Inspector browser, press Ctrl+Alt+Enter or click the update bar. The change in the highlight color appears in the browser.

Mapping Dynamic Page Elements to JavaScript

In modern web applications, elements in the page are often generated dynamically with JavaScript. That means there is no static markup (either HTML or Razor) that corresponds to these page elements.

With version 1.3, Page Inspector can now map items that were dynamically added to the page back to the corresponding JavaScript code. To demonstrate this feature, we'll use the Single Page Application (SPA) template.

> **NOTE**
>
> The SPA template requires the ASP.NET and Web Tools 2012.2 update.

In Visual Studio, choose **File** > **New Project**. On the left, expand **Visual C#**, select **Web**, and then select **ASP.NET MVC4 Web Application**. Click **OK**.

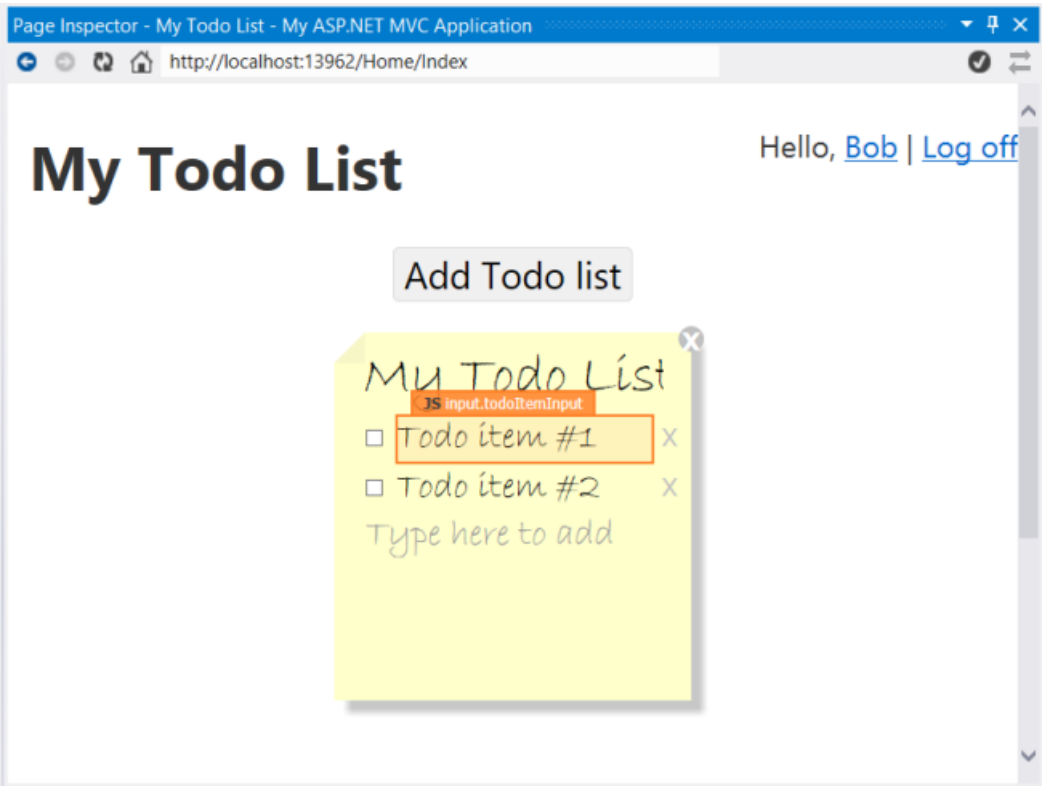
In the **New ASP.NET MVC 4 Project** dialog, select **Single Page Application**.

In Solution Explorer, expand the **Views** folder and then the **Home** folder. Right click the Index.cshtml file and choose **View in Page Inspector**.

The first thing that is displayed in the Page Inspector browser is a login page. Click "Sign Up" and create a user

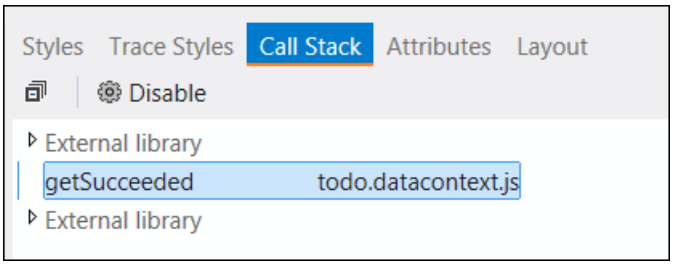
name and password. Once you sign up, the application logs you in and creates a to-do list with some sample items.

Click **Inspect** to put Page Inspector in Inspection Mode. In the Page Inspector browser, click on one of the to-do items. Notice that instead of being highlighted in blue, the element is highlighted in orange, with "JS" next to the element name. This indicates that the element was created dynamically through script.

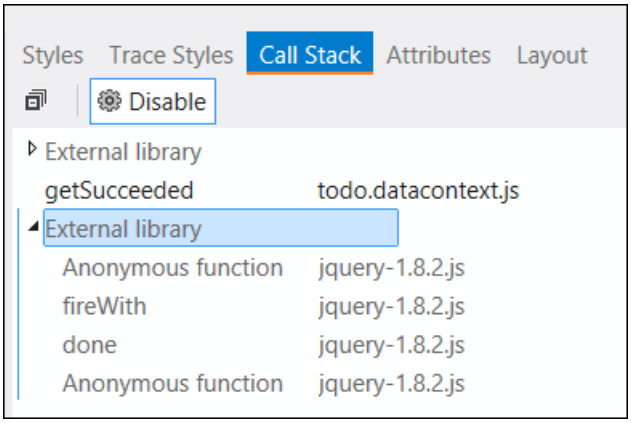


In addition, an orange underline appears on the **Call Stack** tab. This indicates that the **Call Stack** pane has more information about the element.

Click on the **Call Stack** tab. The **Call Stack** pane shows the call stack for the JavaScript call that created the element. Calls to external libraries such as jQuery are collapsed, so that you can easily see the calls to your application script.



To see the full stack, including calls to external libraries, you can expand the nodes labeled "External Libraries":



If you click an item in the call stack, Visual Studio opens the code file and highlights the corresponding script.

```
function getTodoLists(todoListsObservable, errorObservable) {
    return ajaxRequest("get", todoListUrl())
        .done(getSucceeded)
        .fail(getFailed);

    function getSucceeded(data) {
        var mappedTodoLists = $.map(data, function (list) { returr
        todoListsObservable(mappedTodoLists);
    }
```

See Also

Intro to ASP.NET MVC 4 with Visual Studio (ASP.net website)

Introducing Page Inspector (Channel 9 video)

Page Inspector Error Messages (MSDN)

Dynamic v. Strongly Typed Views

9/16/2020 • 2 minutes to read • Edit Online

by Rick Anderson

There are three ways to pass information from a controller to a view in ASP.NET MVC 3:

1. As a strongly typed model object.
2. As a dynamic type (using @model dynamic)
3. Using the ViewBag

I've written a simple MVC 3 Top Blog application to compare and contrast dynamic and strongly typed views. The controller starts out with a simple list of blogs:

```
using System.Collections.Generic;
using System.Web.Mvc;

namespace Mvc3ViewDemo.Controllers {

    public class Blog {
        public string Name;
        public string URL;
    }

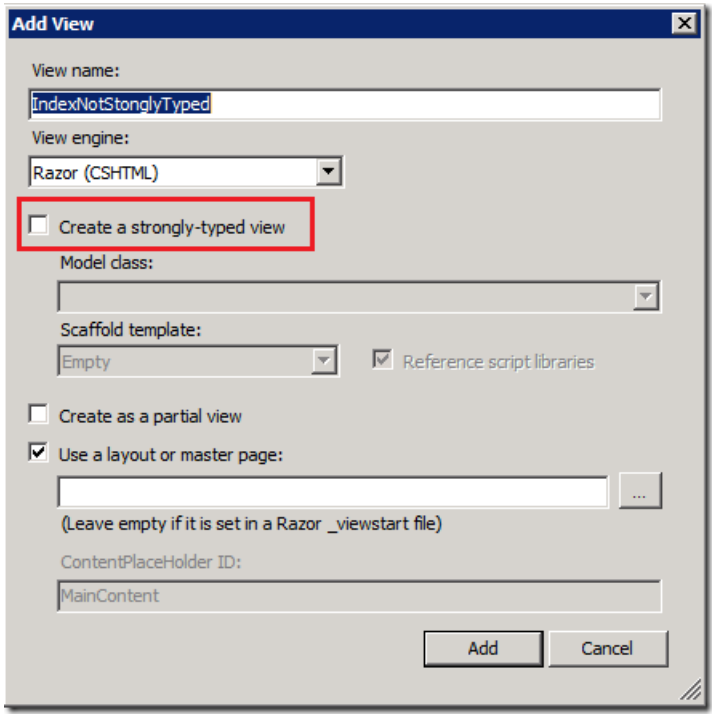
    public class HomeController : Controller {

        List<Blog> topBlogs = new List<Blog>
      {
          new Blog { Name = "ScottGu", URL = "http://weblogs.asp.net/scottgu/"},
          new Blog { Name = "Scott Hanselman", URL = "http://www.hanselman.com/blog/"},
          new Blog { Name = "Jon Galloway", URL = "http://www.asp.net/mvc"}
      };

        public ActionResult IndexNotStonglyTyped() {
            return View(topBlogs);
        }

        public ActionResult About() {
            ViewBag.Message = "Welcome to ASP.NET MVC!";
            return View();
        }
    }
}
```

Right click in the IndexNotStonglyTyped() method and add a Razor view.



Make sure the **Create a strongly-typed view** box is not checked. The resulting view doesn't contain much:

```
@{
    ViewBag.Title = "IndexNotStonglyTyped";
}

<h2>IndexNotStonglyTyped</h2>

On the first line of the Views\Home\IndexNotStonglyTyped.cshtml file, add the model directive and the dynamic
keyword.
```

```
@model dynamic
```

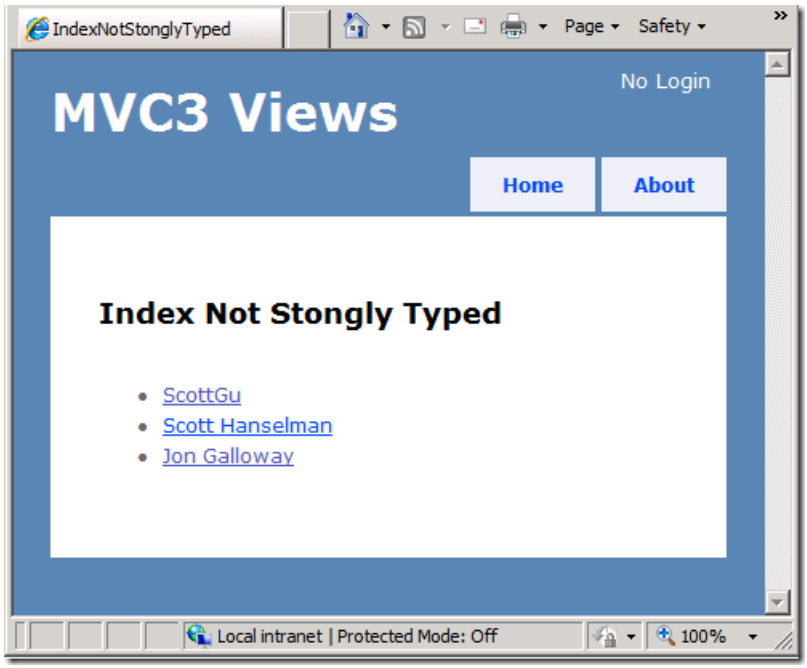
Because we're using a dynamic and not a strongly typed view, intellisense doesn't help us. The completed code is shown below:

```
@model dynamic

@{
    ViewBag.Title = "IndexNotStonglyTyped";
}

<h2>Index Not Stongly Typed</h2>

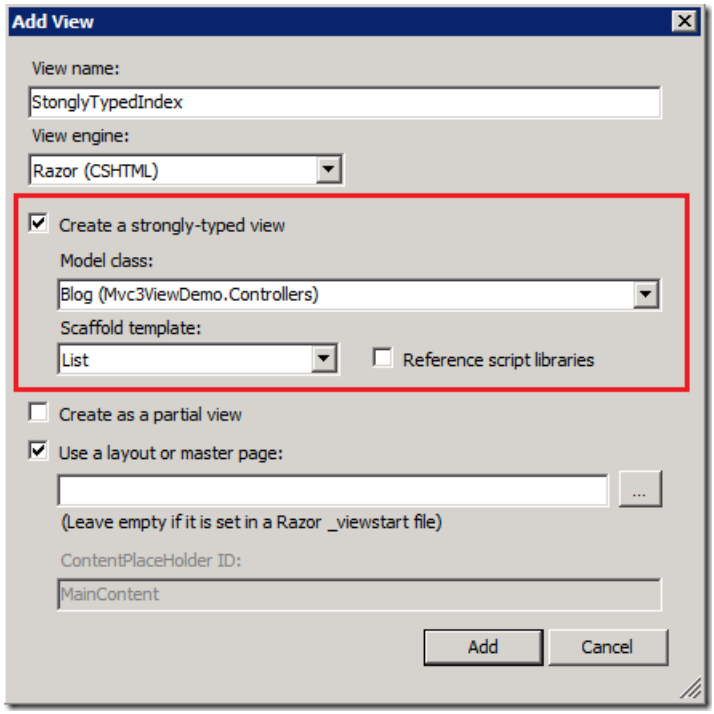
<p>
 <ul>
@foreach (var blog in Model) {
   <li>
    <a href="@blog.URL">@blog.Name</a>
   </li>
}
 </ul>
</p>
```



Now we'll add a strongly typed view. Add the following code to the controller:

```
public ActionResult StonglyTypedIndex() {
    return View(topBlogs);
}
```

Notice it's exactly the same return View(topBlogs); call as the non-strongly typed view. Right click inside of *StonglyTypedIndex()* and select **Add View**. This time select the **Blog** Model class and select **List** as the Scaffold template.



Inside the new view template we get intellisense support.

```
@model IEnumerable<Mvc3ViewDemo.Controllers.Blog>
@{
    ViewBag.Title = "StonglyTypedIndex";
}
<h2> Stongly Typed Index</h2>

<ul>
@foreach (var item in Model) {

    <li><a href="@item.URL">@item.</a> </li>
}
</ul>
```

Equals
GetHashCode
GetType
Name
ToString
URL

Security, Authentication, and Authorization with ASP.NET MVC

2/20/2020 • 2 minutes to read • Edit Online

How to make an ASP.NET site more secure, and how to implement authentication and authorization.

- Create an ASP.NET MVC 5 App with Facebook, Twitter, LinkedIn and Google OAuth2 Sign-on (C#)
- Create a secure ASP.NET MVC 5 web app with log in, email confirmation and password reset (C#)
- ASP.NET MVC 5 app with SMS and email Two-Factor Authentication
- XSRF/CSRF Prevention in ASP.NET MVC and Web Pages
- Preventing Open Redirection Attacks (C#)

Create an ASP.NET MVC 5 App with Facebook, Twitter, LinkedIn and Google OAuth2 Sign-on (C#)

2/20/2020 • 11 minutes to read • Edit Online

by Rick Anderson

This tutorial shows you how to build an ASP.NET MVC 5 web application that enables users to log in using OAuth 2.0 with credentials from an external authentication provider, such as Facebook, Twitter, LinkedIn, Microsoft, or Google. For simplicity, this tutorial focuses on working with credentials from Facebook and Google.

Enabling these credentials in your web sites provides a significant advantage because millions of users already have accounts with these external providers. These users may be more inclined to sign up for your site if they do not have to create and remember a new set of credentials.

See also ASP.NET MVC 5 app with SMS and email Two-Factor Authentication.

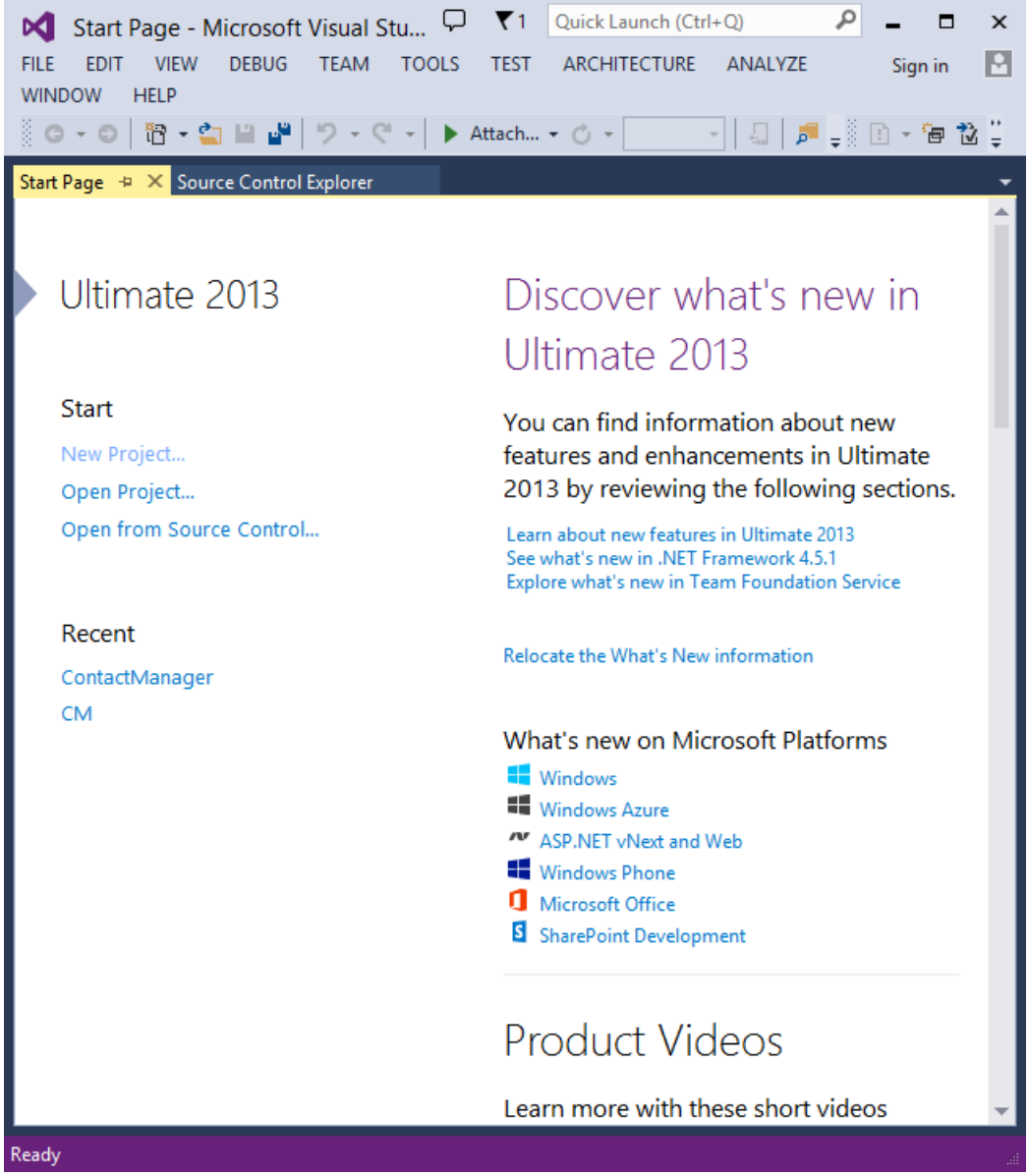
The tutorial also shows how to add profile data for the user, and how to use the Membership API to add roles. This tutorial was written by Rick Anderson (Please follow me on Twitter: @RickAndMSFT).

Getting Started

Start by installing and running Visual Studio Express 2013 for Web or Visual Studio 2013. Install Visual Studio 2013 Update 3 or higher. For help with Dropbox, GitHub, Linkedin, Instagram, Buffer, Salesforce, STEAM, Stack Exchange, Tripit, Twitch, Twitter, Yahoo!, and more, see this sample project.

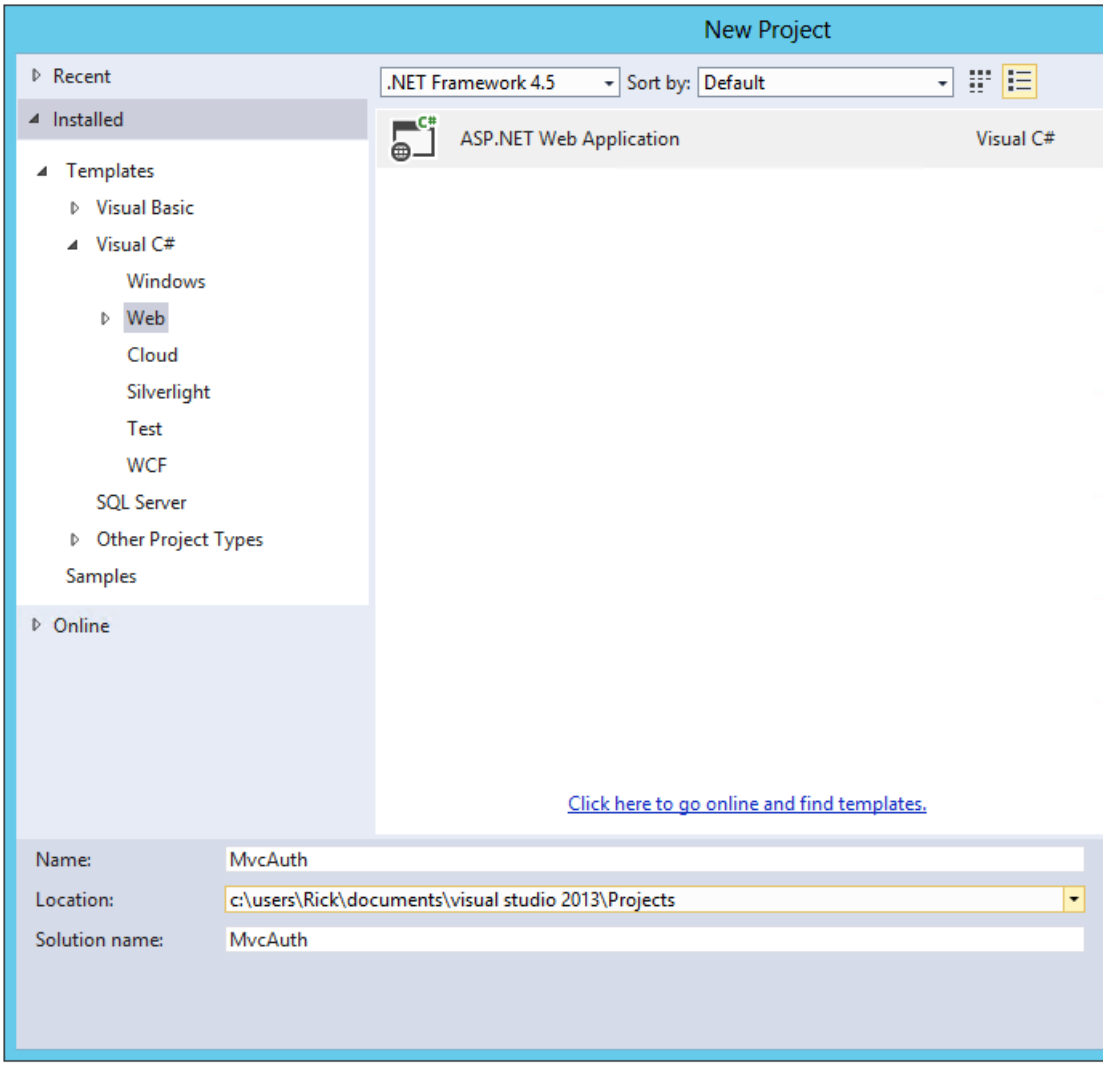
> **NOTE**
>
> You must install Visual Studio 2013 Update 3 or higher to use Google OAuth 2 and to debug locally without SSL warnings.

Click **New Project** from the **Start** page, or you can use the menu and select **File**, and then **New Project**.

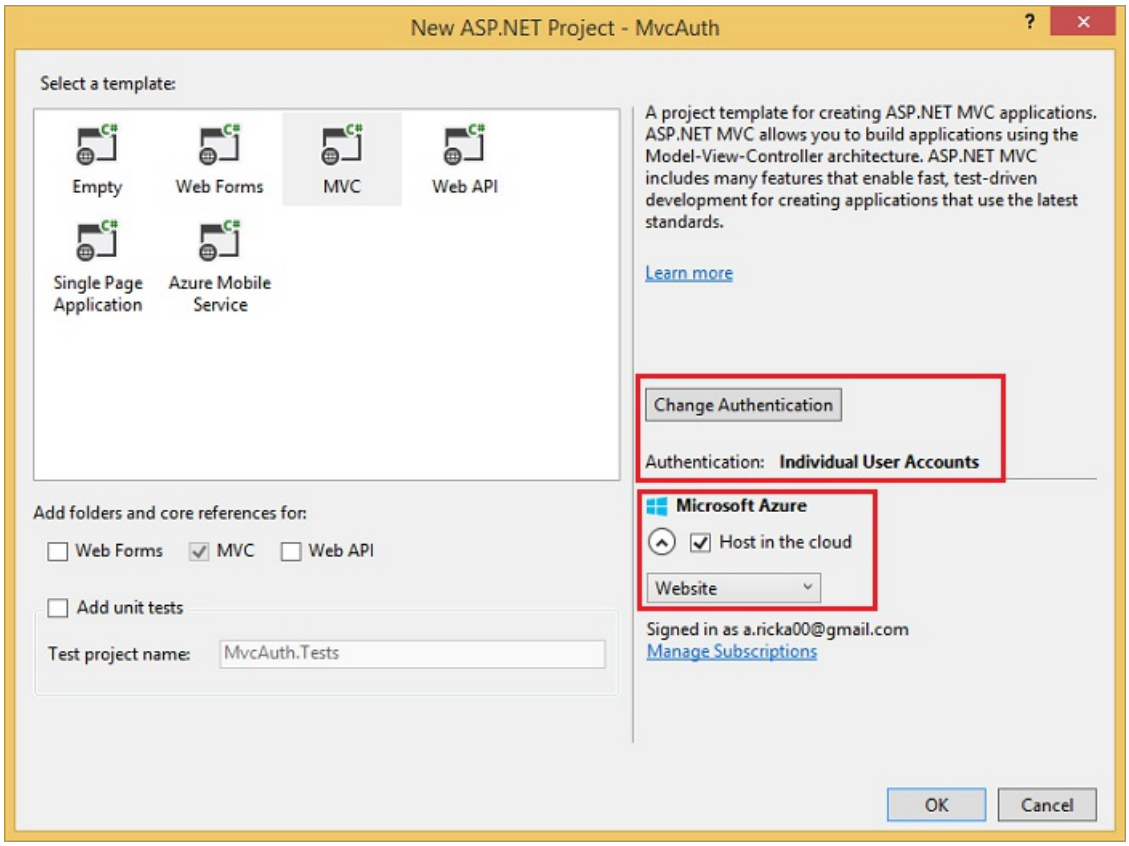


Creating Your First Application

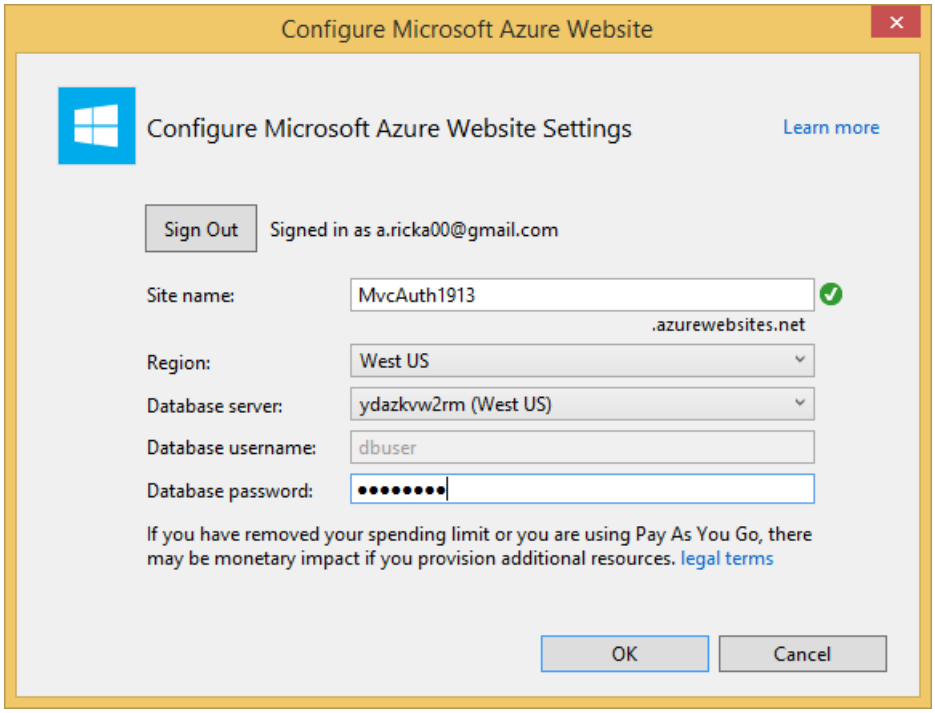
Click **New Project**, then select **Visual C#** on the left, then **Web** and then select **ASP.NET Web Application**. Name your project "MvcAuth" and then click **OK**.



In the **New ASP.NET Project** dialog, click **MVC**. If the Authentication is not **Individual User Accounts**, click the **Change Authentication** button and select **Individual User Accounts**. By checking **Host in the cloud**, the app will be very easy to host in Azure.

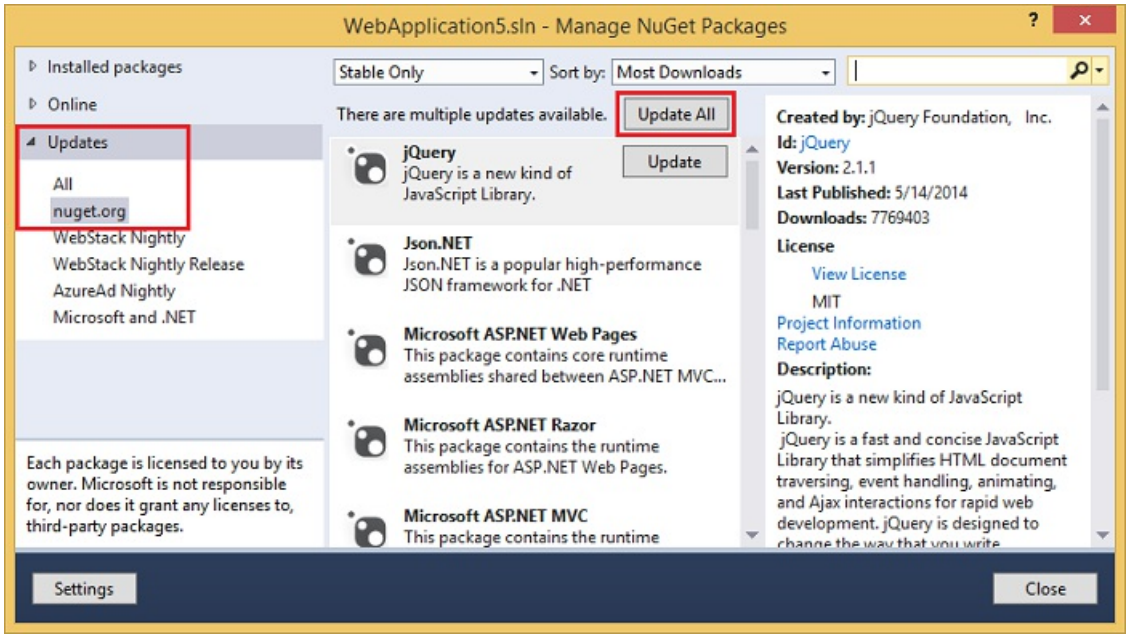


If you selected **Host in the cloud**, complete the configure dialog.

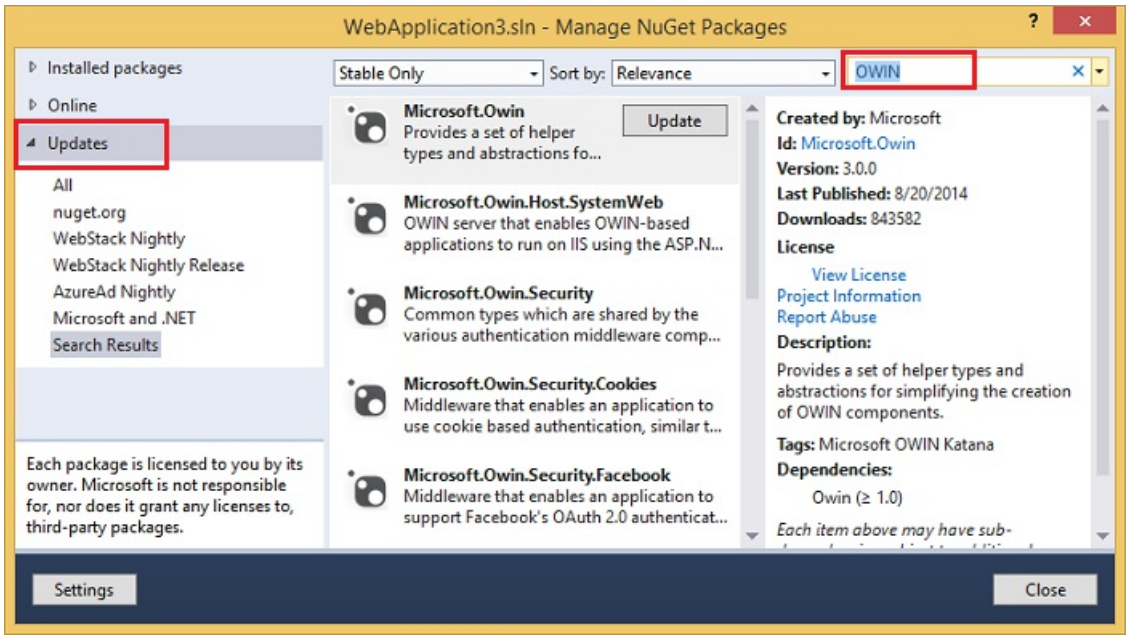


Use NuGet to update to the latest OWIN middleware

Use the NuGet package manager to update the OWIN middleware. Select **Updates** in the left menu. You can click on the **Update All** button or you can search for only OWIN packages (shown in the next image):



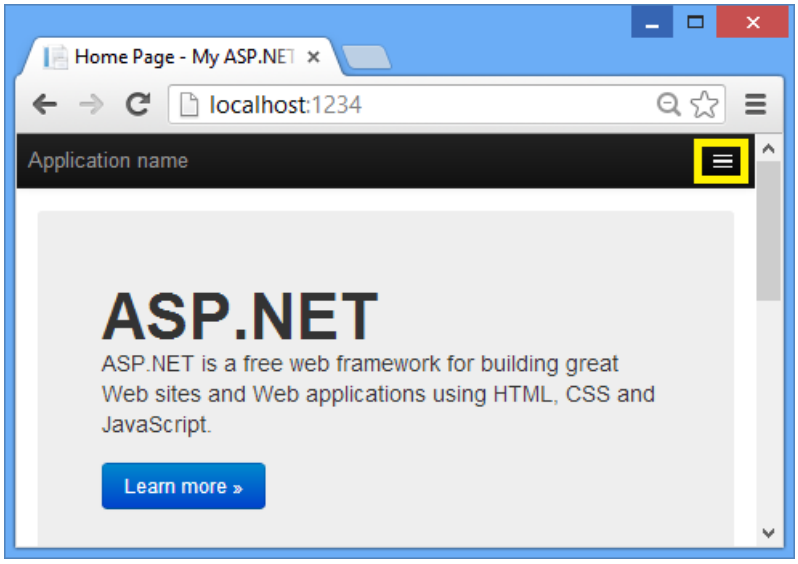
In the image below, only OWIN packages are shown:

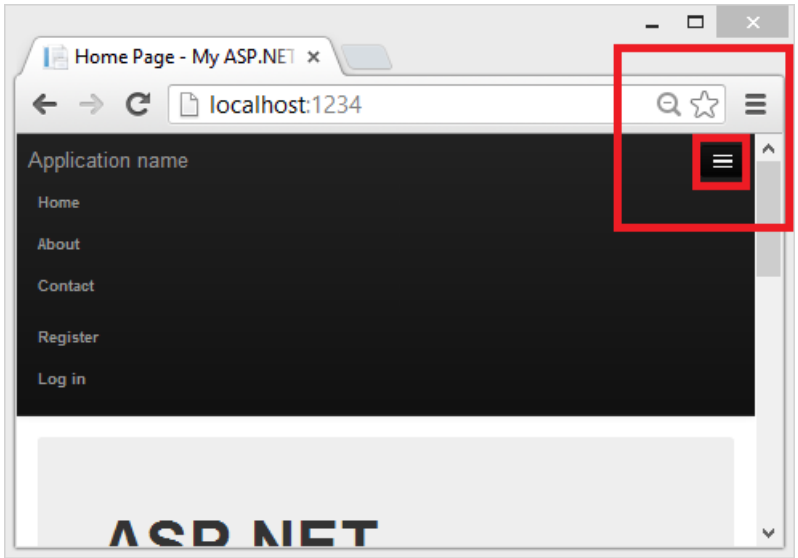


From the Package Manager Console (PMC), you can enter the `Update-Package` command, which will update all packages.

Press **F5** or **Ctrl+F5** to run the application. In the image below, the port number is 1234. When you run the application, you'll see a different port number.

Depending on the size of your browser window, you might need to click the navigation icon to see the **Home**, **About**, **Contact**, **Register** and **Log in** links.

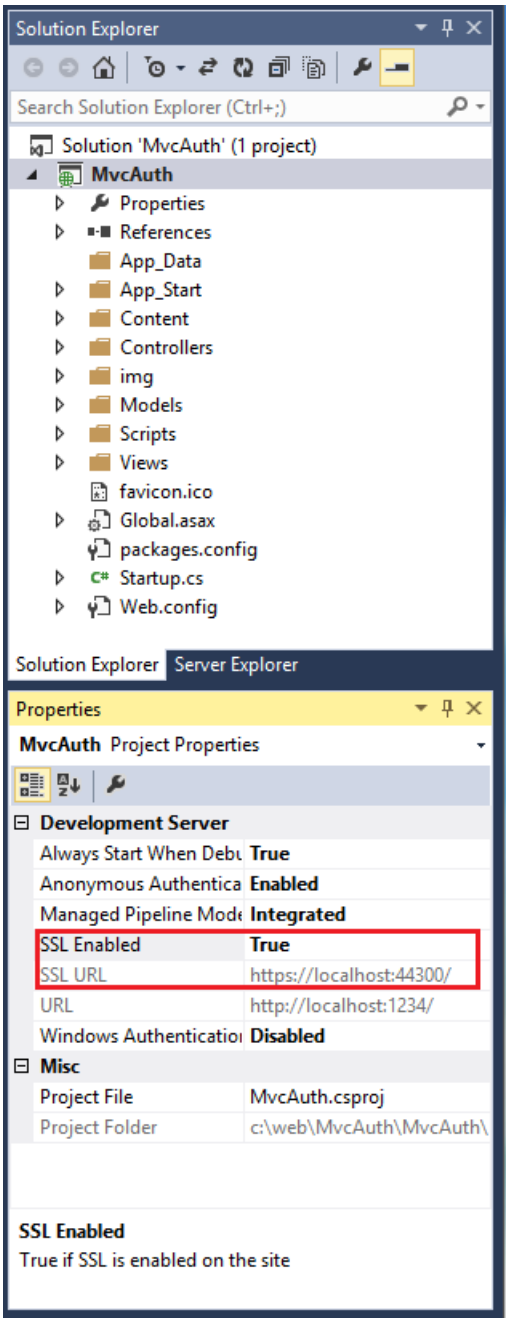




Setting up SSL in the Project

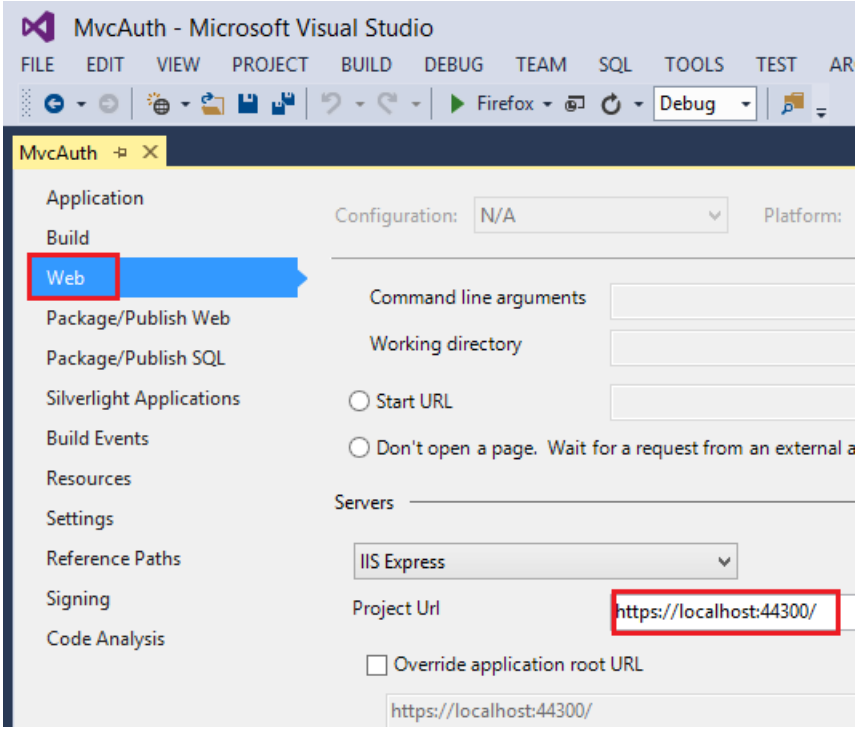
To connect to authentication providers like Google and Facebook, you will need to set up IIS-Express to use SSL. It's important to keep using SSL after login and not drop back to HTTP, your login cookie is just as secret as your username and password, and without using SSL you're sending it in clear-text across the wire. Besides, you've already taken the time to perform the handshake and secure the channel (which is the bulk of what makes HTTPS slower than HTTP) before the MVC pipeline is run, so redirecting back to HTTP after you're logged in won't make the current request or future requests much faster.

1. In **Solution Explorer**, click the **MvcAuth** project.
2. Hit the F4 key to show the project properties. Alternatively, from the **View** menu you can select **Properties Window**.
3. Change **SSL Enabled** to True.



4. Copy the SSL URL (which will be `https://localhost:44300/` unless you've created other SSL projects).
5. In **Solution Explorer**, right click the **MvcAuth** project and select **Properties**.

6. Select the **Web** tab, and then paste the SSL URL into the **Project Url** box. Save the file (Ctl+S). You will need this URL to configure Facebook and Google authentication apps.

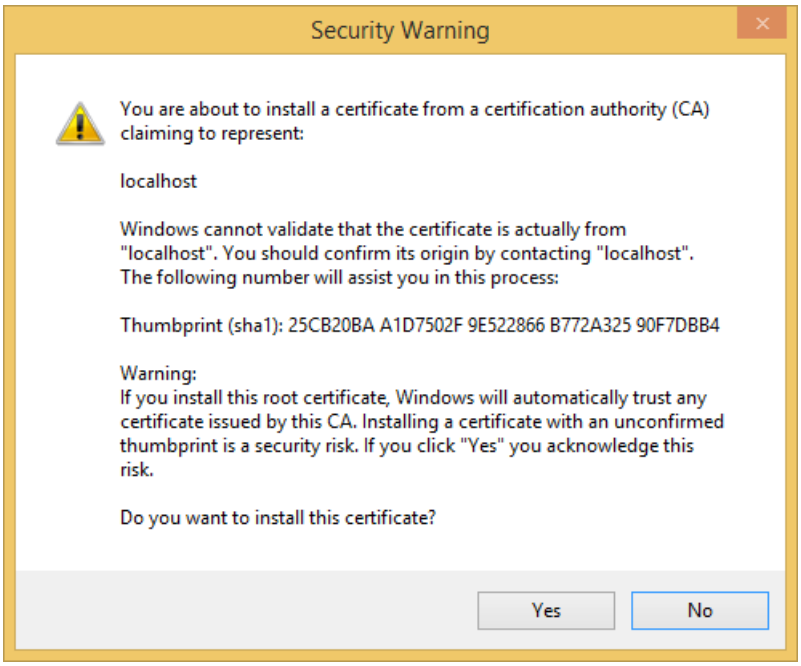


7. Add the RequireHttps attribute to the `Home` controller to require all requests must use HTTPS. A more secure approach is to add the RequireHttps filter to the application. See the section "Protect the Application with SSL and the Authorize Attribute" in my tutorial Create an ASP.NET MVC app with auth and SQL DB and deploy to Azure App Service. A portion of the Home controller is shown below.

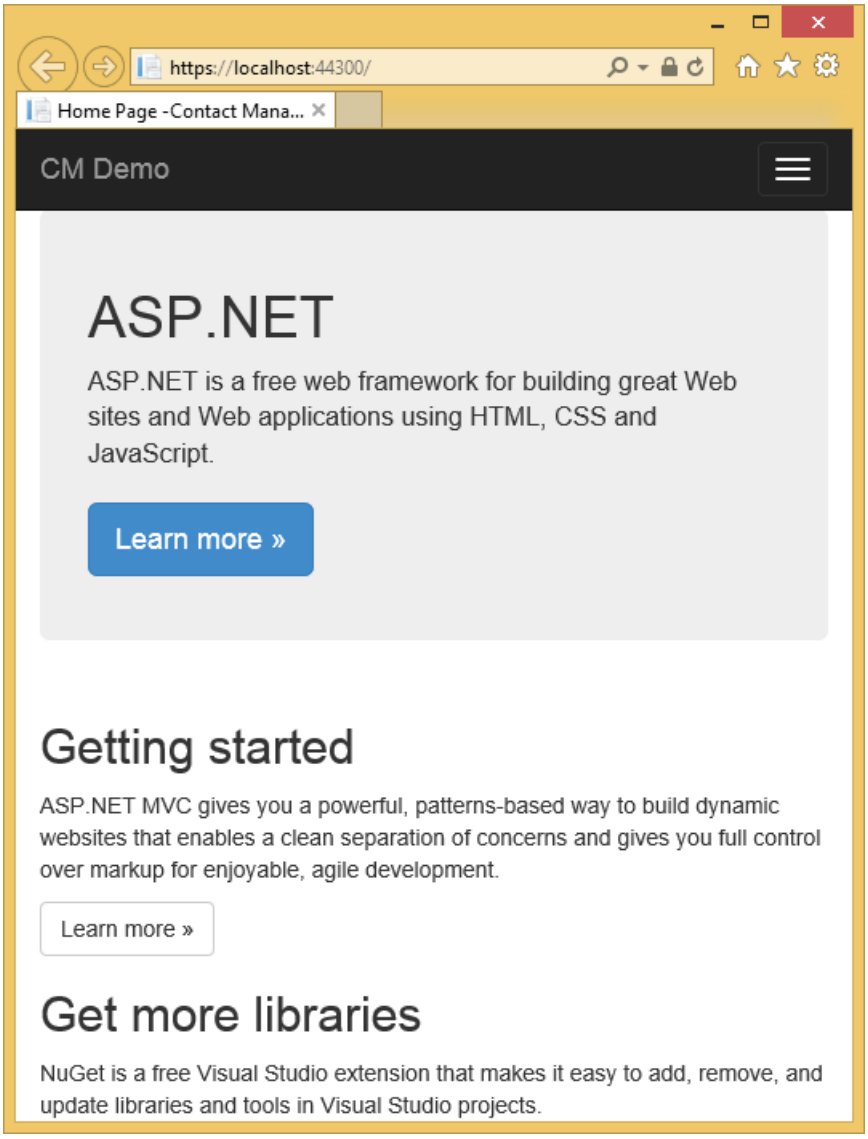
```
[RequireHttps]
public class HomeController : Controller
{
   public ActionResult Index()
   {
      return View();
   }
```

8. Press CTRL+F5 to run the application. If you've installed the certificate in the past, you can skip the rest of this section and jump to Creating a Google app for OAuth 2 and connecting the app to the project, otherwise, follow the instructions to trust the self-signed certificate that IIS Express has generated.

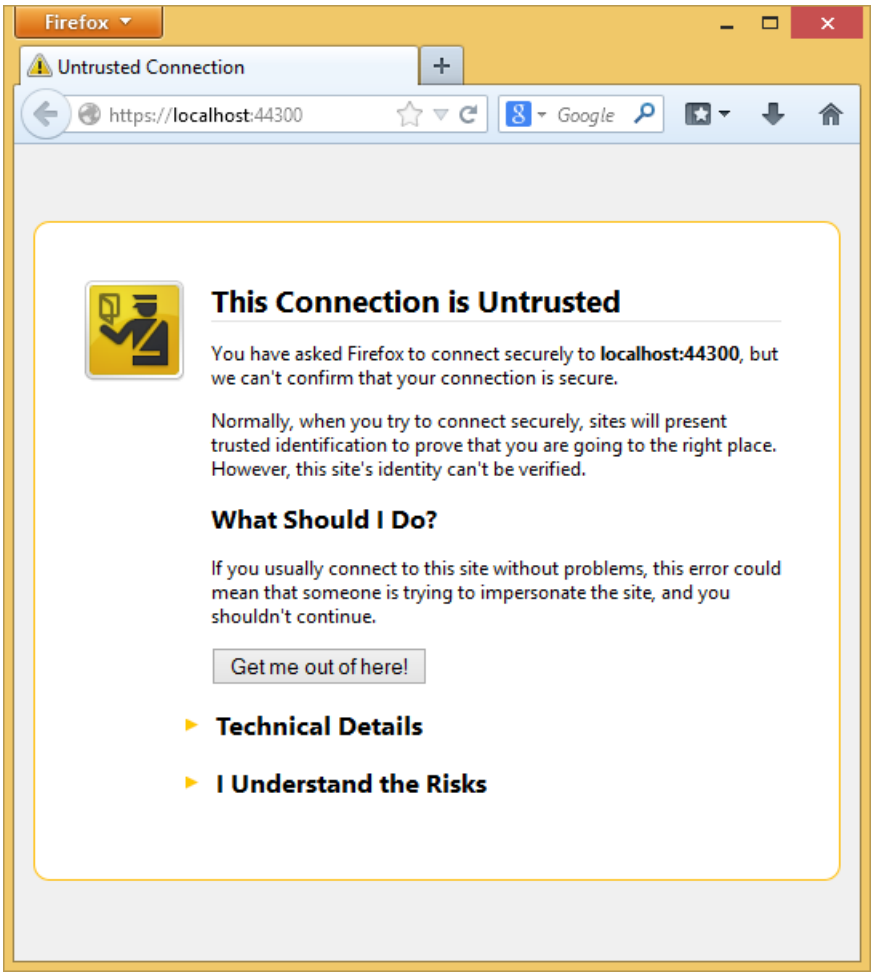
9. Read the **Security Warning** dialog and then click **Yes** if you want to install the certificate representing localhost.



10. IE shows the *Home* page and there are no SSL warnings.



11. Google Chrome also accepts the certificate and will show HTTPS content without a warning. Firefox uses its own certificate store, so it will display a warning. For our application you can safely click **I Understand the Risks**.



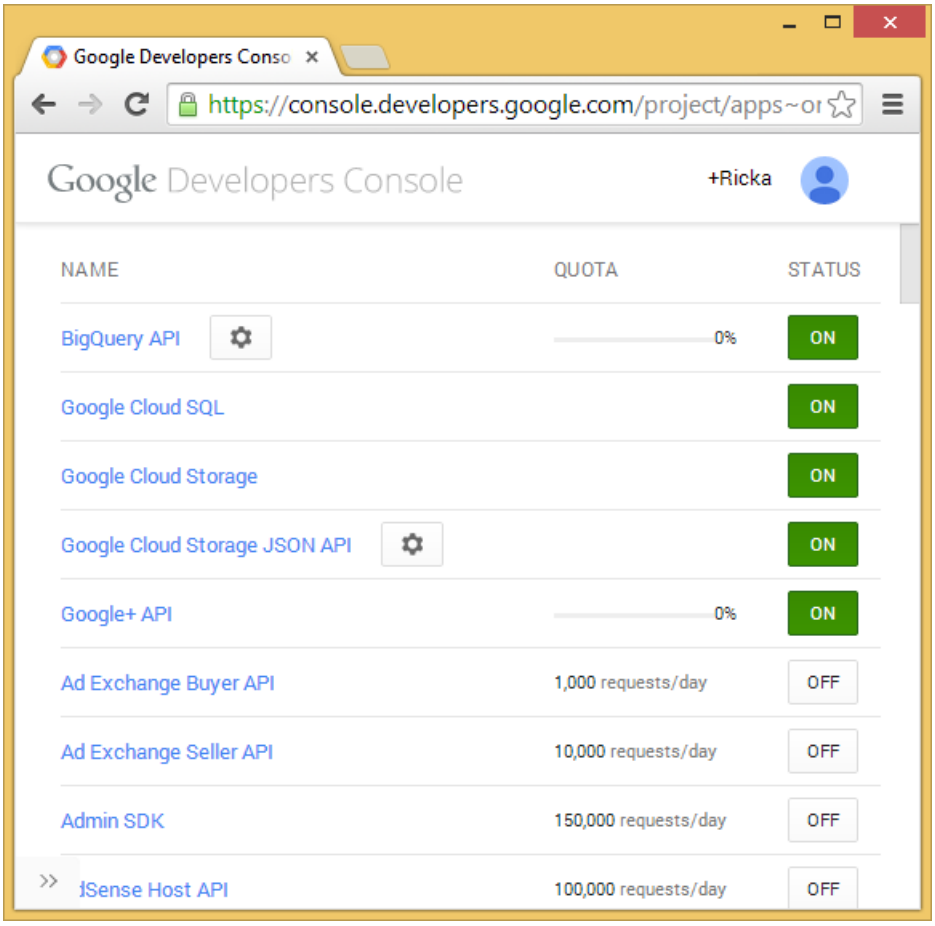
Creating a Google app for OAuth 2 and connecting the app to the project

> **WARNING**
>
> For current Google OAuth instructions, see Configuring Google authentication in ASP.NET Core.

1. Navigate to the Google Developers Console.
2. If you haven't created a project before, select **Credentials** in the left tab, and then select **Create**.
3. In the left tab, click **Credentials**.
4. Click **Create credentials** then **OAuth client ID**.
   a. In the **Create Client ID** dialog, keep the default **Web application** for the application type.
   b. Set the **Authorized JavaScript** origins to the SSL URL you used above (`https://localhost:44300/` unless you've created other SSL projects)
   c. Set the **Authorized redirect URI** to: `https://localhost:44300/signin-google`
5. Click the OAuth Consent screen menu item, then set your email address and product name. When you have completed the form click **Save**.
6. Click the Library menu item, search **Google+ API**, click on it then press Enable.

Google Spectrum Database API	1,000 requests/day	OFF
Google Webmaster Tools API	1,000,000 requests/day	OFF
Google+ API	10,000 requests/day	OFF
Google+ Domains API	10,000 requests/day	OFF
Google+ Hangouts API		OFF
Groups Migration API	500,000 requests/day	OFF
Groups Settings API	100,000 requests/day	OFF
Identity Toolkit API	1,000,000 requests/day	OFF

The image below shows the enabled APIs.



7. From the Google APIs API Manager, visit the **Credentials** tab to obtain the **Client ID**. Download to save a JSON file with application secrets. Copy and paste the **ClientId** and **ClientSecret** into the `UseGoogleAuthentication` method found in the *Startup.Auth.cs* file in the *App_Start* folder. The **ClientId** and **ClientSecret** values shown below are samples and don't work.

```
public void ConfigureAuth(IAppBuilder app)
{
    // Configure the db context and user manager to use a single instance per request
    app.CreatePerOwinContext(ApplicationDbContext.Create);
    app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Create);

    // Enable the application to use a cookie to store information for the signed in user
    // and to use a cookie to temporarily store information about a user logging in with a third
party login provider
    // Configure the sign in cookie
    app.UseCookieAuthentication(new CookieAuthenticationOptions
    {
        AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
        LoginPath = new PathString("/Account/Login"),
        Provider = new CookieAuthenticationProvider
        {
            OnValidateIdentity = SecurityStampValidator.OnValidateIdentity<ApplicationUserManager,
ApplicationUser>(
                validateInterval: TimeSpan.FromMinutes(30),
                regenerateIdentity: (manager, user) => user.GenerateUserIdentityAsync(manager))
        }
    });

    app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);

    // Uncomment the following lines to enable logging in with third party login providers
    //app.UseMicrosoftAccountAuthentication(
    //    clientId: "",
    //    clientSecret: "");

    //app.UseTwitterAuthentication(
    //   consumerKey: "",
    //   consumerSecret: "");

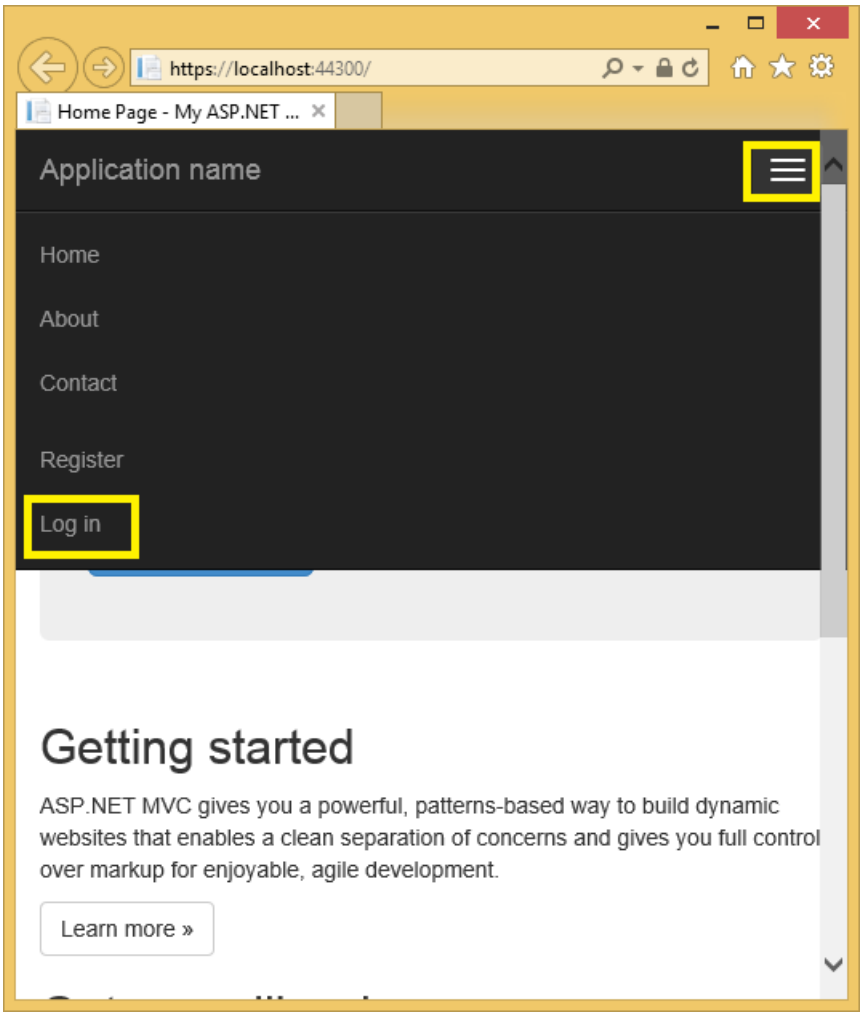
    //app.UseFacebookAuthentication(
    //   appId: "",
    //   appSecret: "");

    app.UseGoogleAuthentication(
         clientId: "000-000.apps.googleusercontent.com",
         clientSecret: "00000000000");
}
```

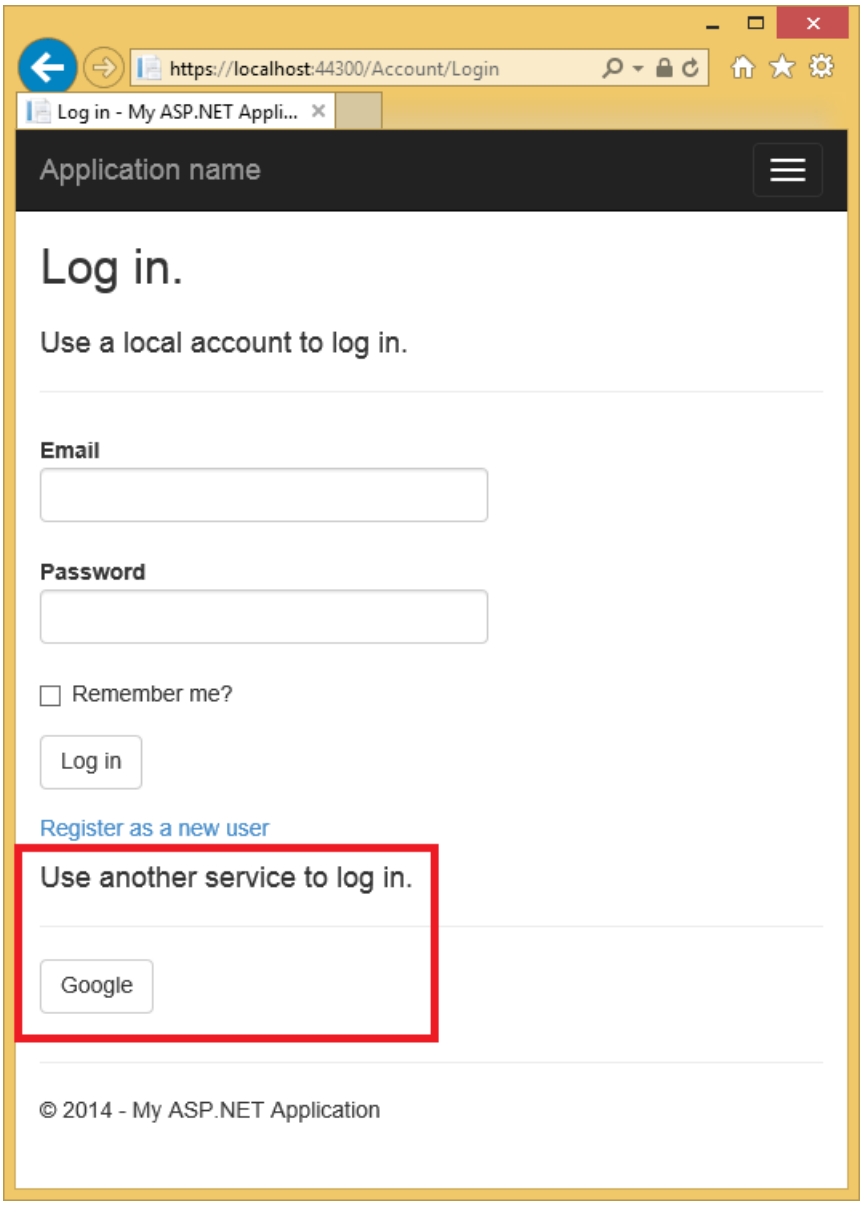
WARNING

Security - Never store sensitive data in your source code. The account and credentials are added to the code above to keep the sample simple. See Best practices for deploying passwords and other sensitive data to ASP.NET and Azure App Service.

8. Press **CTRL+F5** to build and run the application. Click the **Log in** link.

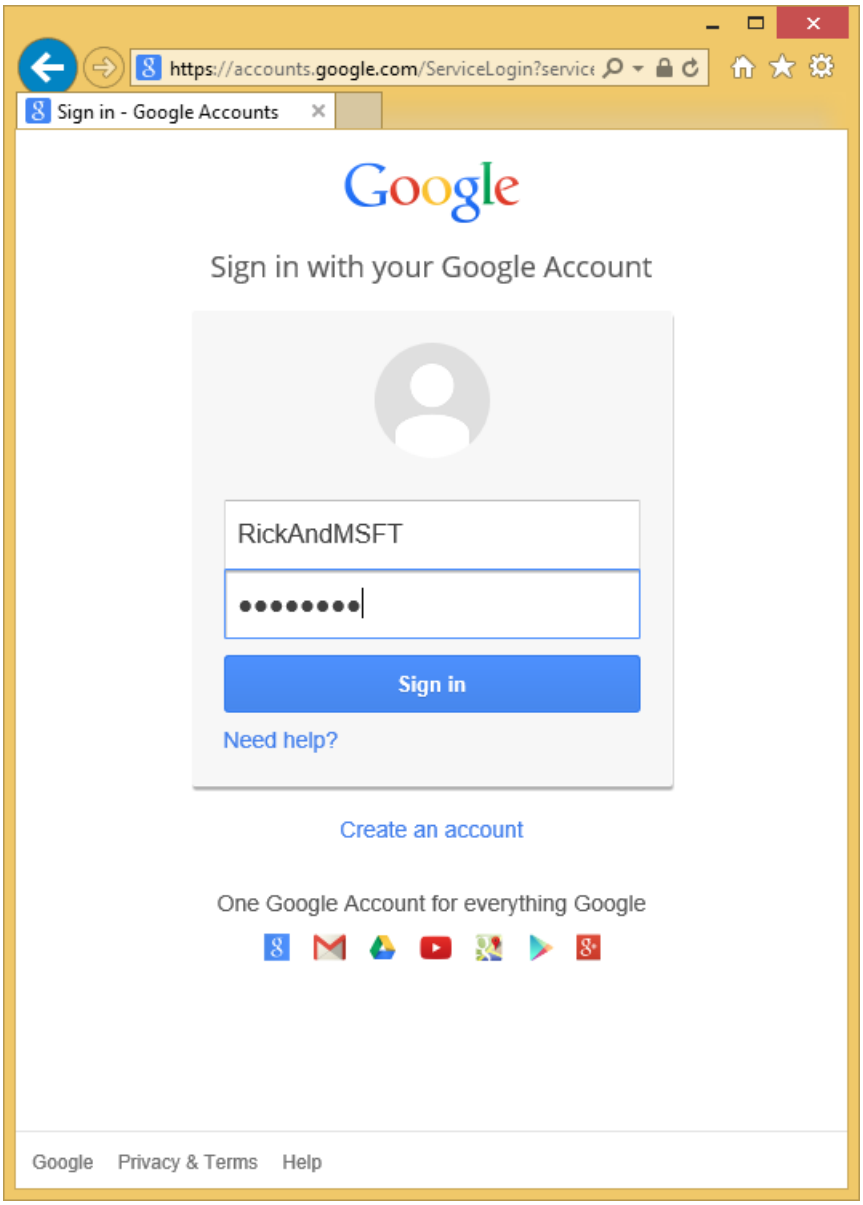


9. Under **Use another service to log in**, click **Google**.

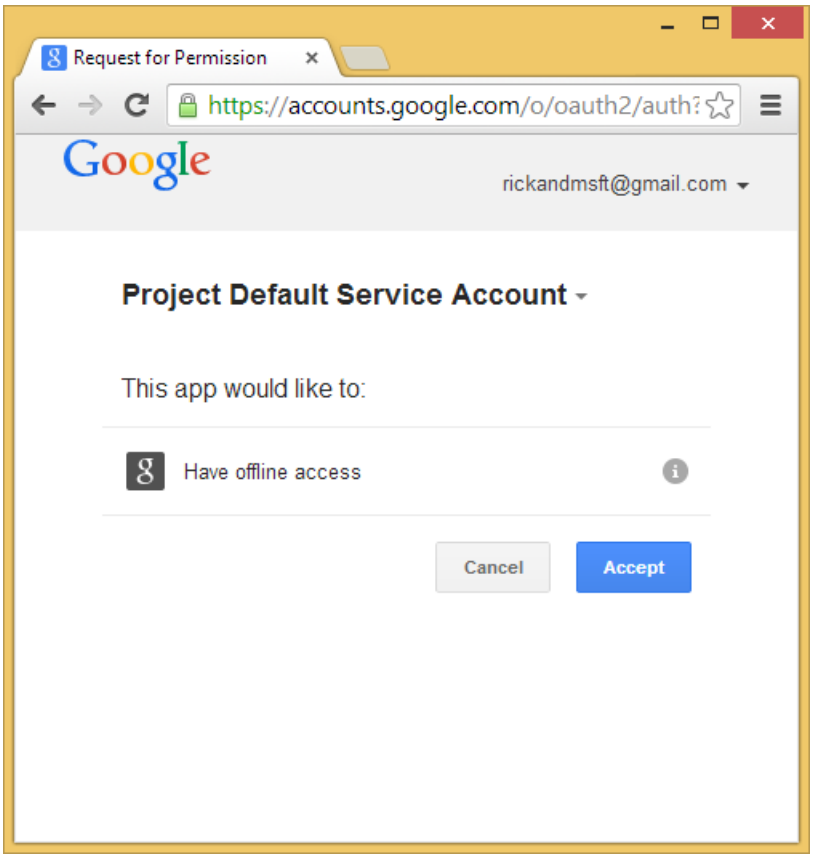


> **NOTE**
>
> If you miss any of the steps above you will get a HTTP 401 error. Recheck your steps above. If you miss a required setting (for example **product name**), add the missing item and save; it can take a few minutes for authentication to work.

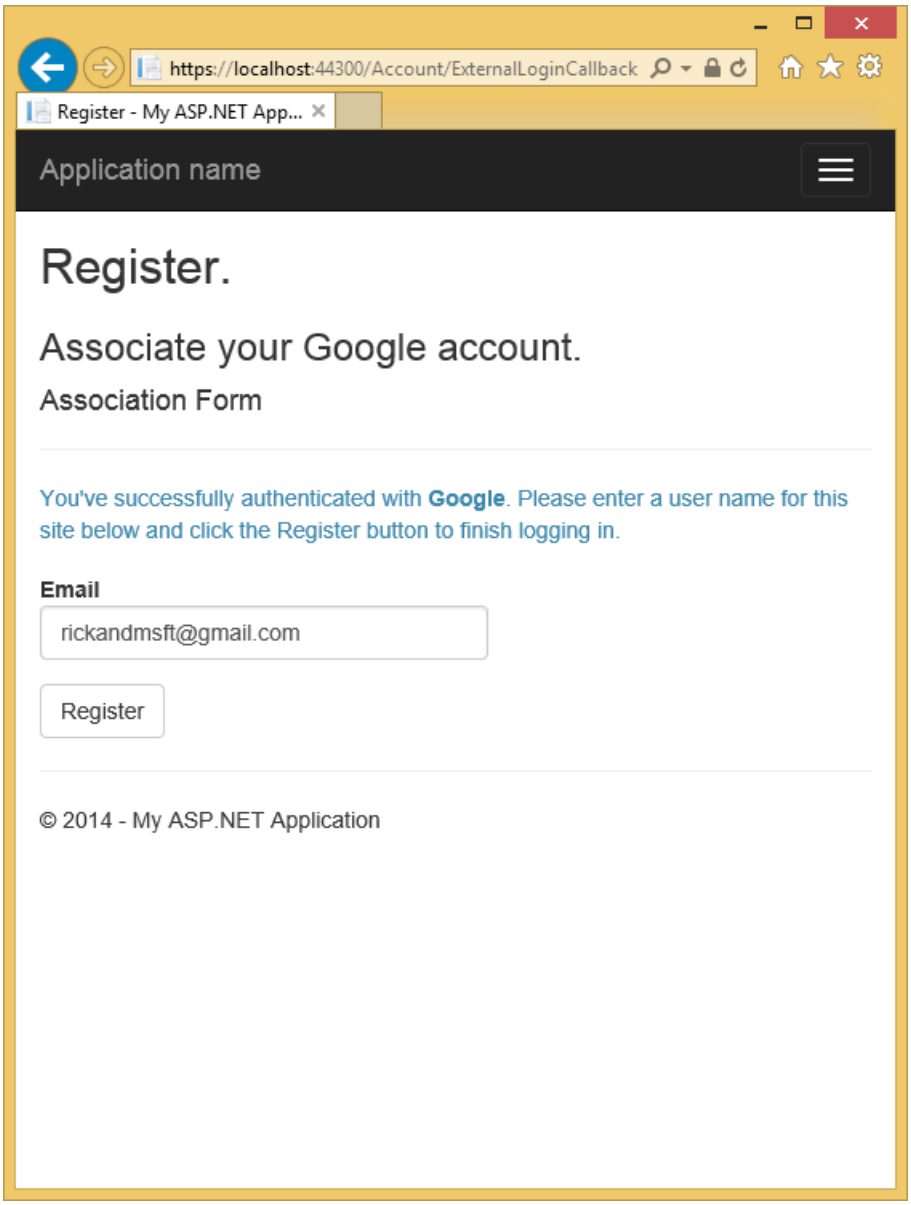
10. You will be redirected to the Google site where you will enter your credentials.



11. After you enter your credentials, you will be prompted to give permissions to the web application you just created:



12. Click **Accept**. You will now be redirected back to the **Register** page of the MvcAuth application where you can register your Google account. You have the option of changing the local email registration name used for your Gmail account, but you generally want to keep the default email alias (that is, the one you used for authentication). Click **Register**.

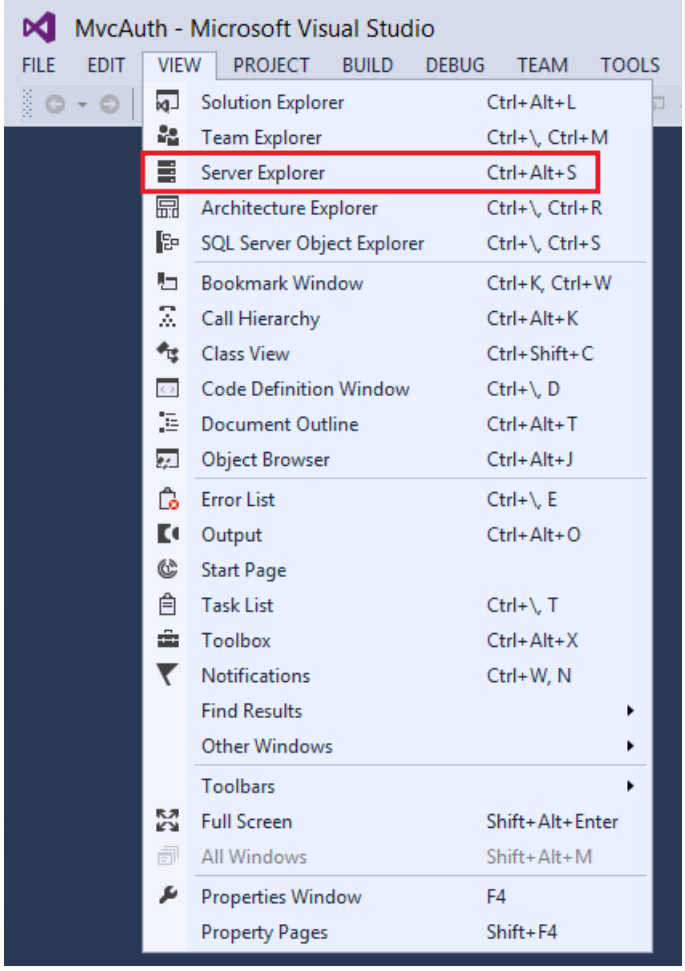


Creating the app in Facebook and connecting the app to the project

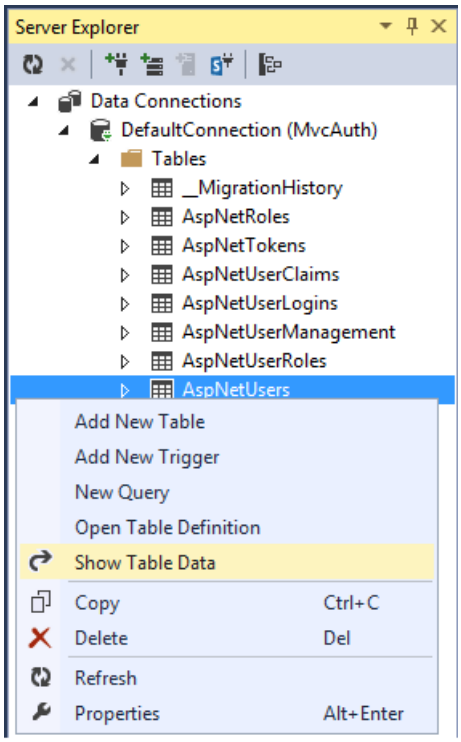
> **WARNING**
>
> For current Facebook OAuth2 authentication instructions, see Configuring Facebook authentication

Examine the Membership Data

In the **View** menu, click **Server Explorer**.



Expand **DefaultConnection (MvcAuth)**, expand **Tables**, right click **AspNetUsers** and click **Show Table Data**.



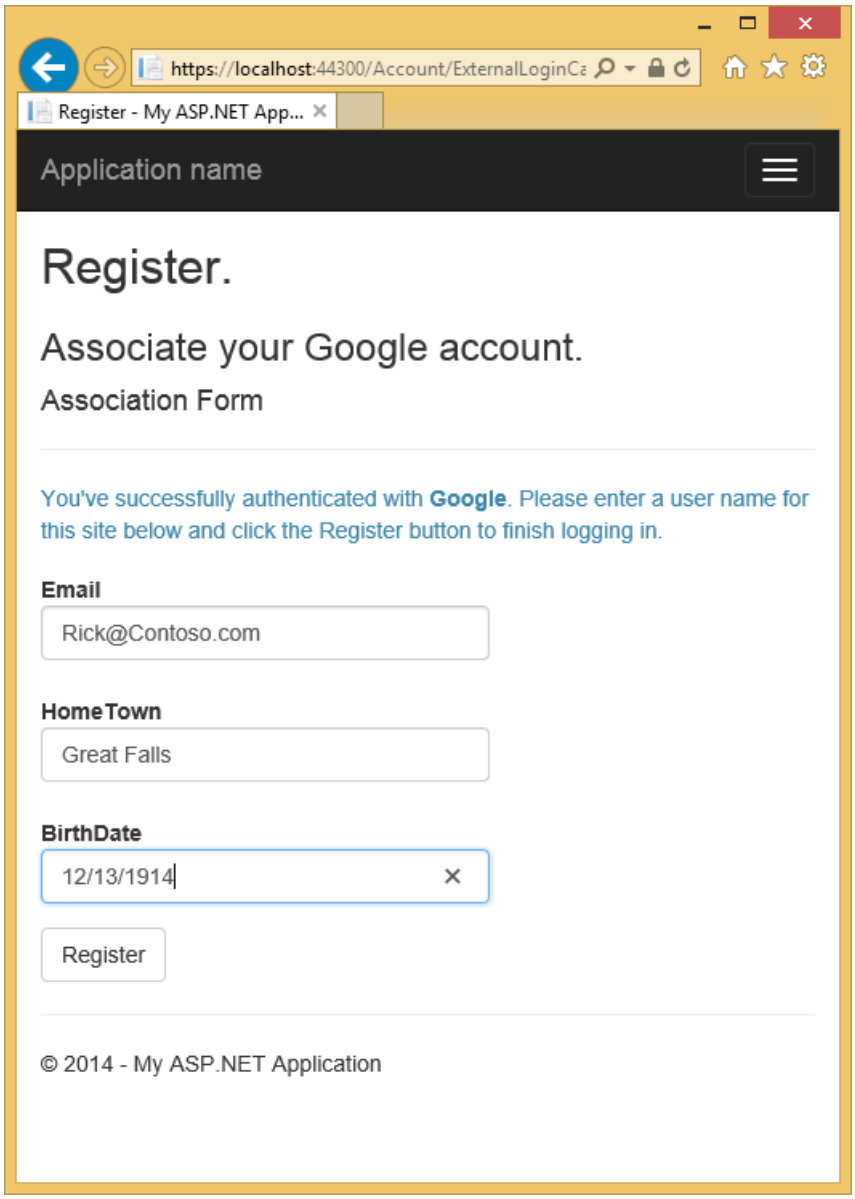
dbo.AspNetUsers [Data]

Max Rows: 1000

Id	Email	EmailConfirmed	PasswordHash	SecurityStamp	PhoneNumber
3878-475f87d6efa0	rick.FB@contoso.com	False	NULL	4fb9fed9-0028-...	NULL
727d3bc0-74cf-...	rickandmsft@gmail.com	False	NULL	228178d3-d11f-...	NULL
NULL	NULL	NULL	NULL	NULL	NULL

Adding Profile Data to the User Class

In this section you'll add birth date and home town to the user data during registration, as shown in the following image.



Open the *Models\IdentityModels.cs* file and add birth date and home town properties:

```
public class ApplicationUser : IdentityUser
{
    public string HomeTown { get; set; }
    public System.DateTime? BirthDate { get; set; }
    public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser> manager)
    {
        // Note the authenticationType must match the one defined in
CookieAuthenticationOptions.AuthenticationType
        var userIdentity = await manager.CreateIdentityAsync(this,
DefaultAuthenticationTypes.ApplicationCookie);
        // Add custom user claims here
        return userIdentity;
    }
}
```

Open the *Models\AccountViewModels.cs* file and the set birth date and home town properties in `ExternalLoginConfirmationViewModel`.

```
public class ExternalLoginConfirmationViewModel
{
    [Required]
    [EmailAddress]
    [Display(Name = "Email")]
    public string Email { get; set; }

    public string HomeTown { get; set; }
    public System.DateTime? BirthDate { get; set; }
}
```

Open the *Controllers\AccountController.cs* file and add code for birth date and home town in the `ExternalLoginConfirmation` action method as shown:

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> ExternalLoginConfirmation(ExternalLoginConfirmationViewModel model, string
returnUrl)
{
    if (User.Identity.IsAuthenticated)
    {
        return RedirectToAction("Manage");
    }

    if (ModelState.IsValid)
    {
        // Get the information about the user from the external login provider
        var info = await AuthenticationManager.GetExternalLoginInfoAsync();
        if (info == null)
        {
            return View("ExternalLoginFailure");
        }
        var user = new ApplicationUser()
        {
            UserName = model.Email, Email = model.Email,
            BirthDate = model.BirthDate,
            HomeTown  = model.HomeTown

        };
        IdentityResult result = await UserManager.CreateAsync(user);
        if (result.Succeeded)
        {
            result = await UserManager.AddLoginAsync(user.Id, info.Login);
            if (result.Succeeded)
            {
                await SignInAsync(user, isPersistent: false);

                // For more information on how to enable account confirmation and password reset please
visit https://go.microsoft.com/fwlink/?LinkID=320771
                // Send an email with this link
                // string code = await UserManager.GenerateEmailConfirmationTokenAsync(user.Id);
                // var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code =
code }, protocol: Request.Url.Scheme);
                // SendEmail(user.Email, callbackUrl, "Confirm your account", "Please confirm your account
by clicking this link");

                return RedirectToLocal(returnUrl);
            }
        }
        AddErrors(result);
    }

    ViewBag.ReturnUrl = returnUrl;
    return View(model);
}
```

Add birth date and home town to the *Views\Account\ExternalLoginConfirmation.cshtml* file:

```
@model MvcAuth.Models.ExternalLoginConfirmationViewModel
@{
    ViewBag.Title = "Register";
}
<h2>@ViewBag.Title.</h2>
<h3>Associate your @ViewBag.LoginProvider account.</h3>

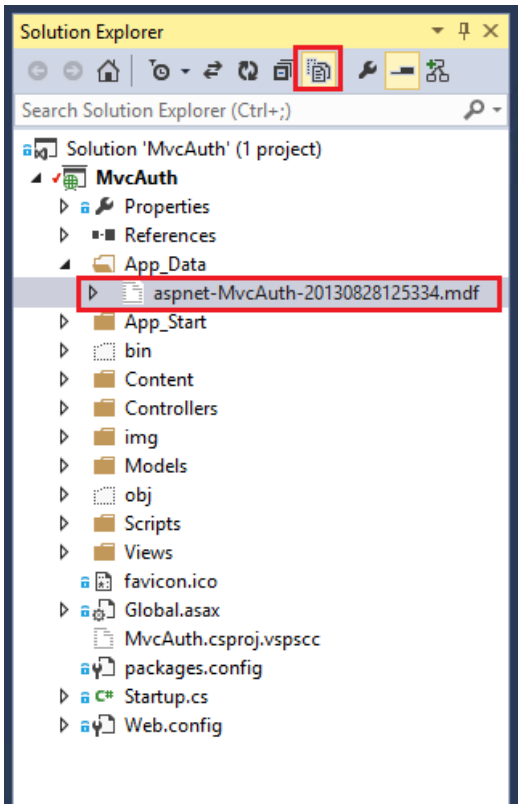
@using (Html.BeginForm("ExternalLoginConfirmation", "Account", new { ReturnUrl = ViewBag.ReturnUrl },
FormMethod.Post, new { @class = "form-horizontal", role = "form" }))
{
    @Html.AntiForgeryToken()

    <h4>Association Form</h4>
    <hr />
    @Html.ValidationSummary(true, "", new { @class = "text-danger" })
    <p class="text-info">
        You've successfully authenticated with <strong>@ViewBag.LoginProvider</strong>.
            Please enter a user name for this site below and click the Register button to finish
            logging in.
    </p>
    <div class="form-group">
        @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
            @Html.ValidationMessageFor(m => m.Email, "", new { @class = "text-danger" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.HomeTown, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.HomeTown, new { @class = "form-control" })
            @Html.ValidationMessageFor(m => m.HomeTown)
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.BirthDate, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.BirthDate, new { @class = "form-control" })
            @Html.ValidationMessageFor(m => m.BirthDate)
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Register" />
        </div>
    </div>
}

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Delete the membership database so you can again register your Facebook account with your application and verify you can add the new birth date and home town profile information.

From **Solution Explorer**, click the **Show All Files** icon, then right click *Add_Data\aspnet-MvcAuth-<dateStamp>.mdf* and click **Delete**.



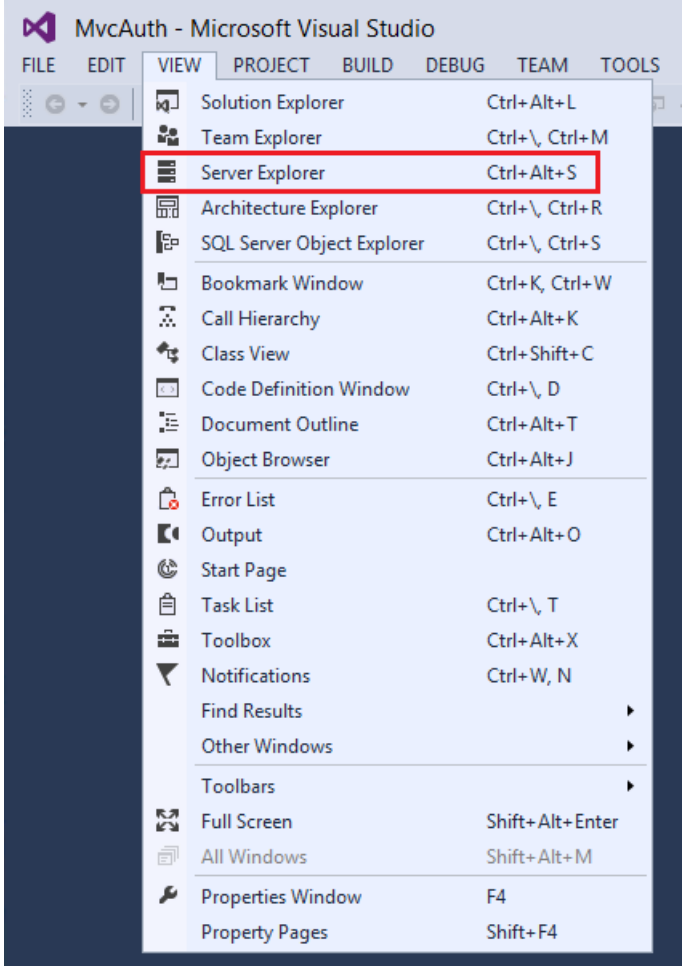
From the **Tools** menu, click **NuGet Package Manger**, then click **Package Manager Console** (PMC). Enter the following commands in the PMC.

1. Enable-Migrations
2. Add-Migration Init
3. Update-Database

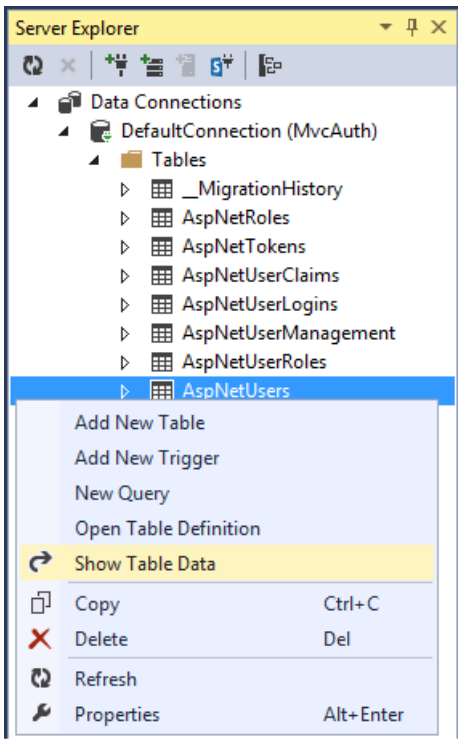
Run the application and use FaceBook and Google to log in and register some users.

Examine the Membership Data

In the **View** menu, click **Server Explorer**.



Right click **AspNetUsers** and click **Show Table Data**.



The `HomeTown` and `BirthDate` fields are shown below.

dbo.AspNetUsers [Data]

Max Rows: 1000

Id	HomeTown	BirthDate	Email	EmailConfirmed
eaa1081f	Captown	7/7/1977 12:00:...	Scott@contoso.com	False
a64159...	Redmond	1/2/1903 12:00:...	Rick2@Contoso.com	False
b2d94...	Great Falls	8/8/1988 12:00:...	RickAndMSFT@gm...	False
NULL	NULL	NULL	NULL	NULL

Logging off your App and Logging in With Another Account

If you log on to your app with Facebook,, and then log out and try to log in again with a different Facebook account (using the same browser), you will be immediately logged in to the previous Facebook account you used. In order to use another account, you need to navigate to Facebook and log out at Facebook. The same rule applies to any other 3rd party authentication provider. Alternatively, you can log in with another account by using a different browser.

Next Steps

See Introducing the Yahoo and LinkedIn OAuth security providers for OWIN by Jerrie Pelser for Yahoo and LinkedIn instructions. See Jerrie's Pretty social login buttons for ASP.NET MVC 5 to get enable social login buttons.

Follow my tutorial Create an ASP.NET MVC app with auth and SQL DB and deploy to Azure App Service, which continues this tutorial and shows the following:

1. How to deploy your app to Azure.
2. How to secure you app with roles.
3. How to secure your app with the RequireHttps and Authorize filters.
4. How to use the membership API to add users and roles.

Please leave feedback on how you liked this tutorial and what we could improve. You can also request new topics at Show Me How With Code. You can even ask for and vote on new features to be added to ASP.NET. For example, you can vote for a tool to create and manage users and roles.

For an good explanation of how ASP.NET External Authentication Services work, see Robert McMurray's External Authentication Services. Robert's article also goes into detail in enabling Microsoft and Twitter authentication. Tom Dykstra's excellent EF/MVC tutorial shows how to work with the Entity Framework.

Create a secure ASP.NET MVC 5 web app with log in, email confirmation and password reset (C#)

2/20/2020 • 12 minutes to read • Edit Online

by Rick Anderson

This tutorial shows you how to build an ASP.NET MVC 5 web app with email confirmation and password reset using the ASP.NET Identity membership system.

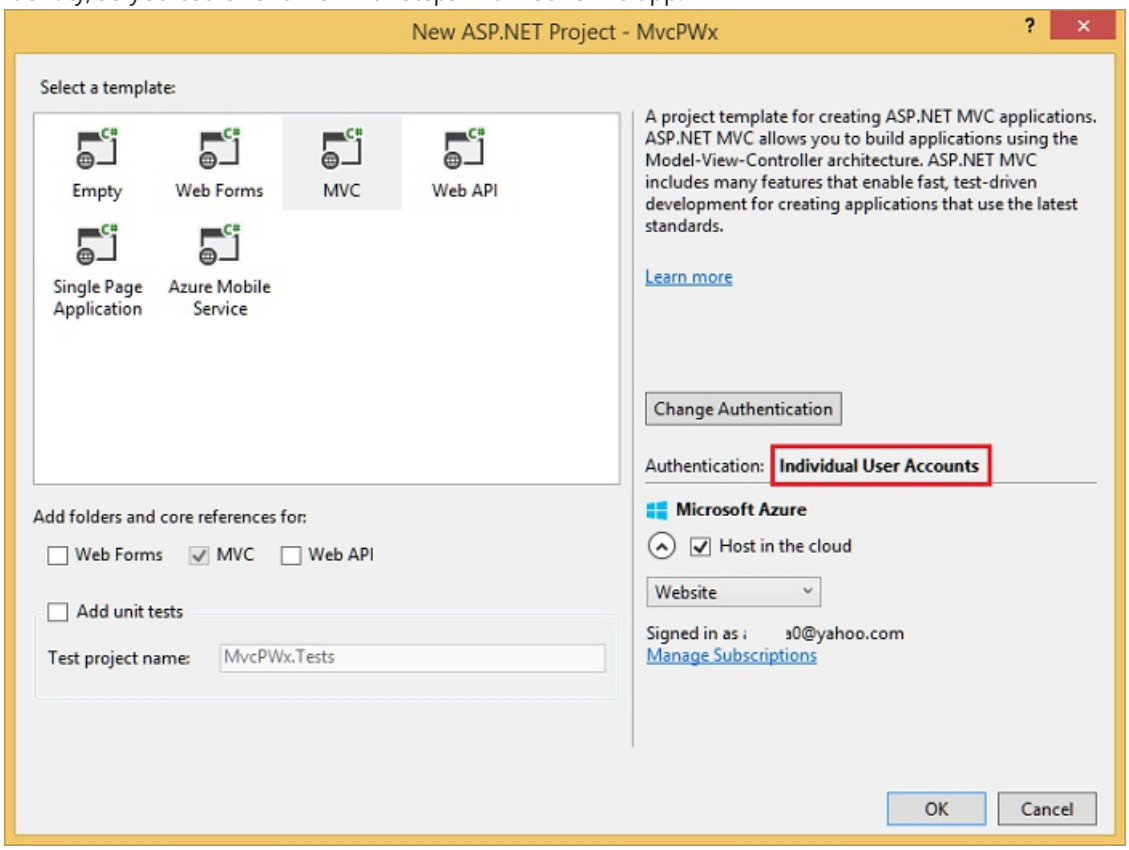
For an updated version of this tutorial that uses .NET Core, see Account confirmation and password recovery in ASP.NET Core.

Create an ASP.NET MVC app

Start by installing and running Visual Studio Express 2013 for Web or Visual Studio 2013. Install Visual Studio 2013 Update 3 or higher.

> **NOTE**
>
> Warning: You must install Visual Studio 2013 Update 3 or higher to complete this tutorial.

1. Create a new ASP.NET Web project and select the MVC template. Web Forms also supports ASP.NET Identity, so you could follow similar steps in a web forms app.



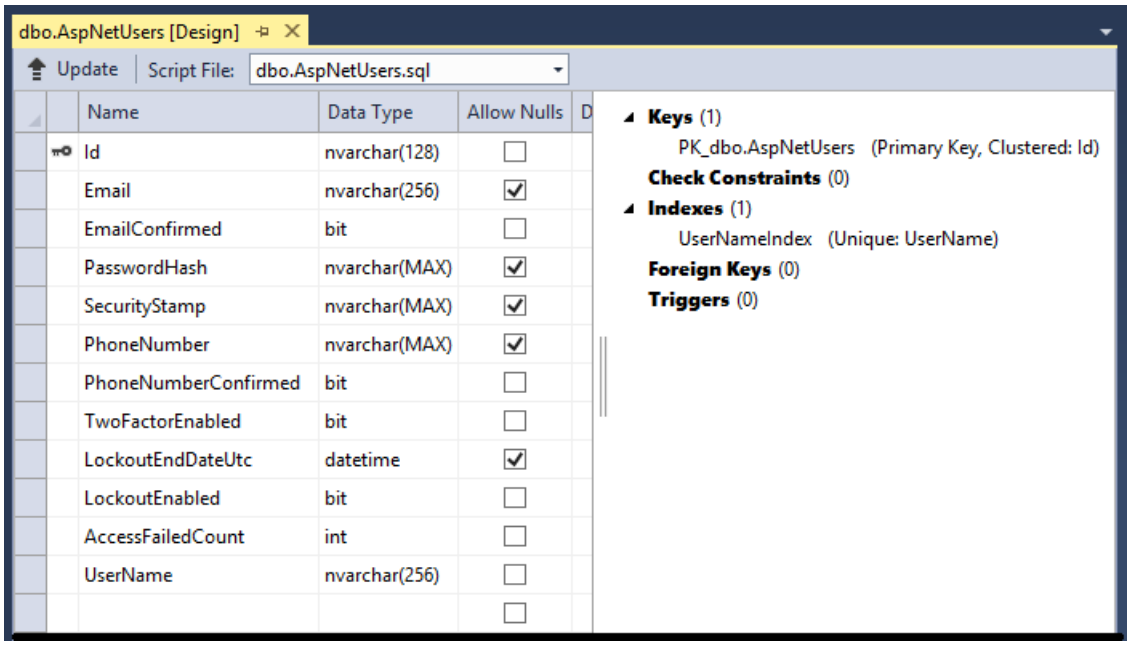
2. Leave the default authentication as **Individual User Accounts**. If you'd like to host the app in Azure, leave the check box checked. Later in the tutorial we will deploy to Azure. You can open an Azure account for free.

3. Set the project to use SSL.

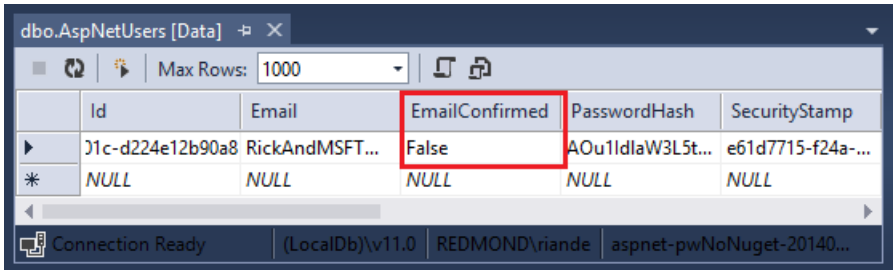
4. Run the app, click the **Register** link and register a user. At this point, the only validation on the email is with the [EmailAddress] attribute.

5. In Server Explorer, navigate to **Data Connections\DefaultConnection\Tables\AspNetUsers**, right click and select **Open table definition**.

   The following image shows the `AspNetUsers` schema:



6. Right click on the **AspNetUsers** table and select **Show Table Data**.



   At this point the email has not been confirmed.

7. Click on the row and select delete. You'll add this email again in the next step, and send a confirmation email.

Email confirmation

It's a best practice to confirm the email of a new user registration to verify they are not impersonating someone else (that is, they haven't registered with someone else's email). Suppose you had a discussion forum, you would want to prevent `"bob@example.com"` from registering as `"joe@contoso.com"`. Without email confirmation, `"joe@contoso.com"` could get unwanted email from your app. Suppose Bob accidentally registered as `"bib@example.com"` and hadn't noticed it, he wouldn't be able to use password recover because the app doesn't have his correct email. Email confirmation provides only limited protection from bots and doesn't provide protection from determined spammers, they have many working email aliases they can use to register.

You generally want to prevent new users from posting any data to your web site before they have been confirmed by email, a SMS text message or another mechanism. In the sections below, we will enable email confirmation and modify the code to prevent newly registered users from logging in until their email has been confirmed.

Hook up SendGrid

The instructions in this section are not current. See Configure SendGrid email provider for updated instructions.

Although this tutorial only shows how to add email notification through SendGrid, you can send email using SMTP and other mechanisms (see additional resources).

1. In the Package Manager Console, enter the following command:

```
Install-Package SendGrid
```

2. Go to the Azure SendGrid sign up page and register for a free SendGrid account. Configure SendGrid by adding code similar to the following in *App_Start/IdentityConfig.cs*:

```
public class EmailService : IIdentityMessageService
{
   public async Task SendAsync(IdentityMessage message)
   {
      await configSendGridasync(message);
   }

   // Use NuGet to install SendGrid (Basic C# client lib)
   private async Task configSendGridasync(IdentityMessage message)
   {
      var myMessage = new SendGridMessage();
      myMessage.AddTo(message.Destination);
      myMessage.From = new System.Net.Mail.MailAddress(
                          "Joe@contoso.com", "Joe S.");
      myMessage.Subject = message.Subject;
      myMessage.Text = message.Body;
      myMessage.Html = message.Body;

      var credentials = new NetworkCredential(
                 ConfigurationManager.AppSettings["mailAccount"],
                 ConfigurationManager.AppSettings["mailPassword"]
                 );

      // Create a Web transport for sending email.
      var transportWeb = new Web(credentials);

      // Send the email.
      if (transportWeb != null)
      {
         await transportWeb.DeliverAsync(myMessage);
      }
      else
      {
         Trace.TraceError("Failed to create Web transport.");
         await Task.FromResult(0);
      }
   }
}
```

You'll need to add the following includes:

```
using SendGrid;
using System.Net;
using System.Configuration;
using System.Diagnostics;
```

To keep this sample simple, we'll store the app settings in the *web.config* file:

```
</connectionStrings>
   <appSettings>
      <add key="webpages:Version" value="3.0.0.0" />
      <!-- Markup removed for clarity. -->

      <add key="mailAccount" value="xyz" />
      <add key="mailPassword" value="password" />
   </appSettings>
  <system.web>
```

WARNING

Security - Never store sensitive data in your source code. The account and credentials are stored in the appSetting. On Azure, you can securely store these values on the **Configure** tab in the Azure portal. See Best practices for deploying passwords and other sensitive data to ASP.NET and Azure.

Enable email confirmation in the Account controller

```
//
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            await SignInManager.SignInAsync(user, isPersistent:false, rememberBrowser:false);

            string code = await UserManager.GenerateEmailConfirmationTokenAsync(user.Id);
            var callbackUrl = Url.Action("ConfirmEmail", "Account",
               new { userId = user.Id, code = code }, protocol: Request.Url.Scheme);
            await UserManager.SendEmailAsync(user.Id,
               "Confirm your account", "Please confirm your account by clicking <a href=\""
               + callbackUrl + "\">here</a>");

            return RedirectToAction("Index", "Home");
        }
        AddErrors(result);
    }

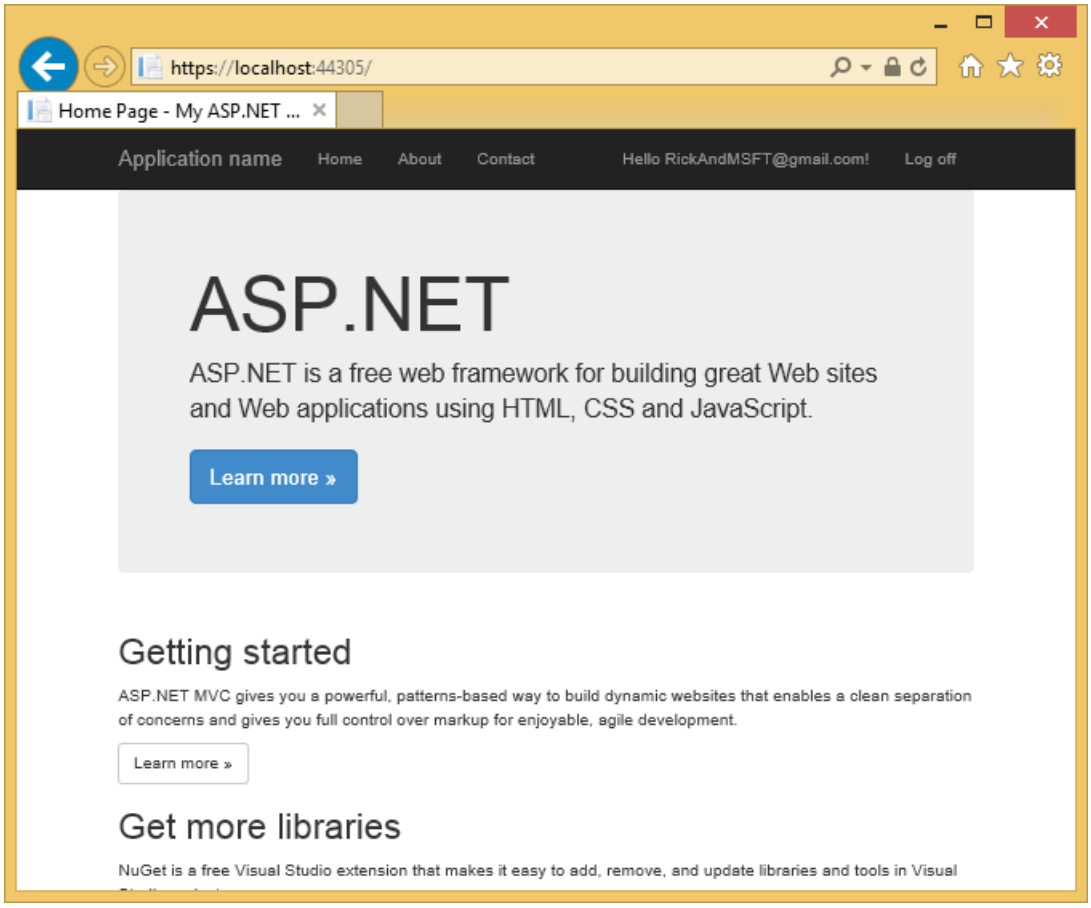
    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Verify the *Views\Account\ConfirmEmail.cshtml* file has correct razor syntax. (The @ character in the first line might be missing.)

```
@{
    ViewBag.Title = "Confirm Email";
}

<h2>@ViewBag.Title.</h2>
<div>
    <p>
        Thank you for confirming your email. Please @Html.ActionLink("Click here to Log in", "Login",
"Account", routeValues: null, htmlAttributes: new { id = "loginLink" })
    </p>
</div>
```

Run the app and click the Register link. Once you submit the registration form, you are logged in.



Check your email account and click on the link to confirm your email.

Require email confirmation before log in

Currently once a user completes the registration form, they are logged in. You generally want to confirm their email before logging them in. In the section below, we will modify the code to require new users to have a confirmed email before they are logged in (authenticated). Update the `HttpPost Register` method with the following highlighted changes:

```
//
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
   if (ModelState.IsValid)
   {
      var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
      var result = await UserManager.CreateAsync(user, model.Password);
      if (result.Succeeded)
      {
         //  Comment the following line to prevent log in until the user is confirmed.
         //  await SignInManager.SignInAsync(user, isPersistent:false, rememberBrowser:false);

         string code = await UserManager.GenerateEmailConfirmationTokenAsync(user.Id);
         var callbackUrl = Url.Action("ConfirmEmail", "Account",
            new { userId = user.Id, code = code }, protocol: Request.Url.Scheme);
         await UserManager.SendEmailAsync(user.Id, "Confirm your account",
            "Please confirm your account by clicking <a href=\"" + callbackUrl + "\">here</a>");

         // Uncomment to debug locally
         // TempData["ViewBagLink"] = callbackUrl;

         ViewBag.Message = "Check your email and confirm your account, you must be confirmed "
                         + "before you can log in.";

         return View("Info");
         //return RedirectToAction("Index", "Home");
      }
      AddErrors(result);
   }

   // If we got this far, something failed, redisplay form
   return View(model);
}
```

By commenting out the `SignInAsync` method, the user will not be signed in by the registration. The `TempData["ViewBagLink"] = callbackUrl;` line can be used to debug the app and test registration without sending email. `ViewBag.Message` is used to display the confirm instructions. The download sample contains code to test email confirmation without setting up email, and can also be used to debug the application.

Create a `Views\Shared\Info.cshtml` file and add the following razor markup:

```
@{
   ViewBag.Title = "Info";
}
<h2>@ViewBag.Title.</h2>
<h3>@ViewBag.Message</h3>
```

Add the Authorize attribute to the `Contact` action method of the Home controller. You can click on the **Contact** link to verify anonymous users don't have access and authenticated users do have access.

```
[Authorize]
public ActionResult Contact()
{
   ViewBag.Message = "Your contact page.";

   return View();
}
```

You must also update the `HttpPost Login` action method:

```
//
// POST: /Account/Login
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }

    // Require the user to have a confirmed email before they can log on.
    var user = await UserManager.FindByNameAsync(model.Email);
    if (user != null)
    {
       if (!await UserManager.IsEmailConfirmedAsync(user.Id))
       {
          ViewBag.errorMessage = "You must have a confirmed email to log on.";
          return View("Error");
       }
    }

    // This doesn't count login failures towards account lockout
    // To enable password failures to trigger account lockout, change to shouldLockout: true
    var result = await SignInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe,
shouldLockout: false);
    switch (result)
    {
        case SignInStatus.Success:
            return RedirectToLocal(returnUrl);
        case SignInStatus.LockedOut:
            return View("Lockout");
        case SignInStatus.RequiresVerification:
            return RedirectToAction("SendCode", new { ReturnUrl = returnUrl, RememberMe = model.RememberMe
});
        case SignInStatus.Failure:
        default:
            ModelState.AddModelError("", "Invalid login attempt.");
            return View(model);
    }
}
```

Update the *Views\Shared\Error.cshtml* view to display the error message:

```
@model System.Web.Mvc.HandleErrorInfo

@{
    ViewBag.Title = "Error";
}

<h1 class="text-danger">Error.</h1>
@{
   if (String.IsNullOrEmpty(ViewBag.errorMessage))
   {
      <h2 class="text-danger">An error occurred while processing your request.</h2>
   }
   else
   {
      <h2 class="text-danger">@ViewBag.errorMessage</h2>
   }
}
```

Delete any accounts in the **AspNetUsers** table that contain the email alias you wish to test with. Run the app and verify you can't log in until you have confirmed your email address. Once you confirm your email address, click the **Contact** link.

Password recovery/reset

Remove the comment characters from the `HttpPost ForgotPassword` action method in the account controller:

```
//
// POST: /Account/ForgotPassword
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = await UserManager.FindByNameAsync(model.Email);
        if (user == null || !(await UserManager.IsEmailConfirmedAsync(user.Id)))
        {
            // Don't reveal that the user does not exist or is not confirmed
            return View("ForgotPasswordConfirmation");
        }

        string code = await UserManager.GeneratePasswordResetTokenAsync(user.Id);
        var callbackUrl = Url.Action("ResetPassword", "Account", new { userId = user.Id, code = code },
protocol: Request.Url.Scheme);
        await UserManager.SendEmailAsync(user.Id, "Reset Password", "Please reset your password by clicking
<a href=\"" + callbackUrl + "\">here</a>");
        return RedirectToAction("ForgotPasswordConfirmation", "Account");
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Remove the comment characters from the `ForgotPassword` ActionLink in the *Views\Account\Login.cshtml* razor view file:

```
@using MvcPWy.Models
@model LoginViewModel
@{
   ViewBag.Title = "Log in";
}

<h2>@ViewBag.Title.</h2>
<div class="row">
   <div class="col-md-8">
      <section id="loginForm">
         @using (Html.BeginForm("Login", "Account", new { ReturnUrl = ViewBag.ReturnUrl }, FormMethod.Post,
new { @class = "form-horizontal", role = "form" }))
         {
            @Html.AntiForgeryToken()
            <h4>Use a local account to log in.</h4>
            <hr />
            @Html.ValidationSummary(true, "", new { @class = "text-danger" })
            <div class="form-group">
               @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
               <div class="col-md-10">
                  @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
                  @Html.ValidationMessageFor(m => m.Email, "", new { @class = "text-danger" })
               </div>
            </div>
            <div class="form-group">
               @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
               <div class="col-md-10">
                  @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
                  @Html.ValidationMessageFor(m => m.Password, "", new { @class = "text-danger" })
               </div>
            </div>
            <div class="form-group">
               <div class="col-md-offset-2 col-md-10">
                  <div class="checkbox">
                     @Html.CheckBoxFor(m => m.RememberMe)
                     @Html.LabelFor(m => m.RememberMe)
                  </div>
               </div>
            </div>
            <div class="form-group">
               <div class="col-md-offset-2 col-md-10">
                  <input type="submit" value="Log in" class="btn btn-default" />
               </div>
            </div>
            <p>
               @Html.ActionLink("Register as a new user", "Register")
            </p>
            @* Enable this once you have account confirmation enabled for password reset functionality *@
            <p>
               @Html.ActionLink("Forgot your password?", "ForgotPassword")
            </p>
         }
      </section>
   </div>
   <div class="col-md-4">
      <section id="socialLoginForm">
         @Html.Partial("_ExternalLoginsListPartial", new ExternalLoginListViewModel { ReturnUrl =
ViewBag.ReturnUrl })
      </section>
   </div>
</div>

@section Scripts {
   @Scripts.Render("~/bundles/jqueryval")
}
```

The Log in page will now show a link to reset the password.

Resend email confirmation link

Once a user creates a new local account, they are emailed a confirmation link they are required to use before they can log on. If the user accidentally deletes the confirmation email, or the email never arrives, they will need the confirmation link sent again. The following code changes show how to enable this.

Add the following helper method to the bottom of the *Controllers\AccountController.cs* file:

```
private async Task<string> SendEmailConfirmationTokenAsync(string userID, string subject)
{
   string code = await UserManager.GenerateEmailConfirmationTokenAsync(userID);
   var callbackUrl = Url.Action("ConfirmEmail", "Account",
      new { userId = userID, code = code }, protocol: Request.Url.Scheme);
   await UserManager.SendEmailAsync(userID, subject,
      "Please confirm your account by clicking <a href=\"" + callbackUrl + "\">here</a>");

   return callbackUrl;
}
```

Update the Register method to use the new helper:

```
//
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
   if (ModelState.IsValid)
   {
      var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
      var result = await UserManager.CreateAsync(user, model.Password);
      if (result.Succeeded)
      {
         //  Comment the following line to prevent log in until the user is confirmed.
         //  await SignInManager.SignInAsync(user, isPersistent:false, rememberBrowser:false);

         string callbackUrl = await SendEmailConfirmationTokenAsync(user.Id, "Confirm your account");

         ViewBag.Message = "Check your email and confirm your account, you must be confirmed "
                         + "before you can log in.";

         return View("Info");
         //return RedirectToAction("Index", "Home");
      }
      AddErrors(result);
   }

   // If we got this far, something failed, redisplay form
   return View(model);
}
```

Update the Login method to resend the password if the user account has not been confirmed:

```
//
// POST: /Account/Login
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
   if (!ModelState.IsValid)
   {
      return View(model);
   }

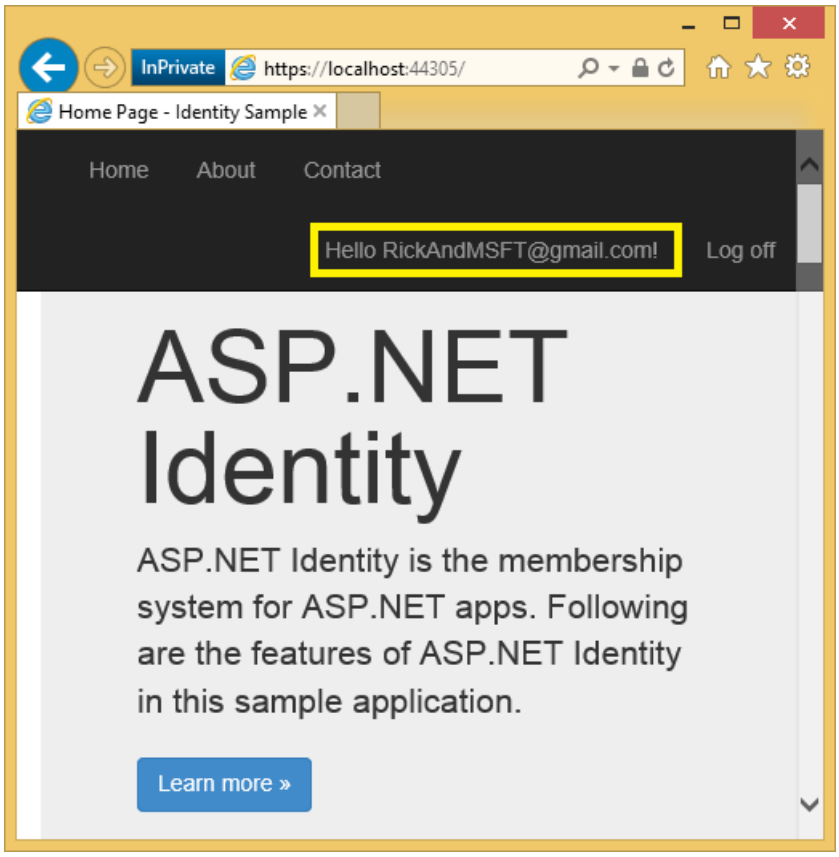
   // Require the user to have a confirmed email before they can log on.
  // var user = await UserManager.FindByNameAsync(model.Email);
   var user =  UserManager.Find(model.Email, model.Password);
   if (user != null)
   {
      if (!await UserManager.IsEmailConfirmedAsync(user.Id))
      {
         string callbackUrl = await SendEmailConfirmationTokenAsync(user.Id, "Confirm your account-Resend");

          // Uncomment to debug locally
          // ViewBag.Link = callbackUrl;
         ViewBag.errorMessage = "You must have a confirmed email to log on. "
                              + "The confirmation token has been resent to your email account.";
         return View("Error");
      }
   }

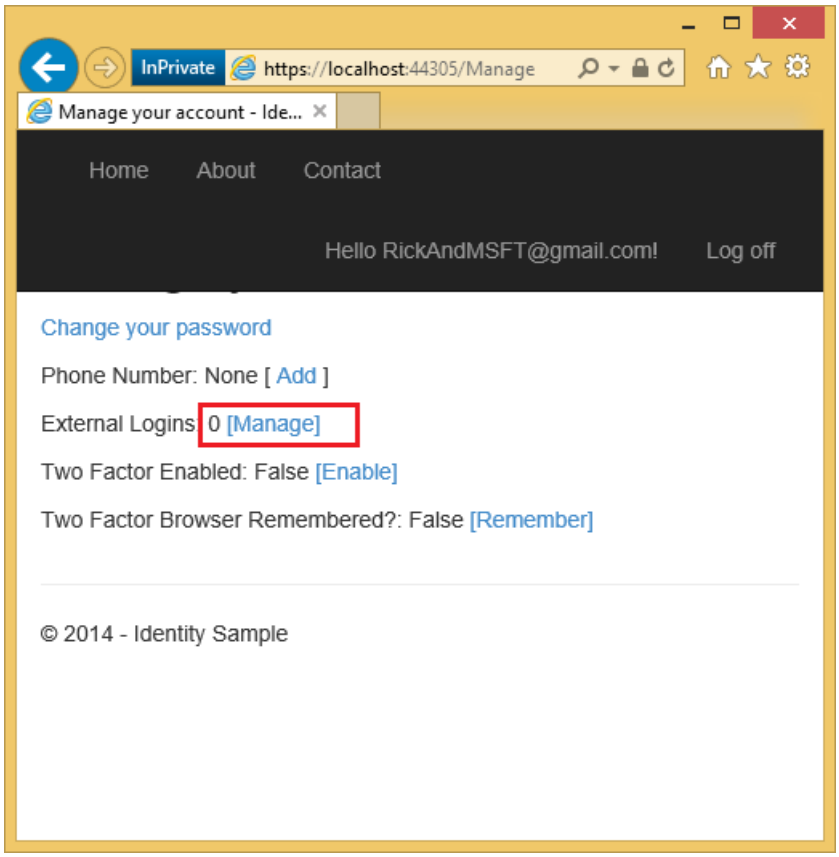
   // This doesn't count login failures towards account lockout
   // To enable password failures to trigger account lockout, change to shouldLockout: true
   var result = await SignInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe,
shouldLockout: false);
   switch (result)
   {
      case SignInStatus.Success:
         return RedirectToLocal(returnUrl);
      case SignInStatus.LockedOut:
         return View("Lockout");
      case SignInStatus.RequiresVerification:
         return RedirectToAction("SendCode", new { ReturnUrl = returnUrl, RememberMe = model.RememberMe });
      case SignInStatus.Failure:
      default:
         ModelState.AddModelError("", "Invalid login attempt.");
         return View(model);
   }
}
```

Combine social and local login accounts

You can combine local and social accounts by clicking on your email link. In the following sequence **RickAndMSFT@gmail.com** is first created as a local login, but you can create the account as a social log in first, then add a local login.

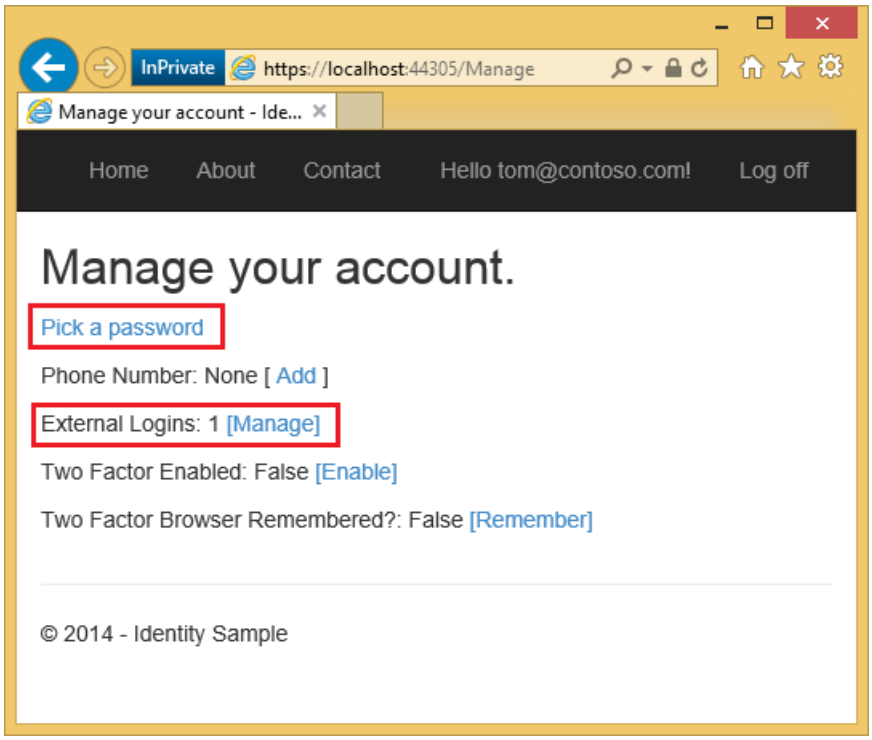


Click on the **Manage** link. Note the **External Logins: 0** associated with this account.

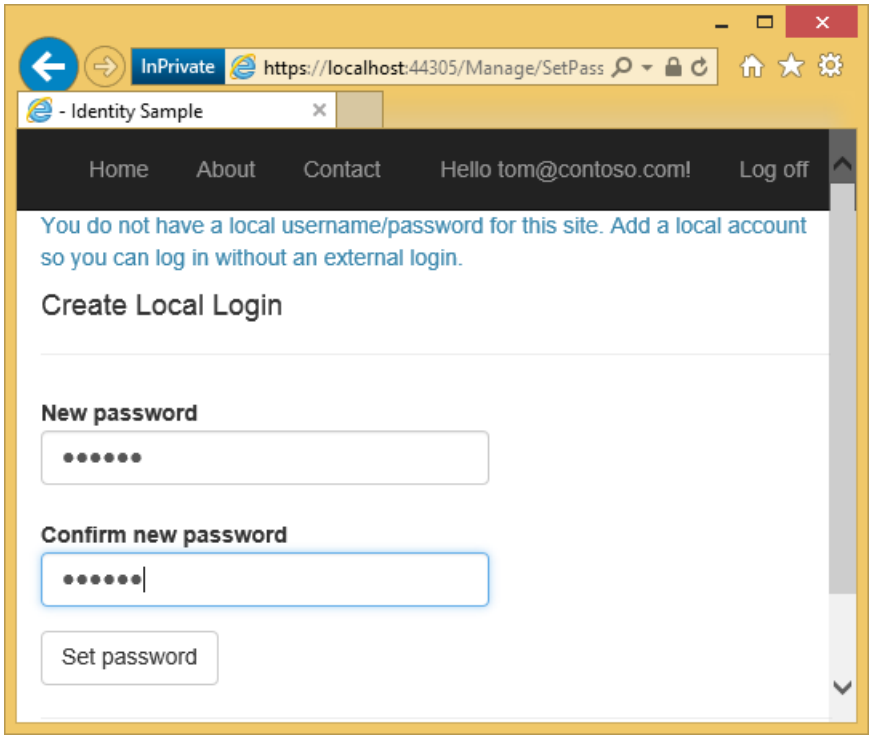


Click the link to another log in service and accept the app requests. The two accounts have been combined, you will be able to log on with either account. You might want your users to add local accounts in case their social log in authentication service is down, or more likely they have lost access to their social account.

In the following image, Tom is a social log in (which you can see from the **External Logins: 1** shown on the page).



Clicking on **Pick a password** allows you to add a local log on associated with the same account.



Email confirmation in more depth

My tutorial Account Confirmation and Password Recovery with ASP.NET Identity goes into this topic with more details.

Debugging the app

If you don't get an email containing the link:

- Check your junk or spam folder.
- Log into your SendGrid account and click on the Email Activity link.

To test the verification link without email, download the completed sample. The confirmation link and

confirmation codes will be displayed on the page.

Additional Resources

- Links to ASP.NET Identity Recommended Resources
- Account Confirmation and Password Recovery with ASP.NET Identity Goes into more detail on password recovery and account confirmation.
- MVC 5 App with Facebook, Twitter, LinkedIn and Google OAuth2 Sign-on This tutorial shows you how to write an ASP.NET MVC 5 app with Facebook and Google OAuth 2 authorization. It also shows how to add additional data to the Identity database.
- Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL Database to Azure. This tutorial adds Azure deployment, how to secure your app with roles, how to use the membership API to add users and roles, and additional security features.
- Creating a Google app for OAuth 2 and connecting the app to the project
- Creating the app in Facebook and connecting the app to the project
- Setting up SSL in the Project

ASP.NET MVC 5 app with SMS and email Two-Factor Authentication

2/20/2020 • 6 minutes to read • Edit Online

by Rick Anderson

> This tutorial shows you how to build an ASP.NET MVC 5 web app with Two-Factor Authentication. You should complete Create a secure ASP.NET MVC 5 web app with log in, email confirmation and password reset before proceeding. You can download the completed application here. The download contains debugging helpers that let you test email confirmation and SMS without setting up an email or SMS provider.
>
> This tutorial was written by Rick Anderson (Twitter: @RickAndMSFT).

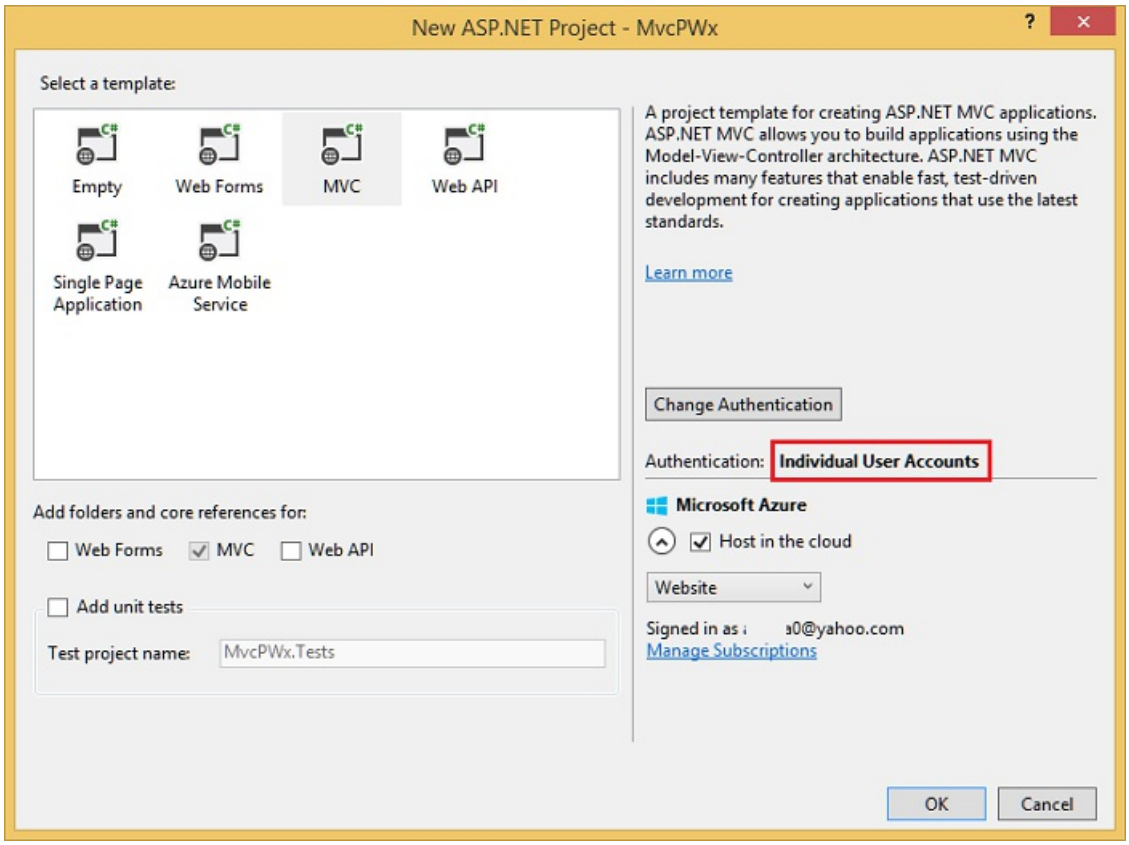
- Create an ASP.NET MVC app
- Set up SMS for Two-factor authentication
- Enable Two-factor authentication
- Additional Resources

Create an ASP.NET MVC app

Start by installing and running Visual Studio Express 2013 for Web or Visual Studio 2013. Install Visual Studio 2013 Update 3 or higher.

> **NOTE**
>
> Warning: You should complete Create a secure ASP.NET MVC 5 web app with log in, email confirmation and password reset before proceeding. You must install Visual Studio 2013 Update 3 or higher to complete this tutorial.

1. Create a new ASP.NET Web project and select the MVC template. Web Forms also supports ASP.NET Identity, so you could follow similar steps in a web forms app.



2. Leave the default authentication as **Individual User Accounts**. If you'd like to host the app in Azure, leave the check box checked. Later in the tutorial we will deploy to Azure. You can open an Azure account for free.
3. Set the project to use SSL.

Set up SMS for Two-factor authentication

This tutorial provides instructions for using either Twilio or ASPSMS but you can use any other SMS provider.

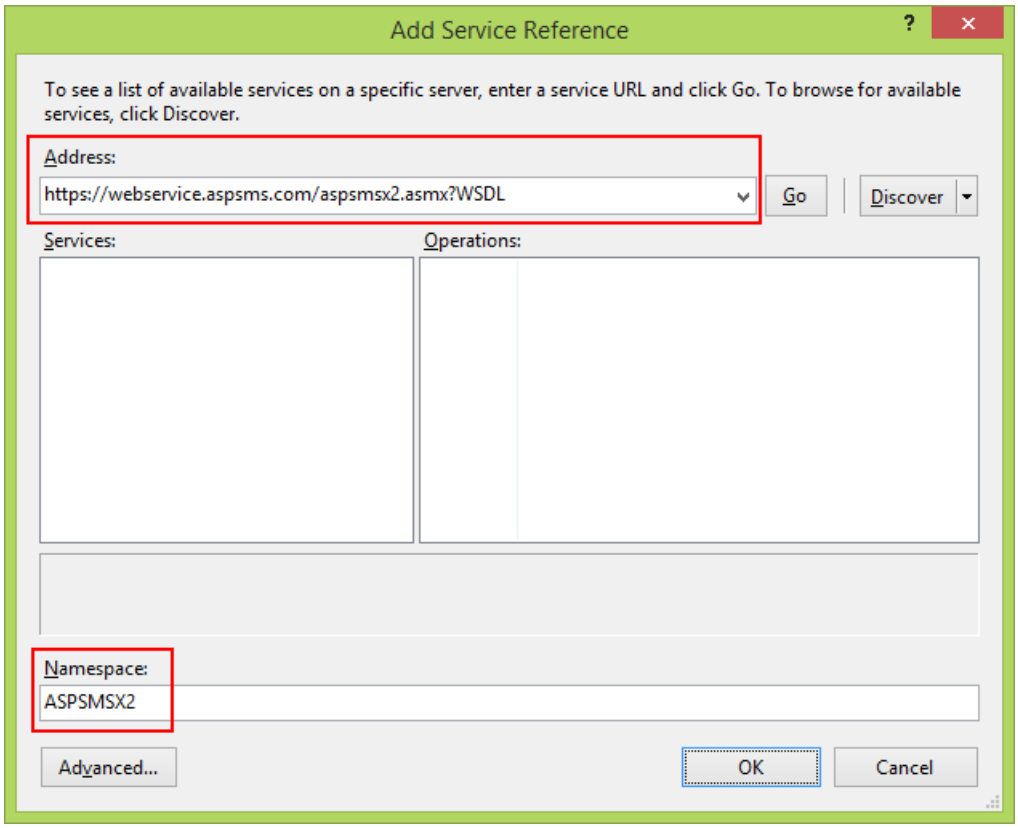
1. **Creating a User Account with an SMS provider**

   Create a Twilio or an ASPSMS account.

2. **Installing additional packages or adding service references**

   Twilio:
   In the Package Manager Console, enter the following command:
   `Install-Package Twilio`

   ASPSMS:
   The following service reference needs to be added:



Address:
`https://webservice.aspsms.com/aspsmsx2.asmx?WSDL`

Namespace:
`ASPSMSX2`

3. **Figuring out SMS Provider User credentials**

   Twilio:
   From the **Dashboard** tab of your Twilio account, copy the **Account SID** and **Auth token**.

   ASPSMS:
   From your account settings, navigate to **Userkey** and copy it together with your self-defined **Password**.

   We will later store these values in the *web.config* file within the keys `"SMSAccountIdentification"` and `"SMSAccountPassword"` .

4. **Specifying SenderID / Originator**

   Twilio:
   From the **Numbers** tab, copy your Twilio phone number.

   ASPSMS:
   Within the **Unlock Originators** Menu, unlock one or more Originators or choose an alphanumeric Originator (Not supported by all networks).

   We will later store this value in the *web.config* file within the key `"SMSAccountFrom"` .

5. **Transferring SMS provider credentials into app**

   Make the credentials and sender phone number available to the app. To keep things simple we will store these values in the *web.config* file. When we deploy to Azure, we can store the values securely in the **app settings** section on the web site configure tab.

```
</connectionStrings>
   <appSettings>
      <add key="webpages:Version" value="3.0.0.0" />
      <!-- Markup removed for clarity. -->
      <!-- SendGrid-->
      <add key="mailAccount" value="account" />
      <add key="mailPassword" value="password" />
      <add key="SMSAccountIdentification" value="My Identification" />
      <add key="SMSAccountPassword" value="My Password" />
      <add key="SMSAccountFrom" value="+12065551234" />
   </appSettings>
  <system.web>
```

WARNING

Security - Never store sensitive data in your source code. The account and credentials are added to the code above to keep the sample simple. See Best practices for deploying passwords and other sensitive data to ASP.NET and Azure.

6. **Implementation of data transfer to SMS provider**

   Configure the `SmsService` class in the *App_Start\IdentityConfig.cs* file.

   Depending on the used SMS provider activate either the **Twilio** or the **ASPSMS** section:

```
public class SmsService : IIdentityMessageService
{
    public Task SendAsync(IdentityMessage message)
    {
        // Twilio Begin
        //var accountSid = ConfigurationManager.AppSettings["SMSAccountIdentification"];
        //var authToken = ConfigurationManager.AppSettings["SMSAccountPassword"];
        //var fromNumber = ConfigurationManager.AppSettings["SMSAccountFrom"];

        //TwilioClient.Init(accountSid, authToken);

        //MessageResource result = MessageResource.Create(
            //new PhoneNumber(message.Destination),
            //from: new PhoneNumber(fromNumber),
           //body: message.Body
        //);

        ////Status is one of Queued, Sending, Sent, Failed or null if the number is not valid
         //Trace.TraceInformation(result.Status.ToString());
        ////Twilio doesn't currently have an async API, so return success.
         //return Task.FromResult(0);
        // Twilio End

        // ASPSMS Begin
        // var soapSms = new MvcPWx.ASPSMSX2.ASPSMSX2SoapClient("ASPSMSX2Soap");
        // soapSms.SendSimpleTextSMS(
        //   System.Configuration.ConfigurationManager.AppSettings["SMSAccountIdentification"],
        //   System.Configuration.ConfigurationManager.AppSettings["SMSAccountPassword"],
        //   message.Destination,
        //   System.Configuration.ConfigurationManager.AppSettings["SMSAccountFrom"],
        //   message.Body);
        // soapSms.Close();
        // return Task.FromResult(0);
        // ASPSMS End
    }
}
```

7. Update the *Views\Manage\Index.cshtml* Razor view: (note: don't just remove the comments in the exiting code, use the code below.)

```
@model MvcPWy.Models.IndexViewModel
@{
   ViewBag.Title = "Manage";
}
<h2>@ViewBag.Title.</h2>
<p class="text-success">@ViewBag.StatusMessage</p>
<div>
   <h4>Change your account settings</h4>
   <hr />
   <dl class="dl-horizontal">
      <dt>Password:</dt>
      <dd>
         [
         @if (Model.HasPassword)
         {
            @Html.ActionLink("Change your password", "ChangePassword")
         }
         else
         {
            @Html.ActionLink("Create", "SetPassword")
         }
         ]
      </dd>
      <dt>External Logins:</dt>
      <dd>
         @Model.Logins.Count [
         @Html.ActionLink("Manage", "ManageLogins") ]
      </dd>
        <dt>Phone Number:</dt>
      <dd>
         @(Model.PhoneNumber ?? "None") [
         @if (Model.PhoneNumber != null)
         {
            @Html.ActionLink("Change", "AddPhoneNumber")
            @:  | 
            @Html.ActionLink("Remove", "RemovePhoneNumber")
         }
         else
         {
            @Html.ActionLink("Add", "AddPhoneNumber")
         }
         ]
      </dd>
      <dt>Two-Factor Authentication:</dt>
      <dd>
         @if (Model.TwoFactor)
         {
            using (Html.BeginForm("DisableTwoFactorAuthentication", "Manage", FormMethod.Post, new {
@class = "form-horizontal", role = "form" }))
            {
               @Html.AntiForgeryToken()
               <text>Enabled
                  <input type="submit" value="Disable" class="btn btn-link" />
               </text>
            }
         }
         else
         {
            using (Html.BeginForm("EnableTwoFactorAuthentication", "Manage", FormMethod.Post, new {
@class = "form-horizontal", role = "form" }))
            {
               @Html.AntiForgeryToken()
               <text>Disabled
                  <input type="submit" value="Enable" class="btn btn-link" />
               </text>
```

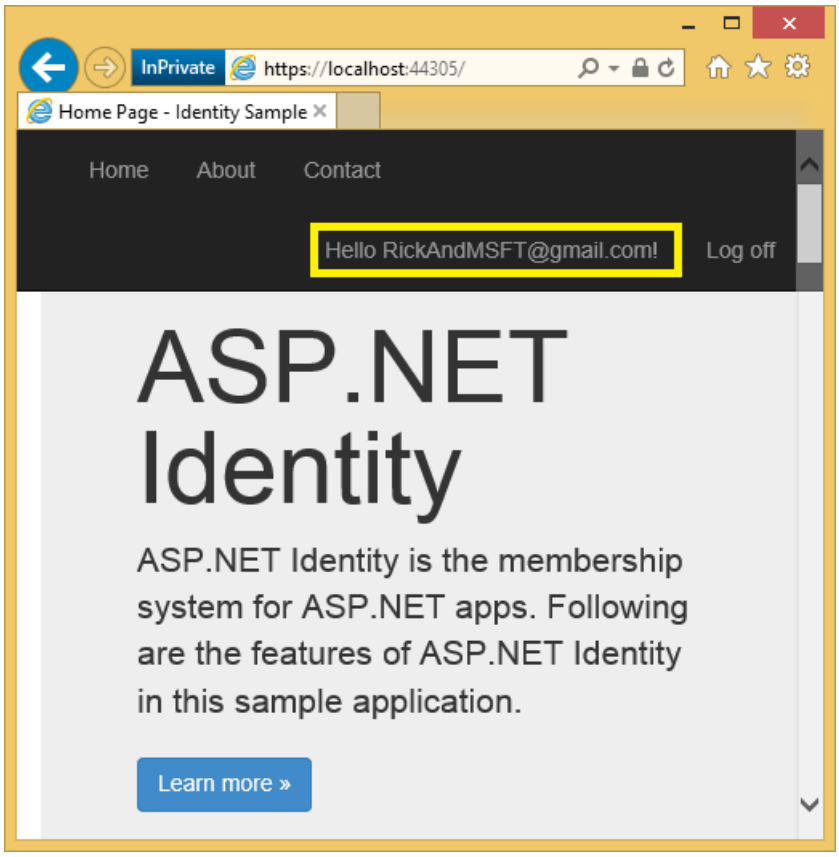
```
                </text>
            }
        }
    </dd>
  </dl>
</div>
```

8. Verify the `EnableTwoFactorAuthentication` and `DisableTwoFactorAuthentication` action methods in the `ManageController` have the[ValidateAntiForgeryToken] attribute:

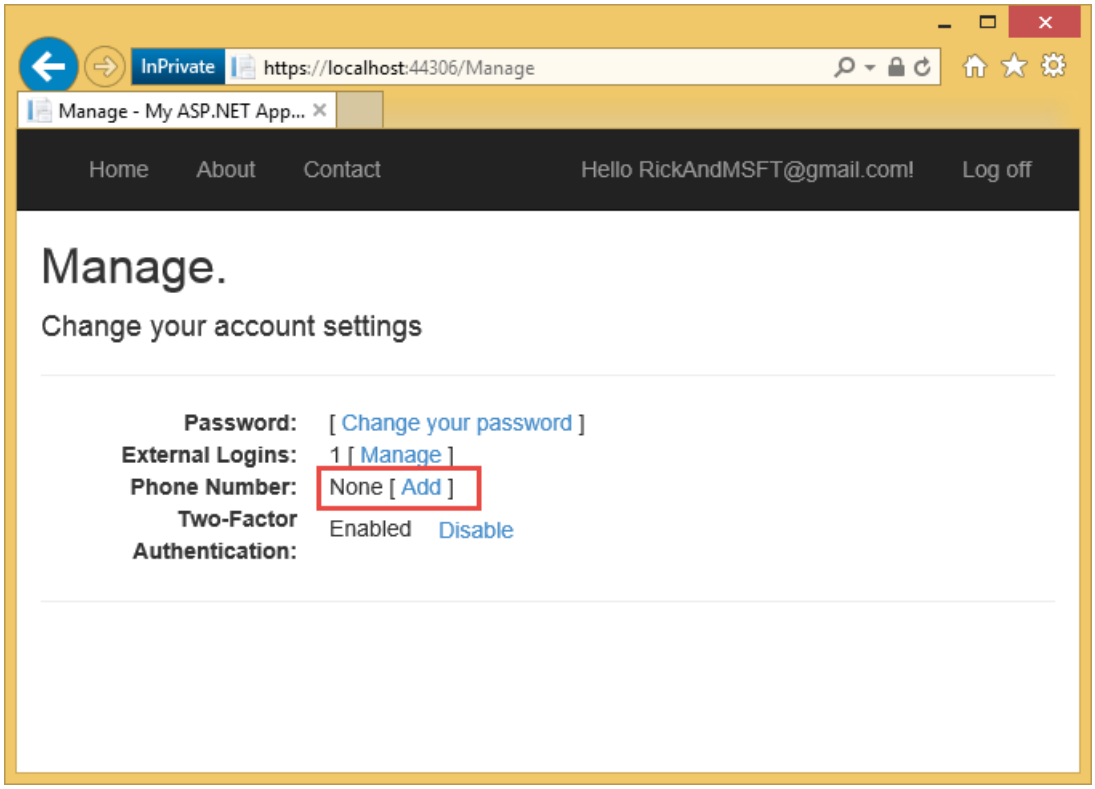
```
//
// POST: /Manage/EnableTwoFactorAuthentication
[HttpPost,ValidateAntiForgeryToken]
public async Task<ActionResult> EnableTwoFactorAuthentication()
{
    await UserManager.SetTwoFactorEnabledAsync(User.Identity.GetUserId(), true);
    var user = await UserManager.FindByIdAsync(User.Identity.GetUserId());
    if (user != null)
    {
        await SignInAsync(user, isPersistent: false);
    }
    return RedirectToAction("Index", "Manage");
}
//
// POST: /Manage/DisableTwoFactorAuthentication
[HttpPost, ValidateAntiForgeryToken]
public async Task<ActionResult> DisableTwoFactorAuthentication()
{
    await UserManager.SetTwoFactorEnabledAsync(User.Identity.GetUserId(), false);
    var user = await UserManager.FindByIdAsync(User.Identity.GetUserId());
    if (user != null)
    {
        await SignInAsync(user, isPersistent: false);
    }
    return RedirectToAction("Index", "Manage");
}
```

9. Run the app and log in with the account you previously registered.

10. Click on your User ID, which activates the `Index` action method in `Manage` controller.

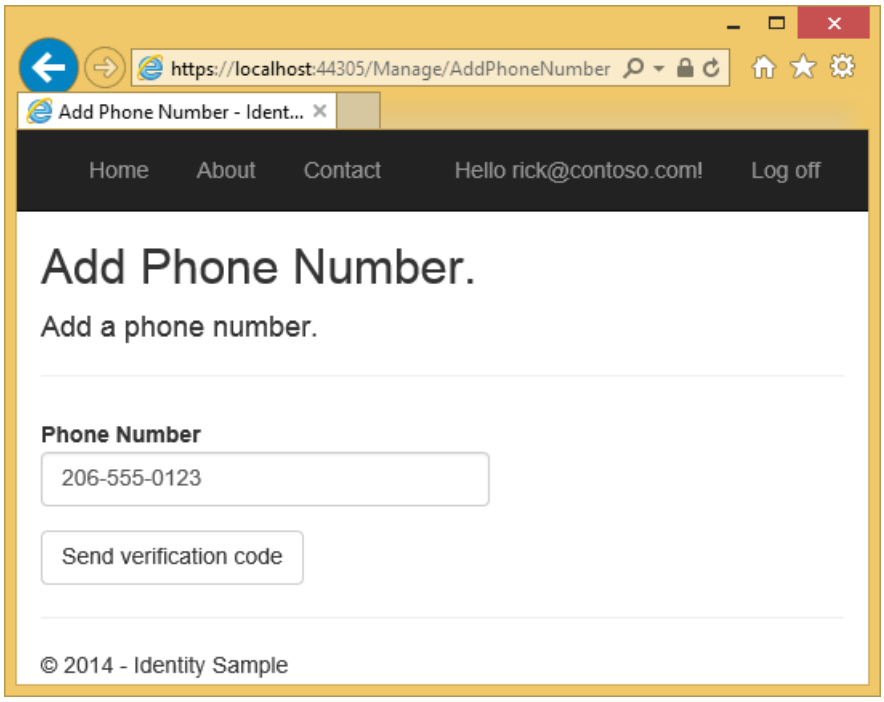


11. Click Add.

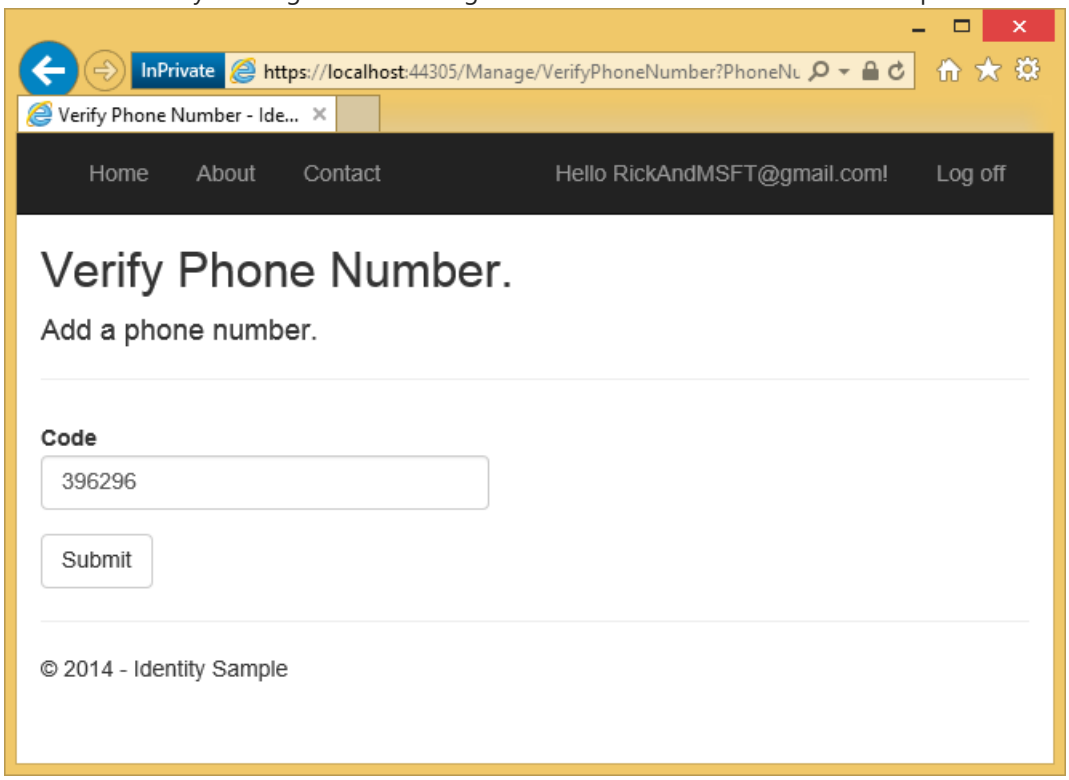


12. The `AddPhoneNumber` action method displays a dialog box to enter a phone number that can receive SMS messages.

```
// GET: /Account/AddPhoneNumber
public ActionResult AddPhoneNumber()
{
   return View();
}
```



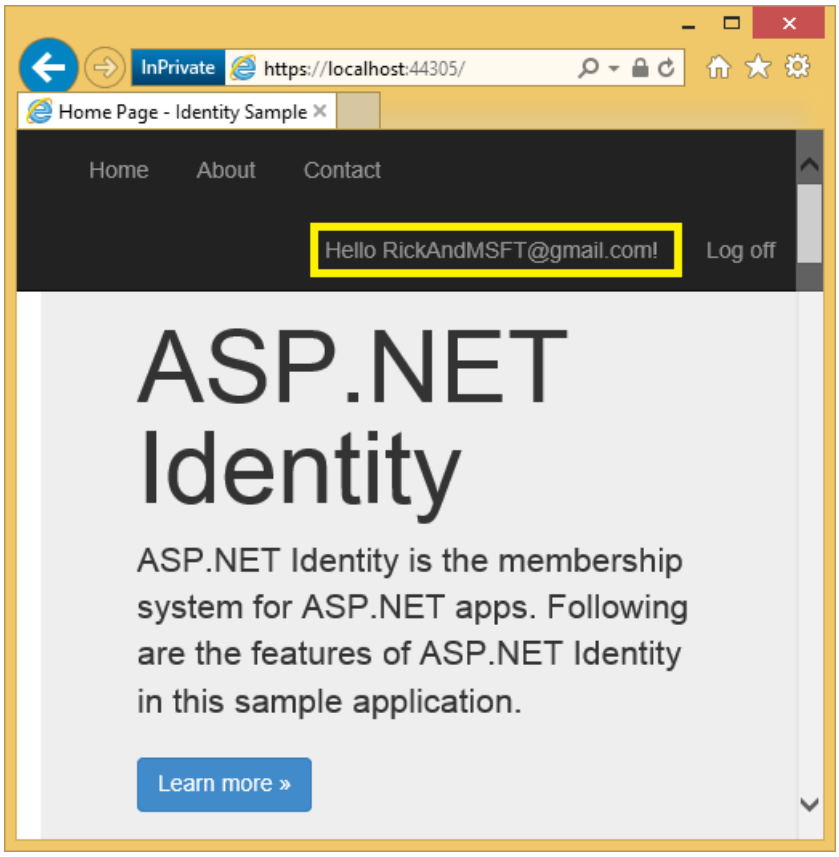
13. In a few seconds you will get a text message with the verification code. Enter it and press **Submit**.



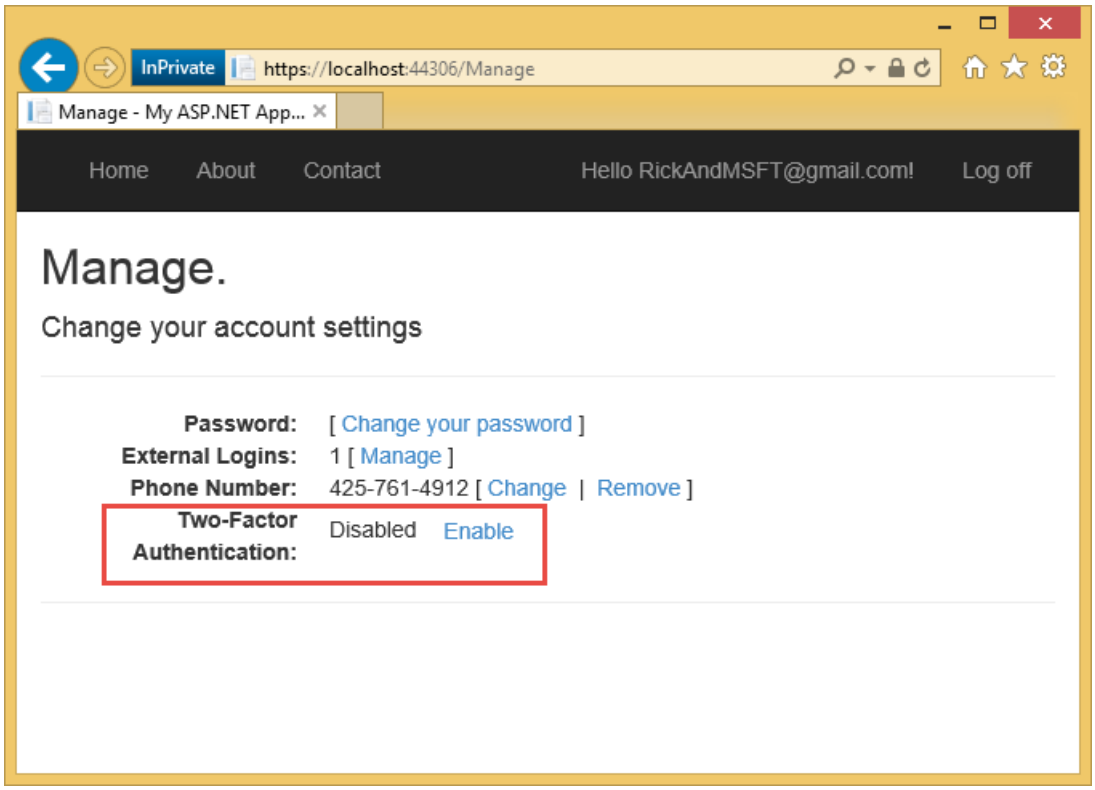
14. The Manage view shows your phone number was added.

Enable two-factor authentication

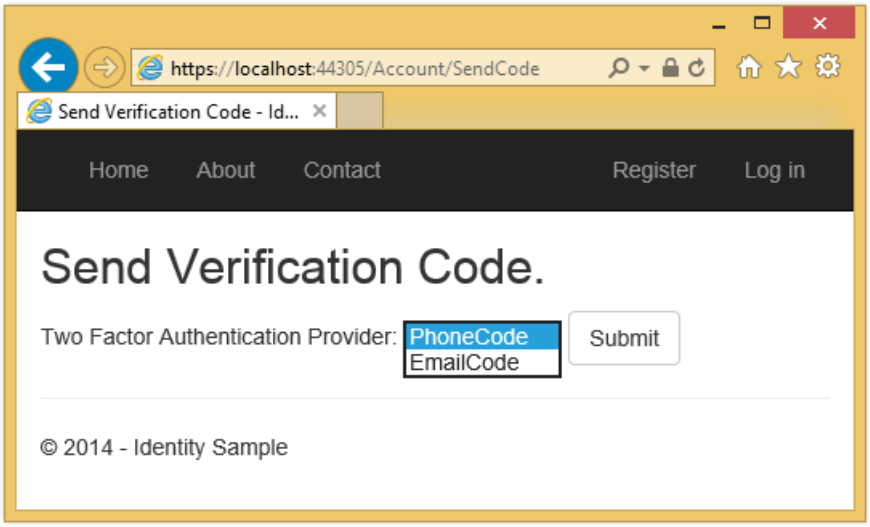
In the template generated app, you need to use the UI to enable two-factor authentication (2FA). To enable 2FA, click on your user ID (email alias) in the navigation bar.



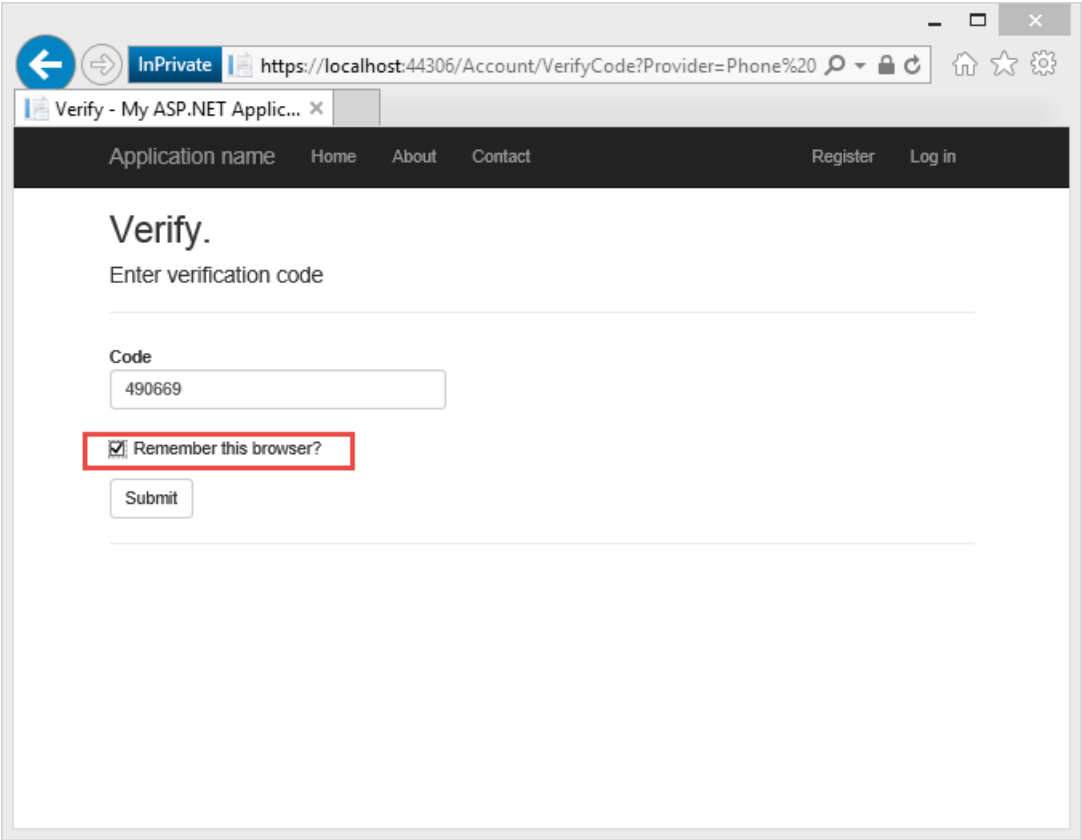
Click on enable 2FA.



Logout, then log back in. If you've enabled email (see my previous tutorial), you can select the SMS or email for 2FA.



The Verify Code page is displayed where you can enter the code (from SMS or email).



Clicking on the **Remember this browser** check box will exempt you from needing to use 2FA to log in when using the browser and device where you checked the box. As long as malicious users can't gain access to your device, enabling 2FA and clicking on the **Remember this browser** will provide you with convenient one step password access, while still retaining strong 2FA protection for all access from non-trusted devices. You can do this on any private device you regularly use.

This tutorial provides a quick introduction to enabling 2FA on a new ASP.NET MVC app. My tutorial Two-factor authentication using SMS and email with ASP.NET Identity goes into detail on the code behind the sample.

Additional Resources

- Two-factor authentication using SMS and email with ASP.NET Identity Goes into detail on Two-factor authentication
- Links to ASP.NET Identity Recommended Resources
- Account Confirmation and Password Recovery with ASP.NET Identity Goes into more detail on password

recovery and account confirmation.

- MVC 5 App with Facebook, Twitter, LinkedIn and Google OAuth2 Sign-on This tutorial shows you how to write an ASP.NET MVC 5 app with Facebook and Google OAuth 2 authorization. It also shows how to add additional data to the Identity database.
- Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL Database to Azure Web. This tutorial adds Azure deployment, how to secure your app with roles, how to use the membership API to add users and roles, and additional security features.
- Creating a Google app for OAuth 2 and connecting the app to the project
- Creating the app in Facebook and connecting the app to the project
- Setting up SSL in the Project
- How to set up your C# and ASP.NET MVC development environment

XSRF/CSRF Prevention in ASP.NET MVC and Web Pages

2/20/2020 • 15 minutes to read • Edit Online

by Rick Anderson

Cross-site request forgery (also known as XSRF or CSRF) is an attack against web-hosted applications whereby a malicious web site can influence the interaction between a client browser and a web site trusted by that browser. These attacks are made possible because web browsers will send authentication tokens automatically with every request to a web site. The canonical example is an authentication cookie, such as ASP.NET's Forms Authentication ticket. However, web sites which use any persistent authentication mechanism (such as Windows Authentication, Basic, and so forth) can be targeted by these attacks.

An XSRF attack is distinct from a phishing attack. Phishing attacks require interaction from the victim. In a phishing attack, a malicious web site will mimic the target web site, and the victim is fooled into providing sensitive information to the attacker. In an XSRF attack, there is often no interaction necessary from the victim. Rather, the attacker is relying on the browser automatically sending all relevant cookies to the destination web site.

For more information, see the Open Web Application Security Project(OWASP) XSRF.

Anatomy of an attack

To walk through an XSRF attack, consider a user who wants to perform some online banking transactions. This user first visits WoodgroveBank.com and logs in, at which point the response header will contain her authentication cookie:

```
HTTP/1.1 200 OK
Date: Mon, 18 Jun 2012 21:22:33 GMT
X-AspNet-Version: 4.0.30319
Set-Cookie: .ASPXAUTH={authentication-token}; path=/; secure; HttpOnly;
{ Cache-Control, Content-Type, Location, Server and other keys/values not listed. }
```

Because the authentication cookie is a session cookie, it will be automatically cleared by the browser when the browser process exits. However, until that time, the browser will automatically include the cookie with each request to WoodgroveBank.com. The user now wants to transfer $1000 to another account, so she fills out a form on the banking site, and the browser makes this request to the server:

```
POST /DoTransfer HTTP/1.1
Host: WoodgroveBank.com
Content-Type: application/x-www-form-urlencoded
Cookie: .ASPXAUTH={authentication-token}
toAcct=12345&amount=1,000.00
```

Because this operation has a side effect (it initiates a monetary transaction), the banking site has chosen to require an HTTP POST in order to initiate this operation. The server reads the authentication token from the request, looks up the current user's account number, verifies that sufficient funds exist, and then initiates the transaction into the destination account.

Her online banking complete, the user navigates away from the banking site and visits other locations on the

web. One of those sites – fabrikam.com – includes the following markup on a page embedded within an <iframe>:

```
<form id="theForm" action="https://WoodgroveBank.com/DoTransfer" method="post">
    <input type="hidden" name="toAcct" value="67890" />
    <input type="hidden" name="amount" value="250.00" />
</form>
<script type="text/javascript">
    document.getElementById('theForm').submit();
</script>
```

Which then causes the browser to make this request:

```
POST /DoTransfer HTTP/1.1
Host: WoodgroveBank.com
Content-Type: application/x-www-form-urlencoded
Cookie: .ASPXAUTH={authentication-token}
toAcct=67890&amount=250.00
```

The attacker is exploiting the fact that the user might still have a valid authentication token for the target web site, and she is using a small snippet of Javascript to cause the browser to make an HTTP POST to the target site automatically. If the authentication token is still valid, the banking site will initiate a transfer of $250 into the account of the attacker's choosing.

Ineffective mitigations

It is interesting to note that in the above scenario, the fact that WoodgroveBank.com was being accessed via SSL and had an SSL-only authentication cookie was insufficient to thwart the attack. The attacker is able to specify the URI scheme (https) in her <form> element, and the browser will continue to send unexpired cookies to the target site as long as those cookies are consistent with the URI scheme of the intended target.

One could argue that the user should simply not visit untrusted sites, as visiting only trusted sites is helps to remain safe online. There is some truth to this, but unfortunately this advice is not always practical. Perhaps the user "trusts" the local news site ConsolidatedMessenger. ConsolidatedMessenger.com and goes to visit that site instead, but that site has an XSS vulnerability which allows an attacker to inject the same snippet of code that was running on fabrikam.com.

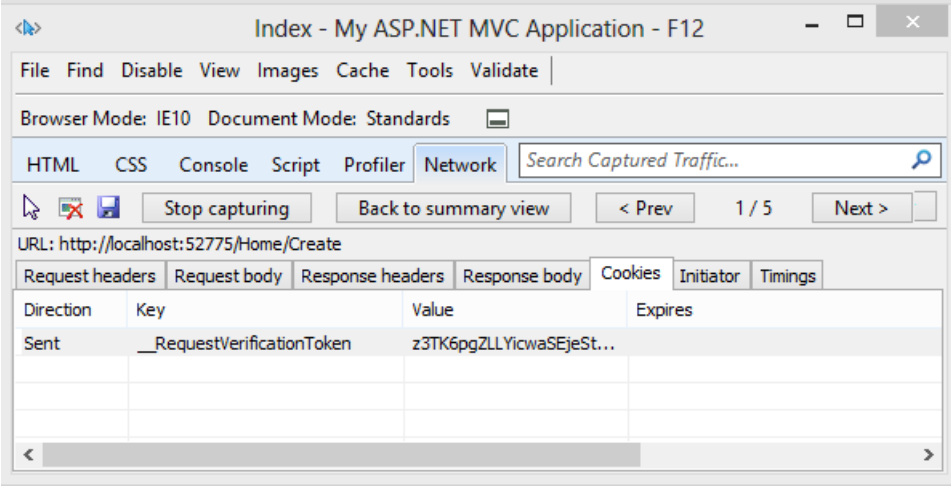
You can verify that incoming requests have a Referer header referencing your domain. This will stop requests unwittingly submitted from a third-party domain. However, some people disable their browser's Referer header for privacy reasons, and attackers can sometimes spoof that header if the victim has certain insecure software installed. Verifying the Referer header is not considered a secure approach to preventing XSRF attacks.

Web Stack Runtime XSRF mitigations

The ASP.NET Web Stack Runtime uses a variant of the synchronizer token pattern to defend against XSRF attacks. The general form of the synchronizer token pattern is that two anti-XSRF tokens are submitted to the server with each HTTP POST (In addition to the authentication token): one token as a cookie, and the other as a form value. The token values generated by the ASP.NET runtime are not deterministic or predictable by an attacker. When the tokens are submitted, the server will allow the request to proceed only if both tokens pass a comparison check.

The XSRF request verification *session token* is stored as an HTTP cookie and currently contains the following information in its payload:

- A security token, consisting of a random 128-bit identifier.
  The following image shows the XSRF request verification session token displayed with the Internet Explorer F12 developer tools: (Note this is the current implementation and is subject, even likely, to change.)



The *field token* is stored as an `<input type="hidden" />` and contains the following information in its payload:

- The logged-in user's username (if authenticated).
- Any additional data provided by an IAntiForgeryAdditionalDataProvider.

The payloads of the anti-XSRF tokens are encrypted and signed, so you can't view the username when using tools to examine the tokens. When the web application is targeting ASP.NET 4.0, cryptographic services are provided by the MachineKey.Encode routine. When the web application is targeting ASP.NET 4.5 or higher, cryptographic services are provided by the MachineKey.Protect routine, which offers better performance, extensibility, and security. See the following blog posts for more details:

- Cryptographic Improvements in ASP.NET 4.5, pt. 1
- Cryptographic Improvements in ASP.NET 4.5, pt. 2
- Cryptographic Improvements in ASP.NET 4.5, pt. 3

Generating the tokens

To generate the anti-XSRF tokens, call the @Html.AntiForgeryToken method from an MVC view or @AntiForgery.GetHtml() from a Razor page. The runtime will then perform the following steps:

1. If the current HTTP request already contains an anti-XSRF session token (the anti-XSRF cookie __RequestVerificationToken), the security token is extracted from it. If the HTTP request does not contain an anti-XSRF session token or if extraction of the security token fails, a new random anti-XSRF token will be generated.
2. An anti-XSRF field token is generated using the security token from step (1) above and the identity of the current logged-in user. (For more information on determining user identity, see the **Scenarios with special support** section below.) Additionally, if an IAntiForgeryAdditionalDataProvider is configured, the runtime will call its GetAdditionalData method and include the returned string in the field token. (See the **Configuration and extensibility** section for more information.)
3. If a new anti-XSRF token was generated in step (1), a new session token will be created to contain it and will be added to the outbound HTTP cookies collection. The field token from step (2) will be wrapped in an `<input type="hidden" />` element, and this HTML markup will be the return value of `Html.AntiForgeryToken()` or `AntiForgery.GetHtml()`.

Validating the tokens

To validate the incoming anti-XSRF tokens, the developer includes a ValidateAntiForgeryToken attribute on her MVC action or controller, or she calls `@AntiForgery.Validate()` from her Razor page. The runtime will perform the following steps:

1. The incoming session token and field token are read and the anti-XSRF token extracted from each. The anti-

XSRF tokens must be identical per step (2) in the generation routine.

2. If the current user is authenticated, her username is compared with the username stored in the field token. The usernames must match.
3. If an IAntiForgeryAdditionalDataProvider is configured, the runtime calls its *ValidateAdditionalData* method. The method must return the Boolean value *true*.

If validation succeeds, the request is allowed to proceed. If validation fails, the framework will throw an *HttpAntiForgeryException*.

Failure conditions

Starting with The ASP.NET Web Stack Runtime v2, any *HttpAntiForgeryException* that is thrown during validation will contain detailed information about what went wrong. The currently defined failure conditions are:

- The session token or form token is not present in the request.
- The session token or form token is unreadable. The most likely cause of this is a farm running mismatched versions of The ASP.NET Web Stack Runtime or a farm where the <machineKey> element in Web.config differs between machines. You can use a tool such as Fiddler to force this exception by tampering with either anti-XSRF token.
- The session token and field token were swapped.
- The session token and field token contain mismatched security tokens.
- The username embedded within the field token does not match the current logged-in user's username.
- The *IAntiForgeryAdditionalDataProvider.ValidateAdditionalData* method returned *false*.

The anti-XSRF facilities may also perform additional checking during token generation or validation, and failures during these checks may result in exceptions being thrown. See the WIF / ACS / claims-based authentication and **Configuration and extensibility** sections for more information.

Scenarios with special support

Anonymous authentication

The anti-XSRF system contains special support for anonymous users, where "anonymous" is defined as a user where the *IIdentity.IsAuthenticated* property returns *false*. Scenarios include providing XSRF protection to the login page (before the user is authenticated) and custom authentication schemes where the application uses a mechanism other than *IIdentity* to identify users.

To support these scenarios, recall that the session and field tokens are joined by a security token, which is a 128-bit randomly-generated opaque identifier. This security token is used to track an individual user's session as she navigates the site, so it effectively serves the purpose of an anonymous identifier. An empty string is used in place of the username for the generation and validation routines described above.

WIF / ACS / claims-based authentication

Normally, the *IIdentity* classes built in to the .NET Framework have the property that *IIdentity.Name* is sufficient to uniquely identify a particular user within a particular application. For example, *FormsIdentity.Name* returns the username stored in the membership database (which is unique for all applications depending on that database), *WindowsIdentity.Name* returns the domain-qualified identity of the user, and so on. These systems provide not only authentication; they also *identify* users to an application.

Claims-based authentication, on the other hand, does not necessarily require identifying a particular user. Instead, the *ClaimsPrincipal* and *ClaimsIdentity* types are associated with a set of *Claim* instances, where the individual claims might be "is 18+ years of age" or "is an administrator" to anything else. Since the user hasn't necessarily been identified, the runtime cannot use the *ClaimsIdentity.Name* property as a unique identifier for this particular user. The team has seen real-world examples where *ClaimsIdentity.Name* returns *null*, returns a friendly (display) name, or otherwise returns a string that isn't appropriate for use as a unique identifier for the

user.

Many of deployments which use claims-based authentication are using Azure Access Control Service (ACS) in particular. ACS allows the developer to configure individual *identity providers* (such as ADFS, the Microsoft Account provider, OpenID providers like Yahoo!, etc.), and the identity providers return *name identifiers*. These name identifiers may contain Personally Identifiable Information (PII) like an email address, or they could be anonymized like a Private Personal Identifier (PPID). Regardless, the tuple (identity provider, name identifier) sufficiently serves as an appropriate tracking token for a particular user while she is browsing the site, so the ASP.NET Web Stack Runtime can use the tuple in place of the username when generating and validating anti-XSRF field tokens. The particular URIs for the identity provider and the name identifier are :

- `https://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider`
- `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier`

(see this ACS doc page for more info.)

When generating or validating a token, the ASP.NET Web Stack Runtime will at runtime try binding to the types:

- `Microsoft.IdentityModel.Claims.IClaimsIdentity, Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35`
  (For the WIF SDK.)
- `System.Security.Claims.ClaimsIdentity` (For .NET 4.5).

If these types exist, and if the current user's *IIIIdentity* implements or subclasses one of these types, the anti-XSRF facility will use the (identity provider, name identifier) tuple in place of the username when generating and validating the tokens. If no such tuple is present, the request will fail with an error describing to the developer how to configure the anti-XSRF system to understand the particular claims-based authentication mechanism in use. See the **Configuration and extensibility** section for more information.

OAuth / OpenID authentication

Finally, the anti-XSRF facility has special support for applications which use OAuth or OpenID authentication. This support is heuristic-based: if the current *IIdentity.Name* begins with http:// or https://, then username comparisons will be done using an Ordinal comparer rather than the default OrdinalIgnoreCase comparer.

Configuration and extensibility

Occasionally, developers may want tighter control over the anti-XSRF generation and validation behaviors. For example, perhaps the MVC and Web Pages helpers' default behavior of automatically adding HTTP cookies to the response is undesirable, and the developer may wish to persist the tokens elsewhere. There exist two APIs to assist with this:

```
AntiForgery.GetTokens(string oldCookieToken, out string newCookieToken, out string formToken);
AntiForgery.Validate(string cookieToken, string formToken);
```

The *GetTokens* method takes as input an existing XSRF request verification session token (which may be null) and produces as output a new XSRF request verification session token and field token. The tokens are simply opaque strings with no decoration; the *formToken* value will for instance not be wrapped in an <input> tag. The *newCookieToken* value may be null; if this occurs, then the *oldCookieToken* value is still valid and no new response cookie need be set. The caller of *GetTokens* is responsible for persisting any necessary response cookies or generating any necessary markup; the *GetTokens* method itself will not alter the response as a side effect. The *Validate* method takes the incoming session and field tokens and runs the aforementioned validation logic over them.

AntiForgeryConfig

The developer may configure the anti-XSRF system from Application_Start. Configuration is programmatic. The properties of the static *AntiForgeryConfig* type are described below. Most users using claims will want to set the

UniqueClaimTypeIdentifier property.

PROPERTY	DESCRIPTION
AdditionalDataProvider	An IAntiForgeryAdditionalDataProvider that provides additional data during token generation and consumes additional data during token validation. The default value is *null*. For more information, see the IAntiForgeryAdditionalDataProvider section.
CookieName	A string that provides the name of the HTTP cookie that is used to store the anti-XSRF session token. If this value is not set, a name will be automatically generated based on the application's deployed virtual path. The default value is *null*.
RequireSsl	A Boolean that dictates whether the anti-XSRF tokens are required to be submitted over an SSL-secured channel. If this value is *true*, any automatically-generated cookies will have the "secure" flag set, and the anti-XSRF APIs will throw if called from within a request that is not submitted via SSL. The default value is *false*.
SuppressIdentityHeuristicChecks	A Boolean that dictates whether the anti-XSRF system should deactivate its support for claims-based identities. If this value is *true*, the system will assume that *IIdentity.Name* is appropriate for use as a unique per-user identifier and will not try to special-case *IClaimsIdentity* or *ClClaimsIdentity* as described in the WIF / ACS / claims-based authentication section. The default value is `false`.
UniqueClaimTypeIdentifier	A string that indicates which claim type is appropriate for use as a unique per-user identifier. If this value is set and the current *IIdentity* is claims-based, the system will attempt to extract a claim of the type specified by *UniqueClaimTypeIdentifier*, and the corresponding value will be used in place of the user's username when generating the field token. If the claim type is not found, the system will fail the request. The default value is *null*, which indicates that the system should use the (identity provider, name identifier) tuple as previously described in place of the user's username.

IAntiForgeryAdditionalDataProvider

The *IAntiForgeryAdditionalDataProvider* type allows developers to extend the behavior of the anti-XSRF system by round-tripping additional data in each token. The *GetAdditionalData* method is called each time a field token is generated, and the return value is embedded within the generated token. An implementer could return a timestamp, a nonce, or any other value she wishes from this method.

Similarly, the *ValidateAdditionalData* method is called each time a field token is validated, and the "additional data" string that was embedded within the token is passed to the method. The validation routine could implement a timeout (by checking the current time against the time that was stored when the token was created), a nonce checking routine, or any other desired logic.

Design decisions and security considerations

The security token that links the session and field tokens is technically only necessary when trying to protect anonymous / unauthenticated users against XSRF attacks. When the user is authenticated, the authentication token itself (presumably submitted in the form of a cookie) could be used as one half of a synchronizer token pair. However, there are valid scenarios for protecting login pages hit by unauthenticated users, and the anti-XSRF

logic was made simpler by always generating and validating the security token, even for authenticated users. It also does provide some additional protection in the event that a field token is ever compromised by an attacker, as setting or guessing the session token would be another hurdle for the attacker to overcome.

Developers should use caution when multiple applications are hosted in a single domain. For example, even though *example1.cloudapp.net* and *example2.cloudapp.net* are different hosts, there is an implicit trust relationship between all hosts under the **.cloudapp.net* domain. This implicit trust relationship allows potentially untrusted hosts to affect each other's cookies (the same-origin policies that govern AJAX requests do not necessarily apply to HTTP cookies). The ASP.NET Web Stack Runtime provides some mitigation in that the username is embedded into the field token, so even if a malicious subdomain is able to overwrite a session token it will be unable to generate a valid field token for the user. However, when hosted in such an environment the built-in anti-XSRF routines still cannot defend against session hijacking or login XSRF.

The anti-XSRF routines currently do not defend against clickjacking. Applications that wish to defend themselves against clickjacking may easily do so by sending an X-Frame-Options: SAMEORIGIN header with each response. This header is supported by all recent browsers. For more information, see the IE blog, the SDL blog, and OWASP. The ASP.NET Web Stack Runtime may in some future release make the MVC and Web Pages anti-XSRF helpers automatically set this header so that applications are automatically protected against this attack.

Web developers should continue to ensure that their site is not vulnerable to XSS attacks. XSS attacks are very powerful, and a successful exploit would also break the ASP.NET Web Stack Runtime defenses against XSRF attacks.

Acknowledgment

@LeviBroderick, who wrote much of the ASP.NET security code the bulk of this information.

Preventing Open Redirection Attacks (C#)

2/20/2020 • 7 minutes to read • Edit Online

by Jon Galloway

This tutorial explains how you can prevent open redirection attacks in your ASP.NET MVC applications. This tutorial discusses the changes that have been made in the AccountController in ASP.NET MVC 3 and demonstrates how you can apply these changes in your existing ASP.NET MVC 1.0 and 2 applications.

What is an Open Redirection Attack?

Any web application that redirects to a URL that is specified via the request such as the querystring or form data can potentially be tampered with to redirect users to an external, malicious URL. This tampering is called an open redirection attack.

Whenever your application logic redirects to a specified URL, you must verify that the redirection URL hasn't been tampered with. The login used in the default AccountController for both ASP.NET MVC 1.0 and ASP.NET MVC 2 is vulnerable to open redirection attacks. Fortunately, it is easy to update your existing applications to use the corrections from the ASP.NET MVC 3 Preview.

To understand the vulnerability, let's look at how the login redirection works in a default ASP.NET MVC 2 Web Application project. In this application, attempting to visit a controller action that has the [Authorize] attribute will redirect unauthorized users to the /Account/LogOn view. This redirect to /Account/LogOn will include a returnUrl querystring parameter so that the user can be returned to the originally requested URL after they have successfully logged in.

In the screenshot below, we can see that an attempt to access the /Account/ChangePassword view when not logged in results in a redirect to /Account/LogOn?ReturnUrl=%2fAccount%2fChangePassword%2f.

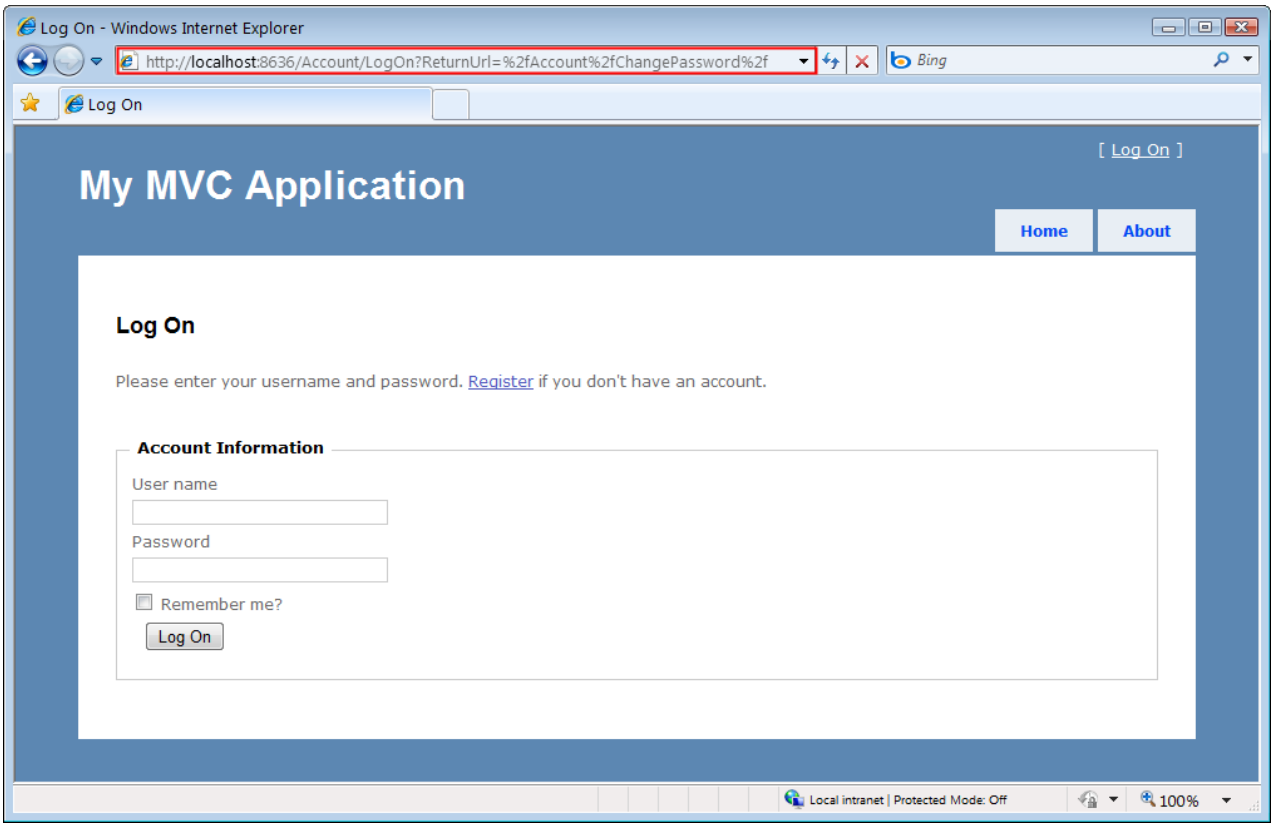


Figure 01: Login page with an open redirection

Since the ReturnUrl querystring parameter is not validated, an attacker can modify it to inject any URL address into the parameter to conduct an open redirection attack. To demonstrate this, we can modify the ReturnUrl parameter to http://bing.com, so the resulting login URL will be /Account/LogOn?ReturnUrl=http://www.bing.com/. Upon successfully logging in to the site, we are redirected to http://bing.com. Since this redirection is not validated, it could instead point to a malicious site that attempts to trick the user.

A more complex Open Redirection Attack

Open redirection attacks are especially dangerous because an attacker knows that we're trying to log into a specific website, which makes us vulnerable to a phishing attack. For example, an attacker could send malicious emails to website users in an attempt to capture their passwords. Let's look at how this would work on the NerdDinner site. (Note that the live NerdDinner site has been updated to protect against open redirection attacks.)

First, an attacker sends us a link to the login page on NerdDinner that includes a redirect to their forged page:

http://nerddinner.com/Account/LogOn?returnUrl=http://nerddiner.com/Account/LogOn

Note that the return URL points to nerddiner.com, which is missing an "n" from the word dinner. In this example, this is a domain that the attacker controls. When we access the above link, we're taken to the legitimate NerdDinner.com login page.

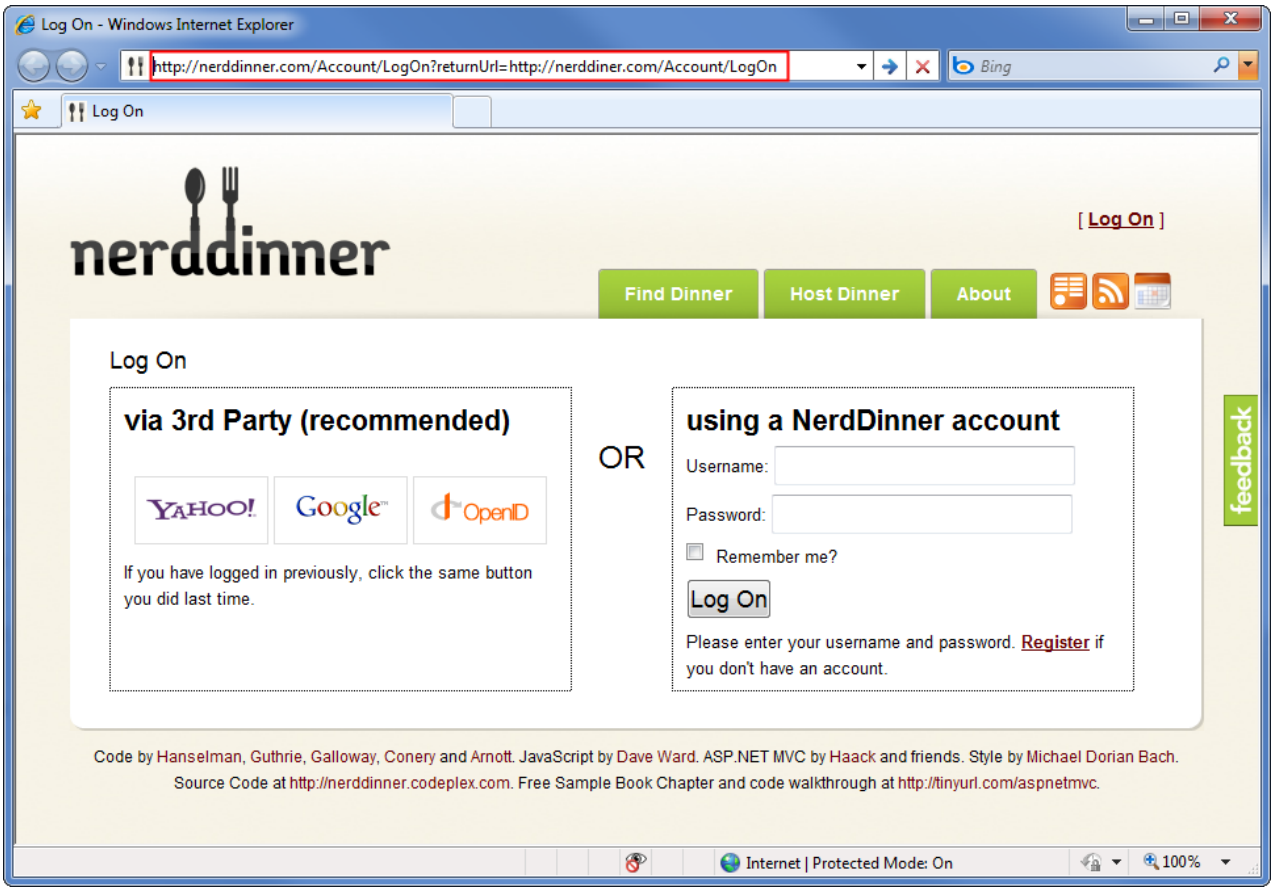


Figure 02: NerdDinner login page with an open redirection

When we correctly log in, the ASP.NET MVC AccountController's LogOn action redirects us to the URL specified in the returnUrl querystring parameter. In this case, it's the URL that the attacker has entered, which is http://nerddiner.com/Account/LogOn. Unless we're extremely watchful, it's very likely we won't notice this, especially because the attacker has been careful to make sure that their forged page looks exactly like the legitimate login page. This login page includes an error message requesting that we login again. Clumsy us, we must have mistyped our password.

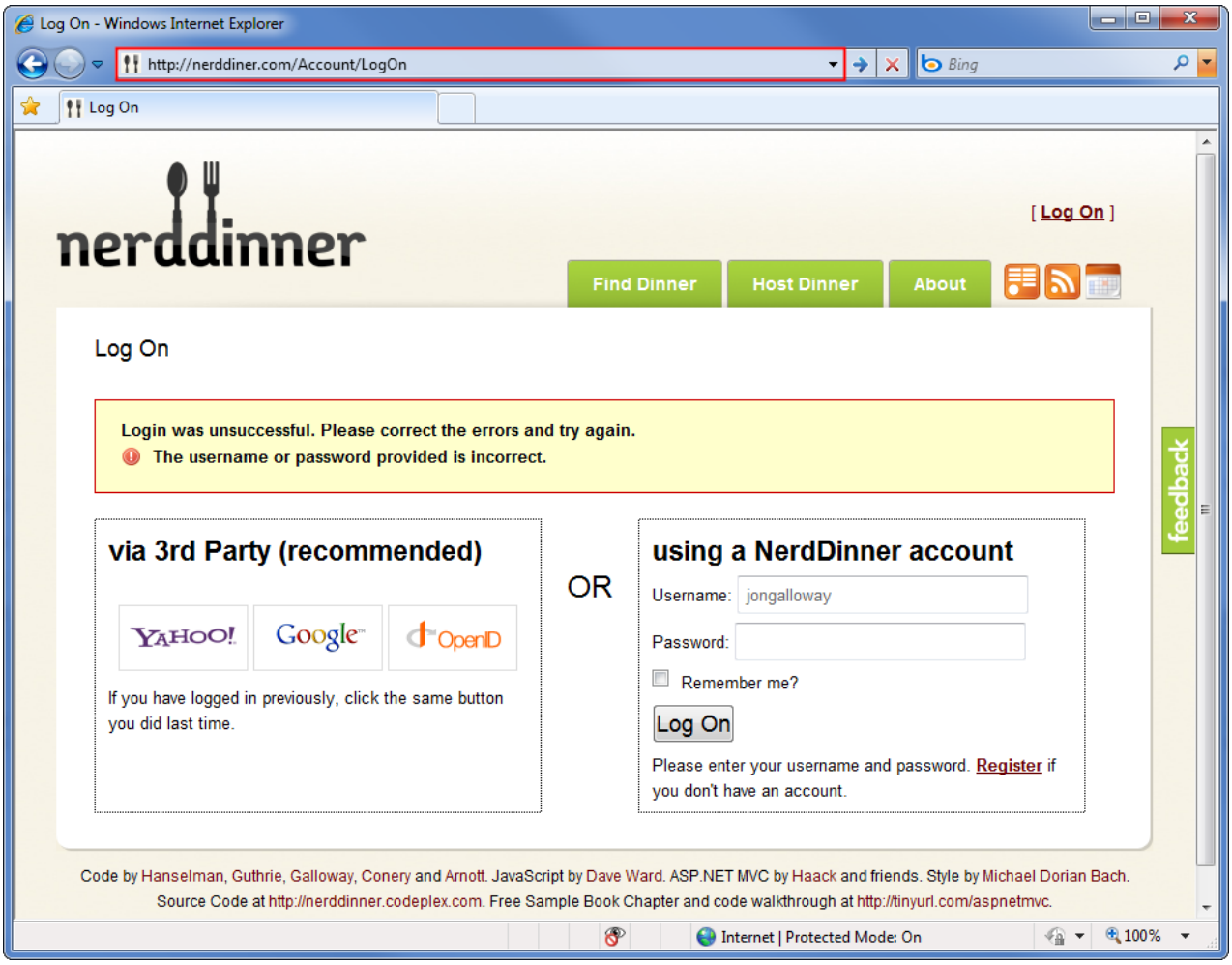


Figure 03: Forged NerdDinner Login screen

When we retype our user name and password, the forged login page saves the information and sends us back to the legitimate NerdDinner.com site. At this point, the NerdDinner.com site has already authenticated us, so the forged login page can redirect directly to that page. The end result is that the attacker has our user name and password, and we are unaware that we've provided it to them.

Looking at the vulnerable code in the AccountController LogOn Action

The code for the LogOn action in an ASP.NET MVC 2 application is shown below. Note that upon a successful login, the controller returns a redirect to the returnUrl. You can see that no validation is being performed against the returnUrl parameter.

Listing 1 – ASP.NET MVC 2 LogOn action in `AccountController.cs`

```
[HttpPost]
public ActionResult LogOn(LogOnModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        if (MembershipService.ValidateUser(model.UserName, model.Password))
        {
            FormsService.SignIn(model.UserName, model.RememberMe);
            if (!String.IsNullOrEmpty(returnUrl))
            {
                return Redirect(returnUrl);
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
        else
        {
            ModelState.AddModelError("", "The user name or password provided is incorrect.");
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Now let's look at the changes to the ASP.NET MVC 3 LogOn action. This code has been changed to validate the returnUrl parameter by calling a new method in the System.Web.Mvc.Url helper class named `IsLocalUrl()`.

Listing 2 – ASP.NET MVC 3 LogOn action in `AccountController.cs`

```
[HttpPost]
public ActionResult LogOn(LogOnModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        if (MembershipService.ValidateUser(model.UserName, model.Password))
        {
            FormsService.SignIn(model.UserName, model.RememberMe);
            if (Url.IsLocalUrl(returnUrl))
            {
                return Redirect(returnUrl);
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
        else
        {
            ModelState.AddModelError("",
        "The user name or password provided is incorrect.");
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

This has been changed to validate the return URL parameter by calling a new method in the System.Web.Mvc.Url helper class, `IsLocalUrl()`.

Protecting Your ASP.NET MVC 1.0 and MVC 2 Applications

We can take advantage of the ASP.NET MVC 3 changes in our existing ASP.NET MVC 1.0 and 2 applications by adding the IsLocalUrl() helper method and updating the LogOn action to validate the returnUrl parameter.

The UrlHelper IsLocalUrl() method actually just calling into a method in System.Web.WebPages, as this validation is also used by ASP.NET Web Pages applications.

Listing 3 – The IsLocalUrl() method from the ASP.NET MVC 3 UrlHelper `class`

```
public bool IsLocalUrl(string url) {
    return System.Web.WebPages.RequestExtensions.IsUrlLocalToHost(
        RequestContext.HttpContext.Request, url);
}
```

The IsUrlLocalToHost method contains the actual validation logic, as shown in Listing 4.

Listing 4 – IsUrlLocalToHost() method from the System.Web.WebPages RequestExtensions class

```
public static bool IsUrlLocalToHost(this HttpRequestBase request, string url)
{
   return !url.IsEmpty() &&
          ((url[0] == '/' && (url.Length == 1 ||
           (url[1] != '/' && url[1] != '\\'))) ||   // "/" or "/foo" but not "//" or "/\"
           (url.Length > 1 &&
            url[0] == '~' && url[1] == '/'));   // "~/" or "~/foo"
}
```

In our ASP.NET MVC 1.0 or 2 application, we'll add a IsLocalUrl() method to the AccountController, but you're encouraged to add it to a separate helper class if possible. We will make two small changes to the ASP.NET MVC 3 version of IsLocalUrl() so that it will work inside the AccountController. First, we'll change it from a public method to a private method, since public methods in controllers can be accessed as controller actions. Second, we'll modify the call that checks the URL host against the application host. That call makes use of a local RequestContext field in the UrlHelper class. Instead of using this.RequestContext.HttpContext.Request.Url.Host, we will use this.Request.Url.Host. The following code shows the modified IsLocalUrl() method for use with a controller class in ASP.NET MVC 1.0 and 2 applications.

Listing 5 – IsLocalUrl() method, which is modified for use with an MVC Controller class

```
private bool IsLocalUrl(string url)
{
   if (string.IsNullOrEmpty(url))
   {
      return false;
   }
   else
   {
      return ((url[0] == '/' && (url.Length == 1 ||
              (url[1] != '/' && url[1] != '\\'))) ||   // "/" or "/foo" but not "//" or "/\"
              (url.Length > 1 &&
               url[0] == '~' && url[1] == '/'));   // "~/" or "~/foo"
   }
}
```

Now that the IsLocalUrl() method is in place, we can call it from our LogOn action to validate the returnUrl parameter, as shown in the following code.

Listing 6 – Updated LogOn method which validates the returnUrl parameter

```
[HttpPost]
public ActionResult LogOn(LogOnModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        if (MembershipService.ValidateUser(model.UserName, model.Password))
        {
            FormsService.SignIn(model.UserName, model.RememberMe);
            if (IsLocalUrl(returnUrl))
            {
                return Redirect(returnUrl);
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
        else
        {
            ModelState.AddModelError("",
            "The user name or password provided is incorrect.");
        }
    }
}
```

Now we can test an open redirection attack by attempting to log in using an external return URL. Let's use /Account/LogOn?ReturnUrl=http://www.bing.com/ again.

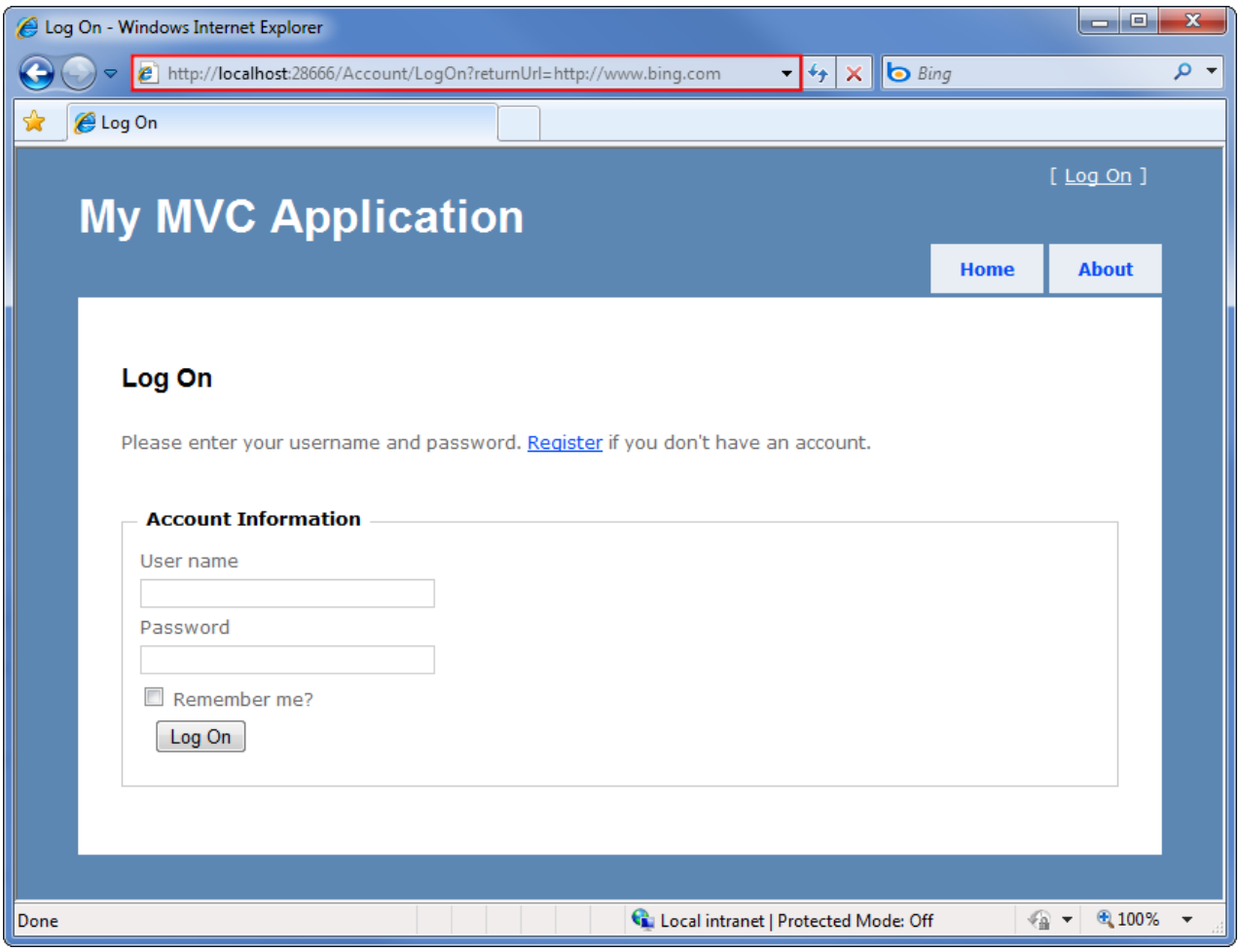


Figure 04: Testing the updated LogOn Action

After successfully logging in, we are redirected to the Home/Index Controller action rather than the external URL.

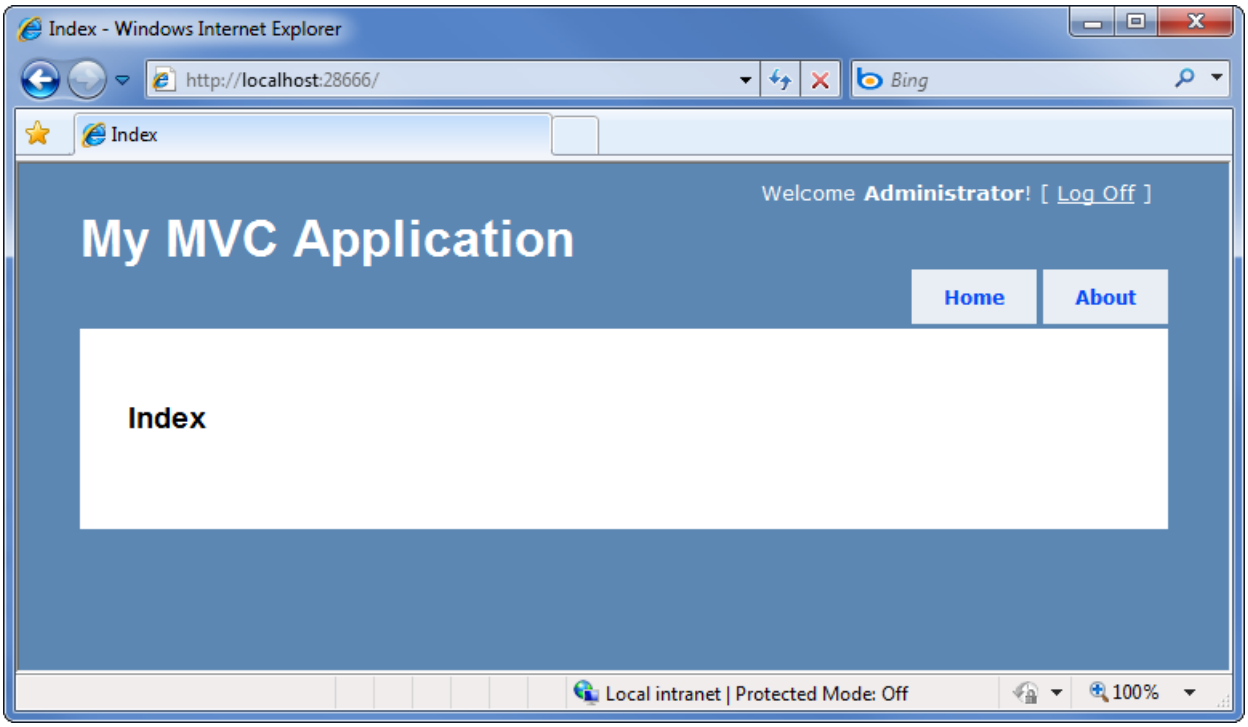


Figure 05: Open Redirection attack defeated

Summary

Open redirection attacks can occur when redirection URLs are passed as parameters in the URL for an application. The ASP.NET MVC 3 template includes code to protect against open redirection attacks. You can add this code with some modification to ASP.NET MVC 1.0 and 2 applications. To protect against open redirection attacks when logging into ASP.NET 1.0 and 2 applications, add a IsLocalUrl() method and validate the returnUrl parameter in the LogOn action.

ASP.NET MVC Deployment

2/20/2020 • 2 minutes to read • Edit Online

How to deploy ASP.NET MVC 5.0 projects to a server.

- Web Deployment Overview for Visual Studio and ASP.NET
- Migrating ASP.NET MVC Applications to Windows Containers
- Continuous Integration and Continuous Delivery (Building Real-World Cloud Apps with Azure)
- ASP.NET Web Deployment using Visual Studio
- ASP.NET 4 - Enterprise Deployment Introduction
- ASP.NET 4 - Enterprise Deployment Series 1
- ASP.NET 4 - Enterprise Deployment Series 2 Configuring Servers
- ASP.NET 4 - Enterprise Deployment Series 3 Configuring TFS
- ASP.NET 4 - Enterprise Deployment Series 4 Advanced Topics
- ASP.NET Web Deployment - Recommended Resources

Migrating ASP.NET MVC Applications to Windows Containers

4/6/2020 • 4 minutes to read • Edit Online

Running an existing .NET Framework-based application in a Windows container doesn't require any changes to your app. To run your app in a Windows container you create a Docker image containing your app and start the container. This topic explains how to take an existing ASP.NET MVC application and deploy it in a Windows container.

You start with an existing ASP.NET MVC app, then build the published assets using Visual Studio. You use Docker to create the image that contains and runs your app. You'll browse to the site running in a Windows container and verify the app is working.

This article assumes a basic understanding of Docker. You can learn about Docker by reading the Docker Overview.

The app you'll run in a container is a simple website that answers questions randomly. This app is a basic MVC application with no authentication or database storage; it lets you focus on moving the web tier to a container. Future topics will show how to move and manage persistent storage in containerized applications.

Moving your application involves these steps:

1. Creating a publish task to build the assets for an image.
2. Building a Docker image that will run your application.
3. Starting a Docker container that runs your image.
4. Verifying the application using your browser.

The finished application is on GitHub.

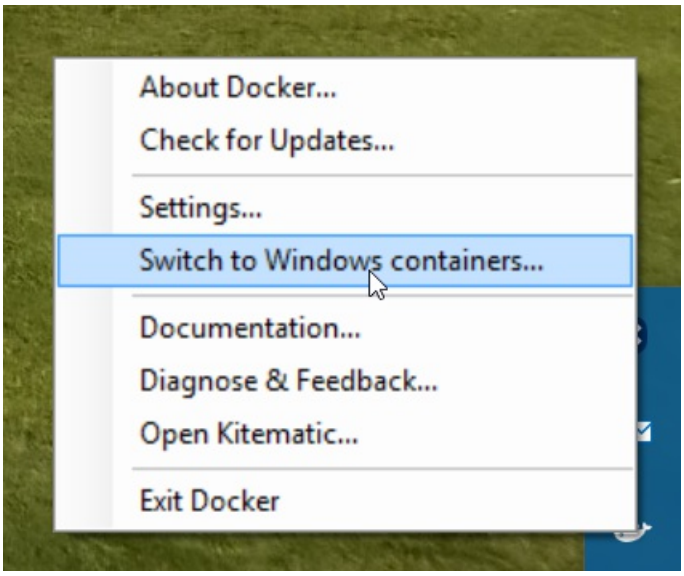
Prerequisites

The development machine must have the following software:

- Windows 10 Anniversary Update (or higher) or Windows Server 2016 (or higher)
- Docker for Windows - version Stable 1.13.0 or 1.12 Beta 26 (or newer versions)
- Visual Studio 2017

> **IMPORTANT**
>
> If you are using Windows Server 2016, follow the instructions for Container Host Deployment - Windows Server.

After installing and starting Docker, right-click on the tray icon and select **Switch to Windows containers**. This is required to run Docker images based on Windows. This command takes a few seconds to execute:

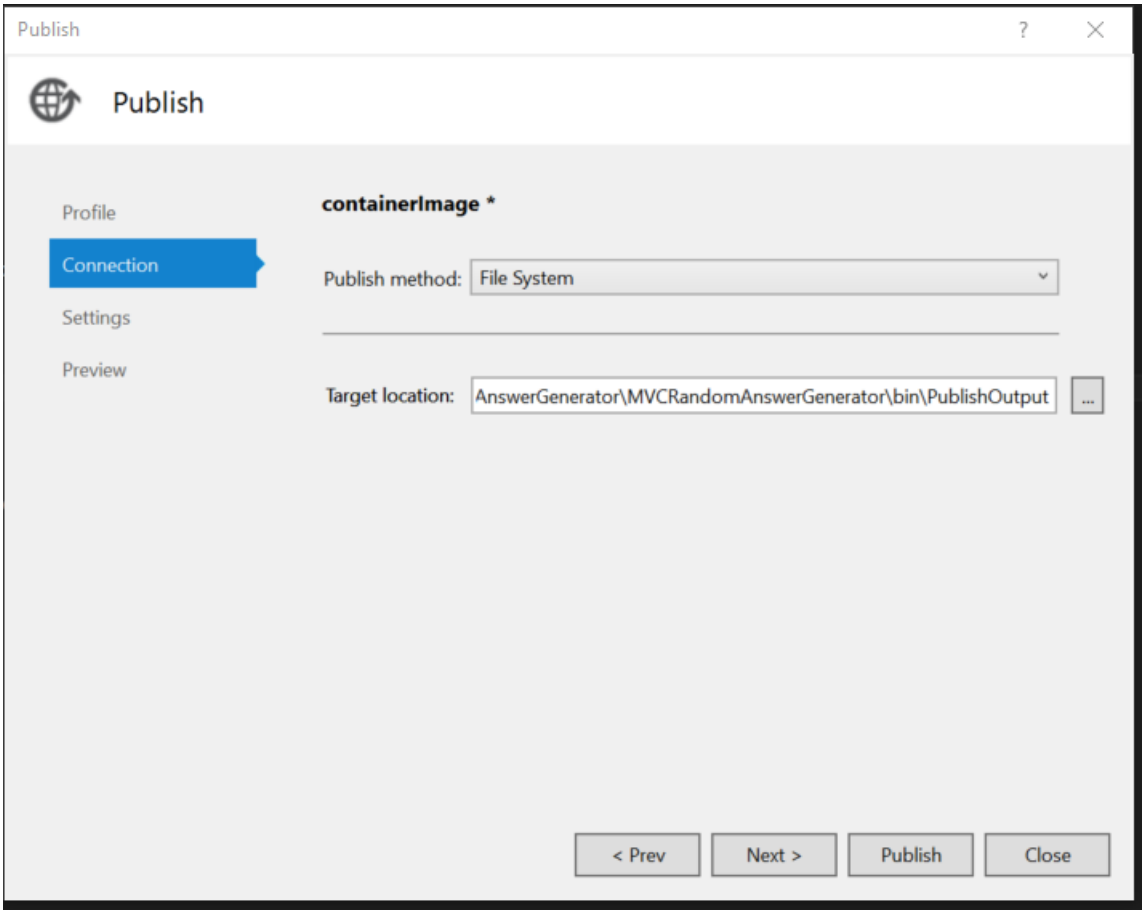


Publish script

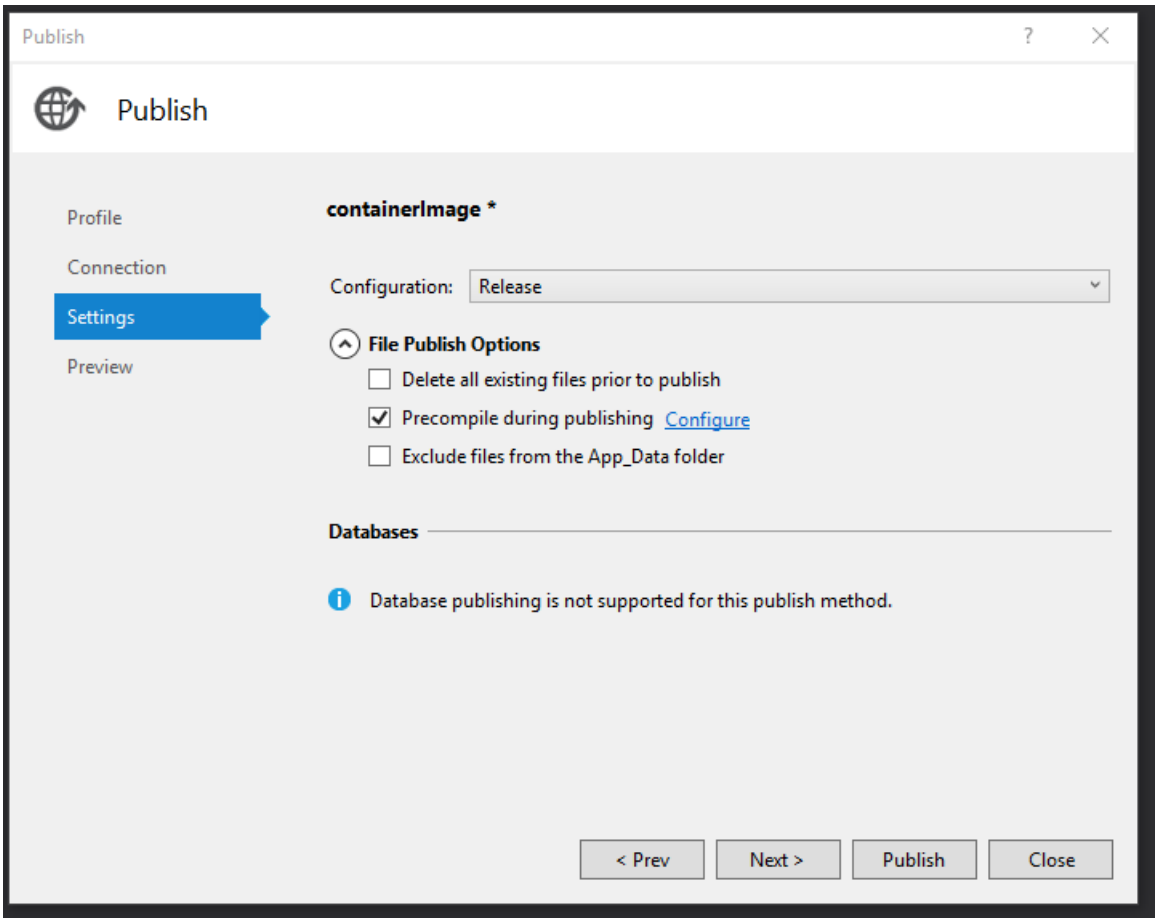
Collect all the assets that you need to load into a Docker image in one place. You can use the Visual Studio **Publish** command to create a publish profile for your app. This profile will put all the assets in one directory tree that you copy to your target image later in this tutorial.

Publish Steps

1. Right click on the web project in Visual Studio, and select **Publish**.
2. Click the **Custom profile button**, and then select **File System** as the method.
3. Choose the directory. By convention, the downloaded sample uses `bin\Release\PublishOutput`.



Open the **File Publish Options** section of the **Settings** tab. Select **Precompile during publishing**. This optimization means that you'll be compiling views in the Docker container, you are copying the precompiled views.



Click **Publish**, and Visual Studio will copy all the needed assets to the destination folder.

Build the image

Create a new file named *Dockerfile* to define your Docker image. *Dockerfile* contains instructions to build the final image and includes any base image names, required components, the app you want to run, and other configuration images. *Dockerfile* is the input to the `docker build` command that creates the image.

For this exercise, you will build an image based on the `microsoft/aspnet` image located on Docker Hub. The base image, `microsoft/aspnet`, is a Windows Server image. It contains Windows Server Core, IIS, and ASP.NET 4.7.2. When you run this image in your container, it will automatically start IIS and installed websites.

The Dockerfile that creates your image looks like this:

```
# The `FROM` instruction specifies the base image. You are
# extending the `microsoft/aspnet` image.

FROM microsoft/aspnet

# The final instruction copies the site you published earlier into the container.
COPY ./bin/Release/PublishOutput/ /inetpub/wwwroot
```

There is no `ENTRYPOINT` command in this Dockerfile. You don't need one. When running Windows Server with IIS, the IIS process is the entrypoint, which is configured to start in the aspnet base image.

Run the Docker build command to create the image that runs your ASP.NET app. To do this, open a PowerShell window in the directory of your project and type the following command in the solution directory:

```
docker build -t mvcrandomanswers .
```

This command will build the new image using the instructions in your Dockerfile, naming (-t tagging) the image as mvcrandomanswers. This may include pulling the base image from Docker Hub, and then adding your app to that image.

Once that command completes, you can run the `docker images` command to see information on the new image:

```
REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
mvcrandomanswers              latest              86838648aab6        2 minutes ago       10.1 GB
```

The IMAGE ID will be different on your machine. Now, let's run the app.

Start a container

Start a container by executing the following `docker run` command:

```
docker run -d --name randomanswers mvcrandomanswers
```

The `-d` argument tells Docker to start the image in detached mode. That means the Docker image runs disconnected from the current shell.

In many docker examples, you may see -p to map the container and host ports. The default aspnet image has already configured the container to listen on port 80 and expose it.

The `--name randomanswers` gives a name to the running container. You can use this name instead of the container ID in most commands.

The `mvcrandomanswers` is the name of the image to start.

Verify in the browser

Once the container starts, connect to the running container using `http://localhost` in the example shown. Type that URL into your browser, and you should see the running site.

> **NOTE**
> Some VPN or proxy software may prevent you from navigating to your site. You can temporarily disable it to make sure your container is working.

The sample directory on GitHub contains a PowerShell script that executes these commands for you. Open a PowerShell window, change directory to your solution directory, and type:

```
./run.ps1
```

The command above builds the image, displays the list of images on your machine, and starts a container.

To stop your container, issue a `docker stop` command:

```
docker stop randomanswers
```

To remove the container, issue a `docker rm` command:

```
docker rm randomanswers
```

ASP.NET MVC Performance

2/20/2020 • 2 minutes to read • Edit Online

Resources for optimizing performance of ASP.NET MVC applications.

- Using Asynchronous Methods in ASP.NET MVC 4
- Profile and debug your ASP.NET MVC app with Glimpse
- Bundling and Minification

Profile and debug your ASP.NET MVC app with Glimpse

2/20/2020 • 5 minutes to read • Edit Online

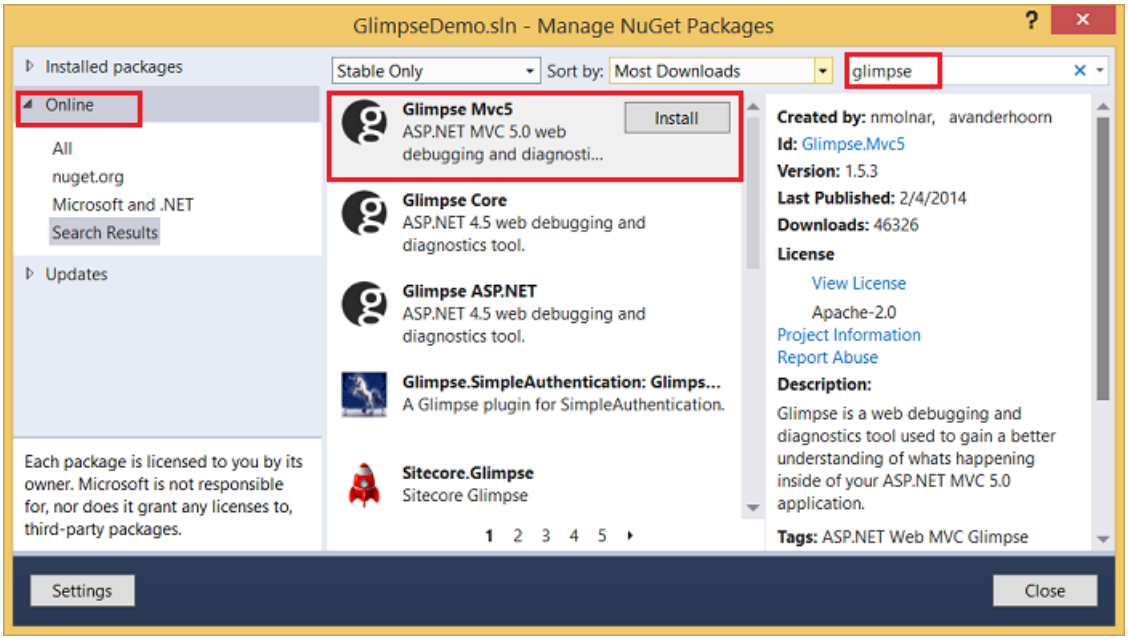
by Rick Anderson

Glimpse is a thriving and growing family of open source NuGet packages that provides detailed performance, debugging and diagnostic information for ASP.NET apps. It's trivial to install, lightweight, ultra-fast, and displays key performance metrics at the bottom of every page. It allows you to drill down into your app when you need to find out what's going on at the server. Glimpse provides so much valuable information we recommend you use it throughout your development cycle, including your Azure test environment. While Fiddler and the F-12 development tools provide a client side view, Glimpse provides a detailed view from the server. This tutorial will focus on using the Glimpse ASP.NET MVC and EF packages, but many other packages are available. Where possible I will link to the appropriate Glimpse docs which I help maintain. Glimpse is an open source project, you too can contribute to the source code and the docs.

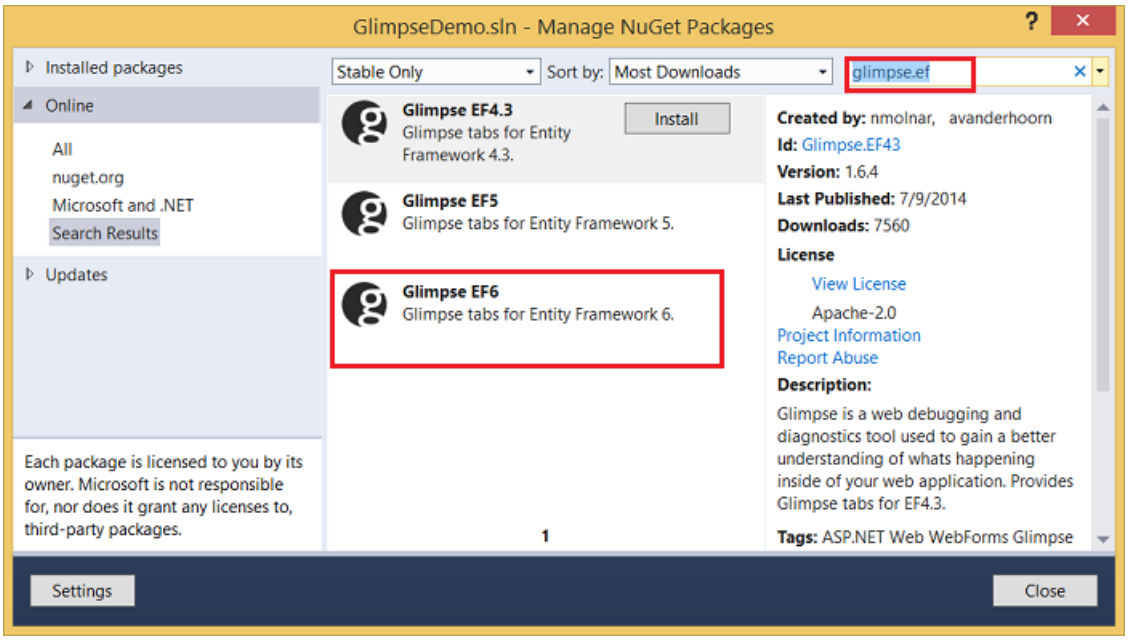
- Installing Glimpse
- Enable Glimpse for localhost
- The Timeline tab
- Model Binding
- Routes
- Using Glimpse on Azure
- Additional Resources

Installing Glimpse

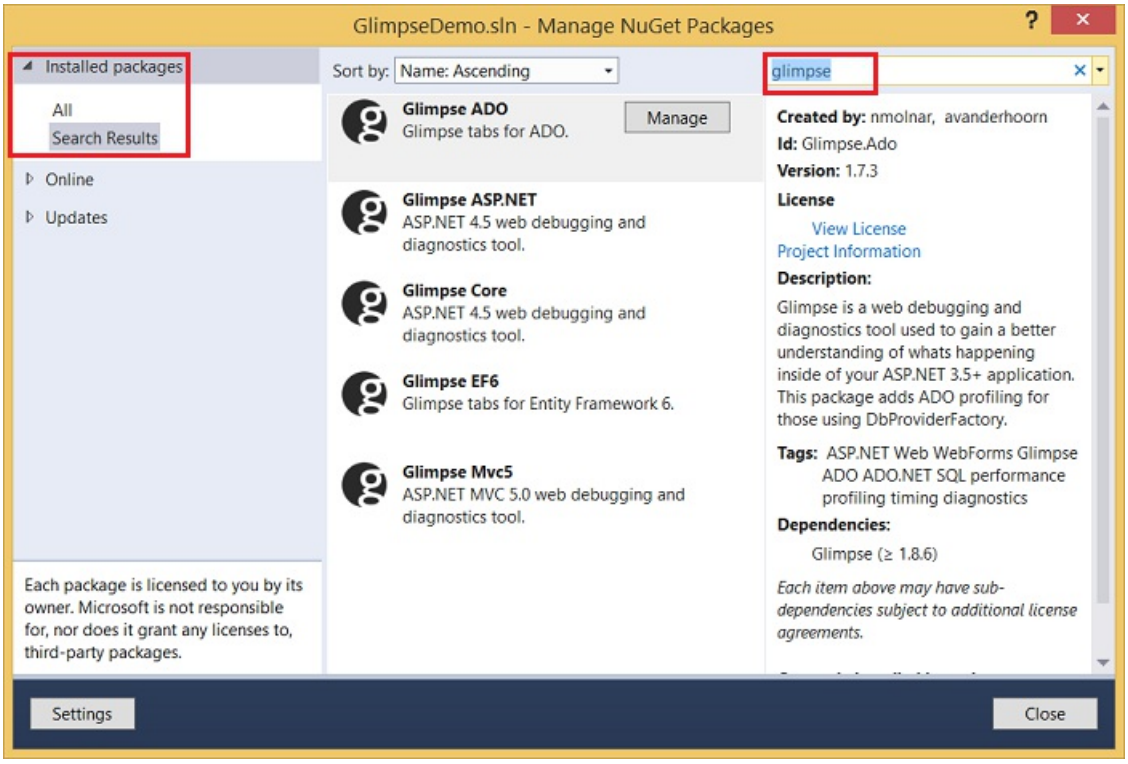
You can install Glimpse from the NuGet package manager console or from the **Manage NuGet Packages** console. For this demo, I'll install the Mvc5 and EF6 packages:



Search for *Glimpse.EF*



By selecting **Installed packages**, you can see the Glimpse dependent modules installed:

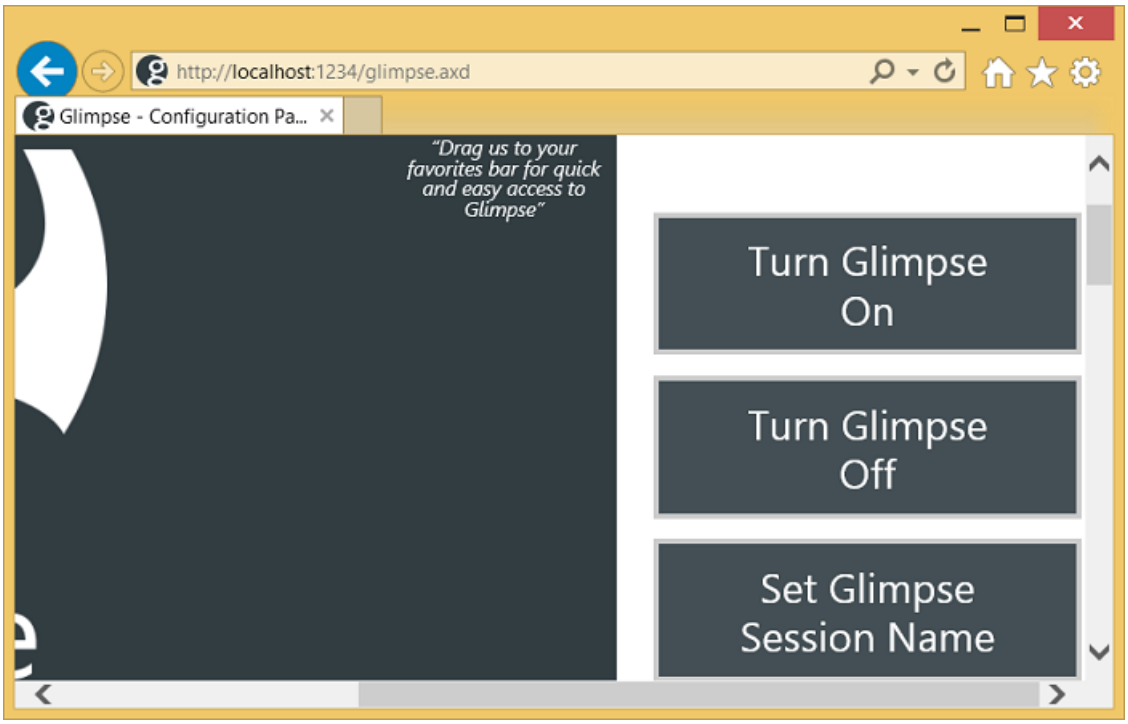


The following commands install Glimpse MVC5 and EF6 modules from the package manager console:

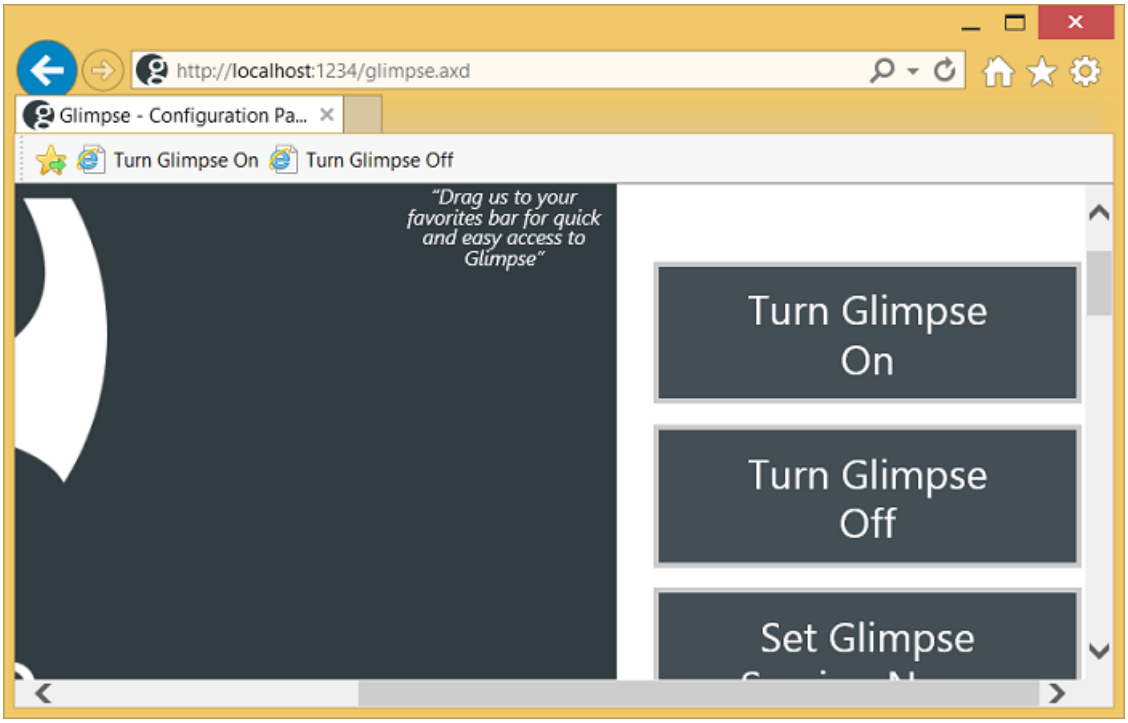
```
PM> Install-Package Glimpse.MVC5
PM> Install-Package Glimpse.EF6
```

Enable Glimpse for localhost

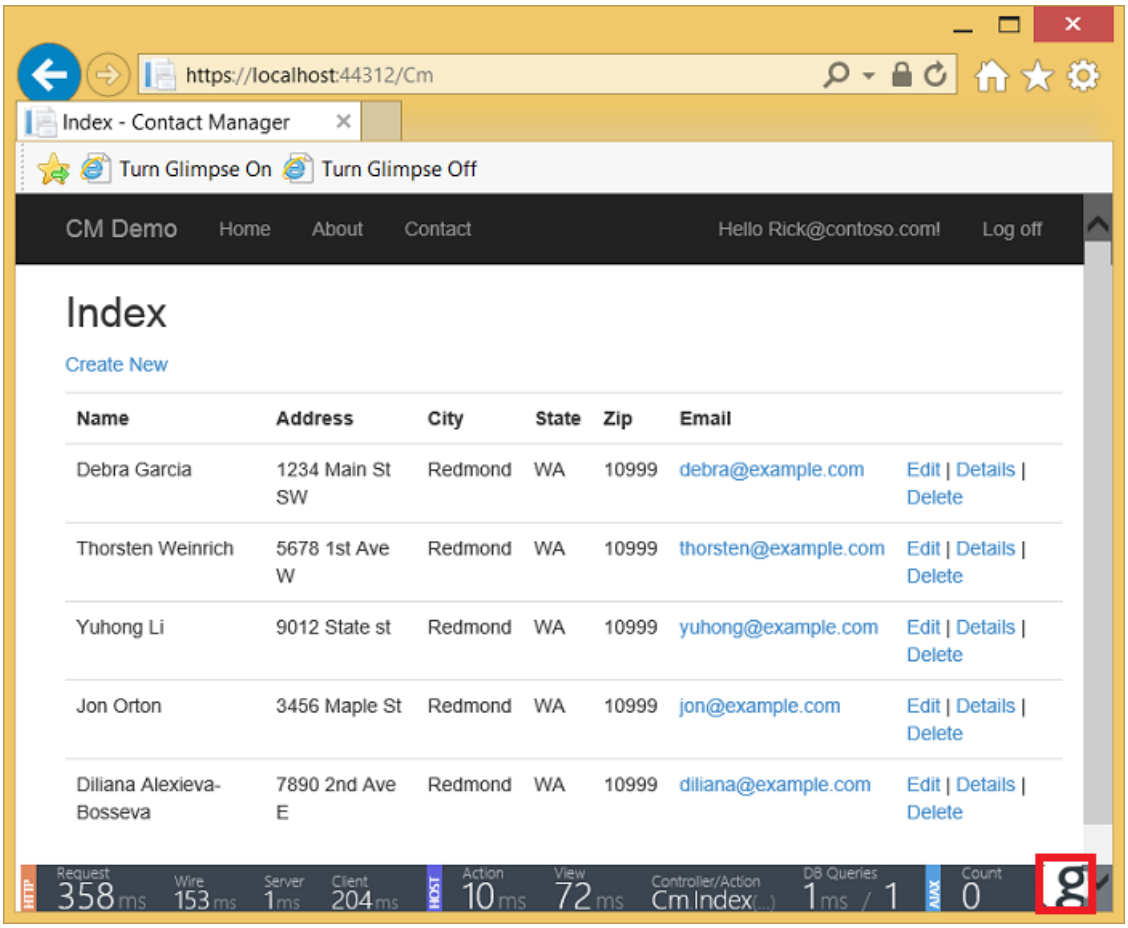
Navigate to http://localhost:<port #>/glimpse.axd and click the **Turn Glimpse On** button.



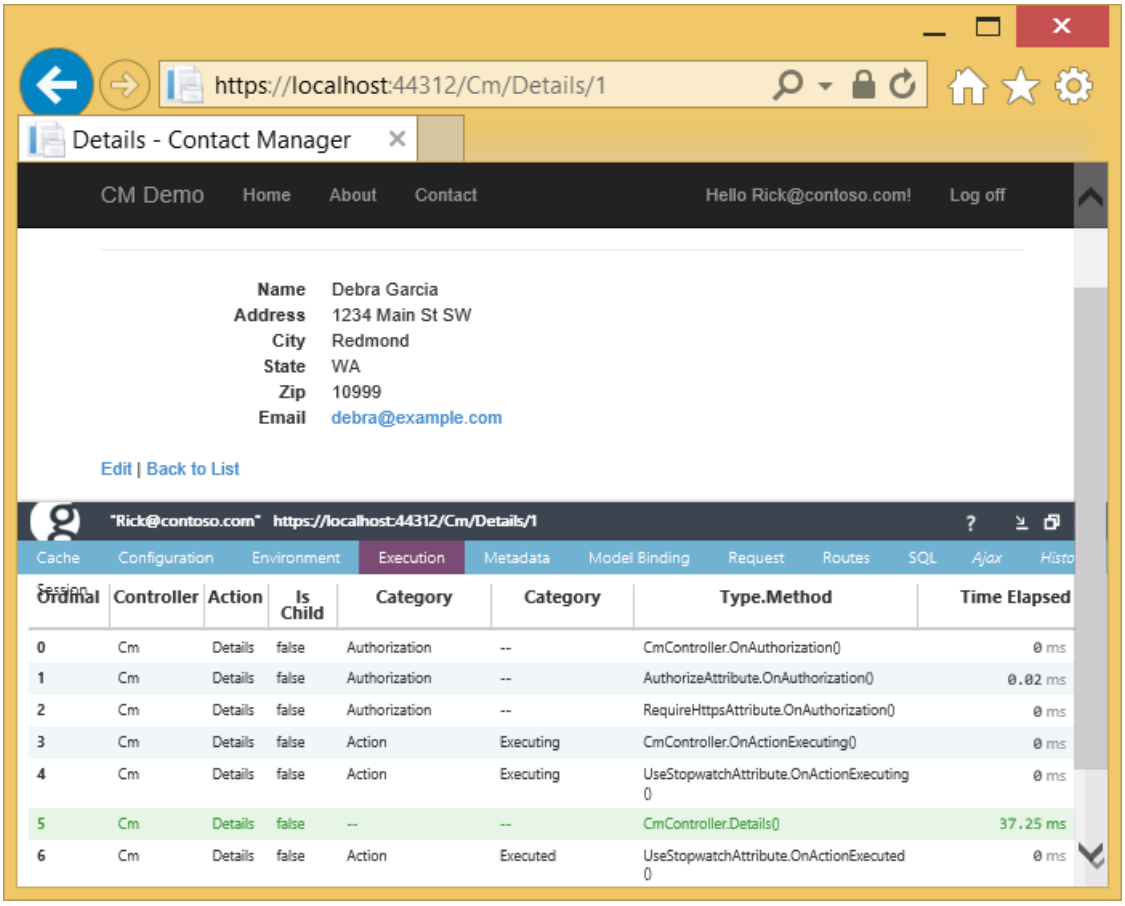
If you have your favorites bar displayed, you can drag and drop the Glimpse buttons and add them as bookmarklets:



You can now navigate your app, and the **Heads Up Display** (HUD) is shown at the bottom of the page.



The Glimpse HUD page details the timing information shown above. The unobtrusive performance data the HUD displays can notify you of a problem immediately - before you get to the test cycle. Clicking on the "g" in the lower right corner brings up the Glimpse panel:



In the image above, the Execution tab is selected, which shows timing details of the actions and filters in the

pipeline. You can see my Stop Watch filter timer start at stage 6 of the pipeline. While my light weight timer can provide useful profile/timing data, it misses all the time spent in authorization and rendering the view. You can read about my timer at Profile and Time your ASP.NET MVC app all the way to Azure. The Tabs page provides links to detailed information on each tab.

The Timeline tab

I've modified Tom Dykstra's outstanding EF 6/MVC 5 tutorial with the following code change to the instructors controller:

```
public ActionResult Index(int? id, int? courseID, int ? eager)
{
    var viewModel = new InstructorIndexData();

    viewModel.Instructors = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.Courses.Select(c => c.Department))
        .OrderBy(i => i.LastName);

    if (id != null)
    {
        ViewBag.InstructorID = id.Value;
        viewModel.Courses = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single().Courses;
    }

    if (courseID != null)
    {
       ViewBag.CourseID = courseID.Value;
       // Eager loading
       if (eager != null && eager > 0)
       {
          ViewBag.eagerMsg = "Eager Loading";

          viewModel.Enrollments = viewModel.Courses.Where(
              x => x.CourseID == courseID).Single().Enrollments;

       }
       else {
        // Explicit loading
          ViewBag.eagerMsg = "Explicit Loading";

        var selectedCourse = viewModel.Courses.Where(x => x.CourseID == courseID).Single();
        db.Entry(selectedCourse).Collection(x => x.Enrollments).Load();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            db.Entry(enrollment).Reference(x => x.Student).Load();
        }

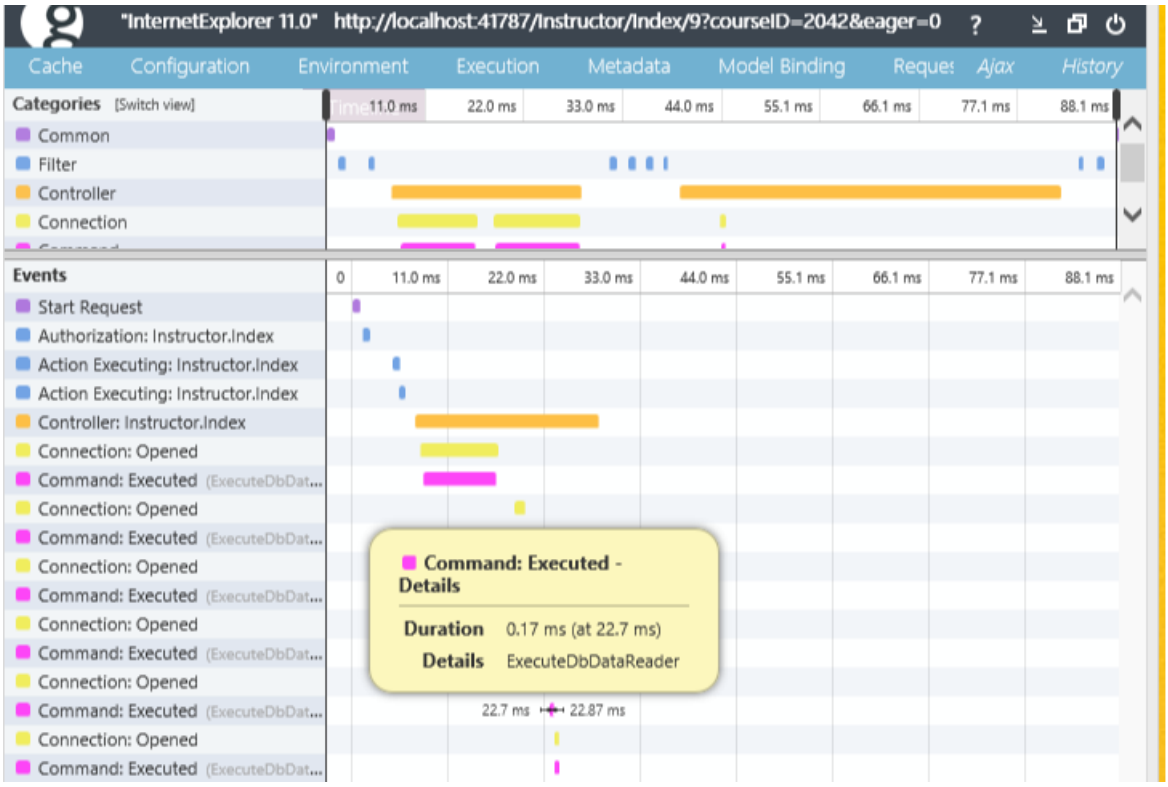
        viewModel.Enrollments = selectedCourse.Enrollments;
       }
    }

    return View(viewModel);
}
```

The code above allows me to pass in query string (`eager`) to control eager or explicit loading of data. In the image below, explicit loading is used and the timing page shows each enrollment loaded in the `Index` action method:

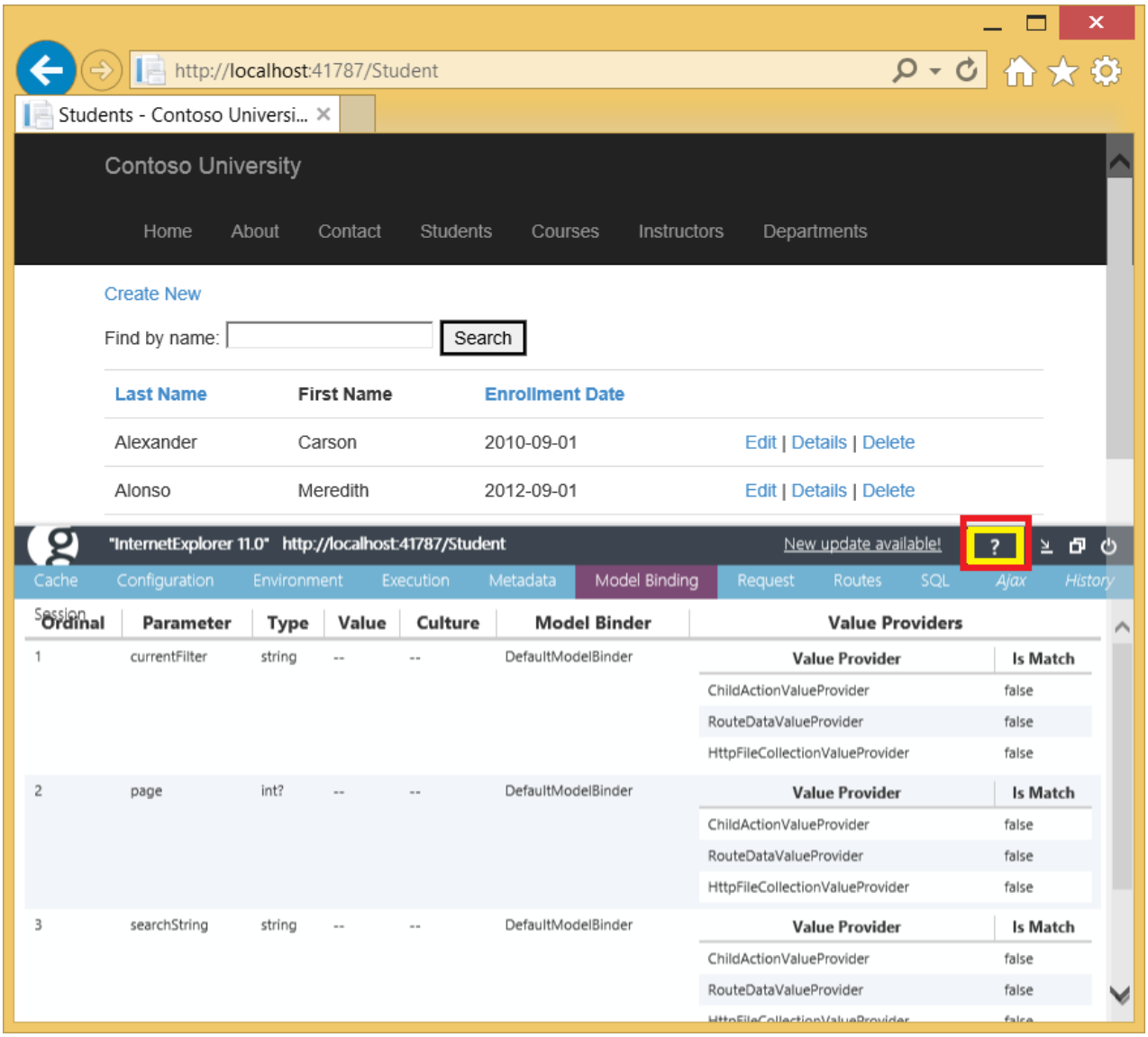
In the following code, eager is specified, and each enrollment is fetched after the `Index` view is called:

You can hover over a time segment to get detailed timing information:



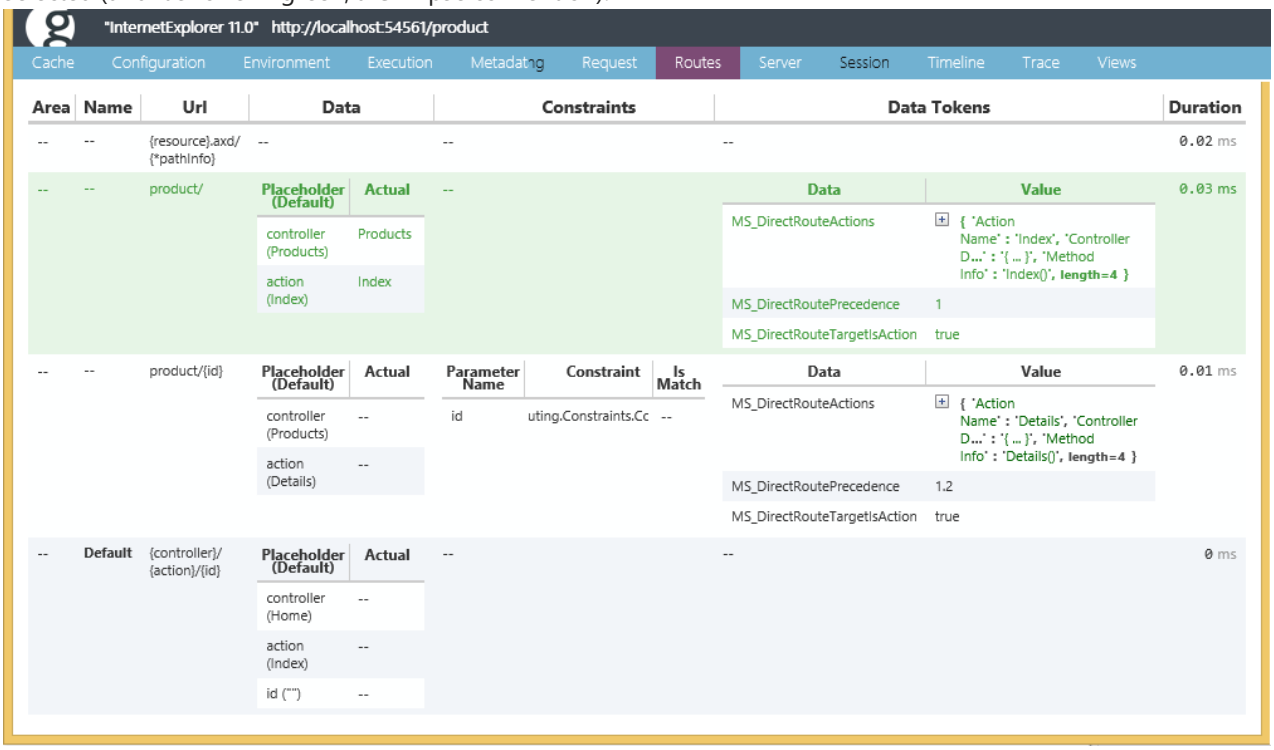
Model Binding

The model binding tab provides a wealth of information to help you understand how your form variables are bound and why some are not bound as would expect. The image below shows the **?** icon, which you can click on to bring up the glimpse help page for that feature.



Routes

The Glimpse Routes tab will can help you debug and understand routing. In the image below, the product route is selected (and it shows in green, a Glimpse convention).



Route constraints, Areas and data tokens are also displayed. See Glimpse Routes and Attribute Routing in ASP.NET MVC 5 for more information.

Using Glimpse on Azure

The Glimpse default security policy only allows Glimpse data to be displayed from local host. You can change this security policy so you can view this data on a remote server (such as a web app on Azure). For test environments on Azure, add the highlighted mark up to the bottom of the *web.config* file to enable Glimpse:

```
<glimpse defaultRuntimePolicy="On" endpointBaseUri="~/Glimpse.axd">
    <runtimePolicies>
      <ignoredTypes>
        <add type="Glimpse.AspNet.Policy.LocalPolicy, Glimpse.AspNet"/>
      </ignoredTypes>
    </runtimePolicies>
  </glimpse>
</configuration>
```

With this change alone, any user can see your Glimpse data on a remote site. Consider adding the markup above to a publish profile so it's only deployed an applied when you use that publish profile (for example, your Azure test profile.) To restrict Glimpse data, we will add the `canViewGlimpseData` role and only allow users in this role to view Glimpse data.

Remove the comments from the *GlimpseSecurityPolicy.cs* file and change the IsInRole call from `Administrator` to the `canViewGlimpseData` role:

```
public class GlimpseSecurityPolicy : IRuntimePolicy
{
    public RuntimePolicy Execute(IRuntimePolicyContext policyContext)
    {
        var httpContext = policyContext.GetHttpContext();
        if (!httpContext.User.IsInRole("canViewGlimpseData"))
        {
            return RuntimePolicy.Off;
        }

        return RuntimePolicy.On;
    }

    public RuntimeEvent ExecuteOn
    {
        get { return RuntimeEvent.EndRequest | RuntimeEvent.ExecuteResource; }
    }
}
```

WARNING

Security - The rich data provided by Glimpse could expose the security of your app. Microsoft has not performed a security audit of Glimpse for use on productions apps.

For information on adding roles, see my Deploy a Secure ASP.NET MVC 5 web app with Membership, OAuth, and SQL Database to Azure tutorial.

Additional Resources

- Deploy a Secure ASP.NET MVC 5 app with Membership, OAuth, and SQL Database to Azure
- Glimpse Configuration - Doc page on configuring tabs, runtime policy, logging and more.

Using Asynchronous Methods in ASP.NET MVC 4

2/20/2020 • 12 minutes to read • Edit Online

by Rick Anderson

> This tutorial will teach you the basics of building an asynchronous ASP.NET MVC Web application using Visual Studio Express 2012 for Web, which is a free version of Microsoft Visual Studio. You can also use Visual Studio 2012.
>
> A complete sample is provided for this tutorial on github https://github.com/RickAndMSFT/Async-ASP.NET/

The ASP.NET MVC 4 Controller class in combination .NET 4.5 enables you to write asynchronous action methods that return an object of type Task<ActionResult>. The .NET Framework 4 introduced an asynchronous programming concept referred to as a Task and ASP.NET MVC 4 supports Task. Tasks are represented by the **Task** type and related types in the System.Threading.Tasks namespace. The .NET Framework 4.5 builds on this asynchronous support with the await and async keywords that make working with Task objects much less complex than previous asynchronous approaches. The await keyword is syntactical shorthand for indicating that a piece of code should asynchronously wait on some other piece of code. The async keyword represents a hint that you can use to mark methods as task-based asynchronous methods. The combination of **await**, **async**, and the **Task** object makes it much easier for you to write asynchronous code in .NET 4.5. The new model for asynchronous methods is called the *Task-based Asynchronous Pattern* (**TAP**). This tutorial assumes you have some familiarity with asynchronous programing using await and async keywords and the Task namespace.

For more information on the using await and async keywords and the Task namespace, see the following references.

- Whitepaper: Asynchrony in .NET
- Async/Await FAQ
- Visual Studio Asynchronous Programming

How Requests Are Processed by the Thread Pool

On the web server, the .NET Framework maintains a pool of threads that are used to service ASP.NET requests. When a request arrives, a thread from the pool is dispatched to process that request. If the request is processed synchronously, the thread that processes the request is busy while the request is being processed, and that thread cannot service another request.

This might not be a problem, because the thread pool can be made large enough to accommodate many busy threads. However, the number of threads in the thread pool is limited (the default maximum for .NET 4.5 is 5,000). In large applications with high concurrency of long-running requests, all available threads might be busy. This condition is known as thread starvation. When this condition is reached, the web server queues requests. If the request queue becomes full, the web server rejects requests with an HTTP 503 status (Server Too Busy). The CLR thread pool has limitations on new thread injections. If concurrency is bursty (that is, your web site can suddenly get a large number of requests) and all available request threads are busy because of backend calls with high latency, the limited thread injection rate can make your application respond very poorly. Additionally, each new thread added to the thread pool has overhead (such as 1 MB of stack memory). A web application using synchronous methods to service high latency calls where the thread pool grows to the .NET 4.5 default maximum of 5, 000 threads would consume approximately 5 GB more memory than an application able the service the same requests using asynchronous methods and only 50 threads. When you're doing asynchronous work, you're not always using a thread. For example, when you make an asynchronous web service request, ASP.NET will not

be using any threads between the **async** method call and the **await**. Using the thread pool to service requests with high latency can lead to a large memory footprint and poor utilization of the server hardware.

Processing Asynchronous Requests

In a web app that sees a large number of concurrent requests at start-up or has a bursty load (where concurrency increases suddenly), making web service calls asynchronous increases the responsiveness of the app. An asynchronous request takes the same amount of time to process as a synchronous request. If a request makes a web service call that requires two seconds to complete, the request takes two seconds whether it's performed synchronously or asynchronously. However during an asynchronous call, a thread isn't blocked from responding to other requests while it waits for the first request to complete. Therefore, asynchronous requests prevent request queuing and thread pool growth when there are many concurrent requests that invoke long-running operations.

Choosing Synchronous or Asynchronous Action Methods

This section lists guidelines for when to use synchronous or asynchronous action methods. These are just guidelines; examine each application individually to determine whether asynchronous methods help with performance.

In general, use synchronous methods for the following conditions:

- The operations are simple or short-running.
- Simplicity is more important than efficiency.
- The operations are primarily CPU operations instead of operations that involve extensive disk or network overhead. Using asynchronous action methods on CPU-bound operations provides no benefits and results in more overhead.

In general, use asynchronous methods for the following conditions:

- You're calling services that can be consumed through asynchronous methods, and you're using .NET 4.5 or higher.
- The operations are network-bound or I/O-bound instead of CPU-bound.
- Parallelism is more important than simplicity of code.
- You want to provide a mechanism that lets users cancel a long-running request.
- When the benefit of switching threads outweighs the cost of the context switch. In general, you should make a method asynchronous if the synchronous method waits on the ASP.NET request thread while doing no work. By making the call asynchronous, the ASP.NET request thread is not stalled doing no work while it waits for the web service request to complete.
- Testing shows that the blocking operations are a bottleneck in site performance and that IIS can service more requests by using asynchronous methods for these blocking calls.

The downloadable sample shows how to use asynchronous action methods effectively. The sample provided was designed to provide a simple demonstration of asynchronous programming in ASP.NET MVC 4 using .NET 4.5. The sample is not intended to be a reference architecture for asynchronous programming in ASP.NET MVC. The sample program calls ASP.NET Web API methods which in turn call Task.Delay to simulate long-running web service calls. Most production applications will not show such obvious benefits to using asynchronous action methods.

Few applications require all action methods to be asynchronous. Often, converting a few synchronous action methods to asynchronous methods provides the best efficiency increase for the amount of work required.

The Sample Application

You can download the sample application from https://github.com/RickAndMSFT/Async-ASP.NET/ on the GitHub site. The repository consists of three projects:

- *Mvc4Async*: The ASP.NET MVC 4 project that contains the code used in this tutorial. It makes Web API calls to the **WebAPIpgw** service.
- *WebAPIpgw*: The ASP.NET MVC 4 Web API project that implements the `Products, Gizmos and Widgets` controllers. It provides the data for the *WebAppAsync* project and the *Mvc4Async* project.
- *WebAppAsync*: The ASP.NET Web Forms project used in another tutorial.

The Gizmos Synchronous Action Method

The following code shows the `Gizmos` synchronous action method that is used to display a list of gizmos. (For this article, a gizmo is a fictional mechanical device.)

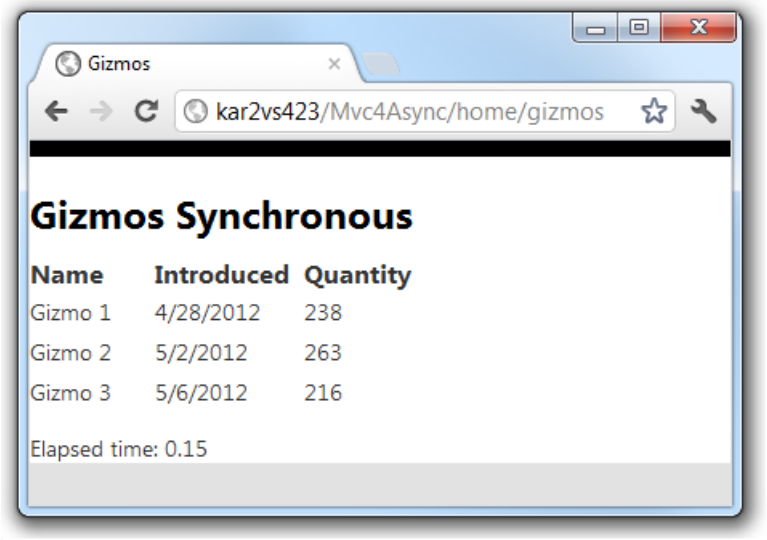
```
public ActionResult Gizmos()
{
    ViewBag.SyncOrAsync = "Synchronous";
    var gizmoService = new GizmoService();
    return View("Gizmos", gizmoService.GetGizmos());
}
```

The following code shows the `GetGizmos` method of the gizmo service.

```
public class GizmoService
{
    public async Task<List<Gizmo>> GetGizmosAsync(
        // Implementation removed.

    public List<Gizmo> GetGizmos()
    {
        var uri = Util.getServiceUri("Gizmos");
        using (WebClient webClient = new WebClient())
        {
            return JsonConvert.DeserializeObject<List<Gizmo>>(
                webClient.DownloadString(uri)
            );
        }
    }
}
```

The `GizmoService GetGizmos` method passes a URI to an ASP.NET Web API HTTP service which returns a list of gizmos data. The *WebAPIpgw* project contains the implementation of the Web API `gizmos, widget` and `product` controllers.
The following image shows the gizmos view from the sample project.



Creating an Asynchronous Gizmos Action Method

The sample uses the new async and await keywords (available in .NET 4.5 and Visual Studio 2012) to let the compiler be responsible for maintaining the complicated transformations necessary for asynchronous programming. The compiler lets you write code using the C#'s synchronous control flow constructs and the compiler automatically applies the transformations necessary to use callbacks in order to avoid blocking threads.

The following code shows the `Gizmos` synchronous method and the `GizmosAsync` asynchronous method. If your browser supports the HTML 5 `<mark>` element, you'll see the changes in `GizmosAsync` in yellow highlight.

```
public ActionResult Gizmos()
{
    ViewBag.SyncOrAsync = "Synchronous";
    var gizmoService = new GizmoService();
    return View("Gizmos", gizmoService.GetGizmos());
}
```

```
public async Task<ActionResult> GizmosAsync()
{
    ViewBag.SyncOrAsync = "Asynchronous";
    var gizmoService = new GizmoService();
    return View("Gizmos", await gizmoService.GetGizmosAsync());
}
```

The following changes were applied to allow the `GizmosAsync` to be asynchronous.

- The method is marked with the async keyword, which tells the compiler to generate callbacks for parts of the body and to automatically create a `Task<ActionResult>` that is returned.
- "Async" was appended to the method name. Appending "Async" is not required but is the convention when writing asynchronous methods.
- The return type was changed from `ActionResult` to `Task<ActionResult>`. The return type of `Task<ActionResult>` represents ongoing work and provides callers of the method with a handle through which to wait for the asynchronous operation's completion. In this case, the caller is the web service. `Task<ActionResult>` represents ongoing work with a result of `ActionResult.`
- The await keyword was applied to the web service call.
- The asynchronous web service API was called (`GetGizmosAsync`).

Inside of the `GetGizmosAsync` method body another asynchronous method, `GetGizmosAsync` is called. `GetGizmosAsync` immediately returns a `Task<List<Gizmo>>` that will eventually complete when the data is available.

Because you don't want to do anything else until you have the gizmo data, the code awaits the task (using the **await** keyword). You can use the **await** keyword only in methods annotated with the **async** keyword.

The **await** keyword does not block the thread until the task is complete. It signs up the rest of the method as a callback on the task, and immediately returns. When the awaited task eventually completes, it will invoke that callback and thus resume the execution of the method right where it left off. For more information on using the await and async keywords and the Task namespace, see the async references.

The following code shows the `GetGizmos` and `GetGizmosAsync` methods.

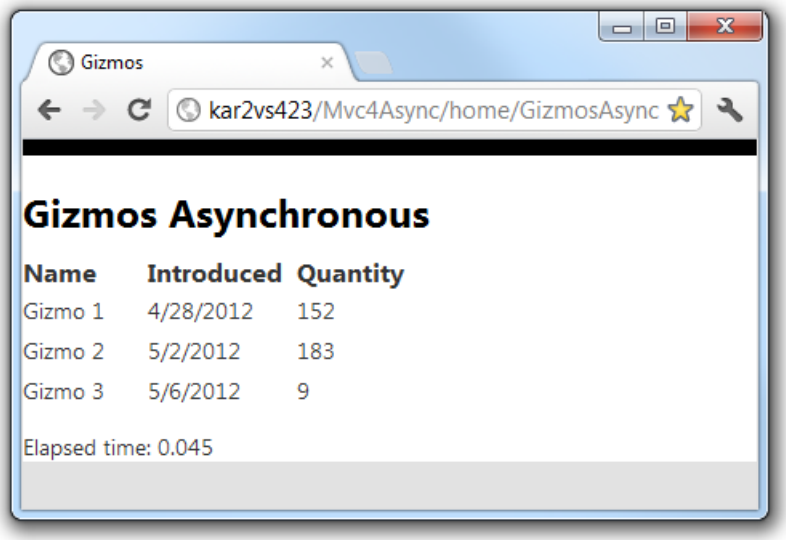
```
public List<Gizmo> GetGizmos()
{
    var uri = Util.getServiceUri("Gizmos");
    using (WebClient webClient = new WebClient())
    {
        return JsonConvert.DeserializeObject<List<Gizmo>>(
            webClient.DownloadString(uri)
        );
    }
}
```

```
public async Task<List<Gizmo>> GetGizmosAsync()
{
    var uri = Util.getServiceUri("Gizmos");
    using (HttpClient httpClient = new HttpClient())
    {
        var response = await httpClient.GetAsync(uri);
        return (await response.Content.ReadAsAsync<List<Gizmo>>());
    }
}
```

The asynchronous changes are similar to those made to the **GizmosAsync** above.

- The method signature was annotated with the async keyword, the return type was changed to `Task<List<Gizmo>>`, and *Async* was appended to the method name.
- The asynchronous HttpClient class is used instead of the WebClient class.
- The await keyword was applied to the HttpClient asynchronous methods.

The following image shows the asynchronous gizmo view.



The browsers presentation of the gizmos data is identical to the view created by the synchronous call. The only difference is the asynchronous version may be more performant under heavy loads.

Performing Multiple Operations in Parallel

Asynchronous action methods have a significant advantage over synchronous methods when an action must perform several independent operations. In the sample provided, the synchronous method `PWG` (for Products, Widgets and Gizmos) displays the results of three web service calls to get a list of products, widgets, and gizmos. The ASP.NET Web API project that provides these services uses Task.Delay to simulate latency or slow network calls. When the delay is set to 500 milliseconds, the asynchronous `PWGasync` method takes a little over 500 milliseconds to complete while the synchronous `PWG` version takes over 1,500 milliseconds. The synchronous `PWG` method is shown in the following code.

```
public ActionResult PWG()
{
    ViewBag.SyncType = "Synchronous";
    var widgetService = new WidgetService();
    var prodService = new ProductService();
    var gizmoService = new GizmoService();

    var pwgVM = new ProdGizWidgetVM(
        widgetService.GetWidgets(),
        prodService.GetProducts(),
        gizmoService.GetGizmos()
       );

    return View("PWG", pwgVM);
}
```

The asynchronous `PWGasync` method is shown in the following code.

```
public async Task<ActionResult> PWGasync()
{
    ViewBag.SyncType = "Asynchronous";
    var widgetService = new WidgetService();
    var prodService = new ProductService();
    var gizmoService = new GizmoService();

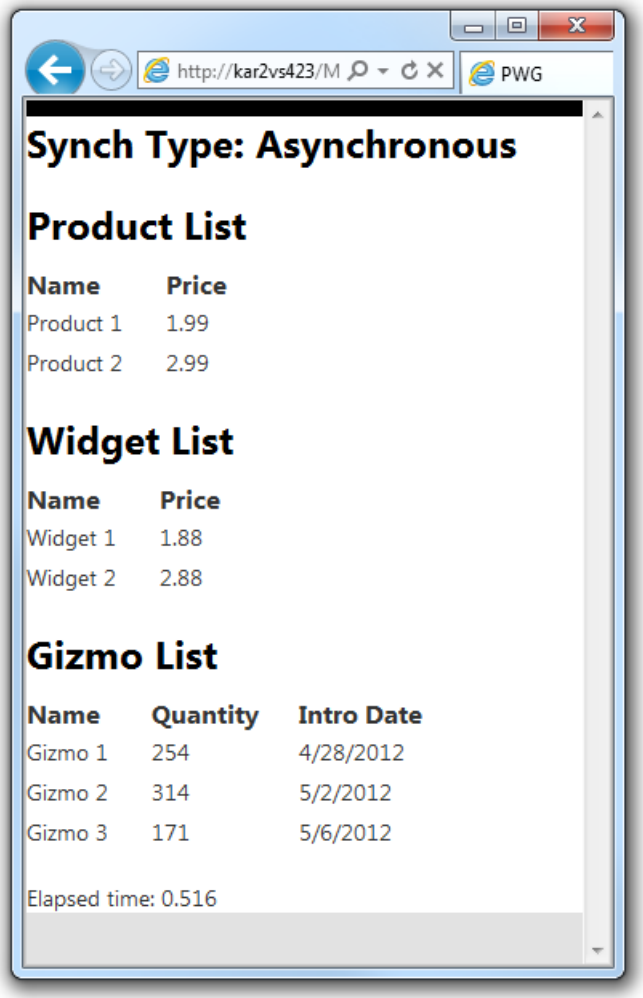
    var widgetTask = widgetService.GetWidgetsAsync();
    var prodTask = prodService.GetProductsAsync();
    var gizmoTask = gizmoService.GetGizmosAsync();

    await Task.WhenAll(widgetTask, prodTask, gizmoTask);

    var pwgVM = new ProdGizWidgetVM(
       widgetTask.Result,
       prodTask.Result,
       gizmoTask.Result
       );

    return View("PWG", pwgVM);
}
```

The following image shows the view returned from the **PWGasync** method.



Using a Cancellation Token

Asynchronous action methods returning `Task<ActionResult>` are cancelable, that is they take a CancellationToken parameter when one is provided with the AsyncTimeout attribute. The following code shows the `GizmosCancelAsync` method with a timeout of 150 milliseconds.

```
[AsyncTimeout(150)]
[HandleError(ExceptionType = typeof(TimeoutException),
                                    View = "TimeoutError")]
public async Task<ActionResult> GizmosCancelAsync(
                        CancellationToken cancellationToken )
{
    ViewBag.SyncOrAsync = "Asynchronous";
    var gizmoService = new GizmoService();
    return View("Gizmos",
        await gizmoService.GetGizmosAsync(cancellationToken));
}
```

The following code shows the GetGizmosAsync overload, which takes a CancellationToken parameter.

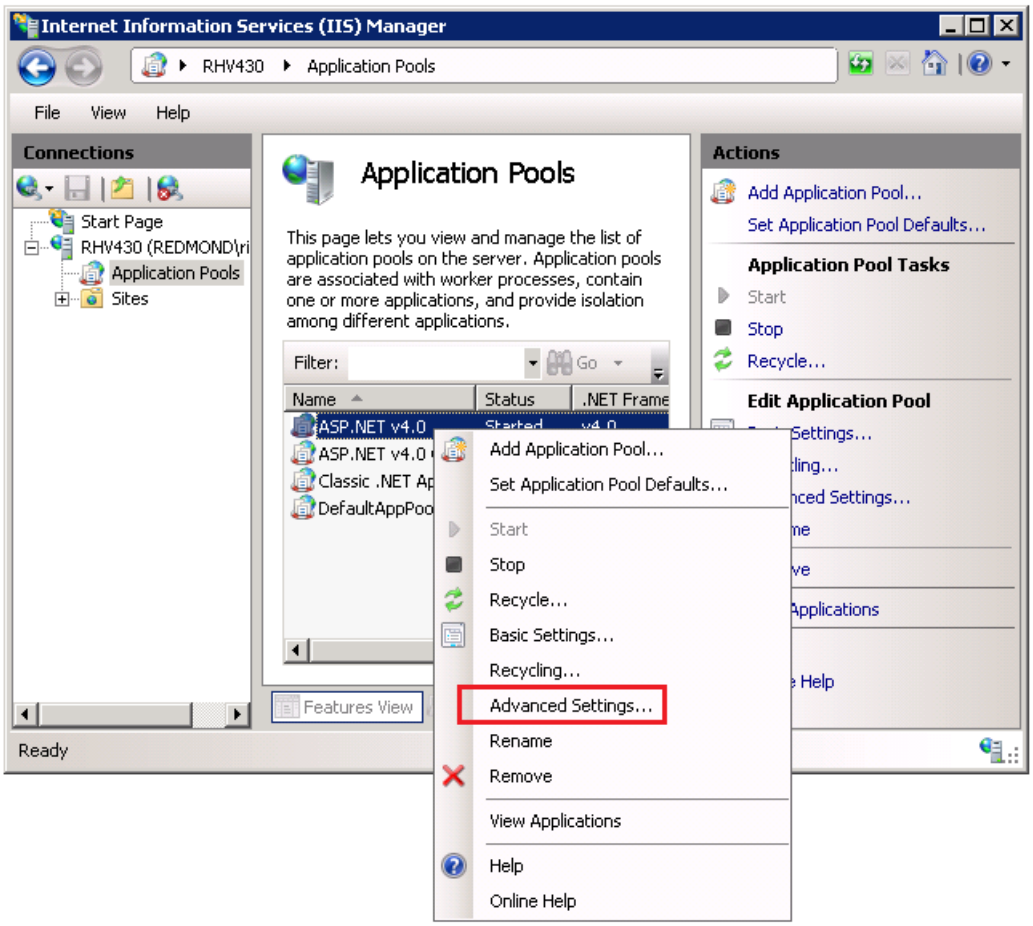
```
public async Task<List<Gizmo>> GetGizmosAsync(string uri,
    CancellationToken cancelToken = default(CancellationToken))
{
    using (HttpClient httpClient = new HttpClient())
    {
        var response = await httpClient.GetAsync(uri, cancelToken);
        return (await response.Content.ReadAsAsync<List<Gizmo>>());
    }
}
```

In the sample application provided, selecting the *Cancellation Token Demo* link calls the `GizmosCancelAsync` method and demonstrates the cancellation of the asynchronous call.

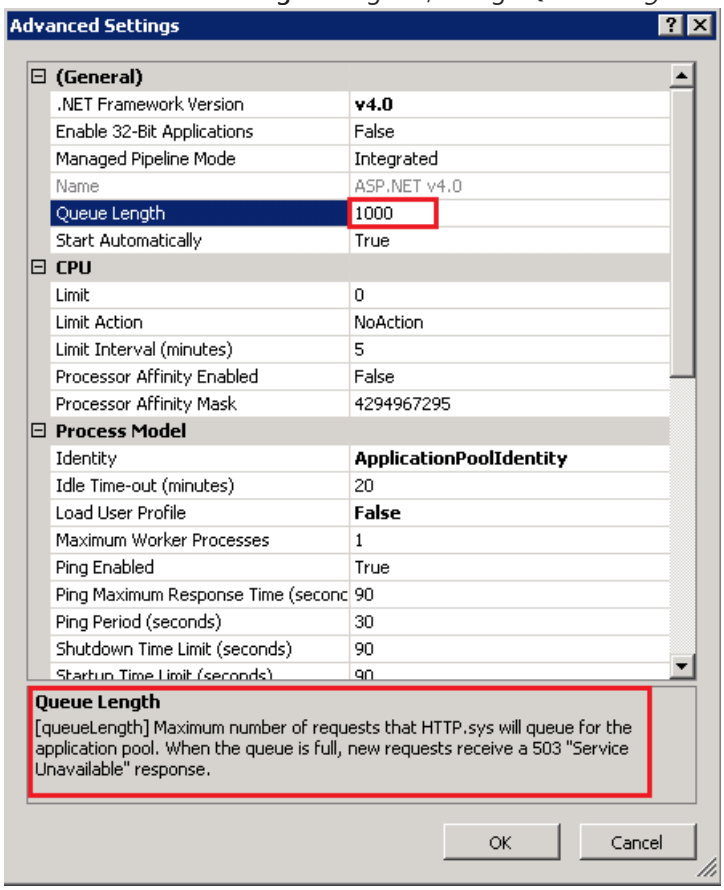
Server Configuration for High Concurrency/High Latency Web Service Calls

To realize the benefits of an asynchronous web application, you might need to make some changes to the default server configuration. Keep the following in mind when configuring and stress testing your asynchronous web application.

- Windows 7, Windows Vista and all Windows client operating systems have a maximum of 10 concurrent requests. You'll need a Windows Server operating system to see the benefits of asynchronous methods under high load.
- Register .NET 4.5 with IIS from an elevated command prompt:
  %windir%\Microsoft.NET\Framework64\v4.0.30319\aspnet_regiis -i
  See ASP.NET IIS Registration Tool (Aspnet_regiis.exe)
- You might need to increase the HTTP.sys queue limit from the default value of 1,000 to 5,000. If the setting is too low, you may see HTTP.sys reject requests with a HTTP 503 status. To change the HTTP.sys queue limit:
  - Open IIS manager and navigate to the Application Pools pane.
  - Right click on the target application pool and select **Advanced Settings**.



- In the **Advanced Settings** dialog box, change *Queue Length* from 1,000 to 5,000.



Note in the images above, the .NET framework is listed as v4.0, even though the application pool is using .NET 4.5. To understand this discrepancy, see the following:

- .NET Versioning and Multi-Targeting - .NET 4.5 is an in-place upgrade to .NET 4.0
- How to set an IIS Application or AppPool to use ASP.NET 3.5 rather than 2.0

  - .NET Framework Versions and Dependencies
- If your application is using web services or System.NET to communicate with a backend over HTTP you may need to increase the connectionManagement/maxconnection element. For ASP.NET applications, this is limited by the autoConfig feature to 12 times the number of CPUs. That means that on a quad-proc, you can have at most 12 * 4 = 48 concurrent connections to an IP end point. Because this is tied to autoConfig, the easiest way to increase `maxconnection` in an ASP.NET application is to set System.Net.ServicePointManager.DefaultConnectionLimit programmatically in the from `Application_Start` method in the *global.asax* file. See the sample download for an example.
- In .NET 4.5, the default of 5000 for MaxConcurrentRequestsPerCPU should be fine.

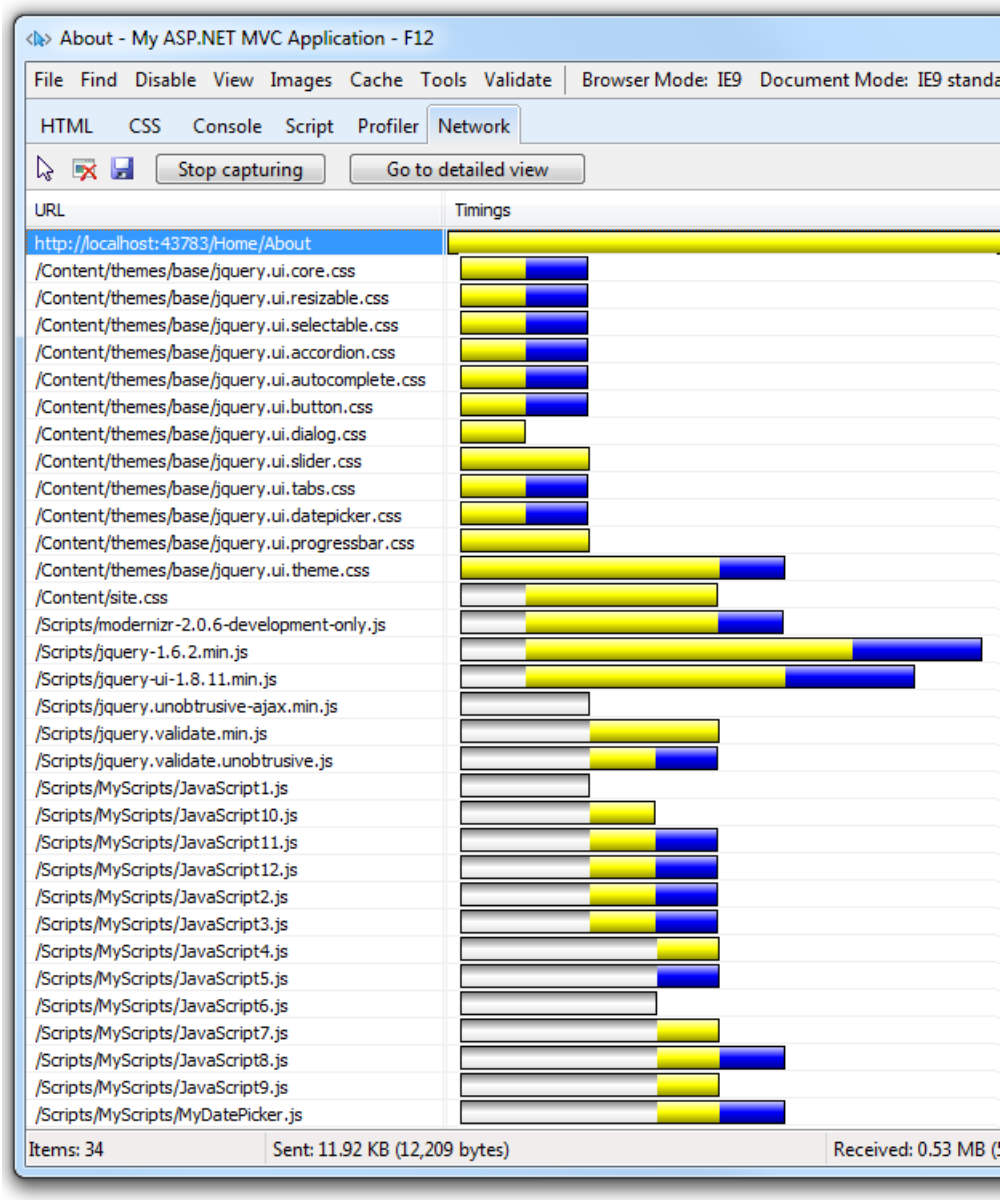
Bundling and Minification

2/20/2020 • 13 minutes to read • Edit Online

by Rick Anderson

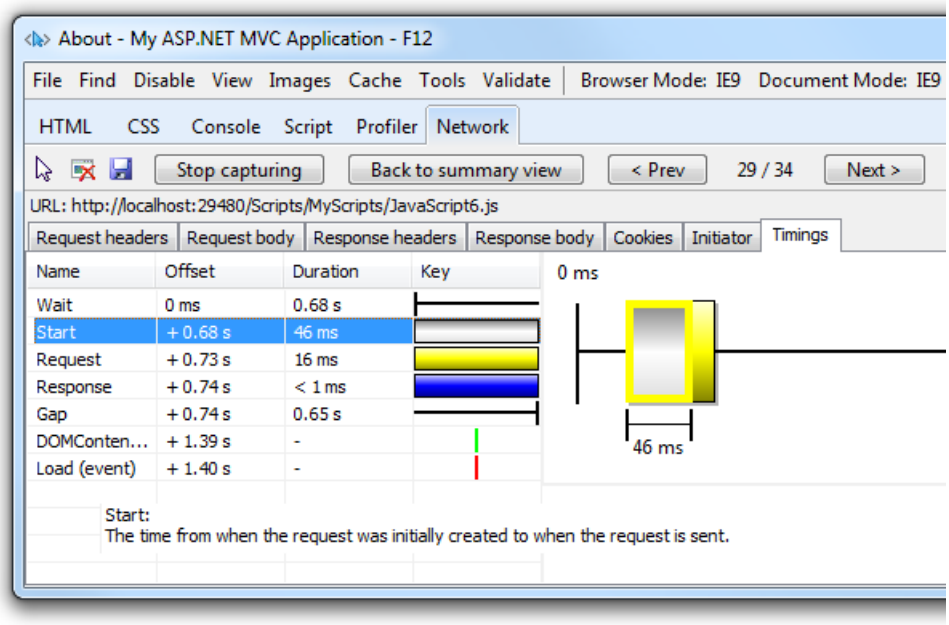
> Bundling and minification are two techniques you can use in ASP.NET 4.5 to improve request load time. Bundling and minification improves load time by reducing the number of requests to the server and reducing the size of requested assets (such as CSS and JavaScript.)

Most of the current major browsers limit the number of simultaneous connections per each hostname to six. That means that while six requests are being processed, additional requests for assets on a host will be queued by the browser. In the image below, the IE F12 developer tools network tabs shows the timing for assets required by the About view of a sample application.



The gray bars show the time the request is queued by the browser waiting on the six connection limit. The yellow bar is the request time to first byte, that is, the time taken to send the request and receive the first response from the server. The blue bars show the time taken to receive the response data from the server. You can double-click on an asset to get detailed timing information. For example, the following image shows the timing details for loading

the */Scripts/MyScripts/JavaScript6.js* file.

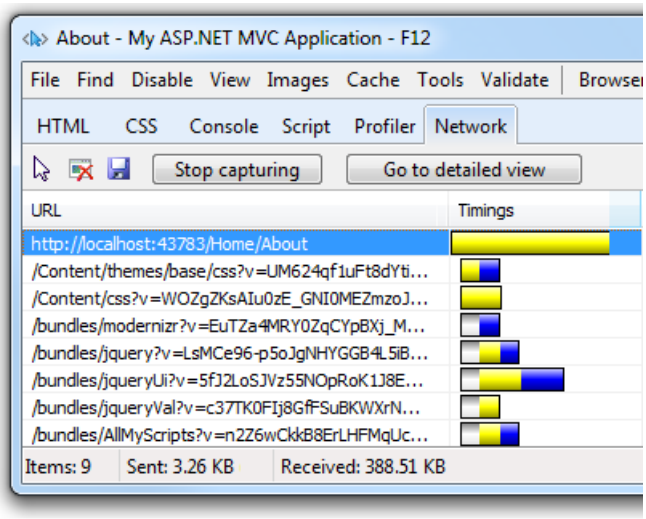


The preceding image shows the **Start** event, which gives the time the request was queued because of the browser limit the number of simultaneous connections. In this case, the request was queued for 46 milliseconds waiting for another request to complete.

Bundling

Bundling is a new feature in ASP.NET 4.5 that makes it easy to combine or bundle multiple files into a single file. You can create CSS, JavaScript and other bundles. Fewer files means fewer HTTP requests and that can improve first page load performance.

The following image shows the same timing view of the About view shown previously, but this time with bundling and minification enabled.



Minification

Minification performs a variety of different code optimizations to scripts or css, such as removing unnecessary white space and comments and shortening variable names to one character. Consider the following JavaScript function.

```
AddAltToImg = function (imageTagAndImageID, imageContext) {
    ///<signature>
    ///<summary> Adds an alt tab to the image
    // </summary>
    //<param name="imgElement" type="String">The image selector.</param>
    //<param name="ContextForImage" type="String">The image context.</param>
    ///</signature>
    var imageElement = $(imageTagAndImageID, imageContext);
    imageElement.attr('alt', imageElement.attr('id').replace(/ID/, ''));
}
```

After minification, the function is reduced to the following:

```
AddAltToImg = function (n, t) { var i = $(n, t); i.attr("alt", i.attr("id").replace(/ID/, "")) }
```

In addition to removing the comments and unnecessary whitespace, the following parameters and variable names were renamed (shortened) as follows:

ORIGINAL	RENAMED
imageTagAndImageID	n
imageContext	t
imageElement	i

Impact of Bundling and Minification

The following table shows several important differences between listing all the assets individually and using bundling and minification (B/M) in the sample program.

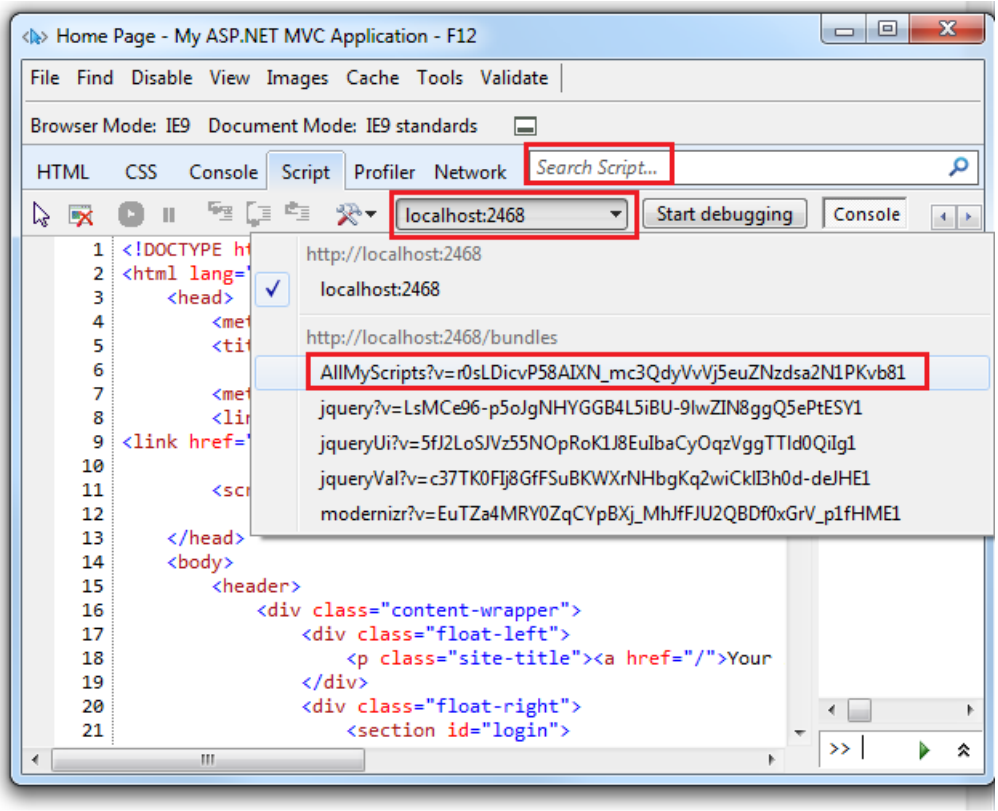
	USING B/M	WITHOUT B/M	CHANGE
File Requests	9	34	256%
KB Sent	3.26	11.92	266%
KB Received	388.51	530	36%
Load Time	510 MS	780 MS	53%

The bytes sent had a significant reduction with bundling as browsers are fairly verbose with the HTTP headers they apply on requests. The received bytes reduction is not as large because the largest files (*Scripts\jquery-ui-1.8.11.min.js* and *Scripts\jquery-1.7.1.min.js*) are already minified. Note: The timings on the sample program used the Fiddler tool to simulate a slow network. (From the Fiddler **Rules** menu, select **Performance** then **Simulate Modem Speeds**.)

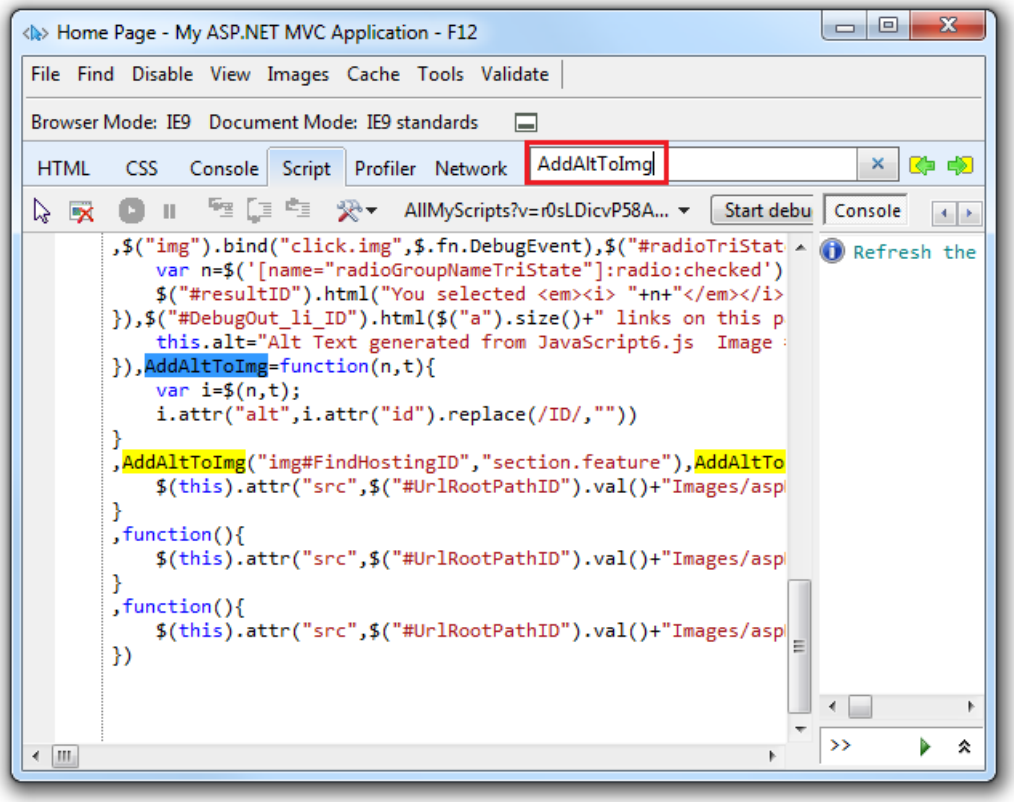
Debugging Bundled and Minified JavaScript

It's easy to debug your JavaScript in a development environment (where the compilation Element in the *Web.config* file is set to `debug="true"`) because the JavaScript files are not bundled or minified. You can also debug a release build where your JavaScript files are bundled and minified. Using the IE F12 developer tools, you debug a JavaScript function included in a minified bundle using the following approach:

1. Select the **Script** tab and then select the **Start debugging** button.
2. Select the bundle containing the JavaScript function you want to debug using the assets button.



3. Format the minified JavaScript by selecting the **Configuration button** , and then selecting **Format JavaScript**.
4. In the **Search Script** input box, select the name of the function you want to debug. In the following image, **AddAltToImg** was entered in the **Search Script** input box.



For more information on debugging with the F12 developer tools, see the MSDN article Using the F12 Developer Tools to Debug JavaScript Errors.

Controlling Bundling and Minification

Bundling and minification is enabled or disabled by setting the value of the debug attribute in the compilation Element in the *Web.config* file. In the following XML, `debug` is set to true so bundling and minification is disabled.

```
<system.web>
    <compilation debug="true" />
    <!-- Lines removed for clarity. -->
</system.web>
```

To enable bundling and minification, set the `debug` value to "false". You can override the *Web.config* setting with the `EnableOptimizations` property on the `BundleTable` class. The following code enables bundling and minification and overrides any setting in the *Web.config* file.

```
public static void RegisterBundles(BundleCollection bundles)
{
    bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
                "~/Scripts/jquery-{version}.js"));

    // Code removed for clarity.
    BundleTable.EnableOptimizations = true;
}
```

NOTE

Unless `EnableOptimizations` is `true` or the debug attribute in the compilation Element in the *Web.config* file is set to `false`, files will not be bundled or minified. Additionally, the .min version of files will not be used, the full debug versions will be selected. `EnableOptimizations` overrides the debug attribute in the compilation Element in the *Web.config* file

Using Bundling and Minification with ASP.NET Web Forms and Web Pages

- For Web Pages, see the blog entry Adding Web Optimization to a Web Pages Site.
- For Web Forms, see the blog entry Adding Bundling and Minification to Web Forms.

Using Bundling and Minification with ASP.NET MVC

In this section we will create an ASP.NET MVC project to examine bundling and minification. First, create a new ASP.NET MVC internet project named **MvcBM** without changing any of the defaults.

Open the *App_Start\BundleConfig.cs* file and examine the `RegisterBundles` method which is used to create, register and configure bundles. The following code shows a portion of the `RegisterBundles` method.

```
public static void RegisterBundles(BundleCollection bundles)
{
     bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
                 "~/Scripts/jquery-{version}.js"));
         // Code removed for clarity.
}
```

The preceding code creates a new JavaScript bundle named *~/bundles/jquery* that includes all the appropriate (that is debug or minified but not *.vsdoc*) files in the *Scripts* folder that match the wild card string "~/Scripts/jquery-{version}.js". For ASP.NET MVC 4, this means with a debug configuration, the file *jquery-1.7.1.js* will be added to the bundle. In a release configuration, *jquery-1.7.1.min.js* will be added. The bundling framework

follows several common conventions such as:

- Selecting ".min" file for release when *FileX.min.js* and *FileX.js* exist.
- Selecting the non ".min" version for debug.
- Ignoring "-vsdoc" files (such as *jquery-1.7.1-vsdoc.js*), which are used only by IntelliSense.

The `{version}` wild card matching shown above is used to automatically create a jQuery bundle with the appropriate version of jQuery in your *Scripts* folder. In this example, using a wild card provides the following benefits:

- Allows you to use NuGet to update to a newer jQuery version without changing the preceding bundling code or jQuery references in your view pages.
- Automatically selects the full version for debug configurations and the ".min" version for release builds.

Using a CDN

The follow code replaces the local jQuery bundle with a CDN jQuery bundle.

```
public static void RegisterBundles(BundleCollection bundles)
{
    //bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
    //            "~/Scripts/jquery-{version}.js"));

    bundles.UseCdn = true;   //enable CDN support

    //add link to jquery on the CDN
    var jqueryCdnPath = "https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.7.1.min.js";

    bundles.Add(new ScriptBundle("~/bundles/jquery",
                jqueryCdnPath).Include(
                "~/Scripts/jquery-{version}.js"));

    // Code removed for clarity.
}
```

In the code above, jQuery will be requested from the CDN while in release mode and the debug version of jQuery will be fetched locally in debug mode. When using a CDN, you should have a fallback mechanism in case the CDN request fails. The following markup fragment from the end of the layout file shows script added to request jQuery should the CDN fail.

```
</footer>

        @Scripts.Render("~/bundles/jquery")

        <script type="text/javascript">
            if (typeof jQuery == 'undefined') {
                var e = document.createElement('script');
                e.src = '@Url.Content("~/Scripts/jquery-1.7.1.js")';
                e.type = 'text/javascript';
                document.getElementsByTagName("head")[0].appendChild(e);

            }
        </script>

        @RenderSection("scripts", required: false)
    </body>
</html>
```

Creating a Bundle

The Bundle class `Include` method takes an array of strings, where each string is a virtual path to resource. The following code from the `RegisterBundles` method in the *App_Start\BundleConfig.cs* file shows how multiple files are added to a bundle:

```
bundles.Add(new StyleBundle("~/Content/themes/base/css").Include(
    "~/Content/themes/base/jquery.ui.core.css",
    "~/Content/themes/base/jquery.ui.resizable.css",
    "~/Content/themes/base/jquery.ui.selectable.css",
    "~/Content/themes/base/jquery.ui.accordion.css",
    "~/Content/themes/base/jquery.ui.autocomplete.css",
    "~/Content/themes/base/jquery.ui.button.css",
    "~/Content/themes/base/jquery.ui.dialog.css",
    "~/Content/themes/base/jquery.ui.slider.css",
    "~/Content/themes/base/jquery.ui.tabs.css",
    "~/Content/themes/base/jquery.ui.datepicker.css",
    "~/Content/themes/base/jquery.ui.progressbar.css",
    "~/Content/themes/base/jquery.ui.theme.css"));
```

The Bundle class `IncludeDirectory` method is provided to add all the files in a directory (and optionally all subdirectories) which match a search pattern. The Bundle class `IncludeDirectory` API is shown below:

```
public Bundle IncludeDirectory(
    string directoryVirtualPath,  // The Virtual Path for the directory.
    string searchPattern)         // The search pattern.

public Bundle IncludeDirectory(
    string directoryVirtualPath,  // The Virtual Path for the directory.
    string searchPattern,         // The search pattern.
    bool searchSubdirectories)    // true to search subdirectories.
```

Bundles are referenced in views using the Render method, (`Styles.Render` for CSS and `Scripts.Render` for JavaScript). The following markup from the *Views\Shared_Layout.cshtml* file shows how the default ASP.NET internet project views reference CSS and JavaScript bundles.

```
<!DOCTYPE html>
<html lang="en">
<head>
    @* Markup removed for clarity.*@
    @Styles.Render("~/Content/themes/base/css", "~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>
    @* Markup removed for clarity.*@

   @Scripts.Render("~/bundles/jquery")
   @RenderSection("scripts", required: false)
</body>
</html>
```

Notice the Render methods takes an array of strings, so you can add multiple bundles in one line of code. You will generally want to use the Render methods which create the necessary HTML to reference the asset. You can use the `Url` method to generate the URL to the asset without the markup needed to reference the asset. Suppose you wanted to use the new HTML5 async attribute. The following code shows how to reference modernizr using the `Url` method.

```
<head>
    @*Markup removed for clarity*@
    <meta charset="utf-8" />
    <title>@ViewBag.Title - MVC 4 B/M</title>
    <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />
    <meta name="viewport" content="width=device-width" />
    @Styles.Render("~/Content/css")

   @* @Scripts.Render("~/bundles/modernizr")*@

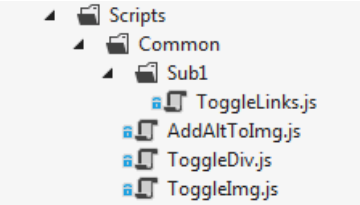
    <script src='@Scripts.Url("~/bundles/modernizr")' async> </script>
</head>
```

Using the "*" Wildcard Character to Select Files

The virtual path specified in the `Include` method and the search pattern in the `IncludeDirectory` method can accept one "*" wildcard character as a prefix or suffix to in the last path segment. The search string is case insensitive. The `IncludeDirectory` method has the option of searching subdirectories.

Consider a project with the following JavaScript files:

- *Scripts\Common\AddAltToImg.js*
- *Scripts\Common\ToggleDiv.js*
- *Scripts\Common\ToggleImg.js*
- *Scripts\Common\Sub1\ToggleLinks.js*



The following table shows the files added to a bundle using the wildcard as shown:

CALL	FILES ADDED OR EXCEPTION RAISED
Include("~/Scripts/Common/*.js")	*AddAltToImg.js*, *ToggleDiv.js*, *ToggleImg.js*
Include("~/Scripts/Common/T*.js")	Invalid pattern exception. The wildcard character is only allowed on the prefix or suffix.
Include("~/Scripts/Common/*og.*")	Invalid pattern exception. Only one wildcard character is allowed.
Include("~/Scripts/Common/T*")	*ToggleDiv.js*, *ToggleImg.js*
Include("~/Scripts/Common/*")	Invalid pattern exception. A pure wildcard segment is not valid.
IncludeDirectory("~/Scripts/Common", "T*")	*ToggleDiv.js*, *ToggleImg.js*
IncludeDirectory("~/Scripts/Common", "T*", true)	*ToggleDiv.js*, *ToggleImg.js*, *ToggleLinks.js*

Explicitly adding each file to a bundle is generally the preferred over wildcard loading of files for the following reasons:

- Adding scripts by wildcard defaults to loading them in alphabetical order, which is typically not what you want. CSS and JavaScript files frequently need to be added in a specific (non-alphabetic) order. You can mitigate this risk by adding a custom IBundleOrderer implementation, but explicitly adding each file is less error prone. For example, you might add new assets to a folder in the future which might require you to modify your IBundleOrderer implementation.

- View specific files added to a directory using wild card loading can be included in all views referencing that bundle. If the view specific script is added to a bundle, you may get a JavaScript error on other views that reference the bundle.

- CSS files that import other files result in the imported files loaded twice. For example, the following code creates a bundle with most of the jQuery UI theme CSS files loaded twice.

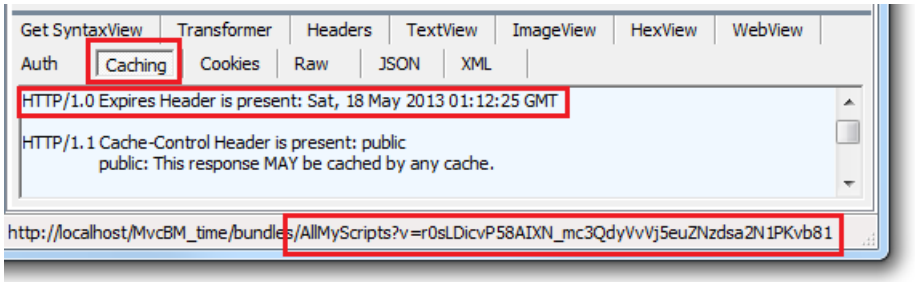
  ```
  bundles.Add(new StyleBundle("~/jQueryUI/themes/baseAll")
      .IncludeDirectory("~/Content/themes/base", "*.css"));
  ```

  The wild card selector "*.css" brings in each CSS file in the folder, including the *Content\themes\base\jquery.ui.all.css* file. The *jquery.ui.all.css* file imports other CSS files.

Bundle Caching

Bundles set the HTTP Expires Header one year from when the bundle is created. If you navigate to a previously viewed page, Fiddler shows IE does not make a conditional request for the bundle, that is, there are no HTTP GET requests from IE for the bundles and no HTTP 304 responses from the server. You can force IE to make a conditional request for each bundle with the F5 key (resulting in a HTTP 304 response for each bundle). You can force a full refresh by using ^F5 (resulting in a HTTP 200 response for each bundle.)

The following image shows the **Caching** tab of the Fiddler response pane:



The request

`http://localhost/MvcBM_time/bundles/AllMyScripts?v=r0sLDicvP58AIXN_mc3QdyVvVj5euZNzdsa2N1PKvb81`

is for the bundle **AllMyScripts** and contains a query string pair **v=r0sLDicvP58AIXN_mc3QdyVvVj5euZNzdsa2N1PKvb81**. The query string **v** has a value token that is a unique identifier used for caching. As long as the bundle doesn't change, the ASP.NET application will request the **AllMyScripts** bundle using this token. If any file in the bundle changes, the ASP.NET optimization framework will generate a new token, guaranteeing that browser requests for the bundle will get the latest bundle.

If you run the IE9 F12 developer tools and navigate to a previously loaded page, IE incorrectly shows conditional GET requests made to each bundle and the server returning HTTP 304. You can read why IE9 has problems determining if a conditional request was made in the blog entry Using CDNs and Expires to Improve Web Site Performance.

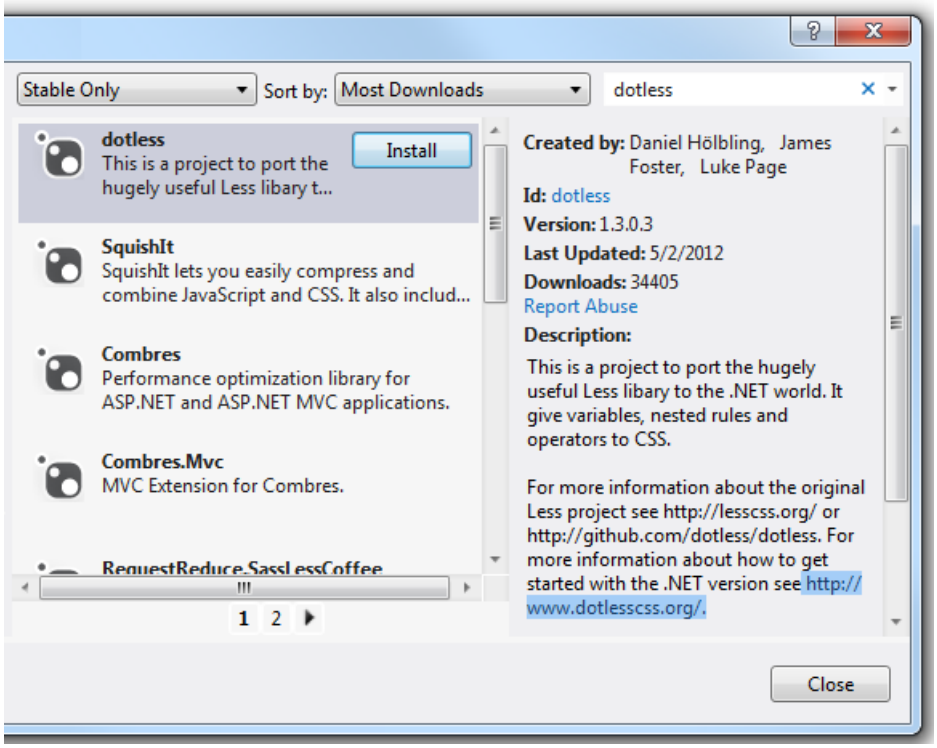
LESS, CoffeeScript, SCSS, Sass Bundling.

The bundling and minification framework provides a mechanism to process intermediate languages such as SCSS, Sass, LESS or Coffeescript, and apply transforms such as minification to the resulting bundle. For example, to add

.less files to your MVC 4 project:

1. Create a folder for your LESS content. The following example uses the *Content\MyLess* folder.

2. Add the .less NuGet package **dotless** to your project.



3. Add a class that implements the IBundleTransform interface. For the .less transform, add the following code to your project.

```
using System.Web.Optimization;

public class LessTransform : IBundleTransform
{
    public void Process(BundleContext context, BundleResponse response)
    {
        response.Content = dotless.Core.Less.Parse(response.Content);
        response.ContentType = "text/css";
    }
}
```

4. Create a bundle of LESS files with the `LessTransform` and the CssMinify transform. Add the following code to the `RegisterBundles` method in the *App_Start\BundleConfig.cs* file.

```
var lessBundle = new Bundle("~/My/Less").IncludeDirectory("~/My", "*.less");
lessBundle.Transforms.Add(new LessTransform());
lessBundle.Transforms.Add(new CssMinify());
bundles.Add(lessBundle);
```

5. Add the following code to any views which references the LESS bundle.

```
@Styles.Render("~/My/Less");
```

Bundle Considerations

A good convention to follow when creating bundles is to include "bundle" as a prefix in the bundle name. This will prevent a possible routing conflict.

Once you update one file in a bundle, a new token is generated for the bundle query string parameter and the full bundle must be downloaded the next time a client requests a page containing the bundle. In traditional markup where each asset is listed individually, only the changed file would be downloaded. Assets that change frequently may not be good candidates for bundling.

Bundling and minification primarily improve the first page request load time. Once a webpage has been requested, the browser caches the assets (JavaScript, CSS and images) so bundling and minification won't provide any performance boost when requesting the same page, or pages on the same site requesting the same assets. If you don't set the expires header correctly on your assets, and you don't use bundling and minification, the browsers freshness heuristics will mark the assets stale after a few days and the browser will require a validation request for each asset. In this case, bundling and minification provide a performance increase after the first page request. For details, see the blog Using CDNs and Expires to Improve Web Site Performance.

The browser limitation of six simultaneous connections per each hostname can be mitigated by using a CDN. Because the CDN will have a different hostname than your hosting site, asset requests from the CDN will not count against the six simultaneous connections limit to your hosting environment. A CDN can also provide common package caching and edge caching advantages.

Bundles should be partitioned by pages that need them. For example, the default ASP.NET MVC template for an internet application creates a jQuery Validation bundle separate from jQuery. Because the default views created have no input and do not post values, they don't include the validation bundle.

The `System.Web.Optimization` namespace is implemented in *System.Web.Optimization.dll*. It leverages the WebGrease library (*WebGrease.dll*) for minification capabilities, which in turn uses *Antlr3.Runtime.dll*.

I use Twitter to make quick posts and share links. My Twitter handle is: @RickAndMSFT

Additional resources

- Video:Bundling and Optimizing by Howard Dierking
- Adding Web Optimization to a Web Pages Site.
- Adding Bundling and Minification to Web Forms.
- Performance Implications of Bundling and Minification on Web Browsing by Henrik F Nielsen @frystyk
- Using CDNs and Expires to Improve Web Site Performance by Rick Anderson @RickAndMSFT
- Minimize RTT (round-trip times)

Contributors

- Hao Kung
- Howard Dierking
- Diana LaRose

ASP.NET MVC Advanced Topics

2/20/2020 • 2 minutes to read • Edit Online

Advanced topics for programming ASP.NET MVC, including accessibility, extensibility, and internationalization.

- Custom MVC Template

Custom MVC Template

2/20/2020 • 6 minutes to read • Edit Online

by Jacques Eloff

The release of MVC 3 Tools Update for Visual Studio 2010 introduced a separate project wizard for MVC projects. The change was driven by two factors. First, the introduction of new templates in MVC 3 and support for additional view engines such as Razor lead to overcrowding the New Project dialog in Visual Studio. Second, customers had been asking for extensibility points and the new MVC project wizard would afford us the opportunity to respond to these requests.

Adding custom templates was an arduous process that relied on using the registry to make new templates visible to the MVC project wizard. The author of a new template had to wrap it inside an MSI to ensure that the necessary registry entries would be created at install time. The alternative was to make a ZIP file containing the template available and have the end-user create the required registry entries by hand.

Neither of the aforementioned approaches is ideal so we decided to leverage some of the existing infrastructure provided by VSIX extensions to make it easier to author, distribute and install custom MVC templates starting with MVC 4 for Visual Studio 2012. Some of the benefits provided by this approach are:

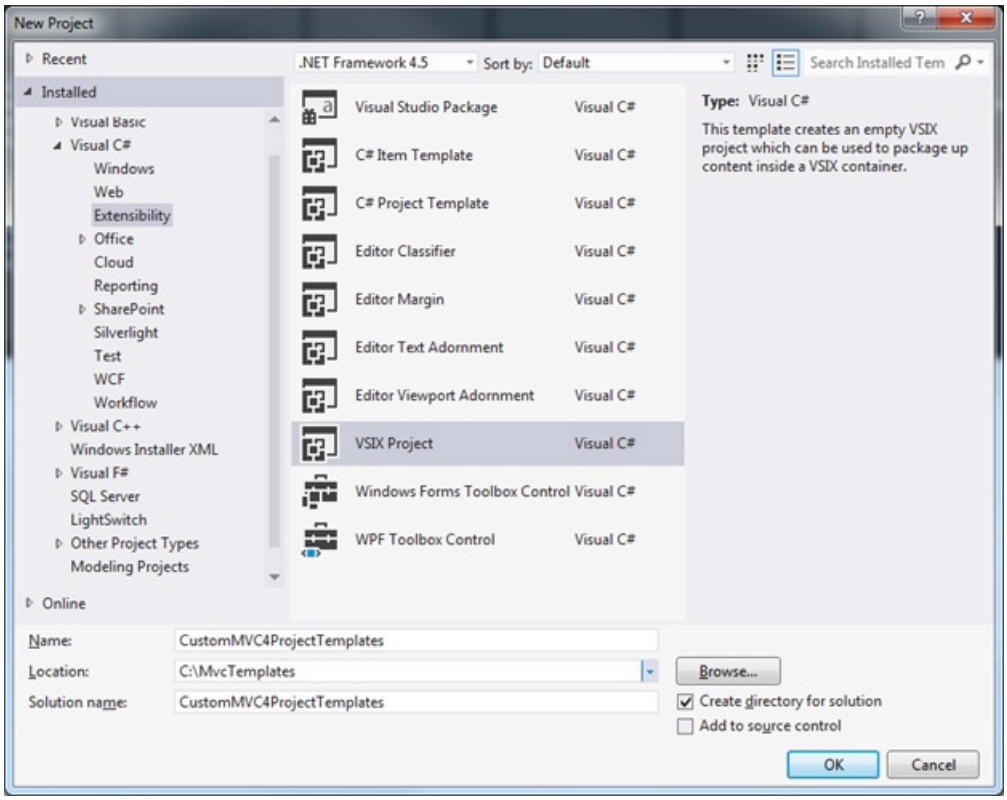
- A VSIX extension can contain multiple templates that support different languages (C# and Visual Basic) and multiple view engines (ASPX and Razor).
- A VSIX extension can target multiple SKUs of Visual Studio including Express SKUs.
- The Visual Studio Gallery facilitates distributing the extension to a wide audience.
- VSIX extensions can be upgraded making it easier to author corrections and updates to your custom templates.

Prerequisites

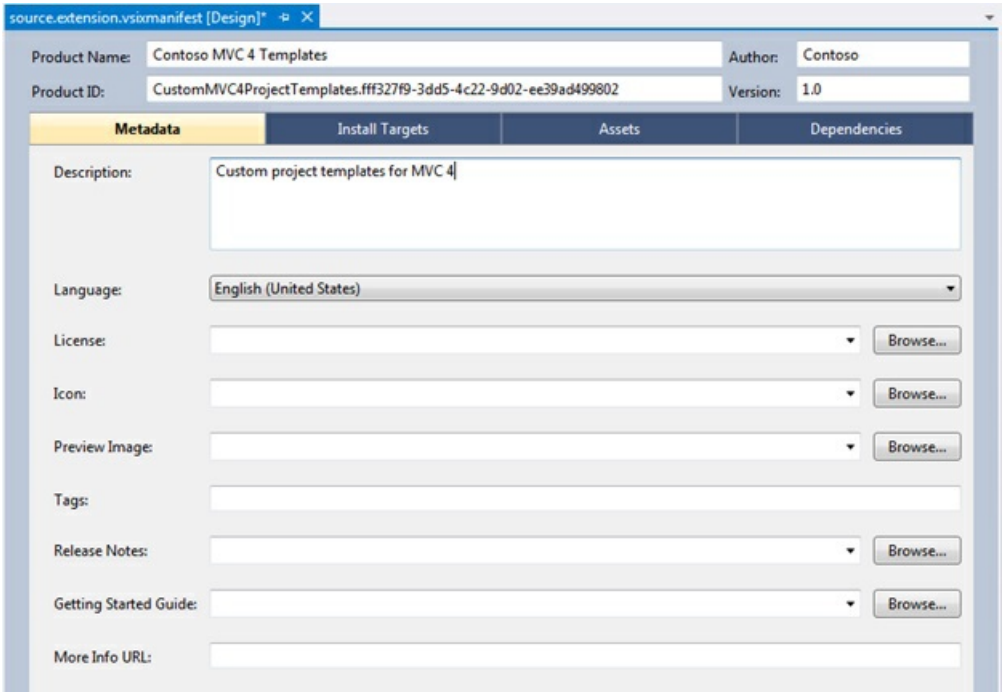
- Users need to be familiar with authoring project templates, including the required markup for vstemplate files, etc.
- Users will need to have Visual Studio Professional and higher installed. Express SKUs do not support creating VSIX projects.
- Visual Studio 2012 SDK installed.

Example

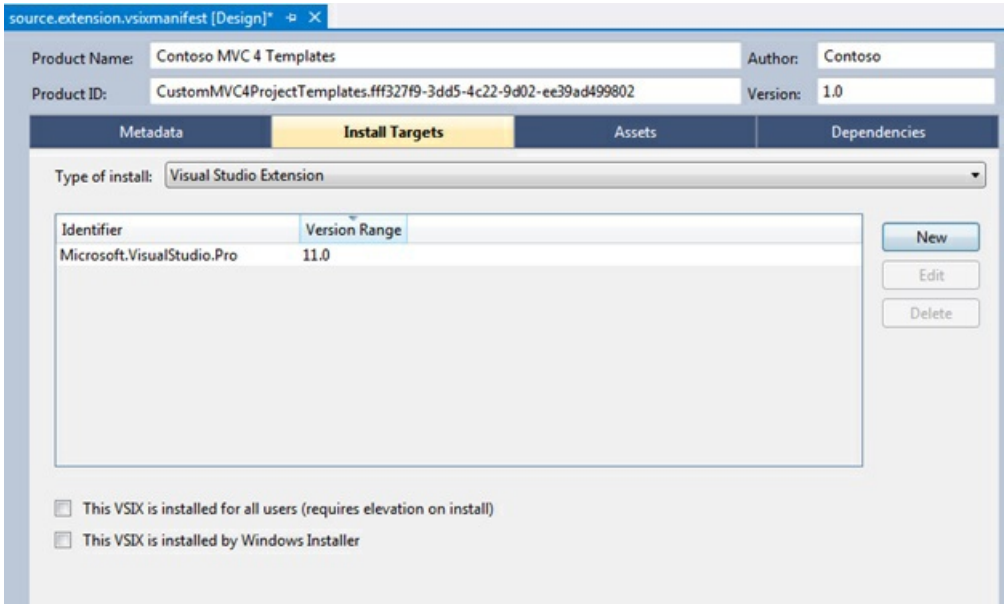
The first step is to create a new VSIX project using either C# or Visual Basic. Select **File > New Project**, then click **Extensibility** in the left pane and select the **VSIX Project**.



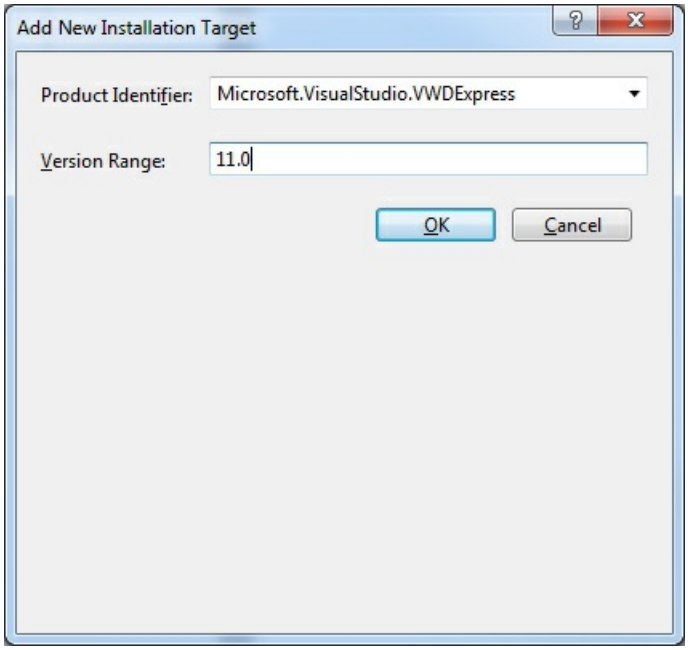
After the project is created, the VSIX designer will be opened.



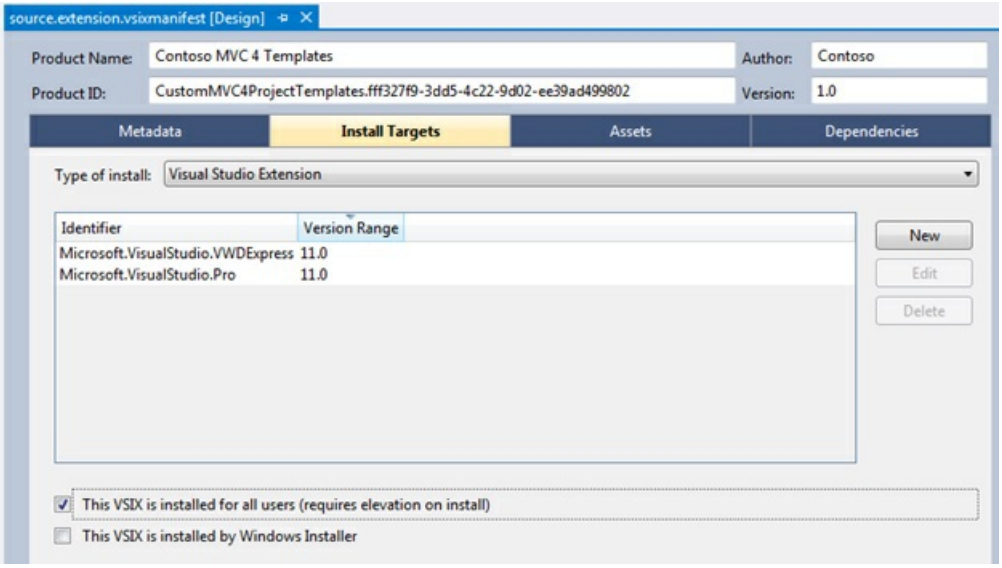
The designer can be used to edit some of the general properties of the extension that will be shown to users when they install the extension or browse the installed extensions in Visual Studio (**Tools > Extensions and Updates**). Once you have completed the general information click on the **Install Targets tab**.



This tab is used to specify the SKUs and versions of Visual Studio that are supported by your extension. Select the checkbox for **This VSIX is installed for all users** to enable per-machine installs of the VSIX. Click on the **New** button on the right to add additional SKUs such as Web Developer Express (VWD).



If you intend to support all the Professional and higher SKUs (Professional, Premium and Ultimate) you only need to select the minimum SKU in the family, **Microsoft.VisualStudio.Pro**. Remember to save all your changes once you have completed the Install Targets.



The **Assets** tab is used to add all your content files to the VSIX. Since MVC requires custom metadata you will be editing the raw XML of the VSIX manifest file instead of using the **Assets** tab to add content. Start by adding the template contents to the VSIX project. It is important that the structure of the folder and the contents mirror the layout of the project. The example below contains four project templates that were derived from the Basic MVC project template. Make sure that all the files that comprise your project template (everything underneath the ProjectTemplates folder) are added to the **Content** itemgroup in the VSIX project file and that each item contains the **CopyToOutputDirectory** and **IncludeInVsix** metadata set as shown in the example below.

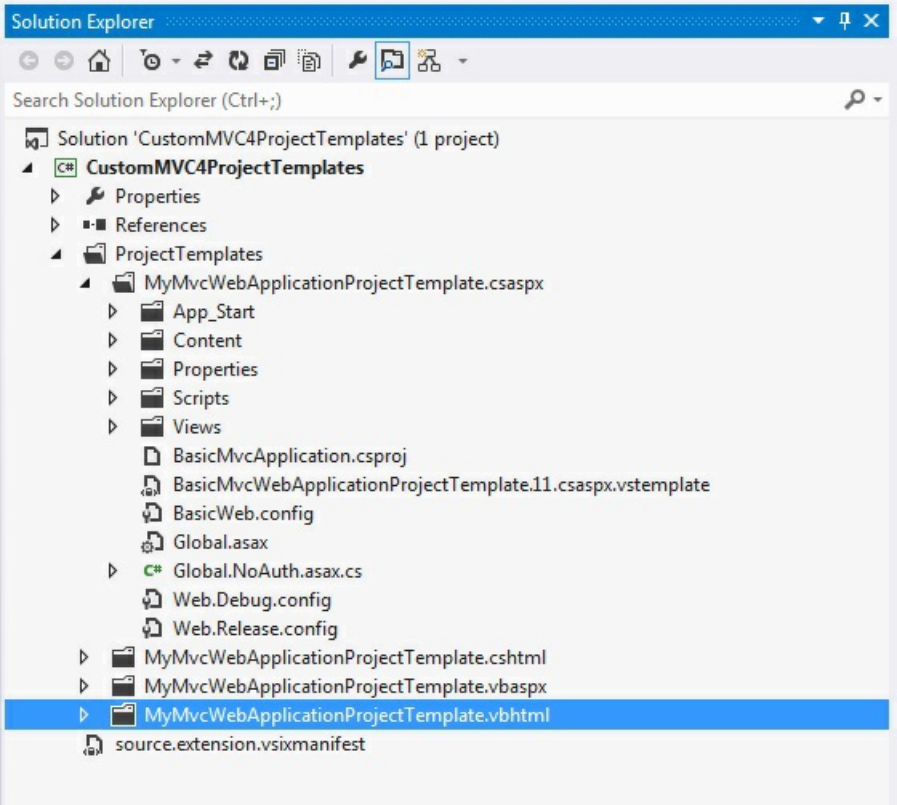
```
<Content Include="ProjectTemplates\MyMvcWebApplicationProjectTemplate.csaspx\BasicWeb.config">

<CopyToOutputDirectory>Always</CopyToOutputDirectory>

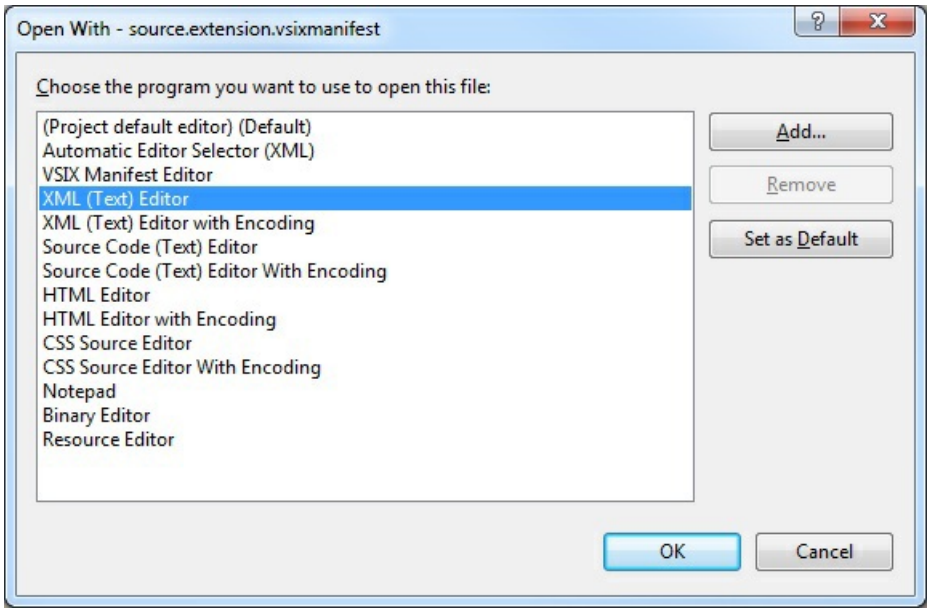
<IncludeInVSIX>true</IncludeInVSIX>

</Content>
```

If not, the IDE will try to compile the contents of the template when you build the VSIX and you will likely see an error. Code files in templates often contain special template parameters used by Visual Studio when the project template is instantiated and therefore cannot be compiled in the IDE.



Close the VSIX designer, then right click on the **source.extension.manifest** file in **Solution Explorer** and select **Open With** and choose the **XML (Text) Editor** option.



Create an **<Assets>** element and add an **<Asset>** element for each file that must be included in the VSIX. The **Type** attribute of each **<Asset>** element must be set to **Microsoft.VisualStudio.Mvc.Template**. This is a custom namespace that only the MVC project wizard understands. Refer to the VSIX 2.0 Schema documentation for additional information on the structure and layout of the manifest file.

Just adding the files to the VSIX is not sufficient to register the templates with the MVC wizard. You need to provide information such as the template name, description, supported view engines and programming language to the MVC wizard. This information is carried in custom attributes associated with the **<Asset>** element for each **vstemplate** file.

<Asset d:VsixSubPath="ProjectTemplates\MyMvcWebApplicationProjectTemplate.csaspx"

Type="Microsoft.VisualStudio.Mvc.Template"

d:Source="File"

Path="ProjectTemplates\MyMvcWebApplicationProjectTemplate.csaspx\BasicMvcWebApplicationProjectTemplate.11.csaspx.vstemplate"

ProjectType="MVC"

Language="C#"

ViewEngine="Aspx"

TemplateId="MyMvcApplication"

Title="Custom Basic Web Application"

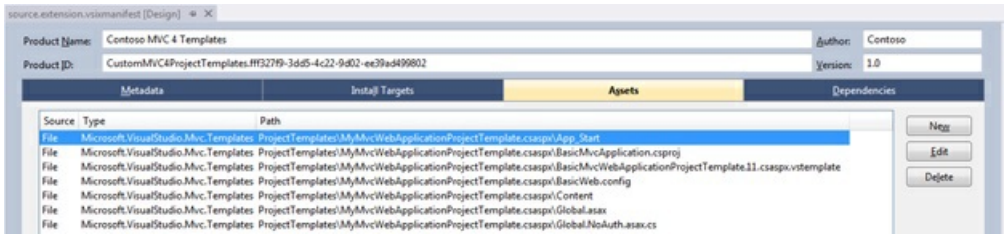
Description="A custom template derived from a Basic MVC web application (Razor)"

Version="4.0"/>

Below is an explanation of the custom attributes that must be present:

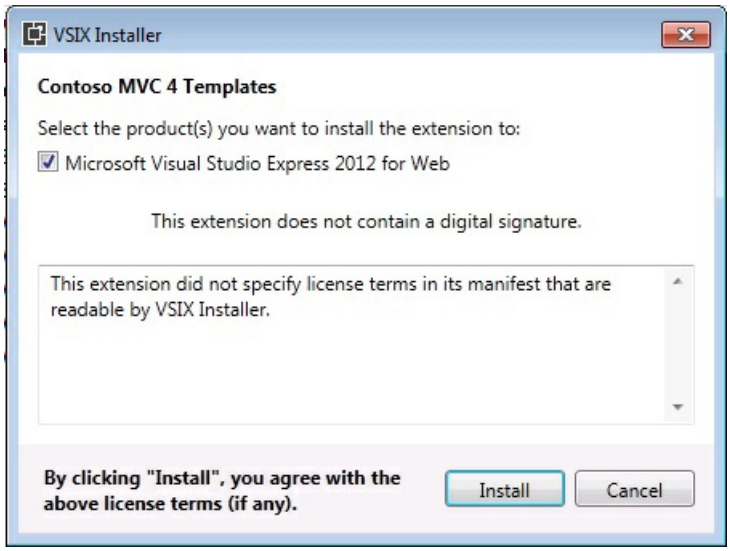
- **ProjectType** must be set to MVC.
- **Language** designates the development language supported by the template. Valid values are either C# or VB.
- **ViewEngine** designates the view engine supported by the template such as Aspx or Razor. You can specify a custom value for this field.
- **TemplateId** is used for grouping the templates. If the value matches an existing template ID it will be override templates previously registered with the MVC wizard.
- **Title** designates the short description displayed in the MVC wizard beneath each project template.
- **Description** designates a more verbose description of the template.

After you have added all the files to the manifest and saved it, you will notice that the **Assets** tab in the designer will display all the files, but not the custom attributes you added to the **<Asset>** elements for the **vstemplate** files.

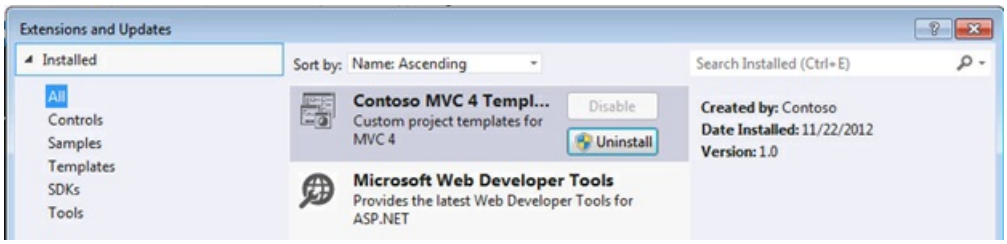


All that remains now is to compile the VSIX project and install it.

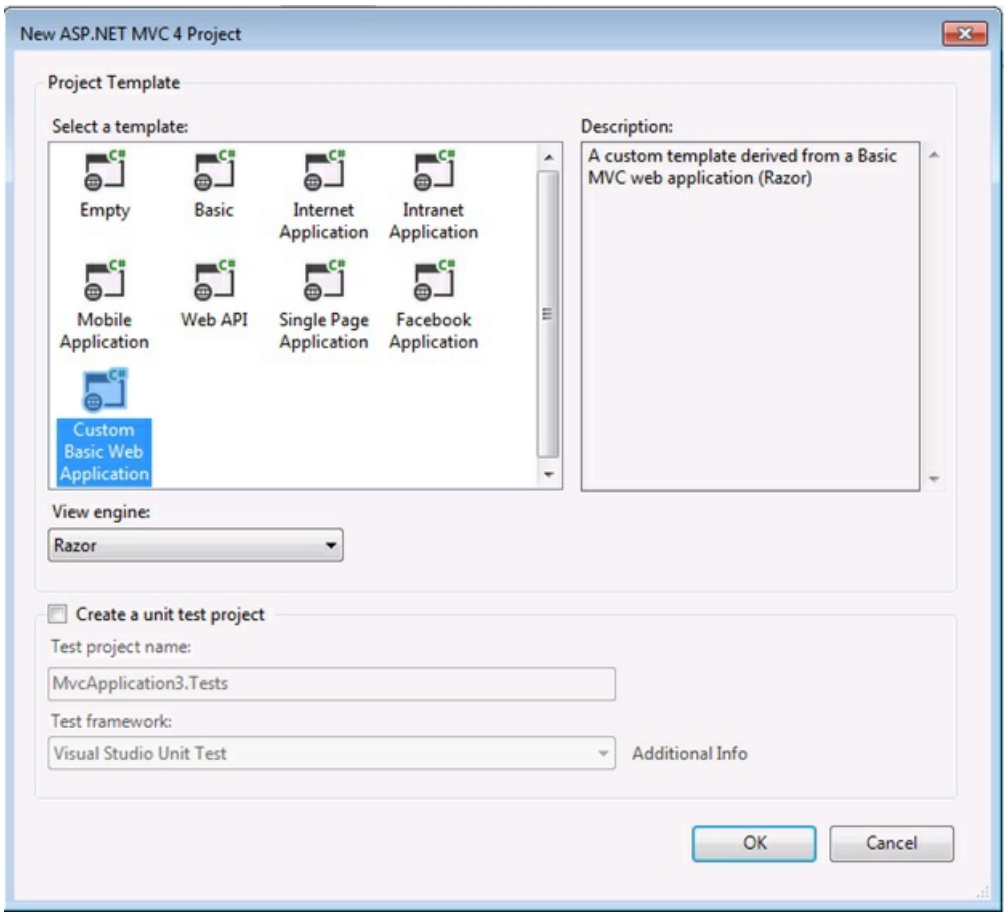
Make sure that all instances of Visual Studio are closed on the machine where you intend to test the VSIX extension. Visual Studio scans for new extensions during startup, so if the IDE is open while installing a VSIX you will need to restart Visual Studio. In Explorer, double click on the VSIX file to launch the **VSIX Installer**, click **Install** and then launch Visual Studio.



From the menu, select **Tools > Extensions and Updates** to confirm that your extension was installed. If the VSIX Installer reported any errors during the installation of the extension you can view the VSIX Installer log for more information. The log is usually created in the **%temp%** folder of the user that installed the extension, for example **C:\Users\Bob\AppData\Local\Temp**.



After closing the window you can create an MVC 4 project to see whether your new templates are shown in the MVC wizard.



Limitations

1. The MVC wizard does not support localized custom templates.
2. The wizard will not report any errors if it fails to locate custom templates. If any of the required custom attributes are absent, the template would simply be excluded from the Wizard.

Releases

2/20/2020 • 2 minutes to read • Edit Online

What's new in recent MVC releases.

- What’s New in ASP.NET MVC 5.2
- What's New in ASP.NET MVC 5.1
- How to Upgrade an ASP.NET MVC 4 and Web API Project to ASP.NET MVC 5 and Web API 2

What's New in ASP.NET MVC 5.2

4/10/2020 • 5 minutes to read • Edit Online

by Microsoft

This topic describes what's new for ASP.NET MVC 5.2, Microsoft.AspNet.MVC 5.2.2 and ASP.NET MVC 5.2.3 Beta

- Software Requirements
- Download
- Documentation
- New Features in ASP.NET MVC 5.2
  - Attribute routing improvements
- Known Issues and Breaking Changes
- Bug Fixes
- Microsoft.AspNet.MVC 5.2.2

Software Requirements

- Visual Studio 2012: Download ASP.NET and Web Tools 2013.1 for Visual Studio 2012.
- Visual Studio 2013: Download Visual Studio 2013 Update or higher. This update is needed for editing ASP.NET MVC 5.2 Razor Views.

Download

The runtime features are released as NuGet packages on the NuGet gallery. All the runtime packages follow the Semantic Versioning specification. The latest ASP.NET MVC 5.2 package has the following version: "5.2.0". You can install or update these packages through NuGet. The release also includes corresponding localized packages on NuGet.

You can install or update to the released NuGet packages by using the NuGet Package Manager Console:

Install-Package Microsoft.AspNet.Mvc -Version 5.2.0

Documentation

Tutorials and other information about ASP.NET MVC 5.2 are available from the ASP.NET web site (https://www.asp.net/mvc).

New Features in ASP.NET MVC 5.2

Attribute routing improvements

Attribute Routing now provides an extensibility point called IDirectRouteProvider, which allows full control over how attribute routes are discovered and configured. An IDirectRouteProvider is responsible for providing a list of actions and controllers along with associated route information to specify exactly what routing configuration is desired for those actions. An IDirectRouteProvider implementation may be specified when calling MapAttributes/MapHttpAttributeRoutes.

Customizing IDirectRouteProvider will be easiest by extending our default implementation, DefaultDirectRouteProvider. This class provides separate overridable virtual methods to change the logic for discovering attributes, creating route entries, and discovering route prefix and area prefix.

With the new attribute routing extensibility of IDirectRouteProvider, a user could do the following:

1. Support Inheritance of attribute routes. For example, in the following scenario Blog and Store controllers are using an attribute route convention that is defined by the BaseController.

```
[InheritedRoute("attributerouting/{controller}/{action=Index}/{id?}")]
public abstract class BaseController : Controller
{
}
public class BlogController : BaseController
{
    public string Index()
    {
        return "Hello from blog!";
    }
}
public class StoreController : BaseController
{
    public string Index()
    {
        return "Hello from store!";
    }
}
[AttributeUsage(AttributeTargets.Class, Inherited=true, AllowMultiple=true)]
public class InheritedRouteAttribute : Attribute, IDirectRouteFactory
{
    public InheritedRouteAttribute(string template)
    {
        Template=template;
    }
    public string Name { get; set; }
    public int Order { get; set; }
    public string Template { get; private set; }
    public new RouteEntry CreateRoute(DirectRouteFactoryContext context)
    {
        // context.Actions will always contain at least one action - and all of the
        // actions will always belong to the same controller.
        var controllerDescriptor=context.Actions.First().ControllerDescriptor;
        var template=Template.Replace("{controller}",
            controllerDescriptor.ControllerName);
        IDirectRouteBuilder builder=context.CreateBuilder(template);
        builder.Name=Name;
        builder.Order=Order;
        return builder.Build();
    }
}
// Custom direct route provider which looks for route attributes of type
// InheritedRouteAttribute and also supports attribute route inheritance.
public class InheritedDirectRouteProvider : DefaultDirectRouteProvider
{
    protected override IReadOnlyList<IDirectRouteFactory>
         GetControllerRouteFactories(ControllerDescriptor controllerDescriptor)
    {
        return controllerDescriptor
            .GetCustomAttributes(typeof(InheritedRouteAttribute), inherit: true)
            .Cast<IDirectRouteFactory>()
            .ToArray();
    }
}
```

2. Automatically generate route names for attribute routes.

```
protected override IReadOnlyList<IDirectRouteFactory>
         GetActionRouteFactories(ActionDescriptor actionDescriptor)
{
    // Get all the route attributes decorated directly on the actions
    IReadOnlyList<IDirectRouteFactory>
actionRouteFactories=base.GetActionRouteFactories(actionDescriptor);
    // Check if the route attribute on each action already has a route name and if no,
    // generate a route name automatically
    // based on the convention: <ControllerName>_<ActionName> (ex: Customers_GetById)
    foreach (IDirectRouteFactory routeFactory in actionRouteFactories)
    {
        RouteAttribute routeAttr=routeFactory as RouteAttribute;
        if (string.IsNullOrEmpty(routeAttr.Name))
        {
            routeAttr.Name=actionDescriptor.ControllerDescriptor.ControllerName + "_"
                  + actionDescriptor.ActionName;
        }
    }
    return actionRouteFactories;
}
protected override IReadOnlyList<IDirectRouteFactory>
      GetControllerRouteFactories(ControllerDescriptor controllerDescriptor)
{
    // Get all the route attributes decorated directly on the controllers
    IReadOnlyList<IDirectRouteFactory>
controllerRouteFactories=base.GetControllerRouteFactories(controllerDescriptor);
    // Check if the route attribute on each controller already has a route name and if no,
    // generate a route name automatically
    // based on the convention: <ControllerName>Route (ex: CustomersRoute)
    foreach (IDirectRouteFactory routeFactory in controllerRouteFactories)
    {
        RouteAttribute routeAttr=routeFactory as RouteAttribute;
        if (string.IsNullOrEmpty(routeAttr.Name))
        {
            routeAttr.Name=controllerDescriptor.ControllerName + "Route";
        }
    }
    return controllerRouteFactories;
}
```

3. Modify route prefixes in one central place before the routes get added to the route table.
4. Filter out the controllers on which you want the attribute routing to look for. We hope to blog on 3 and 4 soon.

Facebook fixes for changed API surface

The MVC Facebook package was broken due to few API changes made by Facebook. We are also releasing a new Facebook package (Microsoft.AspNet.Facebook 1.0.0) to fix these issues.

Known Issues and Breaking Changes

Scaffolding MVC/Web API into a project with 5.2.0 packages results in 5.1.2 packages for ones that don't already exist in the project

Updating NuGet packages for ASP.NET MVC 5.2.0 does not update the Visual Studio tools such as ASP.NET scaffolding or the ASP.NET Web Application project template. They use the previous version of the ASP.NET runtime packages (e.g. 5.1.2 in Update 2). As a result, the ASP.NET scaffolding will install the previous version (e.g. 5.1.2 in Update 2) of the required packages, if they are not already available in your projects. However, the ASP.NET scaffolding in Visual Studio 2013 RTM or Update 1 does not overwrite the latest packages in your projects. If you use ASP.NET scaffolding after updating the packages of your projects to Web API 2.2 or ASP.NET MVC 5.2, make sure the versions of Web API and ASP.NET MVC are consistent.

Microsoft.jQuery.Unobtrusive.Validation NuGet package installation fails because it is unable to find a version of Microsoft.jQuery.Unobtrusive.Validation compatible to jQuery 1.4.1

Microsoft.jQuery.Unobtrusive.Validation requires jQuery >=1.8 and jQuery.Validation >=1.8. But,jQuery.Validation (1.8) needs jQuery (≥ 1.3.2 && ≤ 1.6). Because of this, when NuGet installs the JQuery 1.8 and jQuery.Validation 1.8 at the same time, it fails. When you see this issue, you can simply update the version of jQuery.Validation to >= 1.8.0.1 which has the jQuery cap fixed first, you should be able to install Microsoft.jQuery.Unobtrusive.Validation.

The jquery.Validation nuget package version 1.13.0 does not recognize some international email addresses

jQuery.Validation nuget package version 1.11.1 is the last known version which recognizes following valid email addresses. Any later versions might not be able to recognize them. For example:

E-Mail Address Internationalization (EAI) standard (e.g., 用户@domain.com)
EAI + Internationalized Resource Identifiers (IRIs) (eg., 用户@домен.рф)

The issue is reported at https://github.com/jzaefferer/jquery-validation/issues/1222

Syntax Highlighting for Razor Views in Visual Studio 2013

If you update to ASP.NET MVC 5.2 without updating Visual Studio 2013, you will not get Visual Studio editor support for syntax highlighting while editing the Razor views. You will need to update Visual Studio 2013 to get this support.

Bug Fixes and Minor feature updates

This release also includes several bug fixes and minor feature updates. You can find the complete list here:

- 5.2 package

Microsoft.AspNet.MVC 5.2.2

This release doesn't have any new features or bug fixes in MVC. We made a change in Web Pages for a significant performance improvement and have subsequently updated all other dependent packages we own to depend on this new version of Web Pages.

ASP.NET MVC 5.2.3 Beta

You can read about the release here. This release contains only bug fixes. You can use this query to see the list of issues fixed in this release.

What's New in ASP.NET MVC 5.1

4/10/2020 • 3 minutes to read • Edit Online

by Microsoft

This topic describes what's new for ASP.NET Web MVC 5.1.

- Software Requirements
- Download
- Documentation
- New Features in ASP.NET MVC 5.1
  - Attribute routing improvements
  - Bootstrap support for editor templates
  - Enum support in views
  - Unobtrusive validation for MinLength/MaxLength Attributes
  - Supporting the 'this' context in Unobtrusive Ajax
- Known Issues and Breaking Changes- Bug Fixes

Software Requirements

- Visual Studio 2012: Download ASP.NET and Web Tools 2013.1 for Visual Studio 2012.
- Visual Studio 2013: Download Visual Studio 2013 Update 1. This update is needed for editing ASP.NET MVC 5.1 Razor Views.

Download

The runtime features are released as NuGet packages on the NuGet gallery. All the runtime packages follow the Semantic Versioning specification. The latest ASP.NET MVC 5.1 RTM package has the following version: "5.1.2". You can install or update these packages through NuGet. The release also includes corresponding localized packages on NuGet.

You can install or update to the released NuGet packages by using the NuGet Package Manager Console:

```
Install-Package Microsoft.AspNet.Mvc -Version 5.1.2
```

Documentation

Tutorials and other information about ASP.NET MVC 5.1 RTM are available from the ASP.NET web site (https://www.asp.net).

New Features in ASP.NET MVC 5.1

Attribute routing improvements

Attribute routing now supports constraints, enabling versioning and header based route selection. Many aspects of attribute routes are now customizable via the `IDirectRouteFactory` interface and `RouteFactoryAttribute` class. The route prefix is now extensible via the `IRoutePrefix` interface and `RoutePrefixAttribute` class.

Enum support in views

1. New `@Html.EnumDropDownListFor()` helper methods. These should be used like most of the HTML helpers with the caveat that the expression must evaluate to an enum type or a Nullable<T> where `T` is an enum type. Use `EnumHelper.IsValidForEnumHelper()` to check these requirements.
2. New `EnumHelper.GetSelectList()` methods which return an `IList<SelectListItem>`. This is useful when you need to manipulate a select list prior to calling, for example, `@Html.DropDownListFor()`, or when you wish to display the names which `@Html.EnumDropDownListFor()` shows.

The following code shows these APIs.

```
@if (EnumHelper.IsValidForEnumHelper(ViewData.ModelMetadata))
{
    @Html.EnumDropDownListFor(model => model, htmlAttributes: new { @class = "form-control" })
}
@if (EnumHelper.IsValidForEnumHelper(ViewData.ModelMetadata))
{
    foreach (SelectListItem item in EnumHelper.GetSelectList(ViewData.ModelMetadata,
(Enum)Model)) { … }
}
```

You can see a complete example here.

Bootstrap support for editor templates

We now allow passing in HTML attributes in EditorFor as an anonymous object.

For example:

```
@Html.EditorFor(model => model, new { htmlAttributes = new { @class = "form-control" }, })
```

Unobtrusive validation for MinLengthAttribute and MaxLengthAttribute

Client-side validation for string and array types will now be supported for properties decorated with the MinLength and MaxLength attributes.

Supporting the 'this' context in Unobtrusive Ajax

The callback functions (`OnBegin, OnComplete, OnFailure, OnSuccess`) will now be able to locate the invoking element via the `this` context. For example:

```
@Ajax.ActionLink("Click me", "AjaxAction", new AjaxOptions { UpdateTargetId = "foo", OnBegin = "OnClick" })

<script>
    function OnClick(jqXHR) {
        if ($(this).hasClass("foo")) {
            jqXHR.setRequestHeader("custom-header", "value");
        }
    }
</script>
```

Known Issues and Breaking Changes

Attribute Routing

Ambiguities in attribute routing matches will now report an error rather than choosing the first match.

Attribute routes are prohibited from using the `{controller}` parameter, and from using the `{action}` parameter on routes placed on actions. Uses of these parameters would very likely lead to ambiguities.

Scaffolding MVC/Web API into a project with 5.1 packages results in 5.0 packages for ones that don't already exist in the project

Updating NuGet packages for ASP.NET MVC 5.1 RTM does not update the Visual Studio tools such as ASP.NET scaffolding or the ASP.NET Web Application project template. They use the previous version of the ASP.NET runtime packages (5.0.0.0). As a result, the ASP.NET scaffolding will install the previous version (5.0.0.0) of the required packages, if they are not already available in your projects. However, the ASP.NET scaffolding in Visual Studio 2013 RTM or Update 1 does not overwrite the latest packages in your projects. If you use ASP.NET scaffolding after updating the packages of your projects to Web API 2.1 or ASP.NET MVC 5.1, make sure the versions of Web API and ASP.NET MVC are consistent.

Syntax Highlighting for Razor Views in Visual Studio 2013

If you update to ASP.NET MVC 5.1 RTM without updating Visual Studio 2013, you will not get Visual Studio editor support for syntax highlighting while editing the Razor views. You will need to update Visual Studio 2013 to get this support.

Type Renames

Some of the types used for attribute routing extensibility are renamed in 5.1 RTM.

OLD TYPE NAME (5.1 RC)	NEW TYPE NAME (5.1 RTM)
IDirectRouteProvider	IDirectRouteFactory
RouteProviderAttribute	RouteFactoryAttribute
DirectRouteProviderContext	DirectRouteFactoryContext

Bug Fixes

This release also includes several bug fixes. You can find the complete list here:

- 5.1.0 package
- 5.1.1 package

The 5.1.2 package contains IntelliSense updates but no bug fixes.

How to Upgrade an ASP.NET MVC 4 and Web API Project to ASP.NET MVC 5 and Web API 2

2/20/2020 • 5 minutes to read • Edit Online

by Rick Anderson

ASP.NET MVC 5 and Web API 2 bring a host of new features, including attribute routing, authentication filters, and much more. See https://www.asp.net/vnext for more details.

This walkthrough will guide you with the steps required to upgrade your application to the latest version.

> **NOTE**
>
> Please see ASP.NET and Web Tools for Visual Studio 2013 Release Notes for information on breaking changes from MVC 4 and Web API to the next version.

This article was written by Youngjune Hong and Rick Anderson (@RickAndMSFT)

Upgrade Steps

1. Backup your project. This walkthrough will require you to make changes to your project file, package configuration, and web.config files.

2. For upgrading from Web API to Web API 2, in global.asax, change:

   ```
   WebApiConfig.Register(GlobalConfiguration.Configuration);
   ```

   to

   ```
   GlobalConfiguration.Configure(WebApiConfig.Register);
   ```

3. Make sure all the packages that your projects use are compatible with MVC 5 and Web API 2. The following table shows the MVC 4 and Web API related packages than need to be changed. If you have a package that is dependent on one of the packages listed below, please contact the publishers to get the newer versions that are compatible with MVC 5 and Web API 2. If you have the source code for those packages, you should recompile them with the new assemblies of MVC 5 and Web API 2.

PACKAGE ID	OLD VERSION	NEW VERSION
Microsoft.AspNet.Razor	2.0.x.x	3.0.0
Microsoft.AspNet.WebPages	2.0.x.x	3.0.0
Microsoft.AspNet.WebPages.WebData	2.0.x.x	3.0.0
Microsoft.AspNet.WebPages.OAuth	2.0.x.x	3.0.0

PACKAGE ID	OLD VERSION	NEW VERSION
Microsoft.AspNet.Mvc	4.0.x.x	5.0.0
Microsoft.AspNet.Mvc.Facebook	4.0.x.x	5.0.0
Microsoft.AspNet.WebApi.Core	4.0.x.x	5.0.0
Microsoft.AspNet.WebApi.SelfHost	4.0.x.x	5.0.0
Microsoft.AspNet.WebApi.Client	4.0.x.x	5.0.0
Microsoft.AspNet.WebApi.OData	4.0.x.x	5.0.0
Microsoft.AspNet.WebApi	4.0.x.x	5.0.0
Microsoft.AspNet.WebApi.WebHost	4.0.x.x	5.0.0
Microsoft.AspNet.WebApi.Tracing	4.0.x.x	5.0.0
Microsoft.AspNet.WebApi.HelpPage	4.0.x.x	5.0.0
Microsoft.Net.Http	2.0.x.	2.2.x.
Microsoft.Data.OData	5.2.x	5.6.x
System.Spatial	5.2.x	5.6.x
Microsoft.Data.Edm	5.2.x	5.6.x
Microsoft.AspNet.Mvc.FixedDisplayModes	<o:p> </o:p>	Removed
Microsoft.AspNet.WebPages.Administration	<o:p> </o:p>	Removed
Microsoft-Web-Helpers	<o:p> </o:p>	Microsoft.AspNet.WebHelpers

> **NOTE**
>
> Microsoft-Web-Helpers has been replaced with Microsoft.AspNet.WebHelpers. You should remove the old package first, and then install the newer package.
>
> There is no cross version compatibility among major ASP.NET packages. For example, MVC 5 is compatible with only Razor 3, and not Razor 2.

4. Open your project in Visual Studio.

5. Remove any of the following ASP.NET NuGet packages that are installed. You will remove these using the Package Manager Console (PMC). To open the PMC, select the **Tools** menu and then select **NuGet Package Manager,** then select **Package Manager Console**. Your project might not include all of these.

   a. `Microsoft.AspNet.WebPages.Administration`
   This package is typically added when upgrading from MVC 3 to MVC 4. To remove it, run the following

command in the PMC:
`Uninstall-Package -Id Microsoft.AspNet.WebPages.Administration`

b. `Microsoft-Web-Helpers`
This package has been rebranded as `Microsoft.AspNet.WebHelpers`. To remove it, run the following command in the PMC:
`Uninstall-Package -Id Microsoft-Web-Helpers`

c. `Microsoft.AspNet.Mvc.FixedDisplayMode`
This package contains a work around for a bug in MVC 4 that has been fixed in MVC 5. To remove it, run the following command in the PMC:
`Uninstall-Package -Id Microsoft.AspNet.Mvc.FixedDisplayModes`

6. Upgrade all the ASP.NET NuGet packages using the PMC. In the PMC, run the following command:
`Update-Package`
The `Update-Package` command without any parameters will update every package. You can update packages individually by using the ID argument. For more information about the update command, run `get-help update-package`.

Update the Application *web.config* File

Be sure to make these changes in the app *web.config* file, not the *web.config* file in the *Views* folder.

Locate the `<runtime>/<assemblyBinding>` section, and make the following changes:

1. In the elements with the name attribute "System.Web.Mvc", change the version number from "4.0.0.0" to "5.0.0.0". (Two changes in that element.)

2. In elements with the name attribute "System.Web.Helpers" and "System.Web.WebPages" change the version number from "2.0.0.0" to "3.0.0.0". Four changes will occur, two in each of the elements.

```
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <!--Two elements removed for Clarity -->
    <dependentAssembly>
      <assemblyIdentity name="System.Web.Helpers" publicKeyToken="31bf3856ad364e35" />
      <bindingRedirect oldVersion="1.0.0.0-3.0.0.0" newVersion="3.0.0.0" />
    </dependentAssembly>
    <dependentAssembly>
      <assemblyIdentity name="System.Web.Mvc" publicKeyToken="31bf3856ad364e35" />
      <bindingRedirect oldVersion="1.0.0.0-5.0.0.0" newVersion="5.0.0.0" />
    </dependentAssembly>
    <dependentAssembly>
      <assemblyIdentity name="System.Web.WebPages" publicKeyToken="31bf3856ad364e35" />
      <bindingRedirect oldVersion="1.0.0.0-3.0.0.0" newVersion="3.0.0.0" />
    </dependentAssembly>
    <!--WebGrease element removed for Clarity -->
  </assemblyBinding
```

3. Locate the `<appSettings>` section and update the webpages:version from 2.0.0.0 to 3.0.0.0 as shown below:

```
<appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="PreserveLoginUrl" value="true" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  </appSettings>
```

4. Remove any trust levels other than Full. For example:

```
<securityPolicy>
   <!--<trustLevel name="Medium"  policyFile="web_mediumtrust.config"/>-->
</securityPolicy>
```

Update the *web.config* files under the Views folder

If your application is using areas, you will also need to update each *web.config* file in the *Views* sub-folder of each Area folder.

1. Update all elements that contain "System.Web.Mvc" from version "4.0.0.0" to version "5.0.0.0".

```
<system.web.webPages.razor>
  <host factoryType="System.Web.Mvc.MvcWebRazorHostFactory, System.Web.Mvc, Version=5.0.0.0,
Culture=neutral, PublicKeyToken=31BF3856AD364E35" />
  <pages pageBaseType="System.Web.Mvc.WebViewPage">
    <namespaces>
      <add namespace="System.Web.Mvc" />
      <!--Elements removed for Clarity.-->
    </namespaces>
  </pages>
</system.web.webPages.razor>
```

```
-->
  <pages
      validateRequest="false"
      pageParserFilterType="System.Web.Mvc.ViewTypeParserFilter, System.Web.Mvc, Version=5.0.0.0,
Culture=neutral, PublicKeyToken=31BF3856AD364E35"
      pageBaseType="System.Web.Mvc.ViewPage, System.Web.Mvc, Version=5.0.0.0, Culture=neutral,
PublicKeyToken=31BF3856AD364E35"
      userControlBaseType="System.Web.Mvc.ViewUserControl, System.Web.Mvc, Version=5.0.0.0,
Culture=neutral, PublicKeyToken=31BF3856AD364E35">
    <controls>
      <add assembly="System.Web.Mvc, Version=5.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"
namespace="System.Web.Mvc" tagPrefix="mvc" />
    </controls>
  </pages>
</system.web>
```

2. Update all elements that contain "System.Web.WebPages.Razor" from version "2.0.0.0" to version"3.0.0.0". If this section contains "System.Web.WebPages", update those elements from version "2.0.0.0" to version"3.0.0.0"

```
<configuration>
  <configSections>
    <sectionGroup name="system.web.webPages.razor"
type="System.Web.WebPages.Razor.Configuration.RazorWebSectionGroup, System.Web.WebPages.Razor,
Version=3.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
      <section name="host" type="System.Web.WebPages.Razor.Configuration.HostSection,
System.Web.WebPages.Razor, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"
requirePermission="false" />
      <section name="pages" type="System.Web.WebPages.Razor.Configuration.RazorPagesSection,
System.Web.WebPages.Razor, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"
requirePermission="false" />
    </sectionGroup>
  </configSections>
```

3. If you removed the `Microsoft-Web-Helpers` NuGet package in a previous step, install `Microsoft.AspNet.WebHelpers` with the following command in the PMC:

```
Install-Package -Id Microsoft.AspNet.WebHelpers
```

4. If your app uses the User.IsInRole() method, add the following to the *Web.config* file.

```
<system.webServer>
    <modules>
      <remove name="RoleManager" />
    </modules>
  </system.webServer>
```

Final Steps

Build and test the application.

Remove the MVC 4 project type GUID from the project files.

1. In Solution Explorer, right-click the project name and then select **Unload Project**.
2. Right-click the project and select Edit ProjectName.csproj.
3. Locate the `ProjectTypeGuids` element and then remove the MVC 4 project GUID, `{E3E379DF-F4C6-4180-9B81-6769533ABE47}`.
4. Save and close the open project file.
5. Right-click the project and select **Reload Project**.

Older Versions - MVC 3 and 4

2/20/2020 • 2 minutes to read • Edit Online

Links to resources for learning how to use earlier versions of ASP.NET MVC.

- Getting Started with ASP.NET MVC 4
  - Intro to ASP.NET MVC 4
  - Adding a Controller
  - Adding a View
  - Adding a Model
  - Accessing Your Model's Data from a Controller
  - Examining the Edit Methods and Edit View
  - Adding a New Field to the Movie Model and Table
  - Adding Validation to the Model
  - Examining the Details and Delete Methods
- Getting Started with EF 5 using MVC 4
  - Creating an Entity Framework Data Model
  - Implementing Basic CRUD Functionality
  - Sorting, Filtering, and Paging
  - Creating a More Complex Data Model
  - Reading Related Data
  - Updating Related Data
  - Handling Concurrency
  - Implementing Inheritance
  - Implementing the Repository and Unit of Work Patterns
  - Advanced Entity Framework Scenarios
  - Building the Chapter Downloads
- MVC 4 Hands On Labs
  - What's New in ASP.NET MVC 4
  - ASP.NET MVC 4 Fundamentals
  - ASP.NET MVC 4 Models and Data Access
  - ASP.NET MVC 4 Custom Action Filters
  - ASP.NET MVC 4 Entity Framework Scaffolding and Migrations
  - ASP.NET MVC 4 Dependency Injection
  - ASP.NET MVC 4 Helpers, Forms and Validation
- ASP.NET MVC 4 Mobile Features
- Using OAuth Providers with MVC 4
- Getting Started with ASP.NET MVC3
  - Getting Started with ASP.NET MVC 3 (C#)
    - Intro to ASP.NET MVC 3 (C#)
    - Adding a Controller (C#)

    - Adding a View (C#)
    - Adding a Model (C#)
    - Accessing your Model's Data from a Controller (C#)
    - Examining the Edit Methods and Edit View (C#)
    - Adding a New Field to the Movie Model and Table (C#)
    - Adding Validation to the Model (C#)
    - Improving the Details and Delete Methods (C#)
  - Getting Started with ASP.NET MVC 3 (VB.NET)
    - Intro to ASP.NET MVC 3 (VB)
    - Adding a Controller (VB)
    - Adding a View (VB)
    - Adding a Model (VB)
    - Accessing your Model's Data from a Controller (VB)
    - Examining the Edit Methods and Edit View (VB)
    - Adding a New Field to the Movie Model and Database Table (VB)
    - Adding Validation to the Model (VB)
    - Improving the Details and Delete Methods (VB)
- Creating a MVC 3 Application with Razor and Unobtrusive JavaScript
- MVC 3 Music Store
  - Part 1: Overview and File->New Project
  - Part 2: Controllers
  - Part 3: Views and ViewModels
  - Part 4: Models and Data Access
  - Part 5: Edit Forms and Templating
  - Part 6: Using Data Annotations for Model Validation
  - Part 7: Membership and Authorization
  - Part 8: Shopping Cart with Ajax Updates
  - Part 9: Registration and Checkout
  - Part 10: Final Updates to Navigation and Site Design, Conclusion
- MVC 3 - The HTML5 and jQuery UI Datepicker Popup Calendar
  - HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 1
  - HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 2
  - HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 3
  - HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 4
- MVC 3 - Working with the DropDownList Box and jQuery
  - Using the DropDownList Helper with ASP.NET MVC
  - Examining how ASP.NET MVC scaffolds the DropDownList Helper
  - Adding a New Category to the DropDownList using jQuery UI

Getting Started with ASP.NET MVC 4

2/20/2020 • 2 minutes to read • Edit Online

This tutorial series will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio.

- Intro to ASP.NET MVC 4
- Adding a Controller
- Adding a View
- Adding a Model
- Accessing Your Model's Data from a Controller
- Examining the Edit Methods and Edit View
- Adding a New Field to the Movie Model and Table
- Adding Validation to the Model
- Examining the Details and Delete Methods

Intro to ASP.NET MVC 4

2/20/2020 • 3 minutes to read • Edit Online

by Rick Anderson

An updated version if this tutorial is available here using Visual Studio 2013. The new tutorial uses ASP.NET MVC 5, which provides many improvements over this tutorial.

This tutorial will teach you the basics of building an ASP.NET MVC 4 Web application using Microsoft Visual Studio Express 2012 or Visual Web Developer 2010 Express Service Pack 1. Visual Studio 2012 is recommended, you won't need to install anything to complete the tutorial. If you are using Visual Studio 2010 you must install the components below. You can install all of them by clicking the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- WPI installer for ASP.NET MVC 4
- LocalDB
- SSDT

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the WPI installer for ASP.NET MVC 4 and the: Visual Studio 2010 prerequisites

A Visual Web Developer project with C# source code is available to accompany this topic. Download the C# version.

In the tutorial you run the application in Visual Studio. You can also make the application available over the Internet by deploying it to a hosting provider. Microsoft offers free web hosting for up to 10 web sites in a free Windows Azure trial account. For information about how to deploy a Visual Studio web project to a Windows Azure Web Site, see Create and deploy an ASP.NET web site and SQL Database with Visual Studio. That tutorial also shows how to use Entity Framework Code First Migrations to deploy your SQL Server database to Windows Azure SQL Database (formerly SQL Azure).

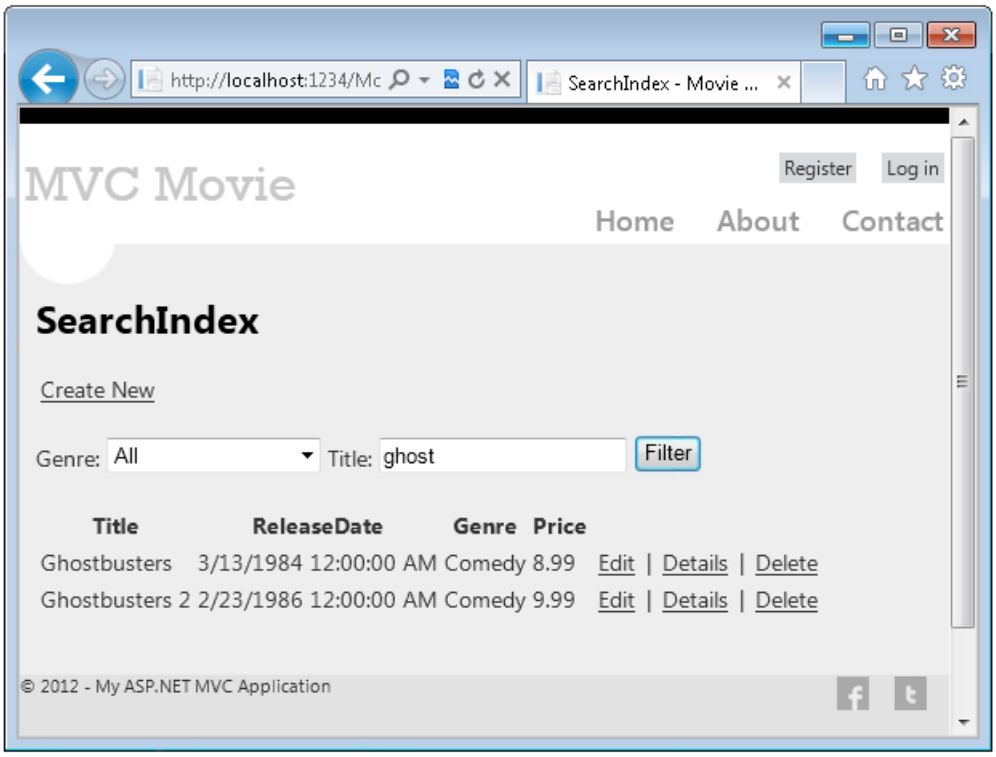
This tutorial was written by Rick Anderson (@RickAndMSFT).

What You'll Build

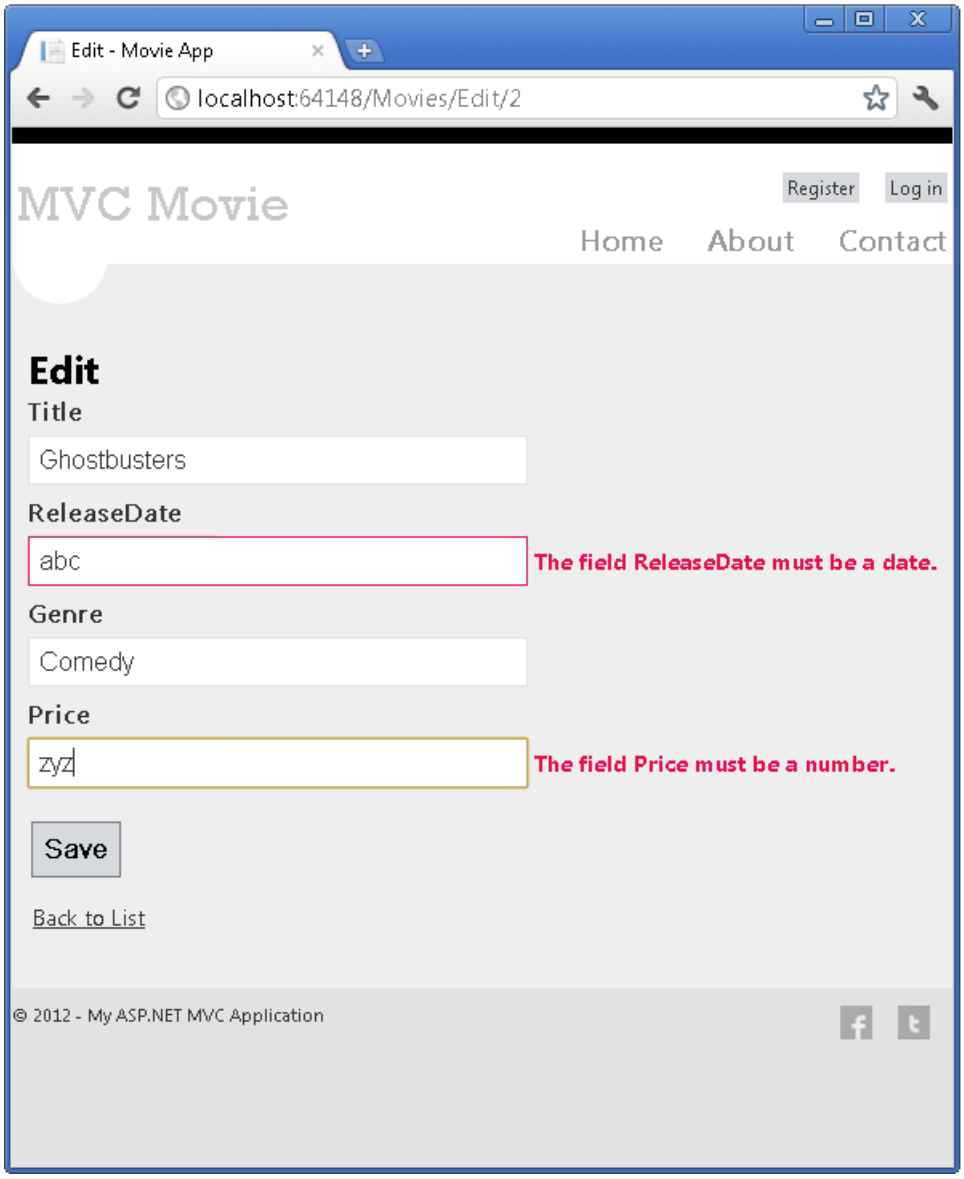
NOTE

An updated version if this tutorial is available here using Visual Studio 2013. The new tutorial uses ASP.NET MVC 5, which provides many improvements over this tutorial.

You'll implement a simple movie-listing application that supports creating, editing, searching and listing movies from a database. Below are two screenshots of the application you'll build. It includes a page that displays a list of movies from a database:



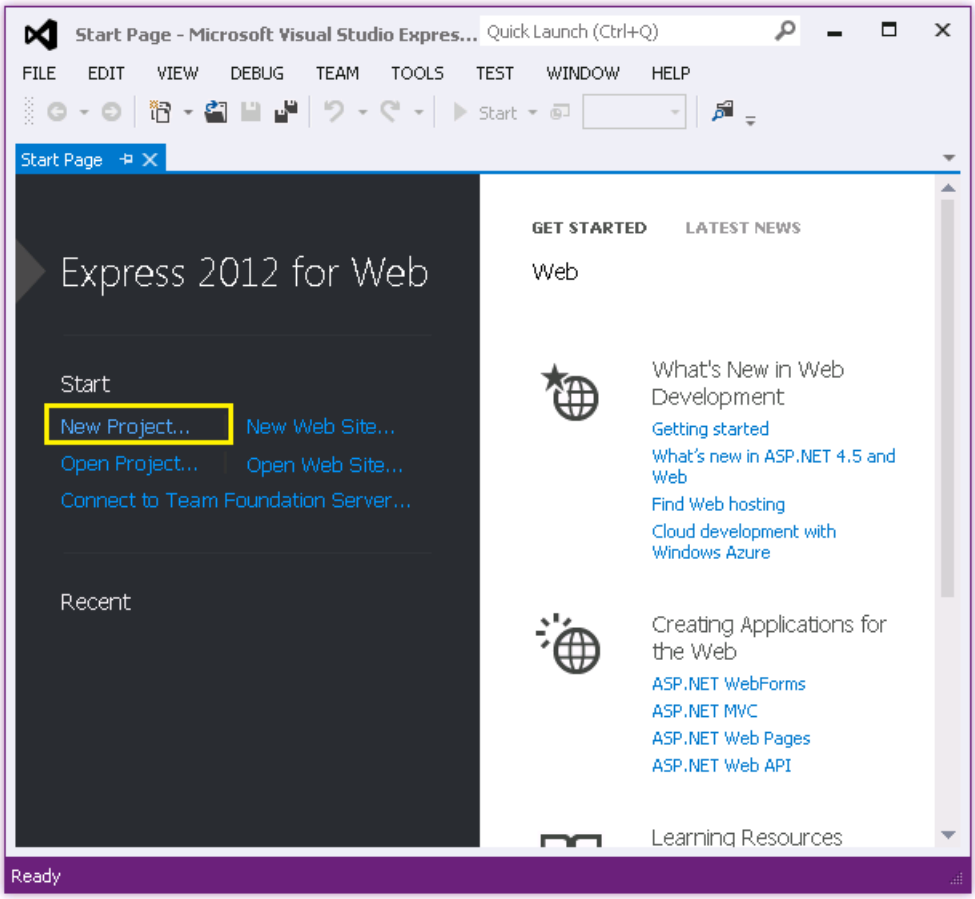
The application also lets you add, edit, and delete movies, as well as see details about individual ones. All data-entry scenarios include validation to ensure that the data stored in the database is correct.



Getting Started

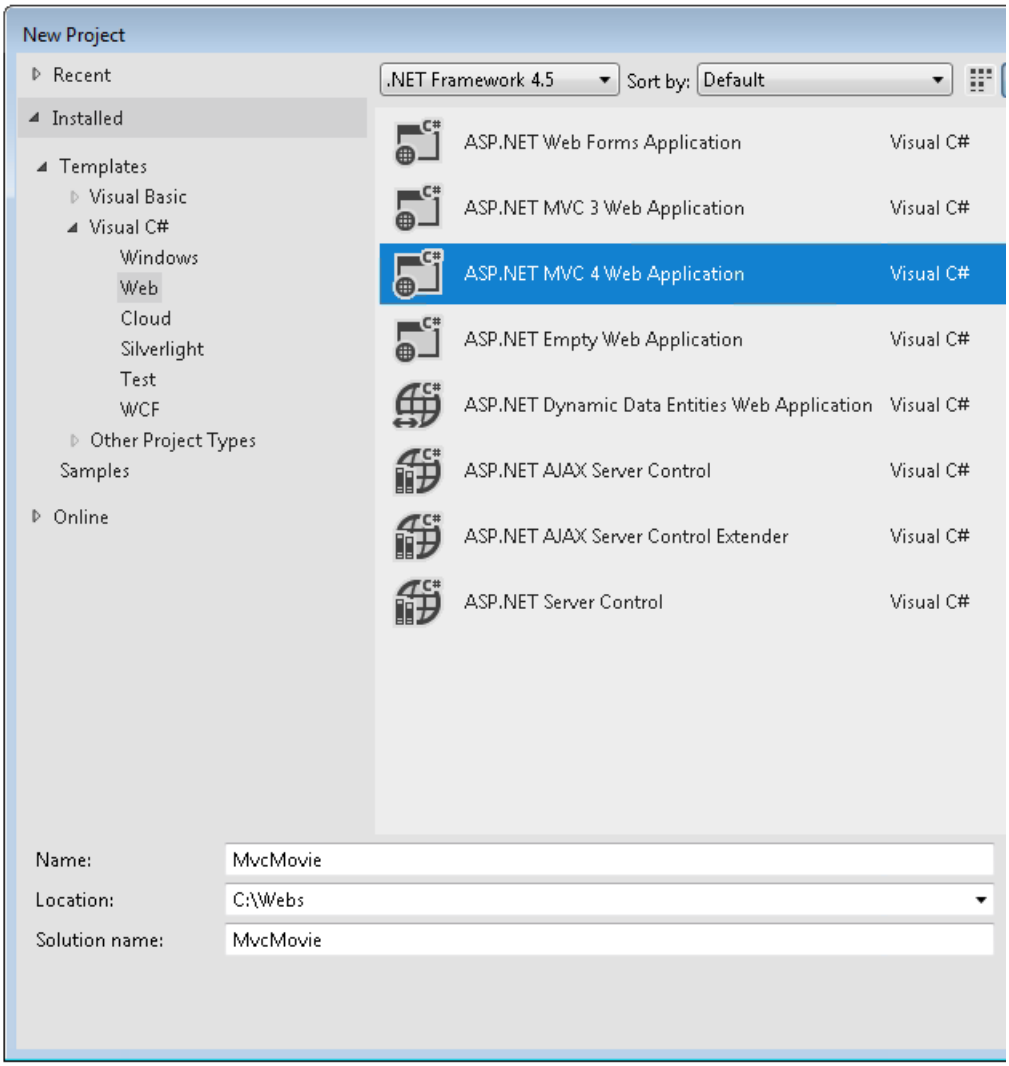
Start by running Visual Studio Express 2012 or Visual Web Developer 2010 Express. Most of the screen shots in this series use Visual Studio Express 2012, but you can complete this tutorial with Visual Studio 2010/SP1, Visual Studio 2012, Visual Studio Express 2012 or Visual Web Developer 2010 Express. Select **New Project** from the **Start** page.

Visual Studio is an IDE, or integrated development environment. Just like you use Microsoft Word to write documents, you'll use an IDE to create applications. In Visual Studio there's a toolbar along the top showing various options available to you. There's also a menu that provides another way to perform tasks in the IDE. (For example, instead of selecting **New Project** from the **Start** page, you can use the menu and select **File** > **New Project**.)

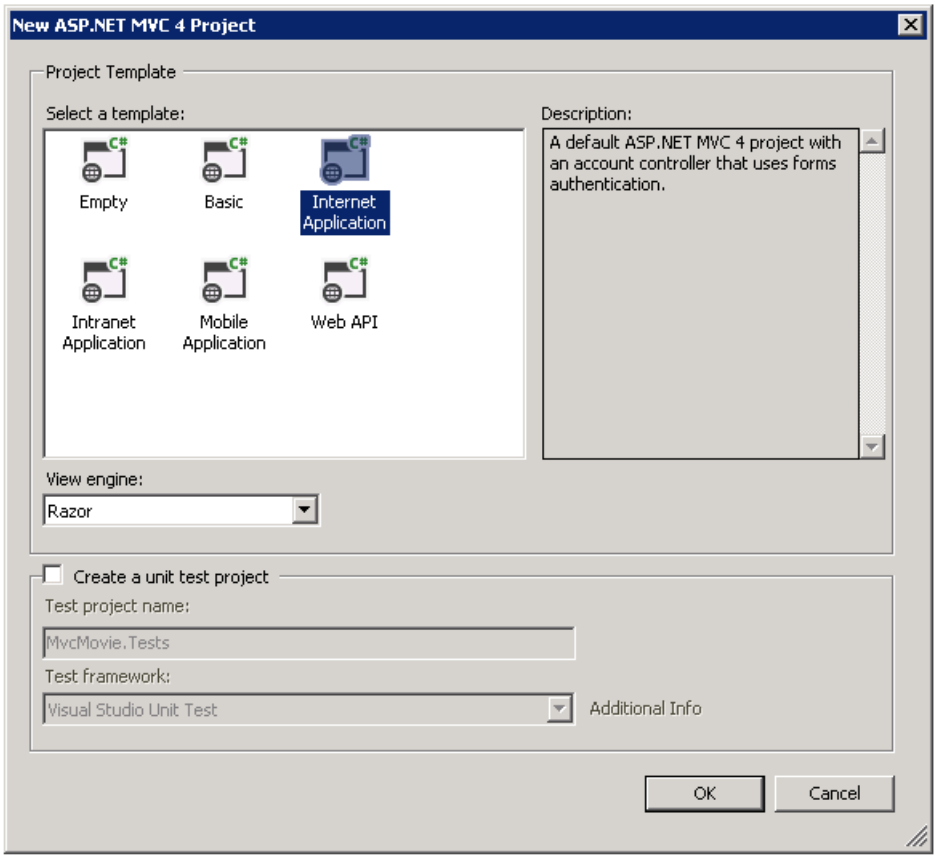


Creating Your First Application

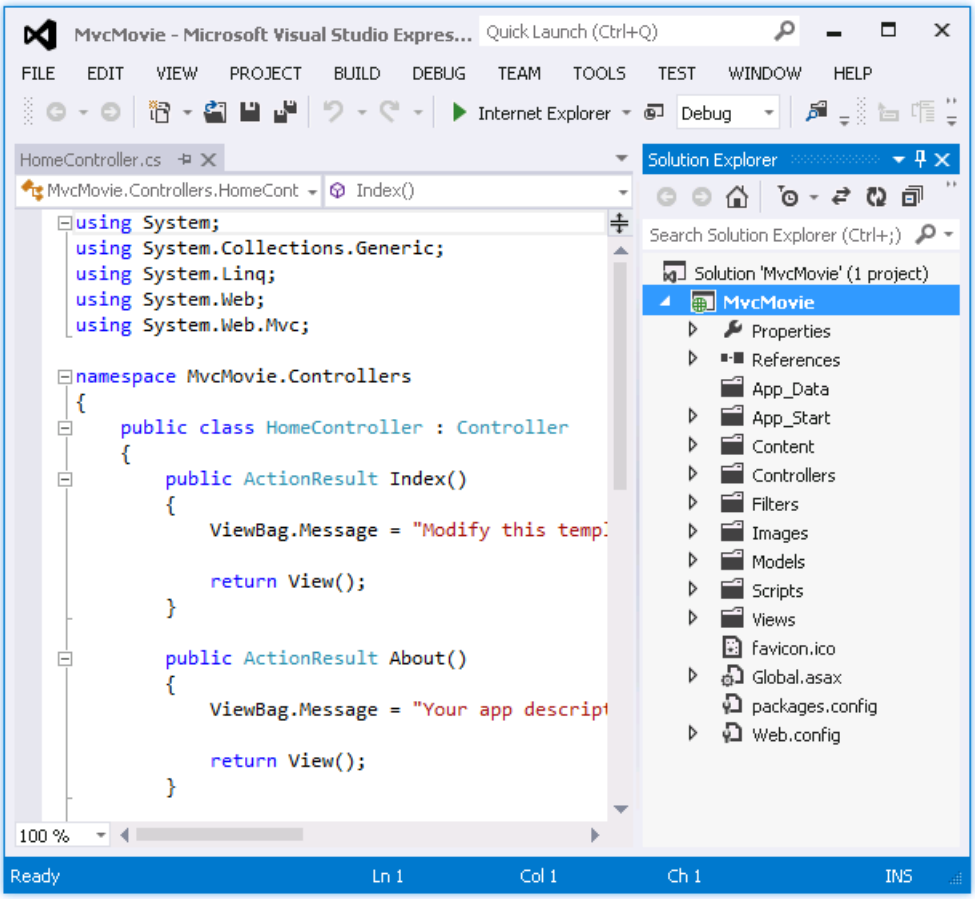
You can create applications using either Visual Basic or Visual C# as the programming language. Select Visual C# on the left and then select **ASP.NET MVC 4 Web Application**. Name your project "MvcMovie" and then click **OK**.



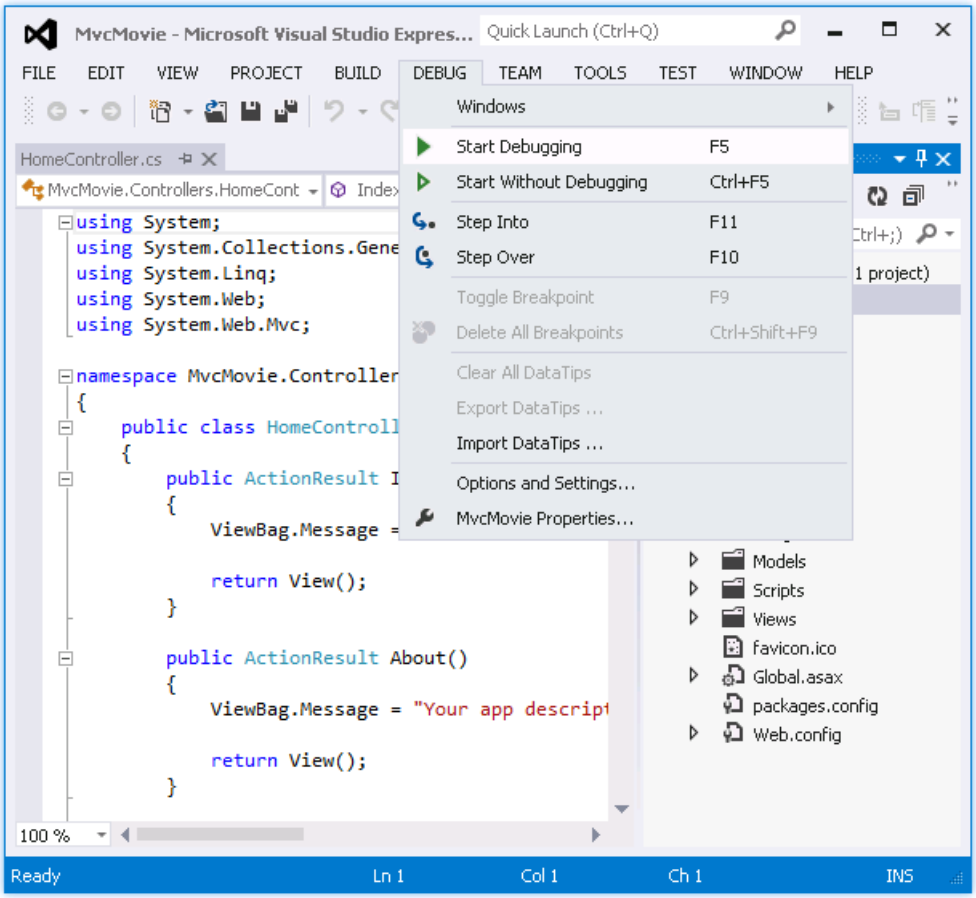
In the **New ASP.NET MVC 4 Project** dialog box, select **Internet Application**. Leave **Razor** as the default view engine.



Click **OK**. Visual Studio used a default template for the ASP.NET MVC project you just created, so you have a working application right now without doing anything! This is a simple "Hello World!" project, and it's a good place to start your application.

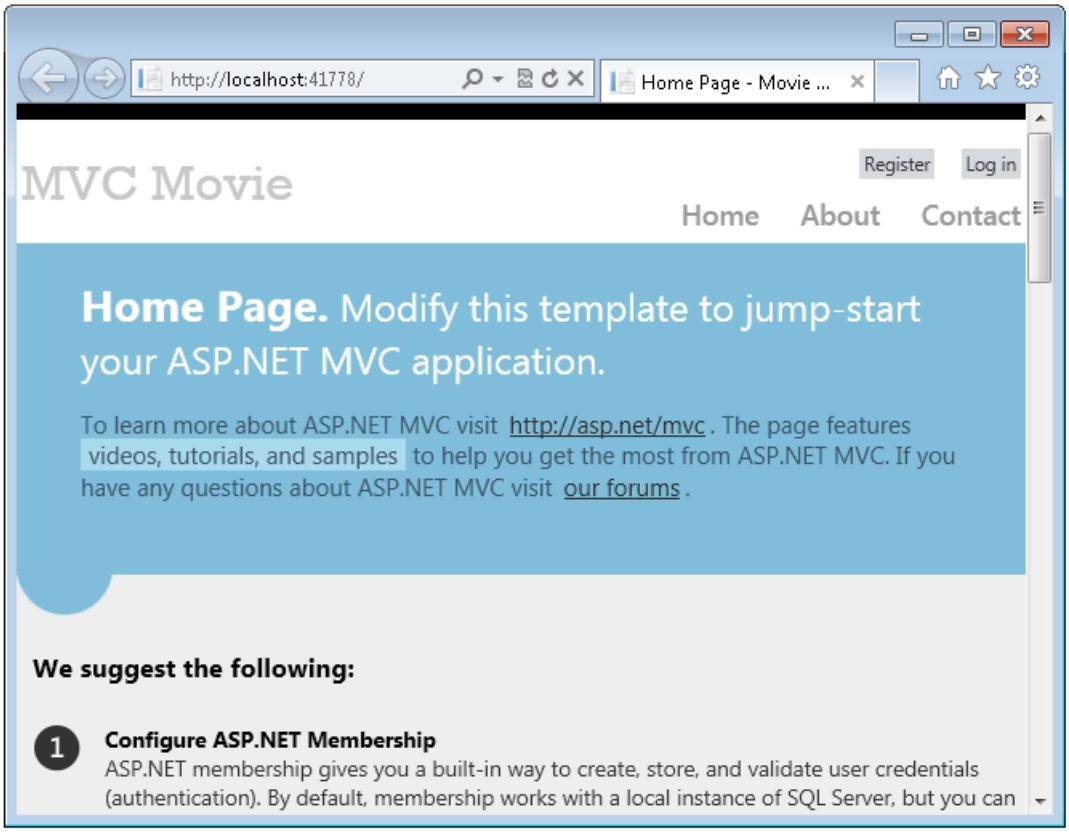


From the **Debug** menu, select **Start Debugging**.



Notice that the keyboard shortcut to start debugging is F5.

F5 causes Visual Studio to start IIS Express and run your web application. Visual Studio then launches a browser and opens the application's home page. Notice that the address bar of the browser says `localhost` and not something like `example.com`. That's because `localhost` always points to your own local computer, which in this case is running the application you just built. When Visual Studio runs a web project, a random port is used for the web server. In the image below, the port number is 41788. When you run the application, you'll probably see a different port number.



Right out of the box this default template gives you Home, Contact and About pages. It also provides support to register and log in, and links to Facebook and Twitter. The next step is to change how this application works and learn a little bit about ASP.NET MVC. Close your browser and let's change some code.

NEXT

Adding a Controller

2/20/2020 • 3 minutes to read • Edit Online

by Rick Anderson

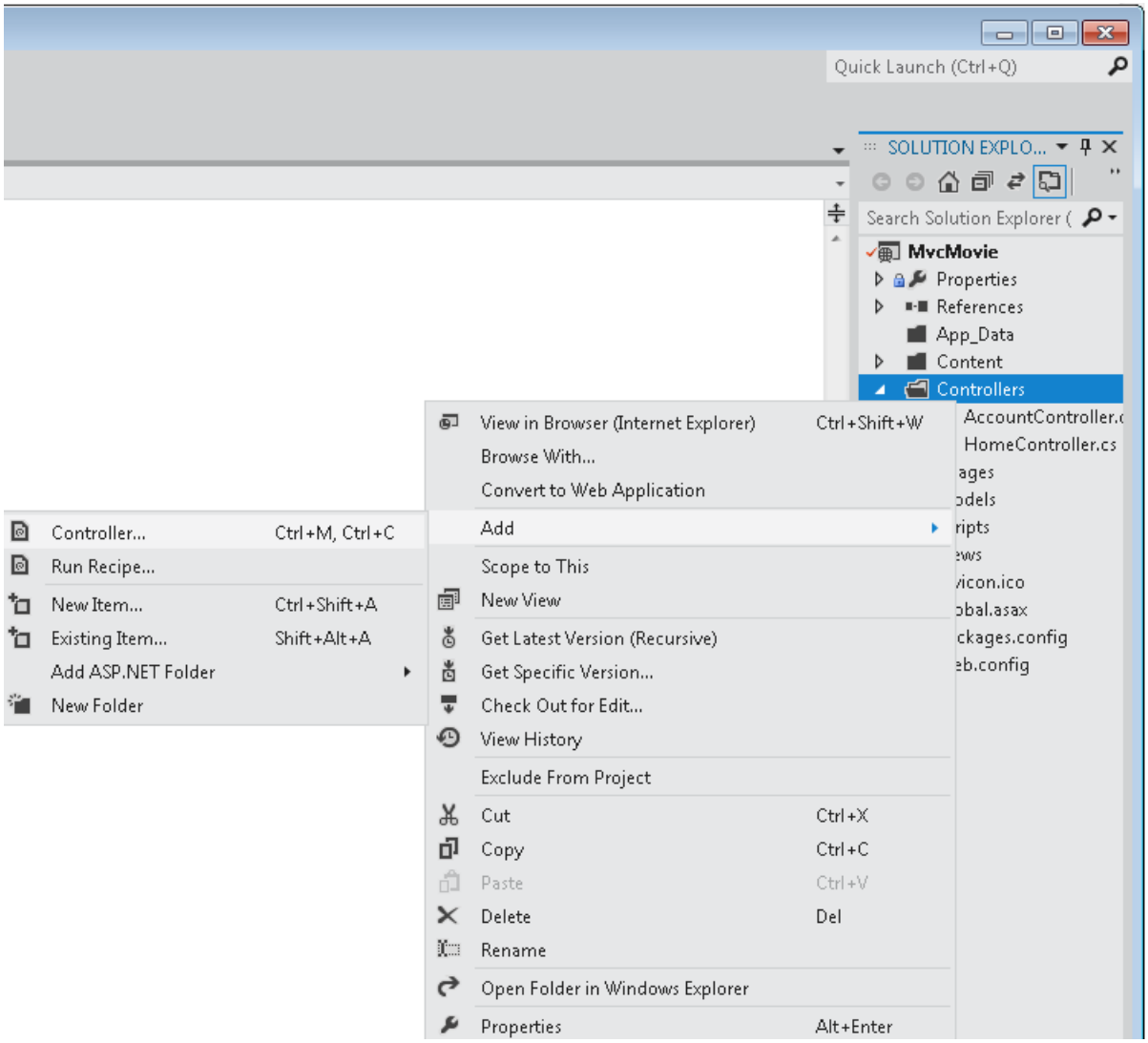
> **NOTE**
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

MVC stands for *model-view-controller*. MVC is a pattern for developing applications that are well architected, testable and easy to maintain. MVC-based applications contain:

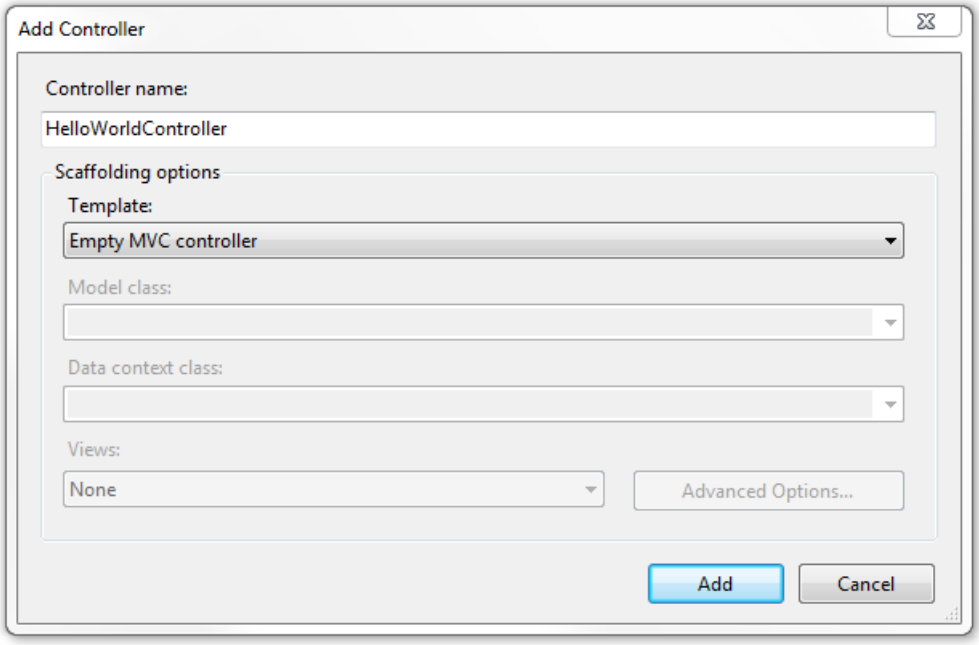
- **M** odels: Classes that represent the data of the application and that use validation logic to enforce business rules for that data.
- **V** iews: Template files that your application uses to dynamically generate HTML responses.
- **C** ontrollers: Classes that handle incoming browser requests, retrieve model data, and then specify view templates that return a response to the browser.

We'll be covering all these concepts in this tutorial series and show you how to use them to build an application.

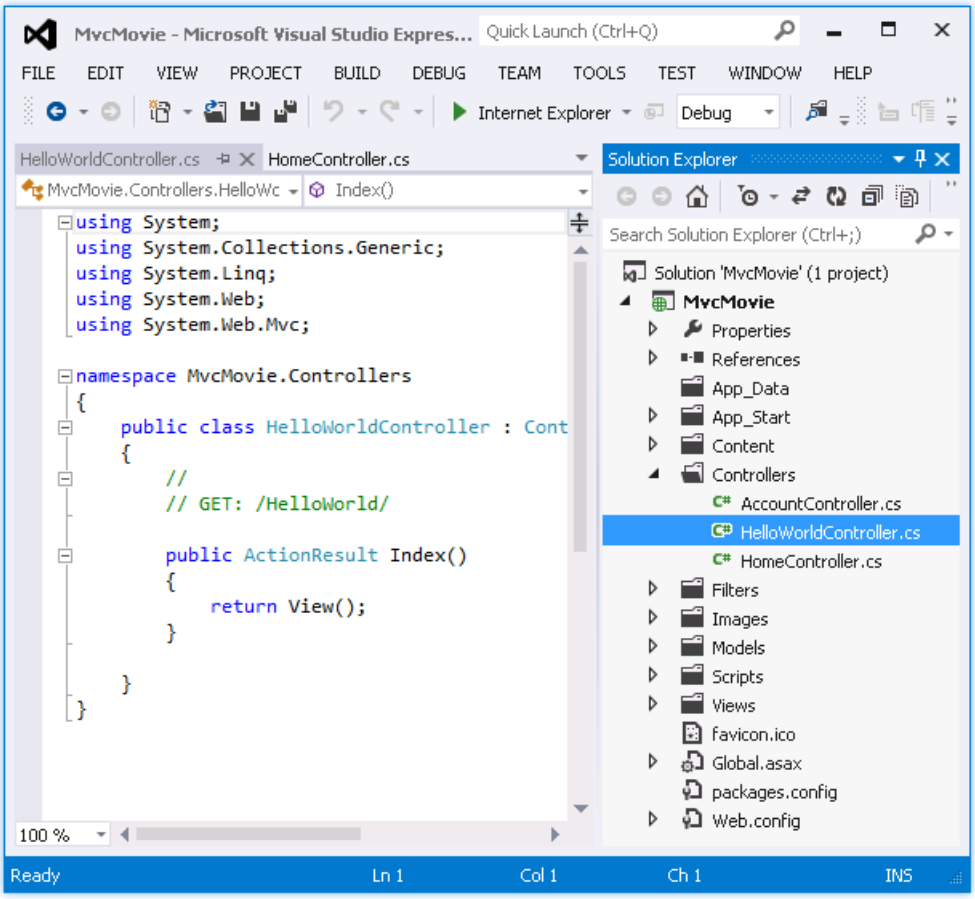
Let's begin by creating a controller class. In **Solution Explorer**, right-click the *Controllers* folder and then select **Add Controller**.



Name your new controller "HelloWorldController". Leave the default template as **Empty MVC controller** and click **Add**.



Notice in **Solution Explorer** that a new file has been created named *HelloWorldController.cs*. The file is open in the IDE.



Replace the contents of the file with the following code.

```
using System.Web;
using System.Web.Mvc;

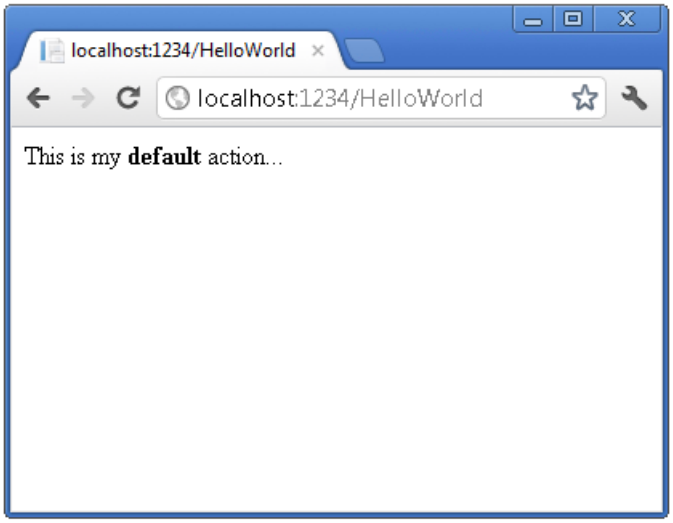
namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my <b>default</b> action...";
        }

        //
        // GET: /HelloWorld/Welcome/

        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

The controller methods will return a string of HTML as an example. The controller is named `HelloWorldController` and the first method above is named `Index`. Let's invoke it from a browser. Run the application (press F5 or Ctrl+F5). In the browser, append "HelloWorld" to the path in the address bar. (For example, in the illustration below, it's `http://localhost:1234/HelloWorld.`) The page in the browser will look like the following screenshot. In the method above, the code returned a string directly. You told the system to just return some HTML, and it did!

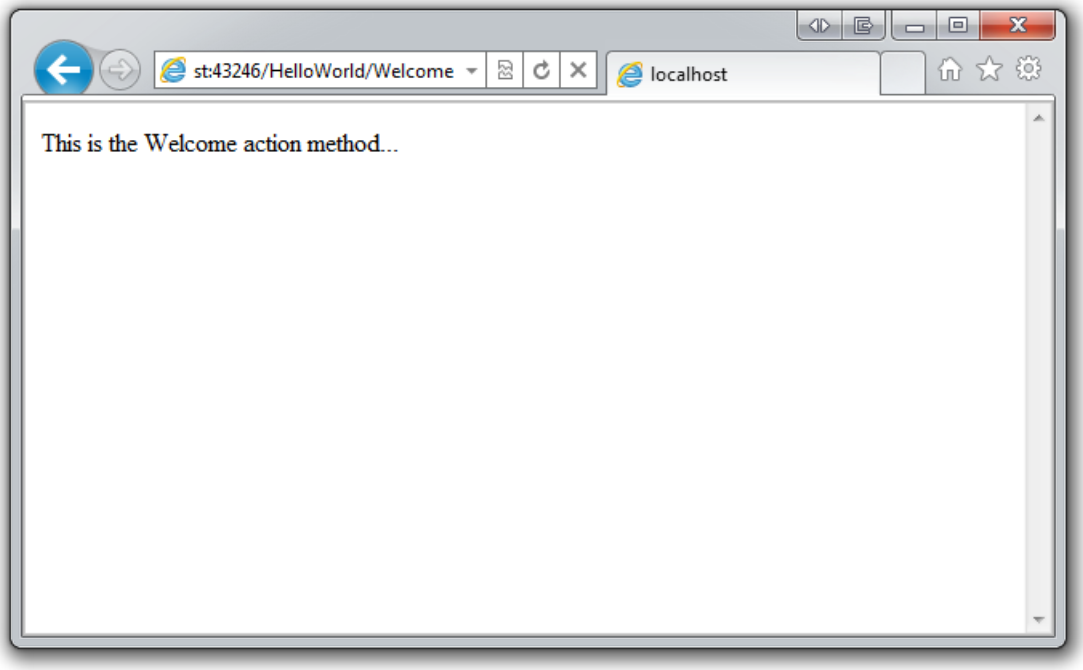


ASP.NET MVC invokes different controller classes (and different action methods within them) depending on the incoming URL. The default URL routing logic used by ASP.NET MVC uses a format like this to determine what code to invoke:

`/[Controller]/[ActionName]/[Parameters]`

The first part of the URL determines the controller class to execute. So */HelloWorld* maps to the `HelloWorldController` class. The second part of the URL determines the action method on the class to execute. So */HelloWorld/Index* would cause the `Index` method of the `HelloWorldController` class to execute. Notice that we only had to browse to */HelloWorld* and the `Index` method was used by default. This is because a method named `Index` is the default method that will be called on a controller if one is not explicitly specified.

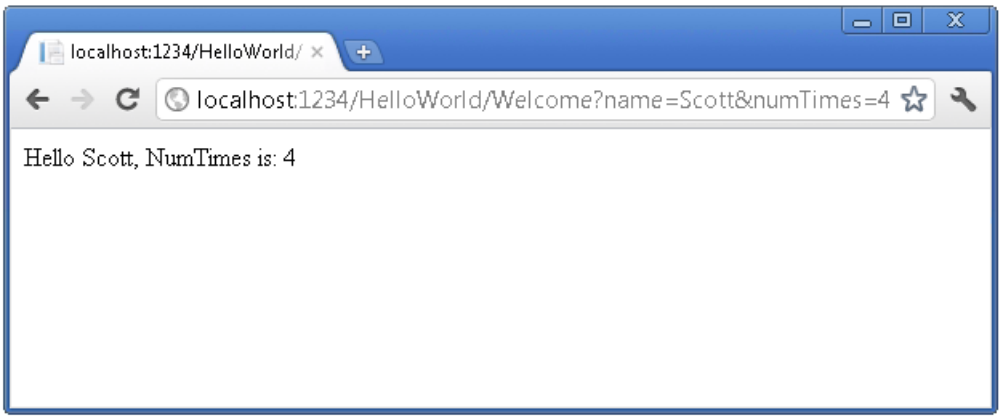
Browse to `http://localhost:xxxx/HelloWorld/Welcome`. The `Welcome` method runs and returns the string "This is the Welcome action method...". The default MVC mapping is `/[Controller]/[ActionName]/[Parameters]`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't used the `[Parameters]` part of the URL yet.



Let's modify the example slightly so that you can pass some parameter information from the URL to the controller (for example, */HelloWorld/Welcome?name=Scott&numtimes=4*). Change your `Welcome` method to include two parameters as shown below. Note that the code uses the C# optional-parameter feature to indicate that the `numTimes` parameter should default to 1 if no value is passed for that parameter.

```
public string Welcome(string name, int numTimes = 1) {
     return HttpUtility.HtmlEncode("Hello " + name + ", NumTimes is: " + numTimes);
}
```

Run your application and browse to the example URL (`http://localhost:xxxx/HelloWorld/Welcome?name=Scott&numtimes=4)` . You can try different values for `name` and `numtimes` in the URL. The ASP.NET MVC model binding system automatically maps the named parameters from the query string in the address bar to parameters in your method.



In both these examples the controller has been doing the "VC" portion of MVC — that is, the view and controller work. The controller is returning HTML directly. Ordinarily you don't want controllers returning HTML directly, since that becomes very cumbersome to code. Instead we'll typically use a separate view template file to help generate the HTML response. Let's look next at how we can do this.

PREVIOUS NEXT

Adding a View

2/20/2020 • 8 minutes to read • Edit Online

by Rick Anderson

> **NOTE**
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

In this section you're going to modify the `HelloWorldController` class to use view template files to cleanly encapsulate the process of generating HTML responses to a client.

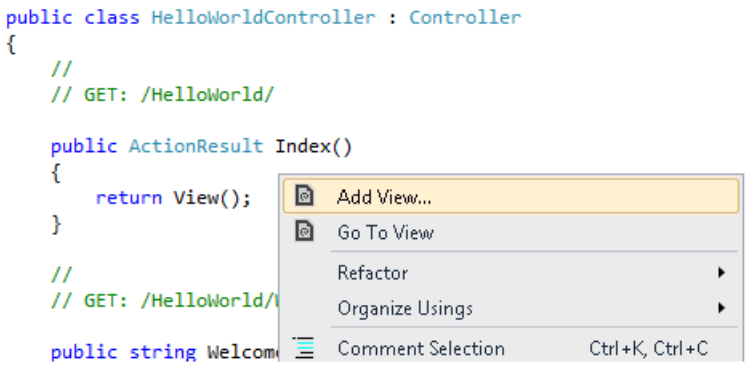
You'll create a view template file using the Razor view engine introduced with ASP.NET MVC 3. Razor-based view templates have a *.cshtml* file extension, and provide an elegant way to create HTML output using C#. Razor minimizes the number of characters and keystrokes required when writing a view template, and enables a fast, fluid coding workflow.

Currently the `Index` method returns a string with a message that is hard-coded in the controller class. Change the `Index` method to return a `View` object, as shown in the following code:

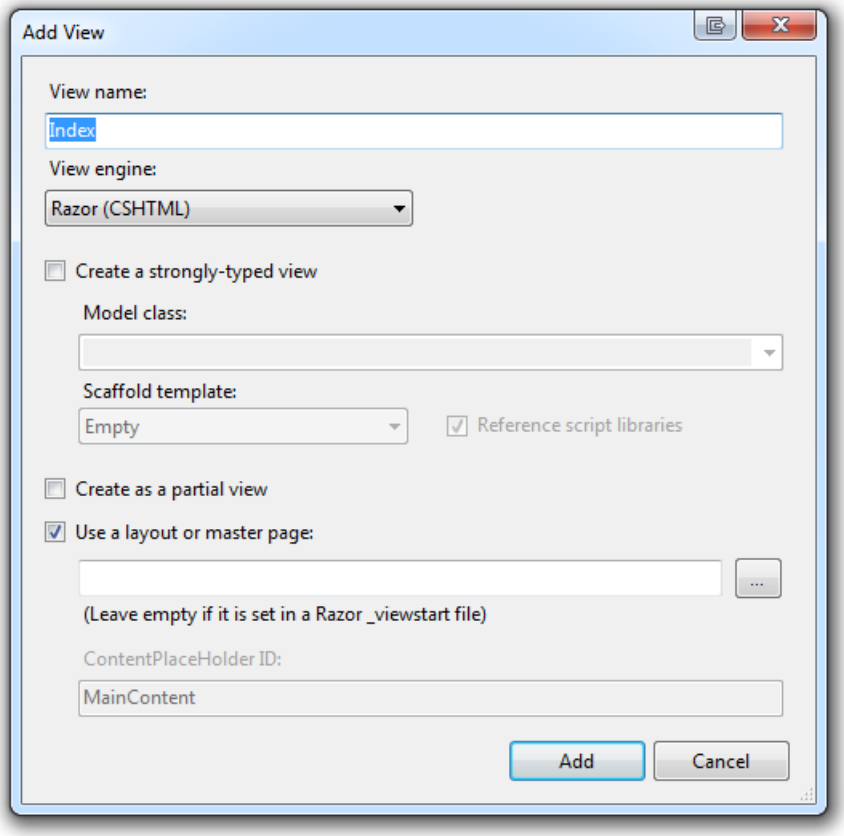
```
public ActionResult Index()
{
    return View();
}
```

The `Index` method above uses a view template to generate an HTML response to the browser. Controller methods (also known as action methods), such as the `Index` method above, generally return an ActionResult (or a class derived from ActionResult), not primitive types like string.

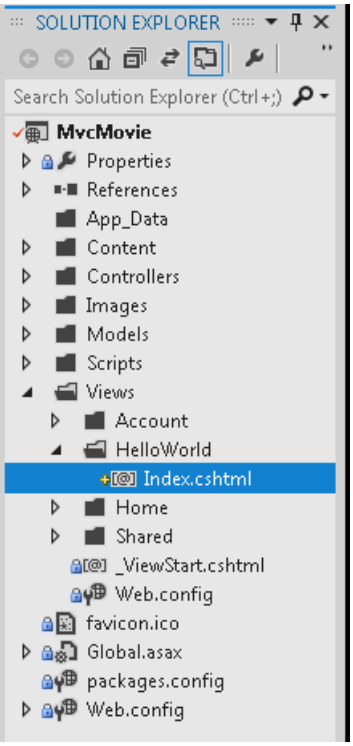
In the project, add a view template that you can use with the `Index` method. To do this, right-click inside the `Index` method and click **Add View**.



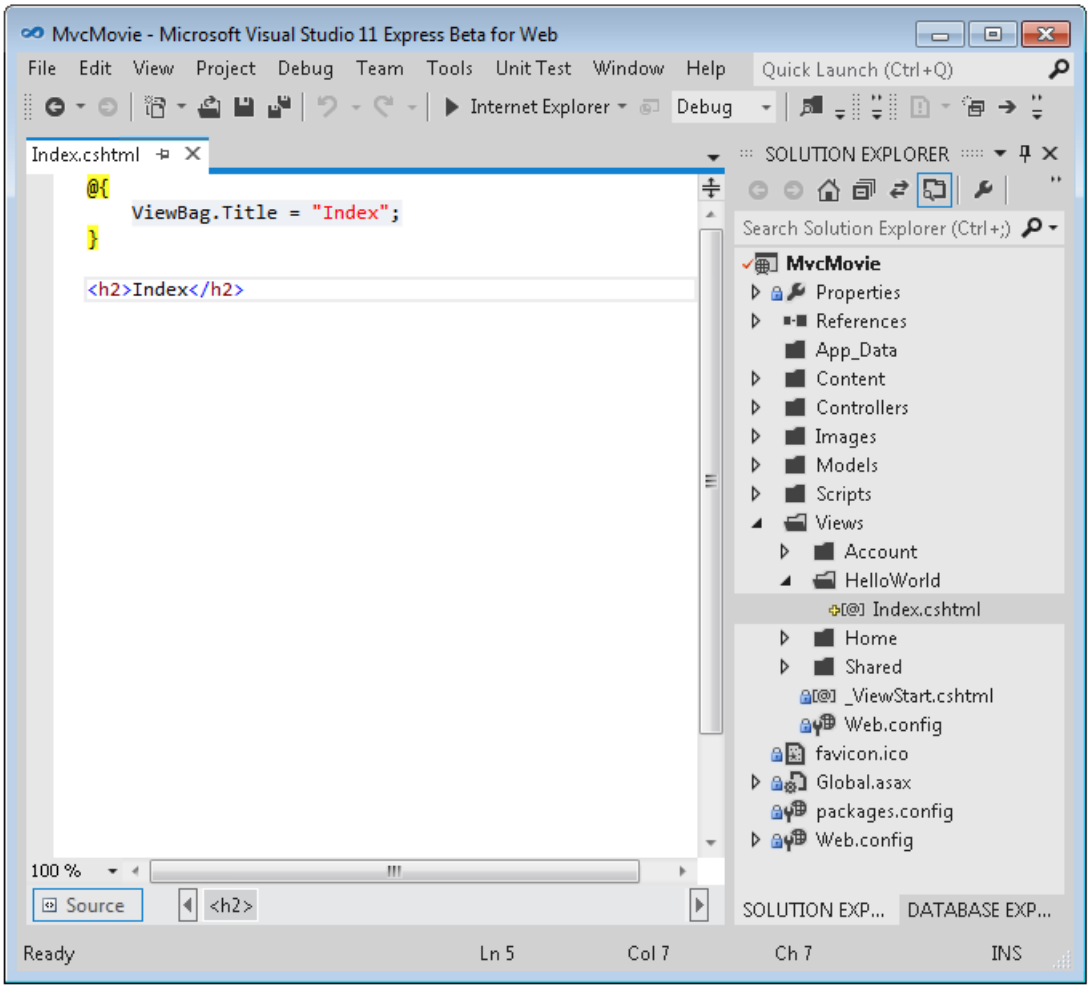
The **Add View** dialog box appears. Leave the defaults the way they are and click the **Add** button:



The *MvcMovie\Views\HelloWorld* folder and the *MvcMovie\Views\HelloWorld\Index.cshtml* file are created. You can see them in **Solution Explorer**:



The following shows the *Index.cshtml* file that was created:



Add the following HTML under the `<h2>` tag.

```
<p>Hello from our View Template!</p>
```

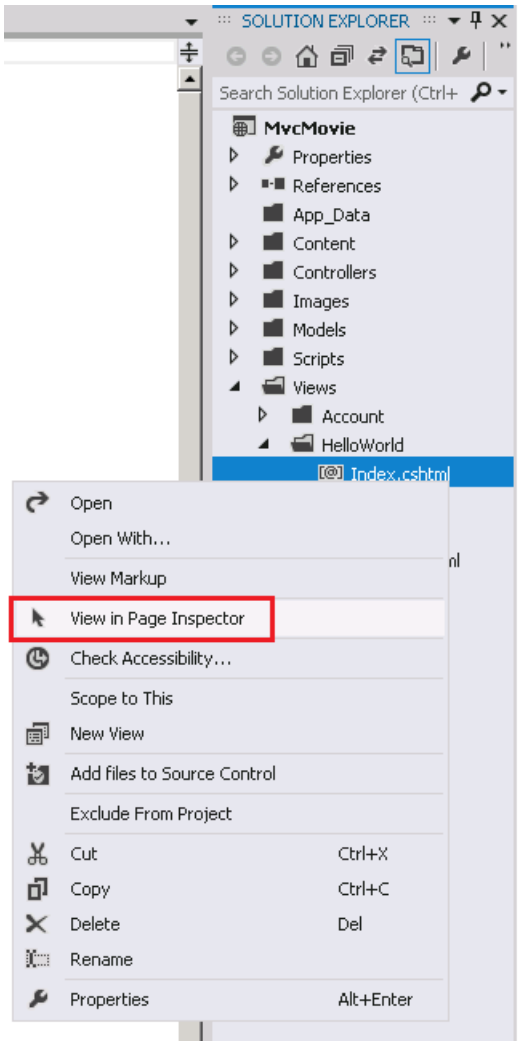
The complete *MvcMovie\Views\HelloWorld\Index.cshtml* file is shown below.

```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

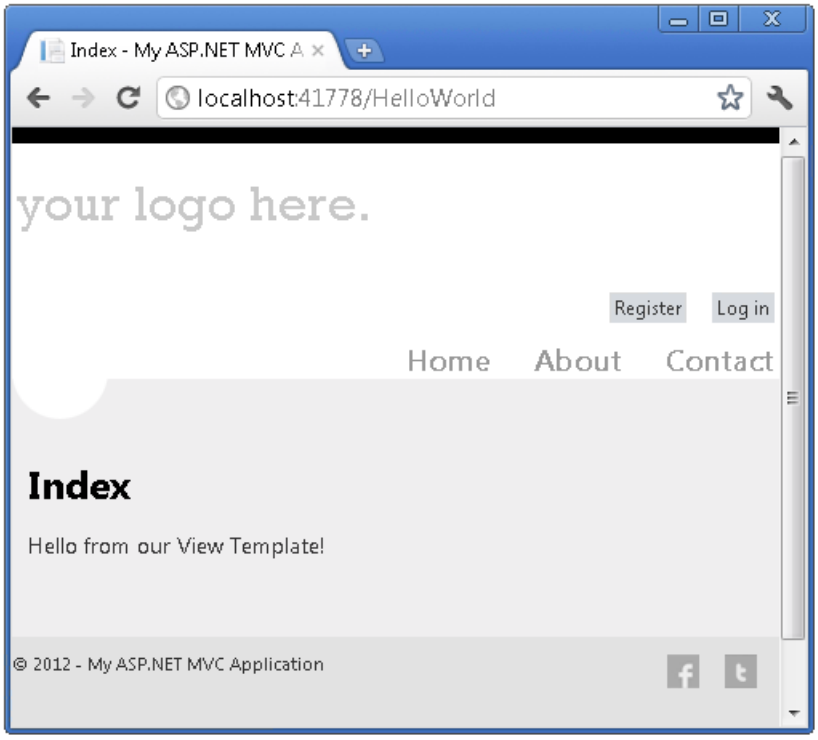
<p>Hello from our View Template!</p>
```

If you are using Visual Studio 2012, in solution explorer, right click the *Index.cshtml* file and select **View in Page Inspector**.



The Page Inspector tutorial has more information about this new tool.

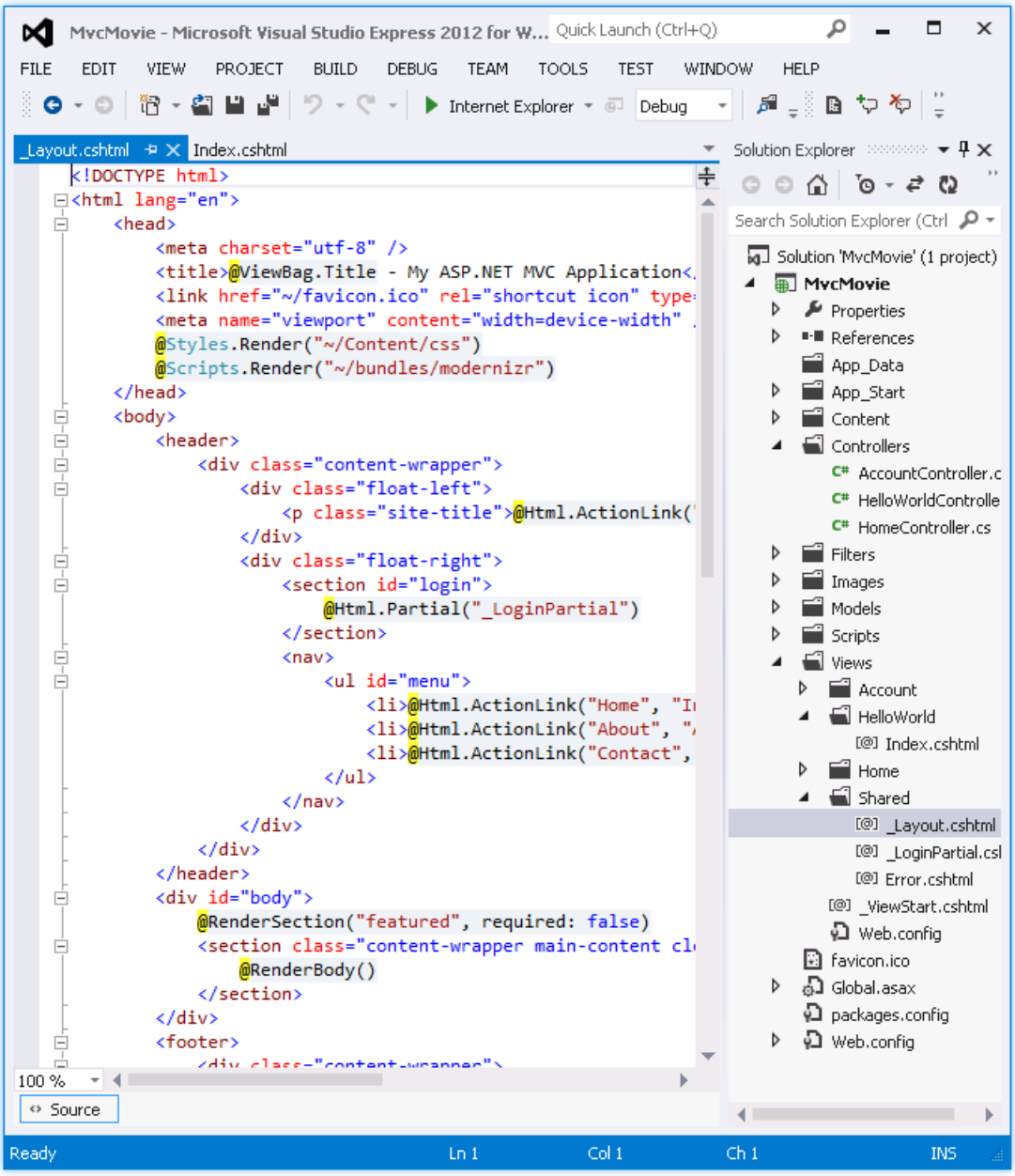
Alternatively, run the application and browse to the `HelloWorld` controller (`http://localhost:xxxx/HelloWorld`). The `Index` method in your controller didn't do much work; it simply ran the statement `return View()`, which specified that the method should use a view template file to render a response to the browser. Because you didn't explicitly specify the name of the view template file to use, ASP.NET MVC defaulted to using the *Index.cshtml* view file in the *\Views\HelloWorld* folder. The image below shows the string "Hello from our View Template!" hard-coded in the view.



Looks pretty good. However, notice that the browser's title bar shows "Index My ASP.NET A" and the big link on the top of the page says "your logo here." Below the "your logo here." link are registration and log in links, and below that links to Home, About and Contact pages. Let's change some of these.

Changing Views and Layout Pages

First, you want to change the "your logo here." title at the top of the page. That text is common to every page. It's actually implemented in only one place in the project, even though it appears on every page in the application. Go to the */Views/Shared* folder in **Solution Explorer** and open the *_Layout.cshtml* file. This file is called a *layout page* and it's the shared "shell" that all other pages use.



Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, "wrapped" in the layout page. For example, if you select the About link, the *Views\Home\About.cshtml* view is rendered inside the `RenderBody` method.

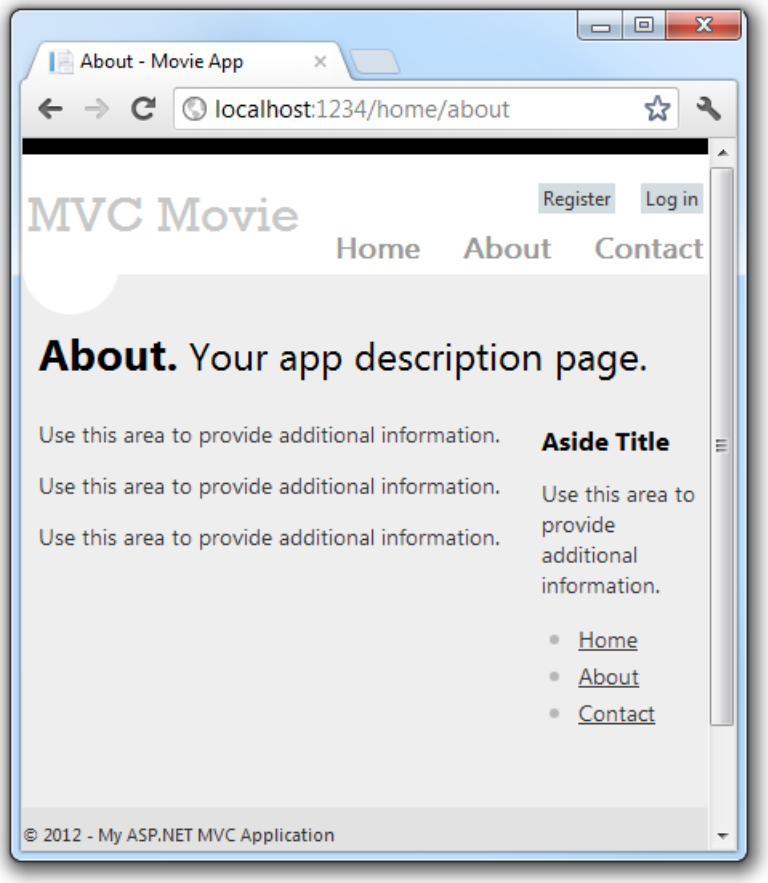
Change the site-title heading in the layout template from "your logo here" to "MVC Movie".

```
<div class="float-left">
    <p class="site-title">@Html.ActionLink("MVC Movie", "Index", "Home")</p>
</div>
```

Replace the contents of the title element with the following markup:

```
<title>@ViewBag.Title - Movie App</title>
```

Run the application and notice that it now says "MVC Movie ". Click the **About** link, and you see how that page shows "MVC Movie", too. We were able to make the change once in the layout template and have all pages on the site reflect the new title.



Now, let's change the title of the Index view.

Open *MvcMovie\Views\HelloWorld\Index.cshtml*. There are two places to make a change: first, the text that appears in the title of the browser, and then in the secondary header (the `<h2>` element). You'll make them slightly different so you can see which bit of code changes which part of the app.

```
@{
    ViewBag.Title = "Movie List";
}

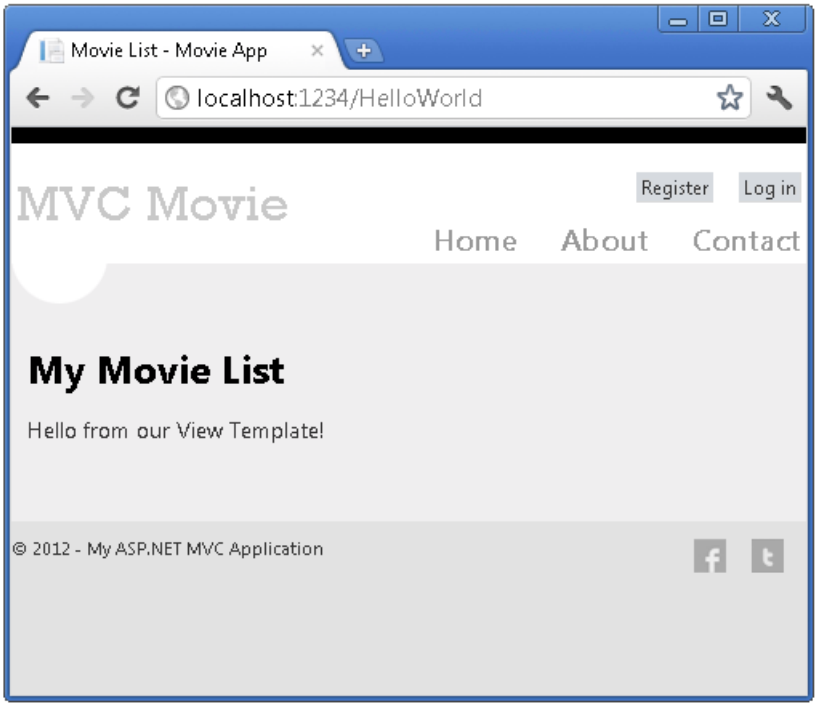
<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

To indicate the HTML title to display, the code above sets a `Title` property of the `ViewBag` object (which is in the *Index.cshtml* view template). If you look back at the source code of the layout template, you'll notice that the template uses this value in the `<title>` element as part of the `<head>` section of the HTML that we modified previously. Using this `ViewBag` approach, you can easily pass other parameters between your view template and your layout file.

Run the application and browse to `http://localhost:xx/HelloWorld`. Notice that the browser title, the primary heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing cached content. Press Ctrl+F5 in your browser to force the response from the server to be loaded.) The browser title is created with the `ViewBag.Title` we set in the *Index.cshtml* view template and the additional "-Movie App" added in the layout file.

Also notice how the content in the *Index.cshtml* view template was merged with the *_Layout.cshtml* view template and a single HTML response was sent to the browser. Layout templates make it really easy to make changes that apply across all of the pages in your application.



Our little bit of "data" (in this case the "Hello from our View Template!" message) is hard-coded, though. The MVC application has a "V" (view) and you've got a "C" (controller), but no "M" (model) yet. Shortly, we'll walk through how create a database and retrieve model data from it.

Passing Data from the Controller to the View

Before we go to a database and talk about models, though, let's first talk about passing information from the controller to a view. Controller classes are invoked in response to an incoming URL request. A controller class is where you write the code that handles the incoming browser requests, retrieves data from a database, and ultimately decides what type of response to send back to the browser. View templates can then be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing whatever data or objects are required in order for a view template to render a response to the browser. A best practice: **A view template should never perform business logic or interact with a database directly**. Instead, a view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep your code clean, testable and more maintainable.

Currently, the `Welcome` action method in the `HelloWorldController` class takes a `name` and a `numTimes` parameter and then outputs the values directly to the browser. Rather than have the controller render this response as a string, let's change the controller to use a view template instead. The view template will generate a dynamic response, which means that you need to pass appropriate bits of data from the controller to the view in order to generate the response. You can do this by having the controller put the dynamic data (parameters) that the view template needs in a `ViewBag` object that the view template can then access.

Return to the *HelloWorldController.cs* file and change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewBag` object. `ViewBag` is a dynamic object, which means you can put whatever you want in to it; the `ViewBag` object has no defined properties until you put something inside it. The ASP.NET MVC model binding system automatically maps the named parameters (`name` and `numTimes`) from the query string in the address bar to parameters in your method. The complete *HelloWorldController.cs* file looks like this:

```
using System.Web;
using System.Web.Mvc;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

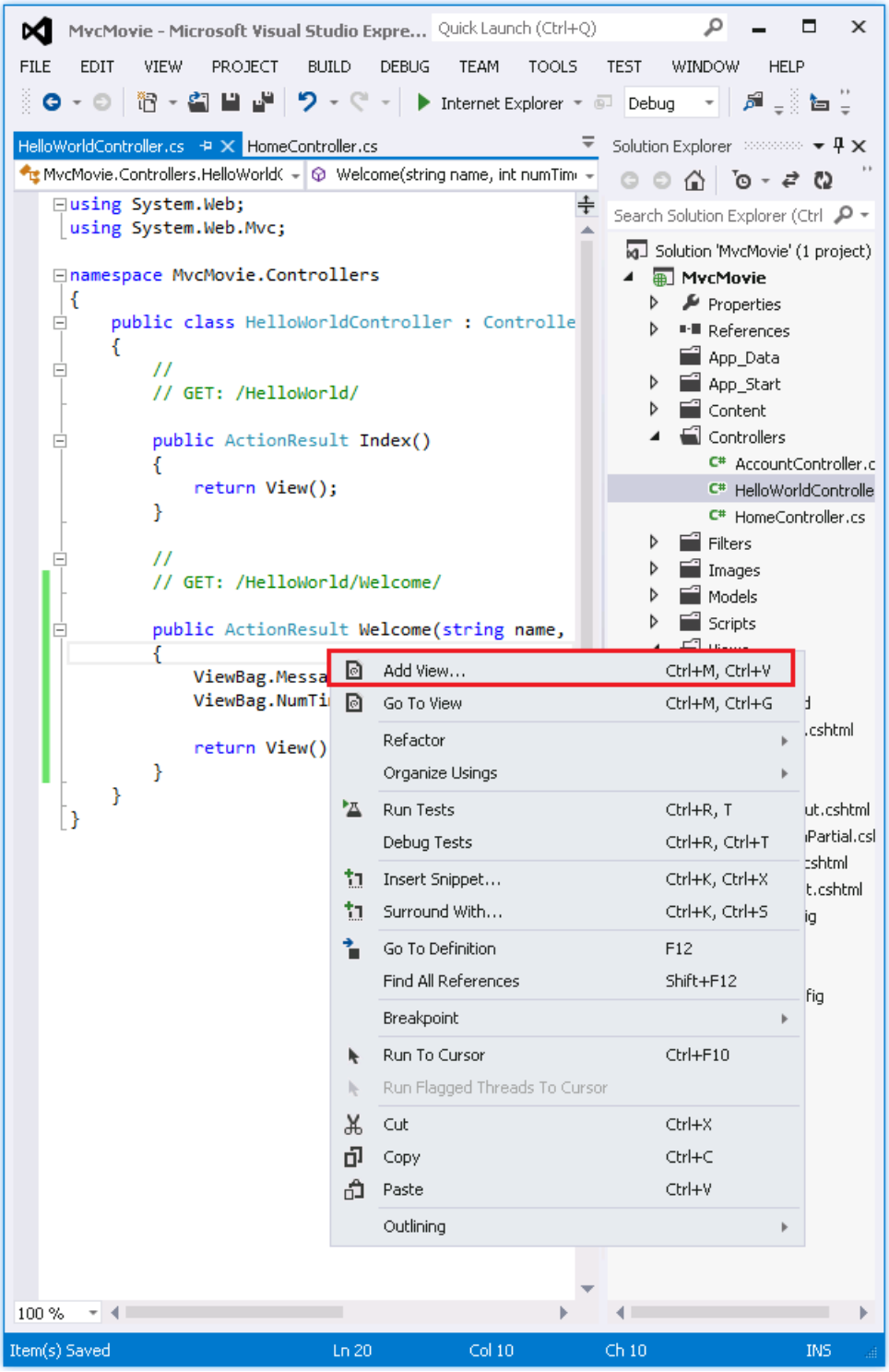
        public ActionResult Welcome(string name, int numTimes = 1)
        {
            ViewBag.Message = "Hello " + name;
            ViewBag.NumTimes = numTimes;

            return View();
        }
    }
}
```

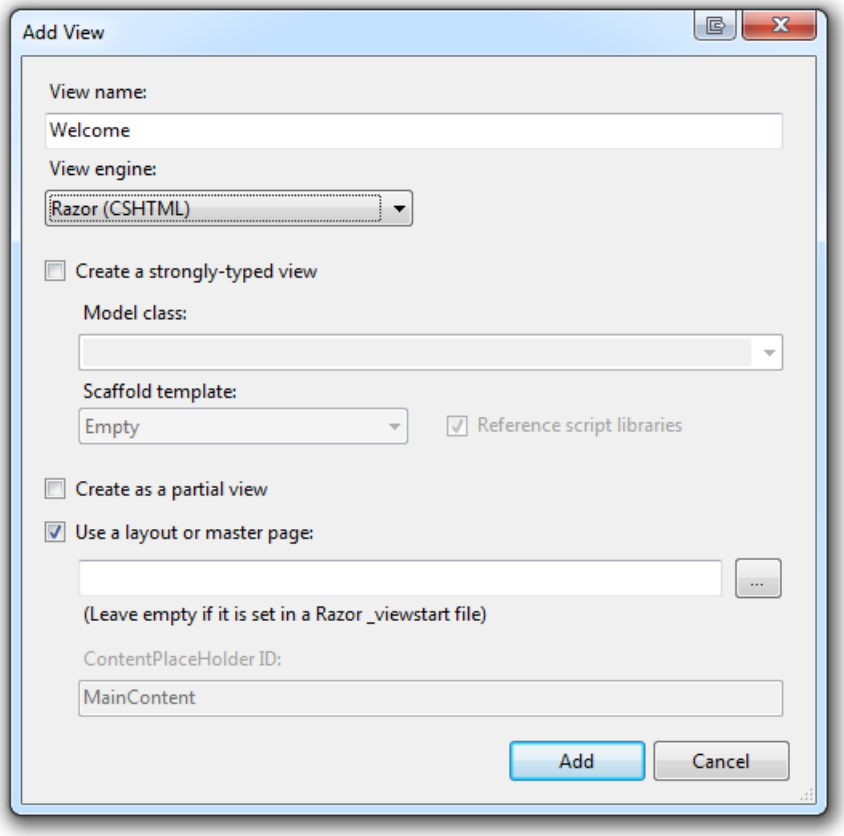
Now the `ViewBag` object contains data that will be passed to the view automatically.

Next, you need a Welcome view template! In the **Build** menu, select **Build MvcMovie** to make sure the project is compiled.

Then right-click inside the `Welcome` method and click **Add View**.



Here's what the **Add View** dialog box looks like:



Click **Add**, and then add the following code under the `<h2>` element in the new *Welcome.cshtml* file. You'll create a loop that says "Hello" as many times as the user says it should. The complete *Welcome.cshtml* file is shown below.

```
@{
    ViewBag.Title = "Welcome";
}

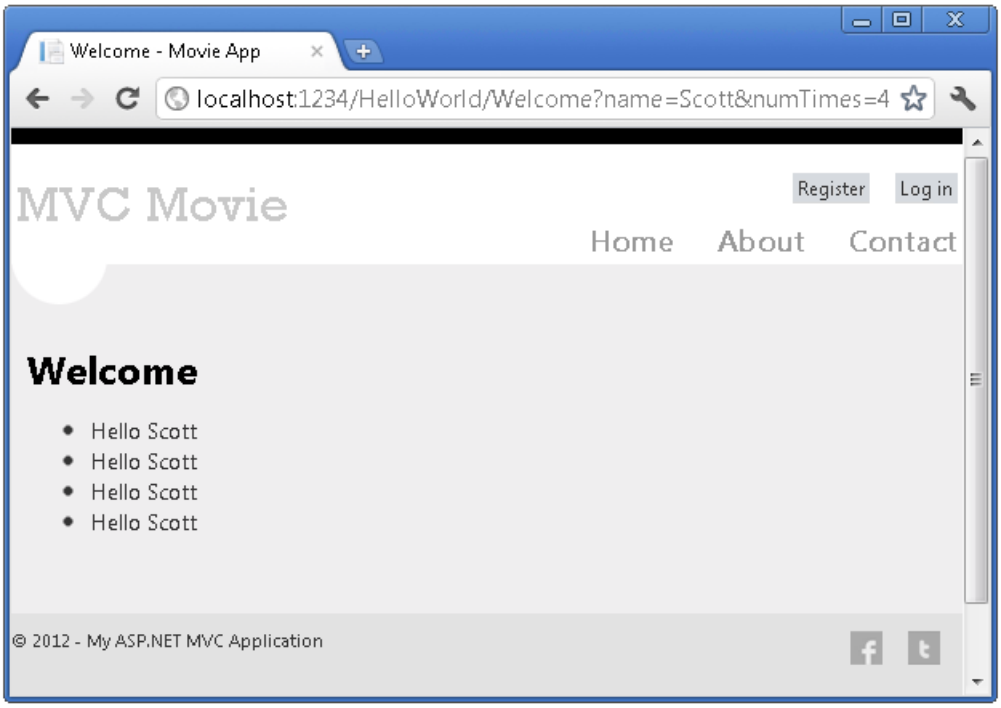
<h2>Welcome</h2>

<ul>
   @for (int i=0; i < ViewBag.NumTimes; i++) {
      <li>@ViewBag.Message</li>
   }
</ul>
```

Run the application and browse to the following URL:

`http://localhost:xx/HelloWorld/Welcome?name=Scott&numtimes=4`

Now data is taken from the URL and passed to the controller using the model binder. The controller packages the data into a `ViewBag` object and passes that object to the view. The view then displays the data as HTML to the user.



In the sample above, we used a `ViewBag` object to pass data from the controller to a view. Latter in the tutorial, we will use a view model to pass data from a controller to a view. The view model approach to passing data is generally much preferred over the view bag approach. See the blog entry Dynamic V Strongly Typed Views for more information.

Well, that was a kind of an "M" for model, but not the database kind. Let's take what we've learned and create a database of movies.

PREVIOUS NEXT

Adding a Model

2/20/2020 • 2 minutes to read • Edit Online

by Rick Anderson

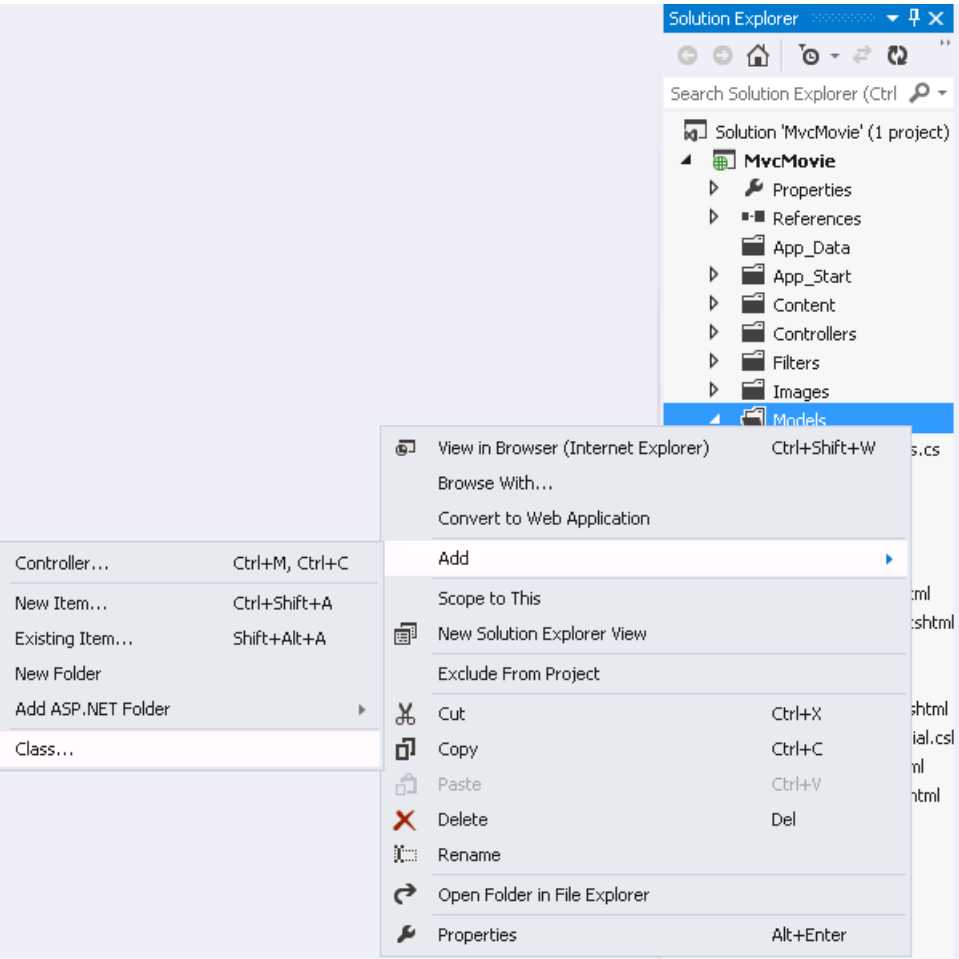
> **NOTE**
>
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

In this section you'll add some classes for managing movies in a database. These classes will be the "model" part of the ASP.NET MVC application.

You'll use a .NET Framework data-access technology known as the Entity Framework to define and work with these model classes. The Entity Framework (often referred to as EF) supports a development paradigm called *Code First*. Code First allows you to create model objects by writing simple classes. (These are also known as POCO classes, from "plain-old CLR objects.") You can then have the database created on the fly from your classes, which enables a very clean and rapid development workflow.

Adding Model Classes

In **Solution Explorer**, right click the *Models* folder, select **Add**, and then select **Class**.



Enter the *class* name "Movie".

Add the following five properties to the `Movie` class:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
}
```

We'll use the `Movie` class to represent movies in a database. Each instance of a `Movie` object will correspond to a row within a database table, and each property of the `Movie` class will map to a column in the table.

In the same file, add the following `MovieDBContext` class:

```
public class MovieDBContext : DbContext
{
    public DbSet<Movie> Movies { get; set; }
}
```

The `MovieDBContext` class represents the Entity Framework movie database context, which handles fetching, storing, and updating `Movie` class instances in a database. The `MovieDBContext` derives from the `DbContext` base class provided by the Entity Framework.

In order to be able to reference `DbContext` and `DbSet`, you need to add the following `using` statement at the top of the file:

```
using System.Data.Entity;
```

The complete *Movie.cs* file is shown below. (Several using statements that are not needed have been removed.)

```
using System;
using System.Data.Entity;

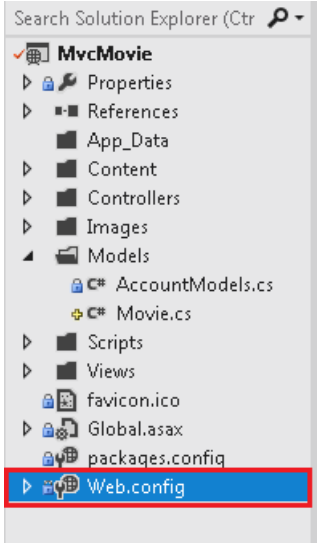
namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }

    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

Creating a Connection String and Working with SQL Server LocalDB

The `MovieDBContext` class you created handles the task of connecting to the database and mapping `Movie` objects to database records. One question you might ask, though, is how to specify which database it will connect to. You'll do that by adding connection information in the *Web.config* file of the application.

Open the application root *Web.config* file. (Not the *Web.config* file in the *Views* folder.) Open the *Web.config* file outlined in red.



Add the following connection string to the `<connectionStrings>` element in the *Web.config* file.

```
<add name="MovieDBContext"
   connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\Movies.mdf;Integrated
Security=True"
   providerName="System.Data.SqlClient"
/>
```

The following example shows a portion of the *Web.config* file with the new connection string added:

```
<connectionStrings>
  <add name="DefaultConnection"
       connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=aspnet-MvcMovie-2012213181139;Integrated
Security=true"
       providerName="System.Data.SqlClient"
  />
  <add name="MovieDBContext"
       connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\Movies.mdf;Integrated
Security=True"
       providerName="System.Data.SqlClient"
  />
</connectionStrings>
```

This small amount of code and XML is everything you need to write in order to represent and store the movie data in a database.

Next, you'll build a new `MoviesController` class that you can use to display the movie data and allow users to create new movie listings.

PREVIOUS NEXT

Accessing Your Model's Data from a Controller

2/20/2020 • 6 minutes to read • Edit Online

by Rick Anderson

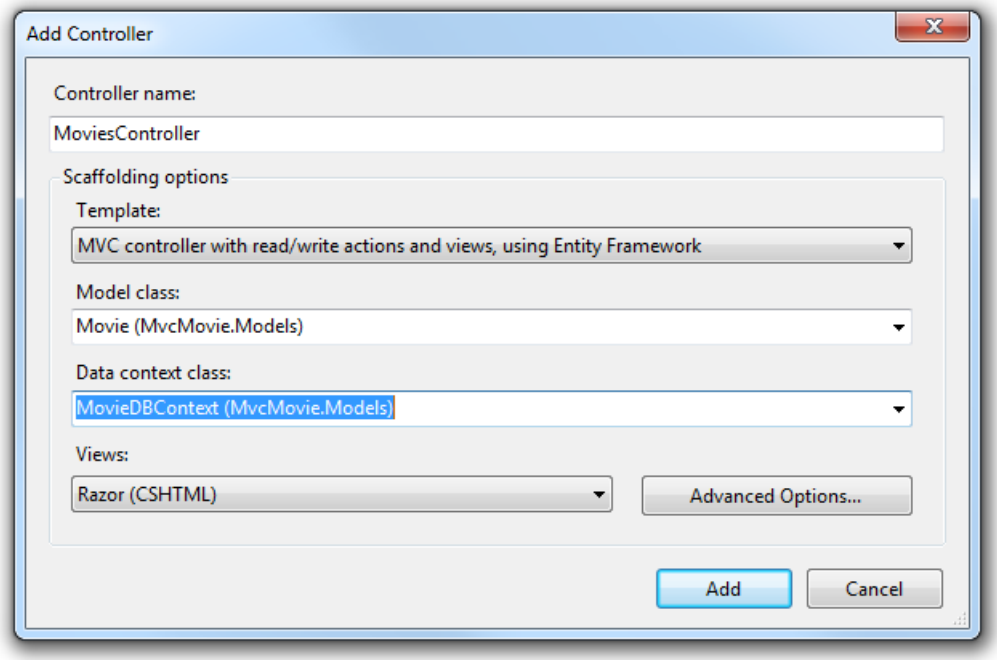
> **NOTE**
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

In this section, you'll create a new `MoviesController` class and write code that retrieves the movie data and displays it in the browser using a view template.

Build the application before going on to the next step.

Right-click the *Controllers* folder and create a new `MoviesController` controller. The options below will not appear until you build your application. Select the following options:

- Controller name: **MoviesController**. (This is the default.)
- Template: **MVC Controller with read/write actions and views, using Entity Framework**.
- Model class: **Movie (MvcMovie.Models)**.
- Data context class: **MovieDBContext (MvcMovie.Models)**.
- Views: **Razor (CSHTML)**. (The default.)



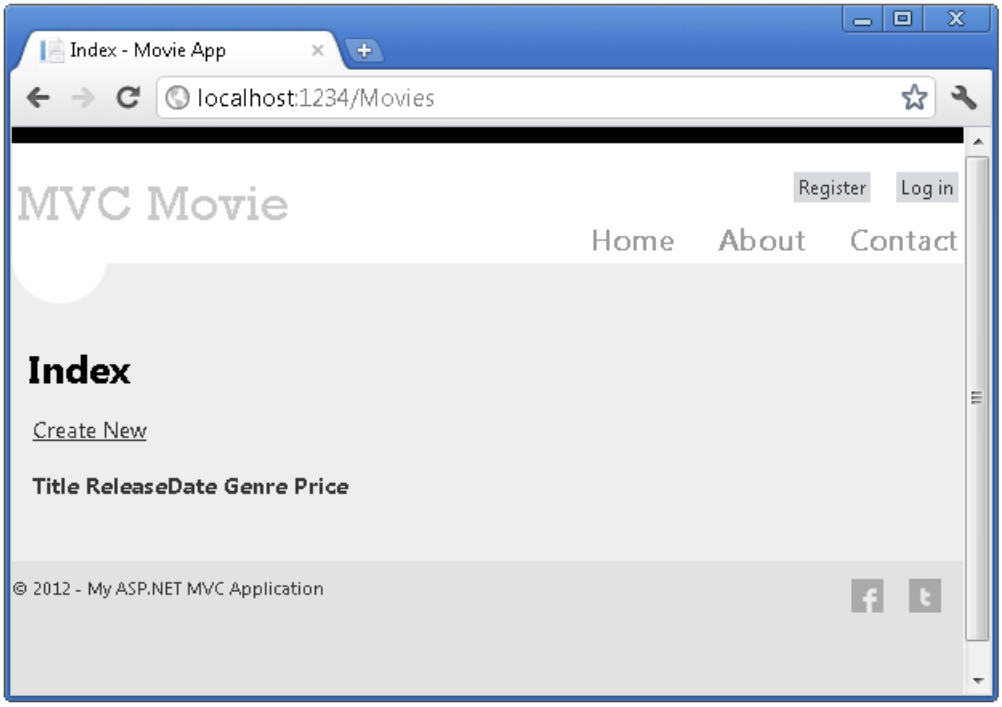
Click **Add**. Visual Studio Express creates the following files and folders:

- *A MoviesController.cs* file in the project's *Controllers* folder.
- A *Movies* folder in the project's *Views* folder.
- *Create.cshtml, Delete.cshtml, Details.cshtml, Edit.cshtml*, and *Index.cshtml* in the new *Views\Movies* folder.

ASP.NET MVC 4 automatically created the CRUD (create, read, update, and delete) action methods and views for you (the automatic creation of CRUD action methods and views is known as scaffolding). You now have a fully

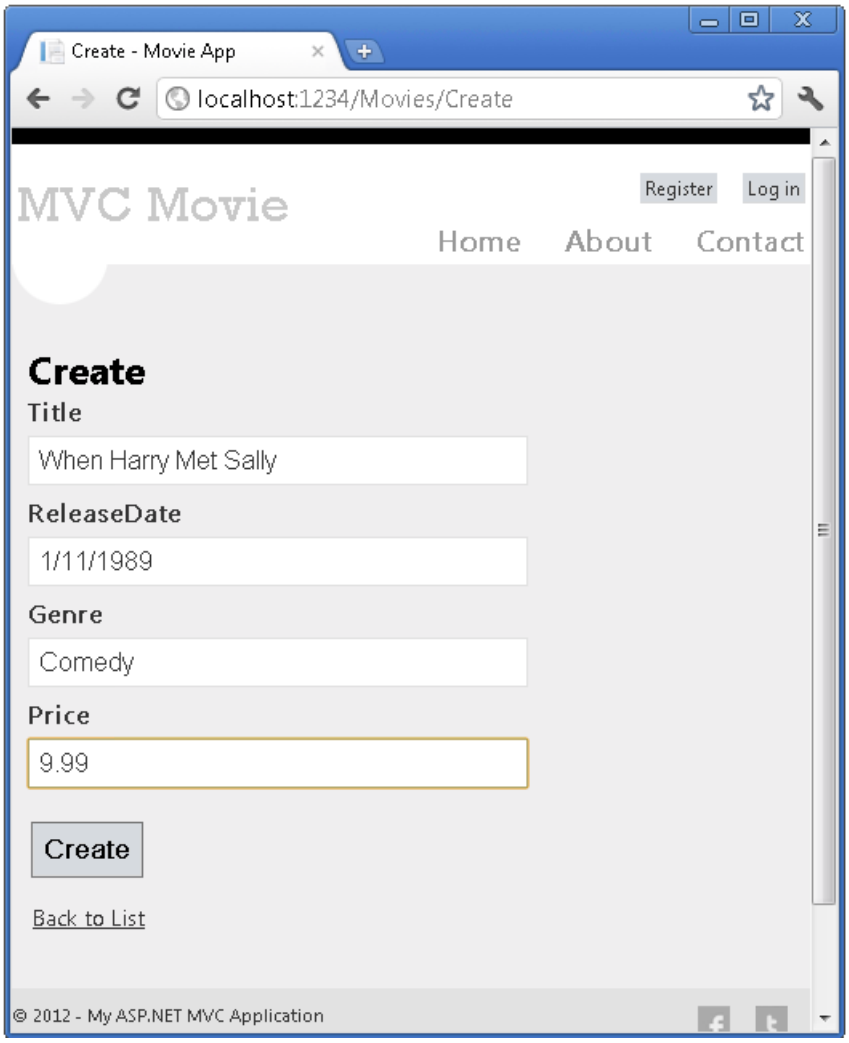
functional web application that lets you create, list, edit, and delete movie entries.

Run the application and browse to the `Movies` controller by appending */Movies* to the URL in the address bar of your browser. Because the application is relying on the default routing (defined in the *Global.asax* file), the browser request `http://localhost:xxxxx/Movies` is routed to the default `Index` action method of the `Movies` controller. In other words, the browser request `http://localhost:xxxxx/Movies` is effectively the same as the browser request `http://localhost:xxxxx/Movies/Index`. The result is an empty list of movies, because you haven't added any yet.

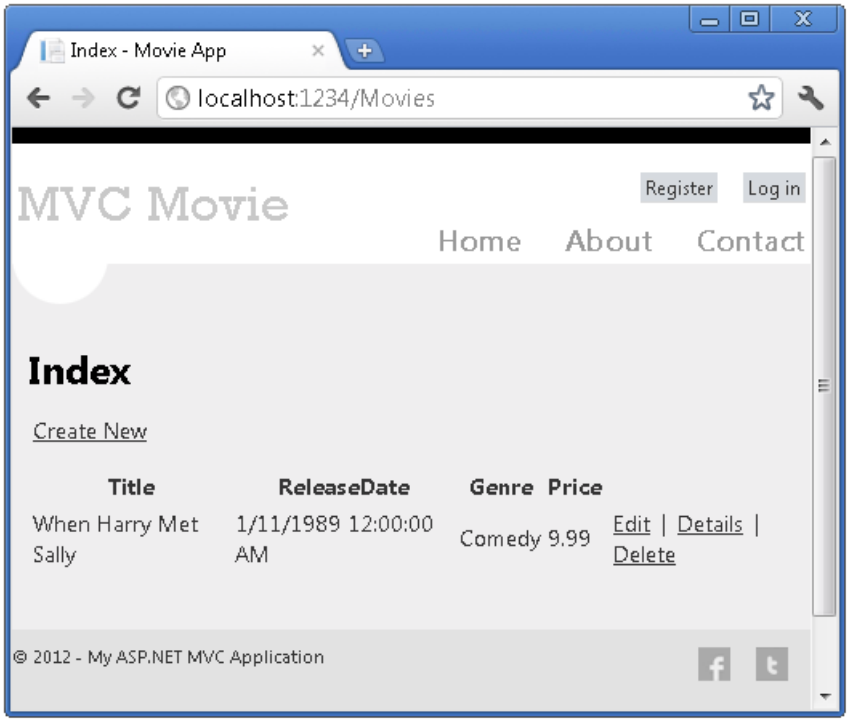


Creating a Movie

Select the **Create New** link. Enter some details about a movie and then click the **Create** button.



Clicking the **Create** button causes the form to be posted to the server, where the movie information is saved in the database. You're then redirected to the */Movies* URL, where you can see the newly created movie in the listing.



Create a couple more movie entries. Try the **Edit**, **Details**, and **Delete** links, which are all functional.

Examining the Generated Code

Open the *Controllers\MoviesController.cs* file and examine the generated `Index` method. A portion of the movie controller with the `Index` method is shown below.

```
public class MoviesController : Controller
{
    private MovieDBContext db = new MovieDBContext();

    //
    // GET: /Movies/

    public ActionResult Index()
    {
        return View(db.Movies.ToList());
    }
```

The following line from the `MoviesController` class instantiates a movie database context, as described previously. You can use the movie database context to query, edit, and delete movies.

```
private MovieDBContext db = new MovieDBContext();
```

A request to the `Movies` controller returns all the entries in the `Movies` table of the movie database and then passes the results to the `Index` view.

Strongly Typed Models and the @model Keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view template using the `ViewBag` object. The `ViewBag` is a dynamic object that provides a convenient late-bound way to pass information to a view.

ASP.NET MVC also provides the ability to pass strongly typed data or objects to a view template. This strongly typed approach enables better compile-time checking of your code and richer IntelliSense in the Visual Studio editor. The scaffolding mechanism in Visual Studio used this approach with the `MoviesController` class and view templates when it created the methods and views.

In the *Controllers\MoviesController.cs* file examine the generated `Details` method. A portion of the movie controller with the `Details` method is shown below.

```
public ActionResult Details(int id = 0)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}
```

If a `Movie` is found, an instance of the `Movie` model is passed to the Details view. Examine the contents of the *Views\Movies\Details.cshtml* file.

By including a `@model` statement at the top of the view template file, you can specify the type of object that the view expects. When you created the movie controller, Visual Studio automatically included the following `@model` statement at the top of the *Details.cshtml* file:

```
@model MvcMovie.Models.Movie
```

This `@model` directive allows you to access the movie that the controller passed to the view by using a `Model`

object that's strongly typed. For example, in the *Details.cshtml* template, the code passes each movie field to the `DisplayNameFor` and DisplayFor HTML Helpers with the strongly typed `Model` object. The Create and Edit methods and view templates also pass a movie model object.

Examine the *Index.cshtml* view template and the `Index` method in the *MoviesController.cs* file. Notice how the code creates a `List` object when it calls the `View` helper method in the `Index` action method. The code then passes this `Movies` list from the controller to the view:

```
public ActionResult Index()
{
    return View(db.Movies.ToList());
}
```

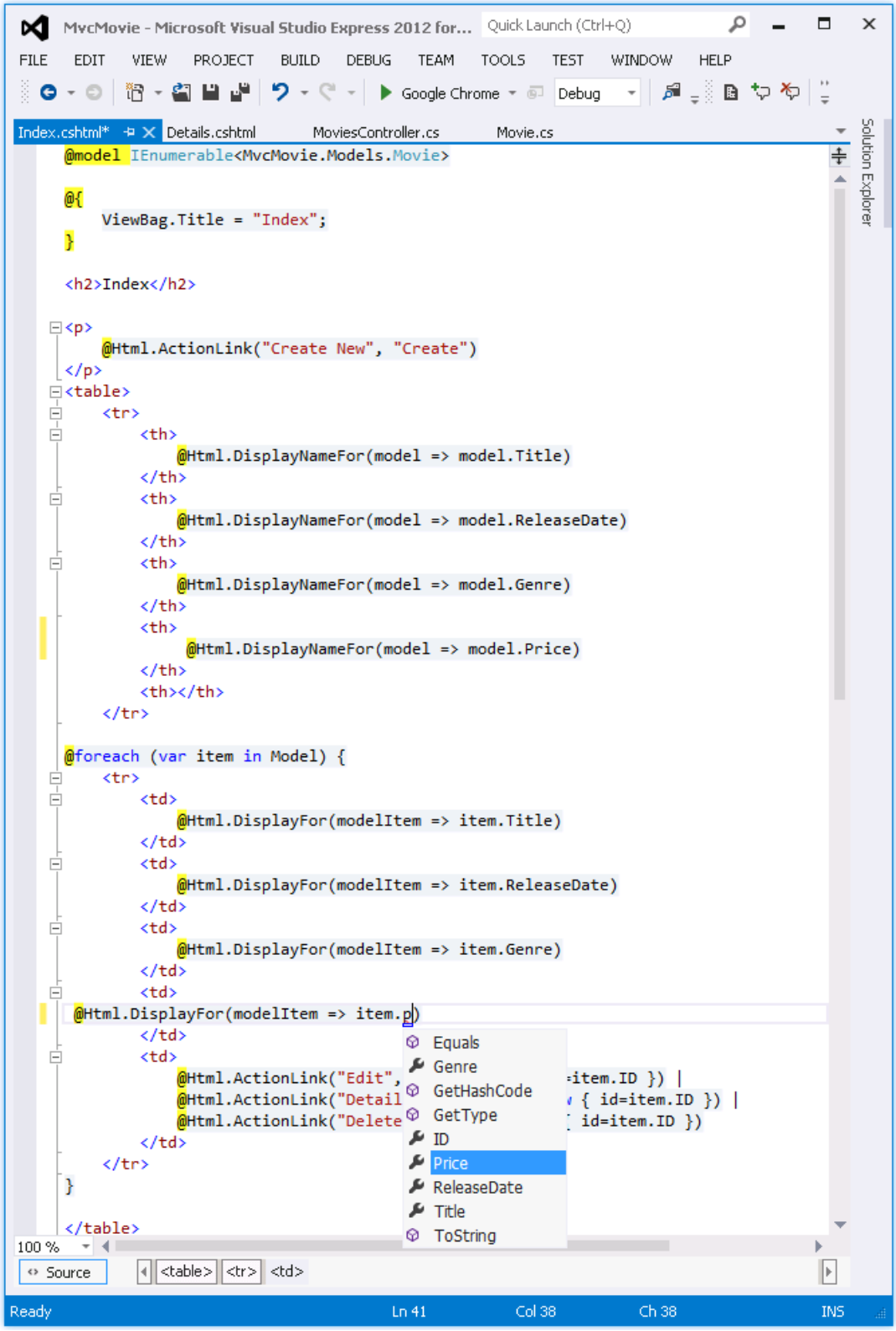
When you created the movie controller, Visual Studio Express automatically included the following `@model` statement at the top of the *Index.cshtml* file:

```
@model IEnumerable<MvcMovie.Models.Movie>
```

This `@model` directive allows you to access the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Index.cshtml* template, the code loops through the movies by doing a `foreach` statement over the strongly typed `Model` object:

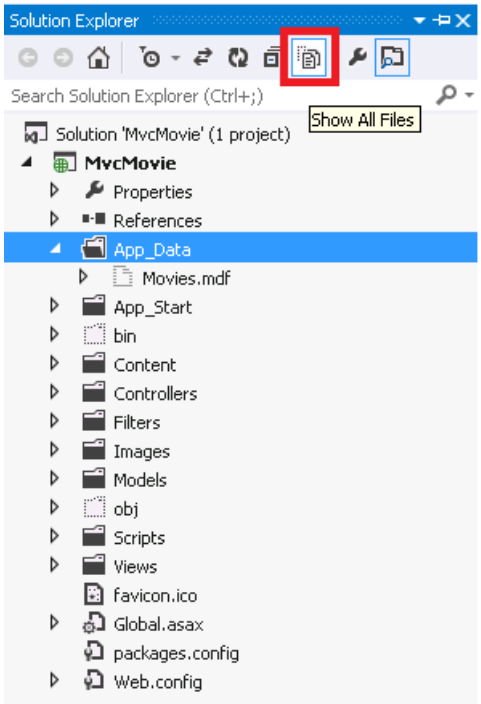
```
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
         <th>
            @Html.DisplayFor(modelItem => item.Rating)
        </th>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", { id=item.ID })  |
            @Html.ActionLink("Delete", "Delete", { id=item.ID })
        </td>
    </tr>
}
```

Because the `Model` object is strongly typed (as an `IEnumerable<Movie>` object), each `item` object in the loop is typed as `Movie`. Among other benefits, this means that you get compile-time checking of the code and full IntelliSense support in the code editor:

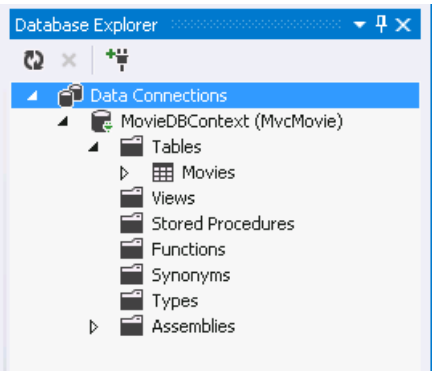


Working with SQL Server LocalDB

Entity Framework Code First detected that the database connection string that was provided pointed to a `Movies` database that didn't exist yet, so Code First created the database automatically. You can verify that it's been created by looking in the *App_Data* folder. If you don't see the *Movies.mdf* file, click the **Show All Files** button in the **Solution Explorer** toolbar, click the **Refresh** button, and then expand the *App_Data* folder.



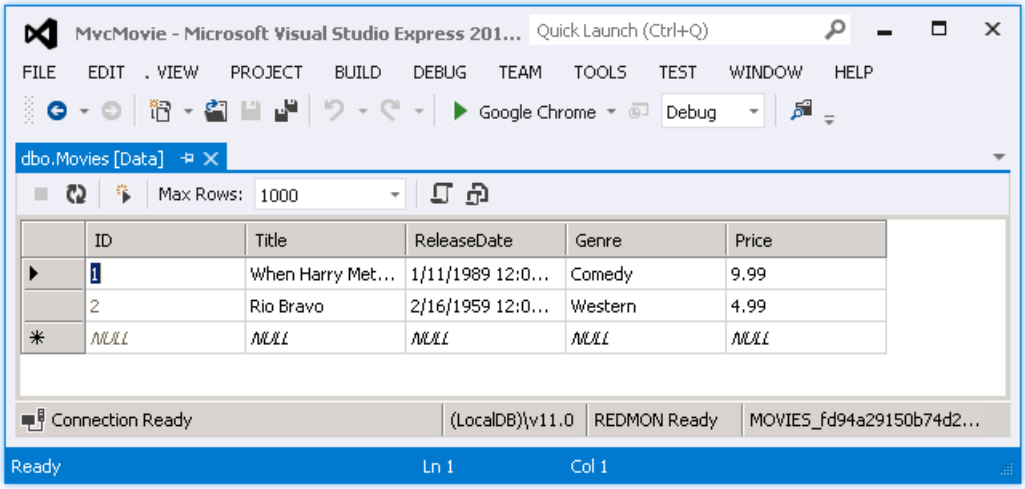
Double-click *Movies.mdf* to open **DATABASE EXPLORER**, then expand the **Tables** folder to see the Movies table.



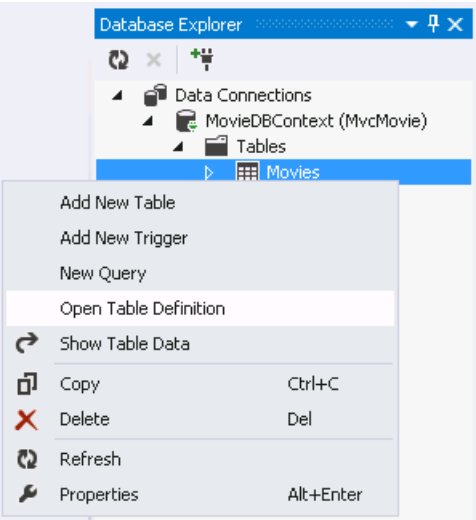
> **NOTE**
>
> If the database explorer doesn't appear, from the **TOOLS** menu, select **Connect to Database**, then cancel the **Choose Data Source** dialog. This will force open the database explorer.

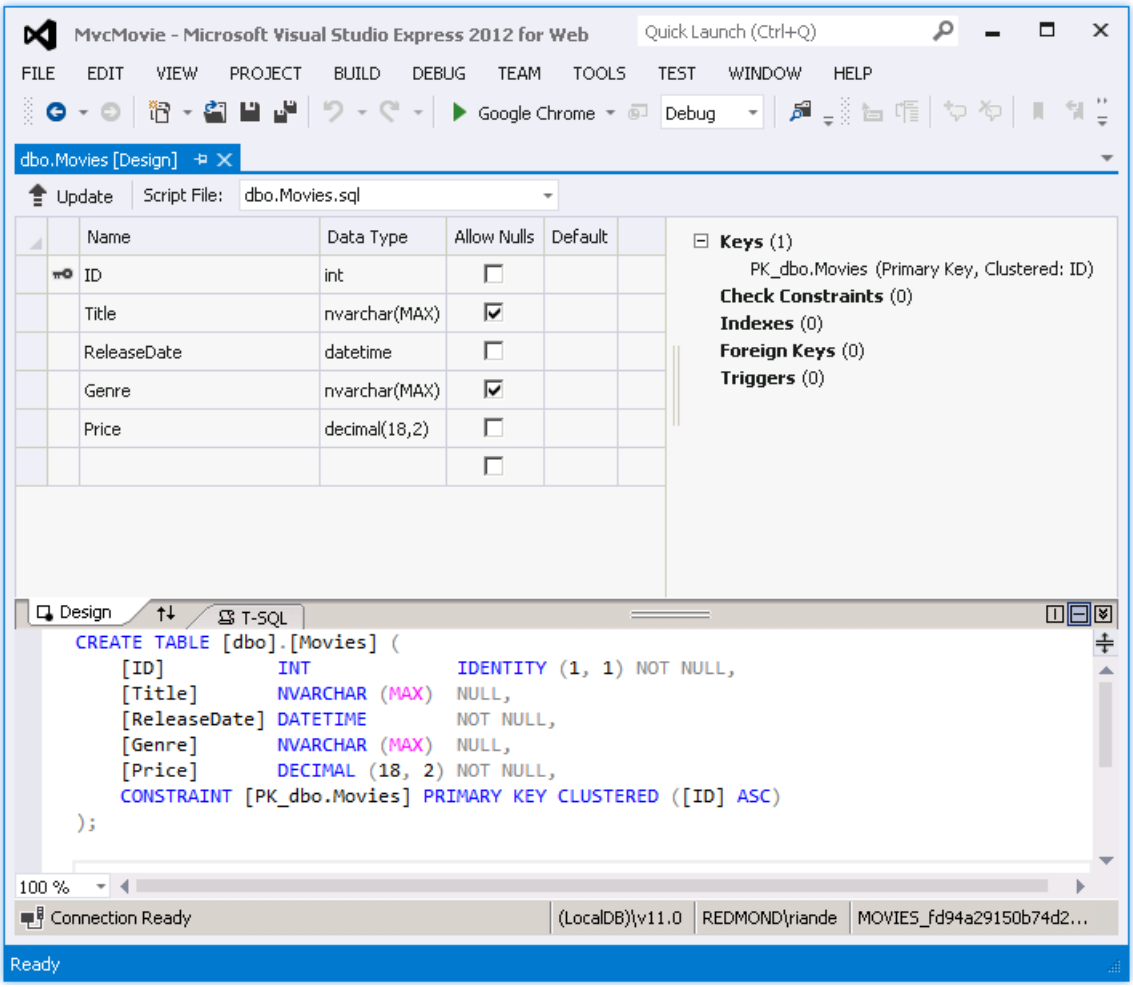
> **NOTE**
>
> If you are using VWD or Visual Studio 2010 and get an error similar to any of the following following:
>
> - The database 'C:\Webs\MVC4\MVCMOVIE\MVCMOVIE\APP_DATA\MOVIES.MDF' cannot be opened because it is version 706. This server supports version 655 and earlier. A downgrade path is not supported.
> - "InvalidOperation Exception was unhandled by user code" The supplied SqlConnection does not specify an initial catalog.
>
> You need to install the SQL Server Data Tools and LocalDB. Verify the `MovieDBContext` connection string specified on the previous page.

Right-click the `Movies` table and select **Show Table Data** to see the data you created.



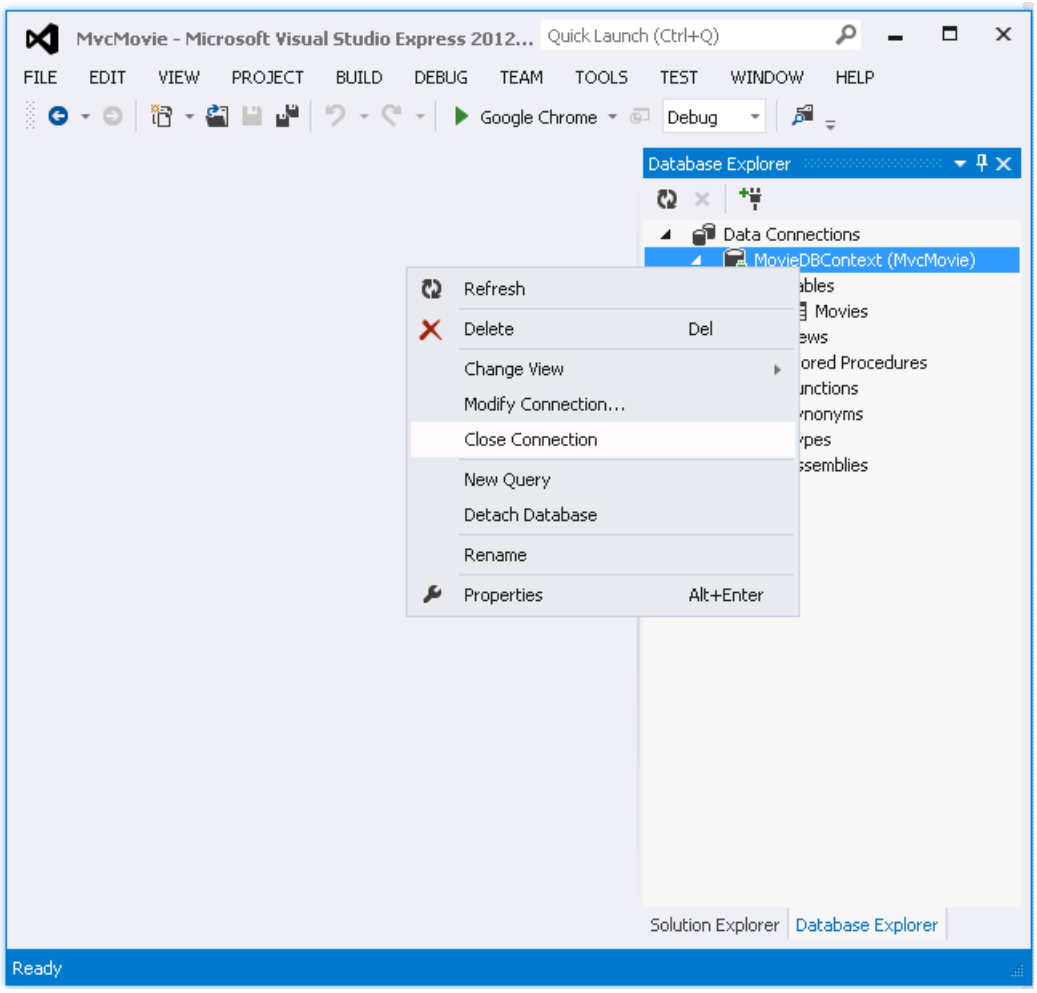
Right-click the `Movies` table and select **Open Table Definition** to see the table structure that Entity Framework Code First created for you.





Notice how the schema of the `Movies` table maps to the `Movie` class you created earlier. Entity Framework Code First automatically created this schema for you based on your `Movie` class.

When you're finished, close the connection by right clicking *MovieDBContext* and selecting **Close Connection**. (If you don't close the connection, you might get an error the next time you run the project).



You now have the database and a simple listing page to display content from it. In the next tutorial, we'll examine the rest of the scaffolded code and add a `SearchIndex` method and a `SearchIndex` view that lets you search for movies in this database.

PREVIOUS NEXT

Examining the Edit Methods and Edit View

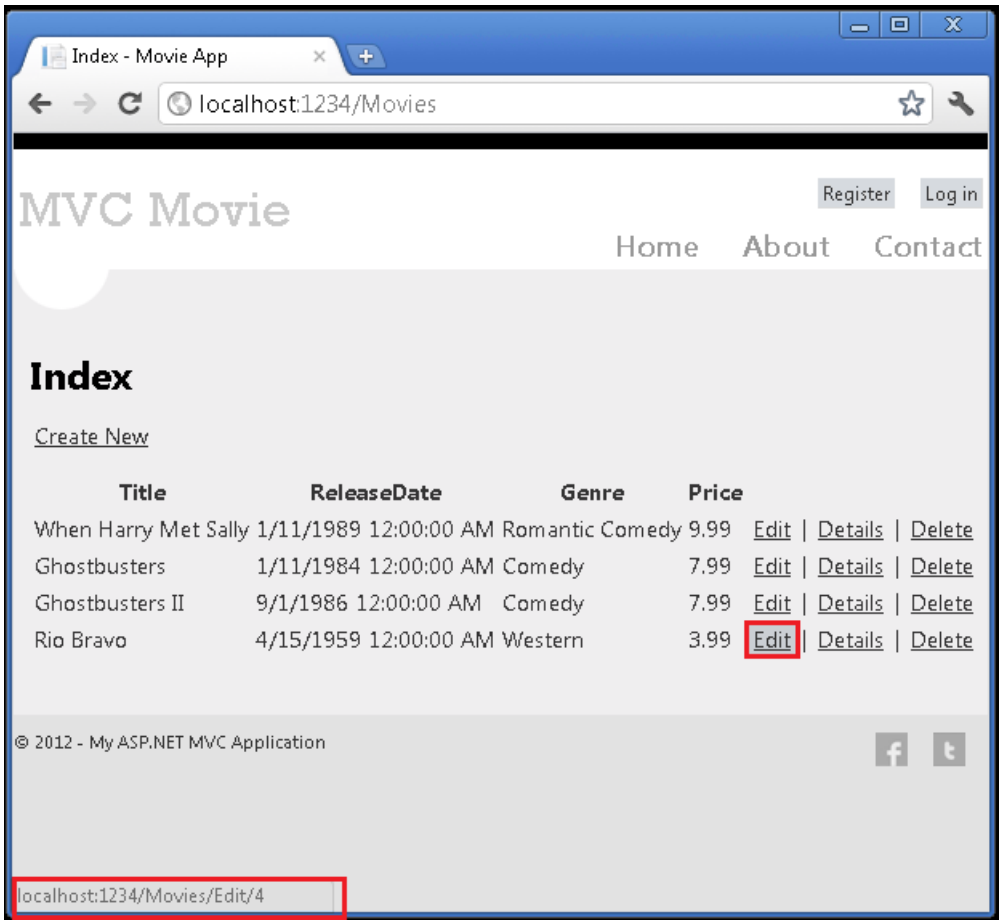
9/16/2020 • 14 minutes to read • Edit Online

by Rick Anderson

> **NOTE**
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

In this section, you'll examine the generated action methods and views for the movie controller. Then you'll add a custom search page.

Run the application and browse to the `Movies` controller by appending */Movies* to the URL in the address bar of your browser. Hold the mouse pointer over an **Edit** link to see the URL that it links to.



The **Edit** link was generated by the `Html.ActionLink` method in the *Views\Movies\Index.cshtml* view:

```
@Html.ActionLink("Edit", "Edit", new { id=item.ID })
```

```
<td>
    @Html.ActionLink("Edit Me", "Edit", new { id=item.ID }) |
    @Html.
    @Html.
```

(extension) MvcHtmlString HtmlHelper.ActionLink(string linkText, string actionName, object routeValues)
Returns an anchor element (a element) that contains the virtual path of the specified action.

Exceptions:
System.ArgumentException

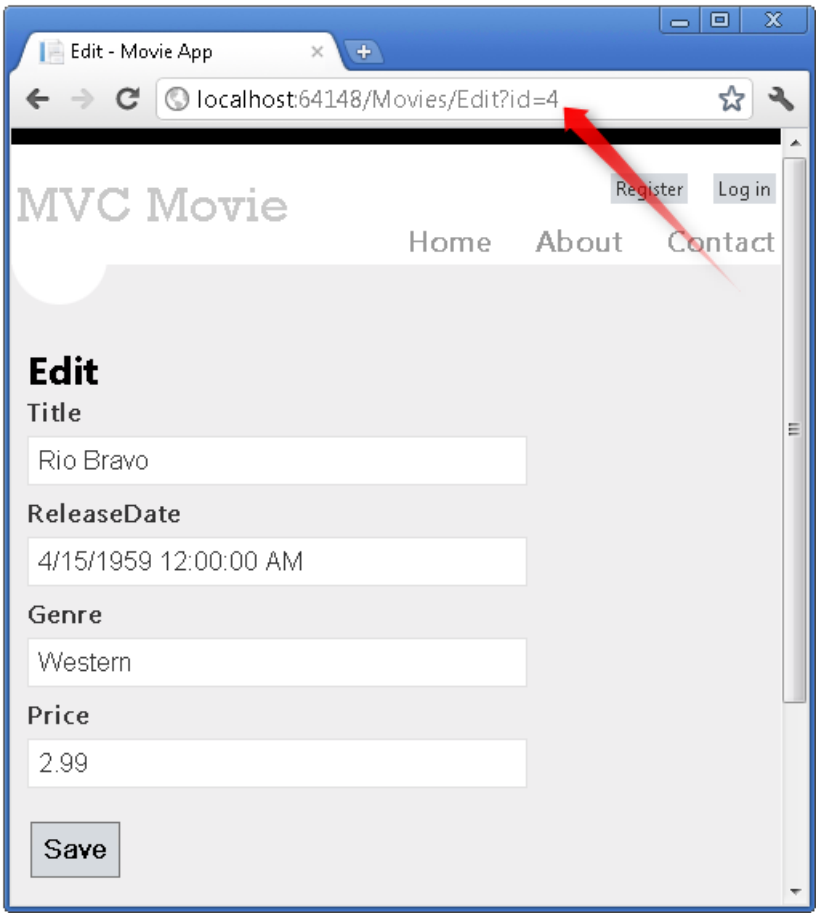
The `Html` object is a helper that's exposed using a property on the System.Web.Mvc.WebViewPage base class. The `ActionLink` method of the helper makes it easy to dynamically generate HTML hyperlinks that link to action methods on controllers. The first argument to the `ActionLink` method is the link text to render (for example, `<a>Edit Me</a>`). The second argument is the name of the action method to invoke. The final argument is an anonymous object that generates the route data (in this case, the ID of 4).

The generated link shown in the previous image is `http://localhost:xxxxx/Movies/Edit/4`. The default route (established in *App_Start\RouteConfig.cs*) takes the URL pattern `{controller}/{action}/{id}`. Therefore, ASP.NET translates `http://localhost:xxxxx/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `ID` equal to 4. Examine the following code from the *App_Start\RouteConfig.cs* file.

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index",
            id = UrlParameter.Optional }
    );
}
```

You can also pass action method parameters using a query string. For example, the URL `http://localhost:xxxxx/Movies/Edit?ID=4` also passes the parameter `ID` of 4 to the `Edit` action method of the `Movies` controller.



Open the `Movies` controller. The two `Edit` action methods are shown below.

```
//
// GET: /Movies/Edit/5

public ActionResult Edit(int id = 0)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}

//
// POST: /Movies/Edit/5

[HttpPost]
public ActionResult Edit(Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Entry(movie).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

Notice the second `Edit` action method is preceded by the `HttpPost` attribute. This attribute specifies that overload of the `Edit` method can be invoked only for POST requests. You could apply the `HttpGet` attribute to the first edit method, but that's not necessary because it's the default. (We'll refer to action methods that are implicitly assigned the `HttpGet` attribute as `HttpGet` methods.)

The `HttpGet` `Edit` method takes the movie ID parameter, looks up the movie using the Entity Framework `Find` method, and returns the selected movie to the Edit view. The ID parameter specifies a default value of zero if the `Edit` method is called without a parameter. If a movie cannot be found, HttpNotFound is returned. When the scaffolding system created the Edit view, it examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The following example shows the Edit view that was generated:

```
@model MvcMovie.Models.Movie

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>Movie</legend>

        @Html.HiddenFor(model => model.ID)

        <div class="editor-label">
            @Html.LabelFor(model => model.Title)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.ReleaseDate)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.ReleaseDate)
            @Html.ValidationMessageFor(model => model.ReleaseDate)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Genre)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Genre)
            @Html.ValidationMessageFor(model => model.Genre)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Price)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Price)
            @Html.ValidationMessageFor(model => model.Price)
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Notice how the view template has a `@model MvcMovie.Models.Movie` statement at the top of the file — this specifies that the view expects the model for the view template to be of type `Movie`.

The scaffolded code uses several *helper methods* to streamline the HTML markup. The `Html.LabelFor` helper

displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The `Html.EditorFor` helper renders an HTML `<input>` element. The `Html.ValidationMessageFor` helper displays any validation messages associated with that property.

Run the application and navigate to the */Movies* URL. Click an **Edit** link. In the browser, view the source for the page. The HTML for the form element is shown below.

```
<form action="/Movies/Edit/4" method="post">    <fieldset>
        <legend>Movie</legend>

        <input data-val="true" data-val-number="The field ID must be a number." data-val-required="The ID
field is required." id="ID" name="ID" type="hidden" value="4" />

        <div class="editor-label">
            <label for="Title">Title</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" id="Title" name="Title" type="text" value="Rio Bravo" />
            <span class="field-validation-valid" data-valmsg-for="Title" data-valmsg-replace="true"></span>
        </div>

        <div class="editor-label">
            <label for="ReleaseDate">ReleaseDate</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" data-val="true" data-val-date="The field ReleaseDate must be a
date." data-val-required="The ReleaseDate field is required." id="ReleaseDate" name="ReleaseDate" type="text"
value="4/15/1959 12:00:00 AM" />
            <span class="field-validation-valid" data-valmsg-for="ReleaseDate" data-valmsg-replace="true">
</span>
        </div>

        <div class="editor-label">
            <label for="Genre">Genre</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" id="Genre" name="Genre" type="text" value="Western" />
            <span class="field-validation-valid" data-valmsg-for="Genre" data-valmsg-replace="true">></span>
        </div>

        <div class="editor-label">
            <label for="Price">Price</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" data-val="true" data-val-number="The field Price must be a
number." data-val-required="The Price field is required." id="Price" name="Price" type="text" value="2.99" />
            <span class="field-validation-valid" data-valmsg-for="Price" data-valmsg-replace="true"></span>
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
</form>
```

The `<input>` elements are in an HTML `<form>` element whose `action` attribute is set to post to the */Movies/Edit* URL. The form data will be posted to the server when the **Edit** button is clicked.

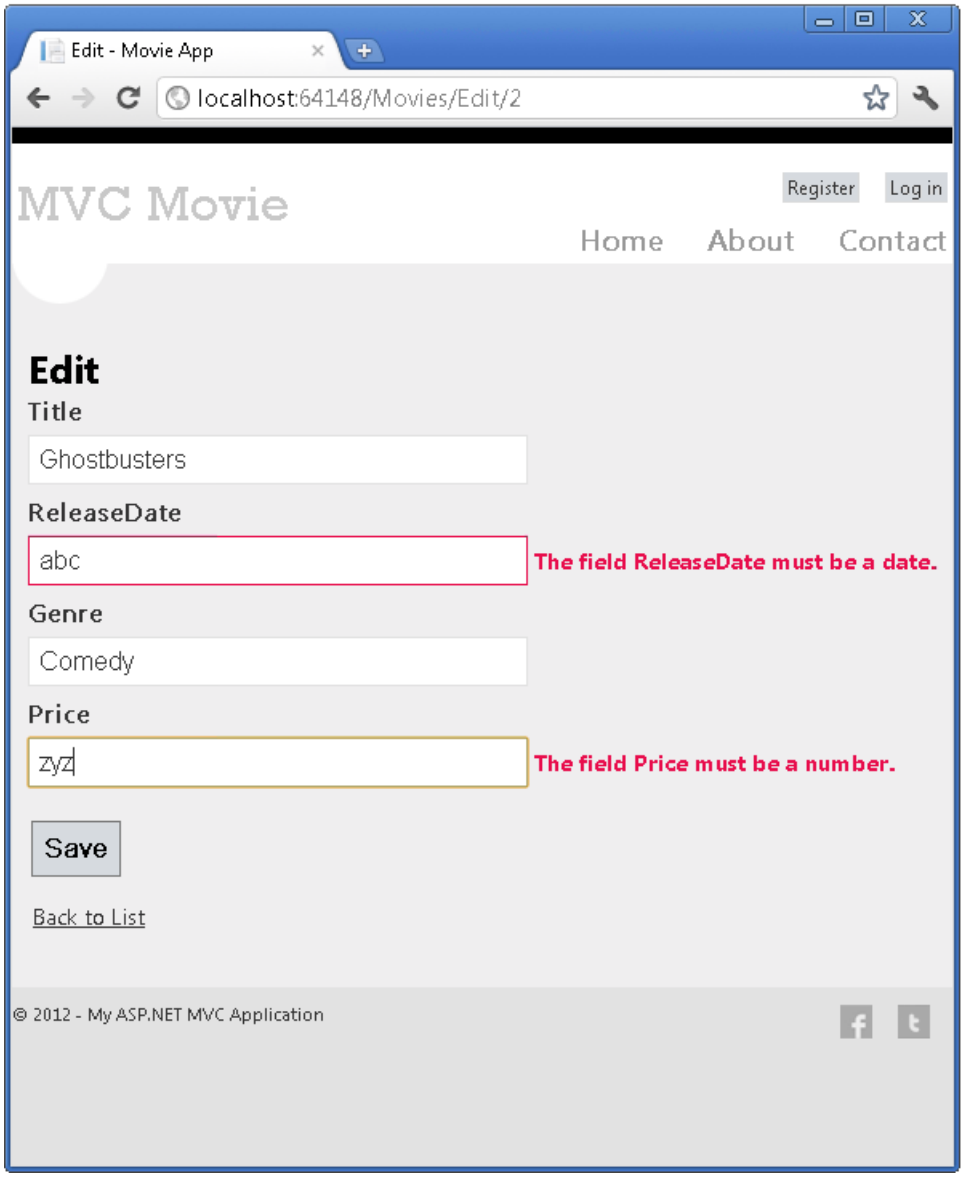
Processing the POST Request

The following listing shows the `HttpPost` version of the `Edit` action method.

```
[HttpPost]
public ActionResult Edit(Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Entry(movie).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The ASP.NET MVC model binder takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` method verifies that the data submitted in the form can be used to modify (edit or update) a `Movie` object. If the data is valid, the movie data is saved to the `Movies` collection of the `db(MovieDBContext` instance). The new movie data is saved to the database by calling the `SaveChanges` method of `MovieDBContext`. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which displays the of movie collection, including the changes just made.

If the posted values aren't valid, they are redisplayed in the form. The `Html.ValidationMessageFor` helpers in the *Edit.cshtml* view template take care of displaying appropriate error messages.



NOTE

to support jQuery validation for non-English locales that use a comma (",") for a decimal point, you must include *globalize.js* and your specific *cultures/globalize.cultures.js* file(from https://github.com/jquery/globalize) and JavaScript to use `Globalize.parseFloat`. The following code shows the modifications to the Views\Movies\Edit.cshtml file to work with the "fr-FR" culture:

```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
    <script src="~/Scripts/globalize.js"></script>
    <script src="~/Scripts/globalize.culture.fr-FR.js"></script>
    <script>
        $.validator.methods.number = function (value, element) {
            return this.optional(element) ||
                !isNaN(Globalize.parseFloat(value));
        }
        $(document).ready(function () {
            Globalize.culture('fr-FR');
        });
    </script>
    <script>
        jQuery.extend(jQuery.validator.methods, {
            range: function (value, element, param) {
                //Use the Globalization plugin to parse the value
                var val = $.global.parseFloat(value);
                return this.optional(element) || (
                    val >= param[0] && val <= param[1]);
            }
        });

    </script>
}
```

The decimal field may require a comma, not a decimal point. As a temporary fix, you can add the globalization element to the projects root web.config file. The following code shows the globalization element with the culture set to United States English.

```
<system.web>
    <globalization culture ="en-US" />
    <!--elements removed for clarity-->
  </system.web>
```

All the `HttpGet` methods follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the model to the view. The `Create` method passes an empty movie object to the Create view. All the methods that create, edit, delete, or otherwise modify data do so in the `HttpPost` overload of the method. Modifying data in an HTTP GET method is a security risk, as described in the blog post entry ASP.NET MVC Tip #46 – Don't use Delete Links because they create Security Holes. Modifying data in a GET method also violates HTTP best practices and the architectural REST pattern, which specifies that GET requests should not change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

Adding a Search Method and Search View

In this section you'll add a `SearchIndex` action method that lets you search movies by genre or name. This will be available using the */Movies/SearchIndex* URL. The request will display an HTML form that contains input elements that a user can enter in order to search for a movie. When a user submits the form, the action method will get the search values posted by the user and use the values to search the database.

Displaying the SearchIndex Form

Start by adding a `SearchIndex` action method to the existing `MoviesController` class. The method will return a view that contains an HTML form. Here's the code:

```
public ActionResult SearchIndex(string searchString)
{
    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

The first line of the `SearchIndex` method creates the following LINQ query to select the movies:

```
var movies = from m in db.Movies
             select m;
```

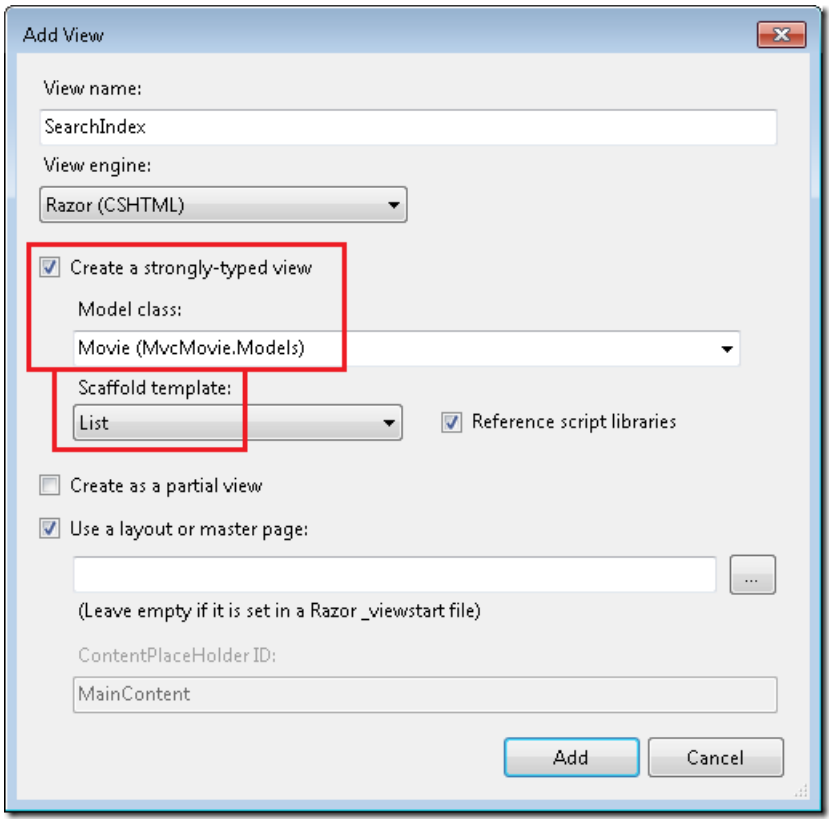
The query is defined at this point, but hasn't yet been run against the data store.

If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string, using the following code:

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

The `s => s.Title` code above is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method used in the above code. LINQ queries are not executed when they are defined or when they are modified by calling a method such as `Where` or `OrderBy` . Instead, query execution is deferred, which means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToList` method is called. In the `SearchIndex` sample, the query is executed in the SearchIndex view. For more information about deferred query execution, see Query Execution.

Now you can implement the `SearchIndex` view that will display the form to the user. Right-click inside the `SearchIndex` method and then click **Add View**. In the **Add View** dialog box, specify that you're going to pass a `Movie` object to the view template as its model class. In the **Scaffold template** list, choose **List**, then click **Add**.



When you click the **Add** button, the *Views\Movies\SearchIndex.cshtml* view template is created. Because you selected **List** in the **Scaffold template** list, Visual Studio automatically generated (scaffolded) some default markup in the view. The scaffolding created an HTML form. It examined the `Movie` class and created code to render `<label>` elements for each property of the class. The listing below shows the Create view that was generated:

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewBag.Title = "SearchIndex";
}

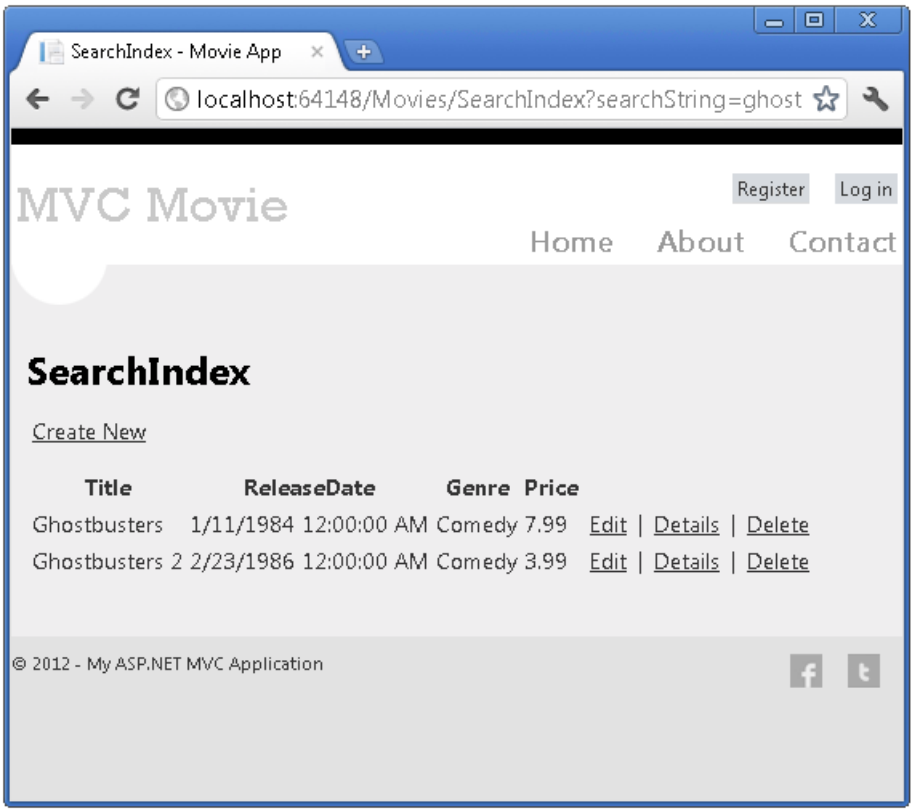
<h2>SearchIndex</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            Title
        </th>
        <th>
            ReleaseDate
        </th>
        <th>
            Genre
        </th>
        <th>
            Price
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
    </tr>
}

</table>
```

Run the application and navigate to */Movies/SearchIndex*. Append a query string such as `?searchString=ghost` to the URL. The filtered movies are displayed.



If you change the signature of the `SearchIndex` method to have a parameter named `id`, the `id` parameter will match the `{id}` placeholder for the default routes set in the *Global.asax* file.

```
{controller}/{action}/{id}
```

The original `SearchIndex` method looks like this::

```
public ActionResult SearchIndex(string searchString)
{
    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

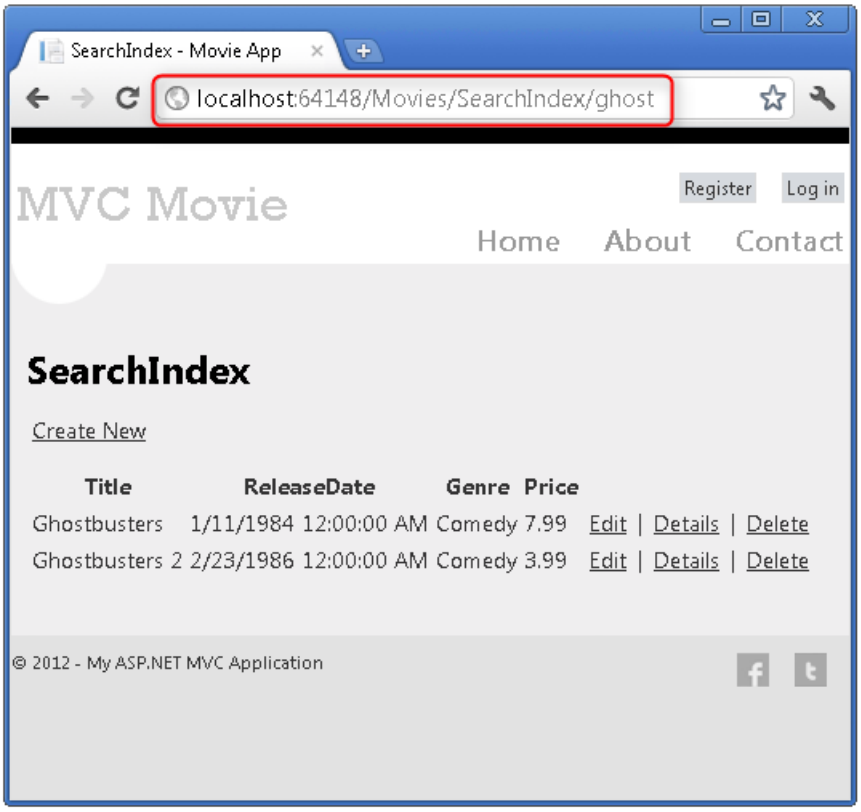
The modified `SearchIndex` method would look as follows:

```
public ActionResult SearchIndex(string id)
{
    string searchString = id;
    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

You can now pass the search title as route data (a URL segment) instead of as a query string value.



However, you can't expect users to modify the URL every time they want to search for a movie. So now you you'll add UI to help them filter movies. If you changed the signature of the `SearchIndex` method to test how to pass the route-bound ID parameter, change it back so that your `SearchIndex` method takes a string parameter named `searchString`:

```
public ActionResult SearchIndex(string searchString)
{
     var movies = from m in db.Movies
                  select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

Open the *Views\Movies\SearchIndex.cshtml* file, and just after `@Html.ActionLink("Create New", "Create")`, add the

following:

```
@using (Html.BeginForm()){
         <p> Title: @Html.TextBox("SearchString")<br />
         <input type="submit" value="Filter" /></p>
        }
```

The following example shows a portion of the *Views\Movies\SearchIndex.cshtml* file with the added filtering markup.

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewBag.Title = "SearchIndex";
}

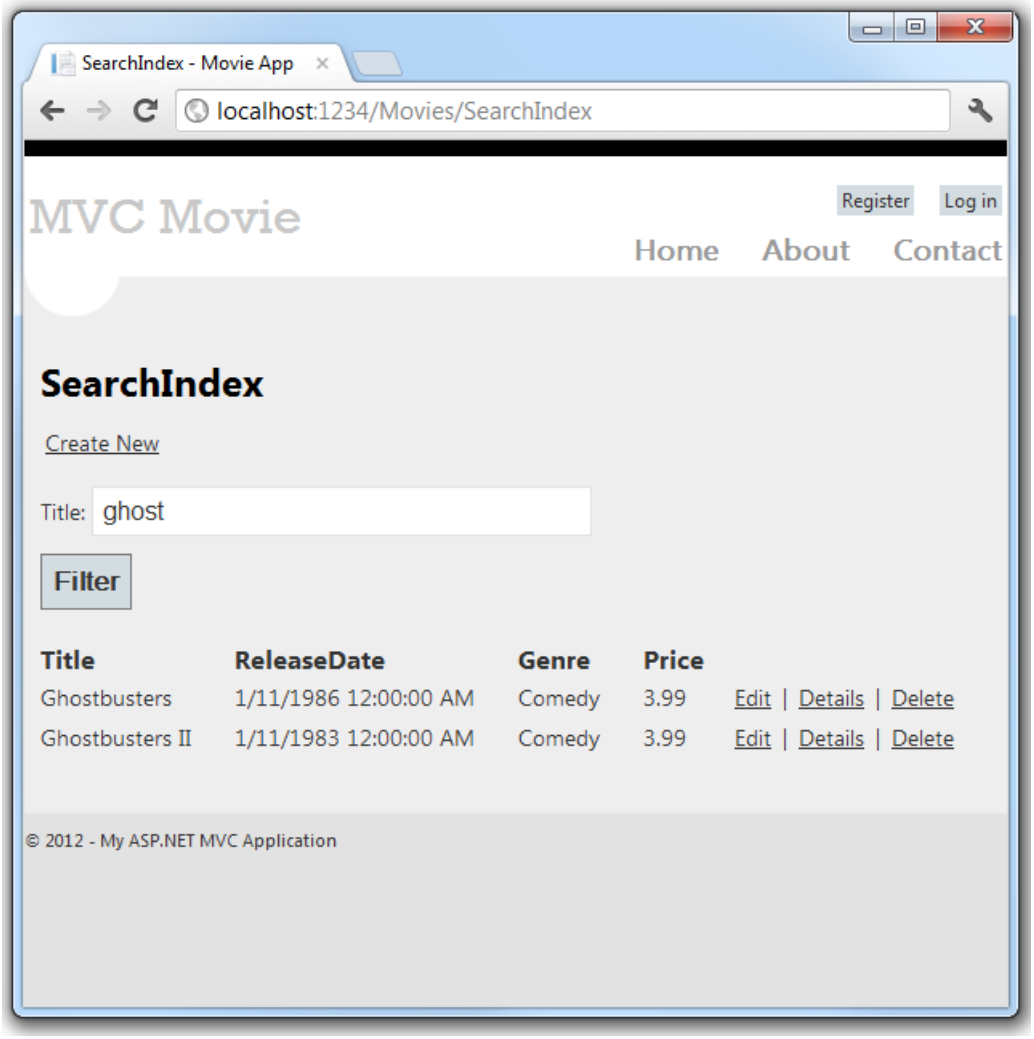
<h2>SearchIndex</h2>

<p>
    @Html.ActionLink("Create New", "Create")

     @using (Html.BeginForm()){
         <p> Title: @Html.TextBox("SearchString") <br />
         <input type="submit" value="Filter" /></p>
        }
</p>
```

The `Html.BeginForm` helper creates an opening `<form>` tag. The `Html.BeginForm` helper causes the form to post to itself when the user submits the form by clicking the **Filter** button.

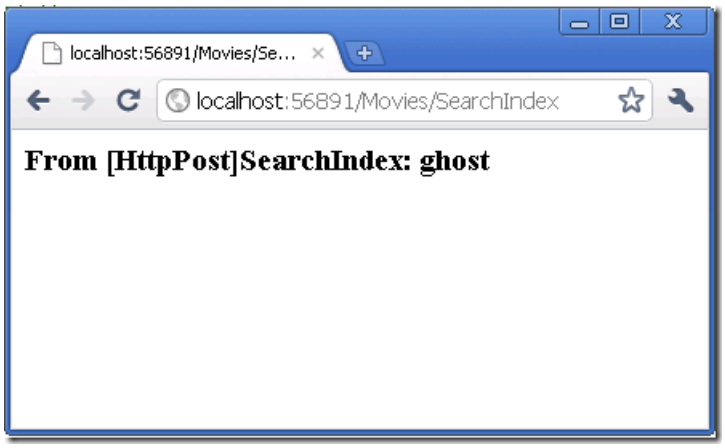
Run the application and try searching for a movie.



There's no `HttpPost` overload of the `SearchIndex` method. You don't need it, because the method isn't changing the state of the application, just filtering data.

You could add the following `HttpPost SearchIndex` method. In that case, the action invoker would match the `HttpPost SearchIndex` method, and the `HttpPost SearchIndex` method would run as shown in the image below.

```
[HttpPost]
public string SearchIndex(FormCollection fc, string searchString)
{
    return "<h3> From [HttpPost]SearchIndex: " + searchString + "</h3>";
}
```

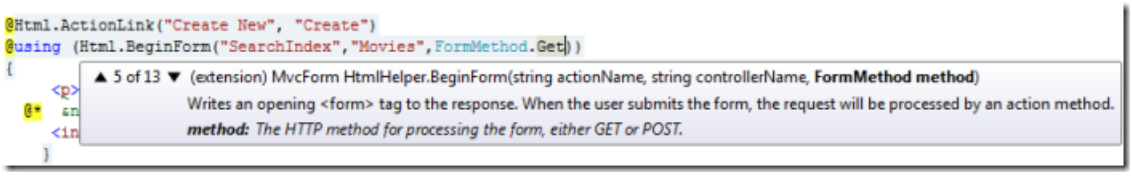


However, even if you add this `HttpPost` version of the `SearchIndex` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends

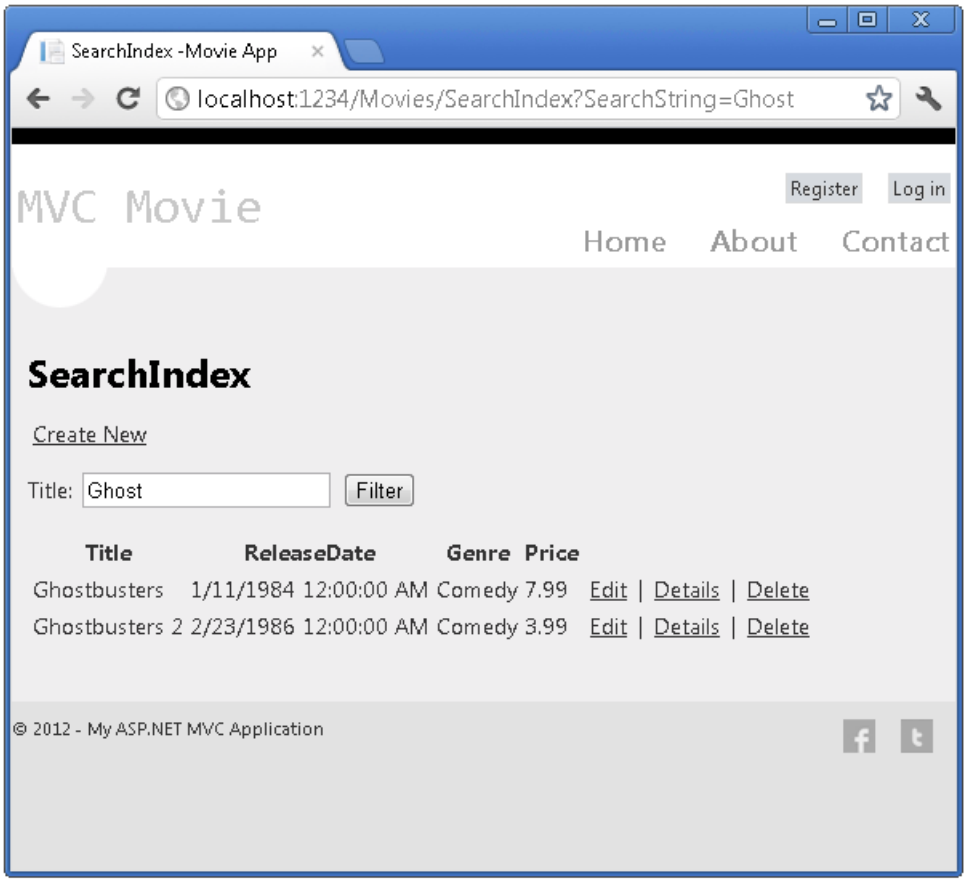
that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (localhost:xxxxx/Movies/SearchIndex) -- there's no search information in the URL itself. Right now, the search string information is sent to the server as a form field value. This means you can't capture that search information to bookmark or send to friends in a URL.

The solution is to use an overload of `BeginForm` that specifies that the POST request should add the search information to the URL and that it should be routed to the HttpGet version of the `SearchIndex` method. Replace the existing parameterless `BeginForm` method with the following:

```
@using (Html.BeginForm("SearchIndex","Movies",FormMethod.Get))
```



Now when you submit a search, the URL contains a search query string. Searching will also go to the `HttpGet SearchIndex` action method, even if you have a `HttpPost SearchIndex` method.



Adding Search by Genre

If you added the `HttpPost` version of the `SearchIndex` method, delete it now.

Next, you'll add a feature to let users search for movies by genre. Replace the `SearchIndex` method with the following code:

```
public ActionResult SearchIndex(string movieGenre, string searchString)
{
    var GenreLst = new List<string>();

    var GenreQry = from d in db.Movies
                   orderby d.Genre
                   select d.Genre;
    GenreLst.AddRange(GenreQry.Distinct());
    ViewBag.movieGenre = new SelectList(GenreLst);

    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (string.IsNullOrEmpty(movieGenre))
        return View(movies);
    else
    {
        return View(movies.Where(x => x.Genre == movieGenre));
    }

}
```

This version of the `SearchIndex` method takes an additional parameter, namely `movieGenre`. The first few lines of code create a `List` object to hold movie genres from the database.

The following code is a LINQ query that retrieves all the genres from the database.

```
var GenreQry = from d in db.Movies
                   orderby d.Genre
                   select d.Genre;
```

The code uses the `AddRange` method of the generic `List` collection to add all the distinct genres to the list. (Without the `Distinct` modifier, duplicate genres would be added — for example, comedy would be added twice in our sample). The code then stores the list of genres in the `ViewBag` object.

The following code shows how to check the `movieGenre` parameter. If it's not empty, the code further constrains the movies query to limit the selected movies to the specified genre.

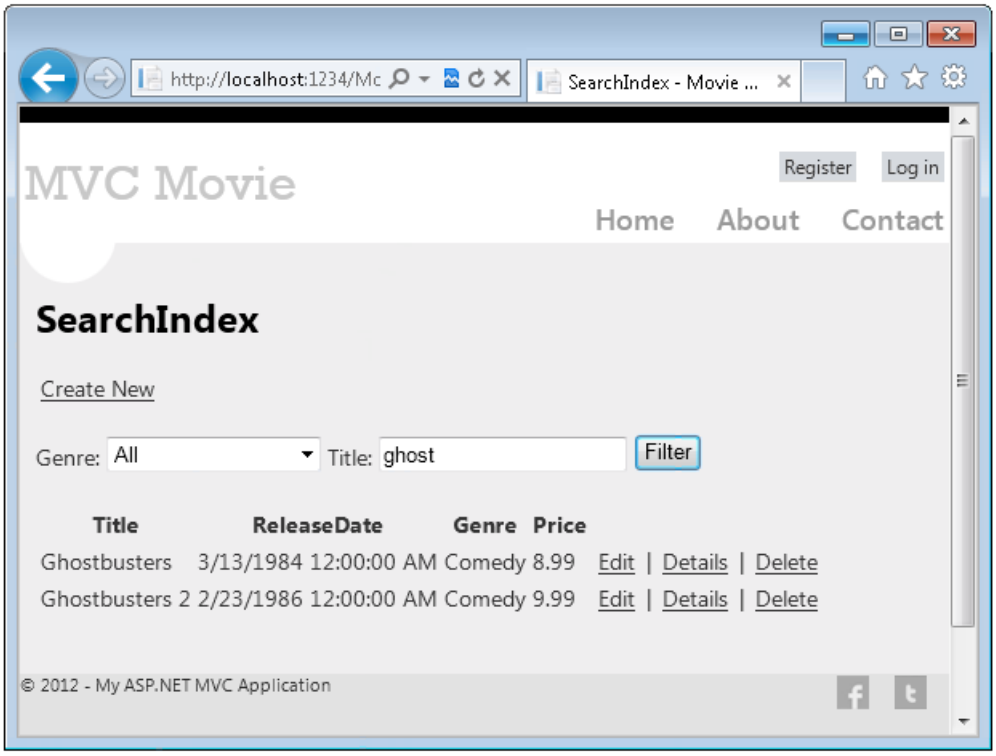
```
if (string.IsNullOrEmpty(movieGenre))
        return View(movies);
else
{
    return View(movies.Where(x => x.Genre == movieGenre));
}
```

Adding Markup to the SearchIndex View to Support Search by Genre

Add an `Html.DropDownList` helper to the *Views\Movies\SearchIndex.cshtml* file, just before the `TextBox` helper. The completed markup is shown below:

```
<p>
    @Html.ActionLink("Create New", "Create")
    @using (Html.BeginForm("SearchIndex","Movies",FormMethod.Get)){
         <p>Genre: @Html.DropDownList("movieGenre", "All")
           Title: @Html.TextBox("SearchString")
         <input type="submit" value="Filter" /></p>
        }
</p>
```

Run the application and browse to */Movies/SearchIndex*. Try a search by genre, by movie name, and by both criteria.



In this section you examined the CRUD action methods and views generated by the framework. You created a search action method and view that let users search by movie title and genre. In the next section, you'll look at how to add a property to the `Movie` model and how to add an initializer that will automatically create a test database.

PREVIOUS NEXT

Adding a New Field to the Movie Model and Table

2/20/2020 • 8 minutes to read • Edit Online

by Rick Anderson

> **NOTE**
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

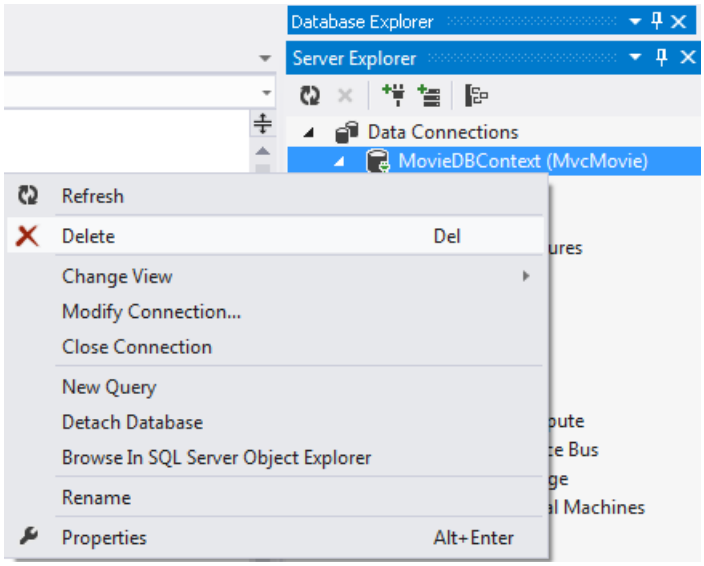
In this section you'll use Entity Framework Code First Migrations to migrate some changes to the model classes so the change is applied to the database.

By default, when you use Entity Framework Code First to automatically create a database, as you did earlier in this tutorial, Code First adds a table to the database to help track whether the schema of the database is in sync with the model classes it was generated from. If they aren't in sync, the Entity Framework throws an error. This makes it easier to track down issues at development time that you might otherwise only find (by obscure errors) at run time.

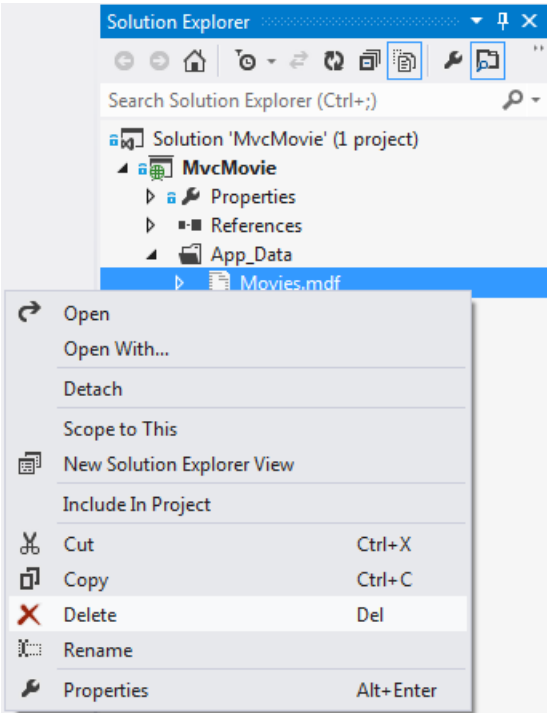
Setting up Code First Migrations for Model Changes

If you are using Visual Studio 2012, double click the *Movies.mdf* file from Solution Explorer to open the database tool. Visual Studio Express for Web will show Database Explorer, Visual Studio 2012 will show Server Explorer. If you are using Visual Studio 2010, use SQL Server Object Explorer.

In the database tool (Database Explorer, Server Explorer or SQL Server Object Explorer), right click on `MovieDBContext` and select **Delete** to drop the movies database.

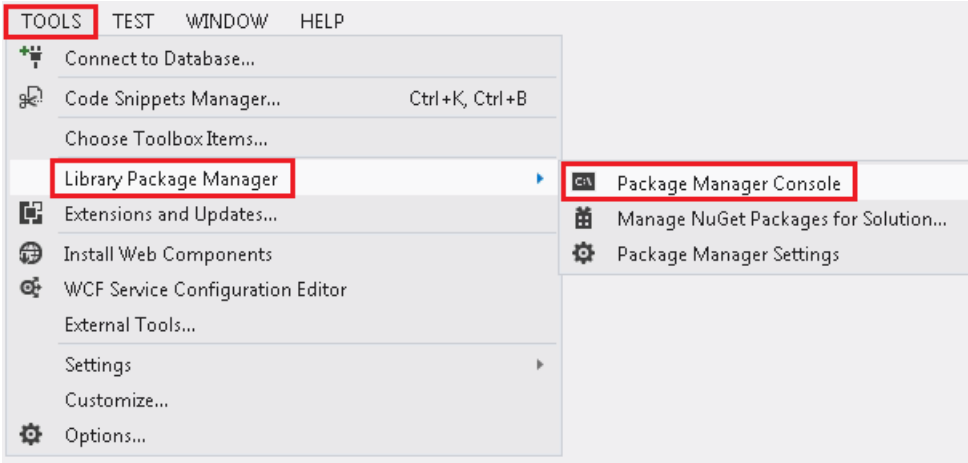


Navigate back to Solution Explorer. Right click on the *Movies.mdf* file and select **Delete** to remove the movies database.

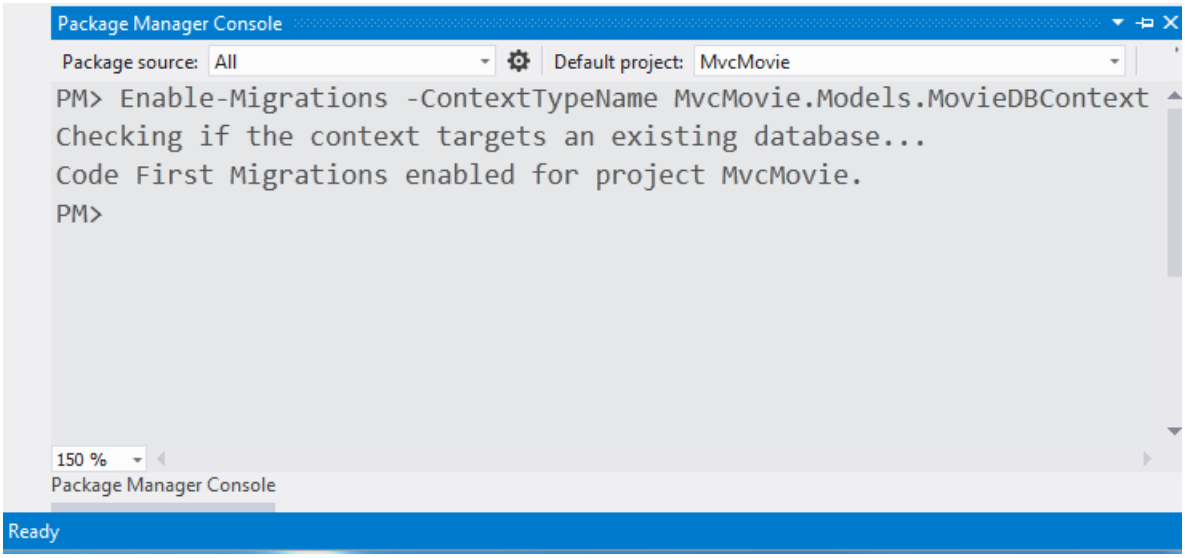


Build the application to make sure there are no errors.

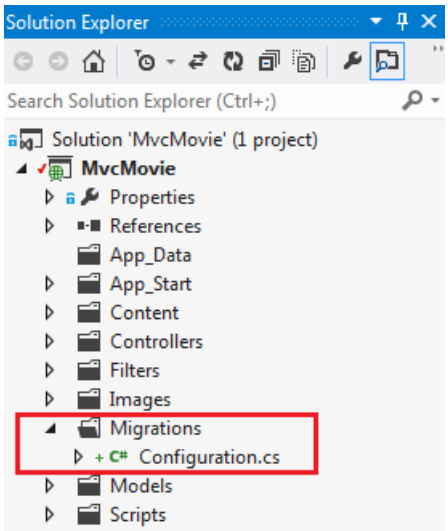
From the **Tools** menu, click **NuGet Package Manager** and then **Package Manager Console**.



In the **Package Manager Console** window at the `PM>` prompt enter "Enable-Migrations -ContextTypeName MvcMovie.Models.MovieDBContext".



The **Enable-Migrations** command (shown above) creates a *Configuration.cs* file in a new *Migrations* folder.



Visual Studio opens the *Configuration.cs* file. Replace the `Seed` method in the *Configuration.cs* file with the following code:

```
protected override void Seed(MvcMovie.Models.MovieDBContext context)
{
    context.Movies.AddOrUpdate( i => i.Title,
        new Movie
        {
            Title = "When Harry Met Sally",
            ReleaseDate = DateTime.Parse("1989-1-11"),
            Genre = "Romantic Comedy",
            Price = 7.99M
        },

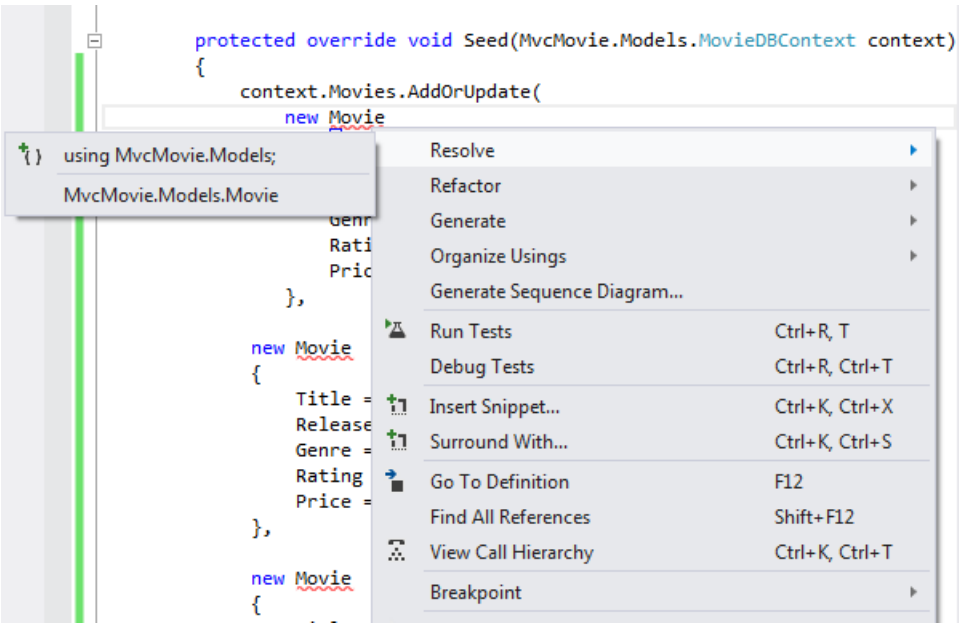
         new Movie
         {
             Title = "Ghostbusters ",
             ReleaseDate = DateTime.Parse("1984-3-13"),
             Genre = "Comedy",
             Price = 8.99M
         },

         new Movie
         {
             Title = "Ghostbusters 2",
             ReleaseDate = DateTime.Parse("1986-2-23"),
             Genre = "Comedy",
             Price = 9.99M
         },

        new Movie
        {
            Title = "Rio Bravo",
            ReleaseDate = DateTime.Parse("1959-4-15"),
            Genre = "Western",
            Price = 3.99M
        }
    );

}
```

Right click on the red squiggly line under `Movie` and select **Resolve** then **using MvcMovie.Models;**



Doing so adds the following using statement:

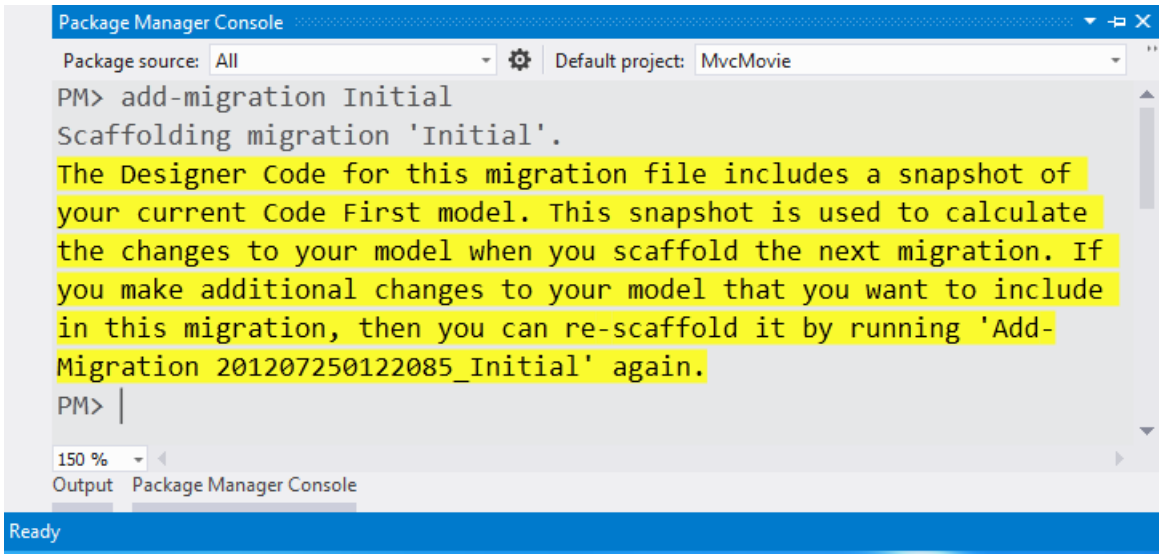
```
using MvcMovie.Models;
```

> **NOTE**
> Code First Migrations calls the `Seed` method after every migration (that is, calling **update-database** in the Package Manager Console), and this method updates rows that have already been inserted, or inserts them if they don't exist yet.

Press CTRL-SHIFT-B to build the project.(The following steps will fail if your don't build at this point.)

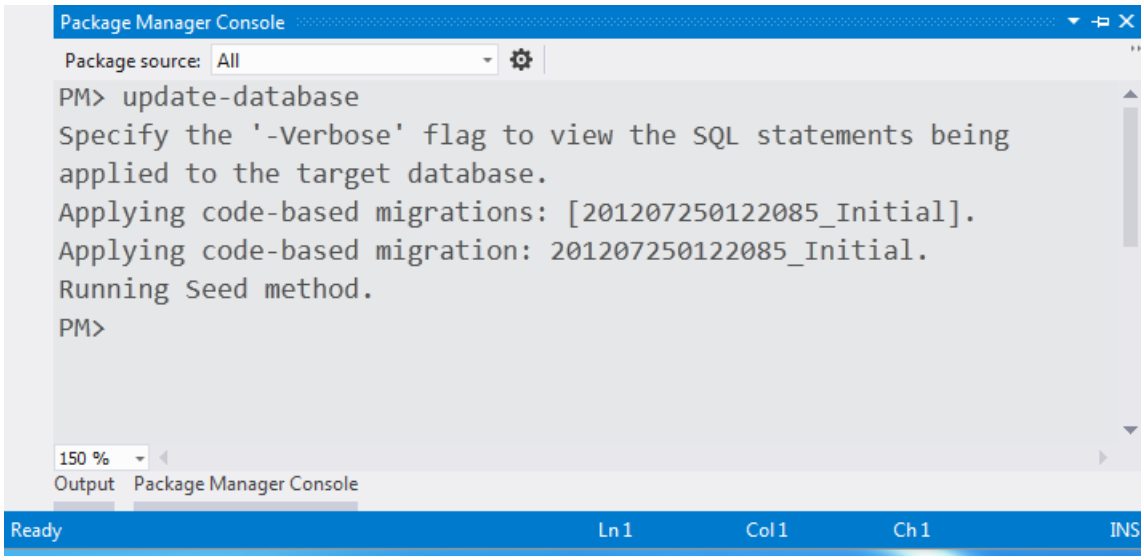
The next step is to create a `DbMigration` class for the initial migration. This migration to creates a new database, that's why you deleted the *movie.mdf* file in a previous step.

In the **Package Manager Console** window, enter the command "add-migration Initial" to create the initial migration. The name "Initial" is arbitrary and is used to name the migration file created.



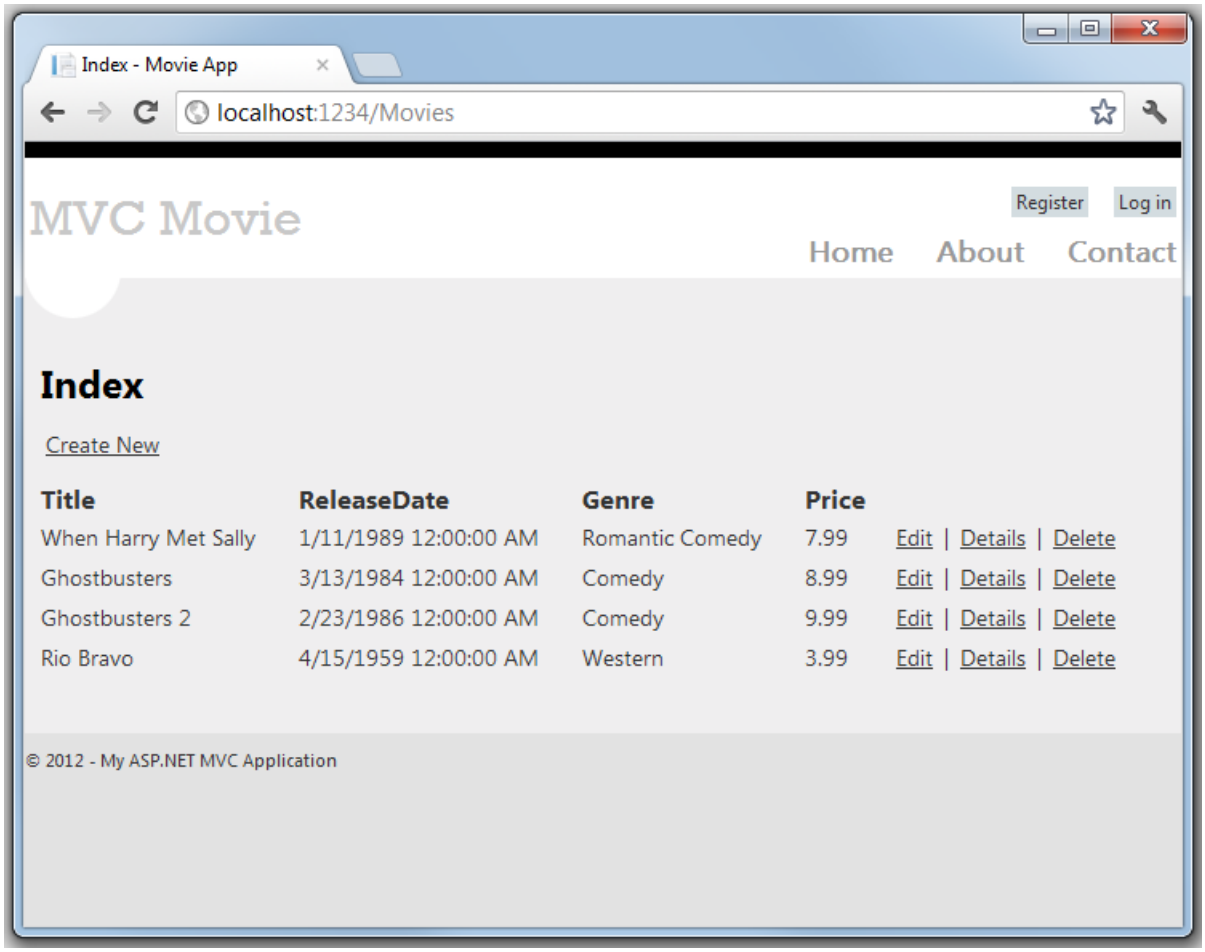
Code First Migrations creates another class file in the *Migrations* folder (with the name *{DateStamp}_Initial.cs*), and this class contains code that creates the database schema. The migration filename is pre-fixed with a timestamp to help with ordering. Examine the *{DateStamp}_Initial.cs* file, it contains the instructions to create the Movies table for the Movie DB. When you update the database in the instructions below, this *{DateStamp}_Initial.cs* file will run and create the DB schema. Then the **Seed** method will run to populate the DB with test data.

In the **Package Manager Console**, enter the command "update-database" to create the database and run the **Seed** method.



If you get an error that indicates a table already exists and can't be created, it is probably because you ran the application after you deleted the database and before you executed `update-database`. In that case, delete the *Movies.mdf* file again and retry the `update-database` command. If you still get an error, delete the migrations folder and contents then start with the instructions at the top of this page (that is delete the *Movies.mdf* file then proceed to Enable-Migrations).

Run the application and navigate to the */Movies* URL. The seed data is displayed.



Adding a Rating Property to the Movie Model

Start by adding a new `Rating` property to the existing `Movie` class. Open the *Models\Movie.cs* file and add the `Rating` property like this one:

```
public string Rating { get; set; }
```

The complete `Movie` class now looks like the following code:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

Build the application using the **Build** >**Build Movie** menu command or by pressing CTRL-SHIFT-B.

Now that you've updated the `Model` class, you also need to update the *\Views\Movies\Index.cshtml* and *\Views\Movies\Create.cshtml* view templates in order to display the new `Rating` property in the browser view.

Open the *\Views\Movies\Index.cshtml* file and add a `<th>Rating</th>` column heading just after the **Price** column. Then add a `<td>` column near the end of the template to render the `@item.Rating` value. Below is what the updated *Index.cshtml* view template looks like:

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Genre)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Price)
        </th>
         <th>
            @Html.DisplayNameFor(model => model.Rating)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Rating)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
    </tr>
}

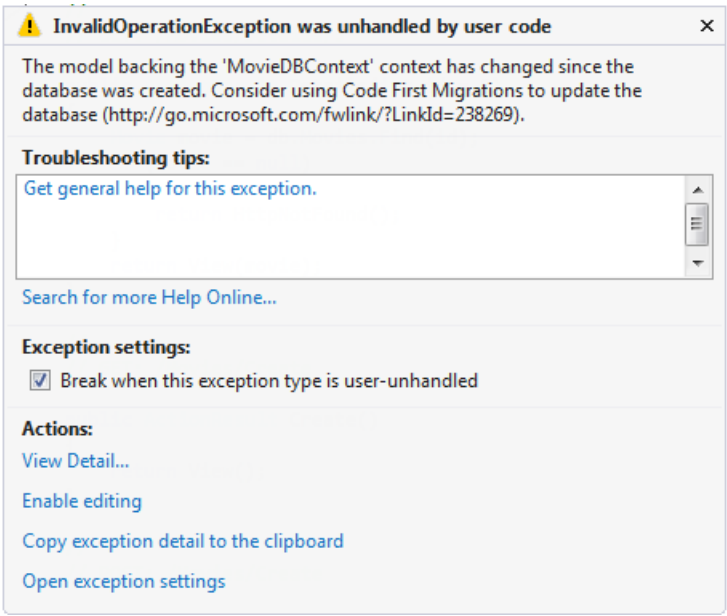
</table>
```

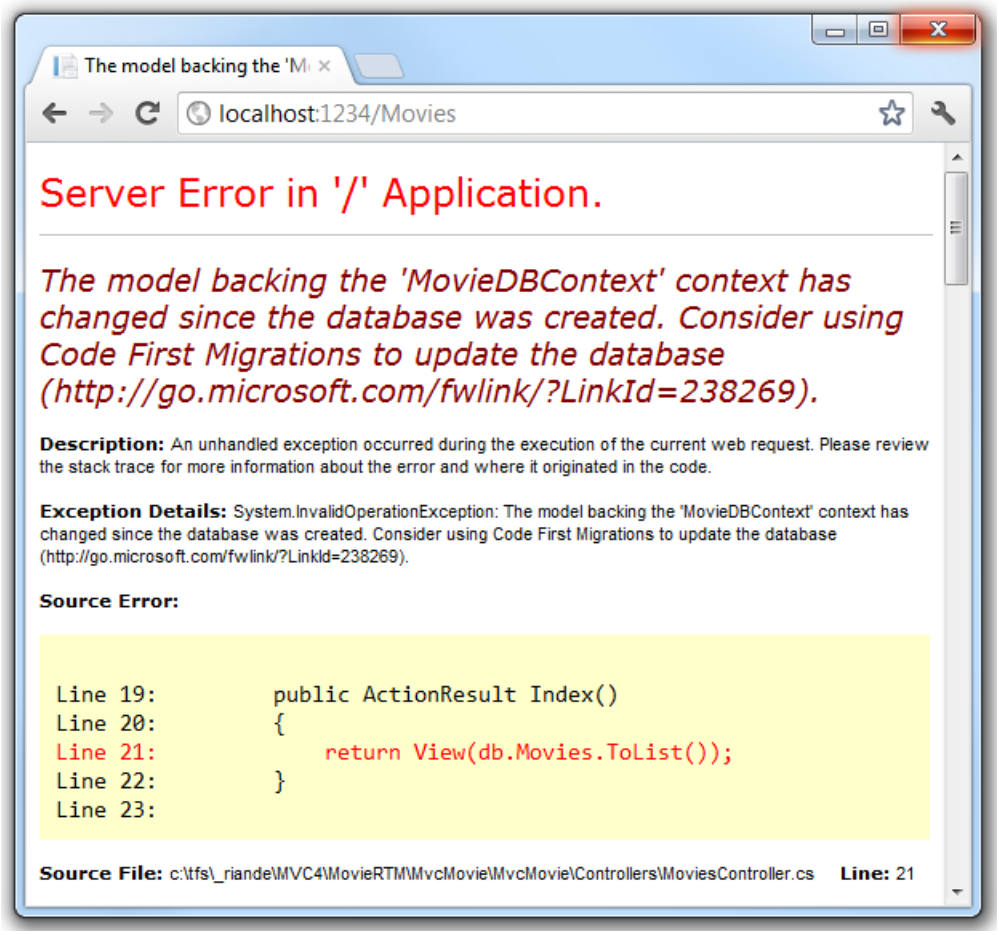
Next, open the *\Views\Movies\Create.cshtml* file and add the following markup near the end of the form. This renders a text box so that you can specify a rating when a new movie is created.

```
<div class="editor-label">
    @Html.LabelFor(model => model.Rating)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Rating)
    @Html.ValidationMessageFor(model => model.Rating)
</div>
```

You've now updated the application code to support the new `Rating` property.

Now run the application and navigate to the */Movies* URL. When you do this, though, you'll see one of the following errors:





You're seeing this error because the updated `Movie` model class in the application is now different than the schema of the `Movie` table of the existing database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient when doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you *don't* want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develope an application. For more information on Entity Framework database initializers, see Tom Dykstra's ASP.NET MVC/Entity Framework tutorial.
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

For this tutorial, we'll use Code First Migrations.

Update the Seed method so that it provides a value for the new column. Open Migrations\Configuration.cs file and add a Rating field to each Movie object.

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "G",
    Price = 7.99M
},
```

Build the solution, and then open the **Package Manager Console** window and enter the following command:

`add-migration AddRatingMig`

The `add-migration` command tells the migration framework to examine the current movie model with the current movie DB schema and create the necessary code to migrate the DB to the new model. The AddRatingMig is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration step.

When this command finishes, Visual Studio opens the class file that defines the new `DbMigration` derived class, and in the `Up` method you can see the code that creates the new column.

```
public partial class AddRatingMig : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.Movies", "Rating", c => c.String());
    }

    public override void Down()
    {
        DropColumn("dbo.Movies", "Rating");
    }
}
```

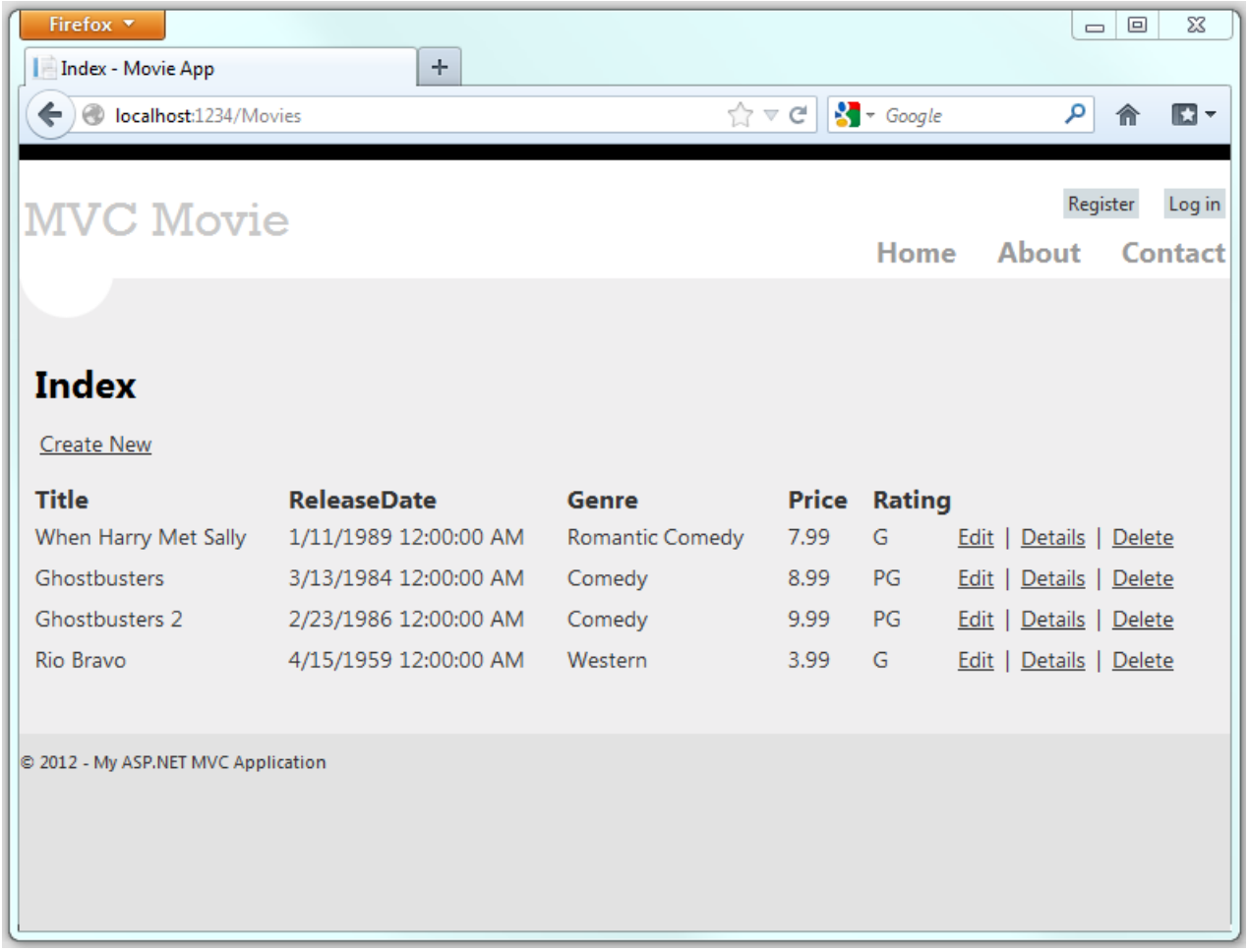
Build the solution, and then enter the "update-database" command in the **Package Manager Console** window.

The following image shows the output in the **Package Manager Console** window (The date stamp prepending

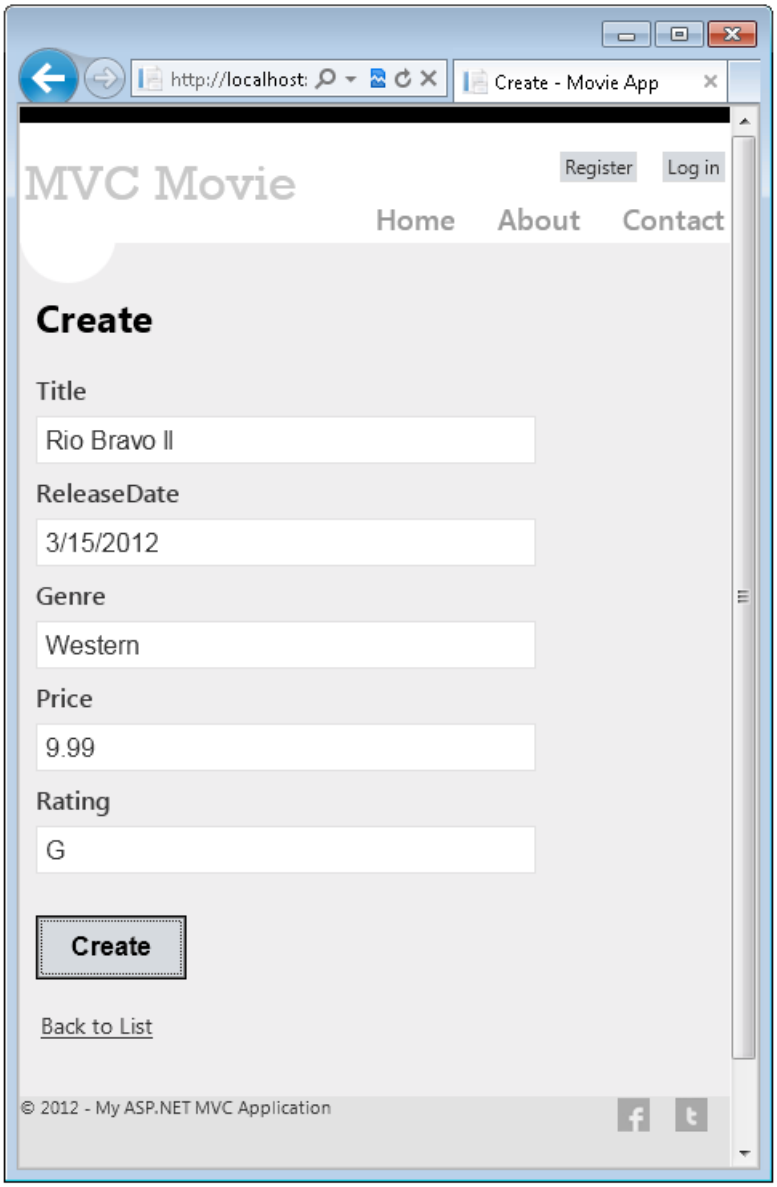
AddRatingMig will be different.)

```
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Applying code-based migrations: [201207260218501_AddRatingMig].
Applying code-based migration: 201207260218501_AddRatingMig.
Running Seed method.
PM>
```

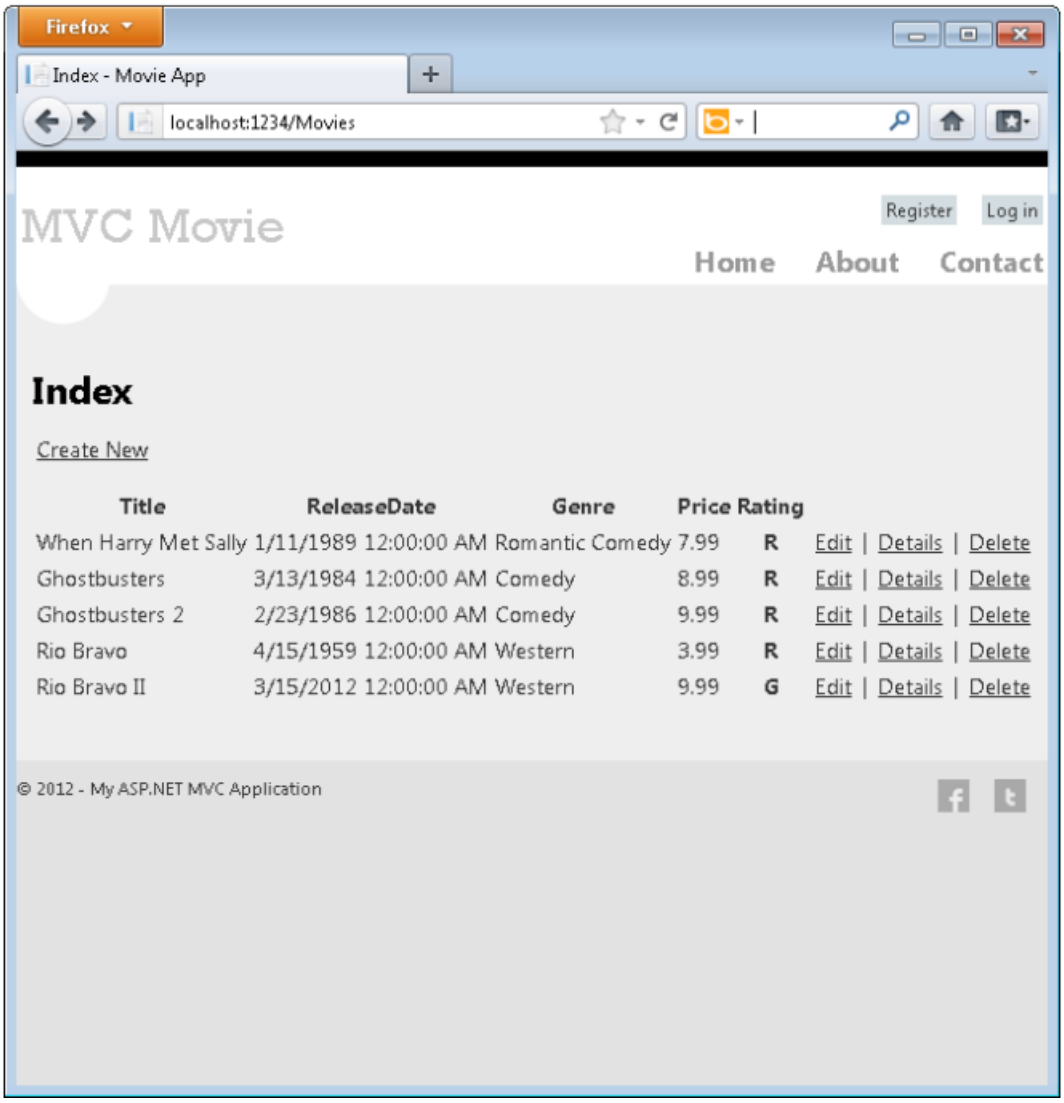
Re-run the application and navigate to the /Movies URL. You can see the new Rating field.



Click the **Create New** link to add a new movie. Note that you can add a rating.



Click **Create**. The new movie, including the rating, now shows up in the movies listing:



You should also add the `Rating` field to the Edit, Details and SearchIndex view templates.

You could enter the "update-database" command in the **Package Manager Console** window again and no changes would be made, because the schema matches the model.

In this section you saw how you can modify model objects and keep the database in sync with the changes. You also learned a way to populate a newly created database with sample data so you can try out scenarios. Next, let's look at how you can add richer validation logic to the model classes and enable some business rules to be enforced.

PREVIOUS NEXT

Adding Validation to the Model

2/20/2020 • 11 minutes to read • Edit Online

by Rick Anderson

> **NOTE**
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

In this section you'll add validation logic to the `Movie` model, and you'll ensure that the validation rules are enforced any time a user attempts to create or edit a movie using the application.

Keeping Things DRY

One of the core design tenets of ASP.NET MVC is DRY ("Don't Repeat Yourself"). ASP.NET MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an application. This reduces the amount of code you need to write and makes the code you do write less error prone and easier to maintain.

The validation support provided by ASP.NET MVC and Entity Framework Code First is a great example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the application.

Let's look at how you can take advantage of this validation support in the movie application.

Adding Validation Rules to the Movie Model

You'll begin by adding some validation logic to the `Movie` class.

Open the *Movie.cs* file. Add a `using` statement at the top of the file that references the `System.ComponentModel.DataAnnotations` namespace:

```
using System.ComponentModel.DataAnnotations;
```

Notice the namespace does not contain `System.Web`. DataAnnotations provides a built-in set of validation attributes that you can apply declaratively to any class or property.

Now update the `Movie` class to take advantage of the built-in `Required`, `StringLength`, and `Range` validation attributes. Use the following code as an example of where to apply the attributes.

```
public class Movie {
    public int ID { get; set; }

    [Required]
    public string Title { get; set; }

    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Required]
    public string Genre { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [StringLength(5)]
    public string Rating { get; set; }
}
```

Run the application and you will again get the following run time error:

The model backing the 'MovieDBContext' context has changed since the database was created. Consider using Code First Migrations to update the database (https://go.microsoft.com/fwlink/?LinkId=238269).

We will use migrations to update the schema. Build the solution, and then open the **Package Manager Console** window and enter the following commands:

```
add-migration AddDataAnnotationsMig
update-database
```

When this command finishes, Visual Studio opens the class file that defines the new `DbMigration` derived class with the name specified (*AddDataAnnotationsMig*), and in the `Up` method you can see the code that updates the schema constraints. The `Title` and `Genre` fields are no longer nullable (that is, you must enter a value) and the `Rating` field has a maximum length of 5.

The validation attributes specify behavior that you want to enforce on the model properties they are applied to. The `Required` attribute indicates that a property must have a value; in this sample, a movie has to have values for the `Title`, `ReleaseDate`, `Genre`, and `Price` properties in order to be valid. The `Range` attribute constrains a value to within a specified range. The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length. Intrinsic types (such as `decimal, int, float, DateTime`) are required by default and don't need the `Required` attribute.

Code First ensures that the validation rules you specify on a model class are enforced before the application saves changes in the database. For example, the code below will throw an exception when the `SaveChanges` method is called, because several required `Movie` property values are missing and the price is zero (which is out of the valid range).

```
MovieDBContext db = new MovieDBContext();

Movie movie = new Movie();
movie.Title = "Gone with the Wind";
movie.Price = 0.0M;

db.Movies.Add(movie);
db.SaveChanges();        // <= Will throw server side validation exception
```

Having validation rules automatically enforced by the .NET Framework helps make your application more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Here's a complete code listing for the updated *Movie.cs* file:

```
using System;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models {
    public class Movie {
        public int ID { get; set; }

        [Required]
        public string Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }

        [Required]
        public string Genre { get; set; }

        [Range(1, 100)]
        [DataType(DataType.Currency)]
        public decimal Price { get; set; }

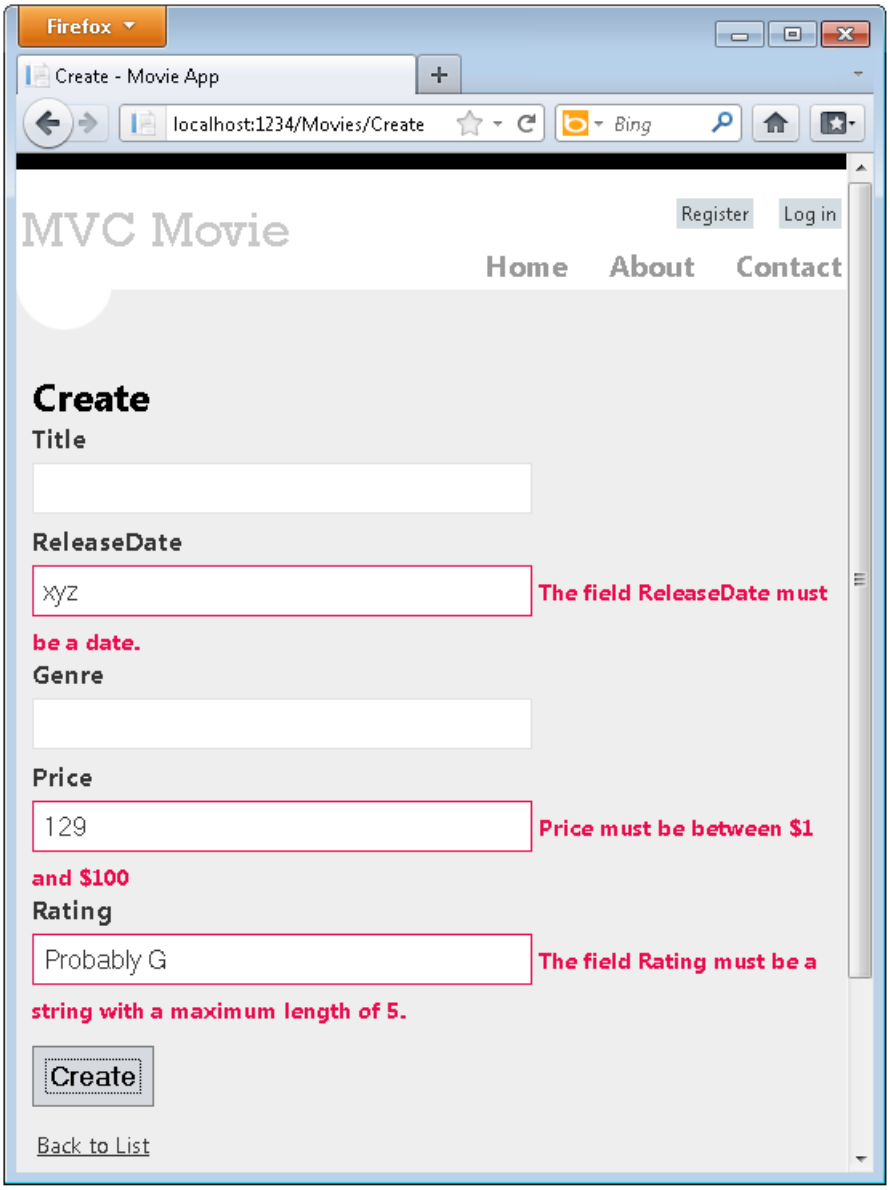
        [StringLength(5)]
        public string Rating { get; set; }
    }

    public class MovieDBContext : DbContext {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

Validation Error UI in ASP.NET MVC

Re-run the application and navigate to the */Movies* URL.

Click the **Create New** link to add a new movie. Fill out the form with some invalid values and then click the **Create** button.



NOTE

to support jQuery validation for non-English locales that use a comma (",") for a decimal point, you must include *globalize.js* and your specific *cultures/globalize.cultures.js* file(from https://github.com/jquery/globalize) and JavaScript to use `Globalize.parseFloat`. The following code shows the modifications to the Views\Movies\Edit.cshtml file to work with the "fr-FR" culture:

```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
    <script src="~/Scripts/globalize.js"></script>
    <script src="~/Scripts/globalize.culture.fr-FR.js"></script>
    <script>
        $.validator.methods.number = function (value, element) {
            return this.optional(element) ||
                !isNaN(Globalize.parseFloat(value));
        }
        $(document).ready(function () {
            Globalize.culture('fr-FR');
        });
    </script>
    <script>
        jQuery.extend(jQuery.validator.methods, {
            range: function (value, element, param) {
                //Use the Globalization plugin to parse the value
                var val = $.global.parseFloat(value);
                return this.optional(element) || (
                    val >= param[0] && val <= param[1]);
            }
        });

    </script>
}
```

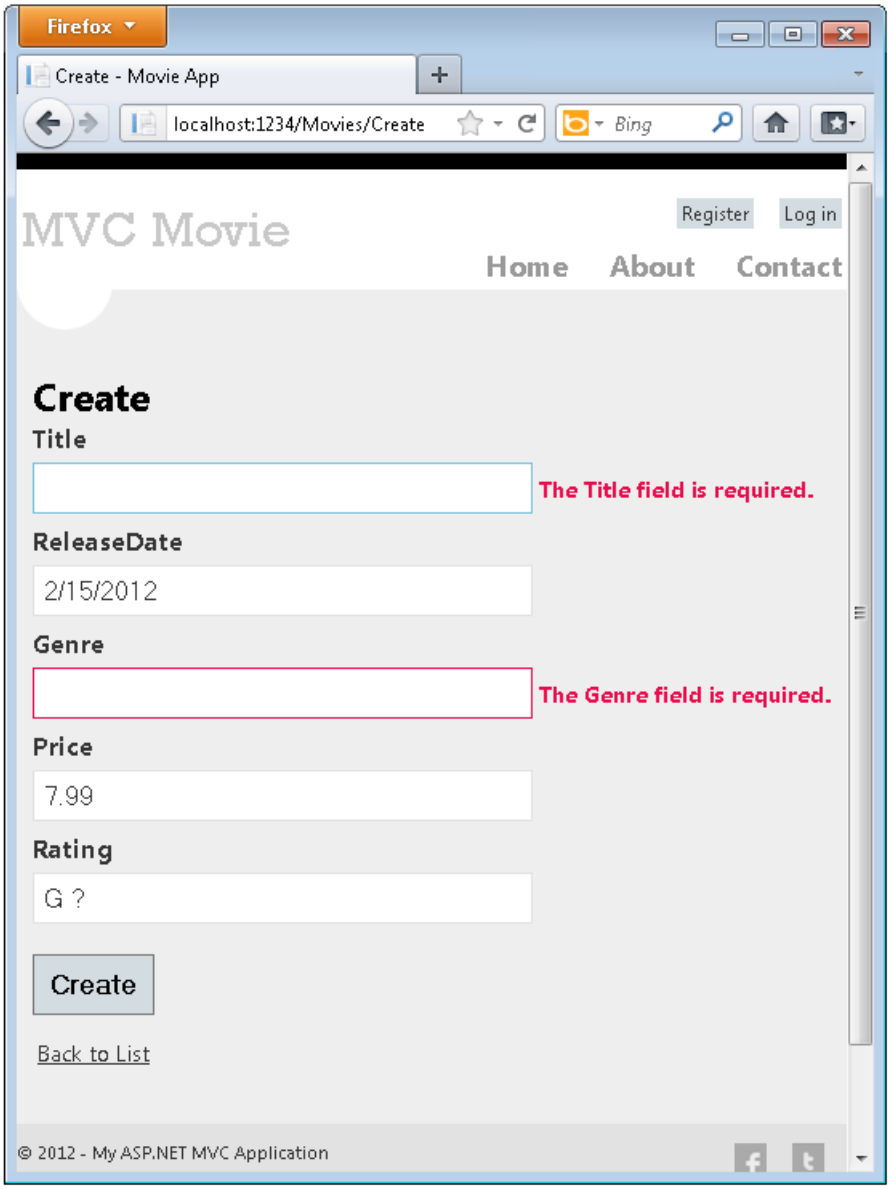
Notice how the form has automatically used a red border color to highlight the text boxes that contain invalid data and has emitted an appropriate validation error message next to each one. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

A real benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the *Create.cshtml* view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the `Movie` model class.

You might have noticed for the properties `Title` and `Genre`, the required attribute is not enforced until you submit the form (hit the **Create** button), or enter text into the input field and removed it. For a field which is initially empty (such as the fields on the Create view) and which has only the required attribute and no other validation attributes, you can do the following to trigger validation:

1. Tab into the field.
2. Enter some text.
3. Tab out.
4. Tab back into the field.
5. Remove the text.
6. Tab out.

The above sequence will trigger the required validation without hitting the submit button. Simply hitting the submit button without entering any of the fields will trigger client side validation. The form data is not sent to the server until there are no client side validation errors. You can test this by putting a break point in the HTTP Post method or using the fiddler tool or the IE 9 F12 developer tools.



How Validation Occurs in the Create View and Create Action Method

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The next listing shows what the `Create` methods in the `MovieController` class look like. They're unchanged from how you created them earlier in this tutorial.

```
//
// GET: /Movies/Create

public ActionResult Create()
{
    return View();
}

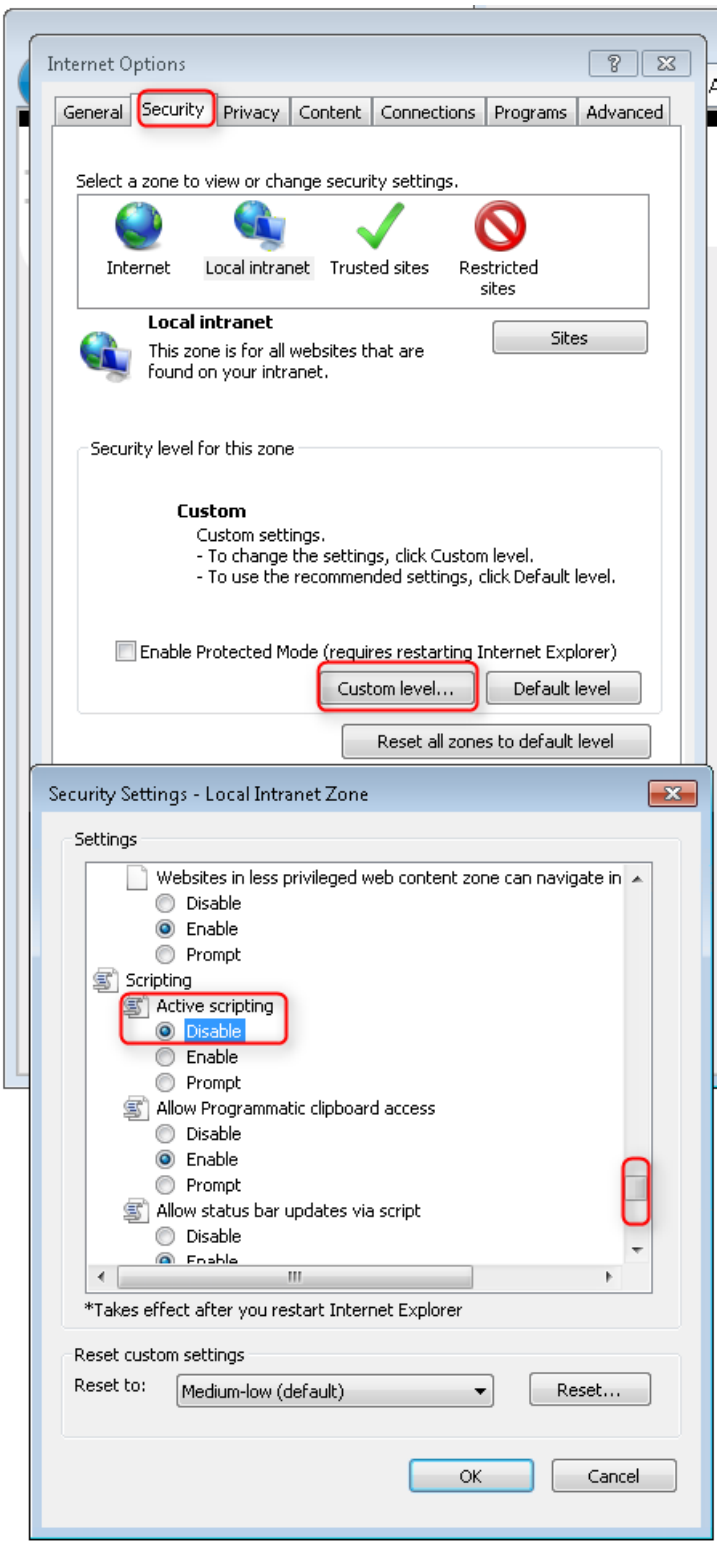
//
// POST: /Movies/Create

[HttpPost]
public ActionResult Create(Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Movies.Add(movie);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(movie);
}
```

The first (HTTP GET) `Create` action method displays the initial Create form. The second (`[HttpPost]`) version handles the form post. The second `Create` method (The `HttpPost` version) calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method re-displays the form. If there are no errors, the method saves the new movie in the database. In our movie example we are using, **the form is not posted to the server when their are validation errors detected on the client side; the second `Create` method is never called**. If you disable JavaScript in your browser, client validation is disabled and the HTTP POST `Create` method calls `ModelState.IsValid` to check whether the movie has any validation errors.

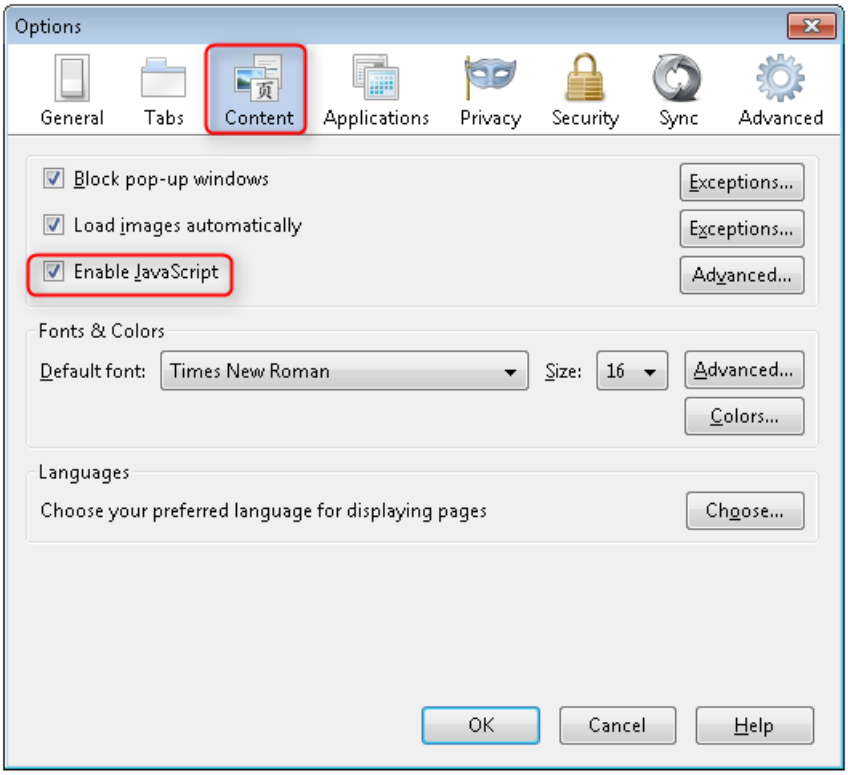
You can set a break point in the `HttpPost Create` method and verify the method is never called, client side validation will not submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript. The following image shows how to disable JavaScript in Internet Explorer.



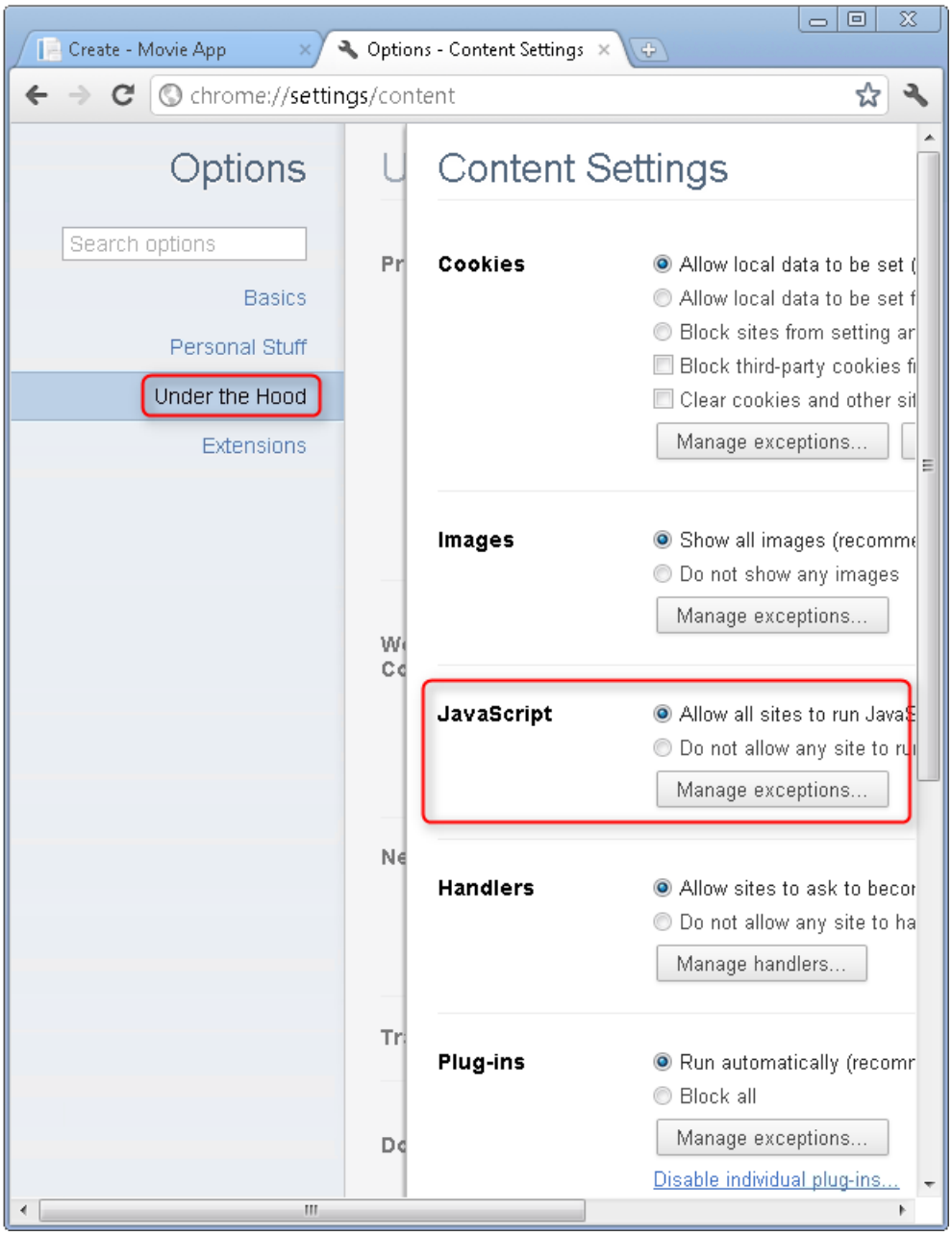

```
[HttpPost]
public ActionResult Create(Movie movie)
{
    if (ModelState.IsValid)
    {                                   ModelState.IsValid | false
        db.Movies.Add(movie);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(movie);
}
```

The following image shows how to disable JavaScript in the FireFox browser.



The following image shows how to disable JavaScript with the Chrome browser.



Below is the *Create.cshtml* view template that you scaffolded earlier in the tutorial. It's used by the action methods shown above both to display the initial form and to redisplay it in the event of an error.

```
@model MvcMovie.Models.Movie

@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>

<script src="@Url.Content("~/Scripts/jquery.validate.min.js")"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>Movie</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.Title)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.ReleaseDate)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.ReleaseDate)
            @Html.ValidationMessageFor(model => model.ReleaseDate)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Genre)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Genre)
            @Html.ValidationMessageFor(model => model.Genre)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Price)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Price)
            @Html.ValidationMessageFor(model => model.Price)
        </div>
        <div class="editor-label">
    @Html.LabelFor(model => model.Rating)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Rating)
    @Html.ValidationMessageFor(model => model.Rating)
</div>
        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Notice how the code uses an `Html.EditorFor` helper to output the `<input>` element for each `Movie` property.

Next to this helper is a call to the `Html.ValidationMessageFor` helper method. These two helper methods work with the model object that's passed by the controller to the view (in this case, a `Movie` object). They automatically look for validation attributes specified on the model and display error messages as appropriate.

What's really nice about this approach is that neither the controller nor the Create view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the `Movie` class. These same validation rules are automatically applied to the Edit view and any other views templates you might create that edit your model.

If you want to change the validation logic later, you can do so in exactly one place by adding validation attributes to the model (in this example, the `movie` class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the DRY principle.

Adding Formatting to the Movie Model

Open the *Movie.cs* file and examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a `DataType` enumeration value to the release date and to the price fields. The following code shows the `ReleaseDate` and `Price` properties with the appropriate `DisplayFormat` attribute.

```
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

The `DataType` attributes are not validation attributes, they are used to tell the view engine how to render the HTML. In the example above, the `DataType.Date` attribute displays the movie dates as dates only, without time. For example, the following `DataType` attributes don't validate the format of the data:

```
[DataType(DataType.EmailAddress)]
[DataType(DataType.PhoneNumber)]
[DataType(DataType.Url)]
```

The attributes listed above only provide hints for the view engine to format the data (and supply attributes such as <a> for URL's and <a href="mailto:EmailAddress.com"> for email. You can use the RegularExpression attribute to validate the format of the data.

An alternative approach to using the `DataType` attributes, you could explicitly set a `DataFormatString` value. The following code shows the release date property with a date format string (namely, "d"). You'd use this to specify that you don't want to time as part of the release date.

```
[DisplayFormat(DataFormatString = "{0:d}")]
public DateTime ReleaseDate { get; set; }
```

The complete `Movie` class is shown below.

```
public class Movie {
    public int ID { get; set; }

    [Required]
    public string Title { get; set; }

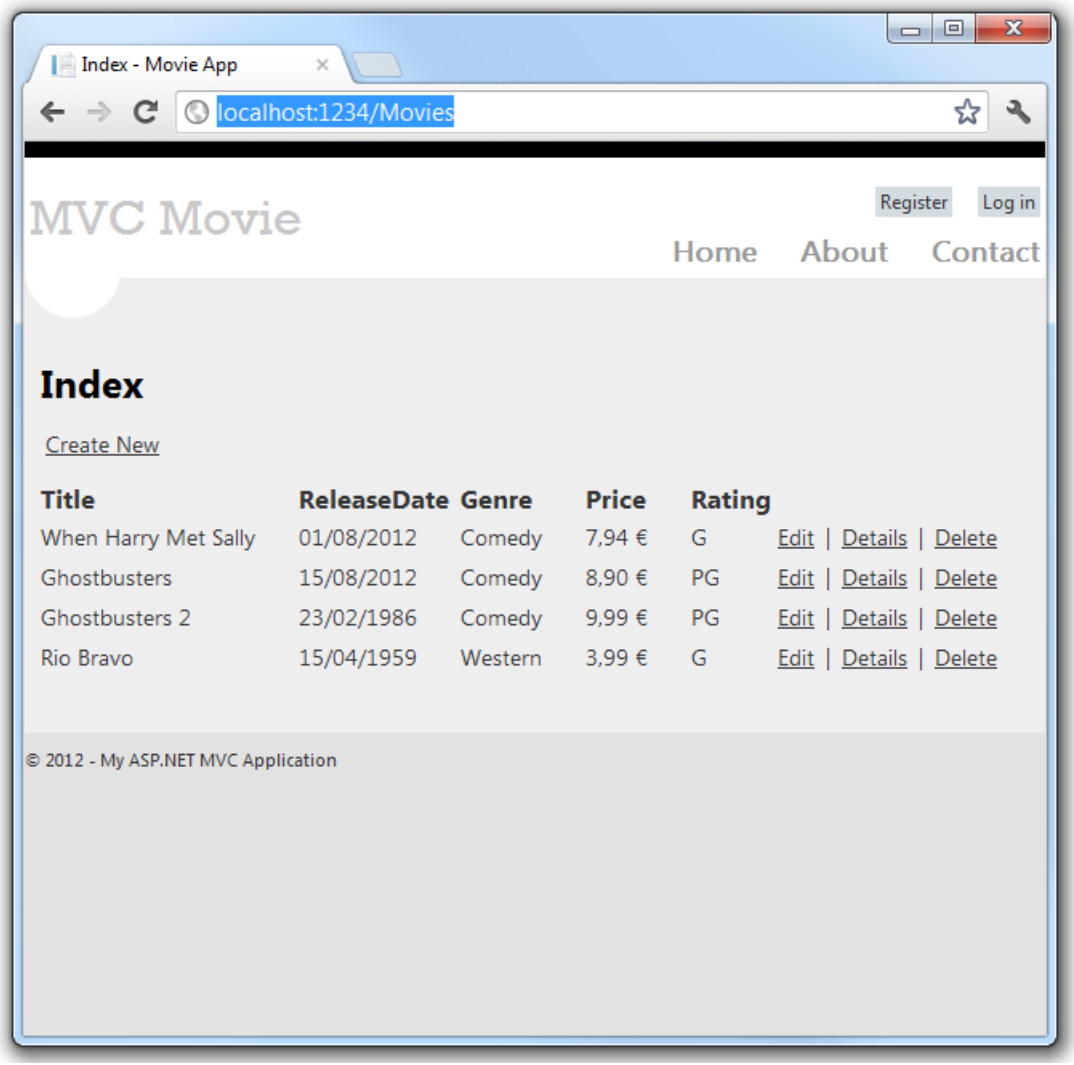
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Required]
    public string Genre { get; set; }

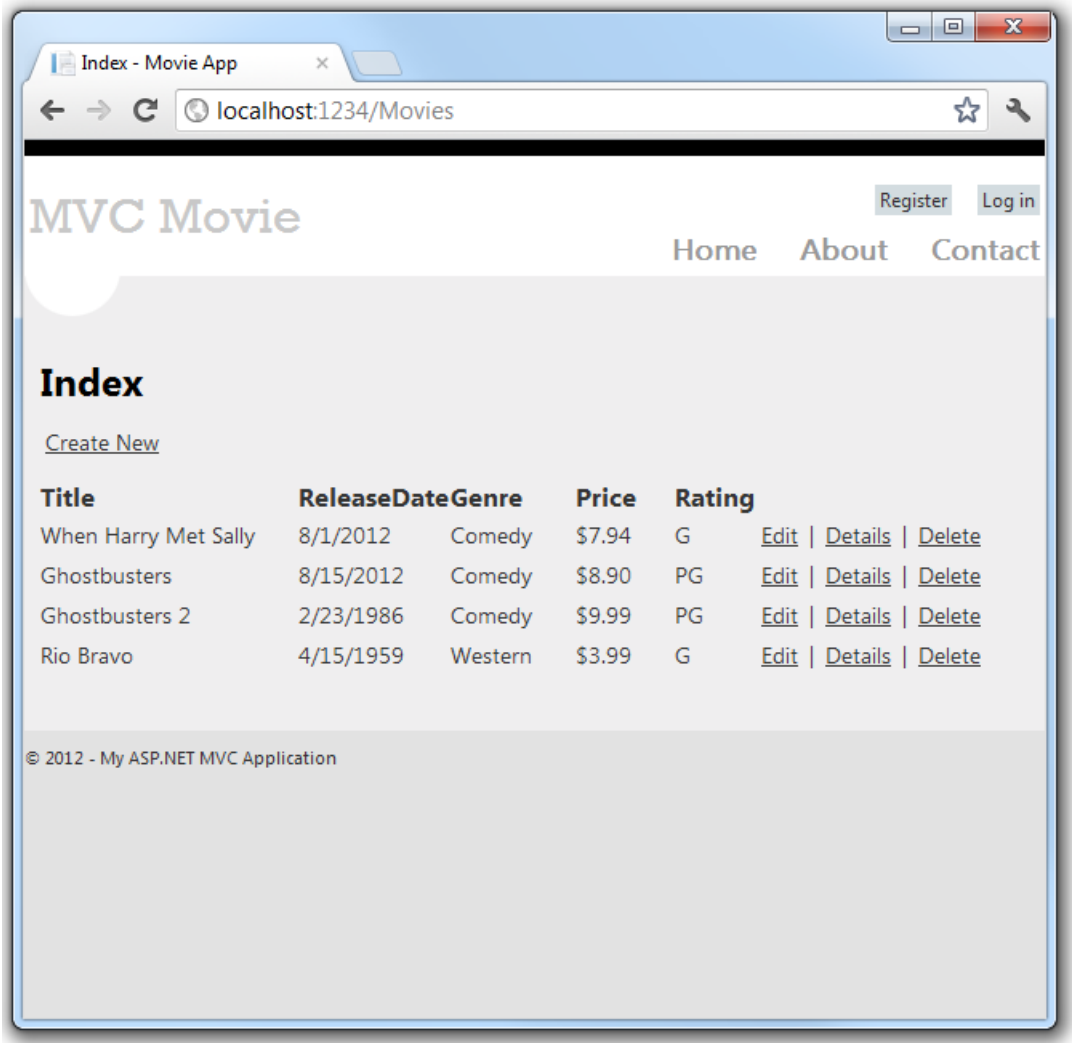
    [Range(1, 100)]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [StringLength(5)]
    public string Rating { get; set; }
}
```

Run the application and browse to the `Movies` controller. The release date and price are nicely formatted. The image below shows the release date and price using "fr-FR" as the culture.



The image below shows the same data displayed with the default culture (English US).



In the next part of the series, we'll review the application and make some improvements to the automatically generated `Details` and `Delete` methods.

PREVIOUS NEXT

Examining the Details and Delete Methods

2/20/2020 • 4 minutes to read • Edit Online

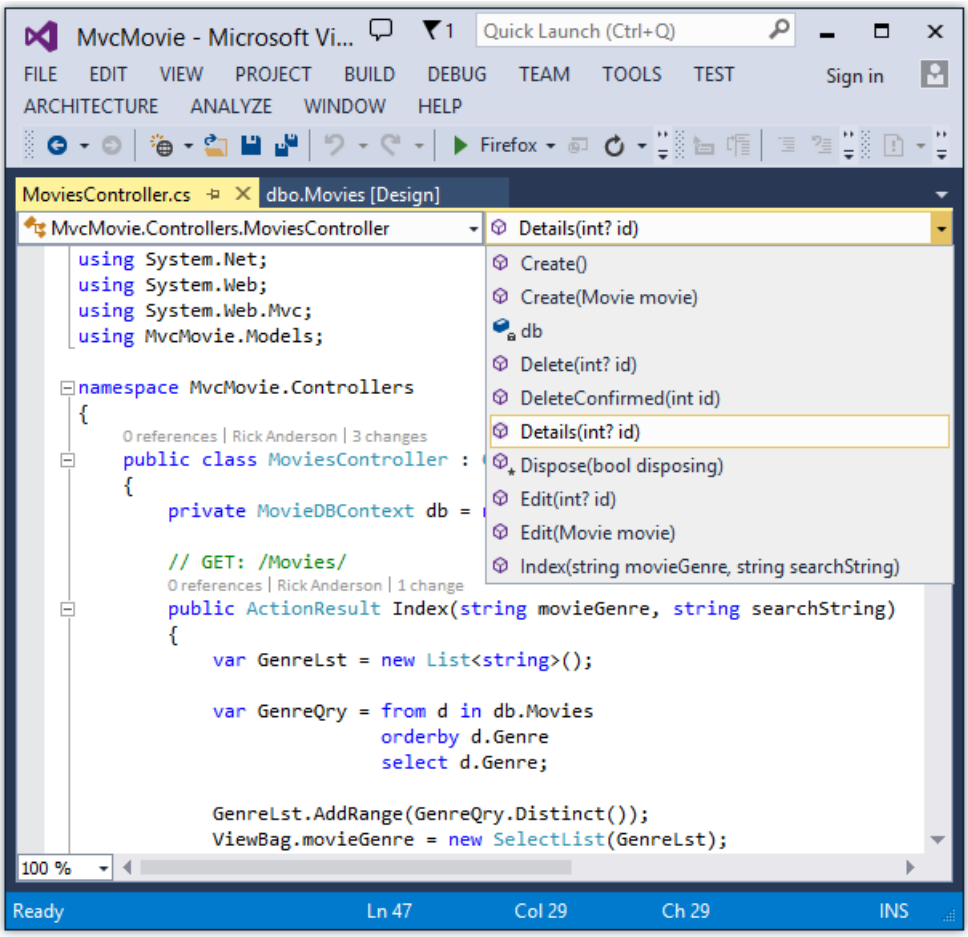
by Rick Anderson

> **NOTE**
>
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

In this part of the tutorial, you'll examine the automatically generated `Details` and `Delete` methods.

Examining the Details and Delete Methods

Open the `Movie` controller and examine the `Details` method.



```
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}
```

The MVC scaffolding engine that created this action method adds a comment showing a HTTP request that invokes the method. In this case it's a `GET` request with three URL segments, the `Movies` controller, the `Details` method and a `ID` value.

Code First makes it easy to search for data using the `Find` method. An important security feature built into the method is that the code verifies that the `Find` method has found a movie before the code tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from `http://localhost:xxxx/Movies/Details/1` to something like `http://localhost:xxxx/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you did not check for a null movie, a null movie would result in a database error.

Examine the `Delete` and `DeleteConfirmed` methods.

```
// GET: /Movies/Delete/5
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}

// POST: /Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
    Movie movie = db.Movies.Find(id);
    db.Movies.Remove(movie);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

Note that the `HTTP Get``Delete` method doesn't delete the specified movie, it returns a view of the movie where you can submit (`HttpPost`) the deletion.. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole. For more information about this, see Stephen Walther's blog entry ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes.

The `HttpPost` method that deletes the data is named `DeleteConfirmed` to give the HTTP POST method a unique

signature or name. The two method signatures are shown below:

```
// GET: /Movies/Delete/5
public ActionResult Delete(int? id)

//
// POST: /Movies/Delete/5
[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
```

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two Delete methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

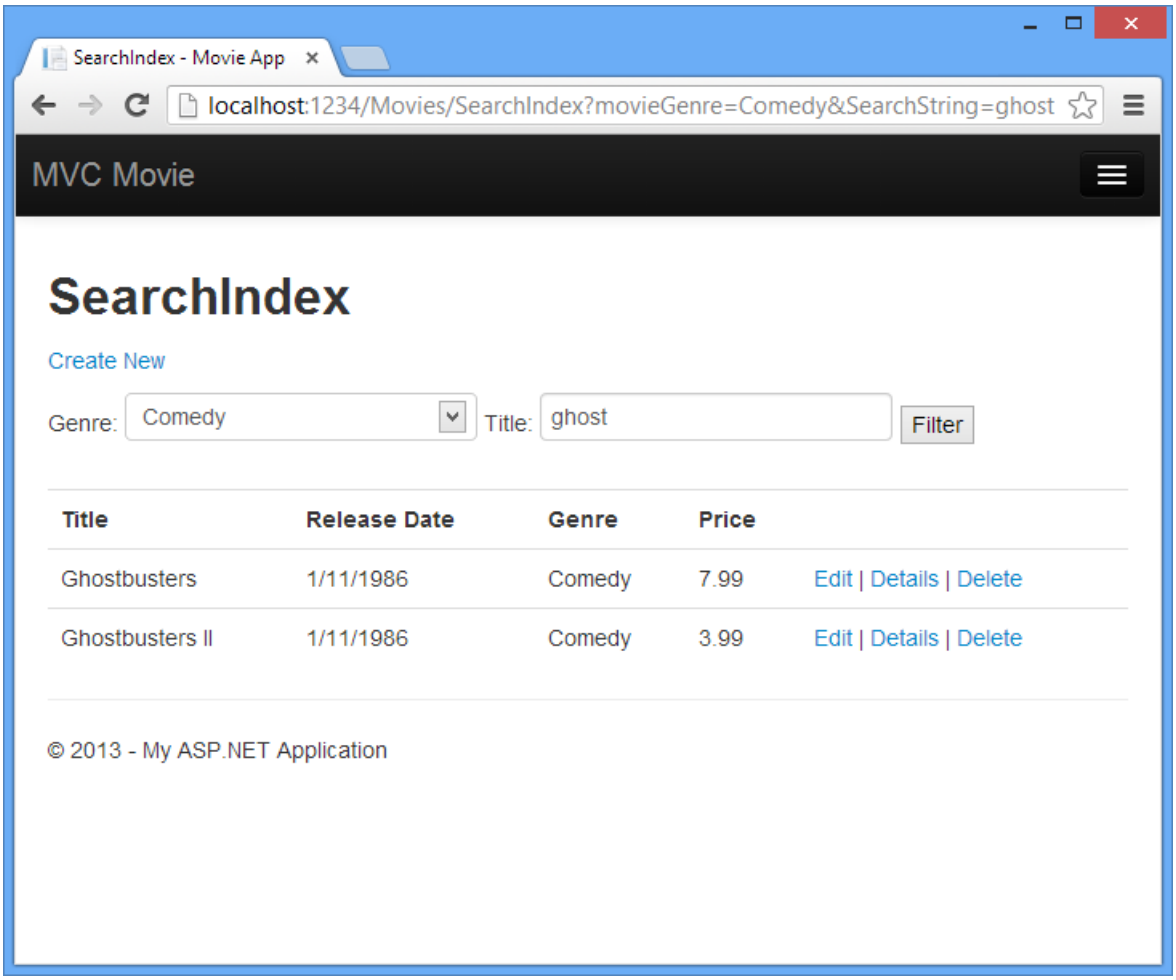
To sort this out, you can do a couple of things. One is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. This effectively performs mapping for the routing system so that a URL that includes */Delete/* for a POST request will find the `DeleteConfirmed` method.

Another common way to avoid a problem with methods that have identical names and signatures is to artificially change the signature of the POST method to include an unused parameter. For example, some developers add a parameter type `FormCollection` that is passed to the POST method, and then simply don't use the parameter:

```
public ActionResult Delete(FormCollection fcNotUsed, int id = 0)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    db.Movies.Remove(movie);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

Summary

You now have a complete ASP.NET MVC application that stores data in a local DB database. You can create, read, update, delete, and search for movies.



Next Steps

After you have built and tested a web application, the next step is to make it available to other people to use over the Internet. To do that, you have to deploy it to a web hosting provider. Microsoft offers free web hosting for up to 10 web sites in a free Windows Azure trial account. I suggest you next follow my tutorial Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL Database to a Windows Azure Web Site. An excellent tutorial is Tom Dykstra's intermediate-level Creating an Entity Framework Data Model for an ASP.NET MVC Application. Stackoverflow and the ASP.NET MVC forums are a great places to ask questions. Follow me on twitter so you can get updates on my latest tutorials.

Feedback is welcome.

— Rick Anderson twitter: @RickAndMSFT
— Scott Hanselman twitter: @shanselman

PREVIOUS

Getting Started with EF 5 using MVC 4

2/20/2020 • 2 minutes to read • Edit Online

The basics of using Entity Framework 5 to display and edit data in an ASP.NET MVC 4 application.

- Creating an Entity Framework Data Model
- Implementing Basic CRUD Functionality
- Sorting, Filtering, and Paging
- Creating a More Complex Data Model
- Reading Related Data
- Updating Related Data
- Handling Concurrency
- Implementing Inheritance
- Implementing the Repository and Unit of Work Patterns
- Advanced Entity Framework Scenarios
- Building the Chapter Downloads

Creating an Entity Framework Data Model for an ASP.NET MVC Application (1 of 10)

2/20/2020 • 23 minutes to read • Edit Online

by Tom Dykstra

Download Completed Project

> **NOTE**
>
> A newer version of this tutorial series is available, for Visual Studio 2013, Entity Framework 6, and MVC 5.

The Contoso University sample web application demonstrates how to create ASP.NET MVC 4 applications using the Entity Framework 5 and Visual Studio 2012. The sample application is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This tutorial series explains how to build the Contoso University sample application. You can download the completed application.

Code First

There are three ways you can work with data in the Entity Framework: *Database First*, *Model First*, and *Code First*. This tutorial is for Code First. For information about the differences between these workflows and guidance on how to choose the best one for your scenario, see Entity Framework Development Workflows.

MVC

The sample application is built on ASP.NET MVC. If you prefer to work with the ASP.NET Web Forms model, see the Model Binding and Web Forms tutorial series and ASP.NET Data Access Content Map.

Software versions

SHOWN IN THE TUTORIAL	ALSO WORKS WITH
Windows 8	Windows 7
Visual Studio 2012	Visual Studio 2012 Express for Web. This is automatically installed by the Windows Azure SDK if you don't already have VS 2012 or VS 2012 Express for Web. Visual Studio 2013 should work, but the tutorial has not been tested with it, and some menu selections and dialog boxes are different. The VS 2013 version of the Windows Azure SDK is required for Windows Azure deployment.
.NET 4.5	Most of the features shown will work in .NET 4, but some won't. For example, enum support in EF requires .NET 4.5.
Entity Framework 5	

SHOWN IN THE TUTORIAL	ALSO WORKS WITH
Windows Azure SDK 2.1	If you skip the Windows Azure deployment steps, you don't need the SDK. When a new version of the SDK is released, the link will install the newer version. In that case, you might have to adapt some of the instructions to new UI and features.

Questions

If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET Entity Framework forum, the Entity Framework and LINQ to Entities forum, or StackOverflow.com.

Acknowledgments

See the last tutorial in the series for acknowledgments and a note about VB.

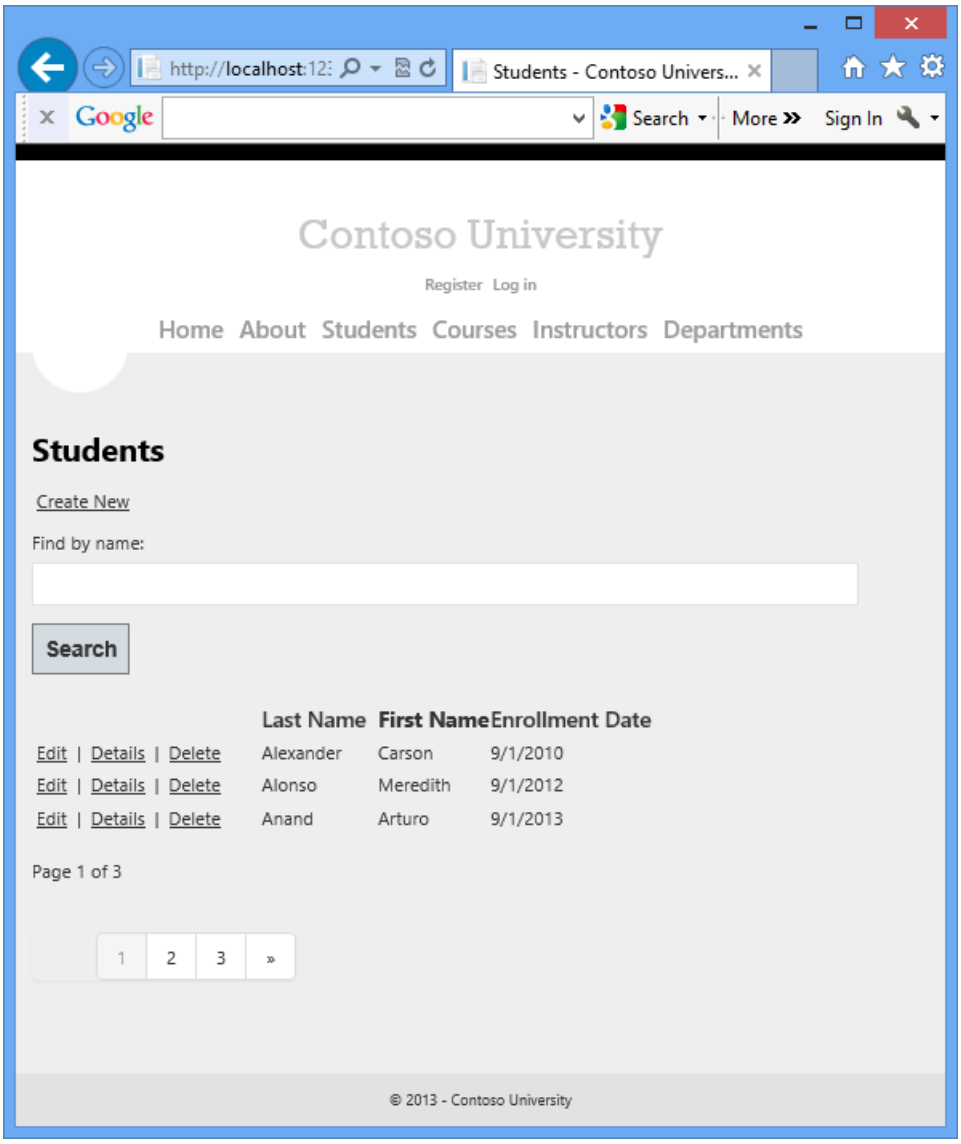
Original version of the tutorial

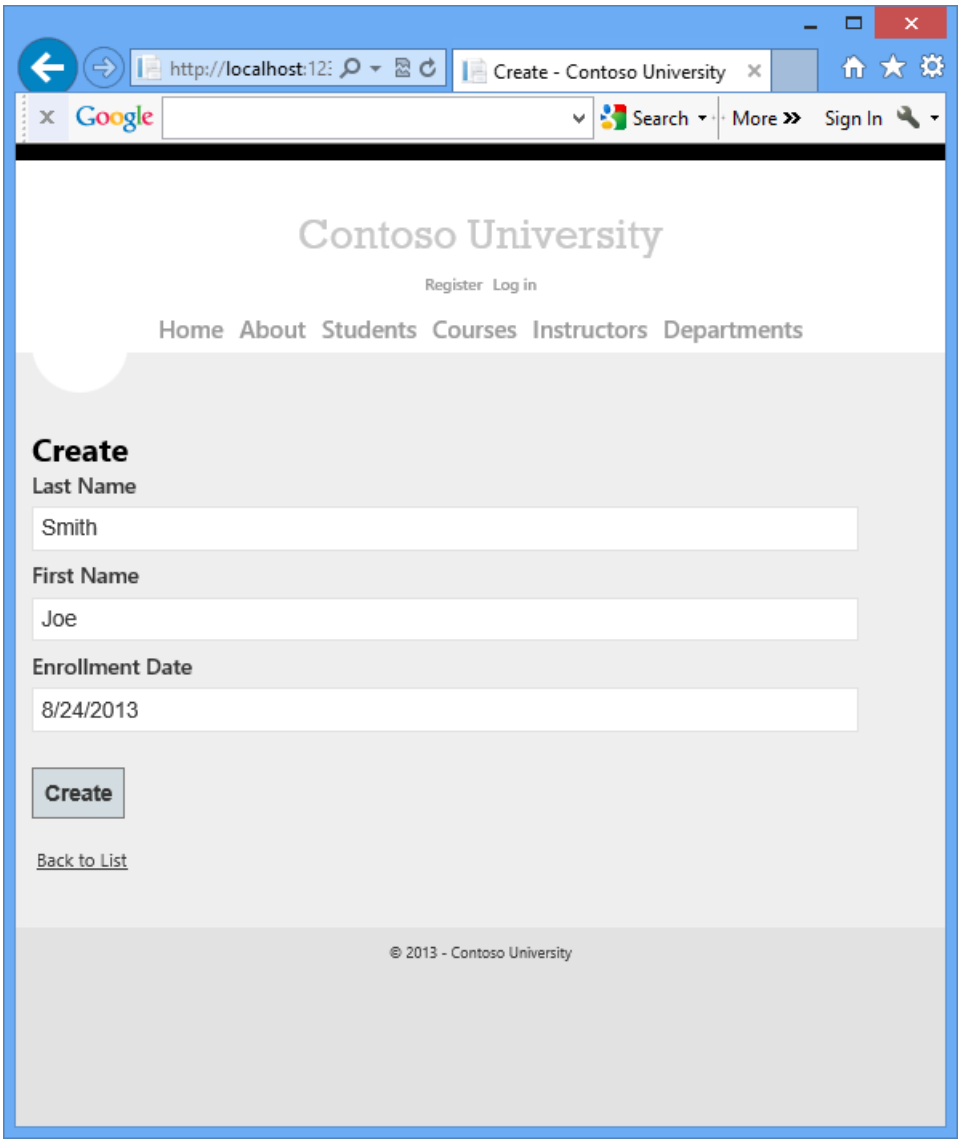
The original version of the tutorial is available in the EF 4.1 / MVC 3 e-book.

The Contoso University Web Application

The application you'll be building in these tutorials is a simple university web site.

Users can view and update student, course, and instructor information. Here are a few of the screens you'll create.


http://localhost:12:
Students - Contoso Univers...
Google
Search
More
Sign In
Contoso University
Register Log in
Home About Students Courses Instructors Departments
Students
Create New
Find by name:
Search
Last Name First Name Enrollment Date
Edit | Details | Delete Alexander Carson 9/1/2010
Edit | Details | Delete Alonso Meredith 9/1/2012
Edit | Details | Delete Anand Arturo 9/1/2013
Page 1 of 3
1 2 3 »
© 2013 - Contoso University



The UI style of this site has been kept close to what's generated by the built-in templates, so that the tutorial can focus mainly on how to use the Entity Framework.

Prerequisites

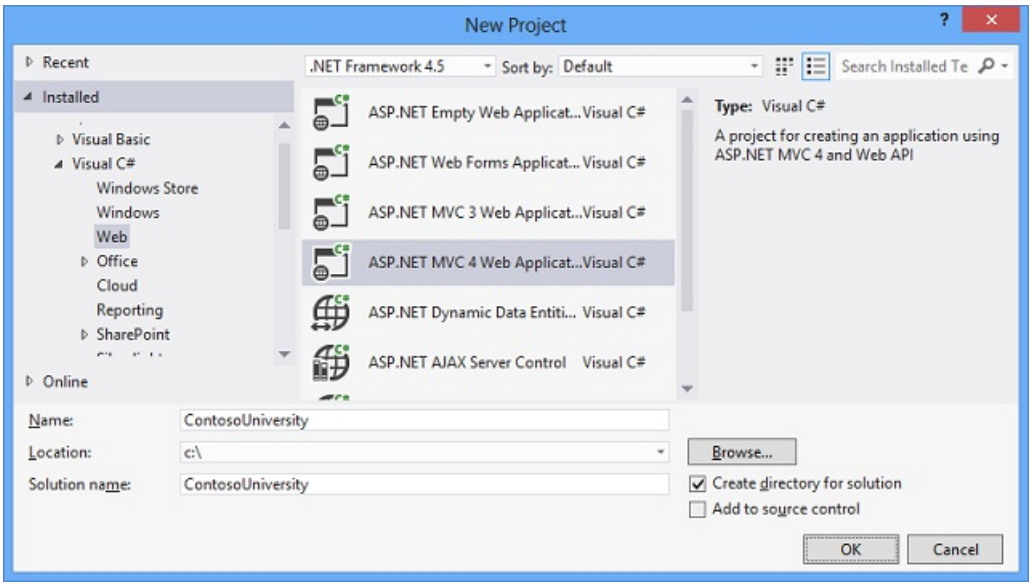
The directions and screen shots in this tutorial assume that you're using Visual Studio 2012 or Visual Studio 2012 Express for Web, with the latest update and Azure SDK for .NET installed as of July, 2013. You can get all of this with the following link:

Azure SDK for .NET (Visual Studio 2012)

If you have Visual Studio installed, the link above will install any missing components. If you don't have Visual Studio, the link will install Visual Studio 2012 Express for Web. You can use Visual Studio 2013, but some of the required procedures and screens will differ.

Create an MVC Web Application

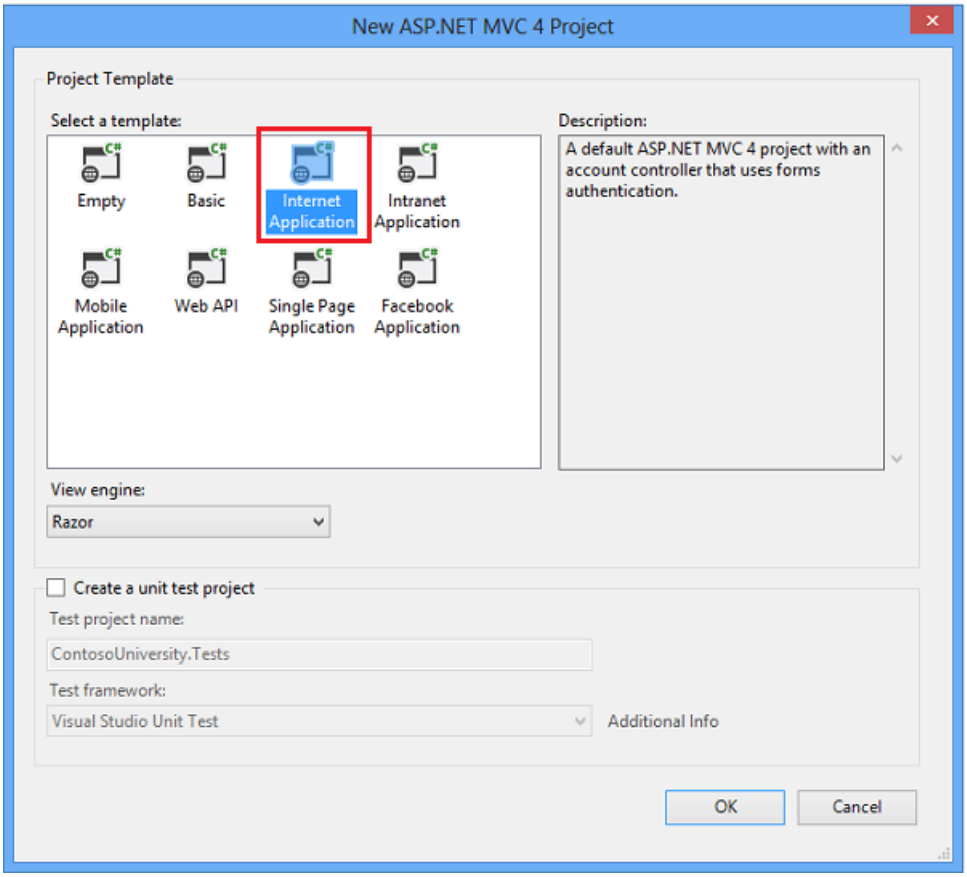
Open Visual Studio and create a new C# project named "ContosoUniversity" using the **ASP.NET MVC 4 Web Application** template. Make sure you target **.NET Framework 4.5** (you'll be using `enum` properties, and that requires .NET 4.5).



In the **New ASP.NET MVC 4 Project** dialog box select the **Internet Application** template.

Leave the **Razor** view engine selected, and leave the **Create a unit test project** check box cleared.

Click **OK**.



Set Up the Site Style

A few simple changes will set up the site menu, layout, and home page.

Open *Views\Shared_Layout.cshtml*, and replace the contents of the file with the following code. The changes are highlighted.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>@ViewBag.Title - Contoso University</title>
        <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />
        <meta name="viewport" content="width=device-width" />
        @Styles.Render("~/Content/css")
        @Scripts.Render("~/bundles/modernizr")
    </head>
    <body>
        <header>
            <div class="content-wrapper">
                <div class="float-left">
                    <p class="site-title">@Html.ActionLink("Contoso University", "Index", "Home")</p>
                </div>
                <div class="float-right">
                    <section id="login">
                        @Html.Partial("_LoginPartial")
                    </section>
                    <nav>
                        <ul id="menu">
                            <li>@Html.ActionLink("Home", "Index", "Home")</li>
                            <li>@Html.ActionLink("About", "About", "Home")</li>
                            <li>@Html.ActionLink("Students", "Index", "Student")</li>
                            <li>@Html.ActionLink("Courses", "Index", "Course")</li>
                            <li>@Html.ActionLink("Instructors", "Index", "Instructor")</li>
                            <li>@Html.ActionLink("Departments", "Index", "Department")</li>
                        </ul>
                    </nav>
                </div>
            </div>
        </header>
        <div id="body">
            @RenderSection("featured", required: false)
            <section class="content-wrapper main-content clear-fix">
                @RenderBody()
            </section>
        </div>
        <footer>
            <div class="content-wrapper">
                <div class="float-left">
                    <p>&copy; @DateTime.Now.Year - Contoso University</p>
                </div>
            </div>
        </footer>

        @Scripts.Render("~/bundles/jquery")
        @RenderSection("scripts", required: false)
    </body>
</html>
```

This code makes the following changes:

- Replaces the template instances of "My ASP.NET MVC Application" and "your logo here" with "Contoso University".
- Adds several action links that will be used later in the tutorial.

In *Views\Home\Index.cshtml*, replace the contents of the file with the following code to eliminate the template paragraphs about ASP.NET and MVC:

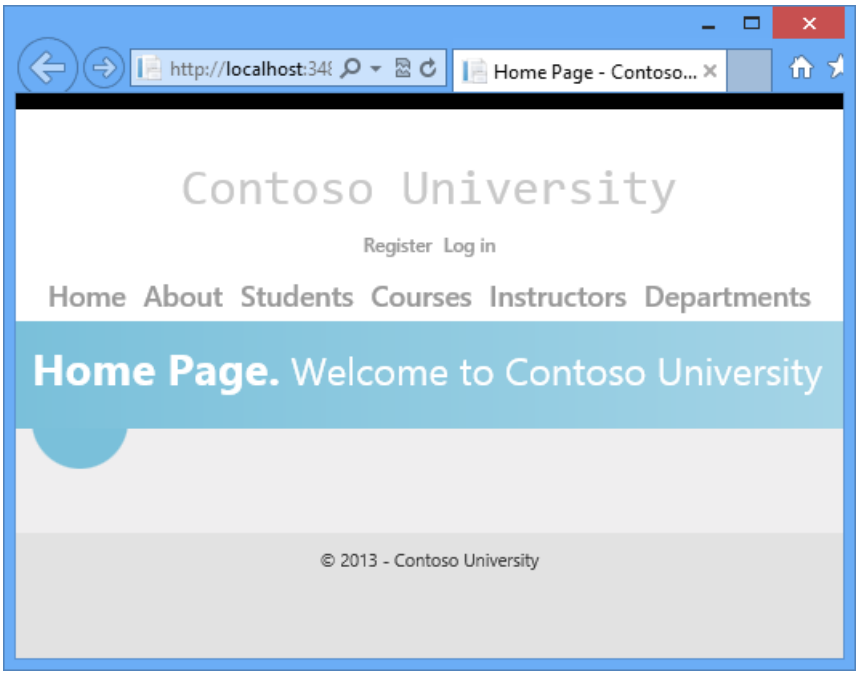
```
@{
    ViewBag.Title = "Home Page";
}
@section featured {
    <section class="featured">
        <div class="content-wrapper">
            <hgroup class="title">
                <h1>@ViewBag.Title.</h1>
                <h2>@ViewBag.Message</h2>
            </hgroup>
        </div>
    </section>
}
```

In *Controllers\HomeController.cs*, change the value for `ViewBag.Message` in the `Index` Action method to "Welcome to Contoso University!", as shown in the following example:

```
public ActionResult Index()
{
    ViewBag.Message = "Welcome to Contoso University";

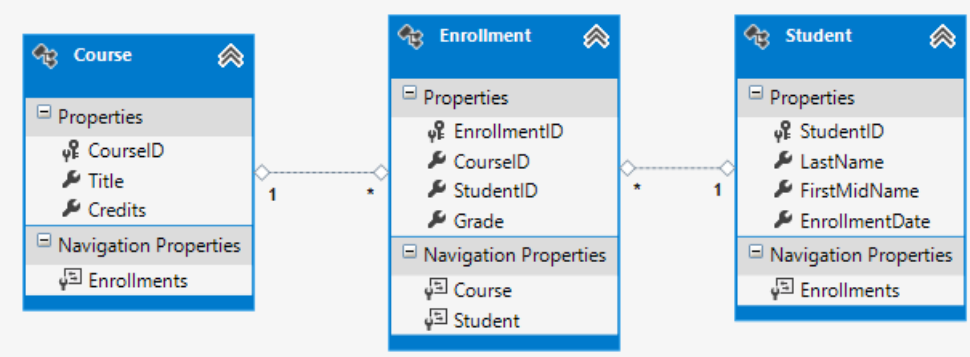
    return View();
}
```

Press CTRL+F5 to run the site. You see the home page with the main menu.



Create the Data Model

Next you'll create entity classes for the Contoso University application. You'll start with the following three entities:

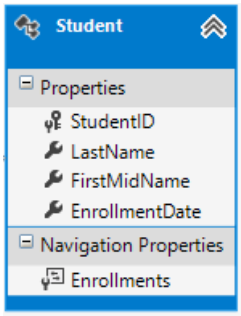


There's a one-to-many relationship between `Student` and `Enrollment` entities, and there's a one-to-many relationship between `Course` and `Enrollment` entities. In other words, a student can be enrolled in any number of courses, and a course can have any number of students enrolled in it.

In the following sections you'll create a class for each one of these entities.

> **NOTE**
>
> If you try to compile the project before you finish creating all of these entity classes, you'll get compiler errors.

The Student Entity



In the *Models* folder, create *Student.cs* and replace the existing code with the following code:

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int StudentID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

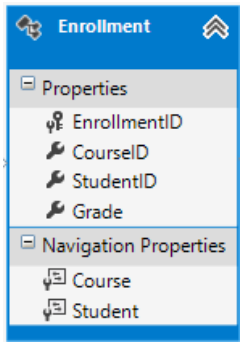
The `StudentID` property will become the primary key column of the database table that corresponds to this class. By default, the Entity Framework interprets a property that's named `ID` or *classname* `ID` as the primary key.

The `Enrollments` property is a *navigation property*. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity will hold all of the `Enrollment` entities that are related to that `Student` entity. In other words, if a given `Student` row in the database has two related `Enrollment` rows (rows that contain that student's primary key value in their `StudentID` foreign key column), that `Student` entity's `Enrollments` navigation property will contain those two `Enrollment` entities.

Navigation properties are typically defined as `virtual` so that they can take advantage of certain Entity Framework functionality such as *lazy loading*. (Lazy loading will be explained later, in the Reading Related Data tutorial later in this series.

If a navigation property can hold multiple entities (as in many-to-many or one-to-many relationships), its type must be a list in which entries can be added, deleted, and updated, such as `ICollection`.

The Enrollment Entity



In the *Models* folder, create *Enrollment.cs* and replace the existing code with the following code:

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

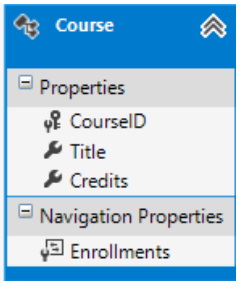
        public virtual Course Course { get; set; }
        public virtual Student Student { get; set; }
    }
}
```

The Grade property is an enum. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade — null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property can only hold a single `Student` entity (unlike the `Student.Enrollments` navigation property you saw earlier, which can hold multiple `Enrollment` entities).

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

The Course Entity



In the *Models* folder, create *Course.cs*, replacing the existing code with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

We'll say more about the [DatabaseGenerated(DatabaseGeneratedOption.None)] attribute in the next tutorial. Basically, this attribute lets you enter the primary key for the course rather than having the database generate it.

Create the Database Context

The main class that coordinates Entity Framework functionality for a given data model is the *database context* class. You create this class by deriving from the System.Data.Entity.DbContext class. In your code you specify which entities are included in the data model. You can also customize certain Entity Framework behavior. In this project, the class is named `SchoolContext`.

Create a folder named *DAL* (for Data Access Layer). In that folder create a new class file named *SchoolContext.cs*, and replace the existing code with the following code:

```
using ContosoUniversity.Models;
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace ContosoUniversity.DAL
{
    public class SchoolContext : DbContext
    {
        public DbSet<Student> Students { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Course> Courses { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
        }
    }
}
```

This code creates a DbSet property for each entity set. In Entity Framework terminology, an *entity set* typically corresponds to a database table, and an *entity* corresponds to a row in the table.

The `modelBuilder.Conventions.Remove` statement in the OnModelCreating method prevents table names from being pluralized. If you didn't do this, the generated tables would be named `Students`, `Courses`, and `Enrollments`. Instead, the table names will be `Student`, `Course`, and `Enrollment`. Developers disagree about whether table names should be pluralized or not. This tutorial uses the singular form, but the important point is

that you can select whichever form you prefer by including or omitting this line of code.

SQL Server Express LocalDB

LocalDB is a lightweight version of the SQL Server Express Database Engine that starts on demand and runs in user mode. LocalDB runs in a special execution mode of SQL Server Express that enables you to work with databases as *.mdf* files. Typically, LocalDB database files are kept in the *App_Data* folder of a web project. The user instance feature in SQL Server Express also enables you to work with *.mdf* files, but the user instance feature is deprecated; therefore, LocalDB is recommended for working with *.mdf* files.

Typically SQL Server Express is not used for production web applications. LocalDB in particular is not recommended for production use with a web application because it is not designed to work with IIS.

In Visual Studio 2012 and later versions, LocalDB is installed by default with Visual Studio. In Visual Studio 2010 and earlier versions, SQL Server Express (without LocalDB) is installed by default with Visual Studio; you have to install it manually if you're using Visual Studio 2010.

In this tutorial you'll work with LocalDB so that the database can be stored in the *App_Data* folder as an *.mdf* file. Open the root *Web.config* file and add a new connection string to the `connectionStrings` collection, as shown in the following example. (Make sure you update the *Web.config* file in the root project folder. There's also a *Web.config* file is in the *Views* subfolder that you don't need to update.)

```
<add name="SchoolContext" connectionString="Data Source=(LocalDb)\v11.0;Initial
Catalog=ContosoUniversity;Integrated Security=SSPI;AttachDBFilename=|DataDirectory|\ContosoUniversity.mdf"
providerName="System.Data.SqlClient" />
```

By default, the Entity Framework looks for a connection string named the same as the `DbContext` class (`SchoolContext` for this project). The connection string you've added specifies a LocalDB database named *ContosoUniversity.mdf* located in the *App_Data* folder. For more information, see SQL Server Connection Strings for ASP.NET Web Applications.

You don't actually need to specify the connection string. If you don't supply a connection string, Entity Framework will create one for you; however, the database might not be in the *App_data* folder of your app. For information on where the database will be created, see Code First to a New Database.

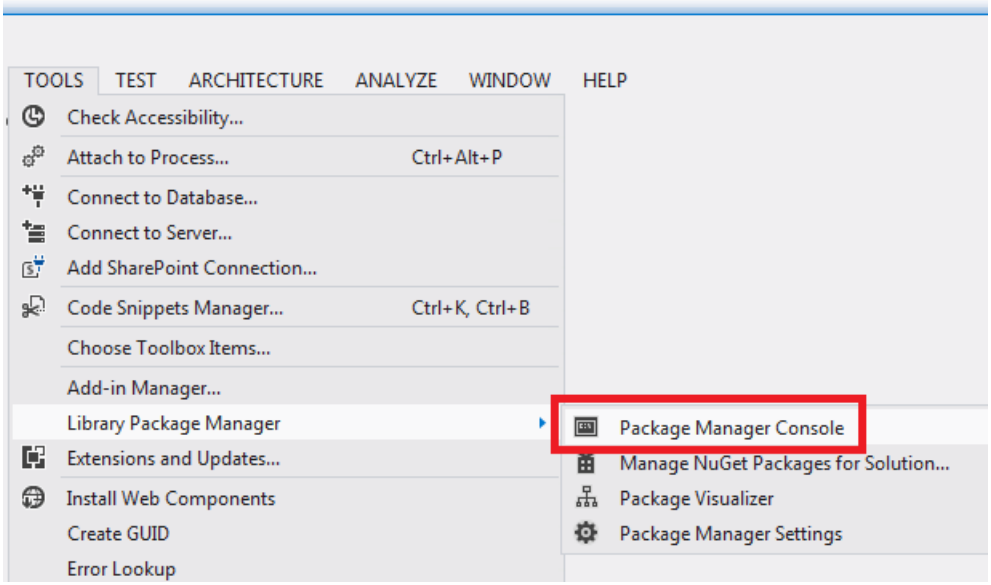
The `connectionStrings` collection also has a connection string named `DefaultConnection` which is used for the membership database. You won't be using the membership database in this tutorial. The only difference between the two connection strings is the database name and the name attribute value.

Set up and Execute a Code First Migration

When you first start to develop an application, your data model changes frequently, and each time the model changes it gets out of sync with the database. You can configure the Entity Framework to automatically drop and re-create the database each time you change the data model. This is not a problem early in development because test data is easily re-created, but after you have deployed to production you usually want to update the database schema without dropping the database. The Migrations feature enables Code First to update the database without dropping and re-creating it. Early in the development cycle of a new project you might want to use DropCreateDatabaseIfModelChanges to drop, recreate and re-seed the database each time the model changes. One you get ready to deploy your application, you can convert to the migrations approach. For this tutorial you'll only use migrations. For more information, see Code First Migrations and Migrations Screencast Series.

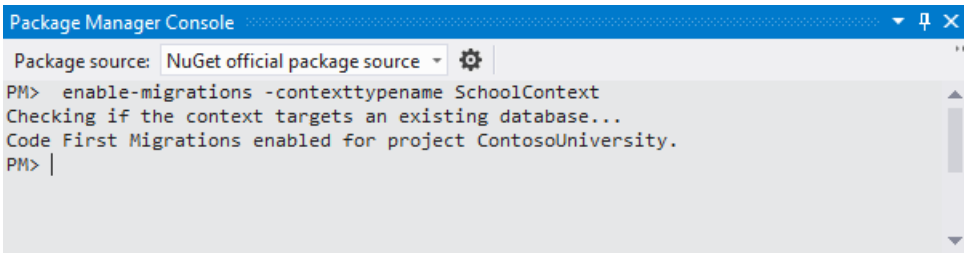
Enable Code First Migrations

1. From the **Tools** menu, click **NuGet Package Manager** and then **Package Manager Console**.

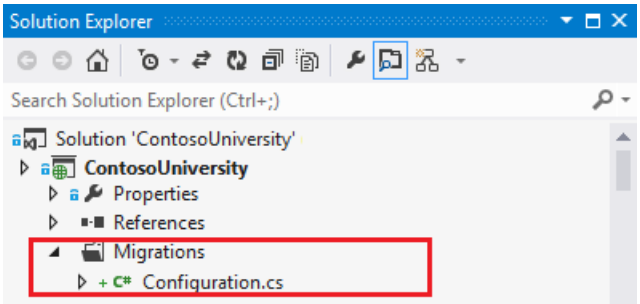


2. At the `PM>` prompt enter the following command:

```
enable-migrations -contexttypename SchoolContext
```



This command creates a *Migrations* folder in the ContosoUniversity project, and it puts in that folder a *Configuration.cs* file that you can edit to configure Migrations.



The `Configuration` class includes a `Seed` method that is called when the database is created and every time it is updated after a data model change.

```
internal sealed class Configuration :
DbMigrationsConfiguration<ContosoUniversity.Models.SchoolContext>
{
    public Configuration()
    {
        AutomaticMigrationsEnabled = false;
    }

    protected override void Seed(ContosoUniversity.Models.SchoolContext context)
    {
        //  This method will be called after migrating to the latest version.

        //  You can use the DbSet<T>.AddOrUpdate() helper extension method
        //  to avoid creating duplicate seed data. E.g.
        //
        //    context.People.AddOrUpdate(
        //      p => p.FullName,
        //      new Person { FullName = "Andrew Peters" },
        //      new Person { FullName = "Brice Lambson" },
        //      new Person { FullName = "Rowan Miller" }
        //    );
        //
    }
}
```

The purpose of this `Seed` method is to enable you to insert test data into the database after Code First creates it or updates it.

Set up the Seed Method

The Seed method runs when Code First Migrations creates the database and every time it updates the database to the latest migration. The purpose of the Seed method is to enable you to insert data into your tables before the application accesses the database for the first time.

In earlier versions of Code First, before Migrations was released, it was common for `Seed` methods to insert test data, because with every model change during development the database had to be completely deleted and re-created from scratch. With Code First Migrations, test data is retained after database changes, so including test data in the Seed method is typically not necessary. In fact, you don't want the `Seed` method to insert test data if you'll be using Migrations to deploy the database to production, because the `Seed` method will run in production. In that case you want the `Seed` method to insert into the database only the data that you want to be inserted in production. For example, you might want the database to include actual department names in the `Department` table when the application becomes available in production.

For this tutorial, you'll be using Migrations for deployment, but your `Seed` method will insert test data anyway in order to make it easier to see how application functionality works without having to manually insert a lot of data.

1. Replace the contents of the *Configuration.cs* file with the following code, which will load test data into the new database.

```
namespace ContosoUniversity.Migrations
{
   using System;
   using System.Collections.Generic;
   using System.Data.Entity.Migrations;
   using System.Linq;
   using ContosoUniversity.Models;

   internal sealed class Configuration :
DbMigrationsConfiguration<ContosoUniversity.DAL.SchoolContext>
   {
      public Configuration()
      {
```

```
            AutomaticMigrationsEnabled = false;
        }

        protected override void Seed(ContosoUniversity.DAL.SchoolContext context)
        {
            var students = new List<Student>
                {
                    new Student { FirstMidName = "Carson",   LastName = "Alexander",
                        EnrollmentDate = DateTime.Parse("2010-09-01") },
                    new Student { FirstMidName = "Meredith", LastName = "Alonso",
                        EnrollmentDate = DateTime.Parse("2012-09-01") },
                    new Student { FirstMidName = "Arturo",   LastName = "Anand",
                        EnrollmentDate = DateTime.Parse("2013-09-01") },
                    new Student { FirstMidName = "Gytis",    LastName = "Barzdukas",
                        EnrollmentDate = DateTime.Parse("2012-09-01") },
                    new Student { FirstMidName = "Yan",      LastName = "Li",
                        EnrollmentDate = DateTime.Parse("2012-09-01") },
                    new Student { FirstMidName = "Peggy",    LastName = "Justice",
                        EnrollmentDate = DateTime.Parse("2011-09-01") },
                    new Student { FirstMidName = "Laura",    LastName = "Norman",
                        EnrollmentDate = DateTime.Parse("2013-09-01") },
                    new Student { FirstMidName = "Nino",     LastName = "Olivetto",
                        EnrollmentDate = DateTime.Parse("2005-08-11") }
                };
            students.ForEach(s => context.Students.AddOrUpdate(p => p.LastName, s));
            context.SaveChanges();

            var courses = new List<Course>
                {
                    new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3, },
                    new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3, },
                    new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3, },
                    new Course {CourseID = 1045, Title = "Calculus",       Credits = 4, },
                    new Course {CourseID = 3141, Title = "Trigonometry",   Credits = 4, },
                    new Course {CourseID = 2021, Title = "Composition",    Credits = 3, },
                    new Course {CourseID = 2042, Title = "Literature",     Credits = 4, }
                };
            courses.ForEach(s => context.Courses.AddOrUpdate(p => p.Title, s));
            context.SaveChanges();

            var enrollments = new List<Enrollment>
                {
                    new Enrollment {
                        StudentID = students.Single(s => s.LastName == "Alexander").StudentID,
                        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
                        Grade = Grade.A
                    },
                     new Enrollment {
                        StudentID = students.Single(s => s.LastName == "Alexander").StudentID,
                        CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
                        Grade = Grade.C
                     },
                     new Enrollment {
                        StudentID = students.Single(s => s.LastName == "Alexander").StudentID,
                        CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
                        Grade = Grade.B
                     },
                     new Enrollment {
                         StudentID = students.Single(s => s.LastName == "Alonso").StudentID,
                        CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
                        Grade = Grade.B
                     },
                     new Enrollment {
                         StudentID = students.Single(s => s.LastName == "Alonso").StudentID,
                        CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
                        Grade = Grade.B
                     },
                     new Enrollment {
                        StudentID = students.Single(s => s.LastName == "Alonso").StudentID,
```

```
                    CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
                    Grade = Grade.B
                 },
                 new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Anand").StudentID,
                    CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
                 },
                 new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Anand").StudentID,
                    CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
                    Grade = Grade.B
                 },
                new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Barzdukas").StudentID,
                    CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
                    Grade = Grade.B
                 },
                 new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Li").StudentID,
                    CourseID = courses.Single(c => c.Title == "Composition").CourseID,
                    Grade = Grade.B
                 },
                 new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Justice").StudentID,
                    CourseID = courses.Single(c => c.Title == "Literature").CourseID,
                    Grade = Grade.B
                 }
             };

          foreach (Enrollment e in enrollments)
          {
             var enrollmentInDataBase = context.Enrollments.Where(
                 s =>
                      s.Student.StudentID == e.StudentID &&
                      s.Course.CourseID == e.CourseID).SingleOrDefault();
             if (enrollmentInDataBase == null)
             {
                context.Enrollments.Add(e);
             }
          }
          context.SaveChanges();
       }
    }
 }
```

The Seed method takes the database context object as an input parameter, and the code in the method uses that object to add new entities to the database. For each entity type, the code creates a collection of new entities, adds them to the appropriate DbSet property, and then saves the changes to the database. It isn't necessary to call the SaveChanges method after each group of entities, as is done here, but doing that helps you locate the source of a problem if an exception occurs while the code is writing to the database.

Some of the statements that insert data use the AddOrUpdate method to perform an "upsert" operation. Because the `Seed` method runs with every migration, you can't just insert data, because the rows you are trying to add will already be there after the first migration that creates the database. The "upsert" operation prevents errors that would happen if you try to insert a row that already exists, but it ***overrides*** any changes to data that you may have made while testing the application. With test data in some tables you might not want that to happen: in some cases when you change data while testing you want your changes to remain after database updates. In that case you want to do a conditional insert operation: insert a row only if it doesn't already exist. The Seed method uses both approaches.

The first parameter passed to the AddOrUpdate method specifies the property to use to check if a row already exists. For the test student data that you are providing, the `LastName` property can be used for this purpose since each last name in the list is unique:

```
context.Students.AddOrUpdate(p => p.LastName, s)
```

This code assumes that last names are unique. If you manually add a student with a duplicate last name, you'll get the following exception the next time you perform a migration.

Sequence contains more than one element

For more information about the `AddOrUpdate` method, see Take care with EF 4.3 AddOrUpdate Method on Julie Lerman's blog.

The code that adds `Enrollment` entities doesn't use the `AddOrUpdate` method. It checks if an entity already exists and inserts the entity if it doesn't exist. This approach will preserve changes you make to an enrollment grade when migrations run. The code loops through each member of the `Enrollment` List and if the enrollment is not found in the database, it adds the enrollment to the database. The first time you update the database, the database will be empty, so it will add each enrollment.

```
foreach (Enrollment e in enrollments)
{
    var enrollmentInDataBase = context.Enrollments.Where(
        s => s.Student.StudentID == e.Student.StudentID &&
             s.Course.CourseID == e.Course.CourseID).SingleOrDefault();
    if (enrollmentInDataBase == null)
    {
        context.Enrollments.Add(e);
    }
}
```

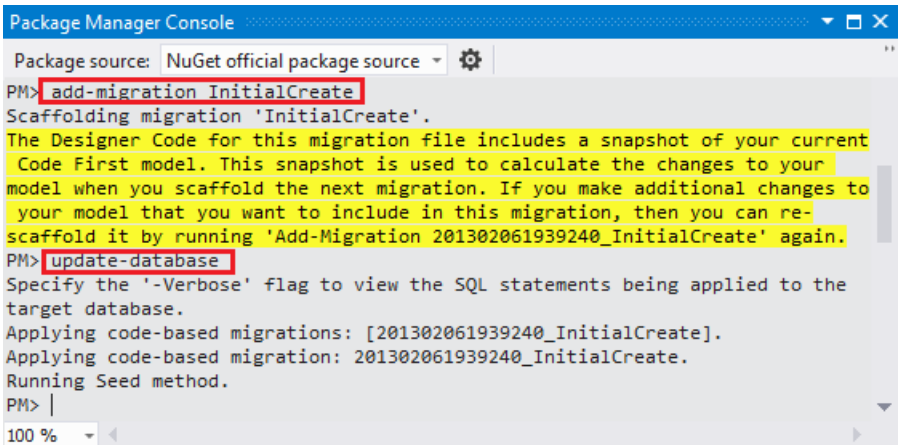
For information about how to debug the `Seed` method and how to handle redundant data such as two students named "Alexander Carson", see Seeding and Debugging Entity Framework (EF) DBs on Rick Anderson's blog.

2. Build the project.

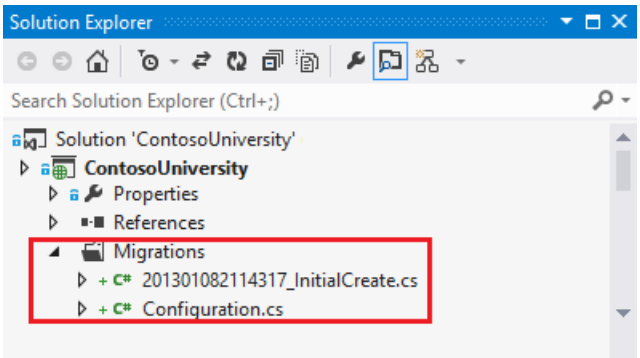
Create and Execute the First Migration

1. In the Package Manager Console window, enter the following commands:

```
add-migration InitialCreate
update-database
```



The `add-migration` command adds to the Migrations folder a *[DateStamp]_InitialCreate.cs* file that contains code which creates the database. The first parameter (`InitialCreate)` is used for the file name and can be whatever you want; you typically choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration "AddDepartmentTable".



The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them. Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the `Down` method. The following code shows the contents of the `InitialCreate` file:

```
namespace ContosoUniversity.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class InitialCreate : DbMigration
    {
        public override void Up()
        {
            CreateTable(
                "dbo.Student",
                c => new
                    {
                        StudentID = c.Int(nullable: false, identity: true),
                        LastName = c.String(),
                        FirstMidName = c.String(),
                        EnrollmentDate = c.DateTime(nullable: false),
                    })
                .PrimaryKey(t => t.StudentID);

            CreateTable(
                "dbo.Enrollment",
                c => new
                    {
                        EnrollmentID = c.Int(nullable: false, identity: true),
                        CourseID = c.Int(nullable: false),
                        StudentID = c.Int(nullable: false),
                        Grade = c.Int(),
                    })
                .PrimaryKey(t => t.EnrollmentID)
                .ForeignKey("dbo.Course", t => t.CourseID, cascadeDelete: true)
                .ForeignKey("dbo.Student", t => t.StudentID, cascadeDelete: true)
                .Index(t => t.CourseID)
                .Index(t => t.StudentID);

            CreateTable(
                "dbo.Course",
                c => new
                    {
                        CourseID = c.Int(nullable: false),
                        Title = c.String(),
                        Credits = c.Int(nullable: false),
                    })
                .PrimaryKey(t => t.CourseID);

        }

        public override void Down()
        {
            DropIndex("dbo.Enrollment", new[] { "StudentID" });
            DropIndex("dbo.Enrollment", new[] { "CourseID" });
            DropForeignKey("dbo.Enrollment", "StudentID", "dbo.Student");
            DropForeignKey("dbo.Enrollment", "CourseID", "dbo.Course");
            DropTable("dbo.Course");
            DropTable("dbo.Enrollment");
            DropTable("dbo.Student");
        }
    }
}
```

The `update-database` command runs the `Up` method to create the database and then it runs the `Seed` method to populate the database.

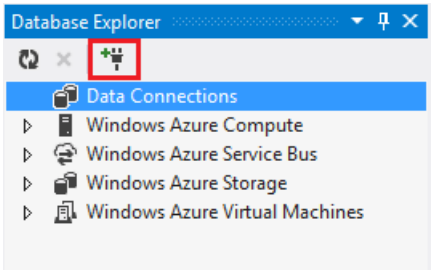
A SQL Server database has now been created for your data model. The name of the database is *ContosoUniversity*, and the *.mdf* file is in your project's *App_Data* folder because that's what you specified in your

connection string.

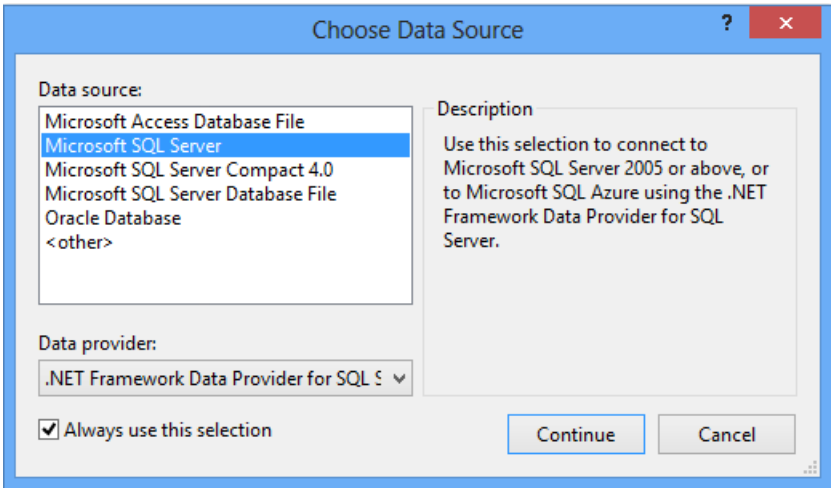
You can use either **Server Explorer** or **SQL Server Object Explorer** (SSOX) to view the database in Visual Studio. For this tutorial you'll use **Server Explorer**. In Visual Studio Express 2012 for Web, **Server Explorer** is called **Database Explorer**.

1. From the **View** menu, click **Server Explorer**.

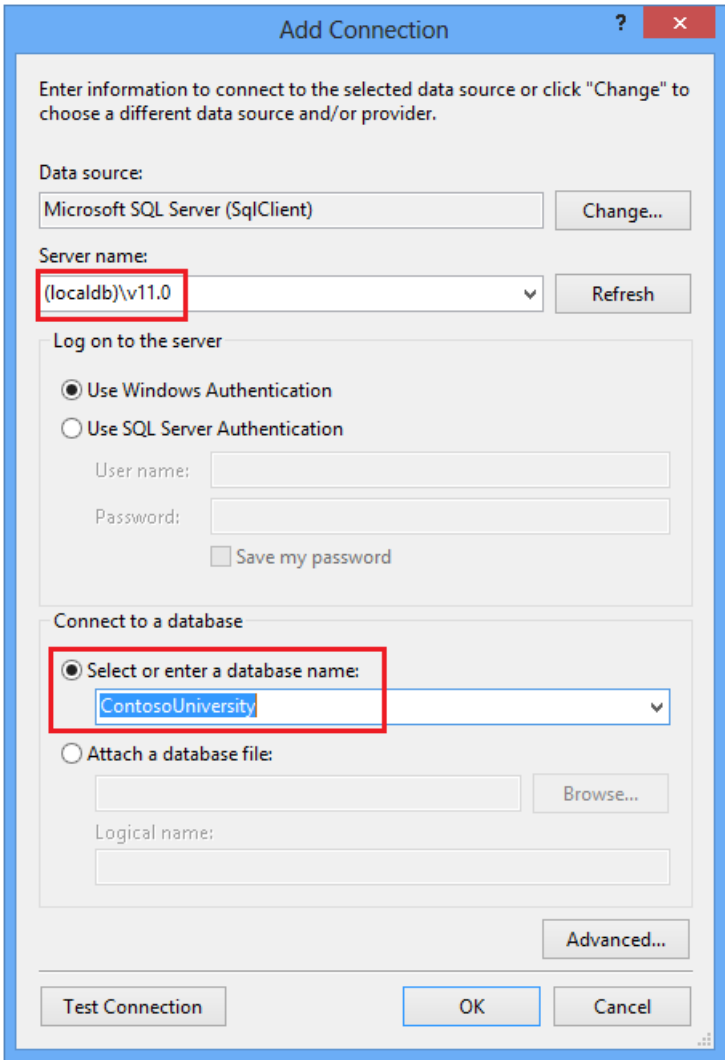
2. Click the **Add Connection** icon.



3. If you are prompted with the **Choose Data Source** dialog, click **Microsoft SQL Server**, and then click **Continue**.

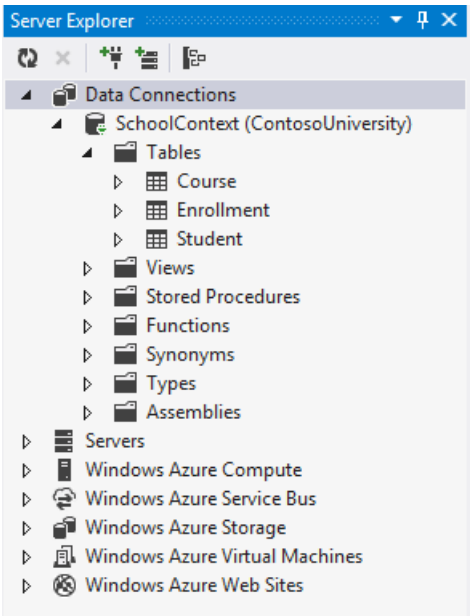


4. In the **Add Connection** dialog box, enter **(localdb)\v11.0** for the **Server Name**. Under **Select or enter a database name**, select **ContosoUniversity**.

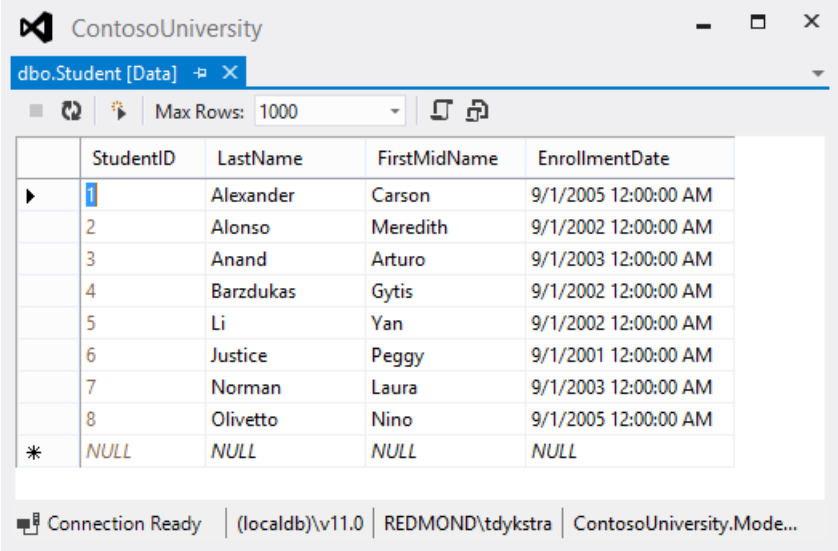


5. Click **OK**.

6. Expand **SchoolContext** and then expand **Tables**.



7. Right-click the **Student** table and click **Show Table Data** to see the columns that were created and the rows that were inserted into the table.

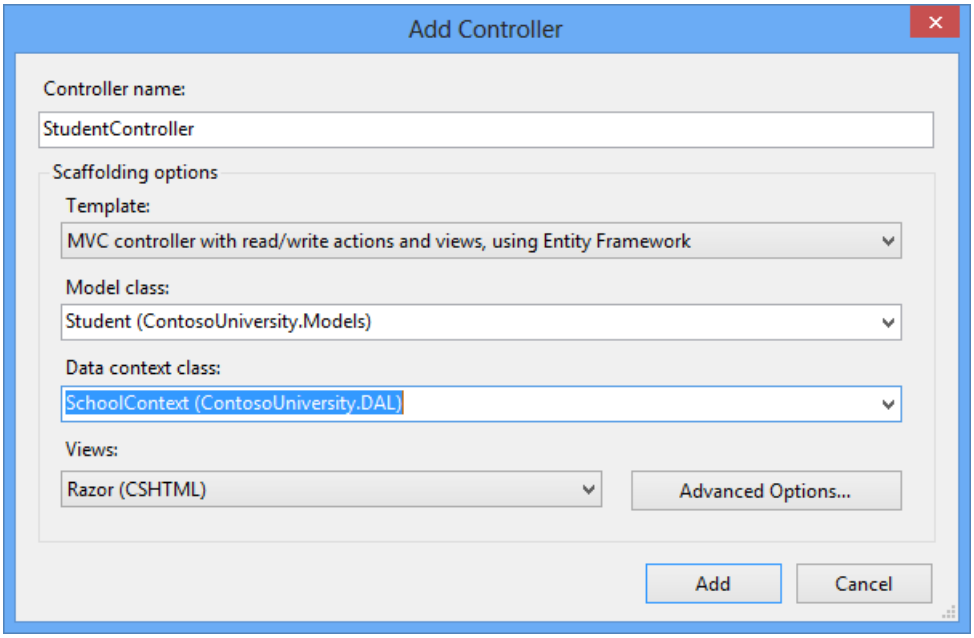


Creating a Student Controller and Views

The next step is to create an ASP.NET MVC controller and views in your application that can work with one of these tables.

1. To create a `Student` controller, right-click the **Controllers** folder in **Solution Explorer**, select **Add**, and then click **Controller**. In the **Add Controller** dialog box, make the following selections and then click **Add**:

   - Controller name: **StudentController**.
   - Template: **MVC controller with read/write actions and views, using Entity Framework**.
   - Model class: **Student (ContosoUniversity.Models)**. (If you don't see this option in the drop-down list, build the project and try again.)
   - Data context class: **SchoolContext (ContosoUniversity.Models)**.
   - Views: **Razor (CSHTML)**. (The default.)



2. Visual Studio opens the *Controllers\StudentController.cs* file. You see a class variable has been created that instantiates a database context object:

```
private SchoolContext db = new SchoolContext();
```

The `Index` action method gets a list of students from the *Students* entity set by reading the `Students` property of the database context instance:

```
public ViewResult Index()
{
    return View(db.Students.ToList());
}
```

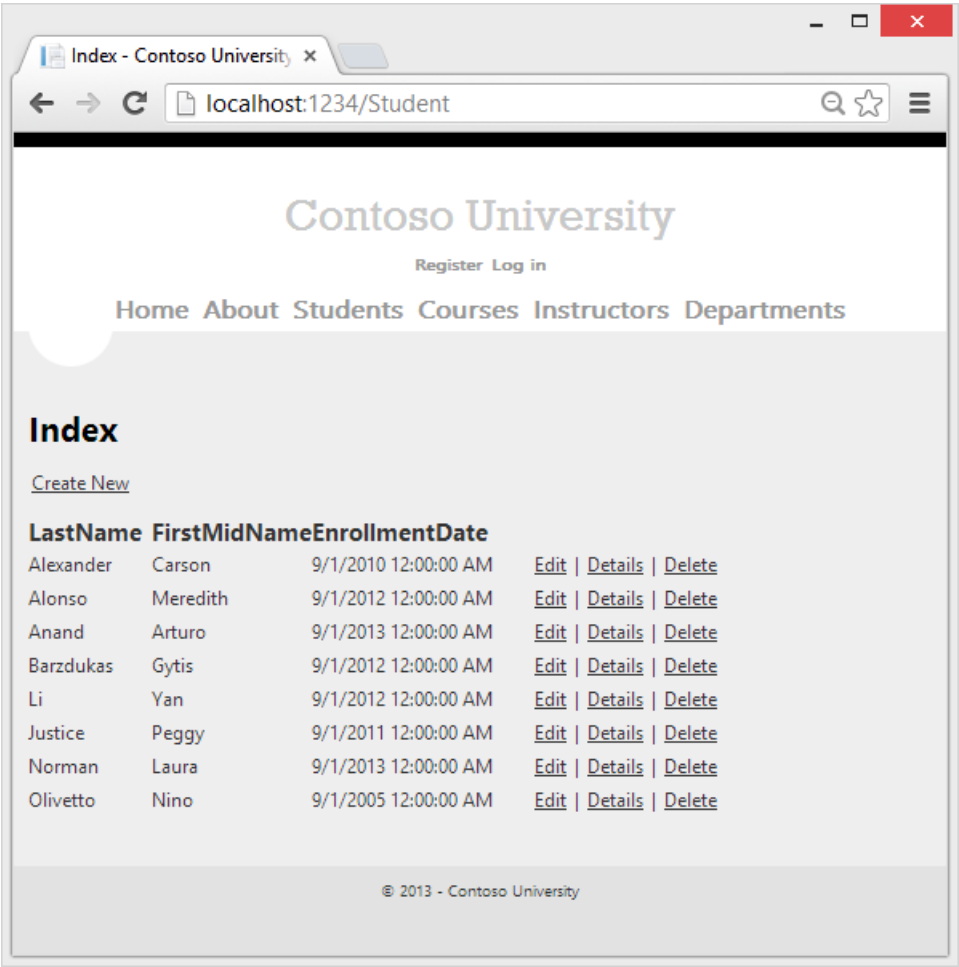
The *Student\Index.cshtml* view displays this list in a table:

```
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.LastName)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.FirstMidName)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.EnrollmentDate)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.LastName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.FirstMidName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.EnrollmentDate)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.StudentID }) |
            @Html.ActionLink("Details", "Details", new { id=item.StudentID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.StudentID })
        </td>
    </tr>
}
```

3. Press CTRL+F5 to run the project.

   Click the **Students** tab to see the test data that the `Seed` method inserted.



Conventions

The amount of code you had to write in order for the Entity Framework to be able to create a complete database for you is minimal because of the use of *conventions*, or assumptions that the Entity Framework makes. Some of them have already been noted:

- The pluralized forms of entity class names are used as table names.
- Entity property names are used for column names.
- Entity properties that are named `ID` or *classname* `ID` are recognized as primary key properties.

You've seen that conventions can be overridden (for example, you specified that table names shouldn't be pluralized), and you'll learn more about conventions and how to override them in the Creating a More Complex Data Model tutorial later in this series. For more information, see Code First Conventions.

Summary

You've now created a simple application that uses the Entity Framework and SQL Server Express to store and display data. In the following tutorial you'll learn how to perform basic CRUD (create, read, update, delete) operations. You can leave feedback at the bottom of this page. Please let us know how you liked this portion of the tutorial and how we could improve it.

Links to other Entity Framework resources can be found in the ASP.NET Data Access Content Map.

NEXT

Implementing Basic CRUD Functionality with the Entity Framework in ASP.NET MVC Application (2 of 10)

2/20/2020 • 14 minutes to read • Edit Online

by Tom Dykstra

Download Completed Project

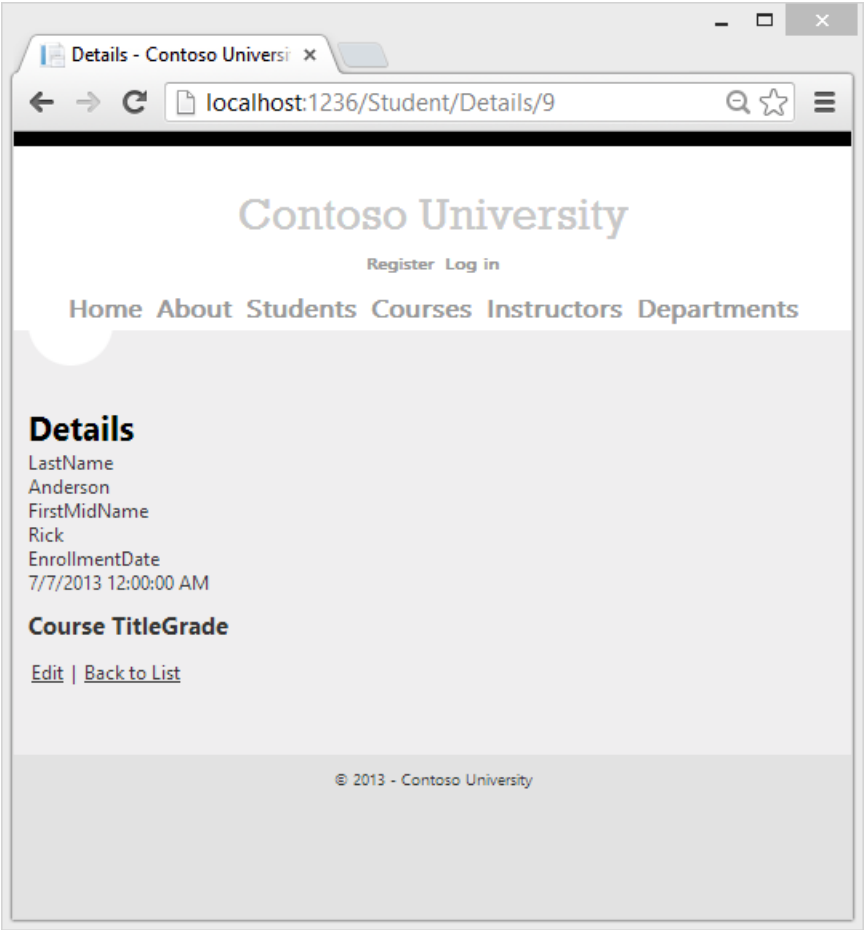
The Contoso University sample web application demonstrates how to create ASP.NET MVC 4 applications using the Entity Framework 5 Code First and Visual Studio 2012. For information about the tutorial series, see the first tutorial in the series. You can start the tutorial series from the beginning or download a starter project for this chapter and start here.

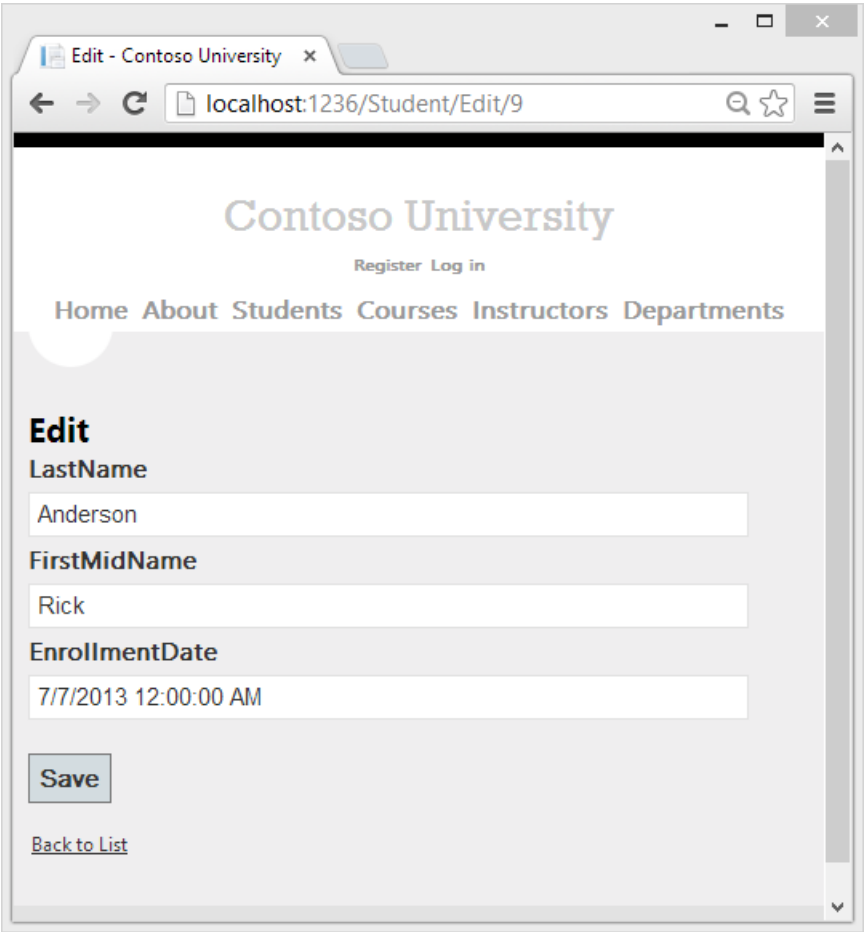
> **NOTE**
> If you run into a problem you can't resolve, download the completed chapter and try to reproduce your problem. You can generally find the solution to the problem by comparing your code to the completed code. For some common errors and how to solve them, see Errors and Workarounds.

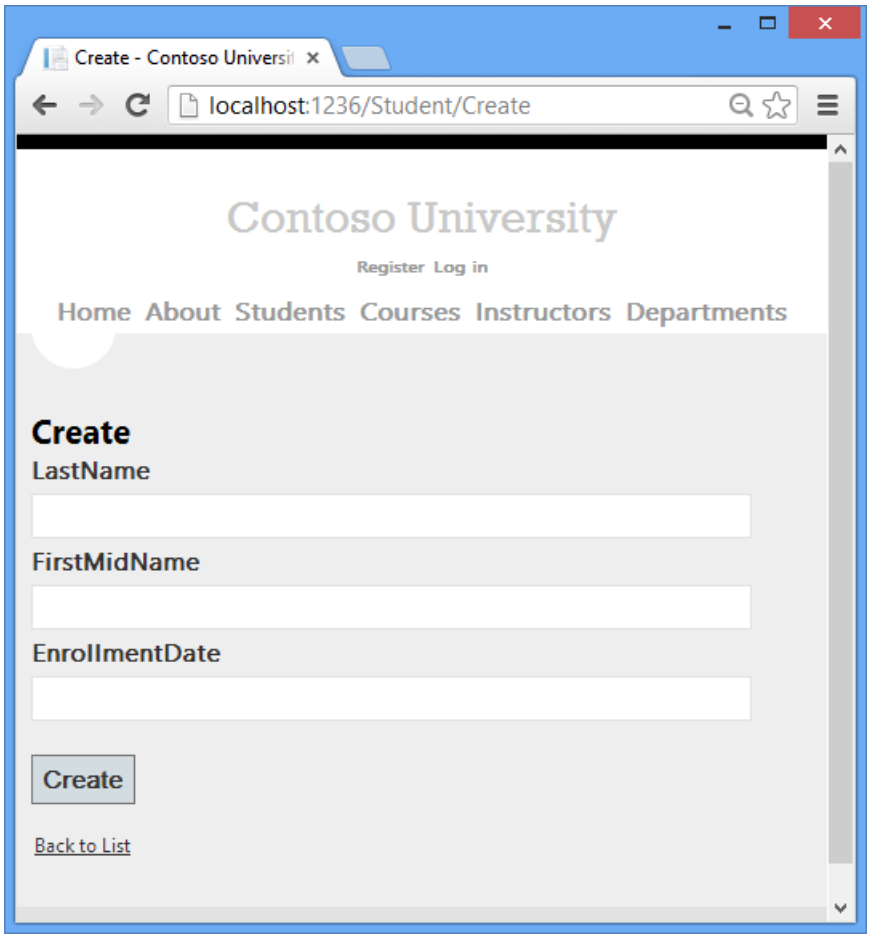
In the previous tutorial you created an MVC application that stores and displays data using the Entity Framework and SQL Server LocalDB. In this tutorial you'll review and customize the CRUD (create, read, update, delete) code that the MVC scaffolding automatically creates for you in controllers and views.

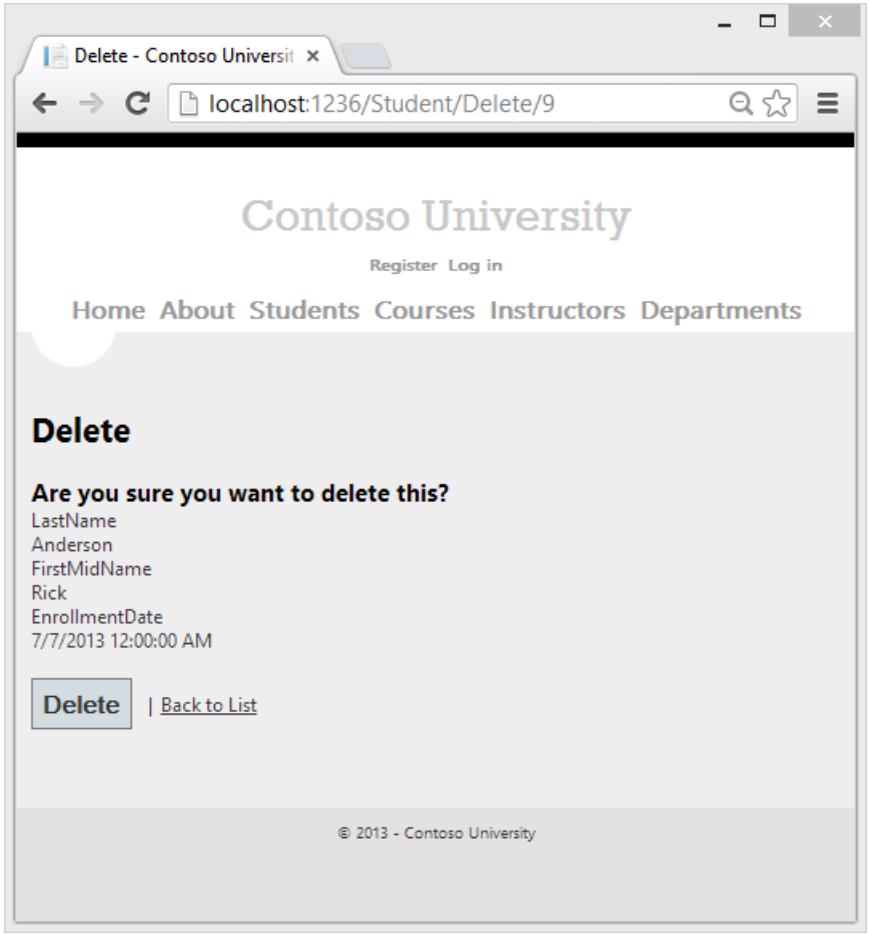
> **NOTE**
> It's a common practice to implement the repository pattern in order to create an abstraction layer between your controller and the data access layer. To keep these tutorials simple, you won't implement a repository until a later tutorial in this series.

In this tutorial, you'll create the following web pages:


Details - Contoso Universi
localhost:1236/Student/Details/9
Contoso University
Register Log in
Home About Students Courses Instructors Departments
Details
LastName
Anderson
FirstMidName
Rick
EnrollmentDate
7/7/2013 12:00:00 AM
Course TitleGrade
Edit | Back to List
© 2013 - Contoso University


Edit - Contoso University
localhost:1236/Student/Edit/9
Contoso University
Register Log in
Home About Students Courses Instructors Departments
Edit
LastName
Anderson
FirstMidName
Rick
EnrollmentDate
7/7/2013 12:00:00 AM
Save
Back to List





Creating a Details Page

The scaffolded code for the Students `Index` page left out the `Enrollments` property, because that property holds a collection. In the `Details` page you'll display the contents of the collection in an HTML table.

In *Controllers\StudentController.cs*, the action method for the `Details` view uses the `Find` method to retrieve a single `Student` entity.

```
public ActionResult Details(int id = 0)
{
    Student student = db.Students.Find(id);
    if (student == null)
    {
        return HttpNotFound();
    }
    return View(student);
}
```

The key value is passed to the method as the `id` parameter and comes from route data in the **Details** hyperlink on the Index page.

1. Open *Views\Student\Details.cshtml*. Each field is displayed using a `DisplayFor` helper, as shown in the following example:

```
<div class="display-label">
         @Html.DisplayNameFor(model => model.LastName)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.LastName)
    </div>
```

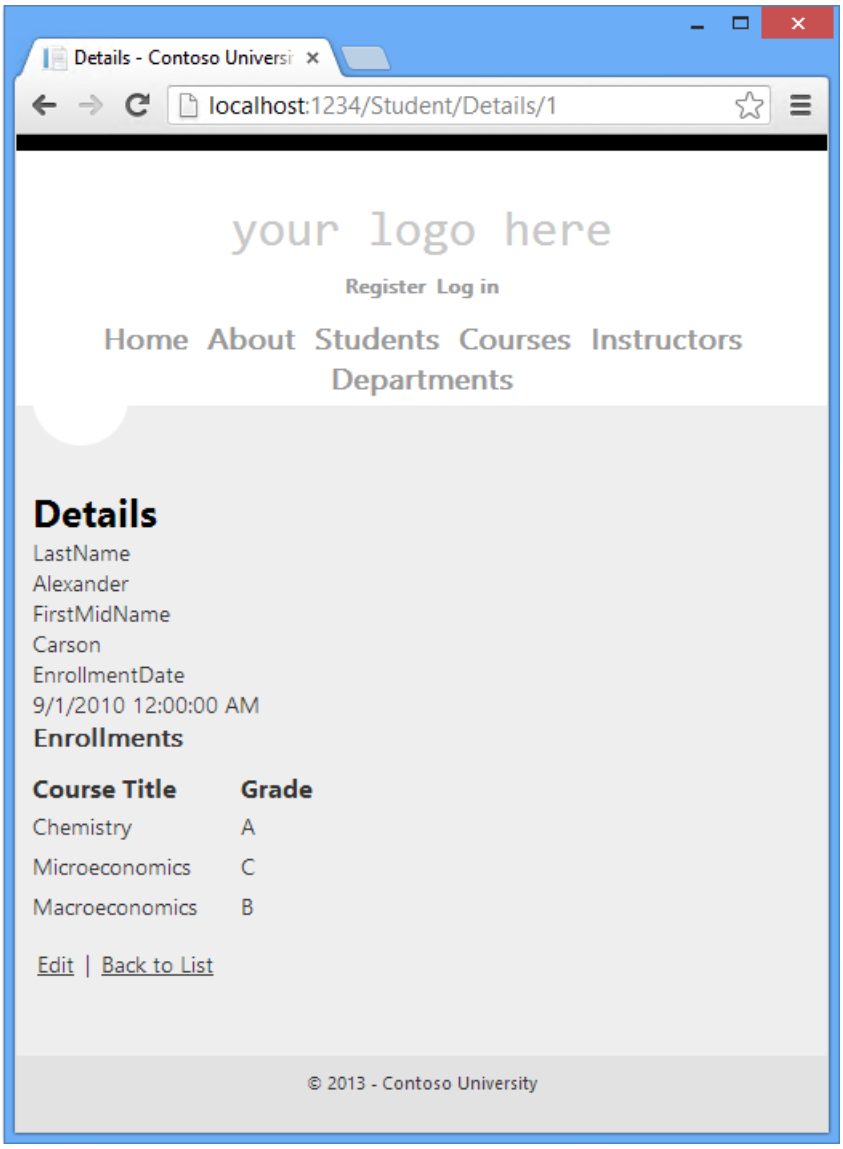
2. After the `EnrollmentDate` field and immediately before the closing `fieldset` tag, add code to display a list of enrollments, as shown in the following example:

```
<div class="display-label">
        @Html.LabelFor(model => model.Enrollments)
    </div>
    <div class="display-field">
        <table>
            <tr>
                <th>Course Title</th>
                <th>Grade</th>
            </tr>
            @foreach (var item in Model.Enrollments)
            {
                <tr>
                    <td>
                        @Html.DisplayFor(modelItem => item.Course.Title)
                    </td>
                    <td>
                        @Html.DisplayFor(modelItem => item.Grade)
                    </td>
                </tr>
            }
        </table>
    </div>
</fieldset>
<p>
    @Html.ActionLink("Edit", "Edit", new { id=Model.StudentID }) |
    @Html.ActionLink("Back to List", "Index")
</p>
```

This code loops through the entities in the `Enrollments` navigation property. For each `Enrollment` entity in the property, it displays the course title and the grade. The course title is retrieved from the `Course` entity that's stored in the `Course` navigation property of the `Enrollments` entity. All of this data is retrieved from

the database automatically when it's needed. (In other words, you are using lazy loading here. You did not specify *eager loading* for the `Courses` navigation property, so the first time you try to access that property, a query is sent to the database to retrieve the data. You can read more about lazy loading and eager loading in the Reading Related Data tutorial later in this series.)

3. Run the page by selecting the **Students** tab and clicking a **Details** link for Alexander Carson. You see the list of courses and grades for the selected student:



Updating the Create Page

1. In *Controllers\StudentController.cs*, replace the `HttpPost``Create` action method with the following code to add a `try-catch` block and the Bind attribute to the scaffolded method:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(
   [Bind(Include = "LastName, FirstMidName, EnrollmentDate")]
   Student student)
{
   try
   {
      if (ModelState.IsValid)
      {
         db.Students.Add(student);
         db.SaveChanges();
         return RedirectToAction("Index");
      }
   }
   catch (DataException /* dex */)
   {
      //Log the error (uncomment dex variable name after DataException and add a line here to write a
log.
      ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists see
your system administrator.");
   }
   return View(student);
}
```

This code adds the `Student` entity created by the ASP.NET MVC model binder to the `Students` entity set and then saves the changes to the database. (*Model binder* refers to the ASP.NET MVC functionality that makes it easier for you to work with data submitted by a form; a model binder converts posted form values to CLR types and passes them to the action method in parameters. In this case, the model binder instantiates a `Student` entity for you using property values from the `Form` collection.)

The `ValidateAntiForgeryToken` attribute helps prevent cross-site request forgery attacks.

```
> [!WARNING]
> Security - The `Bind` attribute is added to protect against *over-posting*. For example, suppose the
`Student` entity includes a `Secret` property that you don't want this web page to update.
>
> [!code-csharp[Main](implementing-basic-crud-functionality-with-the-entity-framework-in-asp-net-mvc-
application/samples/sample5.cs?highlight=7)]
>
> Even if you don't have a `Secret` field on the web page, a hacker could use a tool such as [fiddler]
(http://fiddler2.com/home), or write some JavaScript, to post a `Secret` form value. Without the [Bind]
(https://msdn.microsoft.com/library/system.web.mvc.bindattribute(v=vs.108).aspx) attribute limiting the fields
that the model binder uses when it creates a `Student` instance*,* the model binder would pick up that
`Secret` form value and use it to update the `Student` entity instance. Then whatever value the hacker
specified for the `Secret` form field would be updated in your database. The following image shows the fiddler
tool adding the `Secret` field (with the value "OverPost") to the posted form values.
>
> ![](implementing-basic-crud-functionality-with-the-entity-framework-in-asp-net-mvc-
application/_static/image6.png)
>
> The value "OverPost" would then be successfully added to the `Secret` property of the inserted row, although
you never intended that the web page be able to update that property.
>
> It's a security best practice to use the `Include` parameter with the `Bind` attribute to *whitelist*
fields. It's also possible to use the `Exclude` parameter to *blacklist* fields you want to exclude. The
reason `Include` is more secure is that when you add a new property to the entity, the new field is not
automatically protected by an `Exclude` list.
>
> Another alternative approach, and one preferred by many, is to use only view models with model binding. The
view model contains only the properties you want to bind. Once the MVC model binder has finished, you copy the
view model properties to the entity instance.

Other than the `Bind` attribute, the `try-catch` block is the only change you've made to the scaffolded code.
If an exception that derives from [DataException]
(https://msdn.microsoft.com/library/system.data.dataexception.aspx) is caught while the changes are being
saved, a generic error message is displayed. [DataException]
(https://msdn.microsoft.com/library/system.data.dataexception.aspx) exceptions are sometimes caused by
something external to the application rather than a programming error, so the user is advised to try again.
Although not implemented in this sample, a production quality application would log the exception (and non-
null inner exceptions ) with a logging mechanism such as [ELMAH](https://code.google.com/p/elmah/).

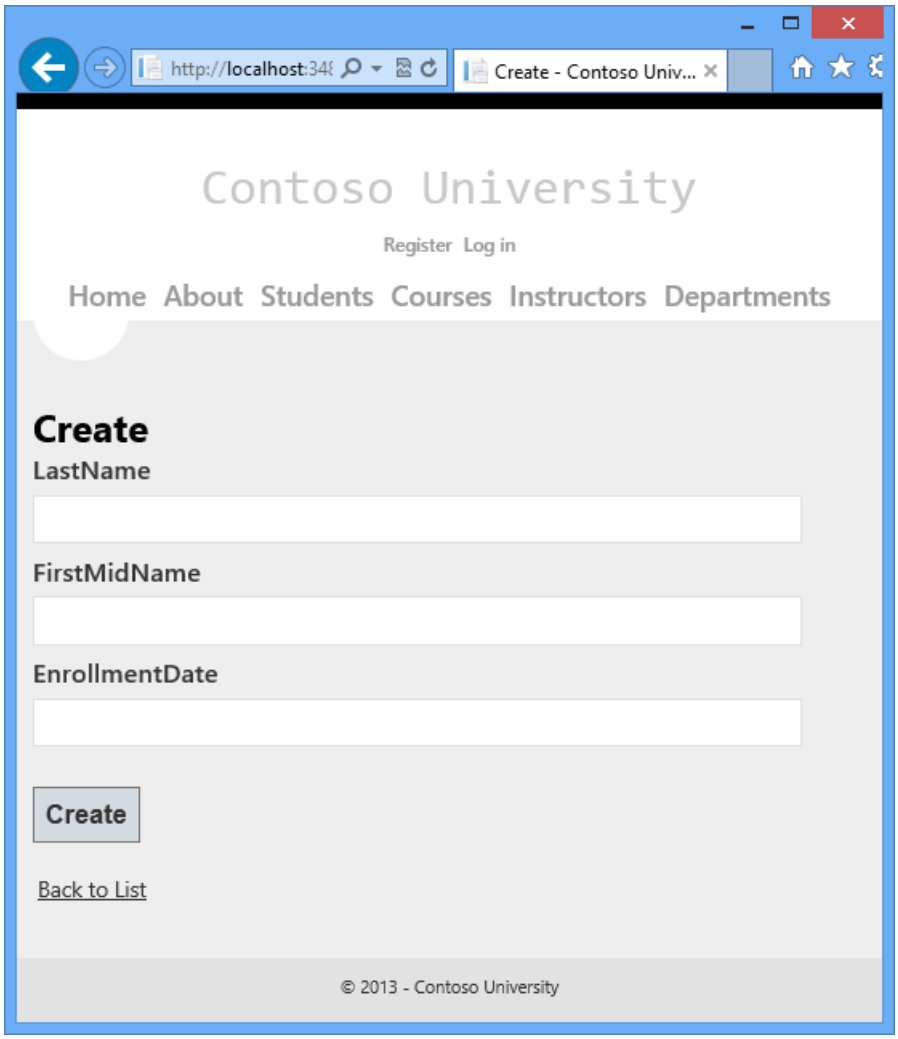
The code in *Views\Student\Create.cshtml* is similar to what you saw in *Details.cshtml*, except that
`EditorFor` and `ValidationMessageFor` helpers are used for each field instead of `DisplayFor`. The following
example shows the relevant code:

[!code-cshtml[Main](implementing-basic-crud-functionality-with-the-entity-framework-in-asp-net-mvc-
application/samples/sample6.cshtml)]

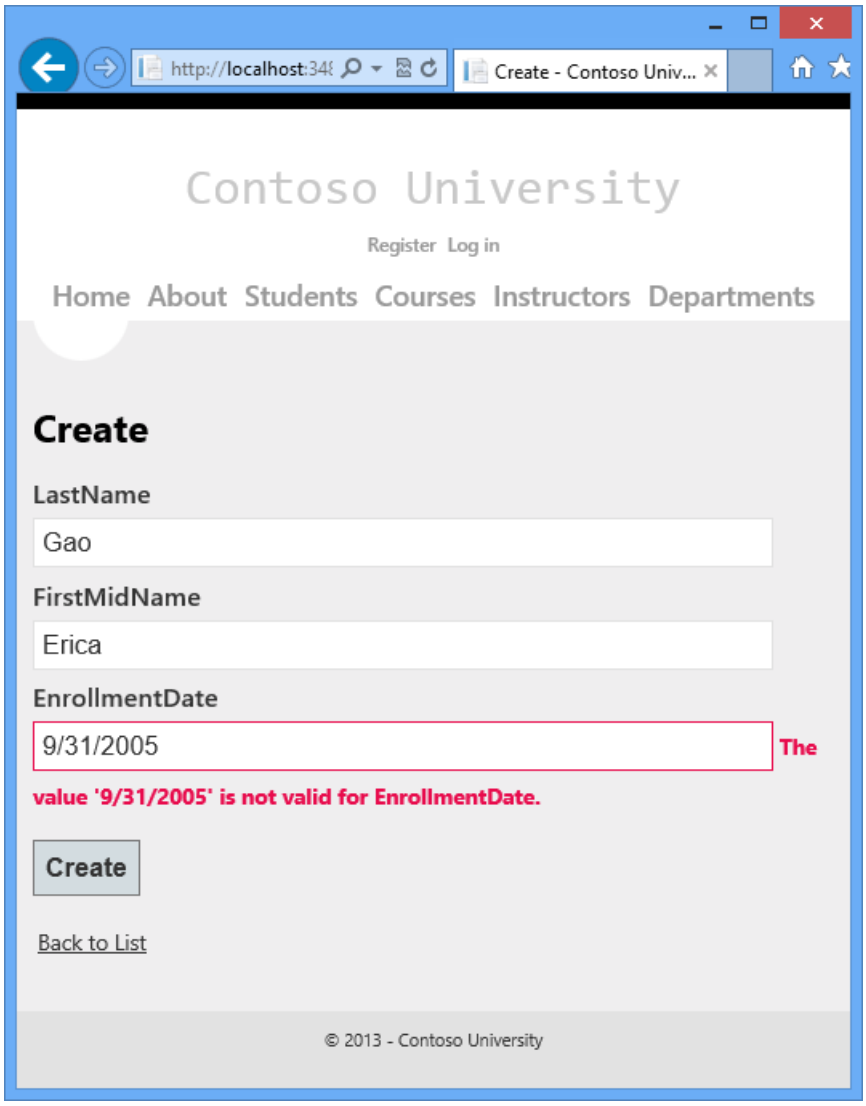
*Create.cshtml* also includes `@Html.AntiForgeryToken()`, which works with the `ValidateAntiForgeryToken`
attribute in the controller to help prevent [cross-site request forgery](../../security/xsrfcsrf-prevention-
in-aspnet-mvc-and-web-pages.md) attacks.

No changes are required in *Create.cshtml*.
```

2. Run the page by selecting the **Students** tab and clicking **Create New**.



Some data validation works by default. Enter names and an invalid date and click **Create** to see the error message.

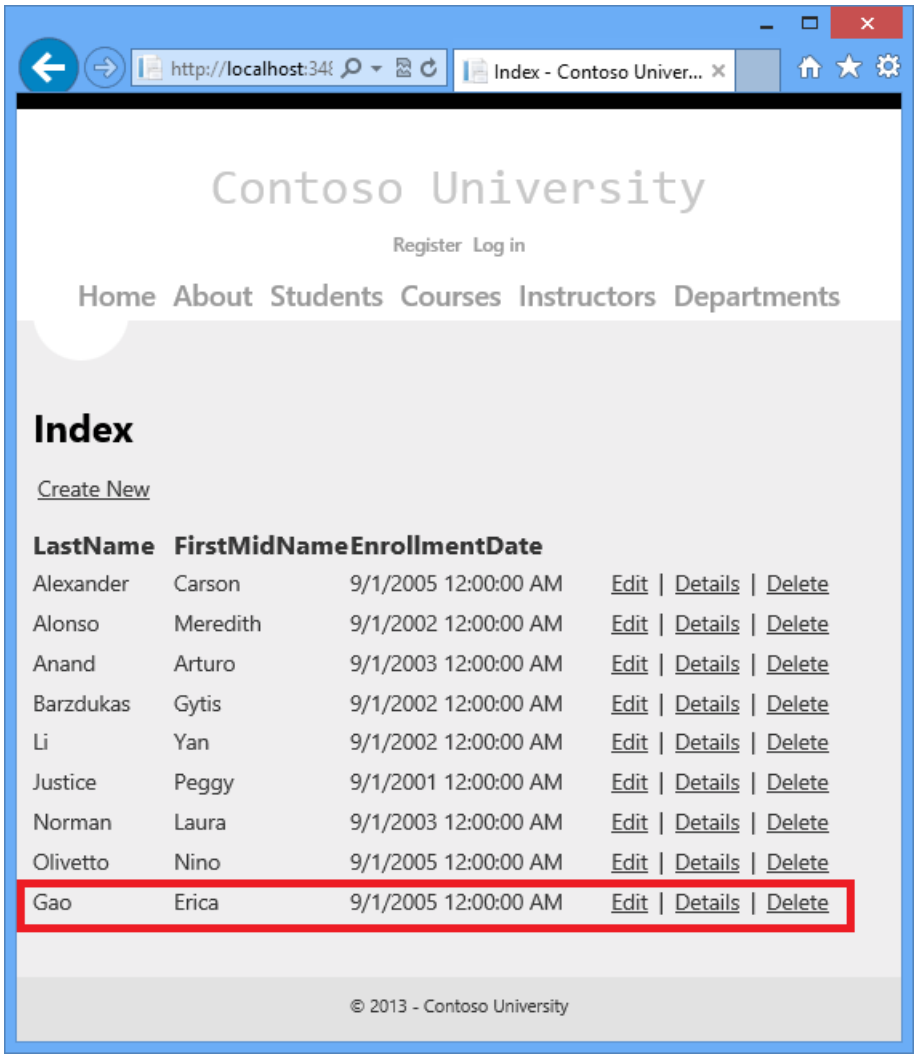


The following highlighted code shows the model validation check.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(Student student)
{
    if (ModelState.IsValid)
    {
        db.Students.Add(student);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(student);
}
```

Change the date to a valid value such as 9/1/2005 and click **Create** to see the new student appear in the **Index** page.



Updating the Edit POST Page

In *Controllers\StudentController.cs*, the `HttpGet` `Edit` method (the one without the `HttpPost` attribute) uses the `Find` method to retrieve the selected `Student` entity, as you saw in the `Details` method. You don't need to change this method.

However, replace the `HttpPost` `Edit` action method with the following code to add a `try-catch` block and the Bind attribute:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(
   [Bind(Include = "StudentID, LastName, FirstMidName, EnrollmentDate")]
   Student student)
{
   try
   {
      if (ModelState.IsValid)
      {
         db.Entry(student).State = EntityState.Modified;
         db.SaveChanges();
         return RedirectToAction("Index");
      }
   }
   catch (DataException /* dex */)
   {
      //Log the error (uncomment dex variable name after DataException and add a line here to write a log.
      ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists see your
system administrator.");
   }
   return View(student);
}
```

This code is similar to what you saw in the `HttpPost` `Create` method. However, instead of adding the entity created by the model binder to the entity set, this code sets a flag on the entity indicating it has been changed. When the SaveChanges method is called, the Modified flag causes the Entity Framework to create SQL statements to update the database row. All columns of the database row will be updated, including those that the user didn't change, and concurrency conflicts are ignored. (You'll learn how to handle concurrency in a later tutorial in this series.)

Entity States and the Attach and SaveChanges Methods

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database, and this information determines what happens when you call the `SaveChanges` method. For example, when you pass a new entity to the Add method, that entity's state is set to `Added`. Then when you call the SaveChanges method, the database context issues a SQL `INSERT` command.

An entity may be in one of thefollowing states:

- `Added`. The entity does not yet exist in the database. The `SaveChanges` method must issue an `INSERT` statement.
- `Unchanged`. Nothing needs to be done with this entity by the `SaveChanges` method. When you read an entity from the database, the entity starts out with this status.
- `Modified`. Some or all of the entity's property values have been modified. The `SaveChanges` method must issue an `UPDATE` statement.
- `Deleted`. The entity has been marked for deletion. The `SaveChanges` method must issue a `DELETE` statement.
- `Detached`. The entity isn't being tracked by the database context.

In a desktop application, state changes are typically set automatically. In a desktop type of application, you read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to `Modified`. Then when you call `SaveChanges`, the Entity Framework generates a SQL `UPDATE` statement that updates only the actual properties that you changed.

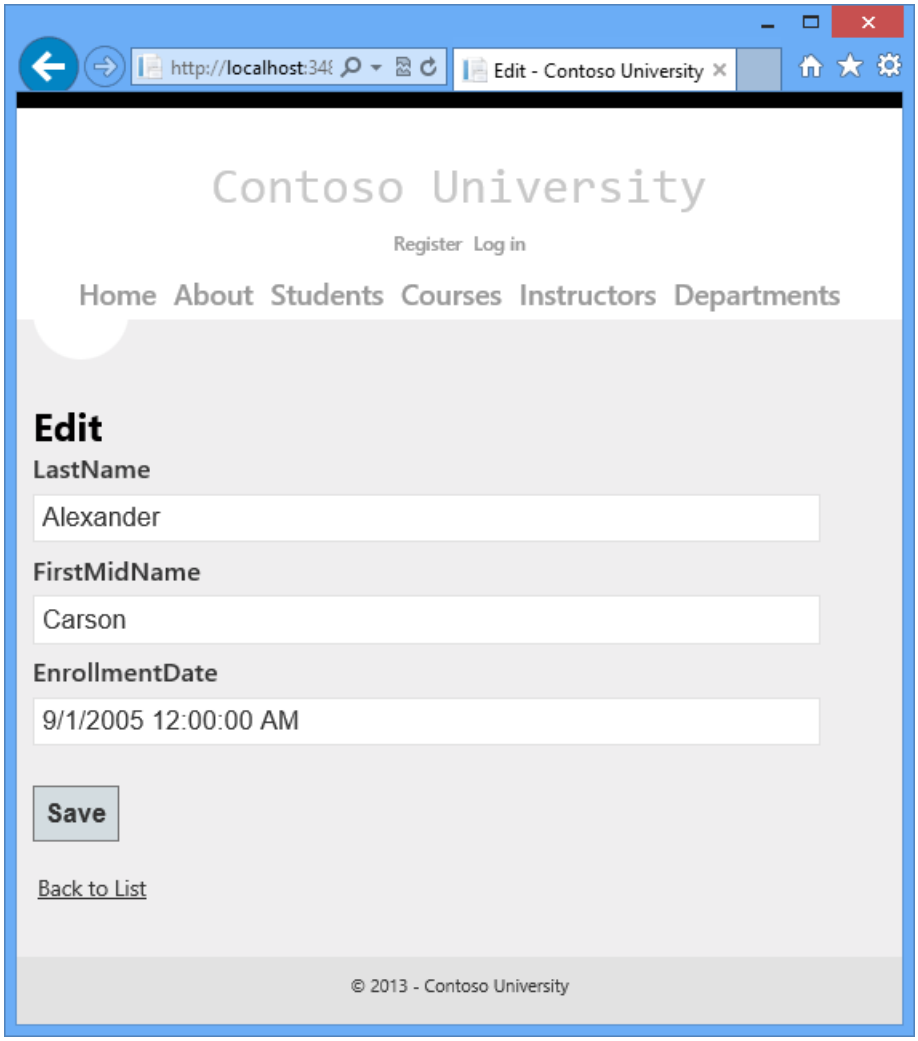
The disconnected nature of web apps doesn't allow for this continuous sequence. The DbContext that reads an entity is disposed after a page is rendered. When the `HttpPost` `Edit` action method is called, a new request is made and you have a new instance of the DbContext, so you have to manually set the entity state to `Modified.` Then when you call `SaveChanges`, the Entity Framework updates all columns of the database row, because the

context has no way to know which properties you changed.

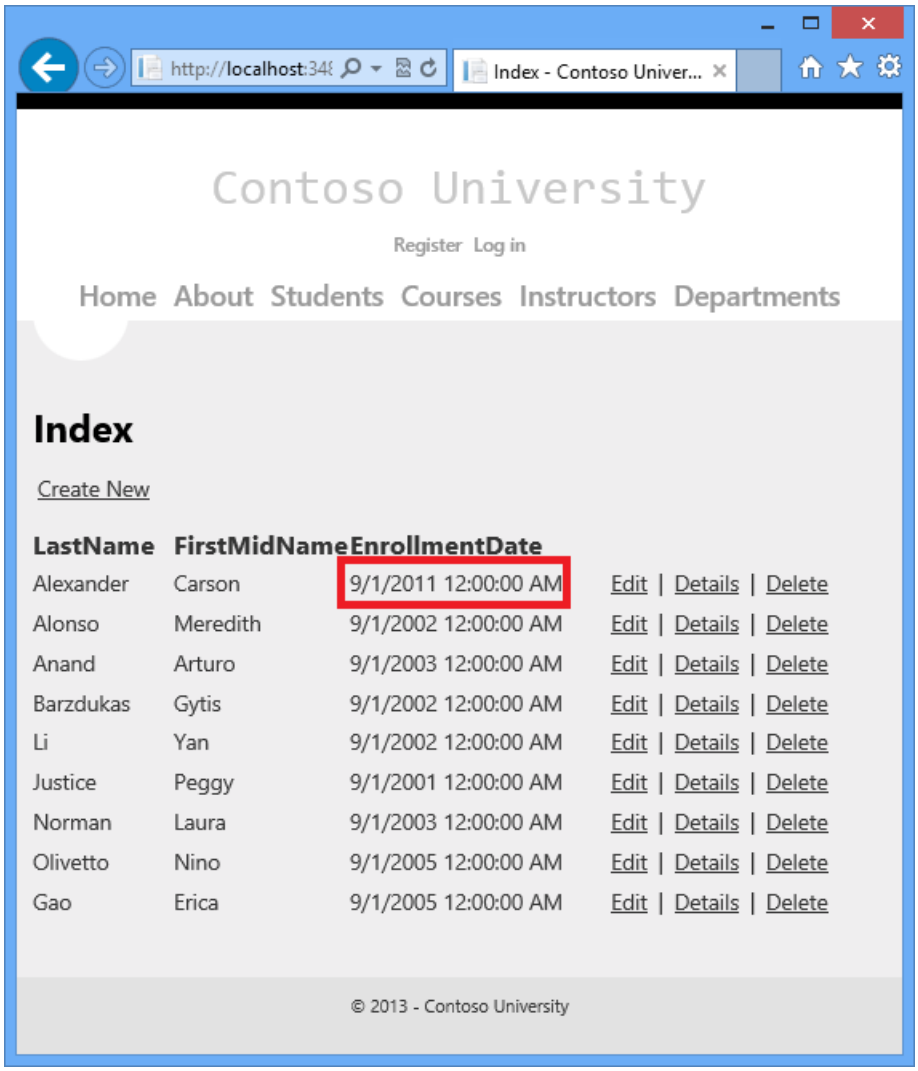
If you want the SQL `Update` statement to update only the fields that the user actually changed, you can save the original values in some way (such as hidden fields) so that they are available when the `HttpPost` `Edit` method is called. Then you can create a `Student` entity using the original values, call the `Attach` method with that original version of the entity, update the entity's values to the new values, and then call `SaveChanges.` For more information, see Entity states and SaveChanges and Local Data in the MSDN Data Developer Center.

The code in *Views\Student\Edit.cshtml* is similar to what you saw in *Create.cshtml*, and no changes are required.

Run the page by selecting the **Students** tab and then clicking an **Edit** hyperlink.



Change some of the data and click **Save**. You see the changed data in the Index page.



Updating the Delete Page

In *Controllers\StudentController.cs*, the template code for the `HttpGet` `Delete` method uses the `Find` method to retrieve the selected `Student` entity, as you saw in the `Details` and `Edit` methods. However, to implement a custom error message when the call to `SaveChanges` fails, you'll add some functionality to this method and its corresponding view.

As you saw for update and create operations, delete operations require two action methods. The method that is called in response to a GET request displays a view that gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST request is created. When that happens, the `HttpPost` `Delete` method is called and then that method actually performs the delete operation.

You'll add a `try-catch` block to the `HttpPost` `Delete` method to handle any errors that might occur when the database is updated. If an error occurs, the `HttpPost` `Delete` method calls the `HttpGet` `Delete` method, passing it a parameter that indicates that an error has occurred. The `HttpGet Delete` method then redisplays the confirmation page along with the error message, giving the user an opportunity to cancel or try again.

1. Replace the `HttpGet` `Delete` action method with the following code, which manages error reporting:

```
public ActionResult Delete(bool? saveChangesError=false, int id = 0)
{
    if (saveChangesError.GetValueOrDefault())
    {
        ViewBag.ErrorMessage = "Delete failed. Try again, and if the problem persists see your system
administrator.";
    }
    Student student = db.Students.Find(id);
    if (student == null)
    {
        return HttpNotFound();
    }
    return View(student);
}
```

This code accepts an optional Boolean parameter that indicates whether it was called after a failure to save changes. This parameter is `false` when the `HttpGet` `Delete` method is called without a previous failure. When it is called by the `HttpPost` `Delete` method in response to a database update error, the parameter is `true` and an error message is passed to the view.

2. Replace the `HttpPost` `Delete` action method (named `DeleteConfirmed`) with the following code, which performs the actual delete operation and catches any database update errors.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete(int id)
{
    try
    {
        Student student = db.Students.Find(id);
        db.Students.Remove(student);
        db.SaveChanges();
    }
    catch (DataException/* dex */)
    {
        // uncomment dex and log error.
        return RedirectToAction("Delete", new { id = id, saveChangesError = true });
    }
    return RedirectToAction("Index");
}
```

This code retrieves the selected entity, then calls the Remove method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL `DELETE` command is generated. You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code named the `HttpPost` `Delete` method `DeleteConfirmed` to give the `HttpPost` method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the `HttpPost` and `HttpGet` delete methods.

If improving performance in a high-volume application is a priority, you could avoid an unnecessary SQL query to retrieve the row by replacing the lines of code that call the `Find` and `Remove` methods with the following code as shown in yellow highlight:

```
Student studentToDelete = new Student() { StudentID = id };
db.Entry(studentToDelete).State = EntityState.Deleted;
```

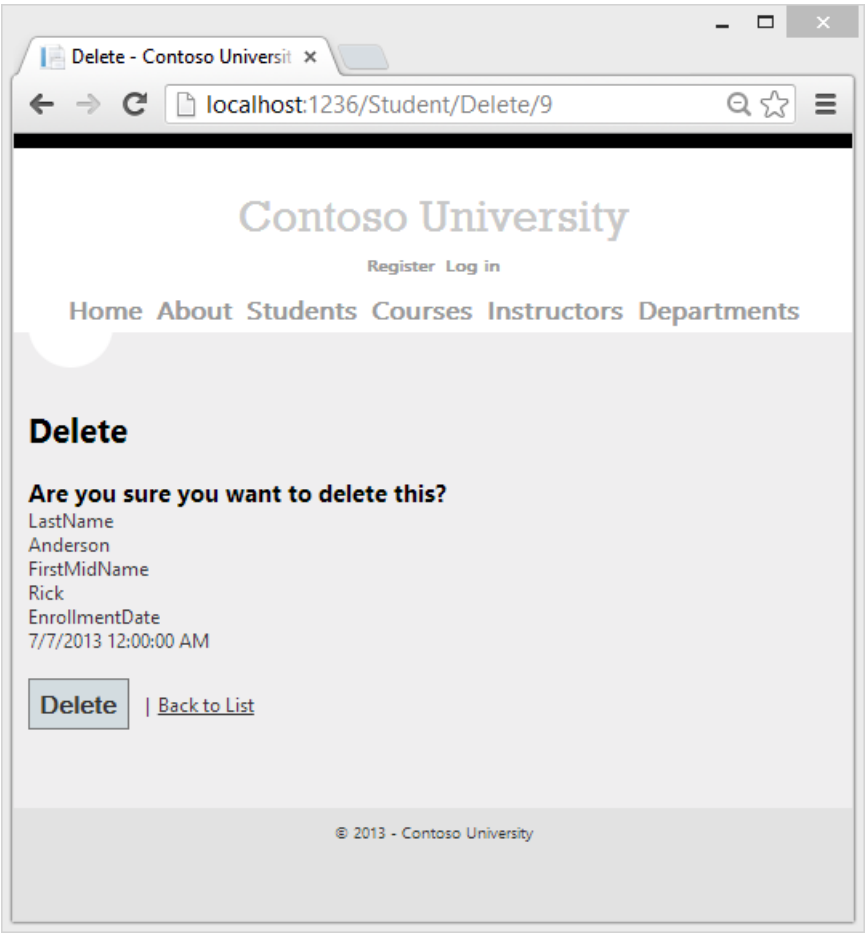
This code instantiates a `Student` entity using only the primary key value and then sets the entity state to `Deleted`. That's all that the Entity Framework needs in order to delete the entity.

As noted, the `HttpGet` `Delete` method doesn't delete the data. Performing a delete operation in response to a GET request (or for that matter, performing any edit operation, create operation, or any other operation that changes data) creates a security risk. For more information, see ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes on Stephen Walther's blog.

3. In *Views\Student\Delete.cshtml*, add an error message between the `h2` heading and the `h3` heading, as shown in the following example:

```
<h2>Delete</h2>
<p class="error">@ViewBag.ErrorMessage</p>
<h3>Are you sure you want to delete this?</h3>
```

Run the page by selecting the **Students** tab and clicking a **Delete** hyperlink:



4. Click **Delete**. The Index page is displayed without the deleted student. (You'll see an example of the error handling code in action in the Handling Concurrency tutorial later in this series.)

Ensuring that Database Connections Are Not Left Open

To make sure that database connections are properly closed and the resources they hold freed up, you should see to it that the context instance is disposed. That is why the scaffolded code provides a Dispose method at the end of the `StudentController` class in *StudentController.cs*, as shown in the following example:

```
protected override void Dispose(bool disposing)
{
    db.Dispose();
    base.Dispose(disposing);
}
```

The base `Controller` class already implements the `IDisposable` interface, so this code simply adds an override to the `Dispose(bool)` method to explicitly dispose the context instance.

Summary

You now have a complete set of pages that perform simple CRUD operations for `Student` entities. You used MVC helpers to generate UI elements for data fields. For more information about MVC helpers, see Rendering a Form Using HTML Helpers (the page is for MVC 3 but is still relevant for MVC 4).

In the next tutorial you'll expand the functionality of the Index page by adding sorting and paging.

Links to other Entity Framework resources can be found in the ASP.NET Data Access Content Map.

PREVIOUS NEXT

Sorting, Filtering, and Paging with the Entity Framework in an ASP.NET MVC Application (3 of 10)

9/16/2020 • 22 minutes to read • Edit Online

by Tom Dykstra

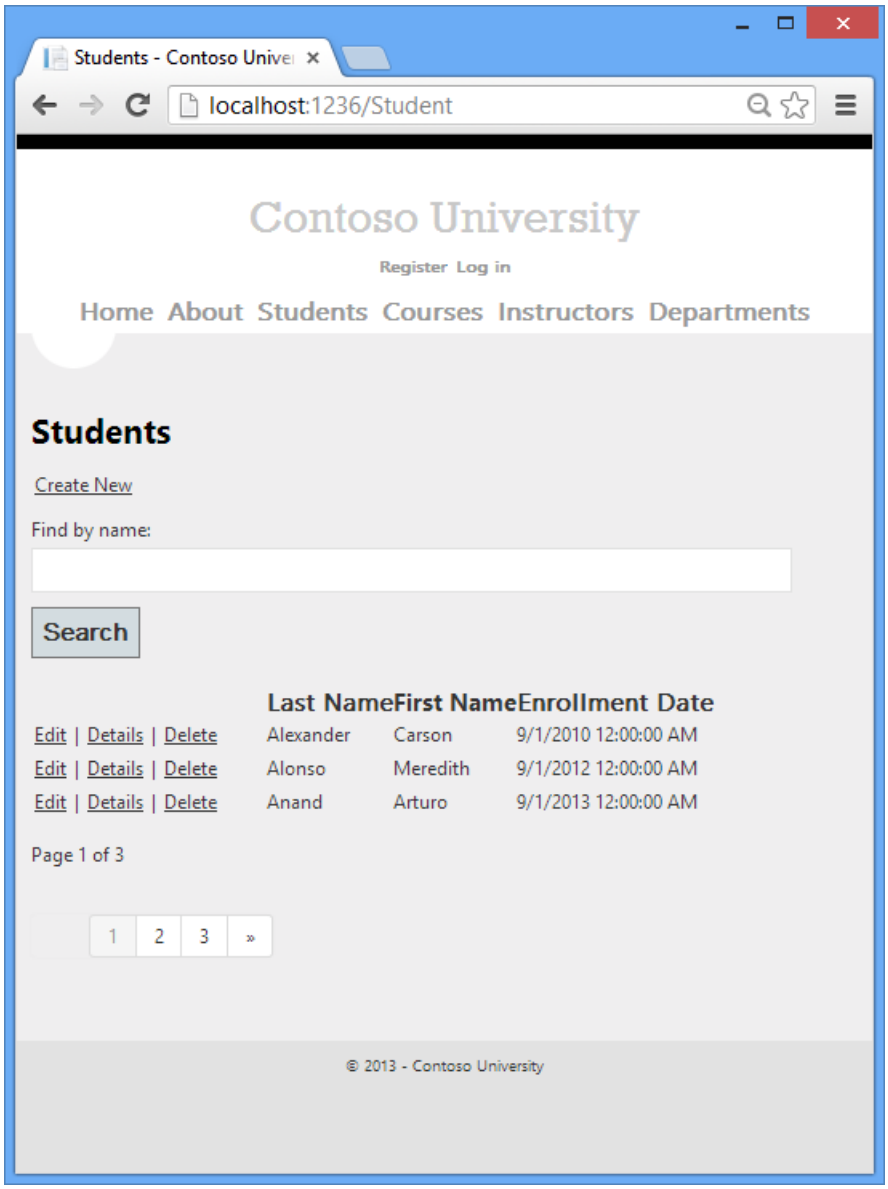
Download Completed Project

The Contoso University sample web application demonstrates how to create ASP.NET MVC 4 applications using the Entity Framework 5 Code First and Visual Studio 2012. For information about the tutorial series, see the first tutorial in the series. You can start the tutorial series from the beginning or download a starter project for this chapter and start here.

> **NOTE**
>
> If you run into a problem you can't resolve, download the completed chapter and try to reproduce your problem. You can generally find the solution to the problem by comparing your code to the completed code. For some common errors and how to solve them, see Errors and Workarounds.

In the previous tutorial you implemented a set of web pages for basic CRUD operations for `Student` entities. In this tutorial you'll add sorting, filtering, and paging functionality to the **Students** Index page. You'll also create a page that does simple grouping.

The following illustration shows what the page will look like when you're done. The column headings are links that the user can click to sort by that column. Clicking a column heading repeatedly toggles between ascending and descending sort order.



Add Column Sort Links to the Students Index Page

To add sorting to the Student Index page, you'll change the `Index` method of the `Student` controller and add code to the `Student` Index view.

Add Sorting Functionality to the Index Method

In *Controllers\StudentController.cs*, replace the `Index` method with the following code:

```
public ActionResult Index(string sortOrder)
{
   ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "Name_desc" : "";
   ViewBag.DateSortParm = sortOrder == "Date" ? "Date_desc" : "Date";
   var students = from s in db.Students
                  select s;
   switch (sortOrder)
   {
      case "Name_desc":
         students = students.OrderByDescending(s => s.LastName);
         break;
      case "Date":
         students = students.OrderBy(s => s.EnrollmentDate);
         break;
      case "Date_desc":
         students = students.OrderByDescending(s => s.EnrollmentDate);
         break;
      default:
         students = students.OrderBy(s => s.LastName);
         break;
   }
   return View(students.ToList());
}
```

This code receives a `sortOrder` parameter from the query string in the URL. The query string value is provided by ASP.NET MVC as a parameter to the action method. The parameter will be a string that's either "Name" or "Date", optionally followed by an underscore and the string "desc" to specify descending order. The default sort order is ascending.

The first time the Index page is requested, there's no query string. The students are displayed in ascending order by `LastName`, which is the default as established by the fall-through case in the `switch` statement. When the user clicks a column heading hyperlink, the appropriate `sortOrder` value is provided in the query string.

The two `ViewBag` variables are used so that the view can configure the column heading hyperlinks with the appropriate query string values:

```
ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "Name_desc" : "";
ViewBag.DateSortParm = sortOrder == "Date" ? "Date_desc" : "Date";
```

These are ternary statements. The first one specifies that if the `sortOrder` parameter is null or empty, `ViewBag.NameSortParm` should be set to "name_desc"; otherwise, it should be set to an empty string. These two statements enable the view to set the column heading hyperlinks as follows:

CURRENT SORT ORDER	LAST NAME HYPERLINK	DATE HYPERLINK
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code creates an IQueryable variable before the `switch` statement, modifies it in the `switch` statement, and calls the `ToList` method after the `switch` statement. When you create and modify `IQueryable` variables, no query is sent to the database. The query is not executed until you convert the `IQueryable` object into a collection by calling a method such as `ToList`. Therefore,

this code results in a single query that is not executed until the `return View` statement.

Add Column Heading Hyperlinks to the Student Index View

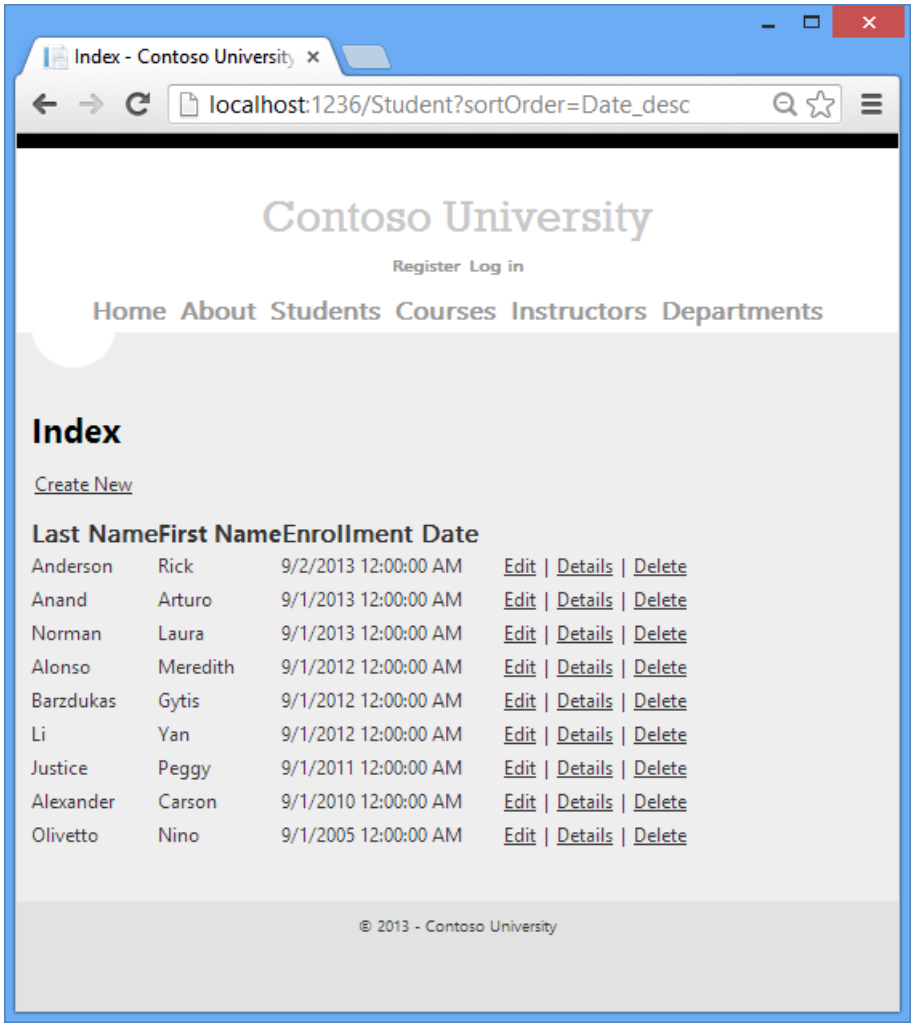
In *Views\Student\Index.cshtml*, replace the `<tr>` and `<th>` elements for the heading row with the highlighted code:

```
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.ActionLink("Last Name", "Index", new { sortOrder = ViewBag.NameSortParm })
        </th>
        <th>First Name
        </th>
        <th>
            @Html.ActionLink("Enrollment Date", "Index", new { sortOrder = ViewBag.DateSortParm })
        </th>
        <th></th>
    </tr>

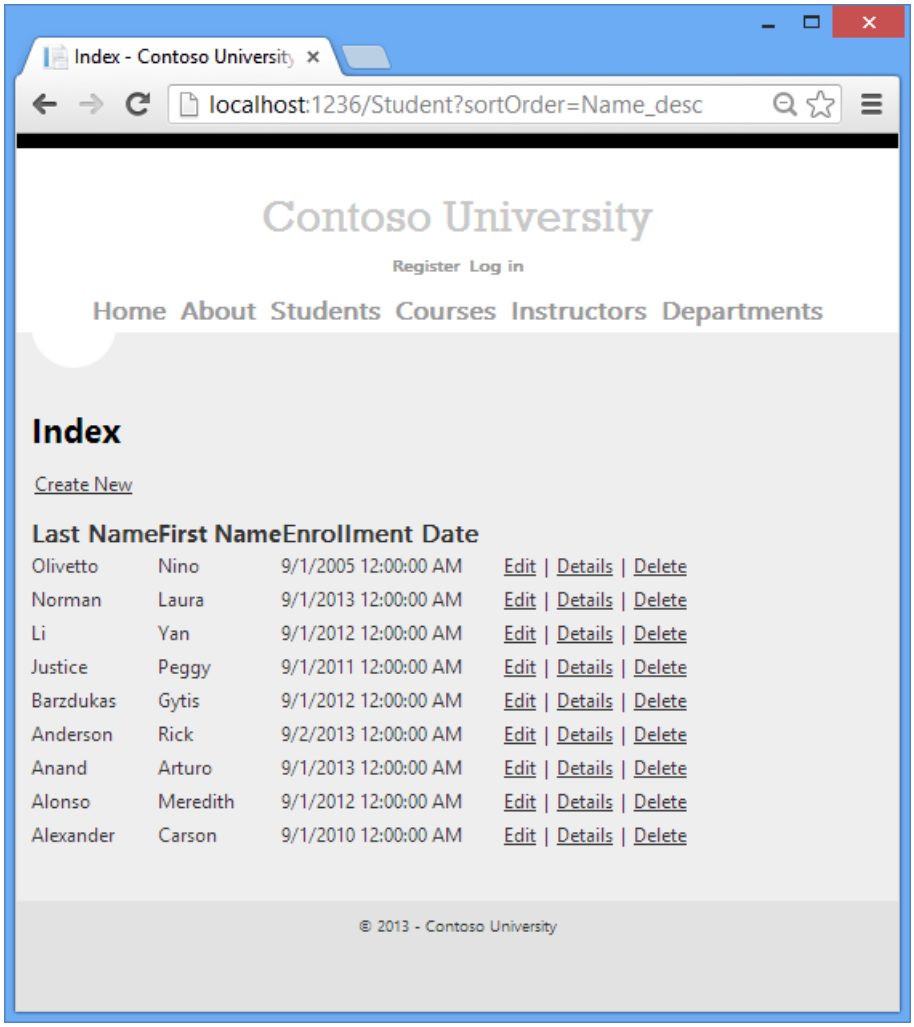
    @foreach (var item in Model)
    {
```

This code uses the information in the `ViewBag` properties to set up hyperlinks with the appropriate query string values.

Run the page and click the **Last Name** and **Enrollment Date** column headings to verify that sorting works.



After you click the **Last Name** heading, students are displayed in descending last name order.



Add a Search Box to the Students Index Page

To add filtering to the Students Index page, you'll add a text box and a submit button to the view and make corresponding changes in the `Index` method. The text box will let you enter a string to search for in the first name and last name fields.

Add Filtering Functionality to the Index Method

In *Controllers\StudentController.cs*, replace the `Index` method with the following code (the changes are highlighted):

```
public ViewResult Index(string sortOrder, string searchString)
{
    ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in db.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())
                                || s.FirstMidName.ToUpper().Contains(searchString.ToUpper()));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }

    return View(students.ToList());
}
```

You've added a `searchString` parameter to the `Index` method. You've also added to the LINQ statement a `where` clause that selects only students whose first name or last name contains the search string. The search string value is received from a text box that you'll add to the Index view.The statement that adds the where clause is executed only if there's a value to search for.

> **NOTE**
>
> In many cases you can call the same method either on an Entity Framework entity set or as an extension method on an in-memory collection. The results are normally the same but in some cases may be different. For example, the .NET Framework implementation of the `Contains` method returns all rows when you pass an empty string to it, but the Entity Framework provider for SQL Server Compact 4.0 returns zero rows for empty strings. Therefore the code in the example (putting the `Where` statement inside an `if` statement) makes sure that you get the same results for all versions of SQL Server. Also, the .NET Framework implementation of the `Contains` method performs a case-sensitive comparison by default, but Entity Framework SQL Server providers perform case-insensitive comparisons by default. Therefore, calling the `ToUpper` method to make the test explicitly case-insensitive ensures that results do not change when you change the code later to use a repository, which will return an `IEnumerable` collection instead of an `IQueryable` object. (When you call the `Contains` method on an `IEnumerable` collection, you get the .NET Framework implementation; when you call it on an `IQueryable` object, you get the database provider implementation.)

Add a Search Box to the Student Index View

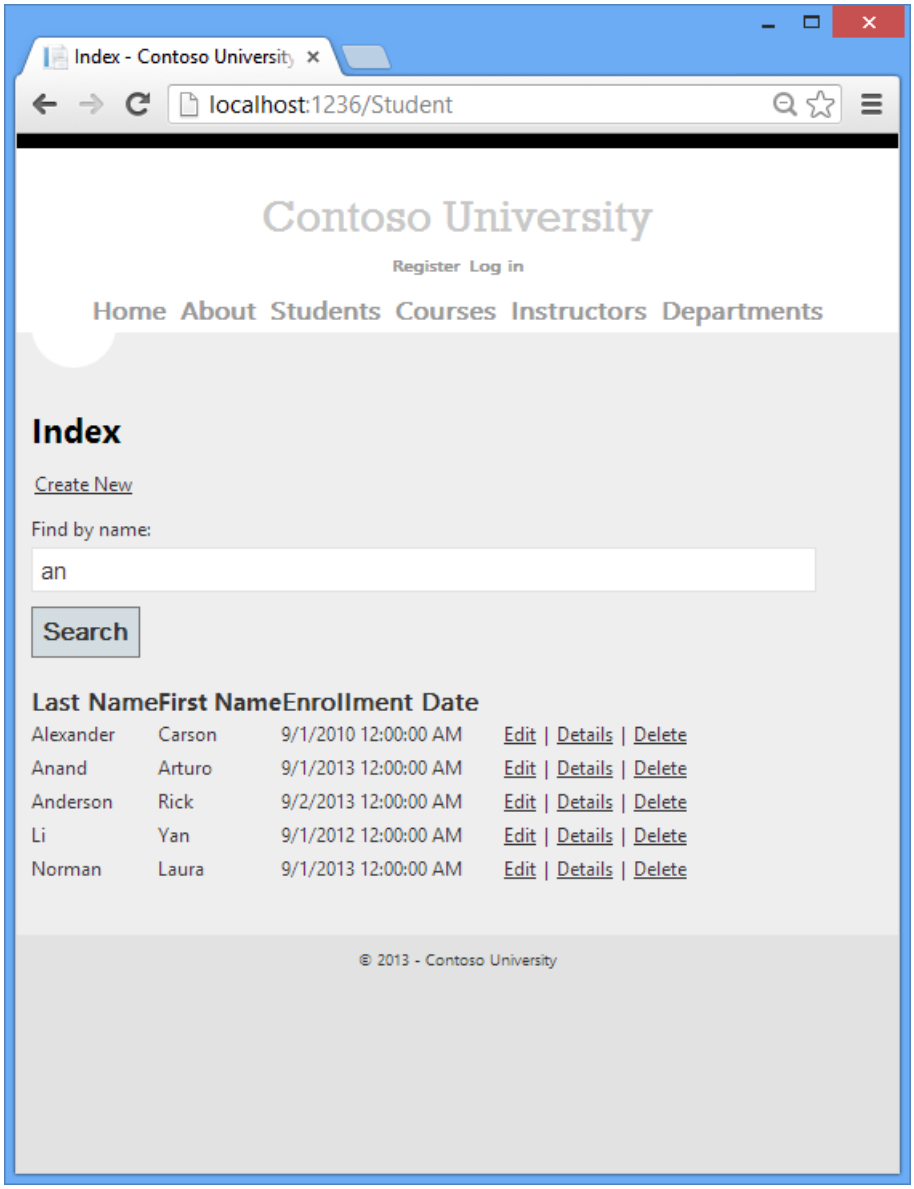
In *Views\Student\Index.cshtml*, add the highlighted code immediately before the opening `table` tag in order to create a caption, a text box, and a **Search** button.

```
<p>
    @Html.ActionLink("Create New", "Create")
</p>

@using (Html.BeginForm())
{
    <p>
        Find by name: @Html.TextBox("SearchString")
        <input type="submit" value="Search" /></p>
}

<table>
    <tr>
```

Run the page, enter a search string, and click **Search** to verify that filtering is working.

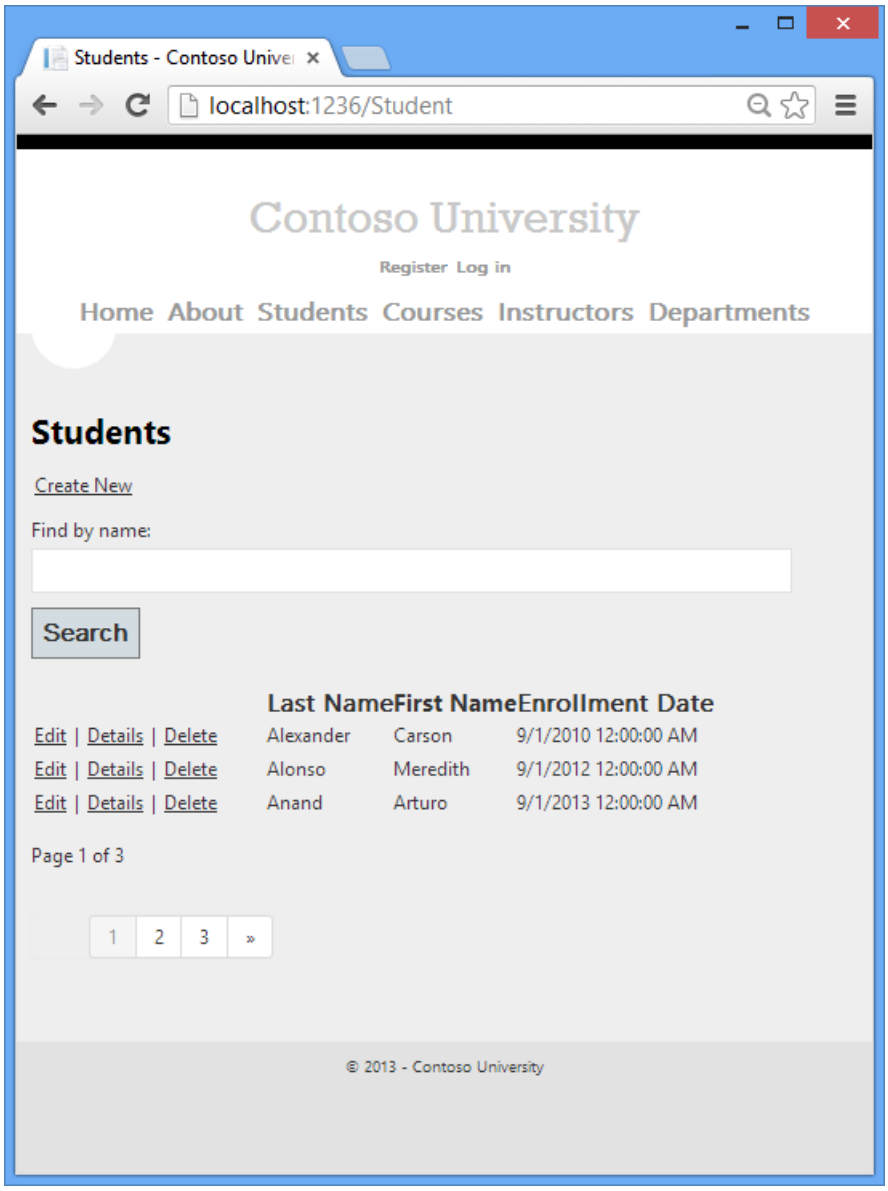


Notice the URL doesn't contain the "an" search string, which means that if you bookmark this page, you won't get the filtered list when you use the bookmark. You'll change the **Search** button to use query strings for filter criteria later in the tutorial.

Add Paging to the Students Index Page

To add paging to the Students Index page, you'll start by installing the **PagedList.Mvc** NuGet package. Then you'll make additional changes in the `Index` method and add paging links to the `Index` view. **PagedList.Mvc** is one of

many good paging and sorting packages for ASP.NET MVC, and its use here is intended only as an example, not as a recommendation for it over other options. The following illustration shows the paging links.

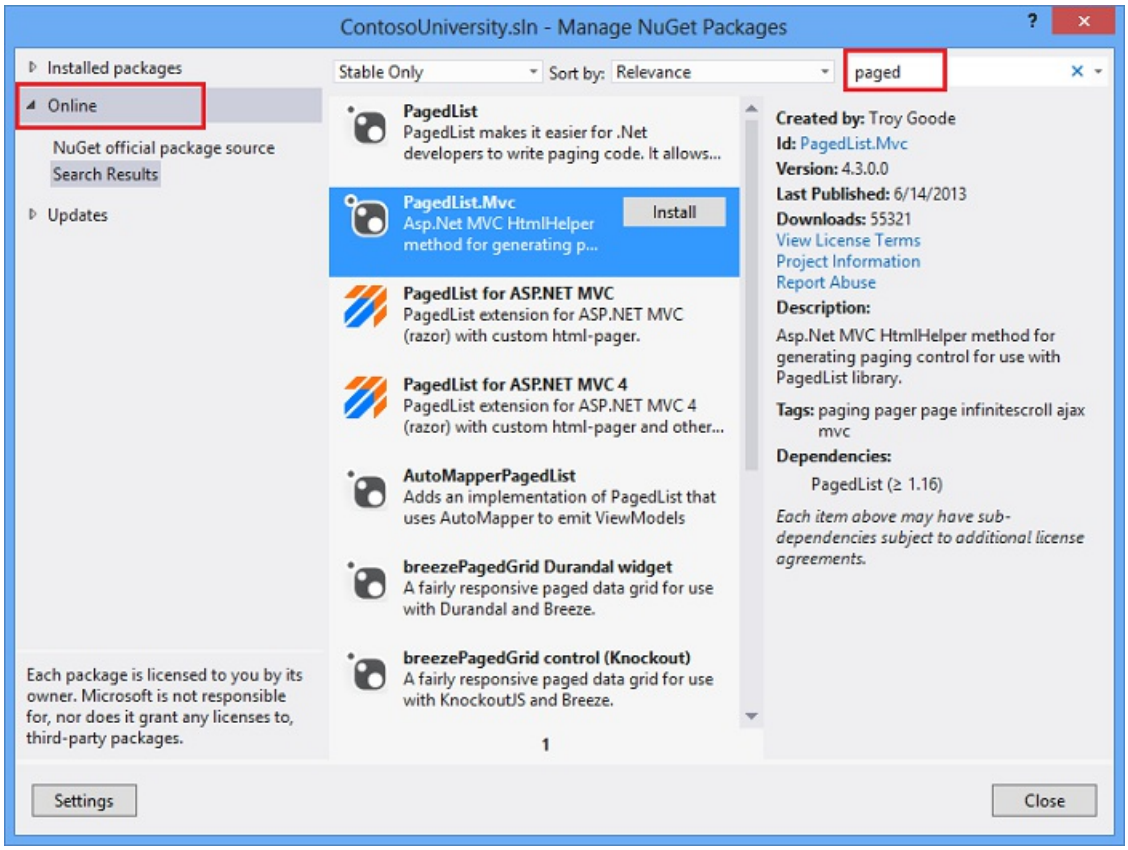


Install the PagedList.MVC NuGet Package

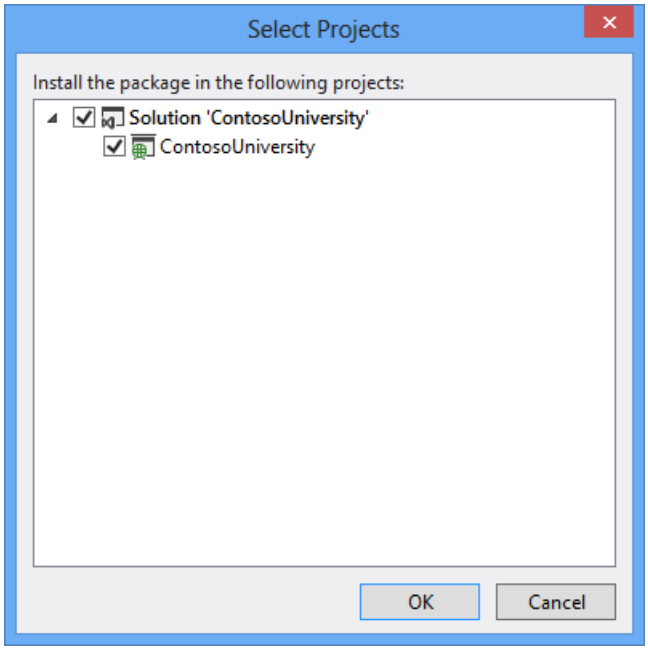
The NuGet **PagedList.Mvc** package automatically installs the **PagedList** package as a dependency. The **PagedList** package installs a `PagedList` collection type and extension methods for `IQueryable` and `IEnumerable` collections. The extension methods create a single page of data in a `PagedList` collection out of your `IQueryable` or `IEnumerable`, and the `PagedList` collection provides several properties and methods that facilitate paging. The **PagedList.Mvc** package installs a paging helper that displays the paging buttons.

From the **Tools** menu, select **NuGet Package Manager** and then **Manage NuGet Packages for Solution**.

In the **Manage NuGet Packages** dialog box, click the **Online** tab on the left and then enter "paged" in the search box. When you see the **PagedList.Mvc** package, click **Install**.



In the **Select Projects** box, click **OK**.



Add Paging Functionality to the Index Method

In *Controllers\StudentController.cs*, add a `using` statement for the `PagedList` namespace:

```
using PagedList;
```

Replace the `Index` method with the following code:

```
public ViewResult Index(string sortOrder, string currentFilter, string searchString, int? page)
{
   ViewBag.CurrentSort = sortOrder;
   ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
   ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";

   if (searchString != null)
   {
      page = 1;
   }
   else
   {
      searchString = currentFilter;
   }

   ViewBag.CurrentFilter = searchString;

   var students = from s in db.Students
                  select s;
   if (!String.IsNullOrEmpty(searchString))
   {
      students = students.Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())
                           || s.FirstMidName.ToUpper().Contains(searchString.ToUpper()));
   }
   switch (sortOrder)
   {
      case "name_desc":
         students = students.OrderByDescending(s => s.LastName);
         break;
      case "Date":
         students = students.OrderBy(s => s.EnrollmentDate);
         break;
      case "date_desc":
         students = students.OrderByDescending(s => s.EnrollmentDate);
         break;
      default:  // Name ascending
         students = students.OrderBy(s => s.LastName);
         break;
   }

   int pageSize = 3;
   int pageNumber = (page ?? 1);
   return View(students.ToPagedList(pageNumber, pageSize));
}
```

This code adds a `page` parameter, a current sort order parameter, and a current filter parameter to the method signature, as shown here:

```
public ActionResult Index(string sortOrder, string currentFilter, string searchString, int? page)
```

The first time the page is displayed, or if the user hasn't clicked a paging or sorting link, all the parameters will be null. If a paging link is clicked, the `page` variable will contain the page number to display.

`A ViewBag` property provides the view with the current sort order, because this must be included in the paging links in order to keep the sort order the same while paging:

```
ViewBag.CurrentSort = sortOrder;
```

Another property, `ViewBag.CurrentFilter`, provides the view with the current filter string. This value must be included in the paging links in order to maintain the filter settings during paging, and it must be restored to the text box when the page is redisplayed. If the search string is changed during paging, the page has to be reset to 1,

because the new filter can result in different data to display. The search string is changed when a value is entered in the text box and the submit button is pressed. In that case, the `searchString` parameter is not null.

```
if (searchString != null)
        page = 1;
else
    searchString = currentFilter;
```

At the end of the method, the `ToPagedList` extension method on the students `IQueryable` object converts the student query to a single page of students in a collection type that supports paging. That single page of students is then passed to the view:

```
int pageSize = 3;
int pageNumber = (page ?? 1);
return View(students.ToPagedList(pageNumber, pageSize));
```

The `ToPagedList` method takes a page number. The two question marks represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type; the expression `(page ?? 1)` means return the value of `page` if it has a value, or return 1 if `page` is null.

Add Paging Links to the Student Index View

In *Views\Student\Index.cshtml*, replace the existing code with the following code:

```
@model PagedList.IPagedList<ContosoUniversity.Models.Student>
@using PagedList.Mvc;
<link href="~/Content/PagedList.css" rel="stylesheet" type="text/css" />

@{
    ViewBag.Title = "Students";
}

<h2>Students</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
@using (Html.BeginForm("Index", "Student", FormMethod.Get))
{
    <p>
        Find by name: @Html.TextBox("SearchString", ViewBag.CurrentFilter as string)
        <input type="submit" value="Search" />
    </p>
}
<table>
<tr>
    <th></th>
    <th>
        @Html.ActionLink("Last Name", "Index", new { sortOrder=ViewBag.NameSortParm,
currentFilter=ViewBag.CurrentFilter })
    </th>
    <th>
        First Name
    </th>
    <th>
        @Html.ActionLink("Enrollment Date", "Index", new { sortOrder = ViewBag.DateSortParm, currentFilter =
ViewBag.CurrentFilter })
    </th>
</tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.StudentID }) |
            @Html.ActionLink("Details", "Details", new { id=item.StudentID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.StudentID })
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.LastName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.FirstMidName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.EnrollmentDate)
        </td>
    </tr>
}

</table>
<br />
Page @(Model.PageCount < Model.PageNumber ? 0 : Model.PageNumber) of @Model.PageCount

@Html.PagedListPager( Model, page => Url.Action("Index", new { page, sortOrder = ViewBag.CurrentSort,
currentFilter=ViewBag.CurrentFilter }) )
```

The `@model` statement at the top of the page specifies that the view now gets a `PagedList` object instead of a `List` object.

The `using` statement for `PagedList.Mvc` gives access to the MVC helper for the paging buttons.

The code uses an overload of BeginForm that allows it to specify FormMethod.Get.

```
@using (Html.BeginForm("Index", "Student", FormMethod.Get))
{
    <p>
        Find by name: @Html.TextBox("SearchString", ViewBag.CurrentFilter as string)
        <input type="submit" value="Search" />
    </p>
}
```

The default BeginForm submits form data with a POST, which means that parameters are passed in the HTTP message body and not in the URL as query strings. When you specify HTTP GET, the form data is passed in the URL as query strings, which enables users to bookmark the URL. The W3C guidelines for the use of HTTP GET specify that you should use GET when the action does not result in an update.

The text box is initialized with the current search string so when you click a new page you can see the current search string.

```
Find by name: @Html.TextBox("SearchString", ViewBag.CurrentFilter as string)
```

The column header links use the query string to pass the current search string to the controller so that the user can sort within filter results:

```
@Html.ActionLink("Last Name", "Index", new { sortOrder=ViewBag.NameSortParm,
currentFilter=ViewBag.CurrentFilter })
```

The current page and total number of pages are displayed.

```
Page @(Model.PageCount < Model.PageNumber ? 0 : Model.PageNumber) of @Model.PageCount
```

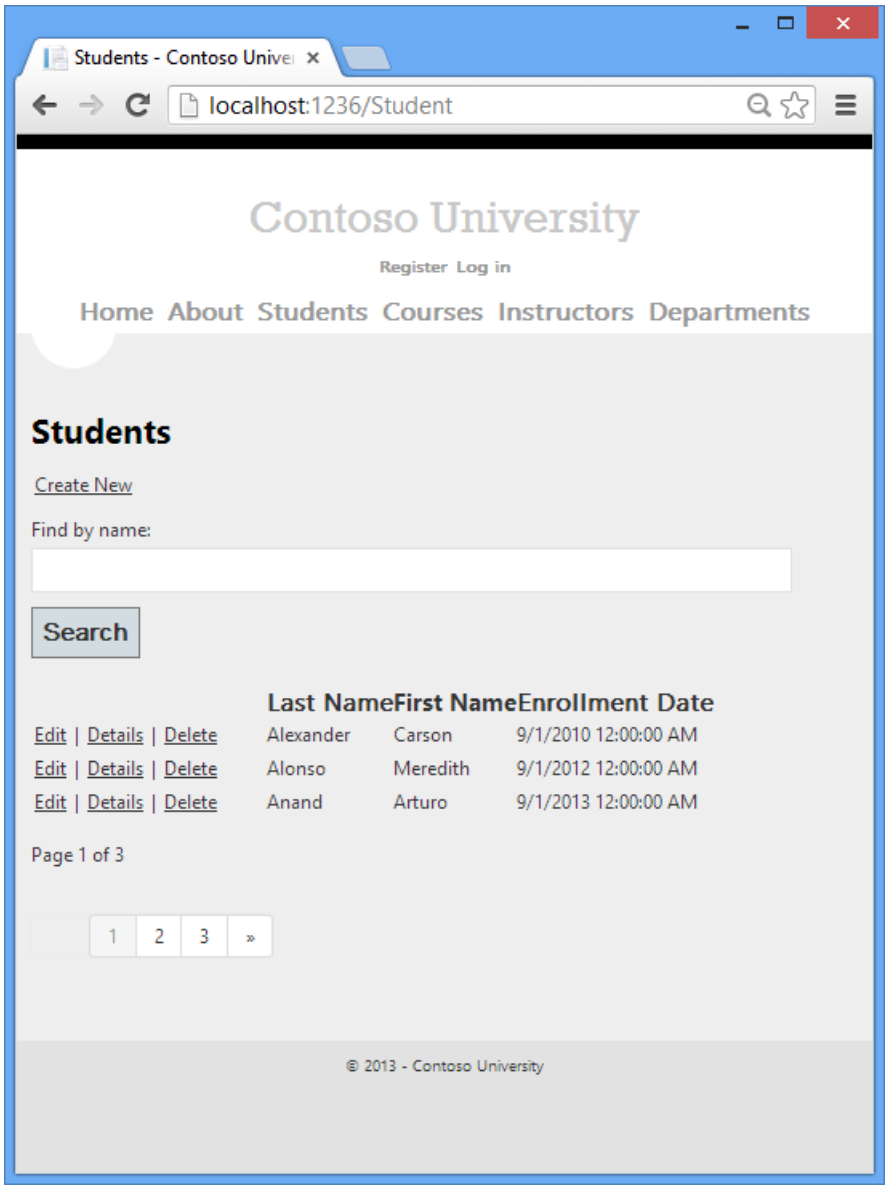
If there are no pages to display, "Page 0 of 0" is shown. (In that case the page number is greater than the page count because `Model.PageNumber` is 1, and `Model.PageCount` is 0.)

The paging buttons are displayed by the `PagedListPager` helper:

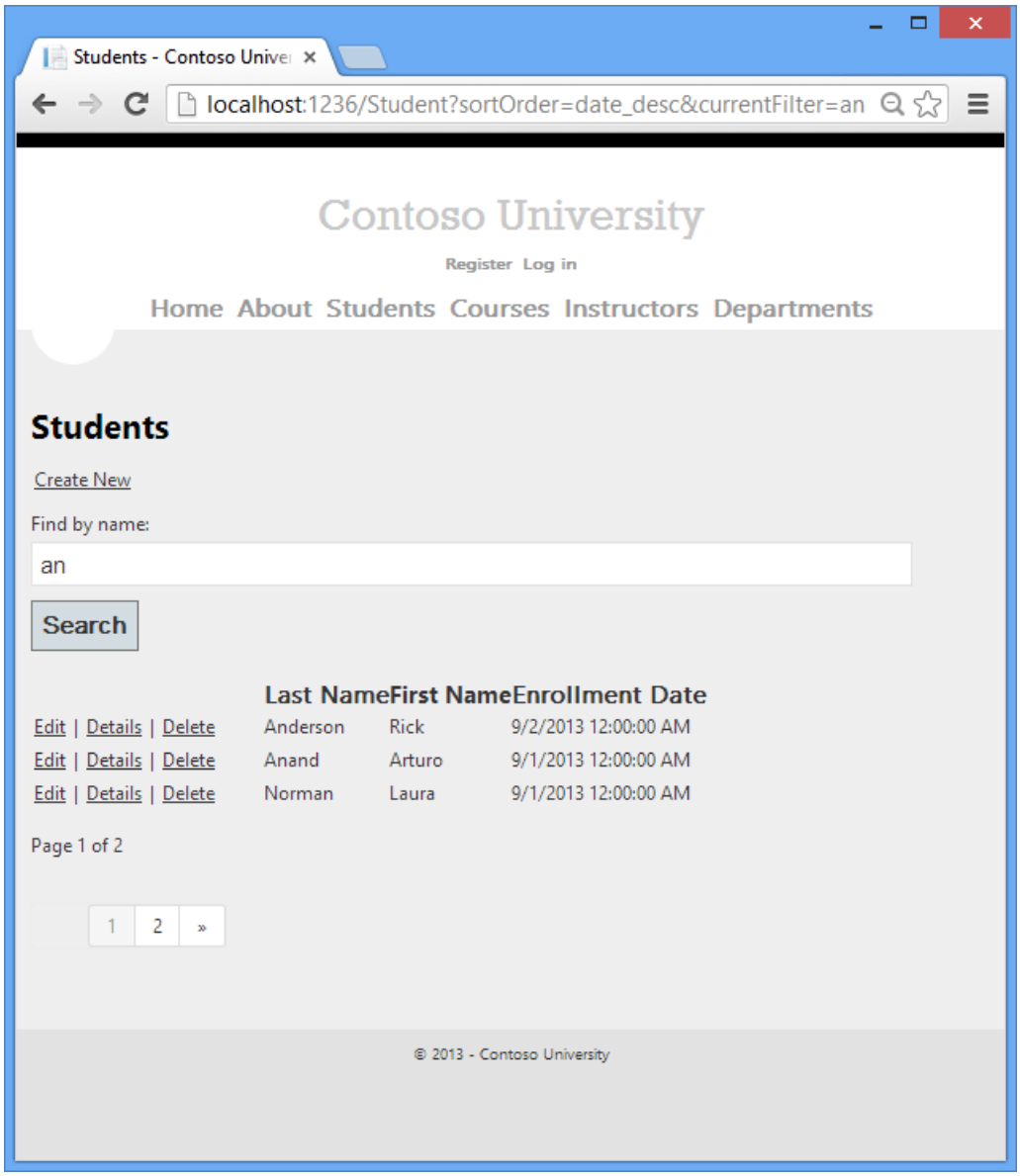
```
@Html.PagedListPager( Model, page => Url.Action("Index", new { page }) )
```

The `PagedListPager` helper provides a number of options that you can customize, including URLs and styling. For more information, see TroyGoode / PagedList on the GitHub site.

Run the page.



Click the paging links in different sort orders to make sure paging works. Then enter a search string and try paging again to verify that paging also works correctly with sorting and filtering.



Create an About Page That Shows Student Statistics

For the Contoso University website's About page, you'll display how many students have enrolled for each enrollment date. This requires grouping and simple calculations on the groups. To accomplish this, you'll do the following:

- Create a view model class for the data that you need to pass to the view.
- Modify the `About` method in the `Home` controller.
- Modify the `About` view.

Create the View Model

Create a *ViewModels* folder. In that folder, add a class file *EnrollmentDateGroup.cs* and replace the existing code with the following code:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.ViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Modify the Home Controller

In *HomeController.cs*, add the following `using` statements at the top of the file:

```
using ContosoUniversity.DAL;
using ContosoUniversity.ViewModels;
```

Add a class variable for the database context immediately after the opening curly brace for the class:

```
public class HomeController : Controller
{
    private SchoolContext db = new SchoolContext();
```

Replace the `About` method with the following code:

```
public ActionResult About()
{
    var data = from student in db.Students
               group student by student.EnrollmentDate into dateGroup
               select new EnrollmentDateGroup()
               {
                   EnrollmentDate = dateGroup.Key,
                   StudentCount = dateGroup.Count()
               };
    return View(data);
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Add a `Dispose` method:

```
protected override void Dispose(bool disposing)
{
    db.Dispose();
    base.Dispose(disposing);
}
```

Modify the About View

Replace the code in the *Views\Home\About.cshtml* file with the following code:

```
@model IEnumerable<ContosoUniversity.ViewModels.EnrollmentDateGroup>

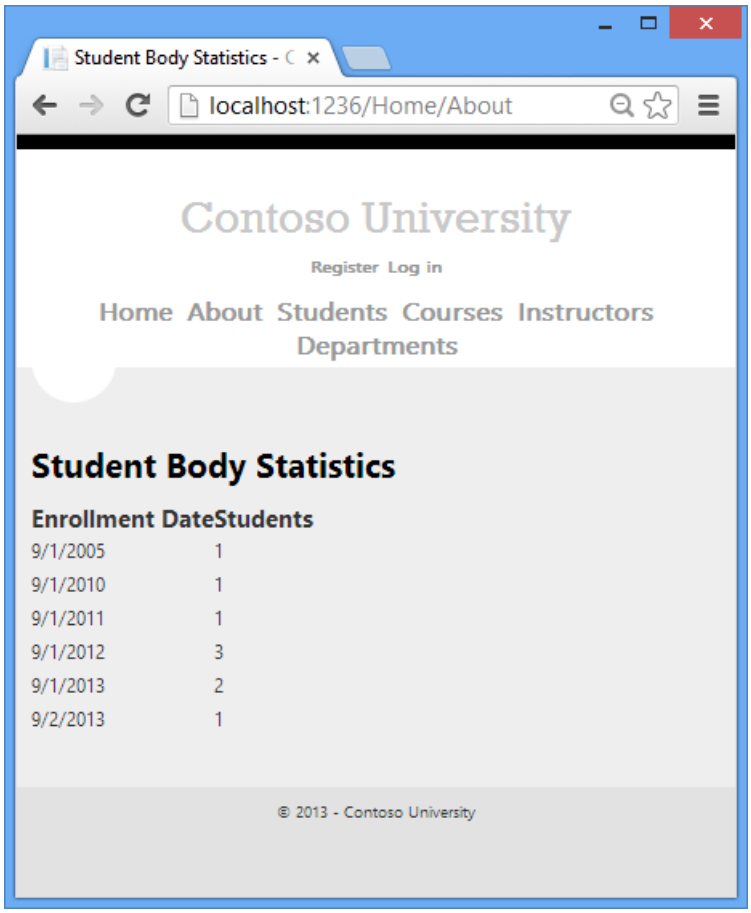
@{
    ViewBag.Title = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.EnrollmentDate)
        </td>
        <td>
            @item.StudentCount
        </td>
    </tr>
}
</table>
```

Run the app and click the **About** link. The count of students for each enrollment date is displayed in a table.



Optional: Deploy the app to Windows Azure

So far your application has been running locally in IIS Express on your development computer. To make it available

for other people to use over the Internet, you have to deploy it to a web hosting provider. In this optional section of the tutorial you'll deploy it to a Windows Azure Web Site.

Using Code First Migrations to Deploy the Database

To deploy the database you'll use Code First Migrations. When you create the publish profile that you use to configure settings for deploying from Visual Studio, you'll select a check box that is labeled **Execute Code First Migrations (runs on application start)**. This setting causes the deployment process to automatically configure the application *Web.config* file on the destination server so that Code First uses the `MigrateDatabaseToLatestVersion` initializer class.

Visual Studio does not do anything with the database during the deployment process. When the deployed application accesses the database for the first time after deployment, Code First automatically creates the database or updates the database schema to the latest version. If the application implements a Migrations `Seed` method, the method runs after the database is created or the schema is updated.

Your Migrations `Seed` method inserts test data. If you were deploying to a production environment, you would have to change the `Seed` method so that it only inserts data that you want to be inserted into your production database. For example, in your current data model you might want to have real courses but fictional students in the development database. You can write a `Seed` method to load both in development, and then comment out the fictional students before you deploy to production. Or you can write a `Seed` method to load only courses, and enter the fictional students in the test database manually by using the application's UI.

Get a Windows Azure account

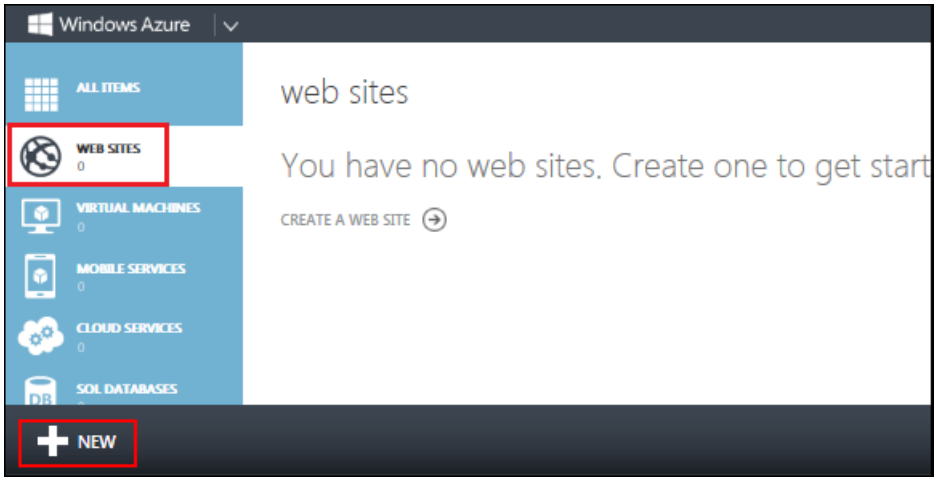
You'll need a Windows Azure account. If you don't already have one, you can create a free trial account in just a couple of minutes. For details, see Windows Azure Free Trial.

Create a web site and a SQL database in Windows Azure

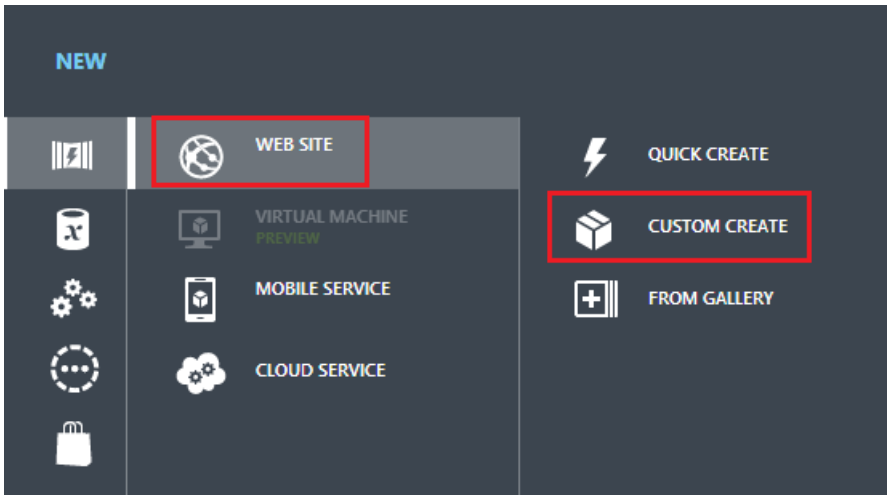
Your Windows Azure Web Site will run in a shared hosting environment, which means it runs on virtual machines (VMs) that are shared with other Windows Azure clients. A shared hosting environment is a low-cost way to get started in the cloud. Later, if your web traffic increases, the application can scale to meet the need by running on dedicated VMs. If you need a more complex architecture, you can migrate to a Windows Azure Cloud Service. Cloud services run on dedicated VMs that you can configure according to your needs.

Windows Azure SQL Database is a cloud-based relational database service that is built on SQL Server technologies. Tools and applications that work with SQL Server also work with SQL Database.

1. In the Windows Azure Management Portal, click **Web Sites** in the left tab, and then click **New**.

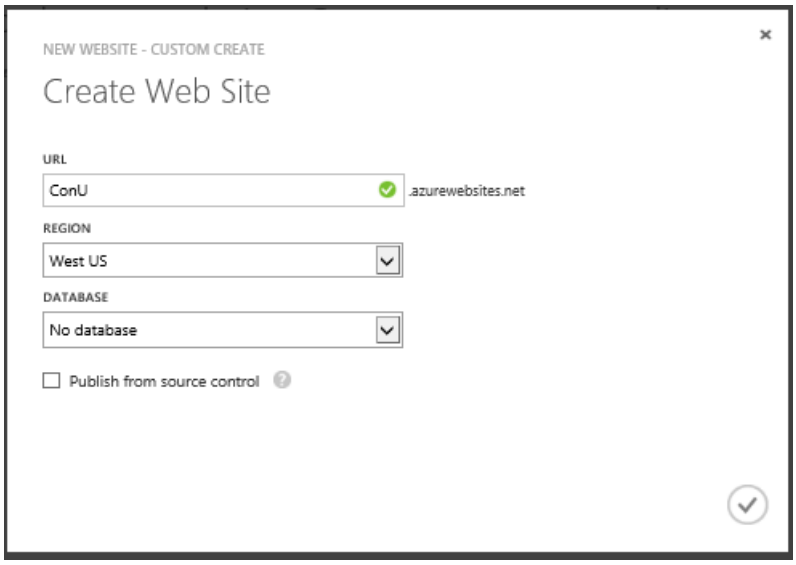


2. Click **CUSTOM CREATE**.



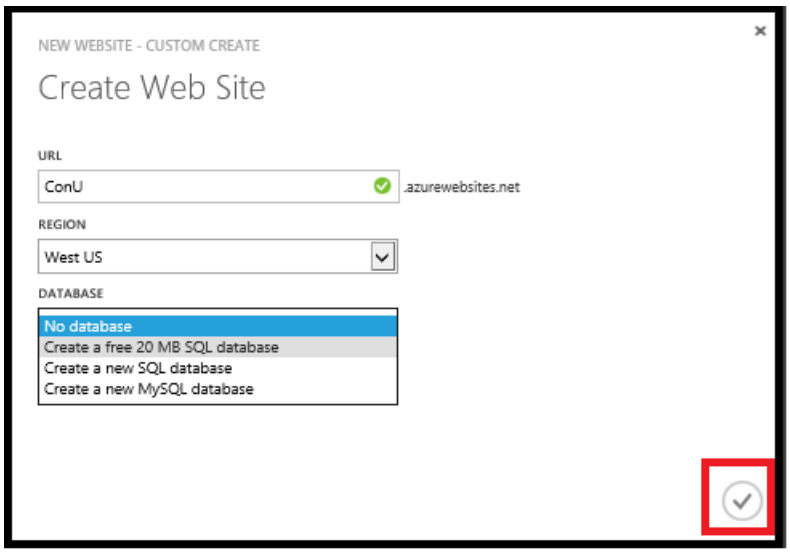
The **New Web Site - Custom Create** wizard opens.

3. In the **New Web Site** step of the wizard, enter a string in the **URL** box to use as the unique URL for your application. The complete URL will consist of what you enter here plus the suffix that you see next to the text box. The illustration shows "ConU", but that URL is probably taken so you will have to choose a different one.

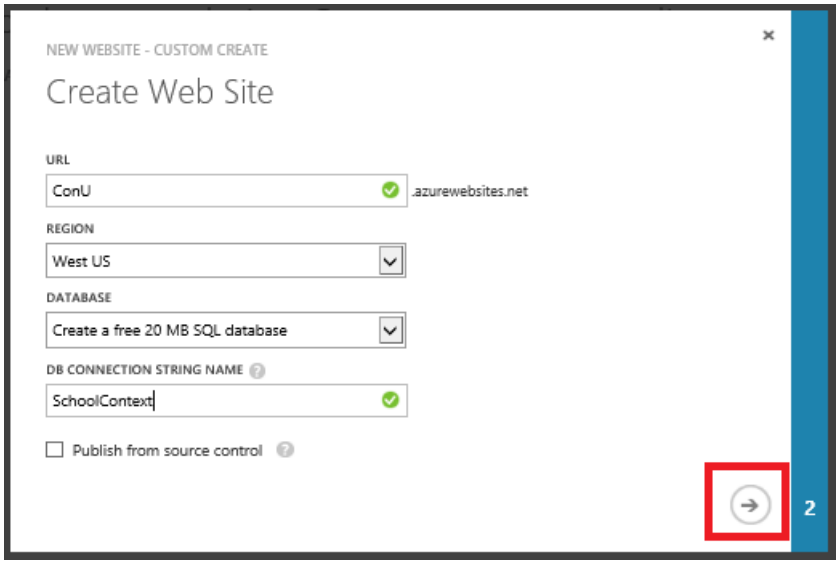


4. In the **Region** drop-down list, choose a region close to you. This setting specifies which data center your web site will run in.

5. In the **Database** drop-down list, choose **Create a free 20 MB SQL database**.



6. In the **DB CONNECTION STRING NAME**, enter *SchoolContext*.



7. Click the arrow that points to the right at the bottom of the box. The wizard advances to the **Database Settings** step.

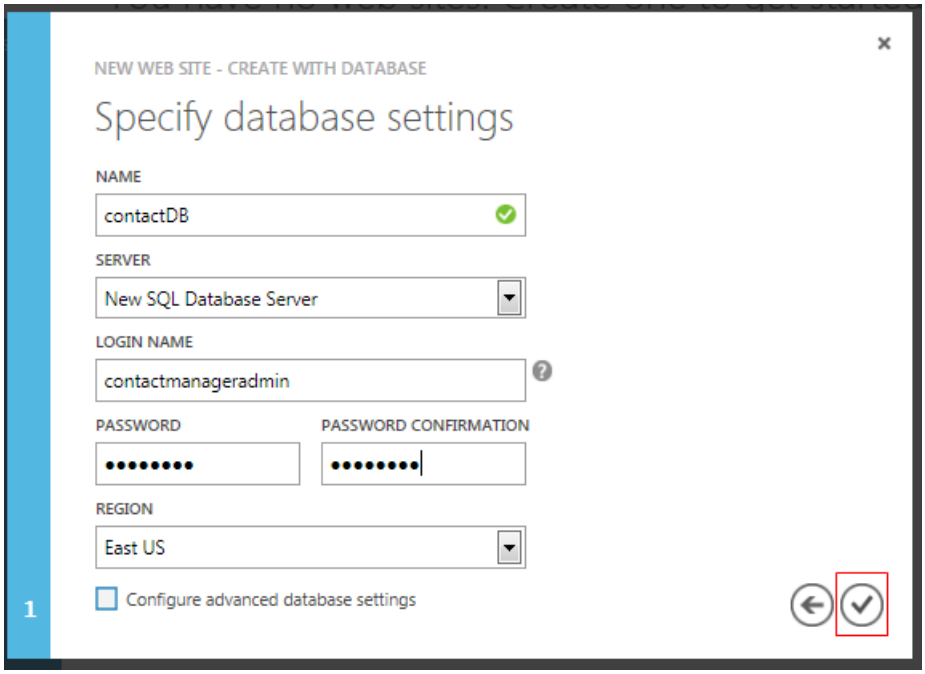
8. In the **Name** box, enter *ContosoUniversityDB*.

9. In the **Server** box, select **New SQL Database server**. Alternatively, if you previously created a server, you can select that server from the drop-down list.

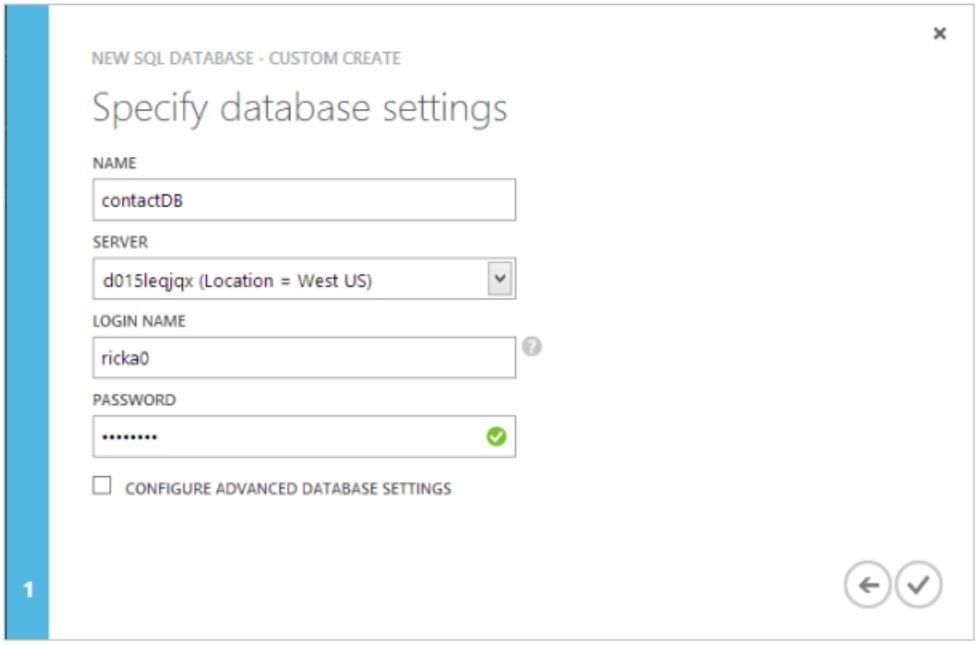
10. Enter an administrator **LOGIN NAME** and **PASSWORD**. If you selected **New SQL Database server** you aren't entering an existing name and password here, you're entering a new name and password that you're defining now to use later when you access the database. If you selected a server that you created previously, you'll enter credentials for that server. For this tutorial, you won't select the ***Advanced*** check box. The ***Advanced*** options enable you to set the database collation.

11. Choose the same **Region** that you chose for the web site.

12. Click the check mark at the bottom right of the box to indicate that you're finished.



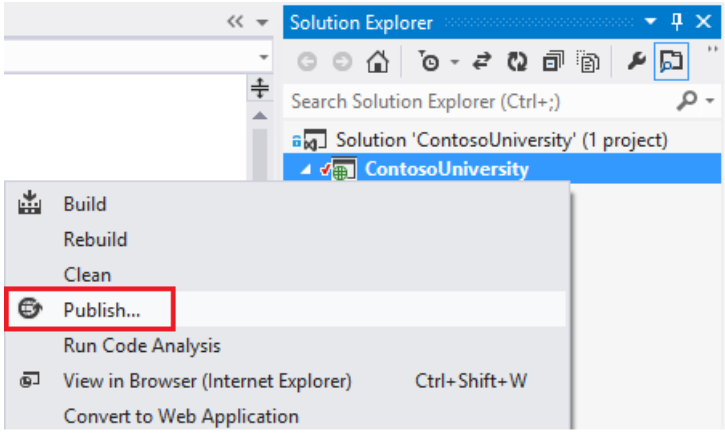
The following image shows using an existing SQL Server and Login.



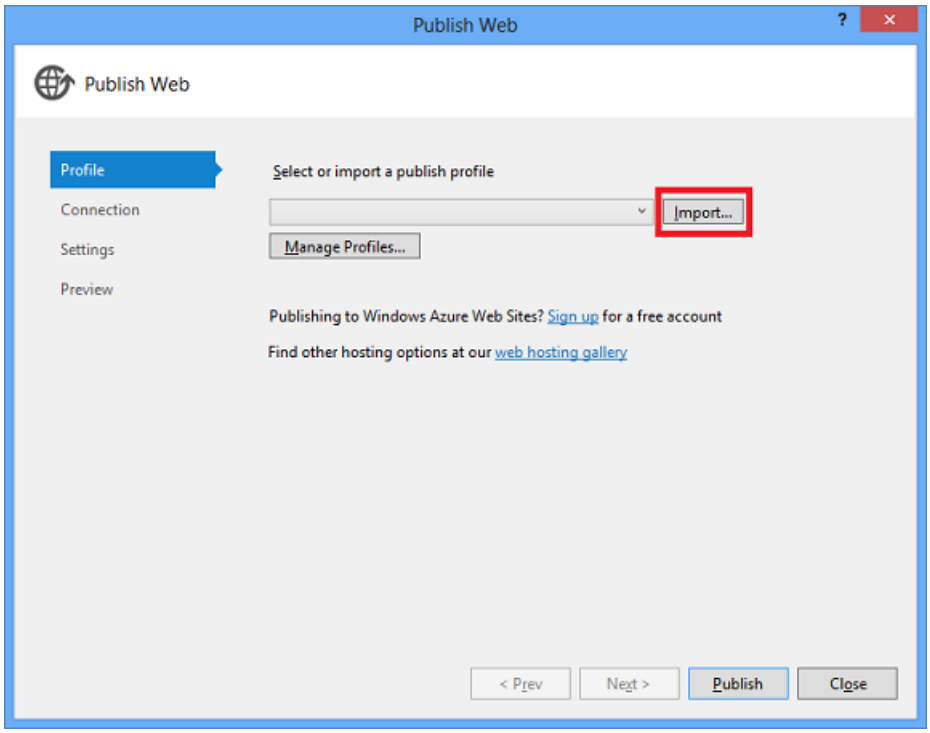
The Management Portal returns to the Web Sites page, and the **Status** column shows that the site is being created. After a while (typically less than a minute), the **Status** column shows that the site was successfully created. In the navigation bar at the left, the number of sites you have in your account appears next to the **Web Sites** icon, and the number of databases appears next to the **SQL Databases** icon.

Deploy the application to Windows Azure

1. In Visual Studio, right-click the project in **Solution Explorer** and select **Publish** from the context menu.

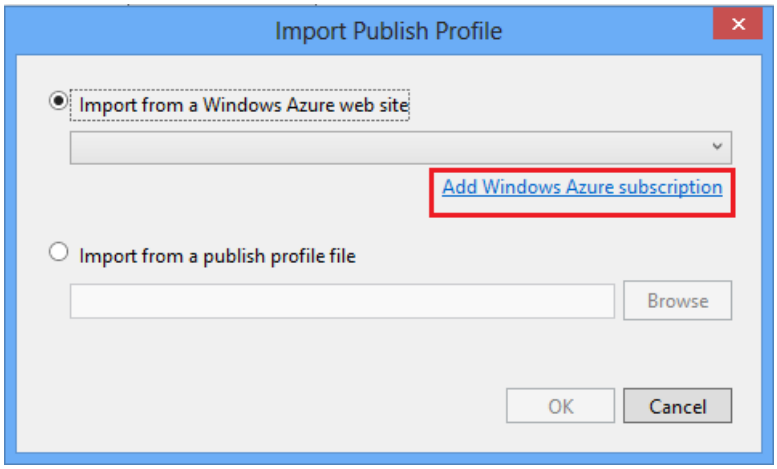


2. In the **Profile** tab of the **Publish Web** wizard, click **Import**.

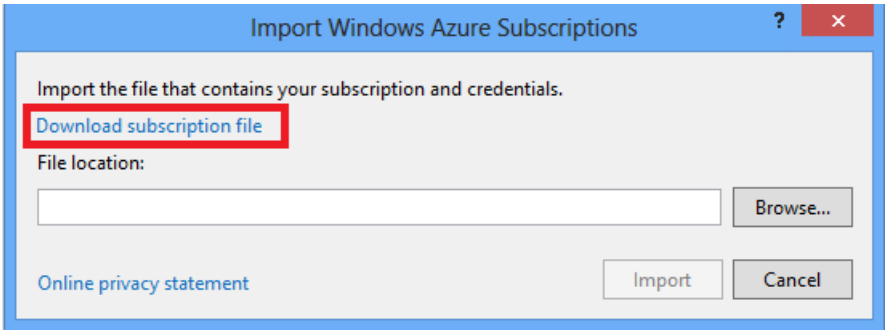


3. If you have not previously added your Windows Azure subscription in Visual Studio, perform the following steps. In these steps you add your subscription so that the drop-down list under **Import from a Windows Azure web site** will include your web site.

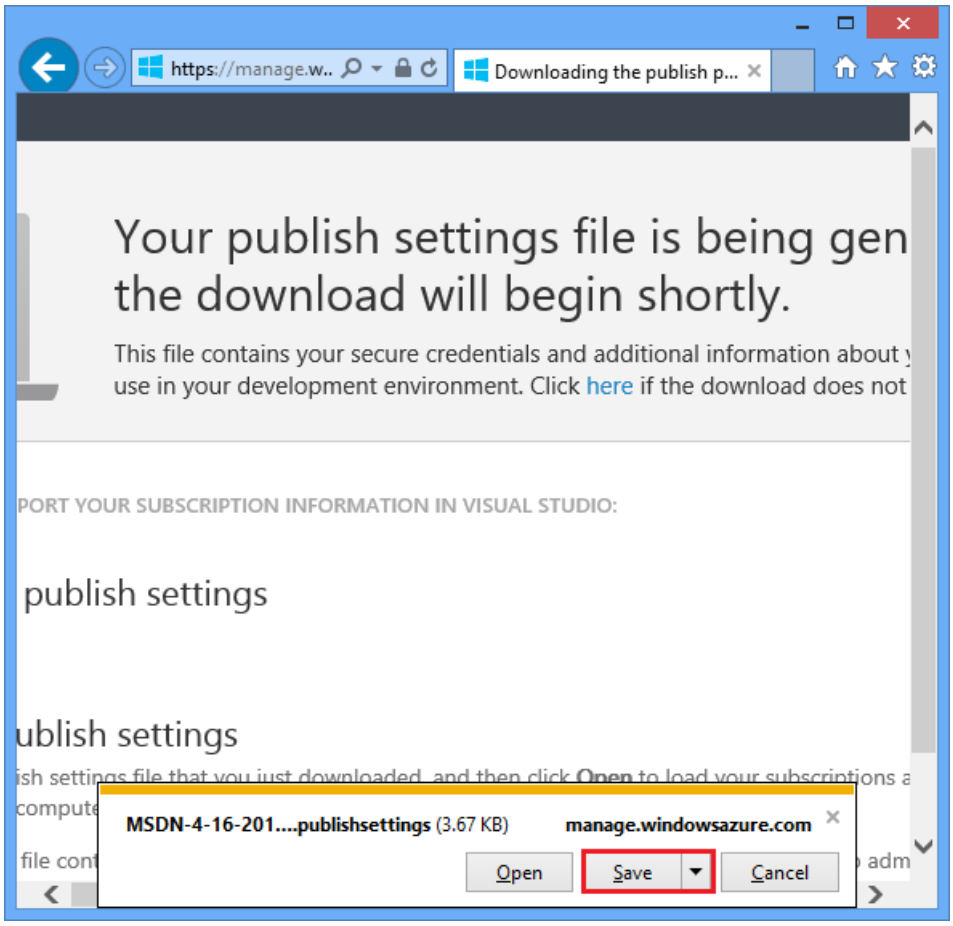
   a. In the **Import Publish Profile** dialog box, click **Import from a Windows Azure web site**, and then click **Add Windows Azure subscription**.



   b. In the **Import Windows Azure Subscriptions** dialog box, click **Download subscription file**.



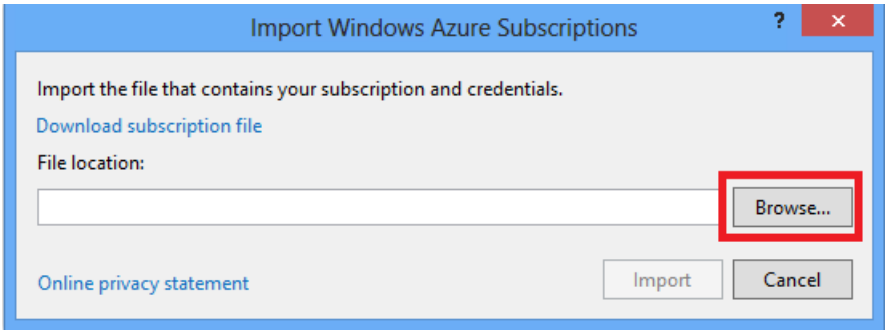
   c. In your browser window, save the *.publishsettings* file.



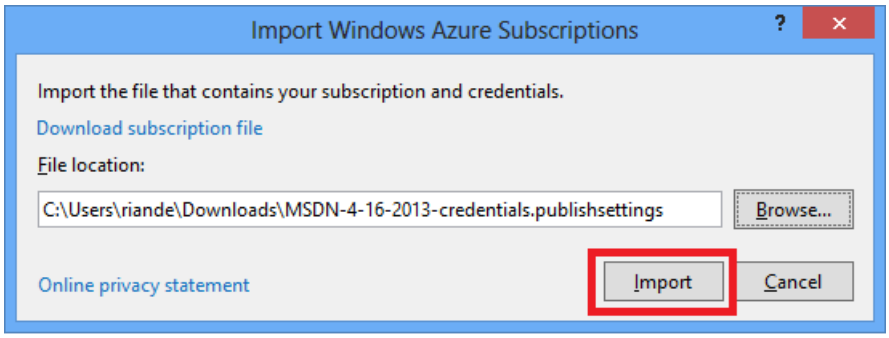
WARNING

Security - The *publishsettings* file contains your credentials (unencoded) that are used to administer your Windows Azure subscriptions and services. The security best practice for this file is to store it temporarily outside your source directories (for example in the *Libraries\Documents* folder), and then delete it once the import has completed. A malicious user who gains access to the `.publishsettings` file can edit, create, and delete your Windows Azure services.

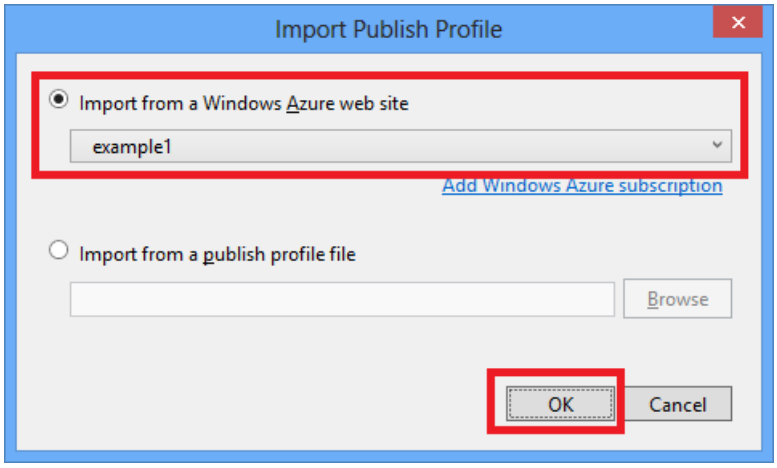
d. In the **Import Windows Azure Subscriptions** dialog box, click **Browse** and navigate to the *.publishsettings* file.



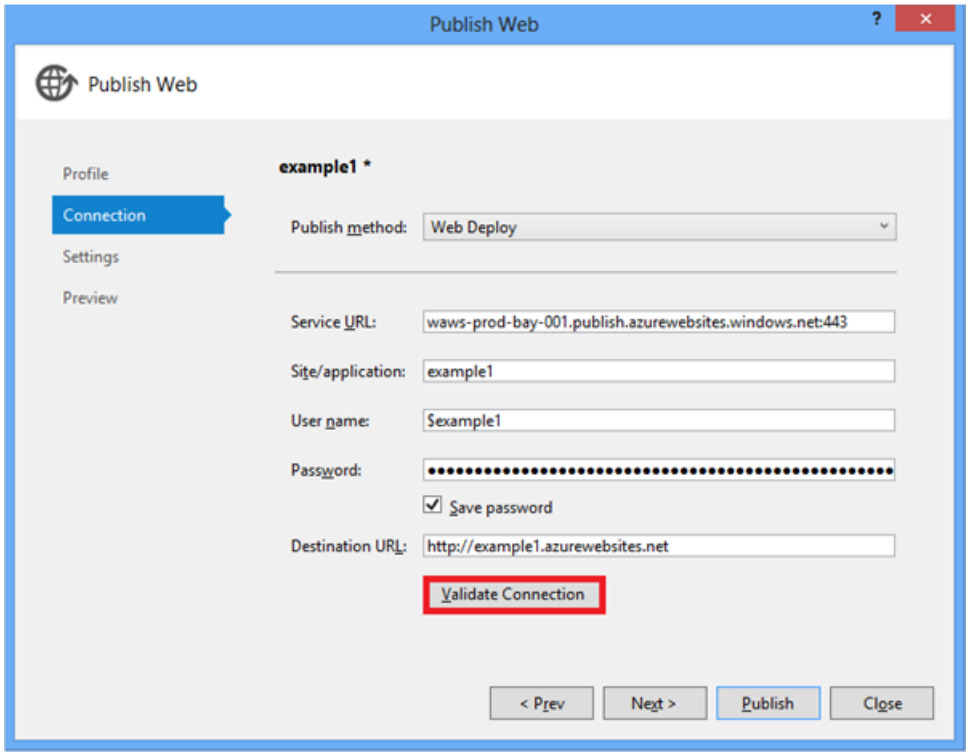
e. Click **Import**.



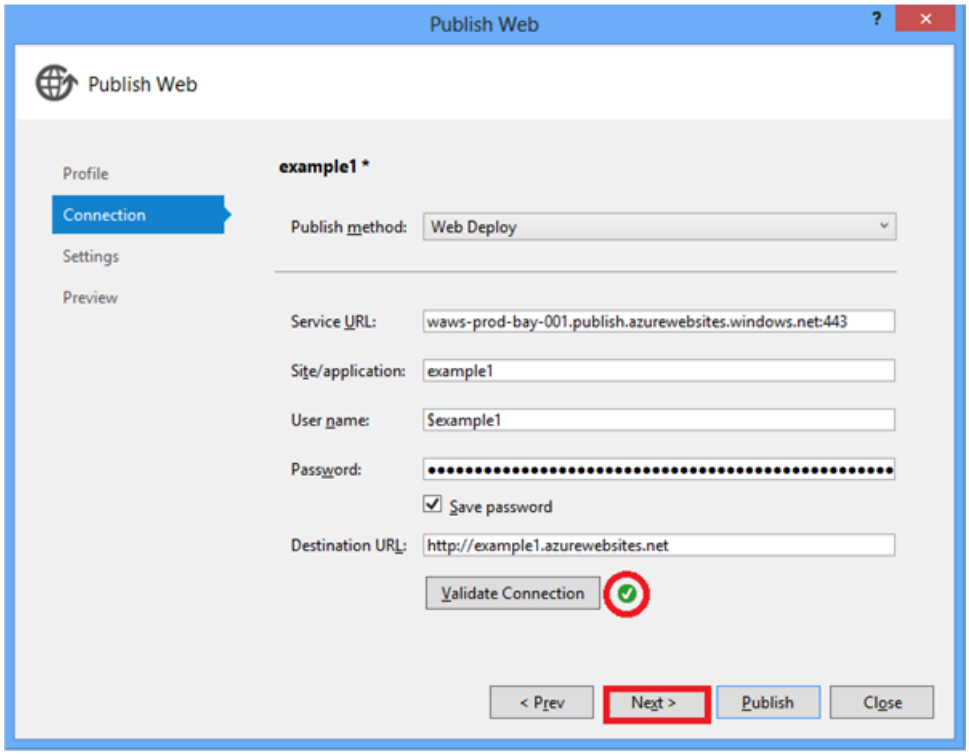
4. In the **Import Publish Profile** dialog box, select **Import from a Windows Azure web site**, select your web site from the drop-down list, and then click **OK**.



5. In the **Connection** tab, click **Validate Connection** to make sure that the settings are correct.



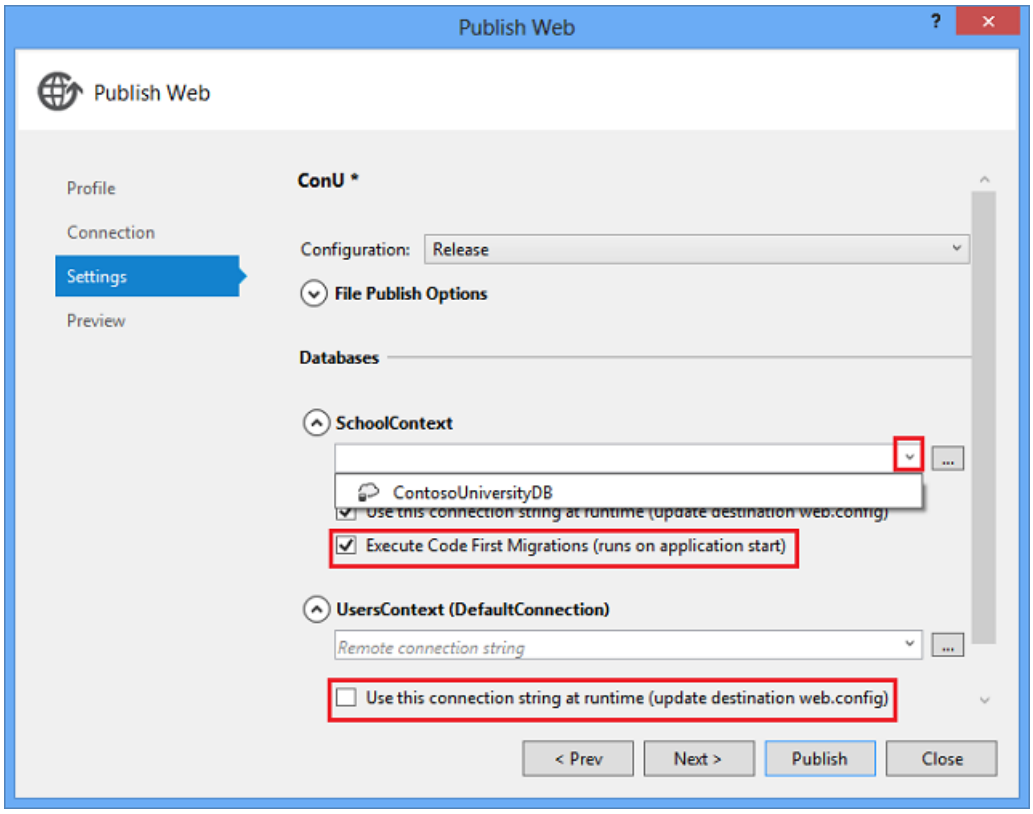
6. When the connection has been validated, a green check mark is shown next to the **Validate Connection** button. Click **Next**.



7. Open the **Remote connection string** drop-down list under **SchoolContext** and select the connection string for the database you created.

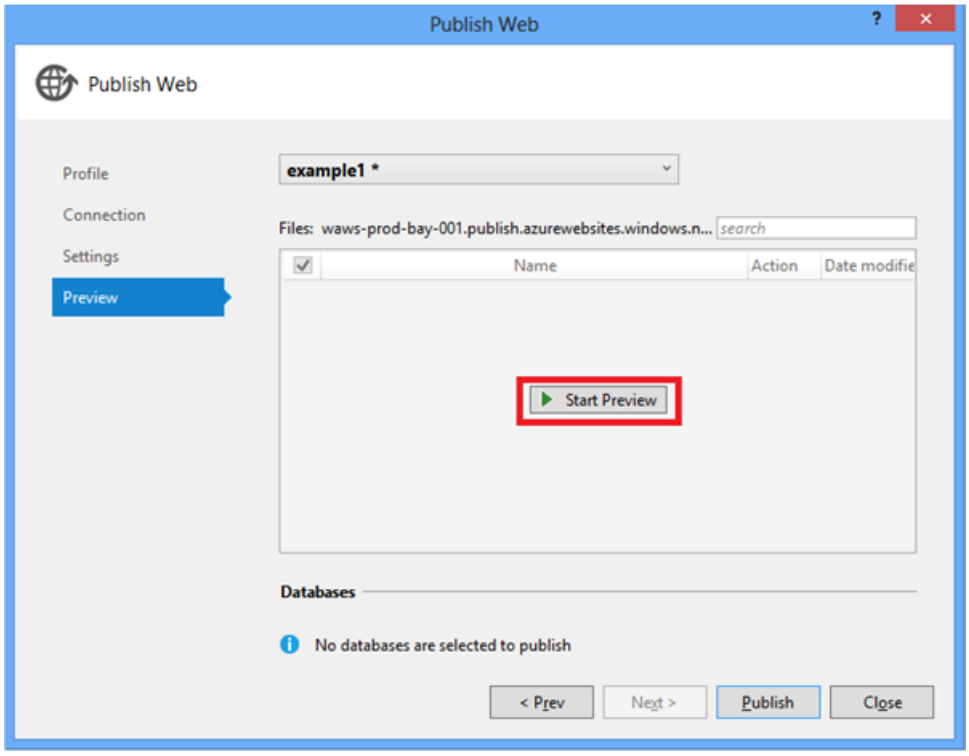
8. Select **Execute Code First Migrations (runs on application start)**.

9. Uncheck **Use this connection string at runtime** for the **UserContext (DefaultConnection)**, since this application is not using the membership database.

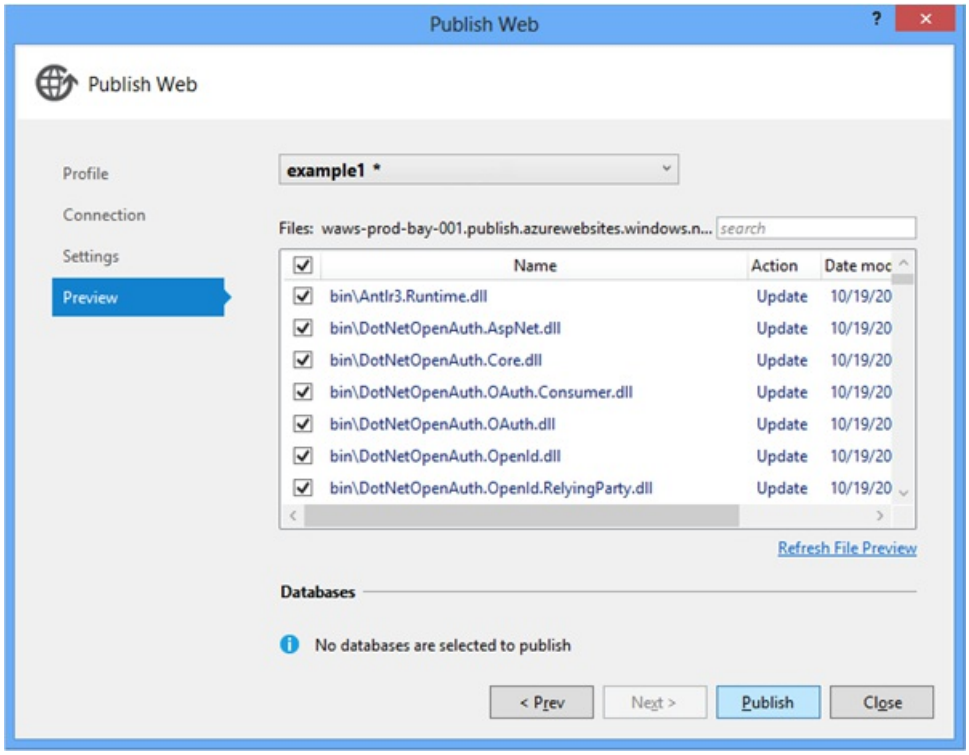


10. Click **Next**.

11. In the **Preview** tab, click **Start Preview**.



The tab displays a list of the files that will be copied to the server. Displaying the preview isn't required to publish the application but is a useful function to be aware of. In this case, you don't need to do anything with the list of files that is displayed. The next time you deploy this application, only the files that have changed will be in this list.

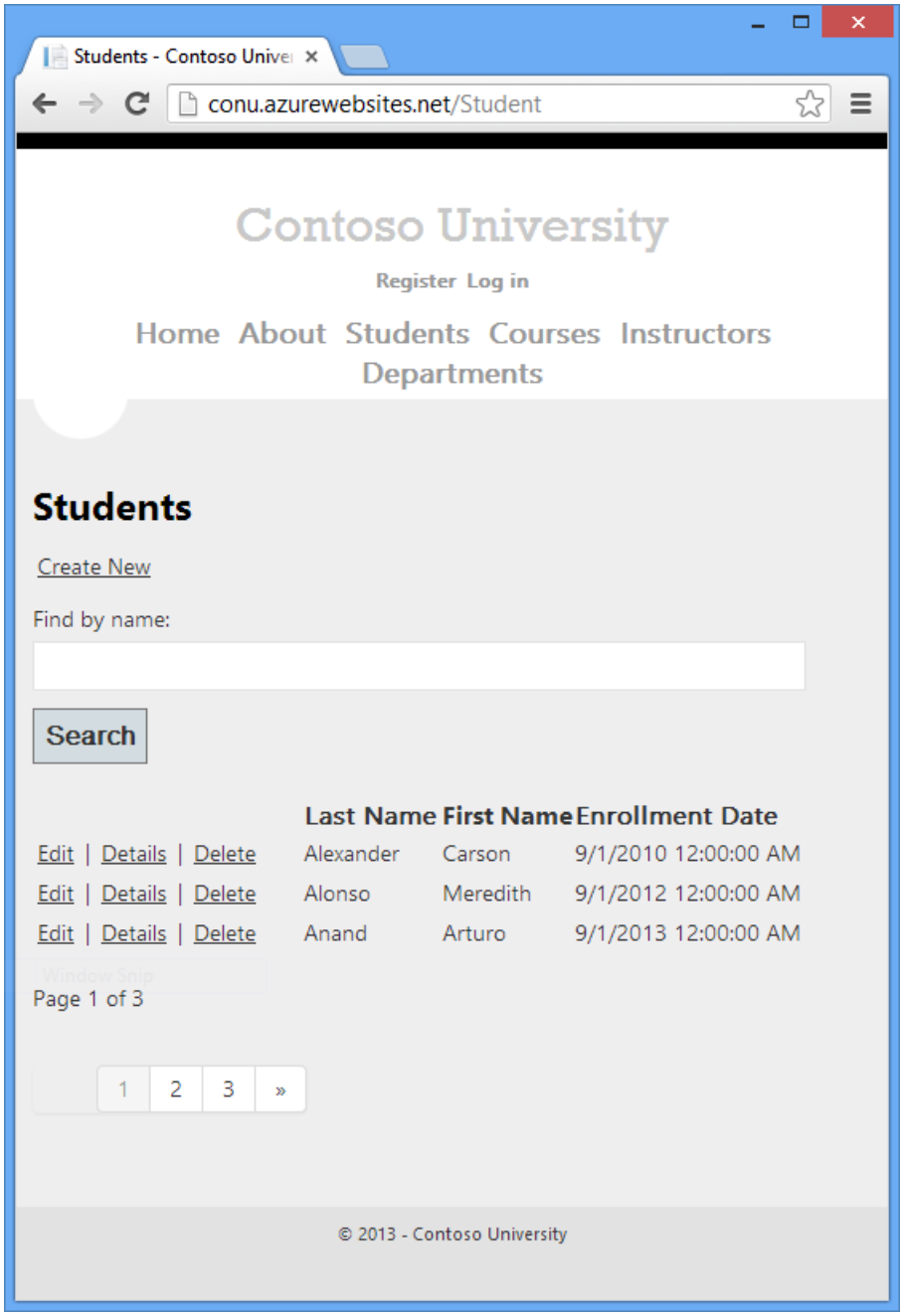


12. Click **Publish**.
    Visual Studio begins the process of copying the files to the Windows Azure server.

13. The **Output** window shows what deployment actions were taken and reports successful completion of the deployment.

```
Output
Show output from: Build
2>Adding file (example1\Views\Home\Contact.cshtml).
2>Adding file (example1\Views\Home\Index.cshtml).
2>Adding file (example1\Views\Shared\Error.cshtml).
2>Adding file (example1\Views\Shared\_Layout.cshtml).
2>Adding file (example1\Views\Shared\_LoginPartial.cshtml).
2>Adding file (example1\Views\Web.config).
2>Adding file (example1\Views\_ViewStart.cshtml).
2>Adding file (example1\Web.config).
2>Adding ACL's for path (example1)
2>Adding ACL's for path (example1)
2>Publish is successfully deployed.
2>Site was published successfully http://example1.azurewebsites.net/
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
========== Publish: 1 succeeded, 0 failed, 0 skipped ==========
```

14. Upon successful deployment, the default browser automatically opens to the URL of the deployed web site. The application you created is now running in the cloud. Click the Students tab.



At this point your *SchoolContext* database has been created in the Windows Azure SQL Database because you selected **Execute Code First Migrations (runs on app start)**. The *Web.config* file in the deployed web site has been changed so that the MigrateDatabaseToLatestVersion initializer would run the first time your code reads or writes data in the database (which happened when you selected the **Students** tab):

```
</runtime>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFa
      <parameters>
        <parameter value="v11.0" />
      </parameters>
    </defaultConnectionFactory>
    <contexts>
      <context type="ContosoUniversity.Models.SchoolContext, ContosoUniversity">
        <databaseInitializer type="System.Data.Entity.MigrateDatabaseToLatestVersion`2[
                              [ContosoUniversity.Models.SchoolContext, ContosoUniversity],
                              [ContosoUniversity.Migrations.Configuration, ContosoUniversi
                              EntityFramework, PublicKeyToken=b77a5c561934e089">
          <parameters>
            <parameter value="SchoolContext_DatabasePublish"/>
          </parameters>
        </databaseInitializer>
      </context>
    </contexts>
  </entityFramework>
</configuration>
```

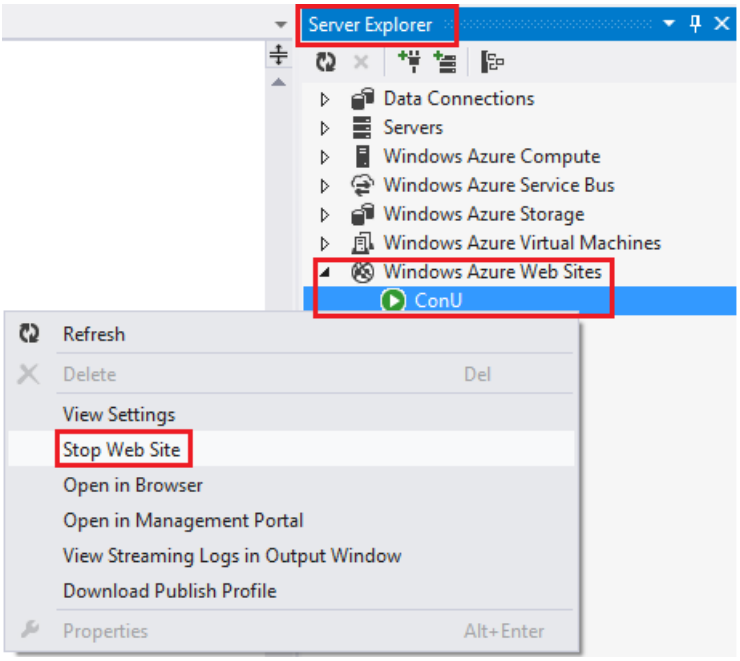
The deployment process also created a new connection string *(SchoolContext_DatabasePublish)* for Code First Migrations to use for updating the database schema and seeding the database.

```
<connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\v11.0;Initial
    <add name="SchoolContext" connectionString="Data Source=tcp:d015leqjqx.database.winc
    <add name="SchoolContext_DatabasePublish" connectionString="Data Source=tcp:d015leq
</connectionStrings>
```

The *DefaultConnection* connection string is for the membership database (which we are not using in this tutorial). The *SchoolContext* connection string is for the ContosoUniversity database.

You can find the deployed version of the Web.config file on your own computer in *ContosoUniversity\obj\Release\Package\PackageTmp\Web.config*. You can access the deployed *Web.config* file itself by using FTP. For instructions, see ASP.NET Web Deployment using Visual Studio: Deploying a Code Update. Follow the instructions that start with "To use an FTP tool, you need three things: the FTP URL, the user name, and the password."

> **NOTE**
>
> The web app doesn't implement security, so anyone who finds the URL can change the data. For instructions on how to secure the web site, see Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL Database to a Windows Azure Web Site. You can prevent other people from using the site by using the Windows Azure Management Portal or **Server Explorer** in Visual Studio to stop the site.



Code First Initializers

In the deployment section you saw the MigrateDatabaseToLatestVersion initializer being used. Code First also provides other initializers that you can use, including CreateDatabaseIfNotExists (the default), DropCreateDatabaseIfModelChanges and DropCreateDatabaseAlways. The `DropCreateAlways` initializer can be useful for setting up conditions for unit tests. You can also write your own initializers, and you can call an initializer explicitly if you don't want to wait until the application reads from or writes to the database. For a comprehensive explanation of initializers, see chapter 6 of the book Programming Entity Framework: Code First by Julie Lerman and Rowan Miller.

Summary

In this tutorial you've seen how to create a data model and implement basic CRUD, sorting, filtering, paging, and grouping functionality. In the next tutorial you'll begin looking at more advanced topics by expanding the data model.

Links to other Entity Framework resources can be found in the ASP.NET Data Access Content Map.

PREVIOUS NEXT

Creating a More Complex Data Model for an ASP.NET MVC Application (4 of 10)

2/20/2020 • 31 minutes to read • Edit Online

by Tom Dykstra

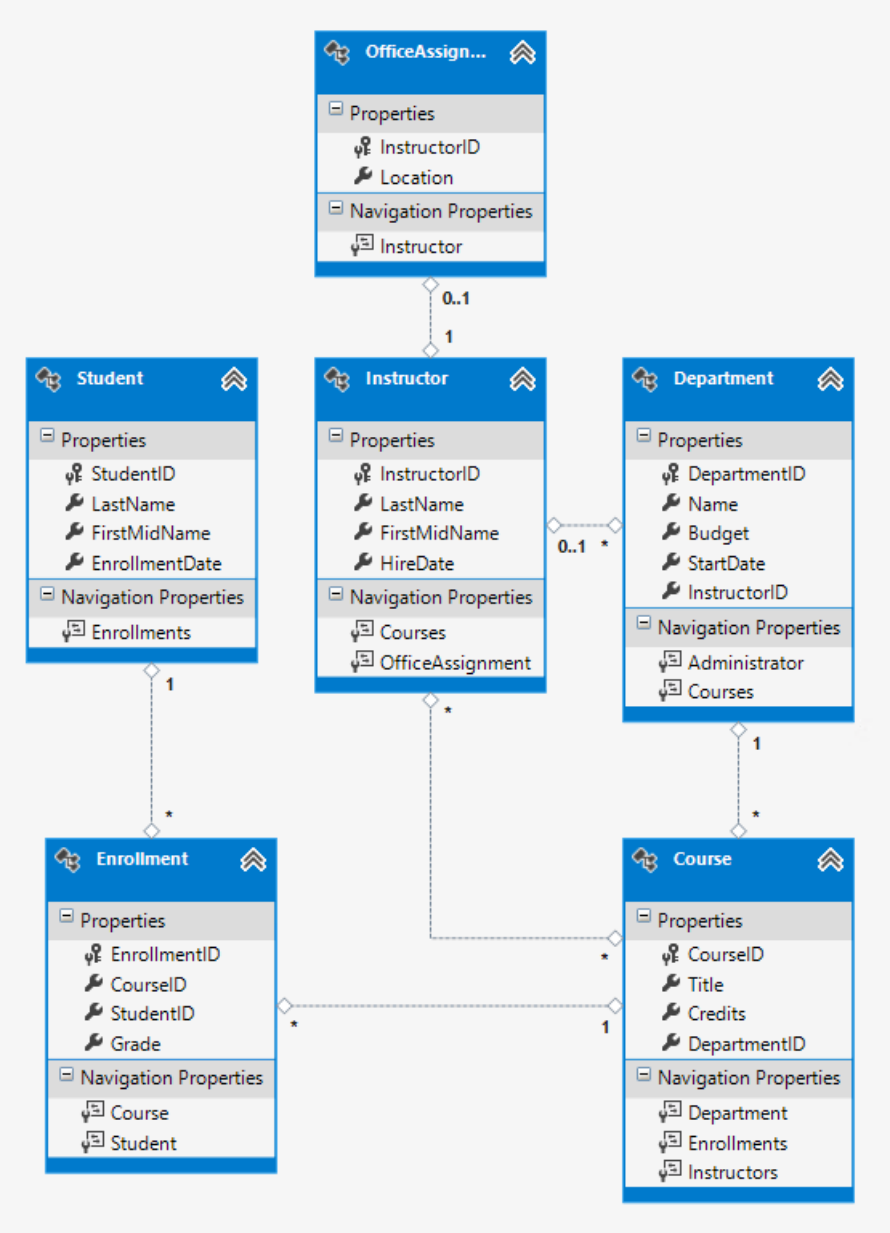
Download Completed Project

The Contoso University sample web application demonstrates how to create ASP.NET MVC 4 applications using the Entity Framework 5 Code First and Visual Studio 2012. For information about the tutorial series, see the first tutorial in the series. You can start the tutorial series from the beginning or download a starter project for this chapter and start here.

> **NOTE**
>
> If you run into a problem you can't resolve, download the completed chapter and try to reproduce your problem. You can generally find the solution to the problem by comparing your code to the completed code. For some common errors and how to solve them, see Errors and Workarounds.

In the previous tutorials you worked with a simple data model that was composed of three entities. In this tutorial you'll add more entities and relationships and you'll customize the data model by specifying formatting, validation, and database mapping rules. You'll see two ways to customize the data model: by adding attributes to entity classes and by adding code to the database context class.

When you're finished, the entity classes will make up the completed data model that's shown in the following illustration:



Customize the Data Model by Using Attributes

In this section you'll see how to customize the data model by using attributes that specify formatting, validation, and database mapping rules. Then in several of the following sections you'll create the complete `School` data model by adding attributes to the classes you already created and creating new classes for the remaining entity types in the model.

The DataType Attribute

For student enrollment dates, all of the web pages currently display the time along with the date, although all you care about for this field is the date. By using data annotation attributes, you can make one code change that will fix the display format in every view that shows the data. To see an example of how to do that, you'll add an attribute to the `EnrollmentDate` property in the `Student` class.

In *Models\Student.cs*, add a `using` statement for the `System.ComponentModel.DataAnnotations` namespace and add `DataType` and `DisplayFormat` attributes to the `EnrollmentDate` property, as shown in the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int StudentID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The DataType attribute is used to specify a data type that is more specific than the database intrinsic type. In this case we only want to keep track of the date, not the date and time. The DataType Enumeration provides for many data types, such as *Date, Time, PhoneNumber, Currency, EmailAddress* and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for DataType.EmailAddress, and a date selector can be provided for DataType.Date in browsers that support HTML5. The DataType attributes emits HTML 5 data- (pronounced *data dash*) attributes that HTML 5 browsers can understand. The DataType attributes do not provide any validation.

`DataType.Date` does not specify the format of the date that is displayed. By default, the data field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime EnrollmentDate { get; set; }
```

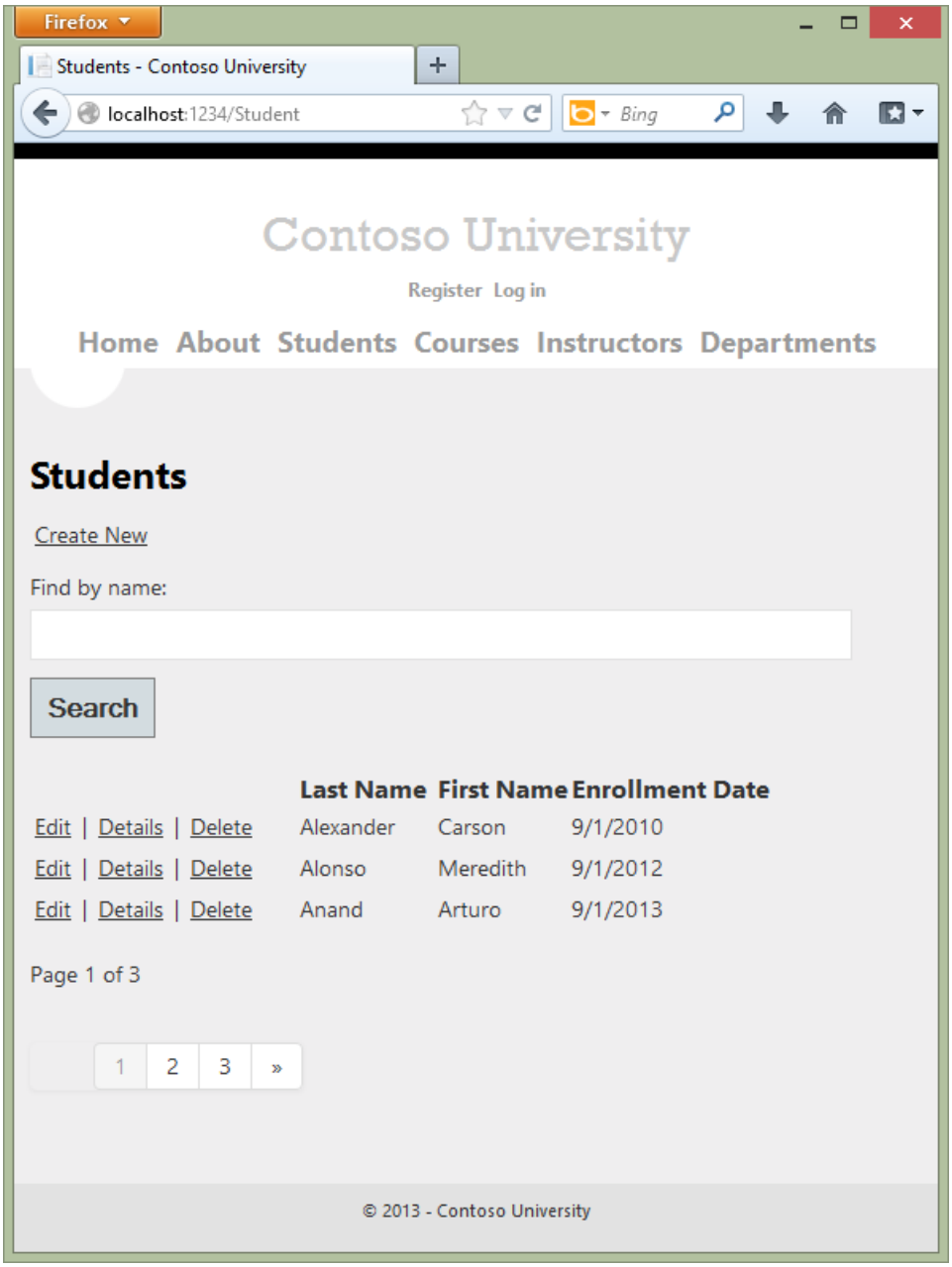
The `ApplyFormatInEditMode` setting specifies that the specified formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you might not want the currency symbol in the text box for editing.)

You can use the DisplayFormat attribute by itself, but it's generally a good idea to use the DataType attribute also. The `DataType` attribute conveys the *semantics* of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.).
- By default, the browser will render data using the correct format based on your locale.
- The DataType attribute can enable MVC to choose the right field template to render the data (the DisplayFormat if used by itself uses the string template). For more information, see Brad Wilson's ASP.NET MVC 2 Templates. (Though written for MVC 2, this article still applies to the current version of ASP.NET MVC.)

If you use the `DataType` attribute with a date field, you have to specify the `DisplayFormat` attribute also in order to ensure that the field renders correctly in Chrome browsers. For more information, see this StackOverflow thread.

Run the Student Index page again and notice that times are no longer displayed for the enrollment dates. The same will be true for any view that uses the `Student` model.



The StringLengthAttribute

You can also specify data validation rules and messages using attributes. Suppose you want to ensure that users don't enter more than 50 characters for a name. To add this limitation, add StringLength attributes to the `LastName` and `FirstMidName` properties, as shown in the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int StudentID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The StringLength attribute won't prevent a user from entering white space for a name. You can use the RegularExpression attribute to apply restrictions to the input. For example the following code requires the first character to be upper case and the remaining characters to be alphabetical:

```
[RegularExpression(@"^[A-Z]+[a-zA-Z""'\s-]*$")]
```

The MaxLength attribute provides similar functionality to the StringLength attribute but doesn't provide client side validation.

Run the application and click the **Students** tab. You get the following error:

The model backing the 'SchoolContext' context has changed since the database was created. Consider using Code First Migrations to update the database (https://go.microsoft.com/fwlink/?LinkId=238269).

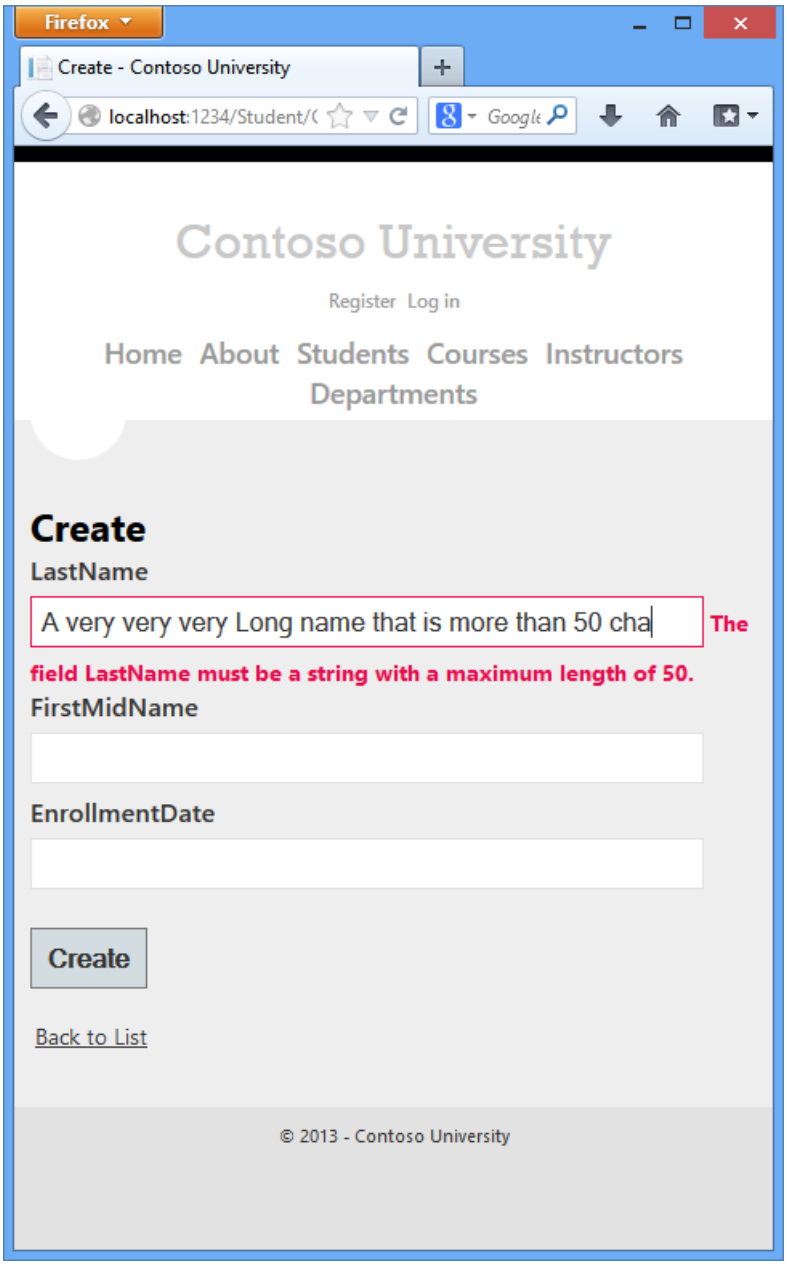
The database model has changed in a way that requires a change in the database schema, and Entity Framework detected that. You'll use migrations to update the schema without losing any data that you added to the database by using the UI. If you changed data that was created by the `Seed` method, that will be changed back to its original state because of the AddOrUpdate method that you're using in the `Seed` method. (AddOrUpdate is equivalent to an "upsert" operation from database terminology.)

In the Package Manager Console (PMC), enter the following commands:

```
add-migration MaxLengthOnNames
update-database
```

The `add-migration MaxLengthOnNames` command creates a file named *<timeStamp>_MaxLengthOnNames.cs*. This file contains code that will update the database to match the current data model. The timestamp prepended to the migrations file name is used by Entity Framework to order the migrations. After you have created multiple migrations, if you drop the database, or if you deploy the project by using Migrations, all of the migrations are applied in the order in which they were created.

Run the **Create** page, and enter either name longer than 50 characters. As soon as you exceed 50 characters, client side validation immediately shows an error message.



The Column Attribute

You can also use attributes to control how your classes and properties are mapped to the database. Suppose you had used the name `FirstMidName` for the first-name field because the field might also contain a middle name. But you want the database column to be named `FirstName`, because users who will be writing ad-hoc queries against the database are accustomed to that name. To make this mapping, you can use the `Column` attribute.

The `Column` attribute specifies that when the database is created, the column of the `Student` table that maps to the `FirstMidName` property will be named `FirstName`. In other words, when your code refers to `Student.FirstMidName`, the data will come from or be updated in the `FirstName` column of the `Student` table. If you don't specify column names, they are given the same name as the property name.

Add a using statement for System.ComponentModel.DataAnnotations.Schema and the column name attribute to the `FirstMidName` property, as shown in the following highlighted code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

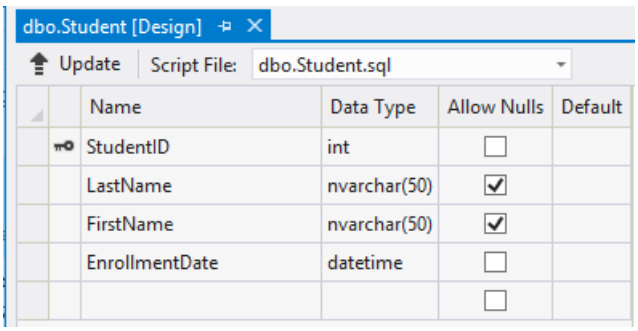
namespace ContosoUniversity.Models
{
    public class Student
    {
        public int StudentID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

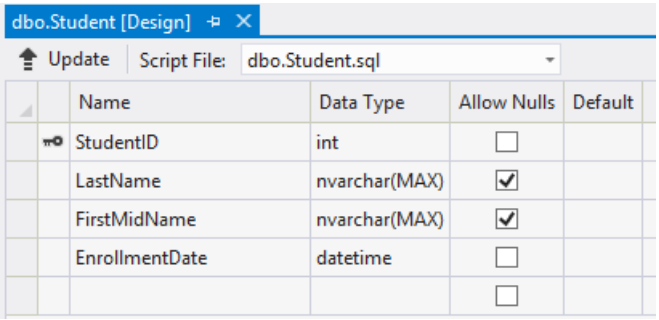
The addition of the Column attribute changes the model backing the SchoolContext, so it won't match the database. Enter the following commands in the PMC to create another migration:

```
add-migration ColumnFirstName
update-database
```

In **Server Explorer** (**Database Explorer** if you are using Express for Web), double-click the *Student* table.



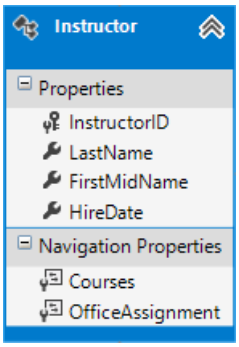
The following image shows the original column name as it was before you applied the first two migrations. In addition to the column name changing from `FirstMidName` to `FirstName`, the two name columns have changed from `MAX` length to 50 characters.



You can also make database mapping changes using the Fluent API, as you'll see later in this tutorial.

> **NOTE**
> If you try to compile before you finish creating all of these entity classes, you might get compiler errors.

Create the Instructor Entity



Create *Models\Instructor.cs*, replacing the template code with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int InstructorID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public virtual ICollection<Course> Courses { get; set; }
        public virtual OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Notice that several properties are the same in the `Student` and `Instructor` entities. In the Implementing Inheritance tutorial later in this series, you'll refactor using inheritance to eliminate this redundancy.

The Required and Display Attributes

The attributes on the `LastName` property specify that it's a required field, that the caption for the text box should

be "Last Name" (instead of the property name, which would be "LastName" with no space), and that the value can't be longer than 50 characters.

```
[Required]
[Display(Name="Last Name")]
[StringLength(50)]
public string LastName { get; set; }
```

The StringLength attribute sets the maximum length in the database and provides client side and server side validation for ASP.NET MVC. You can also specify the minimum string length in this attribute, but the minimum value has no impact on the database schema. The Required attribute is not needed for value types such as DateTime, int, double, and float. Value types cannot be assigned a null value, so they are inherently required. You could remove the Required attribute and replace it with a minimum length parameter for the `StringLength` attribute:

```
[Display(Name = "Last Name")]
[StringLength(50, MinimumLength=1)]
public string LastName { get; set; }
```

You can put multiple attributes on one line, so you could also write the instructor class as follows:

```
public class Instructor
{
   public int InstructorID { get; set; }

   [Display(Name = "Last Name"),StringLength(50, MinimumLength=1)]
   public string LastName { get; set; }

   [Column("FirstName"),Display(Name = "First Name"),StringLength(50, MinimumLength=1)]
   public string FirstMidName { get; set; }

   [DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
   public DateTime HireDate { get; set; }

   public string FullName
   {
      get { return LastName + ", " + FirstMidName; }
   }

   public virtual ICollection<Course> Courses { get; set; }
   public virtual OfficeAssignment OfficeAssignment { get; set; }
}
```

The FullName Calculated Property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. Therefore it has only a `get` accessor, and no `FullName` column will be generated in the database.

```
public string FullName
{
    get { return LastName + ", " + FirstMidName; }
}
```

The Courses and OfficeAssignment Navigation Properties

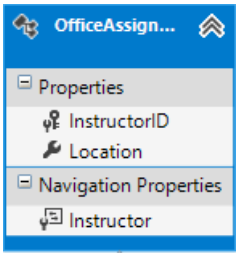
The `Courses` and `OfficeAssignment` properties are navigation properties. As was explained earlier, they are typically defined as virtual so that they can take advantage of an Entity Framework feature called lazy loading. In addition, if a navigation property can hold multiple entities, its type must implement the ICollection<T> Interface.

(For example IList<T> qualifies but not IEnumerable<T> because `IEnumerable<T>` doesn't implement Add.

An instructor can teach any number of courses, so `Courses` is defined as a collection of `Course` entities. Our business rules state an instructor can only have at most one office, so `OfficeAssignment` is defined as a single `OfficeAssignment` entity (which may be `null` if no office is assigned).

```
public virtual ICollection<Course> Courses { get; set; }
public virtual OfficeAssignment OfficeAssignment { get; set; }
```

Create the OfficeAssignment Entity



Create *Models\OfficeAssignment.cs* with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        [ForeignKey("Instructor")]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public virtual Instructor Instructor { get; set; }
    }
}
```

Build the project, which saves your changes and verifies you haven't made any copy and paste errors the compiler can catch.

The Key Attribute

There's a one-to-zero-or-one relationship between the `Instructor` and the `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to, and therefore its primary key is also its foreign key to the `Instructor` entity. But the Entity Framework can't automatically recognize `InstructorID` as the primary key of this entity because its name doesn't follow the `ID` or *classname* `ID` naming convention. Therefore, the `Key` attribute is used to identify it as the key:

```
[Key]
[ForeignKey("Instructor")]
public int InstructorID { get; set; }
```

You can also use the `Key` attribute if the entity does have its own primary key but you want to name the property something different than `classnameID` or `ID`. By default EF treats the key as non-database-generated because the column is for an identifying relationship.

The ForeignKey Attribute

When there is a one-to-zero-or-one relationship or a one-to-one relationship between two entities (such as between `OfficeAssignment` and `Instructor`), EF can't work out which end of the relationship is the principal and which end is dependent. One-to-one relationships have a reference navigation property in each class to the other class. The ForeignKey Attribute can be applied to the dependent class to establish the relationship. If you omit the ForeignKey Attribute, you get the following error when you try to create the migration:

Unable to determine the principal end of an association between the types 'ContosoUniversity.Models.OfficeAssignment' and 'ContosoUniversity.Models.Instructor'. The principal end of this association must be explicitly configured using either the relationship fluent API or data annotations.

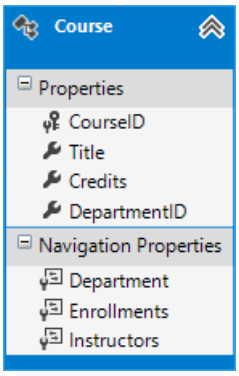
Later in the tutorial we'll show how to configure this relationship with the fluent API.

The Instructor Navigation Property

The `Instructor` entity has a nullable `OfficeAssignment` navigation property (because an instructor might not have an office assignment), and the `OfficeAssignment` entity has a non-nullable `Instructor` navigation property (because an office assignment can't exist without an instructor -- `InstructorID` is non-nullable). When an `Instructor` entity has a related `OfficeAssignment` entity, each entity will have a reference to the other one in its navigation property.

You could put a `[Required]` attribute on the Instructor navigation property to specify that there must be a related instructor, but you don't have to do that because the InstructorID foreign key (which is also the key to this table) is non-nullable.

Modify the Course Entity



In *Models\Course.cs*, replace the code you added earlier with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
   public class Course
   {
      [DatabaseGenerated(DatabaseGeneratedOption.None)]
      [Display(Name = "Number")]
      public int CourseID { get; set; }

      [StringLength(50, MinimumLength = 3)]
      public string Title { get; set; }

      [Range(0, 5)]
      public int Credits { get; set; }

      [Display(Name = "Department")]
      public int DepartmentID { get; set; }

      public virtual Department Department { get; set; }
      public virtual ICollection<Enrollment> Enrollments { get; set; }
      public virtual ICollection<Instructor> Instructors { get; set; }
   }
}
```

The course entity has a foreign key property `DepartmentID` which points to the related `Department` entity and it has a `Department` navigation property. The Entity Framework doesn't require you to add a foreign key property to your data model when you have a navigation property for a related entity. EF automatically creates foreign keys in the database wherever they are needed. But having the foreign key in the data model can make updates simpler and more efficient. For example, when you fetch a course entity to edit, the `Department` entity is null if you don't load it, so when you update the course entity, you would have to first fetch the `Department` entity. When the foreign key property `DepartmentID` is included in the data model, you don't need to fetch the `Department` entity before you update.

The DatabaseGenerated Attribute

The DatabaseGenerated attribute with the None parameter on the `CourseID` property specifies that primary key values are provided by the user rather than generated by the database.

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, the Entity Framework assumes that primary key values are generated by the database. That's what you want in most scenarios. However, for `Course` entities, you'll use a user-specified course number such as a 1000 series for one department, a 2000 series for another department, and so on.

Foreign Key and Navigation Properties

The foreign key properties and navigation properties in the `Course` entity reflect the following relationships:

- A course is assigned to one department, so there's a `DepartmentID` foreign key and a `Department` navigation property for the reasons mentioned above.

  ```
  public int DepartmentID { get; set; }
  public virtual Department Department { get; set; }
  ```

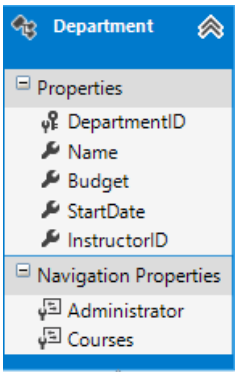
- A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

```
public virtual ICollection<Enrollment> Enrollments { get; set; }
```

- A course may be taught by multiple instructors, so the `Instructors` navigation property is a collection:

```
public virtual ICollection<Instructor> Instructors { get; set; }
```

Creating the Department Entity



Create *Models\Department.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
   public class Department
   {
      public int DepartmentID { get; set; }

      [StringLength(50, MinimumLength=3)]
      public string Name { get; set; }

      [DataType(DataType.Currency)]
      [Column(TypeName = "money")]
      public decimal Budget { get; set; }

      [DataType(DataType.Date)]
      public DateTime StartDate { get; set; }

      [Display(Name = "Administrator")]
      public int? InstructorID { get; set; }

      public virtual Instructor Administrator { get; set; }
      public virtual ICollection<Course> Courses { get; set; }
   }
}
```

The Column Attribute

Earlier you used the Column attribute to change column name mapping. In the code for the `Department` entity, the `Column` attribute is being used to change SQL data type mapping so that the column will be defined using the SQL Server money type in the database:

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required, because the Entity Framework usually chooses the appropriate SQL Server data type based on the CLR type that you define for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. But in this case you know that the column will be holding currency amounts, and the money data type is more appropriate for that.

Foreign Key and Navigation Properties

The foreign key and navigation properties reflect the following relationships:

- A department may or may not have an administrator, and an administrator is always an instructor. Therefore the `InstructorID` property is included as the foreign key to the `Instructor` entity, and a question mark is added after the `int` type designation to mark the property as nullable.The navigation property is named `Administrator` but holds an `Instructor` entity:

  ```
  public int? InstructorID { get; set; }
  public virtual Instructor Administrator { get; set; }
  ```

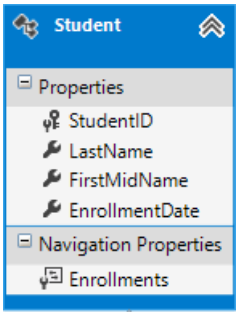
- A department may have many courses, so there's a `Courses` navigation property:

  ```
  public virtual ICollection<Course> Courses { get; set; }
  ```

  > **NOTE**
  > By convention, the Entity Framework enables cascade delete for non-nullable foreign keys and for many-to-many relationships. This can result in circular cascade delete rules, which will cause an exception when your initializer code runs. For example, if you didn't define the `Department.InstructorID` property as nullable, you'd get the following exception message when the initializer runs: "The referential relationship will result in a cyclical reference that's not allowed." If your business rules required `InstructorID` property as non-nullable, you would have to use the following fluent API to disable cascade delete on the relationship:

```
modelBuilder.Entity().HasRequired(d => d.Administrator).WithMany().WillCascadeOnDelete(false);
```

Modifying the Student Entity



In *Models\Student.cs*, replace the code you added earlier with the following code. The changes are highlighted.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
   public class Student
   {
      public int StudentID { get; set; }

      [StringLength(50, MinimumLength = 1)]
      public string LastName { get; set; }

      [StringLength(50, MinimumLength = 1, ErrorMessage = "First name cannot be longer than 50 characters.")]
      [Column("FirstName")]
      public string FirstMidName { get; set; }

      [DataType(DataType.Date)]
      [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
      [Display(Name = "Enrollment Date")]
      public DateTime EnrollmentDate { get; set; }

      public string FullName
      {
         get { return LastName + ", " + FirstMidName; }
      }

      public virtual ICollection<Enrollment> Enrollments { get; set; }
   }
}
```

The Enrollment Entity

In *Models\Enrollment.cs*, replace the code you added earlier with the following code

```
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }

        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public virtual Course Course { get; set; }
        public virtual Student Student { get; set; }
    }
}
```

Foreign Key and Navigation Properties

The foreign key properties and navigation properties reflect the following relationships:

- An enrollment record is for a single course, so there's a `CourseID` foreign key property and a `Course`

navigation property:

```
public int CourseID { get; set; }
public virtual Course Course { get; set; }
```

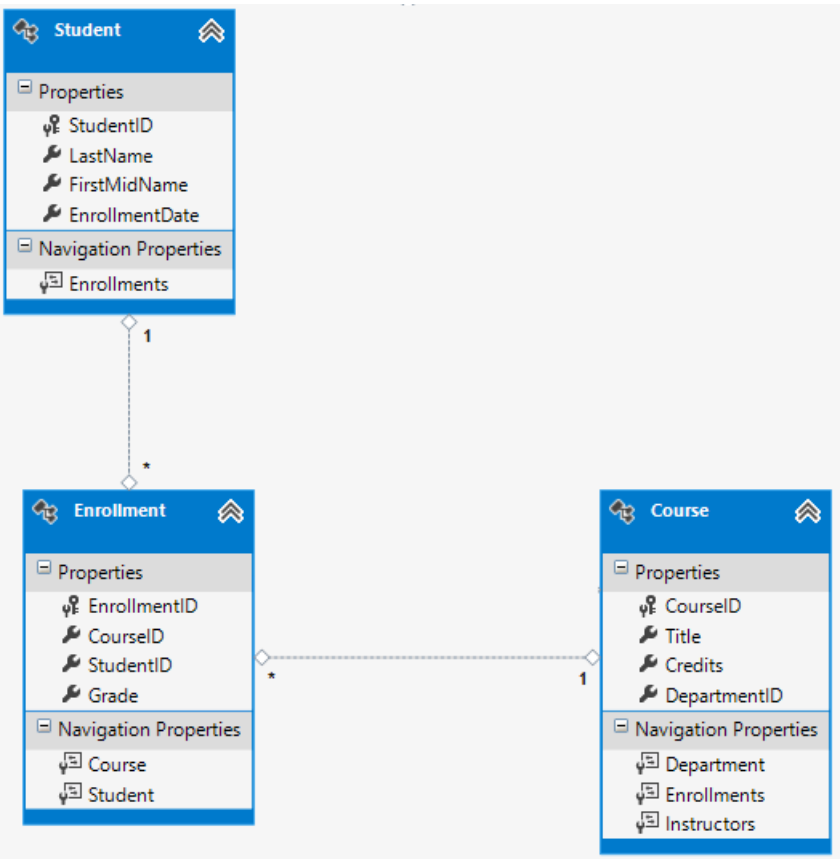
- An enrollment record is for a single student, so there's a `StudentID` foreign key property and a `Student` navigation property:

```
public int StudentID { get; set; }
public virtual Student Student { get; set; }
```

Many-to-Many Relationships

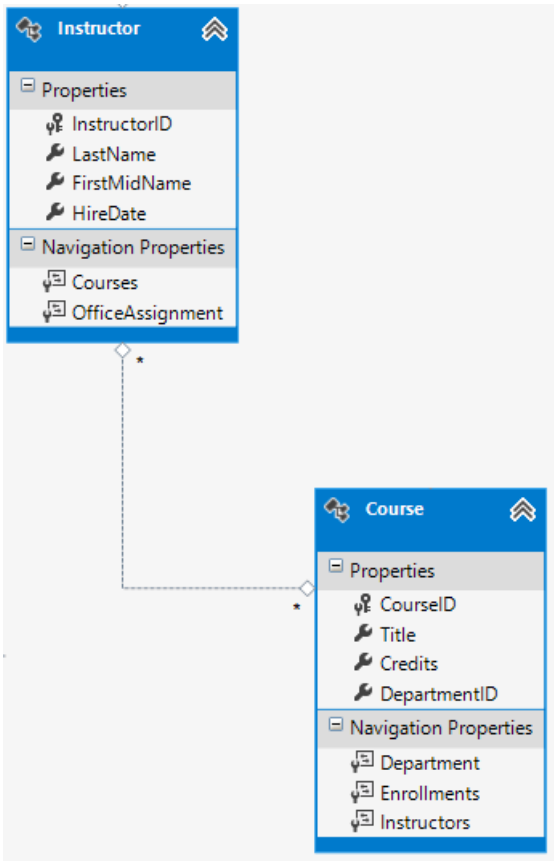
There's a many-to-many relationship between the `Student` and `Course` entities, and the `Enrollment` entity functions as a many-to-many join table *with payload* in the database. This means that the `Enrollment` table contains additional data besides foreign keys for the joined tables (in this case, a primary key and a `Grade` property).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using the Entity Framework Power Tools; creating the diagram isn't part of the tutorial, it's just being used here as an illustration.)

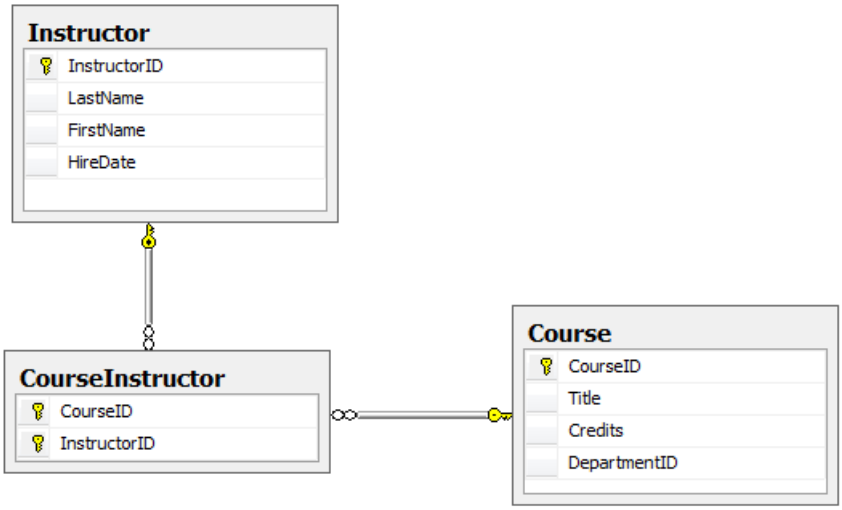


Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two foreign keys `CourseID` and `StudentID`. In that case, it would correspond to a many-to-many join table *without payload* (or a *pure join table*) in the database, and you wouldn't have to create a model class for it at all. The `Instructor` and `Course` entities have that kind of many-to-many relationship, and as you can see, there is no entity class between them:



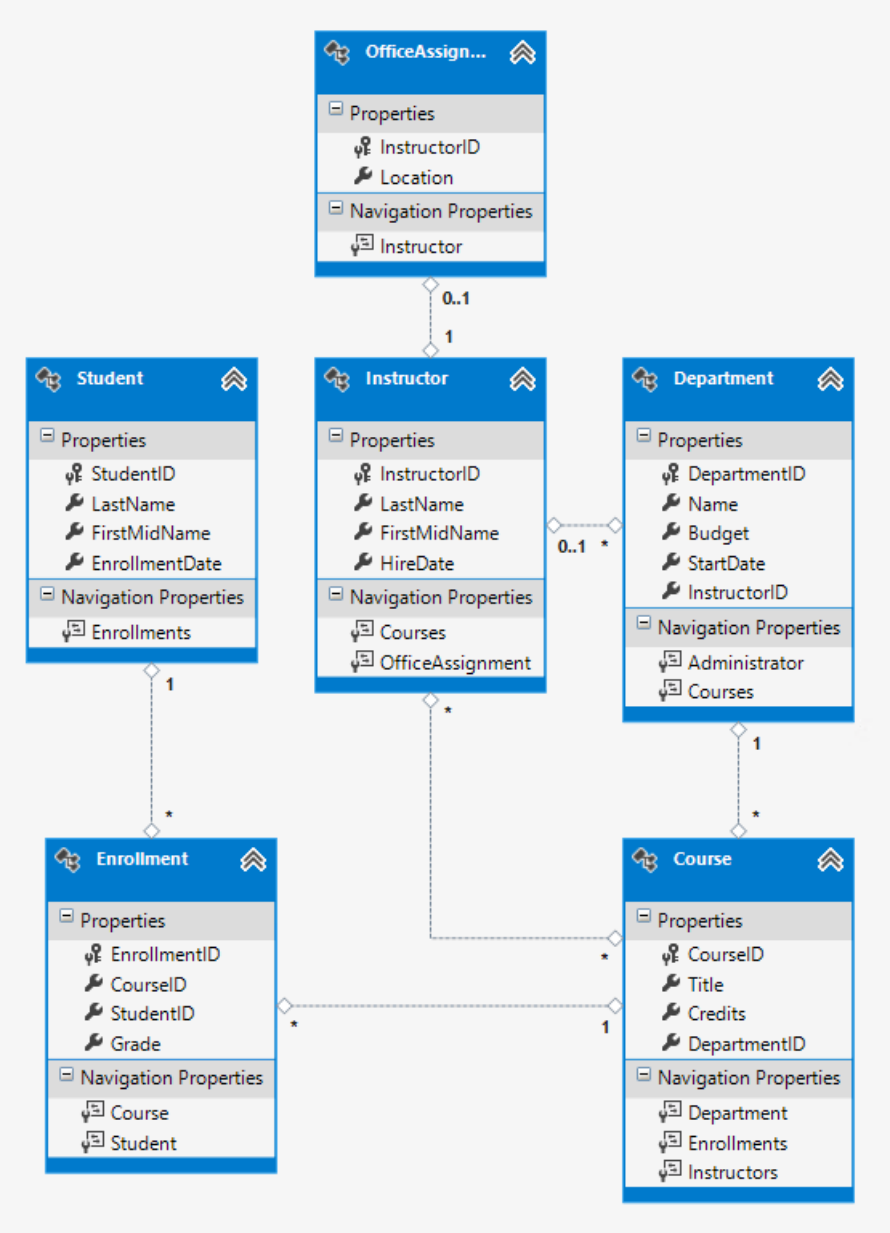
A join table is required in the database, however, as shown in the following database diagram:



The Entity Framework automatically creates the `CourseInstructor` table, and you read and update it indirectly by reading and updating the `Instructor.Courses` and `Course.Instructors` navigation properties.

Entity Diagram Showing Relationships

The following illustration shows the diagram that the Entity Framework Power Tools create for the completed School model.



Besides the many-to-many relationship lines (* to *) and the one-to-many relationship lines (1 to *), you can see here the one-to-zero-or-one relationship line (1 to 0..1) between the `Instructor` and `OfficeAssignment` entities and the zero-or-one-to-many relationship line (0..1 to *) between the Instructor and Department entities.

Customize the Data Model by adding Code to the Database Context

Next you'll add the new entities to the `SchoolContext` class and customize some of the mapping using fluent API calls. (The API is "fluent" because it's often used by stringing a series of method calls together into a single statement.)

In this tutorial you'll use the fluent API only for database mapping that you can't do with attributes. However, you can also use the fluent API to specify most of the formatting, validation, and mapping rules that you can do by using attributes. Some attributes such as `MinimumLength` can't be applied with the fluent API. As mentioned previously, `MinimumLength` doesn't change the schema, it only applies a client and server side validation rule

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." You can mix attributes and fluent API if you want, and there are a few customizations that can only be done by using fluent API, but in general the recommended practice is to choose one of these two approaches and use that consistently as much as possible.

To add the new entities to the data model and perform database mapping that you didn't do by using attributes, replace the code in *DAL\SchoolContext.cs* with the following code:

```
using ContosoUniversity.Models;
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace ContosoUniversity.DAL
{
   public class SchoolContext : DbContext
   {
      public DbSet<Course> Courses { get; set; }
      public DbSet<Department> Departments { get; set; }
      public DbSet<Enrollment> Enrollments { get; set; }
      public DbSet<Instructor> Instructors { get; set; }
      public DbSet<Student> Students { get; set; }
      public DbSet<OfficeAssignment> OfficeAssignments { get; set; }

      protected override void OnModelCreating(DbModelBuilder modelBuilder)
      {
         modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

         modelBuilder.Entity<Course>()
             .HasMany(c => c.Instructors).WithMany(i => i.Courses)
             .Map(t => t.MapLeftKey("CourseID")
                 .MapRightKey("InstructorID")
                 .ToTable("CourseInstructor"));
      }
   }
}
```

The new statement in the OnModelCreating method configures the many-to-many join table:

- For the many-to-many relationship between the `Instructor` and `Course` entities, the code specifies the table and column names for the join table. Code First can configure the many-to-many relationship for you without this code, but if you don't call it, you will get default names such as `InstructorInstructorID` for the `InstructorID` column.

  ```
  modelBuilder.Entity<Course>()
      .HasMany(c => c.Instructors).WithMany(i => i.Courses)
      .Map(t => t.MapLeftKey("CourseID")
          .MapRightKey("InstructorID")
          .ToTable("CourseInstructor"));
  ```

The following code provides an example of how you could have used fluent API instead of attributes to specify the relationship between the `Instructor` and `OfficeAssignment` entities:

```
modelBuilder.Entity<Instructor>()
    .HasOptional(p => p.OfficeAssignment).WithRequired(p => p.Instructor);
```

For information about what "fluent API" statements are doing behind the scenes, see the Fluent API blog post.

Seed the Database with Test Data

Replace the code in the *Migrations\Configuration.cs* file with the following code in order to provide seed data for the new entities you've created.

```
namespace ContosoUniversity.Migrations
{
   using System;
   using System.Collections.Generic;
   using System.Data.Entity;
```

```
using System.Data.Entity.Migrations;
using System.Linq;
using ContosoUniversity.Models;
using ContosoUniversity.DAL;

internal sealed class Configuration : DbMigrationsConfiguration<SchoolContext>
{
   public Configuration()
   {
      AutomaticMigrationsEnabled = false;
   }

   protected override void Seed(SchoolContext context)
   {
      var students = new List<Student>
         {
             new Student { FirstMidName = "Carson",   LastName = "Alexander",
                 EnrollmentDate = DateTime.Parse("2010-09-01") },
             new Student { FirstMidName = "Meredith", LastName = "Alonso",
                 EnrollmentDate = DateTime.Parse("2012-09-01") },
             new Student { FirstMidName = "Arturo",   LastName = "Anand",
                 EnrollmentDate = DateTime.Parse("2013-09-01") },
             new Student { FirstMidName = "Gytis",    LastName = "Barzdukas",
                 EnrollmentDate = DateTime.Parse("2012-09-01") },
             new Student { FirstMidName = "Yan",      LastName = "Li",
                 EnrollmentDate = DateTime.Parse("2012-09-01") },
             new Student { FirstMidName = "Peggy",    LastName = "Justice",
                 EnrollmentDate = DateTime.Parse("2011-09-01") },
             new Student { FirstMidName = "Laura",    LastName = "Norman",
                 EnrollmentDate = DateTime.Parse("2013-09-01") },
             new Student { FirstMidName = "Nino",     LastName = "Olivetto",
                 EnrollmentDate = DateTime.Parse("2005-09-01") }
         };

      students.ForEach(s => context.Students.AddOrUpdate(p => p.LastName, s));
      context.SaveChanges();

      var instructors = new List<Instructor>
         {
             new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie",
                 HireDate = DateTime.Parse("1995-03-11") },
             new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",
                 HireDate = DateTime.Parse("2002-07-06") },
             new Instructor { FirstMidName = "Roger",   LastName = "Harui",
                 HireDate = DateTime.Parse("1998-07-01") },
             new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
                 HireDate = DateTime.Parse("2001-01-15") },
             new Instructor { FirstMidName = "Roger",   LastName = "Zheng",
                 HireDate = DateTime.Parse("2004-02-12") }
         };
      instructors.ForEach(s => context.Instructors.AddOrUpdate(p => p.LastName, s));
      context.SaveChanges();

      var departments = new List<Department>
         {
             new Department { Name = "English",     Budget = 350000,
                 StartDate = DateTime.Parse("2007-09-01"),
                 InstructorID  = instructors.Single( i => i.LastName == "Abercrombie").InstructorID },
             new Department { Name = "Mathematics", Budget = 100000,
                 StartDate = DateTime.Parse("2007-09-01"),
                 InstructorID  = instructors.Single( i => i.LastName == "Fakhouri").InstructorID },
             new Department { Name = "Engineering", Budget = 350000,
                 StartDate = DateTime.Parse("2007-09-01"),
                 InstructorID  = instructors.Single( i => i.LastName == "Harui").InstructorID },
             new Department { Name = "Economics",   Budget = 100000,
                 StartDate = DateTime.Parse("2007-09-01"),
                 InstructorID  = instructors.Single( i => i.LastName == "Kapoor").InstructorID }
         };
      departments.ForEach(s => context.Departments.AddOrUpdate(p => p.Name, s));
```

```
            context.SaveChanges();

            var courses = new List<Course>
               {
                   new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
                     DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID,
                     Instructors = new List<Instructor>()
                   },
                   new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
                     DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID,
                     Instructors = new List<Instructor>()
                   },
                   new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
                     DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID,
                     Instructors = new List<Instructor>()
                   },
                   new Course {CourseID = 1045, Title = "Calculus",       Credits = 4,
                     DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID,
                     Instructors = new List<Instructor>()
                   },
                   new Course {CourseID = 3141, Title = "Trigonometry",   Credits = 4,
                     DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID,
                     Instructors = new List<Instructor>()
                   },
                   new Course {CourseID = 2021, Title = "Composition",    Credits = 3,
                     DepartmentID = departments.Single( s => s.Name == "English").DepartmentID,
                     Instructors = new List<Instructor>()
                   },
                   new Course {CourseID = 2042, Title = "Literature",     Credits = 4,
                     DepartmentID = departments.Single( s => s.Name == "English").DepartmentID,
                     Instructors = new List<Instructor>()
                   },
               };
            courses.ForEach(s => context.Courses.AddOrUpdate(p => p.CourseID, s));
            context.SaveChanges();

            var officeAssignments = new List<OfficeAssignment>
               {
                   new OfficeAssignment {
                       InstructorID = instructors.Single( i => i.LastName == "Fakhouri").InstructorID,
                       Location = "Smith 17" },
                   new OfficeAssignment {
                       InstructorID = instructors.Single( i => i.LastName == "Harui").InstructorID,
                       Location = "Gowan 27" },
                   new OfficeAssignment {
                       InstructorID = instructors.Single( i => i.LastName == "Kapoor").InstructorID,
                       Location = "Thompson 304" },
               };
            officeAssignments.ForEach(s => context.OfficeAssignments.AddOrUpdate(p => p.Location, s));
            context.SaveChanges();

            AddOrUpdateInstructor(context, "Chemistry", "Kapoor");
            AddOrUpdateInstructor(context, "Chemistry", "Harui");
            AddOrUpdateInstructor(context, "Microeconomics", "Zheng");
            AddOrUpdateInstructor(context, "Macroeconomics", "Zheng");

            AddOrUpdateInstructor(context, "Calculus", "Fakhouri");
            AddOrUpdateInstructor(context, "Trigonometry", "Harui");
            AddOrUpdateInstructor(context, "Composition", "Abercrombie");
            AddOrUpdateInstructor(context, "Literature", "Abercrombie");

            context.SaveChanges();

            var enrollments = new List<Enrollment>
               {
                   new Enrollment {
                       StudentID = students.Single(s => s.LastName == "Alexander").StudentID,
                       CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
                       Grade = Grade.A
```

```
            },
             new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alexander").StudentID,
                CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
                Grade = Grade.C
             },
             new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alexander").StudentID,
                CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
                Grade = Grade.B
             },
             new Enrollment {
                 StudentID = students.Single(s => s.LastName == "Alonso").StudentID,
                CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
                Grade = Grade.B
             },
             new Enrollment {
                 StudentID = students.Single(s => s.LastName == "Alonso").StudentID,
                CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
                Grade = Grade.B
             },
             new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").StudentID,
                CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
                Grade = Grade.B
             },
             new Enrollment {
                StudentID = students.Single(s => s.LastName == "Anand").StudentID,
                CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
             },
             new Enrollment {
                StudentID = students.Single(s => s.LastName == "Anand").StudentID,
                CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
                Grade = Grade.B
             },
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Barzdukas").StudentID,
                CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
                Grade = Grade.B
             },
             new Enrollment {
                StudentID = students.Single(s => s.LastName == "Li").StudentID,
                CourseID = courses.Single(c => c.Title == "Composition").CourseID,
                Grade = Grade.B
             },
             new Enrollment {
                StudentID = students.Single(s => s.LastName == "Justice").StudentID,
                CourseID = courses.Single(c => c.Title == "Literature").CourseID,
                Grade = Grade.B
             }
       };

    foreach (Enrollment e in enrollments)
    {
       var enrollmentInDataBase = context.Enrollments.Where(
           s =>
                s.Student.StudentID == e.StudentID &&
                s.Course.CourseID == e.CourseID).SingleOrDefault();
       if (enrollmentInDataBase == null)
       {
          context.Enrollments.Add(e);
       }
    }
    context.SaveChanges();
 }

 void AddOrUpdateInstructor(SchoolContext context, string courseTitle, string instructorName)
 {
    var crs = context.Courses.SingleOrDefault(c => c.Title == courseTitle);
```

```
            var inst = crs.Instructors.SingleOrDefault(i => i.LastName == instructorName);
            if (inst == null)
               crs.Instructors.Add(context.Instructors.Single(i => i.LastName == instructorName));
         }
      }
   }
```

As you saw in the first tutorial, most of this code simply updates or creates new entity objects and loads sample data into properties as required for testing. However, notice how the `Course` entity, which has a many-to-many relationship with the `Instructor` entity, is handled:

```
var courses = new List<Course>
{
     new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
       Department = departments.Single( s => s.Name == "Engineering"),
       Instructors = new List<Instructor>()
     },
     ...
};
courses.ForEach(s => context.Courses.AddOrUpdate(p => p.CourseID, s));
context.SaveChanges();
```

When you create a `Course` object, you initialize the `Instructors` navigation property as an empty collection using the code `Instructors = new List<Instructor>()`. This makes it possible to add `Instructor` entities that are related to this `Course` by using the `Instructors.Add` method. If you didn't create an empty list, you wouldn't be able to add these relationships, because the `Instructors` property would be null and wouldn't have an `Add` method. You could also add the list initialization to the constructor.

Add a Migration and Update the Database

From the PMC, enter the `add-migration` command:

```
PM> add-Migration Chap4
```

If you try to update the database at this point, you'll get the following error:

The ALTER TABLE statement conflicted with the FOREIGN KEY constraint "FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.

Edit the *<timestamp>_Chap4.cs* file, and make the following code changes (you'll add a SQL statement and modify an `AddColumn` statement):

```
CreateTable(
        "dbo.CourseInstructor",
        c => new
            {
                CourseID = c.Int(nullable: false),
                InstructorID = c.Int(nullable: false),
            })
        .PrimaryKey(t => new { t.CourseID, t.InstructorID })
        .ForeignKey("dbo.Course", t => t.CourseID, cascadeDelete: true)
        .ForeignKey("dbo.Instructor", t => t.InstructorID, cascadeDelete: true)
        .Index(t => t.CourseID)
        .Index(t => t.InstructorID);

    // Create  a department for course to point to.
    Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00, GETDATE())");
    //  default value for FK points to department created above.
    AddColumn("dbo.Course", "DepartmentID", c => c.Int(nullable: false, defaultValue: 1));
    //AddColumn("dbo.Course", "DepartmentID", c => c.Int(nullable: false));

    AlterColumn("dbo.Course", "Title", c => c.String(maxLength: 50));
    AddForeignKey("dbo.Course", "DepartmentID", "dbo.Department", "DepartmentID", cascadeDelete: true);
    CreateIndex("dbo.Course", "DepartmentID");
}

public override void Down()
{
```

(Make sure that you comment out or delete the existing `AddColumn` line when you add the new one, or you'll get an error when you enter the `update-database` command.)

Sometimes when you execute migrations with existing data, you need to insert stub data into the database to satisfy foreign key constraints, and that's what you're doing now. The generated code adds a non-nullable `DepartmentID` foreign key to the `Course` table. If there are already rows in the `Course` table when the code runs, the `AddColumn` operation would fail because SQL Server doesn't know what value to put in the column that can't be null. Therefore you've changed the code to give the new column a default value, and you've created a stub department named "Temp" to act as the default department. As a result, if there are existing `Course` rows when this code runs, they will all be related to the "Temp" department.

When the `Seed` method runs, it will insert rows in the `Department` table and it will relate existing `Course` rows to those new `Department` rows. If you haven't added any courses in the UI, you would then no longer need the "Temp" department or the default value on the `Course.DepartmentID` column. To allow for the possibility that someone might have added courses by using the application, you'd also want to update the `Seed` method code to ensure that all `Course` rows (not just the ones inserted by earlier runs of the `Seed` method) have valid `DepartmentID` values before you remove the default value from the column and delete the "Temp" department.

After you have finished editing the *<timestamp>_Chap4.cs* file, enter the `update-database` command in the PMC to execute the migration.

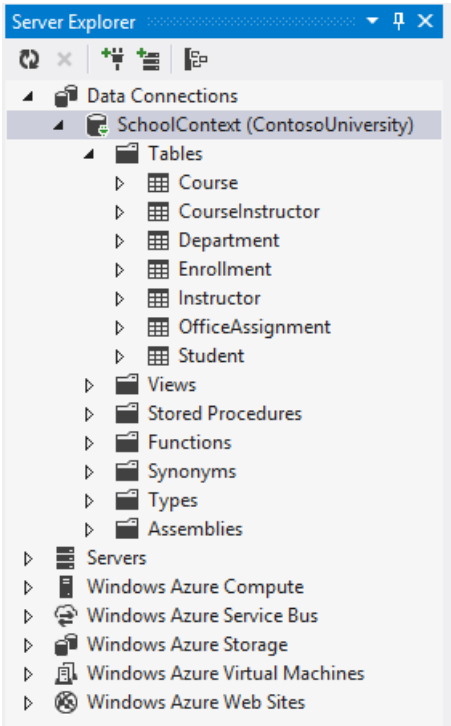
NOTE

It's possible to get other errors when migrating data and making schema changes. If you get migration errors you can't resolve, you can either change the connection string in the *Web.config* file or delete the database. The simplest approach is to rename the database in *Web.config* file. For example, change the database name to CU_test as shown in the following:

```
<add name="SchoolContext" connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=CU_Test;
      Integrated Security=SSPI;AttachDBFilename=|DataDirectory|\CU_Test.mdf"
      providerName="System.Data.SqlClient" />
```

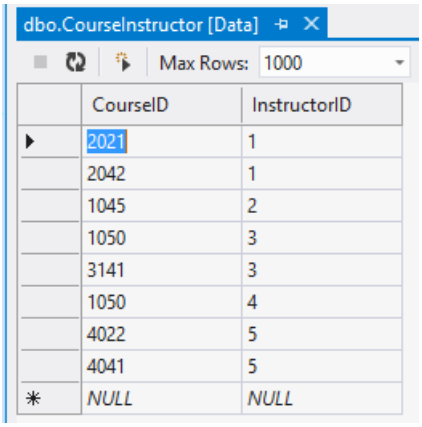
With a new database, there is no data to migrate, and the `update-database` command is much more likely to complete without errors. For instructions on how to delete the database, see How to Drop a Database from Visual Studio 2012.

Open the database in **Server Explorer** as you did earlier, and expand the **Tables** node to see that all of the tables have been created. (If you still have **Server Explorer** open from the earlier time, click the **Refresh** button.)



You didn't create a model class for the `CourseInstructor` table. As explained earlier, this is a join table for the many-to-many relationship between the `Instructor` and `Course` entities.

Right-click the `CourseInstructor` table and select **Show Table Data** to verify that it has data in it as a result of the `Instructor` entities you added to the `Course.Instructors` navigation property.



CourseID	InstructorID
2021	1
2042	1
1045	2
1050	3
3141	3
1050	4
4022	5
4041	5
NULL	NULL

Summary

You now have a more complex data model and corresponding database. In the following tutorial you'll learn more about different ways to access related data.

Links to other Entity Framework resources can be found in the ASP.NET Data Access Content Map.

PREVIOUS NEXT

Reading Related Data with the Entity Framework in an ASP.NET MVC Application (5 of 10)

2/20/2020 • 17 minutes to read • Edit Online

by Tom Dykstra

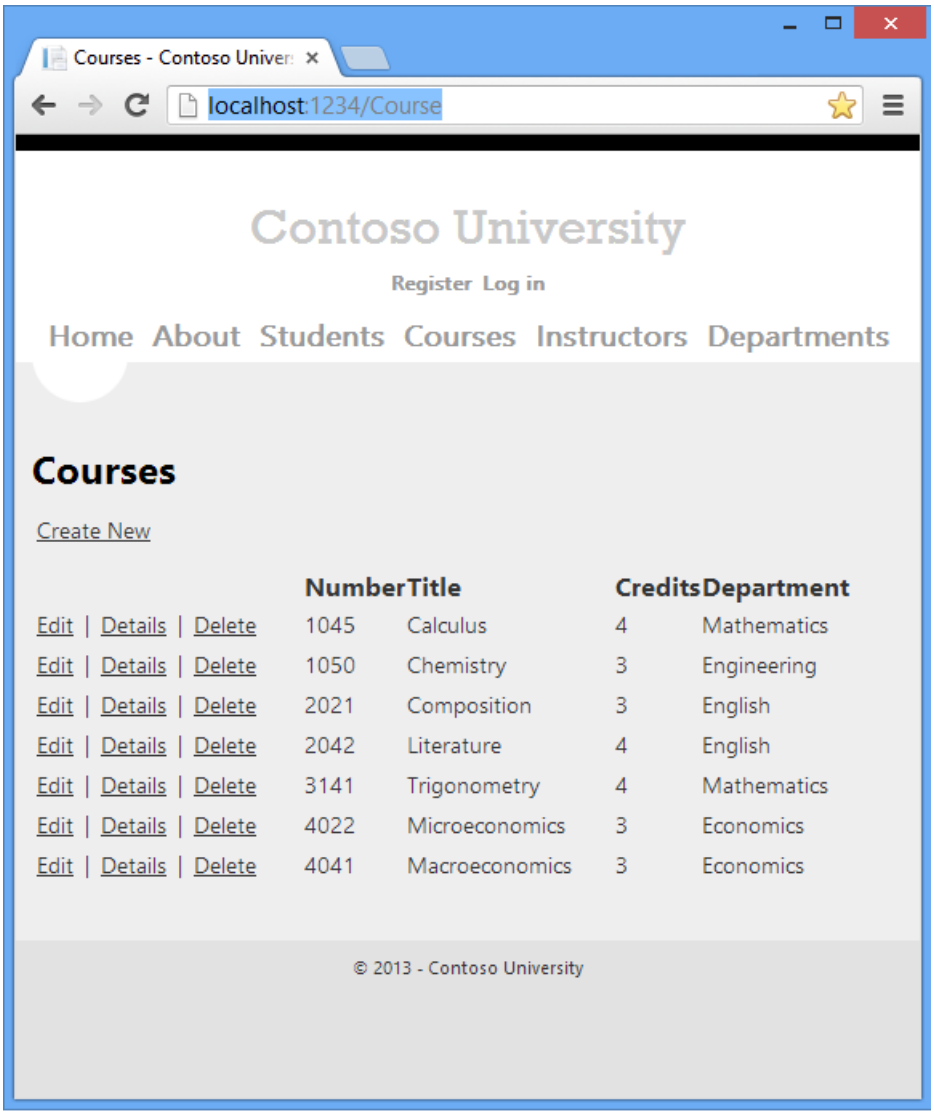
Download Completed Project

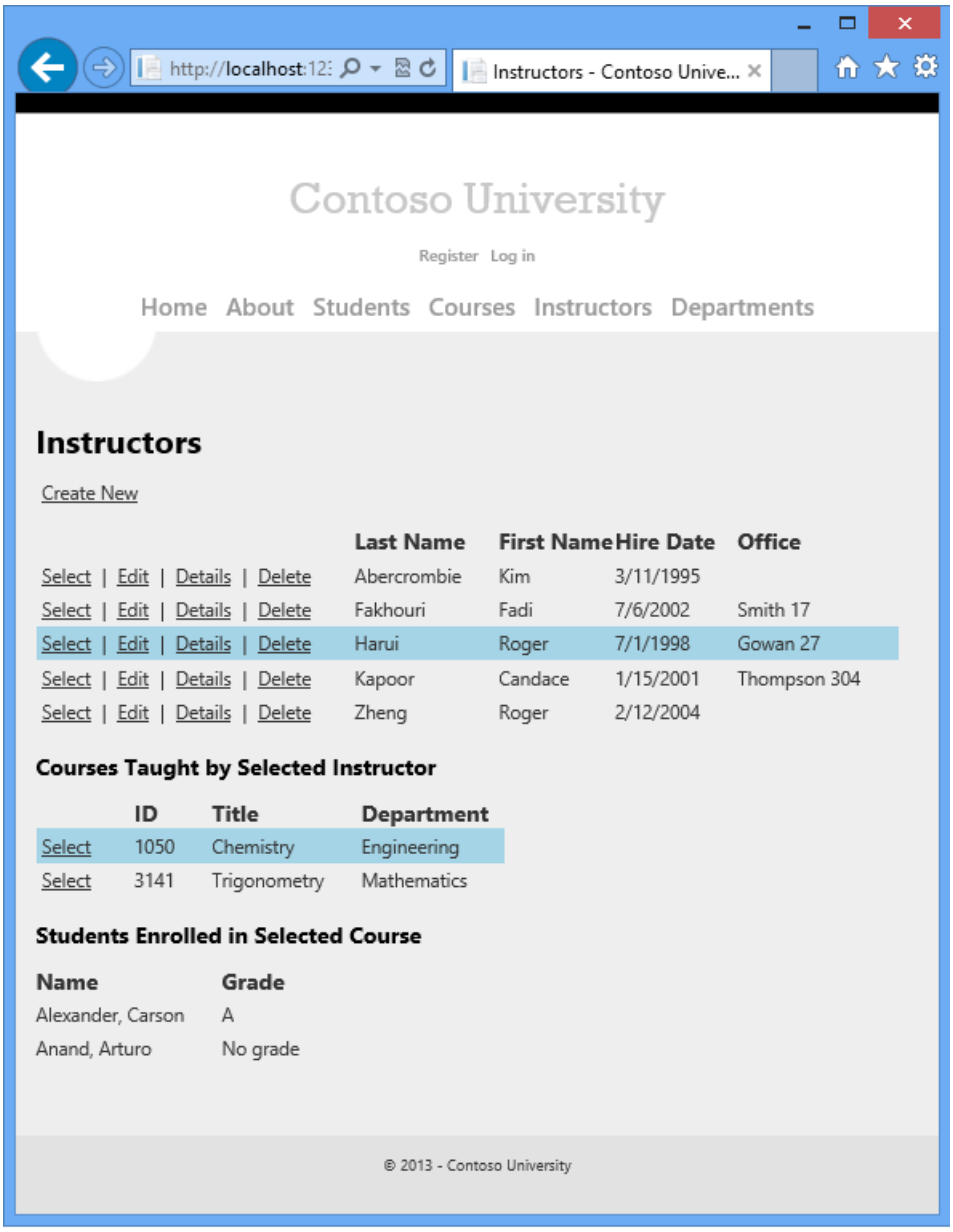
The Contoso University sample web application demonstrates how to create ASP.NET MVC 4 applications using the Entity Framework 5 Code First and Visual Studio 2012. For information about the tutorial series, see the first tutorial in the series. You can start the tutorial series from the beginning or download a starter project for this chapter and start here.

> **NOTE**
>
> If you run into a problem you can't resolve, download the completed chapter and try to reproduce your problem. You can generally find the solution to the problem by comparing your code to the completed code. For some common errors and how to solve them, see Errors and Workarounds.

In the previous tutorial you completed the School data model. In this tutorial you'll read and display related data — that is, data that the Entity Framework loads into navigation properties.

The following illustrations show the pages that you'll work with.


Courses - Contoso Univer
localhost:1234/Course
Contoso University
Register Log in
Home About Students Courses Instructors Departments
Courses
Create New
Number Title Credits Department
Edit | Details | Delete 1045 Calculus 4 Mathematics
Edit | Details | Delete 1050 Chemistry 3 Engineering
Edit | Details | Delete 2021 Composition 3 English
Edit | Details | Delete 2042 Literature 4 English
Edit | Details | Delete 3141 Trigonometry 4 Mathematics
Edit | Details | Delete 4022 Microeconomics 3 Economics
Edit | Details | Delete 4041 Macroeconomics 3 Economics
© 2013 - Contoso University



Lazy, Eager, and Explicit Loading of Related Data

There are several ways that the Entity Framework can load related data into the navigation properties of an entity:

- *Lazy loading*. When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. This results in multiple queries sent to the database — one for the entity itself and one each time that related data for the entity must be retrieved.

```
departments = context.Departments
foreach (Department d in departments)
{
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- *Eager loading*. When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading by using the `Include` method.

```
departments = context.Departments.Include(x => x.Courses)
foreach (Department d in departments)
{
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows and related Course rows

- *Explicit loading*. This is similar to lazy loading, except that you explicitly retrieve the related data in code; it doesn't happen automatically when you access a navigation property. You load related data manually by getting the object state manager entry for an entity and calling the `Collection.Load` method for collections or the `Reference.Load` method for properties that hold a single entity. (In the following example, if you wanted to load the Administrator navigation property, you'd replace `Collection(x => x.Courses)` with `Reference(x => x.Administrator)`.)

```
departments = context.Departments.ToList();
foreach (Department d in departments)
{
    context.Entry(d).Collection(x => x.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

Because they don't immediately retrieve the property values, lazy loading and explicit loading are also both known as *deferred loading*.

In general, if you know you need related data for every entity retrieved, eager loading offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, in the above examples, suppose that each department has ten related courses. The eager loading example would result in just a single (join) query and a single round trip to the database. The lazy loading and explicit loading examples would both result in eleven queries and eleven round trips to the database. The extra round trips to the database are especially detrimental to performance when latency is high.

On the other hand, in some scenarios lazy loading is more efficient. Eager loading might cause a very complex join to be generated, which SQL Server can't process efficiently. Or if you need to access an entity's navigation properties only for a subset of a set of entities you're processing, lazy loading might perform better because eager loading would retrieve more data than you need. If performance is critical, it's best to test performance both ways in order to make the best choice.

Typically you'd use explicit loading only when you've turned lazy loading off. One scenario when you should turn lazy loading off is during serialization. Lazy loading and serialization don't mix well, and if you aren't careful you can end up querying significantly more data than you intended when lazy loading is enabled. Serialization generally works by accessing each property on an instance of a type. Property access triggers lazy loading, and those lazy loaded entities are serialized. The serialization process then accesses each property of the lazy-loaded entities, potentially causing even more lazy loading and serialization. To prevent this run-away chain reaction, turn lazy loading off before you serialize an entity.

The database context class performs lazy loading by default. There are two ways to disable lazy loading:

- For specific navigation properties, omit the `virtual` keyword when you declare the property.
- For all navigation properties, set `LazyLoadingEnabled` to `false`. For example, you can put the following code in the constructor of your context class:

```
this.Configuration.LazyLoadingEnabled = false;
```

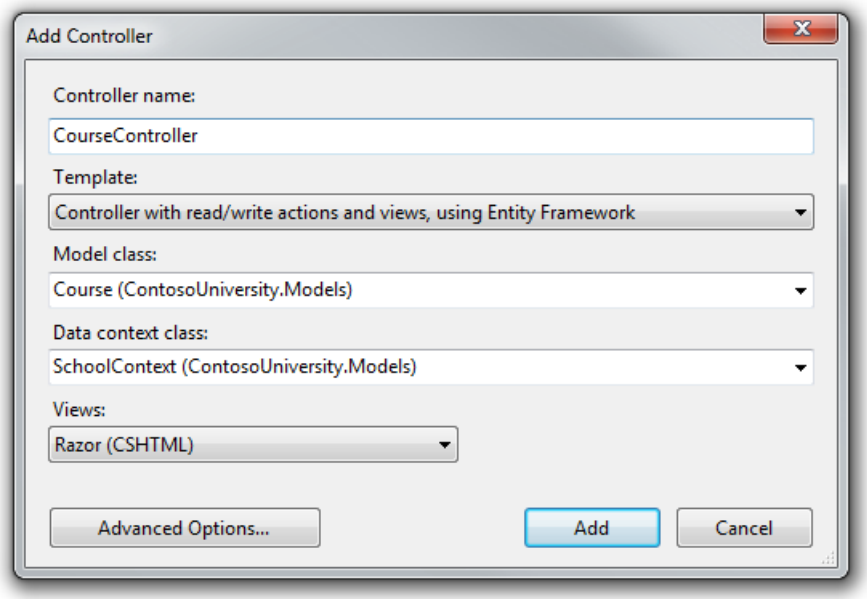
Lazy loading can mask code that causes performance problems. For example, code that doesn't specify eager or

explicit loading but processes a high volume of entities and uses several navigation properties in each iteration might be very inefficient (because of many round trips to the database). An application that performs well in development using an on premise SQL server might have performance problems when moved to Azure SQL Database due to the increased latency and lazy loading. Profiling the database queries with a realistic test load will help you determine if lazy loading is appropriate. For more information see Demystifying Entity Framework Strategies: Loading Related Data and Using the Entity Framework to Reduce Network Latency to SQL Azure.

Create a Courses Index Page That Displays Department Name

The `Course` entity includes a navigation property that contains the `Department` entity of the department that the course is assigned to. To display the name of the assigned department in a list of courses, you need to get the `Name` property from the `Department` entity that is in the `Course.Department` navigation property.

Create a controller named `CourseController` for the `Course` entity type, using the same options that you did earlier for the `Student` controller, as shown in the following illustration (except unlike the image, your context class is in the DAL namespace, not the Models namespace):



Open *Controllers\CourseController.cs* and look at the `Index` method:

```
public ViewResult Index()
{
    var courses = db.Courses.Include(c => c.Department);
    return View(courses.ToList());
}
```

The automatic scaffolding has specified eager loading for the `Department` navigation property by using the `Include` method.

Open *Views\Course\Index.cshtml* and replace the existing code with the following code. The changes are highlighted:

```
@model IEnumerable<ContosoUniversity.Models.Course>

@{
    ViewBag.Title = "Courses";
}

<h2>Courses</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th></th>
        <th>Number</th>
        <th>Title</th>
        <th>Credits</th>
        <th>Department</th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.CourseID }) |
            @Html.ActionLink("Details", "Details", new { id=item.CourseID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.CourseID })
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.CourseID)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Credits)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Department.Name)
        </td>
    </tr>
}
</table>
```

You've made the following changes to the scaffolded code:

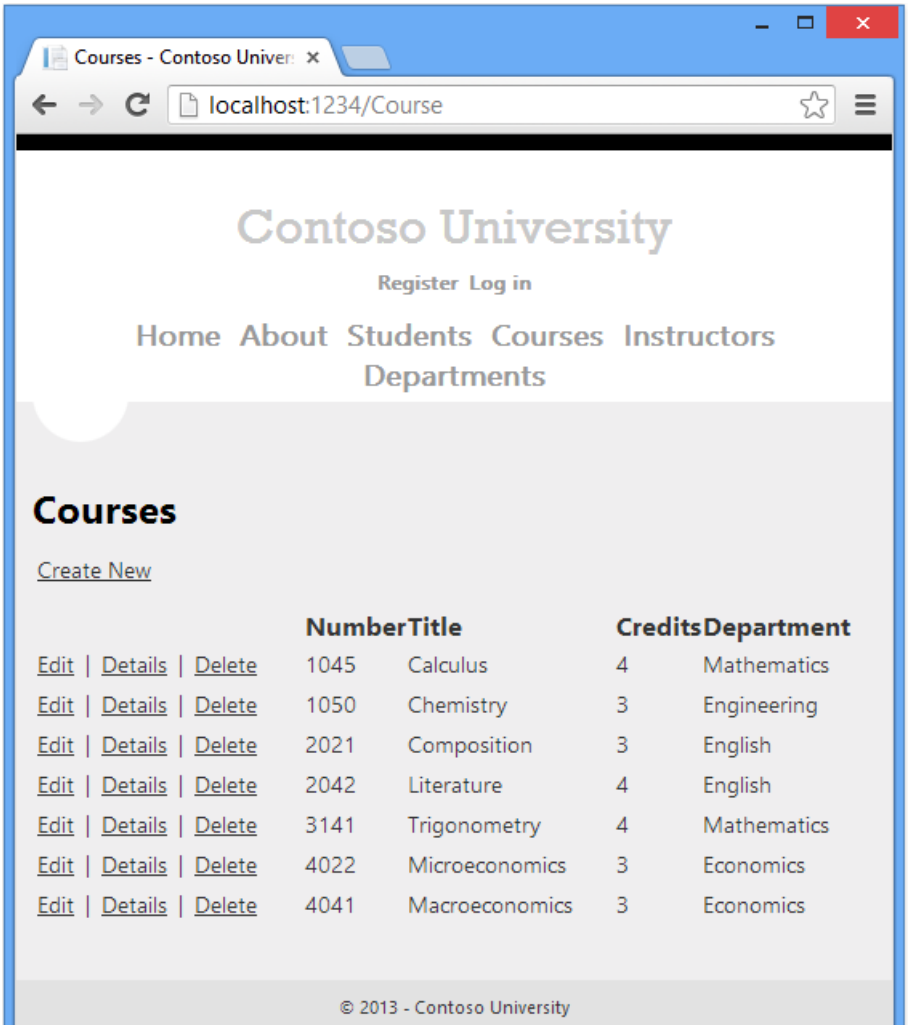
- Changed the heading from **Index** to **Courses**.
- Moved the row links to the left.
- Added a column under the heading **Number** that shows the `CourseID` property value. (By default, primary keys aren't scaffolded because normally they are meaningless to end users. However, in this case the primary key is meaningful and you want to show it.)
- Changed the last column heading from **DepartmentID** (the name of the foreign key to the `Department` entity) to **Department**.

Notice that for the last column, the scaffolded code displays the `Name` property of the `Department` entity that's loaded into the `Department` navigation property:

```
<td>
    @Html.DisplayFor(modelItem => item.Department.Name)
</td>
```

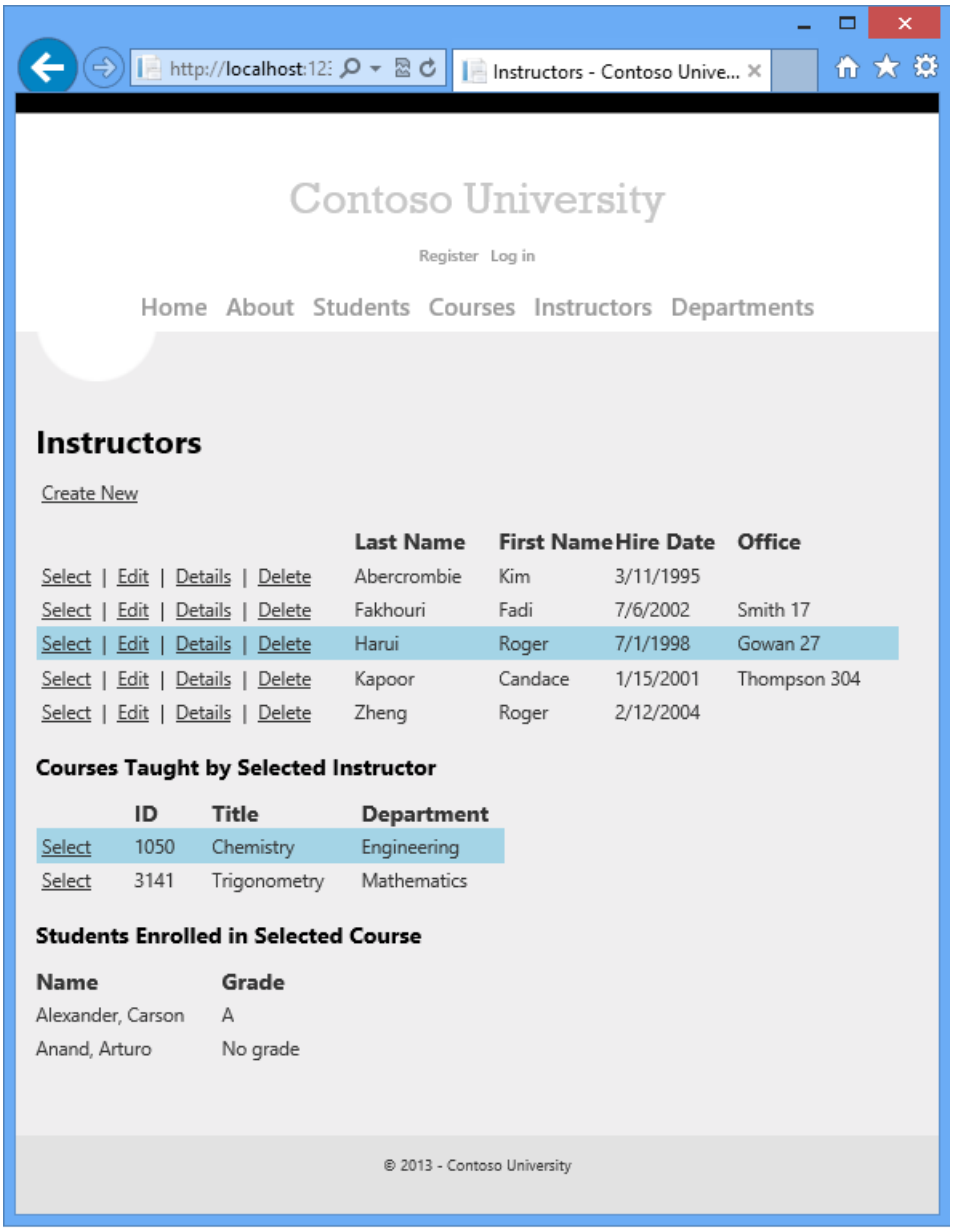
Run the page (select the **Courses** tab on the Contoso University home page) to see the list with department

names.



Create an Instructors Index Page That Shows Courses and Enrollments

In this section you'll create a controller and view for the `Instructor` entity in order to display the Instructors Index page:



This page reads and displays related data in the following ways:

- The list of instructors displays related data from the `OfficeAssignment` entity. The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. You'll use eager loading for the `OfficeAssignment` entities. As explained earlier, eager loading is typically more efficient when you need the related data for all retrieved rows of the primary table. In this case, you want to display office assignments for all displayed instructors.
- When the user selects an instructor, related `Course` entities are displayed. The `Instructor` and `Course` entities are in a many-to-many relationship. You'll use eager loading for the `Course` entities and their related `Department` entities. In this case, lazy loading might be more efficient because you need courses only for the selected instructor. However, this example shows how to use eager loading for navigation properties within entities that are themselves in navigation properties.
- When the user selects a course, related data from the `Enrollments` entity set is displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship. You'll add explicit loading for `Enrollment` entities and their related `Student` entities. (Explicit loading isn't necessary because lazy loading is enabled, but this shows how to do explicit loading.)

Create a View Model for the Instructor Index View

The Instructor Index page shows three different tables. Therefore, you'll create a view model that includes three properties, each holding the data for one of the tables.

In the *ViewModels* folder, create *InstructorIndexData.cs* and replace the existing code with the following code:

```
using System.Collections.Generic;
using ContosoUniversity.Models;

namespace ContosoUniversity.ViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

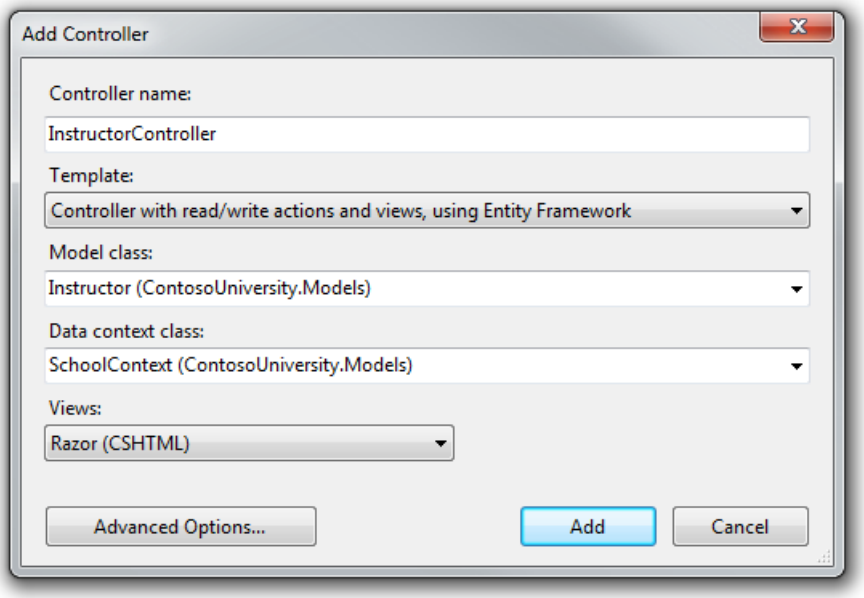
Adding a Style for Selected Rows

To mark selected rows you need a different background color. To provide a style for this UI, add the following highlighted code to the section `/* info and errors */` in *Content\Site.css*, as shown below:

```
/* info and errors */
.selectedrow
{
    background-color: #a4d4e6;
}
.message-info {
    border: 1px solid;
    clear: both;
    padding: 10px 20px;
}
```

Creating the Instructor Controller and Views

Create an `InstructorController` controller as shown in the following illustration:



Open *Controllers\InstructorController.cs* and add a `using` statement for the `ViewModels` namespace:

```
using ContosoUniversity.ViewModels;
```

The scaffolded code in the `Index` method specifies eager loading only for the `OfficeAssignment` navigation property:

```
public ViewResult Index()
{
    var instructors = db.Instructors.Include(i => i.OfficeAssignment);
    return View(instructors.ToList());
}
```

Replace the `Index` method with the following code to load additional related data and put it in the view model:

```
public ActionResult Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.Courses.Select(c => c.Department))
        .OrderBy(i => i.LastName);

    if (id != null)
    {
        ViewBag.InstructorID = id.Value;
        viewModel.Courses = viewModel.Instructors.Where(
            i => i.InstructorID == id.Value).Single().Courses;
    }

    if (courseID != null)
    {
        ViewBag.CourseID = courseID.Value;
        viewModel.Enrollments = viewModel.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }

    return View(viewModel);
}
```

The method accepts optional route data (`id`) and a query string parameter (`courseID`) that provide the ID values of the selected instructor and selected course, and passes all of the required data to the view. The parameters are provided by the **Select** hyperlinks on the page.

TIP

Route data

Route data is data that the model binder found in a URL segment specified in the routing table. For example, the default route specifies `controller`, `action`, and `id` segments:

```
routes.MapRoute(
 name: "Default",
 url: "{controller}/{action}/{id}",
 defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

In the following URL, the default route maps `Instructor` as the `controller`, `Index` as the `action` and 1 as the `id`; these are route data values.

`http://localhost:1230/Instructor/Index/1?courseID=2021`

"?courseID=2021" is a query string value. The model binder will also work if you pass the `id` as a query string value:

`http://localhost:1230/Instructor/Index?id=1&CourseID=2021`

The URLs are created by `ActionLink` statements in the Razor view. In the following code, the `id` parameter matches the default route, so `id` is added to the route data.

```
@Html.ActionLink("Select", "Index", new { id = item.PersonID  })
```

In the following code, `courseID` doesn't match a parameter in the default route, so it's added as a query string.

```
@Html.ActionLink("Select", "Index", new { courseID = item.CourseID })
```

The code begins by creating an instance of the view model and putting in it the list of instructors. The code specifies eager loading for the `Instructor.OfficeAssignment` and the `Instructor.Courses` navigation property.

```
var viewModel = new InstructorIndexData();
viewModel.Instructors = db.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.Courses.Select(c => c.Department))
     .OrderBy(i => i.LastName);
```

The second `Include` method loads Courses, and for each Course that is loaded it does eager loading for the `Course.Department` navigation property.

```
.Include(i => i.Courses.Select(c => c.Department))
```

As mentioned previously, eager loading is not required but is done to improve performance. Since the view always requires the `OfficeAssignment` entity, it's more efficient to fetch that in the same query. `Course` entities are required when an instructor is selected in the web page, so eager loading is better than lazy loading only if the page is displayed more often with a course selected than without.

If an instructor ID was selected, the selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is then loaded with the `Course` entities from that instructor's `Courses` navigation property.

```
if (id != null)
{
    ViewBag.InstructorID = id.Value;
    viewModel.Courses = viewModel.Instructors.Where(i => i.InstructorID == id.Value).Single().Courses;
}
```

The `Where` method returns a collection, but in this case the criteria passed to that method result in only a single `Instructor` entity being returned. The `Single` method converts the collection into a single `Instructor` entity, which gives you access to that entity's `Courses` property.

You use the Single method on a collection when you know the collection will have only one item. The `Single` method throws an exception if the collection passed to it is empty or if there's more than one item. An alternative is SingleOrDefault, which returns a default value (`null` in this case) if the collection is empty. However, in this case that would still result in an exception (from trying to find a `Courses` property on a `null` reference), and the exception message would less clearly indicate the cause of the problem. When you call the `Single` method, you can also pass in the `Where` condition instead of calling the `Where` method separately:

```
.Single(i => i.InstructorID == id.Value)
```

Instead of:

```
.Where(I => i.InstructorID == id.Value).Single()
```

Next, if a course was selected, the selected course is retrieved from the list of courses in the view model. Then the view model's `Enrollments` property is loaded with the `Enrollment` entities from that course's `Enrollments` navigation property.

```
if (courseID != null)
{
    ViewBag.CourseID = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

Modifying the Instructor Index View

In *Views\Instructor\Index.cshtml*, replace the existing code with the following code. The changes are highlighted:

```
@model ContosoUniversity.ViewModels.InstructorIndexData

@{
    ViewBag.Title = "Instructors";
}

<h2>Instructors</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th></th>
        <th>Last Name</th>
        <th>First Name</th>
        <th>Hire Date</th>
        <th>Office</th>
    </tr>
    @foreach (var item in Model.Instructors)
    {
        string selectedRow = "";
        if (item.InstructorID == ViewBag.InstructorID)
        {
            selectedRow = "selectedrow";
        }
        <tr class="@selectedRow" valign="top">
            <td>
                @Html.ActionLink("Select", "Index", new { id = item.InstructorID }) |
                @Html.ActionLink("Edit", "Edit", new { id = item.InstructorID }) |
                @Html.ActionLink("Details", "Details", new { id = item.InstructorID }) |
                @Html.ActionLink("Delete", "Delete", new { id = item.InstructorID })
            </td>
            <td>
                @item.LastName
            </td>
            <td>
                @item.FirstMidName
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.HireDate)
            </td>
            <td>
                @if (item.OfficeAssignment != null)
                {
                    @item.OfficeAssignment.Location
                }
            </td>
        </tr>
    }
</table>
```

You've made the following changes to the existing code:

- Changed the model class to `InstructorIndexData`.
- Changed the page title from **Index** to **Instructors**.
- Moved the row link columns to the left.
- Removed the **FullName** column.
- Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` is not null. (Because this is a one-to-zero-or-one relationship, there might not be a related `OfficeAssignment` entity.)

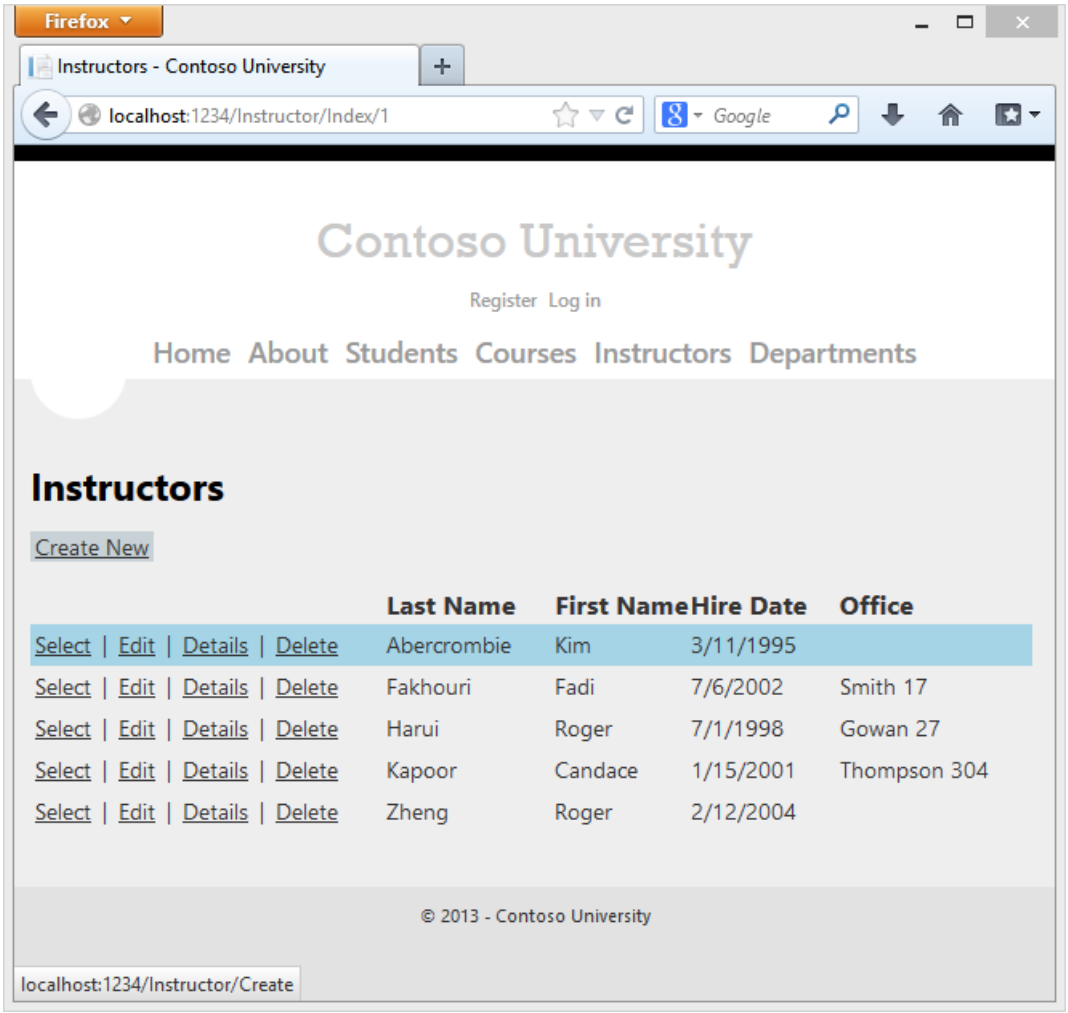
```
<td>
    @if (item.OfficeAssignment != null)
    {
        @item.OfficeAssignment.Location
    }
</td>
```

- Added code that will dynamically add `class="selectedrow"` to the `tr` element of the selected instructor. This sets a background color for the selected row using the CSS class that you created earlier. (The `valign` attribute will be useful in the following tutorial when you add a multi-row column to the table.)

```
string selectedRow = "";
if (item.InstructorID == ViewBag.InstructorID)
{
    selectedRow = "selectedrow";
}
<tr class="@selectedRow" valign="top">
```

- Added a new `ActionLink` labeled **Select** immediately before the other links in each row, which causes the selected instructor ID to be sent to the `Index` method.

Run the application and select the **Instructors** tab. The page displays the `Location` property of related `OfficeAssignment` entities and an empty table cell when there's no related `OfficeAssignment` entity.



In the *Views\Instructor\Index.cshtml* file, after the closing `table` element (at the end of the file), add the following highlighted code. This displays a list of courses related to an instructor when an instructor is selected.

```
<td>
                @if (item.OfficeAssignment != null)
                {
                    @item.OfficeAssignment.Location
                }
            </td>
        </tr>
    }
</table>

@if (Model.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
<table>
    <tr>
        <th></th>
        <th>ID</th>
        <th>Title</th>
        <th>Department</th>
    </tr>

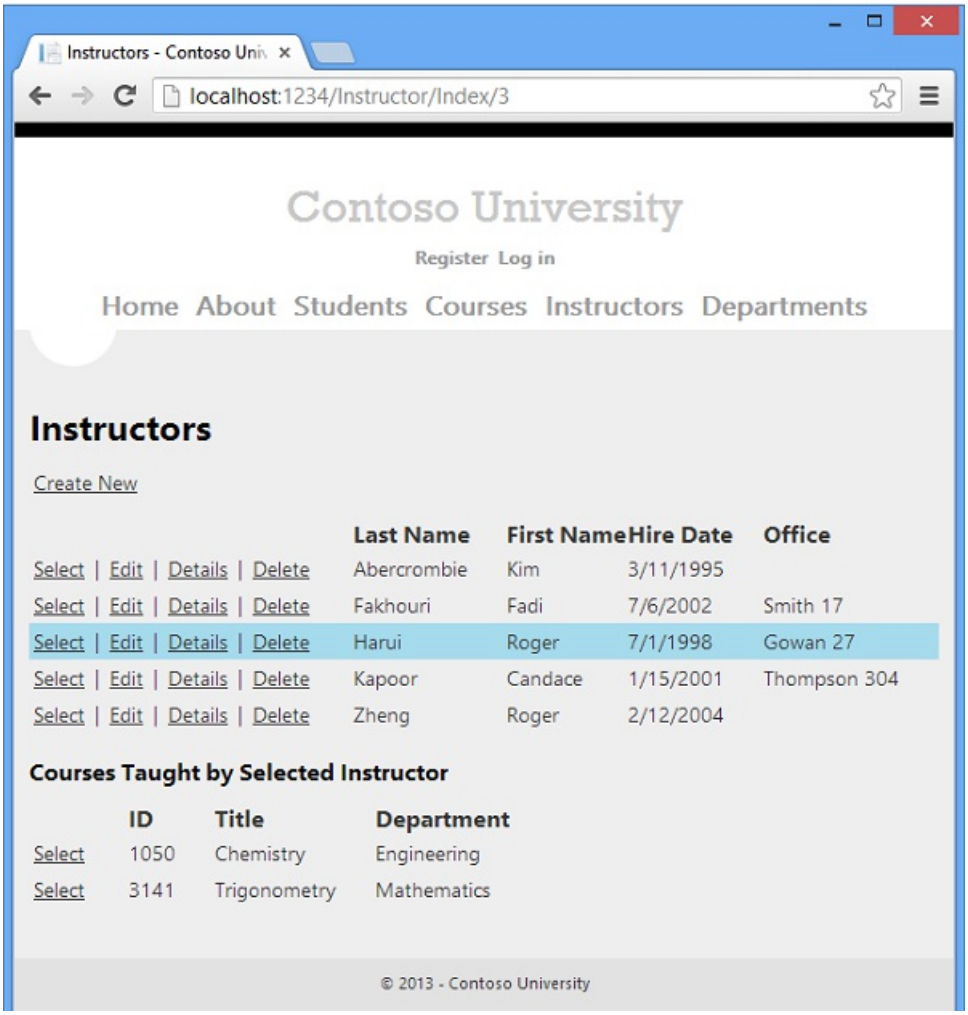
    @foreach (var item in Model.Courses)
    {
        string selectedRow = "";
        if (item.CourseID == ViewBag.CourseID)
        {
            selectedRow = "selectedrow";
        }
    <tr class="@selectedRow">
        <td>
            @Html.ActionLink("Select", "Index", new { courseID = item.CourseID })
        </td>
        <td>
            @item.CourseID
        </td>
        <td>
            @item.Title
        </td>
        <td>
            @item.Department.Name
        </td>
    </tr>
    }

</table>
}
```

This code reads the `Courses` property of the view model to display a list of courses. It also provides a `Select` hyperlink that sends the ID of the selected course to the `Index` action method.

> **NOTE**
>
> The *.css* file is cached by browsers. If you don't see the changes when you run the application, do a hard refresh (hold down the CTRL key while clicking the **Refresh** button, or press CTRL+F5).

Run the page and select an instructor. Now you see a grid that displays courses assigned to the selected instructor, and for each course you see the name of the assigned department.

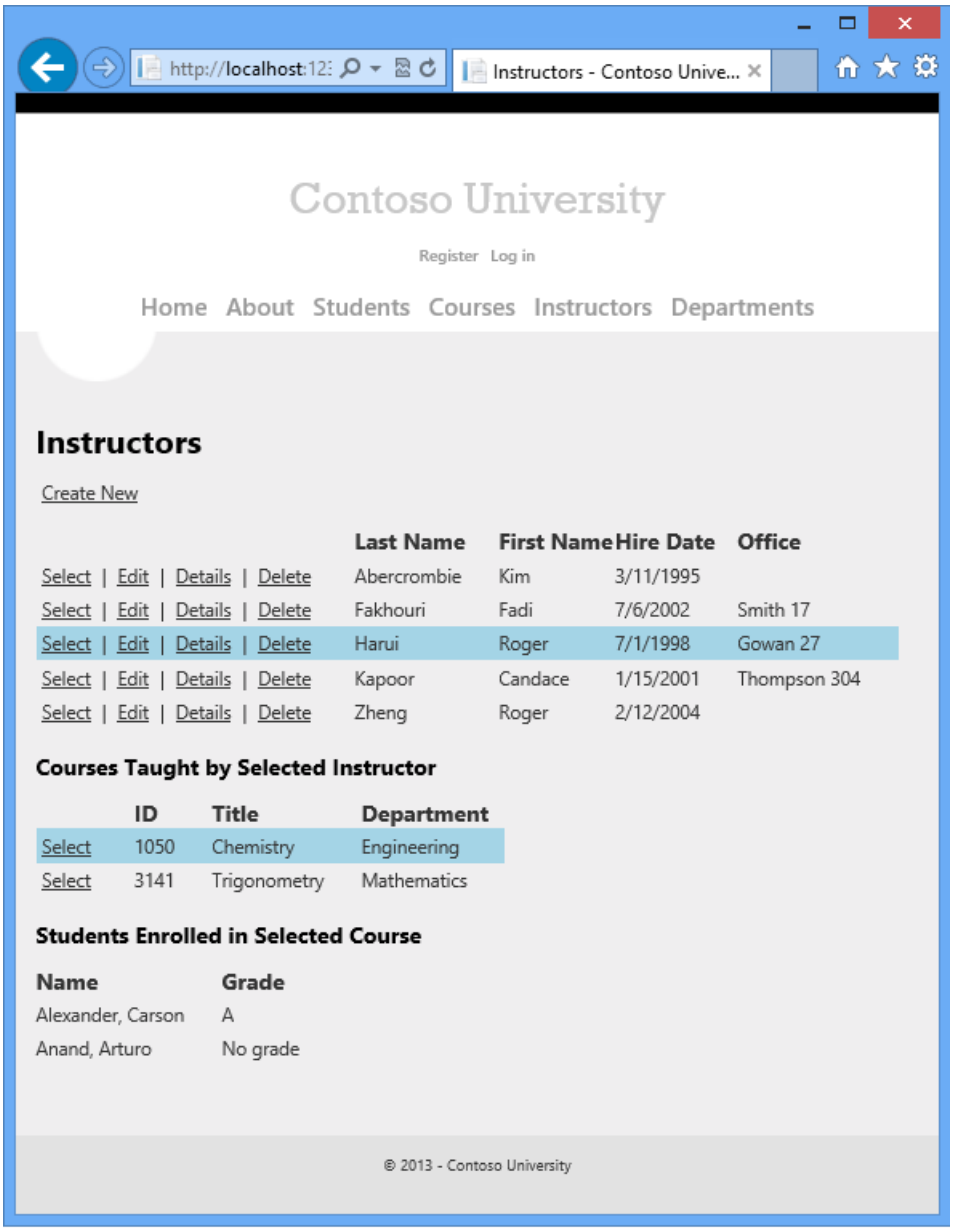


After the code block you just added, add the following code. This displays a list of the students who are enrolled in a course when that course is selected.

```
@if (Model.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course</h3>
    <table>
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

This code reads the `Enrollments` property of the view model in order to display a list of students enrolled in the course.

Run the page and select an instructor. Then select a course to see the list of enrolled students and their grades.



Adding Explicit Loading

Open *InstructorController.cs* and look at how the `Index` method gets the list of enrollments for a selected course:

```
if (courseID != null)
{
    ViewBag.CourseID = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

When you retrieved the list of instructors, you specified eager loading for the `Courses` navigation property and for the `Department` property of each course. Then you put the `Courses` collection in the view model, and now you're accessing the `Enrollments` navigation property from one entity in that collection. Because you didn't specify eager loading for the `Course.Enrollments` navigation property, the data from that property is appearing in the page as a result of lazy loading.

If you disabled lazy loading without changing the code in any other way, the `Enrollments` property would be null regardless of how many enrollments the course actually had. In that case, to load the `Enrollments` property, you'd have to specify either eager loading or explicit loading. You've already seen how to do eager loading. In order to see an example of explicit loading, replace the `Index` method with the following code, which explicitly loads the `Enrollments` property. The code changed are highlighted.

```
public ActionResult Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();

    viewModel.Instructors = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.Courses.Select(c => c.Department))
        .OrderBy(i => i.LastName);

    if (id != null)
    {
        ViewBag.InstructorID = id.Value;
        viewModel.Courses = viewModel.Instructors.Where(
            i => i.InstructorID == id.Value).Single().Courses;
    }

    if (courseID != null)
    {
        ViewBag.CourseID = courseID.Value;
        var selectedCourse = viewModel.Courses.Where(x => x.CourseID == courseID).Single();
        db.Entry(selectedCourse).Collection(x => x.Enrollments).Load();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            db.Entry(enrollment).Reference(x => x.Student).Load();
        }

        viewModel.Enrollments = selectedCourse.Enrollments;
    }

    return View(viewModel);
}
```

After getting the selected `Course` entity, the new code explicitly loads that course's `Enrollments` navigation property:

```
db.Entry(selectedCourse).Collection(x => x.Enrollments).Load();
```

Then it explicitly loads each `Enrollment` entity's related `Student` entity:

```
db.Entry(enrollment).Reference(x => x.Student).Load();
```

Notice that you use the `Collection` method to load a collection property, but for a property that holds just one entity, you use the `Reference` method. You can run the Instructor Index page now and you'll see no difference in what's displayed on the page, although you've changed how the data is retrieved.

Summary

You've now used all three ways (lazy, eager, and explicit) to load related data into navigation properties. In the next tutorial you'll learn how to update related data.

Links to other Entity Framework resources can be found in the ASP.NET Data Access Content Map.

PREVIOUS NEXT

Updating Related Data with the Entity Framework in an ASP.NET MVC Application (6 of 10)

2/20/2020 • 16 minutes to read • Edit Online

by Tom Dykstra

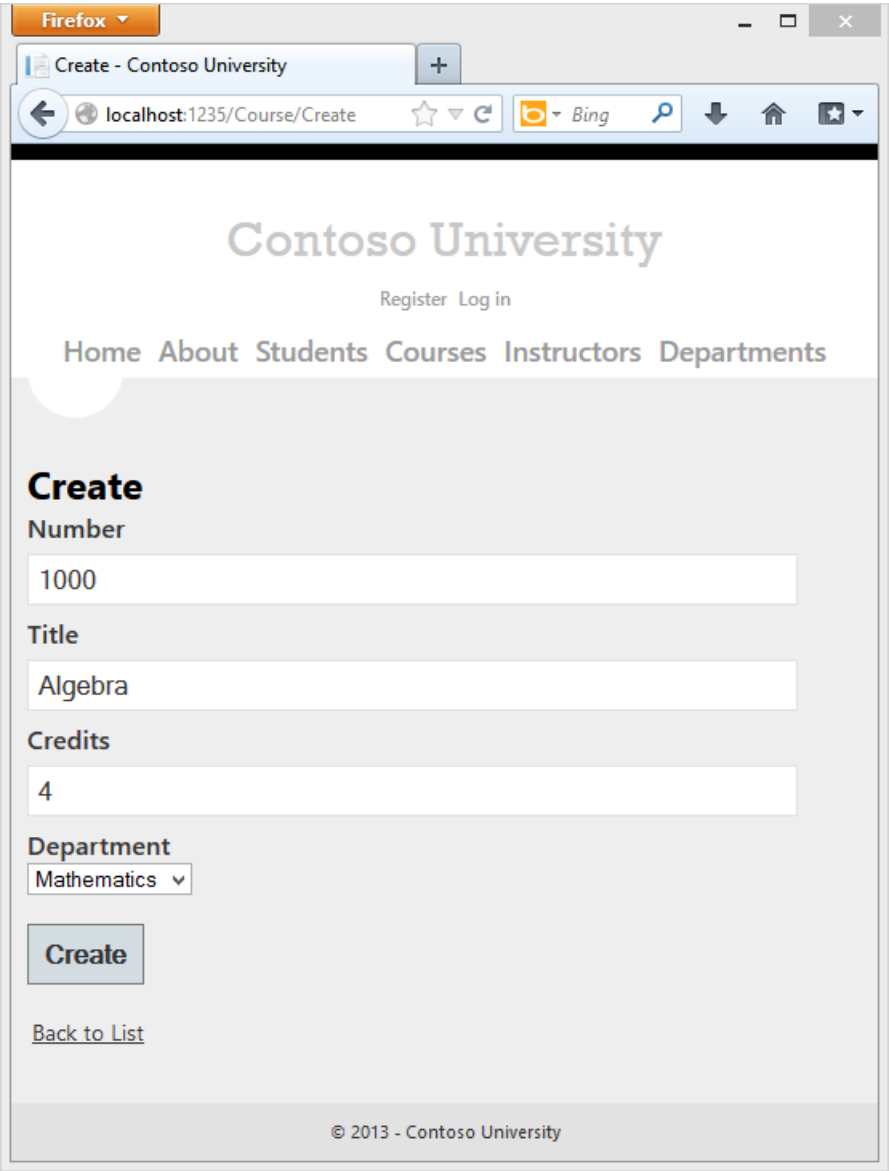
Download Completed Project

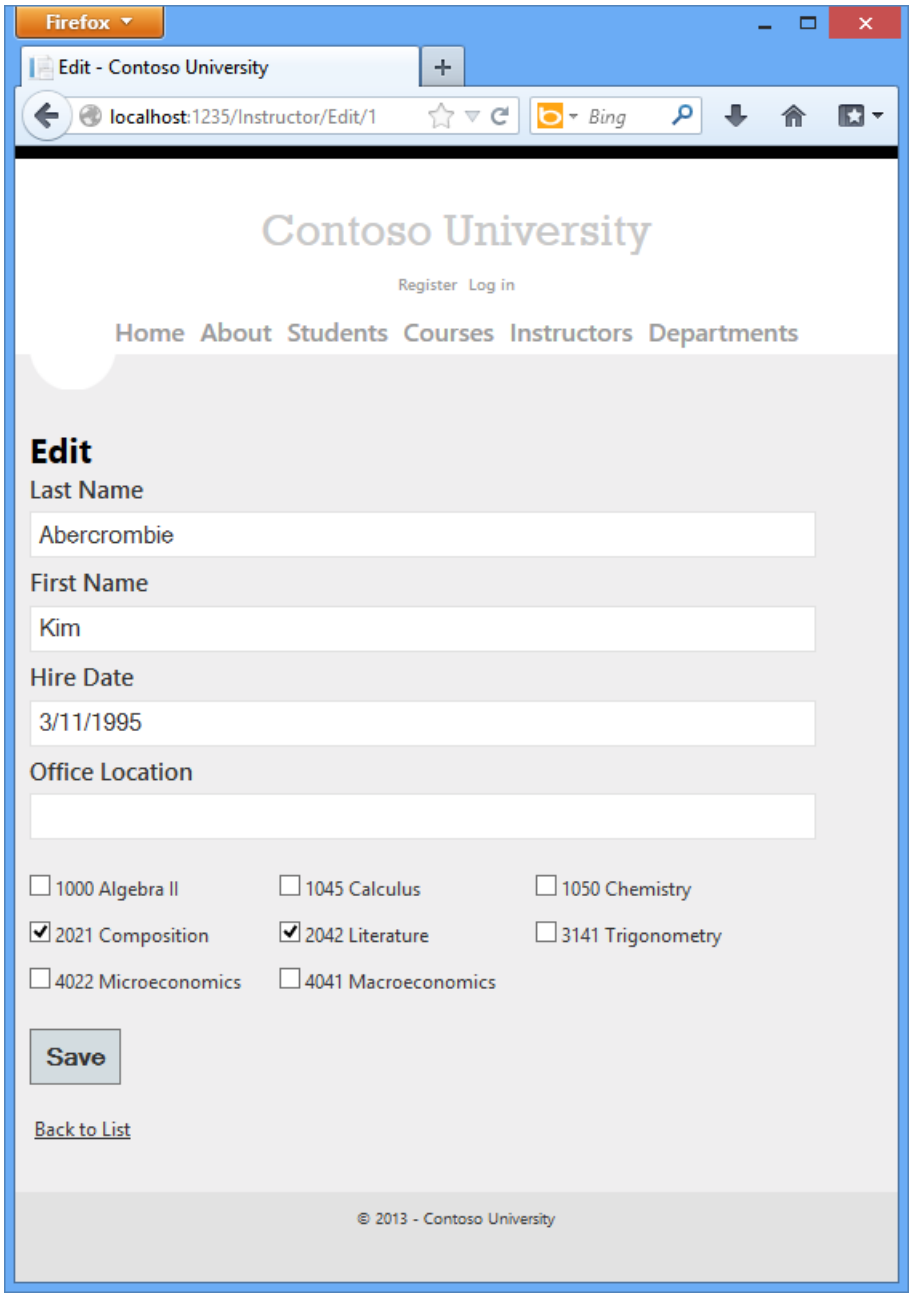
The Contoso University sample web application demonstrates how to create ASP.NET MVC 4 applications using the Entity Framework 5 Code First and Visual Studio 2012. For information about the tutorial series, see the first tutorial in the series. You can start the tutorial series from the beginning or download a starter project for this chapter and start here.

> **NOTE**
>
> If you run into a problem you can't resolve, download the completed chapter and try to reproduce your problem. You can generally find the solution to the problem by comparing your code to the completed code. For some common errors and how to solve them, see Errors and Workarounds.

In the previous tutorial you displayed related data; in this tutorial you'll update related data. For most relationships, this can be done by updating the appropriate foreign key fields. For many-to-many relationships, the Entity Framework doesn't expose the join table directly, so you must explicitly add and remove entities to and from the appropriate navigation properties.

The following illustrations show the pages that you'll work with.


Firefox
Create - Contoso University
localhost:1235/Course/Create
Bing
Contoso University
Register Log in
Home About Students Courses Instructors Departments
Create
Number
1000
Title
Algebra
Credits
4
Department
Mathematics
Create
Back to List
© 2013 - Contoso University



Customize the Create and Edit Pages for Courses

When a new course entity is created, it must have a relationship to an existing department. To facilitate this, the scaffolded code includes controller methods and Create and Edit views that include a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key property, and that's all the Entity Framework needs in order to load the `Department` navigation property with the appropriate `Department` entity. You'll use the scaffolded code, but change it slightly to add error handling and sort the drop-down list.

In *CourseController.cs*, delete the four `Edit` and `Create` methods and replace them with the following code:

```
public ActionResult Create()
{
   PopulateDepartmentsDropDownList();
   return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(
   [Bind(Include = "CourseID,Title,Credits,DepartmentID")]
   Course course)
{
```

```
    try
    {
       if (ModelState.IsValid)
       {
          db.Courses.Add(course);
          db.SaveChanges();
          return RedirectToAction("Index");
       }
    }
    catch (DataException /* dex */)
    {
       //Log the error (uncomment dex variable name after DataException and add a line here to write a log.)
       ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists, see your
system administrator.");
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}

public ActionResult Edit(int id)
{
    Course course = db.Courses.Find(id);
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(
     [Bind(Include = "CourseID,Title,Credits,DepartmentID")]
     Course course)
{
    try
    {
       if (ModelState.IsValid)
       {
          db.Entry(course).State = EntityState.Modified;
          db.SaveChanges();
          return RedirectToAction("Index");
       }
    }
    catch (DataException /* dex */)
    {
       //Log the error (uncomment dex variable name after DataException and add a line here to write a log.)
       ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists, see your
system administrator.");
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}

private void PopulateDepartmentsDropDownList(object selectedDepartment = null)
{
    var departmentsQuery = from d in db.Departments
                           orderby d.Name
                           select d;
    ViewBag.DepartmentID = new SelectList(departmentsQuery, "DepartmentID", "Name", selectedDepartment);
}
```

The `PopulateDepartmentsDropDownList` method gets a list of all departments sorted by name, creates a `SelectList` collection for a drop-down list, and passes the collection to the view in a `ViewBag` property. The method accepts the optional `selectedDepartment` parameter that allows the calling code to specify the item that will be selected when the drop-down list is rendered. The view will pass the name `DepartmentID` to the `DropDownList` helper, and the helper then knows to look in the `ViewBag` object for a `SelectList` named `DepartmentID`.

The `HttpGet` `Create` method calls the `PopulateDepartmentsDropDownList` method without setting the selected item,

because for a new course the department is not established yet:

```
public ActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

The `HttpGet` `Edit` method sets the selected item, based on the ID of the department that is already assigned to the course being edited:

```
public ActionResult Edit(int id)
{
    Course course = db.Courses.Find(id);
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

The `HttpPost` methods for both `Create` and `Edit` also include code that sets the selected item when they redisplay the page after an error:

```
catch (DataException /* dex */)
{
    //Log the error (uncomment dex variable name after DataException and add a line here to write a log.)
    ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists, see your
system administrator.");
}
PopulateDepartmentsDropDownList(course.DepartmentID);
return View(course);
```

This code ensures that when the page is redisplayed to show the error message, whatever department was selected stays selected.

In *Views\Course\Create.cshtml*, add the highlighted code to create a new course number field before the **Title** field. As explained in an earlier tutorial, primary key fields aren't scaffolded by default, but this primary key is meaningful, so you want the user to be able to enter the key value.

```
@model ContosoUniversity.Models.Course

@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>Course</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.CourseID)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.CourseID)
            @Html.ValidationMessageFor(model => model.CourseID)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Title)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Credits)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Credits)
            @Html.ValidationMessageFor(model => model.Credits)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.DepartmentID, "Department")
        </div>
        <div class="editor-field">
            @Html.DropDownList("DepartmentID", String.Empty)
            @Html.ValidationMessageFor(model => model.DepartmentID)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}

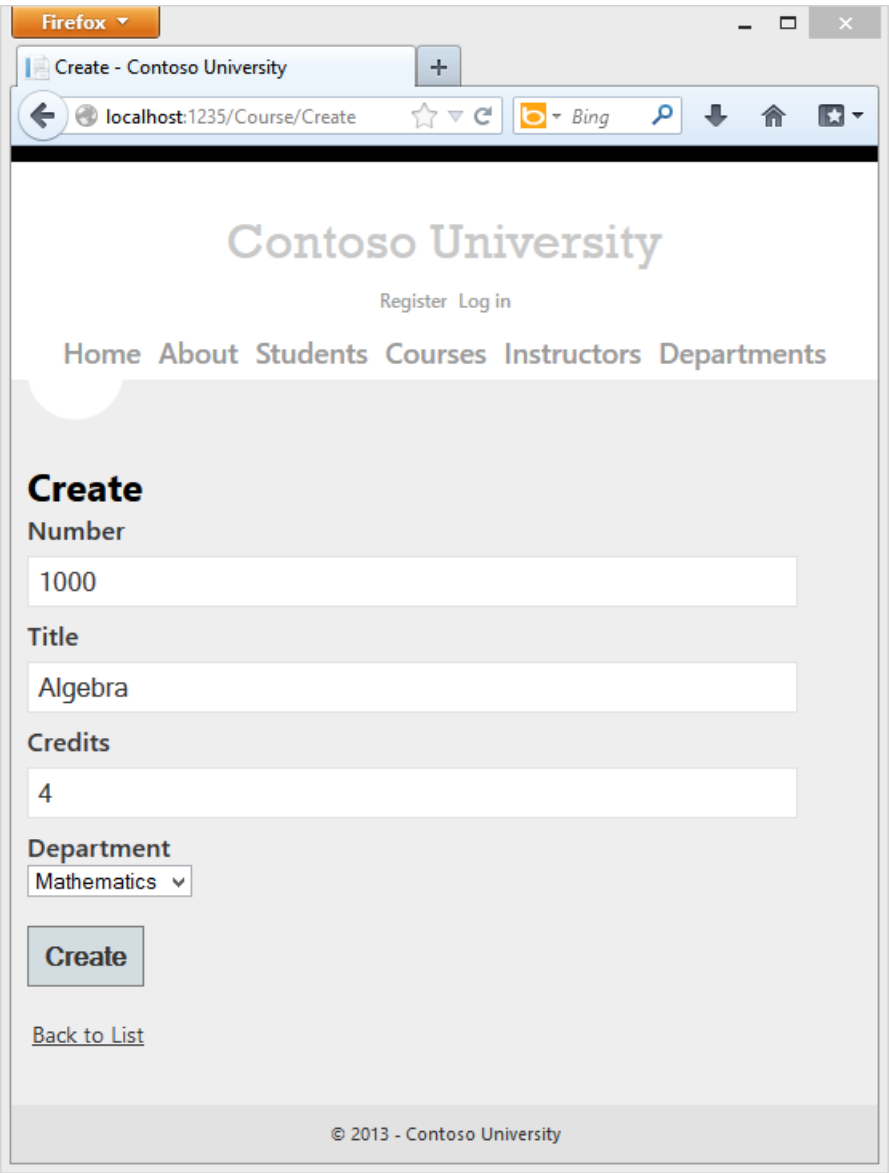
<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

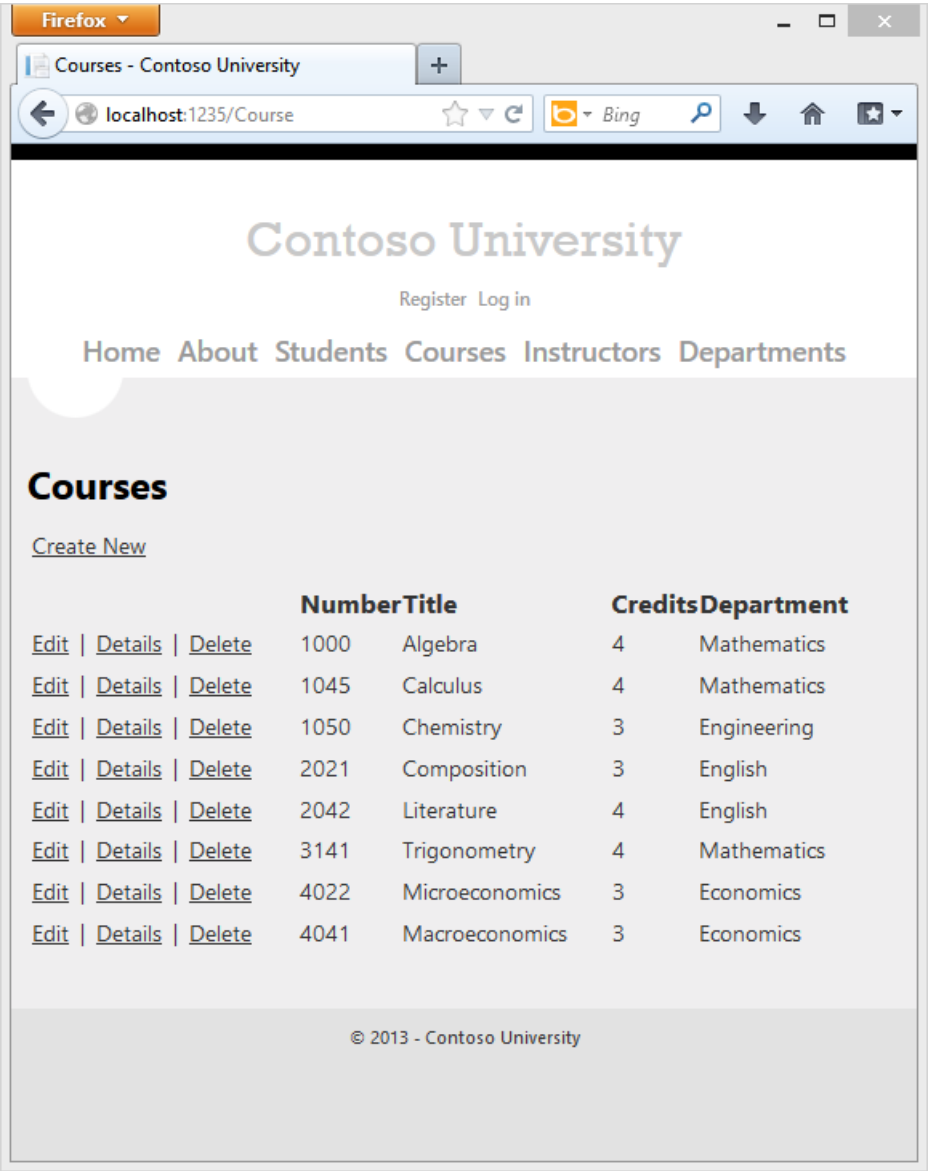
In *Views\Course\Edit.cshtml*, *Views\Course\Delete.cshtml*, and *Views\Course\Details.cshtml*, add a course number field before the **Title** field. Because it's the primary key, it's displayed, but it can't be changed.

```
<div class="editor-label">
    @Html.LabelFor(model => model.CourseID)
</div>
<div class="editor-field">
    @Html.DisplayFor(model => model.CourseID)
</div>
```

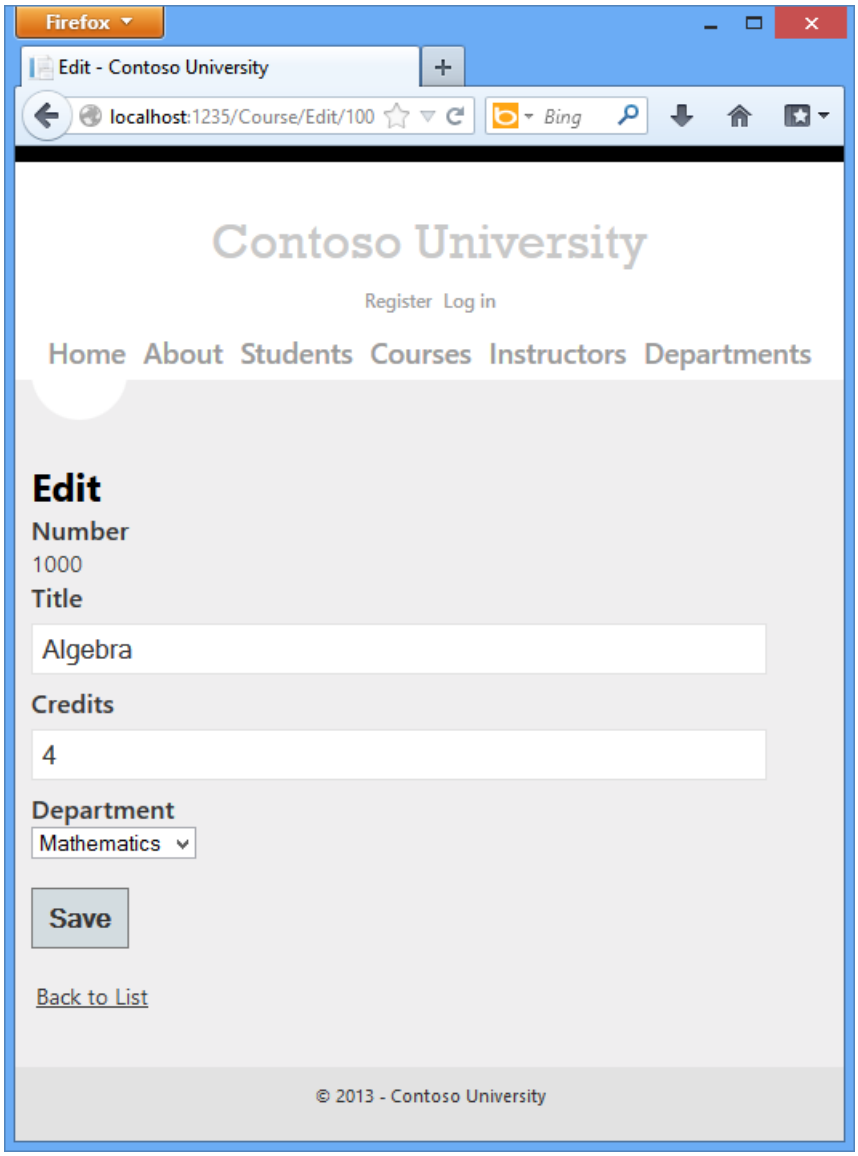
Run the **Create** page (display the Course Index page and click **Create New**) and enter data for a new course:



Click **Create**. The Course Index page is displayed with the new course added to the list. The department name in the Index page list comes from the navigation property, showing that the relationship was established correctly.



Run the **Edit** page (display the Course Index page and click **Edit** on a course).



Change data on the page and click **Save**. The Course Index page is displayed with the updated course data.

Adding an Edit Page for Instructors

When you edit an instructor record, you want to be able to update the instructor's office assignment. The `Instructor` entity has a one-to-zero-or-one relationship with the `OfficeAssignment` entity, which means you must handle the following situations:

- If the user clears the office assignment and it originally had a value, you must remove and delete the `OfficeAssignment` entity.
- If the user enters an office assignment value and it originally was empty, you must create a new `OfficeAssignment` entity.
- If the user changes the value of an office assignment, you must change the value in an existing `OfficeAssignment` entity.

Open *InstructorController.cs* and look at the `HttpGet` `Edit` method:

```
public ActionResult Edit(int id = 0)
{
    Instructor instructor = db.Instructors.Find(id);
    if (instructor == null)
    {
        return HttpNotFound();
    }
    ViewBag.InstructorID = new SelectList(db.OfficeAssignments, "InstructorID", "Location",
instructor.InstructorID);
    return View(instructor);
}
```

The scaffolded code here isn't what you want. It's setting up data for a drop-down list, but you what you need is a text box. Replace this method with the following code:

```
public ActionResult Edit(int id)
{
    Instructor instructor = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Where(i => i.InstructorID == id)
        .Single();
    return View(instructor);
}
```

This code drops the `ViewBag` statement and adds eager loading for the associated `OfficeAssignment` entity. You can't perform eager loading with the `Find` method, so the `Where` and `Single` methods are used instead to select the instructor.

Replace the `HttpPost` `Edit` method with the following code. which handles office assignment updates:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(int id, FormCollection formCollection)
{
   var instructorToUpdate = db.Instructors
       .Include(i => i.OfficeAssignment)
       .Where(i => i.InstructorID == id)
       .Single();

   if (TryUpdateModel(instructorToUpdate, "",
      new string[] { "LastName", "FirstMidName", "HireDate", "OfficeAssignment" }))
   {
      try
      {
         if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment.Location))
         {
            instructorToUpdate.OfficeAssignment = null;
         }

         db.Entry(instructorToUpdate).State = EntityState.Modified;
         db.SaveChanges();

         return RedirectToAction("Index");
      }
      catch (DataException /* dex */)
      {
         //Log the error (uncomment dex variable name after DataException and add a line here to write a log.
         ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists, see
your system administrator.");
      }
   }
   ViewBag.InstructorID = new SelectList(db.OfficeAssignments, "InstructorID", "Location", id);
   return View(instructorToUpdate);
}
```

The code does the following:

- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` navigation property. This is the same as what you did in the `HttpGet` `Edit` method.
- Updates the retrieved `Instructor` entity with values from the model binder. The TryUpdateModel overload used enables you to *whitelist* the properties you want to include. This prevents over-posting, as explained in the second tutorial.

  ```
  if (TryUpdateModel(instructorToUpdate, "",
        new string[] { "LastName", "FirstMidName", "HireDate", "OfficeAssignment" }))
  ```

- If the office location is blank, sets the `Instructor.OfficeAssignment` property to null so that the related row in the `OfficeAssignment` table will be deleted.

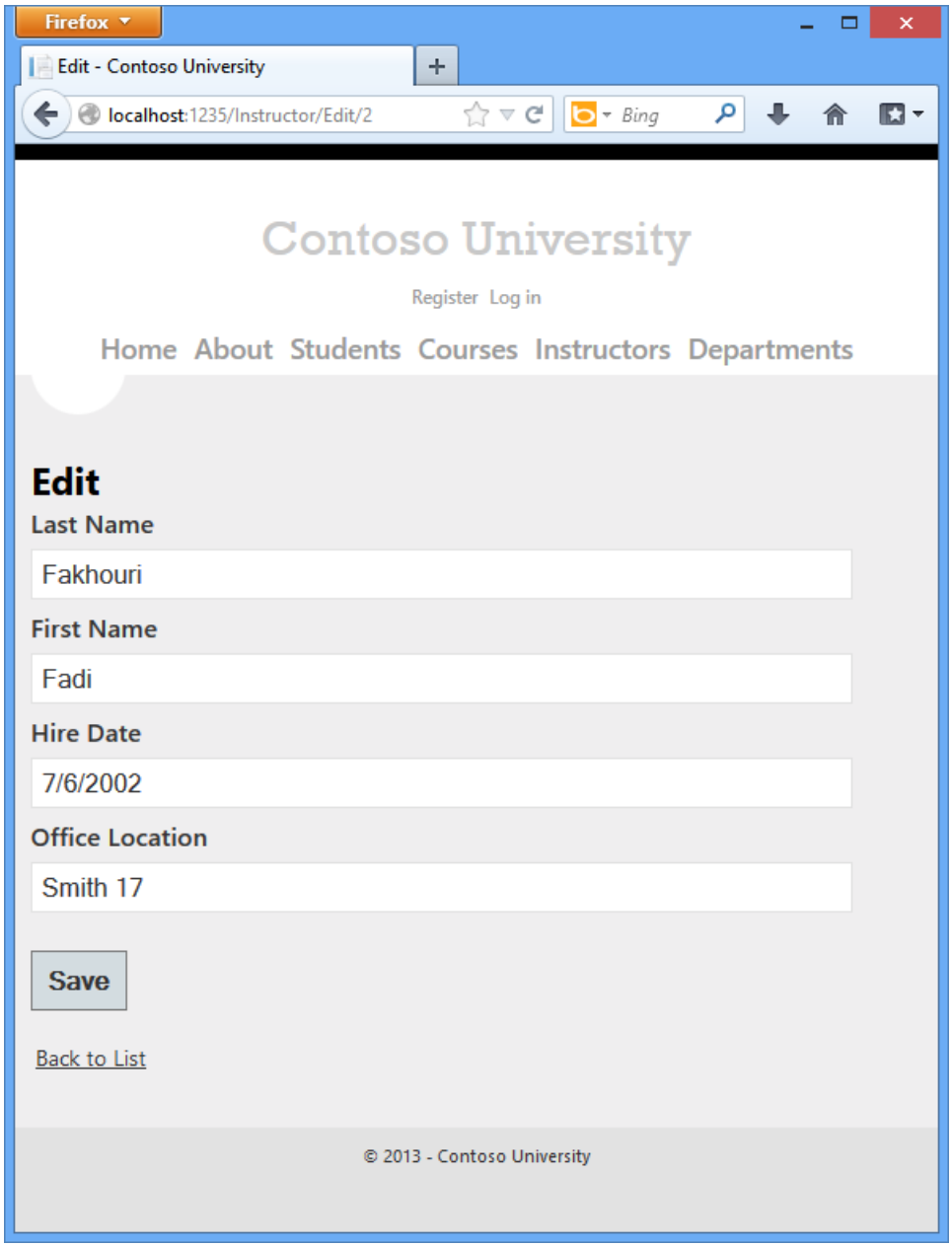
  ```
  if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment.Location))
  {
      instructorToUpdate.OfficeAssignment = null;
  }
  ```

- Saves the changes to the database.

In *Views\Instructor\Edit.cshtml*, after the `div` elements for the **Hire Date** field, add a new field for editing the office location:

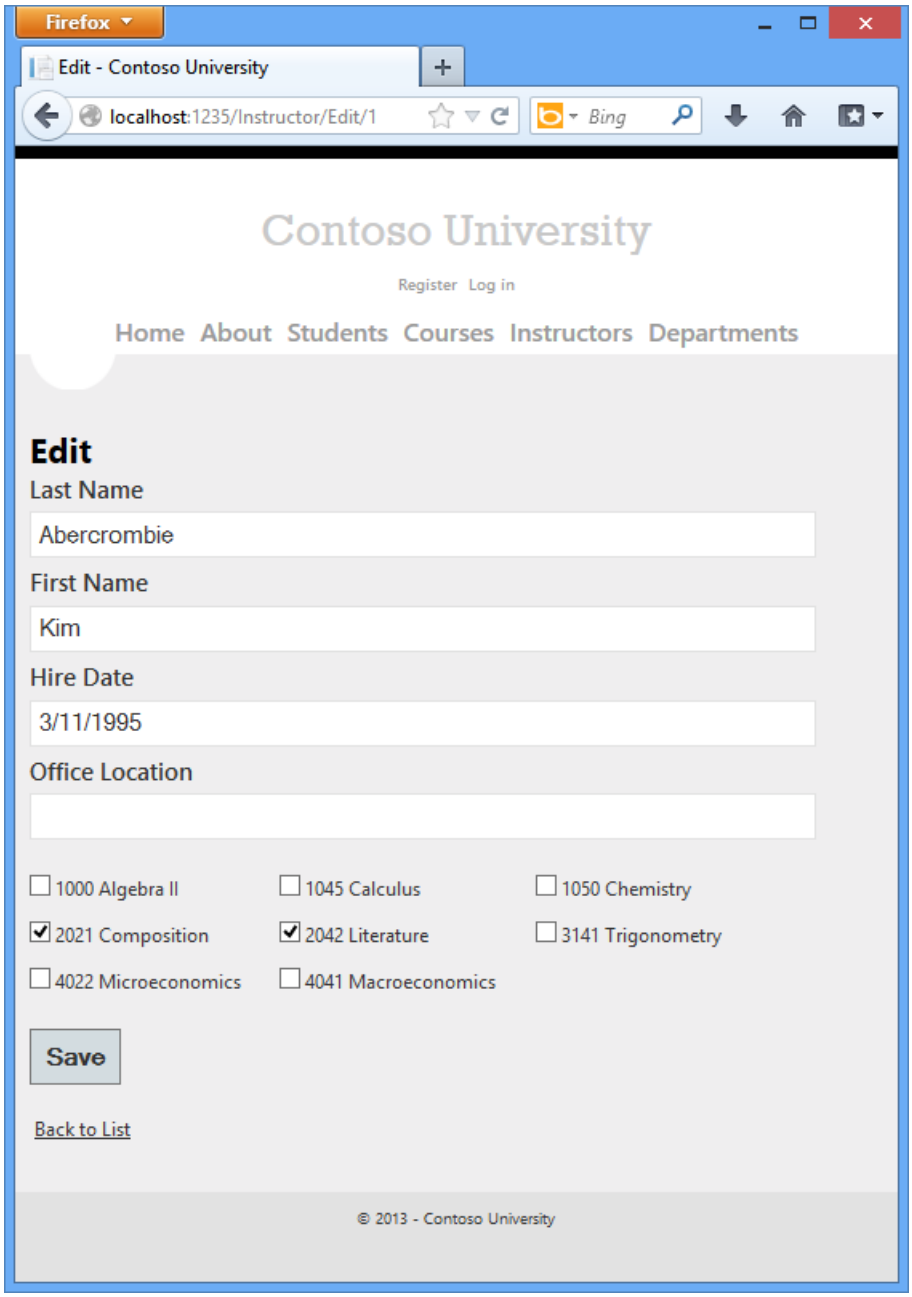
```
<div class="editor-label">
    @Html.LabelFor(model => model.OfficeAssignment.Location)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.OfficeAssignment.Location)
    @Html.ValidationMessageFor(model => model.OfficeAssignment.Location)
</div>
```

Run the page (select the **Instructors** tab and then click **Edit** on an instructor). Change the **Office Location** and click **Save**.



Adding Course Assignments to the Instructor Edit Page

Instructors may teach any number of courses. Now you'll enhance the Instructor Edit page by adding the ability to change course assignments using a group of check boxes, as shown in the following screen shot:



The relationship between the `Course` and `Instructor` entities is many-to-many, which means you do not have direct access to the join table. Instead, you will add and remove entities to and from the `Instructor.Courses` navigation property.

The UI that enables you to change which courses an instructor is assigned to is a group of check boxes. A check box for every course in the database is displayed, and the ones that the instructor is currently assigned to are selected. The user can select or clear check boxes to change course assignments. If the number of courses were much greater, you would probably want to use a different method of presenting the data in the view, but you'd use the same method of manipulating navigation properties in order to create or delete relationships.

To provide data to the view for the list of check boxes, you'll use a view model class. Create *AssignedCourseData.cs* in the *ViewModels* folder and replace the existing code with the following code:

```
namespace ContosoUniversity.ViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

In *InstructorController.cs*, replace the `HttpGet` `Edit` method with the following code. The changes are highlighted.

```
public ActionResult Edit(int id)
{
    Instructor instructor = db.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.Courses)
        .Where(i => i.InstructorID == id)
        .Single();
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

private void PopulateAssignedCourseData(Instructor instructor)
{
    var allCourses = db.Courses;
    var instructorCourses = new HashSet<int>(instructor.Courses.Select(c => c.CourseID));
    var viewModel = new List<AssignedCourseData>();
    foreach (var course in allCourses)
    {
        viewModel.Add(new AssignedCourseData
        {
            CourseID = course.CourseID,
            Title = course.Title,
            Assigned = instructorCourses.Contains(course.CourseID)
        });
    }
    ViewBag.Courses = viewModel;
}
```

The code adds eager loading for the `Courses` navigation property and calls the new `PopulateAssignedCourseData` method to provide information for the check box array using the `AssignedCourseData` view model class.

The code in the `PopulateAssignedCourseData` method reads through all `Course` entities in order to load a list of courses using the view model class. For each course, the code checks whether the course exists in the instructor's `Courses` navigation property. To create efficient lookup when checking whether a course is assigned to the instructor, the courses assigned to the instructor are put into a HashSet collection. The `Assigned` property is set to `true` for courses the instructor is assigned. The view will use this property to determine which check boxes must be displayed as selected. Finally, the list is passed to the view in a `ViewBag` property.

Next, add the code that's executed when the user clicks **Save**. Replace the `HttpPost` `Edit` method with the following code, which calls a new method that updates the `Courses` navigation property of the `Instructor` entity. The changes are highlighted.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(int id, FormCollection formCollection, string[] selectedCourses)
{
   var instructorToUpdate = db.Instructors
       .Include(i => i.OfficeAssignment)
       .Include(i => i.Courses)
       .Where(i => i.InstructorID == id)
       .Single();
   if (TryUpdateModel(instructorToUpdate, "",
      new string[] { "LastName", "FirstMidName", "HireDate", "OfficeAssignment" }))
   {
      try
      {
         if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment.Location))
         {
            instructorToUpdate.OfficeAssignment = null;
         }

         UpdateInstructorCourses(selectedCourses, instructorToUpdate);

         db.Entry(instructorToUpdate).State = EntityState.Modified;
         db.SaveChanges();

         return RedirectToAction("Index");
      }
      catch (DataException /* dex */)
      {
         //Log the error (uncomment dex variable name after DataException and add a line here to write a log.
         ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists see your
system administrator.");
      }
   }
   PopulateAssignedCourseData(instructorToUpdate);
   return View(instructorToUpdate);
}

private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
   if (selectedCourses == null)
   {
      instructorToUpdate.Courses = new List<Course>();
      return;
   }

   var selectedCoursesHS = new HashSet<string>(selectedCourses);
   var instructorCourses = new HashSet<int>
       (instructorToUpdate.Courses.Select(c => c.CourseID));
   foreach (var course in db.Courses)
   {
      if (selectedCoursesHS.Contains(course.CourseID.ToString()))
      {
         if (!instructorCourses.Contains(course.CourseID))
         {
            instructorToUpdate.Courses.Add(course);
         }
      }
      else
      {
         if (instructorCourses.Contains(course.CourseID))
         {
            instructorToUpdate.Courses.Remove(course);
         }
      }
   }
}
```

Since the view doesn't have a collection of `Course` entities, the model binder can't automatically update the `Courses` navigation property. Instead of using the model binder to update the Courses navigation property, you'll do that in the new `UpdateInstructorCourses` method. Therefore you need to exclude the `Courses` property from model binding. This doesn't require any change to the code that calls TryUpdateModel because you're using the *whitelisting* overload and `Courses` isn't in the include list.

If no check boxes were selected, the code in `UpdateInstructorCourses` initializes the `Courses` navigation property with an empty collection:

```
if (selectedCourses == null)
{
    instructorToUpdate.Courses = new List<Course>();
    return;
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the view. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the check box for a course was selected but the course isn't in the `Instructor.Courses` navigation property, the course is added to the collection in the navigation property.

```
if (selectedCoursesHS.Contains(course.CourseID.ToString()))
{
    if (!instructorCourses.Contains(course.CourseID))
    {
        instructorToUpdate.Courses.Add(course);
    }
}
```

If the check box for a course wasn't selected, but the course is in the `Instructor.Courses` navigation property, the course is removed from the navigation property.

```
else
{
    if (instructorCourses.Contains(course.CourseID))
    {
        instructorToUpdate.Courses.Remove(course);
    }
}
```

In *Views\Instructor\Edit.cshtml*, add a **Courses** field with an array of check boxes by adding the following highlighted code immediately after the `div` elements for the `OfficeAssignment` field:

```
@model ContosoUniversity.Models.Instructor

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>Instructor</legend>
```

```
        @Html.HiddenFor(model => model.InstructorID)

        <div class="editor-label">
            @Html.LabelFor(model => model.LastName)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.LastName)
            @Html.ValidationMessageFor(model => model.LastName)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.FirstMidName)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.FirstMidName)
            @Html.ValidationMessageFor(model => model.FirstMidName)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.HireDate)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.HireDate)
            @Html.ValidationMessageFor(model => model.HireDate)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.OfficeAssignment.Location)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.OfficeAssignment.Location)
            @Html.ValidationMessageFor(model => model.OfficeAssignment.Location)
        </div>

        <div class="editor-field">
    <table>
        <tr>
            @{
                int cnt = 0;
                List<ContosoUniversity.ViewModels.AssignedCourseData> courses = ViewBag.Courses;

                foreach (var course in courses) {
                    if (cnt++ % 3 == 0) {
                        @:  </tr> <tr>
                    }
                    @: <td>
                        <input type="checkbox"
                               name="selectedCourses"
                               value="@course.CourseID"
                               @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                        @course.CourseID @:  @course.Title
                    @:</td>
                }
                @: </tr>
            }
    </table>
</div>
        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
```

```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

This code creates an HTML table that has three columns. In each column is a check box followed by a caption that consists of the course number and title. The check boxes all have the same name ("selectedCourses"), which informs the model binder that they are to be treated as a group. The `value` attribute of each check box is set to the value of `CourseID.` When the page is posted, the model binder passes an array to the controller that consists of the `CourseID` values for only the check boxes which are selected.

When the check boxes are initially rendered, those that are for courses assigned to the instructor have `checked` attributes, which selects them (displays them checked).

After changing course assignments, you'll want to be able to verify the changes when the site returns to the `Index` page. Therefore, you need to add a column to the table in that page. In this case you don't need to use the `ViewBag` object, because the information you want to display is already in the `Courses` navigation property of the `Instructor` entity that you're passing to the page as the model.

In *Views\Instructor\Index.cshtml*, add a **Courses** heading immediately following the **Office** heading, as shown in the following example:

```
<tr>
    <th></th>
    <th>Last Name</th>
    <th>First Name</th>
    <th>Hire Date</th>
    <th>Office</th>
    <th>Courses</th>
</tr>
```

Then add a new detail cell immediately following the office location detail cell:

```
@model ContosoUniversity.ViewModels.InstructorIndexData

@{
    ViewBag.Title = "Instructors";
}

<h2>Instructors</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th></th>
        <th>Last Name</th>
        <th>First Name</th>
        <th>Hire Date</th>
        <th>Office</th>
        <th>Courses</th>
    </tr>
    @foreach (var item in Model.Instructors)
    {
        string selectedRow = "";
        if (item.InstructorID == ViewBag.InstructorID)
        {
            selectedRow = "selectedrow";
        }
        <tr class="@selectedRow" valign="top">
            <td>
                @Html.ActionLink("Select", "Index", new { id = item.InstructorID }) |
                @Html.ActionLink("Edit", "Edit", new { id = item.InstructorID }) |
```

```
                @Html.ActionLink("Edit", "Edit", new { id = item.InstructorID }) |
                @Html.ActionLink("Details", "Details", new { id = item.InstructorID }) |
                @Html.ActionLink("Delete", "Delete", new { id = item.InstructorID })
            </td>
            <td>
                @item.LastName
            </td>
            <td>
                @item.FirstMidName
            </td>
            <td>
                @String.Format("{0:d}", item.HireDate)
            </td>
            <td>
                @if (item.OfficeAssignment != null)
                {
                    @item.OfficeAssignment.Location
                }
            </td>
            <td>
                @{
                foreach (var course in item.Courses)
                {
                    @course.CourseID @:  @course.Title <br />
                }
                }
            </td>
        </tr>
    }
</table>

@if (Model.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table>
        <tr>
            <th></th>
            <th>ID</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == ViewBag.CourseID)
            {
                selectedRow = "selectedrow";
            }

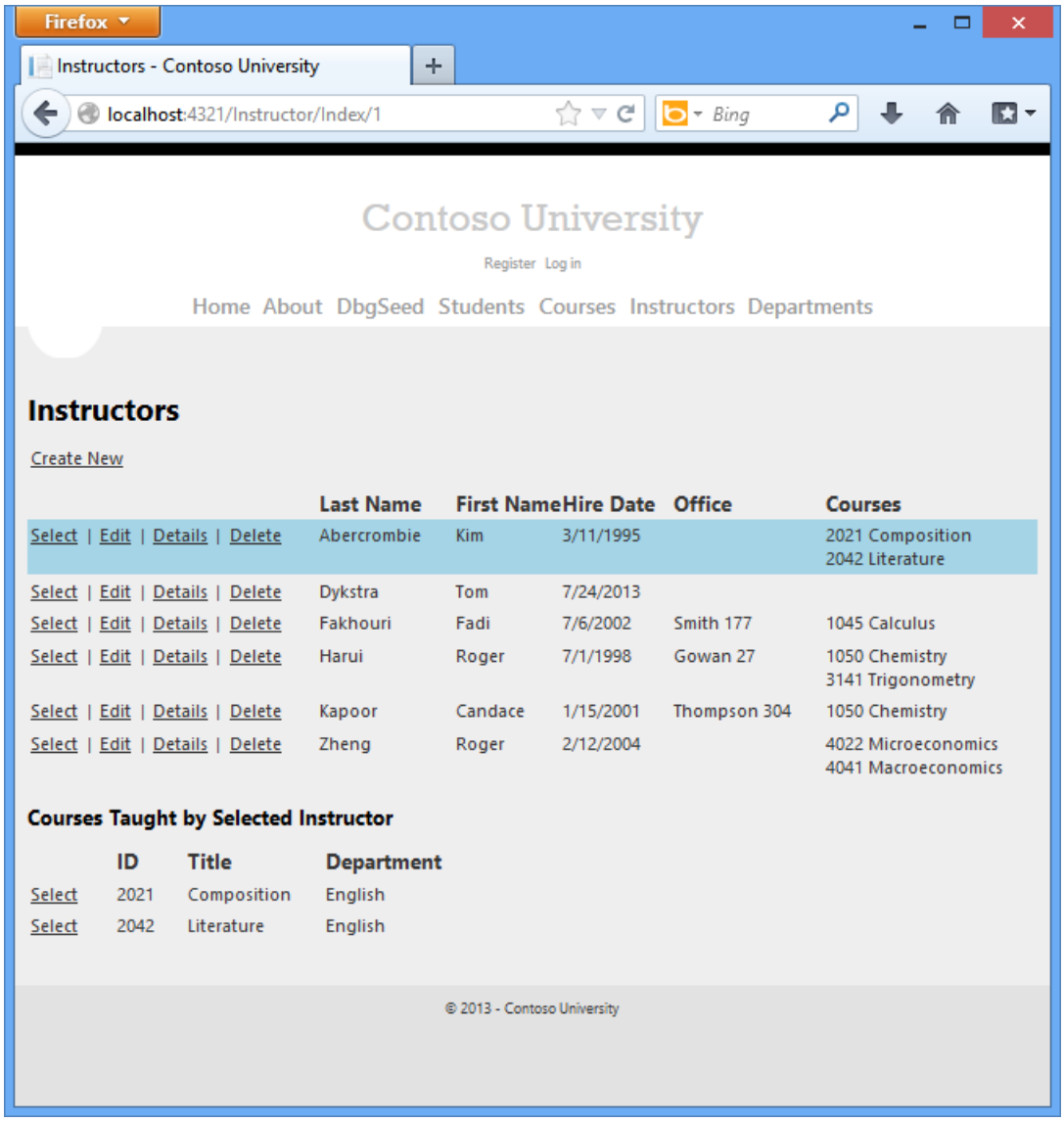
            <tr class="@selectedRow">

                <td>
                    @Html.ActionLink("Select", "Index", new { courseID = item.CourseID })
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

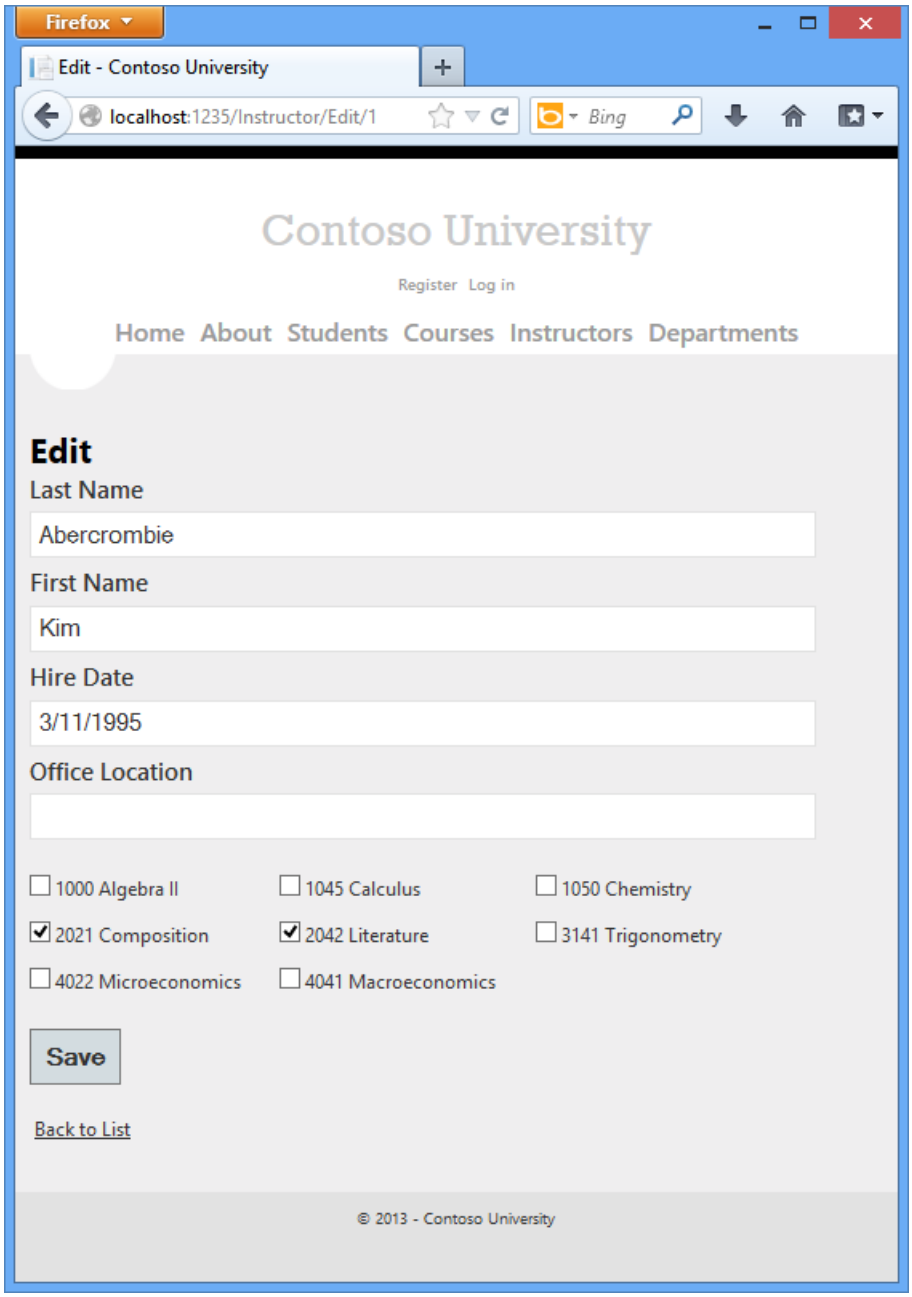
    </table>
}
```

```
@if (Model.Enrollments != null)
{
    <h3>Students Enrolled in Selected Course</h3>
    <table>
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

Run the **Instructor Index** page to see the courses assigned to each instructor:



Click **Edit** on an instructor to see the Edit page.



Change some course assignments and click **Save**. The changes you make are reflected on the Index page.

Note: The approach taken to edit instructor course data works well when there is a limited number of courses. For collections that are much larger, a different UI and a different updating method would be required.

Update the Delete Method

Change the code in the HttpPost Delete method so the office assignment record (if any) is deleted when the instructor is deleted:

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
   Instructor instructor = db.Instructors
     .Include(i => i.OfficeAssignment)
     .Where(i => i.InstructorID == id)
     .Single();

   instructor.OfficeAssignment = null;
   db.Instructors.Remove(instructor);
   db.SaveChanges();
   return RedirectToAction("Index");
}
```

If you try to delete an instructor who is assigned to a department as administrator, you'll get a referential integrity error. See the current version of this tutorial for additional code that will automatically remove the instructor from any department where the instructor is assigned as an administrator.

Summary

You have now completed this introduction to working with related data. So far in these tutorials you've done a full range of CRUD operations, but you haven't dealt with concurrency issues. The next tutorial will introduce the topic of concurrency, explain options for handling it, and add concurrency handling to the CRUD code you've already written for one entity type.

Links to other Entity Framework resources, can be found at the end of the last tutorial in this series.

PREVIOUS NEXT

Handling Concurrency with the Entity Framework in an ASP.NET MVC Application (7 of 10)

2/21/2020 • 16 minutes to read • Edit Online

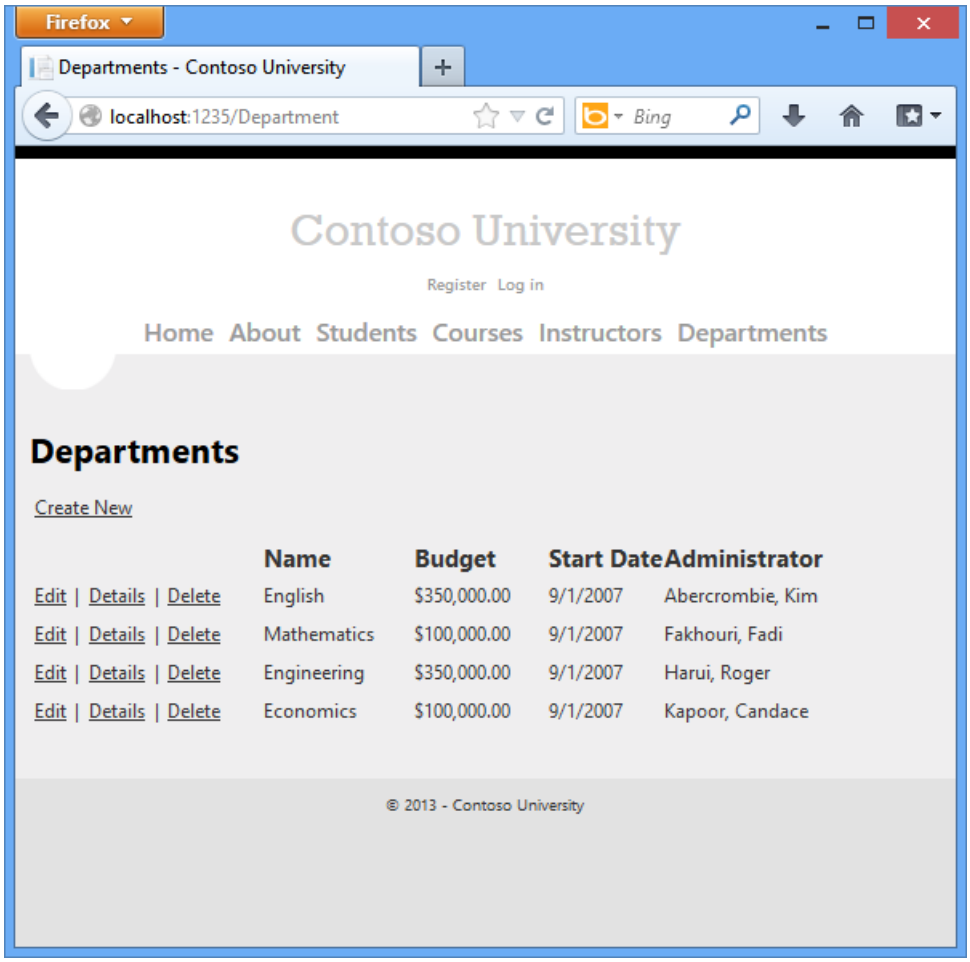
by Tom Dykstra

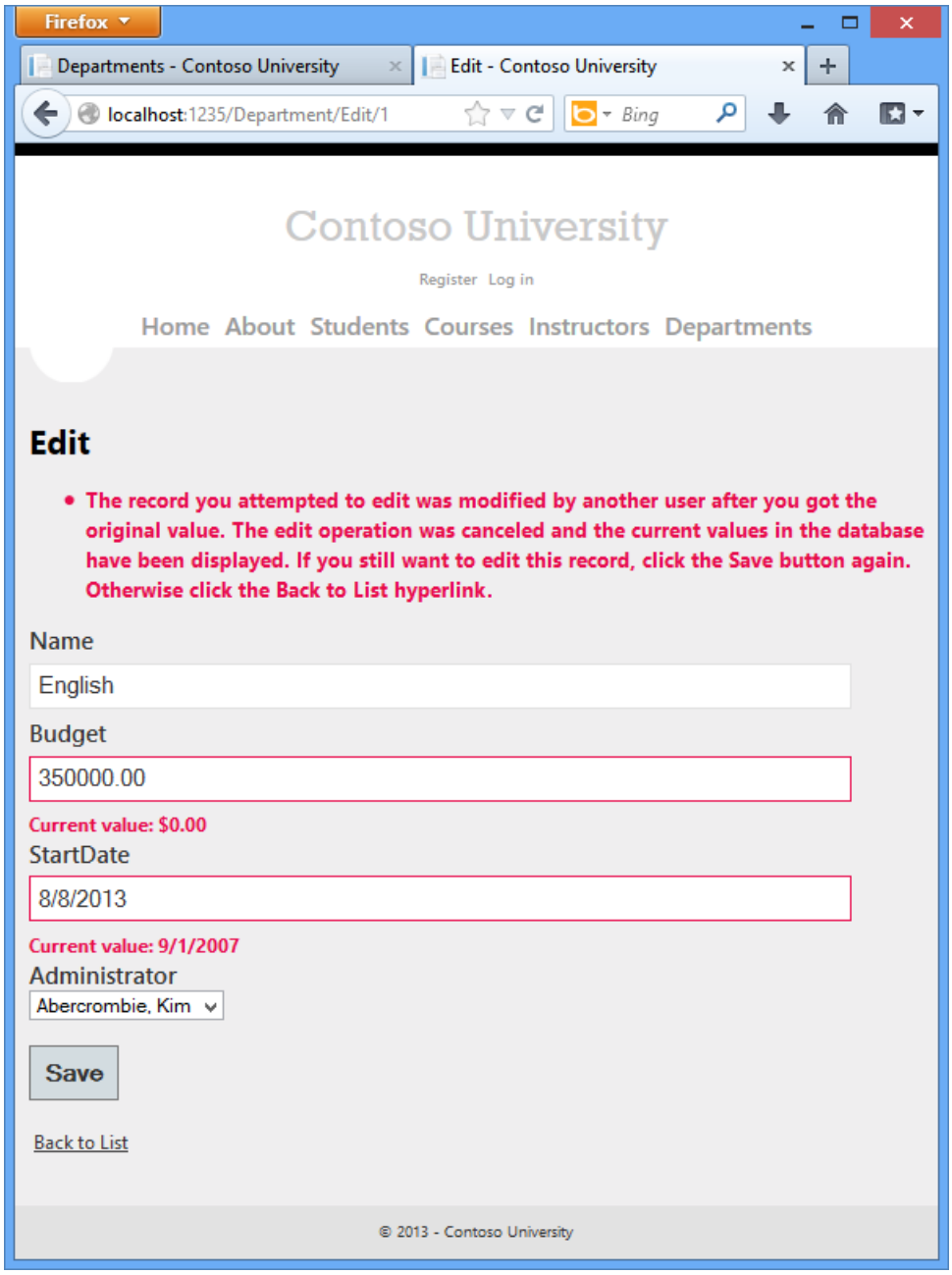
Download Completed Project

The Contoso University sample web application demonstrates how to create ASP.NET MVC 4 applications using the Entity Framework 5 Code First and Visual Studio 2012. For information about the tutorial series, see the first tutorial in the series. You can start the tutorial series from the beginning or download a starter project for this chapter and start here.

> **NOTE**
>
> If you run into a problem you can't resolve, download the completed chapter and try to reproduce your problem. You can generally find the solution to the problem by comparing your code to the completed code. For some common errors and how to solve them, see Errors and Workarounds.

In the previous two tutorials you worked with related data. This tutorial shows how to handle concurrency. You'll create web pages that work with the `Department` entity, and the pages that edit and delete `Department` entities will handle concurrency errors. The following illustrations show the Index and Delete pages, including some messages that are displayed if a concurrency conflict occurs.





Concurrency Conflicts

A concurrency conflict occurs when one user displays an entity's data in order to edit it, and then another user updates the same entity's data before the first user's change is written to the database. If you don't enable the detection of such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable: if there are few users, or few updates, or if isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit. In that case, you don't have to configure the application to handle concurrency conflicts.

Pessimistic Concurrency (Locking)

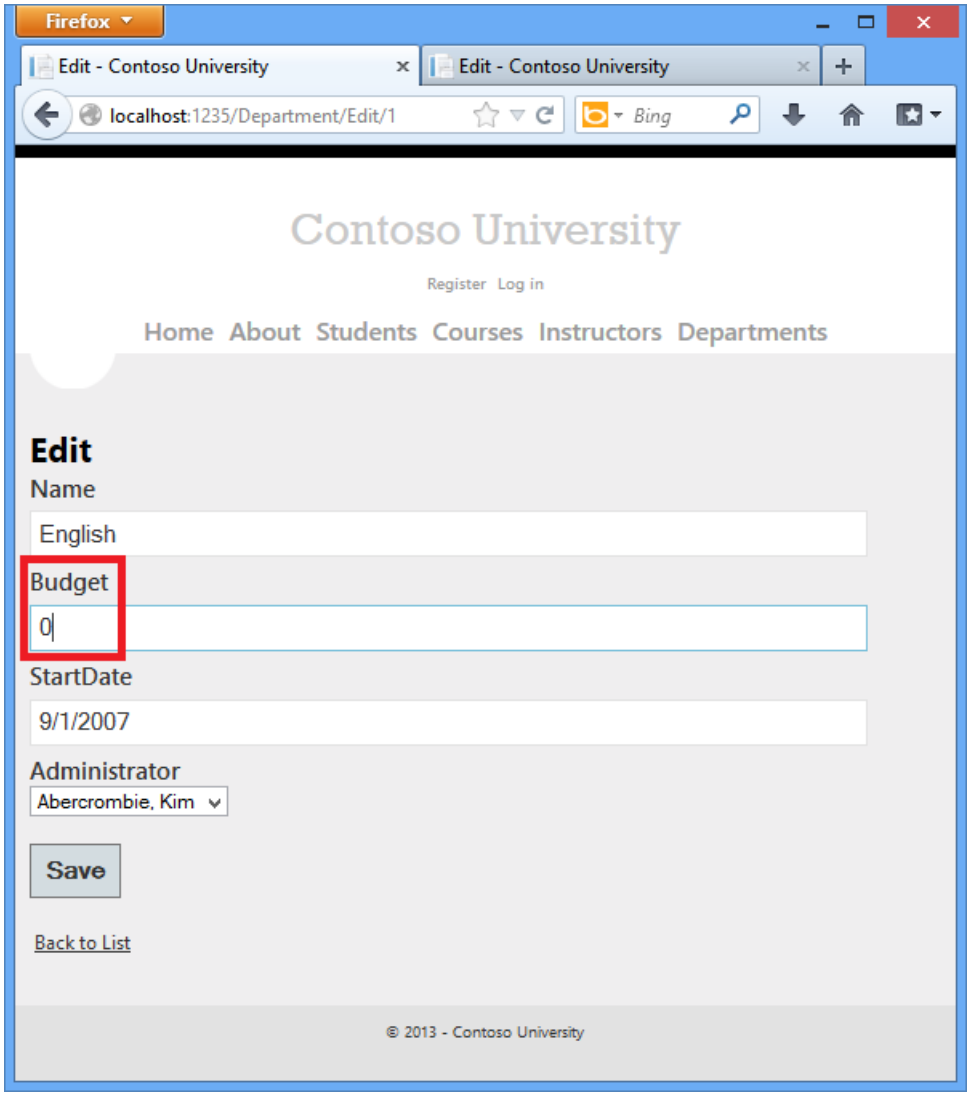
If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called *pessimistic concurrency*. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

Managing locks has disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases (that is, it

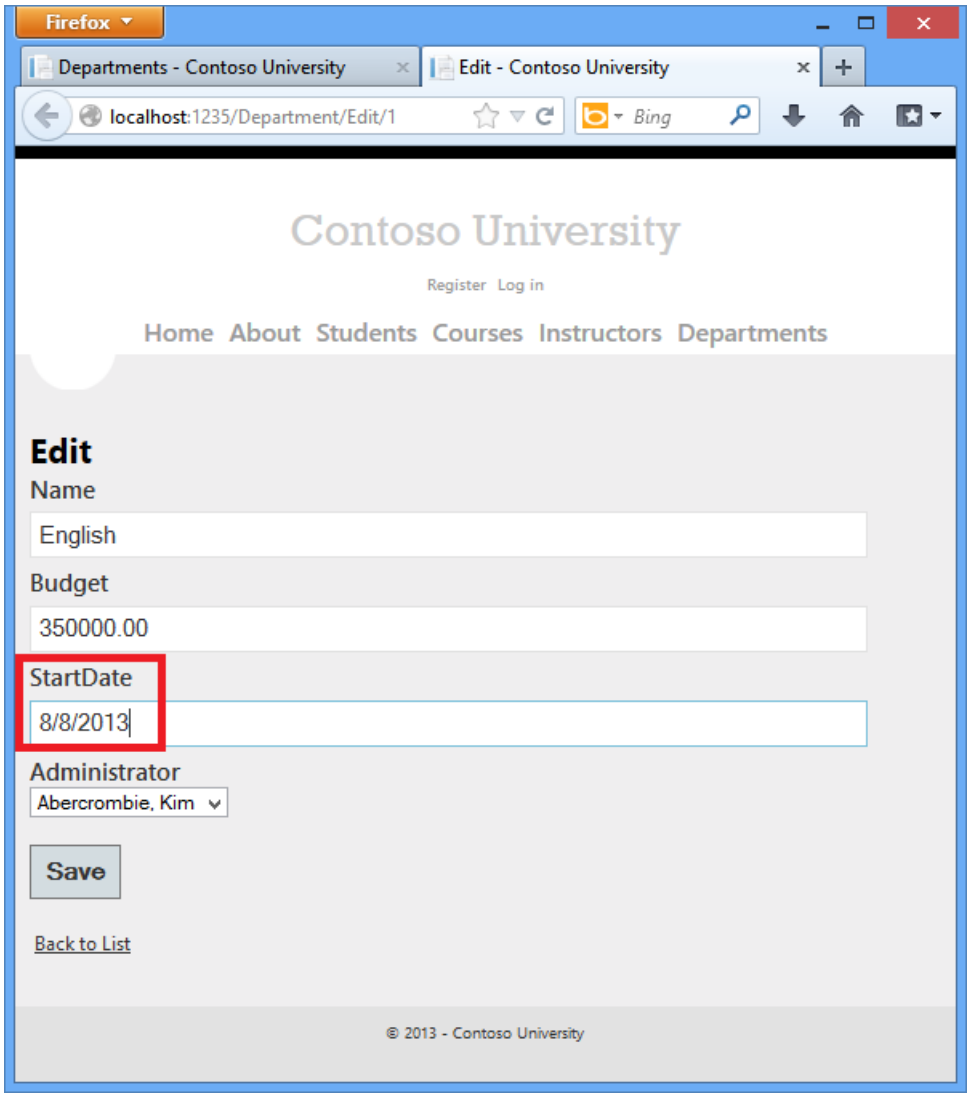
doesn't scale well). For these reasons, not all database management systems support pessimistic concurrency. The Entity Framework provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

The alternative to pessimistic concurrency is *optimistic concurrency*. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, John runs the Departments Edit page, changes the **Budget** amount for the English department from $350,000.00 to $0.00.



Before John clicks **Save**, Jane runs the same page and changes the **Start Date** field from 9/1/2007 to 8/8/2013.



John clicks **Save** first and sees his change when the browser returns to the Index page, then Jane clicks **Save**. What happens next is determined by how you handle concurrency conflicts. Some of the options include the following:

- You can keep track of which property a user has modified and update only the corresponding columns in the database. In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the English department, they'll see both John's and Jane's changes — a start date of 8/8/2013 and a budget of Zero dollars.

  This method of updating can reduce the number of conflicts that could result in data loss, but it can't avoid data loss if competing changes are made to the same property of an entity. Whether the Entity Framework works this way depends on how you implement your update code. It's often not practical in a web application, because it can require that you maintain large amounts of state in order to keep track of all original property values for an entity as well as new values. Maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself (for example, in hidden fields).

- You can let Jane's change overwrite John's change. The next time someone browses the English department, they'll see 8/8/2013 and the restored $350,000.00 value. This is called a *Client Wins* or *Last in Wins* scenario. (The client's values take precedence over what's in the data store.) As noted in the introduction to this section, if you don't do any coding for concurrency handling, this will happen automatically.

- You can prevent Jane's change from being updated in the database. Typically, you would display an error message, show her the current state of the data, and allow her to reapply her changes if she still wants to make them. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You'll implement the Store Wins scenario in this tutorial. This method ensures that

no changes are overwritten without a user being alerted to what's happening.

Detecting Concurrency Conflicts

You can resolve conflicts by handling OptimisticConcurrencyException exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database and the data model appropriately. Some options for enabling conflict detection include the following:

- In the database table, include a tracking column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the `Where` clause of SQL `Update` or `Delete` commands.

  The data type of the tracking column is typically rowversion. The rowversion value is a sequential number that's incremented each time the row is updated. In an `Update` or `Delete` command, the `Where` clause includes the original value of the tracking column (the original row version). If the row being updated has been changed by another user, the value in the `rowversion` column is different than the original value, so the `Update` or `Delete` statement can't find the row to update because of the `Where` clause. When the Entity Framework finds that no rows have been updated by the `Update` or `Delete` command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

- Configure the Entity Framework to include the original values of every column in the table in the `Where` clause of `Update` and `Delete` commands.

  As in the first option, if anything in the row has changed since the row was first read, the `Where` clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. For database tables that have many columns, this approach can result in very large `Where` clauses, and can require that you maintain large amounts of state. As noted earlier, maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself. Therefore this approach generally not recommended, and it isn't the method used in this tutorial.

  If you do want to implement this approach to concurrency, you have to mark all non-primary-key properties in the entity you want to track concurrency for by adding the ConcurrencyCheck attribute to them. That change enables the Entity Framework to include all columns in the SQL `WHERE` clause of `UPDATE` statements.

In the remainder of this tutorial you'll add a rowversion tracking property to the `Department` entity, create a controller and views, and test to verify that everything works correctly.

Add an Optimistic Concurrency Property to the Department Entity

In *Models\Department.cs*, add a tracking property named `RowVersion`:

```
public class Department
{
    public int DepartmentID { get; set; }

    [StringLength(50, MinimumLength = 3)]
    public string Name { get; set; }

    [DataType(DataType.Currency)]
    [Column(TypeName = "money")]
    public decimal Budget { get; set; }

    [DataType(DataType.Date)]
    public DateTime StartDate { get; set; }

    [Display(Name = "Administrator")]
    public int? InstructorID { get; set; }

    [Timestamp]
    public byte[] RowVersion { get; set; }

    public virtual Instructor Administrator { get; set; }
    public virtual ICollection<Course> Courses { get; set; }
}
```

The Timestamp attribute specifies that this column will be included in the `Where` clause of `Update` and `Delete` commands sent to the database. The attribute is called Timestamp because previous versions of SQL Server used a SQL timestamp data type before the SQL rowversion replaced it. The .Net type for `rowversion` is a byte array. If you prefer to use the fluent API, you can use the IsConcurrencyToken method to specify the tracking property, as shown in the following example:

```
modelBuilder.Entity<Department>()
    .Property(p => p.RowVersion).IsConcurrencyToken();
```

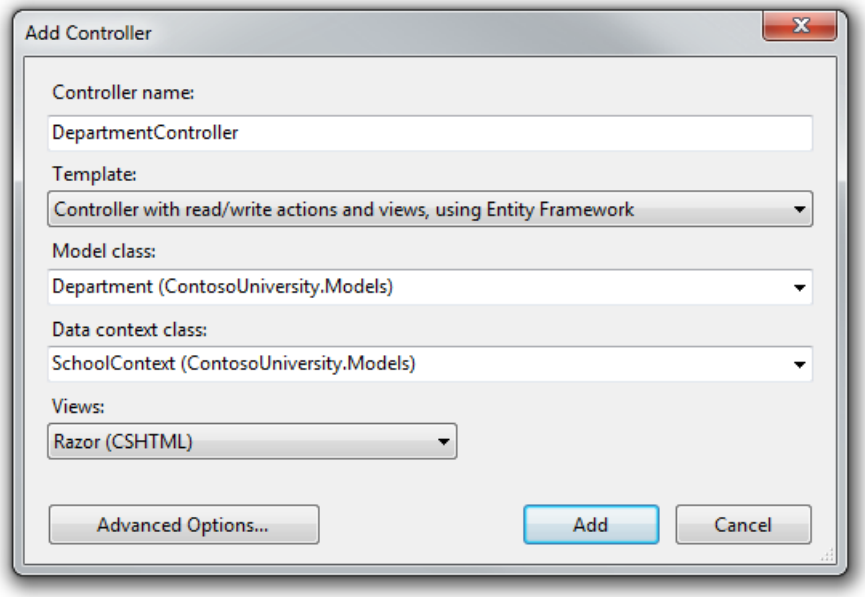
See the GitHub issue Replace IsConcurrencyToken by IsRowVersion.

By adding a property you changed the database model, so you need to do another migration. In the Package Manager Console (PMC), enter the following commands:

```
Add-Migration RowVersion
Update-Database
```

Create a Department Controller

Create a `Department` controller and views the same way you did the other controllers, using the following settings:



In *Controllers\DepartmentController.cs*, add a `using` statement:

```
using System.Data.Entity.Infrastructure;
```

Change "LastName" to "FullName" everywhere in this file (four occurrences) so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

```
ViewBag.InstructorID = new SelectList(db.Instructors, "InstructorID", "FullName");
```

Replace the existing code for the `HttpPost` `Edit` method with the following code:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(
   [Bind(Include = "DepartmentID, Name, Budget, StartDate, RowVersion, InstructorID")]
    Department department)
{
   try
   {
      if (ModelState.IsValid)
      {
         db.Entry(department).State = EntityState.Modified;
         db.SaveChanges();
         return RedirectToAction("Index");
      }
   }
   catch (DbUpdateConcurrencyException ex)
   {
      var entry = ex.Entries.Single();
      var clientValues = (Department)entry.Entity;
      var databaseValues = (Department)entry.GetDatabaseValues().ToObject();

      if (databaseValues.Name != clientValues.Name)
         ModelState.AddModelError("Name", "Current value: "
             + databaseValues.Name);
      if (databaseValues.Budget != clientValues.Budget)
         ModelState.AddModelError("Budget", "Current value: "
             + String.Format("{0:c}", databaseValues.Budget));
      if (databaseValues.StartDate != clientValues.StartDate)
         ModelState.AddModelError("StartDate", "Current value: "
             + String.Format("{0:d}", databaseValues.StartDate));
      if (databaseValues.InstructorID != clientValues.InstructorID)
         ModelState.AddModelError("InstructorID", "Current value: "
             + db.Instructors.Find(databaseValues.InstructorID).FullName);
      ModelState.AddModelError(string.Empty, "The record you attempted to edit "
          + "was modified by another user after you got the original value. The "
          + "edit operation was canceled and the current values in the database "
          + "have been displayed. If you still want to edit this record, click "
          + "the Save button again. Otherwise click the Back to List hyperlink.");
      department.RowVersion = databaseValues.RowVersion;
   }
   catch (DataException /* dex */)
   {
      //Log the error (uncomment dex variable name after DataException and add a line here to write a log.
      ModelState.AddModelError(string.Empty, "Unable to save changes. Try again, and if the problem persists
contact your system administrator.");
   }

   ViewBag.InstructorID = new SelectList(db.Instructors, "InstructorID", "FullName", department.InstructorID);
   return View(department);
}
```

The view will store the original `RowVersion` value in a hidden field. When the model binder creates the `department` instance, that object will have the original `RowVersion` property value and the new values for the other properties, as entered by the user on the Edit page. Then when the Entity Framework creates a SQL `UPDATE` command, that command will include a `WHERE` clause that looks for a row that has the original `RowVersion` value.

If no rows are affected by the `UPDATE` command (no rows have the original `RowVersion` value), the Entity Framework throws a `DbUpdateConcurrencyException` exception, and the code in the `catch` block gets the affected `Department` entity from the exception object. This entity has both the values read from the database and the new values entered by the user:

```
var entry = ex.Entries.Single();
var clientValues = (Department)entry.Entity;
var databaseValues = (Department)entry.GetDatabaseValues().ToObject();
```

Next, the code adds a custom error message for each column that has database values different from what the user entered on the Edit page:

```
if (databaseValues.Name != currentValues.Name)
    ModelState.AddModelError("Name", "Current value: " + databaseValues.Name);
    // ...
```

A longer error message explains what happened and what to do about it:

```
ModelState.AddModelError(string.Empty, "The record you attempted to edit "
    + "was modified by another user after you got the original value. The"
    + "edit operation was canceled and the current values in the database "
    + "have been displayed. If you still want to edit this record, click "
    + "the Save button again. Otherwise click the Back to List hyperlink.");
```

Finally, the code sets the `RowVersion` value of the `Department` object to the new value retrieved from the database. This new `RowVersion` value will be stored in the hidden field when the Edit page is redisplayed, and the next time the user clicks **Save**, only concurrency errors that happen since the redisplay of the Edit page will be caught.

In *Views\Department\Edit.cshtml*, add a hidden field to save the `RowVersion` property value, immediately following the hidden field for the `DepartmentID` property:

```
@model ContosoUniversity.Models.Department

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

@using (Html.BeginForm()) {
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>Department</legend>

        @Html.HiddenFor(model => model.DepartmentID)
        @Html.HiddenFor(model => model.RowVersion)

        <div class="editor-label">
            @Html.LabelFor(model => model.Name)
        </div>
```

In *Views\Department\Index.cshtml*, replace the existing code with the following code to move row links to the left and change the page title and column headings to display `FullName` instead of `LastName` in the **Administrator** column:

```
@model IEnumerable<ContosoUniversity.Models.Department>

@{
    ViewBag.Title = "Departments";
}

<h2>Departments</h2>

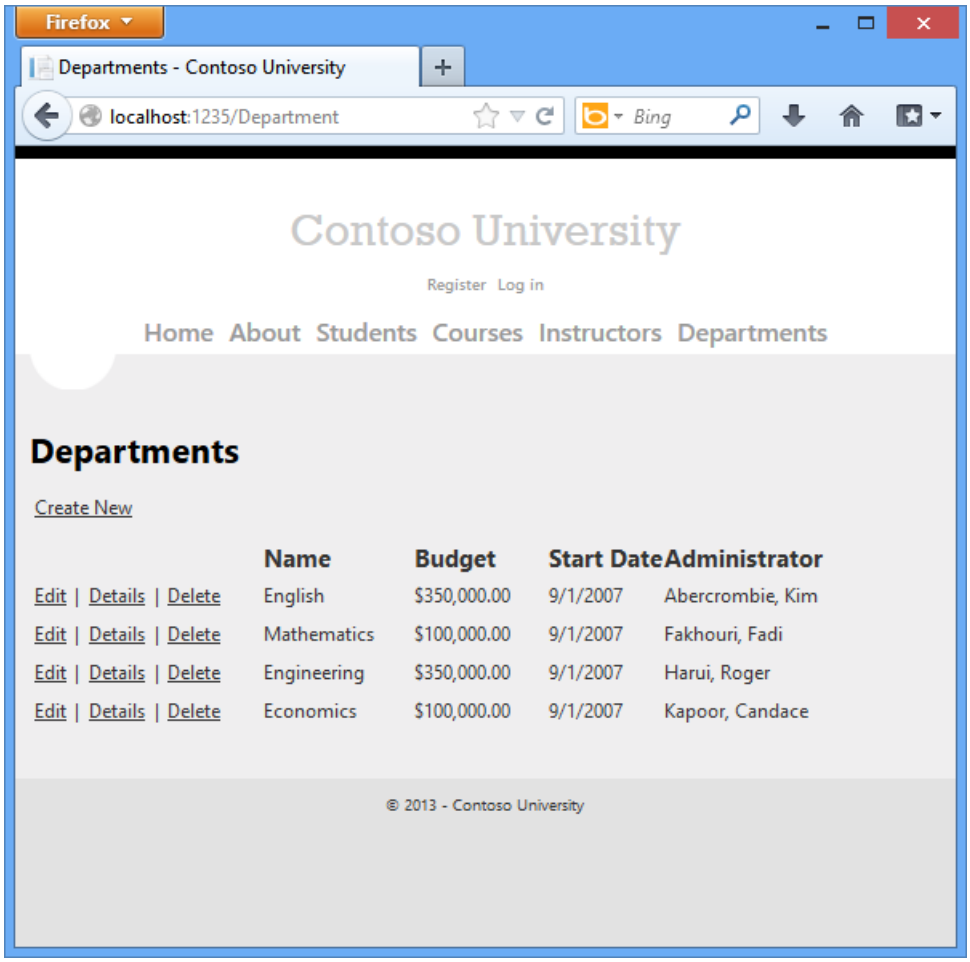
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th></th>
        <th>Name</th>
        <th>Budget</th>
        <th>Start Date</th>
        <th>Administrator</th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.DepartmentID }) |
            @Html.ActionLink("Details", "Details", new { id=item.DepartmentID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.DepartmentID })
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Budget)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.StartDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Administrator.FullName)
        </td>
    </tr>
}

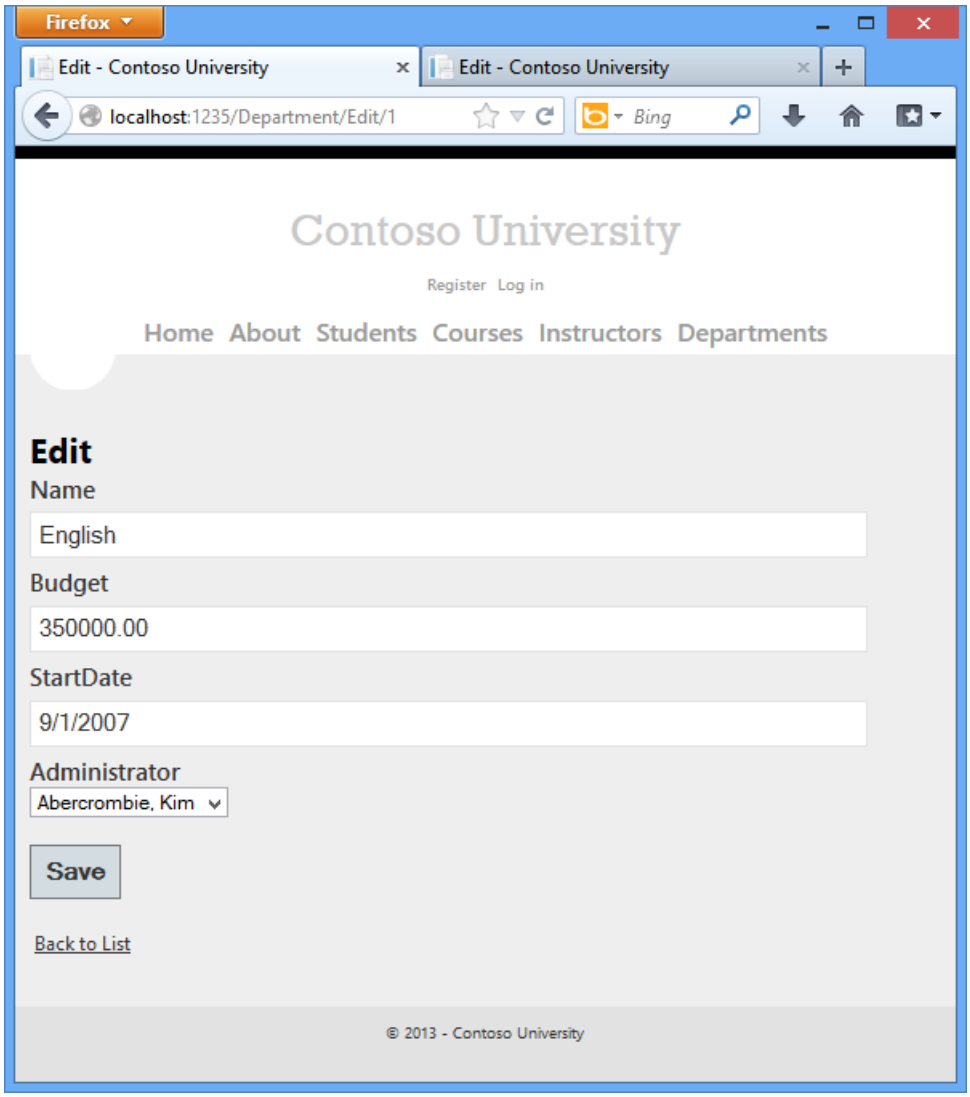
</table>
```

Testing Optimistic Concurrency Handling

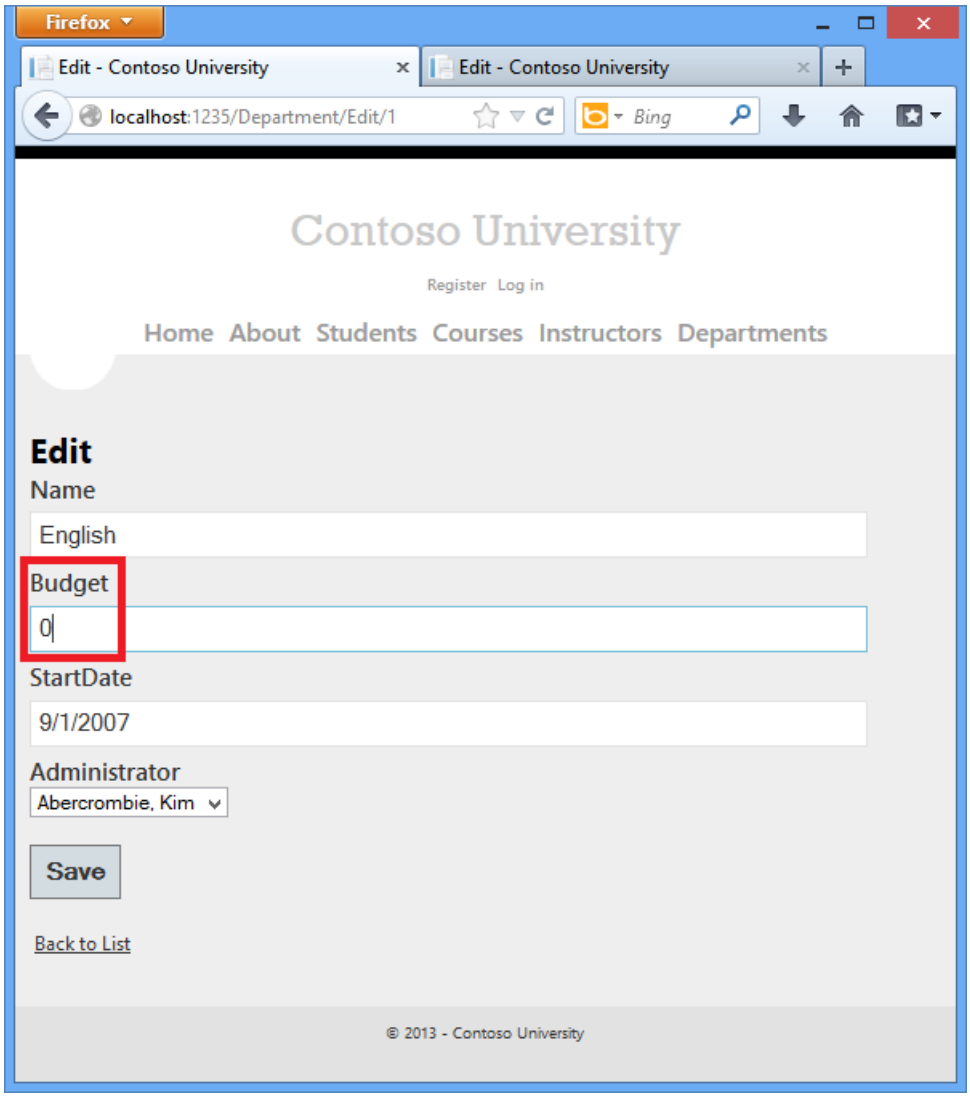
Run the site and click **Departments**:



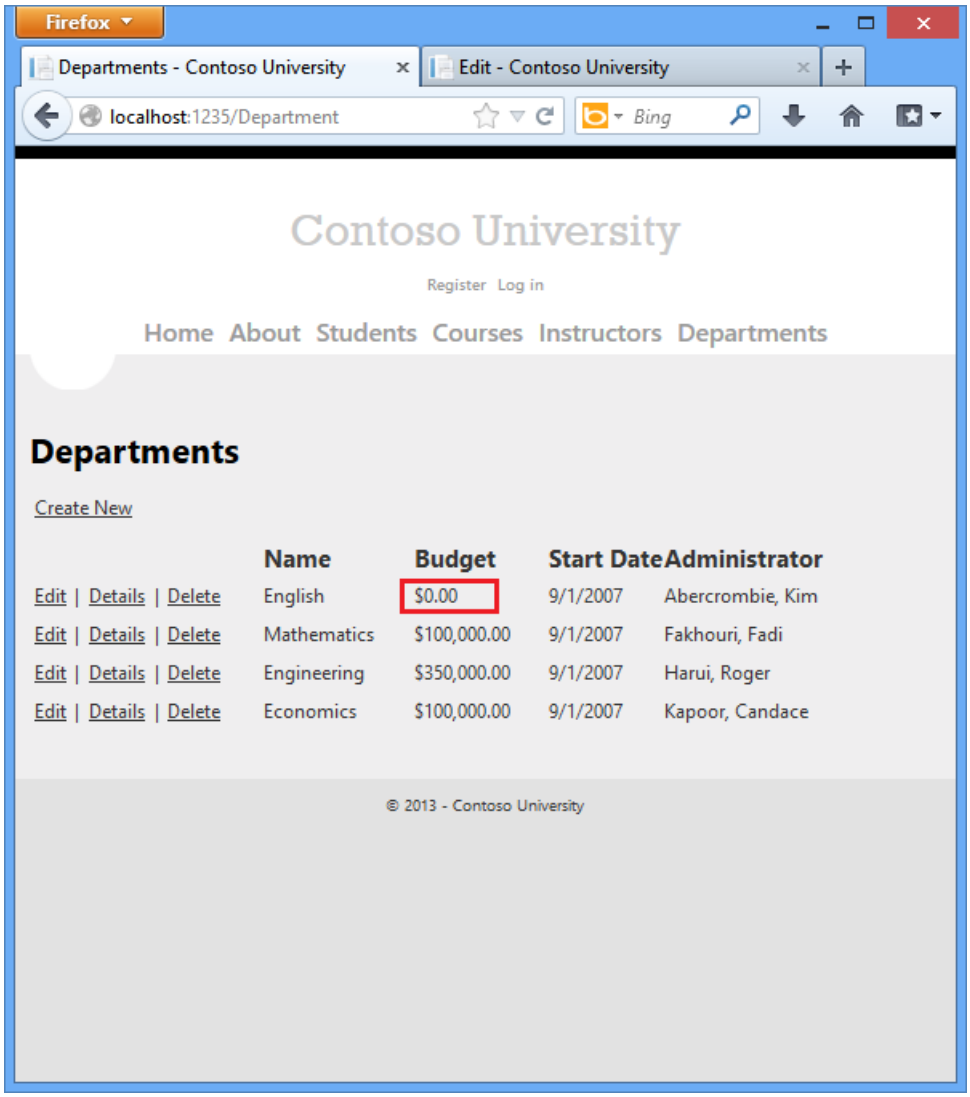
Right click the **Edit** hyperlink for Kim Abercrombie and select **Open in new tab**, then click the **Edit** hyperlink for Kim Abercrombie. The two windows display the same information.



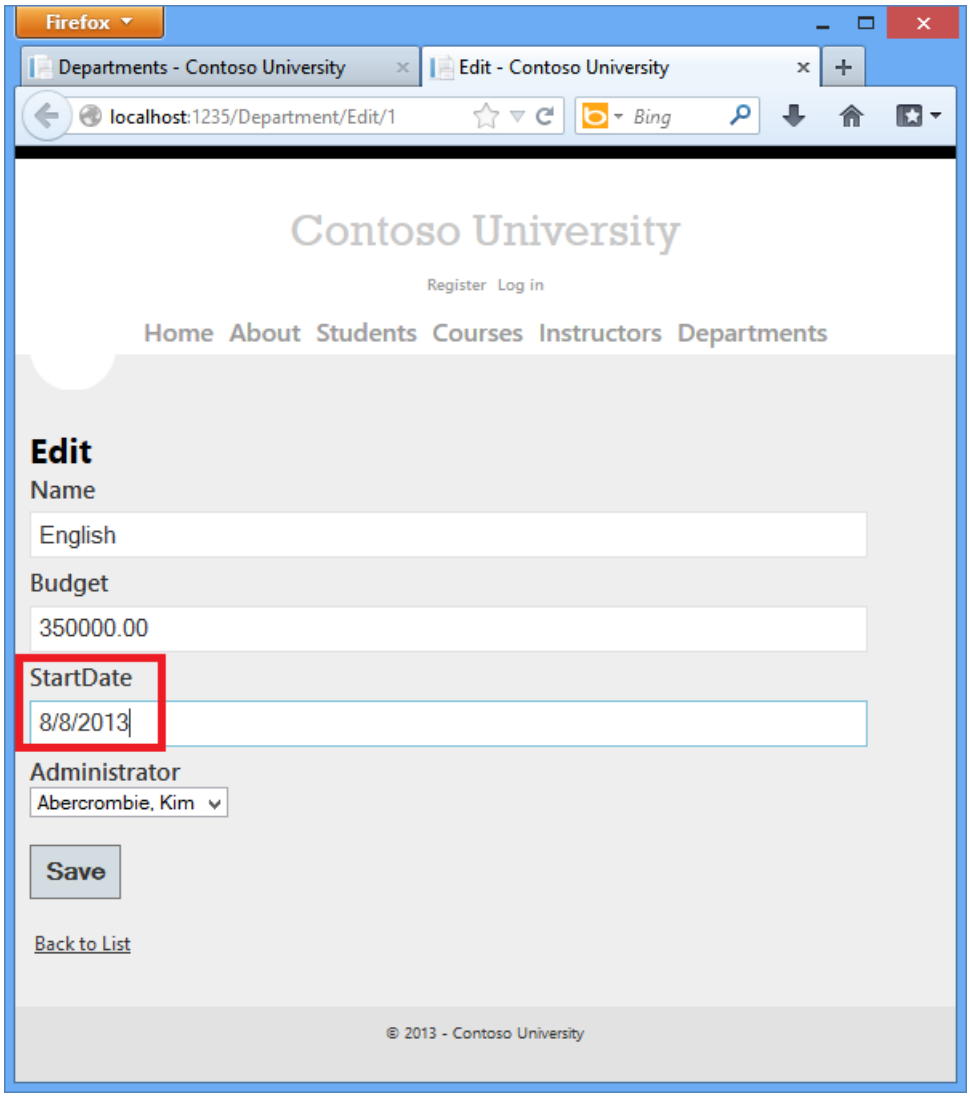
Change a field in the first browser window and click **Save**.



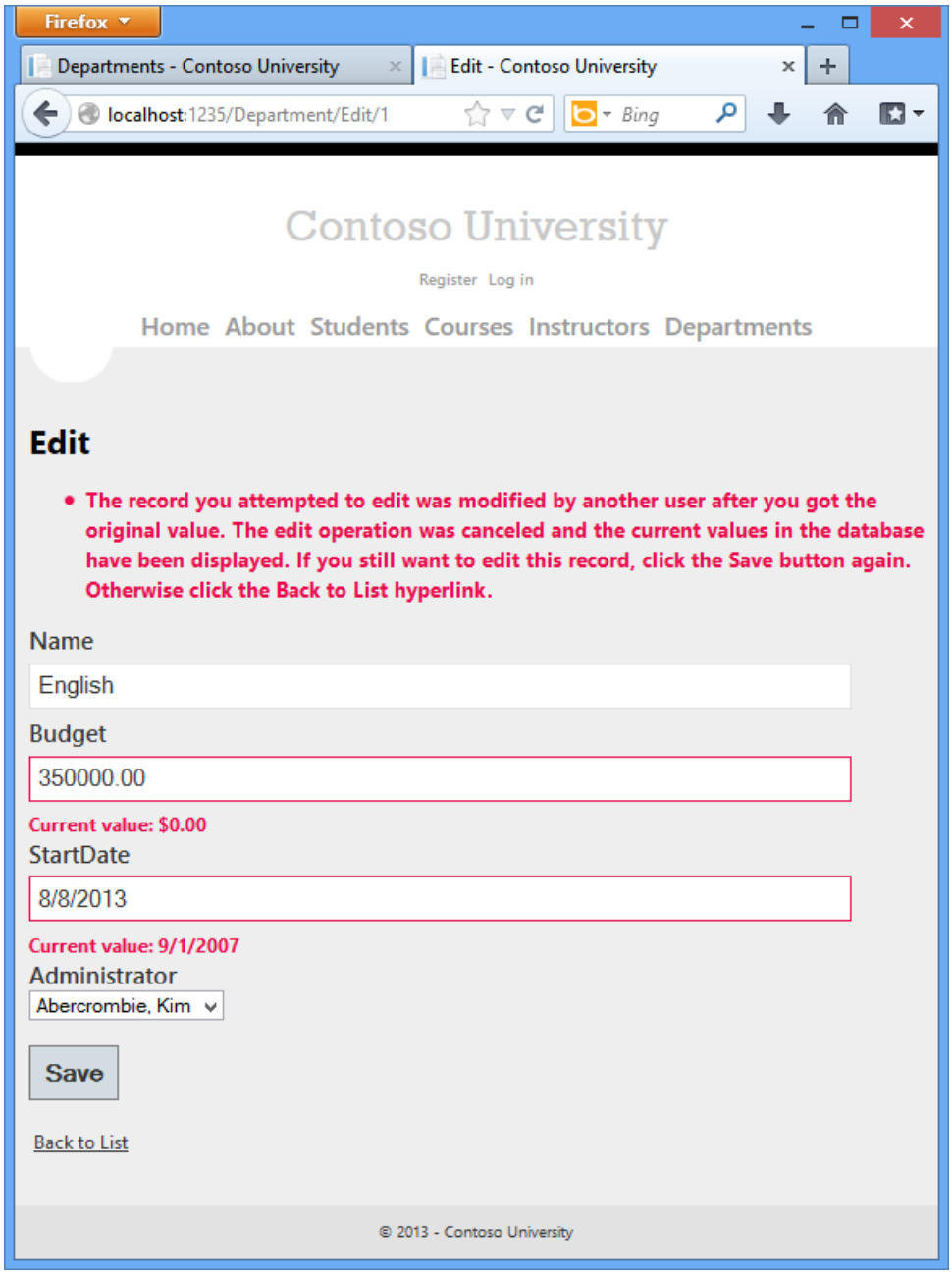
The browser shows the Index page with the changed value.



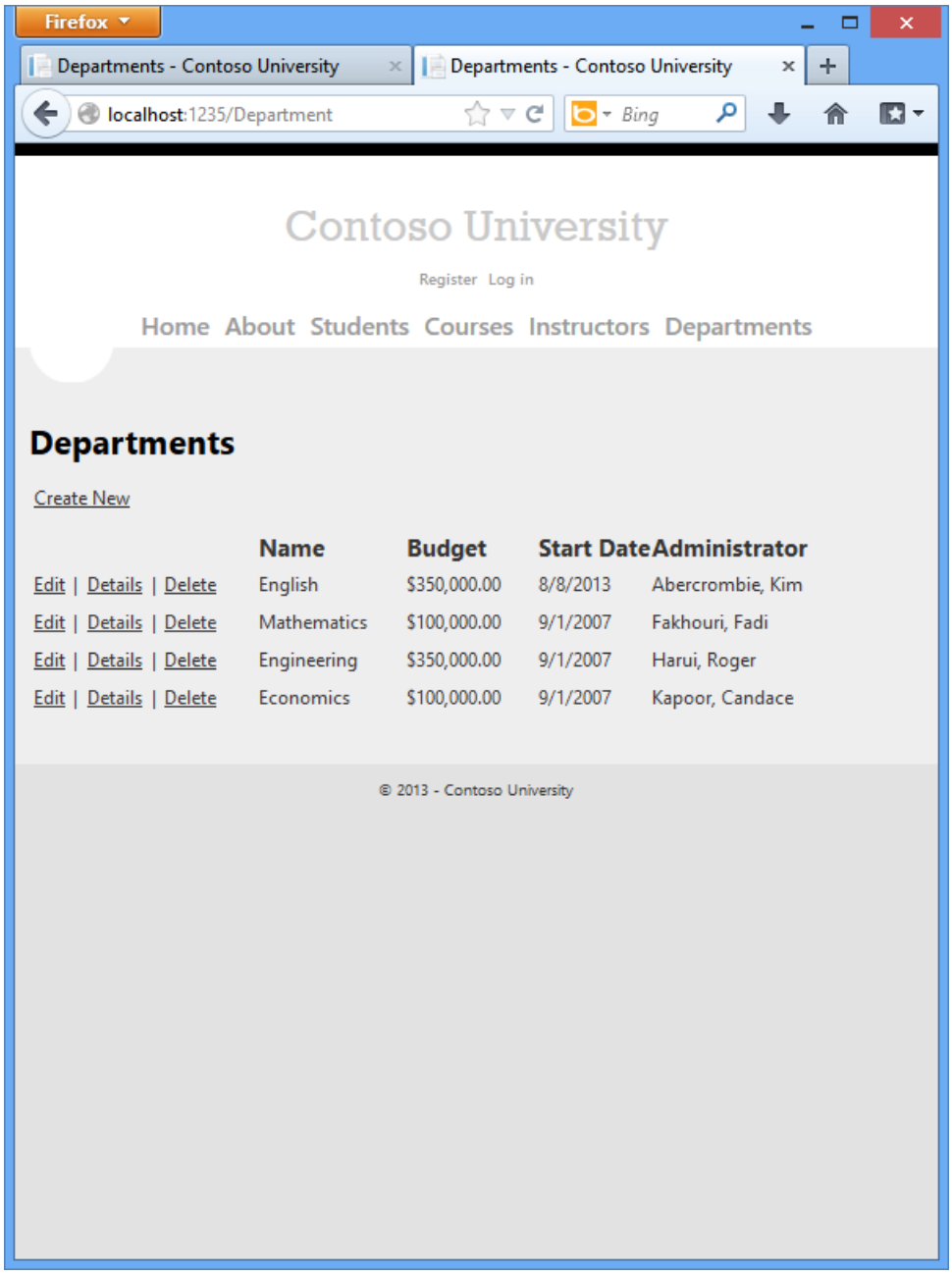
Change the any field in the second browser window and click **Save**.



Click **Save** in the second browser window. You see an error message:



Click **Save** again. The value you entered in the second browser is saved along with the original value of the data you change in the first browser. You see the saved values when the Index page appears.



Updating the Delete Page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the `HttpGet` `Delete` method displays the confirmation view, the view includes the original `RowVersion` value in a hidden field. That value is then available to the `HttpPost` `Delete` method that's called when the user confirms the deletion. When the Entity Framework creates the SQL `DELETE` command, it includes a `WHERE` clause with the original `RowVersion` value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the `HttpGet Delete` method is called with an error flag set to `true` in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case a different error message is displayed.

In *DepartmentController.cs*, replace the `HttpGet` `Delete` method with the following code:

```
public ActionResult Delete(int id, bool? concurrencyError)
{
    Department department = db.Departments.Find(id);

    if (concurrencyError.GetValueOrDefault())
    {
        if (department == null)
        {
            ViewBag.ConcurrencyErrorMessage = "The record you attempted to delete "
                + "was deleted by another user after you got the original values. "
                + "Click the Back to List hyperlink.";
        }
        else
        {
            ViewBag.ConcurrencyErrorMessage = "The record you attempted to delete "
                + "was modified by another user after you got the original values. "
                + "The delete operation was canceled and the current values in the "
                + "database have been displayed. If you still want to delete this "
                + "record, click the Delete button again. Otherwise "
                + "click the Back to List hyperlink.";
        }
    }

    return View(department);
}
```

The method accepts an optional parameter that indicates whether the page is being redisplayed after a concurrency error. If this flag is `true`, an error message is sent to the view using a `ViewBag` property.

Replace the code in the `HttpPost` `Delete` method (named `DeleteConfirmed`) with the following code:

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete(Department department)
{
    try
    {
        db.Entry(department).State = EntityState.Deleted;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    catch (DbUpdateConcurrencyException)
    {
        return RedirectToAction("Delete", new { concurrencyError=true } );
    }
    catch (DataException /* dex */)
    {
        //Log the error (uncomment dex variable name after DataException and add a line here to write a log.
        ModelState.AddModelError(string.Empty, "Unable to delete. Try again, and if the problem persists
contact your system administrator.");
        return View(department);
    }
}
```

In the scaffolded code that you just replaced, this method accepted only a record ID:

```
public ActionResult DeleteConfirmed(int id)
```

You've changed this parameter to a `Department` entity instance created by the model binder. This gives you access to the `RowVersion` property value in addition to the record key.

```
public ActionResult Delete(Department department)
```

You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code named the `HttpPost` `Delete` method `DeleteConfirmed` to give the `HttpPost` method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the `HttpPost` and `HttpGet` delete methods.

If a concurrency error is caught, the code redisplays the Delete confirmation page and provides a flag that indicates it should display a concurrency error message.

In *Views\Department\Delete.cshtml*, replace the scaffolded code with the following code that makes some formatting changes and adds an error message field. The changes are highlighted.

```
@model ContosoUniversity.Models.Department

@{
    ViewBag.Title = "Delete";
}

<h2>Delete</h2>

<p class="error">@ViewBag.ConcurrencyErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<fieldset>
    <legend>Department</legend>

    <div class="display-label">
         @Html.DisplayNameFor(model => model.Name)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Name)
    </div>

    <div class="display-label">
         @Html.DisplayNameFor(model => model.Budget)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Budget)
    </div>

    <div class="display-label">
         @Html.DisplayNameFor(model => model.StartDate)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.StartDate)
    </div>

    <div class="display-label">
         @Html.DisplayNameFor(model => model.Administrator.FullName)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Administrator.FullName)
    </div>
</fieldset>
@using (Html.BeginForm()) {
    @Html.AntiForgeryToken()
   @Html.HiddenFor(model => model.DepartmentID)
    @Html.HiddenFor(model => model.RowVersion)
    <p>
        <input type="submit" value="Delete" /> |
        @Html.ActionLink("Back to List", "Index")
    </p>
}
```

This code adds an error message between the `h2` and `h3` headings:

```
<p class="error">@ViewBag.ConcurrencyErrorMessage</p>
```

It replaces `LastName` with `FullName` in the `Administrator` field:

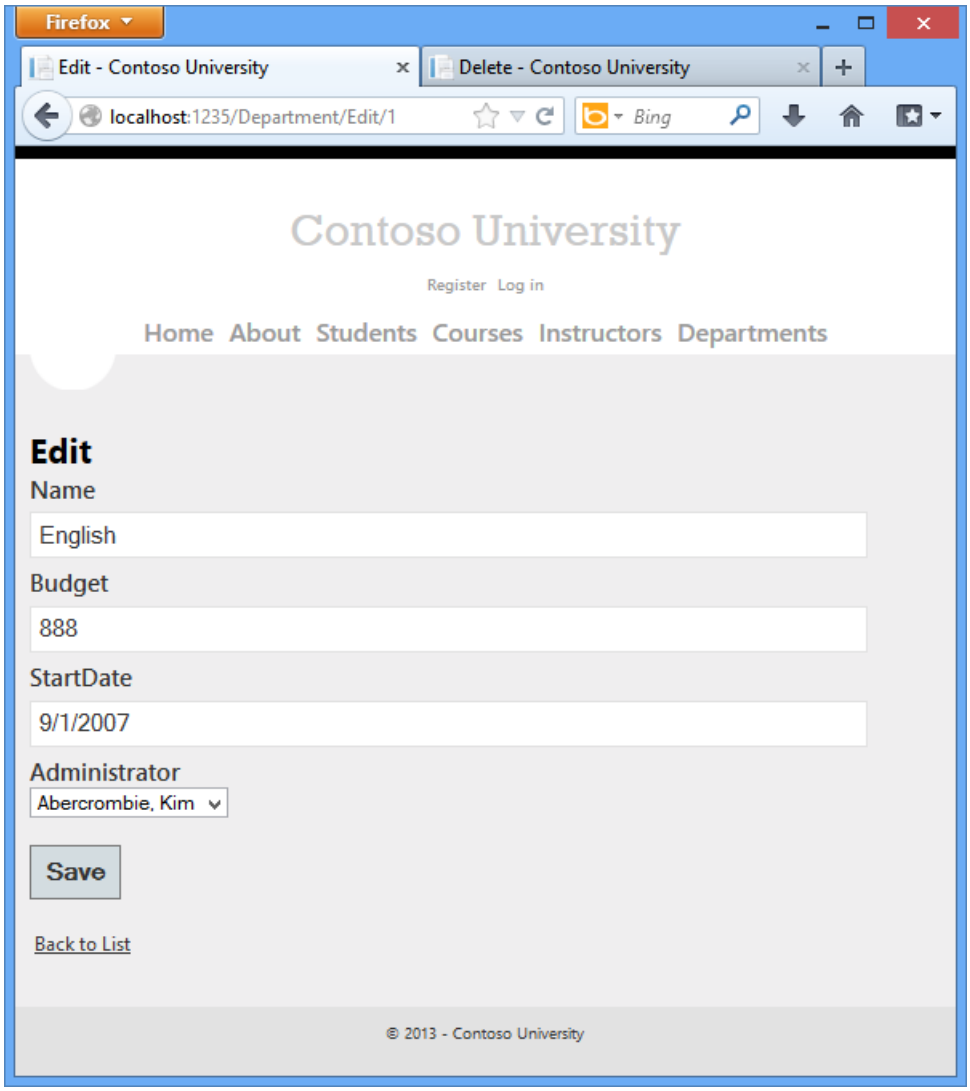
```
<div class="display-label">
    @Html.LabelFor(model => model.InstructorID)
</div>
<div class="display-field">
    @Html.DisplayFor(model => model.Administrator.FullName)
</div>
```

Finally, it adds hidden fields for the `DepartmentID` and `RowVersion` properties after the `Html.BeginForm` statement:

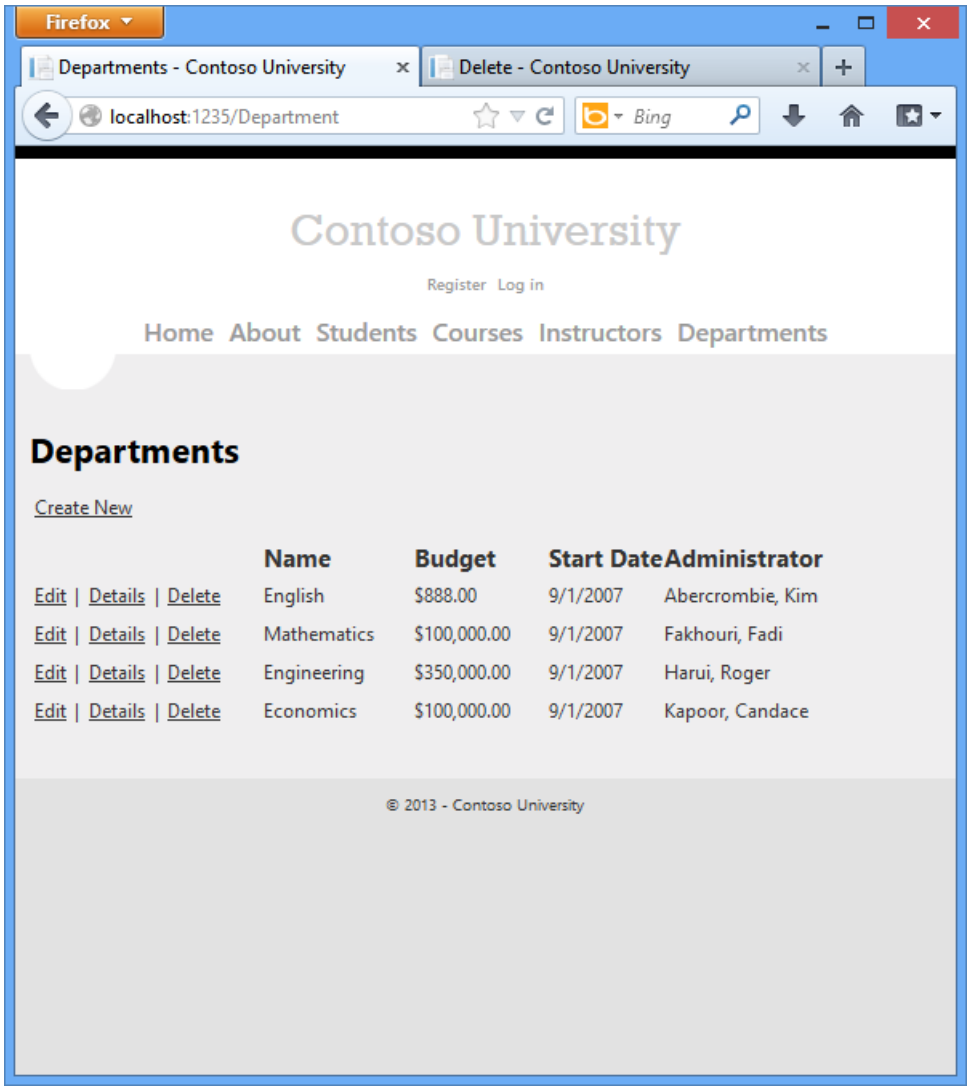
```
@Html.HiddenFor(model => model.DepartmentID)
@Html.HiddenFor(model => model.RowVersion)
```

Run the Departments Index page. Right click the **Delete** hyperlink for the English department and select **Open in new window,** then in the first window click the **Edit** hyperlink for the English department.

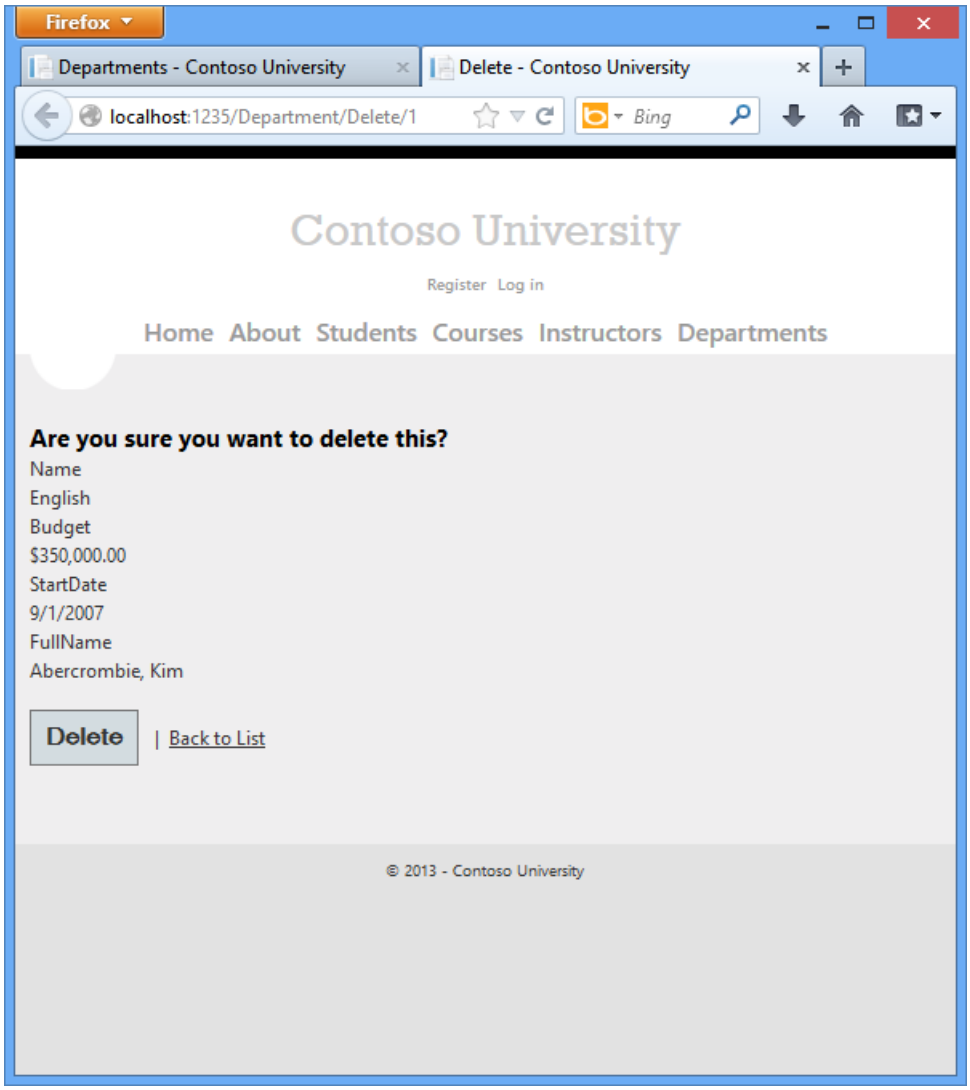
In the first window, change one of the values, and click **Save** :



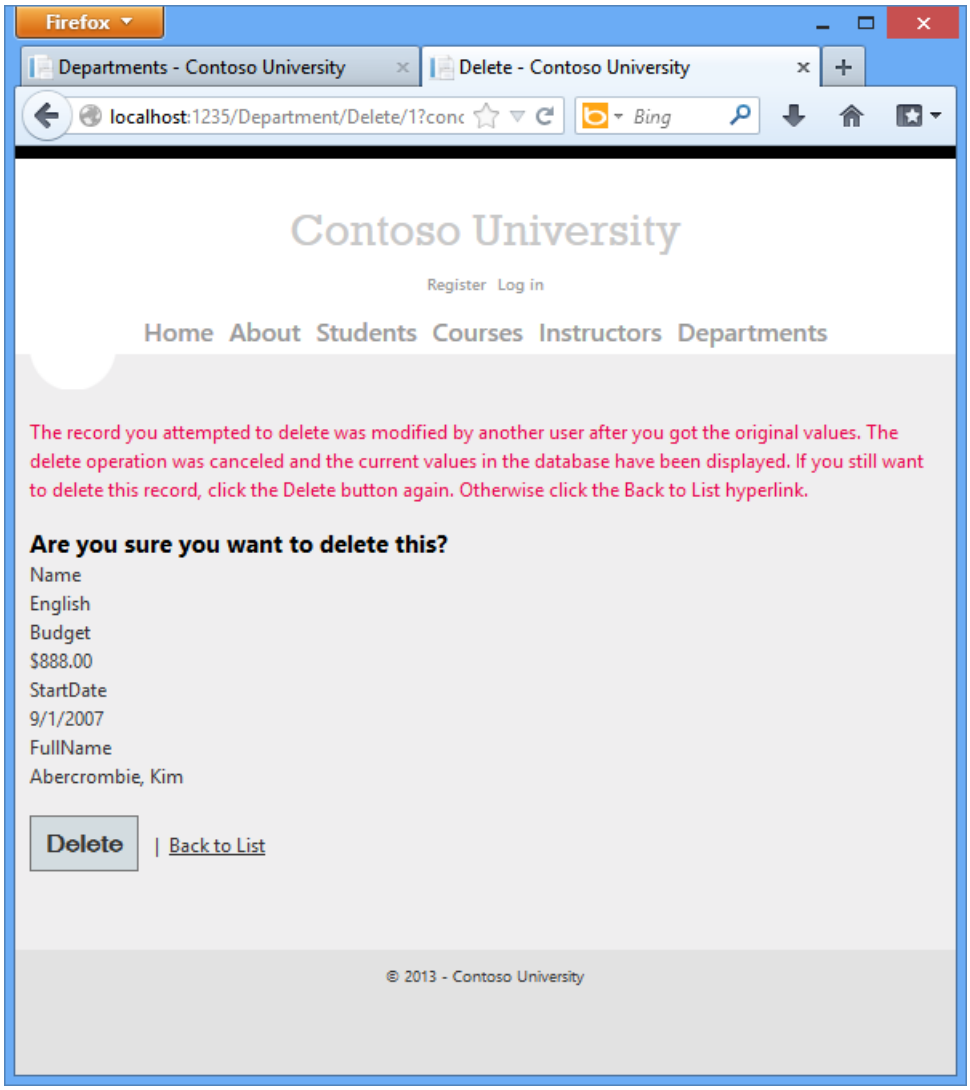
The Index page confirms the change.



In the second window, click **Delete**.



You see the concurrency error message, and the Department values are refreshed with what's currently in the database.



If you click **Delete** again, you're redirected to the Index page, which shows that the department has been deleted.

Summary

This completes the introduction to handling concurrency conflicts. For information about other ways to handle various concurrency scenarios, see Optimistic Concurrency Patterns and Working with Property Values on the Entity Framework team blog. The next tutorial shows how to implement table-per-hierarchy inheritance for the `Instructor` and `Student` entities.

Links to other Entity Framework resources can be found in the ASP.NET Data Access Content Map.

PREVIOUS NEXT

Implementing Inheritance with the Entity Framework in an ASP.NET MVC Application (8 of 10)

2/20/2020 • 9 minutes to read • Edit Online

by Tom Dykstra

Download Completed Project

The Contoso University sample web application demonstrates how to create ASP.NET MVC 4 applications using the Entity Framework 5 Code First and Visual Studio 2012. For information about the tutorial series, see the first tutorial in the series. You can start the tutorial series from the beginning or download a starter project for this chapter and start here.

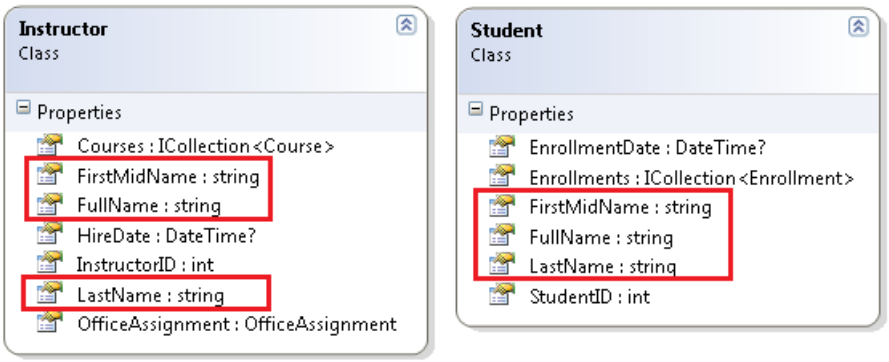
> **NOTE**
>
> If you run into a problem you can't resolve, download the completed chapter and try to reproduce your problem. You can generally find the solution to the problem by comparing your code to the completed code. For some common errors and how to solve them, see Errors and Workarounds.

In the previous tutorial you handled concurrency exceptions. This tutorial will show you how to implement inheritance in the data model.

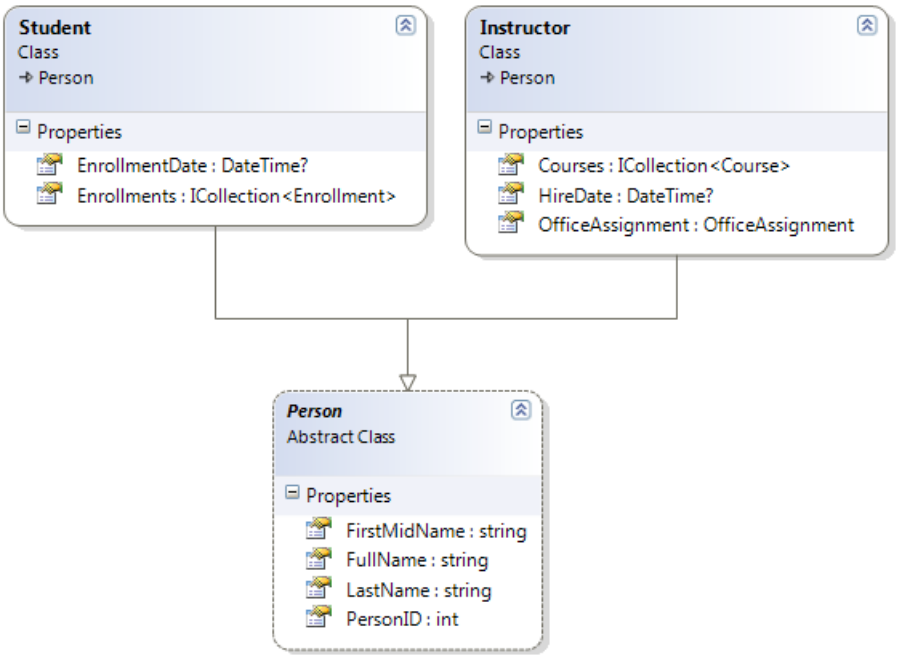
In object-oriented programming, you can use inheritance to eliminate redundant code. In this tutorial, you'll change the `Instructor` and `Student` classes so that they derive from a `Person` base class which contains properties such as `LastName` that are common to both instructors and students. You won't add or change any web pages, but you'll change some of the code and those changes will be automatically reflected in the database.

Table-per-Hierarchy versus Table-per-Type Inheritance

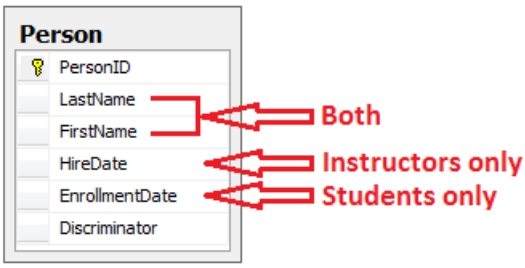
In object-oriented programming, you can use inheritance to make it easier to work with related classes. For example, the `Instructor` and `Student` classes in the `School` data model share several properties, which results in redundant code:



Suppose you want to eliminate the redundant code for the properties that are shared by the `Instructor` and `Student` entities. You could create a `Person` base class which contains only those shared properties, then make the `Instructor` and `Student` entities inherit from that base class, as shown in the following illustration:

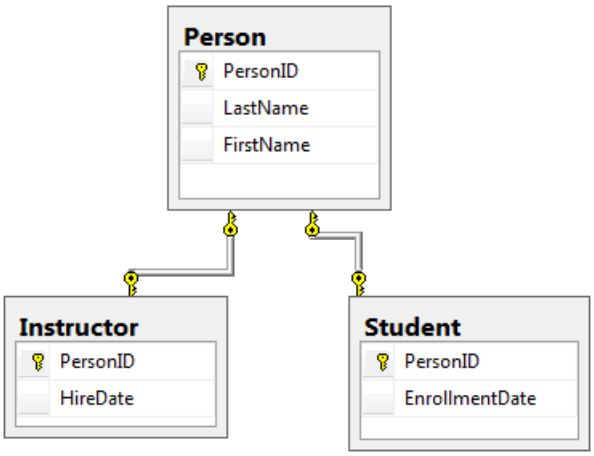


There are several ways this inheritance structure could be represented in the database. You could have a `Person` table that includes information about both students and instructors in a single table. Some of the columns could apply only to instructors (`HireDate`), some only to students (`EnrollmentDate`), some to both (`LastName`, `FirstName`). Typically, you'd have a *discriminator* column to indicate which type each row represents. For example, the discriminator column might have "Instructor" for instructors and "Student" for students.



This pattern of generating an entity inheritance structure from a single database table is called *table-per-hierarchy* (TPH) inheritance.

An alternative is to make the database look more like the inheritance structure. For example, you could have only the name fields in the `Person` table and have separate `Instructor` and `Student` tables with the date fields.



This pattern of making a database table for each entity class is called *table per type* (TPT) inheritance.

TPH inheritance patterns generally deliver better performance in the Entity Framework than TPT inheritance

patterns, because TPT patterns can result in complex join queries. This tutorial demonstrates how to implement TPH inheritance. You'll do that by performing the following steps:

- Create a `Person` class and change the `Instructor` and `Student` classes to derive from `Person`.
- Add model-to-database mapping code to the database context class.
- Change `InstructorID` and `StudentID` references throughout the project to `PersonID`.

Creating the Person Class

Note: You won't be able to compile the project after creating the classes below until you update the controllers that uses these classes.

In the *Models* folder, create *Person.cs* and replace the template code with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
   public abstract class Person
   {
      [Key]
      public int PersonID { get; set; }

      [RegularExpression(@"^[A-Z]+[a-zA-Z""'\s-]*$")]
      [StringLength(50, MinimumLength = 1)]
      [Display(Name = "Last Name")]
      public string LastName { get; set; }

      [Column("FirstName")]
      [Display(Name = "First Name")]
      [StringLength(50, MinimumLength = 2, ErrorMessage = "First name must be between 2 and 50 characters.")]
      public string FirstMidName { get; set; }

      public string FullName
      {
         get
         {
            return LastName + ", " + FirstMidName;
         }
      }
   }
}
```

In *Instructor.cs*, derive the `Instructor` class from the `Person` class and remove the key and name fields. The code will look like the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Instructor : Person
    {
        [DataType(DataType.Date)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        public virtual ICollection<Course> Courses { get; set; }
        public virtual OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Make similar changes to *Student.cs*. The `Student` class will look like the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student : Person
    {
        [DataType(DataType.Date)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }

        public virtual ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

Adding the Person Entity Type to the Model

In *SchoolContext.cs*, add a `DbSet` property for the `Person` entity type:

```
public DbSet<Person> People { get; set; }
```

This is all that the Entity Framework needs in order to configure table-per-hierarchy inheritance. As you'll see, when the database is re-created, it will have a `Person` table in place of the `Student` and `Instructor` tables.

Changing InstructorID and StudentID to PersonID

In *SchoolContext.cs*, in the Instructor-Course mapping statement, change `MapRightKey("InstructorID")` to `MapRightKey("PersonID")`:

```
modelBuilder.Entity<Course>()
    .HasMany(c => c.Instructors).WithMany(i => i.Courses)
    .Map(t => t.MapLeftKey("CourseID")
    .MapRightKey("PersonID")
    .ToTable("CourseInstructor"));
```

This change isn't required; it just changes the name of the InstructorID column in the many-to-many join table. If you left the name as InstructorID, the application would still work correctly. Here is the completed

SchoolContext.cs:

```
using ContosoUniversity.Models;
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace ContosoUniversity.DAL
{
   public class SchoolContext : DbContext
   {
      public DbSet<Course> Courses { get; set; }
      public DbSet<Department> Departments { get; set; }
      public DbSet<Enrollment> Enrollments { get; set; }
      public DbSet<Instructor> Instructors { get; set; }
      public DbSet<Student> Students { get; set; }
      public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
      public DbSet<Person> People { get; set; }

      protected override void OnModelCreating(DbModelBuilder modelBuilder)
      {
         modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

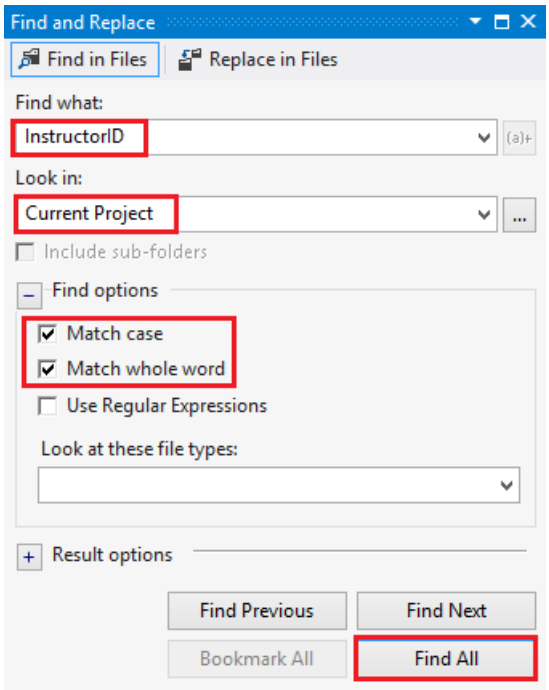
         modelBuilder.Entity<Course>()
             .HasMany(c => c.Instructors).WithMany(i => i.Courses)
             .Map(t => t.MapLeftKey("CourseID")
                 .MapRightKey("PersonID")
                 .ToTable("CourseInstructor"));
      }
   }
}
```

Next you need to change `InstructorID` to `PersonID` and `StudentID` to `PersonID` throughout the project ***except*** in the time-stamped migrations files in the *Migrations* folder. To do that you'll find and open only the files that need to be changed, then perform a global change on the opened files. The only file in the *Migrations* folder you should change is *Migrations\Configuration.cs.*

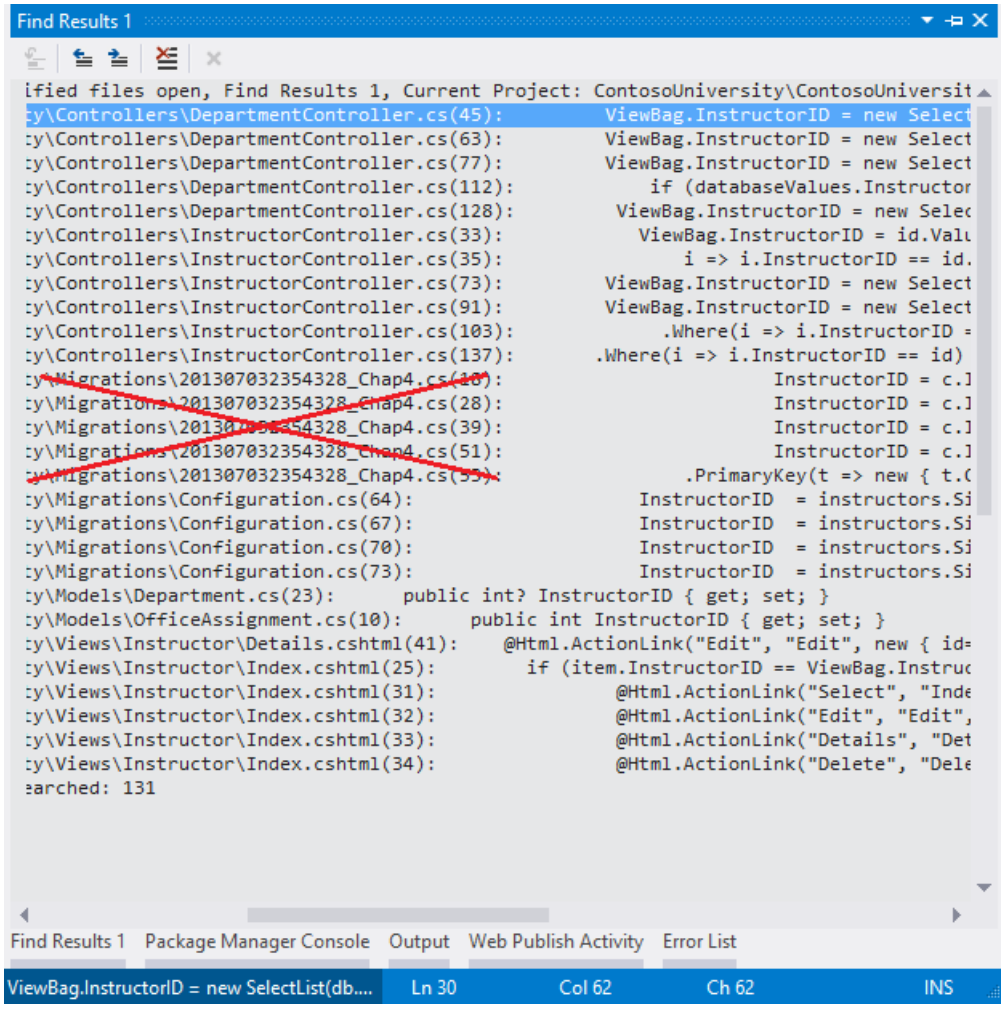
1. **IMPORTANT**

   Begin by closing all the open files in Visual Studio.

2. Click **Find and Replace -- Find all Files** in the **Edit** menu, and then search for all files in the project that contain `InstructorID`.

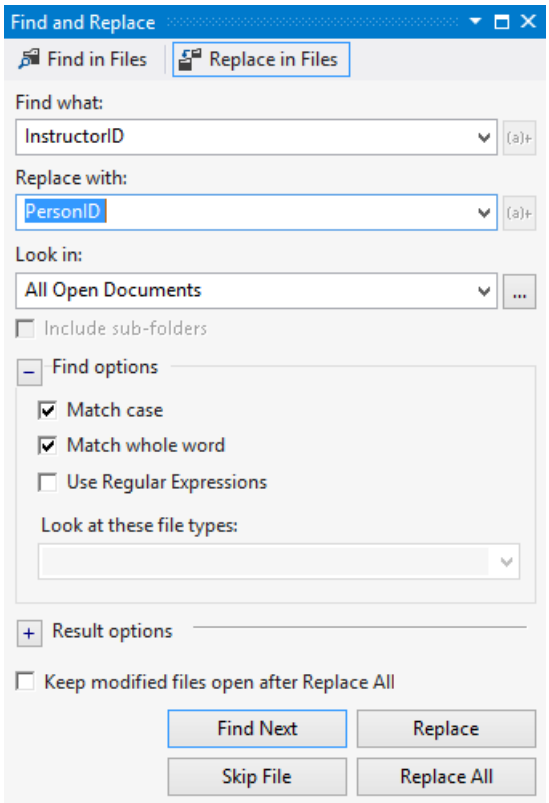


3. Open each file in the **Find Results** window ***except*** the <time-stamp>*_.cs* migration files in the *Migrations* folder, by double-clicking one line for each file.



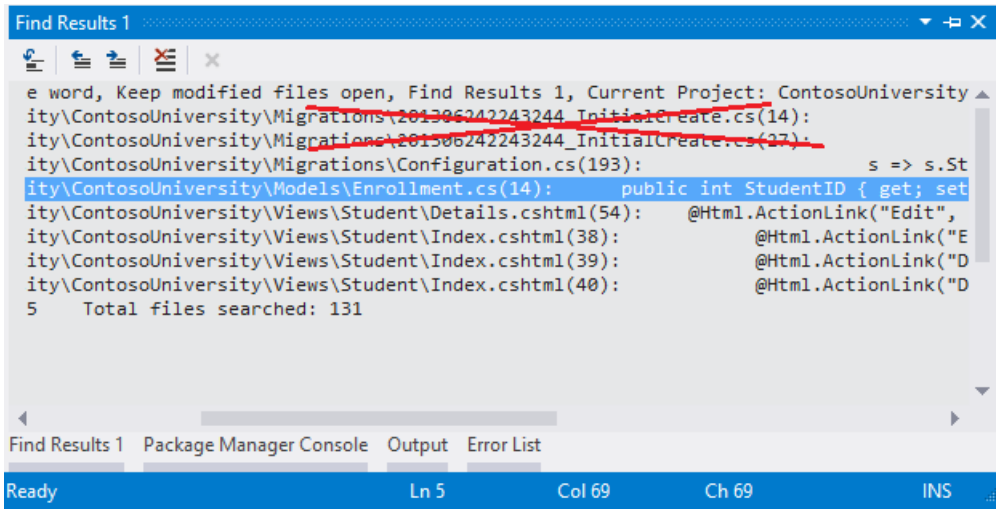
4. Open the **Replace in Files** dialog and change **Look in** to **All Open Documents**.

5. Use the **Replace in Files** dialog to change all `InstructorID` to `PersonID.`



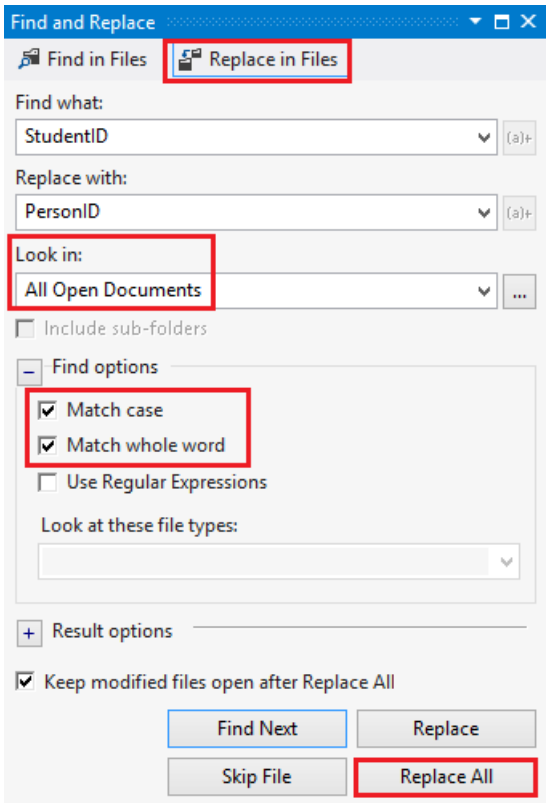
6. Find all the files in the project that contain `StudentID`.

7. Open each file in the **Find Results** window ***except*** the <time-stamp>_*.cs* migration files in the *Migrations* folder, by double-clicking one line for each file.



8. Open the **Replace in Files** dialog and change **Look in** to **All Open Documents**.

9. Use the **Replace in Files** dialog to change all `StudentID` to `PersonID`.



10. Build the project.

(Note that this demonstrates a *disadvantage* of the `classnameID` pattern for naming primary keys. If you had named primary keys ID without prefixing the class name, *no* renaming would be necessary now.)

Create and Update a Migrations File

In the Package Manager Console (PMC), enter the following command:

```
Add-Migration Inheritance
```

Run the `Update-Database` command in the PMC. The command will fail at this point because we have existing data that migrations doesn't know how to handle. You get the following error:

The ALTER TABLE statement conflicted with the FOREIGN KEY constraint "FK_dbo.Department_dbo.Person_PersonID". The conflict occurred in database "ContosoUniversity", table "dbo.Person", column 'PersonID'.

Open *Migrations<timestamp>_Inheritance.cs* and replace the `Up` method with the following code:

```
public override void Up()
{
    DropForeignKey("dbo.Department", "InstructorID", "dbo.Instructor");
    DropForeignKey("dbo.OfficeAssignment", "InstructorID", "dbo.Instructor");
    DropForeignKey("dbo.Enrollment", "StudentID", "dbo.Student");
    DropForeignKey("dbo.CourseInstructor", "InstructorID", "dbo.Instructor");
    DropIndex("dbo.Department", new[] { "InstructorID" });
    DropIndex("dbo.OfficeAssignment", new[] { "InstructorID" });
    DropIndex("dbo.Enrollment", new[] { "StudentID" });
    DropIndex("dbo.CourseInstructor", new[] { "InstructorID" });
    RenameColumn(table: "dbo.Department", name: "InstructorID", newName: "PersonID");
    RenameColumn(table: "dbo.OfficeAssignment", name: "InstructorID", newName: "PersonID");
    RenameColumn(table: "dbo.Enrollment", name: "StudentID", newName: "PersonID");
    RenameColumn(table: "dbo.CourseInstructor", name: "InstructorID", newName: "PersonID");
    CreateTable(
        "dbo.Person",
        c => new
            {
                PersonID = c.Int(nullable: false, identity: true),
                LastName = c.String(maxLength: 50),
                FirstName = c.String(maxLength: 50),
                HireDate = c.DateTime(),
                EnrollmentDate = c.DateTime(),
                Discriminator = c.String(nullable: false, maxLength: 128),
                OldId = c.Int(nullable: false)
            })
        .PrimaryKey(t => t.PersonID);

    // Copy existing Student and Instructor data into new Person table.
    Sql("INSERT INTO dbo.Person (LastName, FirstName, HireDate, EnrollmentDate, Discriminator, OldId) SELECT
LastName, FirstName, null AS HireDate, EnrollmentDate, 'Student' AS Discriminator, StudentId AS OldId FROM
dbo.Student");
    Sql("INSERT INTO dbo.Person (LastName, FirstName, HireDate, EnrollmentDate, Discriminator, OldId) SELECT
LastName, FirstName, HireDate, null AS EnrollmentDate, 'Instructor' AS Discriminator, InstructorId AS OldId
FROM dbo.Instructor");

    // Fix up existing relationships to match new PK's.
    Sql("UPDATE dbo.Enrollment SET PersonId = (SELECT PersonId FROM dbo.Person WHERE OldId =
Enrollment.PersonId AND Discriminator = 'Student')");
    Sql("UPDATE dbo.Department SET PersonId = (SELECT PersonId FROM dbo.Person WHERE OldId =
Department.PersonId AND Discriminator = 'Instructor')");
    Sql("UPDATE dbo.OfficeAssignment SET PersonId = (SELECT PersonId FROM dbo.Person WHERE OldId =
OfficeAssignment.PersonId AND Discriminator = 'Instructor')");
    Sql("UPDATE dbo.CourseInstructor SET PersonId = (SELECT PersonId FROM dbo.Person WHERE OldId =
CourseInstructor.PersonId AND Discriminator = 'Instructor')");

    // Remove temporary key
    DropColumn("dbo.Person", "OldId");

    AddForeignKey("dbo.Department", "PersonID", "dbo.Person", "PersonID");
    AddForeignKey("dbo.OfficeAssignment", "PersonID", "dbo.Person", "PersonID");
    AddForeignKey("dbo.Enrollment", "PersonID", "dbo.Person", "PersonID", cascadeDelete: true);
    AddForeignKey("dbo.CourseInstructor", "PersonID", "dbo.Person", "PersonID", cascadeDelete: true);
    CreateIndex("dbo.Department", "PersonID");
    CreateIndex("dbo.OfficeAssignment", "PersonID");
    CreateIndex("dbo.Enrollment", "PersonID");
    CreateIndex("dbo.CourseInstructor", "PersonID");
    DropTable("dbo.Instructor");
    DropTable("dbo.Student");
}
```

Run the `update-database` command again.

NOTE

It's possible to get other errors when migrating data and making schema changes. If you get migration errors you can't resolve, you can continue with the tutorial by changing the connection string in the *Web.config* file or deleting the database. The simplest approach is to rename the database in the *Web.config* file. For example, change the database name to CU_test as shown in the following example:

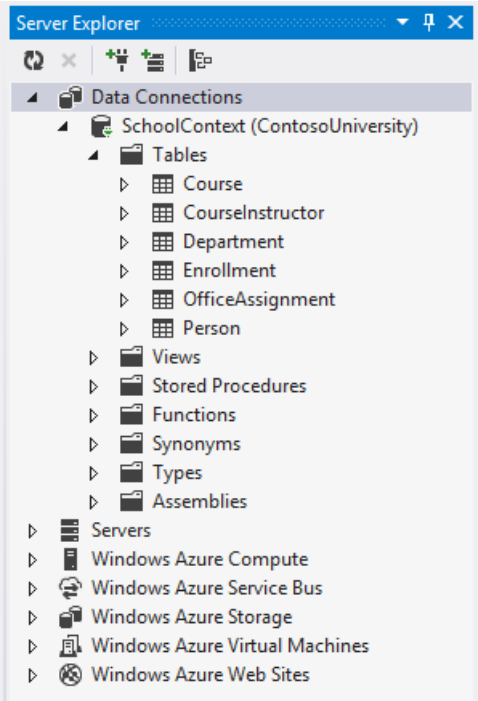
```
<add name="SchoolContext" connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=CU_Test;
      Integrated Security=SSPI;AttachDBFilename=|DataDirectory|\CU_Test.mdf"
      providerName="System.Data.SqlClient" />
```

With a new database, there is no data to migrate, and the `update-database` command is much more likely to complete without errors. For instructions on how to delete the database, see How to Drop a Database from Visual Studio 2012. If you take this approach in order to continue with the tutorial, skip the deployment step at the end of this tutorial, since the deployed site would get the same error when it runs migrations automatically. If you want to troubleshoot a migrations error, the best resource is one of the Entity Framework forums or StackOverflow.com.

Testing

Run the site and try various pages. Everything works the same as it did before.

In **Server Explorer,** expand **SchoolContext** and then **Tables**, and you see that the **Student** and **Instructor** tables have been replaced by a **Person** table. Expand the **Person** table and you see that it has all of the columns that used to be in the **Student** and **Instructor** tables.



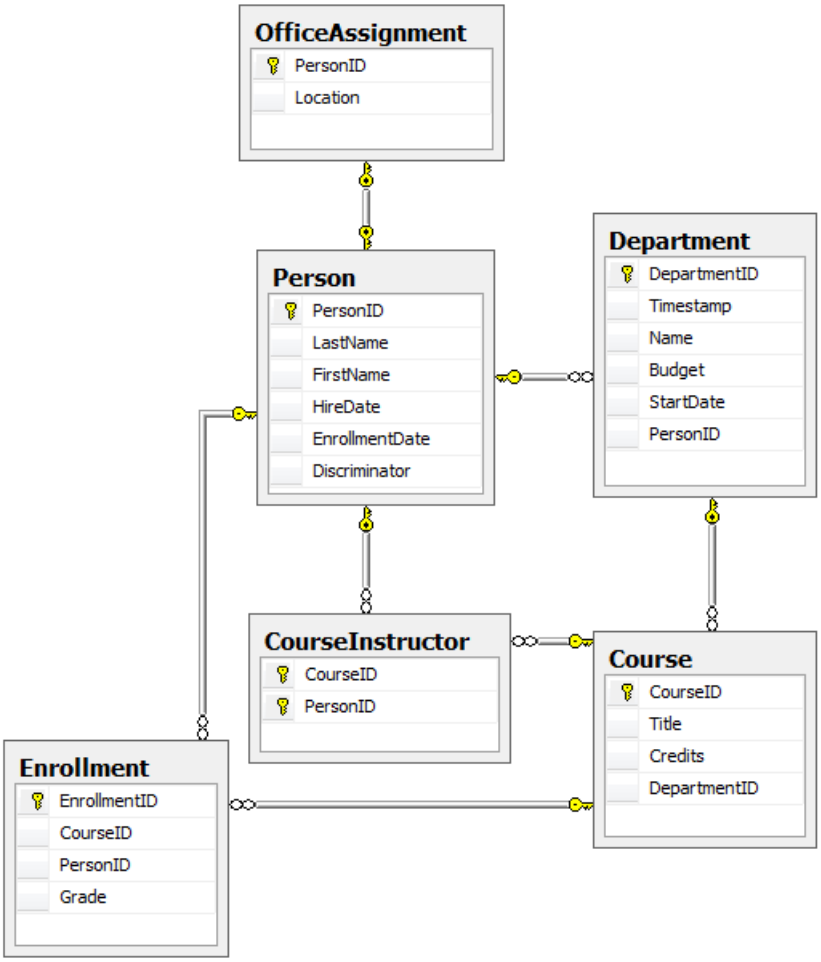
Right-click the Person table, and then click **Show Table Data** to see the discriminator column.

dbo.Person [Data]

Max Rows: 1000

	PersonID	LastName	FirstName	HireDate	EnrollmentDate	Discriminator
▶	1	Alexander	Carson	NULL	9/1/2010 12:00...	Student
	2	Alonso	Meredith	NULL	9/1/2012 12:00...	Student
	3	Anand	Arturo	NULL	9/1/2013 12:00...	Student
	4	Barzdukas	Gytis	NULL	9/1/2012 12:00...	Student
	5	Li	Yan	NULL	9/1/2012 12:00...	Student
	6	Justice	Peggy	NULL	9/1/2011 12:00...	Student
	7	Norman	Laura	NULL	9/1/2013 12:00...	Student
	8	Olivetto	Nino	NULL	9/1/2005 12:00...	Student
	9	Abercrombie	Kim	3/11/1995 1...	NULL	Instructor
	10	Fakhouri	Fadi	7/6/2002 12:...	NULL	Instructor
	11	Harui	Roger	7/1/1998 12:...	NULL	Instructor
	12	Kapoor	Candace	1/15/2001 1...	NULL	Instructor
	13	Zheng	Roger	2/12/2004 1...	NULL	Instructor
	14	Smith	Joe	NULL	5/23/2013 12:0...	Student
*	NULL	NULL	NULL	NULL	NULL	NULL

The following diagram illustrates the structure of the new School database:



Summary

Table-per-hierarchy inheritance has now been implemented for the `Person`, `Student`, and `Instructor` classes. For more information about this and other inheritance structures, see Inheritance Mapping Strategies on Morteza Manavi's blog. In the next tutorial you'll see some ways to implement the repository and unit of work patterns.

Links to other Entity Framework resources can be found in the ASP.NET Data Access Content Map.

Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10)

2/20/2020 • 20 minutes to read • Edit Online

by Tom Dykstra

Download Completed Project

The Contoso University sample web application demonstrates how to create ASP.NET MVC 4 applications using the Entity Framework 5 Code First and Visual Studio 2012. For information about the tutorial series, see the first tutorial in the series. You can start the tutorial series from the beginning or download a starter project for this chapter and start here.

> **NOTE**
>
> If you run into a problem you can't resolve, download the completed chapter and try to reproduce your problem. You can generally find the solution to the problem by comparing your code to the completed code. For some common errors and how to solve them, see Errors and Workarounds.

In the previous tutorial you used inheritance to reduce redundant code in the `Student` and `Instructor` entity classes. In this tutorial you'll see some ways to use the repository and unit of work patterns for CRUD operations. As in the previous tutorial, in this one you'll change the way your code works with pages you already created rather than creating new pages.

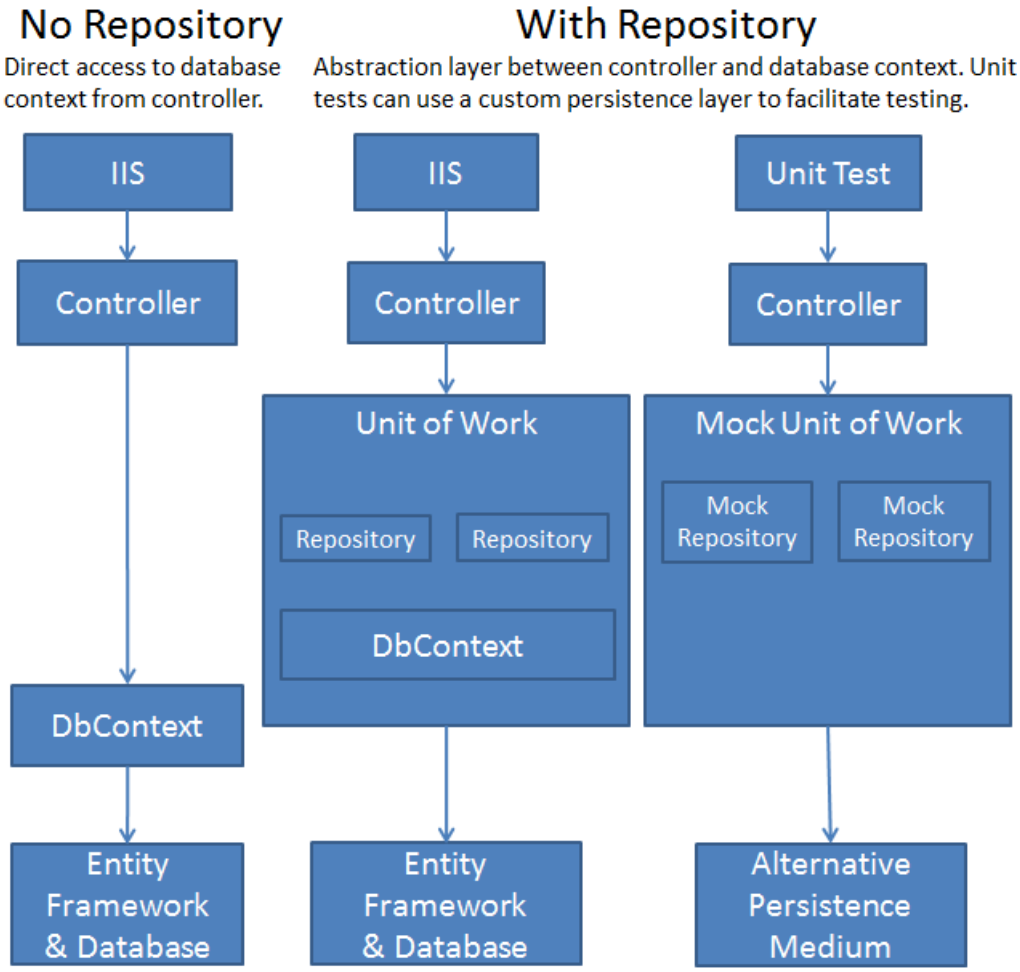
The Repository and Unit of Work Patterns

The repository and unit of work patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD).

In this tutorial you'll implement a repository class for each entity type. For the `Student` entity type you'll create a repository interface and a repository class. When you instantiate the repository in your controller, you'll use the interface so that the controller will accept a reference to any object that implements the repository interface. When the controller runs under a web server, it receives a repository that works with the Entity Framework. When the controller runs under a unit test class, it receives a repository that works with data stored in a way that you can easily manipulate for testing, such as an in-memory collection.

Later in the tutorial you'll use multiple repositories and a unit of work class for the `Course` and `Department` entity types in the `Course` controller. The unit of work class coordinates the work of multiple repositories by creating a single database context class shared by all of them. If you wanted to be able to perform automated unit testing, you'd create and use interfaces for these classes in the same way you did for the `Student` repository. However, to keep the tutorial simple, you'll create and use these classes without interfaces.

The following illustration shows one way to conceptualize the relationships between the controller and context classes compared to not using the repository or unit of work pattern at all.



You won't create unit tests in this tutorial series. For an introduction to TDD with an MVC application that uses the repository pattern, see Walkthrough: Using TDD with ASP.NET MVC. For more information about the repository pattern, see the following resources:

- The Repository Pattern on MSDN.
- Using Repository and Unit of Work patterns with Entity Framework 4.0 on the Entity Framework team blog.
- Agile Entity Framework 4 Repository series of posts on Julie Lerman's blog.
- Building the Account at a Glance HTML5/jQuery Application on Dan Wahlin's blog.

> **NOTE**
> There are many ways to implement the repository and unit of work patterns. You can use repository classes with or without a unit of work class. You can implement a single repository for all entity types, or one for each type. If you implement one for each type, you can use separate classes, a generic base class and derived classes, or an abstract base class and derived classes. You can include business logic in your repository or restrict it to data access logic. You can also build an abstraction layer into your database context class by using IDbSet interfaces there instead of DbSet types for your entity sets. The approach to implementing an abstraction layer shown in this tutorial is one option for you to consider, not a recommendation for all scenarios and environments.

Creating the Student Repository Class

In the *DAL* folder, create a class file named *IStudentRepository.cs* and replace the existing code with the following code:

```
using System;
using System.Collections.Generic;
using ContosoUniversity.Models;

namespace ContosoUniversity.DAL
{
    public interface IStudentRepository : IDisposable
    {
        IEnumerable<Student> GetStudents();
        Student GetStudentByID(int studentId);
        void InsertStudent(Student student);
        void DeleteStudent(int studentID);
        void UpdateStudent(Student student);
        void Save();
    }
}
```

This code declares a typical set of CRUD methods, including two read methods — one that returns all `Student` entities, and one that finds a single `Student` entity by ID.

In the *DAL* folder, create a class file named *StudentRepository.cs* file. Replace the existing code with the following code, which implements the `IStudentRepository` interface:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data;
using ContosoUniversity.Models;

namespace ContosoUniversity.DAL
{
    public class StudentRepository : IStudentRepository, IDisposable
    {
        private SchoolContext context;

        public StudentRepository(SchoolContext context)
        {
            this.context = context;
        }

        public IEnumerable<Student> GetStudents()
        {
            return context.Students.ToList();
        }

        public Student GetStudentByID(int id)
        {
            return context.Students.Find(id);
        }

        public void InsertStudent(Student student)
        {
            context.Students.Add(student);
        }

        public void DeleteStudent(int studentID)
        {
            Student student = context.Students.Find(studentID);
            context.Students.Remove(student);
        }

        public void UpdateStudent(Student student)
        {
            context.Entry(student).State = EntityState.Modified;
        }
```

```
        public void Save()
        {
            context.SaveChanges();
        }

        private bool disposed = false;

        protected virtual void Dispose(bool disposing)
        {
            if (!this.disposed)
            {
                if (disposing)
                {
                    context.Dispose();
                }
            }
            this.disposed = true;
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
    }
}
```

The database context is defined in a class variable, and the constructor expects the calling object to pass in an instance of the context:

```
private SchoolContext context;

public StudentRepository(SchoolContext context)
{
    this.context = context;
}
```

You could instantiate a new context in the repository, but then if you used multiple repositories in one controller, each would end up with a separate context. Later you'll use multiple repositories in the `Course` controller, and you'll see how a unit of work class can ensure that all repositories use the same context.

The repository implements IDisposable and disposes the database context as you saw earlier in the controller, and its CRUD methods make calls to the database context in the same way that you saw earlier.

Change the Student Controller to Use the Repository

In *StudentController.cs*, replace the code currently in the class with the following code. The changes are highlighted.

```
using System;
using System.Data;
using System.Linq;
using System.Web.Mvc;
using ContosoUniversity.Models;
using ContosoUniversity.DAL;
using PagedList;

namespace ContosoUniversity.Controllers
{
   public class StudentController : Controller
   {
      private IStudentRepository studentRepository;
```

```
    public StudentController()
    {
       this.studentRepository = new StudentRepository(new SchoolContext());
    }

    public StudentController(IStudentRepository studentRepository)
    {
       this.studentRepository = studentRepository;
    }

    //
    // GET: /Student/

    public ViewResult Index(string sortOrder, string currentFilter, string searchString, int? page)
    {
       ViewBag.CurrentSort = sortOrder;
       ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
       ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";

       if (searchString != null)
       {
          page = 1;
       }
       else
       {
          searchString = currentFilter;
       }
       ViewBag.CurrentFilter = searchString;

       var students = from s in studentRepository.GetStudents()
                      select s;
       if (!String.IsNullOrEmpty(searchString))
       {
          students = students.Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())
                                  || s.FirstMidName.ToUpper().Contains(searchString.ToUpper()));
       }
       switch (sortOrder)
       {
          case "name_desc":
             students = students.OrderByDescending(s => s.LastName);
             break;
          case "Date":
             students = students.OrderBy(s => s.EnrollmentDate);
             break;
          case "date_desc":
             students = students.OrderByDescending(s => s.EnrollmentDate);
             break;
          default:  // Name ascending
             students = students.OrderBy(s => s.LastName);
             break;
       }

       int pageSize = 3;
       int pageNumber = (page ?? 1);
       return View(students.ToPagedList(pageNumber, pageSize));
    }

    //
    // GET: /Student/Details/5

    public ViewResult Details(int id)
    {
       Student student = studentRepository.GetStudentByID(id);
       return View(student);
    }

    //
    // GET: /Student/Create
```

```
      // GET: /Student/Create

      public ActionResult Create()
      {
         return View();
      }

      //
      // POST: /Student/Create

      [HttpPost]
      [ValidateAntiForgeryToken]
      public ActionResult Create(
         [Bind(Include = "LastName, FirstMidName, EnrollmentDate")]
           Student student)
      {
         try
         {
            if (ModelState.IsValid)
            {
               studentRepository.InsertStudent(student);
               studentRepository.Save();
               return RedirectToAction("Index");
            }
         }
         catch (DataException /* dex */)
         {
            //Log the error (uncomment dex variable name after DataException and add a line here to write a
log.
            ModelState.AddModelError(string.Empty, "Unable to save changes. Try again, and if the problem
persists contact your system administrator.");
         }
         return View(student);
      }

      //
      // GET: /Student/Edit/5

      public ActionResult Edit(int id)
      {
         Student student = studentRepository.GetStudentByID(id);
         return View(student);
      }

      //
      // POST: /Student/Edit/5

      [HttpPost]
      [ValidateAntiForgeryToken]
      public ActionResult Edit(
         [Bind(Include = "LastName, FirstMidName, EnrollmentDate")]
         Student student)
      {
         try
         {
            if (ModelState.IsValid)
            {
               studentRepository.UpdateStudent(student);
               studentRepository.Save();
               return RedirectToAction("Index");
            }
         }
         catch (DataException /* dex */)
         {
            //Log the error (uncomment dex variable name after DataException and add a line here to write a
log.
            ModelState.AddModelError(string.Empty, "Unable to save changes. Try again, and if the problem
persists contact your system administrator.");
         }
         return View(student);
```

```
            return View(student);
        }

        //
        // GET: /Student/Delete/5

        public ActionResult Delete(bool? saveChangesError = false, int id = 0)
        {
            if (saveChangesError.GetValueOrDefault())
            {
                ViewBag.ErrorMessage = "Delete failed. Try again, and if the problem persists see your system
administrator.";
            }
            Student student = studentRepository.GetStudentByID(id);
            return View(student);
        }

        //
        // POST: /Student/Delete/5

        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Delete(int id)
        {
            try
            {
                Student student = studentRepository.GetStudentByID(id);
                studentRepository.DeleteStudent(id);
                studentRepository.Save();
            }
            catch (DataException /* dex */)
            {
                //Log the error (uncomment dex variable name after DataException and add a line here to write a
log.
                return RedirectToAction("Delete", new { id = id, saveChangesError = true });
            }
            return RedirectToAction("Index");
        }

        protected override void Dispose(bool disposing)
        {
            studentRepository.Dispose();
            base.Dispose(disposing);
        }
    }
}
```

The controller now declares a class variable for an object that implements the `IStudentRepository` interface instead of the context class:

```
private IStudentRepository studentRepository;
```

The default (parameterless) constructor creates a new context instance, and an optional constructor allows the caller to pass in a context instance.

```
public StudentController()
{
    this.studentRepository = new StudentRepository(new SchoolContext());
}

public StudentController(IStudentRepository studentRepository)
{
    this.studentRepository = studentRepository;
}
```

(If you were using *dependency injection*, or DI, you wouldn't need the default constructor because the DI software would ensure that the correct repository object would always be provided.)

In the CRUD methods, the repository is now called instead of the context:

```
var students = from s in studentRepository.GetStudents()
               select s;
```

```
Student student = studentRepository.GetStudentByID(id);
```

```
studentRepository.InsertStudent(student);
studentRepository.Save();
```

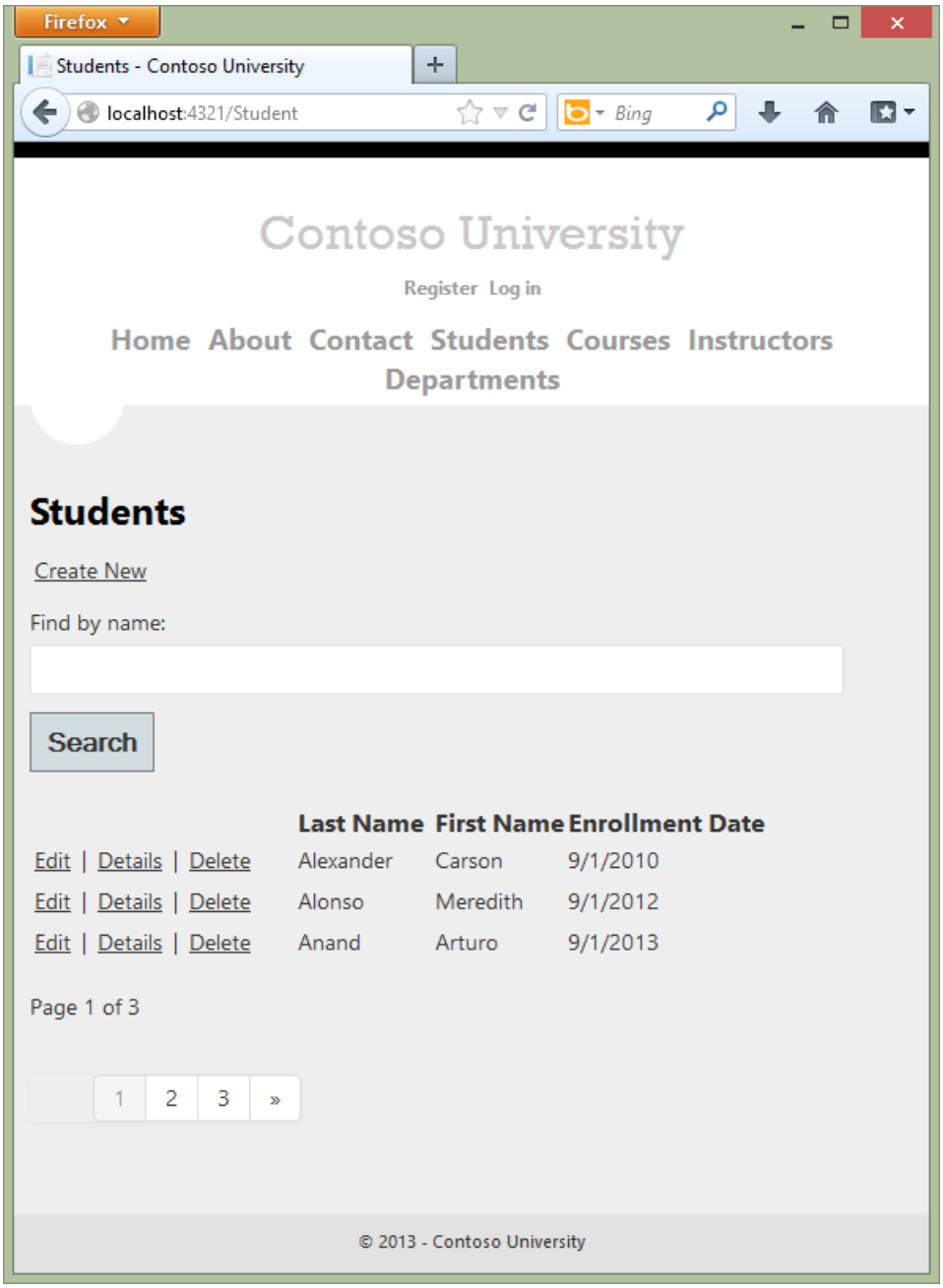
```
studentRepository.UpdateStudent(student);
studentRepository.Save();
```

```
studentRepository.DeleteStudent(id);
studentRepository.Save();
```

And the `Dispose` method now disposes the repository instead of the context:

```
studentRepository.Dispose();
```

Run the site and click the **Students** tab.



The page looks and works the same as it did before you changed the code to use the repository, and the other Student pages also work the same. However, there's an important difference in the way the `Index` method of the controller does filtering and ordering. The original version of this method contained the following code:

```
var students = from s in context.Students
               select s;
if (!String.IsNullOrEmpty(searchString))
{
    students = students.Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())
                           || s.FirstMidName.ToUpper().Contains(searchString.ToUpper()));
}
```

The updated `Index` method contains the following code:

```
var students = from s in studentRepository.GetStudents()
                select s;
if (!String.IsNullOrEmpty(searchString))
{
    students = students.Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())
                        || s.FirstMidName.ToUpper().Contains(searchString.ToUpper()));
}
```

Only the highlighted code has changed.

In the original version of the code, `students` is typed as an `IQueryable` object. The query isn't sent to the database until it's converted into a collection using a method such as `ToList`, which doesn't occur until the Index view accesses the student model. The `Where` method in the original code above becomes a `WHERE` clause in the SQL query that is sent to the database. That in turn means that only the selected entities are returned by the database. However, as a result of changing `context.Students` to `studentRepository.GetStudents()`, the `students` variable after this statement is an `IEnumerable` collection that includes all students in the database. The end result of applying the `Where` method is the same, but now the work is done in memory on the web server and not by the database. For queries that return large volumes of data, this can be inefficient.

TIP

IQueryable vs. IEnumerable

After you implement the repository as shown here, even if you enter something in the **Search** box the query sent to SQL Server returns all Student rows because it doesn't include your search criteria:

```
public ViewResult Index(string sortOrder, string currentFilter, string
{
    ViewBag.CurrentSort = sortOrder;
    ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "Name des
    ViewBag.DateSortParm = sortOrder == "Date" ? "Date desc" : "Date";

    if (searchString != null)
        page = 1;
    else
        searchString = currentFilter;

    ViewBag.CurrentFilter = searchString;

    var students = from s in studentRepository.GetStudents()
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.ToUpper().Contains(sear
                                || s.FirstMidName.ToUpper().Contains(searc
    }
            switch (sortOrder)
            {
                case "Name desc":
                    students = students.OrderByDescending(s => s.LastN
                    break;
                case "Date":
```

```
SELECT
'0X0X' AS [C1],
[Extent1].[PersonID] AS [PersonID],
[Extent1].[LastName] AS [LastName],
[Extent1].[FirstName] AS [FirstName],
[Extent1].[EnrollmentDate] AS [EnrollmentDate]
FROM [dbo].[Person] AS [Extent1]
WHERE [Extent1].[Discriminator] = N'Student'
```

This query returns all of the student data because the repository executed the query without knowing about the search criteria. The process of sorting, applying search criteria, and selecting a subset of the data for paging (showing only 3 rows in this case) is done in memory later when the `ToPagedList` method is called on the `IEnumerable` collection.

In the previous version of the code (before you implemented the repository), the query is not sent to the database until after you apply the search criteria, when `ToPagedList` is called on the `IQueryable` object.

ContosoUniversity (Debugging) - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ARCHITE

Continue Code Map

Process: [6700] iisexpress.exe Suspend Thread: [8312] Worker Thread

StudentController.cs

ContosoUniversity.Controllers.StudentController

```
public ViewResult Index(string sortOrder, string currentFilter, string sea
{
    ViewBag.CurrentSort = sortOrder;
    ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "Name desc" :
    ViewBag.DateSortParm = sortOrder == "Date" ? "Date desc" : "Date";

    if (searchString != null)
        page = 1;
    else
        searchString = currentFilter;

    ViewBag.CurrentFilter = searchString;

    var students = from s in db.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.ToUpper().Contains(searc
                                   || s.FirstMidName.ToUpper().Contains(search
    }
    switch (sortOrder)
    {
        case "Name desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "Date desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:  // Name ascending
            students = students.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    int pageNumber = (page ?? 1);
    return View(students.ToPagedList(pageNumber, pageSize));
}
```

When ToPagedList is called on an `IQueryable` object, the query sent to SQL Server specifies the search string, and as a result only rows that meet the search criteria are returned, and no filtering needs to be done in memory.

```
exec sp_executesql N'SELECT TOP (3)
[Project1].[StudentID] AS [StudentID],
[Project1].[LastName] AS [LastName],
[Project1].[FirstName] AS [FirstName],
[Project1].[EnrollmentDate] AS [EnrollmentDate]
FROM ( SELECT [Project1].[StudentID] AS [StudentID], [Project1].[LastName] AS [LastName], [Project1].
[FirstName] AS [FirstName], [Project1].[EnrollmentDate] AS [EnrollmentDate], row_number() OVER (ORDER BY
[Project1].[LastName] ASC) AS [row_number]
FROM ( SELECT
    [Extent1].[StudentID] AS [StudentID],
    [Extent1].[LastName] AS [LastName],
    [Extent1].[FirstName] AS [FirstName],
    [Extent1].[EnrollmentDate] AS [EnrollmentDate]
    FROM [dbo].[Student] AS [Extent1]
    WHERE (( CAST(CHARINDEX(UPPER(@p__linq__0), UPPER([Extent1].[LastName])) AS int)) > 0) OR ((
CAST(CHARINDEX(UPPER(@p__linq__1), UPPER([Extent1].[FirstName])) AS int)) > 0)
)  AS [Project1]
)  AS [Project1]
WHERE [Project1].[row_number] > 0
ORDER BY [Project1].[LastName] ASC',N'@p__linq__0 nvarchar(4000),@p__linq__1
nvarchar(4000)',@p__linq__0=N'Alex',@p__linq__1=N'Alex'
```

(The following tutorial explains how to examine queries sent to SQL Server.)

The following section shows how to implement repository methods that enable you to specify that this work should be done by the database.

You've now created an abstraction layer between the controller and the Entity Framework database context. If you were going to perform automated unit testing with this application, you could create an alternative repository class in a unit test project that implements `IStudentRepository`. Instead of calling the context to read and write data, this mock repository class could manipulate in-memory collections in order to test controller functions.

Implement a Generic Repository and a Unit of Work Class

Creating a repository class for each entity type could result in a lot of redundant code, and it could result in partial updates. For example, suppose you have to update two different entity types as part of the same transaction. If each uses a separate database context instance, one might succeed and the other might fail. One way to minimize redundant code is to use a generic repository, and one way to ensure that all repositories use the same database context (and thus coordinate all updates) is to use a unit of work class.

In this section of the tutorial, you'll create a `GenericRepository` class and a `UnitOfWork` class, and use them in the `Course` controller to access both the `Department` and the `Course` entity sets. As explained earlier, to keep this part of the tutorial simple, you aren't creating interfaces for these classes. But if you were going to use them to facilitate TDD, you'd typically implement them with interfaces the same way you did the `Student` repository.

Create a Generic Repository

In the *DAL* folder, create *GenericRepository.cs* and replace the existing code with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data;
using System.Data.Entity;
using ContosoUniversity.Models;
using System.Linq.Expressions;

namespace ContosoUniversity.DAL
{
    public class GenericRepository<TEntity> where TEntity : class
    {
        internal SchoolContext context;
        internal DbSet<TEntity> dbSet;
```

```
        internal DbSet<TEntity> dbSet;

        public GenericRepository(SchoolContext context)
        {
            this.context = context;
            this.dbSet = context.Set<TEntity>();
        }

        public virtual IEnumerable<TEntity> Get(
            Expression<Func<TEntity, bool>> filter = null,
            Func<IQueryable<TEntity>, IOrderedQueryable<TEntity>> orderBy = null,
            string includeProperties = "")
        {
            IQueryable<TEntity> query = dbSet;

            if (filter != null)
            {
                query = query.Where(filter);
            }

            foreach (var includeProperty in includeProperties.Split
                (new char[] { ',' }, StringSplitOptions.RemoveEmptyEntries))
            {
                query = query.Include(includeProperty);
            }

            if (orderBy != null)
            {
                return orderBy(query).ToList();
            }
            else
            {
                return query.ToList();
            }
        }

        public virtual TEntity GetByID(object id)
        {
            return dbSet.Find(id);
        }

        public virtual void Insert(TEntity entity)
        {
            dbSet.Add(entity);
        }

        public virtual void Delete(object id)
        {
            TEntity entityToDelete = dbSet.Find(id);
            Delete(entityToDelete);
        }

        public virtual void Delete(TEntity entityToDelete)
        {
            if (context.Entry(entityToDelete).State == EntityState.Detached)
            {
                dbSet.Attach(entityToDelete);
            }
            dbSet.Remove(entityToDelete);
        }

        public virtual void Update(TEntity entityToUpdate)
        {
            dbSet.Attach(entityToUpdate);
            context.Entry(entityToUpdate).State = EntityState.Modified;
        }
    }
}
```

Class variables are declared for the database context and for the entity set that the repository is instantiated for:

```
internal SchoolContext context;
internal DbSet dbSet;
```

The constructor accepts a database context instance and initializes the entity set variable:

```
public GenericRepository(SchoolContext context)
{
    this.context = context;
    this.dbSet = context.Set<TEntity>();
}
```

The `Get` method uses lambda expressions to allow the calling code to specify a filter condition and a column to order the results by, and a string parameter lets the caller provide a comma-delimited list of navigation properties for eager loading:

```
public virtual IEnumerable<TEntity> Get(
    Expression<Func<TEntity, bool>> filter = null,
    Func<IQueryable<TEntity>, IOrderedQueryable<TEntity>> orderBy = null,
    string includeProperties = "")
```

The code `Expression<Func<TEntity, bool>> filter` means the caller will provide a lambda expression based on the `TEntity` type, and this expression will return a Boolean value. For example, if the repository is instantiated for the `Student` entity type, the code in the calling method might specify `student => student.LastName == "Smith` " for the `filter` parameter.

The code `Func<IQueryable<TEntity>, IOrderedQueryable<TEntity>> orderBy` also means the caller will provide a lambda expression. But in this case, the input to the expression is an `IQueryable` object for the `TEntity` type. The expression will return an ordered version of that `IQueryable` object. For example, if the repository is instantiated for the `Student` entity type, the code in the calling method might specify `q => q.OrderBy(s => s.LastName)` for the `orderBy` parameter.

The code in the `Get` method creates an `IQueryable` object and then applies the filter expression if there is one:

```
IQueryable<TEntity> query = dbSet;

if (filter != null)
{
    query = query.Where(filter);
}
```

Next it applies the eager-loading expressions after parsing the comma-delimited list:

```
foreach (var includeProperty in includeProperties.Split
    (new char[] { ',' }, StringSplitOptions.RemoveEmptyEntries))
{
    query = query.Include(includeProperty);
}
```

Finally, it applies the `orderBy` expression if there is one and returns the results; otherwise it returns the results from the unordered query:

```
if (orderBy != null)
{
    return orderBy(query).ToList();
}
else
{
    return query.ToList();
}
```

When you call the `Get` method, you could do filtering and sorting on the `IEnumerable` collection returned by the method instead of providing parameters for these functions. But the sorting and filtering work would then be done in memory on the web server. By using these parameters, you ensure that the work is done by the database rather than the web server. An alternative is to create derived classes for specific entity types and add specialized `Get` methods, such as `GetStudentsInNameOrder` or `GetStudentsByName`. However, in a complex application, this can result in a large number of such derived classes and specialized methods, which could be more work to maintain.

The code in the `GetByID`, `Insert`, and `Update` methods is similar to what you saw in the non-generic repository. (You aren't providing an eager loading parameter in the `GetByID` signature, because you can't do eager loading with the `Find` method.)

Two overloads are provided for the `Delete` method:

```
public virtual void Delete(object id)
{
    TEntity entityToDelete = dbSet.Find(id);
    dbSet.Remove(entityToDelete);
}

public virtual void Delete(TEntity entityToDelete)
{
    if (context.Entry(entityToDelete).State == EntityState.Detached)
    {
        dbSet.Attach(entityToDelete);
    }
    dbSet.Remove(entityToDelete);
}
```

One of these lets you pass in just the ID of the entity to be deleted, and one takes an entity instance. As you saw in the Handling Concurrency tutorial, for concurrency handling you need a `Delete` method that takes an entity instance that includes the original value of a tracking property.

This generic repository will handle typical CRUD requirements. When a particular entity type has special requirements, such as more complex filtering or ordering, you can create a derived class that has additional methods for that type.

Creating the Unit of Work Class

The unit of work class serves one purpose: to make sure that when you use multiple repositories, they share a single database context. That way, when a unit of work is complete you can call the `SaveChanges` method on that instance of the context and be assured that all related changes will be coordinated. All that the class needs is a `Save` method and a property for each repository. Each repository property returns a repository instance that has been instantiated using the same database context instance as the other repository instances.

In the *DAL* folder, create a class file named *UnitOfWork.cs* and replace the template code with the following code:

```
using System;
using ContosoUniversity.Models;

namespace ContosoUniversity.DAL
{
    public class UnitOfWork : IDisposable
    {
        private SchoolContext context = new SchoolContext();
        private GenericRepository<Department> departmentRepository;
        private GenericRepository<Course> courseRepository;

        public GenericRepository<Department> DepartmentRepository
        {
            get
            {

                if (this.departmentRepository == null)
                {
                    this.departmentRepository = new GenericRepository<Department>(context);
                }
                return departmentRepository;
            }
        }

        public GenericRepository<Course> CourseRepository
        {
            get
            {

                if (this.courseRepository == null)
                {
                    this.courseRepository = new GenericRepository<Course>(context);
                }
                return courseRepository;
            }
        }

        public void Save()
        {
            context.SaveChanges();
        }

        private bool disposed = false;

        protected virtual void Dispose(bool disposing)
        {
            if (!this.disposed)
            {
                if (disposing)
                {
                    context.Dispose();
                }
            }
            this.disposed = true;
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
    }
}
```

The code creates class variables for the database context and each repository. For the `context` variable, a new context is instantiated:

```
private SchoolContext context = new SchoolContext();
private GenericRepository<Department> departmentRepository;
private GenericRepository<Course> courseRepository;
```

Each repository property checks whether the repository already exists. If not, it instantiates the repository, passing in the context instance. As a result, all repositories share the same context instance.

```
public GenericRepository<Department> DepartmentRepository
{
    get
    {

        if (this.departmentRepository == null)
        {
            this.departmentRepository = new GenericRepository<Department>(context);
        }
        return departmentRepository;
    }
}
```

The `Save` method calls `SaveChanges` on the database context.

Like any class that instantiates a database context in a class variable, the `UnitOfWork` class implements `IDisposable` and disposes the context.

Changing the Course Controller to use the UnitOfWork Class and Repositories

Replace the code you currently have in *CourseController.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using ContosoUniversity.Models;
using ContosoUniversity.DAL;

namespace ContosoUniversity.Controllers
{
   public class CourseController : Controller
   {
      private UnitOfWork unitOfWork = new UnitOfWork();

      //
      // GET: /Course/

      public ViewResult Index()
      {
         var courses = unitOfWork.CourseRepository.Get(includeProperties: "Department");
         return View(courses.ToList());
      }

      //
      // GET: /Course/Details/5

      public ViewResult Details(int id)
      {
         Course course = unitOfWork.CourseRepository.GetByID(id);
         return View(course);
      }

      //
```

```
      // GET: /Course/Create

      public ActionResult Create()
      {
         PopulateDepartmentsDropDownList();
         return View();
      }

      [HttpPost]
      [ValidateAntiForgeryToken]
      public ActionResult Create(
          [Bind(Include = "CourseID,Title,Credits,DepartmentID")]
         Course course)
      {
         try
         {
            if (ModelState.IsValid)
            {
               unitOfWork.CourseRepository.Insert(course);
               unitOfWork.Save();
               return RedirectToAction("Index");
            }
         }
         catch (DataException /* dex */)
         {
            //Log the error (uncomment dex variable name after DataException and add a line here to write a
log.)
            ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists, see
your system administrator.");
         }
         PopulateDepartmentsDropDownList(course.DepartmentID);
         return View(course);
      }

      public ActionResult Edit(int id)
      {
         Course course = unitOfWork.CourseRepository.GetByID(id);
         PopulateDepartmentsDropDownList(course.DepartmentID);
         return View(course);
      }

      [HttpPost]
      [ValidateAntiForgeryToken]
      public ActionResult Edit(
           [Bind(Include = "CourseID,Title,Credits,DepartmentID")]
         Course course)
      {
         try
         {
            if (ModelState.IsValid)
            {
               unitOfWork.CourseRepository.Update(course);
               unitOfWork.Save();
               return RedirectToAction("Index");
            }
         }
         catch (DataException /* dex */)
         {
            //Log the error (uncomment dex variable name after DataException and add a line here to write a
log.)
            ModelState.AddModelError("", "Unable to save changes. Try again, and if the problem persists, see
your system administrator.");
         }
         PopulateDepartmentsDropDownList(course.DepartmentID);
         return View(course);
      }

      private void PopulateDepartmentsDropDownList(object selectedDepartment = null)
      {
```

```
        {
           var departmentsQuery = unitOfWork.DepartmentRepository.Get(
               orderBy: q => q.OrderBy(d => d.Name));
           ViewBag.DepartmentID = new SelectList(departmentsQuery, "DepartmentID", "Name", selectedDepartment);
        }

        //
        // GET: /Course/Delete/5

        public ActionResult Delete(int id)
        {
           Course course = unitOfWork.CourseRepository.GetByID(id);
           return View(course);
        }

        //
        // POST: /Course/Delete/5

        [HttpPost, ActionName("Delete")]
        [ValidateAntiForgeryToken]
        public ActionResult DeleteConfirmed(int id)
        {
           Course course = unitOfWork.CourseRepository.GetByID(id);
           unitOfWork.CourseRepository.Delete(id);
           unitOfWork.Save();
           return RedirectToAction("Index");
        }

        protected override void Dispose(bool disposing)
        {
           unitOfWork.Dispose();
           base.Dispose(disposing);
        }
     }
  }
```

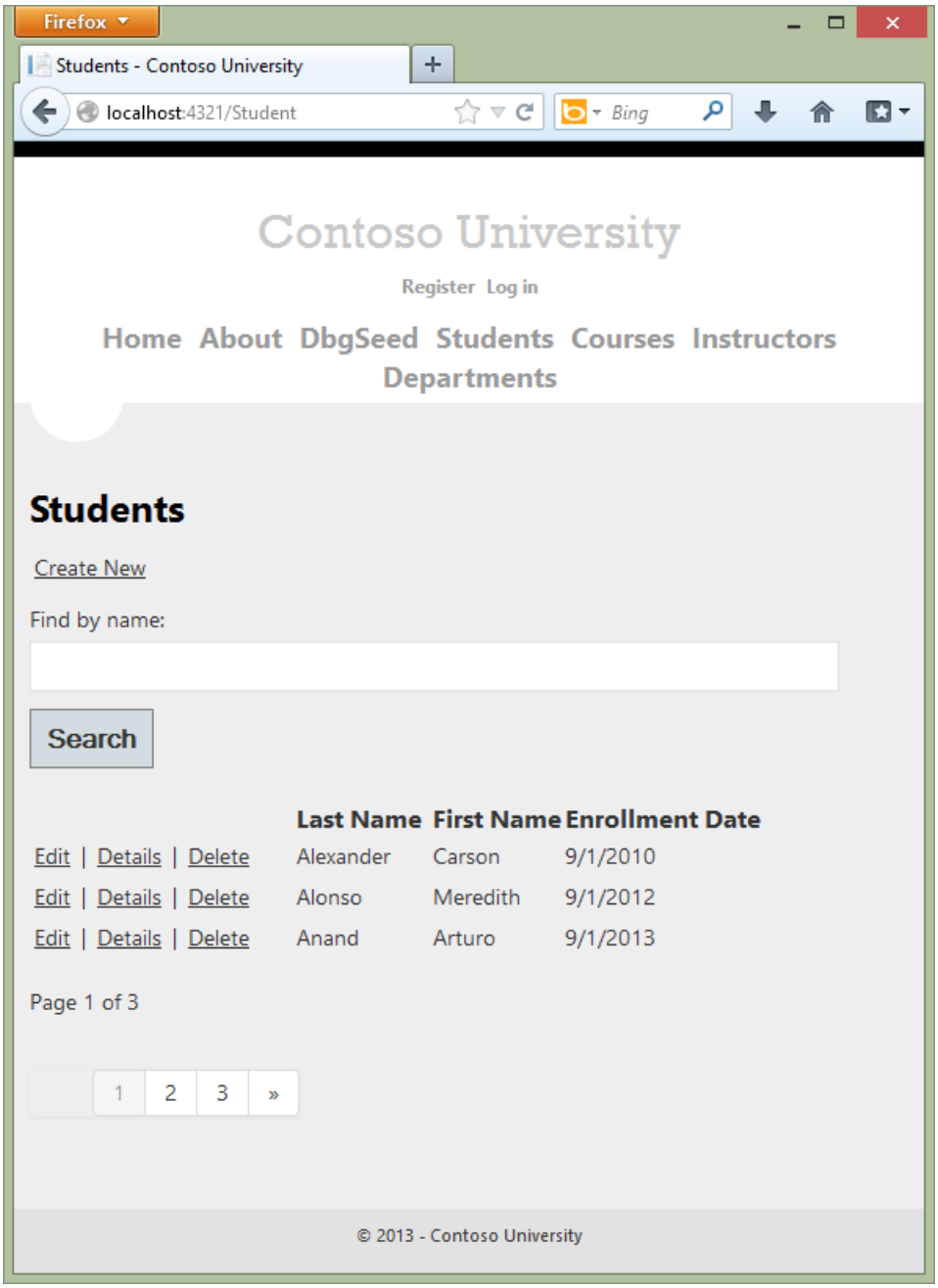
This code adds a class variable for the `UnitOfWork` class. (If you were using interfaces here, you wouldn't initialize the variable here; instead, you'd implement a pattern of two constructors just as you did for the `Student` repository.)

```
private UnitOfWork unitOfWork = new UnitOfWork();
```

In the rest of the class, all references to the database context are replaced by references to the appropriate repository, using `UnitOfWork` properties to access the repository. The `Dispose` method disposes the `UnitOfWork` instance.

```
var courses = unitOfWork.CourseRepository.Get(includeProperties: "Department");
// ...
Course course = unitOfWork.CourseRepository.GetByID(id);
// ...
unitOfWork.CourseRepository.Insert(course);
unitOfWork.Save();
// ...
Course course = unitOfWork.CourseRepository.GetByID(id);
// ...
unitOfWork.CourseRepository.Update(course);
unitOfWork.Save();
// ...
var departmentsQuery = unitOfWork.DepartmentRepository.Get(
    orderBy: q => q.OrderBy(d => d.Name));
// ...
Course course = unitOfWork.CourseRepository.GetByID(id);
// ...
unitOfWork.CourseRepository.Delete(id);
unitOfWork.Save();
// ...
unitOfWork.Dispose();
```

Run the site and click the **Courses** tab.



The page looks and works the same as it did before your changes, and the other Course pages also work the same.

Summary

You have now implemented both the repository and unit of work patterns. You have used lambda expressions as method parameters in the generic repository. For more information about how to use these expressions with an `IQueryable` object, see IQueryable(T) Interface (System.Linq) in the MSDN Library. In the next tutorial you'll learn how to handle some advanced scenarios.

Links to other Entity Framework resources can be found in the ASP.NET Data Access Content Map.

PREVIOUS NEXT

Advanced Entity Framework Scenarios for an MVC Web Application (10 of 10)

9/16/2020 • 18 minutes to read • Edit Online

by Tom Dykstra

Download Completed Project

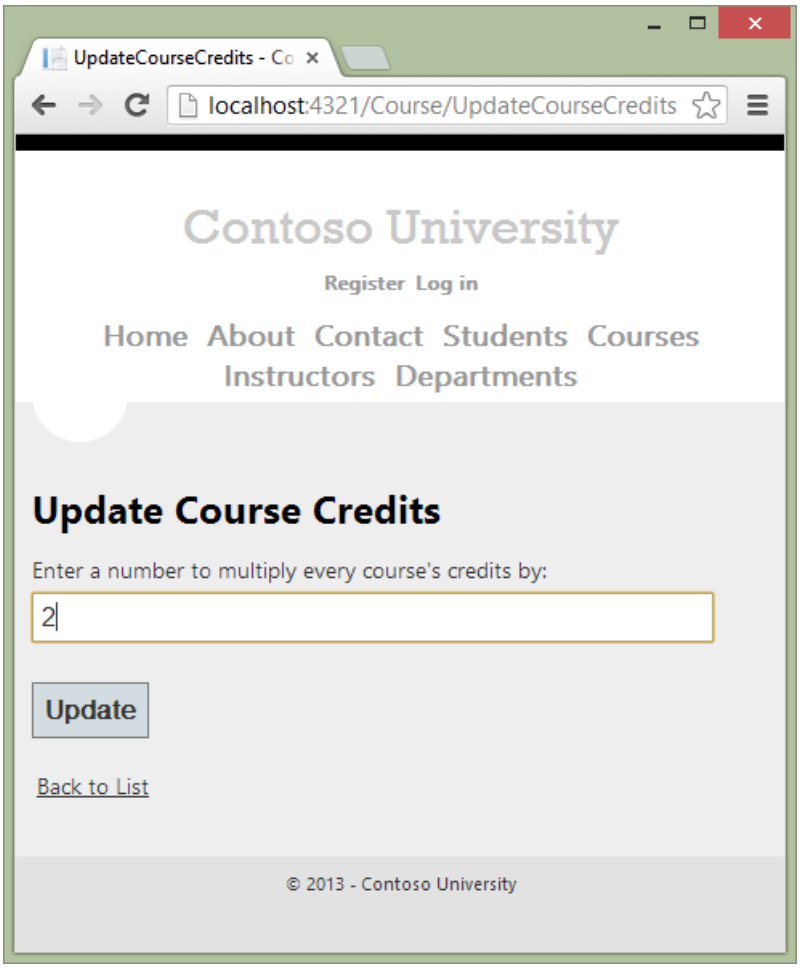
The Contoso University sample web application demonstrates how to create ASP.NET MVC 4 applications using the Entity Framework 5 Code First and Visual Studio 2012. For information about the tutorial series, see the first tutorial in the series. You can start the tutorial series from the beginning or download a starter project for this chapter and start here.

> **NOTE**
>
> If you run into a problem you can't resolve, download the completed chapter and try to reproduce your problem. You can generally find the solution to the problem by comparing your code to the completed code. For some common errors and how to solve them, see Errors and Workarounds.

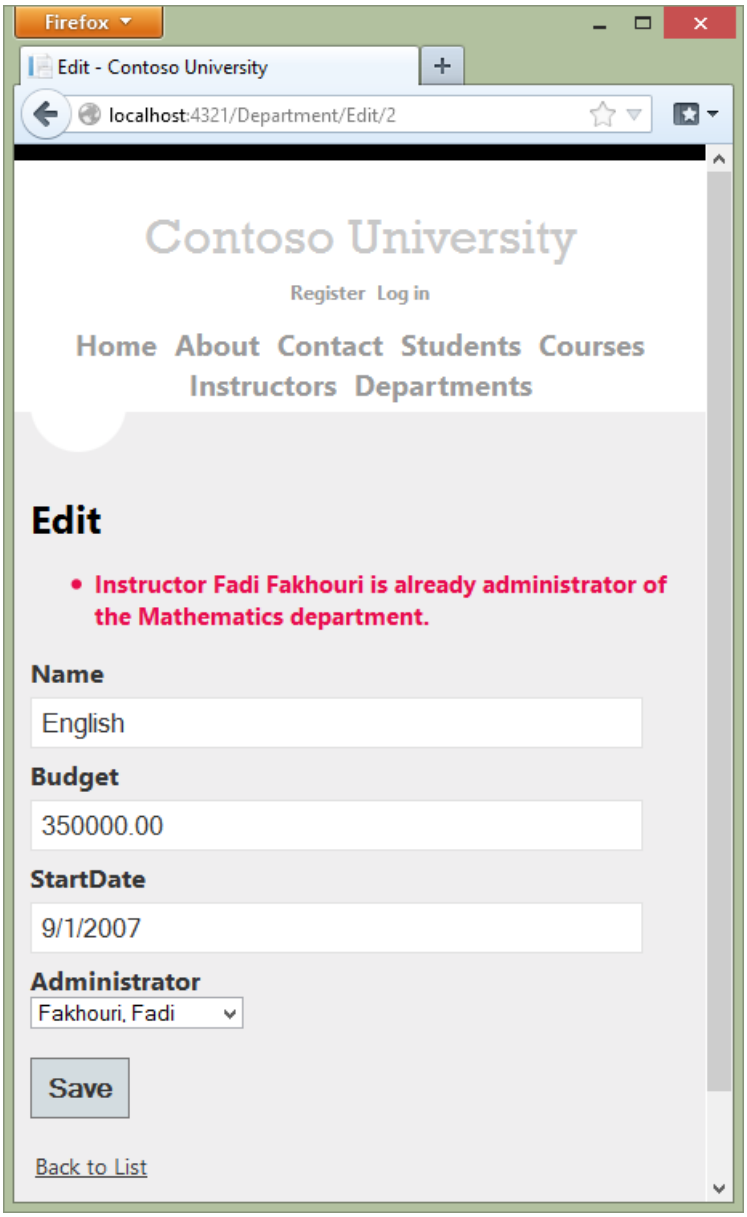
In the previous tutorial you implemented the repository and unit of work patterns. This tutorial covers the following topics:

- Performing raw SQL queries.
- Performing no-tracking queries.
- Examining queries sent to the database.
- Working with proxy classes.
- Disabling automatic detection of changes.
- Disabling validation when saving changes.
- Errors and Work Arounds

For most of these topics, you'll work with pages that you already created. To use raw SQL to do bulk updates you'll create a new page that updates the number of credits of all courses in the database:



And to use a no-tracking query you'll add new validation logic to the Department Edit page:



Performing Raw SQL Queries

The Entity Framework Code First API includes methods that enable you to pass SQL commands directly to the database. You have the following options:

- Use the `DbSet.SqlQuery` method for queries that return entity types. The returned objects must be of the type expected by the `DbSet` object, and they are automatically tracked by the database context unless you turn tracking off. (See the following section about the `AsNoTracking` method.)
- Use the `Database.SqlQuery` method for queries that return types that aren't entities. The returned data isn't tracked by the database context, even if you use this method to retrieve entity types.
- Use the Database.ExecuteSqlCommand for non-query commands.

One of the advantages of using the Entity Framework is that it avoids tying your code too closely to a particular method of storing data. It does this by generating SQL queries and commands for you, which also frees you from having to write them yourself. But there are exceptional scenarios when you need to run specific SQL queries that you have manually created, and these methods make it possible for you to handle such exceptions.

As is always true when you execute SQL commands in a web application, you must take precautions to protect your site against SQL injection attacks. One way to do that is to use parameterized queries to make sure that strings submitted by a web page can't be interpreted as SQL commands. In this tutorial you'll use parameterized queries when integrating user input into a query.

Calling a Query that Returns Entities

Suppose you want the `GenericRepository` class to provide additional filtering and sorting flexibility without requiring that you create a derived class with additional methods. One way to achieve that would be to add a method that accepts a SQL query. You could then specify any kind of filtering or sorting you want in the controller, such as a `Where` clause that depends on a joins or a subquery. In this section you'll see how to implement such a method.

Create the `GetWithRawSql` method by adding the following code to *GenericRepository.cs*:

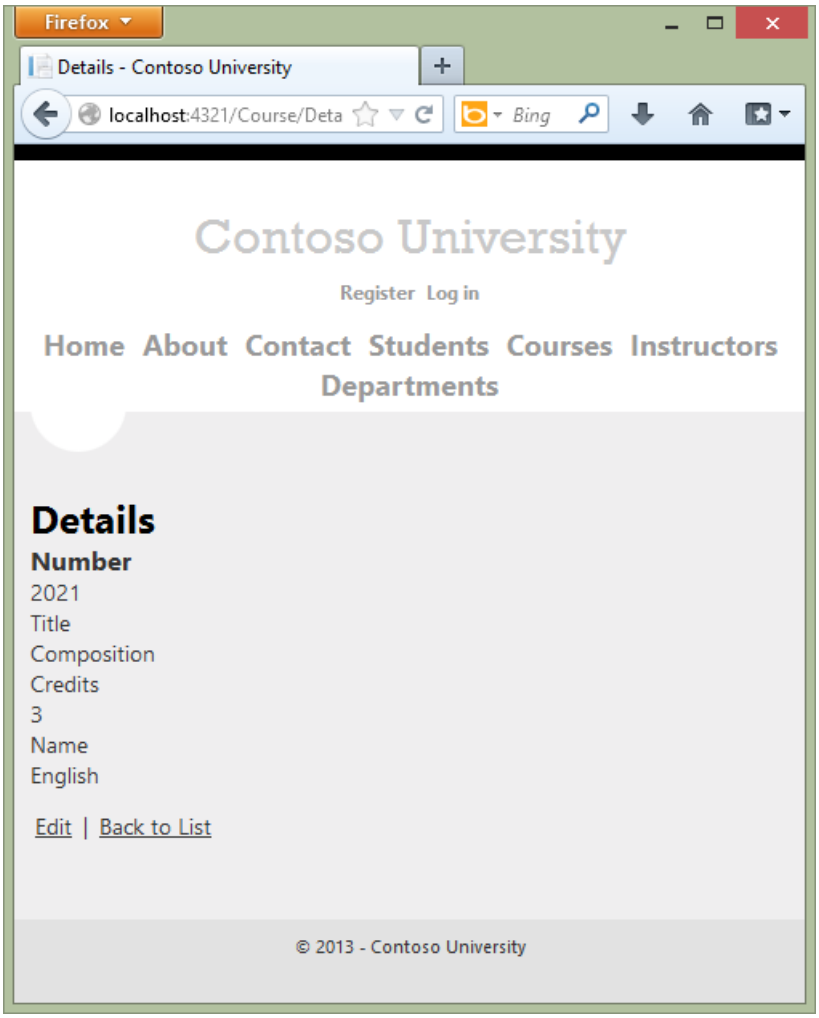
```
public virtual IEnumerable<TEntity> GetWithRawSql(string query, params object[] parameters)
{
    return dbSet.SqlQuery(query, parameters).ToList();
}
```

In *CourseController.cs*, call the new method from the `Details` method, as shown in the following example:

```
public ActionResult Details(int id)
{
    var query = "SELECT * FROM Course WHERE CourseID = @p0";
    return View(unitOfWork.CourseRepository.GetWithRawSql(query, id).Single());
}
```

In this case you could have used the `GetByID` method, but you're using the `GetWithRawSql` method to verify that the `GetWithRawSQL` method works.

Run the Details page to verify that the select query works (select the **Course** tab and then **Details** for one course).



Calling a Query that Returns Other Types of Objects

Earlier you created a student statistics grid for the About page that showed the number of students for each enrollment date. The code that does this in *HomeController.cs* uses LINQ:

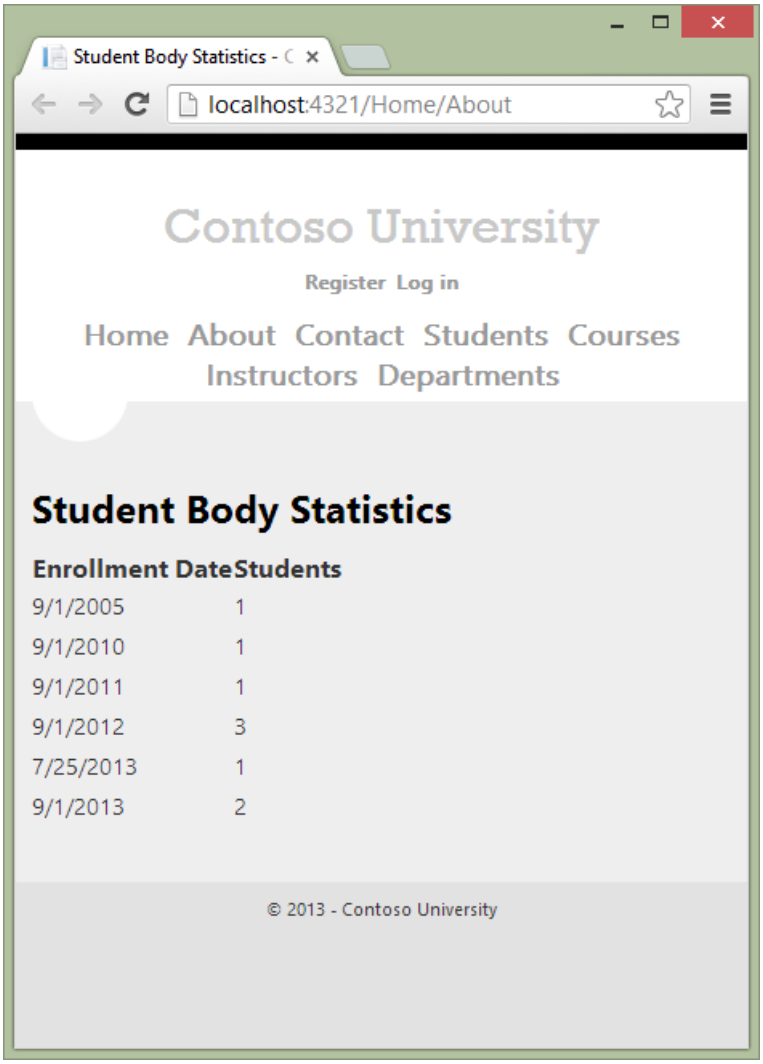
```
var data = from student in db.Students
           group student by student.EnrollmentDate into dateGroup
           select new EnrollmentDateGroup()
           {
               EnrollmentDate = dateGroup.Key,
               StudentCount = dateGroup.Count()
           };
```

Suppose you want to write the code that retrieves this data directly in SQL rather than using LINQ. To do that you need to run a query that returns something other than entity objects, which means you need to use the `Database.SqlQuery` method.

In *HomeController.cs*, replace the LINQ statement in the `About` method with the following code:

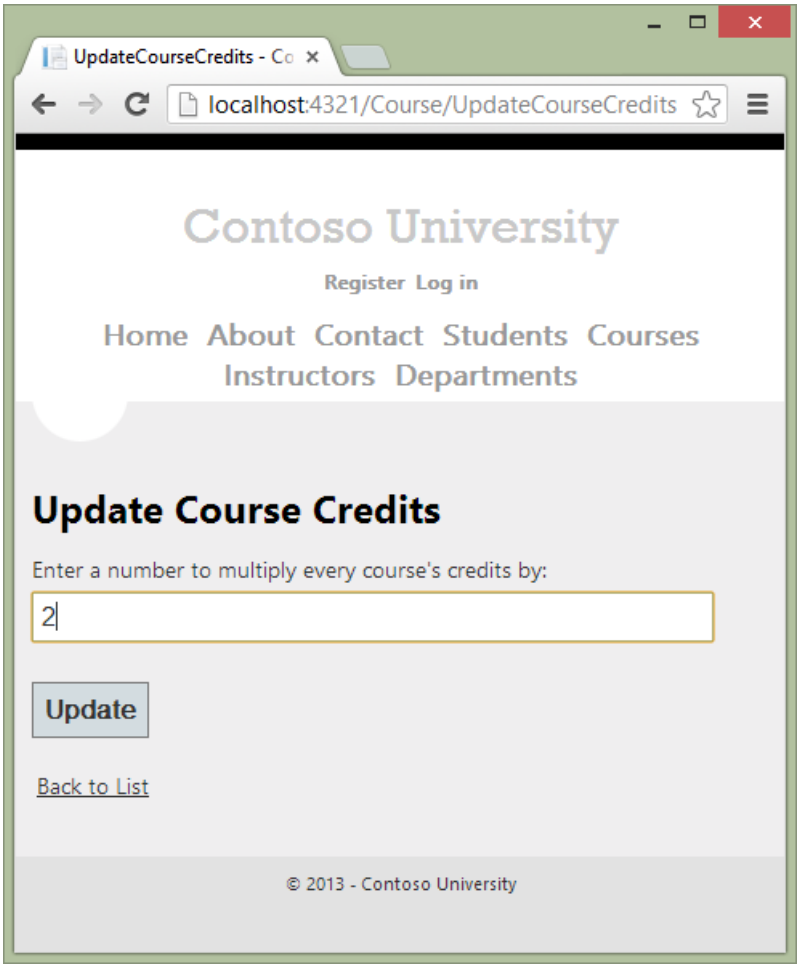
```
var query = "SELECT EnrollmentDate, COUNT(*) AS StudentCount "
    + "FROM Person "
    + "WHERE EnrollmentDate IS NOT NULL "
    + "GROUP BY EnrollmentDate";
var data = db.Database.SqlQuery<EnrollmentDateGroup>(query);
```

Run the About page. It displays the same data it did before.



Calling an Update Query

Suppose Contoso University administrators want to be able to perform bulk changes in the database, such as changing the number of credits for every course. If the university has a large number of courses, it would be inefficient to retrieve them all as entities and change them individually. In this section you'll implement a web page that allows the user to specify a factor by which to change the number of credits for all courses, and you'll make the change by executing a SQL `UPDATE` statement. The web page will look like the following illustration:



In the previous tutorial you used the generic repository to read and update `Course` entities in the `Course` controller. For this bulk update operation, you need to create a new repository method that isn't in the generic repository. To do that, you'll create a dedicated `CourseRepository` class that derives from the `GenericRepository` class.

In the *DAL* folder, create *CourseRepository.cs* and replace the existing code with the following code:

```
using System;
using ContosoUniversity.Models;

namespace ContosoUniversity.DAL
{
    public class CourseRepository : GenericRepository<Course>
    {
        public CourseRepository(SchoolContext context)
            : base(context)
        {
        }

        public int UpdateCourseCredits(int multiplier)
        {
            return context.Database.ExecuteSqlCommand("UPDATE Course SET Credits = Credits * {0}",
multiplier);
        }

    }
}
```

In *UnitOfWork.cs*, change the `Course` repository type from `GenericRepository<Course>` to `CourseRepository:`

```
private CourseRepository courseRepository;
```

```
public CourseRepository CourseRepository
{
    get
    {

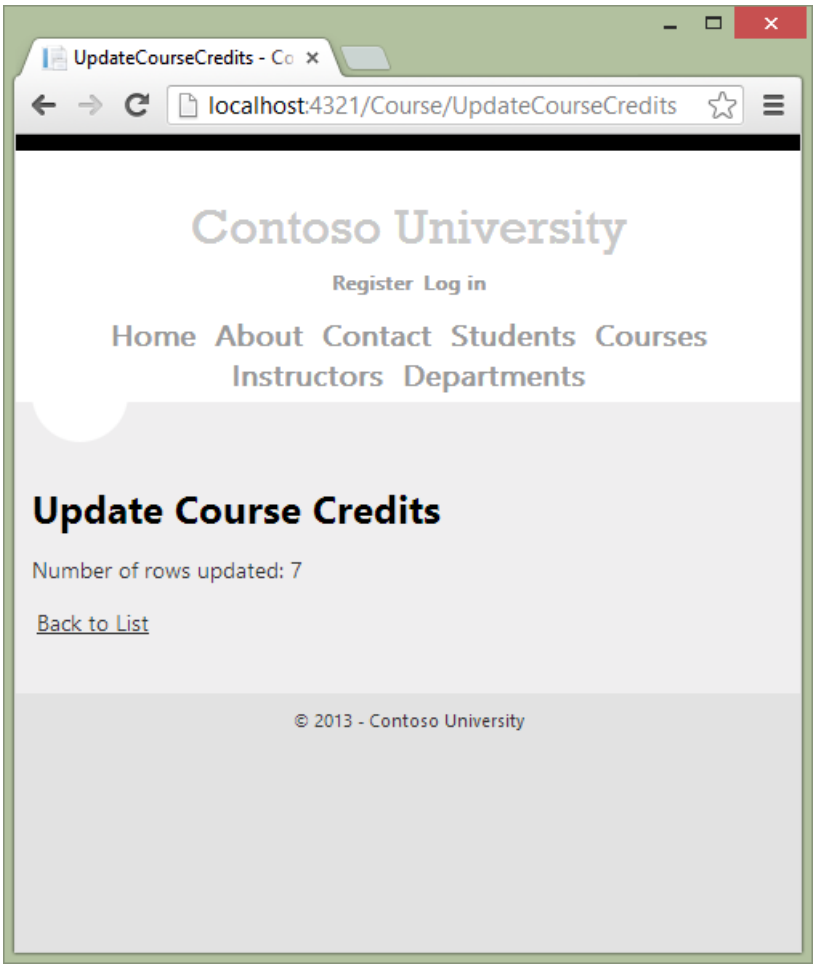
        if (this.courseRepository == null)
        {
            this.courseRepository = new CourseRepository(context);
        }
        return courseRepository;
    }
}
```

In *CourseController.cs*, add an `UpdateCourseCredits` method:

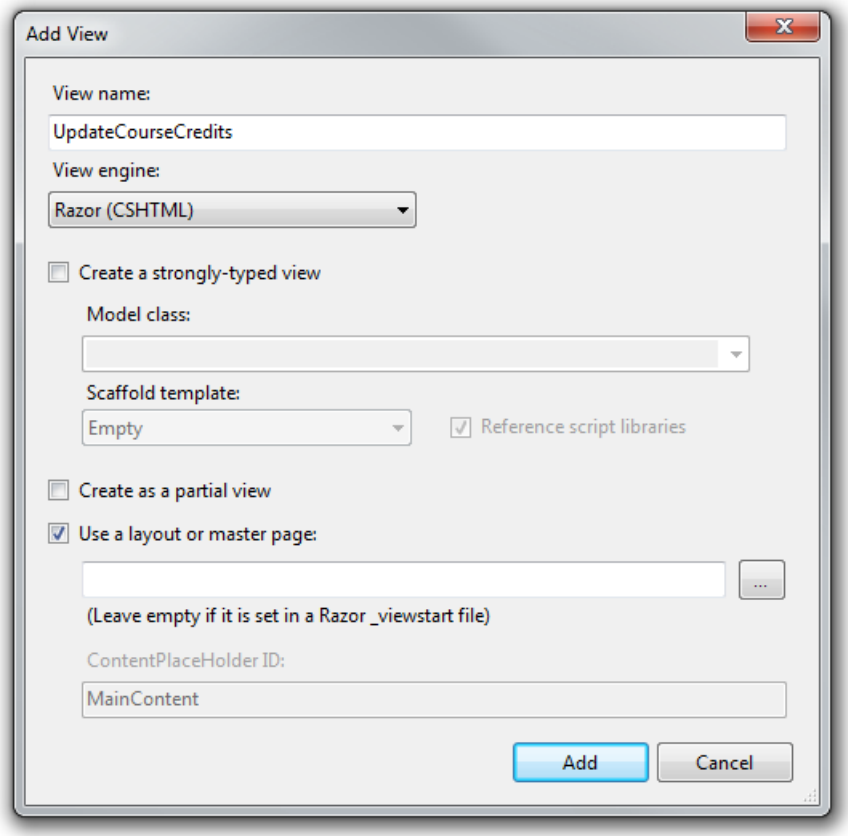
```
public ActionResult UpdateCourseCredits(int? multiplier)
{
    if (multiplier != null)
    {
        ViewBag.RowsAffected = unitOfWork.CourseRepository.UpdateCourseCredits(multiplier.Value);
    }
    return View();
}
```

This method will be used for both `HttpGet` and `HttpPost`. When the `HttpGet` `UpdateCourseCredits` method runs, the `multiplier` variable will be null and the view will display an empty text box and a submit button, as shown in the preceding illustration.

When the **Update** button is clicked and the `HttpPost` method runs, `multiplier` will have the value entered in the text box. The code then calls the repository `UpdateCourseCredits` method, which returns the number of affected rows, and that value is stored in the `ViewBag` object. When the view receives the number of affected rows in the `ViewBag` object, it displays that number instead of the text box and submit button, as shown in the following illustration:



Create a view in the *Views\Course* folder for the Update Course Credits page:



In *Views\Course\UpdateCourseCredits.cshtml*, replace the template code with the following code:

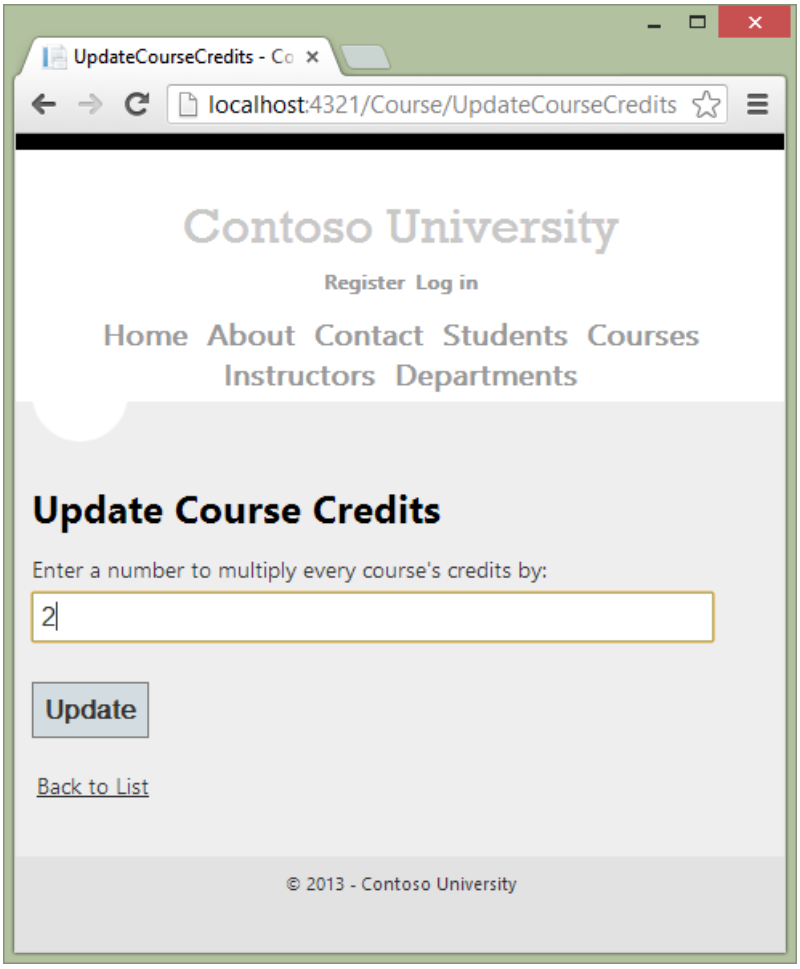
```
@model ContosoUniversity.Models.Course

@{
    ViewBag.Title = "UpdateCourseCredits";
}

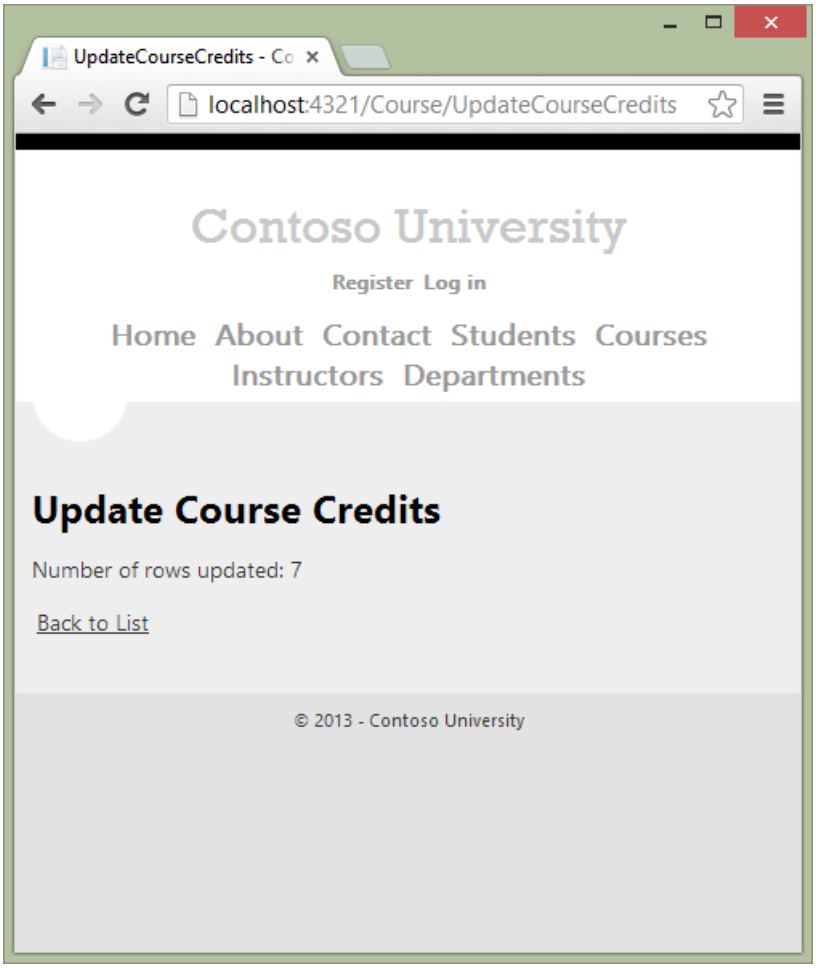
<h2>Update Course Credits</h2>

@if (ViewBag.RowsAffected == null)
{
    using (Html.BeginForm())
    {
        <p>
            Enter a number to multiply every course's credits by: @Html.TextBox("multiplier")
        </p>
        <p>
            <input type="submit" value="Update" />
        </p>
    }
}
@if (ViewBag.RowsAffected != null)
{
    <p>
        Number of rows updated: @ViewBag.RowsAffected
    </p>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

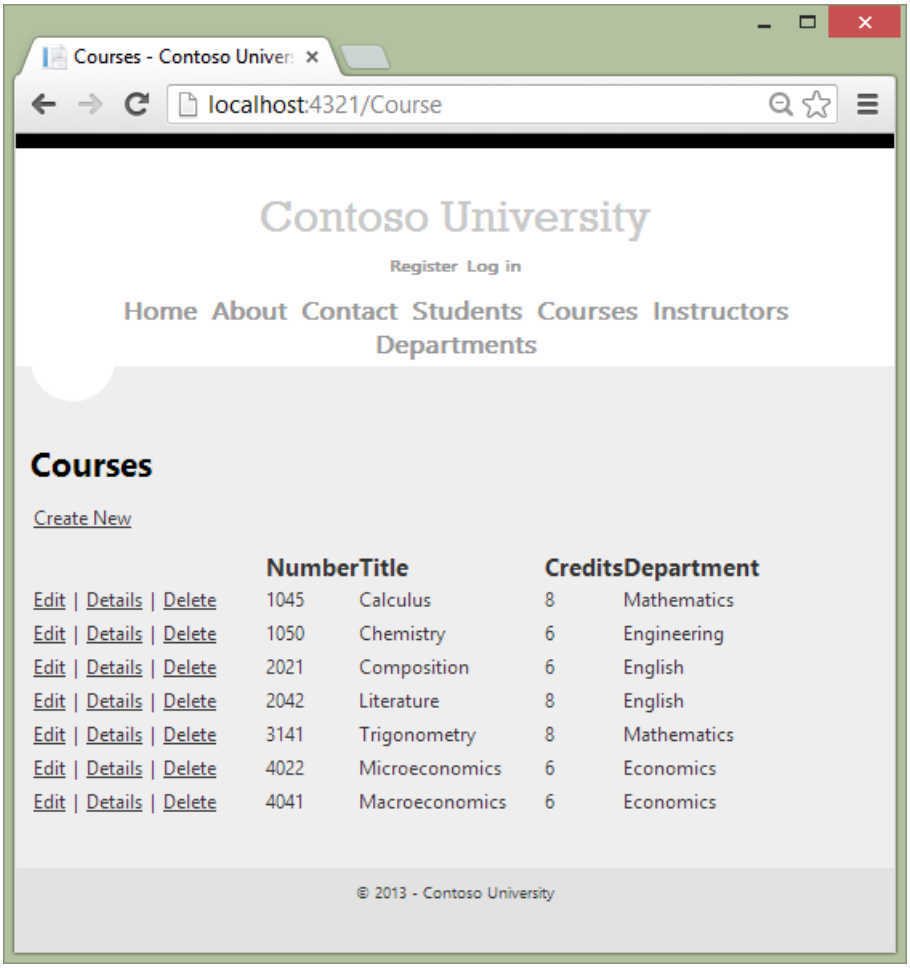
Run the `UpdateCourseCredits` method by selecting the **Courses** tab, then adding "/UpdateCourseCredits" to the end of the URL in the browser's address bar (for example: `http://localhost:50205/Course/UpdateCourseCredits`). Enter a number in the text box:



Click **Update**. You see the number of rows affected:



Click **Back to List** to see the list of courses with the revised number of credits.



For more information about raw SQL queries, see Raw SQL Queries on the Entity Framework team blog.

No-Tracking Queries

When a database context retrieves database rows and creates entity objects that represent them, by default it keeps track of whether the entities in memory are in sync with what's in the database. The data in memory acts as a cache and is used when you update an entity. This caching is often unnecessary in a web application because context instances are typically short-lived (a new one is created and disposed for each request) and the context that reads an entity is typically disposed before that entity is used again.

You can specify whether the context tracks entity objects for a query by using the `AsNoTracking` method. Typical scenarios in which you might want to do that include the following:

- The query retrieves such a large volume of data that turning off tracking might noticeably enhance performance.
- You want to attach an entity in order to update it, but you earlier retrieved the same entity for a different purpose. Because the entity is already being tracked by the database context, you can't attach the entity that you want to change. One way to prevent this from happening is to use the `AsNoTracking` option with the earlier query.

In this section you'll implement business logic that illustrates the second of these scenarios. Specifically, you'll enforce a business rule that says that an instructor can't be the administrator of more than one department.

In *DepartmentController.cs*, add a new method that you can call from the `Edit` and `Create` methods to make sure that no two departments have the same administrator:

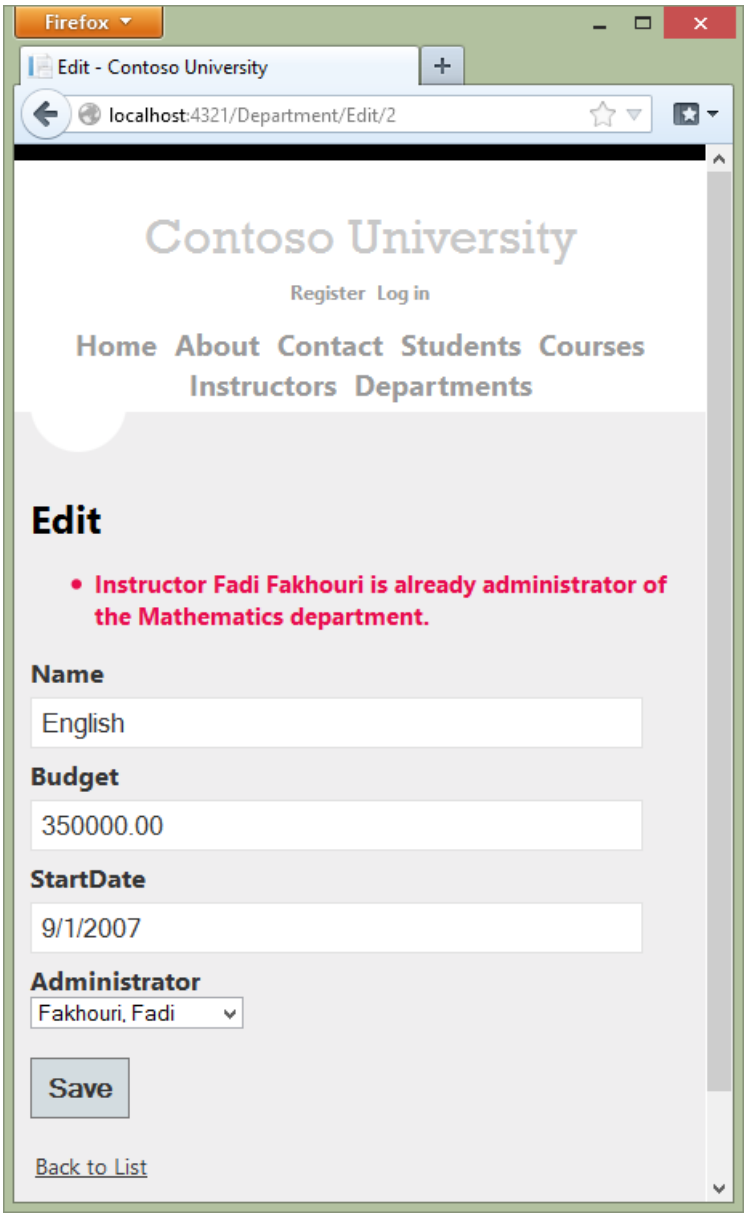
```
private void ValidateOneAdministratorAssignmentPerInstructor(Department department)
{
    if (department.PersonID != null)
    {
        var duplicateDepartment = db.Departments
            .Include("Administrator")
            .Where(d => d.PersonID == department.PersonID)
            .FirstOrDefault();
        if (duplicateDepartment != null && duplicateDepartment.DepartmentID != department.DepartmentID)
        {
            var errorMessage = String.Format(
                "Instructor {0} {1} is already administrator of the {2} department.",
                duplicateDepartment.Administrator.FirstMidName,
                duplicateDepartment.Administrator.LastName,
                duplicateDepartment.Name);
            ModelState.AddModelError(string.Empty, errorMessage);
        }
    }
}
```

Add code in the `try` block of the `HttpPost` `Edit` method to call this new method if there are no validation errors. The `try` block now looks like the following example:

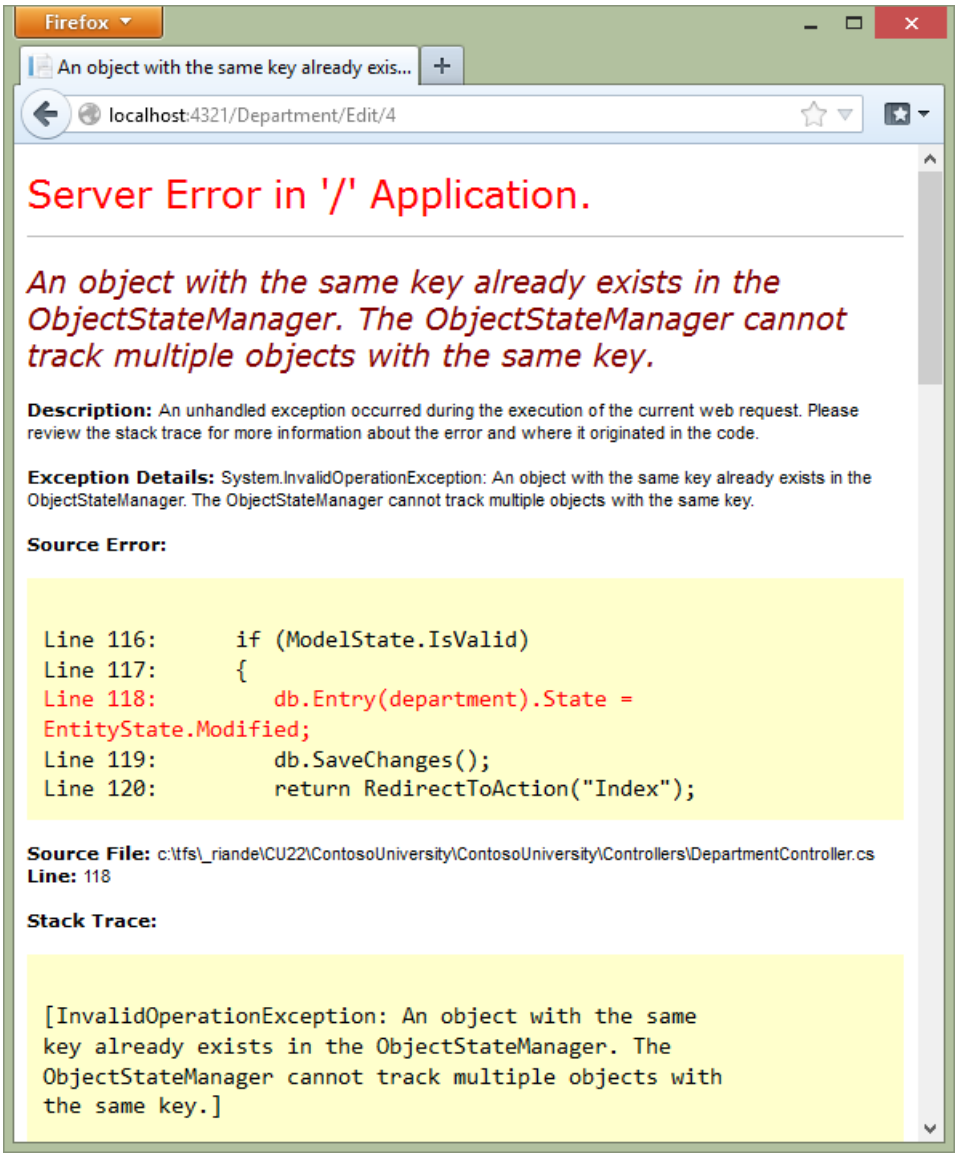
```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(
   [Bind(Include = "DepartmentID, Name, Budget, StartDate, RowVersion, PersonID")]
    Department department)
{
   try
   {
      if (ModelState.IsValid)
      {
         ValidateOneAdministratorAssignmentPerInstructor(department);
      }

      if (ModelState.IsValid)
      {
         db.Entry(department).State = EntityState.Modified;
         db.SaveChanges();
         return RedirectToAction("Index");
      }
   }
   catch (DbUpdateConcurrencyException ex)
   {
      var entry = ex.Entries.Single();
      var clientValues = (Department)entry.Entity;
```

Run the Department Edit page and try to change a department's administrator to an instructor who is already the administrator of a different department. You get the expected error message:



Now run the Department Edit page again and this time change the **Budget** amount. When you click **Save**, you see an error page:



The exception error message is "

```
An object with the same key already exists in the ObjectStateManager. The ObjectStateManager cannot track
multiple objects with the same key.
```

" This happened because of the following sequence of events:

- The `Edit` method calls the `ValidateOneAdministratorAssignmentPerInstructor` method, which retrieves all departments that have Kim Abercrombie as their administrator. That causes the English department to be read. Because that's the department being edited, no error is reported. As a result of this read operation, however, the English department entity that was read from the database is now being tracked by the database context.
- The `Edit` method tries to set the `Modified` flag on the English department entity created by the MVC model binder, but that fails because the context is already tracking an entity for the English department.

One solution to this problem is to keep the context from tracking in-memory department entities retrieved by the validation query. There's no disadvantage to doing this, because you won't be updating this entity or reading it again in a way that would benefit from it being cached in memory.

In *DepartmentController.cs*, in the `ValidateOneAdministratorAssignmentPerInstructor` method, specify no tracking, as shown in the following:

```
var duplicateDepartment = db.Departments
    .Include("Administrator")
    .Where(d => d.PersonID == department.PersonID)
    .AsNoTracking()
    .FirstOrDefault();
```

Repeat your attempt to edit the **Budget** amount of a department. This time the operation is successful, and the site returns as expected to the Departments Index page, showing the revised budget value.

Examining Queries Sent to the Database

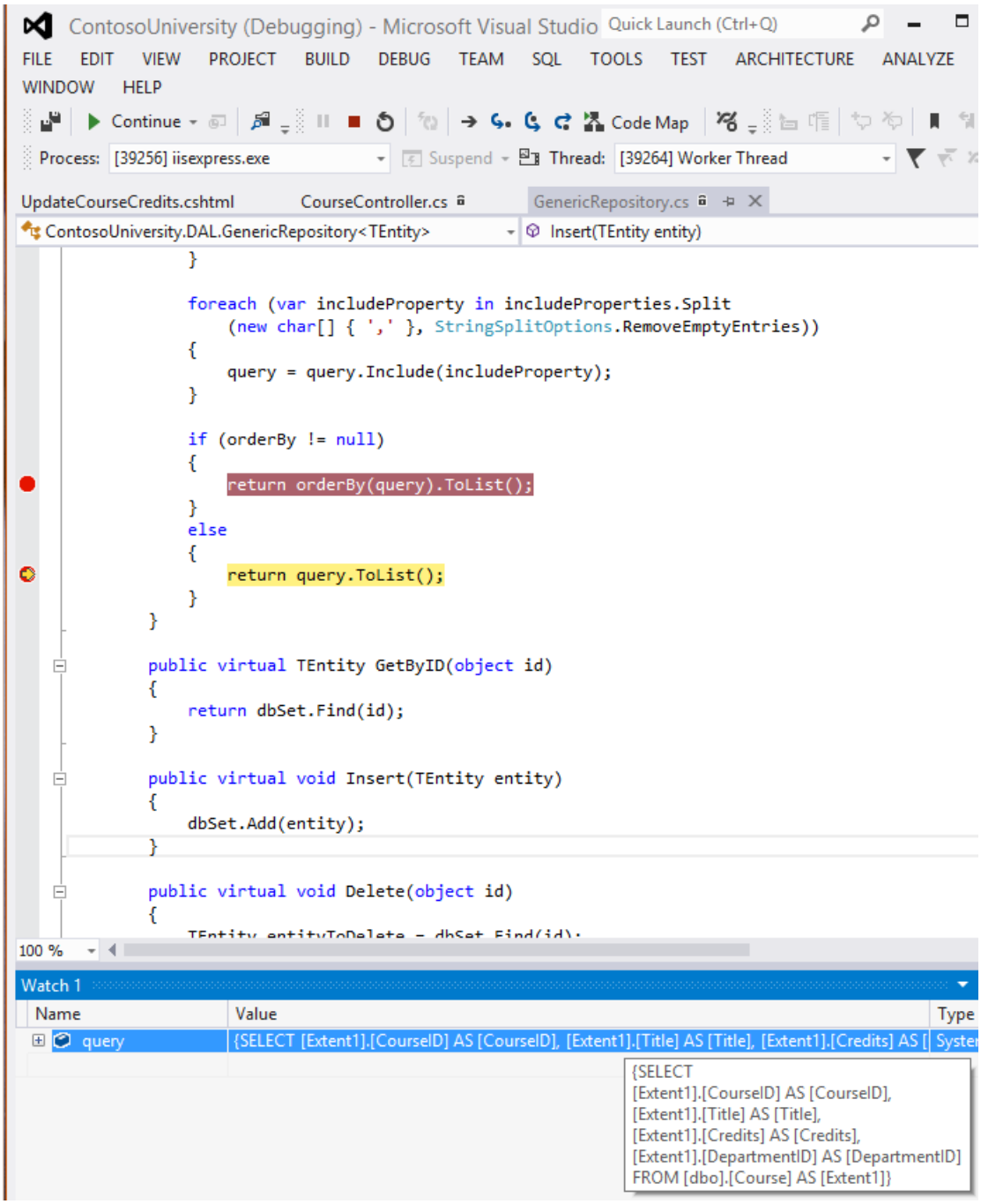
Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. To do this, you can examine a query variable in the debugger or call the query's `ToString` method. To try this out, you'll look at a simple query and then look at what happens to it as you add options such eager loading, filtering, and sorting.

In *Controllers/CourseController*, replace the `Index` method with the following code:

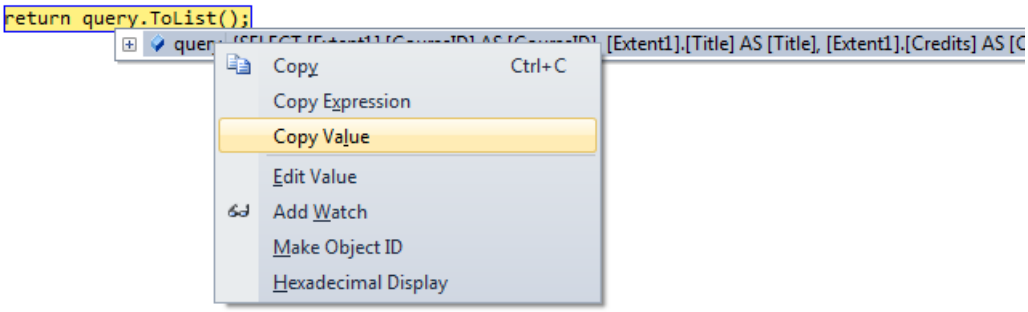
```
public ViewResult Index()
{
    var courses = unitOfWork.CourseRepository.Get();
    return View(courses.ToList());
}
```

Now set a breakpoint in *GenericRepository.cs* on the `return query.ToList();` and the `return orderBy(query).ToList();` statements of the `Get` method. Run the project in debug mode and select the Course Index page. When the code reaches the breakpoint, examine the `query` variable. You see the query that's sent to SQL Server. It's a simple `Select` statement:

```
{SELECT
[Extent1].[CourseID] AS [CourseID],
[Extent1].[Title] AS [Title],
[Extent1].[Credits] AS [Credits],
[Extent1].[DepartmentID] AS [DepartmentID]
FROM [Course] AS [Extent1]}
```



Queries can be too long to display in the debugging windows in Visual Studio. To see the entire query, you can copy the variable value and paste it into a text editor:



Now you'll add a drop-down list to the Course Index page so that users can filter for a particular department. You'll sort the courses by title, and you'll specify eager loading for the `Department` navigation property. In *CourseController.cs*, replace the `Index` method with the following code:

```
public ActionResult Index(int? SelectedDepartment)
{
    var departments = unitOfWork.DepartmentRepository.Get(
        orderBy: q => q.OrderBy(d => d.Name));
    ViewBag.SelectedDepartment = new SelectList(departments, "DepartmentID", "Name", SelectedDepartment);

    int departmentID = SelectedDepartment.GetValueOrDefault();
    return View(unitOfWork.CourseRepository.Get(
        filter: d => !SelectedDepartment.HasValue || d.DepartmentID == departmentID,
        orderBy: q => q.OrderBy(d => d.CourseID),
        includeProperties: "Department"));
}
```

The method receives the selected value of the drop-down list in the `SelectedDepartment` parameter. If nothing is selected, this parameter will be null.

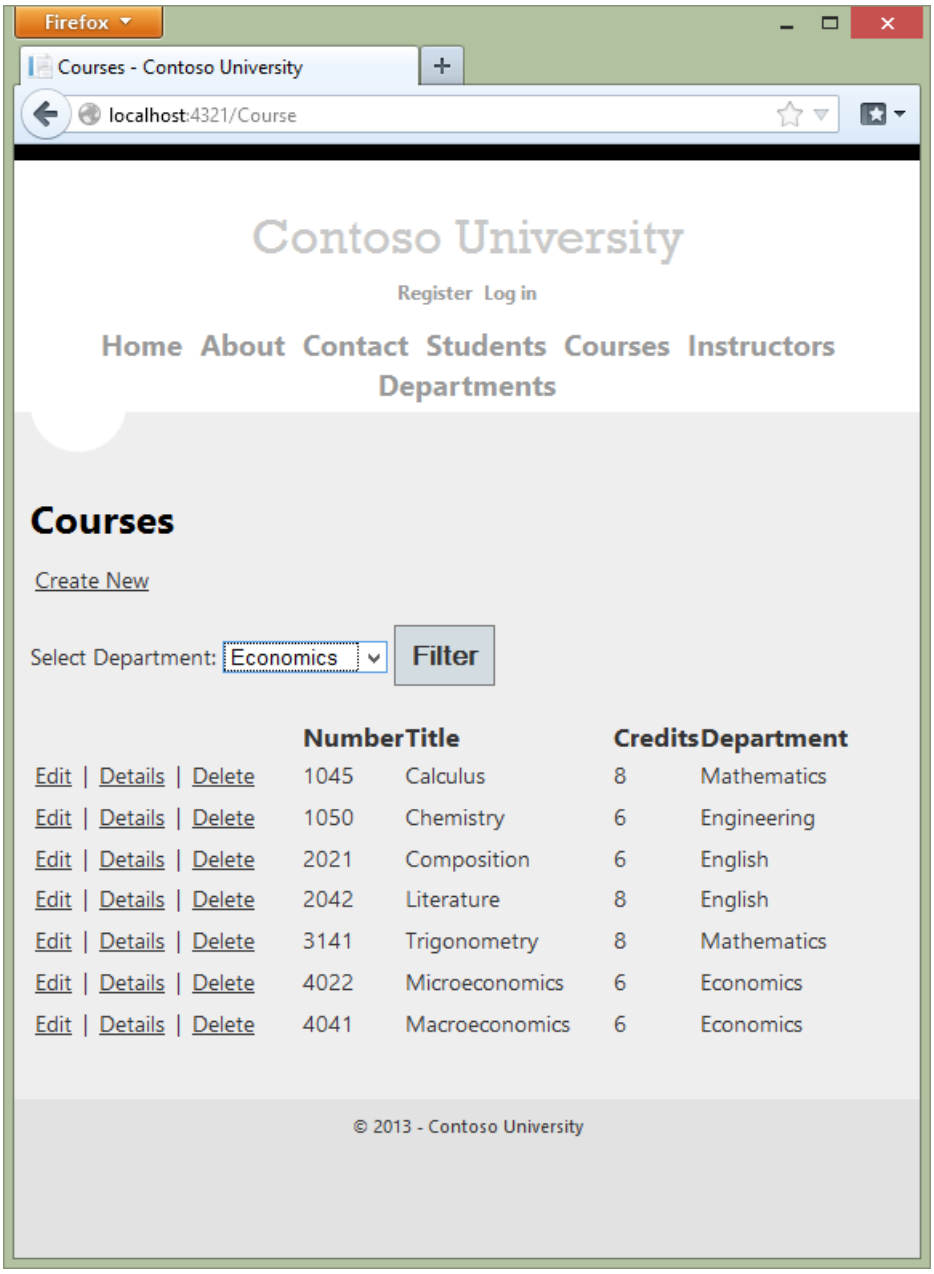
A `SelectList` collection containing all departments is passed to the view for the drop-down list. The parameters passed to the `SelectList` constructor specify the value field name, the text field name, and the selected item.

For the `Get` method of the `Course` repository, the code specifies a filter expression, a sort order, and eager loading for the `Department` navigation property. The filter expression always returns `true` if nothing is selected in the drop-down list (that is, `SelectedDepartment` is null).

In *Views\Course\Index.cshtml*, immediately before the opening `table` tag, add the following code to create the drop-down list and a submit button:

```
@using (Html.BeginForm())
{
    <p>Select Department: @Html.DropDownList("SelectedDepartment","All")
    <input type="submit" value="Filter" /></p>
}
```

With the breakpoints still set in the `GenericRepository` class, run the Course Index page. Continue through the first two times that the code hits a breakpoint, so that the page is displayed in the browser. Select a department from the drop-down list and click **Filter**:



This time the first breakpoint will be for the departments query for the drop-down list. Skip that and view the `query` variable the next time the code reaches the breakpoint in order to see what the `Course` query now looks like. You'll see something like the following:

```
{SELECT
[Extent1].[CourseID] AS [CourseID],
[Extent1].[Title] AS [Title],
[Extent1].[Credits] AS [Credits],
[Extent1].[DepartmentID] AS [DepartmentID],
[Extent2].[DepartmentID] AS [DepartmentID1],
[Extent2].[Name] AS [Name],
[Extent2].[Budget] AS [Budget],
[Extent2].[StartDate] AS [StartDate],
[Extent2].[PersonID] AS [PersonID],
[Extent2].[Timestamp] AS [Timestamp]
FROM  [Course] AS [Extent1]
INNER JOIN [Department] AS [Extent2] ON [Extent1].[DepartmentID] = [Extent2].[DepartmentID]
WHERE (@p__linq__0 IS NULL) OR ([Extent1].[DepartmentID] = @p__linq__1)}
```

You can see that the query is now a `JOIN` query that loads `Department` data along with the `Course` data, and that it includes a `WHERE` clause.

Working with Proxy Classes

When the Entity Framework creates entity instances (for example, when you execute a query), it often creates them as instances of a dynamically generated derived type that acts as a proxy for the entity. This proxy overrides some virtual properties of the entity to insert hooks for performing actions automatically when the property is accessed. For example, this mechanism is used to support lazy loading of relationships.

Most of the time you don't need to be aware of this use of proxies, but there are exceptions:

- In some scenarios you might want to prevent the Entity Framework from creating proxy instances. For example, serializing non-proxy instances might be more efficient than serializing proxy instances.
- When you instantiate an entity class using the `new` operator, you don't get a proxy instance. This means you don't get functionality such as lazy loading and automatic change tracking. This is typically okay; you generally don't need lazy loading, because you're creating a new entity that isn't in the database, and you generally don't need change tracking if you're explicitly marking the entity as `Added`. However, if you do need lazy loading and you need change tracking, you can create new entity instances with proxies using the `Create` method of the `DbSet` class.
- You might want to get an actual entity type from a proxy type. You can use the `GetObjectType` method of the `ObjectContext` class to get the actual entity type of a proxy type instance.

For more information, see Working with Proxies on the Entity Framework team blog.

Disabling Automatic Detection of Changes

The Entity Framework determines how an entity has changed (and therefore which updates need to be sent to the database) by comparing the current values of an entity with the original values. The original values are stored when the entity was queried or attached. Some of the methods that cause automatic change detection are the following:

- `DbSet.Find`
- `DbSet.Local`
- `DbSet.Remove`
- `DbSet.Add`
- `DbSet.Attach`
- `DbContext.SaveChanges`
- `DbContext.GetValidationErrors`
- `DbContext.Entry`
- `DbChangeTracker.Entries`

If you're tracking a large number of entities and you call one of these methods many times in a loop, you might get significant performance improvements by temporarily turning off automatic change detection using the AutoDetectChangesEnabled property. For more information, see Automatically Detecting Changes.

Disabling Validation When Saving Changes

When you call the `SaveChanges` method, by default the Entity Framework validates the data in all properties of all changed entities before updating the database. If you've updated a large number of entities and you've already validated the data, this work is unnecessary and you could make the process of saving the changes take less time by temporarily turning off validation. You can do that using the ValidateOnSaveEnabled property. For more information, see Validation.

Summary

This completes this series of tutorials on using the Entity Framework in an ASP.NET MVC application. Links to other Entity Framework resources can be found in the ASP.NET Data Access Content Map.

For more information about how to deploy your web application after you've built it, see ASP.NET Deployment Content Map in the MSDN Library.

For information about other topics related to MVC, such as authentication and authorization, see the MVC Recommended Resources.

Acknowledgments

- Tom Dykstra wrote the original version of this tutorial and is a senior programming writer on the Microsoft Web Platform and Tools Content Team.
- Rick Anderson (twitter @RickAndMSFT) co-authored this tutorial and did most of the work updating it for EF 5 and MVC 4. Rick is a senior programming writer for Microsoft focusing on Azure and MVC.
- Rowan Miller and other members of the Entity Framework team assisted with code reviews and helped debug many issues with migrations that arose while we were updating the tutorial for EF 5.

VB

When the tutorial was originally produced, we provided both C# and VB versions of the completed download project. With this update we are providing a C# downloadable project for each chapter to make it easier to get started anywhere in the series, but due to time limitations and other priorities we did not do that for VB. If you build a VB project using these tutorials and would be willing to share that with others, please let us know.

Errors and Workarounds

Cannot create/shadow copy

Error Message:

Cannot create/shadow copy 'DotNetOpenAuth.OpenId' when that file already exists.

Solution:

Wait a few seconds and refresh the page.

Update-Database not recognized

Error Message:

The term 'Update-Database' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.(From the `Update-Database` command in the PMC.)

Solution:

Exit Visual Studio. Reopen project and try again.

Validation failed

Error Message:

Validation failed for one or more entities. See 'EntityValidationErrors' property for more details. (From the `Update-Database` command in the PMC.)

Solution:

One cause of this problem is validation errors when the `Seed` method runs. See Seeding and Debugging Entity Framework (EF) DBs for tips on debugging the `Seed` method.

HTTP 500.19 error

Error Message:

HTTP Error 500.19 - Internal Server Error
The requested page cannot be accessed because the related configuration data for the page is invalid.

Solution:

One way you can get this error is from having multiple copies of the solution, each of them using the same port number. You can usually solve this problem by exiting all instances of Visual Studio, then restarting the project your working on. If that doesn't work, try changing the port number. Right click on the project file and then click properties. Select the **Web** tab and then change the port number in the **Project Url** text box.

Error locating SQL Server instance

Error Message:

A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)

Solution:

Check connection string. If you have manually deleted the database, change the name of the database in the construction string.

PREVIOUS NEXT

Building the Chapter Downloads for the EF 5 MVC 4 Tutorials

2/20/2020 • 2 minutes to read • Edit Online

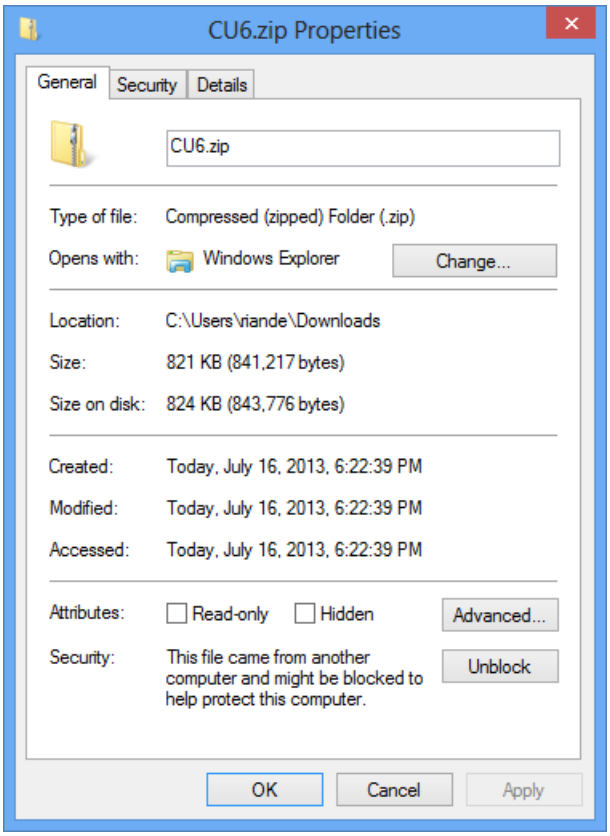
by Rick Anderson

Download Completed Project

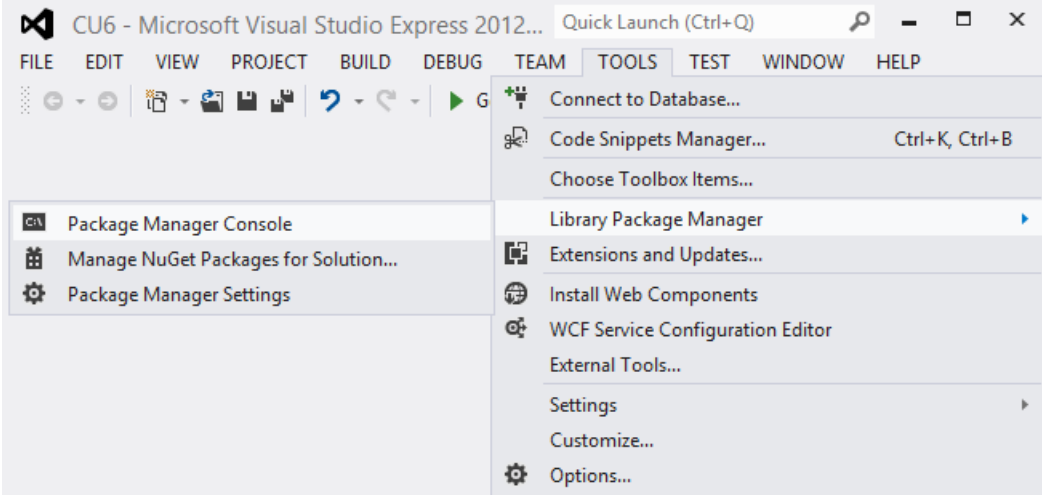
> The Contoso University sample web application demonstrates how to create ASP.NET MVC 4 applications using the Entity Framework 5 Code First and Visual Studio 2012. For information about the tutorial series, see the first tutorial in the series.

Building the Chapter Downloads

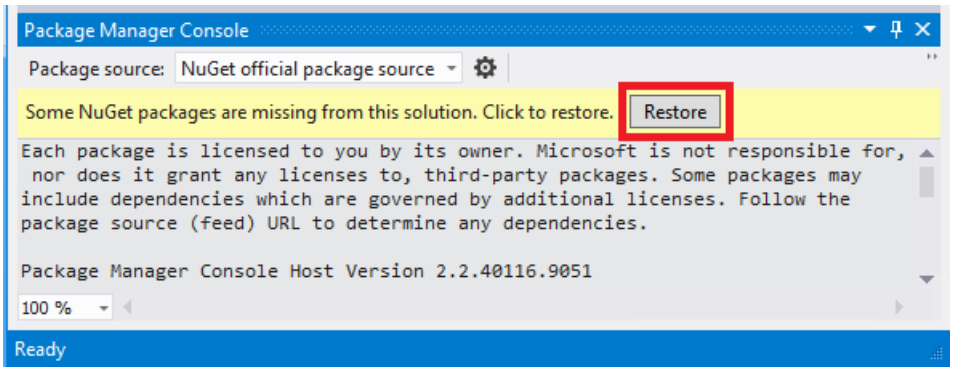
1. Download and unzip the project sample zip file. In the unzipped download package, you will find additional zip files, one for the completion of each chapter.
2. Right click on the desired zip file, click **Properties**, and click the **Unblock** button.



3. Unzip the file.
4. Double-click the *CUx.sln* file to launch Visual Studio.
5. From the **Tools** menu, click **NuGet Package Manager**, then **Package Manager Console**.



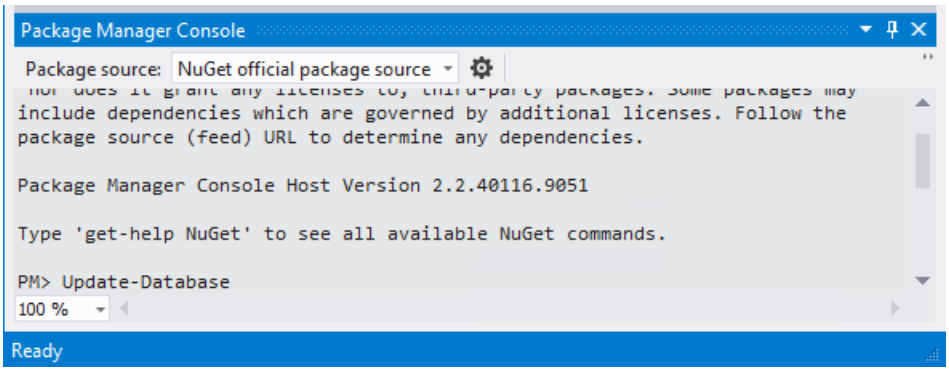
6. In the Package Manager Console (PMC), click **Restore**.



7. Exit Visual Studio.

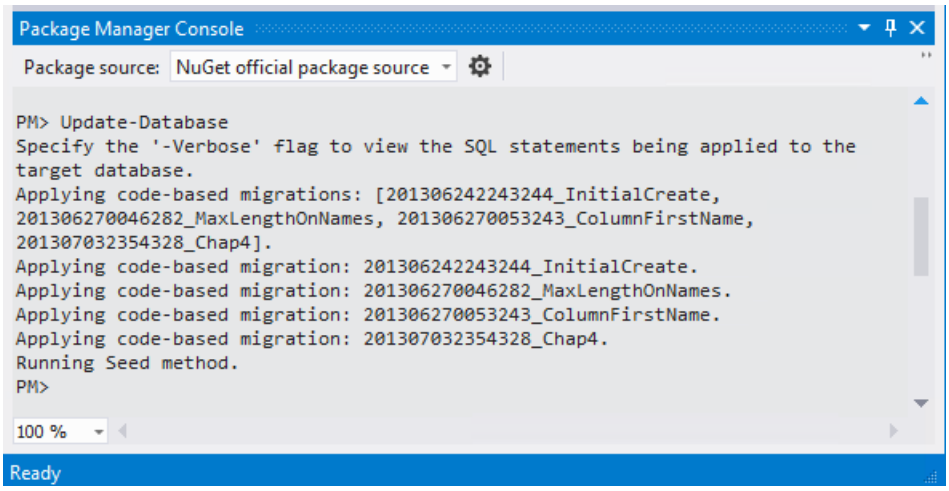
8. Restart Visual Studio, opening the solution file you closed in the step above.

9. In the Package Manager Console (PMC), enter the `Update-Database` command:



> **NOTE**
>
> If you get the following error:
>
> *The term 'Update-Database' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.*
>
> Exit and restart Visual Studio.

Each migration will run, then the seed method will run. You can now run the app.


Package Manager Console
Package source: NuGet official package source
PM> Update-Database
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Applying code-based migrations: [201306242243244_InitialCreate, 201306270046282_MaxLengthOnNames, 201306270053243_ColumnFirstName, 201307032354328_Chap4].
Applying code-based migration: 201306242243244_InitialCreate.
Applying code-based migration: 201306270046282_MaxLengthOnNames.
Applying code-based migration: 201306270053243_ColumnFirstName.
Applying code-based migration: 201307032354328_Chap4.
Running Seed method.
PM>
100 %
Ready

PREVIOUS

ASP.NET MVC 4 Hands On Labs

2/20/2020 • 2 minutes to read • Edit Online

Learn how to develop with ASP.NET MVC with these Hands On Labs

- What's New in ASP.NET MVC 4
- ASP.NET MVC 4 Fundamentals
- ASP.NET MVC 4 Models and Data Access
- ASP.NET MVC 4 Custom Action Filters
- ASP.NET MVC 4 Entity Framework Scaffolding and Migrations
- ASP.NET MVC 4 Dependency Injection
- ASP.NET MVC 4 Helpers, Forms and Validation

What's New in ASP.NET MVC 4

2/20/2020 • 41 minutes to read • Edit Online

By Web Camps Team

Download Web Camps Training Kit

ASP.NET MVC 4 is a framework for building scalable, standards-based web applications using well-established design patterns and the power of the ASP.NET and the .NET framework. This new, fourth version of the framework focuses on making mobile web application development easier.

To begin with, when you create a new ASP.NET MVC 4 project there is now a mobile application project template you can use to build a standalone app specifically for mobile devices. Additionally, ASP.NET MVC 4 integrates with jQuery Mobile through a jQuery.Mobile.MVC NuGet package. jQuery Mobile is an HTML5-based framework for developing web apps that are compatible with all popular mobile device platforms, including Windows Phone, iPhone, Android and so on. However, if you need specialization, ASP.NET MVC 4 also enables you to serve different views for different devices and provide device-specific optimizations.

In this hands-on lab, you will start with the ASP.NET MVC 4 "Internet Application" project template to create a Photo Gallery application. You will progressively enhance the app using jQuery Mobile and ASP.NET MVC 4's new features to make it compatible with different mobile devices and desktop web browsers. You will also learn about new code recipes for code generation and how ASP.NET MVC 4 makes it easier for you to write asynchronous action methods by supporting Task<ActionResult> return types.

> **NOTE**
>
> All sample code and snippets are included in the Web Camps Training Kit, available at Microsoft-Web/WebCampTrainingKit Releases. The project specific to this lab is available at What's New in Web Forms in ASP.NET 4.5.

Objectives

In this hands-on lab, you will learn how to:

- Take advantage of the enhancements to the ASP.NET MVC project templates-including the new mobile application project template
- Use the HTML5 viewport attribute and CSS media queries to improve the display on mobile devices
- Use jQuery Mobile for progressive enhancements and for building touch-optimized web UI
- Create mobile-specific views
- Use the view-switcher component to toggle between mobile and desktop views in the application
- Create asynchronous controllers using task support

Prerequisites

You must have the following items to complete this lab:

- Microsoft Visual Studio Express 2012 for Web or superior (read Appendix B for instructions on how to install it).
- ASP.NET MVC 4 (included in the Microsoft Visual Studio 2012 installation)
- Windows Phone Emulator (included in the Windows Phone 7.1.1 SDK)
- Optional - WebMatrix 2 with **Electric Plum iPhone Simulator** extension (only for Exercise 3 used to browse the web application with an iPhone simulator)

Setup

Throughout the lab document, you will be instructed to insert code blocks. For your convenience, most of that code is provided as Visual Studio Code Snippets, which you can use from within Visual Studio to avoid having to add it manually.

To install the code snippets:

1. Open a Windows Explorer window and browse to the lab's **Source\Setup** folder.
2. Double-click the **Setup.cmd** file in this folder to install the Visual Studio code snippets.

If you are not familiar with the Visual Studio Code Snippets, and want to learn how to use them, you can refer to the appendix from this document "Appendix A: Using Code Snippets".

Exercises

This hands-on lab includes the following exercises:

1. New ASP.NET MVC 4 Project Templates
2. Creating the Photo Gallery Web Application
3. Adding Support for Mobile Devices
4. Using Asynchronous Controllers

> **NOTE**
> Each exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

Estimated time to complete this lab: **60 minutes**.

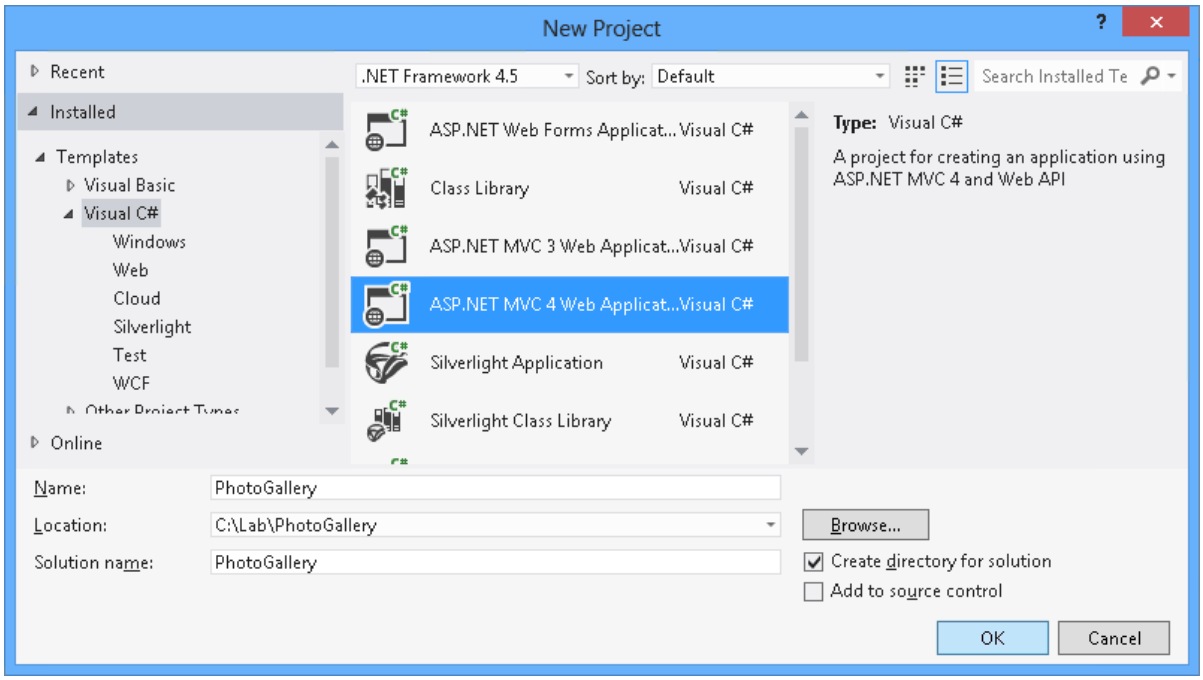
Exercise 1: New ASP.NET MVC 4 Project Templates

In this exercise, you will explore the enhancements in the ASP.NET MVC 4 Project templates. In addition to the Internet Application template, already present in MVC 3, this version now includes a separate template for Mobile applications. First, you will look at some relevant features of each of the templates. Then, you will work on rendering your page properly on the different platforms by using the right approach.

Task 1 - Exploring the Internet Application Template

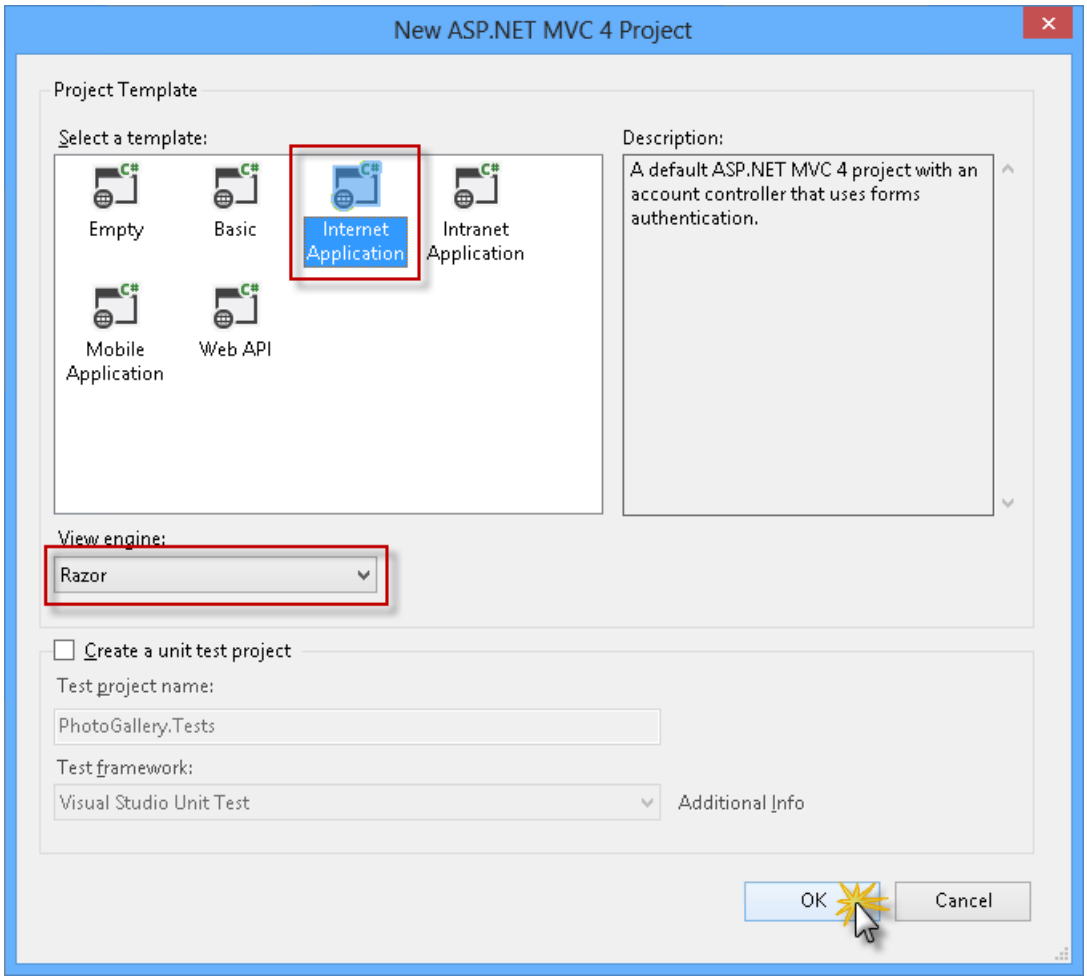
1. Open **Visual Studio**.
2. Select the **File | New | Project** menu command. In the **New Project** dialog, select the **Visual C# | Web** template on the left pane tree, and choose **ASP.NET MVC 4 Web Application.** Name the project **PhotoGallery**, select a location (or leave the default) and click **OK**.

   > **NOTE**
   > You will later customize the PhotoGallery ASP.NET MVC 4 solution you are now creating.



Creating a new project

3. In the **New ASP.NET MVC 4 Project** dialog, select the **Internet Application** project template and click **OK**. Make sure you have selected Razor as the view engine.



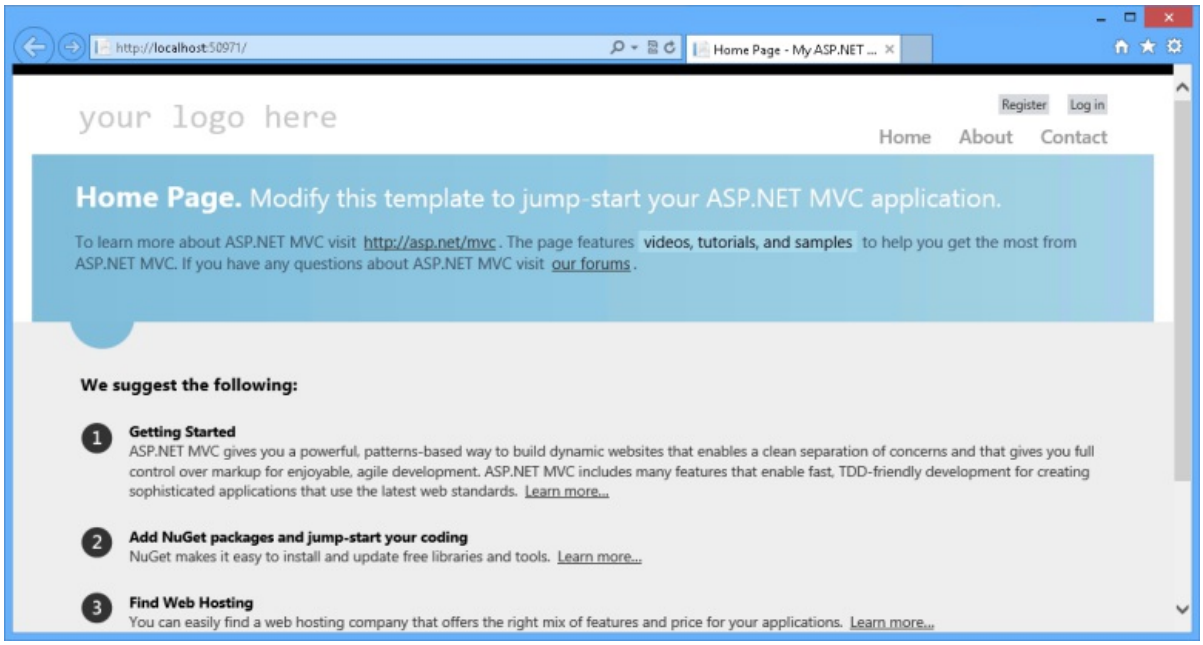
Creating a new ASP.NET MVC 4 Internet Application

> **NOTE**
>
> Razor syntax has been introduced in ASP.NET MVC 3. Its goal is to minimize the number of characters and keystrokes required in a file, enabling a fast and fluid coding workflow. Razor leverages existing C# / VB (or other) language skills and delivers a template markup syntax that enables an awesome HTML construction workflow.

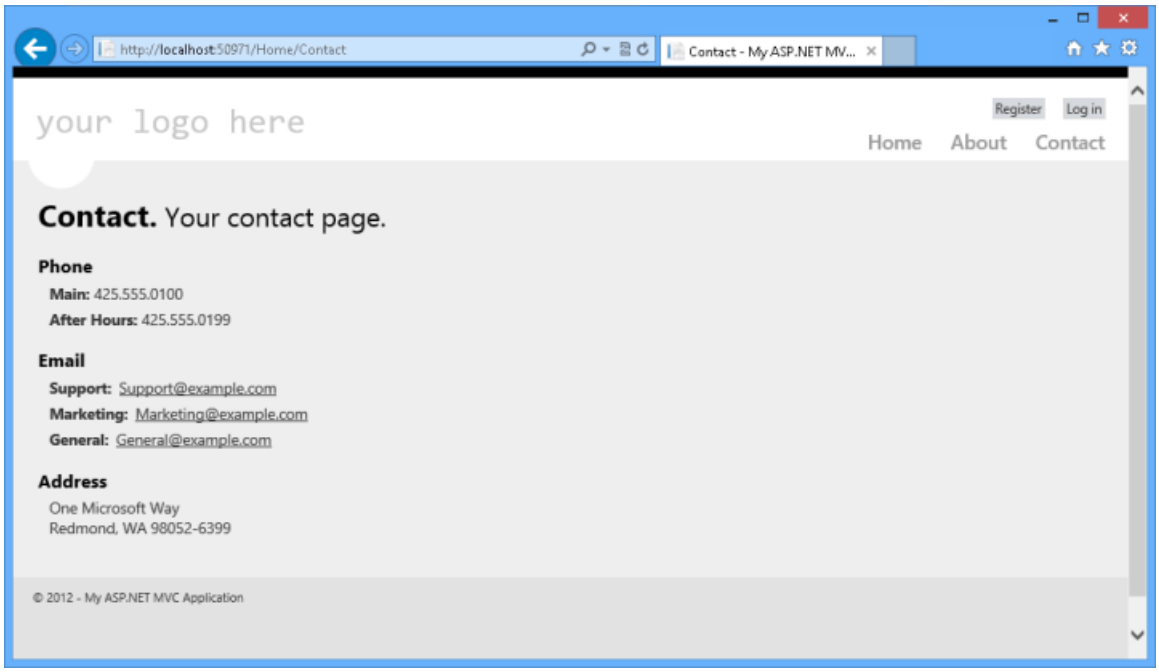
4. Press **F5** to run the solution and see the renewed templates. You can check out the following features:

Modern-style templates

The templates have been renewed, providing more modern-looking styles.



ASP.NET MVC 4 restyled templates

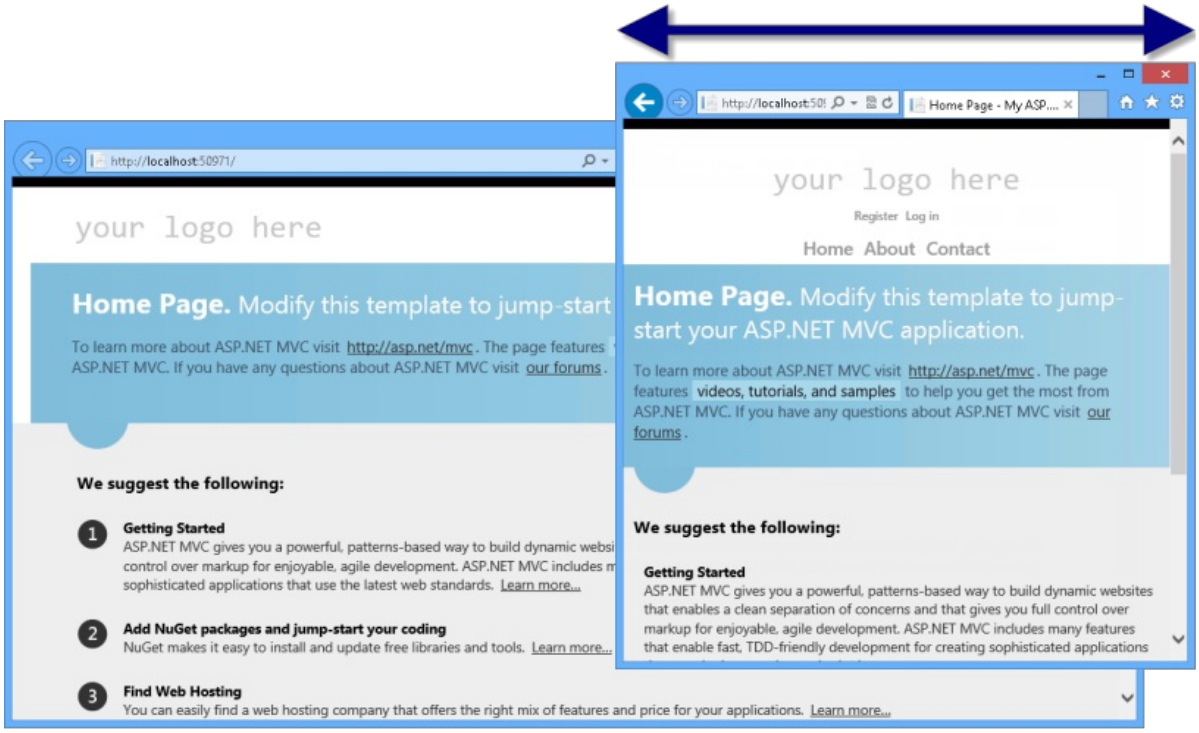


New Contact page

Adaptive Rendering

Check out resizing the browser window and notice how the page layout dynamically adapts to the new window size. These templates use the adaptive rendering technique to render properly in both desktop and

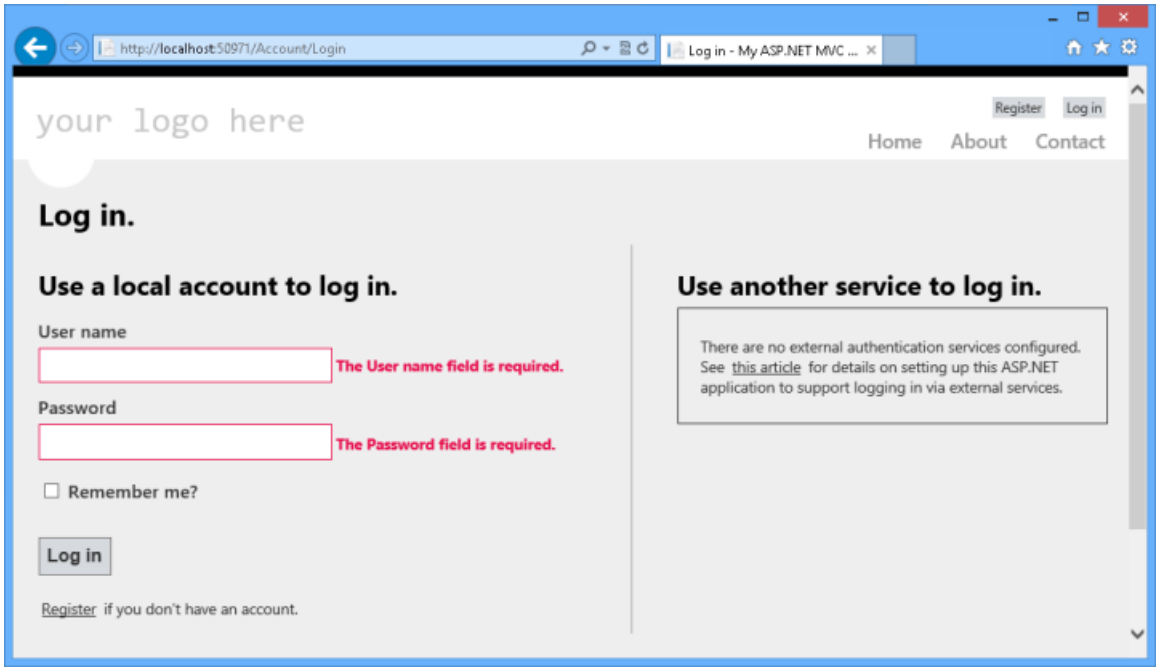
mobile platforms without any customization.



ASP.NET MVC 4 project template in different browser sizes

Richer UI with JavaScript

Another enhancement to default project templates is the use of JavaScript to provide a more interactive JavaScript. The Login and Register links used in the template exemplify how to use the jQuery Validations to validate the input fields from client-side.



jQuery Validation

> **NOTE**
>
> Notice the two log in sections, in the first section you can log in using a registered account from the site and in the second section you can alternatively log in using another authentication service like google (disabled by default).

5. Close the browser to stop the debugger and return to Visual Studio.

6. Open the file **AuthConfig.cs** located under the **App_Start** folder.

7. Remove the comment from the last line to register Google client for *OAuth* authentication.

```
public static class AuthConfig
{
    public static void RegisterAuth()
    {
        // To let users of this site log in using their accounts from other sites such as Microsoft,
Facebook, and Twitter,
        // you must update this site. For more information visit https://go.microsoft.com/fwlink/?
LinkID=252166

        //OAuthWebSecurity.RegisterMicrosoftClient(
        //    clientId: "",
        //    clientSecret: "");

        //OAuthWebSecurity.RegisterTwitterClient(
        //    consumerKey: "",
        //    consumerSecret: "");

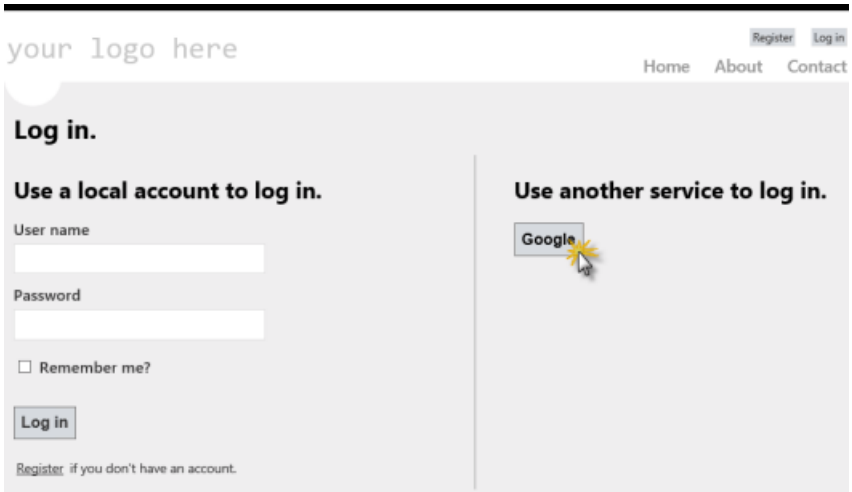
        //OAuthWebSecurity.RegisterFacebookClient(
        //    appId: "",
        //    appSecret: "");

        OAuthWebSecurity.RegisterGoogleClient();
    }
}
```

> **NOTE**
>
> Notice you can easily enable authentication using any OpenID or OAuth service like Facebook, Twitter, Microsoft, etc.

8. Press **F5** to run the solution and navigate to the login page.

9. Select **Google** service to log in.

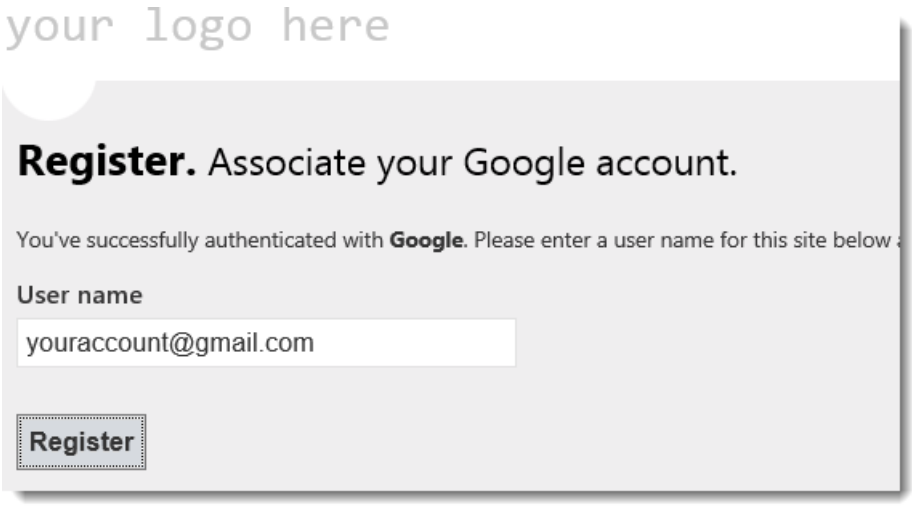


Selecting the log in service

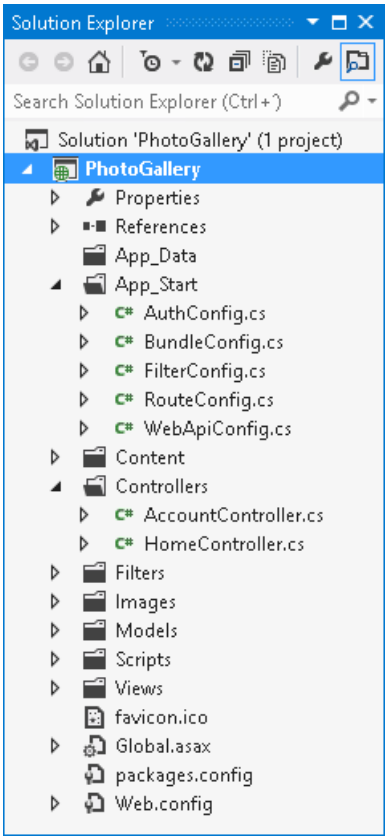
10. Log in using your Google account.

11. Allow the site (localhost) to retrieve information from Google account.

12. Finally, you will have to register in the site to associate the Google account.



Associating your Google account 13. Close the browser to stop the debugger and return to Visual Studio. 14. Now explore the solution to check out some other new features introduced by ASP.NET MVC 4 in the project template.



The ASP.NET MVC 4 Internet Application Project Template

```
- **HTML 5 Markup**

   Browse template views to find out the new theme markup.

   ![New template, using Razor and HTML5 markup About.cshtml.](whats-new-in-aspnet-mvc-4/_static/image10.png
"New template, using Razor and HTML5 markup About.cshtml.")

   *New template, using Razor and HTML5 markup (About.cshtml).*
- **Updated JavaScript libraries**

   The ASP.NET MVC 4 default template now includes KnockoutJS, a JavaScript MVVM framework that lets you
create rich and highly responsive web applications using JavaScript and HTML. Like in MVC3, jQuery and jQuery
UI libraries are also included in ASP.NET MVC 4.

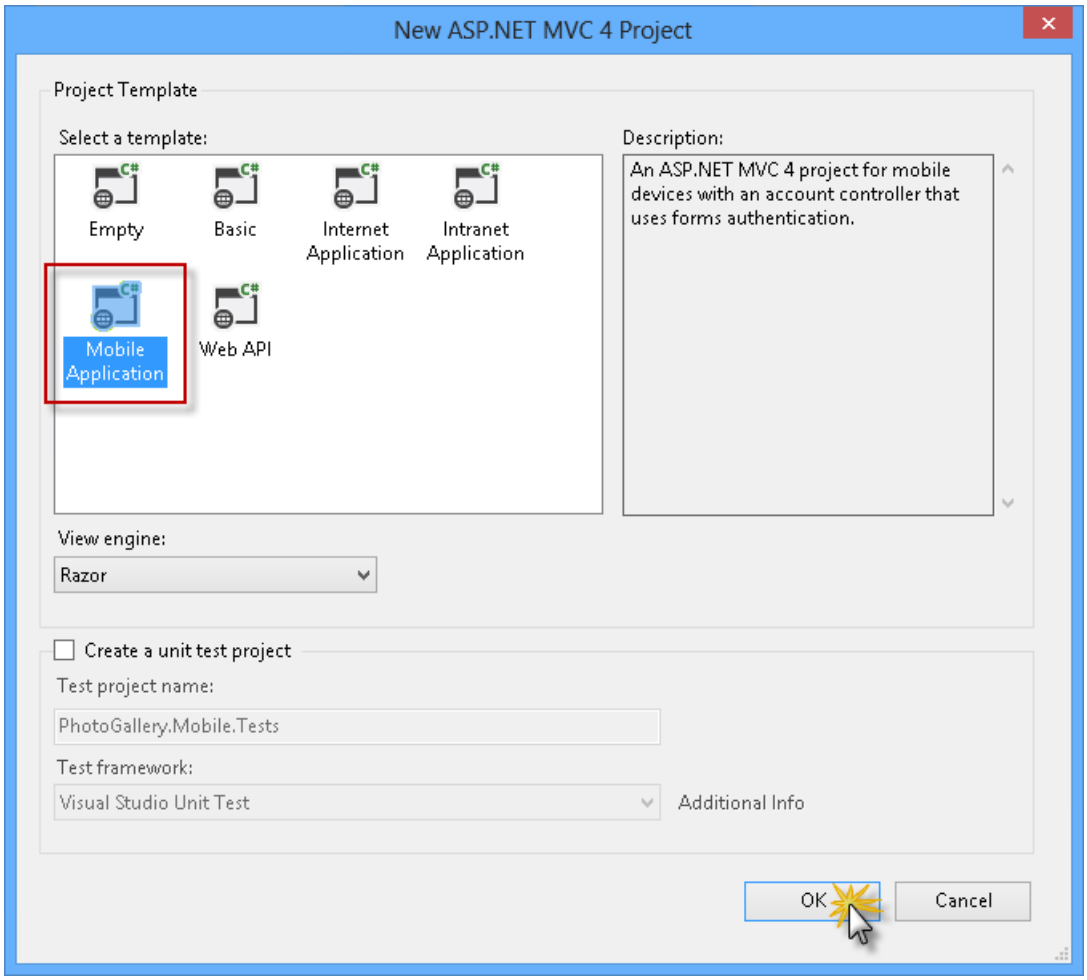
 > [!NOTE]
 > You can get more information about KnockOutJS library in this link: [[http://learn.knockoutjs.com/]
(http://learn.knockoutjs.com/)](http://learn.knockoutjs.com/). Additionally, you can learn about jQuery and
jQuery UI in [[http://docs.jquery.com/](http://docs.jquery.com/)](http://docs.jquery.com/).
```

Task 2 - Exploring the Mobile Application Template

ASP.NET MVC 4 facilitates the development of websites for mobile and tablet browsers. This template has the same application structure as the Internet Application template (notice that the controller code is practically identical), but its style was modified to render properly in touch-based mobile devices.

1. Select the **File | New | Project** menu command. In the **New Project** dialog, select the **Visual C# | Web** template on the left pane tree, and choose the **ASP.NET MVC 4 Web Application.** Name the project **PhotoGallery.Mobile**, select a location (or leave the default), select "Add to solution" and click **OK**.

2. In the **New ASP.NET MVC 4 Project** dialog, select the **Mobile Application** project template and click **OK**. Make sure you have selected Razor as the view engine.

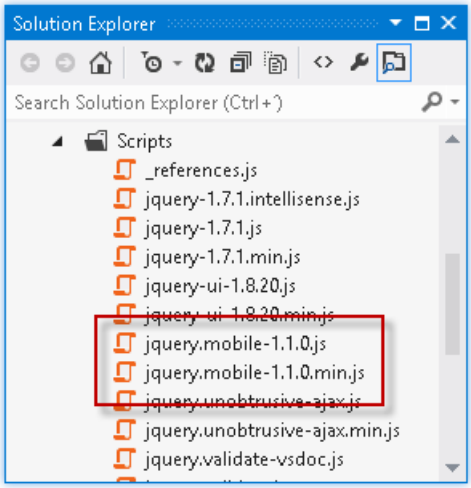


Creating a new ASP.NET MVC 4 Mobile Application

3. Now you are able to explore the solution and check out some of the new features introduced by the ASP.NET MVC 4 solution template for mobile:

   - **jQuery Mobile Library**

     The Mobile Application project template includes the jQuery Mobile library, which is an open source library for mobile browser compatibility. jQuery Mobile applies progressive enhancement to mobile browsers that support CSS and JavaScript. Progressive enhancement enables all browsers to display the basic content of a web page, while it only enables the most powerful browsers to display the rich content. The JavaScript and CSS files, included in the jQuery Mobile style, help mobile browsers to fit the content in the screen without making any change in the page markup.



jQuery mobile library included in the template

   - **HTML5 based markup**

```
@using (Html.BeginForm()) {
    @Html.ValidationSummary()

    <ul data-role="listview" data-inset="true">
        <li data-role="list-divider">Details</li>

        <li data-role="fieldcontain">
            @Html.LabelFor(m => m.UserName)
            @Html.TextBoxFor(m => m.UserName)
        </li>

        <li data-role="fieldcontain">
            @Html.LabelFor(m => m.Password)
            @Html.PasswordFor(m => m.Password)
        </li>

        <li data-role="fieldcontain">
```

```
<ul data-role="listview" data-inset="true">
    <li data-role="list-divider">Navigation</li>
    <li>@Html.ActionLink("About", "About", "Home")</li>
    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>
```

```
            <input type="submit" value="Log in" />
        </li>
    </ul>
}
```

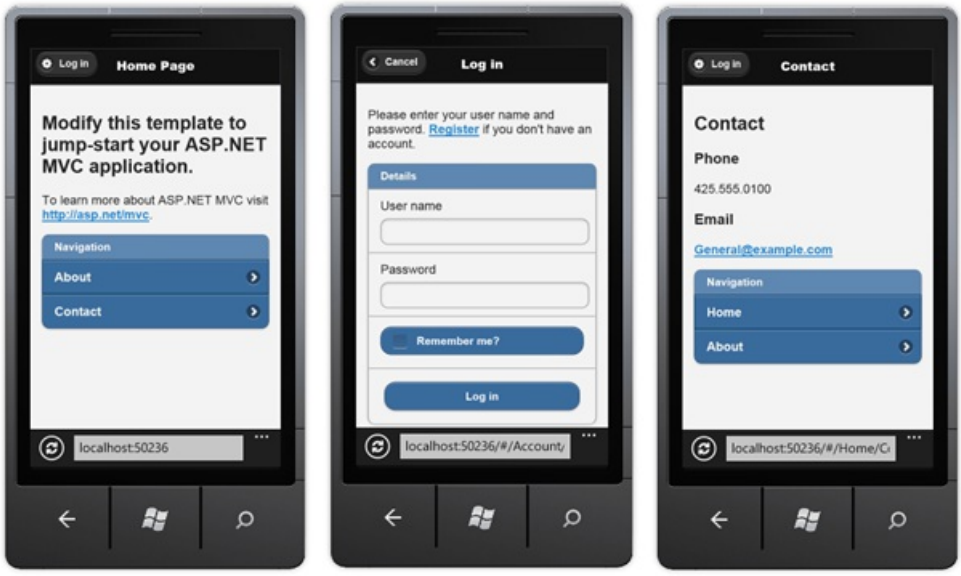
Mobile application template using HTML5 markup, (Login.cshtml and index.cshtml)

4. Press **F5** to run the solution.

5. Open the **Windows Phone 7 Emulator**.

6. In the phone start screen, open Internet Explorer. Check out the URL where the desktop application started and browse to that URL from the phone (e.g. `http://localhost:[PortNumber]/`).

7. Now you are able to enter the login page or check out the about page. Notice that the style of the website is based on the new Metro app for mobile. The ASP.NET MVC 4 project template is correctly displayed on mobile devices, making sure all the elements of the page are visible and enabled. Notice that the links on the header are big enough to be clicked or tapped.

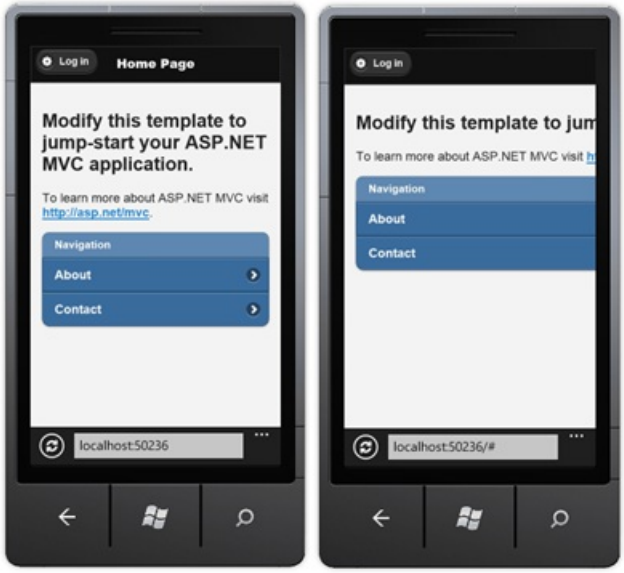


Project template pages in a mobile device

8. The new template also uses the **Viewport meta tag**. Most mobile browsers define a width for a virtual browser window or "viewport", which is larger than the actual width of the mobile device. This enables mobile browsers to display the entire web page inside the virtual display. The **Viewport meta tag** allows web developers to set the width, height and scale of the browser area on mobile devices . The ASP.NET MVC 4 template for Mobile Applications sets the viewport to the device width ("width=device-width") in the layout template (*Views\Shared_Layout.cshtml*), so that all the pages will have their viewport set to the device screen width. Notice that the Viewport meta tag will not change the default browser view.

9. Open **_Layout.cshtml**, located in the **Views | Shared** folder, and comment the Viewport meta tag. Run the application, if not already opened, and check out the differences.

```
...
 <meta charset="utf-8" />
 <title>@ViewBag.Title</title>
 @* <meta name="viewport" content="width=device-width" /> *@
...
```



The site after commenting the viewport meta tag 10. In Visual Studio, press **SHIFT + F5** to stop debugging the application. 11. Uncomment the viewport meta tag.

```
...
 <meta charset="utf-8" />
 <title>@ViewBag.Title</title>
   <meta name="viewport" content="width=device-width" />
...
```

Task 3 - Using Adaptive Rendering

In this task, you will learn another method to render a Web page correctly on mobile devices and Web browsers at the same time without any customization. You have already used Viewport meta tag with a similar purpose. Now you will meet another powerful method: *adaptive rendering*.

Adaptive rendering is a technique that uses **CSS3 media queries** to customize the style applied to a page. Media queries define conditions inside a style sheet, grouping CSS styles under a certain condition. Only when the condition is true, the style is applied to the declared objects.

The flexibility provided by the adaptive rendering technique enables any customization for displaying the site on different devices. You can define as many styles as you want on a single style sheet without writing logic code to choose the style. Therefore, it is a very neat way of adapting page styles, as it reduces the amount of duplicated code and logic for rendering purposes. On the other hand, bandwidth consumption would increase, as the size of your CSS files could grow marginally.

By using the adaptive rendering technique, your site will be **displayed properly, regardless of the browser.** However, you should consider if the bandwidth extra load is a concern.

> **NOTE**
> The basic format of a media query is: @media [Scope: all | handheld | print | projection | screen] ([property:value] and ... [property:value])

Examples of media queries: >**@media all and (max-width: 1000px) and (min-width: 700px) {}:** For all the resolutions between 700px and 1000px.

> **@media screen and (min-width: 400px) and (max-width: 700px) { ... }:** Only for screens. The resolution must be between 400 and 700px.
>
> **@media handheld and (min-width: 20em), screen and (min-width: 20em) { ... }:** For

> handhelds(mobile and devices) and screens. The minimum width must be greater than 20em.
>
> You can find more information about this on the W3C site.

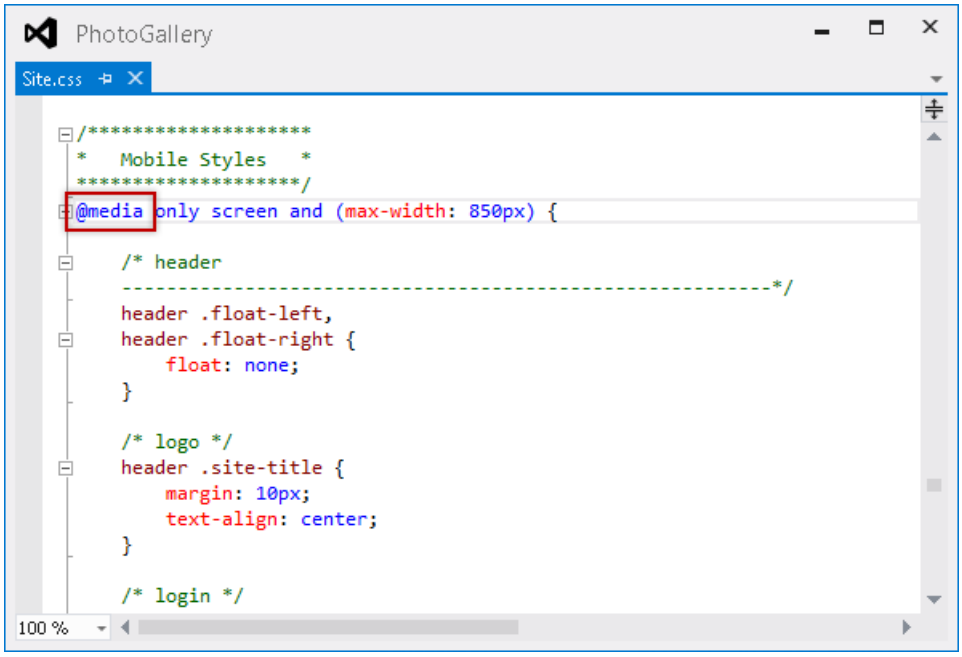
You will now explore how the adaptive rendering works, improving the readability of the ASP.NET MVC 4 default website template.

1. Open the **PhotoGallery.sln** solution you have created at Task 1 and select the **PhotoGallery** project. Press **F5** to run the solution.

2. Resize the browser's width, setting the windows to half or to less than a quarter of its original size. Notice what happens with the items in the header: Some elements will not appear in the visible area of the header.

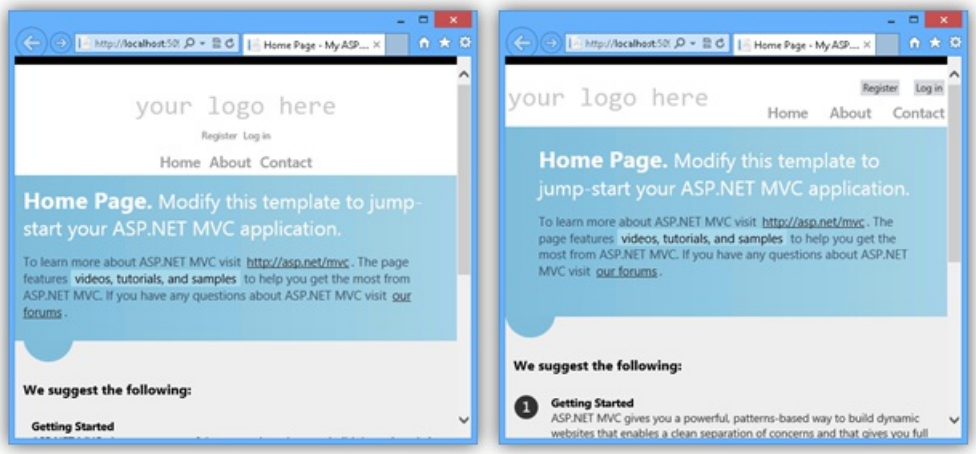
3. Open **Site.css** file from the Visual Studio Solution explorer, located in **Content** project folder. Press **CTRL + F** to open Visual Studio integrated search, and write `@media` to locate the **CSS media query**.

   The media query condition defined in this template works in this way: When the browser's window size is below **850 px**, the CSS rules applied are the ones defined inside this media block.



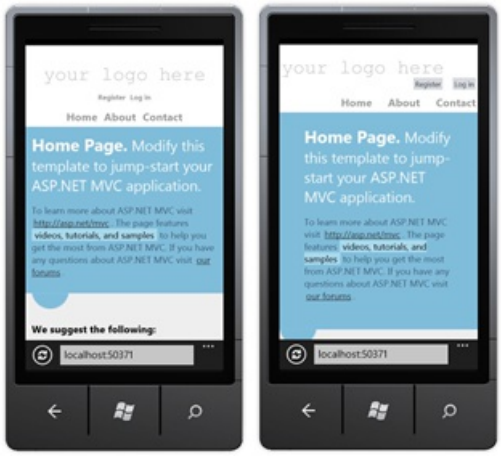
   Locating the media query

4. Replace the max-width attribute value set in 850 px with **10px**, in order to disable the adaptive rendering, and press **CTRL + S** to save the changes. Return to the browser and press **CTRL + F5** to refresh the page with the changes you have made. Notice the differences in both pages when adjusting the width of the window.



In the left, the page is applying the @media style, in the right, the style is omitted

Now, let's check out what happens on mobile devices:



In the left, the page is applying the @media style, in the right, the style is omitted

Although you will notice that the changes when the page is rendered in a Web browser are not very significant, when using a mobile device the differences become more obvious. On the left side of the image, we can see that the custom style improved the readability.

Adaptive rendering can be used in many more scenarios, making it easier to apply conditional styling to a Web site and solving common style issues with a neat approach.

The Viewport meta tag and CSS media queries are not specific to ASP.NET MVC 4, so you can take advantage of these features in any web application.

5. In Visual Studio, press **SHIFT + F5** to stop debugging the application.

Exercise 2: Creating the Photo Gallery Web Application

In this exercise, you will work on a Photo Gallery application to display photos. You will start with the ASP.NET MVC 4 project template, and then you will add a feature to retrieve photos from a service and display them in the home page.

In the following exercise, you will update this solution to enhance the way it is displayed on mobile devices.

Task 1 - Creating a Mock Photo Service

In this task, you will create a mock of the photo service to retrieve the content that will be displayed in the gallery. To do this, you will add a new controller that will simply return a JSON file with the data of each photo.

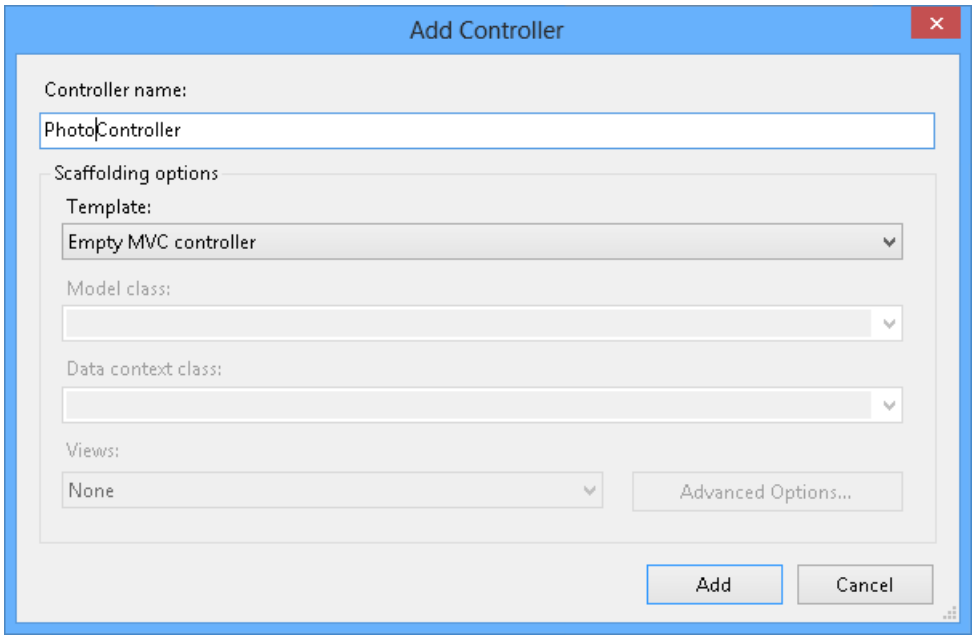
1. Open **Visual Studio** if not already opened.

2. Select the **File | New | Project** menu command. In the **New Project** dialog, select the **Visual C# | Web** template on the left pane tree, and choose **ASP.NET MVC 4 Web Application.** Name the project **PhotoGallery**, select a location (or leave the default) and click **OK**. Alternatively, you can continue working from your existing ASP.NET MVC 4 **Internet Application** solution from **Exercise 1** and skip the next step.

3. In the **New ASP.NET MVC 4 Project** dialog box, select the **Internet Application** project template and click **OK**. Make sure you have Razor selected as the View Engine.

4. In the **Solution Explorer**, right-click the **App_Data** folder of your project, and select **Add | Existing Item**. Browse to the **Source\Assets\App_Data** folder of this lab and add the **Photos.json** file.

5. Create a new controller with the name **PhotoController**. To do this, right-click on the **Controllers** folder, go to **Add** and select **Controller.** Complete the controller name, leave the **Empty MVC controller** template and click **Add**.



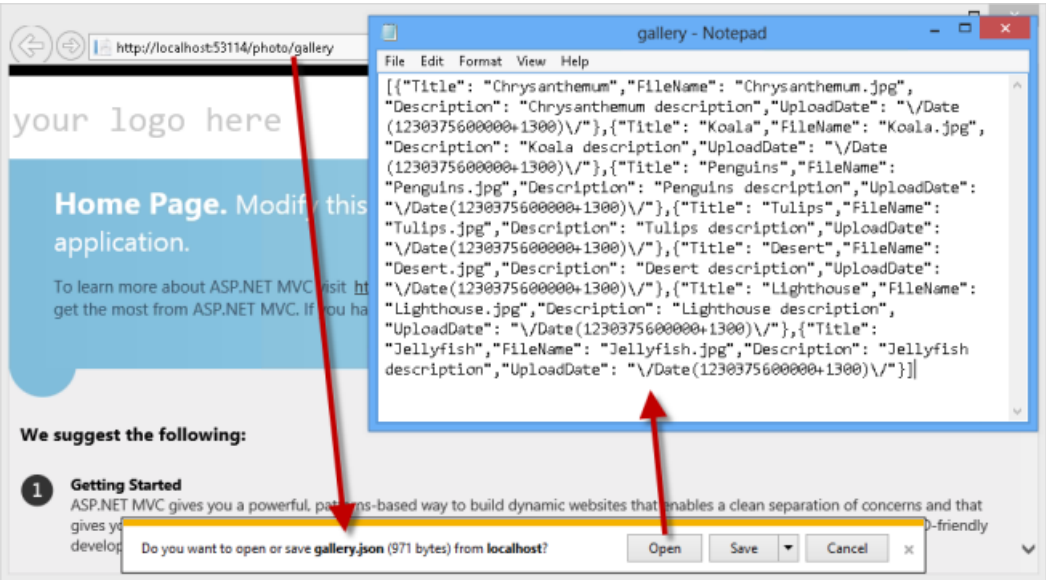
Adding the PhotoController

6. Replace the **Index** method with the following **Gallery** action, and return the content from the JSON file you have recently added to the project.

   (Code Snippet - *ASP.NET MVC 4 Lab - Ex02 - Gallery Action*)

```
public class PhotoController : Controller
{
   public ActionResult Gallery()
   {
      return this.File("~/App_Data/Photos.json", "application/json");
   }
}
```

7. Press **F5** to run the solution, and then browse to the following URL in order to test the mocked photo service: `http://localhost:[port]/photo/gallery` (the [port] value depends on the current port where the application was launched). The request to this URL should retrieve the content of the **Photos.json** file.



Testing the mocked photo service

In a real implementation you could use ASP.NET Web API to implement the Photo Gallery service. ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers

and mobile devices. ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework.

Task 2 - Displaying the Photo Gallery

In this task, you will update the Home page to show the photo gallery by using the mocked service you created in the first task of this exercise. You will add model files and update the gallery views.

1. In Visual Studio, press **SHIFT** + **F5** to stop debugging the application.
2. Create the **Photo** class in the **Models** folder. To do this, right-click on the **Models** folder, select **Add** and click **Class**. Then, set the name to **Photo.cs** and click **Add**.
3. Add the following members to the **Photo** class.

   (Code Snippet - *ASP.NET MVC 4 Lab - Ex02 - Photo model*)

```
public class Photo
{
    public string Title { get; set; }

    public string FileName { get; set; }

    public string Description { get; set; }

    public DateTime UploadDate { get; set; }
}
```

4. Open the **HomeController.cs** file from the **Controllers** folder.
5. Add the following using statements.

   (Code Snippet - *ASP.NET MVC 4 Lab - Ex02 - HomeController Usings*)

```
using System.Net.Http;
using System.Web.Script.Serialization;
using Newtonsoft.Json;
using PhotoGallery.Models;
```

6. Update the **Index** action to use **HttpClient** to retrieve the gallery data, and then use the **JavaScriptSerializer** to deserialize it to the view model.

   (Code Snippet - *ASP.NET MVC 4 Lab - Ex02 - Index Action*)

```
public ActionResult Index()
{
    var client = new HttpClient();
    var response = client.GetAsync(Url.Action("gallery", "photo", null, Request.Url.Scheme)).Result;
    var value = response.Content.ReadAsStringAsync().Result;

    var result = JsonConvert.DeserializeObject<List<Photo>>(value);

    return View(result);
}
```

7. Open the **Index.cshtml** file located under the **Views\Home** folder and replace all the content with the following code.

   This code loops through all the photos retrieved from the service and displays them into an unordered list.

   (Code Snippet - *ASP.NET MVC 4 Lab - Ex02 - Photo List*)

```
@model List<PhotoGallery.Models.Photo>
@{
    ViewBag.Title = "Photo Gallery";
}

<ul class="thumbnails">
    @foreach (var photo in Model)
    {
        <li class="item">
            <a href="@Url.Content("~/photos/" + photo.FileName)">
                <img alt="@photo.Description" src="@Url.Content("~/photos/" + photo.FileName)"
class="thumbnail-border" width="180" />
            </a>
                <span class="image-overlay">@photo.Title</span>
        </li>
    }
</ul>
```

8. In the **Solution Explorer**, right-click the **Content** folder of your project, and select **Add | Existing Item**. Browse to the **Source\Assets\Content** folder of this lab and add the **Site.css** file. You will have to confirm its replacement. If you have the **Site.css** file open, you will have to confirm to reload the file also.

9. Open File Explorer and copy the entire **Photos** folder located under the **Source\Assets** folder of this lab to the root folder of your project in Solution Explorer.

10. Run the application. You should now see the home page displaying the photos in the gallery.

Photo Gallery

11. In Visual Studio, press **SHIFT** + **F5** to stop debugging the application.

Exercise 3: Adding support for mobile devices

One of the key updates in ASP.NET MVC 4 is the support for mobile development. In this exercise, you will explore ASP.NET MVC 4 new features for mobile applications by extending the PhotoGallery solution you have created in the previous exercise.

Task 1 - Installing jQuery Mobile in an ASP.NET MVC 4 Application

1. Open the **Begin** solution located at **Source/Ex3-MobileSupport/Begin/** folder. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.

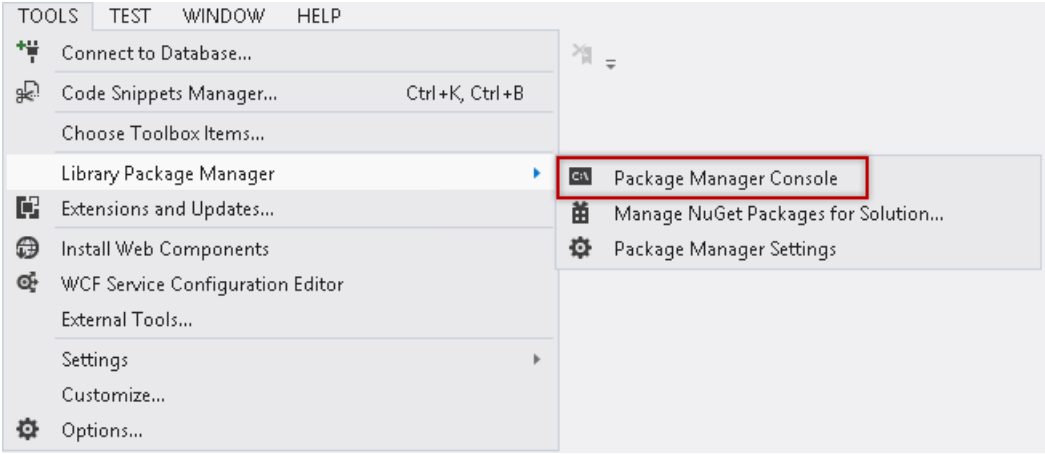
   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

c. Finally, build the solution by clicking **Build | Build Solution**.

> **NOTE**
>
> One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Open the **Package Manager Console** by clicking the **Tools** > **NuGet Package Manager** > **Package Manager Console** menu option.



Opening the NuGet Package Manager Console

3. In the Package Manager Console run the following command to install the **jQuery.Mobile.MVC** package.

jQuery Mobile is an open source library for building touch-optimized web UI. The jQuery.Mobile.MVC NuGet package includes helpers to use jQuery Mobile with an ASP.NET MVC 4 application.

> **NOTE**
>
> By running the following command, you will be downloading the jQuery.Mobile.MVC library from Nuget.

PM

```
Install-Package jQuery.Mobile.MVC
```

This command installs jQuery Mobile and some helper files, including the following:

- **Views/Shared/_Layout.Mobile.cshtml**: is a jQuery Mobile-based layout optimized for a smaller screen. When the website receives a request from a mobile browser, it will replace the original layout (_Layout.cshtml) with this one.
- A view-switcher component: consists of the **Views/Shared/_ViewSwitcher.cshtml** partial view and the **ViewSwitcherController.cs** controller. This component will show a link on mobile browsers to enable users to switch to the desktop version of the page.

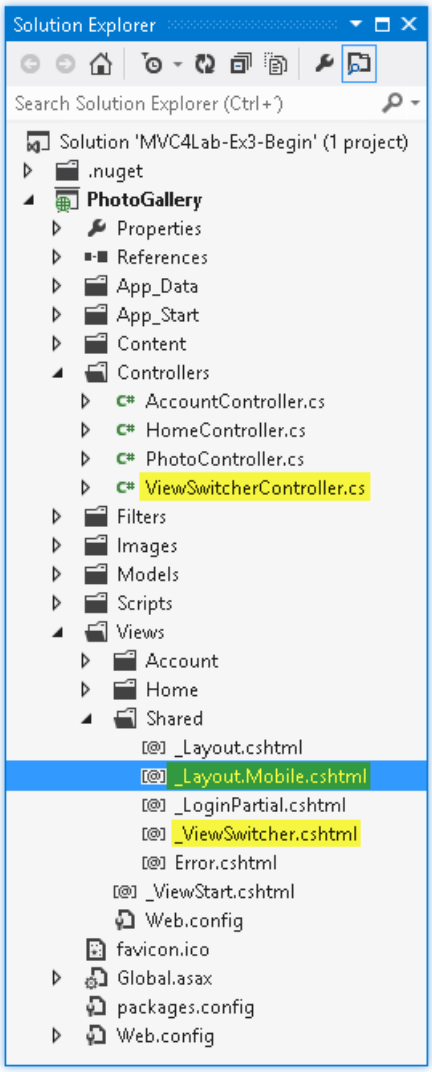


Photo Gallery project with mobile support

4. Register the Mobile bundles. To do this, open the **Global.asax.cs** file and add the following line.

   (Code Snippet - *ASP.NET MVC 4 Lab - Ex03 - Register Mobile Bundles*)

```
protected void Application_Start()
{
     AreaRegistration.RegisterAllAreas();

     WebApiConfig.Register(GlobalConfiguration.Configuration);
     FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
     RouteConfig.RegisterRoutes(RouteTable.Routes);
     BundleConfig.RegisterBundles(BundleTable.Bundles);
     BundleMobileConfig.RegisterBundles(BundleTable.Bundles);
     AuthConfig.RegisterAuth();
}
```

5. Run the application using a desktop web browser.

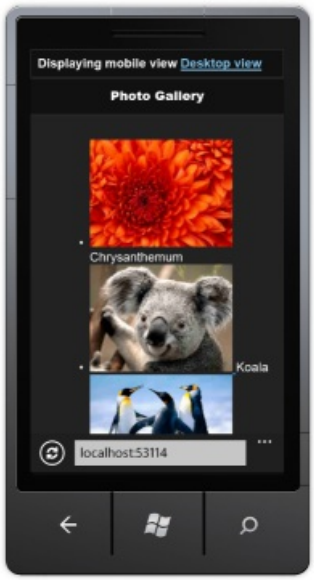
6. Open the **Windows Phone 7 Emulator**, located in **Start Menu | All Programs | Windows Phone SDK 7.1 | Windows Phone Emulator.**

7. In the phone start screen, open Internet Explorer. Check out the URL where the application started and browse to that URL with the phone browser (e.g. `http://localhost:[PortNumber]/`).

   You will notice that your application will look different in the Windows Phone emulator, as the jQuery.Mobile.MVC has created new assets in your project that show views optimized for mobile devices.

Notice the message at the top of the phone, showing the link that switches to the Desktop view. Additionally, the **_Layout.Mobile.cshtml** layout that was created by the package you have installed is including a different layout in the application.

> **NOTE**
>
> So far, there is no link to get back to mobile view. It will be included in later versions.



Mobile view of the Photo Gallery Home page

8. In Visual Studio, press **SHIFT + F5** to stop debugging the application.

Task 2 - Creating Mobile Views

In this task, you will create a mobile version of the index view with content adapted for better appearance in mobile devices.

1. Copy the **Views\Home\Index.cshtml** view and paste it to create a copy, rename the new file to **Index.Mobile.cshtml**.

2. Open the new created **Index.Mobile.cshtml** view and replace the existing <ul> tag with this code. By doing this, you will be updating the <ul> tag with jQuery Mobile data annotations to use the mobile themes from jQuery.

```
<ul data-role="listview" data-inset="true" data-filter="true">
```

> **NOTE**
>
> Notice that:
>
> - The **data-role** attribute set to **listview** will render the list using the listview styles.
> - The **data-inset** attribute set to true will show the list with rounded border and margin.
> - The **data-filter** attribute set to **true** will generate a search box.
>
> You can learn more about jQuery Mobile conventions in the project documentation: http://jquerymobile.com/demos/1.1.1/

3. Press **CTRL + S** to save the changes.

4. Switch to the **Windows Phone Emulator** and refresh the site. Notice the new look and feel of the gallery list, as well as the new search box located on the top. Then, type a word in the search box (for instance, **Tulips**) to start a search in the photo gallery.

Gallery using listview style with filtering

To summarize, you have used the View Mobilizer recipe to create a copy of the Index view with the "mobile" suffix. This suffix indicates to ASP.NET MVC 4 that every request generated from a mobile device will use this copy of the index. Additionally, you have updated the mobile version of the Index view to use jQuery Mobile for enhancing the site look and feel in mobile devices.

5. Go back to Visual Studio and open **Site.Mobile.css** located under the **Content** folder.

6. Fix the positioning of the photo title to make it show at the right side of the image. To do this, add the following code to the **Site.Mobile.css** file.

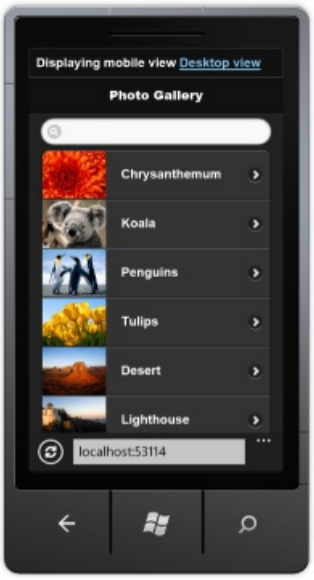
   CSS

```
.ui-li .ui-btn-inner a.ui-link-inherit, .ui-li-static.ui-li {
    padding: 0px !important;
}

li.item span.image-overlay
{
    position:relative;
    left:100px;
    top:-40px;
    height:0px;
    display:block;
}
```

7. Press **CTRL + S** to save the changes.

8. Switch back to the **Windows Phone Emulator** and refresh the site. Notice the photo title is properly positioned now.



Title positioned on the right side of the image

Task 3 - jQuery Mobile Themes

Every layout and widget in jQuery Mobile is designed around a new object-oriented CSS framework that makes it possible to apply a complete unified visual design theme to sites and applications.

jQuery Mobile's default Theme includes 5 swatches that are given letters (a, b, c, d, e) for quick reference.

In this task, you will update the mobile layout to use a different theme than the default.

1. Switch back to Visual Studio.
2. Open the **_Layout.Mobile.cshtml** file located in **Views\Shared**.
3. Find the div element with the data-role set to "page" and update the **data-theme** to "**e**".

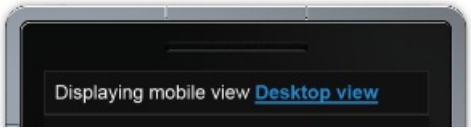
```
<div data-role="page" data-theme="e">
```

4. Press **CTRL + S** to save the changes.
5. Refresh the site in the **Windows Phone Emulator** and notice the new colors scheme.

Mobile layout with a different color scheme

Task 4 - Using the View-Switcher Component and the Browser Overriding Features

A convention for mobile-optimized web pages is to add a link whose text is something like Desktop view or Full site mode that lets users switch to a desktop version of the page. The jQuery.Mobile.MVC package includes a sample **view-switcher** component for this purpose used in the **_Layout.Mobile.cshtml** view.



Link to switch to Desktop View

The view switcher uses a new feature called **Browser Overriding**. This feature lets your application treat requests as if they were coming from a different browser (user agent) than the one they are actually coming from.

In this task, you will explore the sample implementation of a view-switcher added by jQuery.Mobile.MVC and the new browser overriding features in ASP.NET MVC 4.

1. Switch back to Visual Studio.
2. Open the **_Layout.Mobile.cshtml** view located under the **Views\Shared** folder and notice the view-switcher component being referenced as a partial view.

```
_Layout.Mobile.cshtml
        </script>
    </head>
<body>

    <div data-role="page" data-theme="e">
        @Html.Partial("_ViewSwitcher")

        <div data-role="header">
            <h1>@ViewBag.Title</h1>
        </div>

        <div data-role="content">
            @RenderSection("featured", false)
            @RenderBody()
        </div>

    </div>
```

Mobile layout using View-Switcher component

3. Open the **_ViewSwitcher.cshtml** partial view.

   The partial view uses the new method **ViewContext.HttpContext.GetOverriddenBrowser()** to determine the origin of the web request and show the corresponding link to switch either to the Desktop or Mobile views.

   The **GetOverriddenBrowser** method returns an **HttpBrowserCapabilitiesBase** instance that corresponds to the user agent currently set for the request (actual or overridden). You can use this value to get properties such as **IsMobileDevice**.

```
_ViewSwitcher.cshtml   _Layout.Mobile.cshtml
@if (Request.Browser.IsMobileDevice && Request.HttpMethod == "GET")
{
    <div class="view-switcher ui-bar-a">
        @if (ViewContext.HttpContext.GetOverriddenBrowser().IsMobileDevice)
        {
            @: Displaying mobile view
            @Html.ActionLink("Desktop view", "SwitchView", "ViewSwitcher", new { mobile = false, returnUrl =
        }
        else
        {
            @: Displaying desktop view
            @Html.ActionLink("Mobile view", "SwitchView", "ViewSwitcher", new { mobile = true, returnUrl = Re
        }
    </div>
}
```

ViewSwitcher partial view

4. Open the **ViewSwitcherController.cs** class located in the **Controllers** folder. Check out that SwitchView action is called by the link in the ViewSwitcher component, and notice the new HttpContext methods.

   - The **HttpContext.ClearOverriddenBrowser()** method removes any overridden user agent for the current request.

   - The **HttpContext.SetOverriddenBrowser()** method overrides the request's actual user agent value using the specified user agent.

```
ViewSwitcherController.cs
PhotoGallery.Controllers.ViewSwitcherController        SwitchView(bool mobile, string returnUrl)
using System.Web.Mvc;
using System.Web.WebPages;

namespace PhotoGallery.Controllers
{
    public class ViewSwitcherController : Controller
    {
        public RedirectResult SwitchView(bool mobile, string returnUrl) {
            if (Request.Browser.IsMobileDevice == mobile)
                HttpContext.ClearOverriddenBrowser();
            else
                HttpContext.SetOverriddenBrowser(mobile ? BrowserOverride.Mobile : BrowserOverride.Desktop);

            return Redirect(returnUrl);
        }
    }
}
```

   ViewSwitcher Controller

   Browser Overriding is a core feature of ASP.NET MVC 4, which is also available even if you do not install the jQuery.Mobile.MVC package. However, this feature affects only view, layout, and partial-view, and it does not affect any of the features that depend on the Request.Browser object.

Task 5 - Adding the View-Switcher in the Desktop View

In this task, you will update the desktop layout to include the view-switcher. This will allow mobile users to go back to the mobile view when browsing the desktop view.

1. Refresh the site in the **Windows Phone Emulator**.

2. Click on the **Desktop view** link at the top of the gallery. Notice there is no view-switcher in the desktop view to allow you return to the mobile view.

3. Go back to Visual Studio and open the **_Layout.cshtml** view.

4. Find the login section and insert a call to render the **_ViewSwitcher** partial view below the **_LogOnPartial** partial view. Then, press **CTRL + S** to save the changes.

```
<div class="float-right">
    <section id="login">
        @Html.Partial("_LogOnPartial")

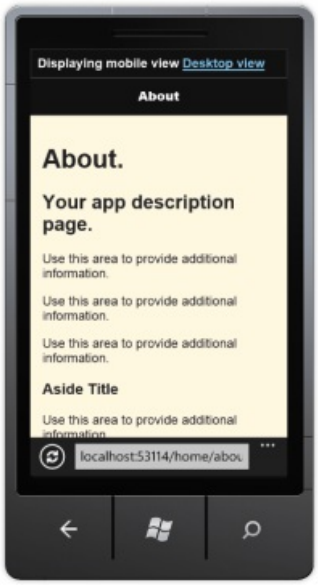
        @Html.Partial("_ViewSwitcher")
    </section>
    <nav>
```

5. Press **CTRL + S** to save the changes.

6. Refresh the page in the Windows Phone Emulator and double-click the screen to zoom in. Notice that the home page now shows the **Mobile view** link that switches from mobile to desktop view.

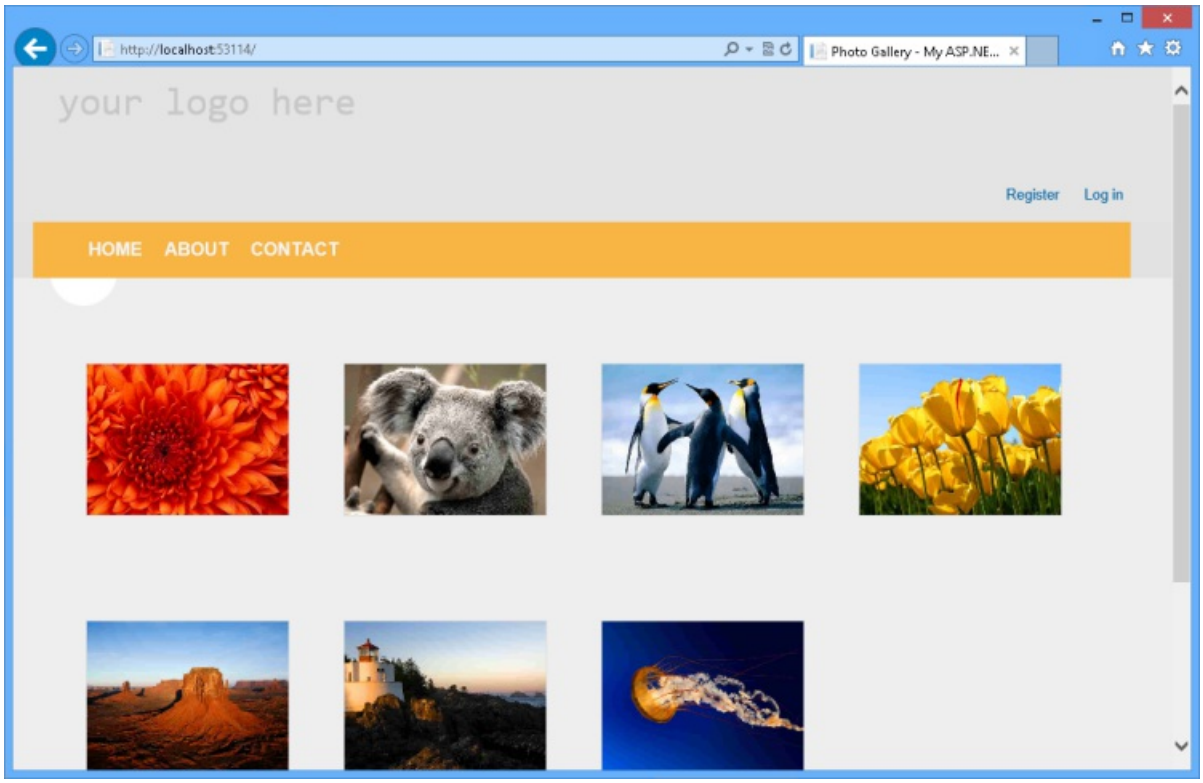
View Switcher rendered in desktop view

7. Switch to the Mobile view again and browse to **About** page (http://localhost[port]/Home/About). Notice that, even if you haven't created an About.Mobile.cshtml view, the About page is displayed using the mobile layout (_Layout.Mobile.cshtml).



About page

8. Finally, open the site in a desktop Web browser. Notice that none of the previous updates has affected the desktop view.



PhotoGallery desktop view

Task 6 - Creating New Display Modes

The new display modes feature lets an application select views depending on the browser that is generating the request. For example, if a desktop browser requests the Home page, the application will return the **Views\Home\Index.cshtml** template. Then, if a mobile browser requests the Home page, the application will return the **Views\Home\Index.mobile.cshtml** template.

In this task, you will create a customized layout for iPhone devices, and you will have to simulate requests from iPhone devices. To do this, you can use either an iPhone emulator/simulator (like Electric Mobile Simulator) or a browser with add-ons that modify the user agent. For instructions on how to set the user agent string in an Safari browser to emulate an iPhone, see How to let Safari pretend it's IE in David Alison's blog.

Notice that this task is optional and you can continue throughout the lab without executing it.

1. In Visual Studio, press **SHIFT** + **F5** to stop debugging the application.
2. Open **Global.asax.cs** and add the following using statement.

```
using System.Web.WebPages;
```

3. Add the following highlighted code into the Application_Start method.

   (Code Snippet - *ASP.NET MVC 4 Lab - Ex03 - iPhone DisplayMode*)

```
protected void Application_Start()
{
    // ...

    DisplayModeProvider.Instance.Modes.Insert(0, new DefaultDisplayMode("iPhone")
    {
        ContextCondition = context =>
            context.Request.UserAgent != null &&
            context.Request.UserAgent.IndexOf("iPhone", StringComparison.OrdinalIgnoreCase) >= 0
    });
}
```

You have registered a new **DefaultDisplayMode named "iPhone"**, within the static **DisplayModeProvider.Instance.Modes** static list, that will be matched against each incoming request. If the incoming request contains the string "iPhone", ASP.NET MVC will find the views whose name contain the "iPhone" suffix. The 0 parameter indicates how specific is the new mode; for instance, this view is more specific than the general ".mobile" rule that matches requests from mobile devices.

After this code runs, when an iPhone browser generates a request, your application will use the **Views\Shared_Layout.iPhone.cshtml** layout you will create in the next steps.

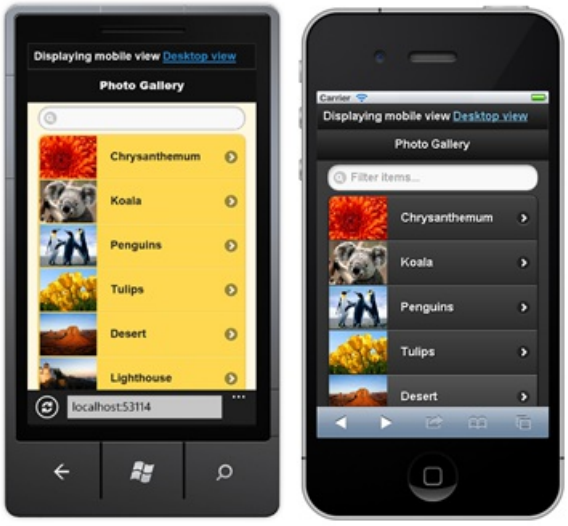
> **NOTE**
> This way of testing the request for iPhone has been simplified for demo purposes and might not work as expected for every iPhone user agent string (for example test is case sensitive).

4. Create a copy of the **_Layout.Mobile.cshtml** file in the **Views\Shared** folder and rename the copy to "**_Layout.iPhone.cshtml**".
5. Open **_Layout.iPhone.cshtml** you created in the previous step.
6. Find the div element with the data-role attribute set to **page** and change the **data-theme** attribute to "**a**".

```
<body>
    <div data-role="page" data-theme="a">
        @Html.Partial("_ViewSwitcher")
...
```

Now you have 3 layouts in your ASP.NET MVC 4 application:

1. **_Layout.cshtml**: default layout used for desktop browsers.
2. **_Layout.mobile.cshtml**: default layout used for mobile devices.
3. **_Layout.iPhone.cshtml**: specific layout for iPhone devices, using a different color scheme to differentiate from _Layout.mobile.cshtml.
4. Press **F5** to run the application and browse the site in the **Windows Phone Emulator**.
5. Open an **iPhone simulator** (see Appendix C for instructions on how to install and configure an iPhone simulator), and browse to the site too. Notice that each phone is using the specific template.



Using different views for each mobile device

Exercise 4: Using Asynchronous Controllers

Microsoft .NET Framework 4.5 introduces new language features in C# and Visual Basic to provide a new foundation for asynchrony in .NET programming. This new foundation makes asynchronous programming similar to - and about as straightforward as - synchronous programming. You are now able to write asynchronous action methods in ASP.NET MVC 4 by using the **AsyncController** class. You can use asynchronous action methods for long-running, non-CPU bound requests. This avoids blocking the Web server from performing work while the request is being processed. The AsyncController class is typically used for long-running Web service calls.

This exercise explains the basics of asynchronous operation in ASP.NET MVC 4. If you want a deeper dive, you can check out the following article: [https://msdn.microsoft.com/library/ee728598%28v=vs.100%29.aspx] (https://msdn.microsoft.com/library/ee728598%28v=vs.100%29.aspx)

Task 1 - Implementing an Asynchronous Controller

1. Open the **Begin** solution located at **Source/Ex4-Async/Begin/** folder. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.

   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Open the **HomeController.cs** class from the **Controllers** folder.

3. Add the following using statement.

```
using System.Threading.Tasks;
```

4. Update the **HomeController** class to inherit from **AsyncController**. Controllers that derive from AsyncController enable ASP.NET to process asynchronous requests, and they can still service synchronous

action methods.

```
public class HomeController : AsyncController
{
```

5. Add the **async** keyword to the **Index** method and make it return the type **Task<ActionResult>**.

```
public async Task<ActionResult> Index()
{
    ...
```

> **NOTE**
>
> The **async** keyword is one of the new keywords the .NET Framework 4.5 provides; it tells the compiler that this method contains asynchronous code. A **Task** object represents an asynchronous operation that may complete at some point in the future.

6. Replace the **client.GetAsync()** call with the full async version using await keyword as shown below.

   (Code Snippet - *ASP.NET MVC 4 Lab - Ex04 - GetAsync*)

```
public async Task<ActionResult> Index()
{
    var client = new HttpClient();
    var response = await client.GetAsync(Url.Action("gallery", "photo", null, Request.Url.Scheme));
    ...
```

> **NOTE**
>
> In the previous version, you were using the **Result** property from the **Task** object to block the thread until the result is returned (sync version).
>
> Adding the **await** keyword tells the compiler to asynchronously wait for the task returned from the method call. This means that the rest of the code will be executed as a callback only after the awaited method completes. Another thing to notice is that you do not need to change your try-catch block in order to make this work: the exceptions that happen in background or in foreground will still be caught without any extra work using a handler provided by the framework.

7. Change the code to continue with the asynchronous implementation by replacing the lines with the new code as shown below

   (Code Snippet - *ASP.NET MVC 4 Lab - Ex04 - ReadAsStringAsync*)

```
public async Task<ActionResult> Index()
{
    var client = new HttpClient();
    var response = await client.GetAsync(Url.Action("gallery", "photo", null, Request.Url.Scheme));
    var value = await response.Content.ReadAsStringAsync();
    var result = await JsonConvert.DeserializeObjectAsync<List<Photo>>(value);

    return View(result);
}
```

8. Run the application. You will notice no major changes, but your code will not block a thread from the thread pool making a better usage of the server resources and improving performance.

> **NOTE**
> You can learn more about the new asynchronous programming features in the lab "**Asynchronous Programming in .NET 4.5 with C# and Visual Basic**" included in the Visual Studio Training Kit.

Task 2 - Handling Time-Outs with Cancellation Tokens

Asynchronous action methods that return Task instances can also support time-outs. In this task, you will update the Index method code to handle a time-out scenario using a cancellation token.

1. Go back to Visual Studio and press **SHIFT + F5** to stop debugging.
2. Add the following using statement to the **HomeController.cs** file.

```
using System.Threading;
```

3. Update the Index action to receive a **CancellationToken** argument.

```
public async Task<ActionResult> Index(CancellationToken cancellationToken)
{
    ...
```

4. Update the **GetAsync** call to pass the cancellation token.

   (Code Snippet - *ASP.NET MVC 4 Lab - Ex04 - SendAsync with CancellationToken*)

```
public async Task<ActionResult> Index(CancellationToken cancellationToken)
{
    var client = new HttpClient();
    var response = await client.GetAsync(Url.Action("gallery", "photo", null, Request.Url.Scheme),
cancellationToken);
    var value = await response.Content.ReadAsStringAsync();
    var result = await JsonConvert.DeserializeObjectAsync<List<Photo>>(value);

    return View(result);
}
```

5. Decorate the *Index* method with an **AsyncTimeout** attribute set to 500 milliseconds and a **HandleError** attribute configured to handle **TaskCanceledException** by redirecting to a **TimedOut** view.

   (Code Snippet - *ASP.NET MVC 4 Lab - Ex04 - Attributes*)

```
[AsyncTimeout(500)]
[HandleError(ExceptionType = typeof(TimeoutException), View = "TimedOut")]
public async Task<ActionResult> Index(CancellationToken cancellationToken)
{
```

6. Open the **PhotoController** class and update the **Gallery** method to delay the execution 1000 milliseconds (1 second) to simulate a long running task.

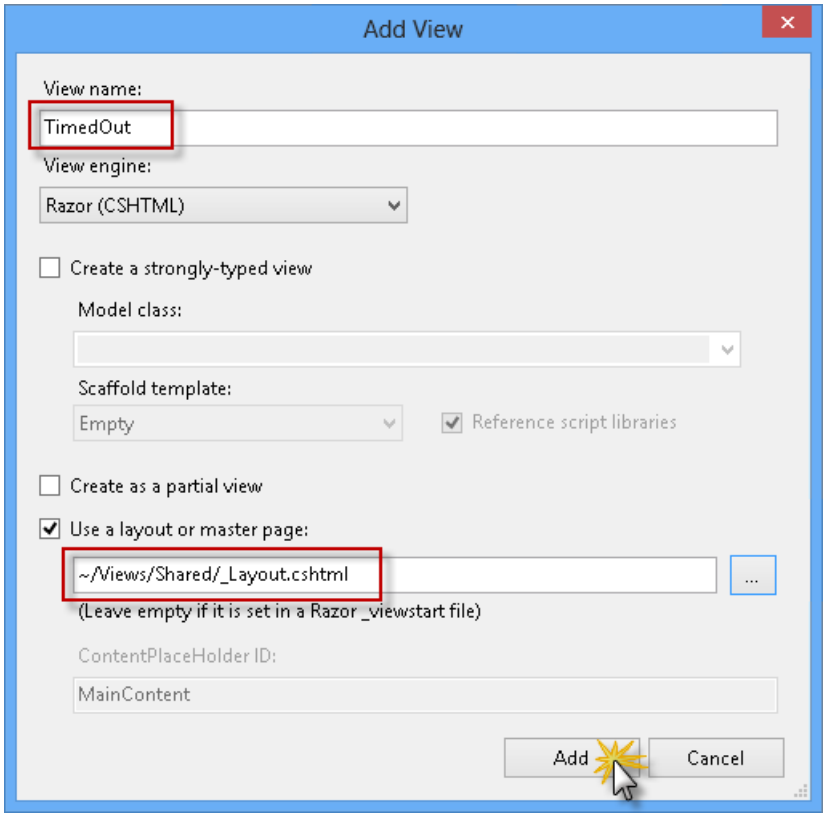
```
public ActionResult Gallery()
{
    System.Threading.Thread.Sleep(1000);

    return this.File("~/App_Data/Photos.json", "application/json");
}
```

7. Open the **Web.config** file and enable custom errors by adding the following element.

```
<system.web>
  <customErrors mode="On"></customErrors>
  ...
```

8. Create a new view in **Views\Shared** named **TimedOut** and use the default layout. In the Solution Explorer, right-click the **Views\Shared** folder and select **Add | View**.



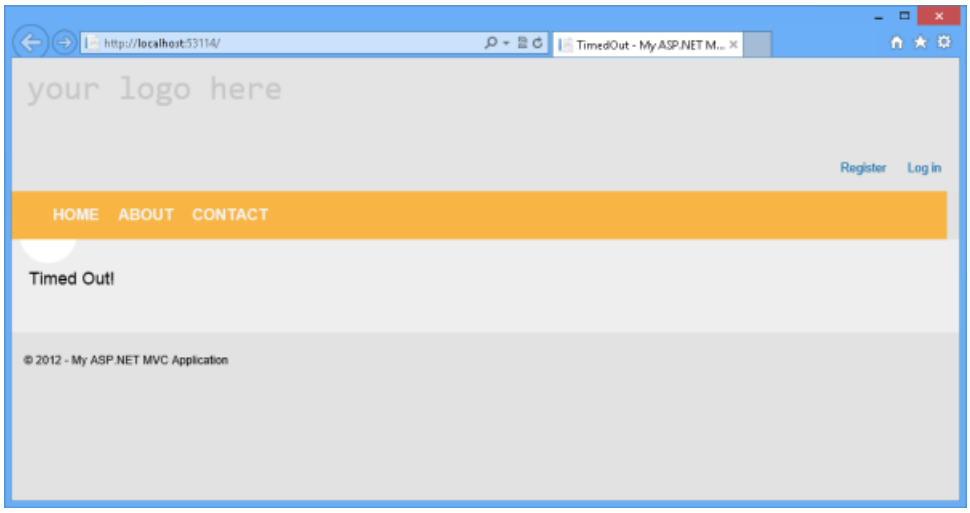
Using different views for each mobile device

9. Update the **TimedOut** view content as shown below.

```
@{
    ViewBag.Title = "TimedOut";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Timed Out!</h2>
```

10. Run the application and navigate to the root URL. As you have added a **Thread.Sleep** of 1000 milliseconds, you will get a time-out error, generated by the **AsyncTimeout** attribute and catch by the **HandleError** attribute.



Time-out exception handled

> **NOTE**
>
> Additionally, you can deploy this application to Windows Azure Web Sites following Appendix D: Publishing an ASP.NET MVC 4 Application using Web Deploy.

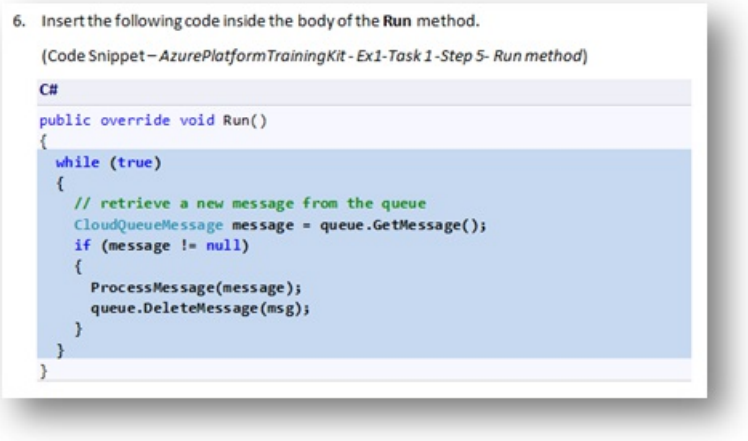
Summary

In this hands-on-lab, you've observed some of the new features resident in ASP.NET MVC 4. The following concepts have been discussed:

- Take advantage of the enhancements to the ASP.NET MVC project templates-including the new mobile application project template
- Use the HTML5 viewport attribute and CSS media queries to improve the display on mobile devices
- Use jQuery Mobile for progressive enhancements and for building touch-optimized web UI
- Create mobile-specific views
- Use the view-switcher component to toggle between mobile and desktop views in the application
- Create asynchronous controllers using task support

Appendix A: Using Code Snippets

With code snippets, you have all the code you need at your fingertips. The lab document will tell you exactly when you can use them, as shown in the following figure.



6. Insert the following code inside the body of the **Run** method.

(Code Snippet – *AzurePlatformTrainingKit - Ex1-Task 1 -Step 5- Run method*)

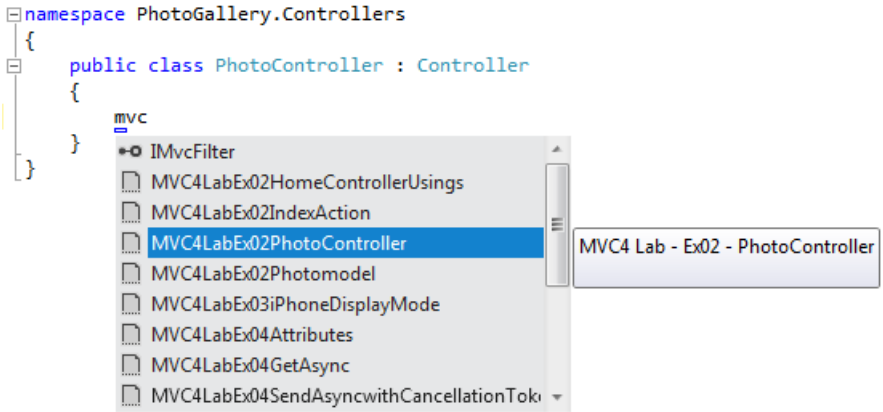
C#

```
public override void Run()
{
  while (true)
  {
    // retrieve a new message from the queue
    CloudQueueMessage message = queue.GetMessage();
    if (message != null)
    {
      ProcessMessage(message);
      queue.DeleteMessage(msg);
    }
  }
}
```

Using Visual Studio code snippets to insert code into your project

To add a code snippet using the keyboard (C# only)

1. Place the cursor where you would like to insert the code.
2. Start typing the snippet name (without spaces or hyphens).
3. Watch as IntelliSense displays matching snippets' names.
4. Select the correct snippet (or keep typing until the entire snippet's name is selected).
5. Press the Tab key twice to insert the snippet at the cursor location.



Start typing the snippet name

```
static void Main(string[] args)
{
    AzureTrainingKitServiceBus
}
```

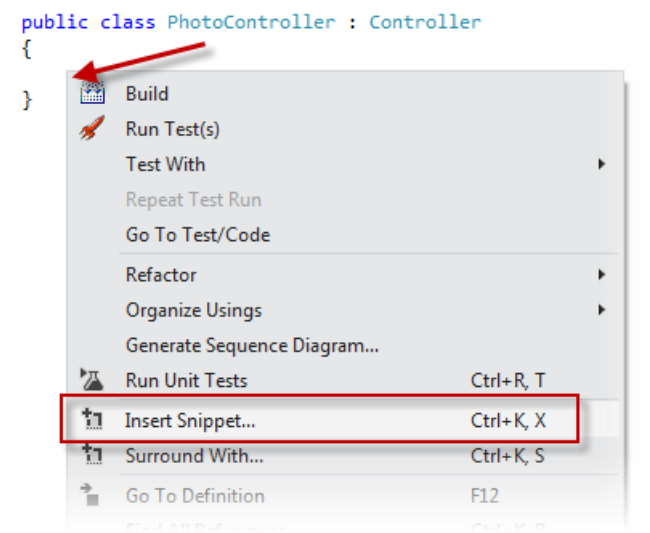
Press Tab to select the highlighted snippet

```
static void Main(string[] args)
{
    request.Headers.Add("$accountKey", accountKey);
    request.Headers.Add("$uniqueUserID", uniqueUserId);
}
```

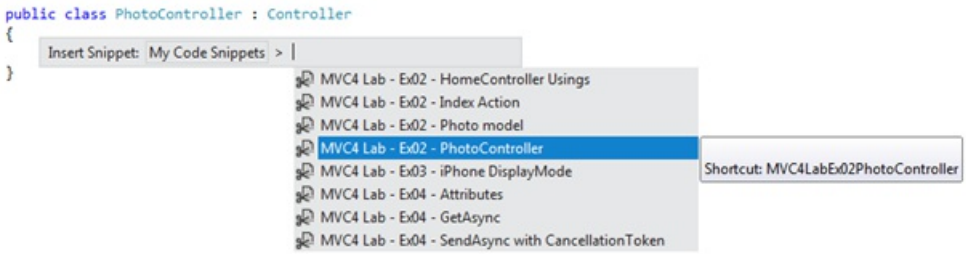
Press Tab again and the snippet will expand

To add a code snippet using the mouse (C#, Visual Basic and XML)

1. Right-click where you want to insert the code snippet.
2. Select **Insert Snippet** followed by **My Code Snippets**.
3. Pick the relevant snippet from the list, by clicking on it.



Right-click where you want to insert the code snippet and select Insert Snippet



Pick the relevant snippet from the list, by clicking on it

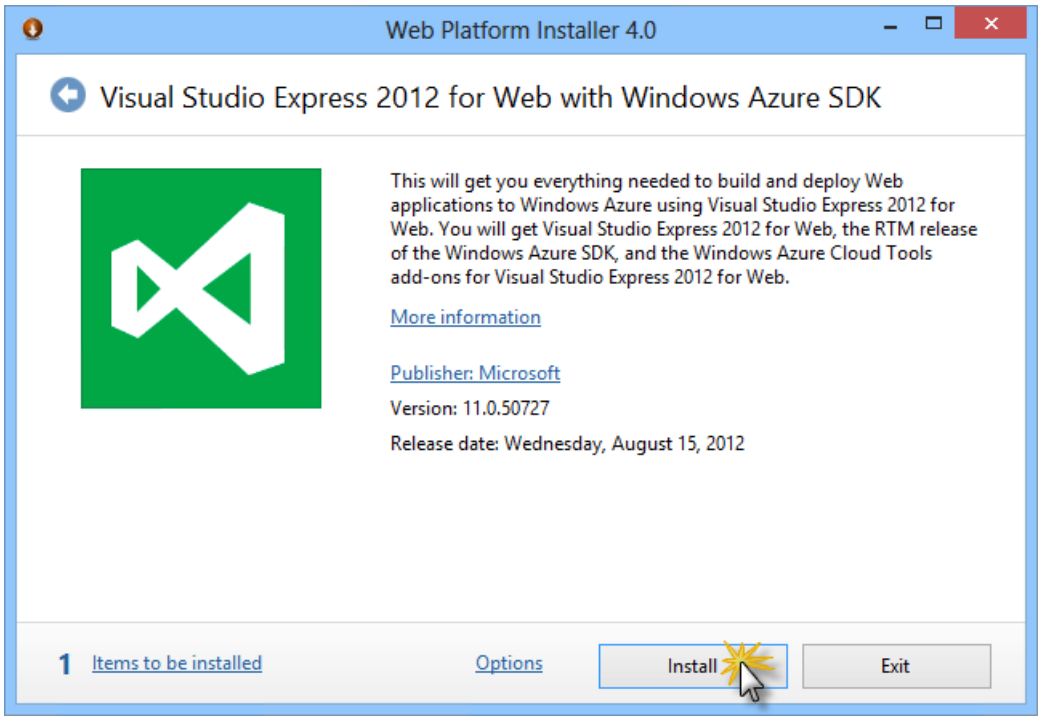
Appendix B: Installing Visual Studio Express 2012 for Web

You can install **Microsoft Visual Studio Express 2012 for Web** or another "Express" version using the **Microsoft Web Platform Installer**. The following instructions guide you through the steps required to install *Visual studio Express 2012 for Web* using *Microsoft Web Platform Installer*.

1. Go to https://go.microsoft.com/?linkid=9810169. Alternatively, if you already have installed Web Platform Installer, you can open it and search for the product "*Visual Studio Express 2012 for Web with Windows Azure SDK*".

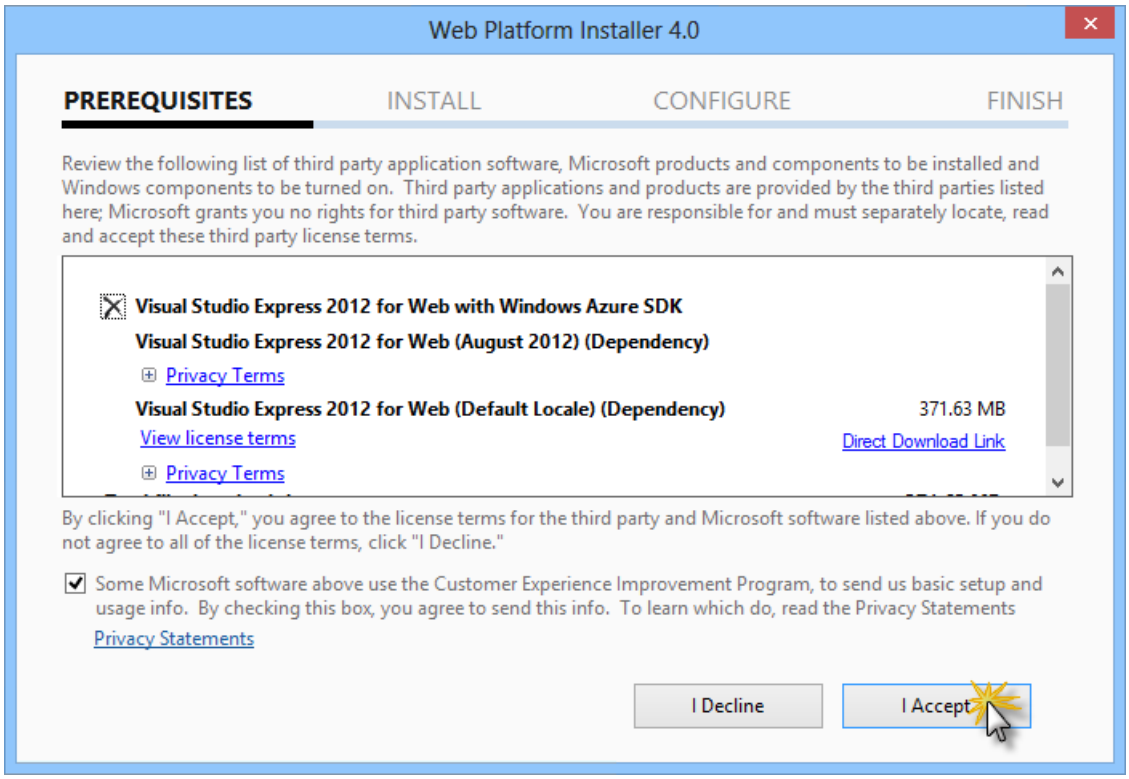
2. Click on **Install Now**. If you do not have **Web Platform Installer** you will be redirected to download and install it first.

3. Once **Web Platform Installer** is open, click **Install** to start the setup.



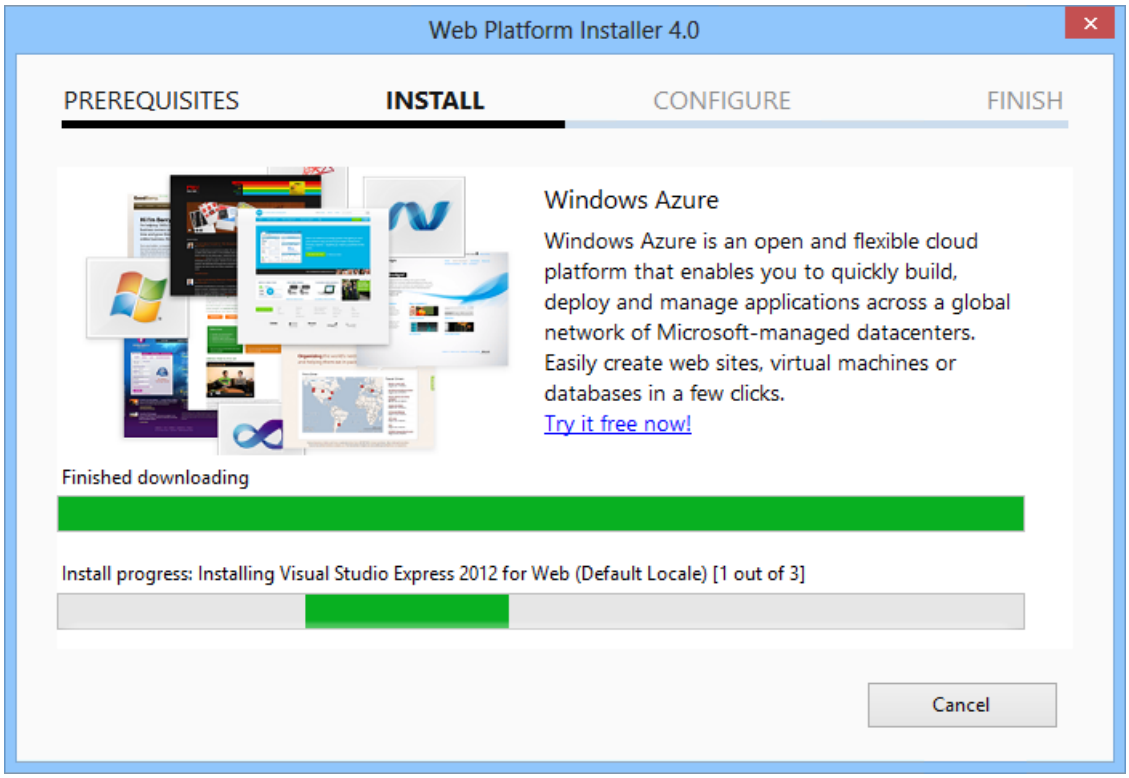
Install Visual Studio Express

4. Read all the products' licenses and terms and click **I Accept** to continue.



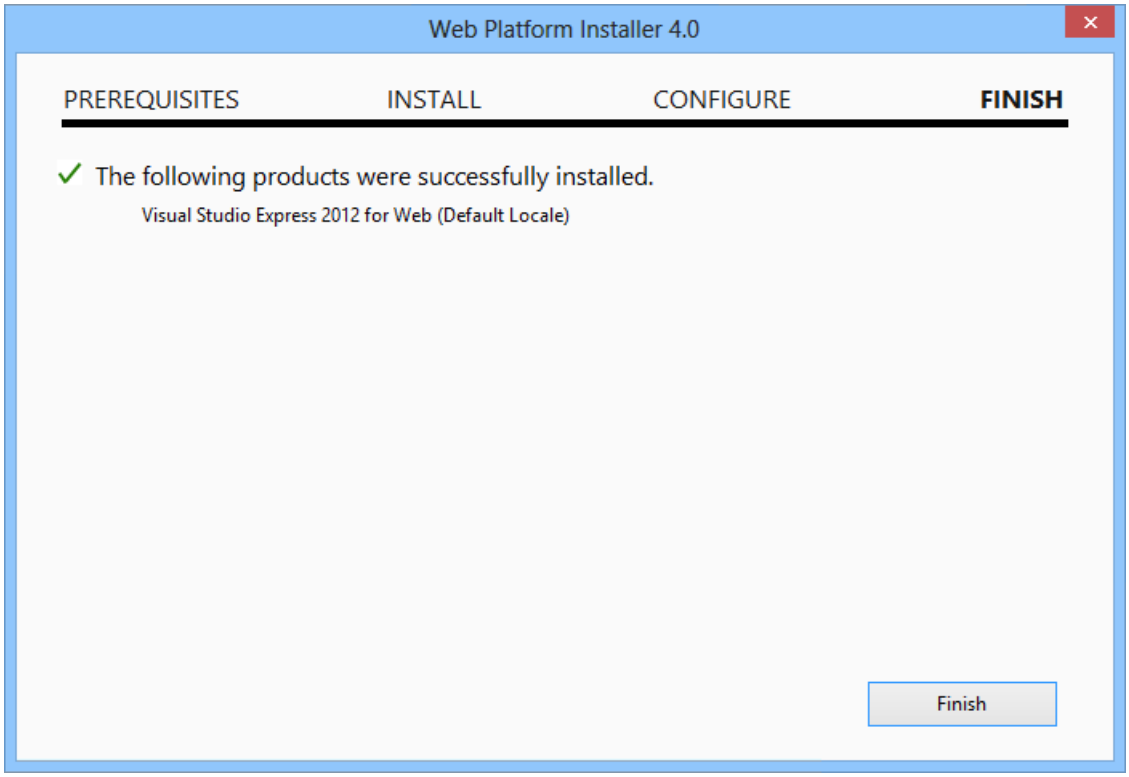
Accepting the license terms

5. Wait until the downloading and installation process completes.



Installation progress

6. When the installation completes, click **Finish**.



Installation completed

7. Click **Exit** to close Web Platform Installer.

8. To open Visual Studio Express for Web, go to the **Start** screen and start writing "**VS Express**", then click on the **VS Express for Web** tile.

VS Express for Web tile

Appendix C: Installing WebMatrix 2 and iPhone Simulator

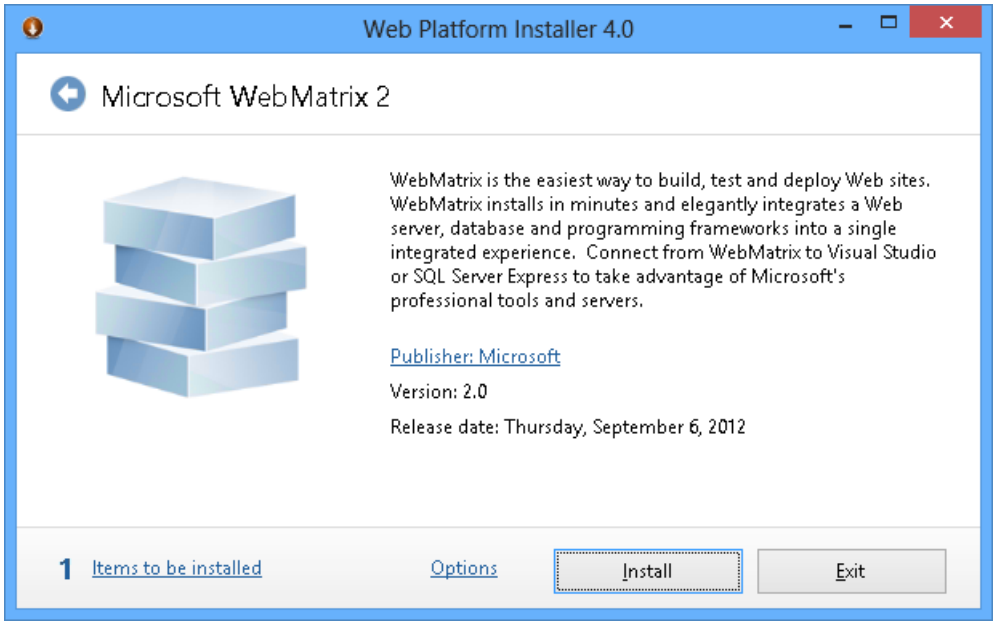
To run your site in a simulated iPhone device you can use the WebMatrix extension "Electric Mobile Simulator for the iPhone". Also, you can configure the same extension to run the simulator from Visual Studio 2012.

Task 1 - Installing WebMatrix 2

1. Go to [https://go.microsoft.com/?linkid=9809776](https://go.microsoft.com/?linkid=9810169). Alternatively, if you already have installed Web Platform Installer, you can open it and search for the product "*WebMatrix 2*".

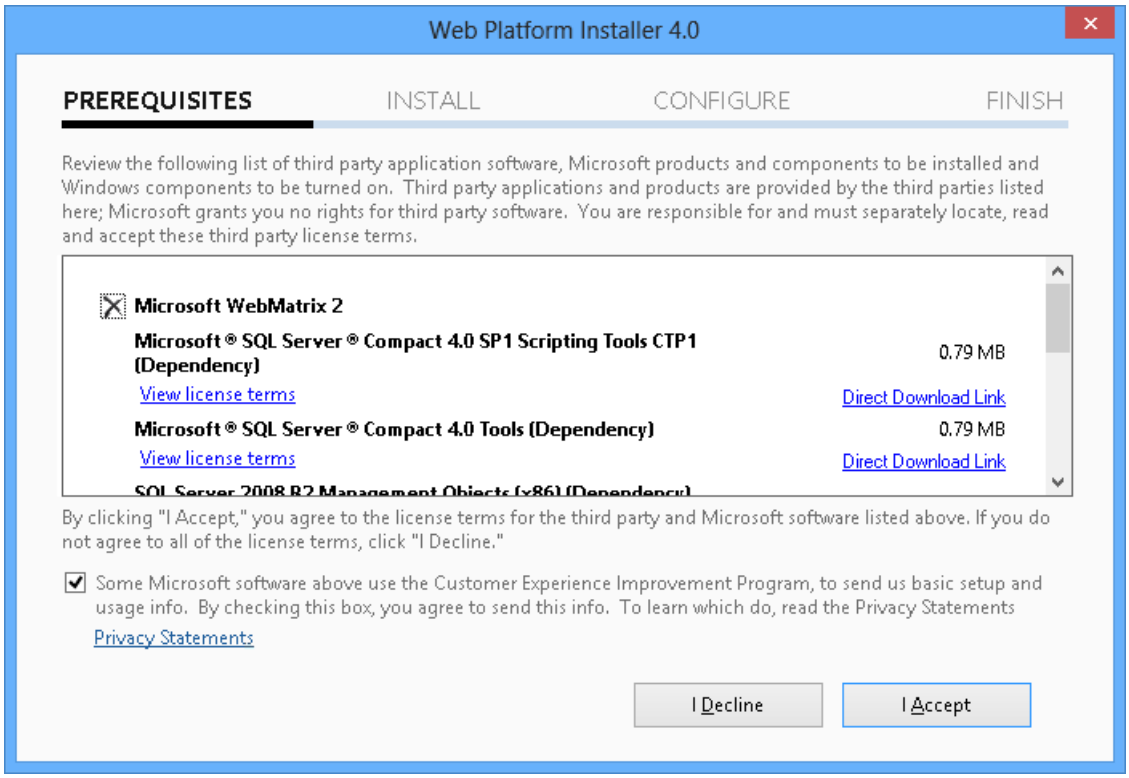
2. Click on **Install Now**. If you do not have **Web Platform Installer** you will be redirected to download and install it first.

3. Once **Web Platform Installer** is open, click **Install** to start the setup.



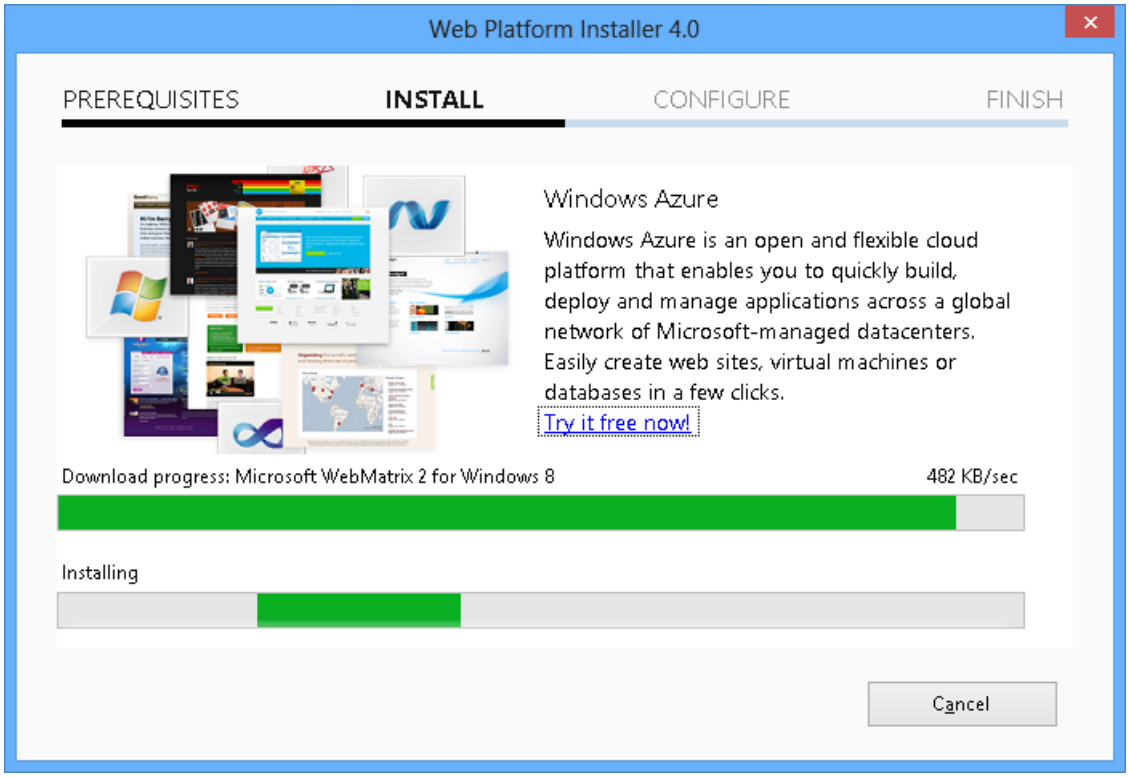
Install WebMatrix 2

4. Read all the products' licenses and terms and click **I Accept** to continue.



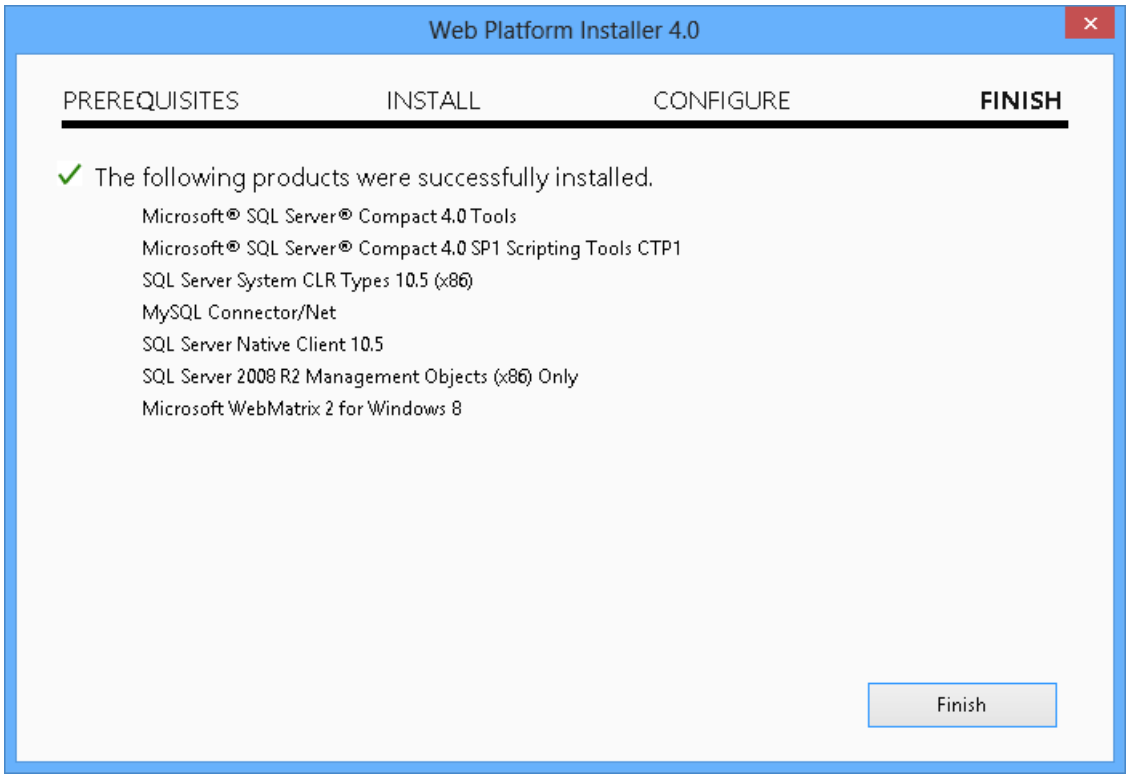
Accepting the license terms

5. Wait until the downloading and installation process completes.



Installation progress

6. When the installation completes, click **Finish**.



Installation completed

7. Click **Exit** to close Web Platform Installer.

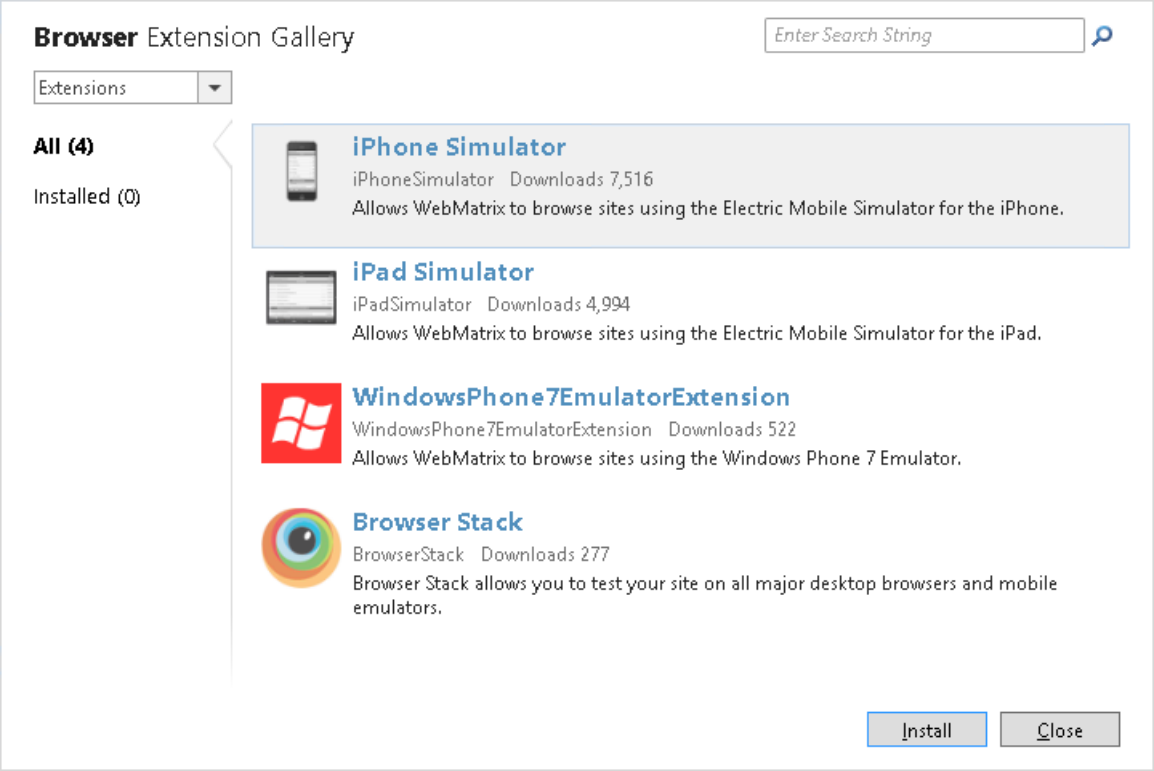
Task 2 - Installing the iPhone Simulator Extension

1. Run **WebMatrix** and open any existing Web site or create a new one.

2. Click the **Run** button from the **Home** ribbon and select **Add new**.

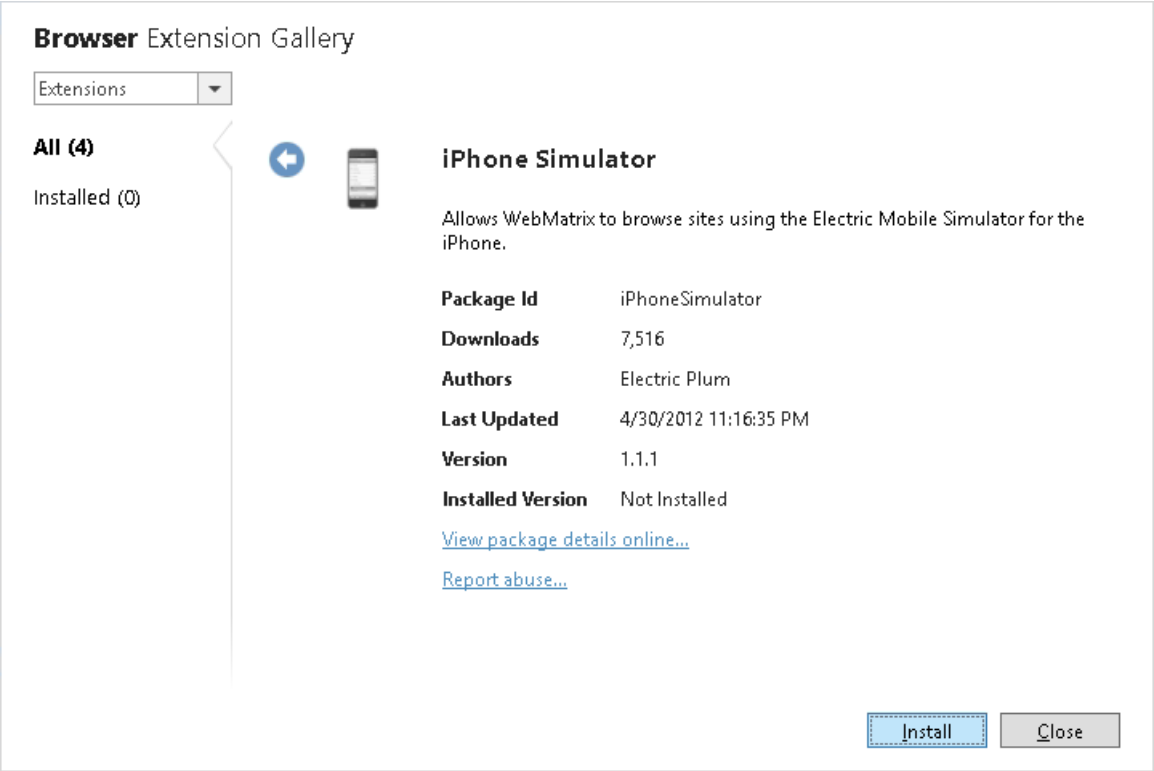
Adding new WebMatrix extension

3. Select **iPhone Simulator** and click **Install**.



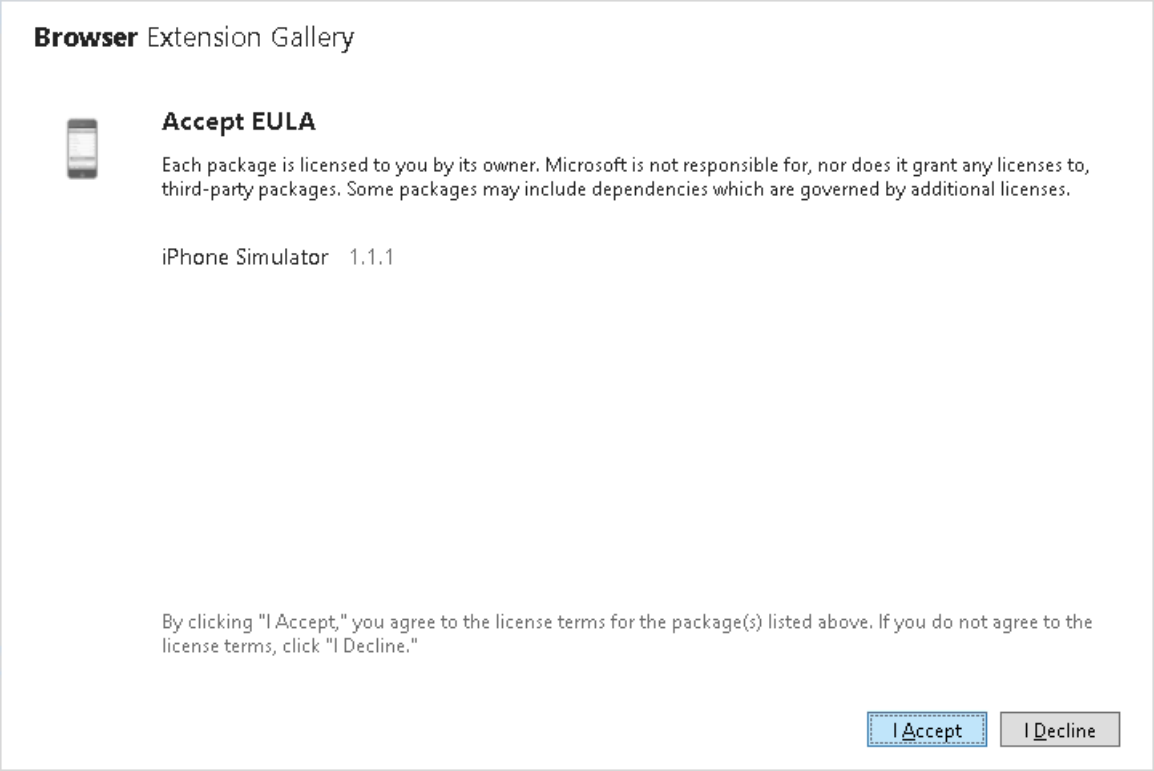
Browsing WebMatrix extensions

4. In the package details, click **Install** to continue with the extension installation.



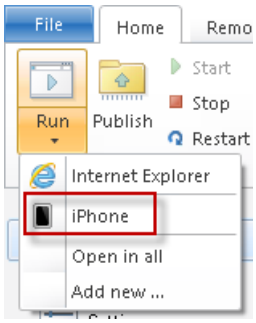
iPhone Simulator extension

5. Read and accept the extension EULA.



WebMatrix extension EULA

6. Now, you can run your Web site from WebMatrix using the iPhone Simulator option.

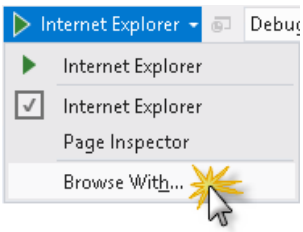


Run using iPhone

Task 3 - Configuring Visual Studio 2012 to run iPhone Simulator

1. Open **Visual Studio 2012** and open any Web site or create a new project.

2. Click the down arrow from the Run button and select **Browse with**.



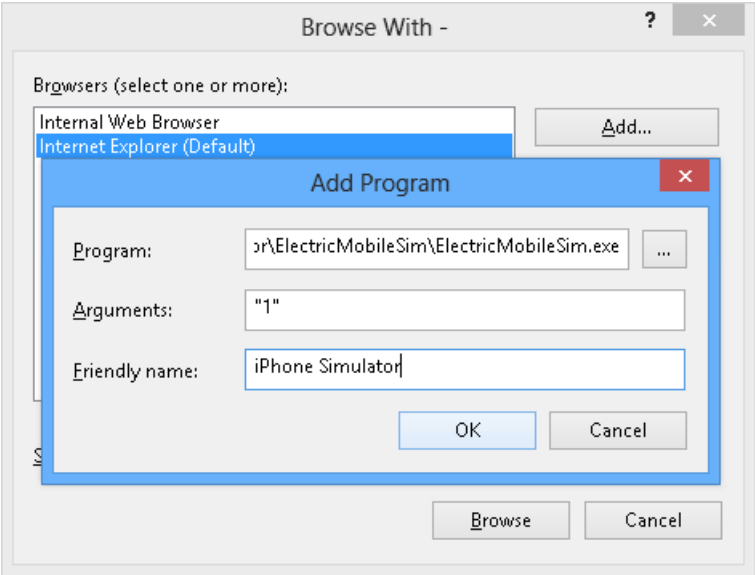
Browse with

3. In the "Browse With" dialog, click **Add**.

4. In the "Add Program" dialog, use the following values:

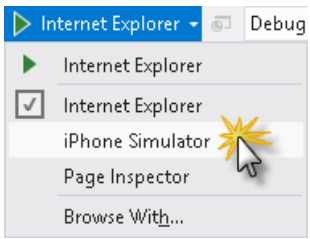
   - **Program**: C:\Users*{CurrentUser}*\AppData\Local\Microsoft\WebMatrix\Extensions\20\iPhoneSimulator\ElectricMobileSim\ElectricMobileSim.exe *(update the path accordingly)*

- **Arguments**: "1"
- **Friendly name**: iPhone Simulator



Add program to browse with

5. Click **OK** and close the dialogs.
6. Now you are able to run your Web applications in the iPhone simulator from Visual Studio 2012.



Browse with iPhone Simulator

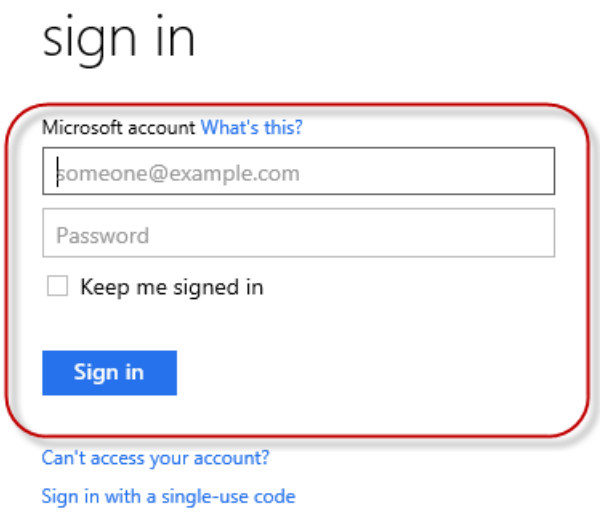
Appendix D: Publishing an ASP.NET MVC 4 Application using Web Deploy

This appendix will show you how to create a new web site from the Windows Azure Management Portal and publish the application you obtained by following the lab, taking advantage of the Web Deploy publishing feature provided by Windows Azure.

Task 1 - Creating a New Web Site from the Windows Azure Portal

1. Go to the Windows Azure Management Portal and sign in using the Microsoft credentials associated with your subscription.

> **NOTE**
> With Windows Azure you can host 10 ASP.NET Web Sites for free and then scale as your traffic grows. You can sign up here.



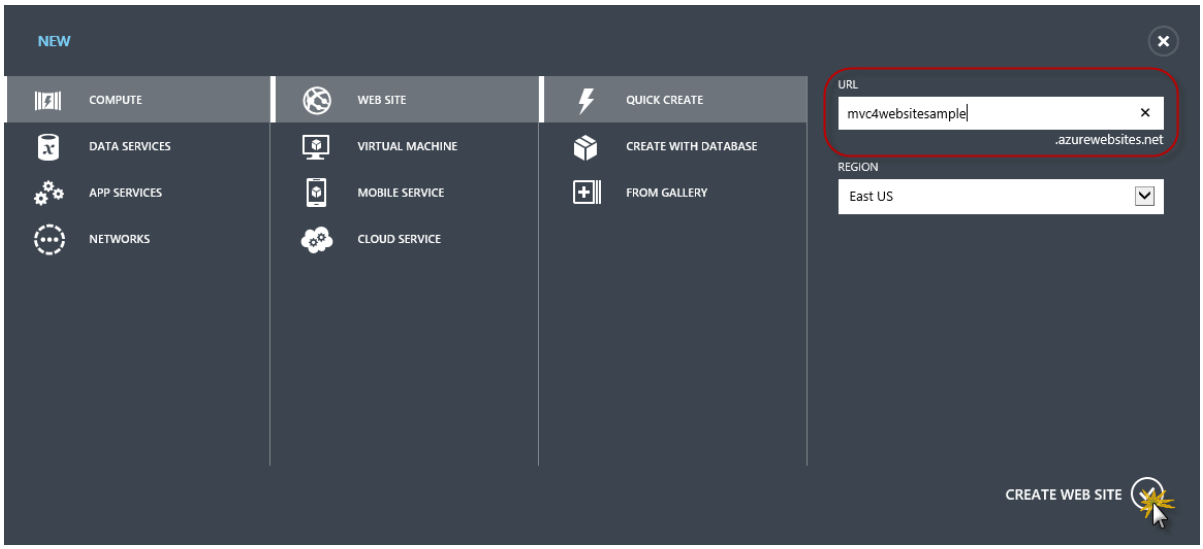
Log on to Windows Azure Management Portal

2. Click **New** on the command bar.

Creating a new Web Site

3. Click **Compute | Web Site**. Then select **Quick Create** option. Provide an available URL for the new web site and click **Create Web Site**.

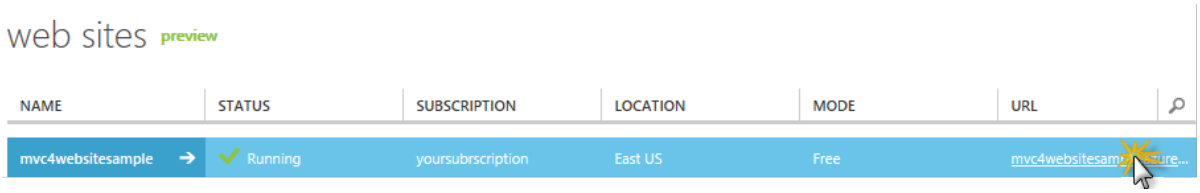
> **NOTE**
>
> A Windows Azure Web Site is the host for a web application running in the cloud that you can control and manage. The Quick Create option allows you to deploy a completed web application to the Windows Azure Web Site from outside the portal. It does not include steps for setting up a database.



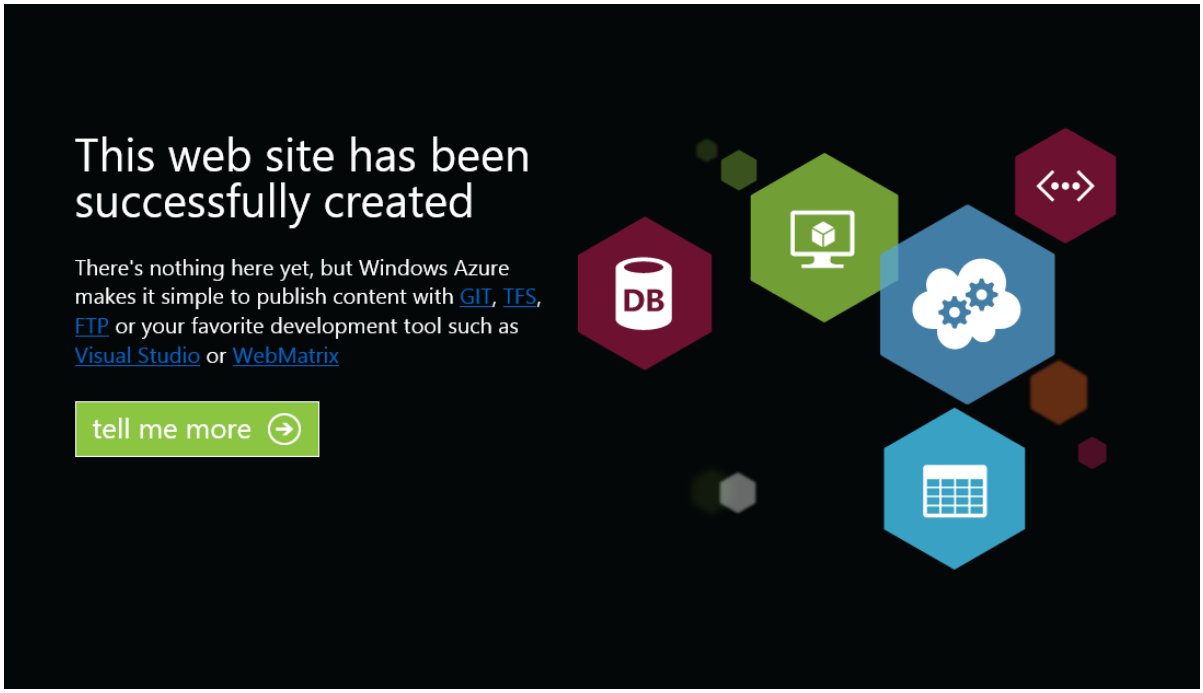
Creating a new Web Site using Quick Create

4. Wait until the new **Web Site** is created.

5. Once the Web Site is created click the link under the **URL** column. Check that the new Web Site is working.

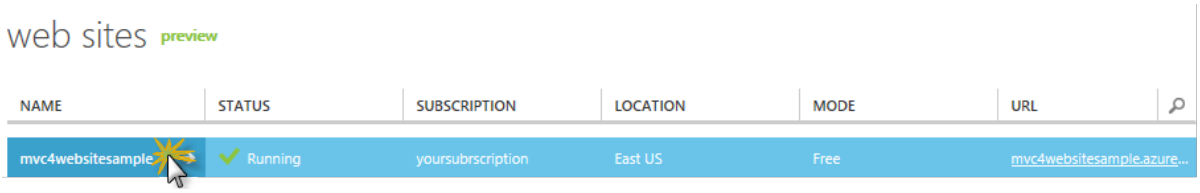


Browsing to the new web site



Web site running

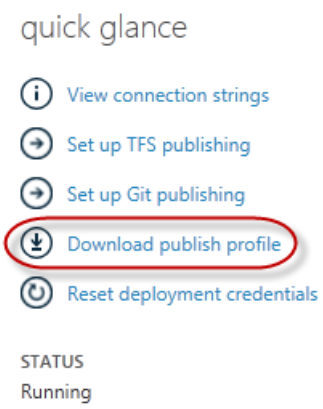
6. Go back to the portal and click the name of the web site under the **Name** column to display the management pages.



Opening the Web Site management pages

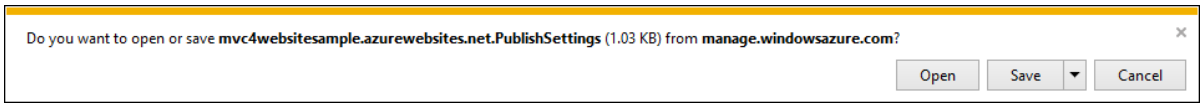
7. In the **Dashboard** page, under the **quick glance** section, click the **Download publish profile** link.

> **NOTE**
>
> The *publish profile* contains all of the information required to publish a web application to a Windows Azure website for each enabled publication method. The publish profile contains the URLs, user credentials and database strings required to connect to and authenticate against each of the endpoints for which a publication method is enabled. **Microsoft WebMatrix 2**, **Microsoft Visual Studio Express for Web** and **Microsoft Visual Studio 2012** support reading publish profiles to automate configuration of these programs for publishing web applications to Windows Azure websites.



Downloading the Web Site publish profile

8. Download the publish profile file to a known location. Further in this exercise you will see how to use this file to publish a web application to a Windows Azure Web Sites from Visual Studio.

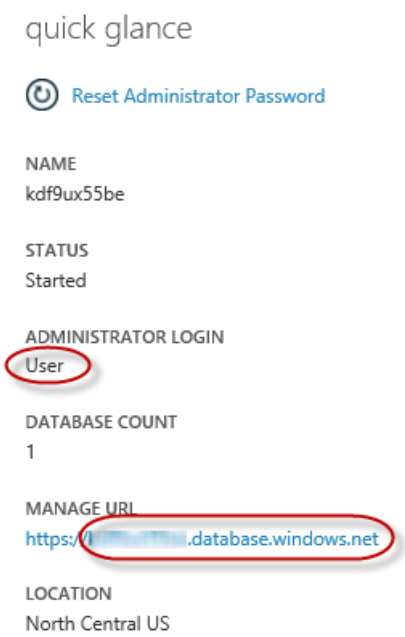


Saving the publish profile file

Task 2 - Configuring the Database Server

If your application makes use of SQL Server databases you will need to create a SQL Database server. If you want to deploy a simple application that does not use SQL Server you might skip this task.

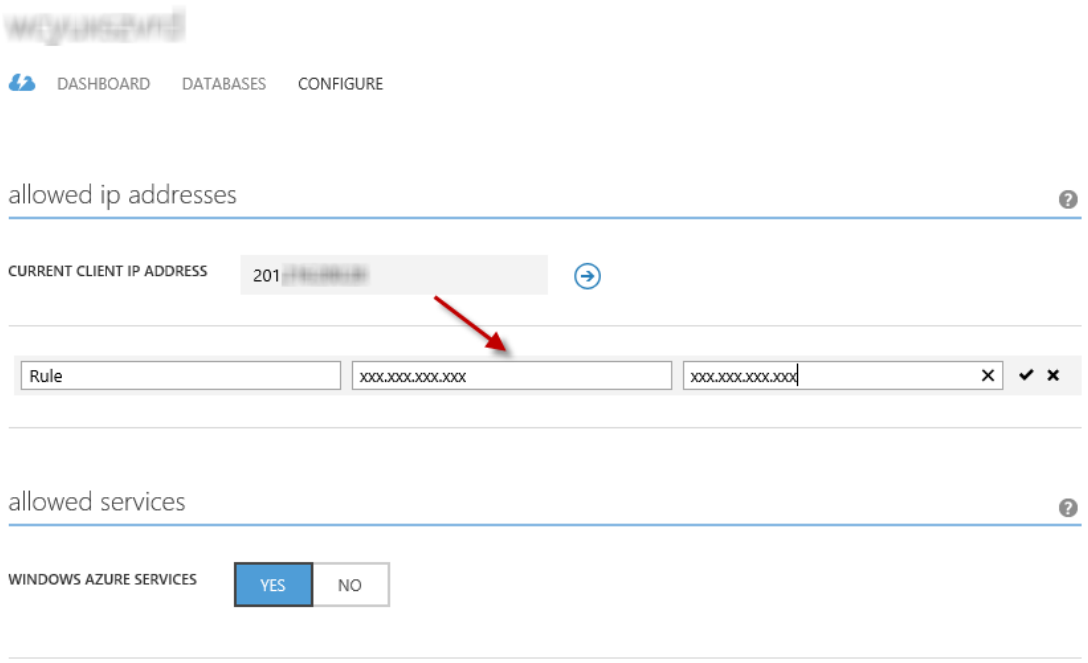
1. You will need a SQL Database server for storing the application database. You can view the SQL Database servers from your subscription in the Windows Azure Management portal at **Sql Databases | Servers | Server's Dashboard**. If you do not have a server created, you can create one using the **Add** button on the command bar. Take note of the **server name and URL, administrator login name and password**, as you will use them in the next tasks. Do not create the database yet, as it will be created in a later stage.



SQL Database Server Dashboard

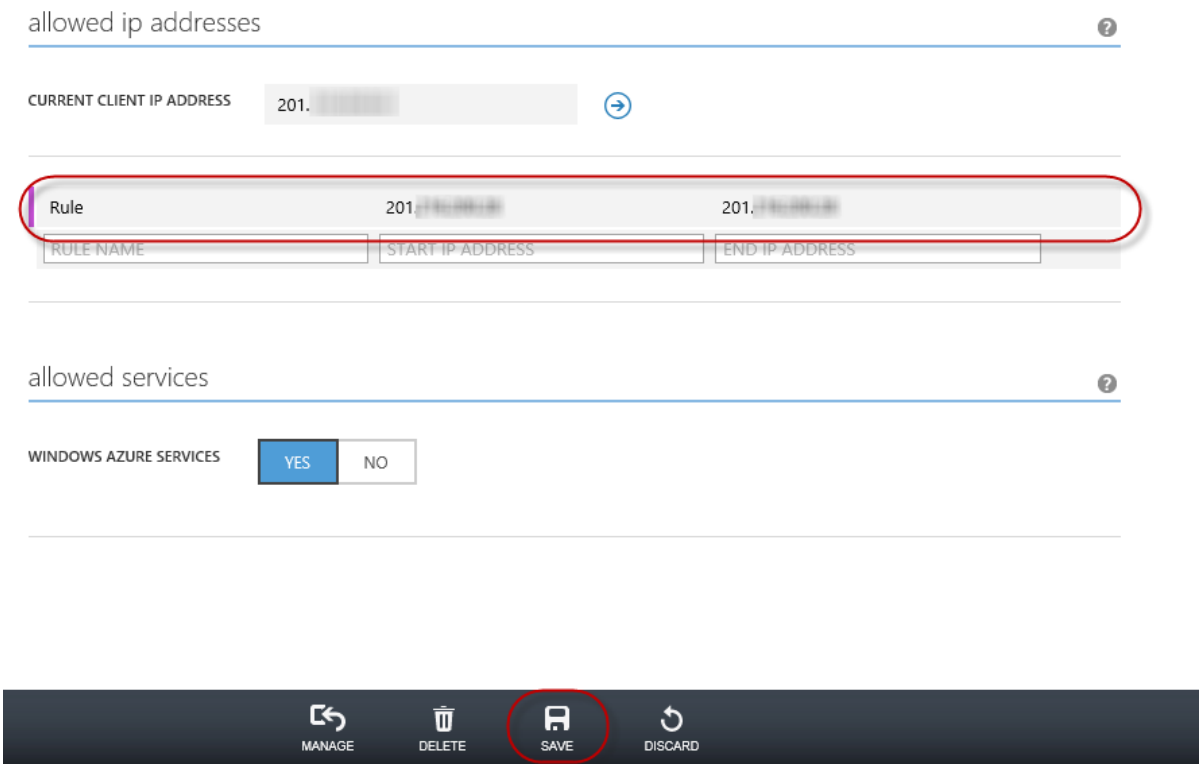
2. In the next task you will test the database connection from Visual Studio, for that reason you need to include your local IP address in the server's list of **Allowed IP Addresses**. To do that, click **Configure**, select the IP address from **Current Client IP Address** and paste it on the **Start IP Address** and **End IP Address** text

boxes and click the ✔ button.



Adding Client IP Address

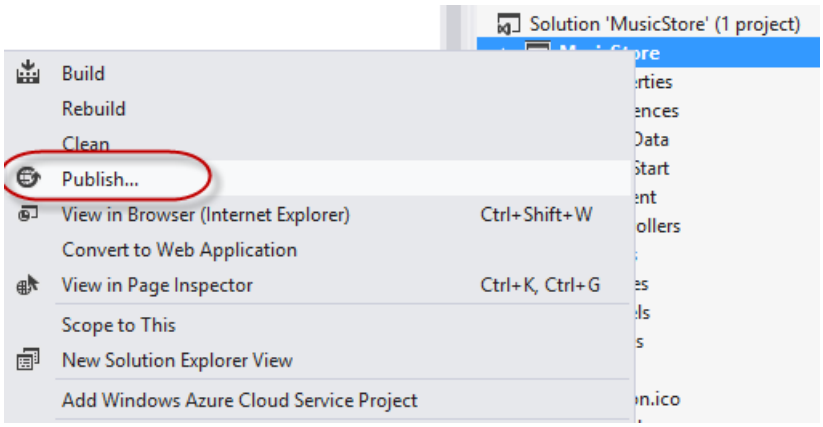
3. Once the **Client IP Address** is added to the allowed IP addresses list, click on **Save** to confirm the changes.



Confirm Changes

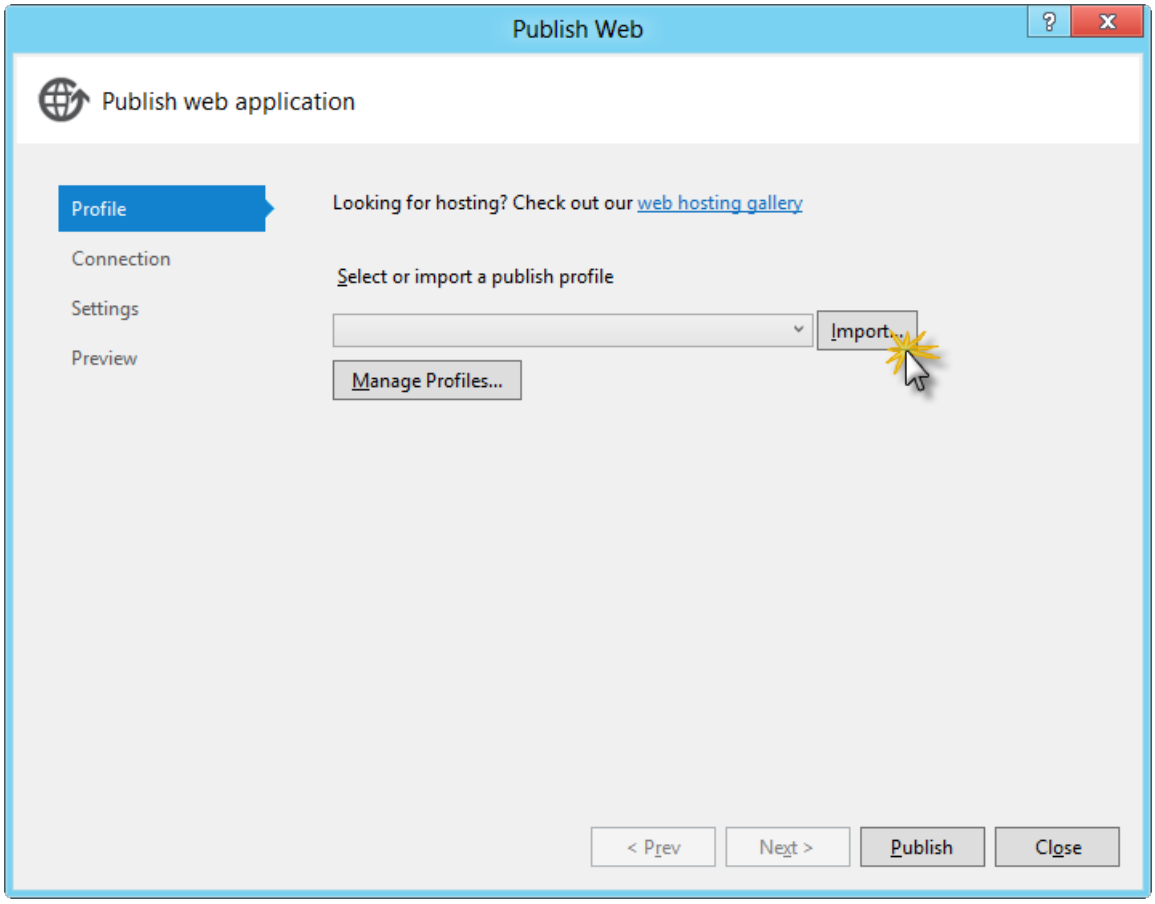
Task 3 - Publishing an ASP.NET MVC 4 Application using Web Deploy

1. Go back to the ASP.NET MVC 4 solution. In the **Solution Explorer**, right-click the web site project and select **Publish**.



Publishing the web site

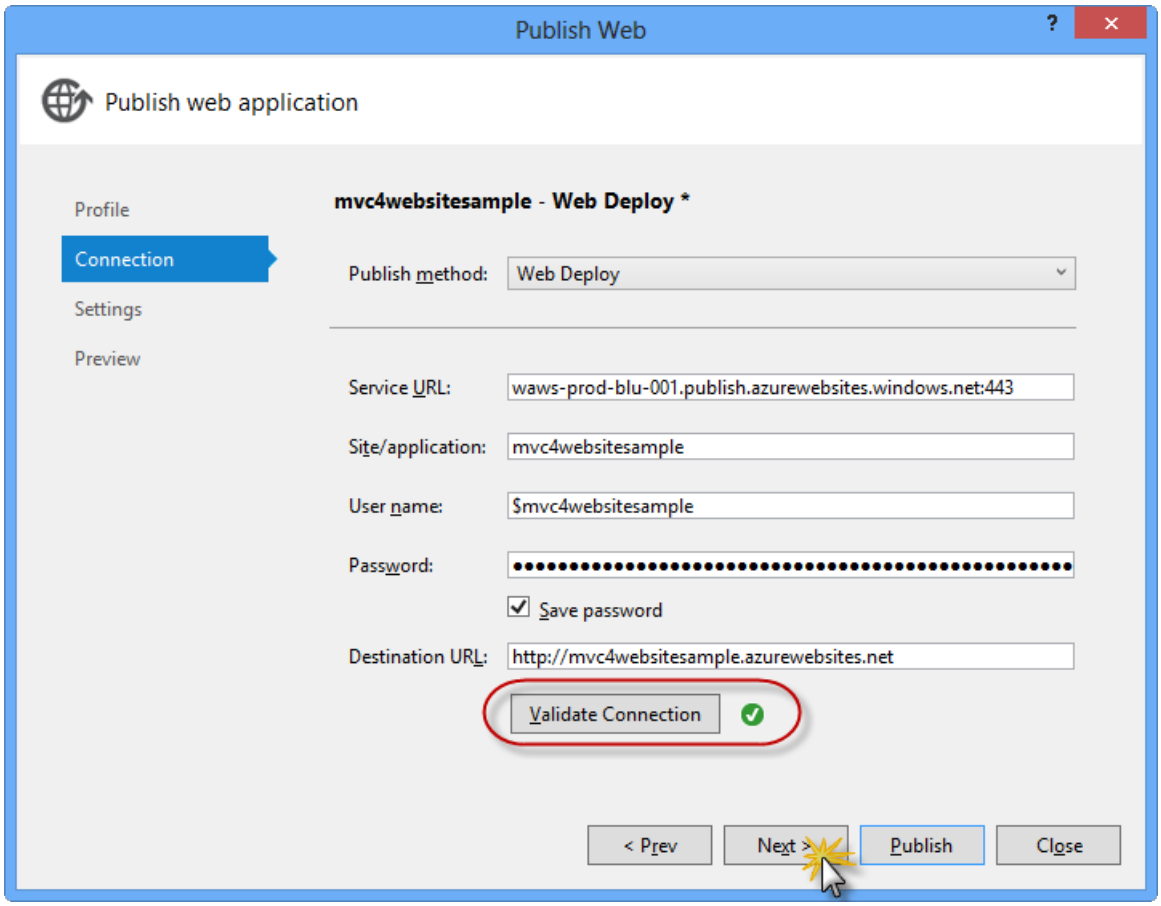
2. Import the publish profile you saved in the first task.



Importing publish profile

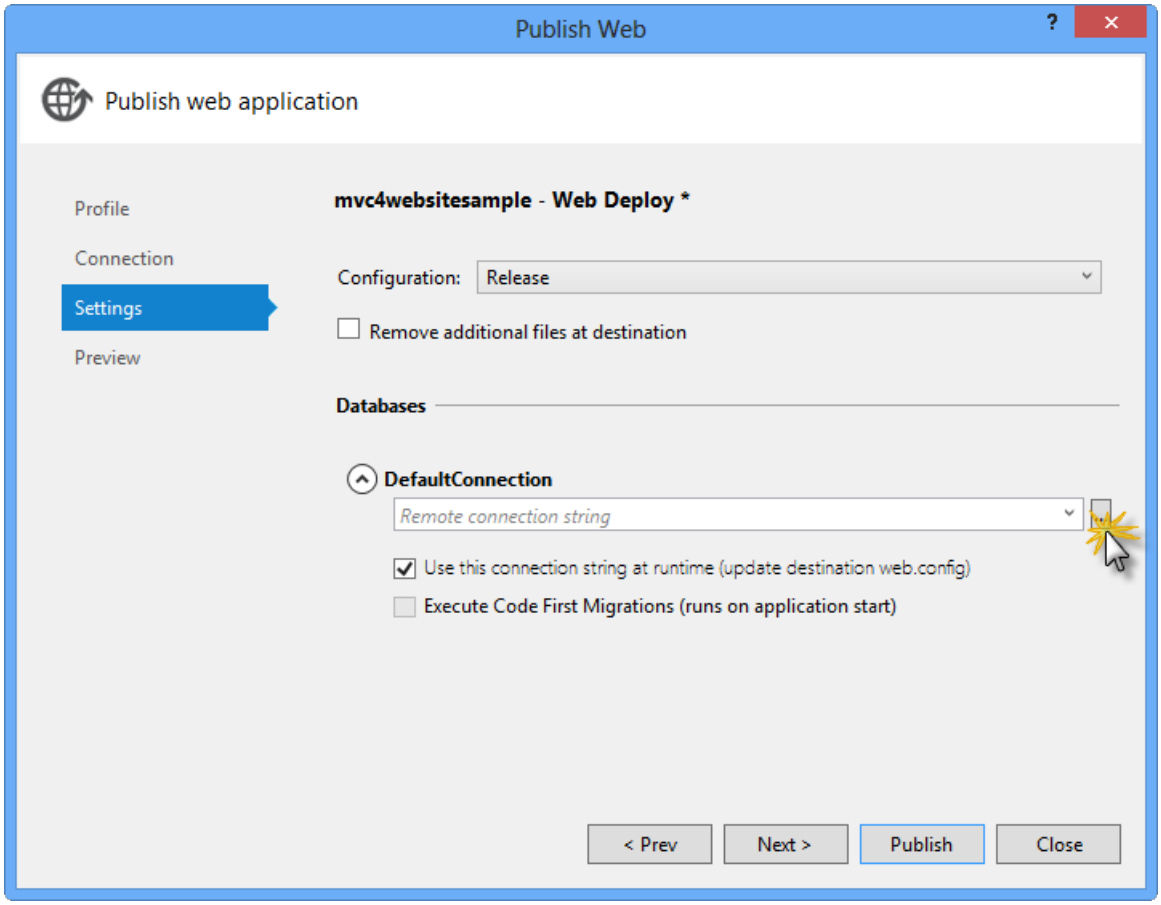
3. Click **Validate Connection**. Once Validation is complete click **Next**.

> **NOTE**
>
> Validation is complete once you see a green checkmark appear next to the Validate Connection button.



Validating connection

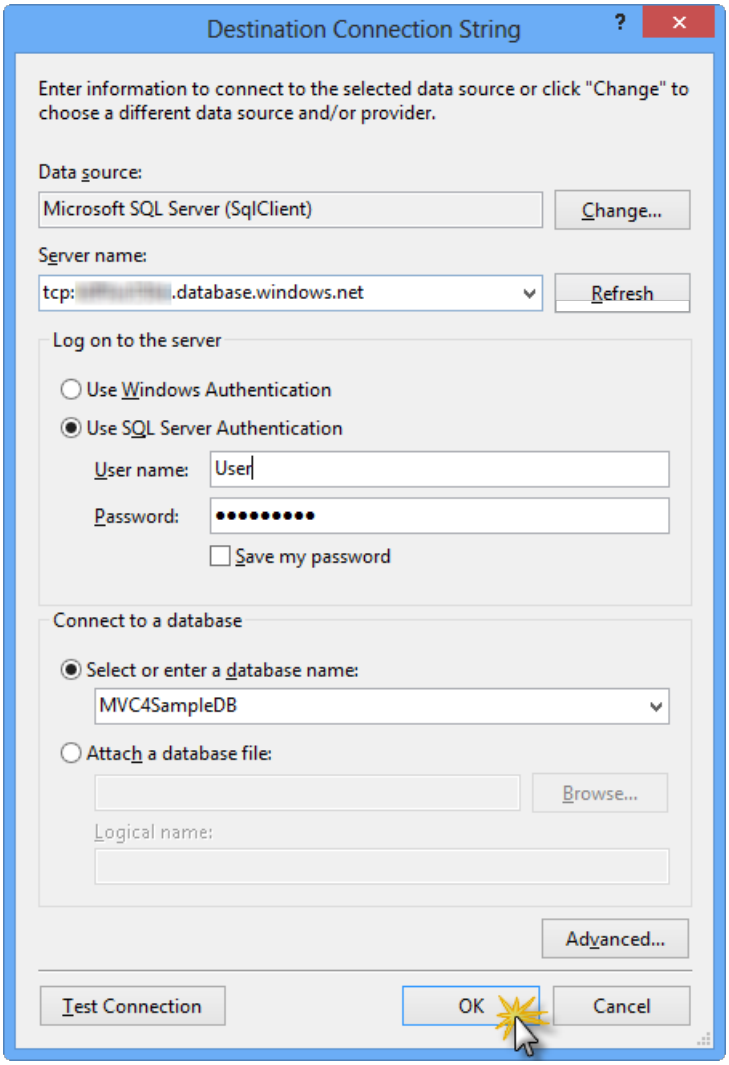
4. In the **Settings** page, under the **Databases** section, click the button next to your database connection's textbox (i.e. **DefaultConnection**).



Web deploy configuration

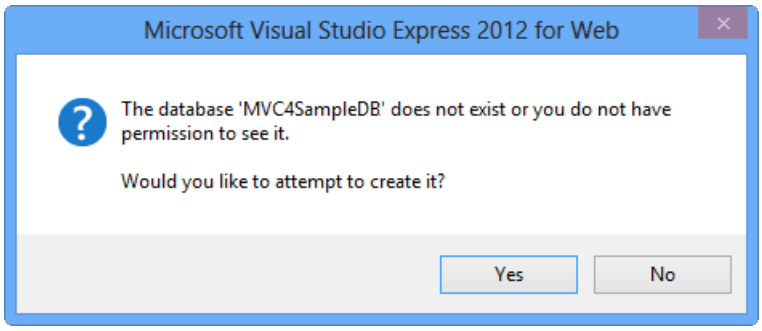
5. Configure the database connection as follows:

   - In the **Server name** type your SQL Database server URL using the *tcp:* prefix.
   - In **User name** type your server administrator login name.
   - In **Password** type your server administrator login password.
   - Type a new database name, for example: *MVC4SampleDB*.



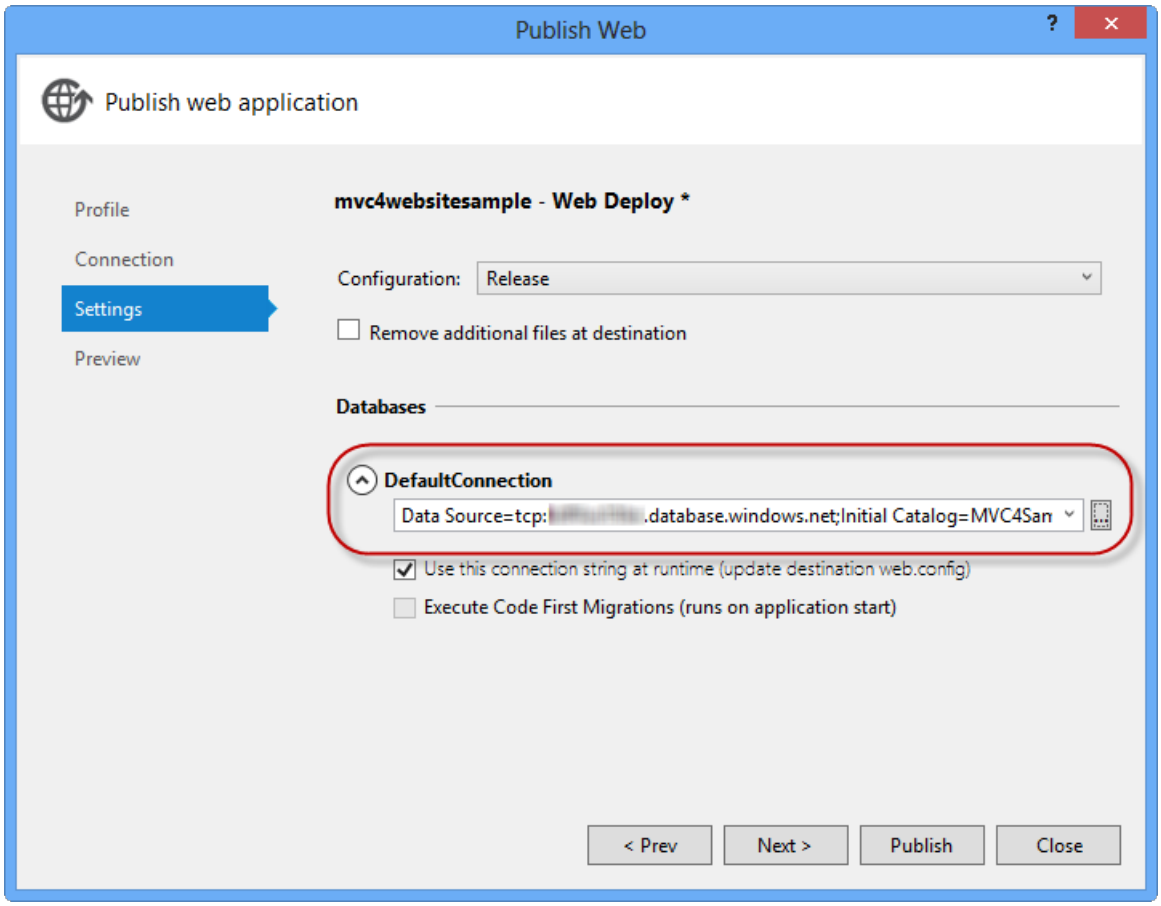
Configuring destination connection string

6. Then click **OK**. When prompted to create the database click **Yes**.



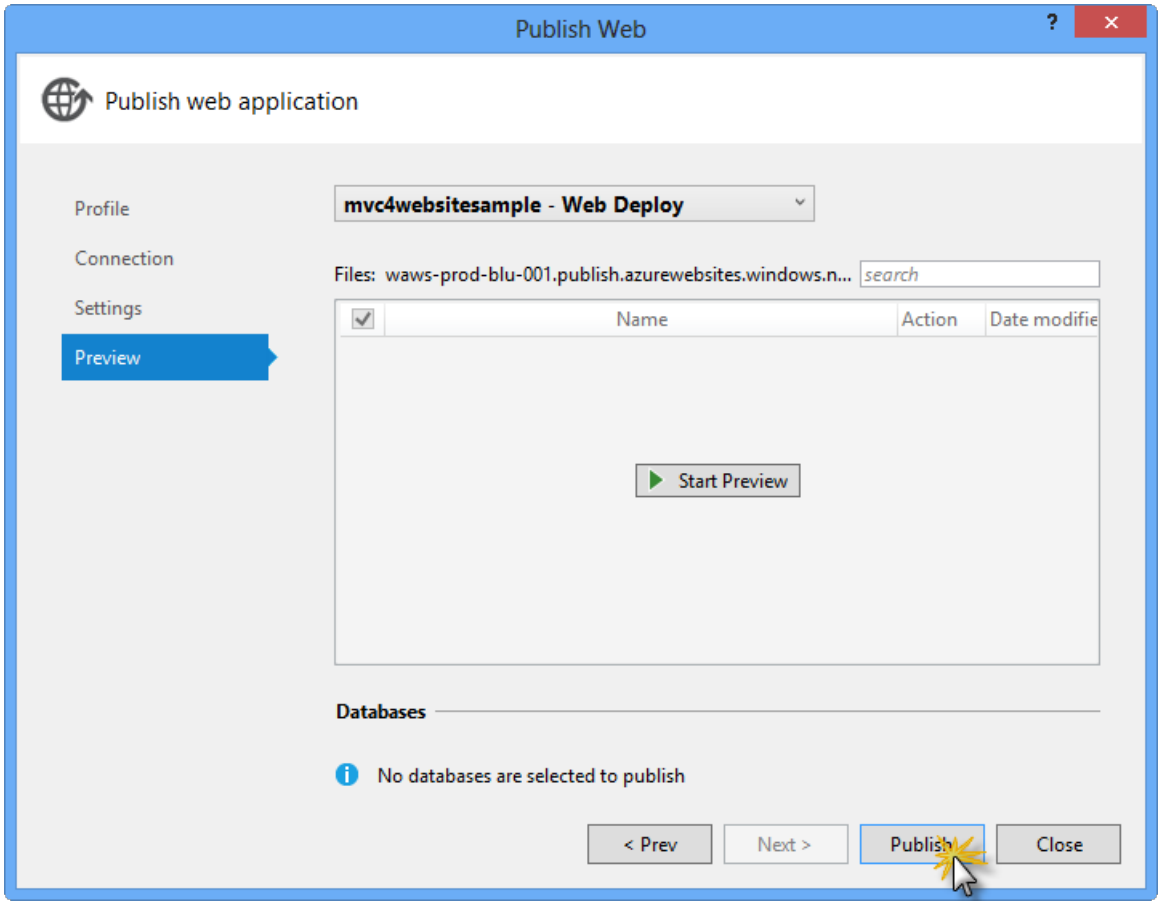
Creating the database

7. The connection string you will use to connect to SQL Database in Windows Azure is shown within Default Connection textbox. Then click **Next**.



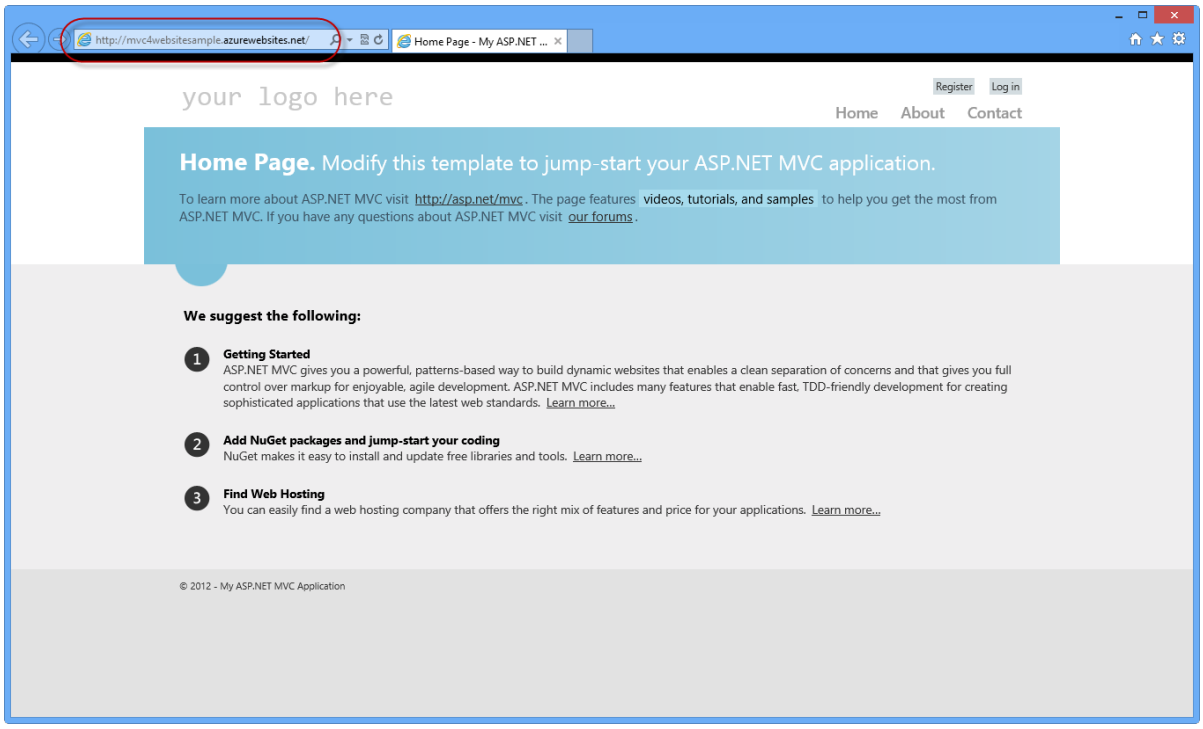
Connection string pointing to SQL Database

8. In the **Preview** page, click **Publish**.



Publishing the web application

9. Once the publishing process finishes, your default browser will open the published web site.



Application published to Windows Azure

ASP.NET MVC 4 Fundamentals

2/20/2020 • 48 minutes to read • Edit Online

By Web Camps Team

Download Web Camps Training Kit

This Hands-On Lab is based on MVC (Model View Controller) Music Store, a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Studio. Throughout the lab you will learn the simplicity, yet power of using these technologies together. You will start with a simple application and will build it until you have a fully functional ASP.NET MVC 4 Web Application.

This Lab works with ASP.NET MVC 4.

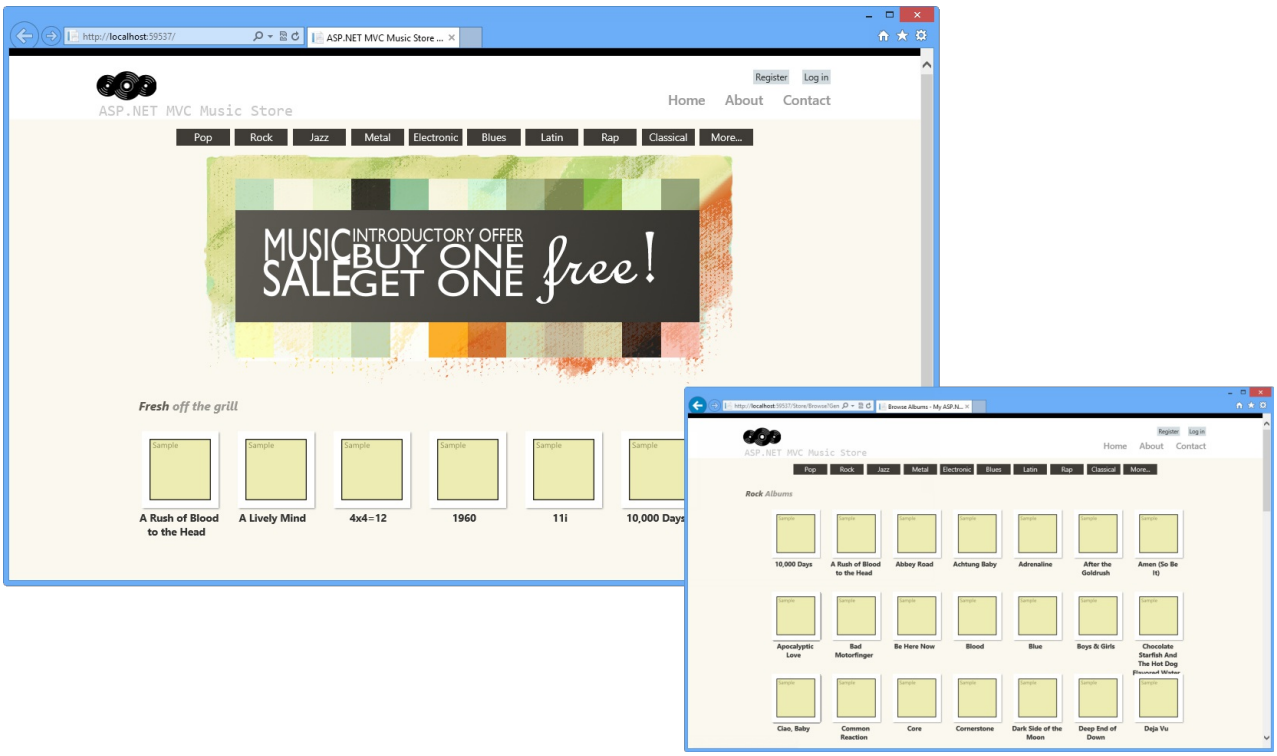
If you wish to explore the ASP.NET MVC 3 version of the tutorial application, you can find it in MVC-Music-Store.

This Hands-On Lab assumes that the developer has experience in Web development technologies, such as HTML and JavaScript.

> **NOTE**
>
> All sample code and snippets are included in the Web Camps Training Kit, available at Microsoft-Web/WebCampTrainingKit Releases. The project specific to this lab is available at ASP.NET MVC 4 Fundamentals.

The Music Store application

The Music Store web application that will be built throughout this Lab comprises three main parts: shopping, checkout, and administration. Visitors will be able to browse albums by genre, add albums to their cart, review their selection and finally proceed to checkout to login and complete the order. Additionally, store administrators will be able to manage the available albums as well as their main properties.



Music Store screens

ASP.NET MVC 4 Essentials

Music Store application will be built using **Model View Controller (MVC)**, an architectural pattern that separates an application into three main components:

- **Models**: Model objects are the parts of the application that implement the domain logic. Often, model objects also retrieve and store model state in a database.
- **Views:** Views are the components that display the application's user interface (UI). Typically, this UI is created from the model data. An example would be the edit view of Albums that displays text boxes and a drop-down list based on the current state of an Album object.
- **Controllers:** Controllers are the components that handle user interaction, manipulate the model, and ultimately select a view to render the UI. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction.

The MVC pattern helps you to create applications that separate the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements. This separation helps you manage complexity when you build an application, as it allows you to focus on one aspect of the implementation at a time. In addition, the MVC pattern makes it easy to test applications, also encouraging the use of test-driven development (TDD) for creating applications.

The **ASP.NET MVC** framework provides an alternative to the ASP.NET Web Forms pattern for creating ASP.NET MVC-based Web applications. The **ASP.NET MVC** framework is a lightweight, highly testable presentation framework that (as with Web-forms-based applications) is integrated with existing ASP.NET features, such as master pages and membership-based authentication so you get all the power of the core .NET framework. This is useful if you are already familiar with ASP.NET Web Forms because all the libraries that you already use are available in ASP.NET MVC 4 as well.

In addition, the loose coupling between the three main components of an MVC application also promotes parallel development. For instance, one developer can work on the view, a second developer can work on the controller logic, and a third developer can focus on the business logic in the model.

Objectives

In this Hands-On Lab, you will learn how to:

- Create an ASP.NET MVC application from scratch based on the Music Store Application tutorial
- Add Controllers to handle URLs to the Home page of the site and for browsing its main functionality
- Add a View to customize the content displayed along with its style
- Add Model classes to contain and manage data and domain logic
- Use View Model pattern to pass information from controller actions to the view templates
- Explore the ASP.NET MVC 4 new template for internet applications

Prerequisites

You must have the following items to complete this lab:

- Visual Studio 2012 Express for Web (read Appendix A for instructions on how to install it)

Setup

Installing Code Snippets

For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. To install the code snippets run **.\Source\Setup\CodeSnippets.vsi** file.

If you are not familiar with the Visual Studio Code Snippets, and want to learn how to use them, you can refer to the appendix from this document "Appendix C: Using Code Snippets".

Exercises

This Hands-On Lab is comprised by the following exercises:

1. Exercise 1: Creating MusicStore ASP.NET MVC Web Application Project
2. Exercise 2: Creating a Controller
3. Exercise 3: Passing parameters to a Controller
4. Exercise 4: Creating a View
5. Exercise 5: Creating a View Model
6. Exercise 6: Using parameters in View
7. Exercise 7: A lap around ASP.NET MVC 4 New Template

> **NOTE**
> Each exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

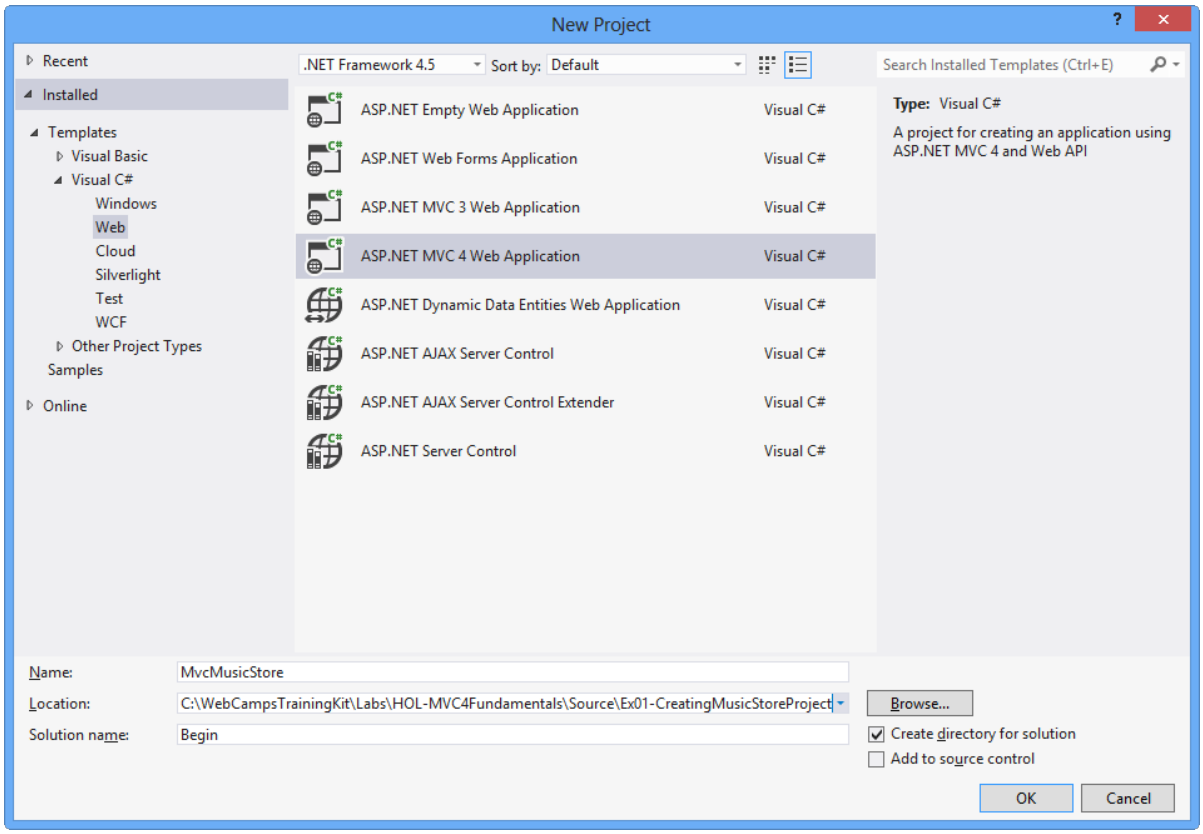
Estimated time to complete this lab: **60 minutes**.

Exercise 1: Creating MusicStore ASP.NET MVC Web Application Project

In this exercise, you will learn how to create an ASP.NET MVC application in Visual Studio 2012 Express for Web as well as its main folder organization. Additionally, you will learn how to add a new Controller and make it display a simple string in the application's home page.

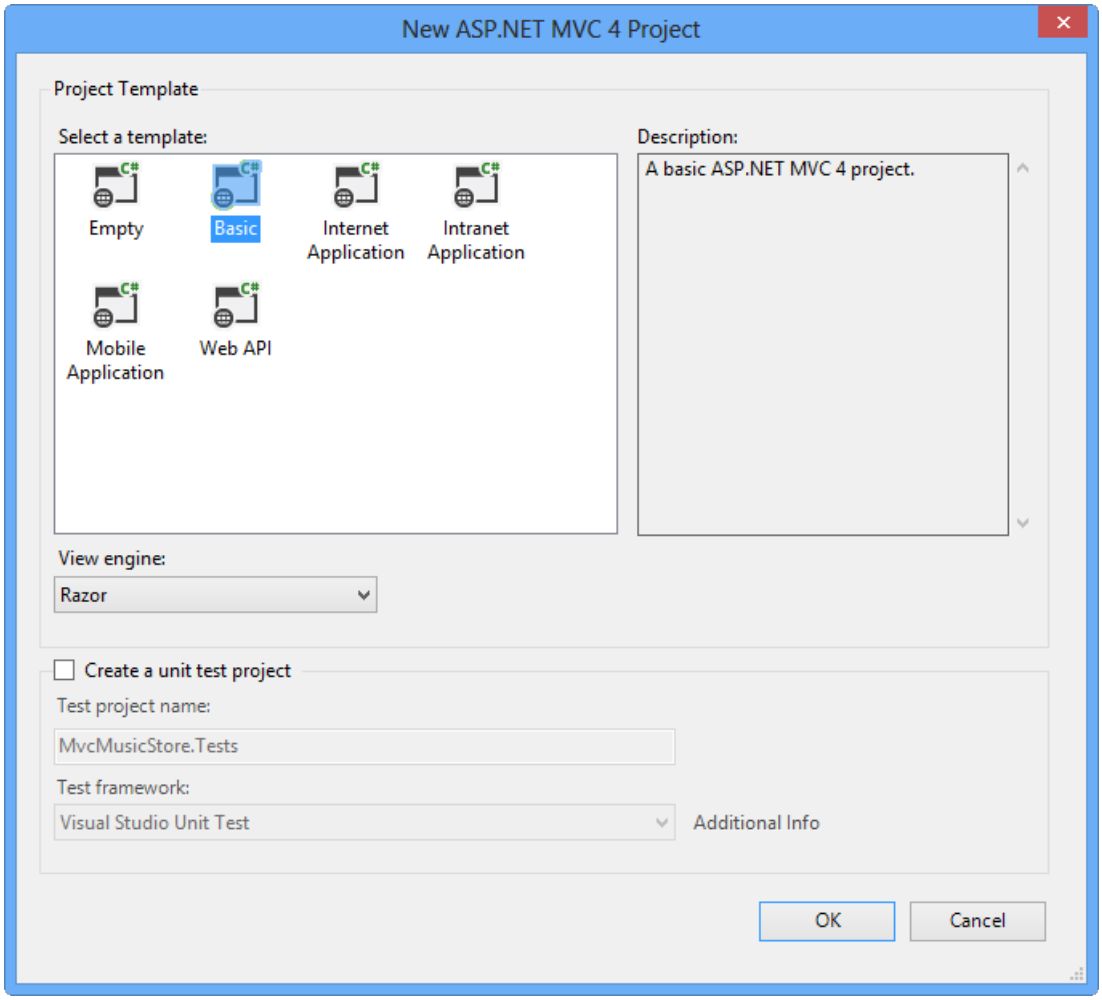
Task 1 - Creating the ASP.NET MVC Web Application Project

1. In this task, you will create an empty ASP.NET MVC application project using the MVC Visual Studio template. Start **VS Express for Web**.
2. On the **File** menu, click **New Project**.
3. In the **New Project** dialog box select the **ASP.NET MVC 4 Web Application** project type, located under **Visual C#, Web** template list.
4. Change the **Name** to *MvcMusicStore*.
5. Set the location of the solution inside a new **Begin** folder in this Exercise's Source folder, for example **[YOUR-HOL-PATH]\Source\Ex01-CreatingMusicStoreProject\Begin**. Click **OK**.



Create New Project Dialog Box

6. In the **New ASP.NET MVC 4 Project** dialog box select the **Basic** template and make sure that the **View engine** selected is **Razor**. Click **OK**.



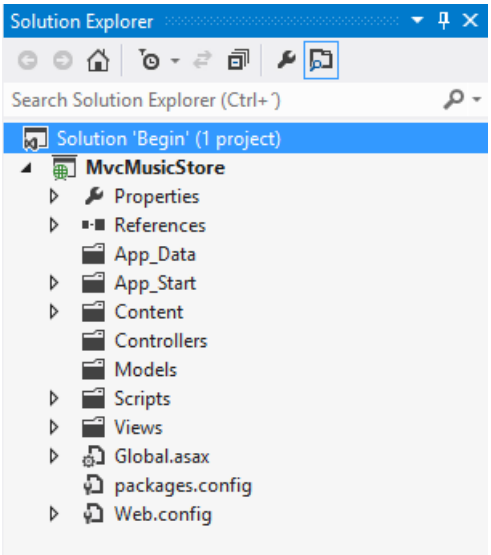
New ASP.NET MVC 4 Project Dialog Box

Task 2 - Exploring the Solution Structure

The ASP.NET MVC framework includes a Visual Studio project template that helps you create Web applications supporting the MVC pattern. This template creates a new ASP.NET MVC Web application with the required folders, item templates, and configuration-file entries.

In this task, you will examine the solution structure to understand the elements that are involved and their relationships. The following folders are included in all the ASP.NET MVC application because the ASP.NET MVC framework by default uses a "convention over configuration" approach, and makes some default assumptions based on folder naming conventions.

1. Once the project is created, review the folder structure that has been created in the Solution Explorer on the right side:



ASP.NET MVC Folder structure in Solution Explorer

a. **Controllers**. This folder will contain the controller classes. In an MVC based application, controllers are responsible for handling end user interaction, manipulating the model, and ultimately choosing a view to render the UI.

> **NOTE**
>
> The MVC framework requires the names of all controllers to end with "Controller"-for example, HomeController, LoginController, or ProductController.

b. **Models**. This folder is provided for classes that represent the application model for the MVC Web application. This usually includes code that defines objects and the logic for interacting with the data store. Typically, the actual model objects will be in separate class libraries. However, when you create a new application, you might include classes and then move them into separate class libraries at a later point in the development cycle.

c. **Views**. This folder is the recommended location for views, the components responsible for displaying the application's user interface. Views use .aspx, .ascx, .cshtml and .master files, in addition to any other files that are related to rendering views. Views folder contains a folder for each controller; the folder is named with the controller-name prefix. For example, if you have a controller named **HomeController**, the Views folder will contain a folder named Home. By default, when the ASP.NET MVC framework loads a view, it looks for an .aspx file with the requested view name in the Views\controllerName folder (**Views[ControllerName][Action].aspx**) or (**Views[ControllerName][Action].cshtml**) for Razor Views.

> **NOTE**
>
> In addition to the folders listed previously, an MVC Web application uses the **Global.asax** file to set global URL routing defaults, and it uses the **Web.config** file to configure the application.

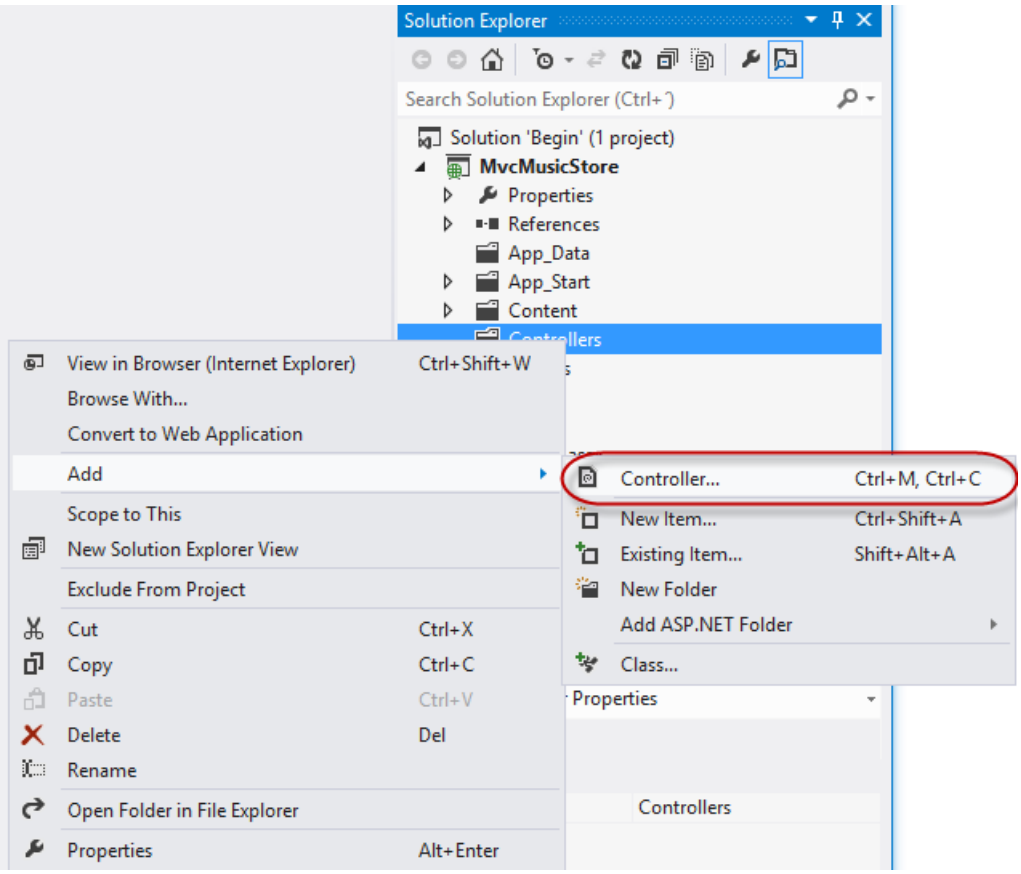
Task 3 - Adding a HomeController

In ASP.NET applications that do not use the MVC framework, user interaction is organized around pages, and around raising and handling events from those pages. In contrast, user interaction with ASP.NET MVC applications is organized around controllers and their action methods.

On the other hand, ASP.NET MVC framework maps URLs to classes that are referred to as controllers. Controllers process incoming requests, handle user input and interactions, execute appropriate application logic and determine the response to send back to the client (display HTML, download a file, redirect to a different URL, etc.). In the case of displaying HTML, a controller class typically calls a separate view component to generate the HTML markup for the request. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction.

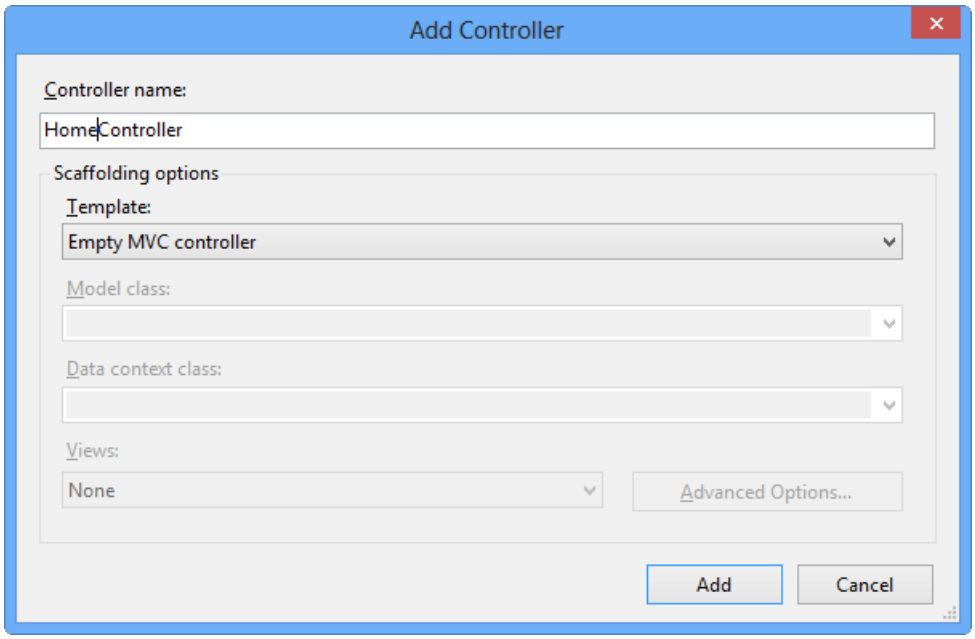
In this task, you will add a Controller class that will handle URLs to the Home page of the Music Store site.

1. Right-click **Controllers** folder within the Solution Explorer, select **Add** and then **Controller** command:



Add Controller Command

2. The **Add Controller** dialog appears. Name the controller *HomeController* and press **Add**.



Add Controller Dialog

3. The file **HomeController.cs** is created in the **Controllers** folder. In order to have the **HomeController** return a string on its Index action, replace the **Index** method with the following code:

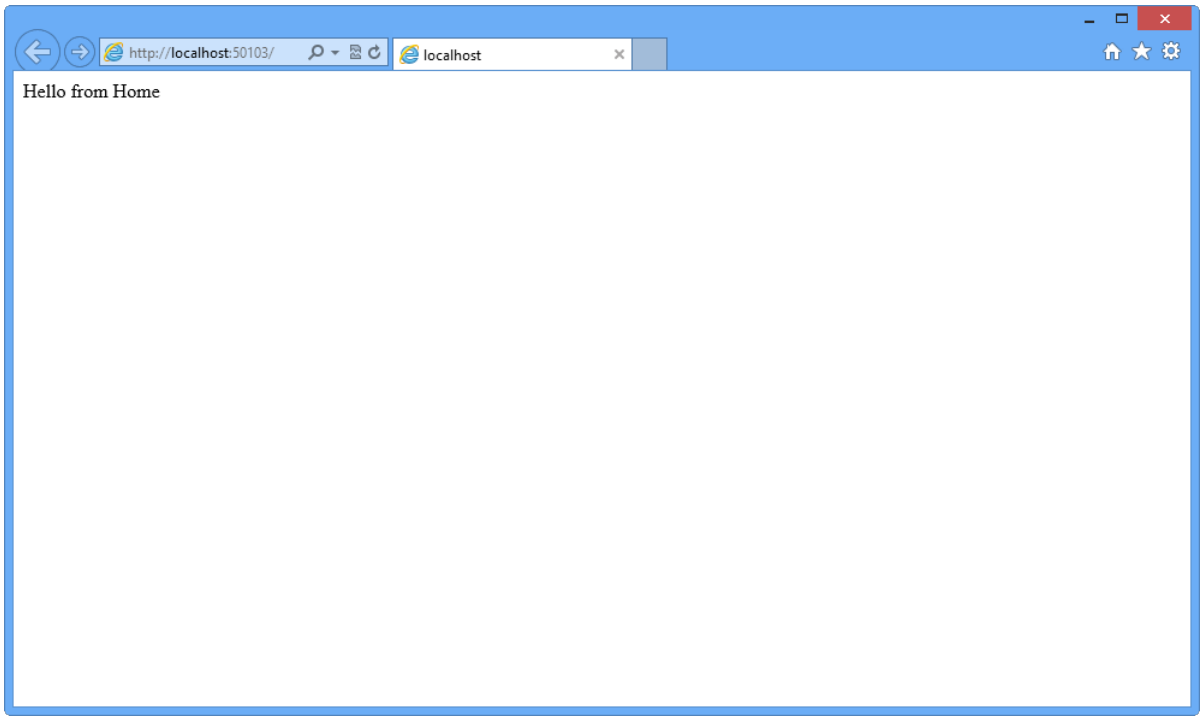
   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex1 HomeController Index*)

```
public string Index()
{
    return "Hello from Home";
}
```

Task 4 - Running the Application

In this task, you will try out the Application in a web browser.

1. Press **F5** to run the Application. The project is compiled and the local IIS Web Server starts. The local IIS Web Server will automatically open a web browser pointing to the URL of the Web server.



Application running in a web browser

> **NOTE**
> The local IIS Web Server will run the website on a random free port number. In the figure above, the site is running at `http://localhost:50103/`, so it's using port 50103. Your port number may vary.

2. Close the browser.

Exercise 2: Creating a Controller

In this exercise, you will learn how to update the controller to implement simple functionality of the Music Store application. That controller will define action methods to handle each of the following specific requests:

- A listing page of the music genres in the Music Store
- A browse page that lists all of the music albums for a particular genre
- A details page that shows information about a specific music album

For the scope of this exercise, those actions will simply return a string by now.

Task 1 - Adding a New StoreController Class

In this task, you will add a new Controller.

1. If not already open, start **VS Express for Web 2012**.

2. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to **Source\Ex02-CreatingAController\Begin**, select **Begin.sln** and click **Open**. Alternatively, you may continue with the solution that you obtained after completing the previous exercise.

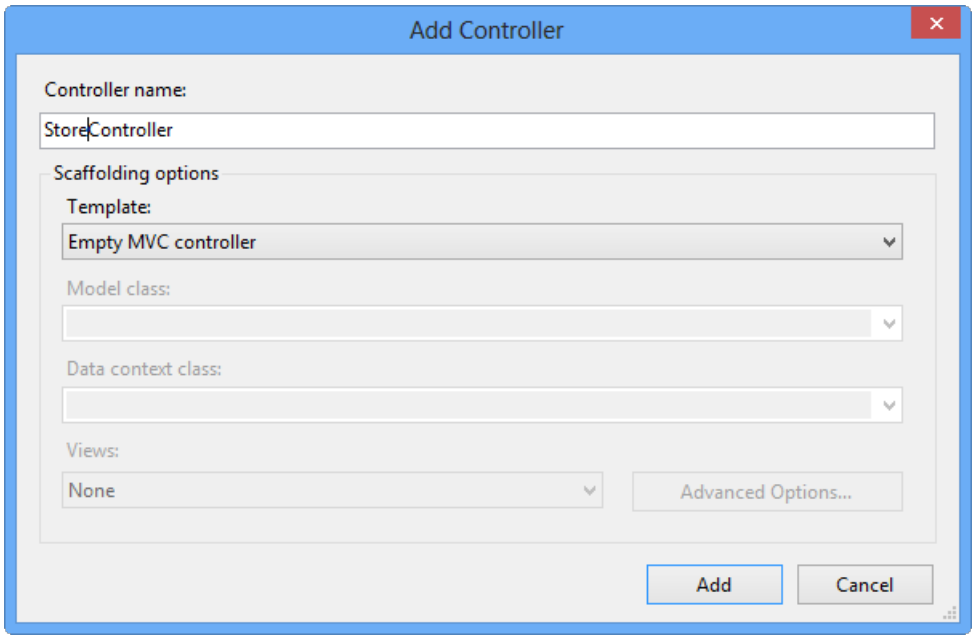
   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

3. Add the new controller. To do this, right-click the **Controllers** folder within the Solution Explorer, select **Add** and then the **Controller** command. Change the **Controller Name** to *StoreController*, and click **Add**.



Add Controller Dialog

Task 2 - Modifying the StoreController's Actions

In this task, you will modify the Controller methods that are called **actions**. Actions are responsible for handling URL requests and determining the content that should be sent back to the browser or user that invoked the URL.

1. The **StoreController** class already has an **Index** method. You will use it later in this Lab to implement the page that lists all genres of the music store. For now, just replace the **Index** method with the following code that returns a string "Hello from Store.Index()":

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex2 StoreController Index*)

```
public string Index()
{
  return "Hello from Store.Index()";
}
```

2. Add **Browse** and **Details** methods. To do this, add the following code to the **StoreController**:

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex2 StoreController BrowseAndDetails*)

```
// GET: /Store/Browse
public string Browse()
{
  return "Hello from Store.Browse()";
}

// GET: /Store/Details
public string Details()
{
  return "Hello from Store.Details()";
}
```

Task 3 - Running the Application

In this task, you will try out the Application in a web browser.

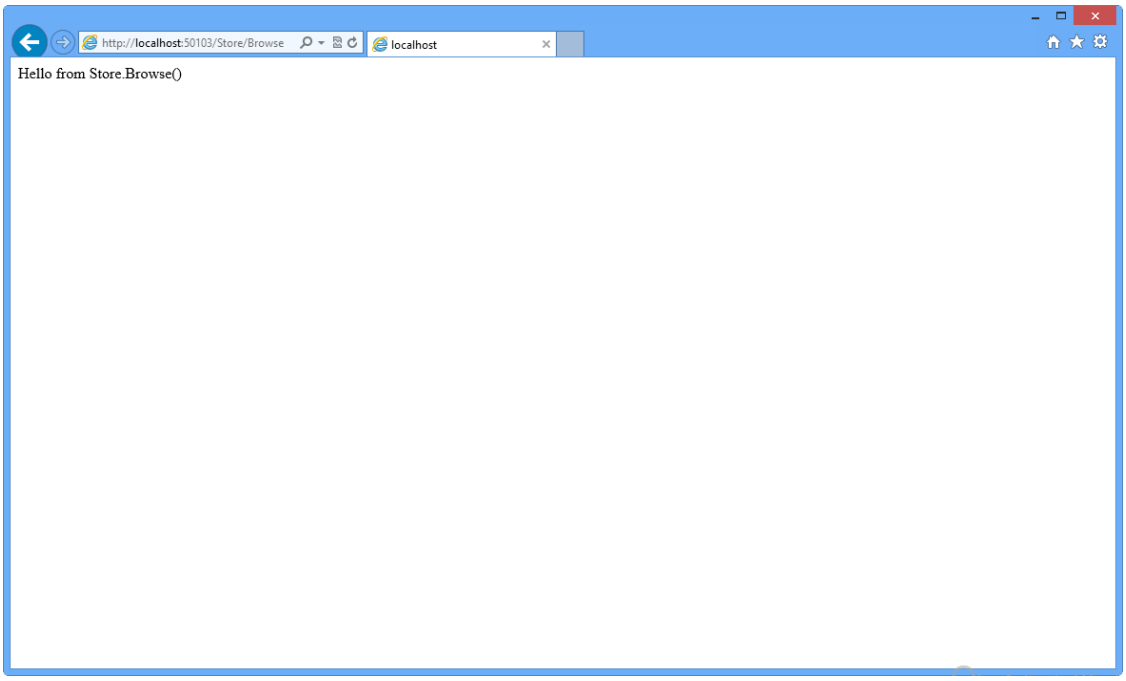
1. Press **F5** to run the Application.

2. The project starts in the **Home** page. Change the URL to verify each action's implementation.

   a. **/Store**. You will see **"Hello from Store.Index()"**.

b. **/Store/Browse**. You will see **"Hello from Store.Browse()"**.

c. **/Store/Details**. You will see **"Hello from Store.Details()"**.



Browsing /Store/Browse

3. Close the browser.

Exercise 3: Passing parameters to a Controller

Until now, you have been returning constant strings from the Controllers. In this exercise you will learn how to pass parameters to a Controller using the URL and querystring, and then making the method actions respond with text to the browser.

Task 1 - Adding Genre Parameter to StoreController

In this task, you will use the **querystring** to send parameters to the **Browse** action method in the **StoreController**.

1. If not already open, start **VS Express for Web**.

2. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to **Source\Ex03-PassingParametersToAController\Begin**, select **Begin.sln** and click **Open**. Alternatively, you may continue with the solution that you obtained after completing the previous exercise.

   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

3. Open **StoreController** class. To do this, in the **Solution Explorer**, expand the **Controllers** folder and double-click **StoreController.cs**.

4. Change the **Browse** method, adding a string parameter to request for a specific genre. ASP.NET MVC will automatically pass any querystring or form post parameters named **genre** to this action method when invoked. To do this, replace the **Browse** method with the following code:

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex3 StoreController BrowseMethod*)

```
// GET: /Store/Browse?genre=Disco
public string Browse(string genre)
{
  string message = HttpUtility.HtmlEncode("Store.Browse, Genre = " + genre);

  return message;
}
```

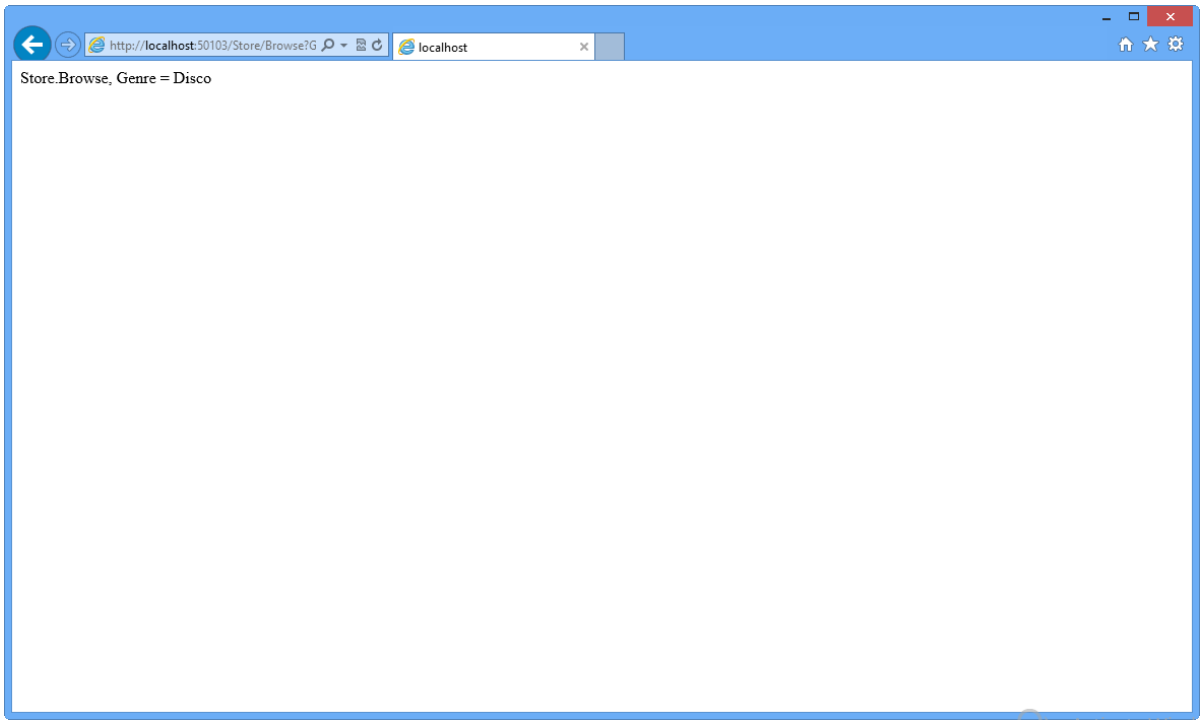
> **NOTE**
>
> You are using the **HttpUtility.HtmlEncode** utility method to prevents users from injecting Javascript into the View with a link like **/Store/Browse?Genre=<script>window.location='http://hackersite.com'</script>**.
>
> For further explanation, please visit this msdn article.

Task 2 - Running the Application

In this task, you will try out the Application in a web browser and use the **genre** parameter.

1. Press **F5** to run the Application.

2. The project starts in the **Home** page. Change the URL to */Store/Browse?Genre=Disco* to verify that the action receives the genre parameter.



   Browsing /Store/Browse?Genre=Disco

3. Close the browser.

Task 3 - Adding an Id Parameter Embedded in the URL

In this task, you will use the **URL** to pass an **Id** parameter to the **Details** action method of the **StoreController**. ASP.NET MVC's default routing convention is to treat the segment of a URL after the action method name as a parameter named **Id**. If your action method has parameter named Id, then ASP.NET MVC will automatically pass

the URL segment to you as a parameter. In the URL **Store/Details/5**, **Id** will be interpreted as **5**.

1. Change the **Details** method of the **StoreController**, adding an **int** parameter called **id**. To do this, replace the **Details** method with the following code:

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex3 StoreController DetailsMethod*)

```
// GET: /Store/Details/5
public string Details(int id)
{
  string message = "Store.Details, ID = " + id;

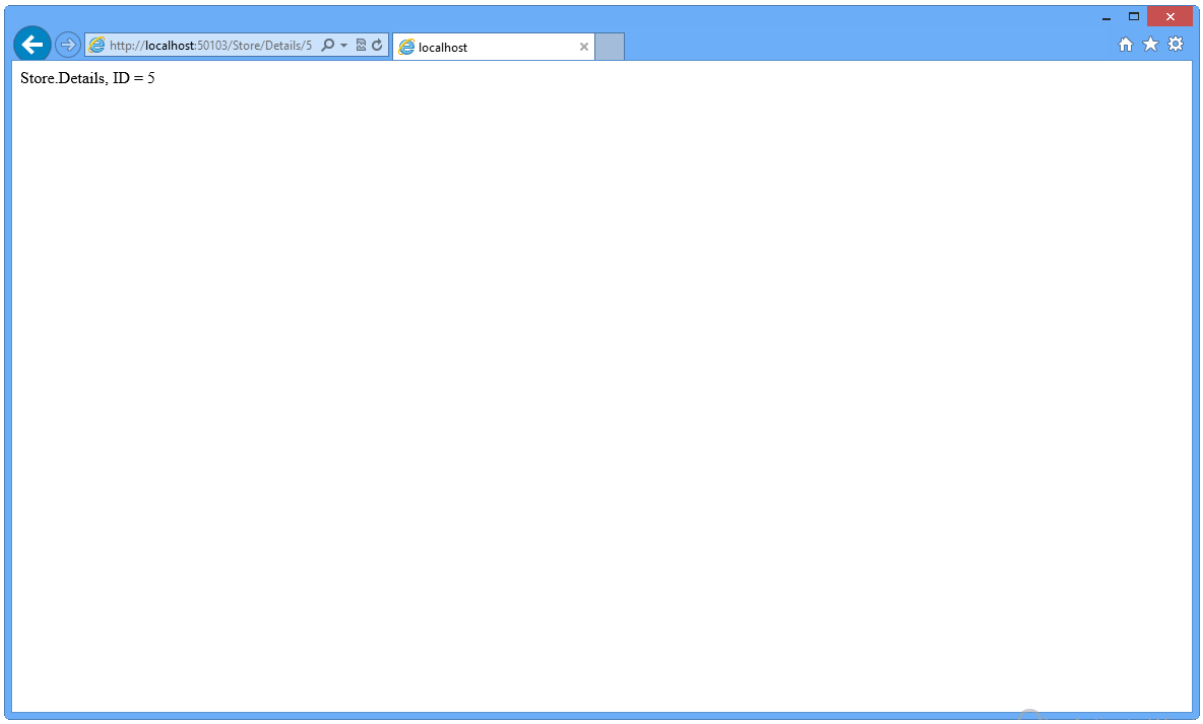
  return message;
}
```

Task 4 - Running the Application

In this task, you will try out the Application in a web browser and use the **Id** parameter.

1. Press **F5** to run the Application.

2. The project starts in the **Home** page. Change the URL to */Store/Details/5* to verify that the action receives the id parameter.



Browsing /Store/Details/5

Exercise 4: Creating a View

So far you have been returning strings from controller actions. Although that is a useful way of understanding how controllers work, it is not how your real Web applications are built. Views are components that provide a better approach for generating HTML back to the browser with the use of template files.

In this exercise you will learn how to add a layout master page to setup a template for common HTML content, a StyleSheet to enhance the look and feel of the site and finally a View template to enable HomeController to return HTML.

Task 1 - Modifying the file _layout.cshtml

The file **~/Views/Shared/_layout.cshtml** allows you to setup a template for common HTML to use across the entire website. In this task you will add a layout master page with a common header with links to the Home page and Store area.

1. If not already open, start **VS Express for Web**.
2. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to **Source\Ex04-CreatingAView\Begin**, select **Begin.sln** and click **Open**. Alternatively, you may continue with the solution that you obtained after completing the previous exercise.
   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.
   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.
   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

3. The file **_layout.cshtml** contains the HTML container layout for all pages on the site. It includes the **<html>** element for the HTML response, as well as the **<head>** and **<body>** elements. **@RenderBody()** within the HTML body identify regions that view templates will be able to fill in with dynamic content. (C#)

```
<!DOCTYPE html>
<html>
<head>
     <meta charset="utf-8" />
     <meta name="viewport" content="width=device-width" />
     <title>@ViewBag.Title</title>
     @Styles.Render("~/Content/css")
     @Scripts.Render("~/bundles/modernizr")
</head>
<body>
     @RenderBody()

     @Scripts.Render("~/bundles/jquery")
     @RenderSection("scripts", required: false)
</body>
</html>
```

4. Add a common header with links to the Home page and Store area on all pages in the site. In order to do that, add the following code below <body> statement. (C#)

```
<!DOCTYPE html>
<html>
<head>
     <meta charset="utf-8" />
     <meta name="viewport" content="width=device-width" />
     <title>@ViewBag.Title</title>
     @Styles.Render("~/Content/css")
     @Scripts.Render("~/bundles/modernizr")
</head>
<body>
     <header>
          <div class="content-wrapper">
                <div class="float-left">
                     <p class="site-title">@Html.ActionLink("ASP.NET MVC Music Store", "Index", "Home")
</p>
                </div>
                <div class="float-right">
                     <nav>
                          <ul id="menu">
                                <li>@Html.ActionLink("Home", "Index", "Home")</li>
                                <li>@Html.ActionLink("Store", "Index", "Store")</li>
                          </ul>
                     </nav>
                </div>
          </div>
     </header>
     @RenderBody()

     @Scripts.Render("~/bundles/jquery")
     @RenderSection("scripts", required: false)
</body>
</html>
```

5. Include a div to render the body section of each page. Replace **@RenderBody()** with the following highlighted code: (C#)

```
...
<body>
     <header>
          <div class="content-wrapper">
                <div class="float-left">
                     <p class="site-title">@Html.ActionLink("ASP.NET MVC Music Store", "Index", "Home")
</p>
                </div>
                <div class="float-right">
                     <nav>
                          <ul id="menu">
                                <li>@Html.ActionLink("Home", "Index", "Home")</li>
                                <li>@Html.ActionLink("Store", "Index", "Store")</li>
                          </ul>
                     </nav>
                </div>
          </div>
     </header>
     <div id="body">
          @RenderSection("featured", required: false)
          <section class="content-wrapper main-content clear-fix">
                @RenderBody()
          </section>
     </div>
     ...
</body>
</html>
```

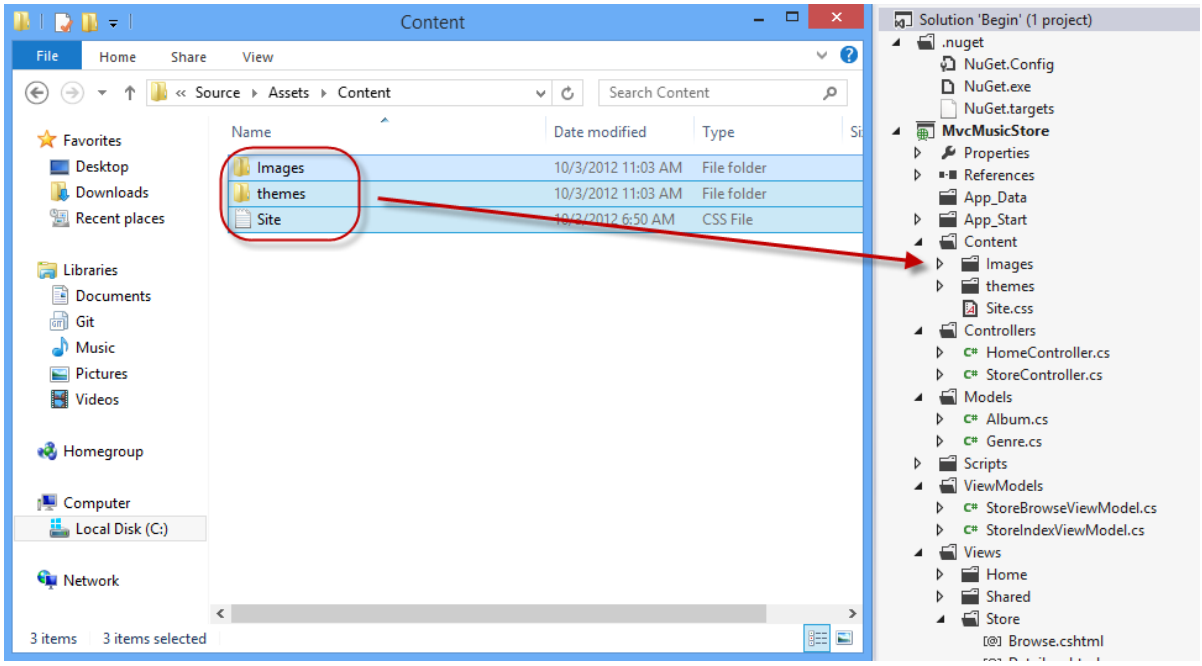
> **NOTE**
>
> Did you know? Visual Studio 2012 has snippets that make it easy to add commonly used code in HTML, code files and more! Try it out by typing ** ** and pressing **TAB** twice to insert a complete **div** tag.

Task 2 - Adding CSS Stylesheet

The empty project template includes a very streamlined CSS file which just includes styles used to display basic forms and validation messages. You will use additional CSS and images (potentially provided by a designer) in order to enhance the look and feel of the site.

In this task, you will add a CSS stylesheet to define the styles of the site.

1. The CSS file and images are included in the **Source\Assets\Content** folder of this Lab. In order to add them to the application, drag their content from a **Windows Explorer** window into the **Solution Explorer** in Visual Studio Express for Web, as shown below:



Dragging style contents

2. A warning dialog will appear, asking for confirmation to replace **Site.css** file and some existing images. Check **Apply to all items** and click **Yes**.

Task 3 - Adding a View Template

In this task, you will add a View template to generate the HTML response that will use the layout master page and CSS added in this exercise.

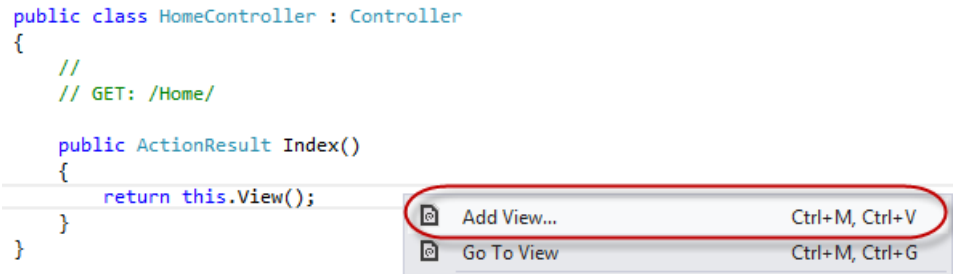
1. To use a View template when browsing the site's home page, you will first need to indicate that instead of returning a string, the **HomeController Index** method will return a **View**. Open **HomeController** class and change its **Index** method to return an **ActionResult**, and have it return **View()**.

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex4 HomeController Index*)

```
public class HomeController : Controller
{
    //
    // GET: /Home/

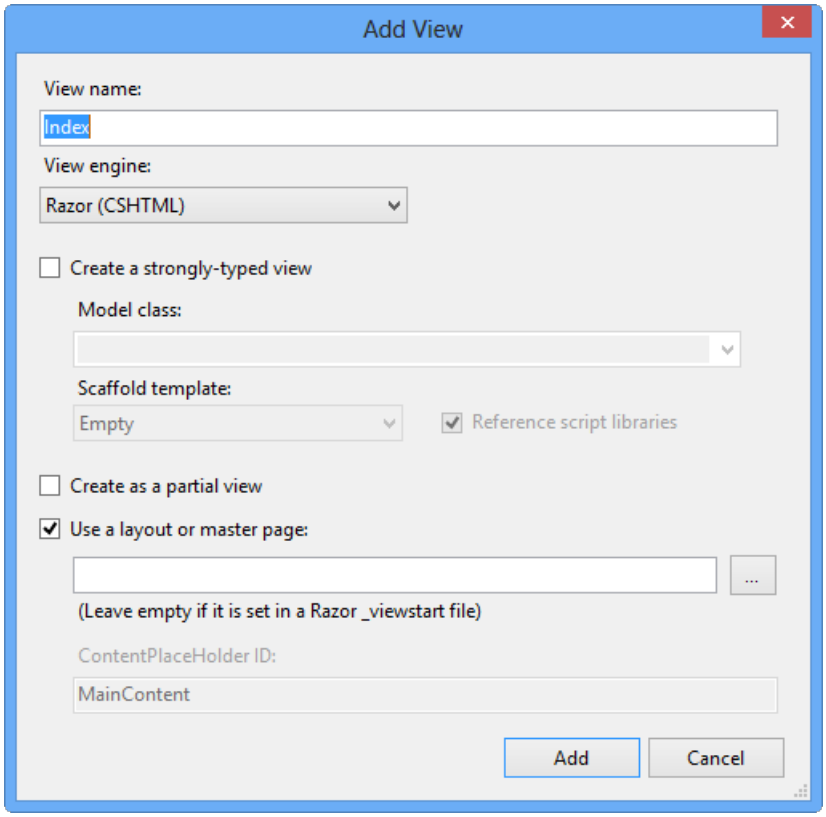
    public ActionResult Index()
    {
        return this.View();
    }
}
```

2. Now, you need to add an appropriate View template. To do this, **right-click** inside the **Index** action method and select **Add View**. This will bring up the **Add View** dialog.



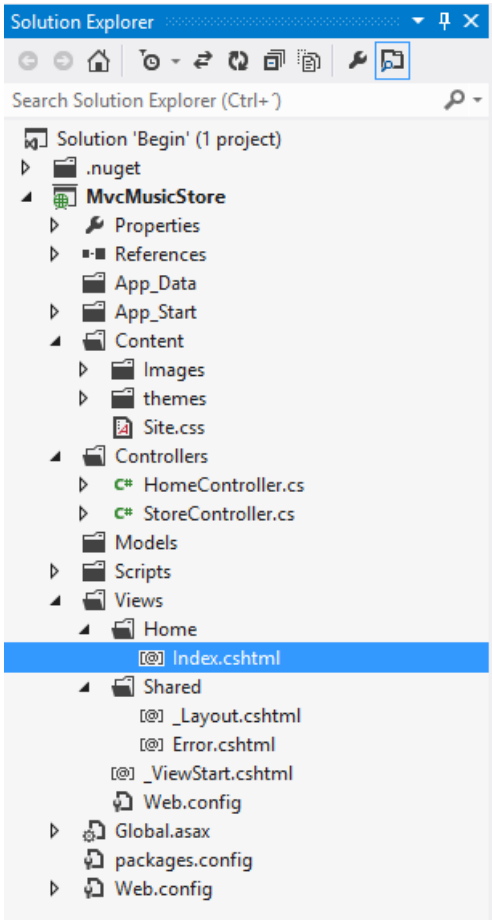
Adding a View from within the Index method

3. The **Add View** Dialog will appear to generate a View template file. By default, this dialog pre-populates the name of the View template so that it matches the action method that will use it. Because you used the **Add View** context menu within the **Index** action method within the HomeController, the **Add View** dialog has Index as the default view name. Click **Add**.



Add View Dialog

4. Visual Studio generates an **Index.cshtml** view template inside the **Views\Home** folder and then opens it.



Home Index view created

> **NOTE**
>
> name and location of the **Index.cshtml** file is relevant and follows the default ASP.NET MVC naming conventions.
>
> The folder \Views*Home* matches the controller name (**Home** Controller). The View template name (**Index**), matches the controller action method which will be displaying the View.
>
> This way, ASP.NET MVC avoids having to explicitly specify the name or location of a View template when using this naming convention to return a View.

5. The generated View template is based on the **_layout.cshtml** template earlier defined. Update the ViewBag.Title property to **Home**, and change the main content to **This is the Home Page**, as shown in the code below:

```
@{
    ViewBag.Title = "Home";
}

<h2>This is the Home Page</h2>
```

6. Select **MvcMusicStore** project in the Solution Explorer and Press **F5** to run the Application.

Task 4: Verification

In order to verify that you have correctly performed all the steps in the previous exercise, proceed as follows:

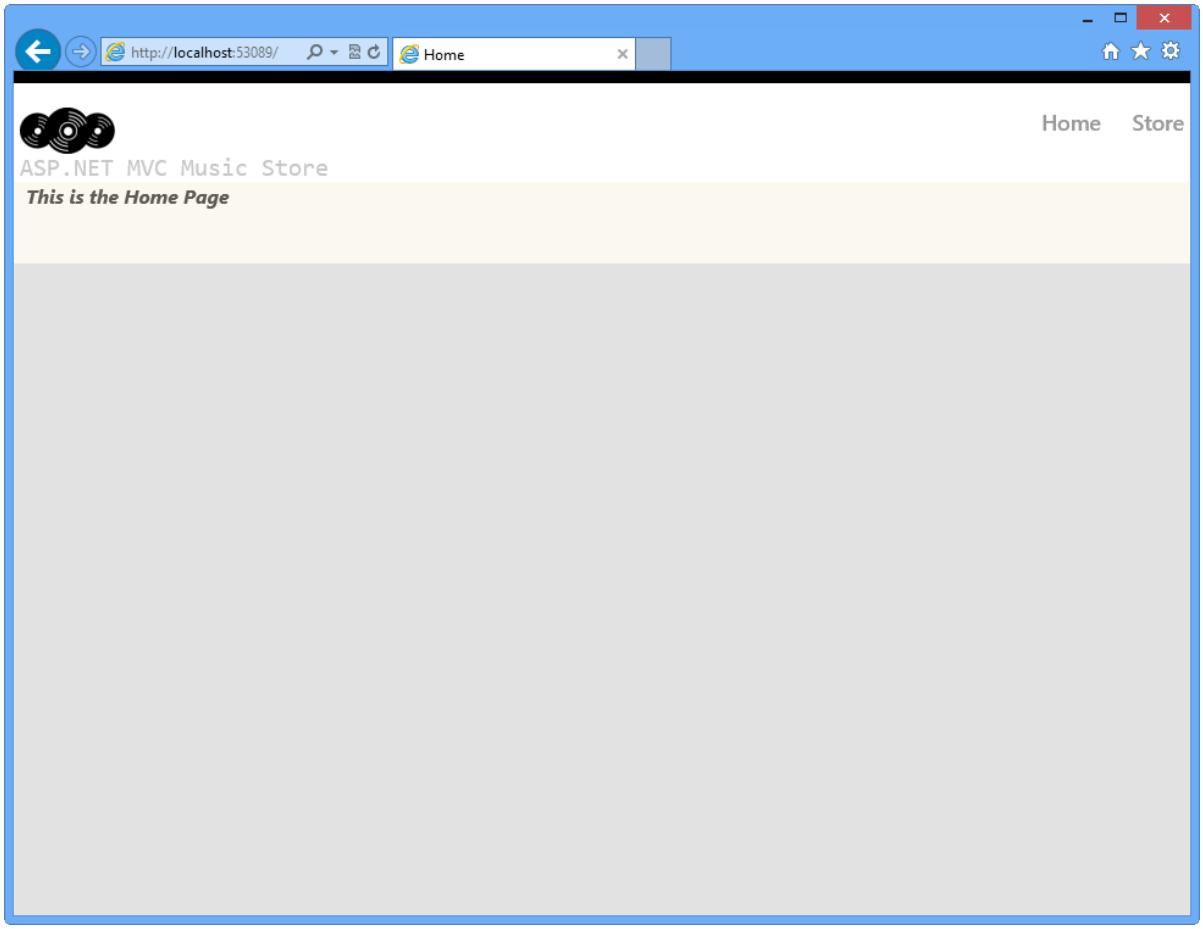
With the application opened in a browser, you should note that:

1. The HomeController's Index action method found and displayed the **\Views\Home\Index.cshtml** View template, even though the code called **return View()**, because the View template followed the standard

naming convention.

2. The Home Page displays the welcome message defined within the **\Views\Home\Index.cshtml** view template.

3. The Home Page is using the **_layout.cshtml** template, and so the welcome message is contained within the standard site HTML layout.



Home Index View using the defined LayoutPage and style

Exercise 5: Creating a View Model

So far, you made your Views display hardcoded HTML, but, in order to create dynamic web applications, the View template should receive information from the Controller. One common technique to be used for that purpose is the **ViewModel** pattern, which allows a Controller to package up all the information needed to generate the appropriate HTML response.

In this exercise, you will learn how to create a ViewModel class and add the required properties: the number of genres in the store and a list of those genres. You will also update the StoreController to use the created ViewModel, and finally, you will create a new View template that will display the mentioned properties in the page.

Task 1 - Creating a ViewModel Class

In this task, you will create a ViewModel class that will implement the Store genre listing scenario.

1. If not already open, start **VS Express for Web**.

2. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to **Source\Ex05-CreatingAViewModel\Begin**, select **Begin.sln** and click **Open**. Alternatively, you may continue with the solution that you obtained after completing the previous exercise.

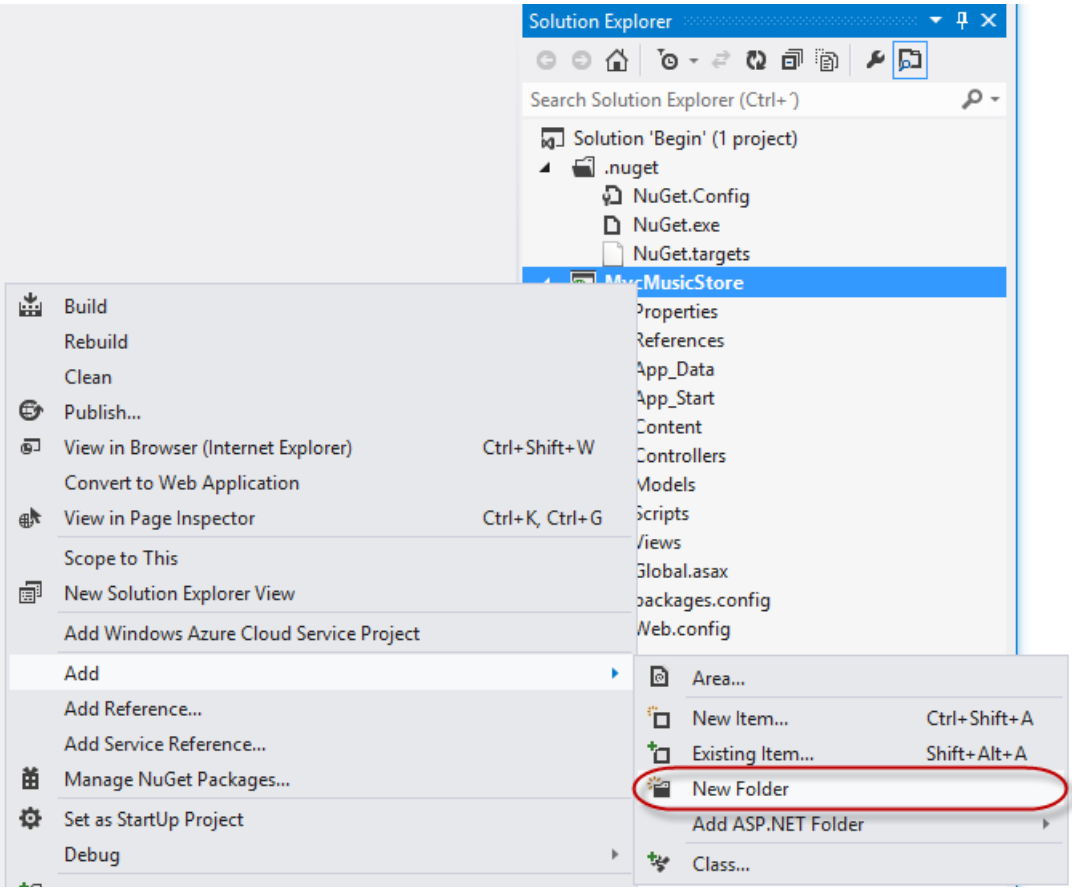
   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

c. Finally, build the solution by clicking **Build | Build Solution**.

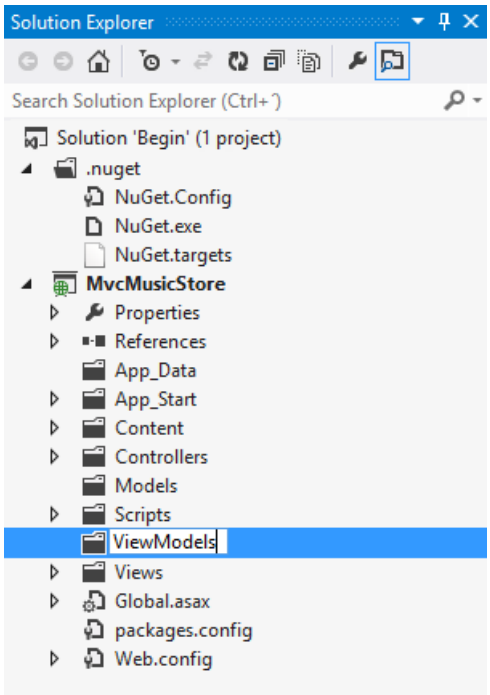
> **NOTE**
>
> One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

3. Create a **ViewModels** folder to hold the ViewModel. To do this, right-click the top-level **MvcMusicStore** project, select **Add** and then **New Folder**.



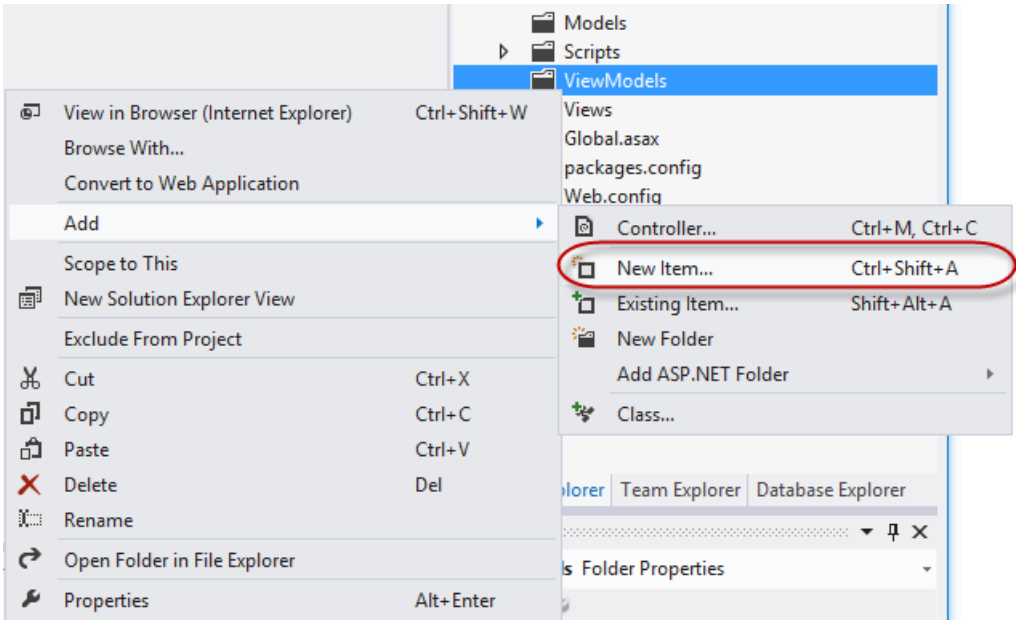
Adding a new folder

4. Name the folder *ViewModels*.

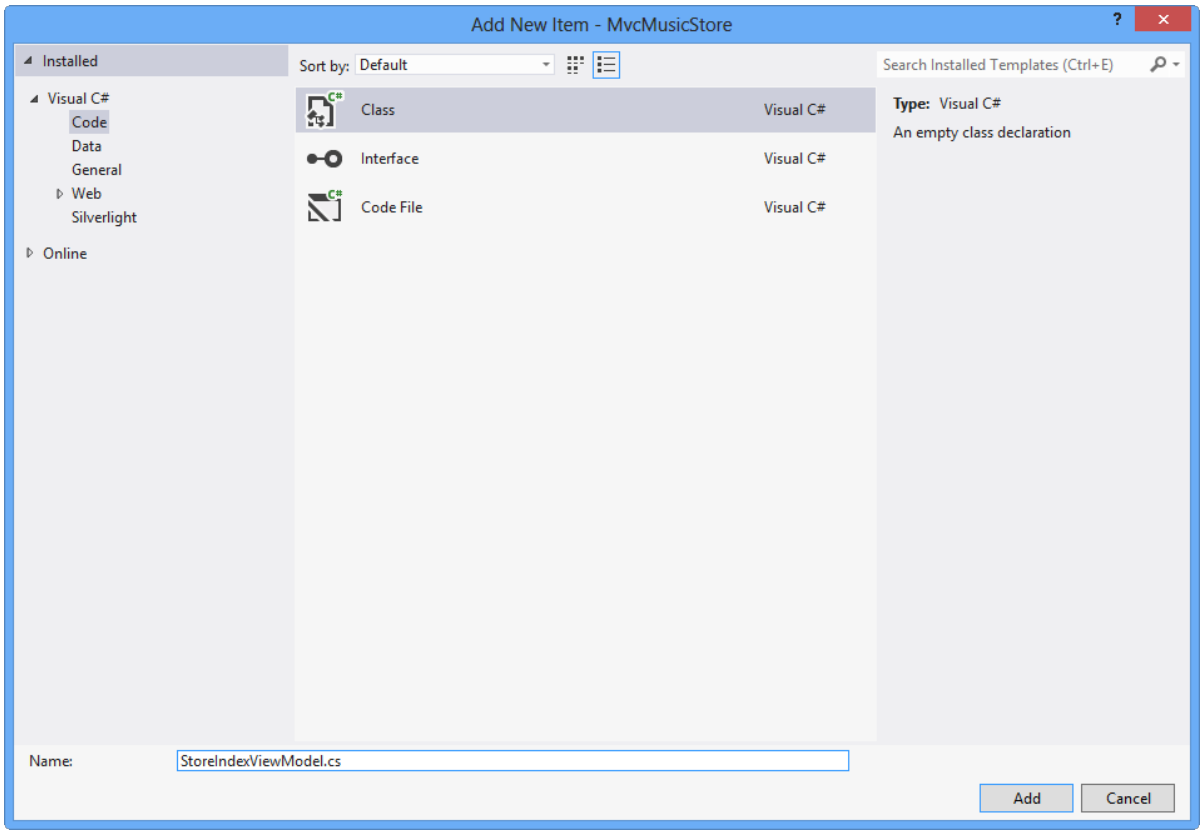


ViewModels folder in Solution Explorer

5. Create a **ViewModel** class. To do this, right-click on the **ViewModels** folder recently created, select **Add** and then **New Item**. Under **Code**, choose the **Class** item and name the file *StoreIndexViewModel.cs*, then click **Add**.



Adding a new Class



Creating StoreIndexViewModel class

Task 2 - Adding Properties to the ViewModel class

There are two parameters to be passed from the StoreController to the View template in order to generate the expected HTML response: the number of genres in the store and a list of those genres.

In this task, you will add those 2 properties to the **StoreIndexViewModel** class: **NumberOfGenres** (an integer) and **Genres** (a list of strings).

1. Add **NumberOfGenres** and **Genres** properties to the **StoreIndexViewModel** class. To do this, add the following 2 lines to the class definition:

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex5 StoreIndexViewModel properties*)

```
public class StoreIndexViewModel
{
    public int NumberOfGenres { get; set; }

    public List<string> Genres { get; set; }
}
```

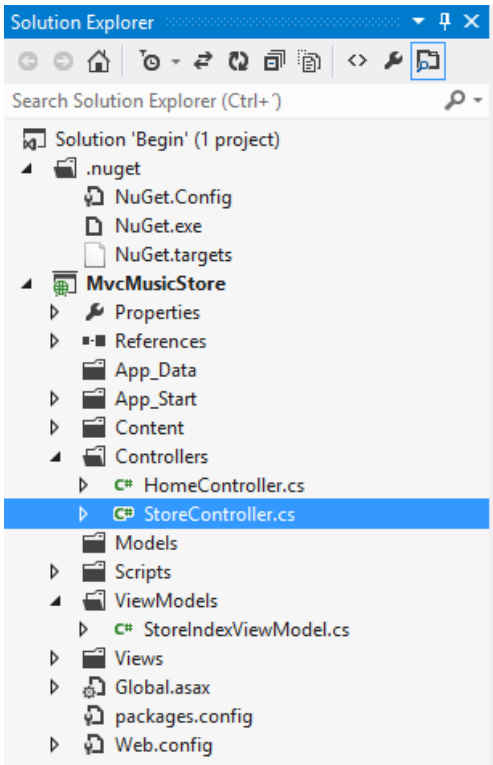
> **NOTE**
>
> The **{ get; set; }** notation makes use of C#'s auto-implemented properties feature. It provides the benefits of a property without requiring us to declare a backing field.

Task 3 - Updating StoreController to use the StoreIndexViewModel

The **StoreIndexViewModel** class encapsulates the information needed to pass from **StoreController**'s **Index** method to a View template in order to generate a response.

In this task, you will update the **StoreController** to use the **StoreIndexViewModel**.

1. Open **StoreController** class.



Opening StoreController class

2. In order to use the **StoreIndexViewModel** class from the **StoreController**, add the following namespace at the top of the **StoreController** code:

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex5 StoreIndexViewModel using ViewModels*)

```
using MvcMusicStore.ViewModels;
```

3. Change the **StoreController**'s **Index** action method so that it creates and populates a **StoreIndexViewModel** object and then passes it off to a View template to generate an HTML response with it.

> **NOTE**
>
> In Lab "ASP.NET MVC Models and Data Access" you will write code that retrieves the list of store genres from a database. In the following code, you will create a **List** of dummy data genres that will populate the **StoreIndexViewModel**.
>
> After creating and setting up the **StoreIndexViewModel** object, it will be passed as an argument to the **View** method. This indicates that the View template will use that object to generate an HTML response with it.

4. Replace the **Index** method with the following code:

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex5 StoreController Index method*)

```
public ActionResult Index()
{
    // Create a list of genres
    var genres = new List<string> {"Rock", "Jazz", "Country", "Pop", "Disco"};

    // Create our view model
    var viewModel = new StoreIndexViewModel {
        NumberOfGenres = genres.Count(),
        Genres = genres
    };

    return this.View(viewModel);
}
```

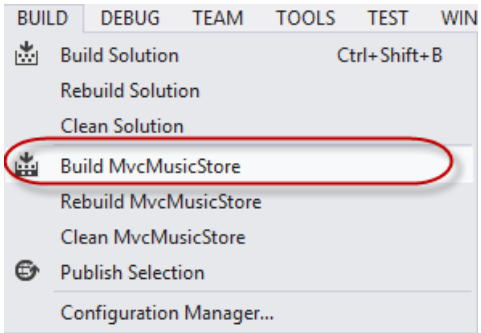
NOTE

If you're unfamiliar with C#, you may assume that using **var** means that the **viewModel** variable is late-bound. That's not correct - the C# compiler is using type-inference based on what you assign to the variable to determine that **viewModel** is of type **StoreIndexViewModel**. Also, by compiling the local **viewModel** variable as a **StoreIndexViewModel** type you get compile-time checking and Visual Studio code-editor support.

Task 4 - Creating a View Template that Uses StoreIndexViewModel

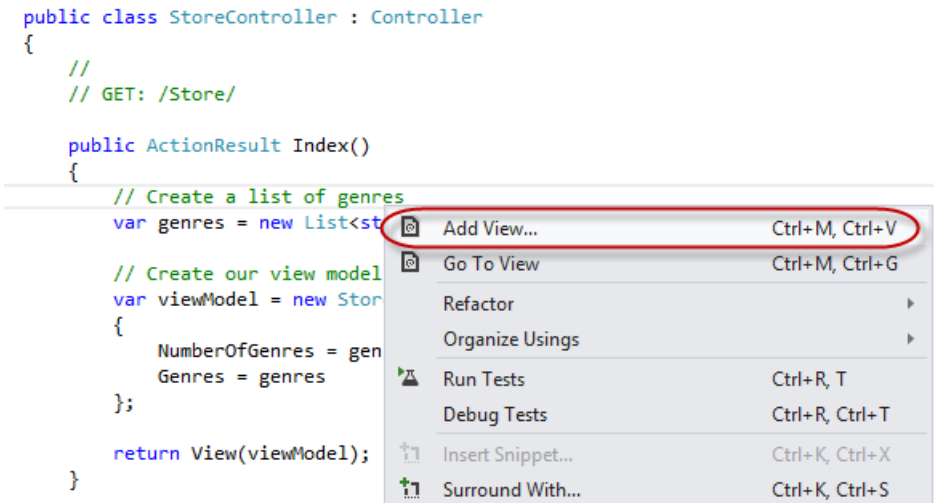
In this task, you will create a View template that will use a StoreIndexViewModel object passed from the Controller to display a list of genres.

1. Before creating the new View template, let's build the project so that the **Add View Dialog** knows about the **StoreIndexViewModel** class. Build the project by selecting the **Build** menu item and then **Build MvcMusicStore**.



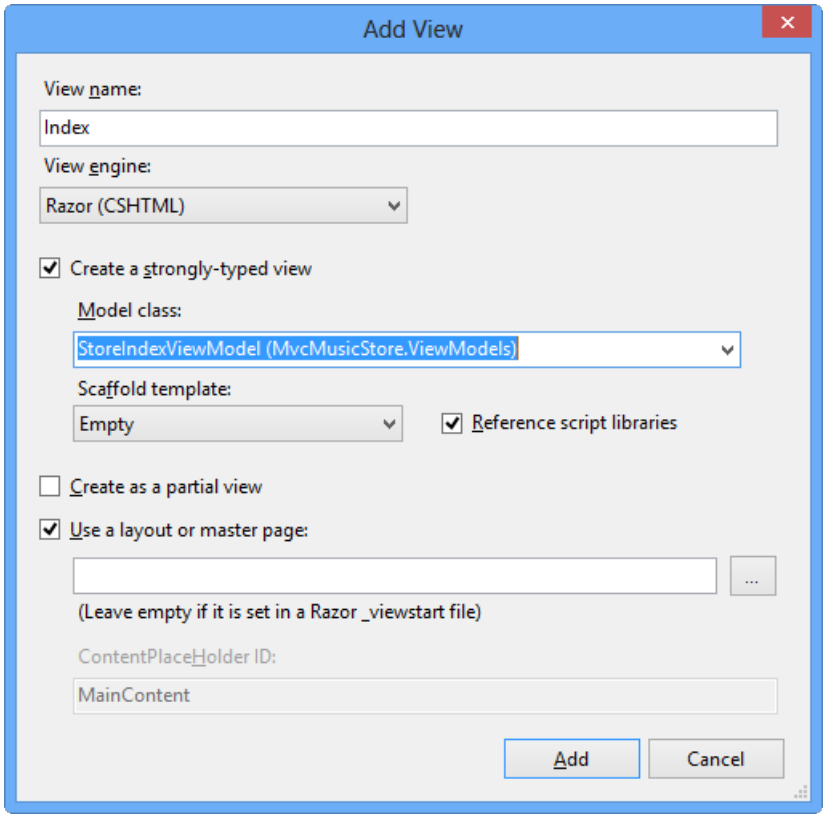
Building the project

2. Create a new View template. To do that, right-click inside the **Index** method and select **Add View**.



Adding a View

3. Because the **Add View Dialog** was invoked from the **StoreController**, it will add the View template by default in a **\Views\Store\Index.cshtml** file. Check the **Create a strongly-typed-view** checkbox and then select **StoreIndexViewModel** as the **Model class**. Also, make sure that the View engine selected is **Razor**. Click **Add**.



Add View Dialog

The **\Views\Store\Index.cshtml** View template file is created and opened. Based on the information provided to the **Add View** dialog in the last step, the View template will expect a **StoreIndexViewModel** instance as the data to use to generate an HTML response. You will notice that the template inherits a `ViewPage<musicstore.viewmodels.storeindexviewmodel>` in C#.

Task 5 - Updating the View Template

In this task, you will update the View template created in the last task to retrieve the number of genres and their names within the page.

> **NOTE**
>
> You will use @ syntax (often referred to as "code nuggets") to execute code within the View template.

1. In the **Index.cshtml** file, within the **Store** folder, replace its code with the following:

```
@model MvcMusicStore.ViewModels.StoreIndexViewModel

@{
    ViewBag.Title = "Browse Genres";
}

<h2>Browse Genres</h2>

<p>Select from @Model.NumberOfGenres genres</p>
```

```
> [!NOTE]
> As soon as you finish typing the period after the word **Model**, Visual Studio's Intellisense will show a
list of possible properties and methods to choose from.
>
> ![](aspnet-mvc-4-fundamentals/_static/image25.png)
>
> *Getting Model properties and methods with Visual Studio's IntelliSense*
>
> The **Model** property references the **StoreIndexViewModel** object that the Controller passed to the View
template. This means that you can access all of the data passed from the Controller to the View template via
the **Model** property, and format it into an appropriate HTML response within the View template.
>
> You can just select the **NumberOfGenres** property from the Intellisense list rather than typing it in and
then it will auto-complete it by pressing the **tab key**.
```

2. Loop over the genre list in **StoreIndexViewModel** and create an HTML **<ul>** list using a **foreach** loop. (C#)

```
@model MvcMusicStore.ViewModels.StoreIndexViewModel

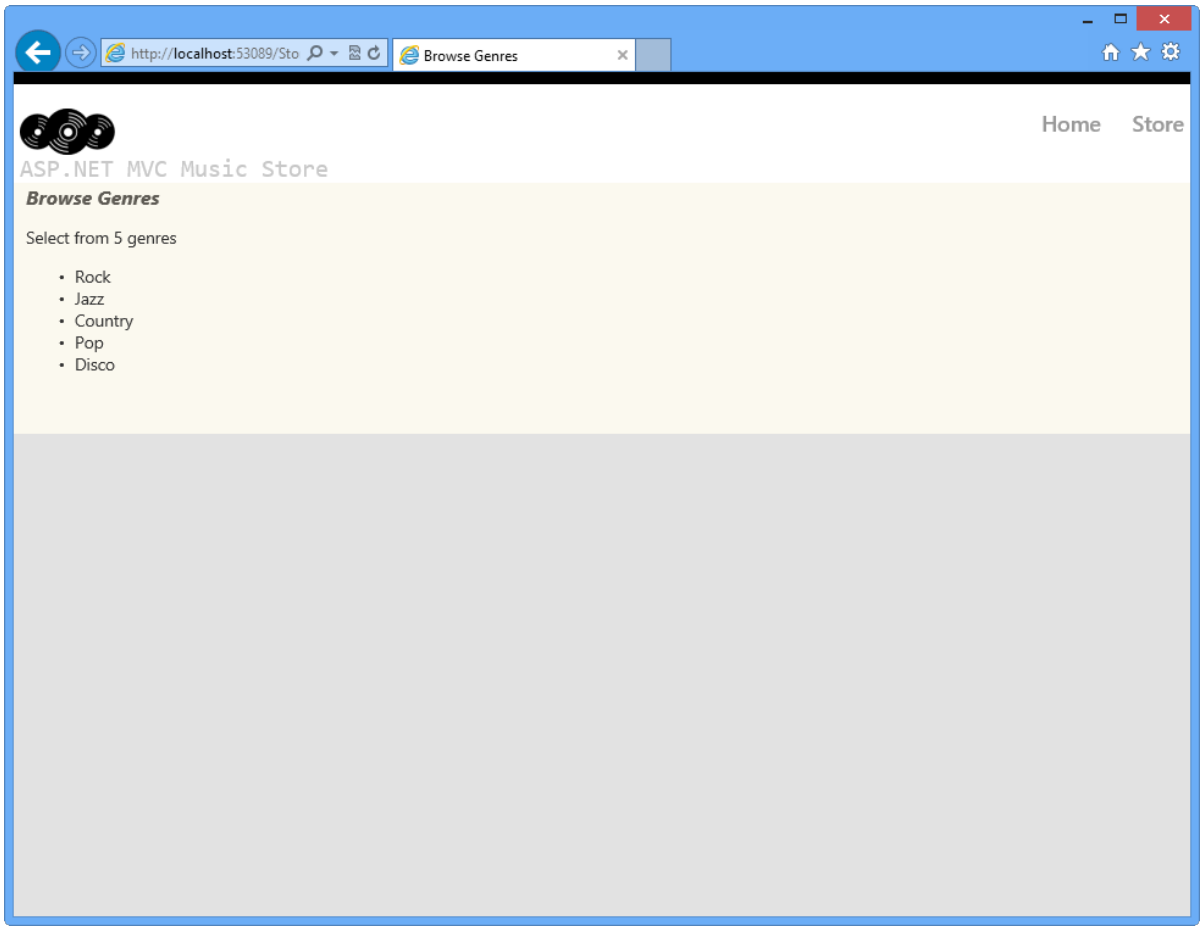
@{
    ViewBag.Title = "Browse Genres";
}

<h2>Browse Genres</h2>

<p>Select from @Model.NumberOfGenres genres</p>

<ul>
@foreach (string genreName in Model.Genres)
{
    <li>
        @genreName
    </li>
}
</ul>
```

3. Press **F5** to run the Application and browse **/Store**. You will see the list of genres passed in the **StoreIndexViewModel** object from the **StoreController** to the View template.



View displaying a list of genres

4. Close the browser.

Exercise 6: Using Parameters in View

In Exercise 3 you learned how to pass parameters to the Controller. In this exercise, you will learn how to use those parameters in the View template. For that purpose, you will be introduced first to Model classes that will help you to manage your data and domain logic. Additionally, you will learn how to create links to pages inside the ASP.NET MVC application without worrying of things like URL paths encoding.

Task 1 - Adding Model Classes

Unlike ViewModels, which are created just to pass information from the Controller to the View, Model classes are built to contain and manage data and domain logic. In this task you will add two model classes to represent these concepts: **Genre** and **Album**.

1. If not already open, start **VS Express for Web**

2. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to **Source\Ex06-UsingParametersInView\Begin**, select **Begin.sln** and click **Open**. Alternatively, you may continue with the solution that you obtained after completing the previous exercise.

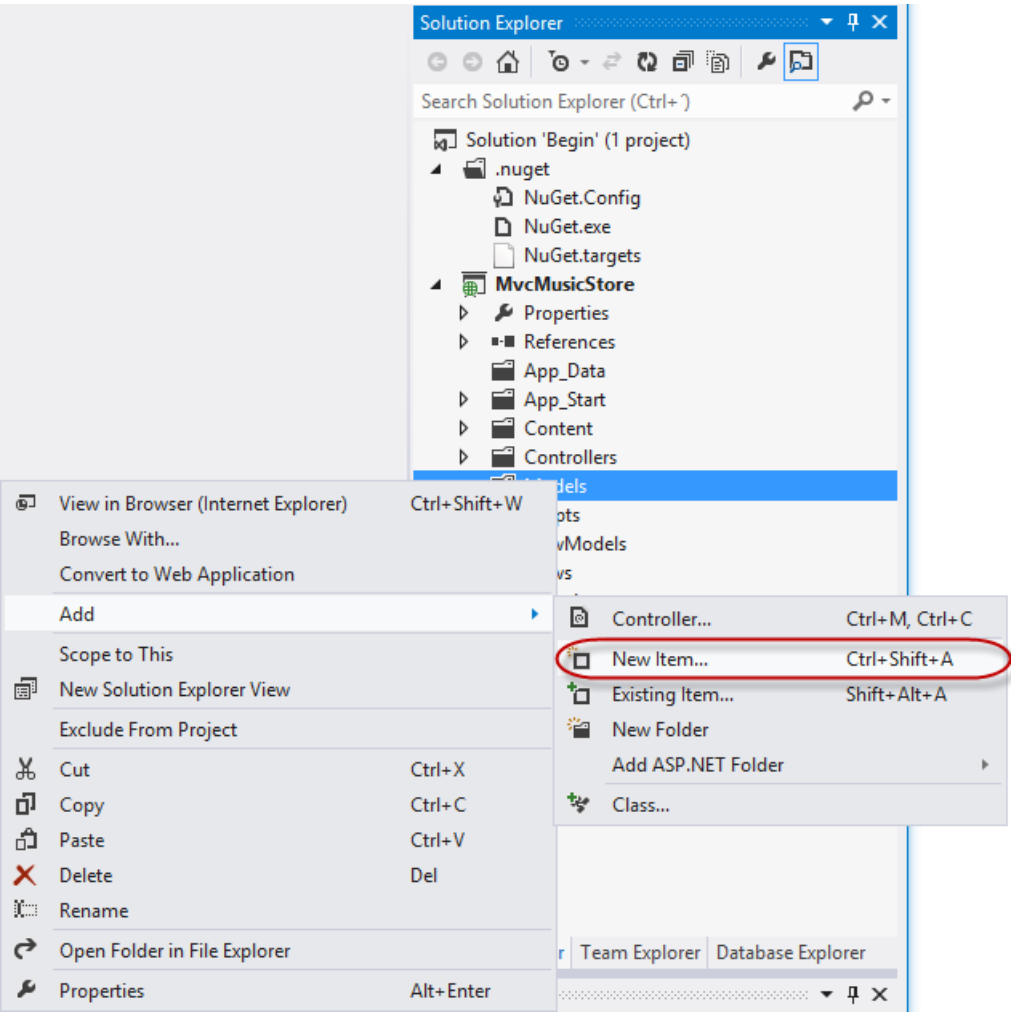
   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

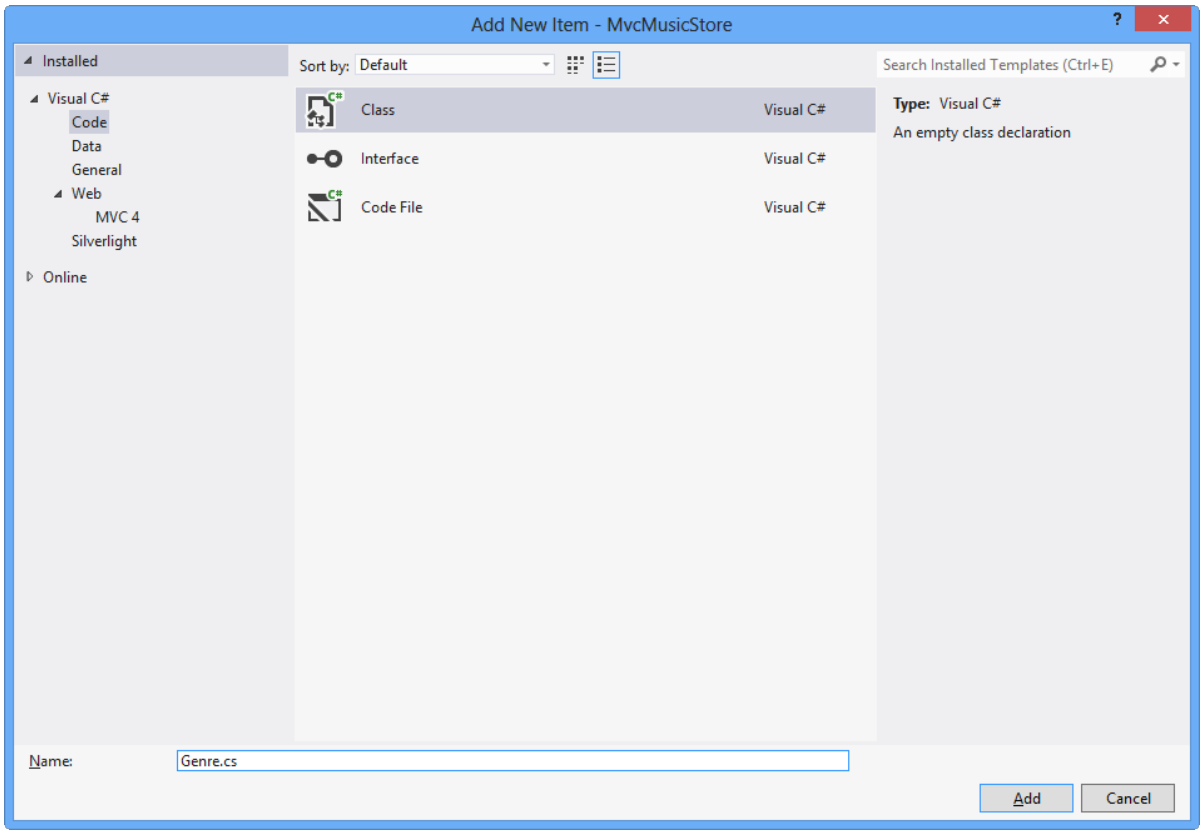
   c. Finally, build the solution by clicking **Build | Build Solution**.

> **NOTE**
>
> One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

3. Add a **Genre** Model class. To do this, right-click the **Models** folder in the **Solution Explorer**, select **Add** and then the **New Item** option. Under **Code**, choose the **Class** item and name the file *Genre.cs*, then click **Add**.



Adding a new item



Add Genre Model Class

4. Add a **Name** property to the Genre class. To do this, add the following code:

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex6 Genre*)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace MvcMusicStore.Models
{
    public class Genre
    {
        public string Name { get; set; }
    }
}
```

5. Following the same procedure as before, add an **Album** class. To do this, right-click the **Models** folder in the **Solution Explorer**, select **Add** and then the **New Item** option. Under **Code**, choose the **Class** item and name the file *Album.cs*, then click **Add**.

6. Add two properties to the Album class: **Genre** and **Title**. To do this, add the following code:

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex6 Album*)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace MvcMusicStore.Models
{
    public class Album
    {
        public string Title { get; set; }

        public Genre Genre { get; set; }
    }
}
```

Task 2 - Adding a StoreBrowseViewModel

A **StoreBrowseViewModel** will be used in this task to show the Albums that match a selected Genre. In this task, you will create this class and then add two properties to handle the **Genre** and its **Album**'s List.

1. Add a **StoreBrowseViewModel** class. To do this, right-click the **ViewModels** folder in the **Solution Explorer**, select **Add** and then the **New Item** option. Under **Code**, choose the **Class** item and name the file *StoreBrowseViewModel.cs*, then click **Add**.

2. Add a reference to the Models in **StoreBrowseViewModel** class. To do this, add the following using namespace:

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex6 UsingModel*)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using MvcMusicStore.Models;

namespace MvcMusicStore.ViewModels
{
    public class StoreBrowseViewModel
    {
    }
}
```

3. Add two properties to **StoreBrowseViewModel** class: **Genre** and **Albums**. To do this, add the following code:

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex6 ModelProperties*)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using MvcMusicStore.Models;

namespace MvcMusicStore.ViewModels
{
    public class StoreBrowseViewModel
    {
        public Genre Genre { get; set; }

        public List<Album> Albums { get; set; }
    }
}
```

NOTE

What is **List<Album>** ?: This definition is using the **List<T>** type, where **T** constrains the type to which elements of this **List** belong to, in this case **Album** (or any of its descendants).

This ability to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code is a feature of the C# language called **Generics**.

List<T> is the generic equivalent of the **ArrayList** type and is available in the **System.Collections.Generic** namespace. One of the benefits of using **generics** is that since the type is specified, you do not need to take care of type checking operations such as casting the elements into **Album** as you would do with an **ArrayList**.

Task 3 - Using the New ViewModel in the StoreController

In this task, you will modify the **StoreController**'s **Browse** and **Details** action methods to use the new **StoreBrowseViewModel**.

1. Add a reference to the Models folder in **StoreController** class. To do this, expand the **Controllers** folder in the **Solution Explorer** and open the **StoreController** class. Then add the following code:

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex6 UsingModelInController*)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MvcMusicStore.ViewModels;
using MvcMusicStore.Models;
```

2. Replace the **Browse** action method to use the **StoreViewBrowseController** class. You will create a Genre and two new Albums objects with dummy data (in the next Hands-on Lab you will consume real data from a database). To do this, replace the **Browse** method with the following code:

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex6 BrowseMethod*)

```
//
// GET: /Store/Browse?genre=Disco

public ActionResult Browse(string genre)
{
    var genreModel = new Genre()
    {
        Name = genre
    };

    var albums = new List<Album>()
    {
        new Album() { Title = genre + " Album 1" },
        new Album() { Title = genre + " Album 2" }
    };

    var viewModel = new StoreBrowseViewModel()
    {
        Genre = genreModel,
        Albums = albums
    };

    return this.View(viewModel);
}
```

3. Replace the **Details** action method to use the **StoreViewBrowseController** class. You will create a new **Album** object to be returned to the **View**. To do this, replace the **Details** method with the following code:

   (Code Snippet - *ASP.NET MVC 4 Fundamentals - Ex6 DetailsMethod*)

```
//
// GET: /Store/Details/5

public ActionResult Details(int id)
{
  var album = new Album { Title = "Sample Album" };

  return this.View(album);
}
```

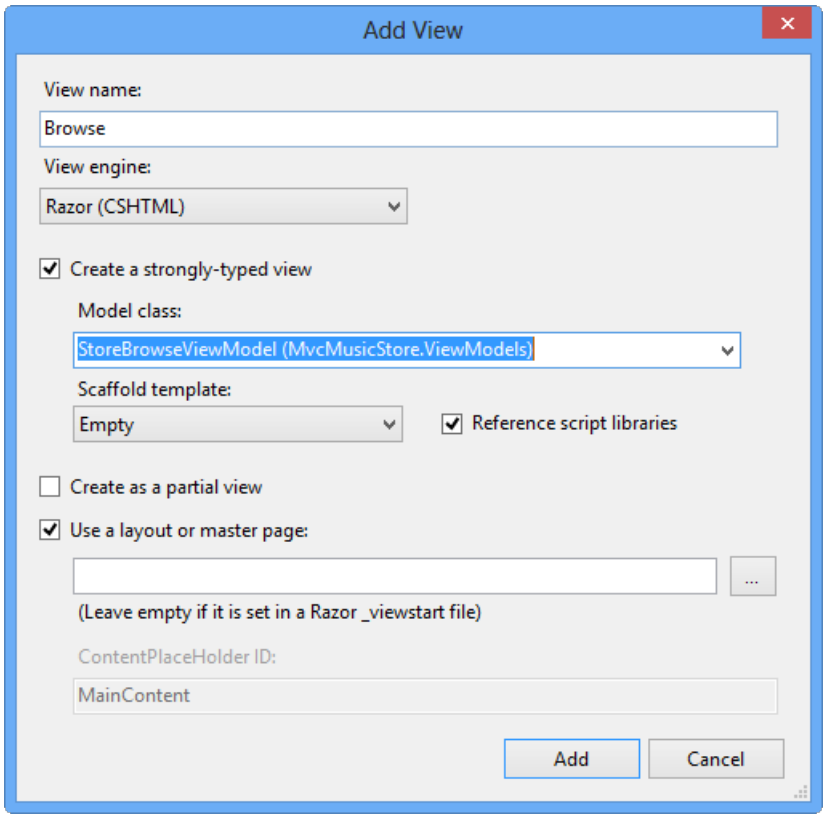
Task 4 - Adding a Browse View Template

In this task, you will add a **Browse** View to show the Albums found for a specific Genre.

1. Before creating the new View template, you should build the project so that the **Add View** Dialog knows about the **ViewModel** class to use. Build the project by selecting the **Build** menu item and then **Build MvcMusicStore**.

2. Add a **Browse** View. To do this, right-click in the **Browse** action method of the **StoreController** and click **Add View**.

3. In the **Add View** dialog box, verify that the View Name is **Browse**. Check the **Create a strongly-typed view** checkbox and select **StoreBrowseViewModel** from the **Model class** dropdown. Leave the other fields with their default value. Then click **Add**.



Adding a Browse View

4. Modify the **Browse.cshtml** to display the Genre's information, accessing the **StoreBrowseViewModel** object that is passed to the view template. To do this, replace the content with the following: (C#)

```
@model MvcMusicStore.ViewModels.StoreBrowseViewModel

@{
    ViewBag.Title = "Browse Albums";
}

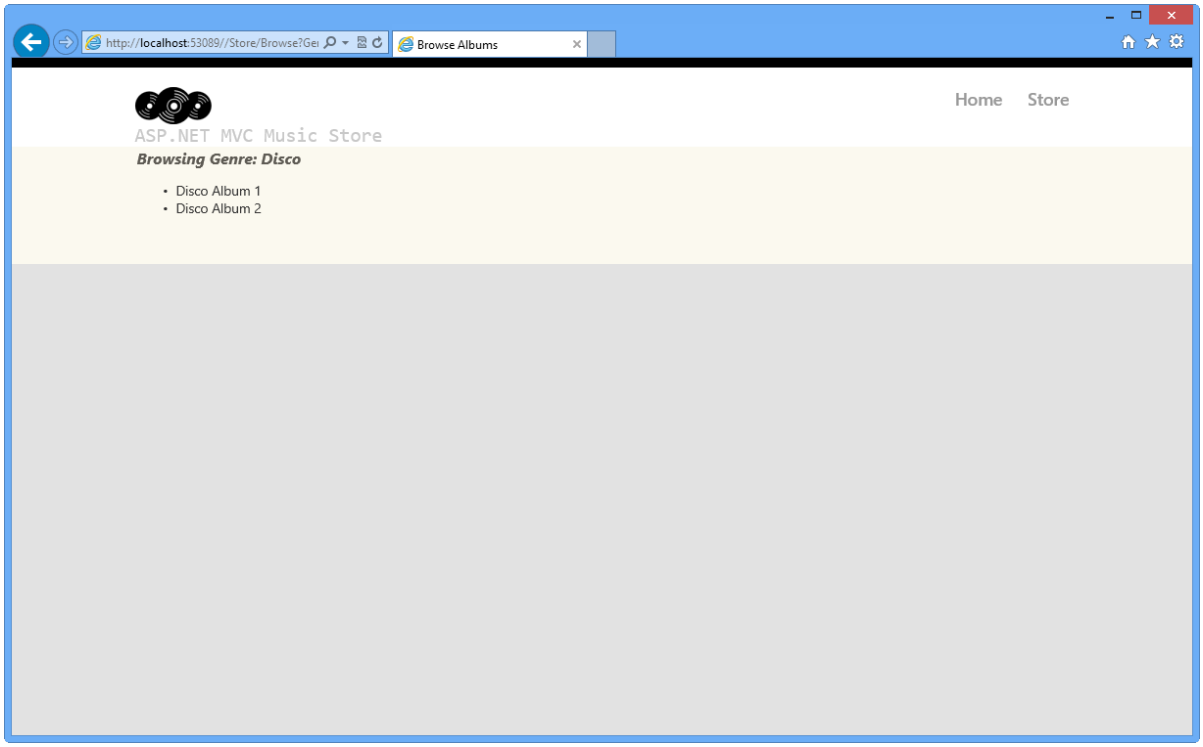
<h2>Browsing Genre: @Model.Genre.Name</h2>

<ul>
    @foreach (var album in Model.Albums)
    {
        <li>@album.Title</li>
    }
</ul>
```

Task 5 - Running the Application

In this task, you will test that the **Browse** method retrieves Albums from the **Browse** method action.

1. Press **F5** to run the Application.
2. The project starts in the Home page. Change the URL to **/Store/Browse?Genre=Disco** to verify that the action returns two Albums.

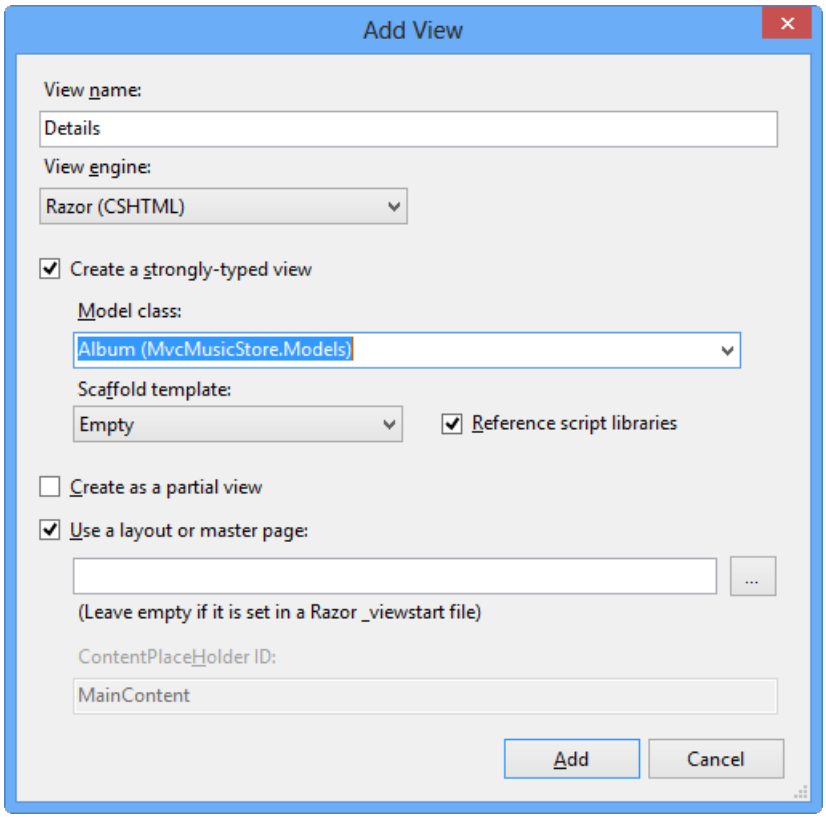


Browsing Store Disco Albums

Task 6 - Displaying information About a Specific Album

In this task, you will implement the **Store/Details** view to display information about a specific album. In this Hands-on Lab, everything you will display about the album is already contained in the **View** template. So, instead of creating a **StoreDetailsViewModel** class, you will use the current **StoreBrowseViewModel** template passing the Album to it.

1. Close the browser if needed, to return to the Visual Studio window. Add a new **Details** view for the **StoreController**'s **Details** action method. To do this, right-click the **Details** method in the **StoreController** class and click **Add View**.
2. In the **Add View** dialog, verify that the **View Name** is **Details**. Check the **Create a strongly-typed view** checkbox and select **Album** from the **Model class** drop-down. Leave the other fields with their default value. Then click **Add**. This will create and open a **\Views\Store\Details.cshtml** file.



Adding a Details View

3. Modify the **Details.cshtml** file to display the Album's information, accessing the **Album** object that is passed to the view template. To do this, replace the content with the following: (C#)

```
@model MvcMusicStore.Models.Album

@{
    ViewBag.Title = "Details";
}

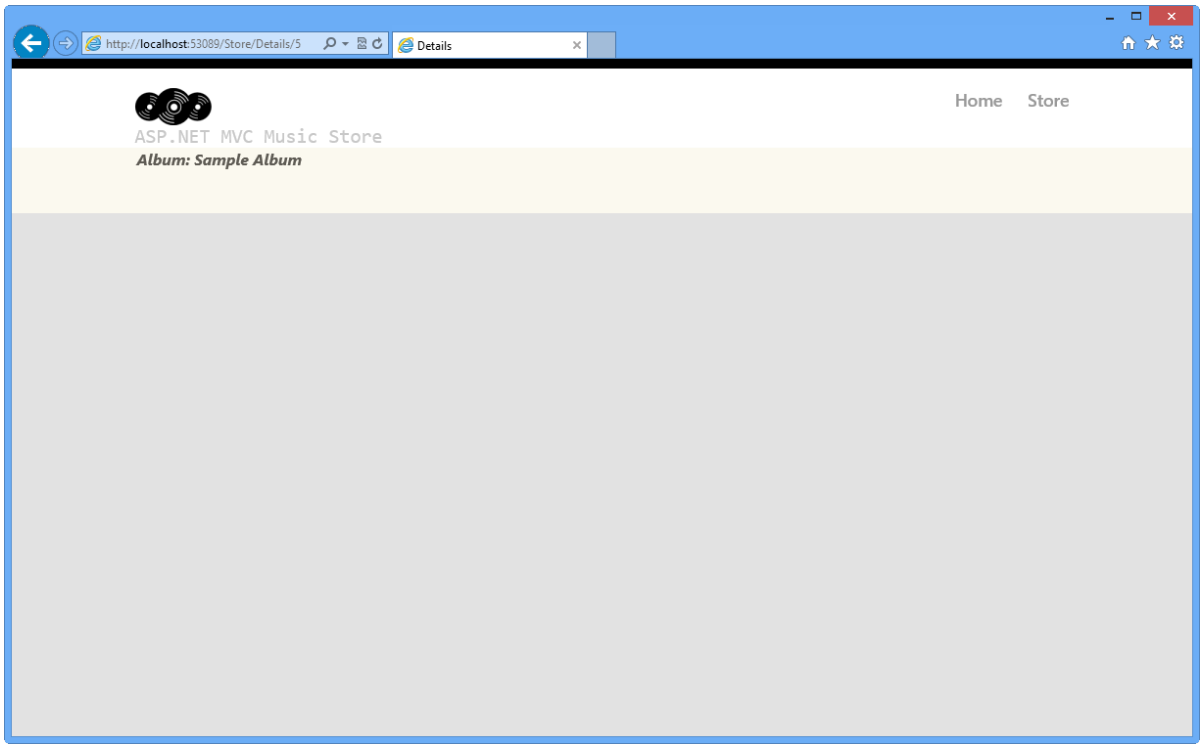
<h2>Album: @Model.Title</h2>
```

Task 7 - Running the Application

In this task, you will test that the **Details** View retrieves Album's information from the **Details action** method.

1. Press **F5** to run the Application.

2. The project starts in the **Home** page. Change the URL to **/Store/Details/5** to verify the album's information.



Browsing Album's Detail

Task 8 - Adding Links Between Pages

In this task, you will add a link in the Store View to have a link in every Genre name to the appropriate **/Store/Browse** URL. This way, when you click on a Genre, for instance **Disco**, it will navigate to **/Store/Browse?genre=Disco** URL.

1. Close the browser if needed, to return to the Visual Studio window. Update the **Index** page to add a link to the **Browse** page. To do this, in the **Solution Explorer** expand the **Views** folder, then the **Store** folder and double-click the **Index.cshtml** page.

2. Add a link to the Browse view indicating the genre selected. To do this, replace the following highlighted code within the **<li>** tags: (C#)

```
@model MvcMusicStore.ViewModels.StoreIndexViewModel

@{
    ViewBag.Title = "Browse Genres";
}

<h2>Browse Genres</h2>

<p> Select from @Model.NumberOfGenres genres</p>

<ul>
    @foreach (string genreName in Model.Genres) {
        <li>
            @Html.ActionLink(genreName, "Browse", new { genre = genreName }, null)
        </li>
        }
</ul>
```

NOTE

another approach would be linking directly to the page, with a code like the following:

@genreName

Although this approach works, it depends on a hardcoded string. If you later rename the Controller, you will have to change this instruction manually. A better alternative is to use an **HTML Helper** method. ASP.NET MVC includes an HTML Helper method which is available for such tasks. The **Html.ActionLink()** helper method makes it easy to build HTML **<a>** links, making sure URL paths are properly URL encoded.

Html.ActionLink has several overloads. In this exercise you will use one that takes three parameters:

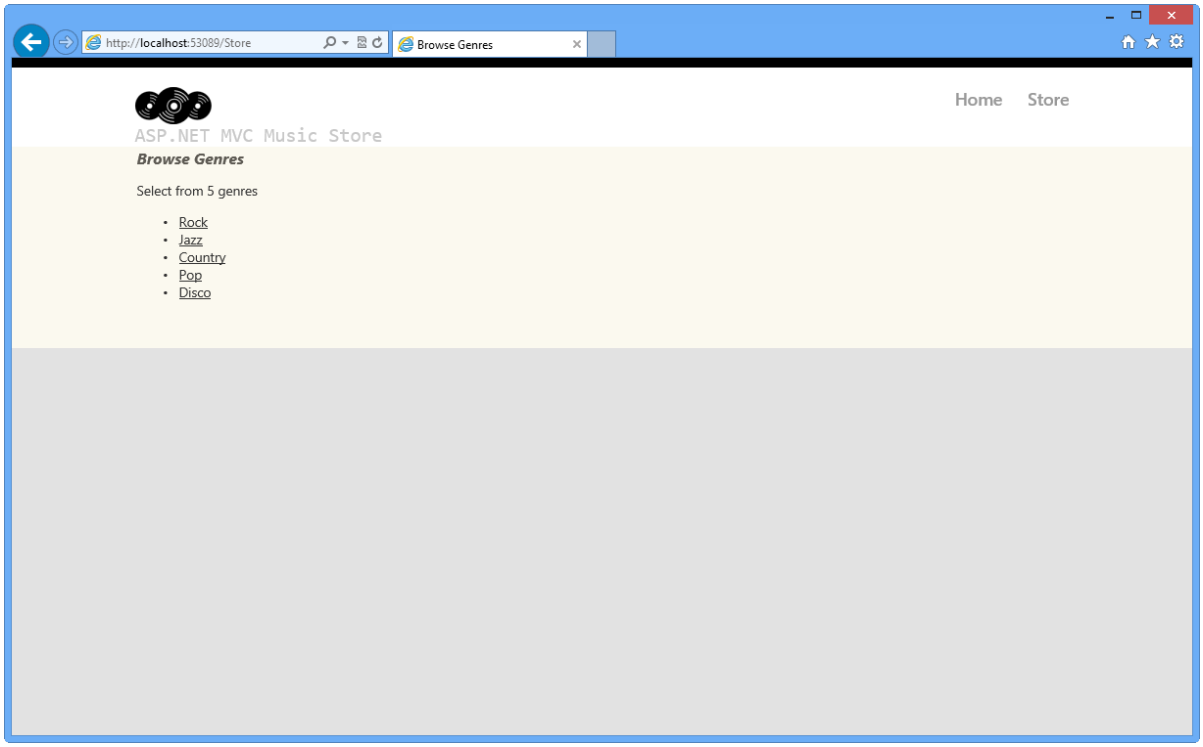
1. Link text, which will display the Genre name
2. Controller action name (**Browse**)
3. Route parameter values, specifying both the name (**Genre**) and the value (**Genre name**)

Task 9 - Running the Application

In this task, you will test that each Genre is displayed with a link to the appropriate **/Store/Browse** URL.

1. Press **F5** to run the Application.

2. The project starts in the Home page. Change the URL to **/Store** to verify that each Genre links to the appropriate **/Store/Browse** URL.



Browsing Genres with links to Browse page

Task 10 - Using Dynamic ViewModel Collection to Pass Values

In this task, you will learn a simple and powerful method to pass values between the Controller and the View without making any changes in the Model. ASP.NET MVC 4 provides the collection "ViewModel", which can be assigned to any dynamic value and accessed inside controllers and views as well.

You will now use the ViewBag dynamic collection to pass a list of "**Starred genres**" from the controller to the view. The Store Index view will access to **ViewModel** and display the information.

1. Close the browser if needed, to return to the Visual Studio window. Open **StoreController.cs** and modify **Index** method to create a list of starred genres into ViewModel collection :

```
public ActionResult Index()
{
    // Create list of genres
    var genres = new List<string> { "Rock", "Jazz", "Country", "Pop", "Disco" };

    // Create your view model
    var viewModel = new StoreIndexViewModel
    {
        NumberOfGenres = genres.Count,
        Genres = genres
    };

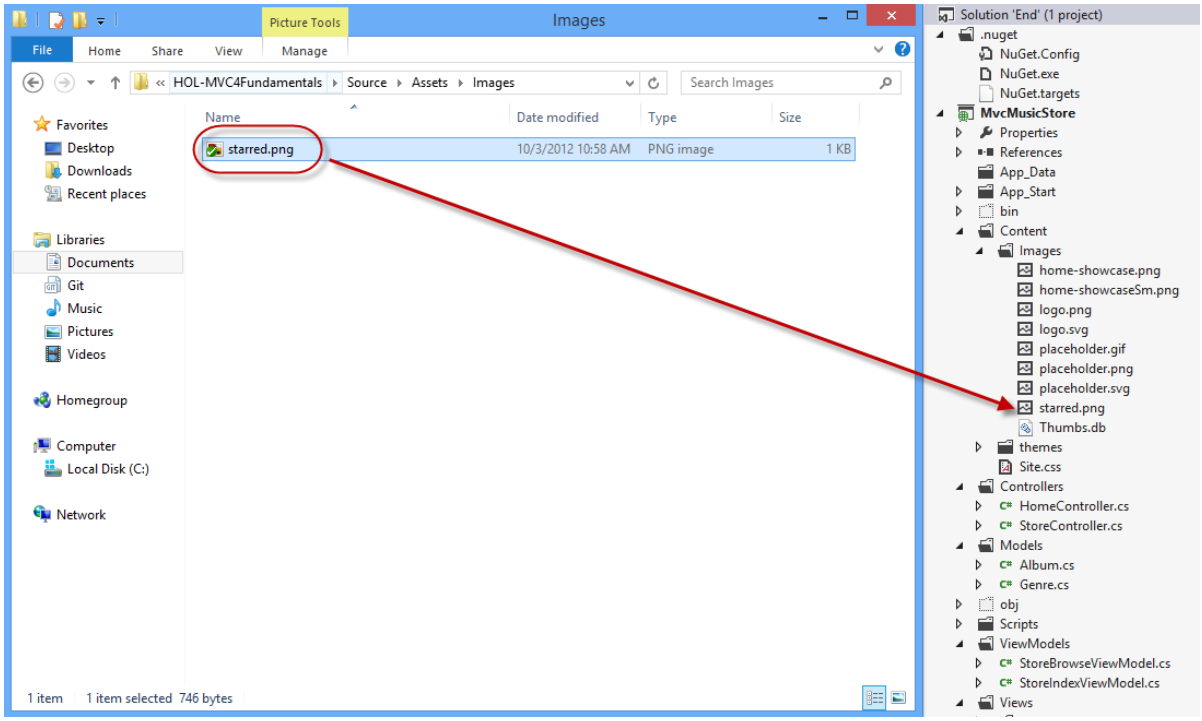
    ViewBag.Starred = new List<string> { "Rock", "Jazz" };

    return this.View(viewModel);
}
```

NOTE

You could also use the syntax **ViewBag["Starred"]** to access the properties.

2. The star icon **"starred.png"** is included in the **Source\Assets\Images** folder of this lab. In order to add it to the application, drag their content from a **Windows Explorer** window into the **Solution Explorer** in Visual Web Developer Express, as shown below:



Adding star image to the solution

3. Open the view **Store/Index.cshtml** and modify the content. You will read the "starred" property in the **ViewBag** collection, and ask if the current genre name is in the list. In that case you will show a star icon right to the genre link. (C#)

```
@model MvcMusicStore.ViewModels.StoreIndexViewModel

@{
     ViewBag.Title = "Browse Genres";
}

<h2>Browse Genres</h2>

<p>Select from @Model.NumberOfGenres genres</p>

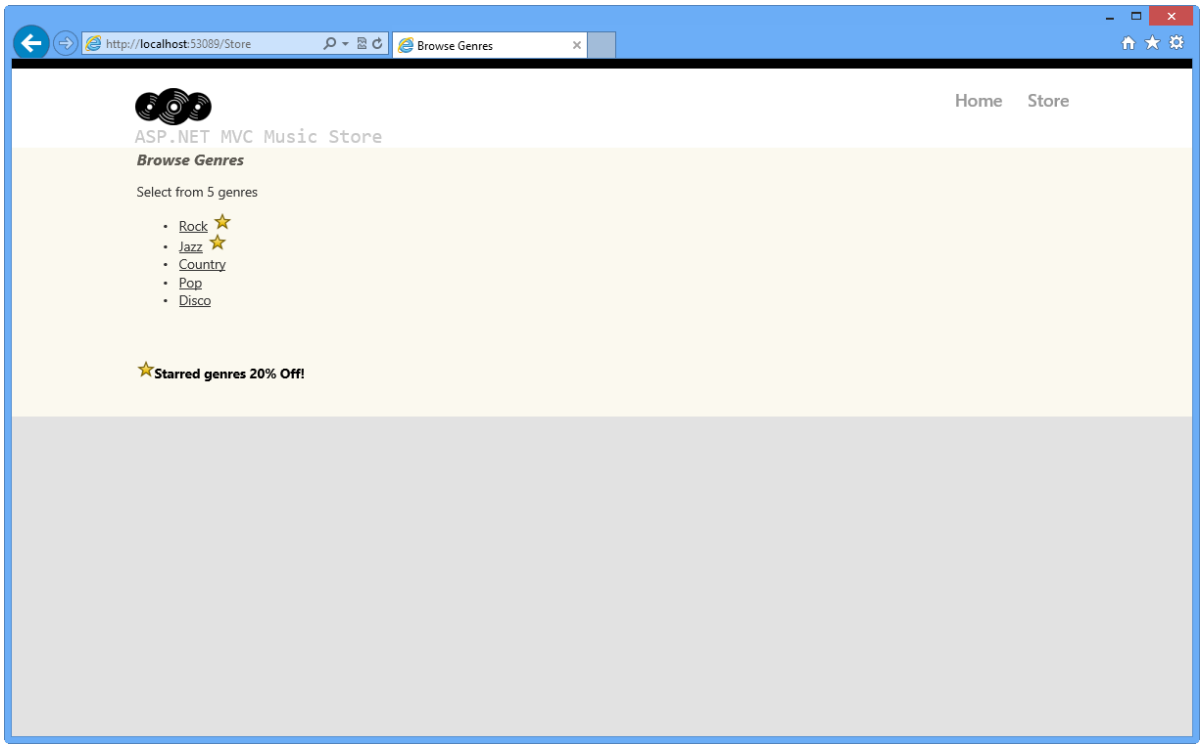
<ul>
     @foreach (string genreName in Model.Genres)
     {
          <li>
                @Html.ActionLink(genreName, "Browse", new { genre = genreName }, null)

                @if (ViewBag.Starred.Contains(genreName))
                {
                     <img src="../../Content/Images/starred.png" alt="Starred element" />
                }
          </li>
     }
</ul>
<br />
<h5><img src="../../Content/Images/starred.png" alt="Starred element" />Starred genres 20% Off!</h5>
```

Task 11 - Running the Application

In this task, you will test that the starred genres display a star icon.

1. Press **F5** to run the Application.
2. The project starts in the **Home** page. Change the URL to **/Store** to verify that each featured genre has the respecting label:



Browsing Genres with starred elements

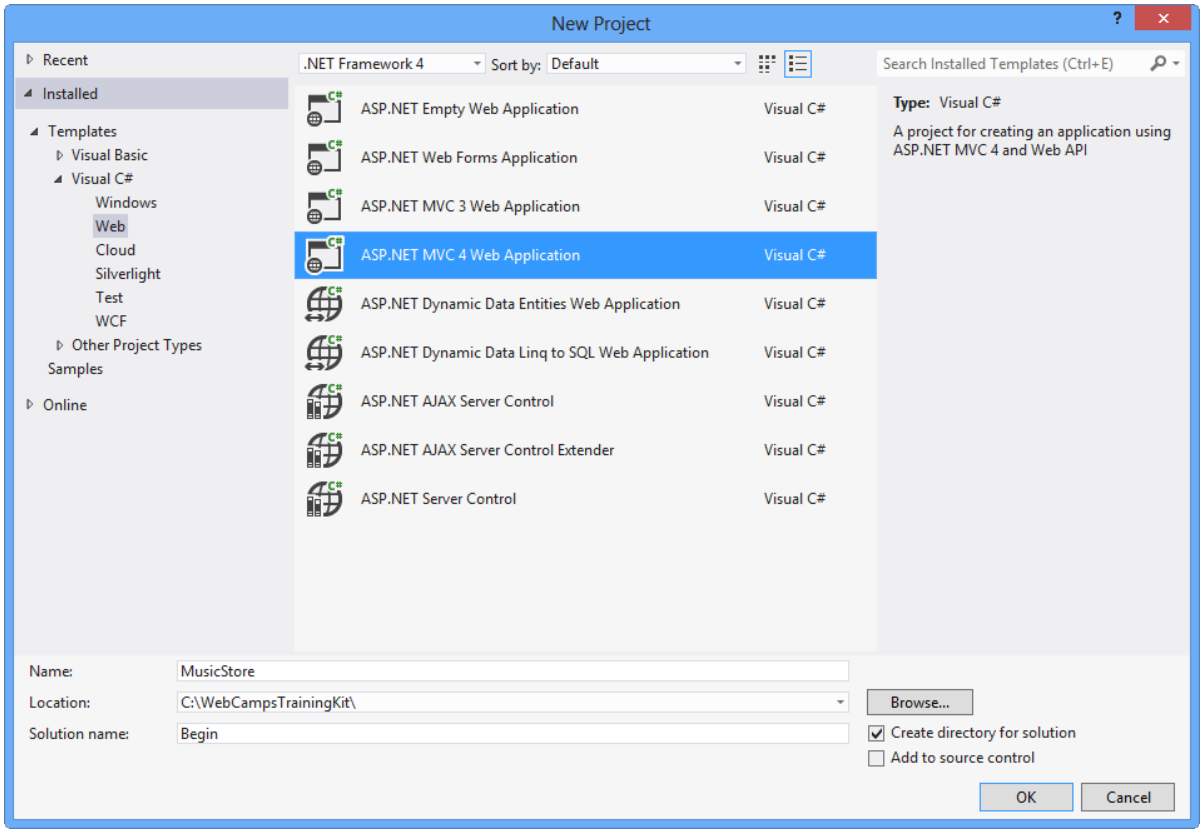
Exercise 7: A lap around ASP.NET MVC 4 new template

In this exercise, you will explore the enhancements in the ASP.NET MVC 4 project templates, taking a look at the most relevant features of the new template.

Task 1: Exploring the ASP.NET MVC 4 Internet Application Template

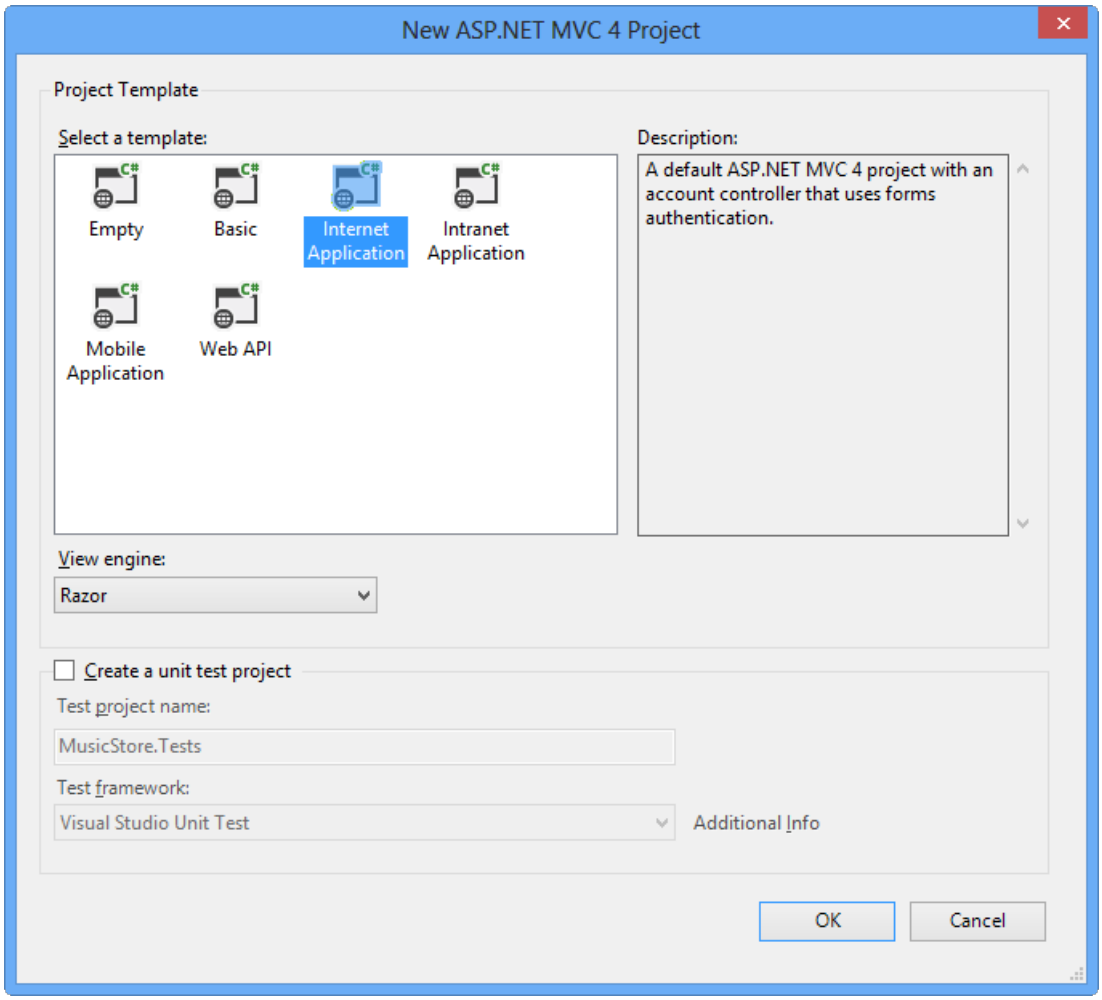
1. If not already open, start **VS Express for Web**

2. Select the **File | New | Project** menu command. In the **New Project** dialog, select the **Visual C#|Web** template on the left pane tree, and choose the **ASP.NET MVC 4 Web Application**. **Name** the project *MusicStore* and update the **solution name** to *Begin*, then select a location (or leave the default) and click **OK**.



Creating a new ASP.NET MVC 4 Project

3. In the **New ASP.NET MVC 4 Project** dialog, select the **Internet Application** project template and click **OK**. Notice you can select either Razor or ASPX as the view engine.



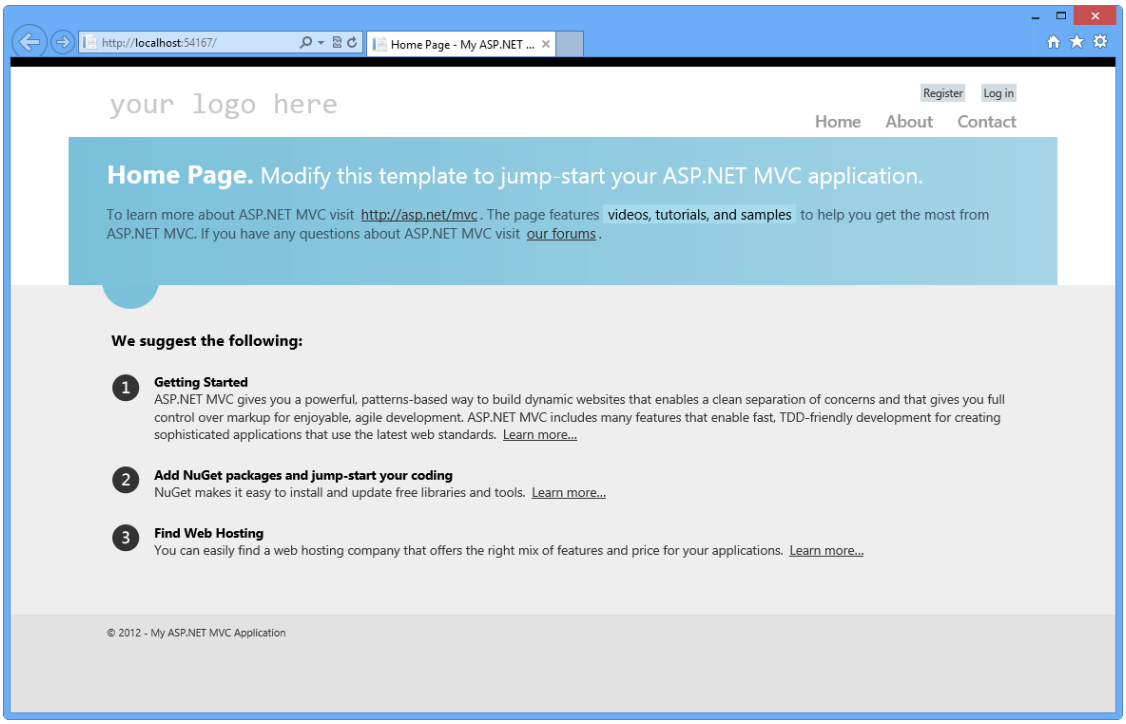
Creating a new ASP.NET MVC 4 Internet Application

> **NOTE**
>
> Razor syntax has been introduced in ASP.NET MVC 3. Its goal is to minimize the number of characters and keystrokes required in a file, enabling a fast and fluid coding workflow. Razor leverages existing C#/VB (or other) language skills and delivers a template markup syntax that enables an awesome HTML construction workflow.

4. Press **F5** to run the solution and see the renewed template. You can check out the following features:

   a. **Modern-style templates**

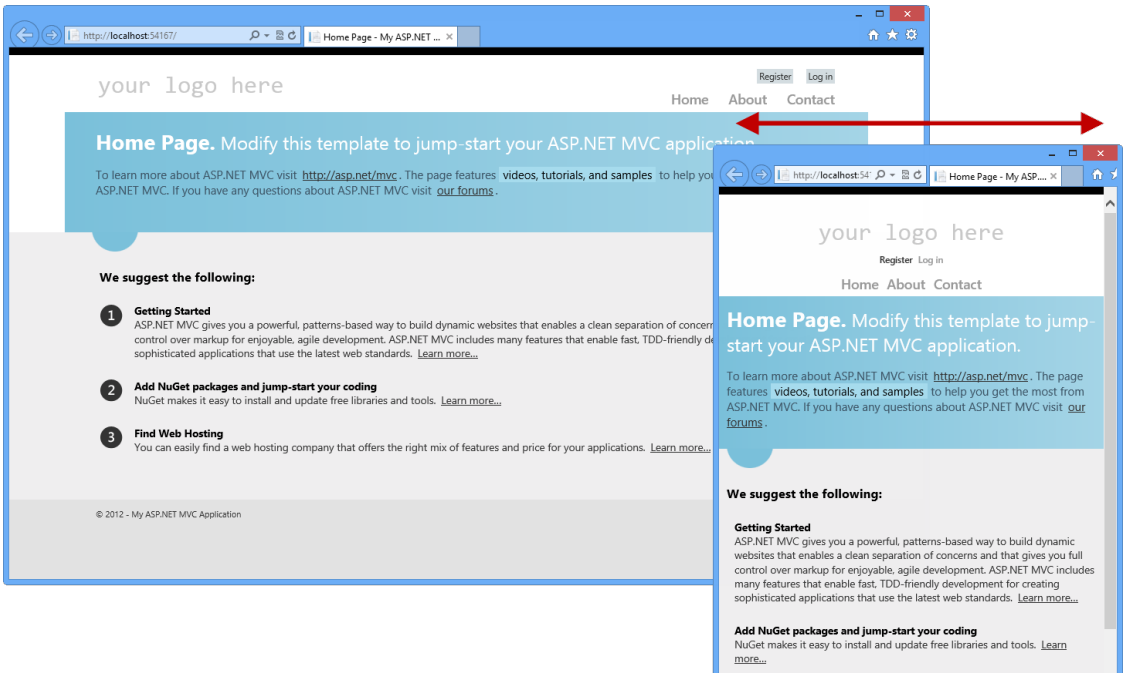
      The templates have been renewed, providing more modern-looking styles.



ASP.NET MVC 4 restyled templates

b. **Adaptive Rendering**

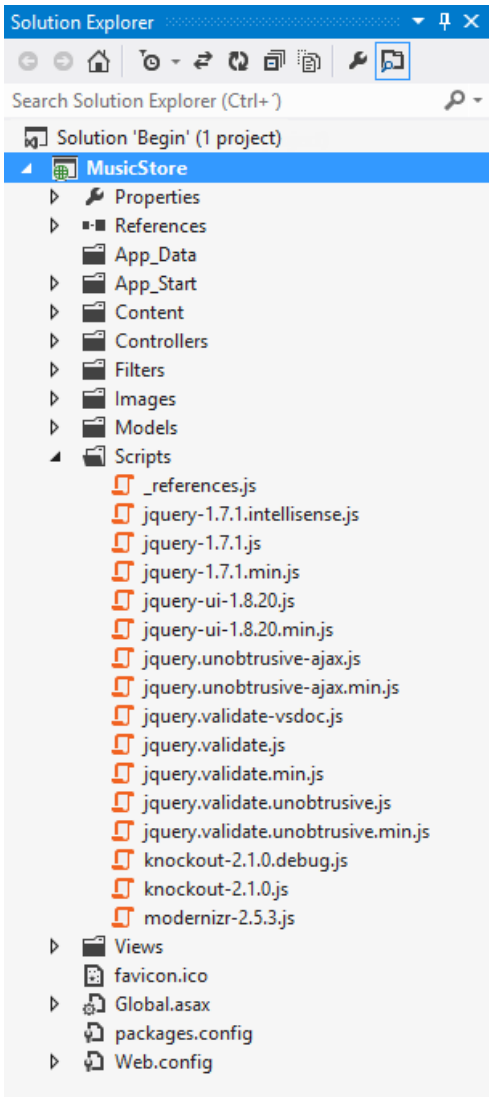
Check out resizing the browser window and notice how the page layout dynamically adapts to the new window size. These templates use the adaptive rendering technique to render properly in both desktop and mobile platforms without any customization.



ASP.NET MVC 4 project template in different browser sizes

5. Close the browser to stop the debugger and return to Visual Studio.

6. Now you are able to explore the solution and check out some of the new features introduced by ASP.NET MVC 4 in the project template.



The ASP.NET MVC 4 Internet Application Project Template

a. **HTML5 markup**

Browse template views to find out the new theme markup, for example open **About.cshtml** view within **Home** folder.

```
    <p>
        Use this area to provide additional information.
    </p>
</article>

<aside>
    <h3>Aside Title</h3>
    <p>
        Use this area to provide additional information.
    </p>
    <ul>
        <li>@Html.ActionLink("Home", "Index", "Home")</li>
        <li>@Html.ActionLink("About", "About", "Home")</li>
        <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
    </ul>
</aside>
```

New template, using Razor and HTML5 markup

b. **JavaScript libraries included**

a. **jQuery**: jQuery simplifies HTML document traversing, event handling, animating, and Ajax interactions.

b. **jQuery UI**: This library provides abstractions for low-level interaction and animation,

advanced effects and themeable widgets, built on top of the jQuery JavaScript Library.

> **NOTE**
>
> You can learn about jQuery and jQuery UI in http://docs.jquery.com/.

c. **KnockoutJS**: The ASP.NET MVC 4 default template now includes **KnockoutJS**, a JavaScript MVVM framework that lets you create rich and highly responsive web applications using JavaScript and HTML. Like in ASP.NET MVC 3, jQuery and jQuery UI libraries are also included in ASP.NET MVC 4.

> **NOTE**
>
> You can get more information about KnockOutJS library in this link: http://learn.knockoutjs.com/.

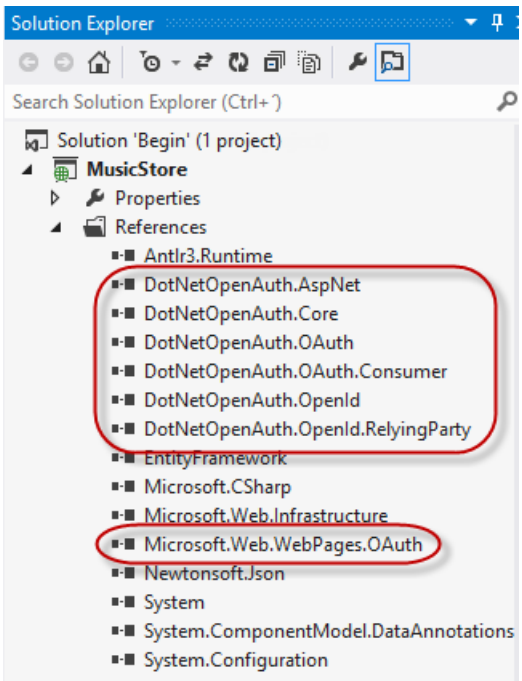
d. **Modernizr**: This library runs automatically, making your site compatible with older browsers when using HTML5 and CSS3 technologies.

> **NOTE**
>
> You can get more information about Modernizr library in this link: http://www.modernizr.com/.

c. **SimpleMembership included in the solution**

SimpleMembership has been designed as a replacement for the previous ASP.NET Role and Membership provider system. It has many new features that make it easier for the developer to secure web pages in a more flexible way.

The Internet template already has set up a few things to integrate SimpleMembership, for example, the AccountController is prepared to use OAuthWebSecurity (for OAuth account registration, login, management, etc.) and Web Security.



SimpleMembership Included in the solution

> **NOTE**
> Find more information about OAuthWebSecurity in MSDN.

> **NOTE**
> Additionally, you can deploy this application to Windows Azure Web Sites following Appendix B: Publishing an ASP.NET MVC 4 Application using Web Deploy.

Summary

By completing this Hands-On Lab you have learned the fundamentals of ASP.NET MVC:

- The core elements of an MVC application and how they interact
- How to create an ASP.NET MVC Application
- How to add and configure Controllers to handle parameters passed through the URL and querystring
- How to add a layout master page to setup a template for common HTML content, a StyleSheet to enhance the look and feel and a View template to display HTML content
- How to use the ViewModel pattern for passing properties to the View template to display dynamic information
- How to use parameters passed to Controllers in the View template
- How to add links to pages inside the ASP.NET MVC application
- How to add and use dynamic properties in a View
- The enhancements in the ASP.NET MVC 4 project templates

Appendix A: Installing Visual Studio Express 2012 for Web

You can install **Microsoft Visual Studio Express 2012 for Web** or another "Express" version using the **Microsoft Web Platform Installer**. The following instructions guide you through the steps required to install *Visual studio Express 2012 for Web* using *Microsoft Web Platform Installer*.

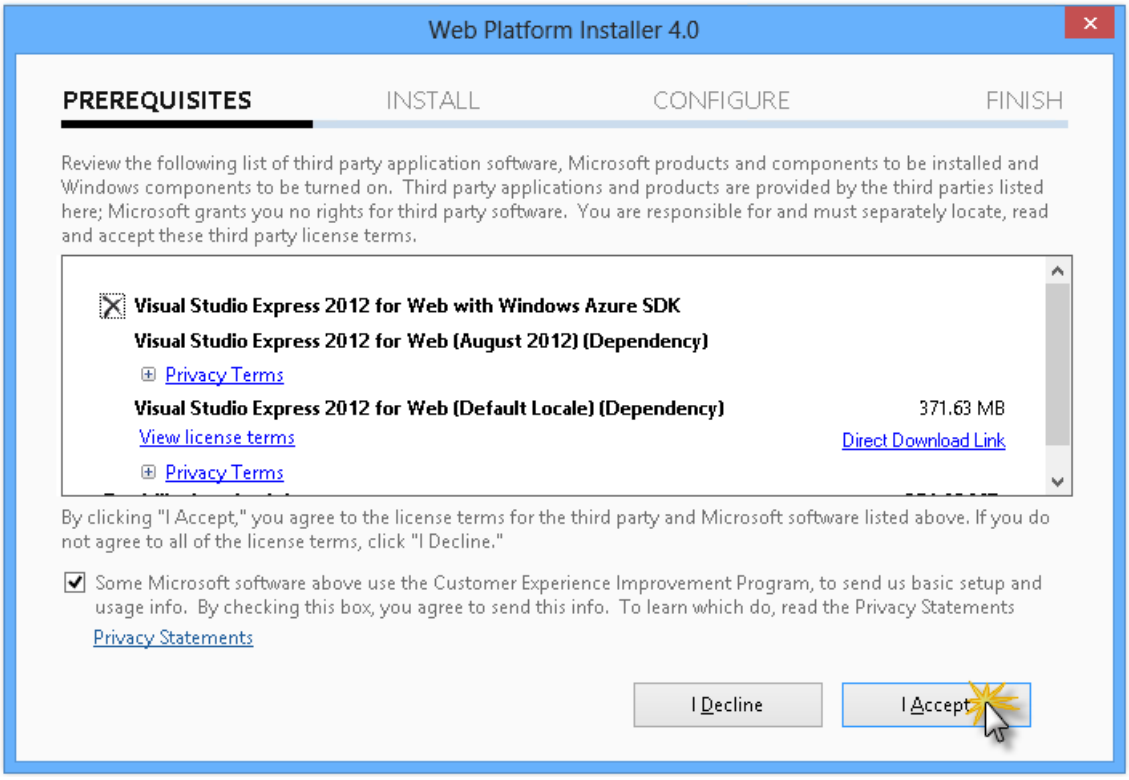
1. Go to https://go.microsoft.com/?linkid=9810169. Alternatively, if you already have installed Web Platform Installer, you can open it and search for the product "*Visual Studio Express 2012 for Web with Windows Azure SDK*".

2. Click on **Install Now**. If you do not have **Web Platform Installer** you will be redirected to download and install it first.

3. Once **Web Platform Installer** is open, click **Install** to start the setup.

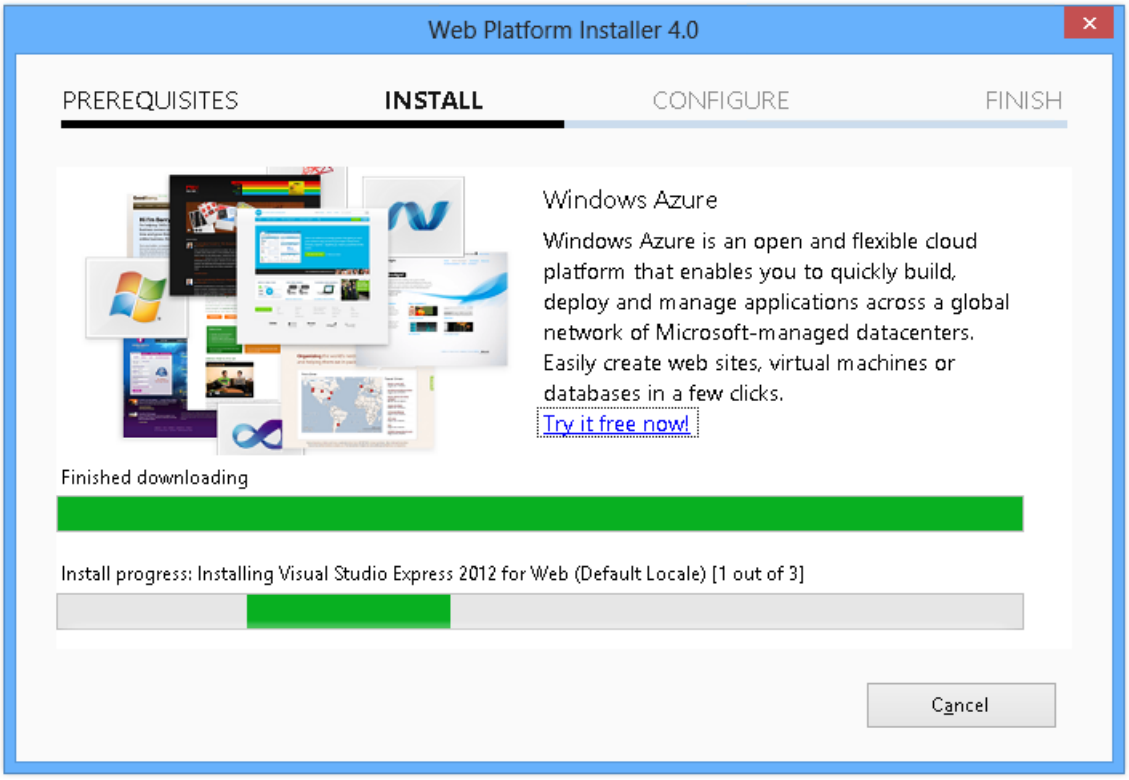
Install Visual Studio Express

4. Read all the products' licenses and terms and click **I Accept** to continue.



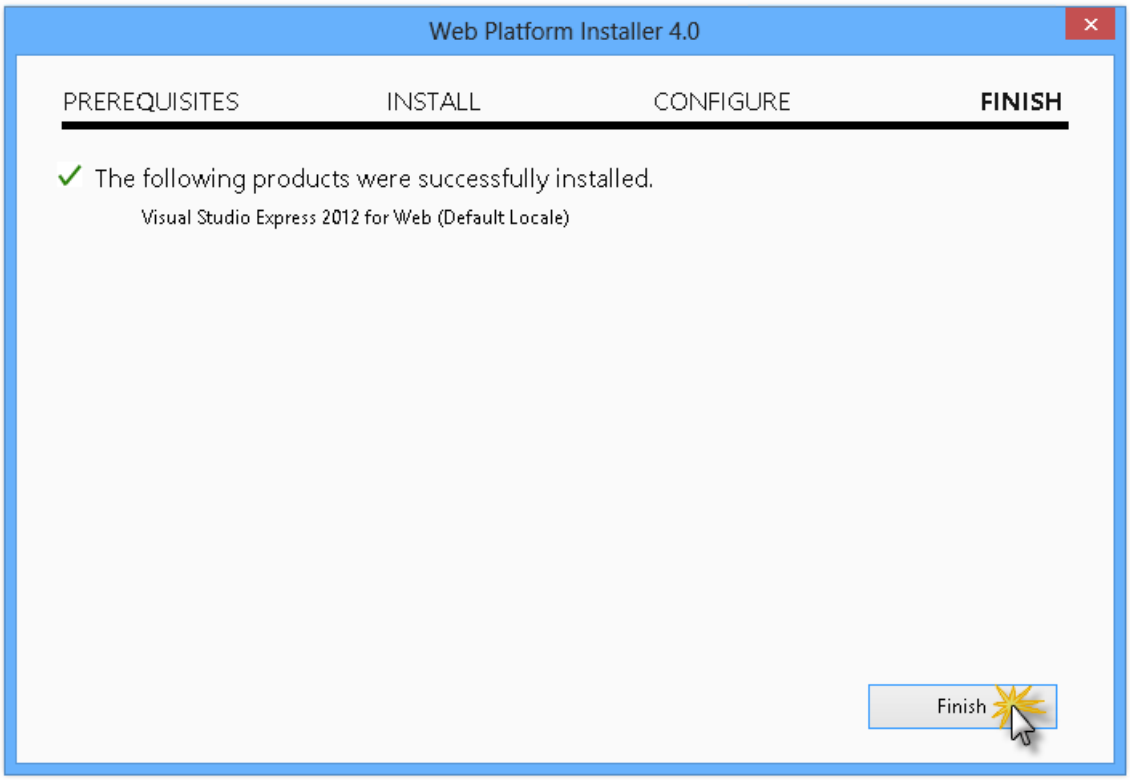
Accepting the license terms

5. Wait until the downloading and installation process completes.



Installation progress

6. When the installation completes, click **Finish**.



Installation completed

7. Click **Exit** to close Web Platform Installer.

8. To open Visual Studio Express for Web, go to the **Start** screen and start writing "**VS Express**", then click on the **VS Express for Web** tile.

VS Express for Web tile

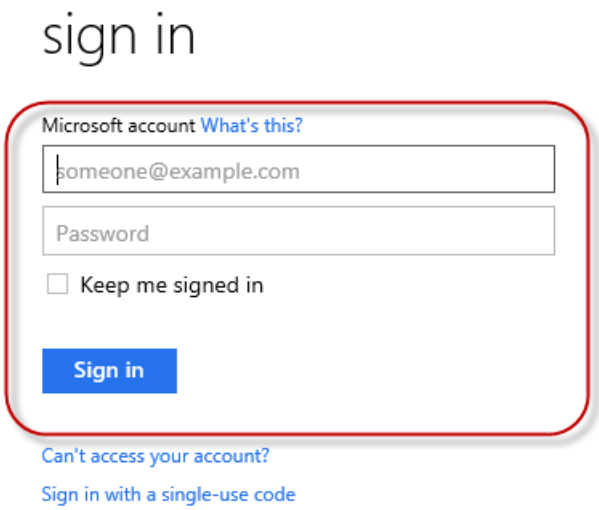
Appendix B: Publishing an ASP.NET MVC 4 Application using Web Deploy

This appendix will show you how to create a new web site from the Windows Azure Management Portal and publish the application you obtained by following the lab, taking advantage of the Web Deploy publishing feature provided by Windows Azure.

Task 1 - Creating a New Web Site from the Windows Azure Portal

1. Go to the Windows Azure Management Portal and sign in using the Microsoft credentials associated with your subscription.

> **NOTE**
> With Windows Azure you can host 10 ASP.NET Web Sites for free and then scale as your traffic grows. You can sign up here.



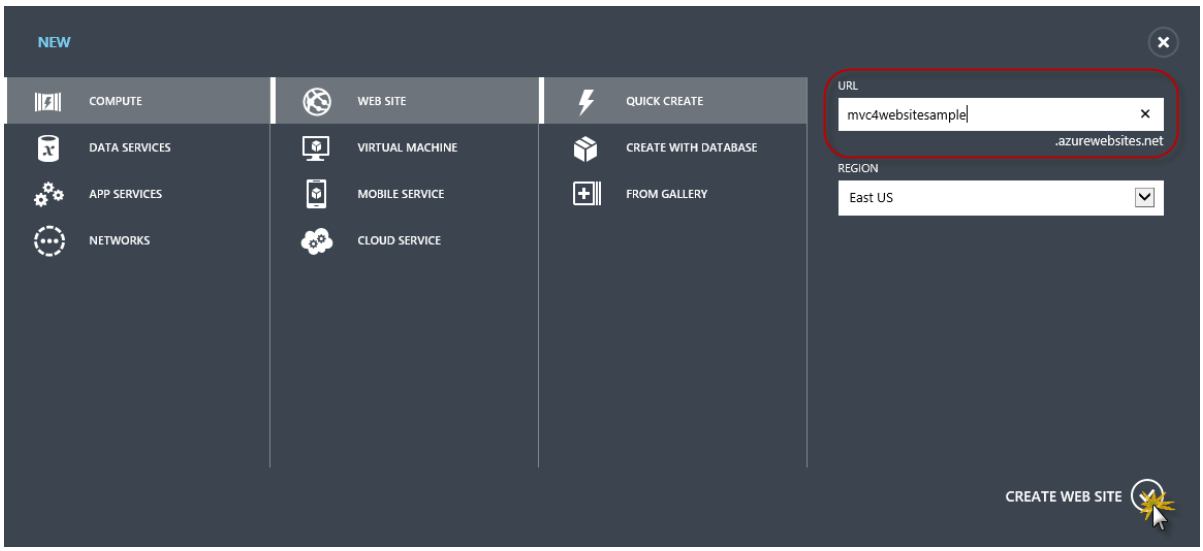
Log on to Windows Azure Management Portal

2. Click **New** on the command bar.

Creating a new Web Site

3. Click **Compute | Web Site**. Then select **Quick Create** option. Provide an available URL for the new web site and click **Create Web Site**.

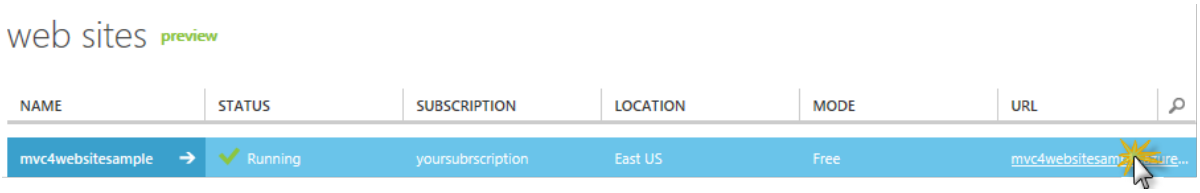
> **NOTE**
>
> A Windows Azure Web Site is the host for a web application running in the cloud that you can control and manage. The Quick Create option allows you to deploy a completed web application to the Windows Azure Web Site from outside the portal. It does not include steps for setting up a database.



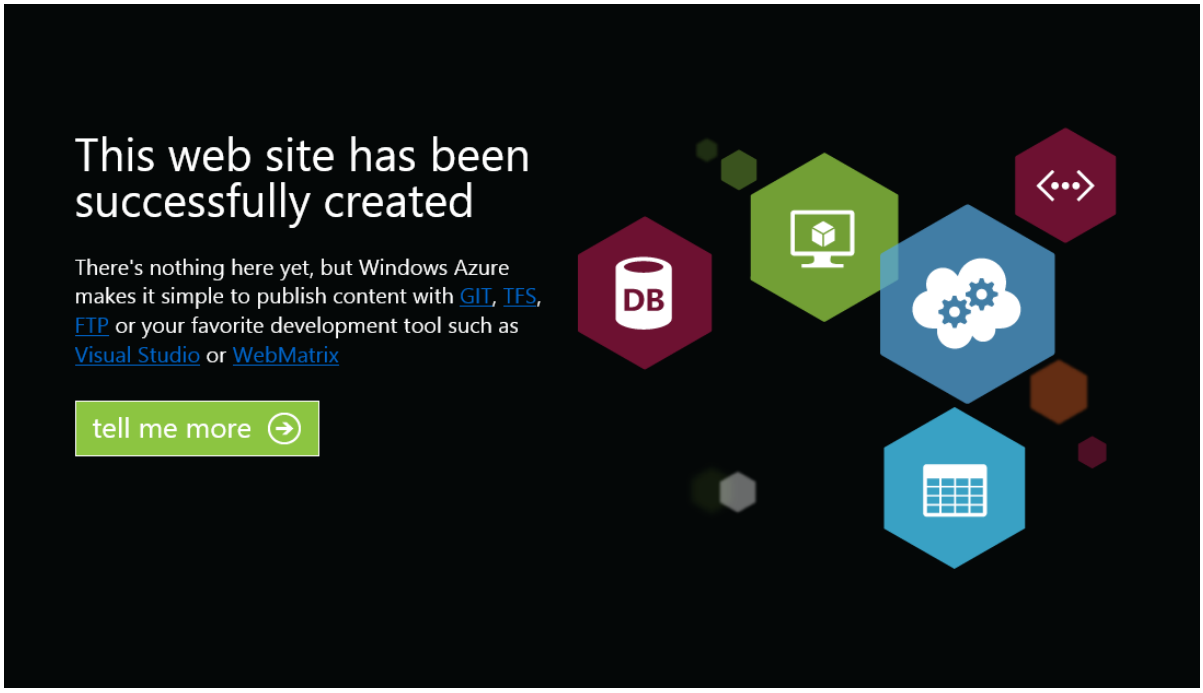
Creating a new Web Site using Quick Create

4. Wait until the new **Web Site** is created.

5. Once the Web Site is created click the link under the **URL** column. Check that the new Web Site is working.

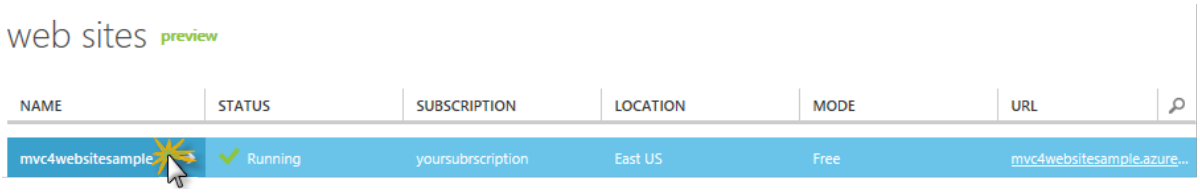


Browsing to the new web site



Web site running

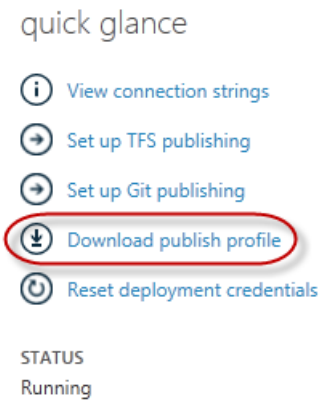
6. Go back to the portal and click the name of the web site under the **Name** column to display the management pages.



Opening the Web Site management pages

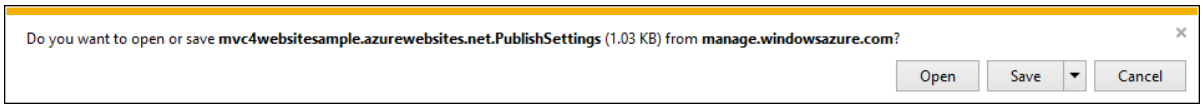
7. In the **Dashboard** page, under the **quick glance** section, click the **Download publish profile** link.

> **NOTE**
>
> The *publish profile* contains all of the information required to publish a web application to a Windows Azure website for each enabled publication method. The publish profile contains the URLs, user credentials and database strings required to connect to and authenticate against each of the endpoints for which a publication method is enabled. **Microsoft WebMatrix 2**, **Microsoft Visual Studio Express for Web** and **Microsoft Visual Studio 2012** support reading publish profiles to automate configuration of these programs for publishing web applications to Windows Azure websites.



Downloading the Web Site publish profile

8. Download the publish profile file to a known location. Further in this exercise you will see how to use this file to publish a web application to a Windows Azure Web Sites from Visual Studio.

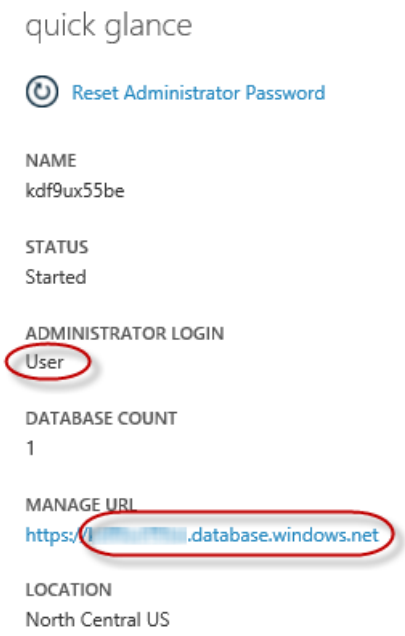


Saving the publish profile file

Task 2 - Configuring the Database Server

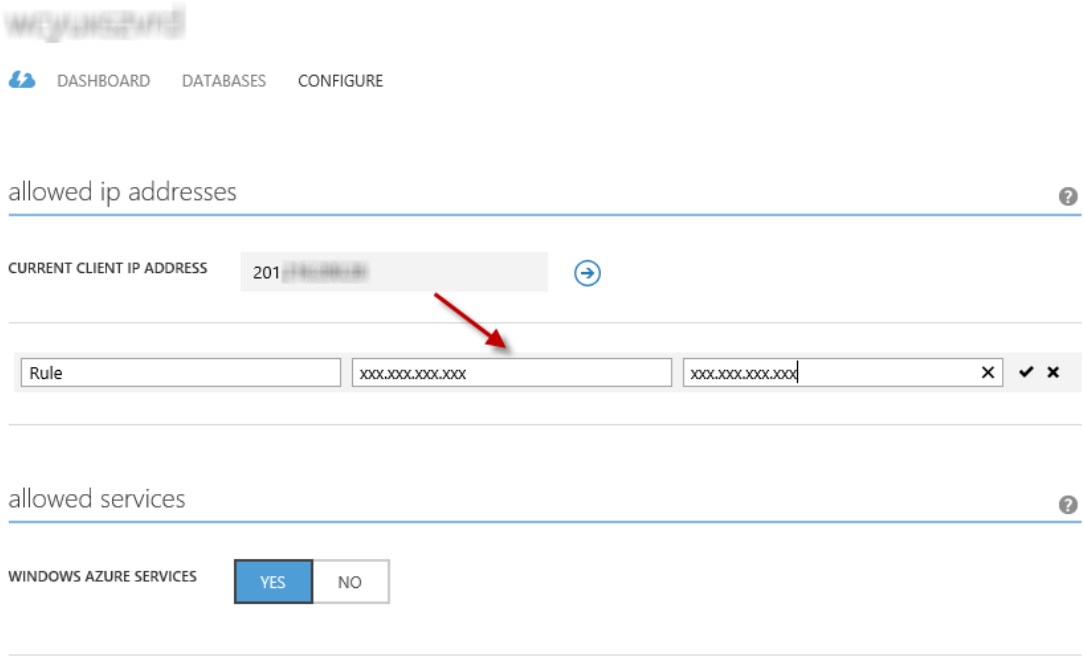
If your application makes use of SQL Server databases you will need to create a SQL Database server. If you want to deploy a simple application that does not use SQL Server you might skip this task.

1. You will need a SQL Database server for storing the application database. You can view the SQL Database servers from your subscription in the Windows Azure Management portal at **Sql Databases | Servers | Server's Dashboard**. If you do not have a server created, you can create one using the **Add** button on the command bar. Take note of the **server name and URL, administrator login name and password**, as you will use them in the next tasks. Do not create the database yet, as it will be created in a later stage.



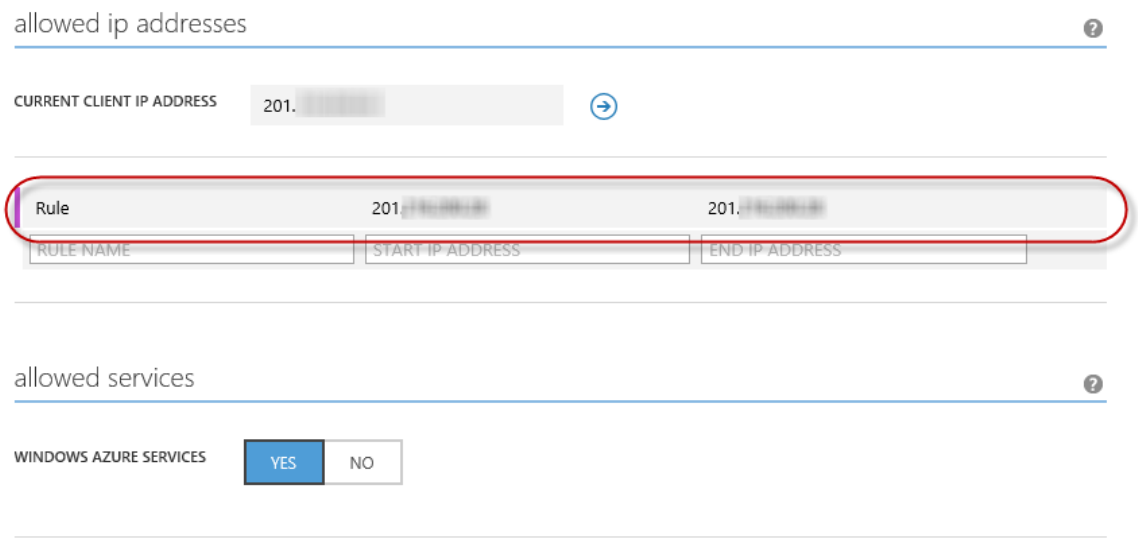
SQL Database Server Dashboard

2. In the next task you will test the database connection from Visual Studio, for that reason you need to include your local IP address in the server's list of **Allowed IP Addresses**. To do that, click **Configure**, select the IP address from **Current Client IP Address** and paste it on the **Start IP Address** and **End IP Address** text boxes and click the ✔ button.



Adding Client IP Address

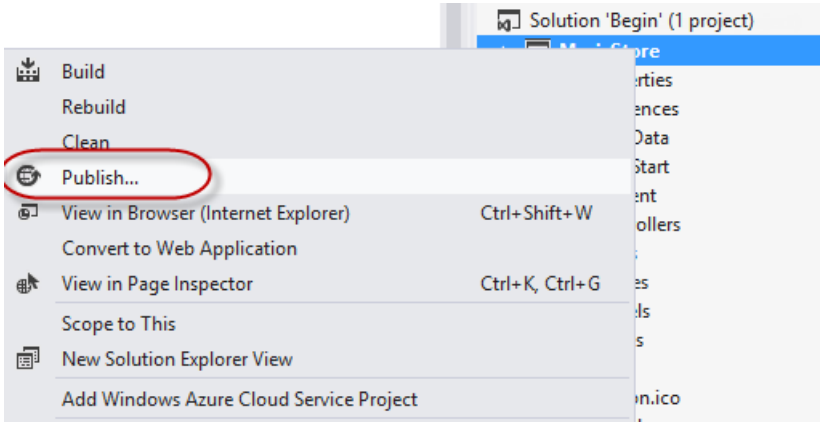
3. Once the **Client IP Address** is added to the allowed IP addresses list, click on **Save** to confirm the changes.



Confirm Changes

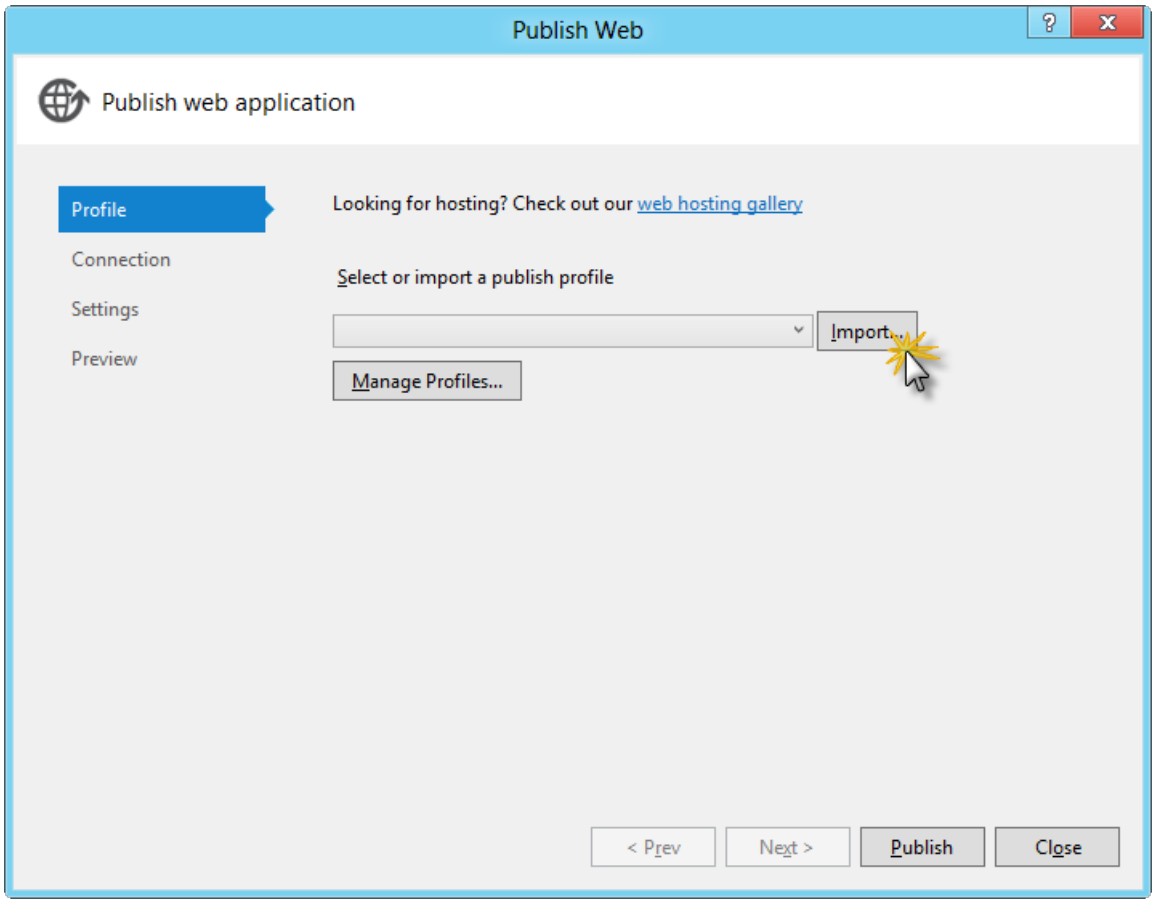
Task 3 - Publishing an ASP.NET MVC 4 Application using Web Deploy

1. Go back to the ASP.NET MVC 4 solution. In the **Solution Explorer**, right-click the web site project and select **Publish**.



Publishing the web site

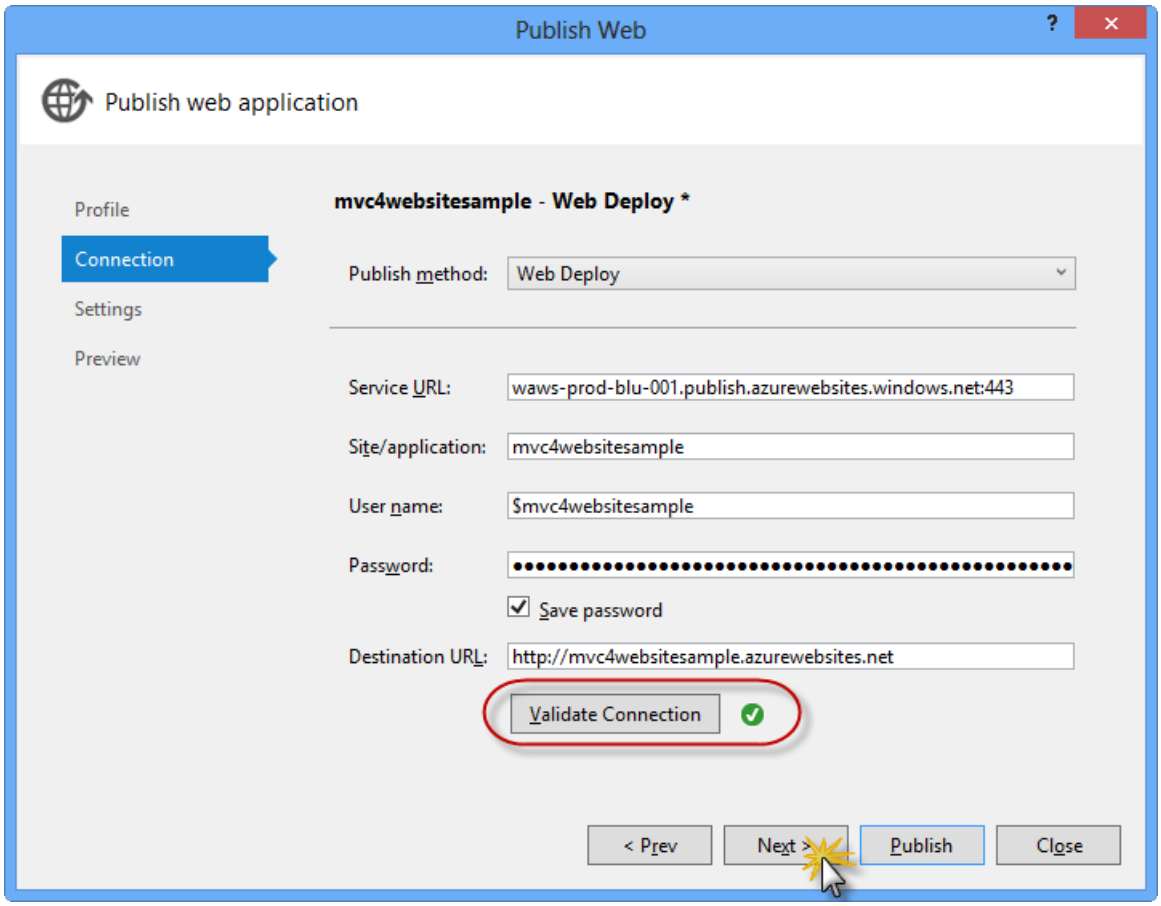
2. Import the publish profile you saved in the first task.



Importing publish profile

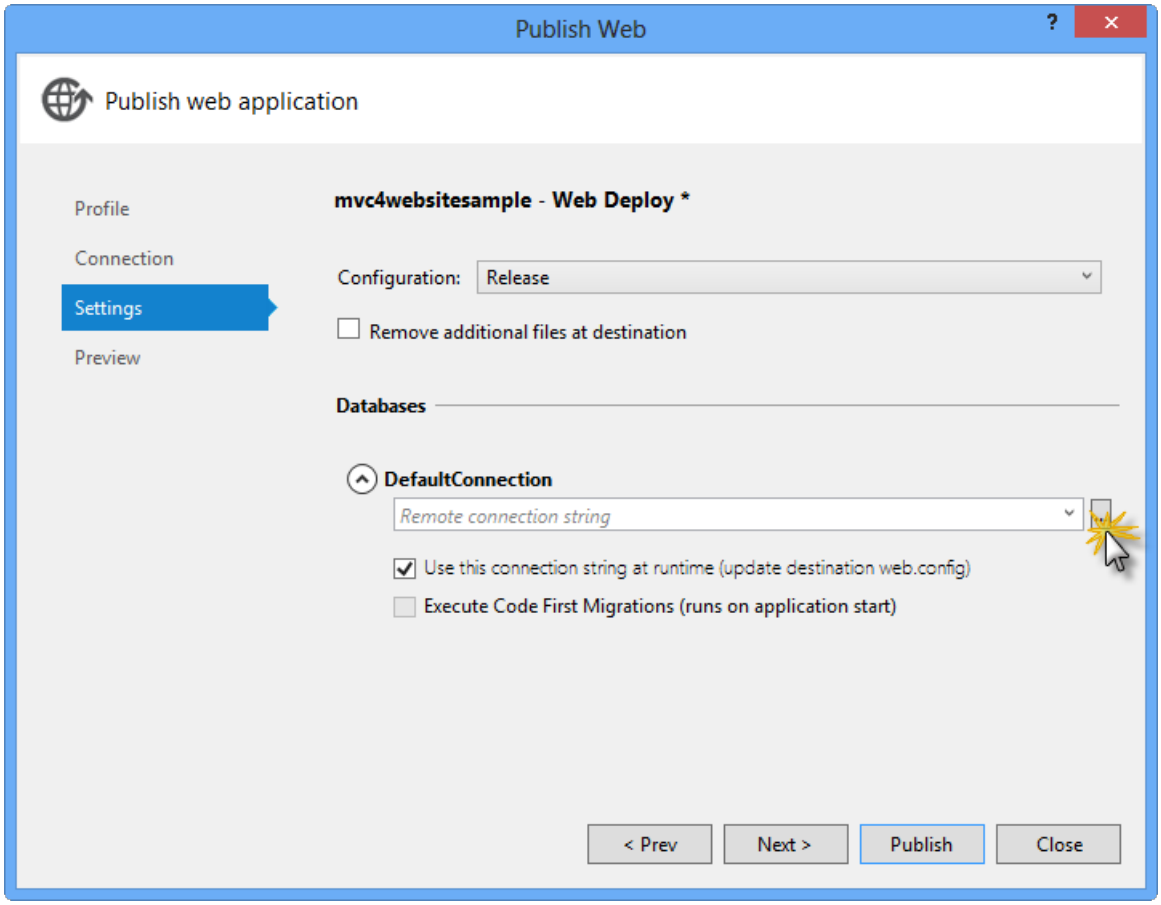
3. Click **Validate Connection**. Once Validation is complete click **Next**.

> **NOTE**
>
> Validation is complete once you see a green checkmark appear next to the Validate Connection button.



Validating connection

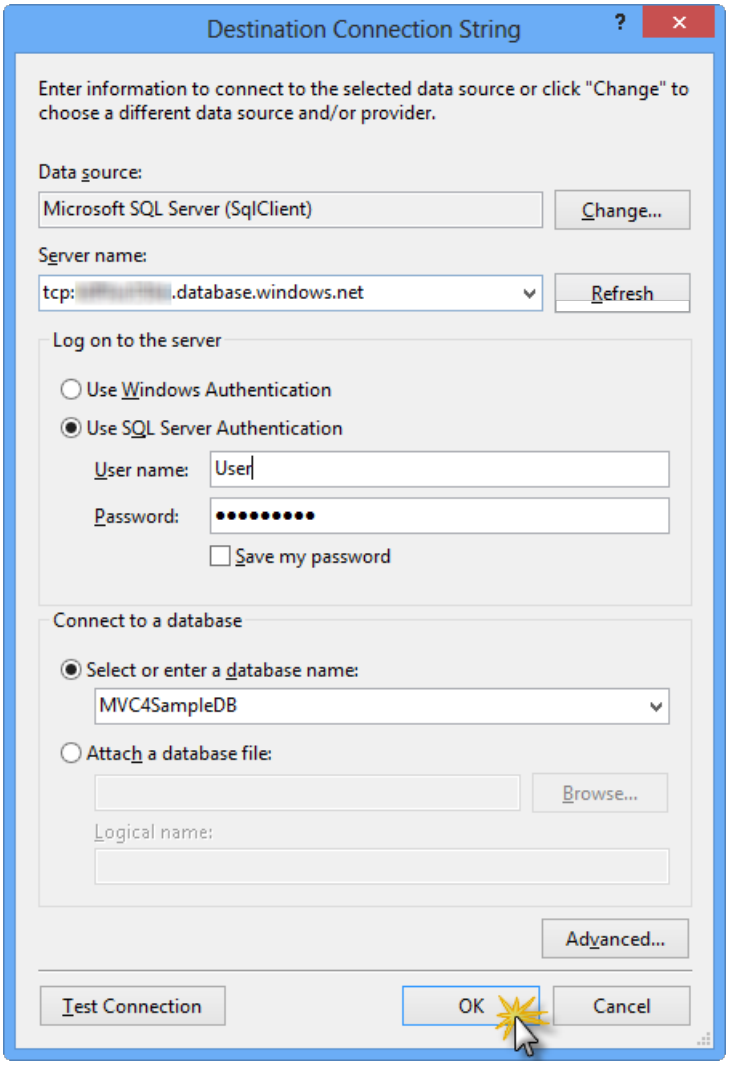
4. In the **Settings** page, under the **Databases** section, click the button next to your database connection's textbox (i.e. **DefaultConnection**).



Web deploy configuration

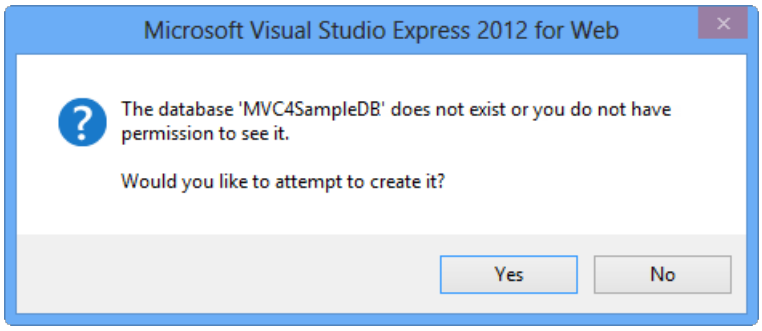
5. Configure the database connection as follows:

   - In the **Server name** type your SQL Database server URL using the *tcp:* prefix.
   - In **User name** type your server administrator login name.
   - In **Password** type your server administrator login password.
   - Type a new database name, for example: *MVC4SampleDB*.



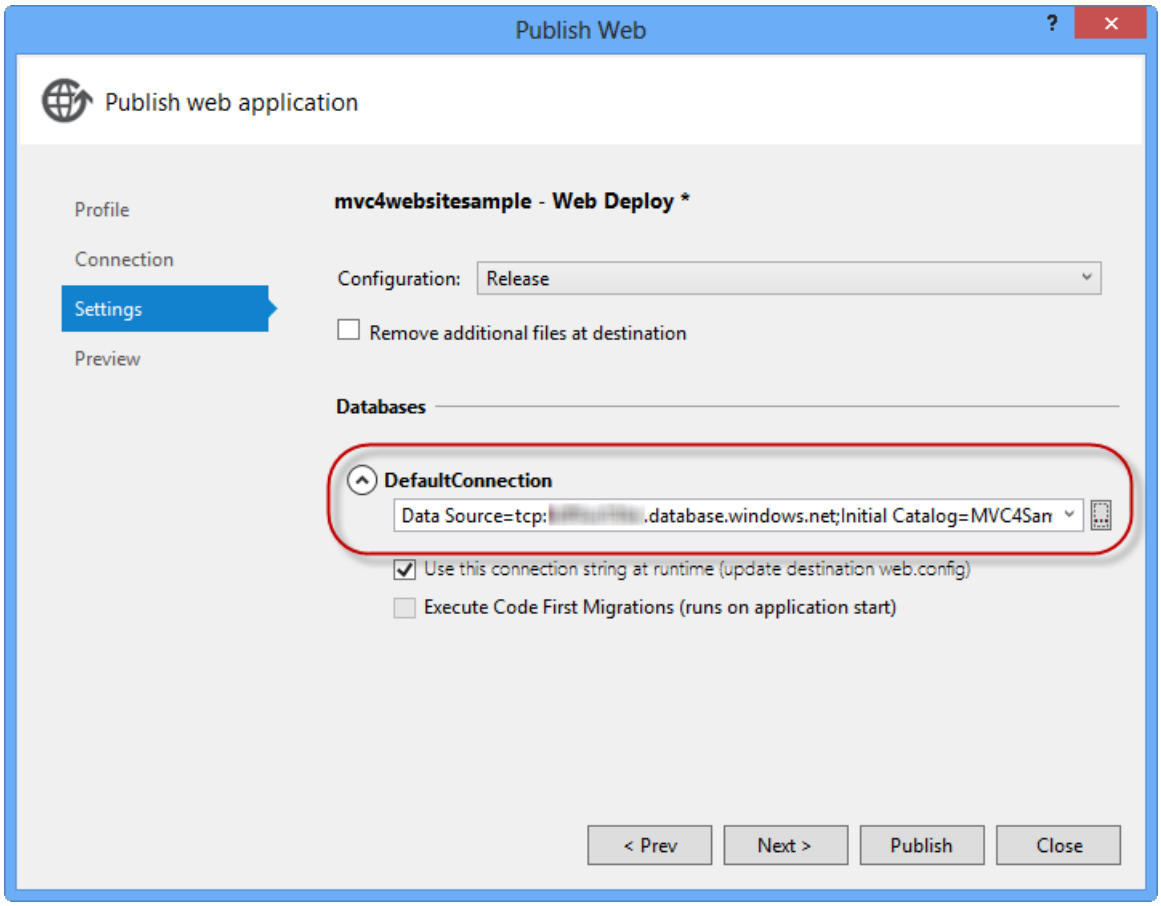
Configuring destination connection string

6. Then click **OK**. When prompted to create the database click **Yes**.



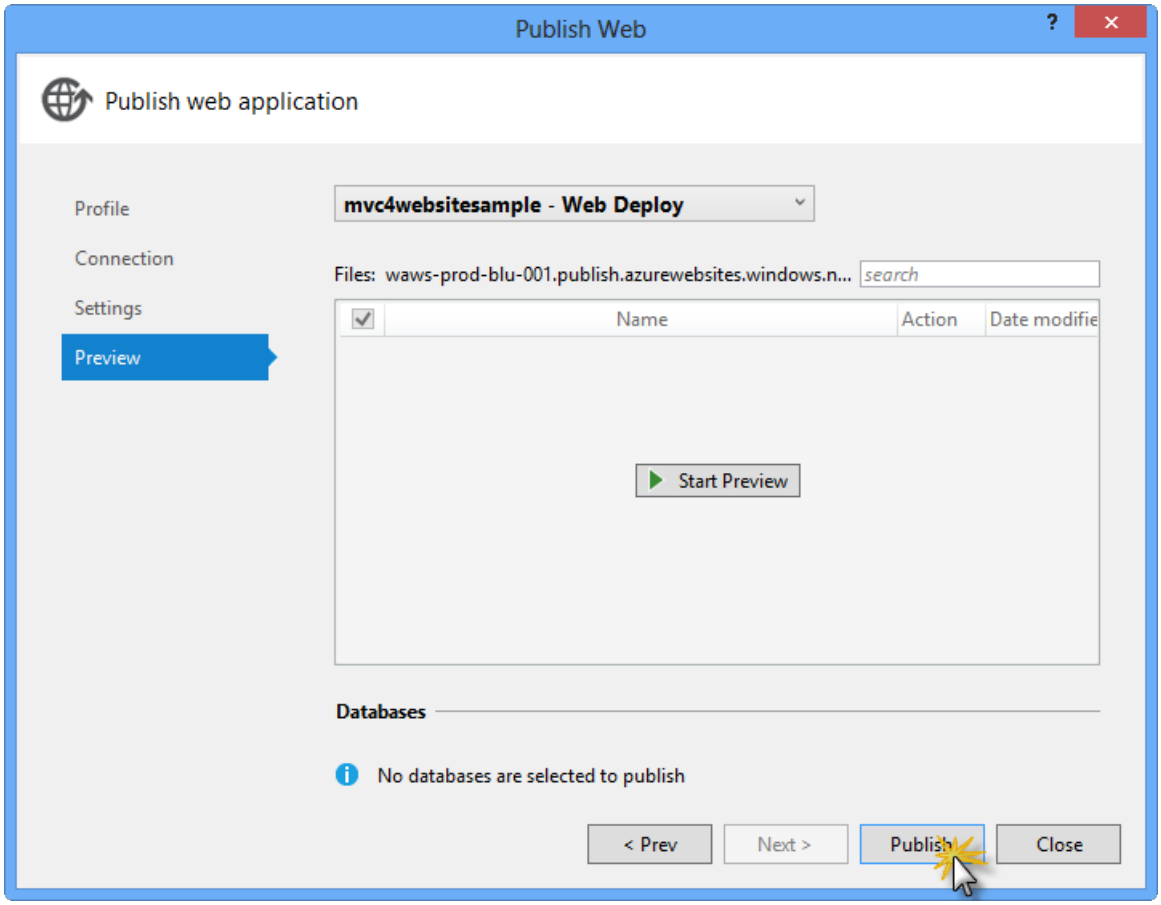
Creating the database

7. The connection string you will use to connect to SQL Database in Windows Azure is shown within Default Connection textbox. Then click **Next**.



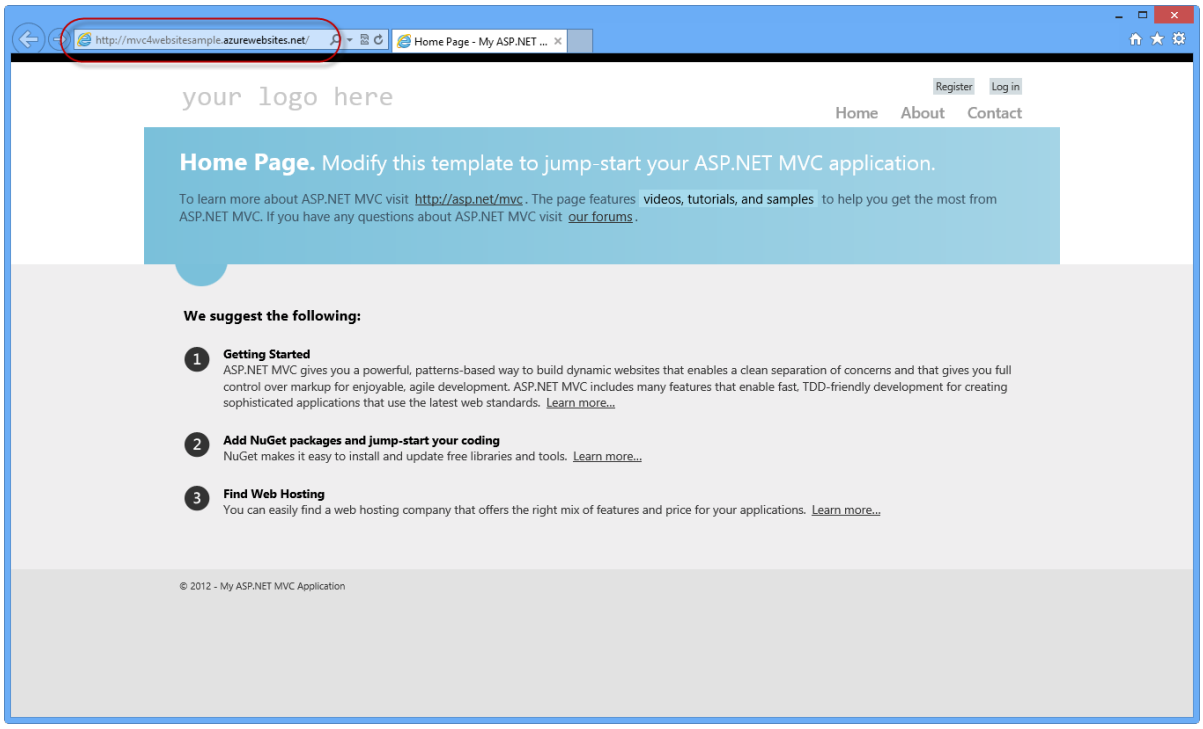
Connection string pointing to SQL Database

8. In the **Preview** page, click **Publish**.



Publishing the web application

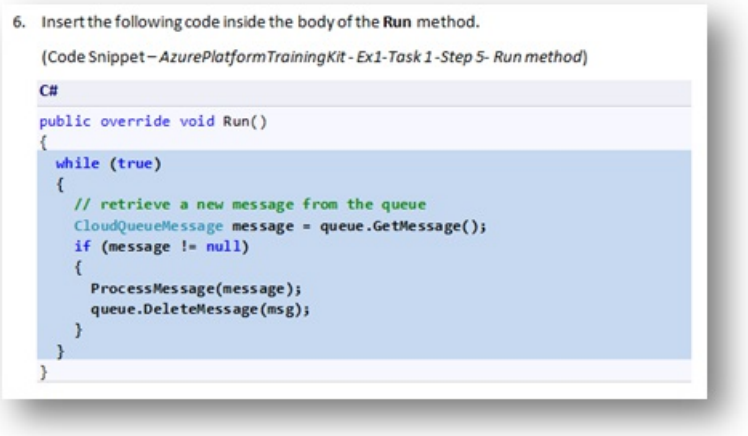
9. Once the publishing process finishes, your default browser will open the published web site.



Application published to Windows Azure

Appendix C: Using Code Snippets

With code snippets, you have all the code you need at your fingertips. The lab document will tell you exactly when you can use them, as shown in the following figure.



6. Insert the following code inside the body of the **Run** method.

(Code Snippet – *AzurePlatformTrainingKit - Ex1-Task 1 -Step 5- Run method*)

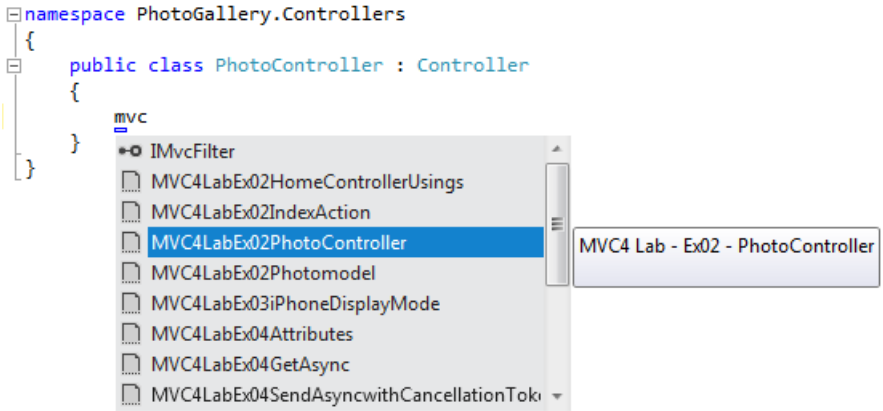
C#

```
public override void Run()
{
  while (true)
  {
    // retrieve a new message from the queue
    CloudQueueMessage message = queue.GetMessage();
    if (message != null)
    {
      ProcessMessage(message);
      queue.DeleteMessage(msg);
    }
  }
}
```

Using Visual Studio code snippets to insert code into your project

To add a code snippet using the keyboard (C# only)

1. Place the cursor where you would like to insert the code.
2. Start typing the snippet name (without spaces or hyphens).
3. Watch as IntelliSense displays matching snippets' names.
4. Select the correct snippet (or keep typing until the entire snippet's name is selected).
5. Press the Tab key twice to insert the snippet at the cursor location.



Start typing the snippet name

```
static void Main(string[] args)
{
    AzureTrainingKitServiceBus
}
```

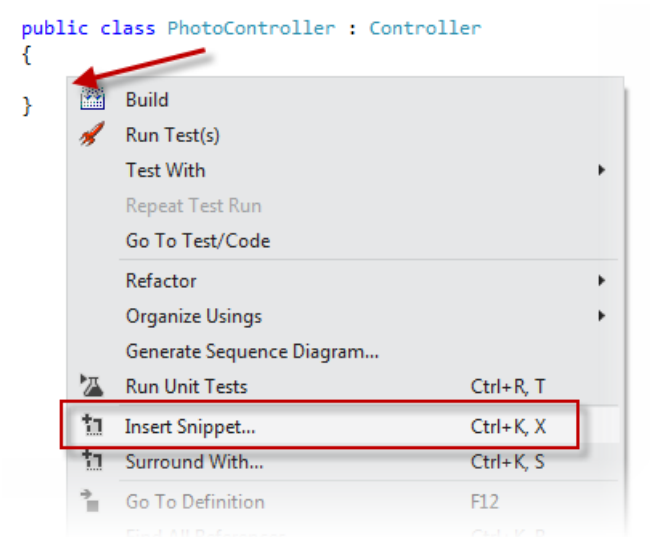
Press Tab to select the highlighted snippet

```
static void Main(string[] args)
{
    request.Headers.Add("$accountKey", accountKey);
    request.Headers.Add("$uniqueUserID", uniqueUserId);
}
```

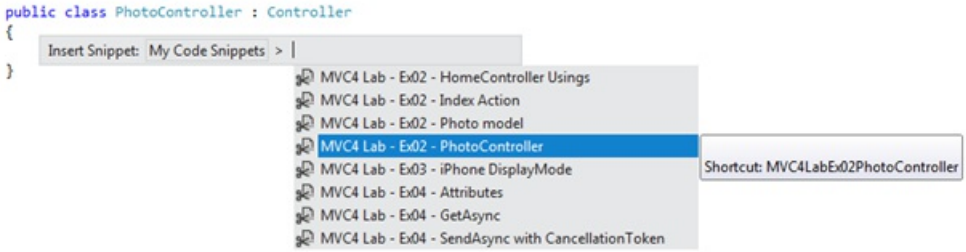
Press Tab again and the snippet will expand

To add a code snippet using the mouse (C#, Visual Basic and XML) 1. Right-click where you want to insert the code snippet.

1. Select **Insert Snippet** followed by **My Code Snippets**.
2. Pick the relevant snippet from the list, by clicking on it.



Right-click where you want to insert the code snippet and select Insert Snippet



Pick the relevant snippet from the list, by clicking on it

ASP.NET MVC 4 Models and Data Access

2/20/2020 • 27 minutes to read • Edit Online

By Web Camps Team

Download Web Camps Training Kit

This Hands-on Lab assumes you have basic knowledge of **ASP.NET MVC**. If you have not used **ASP.NET MVC** before, we recommend you to go over **ASP.NET MVC 4 Fundamentals** Hands-on Lab.

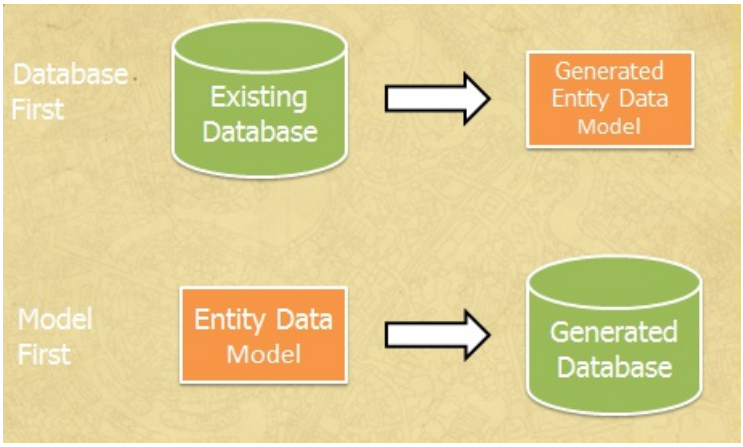
This lab walks you through the enhancements and new features previously described by applying minor changes to a sample Web application provided in the Source folder.

> **NOTE**
>
> All sample code and snippets are included in the Web Camps Training Kit, available at Microsoft-Web/WebCampTrainingKit Releases. The project specific to this lab is available at ASP.NET MVC 4 Models and Data Access.

In **ASP.NET MVC Fundamentals** Hands-on Lab, you have been passing hard-coded data from the Controllers to the View templates. But, in order to build a real Web application, you might want to use a real database.

This Hands-on Lab will show you how to use a database engine in order to store and retrieve the data needed for the Music Store application. To accomplish that, you will start with an existing database and create the Entity Data Model from it. Throughout this lab, you will meet the **Database First** approach as well as the **Code First** approach.

However, you can also use the **Model First** approach, create the same model using the tools, and then generate the database from it.



Database First vs. Model First

After generating the Model, you will make the proper adjustments in the StoreController to provide the Store Views with the data taken from the database, instead of using hard-coded data. You will not need to make any change to the View templates because the StoreController will be returning the same ViewModels to the View templates, although this time the data will come from the database.

The Code First Approach

The Code First approach allows us to define the model from the code without generating classes that are generally coupled with the framework.

In code first, model objects are defined with POCOs, "Plain Old CLR Objects". POCOs are simple plain classes that have no inheritance and do not implement interfaces. We can automatically generate the database from them, or we can use an existing database and generate the class mapping from the code.

The benefits of using this approach is that the Model remains independent from the persistence framework (in this case, Entity Framework), as the POCOs classes are not coupled with the mapping framework.

> **NOTE**
> This Lab is based on ASP.NET MVC 4 and a version of the Music Store sample application customized and minimized to fit only the features shown in this Hands-On Lab.
>
> If you wish to explore the whole **Music Store** tutorial application you can find it in MVC-Music-Store.

Prerequisites

You must have the following items to complete this lab:

- Microsoft Visual Studio Express 2012 for Web or superior (read Appendix A for instructions on how to install it).

Setup

Installing Code Snippets

For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. To install the code snippets run **.\Source\Setup\CodeSnippets.vsi** file.

If you are not familiar with the Visual Studio Code Snippets, and want to learn how to use them, you can refer to the appendix from this document "Appendix C: Using Code Snippets".

Exercises

This Hands-on Lab is comprised by the following exercises:

1. Exercise 1: Adding a Database
2. Exercise 2: Creating a Database using Code First
3. Exercise 3: Querying the Database with Parameters

> **NOTE**
> Each exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

Estimated time to complete this lab: **35 minutes**.

Exercise 1: Adding a Database

In this exercise, you will learn how to add a database with the tables of the MusicStore application to the solution in order to consume its data. Once the database is generated with the model, and added to the solution, you will modify the StoreController class to provide the View template with the data taken from the database, instead of using hard-coded values.

Task 1 - Adding a Database

In this task, you will add an already created database with the main tables of the MusicStore application to the solution.

1. Open the **Begin** solution located at **Source/Ex1-AddingADatabaseDBFirst/Begin/** folder.
    a. You will need to download some missing NuGet packages before continue. To do this, click the

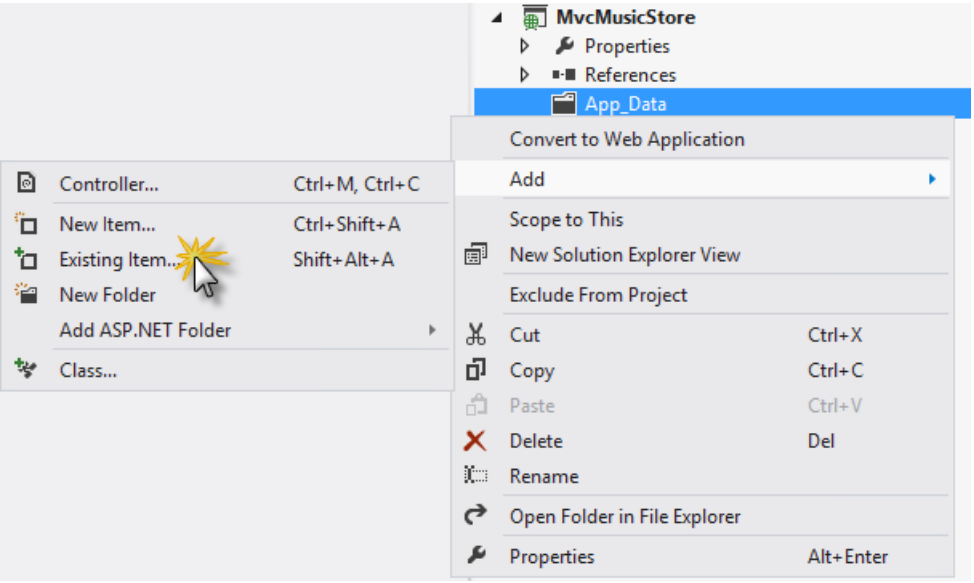
Project menu and select **Manage NuGet Packages**.

b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

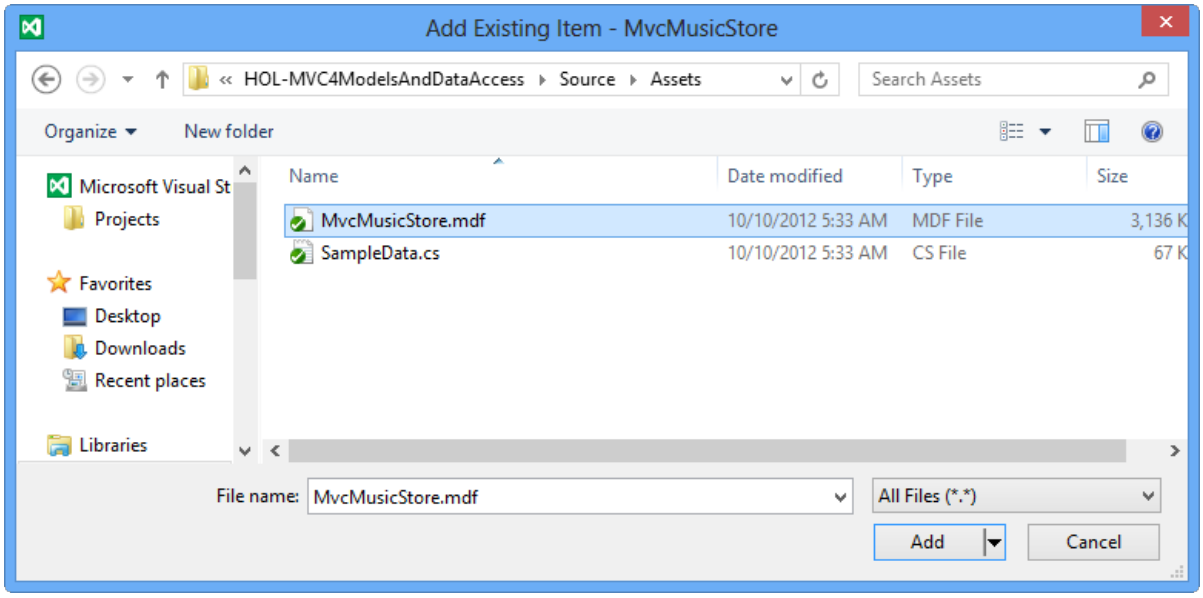
c. Finally, build the solution by clicking **Build | Build Solution**.

> **NOTE**
>
> One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Add **MvcMusicStore** database file. In this Hands-on Lab, you will use an already created database called **MvcMusicStore.mdf**. To do that, right-click **App_Data** folder, point to **Add** and then click **Existing Item**. Browse to **\Source\Assets** and select the **MvcMusicStore.mdf** file.

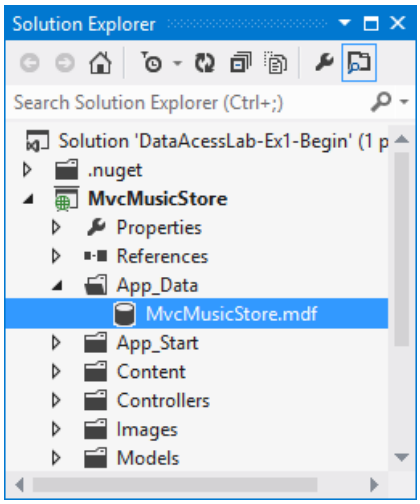


Adding an Existing Item



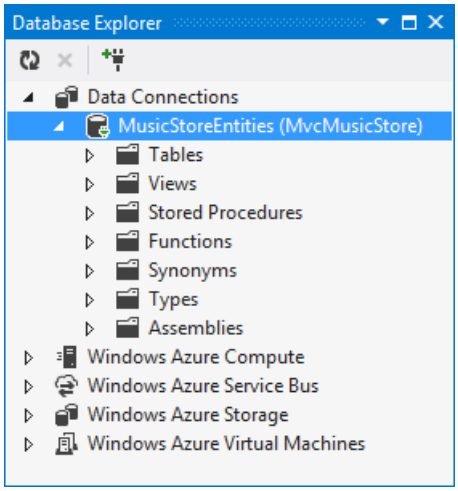
MvcMusicStore.mdf database file

The database has been added to the project. Even when the database is located inside the solution, you can query and update it as it was hosted in a different database server.



MvcMusicStore database in Solution Explorer

3. Verify the connection to the database. To do this, double-click **MvcMusicStore.mdf** to establish a connection.

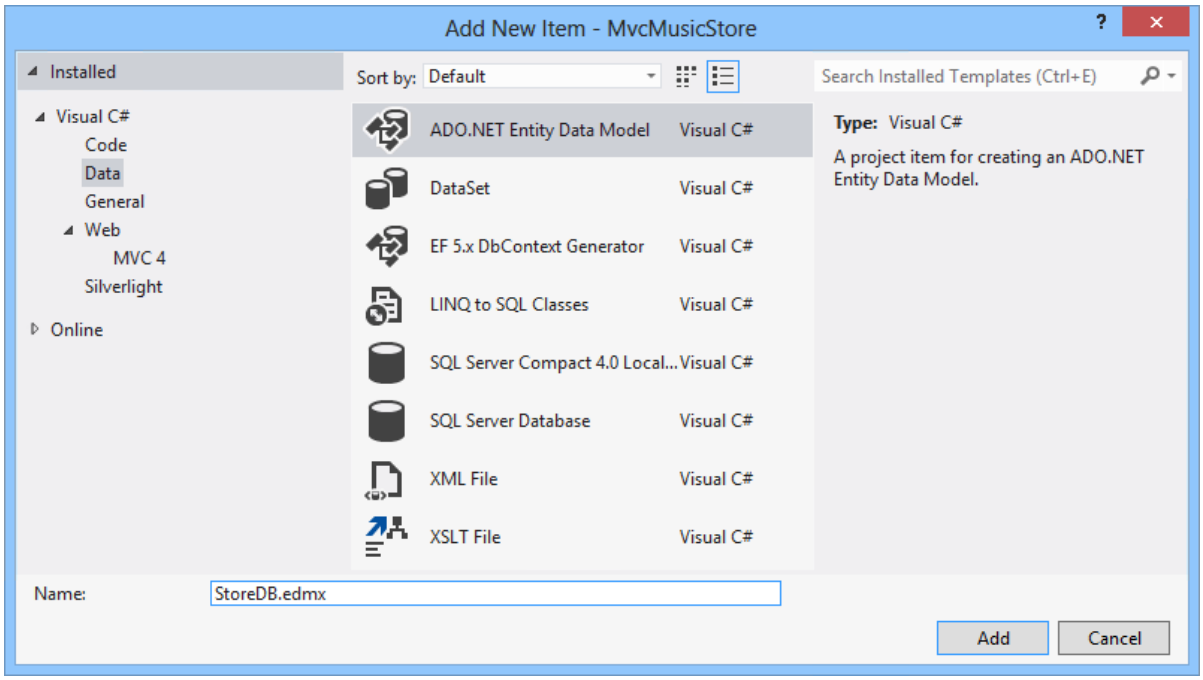


Connecting to MvcMusicStore.mdf

Task 2 - Creating a Data Model

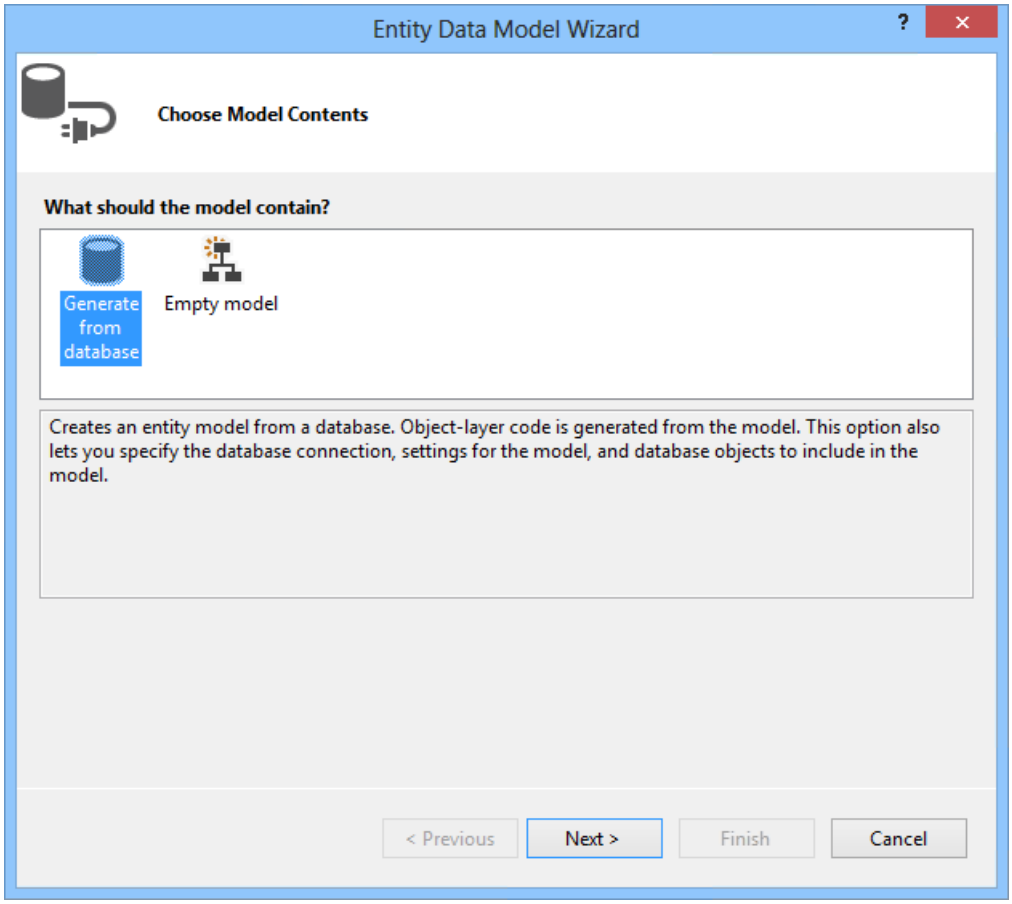
In this task, you will create a data model to interact with the database added in the previous task.

1. Create a data model that will represent the database. To do this, in Solution Explorer right-click the **Models** folder, point to **Add** and then click **New Item**. In the **Add New Item** dialog, select the **Data** template and then the **ADO.NET Entity Data Model** item. Change the data model name to **StoreDB.edmx** and click **Add**.



Adding the StoreDB ADO.NET Entity Data Model

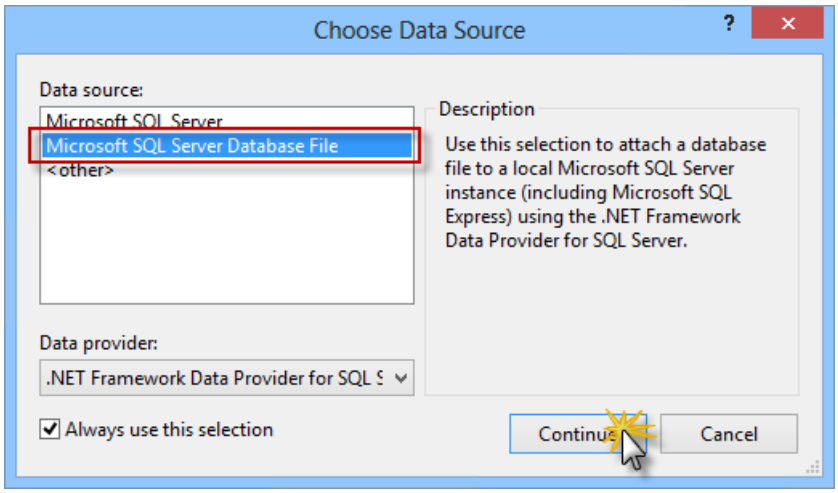
2. The **Entity Data Model Wizard** will appear. This wizard will guide you through the creation of the model layer. Since the model should be created based on the existing database recently added, select **Generate from database** and click **Next**.



Choosing the model content

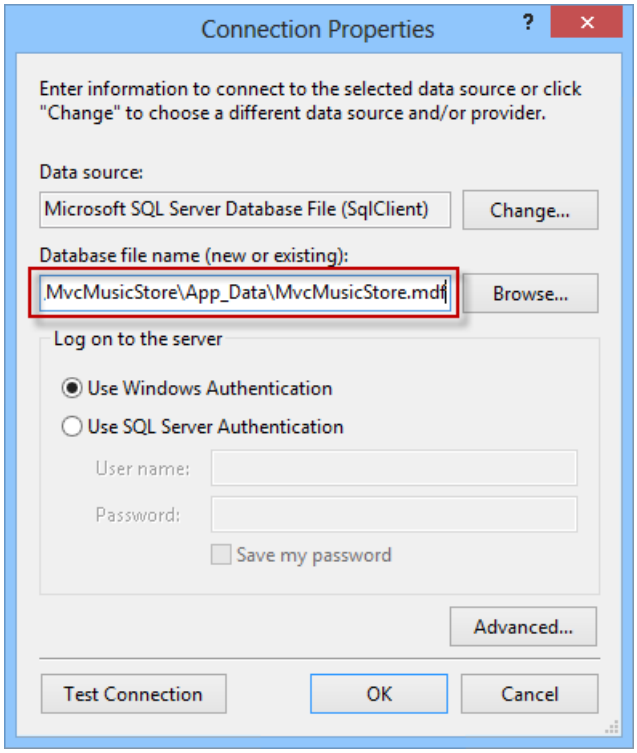
3. Since you are generating a model from a database, you will need to specify the connection to use. Click **New Connection**.

4. Select **Microsoft SQL Server Database File** and click **Continue**.



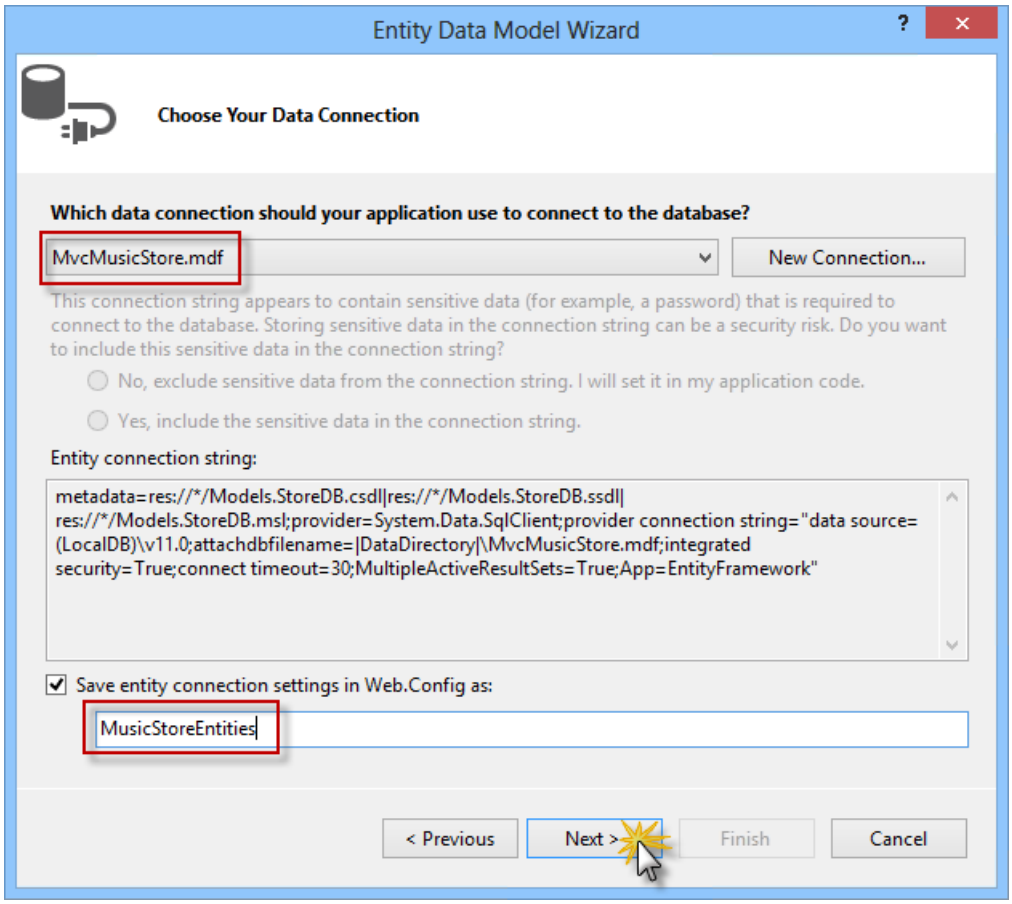
Choose data source dialog

5. Click **Browse** and select the database **MvcMusicStore.mdf** you located in the **App_Data** folder and click **OK**.



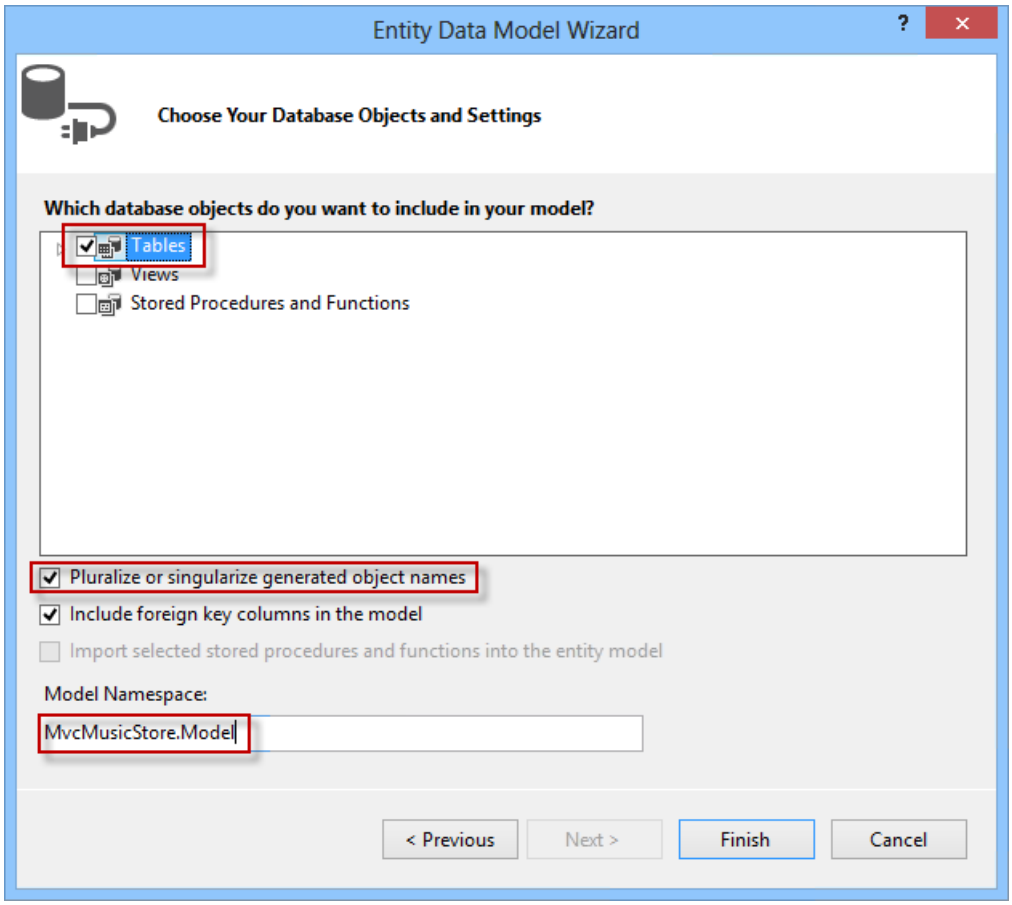
Connection properties

6. The generated class should have the same name as the entity connection string, so change its name to **MusicStoreEntities** and click **Next**.



Choosing the data connection

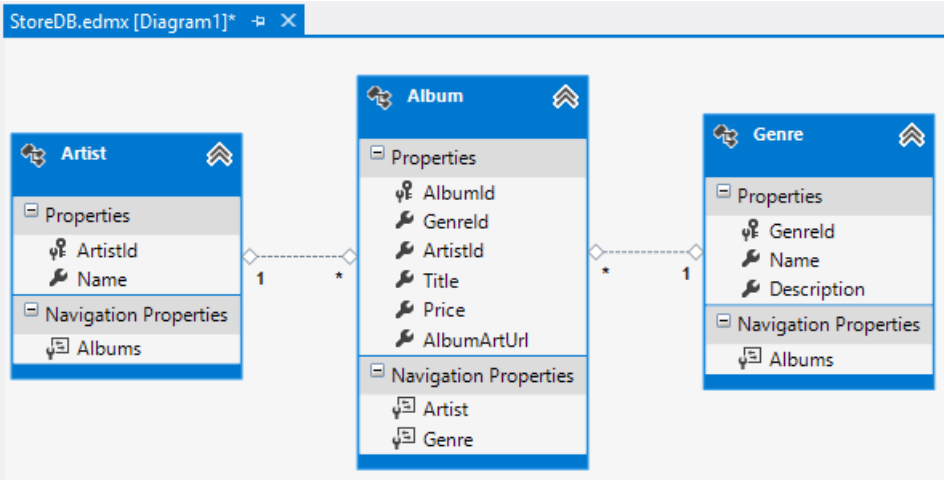
7. Choose the database objects to use. As the Entity Model will use just the database's tables, select the **Tables** option, and make sure that the **Include foreign key columns in the model** and **Pluralize or singularize generated object names** options are also selected. Change the Model Namespace to **MvcMusicStore.Model** and click **Finish**.



Choosing the database objects

> **NOTE**
>
> If a Security Warning dialog is shown, click **OK** to run the template and generate the classes for the model entities.

8. An entity diagram for the database will appear, while a separate class that maps each table to the database will be created. For example, the **Albums** table will be represented by an **Album** class, where each column in the table will map to a class property. This will allow you to query and work with objects that represent rows in the database.



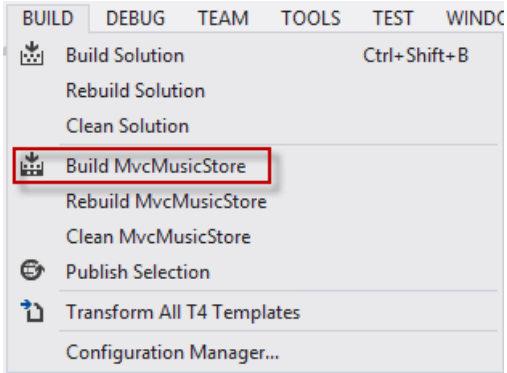
Entity diagram

> **NOTE**
>
> The T4 templates (.tt) run code to generate the entities classes and will overwrite the existing classes with the same name. In this example, the classes "Album", "Genre" and "Artist" were overwritten with the generated code.

Task 3 - Building the Application

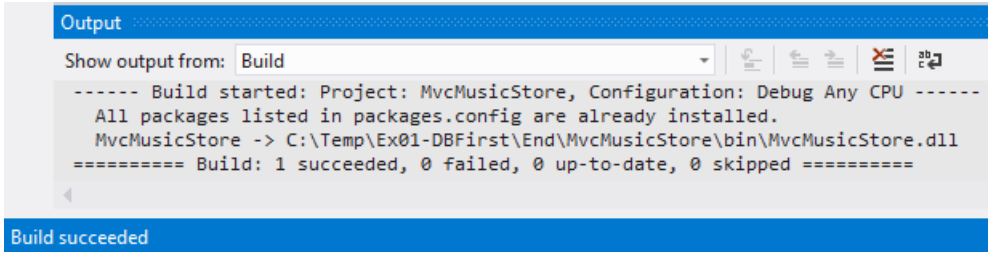
In this task, you will check that, although the model generation have removed the **Album**, **Genre** and **Artist** model classes, the project builds successfully by using the new data model classes.

1. Build the project by selecting the **Build** menu item and then **Build MvcMusicStore**.



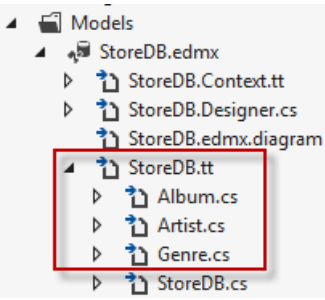
Building the project

2. The project builds successfully. Why does it still work? It works because the database tables have fields that include the properties that you were using in the removed classes **Album** and **Genre**.



Builds succeeded

3. While the designer displays the entities in a diagram format, they are really C# classes. Expand the **StoreDB.edmx** node in the Solution Explorer and then **StoreDB.tt**, you will see the new generated entities.



Generated files

Task 4 - Querying the Database

In this task, you will update the StoreController class so that, instead of using hardcoded data, it will query the database to retrieve the information.

1. Open **Controllers\StoreController.cs** and add the following field to the class to hold an instance of the **MusicStoreEntities** class, named **storeDB**:

   (Code Snippet - *Models And Data Access - Ex1 storeDB*)

```
public class StoreController : Controller
{
    private MusicStoreEntities storeDB = new MusicStoreEntities();
```

2. The **MusicStoreEntities** class exposes a collection property for each table in the database. Update **Browse** action method to retrieve a Genre with all of the **Albums**.

   (Code Snippet - *Models And Data Access - Ex1 Store Browse*)

```
public ActionResult Browse(string genre)
{
    // Retrieve Genre and its Associated Albums from database
    var genreModel = new Genre
    {
        Name = genre,
        Albums = this.storeDB.Albums.ToList()
    };

    return this.View(genreModel);
}
```

> **NOTE**
>
> You are using a capability of .NET called **LINQ** (language-integrated query) to write strongly-typed query expressions against these collections - which will execute code against the database and return objects that you can program against.
>
> For more information about LINQ, please visit the msdn site.

3. Update **Index** action method to retrieve all the genres.

   (Code Snippet - *Models And Data Access - Ex1 Store Index*)

```
public ActionResult Index()
{
    var genres = this.storeDB.Genres;

    return this.View(genres);
}
```

4. Update **Index** action method to retrieve all the genres and transform the collection to a list.

   (Code Snippet - *Models And Data Access - Ex1 Store GenreMenu*)

```
// GET: /Store/GenreMenu
[ChildActionOnly]
public ActionResult GenreMenu()
{
    var genres = this.storeDB.Genres.Take(9).ToList();

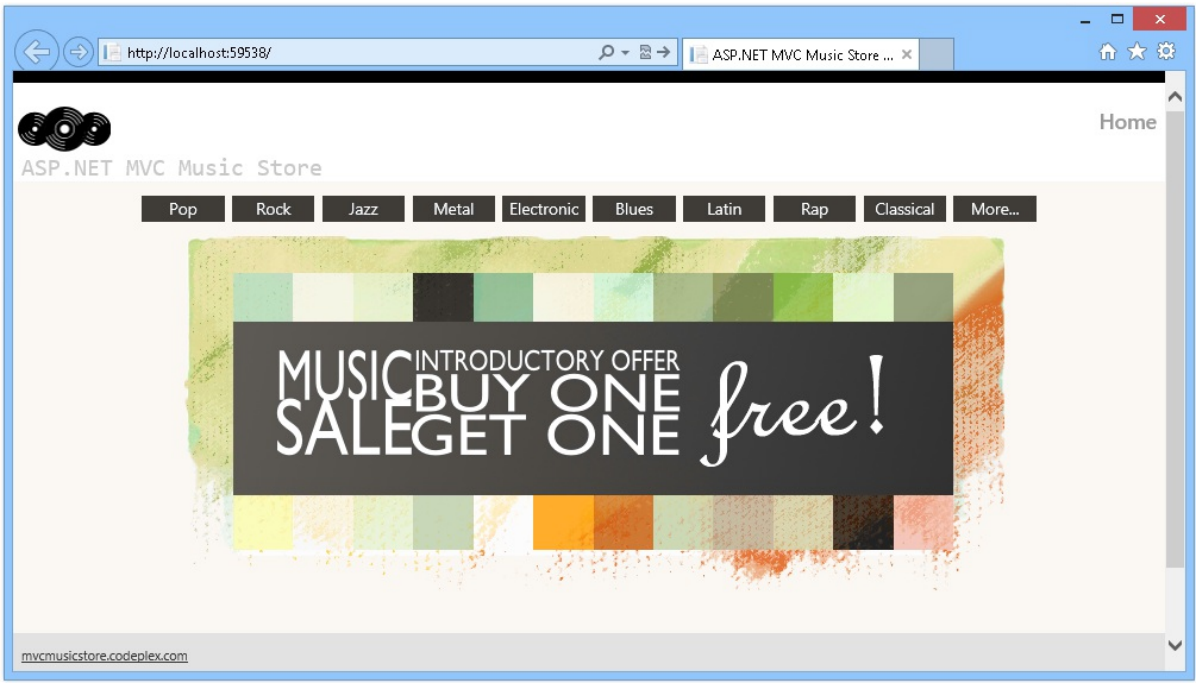
    return this.PartialView(genres);
}
```

Task 5 - Running the Application

In this task, you will check that the Store Index page will now display the Genres stored in the database instead of the hardcoded ones. There is no need to change the View template because the **StoreController** is returning the same entities as before, although this time the data will come from the database.

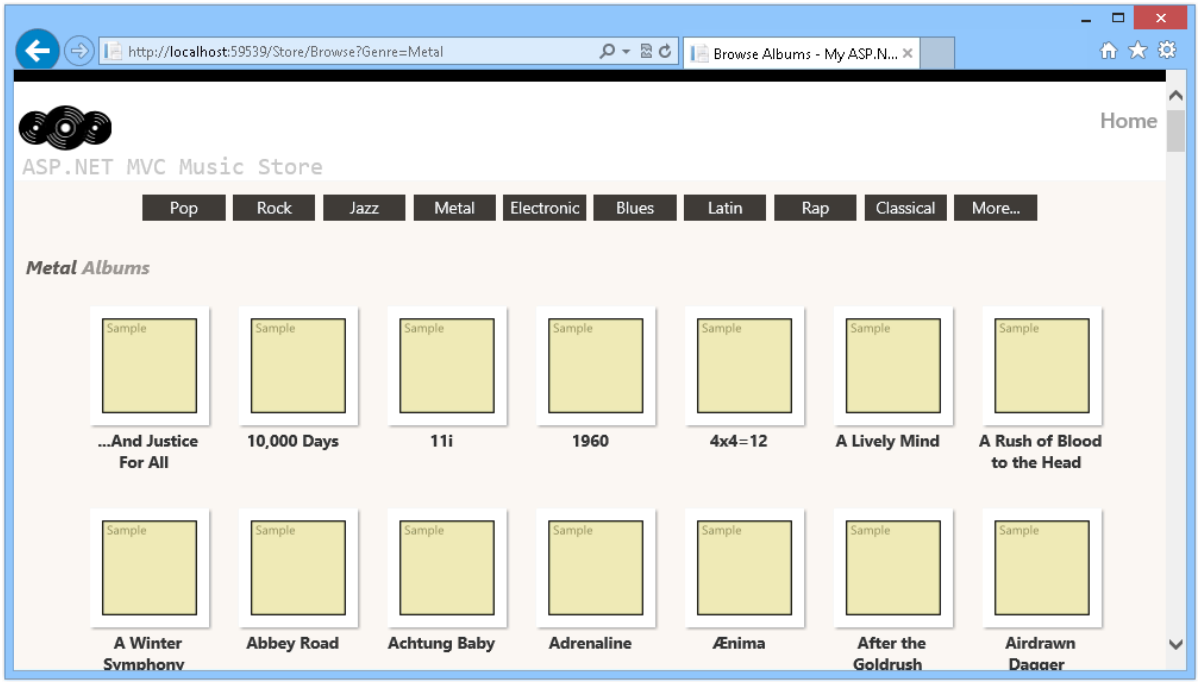
1. Rebuild the solution and press **F5** to run the Application.

2. The project starts in the Home page. Verify that the menu of **Genres** is no longer a hardcoded list, and the data is directly retrieved from the database.



Browsing Genres from the database

3. Now browse to any genre and verify the albums are populated from database.



Browsing Albums from the database

Exercise 2: Creating a Database Using Code First

In this exercise, you will learn how to use the Code First approach to create a database with the tables of the MusicStore application, and how to access its data.

Once the model is generated, you will modify the StoreController to provide the View template with the data taken from the database, instead of using hardcoded values.

> **NOTE**
> If you have completed Exercise 1 and have already worked with the Database First approach, you will now learn how to get the same results with a different process. The tasks that are in common with Exercise 1 have been marked to make your reading easier. If you have not completed Exercise 1 but would like to learn the Code First approach, you can start from this exercise and get a full coverage of the topic.

Task 1 - Populating Sample Data

In this task, you will populate the database with sample data when it is initially created using Code-First.

1. Open the **Begin** solution located at **Source/Ex2-CreatingADatabaseCodeFirst/Begin/** folder. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.

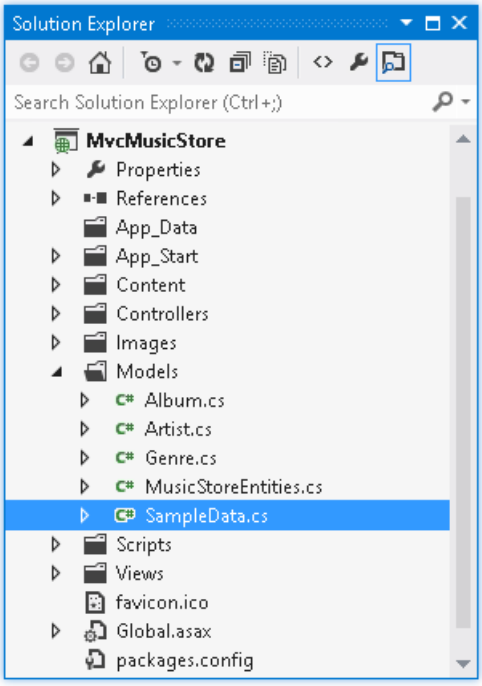
   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Add the **SampleData.cs** file to the **Models** folder. To do that, right-click **Models** folder, point to **Add** and then click **Existing Item**. Browse to **\Source\Assets** and select the **SampleData.cs** file.



Sample data populate code

3. Open the **Global.asax.cs** file and add the following *using* statements.

   (Code Snippet - *Models And Data Access - Ex2 Global Asax Usings*)

```
using MvcMusicStore.Models;
using System.Data.Entity;
```

4. In the **Application_Start()** method add the following line to set the database initializer.

   (Code Snippet - *Models And Data Access - Ex2 Global Asax SetInitializer*)

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();

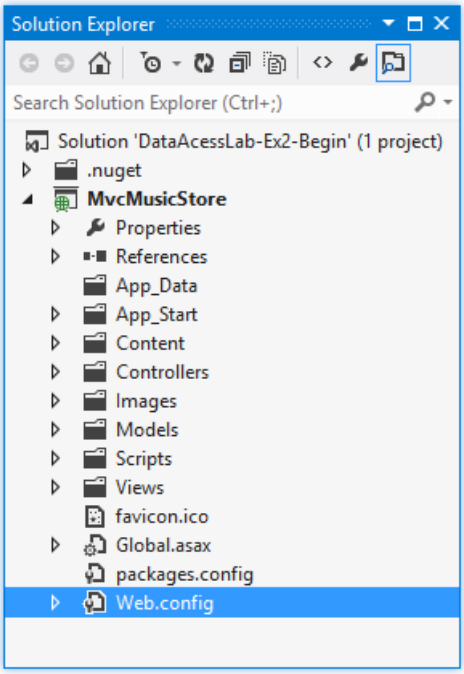
    WebApiConfig.Register(GlobalConfiguration.Configuration);
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);

    Database.SetInitializer(new SampleData());
}
```

Task 2 - Configuring the connection to the Database

Now that you have already added a database to our project, you will write in the **Web.config** file the connection string.

1. Add a connection string at **Web.config**. To do that, open **Web.config** at project root and replace the connection string named DefaultConnection with this line in the **<connectionStrings>** section:



Web.config file location

```
<configuration>
...
  <connectionStrings>
    <add name="MusicStoreEntities" connectionString="data source=(LocalDb)\v11.0;initial
catalog=MvcMusicStore;Integrated Security=SSPI;AttachDBFilename=|DataDirectory|\MvcMusicStore.mdf"
providerName="System.Data.SqlClient" />
  </connectionStrings>
...
```

Task 3 - Working with the Model

Now that you have already configured the connection to the database, you will link the model with the database tables. In this task, you will create a class that will be linked to the database with Code First. Remember that there is an existent POCO model class that should be modified.

> **NOTE**
>
> If you completed Exercise 1, you will note that this step was performed by a wizard. By doing Code First, you will manually create classes that will be linked to data entities.

1. Open the POCO model class **Genre** from **Models** project folder and include an ID. Use an int property with the name **GenreId**.

   (Code Snippet - *Models And Data Access - Ex2 Code First Genre*)

```
namespace MvcMusicStore.Models
{
    using System.Collections.Generic;

    public class Genre
    {
      public int GenreId { get; set; }

      public string Name { get; set; }

      public string Description { get; set; }

      public List<Album> Albums { get; set; }
    }
}
```

   > **NOTE**
   >
   > To work with Code First conventions, the class Genre must have a primary key property that will be automatically detected.
   >
   > You can read more about Code First Conventions in this msdn article.

2. Now, open the POCO model class **Album** from **Models** project folder and include the foreign keys, create properties with the names **GenreId** and **ArtistId**. This class already have the **GenreId** for the primary key.

   (Code Snippet - *Models And Data Access - Ex2 Code First Album*)

```
namespace MvcMusicStore.Models
{
    public class Album
    {
        public int AlbumId { get; set; }

        public int GenreId { get; set; }

        public int ArtistId { get; set; }

        public string Title { get; set; }

        public decimal Price { get; set; }

        public string AlbumArtUrl { get; set; }

        public virtual Genre Genre { get; set; }

        public virtual Artist Artist { get; set; }
    }
}
```

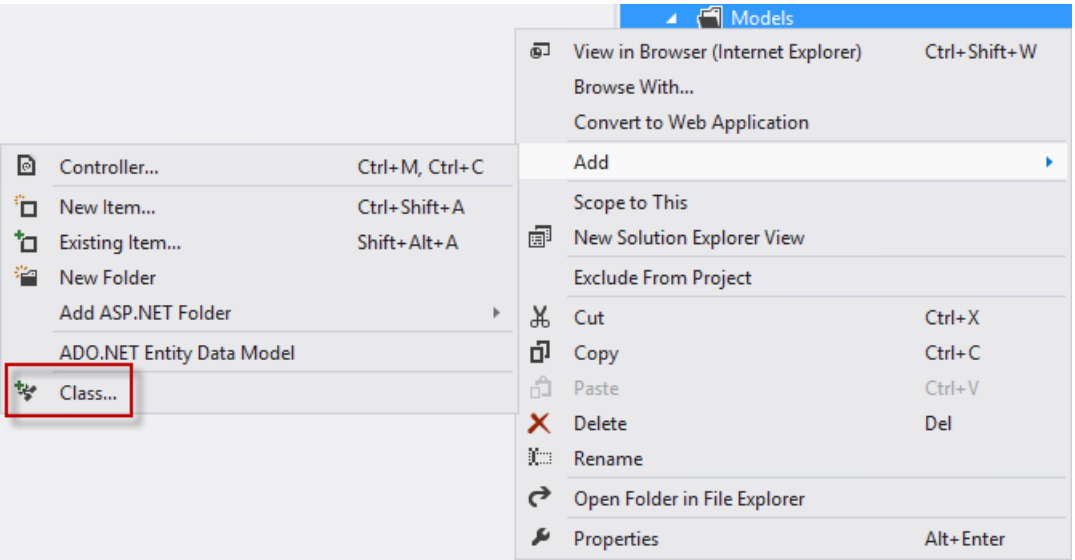
3. Open the POCO model class **Artist** and include the **ArtistId** property.

   (Code Snippet - *Models And Data Access - Ex2 Code First Artist*)

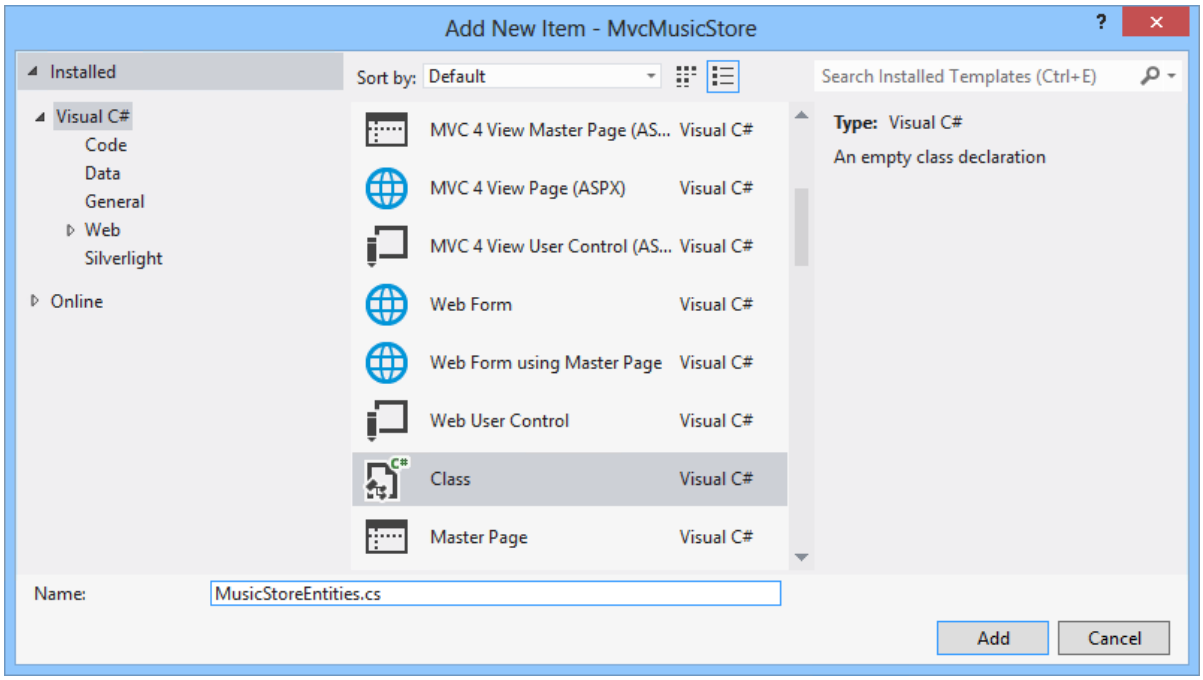
```
namespace MvcMusicStore.Models
{
    public class Artist
    {
        public int ArtistId { get; set; }

        public string Name { get; set; }
    }
}
```

4. Right-click the **Models** project folder and select **Add | Class**. Name the file **MusicStoreEntities.cs**. Then, click **Add**.



Adding a new item



Adding a class

5. Open the class you have just created, **MusicStoreEntities.cs**, and include the namespaces **System.Data.Entity** and **System.Data.Entity.Infrastructure**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
...
```

6. Replace the class declaration to extend the **DbContext** class: declare a public **DBSet** and override **OnModelCreating** method. After this step you will get a domain class that will link your model with the Entity Framework. In order to do that, replace the class code with the following:

   (Code Snippet - *Models And Data Access - Ex2 Code First MusicStoreEntities*)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;

namespace MvcMusicStore.Models
{
    public class MusicStoreEntities : DbContext
    {
        public DbSet<Genre> Genres { get; set; }

        public DbSet<Album> Albums { get; set; }

        public DbSet<Artist> Artists { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Genre>().ToTable("Genres");
            modelBuilder.Entity<Album>().ToTable("Albums");
            modelBuilder.Entity<Artist>().ToTable("Artists");

            base.OnModelCreating(modelBuilder);
        }
    }
}
```

NOTE

With Entity Framework **DbContext** and **DBSet** you will be able to query the POCO class Genre. By extending **OnModelCreating** method, you are specifying in the **code** how Genre will be mapped to a database table. You can find more information about DBContext and DBSet in this msdn article: link

Task 4 - Querying the Database

In this task, you will update the StoreController class so that, instead of using hardcoded data, it will retrieve it from the database.

NOTE

This task is in common with Exercise 1.

If you completed Exercise 1 you will note these steps are the same in both approaches (Database first or Code first). They are different in how the data is linked with the model, but the access to data entities is yet transparent from the controller.

1. Open **Controllers\StoreController.cs** and add the following field to the class to hold an instance of the **MusicStoreEntities** class, named **storeDB**:

   (Code Snippet - *Models And Data Access - Ex1 storeDB*)

```
public class StoreController : Controller
{
    private MusicStoreEntities storeDB = new MusicStoreEntities();
    ...
}
```

2. The **MusicStoreEntities** class exposes a collection property for each table in the database. Update **Browse** action method to retrieve a Genre with all of the **Albums**.

(Code Snippet - *Models And Data Access - Ex2 Store Browse*)

```
public ActionResult Browse(string genre)
{
     // Retrieve Genre and its Associated Albums from database
     var genreModel = new Genre
    {
        Name = genre,
        Albums = this.storeDB.Albums.ToList()
    };

     return this.View(genreModel);
}
```

> **NOTE**
>
> You are using a capability of .NET called **LINQ** (language-integrated query) to write strongly-typed query expressions against these collections - which will execute code against the database and return objects that you can program against.
>
> For more information about LINQ, please visit the msdn site.

3. Update **Index** action method to retrieve all the genres.

   (Code Snippet - *Models And Data Access - Ex2 Store Index*)

```
public ActionResult Index()
{
    var genres = this.storeDB.Genres;

    return this.View(genres);
}
```

4. Update **Index** action method to retrieve all the genres and transform the collection to a list.

   (Code Snippet - *Models And Data Access - Ex2 Store GenreMenu*)

```
// GET: /Store/GenreMenu
[ChildActionOnly]
public ActionResult GenreMenu()
{
    var genres = this.storeDB.Genres.Take(9).ToList();

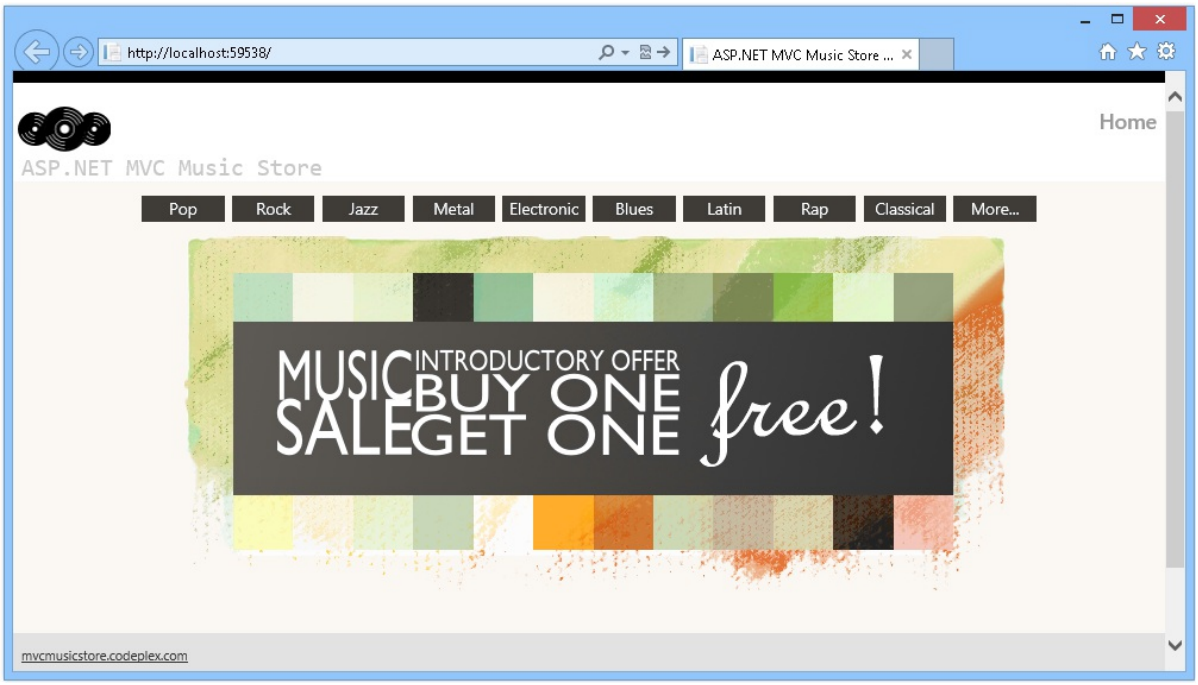
    return this.PartialView(genres);
}
```

Task 5 - Running the Application

In this task, you will check that the Store Index page will now display the Genres stored in the database instead of the hardcoded ones. There is no need to change the View template because the **StoreController** is returning the same **StoreIndexViewModel** as before, but this time the data will come from the database.

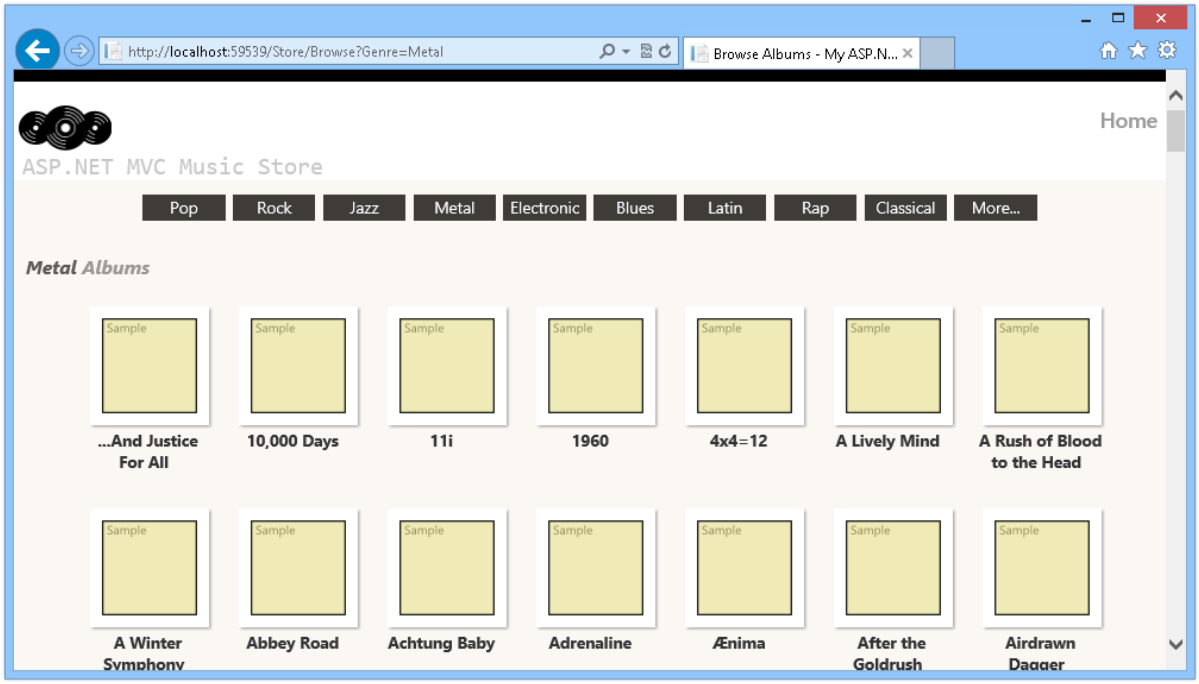
1. Rebuild the solution and press **F5** to run the Application.

2. The project starts in the Home page. Verify that the menu of **Genres** is no longer a hardcoded list, and the data is directly retrieved from the database.



Browsing Genres from the database

3. Now browse to any genre and verify the albums are populated from database.



Browsing Albums from the database

Exercise 3: Querying the Database with Parameters

In this exercise, you will learn how to query the database using parameters, and how to use Query Result Shaping, a feature that reduces the number database accesses retrieving data in a more efficient way.

> **NOTE**
>
> For further information on Query Result Shaping, visit the following msdn article.

Task 1 - Modifying StoreController to Retrieve Albums from Database

In this task, you will change the `StoreController` class to access the database to retrieve albums from a specific genre.

1. Open the **Begin** solution located at the **Source\Ex3-QueryingTheDatabaseWithParametersCodeFirst\Begin** folder if you want to use Code-First approach or **Source\Ex3-QueryingTheDatabaseWithParametersDBFirst\Begin** folder if you want to use Database-First approach. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.

   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Open the **StoreController** class to change the **Browse** action method. To do this, in the **Solution Explorer**, expand the **Controllers** folder and double-click **StoreController.cs**.

3. Change the **Browse** action method to retrieve albums for a specific genre. To do this, replace the following code:

   (Code Snippet - *Models And Data Access - Ex3 StoreController BrowseMethod*)

```
public ActionResult Browse(string genre)
{
    // Retrieve Genre and its Associated Albums from database
    var genreModel = this.storeDB.Genres.Include("Albums")
        .Single(g => g.Name == genre);

    return this.View(genreModel);
}
```

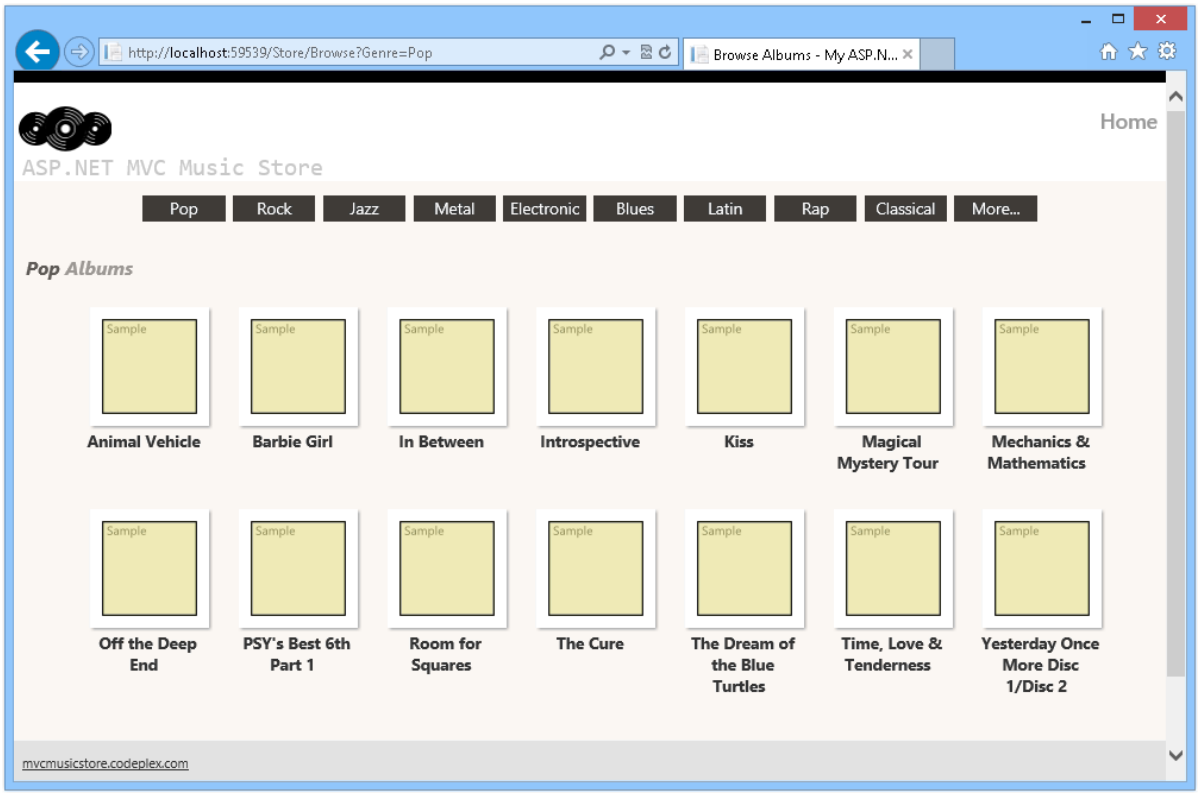
> **NOTE**
> To populate a collection of the entity, you need to use the **Include** method to specify you want to retrieve the albums too. You can use the **.Single()** extension in LINQ because in this case only one genre is expected for an album. The **Single()** method takes a Lambda expression as a parameter, which in this case specifies a single Genre object such that its name matches the value defined.
>
> You will take advantage of a feature that allows you to indicate other related entities you want loaded as well when the Genre object is retrieved. This feature is called **Query Result Shaping**, and enables you to reduce the number of times needed to access the database to retrieve information. In this scenario, you will want to pre-fetch the Albums for the Genre you retrieve.
>
> The query includes **Genres.Include("Albums")** to indicate that you want related albums as well. This will result in a more efficient application, since it will retrieve both Genre and Album data in a single database request.

Task 2 - Running the Application

In this task, you will run the application and retrieve albums of a specific genre from the database.

1. Press **F5** to run the Application.

2. The project starts in the Home page. Change the URL to **/Store/Browse?genre=Pop** to verify that the results are being retrieved from the database.



Browsing /Store/Browse?genre=Pop

Task 3 - Accessing Albums by Id

In this task, you will repeat the previous procedure to get albums by their Id.

1. Close the browser if needed, to return to Visual Studio. Open the **StoreController** class to change the **Details** action method. To do this, in the **Solution Explorer**, expand the **Controllers** folder and double-click **StoreController.cs**.

2. Change the **Details** action method to retrieve albums details based on their **Id**. To do this, replace the following code:

   (Code Snippet - *Models And Data Access - Ex3 StoreController DetailsMethod*)

```
// GET: /Store/
public ActionResult Details(int id)
{
    var album = this.storeDB.Albums.Find(id);

    if (album == null)
    {
        return this.HttpNotFound();
    }

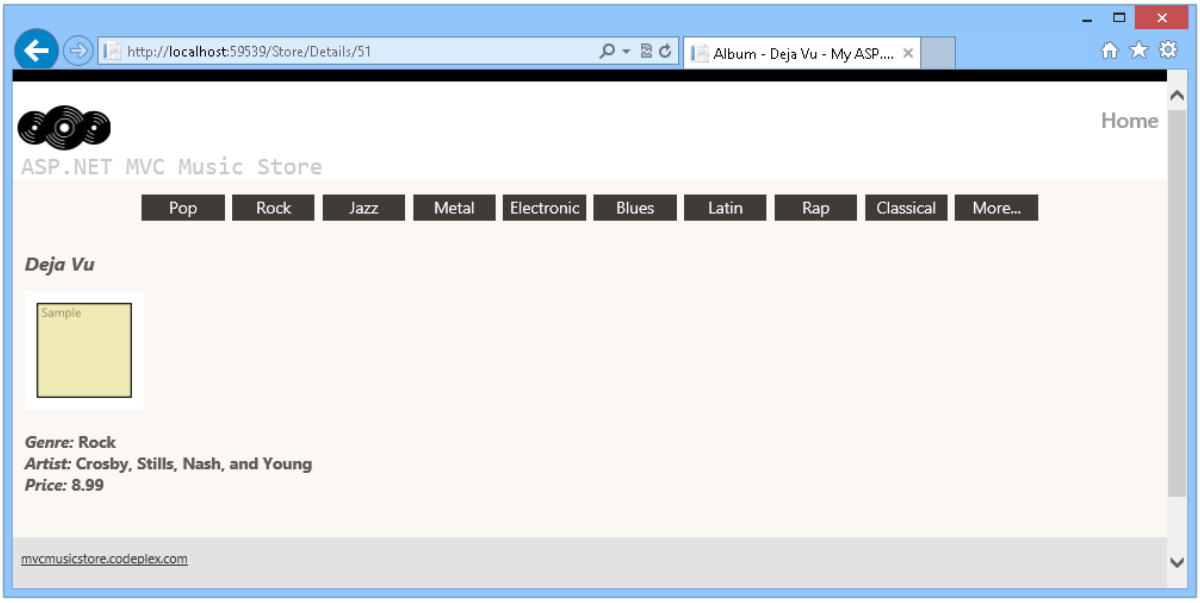
    return this.View(album);
}
```

Task 4 - Running the Application

In this task, you will run the Application in a web browser and obtain album details by their Id.

1. Press **F5** to run the Application.

2. The project starts in the Home page. Change the URL to **/Store/Details/51** or browse the genres and select an album to verify that the results are being retrieved from the database.



Browsing /Store/Details/51

> **NOTE**
>
> Additionally, you can deploy this application to Windows Azure Web Sites following Appendix B: Publishing an ASP.NET MVC 4 Application using Web Deploy.

Summary

By completing this Hands-on Lab you have learned the fundamentals of ASP.NET MVC Models and Data Access, using the **Database First** approach as well as the **Code First** Approach:

- How to add a database to the solution in order to consume its data
- How to update Controllers to provide View templates with the data taken from the database instead of hard-coded one
- How to query the database using parameters
- How to use the Query Result Shaping, a feature that reduces the number of database accesses, retrieving data in a more efficient way
- How to use both Database First and Code First approaches in Microsoft Entity Framework to link the database with the model

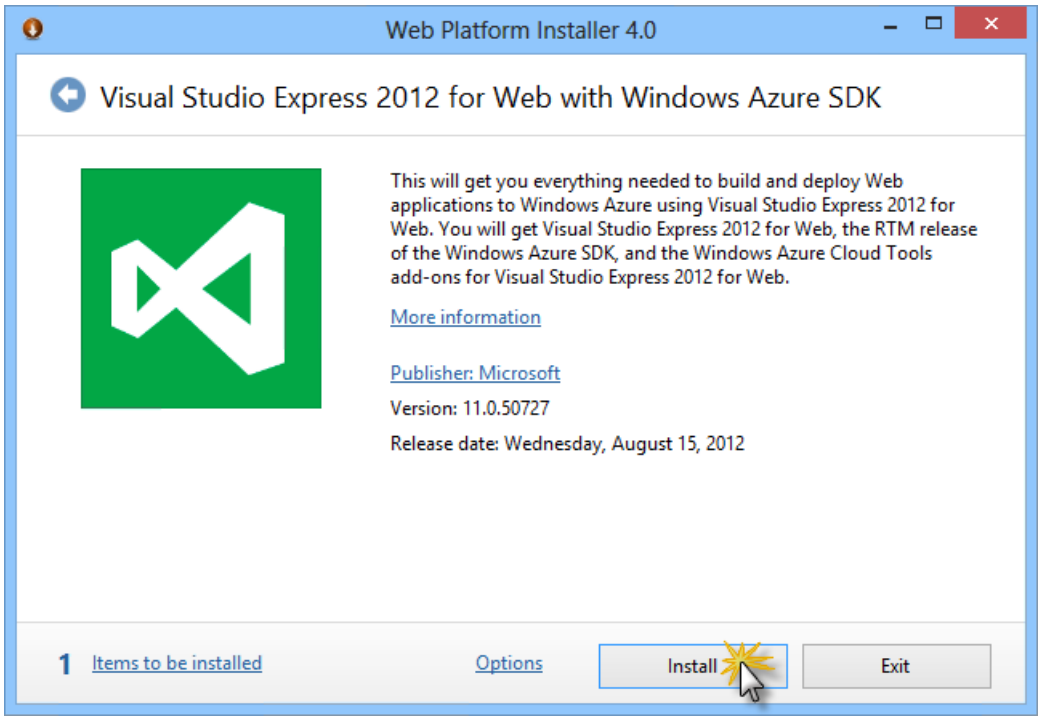
Appendix A: Installing Visual Studio Express 2012 for Web

You can install **Microsoft Visual Studio Express 2012 for Web** or another "Express" version using the **Microsoft Web Platform Installer**. The following instructions guide you through the steps required to install *Visual studio Express 2012 for Web* using *Microsoft Web Platform Installer*.

1. Go to https://go.microsoft.com/?linkid=9810169. Alternatively, if you already have installed Web Platform Installer, you can open it and search for the product "*Visual Studio Express 2012 for Web with Windows Azure SDK*".

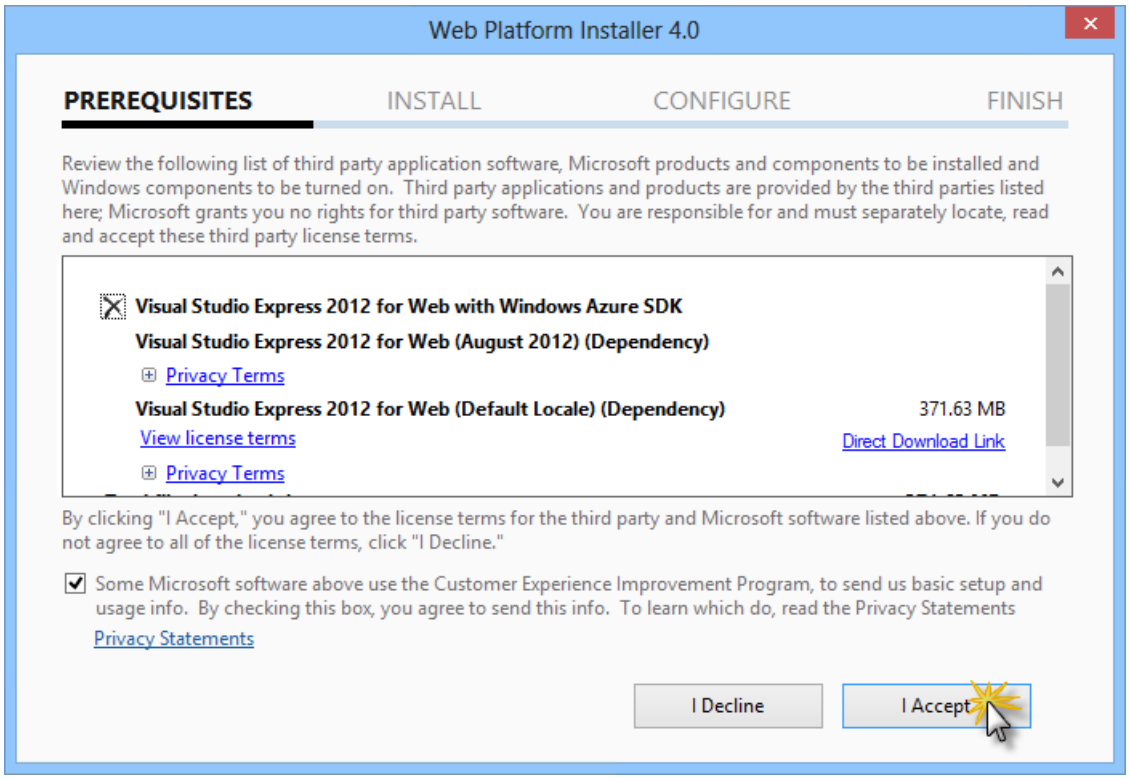
2. Click on **Install Now**. If you do not have **Web Platform Installer** you will be redirected to download and install it first.

3. Once **Web Platform Installer** is open, click **Install** to start the setup.



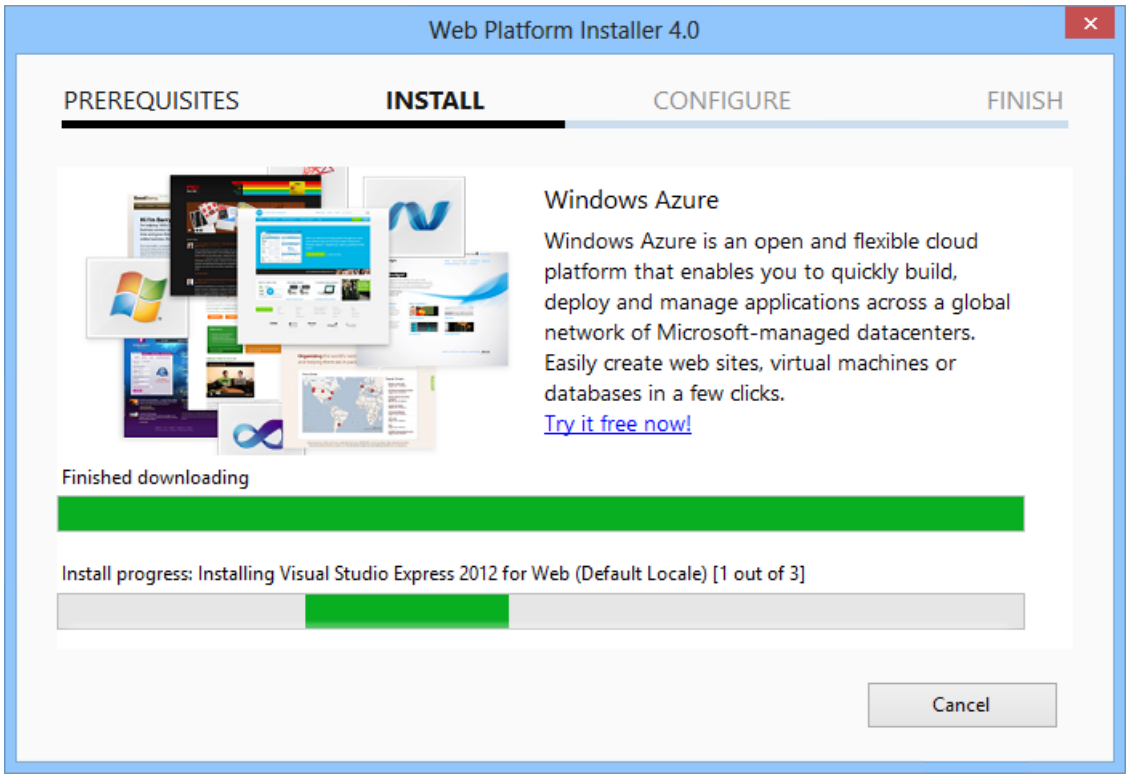
Install Visual Studio Express

4. Read all the products' licenses and terms and click **I Accept** to continue.



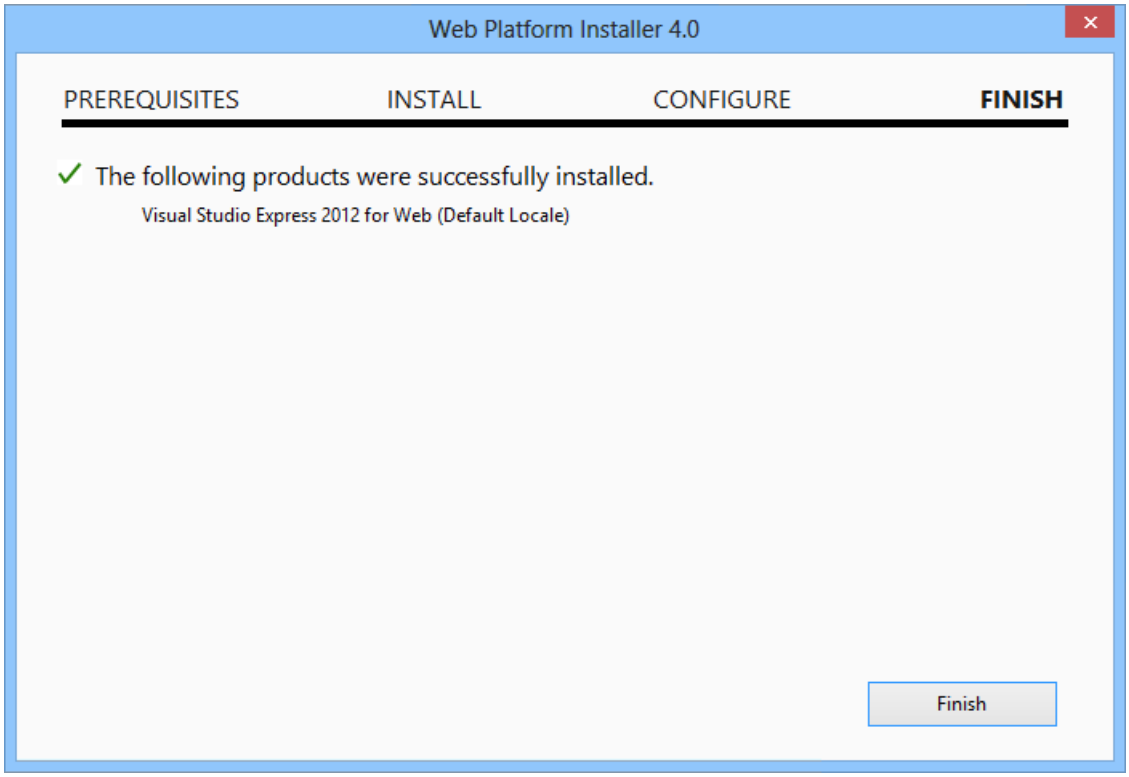
Accepting the license terms

5. Wait until the downloading and installation process completes.



Installation progress

6. When the installation completes, click **Finish**.



Installation completed

7. Click **Exit** to close Web Platform Installer.

8. To open Visual Studio Express for Web, go to the **Start** screen and start writing "**VS Express**", then click on the **VS Express for Web** tile.

VS Express for Web tile

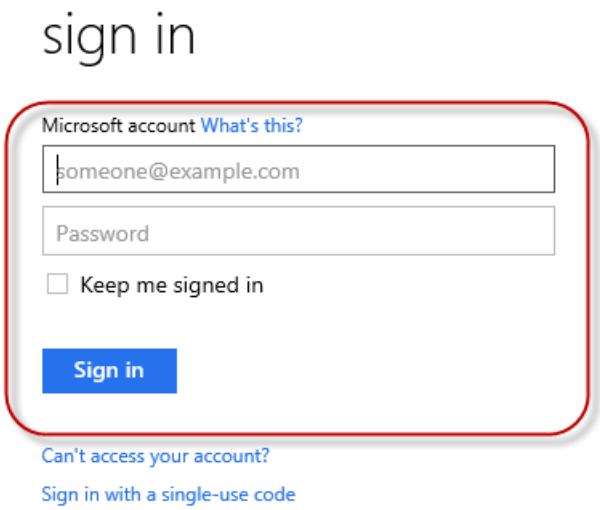
Appendix B: Publishing an ASP.NET MVC 4 Application using Web Deploy

This appendix will show you how to create a new web site from the Windows Azure Management Portal and publish the application you obtained by following the lab, taking advantage of the Web Deploy publishing feature provided by Windows Azure.

Task 1 - Creating a New Web Site from the Windows Azure Portal

1. Go to the Windows Azure Management Portal and sign in using the Microsoft credentials associated with your subscription.

> **NOTE**
> With Windows Azure you can host 10 ASP.NET Web Sites for free and then scale as your traffic grows. You can sign up here.



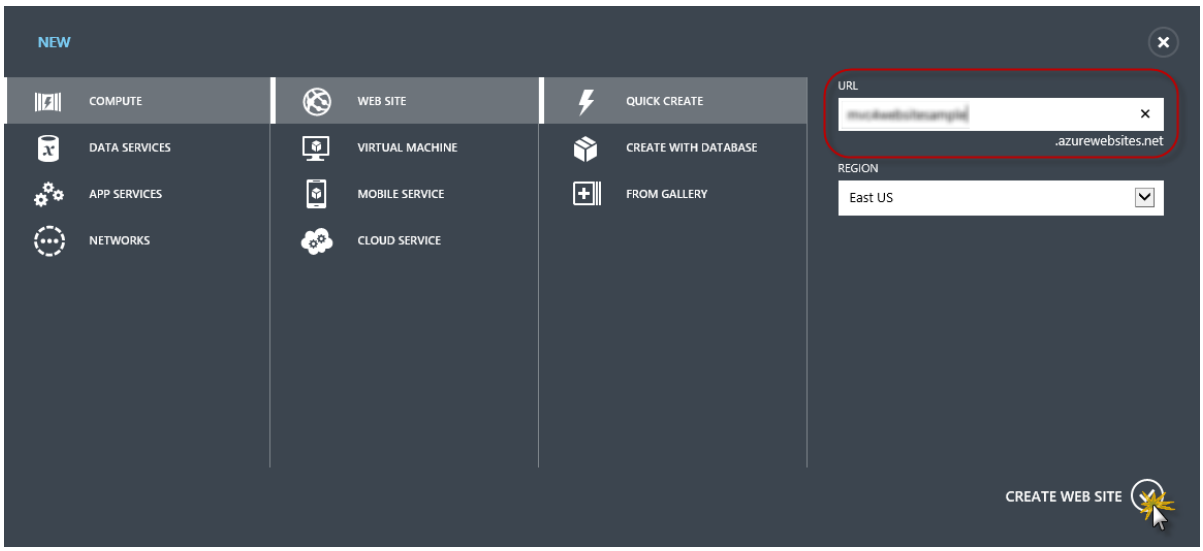
Log on to Windows Azure Management Portal

2. Click **New** on the command bar.

Creating a new Web Site

3. Click **Compute | Web Site**. Then select **Quick Create** option. Provide an available URL for the new web site and click **Create Web Site**.

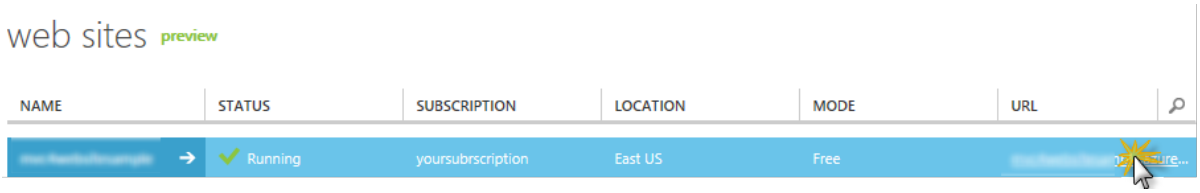
> **NOTE**
>
> A Windows Azure Web Site is the host for a web application running in the cloud that you can control and manage. The Quick Create option allows you to deploy a completed web application to the Windows Azure Web Site from outside the portal. It does not include steps for setting up a database.



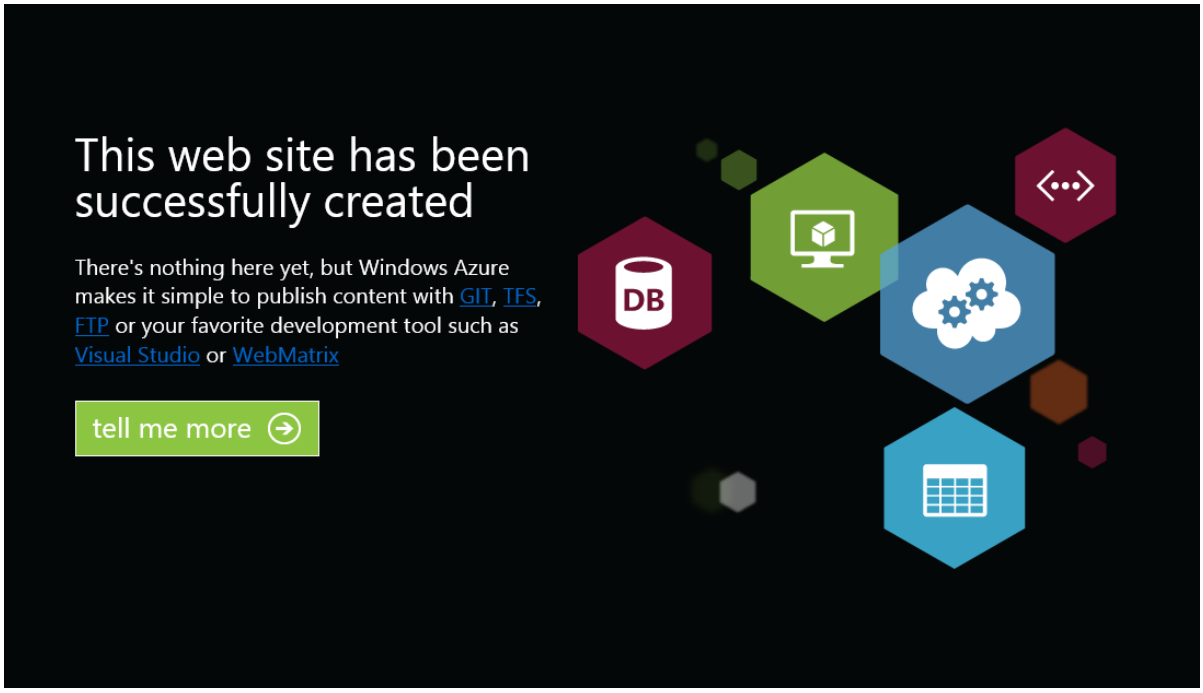
Creating a new Web Site using Quick Create

4. Wait until the new **Web Site** is created.

5. Once the Web Site is created click the link under the **URL** column. Check that the new Web Site is working.

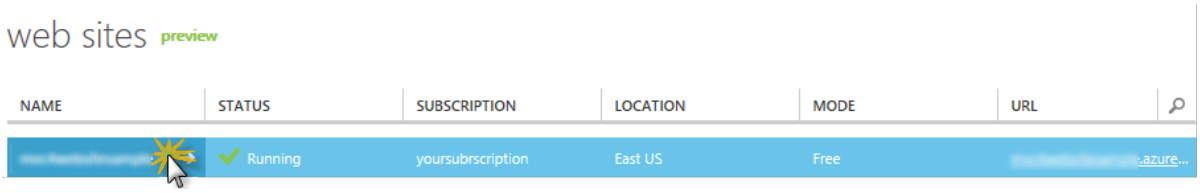


Browsing to the new web site



Web site running

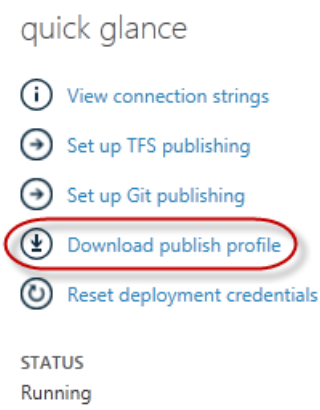
6. Go back to the portal and click the name of the web site under the **Name** column to display the management pages.



Opening the Web Site management pages

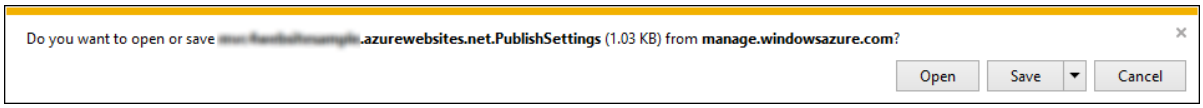
7. In the **Dashboard** page, under the **quick glance** section, click the **Download publish profile** link.

> **NOTE**
>
> The *publish profile* contains all of the information required to publish a web application to a Windows Azure website for each enabled publication method. The publish profile contains the URLs, user credentials and database strings required to connect to and authenticate against each of the endpoints for which a publication method is enabled. **Microsoft WebMatrix 2**, **Microsoft Visual Studio Express for Web** and **Microsoft Visual Studio 2012** support reading publish profiles to automate configuration of these programs for publishing web applications to Windows Azure websites.



Downloading the Web Site publish profile

8. Download the publish profile file to a known location. Further in this exercise you will see how to use this file to publish a web application to a Windows Azure Web Sites from Visual Studio.

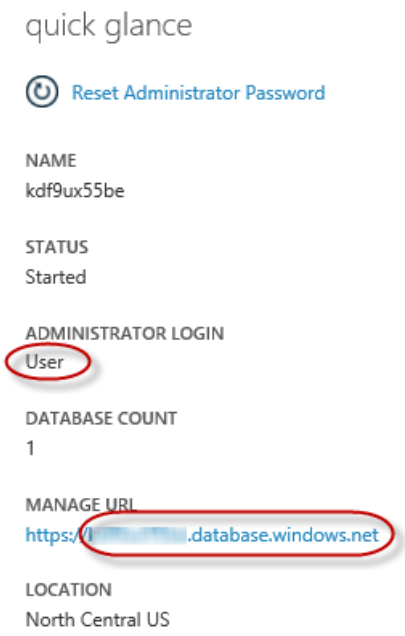


Saving the publish profile file

Task 2 - Configuring the Database Server

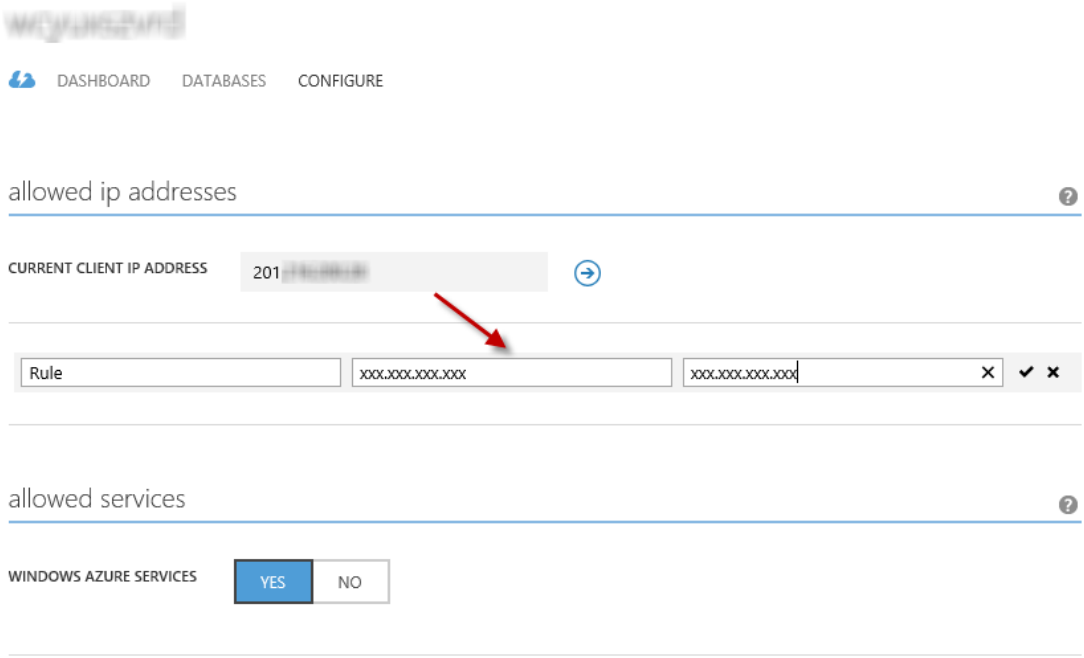
If your application makes use of SQL Server databases you will need to create a SQL Database server. If you want to deploy a simple application that does not use SQL Server you might skip this task.

1. You will need a SQL Database server for storing the application database. You can view the SQL Database servers from your subscription in the Windows Azure Management portal at **Sql Databases | Servers | Server's Dashboard**. If you do not have a server created, you can create one using the **Add** button on the command bar. Take note of the **server name and URL, administrator login name and password**, as you will use them in the next tasks. Do not create the database yet, as it will be created in a later stage.



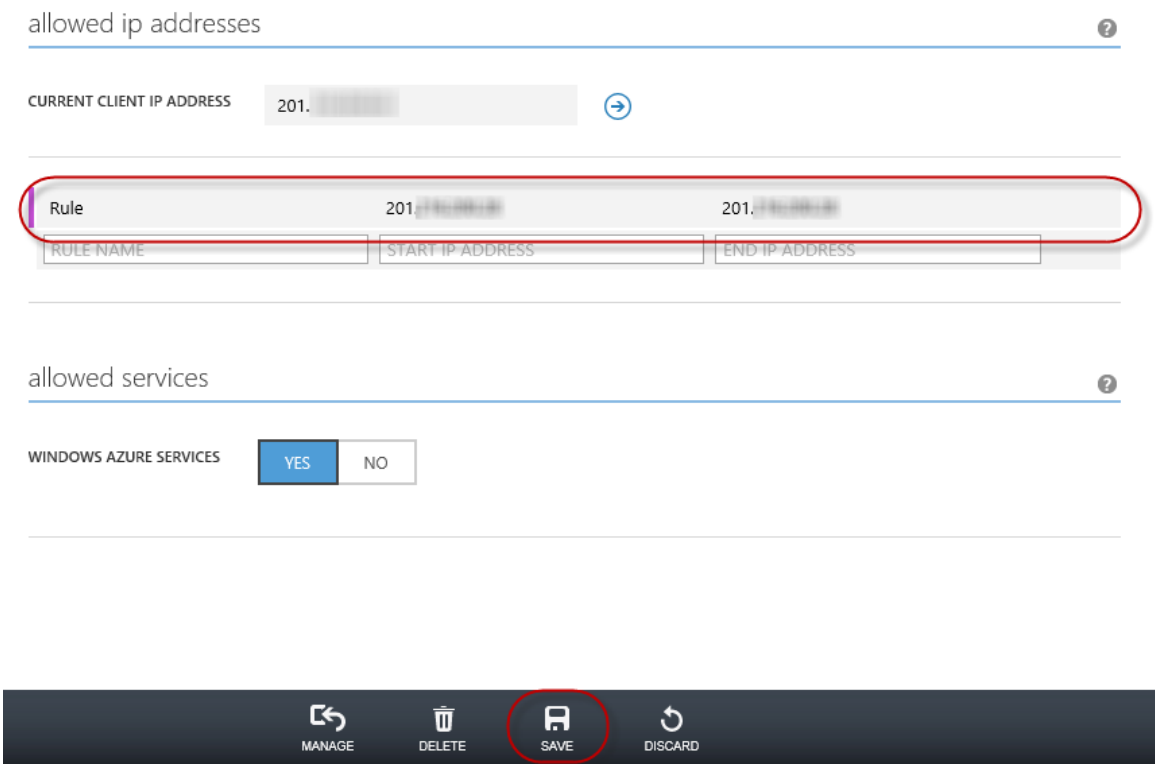
SQL Database Server Dashboard

2. In the next task you will test the database connection from Visual Studio, for that reason you need to include your local IP address in the server's list of **Allowed IP Addresses**. To do that, click **Configure**, select the IP address from **Current Client IP Address** and paste it on the **Start IP Address** and **End IP Address** text boxes and click the ✔ button.



Adding Client IP Address

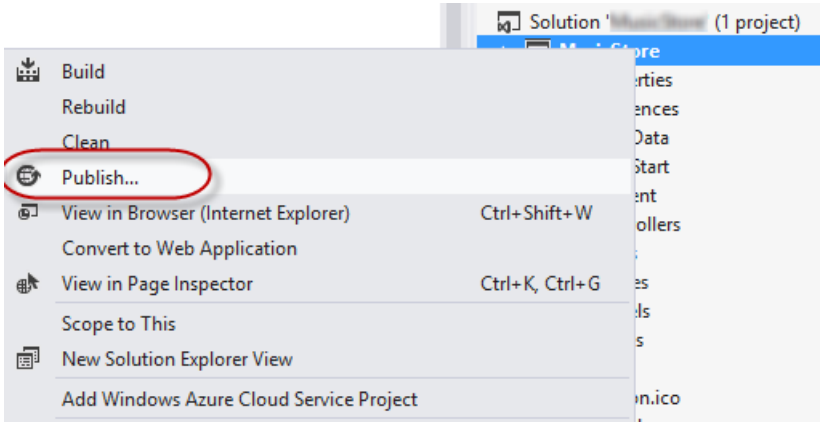
3. Once the **Client IP Address** is added to the allowed IP addresses list, click on **Save** to confirm the changes.



Confirm Changes

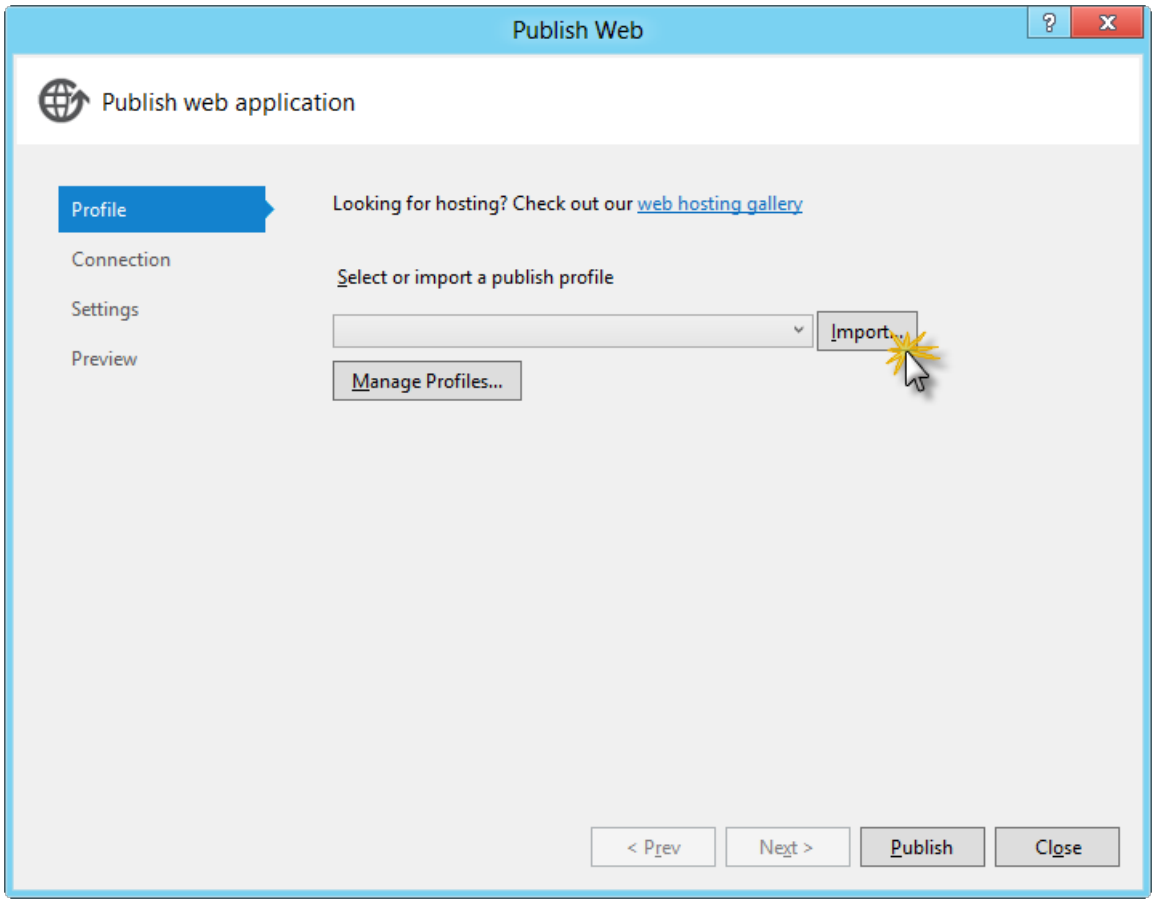
Task 3 - Publishing an ASP.NET MVC 4 Application using Web Deploy

1. Go back to the ASP.NET MVC 4 solution. In the **Solution Explorer**, right-click the web site project and select **Publish**.



Publishing the web site

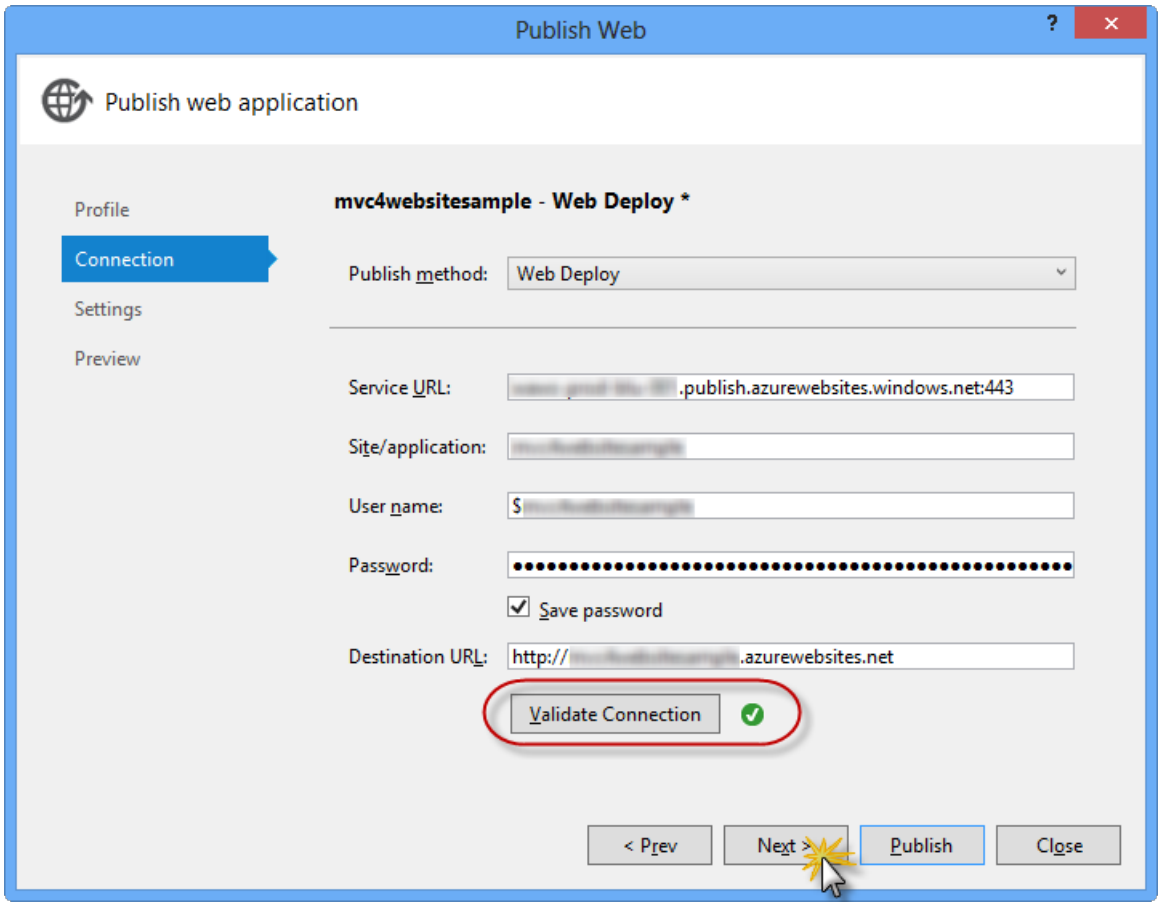
2. Import the publish profile you saved in the first task.



Importing publish profile

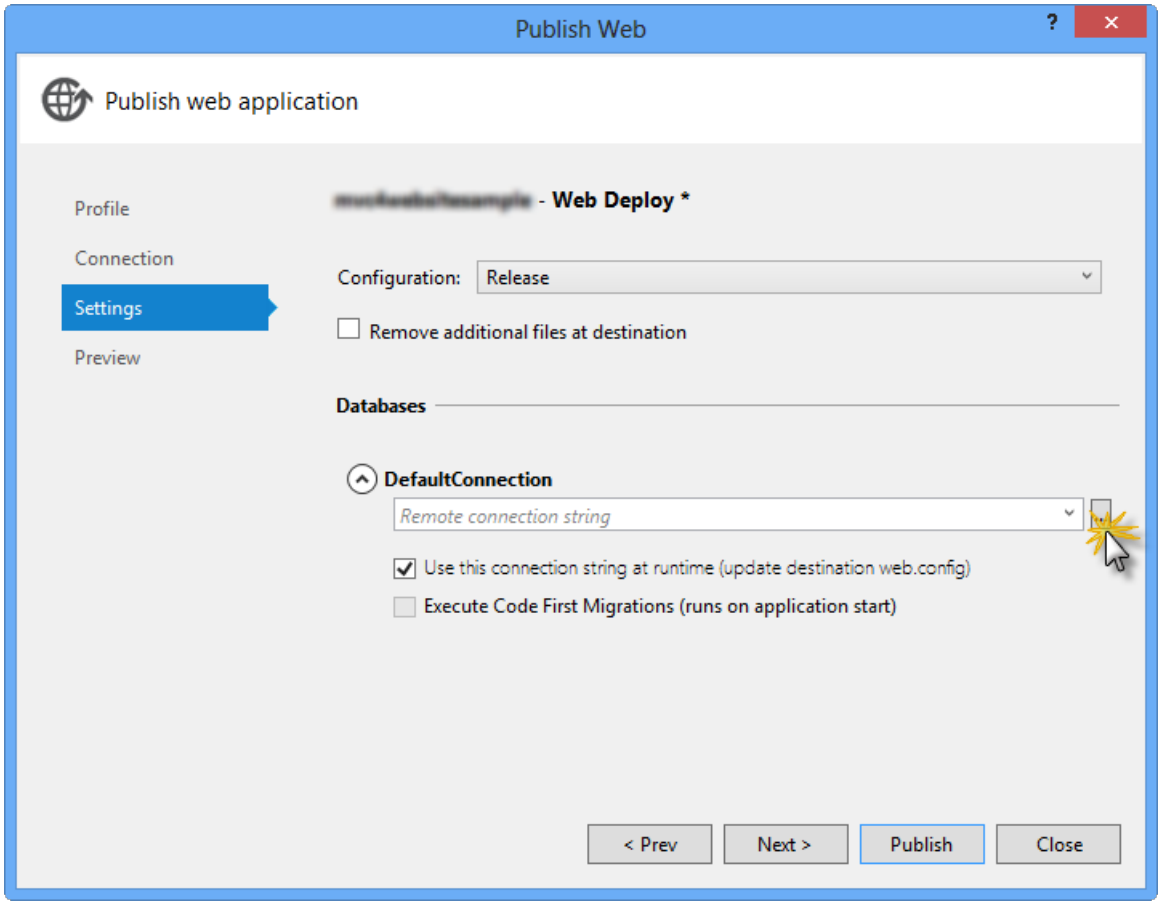
3. Click **Validate Connection**. Once Validation is complete click **Next**.

> **NOTE**
>
> Validation is complete once you see a green checkmark appear next to the Validate Connection button.



Validating connection

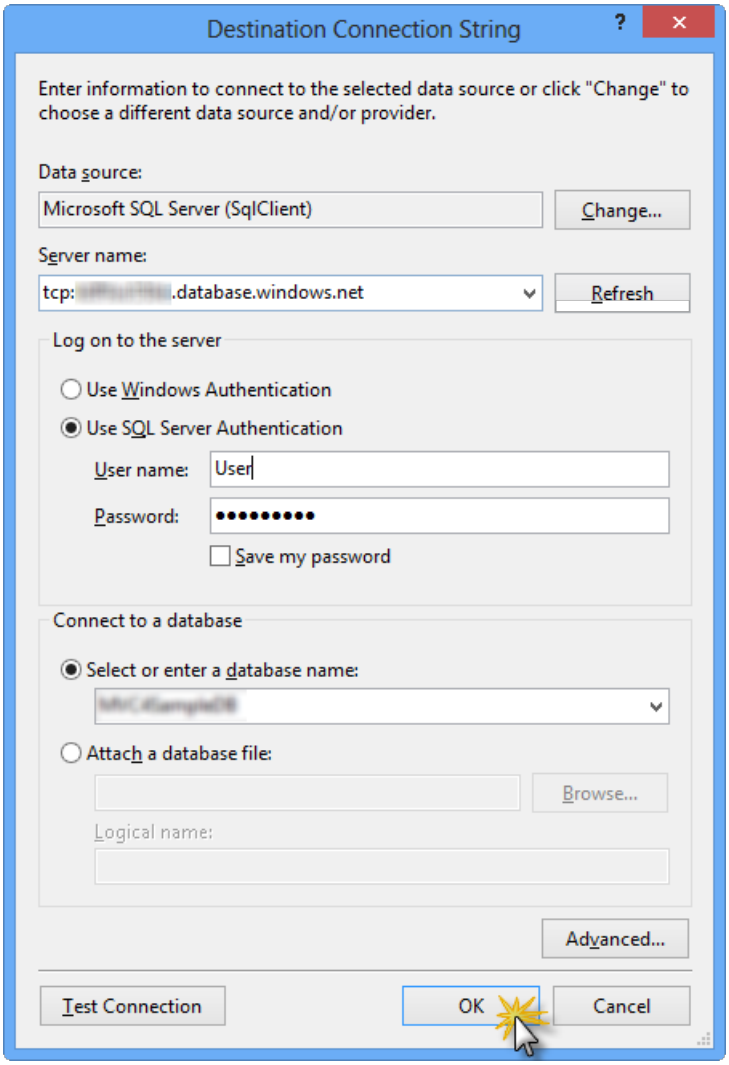
4. In the **Settings** page, under the **Databases** section, click the button next to your database connection's textbox (i.e. **DefaultConnection**).



Web deploy configuration

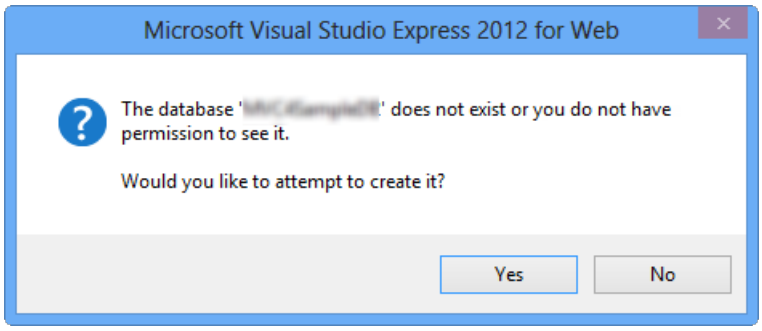
5. Configure the database connection as follows:

   - In the **Server name** type your SQL Database server URL using the *tcp:* prefix.
   - In **User name** type your server administrator login name.
   - In **Password** type your server administrator login password.
   - Type a new database name.



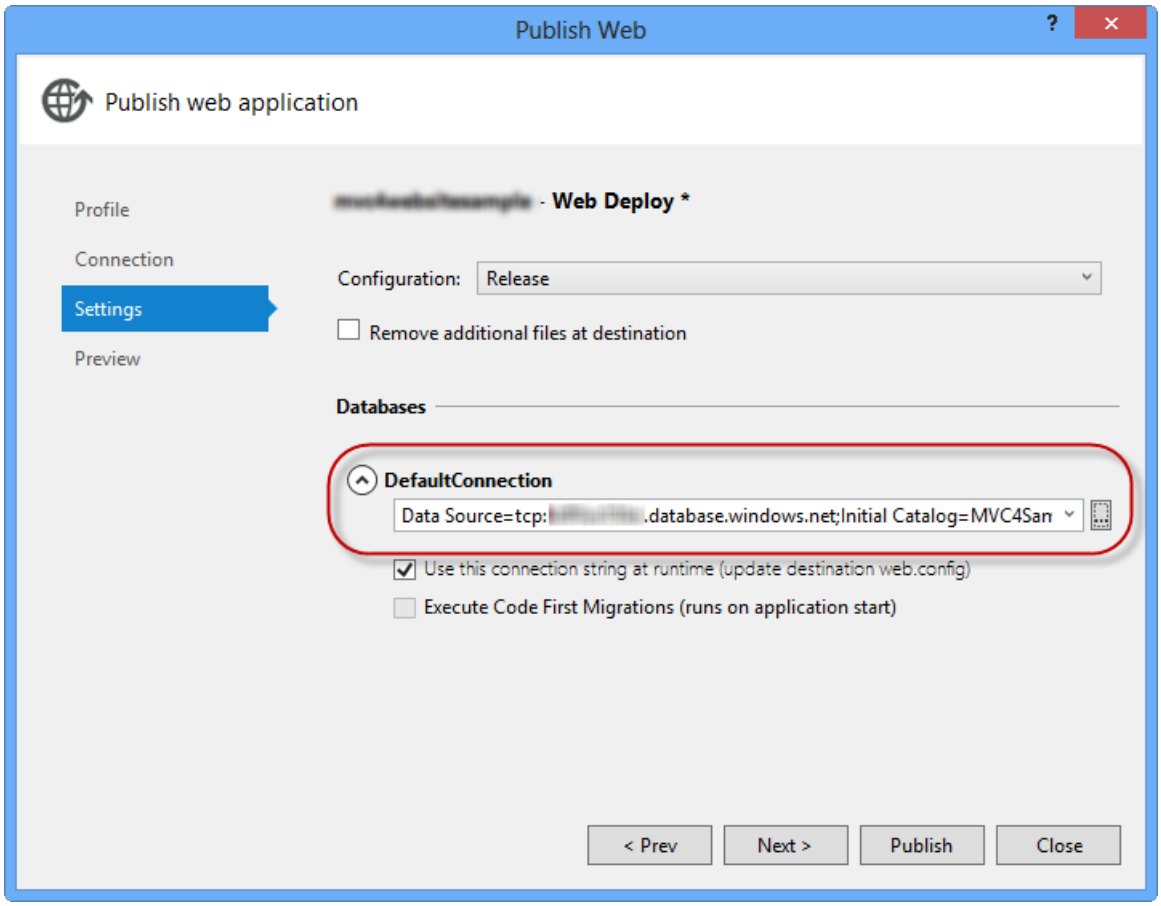
Configuring destination connection string

6. Then click **OK**. When prompted to create the database click **Yes**.



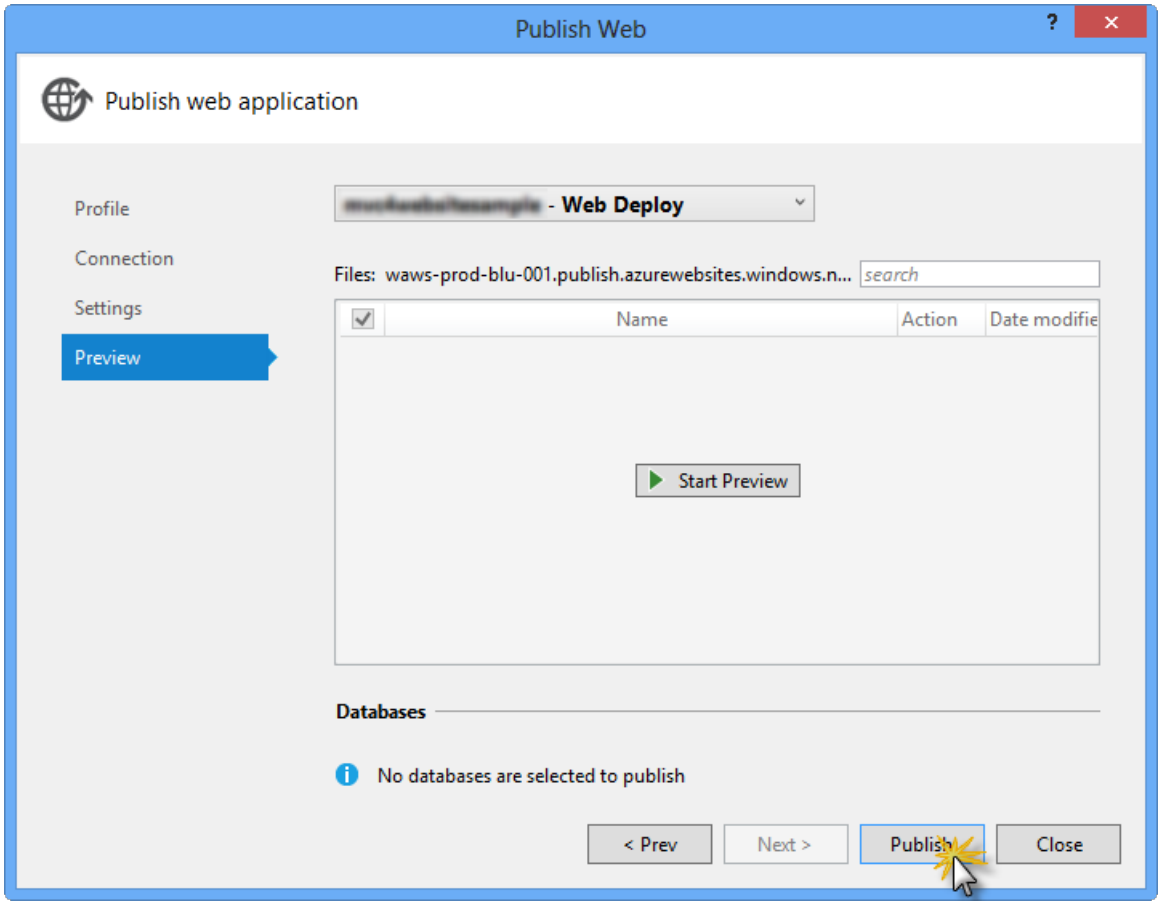
Creating the database

7. The connection string you will use to connect to SQL Database in Windows Azure is shown within Default Connection textbox. Then click **Next**.



Connection string pointing to SQL Database

8. In the **Preview** page, click **Publish**.



Publishing the web application

9. Once the publishing process finishes, your default browser will open the published web site.

Appendix C: Using Code Snippets

With code snippets, you have all the code you need at your fingertips. The lab document will tell you exactly when you can use them, as shown in the following figure.

6. Insert the following code inside the body of the **Run** method.

(Code Snippet – *AzurePlatformTrainingKit - Ex1-Task 1 -Step 5- Run method*)

C#

```
public override void Run()
{
  while (true)
  {
    // retrieve a new message from the queue
    CloudQueueMessage message = queue.GetMessage();
    if (message != null)
    {
      ProcessMessage(message);
      queue.DeleteMessage(msg);
    }
  }
}
```

Using Visual Studio code snippets to insert code into your project

To add a code snippet using the keyboard (C# only)

1. Place the cursor where you would like to insert the code.
2. Start typing the snippet name (without spaces or hyphens).
3. Watch as IntelliSense displays matching snippets' names.
4. Select the correct snippet (or keep typing until the entire snippet's name is selected).
5. Press the Tab key twice to insert the snippet at the cursor location.

```
namespace PhotoGallery.Controllers
{
    public class PhotoController : Controller
    {
        mvc
    }
}
```

IMvcFilter
MVC4LabEx02HomeControllerUsings
MVC4LabEx02IndexAction
MVC4LabEx02PhotoController
MVC4LabEx02Photomodel
MVC4LabEx03iPhoneDisplayMode
MVC4LabEx04Attributes
MVC4LabEx04GetAsync
MVC4LabEx04SendAsyncwithCancellationTok

MVC4 Lab - Ex02 - PhotoController

Start typing the snippet name

```
static void Main(string[] args)
{
    AzureTrainingKitServiceBus
}
```

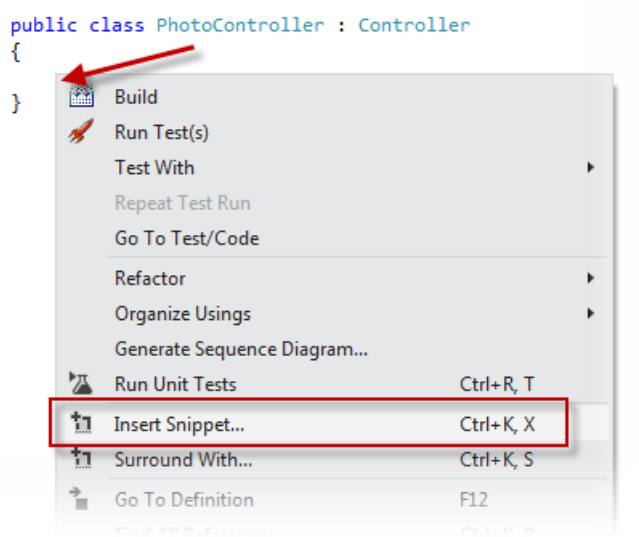
Press Tab to select the highlighted snippet

```
static void Main(string[] args)
{
    request.Headers.Add("$accountKey", accountKey);
    request.Headers.Add("$uniqueUserID", uniqueUserId);
}
```

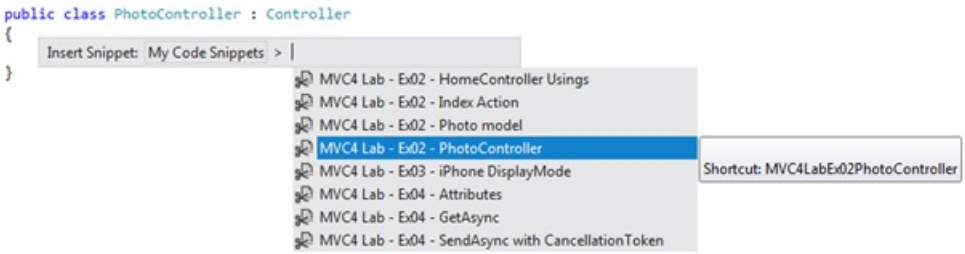
Press Tab again and the snippet will expand

To add a code snippet using the mouse (C#, Visual Basic and XML) 1. Right-click where you want to insert the code snippet.

1. Select **Insert Snippet** followed by **My Code Snippets**.
2. Pick the relevant snippet from the list, by clicking on it.



Right-click where you want to insert the code snippet and select Insert Snippet



Pick the relevant snippet from the list, by clicking on it

ASP.NET MVC 4 Custom Action Filters

2/20/2020 • 20 minutes to read • Edit Online

By Web Camps Team

Download Web Camps Training Kit

ASP.NET MVC provides Action Filters for executing filtering logic either before or after an action method is called. Action Filters are custom attributes that provide declarative means to add pre-action and post-action behavior to the controller's action methods.

In this Hands-on Lab you will create a custom action filter attribute into MvcMusicStore solution to catch controller's requests and log the activity of a site into a database table. You will be able to add your logging filter by injection to any controller or action. Finally, you will see the log view that shows the list of visitors.

This Hands-on Lab assumes you have basic knowledge of **ASP.NET MVC**. If you have not used **ASP.NET MVC** before, we recommend you to go over **ASP.NET MVC 4 Fundamentals** Hands-on Lab.

> **NOTE**
>
> All sample code and snippets are included in the Web Camps Training Kit, available from at Microsoft-Web/WebCampTrainingKit Releases. The project specific to this lab is available at ASP.NET MVC 4 Custom Action Filters.

Objectives

In this Hands-On Lab, you will learn how to:

- Create a custom action filter attribute to extend filtering capabilities
- Apply a custom filter attribute by injection to a specific level
- Register a custom action filters globally

Prerequisites

You must have the following items to complete this lab:

- Microsoft Visual Studio Express 2012 for Web or superior (read Appendix A for instructions on how to install it).

Setup

Installing Code Snippets

For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. To install the code snippets run **.\Source\Setup\CodeSnippets.vsi** file.

If you are not familiar with the Visual Studio Code Snippets, and want to learn how to use them, you can refer to the appendix from this document "Appendix C: Using Code Snippets".

Exercises

This Hands-On Lab is comprised by the following exercises:

1. Exercise 1: Logging actions
2. Exercise 2: Managing Multiple Action Filters

Estimated time to complete this lab: **30 minutes**.

> **NOTE**
>
> Each exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

Exercise 1: Logging Actions

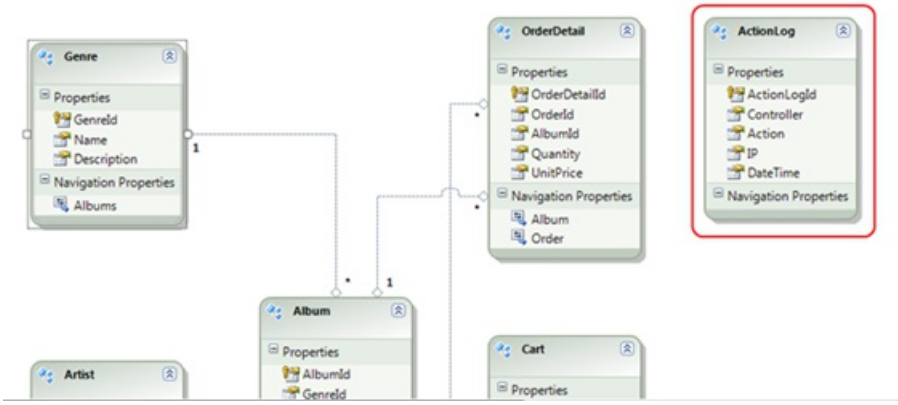
In this exercise, you will learn how to create a custom action log filter by using ASP.NET MVC 4 Filter Providers. For that purpose you will apply a logging filter to the MusicStore site that will record all the activities in the selected controllers.

The filter will extend **ActionFilterAttributeClass** and override **OnActionExecuting** method to catch each request and then perform the logging actions. The context information about HTTP requests, executing methods, results and parameters will be provided by ASP.NET MVC **ActionExecutingContext** class .

> **NOTE**
>
> ASP.NET MVC 4 also has default filters providers you can use without creating a custom filter. ASP.NET MVC 4 provides the following types of filters:
>
> - **Authorization** filter, which makes security decisions about whether to execute an action method, such as performing authentication or validating properties of the request.
> - **Action** filter, which wraps the action method execution. This filter can perform additional processing, such as providing extra data to the action method, inspecting the return value, or canceling execution of the action method
> - **Result** filter, which wraps execution of the ActionResult object. This filter can perform additional processing of the result, such as modifying the HTTP response.
> - **Exception** filter, which executes if there is an unhandled exception thrown somewhere in action method, starting with the authorization filters and ending with the execution of the result. Exception filters can be used for tasks such as logging or displaying an error page.
>
> For more information about Filters Providers please visit this MSDN link: (https://msdn.microsoft.com/library/dd410209.aspx) .

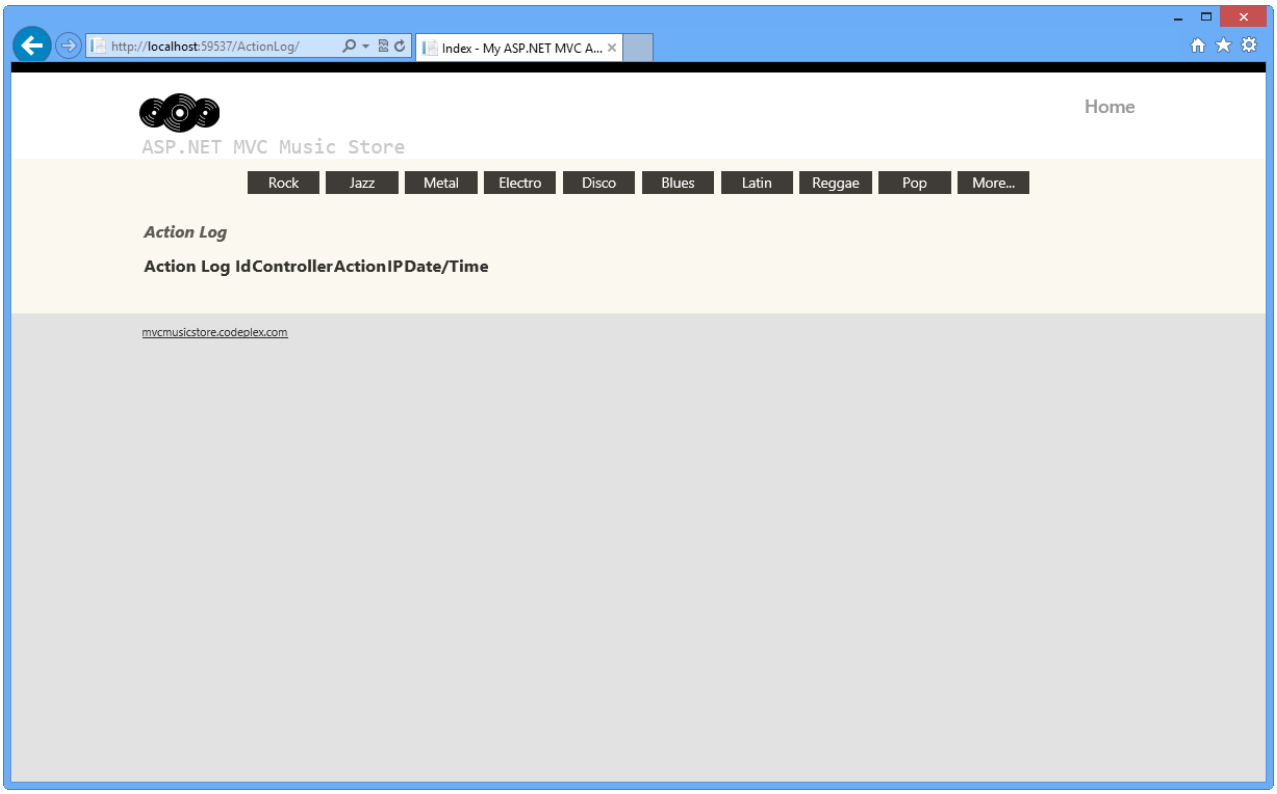
About MVC Music Store Application logging feature

This Music Store solution has a new data model table for site logging, **ActionLog**, with the following fields: Name of the controller that received a request, Called action, Client IP and Time stamp.



Data model - ActionLog table

The solution provides an ASP.NET MVC View for the Action log that can be found at **MvcMusicStores/Views/ActionLog**:



Action Log view

With this given structure, all the work will be focused on interrupting controller's request and performing the logging by using custom filtering.

Task 1 - Creating a Custom Filter to Catch a Controller's Request

In this task you will create a custom filter attribute class that will contain the logging logic. For that purpose you will extend ASP.NET MVC **ActionFilterAttribute** Class and implement the interface **IActionFilter**.

> **NOTE**
>
> The **ActionFilterAttribute** is the base class for all the attribute filters. It provides the following methods to execute a specific logic after and before controller action's execution:
>
> - **OnActionExecuting**(ActionExecutingContext filterContext): Just before the action method is called.
> - **OnActionExecuted**(ActionExecutedContext filterContext): After the action method is called and before the result is executed (before view render).
> - **OnResultExecuting**(ResultExecutingContext filterContext): Just before the result is executed (before view render).
> - **OnResultExecuted**(ResultExecutedContext filterContext): After the result is executed (after the view is rendered).
>
> By overriding any of these methods into a derived class, you can execute your own filtering code.

1. Open the **Begin** solution located at **\Source\Ex01-LoggingActions\Begin** folder.

   a. You will need to download some missing NuGet packages before you continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

> **NOTE**
>
> One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.
>
> For more information, see this article: http://docs.nuget.org/docs/workflows/using-nuget-without-committing-packages.

2. Add a new C# class into the **Filters** folder and name it *CustomActionFilter.cs*. This folder will store all the custom filters.

3. Open **CustomActionFilter.cs** and add a reference to **System.Web.Mvc** and **MvcMusicStore.Models** namespaces:

   (Code Snippet - *ASP.NET MVC 4 Custom Action Filters - Ex1-CustomActionFilterNamespaces*)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MvcMusicStore.Models;
```

4. Inherit the **CustomActionFilter** class from **ActionFilterAttribute** and then make **CustomActionFilter** class implement **IActionFilter** interface.

```
//...
namespace MvcMusicStore.Filters
{
    public class CustomActionFilter : ActionFilterAttribute, IActionFilter
    {
        //...
    }
}
```

5. Make **CustomActionFilter** class override the method **OnActionExecuting** and add the necessary logic to log the filter's execution. To do this, add the following highlighted code within **CustomActionFilter** class.

   (Code Snippet - *ASP.NET MVC 4 Custom Action Filters - Ex1-LoggingActions*)

```
public class CustomActionFilter : ActionFilterAttribute, IActionFilter
{
    void IActionFilter.OnActionExecuting(ActionExecutingContext filterContext)
    {
        // TODO: Add your action filter's tasks here

        // Log Action Filter call
        using (MusicStoreEntities storeDb = new MusicStoreEntities())
        {
            ActionLog log = new ActionLog()
            {
                Controller = filterContext.ActionDescriptor.ControllerDescriptor.ControllerName,
                Action = string.Concat(filterContext.ActionDescriptor.ActionName, " (Logged By: Custom
Action Filter)"),
                IP = filterContext.HttpContext.Request.UserHostAddress,
                DateTime = filterContext.HttpContext.Timestamp
            };
            storeDb.ActionLogs.Add(log);
            storeDb.SaveChanges();
            OnActionExecuting(filterContext);
        }
    }
}
```

NOTE

OnActionExecuting method is using **Entity Framework** to add a new ActionLog register. It creates and fills a new entity instance with the context information from **filterContext**.

You can read more about **ControllerContext** class at msdn.

Task 2 - Injecting a Code Interceptor into the Store Controller Class

In this task you will add the custom filter by injecting it to all controller classes and controller actions that will be logged. For the purpose of this exercise, the Store Controller class will have a log.

The method **OnActionExecuting** from **ActionLogFilterAttribute** custom filter runs when an injected element is called.

It is also possible to intercept a specific controller method.

1. Open the **StoreController** at **MvcMusicStore\Controllers** and add a reference to the **Filters** namespace:

```
using System.Linq;
using System.Web.Mvc;
using MvcMusicStore.Models;
using MvcMusicStore.Filters;
```

2. Inject the custom filter **CustomActionFilter** into **StoreController** class by adding **[CustomActionFilter]** attribute before the class declaration.

```
...
[CustomActionFilter]
public class StoreController : Controller
{
    ...
}
```

NOTE

When a filter is injected into a controller class, all its actions are also injected. If you would like to apply the filter only for a set of actions, you would have to inject **[CustomActionFilter]** to each one of them:

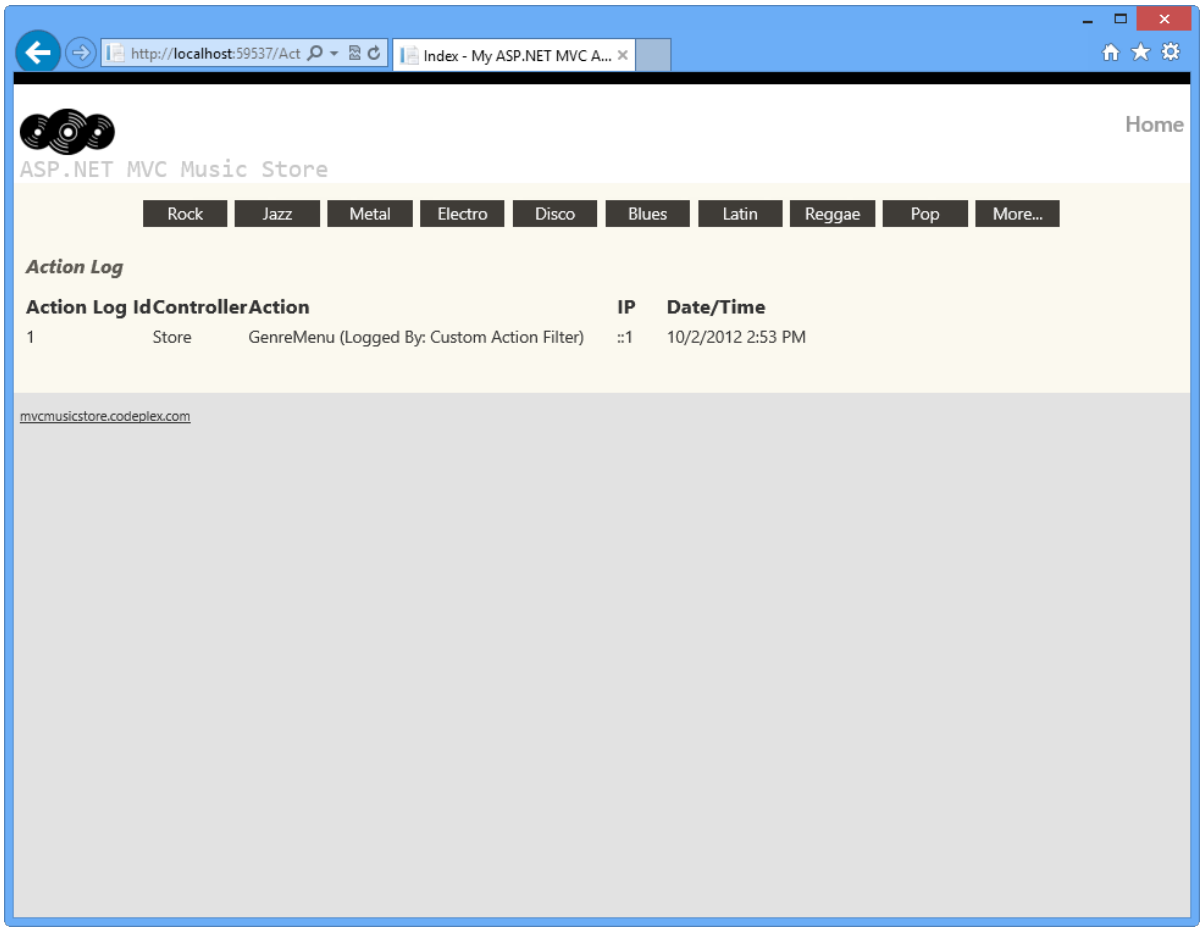
```
[CustomActionFilter]
public ActionResult Index()
{
  ...
}

[CustomActionFilter]
public ActionResult Browse(string genre)
{
  ...
}
```

Task 3 - Running the Application

In this task, you will test that the logging filter is working. You will start the application and visit the store, and then you will check logged activities.

1. Press **F5** to run the application.
2. Browse to **/ActionLog** to see log view initial state:



Log tracker status before page activity

NOTE

By default, it will always show one item that is generated when retrieving the existing genres for the menu.

For simplicity purposes we're cleaning up the **ActionLog** table each time the application runs so it will only show the logs of each particular task's verification.

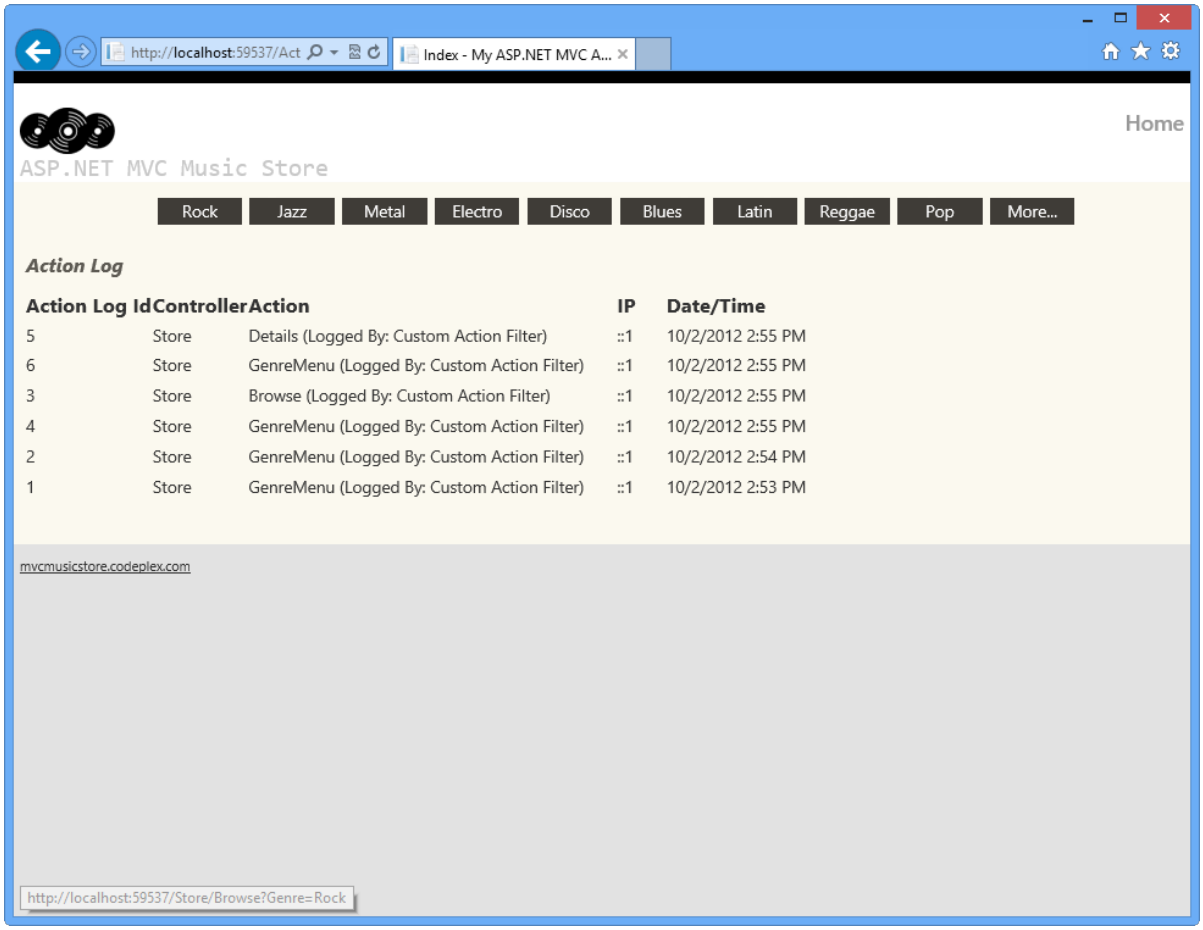
You might need to remove the following code from the **Session_Start** method (in the **Global.asax** class), in order to save an historical log for all the actions executed within the Store Controller.

```
// Clean up Logs Table
MusicStoreEntities storeDB = new MusicStoreEntities();
foreach (var log in

storeDB.ActionLogs.ToList())
{
   storeDB.ActionLogs.Remove(log);
}

storeDB.SaveChanges();
```

3. Click one of the **Genres** from the menu and perform some actions there, like browsing an available album.
4. Browse to **/ActionLog** and if the log is empty press **F5** to refresh the page. Check that your visits were tracked:



Action log with activity logged

Exercise 2: Managing Multiple Action Filters

In this exercise you will add a second Custom Action Filter to the StoreController class and define the specific order in which both filters will be executed. Then you will update the code to register the filter Globally.

There are different options to take into account when defining the Filters' execution order. For example, the Order

property and the Filters' scope:

You can define a **Scope** for each of the Filters, for example, you could scope all the Action Filters to run within the **Controller Scope**, and all Authorization Filters to run in **Global scope**. The scopes have a defined execution order.

Additionally, each action filter has an Order property which is used to determine the execution order in the scope of the filter.

For more information about Custom Action Filters execution order, please visit this MSDN article: (https://msdn.microsoft.com/library/dd381609(v=vs.98).aspx).

Task 1: Creating a new Custom Action Filter

In this task, you will create a new Custom Action Filter to inject into the StoreController class, learning how to manage the execution order of the filters.

1. Open the **Begin** solution located at **\Source\Ex02-ManagingMultipleActionFilters\Begin** folder. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.

   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   >
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.
   >
   > For more information, see this article: http://docs.nuget.org/docs/workflows/using-nuget-without-committing-packages.

2. Add a new C# class into the **Filters** folder and name it *MyNewCustomActionFilter.cs*

3. Open **MyNewCustomActionFilter.cs** and add a reference to **System.Web.Mvc** and the **MvcMusicStore.Models** namespace:

   (Code Snippet - *ASP.NET MVC 4 Custom Action Filters - Ex2-MyNewCustomActionFilterNamespaces*)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MvcMusicStore.Models;
```

4. Replace the default class declaration with the following code.

   (Code Snippet - *ASP.NET MVC 4 Custom Action Filters - Ex2-MyNewCustomActionFilterClass*)

```
public class MyNewCustomActionFilter : ActionFilterAttribute, IActionFilter
{
  void IActionFilter.OnActionExecuting(ActionExecutingContext filterContext)

  {
        // TODO: Add your acction filter's tasks here

        // Log Action Filter Call
        MusicStoreEntities storeDB = new MusicStoreEntities();

        ActionLog log = new ActionLog()
        {
             Controller = filterContext.ActionDescriptor.ControllerDescriptor.ControllerName,
             Action = filterContext.ActionDescriptor.ActionName + " (Logged By:

MyNewCustomActionFilter)",
             IP = filterContext.HttpContext.Request.UserHostAddress,
             DateTime = filterContext.HttpContext.Timestamp
        };

        storeDB.ActionLogs.Add(log);
        storeDB.SaveChanges();

        this.OnActionExecuting(filterContext);
  }
}
```

NOTE

This Custom Action Filter is almost the same than the one you created in the previous exercise. The main difference is that it has the *"Logged By"* attribute updated with this new class' name to identify which filter registered the log.

Task 2: Injecting a new Code Interceptor into the StoreController Class

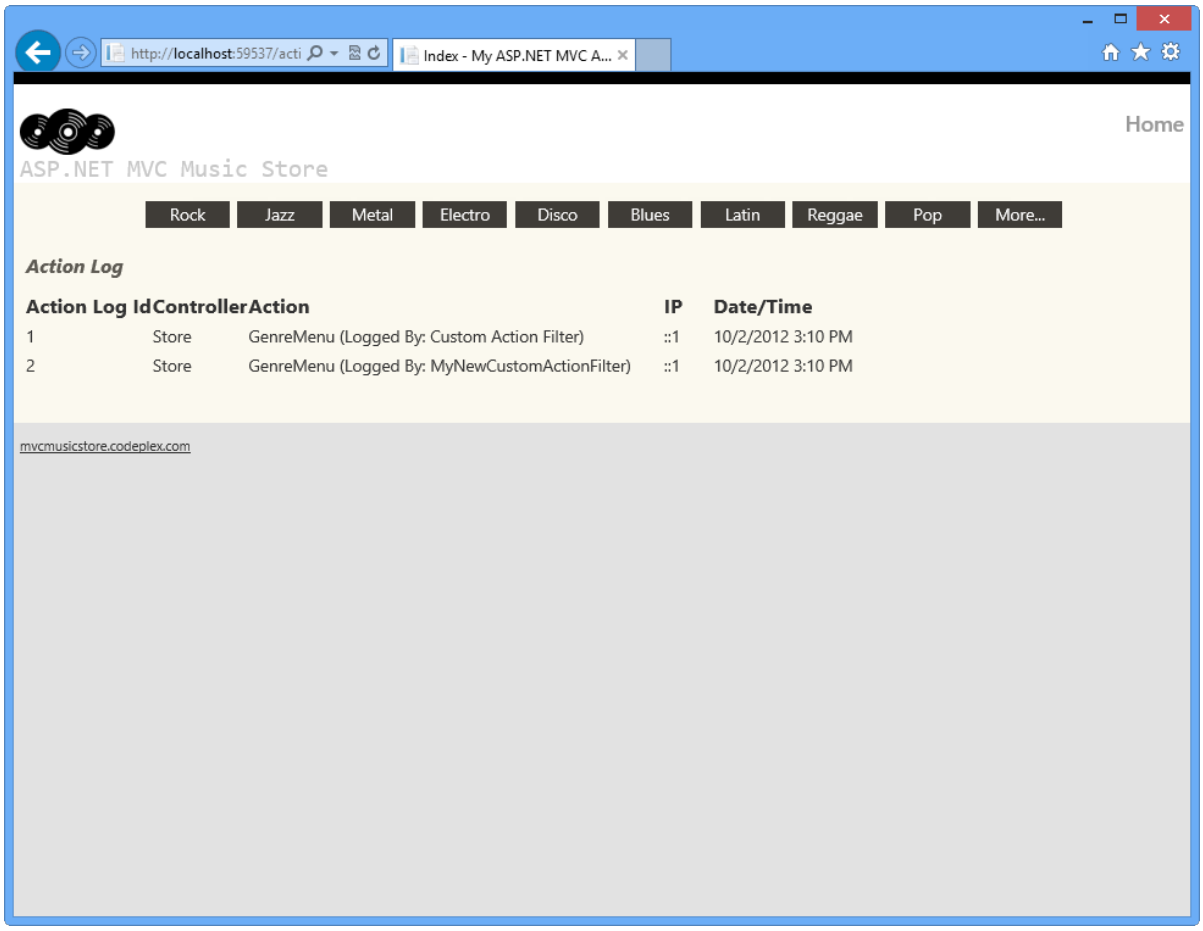
In this task, you will add a new custom filter into the StoreController Class and run the solution to verify how both filters work together.

1. Open the **StoreController** class located at **MvcMusicStore\Controllers** and inject the new custom filter **MyNewCustomActionFilter** into **StoreController** class like is shown in the following code.

```
[MyNewCustomActionFilter]
[CustomActionFilter]
public class StoreController : Controller
{
...
}
```

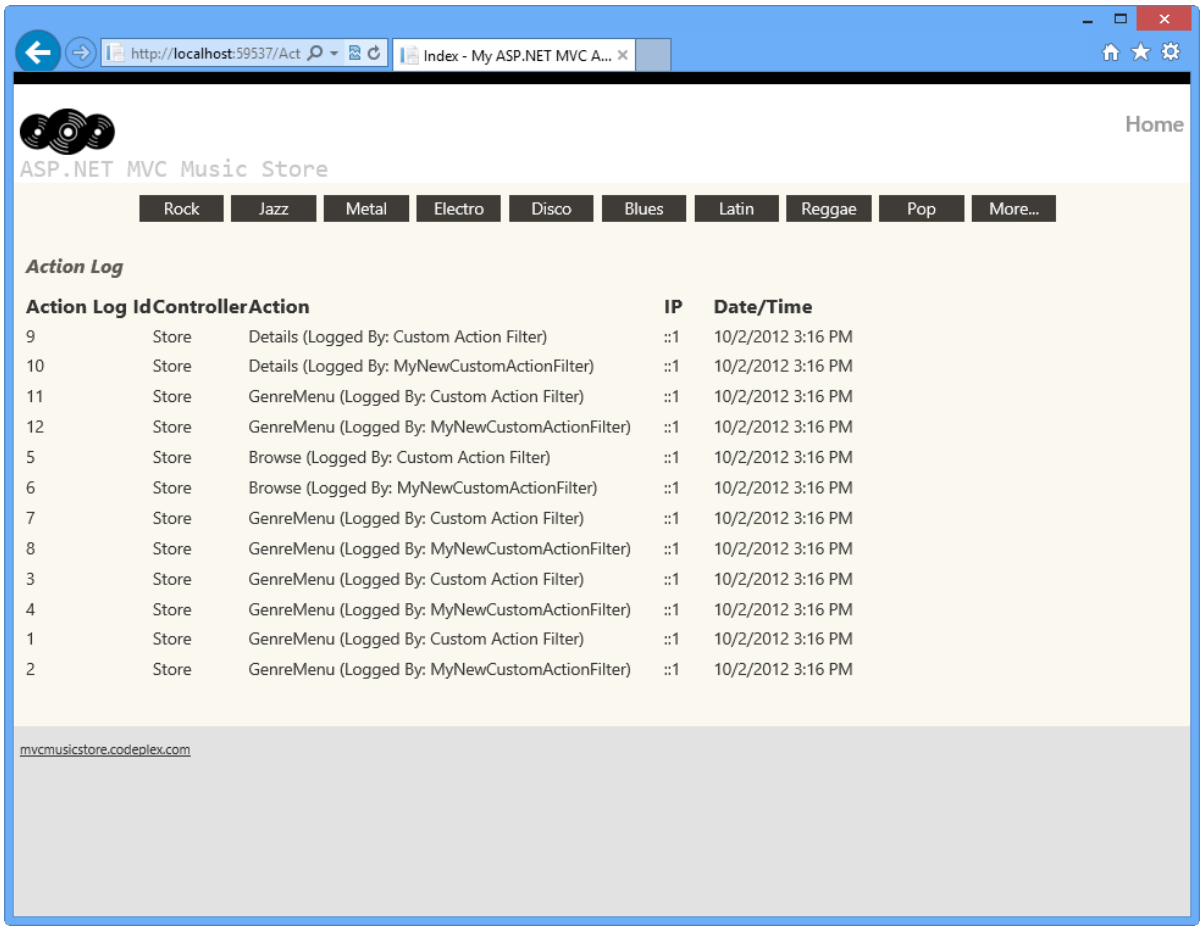
2. Now, run the application in order to see how these two Custom Action Filters work. To do this, press **F5** and wait until the application starts.

3. Browse to **/ActionLog** to see log view initial state.



Log tracker status before page activity

4. Click one of the **Genres** from the menu and perform some actions there, like browsing an available album.
5. Check that this time; your visits were tracked twice: once for each of the Custom Action Filters you added in the **StorageController** class.



Action log with activity logged

6. Close the browser.

Task 3: Managing Filter Ordering

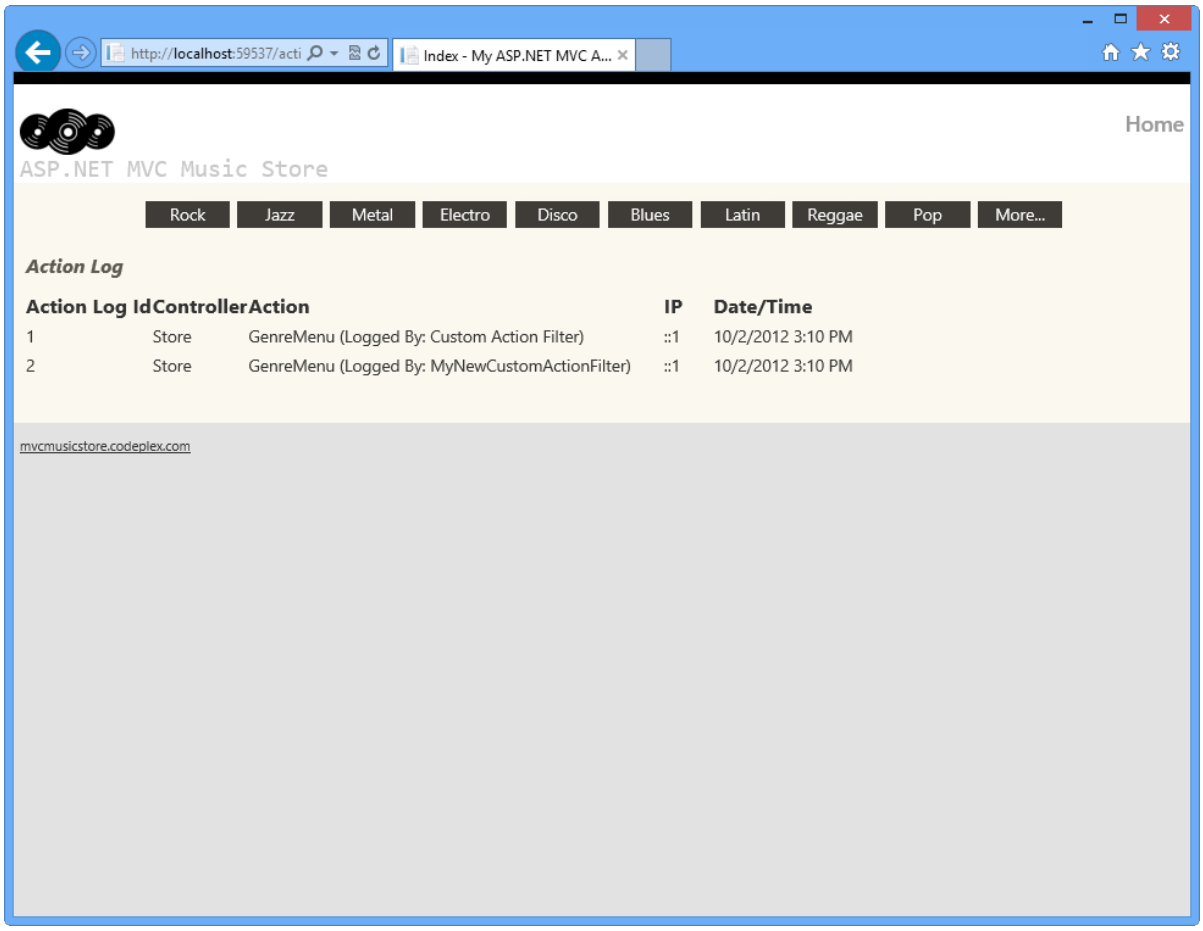
In this task, you will learn how to manage the filters' execution order by using the Order property.

1. Open the **StoreController** class located at **MvcMusicStore\Controllers** and specify the **Order** property in both filters like shown below.

```
[MyNewCustomActionFilter(Order = 2)]
[CustomActionFilter(Order = 1)]
public class StoreController : Controller
{
...
}
```

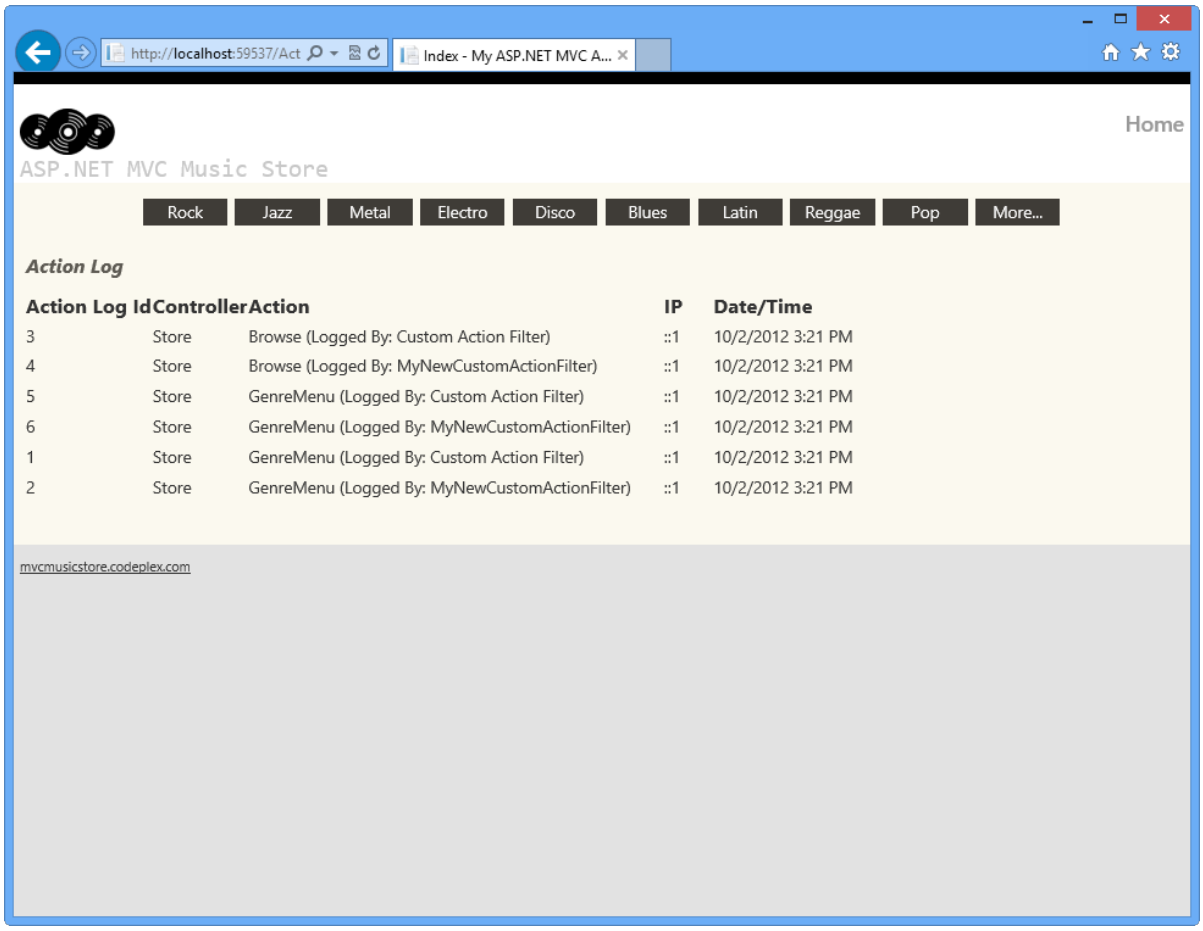
2. Now, verify how the filters are executed depending on its Order property's value. You will find that the filter with the smallest Order value (**CustomActionFilter**) is the first one that is executed. Press **F5** and wait until the application starts.

3. Browse to **/ActionLog** to see log view initial state.



Log tracker status before page activity

4. Click one of the **Genres** from the menu and perform some actions there, like browsing an available album.
5. Check that this time, your visits were tracked ordered by the filters' Order value: **CustomActionFilter** logs' first.



Action log with activity logged

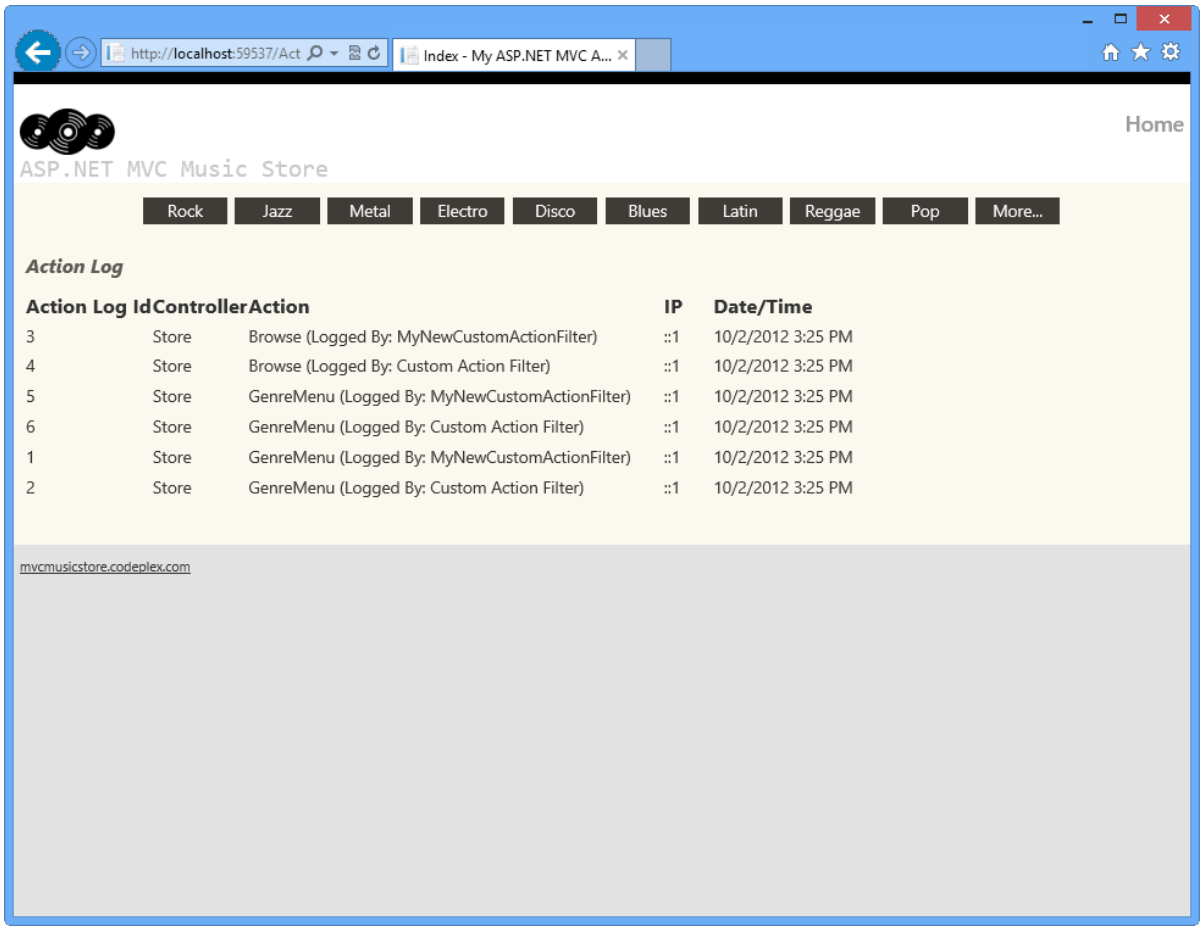
6. Now, you will update the Filters' order value and verify how the logging order changes. In the **StoreController** class, update the Filters' Order value like shown below.

```
[MyNewCustomActionFilter(Order = 1)]
[CustomActionFilter(Order = 2)]
public class StoreController : Controller
{
...
}
```

7. Run the application again by pressing **F5**.

8. Click one of the **Genres** from the menu and perform some actions there, like browsing an available album.

9. Check that this time, the logs created by **MyNewCustomActionFilter** filter appears first.



Action log with activity logged

Task 4: Registering Filters Globally

In this task, you will update the solution to register the new filter (**MyNewCustomActionFilter**) as a global filter. By doing this, it will be triggered by all the actions performed in the application and not only in the StoreController ones as in the previous task.

1. In **StoreController** class, remove **[MyNewCustomActionFilter]** attribute and the order property from **[CustomActionFilter]**. It should look like the following:

```
[CustomActionFilter]
public class StoreController : Controller
{
...
}
```

2. Open **Global.asax** file and locate the **Application_Start** method. Notice that each time the application starts it is registering the global filters by calling **RegisterGlobalFilters** method within **FilterConfig** class.

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();

    WebApiConfig.Register(GlobalConfiguration.Configuration);
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);

    AppConfig.Configure();
}
```

Registering Global Filters in Global.asax

3. Open **FilterConfig.cs** file within **App_Start** folder.

4. Add a reference to using System.Web.Mvc; using MvcMusicStore.Filters; namespace.

```
using System.Web.Mvc;
using MvcMusicStore.Filters;
```

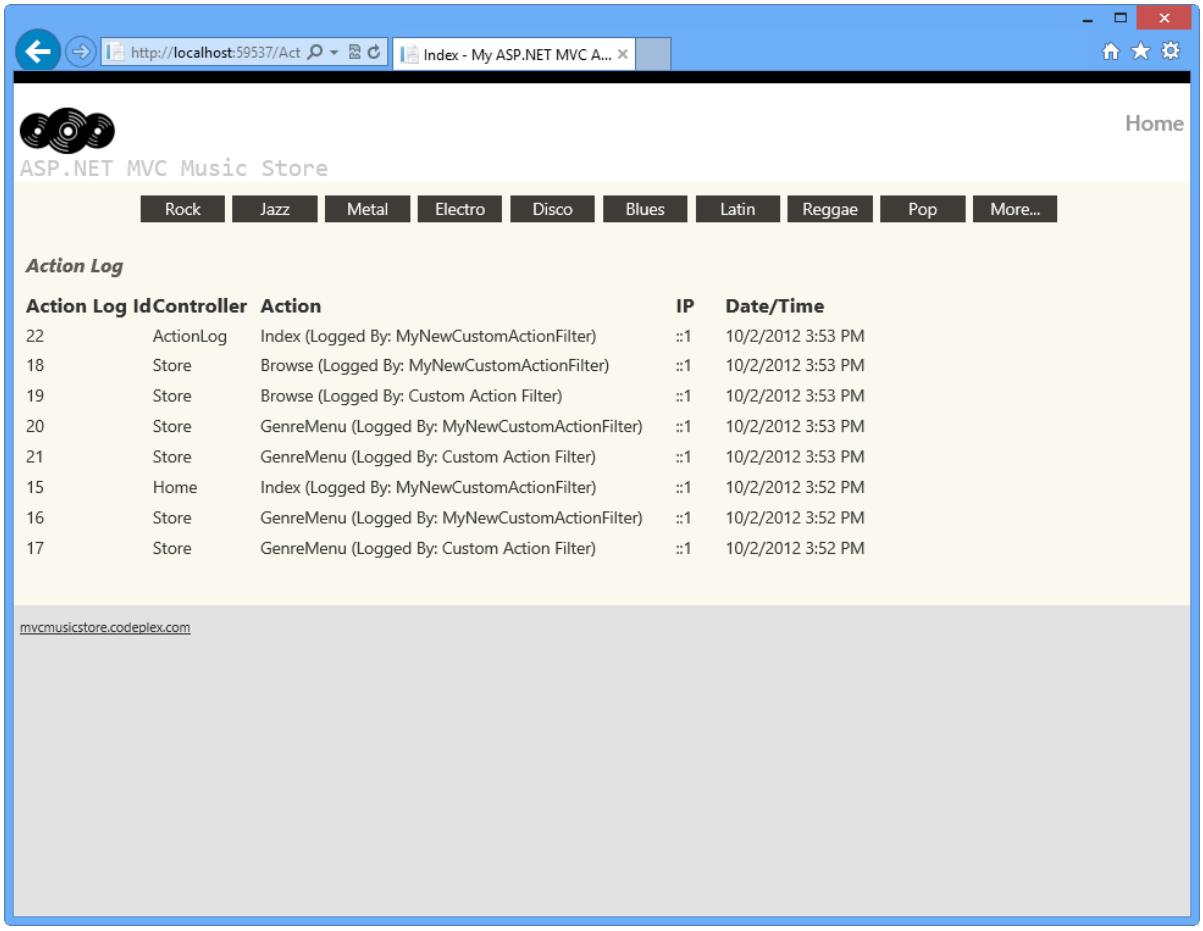
5. Update **RegisterGlobalFilters** method adding your custom filter. To do this, add the highlighted code:

```
public static void RegisterGlobalFilters(GlobalFilterCollection filters)
{
    filters.Add(new HandleErrorAttribute());
    filters.Add(new MyNewCustomActionFilter());
}
```

6. Run the application by pressing **F5**.

7. Click one of the **Genres** from the menu and perform some actions there, like browsing an available album.

8. Check that now **[MyNewCustomActionFilter]** is being injected in HomeController and ActionLogController too.



Action log with global activity logged

> **NOTE**
>
> Additionally, you can deploy this application to Windows Azure Web Sites following Appendix B: Publishing an ASP.NET MVC 4 Application using Web Deploy.

Summary

By completing this Hands-On Lab you have learned how to extend an action filter to execute custom actions. You have also learned how to inject any filter to your page controllers. The following concepts were used:

- How to create Custom Action filters with the ASP.NET MVC ActionFilterAttribute class
- How to inject filters into ASP.NET MVC controllers
- How to manage filter ordering using the Order property
- How to register filters globally

Appendix A: Installing Visual Studio Express 2012 for Web

You can install **Microsoft Visual Studio Express 2012 for Web** or another "Express" version using the **Microsoft Web Platform Installer**. The following instructions guide you through the steps required to install *Visual studio Express 2012 for Web* using *Microsoft Web Platform Installer*.

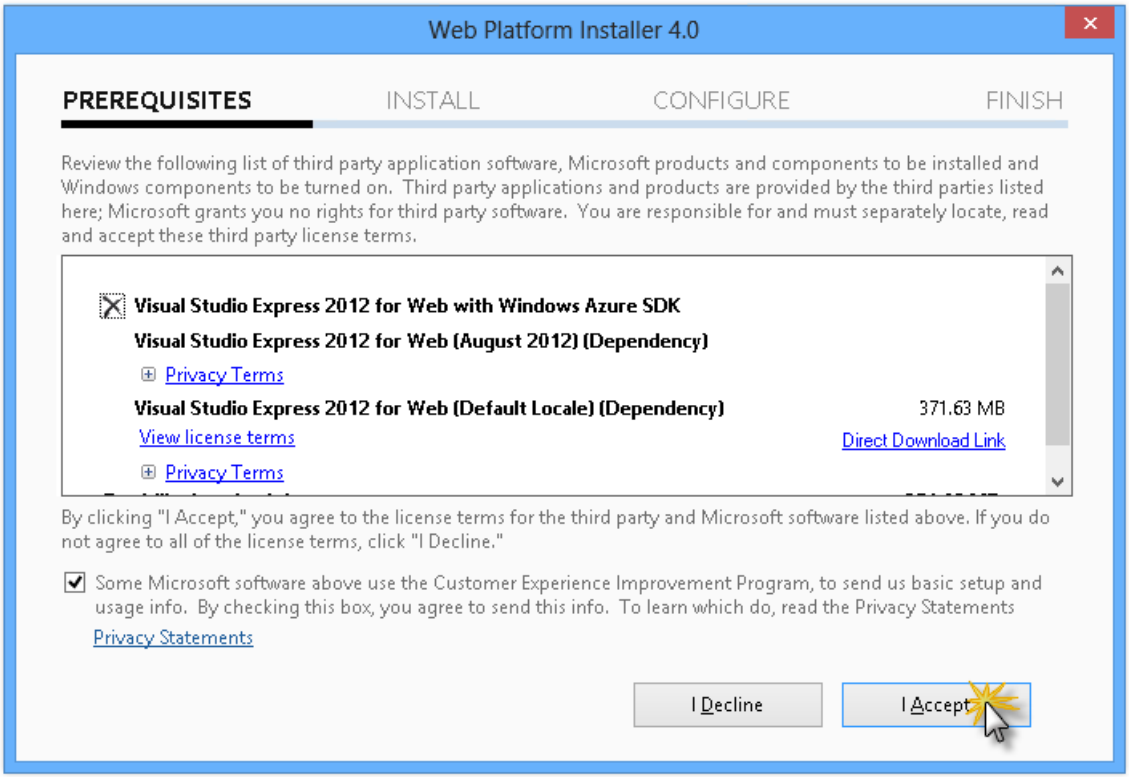
1. Go to https://go.microsoft.com/?linkid=9810169. Alternatively, if you already have installed Web Platform Installer, you can open it and search for the product "*Visual Studio Express 2012 for Web with Windows Azure SDK*".

2. Click on **Install Now**. If you do not have **Web Platform Installer** you will be redirected to download and install it first.

3. Once **Web Platform Installer** is open, click **Install** to start the setup.

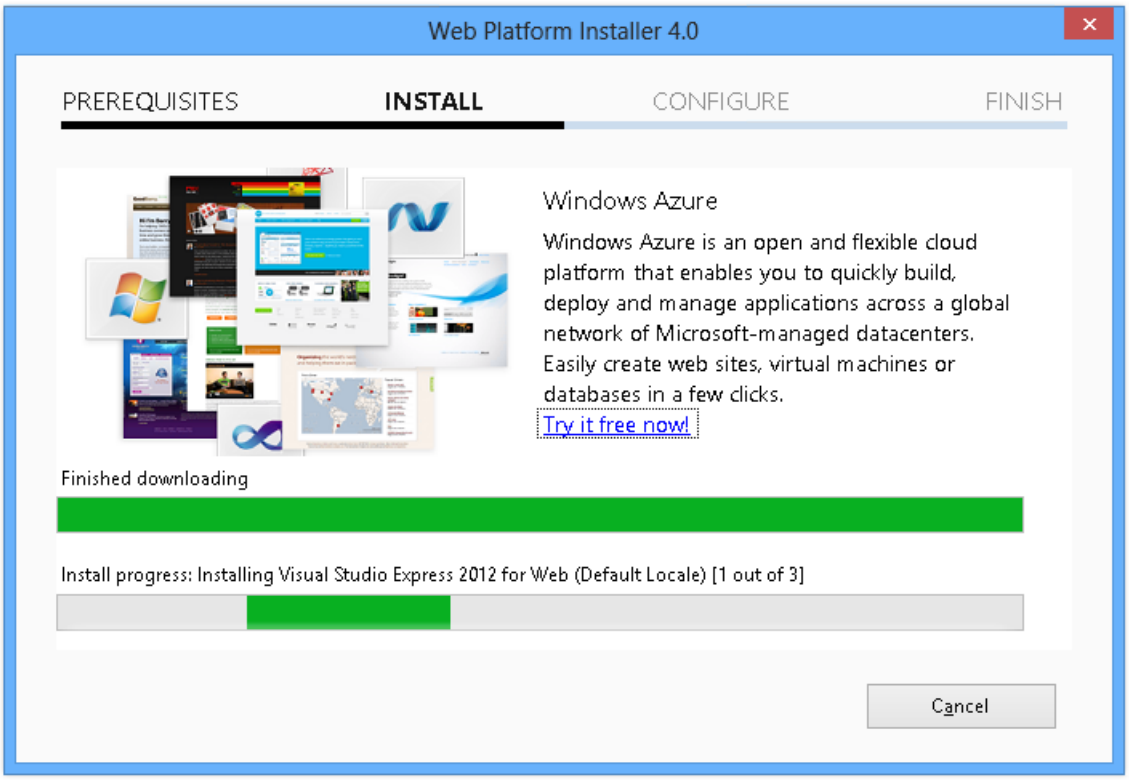
Install Visual Studio Express

4. Read all the products' licenses and terms and click **I Accept** to continue.



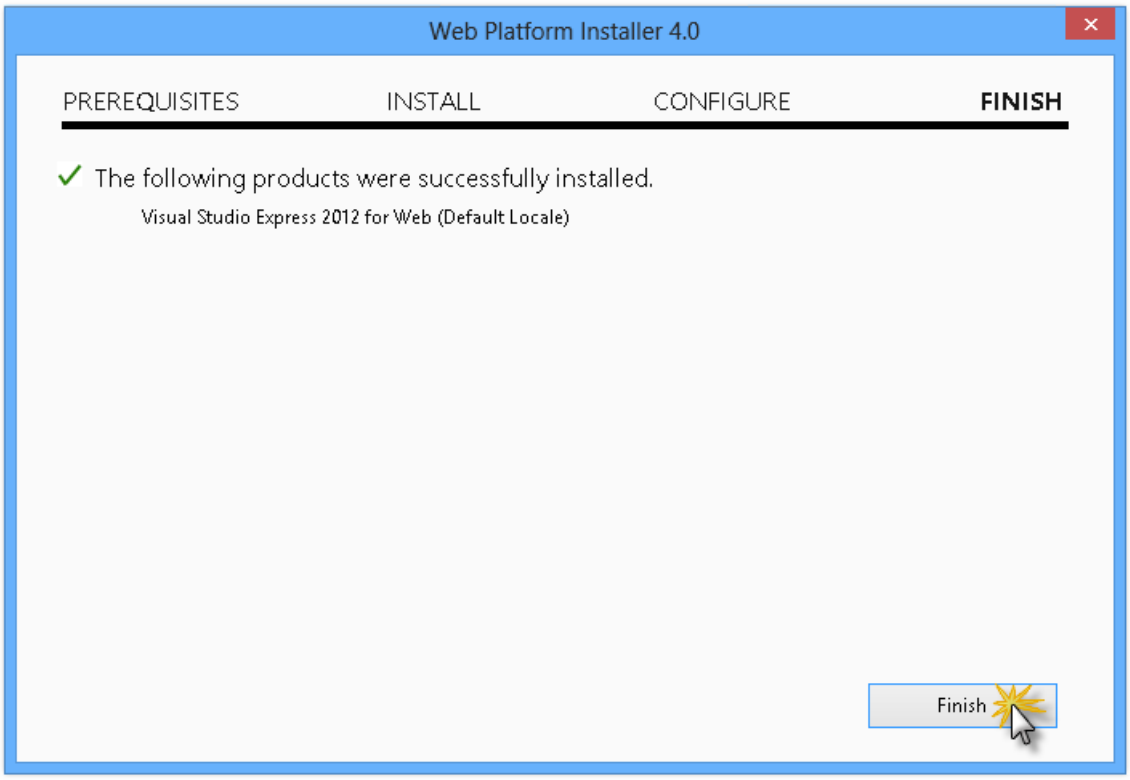
Accepting the license terms

5. Wait until the downloading and installation process completes.



Installation progress

6. When the installation completes, click **Finish**.



Installation completed

7. Click **Exit** to close Web Platform Installer.

8. To open Visual Studio Express for Web, go to the **Start** screen and start writing "**VS Express**", then click on the **VS Express for Web** tile.

VS Express for Web tile

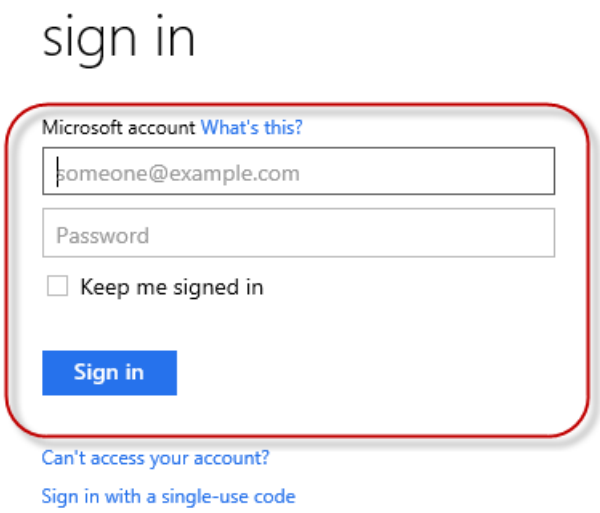
Appendix B: Publishing an ASP.NET MVC 4 Application using Web Deploy

This appendix will show you how to create a new web site from the Windows Azure Management Portal and publish the application you obtained by following the lab, taking advantage of the Web Deploy publishing feature provided by Windows Azure.

Task 1 - Creating a New Web Site from the Windows Azure Portal

1. Go to the Windows Azure Management Portal and sign in using the Microsoft credentials associated with your subscription.

> **NOTE**
>
> With Windows Azure you can host 10 ASP.NET Web Sites for free and then scale as your traffic grows. You can sign up here.



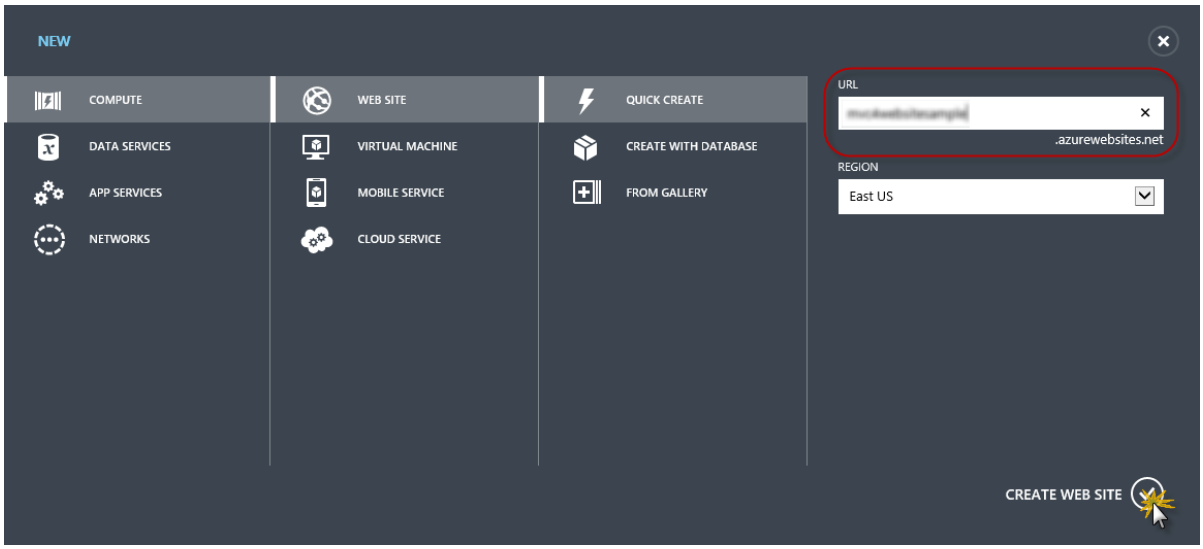
Log on to Windows Azure Management Portal

2. Click **New** on the command bar.

Creating a new Web Site

3. Click **Compute | Web Site**. Then select **Quick Create** option. Provide an available URL for the new web site and click **Create Web Site**.

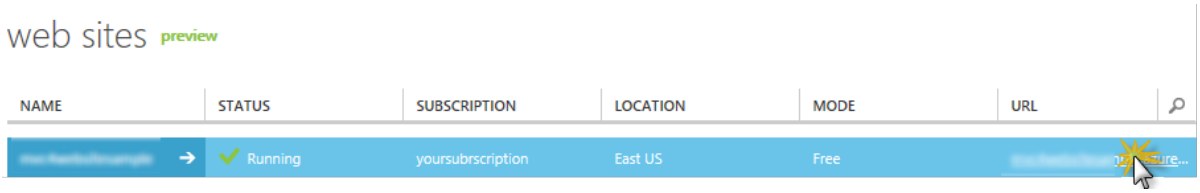
> **NOTE**
>
> A Windows Azure Web Site is the host for a web application running in the cloud that you can control and manage. The Quick Create option allows you to deploy a completed web application to the Windows Azure Web Site from outside the portal. It does not include steps for setting up a database.



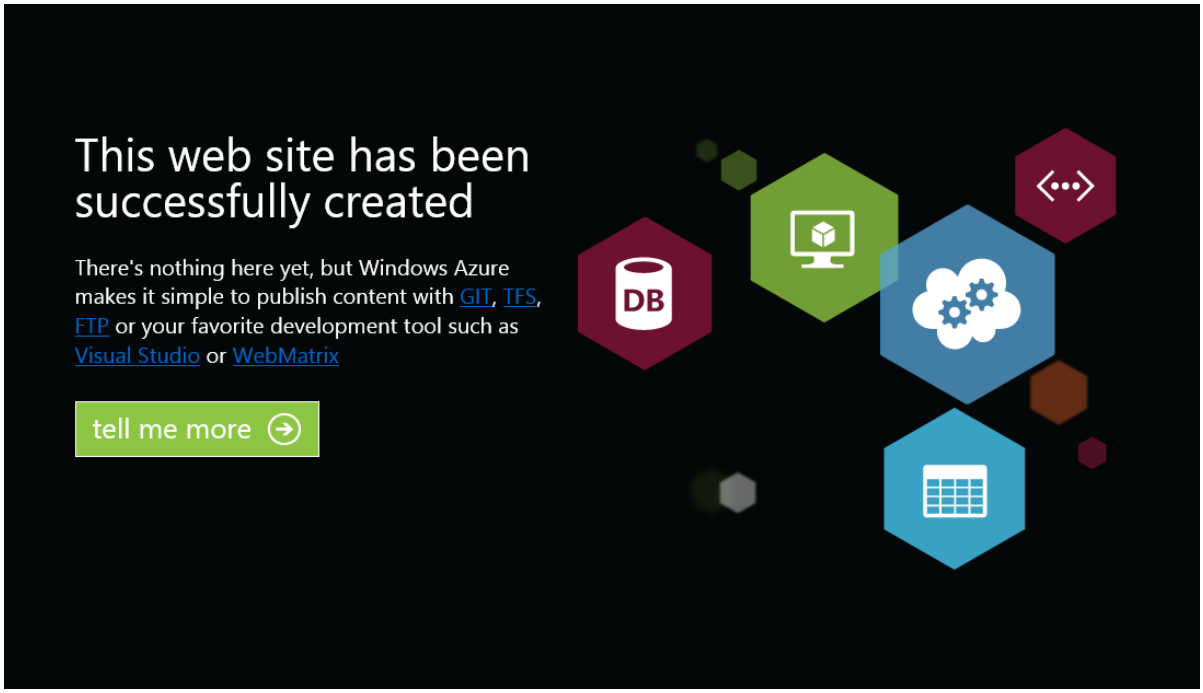
Creating a new Web Site using Quick Create

4. Wait until the new **Web Site** is created.

5. Once the Web Site is created click the link under the **URL** column. Check that the new Web Site is working.

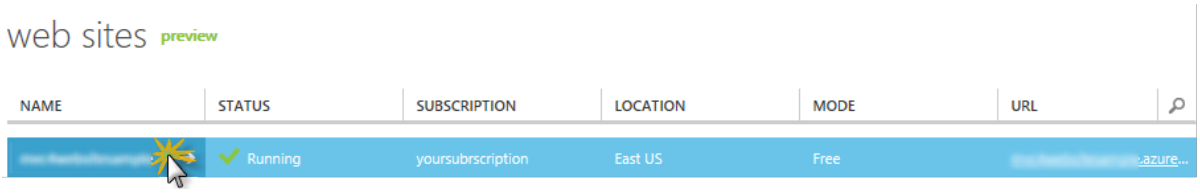


Browsing to the new web site



Web site running

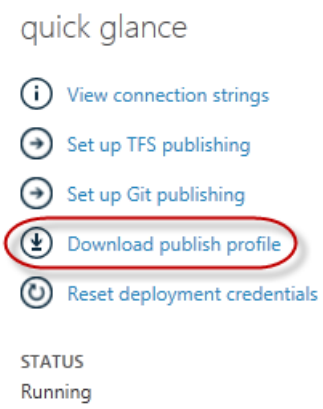
6. Go back to the portal and click the name of the web site under the **Name** column to display the management pages.



Opening the Web Site management pages

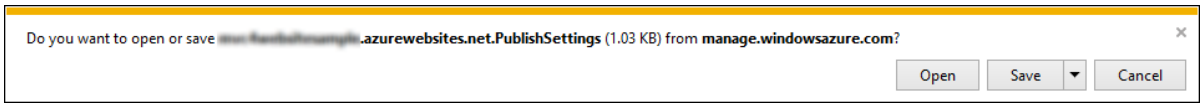
7. In the **Dashboard** page, under the **quick glance** section, click the **Download publish profile** link.

> **NOTE**
>
> The *publish profile* contains all of the information required to publish a web application to a Windows Azure website for each enabled publication method. The publish profile contains the URLs, user credentials and database strings required to connect to and authenticate against each of the endpoints for which a publication method is enabled. **Microsoft WebMatrix 2**, **Microsoft Visual Studio Express for Web** and **Microsoft Visual Studio 2012** support reading publish profiles to automate configuration of these programs for publishing web applications to Windows Azure websites.



Downloading the Web Site publish profile

8. Download the publish profile file to a known location. Further in this exercise you will see how to use this file to publish a web application to a Windows Azure Web Sites from Visual Studio.

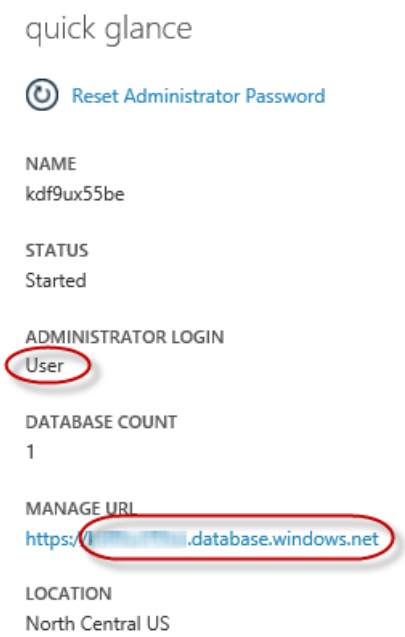


Saving the publish profile file

Task 2 - Configuring the Database Server

If your application makes use of SQL Server databases you will need to create a SQL Database server. If you want to deploy a simple application that does not use SQL Server you might skip this task.

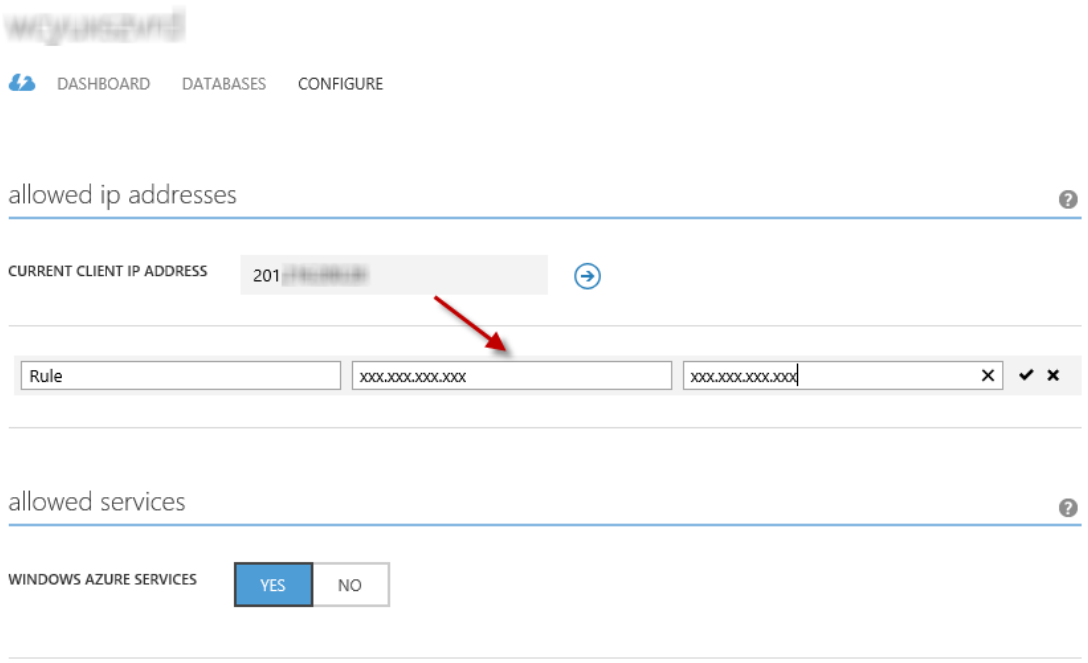
1. You will need a SQL Database server for storing the application database. You can view the SQL Database servers from your subscription in the Windows Azure Management portal at **Sql Databases | Servers | Server's Dashboard**. If you do not have a server created, you can create one using the **Add** button on the command bar. Take note of the **server name and URL, administrator login name and password**, as you will use them in the next tasks. Do not create the database yet, as it will be created in a later stage.



SQL Database Server Dashboard

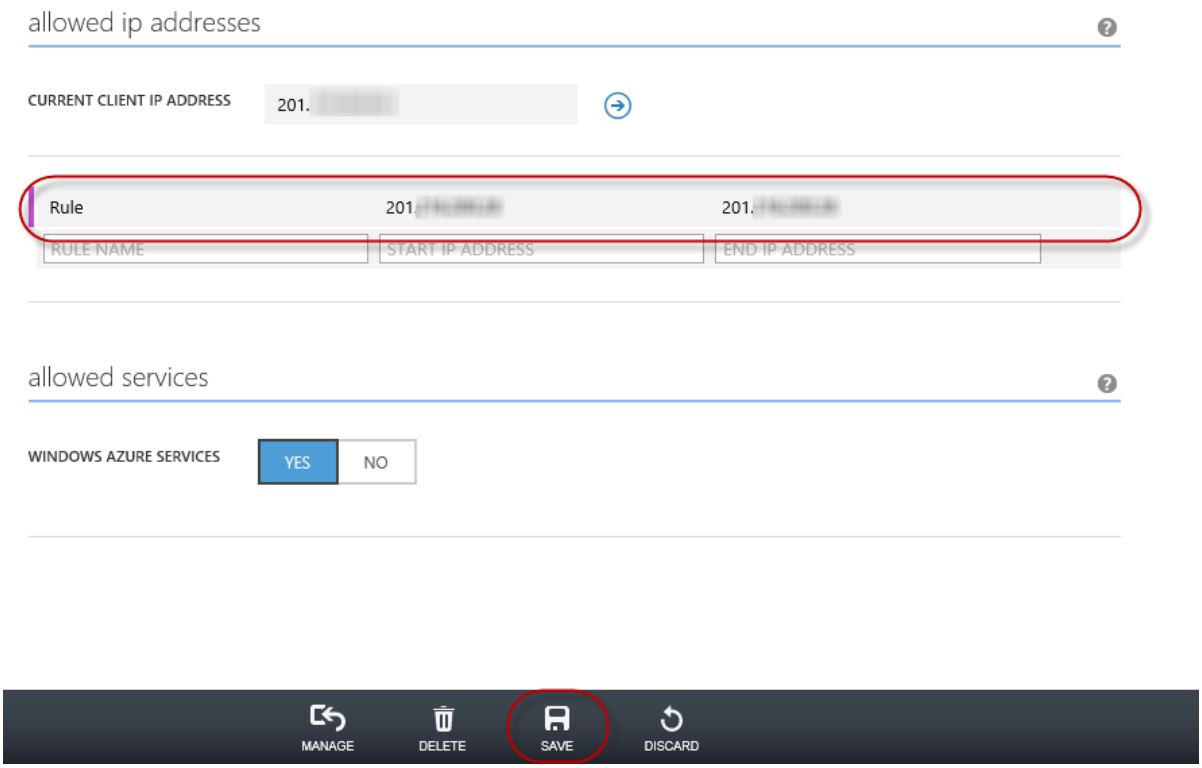
2. In the next task you will test the database connection from Visual Studio, for that reason you need to include your local IP address in the server's list of **Allowed IP Addresses**. To do that, click **Configure**, select the IP address from **Current Client IP Address** and paste it on the **Start IP Address** and **End IP Address** text

boxes and click the ✔ button.



Adding Client IP Address

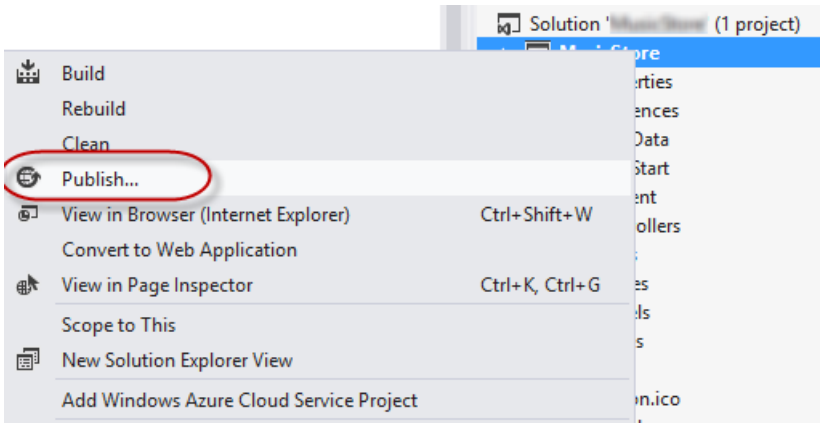
3. Once the **Client IP Address** is added to the allowed IP addresses list, click on **Save** to confirm the changes.



Confirm Changes

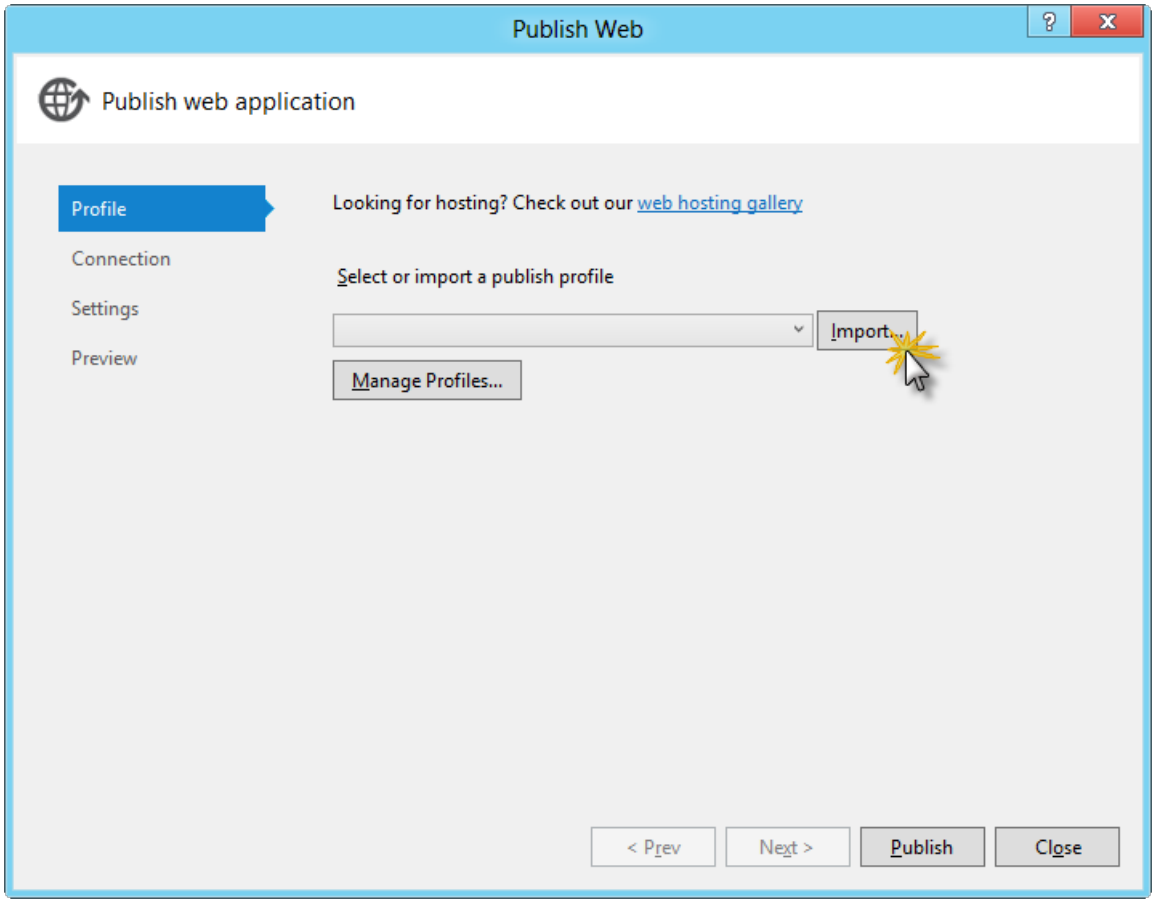
Task 3 - Publishing an ASP.NET MVC 4 Application using Web Deploy

1. Go back to the ASP.NET MVC 4 solution. In the **Solution Explorer**, right-click the web site project and select **Publish**.



Publishing the web site

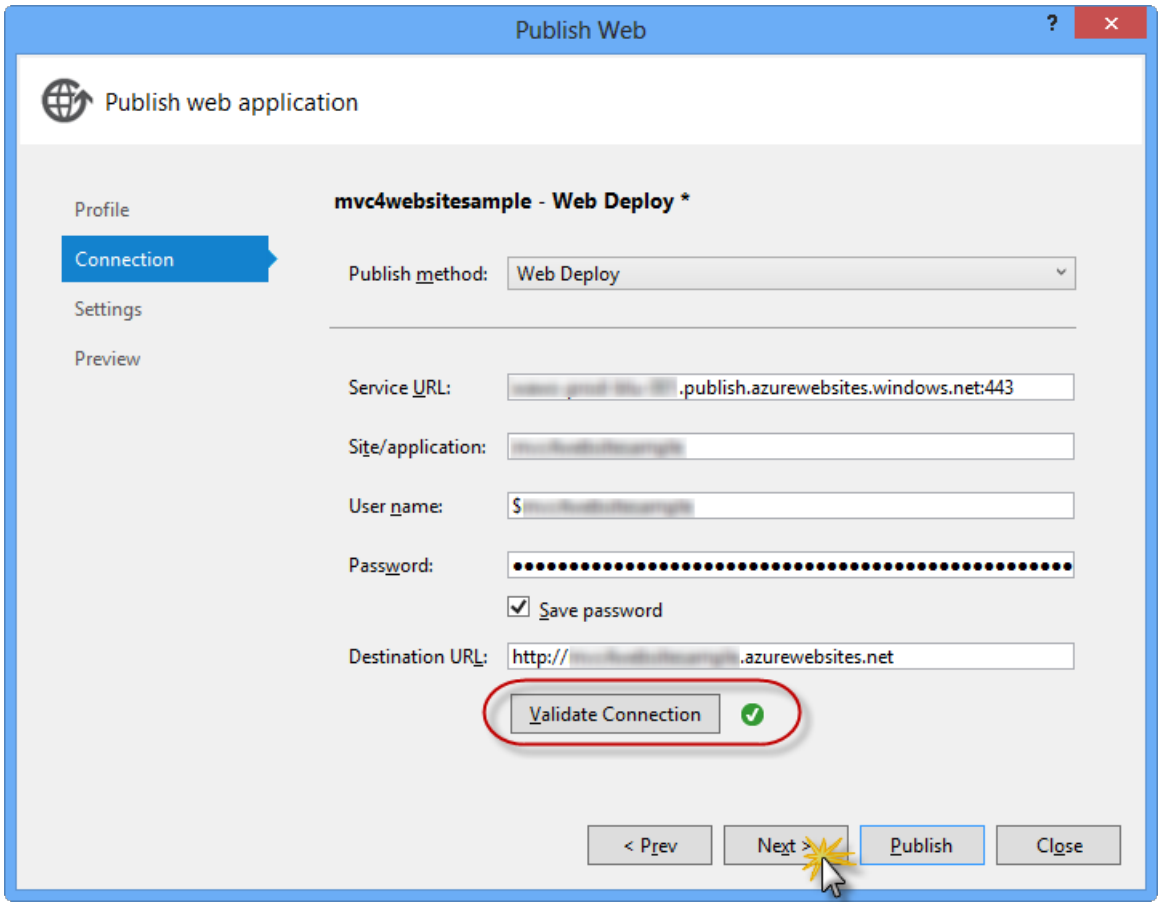
2. Import the publish profile you saved in the first task.



Importing publish profile

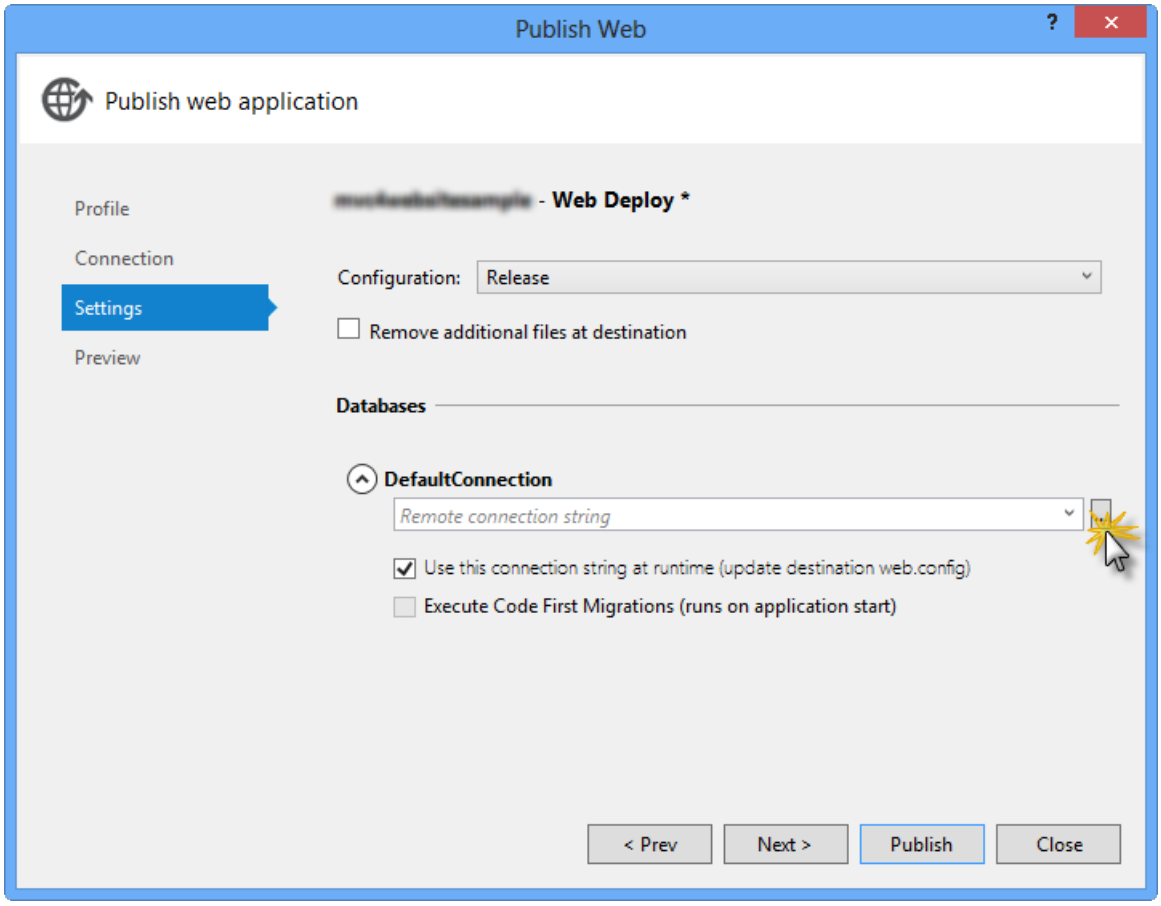
3. Click **Validate Connection**. Once Validation is complete click **Next**.

> **NOTE**
>
> Validation is complete once you see a green checkmark appear next to the Validate Connection button.



Validating connection

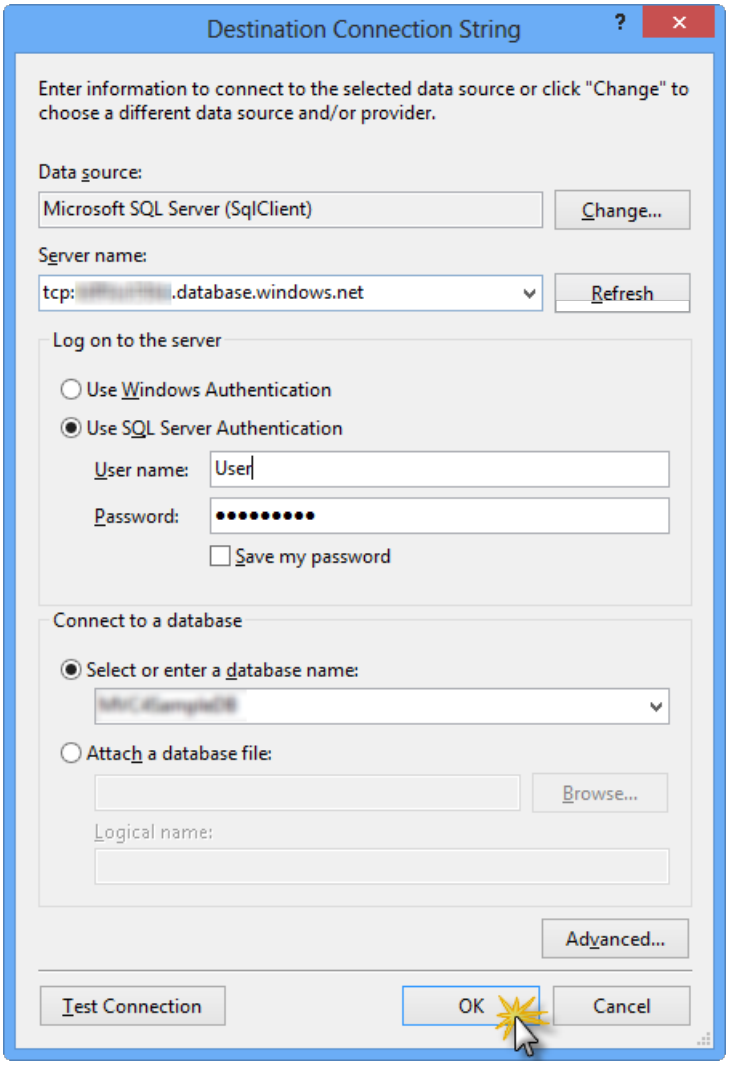
4. In the **Settings** page, under the **Databases** section, click the button next to your database connection's textbox (i.e. **DefaultConnection**).



Web deploy configuration

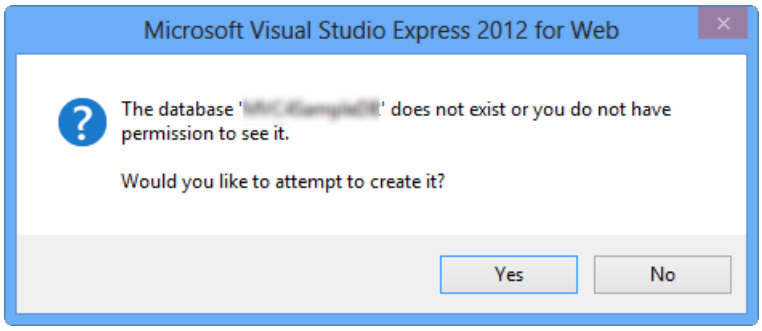
5. Configure the database connection as follows:

   - In the **Server name** type your SQL Database server URL using the *tcp:* prefix.
   - In **User name** type your server administrator login name.
   - In **Password** type your server administrator login password.
   - Type a new database name.



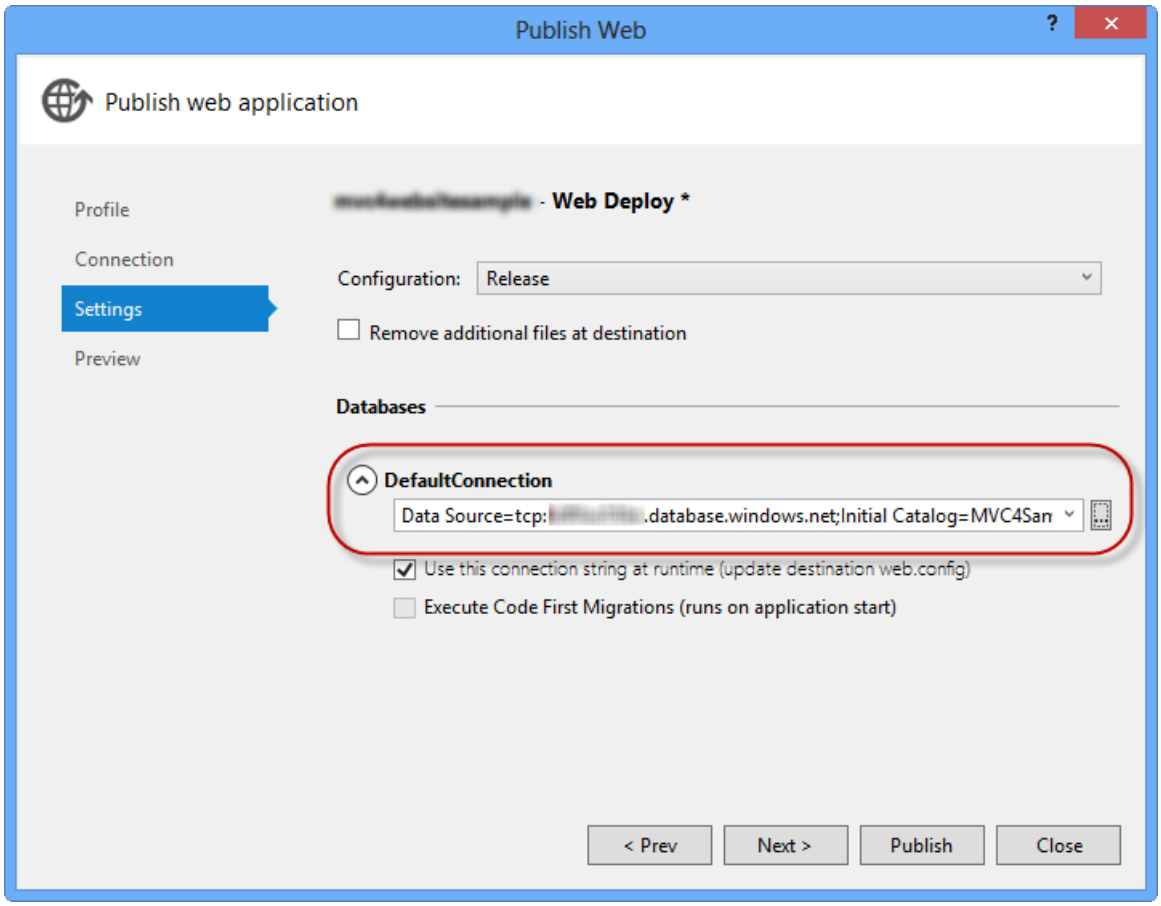
Configuring destination connection string

6. Then click **OK**. When prompted to create the database click **Yes**.



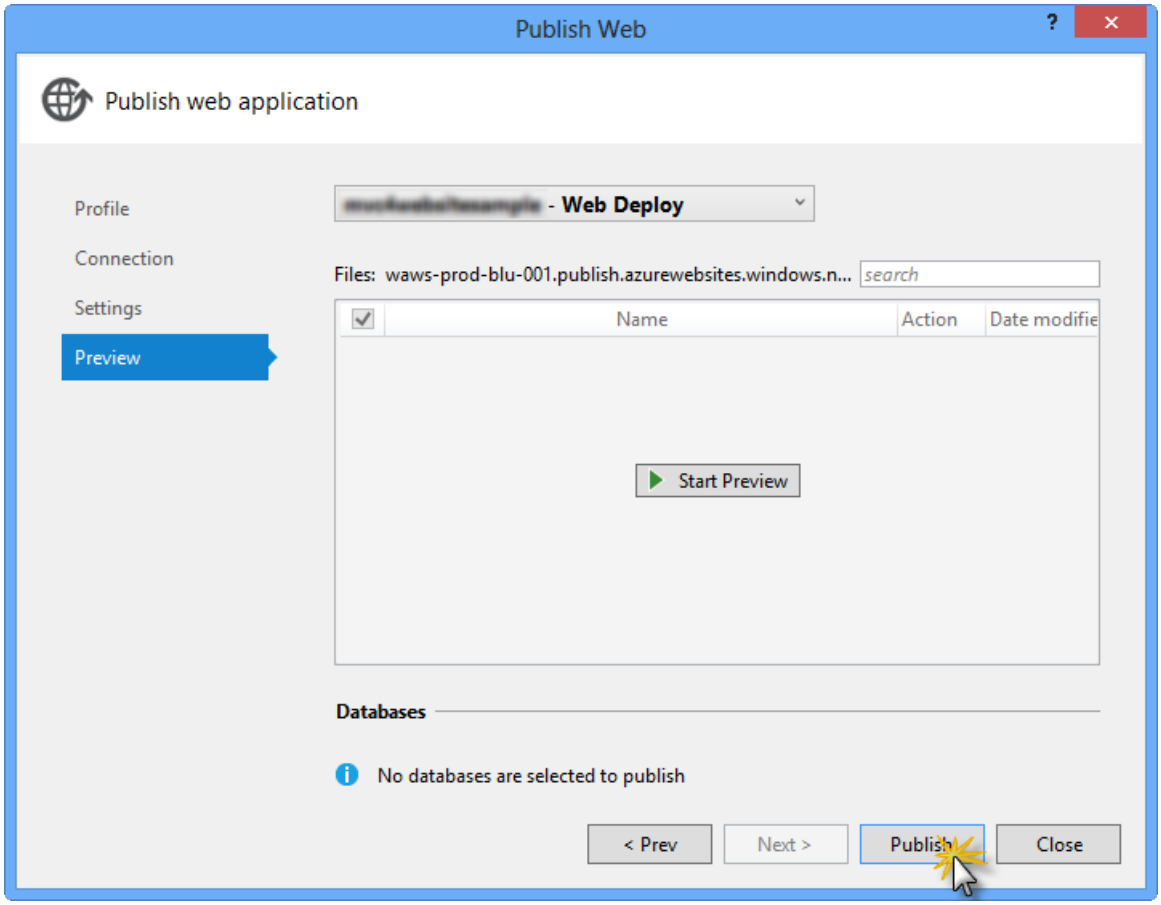
Creating the database

7. The connection string you will use to connect to SQL Database in Windows Azure is shown within Default Connection textbox. Then click **Next**.



Connection string pointing to SQL Database

8. In the **Preview** page, click **Publish**.



Publishing the web application

9. Once the publishing process finishes, your default browser will open the published web site.

Appendix C: Using Code Snippets

With code snippets, you have all the code you need at your fingertips. The lab document will tell you exactly when you can use them, as shown in the following figure.

6. Insert the following code inside the body of the **Run** method.

(Code Snippet – *AzurePlatformTrainingKit - Ex1-Task 1 -Step 5- Run method*)

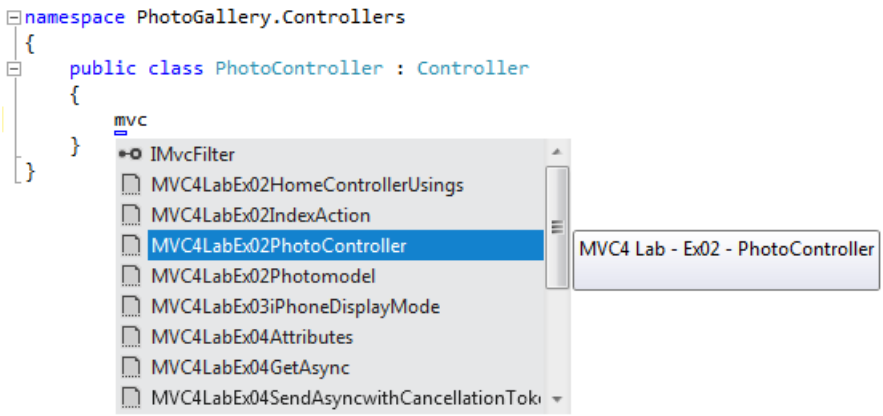
C#

```
public override void Run()
{
  while (true)
  {
    // retrieve a new message from the queue
    CloudQueueMessage message = queue.GetMessage();
    if (message != null)
    {
      ProcessMessage(message);
      queue.DeleteMessage(msg);
    }
  }
}
```

Using Visual Studio code snippets to insert code into your project

To add a code snippet using the keyboard (C# only)

1. Place the cursor where you would like to insert the code.
2. Start typing the snippet name (without spaces or hyphens).
3. Watch as IntelliSense displays matching snippets' names.
4. Select the correct snippet (or keep typing until the entire snippet's name is selected).
5. Press the Tab key twice to insert the snippet at the cursor location.



Start typing the snippet name

```
static void Main(string[] args)
{
    AzureTrainingKitServiceBus
}
```

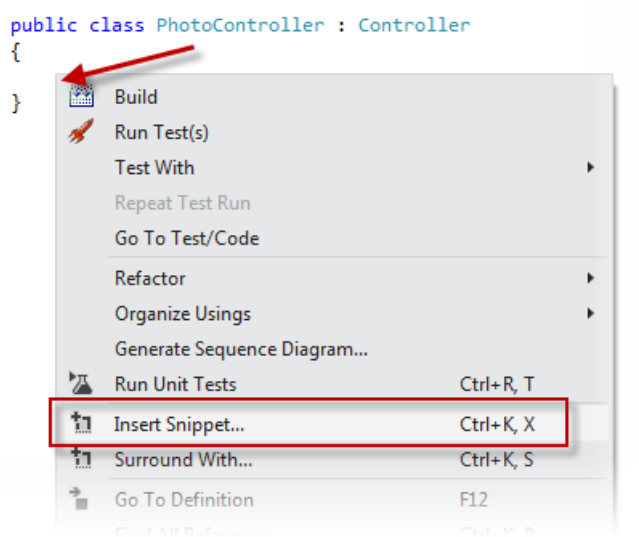
Press Tab to select the highlighted snippet

```
static void Main(string[] args)
{
    request.Headers.Add("$accountKey", accountKey);
    request.Headers.Add("$uniqueUserID", uniqueUserId);
}
```

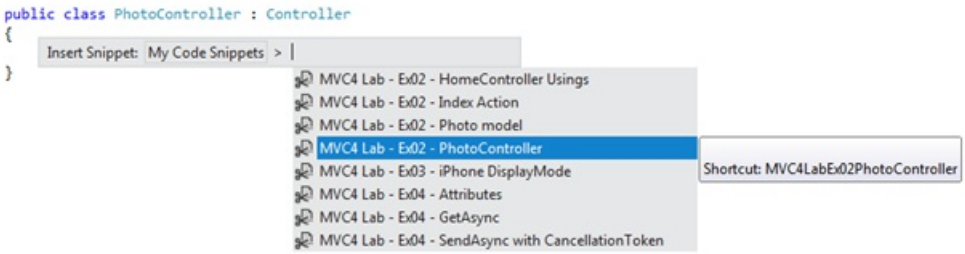
Press Tab again and the snippet will expand

To add a code snippet using the mouse (C#, Visual Basic and XML) 1. Right-click where you want to insert the code snippet.

1. Select **Insert Snippet** followed by **My Code Snippets**.
2. Pick the relevant snippet from the list, by clicking on it.



Right-click where you want to insert the code snippet and select Insert Snippet



Pick the relevant snippet from the list, by clicking on it

ASP.NET MVC 4 Entity Framework Scaffolding and Migrations

2/20/2020 • 9 minutes to read • Edit Online

By Web Camps Team

Download Web Camps Training Kit

If you are familiar with ASP.NET MVC 4 controller methods, or have completed the "Helpers, Forms and Validation" Hands-On lab, you should be aware that many of the logic to create, update, list and remove any data entity it is repeated among the application. Not to mention that, if your model has several classes to manipulate, you will be likely to spend a considerable time writing the POST and GET action methods for each entity operation, as well as each of the views.

In this lab you will learn how to use the ASP.NET MVC 4 scaffolding to automatically generate the baseline of your application's CRUD (Create, Read, Update and Delete). Starting from a simple model class, and, without writing a single line of code, you will create a controller that will contain all the CRUD operations, as well as the all the necessary views. After building and running the simple solution, you will have the application database generated, together with the MVC logic and views for data manipulation.

In addition, you will learn how easy it is to use Entity Framework Migrations to perform model updates throughout your entire application. Entity Framework Migrations will let you modify your database after the model has changed with simple steps. With all these in mind, you will be able to build and maintain web applications more efficiently, taking advantage of the latest features of ASP.NET MVC 4.

> **NOTE**
>
> All sample code and snippets are included in the Web Camps Training Kit, available from at Microsoft-Web/WebCampTrainingKit Releases. The project specific to this lab is available at ASP.NET MVC 4 Entity Framework Scaffolding and Migrations.

Objectives

In this Hands-On Lab, you will learn how to:

- Use ASP.NET scaffolding for CRUD operations in controllers.
- Change the database model using Entity Framework Migrations.

Prerequisites

You must have the following items to complete this lab:

- Microsoft Visual Studio Express 2012 for Web or superior (read Appendix A for instructions on how to install it).

Setup

Installing Code Snippets

For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. To install the code snippets run `.\Source\Setup\CodeSnippets.vsi` file.

If you are not familiar with the Visual Studio Code Snippets, and want to learn how to use them, you can refer to the appendix from this document "Appendix B: Using Code Snippets".

Exercises

The following exercise make up this Hands-On Lab:

1. Using ASP.NET MVC 4 Scaffolding with Entity Framework Migrations

> **NOTE**
>
> This exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercise. You can use this solution as a guide if you need additional help working through the exercise.

Estimated time to complete this lab: **30 minutes**

Exercise 1: Using ASP.NET MVC 4 Scaffolding with Entity Framework Migrations

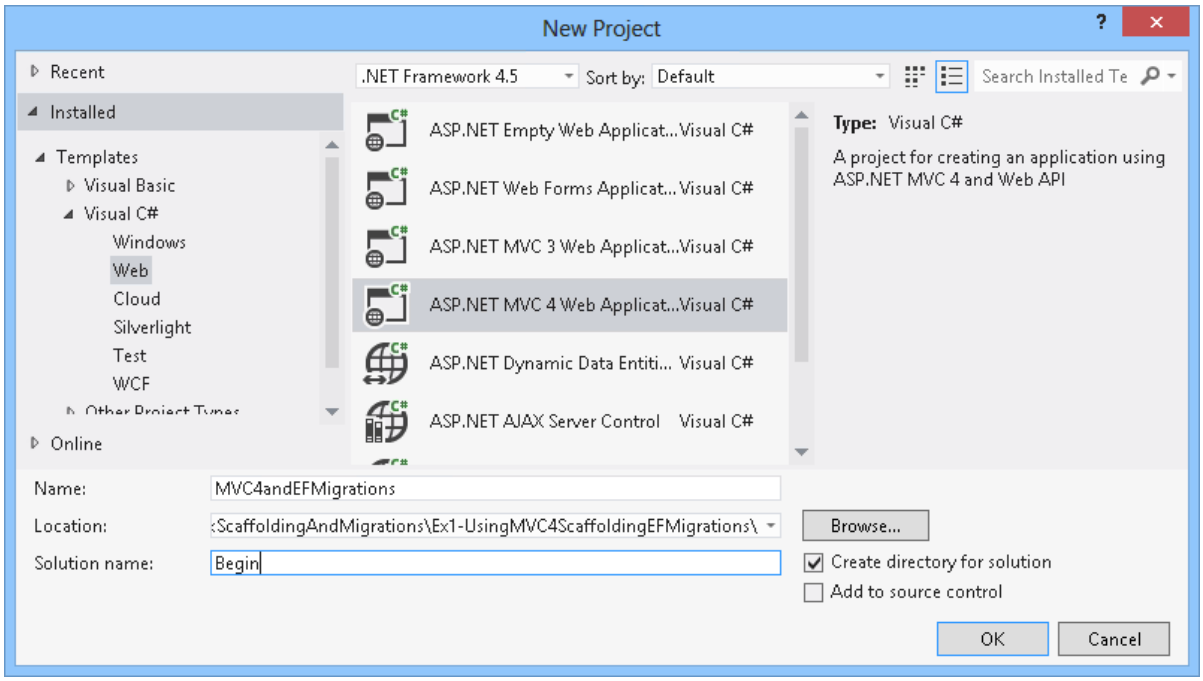
ASP.NET MVC scaffolding provides a quick way to generate the CRUD operations in a standardized way, creating the necessary logic that lets your application interact with the database layer.

In this exercise, you will learn how to use ASP.NET MVC 4 scaffolding with code first to create the CRUD methods. Then, you will learn how to update your model applying the changes in the database by using Entity Framework Migrations.

Task 1- Creating a new ASP.NET MVC 4 project using Scaffolding

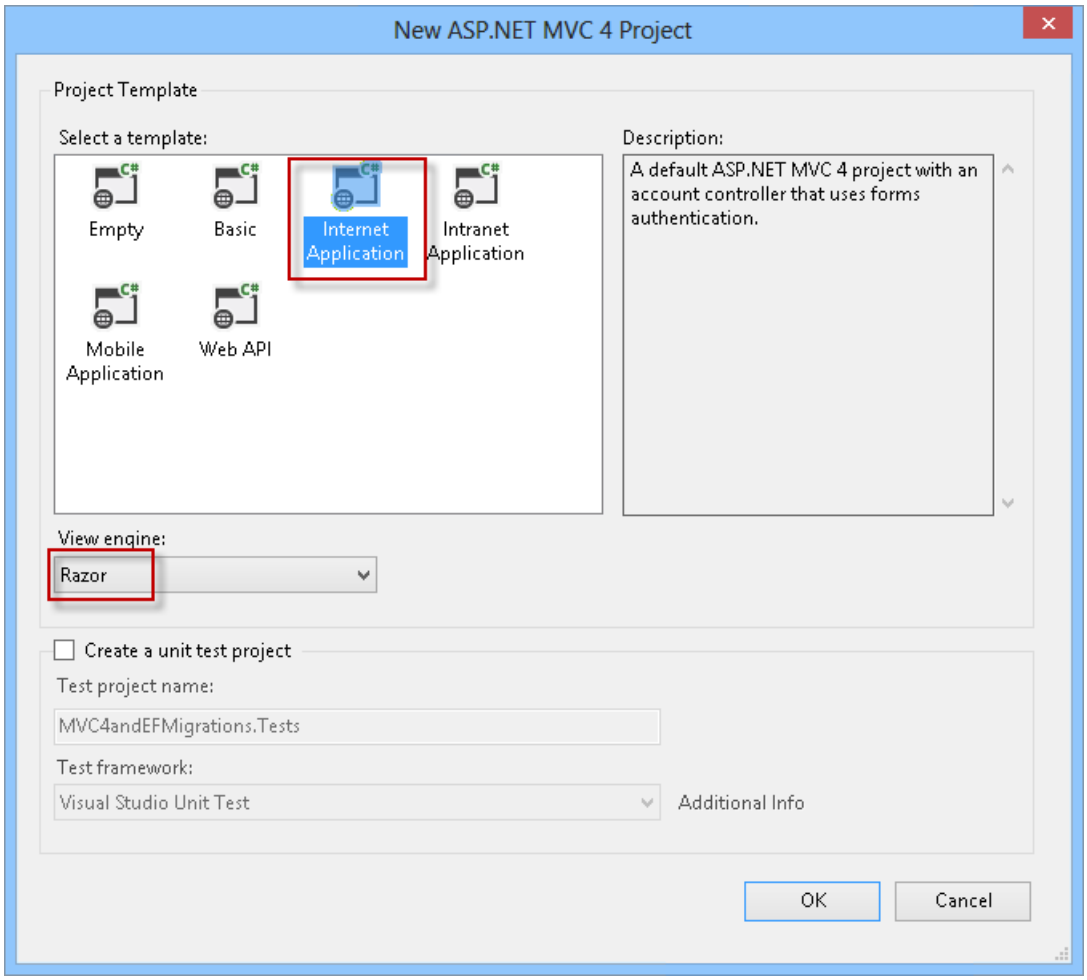
1. If not already open, start **Visual Studio 2012**.

2. Select **File | New Project**. In the New Project dialog, under the **Visual C# | Web** section, select **ASP.NET MVC 4 Web Application**. Name the project to **MVC4andEFMigrations** and set the location to **Source\Ex1-UsingMVC4ScaffoldingEFMigrations** folder of this lab. Set the **Solution name** to **Begin** and ensure **Create directory for solution** is checked. Click **OK**.



New ASP.NET MVC 4 Project Dialog Box

3. In the **New ASP.NET MVC 4 Project** dialog box select the **Internet Application** template, and make sure that **Razor** is the selected **View engine**. Click **OK** to create the project.



New ASP.NET MVC 4 Internet Application

4. In the Solution Explorer, right-click **Models** and select **Add | Class** to create a simple class person (POCO). Name it **Person** and click **OK**.

5. Open the Person class and insert the following properties.

   (Code Snippet - *ASP.NET MVC 4 and Entity Framework Migrations - Ex1 Person Properties*)

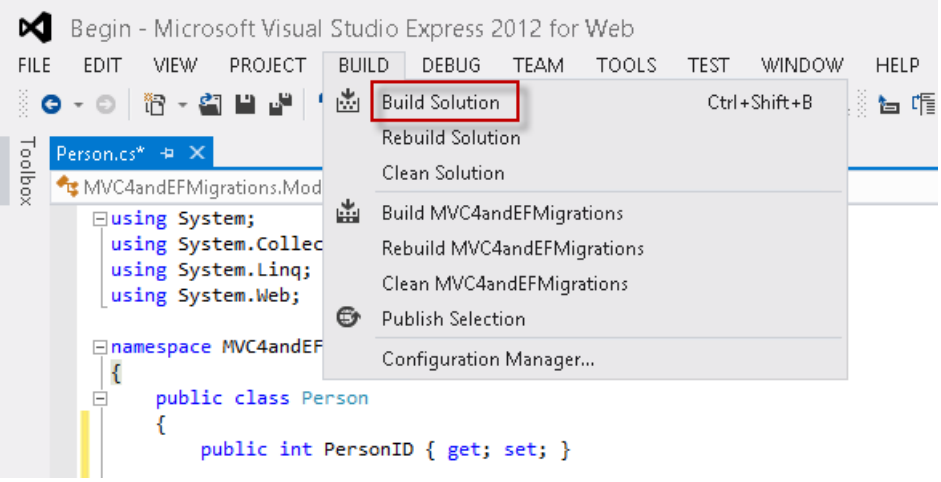
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace MVC4EF.Models
{
    public class Person
    {
        public int PersonID { get; set; }

        public string FirstName { get; set; }

        public string LastName { get; set; }
    }
}
```

6. Click **Build | Build Solution** to save the changes and build the project.



Building the Application

7. In the Solution Explorer, right-click the controllers folder and select **Add | Controller**.

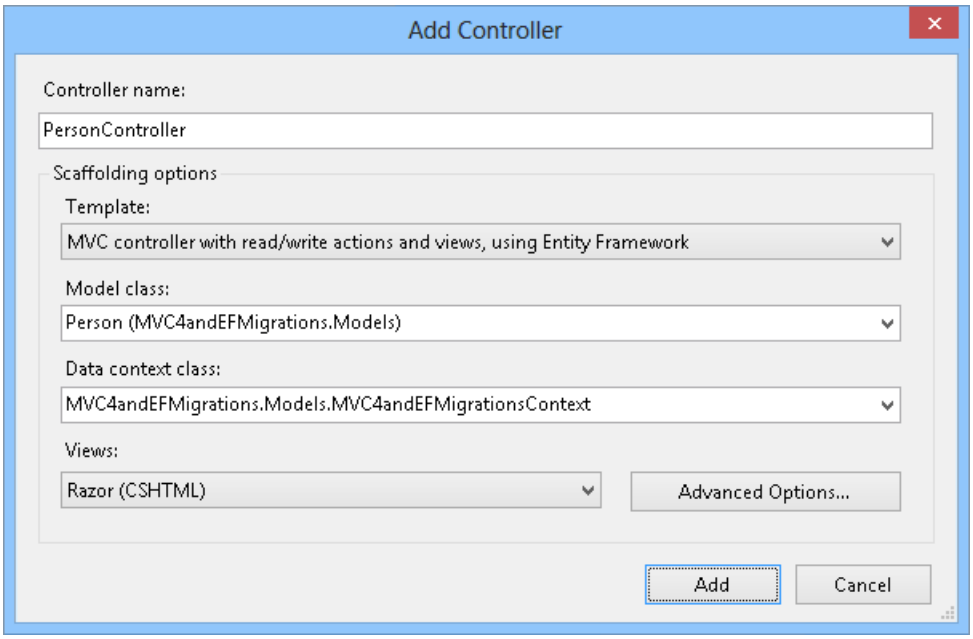
8. Name the controller *PersonController* and complete the **Scaffolding options** with the following values.

   a. In the **Template** drop-down list, select the **MVC controller with read/write actions and views, using Entity Framework** option.

   b. In the **Model class** drop-down list, select the **Person** class.

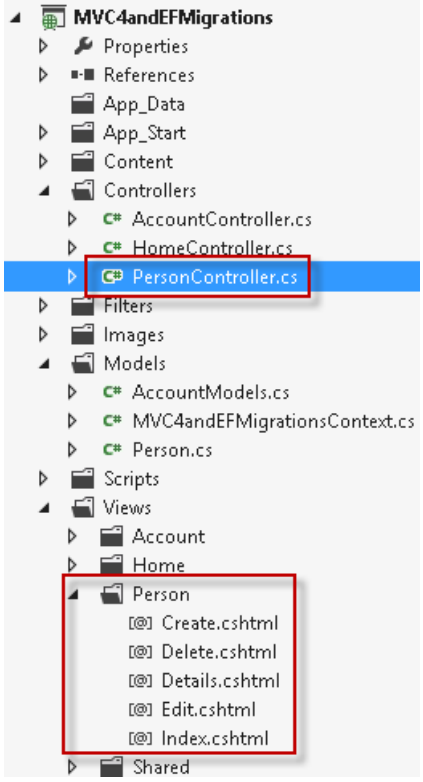
   c. In the **Data Context class** list, select **<New data context...>**. Choose any name and click **OK**.

   d. In the **Views** drop-down list, make sure that **Razor** is selected.



Adding the Person controller with scaffolding

9. Click **Add** to create the new controller for Person with scaffolding. You have now generated the controller actions as well as the views.



After creating the Person controller with scaffolding

10. Open **PersonController** class. Notice that the full CRUD action methods have been generated automatically.

```
//
// GET: /Person/Details/5

public ActionResult Details(int id = 0)...

//
// GET: /Person/Create

public ActionResult Create()...

//
// POST: /Person/Create

[HttpPost]
public ActionResult Create(Person person)...

//
// GET: /Person/Edit/5

public ActionResult Edit(int id = 0)...

//
// POST: /Person/Edit/5

[HttpPost]
public ActionResult Edit(Person person)...

//
// GET: /Person/Delete/5

public ActionResult Delete(int id = 0)...
```

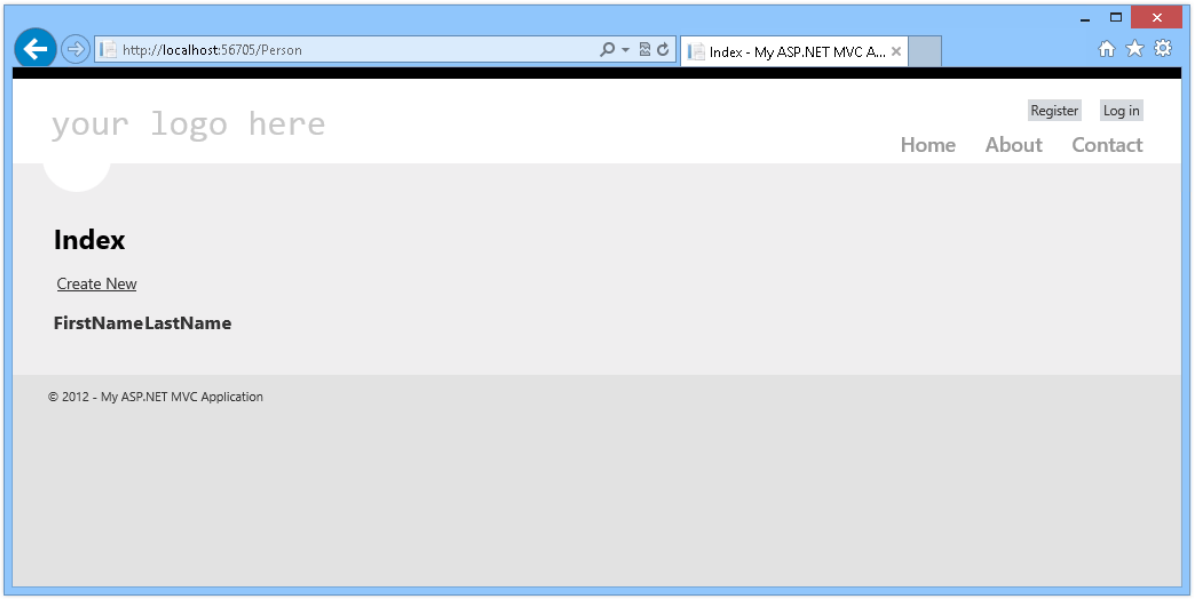
Inside the Person controller

Task 2- Running the application

At this point, the database is not yet created. In this task, you will run the application for the first time and test the CRUD operations. The database will be created on the fly with Code First.

1. Press **F5** to run the application.

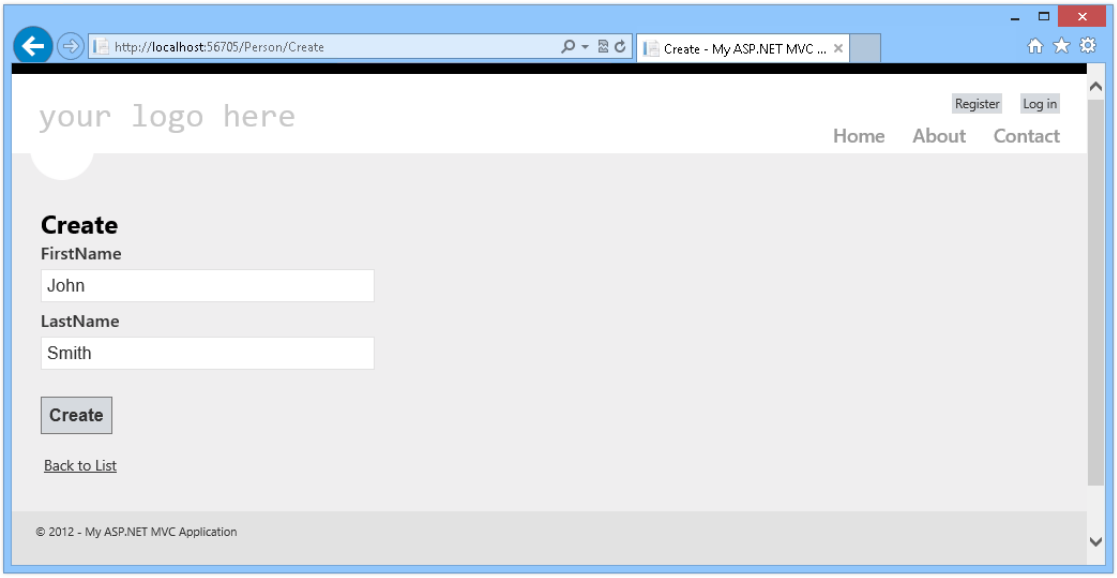
2. In the browser, add **/Person** to the URL to open the Person page.



Application: first run

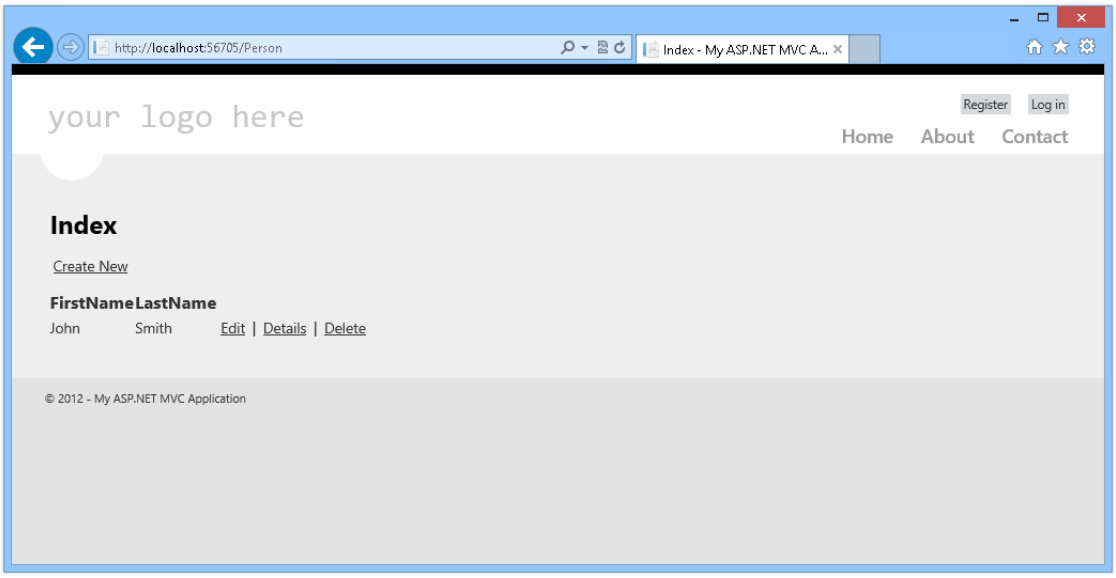
3. You will now explore the Person pages and test the CRUD operations.

   a. Click **Create New** to add a new person. Enter a first name and a last name and click **Create**.



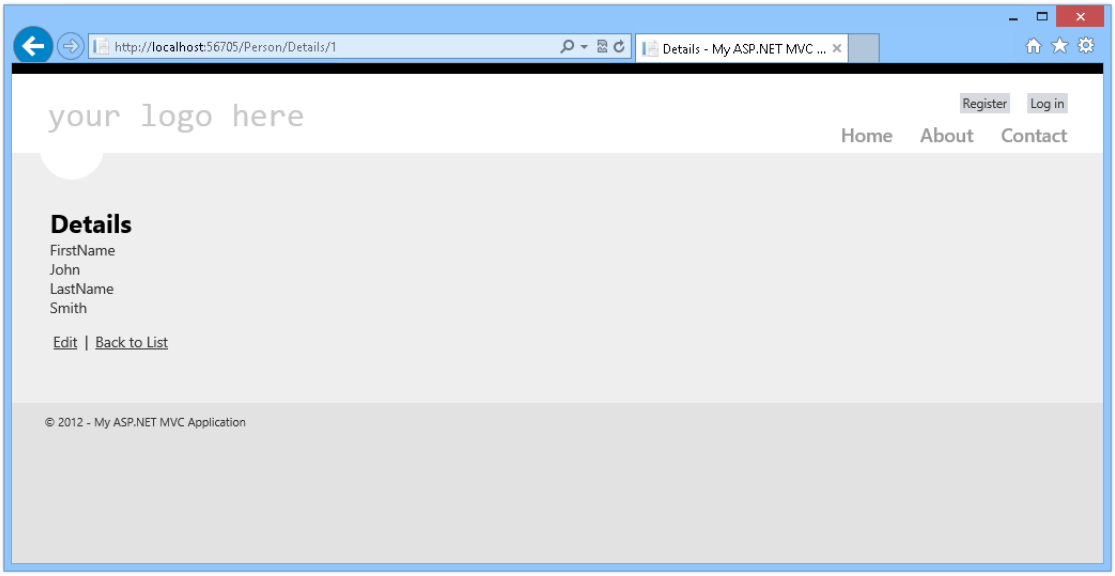
Adding a new person

   b. In the person's list, you can delete, edit or add items.



Person list

c. Click **Details** to open the person's details.



Person's details

4. Close the browser and return to Visual Studio. Notice that you have created the whole CRUD for the person entity throughout your application -from the model to the views- without having to write a single line of code!

Task 3- Updating the database using Entity Framework Migrations

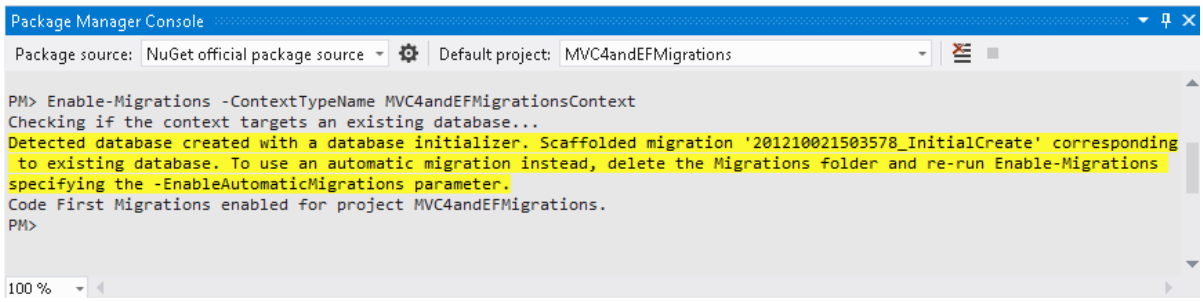
In this task you will update the database using Entity Framework Migrations. You will discover how easy it is to change the model and reflect the changes in your databases by using the Entity Framework Migrations feature.

1. Open the Package Manager Console. Select **Tools** > **NuGet Package Manager** > **Package Manager Console**.

2. In the Package Manager Console, enter the following command:

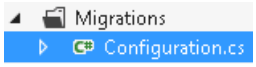
   PMC

```
Enable-Migrations -ContextTypeName [ContextClassName]
```



Enabling migrations

The Enable-Migration command creates the **Migrations** folder, which contains a script to initialize the database.



Migrations folder

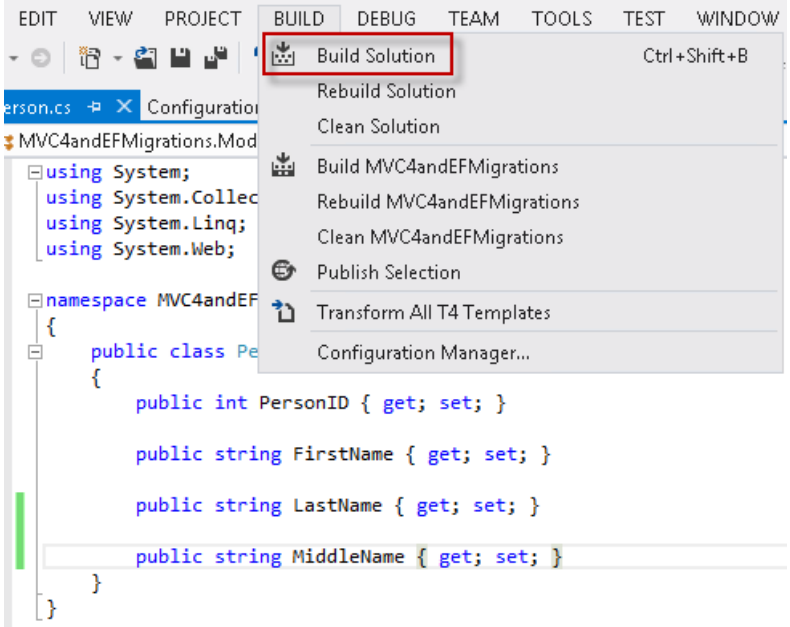
3. Open the **Configuration.cs** file in the Migrations folder. Locate the class constructor and change the **AutomaticMigrationsEnabled** value to *true*.

```
public Configuration()
{
    AutomaticMigrationsEnabled = true;
}
```

4. Open the Person class and add an attribute for the person's middle name. With this new attribute, you are changing the model.

```
public class Person
{
    public int PersonID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string MiddleName { get; set; }
}
```

5. Select **Build | Build Solution** on the menu to build the application.



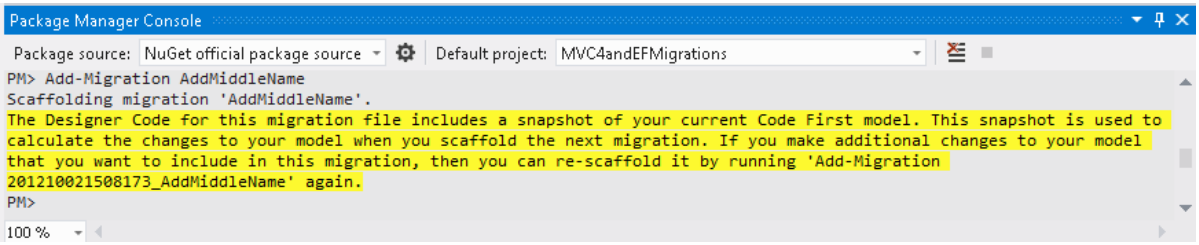
Building the application

6. In the Package Manager Console, enter the following command:

   PMC

   ```
   Add-Migration AddMiddleName
   ```

   This command will look for changes in the data objects, and then, it will add the necessary commands to modify the database accordingly.



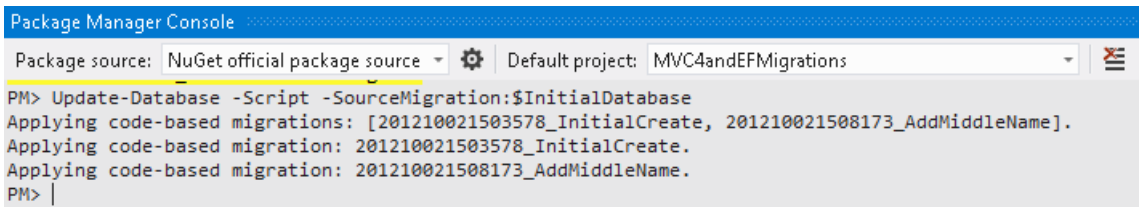
Adding a middle name

7. (Optional) You can run the following command to generate a SQL script with the differential update. This will let you update the database manually (In this case it's not necessary), or apply the changes in other databases:

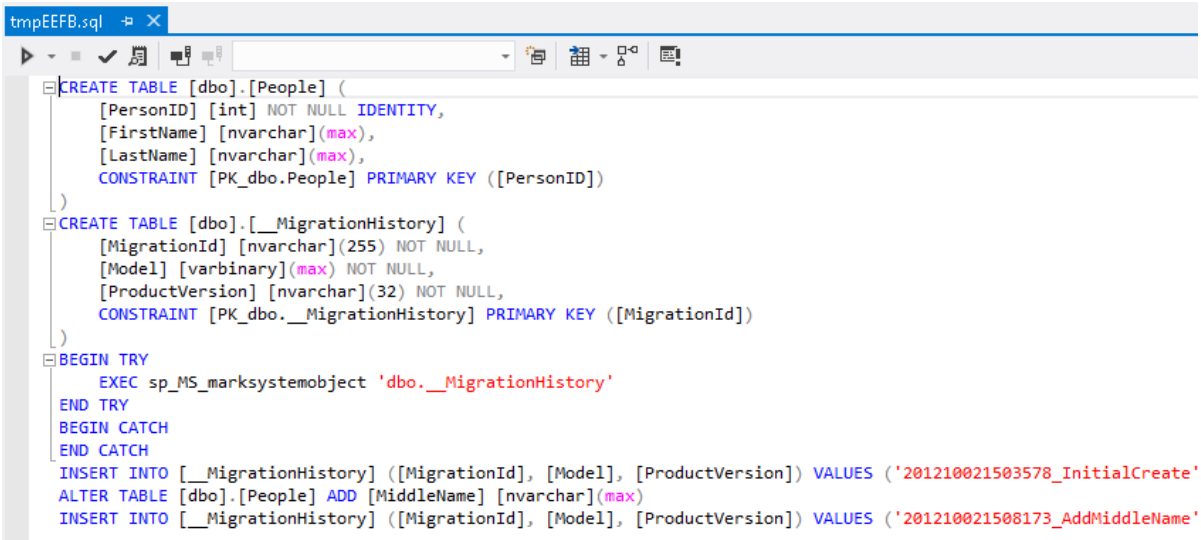
   PMC

   ```
   Update-Database -Script -SourceMigration:

   $InitialDatabase
   ```



Generating a SQL script

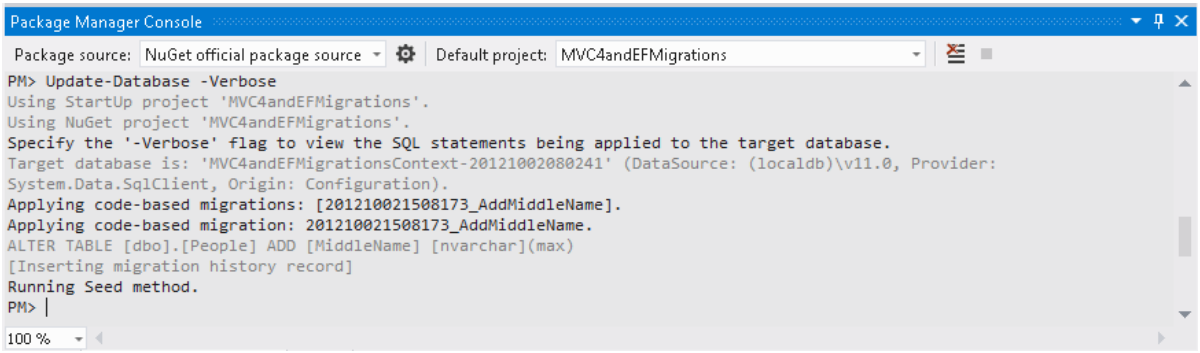


SQL Script update

8. In the Package Manager Console, enter the following command to update the database:

PMC

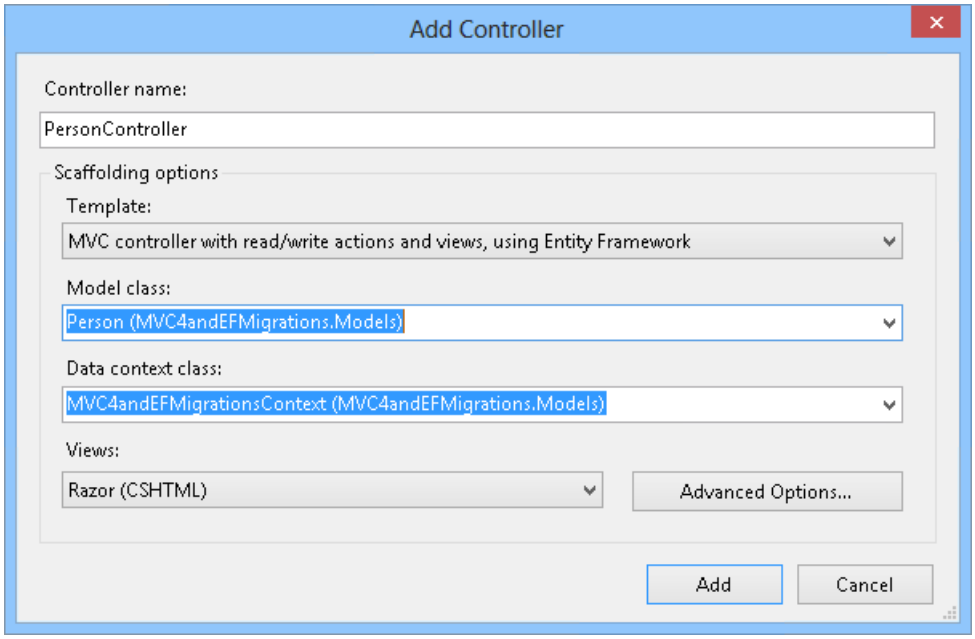
```
Update-Database -Verbose
```



Updating the Database

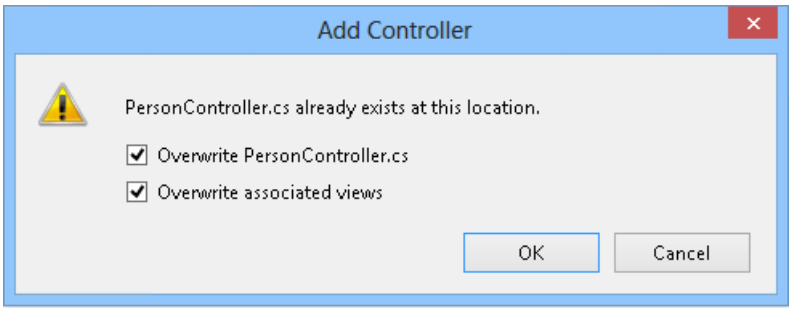
This will add the **MiddleName** column in the **People** table to match the current definition of the **Person** class.

9. Once the database is updated, right-click the Controller folder and select **Add | Controller** to add the Person controller again (Complete with the same values). This will update the existing methods and views adding the new attribute.



Updating the controller

10. Click **Add**. Then, select the values **Overwrite PersonController.cs** and the **Overwrite associated views** and click **OK**.

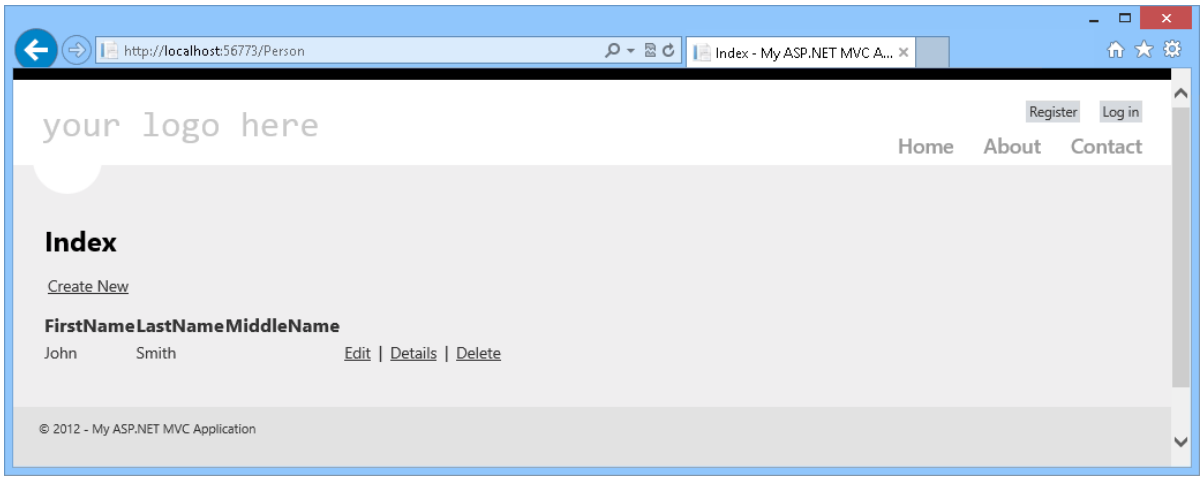


Updating the controller

Task4- Running the application

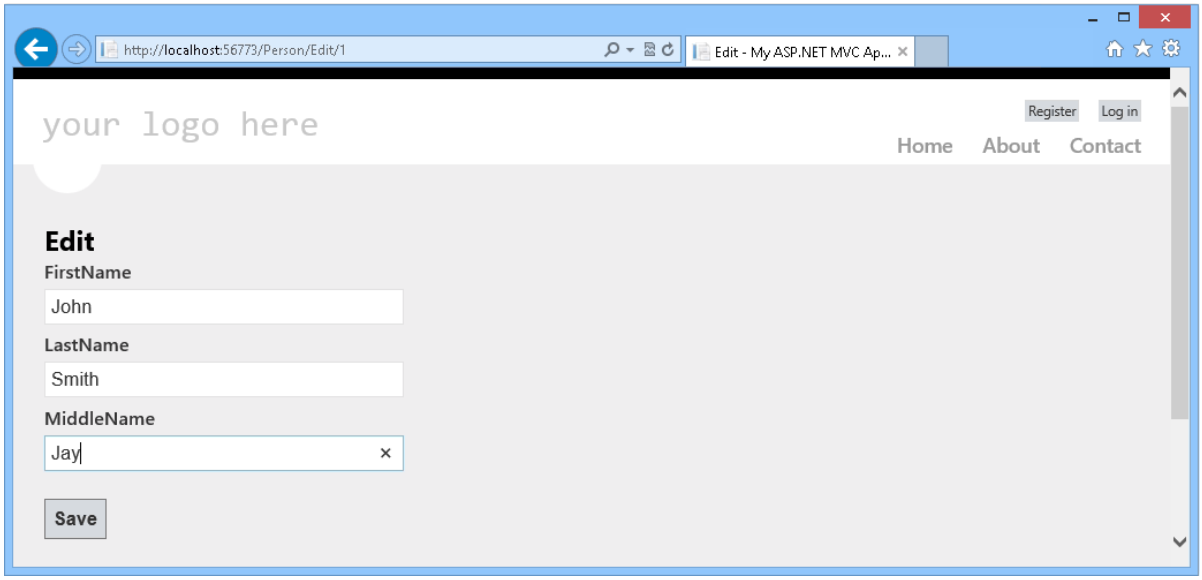
1. Press **F5** to run the application.

2. Open **/Person**. Notice that the data was preserved, while the middle name column was added.



Middle Name added

3. If you click **Edit**, you will be able to add a middle name to the current person.



Summary

In this Hands-On lab, you have learned simple steps to create CRUD operations with ASP.NET MVC 4 Scaffolding using any model class. Then, you have learned how to perform an end to end update in your application -from the database to the views- by using Entity Framework Migrations.

Appendix A: Installing Visual Studio Express 2012 for Web

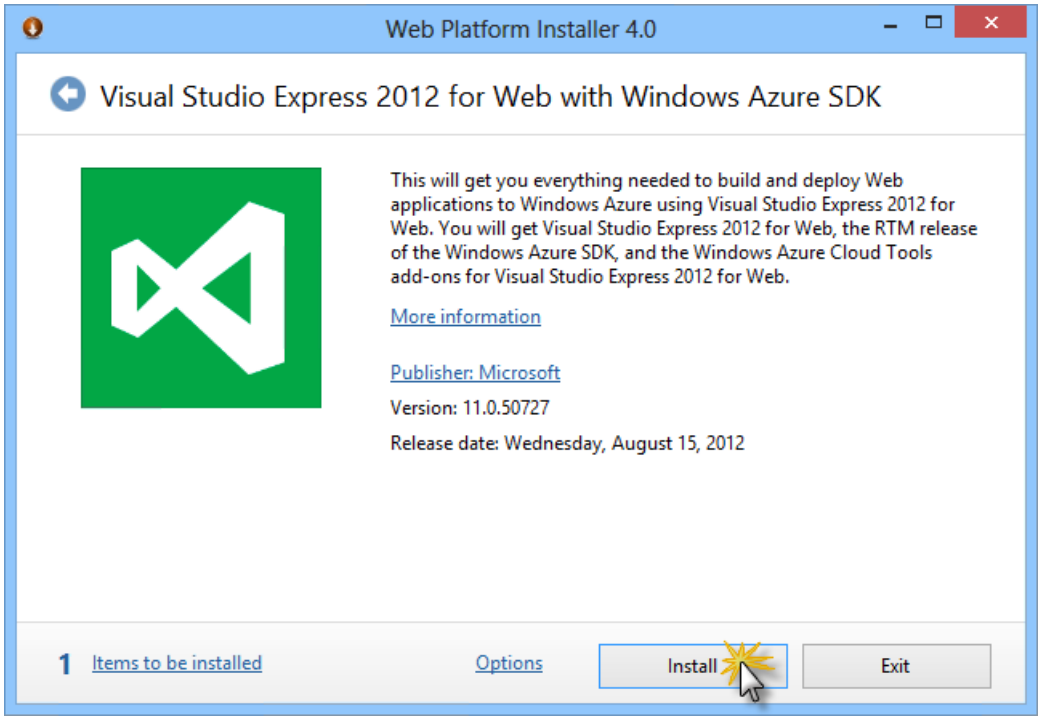
You can install **Microsoft Visual Studio Express 2012 for Web** or another "Express" version using the **Microsoft Web Platform Installer**. The following instructions guide you through the steps required to install *Visual studio Express 2012 for Web* using *Microsoft Web Platform Installer*.

1. Go to https://go.microsoft.com/?linkid=9810169. Alternatively, if you already have installed Web Platform Installer, you can open it and search for the product "*Visual Studio Express 2012 for Web with Windows Azure SDK*".

2. Click on **Install Now**. If you do not have **Web Platform Installer** you will be redirected to download and

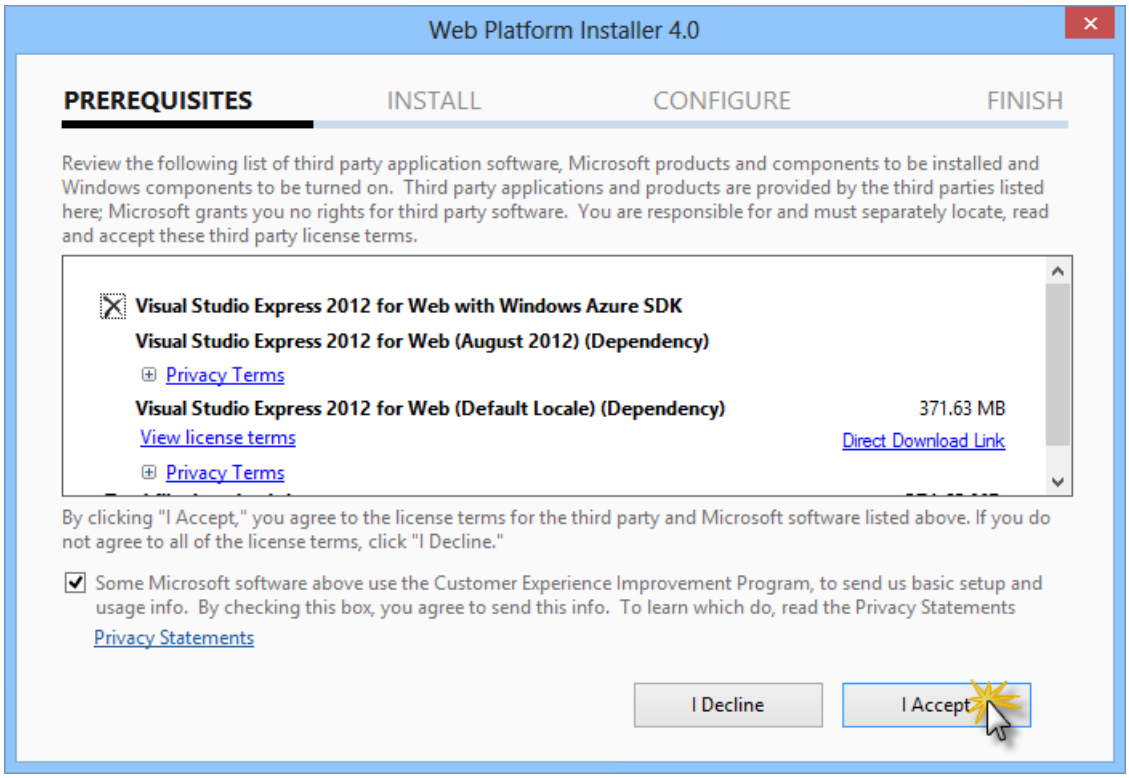
install it first.

3. Once **Web Platform Installer** is open, click **Install** to start the setup.



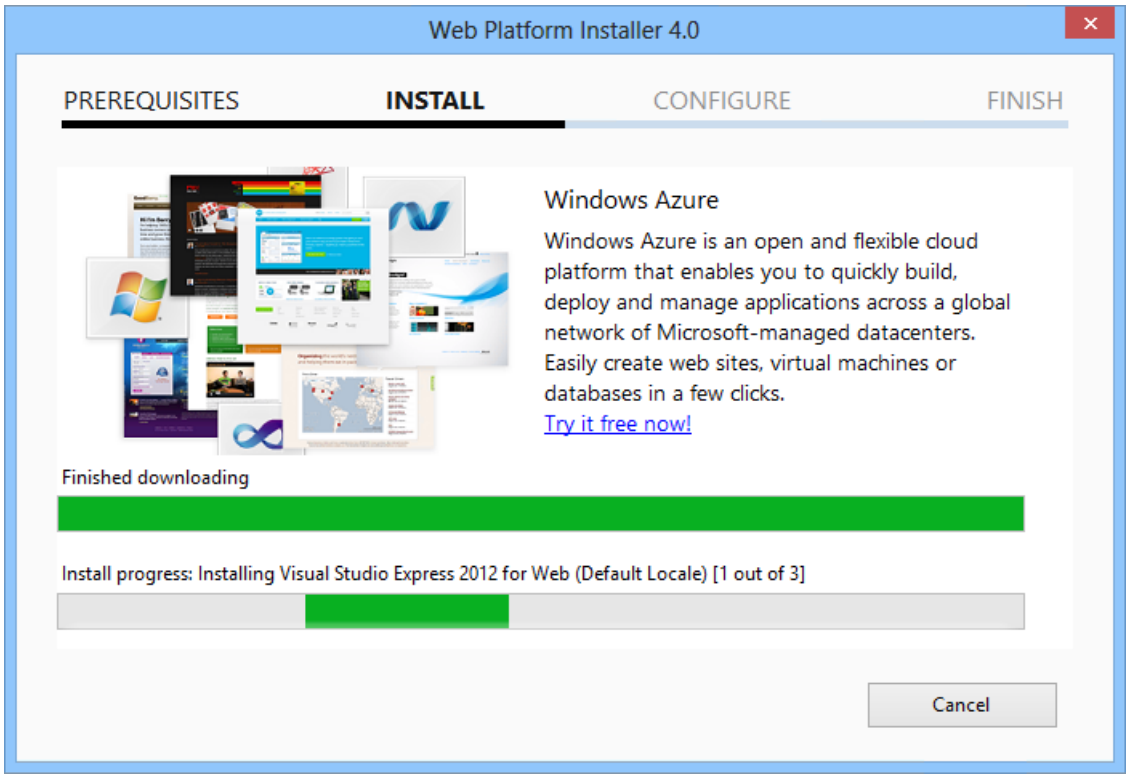
Install Visual Studio Express

4. Read all the products' licenses and terms and click **I Accept** to continue.



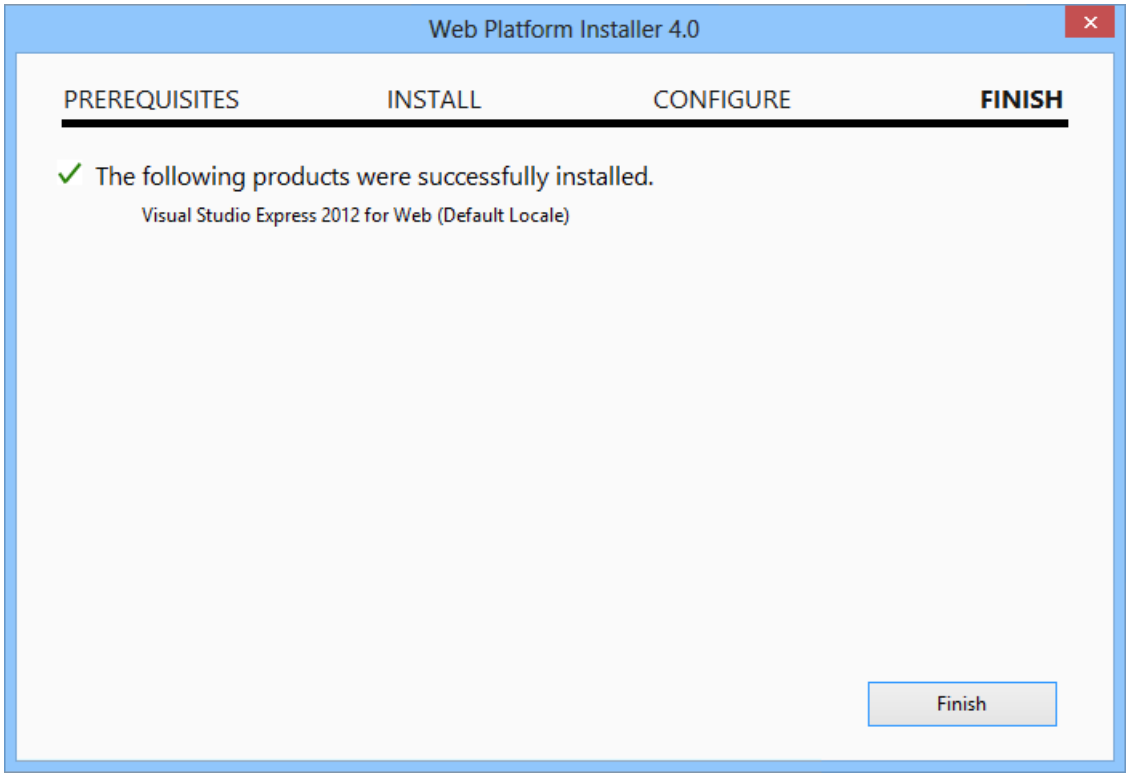
Accepting the license terms

5. Wait until the downloading and installation process completes.



Installation progress

6. When the installation completes, click **Finish**.



Installation completed

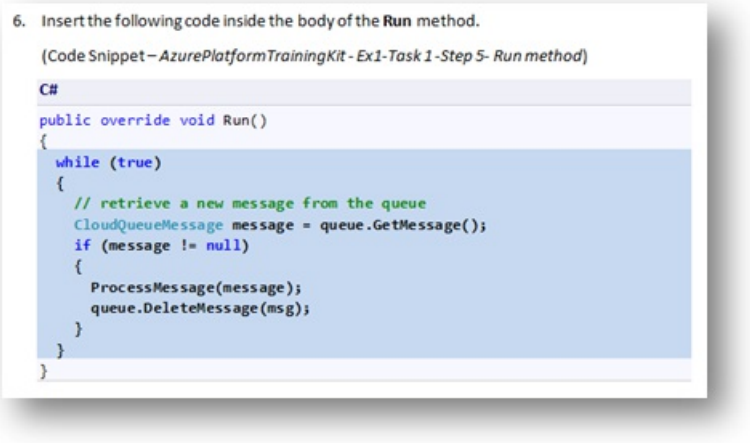
7. Click **Exit** to close Web Platform Installer.

8. To open Visual Studio Express for Web, go to the **Start** screen and start writing "**VS Express**", then click on the **VS Express for Web** tile.

VS Express for Web tile

Appendix B: Using Code Snippets

With code snippets, you have all the code you need at your fingertips. The lab document will tell you exactly when you can use them, as shown in the following figure.



6. Insert the following code inside the body of the **Run** method.

(Code Snippet – *AzurePlatformTrainingKit - Ex1-Task 1 -Step 5- Run method*)

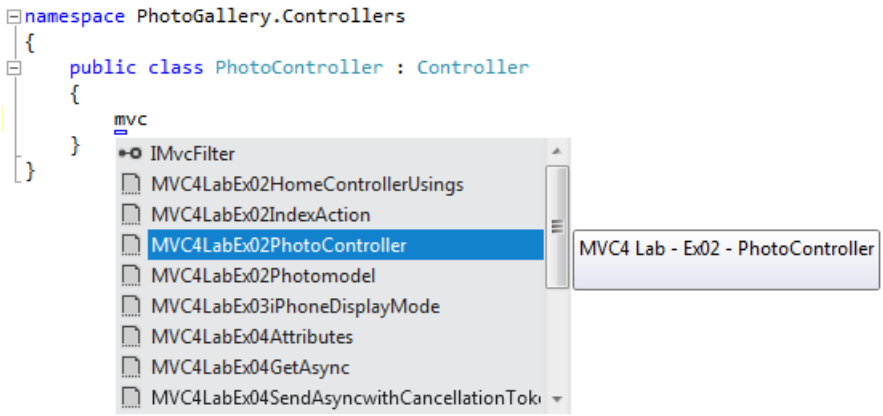
C#

```
public override void Run()
{
  while (true)
  {
    // retrieve a new message from the queue
    CloudQueueMessage message = queue.GetMessage();
    if (message != null)
    {
      ProcessMessage(message);
      queue.DeleteMessage(msg);
    }
  }
}
```

Using Visual Studio code snippets to insert code into your project

To add a code snippet using the keyboard (C# only)

1. Place the cursor where you would like to insert the code.
2. Start typing the snippet name (without spaces or hyphens).
3. Watch as IntelliSense displays matching snippets' names.
4. Select the correct snippet (or keep typing until the entire snippet's name is selected).
5. Press the Tab key twice to insert the snippet at the cursor location.



Start typing the snippet name

```
static void Main(string[] args)
{
    AzureTrainingKitServiceBus
}
```

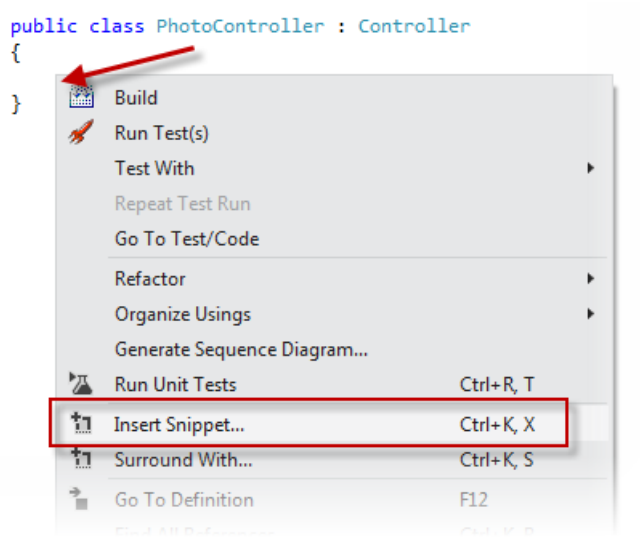
Press Tab to select the highlighted snippet

```
static void Main(string[] args)
{
    request.Headers.Add("$accountKey", accountKey);
    request.Headers.Add("$uniqueUserID", uniqueUserId);
}
```

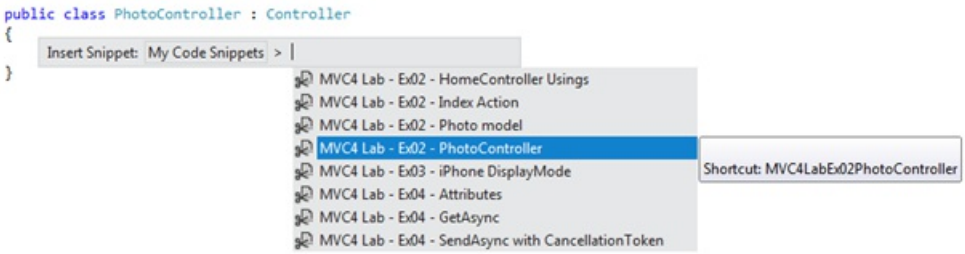
Press Tab again and the snippet will expand

To add a code snippet using the mouse (C#, Visual Basic and XML) 1. Right-click where you want to insert the code snippet.

1. Select **Insert Snippet** followed by **My Code Snippets**.
2. Pick the relevant snippet from the list, by clicking on it.



Right-click where you want to insert the code snippet and select Insert Snippet



Pick the relevant snippet from the list, by clicking on it

ASP.NET MVC 4 Dependency Injection

2/20/2020 • 22 minutes to read • Edit Online

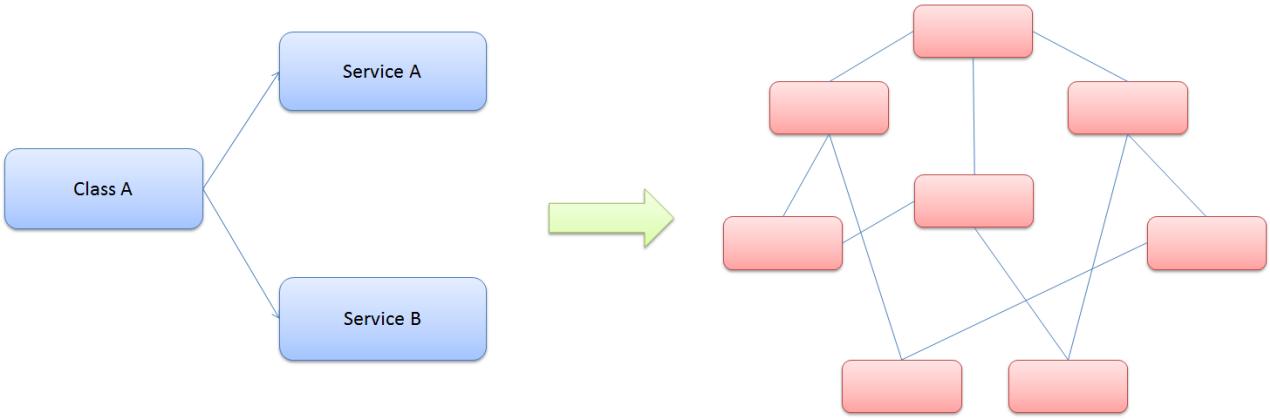
By Web Camps Team

Download Web Camps Training Kit

This Hands-on Lab assumes you have basic knowledge of **ASP.NET MVC** and **ASP.NET MVC 4 filters**. If you have not used **ASP.NET MVC 4 filters** before, we recommend you to go over **ASP.NET MVC Custom Action Filters** Hands-on Lab.

> **NOTE**
>
> All sample code and snippets are included in the Web Camps Training Kit, available from at Microsoft-Web/WebCampTrainingKit Releases. The project specific to this lab is available at ASP.NET MVC 4 Dependency Injection.

In **Object Oriented Programming** paradigm, objects work together in a collaboration model where there are contributors and consumers. Naturally, this communication model generates dependencies between objects and components, becoming difficult to manage when complexity increases.



Class dependencies and model complexity

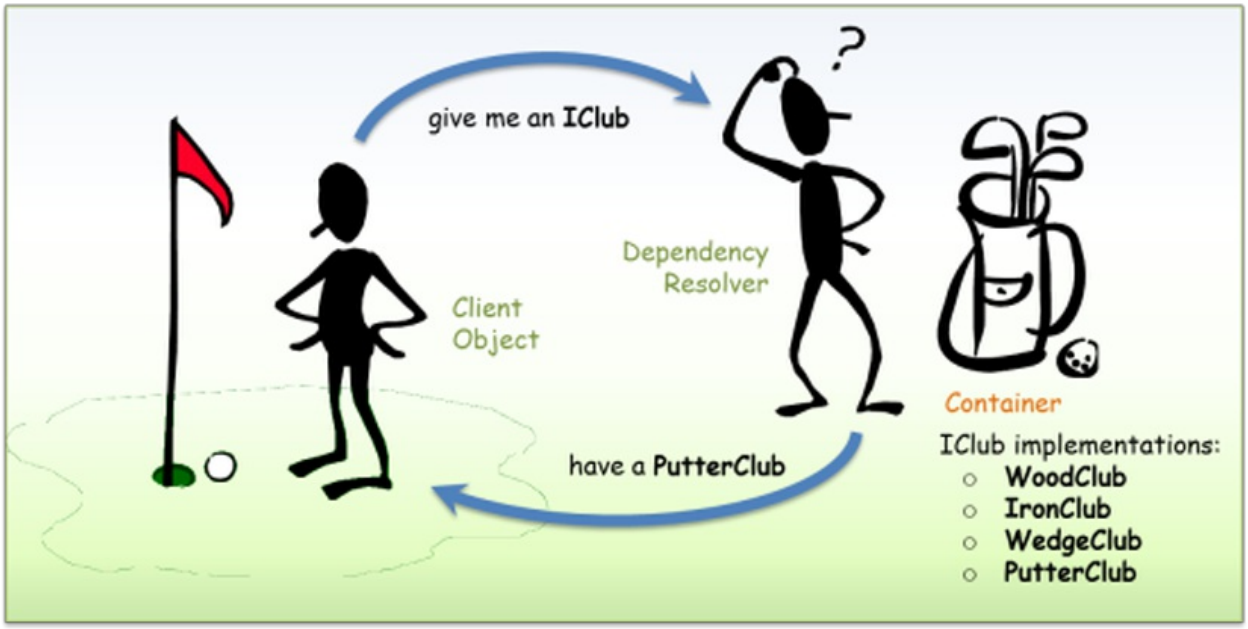
You have probably heard about the **Factory Pattern** and the separation between the interface and the implementation using services, where the client objects are often responsible for service location.

The Dependency Injection pattern is a particular implementation of Inversion of Control. **Inversion of Control (IoC)** means that objects do not create other objects on which they rely to do their work. Instead, they get the objects that they need from an outside source (for example, an xml configuration file).

Dependency Injection (DI) means that this is done without the object intervention, usually by a framework component that passes constructor parameters and set properties.

The Dependency Injection (DI) Design Pattern

At a high level, the goal of Dependency Injection is that a client class (e.g. *the golfer*) needs something that satisfies an interface (e.g. *IClub*). It doesn't care what the concrete type is (e.g. *WoodClub, IronClub, WedgeClub* or *PutterClub*), it wants someone else to handle that (e.g. a good *caddy*). The Dependency Resolver in ASP.NET MVC can allow you to register your dependency logic somewhere else (e.g. a container or a *bag of clubs*).



Dependency Injection - Golf analogy

The advantages of using Dependency Injection pattern and Inversion of Control are the following:

- Reduces class coupling
- Increases code reusing
- Improves code maintainability
- Improves application testing

> **NOTE**
> Dependency Injection is sometimes compared with Abstract Factory Design Pattern, but there is a slight difference between both approaches. DI has a Framework working behind to solve dependencies by calling the factories and the registered services.

Now that you understand the Dependency Injection Pattern, you will learn throughout this lab how to apply it in ASP.NET MVC 4. You will start using Dependency Injection in the **Controllers** to include a database access service. Next, you will apply Dependency Injection to the **Views** to consume a service and show information. Finally, you will extend the DI to ASP.NET MVC 4 Filters, injecting a custom action filter in the solution.

In this Hands-on Lab, you will learn how to:

- Integrate ASP.NET MVC 4 with Unity for Dependency Injection using NuGet Packages
- Use Dependency Injection inside an ASP.NET MVC Controller
- Use Dependency Injection inside an ASP.NET MVC View
- Use Dependency Injection inside an ASP.NET MVC Action Filter

> **NOTE**
> This Lab is using Unity.Mvc3 NuGet Package for dependency resolution, but it is possible to adapt any Dependency Injection Framework to work with ASP.NET MVC 4.

Prerequisites

You must have the following items to complete this lab:

- Microsoft Visual Studio Express 2012 for Web or superior (read Appendix A for instructions on how to install it).

Setup

Installing Code Snippets

For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. To install the code snippets run **.\Source\Setup\CodeSnippets.vsi** file.

If you are not familiar with the Visual Studio Code Snippets, and want to learn how to use them, you can refer to the appendix from this document "Appendix B: Using Code Snippets".

Exercises

This Hands-On Lab is comprised by the following exercises:

1. Exercise 1: Injecting a Controller
2. Exercise 2: Injecting a View
3. Exercise 3: Injecting Filters

> **NOTE**
> Each exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

Estimated time to complete this lab: **30 minutes**.

Exercise 1: Injecting a Controller

In this exercise, you will learn how to use Dependency Injection in ASP.NET MVC Controllers by integrating Unity using a NuGet Package. For that reason, you will include services into your MvcMusicStore controllers to separate the logic from the data access. The services will create a new dependency in the controller constructor, which will be resolved using Dependency Injection with the help of **Unity**.

This approach will show you how to generate less coupled applications, which are more flexible and easier to maintain and test. You will also learn how to integrate ASP.NET MVC with Unity.

About StoreManager Service

The MVC Music Store provided in the begin solution now includes a service that manages the Store Controller data named **StoreService**. Below you will find the Store Service implementation. Note that all the methods return Model entities.

```
namespace MvcMusicStore.Controllers
{
    using System.Web.Mvc;
    using MvcMusicStore.Filters;
    using MvcMusicStore.Services;

    [MyNewCustomActionFilter(Order = 1)]
    [CustomActionFilter(Order = 2)]
    public class StoreController : Controller
    {
        private IStoreService service;

        public StoreController(IStoreService service)
        {
            this.service = service;
        }

        // GET: /Store/
        public ActionResult Details(int id)
        {
            var album = this.service.GetAlbum(id);
            if (album == null)
            {
                return this.HttpNotFound();
            }

            return this.View(album);
        }

        public ActionResult Browse(string genre)
        {
            // Retrieve Genre and its Associated Albums from database
            var genreModel = this.service.GetGenreByName(genre);

            return this.View(genreModel);
        }

        public ActionResult Index()
        {
            var genres = this.service.GetGenres();

            return this.View(genres);
        }

        // GET: /Store/GenreMenu
        public ActionResult GenreMenu()
        {
            var genres = this.service.GetGenres();

            return this.PartialView(genres);
        }
    }
}
```

StoreController from the begin solution now consumes **StoreService**. All the data references were removed from **StoreController**, and now possible to modify the current data access provider without changing any method that consumes **StoreService**.

You will find below that the **StoreController** implementation has a dependency with **StoreService** inside the class constructor.

NOTE

The dependency introduced in this exercise is related to **Inversion of Control** (IoC).

The **StoreController** class constructor receives an **IStoreService** type parameter, which is essential to perform service calls from inside the class. However, **StoreController** does not implement the default constructor (with no parameters) that any controller must have to work with ASP.NET MVC.

To resolve the dependency, the controller has to be created by an abstract factory (a class that returns any object of the specified type).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MvcMusicStore.ViewModels;
using MvcMusicStore.Models;
using MvcMusicStore.Services;

namespace MvcMusicStore.Controllers
{
    public class StoreController : Controller
    {
        private IStoreService service;

        public StoreController(IStoreService service)
        {
            this.service = service;
        }

        //
        // GET: /Store/
        public ActionResult Index()
        {
            // Create list of genres
            var genres = this.service.GetGenreNames();

            // Create your view model
            var viewModel = new StoreIndexViewModel
            {
                Genres = genres.ToList(),
                NumberOfGenres = genres.Count()
            };

            return View(viewModel);
        }

        //
        // GET: /Store/Browse?genre=Disco
        public ActionResult Browse(string genre)
        {
            var genreModel = this.service.GetGenreByName(genre);

            var viewModel = new StoreBrowseViewModel()
            {
                Genre = genreModel,
                Albums = genreModel.Albums.ToList()
            };

            return View(viewModel);
        }

        //
        // GET: /Store/Details/5
        public ActionResult Details(int id)
        {
            var album = this.service.GetAlbum(id);

            return View(album);
        }
    }
}
```

> **NOTE**
> You will get an error when the class tries to create the StoreController without sending the service object, as there is no parameterless constructor declared.

Task 1 - Running the Application

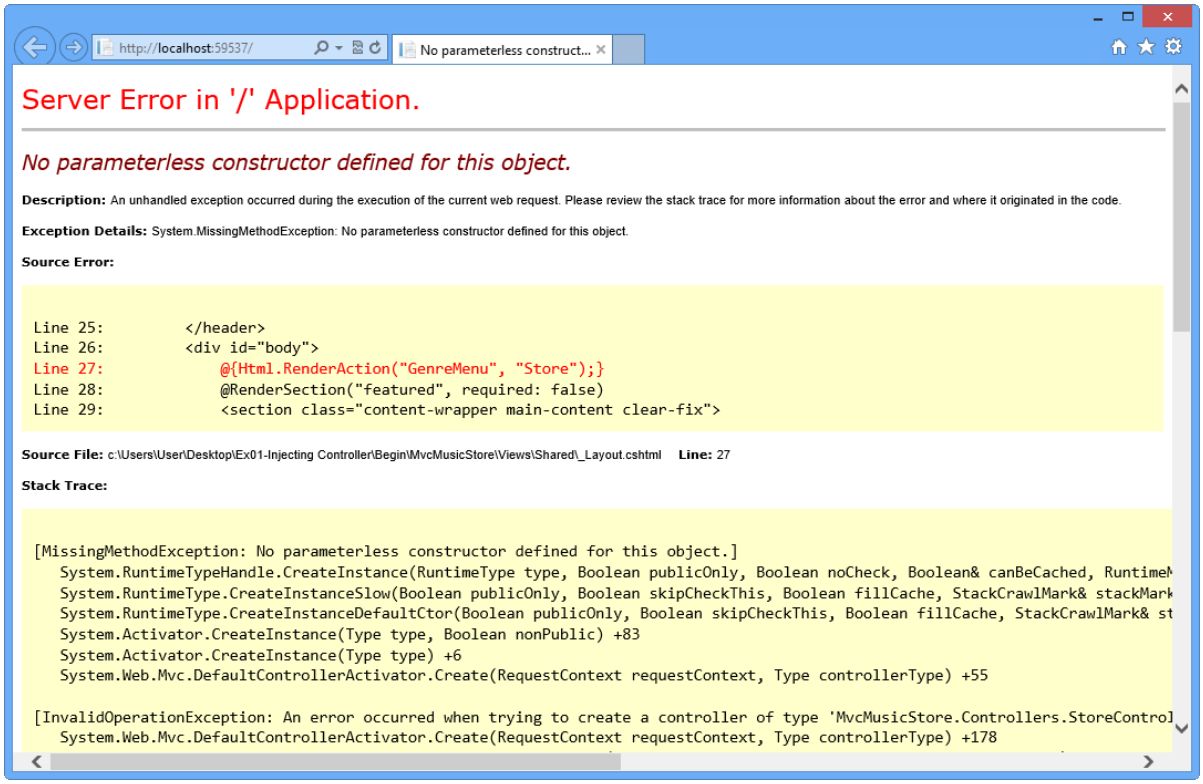
In this task, you will run the Begin application, which includes the service into the Store Controller that separates the data access from the application logic.

When running the application, you will receive an exception, as the controller service is not passed as a parameter by default:

1. Open the **Begin** solution located in **Source\Ex01-Injecting Controller\Begin**.
   a. You will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.
   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.
   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Press **Ctrl + F5** to run the application without debugging. You will get the error message "**No parameterless constructor defined for this object**":



Error while running ASP.NET MVC Begin Application

3. Close the browser.

In the following steps you will work on the Music Store Solution to inject the dependency this controller needs.

Task 2 - Including Unity into MvcMusicStore Solution

In this task, you will include **Unity.Mvc3** NuGet Package to the solution.

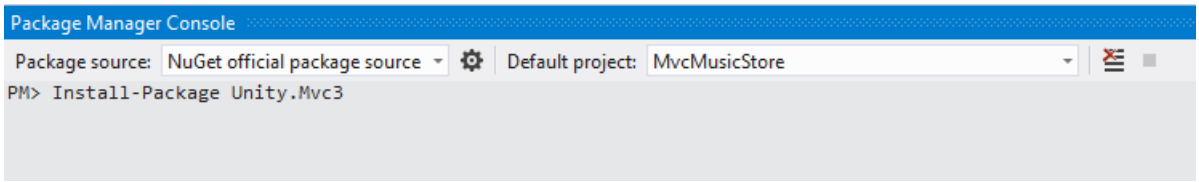
> **NOTE**
>
> Unity.Mvc3 package was designed for ASP.NET MVC 3, but it is fully compatible with ASP.NET MVC 4.
>
> Unity is a lightweight, extensible dependency injection container with optional support for instance and type interception. It is a general-purpose container for use in any type of .NET application. It provides all the common features found in dependency injection mechanisms including: object creation, abstraction of requirements by specifying dependencies at runtime and flexibility, by deferring the component configuration to the container.

1. Install **Unity.Mvc3** NuGet Package in the **MvcMusicStore** project. To do this, open the **Package Manager Console** from **View | Other Windows**.

2. Run the following command.

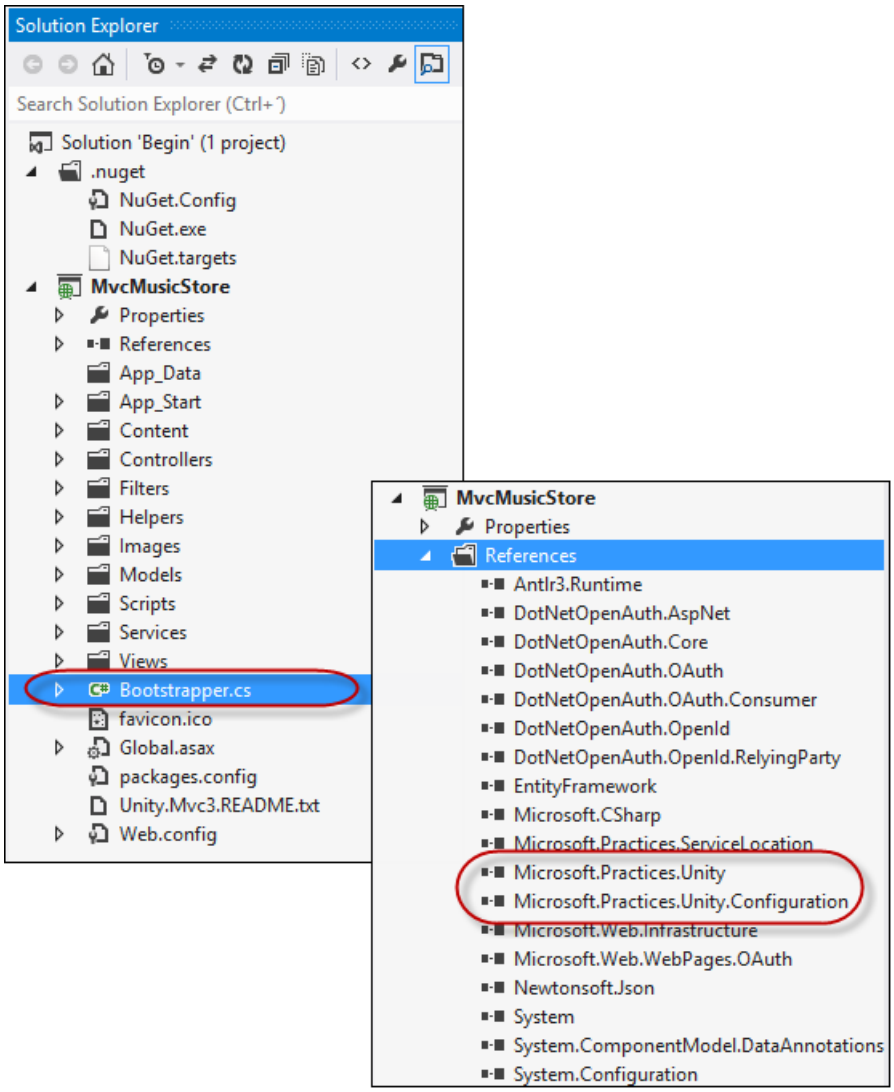
   PMC

   ```
   Install-Package Unity.Mvc3
   ```

   Installing Unity.Mvc3 NuGet Package

3. Once the **Unity.Mvc3** package is installed, explore the files and folders it automatically adds in order to simplify Unity configuration.



Unity.Mvc3 package installed

Task 3 - Registering Unity in Global.asax.cs Application_Start

In this task, you will update the **Application_Start** method located in **Global.asax.cs** to call the Unity Bootstrapper initializer and then, update the Bootstrapper file registering the Service and Controller you will use for Dependency Injection.

1. Now, you will hook up the Bootstrapper which is the file that initializes the Unity container and Dependency Resolver. To do this, open **Global.asax.cs** and add the following highlighted code within the **Application_Start** method.

   (Code Snippet - *ASP.NET Dependency Injection Lab - Ex01 - Initialize Unity*)

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();

    WebApiConfig.Register(GlobalConfiguration.Configuration);
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);

    Bootstrapper.Initialise();

    AppConfig.Configure();
}
```

2. Open **Bootstrapper.cs** file.

3. Include the following namespaces: **MvcMusicStore.Services** and **MusicStore.Controllers**.

   (Code Snippet - *ASP.NET Dependency Injection Lab - Ex01 - Bootstrapper Adding Namespaces*)

```
using System.Web.Mvc;
using Microsoft.Practices.Unity;
using Unity.Mvc3;
using MvcMusicStore.Services;
using MvcMusicStore.Controllers;
```

4. Replace **BuildUnityContainer** method's content with the following code that registers Store Controller and Store Service.

   (Code Snippet - *ASP.NET Dependency Injection Lab - Ex01 - Register Store Controller and Service*)

```
private static IUnityContainer BuildUnityContainer()
{
    var container = new UnityContainer();

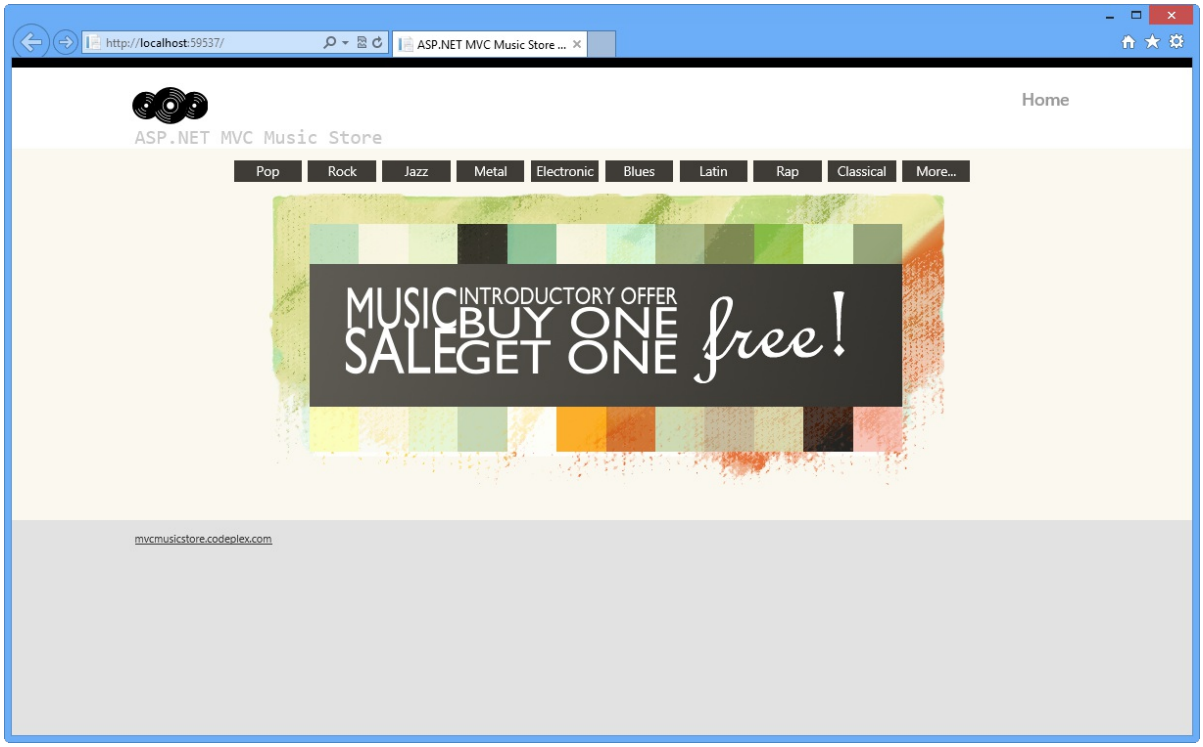
    container.RegisterType<IStoreService, StoreService>();
    container.RegisterType<IController, StoreController>("Store");

    return container;
}
```

Task 4 - Running the Application

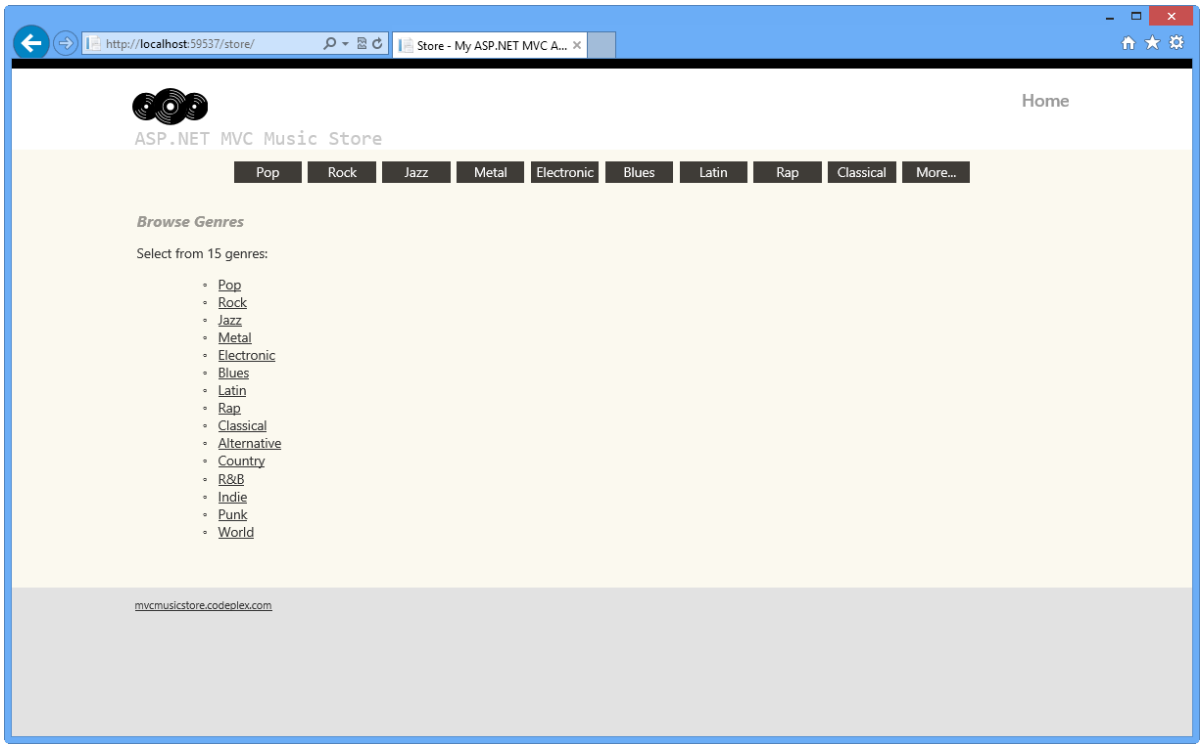
In this task, you will run the application to verify that it can now be loaded after including Unity.

1. Press **F5** to run the application, the application should now load without showing any error message.



Running Application with Dependency Injection

2. Browse to **/Store**. This will invoke **StoreController**, which is now created using **Unity**.



MVC Music Store

3. Close the browser.

In the following exercises you will learn how to extend the Dependency Injection scope to use it inside ASP.NET MVC Views and Action Filters.

Exercise 2: Injecting a View

In this exercise, you will learn how to use Dependency Injection in a view with the new features of ASP.NET MVC 4 for Unity integration. In order to do that, you will call a custom service inside the Store Browse View, which will show a message and an image below.

Then, you will integrate the project with Unity and create a custom dependency resolver to inject the dependencies.

Task 1 - Creating a View that Consumes a Service

In this task, you will create a view that performs a service call to generate a new dependency. The service consists in a simple messaging service included in this solution.

1. Open the **Begin** solution located in the **Source\Ex02-Injecting View\Begin** folder. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.

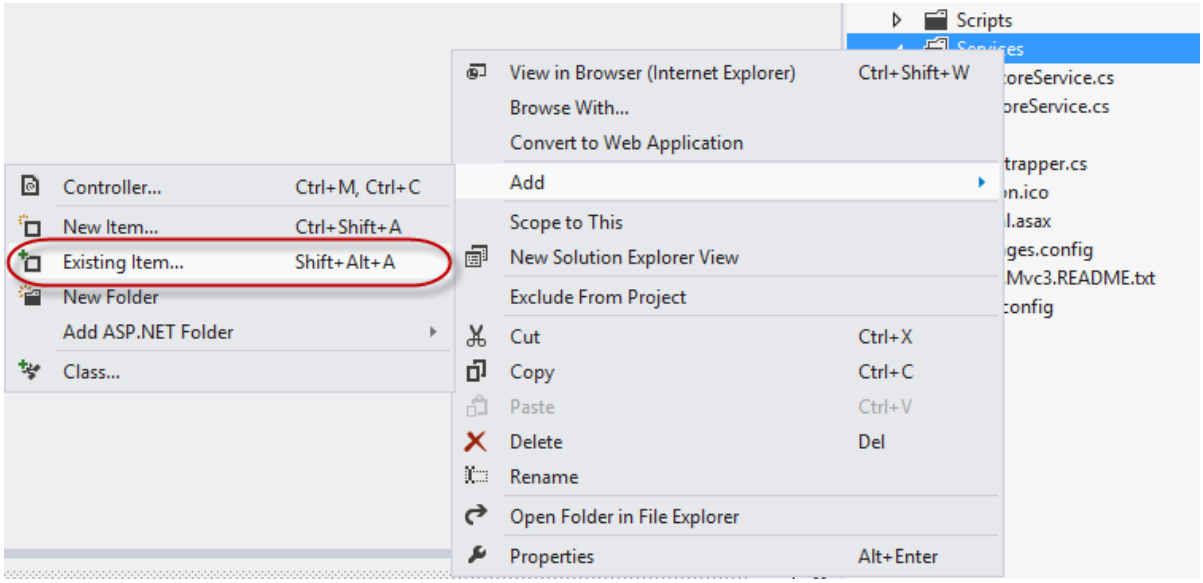
   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

> **NOTE**
>
> One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.
>
> For more information, see this article: http://docs.nuget.org/docs/workflows/using-nuget-without-committing-packages.

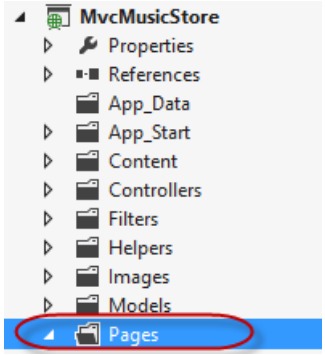
2. Include the **MessageService.cs** and the **IMessageService.cs** classes located in the **Source \Assets** folder in **/Services**. To do this, right-click **Services** folder and select **Add Existing Item**. Browse to the files' location and include them.



Adding Message Service and Service Interface

> **NOTE**
>
> The **IMessageService** interface defines two properties implemented by the **MessageService** class. These properties -**Message** and **ImageUrl**- store the message and the URL of the image to be displayed.

3. Create the folder **/Pages** in the project's root folder, and then add the existing class **MyBasePage.cs** from **Source\Assets**. The base page you will inherit from has the following structure.



```
namespace MvcMusicStore.Pages
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Web;
    using Microsoft.Practices.Unity;
    using MvcMusicStore.Models;
    using MvcMusicStore.Services;

    public class MyBasePage : System.Web.Mvc.WebViewPage<Genre>
    {
        [Dependency]
        public IMessageService MessageService { get; set; }

        public override void

        Execute()
        {
        }
    }
}
```

4. Open **Browse.cshtml** view from **/Views/Store** folder, and make it inherit from **MyBasePage.cs**.

```
@inherits MvcMusicStore.Pages.MyBasePage
@{
     ViewBag.Title = "Browse Albums";
}
```

5. In the **Browse** view, add a call to **MessageService** to display an image and a message retrieved by the service. (C#)

```
@inherits MvcMusicStore.Pages.MyBasePage
@{
    Viewbag.Title = "Browse Albums";
}
<div>
    @this.MessageService.Message
    <br />
    <img alt="@this.MessageService.Message" src="@this.MessageService.ImageUrl" />
</div>
...
```

Task 2 - Including a Custom Dependency Resolver and a Custom View Page Activator

In the previous task, you injected a new dependency inside a view to perform a service call inside it. Now, you will resolve that dependency by implementing the ASP.NET MVC Dependency Injection interfaces **IViewPageActivator** and **IDependencyResolver**. You will include in the solution an implementation of **IDependencyResolver** that will deal with the service retrieval by using Unity. Then, you will include another custom implementation of **IViewPageActivator** interface that will solve the creation of the views.

NOTE

Since ASP.NET MVC 3, the implementation for Dependency Injection had simplified the interfaces to register services. **IDependencyResolver** and **IViewPageActivator** are part of ASP.NET MVC 3 features for Dependency Injection.

- **IDependencyResolver** interface replaces the previous IMvcServiceLocator. Implementers of IDependencyResolver must return an instance of the service or a service collection.

```
public interface IDependencyResolver {
    object GetService(Type serviceType);
    IEnumerable<object> GetServices(Type serviceType);
}
```

- **IViewPageActivator** interface provides more fine-grained control over how view pages are instantiated via dependency injection. The classes that implement **IViewPageActivator** interface can create view instances using context information.

```
public interface IViewPageActivator {
    object Create(ControllerContext controllerContext, Type type);
}
```

1. Create the **/Factories** folder in the project's root folder.

2. Include **CustomViewPageActivator.cs** to your solution from **/Sources/Assets/** to **Factories** folder. To do that, right-click the **/Factories** folder, select **Add | Existing Item** and then select **CustomViewPageActivator.cs**. This class implements the **IViewPageActivator** interface to hold the Unity Container.

```
namespace MvcMusicStore.Factories
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Web;
    using System.Web.Mvc;
    using Microsoft.Practices.Unity;

    public class CustomViewPageActivator : IViewPageActivator
    {
        private IUnityContainer container;

        public CustomViewPageActivator(IUnityContainer container)
        {
            this.container = container;
        }

        public object Create(ControllerContext controllerContext, Type type)
        {
            return this.container.Resolve(type);
        }
    }
}
```

NOTE

CustomViewPageActivator is responsible for managing the creation of a view by using a Unity container.

3. Include **UnityDependencyResolver.cs** file from **/Sources/Assets** to **/Factories** folder. To do that, right-click the **/Factories** folder, select **Add | Existing Item** and then select **UnityDependencyResolver.cs** file.

```
namespace MvcMusicStore.Factories
{
     using System;
     using System.Collections.Generic;
     using System.Linq;
     using System.Web;
     using System.Web.Mvc;
     using Microsoft.Practices.Unity;

     public class UnityDependencyResolver : IDependencyResolver
     {
          private IUnityContainer container;

          private IDependencyResolver resolver;

          public UnityDependencyResolver(IUnityContainer container, IDependencyResolver resolver)
          {
                this.container = container;
                this.resolver = resolver;
          }

          public object GetService(Type serviceType)
          {
                try
                {
                     return this.container.Resolve(serviceType);
                }
                catch
                {
                     return this.resolver.GetService(serviceType);
                }
          }

          public IEnumerable<object> GetServices(Type serviceType)
          {
                try
                {
                     return this.container.ResolveAll(serviceType);
                }
                catch
                {
                     return this.resolver.GetServices(serviceType);
                }
          }
     }
}
```

NOTE

UnityDependencyResolver class is a custom DependencyResolver for Unity. When a service cannot be found inside the Unity container, the base resolver is invocated.

In the following task both implementations will be registered to let the model know the location of the services and the views.

Task 3 - Registering for Dependency Injection within Unity container

In this task, you will put all the previous things together to make Dependency Injection work.

Up to now your solution has the following elements:

- A **Browse** View that inherits from **MyBaseClass** and consumes **MessageService**.
- An intermediate class -**MyBaseClass**- that has dependency injection declared for the service interface.
- A service - **MessageService** - and its interface **IMessageService**.

- A custom dependency resolver for Unity - **UnityDependencyResolver** - that deals with service retrieval.
- A View Page activator - **CustomViewPageActivator** - that creates the page.

To inject **Browse** View, you will now register the custom dependency resolver in the Unity container.

1. Open **Bootstrapper.cs** file.

2. Register an instance of **MessageService** into the Unity container to initialize the service:

   (Code Snippet - *ASP.NET Dependency Injection Lab - Ex02 - Register Message Service*)

```
private static IUnityContainer BuildUnityContainer()
{
    var container = new UnityContainer();

    container.RegisterType<IStoreService, StoreService>();
    container.RegisterType<IController, StoreController>("Store");

    container.RegisterInstance<IMessageService>(new MessageService
    {
        Message = "You are welcome to our Web Camps Training Kit!",
        ImageUrl = "/Content/Images/webcamps.png"
    });
    //...
}
```

3. Add a reference to **MvcMusicStore.Factories** namespace.

   (Code Snippet - *ASP.NET Dependency Injection Lab - Ex02 - Factories Namespace*)

```
using System.Web.Mvc;
using Microsoft.Practices.Unity;
using Unity.Mvc3;
using MvcMusicStore.Services;
using MvcMusicStore.Controllers;
using MvcMusicStore.Factories;
```

4. Register **CustomViewPageActivator** as a View Page activator into the Unity container:

   (Code Snippet - *ASP.NET Dependency Injection Lab - Ex02 - Register CustomViewPageActivator*)

```
private static IUnityContainer BuildUnityContainer()
{
    var container = new UnityContainer();

    container.RegisterType<IStoreService, StoreService>();
    container.RegisterType<IController, StoreController>("Store");

    container.RegisterInstance<IMessageService>(new MessageService
    {
        Message = "You are welcome to our Web Camps Training Kit!",
        ImageUrl = "/Content/Images/webcamps.png"
    });

    container.RegisterType<IViewPageActivator, CustomViewPageActivator>(new
InjectionConstructor(container));

    return container;
}
```

5. Replace ASP.NET MVC 4 default dependency resolver with an instance of **UnityDependencyResolver**. To

do this, replace **Initialize** method content with the following code:

(Code Snippet - *ASP.NET Dependency Injection Lab - Ex02 - Update Dependency Resolver*)

```
public static void Initialise()
{
    var container = BuildUnityContainer();

    DependencyResolver.SetResolver(new Unity.Mvc3.UnityDependencyResolver(container));

    IDependencyResolver resolver = DependencyResolver.Current;

    IDependencyResolver newResolver = new Factories.UnityDependencyResolver(container, resolver);

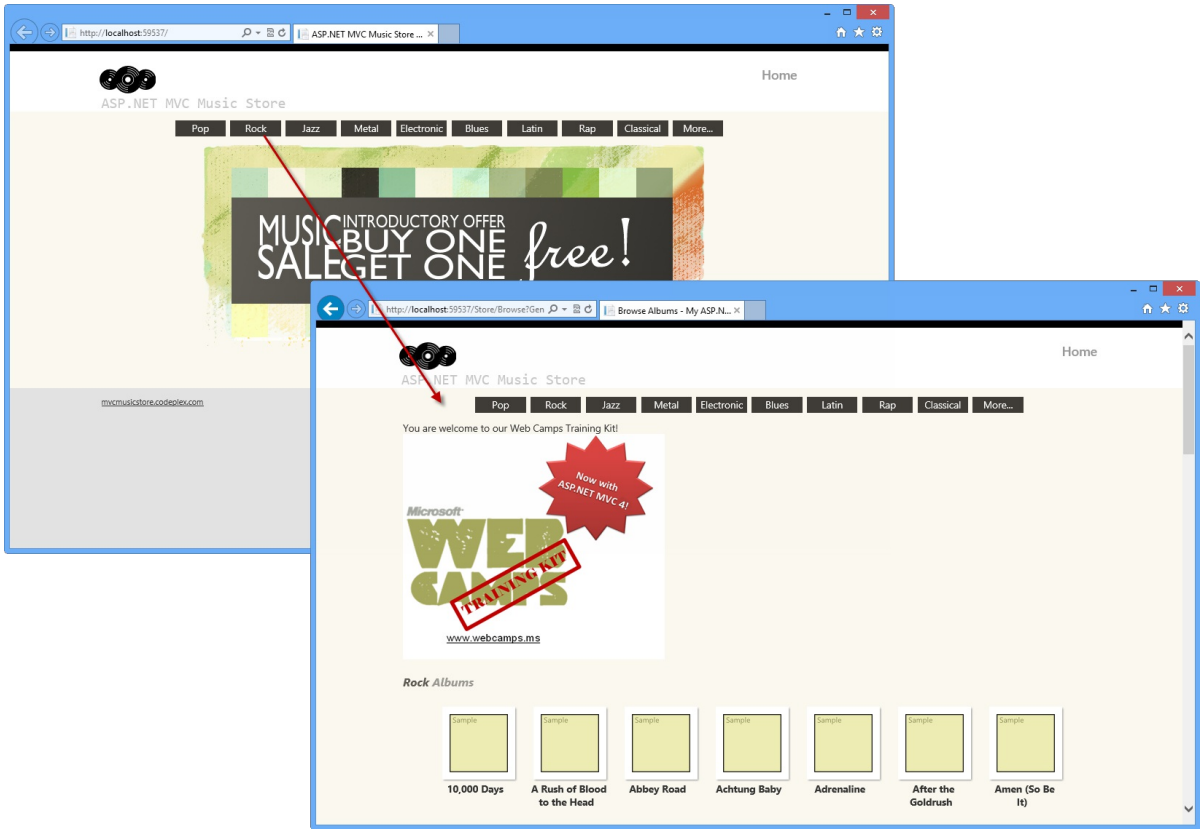
    DependencyResolver.SetResolver(newResolver);
}
```

> **NOTE**
>
> ASP.NET MVC provides a default dependency resolver class. To work with custom dependency resolvers as the one we have created for unity, this resolver has to be replaced.

Task 4 - Running the Application

In this task, you will run the application to verify that the Store Browser consumes the service and shows the image and the message retrieved:

1. Press **F5** to run the application.
2. Click **Rock** within the Genres Menu and see how the **MessageService** was injected to the view and loaded the welcome message and the image. In this example, we are entering to "**Rock**":



MVC Music Store - View Injection

3. Close the browser.

Exercise 3: Injecting Action Filters

In the previous Hands-On lab **Custom Action Filters** you have worked with filters customization and injection. In this exercise, you will learn how to inject filters with Dependency Injection by using the Unity container. To do that, you will add to the Music Store solution a custom action filter that will trace the activity of the site.

Task 1 - Including the Tracking Filter in the Solution

In this task, you will include in the Music Store a custom action filter to trace events. As custom action filter concepts are already treated in the previous Lab "Custom Action Filters", you will just include the filter class from the Assets folder of this lab, and then create a Filter Provider for Unity:

1. Open the **Begin** solution located in the **Source\Ex03 - Injecting Action Filter\Begin** folder. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.

   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   >
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.
   >
   > For more information, see this article: http://docs.nuget.org/docs/workflows/using-nuget-without-committing-packages.

2. Include **TraceActionFilter.cs** file from **/Sources/Assets** to **/Filters** folder.

```
namespace MvcMusicStore.Filters
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Web;
    using System.Web.Mvc;

    public class TraceActionFilter : IActionFilter
    {
        public void OnActionExecuted(ActionExecutedContext filterContext)
        {
            filterContext.HttpContext.Trace.Write("OnActionExecuted");
            filterContext.HttpContext.Trace.Write("Action " +
filterContext.ActionDescriptor.ActionName);
            filterContext.HttpContext.Trace.Write("Controller " +
filterContext.ActionDescriptor.ControllerDescriptor.ControllerName);
        }

        public void OnActionExecuting(ActionExecutingContext filterContext)
        {
            filterContext.HttpContext.Trace.Write("OnActionExecuting");
            filterContext.HttpContext.Trace.Write("Action " +
filterContext.ActionDescriptor.ActionName);
            filterContext.HttpContext.Trace.Write("Controller " +
filterContext.ActionDescriptor.ControllerDescriptor.ControllerName);
        }
    }
}
```

> **NOTE**
> This custom action filter performs ASP.NET tracing. You can check "ASP.NET MVC 4 local and Dynamic Action Filters" Lab for more reference.

3. Add the empty class **FilterProvider.cs** to the project in the folder **/Filters.**

4. Add the **System.Web.Mvc** and **Microsoft.Practices.Unity** namespaces in **FilterProvider.cs**.

   (Code Snippet - *ASP.NET Dependency Injection Lab - Ex03 - Filter Provider Adding Namespaces*)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Microsoft.Practices.Unity;

namespace MvcMusicStore.Filters
{
     public class FilterProvider
     {
     }
}
```

5. Make the class inherit from **IFilterProvider** Interface.

```
namespace MvcMusicStore.Filters
{
    public class FilterProvider : IFilterProvider
    {
    }
}
```

6. Add a **IUnityContainer** property in the **FilterProvider** class, and then create a class constructor to assign the container.

   (Code Snippet - *ASP.NET Dependency Injection Lab - Ex03 - Filter Provider Constructor*)

```
public class FilterProvider : IFilterProvider
{
    private IUnityContainer container;

    public FilterProvider(IUnityContainer container)
    {
        this.container = container;
    }
}
```

> **NOTE**
> The filter provider class constructor is not creating a **new** object inside. The container is passed as a parameter, and the dependency is solved by Unity.

7. In the **FilterProvider** class, implement the method **GetFilters** from **IFilterProvider** interface.

   (Code Snippet - *ASP.NET Dependency Injection Lab - Ex03 - Filter Provider GetFilters*)

```
public class FilterProvider : IFilterProvider
{
    private IUnityContainer container;

    public FilterProvider(IUnityContainer container)
    {
        this.container = container;
    }

    public IEnumerable<Filter> GetFilters(ControllerContext controllerContext, ActionDescriptor
actionDescriptor)
    {
        foreach (IActionFilter actionFilter in this.container.ResolveAll<IActionFilter>())
        {
            yield return new Filter(actionFilter, FilterScope.First, null);
        }
    }
}
```

Task 2 - Registering and Enabling the Filter

In this task, you will enable site tracking. To do that, you will register the filter in **Bootstrapper.cs** **BuildUnityContainer** method to start tracing:

1. Open **Web.config** located in the project root and enable trace tracking at System.Web group.

```
<system.web>
    <trace enabled="true"/>
    <compilation debug="true" targetFramework="4.5">
```

2. Open **Bootstrapper.cs** at project root.

3. Add a reference to the **MvcMusicStore.Filters** namespace.

   (Code Snippet - *ASP.NET Dependency Injection Lab - Ex03 - Bootstrapper Adding Namespaces*)

```
using System.Web.Mvc;
using Microsoft.Practices.Unity;
using Unity.Mvc3;
using MvcMusicStore.Services;
using MvcMusicStore.Controllers;
using MvcMusicStore.Factories;
using MvcMusicStore.Filters;
```

4. Select the **BuildUnityContainer** method and register the filter in the Unity Container. You will have to register the filter provider as well as the action filter.

   (Code Snippet - *ASP.NET Dependency Injection Lab - Ex03 - Register FilterProvider and ActionFilter*)

```
private static IUnityContainer BuildUnityContainer()
{
    var container = new UnityContainer();

    //...

    container.RegisterInstance<IFilterProvider>("FilterProvider", new FilterProvider(container));
    container.RegisterInstance<IActionFilter>("LogActionFilter", new TraceActionFilter());

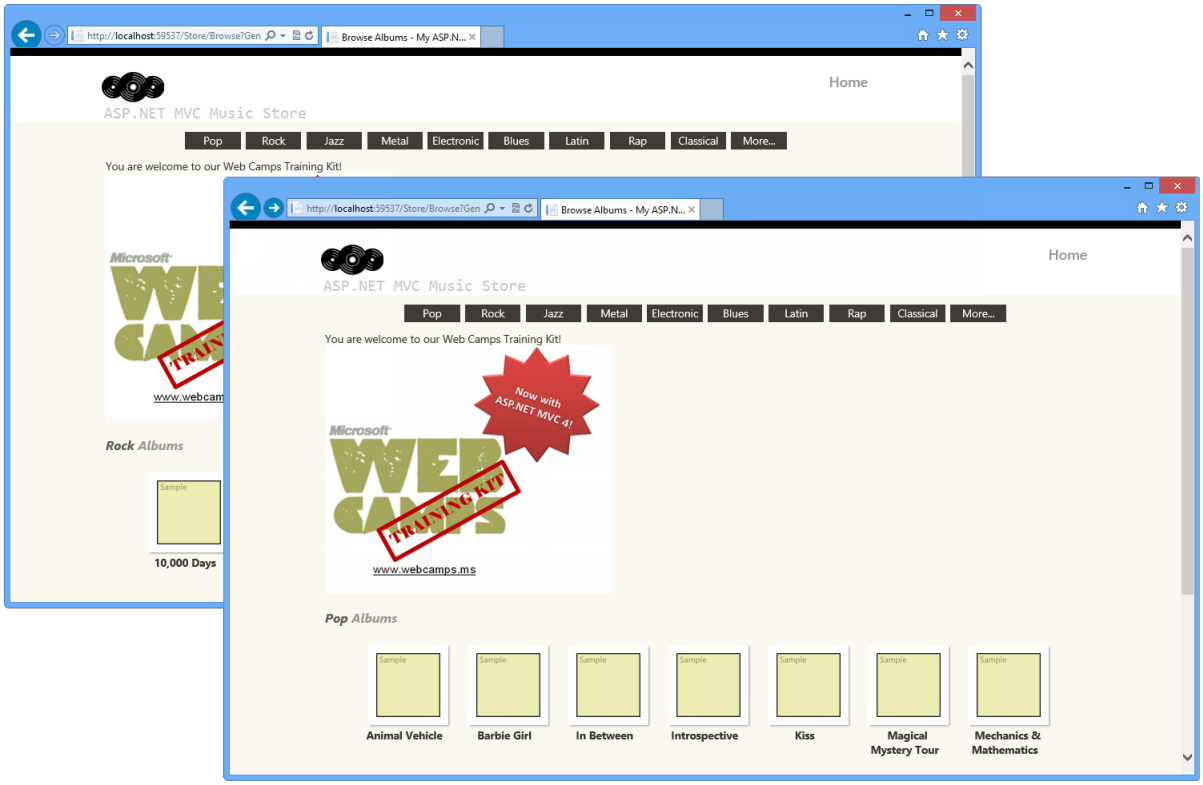
    return container;
}
```

Task 3 - Running the Application

In this task, you will run the application and test that the custom action filter is tracing the activity:

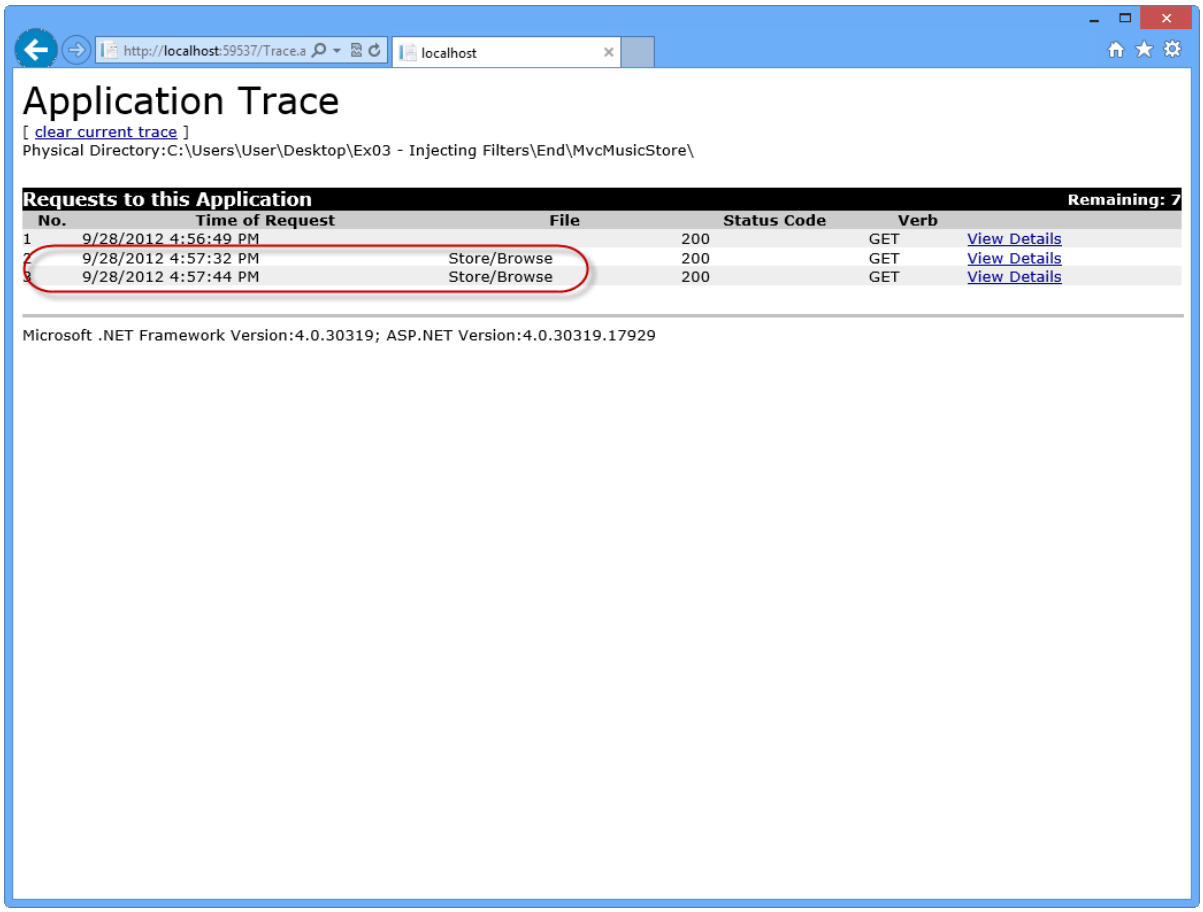
1. Press **F5** to run the application.

2. Click **Rock** within the Genres Menu. You can browse to more genres if you want to.

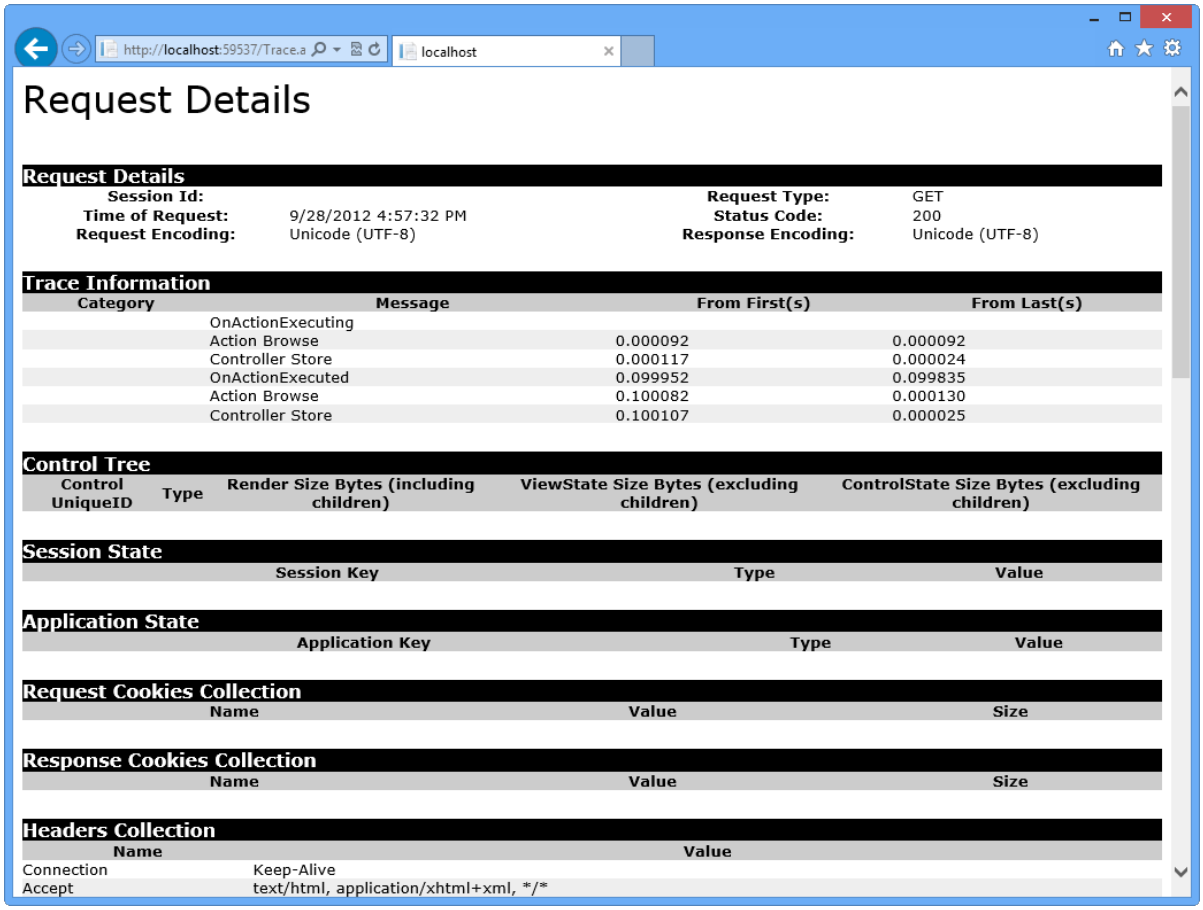


Music Store

3. Browse to **/Trace.axd** to see the Application Trace page, and then click **View Details**.



Application Trace Log



Application Trace - Request Details

4. Close the browser.

Summary

By completing this Hands-On Lab you have learned how to use Dependency Injection in ASP.NET MVC 4 by integrating Unity using a NuGet Package. To achieve that, you have used Dependency Injection inside controllers, views and action filters.

The following concepts were covered:

- ASP.NET MVC 4 Dependency Injection features
- Unity integration using Unity.Mvc3 NuGet Package
- Dependency Injection in Controllers
- Dependency Injection in Views
- Dependency injection of Action Filters

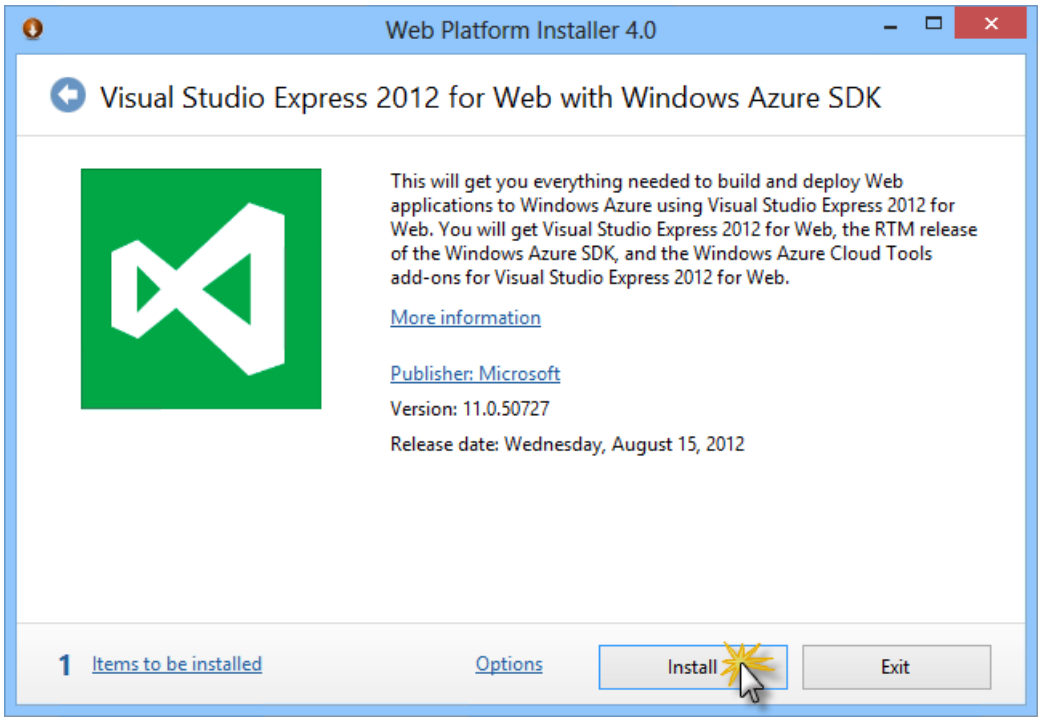
Appendix A: Installing Visual Studio Express 2012 for Web

You can install **Microsoft Visual Studio Express 2012 for Web** or another "Express" version using the **Microsoft Web Platform Installer**. The following instructions guide you through the steps required to install *Visual studio Express 2012 for Web* using *Microsoft Web Platform Installer*.

1. Go to https://go.microsoft.com/?linkid=9810169. Alternatively, if you already have installed Web Platform Installer, you can open it and search for the product "*Visual Studio Express 2012 for Web with Windows Azure SDK*".

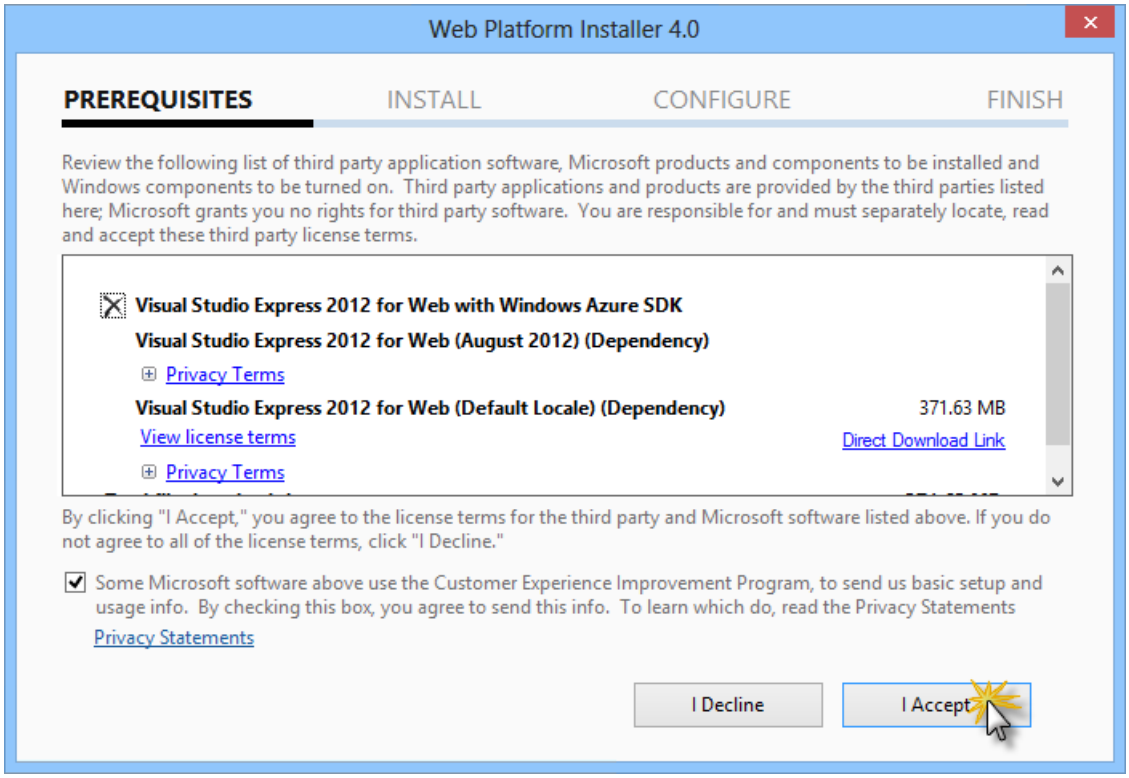
2. Click on **Install Now**. If you do not have **Web Platform Installer** you will be redirected to download and install it first.

3. Once **Web Platform Installer** is open, click **Install** to start the setup.



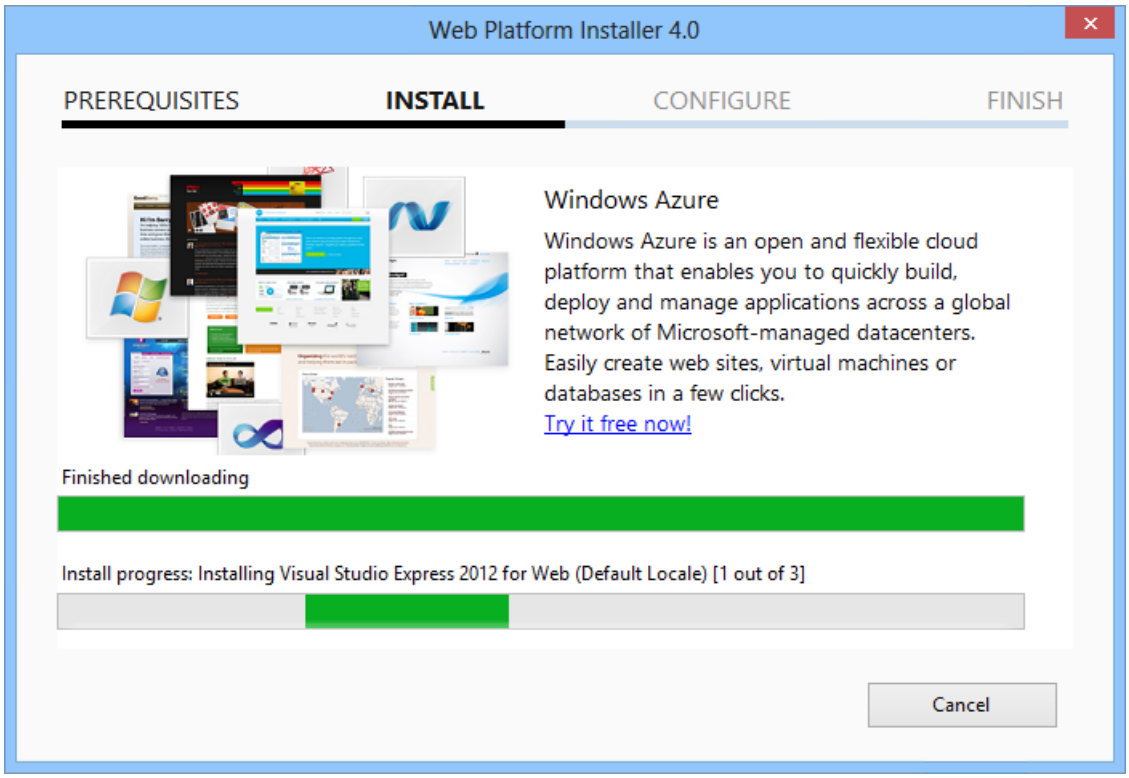
Install Visual Studio Express

4. Read all the products' licenses and terms and click **I Accept** to continue.



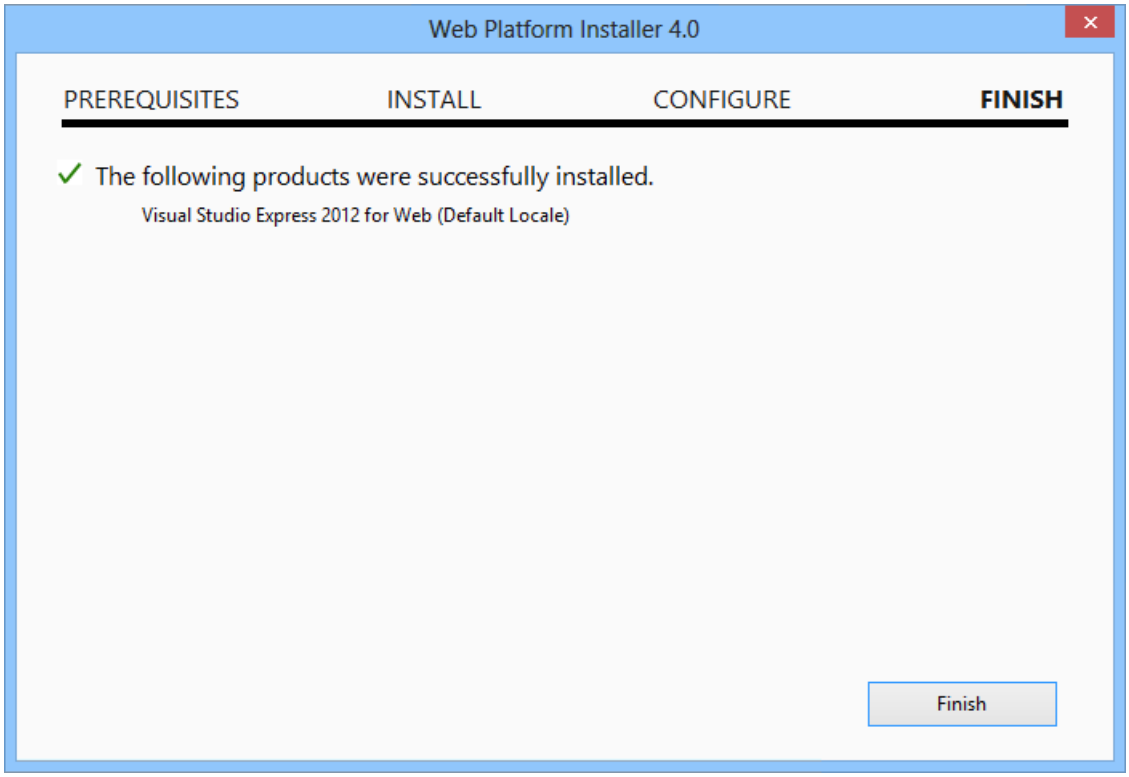
Accepting the license terms

5. Wait until the downloading and installation process completes.



Installation progress

6. When the installation completes, click **Finish**.



Installation completed

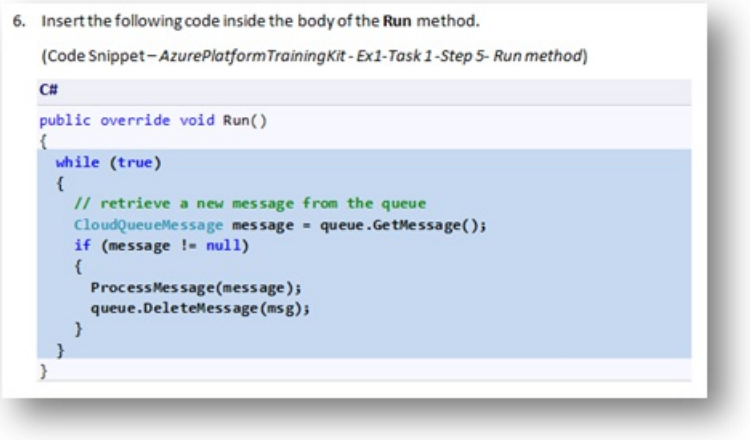
7. Click **Exit** to close Web Platform Installer.

8. To open Visual Studio Express for Web, go to the **Start** screen and start writing "**VS Express**", then click on the **VS Express for Web** tile.

VS Express for Web tile

Appendix B: Using Code Snippets

With code snippets, you have all the code you need at your fingertips. The lab document will tell you exactly when you can use them, as shown in the following figure.



6. Insert the following code inside the body of the **Run** method.

(Code Snippet – *AzurePlatformTrainingKit - Ex1-Task 1 -Step 5- Run method*)

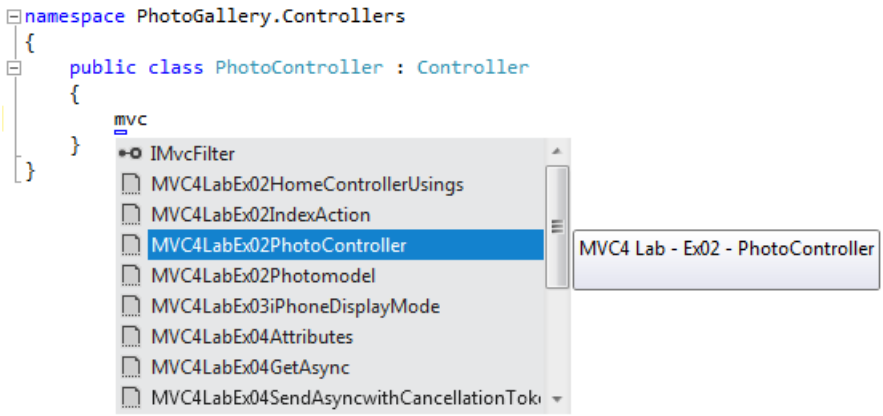
C#

```
public override void Run()
{
  while (true)
  {
    // retrieve a new message from the queue
    CloudQueueMessage message = queue.GetMessage();
    if (message != null)
    {
      ProcessMessage(message);
      queue.DeleteMessage(msg);
    }
  }
}
```

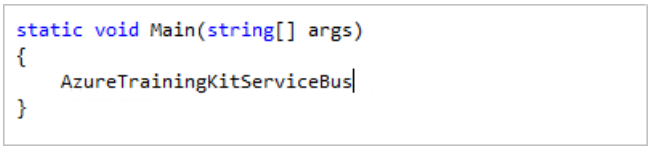
Using Visual Studio code snippets to insert code into your project

To add a code snippet using the keyboard (C# only)

1. Place the cursor where you would like to insert the code.
2. Start typing the snippet name (without spaces or hyphens).
3. Watch as IntelliSense displays matching snippets' names.
4. Select the correct snippet (or keep typing until the entire snippet's name is selected).
5. Press the Tab key twice to insert the snippet at the cursor location.



Start typing the snippet name



```
static void Main(string[] args)
{
    AzureTrainingKitServiceBus
}
```

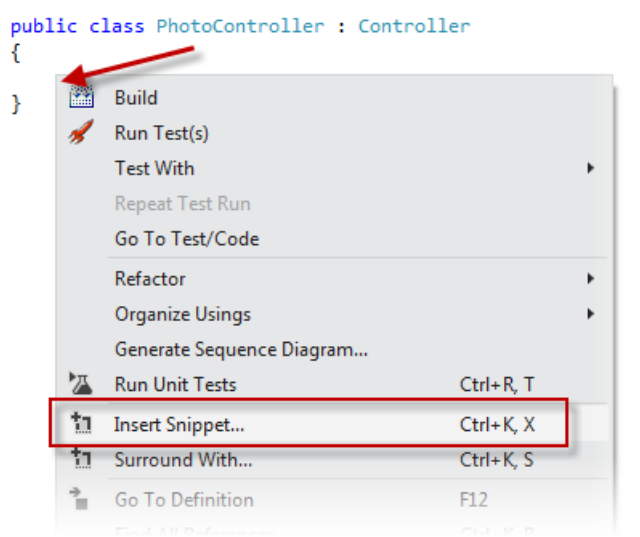
Press Tab to select the highlighted snippet

```
static void Main(string[] args)
{
    request.Headers.Add("$accountKey", accountKey);
    request.Headers.Add("$uniqueUserID", uniqueUserId);
}
```

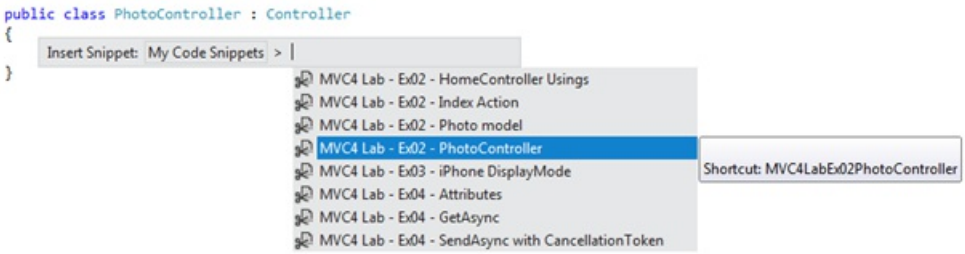
Press Tab again and the snippet will expand

To add a code snippet using the mouse (C#, Visual Basic and XML) 1. Right-click where you want to insert the code snippet.

1. Select **Insert Snippet** followed by **My Code Snippets**.
2. Pick the relevant snippet from the list, by clicking on it.



Right-click where you want to insert the code snippet and select Insert Snippet



Pick the relevant snippet from the list, by clicking on it

ASP.NET MVC 4 Helpers, Forms and Validation

2/20/2020 • 35 minutes to read • Edit Online

By Web Camps Team

Download Web Camps Training Kit

In **ASP.NET MVC 4 Models and Data Access** Hands-on Lab, you have been loading and displaying data from the database. In this Hands-on Lab, you will add to the **Music Store** application the ability to edit that data.

With that goal in mind, you will first create the controller that will support the Create, Read, Update and Delete (CRUD) actions of albums. You will generate an Index View template taking advantage of ASP.NET MVC's scaffolding feature to display the albums' properties in an HTML table. To enhance that view, you will add a custom HTML helper that will truncate long descriptions.

Afterwards, you will add the Edit and Create Views that will let you alter the albums in the database, with the help of form elements like dropdowns.

Lastly, you will let users delete an album and also you will prevent them from entering wrong data by validating their input.

This Hands-on Lab assumes you have basic knowledge of **ASP.NET MVC**. If you have not used **ASP.NET MVC** before, we recommend you to go over **ASP.NET MVC Fundamentals** Hands-on Lab.

This lab walks you through the enhancements and new features previously described by applying minor changes to a sample Web application provided in the Source folder.

> **NOTE**
>
> All sample code and snippets are included in the Web Camps Training Kit, available at Microsoft-Web/WebCampTrainingKit Releases. The project specific to this lab is available at ASP.NET MVC 4 Helpers, Forms and Validation.

Objectives

In this Hands-On Lab, you will learn how to:

- Create a controller to support CRUD operations
- Generate an Index View to display entity properties in an HTML table
- Add a custom HTML helper
- Create and customize an Edit View
- Differentiate between action methods that react to either HTTP-GET or HTTP-POST calls
- Add and customize a Create View
- Handle the deletion of an entity
- Validate user input

Prerequisites

You must have the following items to complete this lab:

- Microsoft Visual Studio Express 2012 for Web or superior (read Appendix A for instructions on how to install it).

Setup

Installing Code Snippets

For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. To install the code snippets run **.\Source\Setup\CodeSnippets.vsi** file.

If you are not familiar with the Visual Studio Code Snippets, and want to learn how to use them, you can refer to the appendix from this document "Appendix B: Using Code Snippets".

Exercises

The following exercises make up this Hands-On Lab:

1. Creating the Store Manager controller and its Index view
2. Adding an HTML Helper
3. Creating the Edit View
4. Adding a Create View
5. Handling Deletion
6. Adding Validation
7. Using Unobtrusive jQuery at Client Side

> **NOTE**
> Each exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

Estimated time to complete this lab: **60 minutes**

Exercise 1: Creating the Store Manager controller and its Index view

In this exercise, you will learn how to create a new controller to support CRUD operations, customize its Index action method to return a list of albums from the database and finally generating an Index View template taking advantage of ASP.NET MVC's scaffolding feature to display the albums' properties in an HTML table.

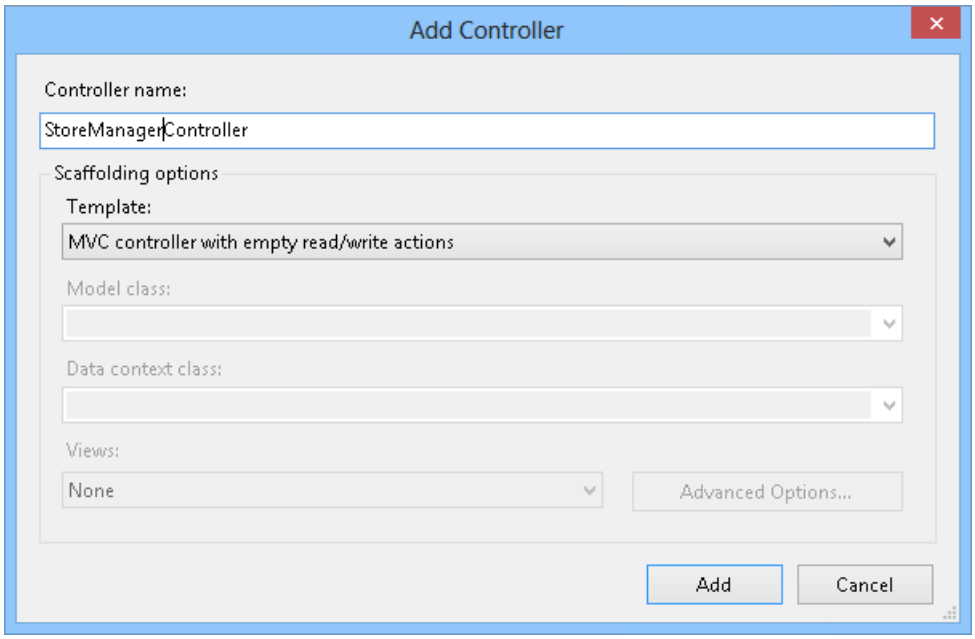
Task 1 - Creating the StoreManagerController

In this task, you will create a new controller called **StoreManagerController** to support CRUD operations.

1. Open the **Begin** solution located at **Source/Ex1-CreatingTheStoreManagerController/Begin/** folder.
   a. You will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.
   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.
   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Add a new controller. To do this, right-click the **Controllers** folder within the Solution Explorer, select **Add** and then the **Controller** command. Change the **Controller Name** to **StoreManagerController** and make sure the option **MVC controller with empty read/write actions** is selected. Click **Add**.



Add Controller Dialog

A new Controller class is generated. Since you indicated to add actions for read/write, stub methods for those, common CRUD actions are created with TODO comments filled in, prompting to include the application specific logic.

Task 2 - Customizing the StoreManager Index

In this task, you will customize the StoreManager Index action method to return a View with the list of albums from the database.

1. In the StoreManagerController class, add the following *using* directives.

   (Code Snippet - *ASP.NET MVC 4 Helpers and Forms and Validation - Ex1 using MvcMusicStore*)

```
using System.Data;
using System.Data.Entity;
using MvcMusicStore.Models;
```

2. Add a field to the **StoreManagerController** to hold an instance of **MusicStoreEntities.**

   (Code Snippet - *ASP.NET MVC 4 Helpers and Forms and Validation - Ex1 MusicStoreEntities*)

```
public class StoreManagerController : Controller
{
    private MusicStoreEntities db = new MusicStoreEntities();
```

3. Implement the StoreManagerController Index action to return a View with the list of albums.

   The Controller action logic will be very similar to the StoreController's Index action written earlier. Use LINQ to retrieve all albums, including Genre and Artist information for display.

   (Code Snippet - *ASP.NET MVC 4 Helpers and Forms and Validation - Ex1 StoreManagerController Index*)

```
//
// GET: /StoreManager/

public ActionResult Index()
{
    var albums = this.db.Albums.Include(a => a.Genre).Include(a => a.Artist)
         .OrderBy(a => a.Price);

    return this.View(albums.ToList());
}
```

Task 3 - Creating the Index View

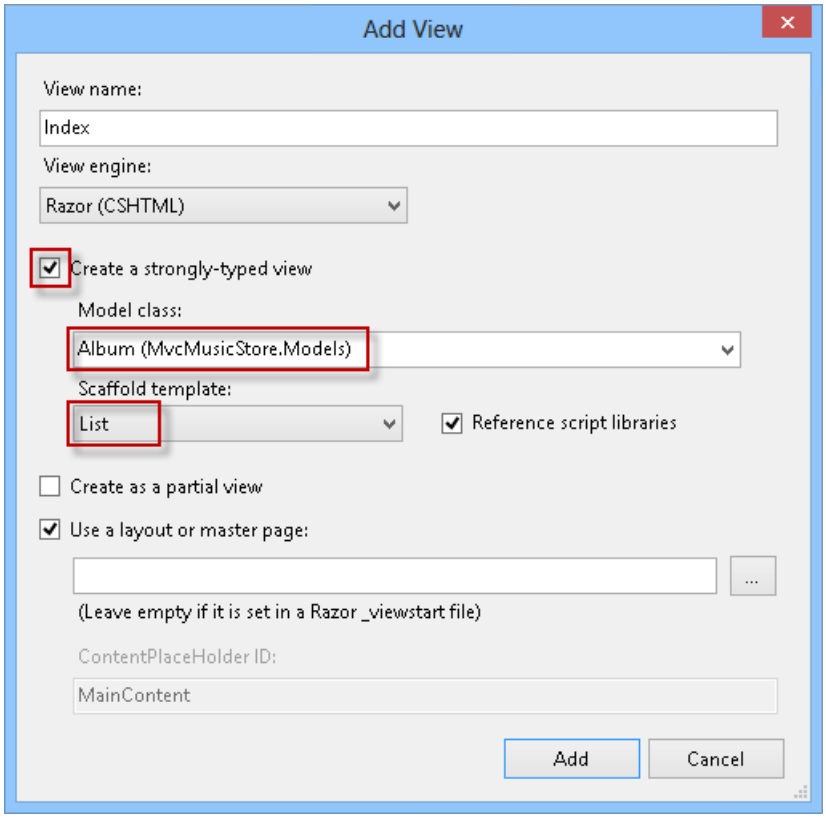
In this task, you will create the Index View template to display the list of albums returned by the **StoreManager** Controller.

1. Before creating the new View template, you should build the project so that the **Add View Dialog** knows about the **Album** class to use. Select **Build | Build MvcMusicStore** to build the project.

2. Right-click inside the **Index** action method and select **Add View**. This will bring up the **Add View** dialog.

Adding a View from within the Index method

3. In the Add View dialog, verify that the View Name is **Index**. Select the **Create a strongly-typed view** option, and select **Album (MvcMusicStore.Models)** from the **Model class** drop-down. Select **List** from the **Scaffold template** drop-down. Leave the **View engine** to **Razor** and the other fields with their default value and then click **Add**.



Adding an Index View

Task 4 - Customizing the scaffold of the Index View

In this task, you will adjust the simple View template created with ASP.NET MVC scaffolding feature to have it display the fields you want.

> **NOTE**
>
> The **scaffolding** support within ASP.NET MVC generates a simple View template which lists all fields in the Album model. **Scaffolding** provides a quick way to get started on a strongly typed view: rather than having to write the View template manually, scaffolding quickly generates a default template and then you can modify the generated code.

1. Review the code created. The generated list of fields will be part of the following HTML table that **Scaffolding** is using for displaying tabular data.

```
@model IEnumerable<MvcMusicStore.Models.Album>

@{
     ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
     @Html.ActionLink("Create New", "Create")
</p>
<table>
     <tr>
          <th>
                @Html.DisplayNameFor(model => model.GenreId)
          </th>
          <th>
                @Html.DisplayNameFor(model => model.ArtistId)
          </th>
          <th>
                @Html.DisplayNameFor(model => model.Title)
          </th>
          <th>
                @Html.DisplayNameFor(model => model.Price)
          </th>
          <th>
                @Html.DisplayNameFor(model => model.AlbumArtUrl)
          </th>
          <th></th>
     </tr>

@foreach (var item in Model) {
     <tr>
          <td>
                @Html.DisplayFor(modelItem => item.GenreId)
          </td>
          <td>
                @Html.DisplayFor(modelItem => item.ArtistId)
          </td>
          <td>
                @Html.DisplayFor(modelItem => item.Title)
          </td>
          <td>
                @Html.DisplayFor(modelItem => item.Price)
          </td>
          <td>
                @Html.DisplayFor(modelItem => item.AlbumArtUrl)
          </td>
          <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) |
                @Html.ActionLink("Details", "Details", new { id=item.AlbumId }) |
                @Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })
          </td>
     </tr>
}

</table>
```

2. Replace the **<table>** code with the following code to display only the **Genre**, **Artist**, **Album Title**, and **Price** fields. This deletes the **AlbumId** and **Album Art URL** columns. Also, it changes GenreId and ArtistId columns to display their linked class properties of **Artist.Name** and **Genre.Name**, and removes the **Details** link.

```
<table>
    <tr>
      <th></th>
      <th>Genre</th>
      <th>Artist</th>
      <th>Title</th>
      <th>Price</th>
    </tr>

    @foreach (var item in Model) {
     <tr>
          <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) |

                @Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })
          </td>
          <td>
                @Html.DisplayFor(modelItem => item.Genre.Name)
          </td>
          <td>
                @Html.DisplayFor(modelItem => item.Artist.Name)
          </td>
          <td>
                @Html.DisplayFor(modelItem => item.Title)
          </td>
          <td>
                @Html.DisplayFor(modelItem => item.Price)
          </td>
     </tr>
    }
</table>
```

3. Change the following descriptions.

```
@model IEnumerable<MvcMusicStore.Models.Album>
@{
    ViewBag.Title = "Store Manager - All Albums";
}

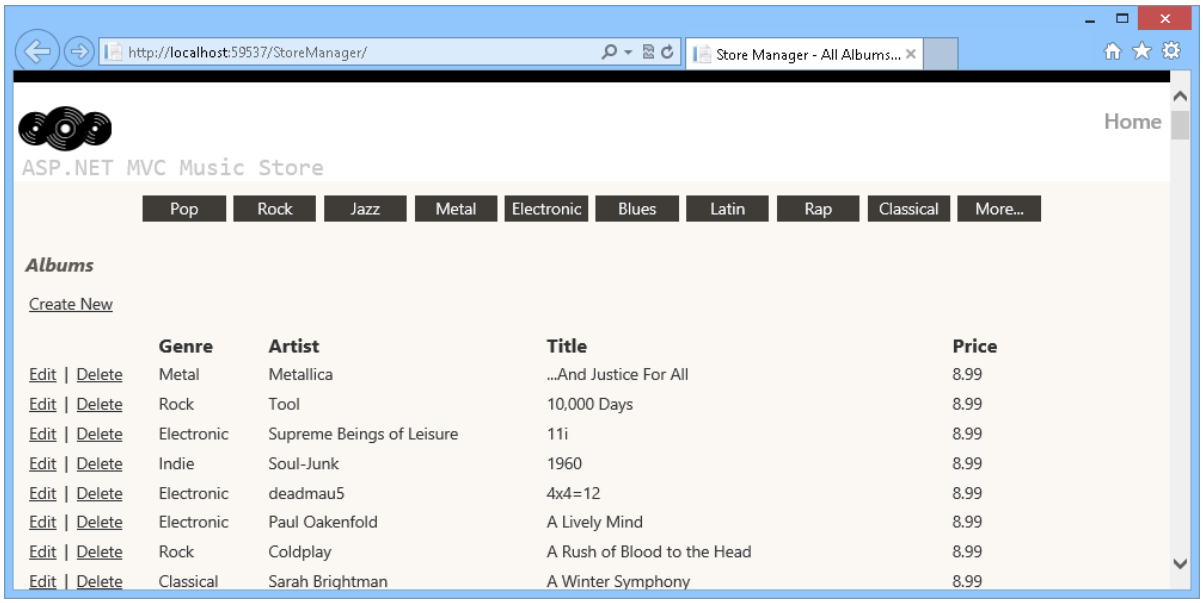
<h2>Albums</h2>
```

Task 5 - Running the Application

In this task, you will test that the **StoreManager Index** View template displays a list of albums according to the design of the previous steps.

1. Press **F5** to run the Application.

2. The project starts in the Home page. Change the URL to **/StoreManager** to verify that a list of albums is displayed, showing their **Title**, **Artist** and **Genre**.



Browsing the list of albums

Exercise 2: Adding an HTML Helper

The StoreManager Index page has one potential issue: Title and Artist Name properties can both be long enough to throw off the table formatting. In this exercise you will learn how to add a custom HTML helper to truncate that text.

In the following figure, you can see how the format is modified because of the length of the text when you use a small browser size.

Browsing the list of Albums with not truncated text

Task 1 - Extending the HTML Helper

In this task, you will add a new method **Truncate** to the **HTML** object exposed within ASP.NET MVC Views. To do this, you will implement an **extension method** to the built-in **System.Web.Mvc.HtmlHelper** class provided by ASP.NET MVC.

> **NOTE**
>
> To read more about **Extension Methods**, please visit this msdn article. https://msdn.microsoft.com/library/bb383977.aspx.

1. Open the **Begin** solution located at **Source/Ex2-AddingAnHTMLHelper/Begin/** folder. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.

   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

c. Finally, build the solution by clicking **Build | Build Solution**.

> **NOTE**
>
> One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Open StoreManager's Index View. To do this, in the Solution Explorer expand the **Views** folder, then the **StoreManager** and open the **Index.cshtml** file.

3. Add the following code below the **@model** directive to define the **Truncate** helper method.

```
@model IEnumerable<MvcMusicStore.Models.Album>

@helper Truncate(string input, int length)
{
     if (input.Length <= length) {
          @input
     } else {
          @input.Substring(0, length)<text>...</text>
     }
}

@{
     ViewBag.Title = "Store Manager - All Albums";
}

<h2>Albums</h2>
```

Task 2 - Truncating Text in the Page

In this task, you will use the **Truncate** method to truncate the text in the View template.

1. Open StoreManager's Index View. To do this, in the Solution Explorer expand the **Views** folder, then the **StoreManager** and open the **Index.cshtml** file.

2. Replace the lines that show the **Artist Name** and Album's **Title**. To do this, replace the following lines.

```
<tr>
     <td>
          @Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) |

          @Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })
     </td>
     <td>
          @Html.DisplayFor(modelItem => item.Genre.Name)
     </td>
     <td>
          @Truncate(item.Artist.Name, 25)
     </td>
     <td>
          @Truncate(item.Title, 25)
     </td>
     <td>
          @Html.DisplayFor(modelItem => item.Price)
     </td>
</tr>
```

Task 3 - Running the Application

In this task, you will test that the **StoreManager Index** View template truncates the Album's Title and Artist Name.

1. Press **F5** to run the Application.

2. The project starts in the Home page. Change the URL to **/StoreManager** to verify that long texts in the **Title** and **Artist** column are truncated.

Truncated Titles and Artist Names

Exercise 3: Creating the Edit View

In this exercise, you will learn how to create a form to allow store managers to edit an Album. They will browse the **/StoreManager/Edit/id** URL (**id** being the unique id of the album to edit), thus making an HTTP-GET call to the server.

The Controller Edit action method will retrieve the appropriate Album from the database, create a **StoreManagerViewModel** object to encapsulate it (along with a list of Artists and Genres), and then pass it off to a View template to render the HTML page back to the user. This page will contain a **<form>** element with textboxes and dropdowns for editing the Album properties.

Once the user updates the Album form values and clicks the **Save** button, the changes are submitted via an HTTP-POST call back to **/StoreManager/Edit/id**. Although the URL remains the same as in the last call, ASP.NET MVC identifies that this time it is an HTTP-POST and therefore executes a different Edit action method (one decorated with **[HttpPost]**).

Task 1 - Implementing the HTTP-GET Edit Action Method

In this task, you will implement the HTTP-GET version of the Edit action method to retrieve the appropriate Album from the database, as well as a list of all Genres and Artists. It will package this data up into the **StoreManagerViewModel** object defined in the last step, which will then be passed to a View template to render the response with.

1. Open the **Begin** solution located at **Source/Ex3-CreatingTheEditView/Begin/** folder. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.

   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

> **NOTE**
>
> One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Open the **StoreManagerController** class. To do this, expand the **Controllers** folder and double-click **StoreManagerController.cs**.

3. Replace the **HTTP-GET Edit** action method with the following code to retrieve the appropriate **Album** as well as the **Genres** and **Artists** lists.

   (Code Snippet - *ASP.NET MVC 4 Helpers and Forms and Validation - Ex3 StoreManagerController HTTP-GET Edit action*)

```
public ActionResult Edit(int id)
{
    Album album = this.db.Albums.Find(id);

    if (album == null)
    {
         return this.HttpNotFound();
    }

    this.ViewBag.GenreId = new SelectList(this.db.Genres, "GenreId", "Name", album.GenreId);
    this.ViewBag.ArtistId = new SelectList(this.db.Artists, "ArtistId", "Name", album.ArtistId);

    return this.View(album);
}
```

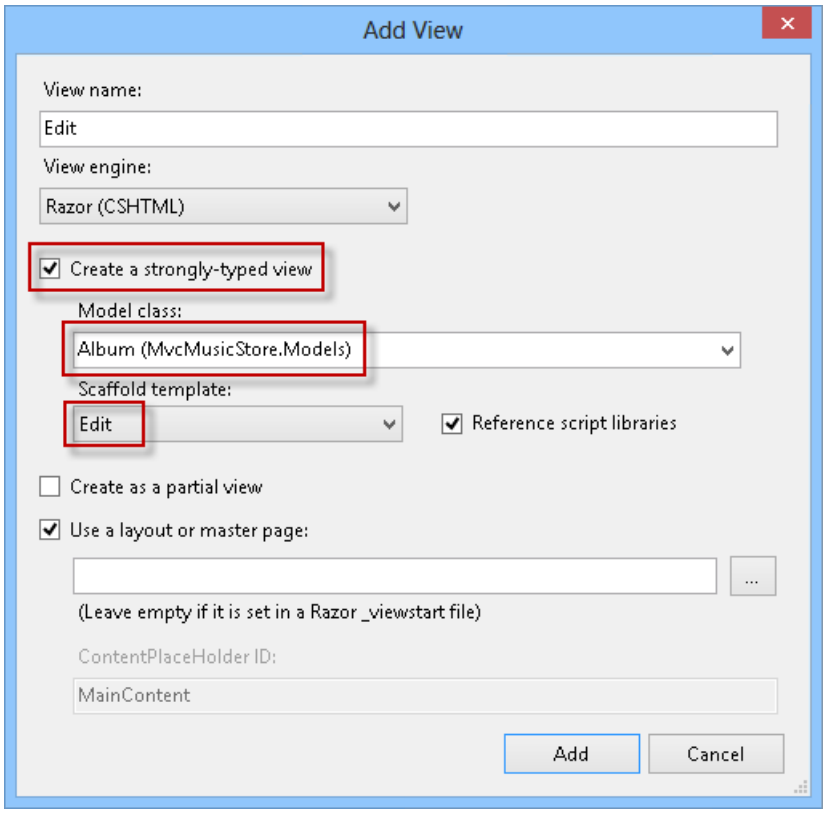
> **NOTE**
>
> You are using **System.Web.Mvc SelectList** for Artists and Genres instead of the **System.Collections.Generic** List.
>
> **SelectList** is a cleaner way to populate HTML dropdowns and manage things like current selection. Instantiating and later setting up these ViewModel objects in the controller action will make the Edit form scenario cleaner.

Task 2 - Creating the Edit View

In this task, you will create an Edit View template that will later display the album properties.

1. Create the Edit View. To do this, right-click inside the **Edit** action method and select **Add View**.

2. In the Add View dialog, verify that the View Name is **Edit**. Check the **Create a strongly-typed view** checkbox and select **Album (MvcMusicStore.Models)** from the **View data class** drop-down. Select **Edit** from the **Scaffold template** drop-down. Leave the other fields with their default value and then click **Add**.

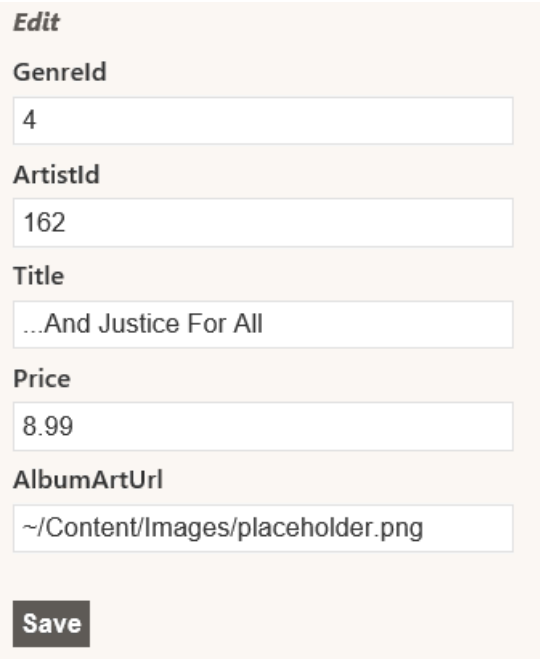


Adding an Edit view

Task 3 - Running the Application

In this task, you will test that the **StoreManager Edit** View page displays the properties' values for the album passed as parameter.

1. Press **F5** to run the Application.
2. The project starts in the Home page. Change the URL to **/StoreManager/Edit/1** to verify that the properties' values for the album passed are displayed.



Browsing Album's Edit view

Task 4 - Implementing drop-downs on the Album Editor Template

In this task, you will add drop-downs to the View template created in the last task, so that the user can select from a list of Artists and Genres.

1. Replace all the **Album** fieldset code with the following:

```
<fieldset>
     <legend>Album</legend>

     @Html.HiddenFor(model => model.AlbumId)

     <div class="editor-label">
          @Html.LabelFor(model => model.Title)
     </div>
     <div class="editor-field">
          @Html.EditorFor(model => model.Title)
          @Html.ValidationMessageFor(model => model.Title)
     </div>

     <div class="editor-label">
          @Html.LabelFor(model => model.Price)
     </div>
     <div class="editor-field">
          @Html.EditorFor(model => model.Price)
          @Html.ValidationMessageFor(model => model.Price)
     </div>

     <div class="editor-label">
          @Html.LabelFor(model => model.AlbumArtUrl)
     </div>
     <div class="editor-field">
          @Html.EditorFor(model => model.AlbumArtUrl)
          @Html.ValidationMessageFor(model => model.AlbumArtUrl)
     </div>

     <div class="editor-label">
          @Html.LabelFor(model => model.Artist)
     </div>
     <div class="editor-field">
          @Html.DropDownList("ArtistId", (SelectList) ViewData["Artists"])
          @Html.ValidationMessageFor(model => model.ArtistId)
     </div>

     <div class="editor-label">
          @Html.LabelFor(model => model.Genre)
     </div>
     <div class="editor-field">
          @Html.DropDownList("GenreId", (SelectList) ViewData["Genres"])
          @Html.ValidationMessageFor(model => model.GenreId)
     </div>

     <p>
          <input type="submit" value="Save" />
     </p>
</fieldset>
```

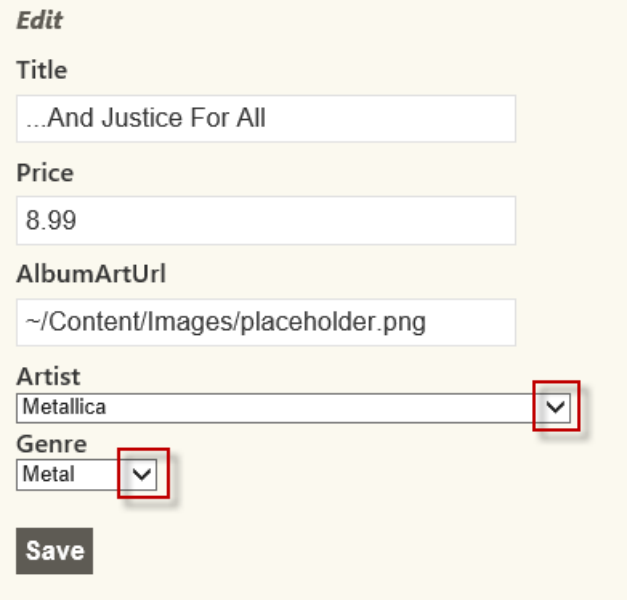
> **NOTE**
>
> An **Html.DropDownList** helper has been added to render drop-downs for choosing Artists and Genres. The parameters passed to **Html.DropDownList** are:
>
> 1. The name of the form field (**"ArtistId"**).
> 2. The **SelectList** of values for the drop-down.

Task 5 - Running the Application

In this task, you will test that the **StoreManager Edit** View page displays drop-downs instead of Artist and Genre ID text fields.

1. Press **F5** to run the Application.

2. The project starts in the Home page. Change the URL to **/StoreManager/Edit/1** to verify that it displays drop-downs instead of Artist and Genre ID text fields.



Browsing Album's Edit view, this time with dropdowns

Task 6 - Implementing the HTTP-POST Edit action method

Now that the Edit View displays as expected, you need to implement the HTTP-POST Edit Action method to save the changes made to the Album.

1. Close the browser if needed, to return to the Visual Studio window. Open **StoreManagerController** from the **Controllers** folder.

2. Replace **HTTP-POST Edit** action method code with the following (note that the method that must be replaced is overloaded version that receives two parameters):

   (Code Snippet - *ASP.NET MVC 4 Helpers and Forms and Validation - Ex3 StoreManagerController HTTP-POST Edit action*)

```
[HttpPost]
public ActionResult Edit(Album album)
{
     if (ModelState.IsValid)
     {
          this.db.Entry(album).State = EntityState.Modified;
          this.db.SaveChanges();

          return this.RedirectToAction("Index");
     }

     this.ViewBag.GenreId = new SelectList(this.db.Genres, "GenreId", "Name", album.GenreId);
     this.ViewBag.ArtistId = new SelectList(this.db.Artists, "ArtistId", "Name", album.ArtistId);

     return this.View(album);
}
```

NOTE

This method will be executed when the user clicks the **Save** button of the View and performs an HTTP-POST of the form values back to the server to persist them in the database. The decorator **[HttpPost]** indicates that the method should be used for those HTTP-POST scenarios. The method takes an **Album** object. ASP.NET MVC will automatically create the Album object from the posted <form> values.

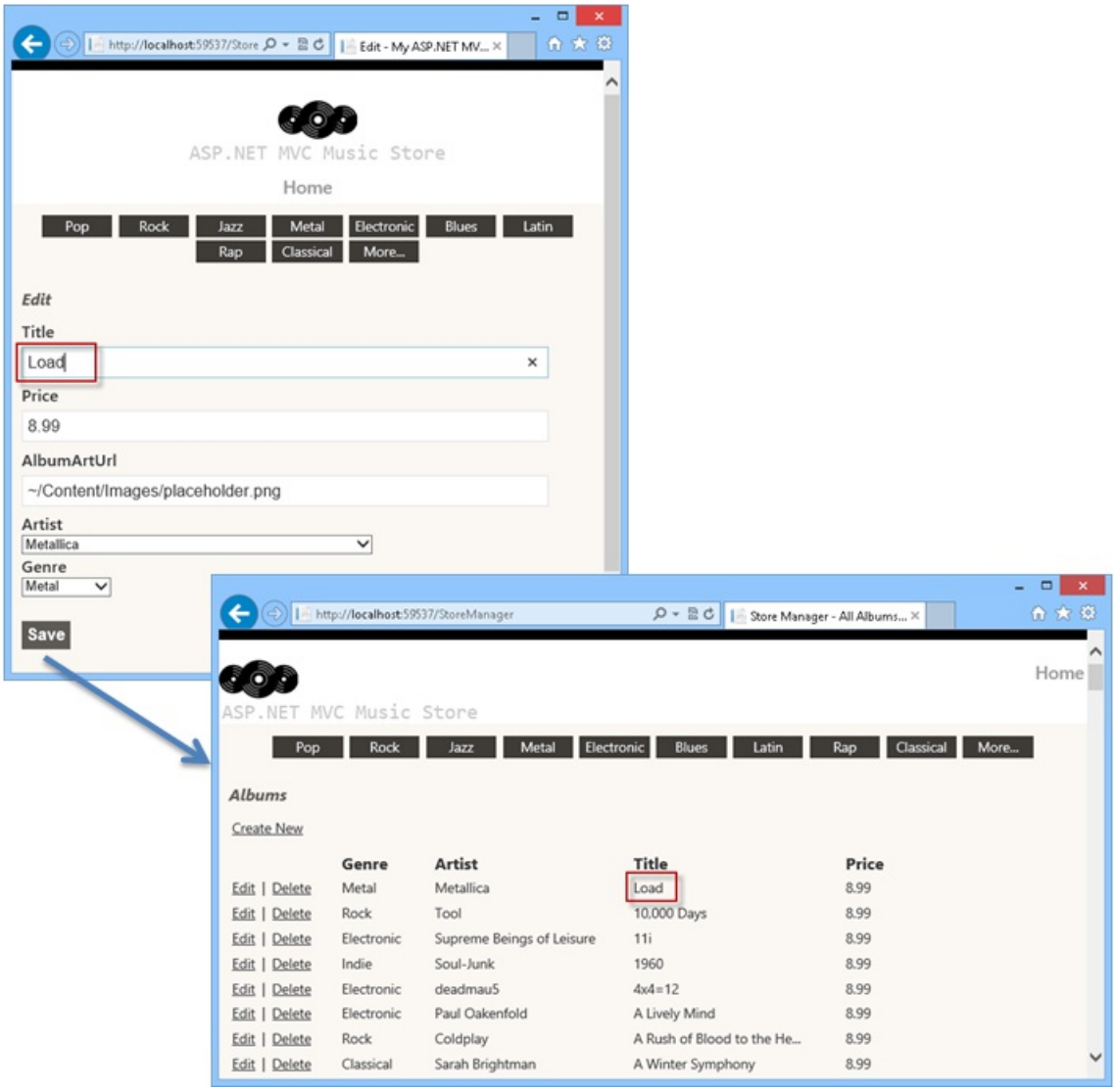
The method will perform these steps:

1. If model is valid:
   a. Update the album entry in the context to mark it as a modified object.
   b. Save the changes and redirect to the index view.
2. If the model is not valid, it will populate the ViewBag with the **GenreId** and **ArtistId**, then it will return the view with the received Album object to allow the user perform any required update.

Task 7 - Running the Application

In this task, you will test that the **StoreManager Edit** View page actually saves the updated Album data in the database.

1. Press **F5** to run the Application.
2. The project starts in the Home page. Change the URL to **/StoreManager/Edit/1**. Change the Album title to **Load** and click on **Save**. Verify that album's title actually changed in the list of albums.



Updating an Album

Exercise 4: Adding a Create View

Now that the **StoreManagerController** supports the **Edit** ability, in this exercise you will learn how to add a Create View template to let store managers add new Albums to the application.

Like you did with the Edit functionality, you will implement the Create scenario using two separate methods within the **StoreManagerController** class:

1. One action method will display an empty form when store managers first visit the **/StoreManager/Create** URL.
2. A second action method will handle the scenario where the store manager clicks the **Save** button within the form and submits the values back to the **/StoreManager/Create** URL as an HTTP-POST.

Task 1 - Implementing the HTTP-GET Create action method

In this task, you will implement the HTTP-GET version of the Create action method to retrieve a list of all Genres and Artists, package this data up into a **StoreManagerViewModel** object, which will then be passed to a View template.

1. Open the **Begin** solution located at **Source/Ex4-AddingACreateView/Begin/** folder. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.

   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Open **StoreManagerController** class. To do this, expand the **Controllers** folder and double-click **StoreManagerController.cs**.

3. Replace the **Create** action method code with the following:

   (Code Snippet - *ASP.NET MVC 4 Helpers and Forms and Validation - Ex4 StoreManagerController HTTP-GET Create action*)

```
//
// GET: /StoreManager/Create

public ActionResult Create()
{
     this.ViewBag.GenreId = new SelectList(this.db.Genres, "GenreId", "Name");
     this.ViewBag.ArtistId = new SelectList(this.db.Artists, "ArtistId", "Name");

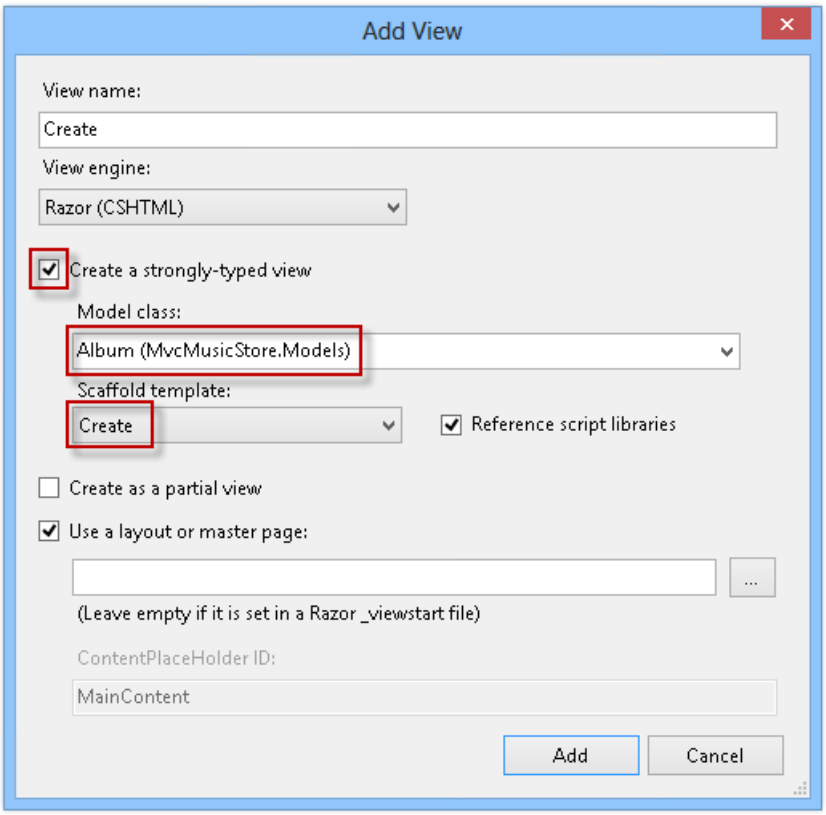
     return this.View();
}
```

Task 2 - Adding the Create View

In this task, you will add the Create View template that will display a new (empty) Album form.

1. Right-click inside the **Create** action method and select **Add View**. This will bring up the Add View dialog.

2. In the Add View dialog, verify that the View Name is **Create**. Select the **Create a strongly-typed view** option and select **Album (MvcMusicStore.Models)** from the **Model class** drop-down and **Create** from the **Scaffold template** drop-down. Leave the other fields with their default value and then click **Add**.



Adding the Create View

3. Update the **GenreId** and **ArtistId** fields to use a drop-down list as shown below:

```
...
<fieldset>
     <legend>Album</legend>

     <div class="editor-label">
          @Html.LabelFor(model => model.GenreId, "Genre")
     </div>
     <div class="editor-field">
          @Html.DropDownList("GenreId", String.Empty)
          @Html.ValidationMessageFor(model => model.GenreId)
     </div>

     <div class="editor-label">
          @Html.LabelFor(model => model.ArtistId, "Artist")
     </div>
     <div class="editor-field">
          @Html.DropDownList("ArtistId", String.Empty)
          @Html.ValidationMessageFor(model => model.ArtistId)
     </div>

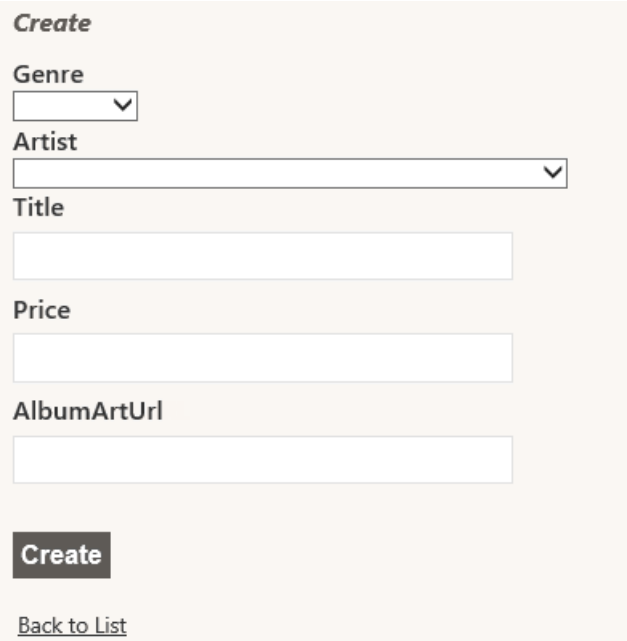
     <div class="editor-label">
          @Html.LabelFor(model => model.Title)
     </div>
    ...
```

Task 3 - Running the Application

In this task, you will test that the **StoreManager Create** View page displays an empty Album form.

1. Press **F5** to run the Application.

2. The project starts in the Home page. Change the URL to **/StoreManager/Create**. Verify that an empty form is displayed for filling the new Album properties.



Create View with an empty form

Task 4 - Implementing the HTTP-POST Create Action Method

In this task, you will implement the HTTP-POST version of the Create action method that will be invoked when a user clicks the **Save** button. The method should save the new album in the database.

1. Close the browser if needed, to return to the Visual Studio window. Open **StoreManagerController** class. To do this, expand the **Controllers** folder and double-click **StoreManagerController.cs**.

2. Replace **HTTP-POST Create** action method code with the following:

   (Code Snippet - *ASP.NET MVC 4 Helpers and Forms and Validation - Ex4 StoreManagerController HTTP-POST Create action*)

```
[HttpPost]
public ActionResult Create(Album album)
{
     if (ModelState.IsValid)
     {
          this.db.Albums.Add(album);
          this.db.SaveChanges();

          return this.RedirectToAction("Index");
     }

     this.ViewBag.GenreId = new SelectList(this.db.Genres, "GenreId", "Name", album.GenreId);
     this.ViewBag.ArtistId = new SelectList(this.db.Artists, "ArtistId", "Name", album.ArtistId);

     return this.View(album);
}
```

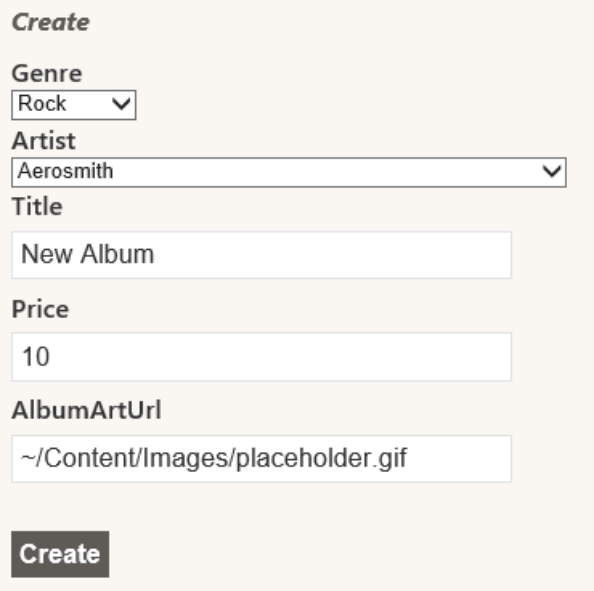
> **NOTE**
>
> The Create action is pretty similar to the previous Edit action method but instead of setting the object as modified, it is being added to the context.

Task 5 - Running the Application

In this task, you will test that the **StoreManager Create** View page lets you create a new Album and then redirects to the StoreManager Index View.

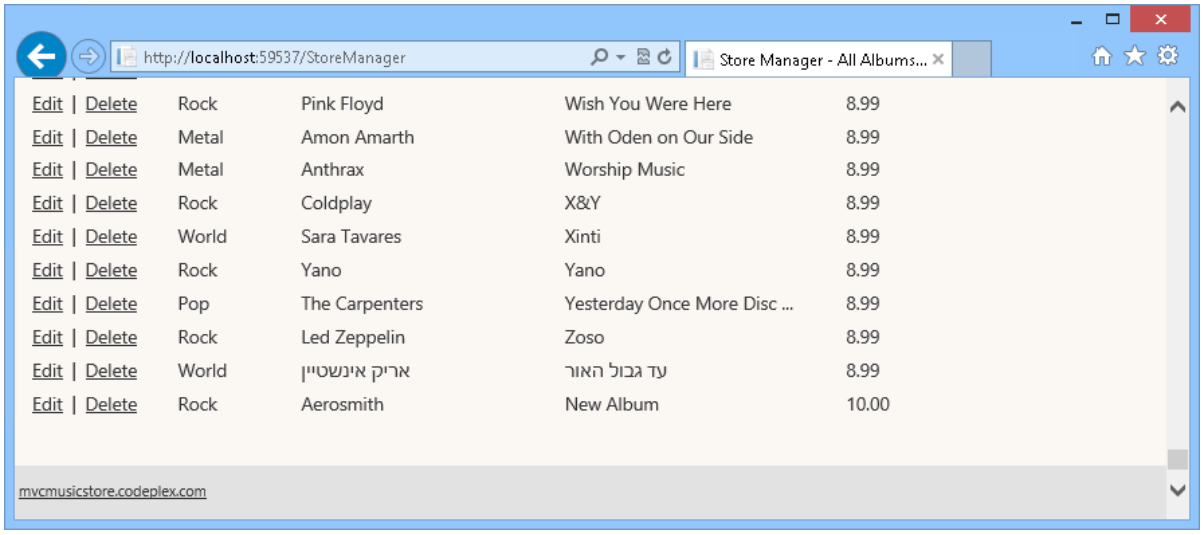
1. Press **F5** to run the Application.

2. The project starts in the Home page. Change the URL to **/StoreManager/Create**. Fill all the form fields with data for a new Album, like the one in the following figure:



Creating an Album

3. Verify that you get redirected to the StoreManager Index View that includes the new Album just created.



New Album Created

Exercise 5: Handling Deletion

The ability to delete albums is not yet implemented. This is what this exercise will be about. Like before, you will implement the Delete scenario using two separate methods within the **StoreManagerController** class:

1. One action method will display a confirmation form
2. A second action method will handle the form submission

Task 1 - Implementing the HTTP-GET Delete Action Method

In this task, you will implement the HTTP-GET version of the Delete action method to retrieve the album's information.

1. Open the **Begin** solution located at **Source/Ex5-HandlingDeletion/Begin/** folder. Otherwise, you might

continue using the **End** solution obtained by completing the previous exercise.

a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

c. Finally, build the solution by clicking **Build | Build Solution**.

> **NOTE**
>
> One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Open **StoreManagerController** class. To do this, expand the **Controllers** folder and double-click **StoreManagerController.cs**.

3. The Delete controller action is exactly the same as the previous Store Details controller action: it queries the **album** object from the database using the **id** provided in the URL and returns the appropriate **View**. To do this, replace the HTTP-GET **Delete** action method code with the following:

   (Code Snippet - *ASP.NET MVC 4 Helpers and Forms and Validation - Ex5 Handling Deletion HTTP-GET Delete action*)

```
//
// GET: /StoreManager/Delete/5

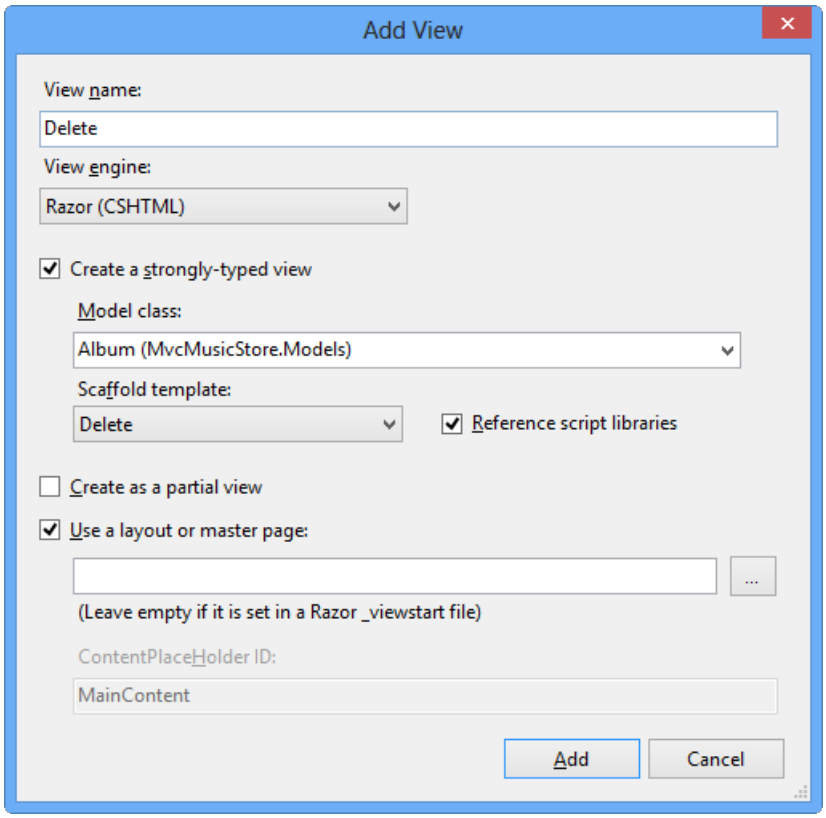
public ActionResult Delete(int id)
{
     Album album = this.db.Albums.Find(id);

     if (album == null)
     {
          return this.HttpNotFound();
     }

     return this.View(album);
}
```

4. Right-click inside the **Delete** action method and select **Add View**. This will bring up the Add View dialog.

5. In the Add View dialog, verify that the View name is **Delete**. Select the **Create a strongly-typed view** option and select **Album (MvcMusicStore.Models)** from the **Model class** drop-down. Select **Delete** from the **Scaffold template** drop-down. Leave the other fields with their default value and then click **Add**.



Adding a Delete View

6. The Delete template shows all the fields from the model. You will show only the album's title. To do this, replace the content of the view with the following code:

```
@model MvcMusicStore.Models.Album
@{
     ViewBag.Title = "Delete";
}
<h2>Delete Confirmation</h2>

<h3> Are you sure you want to delete the album title <strong>@Model.Title </strong> ? </h3>

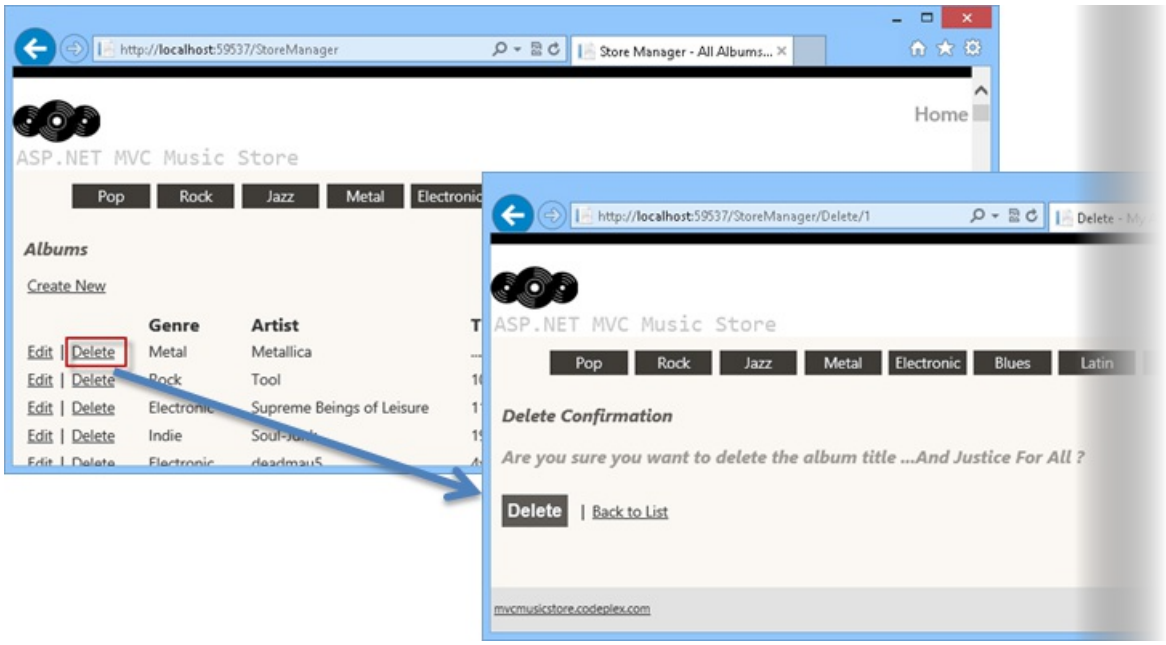
@using (Html.BeginForm()) {
     <p>
          <input type="submit" value="Delete" /> |
          @Html.ActionLink("Back to List", "Index")
     </p>
}
```

Task 2 - Running the Application

In this task, you will test that the **StoreManager Delete** View page displays a confirmation deletion form.

1. Press **F5** to run the Application.

2. The project starts in the Home page. Change the URL to **/StoreManager**. Select one album to delete by clicking **Delete** and verify that the new view is uploaded.



Deleting an Album

Task 3- Implementing the HTTP-POST Delete Action Method

In this task, you will implement the HTTP-POST version of the Delete action method that will be invoked when a user clicks the **Delete** button. The method should delete the album in the database.

1. Close the browser if needed, to return to the Visual Studio window. Open **StoreManagerController** class. To do this, expand the **Controllers** folder and double-click **StoreManagerController.cs**.

2. Replace **HTTP-POST Delete** action method code with the following:

   (Code Snippet - *ASP.NET MVC 4 Helpers and Forms and Validation - Ex5 Handling Deletion HTTP-POST Delete action*)

```
//
// POST: /StoreManager/Delete/5

[HttpPost]
public ActionResult Delete(int id, FormCollection collection)
{
     Album album = this.db.Albums.Find(id);
     this.db.Albums.Remove(album);
     this.db.SaveChanges();

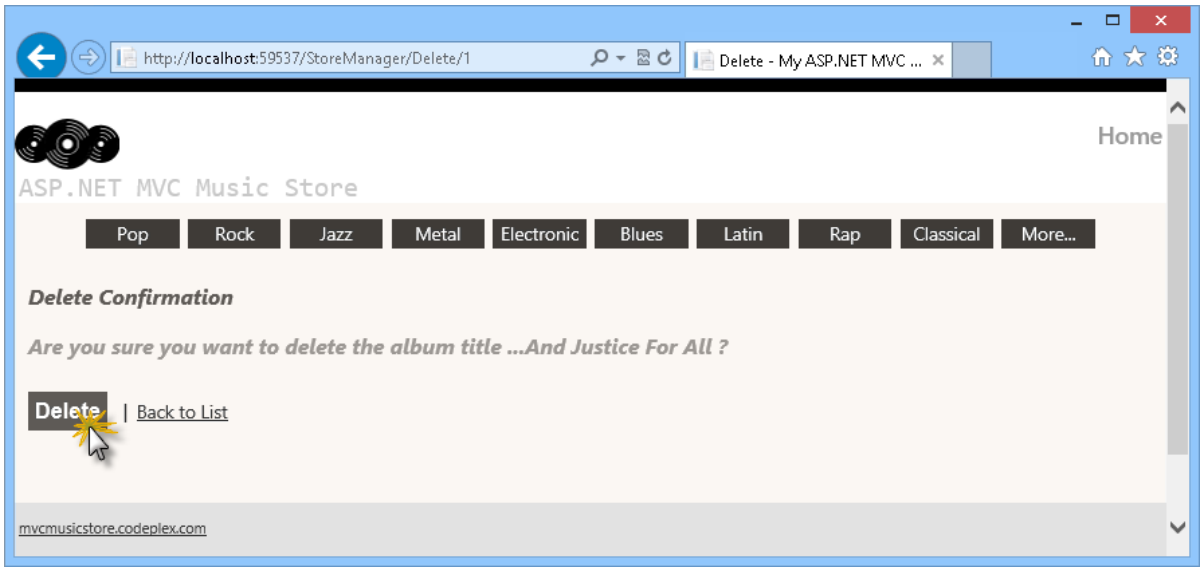
     return this.RedirectToAction("Index");
}
```

Task 4 - Running the Application

In this task, you will test that the **StoreManager Delete** View page lets you delete an Album and then redirects to the StoreManager Index View.

1. Press **F5** to run the Application.

2. The project starts in the Home page. Change the URL to **/StoreManager**. Select one album to delete by clicking **Delete.** Confirm the deletion by clicking **Delete** button:



Deleting an Album

3. Verify that the album was deleted since it does not appear in the **Index** page.

Exercise 6: Adding Validation

Currently, the Create and Edit forms you have in place do not perform any kind of validation. If the user leaves a required field blank or type letters in the price field, the first error you will get will be from the database.

You can add validation to the application by adding Data Annotations to your model class. Data Annotations allow describing the rules you want applied to your model properties, and ASP.NET MVC will take care of enforcing and displaying appropriate message to users.

Task 1 - Adding Data Annotations

In this task, you will add Data Annotations to the Album Model that will make the Create and Edit page display validation messages when appropriate.

For a simple Model class, adding a Data Annotation is just handled by adding a **using** statement for **System.ComponentModel.DataAnnotation**, then placing a **[Required]** attribute on the appropriate properties. The following example would make the **Name** property a required field in the View.

```
using System.ComponentModel.DataAnnotations;
namespace SuperheroSample.Models
{
    public class Superhero
    {
        [Required]
        public string Name { get; set; }

        public bool WearsCape { get; set; }
    }
}
```

This is a little more complex in cases like this application where the Entity Data Model is generated. If you added Data Annotations directly to the model classes, they would be overwritten if you update the model from the database. Instead, you can make use of metadata partial classes which will exist to hold the annotations and are associated with the model classes using the **[MetadataType]** attribute.

1. Open the **Begin** solution located at **Source/Ex6-AddingValidation/Begin/** folder. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.

   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

c. Finally, build the solution by clicking **Build | Build Solution**.

> **NOTE**
> One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Open the **Album.cs** from the **Models** folder.

3. Replace **Album.cs** content with the highlighted code, so that it looks like the following:

> **NOTE**
> The line **[DisplayFormat(ConvertEmptyStringToNull=false)]** indicates that empty strings from the model won't be converted to null when the data field is updated in the data source. This setting will avoid an exception when the Entity Framework assigns null values to the model before Data Annotation validates the fields.

(Code Snippet - *ASP.NET MVC 4 Helpers and Forms and Validation - Ex6 Album metadata partial class*)

```
namespace MvcMusicStore.Models
{
    using System.ComponentModel;
    using System.ComponentModel.DataAnnotations;

    public class Album
    {
        [ScaffoldColumn(false)]
        public int AlbumId { get; set; }

        [DisplayName("Genre")]
        public int GenreId { get; set; }

        [DisplayName("Artist")]
        public int ArtistId { get; set; }

        [Required(ErrorMessage = "An Album Title is required")]
        [DisplayFormat(ConvertEmptyStringToNull = false)]
        [StringLength(160, MinimumLength = 2)]
        public string Title { get; set; }

        [Range(0.01, 100.00, ErrorMessage = "Price must be between 0.01 and 100.00")]
        [DataType(DataType.Currency)]
        public decimal Price { get; set; }

        [DisplayName("Album Art URL")]
        [DataType(DataType.ImageUrl)]
        [StringLength(1024)]
        public string AlbumArtUrl { get; set; }

        public virtual Genre Genre { get; set; }

        public virtual Artist Artist { get; set; }
    }
}
```

> **NOTE**
>
> This **Album** partial class has a **MetadataType** attribute which points to the **AlbumMetaData** class for the Data Annotations. These are some of the Data Annotation attributes you are using to annotate the Album model:
>
> - Required - Indicates that the property is a required field
> - DisplayName - Defines the text to be used on form fields and validation messages
> - DisplayFormat - Specifies how data fields are displayed and formatted.
> - StringLength - Defines a maximum length for a string field
> - Range - Gives a maximum and minimum value for a numeric field
> - ScaffoldColumn - Allows hiding fields from editor forms

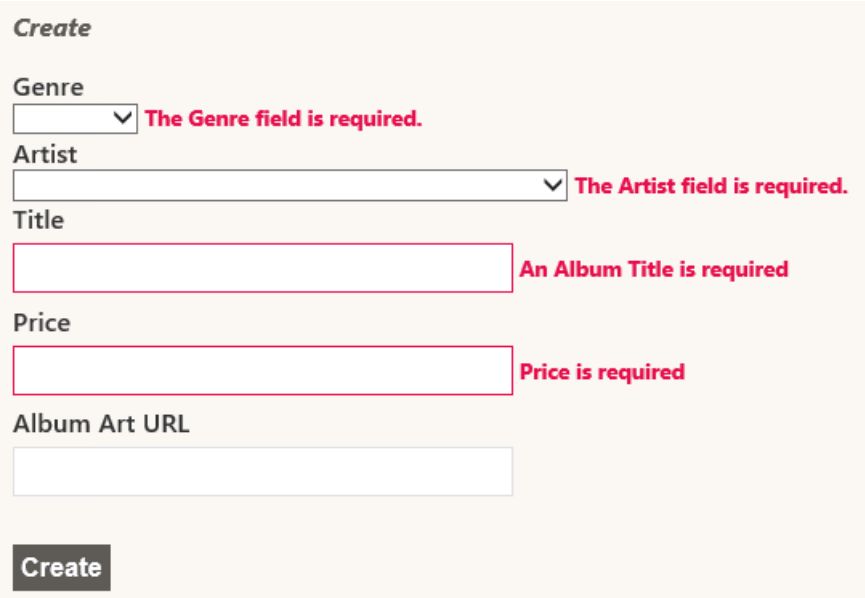
Task 2 - Running the Application

In this task, you will test that the Create and Edit pages validate fields, using the display names chosen in the last task.

1. Press **F5** to run the Application.

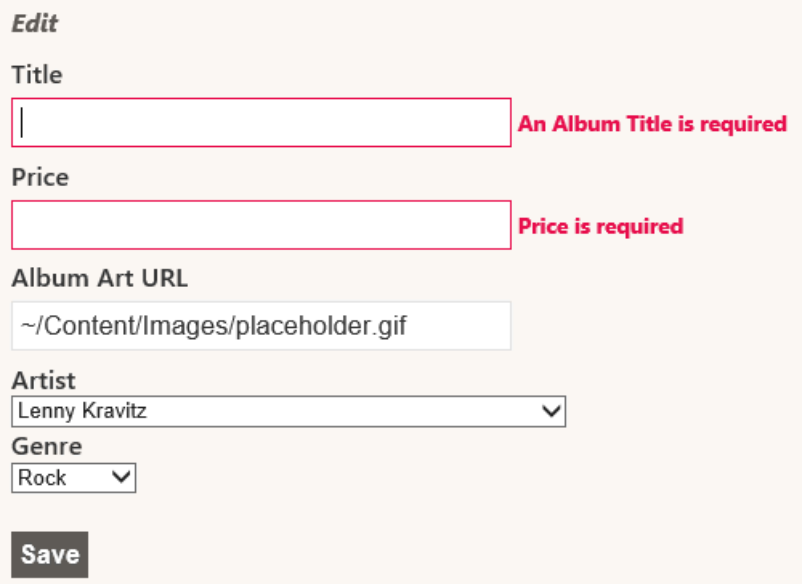
2. The project starts in the Home page. Change the URL to **/StoreManager/Create**. Verify that the display names match the ones in the partial class (like **Album Art URL** instead of **AlbumArtUrl**)

3. Click **Create**, without filling the form. Verify that you get the corresponding validation messages.



Validated fields in the Create page

4. You can verify that the same occurs with the **Edit** page. Change the URL to **/StoreManager/Edit/1** and verify that the display names match the ones in the partial class (like **Album Art URL** instead of **AlbumArtUrl**). Empty the **Title** and **Price** fields and click **Save**. Verify that you get the corresponding validation messages.



Validated fields in the Edit page

Exercise 7: Using Unobtrusive jQuery at Client Side

In this exercise, you will learn how to enable MVC 4 Unobtrusive jQuery validation at client side.

> **NOTE**
> The Unobtrusive jQuery uses data-ajax prefix JavaScript to invoke action methods on the server rather than intrusively emitting inline client scripts.

Task 1 - Running the Application before Enabling Unobtrusive jQuery

In this task, you will run the application before including jQuery in order to compare both validation models.

1. Open the **Begin** solution located at **Source/Ex7-UnobtrusivejQueryValidation/Begin/** folder. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.

   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.

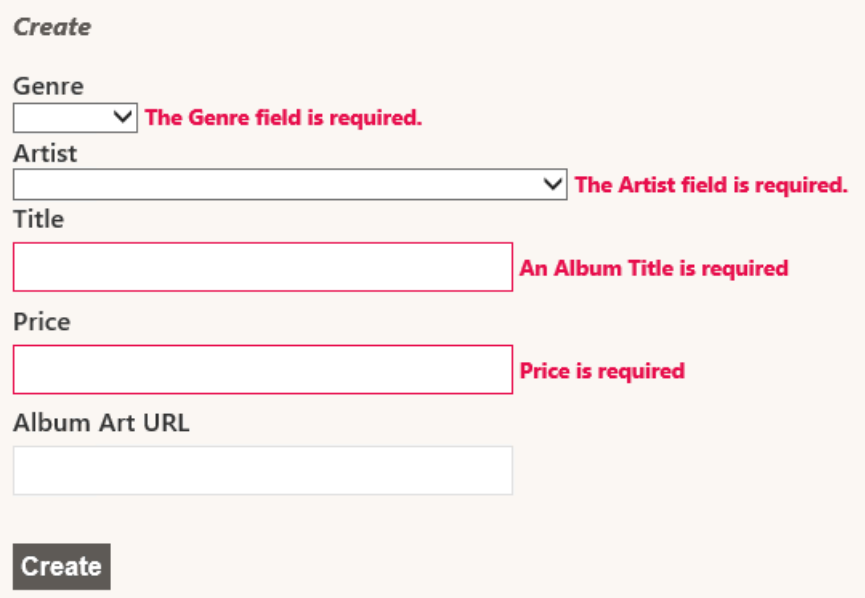
   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.

   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

2. Press **F5** to run the application.

3. The project starts in the Home page. Browse **/StoreManager/Create** and click **Create** without filling the form to verify that you get validation messages:



Client validation disabled

4. In the browser, open the HTML source code:

```
...
          <div class="editor-label">
                <label for="Price">Price</label>
          </div>
          <div class="editor-field">
                <input class="text-box single-line" id="Price" name="Price" type="text" value="" />
                <span class="field-validation-valid" id="Price_validationMessage"></span>
          </div>

          <div class="editor-label">
                <label for="AlbumArtUrl">Album Art URL</label>
          </div>
          <div class="editor-field">
                <input class="text-box single-line" id="AlbumArtUrl" name="AlbumArtUrl" type="text"
value="" />
                <span class="field-validation-valid" id="AlbumArtUrl_validationMessage"></span>
          </div>

          <p>
                <input type="submit" value="Create" />
          </p>
     </fieldset>
</form><script type="text/javascript">
//<![CDATA[
if (!window.mvcClientValidationMetadata) { window.mvcClientValidationMetadata = []; }
window.mvcClientValidationMetadata.push({"Fields":
[{"FieldName":"GenreId","ReplaceValidationMessageContents":true,"ValidationMessageId":"GenreId_validati
onMessage","ValidationRules":[{"ErrorMessage":"The Genre field is required.","ValidationParameters":
{},"ValidationType":"required"},{"ErrorMessage":"The field Genre must be a
number.","ValidationParameters":{},"ValidationType":"number"}]},
{"FieldName":"ArtistId","ReplaceValidationMessageContents":true,"ValidationMessageId":"ArtistId_validat
ionMessage","ValidationRules":[{"ErrorMessage":"The Artist field is required.","ValidationParameters":
{},"ValidationType":"required"},{"ErrorMessage":"The field Artist must be a
number.","ValidationParameters":{},"ValidationType":"number"}]},
{"FieldName":"Title","ReplaceValidationMessageContents":true,"ValidationMessageId":"Title_validationMes
sage","ValidationRules":[{"ErrorMessage":"An Album Title is required","ValidationParameters":
{},"ValidationType":"required"},{"ErrorMessage":"The field Title must be a string with a minimum length
of 2 and a maximum length of 160.","ValidationParameters":
{"min":2,"max":160},"ValidationType":"length"}]},
{"FieldName":"Price","ReplaceValidationMessageContents":true,"ValidationMessageId":"Price_validationMes
sage","ValidationRules":[{"ErrorMessage":"Price must be between 0.01 and
100.00","ValidationParameters":{"min":0.01,"max":100},"ValidationType":"range"},{"ErrorMessage":"Price
is required","ValidationParameters":{},"ValidationType":"required"},{"ErrorMessage":"The field Price
must be a number.","ValidationParameters":{},"ValidationType":"number"}]},
{"FieldName":"AlbumArtUrl","ReplaceValidationMessageContents":true,"ValidationMessageId":"AlbumArtUrl_v
alidationMessage","ValidationRules":[{"ErrorMessage":"The field Album Art URL must be a string with a
maximum length of 1024.","ValidationParameters":
{"max":1024},"ValidationType":"length"}]}],"FormId":"form0","ReplaceValidationSummary":false,"Validatio
nSummaryId":"validationSummary"});
//]]>
</script>
...
```

Task 2 - Enabling Unobtrusive Client Validation

In this task, you will enable jQuery **unobtrusive client validation** from **Web.config** file, which is by default set to false in all new ASP.NET MVC 4 projects. Additionally, you will add the necessary scripts references to make jQuery Unobtrusive Client Validation work.

1. Open **Web.Config** file at project root, and make sure that the **ClientValidationEnabled** and **UnobtrusiveJavaScriptEnabled** keys values are set to **true**.

```
...
<configuration>
  <appSettings>
     <add key="webpages:Version" value="2.0.0.0" />
     <add key="webpages:Enabled" value="false" />
     <add key="PreserveLoginUrl" value="true" />
     <add key="ClientValidationEnabled" value="true" />
     <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
...
```

> **NOTE**
>
> You can also enable client validation by code at Global.asax.cs to get the same results:
>
> **HtmlHelper.ClientValidationEnabled = true;**
>
> Additionally, you can assign ClientValidationEnabled attribute into any controller to have a custom behavior.

2. Open **Create.cshtml** at **Views\StoreManager**.

3. Make sure the following script files, **jquery.validate** and **jquery.validate.unobtrusive**, are referenced in the view through the "**~/bundles/jqueryval**" bundle.

```
...
@section Scripts {
     @Scripts.Render("~/bundles/jqueryval")
}
```

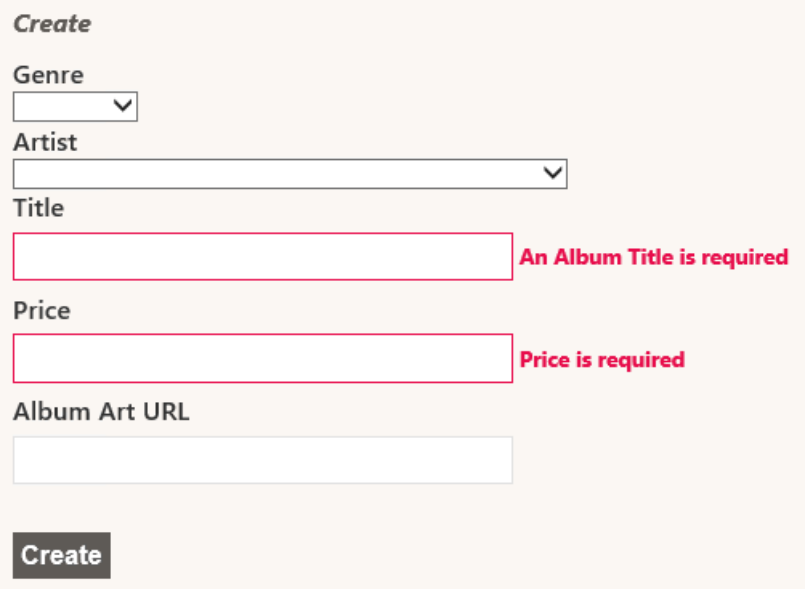
> **NOTE**
>
> All these jQuery libraries are included in MVC 4 new projects. You can find more libraries in the **/Scripts** folder of you project.
>
> In order to make this validation libraries work, you need to add a reference to the jQuery framework library. Since this reference is already added in the **_Layout.cshtml** file, you do not need to add it in this particular view.

Task 3 - Running the Application Using Unobtrusive jQuery Validation

In this task, you will test that the **StoreManager** create view template performs client side validation using jQuery libraries when the user creates a new album.

1. Press **F5** to run the application.

2. The project starts in the Home page. Browse **/StoreManager/Create** and click **Create** without filling the form to verify that you get validation messages:



Client validation with jQuery enabled

3. In the browser, open the source code for Create view:

```
...
<div class="editor-label">
    <label for="Title">Title</label>
</div>
<div class="editor-field">
    <input class="text-box single-line" data-val="true" data-val-length="The field Title must be a
string with a minimum length of 2 and a maximum length of 160." data-val-length-max="160" data-val-
length-min="2" data-val-required="An Album Title is required" id="Title" name="Title" type="text"
value="" />
    <span class="field-validation-valid" data-valmsg-for="Title" data-valmsg-replace="true"></span>
</div>

<div class="editor-label">
    <label for="Price">Price</label>
</div>
<div class="editor-field">
    <input class="text-box single-line" data-val="true" data-val-number="The field Price must be a
number." data-val-range="Price must be between 0.01 and 100.00" data-val-range-max="100" data-val-
range-min="0.01" data-val-required="Price is required" id="Price" name="Price" type="text" value="" />
    <span class="field-validation-valid" data-valmsg-for="Price" data-valmsg-replace="true"></span>
</div>

<div class="editor-label">
    <label for="AlbumArtUrl">Album Art URL</label>
</div>
<div class="editor-field">
    <input class="text-box single-line" data-val="true" data-val-length="The field Album Art URL must
be a string with a maximum length of 1024." data-val-length-max="1024" id="AlbumArtUrl"
name="AlbumArtUrl" type="text" value="" />
    <span class="field-validation-valid" data-valmsg-for="AlbumArtUrl" data-valmsg-replace="true">
</span>
</div>
...
```

NOTE

For each client validation rule, Unobtrusive jQuery adds an attribute with data-val-*rulename*="*message*". Below is a list of tags that Unobtrusive jQuery inserts into the html input field to perform client validation:

- Data-val
- Data-val-number
- Data-val-range
- Data-val-range-min / Data-val-range-max
- Data-val-required
- Data-val-length
- Data-val-length-max / Data-val-length-min

All the data values are filled with model **Data Annotation**. Then, all the logic that works at server side can be run at client side. For example, Price attribute has the following data annotation in the model:

```
[Required(ErrorMessage = "Price is required")]
[Range(0.01, 100.00, ErrorMessage = "Price must be between 0.01 and 100.00")]
public object Price { get; set; }
```

After using Unobtrusive jQuery, the generated code is:

```
<input data-val="true"
data-val-number="The field Price must be a number."
data-val-range="Price must be between 0.01 and 100.00"
data-val-range-max="100"
data-val-range-min="0.01"
data-val-required="Price is required"
id="Album_Price" name="Album.Price" type="text" value="0" />
```

Summary

By completing this Hands-On Lab you have learned how to enable users to change the data stored in the database with the use of the following:

- Controller actions like Index, Create, Edit, Delete
- ASP.NET MVC's scaffolding feature for displaying properties in an HTML table
- Custom HTML helpers to improve user experience
- Action methods that react to either HTTP-GET or HTTP-POST calls
- A shared editor template for similar View templates like Create and Edit
- Form elements like drop-downs
- Data annotations for Model validation
- Client Side Validation using jQuery Unobtrusive library

Appendix A: Installing Visual Studio Express 2012 for Web

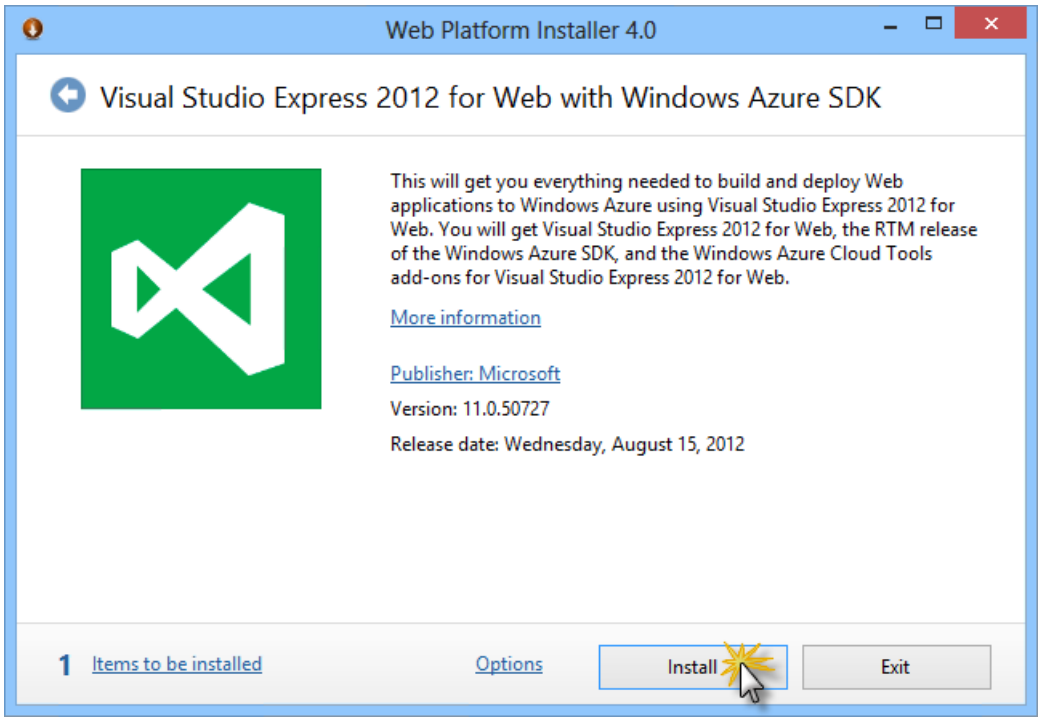
You can install **Microsoft Visual Studio Express 2012 for Web** or another "Express" version using the **Microsoft Web Platform Installer**. The following instructions guide you through the steps required to install *Visual studio Express 2012 for Web* using *Microsoft Web Platform Installer*.

1. Go to https://go.microsoft.com/?linkid=9810169. Alternatively, if you already have installed Web Platform Installer, you can open it and search for the product "*Visual Studio Express 2012 for Web with Windows Azure SDK*".

2. Click on **Install Now**. If you do not have **Web Platform Installer** you will be redirected to download and

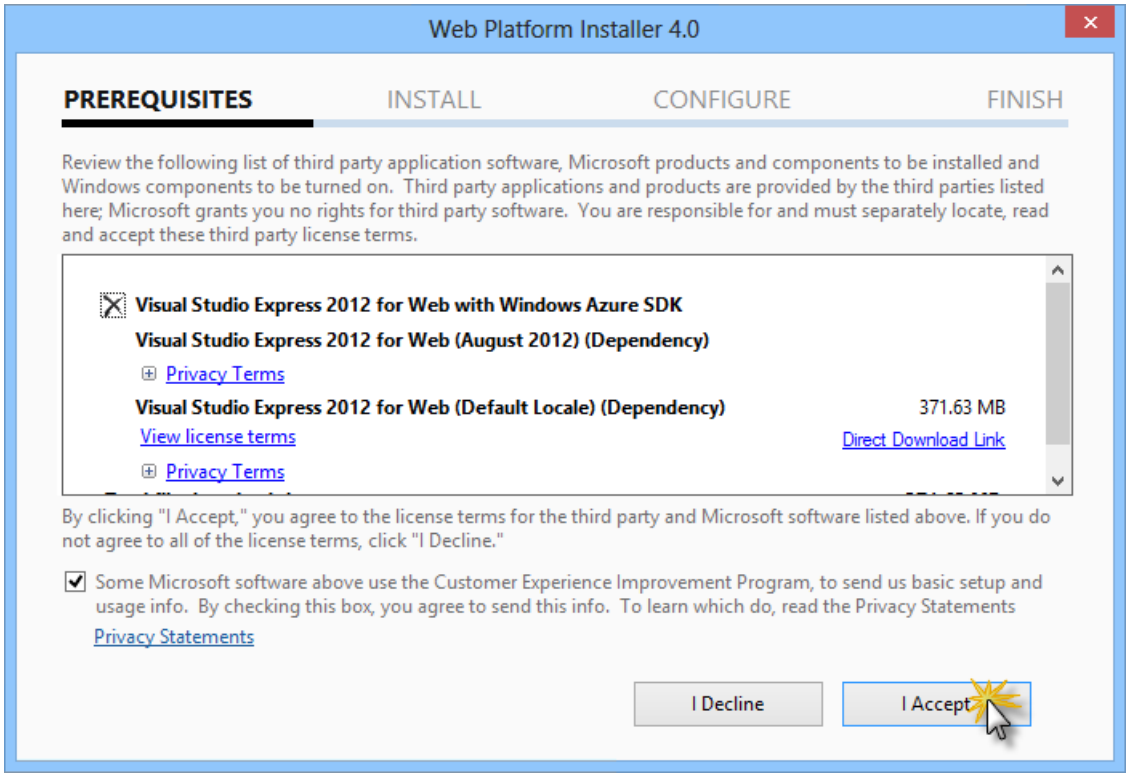
install it first.

3. Once **Web Platform Installer** is open, click **Install** to start the setup.



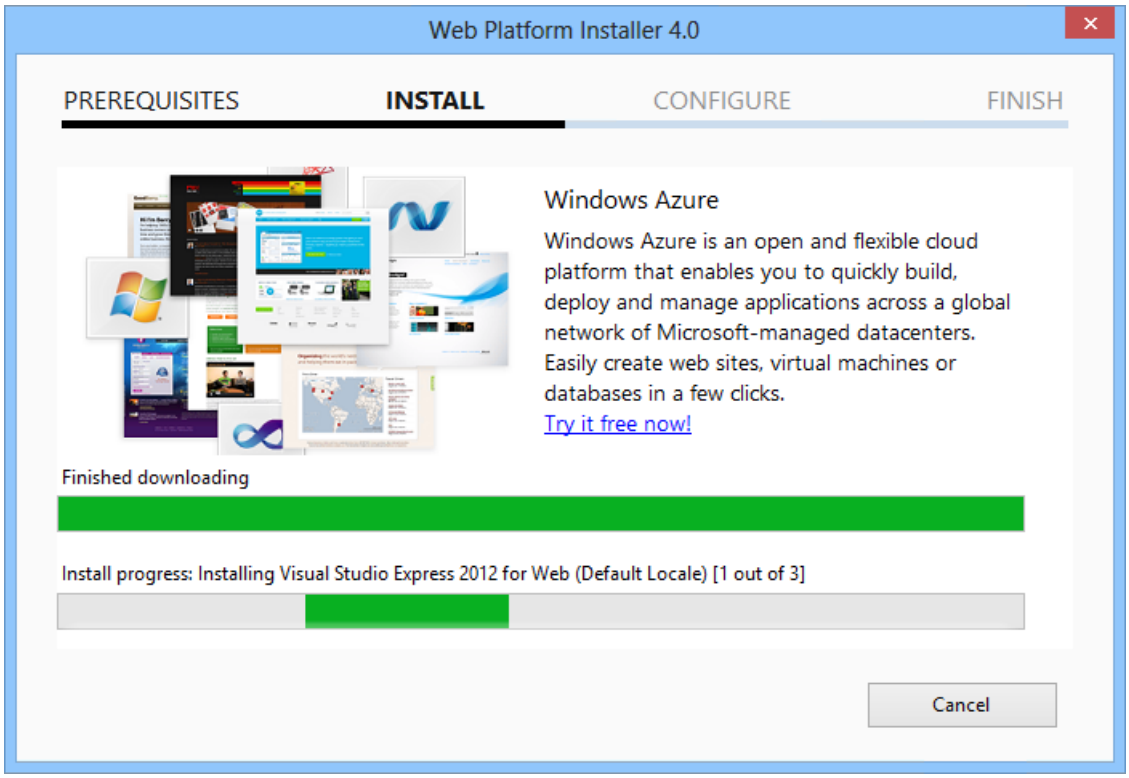
Install Visual Studio Express

4. Read all the products' licenses and terms and click **I Accept** to continue.



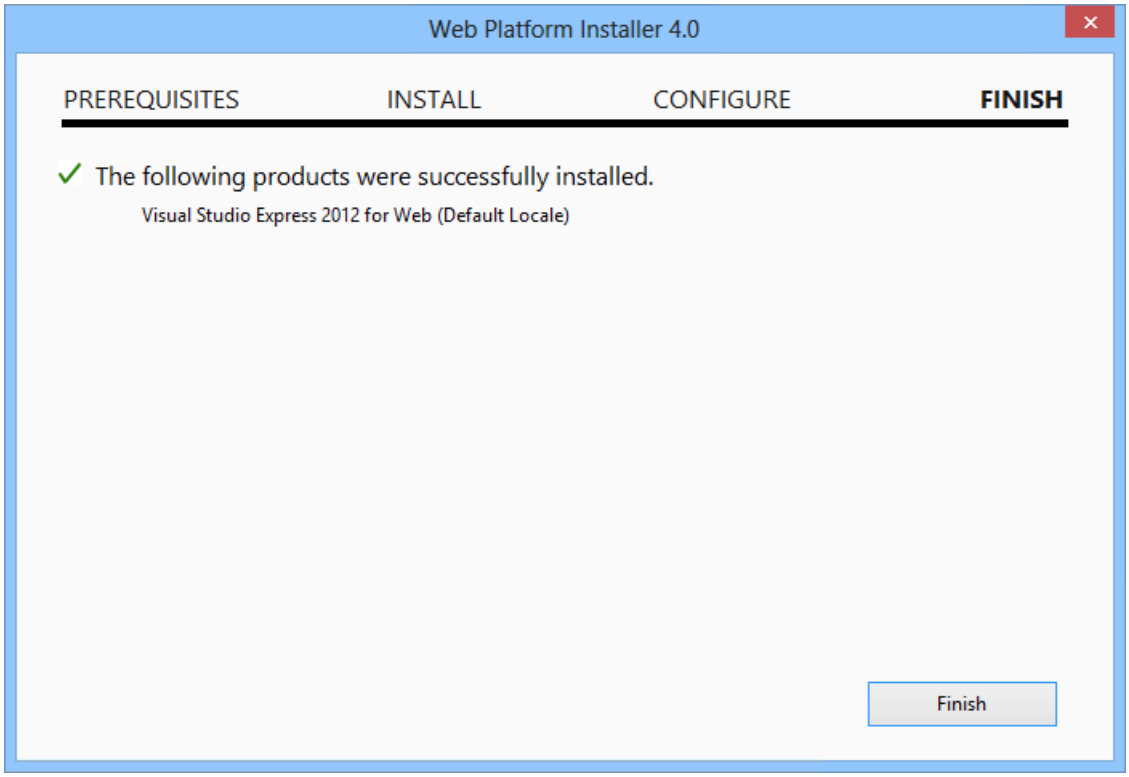
Accepting the license terms

5. Wait until the downloading and installation process completes.



Installation progress

6. When the installation completes, click **Finish**.



Installation completed

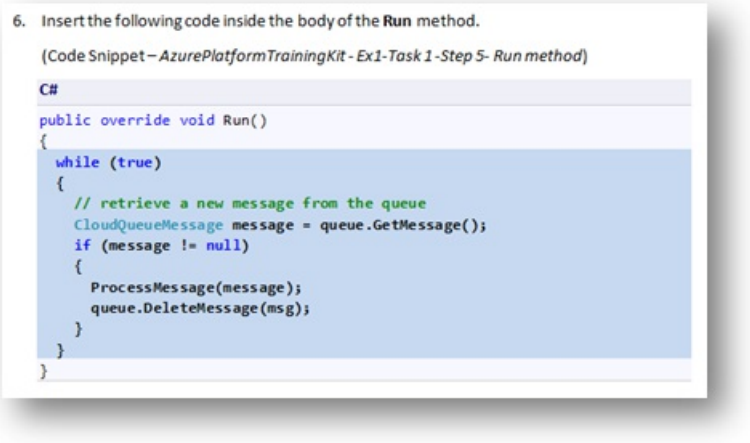
7. Click **Exit** to close Web Platform Installer.

8. To open Visual Studio Express for Web, go to the **Start** screen and start writing "**VS Express**", then click on the **VS Express for Web** tile.

VS Express for Web tile

Appendix B: Using Code Snippets

With code snippets, you have all the code you need at your fingertips. The lab document will tell you exactly when you can use them, as shown in the following figure.



6. Insert the following code inside the body of the **Run** method.

(Code Snippet – *AzurePlatformTrainingKit - Ex1-Task 1 -Step 5- Run method*)

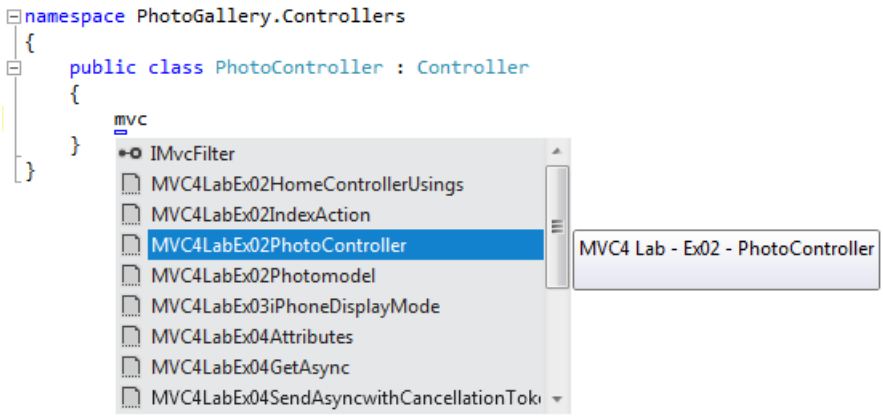
C#

```
public override void Run()
{
  while (true)
  {
    // retrieve a new message from the queue
    CloudQueueMessage message = queue.GetMessage();
    if (message != null)
    {
      ProcessMessage(message);
      queue.DeleteMessage(msg);
    }
  }
}
```

Using Visual Studio code snippets to insert code into your project

To add a code snippet using the keyboard (C# only)

1. Place the cursor where you would like to insert the code.
2. Start typing the snippet name (without spaces or hyphens).
3. Watch as IntelliSense displays matching snippets' names.
4. Select the correct snippet (or keep typing until the entire snippet's name is selected).
5. Press the Tab key twice to insert the snippet at the cursor location.



Start typing the snippet name

```
static void Main(string[] args)
{
    AzureTrainingKitServiceBus
}
```

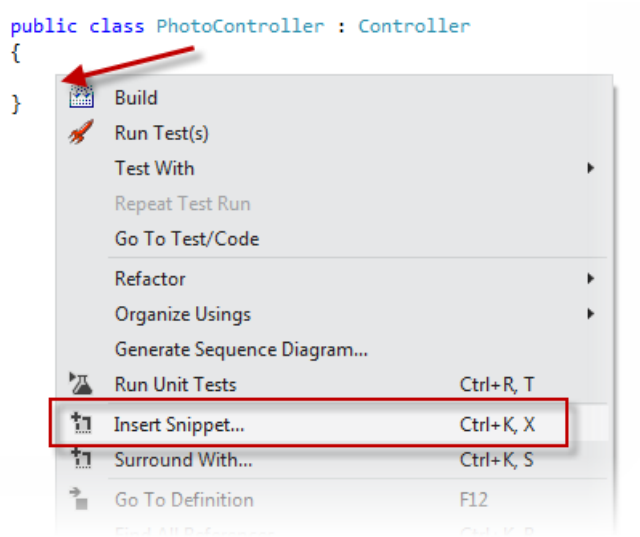
Press Tab to select the highlighted snippet

```
static void Main(string[] args)
{
    request.Headers.Add("$accountKey", accountKey);
    request.Headers.Add("$uniqueUserID", uniqueUserId);
}
```

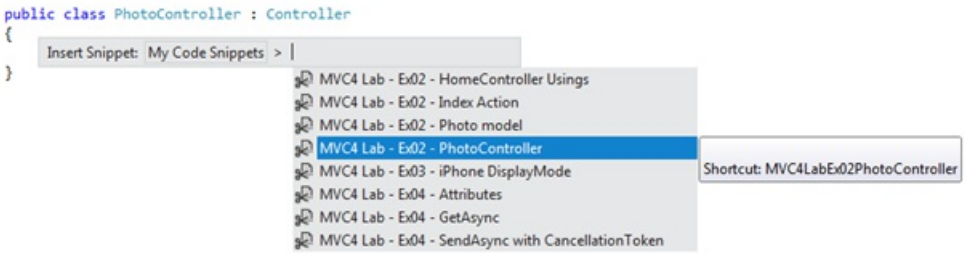
Press Tab again and the snippet will expand

To add a code snippet using the mouse (C#, Visual Basic and XML) 1. Right-click where you want to insert the code snippet.

1. Select **Insert Snippet** followed by **My Code Snippets**.
2. Pick the relevant snippet from the list, by clicking on it.



Right-click where you want to insert the code snippet and select Insert Snippet



Pick the relevant snippet from the list, by clicking on it

ASP.NET MVC 4 Mobile Features

2/20/2020 • 17 minutes to read • Edit Online

by Rick Anderson

> There is now an MVC 5 version of this tutorial with code samples at Deploy an ASP.NET MVC 5 Mobile Web Application on Azure Web Sites.

This tutorial will teach you the basics of how to work with mobile features in an ASP.NET MVC 4 Web application. For this tutorial, you can use Visual Studio Express 2012 or Visual Web Developer 2010 Express Service Pack 1 ("Visual Web Developer or VWD"). You can use the professional version of Visual Studio if you already have that.

Before you start, make sure you've installed the prerequisites listed below.

- Visual Studio Express 2012 (recommended) or Visual Studio Web Developer Express SP1. Visual Studio 2012 contains ASP.NET MVC 4. If you are using Visual Web Developer 2010, you must install ASP.NET MVC 4.

You will also need a mobile browser emulator. Any of the following will work:

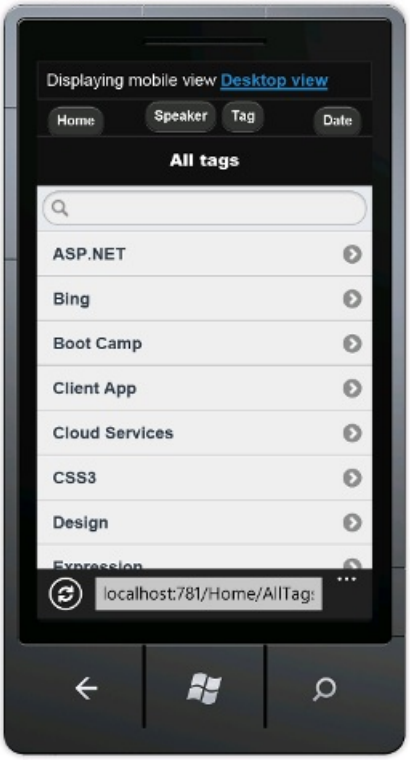
- Windows 7 Phone Emulator. (This is the emulator that's used in most of the screen shots in this tutorial.)
- Change the user agent string to emulate an iPhone. See this blog entry.
- Opera Mobile Emulator
- Apple Safari with the user agent set to iPhone. For instructions on how to set the user agent in Safari to "iPhone", see How to let Safari pretend it's IE on David Alison's blog.

Visual Studio projects with C# source code are available to accompany this topic:

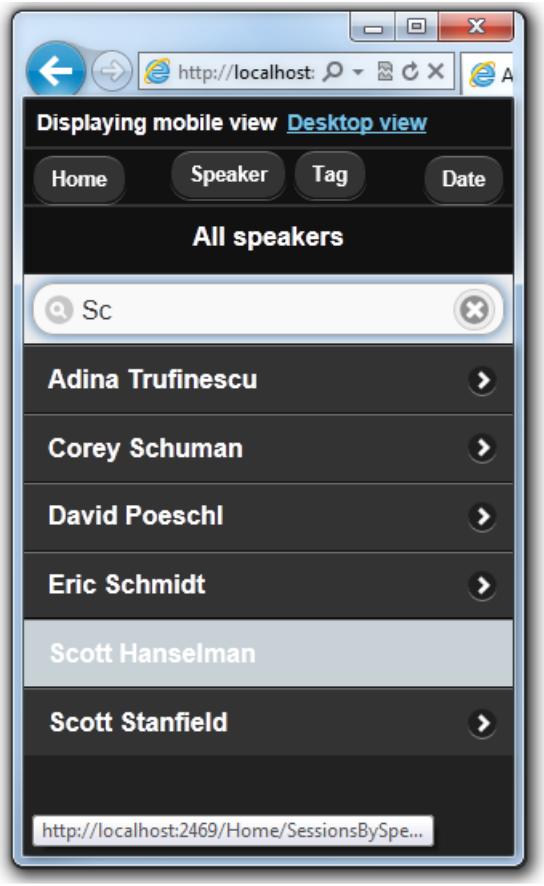
- Starter project download
- Completed project download

What You'll Build

For this tutorial, you'll add mobile features to the simple conference-listing application that's provided in the starter project. The following screenshot shows the tags page of the completed application as seen in the Windows 7 Phone Emulator. See Keyboard Mapping for Windows Phone Emulator to simplify keyboard input.



You can use Internet Explorer version 9 or 10, FireFox or Chrome to develop your mobile application by setting the user agent string. The following image shows the completed tutorial using Internet Explorer emulating an iPhone. You can use the Internet Explorer F-12 developer tools and the Fiddler tool to help debug your application.



Skills You'll Learn

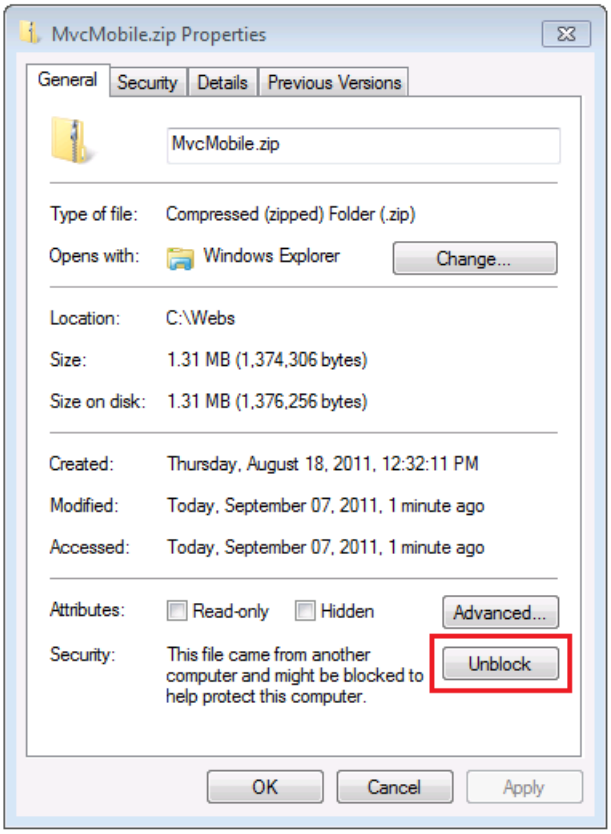
Here's what you'll learn:

- How the ASP.NET MVC 4 templates use the HTML5 `viewport` attribute and adaptive rendering to improve display on mobile devices.
- How to create mobile-specific views.

- How to create a view switcher that lets users toggle between a mobile view and a desktop view of the application.

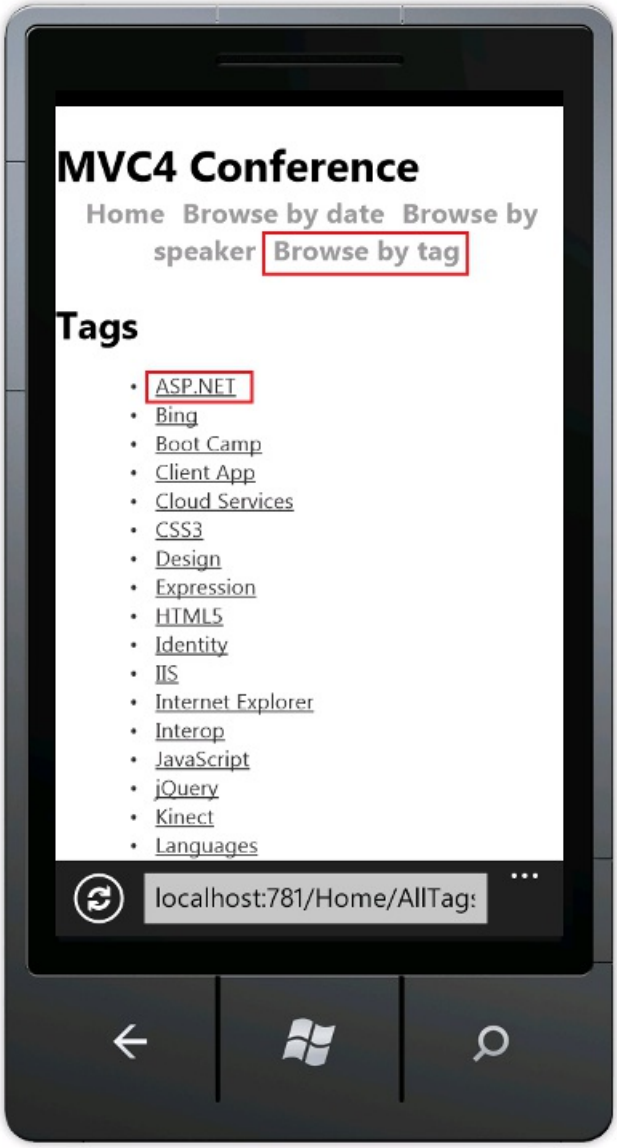
Getting Started

Download the conference-listing application for the starter project using the following link: Download. Then in Windows Explorer, right-click the *MvcMobile.zip* file and choose **Properties**. In the **MvcMobile.zip Properties** dialog box, choose the **Unblock** button. (Unblocking prevents a security warning that occurs when you try to use a *.zip* file that you've downloaded from the web.)



Right-click the *MvcMobile.zip* file and select **Extract All** to unzip the file. In Visual Studio, open the *MvcMobile.sln* file.

Press CTRL+F5 to run the application, which will display it in your desktop browser. Start your mobile browser emulator, copy the URL for the conference application into the emulator, and then click the **Browse by tag** link. If you are using the Windows Phone Emulator, click in the URL bar and press the Pause key to get keyboard access. The image below shows the *AllTags* view (from choosing **Browse by tag**).



The display is very readable on a mobile device. Choose the ASP.NET link.

The ASP.NET tag view is very cluttered. For example, the **Date** column is very difficult to read. Later in the tutorial you'll create a version of the *AllTags* view that's specifically for mobile browsers and that will make the display readable.

Note: Currently a bug exists in the mobile caching engine. For production applications, you must install the Fixed DisplayModes nugget package. See ASP.NET MVC 4 Mobile Caching Bug Fixed for details on the fix.

CSS Media Queries

CSS media queries are an extension to CSS for media types. They allow you to create rules that override the default CSS rules for specific browsers (user agents). A common rule for CSS that targets mobile browsers is defining the maximum screen size. The *Content\Site.css* file that's created when you create a new ASP.NET MVC 4 Internet project contains the following media query:

```
@media only screen and (max-width: 850px) {
```

If the browser window is 850 pixels wide or less, it will use the CSS rules inside this media block. You can use CSS media queries like this to provide a better display of HTML content on small browsers (like mobile browsers) than the default CSS rules that are designed for the wider displays of desktop browsers.

The Viewport Meta Tag

Most mobile browsers define a virtual browser window width (the *viewport*) that's much larger than the actual width of the mobile device. This allows mobile browsers to fit the entire web page inside the virtual display. Users can then zoom in on interesting content. However, if you set the viewport width to the actual device width, no zooming is required, because the content fits in the mobile browser.

The viewport `<meta>` tag in the ASP.NET MVC 4 layout file sets the viewport to the device width. The following line shows the viewport `<meta>` tag in the ASP.NET MVC 4 layout file.

```
<meta name="viewport" content="width=device-width">
```

Examining the Effect of CSS Media Queries and the Viewport Meta Tag

Open the *Views\Shared_Layout.cshtml* file in the editor and comment out the viewport `<meta>` tag. The following markup shows the commented-out line.

```
@*<meta name="viewport" content="width=device-width">*@
```

Open the *MvcMobile\Content\Site.css* file in the editor and change the maximum width in the media query to zero pixels. This will prevent the CSS rules from being used in mobile browsers. The following line shows the modified media query:

```
@media only screen and (max-width: 0px) { ...
```

Save your changes and browse to the Conference application in a mobile browser emulator. The tiny text in the following image is the result of removing the viewport `<meta>` tag. With no viewport `<meta>` tag, the browser is zooming out to the default viewport width (850 pixels or wider for most mobile browsers.)

Undo your changes — uncomment the viewport `<meta>` tag in the layout file and restore the media query to 850 pixels in the *Site.css* file. Save your changes and refresh the mobile browser to verify that the mobile-friendly display has been restored.

The viewport `<meta>` tag and the CSS media query are not specific to ASP.NET MVC 4, and you can take advantage of these features in any web application. But they are now built into the files that are generated when you create a new ASP.NET MVC 4 project.

For more information about the viewport `<meta>` tag, see A tale of two viewports — part two.

In the next section you'll see how to provide mobile-browser specific views.

Overriding Views, Layouts, and Partial Views

A significant new feature in ASP.NET MVC 4 is a simple mechanism that lets you override any view (including layouts and partial views) for mobile browsers in general, for an individual mobile browser, or for any specific browser. To provide a mobile-specific view, you can copy a view file and add *.Mobile* to the file name. For example, to create a mobile *Index* view, copy *Views\Home\Index.cshtml* to *Views\Home\Index.Mobile.cshtml*.

In this section, you'll create a mobile-specific layout file.

To start, copy *Views\Shared_Layout.cshtml* to *Views\Shared_Layout.Mobile.cshtml*. Open *_Layout.Mobile.cshtml* and change the title from **MVC4 Conference** to **Conference (Mobile)**.

In each `Html.ActionLink` call, remove "Browse by" in each link *ActionLink*. The following code shows the completed body section of the mobile layout file.

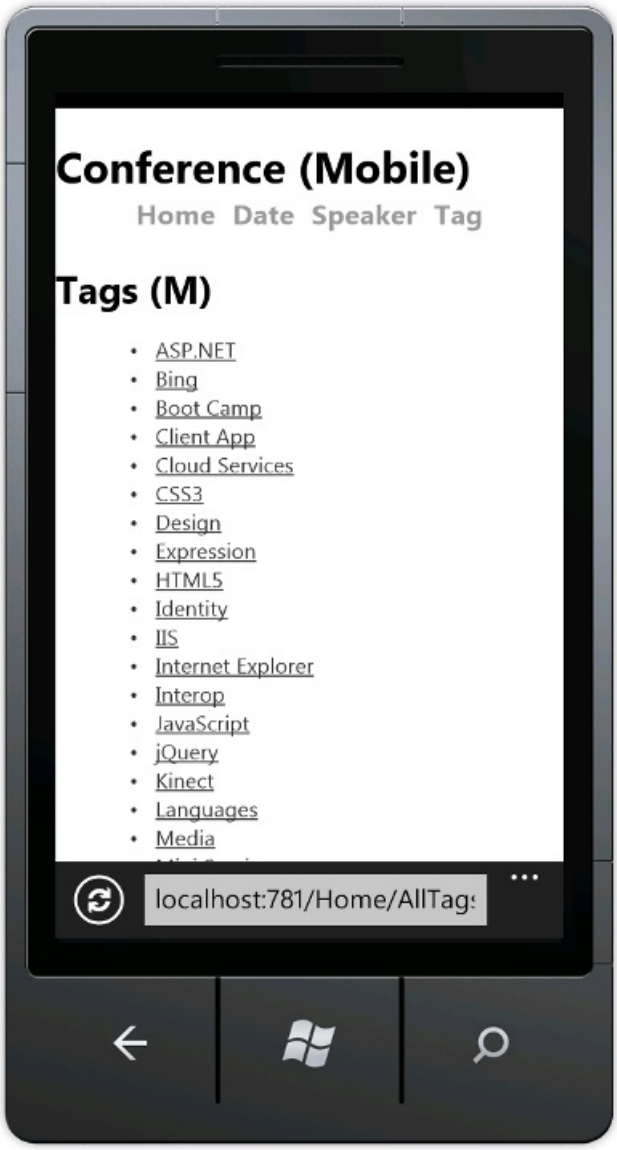
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>
    <div id="title">
        <h1> Conference (Mobile)</h1>
    </div>
    <div id="menucontainer">
        <ul id="menu">
            <li>@Html.ActionLink("Home", "Index", "Home")</li>
            <li>@Html.ActionLink("Date", "AllDates", "Home")</li>
            <li>@Html.ActionLink("Speaker", "AllSpeakers", "Home")</li>
            <li>@Html.ActionLink("Tag", "AllTags", "Home")</li>
        </ul>
    </div>
    @RenderBody()

    @Scripts.Render("~/bundles/jquery")
    @RenderSection("scripts", required: false)
</body>
</html>
```

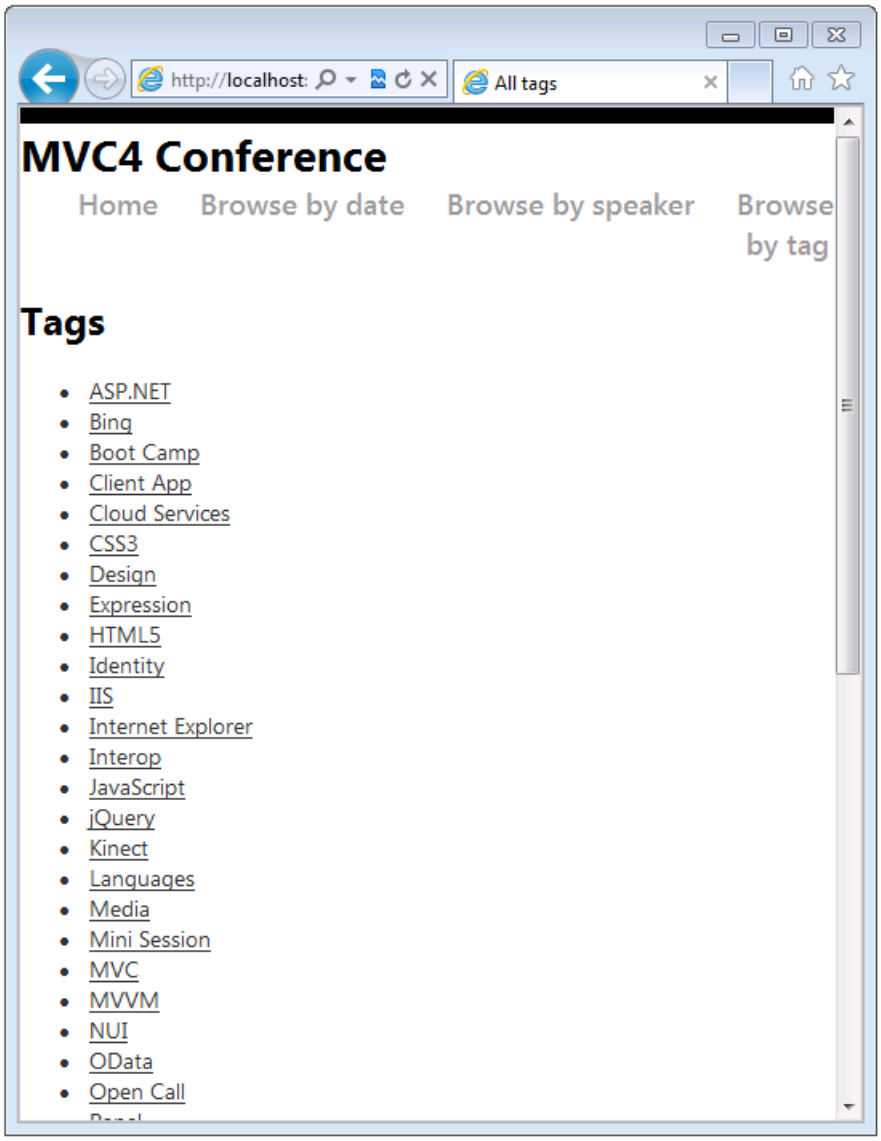
Copy the *Views\Home\AllTags.cshtml* file to *Views\Home\AllTags.Mobile.cshtml*. Open the new file and change the `<h2>` element from "Tags" to "Tags (M)":

```
<h2>Tags (M)</h2>
```

Browse to the tags page using a desktop browser and using mobile browser emulator. The mobile browser emulator shows the two changes you made.



In contrast, the desktop display has not changed.



Browser-Specific Views

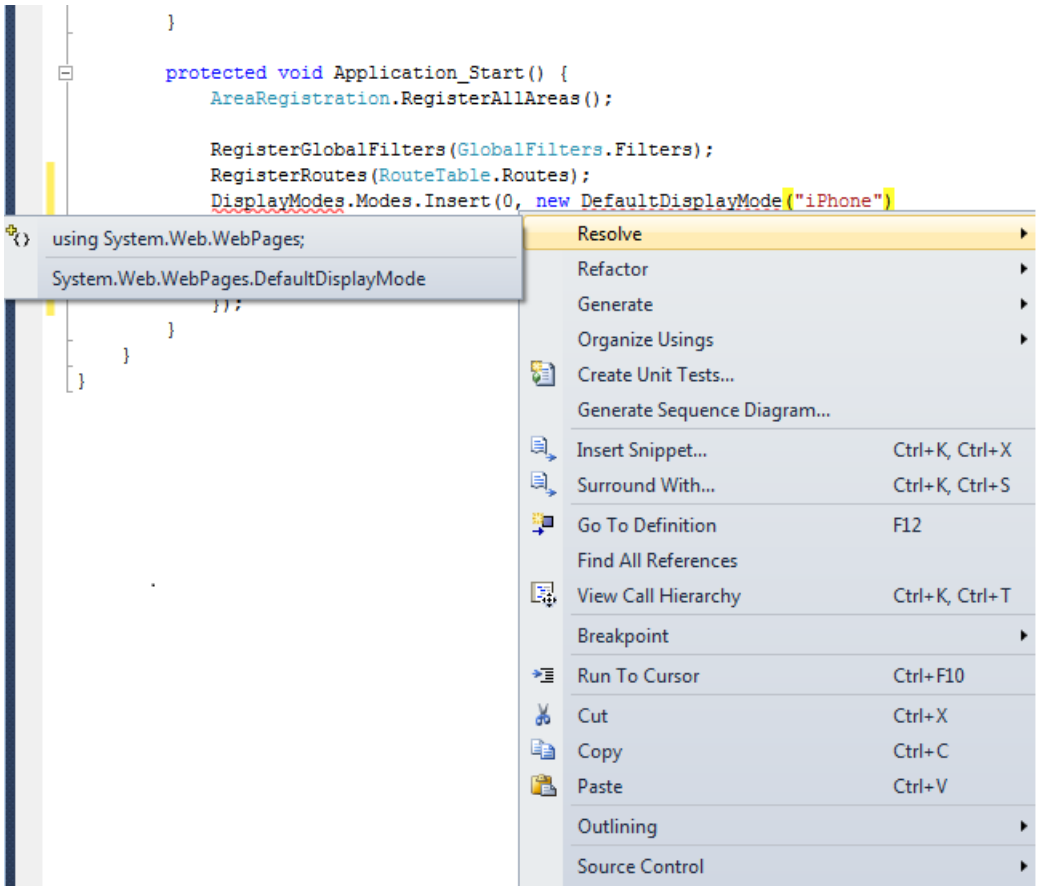
In addition to mobile-specific and desktop-specific views, you can create views for an individual browser. For example, you can create views that are specifically for the iPhone browser. In this section, you'll create a layout for the iPhone browser and an iPhone version of the *AllTags* view.

Open the *Global.asax* file and add the following code to the `Application_Start` method.

```
DisplayModeProvider.Instance.Modes.Insert(0, new DefaultDisplayMode("iPhone")
{
    ContextCondition = (context => context.GetOverriddenUserAgent().IndexOf
        ("iPhone", StringComparison.OrdinalIgnoreCase) >= 0)
});
```

This code defines a new display mode named "iPhone" that will be matched against each incoming request. If the incoming request matches the condition you defined (that is, if the user agent contains the string "iPhone"), ASP.NET MVC will look for views whose name contains the "iPhone" suffix.

In the code, right-click `DefaultDisplayMode`, choose **Resolve**, and then choose `using System.Web.WebPages;`. This adds a reference to the `System.Web.WebPages` namespace, which is where the `DisplayModes` and `DefaultDisplayMode` types are defined.



Alternatively, you can just manually add the following line to the `using` section of the file.

```
using System.Web.WebPages;
```

The complete contents of the *Global.asax* file is shown below.

```
using System;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using System.Web.WebPages;

namespace MvcMobile
{

    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            DisplayModeProvider.Instance.Modes.Insert(0, new DefaultDisplayMode("iPhone")
            {
                ContextCondition = (context => context.GetOverriddenUserAgent().IndexOf
                    ("iPhone", StringComparison.OrdinalIgnoreCase) >= 0)
            });

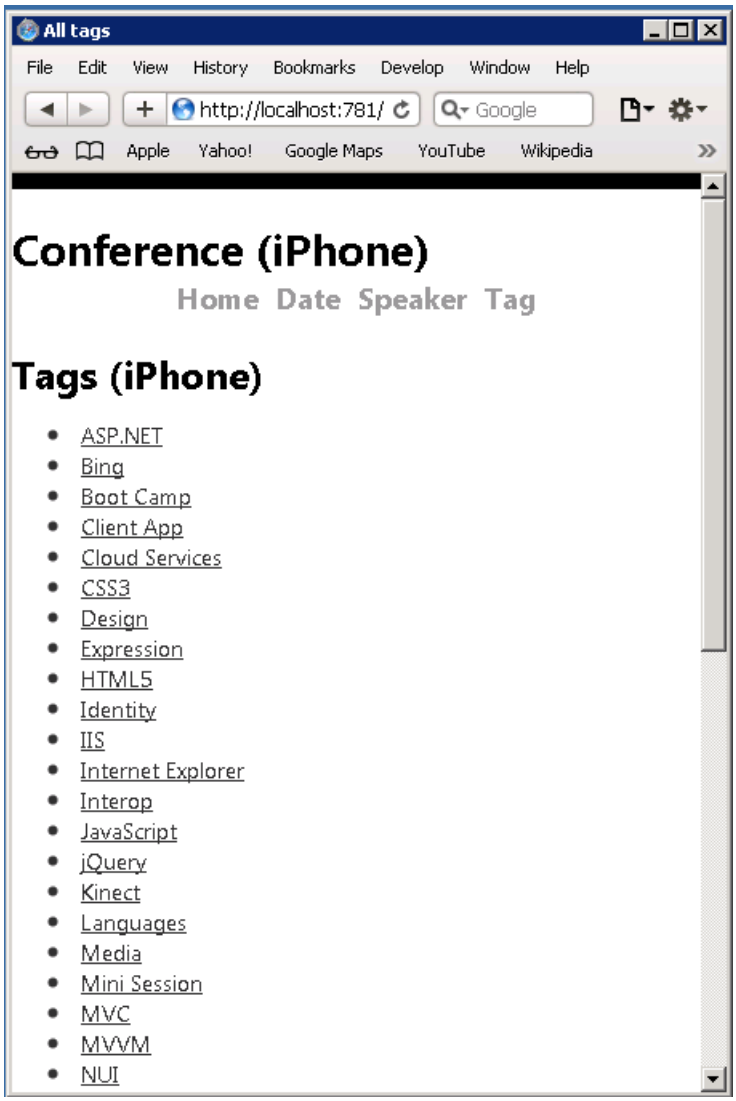
            AreaRegistration.RegisterAllAreas();

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

Save the changes. Copy the *MvcMobile\Views\Shared_Layout.Mobile.cshtml* file to *MvcMobile\Views\Shared_Layout.iPhone.cshtml*. Open the new file and then change the `h1` heading from `Conference (Mobile)` to `Conference (iPhone)`.

Copy the *MvcMobile\Views\Home\AllTags.Mobile.cshtml* file to *MvcMobile\Views\Home\AllTags.iPhone.cshtml*. In the new file, change the `<h2>` element from "Tags (M)" to "Tags (iPhone)".

Run the application. Run a mobile browser emulator, make sure its user agent is set to "iPhone", and browse to the *AllTags* view. The following screenshot shows the *AllTags* view rendered in the Safari browser. You can download Safari for Windows here.



In this section we've seen how to create mobile layouts and views and how to create layouts and views for specific devices such as the iPhone. In the next section you'll see how to leverage jQuery Mobile for more compelling mobile views.

Using jQuery Mobile

The jQuery Mobile library provides a user interface framework that works on all the major mobile browsers. jQuery Mobile applies *progressive enhancement* to mobile browsers that support CSS and JavaScript. Progressive enhancement allows all browsers to display the basic content of a web page, while allowing more powerful browsers and devices to have a richer display. The JavaScript and CSS files that are included with jQuery Mobile style many elements to fit mobile browsers without making any markup changes.

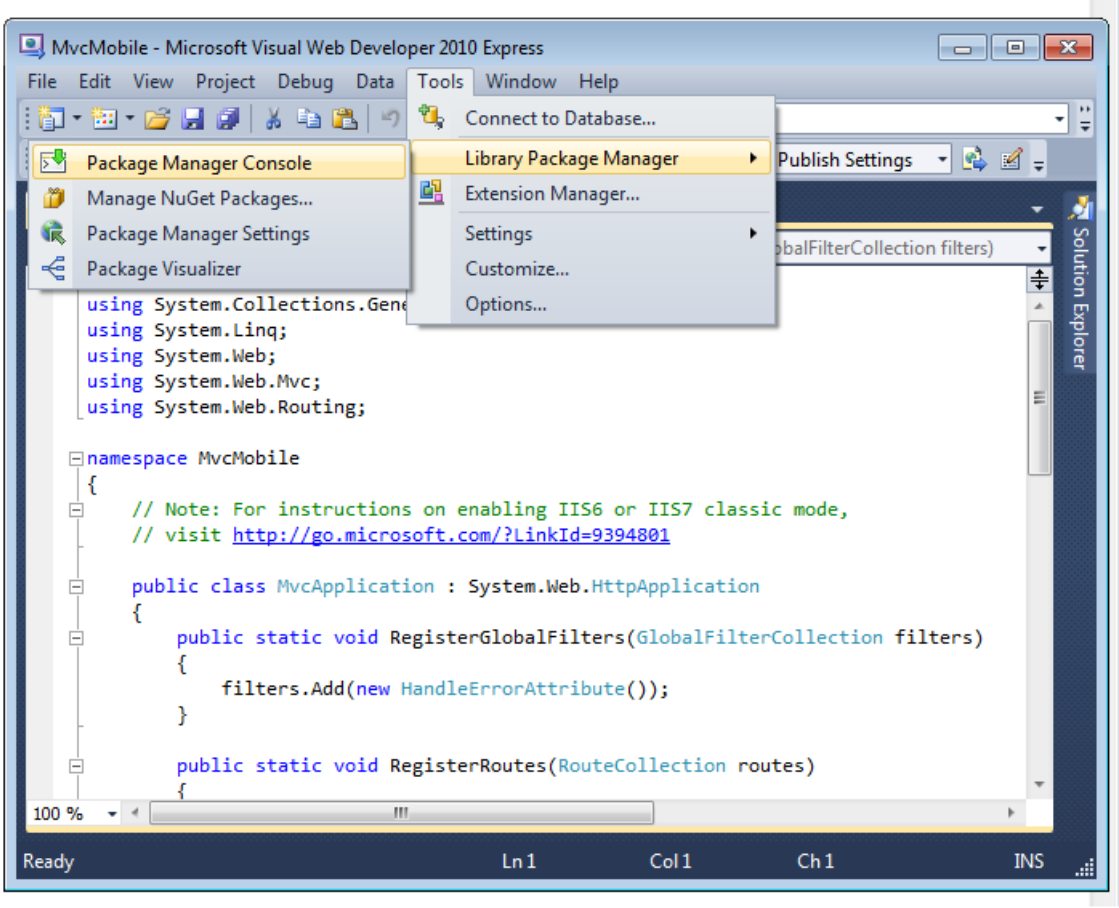
In this section you'll install the *jQuery.Mobile.MVC* NuGet package, which installs jQuery Mobile and a view-switcher widget.

To start, delete the *Shared_Layout.Mobile.cshtml* and *Shared_Layout.iPhone.cshtml* files that you created earlier.

Rename *Views\Home\AllTags.Mobile.cshtml* and *Views\Home\AllTags.iPhone.cshtml* files to *Views\Home\AllTags.iPhone.cshtml.hide* and *Views\Home\AllTags.Mobile.cshtml.hide*. Because the files no longer have a *.cshtml* extension, they won't be used by the ASP.NET MVC runtime to render the *AllTags* view.

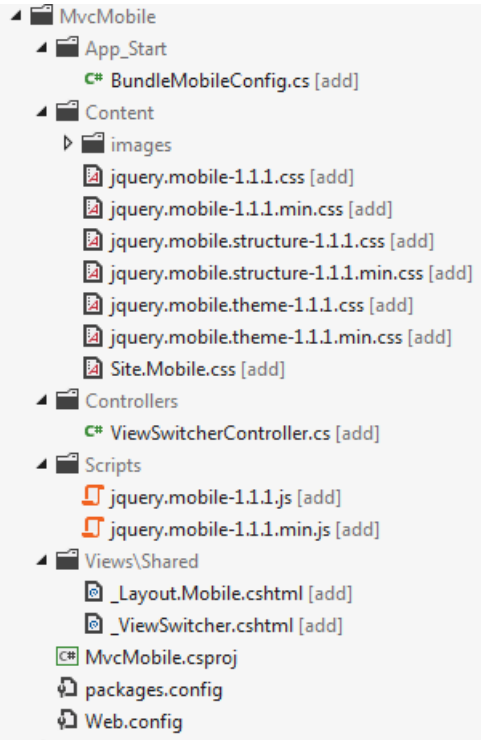
Install the *jQuery.Mobile.MVC* NuGet package by doing this:

1. From the **Tools** menu, select **NuGet Package Manager**, and then select **Package Manager Console**.



2. In the **Package Manager Console**, enter `Install-Package jQuery.Mobile.MVC -version 1.0.0`

The following image shows the files added and changed to the MvcMobile project by the NuGet jQuery.Mobile.MVC package. Files which are added have [add] appended after the file name. The image does not show the GIF and PNG files added to the *Content\images* folder.



The jQuery.Mobile.MVC NuGet package installs the following:

- The *App_Start\BundleMobileConfig.cs* file, which is needed to reference the jQuery JavaScript and CSS files added. You must follow the instructions below and reference the mobile bundle defined in this file.

- jQuery Mobile CSS files.
- A `ViewSwitcher` controller widget (*Controllers\ViewSwitcherController.cs*).
- jQuery Mobile JavaScript files.
- A jQuery Mobile-styled layout file (*Views\Shared_Layout.Mobile.cshtml*).
- A view-switcher partial view *(MvcMobile\Views\Shared_ViewSwitcher.cshtml*) that provides a link at the top of each page to switch from desktop view to mobile view and vice versa.
- Several *.png* and *.gif* image files in the *Content\images* folder.

Open the *Global.asax* file and add the following code as the last line of the `Application_Start` method.

```
BundleMobileConfig.RegisterBundles(BundleTable.Bundles);
```

The following code shows the complete *Global.asax* file.

```
using System;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using System.Web.WebPages;

namespace MvcMobile
{

    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            DisplayModeProvider.Instance.Modes.Insert(0, new DefaultDisplayMode("iPhone")
            {
                ContextCondition = (context => context.GetOverriddenUserAgent().IndexOf
                    ("iPhone", StringComparison.OrdinalIgnoreCase) >= 0)
            });
            AreaRegistration.RegisterAllAreas();

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            BundleMobileConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

NOTE

If you are using Internet Explorer 9 and you don't see the `BundleMobileConfig` line above in yellow highlight, click the Compatibility View button![Picture of the Compatibility View button (off)](https://res2.windows.microsoft.com/resbox/en/Windows 7/main/f080e77f-9b66-4ac8-9af0-803c4f8a859c_15.jpg "Picture of the Compatibility View button (off)") in IE to make the icon change from an outline ![Picture of the Compatibility View button (off)](https://res2.windows.microsoft.com/resbox/en/Windows 7/main/f080e77f-9b66-4ac8-9af0-803c4f8a859c_15.jpg "Picture of the Compatibility View button (off)") to a solid color ![Picture of the Compatibility View button (on)](https://res1.windows.microsoft.com/resbox/en/Windows 7/main/156805ff-3130-481b-a12d-4d3a96470f36_14.jpg "Picture of the Compatibility View button (on)"). Alternatively you can view this tutorial in FireFox or Chrome.

Open the *MvcMobile\Views\Shared_Layout.Mobile.cshtml* file and add the following markup directly after the

`Html.Partial` call:

```
<div data-role="header" align="center">
    @Html.ActionLink("Home", "Index", "Home")
    @Html.ActionLink("Date", "AllDates")
    @Html.ActionLink("Speaker", "AllSpeakers")
    @Html.ActionLink("Tag", "AllTags")
</div>
```

The complete *MvcMobile\Views\Shared_Layout.Mobile.cshtml* file is shown below:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
    <meta name="viewport" content="width=device-width" />
    @Styles.Render("~/Content/Mobile/css", "~/Content/jquerymobile/css")
    @Scripts.Render("~/bundles/jquery", "~/bundles/jquerymobile")
</head>
<body>

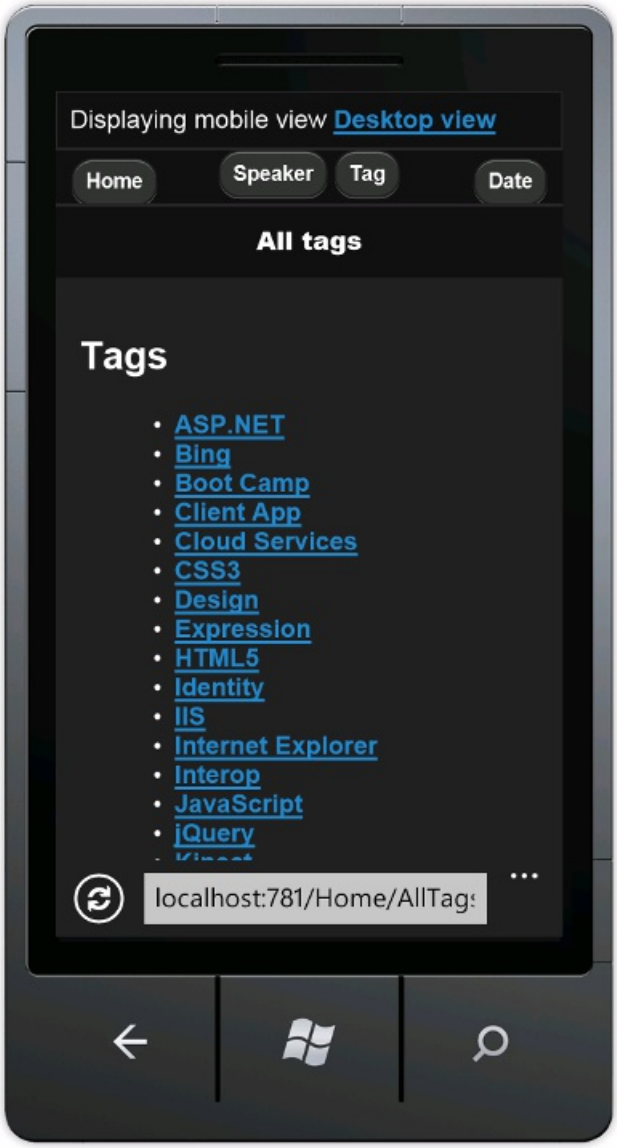
    <div data-role="page" data-theme="a">
        @Html.Partial("_ViewSwitcher")
        <div data-role="header" align="center">
            @Html.ActionLink("Home", "Index", "Home")
            @Html.ActionLink("Date", "AllDates")
            @Html.ActionLink("Speaker", "AllSpeakers")
            @Html.ActionLink("Tag", "AllTags")
        </div>
        <div data-role="header">
            <h1>@ViewBag.Title</h1>
        </div>

        <div data-role="content">
            @RenderSection("featured", false)
            @RenderBody()
        </div>

    </div>

</body>
</html>
```

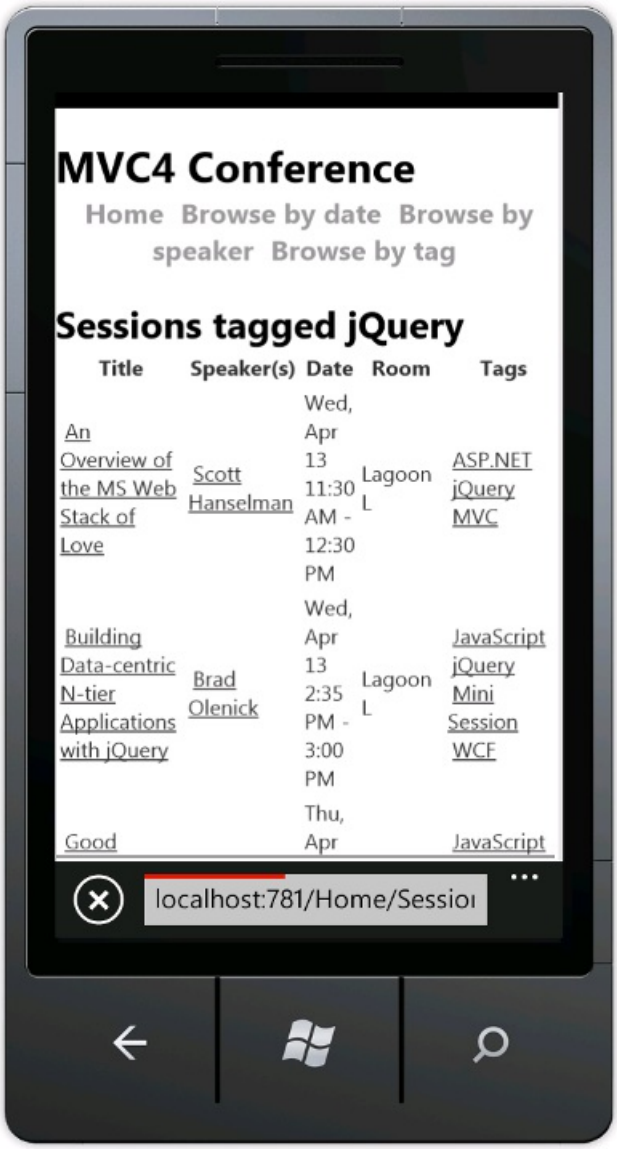
Build the application, and in your mobile browser emulator browse to the *AllTags* view. You see the following:



NOTE

You can debug the mobile specific code by setting the user agent string for IE or Chrome to iPhone and then using the F-12 developer tools. If your mobile browser doesn't display the **Home**, **Speaker**, **Tag**, and **Date** links as buttons, the references to jQuery Mobile scripts and CSS files are probably not correct.

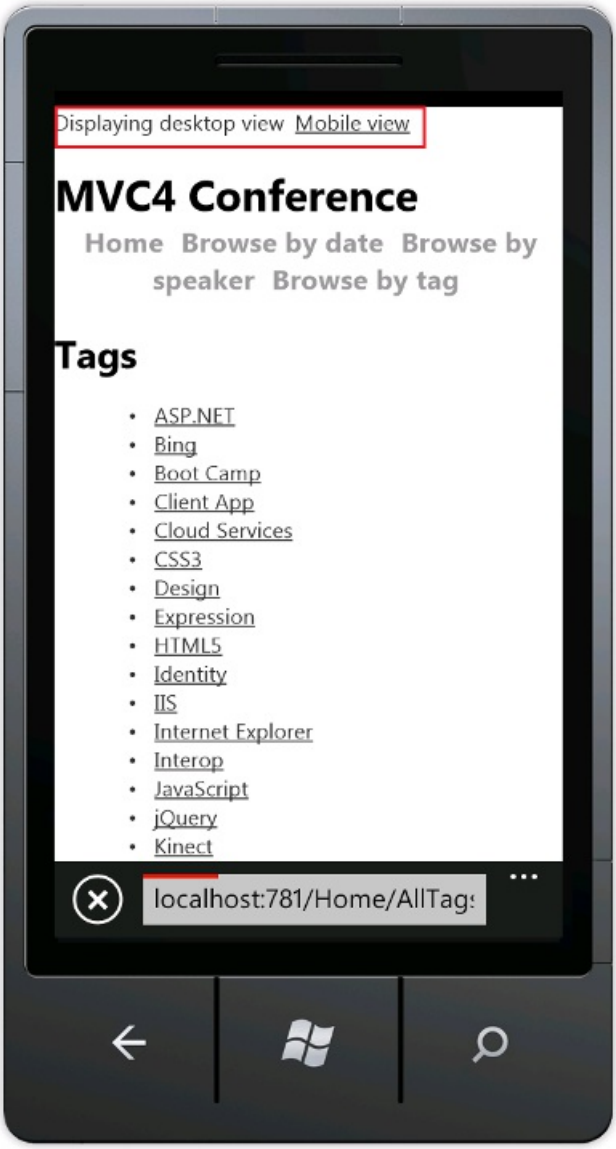
In addition to the style changes, you see **Displaying mobile view** and a link that lets you switch from mobile view to desktop view. Choose the **Desktop view** link, and the desktop view is displayed.



The desktop view doesn't provide a way to directly navigate back to the mobile view. You'll fix that now. Open the *Views\Shared_Layout.cshtml* file. Just under the page `body` element, add the following code, which renders the view-switcher widget:

```
@Html.Partial("_ViewSwitcher")
```

Refresh the *AllTags* view in the mobile browser. You can now navigate between desktop and mobile views.



NOTE

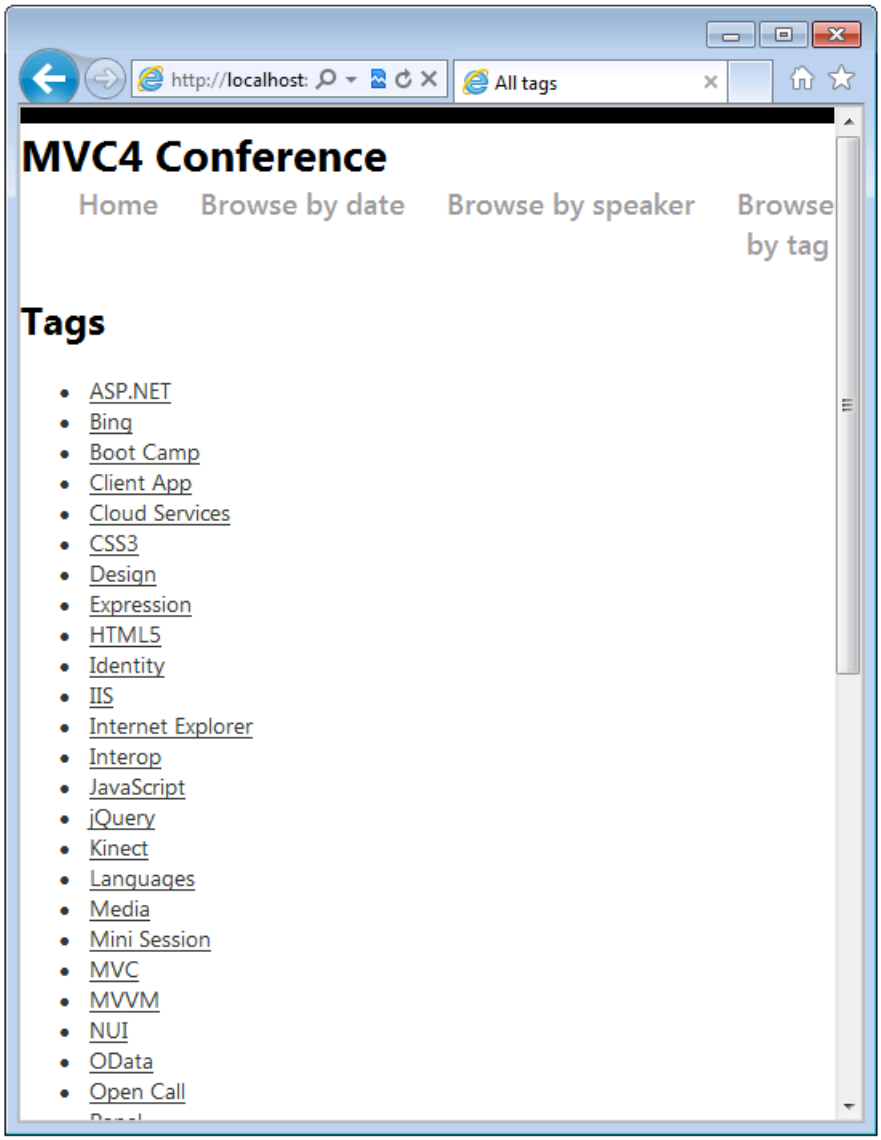
Debug note: You can add the following code to the end of the Views\Shared_ViewSwitcher.cshtml to help debug views when using a browser the user agent string set to a mobile device.

```
else
{
     @:Not Mobile/Get
}
```

and adding the following heading to the *Views\Shared_Layout.cshtml* file.

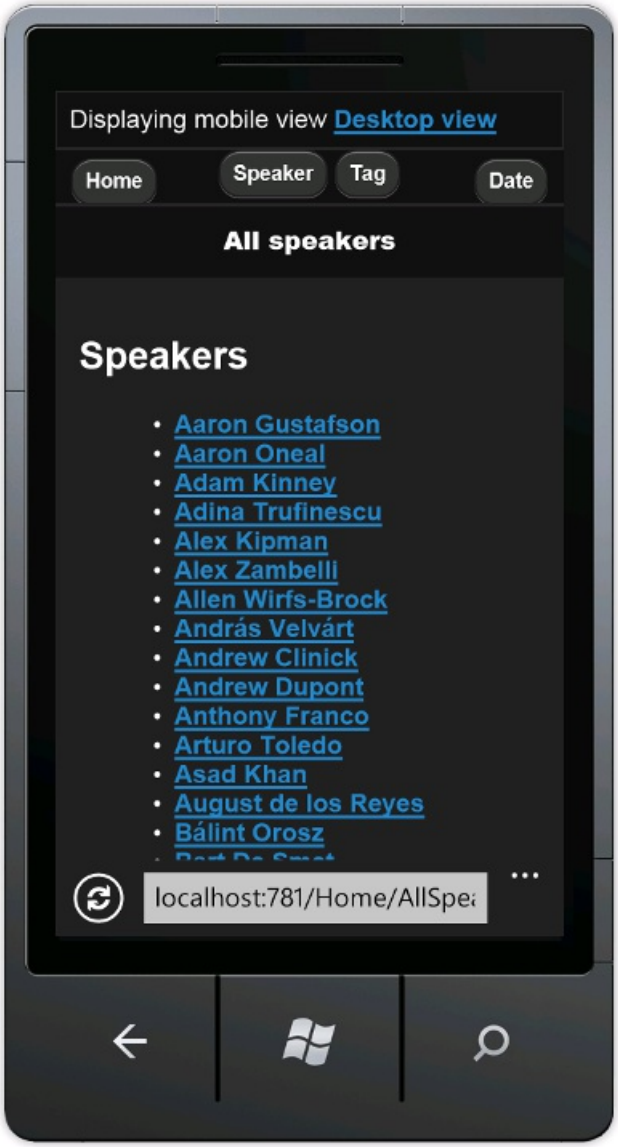
```
<h1> Non Mobile Layout MVC4 Conference </h1>
```

Browse to the *AllTags* page in a desktop browser. The view-switcher widget is not displayed in a desktop browser because it's added only to the mobile layout page. Later in the tutorial you'll see how you can add the view-switcher widget to the desktop view.



Improving the Speakers List

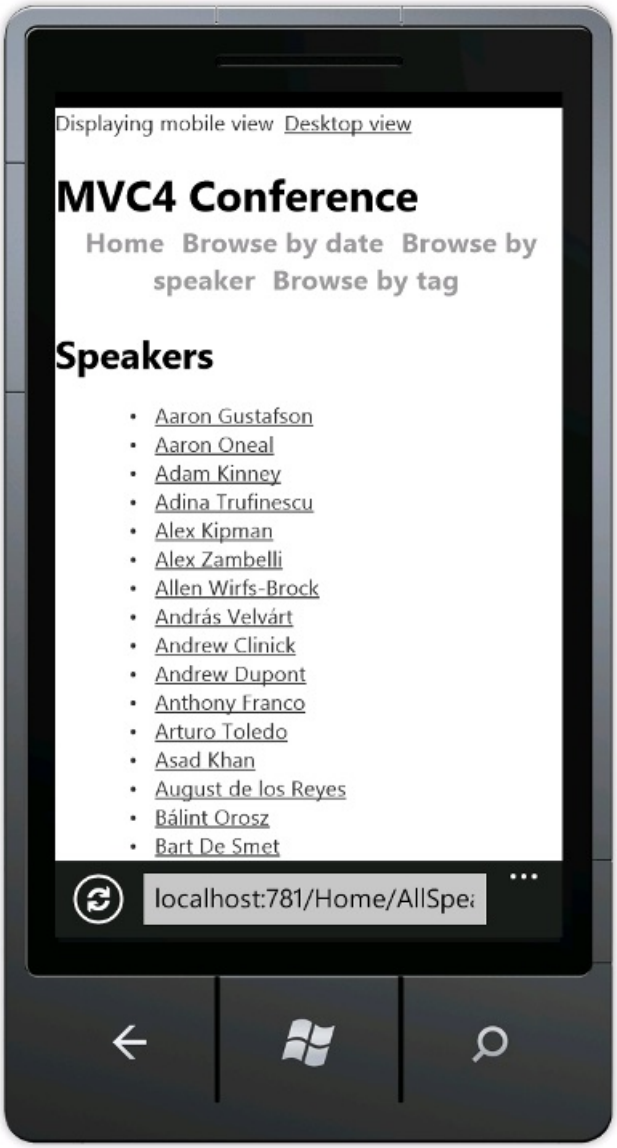
In the mobile browser, select the **Speakers** link. Because there's no mobile view(*AllSpeakers.Mobile.cshtml*), the default speakers display (*AllSpeakers.cshtml*) is rendered using the mobile layout view (*_Layout.Mobile.cshtml*).



You can globally disable a default (non-mobile) view from rendering inside a mobile layout by setting `RequireConsistentDisplayMode` to `true` in the *Views_ViewStart.cshtml* file, like this:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    DisplayModeProvider.Instance.RequireConsistentDisplayMode = true;
}
```

When `RequireConsistentDisplayMode` is set to `true`, the mobile layout (*_Layout.Mobile.cshtml*) is used only for mobile views. (That is, the view file is of the form ***ViewName.Mobile.cshtml*.) You might want to set `RequireConsistentDisplayMode` to `true` if your mobile layout doesn't work well with your non-mobile views. The screenshot below shows how the *Speakers* page renders when `RequireConsistentDisplayMode` is set to `true`.



You can disable consistent display mode in a view by setting `RequireConsistentDisplayMode` to `false` in the view file. The following markup in the *Views\Home\AllSpeakers.cshtml* file sets `RequireConsistentDisplayMode` to `false` :

```
@model IEnumerable<string>

@{
    ViewBag.Title = "All speakers";
    DisplayModeProvider.Instance.RequireConsistentDisplayMode = false;
}
```

Creating a Mobile Speakers View

As you just saw, the *Speakers* view is readable, but the links are small and are difficult to tap on a mobile device. In this section, you'll create a mobile-specific *Speakers* view that looks like a modern mobile application — it displays large, easy-to-tap links and contains a search box to quickly find speakers.

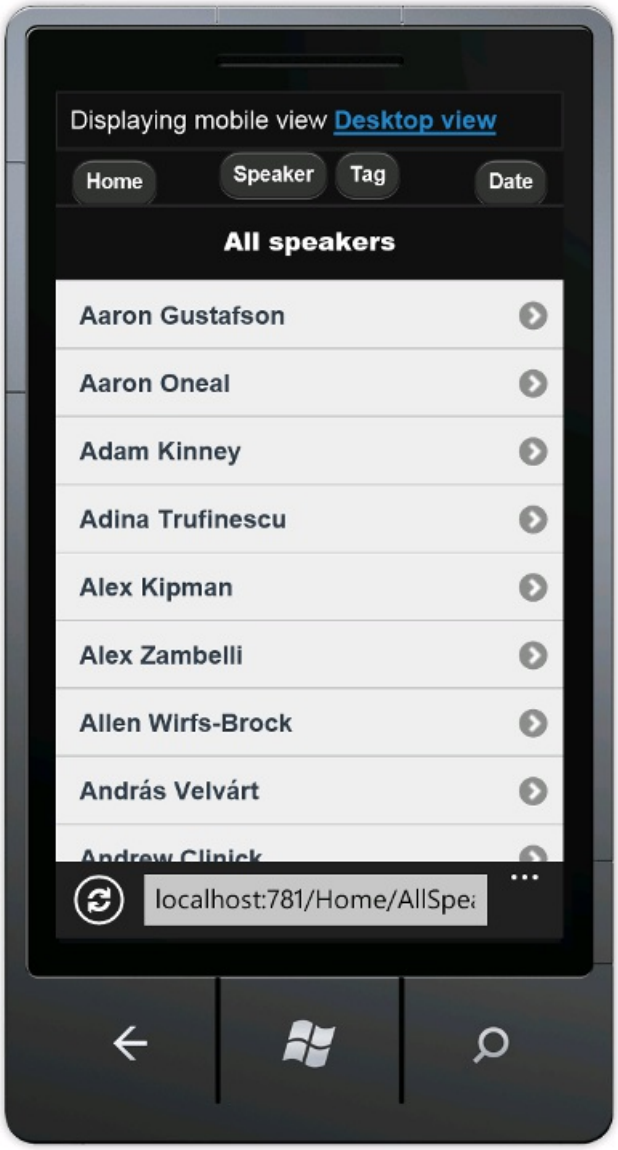
Copy *AllSpeakers.cshtml* to *AllSpeakers.Mobile.cshtml*. Open the *AllSpeakers.Mobile.cshtml* file and remove the `<h2>` heading element.

In the `<ul>` tag, add the `data-role` attribute and set its value to `listview`. Like other `data-*` attributes, `data-role="listview"` makes the large list items easier to tap. This is what the completed markup looks like:

```
@model IEnumerable<string>

@{
    ViewBag.Title = "All speakers";
}
<ul data-role="listview">
    @foreach(var speaker in Model) {
        <li>@Html.ActionLink(speaker, "SessionsBySpeaker", new { speaker })</li>
    }
</ul>
```

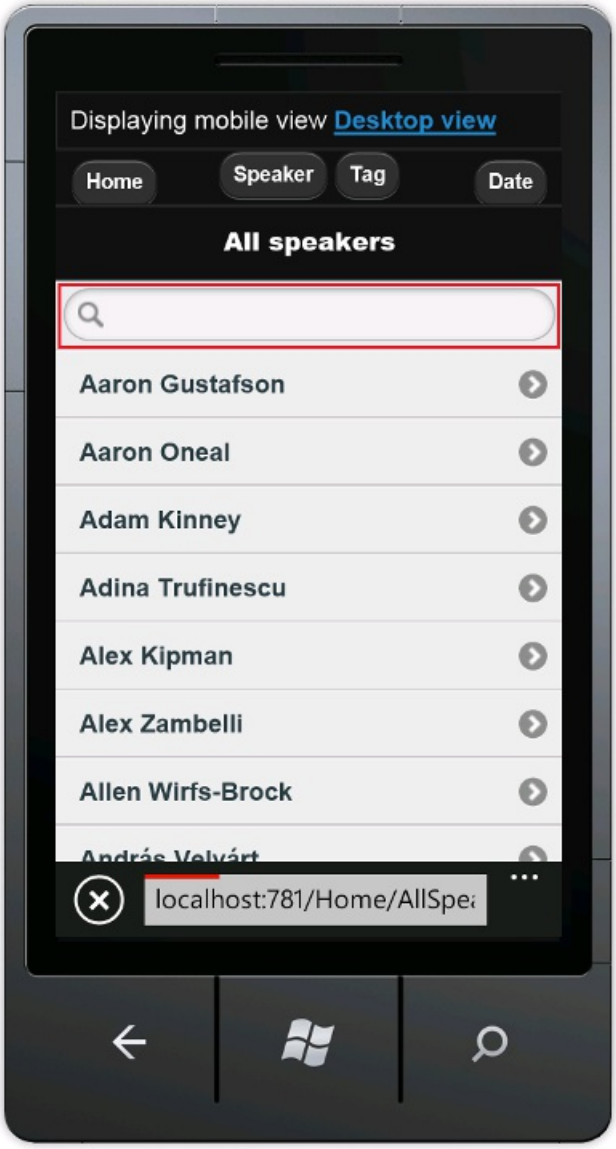
Refresh the mobile browser. The updated view looks like this:



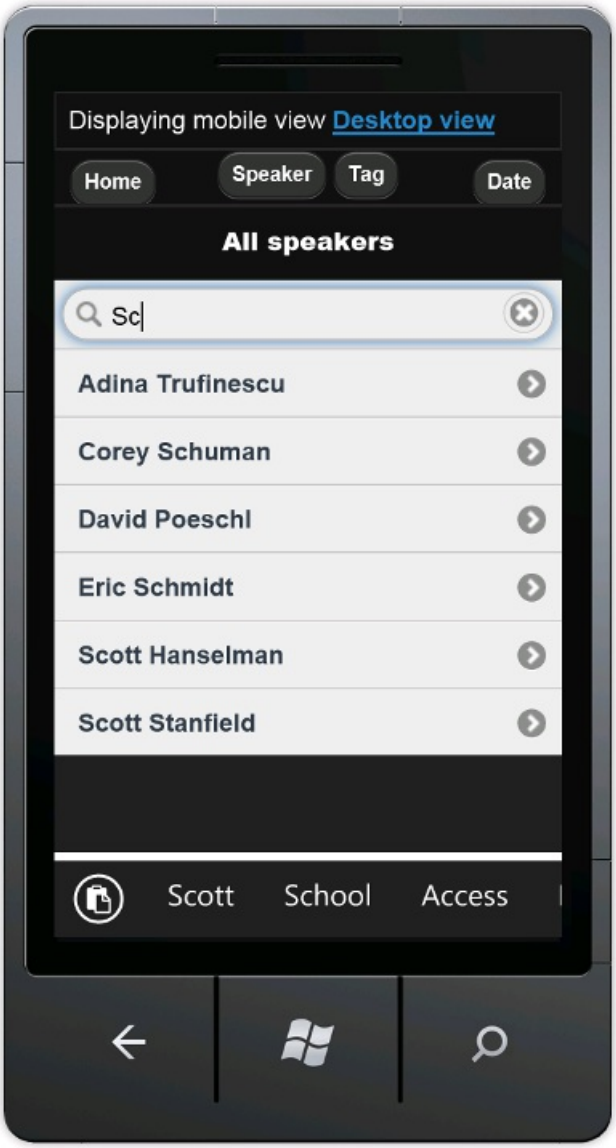
Although the mobile view has improved, it's difficult to navigate the long list of speakers. To fix this, in the `<ul>` tag, add the `data-filter` attribute and set it to `true`. The code below shows the `ul` markup.

```
<ul data-role="listview" data-filter="true">
```

The following image shows the search filter box at the top of the page that results from the `data-filter` attribute.



As you type each letter in the search box, jQuery Mobile filters the displayed list as shown in the image below.



Improving the Tags List

Like the default *Speakers* view, the *Tags* view is readable, but the links are small and difficult to tap on a mobile device. In this section, you'll fix the *Tags* view the same way you fixed the *Speakers* view.

Remove the "hide" suffix to the *Views\Home\AllTags.Mobile.cshtml.hide* file so the name is *Views\Home\AllTags.Mobile.cshtml*. Open the renamed file and remove the `<h2>` element.

Add the `data-role` and `data-filter` attributes to the `<ul>` tag, as shown here:

```
<ul data-role="listview" data-filter="true">
```

The image below shows the tags page filtering on the letter `J`.

Improving the Dates List

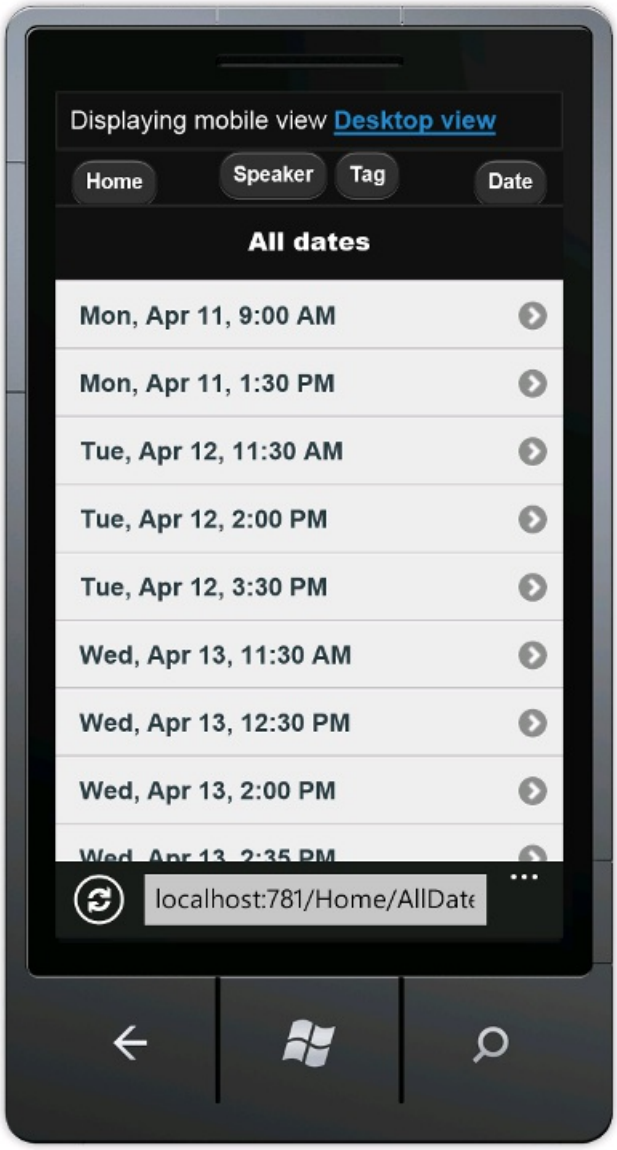
You can improve the *Dates* view like you improved the *Speakers* and *Tags* views, so that it's easier to use on a mobile device.

Copy the *Views\Home\AllDates.cshtml* file to *Views\Home\AllDates.Mobile.cshtml*. Open the new file and remove the `<h2>` element.

Add `data-role="listview"` to the `<ul>` tag, like this:

```
<ul data-role="listview">
```

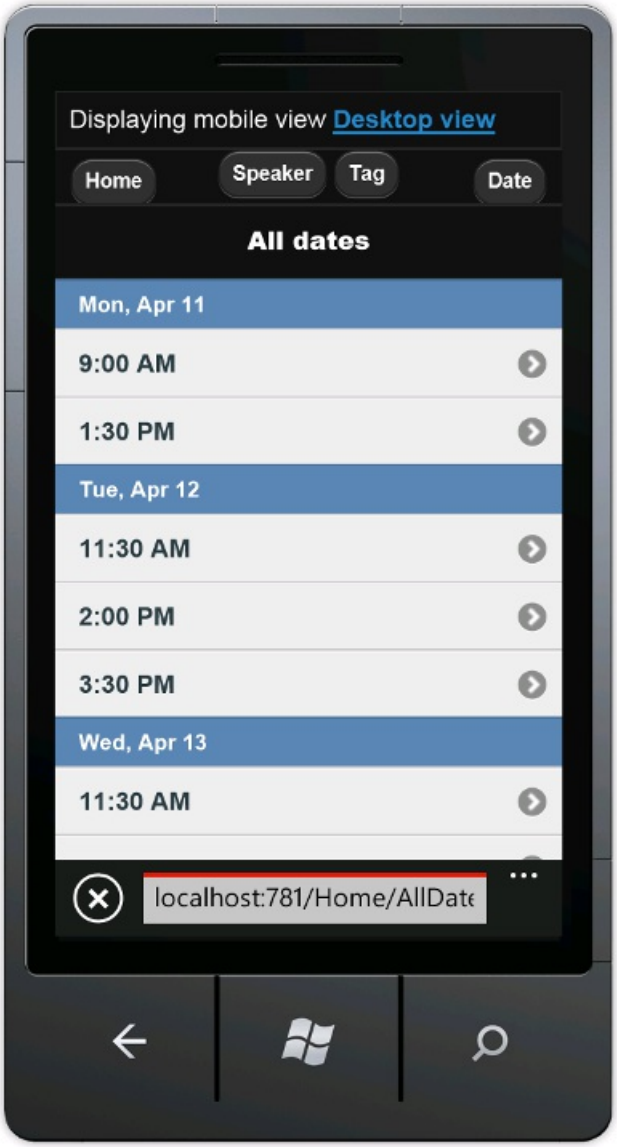
The image below shows what the **Date** page looks like with the `data-role` attribute in place.



Replace the contents of the *Views\Home\AllDates.Mobile.cshtml* file with the following code:

```
@model IEnumerable<DateTime>
@{
    ViewBag.Title = "All dates";
    DateTime lastDay = default(DateTime);
}
<ul data-role="listview">
    @foreach(var date in Model) {
        if (date.Date != lastDay) {
            lastDay = date.Date;
            <li data-role="list-divider">@date.Date.ToString("ddd, MMM dd")</li>
        }
        <li>@Html.ActionLink(date.ToString("h:mm tt"), "SessionsByDate", new { date })</li>
    }
</ul>
```

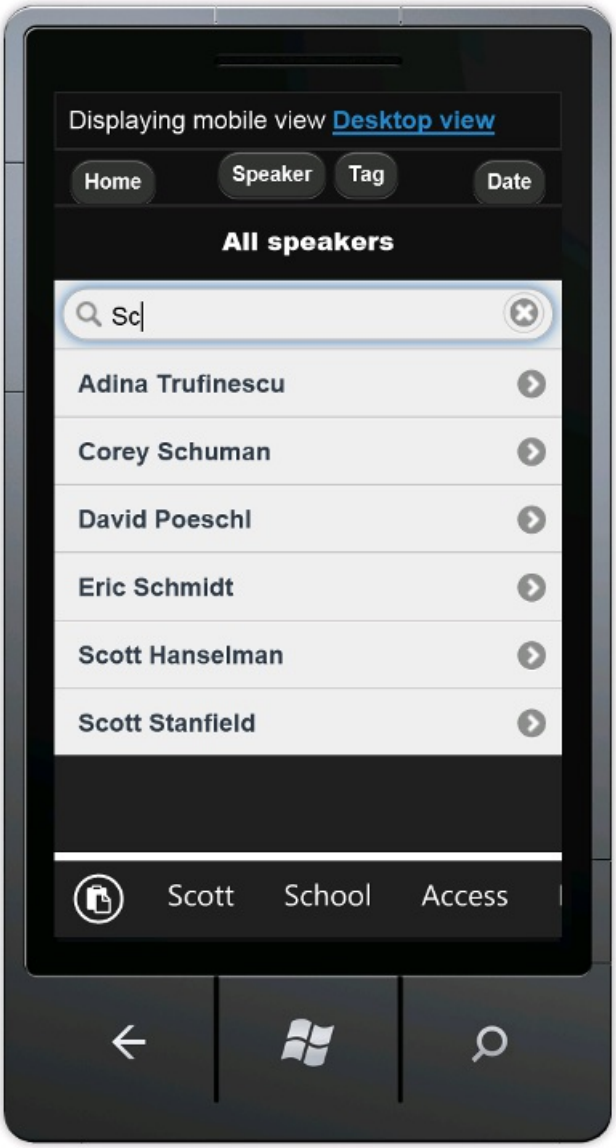
This code groups all sessions by days. It creates a list divider for each new day, and it lists all the sessions for each day under a divider. Here's what it looks like when this code runs:



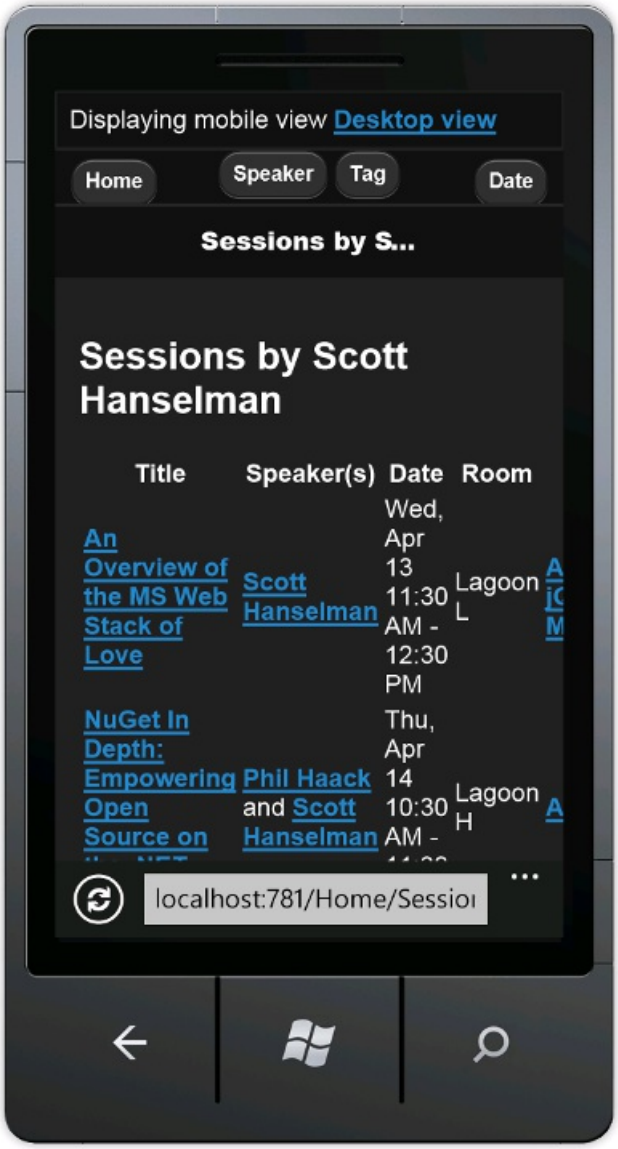
Improving the SessionsTable View

In this section, you'll create a mobile-specific view of sessions. The changes we make will be more extensive than in other views we have created.

In the mobile browser, tap the **Speaker** button, then enter `Sc` in the search box.



Tap the **Scott Hanselman** link.

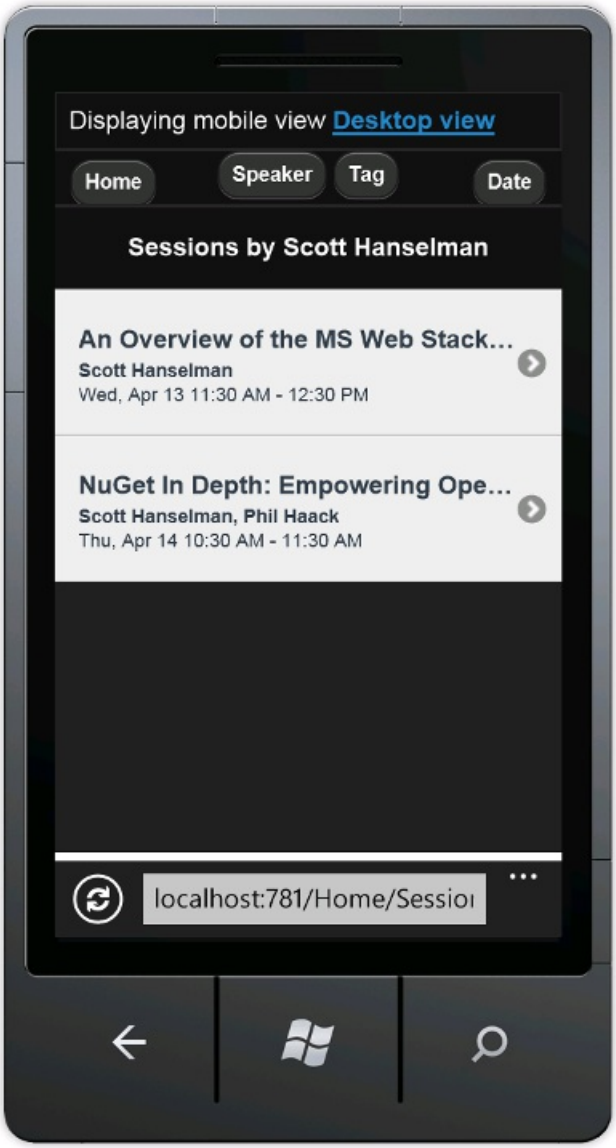


As you can see, the display is difficult to read on a mobile browser. The date column is hard to read and the tags column is out of the view. To fix this, copy *Views\Home\SessionsTable.cshtml* to *Views\Home\SessionsTable.Mobile.cshtml*, and then replace the contents of the file with the following code:

```
@using MvcMobile.Models
@model IEnumerable<Session>

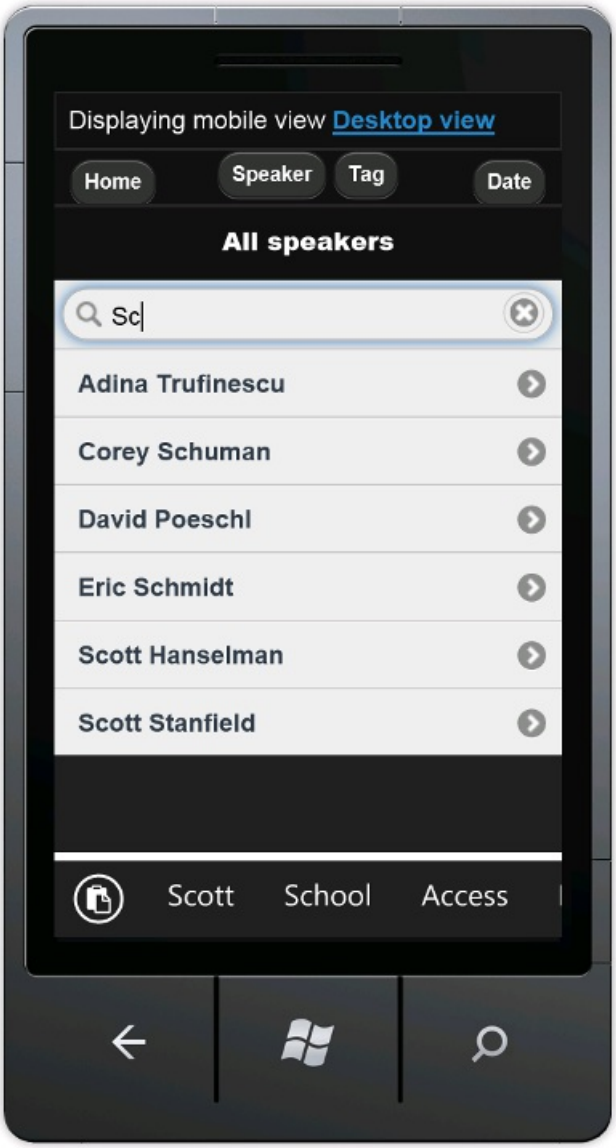
<ul data-role="listview">
    @foreach(var session in Model) {
        <li>
            <a href="@Url.Action("SessionByCode", new { session.Code })">
                <h3>@session.Title</h3>
                <p><strong>@string.Join(", ", session.Speakers)</strong></p>
                <p>@session.DateText</p>
            </a>
        </li>
    }
</ul>
```

The code removes the room and tags columns, and formats the title, speaker, and date vertically, so that all this information is readable on a mobile browser. The image below reflects the code changes.

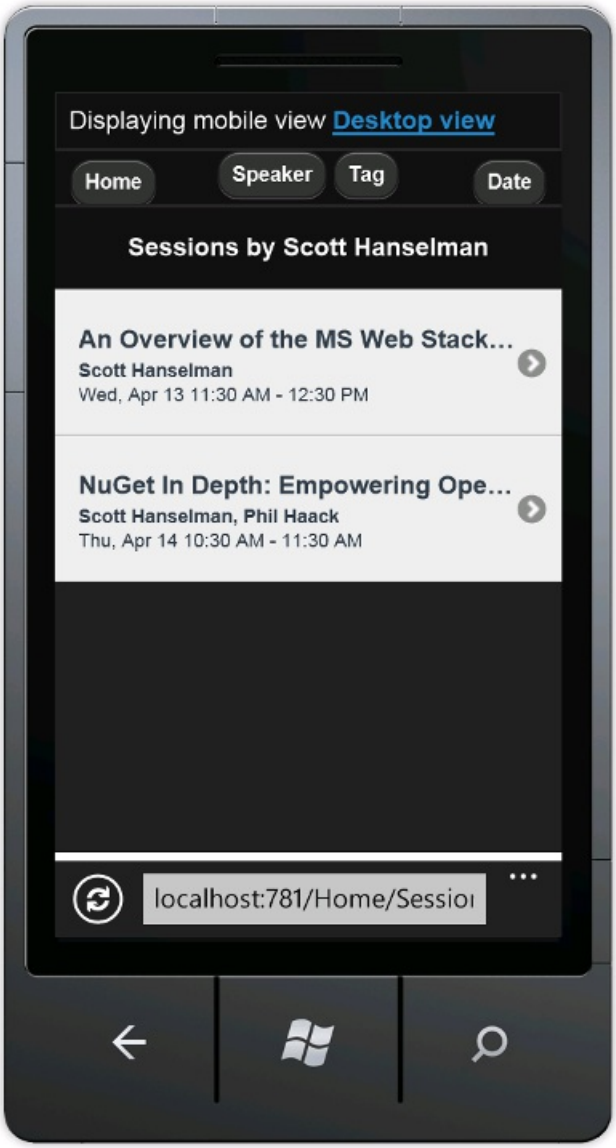


Improving the SessionByCode View

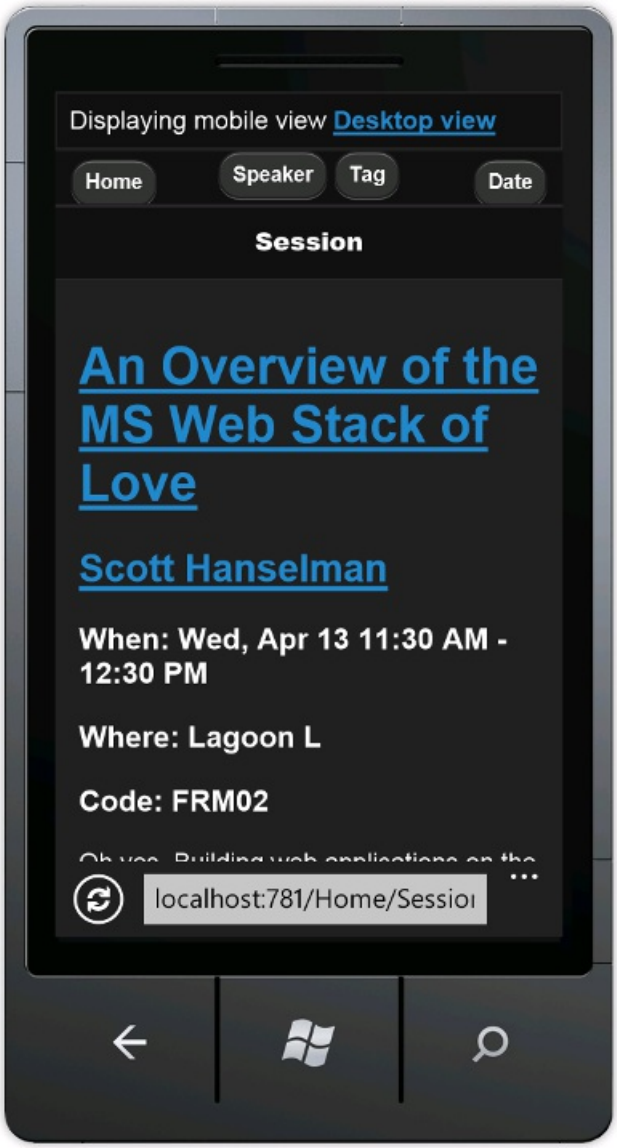
Finally, you'll create a mobile-specific view of the *SessionByCode* view. In the mobile browser, tap the **Speaker** button, then enter `Sc` in the search box.



Tap the **Scott Hanselman** link. Scott Hanselman's sessions are displayed.



Choose the **An Overview of the MS Web Stack of Love** link.



The default desktop view is fine, but you can improve it.

Copy the *Views\Home\SessionByCode.cshtml* to *Views\Home\SessionByCode.Mobile.cshtml* and replace the contents of the *Views\Home\SessionByCode.Mobile.cshtml* file with the following markup:

```
@model MvcMobile.Models.Session

@{
    ViewBag.Title = "Session details";
}
<h2>@Model.Title</h2>
<p>
    <strong>@Model.DateText</strong> in <strong>@Model.Room</strong>
</p>

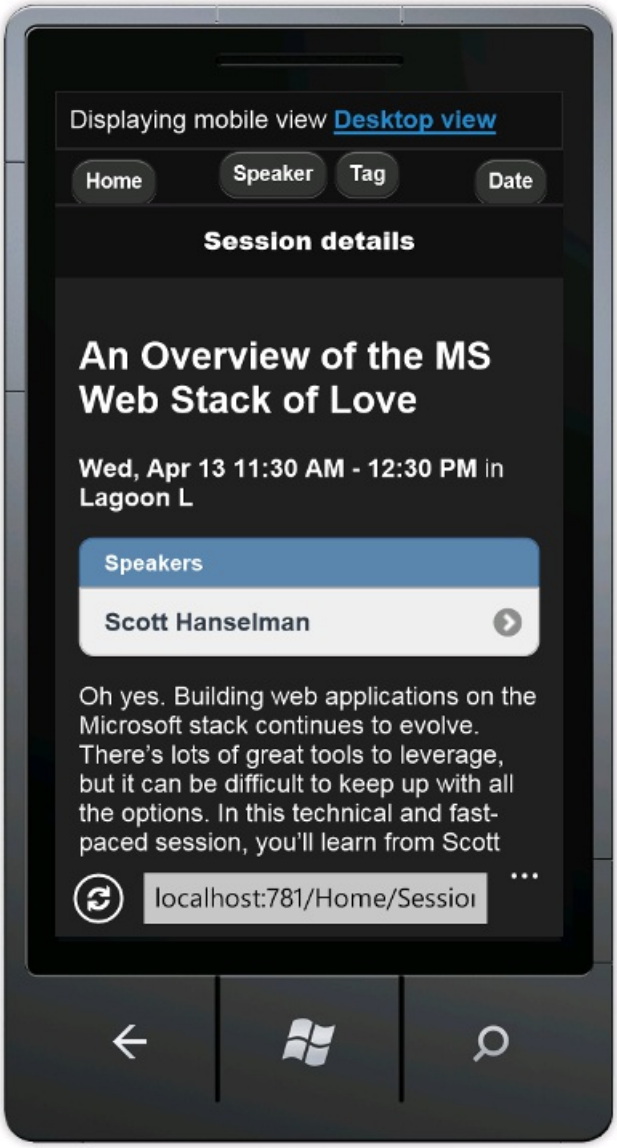
<ul data-role="listview" data-inset="true">
    <li data-role="list-divider">Speakers</li>
    @foreach (var speaker in Model.Speakers) {
        <li>@Html.ActionLink(speaker, "SessionsBySpeaker", new { speaker })</li>
    }
</ul>

<p>@Model.Description</p>
<h4>Code: @Model.Code</h4>

<ul data-role="listview" data-inset="true">
    <li data-role="list-divider">Tags</li>
    @foreach (var tag in Model.Tags) {
        <li>@Html.ActionLink(tag, "SessionsByTag", new { tag })</li>
    }
</ul>
```

The new markup uses the `data-role` attribute to improve the layout of the view.

Refresh the mobile browser. The following image reflects the code changes that you just made:



Wrapup and Review

This tutorial has introduced the new mobile features of ASP.NET MVC 4 Developer Preview. The mobile features include:

- The ability to override layout, views, and partial views, both globally and for an individual view.
- Control over layout and partial override enforcement using the `RequireConsistentDisplayMode` property.
- A view-switcher widget for mobile views than can also be displayed in desktop views.
- Support for supporting specific browsers, such as the iPhone browser.

See Also

- jQuery Mobile site.
- jQuery Mobile Overview
- W3C Recommendation Mobile Web Application Best Practices
- W3C Candidate Recommendation for media queries

Using OAuth Providers with MVC 4

2/20/2020 • 15 minutes to read • Edit Online

by Tom FitzMacken

> This tutorial shows you how to build an ASP.NET MVC 4 web application that enables users to log in with credentials from an external provider, such as Facebook, Twitter, Microsoft, or Google, and then integrate some of the functionality from those providers into your web application. For simplicity, this tutorial focuses on working with credentials from Facebook.
>
> To use external credentials in an ASP.NET MVC 5 web application, see Create an ASP.NET MVC 5 App with Facebook and Google OAuth2 and OpenID Sign-on.
>
> Enabling these credentials in your web sites provides a significant advantage because millions of users already have accounts with these external providers. These users may be more inclined to sign up for your site if they do not have to create and remember a new set of credentials. Also, after a user has logged in through one of these providers, you can incorporate social operations from the provider.

What you'll build

There are two main goals in this tutorial:

1. Enable a user to log in with credentials from an OAuth provider.
2. Retrieve account information from the provider and integrate that information with the account registration for your site.

Although the examples in this tutorial focus on using Facebook as the authentication provider, you can modify the code to use any of the providers. The steps to implement any provider are very similar to the steps you will see in this tutorial. You will only notice significant differences when making direct calls to the provider's API set.

Prerequisites

- Microsoft Visual Studio 2012 or Microsoft Visual Studio Express 2012 for Web

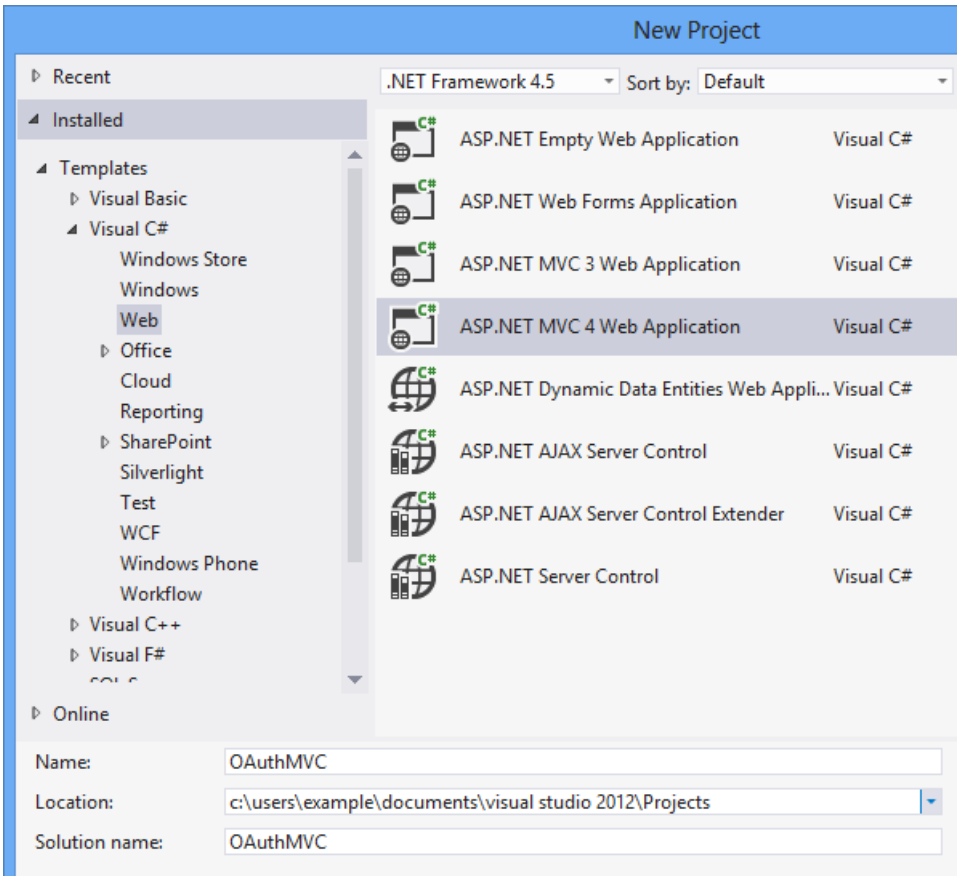
Or

- Microsoft Visual Studio 2010 SP1 or Visual Web Developer Express 2010 SP1
- ASP.NET MVC 4

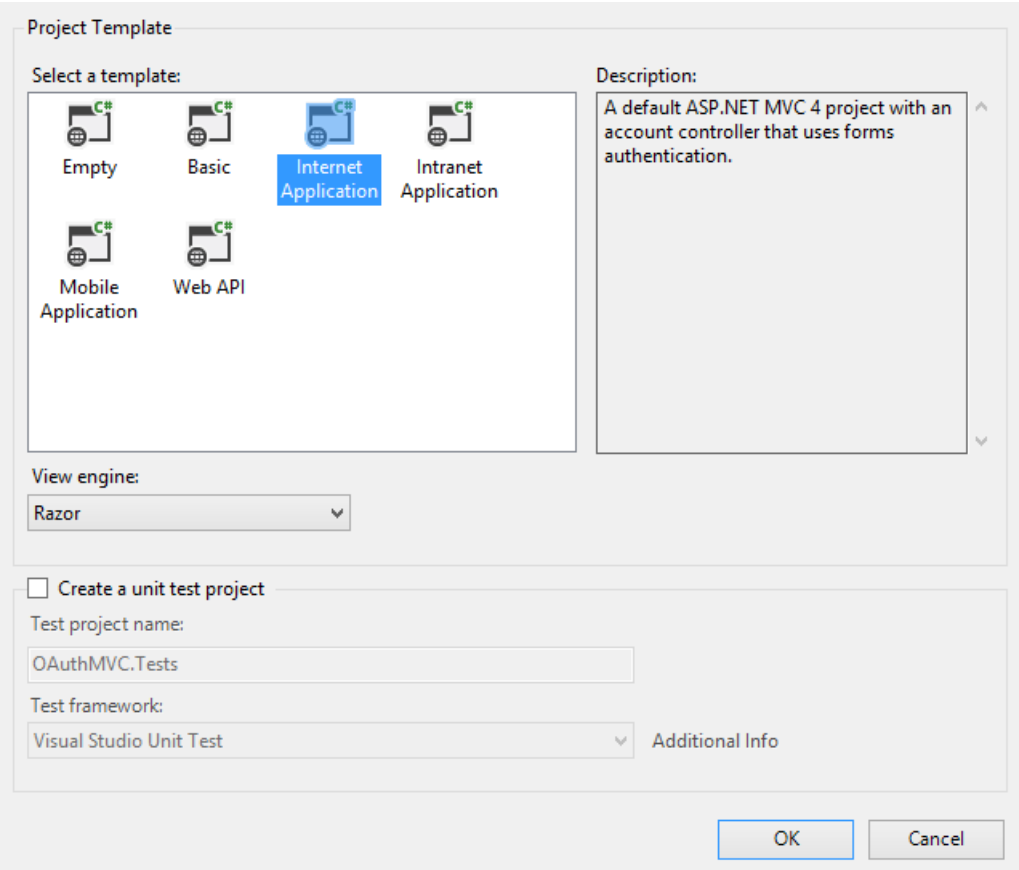
Furthermore, this topic assumes you have basic knowledge about ASP.NET MVC and Visual Studio. If you need an introduction to ASP.NET MVC 4, see Intro to ASP.NET MVC 4.

Create the project

In Visual Studio, create a new ASP.NET MVC 4 Web Application, and name it "OAuthMVC". You can target either .NET Framework 4.5 or 4.

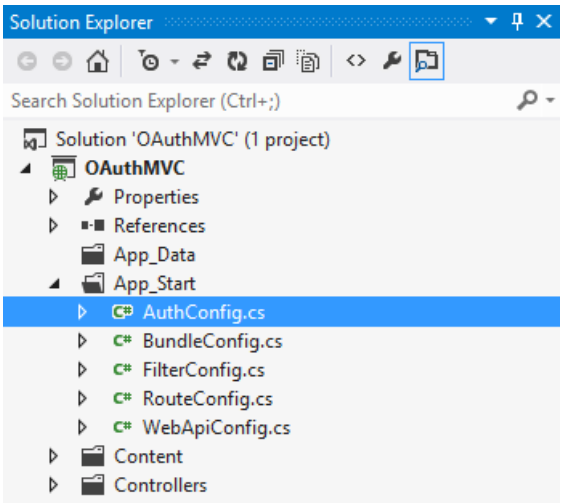


In the New ASP.NET MVC 4 Project window, select **Internet Application** and leave **Razor** as the view engine.



Enable a provider

When you create an MVC 4 web application with the Internet Application template, the project is created with a file named AuthConfig.cs in the App_Start folder.



The AuthConfig file contains code to register clients for external authentication providers. By default, this code is commented out, so none of the external providers are enabled.

```
public static class AuthConfig
{
    public static void RegisterAuth()
    {
        // To let users of this site log in using their accounts from other sites such as Microsoft, Facebook,
and Twitter,
        // you must update this site. For more information visit https://go.microsoft.com/fwlink/?
LinkID=252166

        //OAuthWebSecurity.RegisterMicrosoftClient(
        //    clientId: "",
        //    clientSecret: "");

        //OAuthWebSecurity.RegisterTwitterClient(
        //    consumerKey: "",
        //    consumerSecret: "");

        //OAuthWebSecurity.RegisterFacebookClient(
        //    appId: "",
        //    appSecret: "");

        //OAuthWebSecurity.RegisterGoogleClient();
    }
}
```

You must uncomment this code to use the external authentication client. You uncomment only the providers you want to include in your site. For this tutorial, you will only enable the Facebook credentials.

```
public static class AuthConfig
{
    public static void RegisterAuth()
    {
        //OAuthWebSecurity.RegisterMicrosoftClient(
        //    clientId: "",
        //    clientSecret: "");

        //OAuthWebSecurity.RegisterTwitterClient(
        //    consumerKey: "",
        //    consumerSecret: "");

        OAuthWebSecurity.RegisterFacebookClient(
            appId: "",
            appSecret: "");

        //OAuthWebSecurity.RegisterGoogleClient();
    }
}
```

Notice in the above example that the method includes empty strings for the registration parameters. If you try to run the application now, the application throws an argument exception because empty strings are not allowed for the parameters. To provide valid values, you must register your web site with the external providers, as shown in the next section.

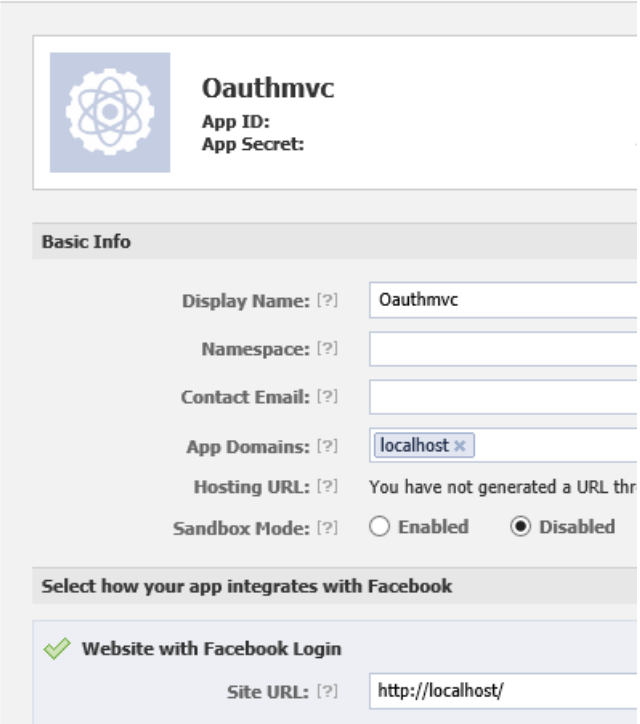
Registering with an external provider

To authenticate users with credentials from an external provider, you must register your web site with the provider. When you register your site, you will receive the parameters (such as key or id, and secret) to include when registering the client. You must have an account with the providers you wish to use.

This tutorial does not show all of the steps you must perform to register with these providers. The steps are typically not difficult. To successfully register your site, follow the instructions provided on those sites. To get started with registering your site, see the developer site for:

- Facebook
- Google
- Microsoft
- Twitter

When registering your site with Facebook, you can provide "localhost" for the site domain and `"http://localhost/"` for the URL, as shown in the image below. Using localhost works with most providers, but currently does not work with the Microsoft provider. For the Microsoft provider, you must include a valid web site URL.



In the previous image, the values for the app id, app secret, and contact email have been removed. When you actually register your site, those values will be present. You will want to note the values for app id and app secret because you will add them to your application.

Creating test users

If you do not mind using an existing Facebook account to test your site, you can skip this section.

You can easily create test users for your application within the Facebook app management page. You can use these test accounts to log in to your site. You create test users by clicking the **Roles** link in the left navigation pane and the clicking the **Create** link.

The Facebook site automatically creates the number of test accounts that you request.

Adding application id and secret from the provider

Now that you have received the id and secret from Facebook, go back to the AuthConfig file and add them as the parameter values. The values shown below are not real values.

```
public static class AuthConfig
{
    public static void RegisterAuth()
    {
        //OAuthWebSecurity.RegisterMicrosoftClient(
        //    clientId: "",
        //    clientSecret: "");

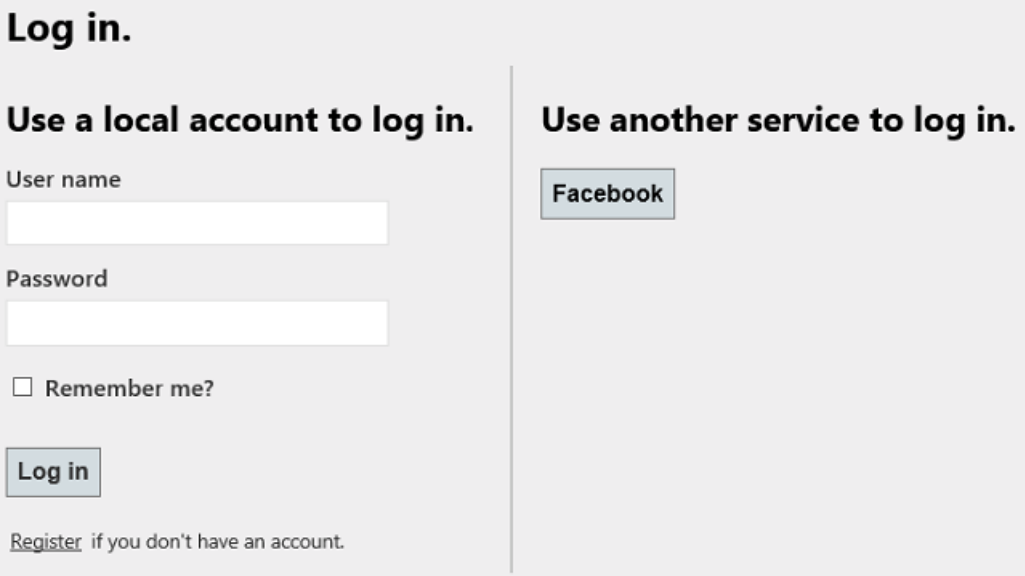
        //OAuthWebSecurity.RegisterTwitterClient(
        //    consumerKey: "",
        //    consumerSecret: "");

        //OAuthWebSecurity.RegisterFacebookClient(
        //    appId: "",
        //    appSecret: "");

        //OAuthWebSecurity.RegisterGoogleClient();
    }
}
```

Log in with external credentials

That is all you have to do to enable external credentials in your site. Run your application and click the login link in the upper right corner. The template automatically recognizes that you have registered Facebook as a provider and includes a button for the provider. If you register multiple providers, a button for each one is automatically included.

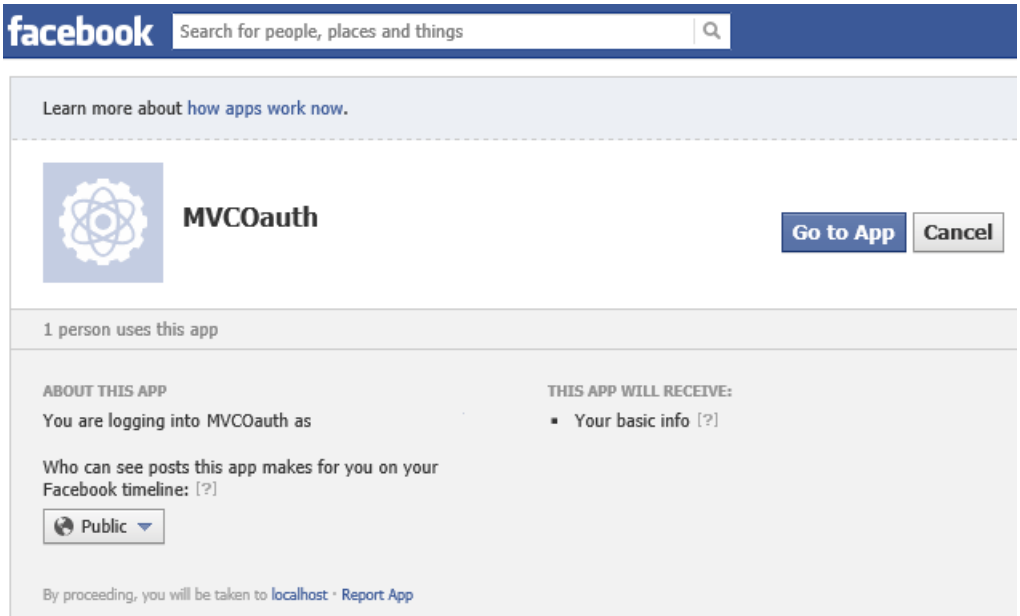


This tutorial does not cover how to customize the log in buttons for the external providers. For that information, see Customizing the login UI when using OAuth/OpenID.

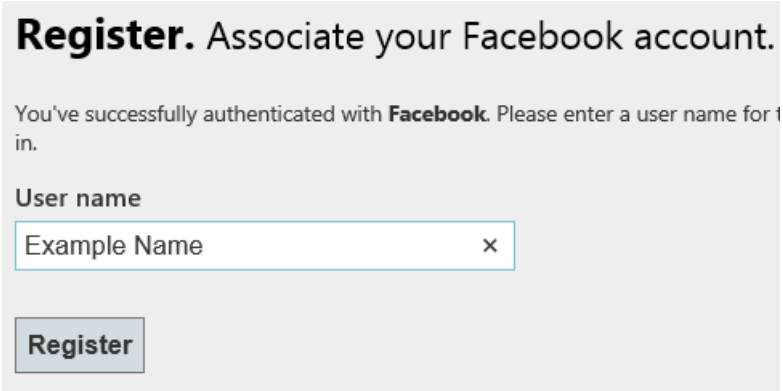
Click on the Facebook button to log in with Facebook credentials. When you select one of the external providers, you are redirected to that site and prompted by that service to log in.

The following image shows the login screen for Facebook. It notes that you are using your Facebook account to log in to a site named oauthmvcexample.

After logging in with Facebook credentials, a page informs the user that the site will have access to basic information.

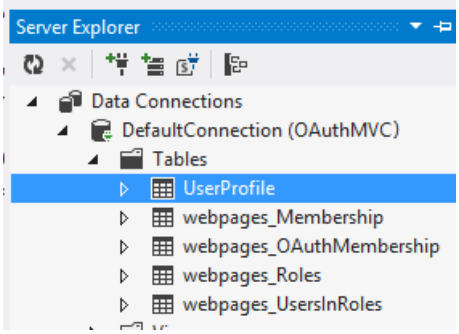


After selecting **Go to App**, the user must register for the site. The following image shows the registration page after a user has logged in with Facebook credentials. The user name is typically pre-filled with a name from the provider.

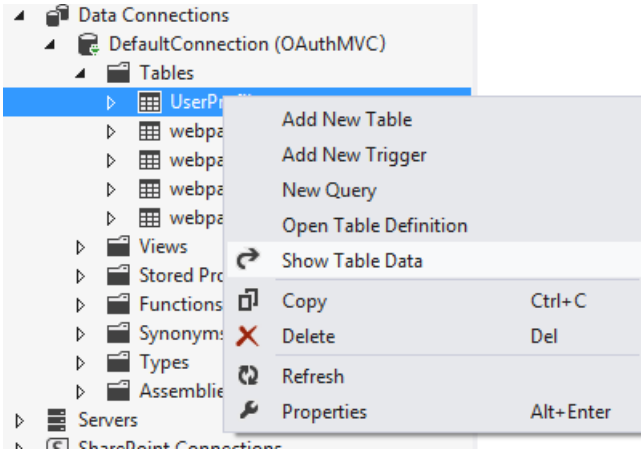


Click **Register** to complete registration. Close the browser.

You can see that the new account has been added to your database. In Server Explorer, open the **DefaultConnection** database and open the **Tables** folder.



Right-click the **UserProfile** table and select **Show Table Data**.



You will see the new account you added. Look at the data in **webpage_OAuthMembership** table. You will see more data related to the external provider for the account you just added.

If you only want to enable external authentication, you are done. However, you can further integrate information from the provider into the new user registration process, as shown in the following sections.

Create models for additional user information

As you noticed in the previous sections, you do not need to retrieve any additional information for the built-in account registration to work. However, most external providers pass back additional information about the user. The following sections show how to retain that information and save it to a database. Specifically, you will retain values for the user's full name, the URI of the user's personal web page, and a value that indicates whether Facebook has verified the account.

You will use Code First Migrations to add a table for storing additional user information. You are adding the table to an existing database, so you will first need to create a snapshot of the current database. By creating a snapshot of the current database, you can later create a migration that contains only the new table. To create a snapshot of the current database:

1. Open the **Package Manager Console**
2. Run the command **enable-migrations**
3. Run the command **add-migration initial –IgnoreChanges**
4. Run the command **update-database**

Now, you will add the new properties. In the Models folder, open the AccountModels.cs file, and find the RegisterExternalLoginModel class. The RegisterExternalLoginModel class holds values that come back from the authentication provider. Add properties named FullName and Link, as highlighted below.

```
public class RegisterExternalLoginModel
{
    [Required]
    [Display(Name = "User name")]
    public string UserName { get; set; }

    public string ExternalLoginData { get; set; }

    [Display(Name = "Full name")]
    public string FullName { get; set; }

    [Display(Name = "Personal page link")]
    public string Link { get; set; }
}
```

Also in AccountModels.cs, add a new class called ExtraUserInformation. This class represents the new table which will be created in the database.

```
[Table("ExtraUserInformation")]
public class ExternalUserInformation
{
    public int Id { get; set; }
    public int UserId { get; set; }
    public string FullName { get; set; }
    public string Link { get; set; }
    public bool? Verified { get; set; }
}
```

In the UsersContext class, add the highlighted code below to create a DbSet property for the new class.

```
public class UsersContext : DbContext
{
    public UsersContext()
        : base("DefaultConnection")
    {
    }

    public DbSet<UserProfile> UserProfiles { get; set; }
    public DbSet<ExternalUserInformation> ExternalUsers { get; set; }
}
```

You are now ready to create the new table. Open the Package Manager Console again and this time:

1. Run the command **add-migration AddExtraUserInformation**
2. Run the command **update-database**

The new table now exists in the database.

Retrieve the additional data

There are two ways to retrieve additional user data. The first way is to retain user data that is passed back, by default, during the authentication request. The second way is to specifically call the provider API and request more information. Values for FullName and Link are automatically passed back by Facebook. A value that indicates whether Facebook has verified the account is retrieved through a call to the Facebook API. First, you will populate values for FullName and Link, and then later, you will get the verified value.

To retrieve the additional user data, open the **AccountController.cs** file in the **Controllers** folder.

This file contains the logic for logging, registering, and managing accounts. In particular, notice the methods called

ExternalLoginCallback and **ExternalLoginConfirmation**. Within these methods, you can add code to customize external login operations for your application. The first line of the **ExternalLoginCallback** method contains:

```
AuthenticationResult result = OAuthWebSecurity.VerifyAuthentication(
    Url.Action("ExternalLoginCallback", new { ReturnUrl = returnUrl }));
```

Additional user data is passed back in the **ExtraData** property of the **AuthenticationResult** object that is returned from the **VerifyAuthentication** method. The Facebook client contains the following values in the **ExtraData** property:

- id
- name
- link
- gender
- accesstoken

Other providers will have similar but slightly different data in the ExtraData property.

If the user is new to your site, you will retrieve some the additional data and pass that data to the confirmation view. The last block of code in the method is run only if the user is new to your site. Replace the following line:

```
return View("ExternalLoginConfirmation", new RegisterExternalLoginModel
{
    UserName = result.UserName,
    ExternalLoginData = loginData
});
```

With this line:

```
return View("ExternalLoginConfirmation", new RegisterExternalLoginModel
{
    UserName = result.UserName,
    ExternalLoginData = loginData,
    FullName = result.ExtraData["name"],
    Link = result.ExtraData["link"]
});
```

This change merely includes values for the FullName and Link properties.

In the **ExternalLoginConfirmation** method, modify the code as highlighted below to save the additional user information.

```
if (user == null)
{
    // Insert name into the profile table
    UserProfile newUser = db.UserProfiles.Add(new UserProfile { UserName = model.UserName });
    db.SaveChanges();

    db.ExternalUsers.Add(new ExternalUserInformation
    {
        UserId = newUser.UserId,
        FullName = model.FullName,
        Link = model.Link
    });
    db.SaveChanges();

    OAuthWebSecurity.CreateOrUpdateAccount(provider, providerUserId, model.UserName);
    OAuthWebSecurity.Login(provider, providerUserId, createPersistentCookie: false);

    return RedirectToLocal(returnUrl);
}
else
{
    ModelState.AddModelError("UserName", "User name already exists. Please enter a different user name.");
}
```

Adjusting the view

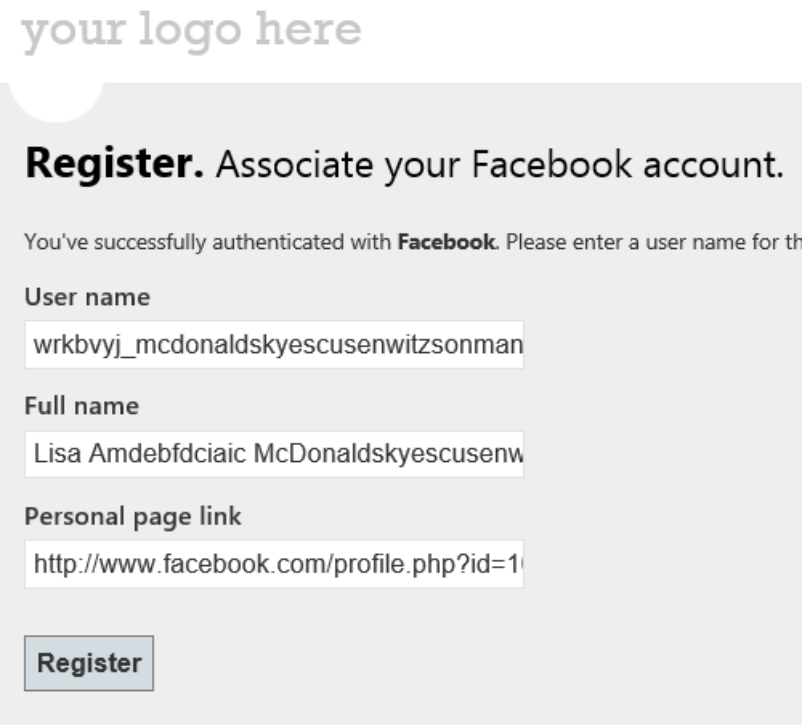
The additional user data that you retrieve from the provider will be displayed in the registration page.

In the **Views/Account** folder, open **ExternalLoginConfirmation.cshtml**. Below the existing field for user name, add fields for FullName, Link, and PictureLink.

```
<li>
    @Html.LabelFor(m => m.FullName)
    @Html.TextBoxFor(m => m.FullName)
</li>
<li>
    @Html.LabelFor(m => m.Link)
    @Html.TextBoxFor(m => m.Link)
</li>
```

You are now almost ready to run the application and register a new user with the additional information saved. You must have an account that is not already registered with the site. You can either use a different test account, or delete the rows in the **UserProfile** and **webpages_OAuthMembership** tables for the account you wish to reuse. By deleting those rows, you will ensure that the account is registered again.

Run the application and register the new user. Notice that this time the confirmation page contains more values.

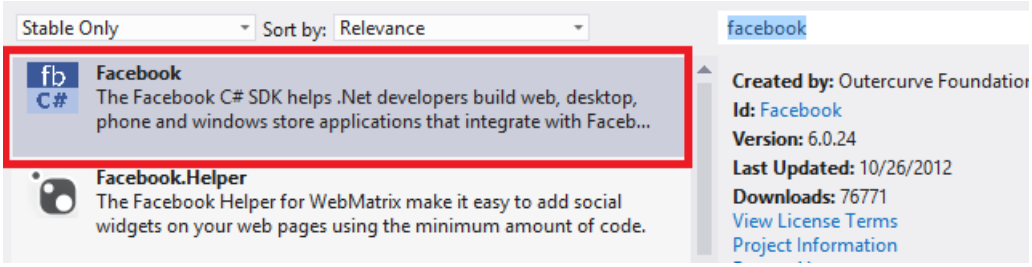


After completing registration, close the browser. Look in the database to notice the new values in the **ExtraUserInformation** table.

Install NuGet package for Facebook API

Facebook provides an API that you can call to perform operations. You can call the Facebook API either by directing sending HTTP requests, or by using installing a NuGet package that facilitates sending those requests. Using a NuGet package is shown in this tutorial, but installing NuGet package is not essential. This tutorial shows how to use the Facebook C# SDK package. There are other NuGet packages that assist with calling the Facebook API.

From the **Manage NuGet Packages** windows, select the Facebook C# SDK package.



You will use the Facebook C# SDK to call an operation that requires the access token for the user. The next section shows how to get the access token.

Retrieve access token

Most external providers pass back an access token after the user's credentials are verified. This access token is very important because it enables you to call operations that are only available to authenticated users. Therefore, retrieving and storing the access token is essential when you want to provide more functionality.

Depending on the external provider, the access token may be valid for only a limited amount of time. To ensure that you have a valid access token, you will retrieve it each time the user logs in, and store it as a session value rather than save it to a database.

In the **ExternalLoginCallback** method, the access token is also passed back in the **ExtraData** property of the **AuthenticationResult** object. Add the highlighted code to **ExternalLoginCallback** to save the access token in the **Session** object. This code is run every time the user logs in with a Facebook account.

```
[AllowAnonymous]
public ActionResult ExternalLoginCallback(string returnUrl)
{
    AuthenticationResult result = OAuthWebSecurity.VerifyAuthentication(
        Url.Action("ExternalLoginCallback", new { ReturnUrl = returnUrl }));
    if (!result.IsSuccessful)
    {
        return RedirectToAction("ExternalLoginFailure");
    }

    if (result.ExtraData.Keys.Contains("accesstoken"))
    {
        Session["facebooktoken"] = result.ExtraData["accesstoken"];
    }

    if (OAuthWebSecurity.Login(
        result.Provider,
        result.ProviderUserId,
        createPersistentCookie: false))
    {
        return RedirectToLocal(returnUrl);
    }

    if (User.Identity.IsAuthenticated)
    {
        // If the current user is logged in add the new account
        OAuthWebSecurity.CreateOrUpdateAccount(
            result.Provider,
            result.ProviderUserId,
            User.Identity.Name);
        return RedirectToLocal(returnUrl);
    }
    else
    {
        // User is new, ask for their desired membership name
        string loginData = OAuthWebSecurity.SerializeProviderUserId(
            result.Provider,
            result.ProviderUserId);
        ViewBag.ProviderDisplayName =
            OAuthWebSecurity.GetOAuthClientData(result.Provider).DisplayName;
        ViewBag.ReturnUrl = returnUrl;
        return View("ExternalLoginConfirmation", new RegisterExternalLoginModel
        {
            UserName = result.UserName,
            ExternalLoginData = loginData,
            FullName = result.ExtraData["name"],
            Link = result.ExtraData["link"]
        });
    }
}
```

Although this example retrieves an access token from Facebook, you can retrieve the access token from any external provider through the same key named "accesstoken".

Logging off

The default **LogOff** method logs the user out of your application, but does not log the user out of the external provider. To log the user out of Facebook, and prevent the token from persisting after the user has logged off, you can add the following highlighted code to the **LogOff** method in the AccountController.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult LogOff()
{
    WebSecurity.Logout();
    if (Session["facebooktoken"] != null)
    {
        var fb = new Facebook.FacebookClient();
        string accessToken = Session["facebooktoken"] as string;
        var logoutUrl = fb.GetLogoutUrl(new { access_token = accessToken, next = "http://localhost:39852/" });

        Session.RemoveAll();
        return Redirect(logoutUrl.AbsoluteUri);
    }

    return RedirectToAction("Index", "Home");
}
```

The value you provide in the `next` parameter is the URL to use after the user has logged out of Facebook. When testing on your local computer, you would provide the correct port number for your local site. In production, you would provide a default page, such as contoso.com.

Retrieve user information that requires the access token

Now that you have stored the access token and installed the Facebook C# SDK package, you can use them together to request additional user information from Facebook. In the **ExternalLoginConfirmation** method, create an instance of the **FacebookClient** class by passing the value of the access token. Request the value of the **verified** property for the current, authenticated user. The **verified** property indicates whether Facebook has validated the account through some other means, such as sending a message to a cell phone. Save this value in the database.

```
if (user == null)
{
    // Insert name into the profile table
    UserProfile newUser = db.UserProfiles.Add(new UserProfile { UserName = model.UserName });
    db.SaveChanges();

    bool facebookVerified;

    var client = new Facebook.FacebookClient(Session["facebooktoken"].ToString());
    dynamic response = client.Get("me", new { fields = "verified" });
    if (response.ContainsKey("verified"))
    {
        facebookVerified = response["verified"];
    }
    else
    {
        facebookVerified = false;
    }

    db.ExternalUsers.Add(new ExternalUserInformation
    {
        UserId = newUser.UserId,
        FullName = model.FullName,
        Link = model.Link,
        Verified = facebookVerified
    });
    db.SaveChanges();

    OAuthWebSecurity.CreateOrUpdateAccount(provider, providerUserId, model.UserName);
    OAuthWebSecurity.Login(provider, providerUserId, createPersistentCookie: false);

    return RedirectToLocal(returnUrl);
}
```

You will again need to either delete the records in the database for the user, or use a different Facebook account.

Run the application, and register the new user. Look at the **ExtraUserInformation** table to see the value for the Verified property.

Conclusion

In this tutorial, you created a site that is integrated with Facebook for user authentication and registration data. You learned about the default behavior that is set up for MVC 4 web application, and how to customize that default behavior.

Related topics

- Create an ASP.NET MVC app with auth and SQL DB and deploy to Azure App Service

Getting Started with ASP.NET MVC3

2/20/2020 • 2 minutes to read • Edit Online

The basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express.

- Getting Started with ASP.NET MVC 3 (C#)
  - Intro to ASP.NET MVC 3 (C#)
  - Adding a Controller (C#)
  - Adding a View (C#)
  - Adding a Model (C#)
  - Accessing your Model's Data from a Controller (C#)
  - Examining the Edit Methods and Edit View (C#)
  - Adding a New Field to the Movie Model and Table (C#)
  - Adding Validation to the Model (C#)
  - Improving the Details and Delete Methods (C#)
- Getting Started with ASP.NET MVC 3 (VB.NET)
  - Intro to ASP.NET MVC 3 (VB)
  - Adding a Controller (VB)
  - Adding a View (VB)
  - Adding a Model (VB)
  - Accessing your Model's Data from a Controller (VB)
  - Examining the Edit Methods and Edit View (VB)
  - Adding a New Field to the Movie Model and Database Table (VB)
  - Adding Validation to the Model (VB)
  - Improving the Details and Delete Methods (VB)

Getting Started with ASP.NET MVC 3 (C#)

2/20/2020 • 2 minutes to read • Edit Online

The basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express using C#.

- Intro to ASP.NET MVC 3 (C#)
- Adding a Controller (C#)
- Adding a View (C#)
- Adding a Model (C#)
- Accessing your Model's Data from a Controller (C#)
- Examining the Edit Methods and Edit View (C#)
- Adding a New Field to the Movie Model and Table (C#)
- Adding Validation to the Model (C#)
- Improving the Details and Delete Methods (C#)

Intro to ASP.NET MVC 3 (C#)

2/20/2020 • 3 minutes to read • Edit Online

by Rick Anderson

> **NOTE**
>
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

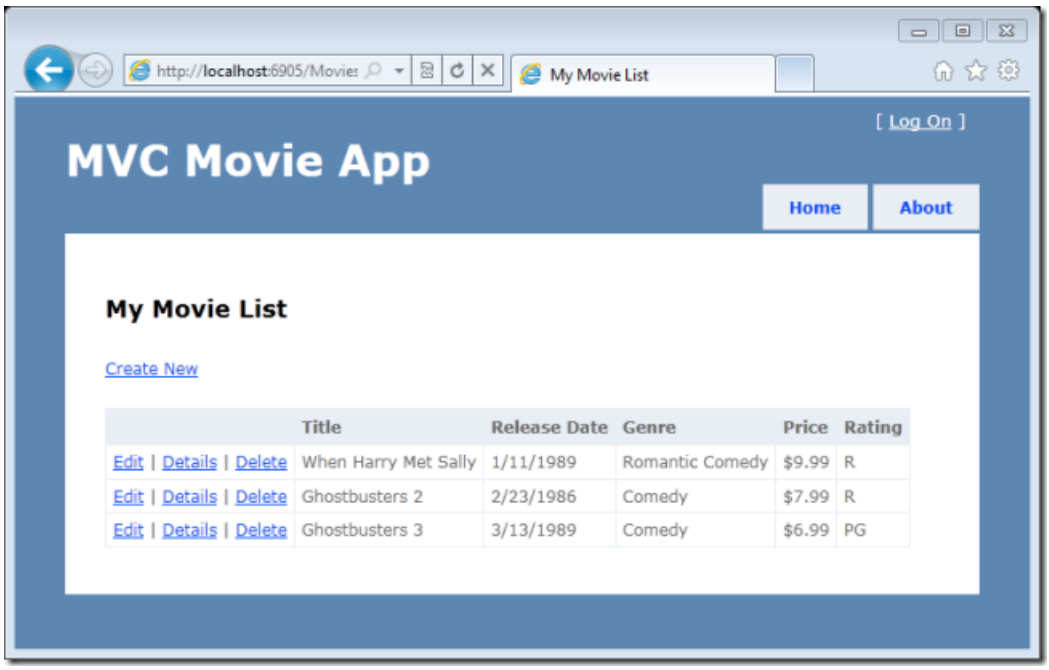
- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

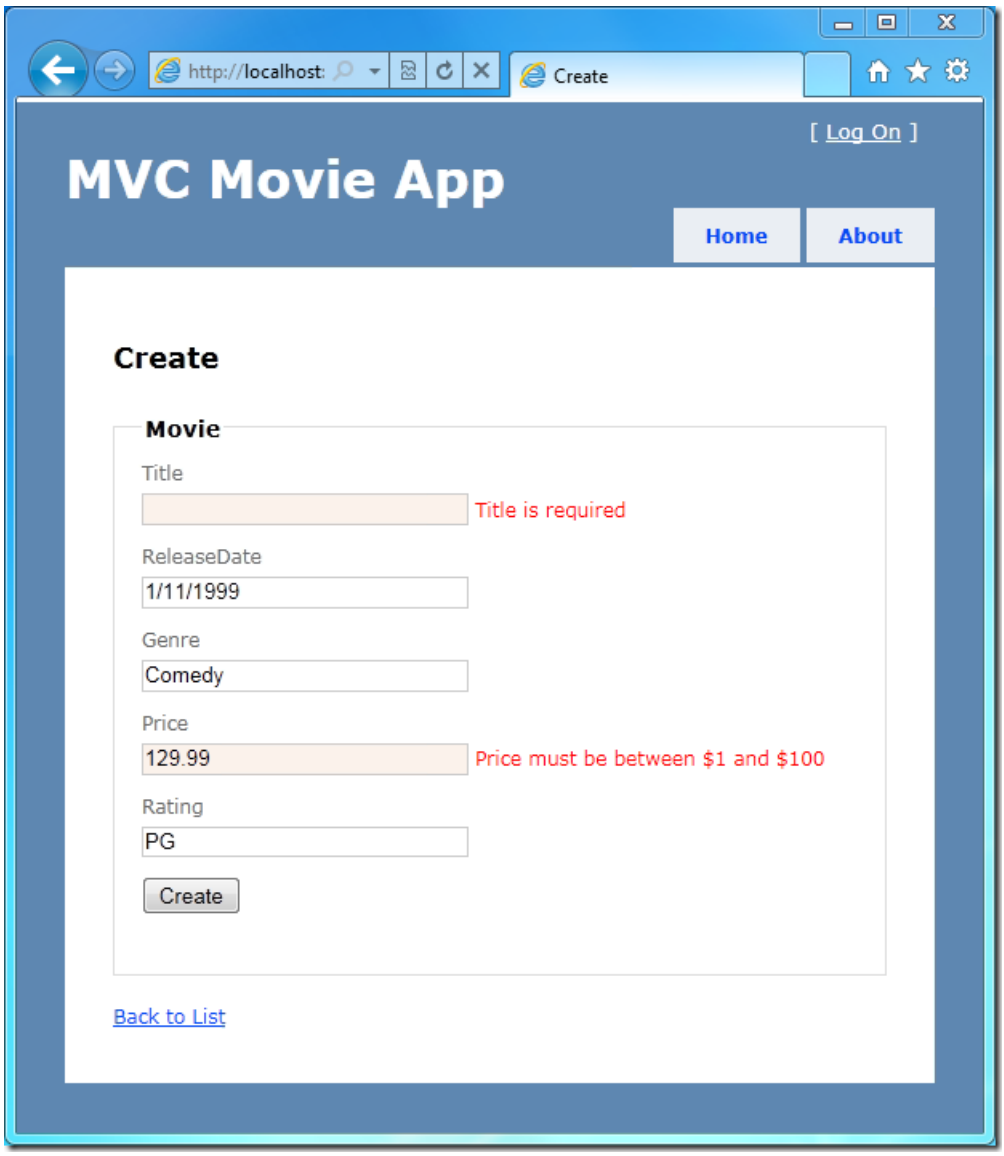
A Visual Web Developer project with C# source code is available to accompany this topic. Download the C# version. If you prefer Visual Basic, switch to the Visual Basic version of this tutorial.

What You'll Build

You'll implement a simple movie-listing application that supports creating, editing, and listing movies from a database. Below are two screenshots of the application you'll build. It includes a page that displays a list of movies from a database:



The application also lets you add, edit, and delete movies, as well as see details about individual ones. All data-entry scenarios include validation to ensure that the data stored in the database is correct.



Skills You'll Learn

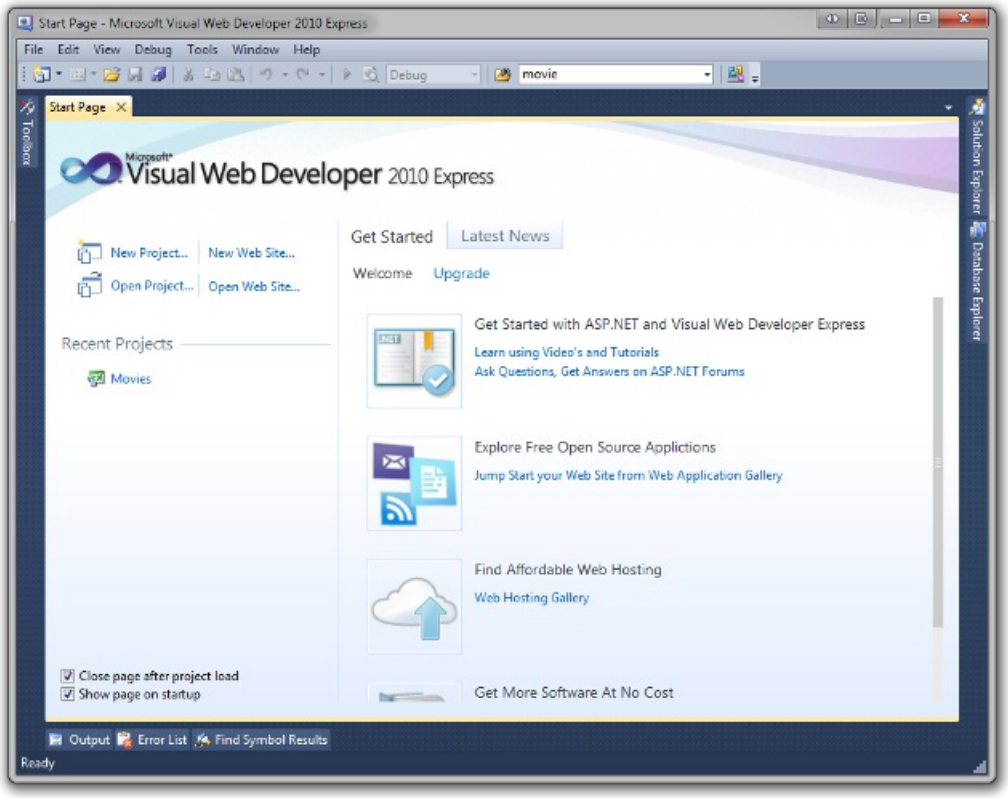
Here's what you'll learn:

- How to create a new ASP.NET MVC project.
- How to create ASP.NET MVC controllers and views.
- How to create a new database using the Entity Framework Code First paradigm.
- How to retrieve and display data.
- How to edit data and enable data validation.

Getting Started

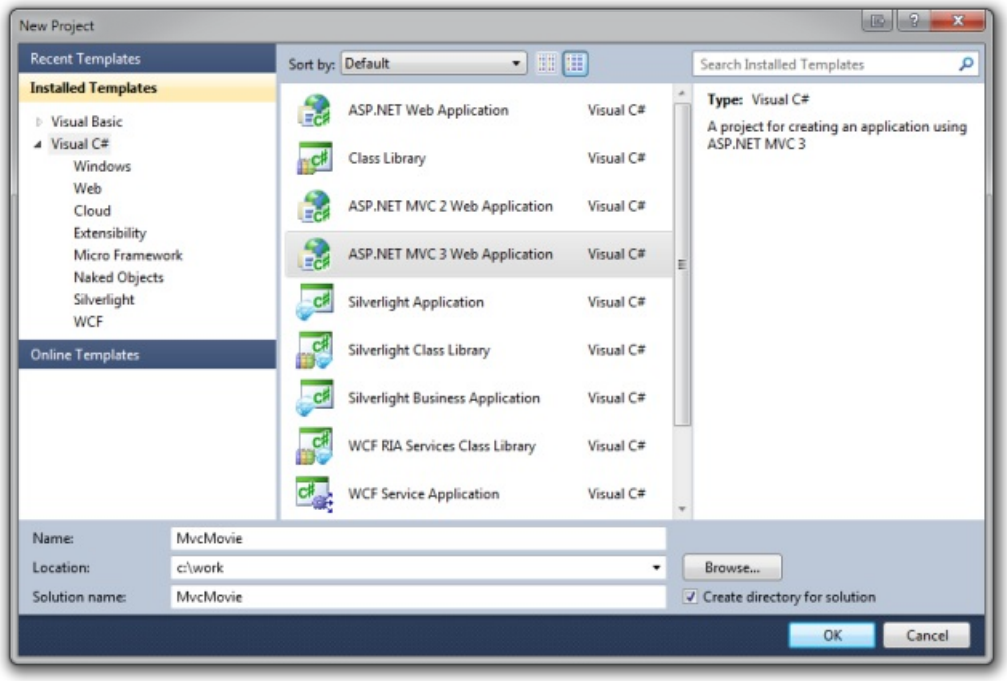
Start by running Visual Web Developer 2010 Express ("Visual Web Developer" for short) and select **New Project** from the **Start** page.

Visual Web Developer is an IDE, or integrated development environment. Just like you use Microsoft Word to write documents, you'll use an IDE to create applications. In Visual Web Developer there's a toolbar along the top showing various options available to you. There's also a menu that provides another way to perform tasks in the IDE. (For example, instead of selecting **New Project** from the **Start** page, you can use the menu and select **File** > **New Project**.)

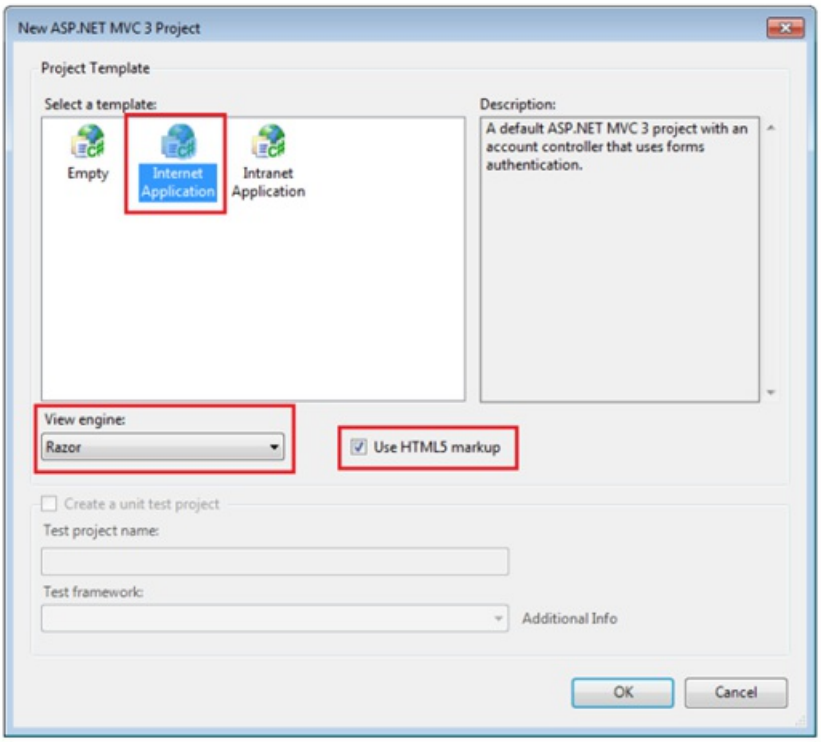


Creating Your First Application

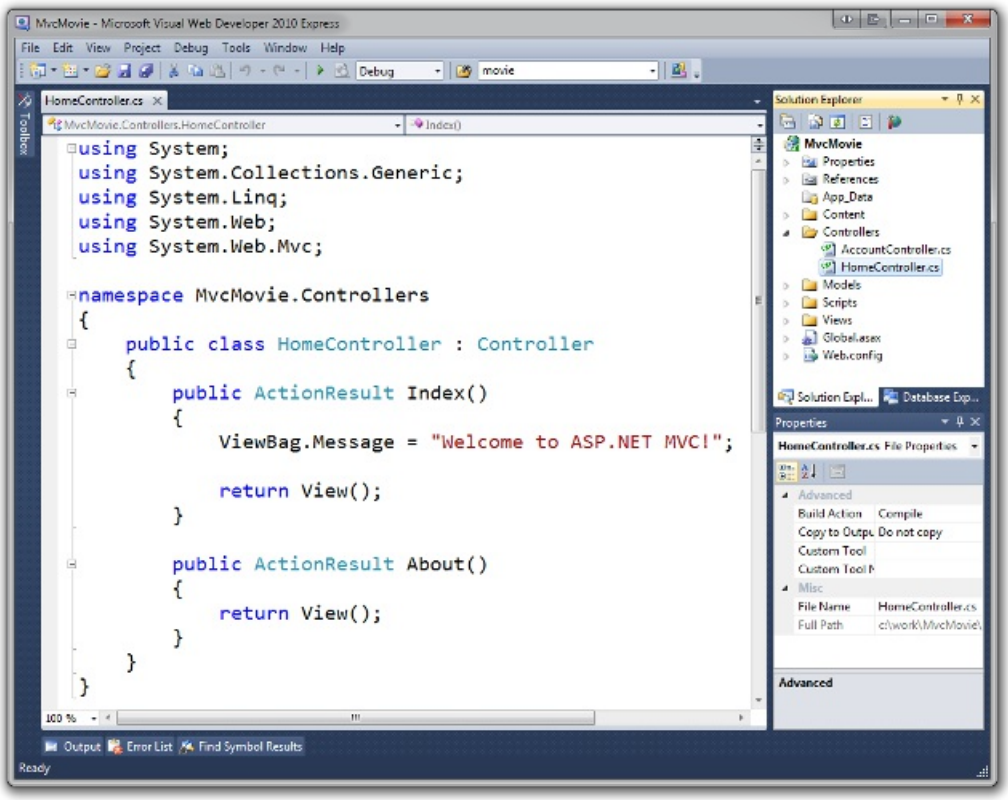
You can create applications using either Visual Basic or Visual C# as the programming language. Select Visual C# on the left and then select **ASP.NET MVC 3 Web Application**. Name your project "MvcMovie" and then click **OK**. (If you prefer Visual Basic, switch to the Visual Basic version of this tutorial.)



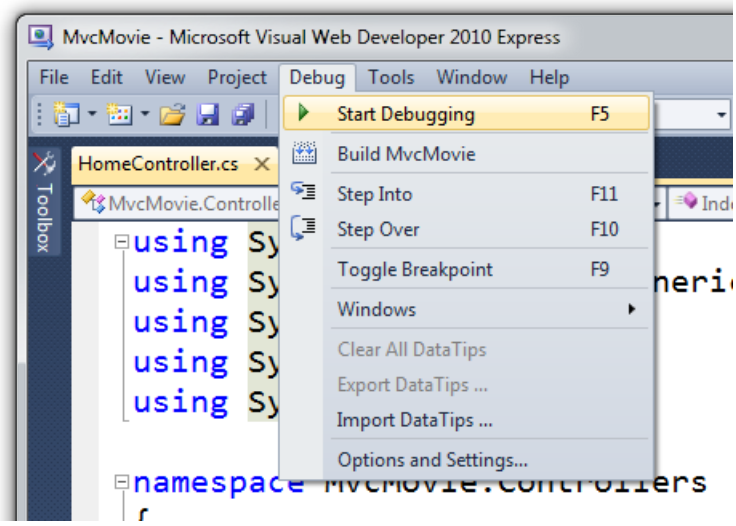
In the **New ASP.NET MVC 3 Project** dialog box, select **Internet Application**. Check **Use HTML5 markup** and leave **Razor** as the default view engine.



Click **OK**. Visual Web Developer used a default template for the ASP.NET MVC project you just created, so you have a working application right now without doing anything! This is a simple "Hello World!" project, and it's a good place to start your application.

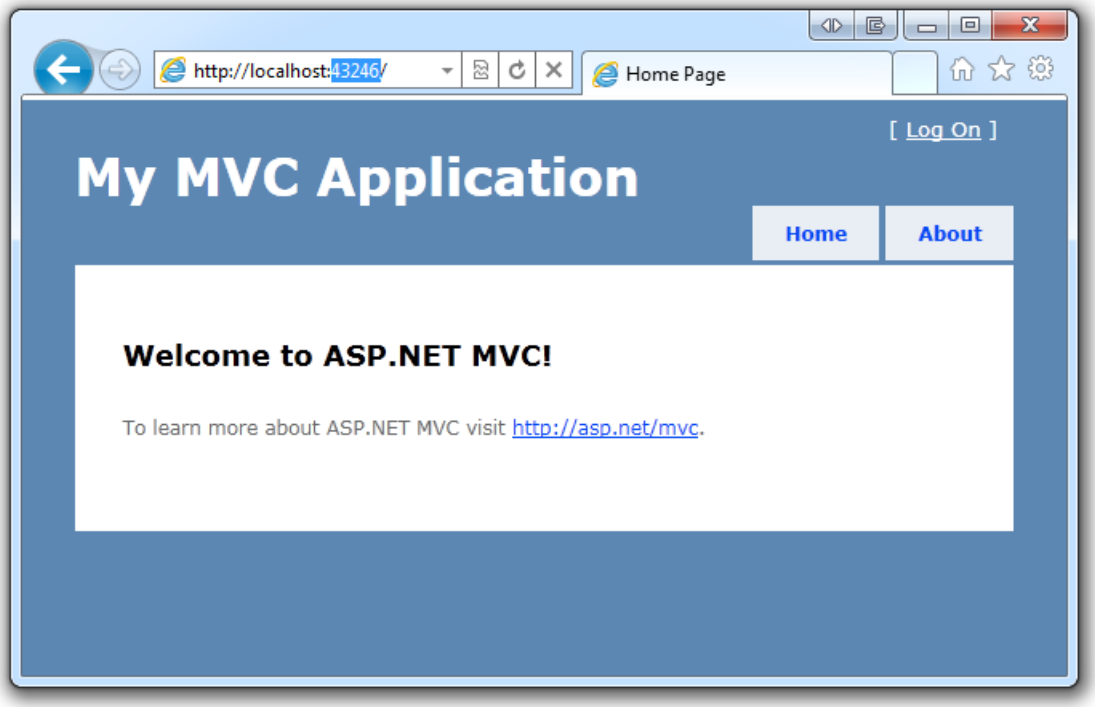


From the **Debug** menu, select **Start Debugging**.



Notice that the keyboard shortcut to start debugging is F5.

F5 causes Visual Web Developer to start a development web server and run your web application. Visual Web Developer then launches a browser and opens the application's home page. Notice that the address bar of the browser says `localhost` and not something like `example.com`. That's because `localhost` always points to your own local computer, which in this case is running the application you just built. When Visual Web Developer runs a web project, a random port is used for the web server. In the image below, the random port number is 43246. When you run the application, you'll probably see a different port number.



Right out of the box this default template gives you two pages to visit and a basic login page. The next step is to change how this application works and learn a little bit about ASP.NET MVC in the process. Close your browser and let's change some code.

NEXT

Adding a Controller (C#)

2/20/2020 • 4 minutes to read • Edit Online

by Rick Anderson

> **NOTE**
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

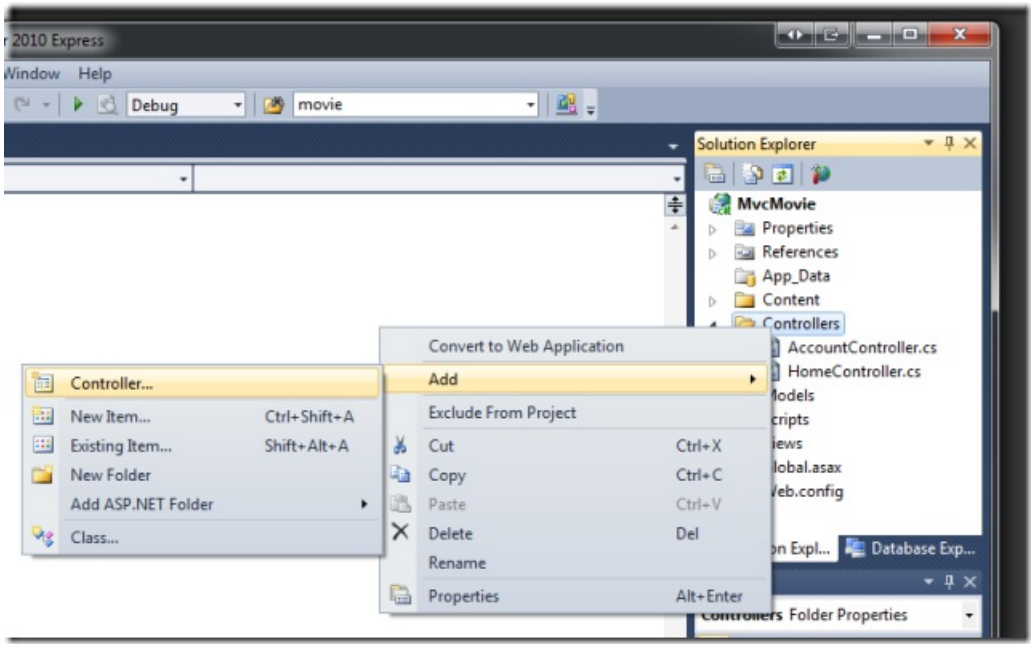
A Visual Web Developer project with C# source code is available to accompany this topic. Download the C# version. If you prefer Visual Basic, switch to the Visual Basic version of this tutorial.

MVC stands for *model-view-controller*. MVC is a pattern for developing applications that are well architected and easy to maintain. MVC-based applications contain:

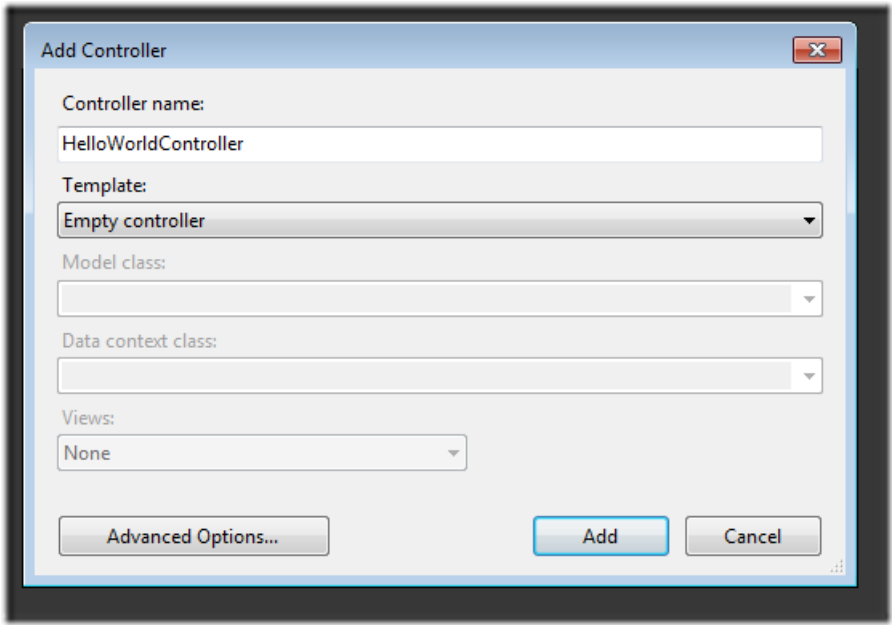
- Controllers: Classes that handle incoming requests to the application, retrieve model data, and then specify view templates that return a response to the client.
- Models: Classes that represent the data of the application and that use validation logic to enforce business rules for that data.
- Views: Template files that your application uses to dynamically generate HTML responses.

We'll be covering all these concepts in this tutorial series and show you how to use them to build an application.

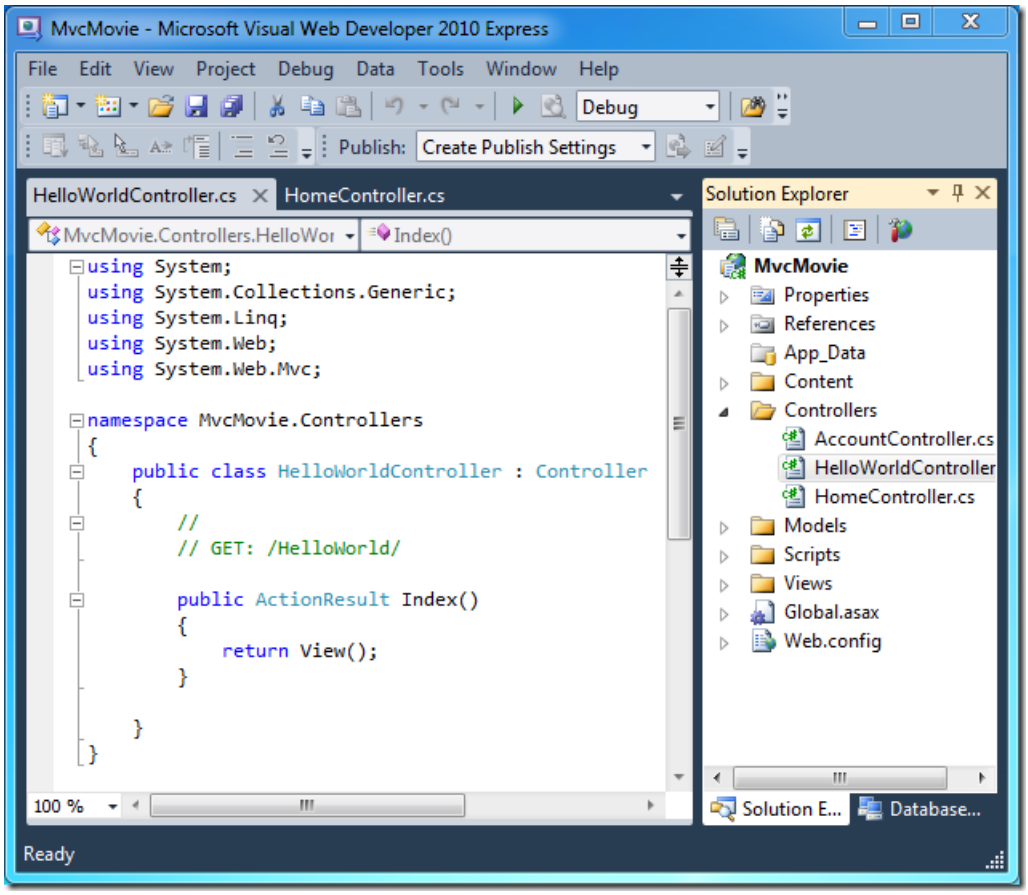
Let's begin by creating a controller class. In **Solution Explorer**, right-click the *Controllers* folder and then select **Add Controller**.



Name your new controller "HelloWorldController". Leave the default template as **Empty controller** and click **Add**.



Notice in **Solution Explorer** that a new file has been created named *HelloWorldController.cs*. The file is open in the IDE.



Inside the `public class HelloWorldController` block, create two methods that look like the following code. The controller will return a string of HTML as an example.

```
using System.Web;
using System.Web.Mvc;

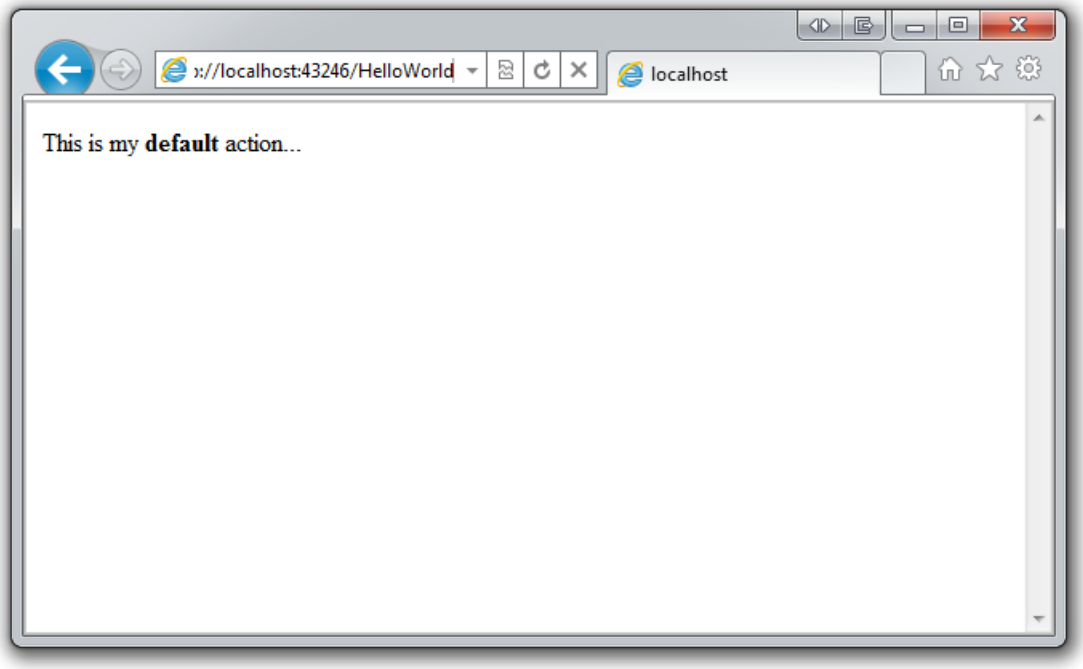
namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my <b>default</b> action...";
        }

        //
        // GET: /HelloWorld/Welcome/

        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

Your controller is named `HelloWorldController` and the first method above is named `Index`. Let's invoke it from a browser. Run the application (press F5 or Ctrl+F5). In the browser, append "HelloWorld" to the path in the address bar. (For example, in the illustration below, it's `http://localhost:43246/HelloWorld.`) The page in the browser will look like the following screenshot. In the method above, the code returned a string directly. You told the system to just return some HTML, and it did!

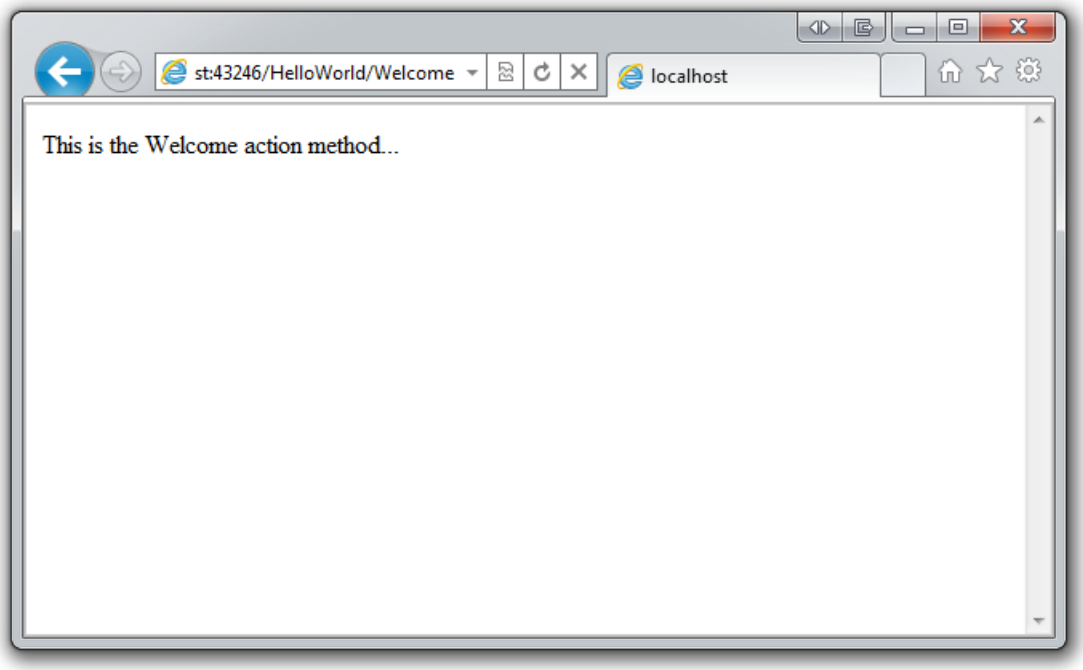


ASP.NET MVC invokes different controller classes (and different action methods within them) depending on the incoming URL. The default mapping logic used by ASP.NET MVC uses a format like this to determine what code to invoke:

```
/[Controller]/[ActionName]/[Parameters]
```

The first part of the URL determines the controller class to execute. So */HelloWorld* maps to the `HelloWorldController` class. The second part of the URL determines the action method on the class to execute. So */HelloWorld/Index* would cause the `Index` method of the `HelloWorldController` class to execute. Notice that we only had to browse to */HelloWorld* and the `Index` method was used by default. This is because a method named `Index` is the default method that will be called on a controller if one is not explicitly specified.

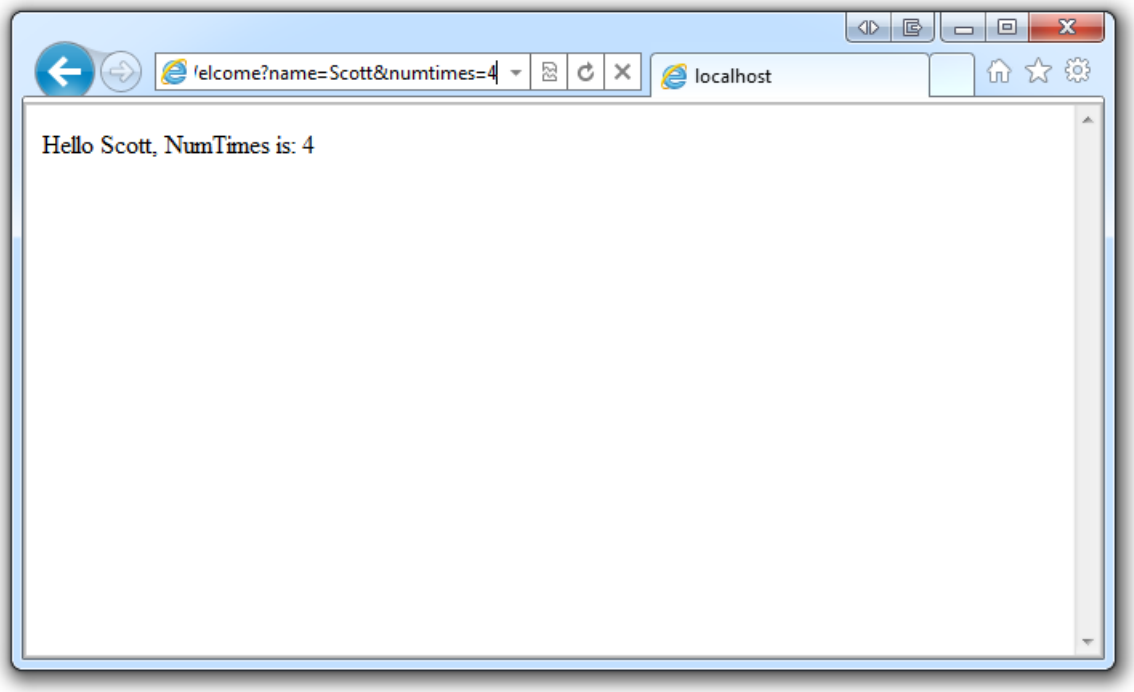
Browse to `http://localhost:xxxx/HelloWorld/Welcome`. The `Welcome` method runs and returns the string "This is the Welcome action method...". The default MVC mapping is `/[Controller]/[ActionName]/[Parameters]`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't used the `[Parameters]` part of the URL yet.



Let's modify the example slightly so that you can pass some parameter information from the URL to the controller (for example, */HelloWorld/Welcome?name=Scott&numtimes=4*). Change your `Welcome` method to include two parameters as shown below. Note that the code uses the C# optional-parameter feature to indicate that the `numTimes` parameter should default to 1 if no value is passed for that parameter.

```
public string Welcome(string name, int numTimes = 1) {
     return HttpUtility.HtmlEncode("Hello " + name + ", NumTimes is: " + numTimes);
}
```

Run your application and browse to the example URL (`http://localhost:xxxx/HelloWorld/Welcome?name=Scott&numtimes=4)` . You can try different values for `name` and `numtimes` in the URL. The system automatically maps the named parameters from the query string in the address bar to parameters in your method.



In both these examples the controller has been doing the "VC" portion of MVC — that is, the view and controller work. The controller is returning HTML directly. Ordinarily you don't want controllers returning HTML directly, since that becomes very cumbersome to code. Instead we'll typically use a separate view template file to help generate the HTML response. Let's look next at how we can do this.

PREVIOUS NEXT

Adding a View (C#)

2/20/2020 • 8 minutes to read • Edit Online

by Rick Anderson

> **NOTE**
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

A Visual Web Developer project with C# source code is available to accompany this topic. Download the C# version. If you prefer Visual Basic, switch to the Visual Basic version of this tutorial.

In this section you're going to modify the `HelloWorldController` class to use view template files to cleanly encapsulate the process of generating HTML responses to a client.

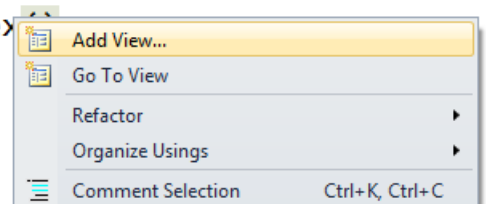
You'll create a view template file using the new Razor view engine introduced with ASP.NET MVC 3. Razor-based view templates have a *.cshtml* file extension, and provide an elegant way to create HTML output using C#. Razor minimizes the number of characters and keystrokes required when writing a view template, and enables a fast, fluid coding workflow.

Start by using a view template with the `Index` method in the `HelloWorldController` class. Currently the `Index` method returns a string with a message that is hard-coded in the controller class. Change the `Index` method to return a `View` object, as shown in the following:

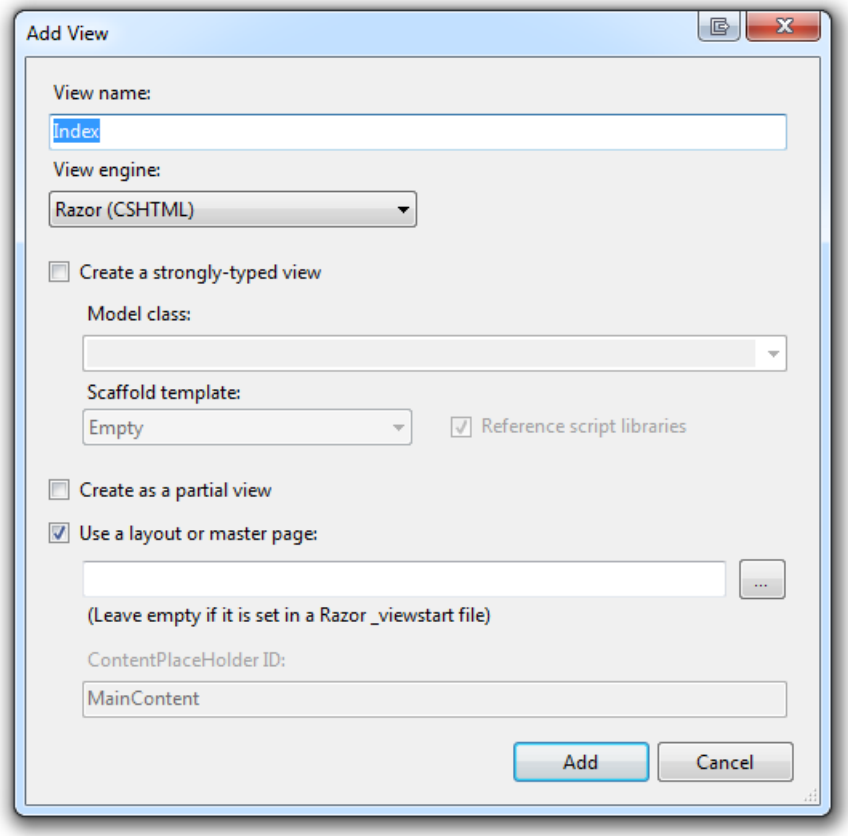
```
public ActionResult Index()
{
    return View();
}
```

This code uses a view template to generate an HTML response to the browser. In the project, add a view template that you can use with the `Index` method. To do this, right-click inside the `Index` method and click **Add View**.

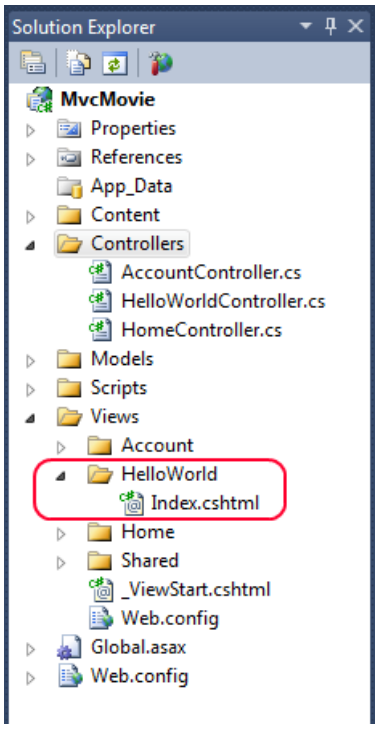
```
public class HelloWorldController : Controller
{
    public ActionResult Index
    {
        return View();
    }
```



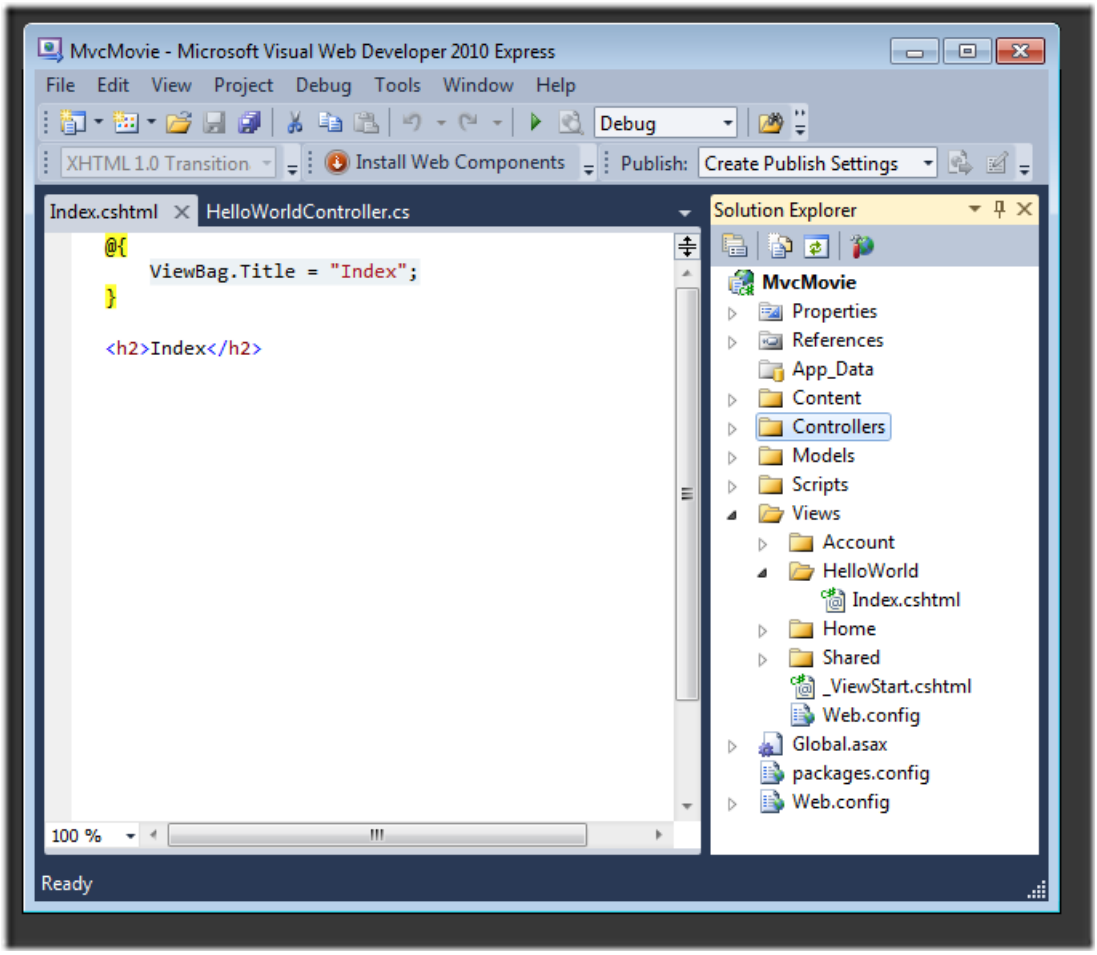
The **Add View** dialog box appears. Leave the defaults the way they are and click the **Add** button:



The *MvcMovie\Views\HelloWorld* folder and the *MvcMovie\Views\HelloWorld\Index.cshtml* file are created. You can see them in **Solution Explorer**:



The following shows the *Index.cshtml* file that was created:



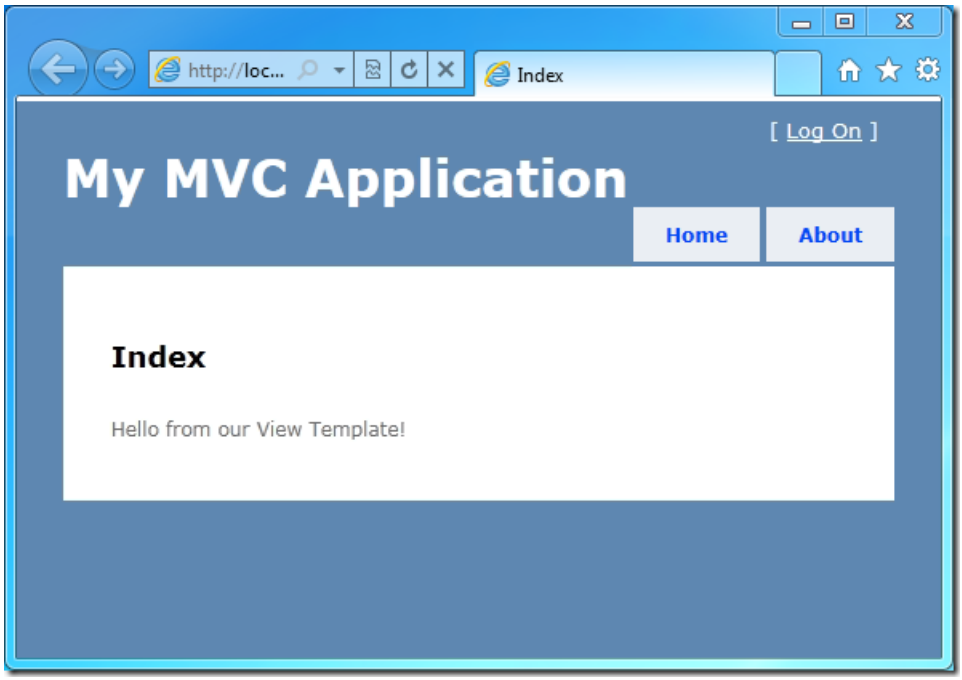
Add some HTML under the `<h2>` tag. The modified *MvcMovie\Views\HelloWorld\Index.cshtml* file is shown below.

```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

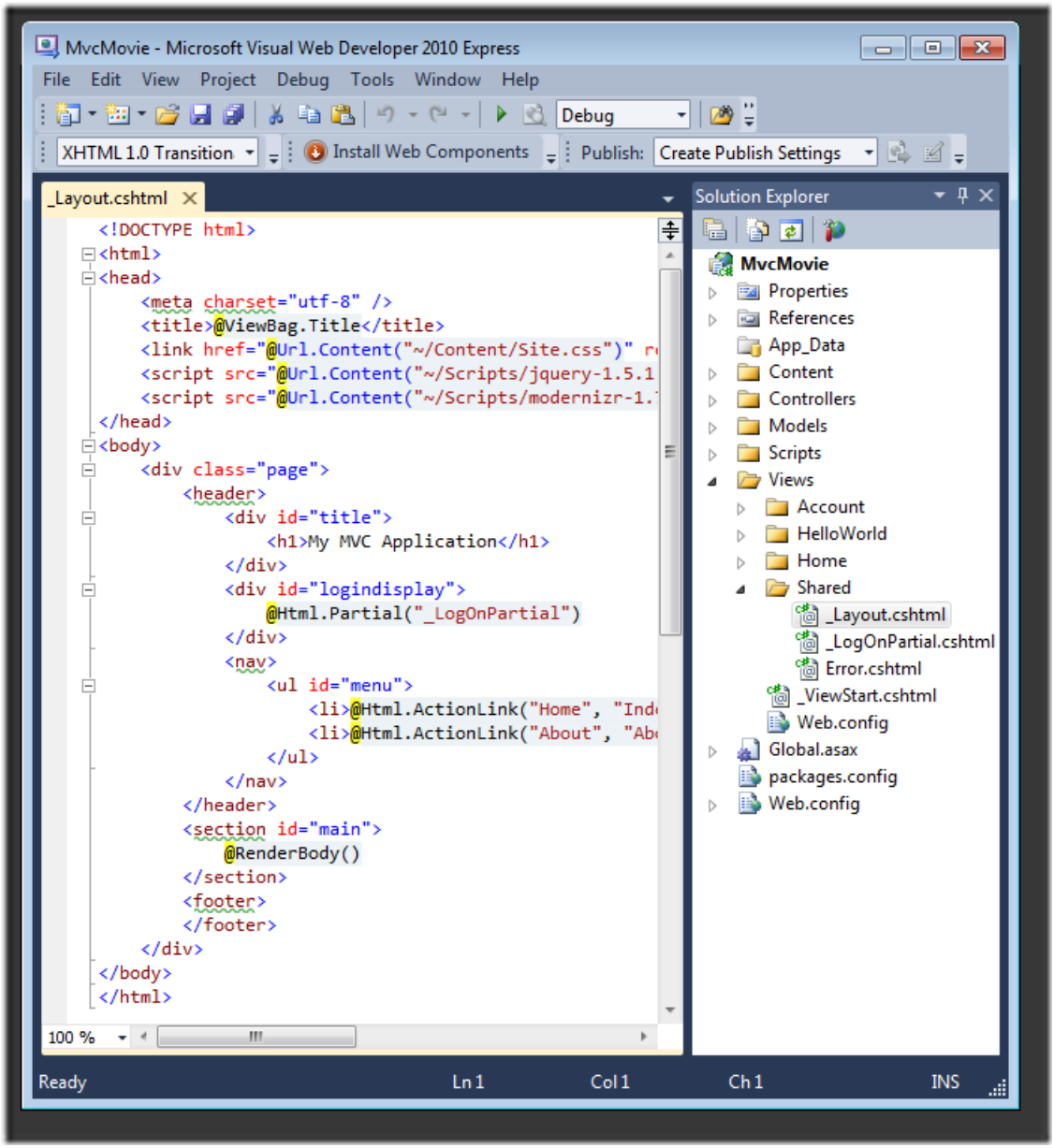
Run the application and browse to the `HelloWorld` controller (`http://localhost:xxxx/HelloWorld`). The `Index` method in your controller didn't do much work; it simply ran the statement `return View()`, which specified that the method should use a view template file to render a response to the browser. Because you didn't explicitly specify the name of the view template file to use, ASP.NET MVC defaulted to using the *Index.cshtml* view file in the *\Views\HelloWorld* folder. The image below shows the string hard-coded in the view.



Looks pretty good. However, notice that the browser's title bar says "Index" and the big title on the page says "My MVC Application." Let's change those.

Changing Views and Layout Pages

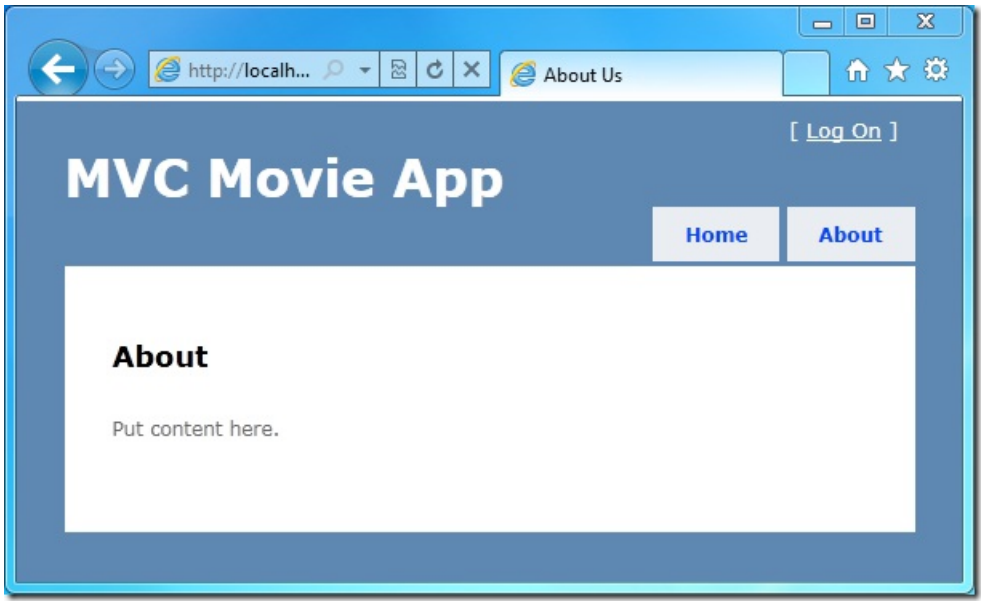
First, you want to change the "My MVC Application" title at the top of the page. That text is common to every page. It actually is implemented in only one place in the project, even though it appears on every page in the application. Go to the */Views/Shared* folder in **Solution Explorer** and open the *_Layout.cshtml* file. This file is called a *layout page* and it's the shared "shell" that all other pages use.



Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Note the `@RenderBody()` line near the bottom of the file. `RenderBody` is a placeholder where all the view-specific pages you create show up, "wrapped" in the layout page. Change the title heading in the layout template from "My MVC Application" to "MVC Movie App".

```
<div id="title">
      <h1>MVC Movie App</h1>
  </div>
```

Run the application and notice that it now says "MVC Movie App". Click the **About** link, and you see how that page shows "MVC Movie App", too. We were able to make the change once in the layout template and have all pages on the site reflect the new title.



The complete *_Layout.cshtml* file is shown below:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/modernizr-1.7.min.js")" type="text/javascript"></script>
</head>
<body>
    <div class="page">
        <header>
            <div id="title">
                <h1>MVC Movie App</h1>
            </div>
            <div id="logindisplay">
                @Html.Partial("_LogOnPartial")
            </div>
            <nav>
                <ul id="menu">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                </ul>
            </nav>
        </header>
        <section id="main">
            @RenderBody()
        </section>
        <footer>
        </footer>
    </div>
</body>
</html>
```

Now, let's change the title of the Index page (view).

Open *MvcMovie\Views\HelloWorld\Index.cshtml*. There are two places to make a change: first, the text that appears in the title of the browser, and then in the secondary header (the `<h2>` element). You'll make them slightly different so you can see which bit of code changes which part of the app.

```
@{
    ViewBag.Title = "Movie List";
}

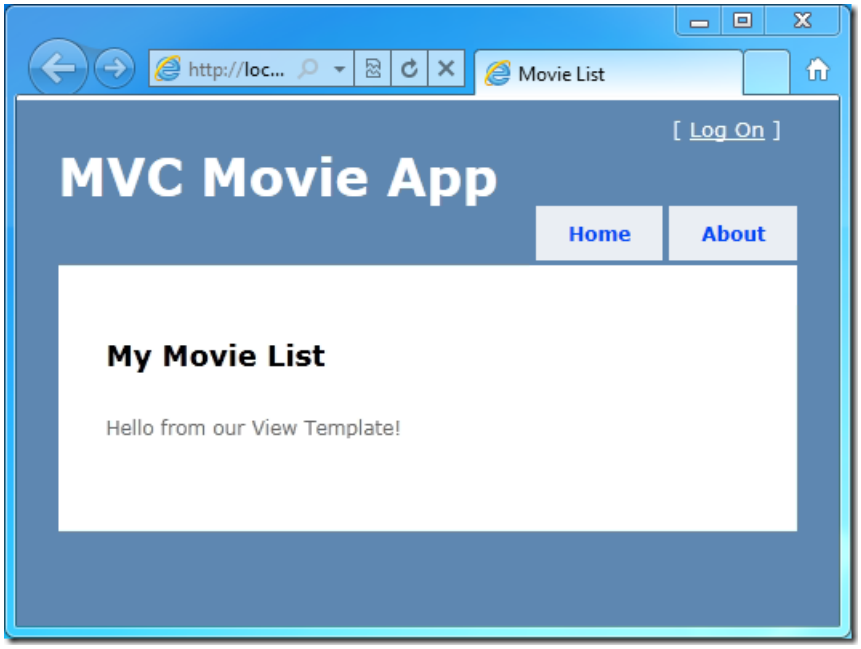
<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

To indicate the HTML title to display, the code above sets a `Title` property of the `ViewBag` object (which is in the *Index.cshtml* view template). If you look back at the source code of the layout template, you'll notice that the template uses this value in the `<title>` element as part of the `<head>` section of the HTML. Using this approach, you can easily pass other parameters between your view template and your layout file.

Run the application and browse to `http://localhost:xx/HelloWorld`. Notice that the browser title, the primary heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing cached content. Press Ctrl+F5 in your browser to force the response from the server to be loaded.)

Also notice how the content in the *Index.cshtml* view template was merged with the *_Layout.cshtml* view template and a single HTML response was sent to the browser. Layout templates make it really easy to make changes that apply across all of the pages in your application.



Our little bit of "data" (in this case the "Hello from our View Template!" message) is hard-coded, though. The MVC application has a "V" (view) and you've got a "C" (controller), but no "M" (model) yet. Shortly, we'll walk through how create a database and retrieve model data from it.

Passing Data from the Controller to the View

Before we go to a database and talk about models, though, let's first talk about passing information from the controller to a view. Controller classes are invoked in response to an incoming URL request. A controller class is where you write the code that handles the incoming parameters, retrieves data from a database, and ultimately decides what type of response to send back to the browser. View templates can then be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing whatever data or objects are required in order for a view template to render a response to the browser. A view template should never perform business logic or interact with a database directly. Instead, it should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep your code clean and more maintainable.

Currently, the `Welcome` action method in the `HelloWorldController` class takes a `name` and a `numTimes` parameter and then outputs the values directly to the browser. Rather than have the controller render this response as a string, let's change the controller to use a view template instead. The view template will generate a dynamic response, which means that you need to pass appropriate bits of data from the controller to the view in order to generate the response. You can do this by having the controller put the dynamic data that the view template needs in a `ViewBag` object that the view template can then access.

Return to the *HelloWorldController.cs* file and change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewBag` object. `ViewBag` is a dynamic object, which means you can put whatever you want in to it; the `ViewBag` object has no defined properties until you put something inside it. The complete *HelloWorldController.cs* file looks like this:

```
using System.Web;
using System.Web.Mvc;

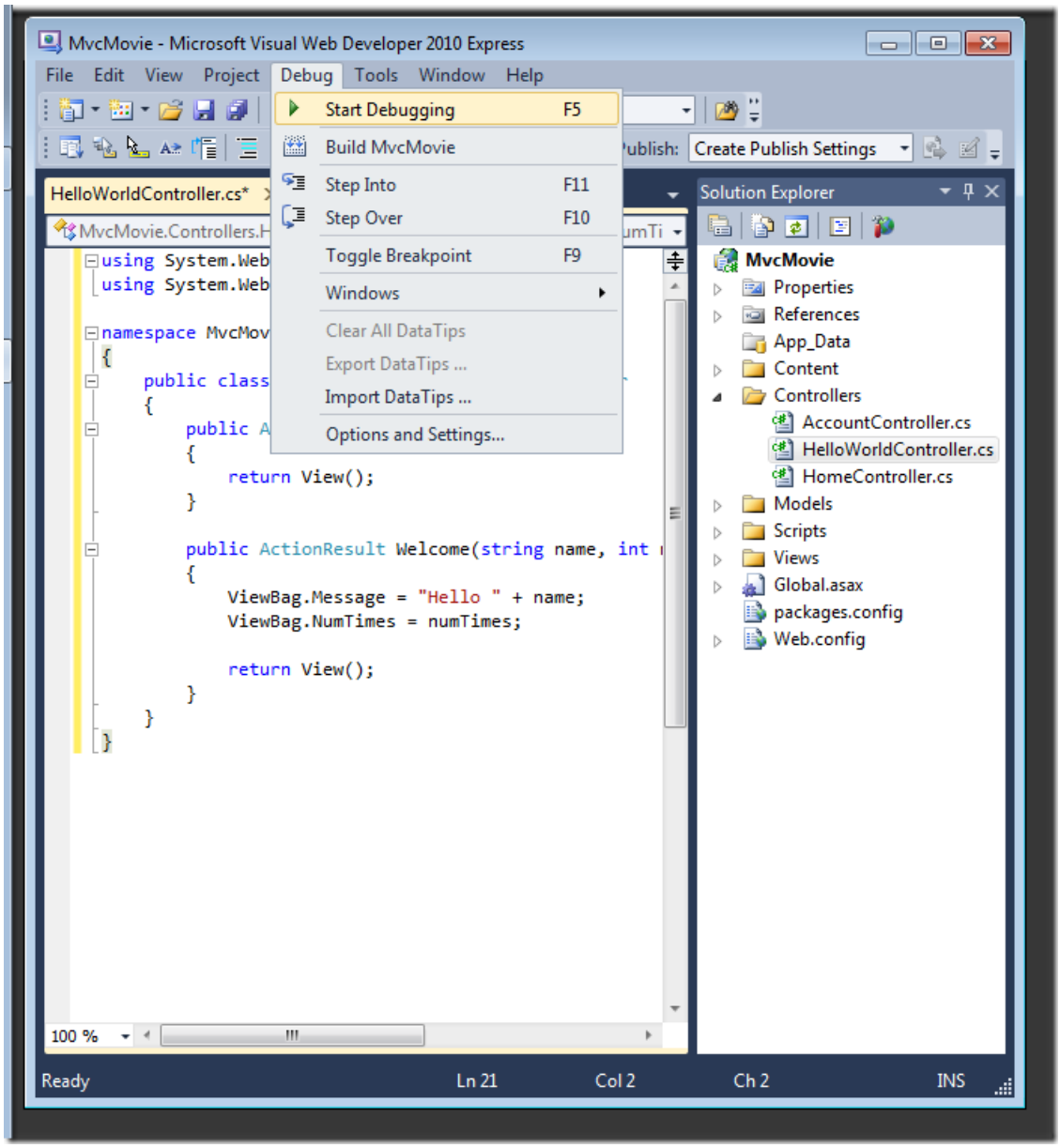
namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Welcome(string name, int numTimes = 1)
        {
            ViewBag.Message = "Hello " + name;
            ViewBag.NumTimes = numTimes;

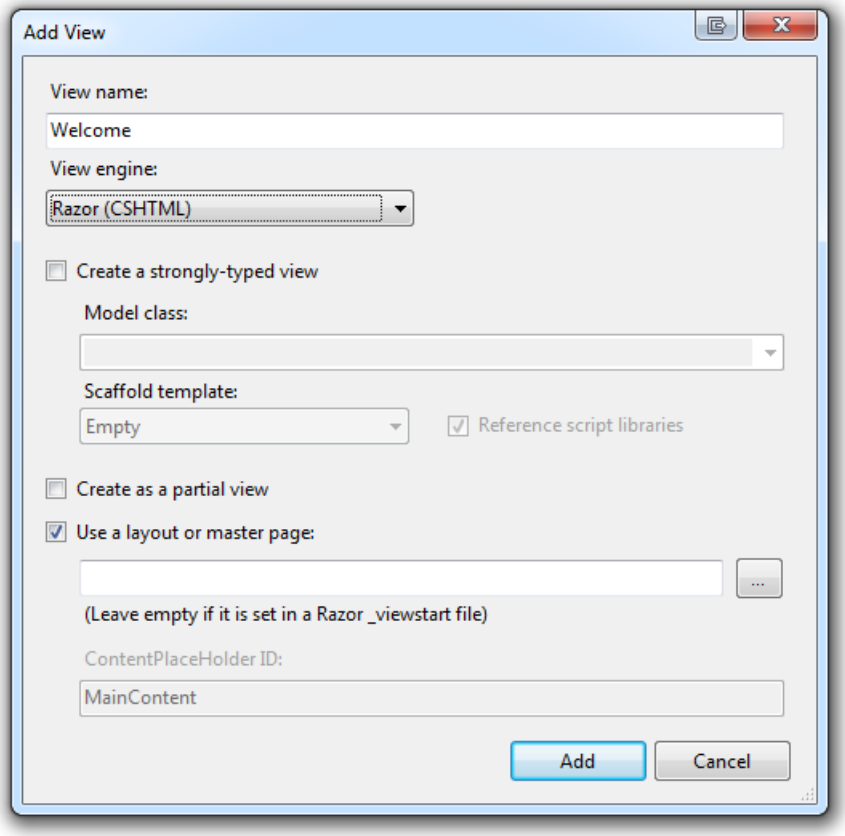
            return View();
        }
    }
}
```

Now the `ViewBag` object contains data that will be passed to the view automatically.

Next, you need a Welcome view template! In the **Debug** menu, select **Build MvcMovie** to make sure the project is compiled.



Then right-click inside the `Welcome` method and click **Add View**. Here's what the **Add View** dialog box looks like:



Click **Add**, and then add the following code under the `<h2>` element in the new *Welcome.cshtml* file. You'll create a loop that says "Hello" as many times as the user says it should. The complete *Welcome.cshtml* file is shown below.

```
@{
    ViewBag.Title = "Welcome";
}

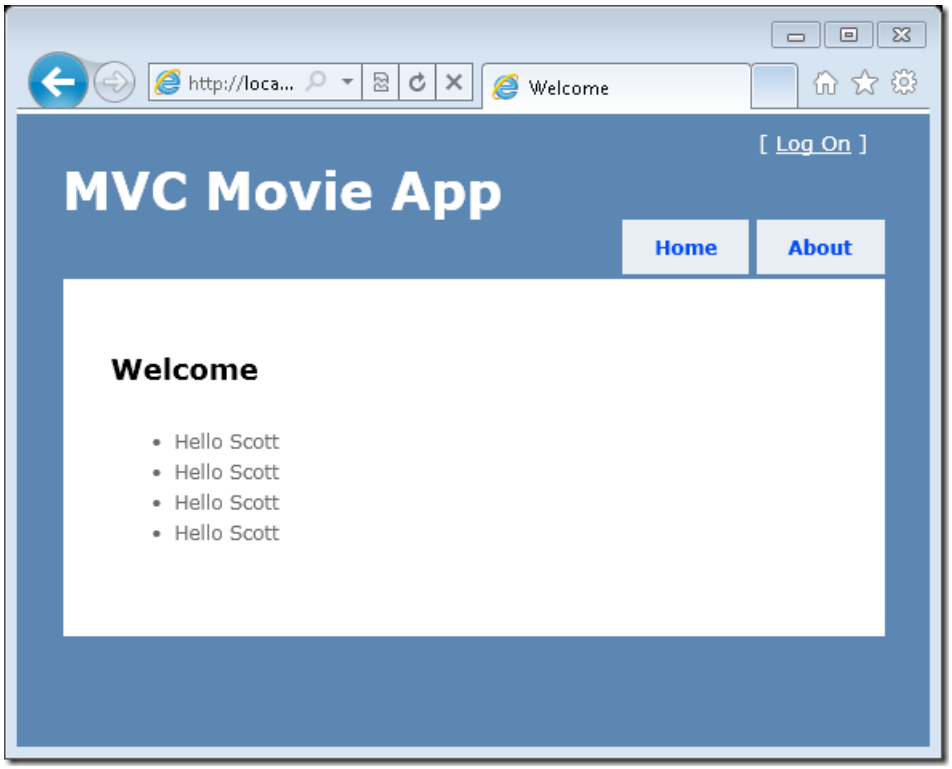
<h2>Welcome</h2>

<ul>
   @for (int i=0; i < ViewBag.NumTimes; i++) {
      <li>@ViewBag.Message</li>
   }
</ul>
```

Run the application and browse to the following URL:

`http://localhost:xx/HelloWorld/Welcome?name=Scott&numtimes=4`

Now data is taken from the URL and passed to the controller automatically. The controller packages the data into a `ViewBag` object and passes that object to the view. The view then displays the data as HTML to the user.



Well, that was a kind of an "M" for model, but not the database kind. Let's take what we've learned and create a database of movies.

PREVIOUS NEXT

Adding a Model (C#)

2/20/2020 • 3 minutes to read • Edit Online

by Rick Anderson

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

A Visual Web Developer project with C# source code is available to accompany this topic. Download the C# version. If you prefer Visual Basic, switch to the Visual Basic version of this tutorial.

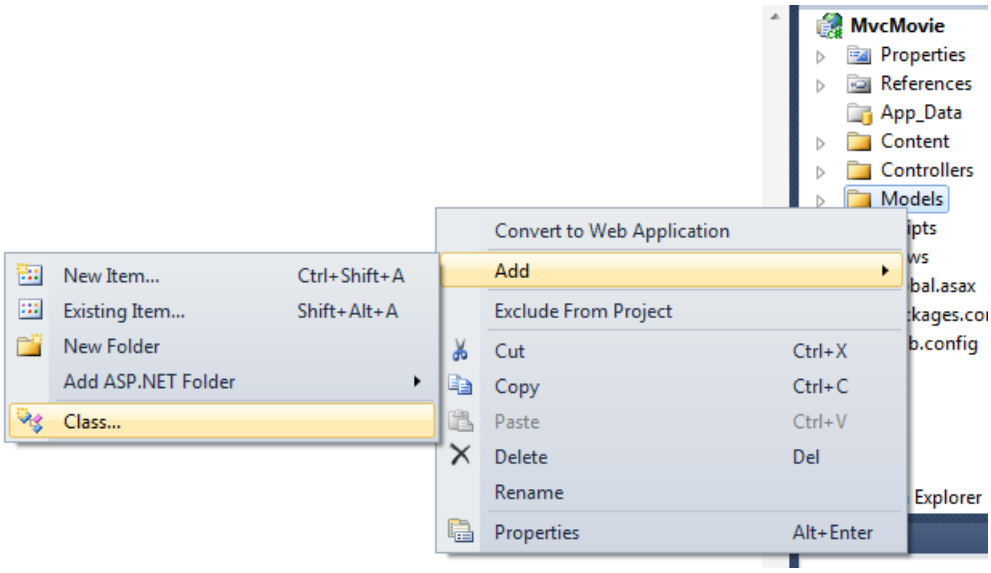
Adding a Model

In this section you'll add some classes for managing movies in a database. These classes will be the "model" part of the ASP.NET MVC application.

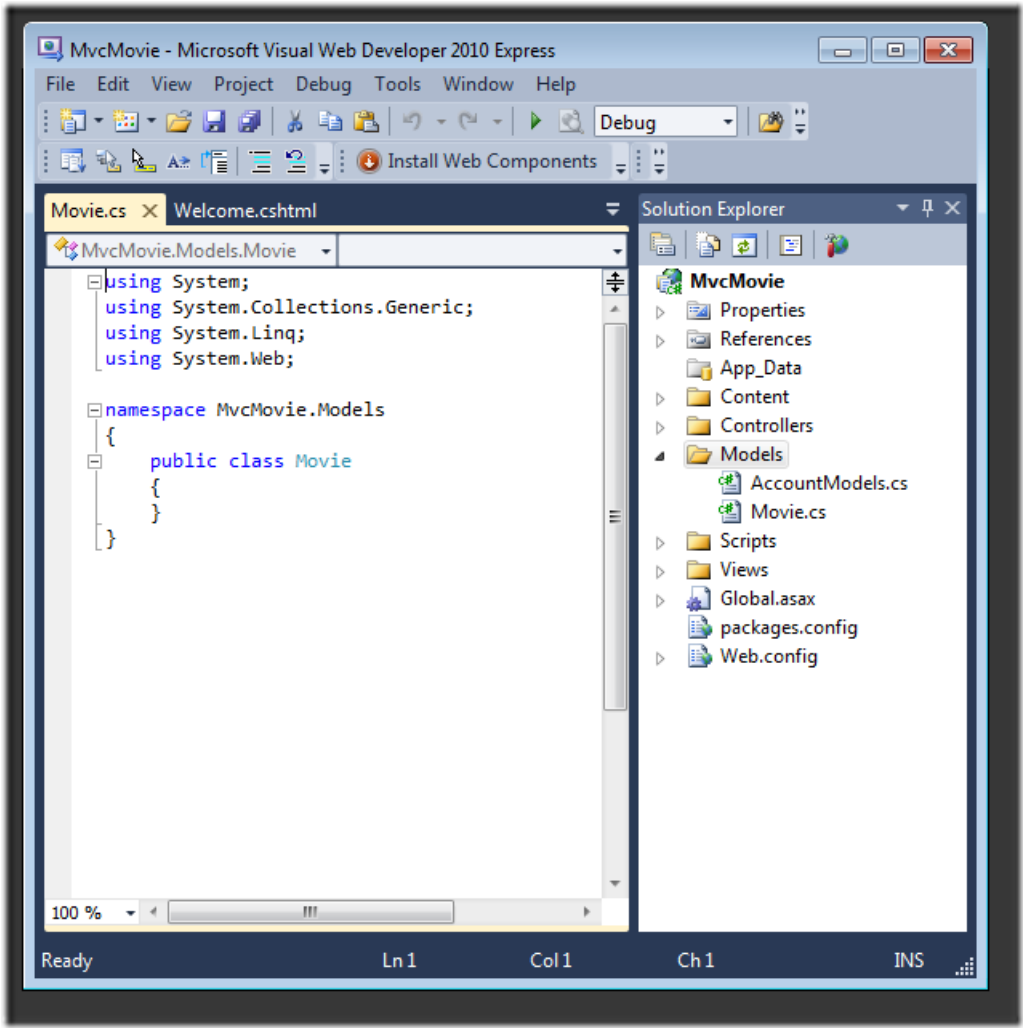
You'll use a .NET Framework data-access technology known as the Entity Framework to define and work with these model classes. The Entity Framework (often referred to as EF) supports a development paradigm called *Code First*. Code First allows you to create model objects by writing simple classes. (These are also known as POCO classes, from "plain-old CLR objects.") You can then have the database created on the fly from your classes, which enables a very clean and rapid development workflow.

Adding Model Classes

In **Solution Explorer**, right click the *Models* folder, select **Add**, and then select **Class**.



Name the *class* "Movie".



Add the following five properties to the `Movie` class:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
}
```

We'll use the `Movie` class to represent movies in a database. Each instance of a `Movie` object will correspond to a row within a database table, and each property of the `Movie` class will map to a column in the table.

In the same file, add the following `MovieDBContext` class:

```
public class MovieDBContext : DbContext
{
    public DbSet<Movie> Movies { get; set; }
}
```

The `MovieDBContext` class represents the Entity Framework movie database context, which handles fetching, storing, and updating `Movie` class instances in a database. The `MovieDBContext` derives from the `DbContext` base class provided by the Entity Framework. For more information about `DbContext` and `DbSet`, see Productivity Improvements for the Entity Framework.

In order to be able to reference `DbContext` and `DbSet`, you need to add the following `using` statement at the top of the file:

```
using System.Data.Entity;
```

The complete *Movie.cs* file is shown below.

```
using System;
using System.Data.Entity;

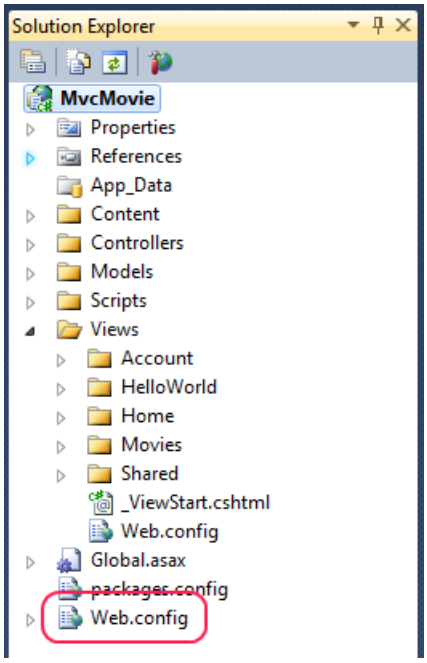
namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }

    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

Creating a Connection String and Working with SQL Server Compact

The `MovieDBContext` class you created handles the task of connecting to the database and mapping `Movie` objects to database records. One question you might ask, though, is how to specify which database it will connect to. You'll do that by adding connection information in the *Web.config* file of the application.

Open the application root *Web.config* file. (Not the *Web.config* file in the *Views* folder.) The image below show both *Web.config* files; open the *Web.config* file circled in red.



Add the following connection string to the `<connectionStrings>` element in the *Web.config* file.

```
<add name="MovieDBContext"
         connectionString="Data Source=|DataDirectory|Movies.sdf"
         providerName="System.Data.SqlServerCe.4.0"/>
```

The following example shows a portion of the *Web.config* file with the new connection string added:

```
<configuration>
  <connectionStrings>
    <add name="MovieDBContext"
         connectionString="Data Source=|DataDirectory|Movies.sdf"
         providerName="System.Data.SqlServerCe.4.0"/>
    <add name="ApplicationServices"
         connectionString="data source=.\SQLEXPRESS;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User Instance=true"
         providerName="System.Data.SqlClient" />
  </connectionStrings>
```

This small amount of code and XML is everything you need to write in order to represent and store the movie data in a database.

Next, you'll build a new `MoviesController` class that you can use to display the movie data and allow users to create new movie listings.

PREVIOUS NEXT

Accessing your Model's Data from a Controller (C#)

2/20/2020 • 6 minutes to read • Edit Online

by Rick Anderson

> **NOTE**
>
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

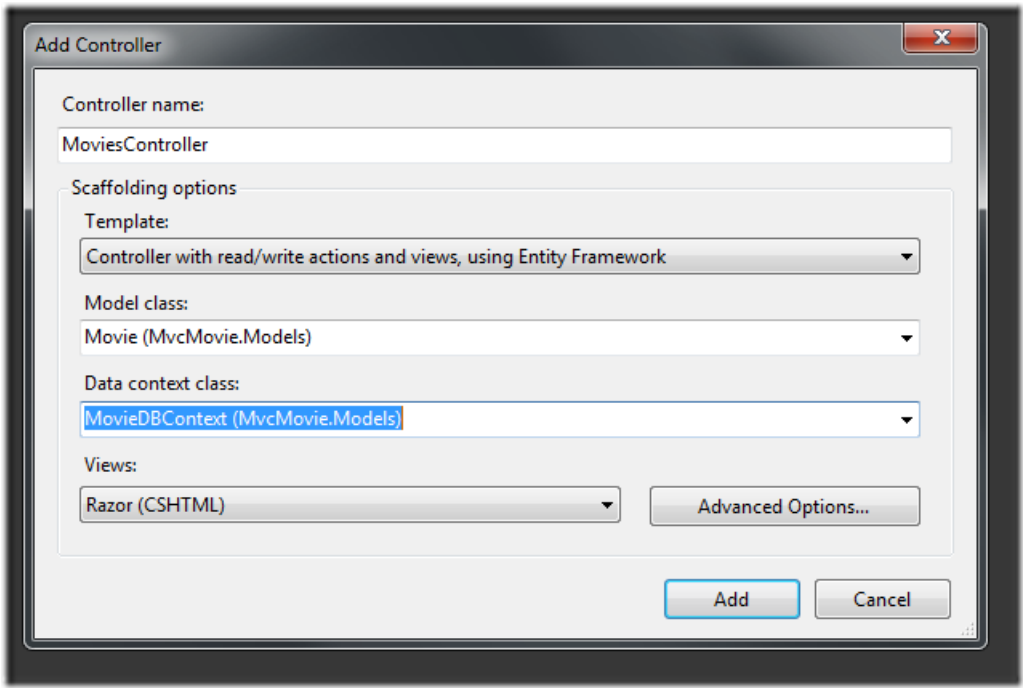
If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

A Visual Web Developer project with C# source code is available to accompany this topic. Download the C# version. If you prefer Visual Basic, switch to the Visual Basic version of this tutorial.

In this section, you'll create a new `MoviesController` class and write code that retrieves the movie data and displays it in the browser using a view template. Be sure to build your application before proceeding.

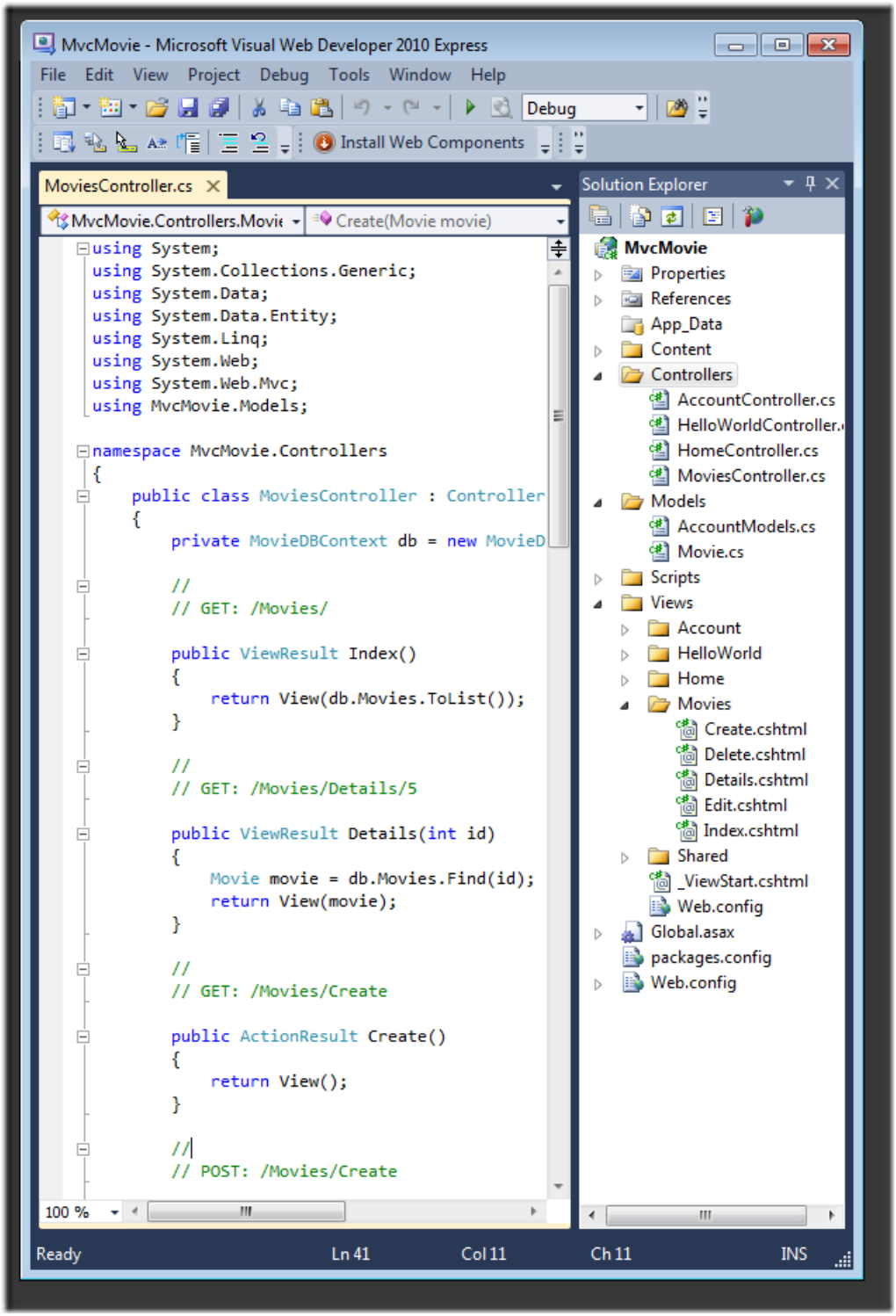
Right-click the *Controllers* folder and create a new `MoviesController` controller. Select the following options:

- Controller name: **MoviesController**. (This is the default.)
- Template: **Controller with read/write actions and views, using Entity Framework**.
- Model class: **Movie (MvcMovie.Models)**.
- Data context class: **MovieDBContext (MvcMovie.Models)**.
- Views: **Razor (CSHTML)**. (The default.)



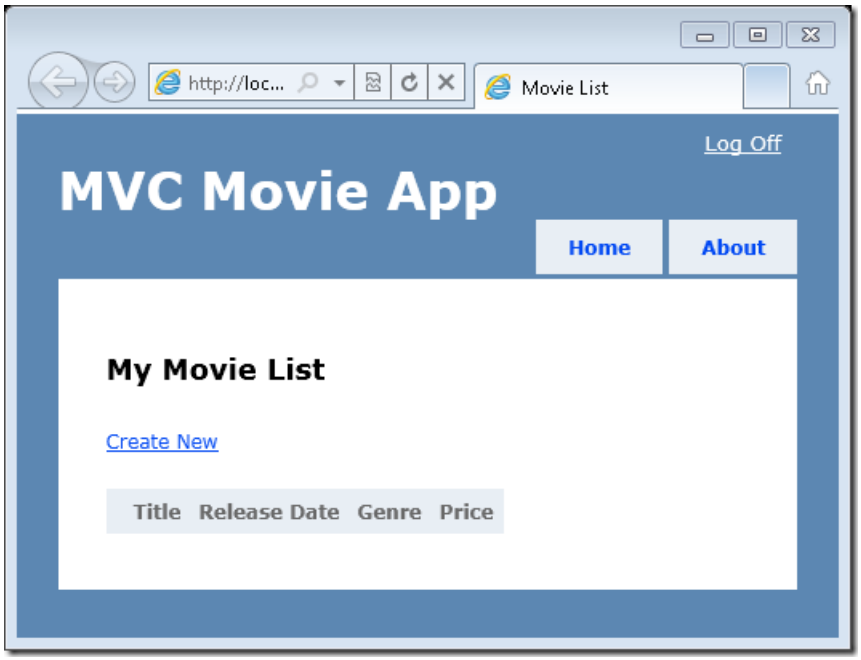
Click **Add**. Visual Web Developer creates the following files and folders:

- *A MoviesController.cs* file in the project's *Controllers* folder.
- A *Movies* folder in the project's *Views* folder.
- *Create.cshtml, Delete.cshtml, Details.cshtml, Edit.cshtml*, and *Index.cshtml* in the new *Views\Movies* folder.



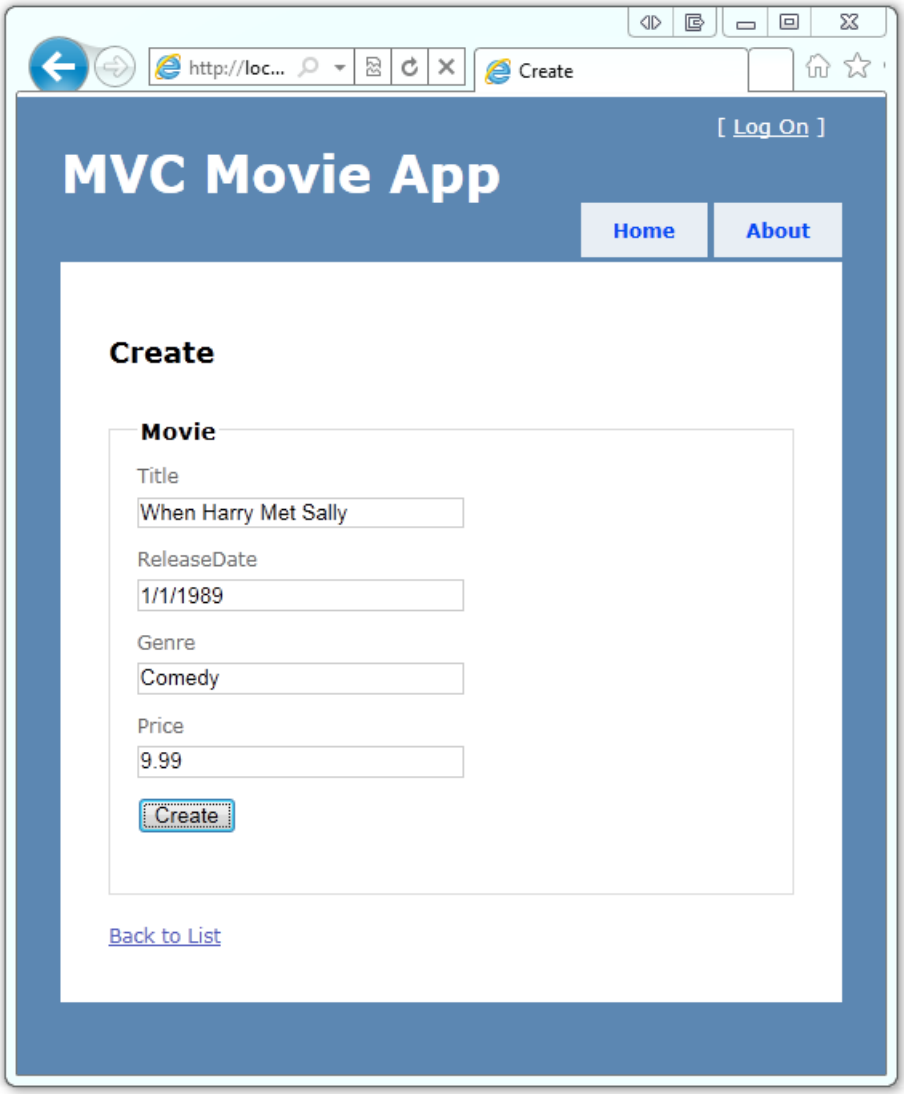
The ASP.NET MVC 3 scaffolding mechanism automatically created the CRUD (create, read, update, and delete) action methods and views for you. You now have a fully functional web application that lets you create, list, edit, and delete movie entries.

Run the application and browse to the `Movies` controller by appending */Movies* to the URL in the address bar of your browser. Because the application is relying on the default routing (defined in the *Global.asax* file), the browser request `http://localhost:xxxxx/Movies` is routed to the default `Index` action method of the `Movies` controller. In other words, the browser request `http://localhost:xxxxx/Movies` is effectively the same as the browser request `http://localhost:xxxxx/Movies/Index`. The result is an empty list of movies, because you haven't added any yet.

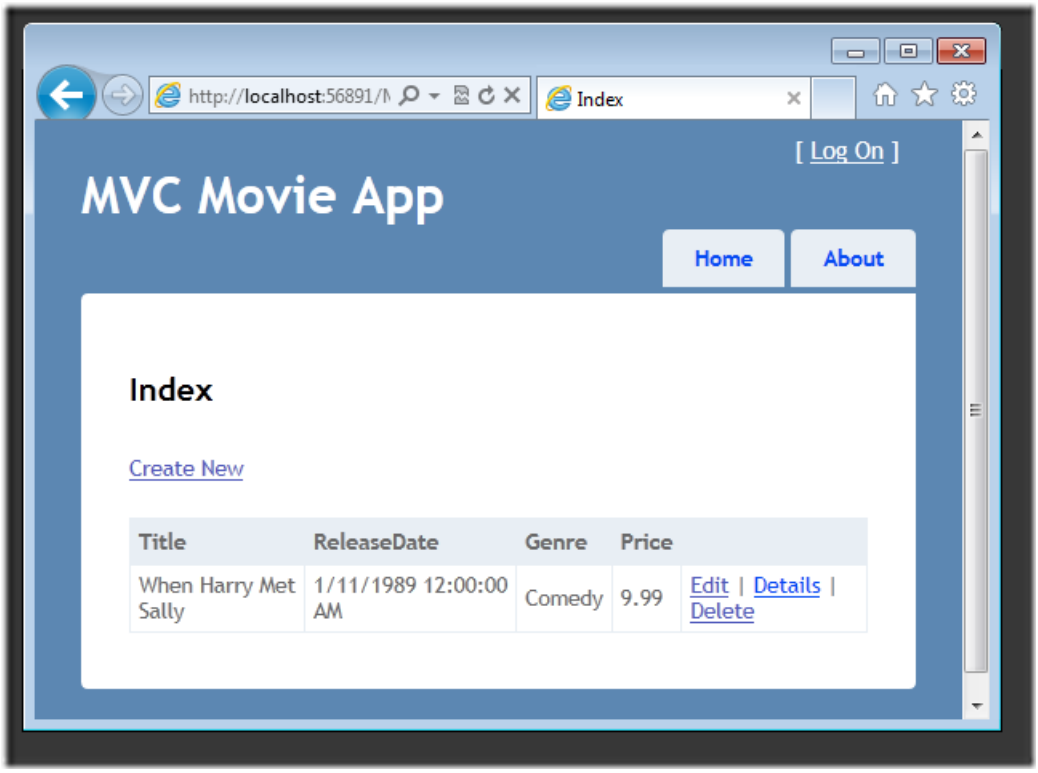


Creating a Movie

Select the **Create New** link. Enter some details about a movie and then click the **Create** button.



Clicking the **Create** button causes the form to be posted to the server, where the movie information is saved in the database. You're then redirected to the */Movies* URL, where you can see the newly created movie in the listing.



Create a couple more movie entries. Try the **Edit**, **Details**, and **Delete** links, which are all functional.

Examining the Generated Code

Open the *Controllers\MoviesController.cs* file and examine the generated `Index` method. A portion of the movie controller with the `Index` method is shown below.

```
public class MoviesController : Controller
{
    private MovieDBContext db = new MovieDBContext();

    //
    // GET: /Movies/

    public ViewResult Index()
    {
        return View(db.Movies.ToList());
    }
```

The following line from the `MoviesController` class instantiates a movie database context, as described previously. You can use the movie database context to query, edit, and delete movies.

```
private MovieDBContext db = new MovieDBContext();
```

A request to the `Movies` controller returns all the entries in the `Movies` table of the movie database and then passes the results to the `Index` view.

Strongly Typed Models and the @model Keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view template using the `ViewBag` object. The `ViewBag` is a dynamic object that provides a convenient late-bound way to pass information to a view.

ASP.NET MVC also provides the ability to pass strongly typed data or objects to a view template. This strongly typed approach enables better compile-time checking of your code and richer IntelliSense in the Visual Web

Developer editor. We're using this approach with the `MoviesController` class and *Index.cshtml* view template.

Notice how the code creates a `List` object when it calls the `View` helper method in the `Index` action method. The code then passes this `Movies` list from the controller to the view:

```
public ViewResult Index()
{
    return View(db.Movies.ToList());
}
```

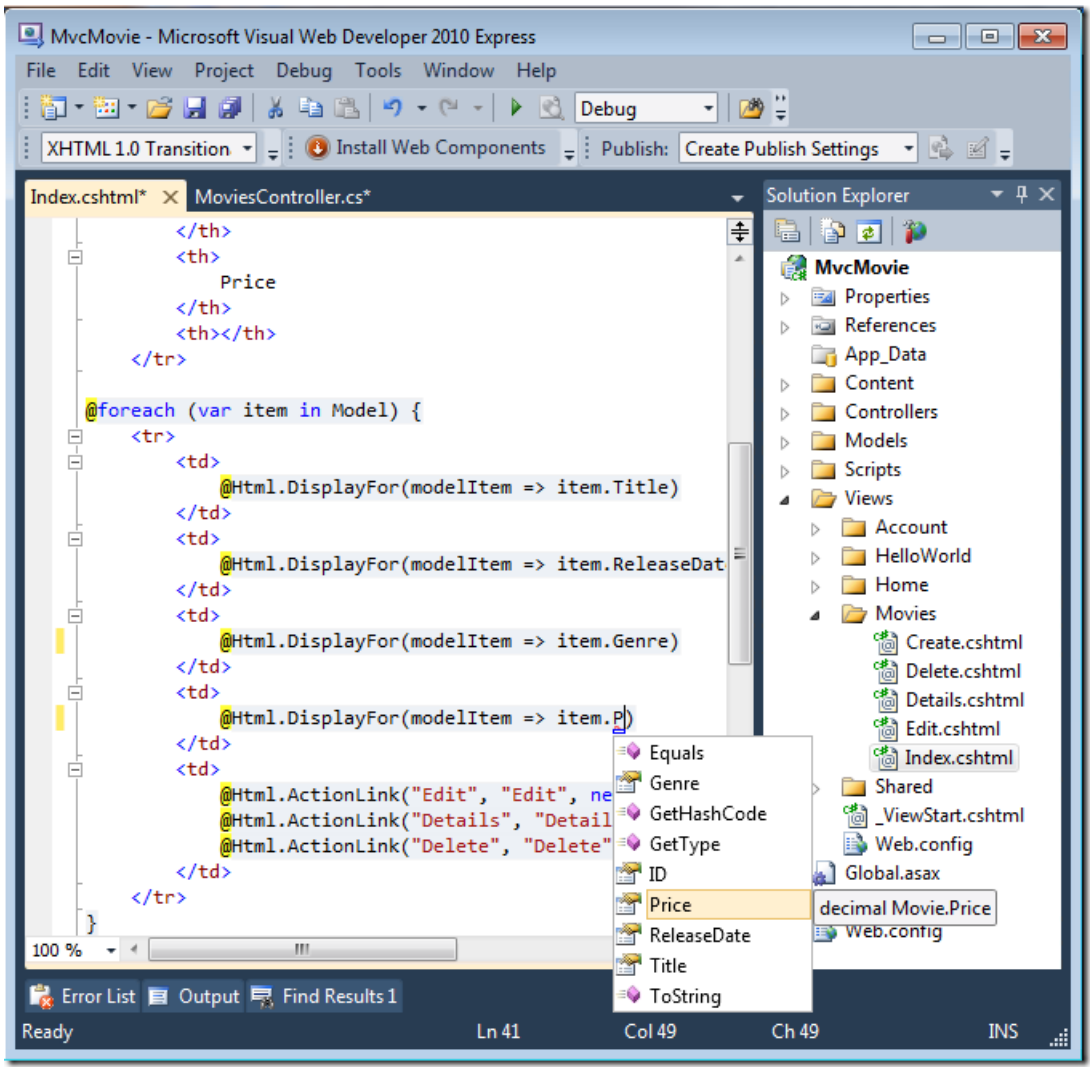
By including a `@model` statement at the top of the view template file, you can specify the type of object that the view expects. When you created the movie controller, Visual Web Developer automatically included the following `@model` statement at the top of the *Index.cshtml* file:

```
@model IEnumerable<MvcMovie.Models.Movie>
```

This `@model` directive allows you to access the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Index.cshtml* template, the code loops through the movies by doing a `foreach` statement over the strongly typed `Model` object:

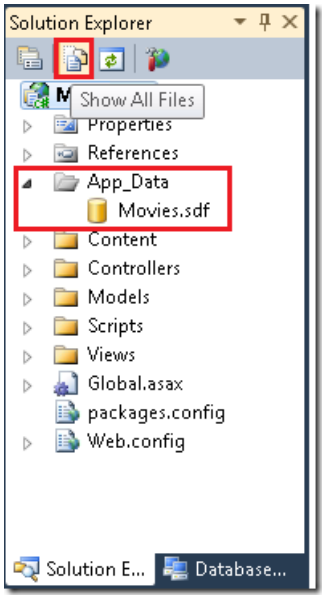
```
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
    </tr>
}
```

Because the `Model` object is strongly typed (as an `IEnumerable<Movie>` object), each `item` object in the loop is typed as `Movie`. Among other benefits, this means that you get compile-time checking of the code and full IntelliSense support in the code editor:



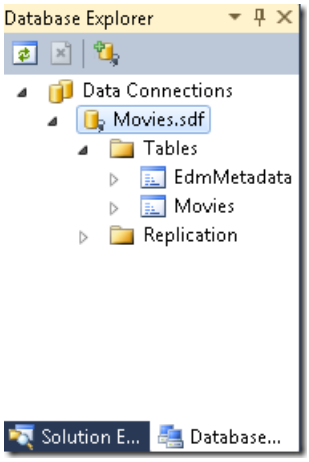
Working with SQL Server Compact

Entity Framework Code First detected that the database connection string that was provided pointed to a `Movies` database that didn't exist yet, so Code First created the database automatically. You can verify that it's been created by looking in the *App_Data* folder. If you don't see the *Movies.sdf* file, click the **Show All Files** button in the **Solution Explorer** toolbar, click the **Refresh** button, and then expand the *App_Data* folder.



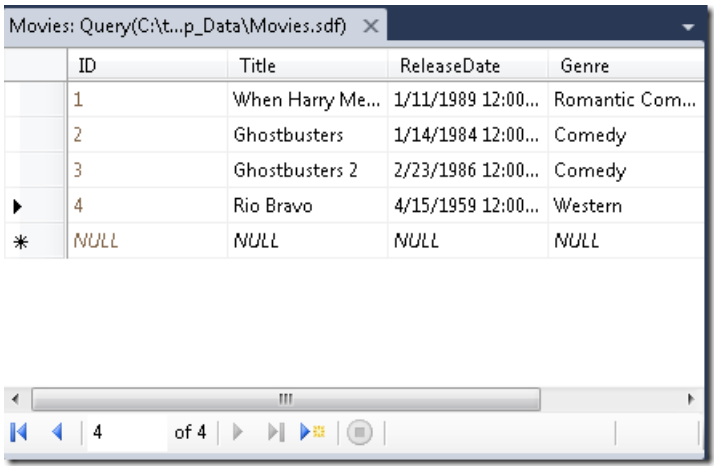
Double-click *Movies.sdf* to open **Server Explorer**. Then expand the **Tables** folder to see the tables that have been created in the database.

> **NOTE**
>
> If you get an error when you double-click *Movies.sdf*, make sure you've installed SQL Server Compact 4.0(runtime + tools support). (For links to the software, see the list of prerequisites in part 1 of this tutorial series.) If you install the release now, you'll have to close and re-open Visual Web Developer.

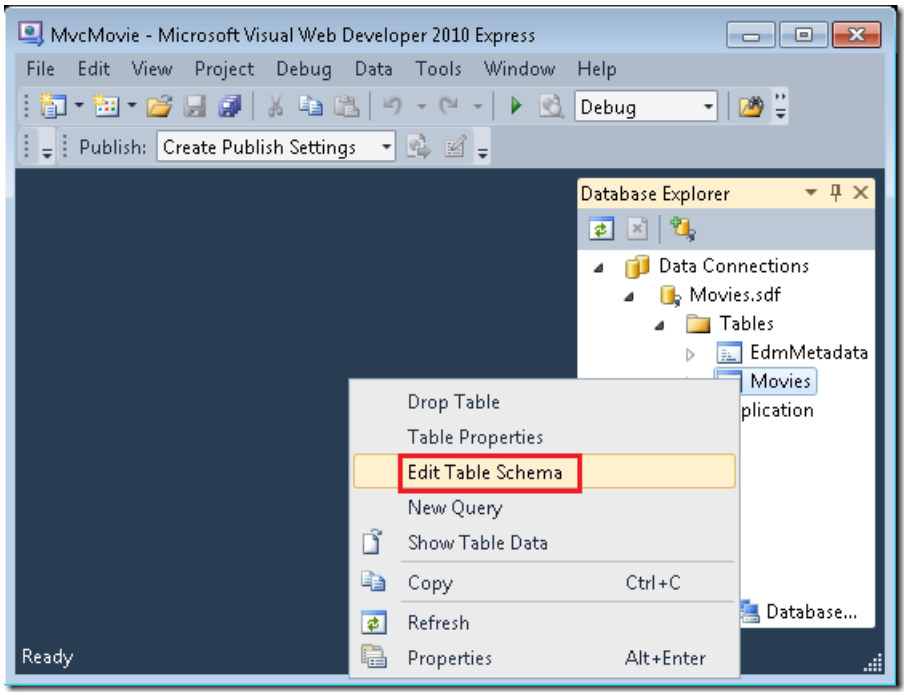


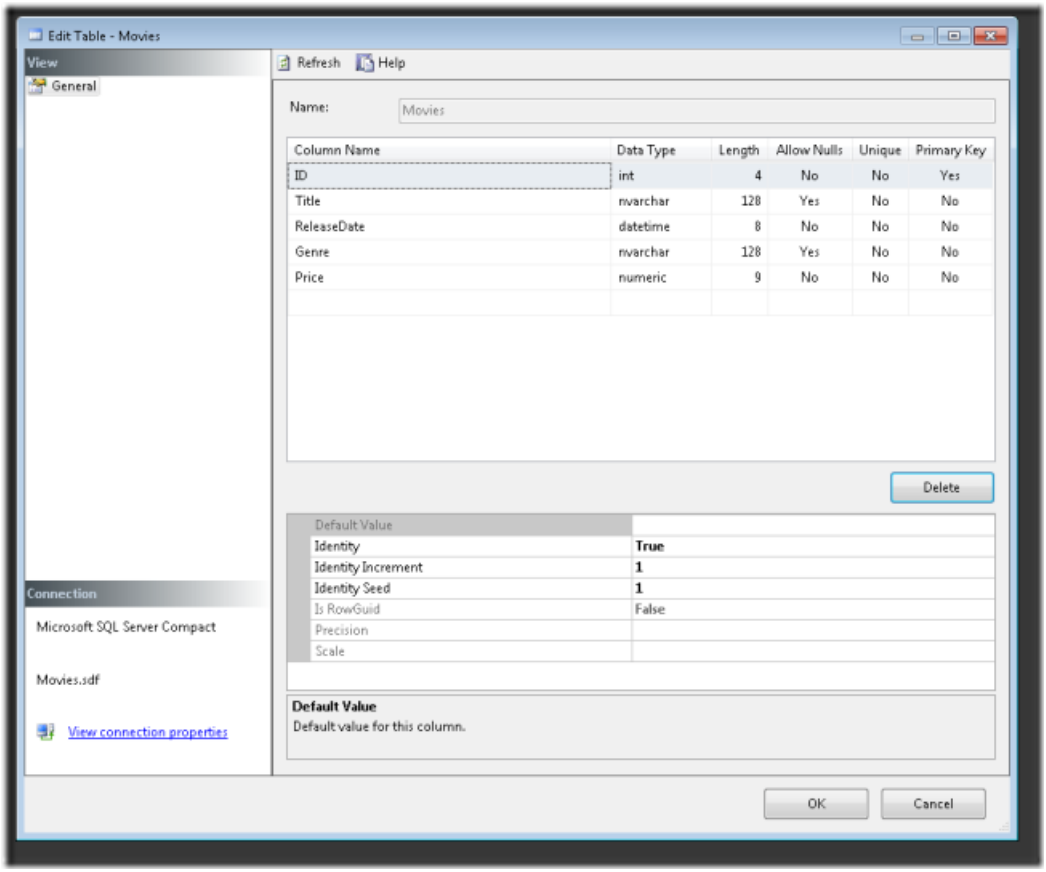
There are two tables, one for the `Movie` entity set and then the `EdmMetadata` table. The `EdmMetadata` table is used by the Entity Framework to determine when the model and the database are out of sync.

Right-click the `Movies` table and select **Show Table Data** to see the data you created.



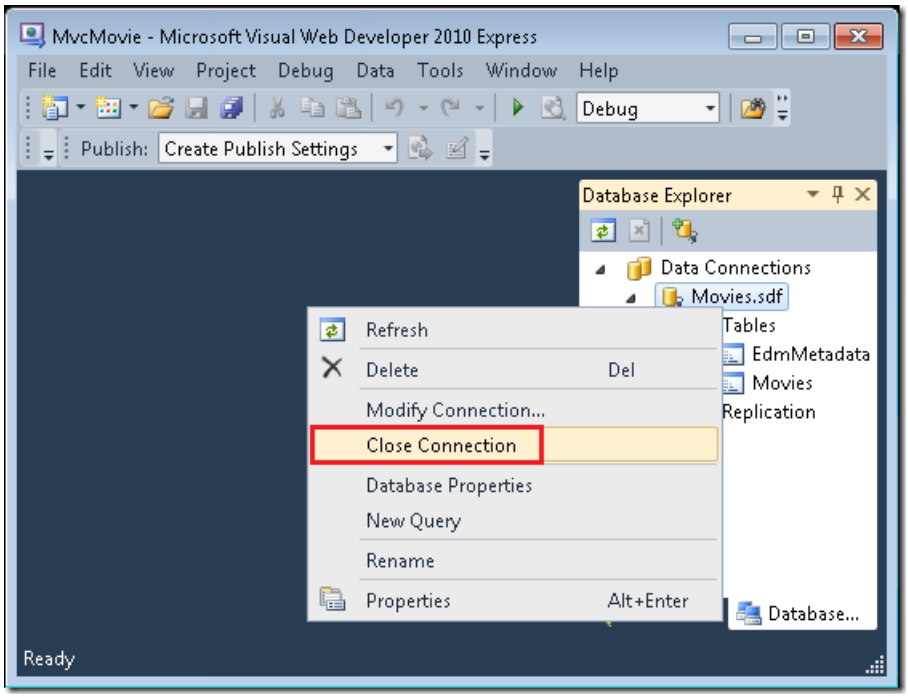
Right-click the `Movies` table and select **Edit Table Schema**.





Notice how the schema of the `Movies` table maps to the `Movie` class you created earlier. Entity Framework Code First automatically created this schema for you based on your `Movie` class.

When you're finished, close the connection. (If you don't close the connection, you might get an error the next time you run the project).



You now have the database and a simple listing page to display content from it. In the next tutorial, we'll examine the rest of the scaffolded code and add a `SearchIndex` method and a `SearchIndex` view that lets you search for movies in this database.

PREVIOUS NEXT

Examining the Edit Methods and Edit View (C#)

9/16/2020 • 14 minutes to read • Edit Online

by Rick Anderson

> **NOTE**
>
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

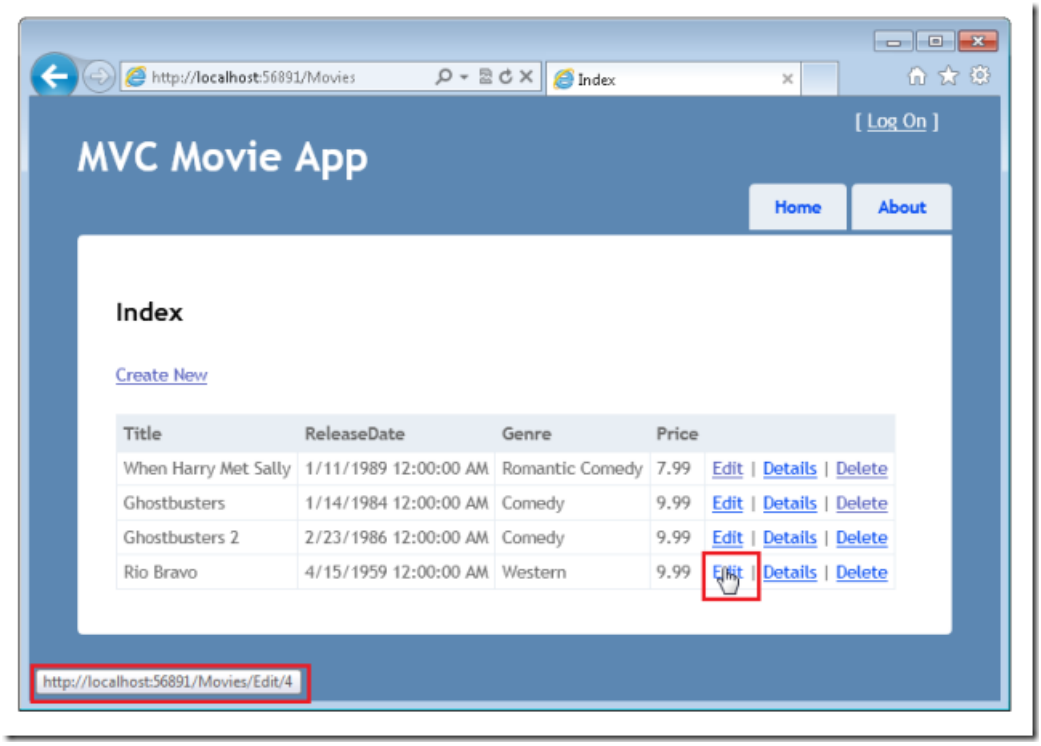
- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

A Visual Web Developer project with C# source code is available to accompany this topic. Download the C# version. If you prefer Visual Basic, switch to the Visual Basic version of this tutorial.

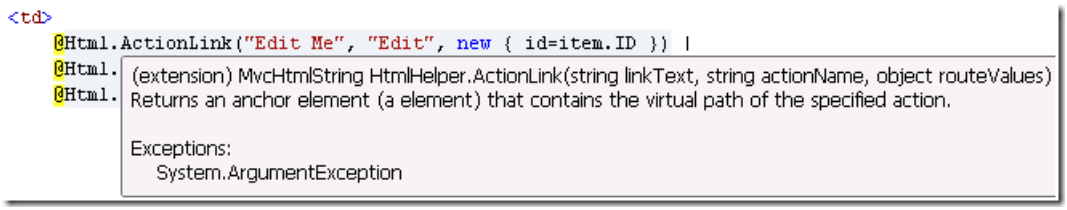
In this section, you'll examine the generated action methods and views for the movie controller. Then you'll add a custom search page.

Run the application and browse to the `Movies` controller by appending */Movies* to the URL in the address bar of your browser. Hold the mouse pointer over an **Edit** link to see the URL that it links to.



The **Edit** link was generated by the `Html.ActionLink` method in the *Views\Movies\Index.cshtml* view:

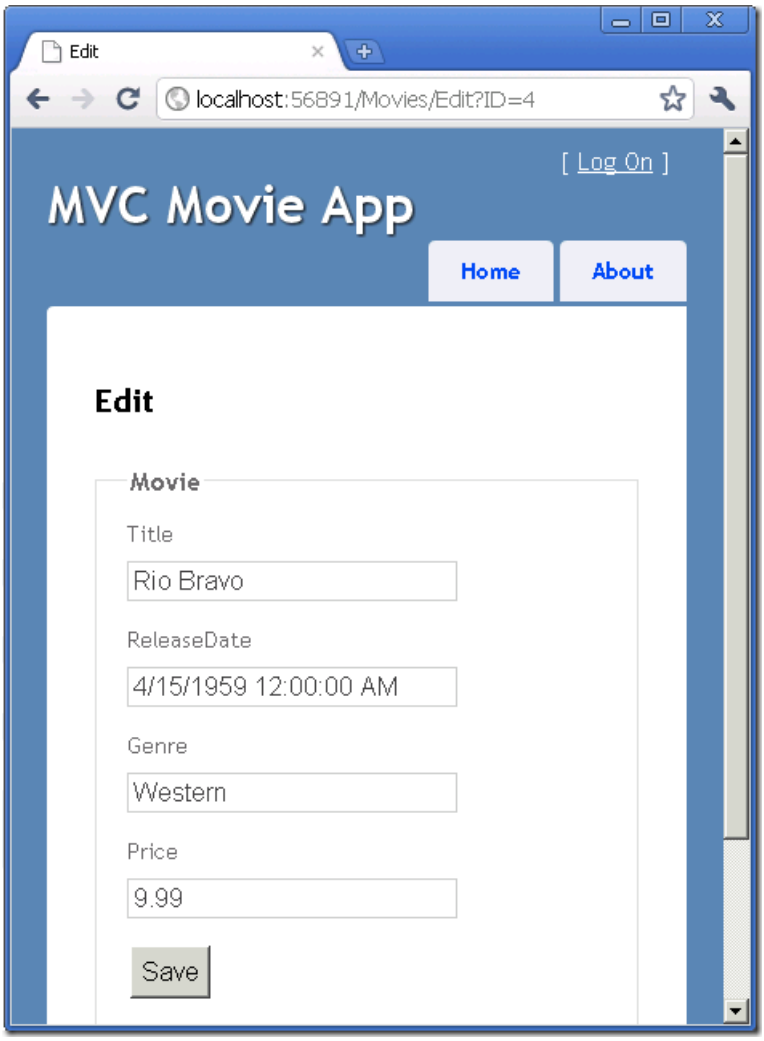
```
@Html.ActionLink("Edit", "Edit", new { id=item.ID })
```



The `Html` object is a helper that's exposed using a property on the `WebViewPage` base class. The `ActionLink` method of the helper makes it easy to dynamically generate HTML hyperlinks that link to action methods on controllers. The first argument to the `ActionLink` method is the link text to render (for example, `<a>Edit Me</a>`). The second argument is the name of the action method to invoke. The final argument is an anonymous object that generates the route data (in this case, the ID of 4).

The generated link shown in the previous image is `http://localhost:xxxxx/Movies/Edit/4`. The default route takes the URL pattern `{controller}/{action}/{id}`. Therefore, ASP.NET translates `http://localhost:xxxxx/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `ID` equal to 4.

You can also pass action method parameters using a query string. For example, the URL `http://localhost:xxxxx/Movies/Edit?ID=4` also passes the parameter `ID` of 4 to the `Edit` action method of the `Movies` controller.



Open the `Movies` controller. The two `Edit` action methods are shown below.

```
//
// GET: /Movies/Edit/5

public ActionResult Edit(int id)
{
    Movie movie = db.Movies.Find(id);
    return View(movie);
}

//
// POST: /Movies/Edit/5

[HttpPost]
public ActionResult Edit(Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Entry(movie).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

Notice the second `Edit` action method is preceded by the `HttpPost` attribute. This attribute specifies that overload of the `Edit` method can be invoked only for POST requests. You could apply the `HttpGet` attribute to the first edit method, but that's not necessary because it's the default. (We'll refer to action methods that are implicitly assigned the `HttpGet` attribute as `HttpGet` methods.)

The `HttpGet` `Edit` method takes the movie ID parameter, looks up the movie using the Entity Framework `Find` method, and returns the selected movie to the Edit view. When the scaffolding system created the Edit view, it examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The following example shows the Edit view that was generated:

```
@model MvcMovie.Models.Movie

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Movie</legend>

        @Html.HiddenFor(model => model.ID)

        <div class="editor-label">
            @Html.LabelFor(model => model.Title)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.ReleaseDate)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.ReleaseDate)
            @Html.ValidationMessageFor(model => model.ReleaseDate)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Genre)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Genre)
            @Html.ValidationMessageFor(model => model.Genre)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Price)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Price)
            @Html.ValidationMessageFor(model => model.Price)
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Notice how the view template has a `@model MvcMovie.Models.Movie` statement at the top of the file — this specifies that the view expects the model for the view template to be of type `Movie`.

The scaffolded code uses several *helper methods* to streamline the HTML markup. The `Html.LabelFor` helper displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The `Html.EditorFor` helper displays an HTML `<input>` element. The `Html.ValidationMessageFor` helper displays any validation messages associated with that property.

Run the application and navigate to the */Movies* URL. Click an **Edit** link. In the browser, view the source for the page. The HTML in the page looks like the following example. (The menu markup was excluded for clarity.)

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Edit</title>
    <link href="/Content/Site.css" rel="stylesheet" type="text/css" />
    <script src="/Scripts/jquery-1.5.1.min.js" type="text/javascript"></script>
    <script src="/Scripts/modernizr-1.7.min.js" type="text/javascript"></script>
</head>
<body>
    <div class="page">
        <header>
            <div id="title">
                <h1>MVC Movie App</h1>
            </div>
           ...
        </header>
        <section id="main">

<h2>Edit</h2>

<script src="/Scripts/jquery.validate.min.js" type="text/javascript"></script>
<script src="/Scripts/jquery.validate.unobtrusive.min.js" type="text/javascript"></script>

<form action="/Movies/Edit/4" method="post">    <fieldset>
        <legend>Movie</legend>

        <input data-val="true" data-val-number="The field ID must be a number."
    data-val-required="The ID field is required." id="ID" name="ID" type="hidden" value="4" />

        <div class="editor-label">
            <label for="Title">Title</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" id="Title" name="Title" type="text" value="Rio Bravo" />
            <span class="field-validation-valid" data-valmsg-for="Title" data-valmsg-replace="true"></span>
        </div>

        <div class="editor-label">
            <label for="ReleaseDate">ReleaseDate</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" data-val="true" data-val-required="The ReleaseDate field is
required."
    id="ReleaseDate" name="ReleaseDate" type="text" value="4/15/1959 12:00:00 AM" />
            <span class="field-validation-valid" data-valmsg-for="ReleaseDate" data-valmsg-replace="true">
</span>
        </div>

        <div class="editor-label">
            <label for="Genre">Genre</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" id="Genre" name="Genre" type="text" value="Western" />
            <span class="field-validation-valid" data-valmsg-for="Genre" data-valmsg-replace="true"></span>
```

```
        </div>

        <div class="editor-label">
            <label for="Price">Price</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" data-val="true" data-val-number="The field Price must be a
number."
    data-val-required="The Price field is required." id="Price" name="Price" type="text" value="9.99" />
            <span class="field-validation-valid" data-valmsg-for="Price" data-valmsg-replace="true"></span>
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
</form>
<div>
    <a href="/Movies">Back to List</a>
</div>

        </section>
        <footer>
        </footer>
    </div>
</body>
</html>
```

The `<input>` elements are in an HTML `<form>` element whose `action` attribute is set to post to the */Movies/Edit* URL. The form data will be posted to the server when the **Edit** button is clicked.

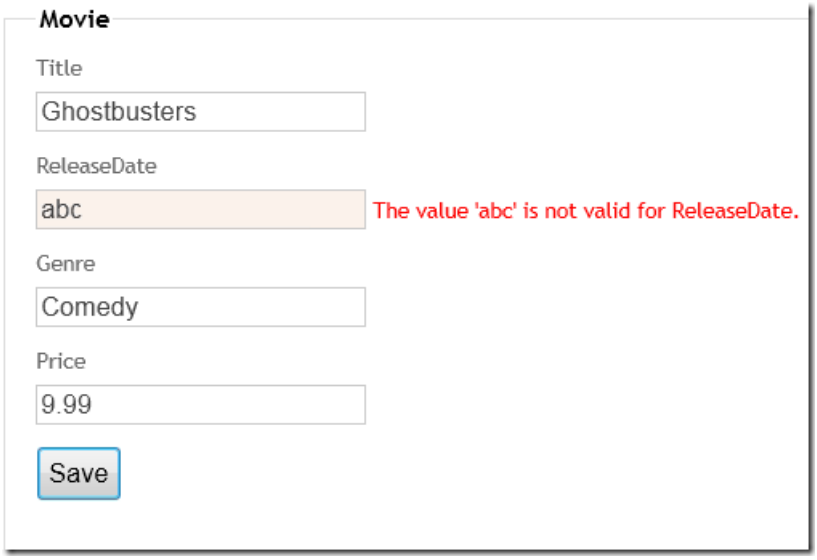
Processing the POST Request

The following listing shows the `HttpPost` version of the `Edit` action method.

```
[HttpPost]
public ActionResult Edit(Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Entry(movie).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The ASP.NET framework model binder takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` check in the code verifies that the data submitted in the form can be used to modify a `Movie` object. If the data is valid, the code saves the movie data to the `Movies` collection of the `MovieDBContext` instance. The code then saves the new movie data to the database by calling the `SaveChanges` method of `MovieDBContext`, which persists changes to the database. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which causes the updated movie to be displayed in the listing of movies.

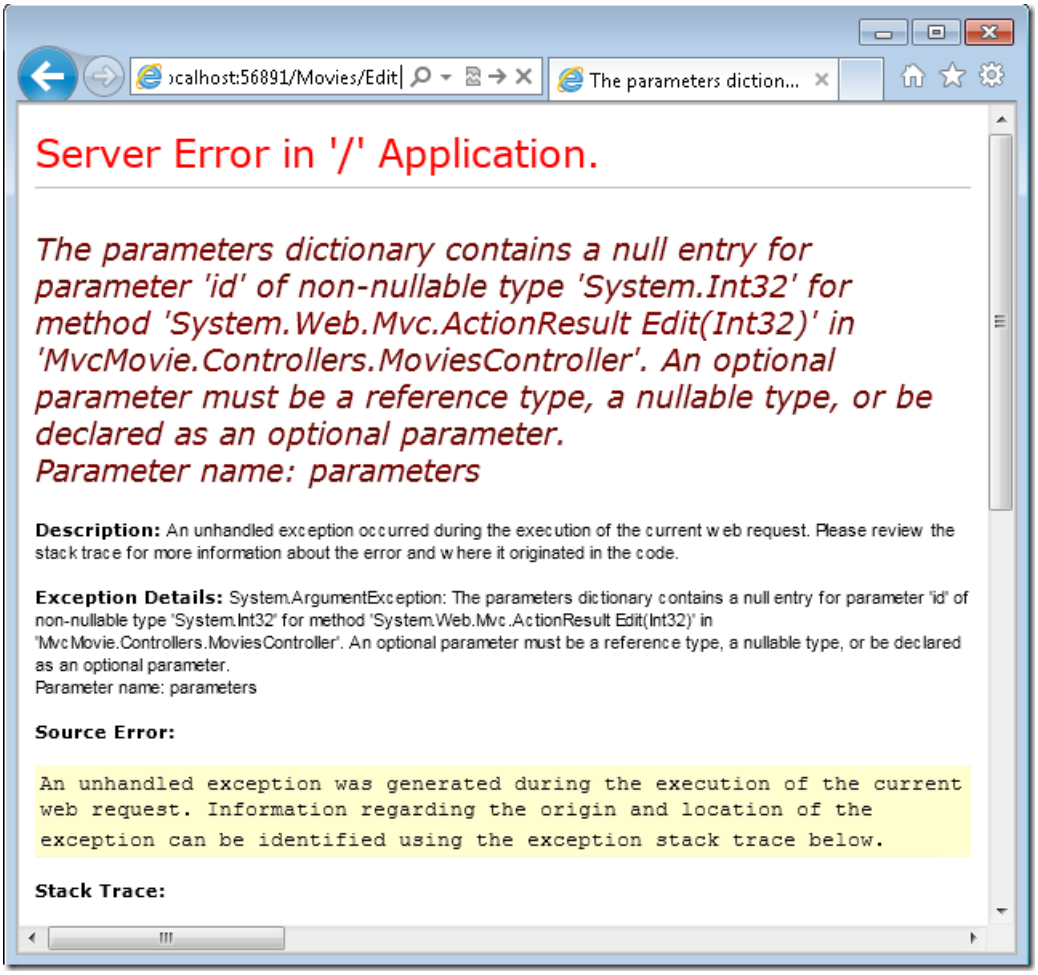
If the posted values aren't valid, they are redisplayed in the form. The `Html.ValidationMessageFor` helpers in the *Edit.cshtml* view template take care of displaying appropriate error messages.



Note about locales If you normally work with a locale other than English, see Supporting ASP.NET MVC 3 Validation with Non-English Locales.

Making the Edit Method More Robust

The `HttpGet` `Edit` method generated by the scaffolding system doesn't check that the ID that's passed to it is valid. If a user removes the ID segment from the URL (`http://localhost:xxxxx/Movies/Edit`), the following error is displayed:



A user could also pass an ID that doesn't exist in the database, such as `http://localhost:xxxxx/Movies/Edit/1234`. You can make two changes to the `HttpGet` `Edit` action method to address this limitation. First, change the `ID` parameter to have a default value of zero when an ID isn't explicitly passed. You can also check that the `Find`

method actually found a movie before returning the movie object to the view template. The updated `Edit` method is shown below.

```
public ActionResult Edit(int id = 0)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}
```

If no movie is found, the `HttpNotFound` method is called.

All the `HttpGet` methods follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the model to the view. The `Create` method passes an empty movie object to the Create view. All the methods that create, edit, delete, or otherwise modify data do so in the `HttpPost` overload of the method. Modifying data in an HTTP GET method is a security risk, as described in the blog post entry ASP.NET MVC Tip #46 – Don't use Delete Links because they create Security Holes. Modifying data in a GET method also violates HTTP best practices and the architectural REST pattern, which specifies that GET requests should not change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects.

Adding a Search Method and Search View

In this section you'll add a `SearchIndex` action method that lets you search movies by genre or name. This will be available using the */Movies/SearchIndex* URL. The request will display an HTML form that contains input elements that a user can fill in in order to search for a movie. When a user submits the form, the action method will get the search values posted by the user and use the values to search the database.

Displaying the SearchIndex Form

Start by adding a `SearchIndex` action method to the existing `MoviesController` class. The method will return a view that contains an HTML form. Here's the code:

```
public ActionResult SearchIndex(string searchString)
{
    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

The first line of the `SearchIndex` method creates the following LINQ query to select the movies:

```
var movies = from m in db.Movies
             select m;
```

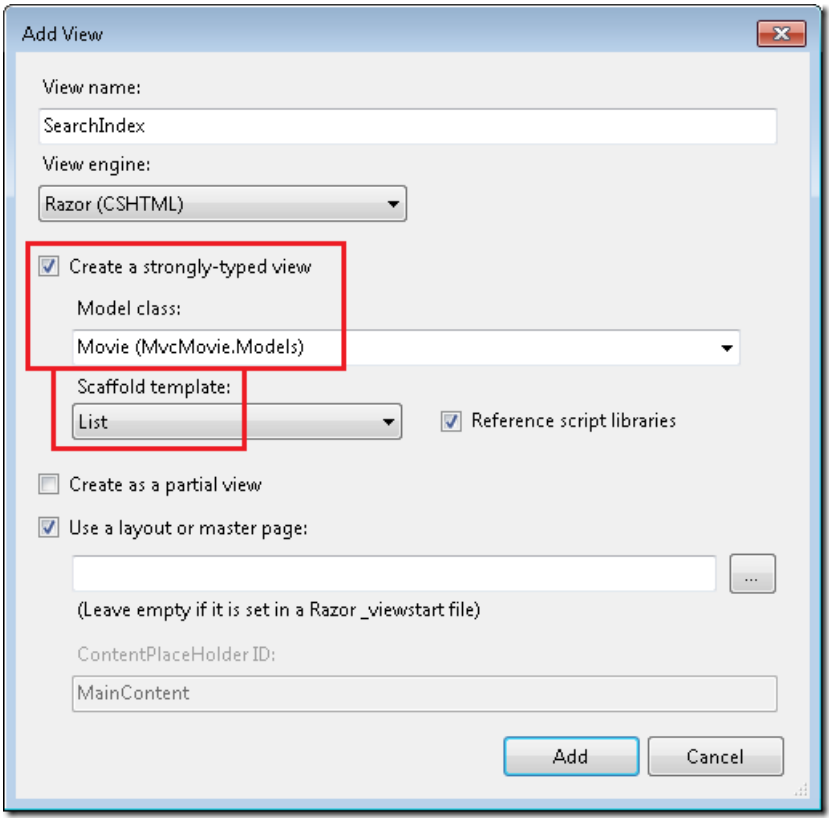
The query is defined at this point, but hasn't yet been run against the data store.

If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string, using the following code:

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

LINQ queries are not executed when they are defined or when they are modified by calling a method such as `Where` or `OrderBy`. Instead, query execution is deferred, which means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToList` method is called. In the `SearchIndex` sample, the query is executed in the SearchIndex view. For more information about deferred query execution, see Query Execution.

Now you can implement the `SearchIndex` view that will display the form to the user. Right-click inside the `SearchIndex` method and then click **Add View**. In the **Add View** dialog box, specify that you're going to pass a `Movie` object to the view template as its model class. In the **Scaffold template** list, choose **List**, then click **Add**.



When you click the **Add** button, the *Views\Movies\SearchIndex.cshtml* view template is created. Because you selected **List** in the **Scaffold template** list, Visual Web Developer automatically generated (scaffolded) some default content in the view. The scaffolding created an HTML form. It examined the `Movie` class and created code to render `<label>` elements for each property of the class. The listing below shows the Create view that was generated:

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewBag.Title = "SearchIndex";
}

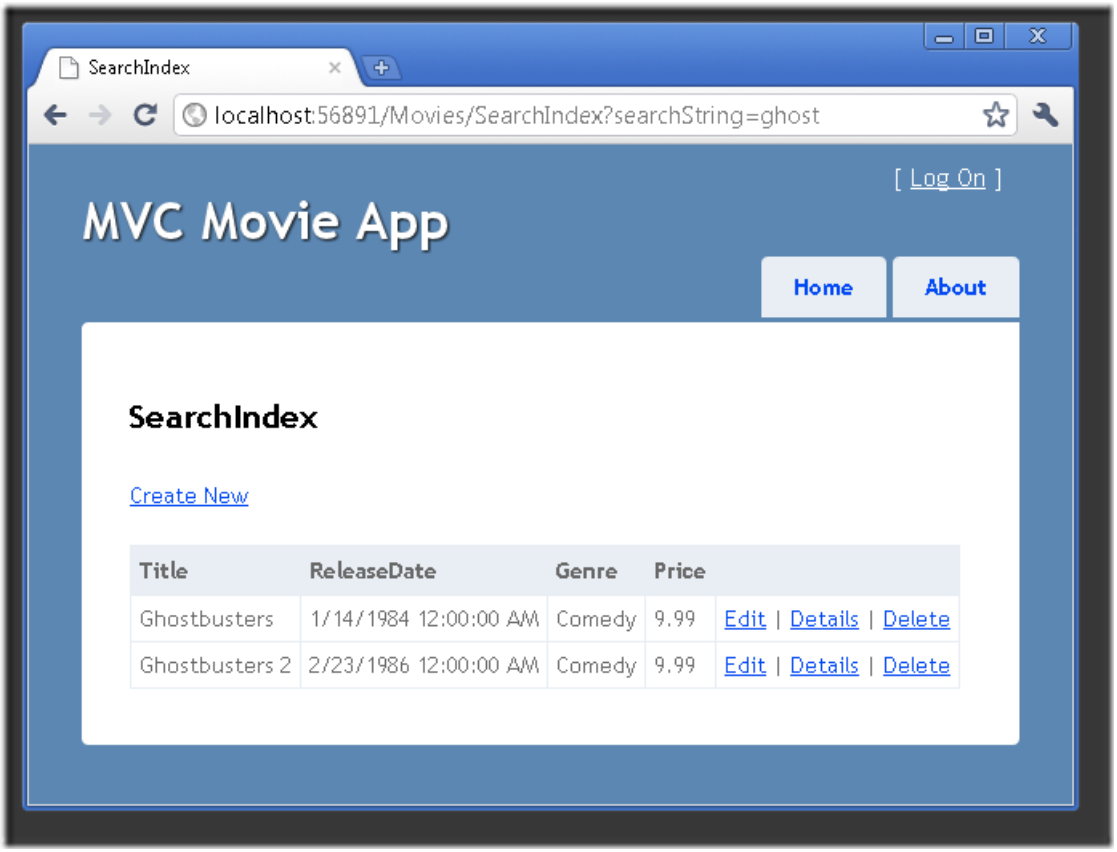
<h2>SearchIndex</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            Title
        </th>
        <th>
            ReleaseDate
        </th>
        <th>
            Genre
        </th>
        <th>
            Price
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
    </tr>
}

</table>
```

Run the application and navigate to */Movies/SearchIndex*. Append a query string such as `?searchString=ghost` to the URL. The filtered movies are displayed.



If you change the signature of the `SearchIndex` method to have a parameter named `id`, the `id` parameter will match the `{id}` placeholder for the default routes set in the *Global.asax* file.

```
{controller}/{action}/{id}
```

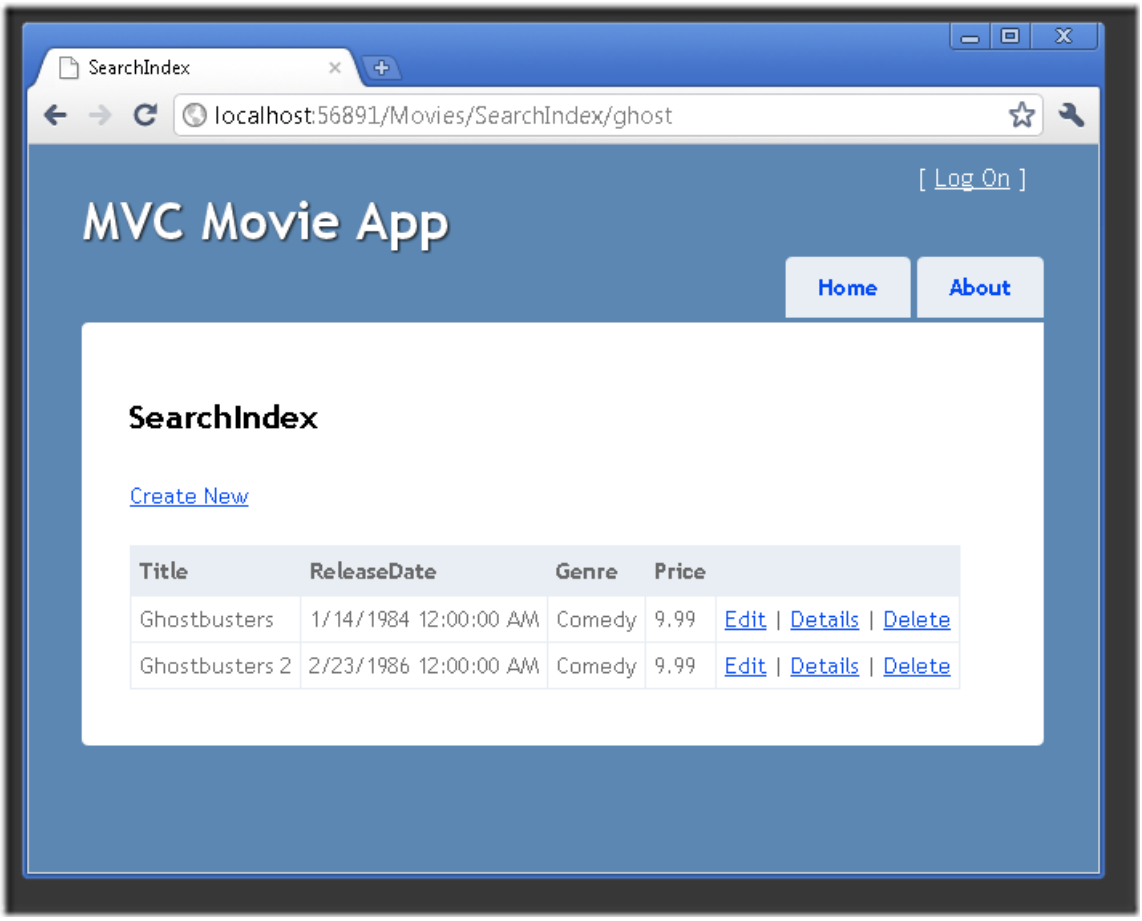
The modified `SearchIndex` method would look as follows:

```
public ActionResult SearchIndex(string id)
{
    string searchString = id;
    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

You can now pass the search title as route data (a URL segment) instead of as a query string value.



However, you can't expect users to modify the URL every time they want to search for a movie. So now you you'll add UI to help them filter movies. If you changed the signature of the `SearchIndex` method to test how to pass the route-bound ID parameter, change it back so that your `SearchIndex` method takes a string parameter named `searchString`:

```
public ActionResult SearchIndex(string searchString)
{
     var movies = from m in db.Movies
                  select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

Open the *Views\Movies\SearchIndex.cshtml* file, and just after `@Html.ActionLink("Create New", "Create")`, add the following:

```
@using (Html.BeginForm()){
         <p> Title: @Html.TextBox("SearchString")
         <input type="submit" value="Filter" /></p>
        }
```

The following example shows a portion of the *Views\Movies\SearchIndex.cshtml* file with the added filtering markup.

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewBag.Title = "SearchIndex";
}

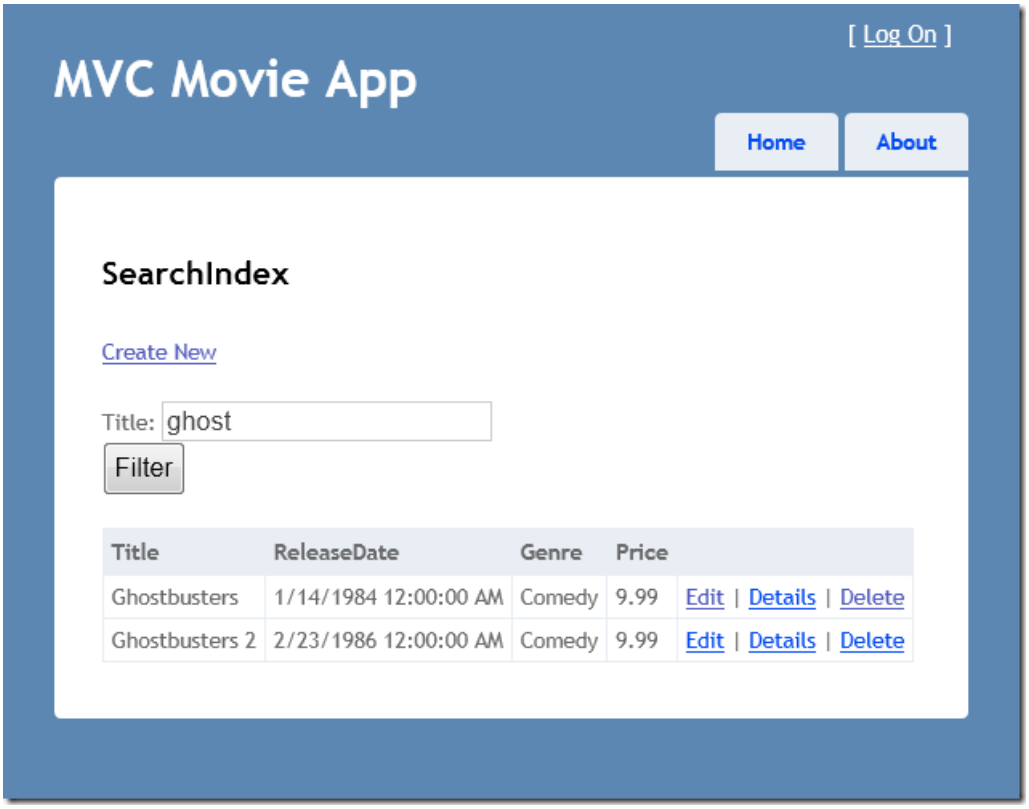
<h2>SearchIndex</h2>

<p>
    @Html.ActionLink("Create New", "Create")

     @using (Html.BeginForm()){
         <p> Title: @Html.TextBox("SearchString") <br />
         <input type="submit" value="Filter" /></p>
        }
</p>
```

The `Html.BeginForm` helper creates an opening `<form>` tag. The `Html.BeginForm` helper causes the form to post to itself when the user submits the form by clicking the **Filter** button.

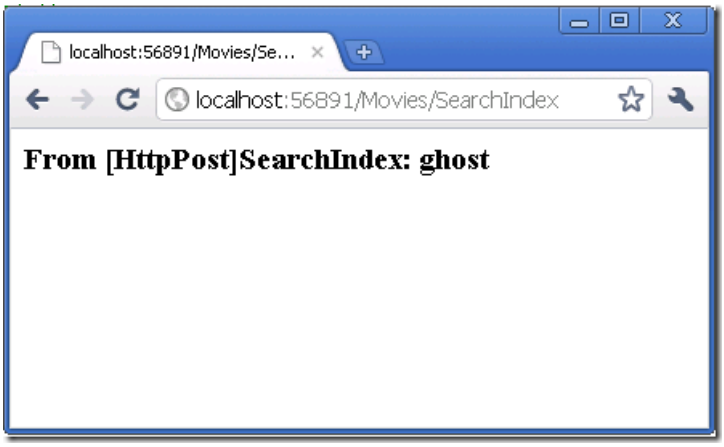
Run the application and try searching for a movie.



There's no `HttpPost` overload of the `SearchIndex` method. You don't need it, because the method isn't changing the state of the application, just filtering data.

You could add the following `HttpPost SearchIndex` method. In that case, the action invoker would match the `HttpPost SearchIndex` method, and the `HttpPost SearchIndex` method would run as shown in the image below.

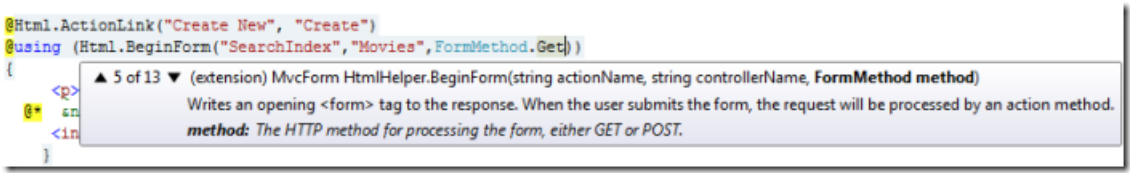
```
[HttpPost]
public string SearchIndex(FormCollection fc, string searchString)
{
    return "<h3> From [HttpPost]SearchIndex: " + searchString + "</h3>";
}
```



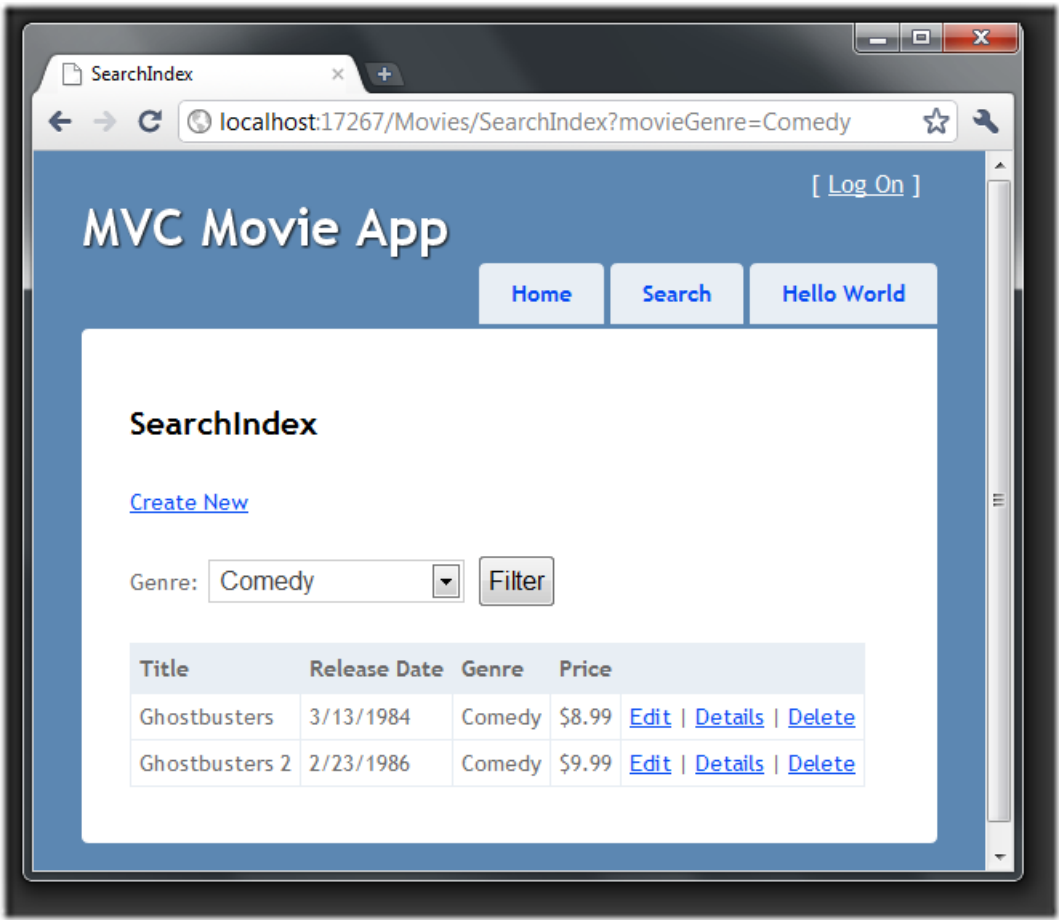
However, even if you add this `HttpPost` version of the `SearchIndex` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (localhost:xxxxx/Movies/SearchIndex) -- there's no search information in the URL itself. Right now, the search string information is sent to the server as a form field value. This means you can't capture that search information to bookmark or send to friends in a URL.

The solution is to use an overload of `BeginForm` that specifies that the POST request should add the search information to the URL and that is should be routed to the HttpGet version of the `SearchIndex` method. Replace the existing parameterless `BeginForm` method with the following:

```
@using (Html.BeginForm("SearchIndex","Movies",FormMethod.Get))
```



Now when you submit a search, the URL contains a search query string. Searching will also go to the `HttpGet SearchIndex` action method, even if you have a `HttpPost SearchIndex` method.



Adding Search by Genre

If you added the `HttpPost` version of the `SearchIndex` method, delete it now.

Next, you'll add a feature to let users search for movies by genre. Replace the `SearchIndex` method with the following code:

```
public ActionResult SearchIndex(string movieGenre, string searchString)
{
    var GenreLst = new List<string>();

    var GenreQry = from d in db.Movies
                   orderby d.Genre
                   select d.Genre;
    GenreLst.AddRange(GenreQry.Distinct());
    ViewBag.movieGenre = new SelectList(GenreLst);

    var movies = from m in db.Movies
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (string.IsNullOrEmpty(movieGenre))
        return View(movies);
    else
    {
        return View(movies.Where(x => x.Genre == movieGenre));
    }
}
```

This version of the `SearchIndex` method takes an additional parameter, namely `movieGenre`. The first few lines of

code create a `List` object to hold movie genres from the database.

The following code is a LINQ query that retrieves all the genres from the database.

```
var GenreQry = from d in db.Movies
               orderby d.Genre
               select d.Genre;
```

The code uses the `AddRange` method of the generic `List` collection to add all the distinct genres to the list. (Without the `Distinct` modifier, duplicate genres would be added — for example, comedy would be added twice in our sample). The code then stores the list of genres in the `ViewBag` object.

The following code shows how to check the `movieGenre` parameter. If it's not empty the code further constrains the movies query to limit the selected movies to the specified genre.

```
if (string.IsNullOrEmpty(movieGenre))
        return View(movies);
else
{
    return View(movies.Where(x => x.Genre == movieGenre));
}
```

Adding Markup to the SearchIndex View to Support Search by Genre

Add an `Html.DropDownList` helper to the *Views\Movies\SearchIndex.cshtml* file, just before the `TextBox` helper. The completed markup is shown below:

```
<p>
    @Html.ActionLink("Create New", "Create")
    @using (Html.BeginForm()){
         <p>Genre: @Html.DropDownList("movieGenre", "All")
           Title: @Html.TextBox("SearchString")
         <input type="submit" value="Filter" /></p>
        }
</p>
```

Run the application and browse to */Movies/SearchIndex*. Try a search by genre, by movie name, and by both criteria.

In this section you examined the CRUD action methods and views generated by the framework. You created a search action method and view that let users search by movie title and genre. In the next section, you'll look at how to add a property to the `Movie` model and how to add an initializer that will automatically create a test database.

PREVIOUS NEXT

Adding a New Field to the Movie Model and Table (C#)

2/20/2020 • 7 minutes to read • Edit Online

by Rick Anderson

> **NOTE**
>
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

A Visual Web Developer project with C# source code is available to accompany this topic. Download the C# version. If you prefer Visual Basic, switch to the Visual Basic version of this tutorial.

In this section you'll make some changes to the model classes and learn how you can update the database schema to match the model changes.

Adding a Rating Property to the Movie Model

Start by adding a new `Rating` property to the existing `Movie` class. Open the *Movie.cs* file and add the `Rating` property like this one:

```
public string Rating { get; set; }
```

The complete `Movie` class now looks like the following code:

```
public class Movie
{
    public int      ID          { get; set; }
    public string   Title       { get; set; }
    public DateTime ReleaseDate { get; set; }
    public string   Genre       { get; set; }
    public decimal  Price       { get; set; }
    public string   Rating      { get; set; }
}
```

Recompile the application using the **Debug** >**Build Movie** menu command.

Now that you've updated the `Model` class, you also need to update the *\Views\Movies\Index.cshtml* and *\Views\Movies\Create.cshtml* view templates in order to support the new `Rating` property.

Open the *\Views\Movies\Index.cshtml* file and add a `<th>Rating</th>` column heading just after the **Price** column. Then add a `<td>` column near the end of the template to render the `@item.Rating` value. Below is what the updated *Index.cshtml* view template looks like:

```
<table>
    <tr>
        <th></th>
        <th>Title</th>
        <th>Release Date</th>
        <th>Genre</th>
        <th>Price</th>
        <th>Rating</th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Rating   )
        </td>
        <td>
            @Html.ActionLink("Edit Me", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
    </tr>
}
</table>
```

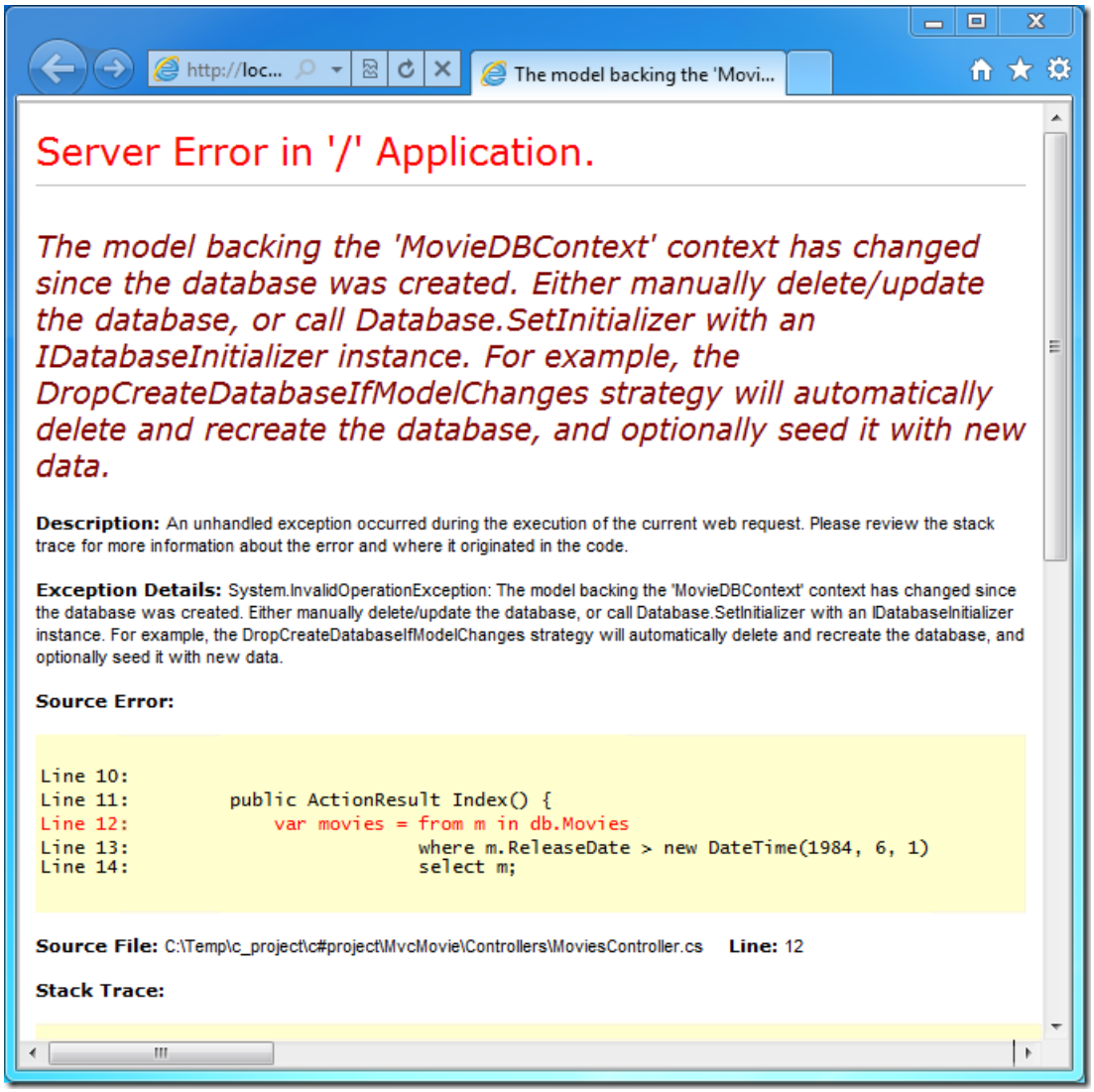
Next, open the *\Views\Movies\Create.cshtml* file and add the following markup near the end of the form. This renders a text box so that you can specify a rating when a new movie is created.

```
<div class="editor-label">
    @Html.LabelFor(model => model.Rating)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Rating)
    @Html.ValidationMessageFor(model => model.Rating)
</div>
```

Managing Model and Database Schema Differences

You've now updated the application code to support the new `Rating` property.

Now run the application and navigate to the */Movies* URL. When you do this, though, you'll see the following error:



You're seeing this error because the updated `Movie` model class in the application is now different than the schema of the `Movie` table of the existing database. (There's no `Rating` column in the database table.)

By default, when you use Entity Framework Code First to automatically create a database, as you did earlier in this tutorial, Code First adds a table to the database to help track whether the schema of the database is in sync with the model classes it was generated from. If they aren't in sync, the Entity Framework throws an error. This makes it easier to track down issues at development time that you might otherwise only find (by obscure errors) at run time. The sync-checking feature is what causes the error message to be displayed that you just saw.

There are two approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient when doing active development on a test database, because it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you *don't* want to use this approach on a production database!
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.

For this tutorial, we'll use the first approach — you'll have the Entity Framework Code First automatically re-create the database anytime the model changes.

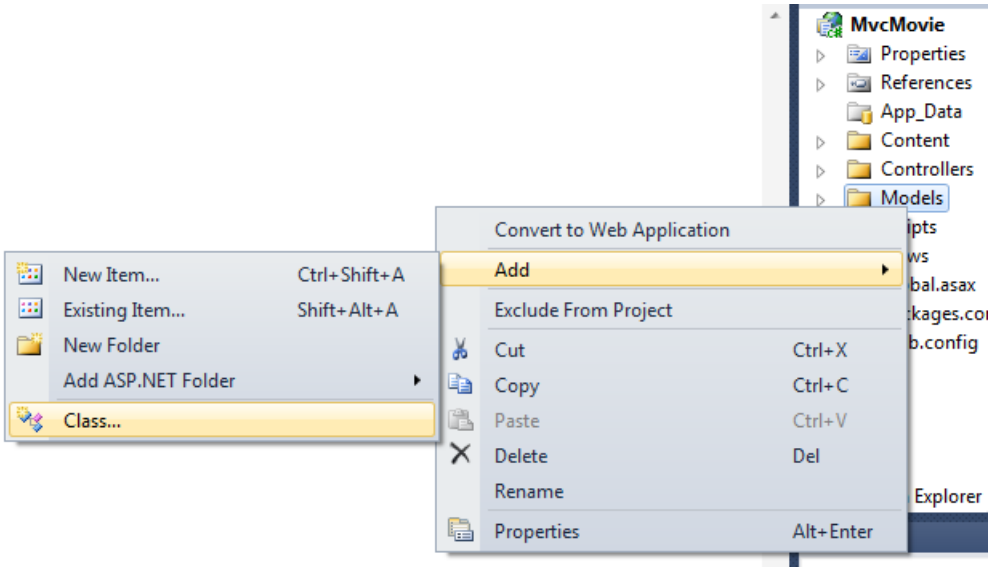
Automatically Re-Creating the Database on Model Changes

Let's update the application so that Code First automatically drops and re-creates the database anytime you

change the model for the application.

> **NOTE**
>
> **Warning** You should enable this approach of automatically dropping and re-creating the database only when you're using a development or test database, and *never* on a production database that contains real data. Using it on a production server can lead to data loss.

In **Solution Explorer**, right click the *Models* folder, select **Add**, and then select **Class**.



Name the class "MovieInitializer". Update the `MovieInitializer` class to contain the following code:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;

namespace MvcMovie.Models {
    public class MovieInitializer : DropCreateDatabaseIfModelChanges<MovieDBContext> {
        protected override void Seed(MovieDBContext context) {
            var movies = new List<Movie> {

                 new Movie { Title = "When Harry Met Sally",
                             ReleaseDate=DateTime.Parse("1989-1-11"),
                             Genre="Romantic Comedy",
                             Rating="R",
                             Price=7.99M},

                     new Movie { Title = "Ghostbusters ",
                             ReleaseDate=DateTime.Parse("1984-3-13"),
                             Genre="Comedy",
                              Rating="R",
                             Price=8.99M},

                 new Movie { Title = "Ghostbusters 2",
                             ReleaseDate=DateTime.Parse("1986-2-23"),
                             Genre="Comedy",
                             Rating="R",
                             Price=9.99M},

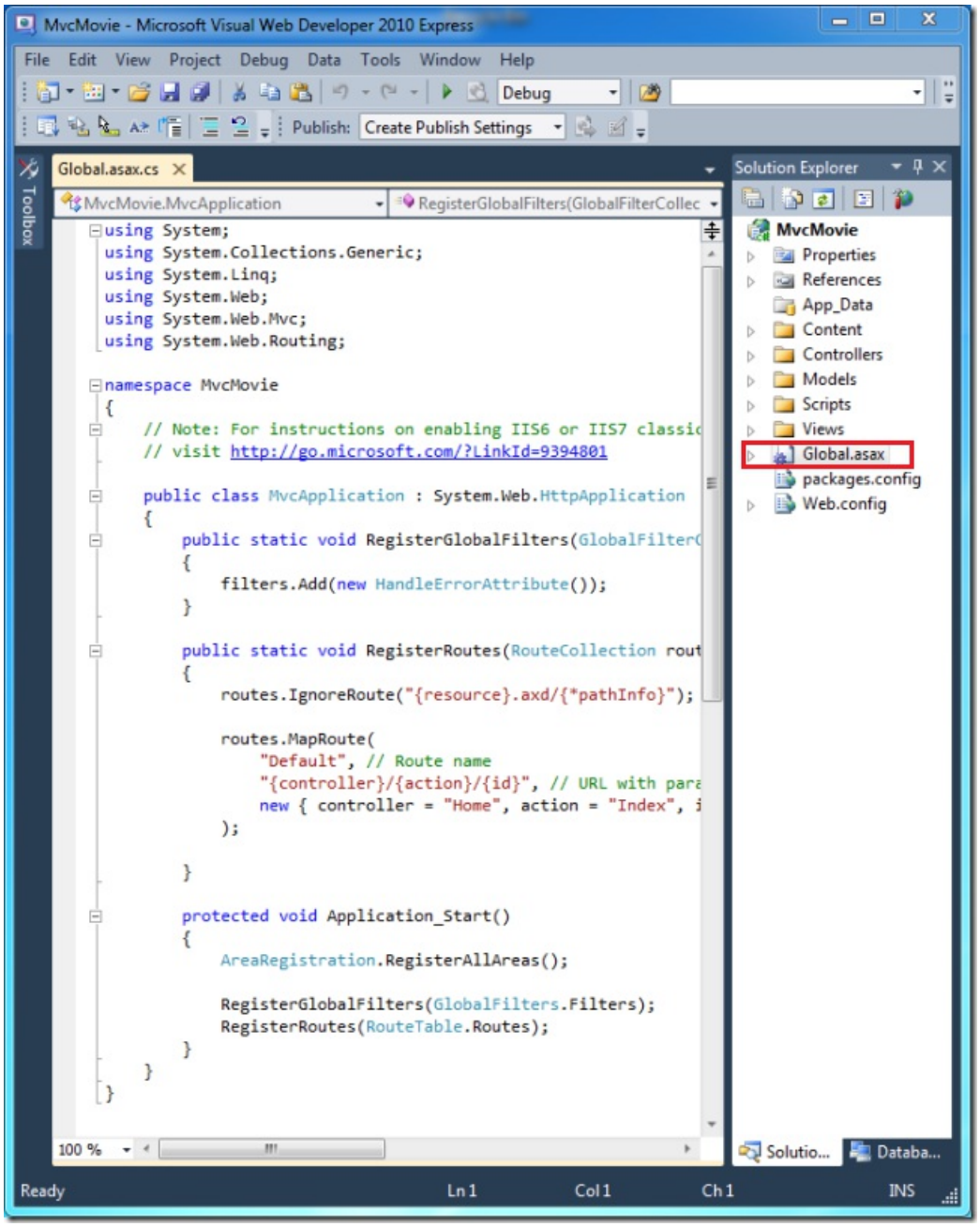
               new Movie { Title = "Rio Bravo",
                             ReleaseDate=DateTime.Parse("1959-4-15"),
                             Genre="Western",
                             Rating="R",
                             Price=3.99M},
             };

            movies.ForEach(d => context.Movies.Add(d));
        }
    }
}
```

The `MovieInitializer` class specifies that the database used by the model should be dropped and automatically re-created if the model classes ever change. The code includes a `Seed` method to specify some default data to automatically add to the database any time it's created (or re-created). This provides a useful way to populate the database with some sample data, without requiring you to manually populate it each time you make a model change.

Now that you've defined the `MovieInitializer` class, you'll want to wire it up so that each time the application runs, it checks whether the model classes are different from the schema in the database. If they are, you can run the initializer to re-create the database to match the model and then populate the database with the sample data.

Open the *Global.asax* file that's at the root of the `MvcMovies` project:



The *Global.asax* file contains the class that defines the entire application for the project, and contains an `Application_Start` event handler that runs when the application first starts.

Let's add two using statements to the top of the file. The first references the Entity Framework namespace, and the second references the namespace where our `MovieInitializer` class lives:

```
using System.Data.Entity;            // Database.SetInitialize
using MvcMovie.Models;              // MovieInitializer
```

Then find the `Application_Start` method and add a call to `Database.SetInitializer` at the beginning of the method, as shown below:

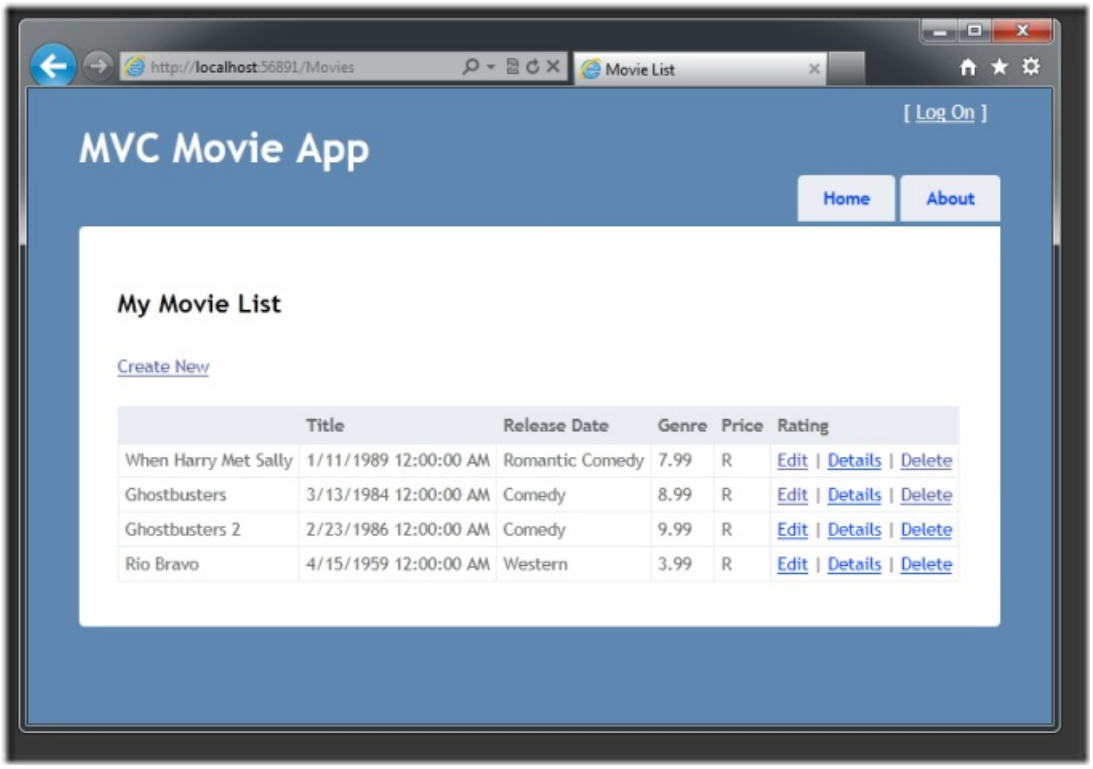
```
protected void Application_Start()
{
    Database.SetInitializer<MovieDBContext>(new MovieInitializer());

    AreaRegistration.RegisterAllAreas();
    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
}
```

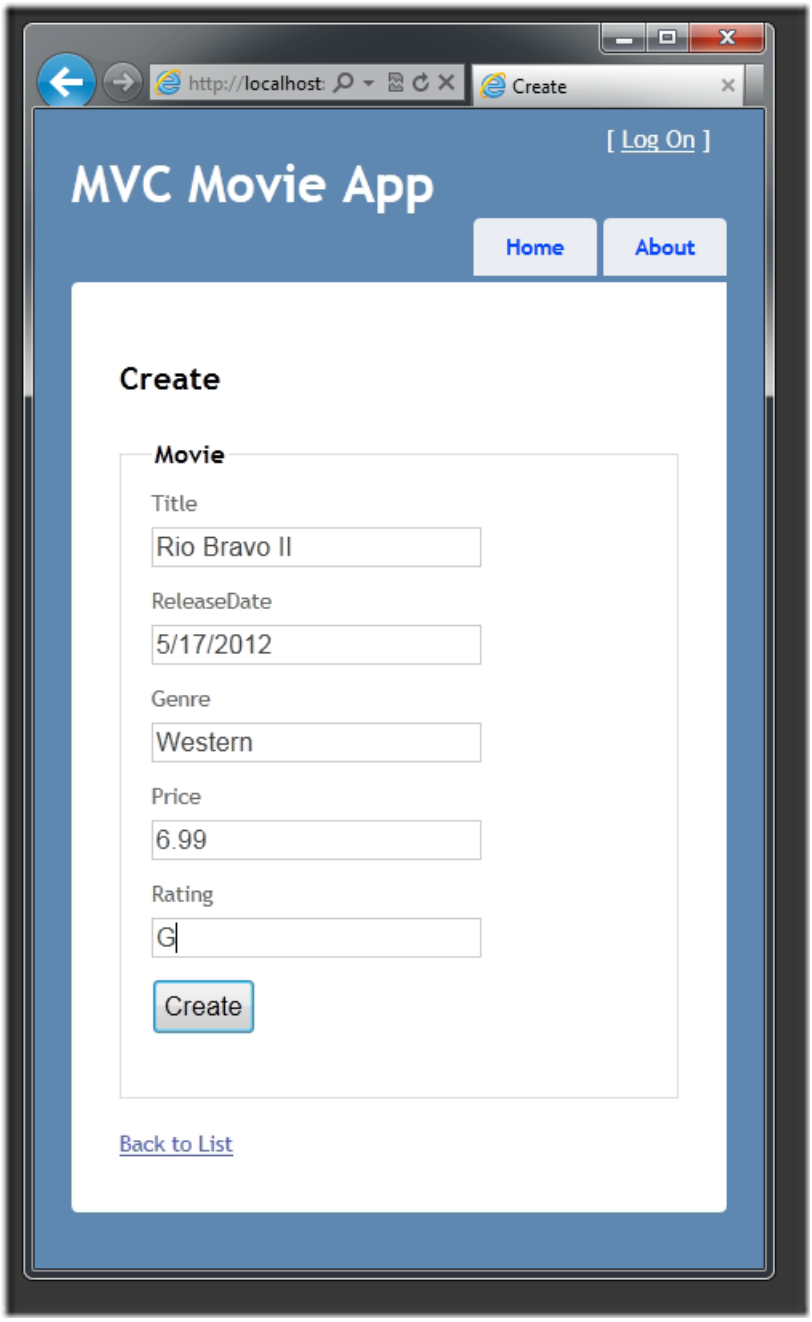
The `Database.SetInitializer` statement you just added indicates that the database used by the `MovieDBContext` instance should be automatically deleted and re-created if the schema and the database don't match. And as you saw, it will also populate the database with the sample data that's specified in the `MovieInitializer` class.

Close the *Global.asax* file.

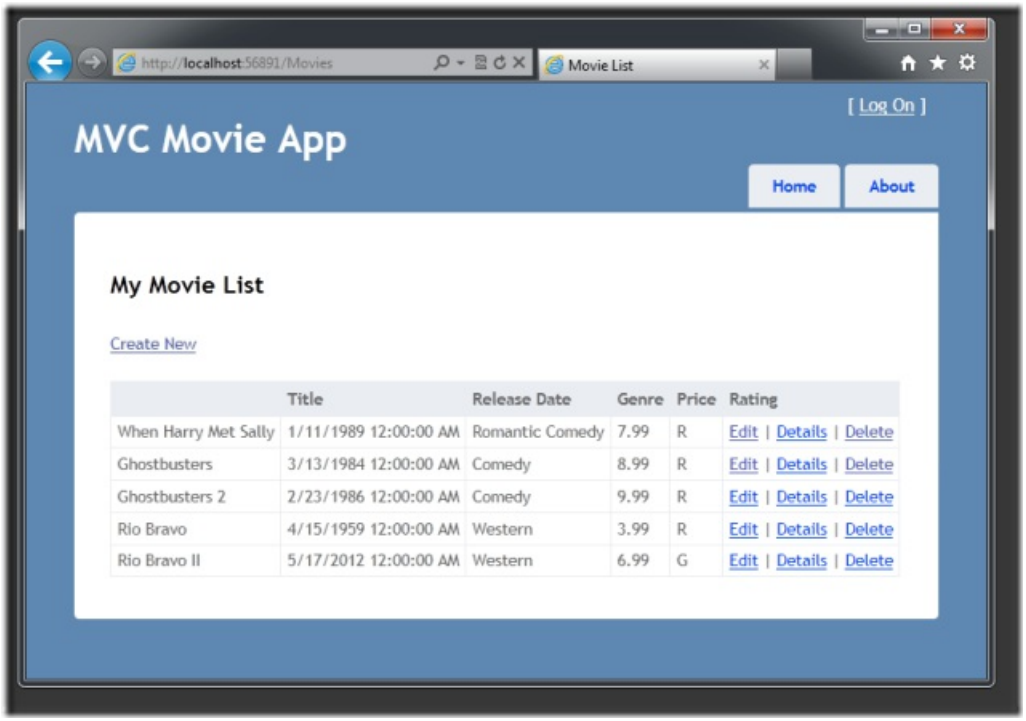
Re-run the application and navigate to the */Movies* URL. When the application starts, it detects that the model structure no longer matches the database schema. It automatically re-creates the database to match the new model structure and populates the database with the sample movies:



Click the **Create New** link to add a new movie. Note that you can add a rating.



Click **Create**. The new movie, including the rating, now shows up in the movies listing:



In this section you saw how you can modify model objects and keep the database in sync with the changes. You also learned a way to populate a newly created database with sample data so you can try out scenarios. Next, let's look at how you can add richer validation logic to the model classes and enable some business rules to be enforced.

PREVIOUS NEXT

Adding Validation to the Model (C#)

2/20/2020 • 8 minutes to read • Edit Online

by Rick Anderson

> **NOTE**
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

A Visual Web Developer project with C# source code is available to accompany this topic. Download the C# version. If you prefer Visual Basic, switch to the Visual Basic version of this tutorial.

In this section you'll add validation logic to the `Movie` model, and you'll ensure that the validation rules are enforced any time a user attempts to create or edit a movie using the application.

Keeping Things DRY

One of the core design tenets of ASP.NET MVC is DRY ("Don't Repeat Yourself"). ASP.NET MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an application. This reduces the amount of code you need to write and makes the code you do write much easier to maintain.

The validation support provided by ASP.NET MVC and Entity Framework Code First is a great example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and then those rules are enforced everywhere in the application.

Let's look at how you can take advantage of this validation support in the movie application.

Adding Validation Rules to the Movie Model

You'll begin by adding some validation logic to the `Movie` class.

Open the *Movie.cs* file. Add a `using` statement at the top of the file that references the `System.ComponentModel.DataAnnotations` namespace:

```
using System.ComponentModel.DataAnnotations;
```

The namespace is part of the .NET Framework. It provides a built-in set of validation attributes that you can apply declaratively to any class or property.

Now update the `Movie` class to take advantage of the built-in `Required`, `StringLength`, and `Range` validation attributes. Use the following code as an example of where to apply the attributes.

```
public class Movie
{
    public int ID { get; set; }

    [Required(ErrorMessage = "Title is required")]
    public string Title { get; set; }

    [Required(ErrorMessage = "Date is required")]
    public DateTime ReleaseDate { get; set; }

    [Required(ErrorMessage = "Genre must be specified")]
    public string Genre { get; set; }

    [Range(1, 100, ErrorMessage = "Price must be between $1 and $100")]
    public decimal Price { get; set; }

    [StringLength(5)]
    public string Rating { get; set; }
}
```

The validation attributes specify behavior that you want to enforce on the model properties they are applied to. The `Required` attribute indicates that a property must have a value; in this sample, a movie has to have values for the `Title`, `ReleaseDate`, `Genre`, and `Price` properties in order to be valid. The `Range` attribute constrains a value to within a specified range. The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length.

Code First ensures that the validation rules you specify on a model class are enforced before the application saves changes in the database. For example, the code below will throw an exception when the `SaveChanges` method is called, because several required `Movie` property values are missing and the price is zero (which is out of the valid range).

```
MovieDBContext db = new MovieDBContext();

Movie movie = new Movie();
movie.Title = "Gone with the Wind";
movie.Price = 0.0M;

db.Movies.Add(movie);
db.SaveChanges();        // <= Will throw validation exception
```

Having validation rules automatically enforced by the .NET Framework helps make your application more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Here's a complete code listing for the updated *Movie.cs* file:

```
using System;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }

        [Required(ErrorMessage = "Title is required")]
        public string Title { get; set; }

        public DateTime ReleaseDate { get; set; }

        [Required(ErrorMessage = "Genre must be specified")]
        public string Genre { get; set; }

        [Range(1, 100, ErrorMessage = "Price must be between $1 and $100")]
        public decimal Price { get; set; }

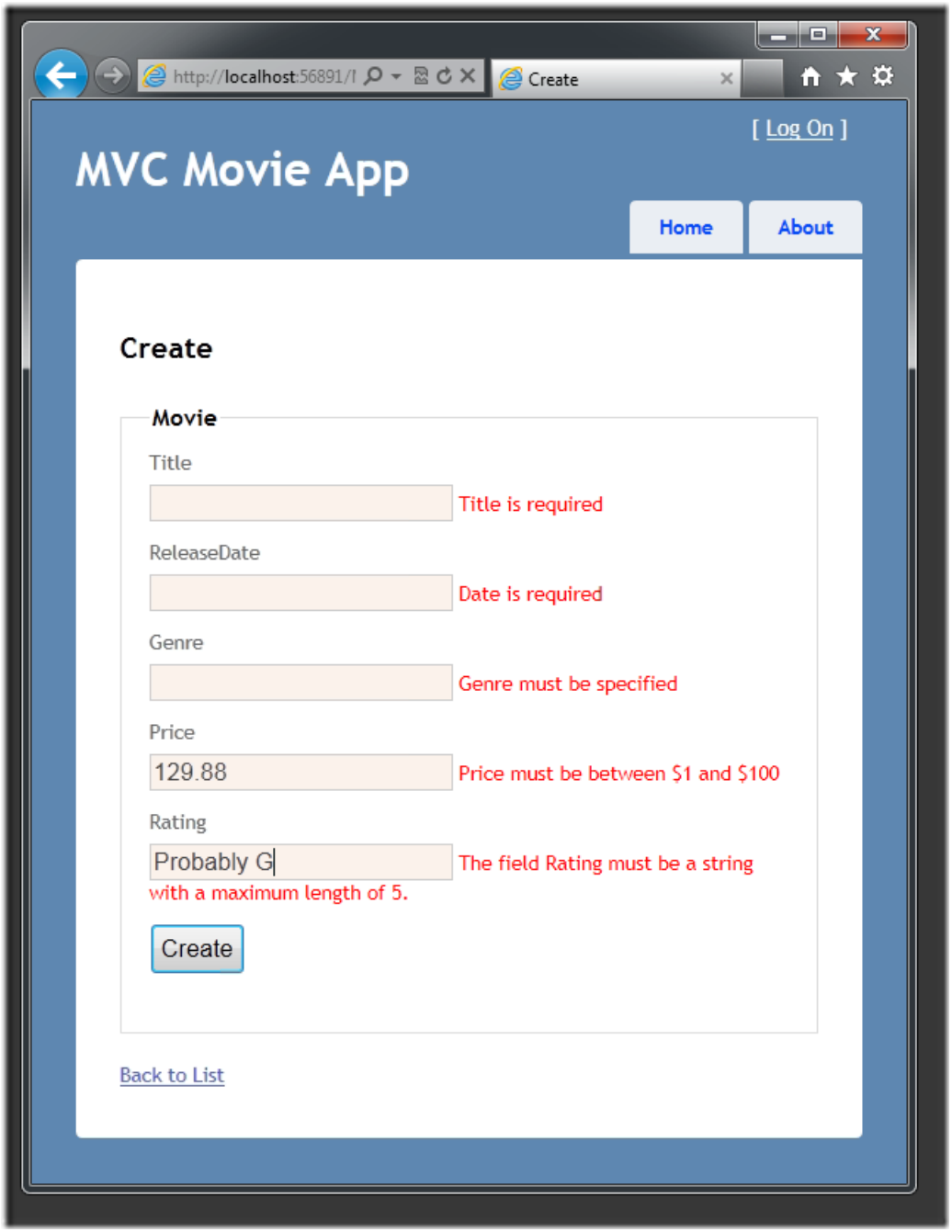
        [StringLength(5)]
        public string Rating { get; set; }
    }

    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

Validation Error UI in ASP.NET MVC

Re-run the application and navigate to the */Movies* URL.

Click the **Create Movie** link to add a new movie. Fill out the form with some invalid values and then click the **Create** button.



Notice how the form has automatically used a background color to highlight the text boxes that contain invalid data and has emitted an appropriate validation error message next to each one. The error messages match the error strings you specified when you annotated the `Movie` class. The errors are enforced both client-side (using JavaScript) and server-side (in case a user has JavaScript disabled).

A real benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the *Create.cshtml* view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified using attributes on the `Movie` model class.

How Validation Occurs in the Create View and Create Action Method

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The next listing shows what the `Create` methods in the `MovieController` class look like. They're unchanged from how you created them earlier in this tutorial.

```
//
// GET: /Movies/Create

public ActionResult Create()
{
    return View();
}

//
// POST: /Movies/Create

[HttpPost]
public ActionResult Create(Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Movies.Add(movie);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(movie);
}
```

The first action method displays the initial Create form. The second handles the form post. The second `Create` method calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method redisplays the form. If there are no errors, the method saves the new movie in the database.

Below is the *Create.cshtml* view template that you scaffolded earlier in the tutorial. It's used by the action methods shown above both to display the initial form and to redisplay it in the event of an error.

```
@model MvcMovie.Models.Movie
@{
    ViewBag.Title = "Create";
}
<h2>
    Create</h2>
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>
@using (Html.BeginForm())
{
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Movie</legend>
        <div class="editor-label">
            @Html.LabelFor(model => model.Title)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.ReleaseDate)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.ReleaseDate)
            @Html.ValidationMessageFor(model => model.ReleaseDate)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.Genre)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Genre)
            @Html.ValidationMessageFor(model => model.Genre)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.Price)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Price)
            @Html.ValidationMessageFor(model => model.Price)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.Rating)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Rating)
            @Html.ValidationMessageFor(model => model.Rating)
        </div>
        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Notice how the code uses an `Html.EditorFor` helper to output the `<input>` element for each `Movie` property. Next to this helper is a call to the `Html.ValidationMessageFor` helper method. These two helper methods work with the model object that's passed by the controller to the view (in this case, a `Movie` object). They automatically look for validation attributes specified on the model and display error messages as appropriate.

What's really nice about this approach is that neither the controller nor the Create view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation

rules and the error strings are specified only in the `Movie` class.

If you want to change the validation logic later, you can do so in exactly one place. You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the DRY principle.

Adding Formatting to the Movie Model

Open the *Movie.cs* file. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. You'll apply the `DisplayFormat` attribute and a `DataType` enumeration value to the release date and to the price fields. The following code shows the `ReleaseDate` and `Price` properties with the appropriate `DisplayFormat` attribute.

```
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

Alternatively, you could explicitly set a `DataFormatString` value. The following code shows the release date property with a date format string (namely, "d"). You'd use this to specify that you don't want to time as part of the release date.

```
[DisplayFormat(DataFormatString = "{0:d}")]
public DateTime ReleaseDate { get; set; }
```

The following code formats the `Price` property as currency.

```
[DisplayFormat(DataFormatString = "{0:c}")]
public decimal Price { get; set; }
```

The complete `Movie` class is shown below.

```
public class Movie
{
    public int ID { get; set; }

    [Required(ErrorMessage = "Title is required")]
    public string Title { get; set; }

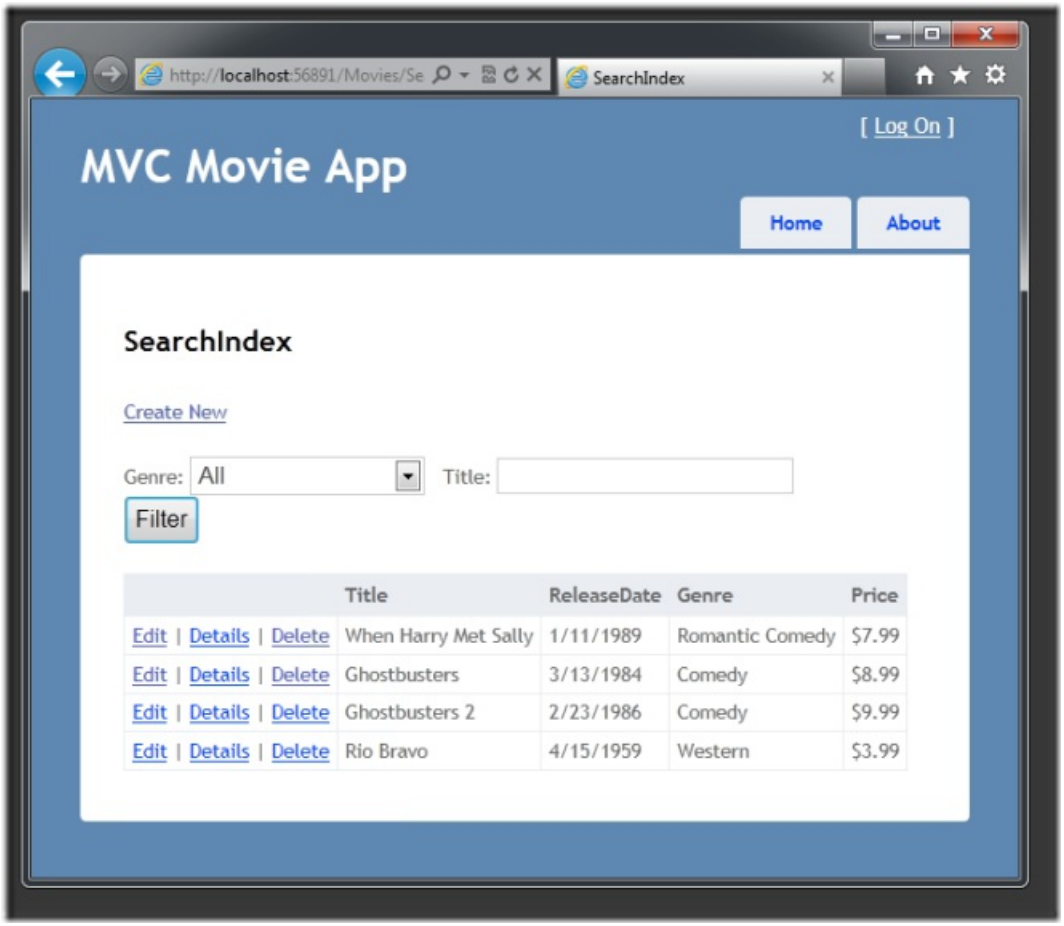
    [DisplayFormat(DataFormatString = "{0:d}")]
    public DateTime ReleaseDate { get; set; }

    [Required(ErrorMessage = "Genre must be specified")]
    public string Genre { get; set; }

    [Range(1, 100, ErrorMessage = "Price must be between $1 and $100")]
    [DisplayFormat(DataFormatString = "{0:c}")]
    public decimal Price { get; set; }

    [StringLength(5)]
    public string Rating { get; set; }
}
```

Run the application and browse to the `Movies` controller.



In the next part of the series, we'll review the application and make some improvements to the automatically generated `Details` and `Delete` methods.

PREVIOUS NEXT

Improving the Details and Delete Methods (C#)

2/20/2020 • 5 minutes to read • Edit Online

by Rick Anderson

> **NOTE**
>
> An updated version of this tutorial is available here that uses ASP.NET MVC 5 and Visual Studio 2013. It's more secure, much simpler to follow and demonstrates more features.

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

A Visual Web Developer project with C# source code is available to accompany this topic. Download the C# version. If you prefer Visual Basic, switch to the Visual Basic version of this tutorial.

In this part of the tutorial, you'll make some improvements to the automatically generated `Details` and `Delete` methods. These changes aren't required, but with just a few small bits of code, you can easily enhance the application.

Improving the Details and Delete Methods

When you scaffolded the `Movie` controller, ASP.NET MVC generated code that worked great, but that can be made more robust with just a few small changes.

Open the `Movie` controller and modify the `Details` method by returning `HttpNotFound` when a movie isn't found. You should also modify the `Details` method to set a default value for the ID that's passed to it. (You made similar changes to the `Edit` method in part 6 of this tutorial.) However, you must change the return type of the `Details` method from `ViewResult` to `ActionResult`, because the `HttpNotFound` method doesn't return a `ViewResult` object. The following example shows the modified `Details` method.

```
public ActionResult Details(int id = 0)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}
```

Code First makes it easy to search for data using the `Find` method. An important security feature that we built into the method is that the code verifies that the `Find` method has found a movie before the code tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from `http://localhost:xxxx/Movies/Details/1` to something like `http://localhost:xxxx/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you don't the check for a null movie, this could result in a database error.

Similarly, change the `Delete` and `DeleteConfirmed` methods to specify a default value for the ID parameter and to return `HttpNotFound` when a movie isn't found. The updated `Delete` methods in the `Movie` controller are shown below.

```
// GET: /Movies/Delete/5

public ActionResult Delete(int id = 0)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}

//
// POST: /Movies/Delete/5

[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id = 0)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    db.Movies.Remove(movie);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

Note that the `Delete` method doesn't delete the data. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole. For more information about this, see Stephen Walther's blog entry ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes.

The `HttpPost` method that deletes the data is named `DeleteConfirmed` to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

```
// GET: /Movies/Delete/5
public ActionResult Delete(int id = 0)

//
// POST: /Movies/Delete/5
[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id = 0)
```

The common language runtime (CLR) requires overloaded methods to have a unique signature (same name, different list of parameters). However, here you need two Delete methods -- one for GET and one for POST -- that both require the same signature. (They both need to accept a single integer as a parameter.)

To sort this out, you can do a couple of things. One is to give the methods different names. That's what we did in

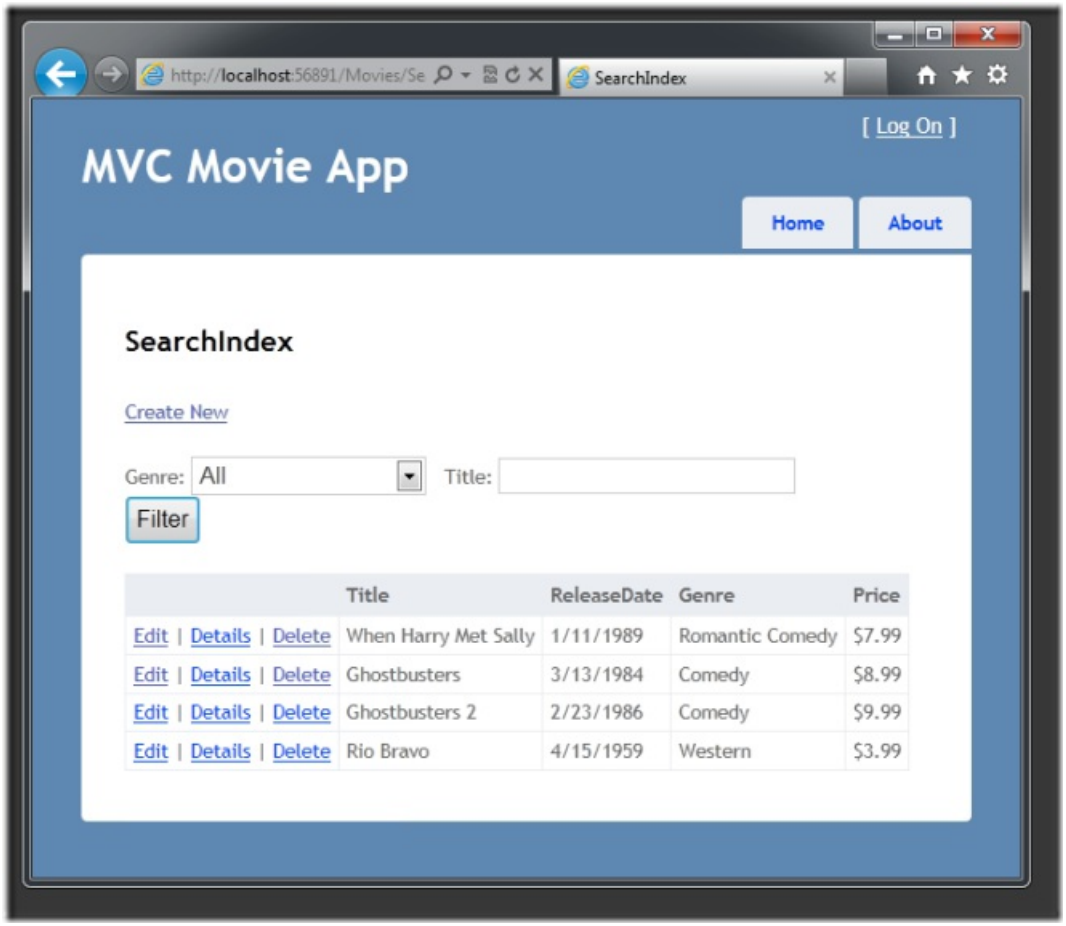
he preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. This effectively performs mapping for the routing system so that a URL that includes */Delete/* for a POST request will find the `DeleteConfirmed` method.

Another way to avoid a problem with methods that have identical names and signatures is to artificially change the signature of the POST method to include an unused parameter. For example, some developers add a parameter type `FormCollection` that is passed to the POST method, and then simply don't use the parameter:

```
public ActionResult Delete(FormCollection fcNotUsed, int id = 0)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    db.Movies.Remove(movie);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

Wrapping Up

You now have a complete ASP.NET MVC application that stores data in a SQL Server Compact database. You can create, read, update, delete, and search for movies.



This basic tutorial got you started making controllers, associating them with views, and passing around hard-coded data. Then you created and designed a data model. Entity Framework Code First created a database from the data model on the fly, and the ASP.NET MVC scaffolding system automatically generated the action methods and views for basic CRUD operations. You then added a search form that let users search the database. You

changed the database to include a new column of data, and then updated two pages to create and display this new data. You added validation by marking the data model with attributes from the `DataAnnotations` namespace. The resulting validation runs on the client and on the server.

If you'd like to deploy your application, it's helpful to first test the application on your local IIS 7 server. You can use this Web Platform Installer link to enable IIS setting for ASP.NET applications. See the following deployment links:

- ASP.NET Deployment Content Map
- Enabling IIS 7.x
- Web Application Projects Deployment

I now encourage you to move on to our intermediate-level Creating an Entity Framework Data Model for an ASP.NET MVC Application and MVC Music Store tutorials, to explore the ASP.NET articles on MSDN, and to check out the many videos and resources at https://asp.net/mvc to learn even more about ASP.NET MVC! The ASP.NET MVC forums are a great place to ask questions.

Enjoy!

— Scott Hanselman (http://hanselman.com and @shanselman on Twitter) and Rick Anderson blogs.msdn.com/rickAndy

PREVIOUS

Getting Started with ASP.NET MVC 3 (VB.NET)

2/20/2020 • 2 minutes to read • Edit Online

The basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express using VB.NET.

- Intro to ASP.NET MVC 3 (VB)
- Adding a Controller (VB)
- Adding a View (VB)
- Adding a Model (VB)
- Accessing your Model's Data from a Controller (VB)
- Examining the Edit Methods and Edit View (VB)
- Adding a New Field to the Movie Model and Database Table (VB)
- Adding Validation to the Model (VB)
- Improving the Details and Delete Methods (VB)

Intro to ASP.NET MVC 3 (VB)

2/20/2020 • 3 minutes to read • Edit Online

by Rick Anderson

> This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:
>
> - Visual Studio Web Developer Express SP1 prerequisites
> - ASP.NET MVC 3 Tools Update
> - SQL Server Compact 4.0(runtime + tools support)
>
> If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.
>
> A Visual Web Developer project with VB.NET source code is available to accompany this topic. Download the VB.NET version. If you prefer C#, switch to the C# version of this tutorial.

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

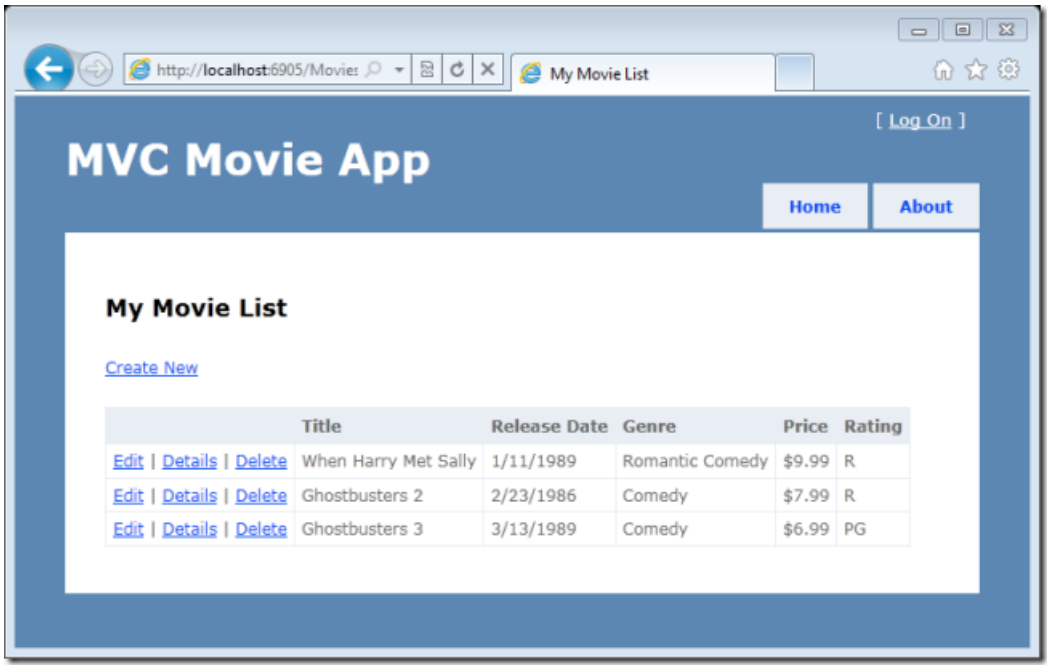
- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

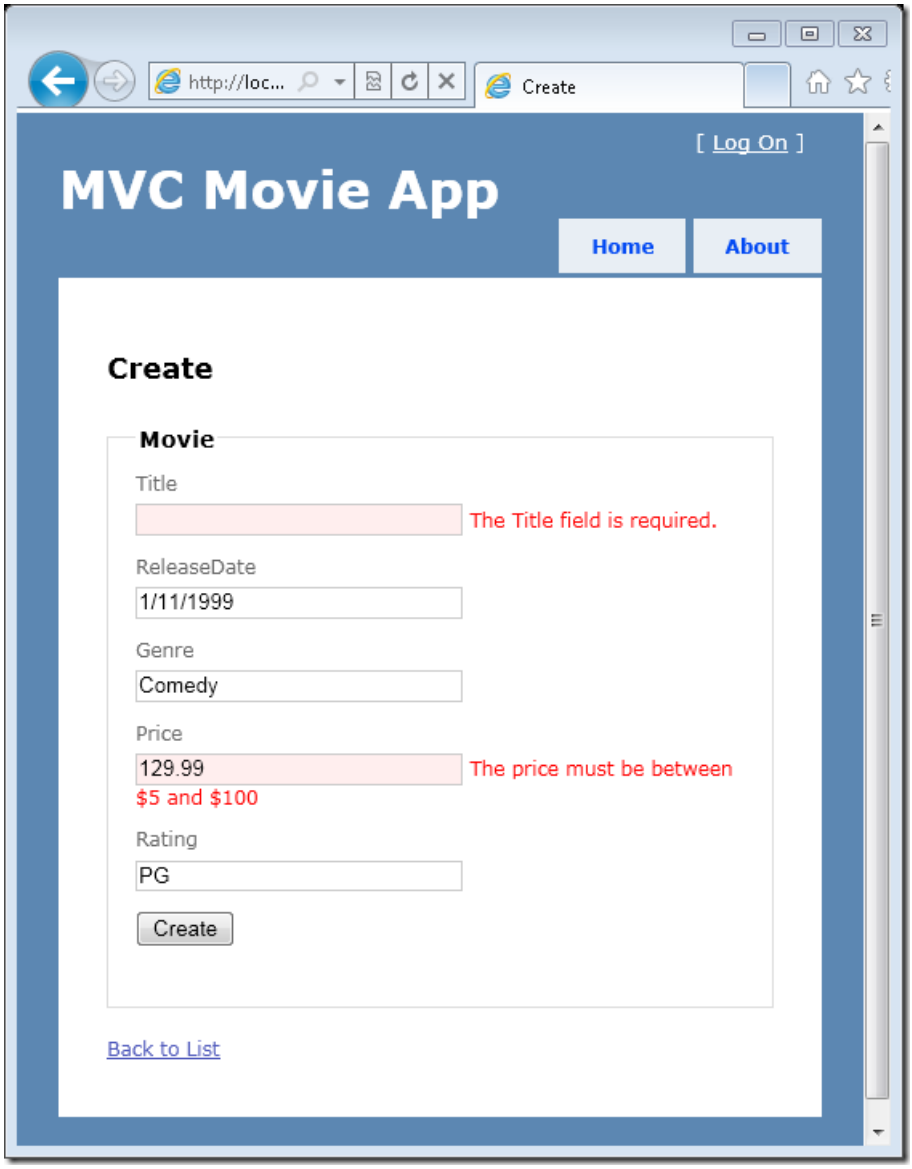
A Visual Web Developer project with VB source code is available to accompany this topic. Download the VB version here. If you prefer CSharp, switch to the CSharp version of this tutorial.

What You'll Build

You'll implement a simple movie-listing application that supports creating, editing, and listing movies from a database. Below are two screenshots of the application you'll build. It includes a page that displays a list of movies from a database:



The application also lets you add, edit, and delete movies, as well as see details about individual ones. All data-entry scenarios include validation to ensure that the data stored in the database is correct.



Skills You'll Learn

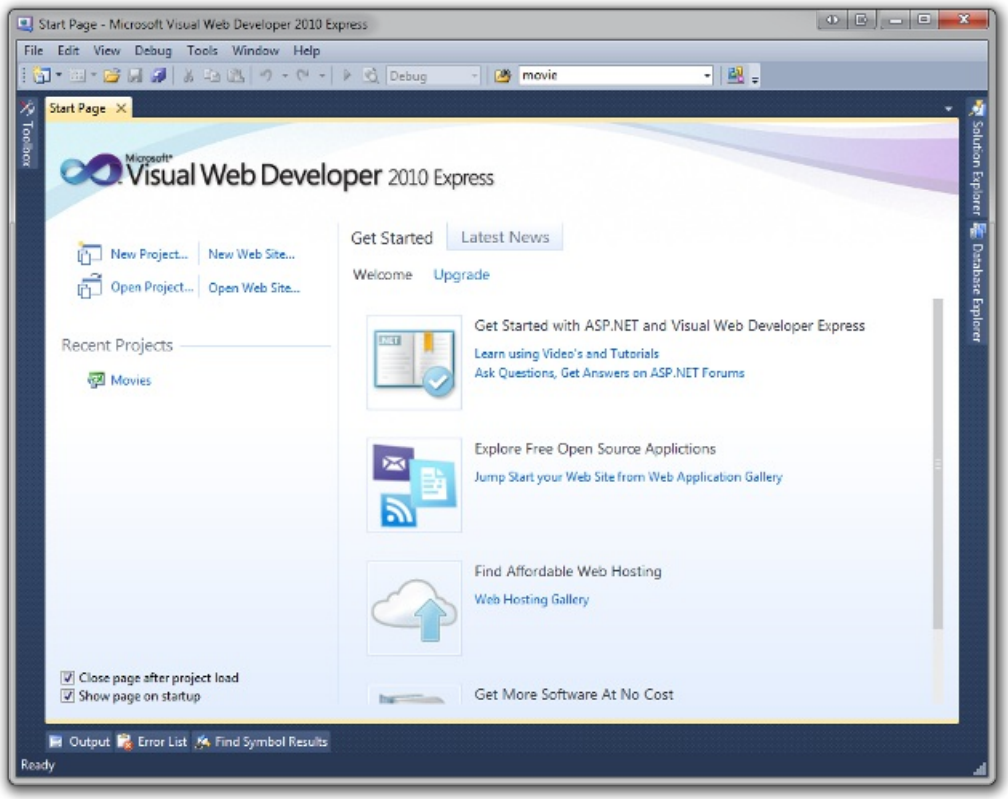
Here's what you'll learn:

- How to create a new ASP.NET MVC project
- How to create a new database using Entity Framework code-first
- How to create ASP.NET MVC controllers and views
- How to retrieve and display data
- How to edit data and enable data validation

Getting Started

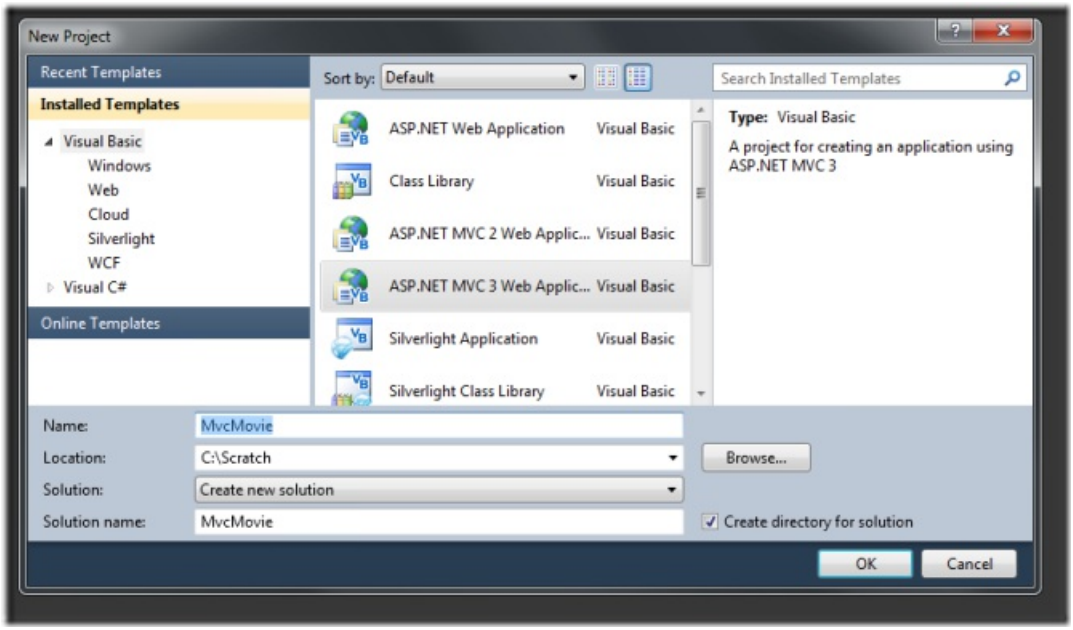
Start by running Visual Web Developer 2010 Express ("VWD" for short) and select **New Project** from the **Start** page.

Visual Web Developer is an IDE, or integrated development environment. Just like you use Microsoft Word to write documents, you'll use an IDE to create applications. In Visual Web Developer there's a toolbar along the top showing various options available to you. There's also a menu that provides another way to perform tasks in the IDE. (For example, instead of selecting **New Project** from the **Start** page, you can use the menu and select **File** > **New Project**.)

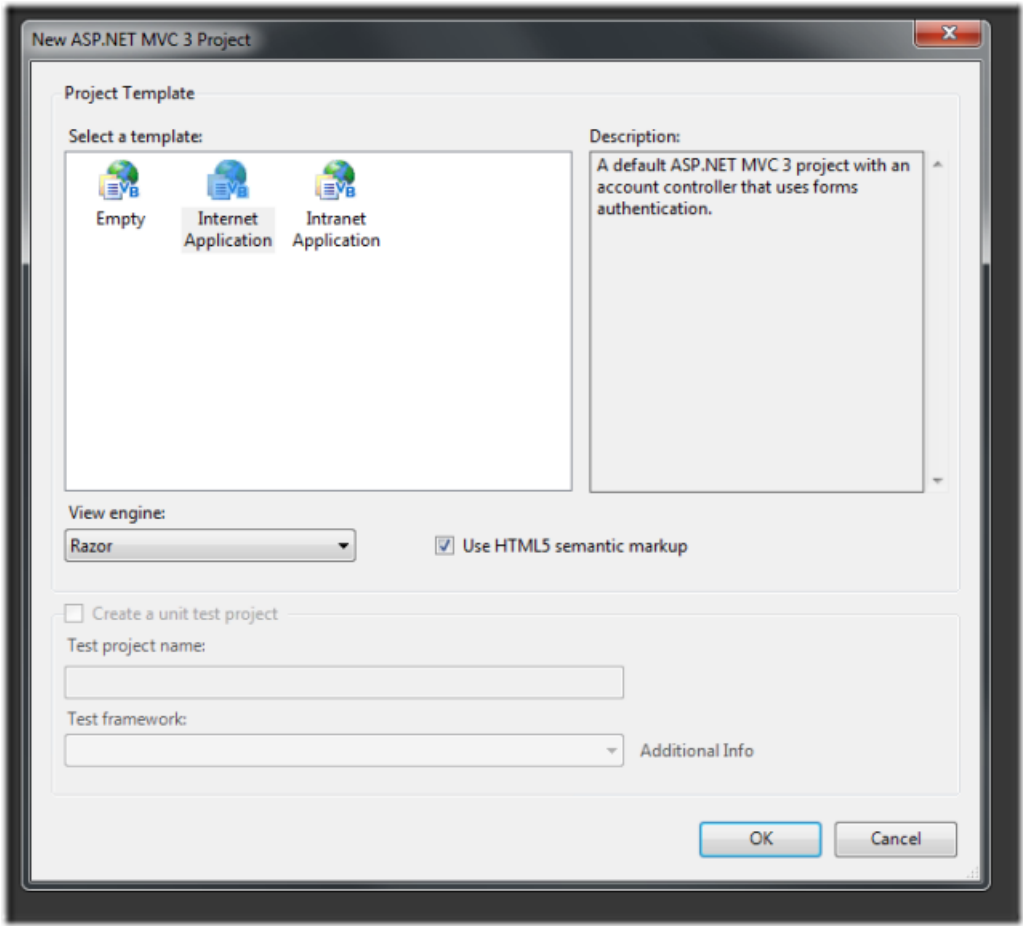


Creating Your First Application

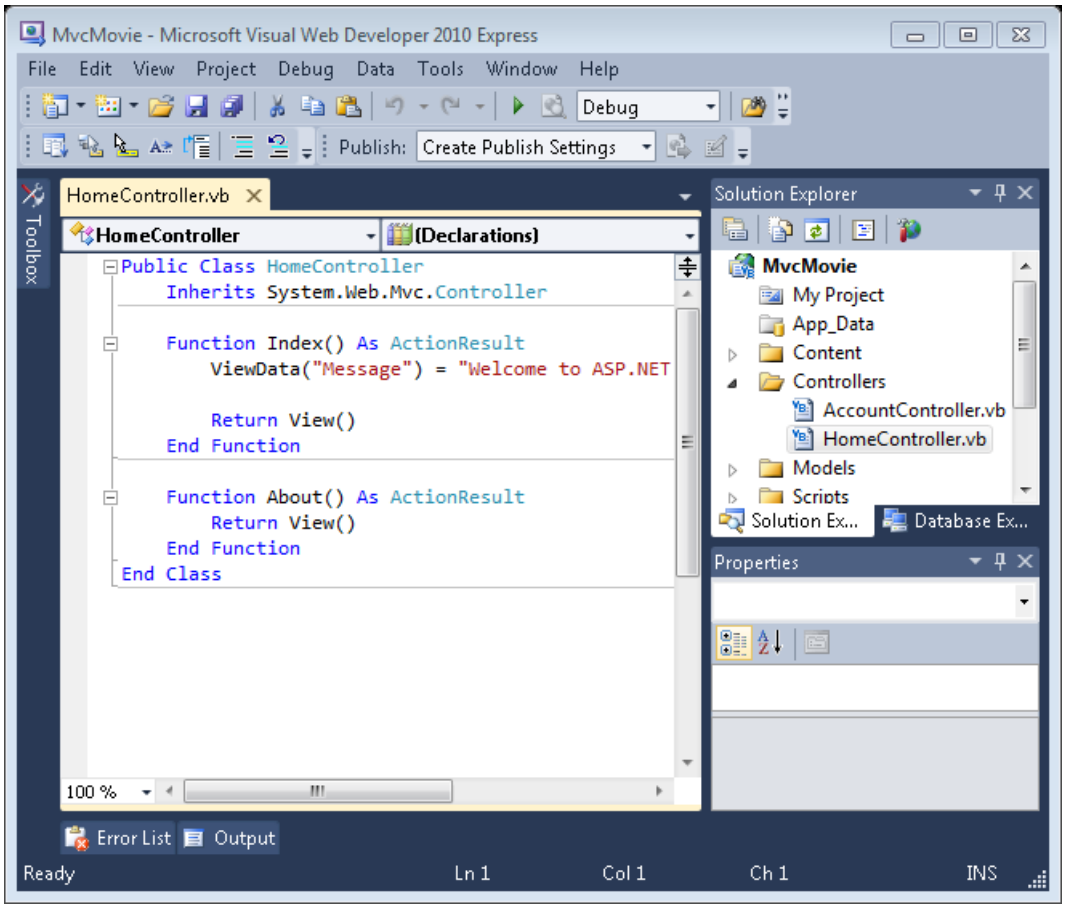
You can create applications using your choice of either Visual Basic or Visual C# as the programming language. For this tutorial, select Visual Basic on the left, then select **ASP.NET MVC 3 Web Application**. Name your project "MvcMovie" and then click **OK**.



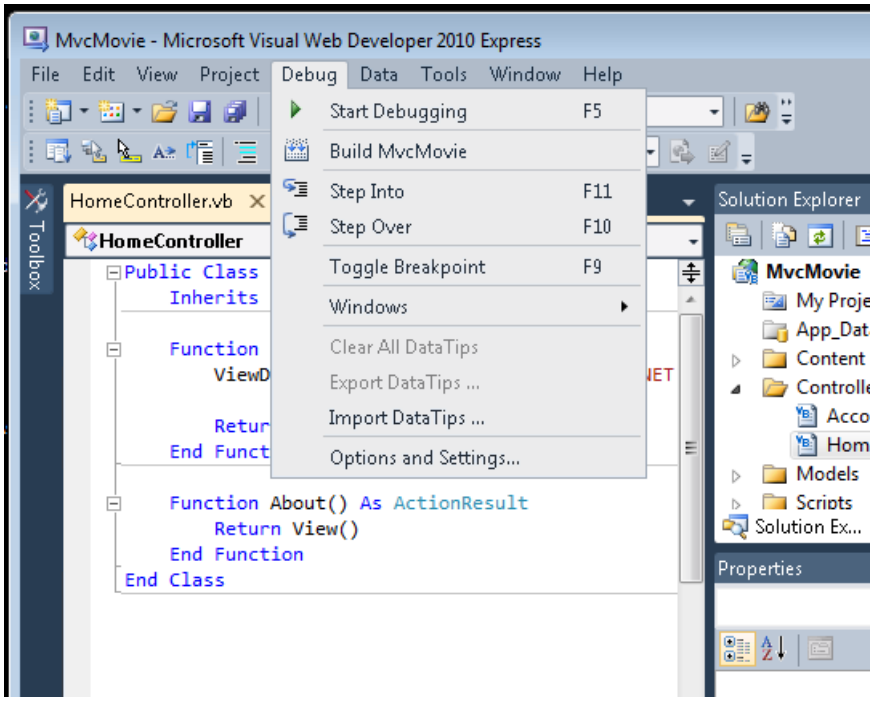
In the **New ASP.NET MVC 3 Project** dialog box, select **Internet Application**. Leave **Razor** as the default view engine.



Click **OK**. Visual Web Developer used a default template for the ASP.NET MVC project you just created, so you have a working application right now without doing anything! This is a simple "Hello World!" project, and it's a good place to start your application.

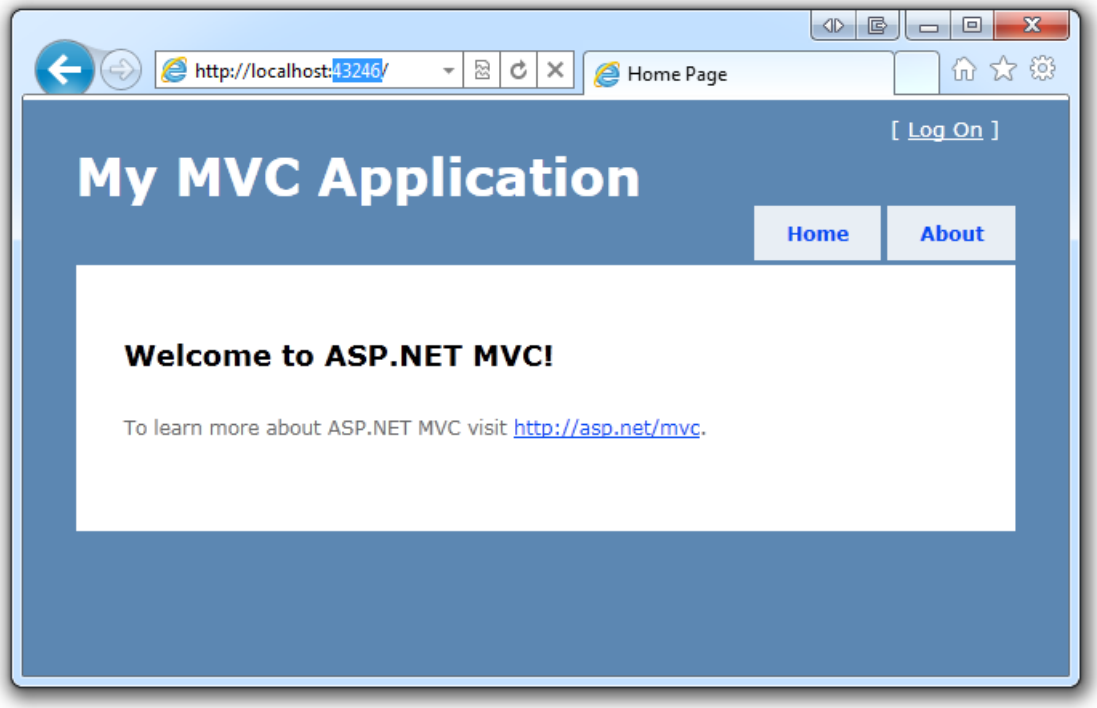


From the **Debug** menu, select **Start Debugging**.



Notice that the keyboard shortcut to start debugging is F5.

F5 causes Visual Web Developer to start a development web server and run your web application. VWD then launches a browser and opens the application's home page. Notice that the address bar of the browser says `localhost` and not something like `example.com`. That's because `localhost` always points to your own local computer, which in this case is running the application you just built. When VWD runs a web project, a random port is used for the project. In the image below, the random port number is 43246. Your project will probably use a different port number.



Out of the box this default template gives you two pages to visit and a basic login page. Let's change how this application works and learn a little bit about ASP.NET MVC in the process. Close your browser and let's change some code.

NEXT

Adding a Controller (VB)

2/20/2020 • 3 minutes to read • Edit Online

by Rick Anderson

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

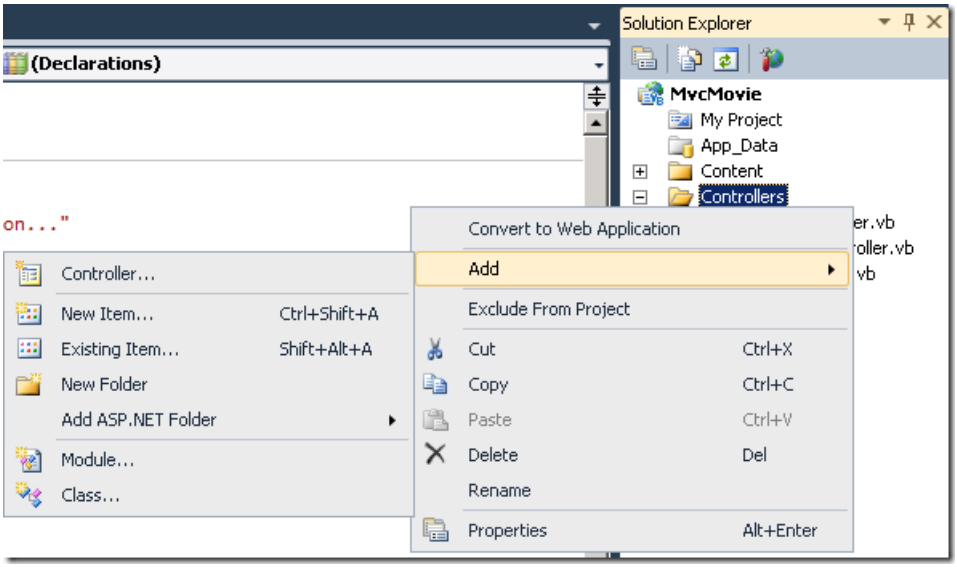
A Visual Web Developer project with VB.NET source code is available to accompany this topic. Download the VB.NET version. If you prefer C#, switch to the C# version of this tutorial.

MVC stands for *model-view-controller*. MVC is a pattern for developing applications such that each part has a separate responsibility:

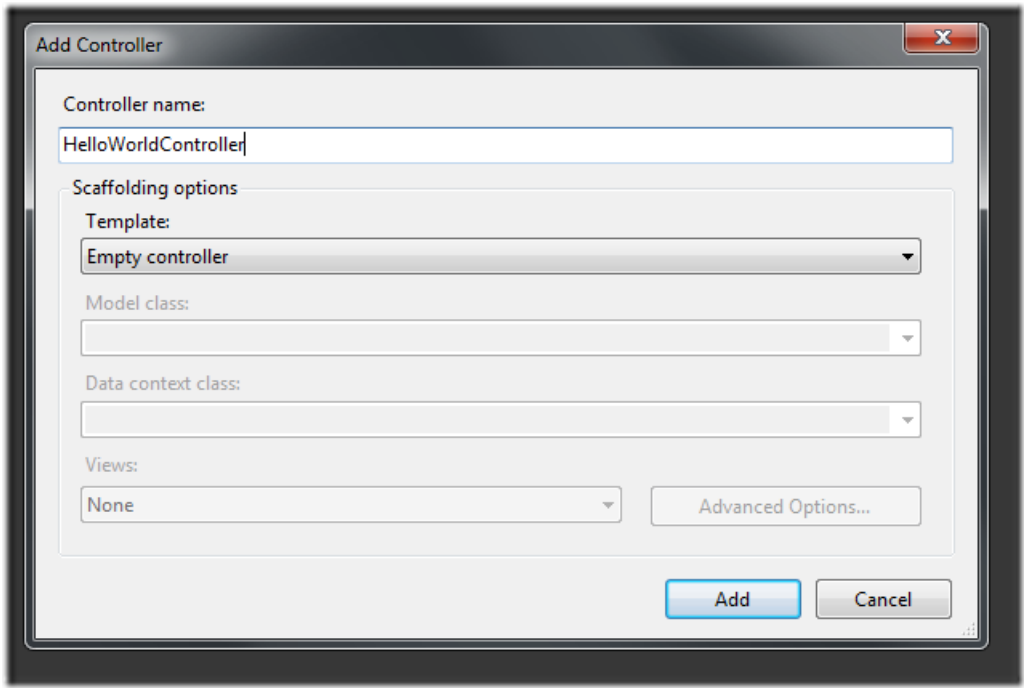
- Model: The data for your application.
- Views: The template files your application will use to dynamically generate HTML responses.
- Controllers: Classes that handle incoming URL requests to the application, retrieve model data, and then specify view templates that render a response to the client.

We'll be covering all these concepts in this tutorial and show you how to use them to build an application.

Create a new controller by right-clicking the *Controllers* folder in **Solution Explorer** and then selecting **Add Controller**.



Name your new controller "HelloWorldController" and click **Add**.



Notice in **Solution Explorer** on the right that a new file has been created for you named *HelloWorldController.cs* and that the file is open in the IDE.

Inside the new `public class HelloWorldController` block, create two new methods that look like the following code. We'll return a string of HTML directly from the controller as an example.

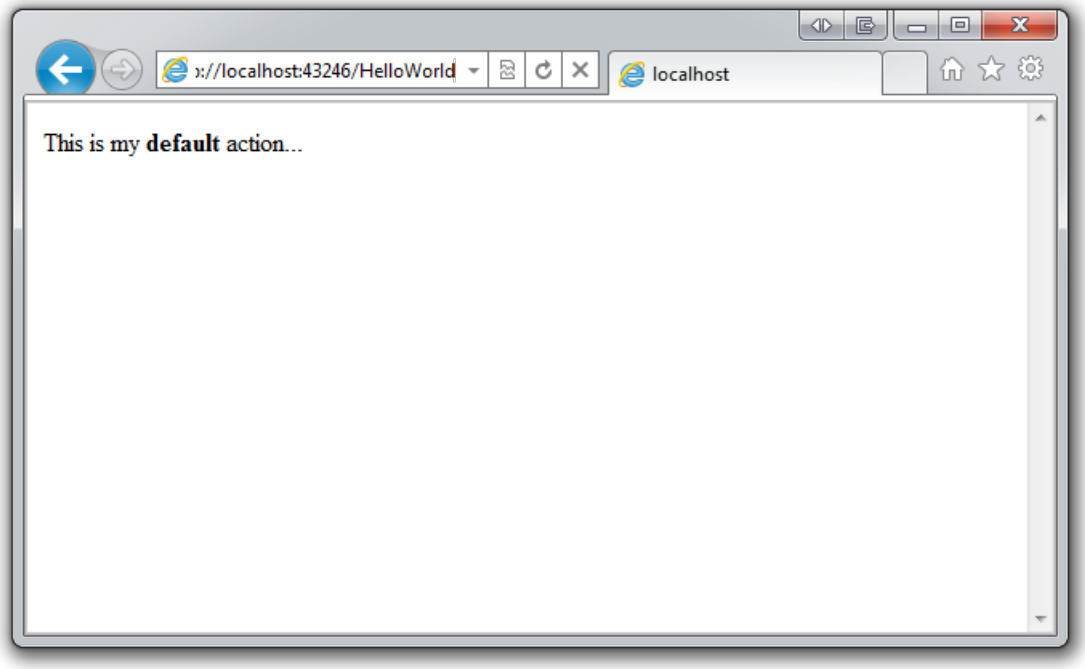
```
Namespace MvcMovie
    Public Class HelloWorldController
        Inherits System.Web.Mvc.Controller

        Public Function Index() As String
            Return "This is my default action..."
        End Function

        Public Function Welcome() As String
            Return "This is the Welcome action method..."
        End Function

    End Class
End Namespace
```

Your controller is named `HelloWorldController` and your new method is named `Index`. Run the application (press F5 or Ctrl+F5). Once your browser has started up, append "HelloWorld" to the path in the address bar. (On my computer, it's `http://localhost:43246/HelloWorld`) Your browser will look like the screenshot below. In the method above, the code returned a string directly. We told the system to just return some HTML, and it did!

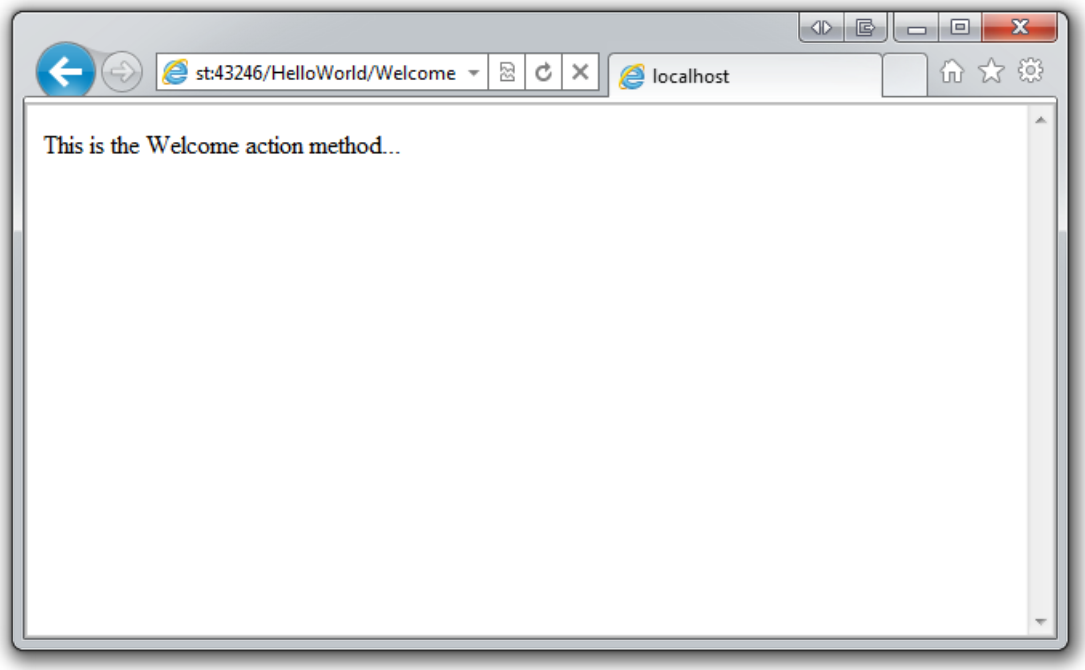


ASP.NET MVC invokes different controller classes (and different action methods within them) depending on the incoming URL. The default mapping logic used by ASP.NET MVC uses a format like this to control what code is invoked:

```
/[Controller]/[ActionName]/[Parameters]
```

The first part of the URL determines the controller class to execute. So */HelloWorld* maps to the `HelloWorldController` class. The second part of the URL determines the action method on the class to execute. So */HelloWorld/Index* would cause the `Index` method of the `HelloWorldController` class to execute. Notice that we only had to visit */HelloWorld* above and the `Index` method was used by default. This is because a method named `Index` is the default method that will be called on a controller if one is not explicitly specified.

Browse to `http://localhost:xxxx/HelloWorld/Welcome`. The `Welcome` method runs and returns the string "This is the Welcome action method...". The default MVC mapping is `/[Controller]/[ActionName]/[Parameters]`. For this URL, the controller is `HelloWorld` and `Welcome` is the method. We haven't used the `[Parameters]` part of the URL yet.

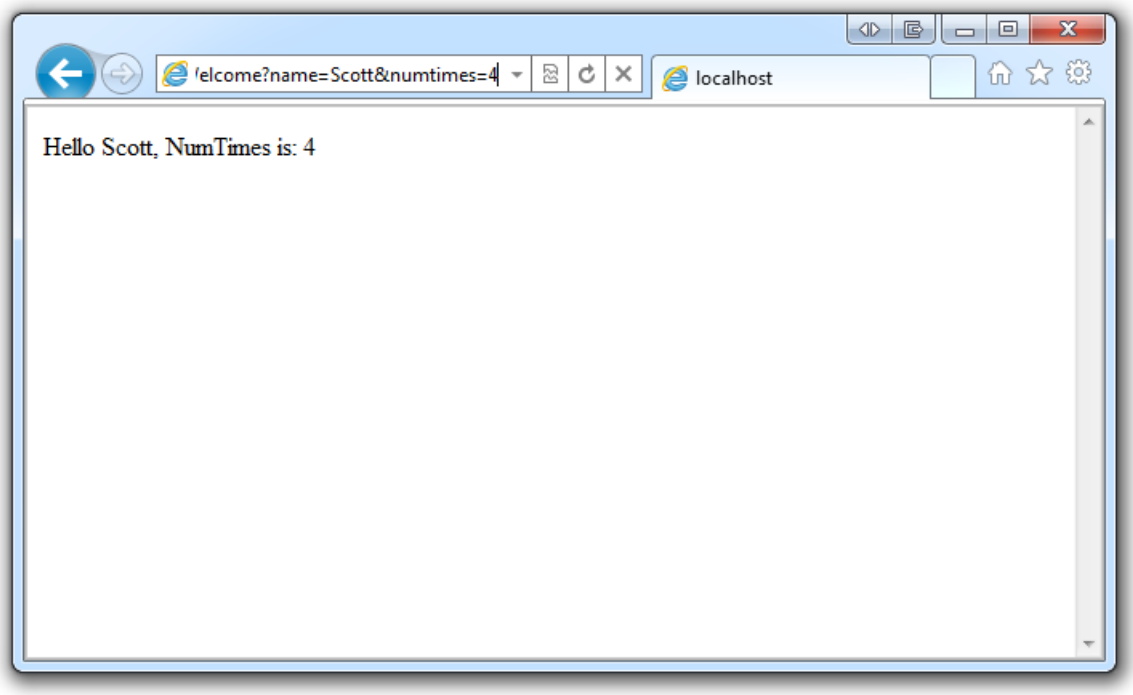


Let's modify the example slightly so that we can pass some parameter information in from the URL to the

controller (for example, */HelloWorld/Welcome?name=Scott&numtimes=4*). Change your `Welcome` method to include two parameters as shown below. Note that we've used the VB optional parameter feature to indicate that the `numTimes` parameter should default to 1 if no value is passed for that parameter.

```
Public Function Welcome(ByVal name As String, Optional ByVal numTimes As Integer = 1) As String
    Dim message As String = "Hello " & name & ", NumTimes is: " & numTimes
    Return "" & Server.HtmlEncode(message) & ""
End Function
```

Run your application and browse to `http://localhost:xxxx/HelloWorld/Welcome?name=Scott&numtimes=4`. You can try different values for `name` and `numtimes`. The system automatically maps the named parameters from your query string in the address bar to parameters in your method.



In both these examples the controller has been doing the VC portion of MVC — that is the view and controller work. The controller is returning HTML directly. Ordinarily we don't want controllers returning HTML directly, since that becomes very cumbersome to code. Instead we'll typically use a separate view template file to help generate the HTML response. Let's look at how we can do this.

PREVIOUS NEXT

Adding a View (VB)

2/20/2020 • 6 minutes to read • Edit Online

by Rick Anderson

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

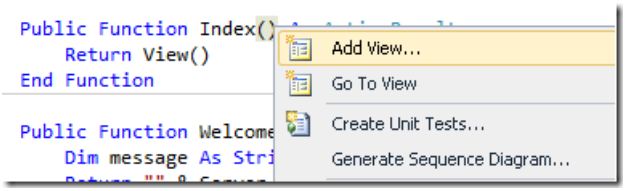
A Visual Web Developer project with VB.NET source code is available to accompany this topic. Download the VB.NET version. If you prefer C#, switch to the C# version of this tutorial.

In this section we're going to modify the `HelloWorldController` class to use a view template file to cleanly encapsulate the process of generating HTML responses to a client.

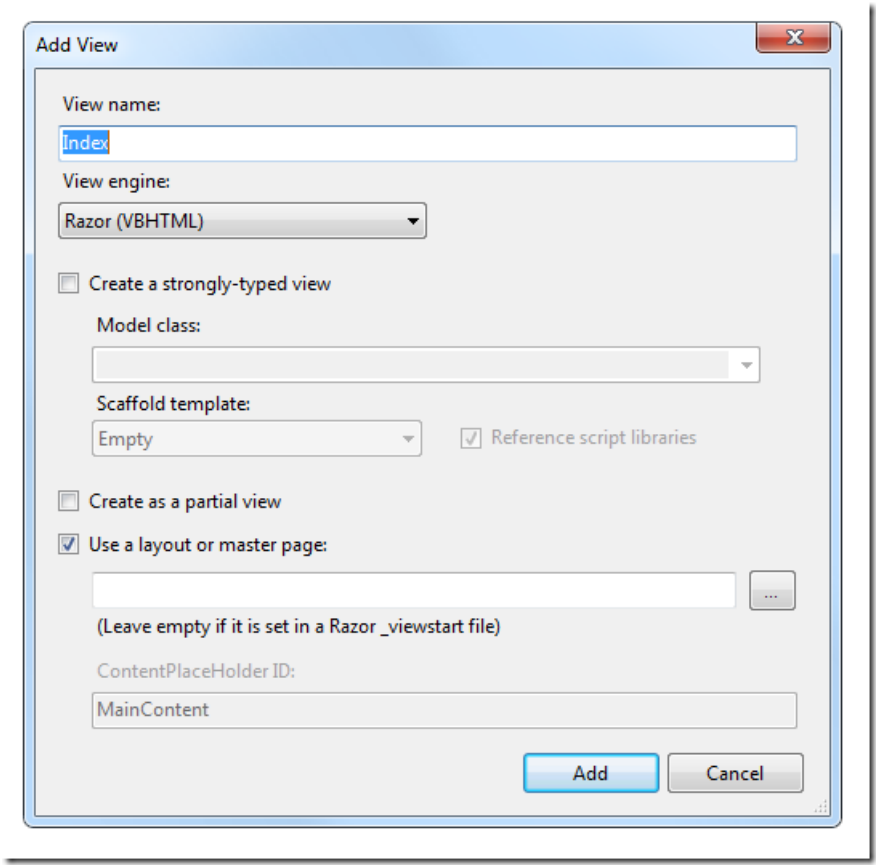
Let's start by using a view template with the `Index` method in the `HelloWorldController` class. Currently the `Index` method returns a string with a message that is hard-coded within the controller class. Change the `Index` method to return a `View` object, as shown in the following:

```
Public Function Index() As ActionResult
            Return View()
        End Function
```

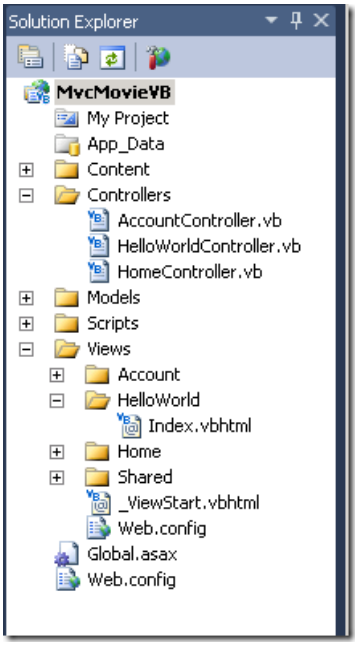
Let's now add a view template to our project that we can invoke with the `Index` method. To do this, right-click inside the `Index` method and click **Add View**.



The **Add View** dialog box appears. Leave the default entries and click the **Add** button.



The *MvcMovie\Views\HelloWorld* folder and the *MvcMovie\Views\HelloWorld\Index.vbhtml* file are created. You can see them in **Solution Explorer**:



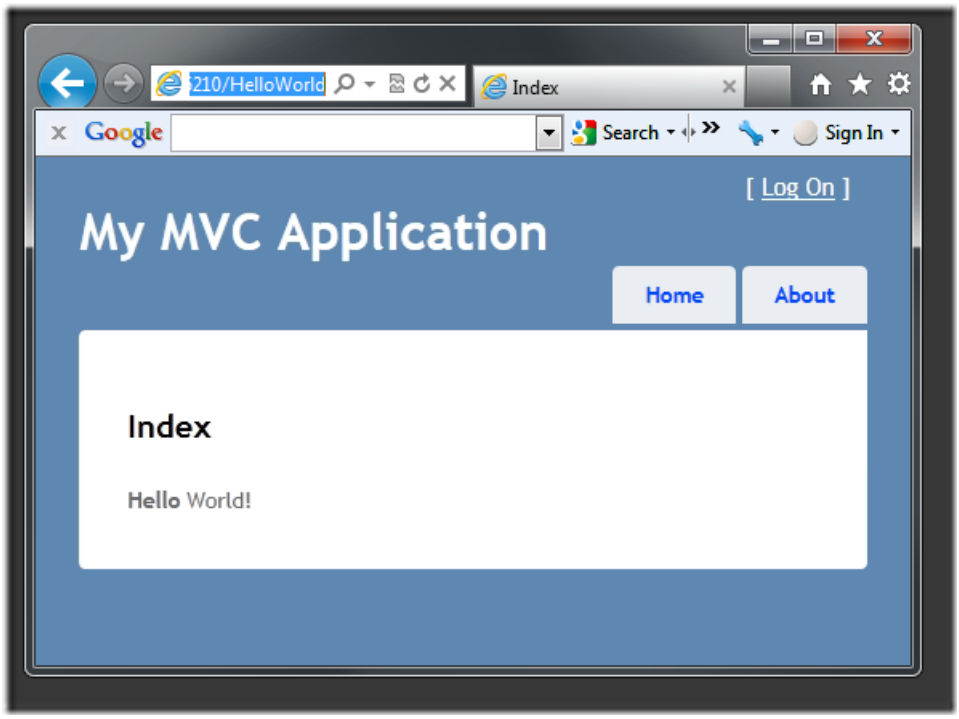
Add some HTML under the `<h2>` tag. The modified *MvcMovie\Views\HelloWorld\Index.vbhtml* file is shown below.

```
@Code
    ViewData("Title") = "Index"
End Code

<h2>Index</h2>

<b>Hello</b> World!
```

Run the application and browse to the "hello world" controller (`http://localhost:xxxx/HelloWorld`). The `Index` method in your controller didn't do much work; it simply ran the statement `return View()`, which indicated that we wanted to use a view template file to render a response to the client. Because we did not explicitly specify the name of the view template file to use, ASP.NET MVC defaulted to using the *Index.vbhtml* view file within the *\Views\HelloWorld* folder. The image below shows the string hard-coded in the view.



Looks pretty good. However, notice that the browser's title bar says "Index" and the big title on the page says "My MVC Application." Let's change those.

Changing views and layout pages

First, let's change the text "My MVC Application." That text is shared and appears on every page. It actually appears in only one place in our project, even though it's on every page in our application. Go to the */Views/Shared* folder in **Solution Explorer** and open the *_Layout.vbhtml* file. This file is called a layout page and it's the shared "shell" that all other pages use.

Note the `@RenderBody()` line of code near the bottom of the file. `RenderBody` is a placeholder where all the pages you create show up, "wrapped" in the layout page. Change the `<h1>` heading from " My MVC Application" to "MVC Movie App".

```
<div id="title">
        <h1>MVC Movie App</h1>
  </div>
```

Run the application and note it now says "MVC Movie App". Click the **About** link, and that page shows "MVC Movie App", too.

The complete *_Layout.vbhtml* file is shown below:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewData("Title")</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")" type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/modernizr-1.7.min.js")" type="text/javascript"></script>
</head>
<body>
    <div class="page">
        <header>
            <div id="title">
                <h1>MVC Movie App</h1>
            </div>
            <div id="logindisplay">
                @Html.Partial("_LogOnPartial")
            </div>
            <nav>
                <ul id="menu">
                    <li>@Html.ActionLink("Home", "Index", "HelloWorld")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                </ul>
            </nav>
        </header>
        <section id="main">
            @RenderBody()
        </section>
        <footer>
        </footer>
    </div>
</body>
</html>
```

Now, let's change the title of the Index page (view).

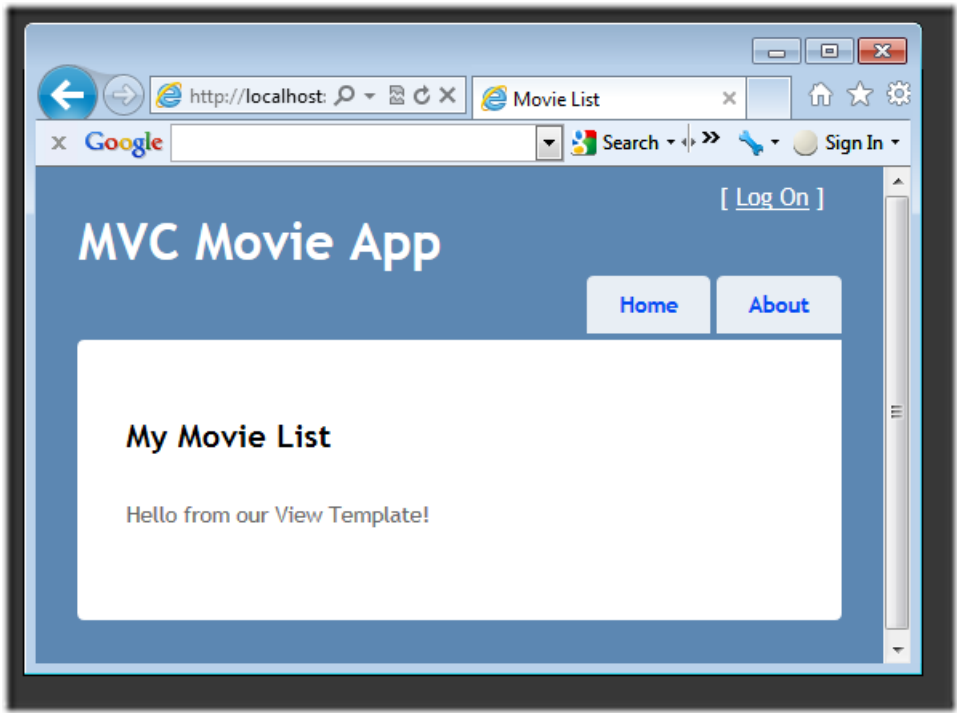
```
@Code
    ViewData("Title") = "Movie List"
End Code

<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

Open *MvcMovie\Views\HelloWorld\Index.vbhtml*. There are two places to make a change: first, the text that appears in the title of the browser, and then in the secondary header (the `<h2>` element). We'll make them slightly different so you can see which bit of code changes which part of the app.

Run the application and browse to `http://localhost:xx/HelloWorld`. Notice that the browser title, the primary heading, and the secondary headings have changed. It's easy to make big changes in your application with small changes to a view. (If you don't see changes in the browser, you might be viewing cached content. Press Ctrl+F5 in your browser to force the response from the server to be loaded.)



Our little bit of "data" (in this case the "Hello World!" message) is hard-coded, though. Our MVC application has V (views) and we've got C (controllers), but no M (model) yet. Shortly, we'll walk through how create a database and retrieve model data from it.

Passing Data from the Controller to the View

Before we go to a database and talk about models, though, let's first talk about passing information from the Controller to a View. We want to pass what a view template requires in order to render an HTML response to a client. These objects are typically created and passed by a controller class to a view template, and they should contain only the data that the view template requires — and no more.

Previously with the `HelloWorldController` class, the `Welcome` action method took a `name` and a `numTimes` parameter and then output the parameter values to the browser. Rather than have the controller continue to render this response directly, let's instead we'll put that data in a bag for the View. Controllers and Views can use a `ViewBag` object to hold that data. That will be passed over to a view template automatically, and used to render the HTML response using the contents of the bag as data. That way the controller is concerned with one thing and the view template with another — enabling us to maintain clean "separation of concerns" within the application.

Alternatively, we could define a custom class, then create an instance of that object on our own, fill it with data and pass it to the View. That is often called a ViewModel, because it's a custom Model for the View. For small amounts of data, however, the ViewBag works great.

Return to the *HelloWorldController.vb* file change the `Welcome` method inside the controller to put the Message and NumTimes into the ViewBag. The ViewBag is a dynamic object. That means you can put whatever you want in to it. The ViewBag has no defined properties until you put something inside it.

The complete `HelloWorldController.vb` with the new class in the same file.

```
Namespace MvcMovie
    Public Class HelloWorldController
        Inherits System.Web.Mvc.Controller
        '
        ' GET: /HelloWorld

        Function Index() As ActionResult
            Return View()
        End Function

        Public Function Welcome(ByVal name As String, Optional ByVal numTimes As Integer = 1) As ActionResult
            ViewBag.Message = "Hello " & name
            ViewBag.NumTimes = numTimes
            Return View()
        End Function

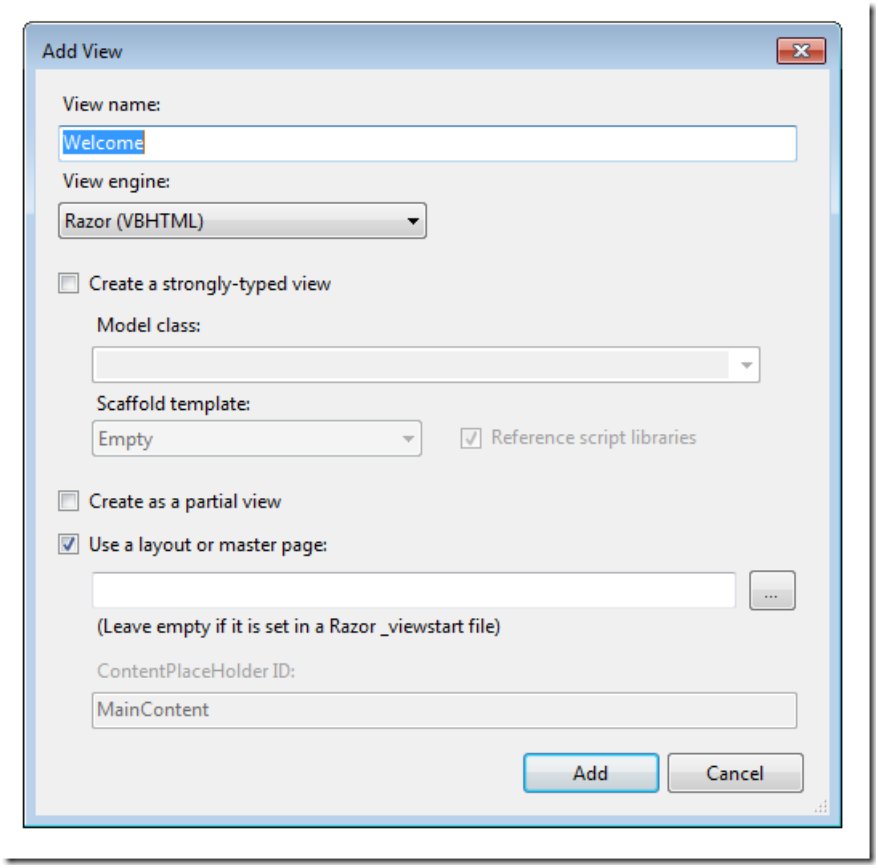
    End Class
End Namespace
```

Now our ViewBag contains data that will be passed over to the View automatically. Again, alternatively we could have passed in our own object like this if we liked:

```
return View(myCustomObject)
```

Now we need a `WelcomeView` template! Run the application so the new code is compiled. Close the browser, right-click inside the `Welcome` method, and then click **Add View**.

Here's what your **Add View** dialog box looks like.

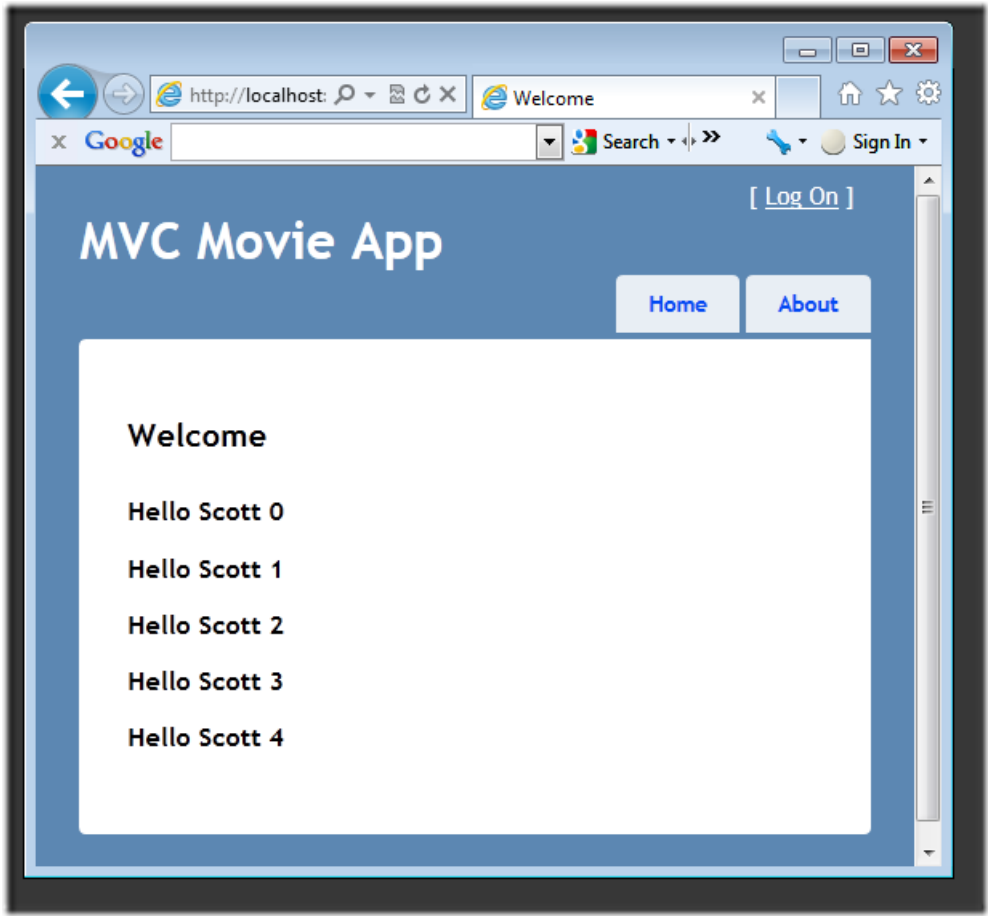


Add the following code under the `<h2>` element in the new *Welcome*.vbhtml file. We'll make a loop and say "Hello" as many times as the user says we should!

```
@Code
    For i As Integer = 0 To ViewBag.NumTimes
@<h3> @ViewBag.Message @i.ToString </h3>
     Next i
 End Code
```

Run the application and browse to `http://localhost:xx/HelloWorld/Welcome?name=Scott&numtimes=4`

Now data is taken from the URL and passed to the controller automatically. The controller packages up the data into a `Model` object and passes that object to the view. The view than displays the data as HTML to the user.



Well, that was a kind of an "M" for model, but not the database kind. Let's take what we've learned and create a database of movies.

PREVIOUS NEXT

Adding a Model (VB)

2/20/2020 • 3 minutes to read • Edit Online

by Rick Anderson

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

A Visual Web Developer project with VB.NET source code is available to accompany this topic. Download the VB.NET version. If you prefer C#, switch to the C# version of this tutorial.

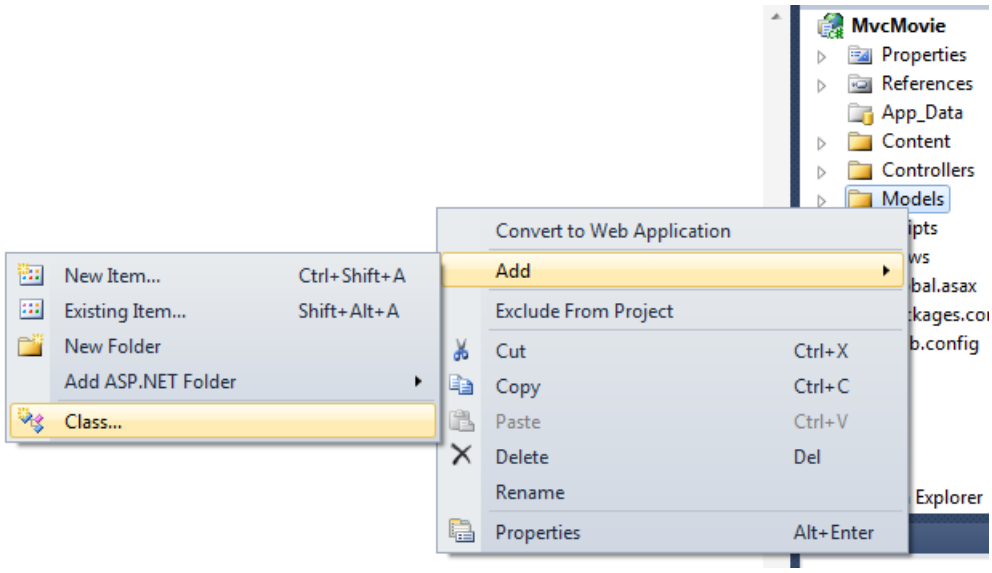
Adding a Model

In this section you'll add some classes for managing movies in a database. These classes will be the "model" part of the ASP.NET MVC application.

You'll use a .NET Framework data-access technology known as the Entity Framework to define and work with these model classes. The Entity Framework (often referred to as EF) supports a development paradigm called *Code First*. Code First allows you to create model objects by writing simple classes. (These are also known as POCO classes, from "plain-old CLR objects.") You can then have the database created on the fly from your classes, which enables a very clean and rapid development workflow.

Adding Model Classes

In **Solution Explorer**, right click the *Models* folder, select **Add**, and then select **Class**.



Name the class "Movie".

Add the following five properties to the `Movie` class:

```
Public Class Movie
        Public Property ID() As Integer
        Public Property Title() As String
        Public Property ReleaseDate() As Date
        Public Property Genre() As String
        Public Property Price() As Decimal
End Class
```

We'll use the `Movie` class to represent movies in a database. Each instance of a `Movie` object will correspond to a row within a database table, and each property of the `Movie` class will map to a column in the table.

In the same file, add the following `MovieDBContext` class:

```
Public Class MovieDBContext
    Inherits DbContext
    Public Property Movies() As DbSet(Of Movie)
End Class
```

The `MovieDBContext` class represents the Entity Framework movie database context, which handles fetching, storing, and updating `Movie` class instances in a database. The `MovieDBContext` derives from the `DbContext` base class provided by the Entity Framework. For more information about `DbContext` and `DbSet`, see Productivity Improvements for the Entity Framework.

In order to be able to reference `DbContext` and `DbSet`, you need to add the following `imports` statement at the top of the file:

```
Imports System.Data.Entity
```

The complete *Movie.vb* file is shown below.

```
Imports System.Data.Entity

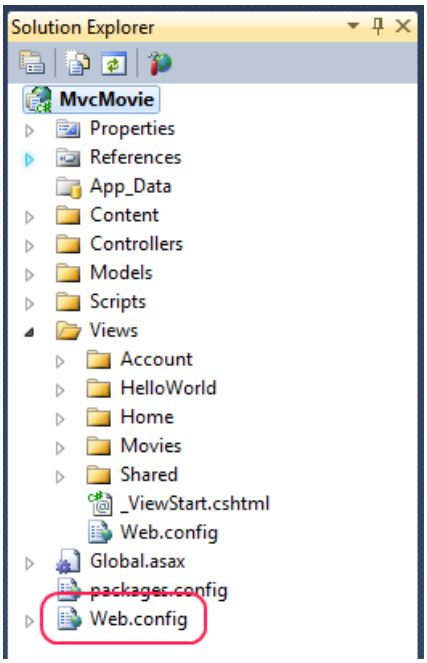
Public Class Movie
        Public Property ID() As Integer
        Public Property Title() As String
        Public Property ReleaseDate() As Date
        Public Property Genre() As String
        Public Property Price() As Decimal
End Class

Public Class MovieDBContext
    Inherits DbContext
    Public Property Movies() As DbSet(Of Movie)
End Class
```

Creating a Connection String and Working with SQL Server Compact

The `MovieDBContext` class you created handles the task of connecting to the database and mapping `Movie` objects to database records. One question you might ask, though, is how to specify which database it will connect to. You'll do that by adding connection information in the *Web.config* file of the application.

Open the application root *Web.config* file. (Not the *Web.config* file in the *Views* folder.) The image below show both *Web.config* files; open the *Web.config* file circled in red.



Add the following connection string to the `<connectionStrings>` element in the *Web.config* file.

```
<add name="MovieDBContext"
         connectionString="Data Source=|DataDirectory|Movies.sdf"
         providerName="System.Data.SqlServerCe.4.0"/>
```

The following example shows a portion of the *Web.config* file with the new connection string added:

```
<configuration>
  <connectionStrings>
    <add name="MovieDBContext"
         connectionString="Data Source=|DataDirectory|Movies.sdf"
         providerName="System.Data.SqlServerCe.4.0"/>
    <add name="ApplicationServices"
         connectionString="data source=.\SQLEXPRESS;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User Instance=true"
         providerName="System.Data.SqlClient" />
  </connectionStrings>
```

This small amount of code and XML is everything you need to write in order to represent and store the movie data in a database.

Next, you'll build a new `MoviesController` class that you can use to display the movie data and allow users to create new movie listings.

PREVIOUS NEXT

Accessing your Model's Data from a Controller (VB)

2/20/2020 • 6 minutes to read • Edit Online

by Rick Anderson

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

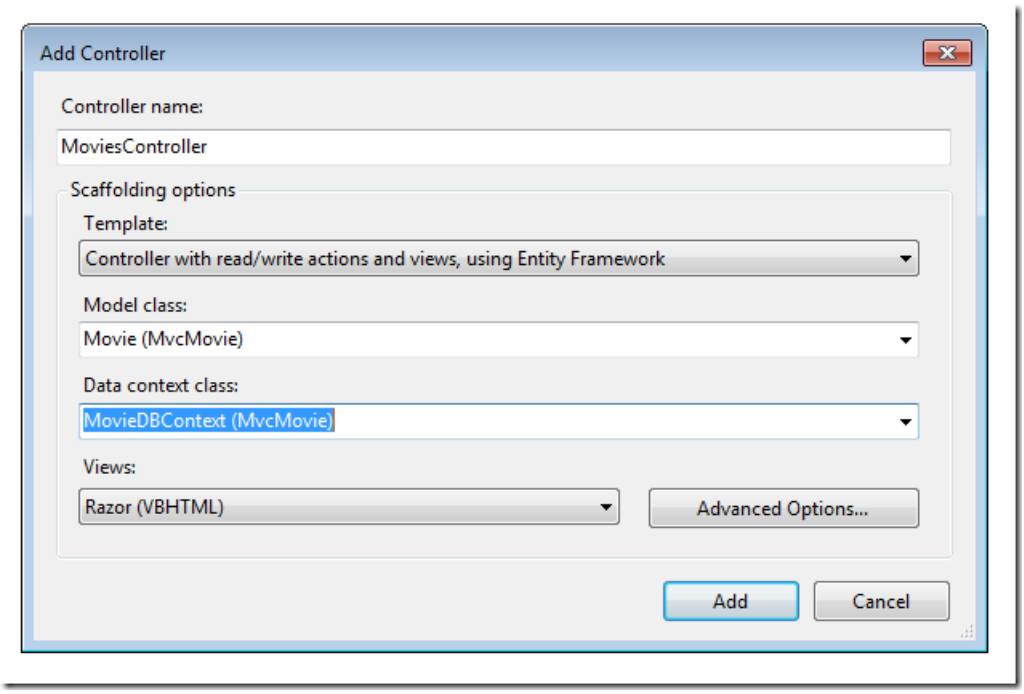
If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

A Visual Web Developer project with VB.NET source code is available to accompany this topic. Download the VB.NET version. If you prefer C#, switch to the C# version of this tutorial.

In this section, you'll create a new `MoviesController` class and write code that retrieves the movie data and displays it in the browser using a view template. Be sure to build your application before proceeding.

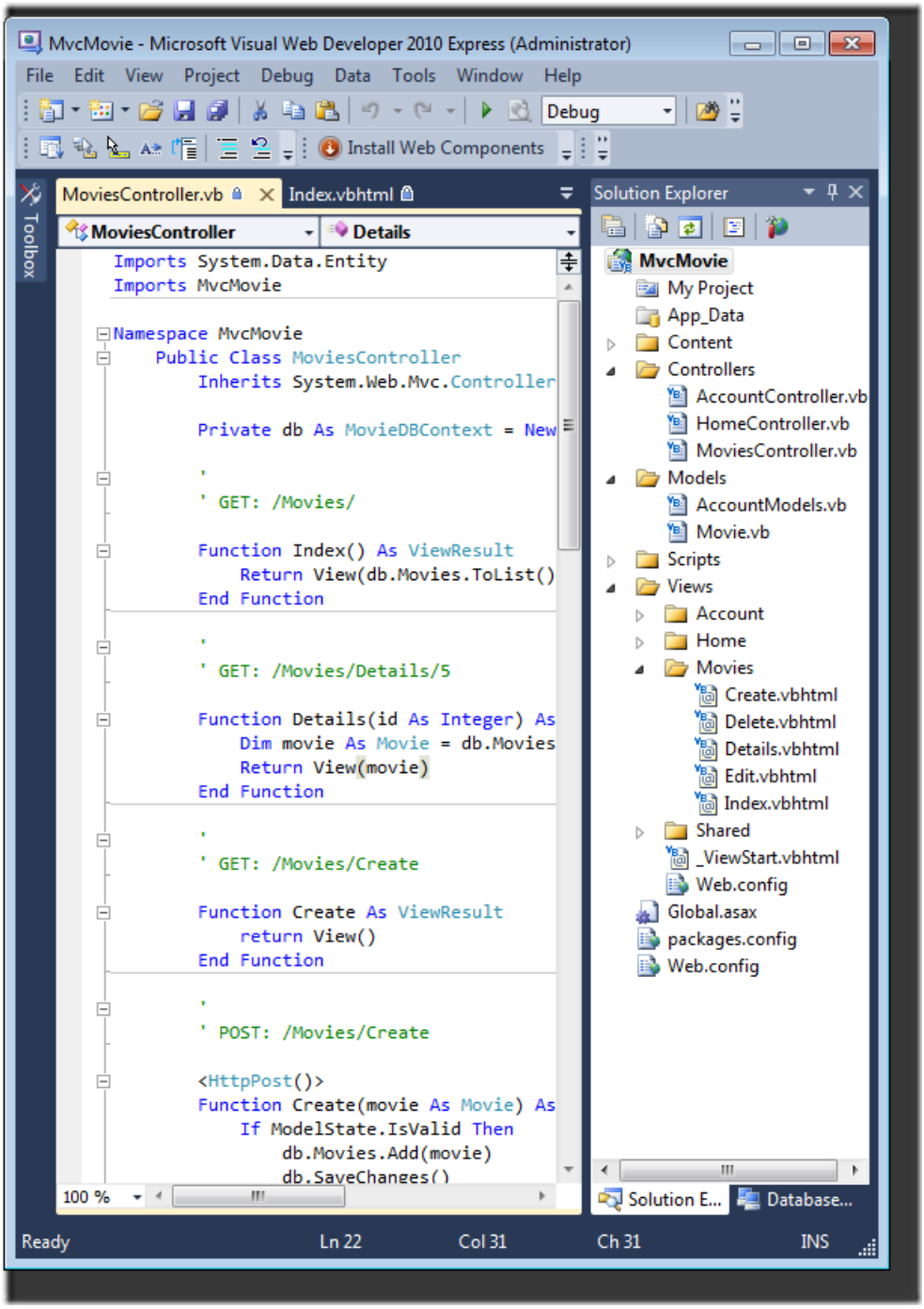
Right-click the *Controllers* folder and create a new `MoviesController` controller. Select the following options:

- Controller name: **MoviesController**. (This is the default.)
- Template: **Controller with read/write actions and views, using Entity Framework**.
- Model class: **Movie (MvcMovie.Models)**.
- Data context class: **MovieDBContext (MvcMovie.Models)**.
- Views: **Razor (CSHTML)**. (The default.)



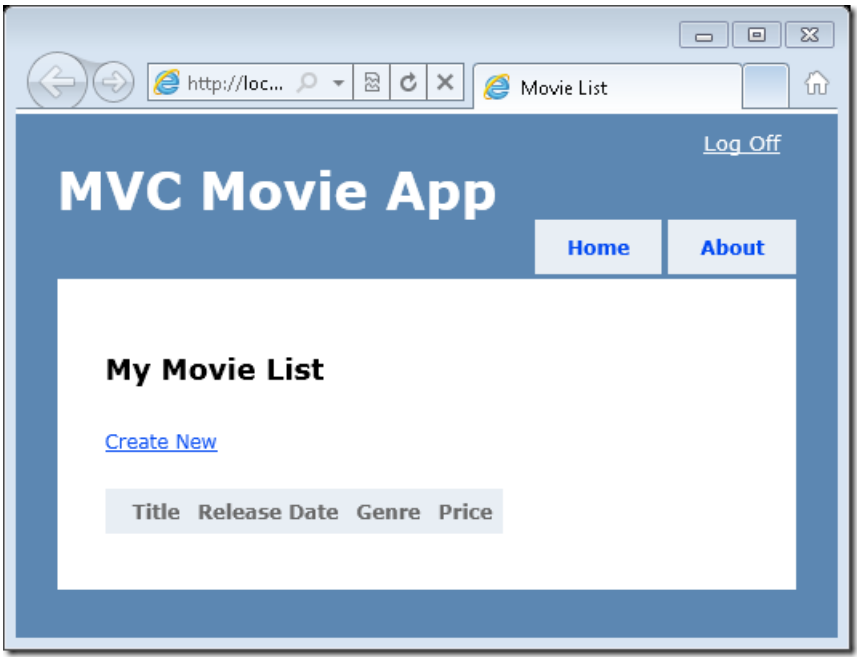
Click **Add**. Visual Web Developer creates the following files and folders:

- *A MoviesController.vb* file in the project's *Controllers* folder.
- A *Movies* folder in the project's *Views* folder.
- *Create.vbhtml, Delete.vbhtml, Details.vbhtml, Edit.vbhtml*, and *Index.vbhtml* in the new *Views\Movies* folder.



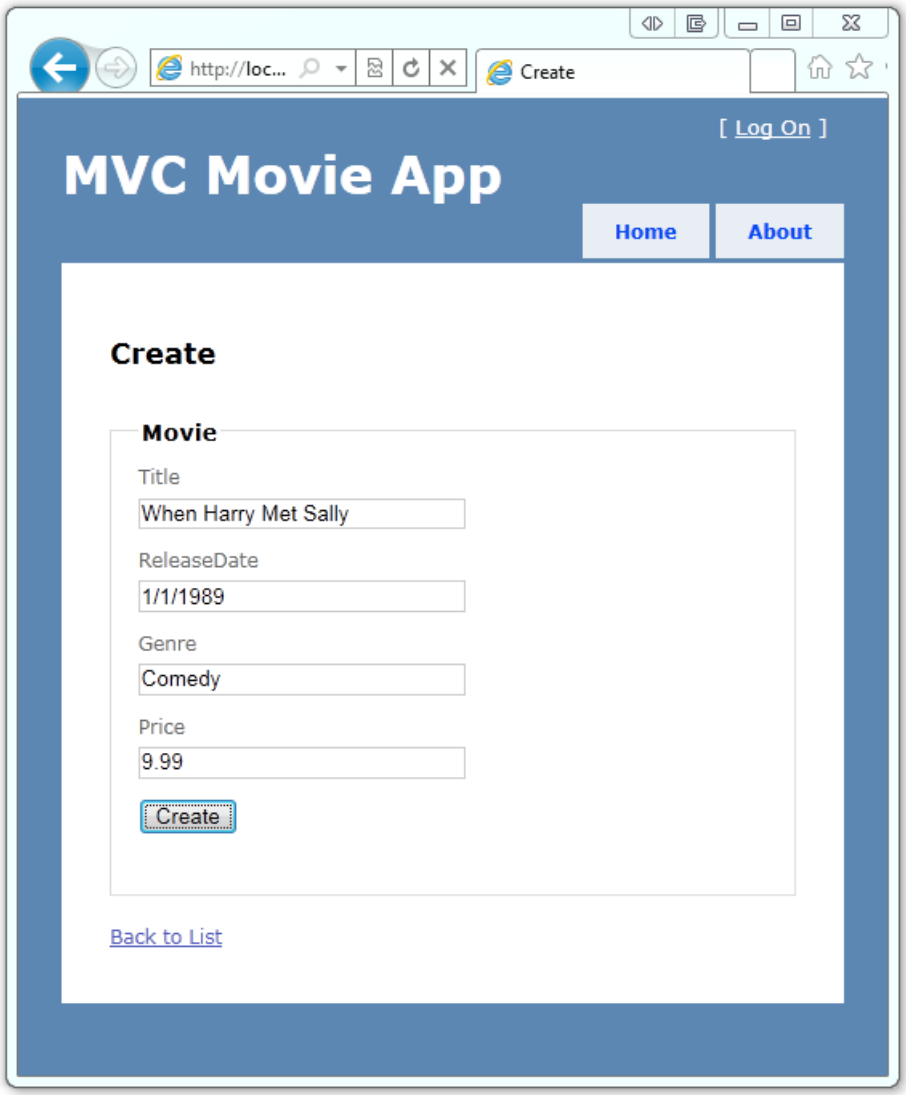
The ASP.NET MVC 3 scaffolding mechanism automatically created the CRUD (create, read, update, and delete) action methods and views for you. You now have a fully functional web application that lets you create, list, edit, and delete movie entries.

Run the application and browse to the `Movies` controller by appending */Movies* to the URL in the address bar of your browser. Because the application is relying on the default routing (defined in the *Global.asax* file), the browser request `http://localhost:xxxxx/Movies` is routed to the default `Index` action method of the `Movies` controller. In other words, the browser request `http://localhost:xxxxx/Movies` is effectively the same as the browser request `http://localhost:xxxxx/Movies/Index`. The result is an empty list of movies, because you haven't added any yet.

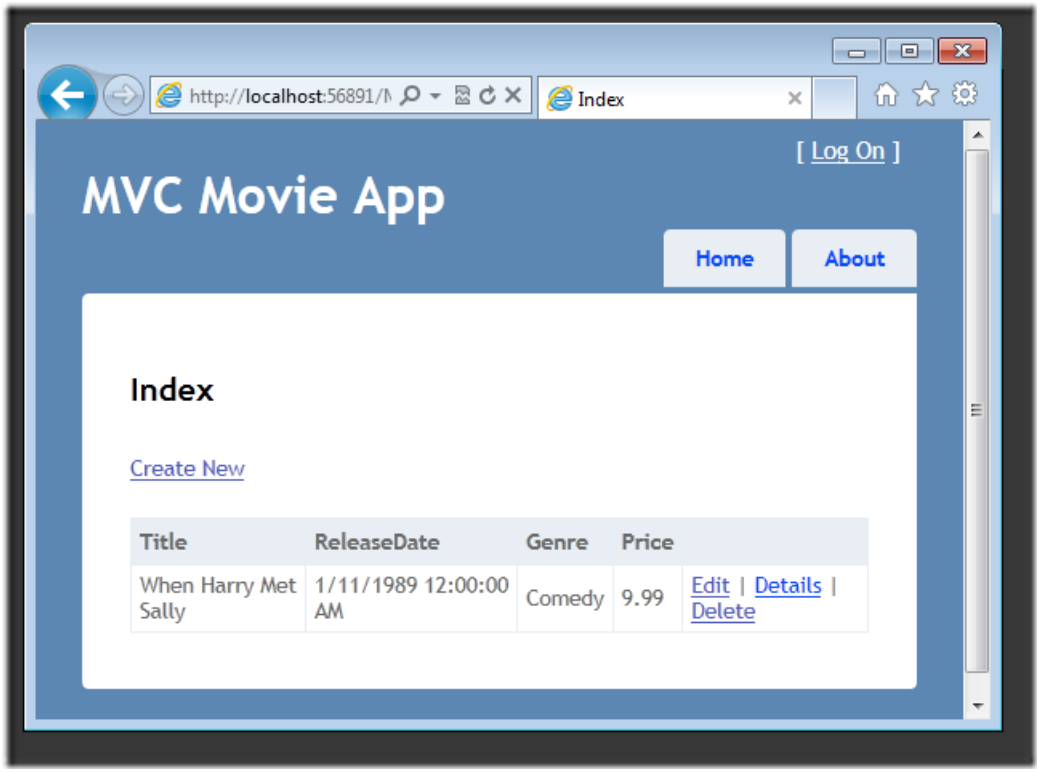


Creating a Movie

Select the **Create New** link. Enter some details about a movie and then click the **Create** button.



Clicking the **Create** button causes the form to be posted to the server, where the movie information is saved in the database. You're then redirected to the */Movies* URL, where you can see the newly created movie in the listing.



Create a couple more movie entries. Try the **Edit**, **Details**, and **Delete** links, which are all functional.

Examining the Generated Code

Open the *Controllers\MoviesController.vb* file and examine the generated `Index` method. A portion of the movie controller with the `Index` method is shown below.

```
Public Class MoviesController
        Inherits System.Web.Mvc.Controller

        Private db As MovieDBContext = New MovieDBContext

        '
        ' GET: /Movies/

        Function Index() As ViewResult
            Return View(db.Movies.ToList())
        End Function
```

The following line from the `MoviesController` class instantiates a movie database context, as described previously. You can use the movie database context to query, edit, and delete movies.

```
Private db As MovieDBContext = New MovieDBContex
```

A request to the `Movies` controller returns all the entries in the `Movies` table of the movie database and then passes the results to the `Index` view.

Strongly Typed Models and the @model Keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view template using the `ViewBag` object. The `ViewBag` is a dynamic object that provides a convenient late-bound way to pass information to a view.

ASP.NET MVC also provides the ability to pass strongly typed data or objects to a view template. This strongly typed approach enables better compile-time checking of your code and richer IntelliSense in the Visual Web

Developer editor. We're using this approach with the `MoviesController` class and *Index.vbhtml* view template.

Notice how the code creates a `List` object when it calls the `View` helper method in the `Index` action method. The code then passes this `Movies` list from the controller to the view:

```
Function Index() As ViewResult
            Return View(db.Movies.ToList())
       End Function
```

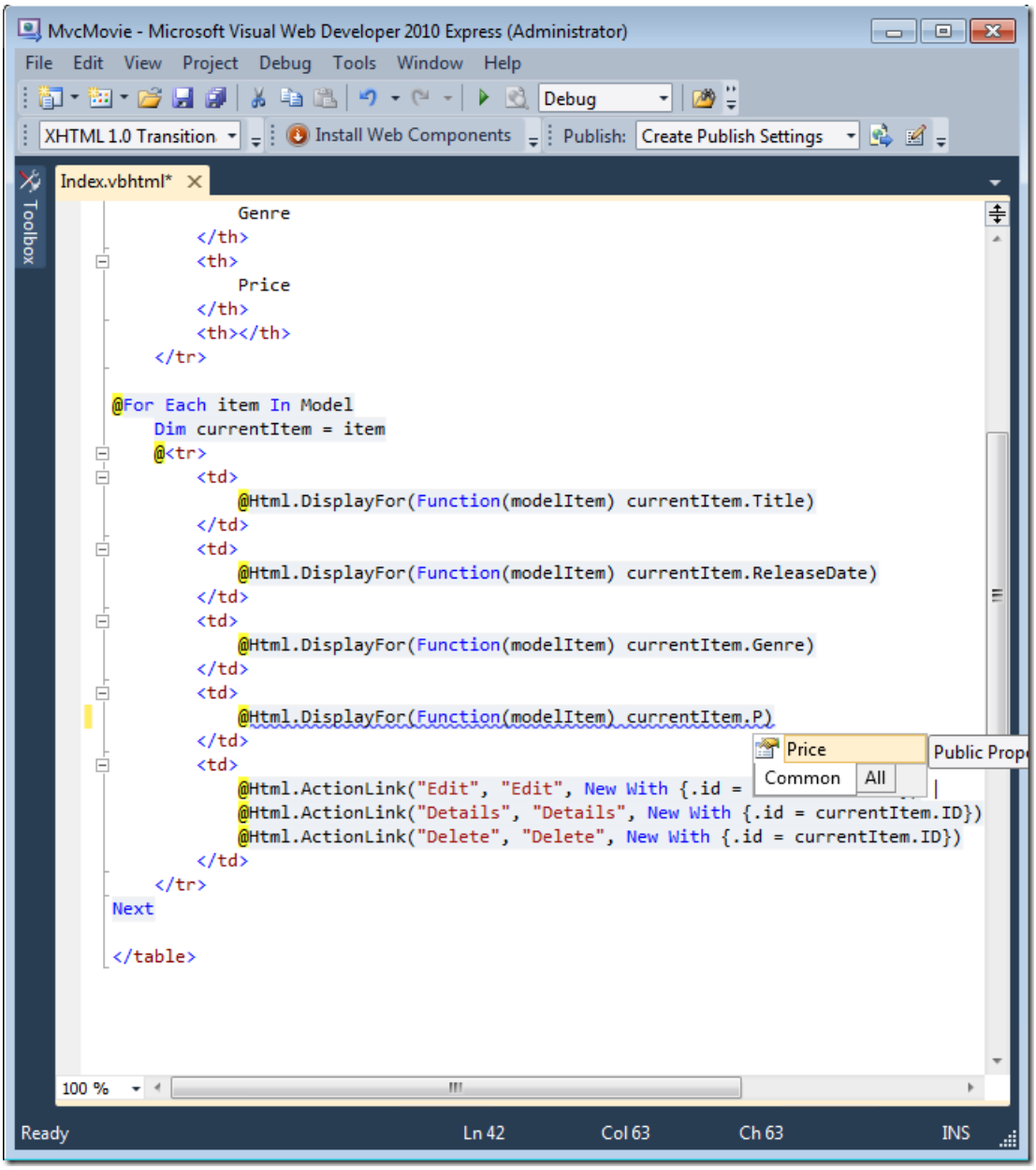
By including a `@ModelType` statement at the top of the view template file, you can specify the type of object that the view expects. When you created the movie controller, Visual Web Developer automatically included the following `@model` statement at the top of the *Index.vbhtml* file:

```
@ModelType IEnumerable(Of MvcMovie.Movie)
```

This `@ModelType` directive allows you to access the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Index.vbhtml* template, the code loops through the movies by doing a `foreach` statement over the strongly typed `Model` object:

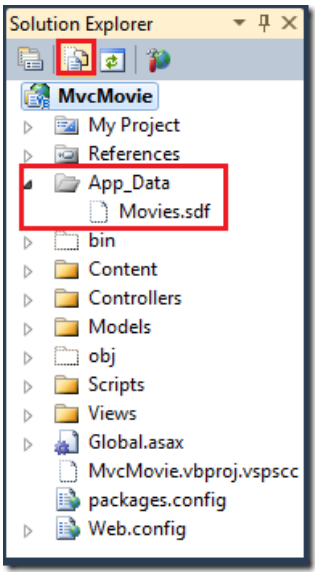
```
@For Each item In Model
    Dim currentItem = item
    @<tr>
        <td>
            @Html.DisplayFor(Function(modelItem) currentItem.Title)
        </td>
        <td>
            @Html.DisplayFor(Function(modelItem) currentItem.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(Function(modelItem) currentItem.Genre)
        </td>
        <td>
            @Html.DisplayFor(Function(modelItem) currentItem.Price)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", New With {.id = currentItem.ID}) |
            @Html.ActionLink("Details", "Details", New With {.id = currentItem.ID}) |
            @Html.ActionLink("Delete", "Delete", New With {.id = currentItem.ID})
        </td>
    </tr>
Next
```

Because the `Model` object is strongly typed (as an `IEnumerable<Movie>` object), each `item` object in the loop is typed as `Movie`. Among other benefits, this means that you get compile-time checking of the code and full IntelliSense support in the code editor:



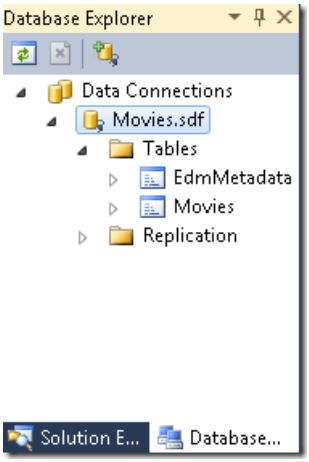
Working with SQL Server Compact

Entity Framework Code First detected that the database connection string that was provided pointed to a `Movies` database that didn't exist yet, so Code First created the database automatically. You can verify that it's been created by looking in the *App_Data* folder. If you don't see the *Movies.sdf* file, click the **Show All Files** button in the **Solution Explorer** toolbar, click the **Refresh** button, and then expand the *App_Data* folder.



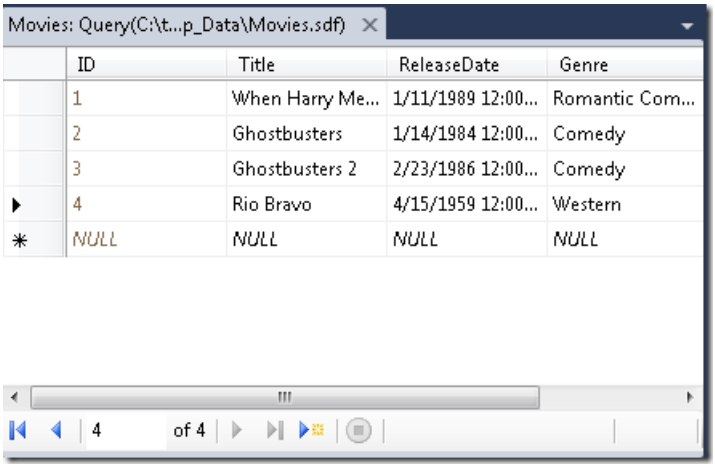
Double-click *Movies.sdf* to open **Server Explorer**. Then expand the **Tables** folder to see the tables that have been created in the database.

> **NOTE**
>
> If you get an error when you double-click *Movies.sdf*, make sure you've installed **Visual Studio 2010 SP1 Tools for SQL Server Compact 4.0**. (For links to the software, see the list of prerequisites in part 1 of this tutorial series.) If you install the release now, you'll have to close and re-open Visual Web Developer.

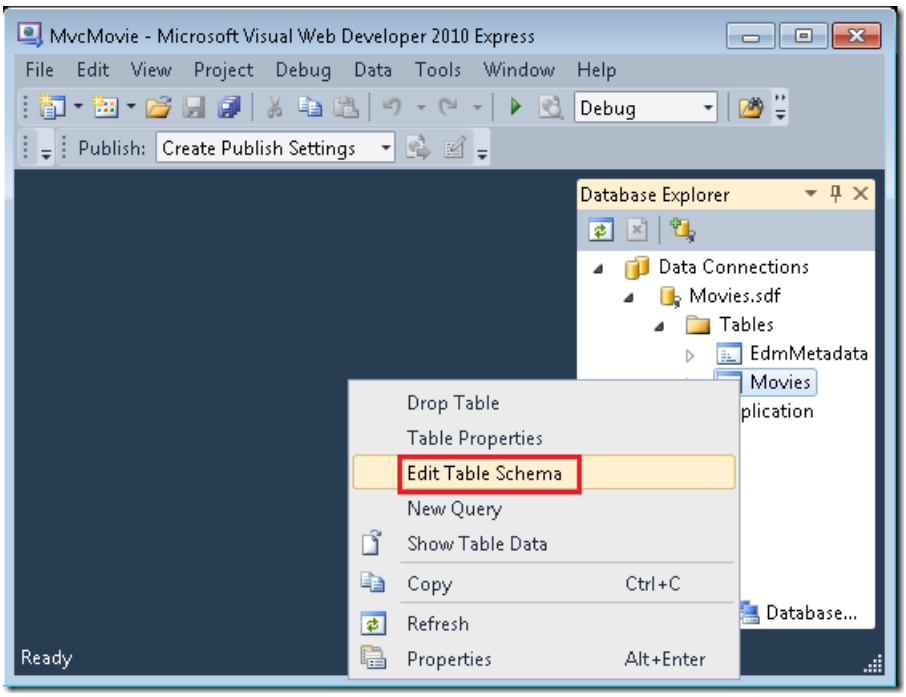


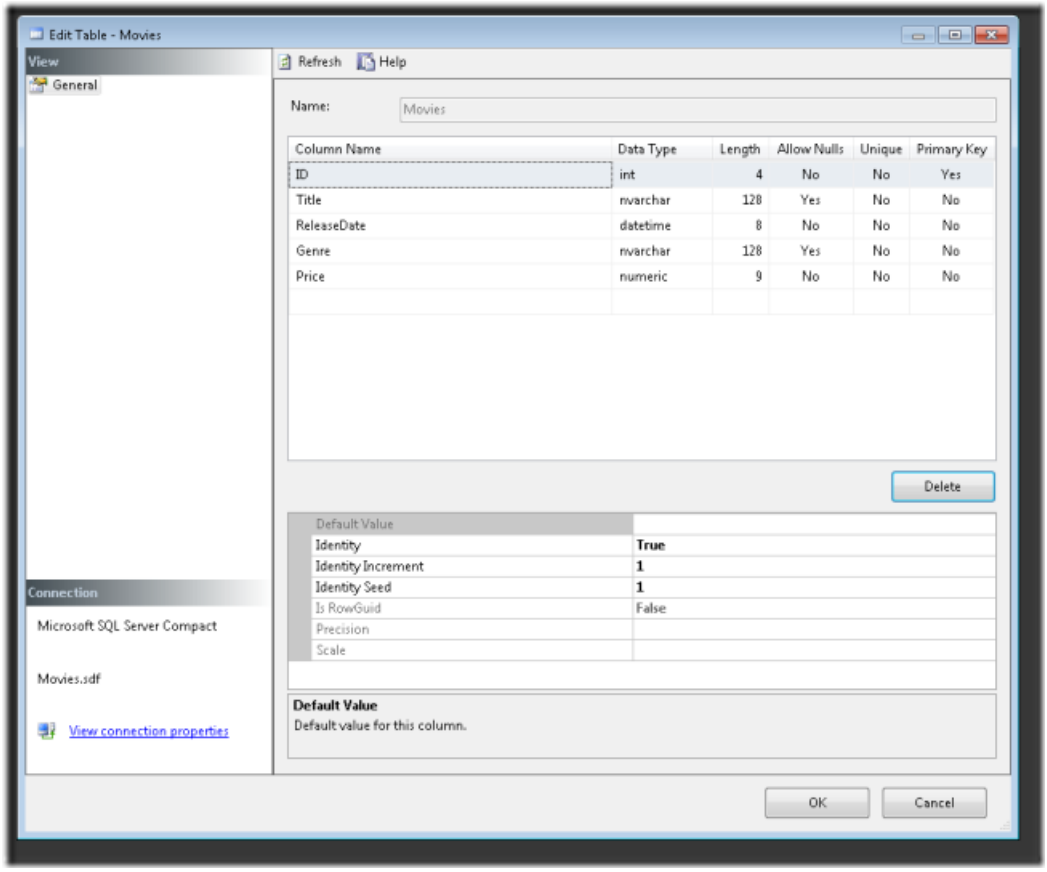
There are two tables, one for the `Movie` entity set and then the `EdmMetadata` table. The `EdmMetadata` table is used by the Entity Framework to determine when the model and the database are out of sync.

Right-click the `Movies` table and select **Show Table Data** to see the data you created.



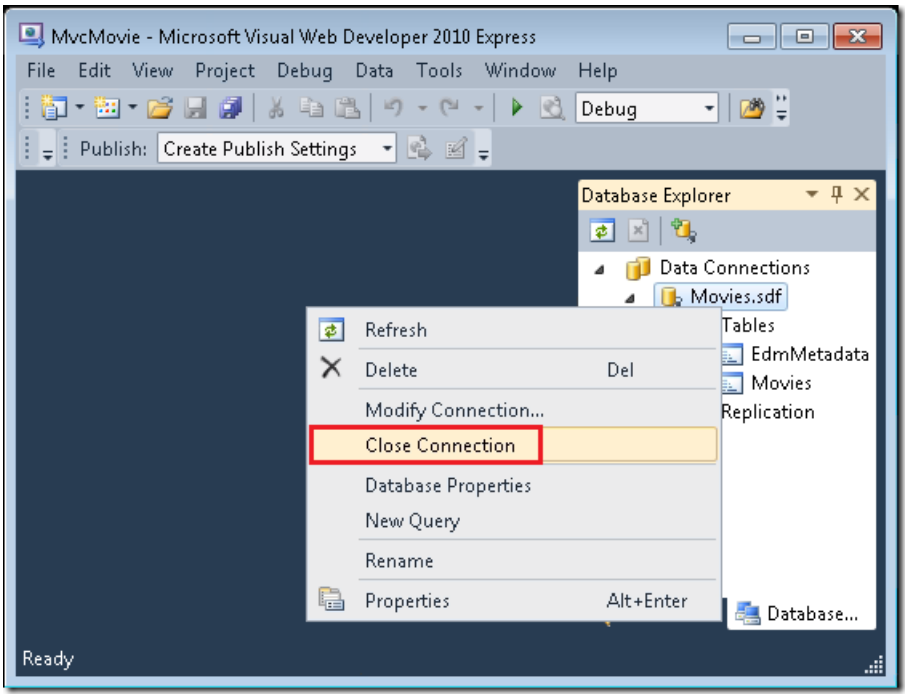
Right-click the `Movies` table and select **Edit Table Schema**.





Notice how the schema of the `Movies` table maps to the `Movie` class you created earlier. Entity Framework Code First automatically created this schema for you based on your `Movie` class.

When you're finished, close the connection. (If you don't close the connection, you might get an error the next time you run the project).



You now have the database and a simple listing page to display content from it. In the next tutorial, we'll examine the rest of the scaffolded code and add a `SearchIndex` method and a `SearchIndex` view that lets you search for movies in this database.

PREVIOUS NEXT

Examining the Edit Methods and Edit View (VB)

9/16/2020 • 13 minutes to read • Edit Online

by Rick Anderson

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

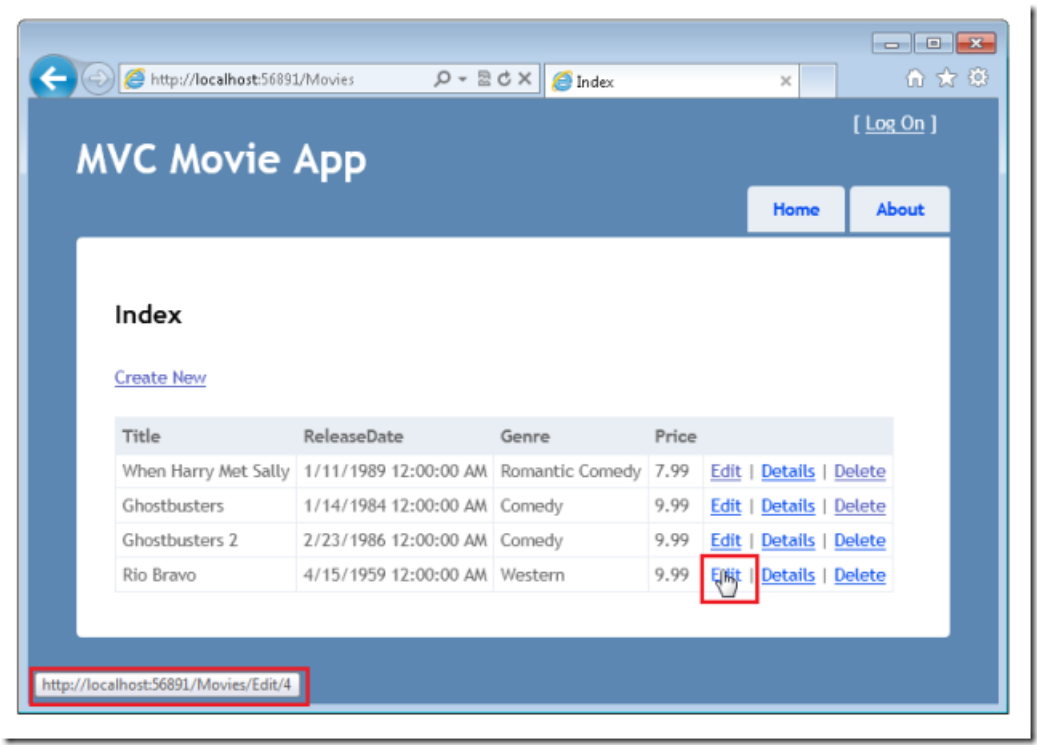
- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

A Visual Web Developer project with VB.NET source code is available to accompany this topic. Download the VB.NET version. If you prefer C#, switch to the C# version of this tutorial.

In this section, you'll examine the generated action methods and views for the movie controller. Then you'll add a custom search page.

Run the application and browse to the `Movies` controller by appending */Movies* to the URL in the address bar of your browser. Hold the mouse pointer over an **Edit** link to see the URL that it links to.



The **Edit** link was generated by the `Html.ActionLink` method in the *Views\Movies\Index.vbhtml* view:

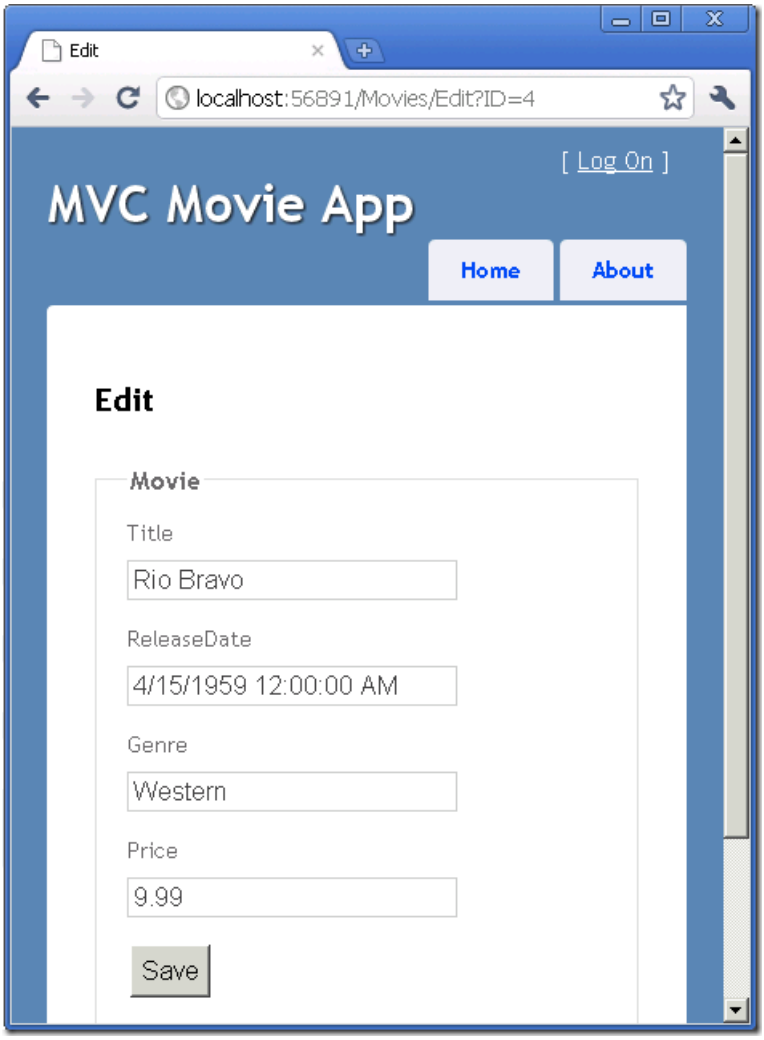
```
@Html.ActionLink("Edit", "Edit", New With {.id = currentItem.ID}) |
```

```
<td>
    @Html.ActionLink("Edit", "Edit", New With {.id = currentItem.ID}) |
    @ <Extension> Public Function ActionLink(linkText As String, actionName As String, routeValues As Object)
    @ Returns an anchor element (a element) that contains the virtual path of the specified action.
</td>
```

The `Html` object is a helper that's exposed using a property on the `WebViewPage` base class. The `ActionLink` method of the helper makes it easy to dynamically generate HTML hyperlinks that link to action methods on controllers. The first argument to the `ActionLink` method is the link text to render (for example, `<a>Edit Me</a>`). The second argument is the name of the action method to invoke. The final argument is an anonymous object that generates the route data (in this case, the ID of 4).

The generated link shown in the previous image is `http://localhost:xxxxx/Movies/Edit/4`. The default route takes the URL pattern `{controller}/{action}/{id}`. Therefore, ASP.NET translates `http://localhost:xxxxx/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `ID` equal to 4.

You can also pass action method parameters using a query string. For example, the URL `http://localhost:xxxxx/Movies/Edit?ID=4` also passes the parameter `ID` of 4 to the `Edit` action method of the `Movies` controller.



Open the `Movies` controller. The two `Edit` action methods are shown below.

```
'
' GET: /Movies/Edit/5

Function Edit(id As Integer) As ViewResult
    Dim movie As Movie = db.Movies.Find(id)
    Return View(movie)
End Function

'
' POST: /Movies/Edit/5

<HttpPost()>
Function Edit(movie As Movie) As ActionResult
    If ModelState.IsValid Then
        db.Entry(movie).State = EntityState.Modified
        db.SaveChanges()
        Return RedirectToAction("Index")
    End If

    Return View(movie)
End Function
```

Notice the second `Edit` action method is preceded by the `HttpPost` attribute. This attribute specifies that overload of the `Edit` method can be invoked only for POST requests. You could apply the `HttpGet` attribute to the first edit method, but that's not necessary because it's the default. (We'll refer to action methods that are implicitly assigned the `HttpGet` attribute as `HttpGet` methods.)

The `HttpGet` `Edit` method takes the movie ID parameter, looks up the movie using the Entity Framework `Find` method, and returns the selected movie to the Edit view. When the scaffolding system created the Edit view, it examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The following example shows the Edit view that was generated:

```
@ModelType MvcMovie.Movie

@Code
    ViewData("Title") = "Edit"
End Code

<h2>Edit</h2>

<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

@Using Html.BeginForm()
    @Html.ValidationSummary(True)
    @<fieldset>
        <legend>Movie</legend>

        @Html.HiddenFor(Function(model) model.ID)

        <div class="editor-label">
            @Html.LabelFor(Function(model) model.Title)
        </div>
        <div class="editor-field">
            @Html.EditorFor(Function(model) model.Title)
            @Html.ValidationMessageFor(Function(model) model.Title)
        </div>

        <div class="editor-label">
            @Html.LabelFor(Function(model) model.ReleaseDate)
        </div>
        <div class="editor-field">
            @Html.EditorFor(Function(model) model.ReleaseDate)
            @Html.ValidationMessageFor(Function(model) model.ReleaseDate)
        </div>

        <div class="editor-label">
            @Html.LabelFor(Function(model) model.Genre)
        </div>
        <div class="editor-field">
            @Html.EditorFor(Function(model) model.Genre)
            @Html.ValidationMessageFor(Function(model) model.Genre)
        </div>

        <div class="editor-label">
            @Html.LabelFor(Function(model) model.Price)
        </div>
        <div class="editor-field">
            @Html.EditorFor(Function(model) model.Price)
            @Html.ValidationMessageFor(Function(model) model.Price)
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
End Using

<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Notice how the view template has a `@ModelType MvcMovie.Models.Movie` statement at the top of the file — this specifies that the view expects the model for the view template to be of type `Movie`.

The scaffolded code uses several *helper methods* to streamline the HTML markup. The `Html.LabelFor` helper displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The `Html.EditorFor` helper displays an

HTML `<input>` element. The `Html.ValidationMessageFor` helper displays any validation messages associated with that property.

Run the application and navigate to the */Movies* URL. Click an **Edit** link. In the browser, view the source for the page. The HTML in the page looks like the following example. (The menu markup was excluded for clarity.)

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Edit</title>
    <link href="/Content/Site.css" rel="stylesheet" type="text/css" />
    <script src="/Scripts/jquery-1.5.1.min.js" type="text/javascript"></script>
    <script src="/Scripts/modernizr-1.7.min.js" type="text/javascript"></script>
</head>
<body>
    <div class="page">
        <header>
            <div id="title">
                <h1>MVC Movie App</h1>
            </div>
           ...
        </header>
        <section id="main">

<h2>Edit</h2>

<script src="/Scripts/jquery.validate.min.js" type="text/javascript"></script>
<script src="/Scripts/jquery.validate.unobtrusive.min.js" type="text/javascript"></script>

<form action="/Movies/Edit/4" method="post">    <fieldset>
        <legend>Movie</legend>

        <input data-val="true" data-val-number="The field ID must be a number."
    data-val-required="The ID field is required." id="ID" name="ID" type="hidden" value="4" />

        <div class="editor-label">
            <label for="Title">Title</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" id="Title" name="Title" type="text" value="Rio Bravo" />
            <span class="field-validation-valid" data-valmsg-for="Title" data-valmsg-replace="true"></span>
        </div>

        <div class="editor-label">
            <label for="ReleaseDate">ReleaseDate</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" data-val="true" data-val-required="The ReleaseDate field is
required."
    id="ReleaseDate" name="ReleaseDate" type="text" value="4/15/1959 12:00:00 AM" />
            <span class="field-validation-valid" data-valmsg-for="ReleaseDate" data-valmsg-replace="true">
</span>
        </div>

        <div class="editor-label">
            <label for="Genre">Genre</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" id="Genre" name="Genre" type="text" value="Western" />
            <span class="field-validation-valid" data-valmsg-for="Genre" data-valmsg-replace="true">></span>
        </div>

        <div class="editor-label">
            <label for="Price">Price</label>
        </div>
        <div class="editor-field">
```

```
            <input class="text-box single-line" data-val="true" data-val-number="The field Price must be a
number."
    data-val-required="The Price field is required." id="Price" name="Price" type="text" value="9.99" />
            <span class="field-validation-valid" data-valmsg-for="Price" data-valmsg-replace="true"></span>
        </div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
</form>
<div>
    <a href="/Movies">Back to List</a>
</div>

        </section>
        <footer>
        </footer>
    </div>
</body>
</html>
```

The `<input>` elements are in an HTML `<form>` element whose `action` attribute is set to post to the */Movies/Edit* URL. The form data will be posted to the server when the **Edit** button is clicked.

Processing the POST Request

The following listing shows the `HttpPost` version of the `Edit` action method.

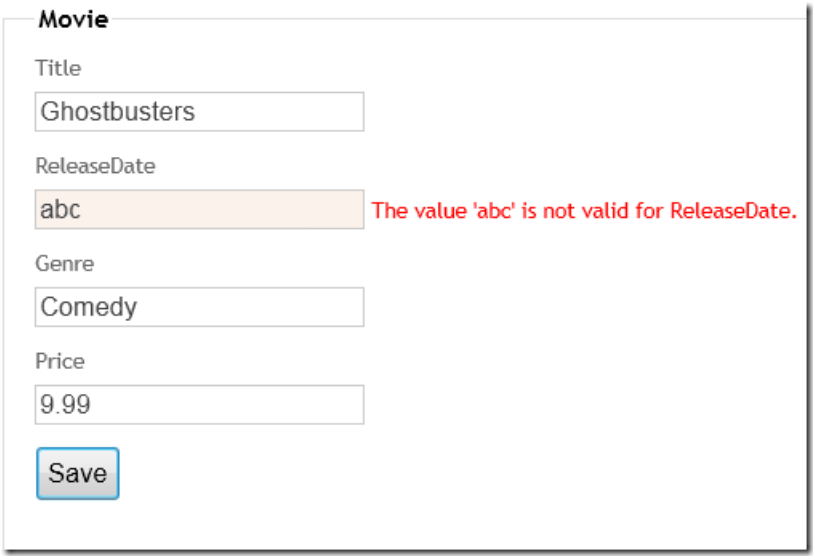
```
'
' POST: /Movies/Edit/5

<HttpPost()>
Function Edit(movie As Movie) As ActionResult
    If ModelState.IsValid Then
        db.Entry(movie).State = EntityState.Modified
        db.SaveChanges()
        Return RedirectToAction("Index")
    End If

    Return View(movie)
End Function
```

The ASP.NET framework model binder takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` check in the code verifies that the data submitted in the form can be used to modify a `Movie` object. If the data is valid, the code saves the movie data to the `Movies` collection of the `MovieDBContext` instance. The code then saves the new movie data to the database by calling the `SaveChanges` method of `MovieDBContext`, which persists changes to the database. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which causes the updated movie to be displayed in the listing of movies.

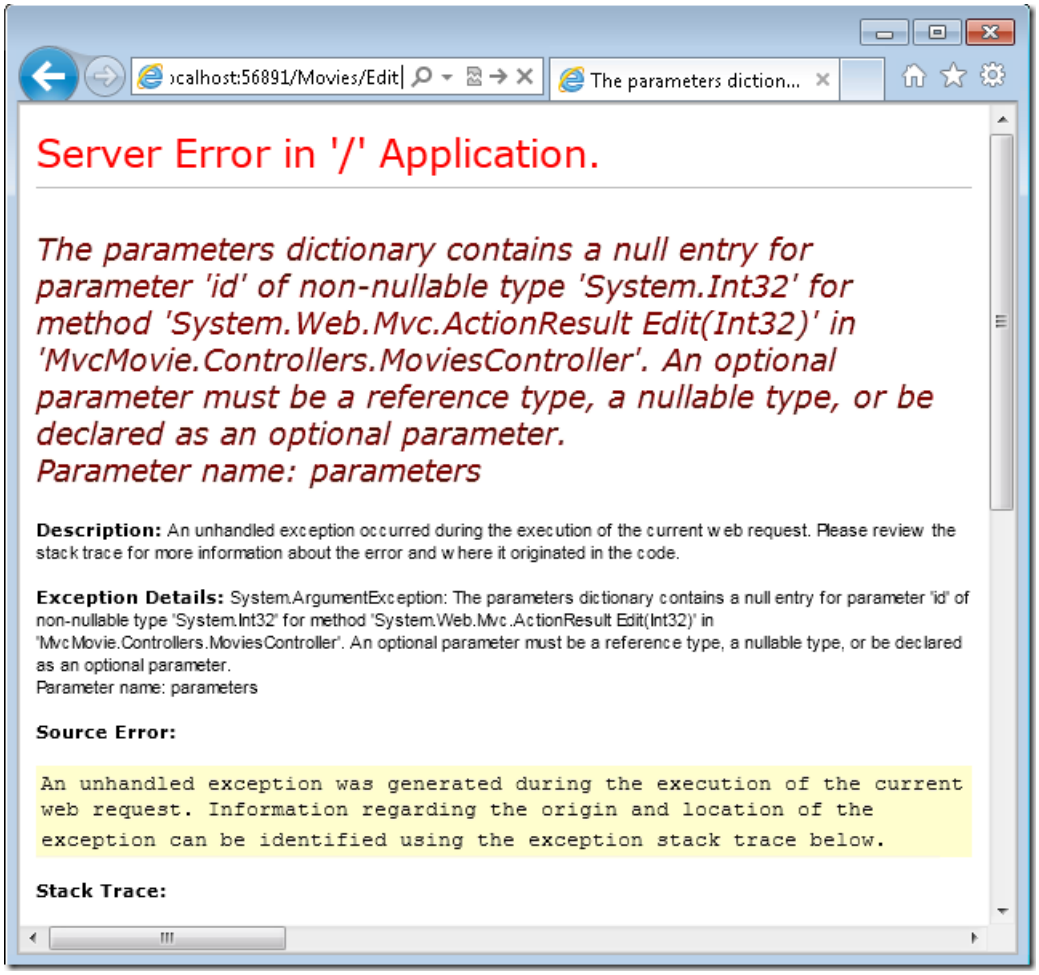
If the posted values aren't valid, they are redisplayed in the form. The `Html.ValidationMessageFor` helpers in the *Edit.vbhtml* view template take care of displaying appropriate error messages.



Note about locales If you normally work with a locale other than English, see Supporting ASP.NET MVC 3 Validation with Non-English Locales.

Making the Edit Method More Robust

The `HttpGet` `Edit` method generated by the scaffolding system doesn't check that the ID that's passed to it is valid. If a user removes the ID segment from the URL (`http://localhost:xxxxx/Movies/Edit`), the following error is displayed:



A user could also pass an ID that doesn't exist in the database, such as `http://localhost:xxxxx/Movies/Edit/1234`. You can make two changes to the `HttpGet` `Edit` action method to address this limitation. First, change the `ID` parameter to have a default value of zero when an ID isn't explicitly passed. You can also check that the `Find`

method actually found a movie before returning the movie object to the view template. The updated `Edit` method is shown below.

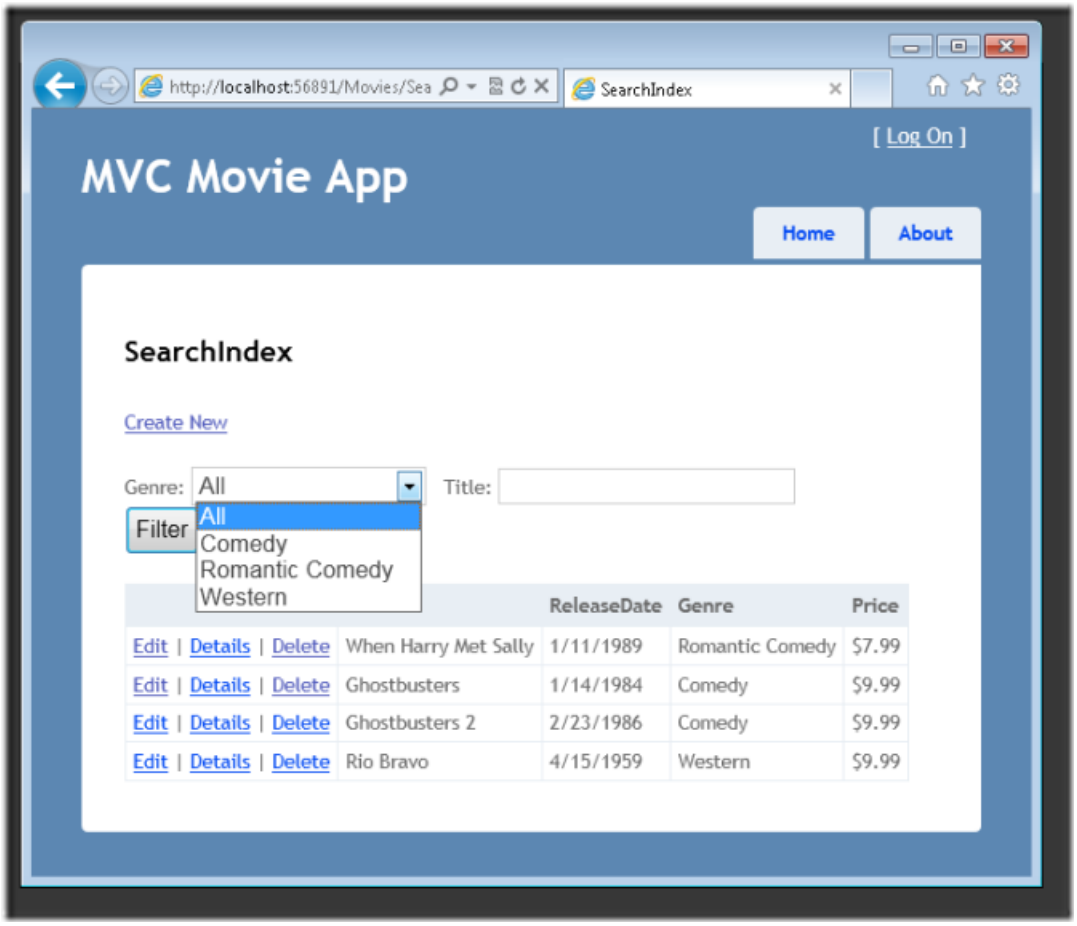
```
Public Function Edit(Optional ByVal id As Integer = 0) As ActionResult
    Dim movie As Movie = db.Movies.Find(id)
    If movie Is Nothing Then
        Return HttpNotFound()
    End If
    Return View(movie)
End Function
```

If no movie is found, the `HttpNotFound` method is called.

All the `HttpGet` methods follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the model to the view. The `Create` method passes an empty movie object to the Create view. All the methods that create, edit, delete, or otherwise modify data do so in the `HttpPost` overload of the method. Modifying data in an HTTP GET method is a security risk, as described in the blog post entry ASP.NET MVC Tip #46 – Don't use Delete Links because they create Security Holes. Modifying data in a GET method also violates HTTP best practices and the architectural REST pattern, which specifies that GET requests should not change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects.

Adding a Search Method and Search View

In this section you'll add a `SearchIndex` action method that lets you search movies by genre or name. This will be available using the */Movies/SearchIndex* URL. The request will display an HTML form that contains input elements that a user can fill in in order to search for a movie. When a user submits the form, the action method will get the search values posted by the user and use the values to search the database.



Displaying the SearchIndex Form

Start by adding a `SearchIndex` action method to the existing `MoviesController` class. The method will return a view that contains an HTML form. Here's the code:

```
Public Function SearchIndex(ByVal searchString As String) As ActionResult
    Dim movies = From m In db.Movies
                 Select m

    If Not String.IsNullOrEmpty(searchString) Then
        movies = movies.Where(Function(s) s.Title.Contains(searchString))
    End If
    Return View(movies)
End Function
```

The first line of the `SearchIndex` method creates the following LINQ query to select the movies:

```
Dim movies = From m In db.Movies    Select m
```

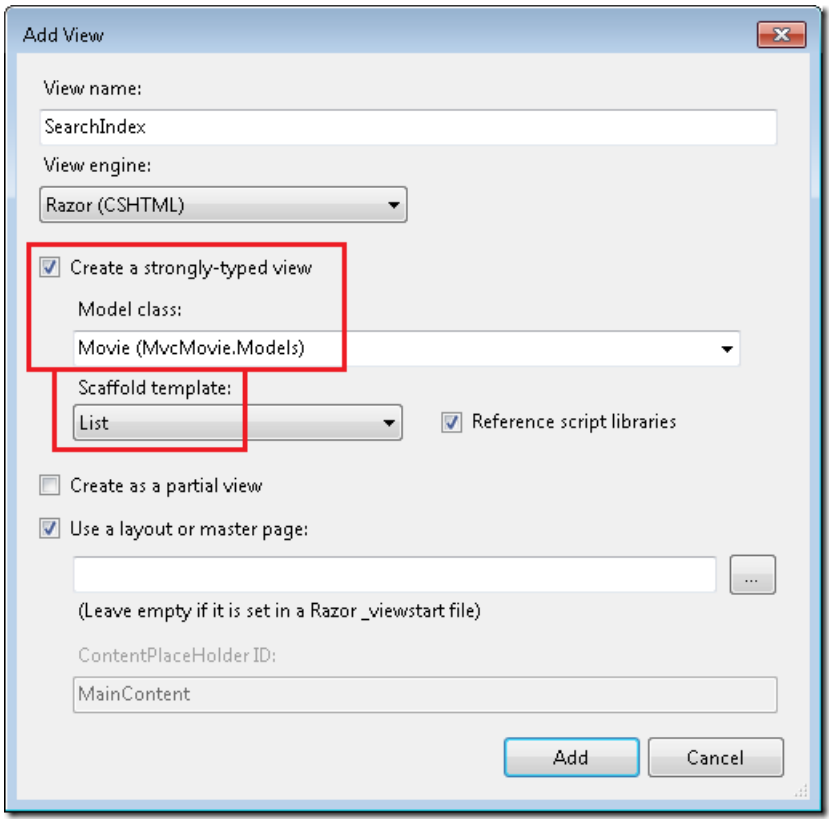
The query is defined at this point, but hasn't yet been run against the data store.

If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string, using the following code:

If Not String.IsNullOrEmpty(searchString) Then
movies = movies.Where(Function(s) s.Title.Contains(searchString))
End If

LINQ queries are not executed when they are defined or when they are modified by calling a method such as `Where` or `OrderBy`. Instead, query execution is deferred, which means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToList` method is called. In the `SearchIndex` sample, the query is executed in the SearchIndex view. For more information about deferred query execution, see Query Execution.

Now you can implement the `SearchIndex` view that will display the form to the user. Right-click inside the `SearchIndex` method and then click **Add View**. In the **Add View** dialog box, specify that you're going to pass a `Movie` object to the view template as its model class. In the **Scaffold template** list, choose **List**, then click **Add**.



When you click the **Add** button, the *Views\Movies\SearchIndex.vbhtml* view template is created. Because you selected **List** in the **Scaffold template** list, Visual Web Developer automatically generated (scaffolded) some default content in the view. The scaffolding created an HTML form. It examined the `Movie` class and created code to render `<label>` elements for each property of the class. The listing below shows the Create view that was generated:

```
@ModelType IEnumerable(Of MvcMovie.Movie)

@Code
    ViewData("Title") = "SearchIndex"
End Code

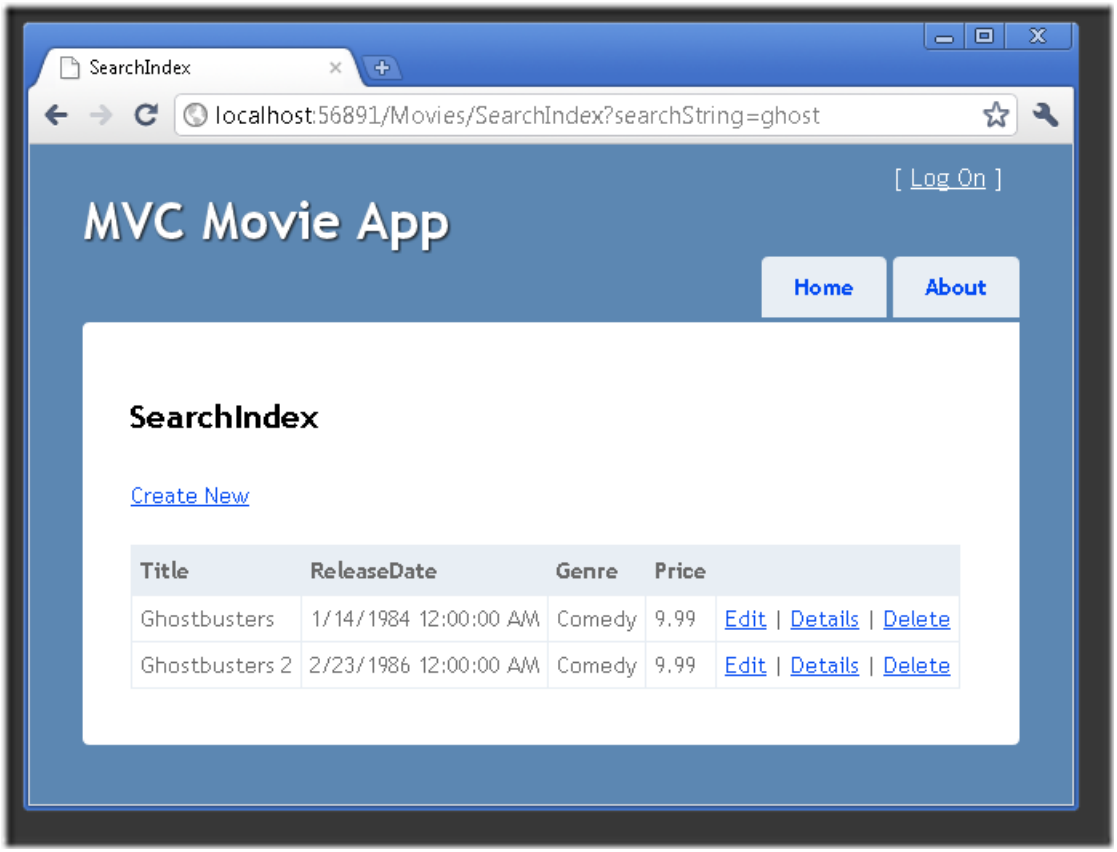
<h2>SearchIndex</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            Title
        </th>
        <th>
            ReleaseDate
        </th>
        <th>
            Genre
        </th>
        <th>
            Price
        </th>
        <th></th>
    </tr>

@For Each item In Model
    Dim currentItem = item
    @<tr>
        <td>
            @Html.DisplayFor(Function(modelItem) currentItem.Title)
        </td>
        <td>
            @Html.DisplayFor(Function(modelItem) currentItem.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(Function(modelItem) currentItem.Genre)
        </td>
        <td>
            @Html.DisplayFor(Function(modelItem) currentItem.Price)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", New With {.id = currentItem.ID}) |
            @Html.ActionLink("Details", "Details", New With {.id = currentItem.ID}) |
            @Html.ActionLink("Delete", "Delete", New With {.id = currentItem.ID})
        </td>
    </tr>
Next

</table>
```

Run the application and navigate to */Movies/SearchIndex*. Append a query string such as `?searchString=ghost` to the URL. The filtered movies are displayed.



If you change the signature of the `SearchIndex` method to have a parameter named `id`, the `id` parameter will match the `{id}` placeholder for the default routes set in the *Global.asax* file.

```
{controller}/{action}/{id}
```

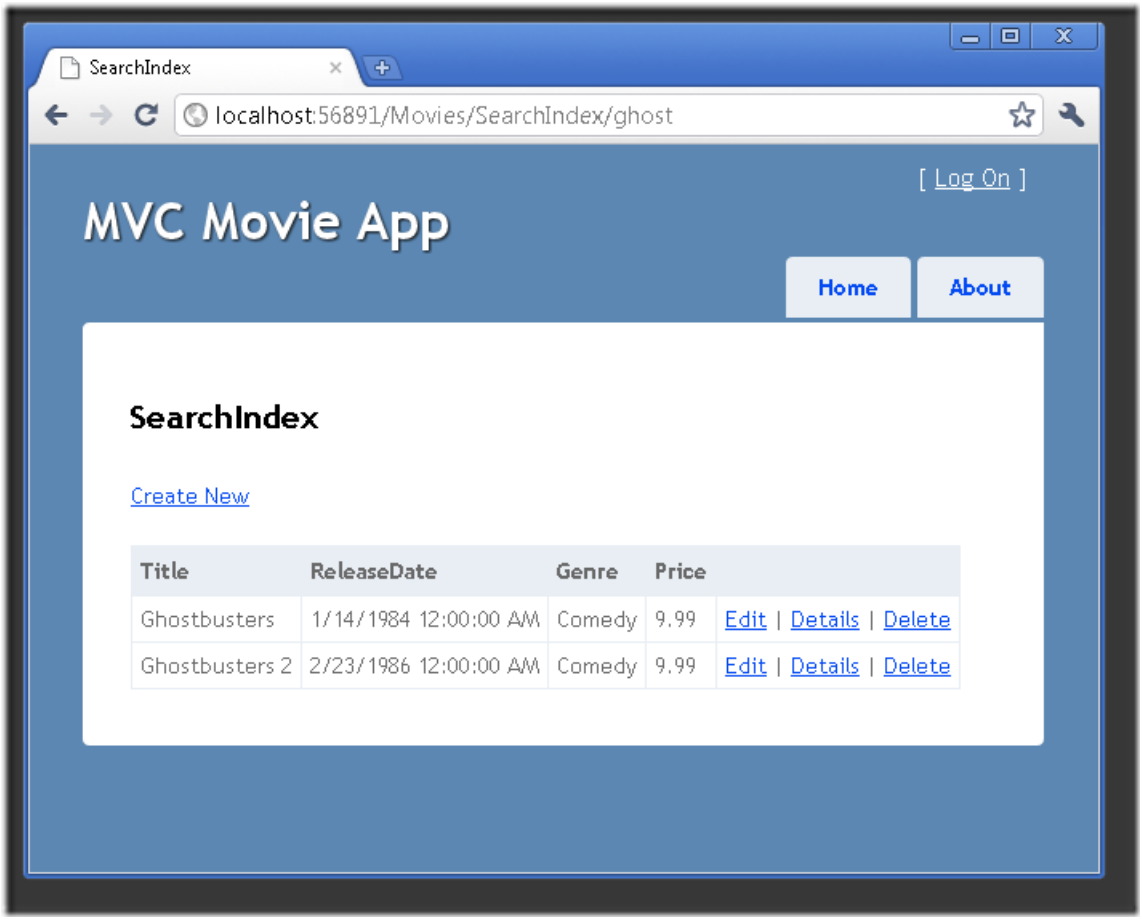
The modified `SearchIndex` method would look as follows:

```
Public Function SearchIndex(ByVal id As String) As ActionResult
Dim searchString As String = id
Dim movies = From m In db.Movies
             Select m

If Not String.IsNullOrEmpty(searchString) Then
    movies = movies.Where(Function(s) s.Title.Contains(searchString))
End If

Return View(movies)
End Function
```

You can now pass the search title as route data (a URL segment) instead of as a query string value.



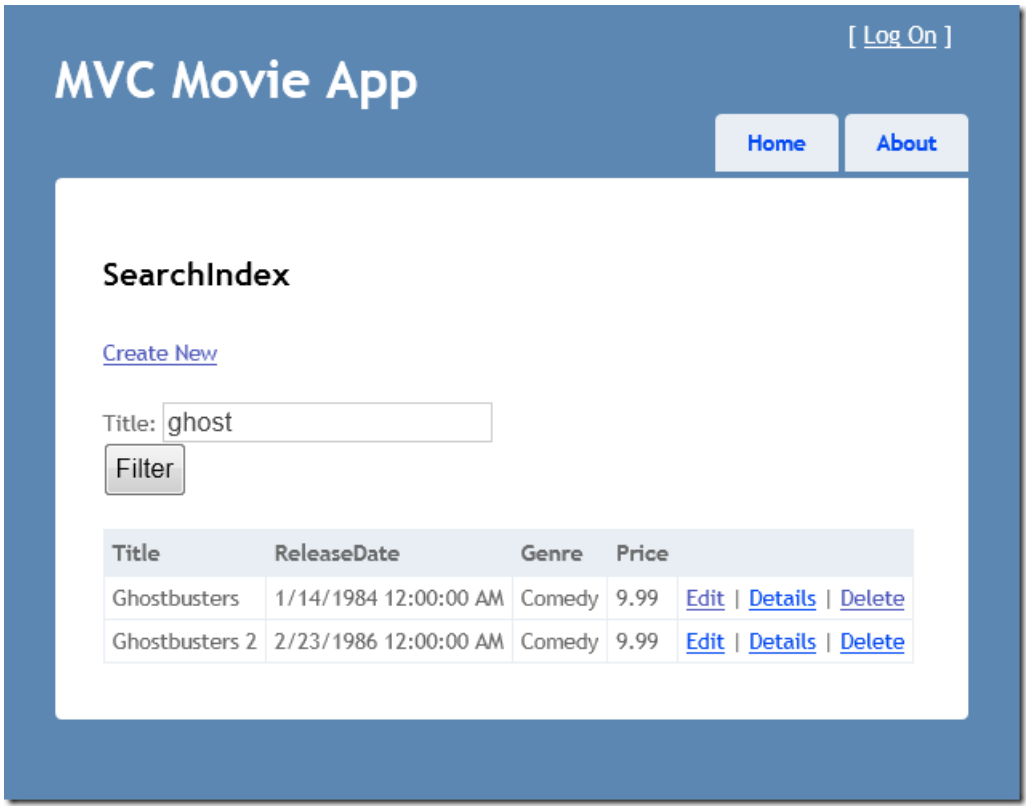
However, you can't expect users to modify the URL every time they want to search for a movie. So now you you'll add UI to help them filter movies. If you changed the signature of the `SearchIndex` method to test how to pass the route-bound ID parameter, change it back so that your `SearchIndex` method takes a string parameter named `searchString`:

Open the *Views\Movies\SearchIndex.vbhtml* file, and just after `@Html.ActionLink("Create New", "Create")`, add the following:

```
@Code
    ViewData("Title") = "SearchIndex"
    Using (Html.BeginForm())
         @<p> Title: @Html.TextBox("SearchString")
         <input type="submit" value="Filter" /></p>
        End Using
End Code
```

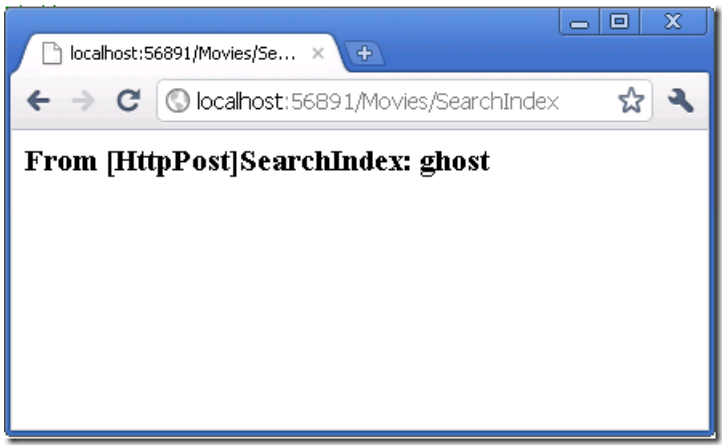
The `Html.BeginForm` helper creates an opening `<form>` tag. The `Html.BeginForm` helper causes the form to post to itself when the user submits the form by clicking the **Filter** button.

Run the application and try searching for a movie.



There's no `HttpPost` overload of the `SearchIndex` method. You don't need it, because the method isn't changing the state of the application, just filtering data. If you added the following `HttpPost` `SearchIndex` method, the action invoker would match the `HttpPost` `SearchIndex` method, and the `HttpPost` `SearchIndex` method would run as shown in the image below.

```
<HttpPost()>
 Public Function SearchIndex(ByVal fc As FormCollection, ByVal searchString As String) As String
     Return "<h3> From [HttpPost]SearchIndex: " & searchString & "</h3>"
 End Function
```



Adding Search by Genre

If you added the `HttpPost` version of the `SearchIndex` method, delete it now.

Next, you'll add a feature to let users search for movies by genre. Replace the `SearchIndex` method with the following code:

```
Public Function SearchIndex(ByVal movieGenre As String, ByVal searchString As String) As ActionResult
    Dim GenreLst = New List(Of String)()

    Dim GenreQry = From d In db.Movies
                   Order By d.Genre
                   Select d.Genre
    GenreLst.AddRange(GenreQry.Distinct())
    ViewBag.movieGenre = New SelectList(GenreLst)

    Dim movies = From m In db.Movies
                 Select m

    If Not String.IsNullOrEmpty(searchString) Then
        movies = movies.Where(Function(s) s.Title.Contains(searchString))
    End If

    If String.IsNullOrEmpty(movieGenre) Then
        Return View(movies)
    Else
        Return View(movies.Where(Function(x) x.Genre = movieGenre))
    End If

End Function
```

This version of the `SearchIndex` method takes an additional parameter, namely `movieGenre`. The first few lines of code create a `List` object to hold movie genres from the database.

The following code is a LINQ query that retrieves all the genres from the database.

```
Dim GenreQry = From d In db.Movies
                   Order By d.Genre
                   Select d.Genre
```

The code uses the `AddRange` method of the generic `List` collection to add all the distinct genres to the list. (Without the `Distinct` modifier, duplicate genres would be added — for example, comedy would be added twice in our sample). The code then stores the list of genres in the `ViewBag` object.

The following code shows how to check the `movieGenre` parameter. If it's not empty the code further constrains the movies query to limit the selected movies to the specified genre.

```
If String.IsNullOrEmpty(movieGenre) Then
        Return View(movies)
    Else
        Return View(movies.Where(Function(x) x.Genre = movieGenre))
    End If
```

Adding Markup to the SearchIndex View to Support Search by Genre

Add an `Html.DropDownList` helper to the *Views\Movies\SearchIndex.vbhtml* file, just before the `TextBox` helper. The completed markup is shown below:

```
<p>
    @Html.ActionLink("Create New", "Create")
    @Code
    ViewData("Title") = "SearchIndex"
    Using (Html.BeginForm())
         @<p> Genre: @Html.DropDownList("movieGenre", "All")
         Title: @Html.TextBox("SearchString")
         <input type="submit" value="Filter" /></p>
        End Using
End Code
</p>
```

Run the application and browse to */Movies/SearchIndex*. Try a search by genre, by movie name, and by both criteria.

In this section you examined the CRUD action methods and views generated by the framework. You created a search action method and view that let users search by movie title and genre. In the next section, you'll look at how to add a property to the `Movie` model and how to add an initializer that will automatically create a test database.

PREVIOUS NEXT

Adding a New Field to the Movie Model and Database Table (VB)

2/20/2020 • 6 minutes to read • Edit Online

by Rick Anderson

> This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:
>
> - Visual Studio Web Developer Express SP1 prerequisites
> - ASP.NET MVC 3 Tools Update
> - SQL Server Compact 4.0(runtime + tools support)
>
> If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.
>
> A Visual Web Developer project with VB.NET source code is available to accompany this topic. Download the VB.NET version. If you prefer C#, switch to the C# version of this tutorial.

In this section you'll make some changes to the model classes and learn how you can update the database schema to match the model changes.

Adding a Rating Property to the Movie Model

Start by adding a new `Rating` property to the existing `Movie` class. Open the *Movie.cs* file and add the `Rating` property like this one:

```
Public Property Rating() As String
```

The complete `Movie` class now looks like the following code:

```
Public Class Movie
    Public Property ID() As Integer
    Public Property Title() As String
    Public Property ReleaseDate() As Date
    Public Property Genre() As String
    Public Property Price() As Decimal
    Public Property Rating() As String
End Class
```

Recompile the application using the **Debug** >**Build Movie** menu command.

Now that you've updated the `Model` class, you also need to update the *\Views\Movies\Index.vbhtml* and *\Views\Movies\Create.vbhtml* view templates in order to support the new `Rating` property.

Open the *\Views\Movies\Index.vbhtml* file and add a `<th>Rating</th>` column heading just after the **Price** column. Then add a `<td>` column near the end of the template to render the `@item.Rating` value. Below is what the

updated *Index.vbhtml* view template looks like:

```
<table>
    <tr>
        <th>            Title        </th>
        <th>            ReleaseDate        </th>
        <th>            Genre        </th>
        <th>            Price        </th>
        <th>Rating</th>
        <th></th>
    </tr>

@For Each item In Model
    Dim currentItem = item
    @<tr>
        <td>
            @Html.DisplayFor(Function(modelItem) currentItem.Title)
        </td>
        <td>
            @Html.DisplayFor(Function(modelItem) currentItem.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(Function(modelItem) currentItem.Genre)
        </td>
        <td>
            @Html.DisplayFor(Function(modelItem) currentItem.Price)
        </td>
         <td>
            @Html.DisplayFor(Function(modelItem) currentItem.Rating)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", New With {.id = currentItem.ID}) |
            @Html.ActionLink("Details", "Details", New With {.id = currentItem.ID}) |
            @Html.ActionLink("Delete", "Delete", New With {.id = currentItem.ID})
        </td>
    </tr>
Next

</table>
```

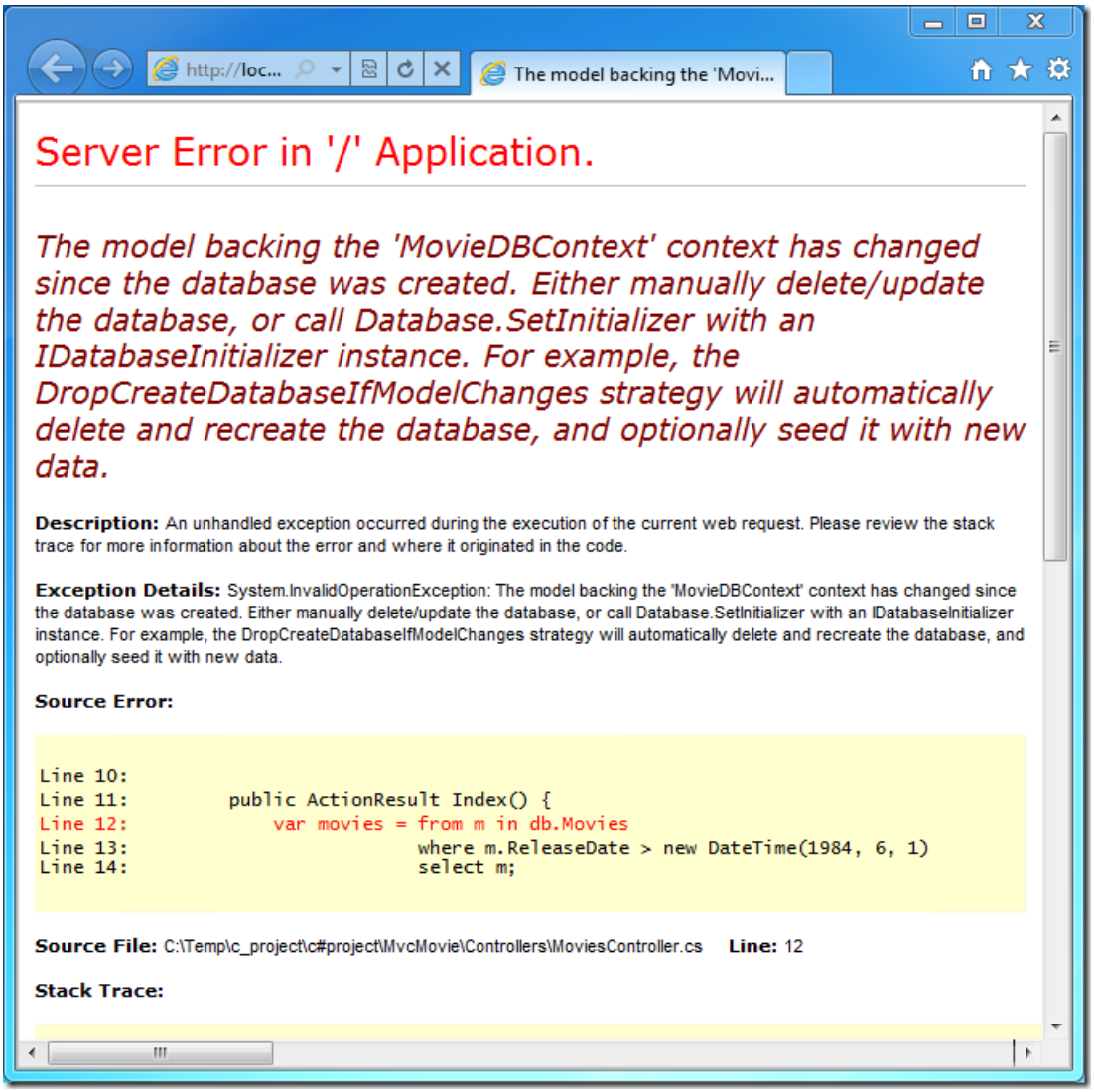
Next, open the *\Views\Movies\Create.vbhtml* file and add the following markup near the end of the form. This renders a text box so that you can specify a rating when a new movie is created.

```
<div class="editor-label">
     @Html.LabelFor(Function(model) model.Rating)
 </div>
 <div class="editor-field">
     @Html.EditorFor(Function(model) model.Rating)
     @Html.ValidationMessageFor(Function(model) model.Rating)
 </div>
```

Managing Model and Database Schema Differences

You've now updated the application code to support the new `Rating` property.

Now run the application and navigate to the */Movies* URL. When you do this, though, you'll see the following error:



You're seeing this error because the updated `Movie` model class in the application is now different than the schema of the `Movie` table of the existing database. (There's no `Rating` column in the database table.)

By default, when you use Entity Framework Code First to automatically create a database, as you did earlier in this tutorial, Code First adds a table to the database to help track whether the schema of the database is in sync with the model classes it was generated from. If they aren't in sync, the Entity Framework throws an error. This makes it easier to track down issues at development time that you might otherwise only find (by obscure errors) at run time. The sync-checking feature is what causes the error message to be displayed that you just saw.

There are two approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient when doing active development on a test database, because it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you *don't* want to use this approach on a production database!
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.

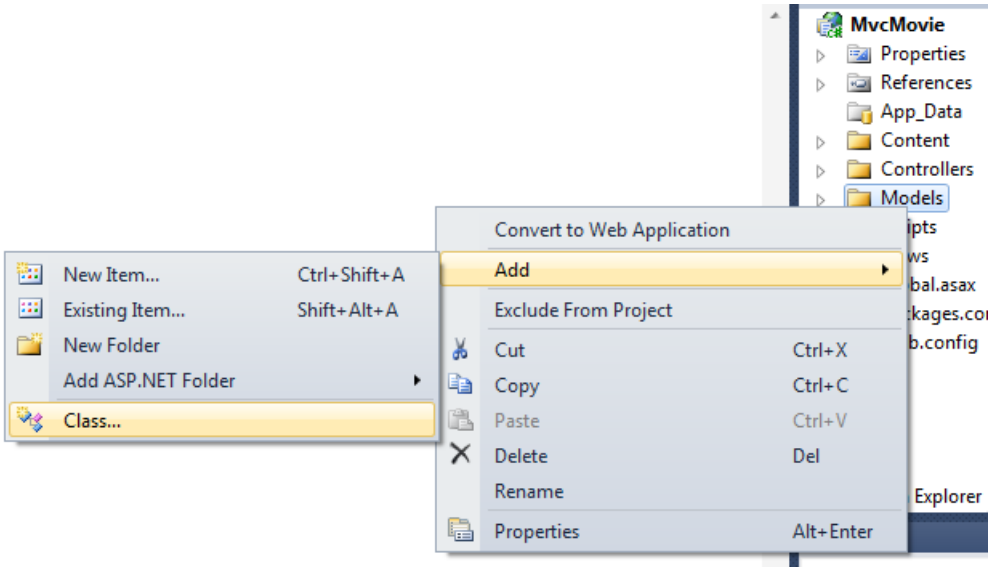
For this tutorial, we'll use the first approach — you'll have the Entity Framework Code First automatically re-create the database anytime the model changes.

Automatically Re-Creating the Database on Model Changes

Let's update the application so that Code First automatically drops and re-creates the database anytime you change the model for the application.

> **NOTE**
>
> **Warning** You should enable this approach of automatically dropping and re-creating the database only when you're using a development or test database, and *never* on a production database that contains real data. Using it on a production server can lead to data loss.

In **Solution Explorer**, right click the *Models* folder, select **Add**, and then select **Class**.



Name the class "MovieInitializer". Update the `MovieInitializer` class to contain the following code:

```
using System;
Imports System
Imports System.Collections.Generic
Imports System.Data.Entity

Namespace MvcMovie.Models
    Public Class MovieInitializer
        Inherits DropCreateDatabaseIfModelChanges(Of MovieDBContext)
        Protected Overrides Sub Seed(ByVal context As MovieDBContext)
            Dim movies = New List(Of Movie) From {
             New Movie With {.Title = "When Harry Met Sally", .ReleaseDate = Date.Parse("1989-1-11"), .Genre =
"Romantic Comedy", .Rating = "R", .Price = 7.99D},
             New Movie With {.Title = "Ghostbusters ", .ReleaseDate = Date.Parse("1984-3-13"), .Genre =
"Comedy", .Rating = "R", .Price = 8.99D},
             New Movie With {.Title = "Ghostbusters 2", .ReleaseDate = Date.Parse("1986-2-23"), .Genre =
"Comedy", .Rating = "R", .Price = 9.99D},
             New Movie With {.Title = "Rio Bravo", .ReleaseDate = Date.Parse("1959-4-15"), .Genre = "Western",
.Rating = "R", .Price = 3.99D}}

            movies.ForEach(Function(d) context.Movies.Add(d))
        End Sub
    End Class
End Namespace
```

The `MovieInitializer` class specifies that the database used by the model should be dropped and automatically re-created if the model classes ever change. The code includes a `Seed` method to specify some default data to automatically add to the database any time it's created (or re-created). This provides a useful way to populate the database with some sample data, without requiring you to manually populate it each time you make a model change.

Now that you've defined the `MovieInitializer` class, you'll want to wire it up so that each time the application runs, it checks whether the model classes are different from the schema in the database. If they are, you can run

the initializer to re-create the database to match the model and then populate the database with the sample data.

Open the *Global.asax* file that's at the root of the `MvcMovies` project:

The *Global.asax* file contains the class that defines the entire application for the project, and contains an `Application_Start` event handler that runs when the application first starts.

Find the `Application_Start` method and add a call to `Database.SetInitializer` at the beginning of the method, as shown below:

```
Sub Application_Start()
        System.Data.Entity.Database.SetInitializer(Of MovieDBContext)(New MvcMovie.Models.MovieInitializer())

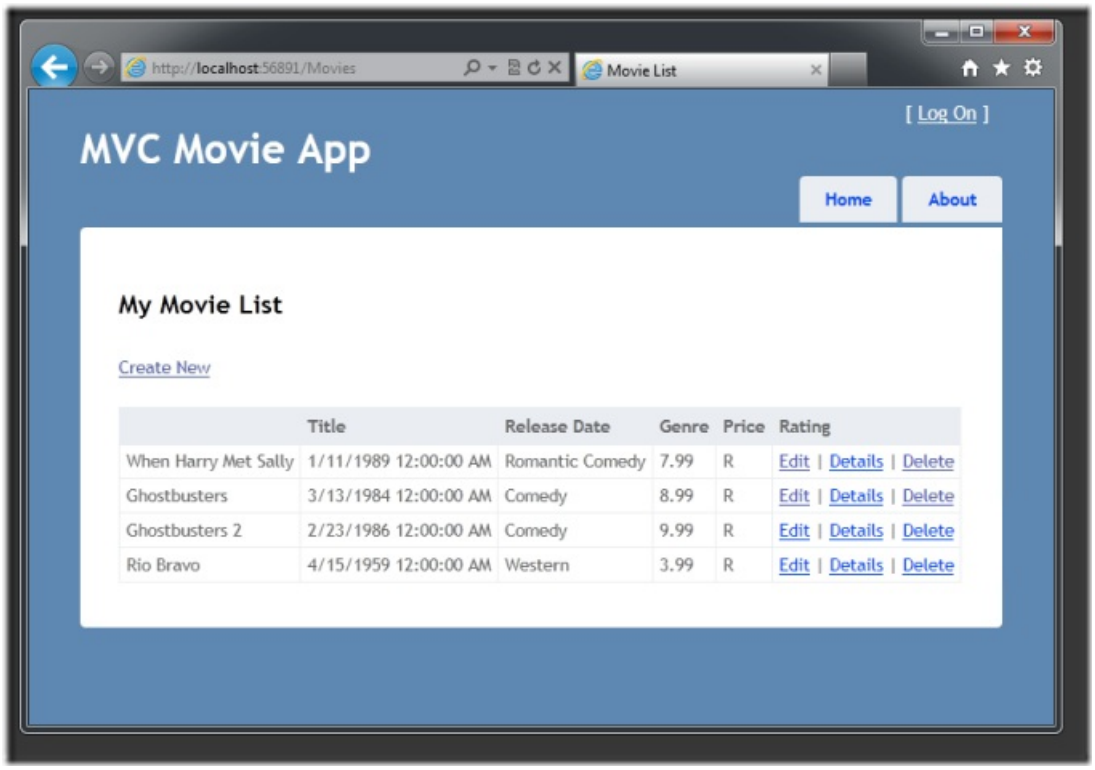
        AreaRegistration.RegisterAllAreas()

        RegisterGlobalFilters(GlobalFilters.Filters)
        RegisterRoutes(RouteTable.Routes)
    End Sub
```

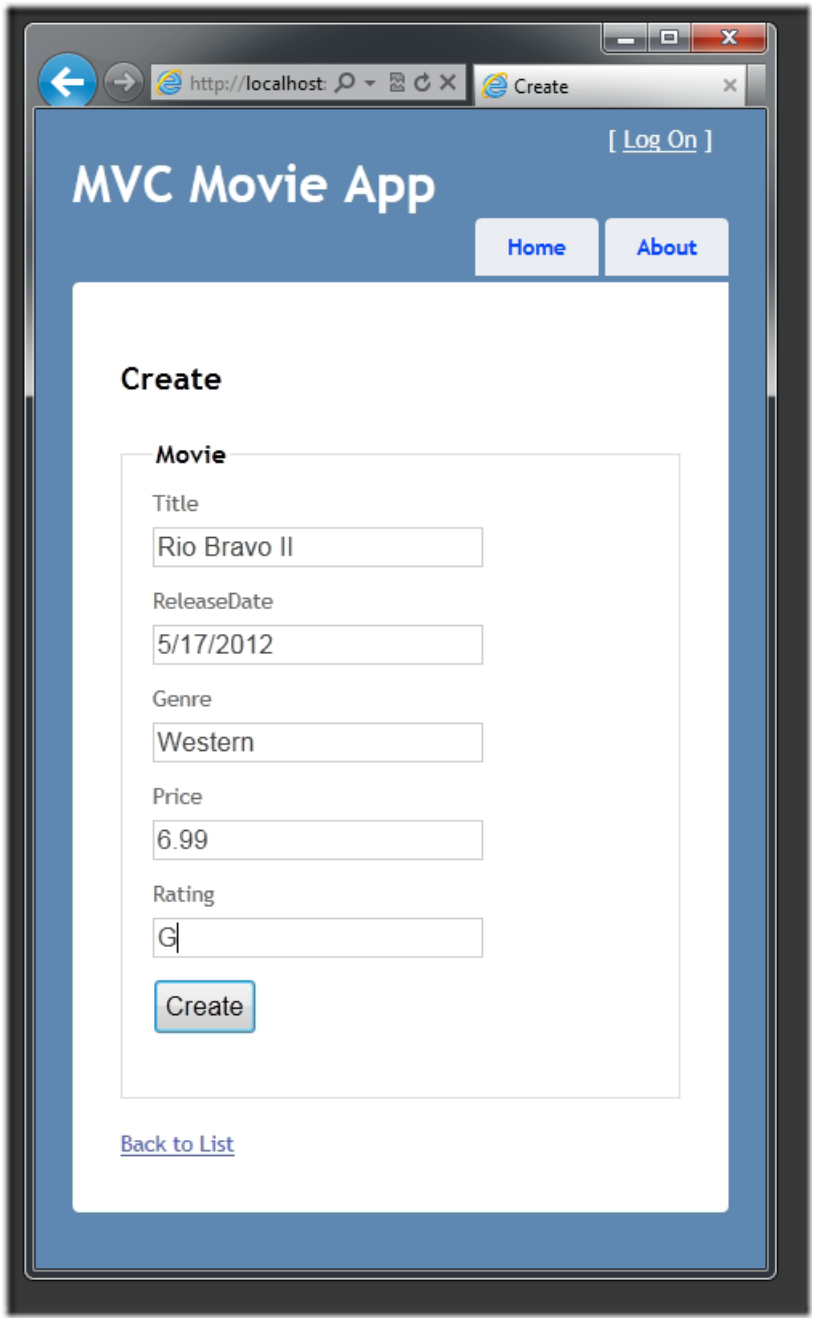
The `Database.SetInitializer` statement you just added indicates that the database used by the `MovieDBContext` instance should be automatically deleted and re-created if the schema and the database don't match. And as you saw, it will also populate the database with the sample data that's specified in the `MovieInitializer` class.

Close the *Global.asax* file.

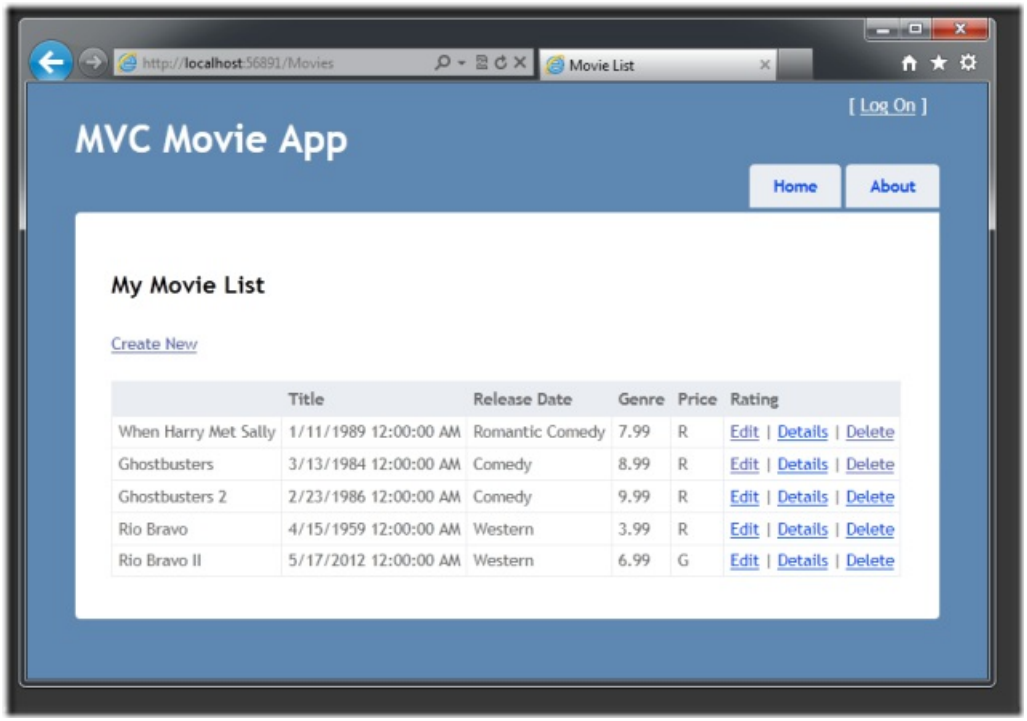
Re-run the application and navigate to the */Movies* URL. When the application starts, it detects that the model structure no longer matches the database schema. It automatically re-creates the database to match the new model structure and populates the database with the sample movies:



Click the **Create New** link to add a new movie. Note that you can add a rating.



Click **Create**. The new movie, including the rating, now shows up in the movies listing:



In this section you saw how you can modify model objects and keep the database in sync with the changes. You also learned a way to populate a newly created database with sample data so you can try out scenarios. Next, let's look at how you can add richer validation logic to the model classes and enable some business rules to be enforced.

PREVIOUS NEXT

Adding Validation to the Model (VB)

2/20/2020 • 8 minutes to read • Edit Online

by Rick Anderson

> This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:
>
> - Visual Studio Web Developer Express SP1 prerequisites
> - ASP.NET MVC 3 Tools Update
> - SQL Server Compact 4.0(runtime + tools support)
>
> If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.
>
> A Visual Web Developer project with VB.NET source code is available to accompany this topic. Download the VB.NET version. If you prefer C#, switch to the C# version of this tutorial.

In this section you'll add validation logic to the `Movie` model, and you'll ensure that the validation rules are enforced any time a user attempts to create or edit a movie using the application.

Keeping Things DRY

One of the core design tenets of ASP.NET MVC is DRY ("Don't Repeat Yourself"). ASP.NET MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an application. This reduces the amount of code you need to write and makes the code you do write much easier to maintain.

The validation support provided by ASP.NET MVC and Entity Framework Code First is a great example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and then those rules are enforced everywhere in the application.

Let's look at how you can take advantage of this validation support in the movie application.

Adding Validation Rules to the Movie Model

You'll begin by adding some validation logic to the `Movie` class.

Open the *Movie.vb* file. Add a `Imports` statement at the top of the file that references the `System.ComponentModel.DataAnnotations` namespace:

```
Imports System.ComponentModel.DataAnnotations
```

The namespace is part of the .NET Framework. It provides a built-in set of validation attributes that you can apply declaratively to any class or property.

Now update the `Movie` class to take advantage of the built-in `Required`, `StringLength`, and `Range` validation attributes. Use the following code as an example of where to apply the attributes.

```
Public Class Movie
    Public Property ID() As Integer

    <Required(ErrorMessage:="Title is required")>
    Public Property Title() As String

    <Required(ErrorMessage:="Date is required")>
    Public Property ReleaseDate() As Date

    <Required(ErrorMessage:="Genre must be specified")>
    Public Property Genre() As String

    <Required(ErrorMessage:="Price Required"), Range(1, 100, ErrorMessage:="Price must be between $1 and
$100")>
    Public Property Price() As Decimal

    <StringLength(5)>
    Public Property Rating() As String
End Class
```

The validation attributes specify behavior that you want to enforce on the model properties they are applied to. The `Required` attribute indicates that a property must have a value; in this sample, a movie has to have values for the `Title`, `ReleaseDate`, `Genre`, and `Price` properties in order to be valid. The `Range` attribute constrains a value to within a specified range. The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length.

Code First ensures that the validation rules you specify on a model class are enforced before the application saves changes in the database. For example, the code below will throw an exception when the `SaveChanges` method is called, because several required `Movie` property values are missing and the price is zero (which is out of the valid range).

```
Dim db As New MovieDBContext()

Dim movie As New Movie()
movie.Title = "Gone with the Wind"
movie.Price = 0.0D

db.Movies.Add(movie)
db.SaveChanges() ' <= Will throw validation exception
```

Having validation rules automatically enforced by the .NET Framework helps make your application more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Here's a complete code listing for the updated *Movie.vb* file:

```
Imports System.Data.Entity
Imports System.ComponentModel.DataAnnotations

Public Class Movie
    Public Property ID() As Integer

    <Required(ErrorMessage:="Title is required")>
    Public Property Title() As String

    <Required(ErrorMessage:="Date is required")>
    Public Property ReleaseDate() As Date

    <Required(ErrorMessage:="Genre must be specified")>
    Public Property Genre() As String

    <Required(ErrorMessage:="Price Required"), Range(1, 100, ErrorMessage:="Price must be between $1 and
$100")>
    Public Property Price() As Decimal

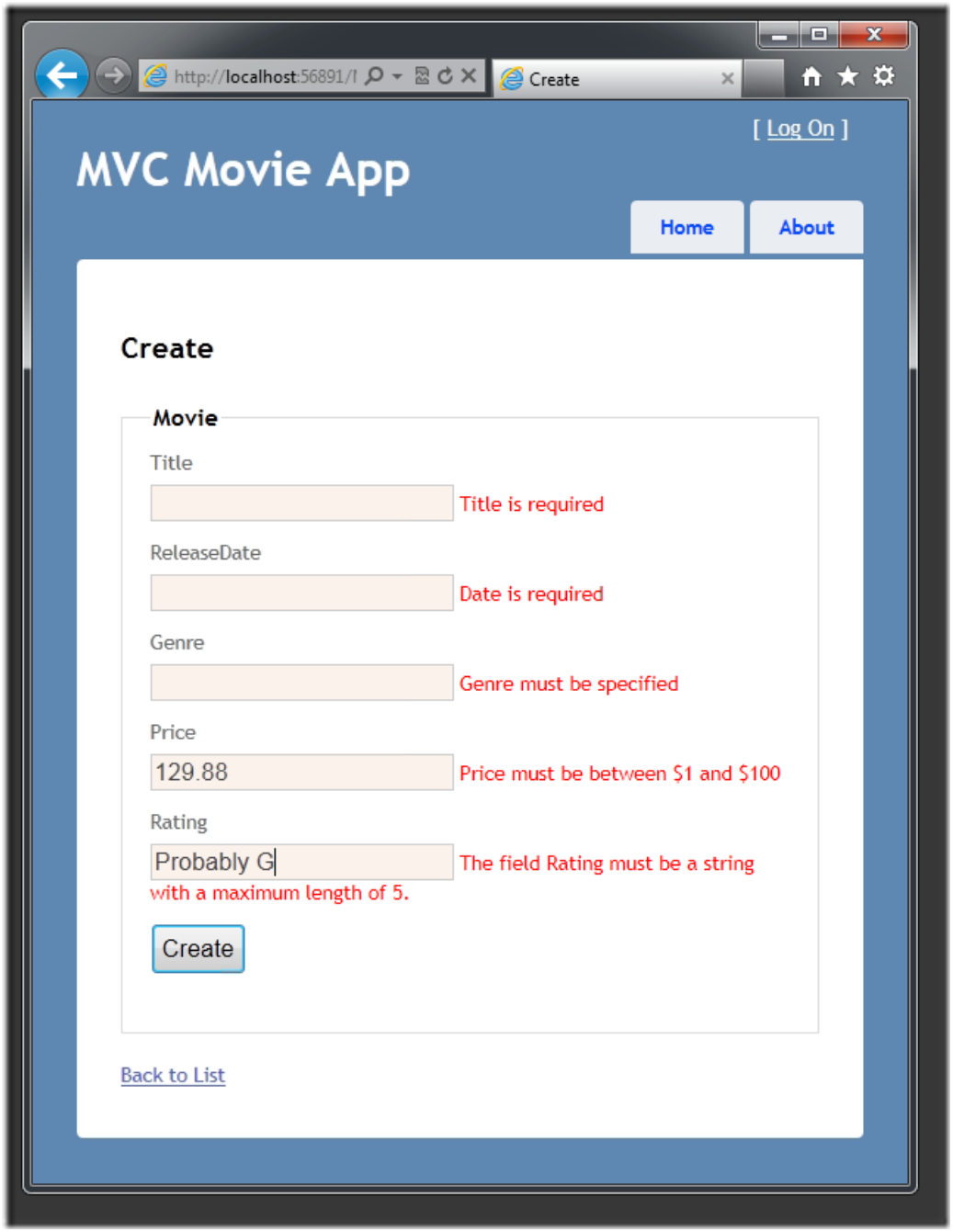
    <StringLength(5)>
    Public Property Rating() As String
End Class

Public Class MovieDBContext
    Inherits DbContext
    Public Property Movies() As DbSet(Of Movie)
End Class
```

Validation Error UI in ASP.NET MVC

Re-run the application and navigate to the */Movies* URL.

Click the **Create Movie** link to add a new movie. Fill out the form with some invalid values and then click the **Create** button.



Notice how the form has automatically used a background color to highlight the text boxes that contain invalid data and has emitted an appropriate validation error message next to each one. The error messages match the error strings you specified when you annotated the `Movie` class. The errors are enforced both client-side (using JavaScript) and server-side (in case a user has JavaScript disabled).

A real benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the *Create.vbhtml* view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified using attributes on the `Movie` model class.

How Validation Occurs in the Create View and Create Action Method

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The next listing shows what the `Create` methods in the `MovieController` class look like. They're unchanged from how you created them earlier in this tutorial.

```
'
' GET: /Movies/Create

Function Create() As ViewResult
    Return View()
End Function

'
' POST: /Movies/Create

<HttpPost()>
Function Create(movie As Movie) As ActionResult
    If ModelState.IsValid Then
        db.Movies.Add(movie)
        db.SaveChanges()
        Return RedirectToAction("Index")
    End If

    Return View(movie)
End Function
```

The first action method displays the initial Create form. The second handles the form post. The second `Create` method calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method redisplays the form. If there are no errors, the method saves the new movie in the database.

Below is the *Create.vbhtml* view template that you scaffolded earlier in the tutorial. It's used by the action methods shown above both to display the initial form and to redisplay it in the event of an error.

```
@ModelType MvcMovie.Movie

@Code
    ViewData("Title") = "Create"
End Code

<h2>Create</h2>

<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>

@Using Html.BeginForm()
    @Html.ValidationSummary(True)
    @<fieldset>
        <legend>Movie</legend>

        <div class="editor-label">
            @Html.LabelFor(Function(model) model.Title)
        </div>
        <div class="editor-field">
            @Html.EditorFor(Function(model) model.Title)
            @Html.ValidationMessageFor(Function(model) model.Title)
        </div>

        <div class="editor-label">
            @Html.LabelFor(Function(model) model.ReleaseDate)
        </div>
        <div class="editor-field">
            @Html.EditorFor(Function(model) model.ReleaseDate)
            @Html.ValidationMessageFor(Function(model) model.ReleaseDate)
        </div>

        <div class="editor-label">
            @Html.LabelFor(Function(model) model.Genre)
        </div>
        <div class="editor-field">
            @Html.EditorFor(Function(model) model.Genre)
            @Html.ValidationMessageFor(Function(model) model.Genre)
        </div>

        <div class="editor-label">
            @Html.LabelFor(Function(model) model.Price)
        </div>
        <div class="editor-field">
            @Html.EditorFor(Function(model) model.Price)
            @Html.ValidationMessageFor(Function(model) model.Price)
        </div>

 <div class="editor-label">
     @Html.LabelFor(Function(model) model.Rating)
 </div>
 <div class="editor-field">
     @Html.EditorFor(Function(model) model.Rating)
     @Html.ValidationMessageFor(Function(model) model.Rating)
 </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
End Using

<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Notice how the code uses an `Html.EditorFor` helper to output the `<input>` element for each `Movie` property. Next to this helper is a call to the `Html.ValidationMessageFor` helper method. These two helper methods work with the model object that's passed by the controller to the view (in this case, a `Movie` object). They automatically look for validation attributes specified on the model and display error messages as appropriate.

What's really nice about this approach is that neither the controller nor the Create view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the `Movie` class.

If you want to change the validation logic later, you can do so in exactly one place. You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the DRY principle.

Adding Formatting to the Movie Model

Open the *Movie.vb* file. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. You'll apply the `DisplayFormat` attribute and a `DataType` enumeration value to the release date and to the price fields. The following code shows the `ReleaseDate` and `Price` properties with the appropriate `DisplayFormat` attribute.

```
<DataType(DataType.Date)>
    Public Property ReleaseDate() As Date

     <DataType(DataType.Currency)>
    Public Property Price() As Decimal
```

Alternatively, you could explicitly set a `DataFormatString` value. The following code shows the release date property with a date format string (namely, "d"). You'd use this to specify that you don't want to time as part of the release date.

```
<DisplayFormat(DataFormatString:="{0:d}")>
    Public Property ReleaseDate() As Date
```

The following code formats the `Price` property as currency.

```
<DisplayFormat(DataFormatString:="{0:c}")>
    Public Property Price() As Decimal
```

The complete `Movie` class is shown below.

```
Public Class Movie
    Public Property ID() As Integer

    <Required(ErrorMessage:="Title is required")>
    Public Property Title() As String

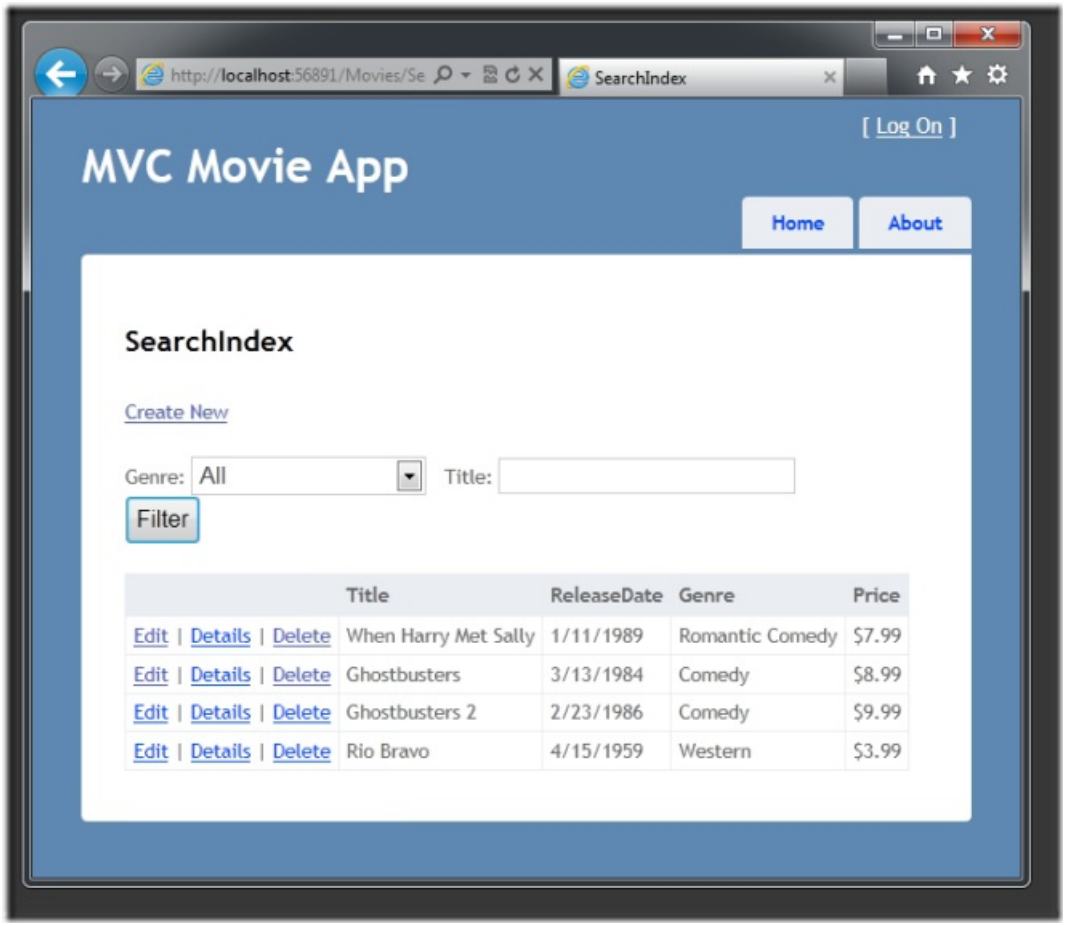
    <Required(ErrorMessage:="Date is required")>
    <DataType(DataType.Date)>
    Public Property ReleaseDate() As Date

    <Required(ErrorMessage:="Genre must be specified")>
    Public Property Genre() As String

    <Required(ErrorMessage:="Price Required"), Range(1, 100, ErrorMessage:="Price must be between $1 and
$100")>
    <DataType(DataType.Currency)>
    Public Property Price() As Decimal

    <StringLength(5)>
    Public Property Rating() As String
End Class
```

Run the application and browse to the `Movies` controller.



In the next part of the series, we'll review the application and make some improvements to the automatically generated `Details` and `Delete` methods..

PREVIOUS NEXT

Improving the Details and Delete Methods (VB)

2/20/2020 • 5 minutes to read • Edit Online

by Rick Anderson

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

A Visual Web Developer project with VB.NET source code is available to accompany this topic. Download the VB.NET version. If you prefer C#, switch to the C# version of this tutorial.

In this part of the tutorial, you'll make some improvements to the automatically generated `Details` and `Delete` methods. These changes aren't required, but with just a few small bits of code, you can easily enhance the application.

Improving the Details and Delete Methods

When you scaffolded the `Movie` controller, ASP.NET MVC generated code that worked great, but that can be made more robust with just a few small changes.

Open the `Movie` controller and modify the `Details` method by returning `HttpNotFound` when a movie isn't found. You should also modify the `Details` method to set a default value for the ID that's passed to it. (You made similar changes to the `Edit` method in part 6 of this tutorial.) However, you must change the return type of the `Details` method from `ViewResult` to `ActionResult`, because the `HttpNotFound` method doesn't return a `ViewResult` object. The following example shows the modified `Details` method.

```
Public Function Details(Optional ByVal id As Integer = 0) As ActionResult
    Dim movie As Movie = db.Movies.Find(id)
    If movie Is Nothing Then
        Return HttpNotFound()
    End If
    Return View(movie)
End Function
```

Code First makes it easy to search for data using the `Find` method. An important security feature that we built into the method is that the code verifies that the `Find` method has found a movie before the code tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from `http://localhost:xxxx/Movies/Details/1` to something like `http://localhost:xxxx/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you don't the check for a null movie, this could result in a database error.

Similarly, change the `Delete` and `DeleteConfirmed` methods to specify a default value for the ID parameter and to return `HttpNotFound` when a movie isn't found. The updated `Delete` methods in the `Movie` controller are shown below.

```
' GET: /Movies/Delete/5

 Public Function Delete(Optional ByVal id As Integer = 0) As ActionResult
     Dim movie As Movie = db.Movies.Find(id)
     If movie Is Nothing Then
         Return HttpNotFound()
     End If
     Return View(movie)
 End Function

 '
 ' POST: /Movies/Delete/5

 <HttpPost(), ActionName("Delete")>
 Public Function DeleteConfirmed(Optional ByVal id As Integer = 0) As ActionResult
     Dim movie As Movie = db.Movies.Find(id)
     If movie Is Nothing Then
         Return HttpNotFound()
     End If
     db.Movies.Remove(movie)
     db.SaveChanges()
     Return RedirectToAction("Index")
 End Function
```

Note that the `Delete` method doesn't delete the data. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole. For more information about this, see Stephen Walther's blog entry ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes.

The `HttpPost` method that deletes the data is named `DeleteConfirmed` to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

```
Public Function Delete(Optional ByVal id As Integer = 0) As ActionResult

 <HttpPost(), ActionName("Delete")>
 Public Function DeleteConfirmed(Optional ByVal id As Integer = 0) As ActionResult
```

The common language runtime (CLR) requires overloaded methods to have a unique signature (same name, different list of parameters). However, here you need two Delete methods -- one for GET and one for POST -- that both require the same signature. (They both need to accept a single integer as a parameter.)

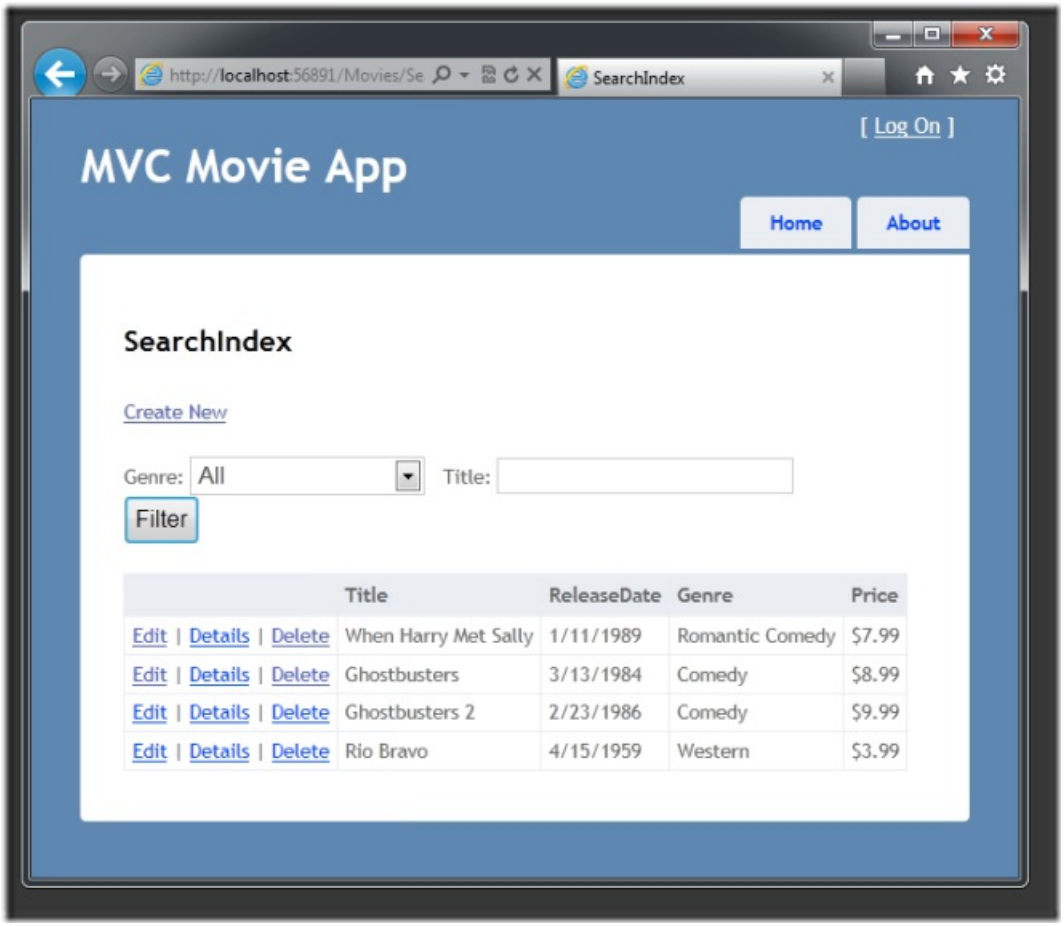
To sort this out, you can do a couple of things. One is to give the methods different names. That's what we did in he preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. This effectively performs mapping for the routing system so that a URL that includes */Delete/*for a POST request will find the `DeleteConfirmed` method.

Another way to avoid a problem with methods that have identical names and signatures is to artificially change the signature of the POST method to include an unused parameter. For example, some developers add a parameter type `FormCollection` that is passed to the POST method, and then simply don't use the parameter:

```
Public Function Delete(ByVal fcNotUsed As FormCollection, Optional ByVal id As Integer = 0) As ActionResult
Dim movie As Movie = db.Movies.Find(id)
If movie Is Nothing Then
    Return HttpNotFound()
End If
db.Movies.Remove(movie)
db.SaveChanges()
Return RedirectToAction("Index")
End Function
```

Wrapping Up

You now have a complete ASP.NET MVC application that stores data in a SQL Server Compact database. You can create, read, update, delete, and search for movies.



This basic tutorial got you started making controllers, associating them with views, and passing around hard-coded data. Then you created and designed a data model. Entity Framework Code First created a database from the data model on the fly, and the ASP.NET MVC scaffolding system automatically generated the action methods and views for basic CRUD operations. You then added a search form that let users search the database. You changed the database to include a new column of data, and then updated two pages to create and display this new data. You added validation by marking the data model with attributes from the `DataAnnotations` namespace. The resulting validation runs on the client and on the server.

If you'd like to deploy your application, it's helpful to first test the application on your local IIS 7 server. You can use this Web Platform Installer link to enable IIS setting for ASP.NET applications. See the following deployment links:

- ASP.NET Deployment Content Map
- Enabling IIS 7.x
- Web Application Projects Deployment

I now encourage you to move on to our intermediate-level Creating an Entity Framework Data Model for an ASP.NET MVC Application and MVC Music Store tutorials, to explore the ASP.NET articles on MSDN, and to check out the many videos and resources at https://asp.net/mvc to learn even more about ASP.NET MVC! The ASP.NET MVC forums are a great place to ask questions.

Enjoy!

— Scott Hanselman (http://hanselman.com and @shanselman on Twitter) and Rick Anderson blogs.msdn.com/rickAndy

PREVIOUS

Creating a MVC 3 Application with Razor and Unobtrusive JavaScript

4/10/2020 • 10 minutes to read • Edit Online

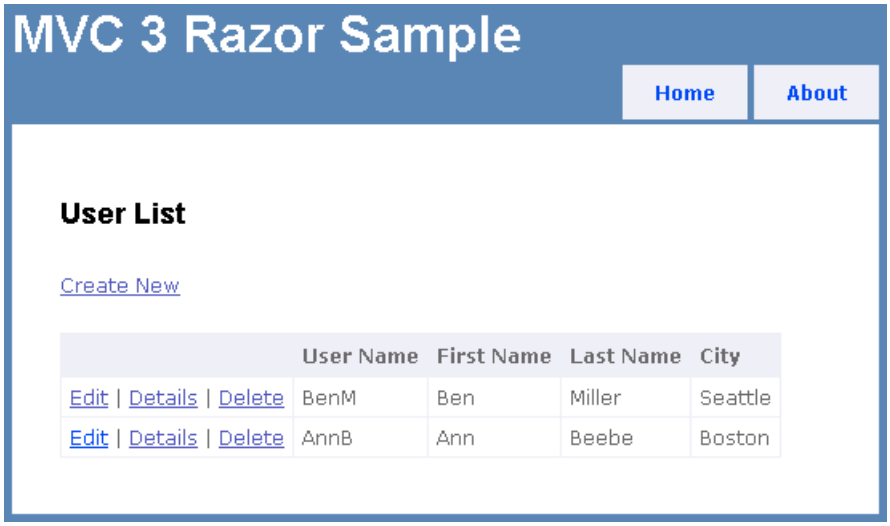
by Microsoft

The User List sample web application demonstrates how simple it is to create ASP.NET MVC 3 applications using the Razor view engine. The sample application shows how to use the new Razor view engine with ASP.NET MVC version 3 and Visual Studio 2010 to create a fictional User List website that includes functionality such as creating, displaying, editing, and deleting users.

This tutorial describes the steps that were taken in order to build the User List sample ASP.NET MVC 3 application. A Visual Studio project with C# and VB source code is available to accompany this topic: Download. If you have questions about this tutorial, please post them to the MVC forum.

Overview

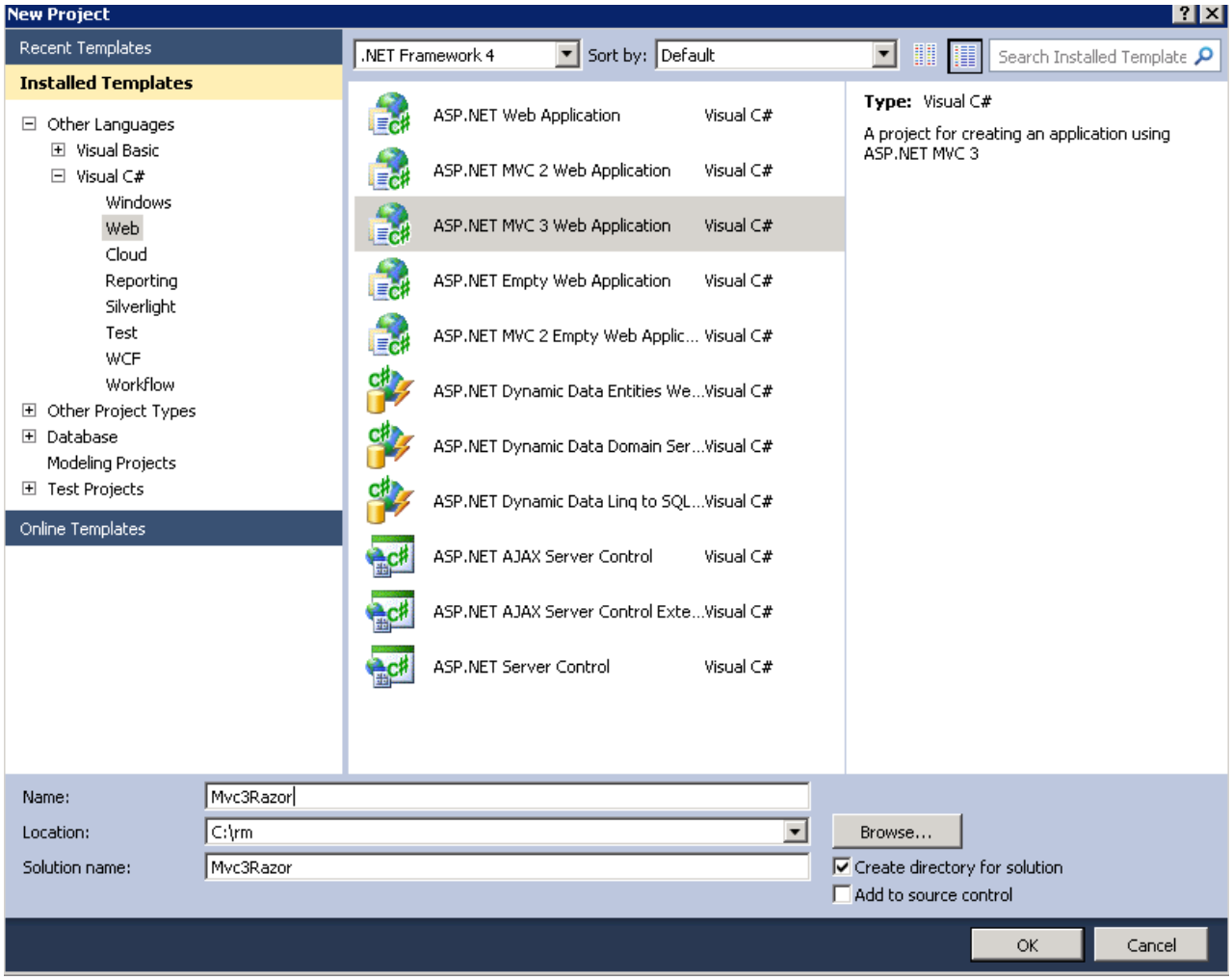
The application you'll be building is a simple user list website. Users can enter, view, and update user information.



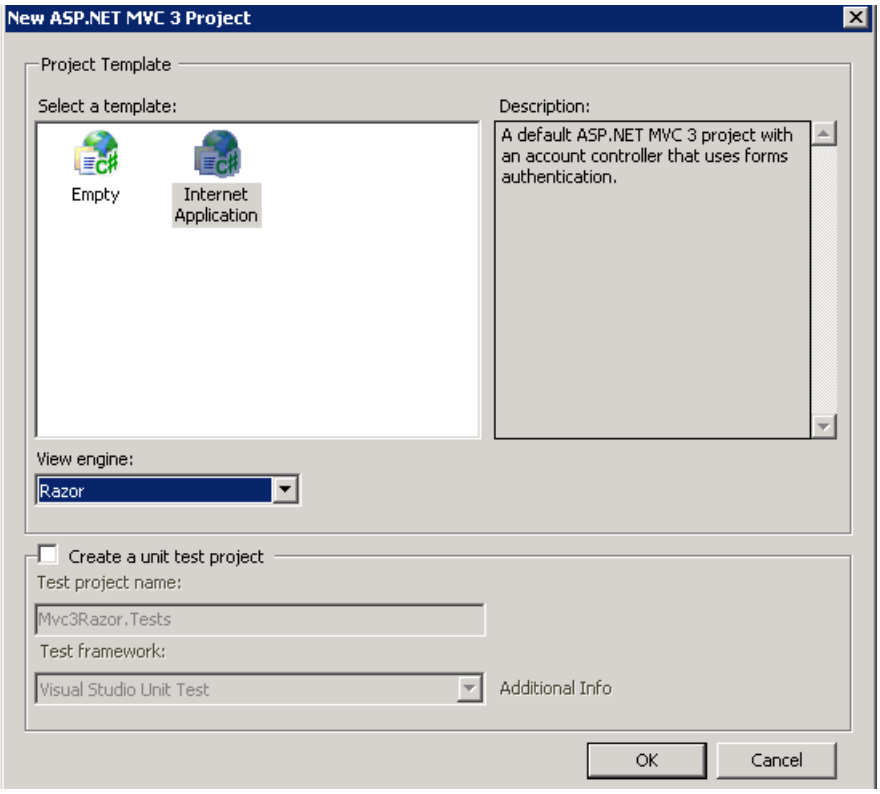
You can download the VB and C# completed project here.

Creating the Web Application

To start the tutorial, open Visual Studio 2010 and create a new project using the *ASP.NET MVC 3 Web Application* template. Name the application "Mvc3Razor".

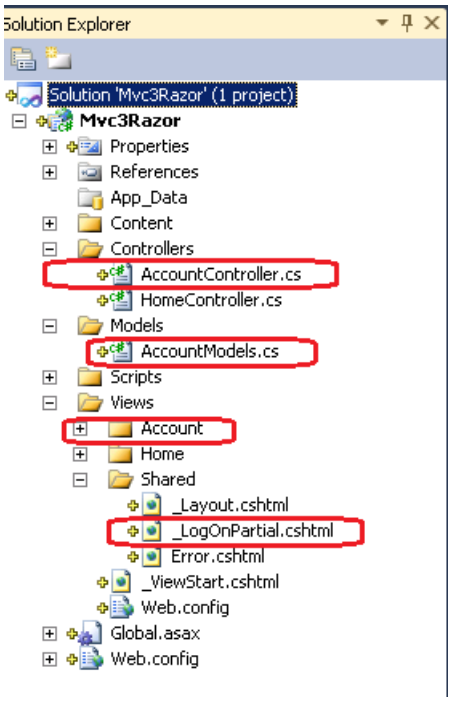


In the **New ASP.NET MVC 3 Project** dialog, select **Internet Application**, select the Razor view engine, and then click **OK**.



In this tutorial you will not be using the ASP.NET membership provider, so you can delete all the files associated with logon and membership. In **Solution Explorer**, remove the following files and directories:

- *Controllers\AccountController*
- *Models\AccountModels*
- *Views\Shared_LogOnPartial*
- *Views\Account* (and all the files in this directory)

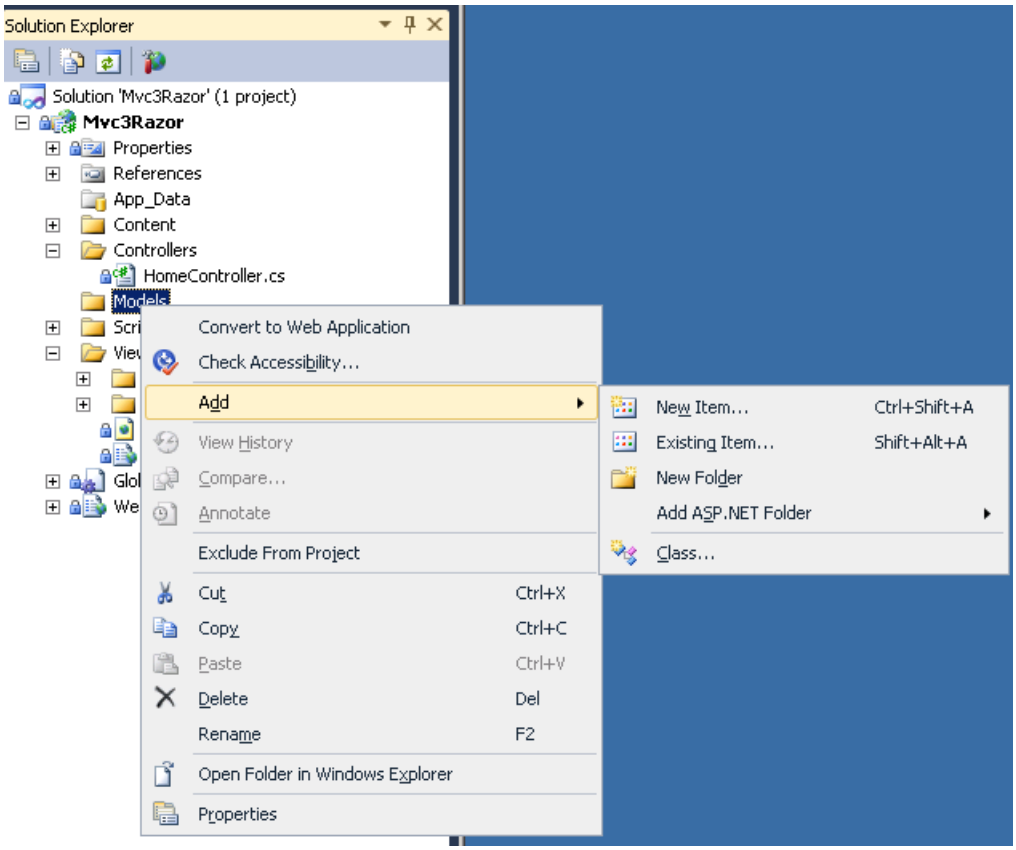


Edit the *_Layout.cshtml* file and replace the markup inside the `<div>` element named `logindisplay` with the message "Login Disabled". The following example shows the new markup:

```
<div id="logindisplay">
  Login Disabled
</div>
```

Adding the Model

In **Solution Explorer**, right-click the *Models* folder, select **Add**, and then click **Class**.



Name the class `UserModel`. Replace the contents of the *UserModel* file with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;

namespace Mvc3Razor.Models {
    public class UserModel {

        [Required]
        [StringLength(6, MinimumLength = 3)]
        [Display(Name = "User Name")]
        [RegularExpression(@"(\S)+", ErrorMessage = "White space is not allowed")]
        [ScaffoldColumn(false)]
        public string UserName { get; set; }

        [Required]
        [StringLength(8, MinimumLength = 3)]
        [Display(Name = "First Name")]
        public string FirstName { get; set; }
        [Required]
        [StringLength(9, MinimumLength = 2)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required()]
        public string City { get; set; }

    }

    public class Users {

        public Users() {
            _usrList.Add(new UserModel
            {
                UserName = "BenM",
                FirstName = "Ben",
                LastName = "Miller",
                City = "Seattle"
            });
            _usrList.Add(new UserModel
```

```
            {
                UserName = "AnnB",
                FirstName = "Ann",
                LastName = "Beebe",
                City = "Boston"
            });
        }

        public List<UserModel> _usrList = new List<UserModel>();

        public void Update(UserModel umToUpdate) {

            foreach (UserModel um in _usrList) {
                if (um.UserName == umToUpdate.UserName) {
                    _usrList.Remove(um);
                    _usrList.Add(umToUpdate);
                    break;
                }
            }
        }

        public void Create(UserModel umToUpdate) {
            foreach (UserModel um in _usrList) {
                if (um.UserName == umToUpdate.UserName) {
                    throw new System.InvalidOperationException("Duplicat username: " + um.UserName);
                }
            }
            _usrList.Add(umToUpdate);
        }

        public void Remove(string usrName) {

            foreach (UserModel um in _usrList) {
                if (um.UserName == usrName) {
                    _usrList.Remove(um);
                    break;
                }
            }
        }

        public  UserModel GetUser(string uid) {
            UserModel usrMdl = null;

            foreach (UserModel um in _usrList)
                if (um.UserName == uid)
                    usrMdl = um;

            return usrMdl;
        }

    }
}
```

The `UserModel` class represents users. Each member of the class is annotated with the Required attribute from the DataAnnotations namespace. The attributes in the DataAnnotations namespace provide automatic client- and server-side validation for web applications.

Open the `HomeController` class and add a `using` directive so that you can access the `UserModel` and `Users` classes:

```
using Mvc3Razor.Models;
```

Just after the `HomeController` declaration, add the following comment and the reference to a `Users` class:

```
public class HomeController : Controller {

// The __usrs class is replacement for a real data access strategy.
private static Users _usrs = new Users();
```

The `Users` class is a simplified, in-memory data store that you'll use in this tutorial. In a real application you would use a database to store user information. The first few lines of the `HomeController` file are shown in the following example:

```
using System.Web.Mvc;
using Mvc3Razor.Models;

namespace Mvc3Razor.Controllers {

    public class HomeController : Controller {

        // The __usrs class is replacement for a real data access strategy.
        private static Users _usrs = new Users();
```

Build the application so that the user model will be available to the scaffolding wizard in the next step.

Creating the Default View

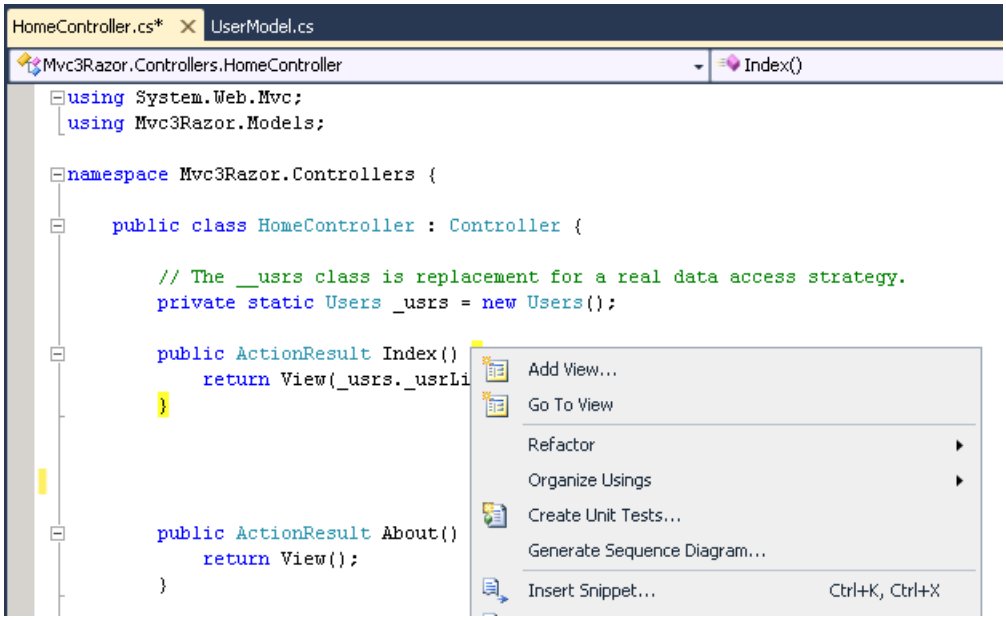
The next step is to add an action method and view to display the users.

Delete the existing *Views\Home\Index* file. You will create a new *Index* file to display the users.

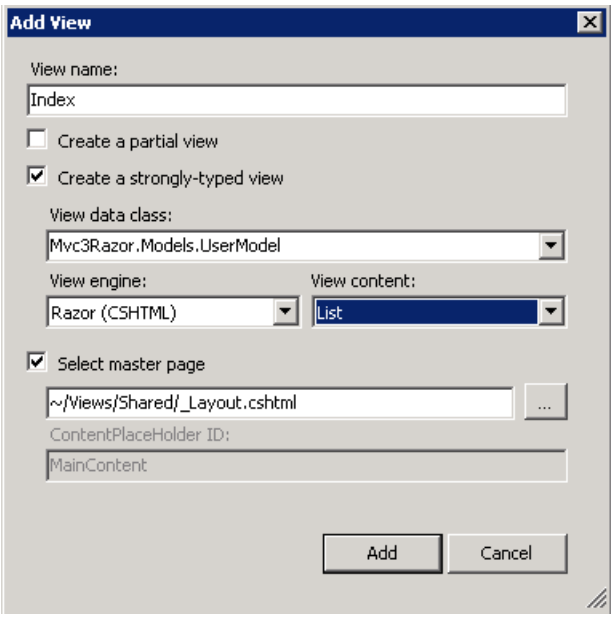
In the `HomeController` class, replace the contents of the `Index` method with the following code:

```
return View(_usrs._usrList);
```

Right-click inside the `Index` method and then click **Add View**.



Select the **Create a strongly-typed view** option. For **View data class**, select **Mvc3Razor.Models.UserModel**. (If you don't see **Mvc3Razor.Models.UserModel** in the **View data class** box, you need to build the project.) Make sure that the view engine is set to **Razor**. Set **View content** to **List** and then click **Add**.



The new view automatically scaffolds the user data that's passed to the `Index` view. Examine the newly generated *Views\Home\Index* file. The **Create New**, **Edit**, **Details**, and **Delete** links don't work, but the rest of the page is functional. Run the page. You see a list of users.

User List

Create New

	User Name	First Name	Last Name	City
Edit | Details | Delete	BenM	Ben	Miller	Seattle
Edit | Details | Delete	AnnB	Ann	Beebe	Boston

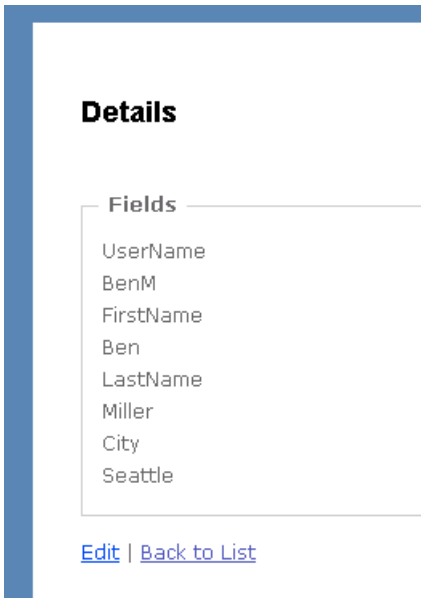
Open the *Index.cshtml* file and replace the `ActionLink` markup for **Edit**, **Details**, and **Delete** with the following code:

```
@Html.ActionLink("Edit", "Edit", new {  id=item.UserName  }) |
@Html.ActionLink("Details", "Details", new {  id=item.UserName  }) |
@Html.ActionLink("Delete", "Delete", new {  id=item.UserName  })
```

The user name is used as the ID to find the selected record in the **Edit**, **Details**, and **Delete** links.

Creating the Details View

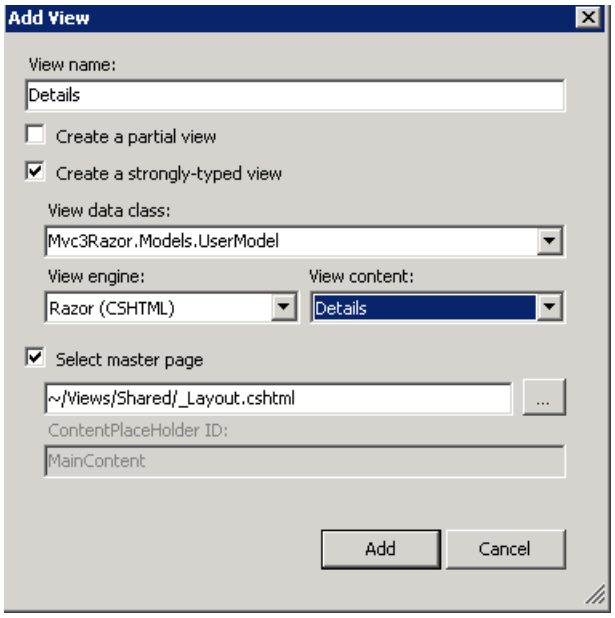
The next step is to add a `Details` action method and view in order to display user details.



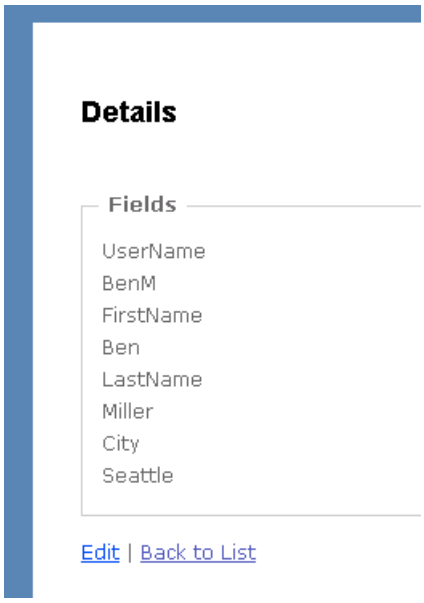
Add the following `Details` method to the home controller:

```
public ViewResult Details(string id) {
    return View(_usrs.GetUser(id));
}
```

Right-click inside the `Details` method and then select **Add View**. Verify that the **View data class** box contains **Mvc3Razor.Models.UserModel**. Set **View content** to **Details** and then click **Add**.



Run the application and select a details link. The automatic scaffolding shows each property in the model.



Creating the Edit View

Add the following `Edit` method to the home controller.

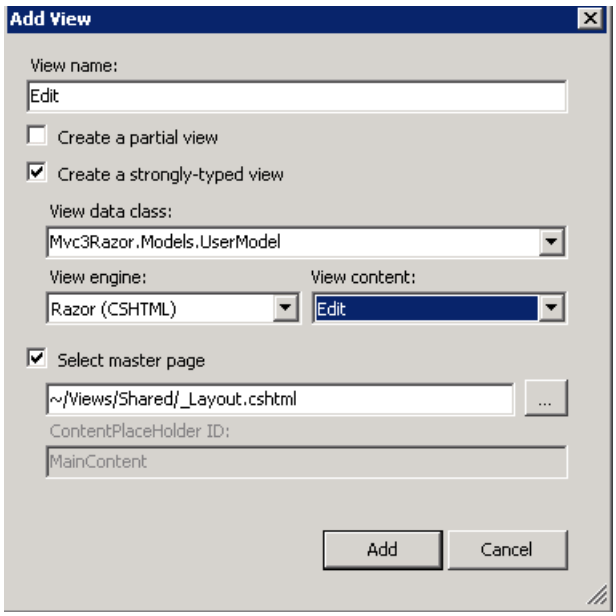
```
public ViewResult Edit(string id) {
    return View(_usrs.GetUser(id));
}

[HttpPost]
public ViewResult Edit(UserModel um) {

    if (!TryUpdateModel(um)) {
        ViewBag.updateError = "Update Failure";
        return View(um);
    }

    // ToDo: add persistent to DB.
    _usrs.Update(um);
    return View("Details", um);
}
```

Add a view as in the previous steps, but set **View content** to **Edit**.



Run the application and edit the first and last name of one of the users. If you violate any `DataAnnotation`

constraints that have been applied to the `UserModel` class, when you submit the form, you will see validation errors that are produced by server code. For example, if you change the first name "Ann" to "A", when you submit the form, the following error is displayed on the form:

```
The field First Name must be a string with a minimum length of 3 and a maximum length of 8.
```

In this tutorial, you're treating the user name as the primary key. Therefore, the user name property cannot be changed. In the *Edit.cshtml* file, just after the `Html.BeginForm` statement, set the user name to be a hidden field. This causes the property to be passed in the model. The following code fragment shows the placement of the `Hidden` statement:

```
<h2>Edit</h2>
    @using (Html.BeginForm()) {
@Html.Hidden("UserName", Model.UserName)
```

Replace the `TextBoxFor` and `ValidationMessageFor` markup for the user name with a `DisplayFor` call. The `DisplayFor` method displays the property as a read-only element. The following example shows the completed markup. The original `TextBoxFor` and `ValidationMessageFor` calls are commented out with the Razor begin-comment and end-comment characters (`@* *@`)

```
<div class="editor-label">
  @Html.LabelFor(model => model.UserName)
</div>

<div class="editor-field">
@*
  @Html.TextBoxFor(model => model.UserName)
  @Html.ValidationMessageFor(model => model.UserName)
*@
@Html.DisplayFor(model => model.UserName)
</div>
```

Enabling Client-Side Validation

To enable client-side validation in ASP.NET MVC 3, you must set two flags and you must include three JavaScript files.

Open the application's *Web.config* file. Verify `that ClientValidationEnabled` and `UnobtrusiveJavaScriptEnabled` are set to true in the application settings. The following fragment from the root *Web.config* file shows the correct settings:

```
<appSettings>
  <add key="ClientValidationEnabled" value="true"/>
  <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
 </appSettings>
```

Setting `UnobtrusiveJavaScriptEnabled` to true enables unobtrusive Ajax and unobtrusive client validation. When you use unobtrusive validation, the validation rules are turned into HTML5 attributes. HTML5 attribute names can consist of only lowercase letters, numbers, and dashes.

Setting `ClientValidationEnabled` to true enables client-side validation. By setting these keys in the application *Web.config* file, you enable client validation and unobtrusive JavaScript for the entire application. You can also enable or disable these settings in individual views or in controller methods using the following code:

```
HtmlHelper.ClientValidationEnabled = true;
HtmlHelper.UnobtrusiveJavaScriptEnabled = true;
```

You also need to include several JavaScript files in the rendered view. An easy way to include the JavaScript in all views is to add them to the *Views\Shared_Layout.cshtml* file. Replace the `<head>` element of the *_Layout.cshtml* file with the following code:

```
<head>
  <title>@View.Title</title>
  <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
  <script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.2.min.js"></script>
  <script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.7/jquery.validate.min.js"></script>
  <script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>
</head>
```

The first two jQuery scripts are hosted by the Microsoft Ajax Content Delivery Network (CDN). By taking advantage of the Microsoft Ajax CDN, you can significantly improve the first-hit performance of your applications.

Run the application and click an edit link. View the page's source in the browser. The browser source shows many attributes of the form `data-val` (for data validation). When client validation and unobtrusive JavaScript is enabled, input fields with a client-validation rule contain the `data-val="true"` attribute to trigger unobtrusive client validation. For example, the `City` field in the model was decorated with the Required attribute, which results in the HTML shown in the following example:

```
<div class="editor-field">
  <input data-val="true" data-val-required="The City field is required." id="City" name="City" type="text"
value="Seattle" />
  <span class="field-validation-valid" data-valmsg-for="City" data-valmsg-replace="true"></span>
</div>
```

For each client-validation rule, an attribute is added that has the form `data-val-rulename="message"`. Using the `City` field example shown earlier, the required client-validation rule generates the `data-val-required` attribute and the message "The City field is required". Run the application, edit one of the users, and clear the `City` field. When you tab out of the field, you see a client-side validation error message.

Similarly, for each parameter in the client-validation rule, an attribute is added that has the form `data-val-rulename-paramname=paramvalue`. For example, the `FirstName` property is annotated with the StringLength attribute and specifies a minimum length of 3 and a maximum length of 8. The data validation rule named `length` has the parameter name `max` and the parameter value 8. The following shows the HTML that is generated for the `FirstName` field when you edit one of the users:

```
<input data-val="true"
       data-val-length="The field First Name must be a string with a minimum length of 3 and a maximum length
of 8."
       data-val-length-max="8"
       data-val-length-min="3"
       data-val-required="The First Name field is required."
       id="FirstName"
       name="FirstName"
       type="text"
       value="Ben" />
```

For more information about unobtrusive client validation, see the entry Unobtrusive Client Validation in ASP.NET MVC 3 in Brad Wilson's blog.

> **NOTE**
> In ASP.NET MVC 3 Beta, you sometimes need to submit the form in order to start client-side validation. This might be changed for the final release.

Creating the Create View

The next step is to add a `Create` action method and view in order to enable the user to create a new user. Add the following `Create` method to the home controller:

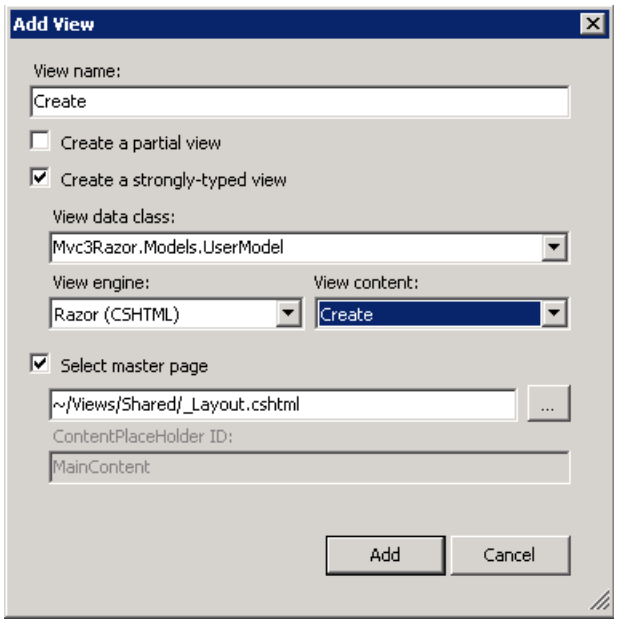
```
public ViewResult Create() {
    return View(new UserModel());
}

[HttpPost]
public ViewResult Create(UserModel um) {

    if (!TryUpdateModel(um)) {
        ViewBag.updateError = "Create Failure";
        return View(um);
    }

    // ToDo: add persistent to DB.
    _usrs.Create(um);
    return View("Details", um);
}
```

Add a view as in the previous steps, but set **View content** to **Create**.



Run the application, select the **Create** link, and add a new user. The `Create` method automatically takes advantage of client-side and server-side validation. Try to enter a user name that contains white space, such as "Ben X". When you tab out of the user name field, a client-side validation error (`White space is not allowed`) is displayed.

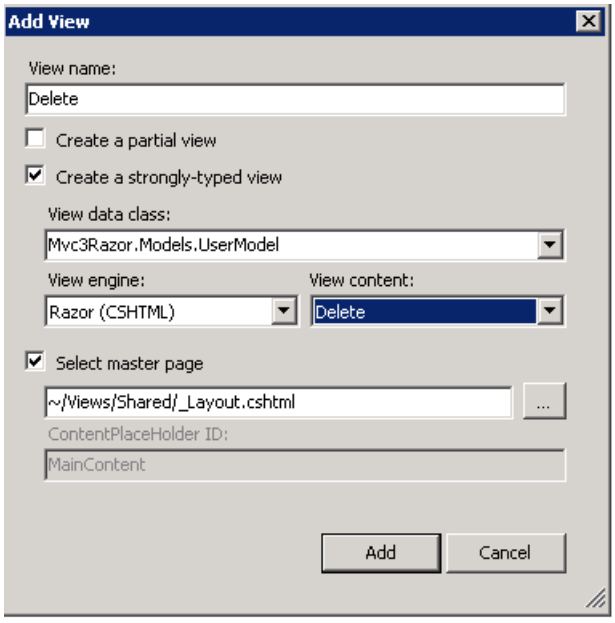
Add the Delete method

To complete the tutorial, add the following `Delete` method to the home controller:

```
public ViewResult Delete(string id) {
    return View(_usrs.GetUser(id));
}

[HttpPost]
public RedirectToRouteResult Delete(string id, FormCollection collection) {
    _usrs.Remove(id);
    return RedirectToAction("Index");
}
```

Add a `Delete` view as in the previous steps, setting **View content** to **Delete**.



You now have a simple but fully functional ASP.NET MVC 3 application with validation.

ASP.NET MVC 3 Music Store

2/20/2020 • 2 minutes to read • Edit Online

How to create the MVC Music Store application, a sample application that sells music albums online, and that implements site administration, user sign-in, and shopping cart functionality.

- Part 1: Overview and File->New Project
- Part 2: Controllers
- Part 3: Views and ViewModels
- Part 4: Models and Data Access
- Part 5: Edit Forms and Templating
- Part 6: Using Data Annotations for Model Validation
- Part 7: Membership and Authorization
- Part 8: Shopping Cart with Ajax Updates
- Part 9: Registration and Checkout
- Part 10: Final Updates to Navigation and Site Design, Conclusion

Part 1: Overview and File->New Project

2/20/2020 • 3 minutes to read • Edit Online

by Jon Galloway

> The MVC Music Store is a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Studio for web development.
>
> The MVC Music Store is a lightweight sample store implementation which sells music albums online, and implements basic site administration, user sign-in, and shopping cart functionality.
>
> This tutorial series details all of the steps taken to build the ASP.NET MVC Music Store sample application. Part 1 covers Overview and File->New Project.

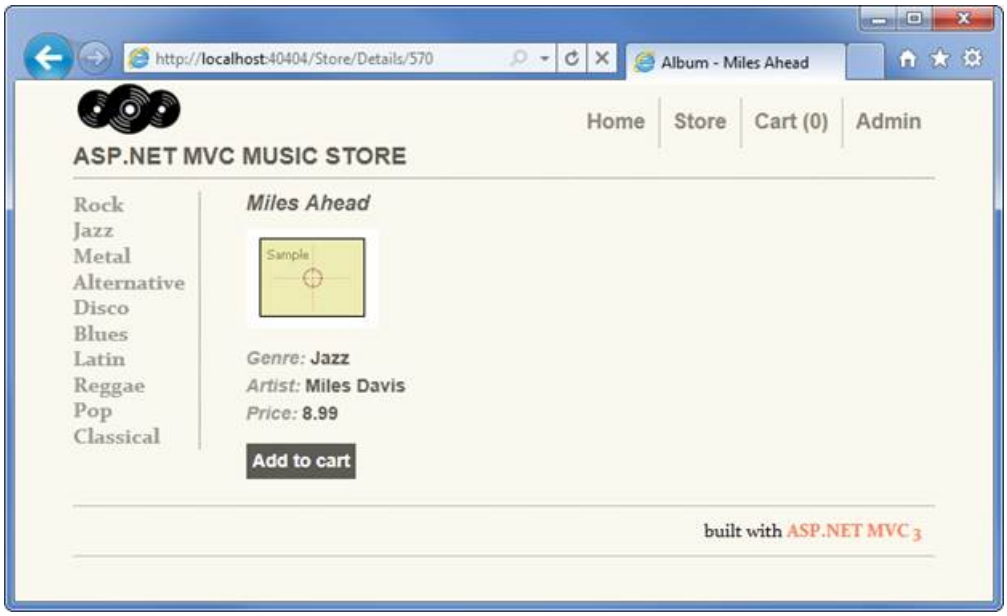
Overview

The MVC Music Store is a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Web Developer for web development. We'll be starting slowly, so beginner level web development experience is okay.

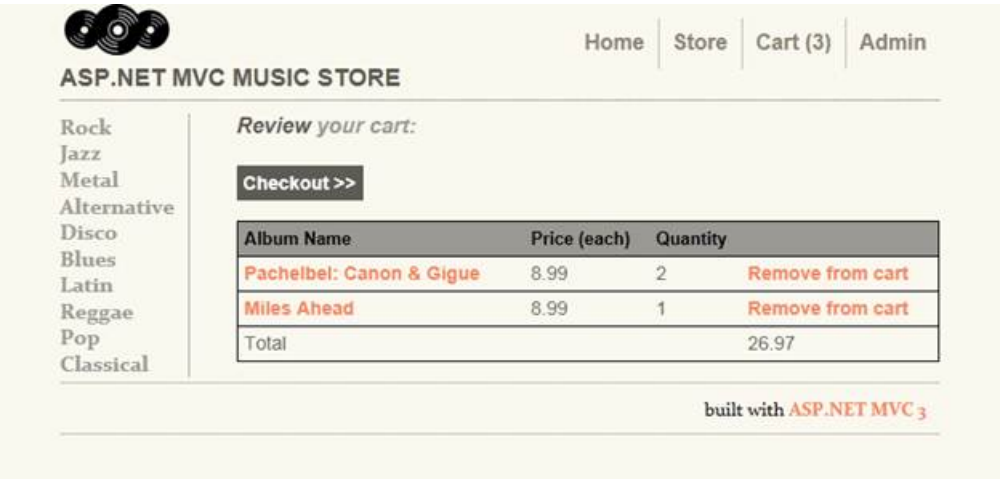
The application we'll be building is a simple music store. There are three main parts to the application: shopping, checkout, and administration.

Visitors can browse Albums by Genre:

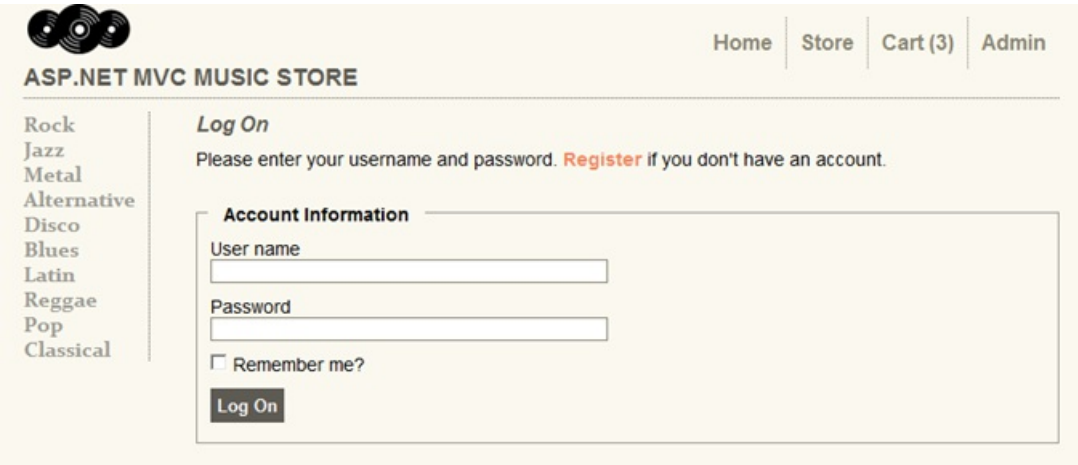
They can view a single album and add it to their cart:

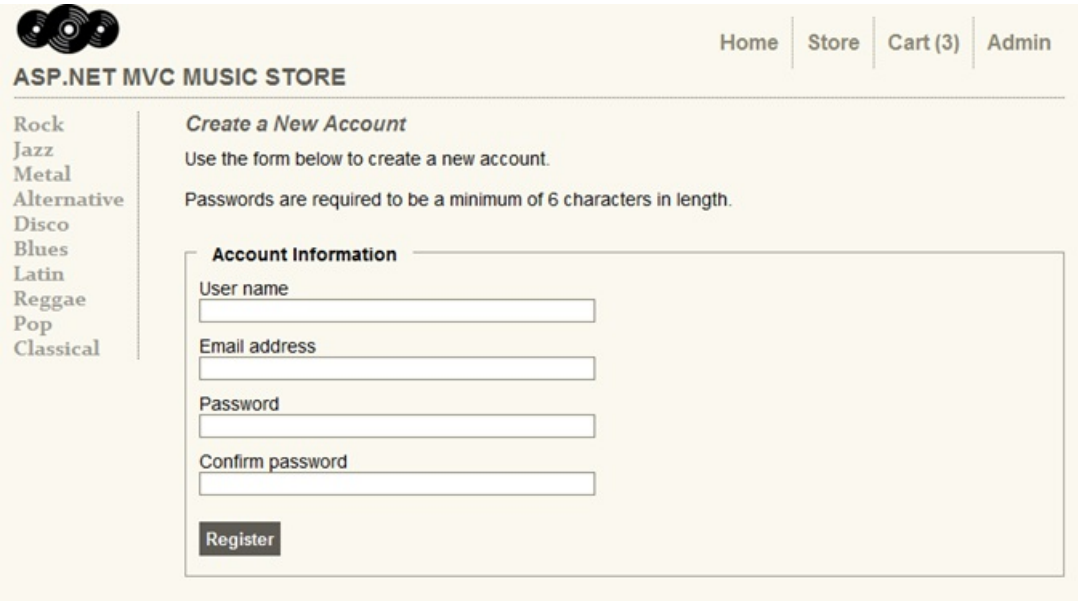


They can review their cart, removing any items they no longer want:

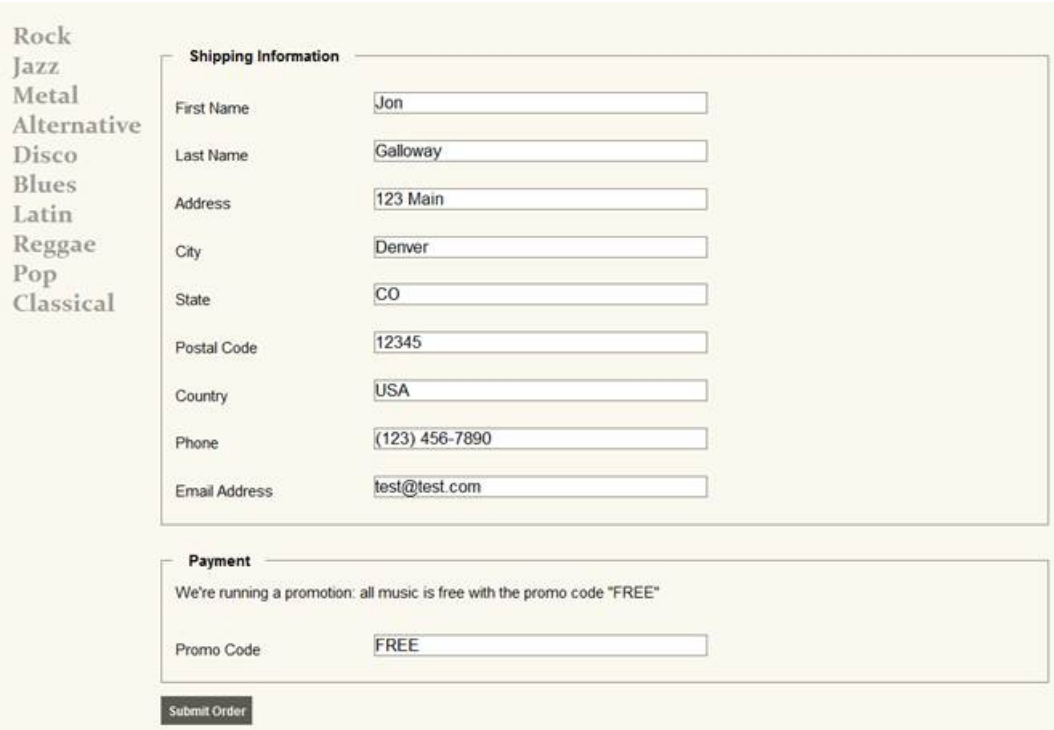


Proceeding to Checkout will prompt them to login or register for a user account.

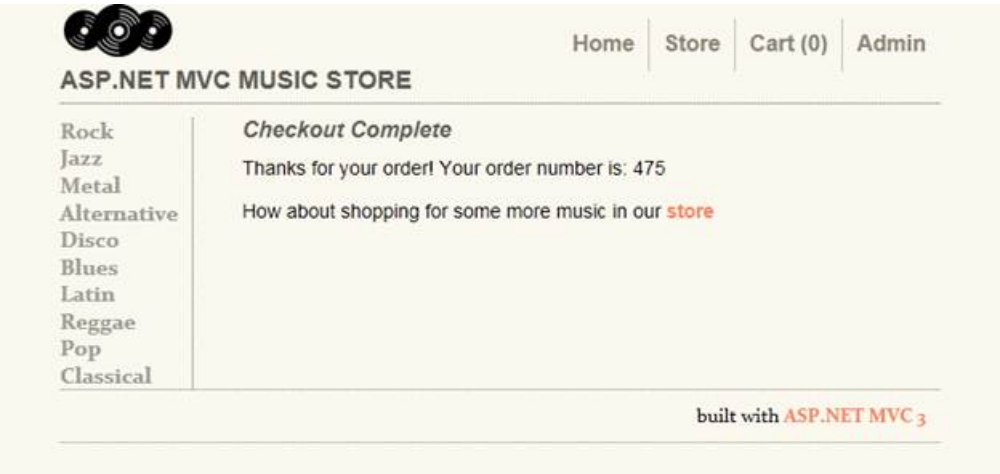




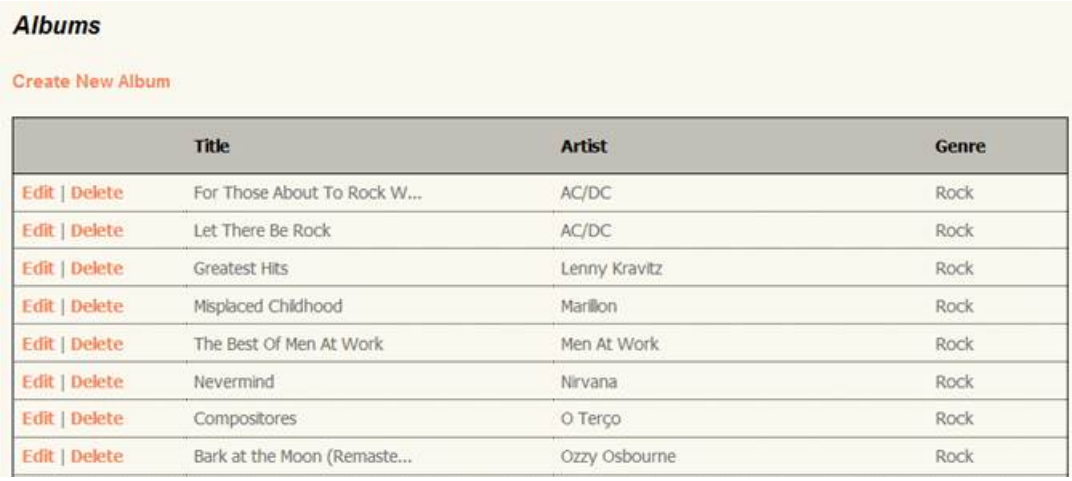
After creating an account, they can complete the order by filling out shipping and payment information. To keep things simple, we're running an amazing promotion: everything's free if they enter promotion code "FREE"!



After ordering, they see a simple confirmation screen:



In addition to customer-facing pages, we'll also build an administrator section that shows a list of albums from which Administrators can Create, Edit, and Delete albums:



1. File -> New Project

Installing the software

This tutorial will begin by creating a new ASP.NET MVC 3 project using the free Visual Web Developer 2010 Express (which is free), and then we'll incrementally add features to create a complete functioning application. Along the way, we'll cover database access, form posting scenarios, data validation, using master pages for consistent page layout, using AJAX for page updates and validation, user login, and more.

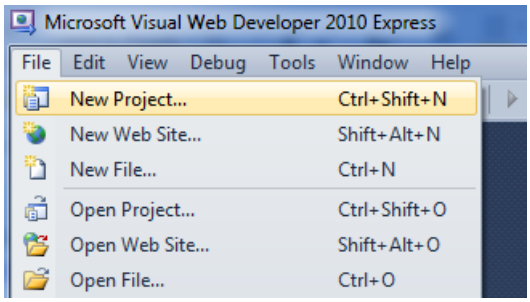
You can follow along step by step, or you can download the completed application from MVC-Music-Store.

You can use either Visual Studio 2010 SP1 or Visual Web Developer 2010 Express SP1 (a free version of Visual Studio 2010) to build the application. We'll be using the SQL Server Compact (also free) to host the database. Before you start, make sure you've installed the prerequisites listed below.

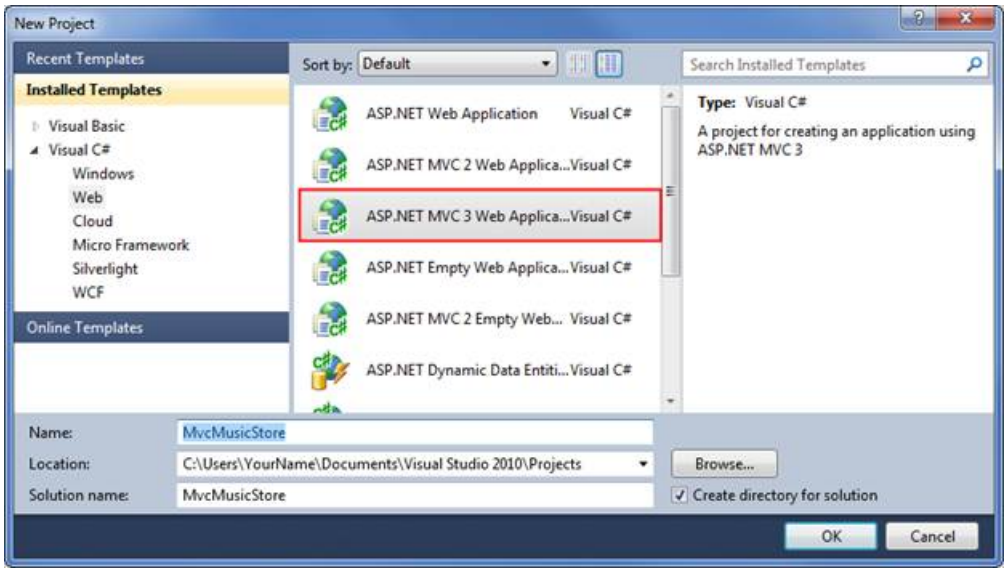
- [Visual Studio Web Developer Express SP1 prerequisites]
- [ASP.NET MVC 3 Tools Update]
- [SQL Server Compact 4.0] - including both runtime and tools support

Creating a new ASP.NET MVC 3 project

We'll start by selecting "New Project" from the File menu in Visual Web Developer. This brings up the New Project dialog.



We'll select the Visual C# -> Web Templates group on the left, then choose the "ASP.NET MVC 3 Web Application" template in the center column. Name your project MvcMusicStore and press the OK button.



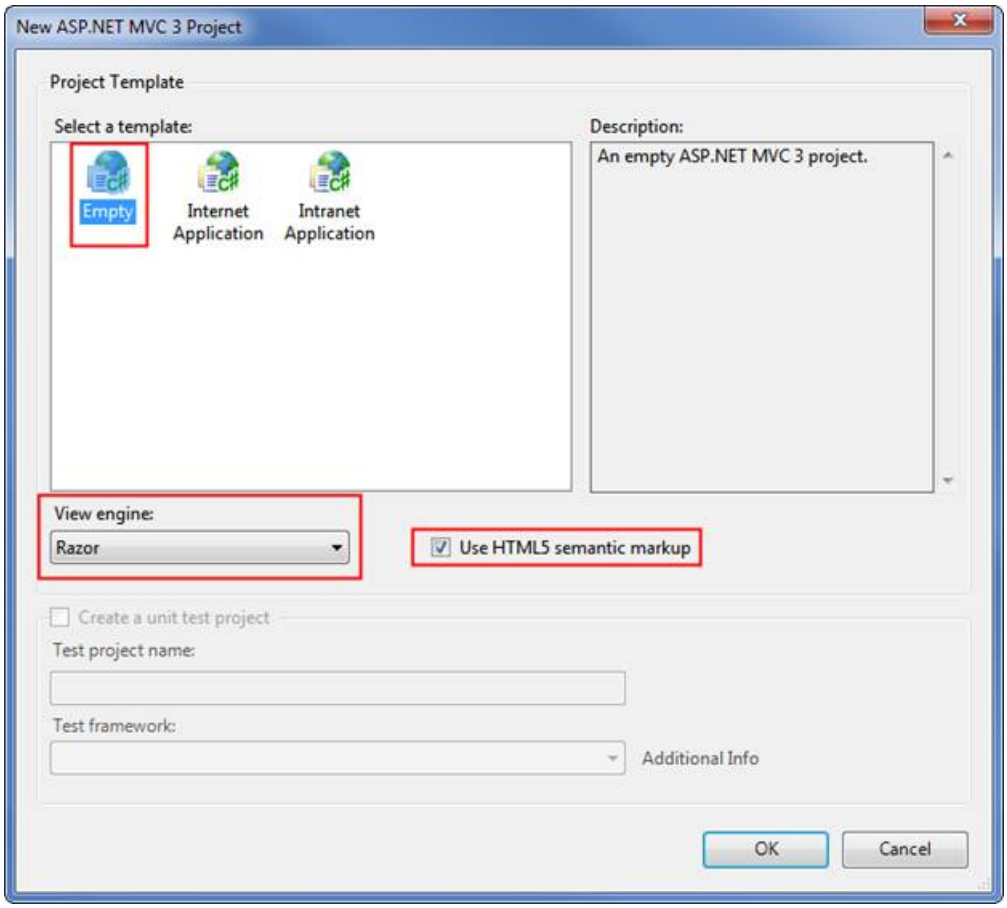
This will display a secondary dialog which allows us to make some MVC specific settings for our project. Select the following:

Project Template - select Empty

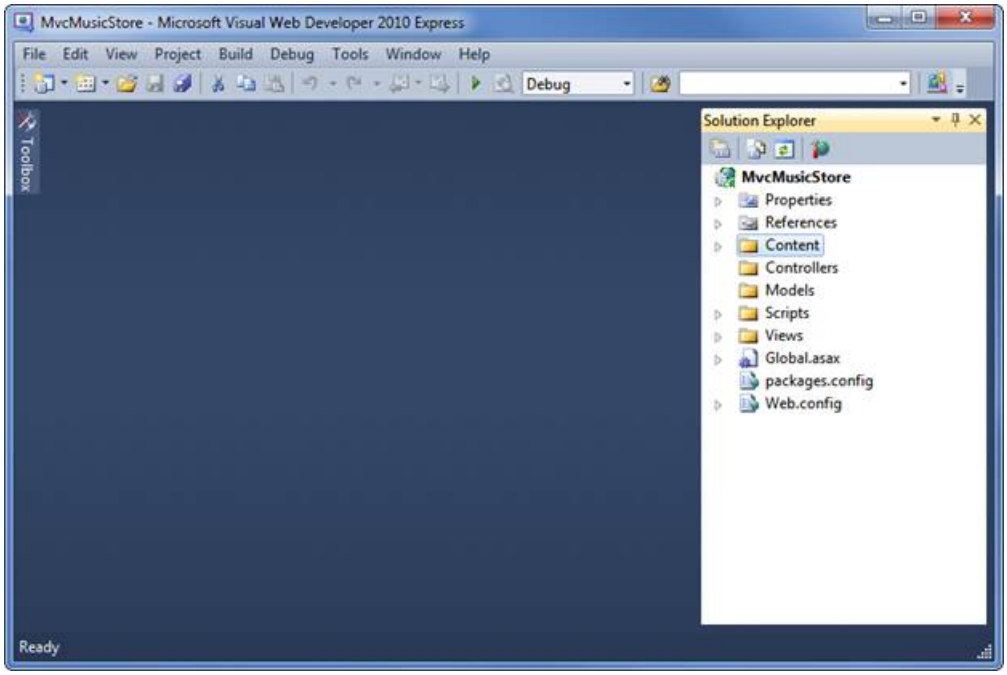
View Engine - select Razor

Use HTML5 semantic markup - checked

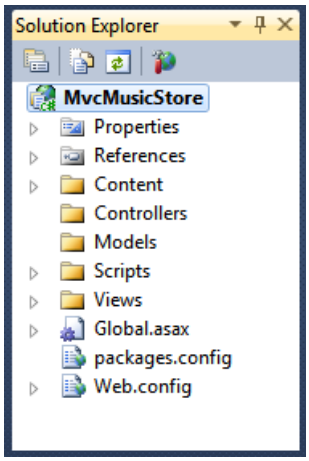
Verify that your settings are as shown below, then press the OK button.



This will create our project. Let's take a look at the folders that have been added to our application in the Solution Explorer on the right side.



The Empty MVC 3 template isn't completely empty – it adds a basic folder structure:



ASP.NET MVC makes use of some basic naming conventions for folder names:

FOLDER	PURPOSE
/Controllers	Controllers respond to input from the browser, decide what to do with it, and return response to the user.
/Views	Views hold our UI templates
/Models	Models hold and manipulate data
/Content	This folder holds our images, CSS, and any other static content
/Scripts	This folder holds our JavaScript files

These folders are included even in an Empty ASP.NET MVC application because the ASP.NET MVC framework by default uses a "convention over configuration" approach and makes some default assumptions based on folder naming conventions. For instance, controllers look for views in the Views folder by default without you having to explicitly specify this in your code. Sticking with the default conventions reduces the amount of code you need to write, and can also make it easier for other developers to understand your project. We'll explain these conventions more as we build our application.

NEXT

Part 2: Controllers

2/20/2020 • 6 minutes to read • Edit Online

by Jon Galloway

> The MVC Music Store is a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Studio for web development.
>
> The MVC Music Store is a lightweight sample store implementation which sells music albums online, and implements basic site administration, user sign-in, and shopping cart functionality.
>
> This tutorial series details all of the steps taken to build the ASP.NET MVC Music Store sample application. Part 2 covers Controllers.

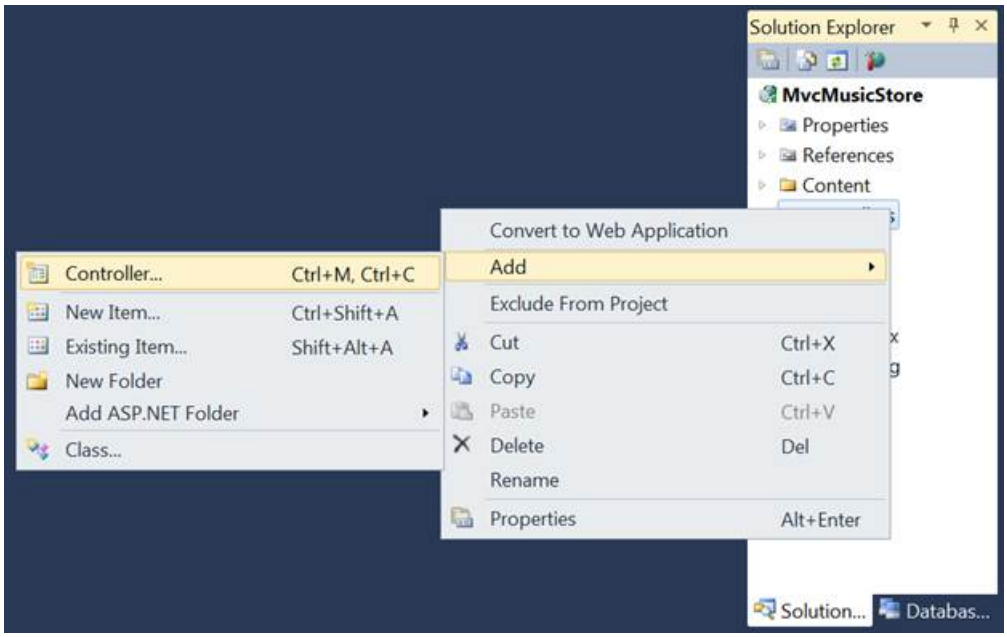
With traditional web frameworks, incoming URLs are typically mapped to files on disk. For example: a request for a URL like "/Products.aspx" or "/Products.php" might be processed by a "Products.aspx" or "Products.php" file.

Web-based MVC frameworks map URLs to server code in a slightly different way. Instead of mapping incoming URLs to files, they instead map URLs to methods on classes. These classes are called "Controllers" and they are responsible for processing incoming HTTP requests, handling user input, retrieving and saving data, and determining the response to send back to the client (display HTML, download a file, redirect to a different URL, etc.).

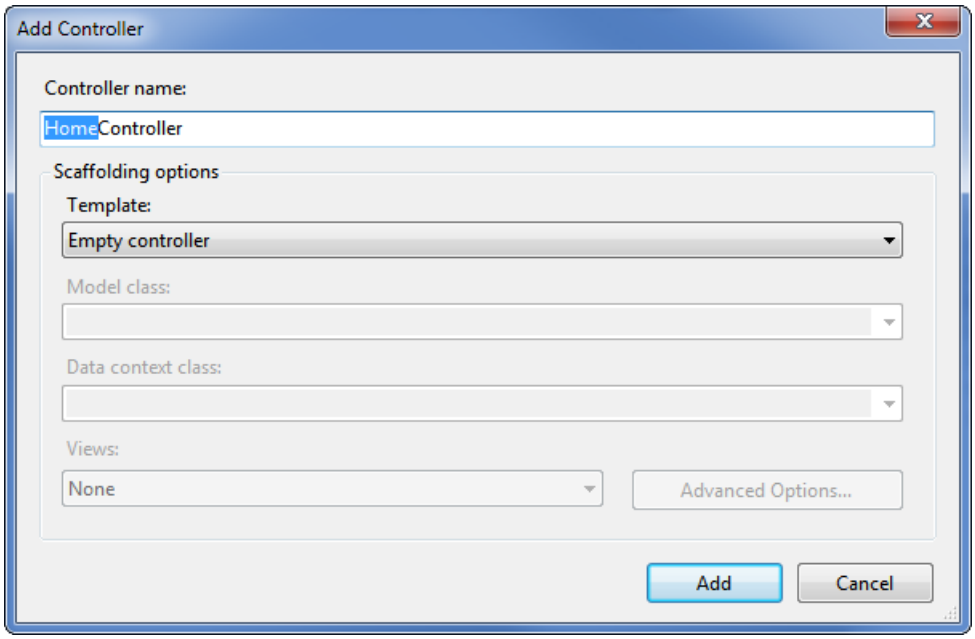
Adding a HomeController

We'll begin our MVC Music Store application by adding a Controller class that will handle URLs to the Home page of our site. We'll follow the default naming conventions of ASP.NET MVC and call it HomeController.

Right-click the "Controllers" folder within the Solution Explorer and select "Add", and then the "Controller..." command:



This will bring up the "Add Controller" dialog. Name the controller "HomeController" and press the Add button.



This will create a new file, HomeController.cs, with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcMusicStore.Controllers
{
    public class HomeController : Controller
    {
        //
        // GET: /Home/
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

To start as simply as possible, let's replace the Index method with a simple method that just returns a string. We'll make two changes:

- Change the method to return a string instead of an ActionResult
- Change the return statement to return "Hello from Home"

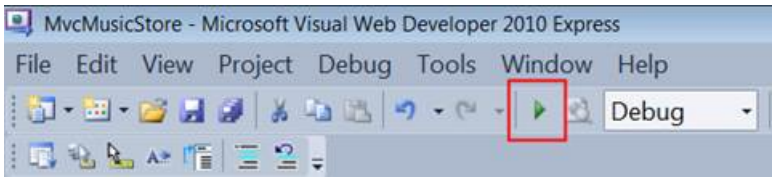
The method should now look like this:

```
public string Index()
{
    return "Hello from Home";
}
```

Running the Application

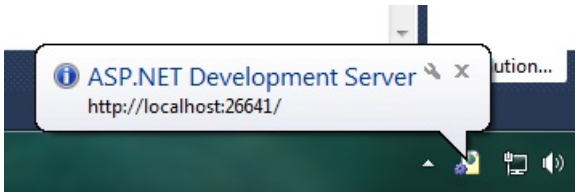
Now let's run the site. We can start our web-server and try out the site using any of the following::

- Choose the Debug ⇨ Start Debugging menu item
- Click the Green arrow button in the toolbar

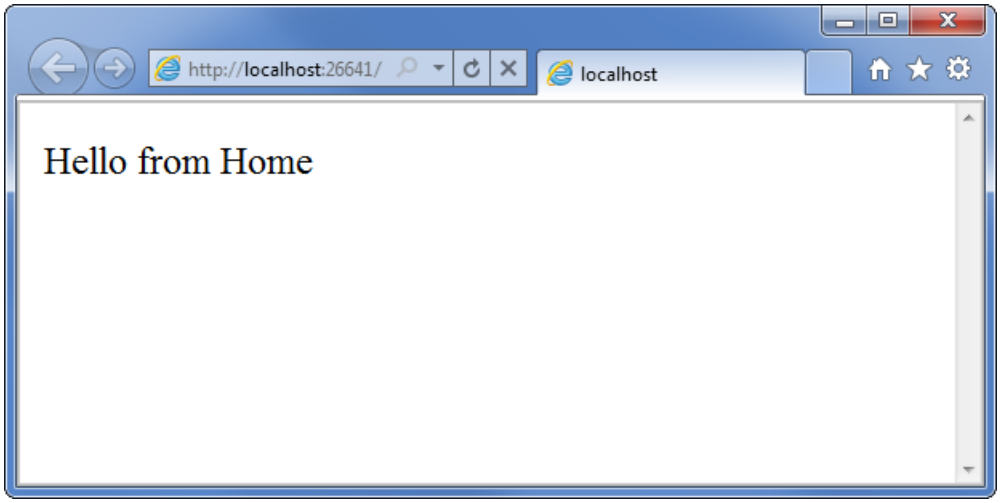


- Use the keyboard shortcut, F5.

Using any of the above steps will compile our project, and then cause the ASP.NET Development Server that is built-into Visual Web Developer to start. A notification will appear in the bottom corner of the screen to indicate that the ASP.NET Development Server has started up, and will show the port number that it is running under.



Visual Web Developer will then automatically open a browser window whose URL points to our web-server. This will allow us to quickly try out our web application:



Okay, that was pretty quick – we created a new website, added a three line function, and we've got text in a browser. Not rocket science, but it's a start.

Note: Visual Web Developer includes the ASP.NET Development Server, which will run your website on a random free "port" number. In the screenshot above, the site is running at `http://localhost:26641/`*, so it's using port 26641. Your port number will be different. When we talk about URL's like /Store/Browse in this tutorial, that will go after the port number. Assuming a port number of 26641, browsing to /Store/Browse will mean browsing to* `http://localhost:26641/Store/Browse`*.*

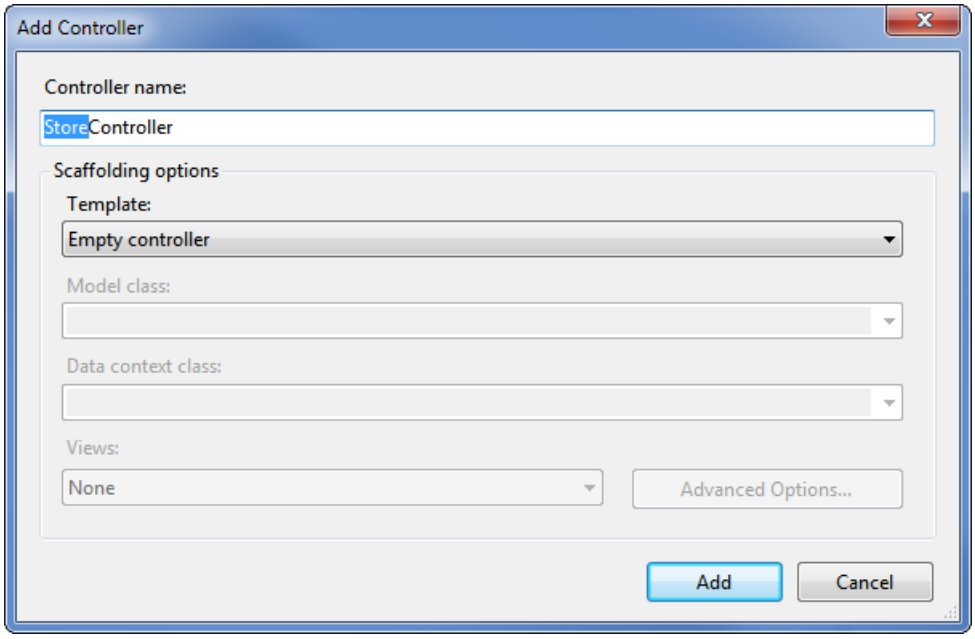
Adding a StoreController

We added a simple HomeController that implements the Home Page of our site. Let's now add another controller that we'll use to implement the browsing functionality of our music store. Our store controller will support three scenarios:

- A listing page of the music genres in our music store
- A browse page that lists all of the music albums in a particular genre
- A details page that shows information about a specific music album

We'll start by adding a new StoreController class.. If you haven't already, stop running the application either by closing the browser or selecting the Debug ⇨ Stop Debugging menu item.

Now add a new StoreController. Just like we did with HomeController, we'll do this by right-clicking on the "Controllers" folder within the Solution Explorer and choosing the Add->Controller menu item



Our new StoreController already has an "Index" method. We'll use this "Index" method to implement our listing page that lists all genres in our music store. We'll also add two additional methods to implement the two other scenarios we want our StoreController to handle: Browse and Details.

These methods (Index, Browse and Details) within our Controller are called "Controller Actions", and as you've already seen with the HomeController.Index()action method, their job is to respond to URL requests and (generally speaking) determine what content should be sent back to the browser or user that invoked the URL.

We'll start our StoreController implementation by changing theIndex() method to return the string "Hello from Store.Index()" and we'll add similar methods for Browse() and Details():

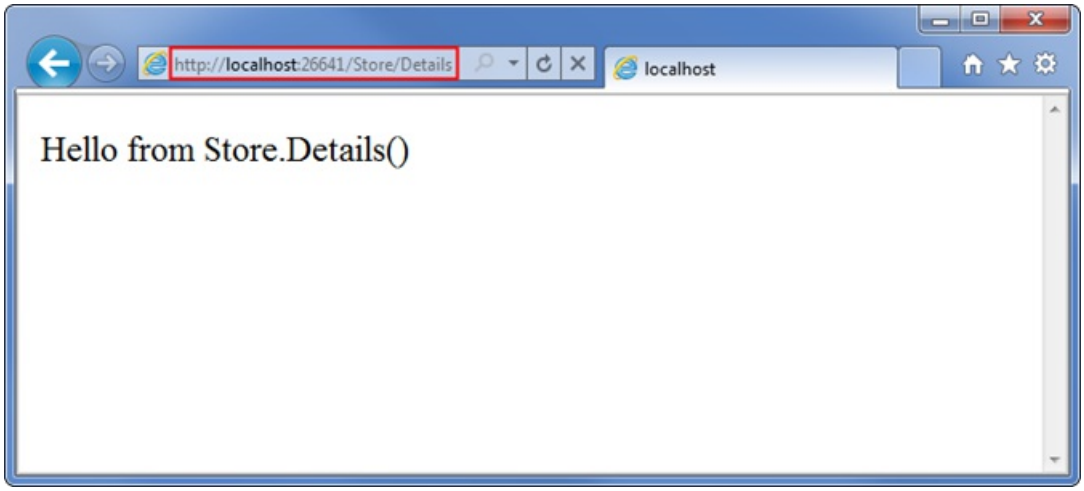
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcMusicStore.Controllers
{
    public class StoreController : Controller
    {
        //
        // GET: /Store/
        public string Index()
        {
            return "Hello from Store.Index()";
        }
        //
        // GET: /Store/Browse
        public string Browse()
        {
            return "Hello from Store.Browse()";
        }
        //
        // GET: /Store/Details
        public string Details()
        {
            return "Hello from Store.Details()";
        }
    }
}
```

Run the project again and browse the following URLs:

- /Store
- /Store/Browse
- /Store/Details

Accessing these URLs will invoke the action methods within our Controller and return string responses:



That's great, but these are just constant strings. Let's make them dynamic, so they take information from the URL and display it in the page output.

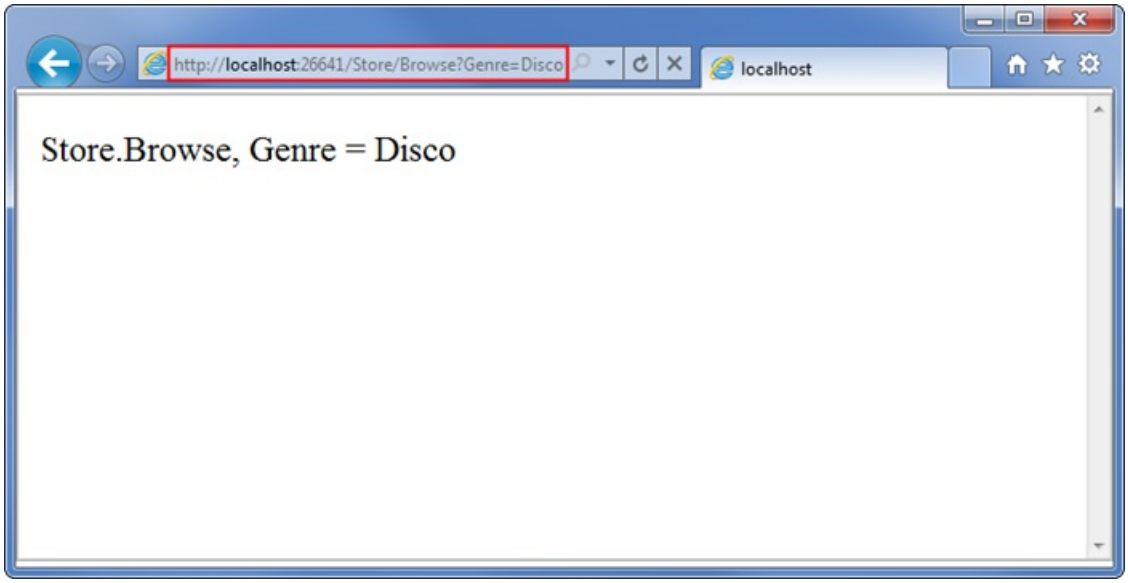
First we'll change the Browse action method to retrieve a querystring value from the URL. We can do this by adding a "genre" parameter to our action method. When we do this ASP.NET MVC will automatically pass any querystring or form post parameters named "genre" to our action method when it is invoked.

```
//
// GET: /Store/Browse?genre=Disco
public string Browse(string genre)
 {
    string message = HttpUtility.HtmlEncode("Store.Browse, Genre = "
+ genre);

    return message;
 }
```

Note: We're using the HttpUtility.HtmlEncode utility method to sanitize the user input. This prevents users from injecting Javascript into our View with a link like /Store/Browse?Genre= <script>window.location='http://hackersite.com'</script>.

Now let's browse to /Store/Browse?Genre=Disco



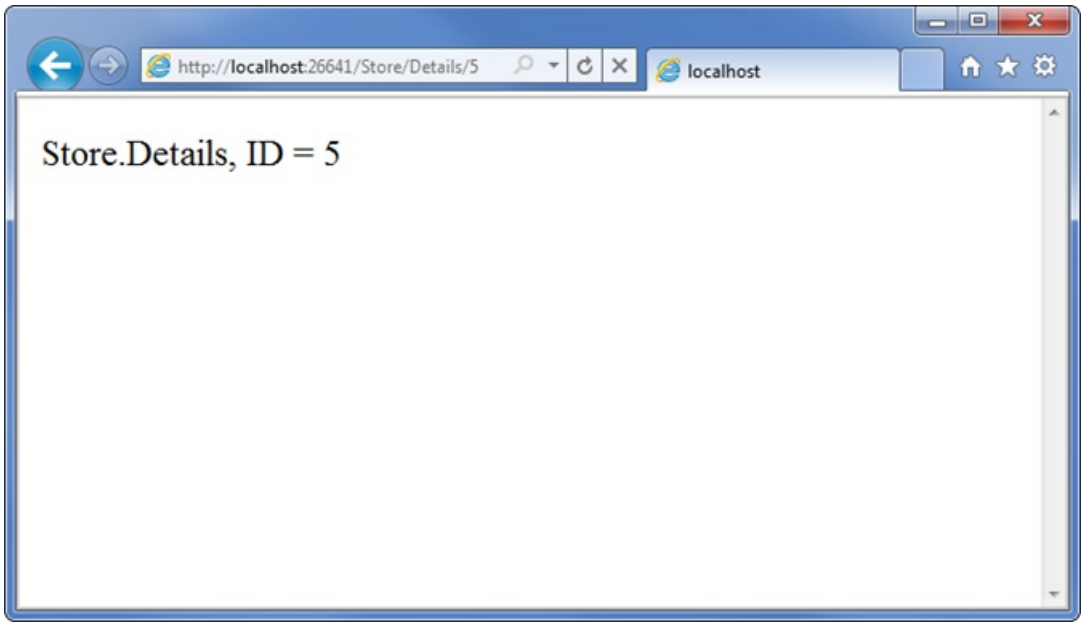
Let's next change the Details action to read and display an input parameter named ID. Unlike our previous method, we won't be embedding the ID value as a querystring parameter. Instead we'll embed it directly within the URL itself. For example: /Store/Details/5.

ASP.NET MVC lets us easily do this without having to configure anything. ASP.NET MVC's default routing convention is to treat the segment of a URL after the action method name as a parameter named "ID". If your action method has a parameter named ID then ASP.NET MVC will automatically pass the URL segment to you as a parameter.

```
//
// GET: /Store/Details/5
public string Details(int id)
 {
    string message = "Store.Details, ID = " + id;

    return message;
 }
```

Run the application and browse to /Store/Details/5:



Let's recap what we've done so far:

- We've created a new ASP.NET MVC project in Visual Web Developer
- We've discussed the basic folder structure of an ASP.NET MVC application
- We've learned how to run our website using the ASP.NET Development Server
- We've created two Controller classes: a HomeController and a StoreController
- We've added Action Methods to our controllers which respond to URL requests and return text to the browser

PREVIOUS NEXT

Part 3: Views and ViewModels

2/20/2020 • 14 minutes to read • Edit Online

by Jon Galloway

> The MVC Music Store is a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Studio for web development.
>
> The MVC Music Store is a lightweight sample store implementation which sells music albums online, and implements basic site administration, user sign-in, and shopping cart functionality.
>
> This tutorial series details all of the steps taken to build the ASP.NET MVC Music Store sample application. Part 3 covers Views and ViewModels.

So far we've just been returning strings from controller actions. That's a nice way to get an idea of how controllers work, but it's not how you'd want to build a real web application. We are going to want a better way to generate HTML back to browsers visiting our site – one where we can use template files to more easily customize the HTML content send back. That's exactly what Views do.

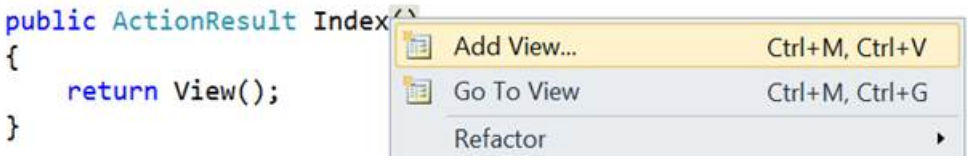
Adding a View template

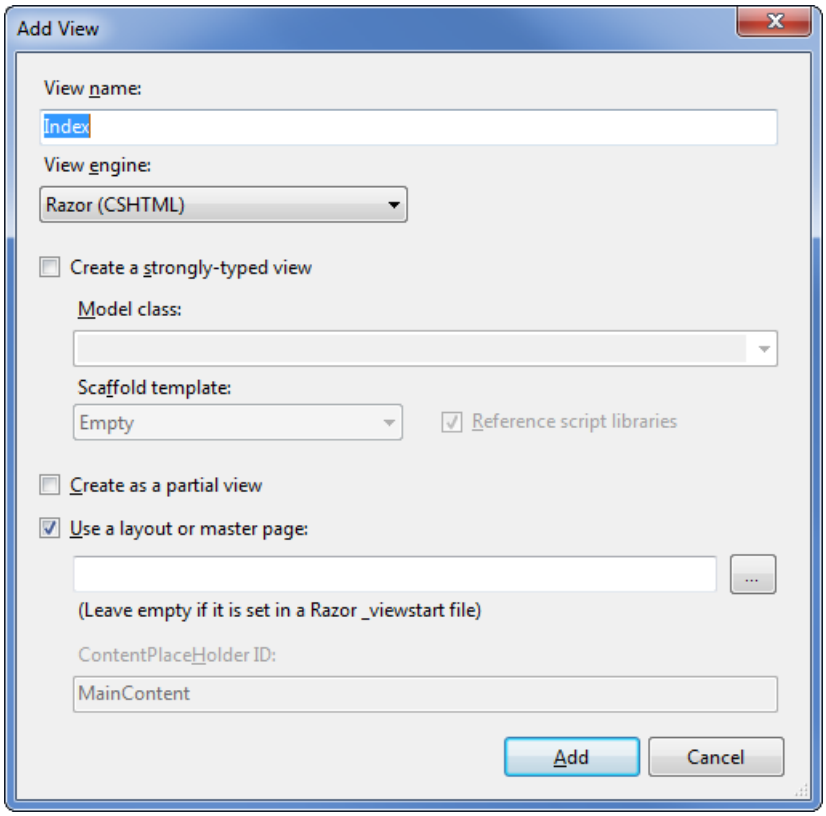
To use a view-template, we'll change the HomeController Index method to return an ActionResult, and have it return View(), like below:

```
public class HomeController : Controller
{
    //
    // GET: /Home/
    public ActionResult Index()
    {
        return View();
    }
}
```

The above change indicates that instead of returned a string, we instead want to use a "View" to generate a result back.

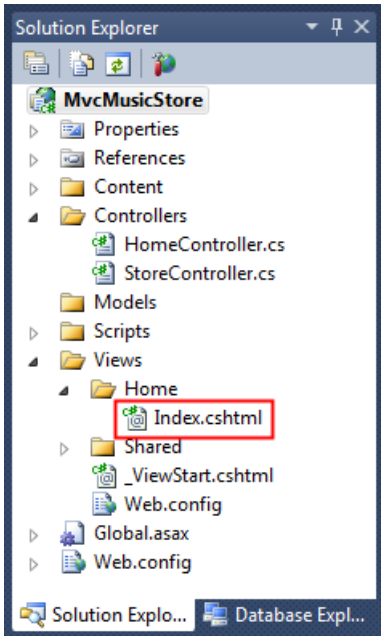
We'll now add an appropriate View template to our project. To do this we'll position the text cursor within the Index action method, then right-click and select "Add View". This will bring up the Add View dialog:





The "Add View" dialog allows us to quickly and easily generate View template files. By default the "Add View" dialog pre-populates the name of the View template to create so that it matches the action method that will use it. Because we used the "Add View" context menu within the Index() action method of our HomeController, the "Add View" dialog above has "Index" as the view name pre-populated by default. We don't need to change any of the options on this dialog, so click the Add button.

When we click the Add button, Visual Web Developer will create a new Index.cshtml view template for us in the \Views\Home directory, creating the folder if doesn't already exist.



The name and folder location of the "Index.cshtml" file is important, and follows the default ASP.NET MVC naming conventions. The directory name, \Views\Home, matches the controller - which is named HomeController. The view template name, Index, matches the controller action method which will be displaying the view.

ASP.NET MVC allows us to avoid having to explicitly specify the name or location of a view template when we use this naming convention to return a view. It will by default render the \Views\Home\Index.cshtml view template when we write code like below within our HomeController:

```
public class HomeController : Controller
{
    //
    // GET: /Home/
    public ActionResult Index()
    {
        return View();
    }
}
```

Visual Web Developer created and opened the "Index.cshtml" view template after we clicked the "Add" button within the "Add View" dialog. The contents of Index.cshtml are shown below.

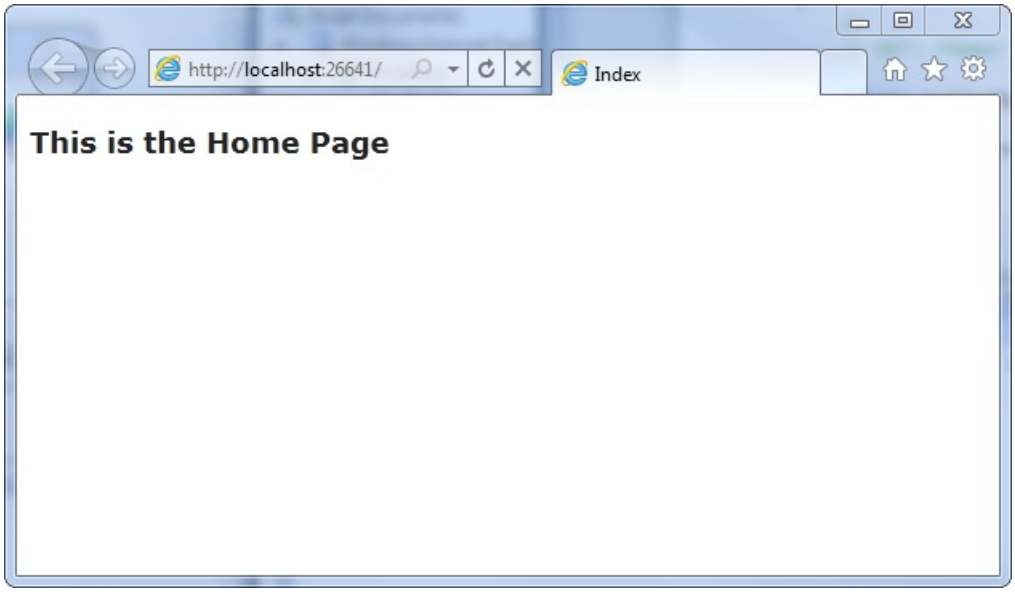
```
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
```

This view is using the Razor syntax, which is more concise than the Web Forms view engine used in ASP.NET Web Forms and previous versions of ASP.NET MVC. The Web Forms view engine is still available in ASP.NET MVC 3, but many developers find that the Razor view engine fits ASP.NET MVC development really well.

The first three lines set the page title using ViewBag.Title. We'll look at how this works in more detail soon, but first let's update the text heading text and view the page. Update the <h2> tag to say "This is the Home Page" as shown below.

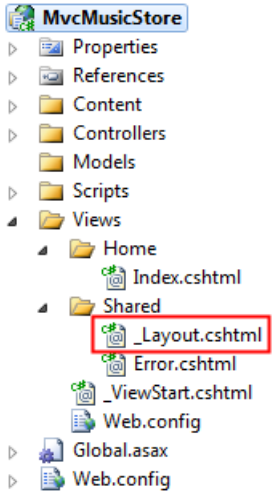
```
@{
    ViewBag.Title = "Index";
}
<h2>This is the Home Page</h2>
```

Running the application shows that our new text is visible on the home page.



Using a Layout for common site elements

Most websites have content which is shared between many pages: navigation, footers, logo images, stylesheet references, etc. The Razor view engine makes this easy to manage using a page called _Layout.cshtml which has automatically been created for us inside the /Views/Shared folder.



Double-click on this folder to view the contents, which are shown below.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")"
rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"
            type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/modernizr-1.7.min.js")"
            type="text/javascript"></script>
</head>
<body>
    @RenderBody()
</body>
</html>
```

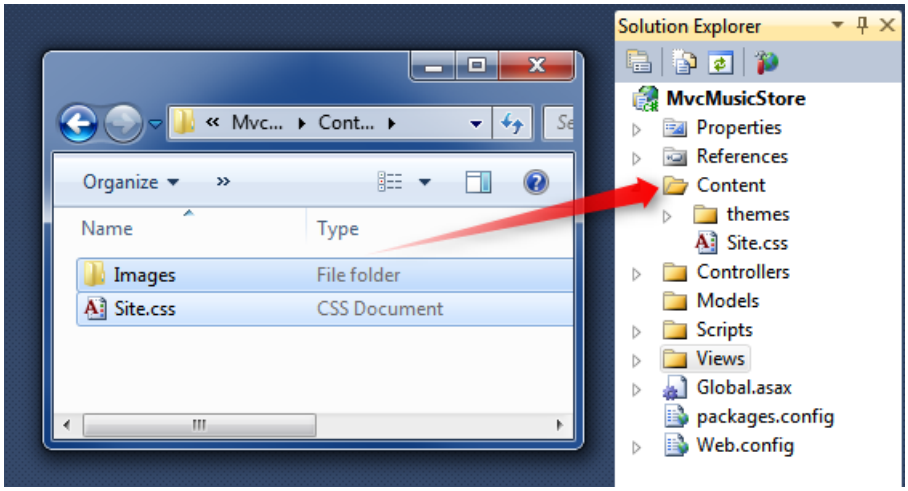
The content from our individual views will be displayed by the @RenderBody() command, and any common content that we want to appear outside of that can be added to the _Layout.cshtml markup. We'll want our MVC Music Store to have a common header with links to our Home page and Store area on all pages in the site, so we'll add that to the template directly above that @RenderBody() statement.

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")"
rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")"
type="text/javascript"></script>
</head>
<body>
    <div id="header">
        <h1>
            ASP.NET MVC MUSIC STORE</h1>
        <ul id="navlist">
            <li class="first"><a href="/"
id="current">Home</a></li>
            <li><a
href="/Store/">Store</a></li>
        </ul>
    </div>
    @RenderBody()
</body>
</html>
```

Updating the StyleSheet

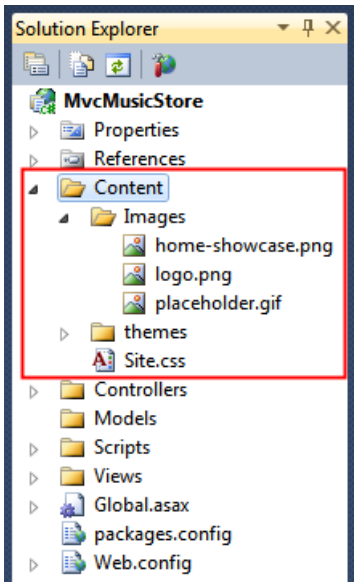
The empty project template includes a very streamlined CSS file which just includes styles used to display validation messages. Our designer has provided some additional CSS and images to define the look and feel for our site, so we'll add those in now.

The updated CSS file and Images are included in the Content directory of MvcMusicStore-Assets.zip which is available at MVC-Music-Store. We'll select both of them in Windows Explorer and drop them into our Solution's Content folder in Visual Web Developer, as shown below:

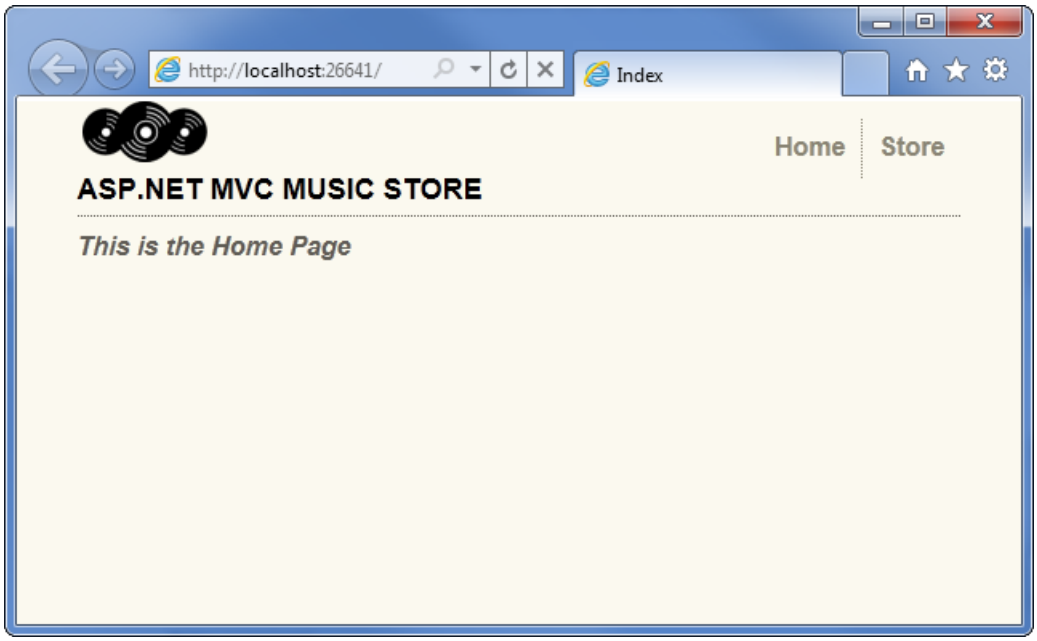


You'll be asked to confirm if you want to overwrite the existing Site.css file. Click Yes.

The Content folder of your application will now appear as follows:



Now let's run the application and see how our changes look on the Home page.



- Let's review what's changed: The HomeController's Index action method found and displayed the \Views\Home\Index.cshtmlView template, even though our code called "return View()", because our View template followed the standard naming convention.
- The Home Page is displaying a simple welcome message that is defined within the \Views\Home\Index.cshtml view template.
- The Home Page is using our _Layout.cshtml template, and so the welcome message is contained within the standard site HTML layout.

Using a Model to pass information to our View

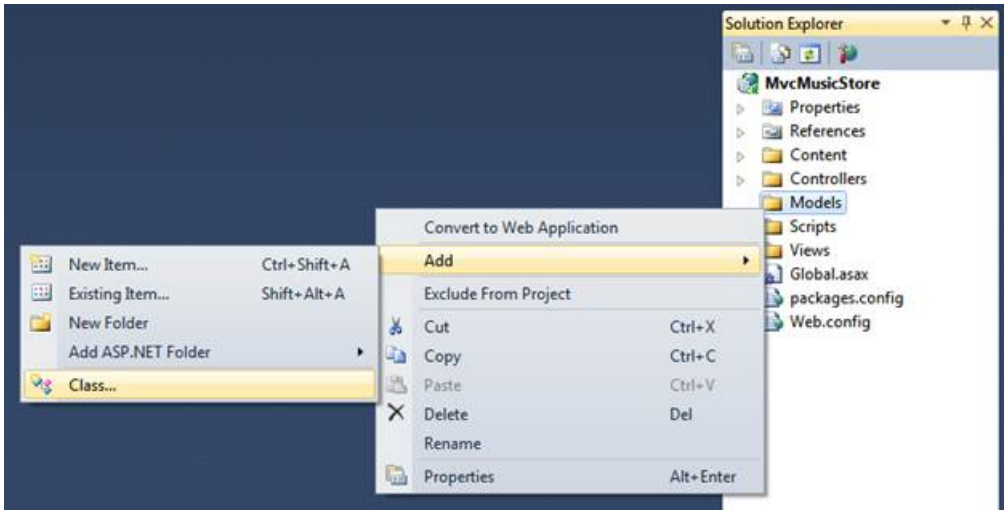
A View template that just displays hardcoded HTML isn't going to make a very interesting web site. To create a dynamic web site, we'll instead want to pass information from our controller actions to our view templates.

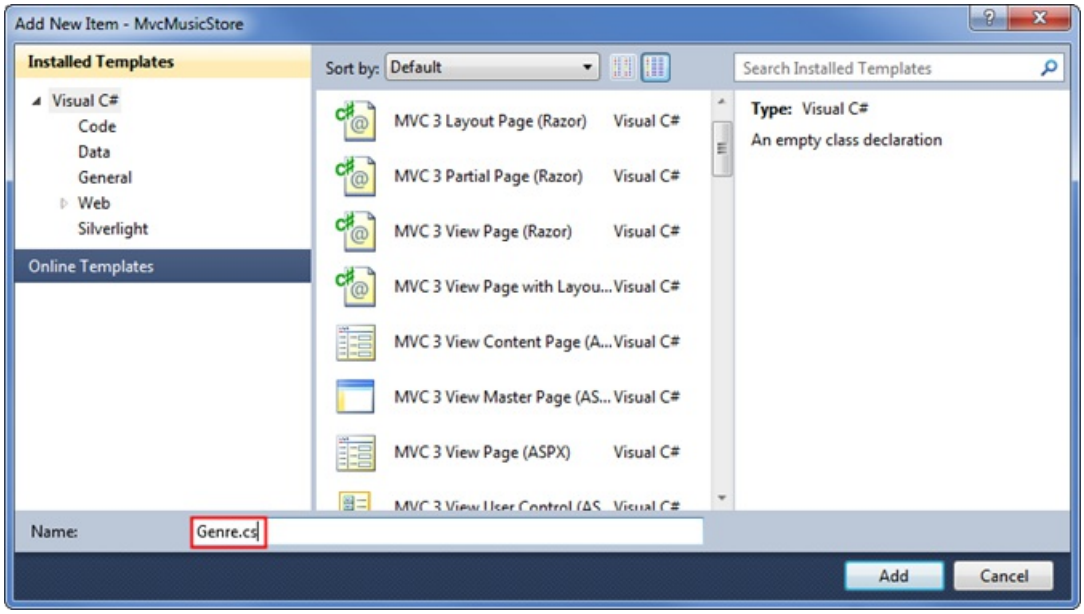
In the Model-View-Controller pattern, the term Model refers to objects which represent the data in the application. Often, model objects correspond to tables in your database, but they don't have to.

Controller action methods which return an ActionResult can pass a model object to the view. This allows a Controller to cleanly package up all the information needed to generate a response, and then pass this information off to a View template to use to generate the appropriate HTML response. This is easiest to understand by seeing it

in action, so let's get started.

First we'll create some Model classes to represent Genres and Albums within our store. Let's start by creating a Genre class. Right-click the "Models" folder within your project, choose the "Add Class" option, and name the file "Genre.cs".





Then add a public string Name property to the class that was created:

```
public class Genre
{
    public string Name { get; set; }
}
```

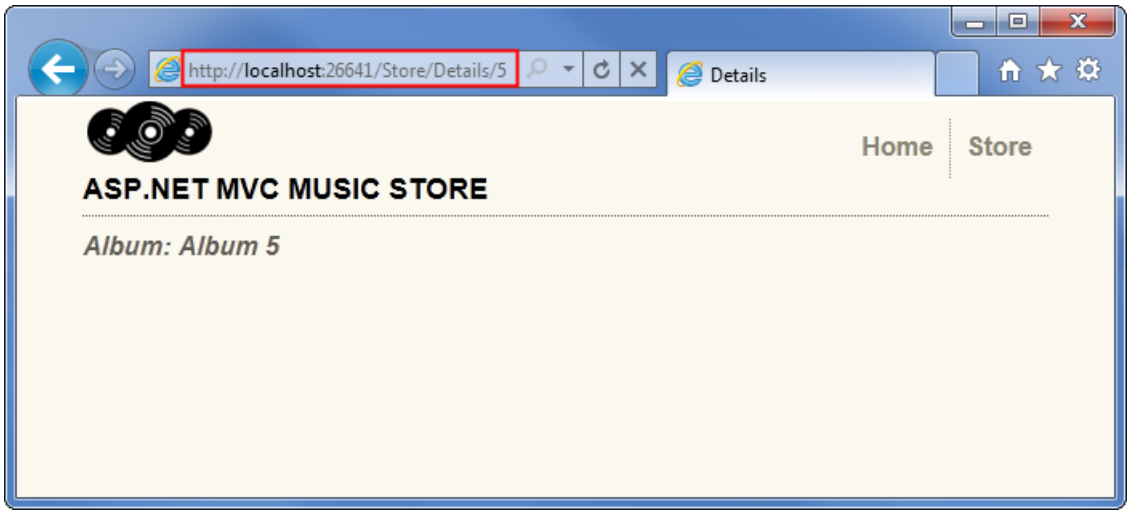
Note: In case you're wondering, the { get; set; } notation is making use of C#'s auto-implemented properties feature. This gives us the benefits of a property without requiring us to declare a backing field.

Next, follow the same steps to create an Album class (named Album.cs) that has a Title and a Genre property:

```
public class Album
{
    public string Title { get; set; }
    public Genre Genre { get; set; }
}
```

Now we can modify the StoreController to use Views which display dynamic information from our Model. If - for

demonstration purposes right now - we named our Albums based on the request ID, we could display that information as in the view below.



We'll start by changing the Store Details action so it shows the information for a single album. Add a "using" statement to the top of the **StoreControllers** class to include the MvcMusicStore.Models namespace, so we don't need to type MvcMusicStore.Models.Album every time we want to use the album class. The "usings" section of that class should now appear as below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MvcMusicStore.Models;
```

Next, we'll update the Details controller action so that it returns an ActionResult rather than a string, as we did with the HomeController's Index method.

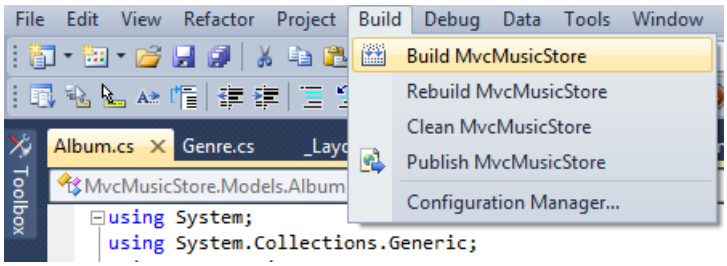
```
public ActionResult Details(int id)
```

Now we can modify the logic to return an Album object to the view. Later in this tutorial we will be retrieving the data from a database – but for right now we will use "dummy data" to get started.

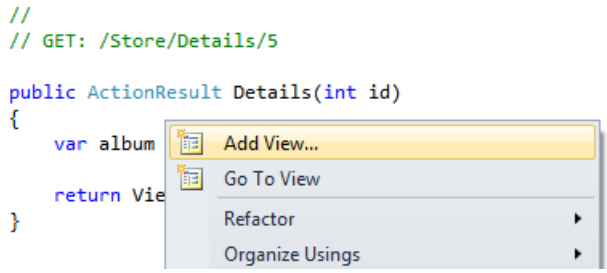
```
public ActionResult Details(int id)
 {
    var album = new Album { Title = "Album " + id };
    return View(album);
 }
```

Note: If you're unfamiliar with C#, you may assume that using var means that our album variable is late-bound. That's not correct – the C# compiler is using type-inference based on what we're assigning to the variable to determine that album is of type Album and compiling the local album variable as an Album type, so we get compile-time checking and Visual Studio code-editor support.

Let's now create a View template that uses our Album to generate an HTML response. Before we do that we need to build the project so that the Add View dialog knows about our newly created Album class. You can build the project by selecting the Debug⇨Build MvcMusicStore menu item (for extra credit, you can use the Ctrl-Shift-B shortcut to build the project).

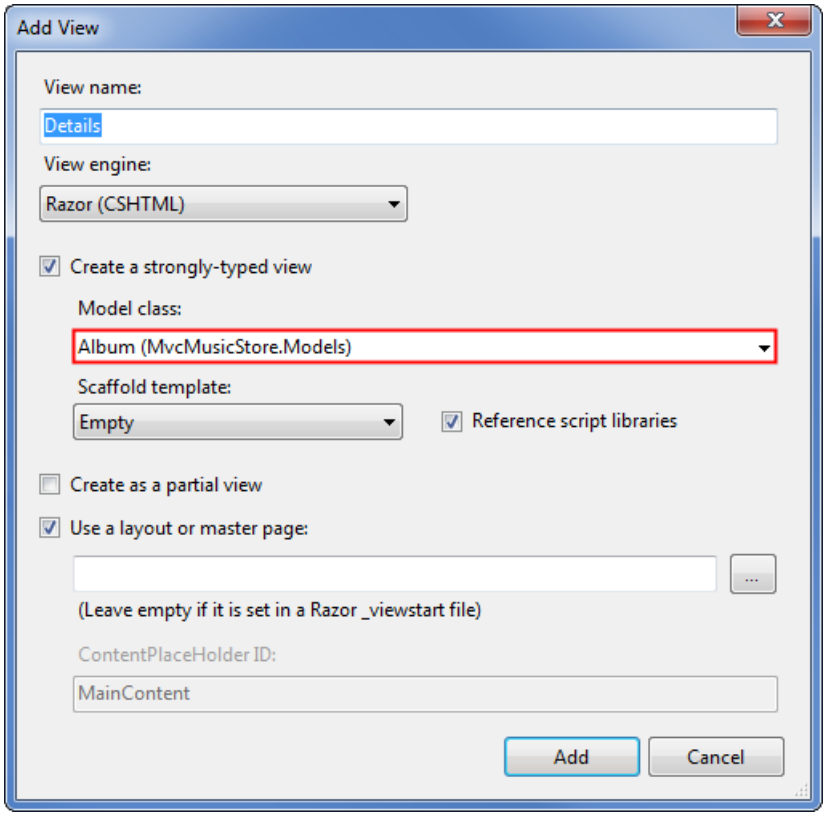


Now that we've set up our supporting classes, we're ready to build our View template. Right-click within the Details method and select "Add View..." from the context menu.



We are going to create a new View template like we did before with the HomeController. Because we are creating it from the StoreController it will by default be generated in a \Views\Store\Index.cshtml file.

Unlike before, we are going to check the "Create a strongly-typed" view checkbox. We are then going to select our "Album" class within the "View data-class" drop-downlist. This will cause the "Add View" dialog to create a View template that expects that an Album object will be passed to it to use.



When we click the "Add" button our \Views\Store\Details.cshtml View template will be created, containing the following code.

```
@model MvcMusicStore.Models.Album
@{
    ViewBag.Title = "Details";
}
<h2>Details</h2>
```

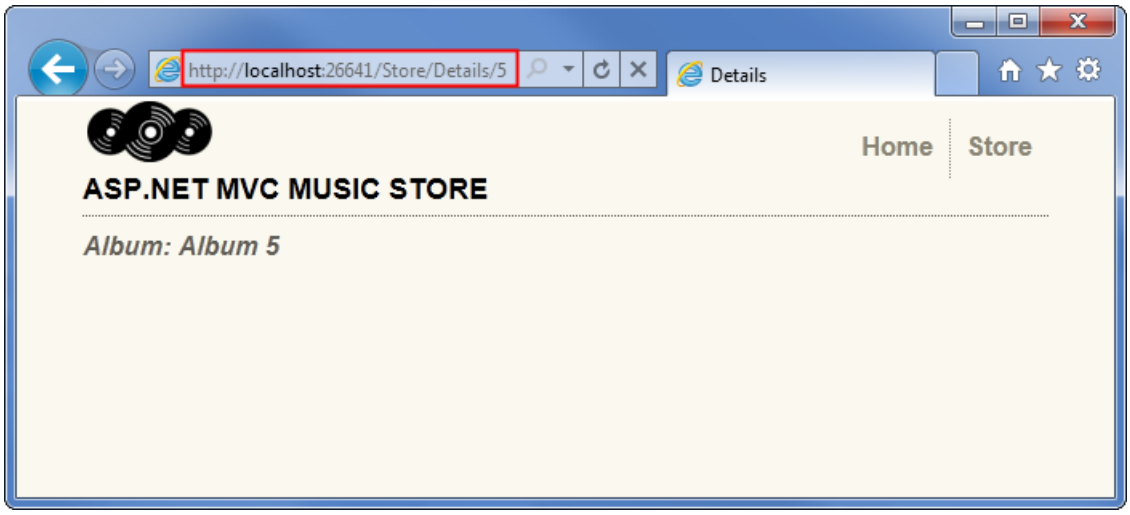
Notice the first line, which indicates that this view is strongly-typed to our Album class. The Razor view engine understands that it has been passed an Album object, so we can easily access model properties and even have the benefit of IntelliSense in the Visual Web Developer editor.

Update the <h2> tag so it displays the Album's Title property by modifying that line to appear as follows.

```
<h2>Album: @Model.Title</h2>
```

Notice that IntelliSense is triggered when you enter the period after the @Model keyword, showing the properties and methods that the Album class supports.

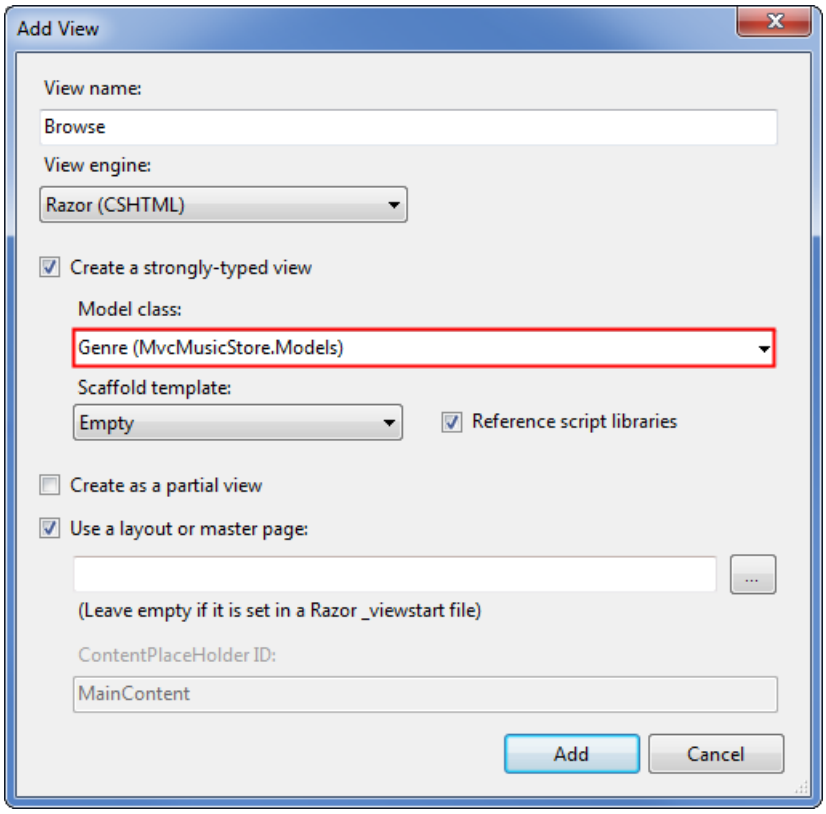
Let's now re-run our project and visit the /Store/Details/5 URL. We'll see details of an Album like below.



Now we'll make a similar update for the Store Browse action method. Update the method so it returns an ActionResult, and modify the method logic so it creates a new Genre object and returns it to the View.

```
public ActionResult Browse(string genre)
 {
    var genreModel = new Genre { Name = genre };
    return View(genreModel);
 }
```

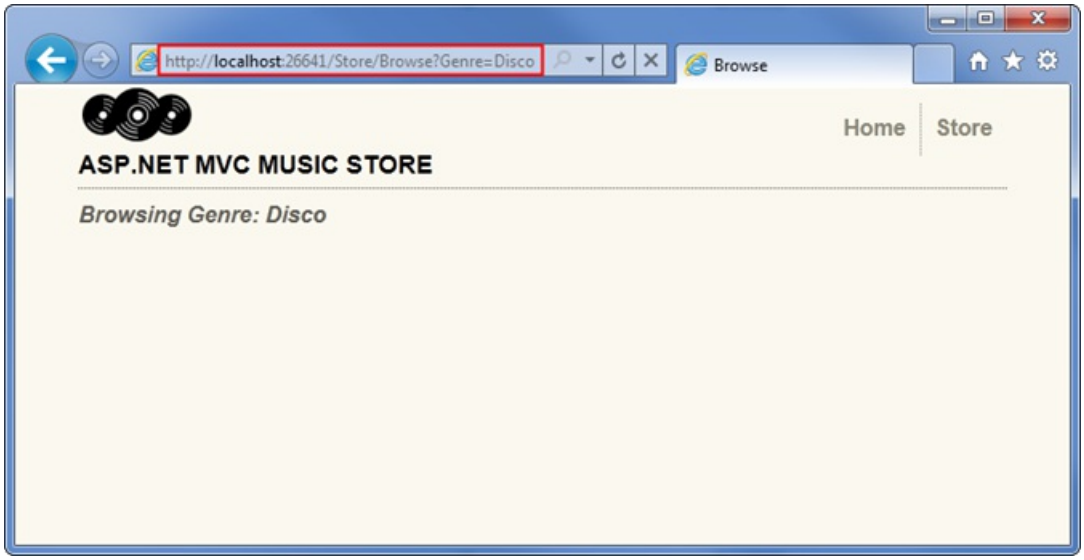
Right-click in the Browse method and select "Add View..." from the context menu, then add a View that is strongly-typed add a strongly typed to the Genre class.



Update the <h2> element in the view code (in /Views/Store/Browse.cshtml) to display the Genre information.

```
@model MvcMusicStore.Models.Genre
@{
    ViewBag.Title = "Browse";
}
<h2>Browsing Genre: @Model.Name</h2>
```

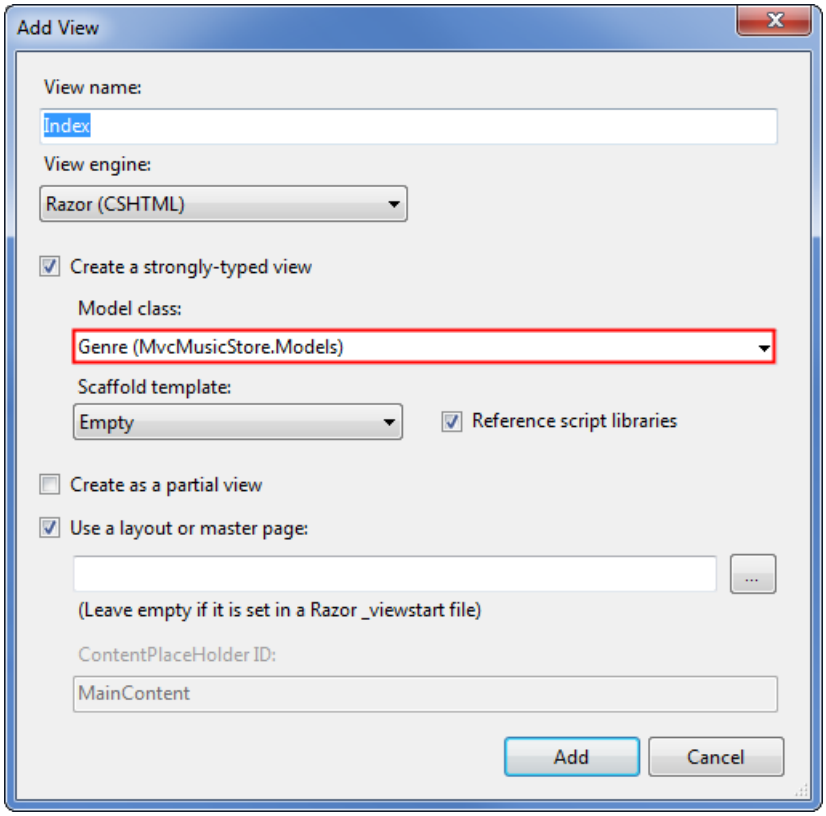
Now let's re-run our project and browse to the /Store/Browse?Genre=Disco URL. We'll see the Browse page displayed like below.



Finally, let's make a slightly more complex update to the **Store Index** action method and view to display a list of all the Genres in our store. We'll do that by using a List of Genres as our model object, rather than just a single Genre.

```
public ActionResult Index()
{
    var genres = new List<Genre>
    {
        new Genre { Name = "Disco"},
        new Genre { Name = "Jazz"},
        new Genre { Name = "Rock"}
    };
    return View(genres);
 }
```

Right-click in the Store Index action method and select Add View as before, select Genre as the Model class, and press the Add button.



First we'll change the @model declaration to indicate that the view will be expecting several Genre objects rather than just one. Change the first line of /Store/Index.cshtml to read as follows:

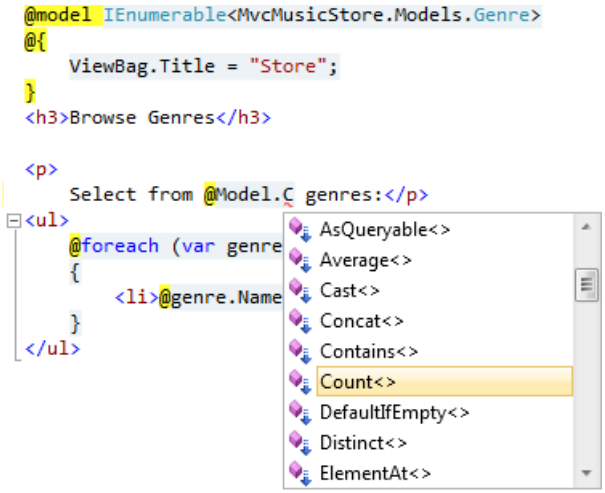
```
@model IEnumerable<MvcMusicStore.Models.Genre>
```

This tells the Razor view engine that it will be working with a model object that can hold several Genre objects. We're using an IEnumerable<Genre> rather than a List<Genre> since it's more generic, allowing us to change our model type later to any object type that supports the IEnumerable interface.

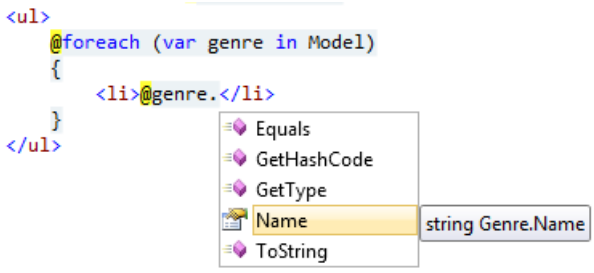
Next, we'll loop through the Genre objects in the model as shown in the completed view code below.

```
@model IEnumerable<MvcMusicStore.Models.Genre>
@{
    ViewBag.Title = "Store";
}
<h3>Browse Genres</h3>
<p>
    Select from @Model.Count()
genres:</p>
<ul>
    @foreach (var genre in Model)
    {
        <li>@genre.Name</li>
    }
</ul>
```

Notice that we have full IntelliSense support as we enter this code, so that when we type "@Model." we see all methods and properties supported by an IEnumerable of type Genre.

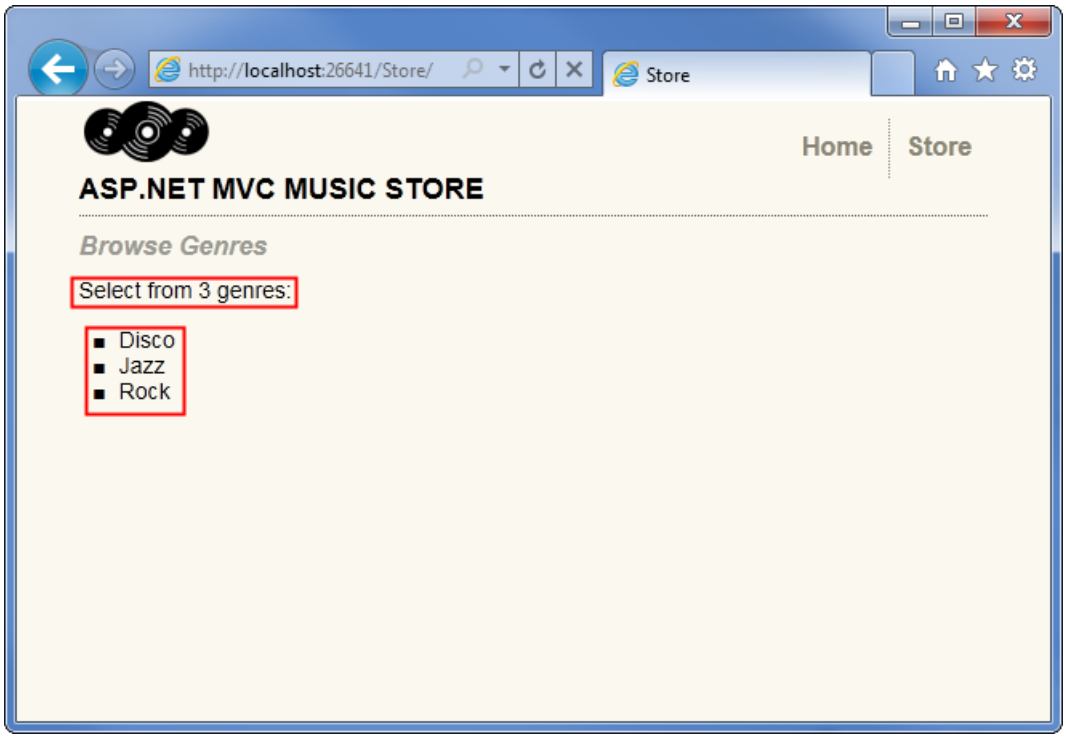


Within our "foreach" loop, Visual Web Developer knows that each item is of type Genre, so we see IntelliSense for each the Genre type.



Next, the scaffolding feature examined the Genre object and determined that each will have a Name property, so it loops through and writes them out. It also generates Edit, Details, and Delete links to each individual item. We'll take advantage of that later in our store manager, but for now we'd like to have a simple list instead.

When we run the application and browse to /Store, we see that both the count and list of Genres is displayed.



Adding Links between pages

Our /Store URL that lists Genres currently lists the Genre names simply as plain text. Let's change this so that instead of plain text we instead have the Genre names link to the appropriate /Store/Browse URL, so that clicking on a music genre like "Disco" will navigate to the /Store/Browse?genre=Disco URL. We could update our \Views\Store\Index.cshtml View template to output these links using code like below **(don't type this in - we're going to improve on it)**:

```
<ul>
    @foreach (var genre in Model)
    {
        <li><a href="/Store/Browse?genre=@genre.Name">@genre.Name</a></li>
    }
</ul>
```

That works, but it could lead to trouble later since it relies on a hardcoded string. For instance, if we wanted to rename the Controller, we'd need to search through our code looking for links that need to be updated.

An alternative approach we can use is to take advantage of an HTML Helper method. ASP.NET MVC includes HTML Helper methods which are available from our View template code to perform a variety of common tasks just like this. The Html.ActionLink() helper method is a particularly useful one, and makes it easy to build HTML <a> links and takes care of annoying details like making sure URL paths are properly URL encoded.

Html.ActionLink() has several different overloads to allow specifying as much information as you need for your links. In the simplest case, you'll supply just the link text and the Action method to go to when the hyperlink is clicked on the client. For example, we can link to "/Store/" Index() method on the Store Details page with the link text "Go to the Store Index" using the following call:

```
@Html.ActionLink("Go
to the Store Index", "Index")
```

Note: In this case, we didn't need to specify the controller name because we're just linking to another action within the same controller that's rendering the current view.

Our links to the Browse page will need to pass a parameter, though, so we'll use another overload of the

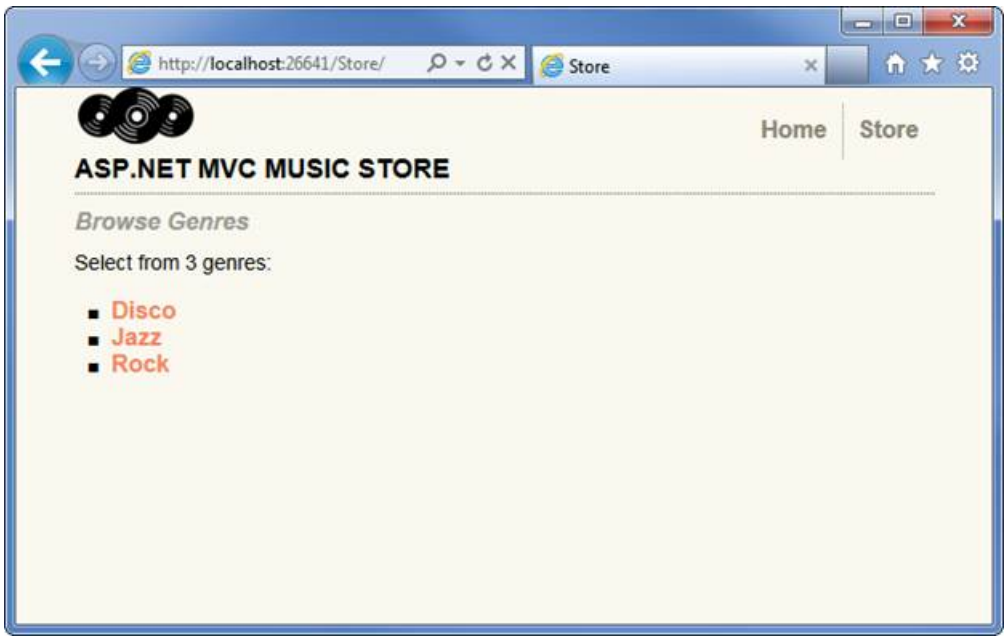
Html.ActionLink method that takes three parameters:

- 1. Link text, which will display the Genre name
- 2. Controller action name (Browse)
- 3. Route parameter values, specifying both the name (Genre) and the value (Genre name)

Putting that all together, here's how we'll write those links to the Store Index view:

```
<ul>
    @foreach (var genre in Model)
    {
        <li>@Html.ActionLink(genre.Name,
"Browse", new { genre = genre.Name })</li>
    }
</ul>
```

Now when we run our project again and access the /Store/ URL we will see a list of genres. Each genre is a hyperlink – when clicked it will take us to our /Store/Browse?genre=*[genre]* URL.



The HTML for the genre list looks like this:

```
<ul>
    <li><a href="/Store/Browse?genre=Disco">Disco</a>
</li>
    <li><a href="/Store/Browse?genre=Jazz">Jazz</a>
</li>
    <li><a href="/Store/Browse?genre=Rock">Rock</a>
</li>
</ul>
```

PREVIOUS NEXT

Part 4: Models and Data Access

2/20/2020 • 7 minutes to read • Edit Online

by Jon Galloway

> The MVC Music Store is a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Studio for web development.
>
> The MVC Music Store is a lightweight sample store implementation which sells music albums online, and implements basic site administration, user sign-in, and shopping cart functionality.
>
> This tutorial series details all of the steps taken to build the ASP.NET MVC Music Store sample application. Part 4 covers Models and Data Access.

So far, we've just been passing "dummy data" from our Controllers to our View templates. Now we're ready to hook up a real database. In this tutorial we'll be covering how to use SQL Server Compact Edition (often called SQL CE) as our database engine. SQL CE is a free, embedded, file based database that doesn't require any installation or configuration, which makes it really convenient for local development.

Database access with Entity Framework Code-First

We'll use the Entity Framework (EF) support that is included in ASP.NET MVC 3 projects to query and update the database. EF is a flexible object relational mapping (ORM) data API that enables developers to query and update data stored in a database in an object-oriented way.

Entity Framework version 4 supports a development paradigm called code-first. Code-first allows you to create model object by writing simple classes (also known as POCO from "plain-old" CLR objects), and can even create the database on the fly from your classes.

Changes to our Model Classes

We will be leveraging the database creation feature in Entity Framework in this tutorial. Before we do that, though, let's make a few minor changes to our model classes to add in some things we'll be using later on.

Adding the Artist Model Classes

Our Albums will be associated with Artists, so we'll add a simple model class to describe an Artist. Add a new class to the Models folder named Artist.cs using the code shown below.

```
namespace MvcMusicStore.Models
{
    public class Artist
    {
        public int ArtistId { get; set; }
        public string Name { get; set; }
    }
}
```

Updating our Model Classes

Update the Album class as shown below.

```
namespace MvcMusicStore.Models
{
    public class Album
    {
        public int      AlbumId     { get; set; }
        public int      GenreId     { get; set; }
        public int      ArtistId    { get; set; }
        public string   Title       { get; set; }
        public decimal  Price       { get; set; }
        public string   AlbumArtUrl { get; set; }
        public Genre    Genre       { get; set; }
        public Artist   Artist      { get; set; }
    }
}
```

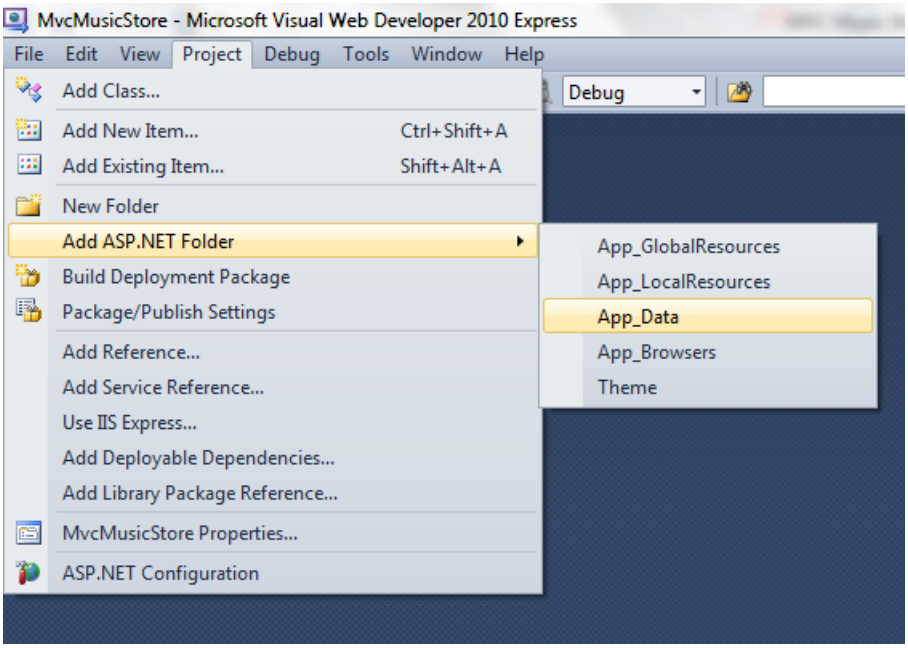
Next, make the following updates to the Genre class.

```
using System.Collections.Generic;

namespace MvcMusicStore.Models
{
    public partial class Genre
    {
        public int      GenreId     { get; set; }
        public string   Name        { get; set; }
        public string   Description { get; set; }
        public List<Album> Albums   { get; set; }
    }
}
```

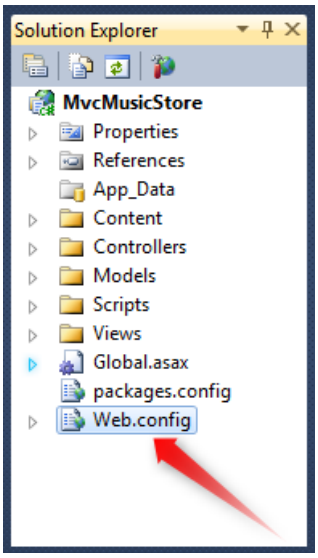
Adding the App_Data folder

We'll add an App_Data directory to our project to hold our SQL Server Express database files. App_Data is a special directory in ASP.NET which already has the correct security access permissions for database access. From the Project menu, select Add ASP.NET Folder, then App_Data.



Creating a Connection String in the web.config file

We will add a few lines to the website's configuration file so that Entity Framework knows how to connect to our database. Double-click on the Web.config file located in the root of the project.

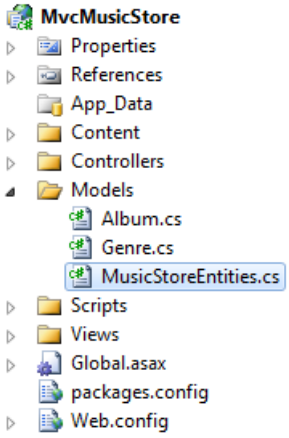


Scroll to the bottom of this file and add a <connectionStrings> section directly above the last line, as shown below.

```
<connectionStrings>
    <add name="MusicStoreEntities"
     connectionString="Data Source=|DataDirectory|MvcMusicStore.sdf"
     providerName="System.Data.SqlServerCe.4.0"/>
  </connectionStrings>
</configuration>
```

Adding a Context Class

Right-click the Models folder and add a new class named MusicStoreEntities.cs.



This class will represent the Entity Framework database context, and will handle our create, read, update, and delete operations for us. The code for this class is shown below.

```
using System.Data.Entity;

namespace MvcMusicStore.Models
{
    public class MusicStoreEntities : DbContext
    {
        public DbSet<Album> Albums { get; set; }
        public DbSet<Genre> Genres { get; set; }
    }
}
```

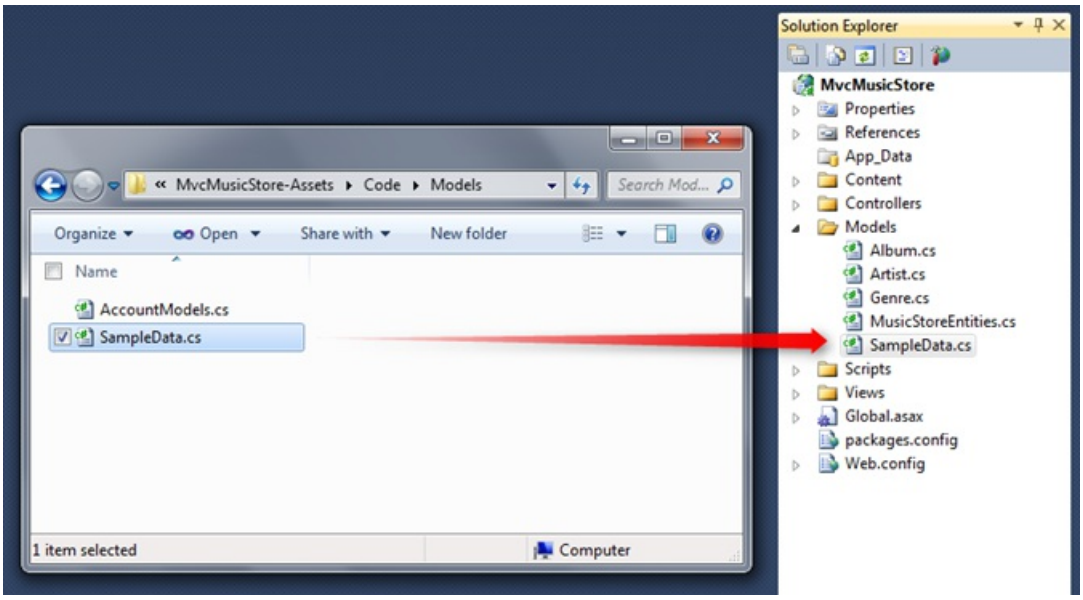
That's it - there's no other configuration, special interfaces, etc. By extending the DbContext base class, our MusicStoreEntities class is able to handle our database operations for us. Now that we've got that hooked up, let's

add a few more properties to our model classes to take advantage of some of the additional information in our database.

Adding our store catalog data

We will take advantage of a feature in Entity Framework which adds "seed" data to a newly created database. This will pre-populate our store catalog with a list of Genres, Artists, and Albums. The MvcMusicStore-Assets.zip download - which included our site design files used earlier in this tutorial - has a class file with this seed data, located in a folder named Code.

Within the Code / Models folder, locate the SampleData.cs file and drop it into the Models folder in our project, as shown below.



Now we need to add one line of code to tell Entity Framework about that SampleData class. Double-click on the Global.asax file in the root of the project to open it and add the following line to the top the Application_Start method.

```
protected void Application_Start()
{
    System.Data.Entity.Database.SetInitializer(
    new MvcMusicStore.Models.SampleData());
    AreaRegistration.RegisterAllAreas();
    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
 }
```

At this point, we've completed the work necessary to configure Entity Framework for our project.

Querying the Database

Now let's update our StoreController so that instead of using "dummy data" it instead calls into our database to query all of its information. We'll start by declaring a field on the **StoreController** to hold an instance of the MusicStoreEntities class, named storeDB:

```
public class StoreController : Controller
{
    MusicStoreEntities storeDB = new MusicStoreEntities();
```

Updating the Store Index to query the database

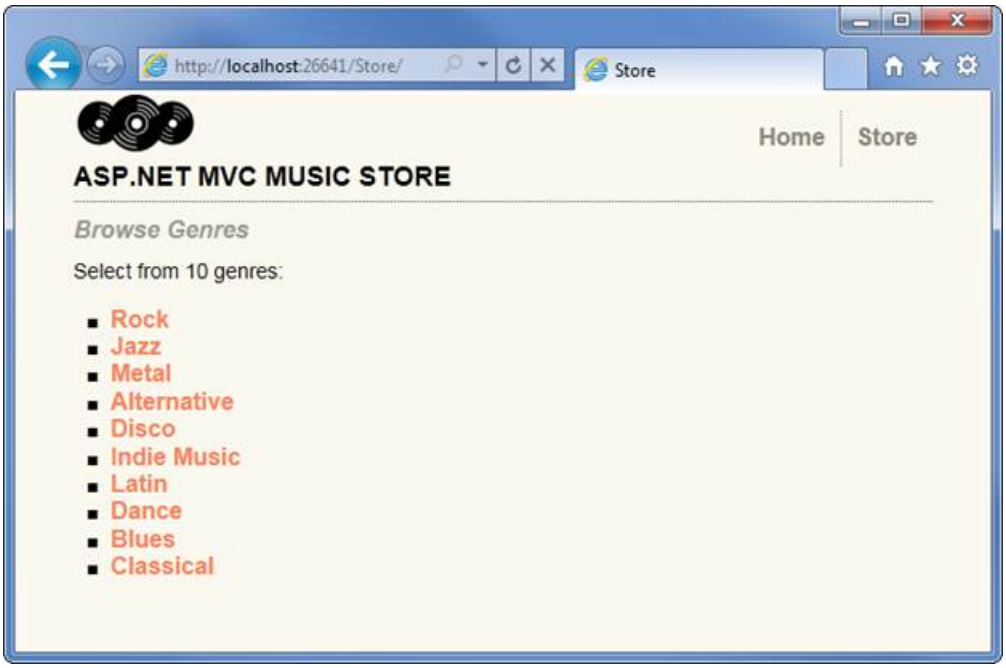
The MusicStoreEntities class is maintained by the Entity Framework and exposes a collection property for each

table in our database. Let's update our StoreController's Index action to retrieve all Genres in our database. Previously we did this by hard-coding string data. Now we can instead just use the Entity Framework context Generes collection:

```
public ActionResult Index()
{
    var genres = storeDB.Genres.ToList();
    return View(genres);
 }
```

No changes need to happen to our View template since we're still returning the same StoreIndexViewModel we returned before - we're just returning live data from our database now.

When we run the project again and visit the "/Store" URL, we'll now see a list of all Genres in our database:



Updating Store Browse and Details to use live data

With the /Store/Browse?genre=*[some-genre]* action method, we're searching for a Genre by name. We only expect one result, since we shouldn't ever have two entries for the same Genre name, and so we can use the .Single() extension in LINQ to query for the appropriate Genre object like this (don't type this yet):

```
var example = storeDB.Genres.Single(g => g.Name == "Disco");
```

The Single method takes a Lambda expression as a parameter, which specifies that we want a single Genre object such that its name matches the value we've defined. In the case above, we are loading a single Genre object with a Name value matching Disco.

We'll take advantage of an Entity Framework feature that allows us to indicate other related entities we want loaded as well when the Genre object is retrieved. This feature is called Query Result Shaping, and enables us to reduce the number of times we need to access the database to retrieve all of the information we need. We want to pre-fetch the Albums for Genre we retrieve, so we'll update our query to include from Genres.Include("Albums") to indicate that we want related albums as well. This is more efficient, since it will retrieve both our Genre and Album data in a single database request.

With the explanations out of the way, here's how our updated Browse controller action looks:

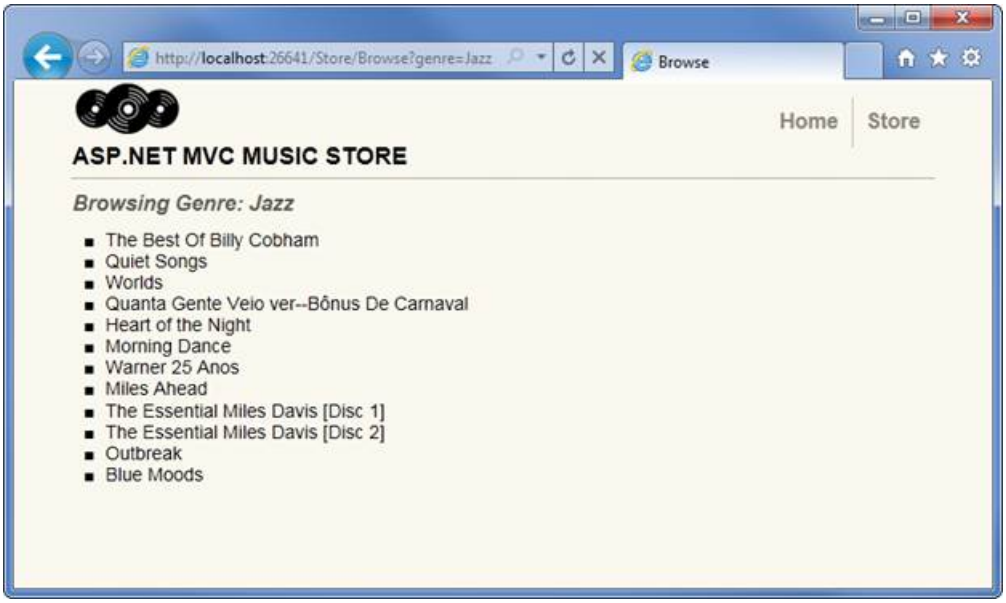
```
public ActionResult Browse(string genre)
{
    // Retrieve Genre and its Associated Albums from database
    var genreModel = storeDB.Genres.Include("Albums")
        .Single(g => g.Name == genre);

    return View(genreModel);
}
```

We can now update the Store Browse View to display the albums which are available in each Genre. Open the view template (found in /Views/Store/Browse.cshtml) and add a bulleted list of Albums as shown below.

```
@model MvcMusicStore.Models.Genre
@{
    ViewBag.Title = "Browse";
}
<h2>Browsing Genre: @Model.Name</h2>
<ul>
    @foreach (var album in Model.Albums)
    {
        <li>
            @album.Title
        </li>
    }
</ul>
```

Running our application and browsing to /Store/Browse?genre=Jazz shows that our results are now being pulled from the database, displaying all albums in our selected Genre.

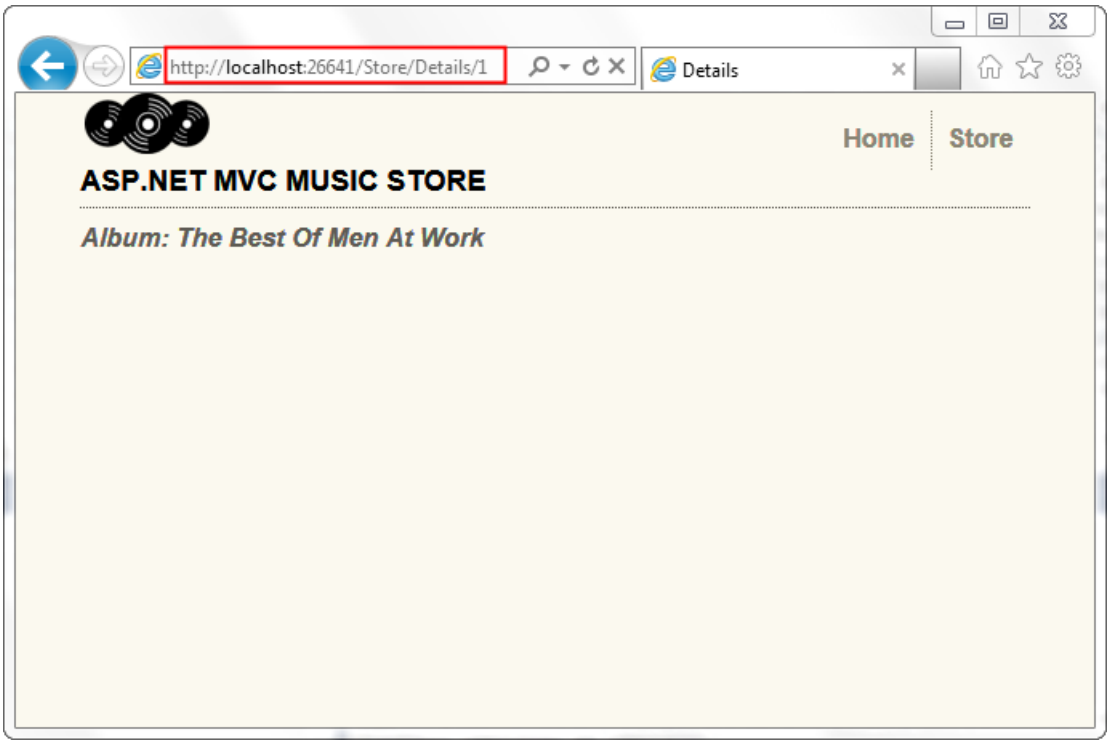


We'll make the same change to our /Store/Details/[id] URL, and replace our dummy data with a database query which loads an Album whose ID matches the parameter value.

```
public ActionResult Details(int id)
{
    var album = storeDB.Albums.Find(id);

    return View(album);
}
```

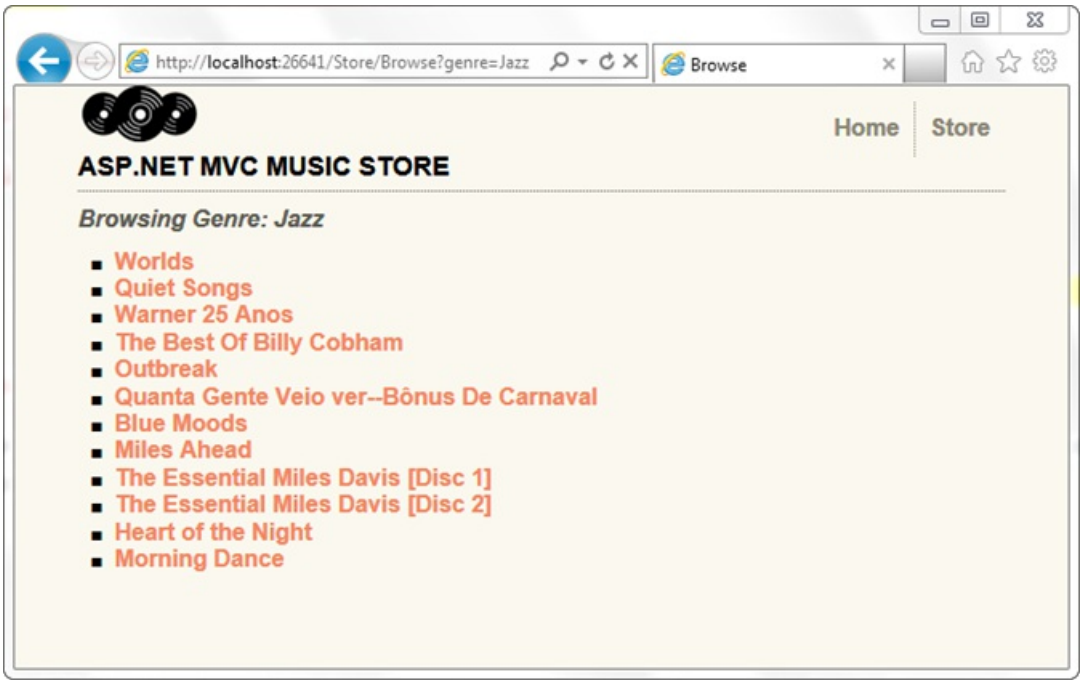
Running our application and browsing to /Store/Details/1 shows that our results are now being pulled from the database.



Now that our Store Details page is set up to display an album by the Album ID, let's update the **Browse** view to link to the Details view. We will use Html.ActionLink, exactly as we did to link from Store Index to Store Browse at the end of the previous section. The complete source for the Browse view appears below.

```
@model MvcMusicStore.Models.Genre
@{
    ViewBag.Title = "Browse";
}
<h2>Browsing Genre: @Model.Name</h2>
<ul>
    @foreach (var album in Model.Albums)
    {
        <li>
            @Html.ActionLink(album.Title,
"Details", new { id = album.AlbumId })
        </li>
    }
</ul>
```

We're now able to browse from our Store page to a Genre page, which lists the available albums, and by clicking on an album we can view details for that album.



PREVIOUS NEXT

Part 5: Edit Forms and Templating

2/20/2020 • 15 minutes to read • Edit Online

by Jon Galloway

The MVC Music Store is a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Studio for web development.

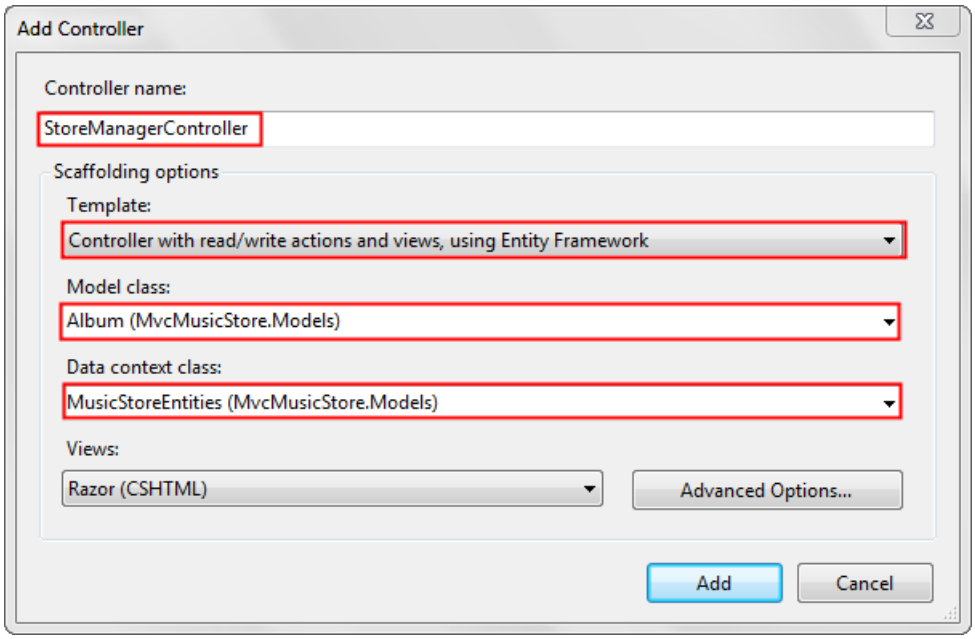
The MVC Music Store is a lightweight sample store implementation which sells music albums online, and implements basic site administration, user sign-in, and shopping cart functionality.

This tutorial series details all of the steps taken to build the ASP.NET MVC Music Store sample application. Part 5 covers Edit Forms and Templating.

In the past chapter, we were loading data from our database and displaying it. In this chapter, we'll also enable editing the data.

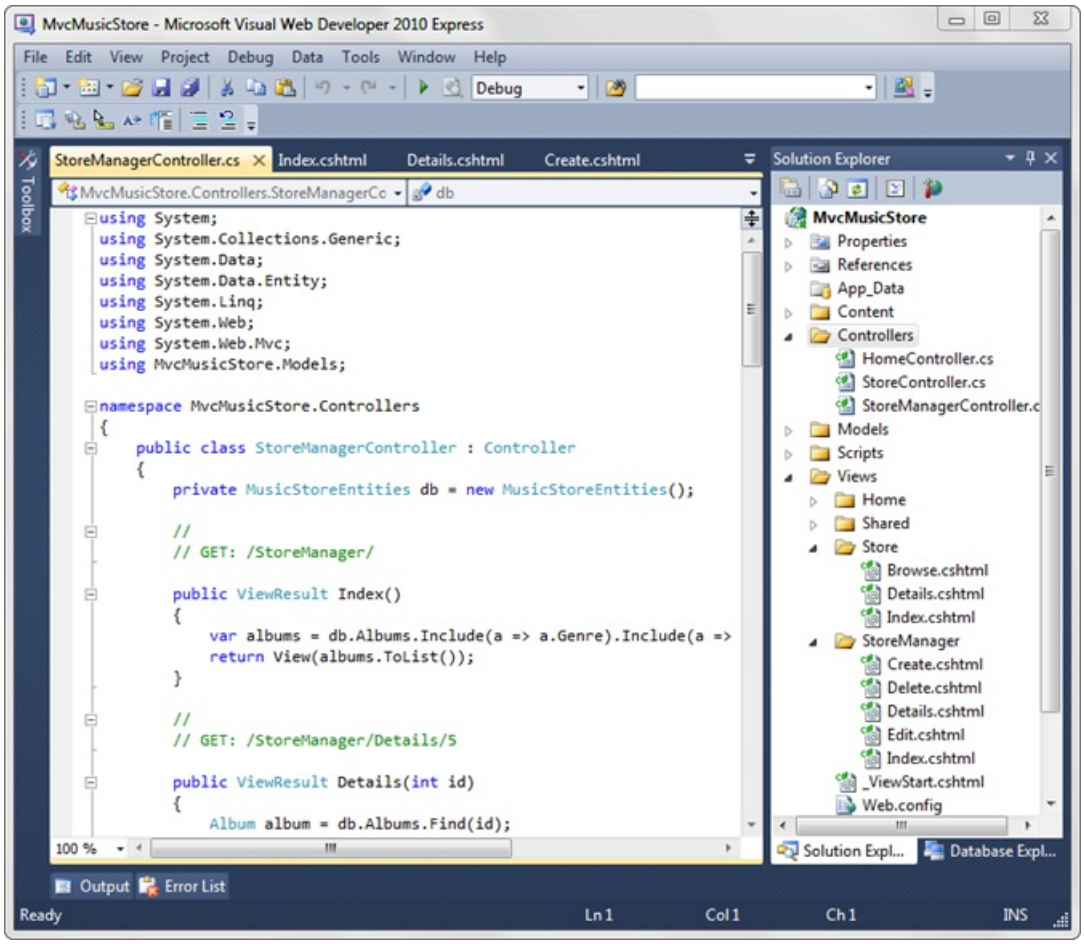
Creating the StoreManagerController

We'll begin by creating a new controller called **StoreManagerController**. For this controller, we will be taking advantage of the Scaffolding features available in the ASP.NET MVC 3 Tools Update. Set the options for the Add Controller dialog as shown below.



When you click the Add button, you'll see that the ASP.NET MVC 3 scaffolding mechanism does a good amount of work for you:

- It creates the new StoreManagerController with a local Entity Framework variable
- It adds a StoreManager folder to the project's Views folder
- It adds Create.cshtml, Delete.cshtml, Details.cshtml, Edit.cshtml, and Index.cshtml view, strongly typed to the Album class



The new StoreManager controller class includes CRUD (create, read, update, delete) controller actions which know how to work with the Album model class and use our Entity Framework context for database access.

Modifying a Scaffolded View

It's important to remember that, while this code was generated for us, it's standard ASP.NET MVC code, just like we've been writing throughout this tutorial. It's intended to save you the time you'd spend on writing boilerplate controller code and creating the strongly typed views manually, but this isn't the kind of generated code you may have seen prefaced with dire warnings in comments about how you mustn't change the code. This is your code, and you're expected to change it.

So, let's start with a quick edit to the StoreManager Index view (/Views/StoreManager/Index.cshtml). This view will display a table which lists the Albums in our store with Edit / Details / Delete links, and includes the Album's public properties. We'll remove the AlbumArtUrl field, as it's not very useful in this display. In <table> section of the view code, remove the <th> and <td> elements surrounding AlbumArtUrl references, as indicated by the highlighted lines below:

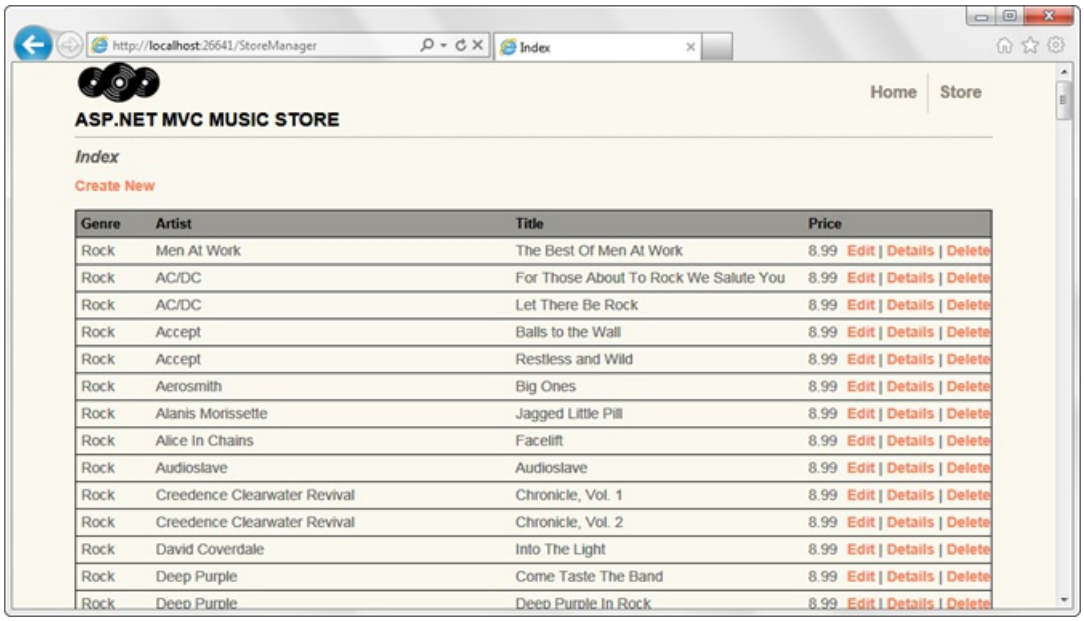
```
<table>
    <tr>
        <th>
            Genre
        </th>
        <th>
            Artist
        </th>
        <th>
            Title
        </th>
        <th>
            Price
        </th>
        <th>
            AlbumArtUrl
        </th>
        <th></th>
    </tr>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Genre.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Artist.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.AlbumArtUrl)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) |
            @Html.ActionLink("Details", "Details", new { id=item.AlbumId }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })
        </td>
    </tr>
}
</table>
```

The modified view code will appear as follows:

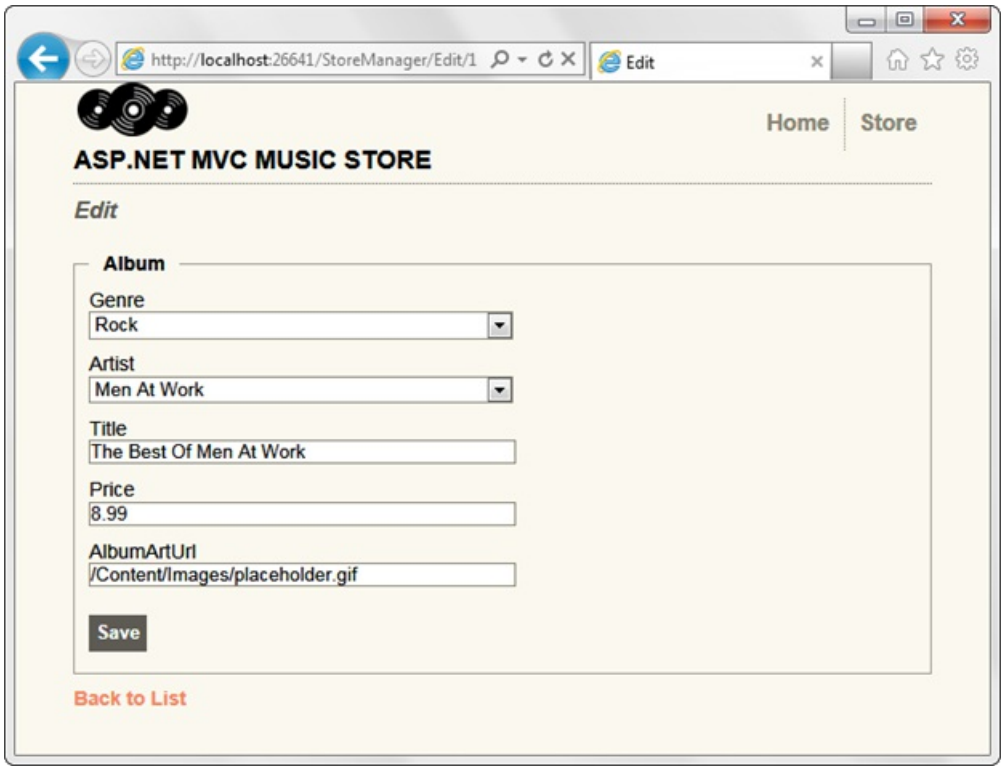
```
@model IEnumerable<MvcMusicStore.Models.Album>
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
<p>
    @Html.ActionLink("Create
New", "Create")
</p>
<table>
    <tr>
        <th>
            Genre
        </th>
        <th>
            Artist
        </th>
        <th>
            Title
        </th>
        <th>
            Price
        </th>
        <th></th>
    </tr>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Genre.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Artist.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) |
            @Html.ActionLink("Details", "Details", new { id=item.AlbumId }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })
        </td>
    </tr>
}
</table>
```

A first look at the Store Manager

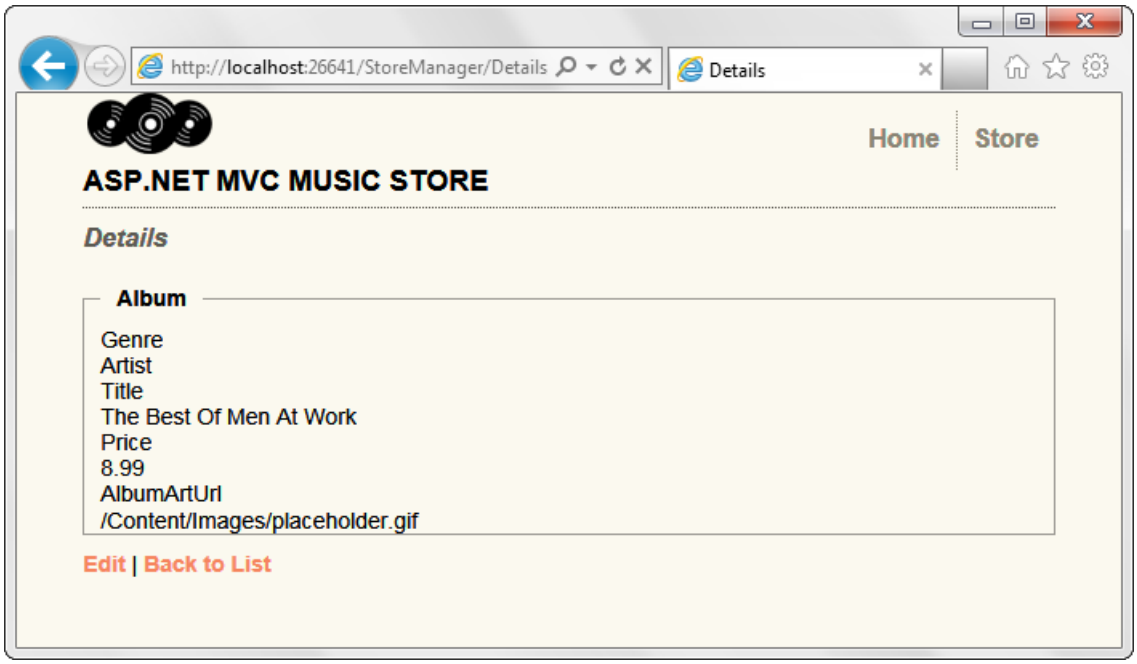
Now run the application and browse to /StoreManager/. This displays the Store Manager Index we just modified, showing a list of the albums in the store with links to Edit, Details, and Delete.



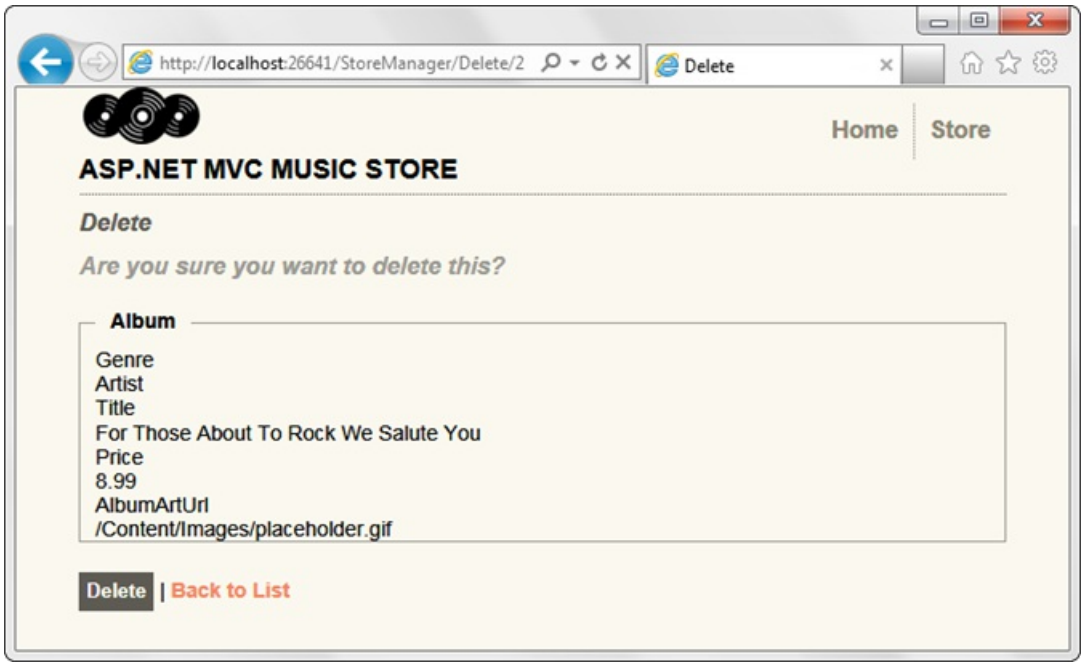
Clicking the Edit link displays an edit form with fields for the Album, including dropdowns for Genre and Artist.



Click the "Back to List" link at the bottom, then click on the Details link for an Album. This displays the detail information for an individual Album.



Again, click the Back to List link, then click on a Delete link. This displays a confirmation dialog, showing the album details and asking if we're sure we want to delete it.



Clicking the Delete button at the bottom will delete the album and return you to the Index page, which shows the album deleted.

We're not done with the Store Manager, but we have working controller and view code for the CRUD operations to start from.

Looking at the Store Manager Controller code

The Store Manager Controller contains a good amount of code. Let's go through this from top to bottom. The controller includes some standard namespaces for an MVC controller, as well as a reference to our Models namespace. The controller has a private instance of MusicStoreEntities, used by each of the controller actions for data access.

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MvcMusicStore.Models;

namespace MvcMusicStore.Controllers
{
    public class StoreManagerController : Controller
    {
        private MusicStoreEntities db = new MusicStoreEntities();
```

Store Manager Index and Details actions

The index view retrieves a list of Albums, including each album's referenced Genre and Artist information, as we previously saw when working on the Store Browse method. The Index view is following the references to the linked objects so that it can display each album's Genre name and Artist name, so the controller is being efficient and querying for this information in the original request.

```
//
// GET: /StoreManager/
public ViewResult Index()
{
    var albums = db.Albums.Include(a => a.Genre).Include(a => a.Artist);
    return View(albums.ToList());
}
```

The StoreManager Controller's Details controller action works exactly the same as the Store Controller Details action we wrote previously - it queries for the Album by ID using the Find() method, then returns it to the view.

```
//
// GET: /StoreManager/Details/5
public ViewResult Details(int id)
{
    Album album = db.Albums.Find(id);
    return View(album);
}
```

The Create Action Methods

The Create action methods are a little different from ones we've seen so far, because they handle form input. When a user first visits /StoreManager/Create/ they will be shown an empty form. This HTML page will contain a <form> element that contains dropdown and textbox input elements where they can enter the album's details.

After the user fills in the Album form values, they can press the "Save" button to submit these changes back to our application to save within the database. When the user presses the "save" button the <form> will perform an HTTP-POST back to the /StoreManager/Create/ URL and submit the <form> values as part of the HTTP-POST.

ASP.NET MVC allows us to easily split up the logic of these two URL invocation scenarios by enabling us to implement two separate "Create" action methods within our StoreManagerController class – one to handle the initial HTTP-GET browse to the /StoreManager/Create/ URL, and the other to handle the HTTP-POST of the submitted changes.

Passing information to a View using ViewBag

We've used the ViewBag earlier in this tutorial, but haven't talked much about it. The ViewBag allows us to pass information to the view without using a strongly typed model object. In this case, our Edit HTTP-GET controller

action needs to pass both a list of Genres and Artists to the form to populate the dropdowns, and the simplest way to do that is to return them as ViewBag items.

The ViewBag is a dynamic object, meaning that you can type ViewBag.Foo or ViewBag.YourNameHere without writing code to define those properties. In this case, the controller code uses ViewBag.GenreId and ViewBag.ArtistId so that the dropdown values submitted with the form will be GenreId and ArtistId, which are the Album properties they will be setting.

These dropdown values are returned to the form using the SelectList object, which is built just for that purpose. This is done using code like this:

```
ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name");
```

As you can see from the action method code, three parameters are being used to create this object:

- The list of items the dropdown will be displaying. Note that this isn't just a string - we're passing a list of Genres.
- The next parameter being passed to the SelectList is the Selected Value. This how the SelectList knows how to pre-select an item in the list. This will be easier to understand when we look at the Edit form, which is pretty similar.
- The final parameter is the property to be displayed. In this case, this is indicating that the Genre.Name property is what will be shown to the user.

With that in mind, then, the HTTP-GET Create action is pretty simple - two SelectLists are added to the ViewBag, and no model object is passed to the form (since it hasn't been created yet).

```
//
// GET: /StoreManager/Create
public ActionResult Create()
{
    ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name");
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name");
    return View();
}
```

HTML Helpers to display the Drop Downs in the Create View

Since we've talked about how the drop down values are passed to the view, let's take a quick look at the view to see how those values are displayed. In the view code (/Views/StoreManager/Create.cshtml), you'll see the following call is made to display the Genre drop down.

```
@Html.DropDownList("GenreId",
String.Empty)
```

This is known as an HTML Helper - a utility method which performs a common view task. HTML Helpers are very useful in keeping our view code concise and readable. The Html.DropDownList helper is provided by ASP.NET MVC, but as we'll see later it's possible to create our own helpers for view code we'll reuse in our application.

The Html.DropDownList call just needs to be told two things - where to get the list to display, and what value (if any) should be pre-selected. The first parameter, GenreId, tells the DropDownList to look for a value named GenreId in either the model or ViewBag. The second parameter is used to indicate the value to show as initially selected in the drop down list. Since this form is a Create form, there's no value to be preselected and String.Empty is passed.

Handling the Posted Form values

As we discussed before, there are two action methods associated with each form. The first handles the HTTP-GET

request and displays the form. The second handles the HTTP-POST request, which contains the submitted form values. Notice that controller action has an [HttpPost] attribute, which tells ASP.NET MVC that it should only respond to HTTP-POST requests.

```
//
// POST: /StoreManager/Create
[HttpPost]
public ActionResult Create(Album album)
{
    if (ModelState.IsValid)
    {
        db.Albums.Add(album);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name", album.GenreId);
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name", album.ArtistId);
    return View(album);
}
```

This action has four responsibilities:

- 1. Read the form values
- 2. Check if the form values pass any validation rules
- 3. If the form submission is valid, save the data and display the updated list
- 4. If the form submission is not valid, redisplay the form with validation errors

Reading Form Values with Model Binding

The controller action is processing a form submission that includes values for GenreId and ArtistId (from the drop down list) and textbox values for Title, Price, and AlbumArtUrl. While it's possible to directly access form values, a better approach is to use the Model Binding capabilities built into ASP.NET MVC. When a controller action takes a model type as a parameter, ASP.NET MVC will attempt to populate an object of that type using form inputs (as well as route and querystring values). It does this by looking for values whose names match properties of the model object, e.g. when setting the new Album object's GenreId value, it looks for an input with the name GenreId. When you create views using the standard methods in ASP.NET MVC, the forms will always be rendered using property names as input field names, so this the field names will just match up.

Validating the Model

The model is validated with a simple call to ModelState.IsValid. We haven't added any validation rules to our Album class yet - we'll do that in a bit - so right now this check doesn't have much to do. What's important is that this ModelStat.IsValid check will adapt to the validation rules we put on our model, so future changes to validation rules won't require any updates to the controller action code.

Saving the submitted values

If the form submission passes validation, it's time to save the values to the database. With Entity Framework, that just requires adding the model to the Albums collection and calling SaveChanges.

```
db.Albums.Add(album);
db.SaveChanges();
```

Entity Framework generates the appropriate SQL commands to persist the value. After saving the data, we redirect back to the list of Albums so we can see our update. This is done by returning RedirectToAction with the name of the controller action we want displayed. In this case, that's the Index method.

Displaying invalid form submissions with Validation Errors

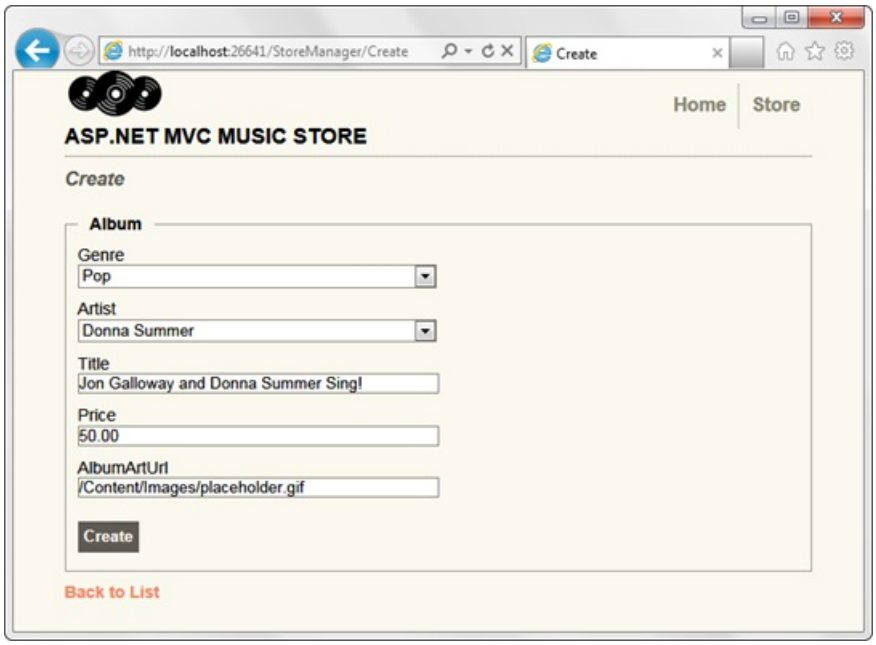
In the case of invalid form input, the dropdown values are added to the ViewBag (as in the HTTP-GET case) and the bound model values are passed back to the view for display. Validation errors are automatically displayed using

the @Html.ValidationMessageFor HTML Helper.

Testing the Create Form

To test this out, run the application and browse to /StoreManager/Create/ - this will show you the blank form which was returned by the StoreController Create HTTP-GET method.

Fill in some values and click the Create button to submit the form.



Jazz	Miles Davis	Miles Ahead	8.99	Edit | Details | Delete
Jazz	Miles Davis	The Essential Miles Davis [Disc 1]	8.99	Edit | Details | Delete
Jazz	Miles Davis	The Essential Miles Davis [Disc 2]	8.99	Edit | Details | Delete
Jazz	Spyro Gyra	Heart of the Night	8.99	Edit | Details | Delete
Jazz	Spyro Gyra	Morning Dance	8.99	Edit | Details | Delete
Pop	Donna Summer	Jon Galloway and Donna Summer Sing!	50.00	Edit | Details | Delete
Pop	Amy Winehouse	Frank	8.99	Edit | Details | Delete

Handling Edits

The Edit action pair (HTTP-GET and HTTP-POST) are very similar to the Create action methods we just looked at. Since the edit scenario involves working with an existing album, the Edit HTTP-GET method loads the Album based on the "id" parameter, passed in via the route. This code for retrieving an album by AlbumId is the same as we've previously looked at in the Details controller action. As with the Create / HTTP-GET method, the drop down values are returned via the ViewBag. This allows us to return an Album as our model object to the view (which is strongly typed to the Album class) while passing additional data (e.g. a list of Genres) via the ViewBag.

```
//
// GET: /StoreManager/Edit/5
public ActionResult Edit(int id)
{
    Album album = db.Albums.Find(id);
    ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name", album.GenreId);
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name", album.ArtistId);
    return View(album);
}
```

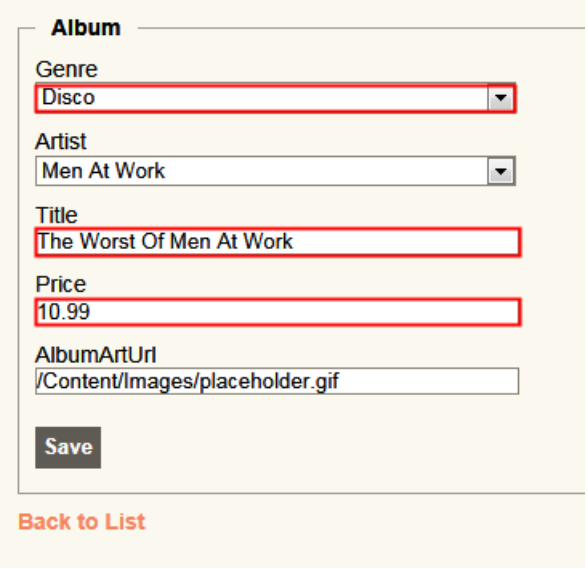
The Edit HTTP-POST action is very similar to the Create HTTP-POST action. The only difference is that instead of adding a new album to the db.Albums collection, we're finding the current instance of the Album using db.Entry(album) and setting its state to Modified. This tells Entity Framework that we are modifying an existing album as opposed to creating a new one.

```
//
// POST: /StoreManager/Edit/5
[HttpPost]
public ActionResult Edit(Album album)
{
    if (ModelState.IsValid)
    {
        db.Entry(album).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name", album.GenreId);
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name", album.ArtistId);
    return View(album);
}
```

We can test this out by running the application and browsing to /StoreManger/, then clicking the Edit link for an album.

Genre	Artist	Title	Price	
Rock	Men At Work	The Best Of Men At Work	8.99	Edit | Details | Delete
Rock	AC/DC	For Those About To Rock We Salute You	8.99	Edit | Details | Delete
Rock	AC/DC	Let There Be Rock	8.99	Edit | Details | Delete
Rock	Accept	Balls to the Wall	8.99	Edit | Details | Delete
Rock	Accept	Restless and Wild	8.99	Edit | Details | Delete
Rock	Aerosmith	Big Ones	8.99	Edit | Details | Delete

This displays the Edit form shown by the Edit HTTP-GET method. Fill in some values and click the Save button.



This posts the form, saves the values, and returns us to the Album list, showing that the values were updated.

Jazz	Spyro Gyra	Heart of the Night	8.99	Edit | Details | Delete
Jazz	Spyro Gyra	Morning Dance	8.99	Edit | Details | Delete
Pop	Amy Winehouse	Frank	8.99	Edit | Details | Delete
Pop	Various Artists	Axé Bahia 2001	8.99	Edit | Details | Delete
Disco	Men At Work	The Worst Of Men At Work	10.99	Edit | Details | Delete
Disco	Anita Ward	Ring My Bell	8.99	Edit | Details | Delete

Handling Deletion

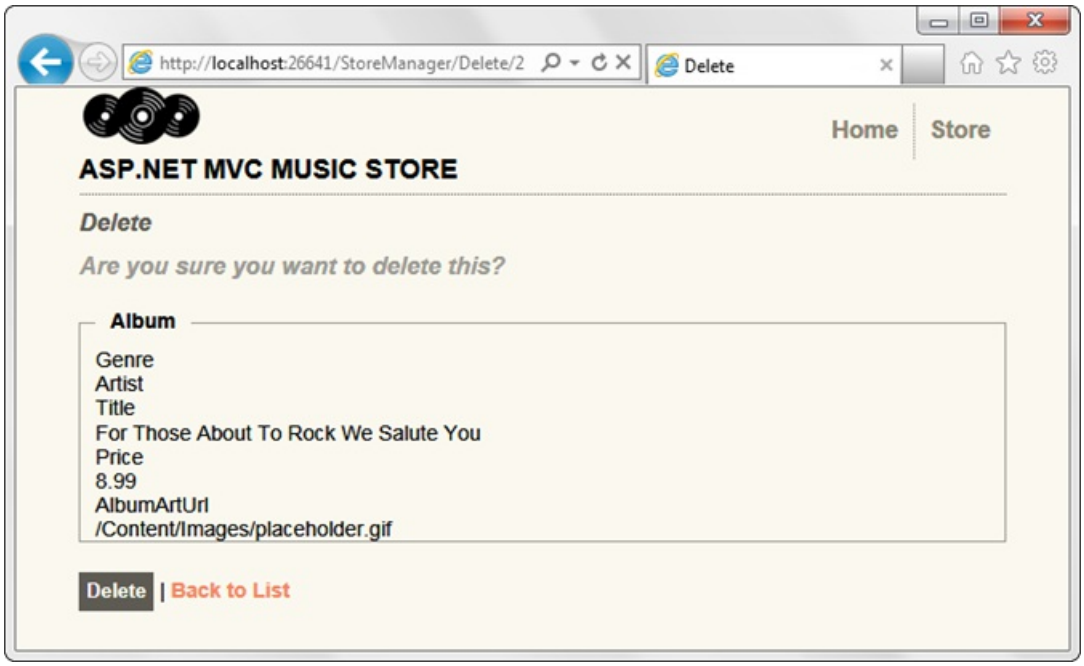
Deletion follows the same pattern as Edit and Create, using one controller action to display the confirmation form, and another controller action to handle the form submission.

The HTTP-GET Delete controller action is exactly the same as our previous Store Manager Details controller action.

```
//
// GET: /StoreManager/Delete/5

public ActionResult Delete(int id)
{
    Album album = db.Albums.Find(id);
    return View(album);
}
```

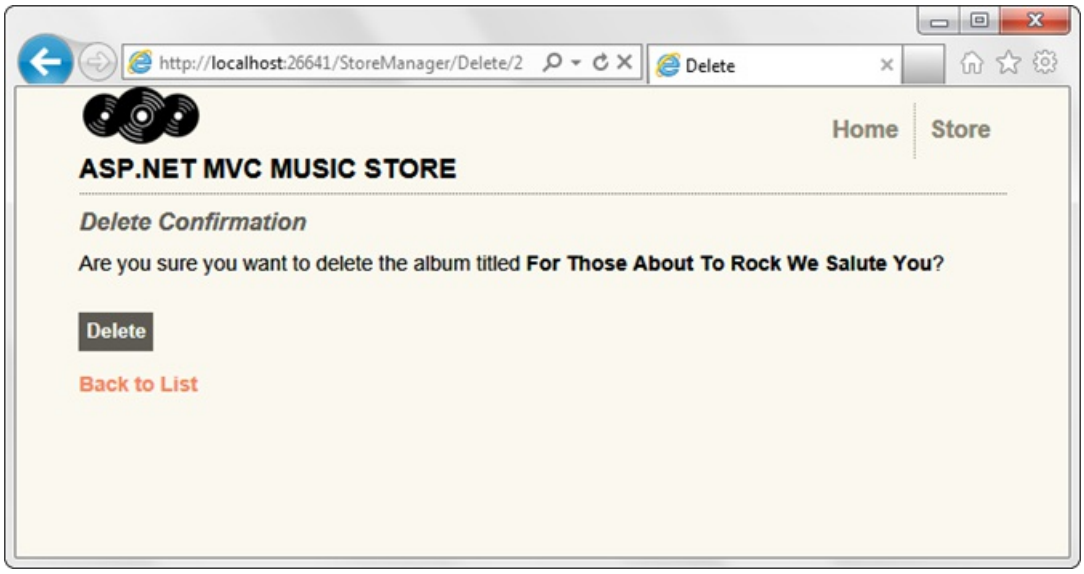
We display a form that's strongly typed to an Album type, using the Delete view content template.



The Delete template shows all the fields for the model, but we can simplify that down quite a bit. Change the view code in /Views/StoreManager/Delete.cshtml to the following.

```
@model MvcMusicStore.Models.Album
@{
    ViewBag.Title = "Delete";
}
<h2>Delete Confirmation</h2>
<p>Are you sure you want to delete the album titled
   <strong>@Model.Title</strong>?
</p>
@using (Html.BeginForm()) {
    <p>
        <input type="submit" value="Delete" />
    </p>
    <p>
        @Html.ActionLink("Back to
List", "Index")
    </p>
}
```

This displays a simplified Delete confirmation.



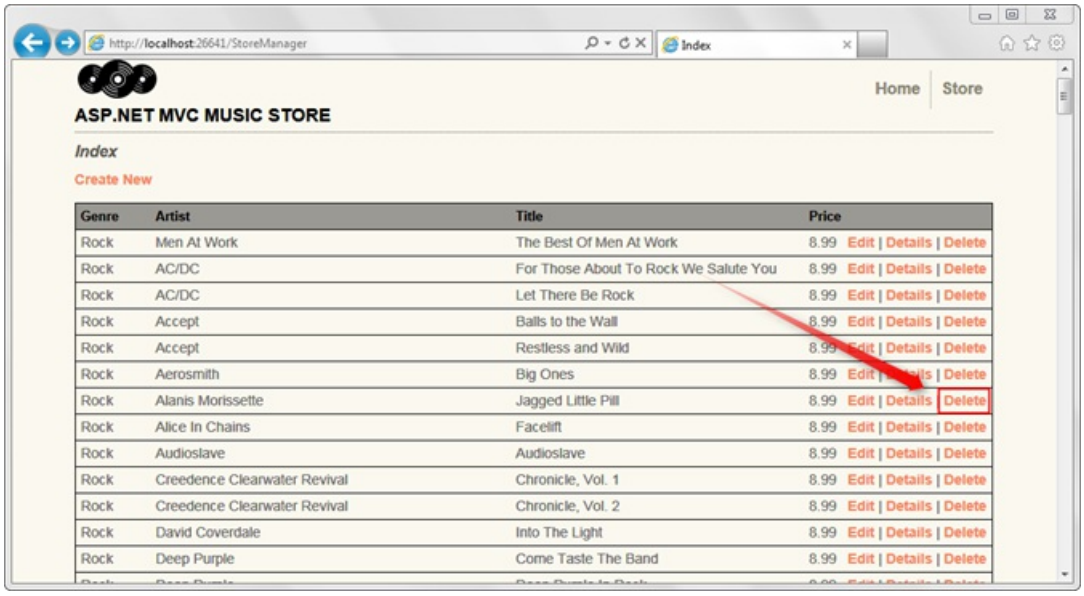
Clicking the Delete button causes the form to be posted back to the server, which executes the DeleteConfirmed action.

```
//
// POST: /StoreManager/Delete/5
[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
{
    Album album = db.Albums.Find(id);
    db.Albums.Remove(album);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

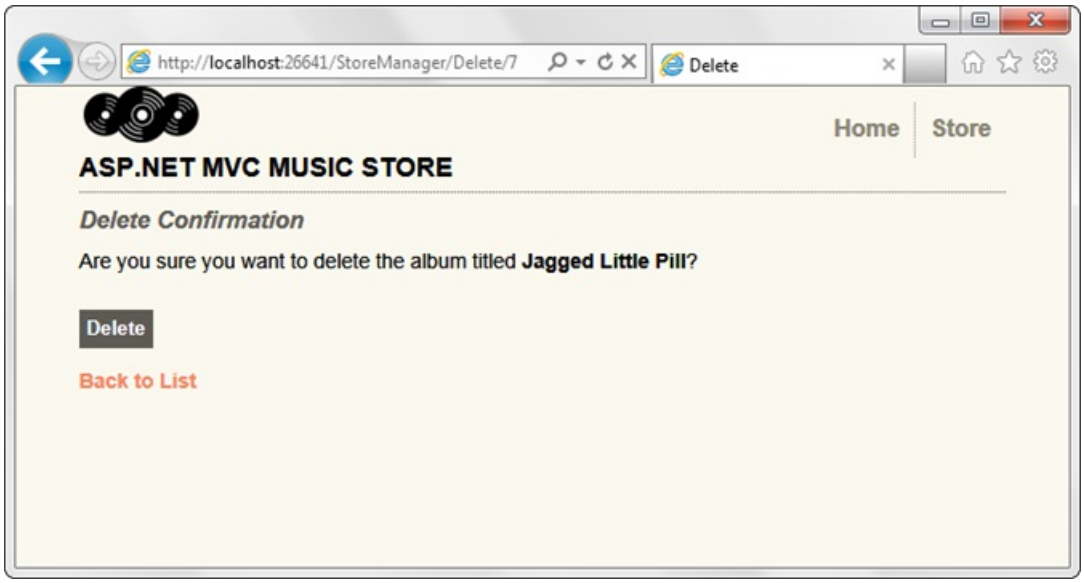
Our HTTP-POST Delete Controller Action takes the following actions:

- 1. Loads the Album by ID
- 2. Deletes it the album and save changes
- 3. Redirects to the Index, showing that the Album was removed from the list

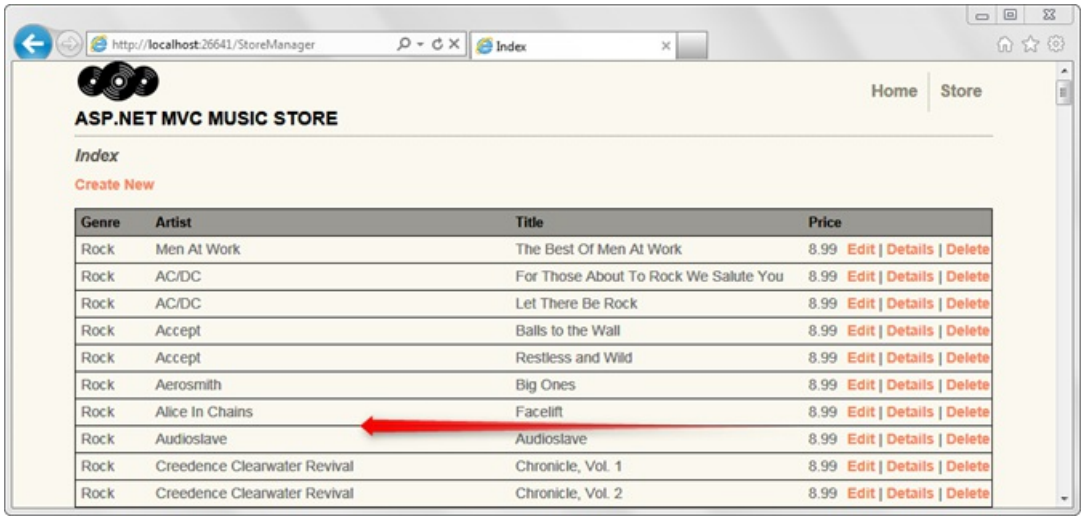
To test this, run the application and browse to /StoreManager. Select an album from the list and click the Delete link.



This displays our Delete confirmation screen.

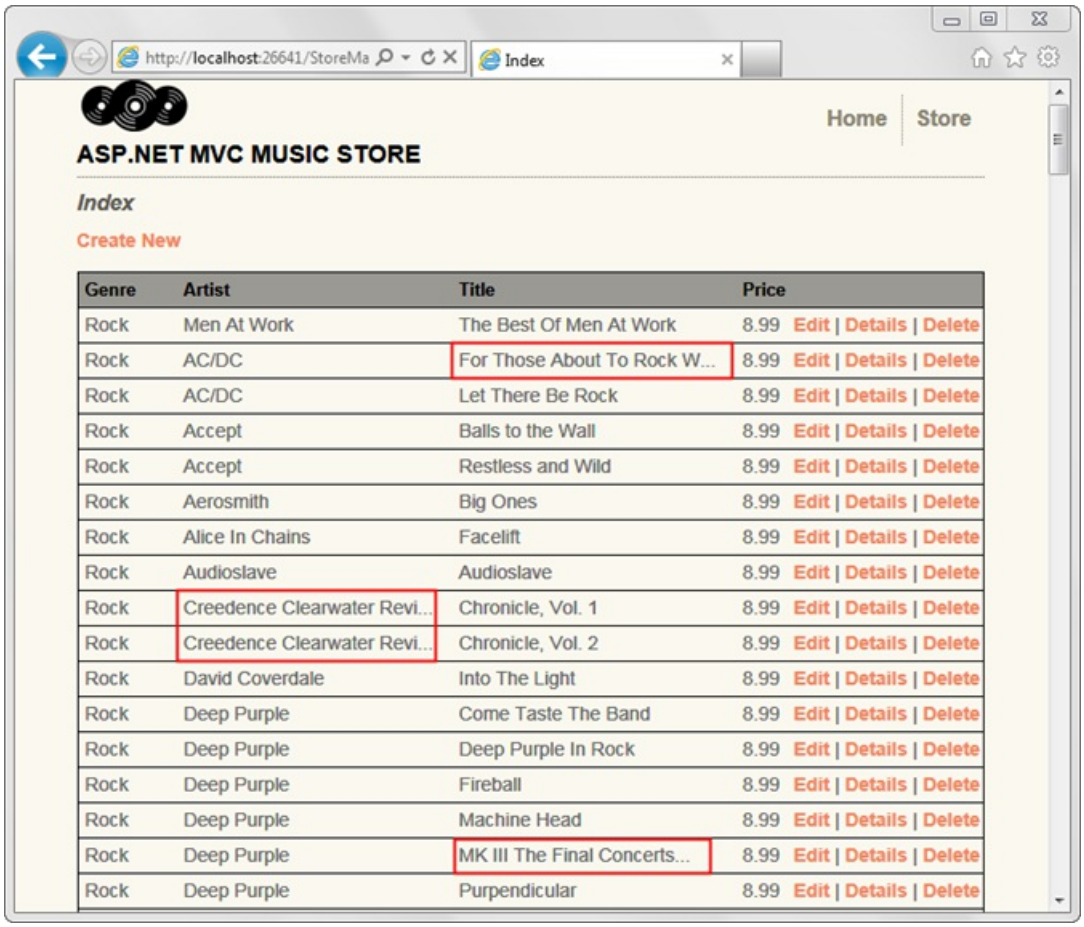


Clicking the Delete button removes the album and returns us to the Store Manager Index page, which shows that the album has been deleted.



Using a custom HTML Helper to truncate text

We've got one potential issue with our Store Manager Index page. Our Album Title and Artist Name properties can both be long enough that they could throw off our table formatting. We'll create a custom HTML Helper to allow us to easily truncate these and other properties in our Views.



Razor's @helper syntax has made it pretty easy to create your own helper functions for use in your views. Open the /Views/StoreManager/Index.cshtml view and add the following code directly after the @model line.

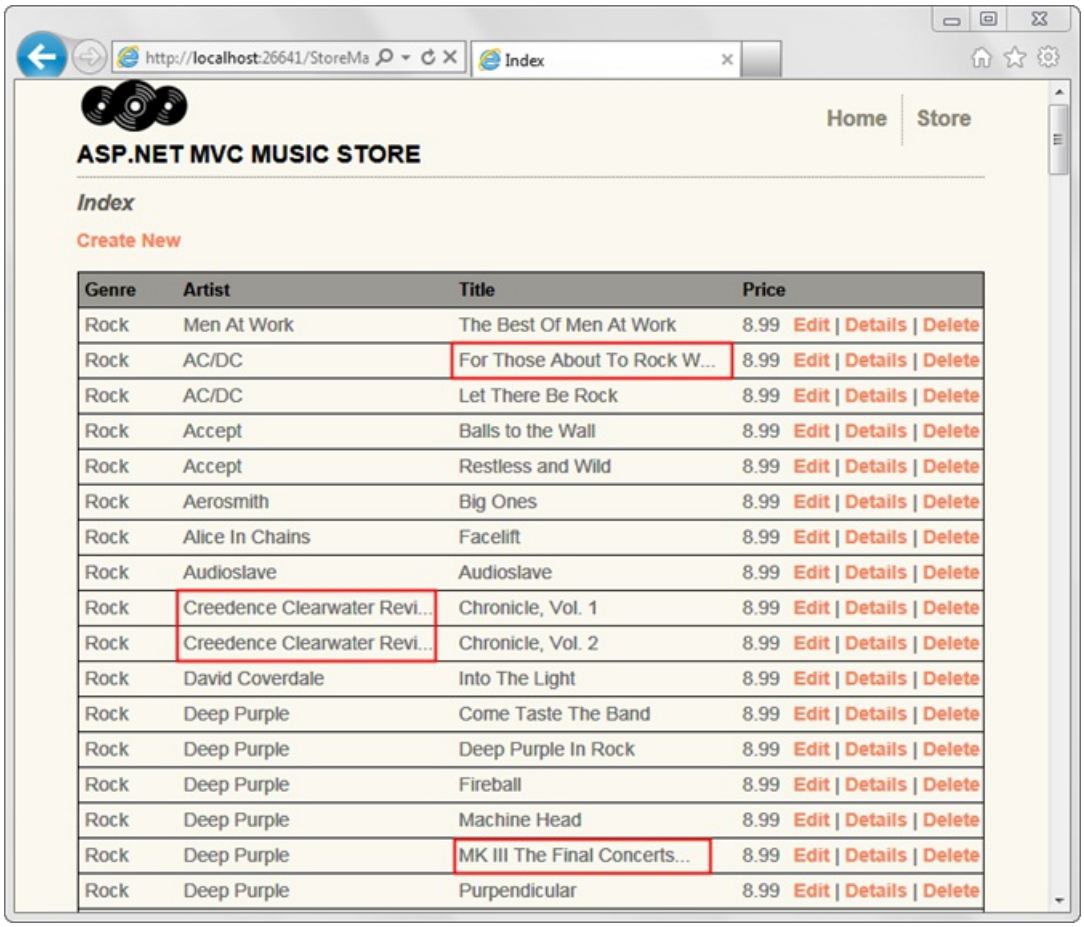
```
@helper Truncate(string
input, int length)
 {
    if (input.Length <= length) {
        @input
    } else {
        @input.Substring(0, length)<text>...</text>
    }
}
```

This helper method takes a string and a maximum length to allow. If the text supplied is shorter than the length specified, the helper outputs it as-is. If it is longer, then it truncates the text and renders "..." for the remainder.

Now we can use our Truncate helper to ensure that both the Album Title and Artist Name properties are less than 25 characters. The complete view code using our new Truncate helper appears below.

```
@model IEnumerable<MvcMusicStore.Models.Album>
@helper Truncate(string input, int length)
 {
    if (input.Length <= length) {
        @input
    } else {
        @input.Substring(0, length)<text>...</text>
    }
}
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
<p>
    @Html.ActionLink("Create
New", "Create")
</p>
<table>
    <tr>
        <th>
            Genre
        </th>
        <th>
            Artist
        </th>
        <th>
            Title
        </th>
        <th>
            Price
        </th>
        <th></th>
    </tr>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Genre.Name)
        </td>
        <td>
            @Truncate(item.Artist.Name, 25)
        </td>
        <td>
            @Truncate(item.Title, 25)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) |
            @Html.ActionLink("Details", "Details", new { id=item.AlbumId }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })
        </td>
    </tr>
}
</table>
```

Now when we browse the /StoreManager/ URL, the albums and titles are kept below our maximum lengths.



Note: This shows the simple case of creating and using a helper in one view. To learn more about creating helpers that you can use throughout your site, see my blog post: http://bit.ly/mvc3-helper-options

PREVIOUS NEXT

Part 6: Using Data Annotations for Model Validation

9/16/2020 • 3 minutes to read • Edit Online

by Jon Galloway

> The MVC Music Store is a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Studio for web development.
>
> The MVC Music Store is a lightweight sample store implementation which sells music albums online, and implements basic site administration, user sign-in, and shopping cart functionality.
>
> This tutorial series details all of the steps taken to build the ASP.NET MVC Music Store sample application. Part 6 covers Using Data Annotations for Model Validation.

We have a major issue with our Create and Edit forms: they're not doing any validation. We can do things like leave required fields blank or type letters in the Price field, and the first error we'll see is from the database.

We can easily add validation to our application by adding Data Annotations to our model classes. Data Annotations allow us to describe the rules we want applied to our model properties, and ASP.NET MVC will take care of enforcing them and displaying appropriate messages to our users.

Adding Validation to our Album Forms

We'll use the following Data Annotation attributes:

- **Required** – Indicates that the property is a required field
- **DisplayName** – Defines the text to use on form fields and validation messages
- **StringLength** – Defines a maximum length for a string field
- **Range** – Gives a maximum and minimum value for a numeric field
- **Bind** – Lists fields to exclude or include when binding parameter or form values to model properties
- **ScaffoldColumn** – Allows hiding fields from editor forms

Note: For more information on Model Validation using Data Annotation attributes, see the MSDN documentation at `https://go.microsoft.com/fwlink/?LinkId=159063`

Open the Album class and add the following *using* statements to the top.

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;
```

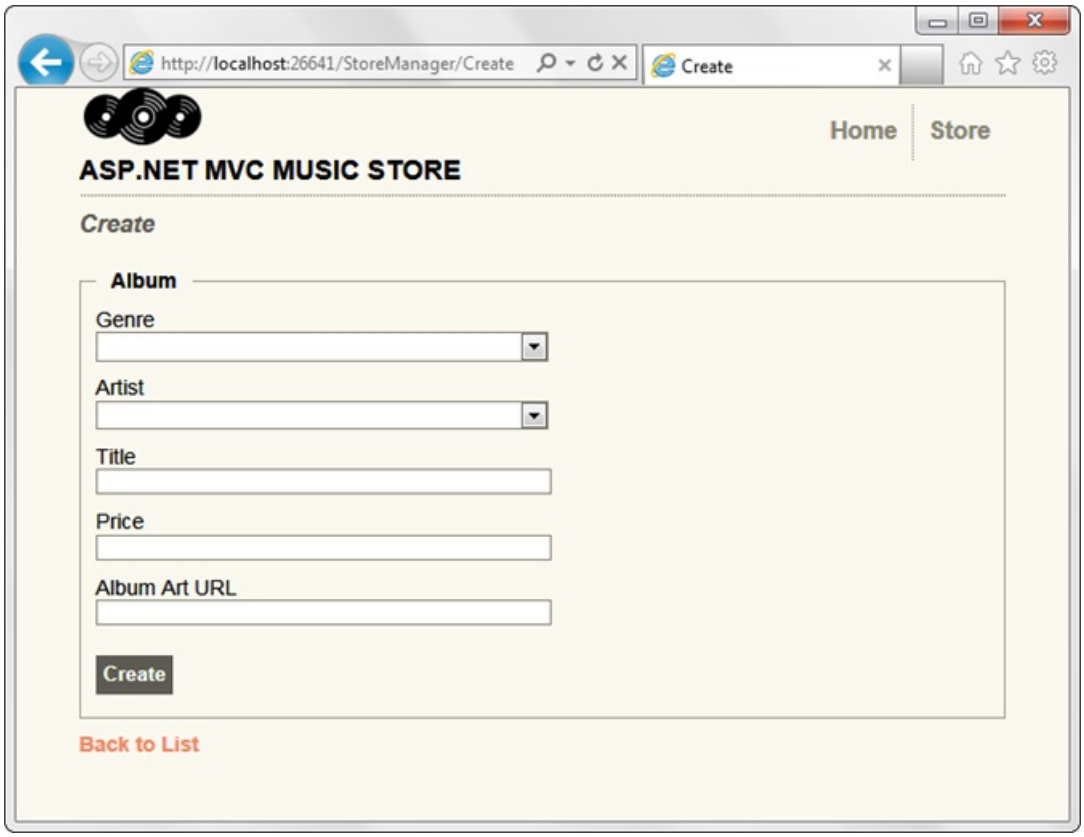
Next, update the properties to add display and validation attributes as shown below.

```
namespace MvcMusicStore.Models
{
    [Bind(Exclude = "AlbumId")]
    public class Album
    {
        [ScaffoldColumn(false)]
        public int      AlbumId    { get; set; }
        [DisplayName("Genre")]
        public int      GenreId    { get; set; }
        [DisplayName("Artist")]
        public int      ArtistId   { get; set; }
        [Required(ErrorMessage = "An Album Title is required")]
        [StringLength(160)]
        public string   Title      { get; set; }
        [Range(0.01, 100.00,
            ErrorMessage = "Price must be between 0.01 and 100.00")]
        public decimal Price       { get; set; }
        [DisplayName("Album Art URL")]
        [StringLength(1024)]
        public string AlbumArtUrl { get; set; }
        public virtual Genre  Genre    { get; set; }
        public virtual Artist Artist   { get; set; }
    }
}
```

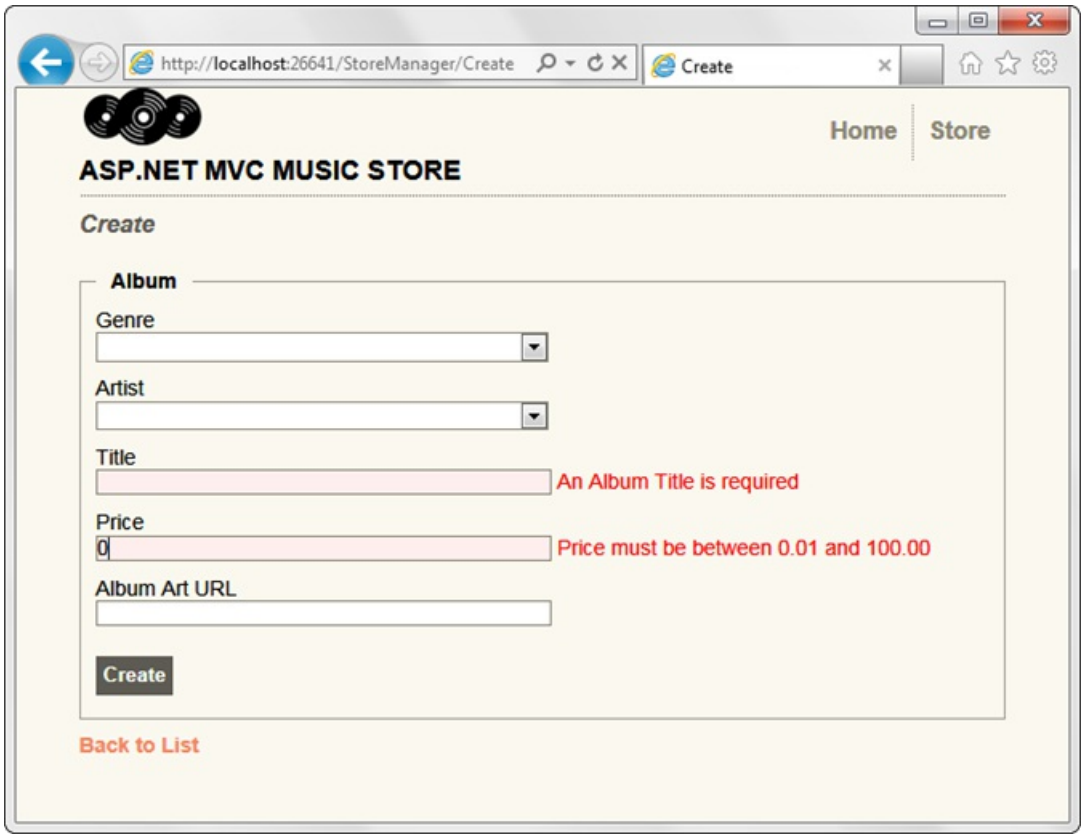
While we're there, we've also changed the Genre and Artist to virtual properties. This allows Entity Framework to lazy-load them as necessary.

```
public virtual Genre    Genre       { get; set; }
public virtual Artist   Artist      { get; set; }
```

After having added these attributes to our Album model, our Create and Edit screen immediately begin validating fields and using the Display Names we've chosen (e.g. Album Art Url instead of AlbumArtUrl). Run the application and browse to /StoreManager/Create.



Next, we'll break some validation rules. Enter a price of 0 and leave the Title blank. When we click on the Create button, we will see the form displayed with validation error messages showing which fields did not meet the validation rules we have defined.



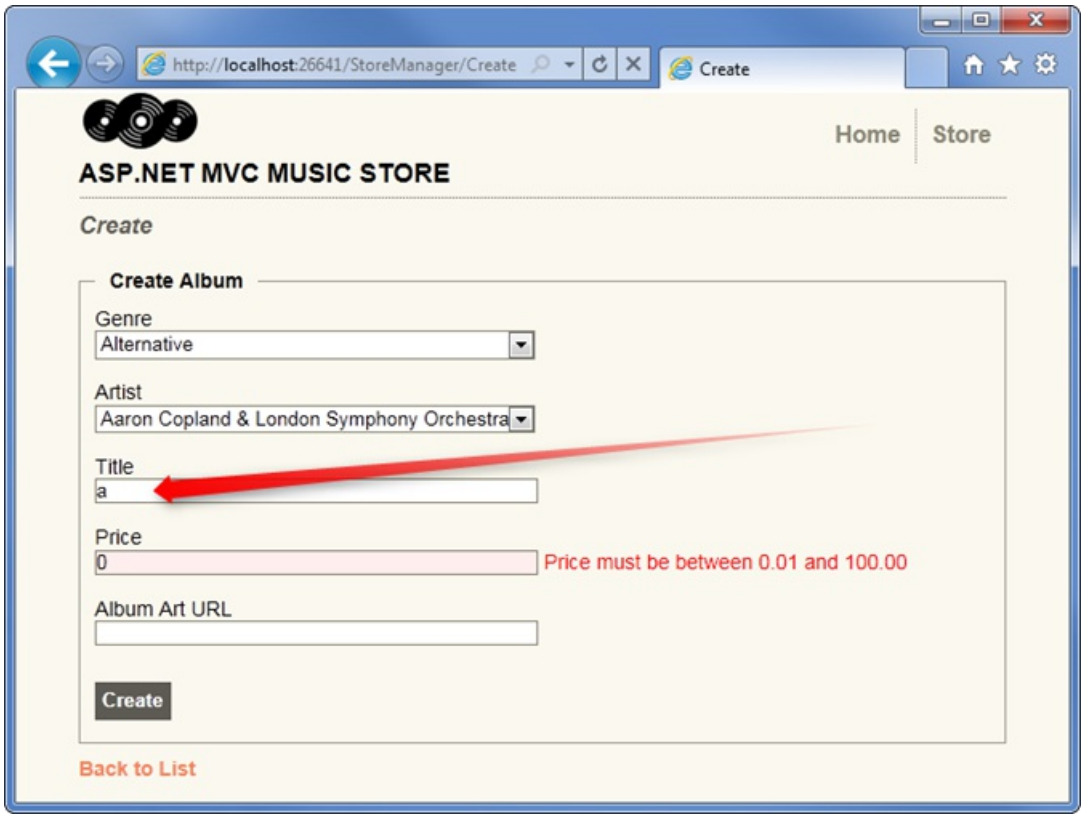
Testing the Client-Side Validation

Server-side validation is very important from an application perspective, because users can circumvent client-side validation. However, webpage forms which only implement server-side validation exhibit three significant problems.

1. The user has to wait for the form to be posted, validated on the server, and for the response to be sent to their browser.
2. The user doesn't get immediate feedback when they correct a field so that it now passes the validation rules.
3. We are wasting server resources to perform validation logic instead of leveraging the user's browser.

Fortunately, the ASP.NET MVC 3 scaffold templates have client-side validation built in, requiring no additional work whatsoever.

Typing a single letter in the Title field satisfies the validation requirements, so the validation message is immediately removed.



PREVIOUS NEXT

Part 7: Membership and Authorization

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

> The MVC Music Store is a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Studio for web development.
>
> The MVC Music Store is a lightweight sample store implementation which sells music albums online, and implements basic site administration, user sign-in, and shopping cart functionality.
>
> This tutorial series details all of the steps taken to build the ASP.NET MVC Music Store sample application. Part 7 covers Membership and Authorization.

Our Store Manager controller is currently accessible to anyone visiting our site. Let's change this to restrict permission to site administrators.

Adding the AccountController and Views

One difference between the full ASP.NET MVC 3 Web Application template and the ASP.NET MVC 3 Empty Web Application template is that the empty template doesn't include an Account Controller. We'll add an Account Controller by copying a few files from a new ASP.NET MVC application created from the full ASP.NET MVC 3 Web Application template.

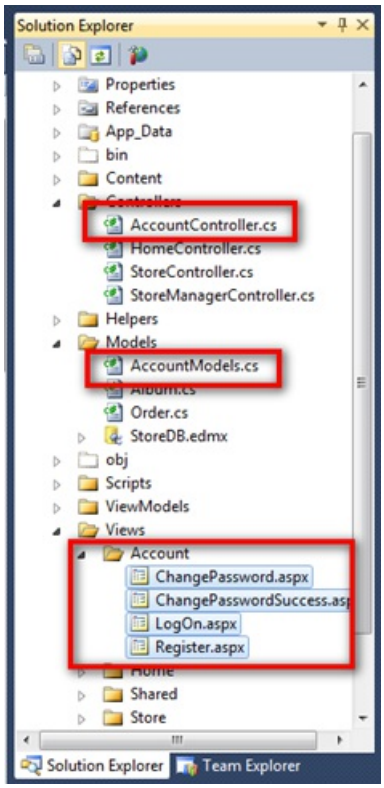
Create a new ASP.NET MVC application using the full ASP.NET MVC 3 Web Application template and copy the following files into the same directories in our project:

1. Copy AccountController.cs in the Controllers directory
2. Copy AccountModels in the Models directory
3. Create an Account directory inside the Views directory and copy all four views in

Change the namespace for the Controller and Model classes so they begin with MvcMusicStore. The AccountController class should use the MvcMusicStore.Controllers namespace, and the AccountModels class should use the MvcMusicStore.Models namespace.

Note: These files are also available in the MvcMusicStore-Assets.zip download from which we copied our site design files at the beginning of the tutorial. The Membership files are located in the Code directory.

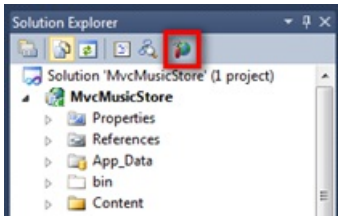
The updated solution should look like the following:



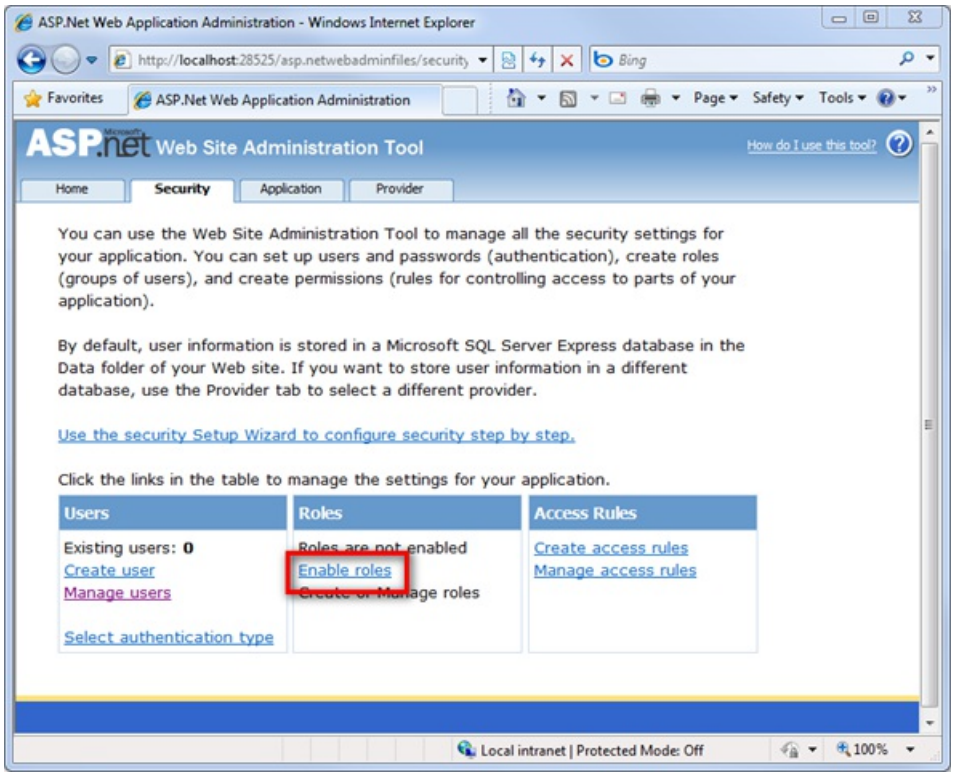
Adding an Administrative User with the ASP.NET Configuration site

Before we require Authorization in our website, we'll need to create a user with access. The easiest way to create a user is to use the built-in ASP.NET Configuration website.

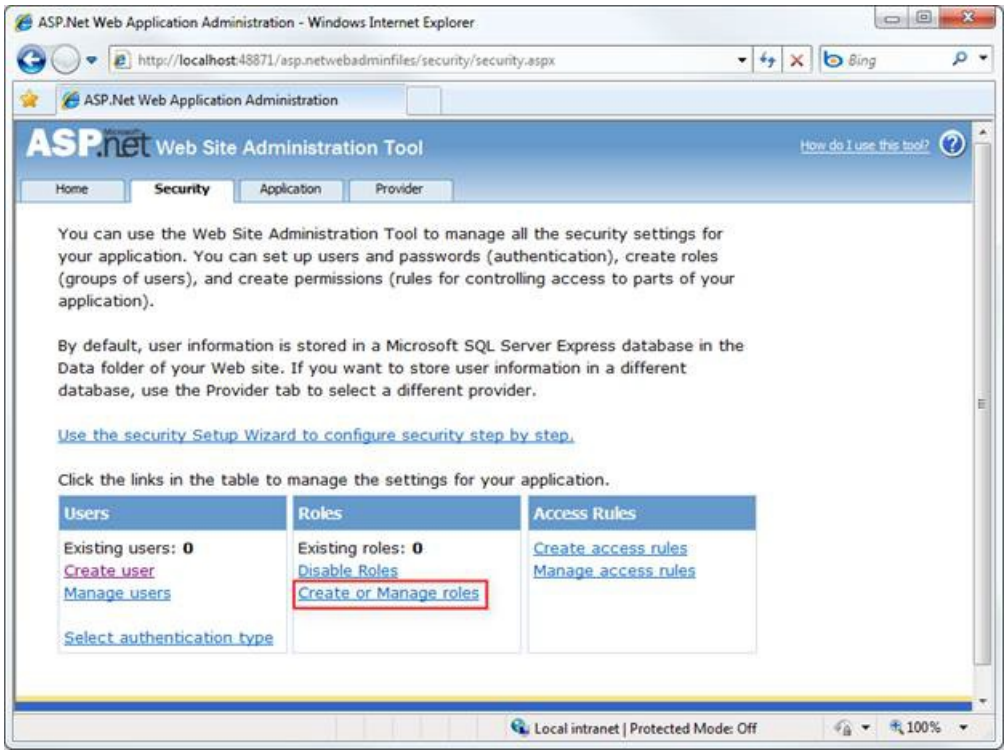
Launch the ASP.NET Configuration website by clicking following the icon in the Solution Explorer.



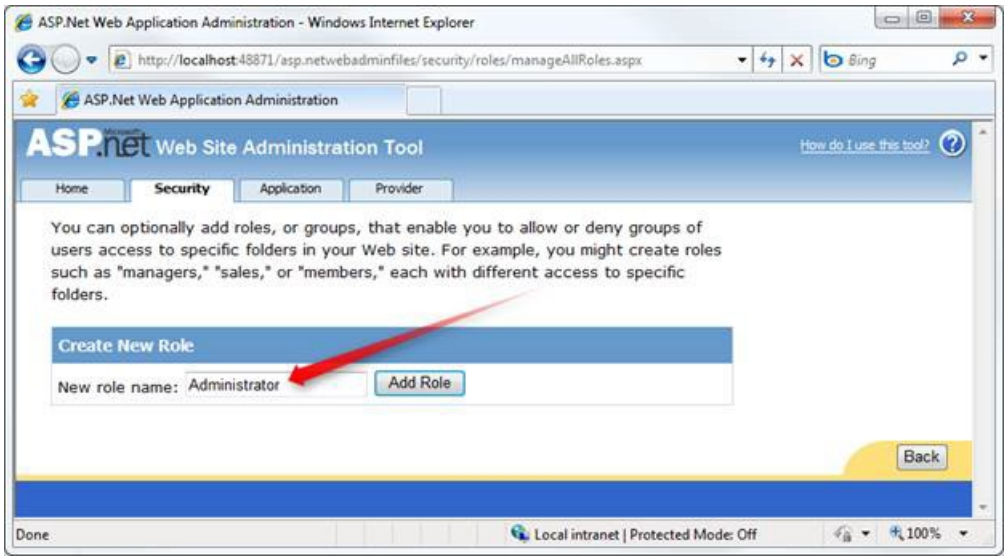
This launches a configuration website. Click on the Security tab on the home screen, then click the "Enable roles" link in the center of the screen.



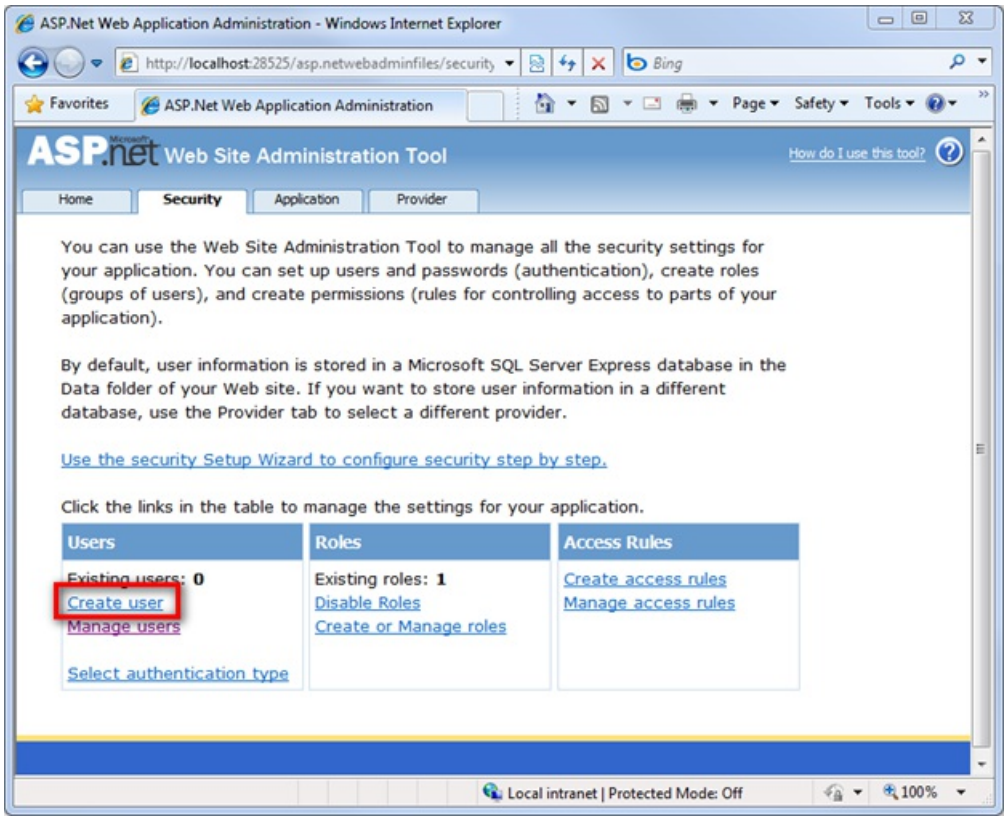
Click the "Create or Manage roles" link.



Enter "Administrator" as the role name and press the Add Role button.



Click the Back button, then click on the Create user link on the left side.

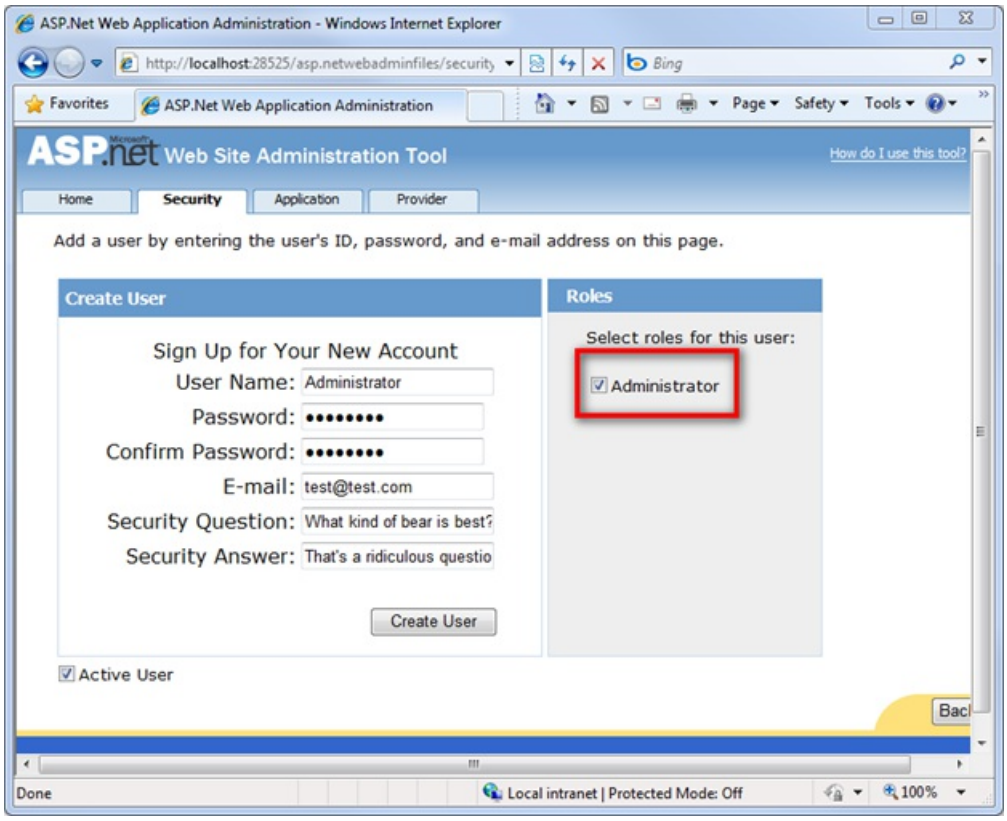


Fill in the user information fields on the left using the following information:

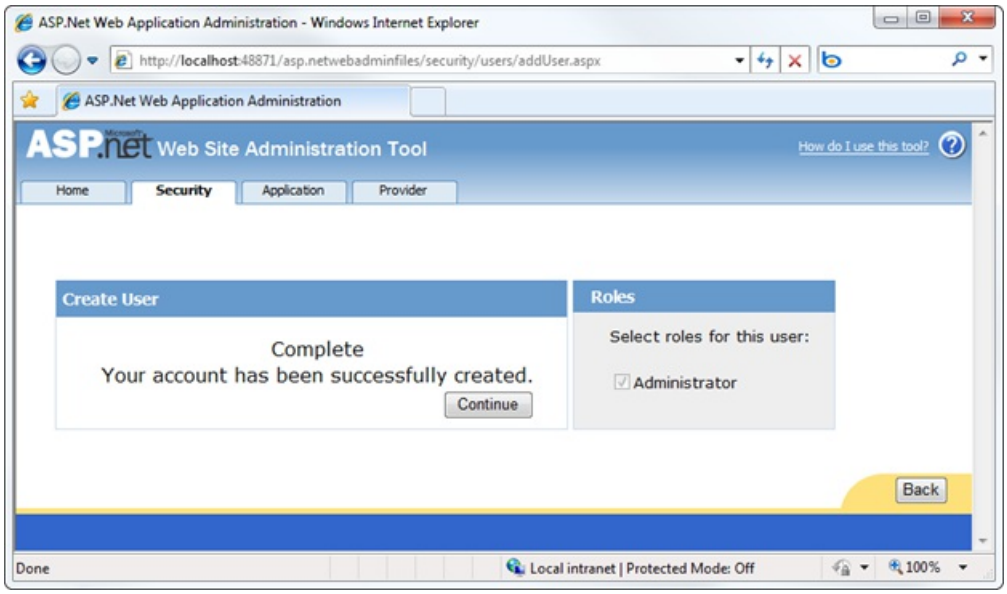
FIELD	VALUE
User Name	Administrator
Password	password123!
Confirm Password	password123!
E-mail	(any email address will work)
Security Question	(whatever you like)
Security Answer	(whatever you like)

Note: You can of course use any password you'd like. The default password security settings require a password that is 7 characters long and contains one non-alphanumeric character.

Select the Administrator role for this user, and click the Create User button.



At this point, you should see a message indicating that the user was created successfully.



You can now close the browser window.

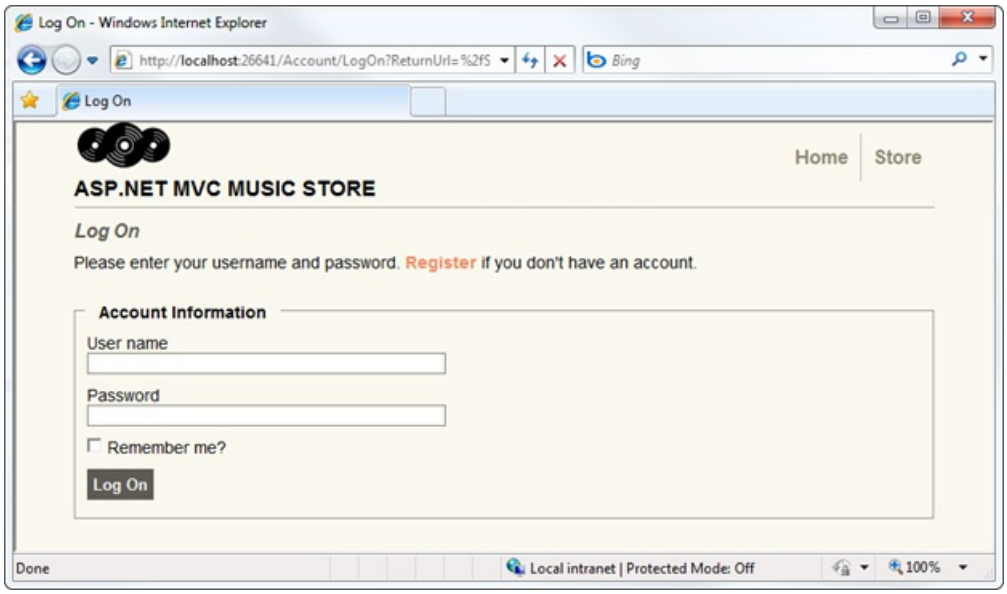
Role-based Authorization

Now we can restrict access to the StoreManagerController using the [Authorize] attribute, specifying that the user must be in the Administrator role to access any controller action in the class.

```
[Authorize(Roles = "Administrator")]
public class StoreManagerController : Controller
{
    // Controller code here
}
```

Note: The [Authorize] attribute can be placed on specific action methods as well as at the Controller class level.

Now browsing to /StoreManager brings up a Log On dialog:



After logging on with our new Administrator account, we're able to go to the Album Edit screen as before.

PREVIOUS NEXT

Part 8: Shopping Cart with Ajax Updates

2/20/2020 • 13 minutes to read • Edit Online

by Jon Galloway

> The MVC Music Store is a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Studio for web development.
>
> The MVC Music Store is a lightweight sample store implementation which sells music albums online, and implements basic site administration, user sign-in, and shopping cart functionality.
>
> This tutorial series details all of the steps taken to build the ASP.NET MVC Music Store sample application. Part 8 covers Shopping Cart with Ajax Updates.

We'll allow users to place albums in their cart without registering, but they'll need to register as guests to complete checkout. The shopping and checkout process will be separated into two controllers: a ShoppingCart Controller which allows anonymously adding items to a cart, and a Checkout Controller which handles the checkout process. We'll start with the Shopping Cart in this section, then build the Checkout process in the following section.

Adding the Cart, Order, and OrderDetail model classes

Our Shopping Cart and Checkout processes will make use of some new classes. Right-click the Models folder and add a Cart class (Cart.cs) with the following code.

```
using System.ComponentModel.DataAnnotations;

namespace MvcMusicStore.Models
{
    public class Cart
    {
        [Key]
        public int      RecordId    { get; set; }
        public string   CartId      { get; set; }
        public int      AlbumId     { get; set; }
        public int      Count       { get; set; }
        public System.DateTime DateCreated { get; set; }
        public virtual Album Album  { get; set; }
    }
}
```

This class is pretty similar to others we've used so far, with the exception of the [Key] attribute for the RecordId property. Our Cart items will have a string identifier named CartID to allow anonymous shopping, but the table includes an integer primary key named RecordId. By convention, Entity Framework Code-First expects that the primary key for a table named Cart will be either CartId or ID, but we can easily override that via annotations or code if we want. This is an example of how we can use the simple conventions in Entity Framework Code-First when they suit us, but we're not constrained by them when they don't.

Next, add an Order class (Order.cs) with the following code.

```
using System.Collections.Generic;

namespace MvcMusicStore.Models
{
    public partial class Order
    {
        public int    OrderId    { get; set; }
        public string Username   { get; set; }
        public string FirstName  { get; set; }
        public string LastName   { get; set; }
        public string Address    { get; set; }
        public string City       { get; set; }
        public string State      { get; set; }
        public string PostalCode { get; set; }
        public string Country    { get; set; }
        public string Phone      { get; set; }
        public string Email      { get; set; }
        public decimal Total     { get; set; }
        public System.DateTime OrderDate      { get; set; }
        public List<OrderDetail> OrderDetails { get; set; }
    }
}
```

This class tracks summary and delivery information for an order. **It won't compile yet**, because it has an OrderDetails navigation property which depends on a class we haven't created yet. Let's fix that now by adding a class named OrderDetail.cs, adding the following code.

```
namespace MvcMusicStore.Models
{
    public class OrderDetail
    {
        public int OrderDetailId { get; set; }
        public int OrderId { get; set; }
        public int AlbumId { get; set; }
        public int Quantity { get; set; }
        public decimal UnitPrice { get; set; }
        public virtual Album Album { get; set; }
        public virtual Order Order { get; set; }
    }
}
```

We'll make one last update to our MusicStoreEntities class to include DbSets which expose those new Model classes, also including a DbSet<Artist>. The updated MusicStoreEntities class appears as below.

```
using System.Data.Entity;

namespace MvcMusicStore.Models
{
    public class MusicStoreEntities : DbContext
    {
        public DbSet<Album>     Albums  { get; set; }
        public DbSet<Genre>     Genres  { get; set; }
        public DbSet<Artist>    Artists {
get; set; }
        public DbSet<Cart>
Carts { get; set; }
        public DbSet<Order>     Orders
{ get; set; }
        public DbSet<OrderDetail>
OrderDetails { get; set; }
    }
}
```

Managing the Shopping Cart business logic

Next, we'll create the ShoppingCart class in the Models folder. The ShoppingCart model handles data access to the Cart table. Additionally, it will handle the business logic to for adding and removing items from the shopping cart.

Since we don't want to require users to sign up for an account just to add items to their shopping cart, we will assign users a temporary unique identifier (using a GUID, or globally unique identifier) when they access the shopping cart. We'll store this ID using the ASP.NET Session class.

Note: The ASP.NET Session is a convenient place to store user-specific information which will expire after they leave the site. While misuse of session state can have performance implications on larger sites, our light use will work well for demonstration purposes.

The ShoppingCart class exposes the following methods:

AddToCart takes an Album as a parameter and adds it to the user's cart. Since the Cart table tracks quantity for each album, it includes logic to create a new row if needed or just increment the quantity if the user has already ordered one copy of the album.

RemoveFromCart takes an Album ID and removes it from the user's cart. If the user only had one copy of the album in their cart, the row is removed.

EmptyCart removes all items from a user's shopping cart.

GetCartItems retrieves a list of CartItems for display or processing.

GetCount retrieves a the total number of albums a user has in their shopping cart.

GetTotal calculates the total cost of all items in the cart.

CreateOrder converts the shopping cart to an order during the checkout phase.

GetCart is a static method which allows our controllers to obtain a cart object. It uses the **GetCartId** method to handle reading the CartId from the user's session. The GetCartId method requires the HttpContextBase so that it can read the user's CartId from user's session.

Here's the complete **ShoppingCart class**:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcMusicStore.Models
{
    public partial class ShoppingCart
    {
        MusicStoreEntities storeDB = new MusicStoreEntities();
        string ShoppingCartId { get; set; }
        public const string CartSessionKey = "CartId";
        public static ShoppingCart GetCart(HttpContextBase context)
        {
            var cart = new ShoppingCart();
            cart.ShoppingCartId = cart.GetCartId(context);
            return cart;
        }
        // Helper method to simplify shopping cart calls
        public static ShoppingCart GetCart(Controller controller)
        {
            return GetCart(controller.HttpContext);
        }
        public void AddToCart(Album album)
        {
```

```
        {
            // Get the matching cart and album instances
            var cartItem = storeDB.Carts.SingleOrDefault(
                c => c.CartId == ShoppingCartId
                && c.AlbumId == album.AlbumId);

            if (cartItem == null)
            {
                // Create a new cart item if no cart item exists
                cartItem = new Cart
                {
                    AlbumId = album.AlbumId,
                    CartId = ShoppingCartId,
                    Count = 1,
                    DateCreated = DateTime.Now
                };
                storeDB.Carts.Add(cartItem);
            }
            else
            {
                // If the item does exist in the cart,
                // then add one to the quantity
                cartItem.Count++;
            }
            // Save changes
            storeDB.SaveChanges();
        }
        public int RemoveFromCart(int id)
        {
            // Get the cart
            var cartItem = storeDB.Carts.Single(
                cart => cart.CartId == ShoppingCartId
                && cart.RecordId == id);

            int itemCount = 0;

            if (cartItem != null)
            {
                if (cartItem.Count > 1)
                {
                    cartItem.Count--;
                    itemCount = cartItem.Count;
                }
                else
                {
                    storeDB.Carts.Remove(cartItem);
                }
                // Save changes
                storeDB.SaveChanges();
            }
            return itemCount;
        }
        public void EmptyCart()
        {
            var cartItems = storeDB.Carts.Where(
                cart => cart.CartId == ShoppingCartId);

            foreach (var cartItem in cartItems)
            {
                storeDB.Carts.Remove(cartItem);
            }
            // Save changes
            storeDB.SaveChanges();
        }
        public List<Cart> GetCartItems()
        {
            return storeDB.Carts.Where(
                cart => cart.CartId == ShoppingCartId).ToList();
        }
```

```
        public int GetCount()
        {
            // Get the count of each item in the cart and sum them up
            int? count = (from cartItems in storeDB.Carts
                          where cartItems.CartId == ShoppingCartId
                          select (int?)cartItems.Count).Sum();
            // Return 0 if all entries are null
            return count ?? 0;
        }
        public decimal GetTotal()
        {
            // Multiply album price by count of that album to get
            // the current price for each of those albums in the cart
            // sum all album price totals to get the cart total
            decimal? total = (from cartItems in storeDB.Carts
                              where cartItems.CartId == ShoppingCartId
                              select (int?)cartItems.Count *
                              cartItems.Album.Price).Sum();

            return total ?? decimal.Zero;
        }
        public int CreateOrder(Order order)
        {
            decimal orderTotal = 0;

            var cartItems = GetCartItems();
            // Iterate over the items in the cart,
            // adding the order details for each
            foreach (var item in cartItems)
            {
                var orderDetail = new OrderDetail
                {
                    AlbumId = item.AlbumId,
                    OrderId = order.OrderId,
                    UnitPrice = item.Album.Price,
                    Quantity = item.Count
                };
                // Set the order total of the shopping cart
                orderTotal += (item.Count * item.Album.Price);

                storeDB.OrderDetails.Add(orderDetail);

            }
            // Set the order's total to the orderTotal count
            order.Total = orderTotal;

            // Save the order
            storeDB.SaveChanges();
            // Empty the shopping cart
            EmptyCart();
            // Return the OrderId as the confirmation number
            return order.OrderId;
        }
        // We're using HttpContextBase to allow access to cookies.
        public string GetCartId(HttpContextBase context)
        {
            if (context.Session[CartSessionKey] == null)
            {
                if (!string.IsNullOrWhiteSpace(context.User.Identity.Name))
                {
                    context.Session[CartSessionKey] =
                        context.User.Identity.Name;
                }
                else
                {
                    // Generate a new random GUID using System.Guid class
                    Guid tempCartId = Guid.NewGuid();
                    // Send tempCartId back to client as a cookie
                    context.Session[CartSessionKey] = tempCartId.ToString();
```

```
                }
            }
            return context.Session[CartSessionKey].ToString();
        }
        // When a user has logged in, migrate their shopping cart to
        // be associated with their username
        public void MigrateCart(string userName)
        {
            var shoppingCart = storeDB.Carts.Where(
                c => c.CartId == ShoppingCartId);

            foreach (Cart item in shoppingCart)
            {
                item.CartId = userName;
            }
            storeDB.SaveChanges();
        }
    }
}
```

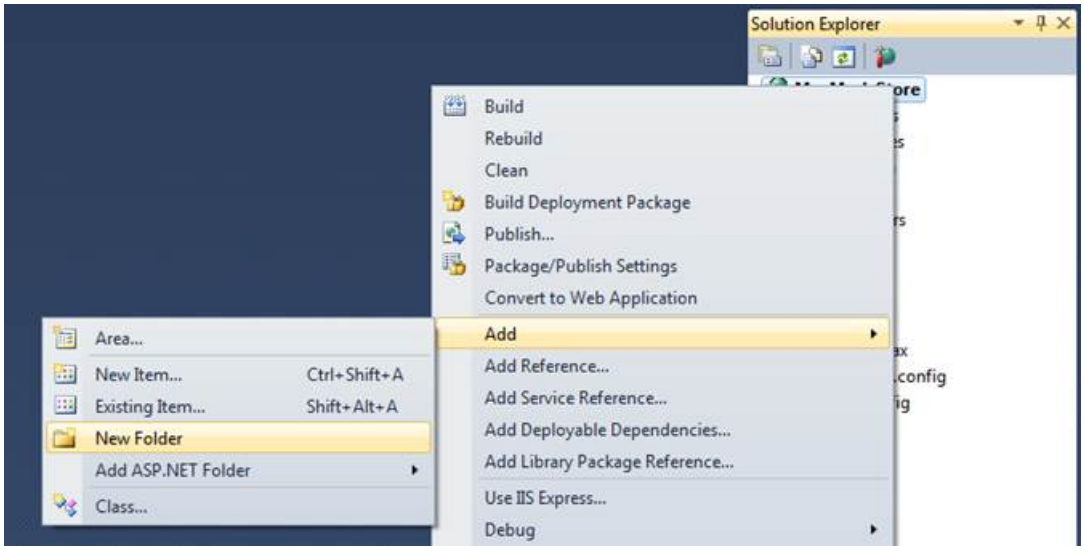
ViewModels

Our Shopping Cart Controller will need to communicate some complex information to its views which doesn't map cleanly to our Model objects. We don't want to modify our Models to suit our views; Model classes should represent our domain, not the user interface. One solution would be to pass the information to our Views using the ViewBag class, as we did with the Store Manager dropdown information, but passing a lot of information via ViewBag gets hard to manage.

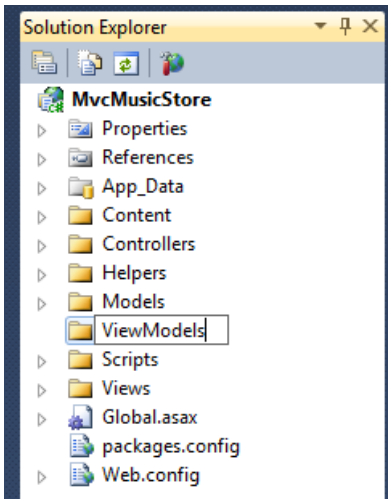
A solution to this is to use the *ViewModel* pattern. When using this pattern we create strongly-typed classes that are optimized for our specific view scenarios, and which expose properties for the dynamic values/content needed by our view templates. Our controller classes can then populate and pass these view-optimized classes to our view template to use. This enables type-safety, compile-time checking, and editor IntelliSense within view templates.

We'll create two View Models for use in our Shopping Cart controller: the ShoppingCartViewModel will hold the contents of the user's shopping cart, and the ShoppingCartRemoveViewModel will be used to display confirmation information when a user removes something from their cart.

Let's create a new ViewModels folder in the root of our project to keep things organized. Right-click the project, select Add / New Folder.



Name the folder ViewModels.



Next, add the ShoppingCartViewModel class in the ViewModels folder. It has two properties: a list of Cart items, and a decimal value to hold the total price for all items in the cart.

```
using System.Collections.Generic;
using MvcMusicStore.Models;

namespace MvcMusicStore.ViewModels
{
    public class ShoppingCartViewModel
    {
        public List<Cart> CartItems { get; set; }
        public decimal CartTotal { get; set; }
    }
}
```

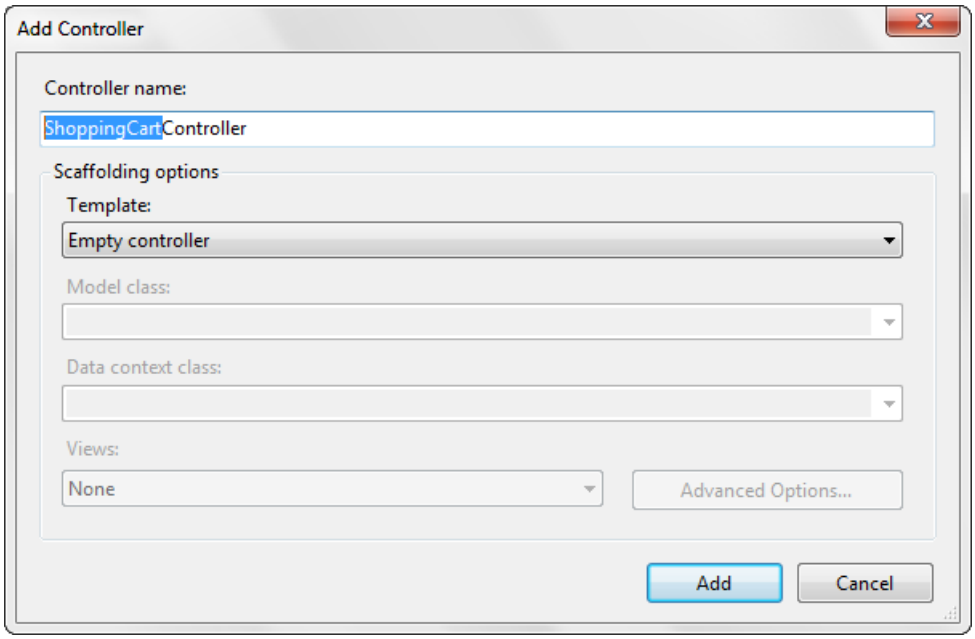
Now add the ShoppingCartRemoveViewModel to the ViewModels folder, with the following four properties.

```
namespace MvcMusicStore.ViewModels
{
    public class ShoppingCartRemoveViewModel
    {
        public string Message { get; set; }
        public decimal CartTotal { get; set; }
        public int CartCount { get; set; }
        public int ItemCount { get; set; }
        public int DeleteId { get; set; }
    }
}
```

The Shopping Cart Controller

The Shopping Cart controller has three main purposes: adding items to a cart, removing items from the cart, and viewing items in the cart. It will make use of the three classes we just created: ShoppingCartViewModel, ShoppingCartRemoveViewModel, and ShoppingCart. As in the StoreController and StoreManagerController, we'll add a field to hold an instance of MusicStoreEntities.

Add a new Shopping Cart controller to the project using the Empty controller template.



Here's the complete ShoppingCart Controller. The Index and Add Controller actions should look very familiar. The Remove and CartSummary controller actions handle two special cases, which we'll discuss in the following section.

```
using System.Linq;
using System.Web.Mvc;
using MvcMusicStore.Models;
using MvcMusicStore.ViewModels;

namespace MvcMusicStore.Controllers
{
    public class ShoppingCartController : Controller
    {
        MusicStoreEntities storeDB = new MusicStoreEntities();
        //
        // GET: /ShoppingCart/
        public ActionResult Index()
        {
            var cart = ShoppingCart.GetCart(this.HttpContext);

            // Set up our ViewModel
            var viewModel = new ShoppingCartViewModel
            {
                CartItems = cart.GetCartItems(),
                CartTotal = cart.GetTotal()
            };
            // Return the view
            return View(viewModel);
        }
        //
        // GET: /Store/AddToCart/5
        public ActionResult AddToCart(int id)
        {
            // Retrieve the album from the database
            var addedAlbum = storeDB.Albums
                .Single(album => album.AlbumId == id);

            // Add it to the shopping cart
            var cart = ShoppingCart.GetCart(this.HttpContext);

            cart.AddToCart(addedAlbum);

            // Go back to the main store page for more shopping
            return RedirectToAction("Index");
        }
        //
        // AJAX: /ShoppingCart/RemoveFromCart/5
```

```
        // AJAX: /ShoppingCart/RemoveFromCart/5
        [HttpPost]
        public ActionResult RemoveFromCart(int id)
        {
            // Remove the item from the cart
            var cart = ShoppingCart.GetCart(this.HttpContext);

            // Get the name of the album to display confirmation
            string albumName = storeDB.Carts
                .Single(item => item.RecordId == id).Album.Title;

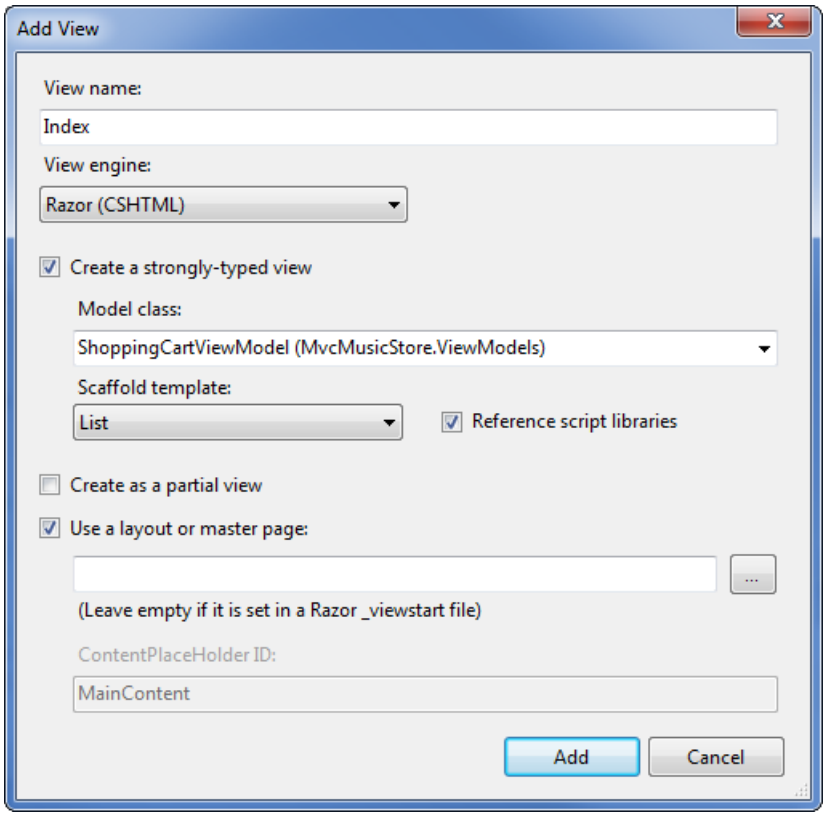
            // Remove from cart
            int itemCount = cart.RemoveFromCart(id);

            // Display the confirmation message
            var results = new ShoppingCartRemoveViewModel
            {
                Message = Server.HtmlEncode(albumName) +
                    " has been removed from your shopping cart.",
                CartTotal = cart.GetTotal(),
                CartCount = cart.GetCount(),
                ItemCount = itemCount,
                DeleteId = id
            };
            return Json(results);
        }
        //
        // GET: /ShoppingCart/CartSummary
        [ChildActionOnly]
        public ActionResult CartSummary()
        {
            var cart = ShoppingCart.GetCart(this.HttpContext);

            ViewData["CartCount"] = cart.GetCount();
            return PartialView("CartSummary");
        }
    }
}
```

Ajax Updates with jQuery

We'll next create a Shopping Cart Index page that is strongly typed to the ShoppingCartViewModel and uses the List View template using the same method as before.



However, instead of using an Html.ActionLink to remove items from the cart, we'll use jQuery to "wire up" the click event for all links in this view which have the HTML class RemoveLink. Rather than posting the form, this click event handler will just make an AJAX callback to our RemoveFromCart controller action. The RemoveFromCart returns a JSON serialized result, which our jQuery callback then parses and performs four quick updates to the page using jQuery:

- 1. Removes the deleted album from the list
- 2. Updates the cart count in the header
- 3. Displays an update message to the user
- 4. Updates the cart total price

Since the remove scenario is being handled by an Ajax callback within the Index view, we don't need an additional view for RemoveFromCart action. Here is the complete code for the /ShoppingCart/Index view:

```
@model MvcMusicStore.ViewModels.ShoppingCartViewModel
@{
    ViewBag.Title = "Shopping Cart";
}
<script src="/Scripts/jquery-1.4.4.min.js"
type="text/javascript"></script>
<script type="text/javascript">
    $(function () {
        // Document.ready -> link up remove event handler
        $(".RemoveLink").click(function () {
            // Get the id from the link
            var recordToDelete = $(this).attr("data-id");
            if (recordToDelete != '') {
                // Perform the ajax post
                $.post("/ShoppingCart/RemoveFromCart", {"id": recordToDelete },
                    function (data) {
                        // Successful requests get here
                        // Update the page elements
                        if (data.ItemCount == 0) {
                            $('#row-' + data.DeleteId).fadeOut('slow');
                        } else {
                            $('#item-count-' + data.DeleteId).text(data.ItemCount);
                        }
```

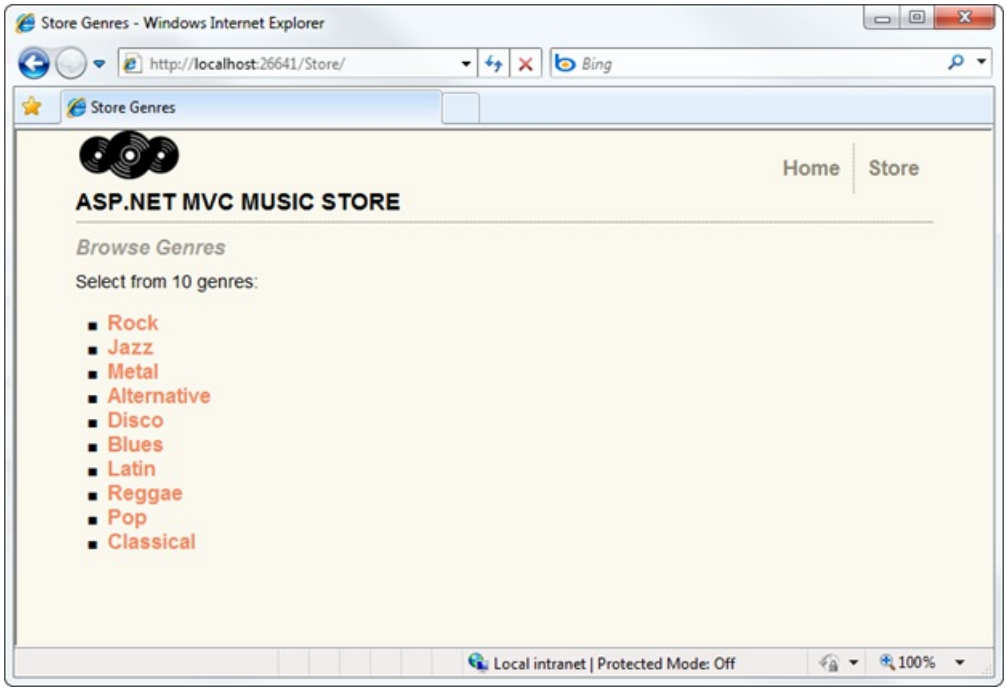
```
                        $('#cart-total').text(data.CartTotal);
                        $('#update-message').text(data.Message);
                        $('#cart-status').text('Cart (' + data.CartCount + ')');
                    });
            }
        });
    });
</script>
<h3>
    <em>Review</em> your cart:
 </h3>
<p class="button">
    @Html.ActionLink("Checkout
>>", "AddressAndPayment", "Checkout")
</p>
<div id="update-message">
</div>
<table>
    <tr>
        <th>
            Album Name
        </th>
        <th>
            Price (each)
        </th>
        <th>
            Quantity
        </th>
        <th></th>
    </tr>
    @foreach (var item in
Model.CartItems)
    {
        <tr id="row-@item.RecordId">
            <td>
                @Html.ActionLink(item.Album.Title,
"Details", "Store", new { id = item.AlbumId }, null)
            </td>
            <td>
                @item.Album.Price
            </td>
            <td id="item-count-@item.RecordId">
                @item.Count
            </td>
            <td>
                <a href="#" class="RemoveLink"
data-id="@item.RecordId">Remove
from cart</a>
            </td>
        </tr>
    }
    <tr>
        <td>
            Total
        </td>
        <td>
        </td>
        <td>
        </td>
        <td id="cart-total">
            @Model.CartTotal
        </td>
    </tr>
</table>
```

In order to test this out, we need to be able to add items to our shopping cart. We'll update our **Store Details** view to include an "Add to cart" button. While we're at it, we can include some of the Album additional information

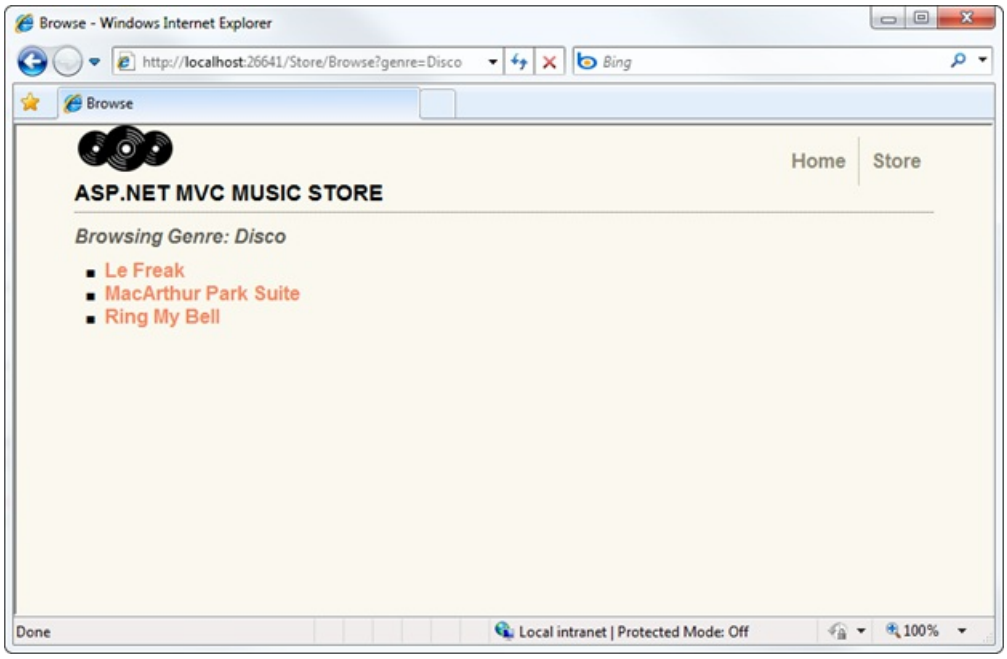
which we've added since we last updated this view: Genre, Artist, Price, and Album Art. The updated Store Details view code appears as shown below.

```
@model MvcMusicStore.Models.Album
@{
    ViewBag.Title = "Album - " + Model.Title;
 }
<h2>@Model.Title</h2>
<p>
    <img alt="@Model.Title"
src="@Model.AlbumArtUrl" />
</p>
<div id="album-details">
    <p>
        <em>Genre:</em>
        @Model.Genre.Name
    </p>
    <p>
        <em>Artist:</em>
        @Model.Artist.Name
    </p>
    <p>
        <em>Price:</em>
        @String.Format("{0:F}",
Model.Price)
    </p>
    <p class="button">
        @Html.ActionLink("Add to
cart", "AddToCart",
        "ShoppingCart", new { id = Model.AlbumId }, "")
    </p>
</div>
```

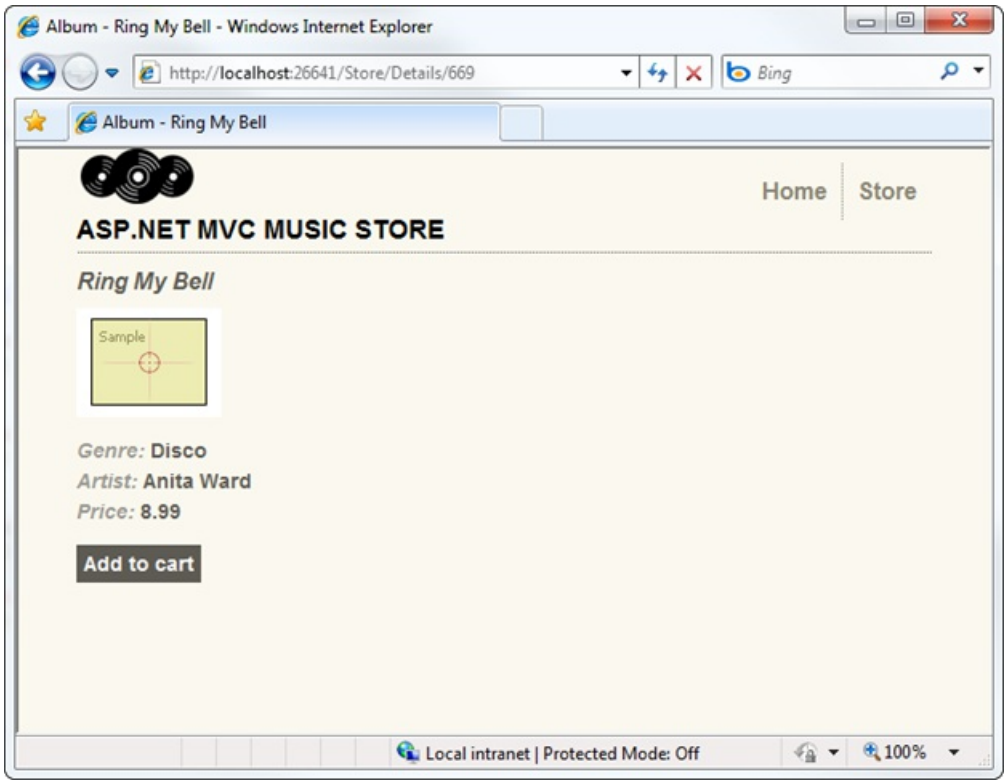
Now we can click through the store and test adding and removing Albums to and from our shopping cart. Run the application and browse to the Store Index.



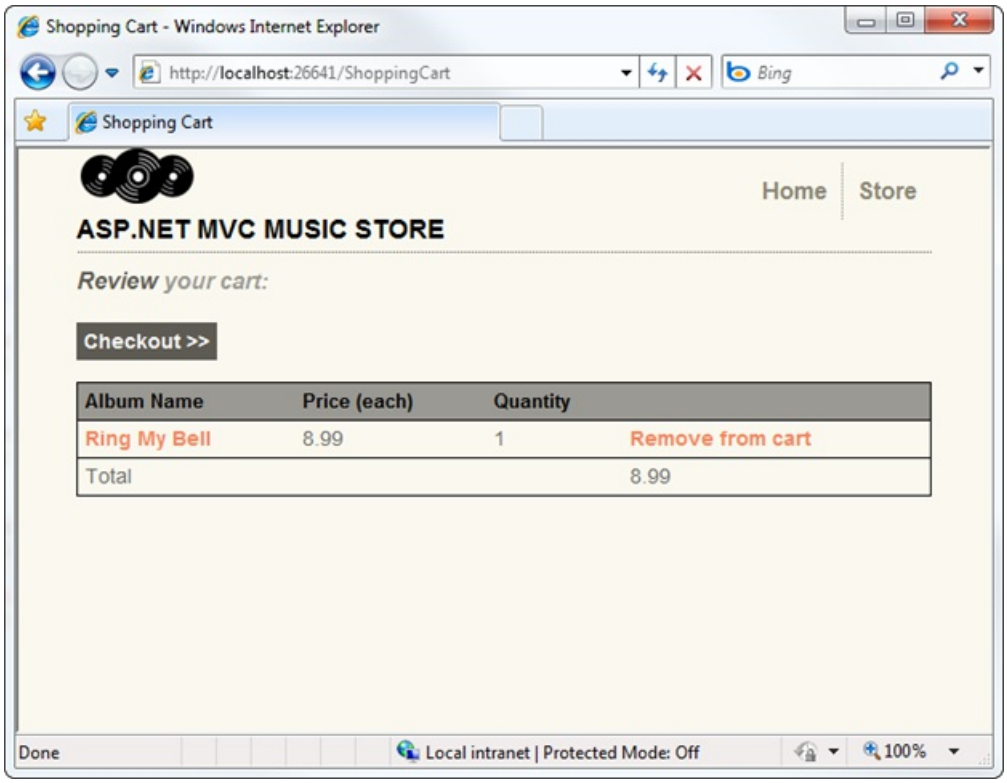
Next, click on a Genre to view a list of albums.



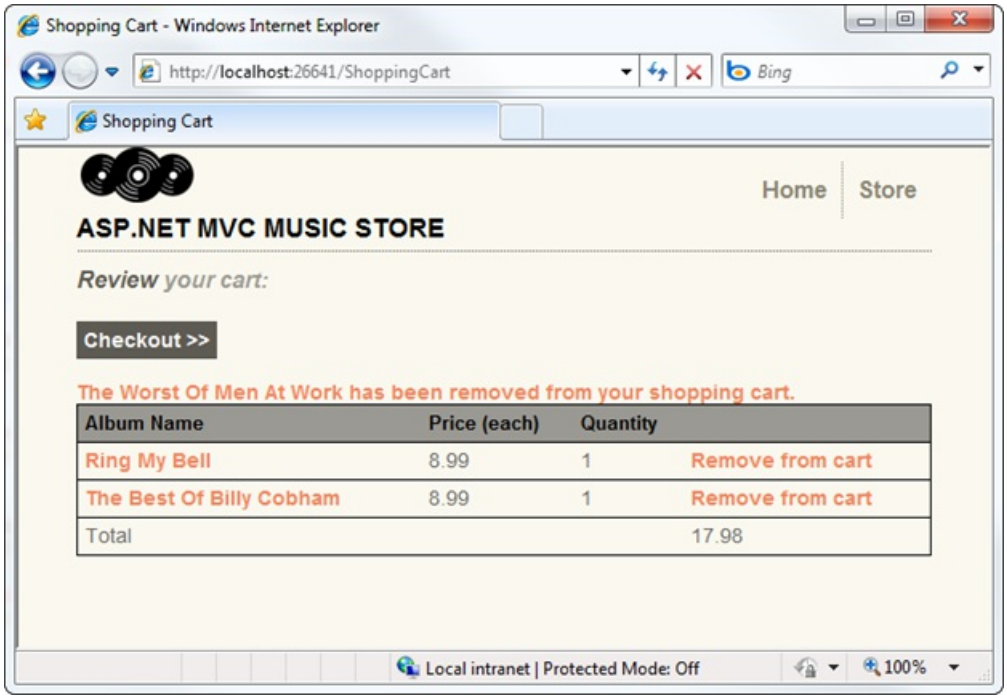
Clicking on an Album title now shows our updated Album Details view, including the "Add to cart" button.



Clicking the "Add to cart" button shows our Shopping Cart Index view with the shopping cart summary list.



After loading up your shopping cart, you can click on the Remove from cart link to see the Ajax update to your shopping cart.



We've built out a working shopping cart which allows unregistered users to add items to their cart. In the following section, we'll allow them to register and complete the checkout process.

PREVIOUS NEXT

Part 9: Registration and Checkout

2/20/2020 • 9 minutes to read • Edit Online

by Jon Galloway

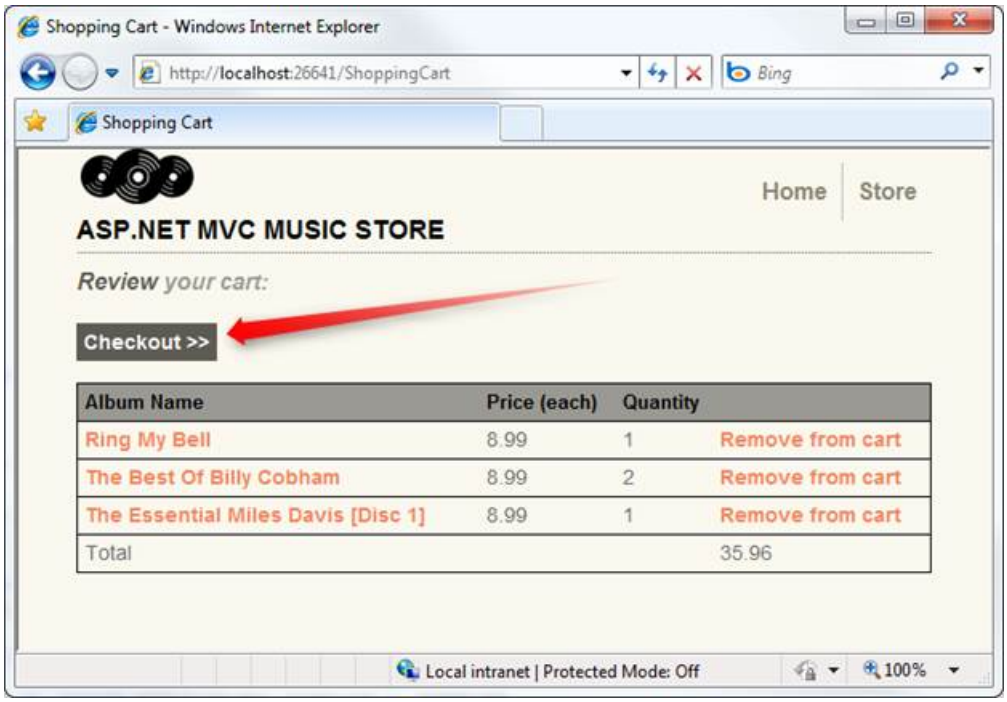
The MVC Music Store is a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Studio for web development.

The MVC Music Store is a lightweight sample store implementation which sells music albums online, and implements basic site administration, user sign-in, and shopping cart functionality.

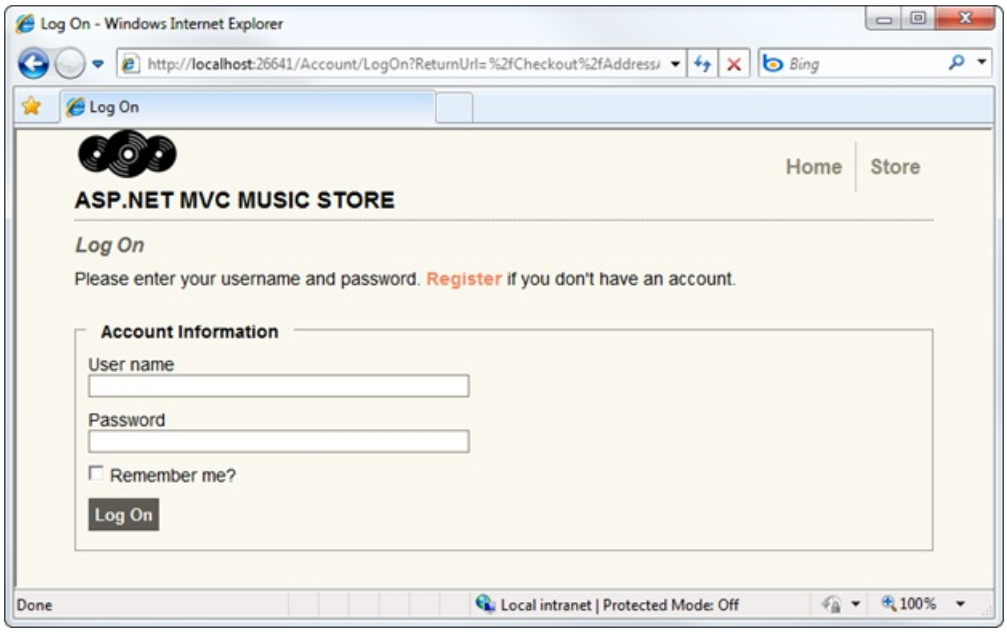
This tutorial series details all of the steps taken to build the ASP.NET MVC Music Store sample application. Part 9 covers Registration and Checkout.

In this section, we will be creating a CheckoutController which will collect the shopper's address and payment information. We will require users to register with our site prior to checking out, so this controller will require authorization.

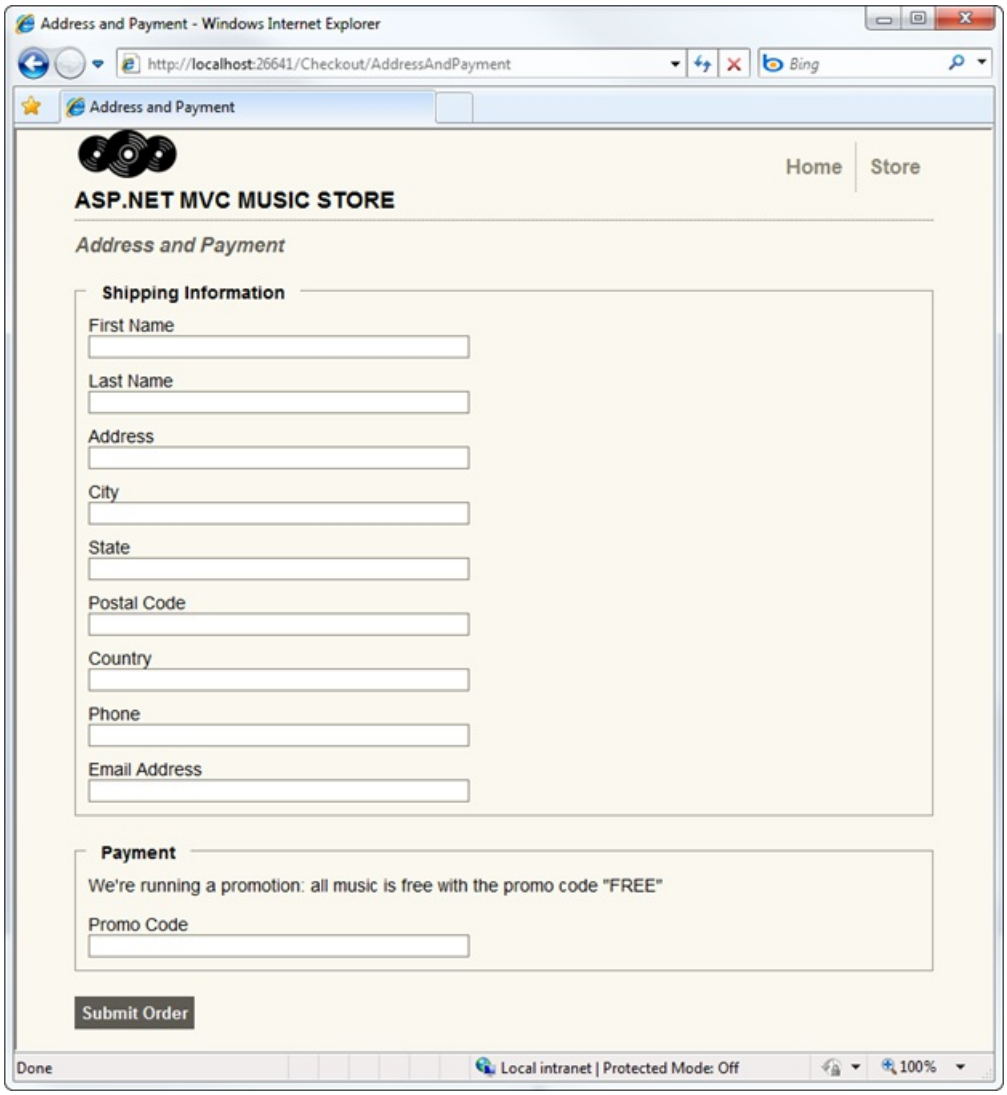
Users will navigate to the checkout process from their shopping cart by clicking the "Checkout" button.



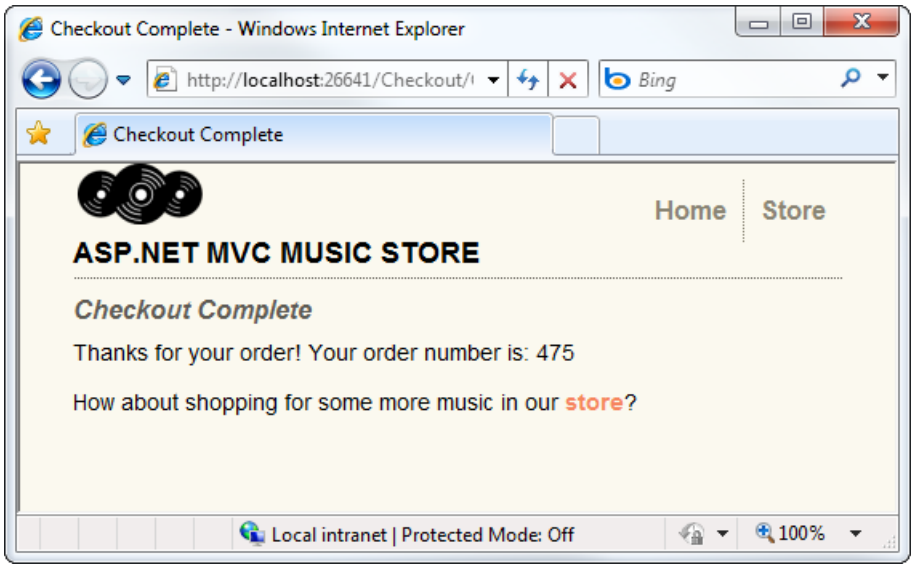
If the user is not logged in, they will be prompted to.



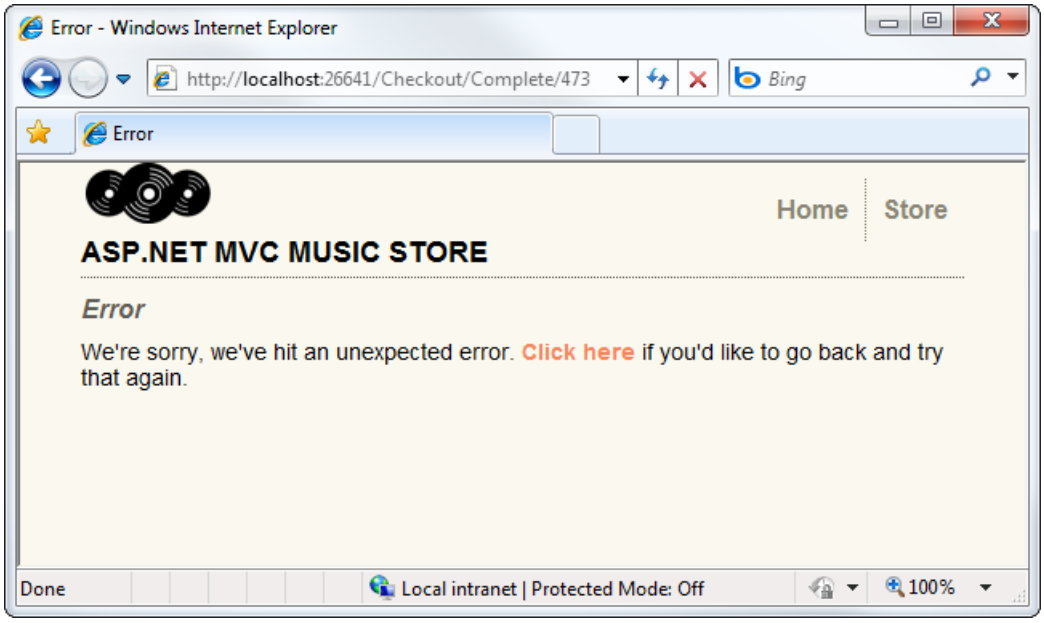
Upon successful login, the user is then shown the Address and Payment view.



Once they have filled the form and submitted the order, they will be shown the order confirmation screen.



Attempting to view either a non-existent order or an order that doesn't belong to you will show the Error view.



Migrating the Shopping Cart

While the shopping process is anonymous, when the user clicks on the Checkout button, they will be required to register and login. Users will expect that we will maintain their shopping cart information between visits, so we will need to associate the shopping cart information with a user when they complete registration or login.

This is actually very simple to do, as our ShoppingCart class already has a method which will associate all the items in the current cart with a username. We will just need to call this method when a user completes registration or login.

Open the **AccountController** class that we added when we were setting up Membership and Authorization. Add a using statement referencing MvcMusicStore.Models, then add the following MigrateShoppingCart method:

```
private void MigrateShoppingCart(string UserName)
{
    // Associate shopping cart items with logged-in user
    var cart = ShoppingCart.GetCart(this.HttpContext);

    cart.MigrateCart(UserName);
    Session[ShoppingCart.CartSessionKey] = UserName;
}
```

Next, modify the LogOn post action to call MigrateShoppingCart after the user has been validated, as shown below:

```
//
// POST: /Account/LogOn
[HttpPost]
 public ActionResult LogOn(LogOnModel model, string returnUrl)
 {
    if (ModelState.IsValid)
    {
        if (Membership.ValidateUser(model.UserName, model.Password))
        {
            MigrateShoppingCart(model.UserName);

            FormsAuthentication.SetAuthCookie(model.UserName,
                model.RememberMe);
            if (Url.IsLocalUrl(returnUrl) && returnUrl.Length > 1
                && returnUrl.StartsWith("/")
                && !returnUrl.StartsWith("//") &&
                !returnUrl.StartsWith("/\\"))
            {
                return Redirect(returnUrl);
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
        else
        {
            ModelState.AddModelError("", "The user name or password provided is incorrect.");
        }
    }
    // If we got this far, something failed, redisplay form
    return View(model);
 }
```

Make the same change to the Register post action, immediately after the user account is successfully created:

```
//
// POST: /Account/Register
[HttpPost]
 public ActionResult Register(RegisterModel model)
 {
    if (ModelState.IsValid)
    {
        // Attempt to register the user
        MembershipCreateStatus createStatus;
        Membership.CreateUser(model.UserName, model.Password, model.Email,
               "question", "answer", true, null, out
               createStatus);

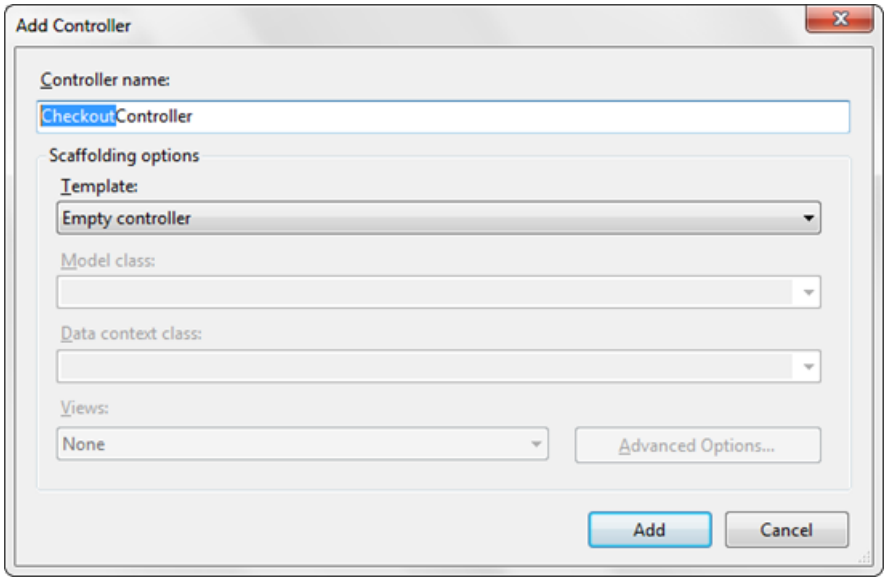
        if (createStatus == MembershipCreateStatus.Success)
        {
            MigrateShoppingCart(model.UserName);

            FormsAuthentication.SetAuthCookie(model.UserName, false /*
                  createPersistentCookie */);
            return RedirectToAction("Index", "Home");
        }
        else
        {
            ModelState.AddModelError("", ErrorCodeToString(createStatus));
        }
    }
    // If we got this far, something failed, redisplay form
    return View(model);
 }
```

That's it - now an anonymous shopping cart will be automatically transferred to a user account upon successful registration or login.

Creating the CheckoutController

Right-click on the Controllers folder and add a new Controller to the project named CheckoutController using the Empty controller template.



First, add the Authorize attribute above the Controller class declaration to require users to register before checkout:

```
namespace MvcMusicStore.Controllers
{
    [Authorize]
    public class CheckoutController : Controller
```

Note: This is similar to the change we previously made to the StoreManagerController, but in that case the Authorize attribute required that the user be in an Administrator role. In the Checkout Controller, we're requiring the user be logged in but aren't requiring that they be administrators.

For the sake of simplicity, we won't be dealing with payment information in this tutorial. Instead, we are allowing users to check out using a promotional code. We will store this promotional code using a constant named PromoCode.

As in the StoreController, we'll declare a field to hold an instance of the MusicStoreEntities class, named storeDB. In order to make use of the MusicStoreEntities class, we will need to add a using statement for the MvcMusicStore.Models namespace. The top of our Checkout controller appears below.

```
using System;
using System.Linq;
using System.Web.Mvc;
using MvcMusicStore.Models;

namespace MvcMusicStore.Controllers
{
    [Authorize]
    public class CheckoutController : Controller
    {
        MusicStoreEntities storeDB = new MusicStoreEntities();
        const string PromoCode = "FREE";
```

The CheckoutController will have the following controller actions:

AddressAndPayment (GET method) will display a form to allow the user to enter their information.

AddressAndPayment (POST method) will validate the input and process the order.

Complete will be shown after a user has successfully finished the checkout process. This view will include the user's order number, as confirmation.

First, let's rename the Index controller action (which was generated when we created the controller) to AddressAndPayment. This controller action just displays the checkout form, so it doesn't require any model information.

```
//
// GET: /Checkout/AddressAndPayment
public ActionResult AddressAndPayment()
{
    return View();
}
```

Our AddressAndPayment POST method will follow the same pattern we used in the StoreManagerController: it will try to accept the form submission and complete the order, and will re-display the form if it fails.

After validating the form input meets our validation requirements for an Order, we will check the PromoCode form value directly. Assuming everything is correct, we will save the updated information with the order, tell the ShoppingCart object to complete the order process, and redirect to the Complete action.

```
//
// POST: /Checkout/AddressAndPayment
[HttpPost]
public ActionResult AddressAndPayment(FormCollection values)
{
    var order = new Order();
    TryUpdateModel(order);

    try
    {
        if (string.Equals(values["PromoCode"], PromoCode,
            StringComparison.OrdinalIgnoreCase) == false)
        {
            return View(order);
        }
        else
        {
            order.Username = User.Identity.Name;
            order.OrderDate = DateTime.Now;

            //Save Order
            storeDB.Orders.Add(order);
            storeDB.SaveChanges();
            //Process the order
            var cart = ShoppingCart.GetCart(this.HttpContext);
            cart.CreateOrder(order);

            return RedirectToAction("Complete",
                new { id = order.OrderId });
        }
    }
    catch
    {
        //Invalid - redisplay with errors
        return View(order);
    }
}
```

Upon successful completion of the checkout process, users will be redirected to the Complete controller action. This action will perform a simple check to validate that the order does indeed belong to the logged-in user before showing the order number as a confirmation.

```
//
// GET: /Checkout/Complete
public ActionResult Complete(int id)
{
    // Validate customer owns this order
    bool isValid = storeDB.Orders.Any(
        o => o.OrderId == id &&
        o.Username == User.Identity.Name);

    if (isValid)
    {
        return View(id);
    }
    else
    {
        return View("Error");
    }
}
```

Note: The Error view was automatically created for us in the /Views/Shared folder when we began the project.

The complete CheckoutController code is as follows:

```
using System;
using System.Linq;
using System.Web.Mvc;
using MvcMusicStore.Models;

namespace MvcMusicStore.Controllers
{
    [Authorize]
    public class CheckoutController : Controller
    {
        MusicStoreEntities storeDB = new MusicStoreEntities();
        const string PromoCode = "FREE";
        //
        // GET: /Checkout/AddressAndPayment
        public ActionResult AddressAndPayment()
        {
            return View();
        }
        //
        // POST: /Checkout/AddressAndPayment
        [HttpPost]
        public ActionResult AddressAndPayment(FormCollection values)
        {
            var order = new Order();
            TryUpdateModel(order);

            try
            {
                if (string.Equals(values["PromoCode"], PromoCode,
                    StringComparison.OrdinalIgnoreCase) == false)
                {
                    return View(order);
                }
                else
                {
                    order.Username = User.Identity.Name;
                    order.OrderDate = DateTime.Now;

                    //Save Order
                    storeDB.Orders.Add(order);
                    storeDB.SaveChanges();
                    //Process the order
                    var cart = ShoppingCart.GetCart(this.HttpContext);
                    cart.CreateOrder(order);

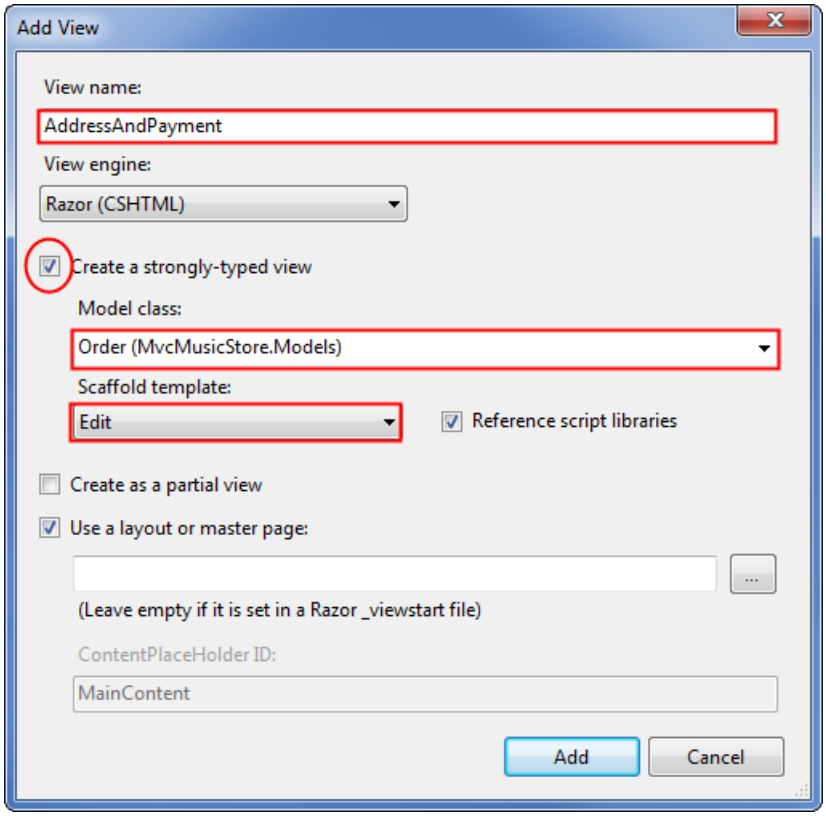
                    return RedirectToAction("Complete",
                        new { id = order.OrderId });
                }
            }
            catch
            {
                //Invalid - redisplay with errors
                return View(order);
            }
        }
        //
        // GET: /Checkout/Complete
        public ActionResult Complete(int id)
        {
            // Validate customer owns this order
            bool isValid = storeDB.Orders.Any(
                o => o.OrderId == id &&
                o.Username == User.Identity.Name);

            if (isValid)
            {
                return View(id);
            }
```

```
            else
            {
                return View("Error");
            }
        }
    }
}
```

Adding the AddressAndPayment view

Now, let's create the AddressAndPayment view. Right-click on one of the AddressAndPayment controller actions and add a view named AddressAndPayment which is strongly typed as an Order and uses the Edit template, as shown below.



This view will make use of two of the techniques we looked at while building the StoreManagerEdit view:

- We will use Html.EditorForModel() to display form fields for the Order model
- We will leverage validation rules using an Order class with validation attributes

We'll start by updating the form code to use Html.EditorForModel(), followed by an additional textbox for the Promo Code. The complete code for the AddressAndPayment view is shown below.

```
@model MvcMusicStore.Models.Order
@{
    ViewBag.Title = "Address And Payment";
}
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")"
type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
type="text/javascript"></script>
@using (Html.BeginForm()) {

    <h2>Address And Payment</h2>
    <fieldset>
        <legend>Shipping Information</legend>
        @Html.EditorForModel()
    </fieldset>
    <fieldset>
        <legend>Payment</legend>
        <p>We're running a promotion: all music is free
            with the promo code: "FREE"</p>
        <div class="editor-label">
            @Html.Label("Promo Code")
        </div>
        <div class="editor-field">
            @Html.TextBox("PromoCode")
        </div>
    </fieldset>

    <input type="submit" value="Submit Order" />
}
```

Defining validation rules for the Order

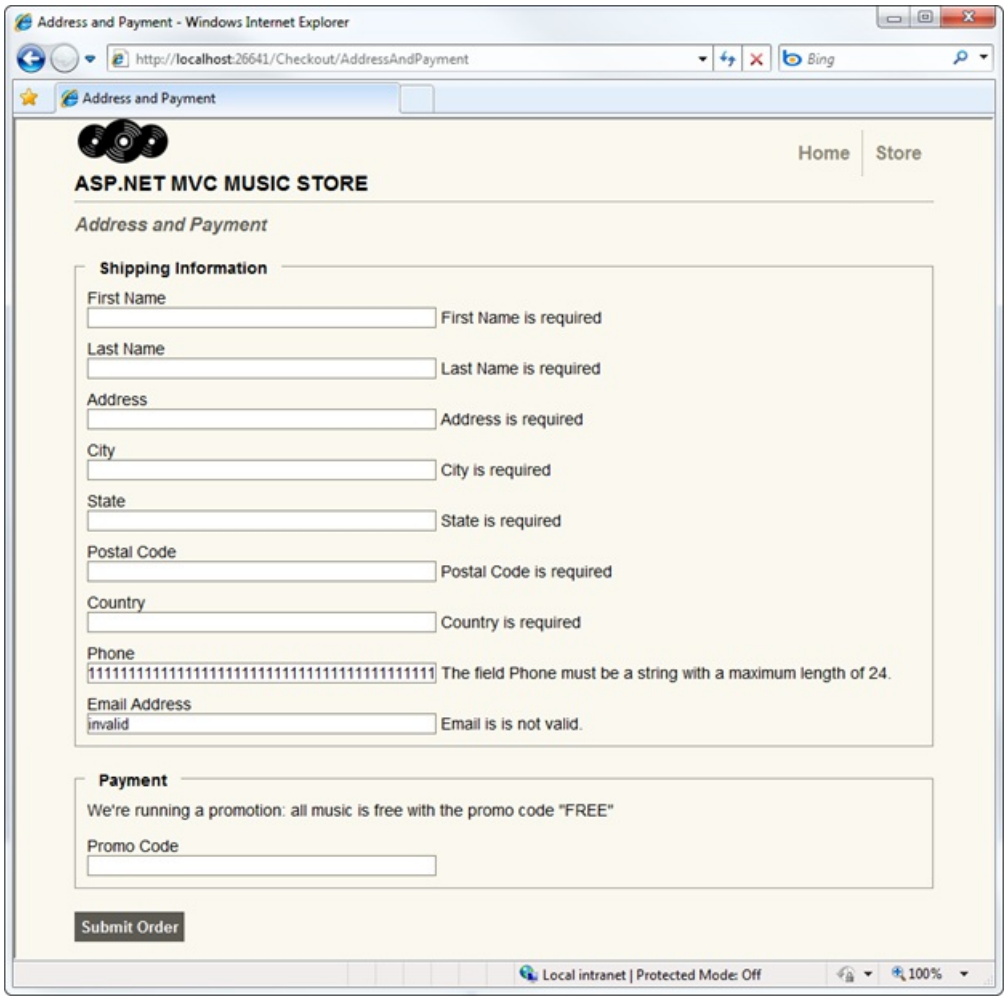
Now that our view is set up, we will set up the validation rules for our Order model as we did previously for the Album model. Right-click on the Models folder and add a class named Order. In addition to the validation attributes we used previously for the Album, we will also be using a Regular Expression to validate the user's email address.

```
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace MvcMusicStore.Models
{
    [Bind(Exclude = "OrderId")]
    public partial class Order
    {
        [ScaffoldColumn(false)]
        public int OrderId { get; set; }
        [ScaffoldColumn(false)]
        public System.DateTime OrderDate { get; set; }
        [ScaffoldColumn(false)]
        public string Username { get; set; }
        [Required(ErrorMessage = "First Name is required")]
        [DisplayName("First Name")]
        [StringLength(160)]
        public string FirstName { get; set; }
        [Required(ErrorMessage = "Last Name is required")]
        [DisplayName("Last Name")]
        [StringLength(160)]
        public string LastName { get; set; }
        [Required(ErrorMessage = "Address is required")]
        [StringLength(70)]
        public string Address { get; set; }
        [Required(ErrorMessage = "City is required")]
        [StringLength(40)]
        public string City { get; set; }
        [Required(ErrorMessage = "State is required")]
        [StringLength(40)]
        public string State { get; set; }
        [Required(ErrorMessage = "Postal Code is required")]
        [DisplayName("Postal Code")]
        [StringLength(10)]
        public string PostalCode { get; set; }
        [Required(ErrorMessage = "Country is required")]
        [StringLength(40)]
        public string Country { get; set; }
        [Required(ErrorMessage = "Phone is required")]
        [StringLength(24)]
        public string Phone { get; set; }
        [Required(ErrorMessage = "Email Address is required")]
        [DisplayName("Email Address")]

        [RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
            ErrorMessage = "Email is is not valid.")]
        [DataType(DataType.EmailAddress)]
        public string Email { get; set; }
        [ScaffoldColumn(false)]
        public decimal Total { get; set; }
        public List<OrderDetail> OrderDetails { get; set; }
    }
}
```

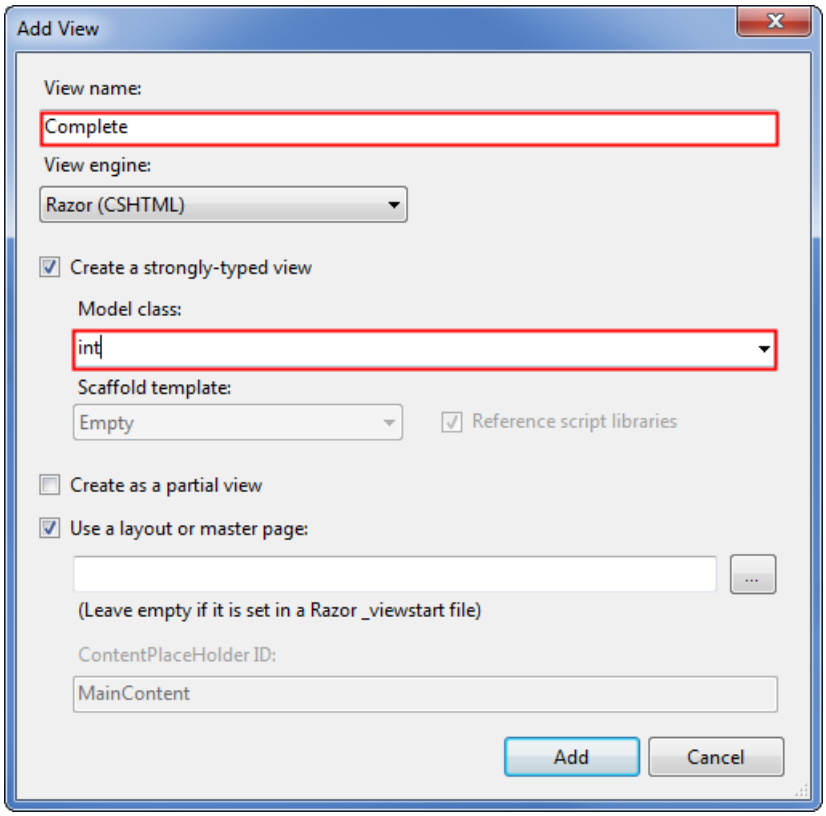
Attempting to submit the form with missing or invalid information will now show error message using client-side validation.



Okay, we've done most of the hard work for the checkout process; we just have a few odds and ends to finish. We need to add two simple views, and we need to take care of the handoff of the cart information during the login process.

Adding the Checkout Complete view

The Checkout Complete view is pretty simple, as it just needs to display the Order ID. Right-click on the Complete controller action and add a view named Complete which is strongly typed as an int.



Now we will update the view code to display the Order ID, as shown below.

```
@model int
@{
    ViewBag.Title = "Checkout Complete";
}
<h2>Checkout Complete</h2>
<p>Thanks for your order! Your order number is: @Model</p>
<p>How about shopping for some more music in our
    @Html.ActionLink("store",
"Index", "Home")
</p>
```

Updating The Error view

The default template includes an Error view in the Shared views folder so that it can be re-used elsewhere in the site. This Error view contains a very simple error and doesn't use our site Layout, so we'll update it.

Since this is a generic error page, the content is very simple. We'll include a message and a link to navigate to the previous page in history if the user wants to re-try their action.

```
@{
    ViewBag.Title = "Error";
}

<h2>Error</h2>

<p>We're sorry, we've hit an unexpected error.
    <a href="javascript:history.go(-1)">Click here</a>
    if you'd like to go back and try that again.</p>
```

PREVIOUS NEXT

Part 10: Final Updates to Navigation and Site Design, Conclusion

2/20/2020 • 6 minutes to read • Edit Online

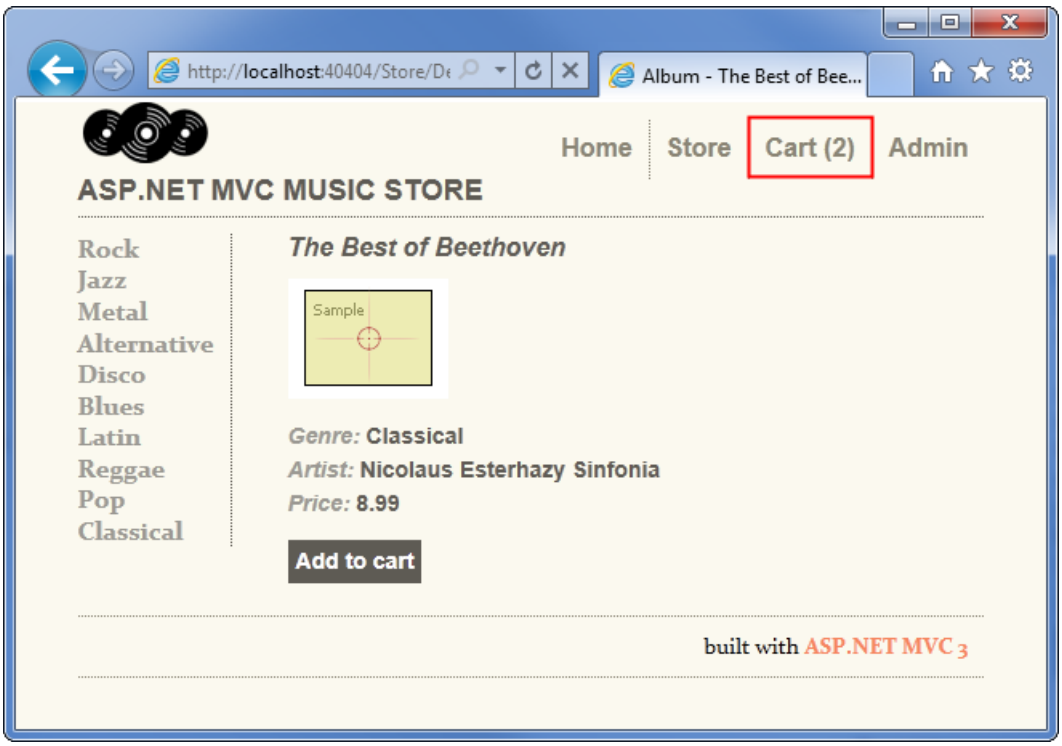
by Jon Galloway

> The MVC Music Store is a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Studio for web development.
>
> The MVC Music Store is a lightweight sample store implementation which sells music albums online, and implements basic site administration, user sign-in, and shopping cart functionality.
>
> This tutorial series details all of the steps taken to build the ASP.NET MVC Music Store sample application. Part 10 covers Final Updates to Navigation and Site Design, Conclusion.

We've completed all the major functionality for our site, but we still have some features to add to the site navigation, the home page, and the Store Browse page.

Creating the Shopping Cart Summary Partial View

We want to expose the number of items in the user's shopping cart across the entire site.



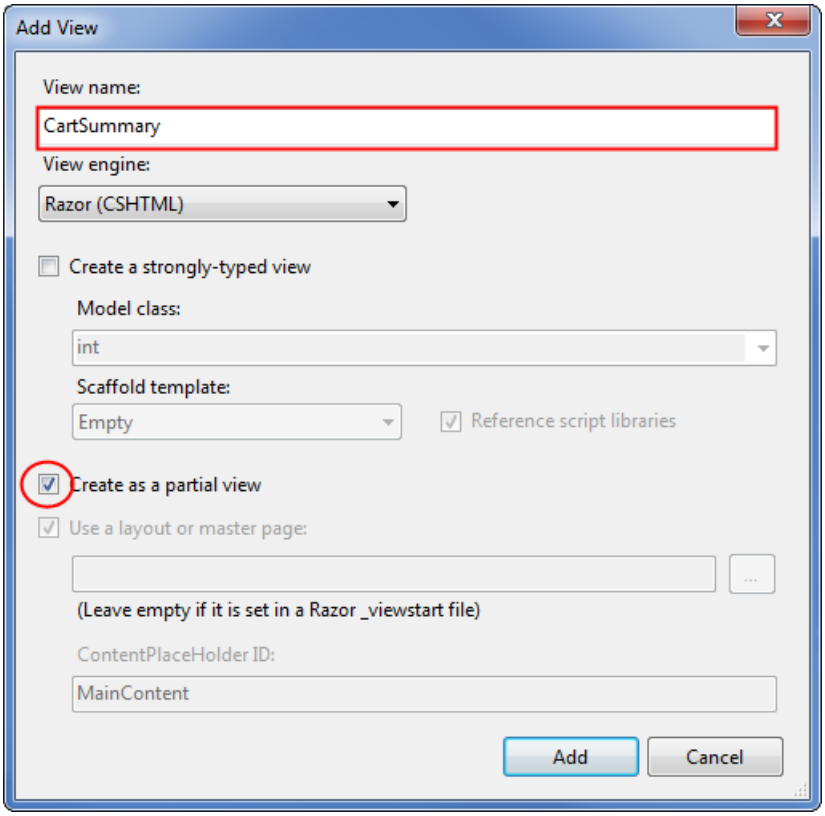
We can easily implement this by creating a partial view which is added to our Site.master.

As shown previously, the ShoppingCart controller includes a CartSummary action method which returns a partial view:

```
//
// GET: /ShoppingCart/CartSummary
[ChildActionOnly]
 public ActionResult CartSummary()
{
    var cart = ShoppingCart.GetCart(this.HttpContext);

    ViewData["CartCount"] = cart.GetCount();
    return PartialView("CartSummary");
}
```

To create the CartSummary partial view, right-click on the Views/ShoppingCart folder and select Add View. Name the view CartSummary and check the "Create a partial view" checkbox as shown below.



The CartSummary partial view is really simple - it's just a link to the ShoppingCart Index view which shows the number of items in the cart. The complete code for CartSummary.cshtml is as follows:

```
@Html.ActionLink("Cart
(" + ViewData["CartCount"] + ")",
    "Index",
    "ShoppingCart",
    new { id = "cart-status" })
```

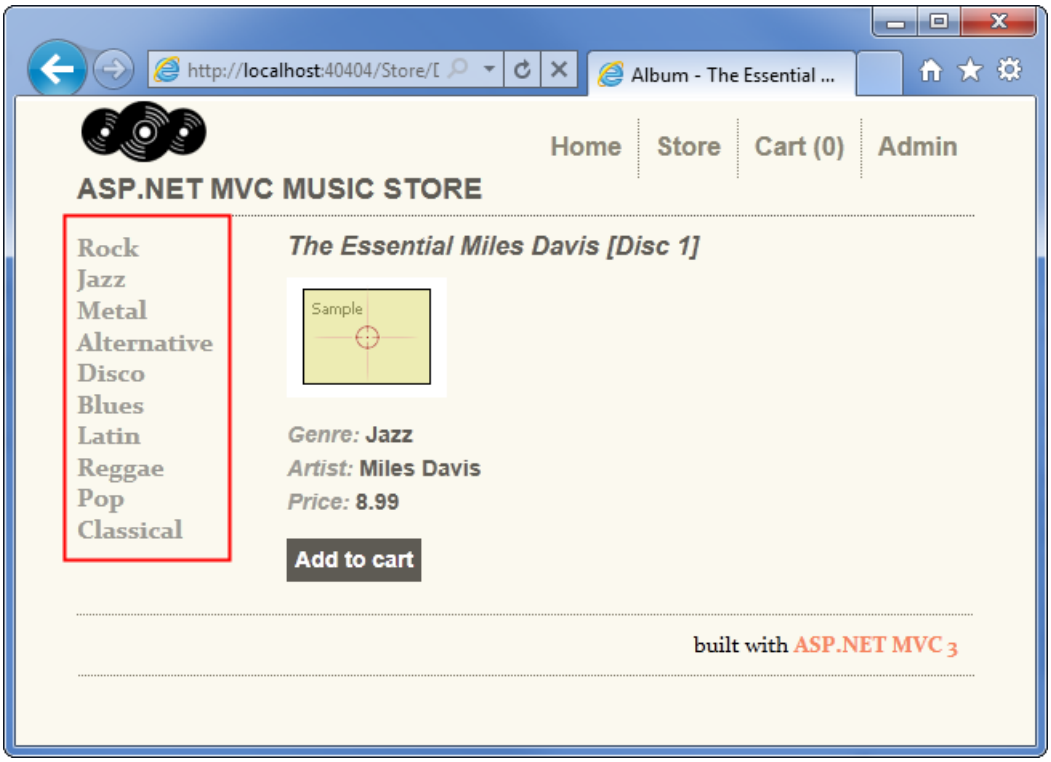
We can include a partial view in any page in the site, including the Site master, by using the Html.RenderAction method. RenderAction requires us to specify the Action Name ("CartSummary") and the Controller Name ("ShoppingCart") as below.

```
@Html.RenderAction("CartSummary",
"ShoppingCart")
```

Before adding this to the site Layout, we will also create the Genre Menu so we can make all of our Site.master updates at one time.

Creating the Genre Menu Partial View

We can make it a lot easier for our users to navigate through the store by adding a Genre Menu which lists all the Genres available in our store.



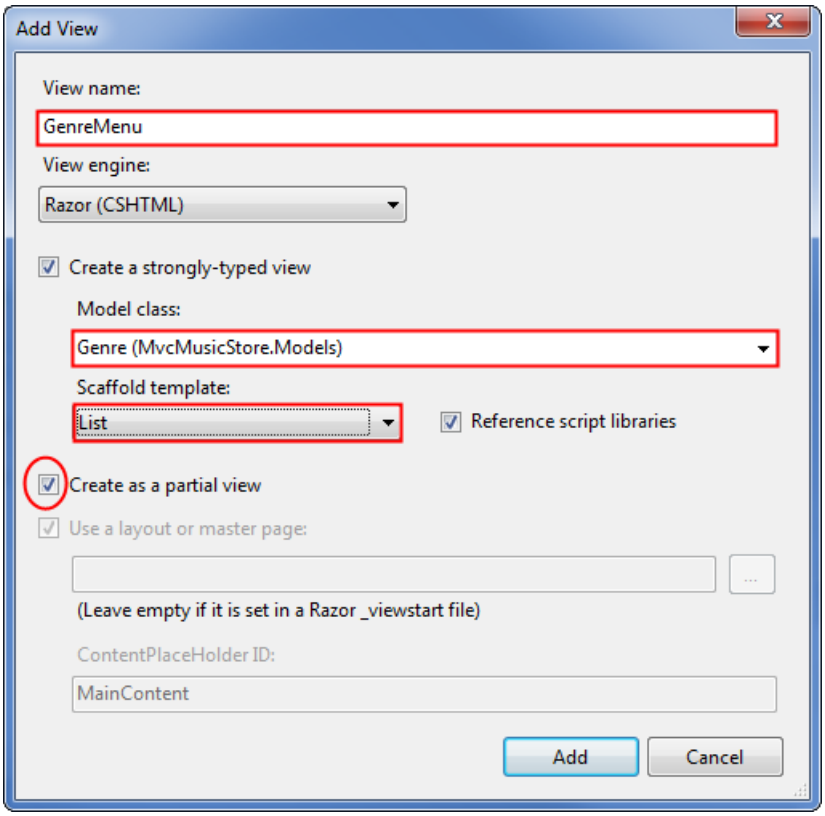
We will follow the same steps also create a GenreMenu partial view, and then we can add them both to the Site master. First, add the following GenreMenu controller action to the StoreController:

```
//
// GET: /Store/GenreMenu
[ChildActionOnly]
 public ActionResult GenreMenu()
{
    var genres = storeDB.Genres.ToList();
    return PartialView(genres);
 }
```

This action returns a list of Genres which will be displayed by the partial view, which we will create next.

Note: We have added the [ChildActionOnly] attribute to this controller action, which indicates that we only want this action to be used from a Partial View. This attribute will prevent the controller action from being executed by browsing to /Store/GenreMenu. This isn't required for partial views, but it is a good practice, since we want to make sure our controller actions are used as we intend. We are also returning PartialView rather than View, which lets the view engine know that it shouldn't use the Layout for this view, as it is being included in other views.

Right-click on the GenreMenu controller action and create a partial view named GenreMenu which is strongly typed using the Genre view data class as shown below.



Update the view code for the GenreMenu partial view to display the items using an unordered list as follows.

```
@model IEnumerable<MvcMusicStore.Models.Genre>
<ul id="categories">
    @foreach (var genre in Model)
    {
        <li>@Html.ActionLink(genre.Name,
                "Browse", "Store",
                new { Genre = genre.Name }, null)
        </li>
    }
</ul>
```

Updating Site Layout to display our Partial Views

We can add our partial views to the Site Layout (/Views/Shared/_Layout.cshtml) by calling Html.RenderAction(). We'll add them both in, as well as some additional markup to display them, as shown below:

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")"
rel="stylesheet"
        type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")"
        type="text/javascript"></script>
</head>
<body>
    <div id="header">
        <h1><a href="/">ASP.NET MVC MUSIC
STORE</a></h1>
        <ul id="navlist">
            <li class="first">
                <a href="@Url.Content("~")" id="current">
                    Home</a></li>
            <li><a href="@Url.Content("~/Store/")">Store</a></li>
            <li>@{Html.RenderAction("CartSummary", "ShoppingCart");}</li>
            <li><a href="@Url.Content("~/StoreManager/")">
                    Admin</a></li>
        </ul>
    </div>
    @{Html.RenderAction("GenreMenu", "Store");}
    <div id="main">
        @RenderBody()
    </div>
    <div id="footer">
        built with <a href="http://asp.net/mvc">ASP.NET MVC 3</a>
    </div>
</body>
</html>
```

Now when we run the application, we will see the Genre in the left navigation area and the Cart Summary at the top.

Update to the Store Browse page

The Store Browse page is functional, but doesn't look very good. We can update the page to show the albums in a better layout by updating the view code (found in /Views/Store/Browse.cshtml) as follows:

```
@model MvcMusicStore.Models.Genre

@{
    ViewBag.Title = "Browse Albums";
}

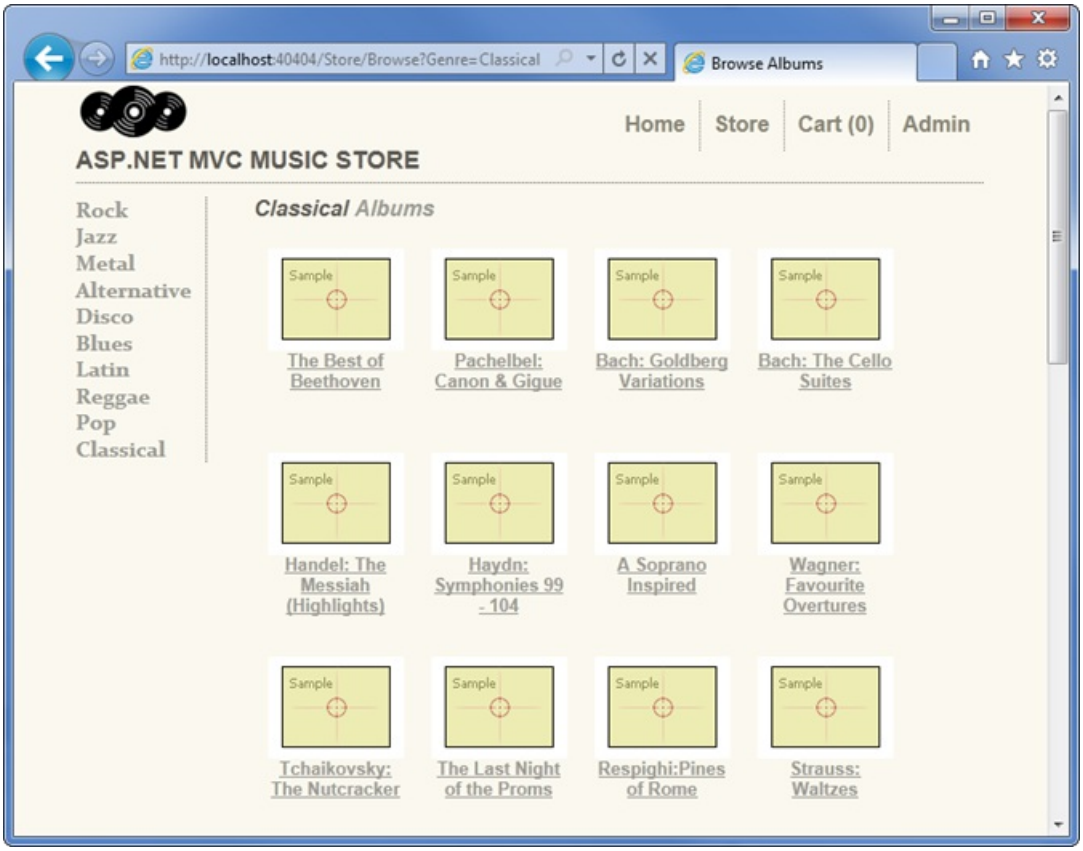
<div class="genre">
    <h3><em>@Model.Name</em> Albums</h3>

    <ul id="album-list">
        @foreach (var album in Model.Albums)
        {
            <li>
                <a href="@Url.Action("Details",
                    new { id = album.AlbumId })">
                    <img alt="@album.Title"
                        src="@album.AlbumArtUrl" />
                    <span>@album.Title</span>
                </a>
            </li>
        }
    </ul>
</div>
```

Here we are making use of Url.Action rather than Html.ActionLink so that we can apply special formatting to the link to include the album artwork.

Note: We are displaying a generic album cover for these albums. This information is stored in the database and is editable via the Store Manager. You are welcome to add your own artwork.

Now when we browse to a Genre, we will see the albums shown in a grid with the album artwork.



Updating the Home Page to show Top Selling Albums

We want to feature our top selling albums on the home page to increase sales. We'll make some updates to our HomeController to handle that, and add in some additional graphics as well.

First, we'll add a navigation property to our Album class so that EntityFramework knows that they're associated. The last few lines of our **Album** class should now look like this:

```
public virtual Genre  Genre                 { get; set; }
public virtual Artist Artist                { get; set; }
public virtual List<OrderDetail>OrderDetails { get; set; }
    }
}
```

Note: This will require adding a using statement to bring in the System.Collections.Generic namespace.

First, we'll add a storeDB field and the MvcMusicStore.Models using statements, as in our other controllers. Next, we'll add the following method to the HomeController which queries our database to find top selling albums according to OrderDetails.

```
private List<Album> GetTopSellingAlbums(int count)
 {
    // Group the order details by album and return
    // the albums with the highest count
    return storeDB.Albums
        .OrderByDescending(a => a.OrderDetails.Count())
        .Take(count)
        .ToList();
}
```

This is a private method, since we don't want to make it available as a controller action. We are including it in the HomeController for simplicity, but you are encouraged to move your business logic into separate service classes as appropriate.

With that in place, we can update the Index controller action to query the top 5 selling albums and return them to the view.

```
public ActionResult Index()
{
    // Get most popular albums
    var albums = GetTopSellingAlbums(5);

    return View(albums);
 }
```

The complete code for the updated HomeController is as shown below.

```
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using MvcMusicStore.Models;

namespace MvcMusicStore.Controllers
{
    public class HomeController : Controller
    {
        //
        // GET: /Home/
        MusicStoreEntities storeDB = new MusicStoreEntities();
        public ActionResult Index()
        {
            // Get most popular albums
            var albums = GetTopSellingAlbums(5);

            return View(albums);
        }
        private List<Album> GetTopSellingAlbums(int count)
        {
            // Group the order details by album and return
            // the albums with the highest count
            return storeDB.Albums
                .OrderByDescending(a => a.OrderDetails.Count())
                .Take(count)
                .ToList();
        }
    }
}
```

Finally, we'll need to update our Home Index view so that it can display a list of albums by updating the Model type and adding the album list to the bottom. We will take this opportunity to also add a heading and a promotion section to the page.

```
@model List<MvcMusicStore.Models.Album>
@{
    ViewBag.Title = "ASP.NET MVC Music Store";
}
<div id="promotion">
</div>
<h3><em>Fresh</em> off the grill</h3>
<ul id="album-list">
    @foreach (var album in Model)
    {
        <li><a href="@Url.Action("Details", "Store",
                new { id = album.AlbumId })">
            <img alt="@album.Title" src="@album.AlbumArtUrl" />
            <span>@album.Title</span>
</a>
        </li>
    }
</ul>
```

Now when we run the application, we'll see our updated home page with top selling albums and our promotional message.

Conclusion

We've seen that ASP.NET MVC makes it easy to create a sophisticated website with database access, membership, AJAX, etc. pretty quickly. Hopefully this tutorial has given you the tools you need to get started building your own ASP.NET MVC applications!

PREVIOUS

MVC 3 - The HTML5 and jQuery UI Datepicker Popup Calendar

2/20/2020 • 2 minutes to read • Edit Online

This tutorial covers how to work with editor templates, display templates, and the jQuery UI datepicker popup calendar in an ASP.NET MVC Web application.

- HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 1
- HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 2
- HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 3
- HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 4

Using the HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 1

2/20/2020 • 4 minutes to read • Edit Online

by Rick Anderson

> This tutorial will teach you the basics of how to work with editor templates, display templates, and the jQuery UI datepicker popup calendar in an ASP.NET MVC Web application.

This tutorial will teach you the basics of how to work with editor templates, display templates, and the jQuery UI datepicker popup calendar in an ASP.NET MVC Web application. For this tutorial, you can use Microsoft Visual Web Developer 2010 Express Service Pack 1 ("Visual Web Developer"), which is a free version of Microsoft Visual Studio, or you can use Visual Studio 2010 SP1 if you already have that.

Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the required software using the following links:

- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites.

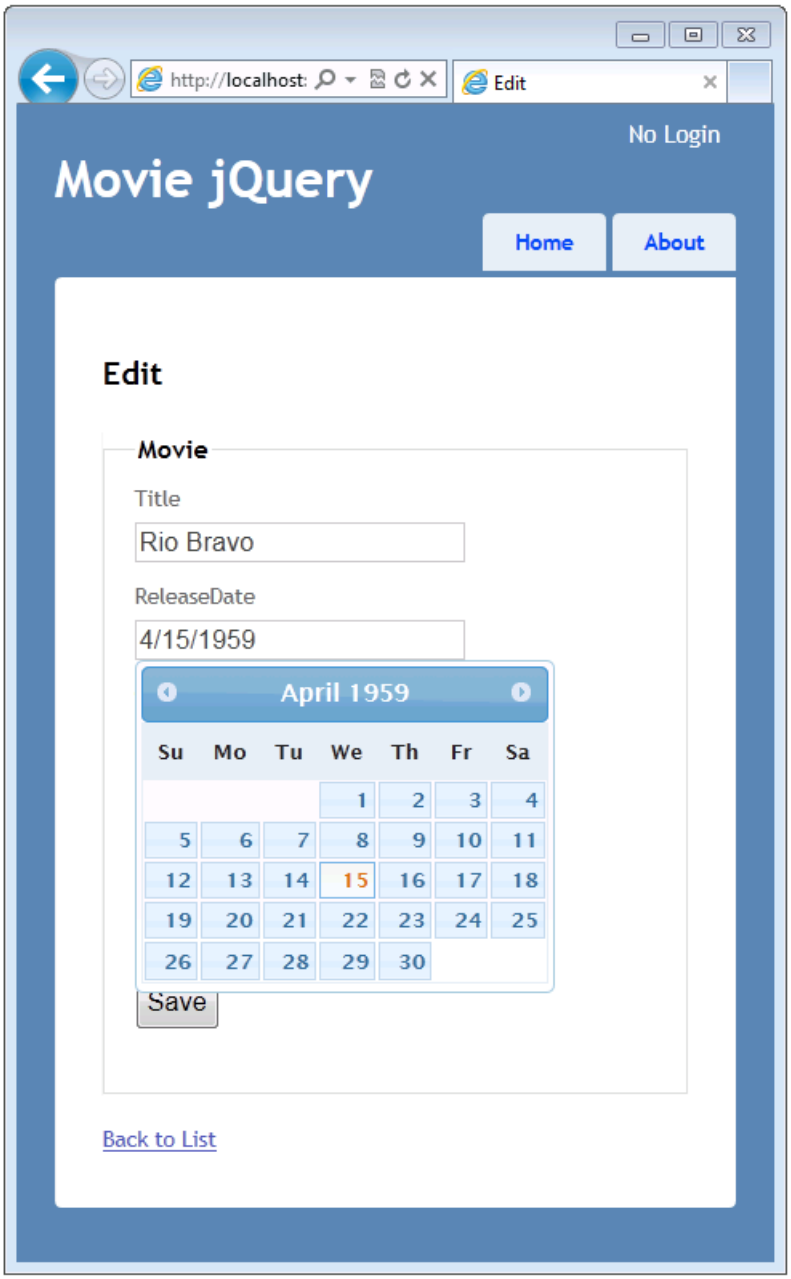
This tutorial assumes you have completed the Getting Started with MVC 3 tutorial or that you're familiar with ASP.NET MVC development. This tutorial starts with the completed project from the Getting Started with MVC 3 tutorial.

This tutorial shows code in C#. However, the starter project and completed project are also available in Visual Basic.

A Visual Studio project with C# and Visual Basic source code is available to accompany this topic: Download.

What You'll Build

You'll add templates (specifically, edit and display templates) to the simple movie-listing application that was created in the Getting Started with MVC 3 tutorial. You will also add a jQuery UI datepicker popup calendar to simplify the process of entering dates. The following screenshot shows the modified application with the jQuery UI datepicker popup calendar displayed.



Skills You'll Learn

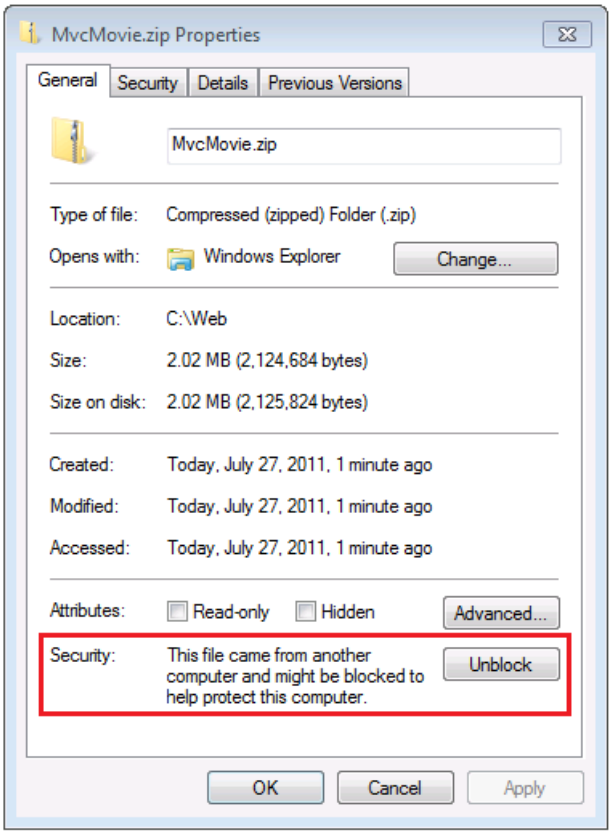
Here's what you'll learn:

- How to use attributes from the DataAnnotations namespace to control the format of data when it's displayed and when it's in edit mode.
- How to create templates (edit and display templates) to control the formatting of data.
- How to add the jQuery UI datepicker as a way to enter date fields.

Getting Started

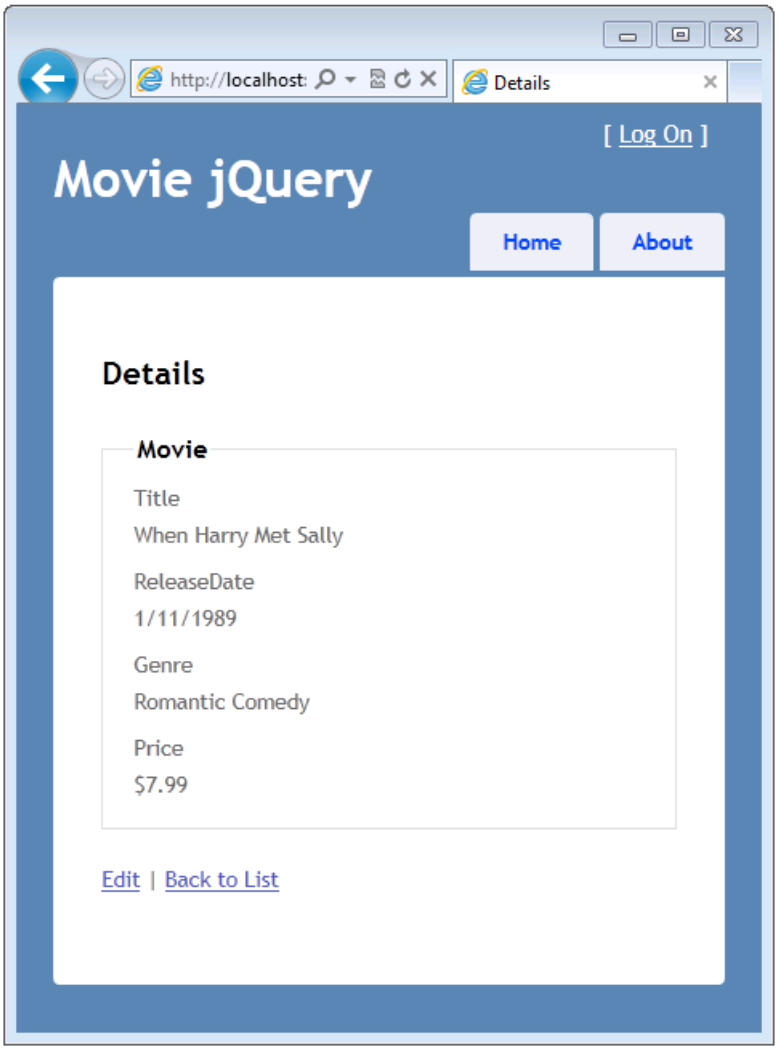
If you don't already have the movie-listing application from the starter project, download it:

- Download.
- In Windows Explorer, right-click the *MvcMovie.zip* file and select **Properties**.
- In the **MvcMovie.zip Properties** dialog box, select **Unblock**. (Unblocking prevents a security warning that occurs when you try to use a *.zip* file that you've downloaded from the web.)



Right-click the *MvcMovie.zip* file and select **Extract All** to unzip the file. In Visual Web Developer or Visual Studio 2010, open the *MvcMovieCS_TU.sln* file.

In **Solution Explorer**, double-click the *Views\Shared_Layout.cshtml* to open it. Change the `H1` header from **MVC Movie App** to **Movie jQuery**. Press CTRL+F5 to run the application and click the **Home** tab, which takes you to the `Index` method of the movie controller. To try out the application, select the **Edit** link and the **Details** link for one of the movies. Notice that in the Index, Edit, and Details views, the release date and price are nicely formatted:



The formatting for the date and the price is the result of using the DisplayFormat attribute on properties of the `Movie` class.

Open the *Movie.cs* file and comment out the `DisplayFormat` attribute on the `ReleaseDate` and `Price` properties. The resulting `Movie` class looks like this:

```
public class Movie {
    public int ID { get; set; }

    [Required(ErrorMessage = "Title is required")]
    public string Title { get; set; }

    //  [DisplayFormat(DataFormatString = "{0:d}")]
    public DateTime ReleaseDate { get; set; }

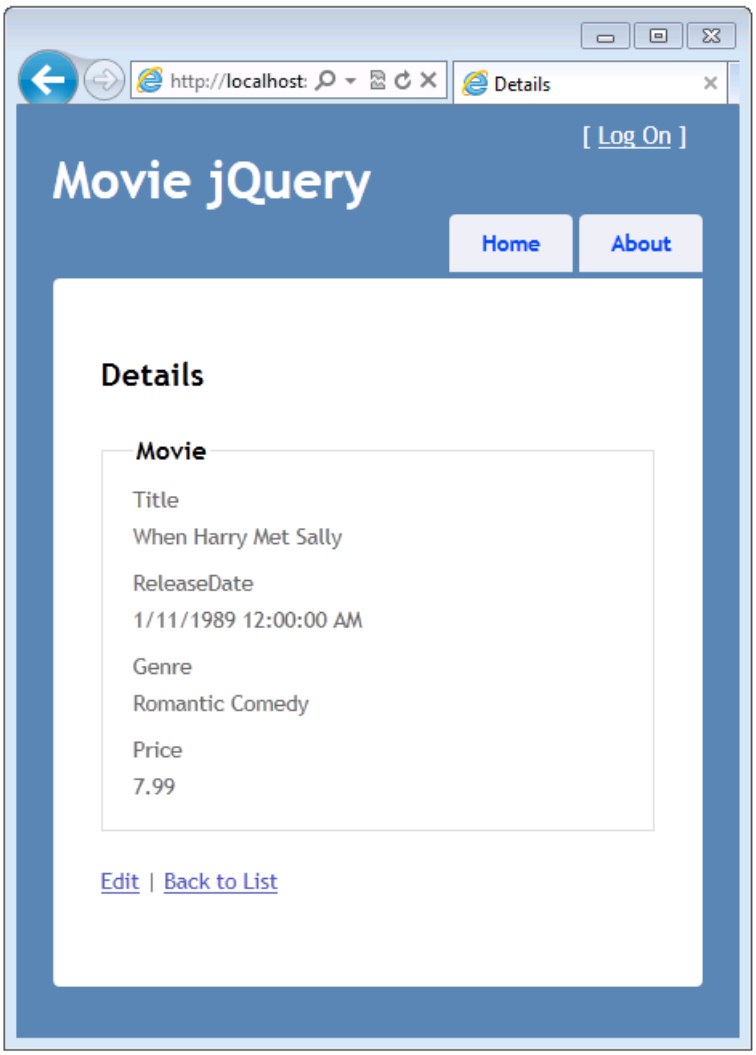
    [Required(ErrorMessage = "Genre must be specified")]
    public string Genre { get; set; }

    [Range(1, 100, ErrorMessage = "Price must be between $1 and $100")]
    //[DisplayFormat(DataFormatString = "{0:c}")]
    public decimal Price { get; set; }

    [StringLength(5)]
    public string Rating { get; set; }
}
```

Press CTRL+F5 again to run the application and select the **Home** tab to view the movie list. This time the release date shows the date and time, and the price field no longer shows the currency symbol. Your change in the `Movie`

class has undone the nice formatting that you saw earlier, but you'll fix that in a moment.



Using the DataAnnotations DataType Attribute to Specify the Data Type

Replace the commented-out `DisplayFormat` attribute for the `ReleaseDate` property with the DataType attribute, using the `Date` enumeration. Replace the `DisplayFormat` attribute for the `Price` property with the DataType attribute again, this time using the `Currency` enumeration. This is what the completed code looks like:

```
public class Movie {
    public int ID { get; set; }

    [Required(ErrorMessage = "Title is required")]
    public string Title { get; set; }

    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Required(ErrorMessage = "Genre must be specified")]
    public string Genre { get; set; }

    [Range(1, 100, ErrorMessage = "Price must be between $1 and $100")]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [StringLength(5)]
    public string Rating { get; set; }
}
```

Run the application. Now the release date and the price properties are formatted correctly (that is, using

appropriate date and currency formats). The DataType attribute provides type metadata for the built-in ASP.NET MVC templates so that the fields render in the correct format. Using the `DataType` attribute is preferable to using the `DisplayFormat` attribute that was originally in the code, because the `DataType` attribute makes the model cleaner and more flexible for purposes like internationalization.

In the next section you'll see how to make custom templates to display date fields.

NEXT

Using the HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 2

2/20/2020 • 6 minutes to read • Edit Online

by Rick Anderson

> This tutorial will teach you the basics of how to work with editor templates, display templates, and the jQuery UI datepicker popup calendar in an ASP.NET MVC Web application.

Adding an Automatic DateTime Template

In the first part of this tutorial, you saw how you can add attributes to the model to explicitly specify formatting, and how you can explicitly specify the template that's used to render the model. For example, the DisplayFormat attribute in the following code explicitly specifies the formatting for the `ReleaseDate` property.

```
[DisplayFormat(DataFormatString = "{0:d}")]
public DateTime  ReleaseDate { get; set; }
```

In the following example, the DataType attribute, using the `Date` enumeration, specifies that the date template should be used to render the model. If there's no date template in your project, the built-in date template is used.

```
[DataType(DataType.Date)]
public DateTime  ReleaseDate { get; set; }
```

However, ASP.MVC can perform type matching using convention-over-configuration, by looking for a template that matches the name of a type. This lets you create a template that automatically formats data without using any attributes or code at all. For this part of the tutorial, you'll create a template that's automatically applied to model properties of type DateTime. You won't need to use an attribute or other configuration to specify that the template should be used to render all model properties of type DateTime.

You'll also learn a way to customize the display of individual properties or even individual fields.

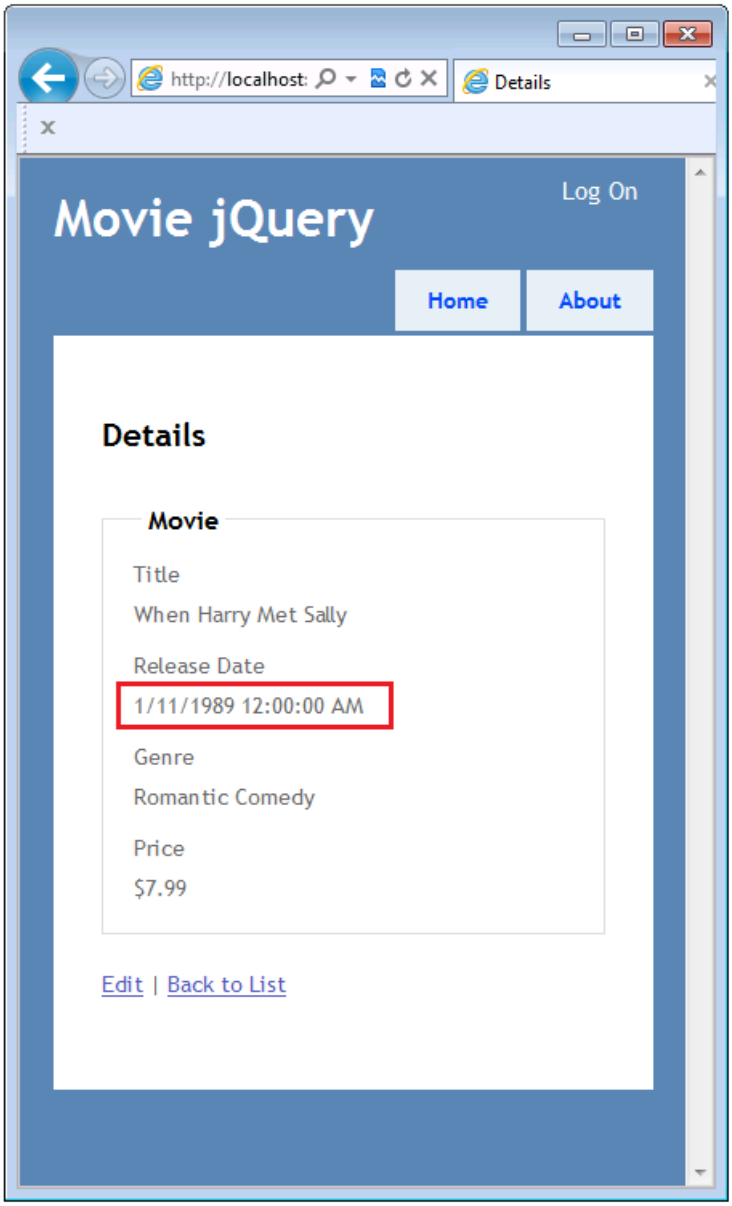
To begin, let's remove existing formatting information and display full dates in the application.

Open the *Movie.cs* file and comment out the `DataType` attribute on the `ReleaseDate` property:

```
// [DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }
```

Press CTRL+F5 to run the application.

Notice that the `ReleaseDate` property now displays both the date and time, because that's the default when no formatting information is provided.



Adding CSS Styles for Testing New Templates

Before you create a template for formatting dates, you'll add a few CSS style rules that you can apply to the new templates. These will help you verify that the rendered page is using the new template.

Open the *Content\Site.cs*s file and add the following CSS rules to the bottom of the file:

```
/* Styles to test new editor and display templates.
-------------------------------------------------------------*/
.loud-1 {
    font-weight: bold;
    color: Red;
}
.loud-2 {
    font-size:  2.0em;
    color:Green;
}
.loud-3 {
    font-style: italic;
    color: yellow;
    background-color: blue;
}
```

Adding DateTime Display Templates

Now you can create the new template. In the *Views\Movies* folder, create a *DisplayTemplates* folder.

In the *Views\Shared* folder, create a *DisplayTemplates* folder and an *EditorTemplates* folder.

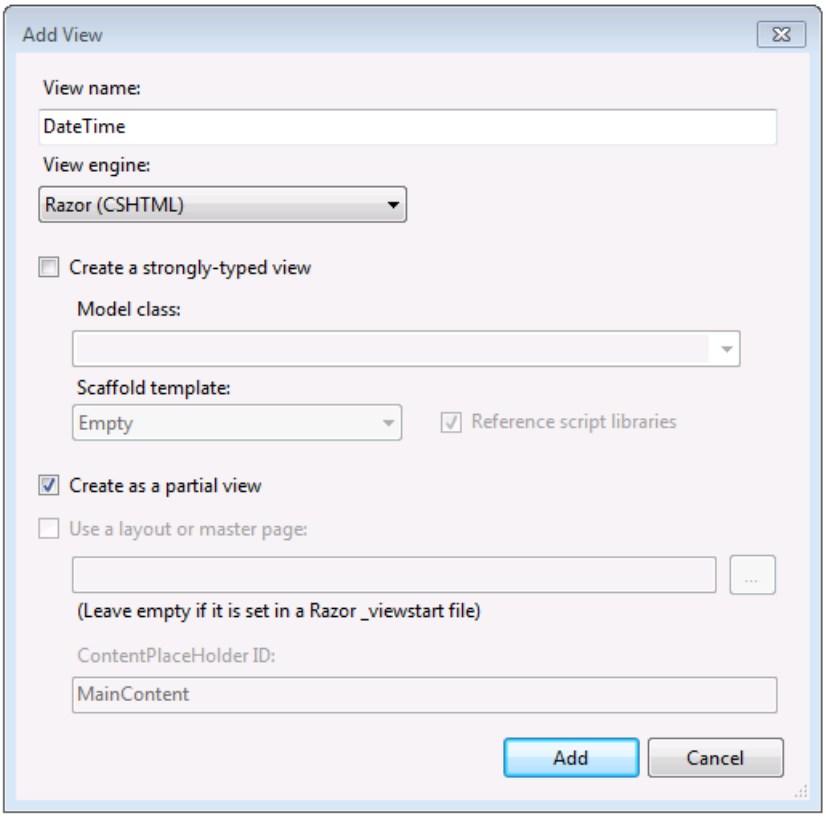
The display templates in the *Views\Shared\DisplayTemplates* folder will be used by all controllers. The display templates in the *Views\Movie\DisplayTemplates* folder will be used only by the `Movie` controller. (If a template with the same name appears in both folders, the template in the *Views\Movie\DisplayTemplates* folder — that is, the more specific template — takes precedence for views returned by the `Movie` controller.)

In **Solution Explorer**, expand the *Views* folder, expand the *Shared* folder, and then right-click the *Views\Shared\DisplayTemplates* folder.

Click **Add** and then click **View**. The **Add View** dialog box is displayed.

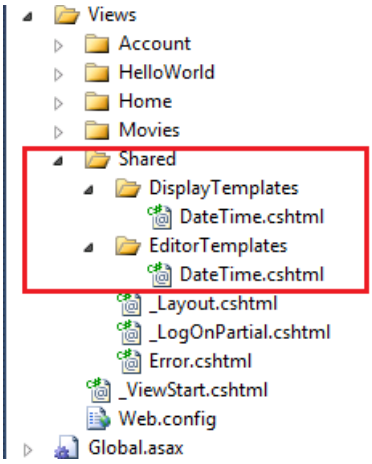
In the **View name** box, type `DateTime`. (You must use this name in order to match the name of the type.)

Select the **Create as a partial view** check box. Make sure that the **Use a layout or master page** and **Create a strongly-typed view** check boxes are not selected.



Click **Add**. A *DateTime.cshtml* template is created in the *Views\Shared\DisplayTemplates*.

The following image shows the *Views* folder in **Solution Explorer** after the `DateTime` display and editor templates are created.



Open the *Views\Shared\DisplayTemplates\DateTime.cshtml* file and add the following markup, which uses the String.Format method to format the property as a date without the time. (The `{0:d}` format specifies short date format.)

```
@String.Format("{0:d}", Model.Date)
```

Repeat this step to create a `DateTime` template in the *Views\Movie\DisplayTemplates* folder. Use the following code in the *Views\Movie\DisplayTemplates\DateTime.cshtml* file.

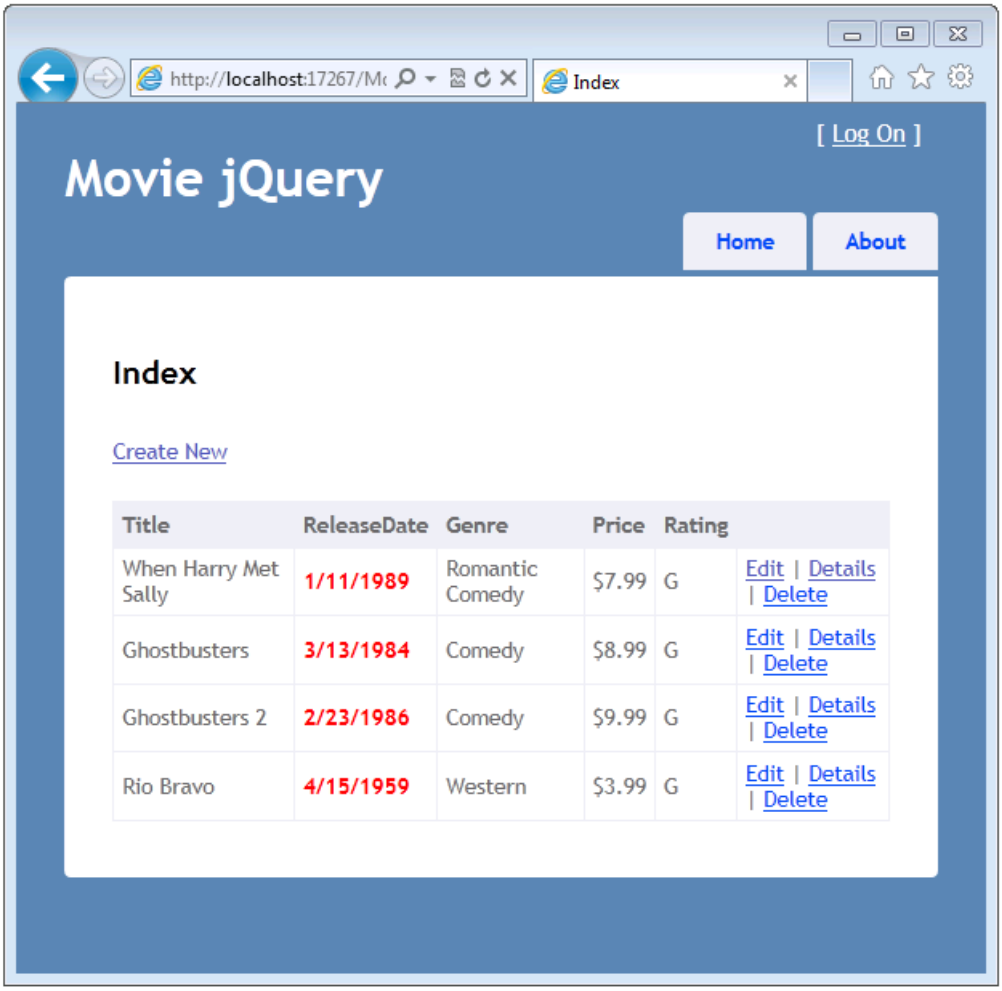
```
<span class="loud-1">
@String.Format("{0:d}", Model.Date)
</span>
```

The `loud-1` CSS class causes the date to be display in bold red text. You added the `loud-1` CSS class just as a temporary measure so you can easily see when this particular template is being used.

What you've done is created and customized templates that ASP.NET will use to display dates. The more general template (in the *Views\Shared\DisplayTemplates* folder) displays a simple short date. The template that's specifically for the `Movie` controller (in the *Views\Movies\DisplayTemplates* folder) displays a short date that's also formatted as bold red text.

Press CTRL+F5 to run the application. The browser renders the Index view for the application.

The `ReleaseDate` property now displays the date in a bold red font without the time.This helps you confirm that the `DateTime` templated helper in the *Views\Movies\DisplayTemplates* folder is selected over the `DateTime` templated helper in the shared folder (*Views\Shared\DisplayTemplates*).



Now rename the *Views\Movies\DisplayTemplates\DateTime.cshtml* file to *Views\Movies\DisplayTemplates\LoudDateTime.cshtml*.

Press CTRL+F5 to run the application.

This time the `ReleaseDate` property displays a date without the time and without the bold red font. This illustrates that a template that has the name of the data type (in this case `DateTime`) is automatically used to display all model properties of that type. After you renamed the *DateTime.cshtml* file to *LoudDateTime.cshtml*, ASP.NET no longer found a template in the *Views\Movies\DisplayTemplates* folder, so it used the *DateTime.cshtml* template from the *Views\Movies\Shared* folder.

(The template matching is case insensitive, so you could have created the template file name with any casing. For example, *DATETIME.cshtml, datetime.cshtml*, and *DaTeTiMe.cshtml* would all match the `DateTime` type.)

To review: at this point, the `ReleaseDate` field is being displayed using the *Views\Movies\DisplayTemplates\DateTime.cshtml* template, which displays the data using a short date format, but otherwise adds no special format.

Using UIHint to Specify a Display Template

If your web application has many `DateTime` fields and by default you want to display all or most of them in date-only format, the *DateTime.cshtml* template is a good approach. But what if you have a few dates where you want to display the full date and time? No problem. You can create an additional template and use the UIHint attribute to specify formatting for the full date and time. You can then selectively apply that template. You can use the UIHint attribute at the model level or you can specify the template inside a view. In this section, you'll see how to use the `UIHint` attribute to selectively change the formatting for some instances of date-time fields.

Open the *Views\Movies\DisplayTemplates\LoudDateTime.cshtml* file and replace the existing code with the following:

```
<span class="loud-2">
@Model.ToString()
</span>
```

This causes the full date and time to be displayed and adds the CSS class that makes the text green and large.

Open the *Movie.cs* file and add the UIHint attribute to the `ReleaseDate` property, as shown in the following example:

```
[UIHint("LoudDateTime")]
public DateTime ReleaseDate { get; set; }
```

This tells ASP.NET MVC that when it displays the `ReleaseDate` property (specifically, and not just any `DateTime` object), it should use the *LoudDateTime.cshtml* template.

Press CTRL+F5 to run the application.

Notice that the `ReleaseDate` property now displays the date and time in a large green font.

Return to the `UIHint` attribute in the *Movie.cs* file and comment it out so the *LoudDateTime.cshtml* template won't be used. Run the application again. The release date is not displayed large and green. This verifies that the *Views\Shared\DisplayTemplates\DateTime.cshtml* template is used in the Index and Details views.

As mentioned earlier, you can also apply a template in a view, which lets you apply the template to an individual instance of some data. Open the *Views\Movies\Details.cshtml* view. Add `"LoudDateTime"` as the second parameter of the Html.DisplayFor call for the `ReleaseDate` field. The completed code looks like this:

```
<div class="display-label">ReleaseDate</div>
<div class="display-field">
    @Html.DisplayFor(model => model.ReleaseDate,"LoudDateTime")
</div>
```

This specifies that the `LoudDateTime` template should be used to display the model property, regardless of what attributes are applied to the model.

Press CTRL+F5 to run the application.

Verify that the movie index page is using the *Views\Shared\DisplayTemplates\DateTime.cshtml* template (red bold) and the *Movie\Details* page is using the *Views\Movies\DisplayTemplates\LoudDateTime.cshtml* template (large and green).

In the next section, you'll create a template for a complex type.

PREVIOUS NEXT

Using the HTML5 and jQuery UI Datepicker Popup Calendar with ASP.NET MVC - Part 4

2/20/2020 • 9 minutes to read • Edit Online

by Rick Anderson

> This tutorial will teach you the basics of how to work with editor templates, display templates, and the jQuery UI datepicker popup calendar in an ASP.NET MVC Web application.

Adding a Template for Editing Dates

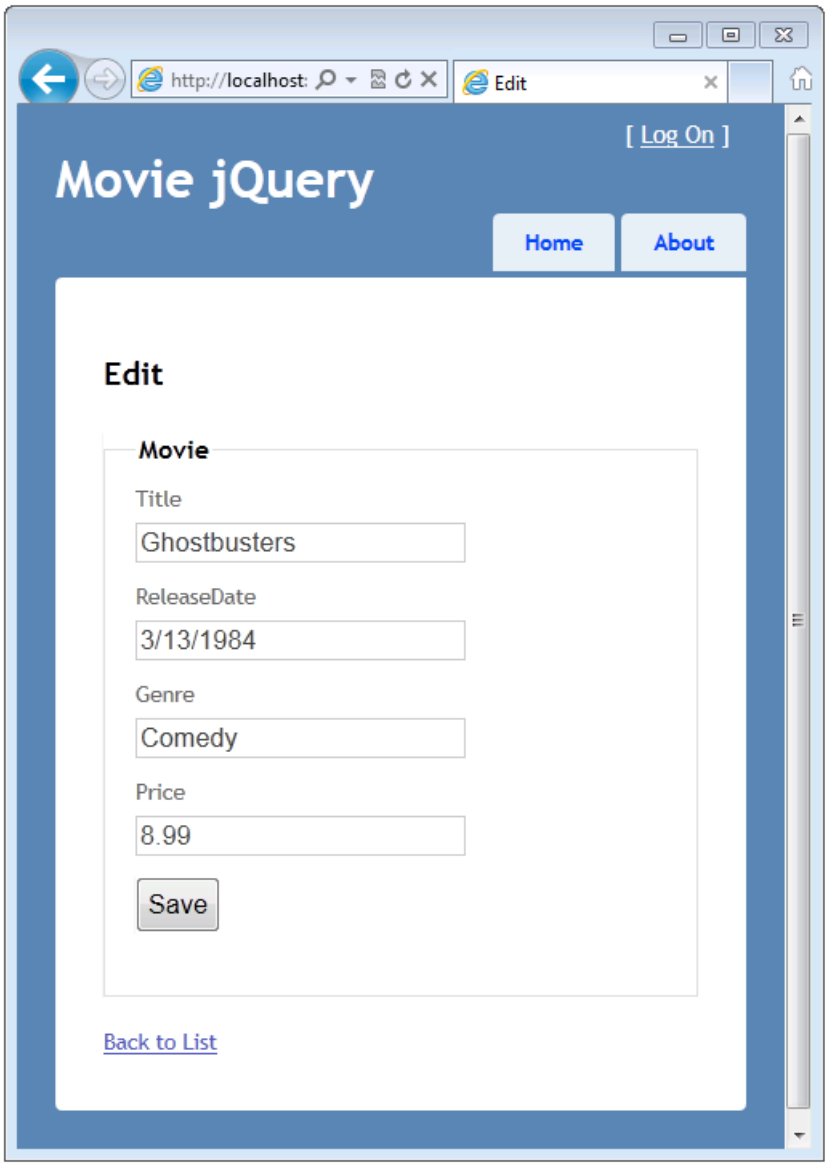
In this section you'll create a template for editing dates that will be applied when ASP.NET MVC displays UI for editing model properties that are marked with the **Date** enumeration of the DataType attribute. The template will render only the date; time will not be displayed. In the template you'll use the jQuery UI Datepicker popup calendar to provide a way to edit dates.

To begin, open the *Movie.cs* file and add the DataType attribute with the **Date** enumeration to the `ReleaseDate` property, as shown in the following code:

```
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }
```

This code causes the `ReleaseDate` field to be displayed without the time in both display templates and edit templates. If your application contains a *date.cshtml* template in the *Views\Shared\EditorTemplates* folder or in the *Views\Movies\EditorTemplates* folder, that template will be used to render any `DateTime` property while editing. Otherwise the built-in ASP.NET templating system will display the property as a date.

Press CTRL+F5 to run the application. Select an edit link to verify that the input field for the release date is showing only the date.



In **Solution Explorer**, expand the *Views* folder, expand the *Shared* folder, and then right-click the *Views\Shared\EditorTemplates* folder.

Click **Add**, and then click **View**. The **Add View** dialog box is displayed.

In the **View name** box, type "Date".

Select the **Create as a partial view** check box. Make sure that the **Use a layout or master page** and **Create a strongly-typed view** check boxes are not selected.

Click **Add**. The *Views\Shared\EditorTemplates\Date.cshtml* template is created.

Add the following code to the *Views\Shared\EditorTemplates\Date.cshtml* template.

```
@model DateTime
Using Date Template
@Html.TextBox("", String.Format("{0:d}", Model.ToShortDateString()),
  new { @class = "datefield", type = "date"  })
```

The first line declares the model to be a `DateTime` type. Although you don't need to declare the model type in edit and display templates, it's a best practice so that you get compile-time checking of the model being passed to the view. (Another benefit is that you then get IntelliSense for the model in the view in Visual Studio.) If the model type is not declared, ASP.NET MVC considers it a dynamic type and there's no compile-time type checking. If you declare

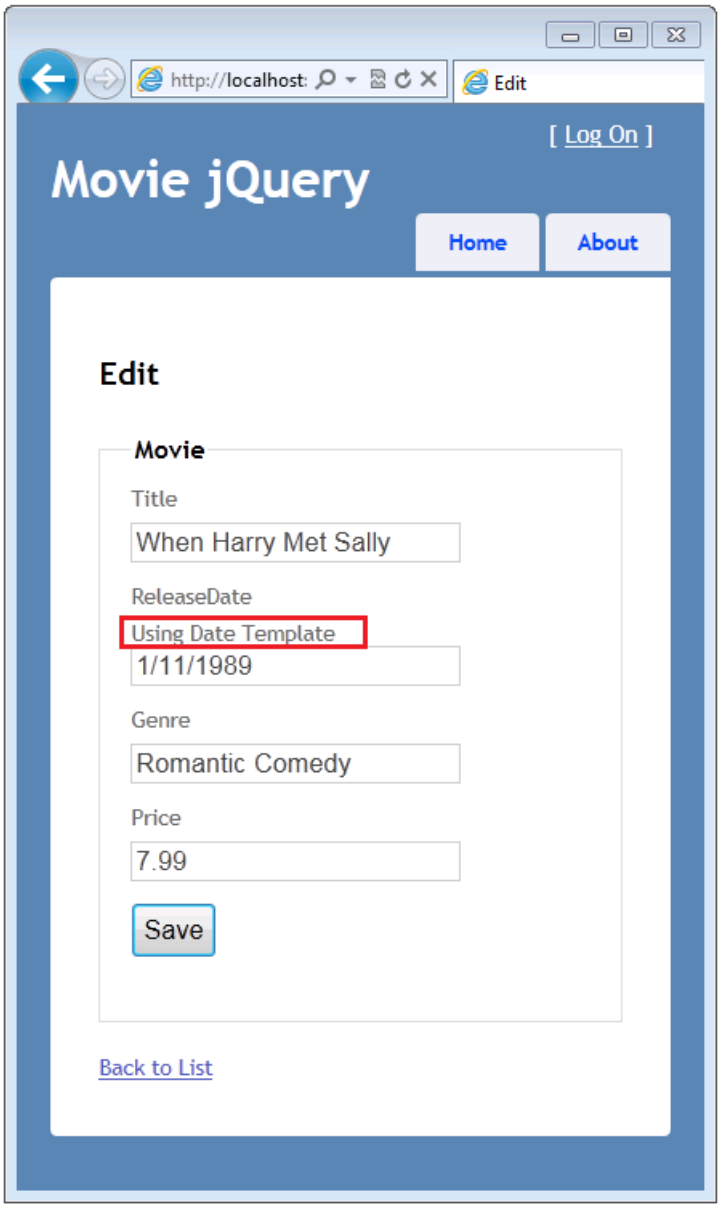
the model to be a `DateTime` type, it becomes strongly typed.

The second line is just literal HTML markup that displays "Using Date Template" before a date field. You'll use this line temporarily to verify that this date template is being used.

The next line is an Html.TextBox helper that renders an `input` field that's a text box. The third parameter for the helper uses an anonymous type to set the class for the text box to `datefield` and the type to `date`. (Because `class` is a reserved in C#, you need to use the `@` character to escape the `class` attribute in the C# parser.)

The `date` type is an HTML5 input type that enables HTML5-aware browsers to render a HTML5 calendar control. Later on you'll add some JavaScript to hook up the jQuery datepicker to the `Html.TextBox` element using the `datefield` class.

Press CTRL+F5 to run the application. You can verify that the `ReleaseDate` property in the edit view is using the edit template because the template displays "Using Date Template" just before the `ReleaseDate` text input box, as shown in this image:



In your browser, view the source of the page. (For example, right-click the page and select **View source**.) The following example shows some of the markup for the page, illustrating the `class` and `type` attributes in the rendered HTML.

```
<input class="datefield" data-val="true" data-val-required="Date is required"
      id="ReleaseDate" name="ReleaseDate" type="date" value="1/11/1989" />
```

Return to the *Views\Shared\EditorTemplates\Date.cshtml* template and remove the "Using Date Template" markup. Now the completed template looks like this:

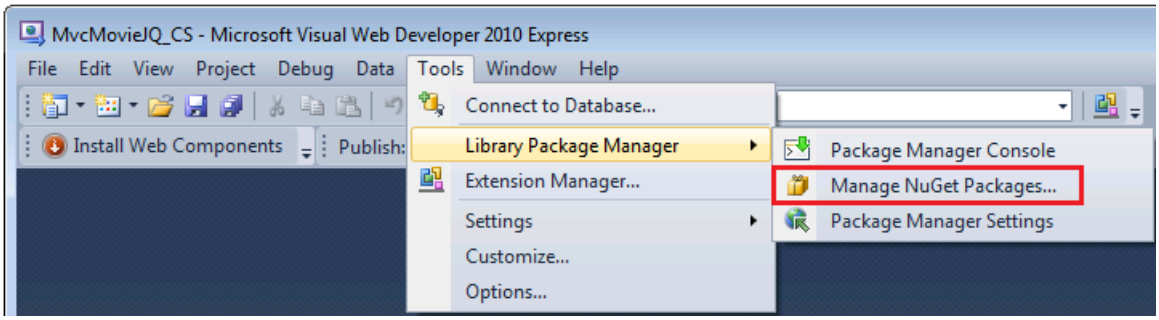
```
@model DateTime
@Html.TextBox("", String.Format("{0:d}", Model.ToShortDateString()),
  new { @class = "datefield", type = "date" })
```

Adding a jQuery UI Datepicker Popup Calendar using NuGet

In this section you'll add the jQuery UI datepicker popup calendar to the date-edit template. The jQuery UI library provides support for animation, advanced effects, and customizable widgets. It's built on top of the jQuery JavaScript library. The datepicker popup calendar makes it easy and natural to enter dates using a calendar instead of entering a string. The popup calendar also limits users to legal dates — ordinary text entry for a date would let you enter something like `2/33/1999` (February 33rd, 1999), but the jQuery UI datepicker popup calendar won't allow that.

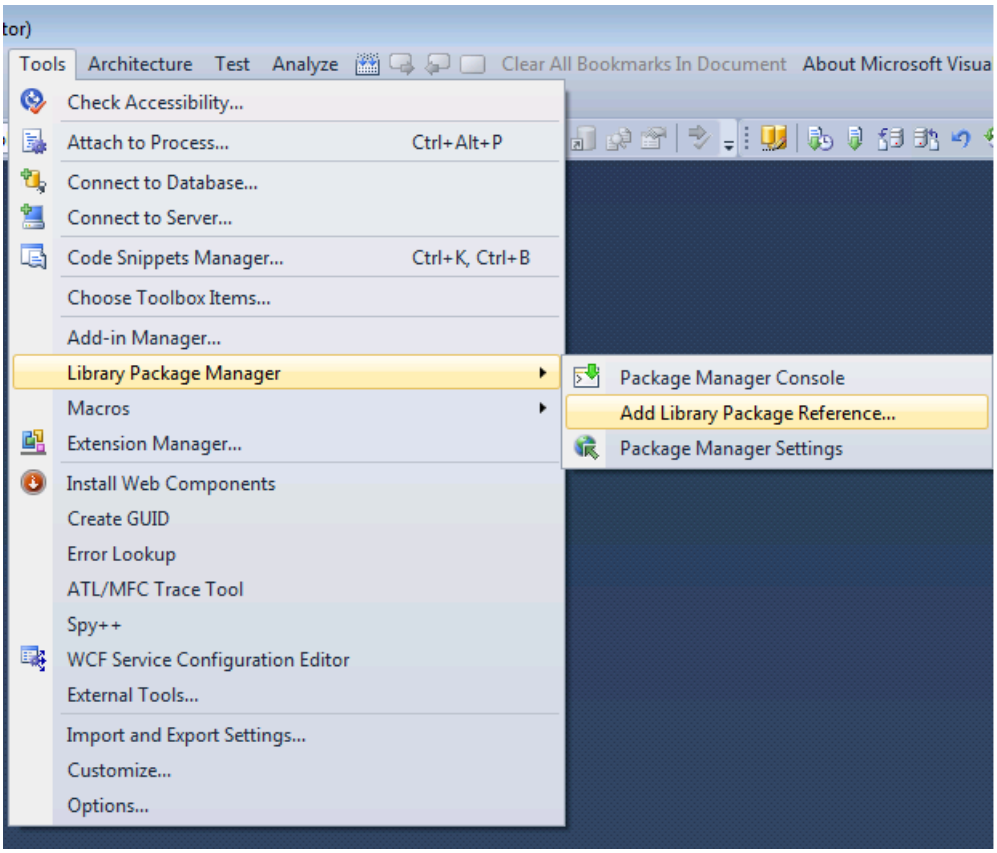
First, you have to install the jQuery UI libraries. To do that, you'll use NuGet, which is a package manager that's included in SP1 versions of Visual Studio 2010 and Visual Web Developer.

In Visual Web Developer, from the **Tools** menu, select **NuGet Package Manager** and then select **Manage NuGet Packages**.

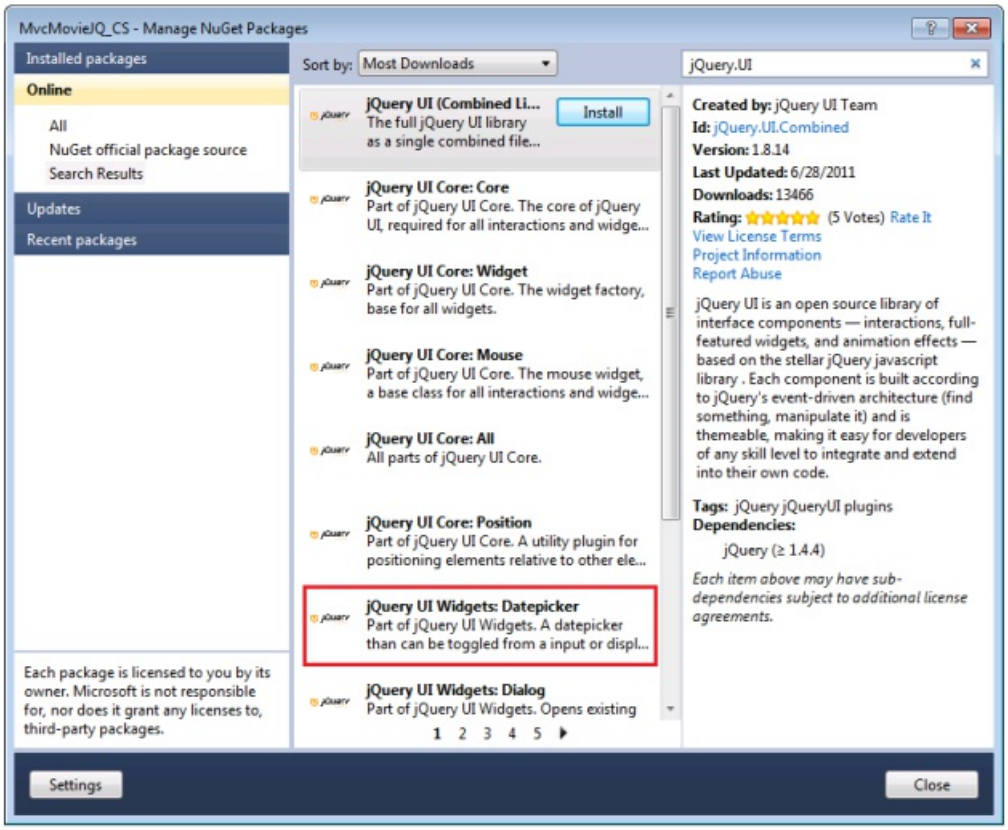


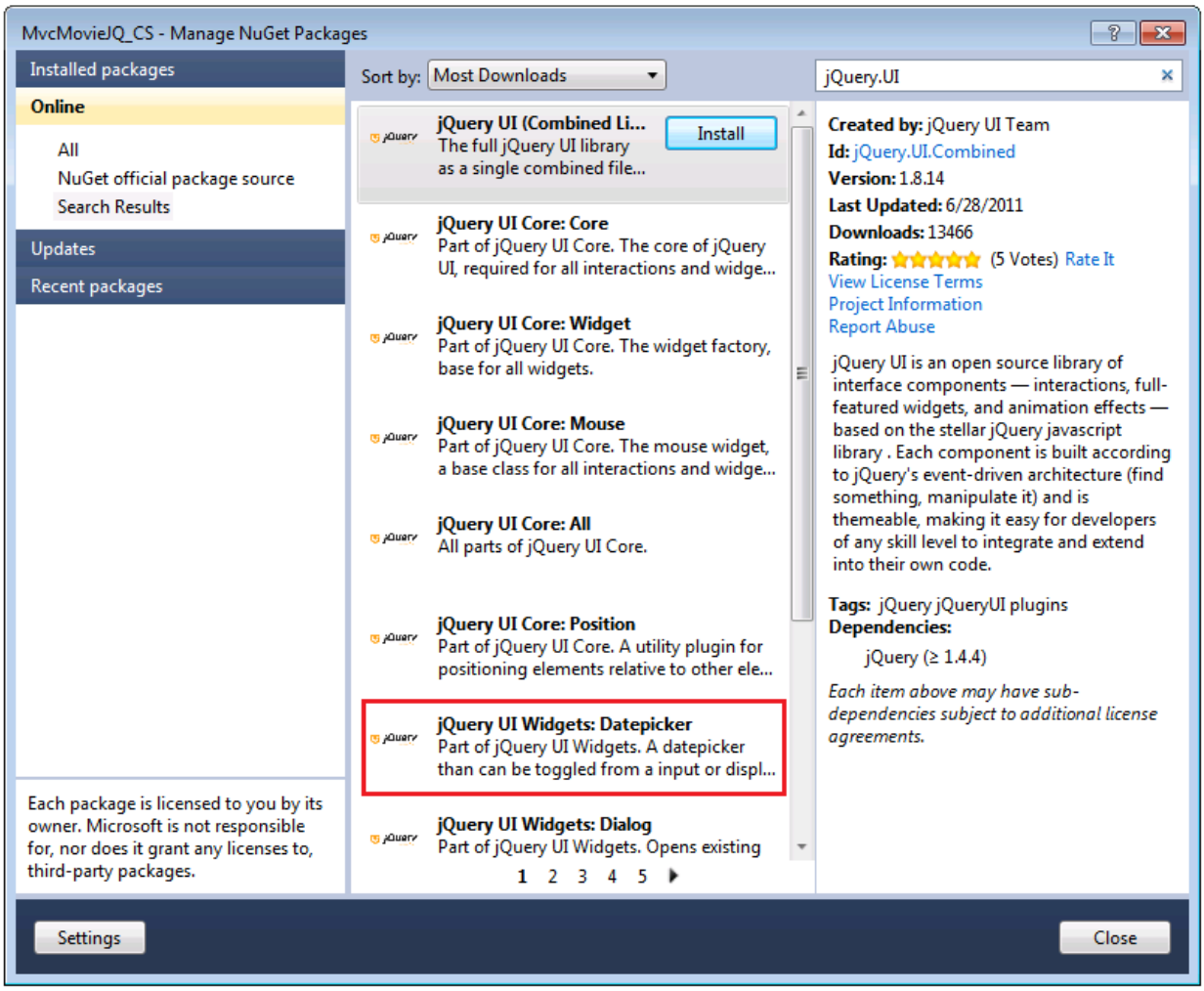
Note: If the **Tools** menu doesn't display the **NuGet Package Manager** command, you need to install NuGet by following the instructions on the Installing NuGet page of the NuGet website.

If you're using Visual Studio instead of Visual Web Developer, from the **Tools** menu, select **NuGet Package Manager** and then select **Add Library Package Reference**.



In the **MVCMovie - Manage NuGet Packages** dialog box, click the **Online** tab on the left and then enter "jQuery.UI" in the search box. Select j **Query UI Widgets:Datepicker**, then select the **Install** button.





NuGet adds these debug versions and minified versions of jQuery UI Core and the jQuery UI date picker to your project:

- *jquery.ui.core.js*
- *jquery.ui.core.min.js*
- *jquery.ui.datepicker.js*
- *jquery.ui.datepicker.min.js*

Note: The debug versions (the files without the *.min.js* extension) are useful for debugging, but in a production site, you'd include only the minified versions.

To actually use the jQuery date picker, you need to create a jQuery script that will hook up the calendar widget to the edit template. In **Solution Explorer**, right-click the *Scripts* folder and select **Add**, then **New Item**, and then **JScript File**. Name the file *DatePickerReady.js*.

Add the following code to the *DatePickerReady.js* file:

```
$(function () {
    $(".datefield").datepicker();
});
```

If you're not familiar with jQuery, here's a brief explanation of what this does: the first line is the "jQuery ready" function, which is called when all the DOM elements in a page have loaded. The second line selects all DOM elements that have the class name `datefield`, then invokes the `datepicker` function for each of them. (Remember that you added the `datefield` class to the *Views\Shared\EditorTemplates\Date.cshtml* template earlier in the tutorial.)

Next, open the *Views\Shared_Layout.cshtml* file. You need to add references to the following files, which are all required so that you can use the date picker:

- *Content/themes/base/jquery.ui.core.css*
- *Content/themes/base/jquery.ui.datepicker.css*
- *Content/themes/base/jquery.ui.theme.css*
- *jquery.ui.core.min.js*
- *jquery.ui.datepicker.min.js*
- *DatePickerReady.js*

The following example shows the actual code that you should add at the bottom of the `head` element in the *Views\Shared_Layout.cshtml* file.

```
<link href="@Url.Content("~/Content/themes/base/jquery.ui.core.css")"
        rel="stylesheet" type="text/css" />
    <link href="@Url.Content("~/Content/themes/base/jquery.ui.datepicker.css")"
        rel="stylesheet"  type="text/css" />
    <link href="@Url.Content("~/Content/themes/base/jquery.ui.theme.css")"
        rel="stylesheet" type="text/css" />

    <script src="@Url.Content("~/Scripts/jquery.ui.core.min.js")"
        type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/jquery.ui.datepicker.min.js")"
        type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/DatePickerReady.js")"
        type="text/javascript"></script>
```

The complete `head` section is shown here:

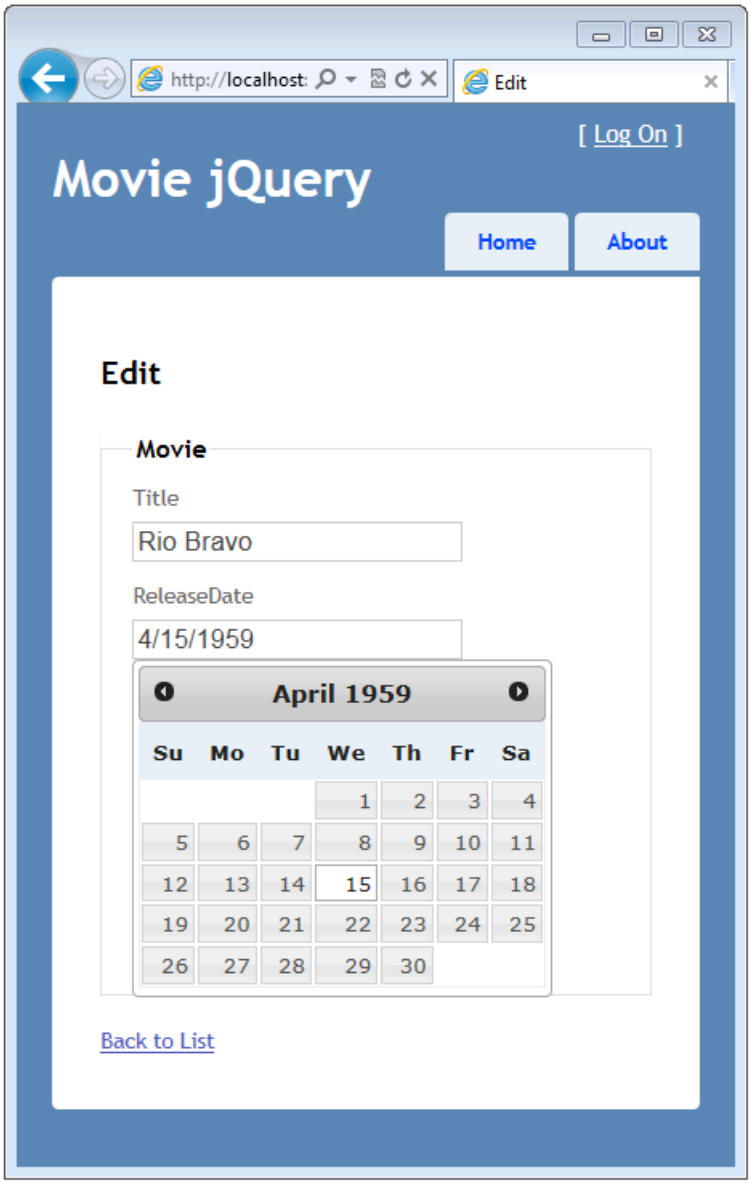
```
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")"
        rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"
        type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/modernizr-1.7.min.js")"
        type="text/javascript"></script>

    <link href="@Url.Content("~/Content/themes/base/jquery.ui.core.css")"
        rel="stylesheet" type="text/css" />
    <link href="@Url.Content("~/Content/themes/base/jquery.ui.datepicker.css")"
        rel="stylesheet"  type="text/css" />
    <link href="@Url.Content("~/Content/themes/base/jquery.ui.theme.css")"
        rel="stylesheet" type="text/css" />

    <script src="@Url.Content("~/Scripts/jquery.ui.core.min.js")"
        type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/jquery.ui.datepicker.min.js")"
        type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/DatePickerReady.js")"
        type="text/javascript"></script>
</head>
```

The URL content helper method converts the resource path to an absolute path. You must use `@URL.Content` to correctly reference these resources when the application is running on IIS.

Press CTRL+F5 to run the application. Select an edit link, then put the insertion point into the **ReleaseDate** field. The jQuery UI popup calendar is displayed.



Like most jQuery controls, the datepicker lets you customize it extensively. For information, see Visual Customization: Designing a jQuery UI theme on the jQuery UI site.

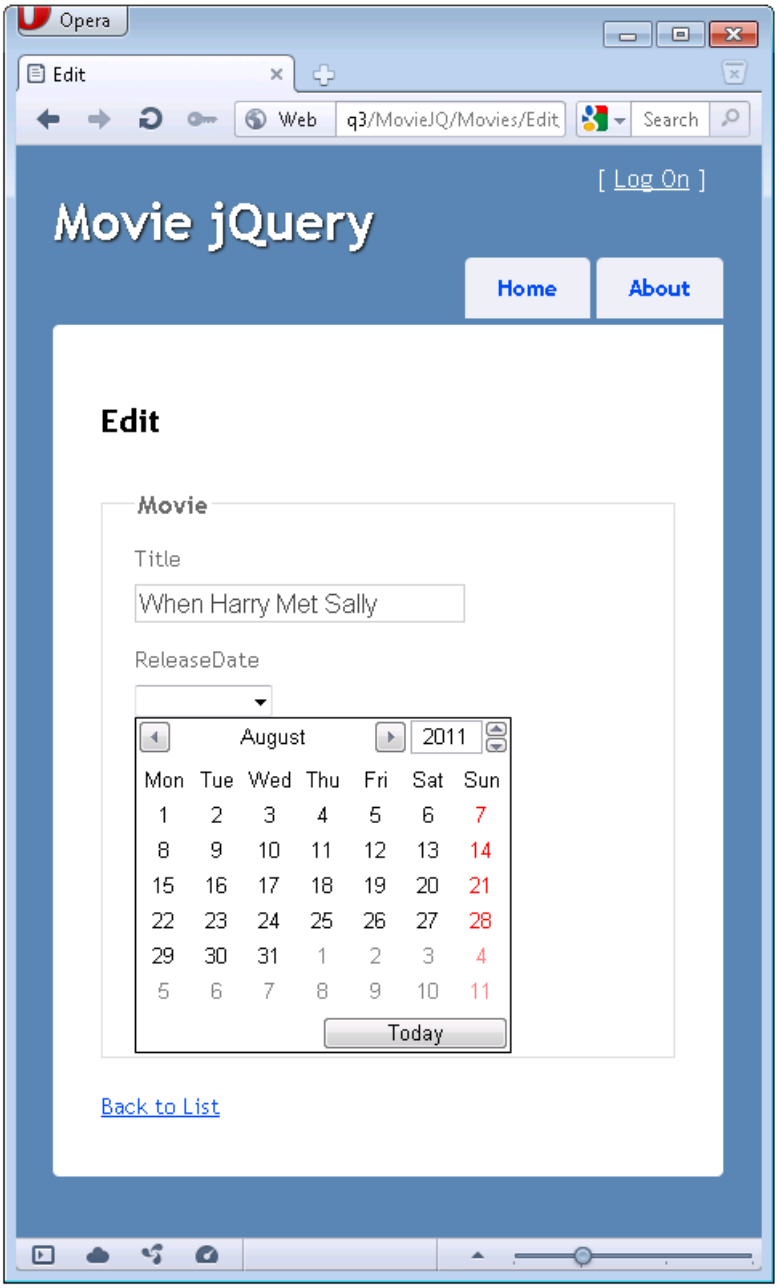
Supporting the HTML5 Date Input Control

As more browsers support HTML5, you'll want to use the native HTML5 input, such as the `date` input element, and not use the jQuery UI calendar. You can add logic to your application to automatically use HTML5 controls if the browser supports them. To do this, replace the contents of the *DatePickerReady.js* file with the following:

```
if (!Modernizr.inputtypes.date) {
    $(function () {
        $(".datefield").datepicker();
    });
}
```

The first line of this script uses Modernizr to verify that HTML5 date input is supported. If it's not supported, the jQuery UI date picker is hooked up instead. (Modernizr is an open-source JavaScript library that detects the availability of native implementations of HTML5 and CSS3. Modernizr is included in any new ASP.NET MVC projects that you create.)

After you've made this change, you can test it by using a browser that supports HTML5, such as Opera 11. Run the application using an HTML5-compatible browser and edit a movie entry. The HTML5 date control is used instead of the jQuery UI popup calendar:



Because new versions of browsers are implementing HTML5 incrementally, a good approach for now is to add code to your website that accommodates a wide variety of HTML5 support. For example, a more robust *DatePickerReady.js* script is shown below that lets your site support browsers that only partially support the HTML5 date control.

```
if (!Modernizr.inputtypes.date) {
    $(function () {
        $("input[type='date']")
                    .datepicker()
                    .get(0)
                    .setAttribute("type", "text");
    })
}
```

This script selects HTML5 `input` elements of type `date` that don't fully support the HTML5 date control. For those elements, it hooks up the jQuery UI popup calendar and then changes the `type` attribute from `date` to `text`. By changing the `type` attribute from `date` to `text`, partial HTML5 date support is eliminated. An even more robust *DatePickerReady.js* script can be found at JSFIDDLE.

Adding Nullable Dates to the Templates

If you use one of the existing date templates and pass a null date, you'll get a run-time error. To make the date templates more robust, you'll change them to handle null values. To support nullable dates, change the code in the *Views\Shared\DisplayTemplates\DateTime.cshtml* to the following:

```
@model Nullable<DateTime>
@(Model != null ? string.Format("{0:d}", Model) : string.Empty)
```

The code returns an empty string when the model is **null**.

Change the code in the *Views\Shared\EditorTemplates\Date.cshtml* file to the following:

```
@model Nullable<DateTime>

 @{
    DateTime dt = DateTime.Now;
    if (Model != null)
    {
       dt  = (System.DateTime) Model;

    }
    @Html.TextBox("", String.Format("{0:d}", dt.ToShortDateString()), new { @class = "datefield", type =
"date"  })
}
```

When this code runs, if the model is not null, the model's `DateTime` value is used. If the model is null, the current date is used instead.

Wrapup

This tutorial has covered the basics of ASP.NET templated helpers and shows you how to use the jQuery UI datepicker popup calendar in an ASP.NET MVC application. For more information, try these resources:

- For information on localization, see Rajeesh's blog JQueryUI Datepicker in ASP.NET MVC.
- For information about jQuery UI, see jQuery UI.
- For information about how to localize the datepicker control, see UI/Datepicker/Localization.
- For more information about the ASP.NET MVC templates, see Brad Wilson's blog series on ASP.NET MVC 2 Templates. Although the series was written for ASP.NET MVC 2, the material still applies for the current version of ASP.NET MVC.

PREVIOUS

MVC 3 - Working with the DropDownList Box and jQuery

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows you how to use the DropDownList Box with jQuery

- Using the DropDownList Helper with ASP.NET MVC
- Examining how ASP.NET MVC scaffolds the DropDownList Helper
- Adding a New Category to the DropDownList using jQuery UI

Using the DropDownList Helper with ASP.NET MVC

2/20/2020 • 12 minutes to read • Edit Online

by Rick Anderson

This tutorial will teach you the basics of working with the DropDownList helper and the ListBox helper in an ASP.NET MVC Web application. You can use Microsoft Visual Web Developer 2010 Express Service Pack 1, which is a free version of Microsoft Visual Studio to follow the tutorial. Before you start, make sure you've installed the prerequisites listed below. You can install all of them by clicking the following link: Web Platform Installer. Alternatively, you can individually install the prerequisites using the following links:

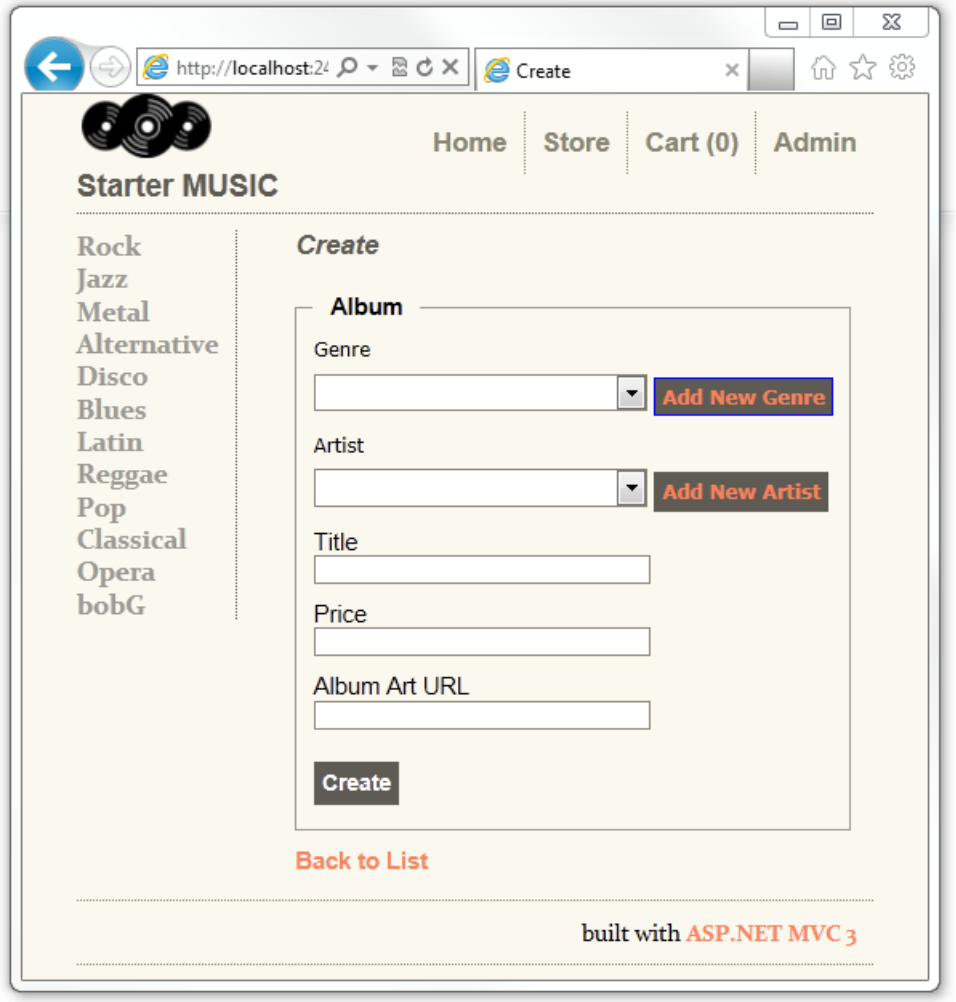
- Visual Studio Web Developer Express SP1 prerequisites
- ASP.NET MVC 3 Tools Update
- SQL Server Compact 4.0(runtime + tools support)

If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites by clicking the following link: Visual Studio 2010 prerequisites. This tutorial assumes you have completed the Intro to ASP.NET MVC tutorial or theASP.NET MVC Music Store tutorial or you are familiar with ASP.NET MVC development. This tutorial starts with a modified project from the ASP.NET MVC Music Store tutorial. You can download the starter project with the following link Download the C# version.

A Visual Web Developer project with the completed tutorial C# source code is available to accompany this topic. Download.

What You'll Build

You'll create action methods and views that use the DropDownList helper to select a category. You will also use `jQuery` to add an insert category dialog that can be used when a new category (such as genre or artist) is needed. Below is a screenshot of the Create view showing links to add a new genre and add a new artist.



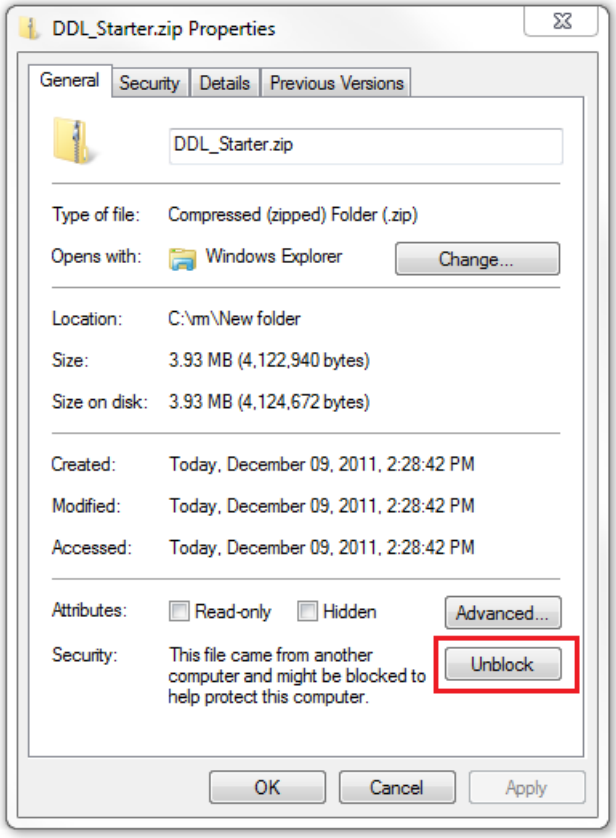
Skills You'll Learn

Here's what you'll learn:

- How to use the DropDownList helper to select category data.
- How to add a **jQuery** dialog to add new categories.

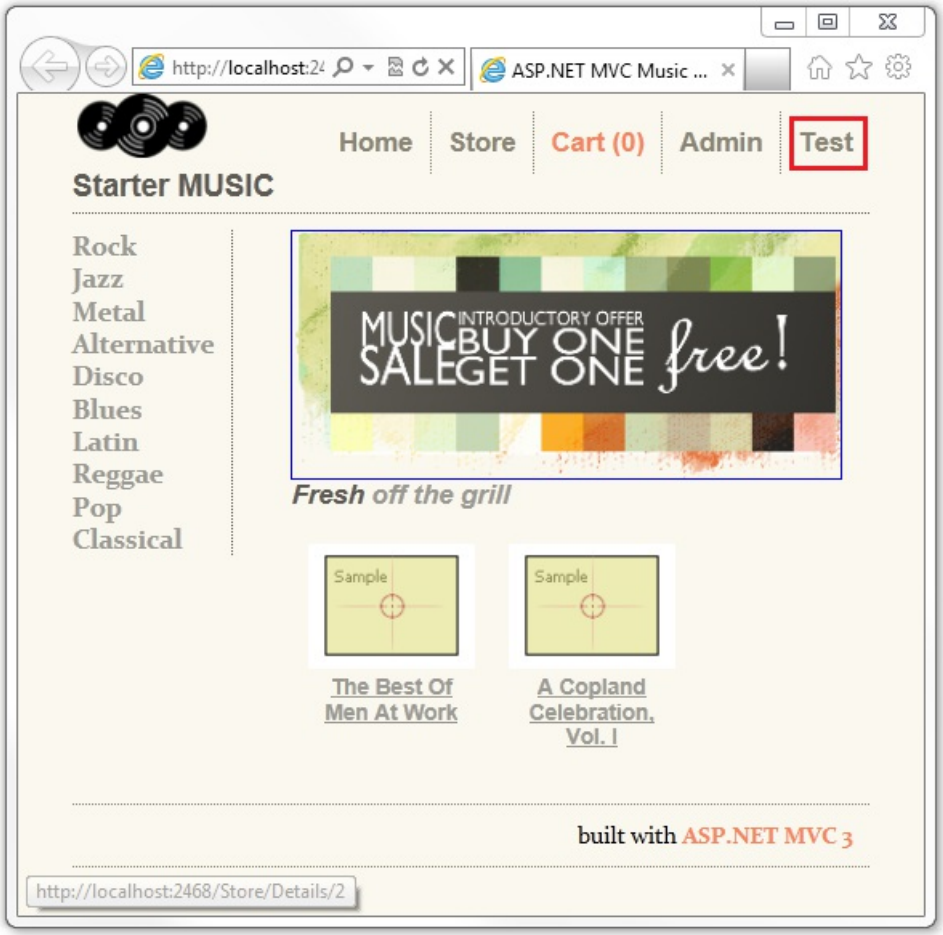
Getting Started

Start by downloading the starter project with the following link, Download. In Windows Explorer, right click on the *DDL_Starter.zip* file and select properties. In the **DDL_Starter.zip Properties** dialog box, select Unblock.



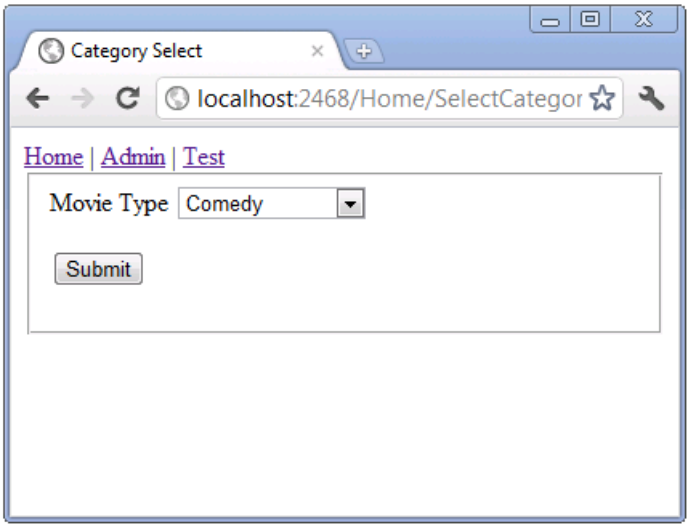
Right click the DDL_Starter.zip file and select **Extract All** to unzip the file. Open the *StartMusicStore.sln* file with Visual Web Developer 2010 Express ("Visual Web Developer" or "VWD" for short) or Visual Studio 2010.

Press CTRL+F5 to run the application and click the **Test** link.



Select the **Select Movie Category (Simple)** link. A Movie Type Select list is displayed, with Comedy the

selected value.



Right click in the browser and select view source. The HTML for the page is displayed. The code below shows the HTML for the select element.

```
<form action="/Home/CategoryChosen" method="get">

<fieldset>Movie Type <select id="MovieType" name="MovieType">

<option value=""></option>

<option value="0">Action</option>

<option value="1">Drama</option>

<option selected="selected" value="2">Comedy</option>

<option value="3">Science Fiction</option>

</select>

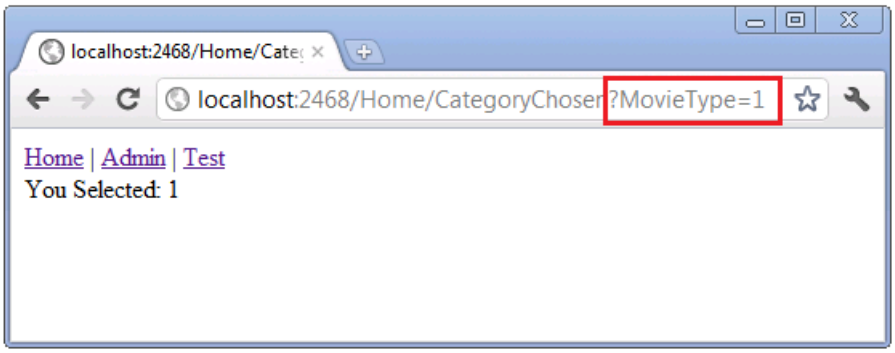
<p><input type="submit" value="Submit" /> </p>

</fieldset>

</form>
```

You can see that each item in the select list has a value (0 for Action, 1 for Drama, 2 for Comedy and 3 for Science Fiction) and a display name (Action, Drama, Comedy and Science Fiction). The code above is standard HTML for a select list.

Change the select list to Drama and hit the **Submit** button. The URL in the browser is `http://localhost:2468/Home/CategoryChosen?MovieType=1` and the page displays **You Selected: 1**.



Open the *Controllers\HomeController.cs* file and examine the `SelectCategory` method.

```
public ActionResult SelectCategory() {

     List<SelectListItem> items = new List<SelectListItem>();

     items.Add(new SelectListItem { Text = "Action", Value = "0"});

     items.Add(new SelectListItem { Text = "Drama", Value = "1" });

     items.Add(new SelectListItem { Text = "Comedy", Value = "2", Selected = true });

     items.Add(new SelectListItem { Text = "Science Fiction", Value = "3" });

     ViewBag.MovieType = items;

     return View();

 }
```

The DropDownList helper used to create an HTML select list requires a **IEnumerable<SelectListItem >**, either explicitly or implicitly. That is, you can pass the **IEnumerable<SelectListItem >** explicitly to the DropDownList helper or you can add the **IEnumerable<SelectListItem >** to the ViewBag using the same name for the **SelectListItem** as the model property. Passing in the **SelectListItem** implicitly and explicitly is covered in the next part of the tutorial. The code above shows the simplest possible way to create an **IEnumerable<SelectListItem >** and populate it with text and values. Note the `Comedy` SelectListItem has the Selected property set to **true;** this will cause the rendered select list to show **Comedy** as the selected item in the list.

The **IEnumerable<SelectListItem >** created above is added to the ViewBag with the name MovieType. This is how we pass the **IEnumerable<SelectListItem >** implicitly to the DropDownList helper shown below.

Open the *Views\Home\SelectCategory.cshtml* file and examine the markup.

```
@{

    ViewBag.Title = "Category Select";

    Layout = "~/Views/Shared/_Simple_Layout.cshtml";

}

@using (Html.BeginForm("CategoryChosen", "Home", FormMethod.Get)) {

    <fieldset>

            Movie Type

            @Html.DropDownList("MovieType")

        <p>

            <input type="submit" value="Submit" />

        </p>

    </fieldset>

}
```

On the third line, we set the layout to Views/Shared/_Simple_Layout.cshtml, which is a simplified version of the standard layout file. We do this to keep the display and rendered HTML simple.

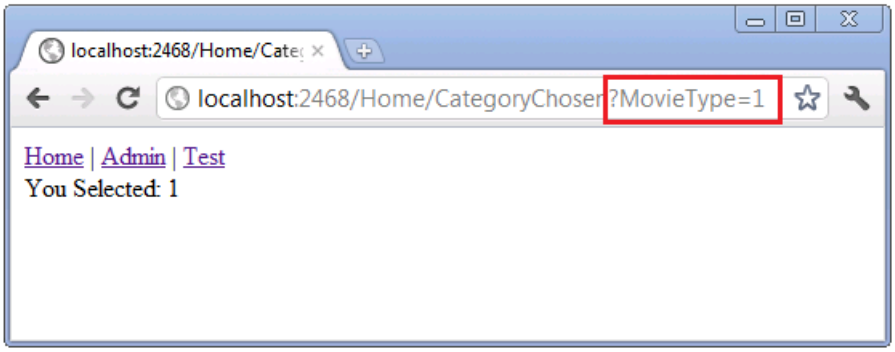
In this sample we are not changing the state of the application, so we will submit the data using an **HTTP GET**, not **HTTP POST**. See the W3C section Quick Checklist for Choosing HTTP GET or POST. Because we are not changing the application and posting the form, we use the Html.BeginForm overload that allows us to specify the action method, controller and form method (**HTTP POST** or **HTTP GET**). Typically views contain the Html.BeginForm overload that takes no parameters. The no parameter version defaults to posting the form data to the POST version of the same action method and controller.

The following line

```
@Html.DropDownList("MovieType")
```

passes a string argument to the **DropDownList** helper. This string, "MovieType" in our example, does two things:

- It provides the key for the **DropDownList** helper to find a **IEnumerable<SelectListItem >** in the **ViewBag**.
- It is data-bound to the MovieType form element. If the submit method is **HTTP GET**, `MovieType` will be a query string. If the submit method is **HTTP POST**, `MovieType` will be added in the message body. The following image shows the query string with the value of 1.



The following code shows the `CategoryChosen` method the form was submitted to.

```
public ViewResult CategoryChosen(string MovieType) {

    ViewBag.messageString = MovieType;

    return View("Information");

}
```

Navigate back to the test page and select the **HTML SelectList** link. The HTML page renders a select element similar to the simple ASP.NET MVC test page. Right click the browser window and select **view source**. The HTML markup for the select list is essentially identical. Test the HTML page, it works like the ASP.NET MVC action method and view we previously tested.

Improving the Movie Select List with Enums

If the categories in your application are fixed and will not change, you can take advantage of enums to make your code more robust and simpler to extend. When you add a new category, the correct category value is generated. The avoids copy and paste errors when you add a new category but forget to update the category value.

Open the *Controllers\HomeController.cs* file and examine the following code:

```
public enum eMovieCategories { Action, Drama, Comedy, Science_Fiction };

private void SetViewBagMovieType(eMovieCategories selectedMovie) {

    IEnumerable<eMovieCategories> values =

                        Enum.GetValues(typeof(eMovieCategories))

                        .Cast<eMovieCategories>();

    IEnumerable<SelectListItem> items =

        from value in values

        select new SelectListItem

        {

            Text = value.ToString(),

            Value = value.ToString(),

            Selected = value == selectedMovie,

        };

    ViewBag.MovieType = items;

}

public ActionResult SelectCategoryEnum() {

    SetViewBagMovieType(eMovieCategories.Drama);

    return View("SelectCategory");

}
```

The enum `eMovieCategories` captures the four movie types. The `SetViewBagMovieType` method creates the **IEnumerable<SelectListItem >** from the `eMovieCategories` **enum**, and sets the `Selected` property from the `selectedMovie` parameter. The `SelectCategoryEnum` action method uses the same view as the `SelectCategory` action method.

Navigate to the Test page and click on the `Select Movie Category (Enum)` link. This time, instead of a value (number) being displayed, a string representing the enum is displayed.

Posting Enum Values

HTML Forms are typically used to post data to the server. The following code shows the `HTTP GET` and `HTTP POST` versions of the `SelectCategoryEnumPost` method.

```
public ActionResult SelectCategoryEnumPost() {

    SetViewBagMovieType(eMovieCategories.Comedy);

    return View();

}

[HttpPost]

public ActionResult SelectCategoryEnumPost(eMovieCategories MovieType) {

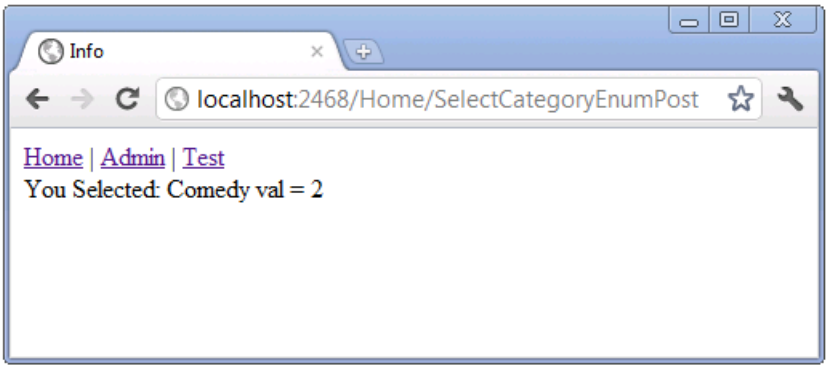
    ViewBag.messageString = MovieType.ToString() +

                            " val = " + (int)MovieType;

    return View("Information");

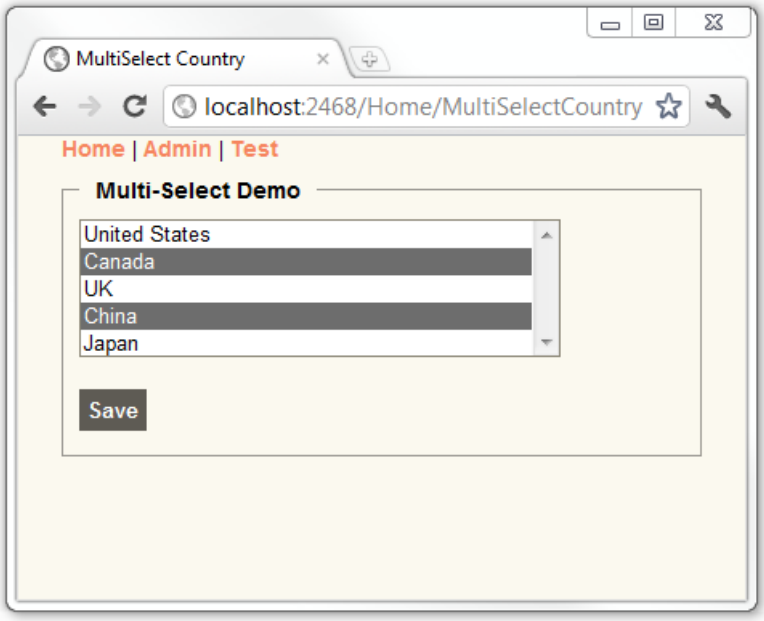
}
```

By passing a `eMovieCategories` enum to the `POST` method, we can extract both the enum value and the enum string. Run the sample and navigate to the Test page. Click on the `Select Movie Category(Enum Post)` link. Select a movie type and then hit the submit button. The display shows both the value and the name of the movie type.



Creating a Multiple Section Select Element

The ListBox HTML helper renders the HTML `<select>` element with the `multiple` attribute, which allows the users to make multiple selections. Navigate to the Test link, then select the **Multi Select Country** link. The rendered UI allows you to select multiple countries. In the image below, Canada and China are selected.



Examining the MultiSelectCountry Code

Examine the following code from the *Controllers\HomeController.cs* file.

```
private MultiSelectList GetCountries(string[] selectedValues) {

    List<Country> Countries = new List<Country>()

        {

            new Country() { ID = 1, Name= "United States" },

            new Country() { ID = 2, Name= "Canada" },

            new Country() { ID = 3, Name= "UK" },

            new Country() { ID = 4, Name= "China" },

            new Country() { ID = 5, Name= "Japan" }

        };

    return new MultiSelectList(Countries, "ID", "Name", selectedValues);

}

public ActionResult MultiSelectCountry() {

    ViewBag.Countrieslist = GetCountries(null);

    return View();

}
```

The `GetCountries` method creates a list of countries, then passes it to the `MultiSelectList` constructor. The `MultiSelectList` constructor overload used in the `GetCountries` method above takes four parameters:

```
public MultiSelectList(

    IEnumerable items,

    string dataValueField,

    string dataTextField,

    IEnumerable selectedValues

)
```

1. *items*: An IEnumerable containing the items in the list. In the example above, the list of Countries.
2. *dataValueField*: The name of the property in the **IEnumerable** list that contains the value. In the example above, the `ID` property.
3. *dataTextField*: The name of the property in the **IEnumerable** list that contains the information to display. In the example above, the `name` property.
4. *selectedValues*: The list of selected values.

In the example above, the `MultiSelectCountry` method passes a `null` value for the selected countries, so no countries are selected when the UI is displayed. The following code shows the Razor markup used to render the `MultiSelectCountry` view.

```
@{

    ViewBag.Title = "MultiSelect Country";

    Layout = "~/Views/Shared/_Simple_Layout.cshtml";

}

@if (ViewBag.YouSelected != null) {

    <div> You Selected:  @ViewBag.YouSelected</div>

}

@using (Html.BeginForm()) {

    <fieldset>

        <legend>Multi-Select Demo</legend>

        <div class="editor-field">

            @Html.ListBox("Countries", ViewBag.Countrieslist as MultiSelectList

            )

        </div>

        <p>

            <input type="submit" value="Save" />

        </p>

    </fieldset>

}
```

The HTML helper ListBox method used above take two parameters, the name of the property to model bind and the MultiSelectList containing the select options and values. The `ViewBag.YouSelected` code above is used to display the values of the countries you selected when you submit the form. Examine the HTTP POST overload of the `MultiSelectCountry` method.

```
[HttpPost]

public ActionResult MultiSelectCountry(FormCollection form) {

    ViewBag.YouSelected = form["Countries"];

    string selectedValues = form["Countries"];

    ViewBag.Countrieslist = GetCountries(selectedValues.Split(','));

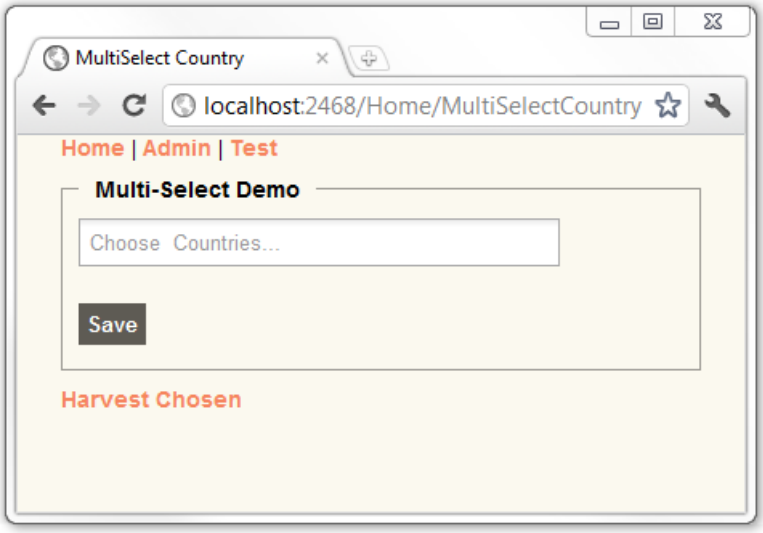
    return View();

}
```

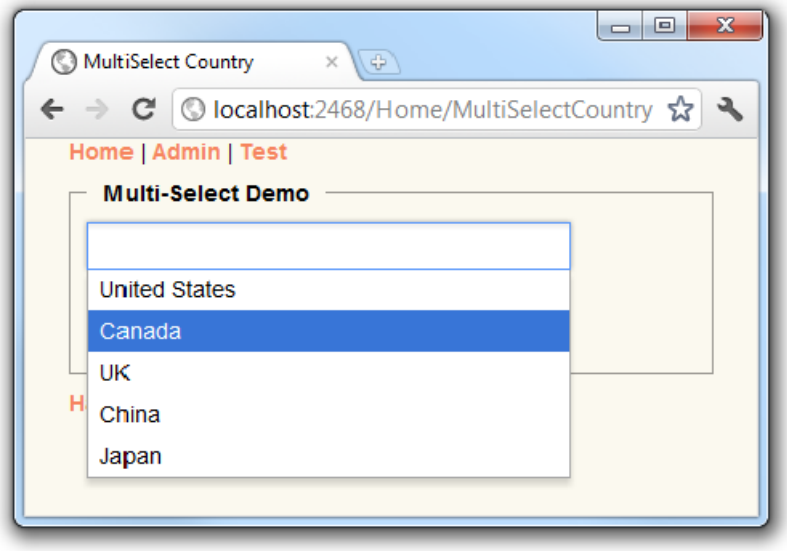
The `ViewBag.YouSelected` dynamic property contains the selected countries, obtained for the `Countries` entry in the form collection. In this version the GetCountries method is passed a list of the selected countries, so when the `MultiSelectCountry` view is displayed, the selected countries are selected in the UI.

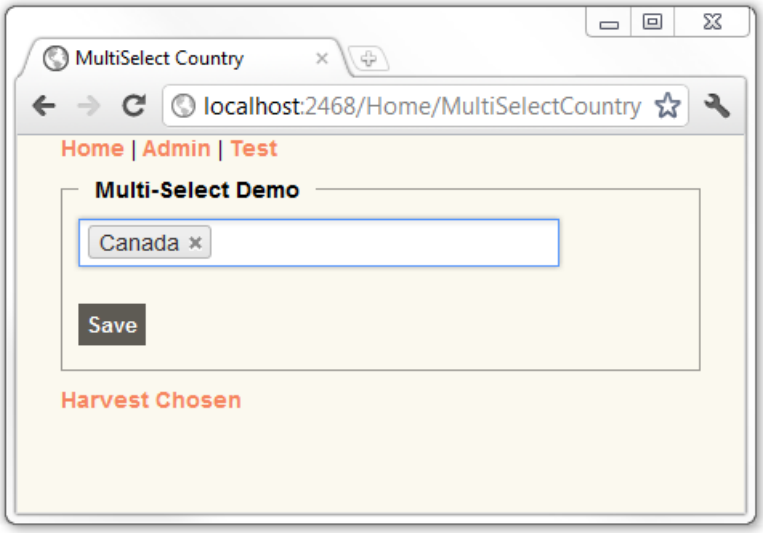
Making a Select Element Friendly with the Harvest Chosen jQuery Plugin

The Harvest Chosen jQuery plugin can be added to an HTML <select> element to create a user friendly UI. The images below demonstrate the Harvest Chosen jQuery plugin with `MultiSelectCountry` view.

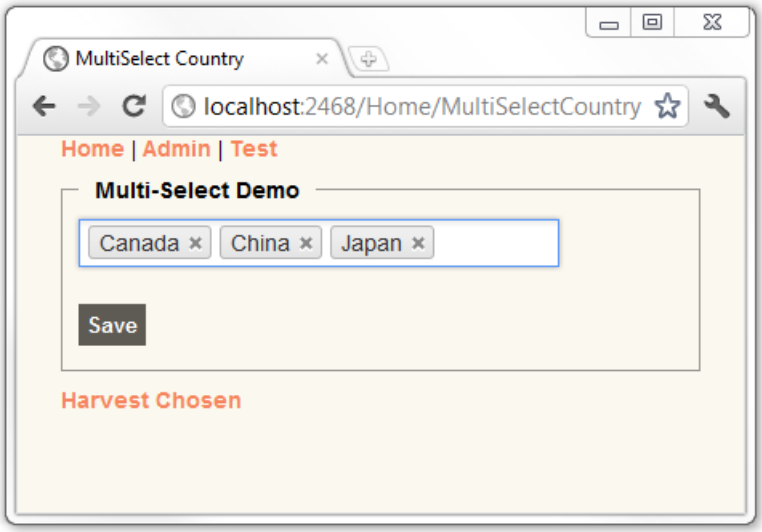


In the two images below, **Canada** is selected.





In the image above, Canada is selected, and it contains an **x** you can click to remove the selection. The image below shows Canada, China, and Japan selected.



Hooking up the Harvest Chosen jQuery Plugin

The following section is easier to follow if you have some experience with jQuery. If you have never used jQuery before, you might want to try one of the following jQuery tutorials.

- How jQuery Works by John Resig
- Getting Started with jQuery by Jörn Zaefferer
- Live Examples of jQuery by Cody Lindley

The Chosen plugin is included in the starter and completed sample projects that accompany this tutorial. For this tutorial you will only need to use jQuery to hook it up to the UI. To use the Harvest Chosen jQuery plugin in an ASP.NET MVC project, you must:

1. Download Chosen plugin from github. This step has been done for you.
2. Add the Chosen folder to your ASP.NET MVC project. Add the assets from the Chosen plugin you downloaded in the previous step to the Chosen folder. This step has been done for you.
3. Hook up the chosen plugin to the **DropDownList** or **ListBox** HTML helper.

Hooking up the Chosen Plugin to the MultiSelectCountry View.

Open the *Views\Home\MultiSelectCountry.cshtml* file and add an `htmlAttributes` parameter to the `Html.ListBox` . The parameter you will add contains a class name for the select list(`@class = "chzn-select"`). The completed code is shown below:

```
<div class="editor-field">

    @Html.ListBox("Countries", ViewBag.Countrieslist as MultiSelectList

     , new

     {

         @class = "chzn-select",

         data_placeholder = "Choose  Countries..."

     }

    )

</div>
```

In the code above, we are adding the HTML attribute and attribute value `class = "chzn-select"`. The @ character

preceding class has nothing to do with the Razor view engine. `class` is a C# keyword. C# keywords cannot be used as identifiers unless they include @ as a prefix. In the example above, `@class` is a valid identifier but **class** is not because **class** is a keyword.

Add references to the *Chosen/chosen.jquery.js* and *Chosen/chosen.css* files. The *Chosen/chosen.jquery.js* and implements the functionally of the Chosen plugin. The *Chosen/chosen.css* file provides the styling. Add these references to the bottom of the *Views\Home\MultiSelectCountry.cshtml* file. The following code shows how to reference the Chosen plugin.

```
<script src="@Url.Content("~/Chosen/chosen.jquery.js")" type="text/javascript"></script><script
src="@Url.Content("~/Chosen/chosen.jquery.js")" type="text/javascript"></script>

<link href="@Url.Content("~/Chosen/chosen.css")" rel="stylesheet"  type="text/css" />
```

Activate the Chosen plugin using the class name used in the **Html.ListBox** code. In the example above, the class name is `chzn-select`. Add the following line to the bottom of the *Views\Home\MultiSelectCountry.cshtml* view file. This line activates the Chosen plugin.

```
<script >    $(".chzn-select").chosen(); </script>  @*Hookup Chosen Plugin*@
```

The following line is the syntax to call the jQuery ready function, which selects the DOM element with class name `chzn-select`.

```
$(".chzn-select")
```

The wrapped set returned from the above call then applies the chosen method (`.chosen();`), which hooks up the Chosen plugin.

The following code shows the completed *Views\Home\MultiSelectCountry.cshtml* view file.

```
@{

    ViewBag.Title = "MultiSelect Country";

    Layout = "~/Views/Shared/_Simple_Layout.cshtml";

}

@if (ViewBag.YouSelected != null) {

    <div> You Selected:  @ViewBag.YouSelected</div>

}

@using (Html.BeginForm()) {

    <fieldset>

        <legend>Multi-Select Demo</legend>

<div class="editor-field">

    @Html.ListBox("Countries", ViewBag.Countrieslist as MultiSelectList

     , new

     {

         @class = "chzn-select"

     }

    )

</div>

        <p>

            <input type="submit" value="Save" />

        </p>

    </fieldset>

}

<script src="@Url.Content("~/Chosen/chosen.jquery.js")" type="text/javascript"></script>

<link href="@Url.Content("~/Chosen/chosen.css")" rel="stylesheet"  type="text/css" />

<script >    $(".chzn-select").chosen(); </script> @*Hookup Chosen Plugin*@
```

Run the application and navigate to the `MultiSelectCountry` view. Try adding and deleting countries. The sample download provided also contains a `MultiCountryVM` method and view that implements the MultiSelectCountry functionality using a view model instead of a **ViewBag**.

In the next section you'll see how the ASP.NET MVC scaffolding mechanism works with the **DropDownList** helper.

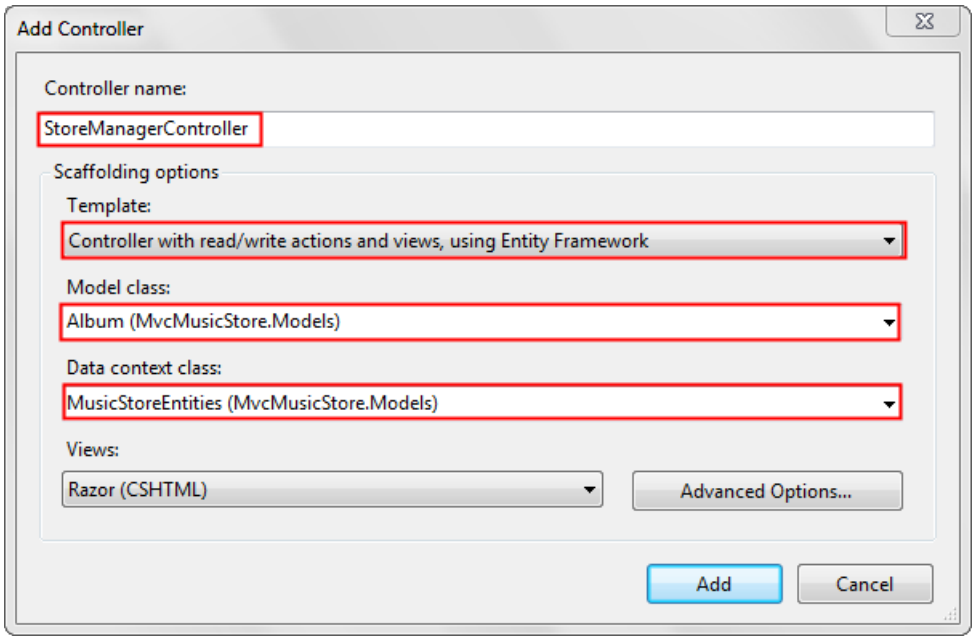
NEXT

Examining how ASP.NET MVC scaffolds the DropDownList Helper

2/20/2020 • 11 minutes to read • Edit Online

by Rick Anderson

In **Solution Explorer**, right-click the *Controllers* folder and then select **Add Controller**. Name the controller **StoreManagerController**. Set the options for the **Add Controller** dialog as shown in the image below.



Edit the *StoreManager\Index.cshtml* view and remove `AlbumArtUrl`. Removing `AlbumArtUrl` will make the presentation more readable. The completed code is shown below.

```
@model IEnumerable<MvcMusicStore.Models.Album>

@{

    ViewBag.Title = "Index";

}

<h2>Index</h2>

<p>

    @Html.ActionLink("Create New", "Create")

</p>

<table>

    <tr>

        <th>

            Genre

        </th>
```

```
        <th>
            Artist
        </th>
        <th>
            Title
        </th>
        <th>
            Price
        </th>
        <th></th>
    </tr>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Genre.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Artist.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) |
            @Html.ActionLink("Details", "Details", new { id=item.AlbumId }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })
        </td>
    </tr>
}
</table>
```

Open the *Controllers\StoreManagerController.cs* file and find the `Index` method. Add the `OrderBy` clause so the

albums will be sorted by price. The complete code is shown below.

```
public ViewResult Index()
{

    var albums = db.Albums.Include(a => a.Genre).Include(a => a.Artist)

        .OrderBy(a => a.Price);

    return View(albums.ToList());

}
```

Sorting by price will make it easier to test changes to the database. When you are testing the edit and create methods, you can use a low price so the saved data will appear first.

Open the *StoreManager\Edit.cshtml* file. Add the following line just after the legend tag.

```
@Html.HiddenFor(model => model.AlbumId)
```

The following code shows the context of this change:

```
@using (Html.BeginForm()) {

    @Html.ValidationSummary(true)

    <fieldset>

        <legend>Album</legend>

        @Html.HiddenFor(model => model.AlbumId)

        <div class="editor-label">

            @Html.LabelFor(model => model.GenreId, "Genre")

        </div>

        <div class="editor-field">

            @Html.DropDownList("GenreId", String.Empty)

            @Html.ValidationMessageFor(model => model.GenreId)

        </div>

        <!-- Items removed for brevity. -->

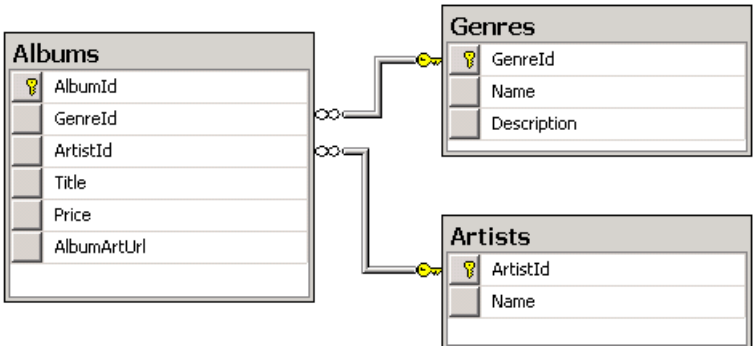
}
```

The `AlbumId` is required to make changes to an album record.

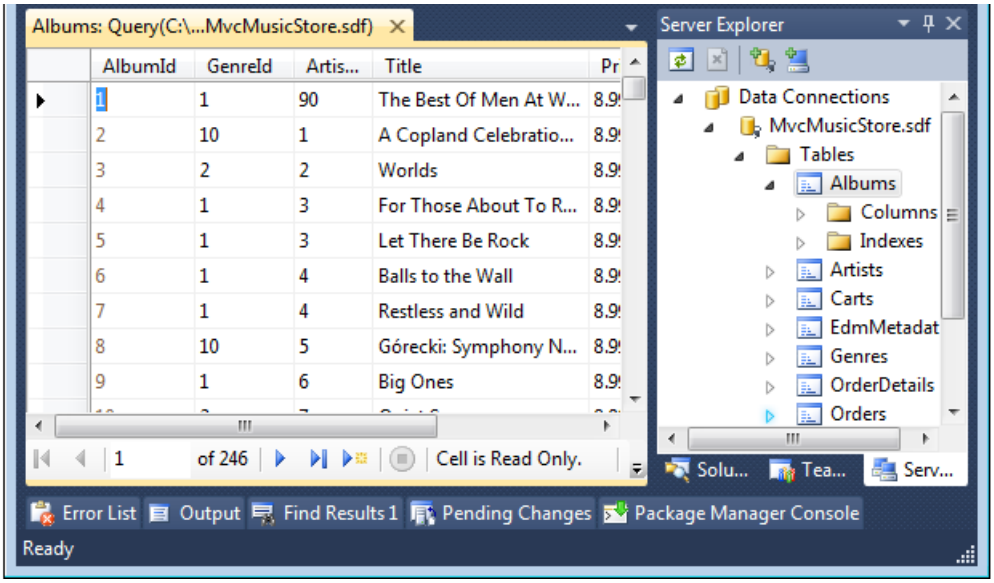
Press CTRL+F5 to run the application. Select to the **Admin** link, then select the **Create New** link to create a new album. Verify the album information was saved. Edit an album and verify the changes you made are persisted.

The Album Schema

The `StoreManager` controller created by the MVC scaffolding mechanism allows CRUD (Create, Read, Update, Delete) access to the albums in the music store database. The schema for album information is shown below:

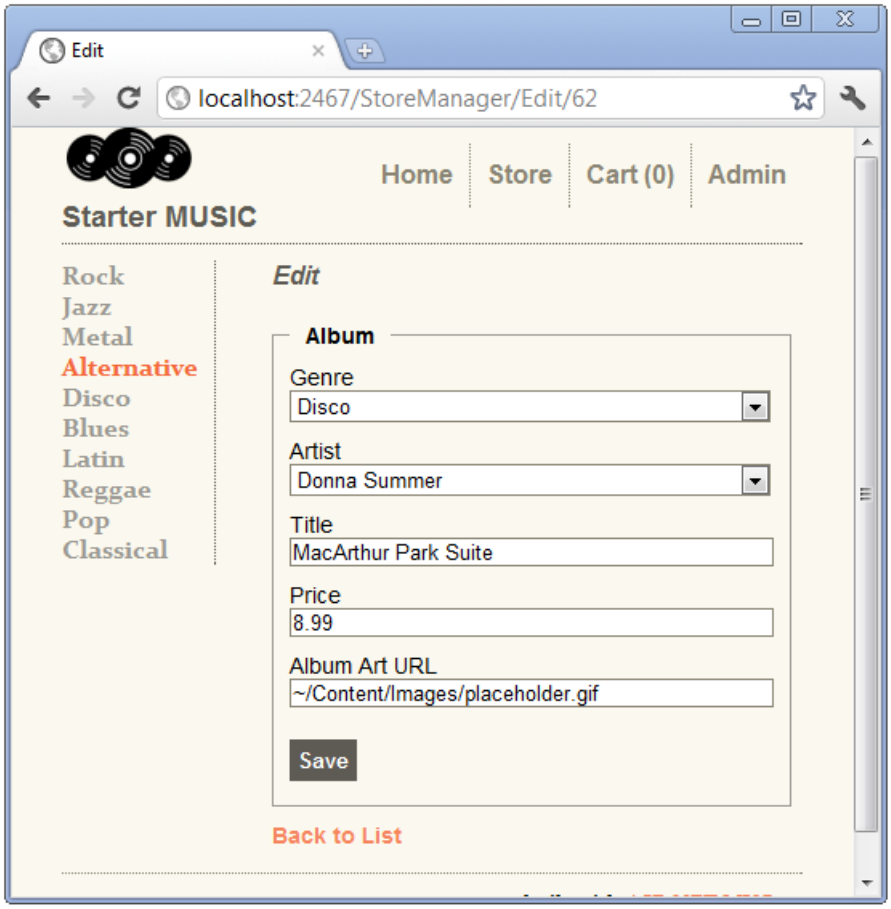


The `Albums` table does not store the album genre and description, it stores a foreign key to the `Genres` table. The `Genres` table contains the genre name and description. Likewise, the `Albums` table doesn't contain the album artists name, but a foreign key to the `Artists` table. The `Artists` table contains the artist's name. If you examine the data in the `Albums` table, you can see each row contains a foreign key to the `Genres` table and a foreign key to the `Artists` table. The image below show some table data from the `Albums` table.



The HTML Select Tag

The HTML `<select>` element (created by the HTML DropDownList helper) is used to display a complete list of values (such as the list of genres). For edit forms, when the current value is known, the select list can display the current value. We saw this previously when we set the selected value to **Comedy**. The select list is ideal for displaying category or foreign key data. The `<select>` element for the Genre foreign key displays the list of possible genre names, but when you save the form the Genre property is updated with the Genre foreign key value, not the displayed genre name. In the image below, the genre selected is **Disco** and the artist is **Donna Summer**.



Examining the ASP.NET MVC Scaffolded Code

Open the *Controllers\StoreManagerController.cs* file and find the `HTTP GET Create` method.

```
public ActionResult Create()

{

    ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name");

    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name");

    return View();

}
```

The `Create` method adds two SelectList objects to the `ViewBag`, one to contain the genre information, and one to contain the artist information. The SelectList constructor overload used above takes three arguments:

```
public SelectList(

    IEnumerable items,

    string dataValueField,

    string dataTextField

)
```

1. *items*: An IEnumerable containing the items in the list. In the example above, the list of genres returned by `db.Genres`.
2. *dataValueField*: The name of the property in the **IEnumerable** list that contains the key value. In the example

above, `GenreId` and `ArtistId`.

3. *dataTextField*: The name of the property in the **IEnumerable** list that contains the information to display. In both the artists and genre table, the `name` field is used.

Open the *Views\StoreManager\Create.cshtml* file and examine the `Html.DropDownList` helper markup for the genre field.

```
@model MvcMusicStore.Models.Album

@*        Markup removed for clarity.*@

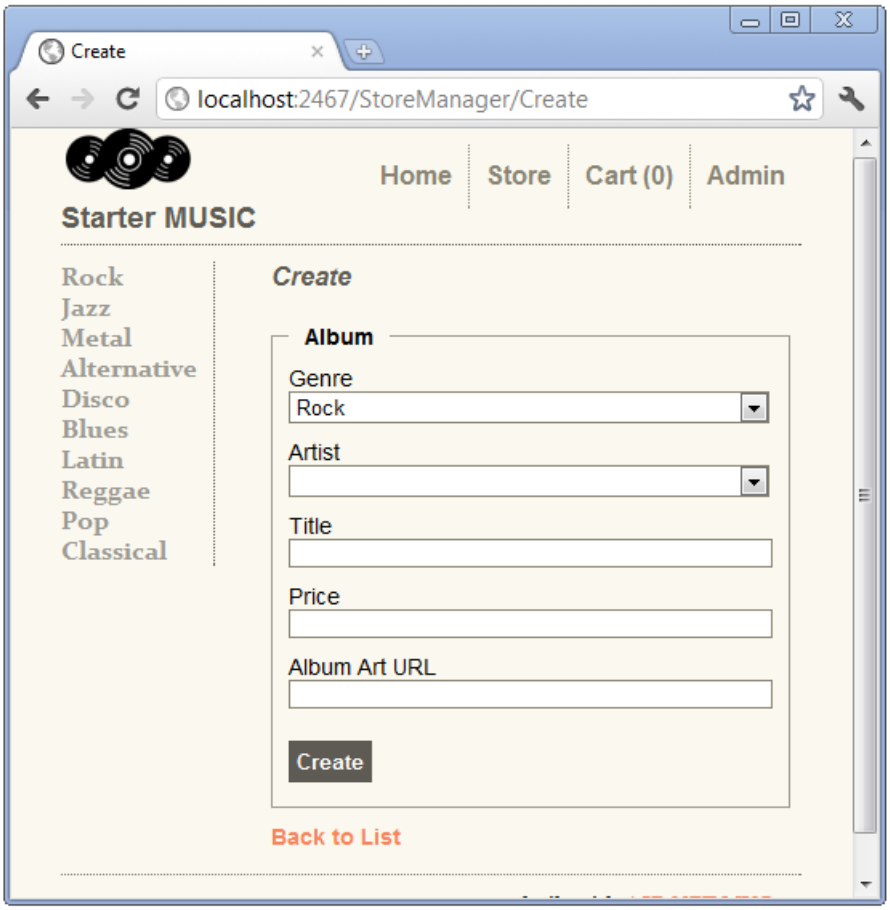
@Html.DropDownList("GenreId", String.Empty)
```

The first line shows that the create view takes an `Album` model. In the `Create` method shown above, no model was passed, so the view gets a **null** `Album` model. At this point we are creating a new album so we don't have any `Album` data for it.

The Html.DropDownList overload shown above takes the name of the field to bind to the model. It also uses this name to look for a **ViewBag** object containing a SelectList object. Using this overload, you are required to name the **ViewBag SelectList** object `GenreId`. The second parameter (`String.Empty`) is the text to display when no item is selected. This is exactly what we want when creating a new album. If you removed the second parameter and used the following code:

```
@Html.DropDownList("GenreId")
```

The select list would default to the first element, or Rock in our sample.



Examining the `HTTP POST Create` method.

```
//

// POST: /StoreManager/Create

[HttpPost]

public ActionResult Create(Album album)

{

    if (ModelState.IsValid)

    {

        db.Albums.Add(album);

        db.SaveChanges();

        return RedirectToAction("Index");

    }

    ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name",

        album.GenreId);

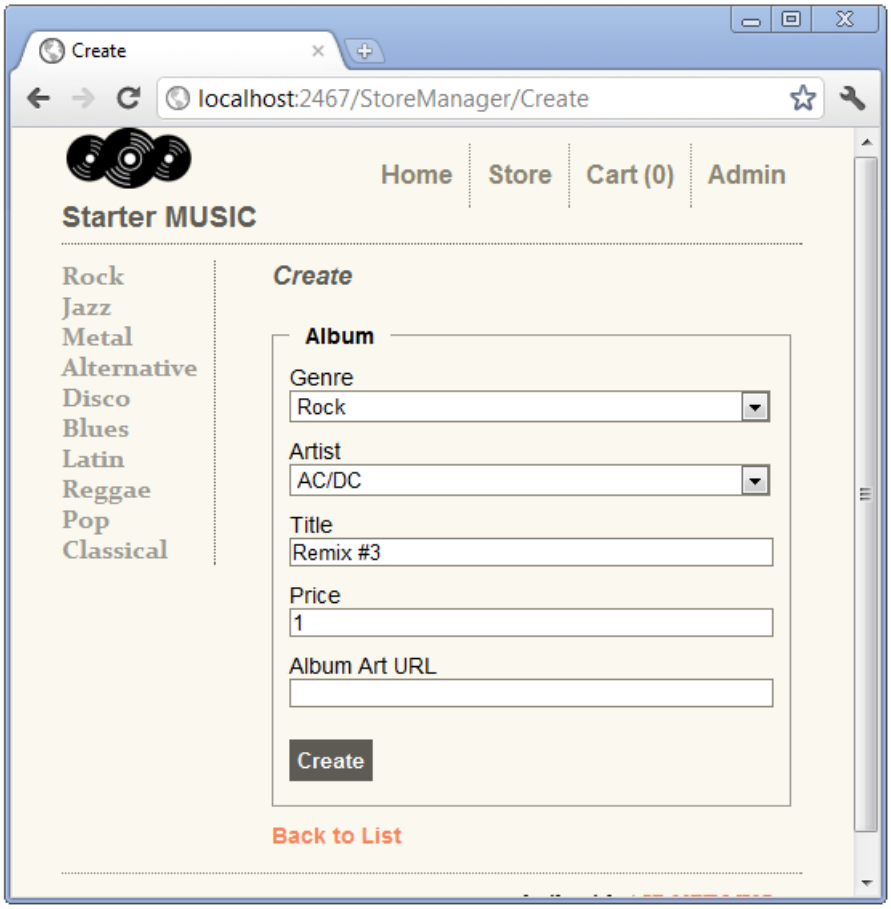
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name",

        album.ArtistId);

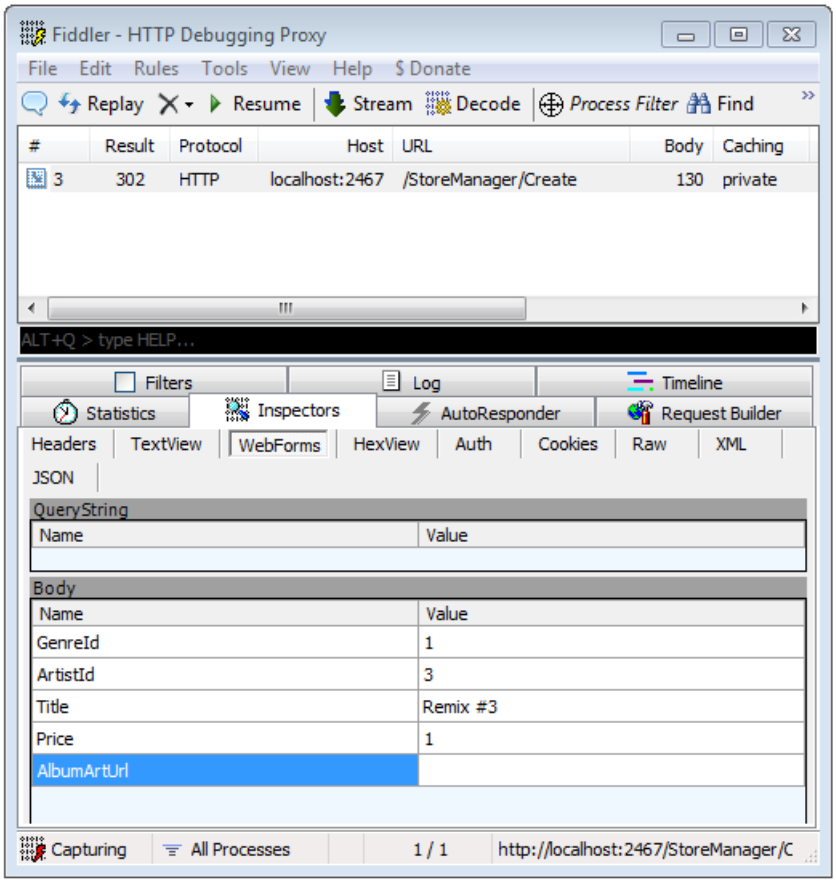
    return View(album);

}
```

This overload of the `Create` method takes an `album` object, created by the ASP.NET MVC model binding system from the form values posted. When you submit a new album, if model state is valid and there are no database errors, the new album is added the database. The following image shows the creation of a new album.



You can use the fiddler tool to examine the posted form values that ASP.NET MVC model binding uses to create the album object.



Refactoring the ViewBag SelectList Creation

Both the `Edit` methods and the `HTTP POST Create` method have identical code to set up the **SelectList** in the **ViewBag**. In the spirit of DRY, we will refactor this code. We'll make use of this refactored code later.

Create a new method to add a genre and artist **SelectList** to the **ViewBag**.

```
private void SetGenreArtistViewBag(int? GenreID = null, int? ArtistID = null) {

    if (GenreID == null)

        ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name");

    else

        ViewBag.GenreId = new SelectList(db.Genres.ToArray(), "GenreId", "Name", GenreID);

    if (ArtistID == null)

        ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name");

    else

        ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name", ArtistID);

}
```

Replace the two lines setting the `ViewBag` in each of the `Create` and `Edit` methods with a call to the `SetGenreArtistViewBag` method. The completed code is shown below.

```
//

// GET: /StoreManager/Create

public ActionResult Create() {

    SetGenreArtistViewBag();

    return View();

}

//

// POST: /StoreManager/Create

[HttpPost]

public ActionResult Create(Album album) {

    if (ModelState.IsValid) {

        db.Albums.Add(album);

        db.SaveChanges();

        return RedirectToAction("Index");

    }

    SetGenreArtistViewBag(album.GenreId, album.ArtistId);

    return View(album);

}

//

// GET: /StoreManager/Edit/5

public ActionResult Edit(int id) {
```

```
public ActionResult Edit(int id) {

    Album album = db.Albums.Find(id);

    SetGenreArtistViewBag(album.GenreId, album.ArtistId);

    return View(album);

}

//

// POST: /StoreManager/Edit/5

[HttpPost]

public ActionResult Edit(Album album) {

    if (ModelState.IsValid) {

        db.Entry(album).State = EntityState.Modified;

        db.SaveChanges();

        return RedirectToAction("Index");

    }

    SetGenreArtistViewBag(album.GenreId, album.ArtistId);

    return View(album);

}
```

Create a new album and edit an album to verify the changes work.

Explicitly Passing the SelectList to the DropDownList

The create and edit views created by the ASP.NET MVC scaffolding use the following **DropDownList** overload:

```
public static MvcHtmlString DropDownList(

    this HtmlHelper htmlHelper,

    string name,         // The name of the ViewModel property to bind.

    string optionLabel   // The string added to the top of the list

                         // typically  String.Empty or "Select a Genre"

)
```

The `DropDownList` markup for the create view is shown below.

```
@Html.DropDownList("GenreId", String.Empty)
```

Because the `ViewBag` property for the `SelectList` is named `GenreId`, the **DropDownList** helper will use the `GenreId` **SelectList** in the **ViewBag**. In the following **DropDownList** overload, the `SelectList` is explicitly passed in.

```
public static MvcHtmlString DropDownList(

    this HtmlHelper htmlHelper,

    string name,            // The name of the ViewModel property to bind.

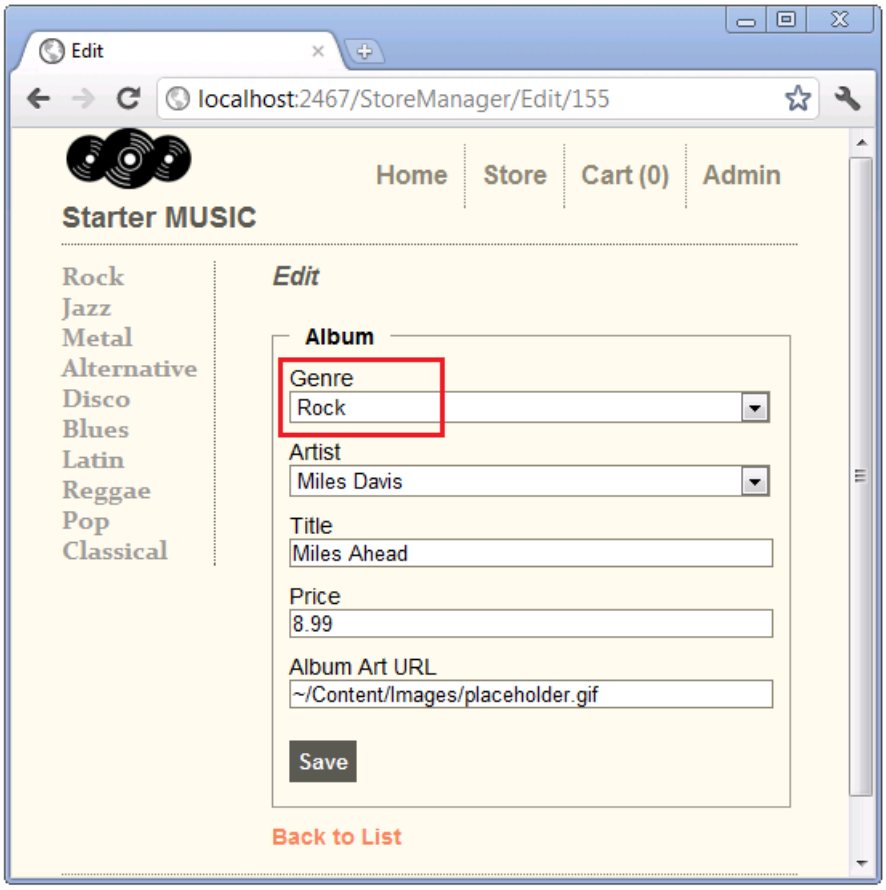
    IEnumerable selectList  // The SelectList

)
```

Open the *Views\StoreManager\Edit.cshtml* file, and change the **DropDownList** call to explicitly pass in the **SelectList**, using the overload above. Do this for the Genre category. The completed code is shown below:

```
@Html.DropDownList("GenreId", ViewBag.GenreId as SelectList)
```

Run the application and click the **Admin** link, then navigate to a Jazz album and select the **Edit** link.



Instead of showing Jazz as the currently selected genre, Rock is displayed. When the string argument (the property to bind) and the **SelectList** object have the same name, the selected value is not used. When there is no selected value provided, browsers default to the first element in the **SelectList**(which is **Rock** in the example above). This is a known limitation of the **DropDownList** helper.

Open the *Controllers\StoreManagerController.cs* file and change the **SelectList** object names to `Genres` and `Artists`. The completed code is shown below:

```
private void SetGenreArtistViewBag(int? GenreID = null, int? ArtistID = null) {
    if (GenreID == null)

        ViewBag.Genres = new SelectList(db.Genres, "GenreId", "Name");

    else

        ViewBag.Genres = new SelectList(db.Genres.ToArray(), "GenreId", "Name", GenreID);

    if (ArtistID == null)

        ViewBag.Artists = new SelectList(db.Artists, "ArtistId", "Name");

    else

        ViewBag.Artists = new SelectList(db.Artists, "ArtistId", "Name", ArtistID);

}
```

The names Genres and Artists are better names for the categories, as they contain more than just the ID of each category. The refactoring we did earlier paid off. Instead of changing the **ViewBag** in four methods, our changes were isolated to the `SetGenreArtistViewBag` method.

Change the **DropDownList** call in the create and edit views to use the new **SelectList** names. The new markup for the edit view is shown below:

```
<div class="editor-label">

    @Html.LabelFor(model => model.GenreId, "Genre")

</div>

<div class="editor-field">

    @Html.DropDownList("GenreId", ViewBag.Genres as SelectList)

    @Html.ValidationMessageFor(model => model.GenreId)

</div>

<div class="editor-label">

    @Html.LabelFor(model => model.ArtistId, "Artist")

</div>

<div class="editor-field">

    @Html.DropDownList("ArtistId", ViewBag.Artists as SelectList)

    @Html.ValidationMessageFor(model => model.ArtistId)

</div>
```

The Create view requires an empty string to prevent the first item in the SelectList from being displayed.

```
<div class="editor-label">

    @Html.LabelFor(model => model.GenreId, "Genre" )

</div>

<div class="editor-field">

    @Html.DropDownList("GenreId", ViewBag.Genres as SelectList, String.Empty)

    @Html.ValidationMessageFor(model => model.GenreId)

</div>

<div class="editor-label">

    @Html.LabelFor(model => model.ArtistId, "Artist")

</div>

<div class="editor-field">

    @Html.DropDownList("ArtistId", ViewBag.Artists as SelectList, String.Empty)

    @Html.ValidationMessageFor(model => model.ArtistId)

</div>
```

Create a new album and edit an album to verify the changes work. Test the edit code by selecting an album with a genre other than Rock.

Using a View Model with the DropDownList Helper

Create a new class in the ViewModels folder named `AlbumSelectListViewModel`. Replace the code in the `AlbumSelectListViewModel` class with the following:

```
using MvcMusicStore.Models;

using System.Web.Mvc;

using System.Collections;

namespace MvcMusicStore.ViewModels {

    public class AlbumSelectListViewModel {

        public Album Album { get; private set; }

        public SelectList Artists { get; private set; }

        public SelectList Genres { get; private set; }

        public AlbumSelectListViewModel(Album album,

                                        IEnumerable artists,

                                        IEnumerable genres) {

            Album = album;

            Artists = new SelectList(artists, "ArtistID", "Name", album.ArtistId);

            Genres = new SelectList(genres, "GenreID", "Name", album.GenreId);

        }

    }

}
```

The `AlbumSelectListViewModel` constructor takes an album, a list of artists and genres and creates an object containing the album and a `SelectList` for genres and artists.

Build the project so the `AlbumSelectListViewModel` is available when we create a view in the next step.

Add an `EditVM` method to the `StoreManagerController`. The completed code is shown below.

```
//

// GET: /StoreManager/EditVM/5

public ActionResult EditVM(int id) {

    Album album = db.Albums.Find(id);

    if (album == null)

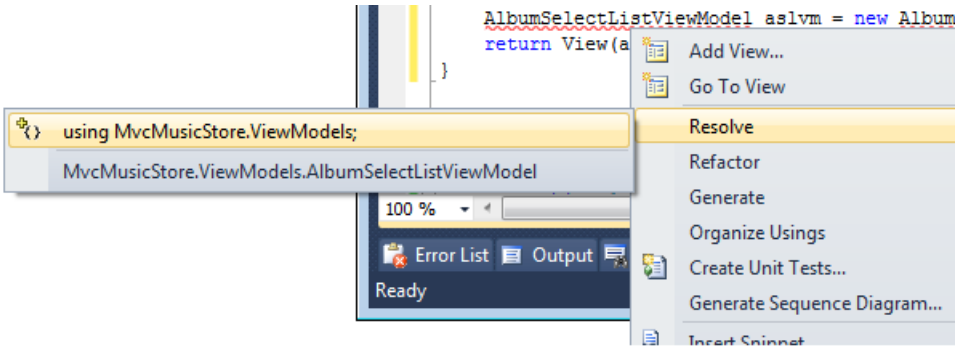
        return HttpNotFound();

    AlbumSelectListViewModel aslvm = new AlbumSelectListViewModel(album, db.Artists, db.Genres);

    return View(aslvm);

}
```

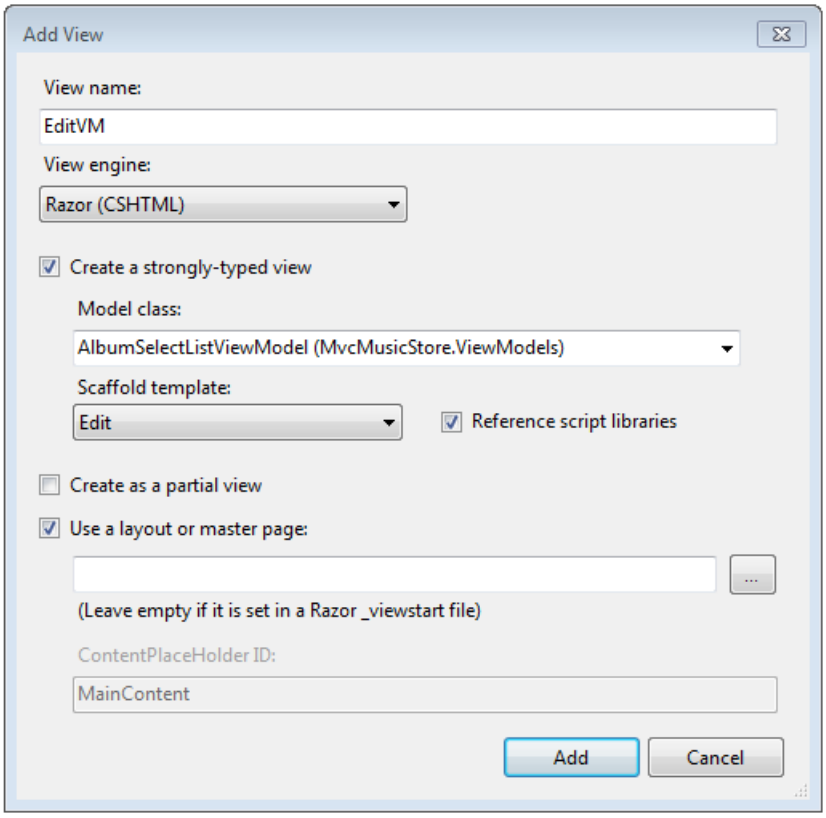
Right click `AlbumSelectListViewModel`, select **Resolve**, then **using MvcMusicStore.ViewModels;**.



Alternatively, you can add the following using statement:

```
using MvcMusicStore.ViewModels;
```

Right click `EditVM` and select **Add View**. Use the options shown below.



Select **Add**, then replace the contents of the *Views\StoreManager\EditVM.cshtml* file with the following:

```
@model MvcMusicStore.ViewModels.AlbumSelectListViewModel

@{

    ViewBag.Title = "EditVM";

}

<h2>Edit VM</h2>

@using (Html.BeginForm("Edit","StoreManager",FormMethod.Post)) {

    @Html.ValidationSummary(true)

    <fieldset>

        <legend>Album</legend>
```

```
        @Html.HiddenFor(model => model.Album.AlbumId )

        <div class="editor-label">

            @Html.LabelFor(model => model.Album.GenreId, "Genre")

        </div>

        <div class="editor-field">

            @Html.DropDownList("Album.GenreId", Model.Genres)

            @Html.ValidationMessageFor(model => model.Album.GenreId)

        </div>

        <div class="editor-label">

            @Html.LabelFor(model => model.Album.ArtistId, "Artist")

        </div>

        <div class="editor-field">

            @Html.DropDownList("Album.ArtistId", Model.Artists)

            @Html.ValidationMessageFor(model => model.Album.ArtistId)

        </div>

        <div class="editor-label">

            @Html.LabelFor(model => model.Album.Title)

        </div>

        <div class="editor-field">

            @Html.EditorFor(model => model.Album.Title)

            @Html.ValidationMessageFor(model => model.Album.Title)

        </div>

        <div class="editor-label">

            @Html.LabelFor(model => model.Album.Price)

        </div>

        <div class="editor-field">

            @Html.EditorFor(model => model.Album.Price)

            @Html.ValidationMessageFor(model => model.Album.Price)

        </div>

        <div class="editor-label">

            @Html.LabelFor(model => model.Album.AlbumArtUrl)

        </div>

        <div class="editor-field">

            @Html.EditorFor(model => model.Album.AlbumArtUrl)
```

```
            @Html.ValidationMessageFor(model => model.Album.AlbumArtUrl)

        </div>

        <p>

            <input type="submit" value="Save" />

        </p>

    </fieldset>

}

<div>

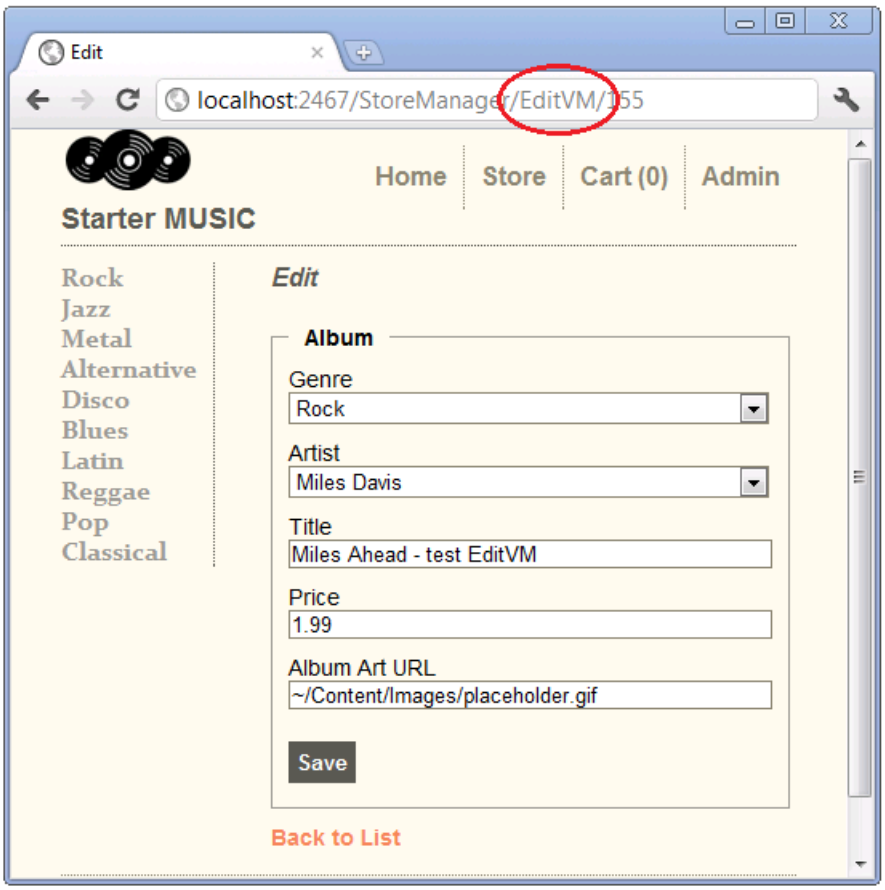
    @Html.ActionLink("Back to List", "Index")

</div>
```

The `EditVM` markup is very similar to the original `Edit` markup with the following exceptions.

- Model properties in the `Edit` view are of the form `model.property` (for example, `model.Title`). Model properties in the `EditVm` view are of the form `model.Album.property` (for example, `model.Album.Title`). That's because the `EditVM` view is passed a container for an `Album` , not an `Album` as in the `Edit` view.
- The **DropDownList** second parameter comes from the view model, not the **ViewBag**.
- The **BeginForm** helper in the `EditVM` view explicitly posts back to the `Edit` action method. By posting back to the `Edit` action, we don't have to write an `HTTP POST EditVM` action and can reuse the `HTTP POST` `Edit` action.

Run the application and edit an album. Change the URL to use `EditVM` . Change a field and hit the **Save** button to verify the code is working.



Which Approach Should You Use?

All three approaches shown are acceptable. Many developers prefer to explicitly pass the `SelectList` to the

`DropDownList` using the `ViewBag`. This approach has the added advantage of giving you the flexibility of using a more appropriate name for the collection. The one caveat is you cannot name the `ViewBag SelectList` object the same name as the model property.

Some developers prefer the ViewModel approach. Others consider the more verbose markup and generated HTML of the ViewModel approach a disadvantage.

In this section we have learned three approaches to using the **DropDownList** with category data. In the next section, we'll show how to add a new category.

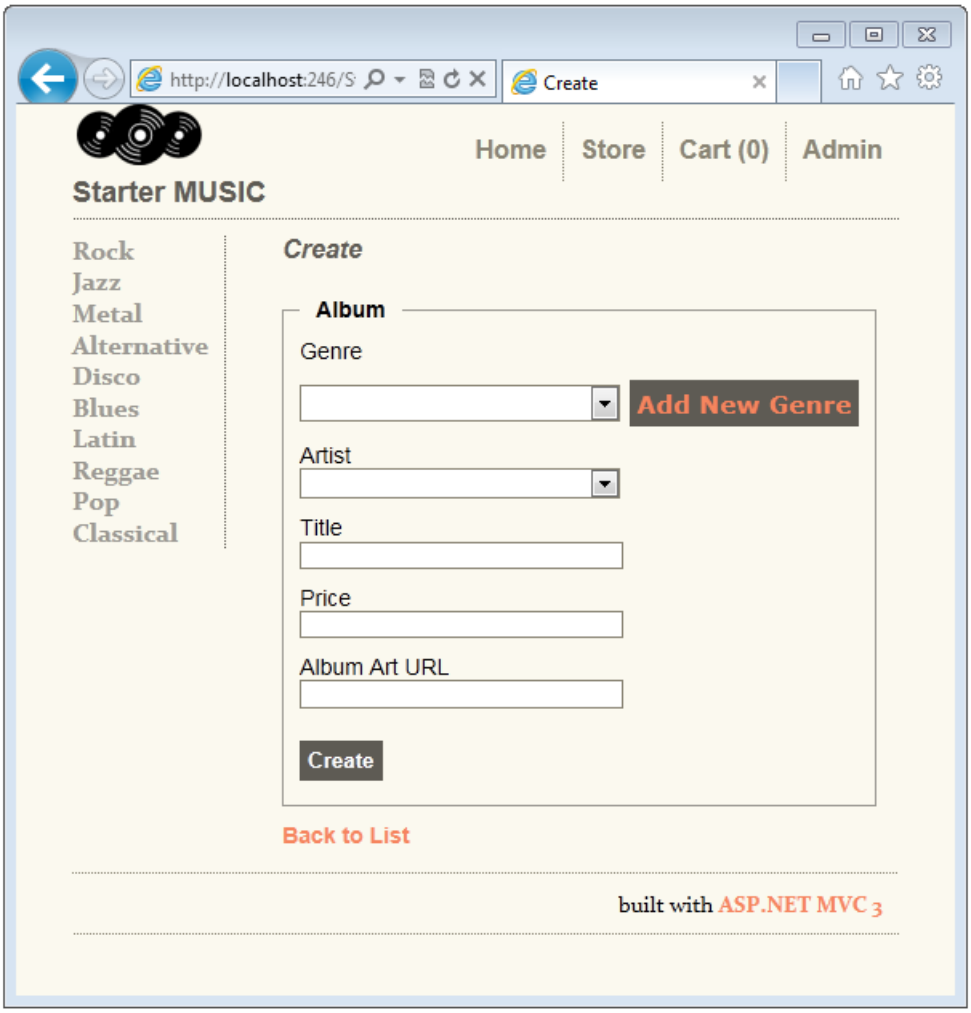
PREVIOUS NEXT

Adding a New Category to the DropDownList using jQuery UI

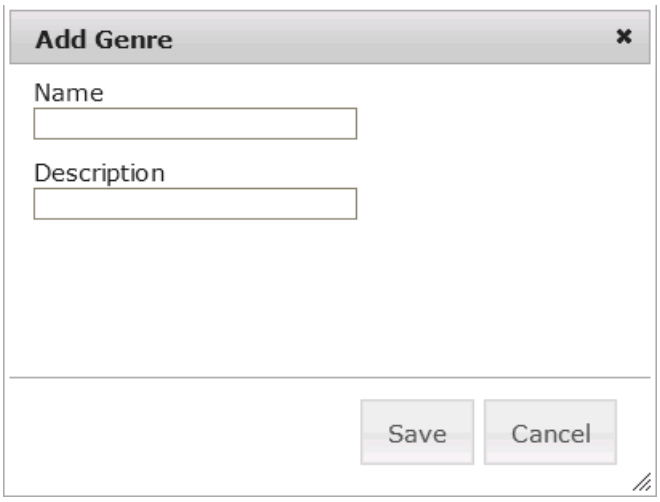
2/20/2020 • 6 minutes to read • Edit Online

by Rick Anderson

The HTML `Select` tag is ideal for presenting a list of fixed category data, but often times you need to add a new category. Suppose we want to add the genre "Opera" to the categories in our database? In this section, we will use jQuery UI to add a dialog box we can use to add a new category. The image below shows how the UI will present in the browser.



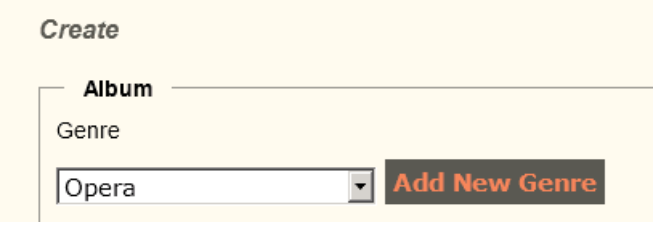
When a user selects the **Add New Genre** link, a pop-up dialog box prompts the user for a new genre name (and optionally a description). The image below show the **Add Genre** pop-up dialog.



When a new genre name is entered and the **Save** button is pushed, the following happens:

1. An AJAX call posts the data to the Create method of the Genre Controller, which saves the new genre to the database and returns the new genre information (genre name and ID) as JSON.
2. JavaScript adds the new genre data to the select list.
3. JavaScript makes the new genre the selected item.

   In the image below, **Opera** was added to the database and selected in the **Genre** drop down list.



Open the *Views\StoreManager\Create.cshtml* file and replace the genre markup with the following the following code:

```
<div class="editor-field ui-widget">

     @Html.Partial("_ChooseGenre")

</div>
```

The `_ChooseGenre` partial view will contain all the logic to hook up the JavaScript and jQuery used to implement the add new genre feature. Once we have completed the code it will be simple to do the same with the artist UI.

In Solution Explorer, right click the *Views\StoreManager* folder and select **Add**, then **View**. In the **View name** input, enter `_ChooseGenre` then select **Add**. Replace the markup in the *Views\StoreManager_ChooseGenre.cshtml* file with the following:

```
@model MvcMusicStore.Models.Album

<div class="editor-label">

    @Html.LabelFor(model => model.GenreId, "Genre" )

</div>

@Html.DropDownList("GenreId", ViewBag.Genres as SelectList, String.Empty)

<a class="button" href="@Url.Content("~/Genre/Create")"

    id="genreAddLink">Add New Genre</a>

@Html.ValidationMessageFor(model => model.GenreId)

<div id="genreDialog" class="hidden">

</div>

<script src="@Url.Content( "~/Scripts/ui/jquery.ui.combobox.js" )"></script>

<script src="@Url.Content("~/Scripts/chooseGenre.js")"></script>
```

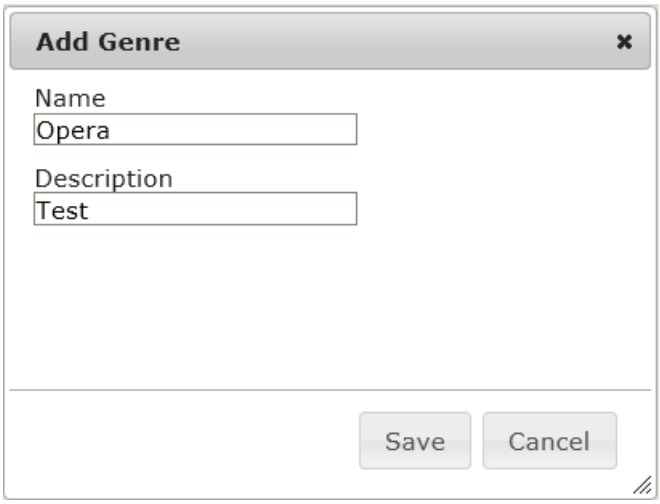
The first line declares that we are passing in an `Album` as our model, exactly the same model statement found in the Create view. The next few lines are the **Label** helper markup. The next line is the **DropDownList** helper call, exactly the same as in the original Create view. The next line adds a link with the name `Add New Genre`, and styles it like a button. The line containing `ValidationMessageFor` is copied directly from the Create view. The following lines:

```
<div id="genreDialog" class="hidden">

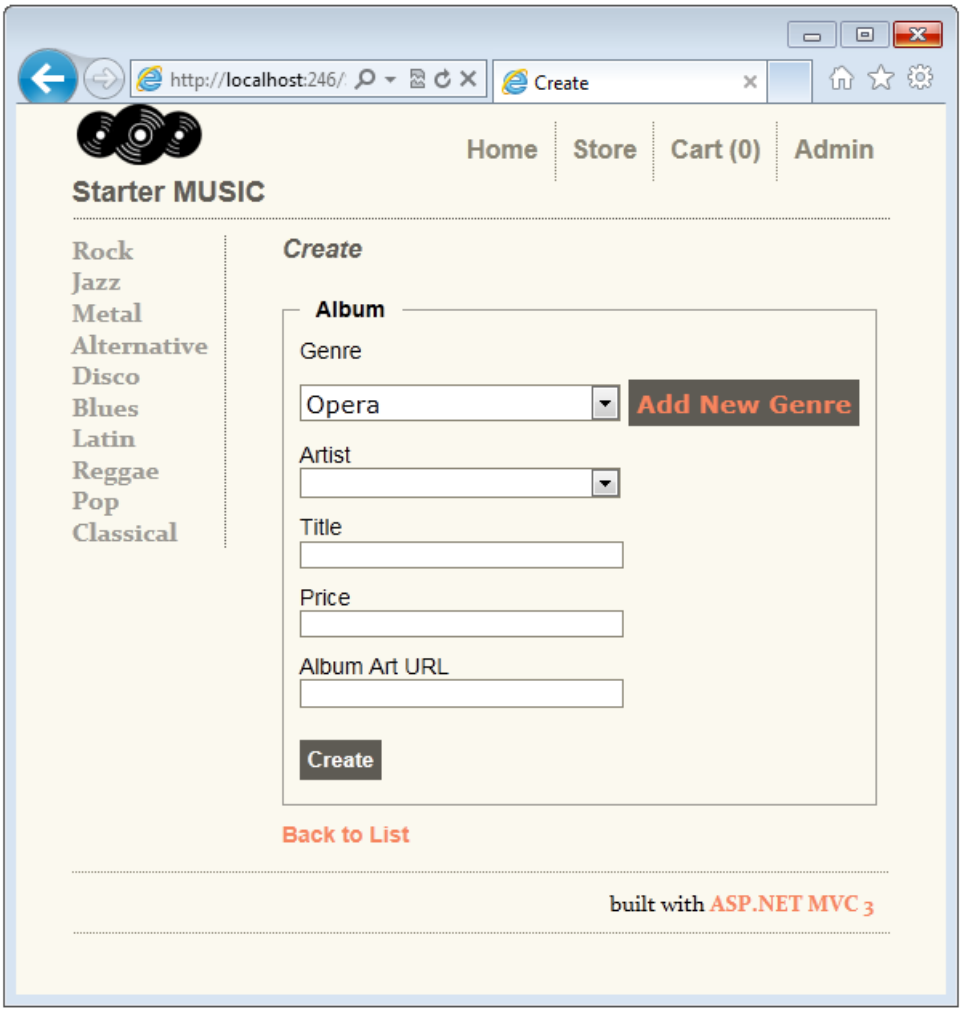
</div>
```

creates a hidden div, with the ID of `genreDialog`. We will use jQuery to hook up our **Add Genre** dialog box with the ID `genreDialog` in this div. The last two script tags contain links to the JavaScript files we will use to implement the add new genre feature. The */Scripts/chooseGenre.js* file is provided for you in the project, we will examine it later in the tutorial.

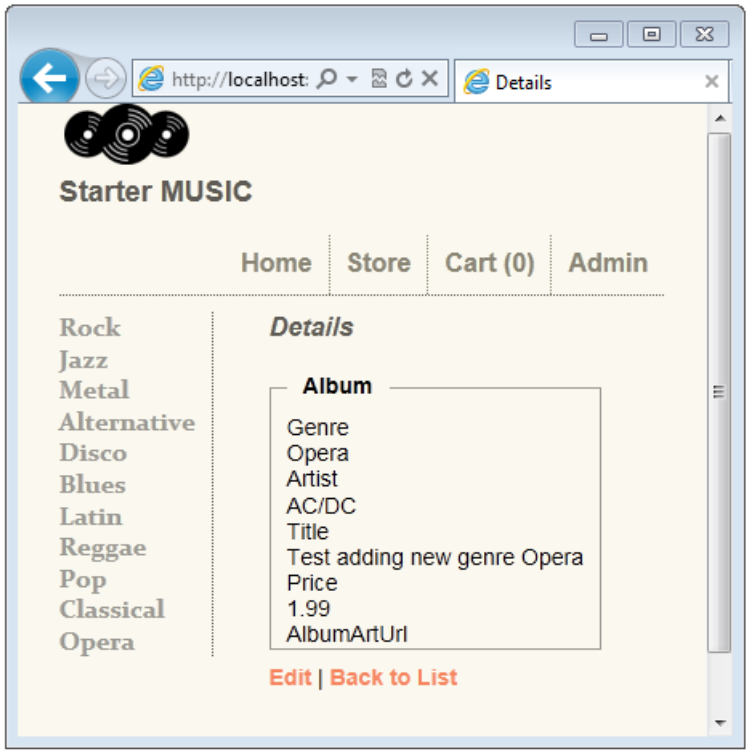
Run the application and click on the **Add New Genre** button. In the **Add Genre** dialog box, enter **Opera** in the **Name** input box.



Click the **Save** button. An AJAX call creates the Opera category and then populates the dropdown list with Opera, and sets Opera as the selected genre.



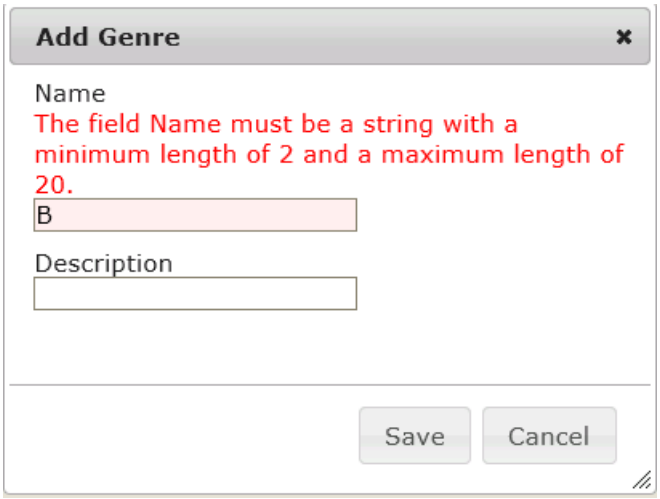
Enter an artist, title and price, then select the **Create** button. If you enter a price less than $8.99, the new album will appear at the top of the Index view. Verify the new album entry was saved in the database.



Try creating a new genre with only one letter. The following code in the *Models\Genre.cs* file sets the minimum and maximum length of the genre name.

```
[StringLength(20, MinimumLength = 2)]
public string Name { get; set; }
```

Client side validation reports you must enter a string between 2 and 20 characters.



Examining How a New Genre is Added to the Database and the Select List.

Open the *Scripts\chooseGenre.js* file and examine the code.

```
$(function () {

    $('#genreDialog').dialog({

        autoOpen: false,

        width: 400,

        height: 300,

        modal: true,

        title: 'Add Genre',

        buttons: {

            'Save': function () {

                // Omitted

            },

            'Cancel': function () {

                $(this).dialog('close');

            }

        }

    });
```

The second line uses the ID `genreDialog` to create a dialog box on the div tag in the *Views\StoreManager_ChooseGenre.cshtml* file. Most of the named parameters are self explanatory. The `autoOpen` parameter is set to false, selecting the **Create Genre** button will open the dialogue explicitly (this is described latter on). The dialog has two buttons, **Save** and **Cancel**. The **Cancel** button closes the dialog. The following code shows the **Save** button function.

```
'Save': function () {

    var createGenreForm = $('#createGenreForm');

    if (createGenreForm.valid()) {

        $.post(createGenreForm.attr('action'), createGenreForm.serialize(), function (data) {

            if (data.Error != '') {

                alert(data.Error);

            }

            else {

                // Add the new genre to the dropdown list and select it

                $('#GenreId').append(

                        $('<option></option>')

                            .val(data.Genre.GenreId)

                            .html(data.Genre.Name)

                            .prop('selected', true)  // Selects the new Genre in the DropDown LB

                    );

                $('#genreDialog').dialog('close');

            }

        });

    }

},
```

The `var createGenreForm` is selected from the `createGenreForm` ID. The `createGenreForm` ID was set in the following code found in the *Views\Genre_CreateGenre.cshtml* file.

```
@model MvcMusicStore.Models.Genre

@using (Html.BeginForm("Create", "Genre", FormMethod.Post, new { id = "createGenreForm" }))

{

    @*Omitted for clarity.*@

}
```

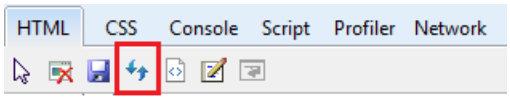
The Html.BeginForm helper overload used in the *Views\Genre_CreateGenre.cshtml* file generates HTML with an action attribute containing the URL to submit the form. You can see this by displaying the create album page in a browser and selecting show source in the browser. The following markup shows the generated HTML containing the form tag.

```
<form action="/StoreManager/Create" method="post">
```

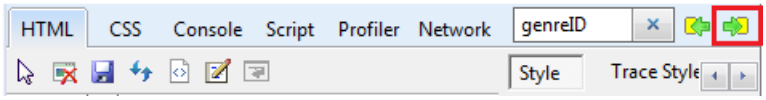
The jQuery `$.post` line makes an AJAX call to the action attribute (`/StoreManager/Create`) and passes in the data

from the **Create Genre** dialog box. The data consists of the name for the new genre and an optional description. If the AJAX call is successful, the new genre name and value are added to the Select markup, and the new genre is set to the selected value. Because this is dynamically generated markup, you can't see the new select option by viewing the source in the browser. You can see the new HTML with the IE 9 F12 developer tools. To view the new select option, in Internet Explorer 9, hit the F12 key to start the F12 developer tools. Navigate to the Create page and add a new genre so the new genre is selected in the genre select list. In the F12 developer tools:

1. Select the HTML tab.
2. Hit the refresh icon.



3. In the search box, enter GenreID.
4. Using the next icon,



navigate to the following select tag:

```
<select name="GenreId" id="GenreId" >
```

5. Expand the last option value.

The following code in the *Scripts\chooseGenre.js* file shows the how the **Add New Genre** button gets connected

to the click event, and how the **Add New Genre** dialog box is created.

```
$('#genreAddLink').click(function () {

    var createFormUrl = $(this).attr('href');

    $('#genreDialog').html('')

    .load(createFormUrl, function () {

        // The createGenreForm is loaded on the fly using jQuery load.

        // In order to have client validation working it is necessary to tell the

        // jQuery.validator to parse the newly added content

        jQuery.validator.unobtrusive.parse('#createGenreForm');

        $('#genreDialog').dialog('open');

    });

    return false;

});
```

The first line creates a click function attached to the **Add New Genre** button. The following markup from the Views\StoreManager_ChooseGenre.cshtml file shows how the **Add New Genre** button is created:

```
<a class="button" href="@Url.Content("~/Genre/Create")"

    id="genreAddLink">Add New Genre</a>
```

The load method creates and opens the Add Genre dialog and calls the jQuery `parse` method so client validation occurs on data entered in the dialog.

In this section you have learned how to create a dialog that can be used to add new category data to a select list. You can follow the same procedure to create UI to add a new artist to the artist select list. This tutorial has given an overview of working with the ASP.NET MVC HTML helper **DropDownList**. For additional information on working with the **DropDownList**, see the addition references section below. Please let us know if this tutorial has been helpful.

Rick.Anderson[at]Microsoft.com

Additional References

- ASP.NET MVC–Cascading Dropdown Lists Tutorial by Radu Enuca
- Chosen A JavaScript plugin that support multi-select and filtering.

Contributors

- Radu Enuca
- Jean-Sébastien Goupil
- Brad Wilson

Reviewers

- Jean-Sébastien Goupil
- Brad Wilson
- Mike Pope
- Tom Dykstra

PREVIOUS

Older Versions - MVC 1 and 2

2/20/2020 • 3 minutes to read • Edit Online

Resources for learning how to program MVC 1 and 2.

- Overview
  - ASP.NET MVC Overview
  - Understanding the ASP.NET MVC Execution Process
  - Understanding Models, Views, and Controllers (C#)
  - Understanding Models, Views, and Controllers (VB)
- Getting Started With MVC
  - Intro to ASP.NET MVC
  - Adding a Controller
  - Adding a View
  - Creating a Database
  - Accessing your Model's Data from a Controller
  - Adding a Create Method and Create View
  - Adding Validation to the Model
  - Adding a Column to the Model
- Models (Data)
  - Creating Model Classes with the Entity Framework (C#)
  - Creating Model Classes with LINQ to SQL (C#)
  - Displaying a Table of Database Data (C#)
  - Performing Simple Validation (C#)
  - Validating with the IDataErrorInfo Interface (C#)
  - Validating with a Service Layer (C#)
  - Validation with the Data Annotation Validators (C#)
  - Creating Model Classes with the Entity Framework (VB)
  - Creating Model Classes with LINQ to SQL (VB)
  - Displaying a Table of Database Data (VB)
  - Performing Simple Validation (VB)
  - Validating with the IDataErrorInfo Interface (VB)
  - Validating with a Service Layer (VB)
  - Validation with the Data Annotation Validators (VB)
- ASP.NET MVC Views
  - ASP.NET MVC Views Overview (C#)
  - Creating Custom HTML Helpers (C#)
  - TagBuilder Class to Build HTML Helpers (C#)
  - Creating Page Layouts with View Master Pages (C#)
  - Passing Data to View Master Pages (C#)
  - ASP.NET MVC Views Overview (VB)
  - Creating Custom HTML Helpers (VB)

  - TagBuilder Class to Build HTML Helpers (VB)
  - Creating Page Layouts with View Master Pages (VB)
  - Passing Data to View Master Pages (VB)
- Controllers and Routing
  - ASP.NET MVC Routing Overview (C#)
  - Understanding Action Filters (C#)
  - Improving Performance with Output Caching (C#)
  - Adding Dynamic Content to a Cached Page (C#)
  - Creating a Controller (C#)
  - Creating an Action (C#)
  - ASP.NET MVC Routing Overview (VB)
  - Understanding Action Filters (VB)
  - Improving Performance with Output Caching (VB)
  - Adding Dynamic Content to a Cached Page (VB)
  - Creating a Controller (VB)
  - Creating an Action (VB)
  - ASP.NET MVC Controller Overview (C#)
  - Creating Custom Routes (C#)
  - Creating a Route Constraint (C#)
  - Creating a Custom Route Constraint (C#)
  - ASP.NET MVC Controller Overview (VB)
  - Creating Custom Routes (VB)
  - Creating a Route Constraint (VB)
  - Creating a Custom Route Constraint (VB)
- Deployment
  - Using ASP.NET MVC with Different Versions of IIS (C#)
  - Using ASP.NET MVC with Different Versions of IIS (VB)
- Security
  - Authenticating Users with Forms Authentication (C#)
  - Authenticating Users with Windows Authentication (C#)
  - Preventing JavaScript Injection Attacks (C#)
  - Authenticating Users with Forms Authentication (VB)
  - Authenticating Users with Windows Authentication (VB)
  - Preventing JavaScript Injection Attacks (VB)
- Unit Testing
  - Creating Unit Tests for ASP.NET MVC Applications (C#)
  - Creating Unit Tests for ASP.NET MVC Applications (VB)
- Contact Manager
  - Iteration #1 – Create the Application (C#)
  - Iteration #2 – Make the application look nice (C#)
  - Iteration #3 – Add form validation (C#)
  - Iteration #4 – Make the application loosely coupled (C#)
  - Iteration #5 – Create unit tests (C#)
  - Iteration #6 – Use test-driven development (C#)

- Iteration #7 – Add Ajax functionality (C#)
- Iteration #1 – Create the Application (VB)
- Iteration #2 – Make the application look nice (VB)
- Iteration #3 – Add form validation (VB)
- Iteration #4 – Make the application loosely coupled (VB)
- Iteration #5 – Create unit tests (VB)
- Iteration #6 – Use test-driven development (VB)
- Iteration #7 – Add Ajax functionality (VB)

- NerdDinner
  - Introducing the NerdDinner Tutorial
  - Create a New ASP.NET MVC Project
  - Create a Database
  - Build a Model with Business Rule Validations
  - Use Controllers and Views to Implement a Listing/Details UI
  - Provide CRUD (Create, Read, Update, Delete) Data Form Entry Support
  - Use ViewData and Implement ViewModel Classes
  - Re-use UI Using Master Pages and Partials
  - Implement Efficient Data Paging
  - Secure Applications Using Authentication and Authorization
  - Use AJAX to Deliver Dynamic Updates
  - Use AJAX to Implement Mapping Scenarios
  - Enable Automated Unit Testing
  - NerdDinner Wrap Up
- Movie Database
  - Create a Movie Database Application in 15 Minutes with ASP.NET MVC (C#)
  - Create a Movie Database Application in 15 Minutes with ASP.NET MVC (VB)

Overview

2/20/2020 • 2 minutes to read • Edit Online

Learn what MVC is, how it differs from Web Forms, and how to create an ASP.NET application.

- ASP.NET MVC Overview
- Understanding the ASP.NET MVC Execution Process
- Understanding Models, Views, and Controllers (C#)
- Understanding Models, Views, and Controllers (VB)

ASP.NET MVC Overview

4/10/2020 • 7 minutes to read • Edit Online

by Microsoft

> Learn about the differences between ASP.NET MVC application and ASP.NET Web Forms applications. Learn how to decide when to build an ASP.NET MVC application.

The Model-View-Controller (MVC) architectural pattern separates an application into three main components: the model, the view, and the controller. The ASP.NET MVC framework provides an alternative to the ASP.NET Web Forms pattern for creating MVC-based Web applications. The ASP.NET MVC framework is a lightweight, highly testable presentation framework that (as with Web Forms-based applications) is integrated with existing ASP.NET features, such as master pages and membership-based authentication. The MVC framework is defined in the **System.Web.Mvc** namespace and is a fundamental, supported part of the **System.Web** namespace.

MVC is a standard design pattern that many developers are familiar with. Some types of Web applications will benefit from the MVC framework. Others will continue to use the traditional ASP.NET application pattern that is based on Web Forms and postbacks. Other types of Web applications will combine the two approaches; neither approach excludes the other.

The MVC framework includes the following components:

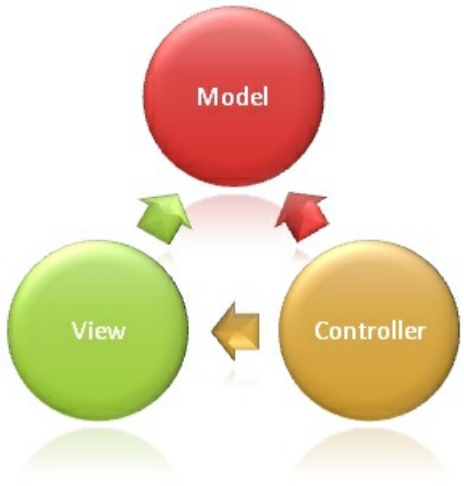


Figure 01: Invoking a controller action that expects a parameter value (Click to view full-size image)

- **Models**. Model objects are the parts of the application that implement the logic for the application s data domain. Often, model objects retrieve and store model state in a database. For example, a Product object might retrieve information from a database, operate on it, and then write updated information back to a Products table in SQL Server.

In small applications, the model is often a conceptual separation instead of a physical one. For example, if the application only reads a data set and sends it to the view, the application does not have a physical model layer and associated classes. In that case, the data set takes on the role of a model object.

- **Views**. Views are the components that display the application s user interface (UI). Typically, this UI is created from the model data. An example would be an edit view of a Products table that displays text boxes, drop-down lists, and check boxes based on the current state of a Products object.
- **Controllers**. Controllers are the components that handle user interaction, work with the model, and

ultimately select a view to render that displays UI. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. For example, the controller handles query-string values, and passes these values to the model, which in turn queries the database by using the values.

The MVC pattern helps you create applications that separate the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements. The pattern specifies where each kind of logic should be located in the application. The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps you manage complexity when you build an application, because it enables you to focus on one aspect of the implementation at a time. For example, you can focus on the view without depending on the business logic.

In addition to managing complexity, the MVC pattern makes it easier to test applications than it is to test a Web Forms-based ASP.NET Web application. For example, in a Web Forms-based ASP.NET Web application, a single class is used both to display output and to respond to user input. Writing automated tests for Web Forms-based ASP.NET applications can be complex, because to test an individual page, you must instantiate the page class, all its child controls, and additional dependent classes in the application. Because so many classes are instantiated to run the page, it can be hard to write tests that focus exclusively on individual parts of the application. Tests for Web Forms-based ASP.NET applications can therefore be more difficult to implement than tests in an MVC application. Moreover, tests in a Web Forms-based ASP.NET application require a Web server. The MVC framework decouples the components and makes heavy use of interfaces, which makes it possible to test individual components in isolation from the rest of the framework.

The loose coupling between the three main components of an MVC application also promotes parallel development. For instance, one developer can work on the view, a second developer can work on the controller logic, and a third developer can focus on the business logic in the model.

Deciding When to Create an MVC Application

You must consider carefully whether to implement a Web application by using either the ASP.NET MVC framework or the ASP.NET Web Forms model. The MVC framework does not replace the Web Forms model; you can use either framework for Web applications. (If you have existing Web Forms-based applications, these continue to work exactly as they always have.)

Before you decide to use the MVC framework or the Web Forms model for a specific Web site, weigh the advantages of each approach.

Advantages of an MVC-Based Web Application

The ASP.NET MVC framework offers the following advantages:

- It makes it easier to manage complexity by dividing an application into the model, the view, and the controller.
- It does not use view state or server-based forms. This makes the MVC framework ideal for developers who want full control over the behavior of an application.
- It uses a Front Controller pattern that processes Web application requests through a single controller. This enables you to design an application that supports a rich routing infrastructure. For more information, see Front Controller on the MSDN Web site.
- It provides better support for test-driven development (TDD).
- It works well for Web applications that are supported by large teams of developers and Web designers who need a high degree of control over the application behavior.

Advantages of a Web Forms-Based Web Application

The Web Forms-based framework offers the following advantages:

- It supports an event model that preserves state over HTTP, which benefits line-of-business Web application development. The Web Forms-based application provides dozens of events that are supported in hundreds of

server controls.

- It uses a Page Controller pattern that adds functionality to individual pages. For more information, see Page Controller on the MSDN Web site.
- It uses view state or server-based forms, which can make managing state information easier.
- It works well for small teams of Web developers and designers who want to take advantage of the large number of components available for rapid application development.
- In general, it is less complex for application development, because the components (the **Page** class, controls, and so on) are tightly integrated and usually require less code than the MVC model.

Features of the ASP.NET MVC Framework

The ASP.NET MVC framework provides the following features:

- Separation of application tasks (input logic, business logic, and UI logic), testability, and test-driven development (TDD) by default. All core contracts in the MVC framework are interface-based and can be tested by using mock objects, which are simulated objects that imitate the behavior of actual objects in the application. You can unit-test the application without having to run the controllers in an ASP.NET process, which makes unit testing fast and flexible. You can use any unit-testing framework that is compatible with the .NET Framework.
- An extensible and pluggable framework. The components of the ASP.NET MVC framework are designed so that they can be easily replaced or customized. You can plug in your own view engine, URL routing policy, action-method parameter serialization, and other components. The ASP.NET MVC framework also supports the use of Dependency Injection (DI) and Inversion of Control (IOC) container models. DI allows you to inject objects into a class, instead of relying on the class to create the object itself. IOC specifies that if an object requires another object, the first objects should get the second object from an outside source such as a configuration file. This makes testing easier.
- A powerful URL-mapping component that lets you build applications that have comprehensible and searchable URLs. URLs do not have to include file-name extensions, and are designed to support URL naming patterns that work well for search engine optimization (SEO) and representational state transfer (REST) addressing.
- Support for using the markup in existing ASP.NET page (.aspx files), user control (.ascx files), and master page (.master files) markup files as view templates. You can use existing ASP.NET features with the ASP.NET MVC framework, such as nested master pages, in-line expressions (<%= %>), declarative server controls, templates, data-binding, localization, and so on.
- Support for existing ASP.NET features. ASP.NET MVC lets you use features such as forms authentication and Windows authentication, URL authorization, membership and roles, output and data caching, session and profile state management, health monitoring, the configuration system, and the provider architecture.

Understanding the ASP.NET MVC Execution Process

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

> Learn how the ASP.NET MVC framework processes a browser request step-by-step.

Requests to an ASP.NET MVC-based Web application first pass through the **UrlRoutingModule** object, which is an HTTP module. This module parses the request and performs route selection. The **UrlRoutingModule** object selects the first route object that matches the current request. (A route object is a class that implements **RouteBase**, and is typically an instance of the **Route** class.) If no routes match, the **UrlRoutingModule** object does nothing and lets the request fall back to the regular ASP.NET or IIS request processing.

From the selected **Route** object, the **UrlRoutingModule** object obtains the **IRouteHandler** object that is associated with the **Route** object. Typically, in an MVC application, this will be an instance of **MvcRouteHandler**. The **IRouteHandler** instance creates an **IHttpHandler** object and passes it the **IHttpContext** object. By default, the **IHttpHandler** instance for MVC is the **MvcHandler** object. The **MvcHandler** object then selects the controller that will ultimately handle the request.

> **NOTE**
>
> When an ASP.NET MVC Web application runs in IIS 7.0, no file name extension is required for MVC projects. However, in IIS 6.0, the handler requires that you map the .mvc file name extension to the ASP.NET ISAPI DLL.

The module and handler are the entry points to the ASP.NET MVC framework. They perform the following actions:

- Select the appropriate controller in an MVC Web application.
- Obtain a specific controller instance.
- Call the controller's **Execute** method.

The following lists the stages of execution for an MVC Web project:

- Receive first request for the application
  - In the Global.asax file, **Route** objects are added to the **RouteTable** object.
- Perform routing
  - The **UrlRoutingModule** module uses the first matching **Route** object in the **RouteTable** collection to create the **RouteData** object, which it then uses to create a **RequestContext** (**IHttpContext**) object.
- Create MVC request handler
  - The **MvcRouteHandler** object creates an instance of the **MvcHandler** class and passes it the **RequestContext** instance.
- Create controller
  - The **MvcHandler** object uses the **RequestContext** instance to identify the **IControllerFactory** object (typically an instance of the **DefaultControllerFactory** class) to create the controller instance with.
- Execute controller - The **MvcHandler** instance calls the controller s **Execute** method. |
- Invoke action
  - Most controllers inherit from the **Controller** base class. For controllers that do so, the

ControllerActionInvoker object that is associated with the controller determines which action method of the controller class to call, and then calls that method.

- Execute result
  - A typical action method might receive user input, prepare the appropriate response data, and then execute the result by returning a result type. The built-in result types that can be executed include the following: **ViewResult** (which renders a view and is the most-often used result type), **RedirectToRouteResult**, **RedirectResult**, **ContentResult**, **JsonResult**, and **EmptyResult**.

Understanding Models, Views, and Controllers (C#)

9/16/2020 • 9 minutes to read • Edit Online

by Stephen Walther

> Confused about Models, Views, and Controllers? In this tutorial, Stephen Walther introduces you to the different parts of an ASP.NET MVC application.

This tutorial provides you with a high-level overview of ASP.NET MVC models, views, and controllers. In other words, it explains the M', V', and C' in ASP.NET MVC.

After reading this tutorial, you should understand how the different parts of an ASP.NET MVC application work together. You should also understand how the architecture of an ASP.NET MVC application differs from an ASP.NET Web Forms application or Active Server Pages application.

The Sample ASP.NET MVC Application

The default Visual Studio template for creating ASP.NET MVC Web Applications includes an extremely simple sample application that can be used to understand the different parts of an ASP.NET MVC application. We take advantage of this simple application in this tutorial.

You create a new ASP.NET MVC application with the MVC template by launching Visual Studio 2008 and selecting the menu option File, New Project (see Figure 1). In the New Project dialog, select your favorite programming language under Project Types (Visual Basic or C#) and select **ASP.NET MVC Web Application** under Templates. Click the OK button.

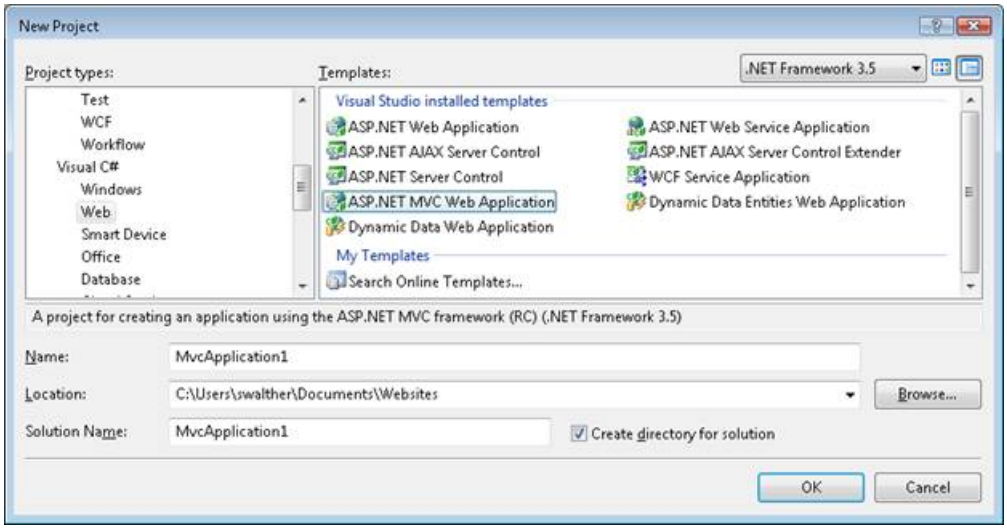


Figure 01: New Project Dialog (Click to view full-size image)

When you create a new ASP.NET MVC application, the **Create Unit Test Project** dialog appears (see Figure 2). This dialog enables you to create a separate project in your solution for testing your ASP.NET MVC application. Select the option **No, do not create a unit test project** and click the **OK** button.

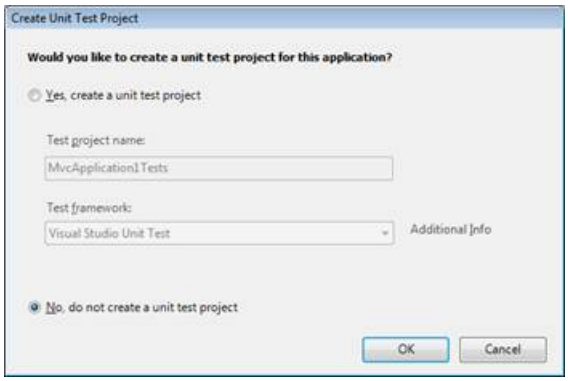


Figure 02: Create Unit Test Dialog (Click to view full-size image)

After the new ASP.NET MVC application is created. You will see several folders and files in the Solution Explorer window. In particular, you'll see three folders named Models, Views, and Controllers. As you might guess from the folder names, these folders contain the files for implementing models, views, and controllers.

If you expand the Controllers folder, you should see a file named AccountController.cs and a file named HomeController.cs. If you expand the Views folder, you should see three subfolders named Account, Home and Shared. If you expand the Home folder, you'll see two additional files named About.aspx and Index.aspx (see Figure 3). These files make up the sample application included with the default ASP.NET MVC template.

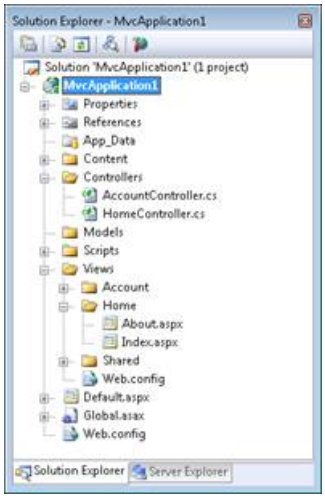


Figure 03: The Solution Explorer Window (Click to view full-size image)

You can run the sample application by selecting the menu option **Debug, Start Debugging**. Alternatively, you can press the F5 key.

When you first run an ASP.NET application, the dialog in Figure 4 appears that recommends that you enable debug mode. Click the OK button and the application will run.

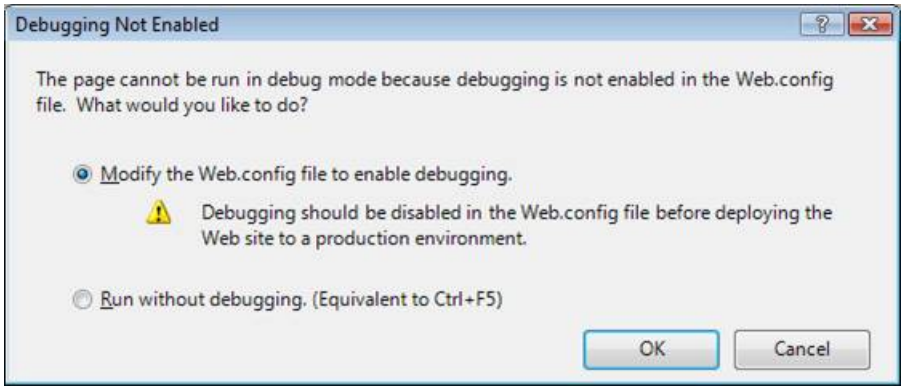


Figure 04: Debugging Not Enabled dialog (Click to view full-size image)

When you run an ASP.NET MVC application, Visual Studio launches the application in your web browser. The

sample application consists of only two pages: the Index page and the About page. When the application first starts, the Index page appears (see Figure 5). You can navigate to the About page by clicking the menu link at the top right of the application.

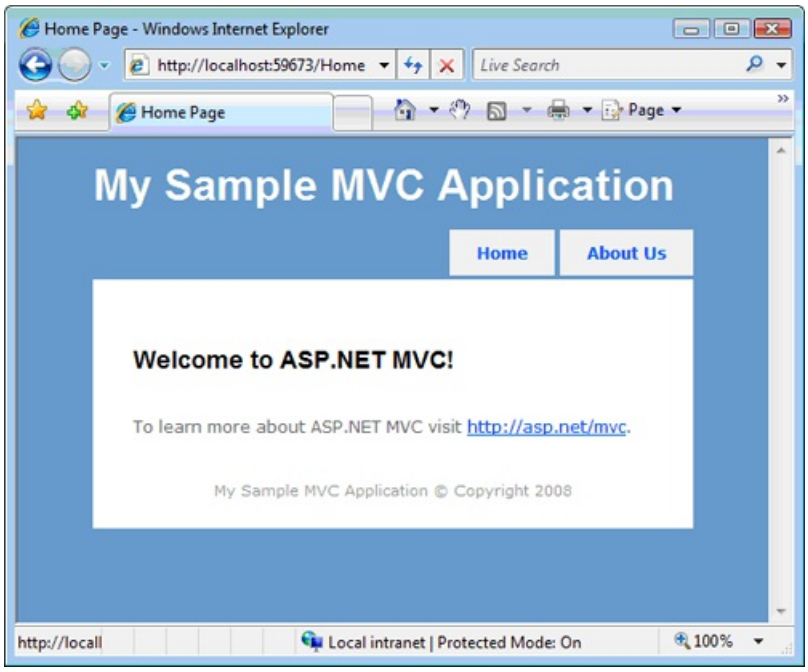


Figure 05: The Index Page (Click to view full-size image)

Notice the URLs in the address bar of your browser. For example, when you click the About menu link, the URL in the browser address bar changes to **/Home/About**.

If you close the browser window and return to Visual Studio, you won't be able to find a file with the path Home/About. The files don't exist. How is this possible?

A URL Does Not Equal a Page

When you build a traditional ASP.NET Web Forms application or an Active Server Pages application, there is a one-to-one correspondence between a URL and a page. If you request a page named SomePage.aspx from the server, then there had better be a page on disk named SomePage.aspx. If the SomePage.aspx file does not exist, you get an ugly **404 - Page Not Found** error.

When building an ASP.NET MVC application, in contrast, there is no correspondence between the URL that you type into your browser's address bar and the files that you find in your application. In an ASP.NET MVC application, a URL corresponds to a controller action instead of a page on disk.

In a traditional ASP.NET or ASP application, browser requests are mapped to pages. In an ASP.NET MVC application, in contrast, browser requests are mapped to controller actions. An ASP.NET Web Forms application is content-centric. An ASP.NET MVC application, in contrast, is application logic centric.

Understanding ASP.NET Routing

A browser request gets mapped to a controller action through a feature of the ASP.NET framework called *ASP.NET Routing*. ASP.NET Routing is used by the ASP.NET MVC framework to *route* incoming requests to controller actions.

ASP.NET Routing uses a route table to handle incoming requests. This route table is created when your web application first starts. The route table is setup in the Global.asax file. The default MVC Global.asax file is contained in Listing 1.

Listing 1 - Global.asax

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcApplication1
{
    // Note: For instructions on enabling IIS6 or IIS7 classic mode,
    // visit https://go.microsoft.com/?LinkId=9394801

    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default",                                          // Route name
                "{controller}/{action}/{id}",                       // URL with parameters
                new { controller = "Home", action = "Index", id = "" }  // Parameter defaults
            );

        }

        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

When an ASP.NET application first starts, the Application_Start() method is called. In Listing 1, this method calls the RegisterRoutes() method and the RegisterRoutes() method creates the default route table.

The default route table consists of one route. This default route breaks all incoming requests into three segments (a URL segment is anything between forward slashes). The first segment is mapped to a controller name, the second segment is mapped to an action name, and the final segment is mapped to a parameter passed to the action named Id.

For example, consider the following URL:

/Product/Details/3

This URL is parsed into three parameters like this:

Controller = Product

Action = Details

Id = 3

The Default route defined in the Global.asax file includes default values for all three parameters. The default Controller is Home, the default Action is Index, and the default Id is an empty string. With these defaults in mind, consider how the following URL is parsed:

/Employee

This URL is parsed into three parameters like this:

Controller = Employee

Action = Index

Id =

Finally, if you open an ASP.NET MVC Application without supplying any URL (for example, `http://localhost`) then the URL is parsed like this:

Controller = Home

Action = Index

Id =

The request is routed to the Index() action on the HomeController class.

Understanding Controllers

A controller is responsible for controlling the way that a user interacts with an MVC application. A controller contains the flow control logic for an ASP.NET MVC application. A controller determines what response to send back to a user when a user makes a browser request.

A controller is just a class (for example, a Visual Basic or C# class). The sample ASP.NET MVC application includes a controller named HomeController.cs located in the Controllers folder. The content of the HomeController.cs file is reproduced in Listing 2.

Listing 2 - HomeController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewData["Title"] = "Home Page";
            ViewData["Message"] = "Welcome to ASP.NET MVC!";

            return View();
        }

        public ActionResult About()
        {
            ViewData["Title"] = "About Page";

            return View();
        }
    }
}
```

Notice that the HomeController has two methods named Index() and About(). These two methods correspond to the two actions exposed by the controller. The URL /Home/Index invokes the HomeController.Index() method and the URL /Home/About invokes the HomeController.About() method.

Any public method in a controller is exposed as a controller action. You need to be careful about this. This means that any public method contained in a controller can be invoked by anyone with access to the Internet by entering the right URL into a browser.

Understanding Views

The two controller actions exposed by the HomeController class, Index() and About(), both return a view. A view contains the HTML markup and content that is sent to the browser. A view is the equivalent of a page when working with an ASP.NET MVC application.

You must create your views in the right location. The HomeController.Index() action returns a view located at the following path:

\Views\Home\Index.aspx

The HomeController.About() action returns a view located at the following path:

\Views\Home\About.aspx

In general, if you want to return a view for a controller action, then you need to create a subfolder in the Views folder with the same name as your controller. Within the subfolder, you must create an .aspx file with the same name as the controller action.

The file in Listing 3 contains the About.aspx view.

Listing 3 - About.aspx

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage" %>

<asp:Content ID="aboutContent" ContentPlaceHolderID="MainContent" runat="server">
    <h2>About</h2>
    <p>
        Put content here.
    </p>
</asp:Content>
```

If you ignore the first line in Listing 3, most of the rest of the view consists of standard HTML. You can modify the contents of the view by entering any HTML that you want here.

A view is very similar to a page in Active Server Pages or ASP.NET Web Forms. A view can contain HTML content and scripts. You can write the scripts in your favorite .NET programming language (for example, C# or Visual Basic .NET). You use scripts to display dynamic content such as database data.

Understanding Models

We have discussed controllers and we have discussed views. The last topic that we need to discuss is models. What is an MVC model?

An MVC model contains all of your application logic that is not contained in a view or a controller. The model should contain all of your application business logic, validation logic, and database access logic. For example, if you are using the Microsoft Entity Framework to access your database, then you would create your Entity Framework classes (your .edmx file) in the Models folder.

A view should contain only logic related to generating the user interface. A controller should only contain the bare minimum of logic required to return the right view or redirect the user to another action (flow control). Everything else should be contained in the model.

In general, you should strive for fat models and skinny controllers. Your controller methods should contain only a few lines of code. If a controller action gets too fat, then you should consider moving the logic out to a new class in the Models folder.

Summary

This tutorial provided you with a high level overview of the different parts of an ASP.NET MVC web application. You learned how ASP.NET Routing maps incoming browser requests to particular controller actions. You learned how controllers orchestrate how views are returned to the browser. Finally, you learned how models contain application business, validation, and database access logic.

Understanding Models, Views, and Controllers (VB)

9/16/2020 • 9 minutes to read • Edit Online

by Stephen Walther

> Confused about Models, Views, and Controllers? In this tutorial, Stephen Walther introduces you to the different parts of an ASP.NET MVC application.

This tutorial provides you with a high-level overview of ASP.NET MVC models, views, and controllers. In other words, it explains the M', V', and C' in ASP.NET MVC.

After reading this tutorial, you should understand how the different parts of an ASP.NET MVC application work together. You should also understand how the architecture of an ASP.NET MVC application differs from an ASP.NET Web Forms application or Active Server Pages application.

The Sample ASP.NET MVC Application

The default Visual Studio template for creating ASP.NET MVC Web Applications includes an extremely simple sample application that can be used to understand the different parts of an ASP.NET MVC application. We take advantage of this simple application in this tutorial.

You create a new ASP.NET MVC application with the MVC template by launching Visual Studio 2008 and selecting the menu option File, New Project (see Figure 1). In the New Project dialog, select your favorite programming language under Project Types (Visual Basic or C#) and select **ASP.NET MVC Web Application** under Templates. Click the OK button.

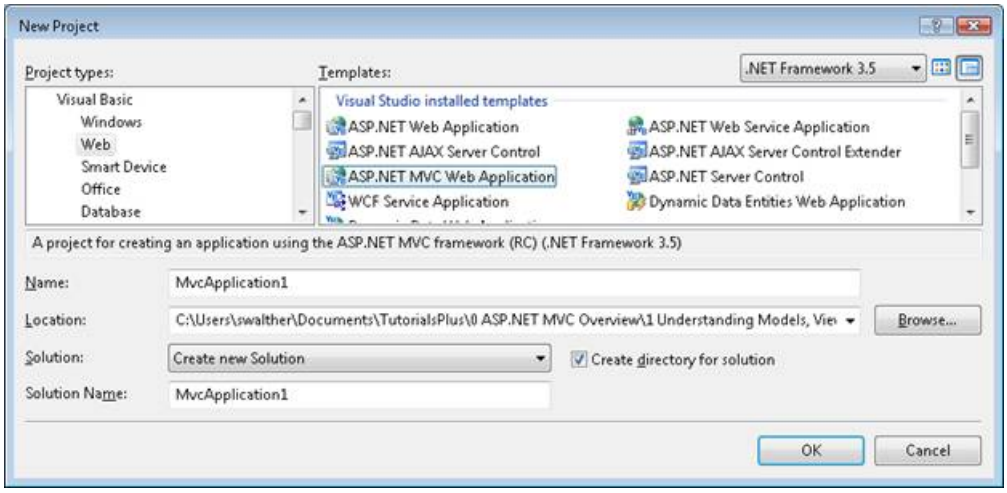


Figure 01: New Project Dialog (Click to view full-size image)

When you create a new ASP.NET MVC application, the **Create Unit Test Project** dialog appears (see Figure 2). This dialog enables you to create a separate project in your solution for testing your ASP.NET MVC application. Select the option **No, do not create a unit test project** and click the **OK** button.

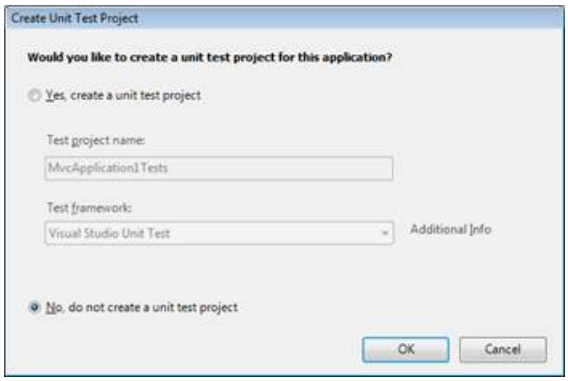


Figure 02: Create Unit Test Dialog (Click to view full-size image)

After the new ASP.NET MVC application is created. You will see several folders and files in the Solution Explorer window. In particular, you'll see three folders named Models, Views, and Controllers. As you might guess from the folder names, these folders contain the files for implementing models, views, and controllers.

If you expand the Controllers folder, you should see a file named AccountController.vb and a file named HomeController.vb. If you expand the Views folder, you should see three subfolders named Account, Home and Shared. If you expand the Home folder, you'll see two additional files named About.aspx and Index.aspx (see Figure 3). These files make up the sample application included with the default ASP.NET MVC template.

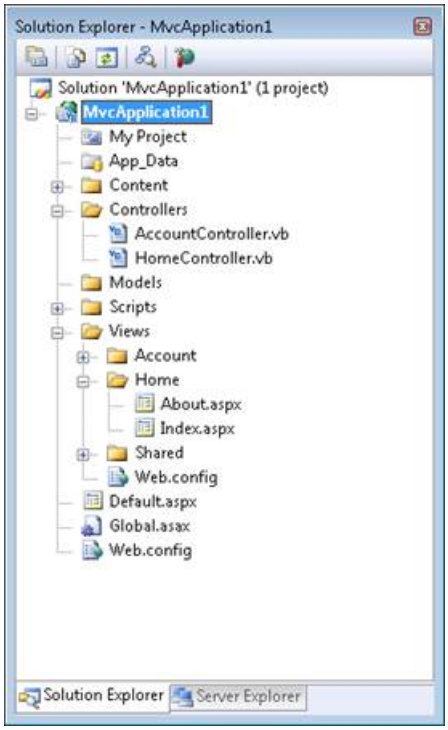


Figure 03: The Solution Explorer Window (Click to view full-size image)

You can run the sample application by selecting the menu option **Debug, Start Debugging**. Alternatively, you can press the F5 key.

When you first run an ASP.NET application, the dialog in Figure 4 appears that recommends that you enable debug mode. Click the OK button and the application will run.

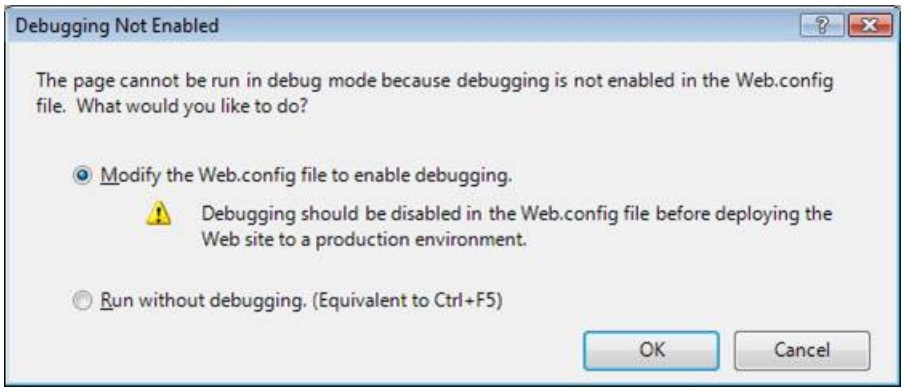


Figure 04: Debugging Not Enabled dialog (Click to view full-size image)

When you run an ASP.NET MVC application, Visual Studio launches the application in your web browser. The sample application consists of only two pages: the Index page and the About page. When the application first starts, the Index page appears (see Figure 5). You can navigate to the About page by clicking the menu link at the top right of the application.

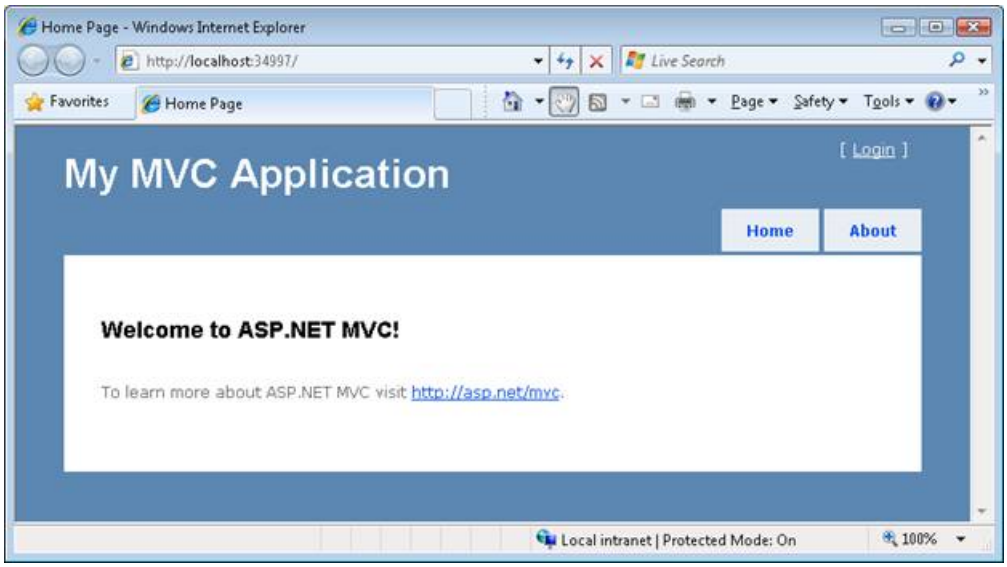


Figure 05: The Index Page (Click to view full-size image)

Notice the URLs in the address bar of your browser. For example, when you click the About menu link, the URL in the browser address bar changes to **/Home/About**.

If you close the browser window and return to Visual Studio, you won't be able to find a file with the path Home/About. The files don't exist. How is this possible?

A URL Does Not Equal a Page

When you build a traditional ASP.NET Web Forms application or an Active Server Pages application, there is a one-to-one correspondence between a URL and a page. If you request a page named SomePage.aspx from the server, then there had better be a page on disk named SomePage.aspx. If the SomePage.aspx file does not exist, you get an ugly **404 - Page Not Found** error.

When building an ASP.NET MVC application, in contrast, there is no correspondence between the URL that you type into your browser's address bar and the files that you find in your application. In an ASP.NET MVC application, a URL corresponds to a controller action instead of a page on disk.

In a traditional ASP.NET or ASP application, browser requests are mapped to pages. In an ASP.NET MVC application, in contrast, browser requests are mapped to controller actions. An ASP.NET Web Forms application is content-centric. An ASP.NET MVC application, in contrast, is application logic centric.

Understanding ASP.NET Routing

A browser request gets mapped to a controller action through a feature of the ASP.NET framework called *ASP.NET Routing*. ASP.NET Routing is used by the ASP.NET MVC framework to *route* incoming requests to controller actions.

ASP.NET Routing uses a route table to handle incoming requests. This route table is created when your web application first starts. The route table is setup in the Global.asax file. The default MVC Global.asax file is contained in Listing 1.

Listing 1 - Global.asax

```
' Note: For instructions on enabling IIS6 or IIS7 classic mode,
' visit https://go.microsoft.com/?LinkId=9394802

Public Class MvcApplication
    Inherits System.Web.HttpApplication

    Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}")

        ' MapRoute takes the following parameters, in order:
        ' (1) Route name
        ' (2) URL with parameters
        ' (3) Parameter defaults
        routes.MapRoute( _
            "Default", _
            "{controller}/{action}/{id}", _
            New With {.controller = "Home", .action = "Index", .id = ""} _
        )

    End Sub

    Sub Application_Start()
        RegisterRoutes(RouteTable.Routes)
    End Sub
End Class
```

When an ASP.NET application first starts, the Application_Start() method is called. In Listing 1, this method calls the RegisterRoutes() method and the RegisterRoutes() method creates the default route table.

The default route table consists of one route. This default route breaks all incoming requests into three segments (a URL segment is anything between forward slashes). The first segment is mapped to a controller name, the second segment is mapped to an action name, and the final segment is mapped to a parameter passed to the action named Id.

For example, consider the following URL:

/Product/Details/3

This URL is parsed into three parameters like this:

Controller = Product

Action = Details

Id = 3

The Default route defined in the Global.asax file includes default values for all three parameters. The default Controller is Home, the default Action is Index, and the default Id is an empty string. With these defaults in mind, consider how the following URL is parsed:

/Employee

This URL is parsed into three parameters like this:

Controller = Employee

Action = Index

Id =

Finally, if you open an ASP.NET MVC Application without supplying any URL (for example, `http://localhost`) then the URL is parsed like this:

Controller = Home

Action = Index

Id =

The request is routed to the Index() action on the HomeController class.

Understanding Controllers

A controller is responsible for controlling the way that a user interacts with an MVC application. A controller contains the flow control logic for an ASP.NET MVC application. A controller determines what response to send back to a user when a user makes a browser request.

A controller is just a class (for example, a Visual Basic or C# class). The sample ASP.NET MVC application includes a controller named HomeController.vb located in the Controllers folder. The content of the HomeController.vb file is reproduced in Listing 2.

Listing 2 - HomeController.cs

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Function Index()
        ViewData("Title") = "Home Page"
        ViewData("Message") = "Welcome to ASP.NET MVC!"

        Return View()
    End Function

    Function About()
        ViewData("Title") = "About Page"

        Return View()
    End Function
End Class
```

Notice that the HomeController has two methods named Index() and About(). These two methods correspond to the two actions exposed by the controller. The URL /Home/Index invokes the HomeController.Index() method and the URL /Home/About invokes the HomeController.About() method.

Any public method in a controller is exposed as a controller action. You need to be careful about this. This means that any public method contained in a controller can be invoked by anyone with access to the Internet by entering the right URL into a browser.

Understanding Views

The two controller actions exposed by the HomeController class, Index() and About(), both return a view. A view contains the HTML markup and content that is sent to the browser. A view is the equivalent of a page when

working with an ASP.NET MVC application.

You must create your views in the right location. The HomeController.Index() action returns a view located at the following path:

\Views\Home\Index.aspx

The HomeController.About() action returns a view located at the following path:

\Views\Home\About.aspx

In general, if you want to return a view for a controller action, then you need to create a subfolder in the Views folder with the same name as your controller. Within the subfolder, you must create an .aspx file with the same name as the controller action.

The file in Listing 3 contains the About.aspx view.

Listing 3 - About.aspx

```
<%@ Page Language="VB" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage" %>

<asp:Content ID="aboutContent" ContentPlaceHolderID="MainContent" runat="server">
    <h2>About</h2>
    <p>
        Put content here.
    </p>
</asp:Content>
```

If you ignore the first line in Listing 3, most of the rest of the view consists of standard HTML. You can modify the contents of the view by entering any HTML that you want here.

A view is very similar to a page in Active Server Pages or ASP.NET Web Forms. A view can contain HTML content and scripts. You can write the scripts in your favorite .NET programming language (for example, C# or Visual Basic .NET). You use scripts to display dynamic content such as database data.

Understanding Models

We have discussed controllers and we have discussed views. The last topic that we need to discuss is models. What is an MVC model?

An MVC model contains all of your application logic that is not contained in a view or a controller. The model should contain all of your application business logic, validation logic, and database access logic. For example, if you are using the Microsoft Entity Framework to access your database, then you would create your Entity Framework classes (your .edmx file) in the Models folder.

A view should contain only logic related to generating the user interface. A controller should only contain the bare minimum of logic required to return the right view or redirect the user to another action (flow control). Everything else should be contained in the model.

In general, you should strive for fat models and skinny controllers. Your controller methods should contain only a few lines of code. If a controller action gets too fat, then you should consider moving the logic out to a new class in the Models folder.

Summary

This tutorial provided you with a high level overview of the different parts of an ASP.NET MVC web application. You learned how ASP.NET Routing maps incoming browser requests to particular controller actions. You learned how controllers orchestrate how views are returned to the browser. Finally, you learned how models contain application

business, validation, and database access logic.

Getting Started With MVC

2/20/2020 • 2 minutes to read • Edit Online

This is a beginner tutorial series that introduces ASP.NET MVC. Create a simple web application that reads and writes from a database.

- Intro to ASP.NET MVC
- Adding a Controller
- Adding a View
- Creating a Database
- Accessing your Model's Data from a Controller
- Adding a Create Method and Create View
- Adding Validation to the Model
- Adding a Column to the Model

Intro to ASP.NET MVC

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

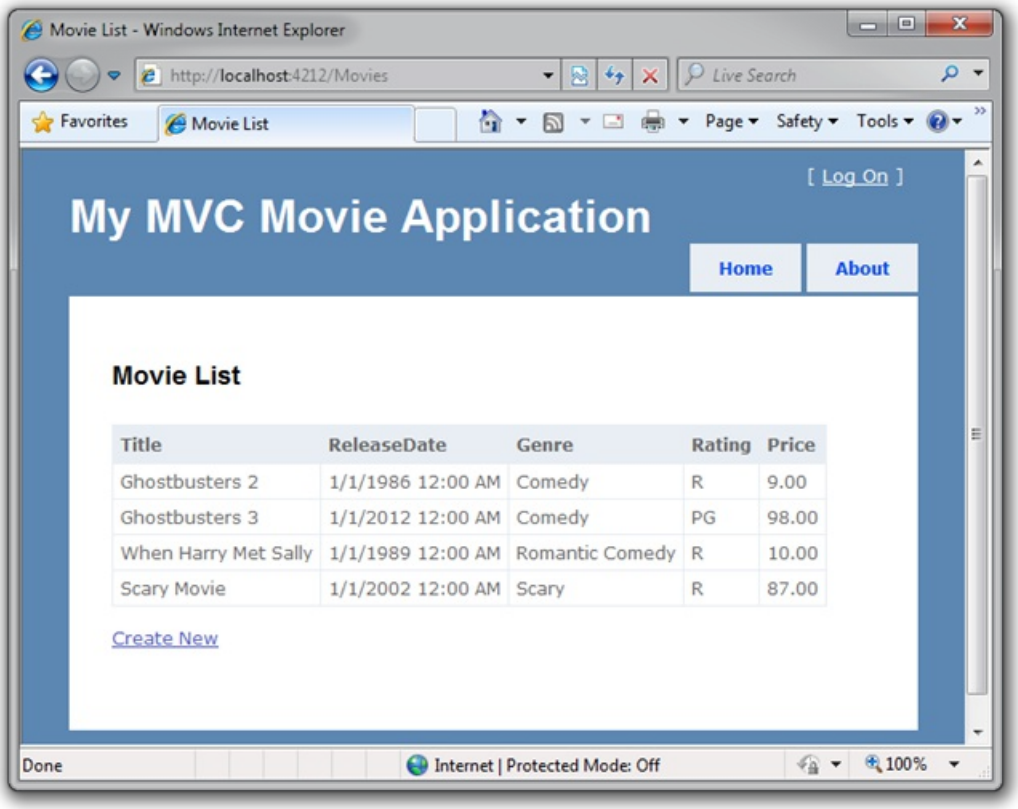
> **NOTE**
>
> An updated version if this tutorial is available here using Visual Studio 2013. The new tutorial uses ASP.NET MVC 5, which provides many improvements over this tutorial.

This is a beginner tutorial that introduces the basics of ASP.NET MVC. You'll create a simple web application that reads and writes from a database. Visit the ASP.NET MVC learning center to find other ASP.NET MVC tutorials and samples.

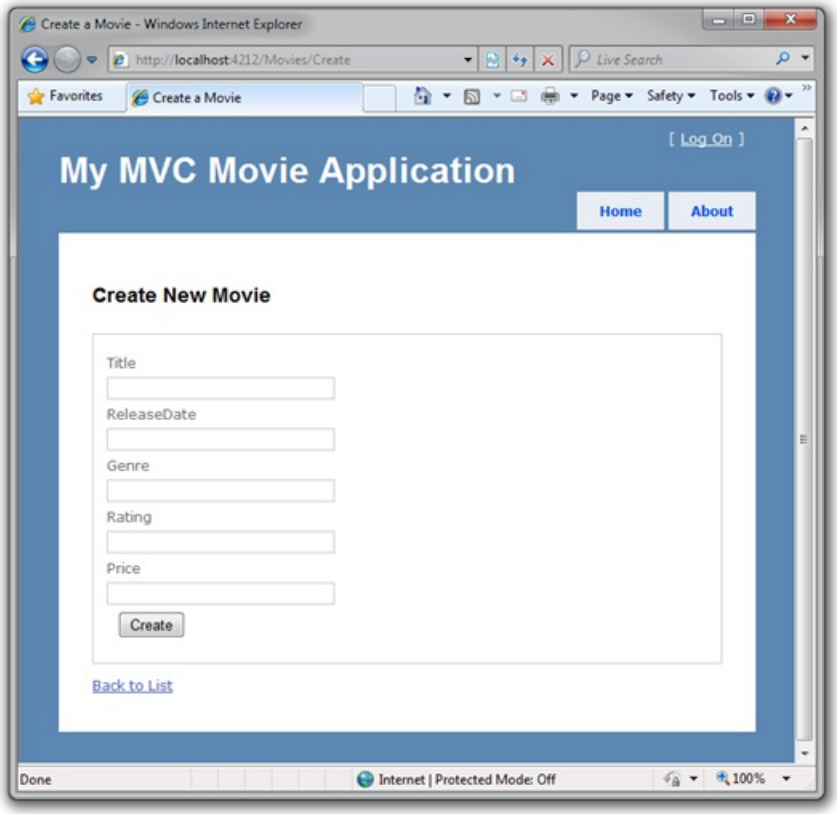
Let's make our first ASP.NET MVC Web Application using Visual Web Developer 2010 Express. We'll make a little Movie list application that will let us create and list movies.

What You'll Build

Here are two screenshots of the application you'll build. You will have a simple table of movies with various columns.



And you'll have a Create Form so we can add movies to the list.



Skills You'll Learn

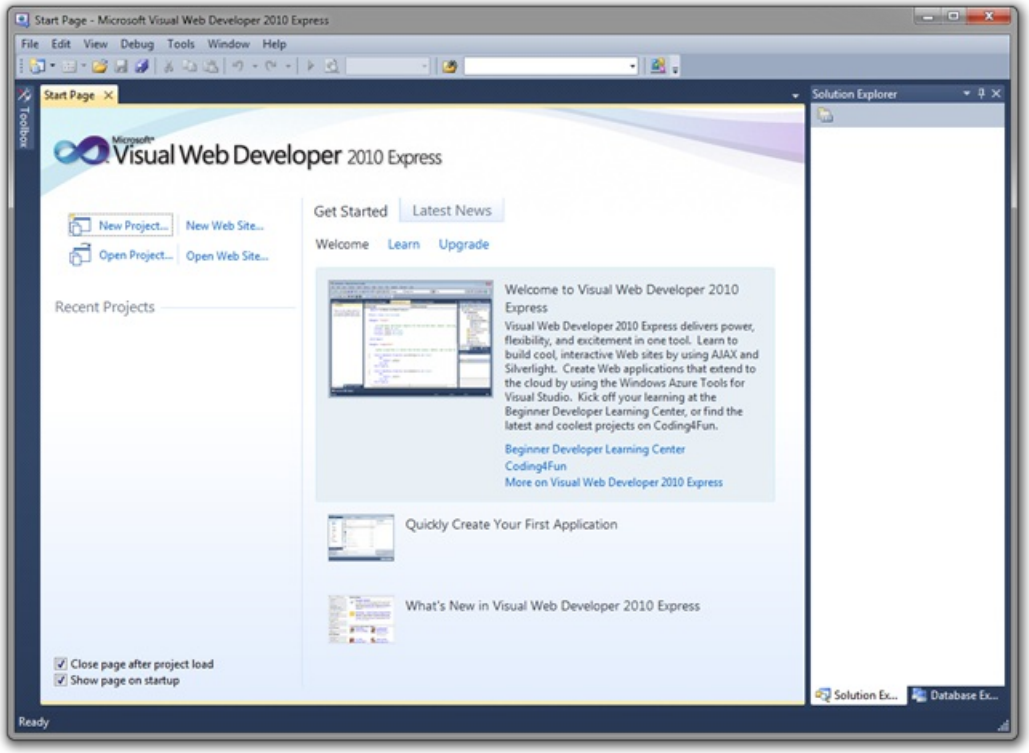
This tutorial will teach you the basics of building an ASP.NET MVC Web Application using Visual Studio. You'll learn:

- How to create a new ASP.NET MVC Project
- How to create a new Database with SQL Server
- How to create ASP.NET MVC Controllers and Views
- How to retrieve and display data
- How to edit data and enable data validation
- How to update the database schema

Get Started

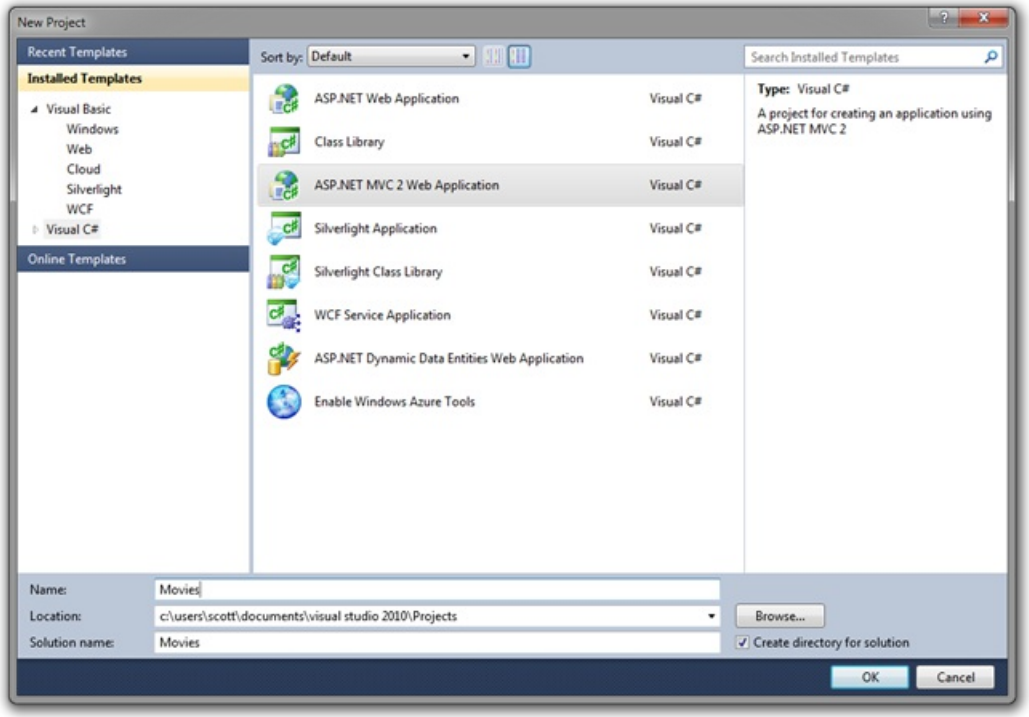
Start by running Visual Web Developer 2010 Express (I'll call it "VWD" from now on) and select New Project from the Start Screen.

Visual Web Developer is an IDE, or Integrated Developer Environment. Just like you use Microsoft Word to write documents, you'll use an IDE to create applications. There's a toolbar along the top showing various options available to you, as well as the menu you could also have used to Select File | New project.

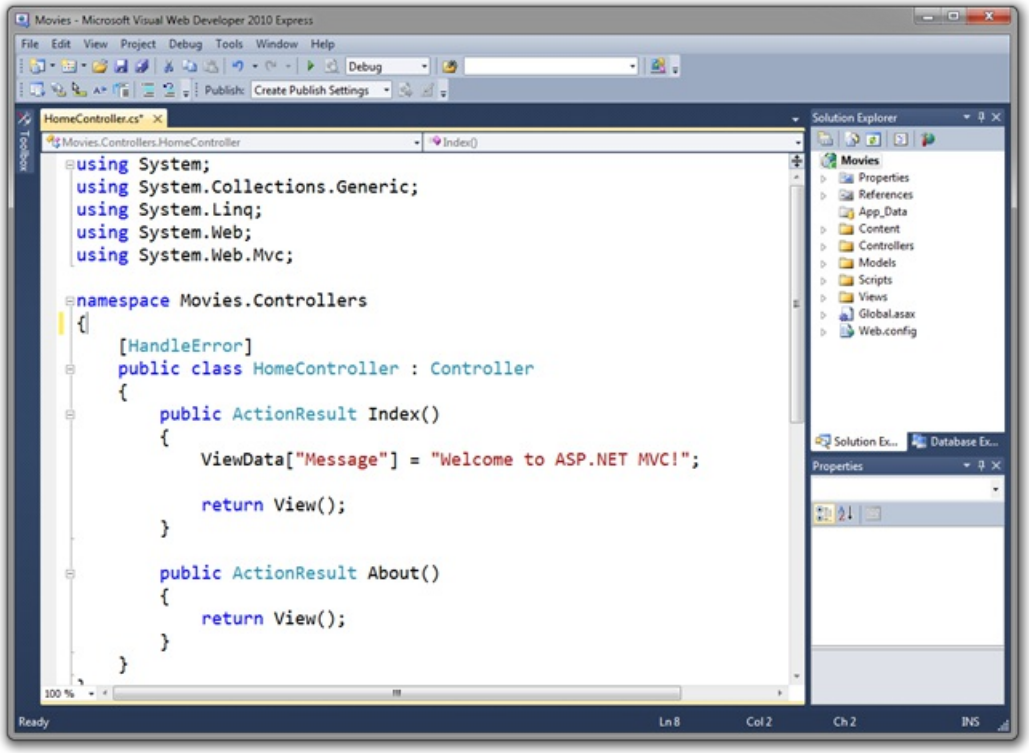


Creating Your First Application

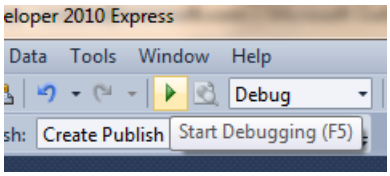
You can create applications using Visual Basic or Visual C#. For now, Select Visual C# on the left, then pick "ASP.NET MVC 2 Web Application." Name your project "Movies" and click OK.



On the right-hand side is the Solution Explorer showing all the files and folders in your application. The big window in the middle is where you edit your code and spend most of your time. Visual Studio used a default template for the ASP.NET MVC project you just created, so you have a working application right now without doing anything! This is a simple "Hello World! project, and it is a good place to start for our application.



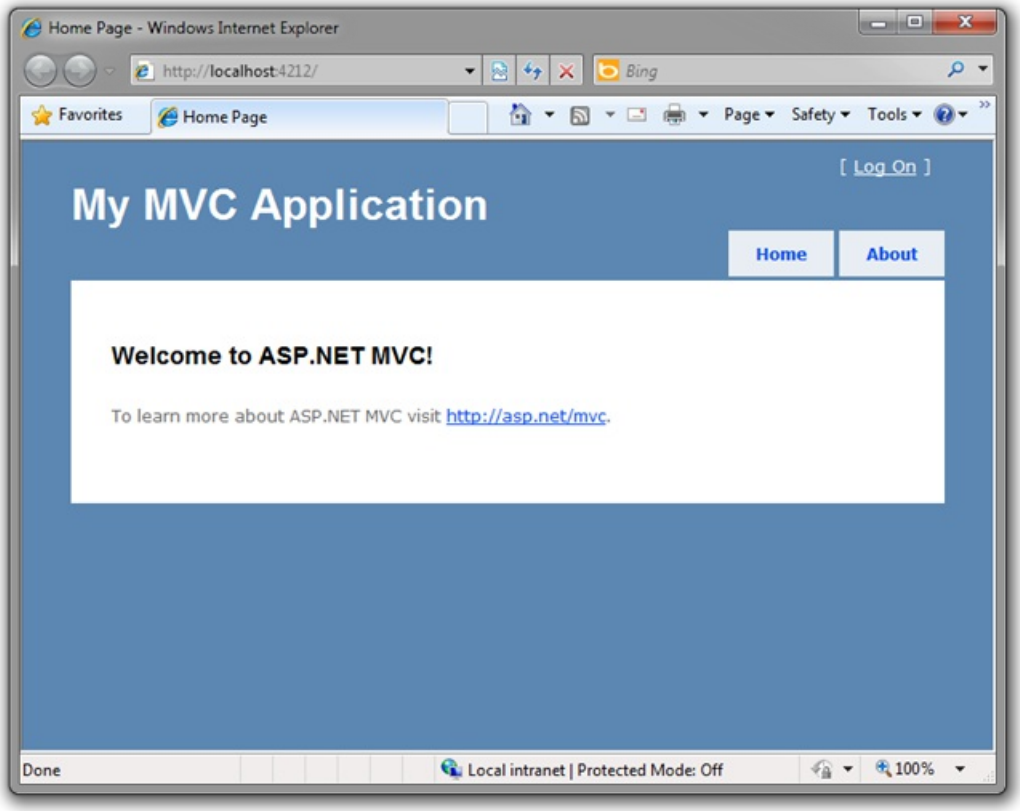
Select the "play" button to the toolbar.



It's a green arrow pointing to the right that will compile your program and start your application in a web browser.

NOTE: You can instead press F5 on your keyboard, or select Debug->Start Debugging from the "Debug" menu.

This will cause Visual Web Developer to start a development web-server and run our web application (there are no configuration or manual steps required to enable this). It will then launch a browser and configure it to browse the application's home-page. Notice below that the address bar of the browser says "localhost", and not something like example.com. That's because localhost always points to your own local computer - which in this case is running the application we just built.



Out of the box this default template gives you two pages to visit and a basic login page. Let's change how this application works and learn a little bit about ASP.NET MVC in the process. Close your browser and lets change some code.

NEXT

Adding a Controller

2/20/2020 • 3 minutes to read • Edit Online

by Scott Hanselman

> **NOTE**
>
> An updated version if this tutorial is available here using Visual Studio 2013. The new tutorial uses ASP.NET MVC 5, which provides many improvements over this tutorial.

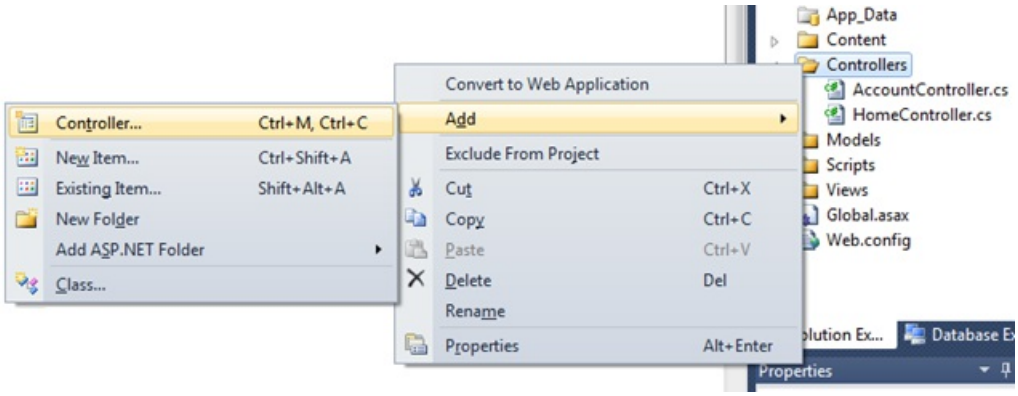
This is a beginner tutorial that introduces the basics of ASP.NET MVC. You'll create a simple web application that reads and writes from a database. Visit the ASP.NET MVC learning center to find other ASP.NET MVC tutorials and samples.

MVC stands for Model, View, Controller. MVC is a pattern for developing applications such that each part has a responsibility that is different from another.

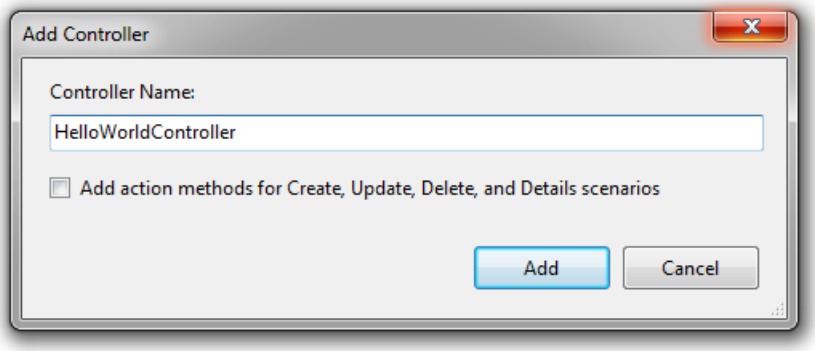
- Model: The data of your application
- Views: The template files your application will use to dynamically generate HTML responses.
- Controllers: Classes that handle incoming URL requests to the application, retrieve model data, and then specify view templates that render a response back to the client

We'll be covering all these concepts in this tutorial and show you how to use them to build an application.

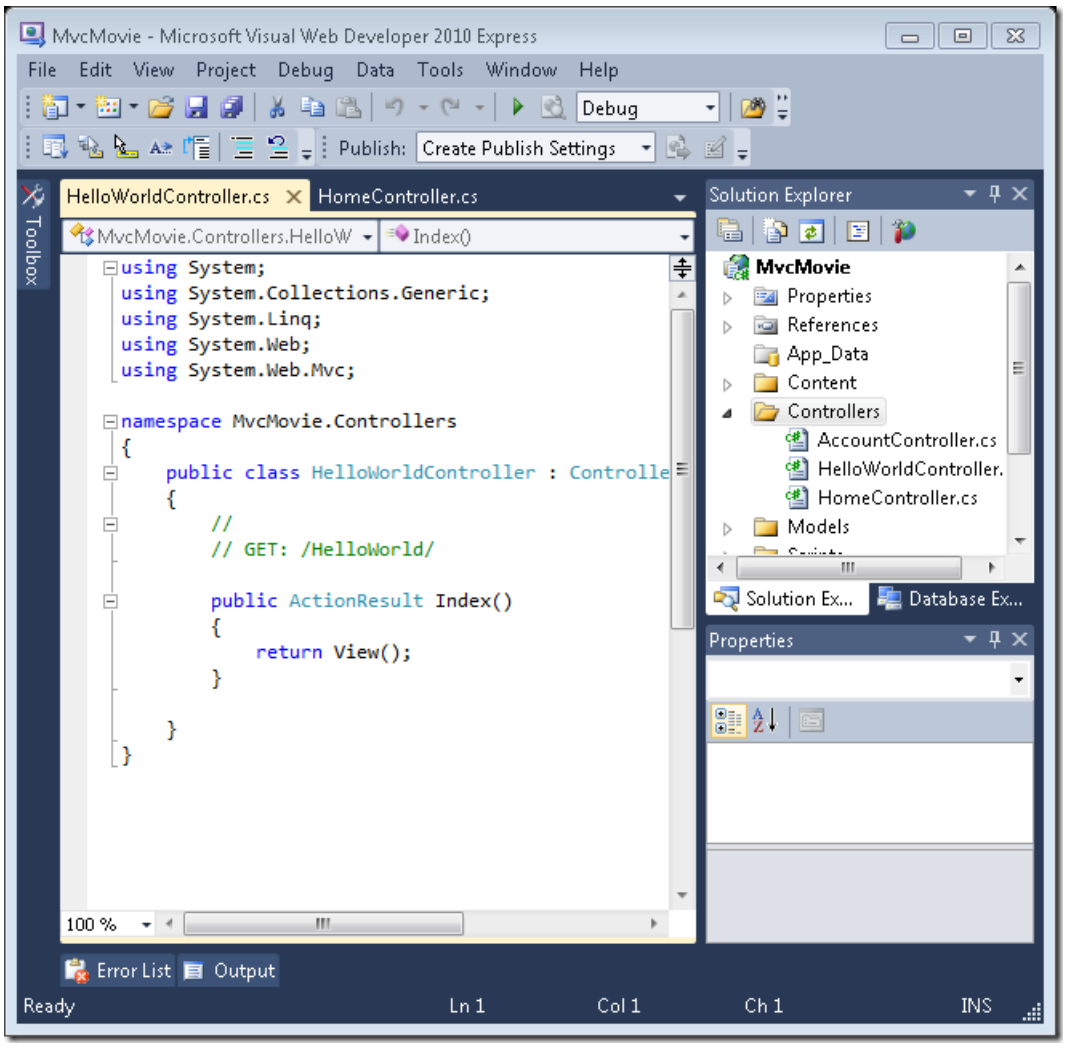
Let's create a new controller by right-clicking the controllers folder in the solution Explorer and selecting Add Controller.



Name your new controller "HelloWorldController" and click Add.



Notice in the Solution Explorer on the right that a new file has been created for you called HelloWorldController.cs and that file is now opened in the **IDE**.



Create two new methods that look like this inside of your new public class HelloWorldController. We'll return a string of HTML directly from our controller as an example.

```
using System.Web.Mvc;

namespace Movies.Controllers
{
    public class HelloWorldController : Controller
    {
        public string Index()
        {
            return "This is my default action...";
        }

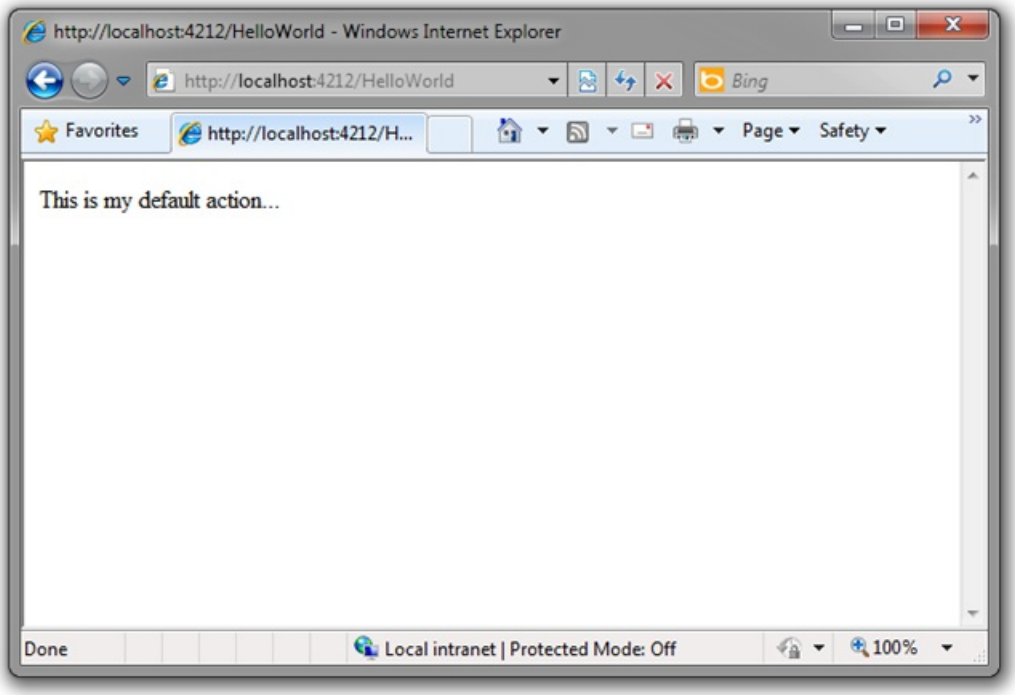
        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

Your Controller is named HelloWorldController and your new Method is called Index. Run your application again, just as before (click the play button or press F5 to do this). Once your browser has started up, change the path in the address bar to `http://localhost:xx/HelloWorld` where xx is whatever number your computer has chosen. Now your browser should look like the screenshot below. In our method above we returned a string passed into a method called "Content." We told the system just returns some HTML, and it did!

ASP.NET MVC invokes different Controller classes (and different Action methods within them) depending on the incoming URL. The default mapping logic used by ASP.NET MVC uses a format like this to control what code is run:

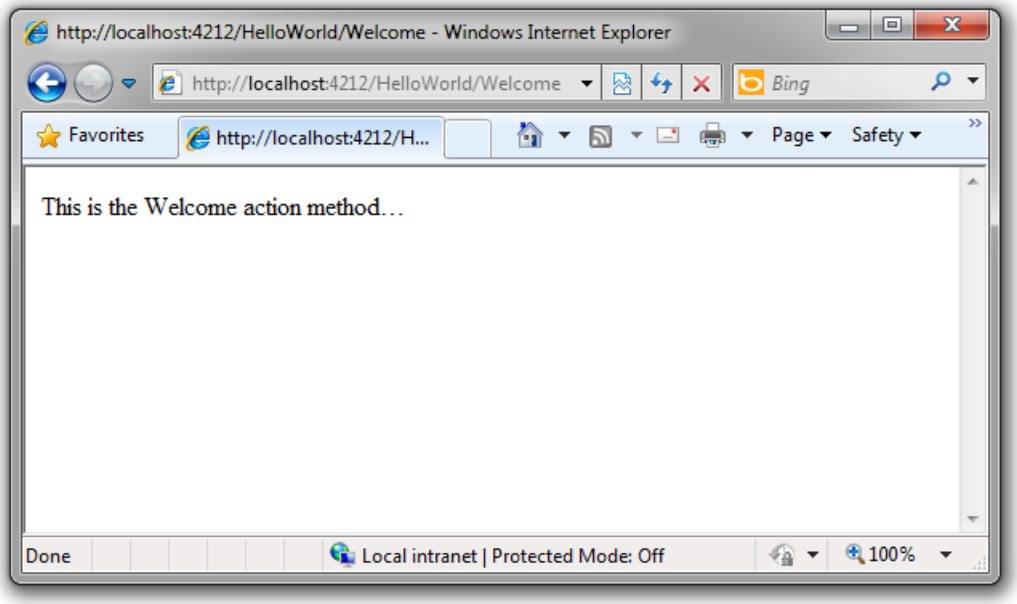
/[Controller]/[ActionName]/[Parameters]

The first part of the URL determines the Controller class to execute. So /HelloWorld maps to the HelloWorldController class. The second part of the URL determines the Action method on the class to execute. So /HelloWorld/Index would cause the Index() method of the HelloWorldController class to execute. Notice that we only had to visit /HelloWorld above and the method Index was implied. This is because a method named "Index" is the default method that will be called on a controller if one is not explicitly specified.



Now, let's visit `http://localhost:xx/HelloWorld/Welcome.` Now our Welcome Method has executed and returned its HTML string.

Again, /[Controller]/[ActionName]/[Parameters] so Controller is HelloWorld and Welcome is the Method in this case. We haven't done Parameters yet.



Let's modify our sample slightly so that we can pass some information in from the URL to our controller, for example like this: /HelloWorld/Welcome?name=Scott&numtimes=4. Change your Welcome method to include

two parameters and update it like below. Note that we've used the C# optional parameter feature to indicate that the parameter numTimes should default to 1 if it's not passed in.

```
public string Welcome(string name, int numTimes = 1)
{
   string message = "Hello " + name + ", NumTimes is: " + numTimes;
   return "" + Server.HtmlEncode(message) + "";
}
```

Run your application and visit `http://localhost:xx/HelloWorld/Welcome?name=Scott&numtimes=4` changing the value of name and numtimes as you like. The system automatically mapped the named parameters from your query string in the address bar to parameters in your method.

In both these examples the controller has been doing all the work, and has been returning HTML directly. Ordinarily we don't want our Controllers returning HTML directly - since that ends up being very cumbersome to code. Instead we'll typically use a separate View template file to help generate the HTML response. Let's look at how we can do this. Close your browser and return to the IDE.

PREVIOUS NEXT

Adding a View

2/20/2020 • 6 minutes to read • Edit Online

by Scott Hanselman

> This is a beginner tutorial that introduces the basics of ASP.NET MVC. You'll create a simple web application that reads and writes from a database. Visit the ASP.NET MVC learning center to find other ASP.NET MVC tutorials and samples.

In this section we are going to look at how we can have our HelloWorldController class use a View template file to cleanly encapsulate generating HTML responses back to a client.

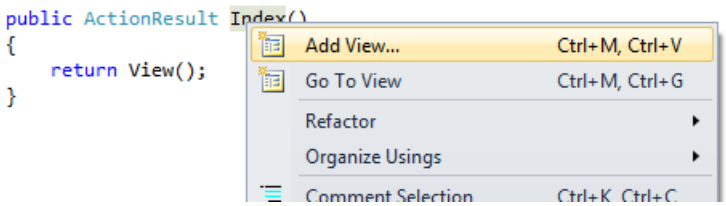
Let's start by using a View template with our Index method. Our method is called Index and it's in the HelloWorldController. Currently our Index() method returns a string with a message that is hardcoded within the Controller class.

```
public string Index()
{
    return "<html><body>This is my default action...</body></html>";
}
```

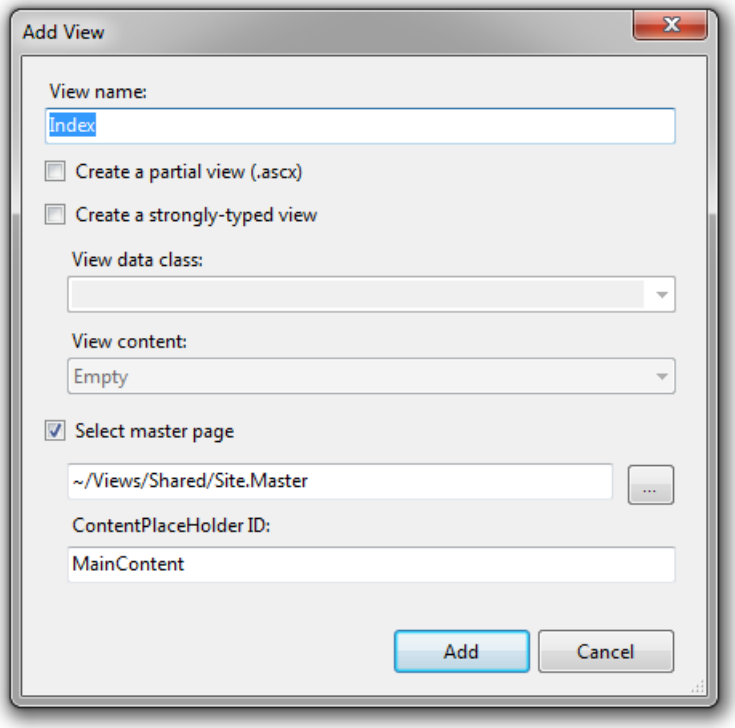
Let's now change the Index method to instead look like this:

```
public ActionResult Index()
{
   return View();
}
```

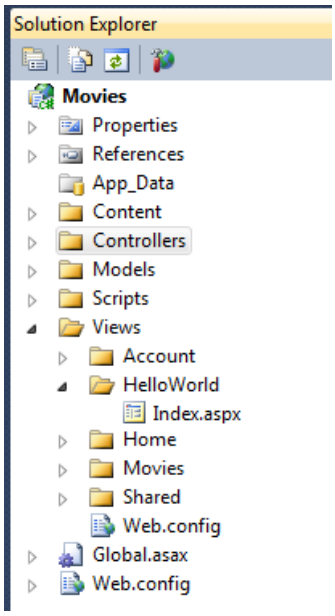
Let's now add a View template to our project that we can use for our Index() method. To do this, right-click with your mouse somewhere in the middle of the Index method and click Add View...



This will bring up the "Add View" dialog which provides us some options for how we want to create a view template that can be used by our Index method. For now, don't change anything, and just click the Add button.



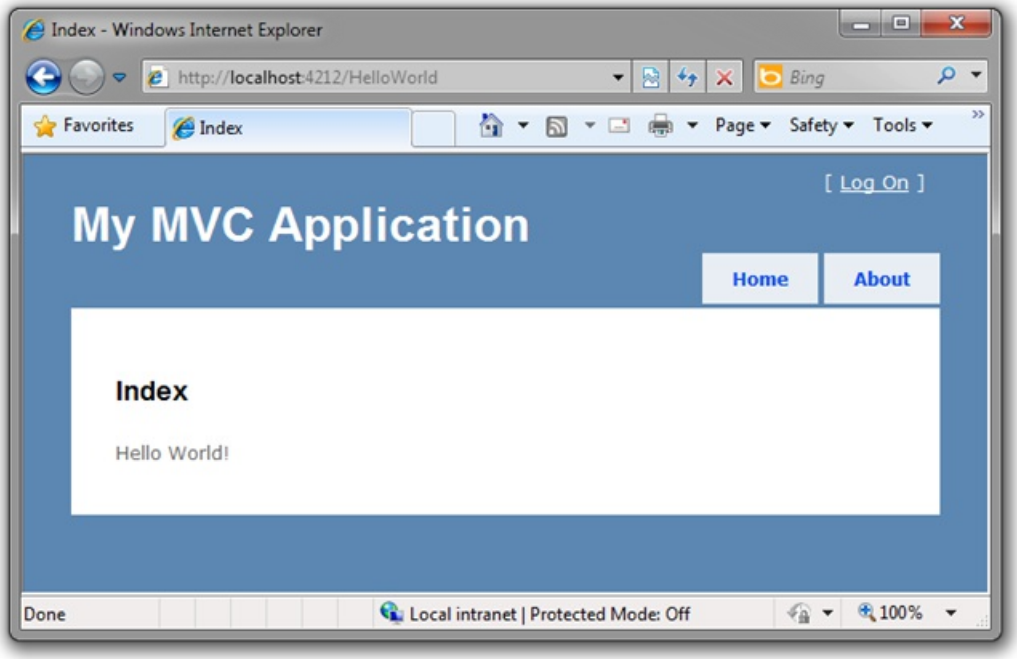
After you click Add, a new folder and a new file will appear in the Solution Folder, as seen here. I now have a HelloWorld folder under Views, and an Index.aspx file inside that folder.



The new Index file is also already opened and ready for editing. Add some text under the first <h2>Index</h2> like "Hello World."

```
<h2>Index</h2>
Hello world!
```

Run your application and visit `http://localhost:xx/HelloWorld` again in your browser. The Index method in our controller in this example didn't do any work, but did call "return View()" which indicated that we wanted to use a view template file to render a response back to the client. Because we did not explicitly specify the name of the view template file to use, ASP.NET MVC defaulted to using the Index.aspx view file within the \Views\HelloWorld folder. Now we see the string we hard-coded in our View.



Looks pretty good. However, notice that the Browser's title says "Index" and the big title on the page says "My MVC Application." Let's change those.

Changing Views and Master Pages

First, let's change the text "My MVC Application." That text is shared and appears on every page. It actually appears in only one place in our code, even though it's on every page in our app. Go to the /Views/Shared folder in the Solution Explorer and open the Site.Master file. This file is called a Master Page and it's the shared "shell" that all our other pages use.

Notice some text that says ContentPlaceholder "MainContent" in this file.

```
<asp:ContentPlaceHolder ID="MainContent" runat="server" />
```

That placeholder is where all the pages you create will show up, "wrapped" in the master page. Try changing the title, then run your app and visit multiple pages. You'll notice that your one change appears on multiple pages.

```
<div id="title">
    <h1>My MVC Movie Application</h1>
</div>
```

Now every page will have the Primary Heading - that's H1 - of "My MVC Movie Application." That handles the white text at the top there that's shared across all the pages.

Here is the Site.Master in its entirety with our changed title:

```
<%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>
    <link href="../../Content/Site.css" rel="stylesheet" type="text/css" />
</head>

<body>
    <div class="page">

        <div id="header">
            <div id="title">
                <h1>My MVC Movie Application</h1>
            </div>

            <div id="logindisplay">
                <% Html.RenderPartial("LogOnUserControl"); %>
            </div>

            <div id="menucontainer">

                <ul id="menu">
                    <li><%: Html.ActionLink("Home", "Index", "Home")%></li>
                    <li><%: Html.ActionLink("About", "About", "Home")%></li>
                </ul>

            </div>
        </div>

        <div id="main">
            <asp:ContentPlaceHolder ID="MainContent" runat="server" />

            <div id="footer">
            </div>
        </div>
    </div>
</body>
</html>
```

Now, let's change the title of the Index page.

Open /HelloWorld/Index.aspx. There's two places to change. First, the Title that appears in the title of the browser, then the secondary header - that's H2 - as well. I'll make them each slightly different so you can see which bit of code changes which part of the app.

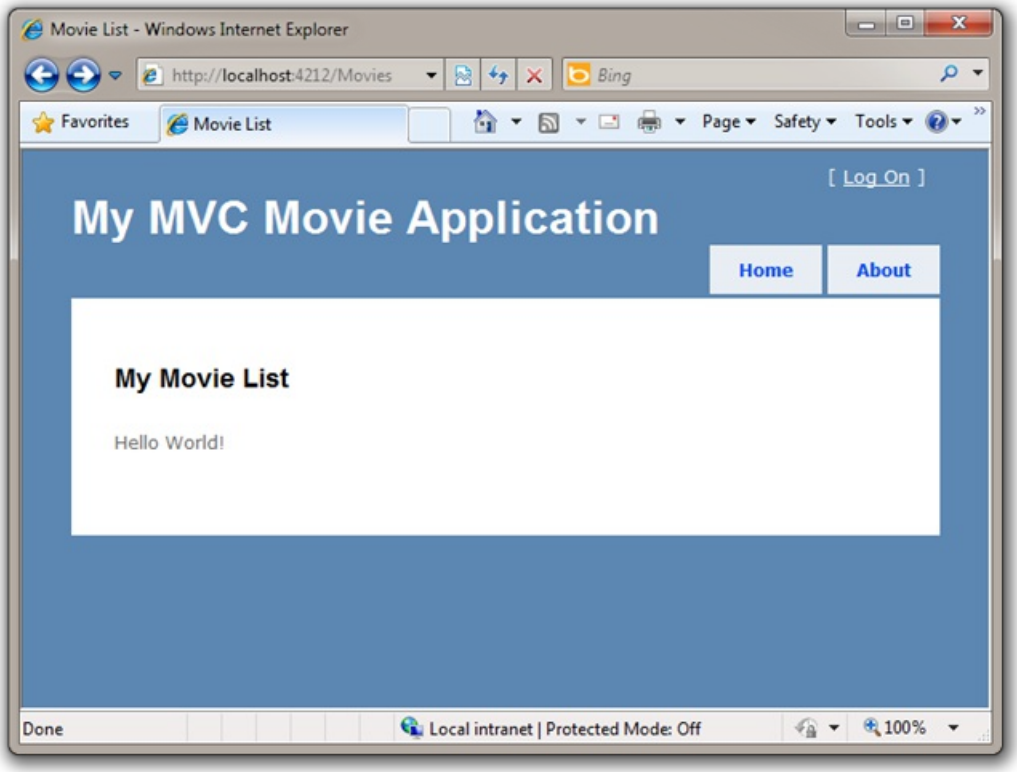
```
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Movie List
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>My Movie List</h2>
    Hello World!

</asp:Content>
```

Run your app and visit /Movies. Notice that the browser title, the primary heading and the secondary headings have changed. It's easy to make big changes in your app with small changes to your view.



Our little bit of "data" (in this case the "Hello World!" message) was hard coded though. We've got V (Views) and we've got C (Controllers), but no M (Model) yet. We'll shortly walk through how create a database and retrieve model data from it.

Passing a ViewModel

Before we go to a database and talk about Models, though, let's first talk about "ViewModels." These are objects that represent what a View template requires to render an HTML response back to a client. They are typically created and passed by a Controller class to a View template, and should only contain the data that the View template requires - and no more.

Previously with our HelloWorld sample, our Welcome() action method took a name and a numTimes parameter and output it to the browser. Rather than have the Controller continue to render this response directly,let's instead make a small class to hold that data and then pass it over to a View template to render back the HTML response using it. This way the Controller is concerned with one thing and the View template another – enabling us to maintain clean "separation of concerns" within our application.

Return to the HelloWorldController.cs file and add a new "WelcomeViewModel" class and change the Welcome method inside your controller. Here is the complete HelloWorldController.cs with the new class in the same file.

```
using System.Web.Mvc;

namespace Movies.Controllers
{
    public class HelloWorldController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Welcome(string name, int numTimes = 1)
        {
            var viewModel = new WelcomeViewModel
            {
                Message = "Hello " + name,
                NumTimes = numTimes
            };

            return View(viewModel);
        }

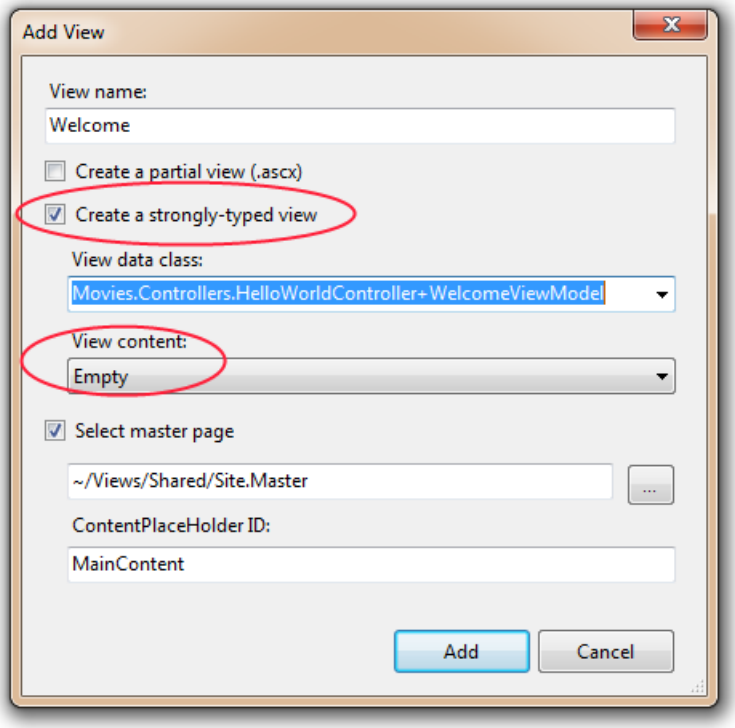
        public class WelcomeViewModel
        {
            public string Message { get; set; }
            public int NumTimes { get; set; }
        }
    }
}
```

Even though it's on multiple lines, our Welcome method is really only two code statements. The first statement packages up our two parameters into a ViewModel object, and the second passes the resulting object onto the View.

Now we need a Welcome View template! Right click in the Welcome method and select Add View. This time, we'll check "Create a strongly-typed view" and select our WelcomeViewModel class from the drop down list. This new view will only know about WelcomeViewModels and no other types of objects.

> *NOTE: You'll need to have compiled once after adding your WelcomeViewModel for to show up in the drop down list.*

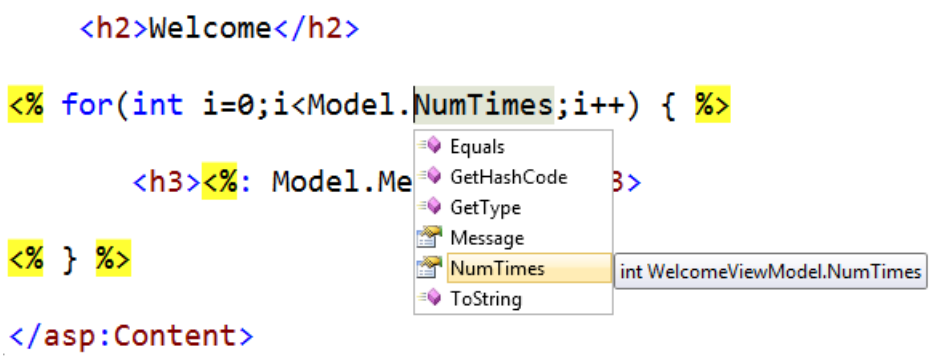
Here's what your Add View dialog should look like. Click the Add button.



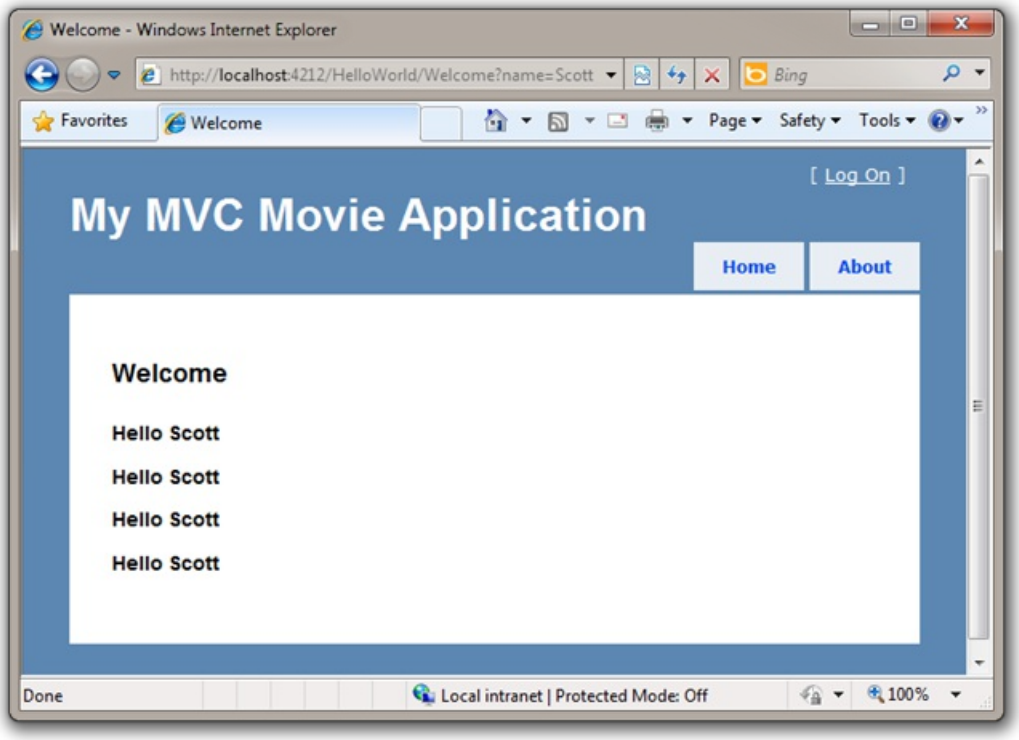
Add this code under the <h2> in your new Welcome.aspx. We'll make a loop and say Hello as many times as the user says we should!

```
<% for(int i=0;i<Model.NumTimes;i++) { %>
        <%h3><%: Model.Message %></h3>
<% } %>
```

Also, notice while you're typing that because we told this View about the WelcomeViewModel (they are married, remember?) that we get helpful Intellisense each time we reference our Model object as seen in the screenshot below:



Run your application and visit `http://localhost:xx/HelloWorld/Welcome?name=Scott&numtimes=4` again. Now we're taking data from the URL, it's passed into our Controller automatically, our Controller packages up the data into a ViewModel and passes that object onto our View. The View than displays the data as HTML to the user.



Well, that was a kind of an "M" for Model, but not the database kind. Let's take what we've learned and create a database of Movies.

PREVIOUS NEXT

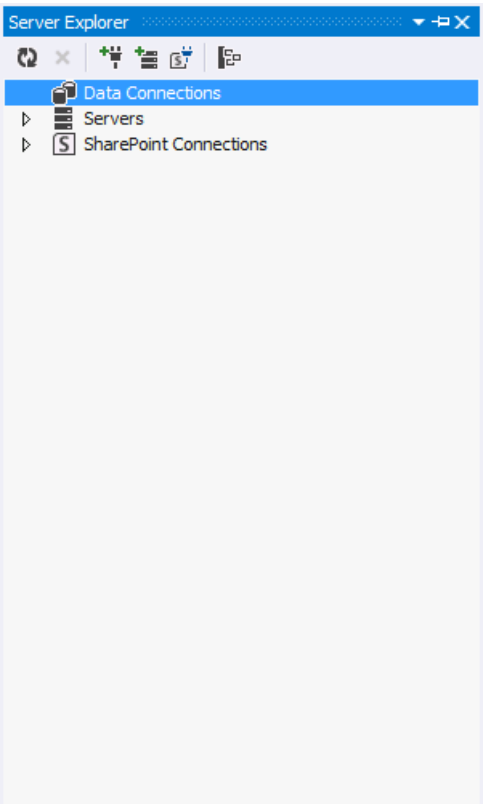
Creating a Database

2/20/2020 • 2 minutes to read • Edit Online

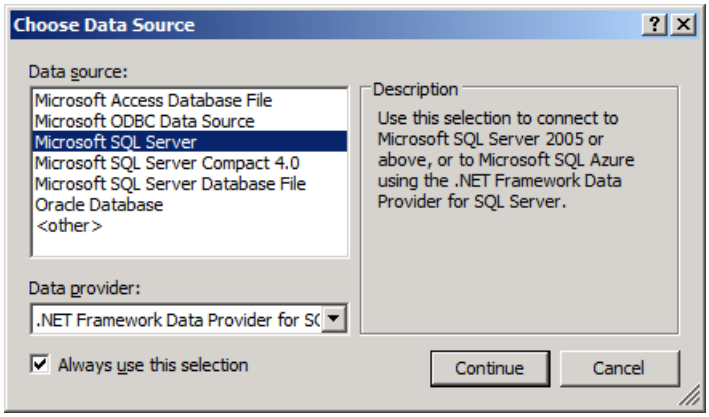
by Scott Hanselman

This is a beginner tutorial that introduces the basics of ASP.NET MVC. You'll create a simple web application that reads and writes from a database. Visit the ASP.NET MVC learning center to find other ASP.NET MVC tutorials and samples.

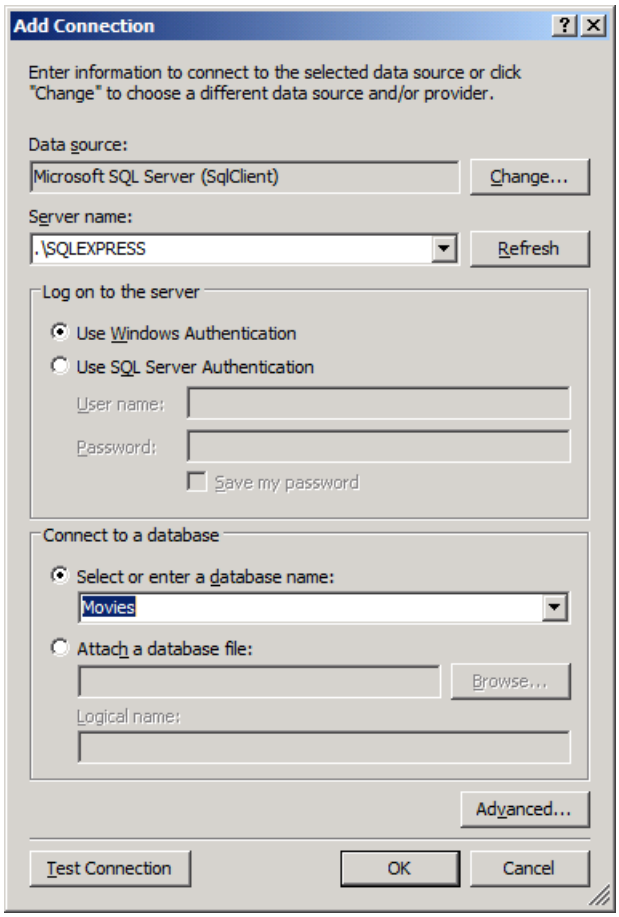
In this section we are going to create a new SQL Express database that we'll use to store and retrieve our movie data. From within the Visual Web Developer IDE, select View | Server Explorer. Right click on Data Connections and click Add Connection...



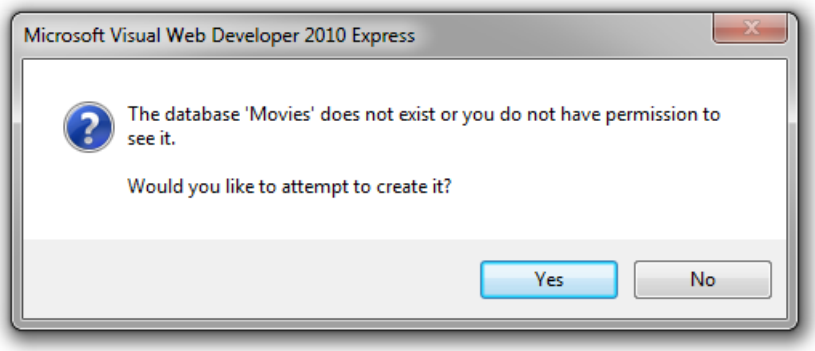
In the Choose Data Source dialog, select Microsoft SQL Server and select Continue.



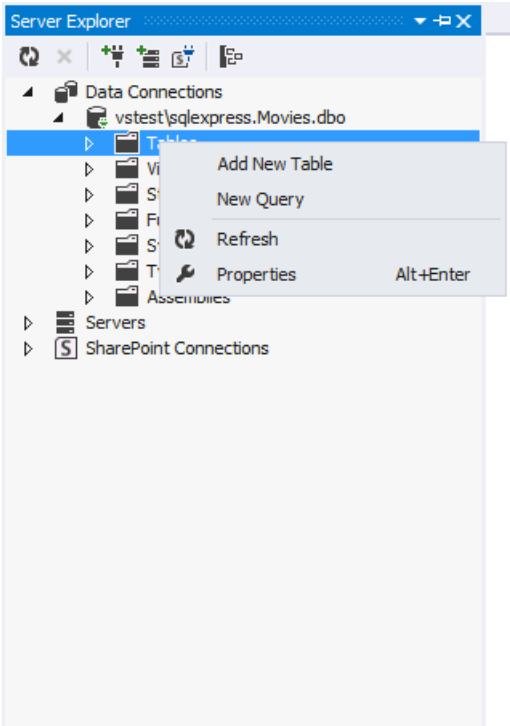
In the Add Connection dialog, enter ".\SQLEXPRESS" for your Server Name, and enter "Movies" as the name for your new database.



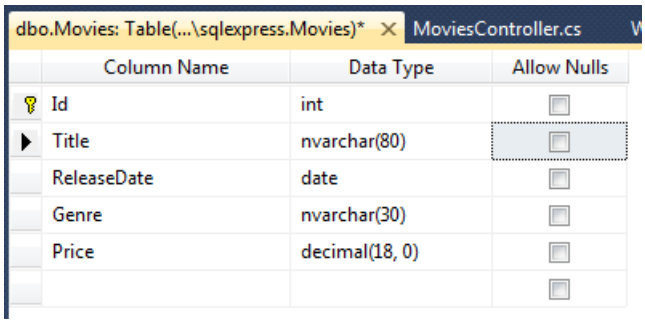
Click OK and you'll be asked if you want to create that database. Select yes.



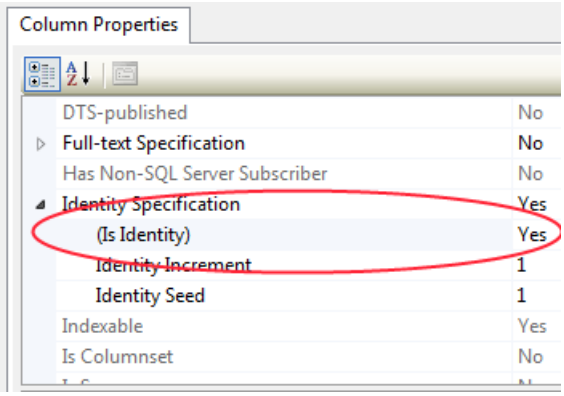
Now you've got an empty database in Server Explorer.



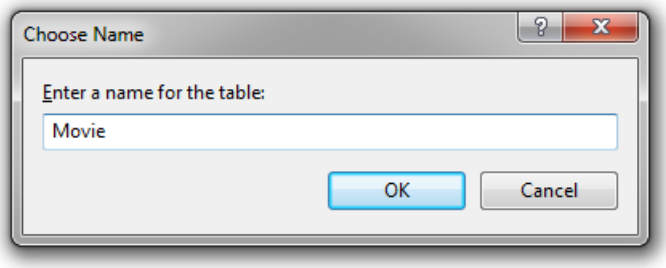
Right click on Tables and click Add Table. The Table Designer will appear. Add columns for Id, Title, ReleaseDate, Genre, and Price. Right click on the ID column and click set Primary Key. Here's what my design areas looks like.



Also, select the Id column and under Column Properties below change "Identity Specification" to "Yes."



When you've got it done, click the Save icon in the toolbar or select File | Save from the menu, and name your table "**Movie**" (singular). We've got a database and a table!



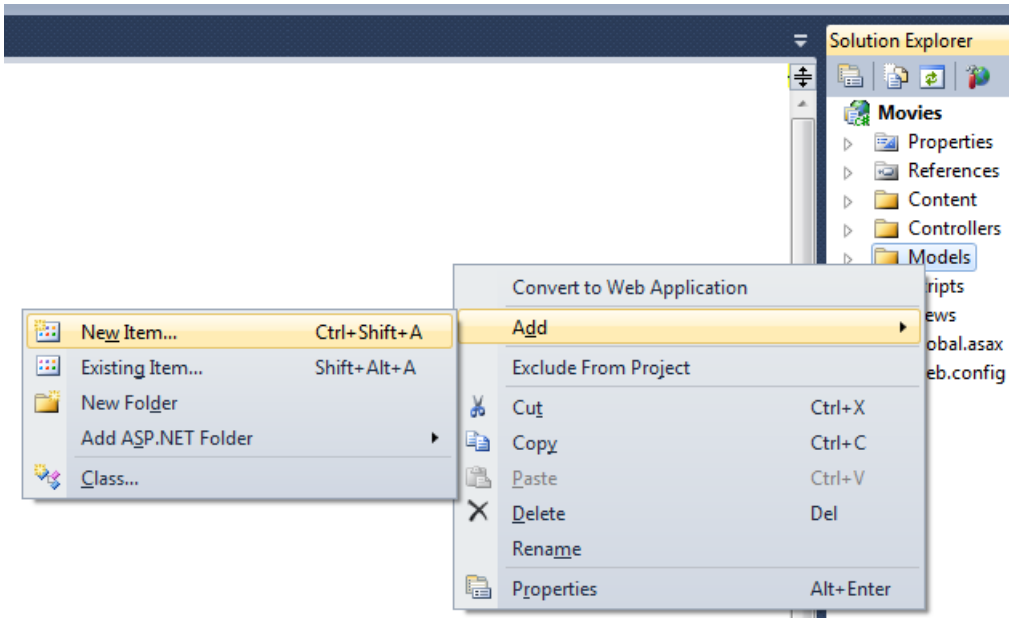
Go back to Server Explorer and right click the Movie table, then select "Show Table Data." Enter a few movies so our database has some data.

Movie: Query(hexpo...sqlexpress.Movies) | Movies.edmx | Web.config* | MoviesController.cs

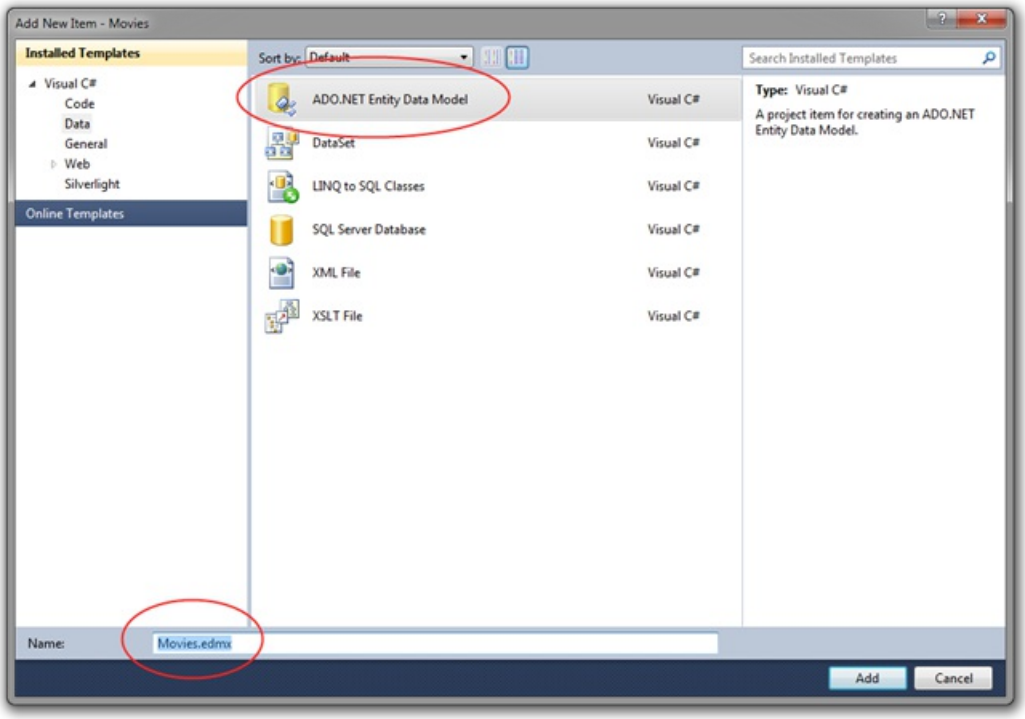
Id	Title	ReleaseDate	Genre	Price
1	Ghostbusters	1/1/1984	Comedy	10
2	Ghostbusters 2	1/1/1986	Comedy	9
3	Ghostbusters 3	1/1/2012	Comedy	98
NULL	NULL	NULL	NULL	NULL

Creating a Model

Now, switch back to the Solution Explorer on the right hand side of the IDE and right-click on the Models folder and select Add | New Item.

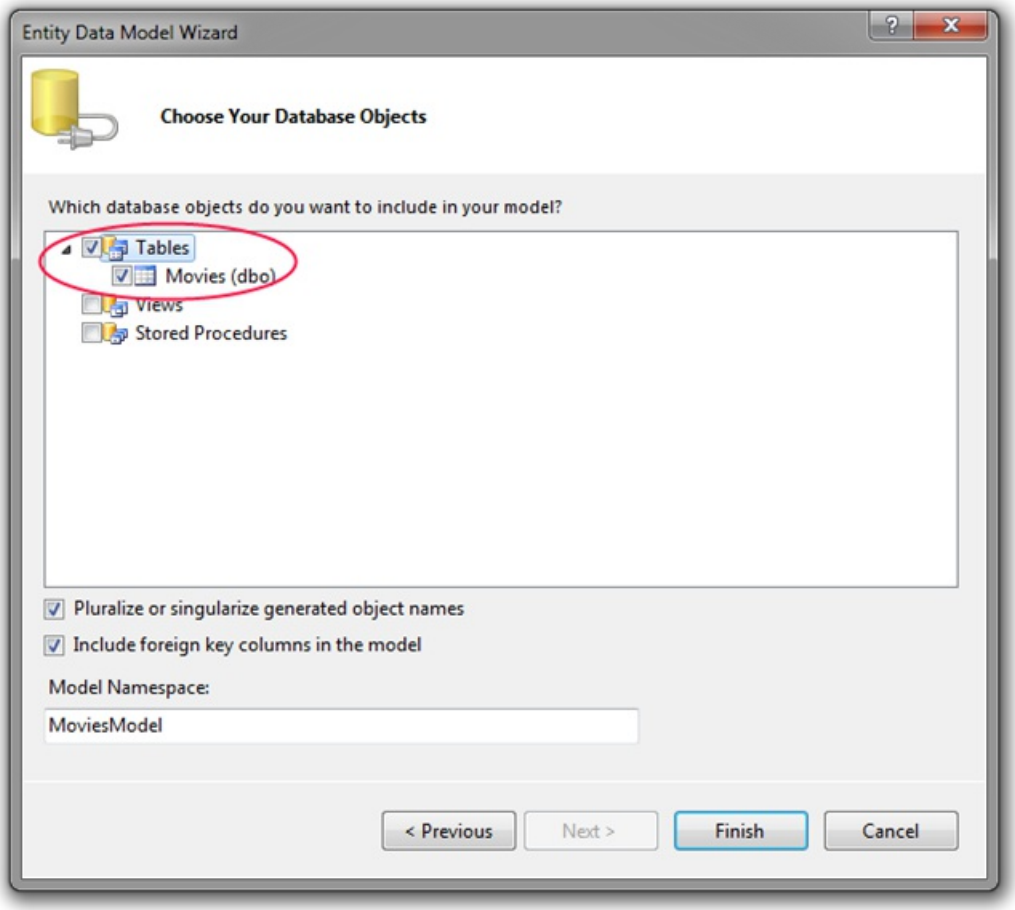


We're going to create an Entity Model from our new database. This will add a set of classes to our project that makes it easy for us to query and manipulate the data within our database. Select the Data node on the left-hand side of the dialog, and then select the ADO.NET Entity Data Model item template. Name it Movies.edmx.

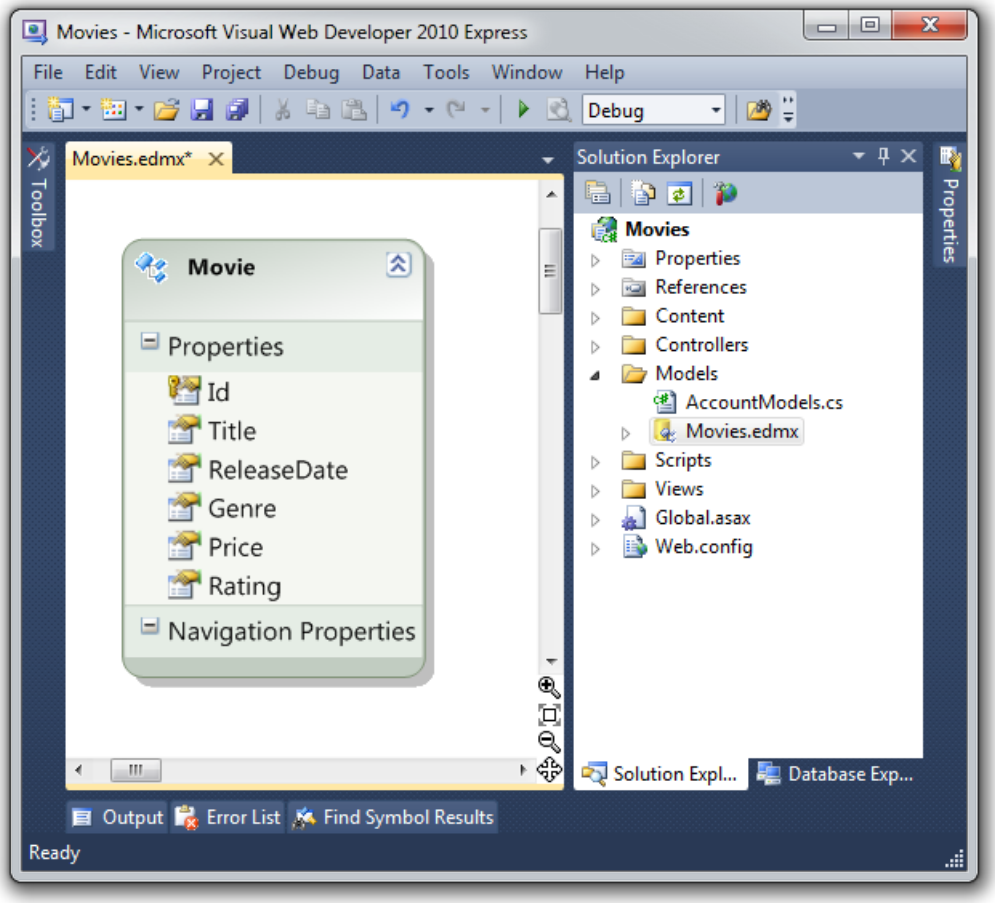


Click the "Add" button. This will then launch the "Entity Data Model Wizard".

In the new dialog that pops up, select Generate from Database. Since we've just made a database, we'll only need to tell the Entity Framework about our new database and its table. Click Next to save our database connection in our web application's configuration. Now, check the Tables and Movie checkbox and click Finish.



Now we can see our new Movie table in the Entity Framework Designer and access it from code.



On the design surface you can see a "Movie" class. This class maps to the "Movie" table in our database, and each property within it maps to a column with the table. Each instance of a "Movie" class will correspond to a row within the "Movie" table.

If you don't like the default naming and mapping conventions used by the Entity Framework, you can use the Entity Framework designer to change or customize them. For this application we'll use the defaults and just save the file as-is.

Now, let's work with some real data!

PREVIOUS NEXT

Accessing your Model's Data from a Controller

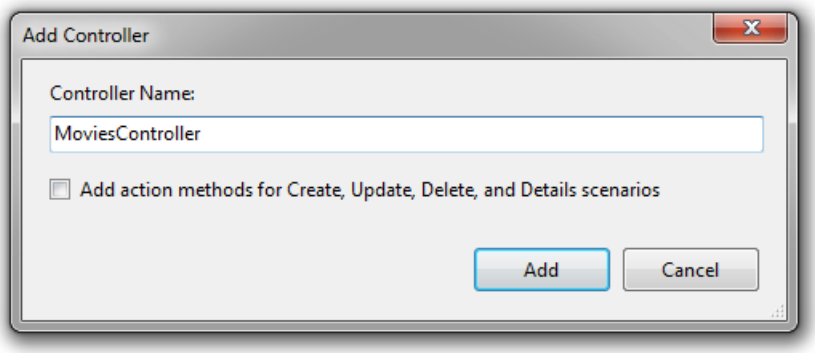
2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

This is a beginner tutorial that introduces the basics of ASP.NET MVC. You'll create a simple web application that reads and writes from a database. Visit the ASP.NET MVC learning center to find other ASP.NET MVC tutorials and samples.

In this section we are going to create a new MoviesController class, and write some code that retrieves our Movie data and displays it back to the browser using a View template.

Right click on the Controllers folder and make a new MoviesController.



This will create a new "MoviesController.cs" file underneath our \Controllers folder within our project. Let's update the MovieController to retrieve the list of movies from our newly populated database.

```
using System;
using System.Linq;
using System.Web.Mvc;
using Movies.Models;

namespace Movies.Controllers
{
    public class MoviesController : Controller
    {
        MoviesEntities db = new MoviesEntities();

        public ActionResult Index()
        {
            var movies = from m in db.Movies
                         where m.ReleaseDate > new DateTime(1984, 6, 1)
                         select m;

            return View(movies.ToList());

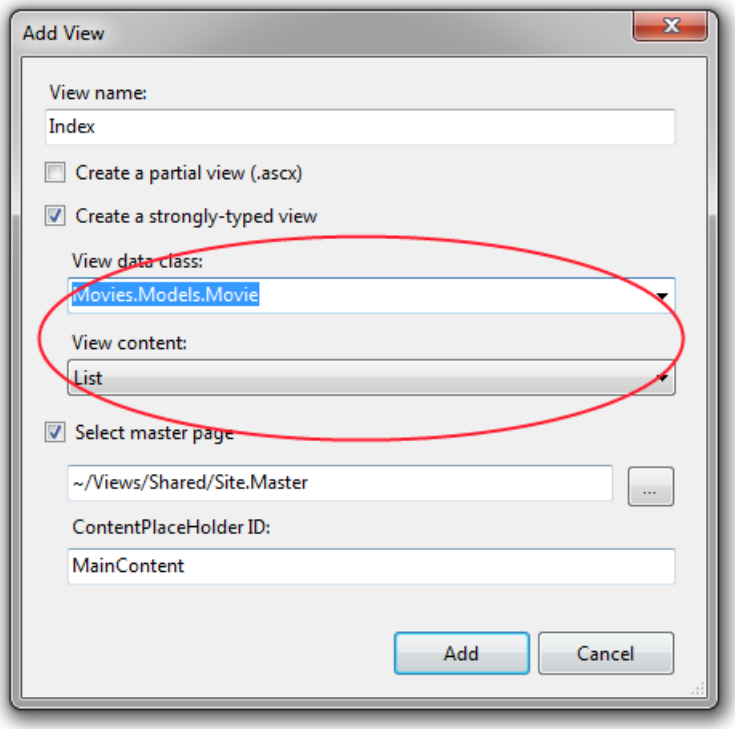
        }
    }
}
```

We are performing a LINQ query so that we only retrieve movies released after the summer of 1984. We'll need a View template to render this list of movies back, so right-click in the method and select Add View to create it.

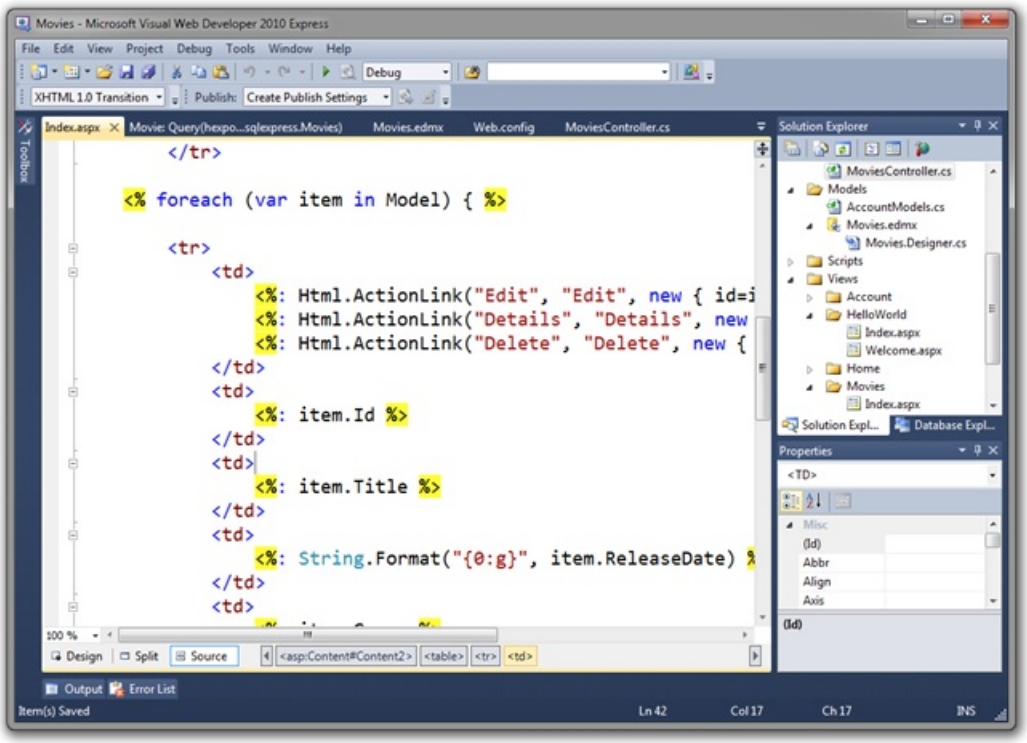
Within the Add View dialog we'll indicate that we are passing a List<Movies.Models.Movie> to our View template.

Unlike the previous times we used the Add View dialog and chose to create an "Empty" template, this time we'll indicate that we want Visual Studio to automatically "scaffold" a view template for us with some default content. We'll do this by selecting the "List" item within the "View content dropdown menu.

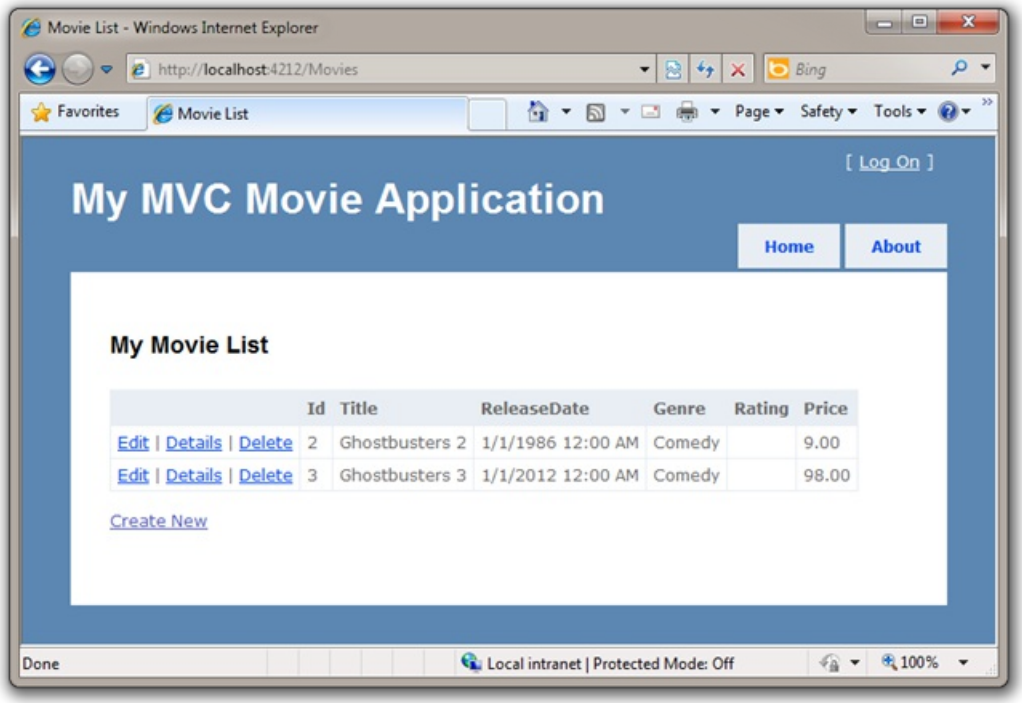
Remember, when you have a created a new class you'll need to compile your application for it to show up in the Add View Dialog.



Click add and the system will automatically generate the code for a View for us that displays our list of movies. This is a good time to change the <h2> heading to something like "My Movie List" like we did earlier with the Hello World view.



Run your application and visit /Movies in the address bar. Now we've retrieved data from the database using a basic query inside the Controller and returned the data to a View that knows about Movies. That View then spins through the list of Movies and creates a table of data for us.



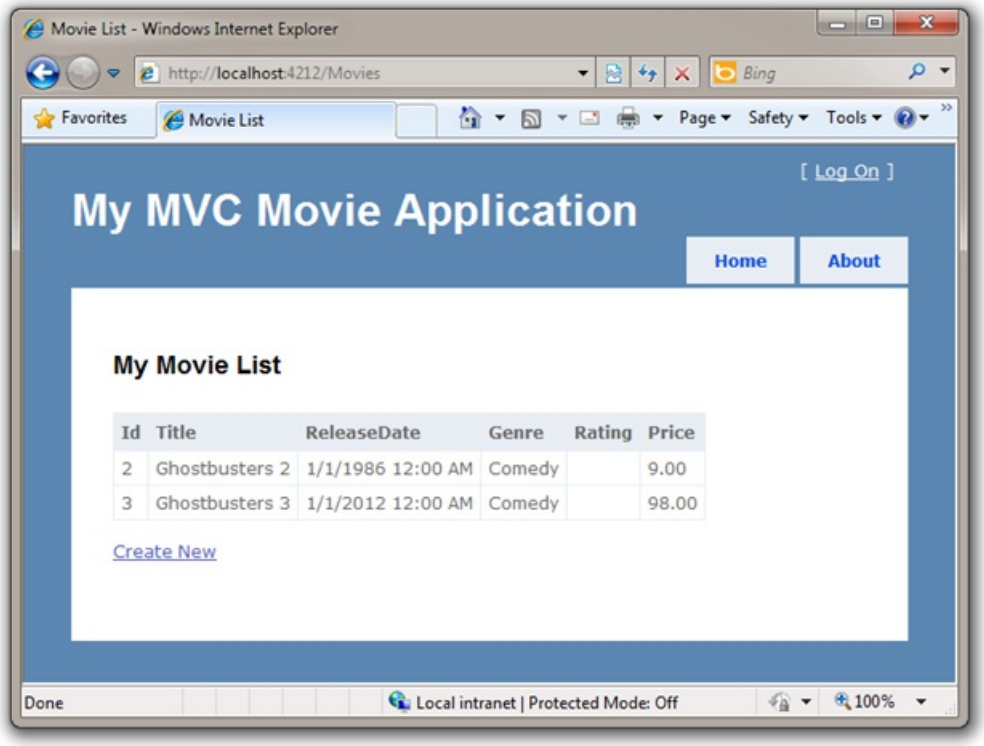
We won't be implementing Edit, Details and Delete functionality with this application - so we don't need the default links that the scaffold template created for us. Open up the /Movies/Index.aspx file and remove them.

Here is the source code for what our updated View template should look like once we make these changes:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<IEnumerable<Movies.Models.Movie>>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Movie List
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>My Movie List</h2>
    <table>
        <tr>
            <th>Title</th>
            <th>ReleaseDate</th>
            <th>Genre</th>
            <th>Rating</th>
            <th>Price</th>
        </tr>
        <% foreach (var item in Model)
           { %>
        <tr>
            <td><%: item.Title %></td>
            <td><%: String.Format("{0:g}", item.ReleaseDate) %></td>
            <td><%: item.Genre %></td>
            <td><%: item.Rating %></td>
            <td><%: String.Format("{0:F}", item.Price) %></td>
        </tr>
        <% } %>
    </table>
    <p>
        <%: Html.ActionLink("Create New", "Create") %>
    </p>
</asp:Content>
```

It's creating links that we won't need, so we'll delete them for this example. We will keep our Create New link though, as that's next! Here's what our app looks like with that column removed.



We now have a simple listing of our movie data. However, if we click the "Create New" link, we'll get an error as it's not hooked up! Let's implement a Create Action method and enable a user to enter new movies in our database.

PREVIOUS NEXT

Adding a Create Method and Create View

2/20/2020 • 4 minutes to read • Edit Online

by Scott Hanselman

> This is a beginner tutorial that introduces the basics of ASP.NET MVC. You'll create a simple web application that reads and writes from a database. Visit the ASP.NET MVC learning center to find other ASP.NET MVC tutorials and samples.

In this section we are going to implement the support necessary to enable users to create new movies in our database. We'll do this by implementing the /Movies/Create URL action.

Implementing the /Movies/Create URL is a two step process. When a user first visits the /Movies/Create URL we want to show them an HTML form that they can fill out to enter a new movie. Then, when the user submits the form and posts the data back to the server, we want to retrieve the posted contents and save it into our database.

We'll implement these two steps within two Create() methods within our MoviesController class. One method will show the <form> that the user should fill out to create a new movie. The second method will handle processing the posted data when the user submits the <form> back to the server, and save a new Movie within our database.

Below is the code we'll add to our MoviesController class to implement this:

```
public ActionResult Create()
{
    return View();
}

[HttpPost]
public ActionResult Create(Movie newMovie)
{

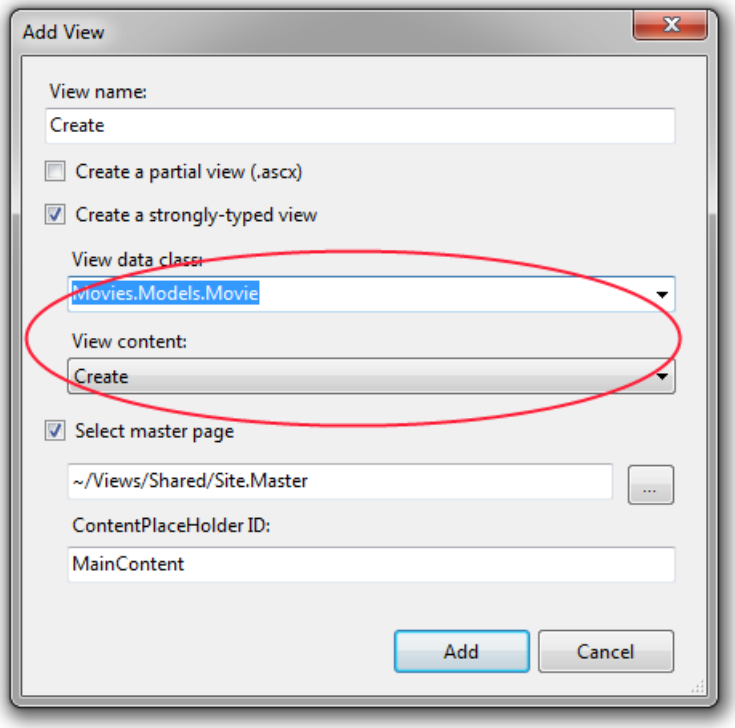
    if (ModelState.IsValid)
    {
        db.AddToMovies(newMovie);
        db.SaveChanges();

        return RedirectToAction("Index");
    }
    else
    {
        return View(newMovie);
    }
}
```

The above code contains all of the code that we'll need within our Controller.

Let's now implement the Create View template that we'll use to display a form to the user. We'll right click in the first Create method and select "Add View" to create the view template for our Movie form.

We'll select that we are going to pass the view template a "Movie" as its view data class, and indicate that we want to "scaffold" a "Create" template.



After you click the Add button, \Movies\Create.aspx View template will be created for you. Because we selected "Create" from the "view content" dropdown, the Add View dialog automatically "scaffolded" some default content for us. The scaffolding created an HTML <form>, a place for validation error messages to go, and since scaffolding knows about Movies, it created Label and Fields for each property of our class.

```
<% using (Html.BeginForm()) {%>
    <%: Html.ValidationSummary(true) %>

    <fieldset>
        <legend>Fields</legend>

        <div class="editor-label">
            <%: Html.LabelFor(model => model.Id) %>
        </div>
        <div class="editor-field">
            <%: Html.TextBoxFor(model => model.Id) %>
            <%: Html.ValidationMessageFor(model => model.Id) %>
        </div>

        <div class="editor-label">
            <%: Html.LabelFor(model => model.Title) %>
        </div>
        <div class="editor-field">
            <%: Html.TextBoxFor(model => model.Title) %>
            <%: Html.ValidationMessageFor(model => model.Title) %>
        </div>

        <div class="editor-label">
            <%: Html.LabelFor(model => model.ReleaseDate) %>
        </div>
        <div class="editor-field">
            <%: Html.TextBoxFor(model => model.ReleaseDate) %>
            <%: Html.ValidationMessageFor(model => model.ReleaseDate) %>
        </div>

        <div class="editor-label">
            <%: Html.LabelFor(model => model.Genre) %>
        </div>
        <div class="editor-field">
            <%: Html.TextBoxFor(model => model.Genre) %>
            <%: Html.ValidationMessageFor(model => model.Genre) %>
        </div>

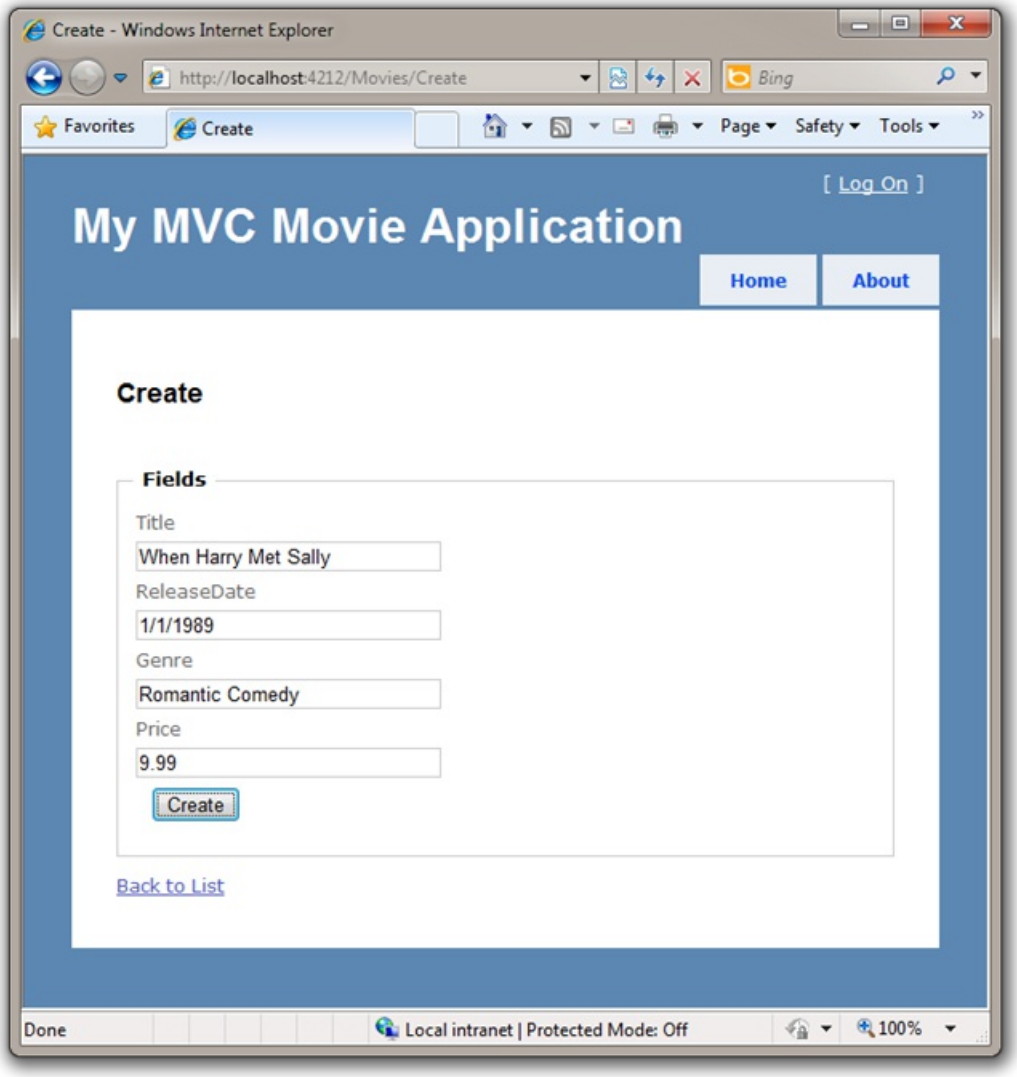
        <div class="editor-label">
            <%: Html.LabelFor(model => model.Price) %>
        </div>
        <div class="editor-field">
            <%: Html.TextBoxFor(model => model.Price) %>
            <%: Html.ValidationMessageFor(model => model.Price) %>
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>

<% } %>
```

Since our database automatically gives a Movie an ID, let's remove those fields that reference model.Id from our Create View. Remove the 7 lines after <legend>Fields</legend> as they show the ID field that we don't want.

Let's now create a new movie and add it to the database. We'll do this by running the application again and visit the "/Movies" URL and click the "Create" link to add a new Movie.



When we click the Create button, we'll be posting back (via HTTP POST) the data on this form to the /Movies/Create method that we just created. Just like when the system automatically took the "numTimes" and "name " parameter out of the URL and mapped them to parameters on a method earlier, the system will automatically take the Form Fields from a POST and map them to an object. In this case, values from fields in HTML like "ReleaseDate" and "Title" will automatically be put into the correct properties of a new instance of a Movie.

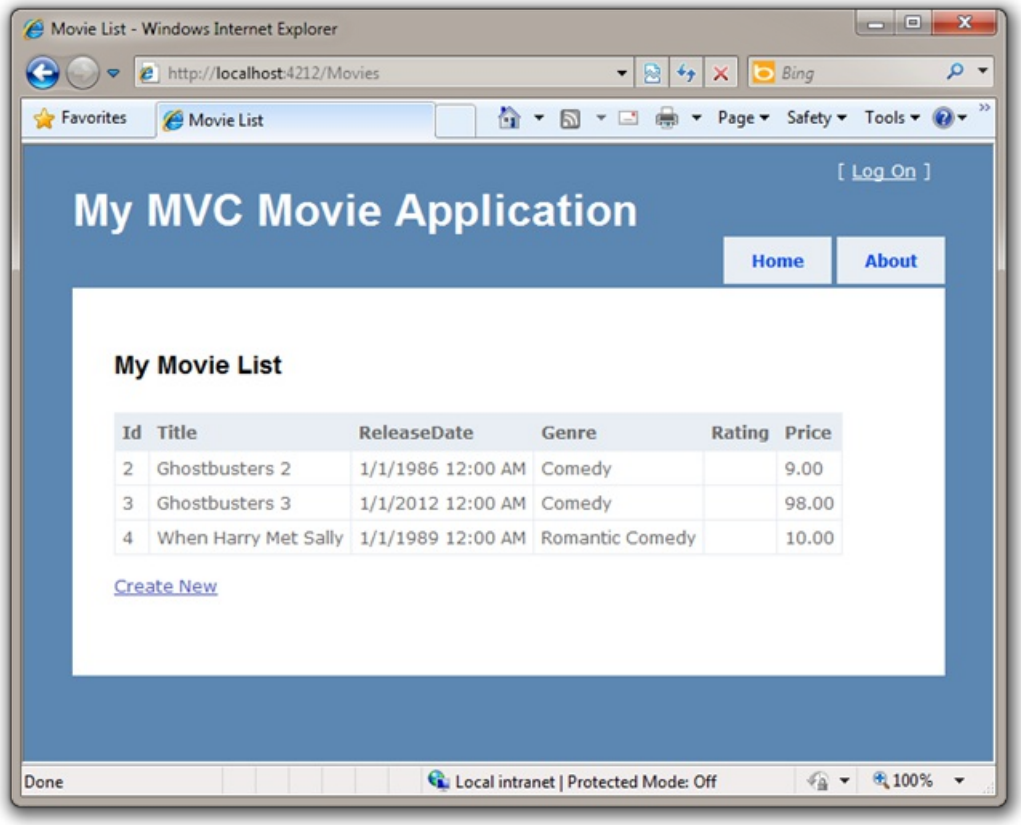
Let's look at the second Create method from our MoviesController again. Notice how it takes a "Movie" object as an argument:

```
[HttpPost]
public ActionResult Create(Movie newMovie)
{
    if (ModelState.IsValid)
    {
        db.AddToMovies(newMovie);
        db.SaveChanges();

        return RedirectToAction("Index");
    }
    else
    {
        return View(newMovie);
    }
}
```

This Movie object was then passed to the [HttpPost] version of our Create action method, and we saved it in the

database and then redirected the user back to the Index() action method which will show the saved result in the movie list:



We aren't checking if our movies are correct, though, and the database won't allow us to save a movie with no Title. It'd be nice if we could tell the user that before the database threw an error. We'll do this next by adding validation support to our application.

PREVIOUS NEXT

Adding Validation to the Model

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

This is a beginner tutorial that introduces the basics of ASP.NET MVC. You'll create a simple web application that reads and writes from a database. Visit the ASP.NET MVC learning center to find other ASP.NET MVC tutorials and samples.

In this section we are going to implement the support necessary to enable input validation within our application. We'll ensure that our database content is always correct, and provide helpful error messages to end users when they try and enter Movie data which is not valid. We'll begin by adding a little validation logic to the Movie class.

Right click on the Model folder and select Add Class. Name your class Movie.

When we created the Movie Entity Model earlier, the IDE created a Movie class. In fact, part of the Movie class can be in one file and part in another. This is called a Partial Class. We're going to extend the Movie class from another file.

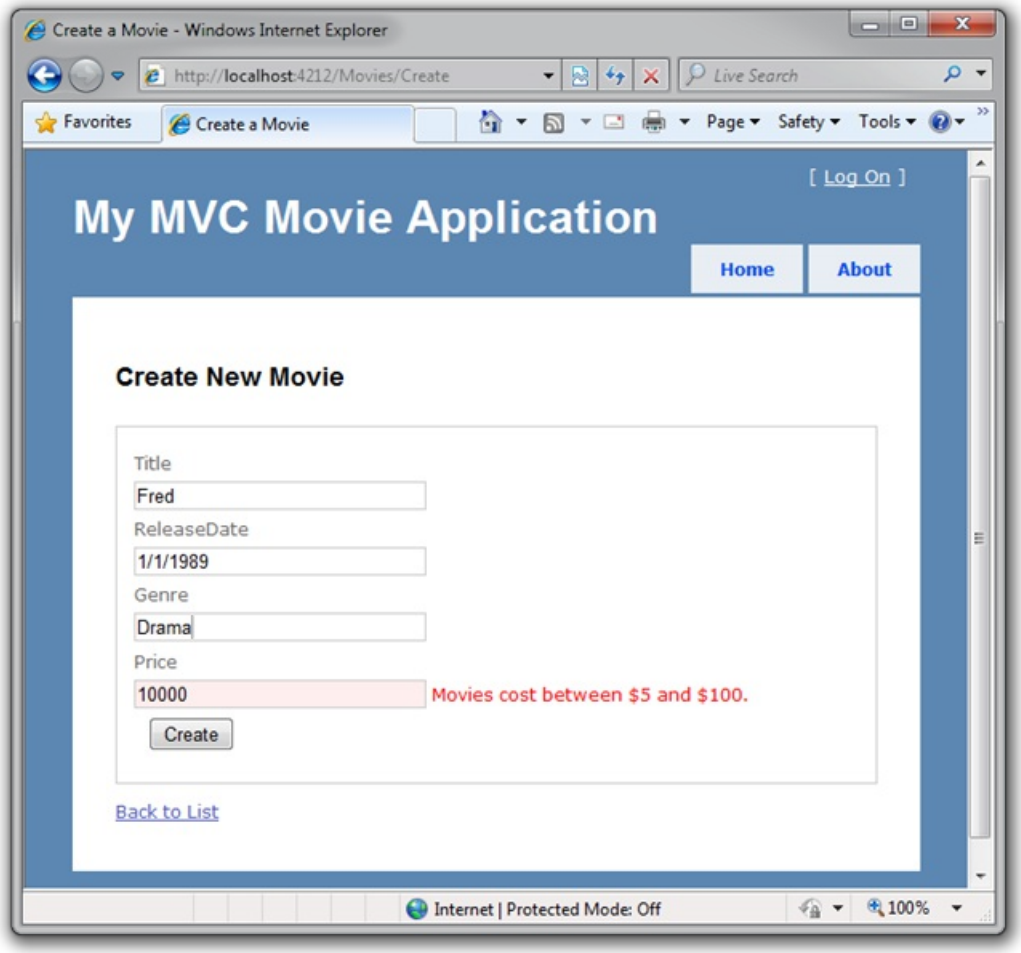
We'll create a partial movie class that points to a "buddy class" with some attributes that will give validation hints to the system. We'll mark the Title and Price as Required, and also insist that the Price be within a certain range. Right click the Models folder and select Add Class. Name your class Movie and Click the OK button. Here's what our partial Movie class looks like.

```
using System.ComponentModel.DataAnnotations;

namespace Movies.Models
{
    [MetadataType(typeof(MovieMetadata))]
    public partial class Movie
    {
        class MovieMetadata
        {
            [Required(ErrorMessage="Titles are required")]
            public string Title { get; set; }

            [Range(5,100,ErrorMessage ="Movies cost between $5 and $100.")]
            public decimal Price { get; set; }
        }
    }
}
```

Re-Run your application and try to enter a movie with a price over 100. You'll get an error after you've submitted the form. The error is caught on the server side and occurs after the Form is POSTed. Notice how ASP.NET MVC's built-in HTML helpers were smart enough to display the error message and maintain the values for us within the textbox elements:



This works great, but it'd be nice if we could tell the user on the client-side, immediately, before the server gets involved.

Let's enable some client-side validation with JavaScript.

Adding Client-Side Validation

Since our Movie class already has some validation attributes, we'll just need to add a few JavaScript files to our Create.aspx View template and add a line of code to enable client-side validation to take place.

From within VWD go our Views/Movie folder and open up Create.aspx.

Open up the Scripts folder in the Solution Explorer and drag the following three scripts to within the <head> tag.

- MicrosoftAjax.js
- MicrosoftMvcValidation.js

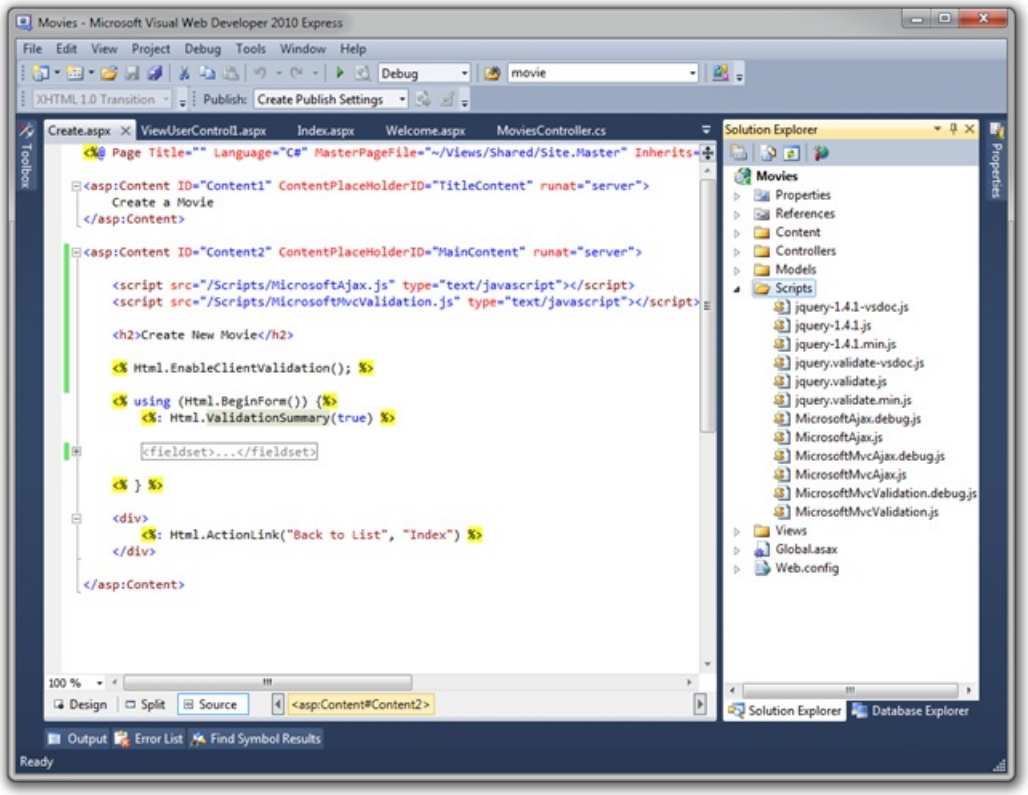
You want these script files to appear in this order.

```
<script src="/Scripts/MicrosoftAjax.js" type="text/javascript"></script>
<script src="/Scripts/MicrosoftMvcValidation.js" type="text/javascript"></script>
```

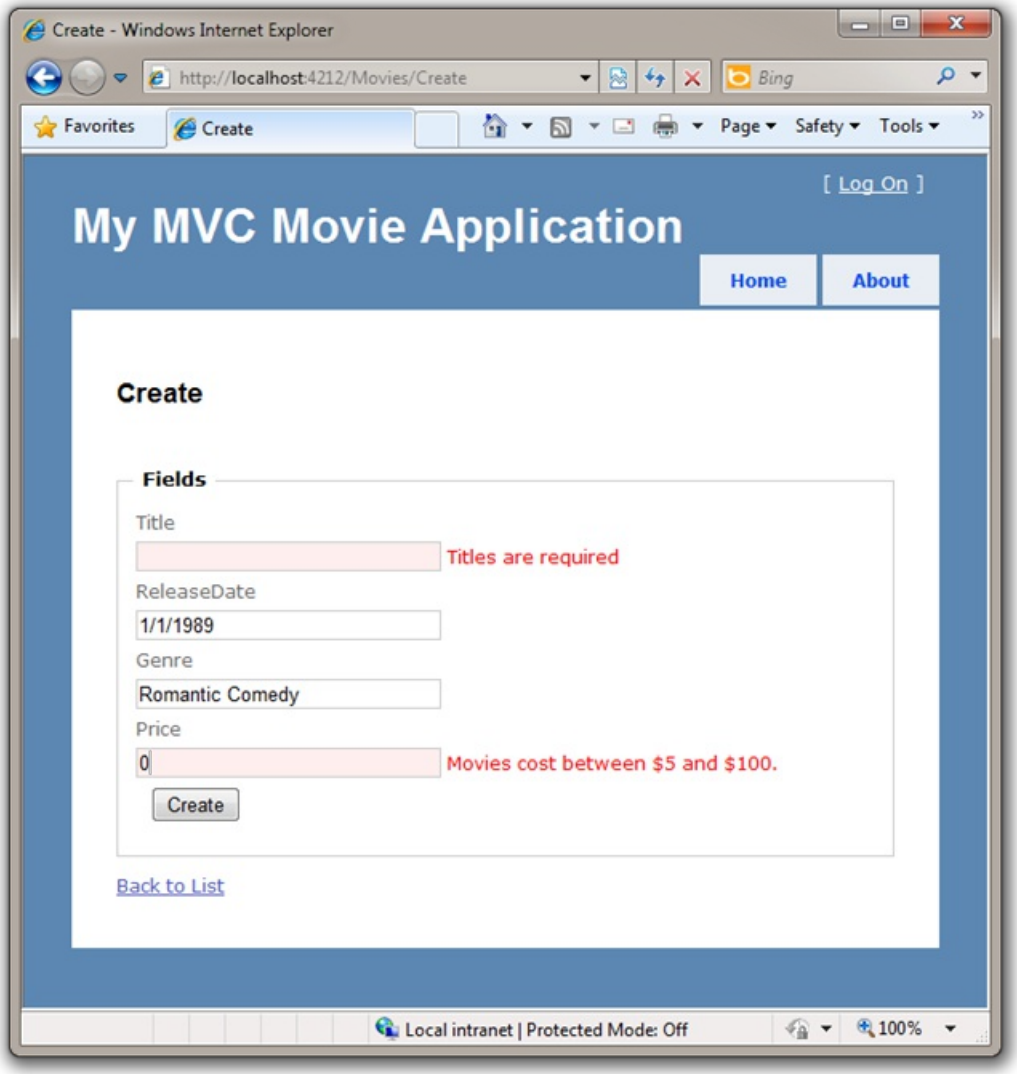
Also, add this single line above the Html.BeginForm:

```
<% Html.EnableClientValidation(); %>
```

Here's the code shown within the IDE.



Run your application and visit /Movies/Create again, and click Create without entering any data. The error messages appear immediately without the page flash that we associate with sending data all the way back to the server. This is because ASP.NET MVC is now validating the input on both the client (using JavaScript) and on the server.



This is looking good! Let's now add one additional column to the database.

PREVIOUS NEXT

Adding a Column to the Model

2/20/2020 • 4 minutes to read • Edit Online

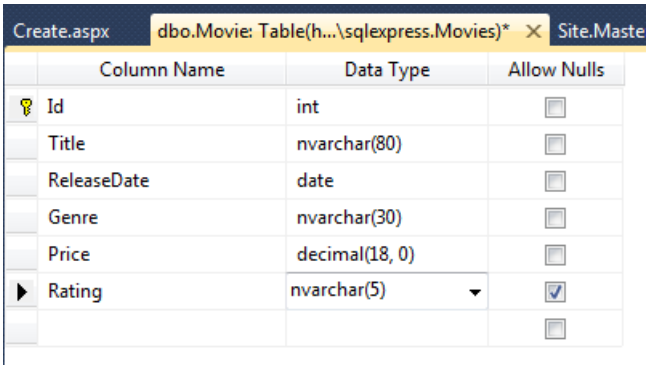
by Scott Hanselman

> This is a beginner tutorial that introduces the basics of ASP.NET MVC. You'll create a simple web application that reads and writes from a database. Visit the ASP.NET MVC learning center to find other ASP.NET MVC tutorials and samples.

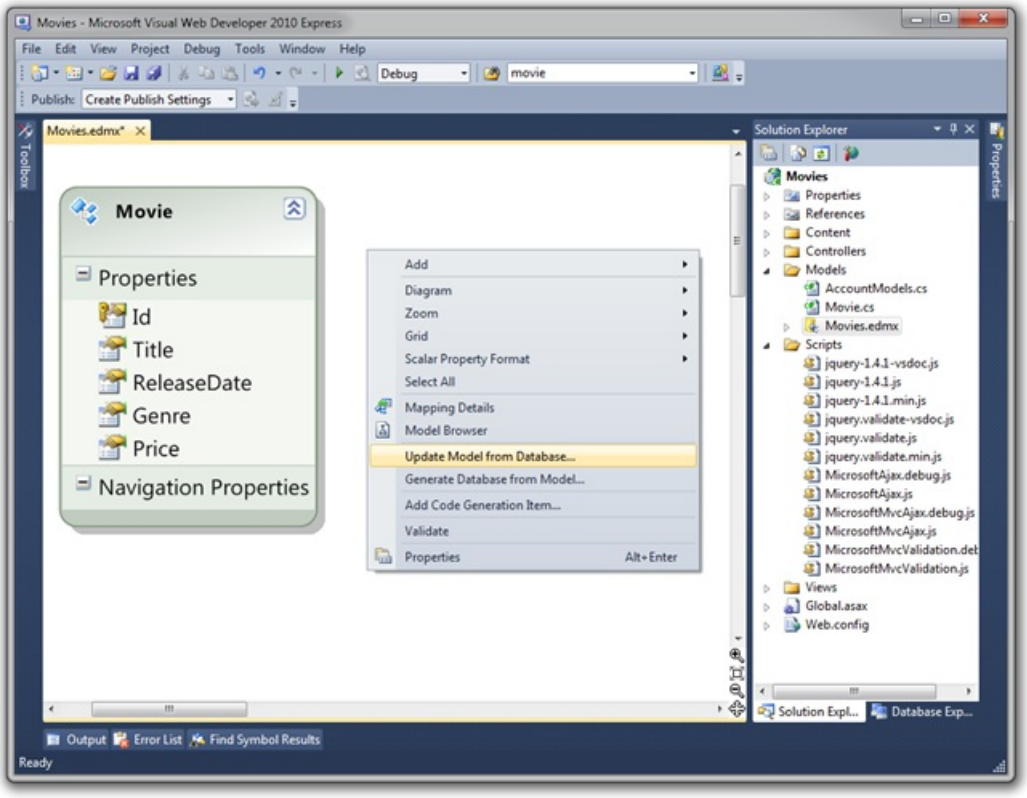
In this section we are going to walk through how we can make changes to the schema of our database, and handle the changes within our application.

Let's add a "Rating" Column to the Movie table. Go back to the IDE and click the Database Explorer. Right click the Movie table and select Open Table Definition.

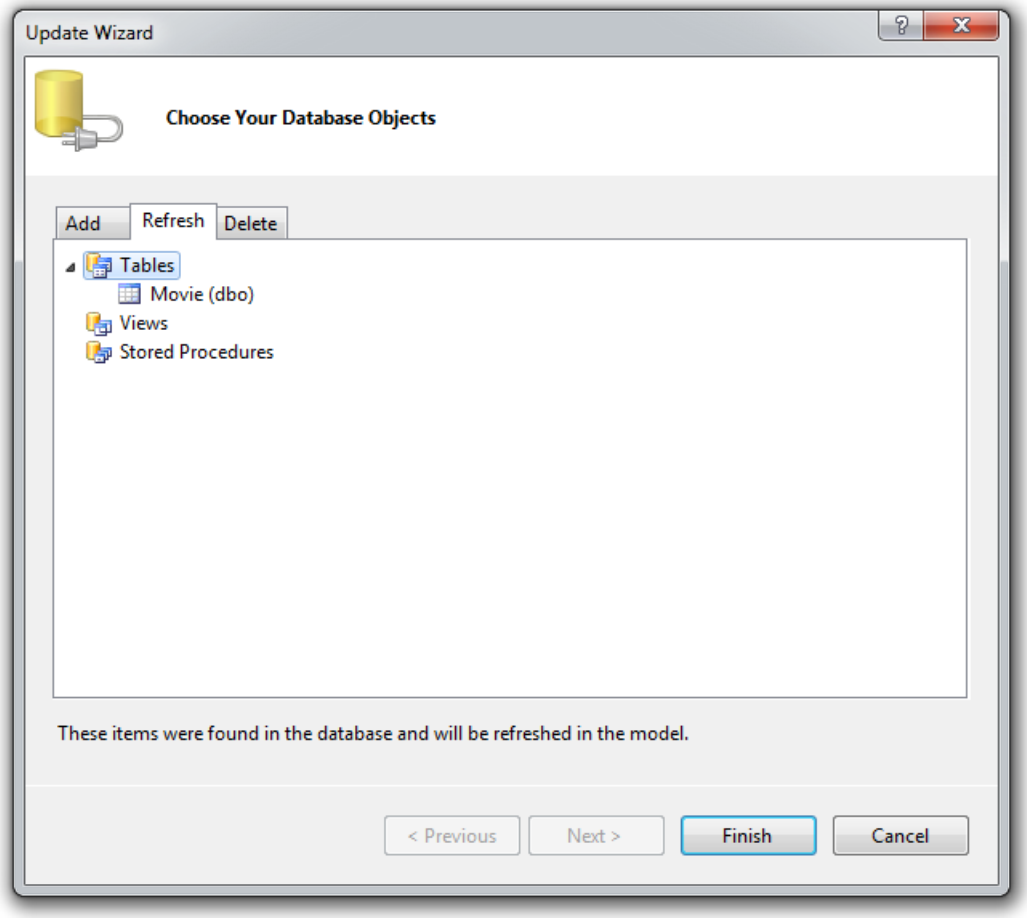
Add a "Rating" column as seen below. Since we don't have any Ratings now, the column can allow nulls. Click Save.



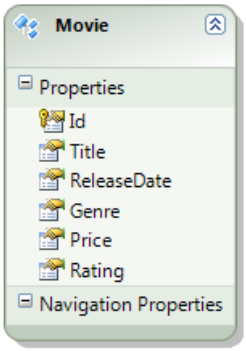
Next, return to the Solution Explorer and open up the Movies.edmx file (which is in the \Models folder). Right click on the design surface (the white area) and select Update Model from Database.



This will launch the "Update Wizard". Click the Refresh tab within it and click Finish. Our Movie model class will then be updated with the new column.



After clicking Finish, you can see the new Rating Column has been added to the Movie Entity in our model.



We've added a column in the database model, but the Views don't know about it.

Update Views with Model Changes

There are a few ways we could update our view templates to reflect the new Rating column. Since we created these Views by generating them via the Add View dialog, we could delete them and recreate them again. However, typically people will have already made modifications to their View templates from the initial scaffolded generation and will want to add or delete fields manually, just as we did with the ID field for Create.

Open up the \Views\Movies\Index.aspx template and add a <th>Rating</th> to the head of the Movie table. I added mine after Genre. Then, in the same column position but lower down, add a line to output our new Rating.

```
<td>
  <%: item.Rating %>
</td>
```

Our final Index.aspx template will look like this:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<IEnumerable<Movies.Models.Movie>>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Movie List
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>My Movie List</h2>
    <table>
        <tr>
            <th>Id</th>
            <th>Title</th>
            <th>ReleaseDate</th>
            <th>Genre</th>
            <th>Rating</th>
            <th>Price</th>
        </tr>
        <% foreach (var item in Model)
           { %>
        <tr>
            <td><%: item.Id %></td>
            <td><%: item.Title %></td>
            <td><%: String.Format("{0:g}", item.ReleaseDate) %></td>
            <td><%: item.Genre %></td>
            <td><%: item.Rating %></td>
            <td><%: String.Format("{0:F}", item.Price) %></td>
        </tr>
        <% } %>
    </table>
    <p>
        <%: Html.ActionLink("Create New", "Create") %>
    </p>
</asp:Content>
```

Let's then open up the \Views\Movies\Create.aspx template and add a Label and Textbox for our new Rating property:

```
<div class="editor-label">
    <%: Html.LabelFor(model => model.Rating) %>
</div>
<div class="editor-field">
    <%: Html.TextBoxFor(model => model.Rating)%>
    <%: Html.ValidationMessageFor(model => model.Rating)%>
</div>
```

Our final Create.aspx template will look like this, and let's change our browser's title and secondary <h2> title to something like "Create a Movie" while we're in here!

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<Movies.Models.Movie>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Create a Movie
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
```

```
    <script src="../../Scripts/MicrosoftAjax.js" type="text/javascript"></script>
    <script src="../../Scripts/MicrosoftMvcValidation.js" type="text/javascript"></script>
    <h2>Create a Movie</h2>
    <% Html.EnableClientValidation(); %>
    <% using (Html.BeginForm()) {%>
        <%: Html.ValidationSummary(true) %>

        <fieldset>
            <legend>Fields</legend>

            <div class="editor-label">
                <%: Html.LabelFor(model => model.Title) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.Title) %>
                <%: Html.ValidationMessageFor(model => model.Title) %>
            </div>

            <div class="editor-label">
                <%: Html.LabelFor(model => model.ReleaseDate) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.ReleaseDate) %>
                <%: Html.ValidationMessageFor(model => model.ReleaseDate) %>
            </div>

            <div class="editor-label">
                <%: Html.LabelFor(model => model.Genre) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.Genre) %>
                <%: Html.ValidationMessageFor(model => model.Genre) %>
            </div>

            <div class="editor-label">
                <%: Html.LabelFor(model => model.Rating) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.Rating)%>
                <%: Html.ValidationMessageFor(model => model.Rating)%>
            </div>

            <div class="editor-label">
                <%: Html.LabelFor(model => model.Price) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.Price) %>
                <%: Html.ValidationMessageFor(model => model.Price) %>
            </div>

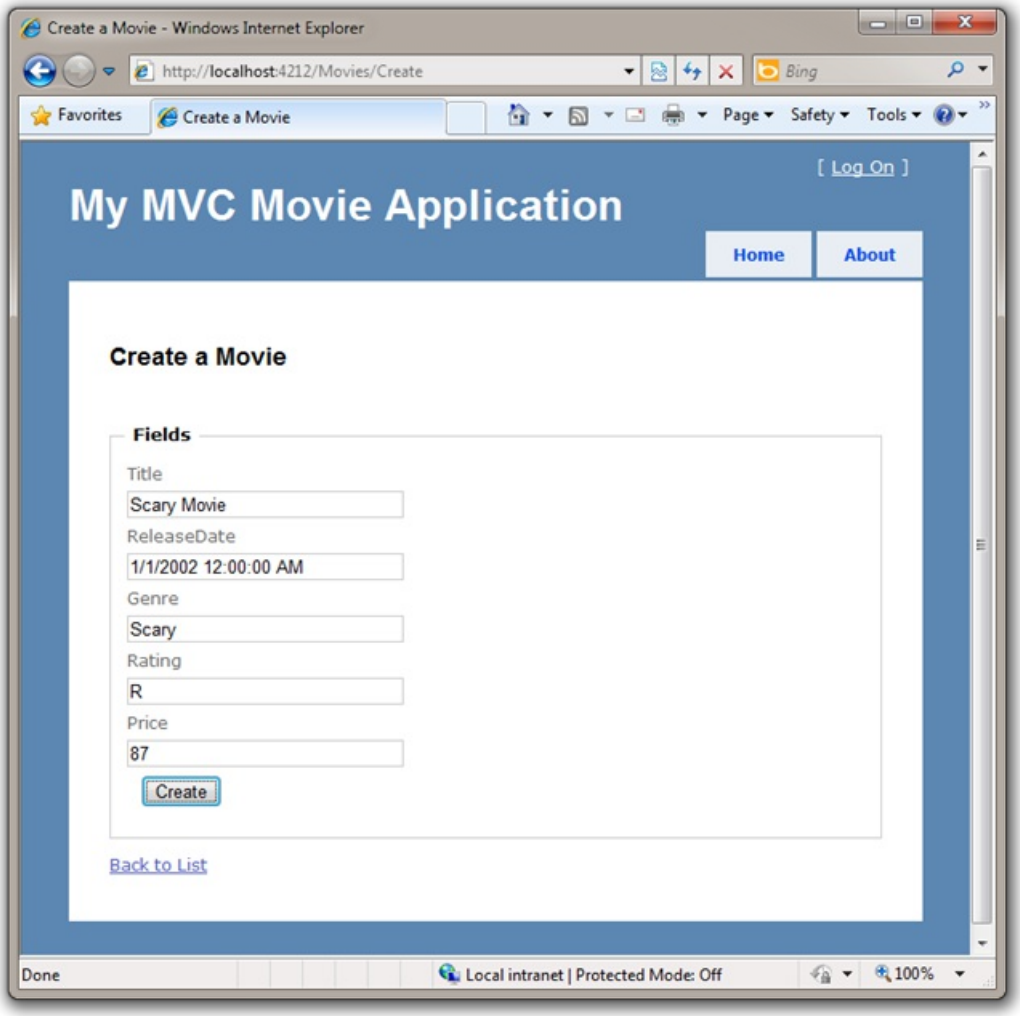
            <p>
                <input type="submit" value="Create" />
            </p>
        </fieldset>

    <% } %>

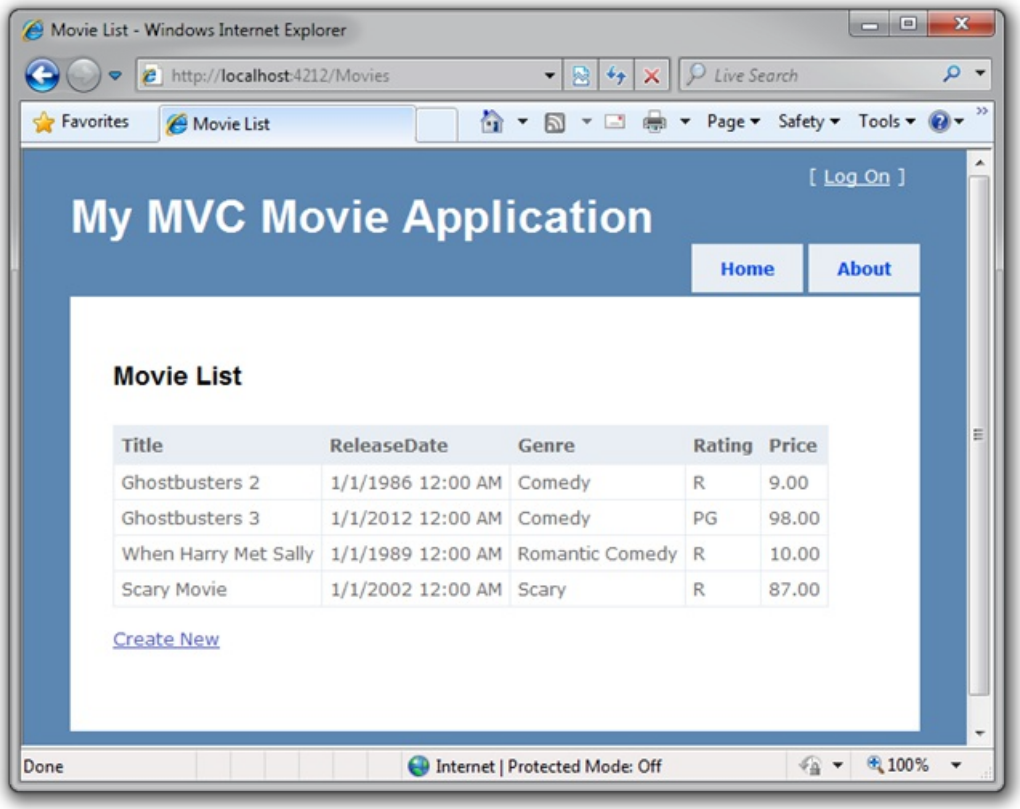
    <div>
        <%: Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

Run your app and now you've got a new field in the database that's been added to the Create page. Add a new Movie - this time with a Rating - and click Create.



After you click Create, you're sent to the Index page where you new Movie is listed with the new Rating Column in the database



This basic tutorial got you started making Controllers, associating them with Views and passing around hard-

coded data. Then we created and designed a Database and put some data it in. We retrieved the data from the database and displayed it in an HTML table. Then we added a Create form that let the user add data to the database themselves from within the Web Application. We added validation, then made the validation use JavaScript on the client-side. Finally, we changed the database to include a new column of data, then updated our two pages to create and display this new data.

I now encourage you to move on to our intermediate-level tutorial "MVC Music Store" as well as the many videos and resources at https://asp.net/mvc to learn even more about ASP.NET MVC!

Enjoy!

- Scott Hanselman - http://hanselman.com and @shanselman on Twitter.

PREVIOUS

Models (Data)

2/20/2020 • 2 minutes to read • Edit Online

In this tutorial series, you learn how to use ASP.NET MVC with the Microsoft Entity Framework. Over the course of this tutorial, you build a web application that illustrates how to select, insert, update, and delete database data by using the Entity Framework.

- Creating Model Classes with the Entity Framework (C#)
- Creating Model Classes with LINQ to SQL (C#)
- Displaying a Table of Database Data (C#)
- Performing Simple Validation (C#)
- Validating with the IDataErrorInfo Interface (C#)
- Validating with a Service Layer (C#)
- Validation with the Data Annotation Validators (C#)
- Creating Model Classes with the Entity Framework (VB)
- Creating Model Classes with LINQ to SQL (VB)
- Displaying a Table of Database Data (VB)
- Performing Simple Validation (VB)
- Validating with the IDataErrorInfo Interface (VB)
- Validating with a Service Layer (VB)
- Validation with the Data Annotation Validators (VB)

Creating Model Classes with the Entity Framework (C#)

4/10/2020 • 12 minutes to read • Edit Online

by Microsoft

> In this tutorial, you learn how to use ASP.NET MVC with the Microsoft Entity Framework. You learn how to use the Entity Wizard to create an ADO.NET Entity Data Model. Over the course of this tutorial, we build a web application that illustrates how to select, insert, update, and delete database data by using the Entity Framework.

The goal of this tutorial is to explain how you can create data access classes using the Microsoft Entity Framework when building an ASP.NET MVC application. This tutorial assumes no previous knowledge of the Microsoft Entity Framework. By the end of this tutorial, you'll understand how to use the Entity Framework to select, insert, update, and delete database records.

The Microsoft Entity Framework is an Object Relational Mapping (O/RM) tool that enables you to generate a data access layer from a database automatically. The Entity Framework enables you to avoid the tedious work of building your data access classes by hand.

In order to illustrate how you can use the Microsoft Entity Framework with ASP.NET MVC, we'll build a simple sample application. We'll create a Movie Database application that enables you to display and edit movie database records.

This tutorial assumes that you have Visual Studio 2008 or Visual Web Developer 2008 with Service Pack 1. You need Service Pack 1 in order to use the Entity Framework. You can download Visual Studio 2008 Service Pack 1 or Visual Web Developer with Service Pack 1 from the following address:

> https://www.asp.net/downloads/

NOTE

There is no essential connection between ASP.NET MVC and the Microsoft Entity Framework. There are several alternatives to the Entity Framework that you can use with ASP.NET MVC. For example, you can build your MVC Model classes using other O/RM tools such as Microsoft LINQ to SQL, NHibernate, or SubSonic.

Creating the Movie Sample Database

The Movie Database application uses a database table named Movies that contains the following columns:

COLUMN NAME	DATA TYPE	ALLOW NULLS?	IS PRIMARY KEY?
Id	int	False	True
Title	nvarchar(100)	False	False
Director	nvarchar(100)	False	False

You can add this table to an ASP.NET MVC project by following these steps:

1. Right-click the App_Data folder in the Solution Explorer window and select the menu option **Add, New Item.**
2. From the **Add New Item** dialog box, select **SQL Server Database**, give the database the name MoviesDB.mdf, and click the **Add** button.
3. Double-click the MoviesDB.mdf file to open the Server Explorer/Database Explorer window.
4. Expand the MoviesDB.mdf database connection, right-click the Tables folder, and select the menu option **Add New Table**.
5. In the Table Designer, add the Id, Title, and Director columns.
6. Click the **Save** button (it has the icon of the floppy) to save the new table with the name Movies.

After you create the Movies database table, you should add some sample data to the table. Right-click the Movies table and select the menu option **Show Table Data**. You can enter fake movie data into the grid that appears.

Creating the ADO.NET Entity Data Model

In order to use the Entity Framework, you need to create an Entity Data Model. You can take advantage of the Visual Studio *Entity Data Model Wizard* to generate an Entity Data Model from a database automatically.

Follow these steps:

1. Right-click the Models folder in the Solution Explorer window and select the menu option **Add, New Item**.
2. In the **Add New Item** dialog, select the Data category (see Figure 1).
3. Select the **ADO.NET Entity Data Model** template, give the Entity Data Model the name MoviesDBModel.edmx, and click the **Add** button. Clicking the **Add** button launches the Data Model Wizard.
4. In the **Choose Model Contents** step, choose the **Generate from a database** option and click the **Next** button (see Figure 2).
5. In the **Choose Your Data Connection** step, select the MoviesDB.mdf database connection, enter the entities connection settings name MoviesDBEntities, and click the **Next** button (see Figure 3).
6. In the **Choose Your Database Objects** step, select the Movie database table and click the **Finish** button (see Figure 4).

After you complete these steps, the ADO.NET Entity Data Model Designer (Entity Designer) opens.

Figure 1 – Creating a new Entity Data Model

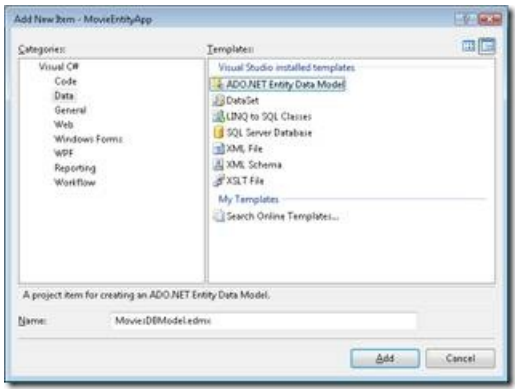


Figure 2 – Choose Model Contents Step

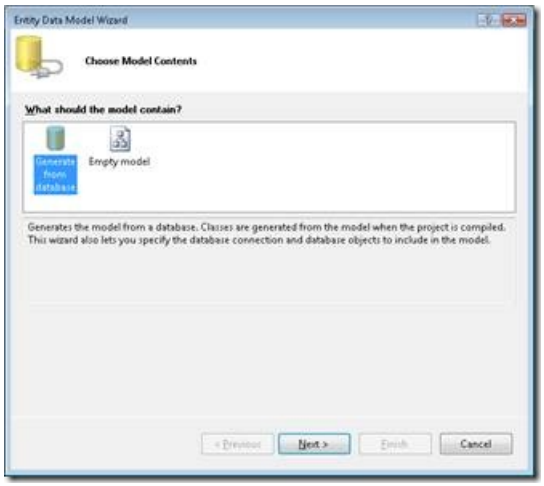
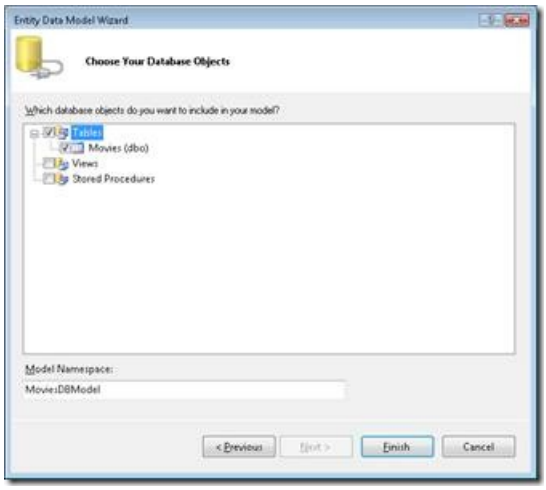


Figure 3 – Choose Your Data Connection

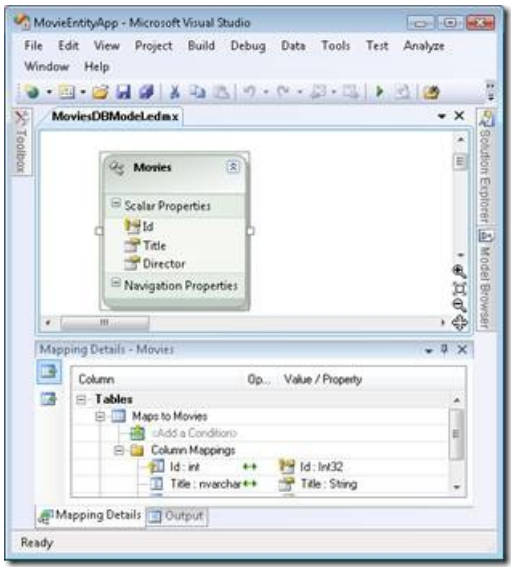
Figure 4 – Choose Your Database Objects



Modifying the ADO.NET Entity Data Model

After you create an Entity Data Model, you can modify the model by taking advantage of the Entity Designer (see Figure 5). You can open the Entity Designer at any time by double-clicking the MoviesDBModel.edmx file contained in the Models folder within the Solution Explorer window.

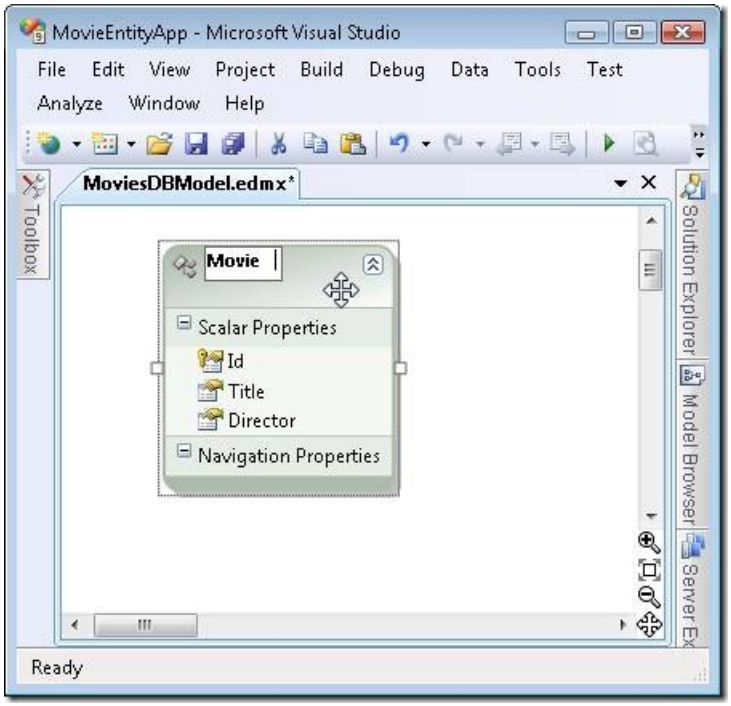
Figure 5 – The ADO.NET Entity Data Model Designer



For example, you can use the Entity Designer to change the names of the classes that the Entity Model Data Wizard generates. The Wizard created a new data access class named Movies. In other words, the Wizard gave the class the very same name as the database table. Because we will use this class to represent a particular Movie instance, we should rename the class from Movies to Movie.

If you want to rename an entity class, you can double-click on the class name in the Entity Designer and enter a new name (see Figure 6). Alternatively, you can change the name of an entity in the Properties window after selecting an entity in the Entity Designer.

Figure 6 – Changing an entity name



Remember to save your Entity Data Model after making a modification by clicking the Save button (the icon of the floppy disk). Behind the scenes, the Entity Designer generates a set of C# classes. You can view these classes by opening the MoviesDBModel.Designer.cs file from the Solution Explorer window.

Don't modify the code in the Designer.cs file since your changes will be overwritten the next time you use the Entity Designer. If you want to extend the functionality of the entity classes defined in the Designer.cs file then you can create *partial classes* in separate files.

Selecting Database Records with the Entity Framework

Let's start building our Movie Database application by creating a page that displays a list of movie records. The

Home controller in Listing 1 exposes an action named Index(). The Index() action returns all of the movie records from the Movie database table by taking advantage of the Entity Framework.

Listing 1 – Controllers\HomeController.cs

```
using System.Linq;
using System.Web.Mvc;
using MovieEntityApp.Models;

namespace MovieEntityApp.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        MoviesDBEntities _db;

        public HomeController()
        {
            _db = new MoviesDBEntities();
        }

        public ActionResult Index()
        {
            ViewData.Model = _db.MovieSet.ToList();
            return View();
        }

    }
}
```

Notice that the controller in Listing 1 includes a constructor. The constructor initializes a class-level field named _db. The _db field represents the database entities generated by the Microsoft Entity Framework. The _db field is an instance of the MoviesDBEntities class that was generated by the Entity Designer.

In order to use theMoviesDBEntities class in the Home controller, you must import the MovieEntityApp.Models namespace (*MVCProjectName*.Models).

The _db field is used within the Index() action to retrieve the records from the Movies database table. The expression _db.MovieSet represents all of the records from the Movies database table. The ToList() method is used to convert the set of movies into a generic collection of Movie objects (List<Movie>).

The movie records are retrieved with the help of LINQ to Entities. The Index() action in Listing 1 uses LINQ *method syntax* to retrieve the set of database records. If you prefer, you can use LINQ *query syntax* instead. The following two statements do the very same thing:

```
ViewData.Model = _db.MovieSet.ToList();
ViewData.Model = (from m in _db.MovieSet select m).ToList();
```

Use whichever LINQ syntax – method syntax or query syntax – that you find most intuitive. There is no difference in performance between the two approaches – the only difference is style.

The view in Listing 2 is used to display the movie records.

Listing 2 – Views\Home\Index.aspx

```
<%@ Page Language="C#"
  Inherits="System.Web.Mvc.ViewPage<List<MovieEntityApp.Models.Movie>>" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Index</title>
</head>
<body>
    <div>

<% foreach (var m in ViewData.Model)
   { %>

    Title: <%= m.Title %>
    <br />
    Director: <%= m.Director %>
    <br />
    <%= Html.ActionLink("Edit", "Edit", new { id = m.Id })%>
    <%= Html.ActionLink("Delete", "Delete", new { id = m.Id })%>

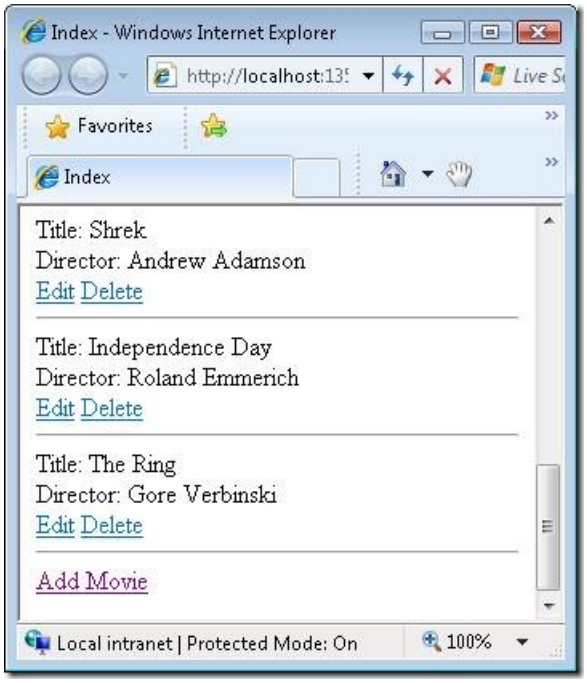
        <hr />
<% } %>

<%= Html.ActionLink("Add Movie", "Add") %>

    </div>
</body>
</html>
```

The view in Listing 2 contains a **foreach** loop that iterates through each movie record and displays the values of the movie record's Title and Director properties. Notice that an Edit and Delete link is displayed next to each record. Furthermore, an Add Movie link appears at the bottom of the view (see Figure 7).

Figure 7 – The Index view



The Index view is a *typed view*. The Index view includes a <%@ Page %> directive with an *Inherits* attribute that casts the Model property to a strongly typed generic List collection of Movie objects (List<Movie).

Inserting Database Records with the Entity Framework

You can use the Entity Framework to make it easy to insert new records into a database table. Listing 3 contains two new actions added to the Home controller class that you can use to insert new records into the Movie database table.

Listing 3 – Controllers\HomeController.cs (Add methods)

```
public ActionResult Add()
{
    return View();
}

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Add(FormCollection form)
{
    var movieToAdd = new Movie();

    // Deserialize (Include white list!)
    TryUpdateModel(movieToAdd, new string[] { "Title", "Director" }, form.ToValueProvider());

    // Validate
    if (String.IsNullOrEmpty(movieToAdd.Title))
        ModelState.AddModelError("Title", "Title is required!");
    if (String.IsNullOrEmpty(movieToAdd.Director))
        ModelState.AddModelError("Director", "Director is required!");

    // If valid, save movie to database
    if (ModelState.IsValid)
    {
        _db.AddToMovieSet(movieToAdd);
        _db.SaveChanges();
        return RedirectToAction("Index");
    }

    // Otherwise, reshow form
    return View(movieToAdd);
}
```

The first Add() action simply returns a view. The view contains a form for adding a new movie database record (see Figure 8). When you submit the form, the second Add() action is invoked.

Notice that the second Add() action is decorated with the AcceptVerbs attribute. This action can be invoked only when performing an HTTP POST operation. In other words, this action can only be invoked when posting an HTML form.

The second Add() action creates a new instance of the Entity Framework Movie class with the help of the ASP.NET MVC TryUpdateModel() method. The TryUpdateModel() method takes the fields in the FormCollection passed to the Add() method and assigns the values of these HTML form fields to the Movie class.

When using the Entity Framework, you must supply a "white list" of properties when using the TryUpdateModel or UpdateModel methods to update the properties of an entity class.

Next, the Add() action performs some simple form validation. The action verifies that both the Title and Director properties have values. If there is a validation error, then a validation error message is added to ModelState.

If there are no validation errors then a new movie record is added to the Movies database table with the help of the Entity Framework. The new record is added to the database with the following two lines of code:

```
_db.AddToMovieSet(movieToAdd);
_db.SaveChanges();
```

The first line of code adds the new Movie entity to the set of movies being tracked by the Entity Framework. The

second line of code saves whatever changes have been made to the Movies being tracked back to the underlying database.

Figure 8 – The Add view

Updating Database Records with the Entity Framework

You can follow almost the same approach to edit a database record with the Entity Framework as the approach that we just followed to insert a new database record. Listing 4 contains two new controller actions named Edit(). The first Edit() action returns an HTML form for editing a movie record. The second Edit() action attempts to update the database.

Listing 4 – Controllers\HomeController.cs (Edit methods)

```
public ActionResult Edit(int id)
{
    // Get movie to update
    var movieToUpdate = _db.MovieSet.First(m => m.Id == id);

    ViewData.Model = movieToUpdate;
    return View();
}

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(FormCollection form)
{
    // Get movie to update
    var id = Int32.Parse(form["id"]);
    var movieToUpdate = _db.MovieSet.First(m => m.Id == id);

    // Deserialize (Include white list!)
    TryUpdateModel(movieToUpdate, new string[] { "Title", "Director" }, form.ToValueProvider());

    // Validate
    if (String.IsNullOrEmpty(movieToUpdate.Title))
        ModelState.AddModelError("Title", "Title is required!");
    if (String.IsNullOrEmpty(movieToUpdate.Director))
        ModelState.AddModelError("Director", "Director is required!");

    // If valid, save movie to database
    if (ModelState.IsValid)
    {
        _db.SaveChanges();
        return RedirectToAction("Index");
    }

    // Otherwise, reshow form
    return View(movieToUpdate);
}
```

The second Edit() action starts by retrieving the Movie record from the database that matches the Id of the movie being edited. The following LINQ to Entities statement grabs the first database record that matches a particular Id:

```
var movieToUpdate = _db.MovieSet.First(m => m.Id == id);
```

Next, the TryUpdateModel() method is used to assign the values of the HTML form fields to the properties of the movie entity. Notice that a white list is supplied to specify the exact properties to update.

Next, some simple validation is performed to verify that both the Movie Title and Director properties have values. If either property is missing a value, then a validation error message is added to ModelState and ModelState.IsValid returns the value false.

Finally, if there are no validation errors, then the underlying Movies database table is updated with any changes by calling the SaveChanges() method.

When editing database records, you need to pass the Id of the record being edited to the controller action that performs the database update. Otherwise, the controller action will not know which record to update in the underlying database. The Edit view, contained in Listing 5, includes a hidden form field that represents the Id of the database record being edited.

Listing 5 – Views\Home\Edit.aspx

```
<%@ Page Language="C#"
  Inherits="System.Web.Mvc.ViewPage<MovieEntityApp.Models.Movie>" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Edit</title>
    <style type="text/css">

    .input-validation-error
    {
        background-color:Yellow;
    }

    </style>
</head>
<body>
    <div>

<h1>Edit Movie</h1>

<form method="post" action="/Home/Edit">

    <!-- Include Hidden Id -->
    <%= Html.Hidden("id") %>

    Title:
    <br />
    <%= Html.TextBox("title") %>

    <br /><br />
    Director:
    <br />
    <%= Html.TextBox("director") %>

    <br /><br />
    <input type="submit" value="Edit Movie" />
</form>

    </div>
</body>
</html>
```

Deleting Database Records with the Entity Framework

The final database operation, which we need to tackle in this tutorial, is deleting database records. You can use the controller action in Listing 6 to delete a particular database record.

Listing 6 -- \Controllers\HomeController.cs (Delete action)

```
public ActionResult Delete(int id)
{
    // Get movie to delete
    var movieToDelete = _db.MovieSet.First(m => m.Id == id);

    // Delete
    _db.DeleteObject(movieToDelete);
    _db.SaveChanges();

    // Show Index view
    return RedirectToAction("Index");
}
```

The Delete() action first retrieves the Movie entity that matches the Id passed to the action. Next, the movie is deleted from the database by calling the DeleteObject() method followed by the SaveChanges() method. Finally, the user is redirected back to the Index view.

Summary

The purpose of this tutorial was to demonstrate how you can build database-driven web applications by taking advantage of ASP.NET MVC and the Microsoft Entity Framework. You learned how to build an application that enables you to select, insert, update, and delete database records.

First, we discussed how you can use the Entity Data Model Wizard to generate an Entity Data Model from within Visual Studio. Next, you learn how to use LINQ to Entities to retrieve a set of database records from a database table. Finally, we used the Entity Framework to insert, update, and delete database records.

NEXT

Creating Model Classes with LINQ to SQL (C#)

4/10/2020 • 11 minutes to read • Edit Online

by Microsoft

Download PDF

The goal of this tutorial is to explain one method of creating model classes for an ASP.NET MVC application. In this tutorial, you learn how to build model classes and perform database access by taking advantage of Microsoft LINQ to SQL.

The goal of this tutorial is to explain one method of creating model classes for an ASP.NET MVC application. In this tutorial, you learn how to build model classes and perform database access by taking advantage of Microsoft LINQ to SQL

In this tutorial, we build a basic Movie database application. We start by creating the Movie database application in the fastest and easiest way possible. We perform all of our data access directly from our controller actions.

Next, you learn how to use the Repository pattern. Using the Repository pattern requires a little more work. However, the advantage of adopting this pattern is that it enables you to build applications that are adaptable to change and can be easily tested.

What is a Model Class?

An MVC model contains all of the application logic that is not contained in an MVC view or MVC controller. In particular, an MVC model contains all of your application business and data access logic.

You can use a variety of different technologies to implement your data access logic. For example, you can build your data access classes using the Microsoft Entity Framework, NHibernate, Subsonic, or ADO.NET classes.

In this tutorial, I use LINQ to SQL to query and update the database. LINQ to SQL provides you with a very easy method of interacting with a Microsoft SQL Server database. However, it is important to understand that the ASP.NET MVC framework is not tied to LINQ to SQL in any way. ASP.NET MVC is compatible with any data access technology.

Create a Movie Database

In this tutorial -- in order to illustrate how you can build model classes -- we build a simple Movie database application. The first step is to create a new database. Right-click the App_Data folder in the Solution Explorer window and select the menu option **Add, New Item**. Select the **SQL Server Database** template, give it the name MoviesDB.mdf, and click the **Add** button (see Figure 1).

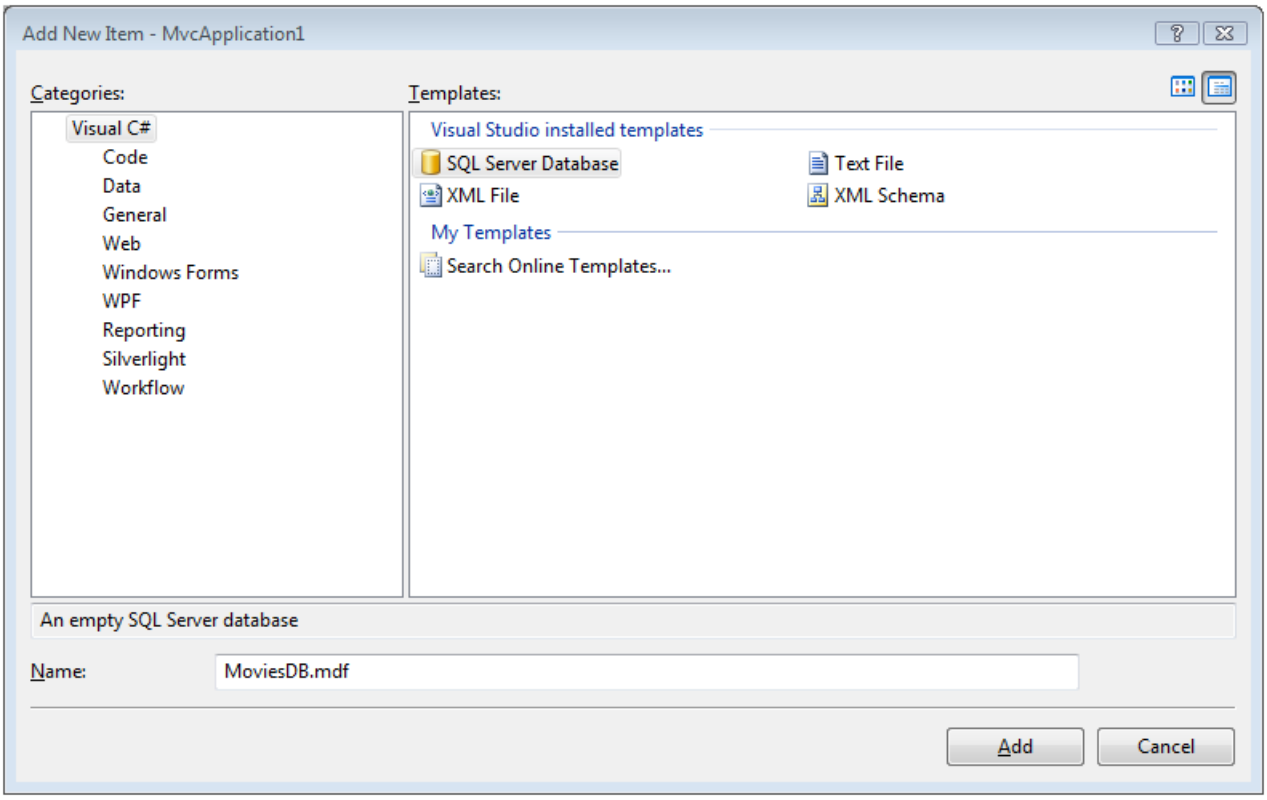


Figure 01: Adding a new SQL Server Database (Click to view full-size image)

After you create the new database, you can open the database by double-clicking the MoviesDB.mdf file in the App_Data folder. Double-clicking the MoviesDB.mdf file opens the Server Explorer window (see Figure 2).

The Server Explorer window is called the Database Explorer window when using Visual Web Developer.

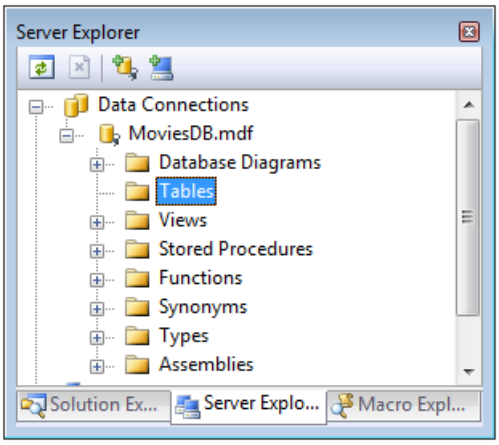


Figure 02: Using the Server Explorer window (Click to view full-size image)

We need to add one table to our database that represents our movies. Right-click the Tables folder and select the menu option **Add New Table**. Selecting this menu option opens the Table Designer (see Figure 3).

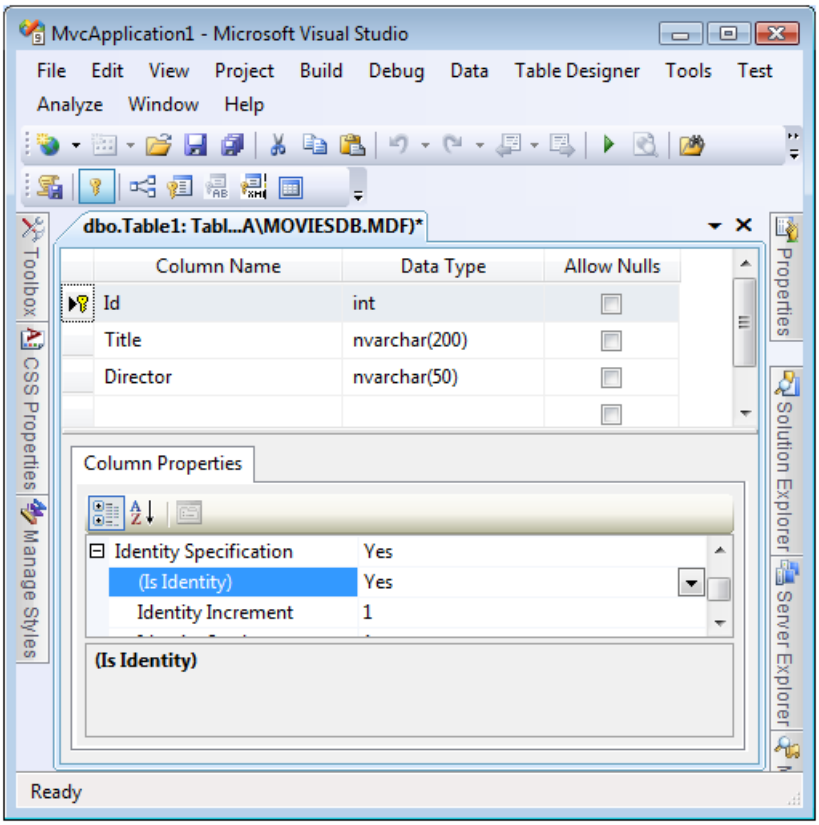


Figure 03: The Table Designer (Click to view full-size image)

We need to add the following columns to our database table:

COLUMN NAME	DATA TYPE	ALLOW NULLS
Id	Int	False
Title	Nvarchar(200)	False
Director	Nvarchar(50)	False

You need to do two special things to the Id column. First, you need to mark the Id column as a primary key column by selecting the column in the Table Designer and clicking the icon of a key. LINQ to SQL requires you to specify your primary key columns when performing inserts or updates against the database.

Next, you need to mark the Id column as an Identity column by assigning the value Yes to the **Is Identity** property (see Figure 3). An Identity column is a column that is assigned a new number automatically whenever you add a new row of data to a table.

Create LINQ to SQL Classes

Our MVC model will contain LINQ to SQL classes that represent the tblMovie database table. The easiest way to create these LINQ to SQL classes is to right-click the Models folder, select **Add, New Item**, select the LINQ to SQL Classes template, give the classes the name Movie.dbml, and click the **Add** button (see Figure 4).

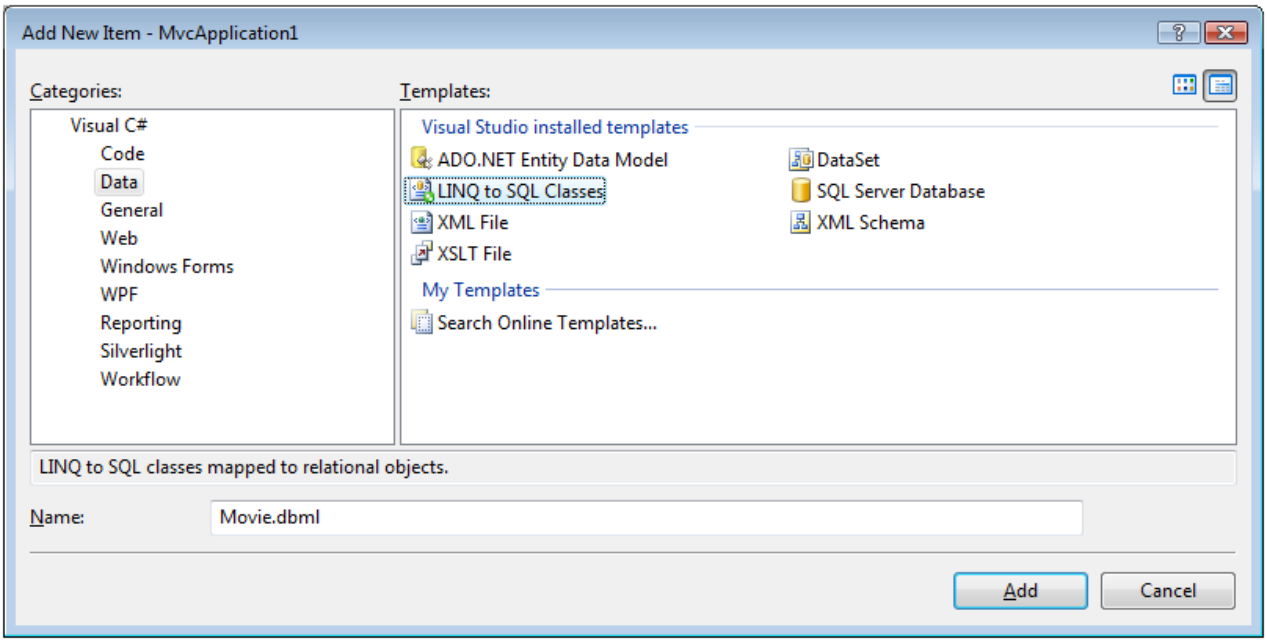


Figure 04: Creating LINQ to SQL classes (Click to view full-size image)

Immediately after you create the Movie LINQ to SQL Classes, the Object Relational Designer appears. You can drag database tables from the Server Explorer window onto the Object Relational Designer to create LINQ to SQL Classes that represent particular database tables. We need to add the tblMovie database table onto the Object Relational Designer (see Figure 5).

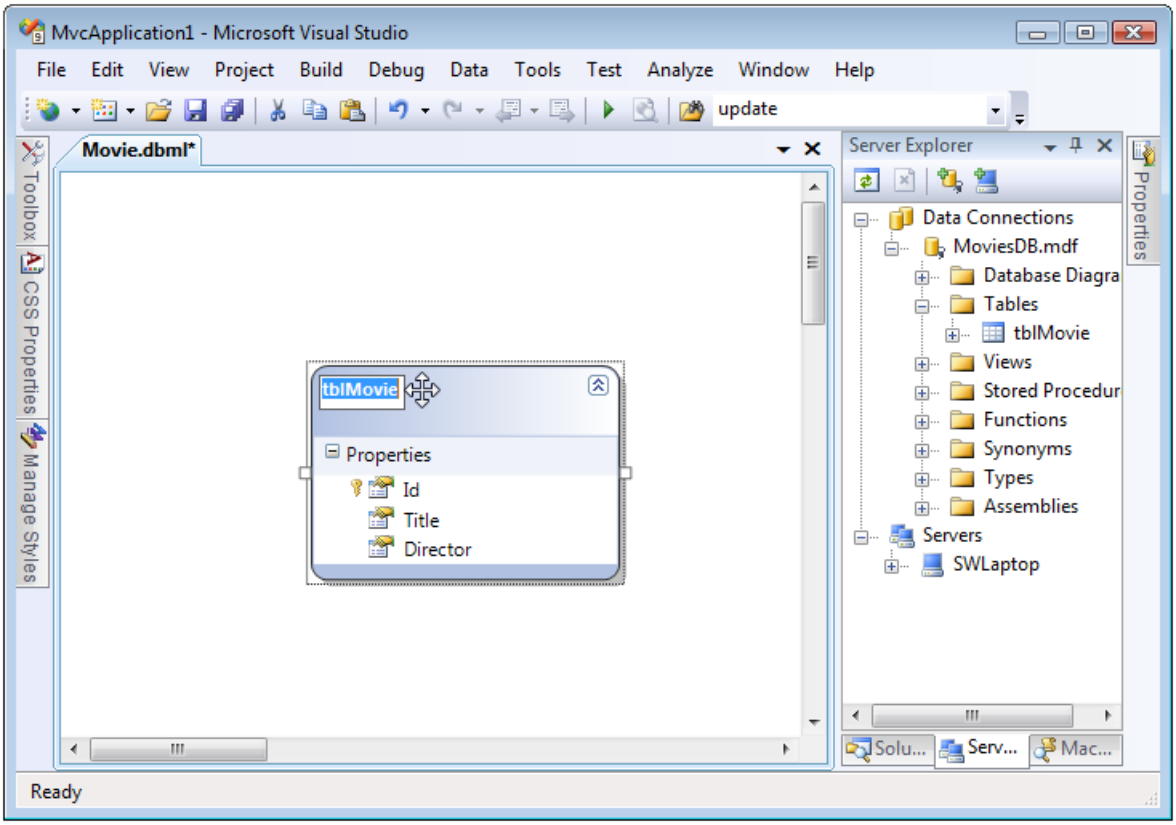


Figure 05: Using the Object Relational Designer (Click to view full-size image)

By default, the Object Relational Designer creates a class with the very same name as the database table that you drag onto the Designer. However, we don't want to call our class `tblMovie`. Therefore, click the name of the class in the Designer and change the name of the class to Movie.

Finally, remember to click the **Save** button (the picture of the floppy) to save the LINQ to SQL Classes. Otherwise, the LINQ to SQL Classes won't be generated by the Object Relational Designer.

Using LINQ to SQL in a Controller Action

Now that we have our LINQ to SQL classes, we can use these classes to retrieve data from the database. In this section, you learn how to use LINQ to SQL classes directly within a controller action. We'll display the list of movies from the tblMovies database table in an MVC view.

First, we need to modify the HomeController class. This class can be found in the Controllers folder of your application. Modify the class so it looks like the class in Listing 1.

Listing 1 – `Controllers\HomeController.cs`

```
using System.Linq;
using System.Web.Mvc;
using MvcApplication1.Models;

namespace MvcApplication1.Controllers
{
     [HandleError]
     public class HomeController : Controller
     {
          public ActionResult Index()
          {
               var dataContext = new MovieDataContext();
               var movies = from m in dataContext.Movies
                    select m;
               return View(movies);
          }
     }
}
```

The `Index()` action in Listing 1 uses a LINQ to SQL DataContext class (the `MovieDataContext`) to represent the `MoviesDB` database. The `MoveDataContext` class was generated by the Visual Studio Object Relational Designer.

A LINQ query is performed against the DataContext to retrieve all of the movies from the `tblMovies` database table. The list of movies is assigned to a local variable named `movies`. Finally, the list of movies is passed to the view through view data.

In order to show the movies, we next need to modify the Index view. You can find the Index view in the `Views\Home\` folder. Update the Index view so that it looks like the view in Listing 2.

Listing 2 – `Views\Home\Index.aspx`

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="true"
CodeBehind="Index.aspx.cs" Inherits="MvcApplication1.Views.Home.Index" %>
<%@ Import Namespace="MvcApplication1.Models" %>
<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent" runat="server">

     <ul>
          <% foreach (Movie m in (IEnumerable)ViewData.Model)

          { %>
               <li> <%= m.Title %> </li>
          <% } %>
     </ul>
</asp:Content>
```

Notice that the modified Index view includes an `<%@ import namespace %>` directive at the top of the view. This directive imports the `MvcApplication1.Models namespace`. We need this namespace in order to work with the `model` classes – in particular, the `Movie` class -- in the view.

The view in Listing 2 contains a `foreach` loop that iterates through all of the items represented by the `ViewData.Model` property. The value of the `Title` property is displayed for each `movie`.

Notice that the value of the `ViewData.Model` property is cast to an `IEnumerable`. This is necessary in order to loop through the contents of `ViewData.Model`. Another option here is to create a strongly-typed `view`. When you create a strongly-typed `view`, you cast the `ViewData.Model` property to a particular type in a view's code-behind class.

If you run the application after modifying the `HomeController` class and the Index view then you will get a blank page. You'll get a blank page because there are no movie records in the `tblMovies` database table.

In order to add records to the `tblMovies` database table, right-click the `tblMovies` database table in the Server Explorer window (Database Explorer window in Visual Web Developer) and select the menu option Show Table Data. You can insert `movie` records by using the grid that appears (see Figure 6).

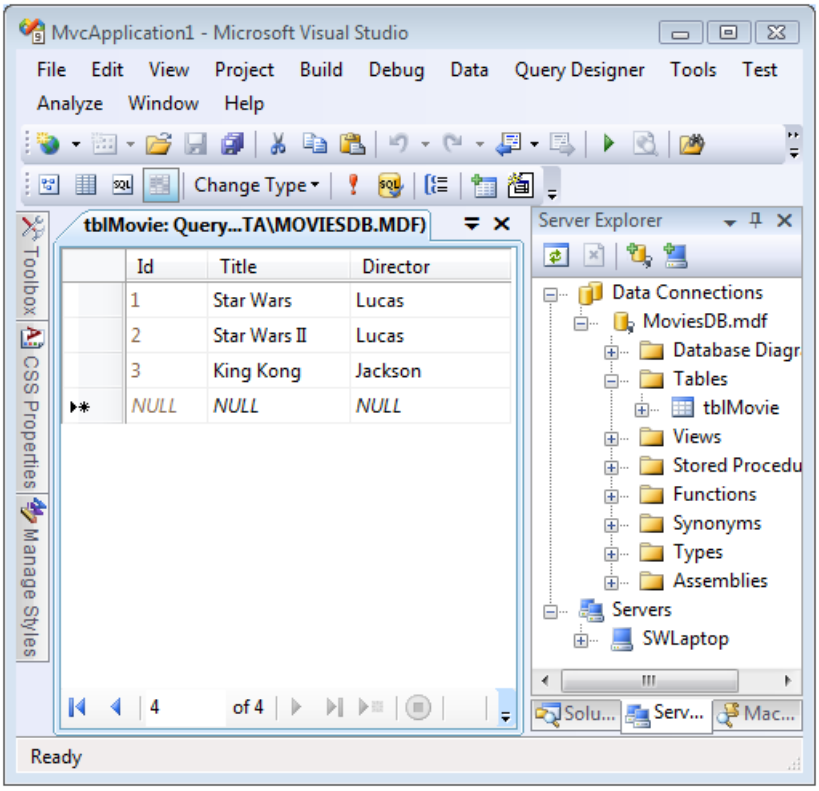


Figure 06: Inserting movies(Click to view full-size image)

After you add some database records to the `tblMovies` table, and you run the application, you'll see the page in Figure 7. All of the movie database records are displayed in a bulleted list.

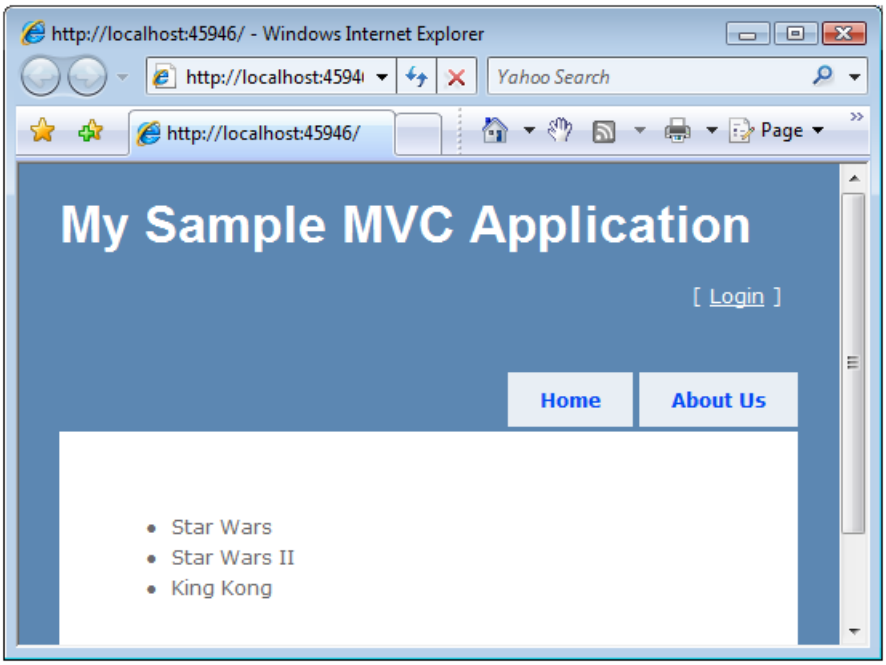


Figure 07: Displaying movies with the Index view(Click to view full-size image)

Using the Repository Pattern

In the previous section, we used LINQ to SQL classes directly within a controller action. We used the `MovieDataContext` class directly from the `Index()` controller action. There is nothing wrong with doing this in the case of a simple application. However, working directly with LINQ to SQL in a controller class creates problems when you need to build a more complex application.

Using LINQ to SQL within a controller class makes it difficult to switch data access technologies in the future. For example, you might decide to switch from using Microsoft LINQ to SQL to using the Microsoft Entity Framework as your data access technology. In that case, you would need to rewrite every controller that accesses the database within your application.

Using LINQ to SQL within a controller class also makes it difficult to build unit tests for your application. Normally, you do not want to interact with a database when performing unit tests. You want to use your unit tests to test your application logic and not your database server.

In order to build an MVC application that is more adaptable to future change and that can be more easily tested, you should consider using the Repository pattern. When you use the Repository pattern, you create a separate repository class that contains all of your database access logic.

When you create the repository class, you create an interface that represents all of the methods used by the repository class. Within your controllers, you write your code against the interface instead of the repository. That way, you can implement the repository using different data access technologies in the future.

The interface in Listing 3 is named `IMovieRepository` and it represents a single method named `ListAll()`.

Listing 3 – `Models\IMovieRepository.cs`

```
using System.Collections.Generic;
namespace MvcApplication1.Models
{
    public interface IMovieRepository
    {
        IList<Movie> ListAll();
    }
}
```

The repository class in Listing 4 implements the `IMovieRepository` interface. Notice that it contains a method named `ListAll()` that corresponds to the method required by the `IMovieRepository` interface.

Listing 4 – `Models\MovieRepository.cs`

```
using System.Collections.Generic;
using System.Linq;

namespace MvcApplication1.Models
{
     public class MovieRepository : IMovieRepository
     {
          private MovieDataContext _dataContext;

          public MovieRepository()
          {
                _dataContext = new MovieDataContext();
          }

          #region IMovieRepository Members

          public IList<Movie> ListAll()
          {
               var movies = from m in _dataContext.Movies
                    select m;
               return movies.ToList();
          }

          #endregion
     }
}
```

Finally, the `MoviesController` class in Listing 5 uses the Repository pattern. It no longer uses LINQ to SQL classes directly.

Listing 5 – `Controllers\MoviesController.cs`

```
using System.Web.Mvc;
using MvcApplication1.Models;

namespace MvcApplication1.Controllers
{
     public class MoviesController : Controller
     {
          private IMovieRepository _repository;

          public MoviesController() : this(new MovieRepository())
          {
          }

          public MoviesController(IMovieRepository repository)
          {
               _repository = repository;
          }

          public ActionResult Index()
          {
               return View(_repository.ListAll());
          }
     }
}
```

Notice that the `MoviesController` class in Listing 5 has two constructors. The first constructor, the parameterless

constructor, is called when your application is running. This constructor creates an instance of the `MovieRepository` class and passes it to the second constructor.

The second constructor has a single parameter: an `IMovieRepository` parameter. This constructor simply assigns the value of the parameter to a class-level field named `_repository`.

The `MoviesController` class is taking advantage of a software design pattern called the Dependency Injection pattern. In particular, it is using something called Constructor Dependency Injection. You can read more about this pattern by reading the following article by Martin Fowler:

http://martinfowler.com/articles/injection.html

Notice that all of the code in the `MoviesController` class (with the exception of the first constructor) interacts with the `IMovieRepository` interface instead of the actual `MovieRepository` class. The code interacts with an abstract interface instead of a concrete implementation of the interface.

If you want to modify the data access technology used by the application then you can simply implement the `IMovieRepository` interface with a class that uses the alternative database access technology. For example, you could create an `EntityFrameworkMovieRepository` class or a `SubSonicMovieRepository` class. Because the controller class is programmed against the interface, you can pass a new implementation of `IMovieRepository` to the controller class and the class would continue to work.

Furthermore, if you want to test the `MoviesController` class, then you can pass a fake movie repository class to the `HomeController`. You can implement the `IMovieRepository` class with a class that does not actually access the database but contains all of the required methods of the `IMovieRepository` interface. That way, you can unit test the `MoviesController` class without actually accessing a real database.

Summary

The goal of this tutorial was to demonstrate how you can create MVC model classes by taking advantage of Microsoft LINQ to SQL. We examined two strategies for displaying database data in an ASP.NET MVC application. First, we created LINQ to SQL classes and used the classes directly within a controller action. Using LINQ to SQL classes within a controller enables you to quickly and easily display database data in an MVC application.

Next, we explored a slightly more difficult, but definitely more virtuous, path for displaying database data. We took advantage of the Repository pattern and placed all of our database access logic in a separate repository class. In our controller, we wrote all of our code against an interface instead of a concrete class. The advantage of the Repository pattern is that it enables us to easily change database access technologies in the future and it enables us to easily test our controller classes.

PREVIOUS NEXT

Displaying a Table of Database Data (C#)

4/10/2020 • 7 minutes to read • Edit Online

by Microsoft

Download PDF

> In this tutorial, I demonstrate two methods of displaying a set of database records. I show two methods of formatting a set of database records in an HTML table. First, I show how you can format the database records directly within a view. Next, I demonstrate how you can take advantage of partials when formatting database records.

The goal of this tutorial is to explain how you can display an HTML table of database data in an ASP.NET MVC application. First, you learn how to use the scaffolding tools included in Visual Studio to generate a view that displays a set of records automatically. Next, you learn how to use a partial as a template when formatting database records.

Create the Model Classes

We are going to display the set of records from the Movies database table. The Movies database table contains the following columns:

COLUMN NAME	DATA TYPE	ALLOW NULLS
Id	Int	False
Title	Nvarchar(200)	False
Director	NVarchar(50)	False
DateReleased	DateTime	False

In order to represent the Movies table in our ASP.NET MVC application, we need to create a model class. In this tutorial, we use the Microsoft Entity Framework to create our model classes.

> **NOTE**
> In this tutorial, we use the Microsoft Entity Framework. However, it is important to understand that you can use a variety of different technologies to interact with a database from an ASP.NET MVC application including LINQ to SQL, NHibernate, or ADO.NET.

Follow these steps to launch the Entity Data Model Wizard:

1. Right-click the Models folder in the Solution Explorer window and the select the menu option **Add, New Item**.
2. Select the **Data** category and select the **ADO.NET Entity Data Model** template.
3. Give your data model the name *MoviesDBModel.edmx* and click the **Add** button.

After you click the Add button, the Entity Data Model Wizard appears (see Figure 1). Follow these steps to complete the wizard:

1. In the **Choose Model Contents** step, select the **Generate from database** option.
2. In the **Choose Your Data Connection** step, use the *MoviesDB.mdf* data connection and the name *MoviesDBEntities* for the connection settings. Click the **Next** button.
3. In the **Choose Your Database Objects** step, expand the Tables node, select the Movies table. Enter the namespace *Models* and click the **Finish** button.

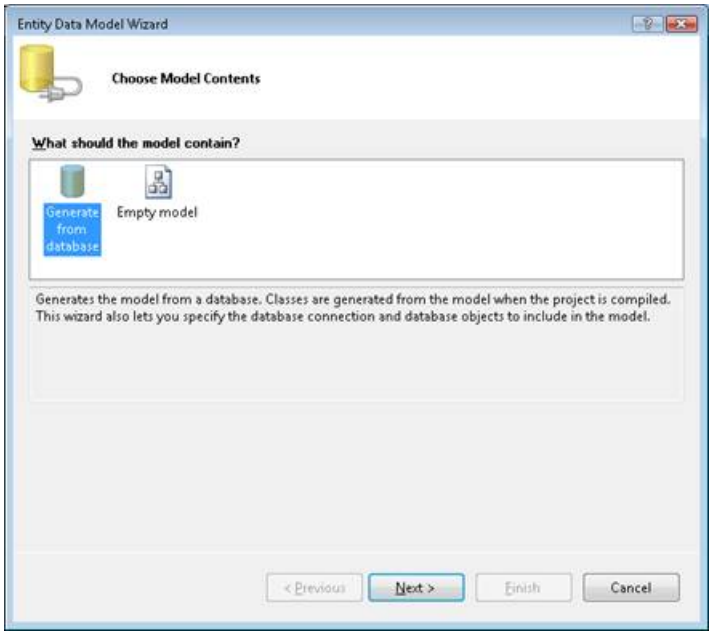


Figure 01: Creating LINQ to SQL classes (Click to view full-size image)

After you complete the Entity Data Model Wizard, the Entity Data Model Designer opens. The Designer should display the Movies entity (see Figure 2).

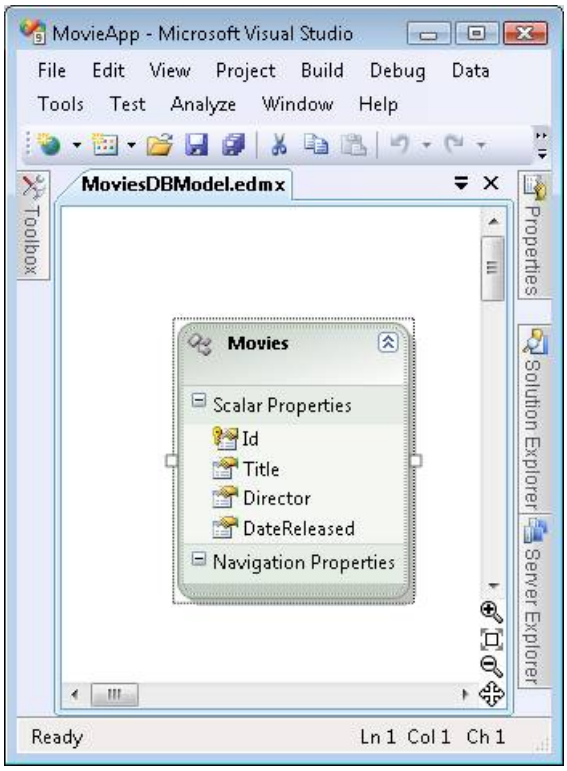


Figure 02: The Entity Data Model Designer (Click to view full-size image)

We need to make one change before we continue. The Entity Data Wizard generates a model class named *Movies* that represents the Movies database table. Because we'll use the Movies class to represent a particular movie, we need to modify the name of the class to be *Movie* instead of *Movies* (singular rather than plural).

Double-click the name of the class on the designer surface and change the name of the class from Movies to

Movie. After making this change, click the **Save** button (the icon of the floppy disk) to generate the Movie class.

Create the Movies Controller

Now that we have a way to represent our database records, we can create a controller that returns the collection of movies. Within the Visual Studio Solution Explorer window, right-click the Controllers folder and select the menu option **Add, Controller** (see Figure 3).

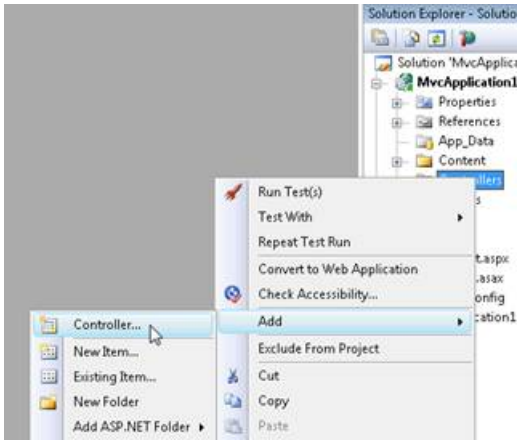


Figure 03: The Add Controller Menu (Click to view full-size image)

When the **Add Controller** dialog appears, enter the controller name MovieController (see Figure 4). Click the **Add** button to add the new controller.

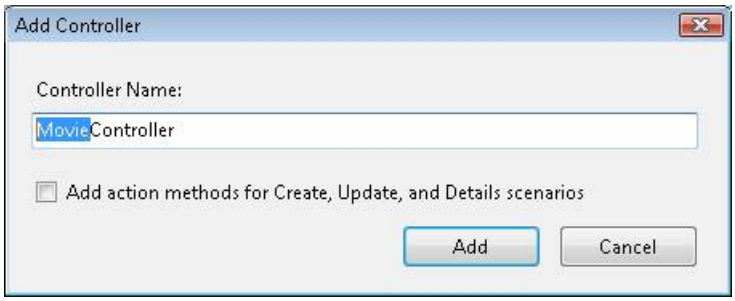


Figure 04: The Add Controller dialog(Click to view full-size image)

We need to modify the Index() action exposed by the Movie controller so that it returns the set of database records. Modify the controller so that it looks like the controller in Listing 1.

Listing 1 – Controllers\MovieController.cs

```
using System.Linq;

using System.Web.Mvc;

using MvcApplication1.Models;

namespace MvcApplication1.Controllers

{

    public class MovieController : Controller

    {

        //

        // GET: /Movie/

        public ActionResult Index()

        {

            var entities = new MoviesDBEntities();

            return View(entities.MovieSet.ToList());

        }

    }

}
```

In Listing 1, the MoviesDBEntities class is used to represent the MoviesDB database. To use this class, you need to import the MvcApplication1.Models namespace like this:

using MvcApplication1.Models;

The expression *entities.MovieSet.ToList()* returns the set of all movies from the Movies database table.

Create the View

The easiest way to display a set of database records in an HTML table is to take advantage of the scaffolding provided by Visual Studio.

Build your application by selecting the menu option **Build, Build Solution**. You must build your application before opening the **Add View** dialog or your data classes won't appear in the dialog.

Right-click the Index() action and select the menu option **Add View** (see Figure 5).

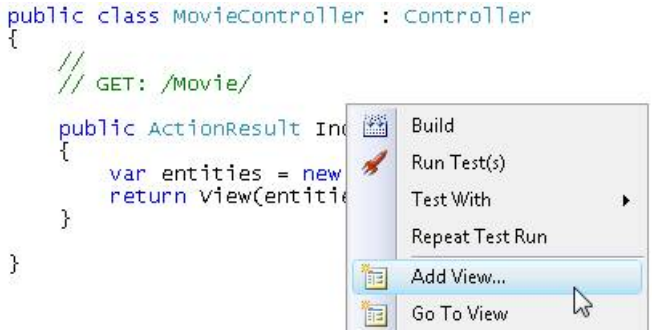


Figure 05: Adding a view (Click to view full-size image)

In the **Add View** dialog, check the checkbox labeled **Create a strongly-typed view**. Select the Movie class as

the **view data class**. Select *List* as the **view content** (see Figure 6). Selecting these options will generate a strongly-typed view that displays a list of movies.

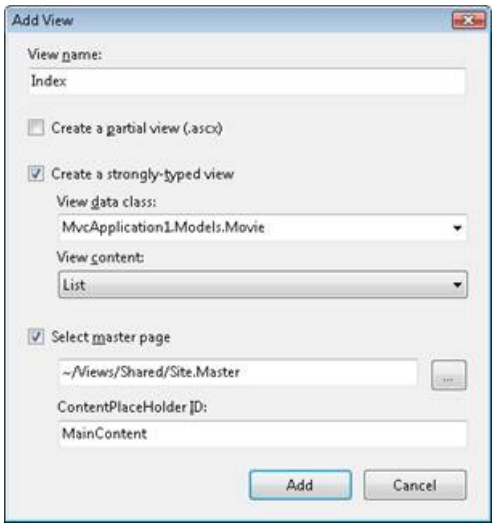


Figure 06: The Add View dialog(Click to view full-size image)

After you click the **Add** button, the view in Listing 2 is generated automatically. This view contains the code required to iterate through the collection of movies and display each of the properties of a movie.

Listing 2 – Views\Movie\Index.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<IEnumerable<MvcApplication1.Models.Movie>>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

      Index

</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Index</h2>

    <table>

        <tr>

            <th></th>

            <th>

                Id

            </th>

            <th>

                Title

            </th>

            <th>

                Director

            </th>

            <th>
```

```
                DateReleased

            </th>

        </tr>

    <% foreach (var item in Model) { %>

        <tr>

            <td>

                <%= Html.ActionLink("Edit", "Edit", new { id=item.Id }) %> |

                <%= Html.ActionLink("Details", "Details", new { id=item.Id })%>

            </td>

            <td>

                <%= Html.Encode(item.Id) %>

            </td>

            <td>

                <%= Html.Encode(item.Title) %>

            </td>

            <td>

                <%= Html.Encode(item.Director) %>

            </td>

            <td>

                <%= Html.Encode(String.Format("{0:g}", item.DateReleased)) %>

            </td>

        </tr>

    <% } %>

    </table>

    <p>

        <%= Html.ActionLink("Create New", "Create") %>

    </p>

</asp:Content>
```

You can run the application by selecting the menu option **Debug, Start Debugging** (or hitting the F5 key). Running the application launches Internet Explorer. If you navigate to the /Movie URL then you'll see the page in Figure 7.

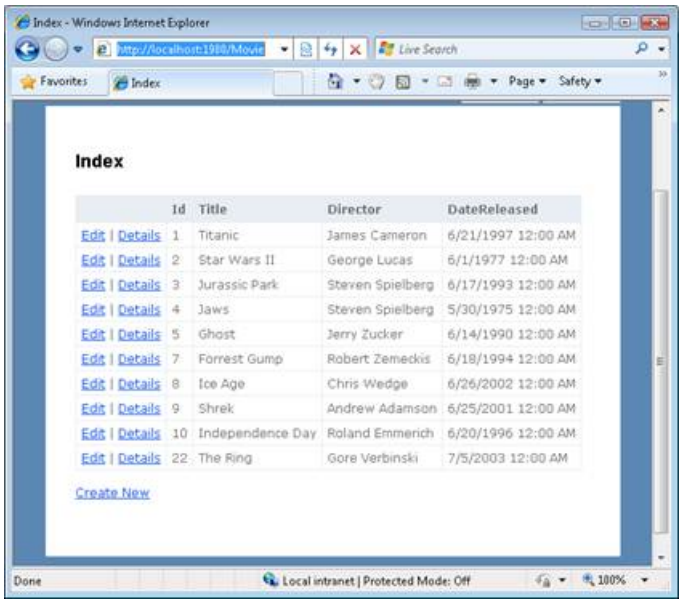


Figure 07: A table of movies(Click to view full-size image)

If you don't like anything about the appearance of the grid of database records in Figure 7 then you can simply modify the Index view. For example, you can change the *DateReleased* header to *Date Released* by modifying the Index view.

Create a Template with a Partial

When a view gets too complicated, it is a good idea to start breaking the view into partials. Using partials makes your views easier to understand and maintain. We'll create a partial that we can use as a template to format each of the movie database records.

Follow these steps to create the partial:

1. Right-click the Views\Movie folder and select the menu option **Add View**.
2. Check the checkbox labeled *Create a partial view (.ascx)*.
3. Name the partial *MovieTemplate*.
4. Check the checkbox labeled **Create a strongly-typed view**.
5. Select Movie as the *view data class*.
6. Select Empty as the *view content*.
7. Click the **Add** button to add the partial to your project.

After you complete these steps, modify the MovieTemplate partial to look like Listing 3.

Listing 3 – Views\Movie\MovieTemplate.ascx

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<MvcApplication1.Models.Movie>" %>

<tr>

    <td>

        <%= Html.Encode(Model.Id) %>

    </td>

    <td>

        <%= Html.Encode(Model.Title) %>

    </td>

    <td>

        <%= Html.Encode(Model.Director) %>

    </td>

    <td>

        <%= Html.Encode(String.Format("{0:g}", Model.DateReleased)) %>

    </td>

</tr>
```

The partial in Listing 3 contains a template for a single row of records.

The modified Index view in Listing 4 uses the MovieTemplate partial.

Listing 4 – Views\Movie\Index.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<IEnumerable<MvcApplication1.Models.Movie>>" %>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Index</h2>

    <table>

        <tr>

            <th>

                Id

            </th>

            <th>

                Title

            </th>

            <th>

                Director

            </th>

            <th>

                DateReleased

            </th>

        </tr>

    <% foreach (var item in Model) { %>

        <% Html.RenderPartial("MovieTemplate", item); %>

    <% } %>

    </table>

</asp:Content>
```

The view in Listing 4 contains a foreach loop that iterates through all of the movies. For each movie, the MovieTemplate partial is used to format the movie. The MovieTemplate is rendered by calling the RenderPartial() helper method.

The modified Index view renders the very same HTML table of database records. However, the view has been greatly simplified.

The RenderPartial() method is different than most of the other helper methods because it does not return a string. Therefore, you must call the RenderPartial() method using <% Html.RenderPartial(); %> instead of <%= Html.RenderPartial(); %>.

Summary

The goal of this tutorial was to explain how you can display a set of database records in an HTML table. First, you learned how to return a set of database records from a controller action by taking advantage of the Microsoft Entity Framework. Next, you learned how to use Visual Studio scaffolding to generate a view that displays a

collection of items automatically. Finally, you learned how to simplify the view by taking advantage of a partial. You learned how to use a partial as a template so that you can format each database record.

PREVIOUS NEXT

Performing Simple Validation (C#)

2/20/2020 • 5 minutes to read • Edit Online

by Stephen Walther

> Learn how to perform validation in an ASP.NET MVC application. In this tutorial, Stephen Walther introduces you to model state and the validation HTML helpers.

The goal of this tutorial is to explain how you can perform validation within an ASP.NET MVC application. For example, you learn how to prevent someone from submitting a form that does not contain a value for a required field. You learn how to use model state and the validation HTML helpers.

Understanding Model State

You use model state - or more accurately, the model state dictionary - to represent validation errors. For example, the Create() action in Listing 1 validates the properties of a Product class before adding the Product class to a database.

I'm not recommending that you add your validation or database logic to a controller. A controller should contain only logic related to application flow control. We are taking a shortcut to keep things simple.

Listing 1 - Controllers\ProductController.cs

```
//
// POST: /Product/Create

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create([Bind(Exclude="Id")] Product productToCreate)
{
    // Validation logic
    if (productToCreate.Name.Trim().Length == 0)
        ModelState.AddModelError("Name", "Name is required.");
    if (productToCreate.Description.Trim().Length == 0)
        ModelState.AddModelError("Description", "Description is required.");
    if (productToCreate.UnitsInStock
```

In Listing 1, the Name, Description, and UnitsInStock properties of the Product class are validated. If any of these properties fail a validation test then an error is added to the model state dictionary (represented by the ModelState property of the Controller class).

If there are any errors in model state then the ModelState.IsValid property returns false. In that case, the HTML form for creating a new product is redisplayed. Otherwise, if there are no validation errors, the new Product is added to the database.

Using the Validation Helpers

The ASP.NET MVC framework includes two validation helpers: the Html.ValidationMessage() helper and the Html.ValidationSummary() helper. You use these two helpers in a view to display validation error messages.

The Html.ValidationMessage() and Html.ValidationSummary() helpers are used in the Create and Edit views that are generated automatically by the ASP.NET MVC scaffolding. Follow these steps to generate the Create view:

1. Right-click the Create() action in the Product controller and select the menu option **Add View** (see Figure 1).

2. In the **Add View** dialog, check the checkbox labeled **Create a strongly-typed view** (see Figure 2).
3. From the **View data class** dropdown list, select the Product class.
4. From the **View content** dropdown list, select Create.
5. Click the **Add** button.

Make sure that you build your application before adding a view. Otherwise, the list of classes won't appear in the **View data class** dropdown list.

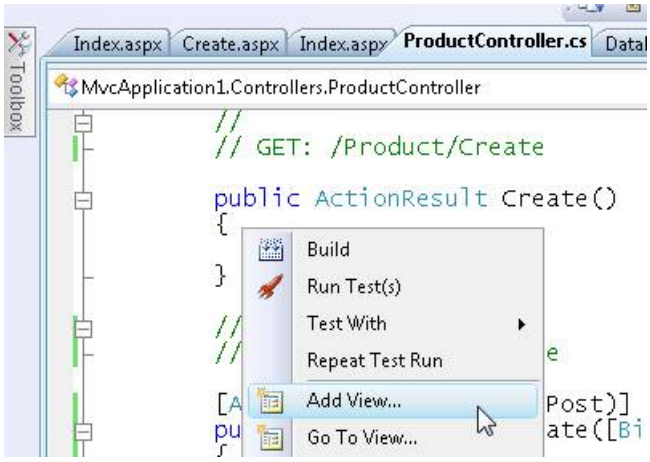


Figure 01: Adding a view(Click to view full-size image)

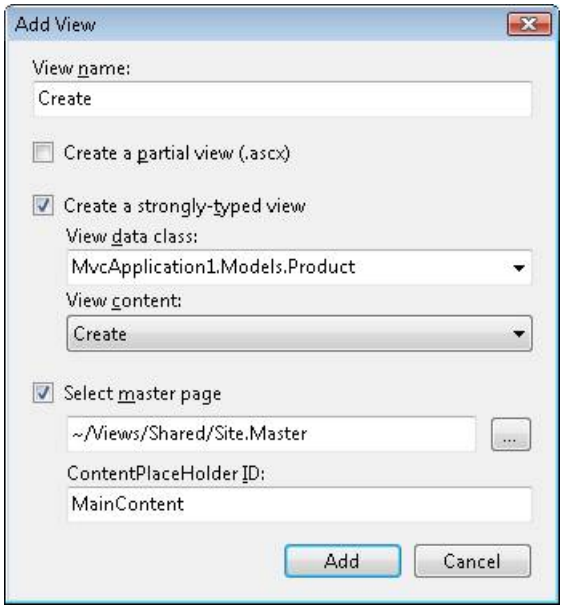


Figure 02: Creating a strongly-typed view (Click to view full-size image)

After you complete these steps, you get the Create view in Listing 2.

Listing 2 - Views\Product\Create.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcApplication1.Models.Product>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Create</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Create</h2>

    <%= Html.ValidationSummary() %>

    <% using (Html.BeginForm()) {%>

        <fieldset>
            <legend>Fields</legend>
            <p>
                <label for="Name">Name:</label>
                <%= Html.TextBox("Name") %>
                <%= Html.ValidationMessage("Name", "*") %>
            </p>
            <p>
                <label for="Description">Description:</label>
                <%= Html.TextBox("Description") %>
                <%= Html.ValidationMessage("Description", "*") %>
            </p>
            <p>
                <label for="Price">Price:</label>
                <%= Html.TextBox("Price") %>
                <%= Html.ValidationMessage("Price", "*") %>
            </p>
            <p>
                <label for="UnitsInStock">UnitsInStock:</label>
                <%= Html.TextBox("UnitsInStock") %>
                <%= Html.ValidationMessage("UnitsInStock", "*") %>
            </p>
            <p>
                <input type="submit" value="Create" />
            </p>
        </fieldset>

    <% } %>

    <div>
        <%=Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

In Listing 2, the Html.ValidationSummary() helper is called immediately above the HTML form. This helper is used to display a list of validation error messages. The Html.ValidationSummary() helper renders the errors in a bulleted list.

The Html.ValidationMessage() helper is called next to each of the HTML form fields. This helper is used to display an error message right next to a form field. In the case of Listing 2, the Html.ValidationMessage() helper displays an asterisk when there is an error.

The page in Figure 3 illustrates the error messages rendered by the validation helpers when the form is submitted with missing fields and invalid values.

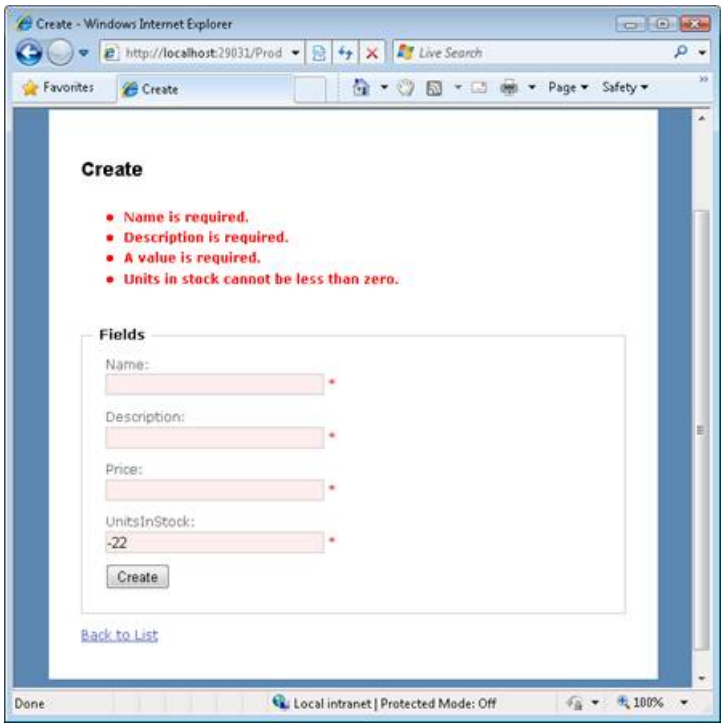


Figure 03: The Create view submitted with problems (Click to view full-size image)

Notice that the appearance of the HTML input fields are also modified when there is a validation error. The Html.TextBox() helper renders a *class="input-validation-error"* attribute when there is a validation error associated with the property rendered by the Html.TextBox() helper.

There are three cascading style sheet classes used to control the appearance of validation errors:

- input-validation-error - Applied to the <input> tag rendered by Html.TextBox() helper.
- field-validation-error - Applied to the tag rendered by the Html.ValidationMessage() helper.
- validation-summary-errors - Applied to the <ul> tag rendered by the Html.ValidationSummary() helper.

You can modify these cascading style sheet classes, and therefore modify the appearance of the validation errors, by modifying the Site.css file located in the Content folder.

> **NOTE**
>
> The HtmlHelper class includes read-only static properties for retrieving the names of the validation related CSS classes. These static properties are named ValidationInputCssClassName, ValidationFieldCssClassName, and ValidationSummaryCssClassName.

Prebinding Validation and Postbinding Validation

If you submit the HTML form for creating a Product, and you enter an invalid value for the price field and no value for the UnitsInStock field, then you'll get the validation messages displayed in Figure 4. Where do these validation error messages come from?

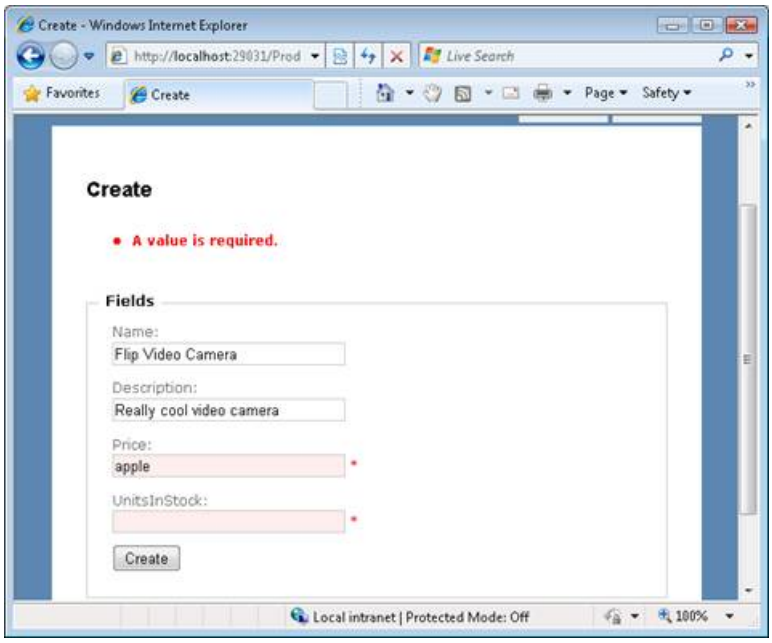


Figure 04: Prebinding Validation Errors(Click to view full-size image)

There are actually two types of validation error messages - those generated before the HTML form fields are bound to a class and those generated after the form fields are bound to the class. In other words, there are prebinding validation errors and postbinding validation errors.

The Create() action exposed by the Product controller in Listing 1 accepts an instance of the Product class. The signature of the Create method looks like this:

```
public ActionResult Create([Bind(Exclude="Id")] Product productToCreate)
```

The values of the HTML form fields from the Create form are bound to the productToCreate class by something called a model binder. The default model binder adds an error message to model state automatically when it cannot bind a form field to a form property.

The default model binder cannot bind the string "apple" to the Price property of the Product class. You can't assign a string to a decimal property. Therefore, the model binder adds an error to model state.

The default model binder also cannot assign a null value to a property that does not accept nulls. In particular, the model binder cannot assign a null value to the UnitsInStock property. Once again, the model binder gives up and adds an error message to model state.

If you want to customize the appearance of these prebinding error messages then you need to create resource strings for these messages.

Summary

The goal of this tutorial was to describe the basic mechanics of validation in the ASP.NET MVC framework. You learned how to use model state and the validation HTML helpers. We also discussed the distinction between prebinding and postbinding validation. In other tutorials, we'll discuss various strategies for moving your validation code out of your controllers and into your model classes.

PREVIOUS NEXT

Validating with the IDataErrorInfo Interface (C#)

2/20/2020 • 7 minutes to read • Edit Online

by Stephen Walther

> Stephen Walther shows you how to display custom validation error messages by implementing the IDataErrorInfo interface in a model class.

The goal of this tutorial is to explain one approach to performing validation in an ASP.NET MVC application. You learn how to prevent someone from submitting an HTML form without providing values for required form fields. In this tutorial, you learn how to perform validation by using the IErrorDataInfo interface.

Assumptions

In this tutorial, I'll use the MoviesDB database and the Movies database table. This table has the following columns:

COLUMN NAME	DATA TYPE	ALLOW NULLS
Id	Int	False
Title	Nvarchar(100)	False
Director	Nvarchar(100)	False
DateReleased	DateTime	False

In this tutorial, I use the Microsoft Entity Framework to generate my database model classes. The Movie class generated by the Entity Framework is displayed in Figure 1.

Figure 01: The Movie entity(Click to view full-size image)

> **NOTE**
>
> To learn more about using the Entity Framework to generate your database model classes, see my tutorial entitled Creating Model Classes with the Entity Framework.

The Controller Class

We use the Home controller to list movies and create new movies. The code for this class is contained in Listing 1.

Listing 1 - Controllers\HomeController.cs

```
using System.Linq;
using System.Web.Mvc;
using MvcApplication1.Models;

namespace MvcApplication1.Controllers
{

    public class HomeController : Controller
    {
        private MoviesDBEntities _db = new MoviesDBEntities();

        public ActionResult Index()
        {
            return View(_db.MovieSet.ToList());
        }

        public ActionResult Create()
        {
            return View();
        }

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create([Bind(Exclude = "Id")] Movie movieToCreate)
        {
            // Validate
            if (!ModelState.IsValid)
                return View();

            // Add to database
            try
            {
                _db.AddToMovieSet(movieToCreate);
                _db.SaveChanges();

                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }

    }
}
```

The Home controller class in Listing 1 contains two Create() actions. The first action displays the HTML form for creating a new movie. The second Create() action performs the actual insert of the new movie into the database. The second Create() action is invoked when the form displayed by the first Create() action is submitted to the server.

Notice that the second Create() action contains the following lines of code:

```
// Validate
if (!ModelState.IsValid)
    return View();
```

The IsValid property returns false when there is a validation error. In that case, the Create view that contains the HTML form for creating a movie is redisplayed.

Creating a Partial Class

The Movie class is generated by the Entity Framework. You can see the code for the Movie class if you expand the MoviesDBModel.edmx file in the Solution Explorer window and open the MoviesDBModel.Designer.cs file in the Code Editor (see Figure 2).

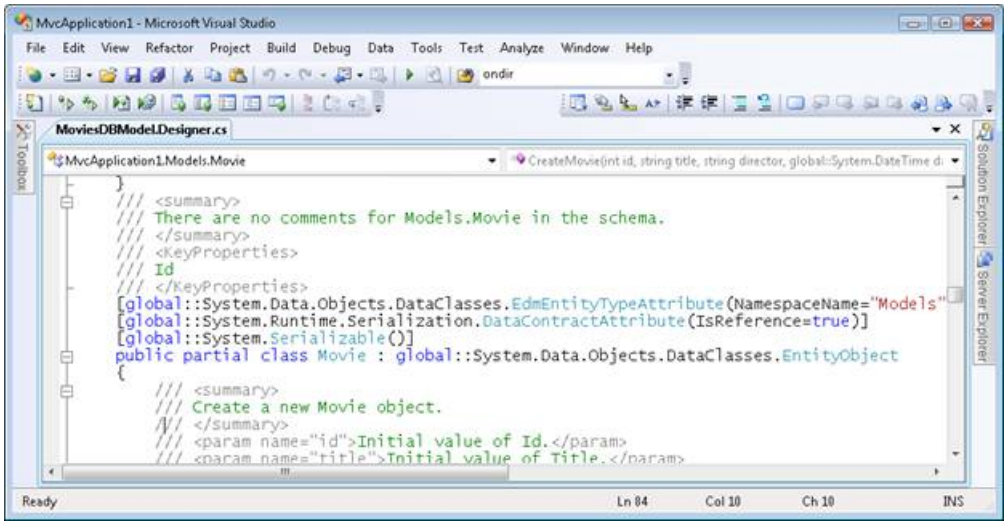


Figure 02: The code for the Movie entity(Click to view full-size image)

The Movie class is a partial class. That means that we can add another partial class with the same name to extend the functionality of the Movie class. We'll add our validation logic to the new partial class.

Add the class in Listing 2 to the Models folder.

Listing 2 - Models\Movie.cs

```
using System.Collections.Generic;
using System.ComponentModel;

namespace MvcApplication1.Models
{

    public partial class Movie
    {

    }
}
```

Notice that the class in Listing 2 includes the *partial* modifier. Any methods or properties that you add to this class become part of the Movie class generated by the Entity Framework.

Adding OnChanging and OnChanged Partial Methods

When the Entity Framework generates an entity class, the Entity Framework adds partial methods to the class automatically. The Entity Framework generates OnChanging and OnChanged partial methods that correspond to each property of the class.

In the case of the Movie class, the Entity Framework creates the following methods:

- OnIdChanging
- OnIdChanged
- OnTitleChanging
- OnTitleChanged
- OnDirectorChanging
- OnDirectorChanged

- OnDateReleasedChanging
- OnDateReleasedChanged

The OnChanging method is called right before the corresponding property is changed. The OnChanged method is called right after the property is changed.

You can take advantage of these partial methods to add validation logic to the Movie class. The update Movie class in Listing 3 verifies that the Title and Director properties are assigned nonempty values.

> **NOTE**
> A partial method is a method defined in a class that you are not required to implement. If you don't implement a partial method then the compiler removes the method signature and all calls to the method so there are no run-time costs associated with the partial method. In the Visual Studio Code Editor, you can add a partial method by typing the keyword *partial* followed by a space to view a list of partials to implement.

Listing 3 - Models\Movie.cs

```
using System.Collections.Generic;
using System.ComponentModel;

namespace MvcApplication1.Models
{

    public partial class Movie : IDataErrorInfo
    {
        private Dictionary<string, string> _errors = new Dictionary<string, string>();

        partial void OnTitleChanging(string value)
        {
            if (value.Trim().Length == 0)
                _errors.Add("Title", "Title is required.");
        }

        partial void OnDirectorChanging(string value)
        {
            if (value.Trim().Length == 0)
                _errors.Add("Director", "Director is required.");
        }

    }
}
```

For example, if you attempt to assign an empty string to the Title property, then an error message is assigned to a Dictionary named _errors.

At this point, nothing actually happens when you assign an empty string to the Title property and an error is added to the private _errors field. We need to implement the IDataErrorInfo interface to expose these validation errors to the ASP.NET MVC framework.

Implementing the IDataErrorInfo Interface

The IDataErrorInfo interface has been part of the .NET framework since the first version. This interface is a very simple interface:

```
public interface IDataErrorInfo
{
    string this[string columnName] { get; }

    string Error { get; }
}
```

If a class implements the IDataErrorInfo interface, the ASP.NET MVC framework will use this interface when creating an instance of the class. For example, the Home controller Create() action accepts an instance of the Movie class:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create([Bind(Exclude = "Id")] Movie movieToCreate)
{
    // Validate
    if (!ModelState.IsValid)
        return View();

    // Add to database
    try
    {
        _db.AddToMovieSet(movieToCreate);
        _db.SaveChanges();

        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

The ASP.NET MVC framework creates the instance of the Movie passed to the Create() action by using a model binder (the DefaultModelBinder). The model binder is responsible for creating an instance of the Movie object by binding the HTML form fields to an instance of the Movie object.

The DefaultModelBinder detects whether or not a class implements the IDataErrorInfo interface. If a class implements this interface then the model binder invokes the IDataErrorInfo.this indexer for each property of the class. If the indexer returns an error message then the model binder adds this error message to model state automatically.

The DefaultModelBinder also checks the IDataErrorInfo.Error property. This property is intended to represent non-property specific validation errors associated with the class. For example, you might want to enforce a validation rule that depends on the values of multiple properties of the Movie class. In that case, you would return a validation error from the Error property.

The updated Movie class in Listing 4 implements the IDataErrorInfo interface.

Listing 4 - Models\Movie.cs (implements IDataErrorInfo)

```
using System.Collections.Generic;
using System.ComponentModel;

namespace MvcApplication1.Models
{

    public partial class Movie : IDataErrorInfo
    {
        private Dictionary<string, string> _errors = new Dictionary<string, string>();

        partial void OnTitleChanging(string value)
        {
            if (value.Trim().Length == 0)
                _errors.Add("Title", "Title is required.");
        }

        partial void OnDirectorChanging(string value)
        {
            if (value.Trim().Length == 0)
                _errors.Add("Director", "Director is required.");
        }

        #region IDataErrorInfo Members

        public string Error
        {
            get
            {
                return string.Empty;
            }
        }

        public string this[string columnName]
        {
            get
            {
                if (_errors.ContainsKey(columnName))
                    return _errors[columnName];
                return string.Empty;
            }
        }

        #endregion
    }
}
```

In Listing 4, the indexer property checks the _errors collection to see if it contains a key that corresponds to the property name passed to the indexer. If there is no validation error associated with the property then an empty string is returned.

You don't need to modify the Home controller in any way to use the modified Movie class. The page displayed in Figure 3 illustrates what happens when no value is entered for the Title or Director form fields.

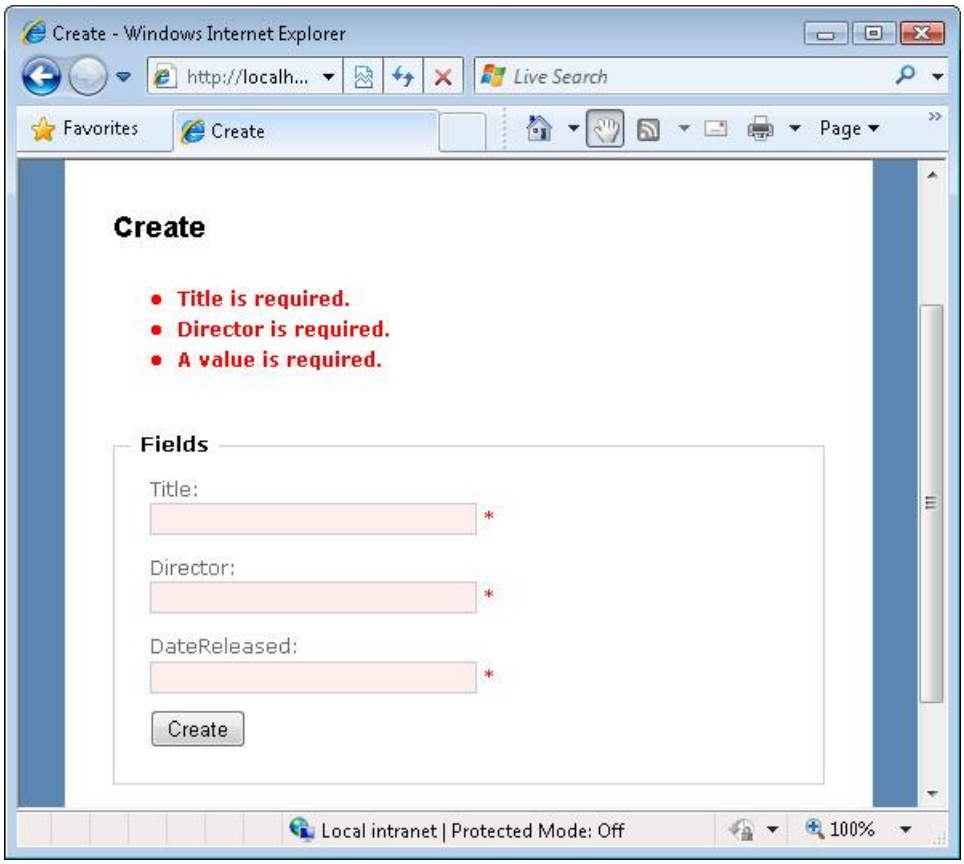


Figure 03: A form with missing values (Click to view full-size image)

Notice that the DateReleased value is validated automatically. Because the DateReleased property does not accept NULL values, the DefaultModelBinder generates a validation error for this property automatically when it does not have a value. If you want to modify the error message for the DateReleased property then you need to create a custom model binder.

Summary

In this tutorial, you learned how to use the IDataErrorInfo interface to generate validation error messages. First, we created a partial Movie class that extends the functionality of the partial Movie class generated by the Entity Framework. Next, we added validation logic to the Movie class OnTitleChanging() and OnDirectorChanging() partial methods. Finally, we implemented the IDataErrorInfo interface in order to expose these validation messages to the ASP.NET MVC framework.

PREVIOUS NEXT

Validating with a Service Layer (C#)

2/20/2020 • 6 minutes to read • Edit Online

by Stephen Walther

Learn how to move your validation logic out of your controller actions and into a separate service layer. In this tutorial, Stephen Walther explains how you can maintain a sharp separation of concerns by isolating your service layer from your controller layer.

The goal of this tutorial is to describe one method of performing validation in an ASP.NET MVC application. In this tutorial, you learn how to move your validation logic out of your controllers and into a separate service layer.

Separating Concerns

When you build an ASP.NET MVC application, you should not place your database logic inside your controller actions. Mixing your database and controller logic makes your application more difficult to maintain over time. The recommendation is that you place all of your database logic in a separate repository layer.

For example, Listing 1 contains a simple repository named the ProductRepository. The product repository contains all of the data access code for the application. The listing also includes the IProductRepository interface that the product repository implements.

Listing 1 -- Models\ProductRepository.cs

```
using System.Collections.Generic;
using System.Linq;

namespace MvcApplication1.Models
{
    public class ProductRepository : MvcApplication1.Models.IProductRepository
    {
        private ProductDBEntities _entities = new ProductDBEntities();

        public IEnumerable<Product> ListProducts()
        {
            return _entities.ProductSet.ToList();
        }

        public bool CreateProduct(Product productToCreate)
        {
            try
            {
                _entities.AddToProductSet(productToCreate);
                _entities.SaveChanges();
                return true;
            }
            catch
            {
                return false;
            }
        }

    }

    public interface IProductRepository
    {
        bool CreateProduct(Product productToCreate);
        IEnumerable<Product> ListProducts();
    }

}
```

The controller in Listing 2 uses the repository layer in both its Index() and Create() actions. Notice that this controller does not contain any database logic. Creating a repository layer enables you to maintain a clean separation of concerns. Controllers are responsible for application flow control logic and the repository is responsible for data access logic.

Listing 2 - Controllers\ProductController.cs

```
using System.Web.Mvc;
using MvcApplication1.Models;

namespace MvcApplication1.Controllers
{
    public class ProductController : Controller
    {
        private IProductRepository _repository;

        public ProductController():
            this(new ProductRepository()) {}

        public ProductController(IProductRepository repository)
        {
            _repository = repository;
        }

        public ActionResult Index()
        {
            return View(_repository.ListProducts());
        }

        //
        // GET: /Product/Create

        public ActionResult Create()
        {
            return View();
        }

        //
        // POST: /Product/Create

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create([Bind(Exclude="Id")] Product productToCreate)
        {
            _repository.CreateProduct(productToCreate);
            return RedirectToAction("Index");
        }

    }
}
```

Creating a Service Layer

So, application flow control logic belongs in a controller and data access logic belongs in a repository. In that case, where do you put your validation logic? One option is to place your validation logic in a *service layer*.

A service layer is an additional layer in an ASP.NET MVC application that mediates communication between a controller and repository layer. The service layer contains business logic. In particular, it contains validation logic.

For example, the product service layer in Listing 3 has a CreateProduct() method. The CreateProduct() method calls the ValidateProduct() method to validate a new product before passing the product to the product repository.

Listing 3 - Models\ProductService.cs

```
using System.Collections.Generic;
using System.Web.Mvc;

namespace MvcApplication1.Models
{
    public class ProductService : IProductService
    {

        private ModelStateDictionary _modelState;
        private IProductRepository _repository;

        public ProductService(ModelStateDictionary modelState, IProductRepository repository)
        {
            _modelState = modelState;
            _repository = repository;
        }

        protected bool ValidateProduct(Product productToValidate)
        {
            if (productToValidate.Name.Trim().Length == 0)
                _modelState.AddModelError("Name", "Name is required.");
            if (productToValidate.Description.Trim().Length == 0)
                _modelState.AddModelError("Description", "Description is required.");
            if (productToValidate.UnitsInStock < 0)
                _modelState.AddModelError("UnitsInStock", "Units in stock cannot be less than zero.");
            return _modelState.IsValid;
        }

        public IEnumerable<Product> ListProducts()
        {
            return _repository.ListProducts();
        }

        public bool CreateProduct(Product productToCreate)
        {
            // Validation logic
            if (!ValidateProduct(productToCreate))
                return false;

            // Database logic
            try
            {
                _repository.CreateProduct(productToCreate);
            }
            catch
            {
                return false;
            }
            return true;
        }

    }

    public interface IProductService
    {
        bool CreateProduct(Product productToCreate);
        IEnumerable<Product> ListProducts();
    }
}
```

The Product controller has been updated in Listing 4 to use the service layer instead of the repository layer. The controller layer talks to the service layer. The service layer talks to the repository layer. Each layer has a separate responsibility.

Listing 4 - Controllers\ProductController.cs

```
Listing 4 – Controllers\ProductController.cs
using System.Web.Mvc;
using MvcApplication1.Models;

namespace MvcApplication1.Controllers
{
    public class ProductController : Controller
    {
        private IProductService _service;

        public ProductController()
        {
            _service = new ProductService(this.ModelState, new ProductRepository());
        }

        public ProductController(IProductService service)
        {
            _service = service;
        }

        public ActionResult Index()
        {
            return View(_service.ListProducts());
        }

        //
        // GET: /Product/Create

        public ActionResult Create()
        {
            return View();
        }

        //
        // POST: /Product/Create

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create([Bind(Exclude = "Id")] Product productToCreate)
        {
            if (!_service.CreateProduct(productToCreate))
                return View();
            return RedirectToAction("Index");
        }

    }
}
```

Notice that the product service is created in the product controller constructor. When the product service is created, the model state dictionary is passed to the service. The product service uses model state to pass validation error messages back to the controller.

Decoupling the Service Layer

We have failed to isolate the controller and service layers in one respect. The controller and service layers communicate through model state. In other words, the service layer has a dependency on a particular feature of the ASP.NET MVC framework.

We want to isolate the service layer from our controller layer as much as possible. In theory, we should be able to use the service layer with any type of application and not only an ASP.NET MVC application. For example, in the future, we might want to build a WPF front-end for our application. We should find a way to remove the dependency on ASP.NET MVC model state from our service layer.

In Listing 5, the service layer has been updated so that it no longer uses model state. Instead, it uses any class that

implements the IValidationDictionary interface.

Listing 5 - Models\ProductService.cs (decoupled)

```
using System.Collections.Generic;

namespace MvcApplication1.Models
{
    public class ProductService : IProductService
    {

        private IValidationDictionary _validatonDictionary;
        private IProductRepository _repository;

        public ProductService(IValidationDictionary validationDictionary, IProductRepository repository)
        {
            _validatonDictionary = validationDictionary;
            _repository = repository;
        }

        protected bool ValidateProduct(Product productToValidate)
        {
            if (productToValidate.Name.Trim().Length == 0)
                _validatonDictionary.AddError("Name", "Name is required.");
            if (productToValidate.Description.Trim().Length == 0)
                _validatonDictionary.AddError("Description", "Description is required.");
            if (productToValidate.UnitsInStock < 0)
                _validatonDictionary.AddError("UnitsInStock", "Units in stock cannot be less than zero.");
            return _validatonDictionary.IsValid;
        }

        public IEnumerable<Product> ListProducts()
        {
            return _repository.ListProducts();
        }

        public bool CreateProduct(Product productToCreate)
        {
            // Validation logic
            if (!ValidateProduct(productToCreate))
                return false;

            // Database logic
            try
            {
                _repository.CreateProduct(productToCreate);
            }
            catch
            {
                return false;
            }
            return true;
        }

    }

    public interface IProductService
    {
        bool CreateProduct(Product productToCreate);
        IEnumerable<Product> ListProducts();
    }
}
```

The IValidationDictionary interface is defined in Listing 6. This simple interface has a single method and a single property.

Listing 6 - Models\IValidationDictionary.cs

```
namespace MvcApplication1.Models
{
    public interface IValidationDictionary
    {
        void AddError(string key, string errorMessage);
        bool IsValid { get; }
    }
}
```

The class in Listing 7, named the ModelStateWrapper class, implements the IValidationDictionary interface. You can instantiate the ModelStateWrapper class by passing a model state dictionary to the constructor.

Listing 7 - Models\ModelStateWrapper.cs

```
using System.Web.Mvc;

namespace MvcApplication1.Models
{
    public class ModelStateWrapper : IValidationDictionary
    {

        private ModelStateDictionary _modelState;

        public ModelStateWrapper(ModelStateDictionary modelState)
        {
            _modelState = modelState;
        }

        #region IValidationDictionary Members

        public void AddError(string key, string errorMessage)
        {
            _modelState.AddModelError(key, errorMessage);
        }

        public bool IsValid
        {
            get { return _modelState.IsValid; }
        }

        #endregion
    }
}
```

Finally, the updated controller in Listing 8 uses the ModelStateWrapper when creating the service layer in its constructor.

Listing 8 - Controllers\ProductController.cs

```
using System.Web.Mvc;
using MvcApplication1.Models;

namespace MvcApplication1.Controllers
{
    public class ProductController : Controller
    {
        private IProductService _service;

        public ProductController()
        {
            _service = new ProductService(new ModelStateWrapper(this.ModelState), new ProductRepository());
        }

        public ProductController(IProductService service)
        {
            _service = service;
        }

        public ActionResult Index()
        {
            return View(_service.ListProducts());
        }

        //
        // GET: /Product/Create

        public ActionResult Create()
        {
            return View();
        }

        //
        // POST: /Product/Create

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create([Bind(Exclude = "Id")] Product productToCreate)
        {
            if (!_service.CreateProduct(productToCreate))
                return View();
            return RedirectToAction("Index");
        }

    }
}
```

Using the IValidationDictionary interface and the ModelStateWrapper class enables us to completely isolate our service layer from our controller layer. The service layer is no longer dependent on model state. You can pass any class that implements the IValidationDictionary interface to the service layer. For example, a WPF application might implement the IValidationDictionary interface with a simple collection class.

Summary

The goal of this tutorial was to discuss one approach to performing validation in an ASP.NET MVC application. In this tutorial, you learned how to move all of your validation logic out of your controllers and into a separate service layer. You also learned how to isolate your service layer from your controller layer by creating a ModelStateWrapper class.

PREVIOUS NEXT

Validation with the Data Annotation Validators (C#)

4/10/2020 • 7 minutes to read • Edit Online

by Microsoft

> Take advantage of the Data Annotation Model Binder to perform validation within an ASP.NET MVC application. Learn how to use the different types of validator attributes and work with them in the Microsoft Entity Framework.

In this tutorial, you learn how to use the Data Annotation validators to perform validation in an ASP.NET MVC application. The advantage of using the Data Annotation validators is that they enable you to perform validation simply by adding one or more attributes – such as the Required or StringLength attribute – to a class property.

Before you can use the Data Annotation validators, you must download the Data Annotations Model Binder. You can download the Data Annotations Model Binder Sample from the CodePlex website by clicking here.

It is important to understand that the Data Annotations Model Binder is not an official part of the Microsoft ASP.NET MVC framework. Although the Data Annotations Model Binder was created by the Microsoft ASP.NET MVC team, Microsoft does not offer official product support for the Data Annotations Model Binder described and used in this tutorial.

Using the Data Annotation Model Binder

In order to use the Data Annotations Model Binder in an ASP.NET MVC application, you first need to add a reference to the Microsoft.Web.Mvc.DataAnnotations.dll assembly and the System.ComponentModel.DataAnnotations.dll assembly. Select the menu option **Project, Add Reference**. Next click the **Browse** tab and browse to the location where you downloaded (and unzipped) the Data Annotations Model Binder sample (see **Figure 1**).

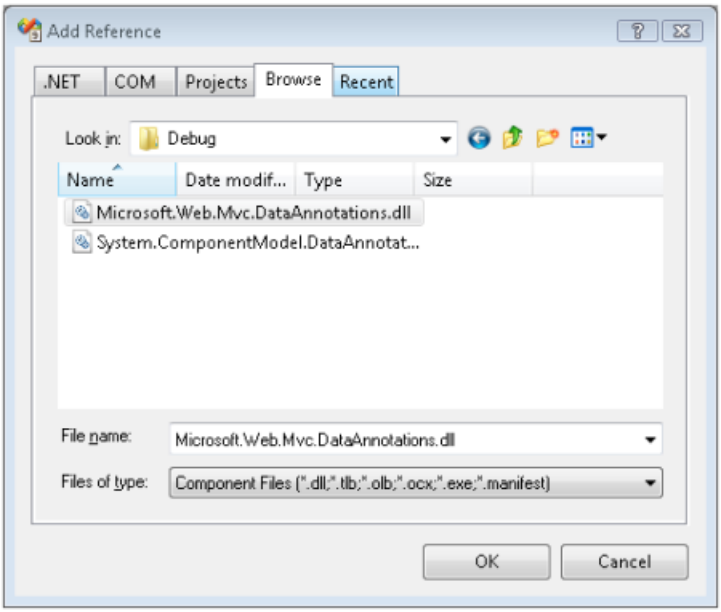


Figure 1: Adding a reference to the Data Annotations Model Binder (Click to view full-size image)

Select both the Microsoft.Web.Mvc.DataAnnotations.dll assembly and the System.ComponentModel.DataAnnotations.dll assembly and click the **OK** button.

You cannot use the System.ComponentModel.DataAnnotations.dll assembly included with .NET Framework

Service Pack 1 with the Data Annotations Model Binder. You must use the version of the System.ComponentModel.DataAnnotations.dll assembly included with the Data Annotations Model Binder Sample download.

Finally, you need to register the DataAnnotations Model Binder in the Global.asax file. Add the following line of code to the Application_Start() event handler so that the Application_Start() method looks like this:

```
protected void Application_Start()
{
    RegisterRoutes(RouteTable.Routes);
    ModelBinders.Binders.DefaultBinder = new Microsoft.Web.Mvc.DataAnnotations.DataAnnotationsModelBinder();
}
```

This line of code registers the ataAnnotationsModelBinder as the default model binder for the entire ASP.NET MVC application.

Using the Data Annotation Validator Attributes

When you use the Data Annotations Model Binder, you use validator attributes to perform validation. The System.ComponentModel.DataAnnotations namespace includes the following validator attributes:

- Range – Enables you to validate whether the value of a property falls between a specified range of values.
- RegularExpression – Enables you to validate whether the value of a property matches a specified regular expression pattern.
- Required – Enables you to mark a property as required.
- StringLength – Enables you to specify a maximum length for a string property.
- Validation – The base class for all validator attributes.

> **NOTE**
> If your validation needs are not satisfied by any of the standard validators then you always have the option of creating a custom validator attribute by inheriting a new validator attribute from the base Validation attribute.

The Product class in **Listing 1** illustrates how to use these validator attributes. The Name, Description, and UnitPrice properties are marked as required. The Name property must have a string length that is less than 10 characters. Finally, the UnitPrice property must match a regular expression pattern that represents a currency amount.

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace MvcApplication1.Models
{

    public class Product
    {
        public int Id { get; set; }

        [Required]
        [StringLength(10)]
        public string Name { get; set; }

        [Required]
        public string Description { get; set; }

        [DisplayName("Price")]
        [RegularExpression(@"^\$?\d+(\.(\d{2}))?$")]
        public decimal UnitPrice { get; set; }
    }
}
```

Listing 1: Models\Product.cs

The Product class illustrates how to use one additional attribute: the DisplayName attribute. The DisplayName attribute enables you to modify the name of the property when the property is displayed in an error message. Instead of displaying the error message "The UnitPrice field is required" you can display the error message "The Price field is required".

> **NOTE**
> If you want to completely customize the error message displayed by a validator then you can assign a custom error message to the validator's ErrorMessage property like this: `<Required(ErrorMessage:="This field needs a value!")>`

You can use the Product class in **Listing 1** with the Create() controller action in **Listing 2**. This controller action redisplays the Create view when model state contains any errors.

```
using System.Web.Mvc;
using MvcApplication1.Models;

namespace MvcApplication1.Controllers
{
    public class ProductController : Controller
    {
         //
        // GET: /Product/Create

        public ActionResult Create()
        {
            return View();
        }

        //
        // POST: /Product/Create

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create([Bind(Exclude="Id")]Product productToCreate)
        {
            if (!ModelState.IsValid)
                return View();

            // TODO: Add insert logic here
            return RedirectToAction("Index");
        }

    }
}
```

Listing 2: Controllers\ProductController.vb

Finally, you can create the view in **Listing 3** by right-clicking the Create() action and selecting the menu option **Add View**. Create a strongly-typed view with the Product class as the model class. Select **Create** from the view content dropdown list (see **Figure 2**).

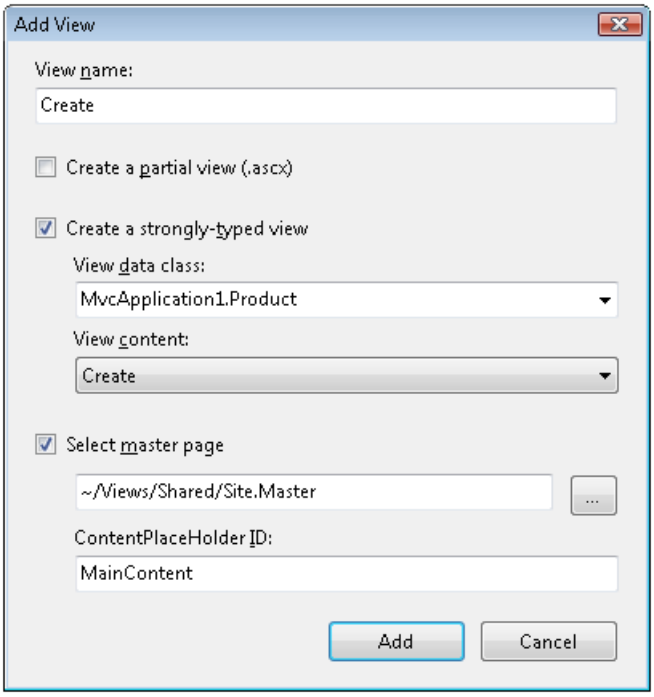


Figure 2: Adding the Create View

```
<%@ Page Title="" Language="C#" MasterPageFile="Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcApplication1.Models.Product>" %>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Create</h2>

    <%= Html.ValidationSummary("Create was unsuccessful. Please correct the errors and try again.") %>

    <% using (Html.BeginForm()) {%>

        <fieldset>
            <legend>Fields</legend>
            <p>
                <label for="Name">Name:</label>
                <%= Html.TextBox("Name") %>
                <%= Html.ValidationMessage("Name", "*") %>
            </p>
            <p>
                <label for="Description">Description:</label>
                <%= Html.TextBox("Description") %>
                <%= Html.ValidationMessage("Description", "*") %>
            </p>
            <p>
                <label for="UnitPrice">Price:</label>
                <%= Html.TextBox("UnitPrice") %>
                <%= Html.ValidationMessage("UnitPrice", "*") %>
            </p>
            <p>
                <input type="submit" value="Create" />
            </p>
        </fieldset>

    <% } %>

    <div>
        <%=Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

Listing 3: Views\Product\Create.aspx

> **NOTE**
> Remove the Id field from the Create form generated by the **Add View** menu option. Because the Id field corresponds to an Identity column, you don't want to allow users to enter a value for this field.

If you submit the form for creating a Product and you do not enter values for the required fields, then the validation error messages in **Figure 3** are displayed.

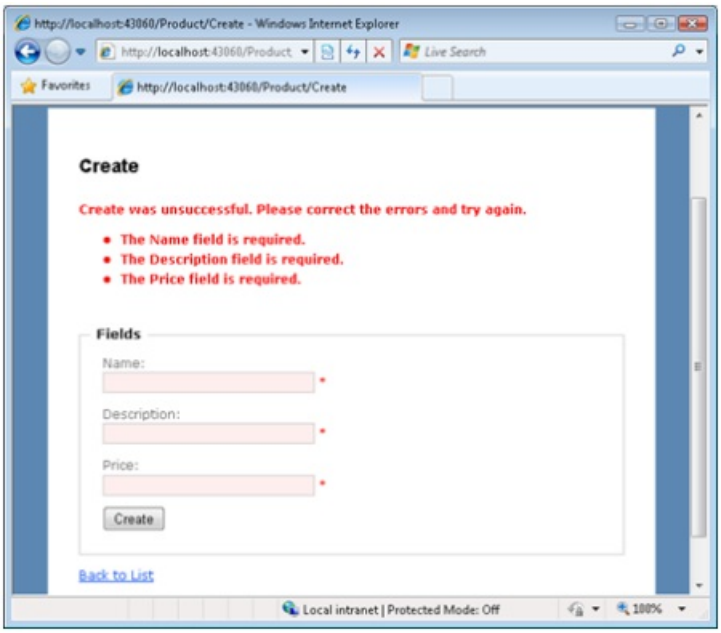


Figure 3: Missing required fields

If you enter an invalid currency amount, then the error message in **Figure 4** is displayed.

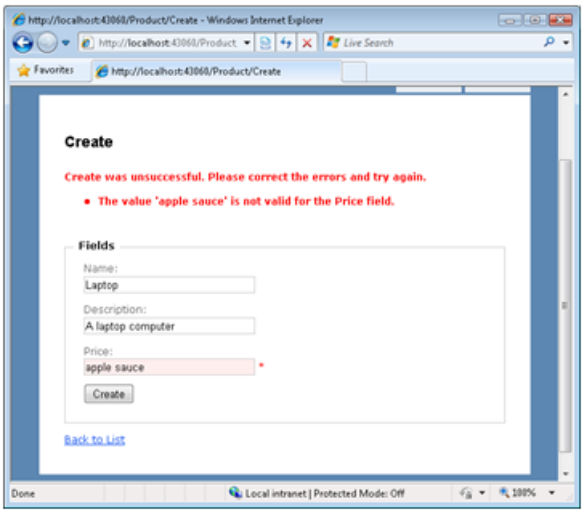


Figure 4: Invalid currency amount

Using Data Annotation Validators with the Entity Framework

If you are using the Microsoft Entity Framework to generate your data model classes then you cannot apply the validator attributes directly to your classes. Because the Entity Framework Designer generates the model classes, any changes you make to the model classes will be overwritten the next time you make any changes in the Designer.

If you want to use the validators with the classes generated by the Entity Framework then you need to create meta data classes. You apply the validators to the meta data class instead of applying the validators to the actual class.

For example, imagine that you have created a Movie class using the Entity Framework (see **Figure 5**). Imagine, furthermore, that you want to make the Movie Title and Director properties required properties. In that case, you can create the partial class and meta data class in **Listing 4**.

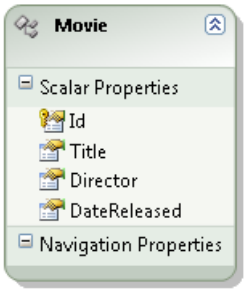


Figure 5: Movie class generated by Entity Framework

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace MvcApplication1.Models
{
    [MetadataType(typeof(MovieMetaData))]
    public partial class Movie
    {
    }

    public class MovieMetaData
    {
        [Required]
        public object Title { get; set; }

        [Required]
        [StringLength(5)]
        public object Director { get; set; }

        [DisplayName("Date Released")]
        [Required]
        public object DateReleased { get; set; }
    }

}
```

Listing 4: Models\Movie.cs

The file in **Listing 4** contains two classes named Movie and MovieMetaData. The Movie class is a partial class. It corresponds to the partial class generated by the Entity Framework that is contained in the DataModel.Designer.vb file.

Currently, the .NET framework does not support partial properties. Therefore, there is no way to apply the validator attributes to the properties of the Movie class defined in the DataModel.Designer.vb file by applying the validator attributes to the properties of the Movie class defined in the file in **Listing 4**.

Notice that the Movie partial class is decorated with a MetadataType attribute that points at the MovieMetaData class. The MovieMetaData class contains proxy properties for the properties of the Movie class.

The validator attributes are applied to the properties of the MovieMetaData class. The Title, Director, and DateReleased properties are all marked as required properties. The Director property must be assigned a string that contains less than 5 characters. Finally, the DisplayName attribute is applied to the DateReleased property to display an error message like "The Date Released field is required." instead of the error "The DateReleased field is required."

NOTE

Notice that the proxy properties in the MovieMetaData class do not need to represent the same types as the corresponding properties in the Movie class. For example, the Director property is a string property in the Movie class and an object property in the MovieMetaData class.

The page in **Figure 6** illustrates the error messages returned when you enter invalid values for the Movie properties.

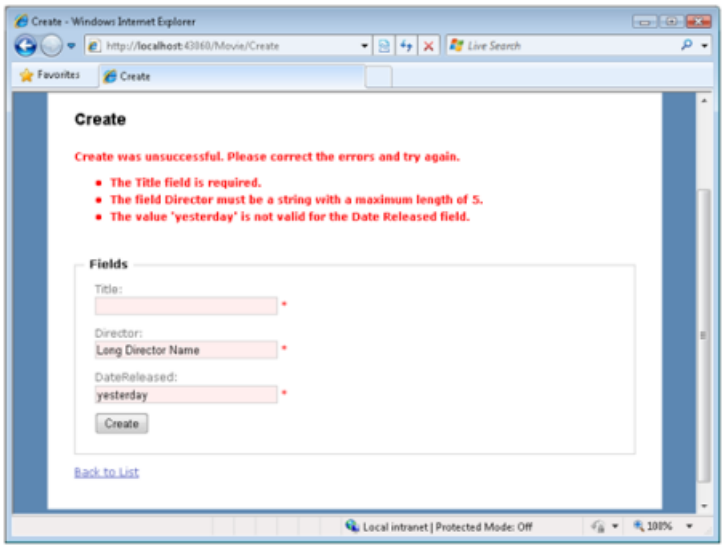


Figure 6: Using validators with the Entity Framework (Click to view full-size image)

Summary

In this tutorial, you learned how to take advantage of the Data Annotation Model Binder to perform validation within an ASP.NET MVC application. You learned how to use the different types of validator attributes such as the Required and StringLength attributes. You also learned how to use these attributes when working with the Microsoft Entity Framework.

PREVIOUS NEXT

Creating Model Classes with the Entity Framework (VB)

4/10/2020 • 12 minutes to read • Edit Online

by Microsoft

> In this tutorial, you learn how to use ASP.NET MVC with the Microsoft Entity Framework. You learn how to use the Entity Wizard to create an ADO.NET Entity Data Model. Over the course of this tutorial, we build a web application that illustrates how to select, insert, update, and delete database data by using the Entity Framework.

The goal of this tutorial is to explain how you can create data access classes using the Microsoft Entity Framework when building an ASP.NET MVC application. This tutorial assumes no previous knowledge of the Microsoft Entity Framework. By the end of this tutorial, you'll understand how to use the Entity Framework to select, insert, update, and delete database records.

The Microsoft Entity Framework is an Object Relational Mapping (O/RM) tool that enables you to generate a data access layer from a database automatically. The Entity Framework enables you to avoid the tedious work of building your data access classes by hand.

> **NOTE**
> There is no essential connection between ASP.NET MVC and the Microsoft Entity Framework. There are several alternatives to the Entity Framework that you can use with ASP.NET MVC. For example, you can build your MVC Model classes using other O/RM tools such as Microsoft LINQ to SQL, NHibernate, or SubSonic.

In order to illustrate how you can use the Microsoft Entity Framework with ASP.NET MVC, we'll build a simple sample application. We'll create a Movie Database application that enables you to display and edit movie database records.

This tutorial assumes that you have Visual Studio 2008 or Visual Web Developer 2008 with Service Pack 1. You need Service Pack 1 in order to use the Entity Framework. You can download Visual Studio 2008 Service Pack 1 or Visual Web Developer with Service Pack 1 from the following address:

> https://www.asp.net/downloads/

Creating the Movie Sample Database

The Movie Database application uses a database table named Movies that contains the following columns:

COLUMN NAME	DATA TYPE	ALLOW NULLS?	IS PRIMARY KEY?
Id	int	False	True
Title	nvarchar(100)	False	False
Director	nvarchar(100)	False	False

You can add this table to an ASP.NET MVC project by following these steps:

1. Right-click the App_Data folder in the Solution Explorer window and select the menu option **Add, New Item.**
2. From the **Add New Item** dialog box, select **SQL Server Database**, give the database the name MoviesDB.mdf, and click the **Add** button.
3. Double-click the MoviesDB.mdf file to open the Server Explorer/Database Explorer window.
4. Expand the MoviesDB.mdf database connection, right-click the Tables folder, and select the menu option **Add New Table**.
5. In the Table Designer, add the Id, Title, and Director columns.
6. Click the **Save** button (it has the icon of the floppy) to save the new table with the name Movies.

After you create the Movies database table, you should add some sample data to the table. Right-click the Movies table and select the menu option **Show Table Data**. You can enter fake movie data into the grid that appears.

Creating the ADO.NET Entity Data Model

In order to use the Entity Framework, you need to create an Entity Data Model. You can take advantage of the Visual Studio *Entity Data Model Wizard* to generate an Entity Data Model from a database automatically.

Follow these steps:

1. Right-click the Models folder in the Solution Explorer window and select the menu option **Add, New Item**.
2. In the **Add New Item** dialog, select the Data category (see Figure 1).
3. Select the **ADO.NET Entity Data Model** template, give the Entity Data Model the name MoviesDBModel.edmx, and click the **Add** button. Clicking the **Add** button launches the Data Model Wizard.
4. In the **Choose Model Contents** step, choose the **Generate from a database** option and click the **Next** button (see Figure 2).
5. In the **Choose Your Data Connection** step, select the MoviesDB.mdf database connection, enter the entities connection settings name MoviesDBEntities, and click the **Next** button (see Figure 3).
6. In the **Choose Your Database Objects** step, select the Movie database table and click the **Finish** button (see Figure 4).

After you complete these steps, the ADO.NET Entity Data Model Designer (Entity Designer) opens.

Figure 1 – Creating a new Entity Data Model

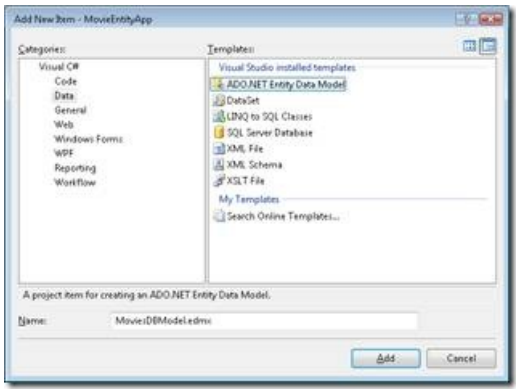


Figure 2 – Choose Model Contents Step

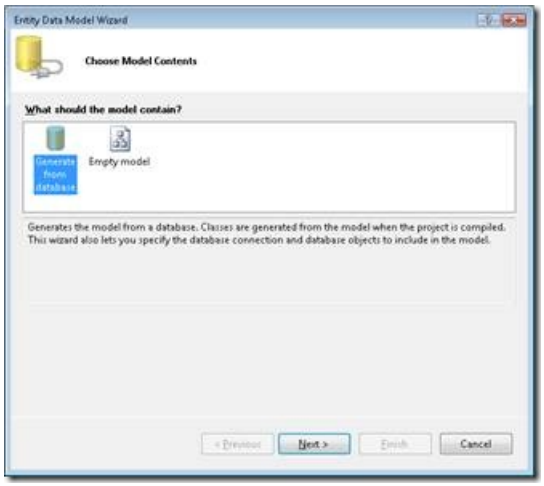
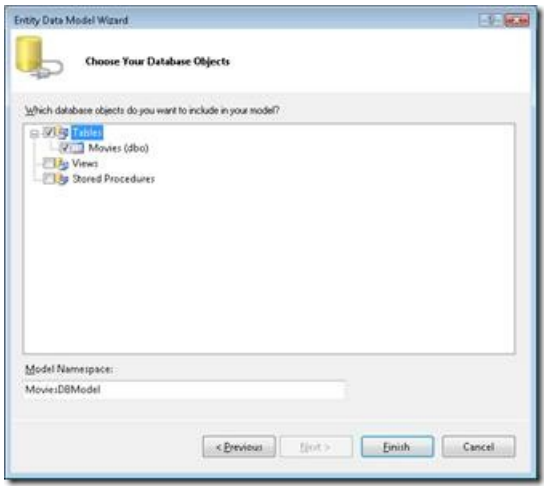


Figure 3 – Choose Your Data Connection

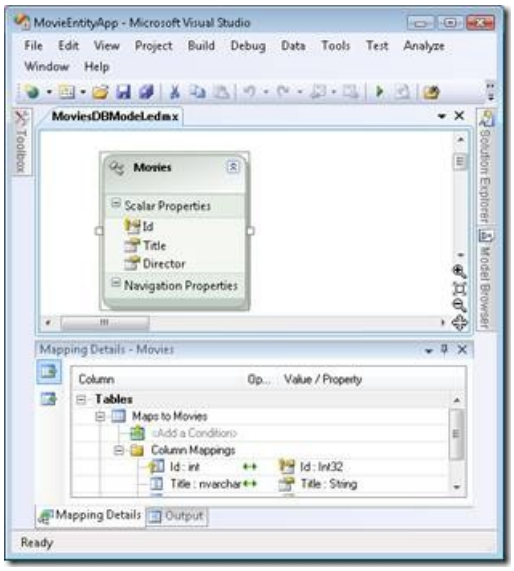
Figure 4 – Choose Your Database Objects



Modifying the ADO.NET Entity Data Model

After you create an Entity Data Model, you can modify the model by taking advantage of the Entity Designer (see Figure 5). You can open the Entity Designer at any time by double-clicking the MoviesDBModel.edmx file contained in the Models folder within the Solution Explorer window.

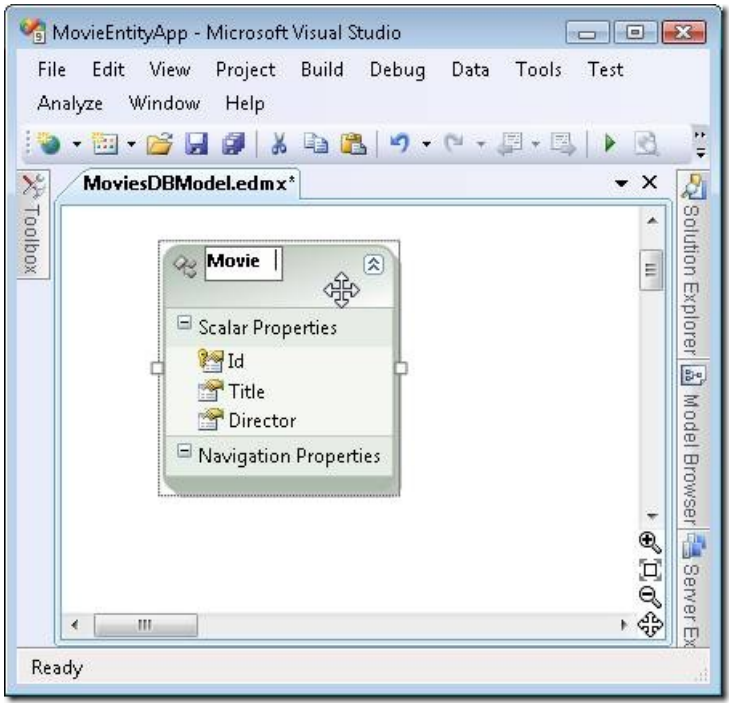
Figure 5 – The ADO.NET Entity Data Model Designer



For example, you can use the Entity Designer to change the names of the classes that the Entity Model Data Wizard generates. The Wizard created a new data access class named Movies. In other words, the Wizard gave the class the very same name as the database table. Because we will use this class to represent a particular Movie instance, we should rename the class from Movies to Movie.

If you want to rename an entity class, you can double-click on the class name in the Entity Designer and enter a new name (see Figure 6). Alternatively, you can change the name of an entity in the Properties window after selecting an entity in the Entity Designer.

Figure 6 – Changing an entity name



Remember to save your Entity Data Model after making a modification by clicking the Save button (the icon of the floppy disk). Behind the scenes, the Entity Designer generates a set of Visual Basic .NET classes. You can view these classes by opening the MoviesDBModel.Designer.vb file from the Solution Explorer window.

Don't modify the code in the Designer.vb file since your changes will be overwritten the next time you use the Entity Designer. If you want to extend the functionality of the entity classes defined in the Designer.vb file then you can create *partial classes* in separate files.

Selecting Database Records with the Entity Framework

Let's start building our Movie Database application by creating a page that displays a list of movie records. The

Home controller in Listing 1 exposes an action named Index(). The Index() action returns all of the movie records from the Movie database table by taking advantage of the Entity Framework.

Listing 1 – Controllers\HomeController.vb

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Private _db As MoviesDBEntities

    Public Sub New()
        _db = New MoviesDBEntities()
    End Sub

    Public Function Index()
        ViewData.Model = _db.MovieSet.ToList()
        Return View()
    End Function

End Class
```

Notice that the controller in Listing 1 includes a constructor. The constructor initializes a class-level field named _db. The _db field represents the database entities generated by the Microsoft Entity Framework. The _db field is an instance of the MoviesDBEntities class that was generated by the Entity Designer.

The _db field is used within the Index() action to retrieve the records from the Movies database table. The expression _db.MovieSet represents all of the records from the Movies database table. The ToList() method is used to convert the set of movies into a generic collection of Movie objects: List(Of Movie).

The movie records are retrieved with the help of LINQ to Entities. The Index() action in Listing 1 uses LINQ *method syntax* to retrieve the set of database records. If you prefer, you can use LINQ *query syntax* instead. The following two statements do the very same thing:

```
ViewData.Model = _db.MovieSet.ToList()
ViewData.Model = (from m in _db.MovieSet select m).ToList()
```

Use whichever LINQ syntax – method syntax or query syntax – that you find most intuitive. There is no difference in performance between the two approaches – the only difference is style.

The view in Listing 2 is used to display the movie records.

Listing 2 – Views\Home\Index.aspx

```
<%@ Page Language="VB"
  Inherits="System.Web.Mvc.ViewPage(Of List(Of MvcApplication1.Movie))" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Index</title>
</head>
<body>
    <div>

<%  For Each m In Model%>

    Title: <%= m.Title %>
    <br />
    Director: <%= m.Director %>
    <br />
    <%=Html.ActionLink("Edit", "Edit", New With {.id = m.Id})%>
    <%=Html.ActionLink("Delete", "Delete", New With {.id = m.Id})%>

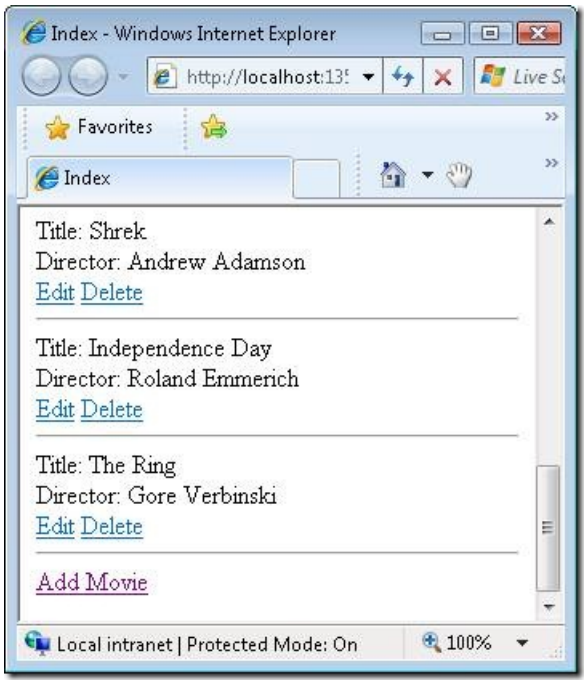
        <hr />
<% Next%>

    <%= Html.ActionLink("Add Movie", "Add") %>

    </div>
</body>
</html>
```

The view in Listing 2 contains a **For Each** loop that iterates through each movie record and displays the values of the movie record's Title and Director properties. Notice that an Edit and Delete link is displayed next to each record. Furthermore, an Add Movie link appears at the bottom of the view (see Figure 7).

Figure 7 – The Index view



The Index view is a *typed view*. The Index view has a <%@ Page %> directive that includes an Inherits attribute. The Inherits attribute casts the ViewData.Model property to a strongly typed generic List collection of Movie objects – a List(Of Movie).

Inserting Database Records with the Entity Framework

You can use the Entity Framework to make it easy to insert new records into a database table. Listing 3 contains two new actions added to the Home controller class that you can use to insert new records into the Movie database table.

Listing 3 – Controllers\HomeController.vb (Add methods)

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Public Function Add()
        Return View()
    End Function

    <AcceptVerbs(HttpVerbs.Post)> _
    Public Function Add(ByVal form As FormCollection)
        Dim movieToAdd As New Movie()

        ' Deserialize (Include white list!)
        TryUpdateModel(movieToAdd, New String() {"Title", "Director"}, form.ToValueProvider())

        ' Validate
        If String.IsNullOrEmpty(movieToAdd.Title) Then
            ModelState.AddModelError("Title", "Title is required!")
        End If
        If String.IsNullOrEmpty(movieToAdd.Director) Then
            ModelState.AddModelError("Director", "Director is required!")
        End If

        ' If valid, save movie to database
        If (ModelState.IsValid) Then
            _db.AddToMovieSet(movieToAdd)
            _db.SaveChanges()
            Return RedirectToAction("Index")
        End If

        ' Otherwise, reshow form
        Return View(movieToAdd)
    End Function

End Class
```

The first Add() action simply returns a view. The view contains a form for adding a new movie database record (see Figure 8). When you submit the form, the second Add() action is invoked.

Notice that the second Add() action is decorated with the AcceptVerbs attribute. This action can be invoked only when performing an HTTP POST operation. In other words, this action can only be invoked when posting an HTML form.

The second Add() action creates a new instance of the Entity Framework Movie class with the help of the ASP.NET MVC TryUpdateModel() method. The TryUpdateModel() method takes the fields in the FormCollection passed to the Add() method and assigns the values of these HTML form fields to the Movie class.

When using the Entity Framework, you must supply a "white list" of properties when using the TryUpdateModel or UpdateModel methods to update the properties of an entity class.

Next, the Add() action performs some simple form validation. The action verifies that both the Title and Director properties have values. If there is a validation error, then a validation error message is added to ModelState.

If there are no validation errors then a new movie record is added to the Movies database table with the help of the Entity Framework. The new record is added to the database with the following two lines of code:

```
_db.AddToMovieSet(movieToAdd)
_db.SaveChanges()
```

The first line of code adds the new Movie entity to the set of movies being tracked by the Entity Framework. The second line of code saves whatever changes have been made to the Movies being tracked back to the underlying database.

Figure 8 – The Add view

Updating Database Records with the Entity Framework

You can follow almost the same approach to edit a database record with the Entity Framework as the approach that we just followed to insert a new database record. Listing 4 contains two new controller actions named Edit(). The first Edit() action returns an HTML form for editing a movie record. The second Edit() action attempts to update the database.

Listing 4 – Controllers\HomeController.vb (Edit methods)

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Public Function Edit(ByVal id As Integer)
        ' Get movie to update
        Dim movieToUpdate As Movie = _db.MovieSet.First(Function(m) m.Id = id)
        ViewData.Model = movieToUpdate
        Return View()
    End Function

    <AcceptVerbs(HttpVerbs.Post)> _
    Public Function Edit(ByVal form As FormCollection)

        ' Get movie to update
        Dim id As Integer = Integer.Parse(form("id"))
        Dim movieToUpdate As Movie = _db.MovieSet.First(Function(m) m.Id = id)

        ' Deserialize (Include white list!)
        TryUpdateModel(movieToUpdate, New String() {"Title", "Director"}, form.ToValueProvider)

        ' Validate
        If String.IsNullOrEmpty(movieToUpdate.Title) Then
            ModelState.AddModelError("Title", "Title is required!")
        End If
        If String.IsNullOrEmpty(movieToUpdate.Director) Then
            ModelState.AddModelError("Director", "Director is required!")
        End If

        ' If valid, save movie to database
        If (ModelState.IsValid) Then
            _db.SaveChanges()
            Return RedirectToAction("Index")
        End If

        ' Otherwise, reshow form
        Return View(movieToUpdate)
    End Function

End Class
```

The second Edit() action starts by retrieving the Movie record from the database that matches the Id of the movie being edited. The following LINQ to Entities statement grabs the first database record that matches a particular Id:

```
Dim movieToUpdate As Movie = _db.MovieSet.First(Function(m) m.Id = id)
```

Next, the TryUpdateModel() method is used to assign the values of the HTML form fields to the properties of the movie entity. Notice that a white list is supplied to specify the exact properties to update.

Next, some simple validation is performed to verify that both the Movie Title and Director properties have values. If either property is missing a value, then a validation error message is added to ModelState and ModelState.IsValid returns the value false.

Finally, if there are no validation errors, then the underlying Movies database table is updated with any changes by calling the SaveChanges() method.

When editing database records, you need to pass the Id of the record being edited to the controller action that performs the database update. Otherwise, the controller action will not know which record to update in the underlying database. The Edit view, contained in Listing 5, includes a hidden form field that represents the Id of the database record being edited.

Listing 5 – Views\Home\Edit.aspx

```
<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage(Of MvcApplication1.Movie)" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Edit</title>
    <style type="text/css">

    .input-validation-error
    {
        background-color:Yellow;
    }

    </style>
</head>
<body>
    <div>

<h1>Edit Movie</h1>

<form method="post" action="/Home/Edit">

    <!-- Include Hidden Id -->
    <%= Html.Hidden("id") %>

    Title:
    <br />
    <%= Html.TextBox("title") %>

    <br /><br />
    Director:
    <br />
    <%= Html.TextBox("director") %>

    <br /><br />
    <input type="submit" value="Edit Movie" />
</form>

    </div>
</body>
</html>
```

Deleting Database Records with the Entity Framework

The final database operation, which we need to tackle in this tutorial, is deleting database records. You can use the controller action in Listing 6 to delete a particular database record.

Listing 6 -- \Controllers\HomeController.vb (Delete action)

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Public Function Delete(ByVal id As Integer)
        ' Get movie to delete
        Dim movieToDelete As Movie = _db.MovieSet.First(Function(m) m.Id = id)

        ' Delete
        _db.DeleteObject(movieToDelete)
        _db.SaveChanges()

        ' Show Index view
        Return RedirectToAction("Index")
    End Function

End Class
```

The Delete() action first retrieves the Movie entity that matches the Id passed to the action. Next, the movie is deleted from the database by calling the DeleteObject() method followed by the SaveChanges() method. Finally, the user is redirected back to the Index view.

Summary

The purpose of this tutorial was to demonstrate how you can build database-driven web applications by taking advantage of ASP.NET MVC and the Microsoft Entity Framework. You learned how to build an application that enables you to select, insert, update, and delete database records.

First, we discussed how you can use the Entity Data Model Wizard to generate an Entity Data Model from within Visual Studio. Next, you learn how to use LINQ to Entities to retrieve a set of database records from a database table. Finally, we used the Entity Framework to insert, update, and delete database records.

PREVIOUS NEXT

Creating Model Classes with LINQ to SQL (VB)

4/10/2020 • 11 minutes to read • Edit Online

by Microsoft

Download PDF

The goal of this tutorial is to explain one method of creating model classes for an ASP.NET MVC application. In this tutorial, you learn how to build model classes and perform database access by taking advantage of Microsoft LINQ to SQL.

The goal of this tutorial is to explain one method of creating model classes for an ASP.NET MVC application. In this tutorial, you learn how to build model classes and perform database access by taking advantage of Microsoft LINQ to SQL.

In this tutorial, we build a basic Movie database application. We start by creating the Movie database application in the fastest and easiest way possible. We perform all of our data access directly from our controller actions.

Next, you learn how to use the Repository pattern. Using the Repository pattern requires a little more work. However, the advantage of adopting this pattern is that it enables you to build applications that are adaptable to change and can be easily tested.

What is a Model Class?

An MVC model contains all of the application logic that is not contained in an MVC view or MVC controller. In particular, an MVC model contains all of your application business and data access logic.

You can use a variety of different technologies to implement your data access logic. For example, you can build your data access classes using the Microsoft Entity Framework, NHibernate, Subsonic, or ADO.NET classes.

In this tutorial, I use LINQ to SQL to query and update the database. LINQ to SQL provides you with a very easy method of interacting with a Microsoft SQL Server database. However, it is important to understand that the ASP.NET MVC framework is not tied to LINQ to SQL in any way. ASP.NET MVC is compatible with any data access technology.

Create a Movie Database

In this tutorial -- in order to illustrate how you can build model classes -- we build a simple Movie database application. The first step is to create a new database. Right-click the App_Data folder in the Solution Explorer window and select the menu option **Add, New Item**. Select the SQL Server Database template, give it the name MoviesDB.mdf, and click the **Add** button (see Figure 1).

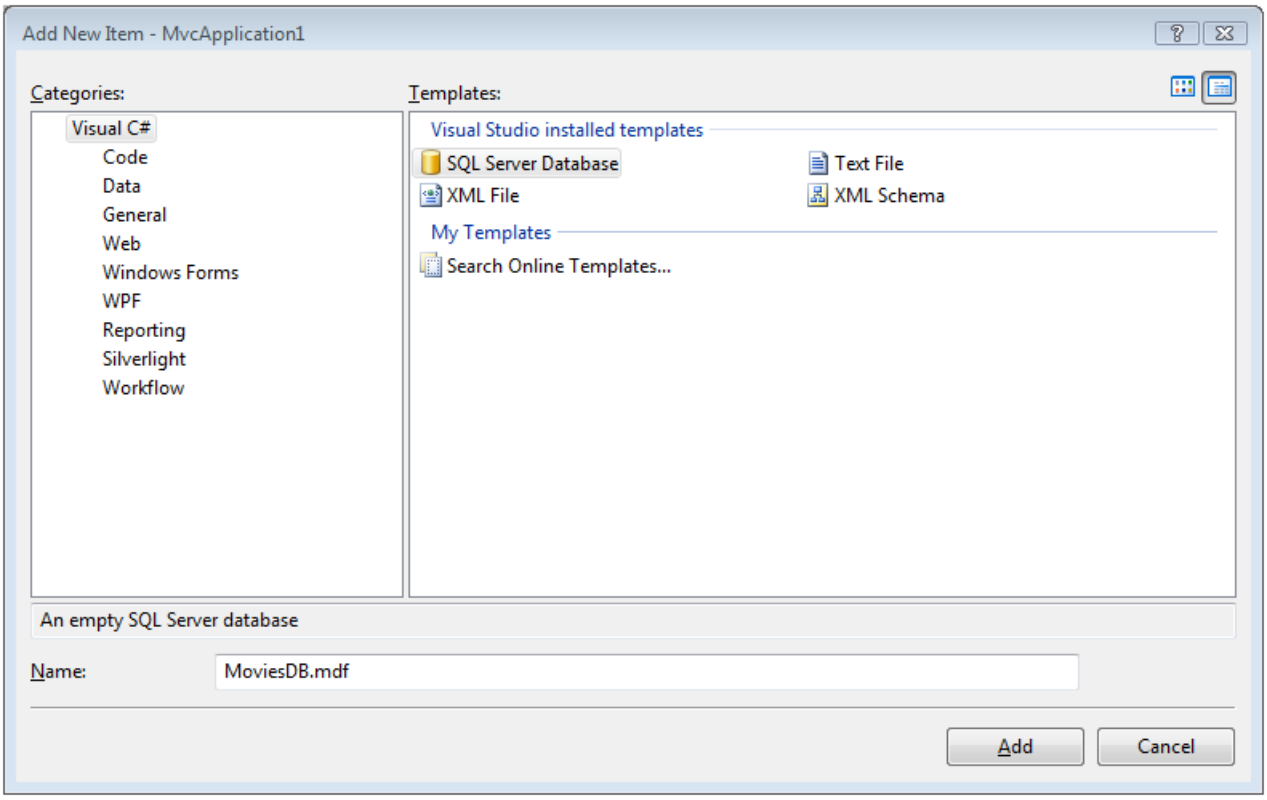


Figure 01: Adding a new SQL Server Database (Click to view full-size image)

After you create the new database, you can open the database by double-clicking the MoviesDB.mdf file in the App_Data folder. Double-clicking the MoviesDB.mdf file opens the Server Explorer window (see Figure 2).

	THE SERVER EXPLORER WINDOW IS CALLED THE DATABASE EXPLORER WINDOW WHEN USING VISUAL WEB DEVELOPER.

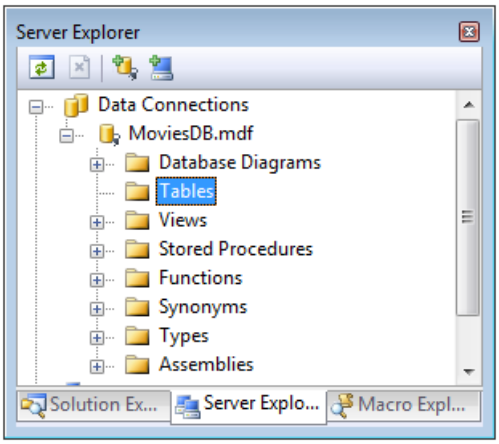


Figure 02: Using the Server Explorer window (Click to view full-size image)

We need to add one table to our database that represents our movies. Right-click the Tables folder and select the menu option **Add New Table**. Selecting this menu option opens the Table Designer (see Figure 3).

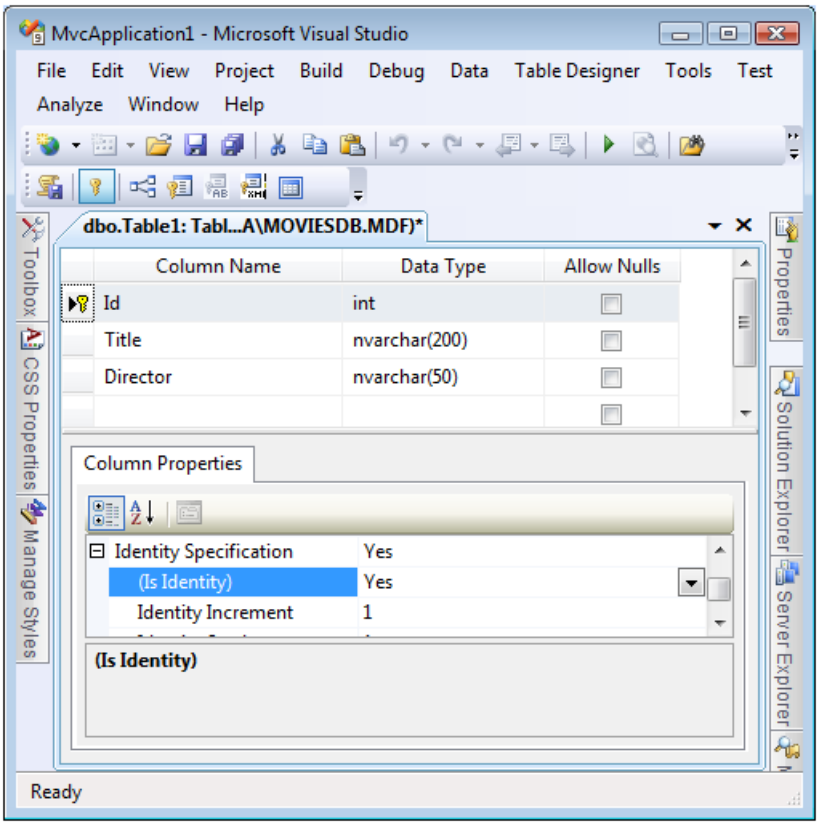


Figure 03: The Table Designer (Click to view full-size image)

We need to add the following columns to our database table:

COLUMN NAME	DATA TYPE	ALLOW NULLS
Id	Int	False
Title	Nvarchar(200)	False
Director	Nvarchar(50)	False

You need to do two special things to the Id column. First, you need to mark the Id column as a primary key column by selecting the column in the Table Designer and clicking the icon of a key. LINQ to SQL requires you to specify your primary key columns when performing inserts or updates against the database.

Next, you need to mark the Id column as an Identity column by assigning the value Yes to the **Is Identity** property (see Figure 3). An Identity column is a column that is assigned a new number automatically whenever you add a new row of data to a table.

After you make these changes, save the table with the name tblMovie. You can save the table by clicking the Save button.

Create LINQ to SQL Classes

Our MVC model will contain LINQ to SQL classes that represent the tblMovie database table. The easiest way to create these LINQ to SQL classes is to right-click the Models folder, select **Add, New Item**, select the LINQ to SQL Classes template, give the classes the name Movie.dbml, and click the **Add** button (see Figure 4).

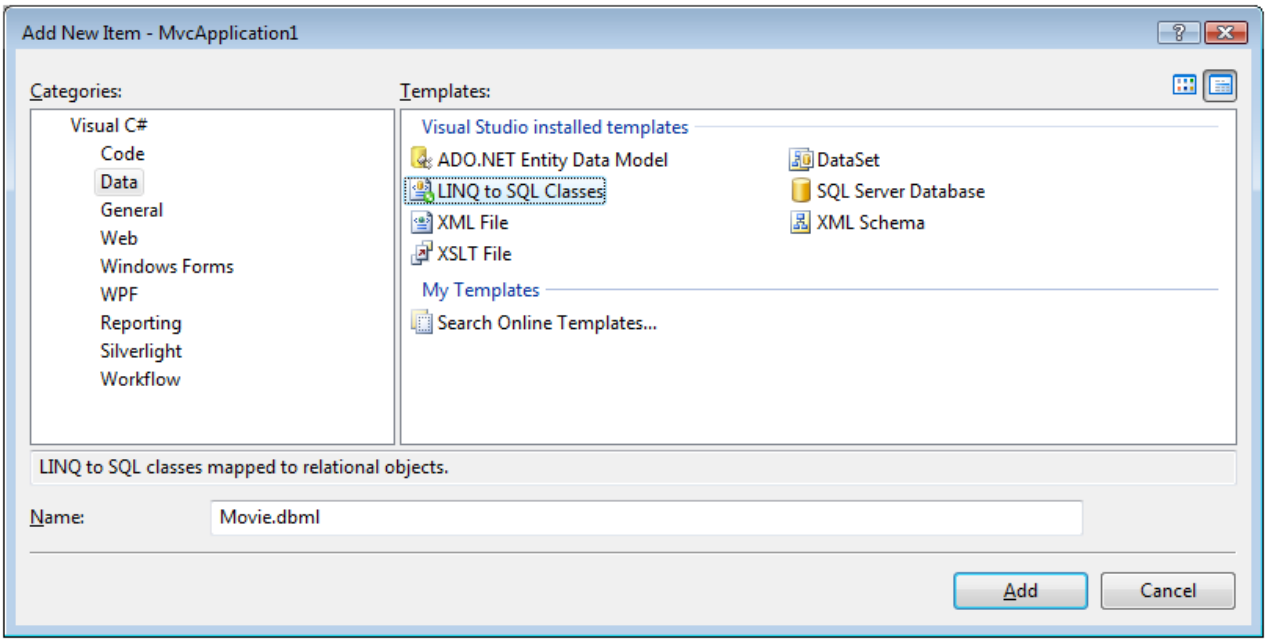


Figure 04: Creating LINQ to SQL classes (Click to view full-size image)

Immediately after you create the Movie LINQ to SQL Classes, the Object Relational Designer appears. You can drag database tables from the Server Explorer window onto the Object Relational Designer to create LINQ to SQL Classes that represent particular database tables. We need to add the tblMovie database table onto the Object Relational Designer (see Figure 4).

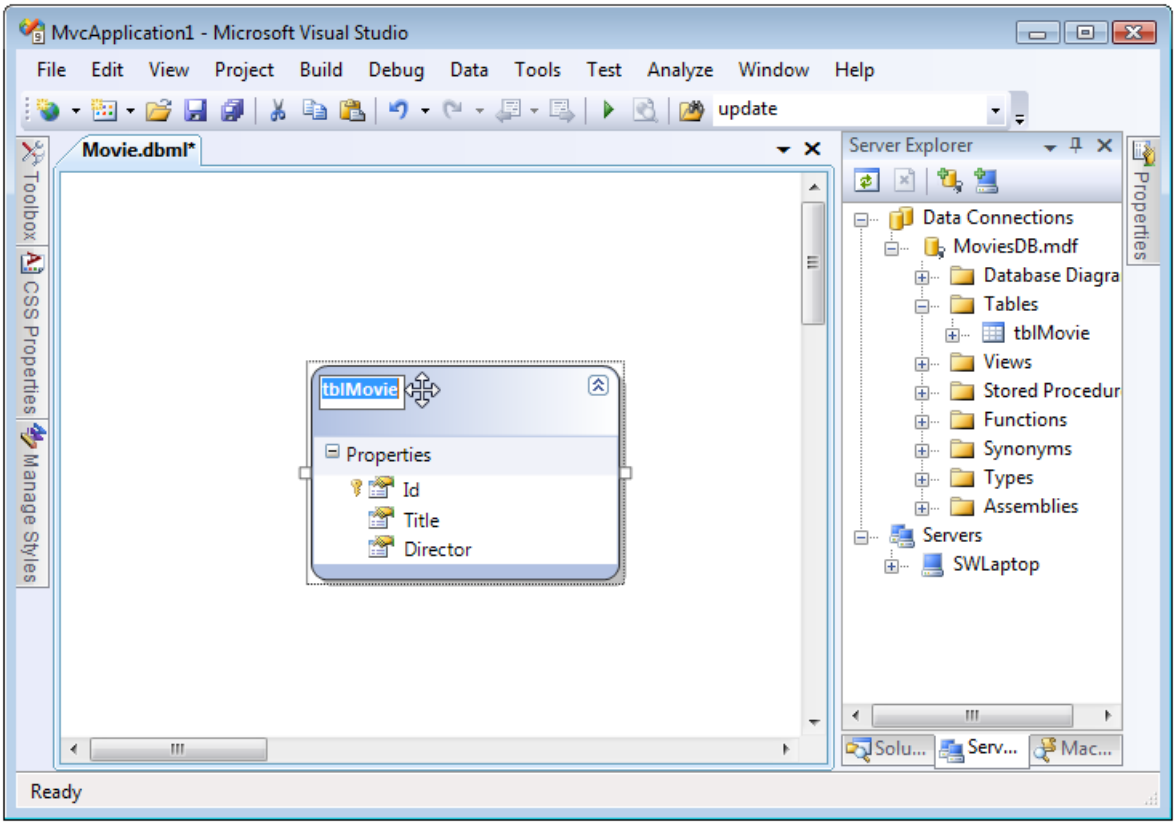


Figure 05: Using the Object Relational Designer (Click to view full-size image)

By default, the Object Relational Designer creates a class with the very same name as the database table that you drag onto the Designer. However, we don't want to call our class tblMovie. Therefore, click the name of the class in the Designer and change the name of the class to Movie.

Finally, remember to click the **Save** button (the picture of the floppy) to save the LINQ to SQL Classes. Otherwise, the LINQ to SQL Classes won't be generated by the Object Relational Designer.

Using LINQ to SQL in a Controller Action

Now that we have our LINQ to SQL classes, we can use these classes to retrieve data from the database. In this section, you learn how to use LINQ to SQL classes directly within a controller action. We'll display the list of movies from the tblMovies database table in an MVC view.

First, we need to modify the HomeController class. This class can be found in the Controllers folder of your application. Modify the class so it looks like the class in Listing 1.

Listing 1 – `Controllers\HomeController.vb`

```
<HandleError()> _
Public Class HomeController

     Inherits System.Web.Mvc.Controller

     Function Index()
          Dim dataContext As New MovieDataContext()
          Dim movies = From m In dataContext.Movies _
               Select m
          return View(movies)
     End Function
End Class
```

The Index() action in Listing 1 uses a LINQ to SQL DataContext class (the MovieDataContext) to represent the MoviesDB database. The MoveDataContext class was generated by the Visual Studio Object Relational Designer.

A LINQ query is performed against the DataContext to retrieve all of the movies from the tblMovies database table. The list of movies is assigned to a local variable named movies. Finally, the list of movies is passed to the view through view data.

In order to show the movies, we next need to modify the Index view. You can find the Index view in the Views\Home\ folder. Update the Index view so that it looks like the view in Listing 2.

Listing 2 – `Views\Home\Index.aspx`

```
<%@ Page Language="VB" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="false"
CodeBehind="Index.aspx.vb" Inherits="MvcApplication1.Index" %>
<%@ Import Namespace="MvcApplication1" %>
<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent" runat="server">

<ul>
<%  For Each m As Movie In ViewData.Model%>
    <li><%= m.Title %></li>
<% Next%>
</ul>
</asp:Content>
```

Notice that the modified Index view includes an <%@ import namespace %> directive at the top of the view. This directive imports the MvcApplication1 namespace. We need this namespace in order to work with the model classes – in particular, the Movie class -- in the view.

The view in Listing 2 contains a For Each loop that iterates through all of the items represented by the ViewData.Model property. The value of the Title property is displayed for each movie.

Notice that the value of the ViewData.Model property is cast to an IEnumerable. This is necessary in order to loop through the contents of ViewData.Model. Another option here is to create a strongly-typed view. When you create a strongly-typed view, you cast the ViewData.Model property to a particular type in a view's code-behind class.

If you run the application after modifying the HomeController class and the Index view then you will get a blank

page. You'll get a blank page because there are no movie records in the tblMovies database table.

In order to add records to the tblMovies database table, right-click the tblMovies database table in the Server Explorer window (Database Explorer window in Visual Web Developer) and select the menu option **Show Table Data**. You can insert movie records by using the grid that appears (see Figure 5).

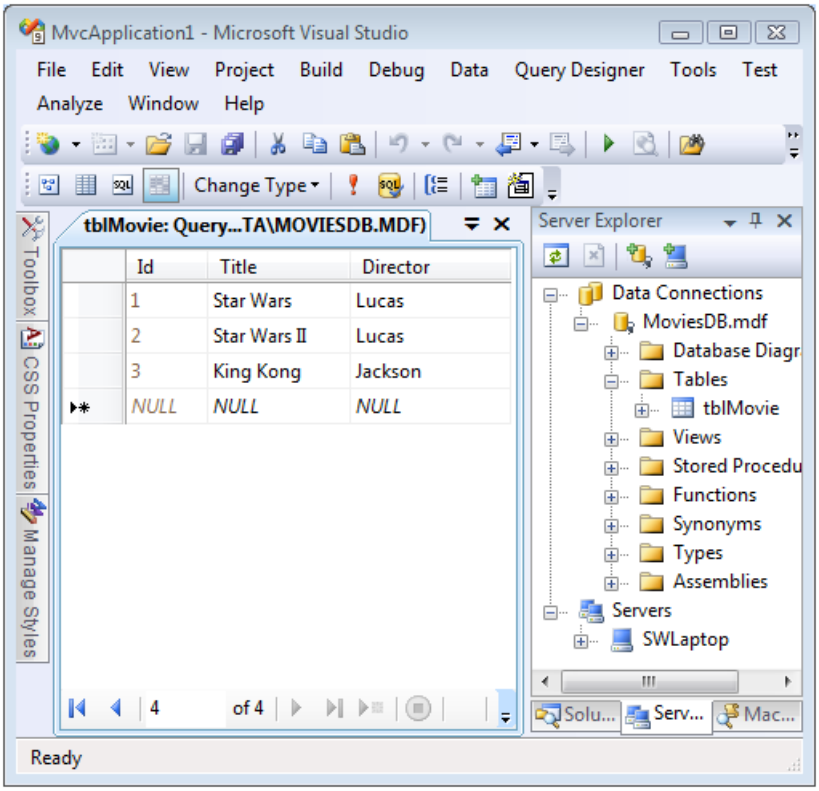


Figure 06: Inserting movies(Click to view full-size image)

After you add some database records to the tblMovies table, and you run the application, you'll see the page in Figure 7. All of the movie database records are displayed in a bulleted list.

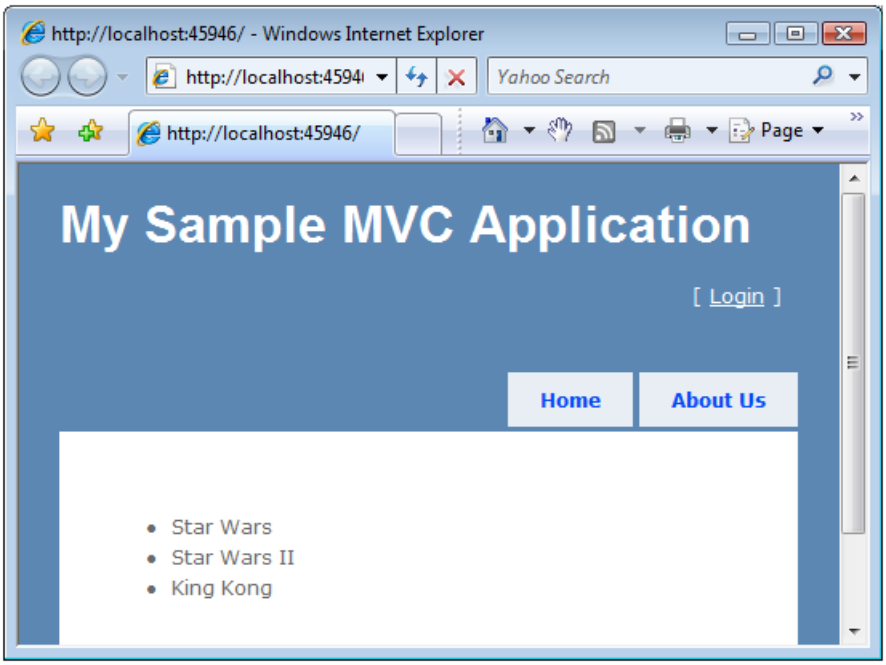


Figure 07: Displaying movies with the Index view(Click to view full-size image)

Using the Repository Pattern

In the previous section, we used LINQ to SQL classes directly within a controller action. We used the

MovieDataContext class directly from the Index() controller action. There is nothing wrong with doing this in the case of a simple application. However, working directly with LINQ to SQL in a controller class creates problems when you need to build a more complex application.

Using LINQ to SQL within a controller class makes it difficult to switch data access technologies in the future. For example, you might decide to switch from using Microsoft LINQ to SQL to using the Microsoft Entity Framework as your data access technology. In that case, you would need to rewrite every controller that accesses the database within your application.

Using LINQ to SQL within a controller class also makes it difficult to build unit tests for your application. Normally, you do not want to interact with a database when performing unit tests. You want to use your unit tests to test your application logic and not your database server.

In order to build an MVC application that is more adaptable to future change and that can be more easily tested, you should consider using the Repository pattern. When you use the Repository pattern, you create a separate repository class that contains all of your database access logic.

When you create the repository class, you create an interface that represents all of the methods used by the repository class. Within your controllers, you write your code against the interface instead of the repository. That way, you can implement the repository using different data access technologies in the future.

The interface in Listing 3 is named IMovieRepository and it represents a single method named ListAll().

Listing 3 – `Models\IMovieRepository.vb`

```
Public Interface IMovieRepository
     Function ListAll() As IList(Of Movie)
End Interface
```

The repository class in Listing 4 implements the IMovieRepository interface. Notice that it contains a method named ListAll() that corresponds to the method required by the IMovieRepository interface.

Listing 4 – `Models\MovieRepository.vb`

```
Public Class MovieRepository Implements IMovieRepository
         Private _dataContext As MovieDataContext

         Public Sub New()
              _dataContext = New MovieDataContext()
         End Sub

         Public Function ListAll() As IList(Of Movie) Implements IMovieRepository.ListAll
              Dim movies = From m In _dataContext.Movies _
                   Select m
              Return movies.ToList()
         End Function
End Class
```

Finally, the MoviesController class in Listing 5 uses the Repository pattern. It no longer uses LINQ to SQL classes directly.

Listing 5 – `Controllers\MoviesController.vb`

```
using System.Web.Mvc;
using MvcApplication1.Models;

namespace MvcApplication1.Controllers
{
    Public Class MoviesController
        Inherits System.Web.Mvc.Controller

             Private _repository As IMovieRepository

             Sub New()
                  Me.New(New MovieRepository())
             End Sub

             Sub New(ByVal repository As IMovieRepository)
                  _repository = repository
             End Sub

             Function Index()
                  Return View(_repository.ListAll())
             End Function
    End Class
}
```

Notice that the MoviesController class in Listing 5 has two constructors. The first constructor, the parameterless constructor, is called when your application is running. This constructor creates an instance of the MovieRepository class and passes it to the second constructor.

The second constructor has a single parameter: an IMovieRepository parameter. This constructor simply assigns the value of the parameter to a class-level field named _repository.

The MoviesController class is taking advantage of a software design pattern called the Dependency Injection pattern. In particular, it is using something called Constructor Dependency Injection. You can read more about this pattern by reading the following article by Martin Fowler:

http://martinfowler.com/articles/injection.html

Notice that all of the code in the MoviesController class (with the exception of the first constructor) interacts with the IMovieRepository interface instead of the actual MovieRepository class. The code interacts with an abstract interface instead of a concrete implementation of the interface.

If you want to modify the data access technology used by the application then you can simply implement the IMovieRepository interface with a class that uses the alternative database access technology. For example, you could create an EntityFrameworkMovieRepository class or a SubSonicMovieRepository class. Because the controller class is programmed against the interface, you can pass a new implementation of IMovieRepository to the controller class and the class would continue to work.

Furthermore, if you want to test the MoviesController class, then you can pass a fake movie repository class to the MoviesController. You can implement the IMovieRepository class with a class that does not actually access the database but contains all of the required methods of the IMovieRepository interface. That way, you can unit test the MoviesController class without actually accessing a real database.

Summary

The goal of this tutorial was to demonstrate how you can create MVC model classes by taking advantage of Microsoft LINQ to SQL. We examined two strategies for displaying database data in an ASP.NET MVC application. First, we created LINQ to SQL classes and used the classes directly within a controller action. Using LINQ to SQL classes within a controller enables you to quickly and easily display database data in an MVC application.

Next, we explored a slightly more difficult, but definitely more virtuous, path for displaying database data. We took

advantage of the Repository pattern and placed all of our database access logic in a separate repository class. In our controller, we wrote all of our code against an interface instead of a concrete class. The advantage of the Repository pattern is that it enables us to easily change database access technologies in the future and it enables us to easily test our controller classes.

PREVIOUS NEXT

Displaying a Table of Database Data (VB)

4/10/2020 • 7 minutes to read • Edit Online

by Microsoft

Download PDF

In this tutorial, I demonstrate two methods of displaying a set of database records. I show two methods of formatting a set of database records in an HTML table. First, I show how you can format the database records directly within a view. Next, I demonstrate how you can take advantage of partials when formatting database records.

The goal of this tutorial is to explain how you can display an HTML table of database data in an ASP.NET MVC application. First, you learn how to use the scaffolding tools included in Visual Studio to generate a view that displays a set of records automatically. Next, you learn how to use a partial as a template when formatting database records.

Create the Model Classes

We are going to display the set of records from the Movies database table. The Movies database table contains the following columns:

COLUMN NAME	DATA TYPE	ALLOW NULLS
Id	Int	False
Title	Nvarchar(200)	False
Director	NVarchar(50)	False
DateReleased	DateTime	False

In order to represent the Movies table in our ASP.NET MVC application, we need to create a model class. In this tutorial, we use the Microsoft Entity Framework to create our model classes.

> **NOTE**
> In this tutorial, we use the Microsoft Entity Framework. However, it is important to understand that you can use a variety of different technologies to interact with a database from an ASP.NET MVC application including LINQ to SQL, NHibernate, or ADO.NET.

Follow these steps to launch the Entity Data Model Wizard:

1. Right-click the Models folder in the Solution Explorer window and the select the menu option **Add, New Item**.
2. Select the **Data** category and select the **ADO.NET Entity Data Model** template.
3. Give your data model the name *MoviesDBModel.edmx* and click the **Add** button.

After you click the Add button, the Entity Data Model Wizard appears (see Figure 1). Follow these steps to complete the wizard:

1. In the **Choose Model Contents** step, select the **Generate from database** option.
2. In the **Choose Your Data Connection** step, use the *MoviesDB.mdf* data connection and the name *MoviesDBEntities* for the connection settings. Click the **Next** button.
3. In the **Choose Your Database Objects** step, expand the Tables node, select the Movies table. Enter the namespace *Models* and click the **Finish** button.

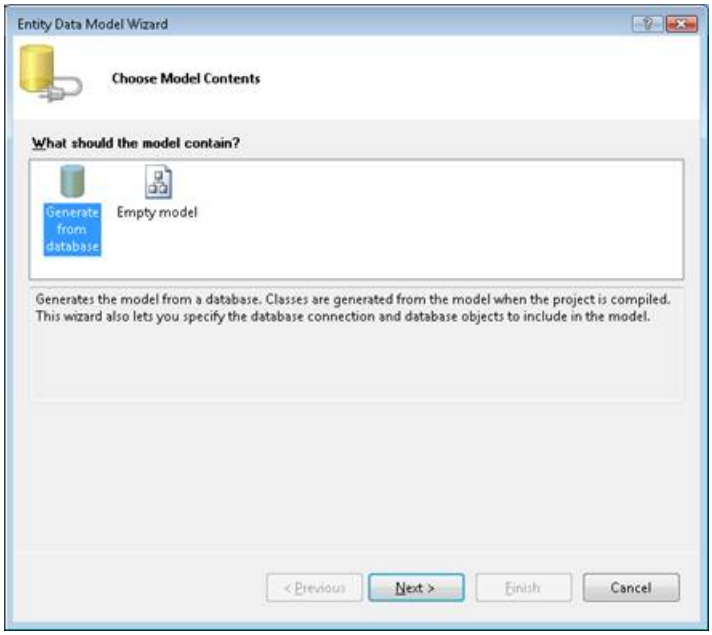


Figure 01: Creating LINQ to SQL classes (Click to view full-size image)

After you complete the Entity Data Model Wizard, the Entity Data Model Designer opens. The Designer should display the Movies entity (see Figure 2).

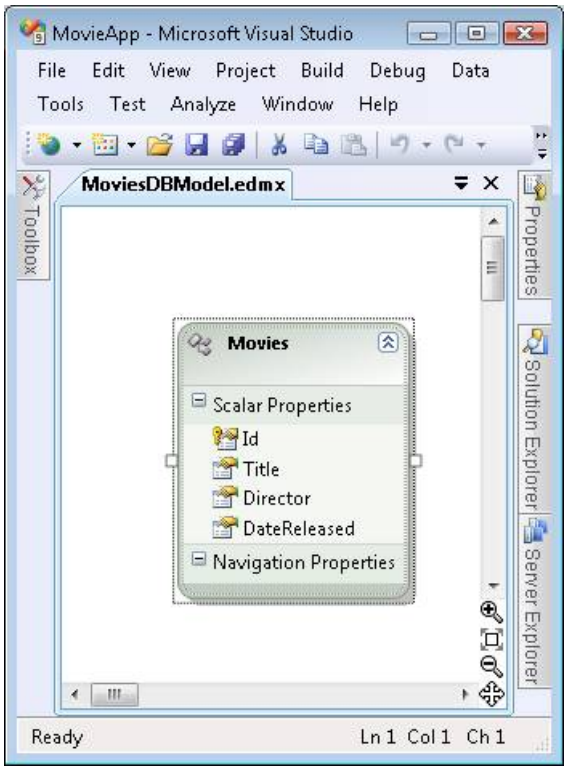


Figure 02: The Entity Data Model Designer (Click to view full-size image)

We need to make one change before we continue. The Entity Data Wizard generates a model class named *Movies* that represents the Movies database table. Because we'll use the Movies class to represent a particular movie, we need to modify the name of the class to be *Movie* instead of *Movies* (singular rather than plural).

Double-click the name of the class on the designer surface and change the name of the class from Movies to

Movie. After making this change, click the **Save** button (the icon of the floppy disk) to generate the Movie class.

Create the Movies Controller

Now that we have a way to represent our database records, we can create a controller that returns the collection of movies. Within the Visual Studio Solution Explorer window, right-click the Controllers folder and select the menu option **Add, Controller** (see Figure 3).

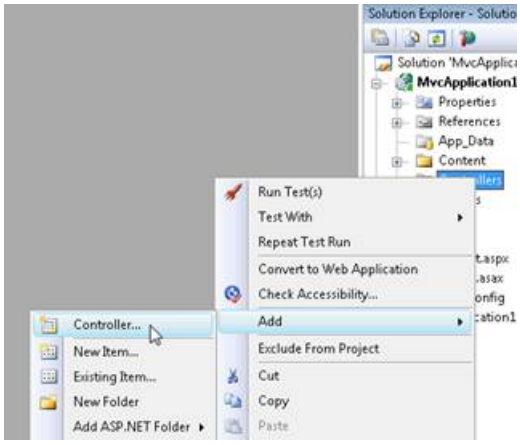


Figure 03: The Add Controller Menu (Click to view full-size image)

When the **Add Controller** dialog appears, enter the controller name MovieController (see Figure 4). Click the **Add** button to add the new controller.

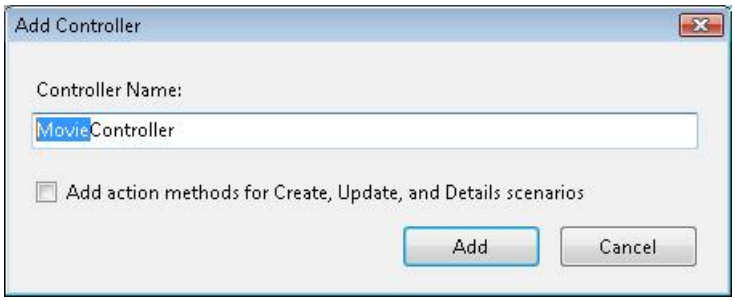


Figure 04: The Add Controller dialog(Click to view full-size image)

We need to modify the Index() action exposed by the Movie controller so that it returns the set of database records. Modify the controller so that it looks like the controller in Listing 1.

Listing 1 – Controllers\MovieController.vb

```
Public Class MovieController

    Inherits System.Web.Mvc.Controller

    '

    ' GET: /Movie/

    Function Index() As ActionResult

        Dim entities As New MoviesDBEntities()

        Return View(entities.MovieSet.ToList())

    End Function

End Class
```

In Listing 1, the MoviesDBEntities class is used to represent the MoviesDB database. The expression *entities.MovieSet.ToList()* returns the set of all movies from the Movies database table.

Create the View

The easiest way to display a set of database records in an HTML table is to take advantage of the scaffolding provided by Visual Studio.

Build your application by selecting the menu option **Build, Build Solution**. You must build your application before opening the **Add View** dialog or your data classes won't appear in the dialog.

Right-click the Index() action and select the menu option **Add View** (see Figure 5).

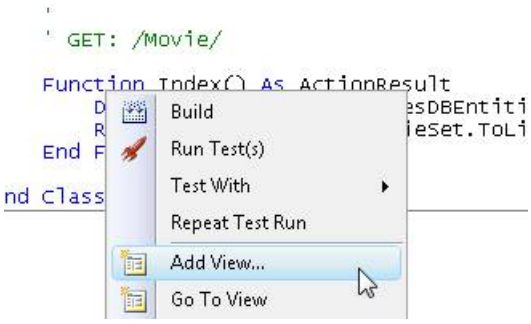


Figure 05: Adding a view (Click to view full-size image)

In the **Add View** dialog, check the checkbox labeled **Create a strongly-typed view**. Select the Movie class as the **view data class**. Select *List* as the **view content** (see Figure 6). Selecting these options will generate a strongly-typed view that displays a list of movies.

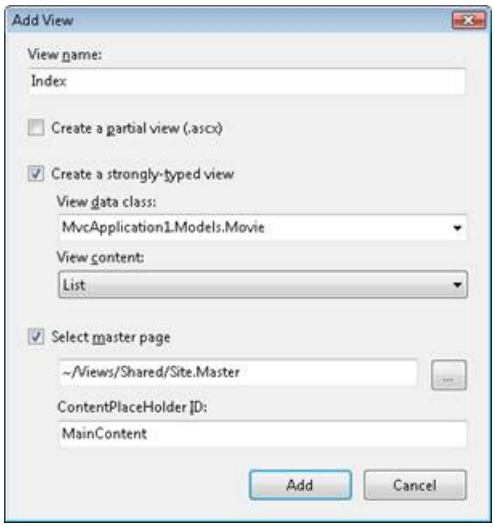


Figure 06: The Add View dialog(Click to view full-size image)

After you click the **Add** button, the view in Listing 2 is generated automatically. This view contains the code required to iterate through the collection of movies and display each of the properties of a movie.

Listing 2 – Views\Movie\Index.aspx

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of IEnumerable (Of MvcApplication1.Movie))" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

      Index

</asp:Content>
```

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Index</h2>

    <p>

        <%=Html.ActionLink("Create New", "Create")%>

    </p>

    <table>

        <tr>

            <th></th>

            <th>

                Id

            </th>

            <th>

                Title

            </th>

            <th>

                Director

            </th>

            <th>

                DateReleased

            </th>

        </tr>

    <% For Each item In Model%>

        <tr>

            <td>

                <%=Html.ActionLink("Edit", "Edit", New With {.id = item.Id})%> |

                <%=Html.ActionLink("Details", "Details", New With {.id = item.Id})%>

            </td>

            <td>

                <%= Html.Encode(item.Id) %>

            </td>

            <td>

                <%= Html.Encode(item.Title) %>

            </td>

            <td>

                <%= Html.Encode(item.Director) %>
```

```
            </td>

            <td>

                <%= Html.Encode(String.Format("{0:g}", item.DateReleased)) %>

            </td>

        </tr>

    <% Next%>

    </table>

</asp:Content>
```

You can run the application by selecting the menu option **Debug, Start Debugging** (or hitting the F5 key). Running the application launches Internet Explorer. If you navigate to the /Movie URL then you'll see the page in Figure 7.

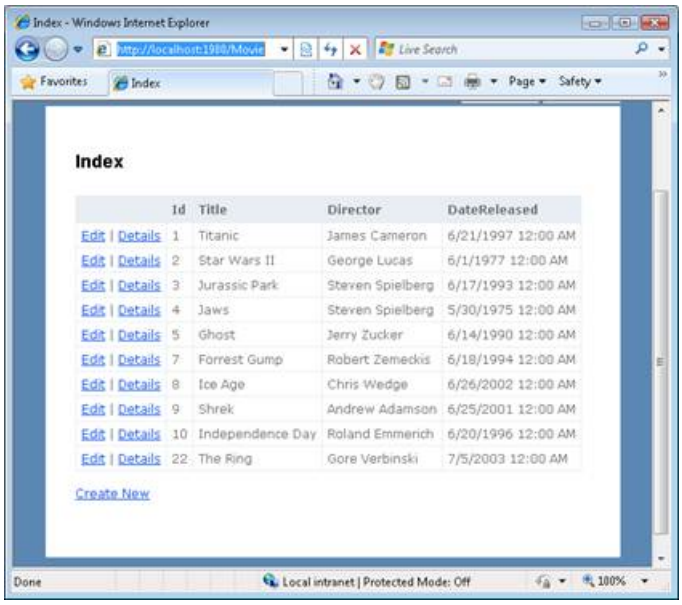


Figure 07: A table of movies(Click to view full-size image)

If you don't like anything about the appearance of the grid of database records in Figure 7 then you can simply modify the Index view. For example, you can change the *DateReleased* header to *Date Released* by modifying the Index view.

Create a Template with a Partial

When a view gets too complicated, it is a good idea to start breaking the view into partials. Using partials makes your views easier to understand and maintain. We'll create a partial that we can use as a template to format each of the movie database records.

Follow these steps to create the partial:

1. Right-click the Views\Movie folder and select the menu option **Add View**.
2. Check the checkbox labeled *Create a partial view (.ascx)*.
3. Name the partial *MovieTemplate*.
4. Check the checkbox labeled **Create a strongly-typed view**.
5. Select Movie as the *view data class*.
6. Select Empty as the *view content*.

7. Click the **Add** button to add the partial to your project.

After you complete these steps, modify the MovieTemplate partial to look like Listing 3.

Listing 3 – Views\Movie\MovieTemplate.ascx

```
<%@ Control Language="VB" Inherits="System.Web.Mvc.ViewUserControl(Of MvcApplication1.Movie)" %>

<tr>

    <td>

        <%= Html.Encode(Model.Id) %>

    </td>

    <td>

        <%= Html.Encode(Model.Title) %>

    </td>

    <td>

        <%= Html.Encode(Model.Director) %>

    </td>

    <td>

        <%= Html.Encode(String.Format("{0:g}", Model.DateReleased)) %>

    </td>

</tr>
```

The partial in Listing 3 contains a template for a single row of records.

The modified Index view in Listing 4 uses the MovieTemplate partial.

Listing 4 – Views\Movie\Index.aspx

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of IEnumerable (Of MvcApplication1.Movie))" %>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Index</h2>

    <table>

        <tr>

            <th>

                Id

            </th>

            <th>

                Title

            </th>

            <th>

                Director

            </th>

            <th>

                DateReleased

            </th>

        </tr>

    <% For Each item In Model%>

        <% Html.RenderPartial("MovieTemplate", item)%>

    <% Next%>

    </table>

</asp:Content>
```

The view in Listing 4 contains a For Each loop that iterates through all of the movies. For each movie, the MovieTemplate partial is used to format the movie. The MovieTemplate is rendered by calling the RenderPartial() helper method.

The modified Index view renders the very same HTML table of database records. However, the view has been greatly simplified.

The RenderPartial() method is different than most of the other helper methods because it does not return a string. Therefore, you must call the RenderPartial() method using <% Html.RenderPartial() %> instead of <%= Html.RenderPartial() %>.

Summary

The goal of this tutorial was to explain how you can display a set of database records in an HTML table. First, you learned how to return a set of database records from a controller action by taking advantage of the Microsoft Entity Framework. Next, you learned how to use Visual Studio scaffolding to generate a view that displays a

collection of items automatically. Finally, you learned how to simplify the view by taking advantage of a partial. You learned how to use a partial as a template so that you can format each database record.

PREVIOUS NEXT

Performing Simple Validation (VB)

2/20/2020 • 5 minutes to read • Edit Online

by Stephen Walther

> Learn how to perform validation in an ASP.NET MVC application. In this tutorial, Stephen Walther introduces you to model state and the validation HTML helpers.

The goal of this tutorial is to explain how you can perform validation within an ASP.NET MVC application. For example, you learn how to prevent someone from submitting a form that does not contain a value for a required field. You learn how to use model state and the validation HTML helpers.

Understanding Model State

You use model state - or more accurately, the model state dictionary - to represent validation errors. For example, the Create() action in Listing 1 validates the properties of a Product class before adding the Product class to a database.

I'm not recommending that you add your validation or database logic to a controller. A controller should contain only logic related to application flow control. We are taking a shortcut to keep things simple.

Listing 1 - Controllers\ProductController.vb

```
'
' POST: /Product/Create

<AcceptVerbs(HttpVerbs.Post)> _
Public Function Create(<Bind(Exclude:="Id")> ByVal productToCreate As Product) As ActionResult
    ' Validation logic
    If productToCreate.Name.Trim().Length = 0 Then
    ModelState.AddModelError("Name", "Name is required.")
    End If
    If productToCreate.Description.Trim().Length = 0 Then
    ModelState.AddModelError("Description", "Description is required.")
    End If
    If productToCreate.UnitsInStock
```

In Listing 1, the Name, Description, and UnitsInStock properties of the Product class are validated. If any of these properties fail a validation test then an error is added to the model state dictionary (represented by the ModelState property of the Controller class).

If there are any errors in model state then the ModelState.IsValid property returns false. In that case, the HTML form for creating a new product is redisplayed. Otherwise, if there are no validation errors, the new Product is added to the database.

Using the Validation Helpers

The ASP.NET MVC framework includes two validation helpers: the Html.ValidationMessage() helper and the Html.ValidationSummary() helper. You use these two helpers in a view to display validation error messages.

The Html.ValidationMessage() and Html.ValidationSummary() helpers are used in the Create and Edit views that are generated automatically by the ASP.NET MVC scaffolding. Follow these steps to generate the Create view:

1. Right-click the Create() action in the Product controller and select the menu option **Add View** (see Figure 1).
2. In the **Add View** dialog, check the checkbox labeled **Create a strongly-typed view** (see Figure 2).
3. From the **View data class** dropdown list, select the Product class.
4. From the **View content** dropdown list, select Create.
5. Click the **Add** button.

Make sure that you build your application before adding a view. Otherwise, the list of classes won't appear in the **View data class** dropdown list.

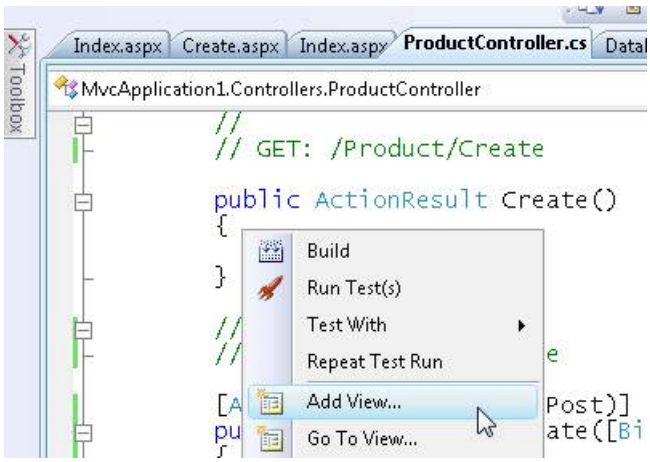


Figure 01: Adding a view(Click to view full-size image)

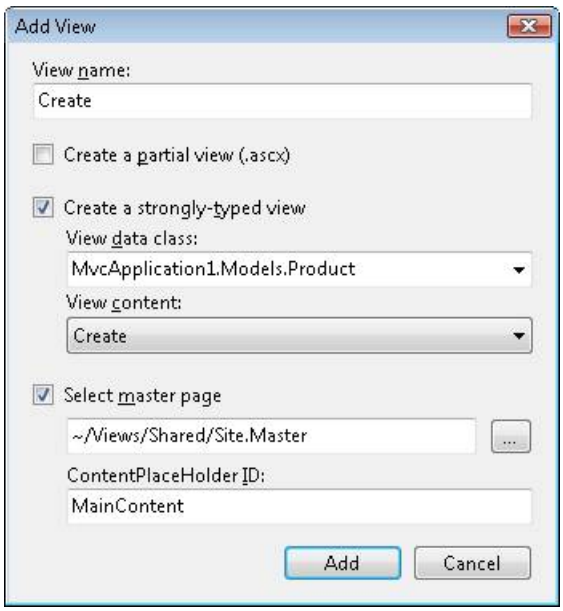


Figure 02: Creating a strongly-typed view (Click to view full-size image)

After you complete these steps, you get the Create view in Listing 2.

Listing 2 - Views\Product\Create.aspx

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of MvcApplication1.Product)" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Create</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Create</h2>

    <%= Html.ValidationSummary() %>

    <% Using Html.BeginForm()%>

        <fieldset>
            <legend>Fields</legend>

            <p>
                <label for="Name">Name:</label>
                <%= Html.TextBox("Name") %>
                <%= Html.ValidationMessage("Name", "*") %>
            </p>
            <p>
                <label for="Description">Description:</label>
                <%= Html.TextBox("Description") %>
                <%= Html.ValidationMessage("Description", "*") %>
            </p>
            <p>
                <label for="Price">Price:</label>
                <%= Html.TextBox("Price") %>
                <%= Html.ValidationMessage("Price", "*") %>
            </p>
            <p>
                <label for="UnitsInStock">UnitsInStock:</label>
                <%= Html.TextBox("UnitsInStock") %>
                <%= Html.ValidationMessage("UnitsInStock", "*") %>
            </p>
            <p>
                <input type="submit" value="Create" />
            </p>
        </fieldset>

    <% End Using %>

    <div>
        <%=Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

In Listing 2, the Html.ValidationSummary() helper is called immediately above the HTML form. This helper is used to display a list of validation error messages. The Html.ValidationSummary() helper renders the errors in a bulleted list.

The Html.ValidationMessage() helper is called next to each of the HTML form fields. This helper is used to display an error message right next to a form field. In the case of Listing 2, the Html.ValidationMessage() helper displays an asterisk when there is an error.

The page in Figure 3 illustrates the error messages rendered by the validation helpers when the form is submitted with missing fields and invalid values.

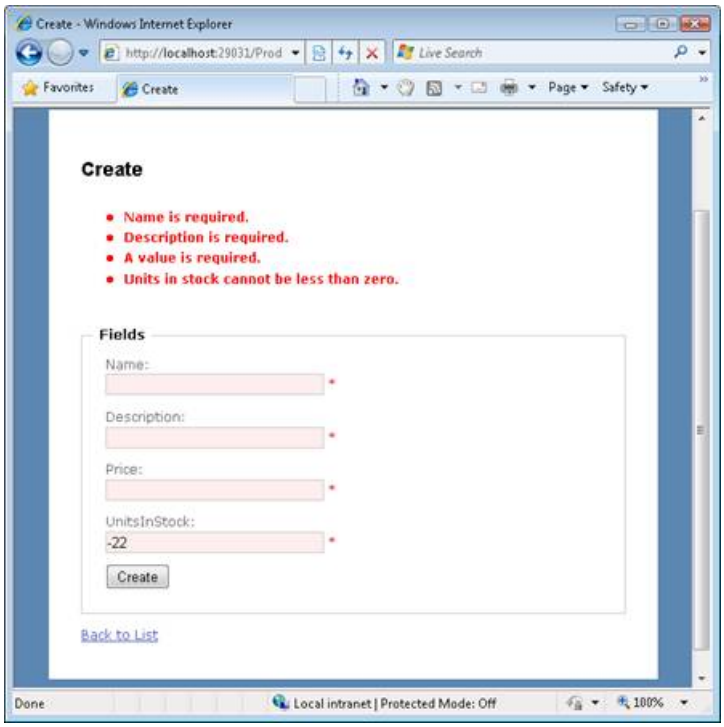


Figure 03: The Create view submitted with problems (Click to view full-size image)

Notice that the appearance of the HTML input fields are also modified when there is a validation error. The Html.TextBox() helper renders a *class="input-validation-error"* attribute when there is a validation error associated with the property rendered by the Html.TextBox() helper.

There are three cascading style sheet classes used to control the appearance of validation errors:

- input-validation-error - Applied to the <input> tag rendered by Html.TextBox() helper.
- field-validation-error - Applied to the tag rendered by the Html.ValidationMessage() helper.
- validation-summary-errors - Applied to the <ul> tag rendered by the Html.ValidationSummary() helper.

You can modify these cascading style sheet classes, and therefore modify the appearance of the validation errors, by modifying the Site.css file located in the Content folder.

> **NOTE**
>
> The HtmlHelper class includes read-only static properties for retrieving the names of the validation related CSS classes. These static properties are named ValidationInputCssClassName, ValidationFieldCssClassName, and ValidationSummaryCssClassName.

Prebinding Validation and Postbinding Validation

If you submit the HTML form for creating a Product, and you enter an invalid value for the price field and no value for the UnitsInStock field, then you'll get the validation messages displayed in Figure 4. Where do these validation error messages come from?

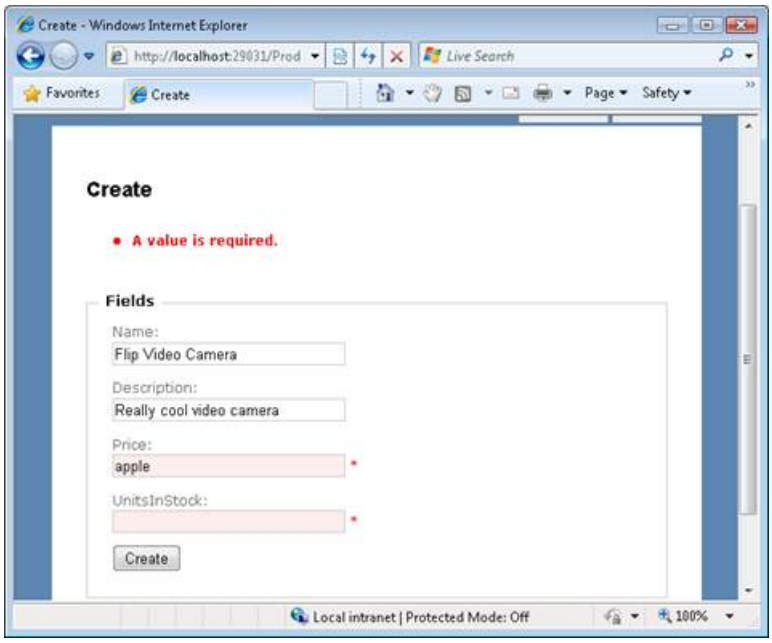


Figure 04: Prebinding Validation Errors(Click to view full-size image)

There are actually two types of validation error messages - those generated before the HTML form fields are bound to a class and those generated after the form fields are bound to the class. In other words, there are prebinding validation errors and postbinding validation errors.

The Create() action exposed by the Product controller in Listing 1 accepts an instance of the Product class. The signature of the Create method looks like this:

```
Public Function Create(<Bind(Exclude:="Id")> ByVal productToCreate As Product) As ActionResult
```

The values of the HTML form fields from the Create form are bound to the productToCreate class by something called a model binder. The default model binder adds an error message to model state automatically when it cannot bind a form field to a form property.

The default model binder cannot bind the string "apple" to the Price property of the Product class. You can't assign a string to a decimal property. Therefore, the model binder adds an error to model state.

The default model binder also cannot assign the value Nothing to a property that does not accept the value Nothing. In particular, the model binder cannot assign the value Nothing to the UnitsInStock property. Once again, the model binder gives up and adds an error message to model state.

If you want to customize the appearance of these prebinding error messages then you need to create resource strings for these messages.

Summary

The goal of this tutorial was to describe the basic mechanics of validation in the ASP.NET MVC framework. You learned how to use model state and the validation HTML helpers. We also discussed the distinction between prebinding and postbinding validation. In other tutorials, we'll discuss various strategies for moving your validation code out of your controllers and into your model classes.

PREVIOUS NEXT

Validating with the IDataErrorInfo Interface (VB)

2/20/2020 • 7 minutes to read • Edit Online

by Stephen Walther

> Stephen Walther shows you how to display custom validation error messages by implementing the IDataErrorInfo interface in a model class.

The goal of this tutorial is to explain one approach to performing validation in an ASP.NET MVC application. You learn how to prevent someone from submitting an HTML form without providing values for required form fields. In this tutorial, you learn how to perform validation by using the IErrorDataInfo interface.

Assumptions

In this tutorial, I'll use the MoviesDB database and the Movies database table. This table has the following columns:

COLUMN NAME	DATA TYPE	ALLOW NULLS
Id	Int	False
Title	Nvarchar(100)	False
Director	Nvarchar(100)	False
DateReleased	DateTime	False

In this tutorial, I use the Microsoft Entity Framework to generate my database model classes. The Movie class generated by the Entity Framework is displayed in Figure 1.

Figure 01: The Movie entity(Click to view full-size image)

> **NOTE**
>
> To learn more about using the Entity Framework to generate your database model classes, see my tutorial entitled Creating Model Classes with the Entity Framework.

The Controller Class

We use the Home controller to list movies and create new movies. The code for this class is contained in Listing 1.

Listing 1 - Controllers\HomeController.vb

```
Public Class HomeController
Inherits Controller

Private _db As New MoviesDBEntities()

Public Function Index() As ActionResult
    Return View(_db.MovieSet.ToList())
End Function

Public Function Create() As ActionResult
    Return View()
End Function

<AcceptVerbs(HttpVerbs.Post)> _
Public Function Create(<Bind(Exclude := "Id")> ByVal movieToCreate As Movie) As ActionResult
        ' Validate
        If (Not ModelState.IsValid) Then
        Return View()
        End If

    ' Add to database
    Try
        _db.AddToMovieSet(movieToCreate)
        _db.SaveChanges()

        Return RedirectToAction("Index")
    Catch
        Return View()
    End Try
End Function

End Class
```

The Home controller class in Listing 1 contains two Create() actions. The first action displays the HTML form for creating a new movie. The second Create() action performs the actual insert of the new movie into the database. The second Create() action is invoked when the form displayed by the first Create() action is submitted to the server.

Notice that the second Create() action contains the following lines of code:

```
' Validate
If (Not ModelState.IsValid) Then
Return View()
End If
```

The IsValid property returns false when there is a validation error. In that case, the Create view that contains the HTML form for creating a movie is redisplayed.

Creating a Partial Class

The Movie class is generated by the Entity Framework. You can see the code for the Movie class if you expand the MoviesDBModel.edmx file in the Solution Explorer window and open the MoviesDBModel.Designer.vb file in the Code Editor (see Figure 2).

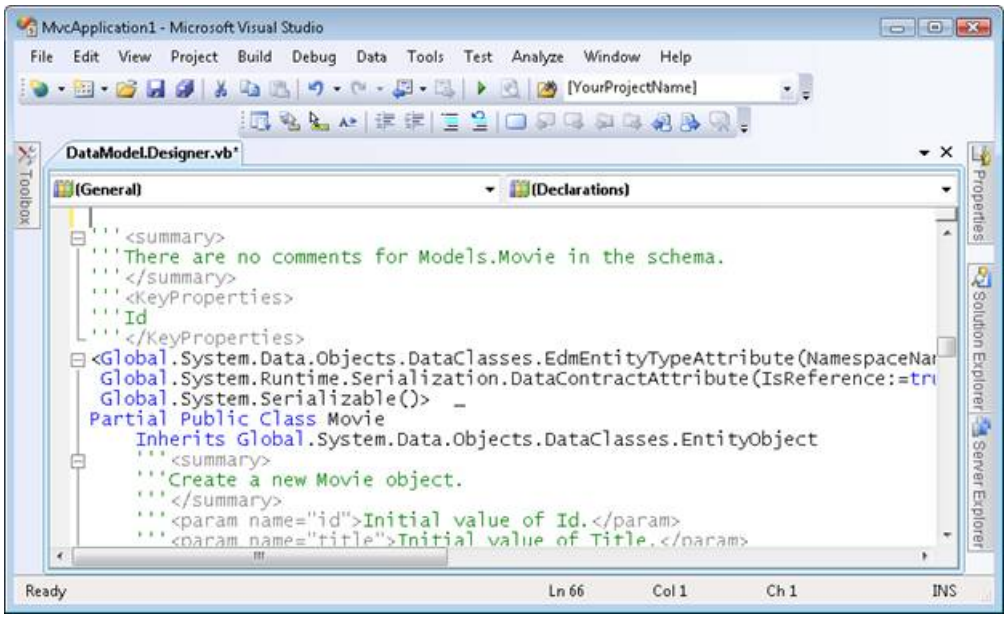


Figure 02: The code for the Movie entity(Click to view full-size image)

The Movie class is a partial class. That means that we can add another partial class with the same name to extend the functionality of the Movie class. We'll add our validation logic to the new partial class.

Add the class in Listing 2 to the Models folder.

Listing 2 - Models\Movie.vb

```
Public Partial Class Movie

End Class
```

Notice that the class in Listing 2 includes the *partial* modifier. Any methods or properties that you add to this class become part of the Movie class generated by the Entity Framework.

Adding OnChanging and OnChanged Partial Methods

When the Entity Framework generates an entity class, the Entity Framework adds partial methods to the class automatically. The Entity Framework generates OnChanging and OnChanged partial methods that correspond to each property of the class.

In the case of the Movie class, the Entity Framework creates the following methods:

- OnIdChanging
- OnIdChanged
- OnTitleChanging
- OnTitleChanged
- OnDirectorChanging
- OnDirectorChanged
- OnDateReleasedChanging
- OnDateReleasedChanged

The OnChanging method is called right before the corresponding property is changed. The OnChanged method is called right after the property is changed.

You can take advantage of these partial methods to add validation logic to the Movie class. The update Movie class in Listing 3 verifies that the Title and Director properties are assigned nonempty values.

NOTE

A partial method is a method defined in a class that you are not required to implement. If you don't implement a partial method then the compiler removes the method signature and all calls to the method so there are no run-time costs associated with the partial method. In the Visual Studio Code Editor, you can add a partial method by typing the keyword *partial* followed by a space to view a list of partials to implement.

Listing 3 - Models\Movie.vb

```
Imports System.ComponentModel

Partial Public Class Movie
    Implements IDataErrorInfo

    Private _errors As New Dictionary(Of String, String)()

    Private Sub OnTitleChanging(ByVal value As String)
        If value.Trim().Length = 0 Then
            _errors.Add("Title", "Title is required.")
        End If
    End Sub

    Private Sub OnDirectorChanging(ByVal value As String)
        If value.Trim().Length = 0 Then
            _errors.Add("Director", "Director is required.")
        End If
    End Sub

End Class
```

For example, if you attempt to assign an empty string to the Title property, then an error message is assigned to a Dictionary named _errors.

At this point, nothing actually happens when you assign an empty string to the Title property and an error is added to the private _errors field. We need to implement the IDataErrorInfo interface to expose these validation errors to the ASP.NET MVC framework.

Implementing the IDataErrorInfo Interface

The IDataErrorInfo interface has been part of the .NET framework since the first version. This interface is a very simple interface:

```
Public Interface IDataErrorInfo
  Default ReadOnly Property Item(ByVal columnName As String) As String
  ReadOnly Property [Error]() As String
End Interface
```

If a class implements the IDataErrorInfo interface, the ASP.NET MVC framework will use this interface when creating an instance of the class. For example, the Home controller Create() action accepts an instance of the Movie class:

```
<AcceptVerbs(HttpVerbs.Post)> _
Public Function Create(<Bind(Exclude := "Id")> ByVal movieToCreate As Movie) As ActionResult
    ' Validate
    If (Not ModelState.IsValid) Then
        Return View()
    End If

    ' Add to database
    Try
        _db.AddToMovieSet(movieToCreate)
        _db.SaveChanges()

        Return RedirectToAction("Index")
    Catch
        Return View()
    End Try
End Function
```

The ASP.NET MVC framework creates the instance of the Movie passed to the Create() action by using a model binder (the DefaultModelBinder). The model binder is responsible for creating an instance of the Movie object by binding the HTML form fields to an instance of the Movie object.

The DefaultModelBinder detects whether or not a class implements the IDataErrorInfo interface. If a class implements this interface then the model binder invokes the IDataErrorInfo.this indexer for each property of the class. If the indexer returns an error message then the model binder adds this error message to model state automatically.

The DefaultModelBinder also checks the IDataErrorInfo.Error property. This property is intended to represent non-property specific validation errors associated with the class. For example, you might want to enforce a validation rule that depends on the values of multiple properties of the Movie class. In that case, you would return a validation error from the Error property.

The updated Movie class in Listing 4 implements the IDataErrorInfo interface.

Listing 4 - Models\Movie.vb (implements IDataErrorInfo)

```
Imports System.ComponentModel

Partial Public Class Movie
    Implements IDataErrorInfo

    Private _errors As New Dictionary(Of String, String)()

    Private Sub OnTitleChanging(ByVal value As String)
        If value.Trim().Length = 0 Then
            _errors.Add("Title", "Title is required.")
        End If
    End Sub

    Private Sub OnDirectorChanging(ByVal value As String)
        If value.Trim().Length = 0 Then
            _errors.Add("Director", "Director is required.")
        End If
    End Sub

#Region "IDataErrorInfo Members"

    Public ReadOnly Property [Error]() As String Implements IDataErrorInfo.Error
        Get
            Return String.Empty
        End Get
    End Property

    Default Public ReadOnly Property Item(ByVal columnName As String) As String Implements IDataErrorInfo.Item
        Get
            If _errors.ContainsKey(columnName) Then
                Return _errors(columnName)
            End If
            Return String.Empty
        End Get
    End Property

#End Region

End Class
```

In Listing 4, the indexer property checks the _errors collection to see if it contains a key that corresponds to the property name passed to the indexer. If there is no validation error associated with the property then an empty string is returned.

You don't need to modify the Home controller in any way to use the modified Movie class. The page displayed in Figure 3 illustrates what happens when no value is entered for the Title or Director form fields.

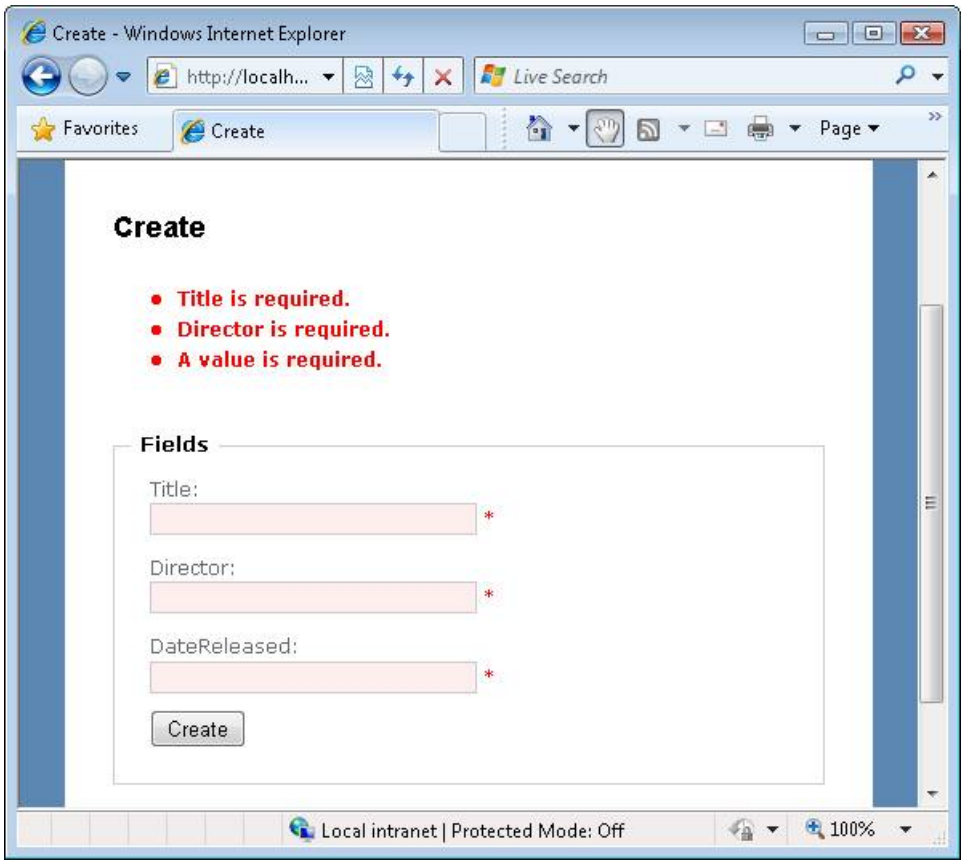


Figure 03: A form with missing values (Click to view full-size image)

Notice that the DateReleased value is validated automatically. Because the DateReleased property does not accept NULL values, the DefaultModelBinder generates a validation error for this property automatically when it does not have a value. If you want to modify the error message for the DateReleased property then you need to create a custom model binder.

Summary

In this tutorial, you learned how to use the IDataErrorInfo interface to generate validation error messages. First, we created a partial Movie class that extends the functionality of the partial Movie class generated by the Entity Framework. Next, we added validation logic to the Movie class OnTitleChanging() and OnDirectorChanging() partial methods. Finally, we implemented the IDataErrorInfo interface in order to expose these validation messages to the ASP.NET MVC framework.

PREVIOUS NEXT

Validating with a Service Layer (VB)

2/20/2020 • 6 minutes to read • Edit Online

by Stephen Walther

> Learn how to move your validation logic out of your controller actions and into a separate service layer. In this tutorial, Stephen Walther explains how you can maintain a sharp separation of concerns by isolating your service layer from your controller layer.

The goal of this tutorial is to describe one method of performing validation in an ASP.NET MVC application. In this tutorial, you learn how to move your validation logic out of your controllers and into a separate service layer.

Separating Concerns

When you build an ASP.NET MVC application, you should not place your database logic inside your controller actions. Mixing your database and controller logic makes your application more difficult to maintain over time. The recommendation is that you place all of your database logic in a separate repository layer.

For example, Listing 1 contains a simple repository named the ProductRepository. The product repository contains all of the data access code for the application. The listing also includes the IProductRepository interface that the product repository implements.

Listing 1 - Models\ProductRepository.vb

```
Public Class ProductRepository
Implements IProductRepository

    Private _entities As New ProductDBEntities()


Public Function ListProducts() As IEnumerable(Of Product) Implements IProductRepository.ListProducts
    Return _entities.ProductSet.ToList()
End Function


Public Function CreateProduct(ByVal productToCreate As Product) As Boolean Implements
IProductRepository.CreateProduct
    Try
        _entities.AddToProductSet(productToCreate)
        _entities.SaveChanges()
        Return True
    Catch
        Return False
    End Try
End Function

End Class

Public Interface IProductRepository
Function CreateProduct(ByVal productToCreate As Product) As Boolean
Function ListProducts() As IEnumerable(Of Product)
End Interface
```

The controller in Listing 2 uses the repository layer in both its Index() and Create() actions. Notice that this controller does not contain any database logic. Creating a repository layer enables you to maintain a clean separation of concerns. Controllers are responsible for application flow control logic and the repository is

responsible for data access logic.

Listing 2 - Controllers\ProductController.vb

```
Public Class ProductController
Inherits Controller

    Private _repository As IProductRepository

Public Sub New()
    Me.New(New ProductRepository())
End Sub


Public Sub New(ByVal repository As IProductRepository)
    _repository = repository
End Sub


Public Function Index() As ActionResult
    Return View(_repository.ListProducts())
End Function


'
' GET: /Product/Create

Public Function Create() As ActionResult
    Return View()
End Function

'
' POST: /Product/Create

<AcceptVerbs(HttpVerbs.Post)> _
Public Function Create(<Bind(Exclude:="Id")> ByVal productToCreate As Product) As ActionResult
    _repository.CreateProduct(productToCreate)
    Return RedirectToAction("Index")
End Function

End Class
```

Creating a Service Layer

So, application flow control logic belongs in a controller and data access logic belongs in a repository. In that case, where do you put your validation logic? One option is to place your validation logic in a *service layer*.

A service layer is an additional layer in an ASP.NET MVC application that mediates communication between a controller and repository layer. The service layer contains business logic. In particular, it contains validation logic.

For example, the product service layer in Listing 3 has a CreateProduct() method. The CreateProduct() method calls the ValidateProduct() method to validate a new product before passing the product to the product repository.

Listing 3 - Models\ProductService.vb

```
Public Class ProductService
Implements IProductService

Private _modelState As ModelStateDictionary
Private _repository As IProductRepository

Public Sub New(ByVal modelState As ModelStateDictionary, ByVal repository As IProductRepository)
    _modelState = modelState
    _repository = repository
End Sub

Protected Function ValidateProduct(ByVal productToValidate As Product) As Boolean
    If productToValidate.Name.Trim().Length = 0 Then
        _modelState.AddModelError("Name", "Name is required.")
    End If
    If productToValidate.Description.Trim().Length = 0 Then
        _modelState.AddModelError("Description", "Description is required.")
    End If
    If productToValidate.UnitsInStock
```

The Product controller has been updated in Listing 4 to use the service layer instead of the repository layer. The controller layer talks to the service layer. The service layer talks to the repository layer. Each layer has a separate responsibility.

Listing 4 - Controllers\ProductController.vb

```
Public Class ProductController
Inherits Controller

Private _service As IProductService

Public Sub New()
    _service = New ProductService(Me.ModelState, New ProductRepository())
End Sub

Public Sub New(ByVal service As IProductService)
    _service = service
End Sub


Public Function Index() As ActionResult
    Return View(_service.ListProducts())
End Function


'
' GET: /Product/Create

Public Function Create() As ActionResult
    Return View()
End Function

'
' POST: /Product/Create

<AcceptVerbs(HttpVerbs.Post)> _
Public Function Create(<Bind(Exclude := "Id")> ByVal productToCreate As Product) As ActionResult
    If Not _service.CreateProduct(productToCreate) Then
        Return View()
    End If
    Return RedirectToAction("Index")
End Function

End Class
```

Notice that the product service is created in the product controller constructor. When the product service is created, the model state dictionary is passed to the service. The product service uses model state to pass validation error messages back to the controller.

Decoupling the Service Layer

We have failed to isolate the controller and service layers in one respect. The controller and service layers communicate through model state. In other words, the service layer has a dependency on a particular feature of the ASP.NET MVC framework.

We want to isolate the service layer from our controller layer as much as possible. In theory, we should be able to use the service layer with any type of application and not only an ASP.NET MVC application. For example, in the future, we might want to build a WPF front-end for our application. We should find a way to remove the dependency on ASP.NET MVC model state from our service layer.

In Listing 5, the service layer has been updated so that it no longer uses model state. Instead, it uses any class that implements the IValidationDictionary interface.

Listing 5 - Models\ProductService.vb (decoupled)

```
Public Class ProductService
Implements IProductService

Private _validatonDictionary As IValidationDictionary
Private _repository As IProductRepository

Public Sub New(ByVal validationDictionary As IValidationDictionary, ByVal repository As IProductRepository)
    _validatonDictionary = validationDictionary
    _repository = repository
End Sub

Protected Function ValidateProduct(ByVal productToValidate As Product) As Boolean
    If productToValidate.Name.Trim().Length = 0 Then
        _validatonDictionary.AddError("Name", "Name is required.")
    End If
    If productToValidate.Description.Trim().Length = 0 Then
        _validatonDictionary.AddError("Description", "Description is required.")
    End If
    If productToValidate.UnitsInStock
```

The IValidationDictionary interface is defined in Listing 6. This simple interface has a single method and a single property.

Listing 6 - Models\IValidationDictionary.cs

```
Public Interface IValidationDictionary
Sub AddError(ByVal key As String, ByVal errorMessage As String)
ReadOnly Property IsValid() As Boolean
End Interface
```

The class in Listing 7, named the ModelStateWrapper class, implements the IValidationDictionary interface. You can instantiate the ModelStateWrapper class by passing a model state dictionary to the constructor.

Listing 7 - Models\ModelStateWrapper.vb

```
Public Class ModelStateWrapper
Implements IValidationDictionary

Private _modelState As ModelStateDictionary

Public Sub New(ByVal modelState As ModelStateDictionary)
    _modelState = modelState
End Sub

#Region "IValidationDictionary Members"

Public Sub AddError(ByVal key As String, ByVal errorMessage As String) Implements
IValidationDictionary.AddError
    _modelState.AddModelError(key, errorMessage)
End Sub

Public ReadOnly Property IsValid() As Boolean Implements IValidationDictionary.IsValid
    Get
        Return _modelState.IsValid
    End Get
End Property

#End Region

End Class
```

Finally, the updated controller in Listing 8 uses the ModelStateWrapper when creating the service layer in its constructor.

Listing 8 - Controllers\ProductController.vb

```
Public Class ProductController
Inherits Controller

Private _service As IProductService

Public Sub New()
    _service = New ProductService(New ModelStateWrapper(Me.ModelState), New ProductRepository())
End Sub

Public Sub New(ByVal service As IProductService)
    _service = service
End Sub


Public Function Index() As ActionResult
    Return View(_service.ListProducts())
End Function


'
' GET: /Product/Create

Public Function Create() As ActionResult
    Return View()
End Function

'
' POST: /Product/Create

<AcceptVerbs(HttpVerbs.Post)> _
Public Function Create(<Bind(Exclude := "Id")> ByVal productToCreate As Product) As ActionResult
    If Not _service.CreateProduct(productToCreate) Then
        Return View()
    End If
    Return RedirectToAction("Index")
End Function

End Class
```

Using the IValidationDictionary interface and the ModelStateWrapper class enables us to completely isolate our service layer from our controller layer. The service layer is no longer dependent on model state. You can pass any class that implements the IValidationDictionary interface to the service layer. For example, a WPF application might implement the IValidationDictionary interface with a simple collection class.

Summary

The goal of this tutorial was to discuss one approach to performing validation in an ASP.NET MVC application. In this tutorial, you learned how to move all of your validation logic out of your controllers and into a separate service layer. You also learned how to isolate your service layer from your controller layer by creating a ModelStateWrapper class.

PREVIOUS NEXT

Validation with the Data Annotation Validators (VB)

4/10/2020 • 8 minutes to read • Edit Online

by Microsoft

> Take advantage of the Data Annotation Model Binder to perform validation within an ASP.NET MVC application. Learn how to use the different types of validator attributes and work with them in the Microsoft Entity Framework.

In this tutorial, you learn how to use the Data Annotation validators to perform validation in an ASP.NET MVC application. The advantage of using the Data Annotation validators is that they enable you to perform validation simply by adding one or more attributes – such as the Required or StringLength attribute – to a class property.

Before you can use the Data Annotation validators, you must download the Data Annotations Model Binder. You can download the Data Annotations Model Binder Sample from the CodePlex website by clicking here.

It is important to understand that the Data Annotations Model Binder is not an official part of the Microsoft ASP.NET MVC framework. Although the Data Annotations Model Binder was created by the Microsoft ASP.NET MVC team, Microsoft does not offer official product support for the Data Annotations Model Binder described and used in this tutorial.

Using the Data Annotation Model Binder

In order to use the Data Annotations Model Binder in an ASP.NET MVC application, you first need to add a reference to the Microsoft.Web.Mvc.DataAnnotations.dll assembly and the System.ComponentModel.DataAnnotations.dll assembly. Select the menu option **Project, Add Reference**. Next click the **Browse** tab and browse to the location where you downloaded (and unzipped) the Data Annotations Model Binder sample (see **Figure 1**).

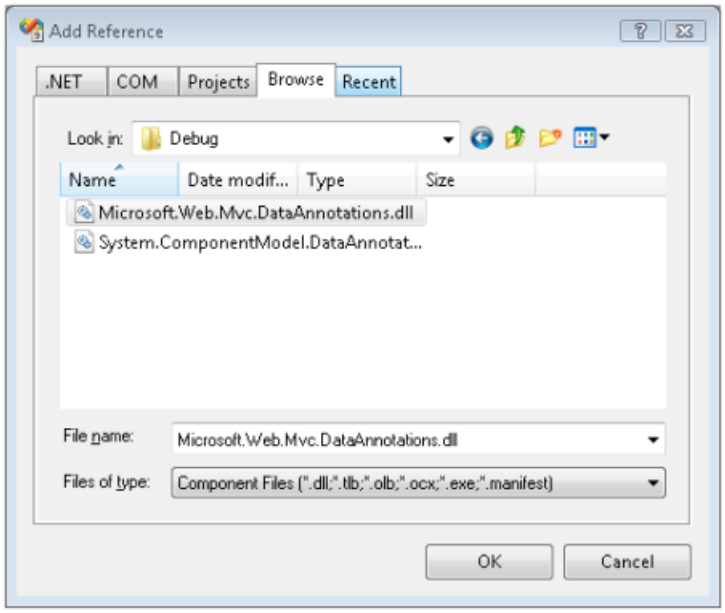


Figure 1: Adding a reference to the Data Annotations Model Binder (Click to view full-size image)

Select both the Microsoft.Web.Mvc.DataAnnotations.dll assembly and the System.ComponentModel.DataAnnotations.dll assembly and click the **OK** button.

You cannot use the System.ComponentModel.DataAnnotations.dll assembly included with .NET Framework

Service Pack 1 with the Data Annotations Model Binder. You must use the version of the System.ComponentModel.DataAnnotations.dll assembly included with the Data Annotations Model Binder Sample download.

Finally, you need to register the DataAnnotations Model Binder in the Global.asax file. Add the following line of code to the Application_Start() event handler so that the Application_Start() method looks like this:

```
Protected Sub Application_Start()
    RegisterRoutes(RouteTable.Routes)
    ModelBinders.Binders.DefaultBinder = New Microsoft.Web.Mvc.DataAnnotations.DataAnnotationsModelBinder()
End Sub
```

This line of code registers the DataAnnotationsModelBinder as the default model binder for the entire ASP.NET MVC application.

Using the Data Annotation Validator Attributes

When you use the Data Annotations Model Binder, you use validator attributes to perform validation. The System.ComponentModel.DataAnnotations namespace includes the following validator attributes:

- Range – Enables you to validate whether the value of a property falls between a specified range of values.
- RegularExpression – Enables you to validate whether the value of a property matches a specified regular expression pattern.
- Required – Enables you to mark a property as required.
- StringLength – Enables you to specify a maximum length for a string property.
- Validation – The base class for all validator attributes.

> **NOTE**
> If your validation needs are not satisfied by any of the standard validators then you always have the option of creating a custom validator attribute by inheriting a new validator attribute from the base Validation attribute.

The Product class in **Listing 1** illustrates how to use these validator attributes. The Name, Description, and UnitPrice properties are marked as required. The Name property must have a string length that is less than 10 characters. Finally, the UnitPrice property must match a regular expression pattern that represents a currency amount.

```
Imports System.ComponentModel
Imports System.ComponentModel.DataAnnotations

Public Class Product

    Private _id As Integer
    Private _name As String
    Private _description As String
    Private _unitPrice As Decimal

    Public Property Id() As Integer
        Get
            Return _id
        End Get
        Set(ByVal value As Integer)
            _id = value
        End Set
    End Property

     _
    Public Property Name() As String
        Get
            Return _name
        End Get
        Set(ByVal value As String)
            _name = value
        End Set
    End Property

     _
    Public Property Description() As String
        Get
            Return _description
        End Get
        Set(ByVal value As String)
            _description = value
        End Set
    End Property

     _
    Public Property UnitPrice() As Decimal
        Get
            Return _unitPrice
        End Get
        Set(ByVal value As Decimal)
            _unitPrice = value
        End Set
    End Property

End Class
```

Listing 1: Models\Product.vb

The Product class illustrates how to use one additional attribute: the DisplayName attribute. The DisplayName attribute enables you to modify the name of the property when the property is displayed in an error message. Instead of displaying the error message "The UnitPrice field is required" you can display the error message "The Price field is required".

> **NOTE**
>
> If you want to completely customize the error message displayed by a validator then you can assign a custom error message to the validator's ErrorMessage property like this: `<Required(ErrorMessage:="This field needs a value!")>`

You can use the Product class in **Listing 1** with the Create() controller action in **Listing 2**. This controller action redisplays the Create view when model state contains any errors.

```
Public Class ProductController
    Inherits System.Web.Mvc.Controller

    '
    ' GET: /Product/Create

    Function Create() As ActionResult
        Return View()
    End Function

    '
    ' POST: /Product/Create

     _
    Function Create( ByVal productToCreate As Product) As ActionResult

        If Not ModelState.IsValid Then
            Return View()
        End If

        Return RedirectToAction("Index")

    End Function

End Class
```

Listing 2: Controllers\ProductController.vb

Finally, you can create the view in **Listing 3** by right-clicking the Create() action and selecting the menu option **Add View**. Create a strongly-typed view with the Product class as the model class. Select **Create** from the view content dropdown list (see **Figure 2**).

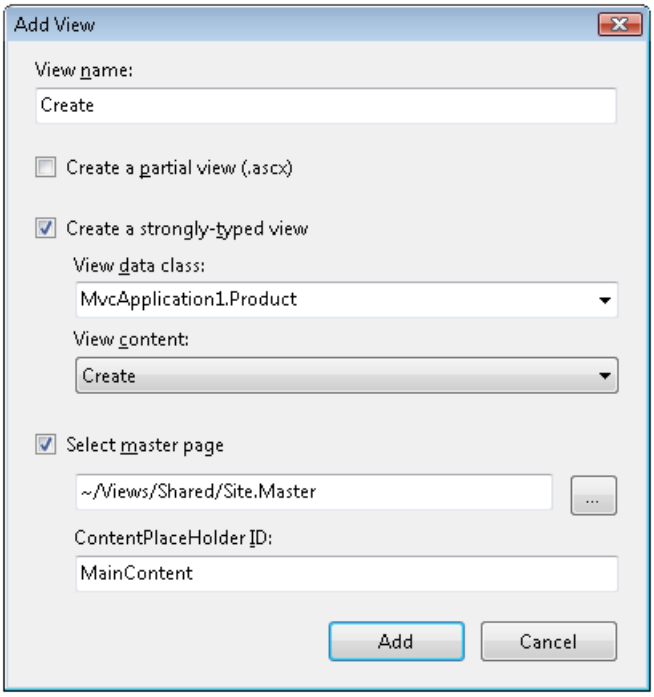


Figure 2: Adding the Create View

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of MvcApplication1.Product)" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
Create
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Create</h2>

    <%= Html.ValidationSummary("Create was unsuccessful. Please correct the errors and try again.") %>

    <% Using Html.BeginForm()%>

        <fieldset>
            <legend>Fields</legend>
            <p>
                <label for="Name">Name:</label>
                <%= Html.TextBox("Name") %>
                <%= Html.ValidationMessage("Name", "*") %>
            </p>
            <p>
                <label for="Description">Description:</label>
                <%= Html.TextBox("Description") %>
                <%= Html.ValidationMessage("Description", "*") %>
            </p>
            <p>
                <label for="UnitPrice">UnitPrice:</label>
                <%= Html.TextBox("UnitPrice") %>
                <%= Html.ValidationMessage("UnitPrice", "*") %>
            </p>
            <p>
                <input type="submit" value="Create" />
            </p>
        </fieldset>

    <% End Using %>

    <div>
        <%=Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

Listing 3: Views\Product\Create.aspx

> **NOTE**
>
> Remove the Id field from the Create form generated by the **Add View** menu option. Because the Id field corresponds to an Identity column, you don't want to allow users to enter a value for this field.

If you submit the form for creating a Product and you do not enter values for the required fields, then the validation error messages in **Figure 3** are displayed.

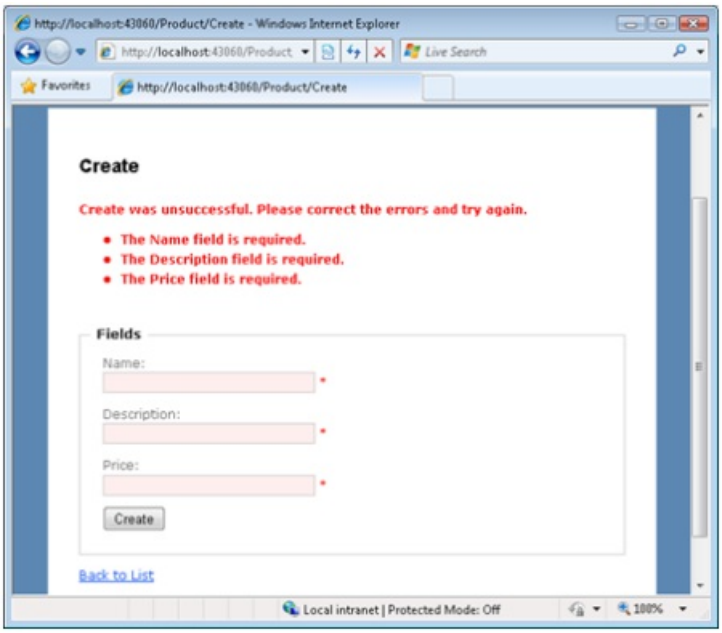


Figure 3: Missing required fields

If you enter an invalid currency amount, then the error message in **Figure 4** is displayed.

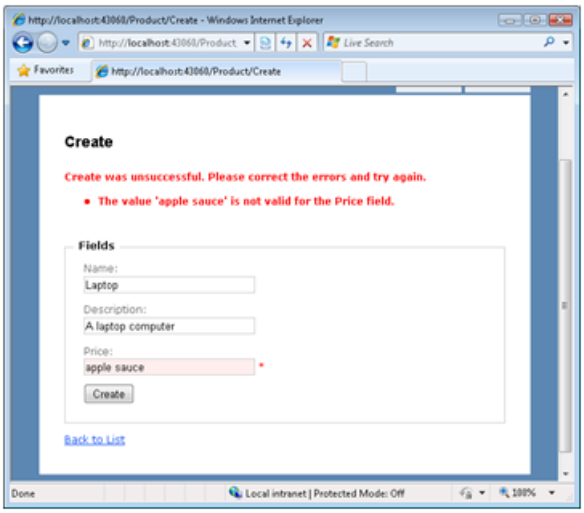


Figure 4: Invalid currency amount

Using Data Annotation Validators with the Entity Framework

If you are using the Microsoft Entity Framework to generate your data model classes then you cannot apply the validator attributes directly to your classes. Because the Entity Framework Designer generates the model classes, any changes you make to the model classes will be overwritten the next time you make any changes in the Designer.

If you want to use the validators with the classes generated by the Entity Framework then you need to create meta data classes. You apply the validators to the meta data class instead of applying the validators to the actual class.

For example, imagine that you have created a Movie class using the Entity Framework (see **Figure 5**). Imagine, furthermore, that you want to make the Movie Title and Director properties required properties. In that case, you can create the partial class and meta data class in **Listing 4**.

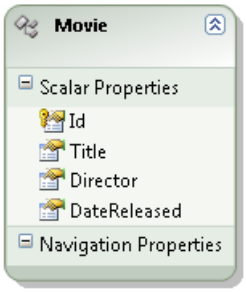


Figure 5: Movie class generated by Entity Framework

```
Imports System.ComponentModel
Imports System.ComponentModel.DataAnnotations

 _
Partial Public Class Movie
End Class

Public Class MovieMetaData

    Private _title As Object
    Private _director As Object
    Private _dateReleased As Object

     _
    Public Property Title() As Object
        Get
            Return _title
        End Get
        Set(ByVal value As Object)
            _title = value
        End Set
    End Property

     _
    Public Property Director() As Object
        Get
            Return _director
        End Get
        Set(ByVal value As Object)
            _director = value
        End Set
    End Property

     _
    Public Property DateReleased() As Object
        Get
            Return _dateReleased
        End Get
        Set(ByVal value As Object)
            _dateReleased = value
        End Set
    End Property
End Class
```

Listing 4: Models\Movie.vb

The file in **Listing 4** contains two classes named Movie and MovieMetaData. The Movie class is a partial class. It corresponds to the partial class generated by the Entity Framework that is contained in the DataModel.Designer.vb file.

Currently, the .NET framework does not support partial properties. Therefore, there is no way to apply the

validator attributes to the properties of the Movie class defined in the DataModel.Designer.vb file by applying the validator attributes to the properties of the Movie class defined in the file in **Listing 4**.

Notice that the Movie partial class is decorated with a MetadataType attribute that points at the MovieMetaData class. The MovieMetaData class contains proxy properties for the properties of the Movie class.

The validator attributes are applied to the properties of the MovieMetaData class. The Title, Director, and DateReleased properties are all marked as required properties. The Director property must be assigned a string that contains less than 5 characters. Finally, the DisplayName attribute is applied to the DateReleased property to display an error message like "The Date Released field is required." instead of the error "The DateReleased field is required."

> **NOTE**
>
> Notice that the proxy properties in the MovieMetaData class do not need to represent the same types as the corresponding properties in the Movie class. For example, the Director property is a string property in the Movie class and an object property in the MovieMetaData class.

The page in **Figure 6** illustrates the error messages returned when you enter invalid values for the Movie properties.

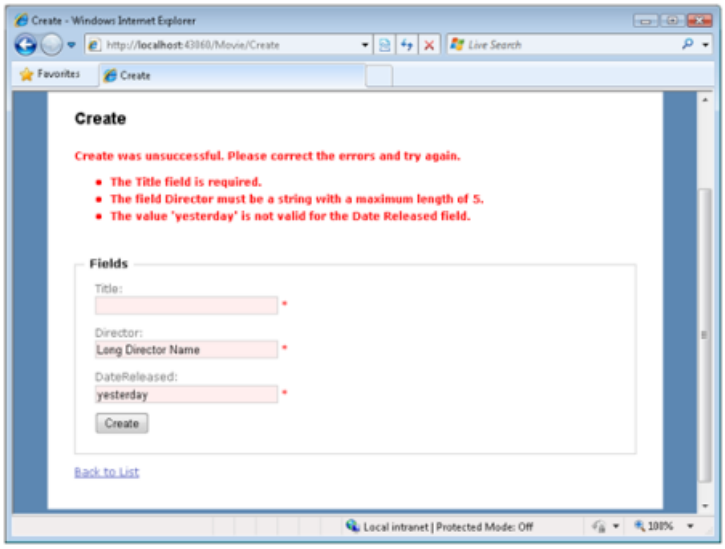


Figure 6: Using validators with the Entity Framework (Click to view full-size image)

Summary

In this tutorial, you learned how to take advantage of the Data Annotation Model Binder to perform validation within an ASP.NET MVC application. You learned how to use the different types of validator attributes such as the Required and StringLength attributes. You also learned how to use these attributes when working with the Microsoft Entity Framework.

PREVIOUS

ASP.NET MVC Views

2/20/2020 • 2 minutes to read • Edit Online

This set of tutorials provides a brief introduction to ASP.NET MVC views, view data, and HTML helpers.

- ASP.NET MVC Views Overview (C#)
- Creating Custom HTML Helpers (C#)
- TagBuilder Class to Build HTML Helpers (C#)
- Creating Page Layouts with View Master Pages (C#)
- Passing Data to View Master Pages (C#)
- ASP.NET MVC Views Overview (VB)
- Creating Custom HTML Helpers (VB)
- TagBuilder Class to Build HTML Helpers (VB)
- Creating Page Layouts with View Master Pages (VB)
- Passing Data to View Master Pages (VB)

ASP.NET MVC Views Overview (C#)

2/20/2020 • 7 minutes to read • Edit Online

by Stephen Walther

> What is an ASP.NET MVC View and how does it differ from a HTML page? In this tutorial, Stephen Walther introduces you to Views and demonstrates how you can take advantage of View Data and HTML Helpers within a View.

The purpose of this tutorial is to provide you with a brief introduction to ASP.NET MVC views, view data, and HTML Helpers. By the end of this tutorial, you should understand how to create new views, pass data from a controller to a view, and use HTML Helpers to generate content in a view.

Understanding Views

For ASP.NET or Active Server Pages, ASP.NET MVC does not include anything that directly corresponds to a page. In an ASP.NET MVC application, there is not a page on disk that corresponds to the path in the URL that you type into the address bar of your browser. The closest thing to a page in an ASP.NET MVC application is something called a *view*.

In an ASP.NET MVC application, incoming browser requests are mapped to controller actions. A controller action might return a view. However, a controller action might perform some other type of action such as redirecting you to another controller action.

Listing 1 contains a simple controller named the HomeController. The HomeController exposes two controller actions named Index() and Details().

Listing 1 - HomeController.cs

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Details()
        {
            return RedirectToAction("Index");
        }
    }
}
```

You can invoke the first action, the Index() action, by typing the following URL into your browser address bar:

/Home/Index

You can invoke the second action, the Details() action, by typing this address into your browser:

/Home/Details

The Index() action returns a view. Most actions that you create will return views. However, an action can return other types of action results. For example, the Details() action returns a RedirectToActionResult that redirects incoming request to the Index() action.

The Index() action contains the following single line of code:

View();

This line of code returns a view that must be located at the following path on your web server:

\Views\Home\Index.aspx

The path to the view is inferred from the name of the controller and the name of the controller action.

If you prefer, you can be explicit about the view. The following line of code returns a view named Fred :

View(Fred);

When this line of code is executed, a view is returned from the following path:

\Views\Home\Fred.aspx

> **NOTE**
> If you plan to create unit tests for your ASP.NET MVC application then it is a good idea to be explicit about view names. That way, you can create a unit test to verify that the expected view was returned by a controller action.

Adding Content to a View

A view is a standard (X)HTML document that can contain scripts. You use scripts to add dynamic content to a view.

For example, the view in Listing 2 displays the current date and time.

Listing 2 - \Views\Home\Index.aspx

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Index</title>
</head>
<body>
    <div>

    The current date and time is
    <% Response.Write(DateTime.Now);%>

    </div>
</body>
</html>
```

Notice that the body of the HTML page in Listing 2 contains the following script:

<% Response.Write(DateTime.Now);%>

You use the script delimiters <% and %> to mark the beginning and end of a script. This script is written in C#. It displays the current date and time by calling the Response.Write() method to render content to the browser. The script delimiters <% and %> can be used to execute one or more statements.

Since you call Response.Write() so often, Microsoft provides you with a shortcut for calling the Response.Write() method. The view in Listing 3 uses the delimiters <%= and %> as a shortcut for calling Response.Write().

Listing 3 - Views\Home\Index2.aspx

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Index</title>
</head>
<body>
    <div>

    The current date and time is
    <%=DateTime.Now %>

    </div>
</body>
</html>
```

You can use any .NET language to generate dynamic content in a view. Normally, you'll use either Visual Basic .NET or C# to write your controllers and views.

Using HTML Helpers to Generate View Content

To make it easier to add content to a view, you can take advantage of something called an *HTML Helper*. An HTML Helper, typically, is a method that generates a string. You can use HTML Helpers to generate standard HTML elements such as textboxes, links, dropdown lists, and list boxes.

For example, the view in Listing 4 takes advantage of three HTML Helpers -- the BeginForm(), the TextBox() and Password() helpers -- to generate a Login form (see Figure 1).

Listing 4 -- \Views\Home\Login.aspx

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Login Form</title>
</head>
<body>
    <div>

    <% using (Html.BeginForm())
       { %>

        <label for="UserName">User Name:</label>
        <br />
        <%= Html.TextBox("UserName") %>

        <br /><br />

        <label for="Password">Password:</label>
        <br />
        <%= Html.Password("Password") %>

        <br /><br />

        <input type="submit" value="Log in" />

    <% } %>

    </div>
</body>
</html>
```

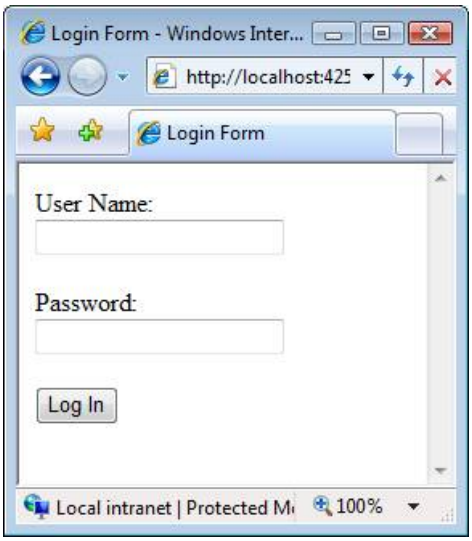


Figure 01: A standard Login form (Click to view full-size image)

All of the HTML Helpers methods are called on the Html property of the view. For example, you render a TextBox by calling the Html.TextBox() method.

Notice that you use the script delimiters <%= and %> when calling both the Html.TextBox() and Html.Password() helpers. These helpers simply return a string. You need to call Response.Write() in order to render the string to the browser.

Using HTML Helper methods is optional. They make your life easier by reducing the amount of HTML and script that you need to write. The view in Listing 5 renders the exact same form as the view in Listing 4 without using

HTML Helpers.

Listing 5 -- \Views\Home\Login.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Index4.aspx.cs" Inherits="MvcApp.Views.Home.Index4"
%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Login Form without Help</title>
</head>
<body>
    <div>

    <form method="post" action="/Home/Login">

    <label for="userName">User Name:</label>
    <br />
    <input name="userName" />

    <br /><br />

    <label for="password">Password:</label>
    <br />
    <input name="password" type="password" />

    <br /><br />
    <input type="submit" value="Log In" />

    </form>

    </div>
</body>
</html>
```

You also have the option of creating your own HTML Helpers. For example, you can create a GridView() helper method that displays a set of database records in an HTML table automatically. We explore this topic in the tutorial **Creating Custom HTML Helpers**.

Using View Data to Pass Data to a View

You use view data to pass data from a controller to a view. Think of view data like a package that you send through the mail. All data passed from a controller to a view must be sent using this package. For example, the controller in Listing 6 adds a message to view data.

Listing 6 - ProductController.cs

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    public class ProductController : Controller
    {
        public ActionResult Index()
        {
            ViewData["message"] = "Hello World!";
            return View();
        }

    }
}
```

The controller ViewData property represents a collection of name and value pairs. In Listing 6, the Index() method adds an item to the view data collection named message with the value Hello World!. When the view is returned by the Index() method, the view data is passed to the view automatically.

The view in Listing 7 retrieves the message from the view data and renders the message to the browser.

Listing 7 -- \Views\Product\Index.aspx

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Product Index</title>
</head>
<body>
    <div>

    <%= Html.Encode(ViewData["message"]) %>

    </div>
</body>
</html>
```

Notice that the view takes advantage of the Html.Encode() HTML Helper method when rendering the message. The Html.Encode() HTML Helper encodes special characters such as < and > into characters that are safe to display in a web page. Whenever you render content that a user submits to a website, you should encode the content to prevent JavaScript injection attacks.

(Because we created the message ourselves in the ProductController, we don't really need to encode the message. However, it is a good habit to always call the Html.Encode() method when displaying content retrieved from view data within a view.)

In Listing 7, we took advantage of view data to pass a simple string message from a controller to a view. You also can use view data to pass other types of data, such as a collection of database records, from a controller to a view. For example, if you want to display the contents of the Products database table in a view, then you would pass the collection of database records in view data.

You also have the option of passing strongly typed view data from a controller to a view. We explore this topic in the tutorial **Understanding Strongly Typed View Data and Views**.

Summary

This tutorial provided a brief introduction to ASP.NET MVC views, view data, and HTML Helpers. In the first section, you learned how to add new views to your project. You learned that you must add a view to the right folder in order to call it from a particular controller. Next, we discussed the topic of HTML Helpers. You learned how HTML Helpers enable you to easily generate standard HTML content. Finally, you learned how to take advantage of view data to pass data from a controller to a view.

NEXT

Creating Custom HTML Helpers (C#)

4/10/2020 • 6 minutes to read • Edit Online

by Microsoft

Download PDF

> The goal of this tutorial is to demonstrate how you can create custom HTML Helpers that you can use within your MVC views. By taking advantage of HTML Helpers, you can reduce the amount of tedious typing of HTML tags that you must perform to create a standard HTML page.

The goal of this tutorial is to demonstrate how you can create custom HTML Helpers that you can use within your MVC views. By taking advantage of HTML Helpers, you can reduce the amount of tedious typing of HTML tags that you must perform to create a standard HTML page.

In the first part of this tutorial, I describe some of the existing HTML Helpers included with the ASP.NET MVC framework. Next, I describe two methods of creating custom HTML Helpers: I explain how to create custom HTML Helpers by creating a static method and by creating an extension method.

Understanding HTML Helpers

An HTML Helper is just a method that returns a string. The string can represent any type of content that you want. For example, you can use HTML Helpers to render standard HTML tags like HTML `<input>` and `<img>` tags. You also can use HTML Helpers to render more complex content such as a tab strip or an HTML table of database data.

The ASP.NET MVC framework includes the following set of standard HTML Helpers (this is not a complete list):

- Html.ActionLink()
- Html.BeginForm()
- Html.CheckBox()
- Html.DropDownList()
- Html.EndForm()
- Html.Hidden()
- Html.ListBox()
- Html.Password()
- Html.RadioButton()
- Html.TextArea()
- Html.TextBox()

For example, consider the form in Listing 1. This form is rendered with the help of two of the standard HTML Helpers (see Figure 1). This form uses the `Html.BeginForm()` and `Html.TextBox()` Helper methods to render a simple HTML form.

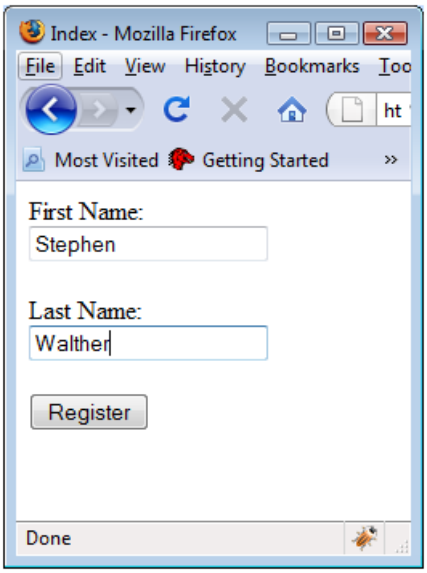


Figure 01: Page rendered with HTML Helpers (Click to view full-size image)

Listing 1 – `Views\Home\Index.aspx`

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Index.aspx.cs"
Inherits="MvcApplication1.Views.Home.Index"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns= "http://www.w3.org/1999/xhtml ">
<head id="Head1" runat="server">
     <title>Index</title>
</head>
<body>
      <div>
          <% using (Html.BeginForm())
          { %>

               <label for="firstName">First Name:</label>
               <br />
               <%= Html.TextBox("firstName")%>
               <br /><br />
               <label for="lastName">Last Name:</label>
               <br />

               <%= Html.TextBox("lastName")%>
               <br /><br />
               <input type="submit" value="Register" />
          <% } %>
     1     </div>
</body>
</html>
```

The Html.BeginForm() Helper method is used to create the opening and closing HTML `<form>` tags. Notice that the `Html.BeginForm()` method is called within a using statement. The using statement ensures that the `<form>` tag gets closed at the end of the using block.

If you prefer, instead of creating a using block, you can call the Html.EndForm() Helper method to close the `<form>` tag. Use whichever approach to creating an opening and closing `<form>` tag that seems most intuitive to you.

The `Html.TextBox()` Helper methods are used in Listing 1 to render HTML `<input>` tags. If you select view source in your browser then you see the HTML source in Listing 2. Notice that the source contains standard HTML tags.

> **IMPORTANT**
> notice that the `Html.TextBox()` -HTML Helper is rendered with `<%= %>` tags instead of `<% %>` tags. If you don't include the equal sign, then nothing gets rendered to the browser.

The ASP.NET MVC framework contains a small set of helpers. Most likely, you will need to extend the MVC framework with custom HTML Helpers. In the remainder of this tutorial, you learn two methods of creating custom HTML Helpers.

Listing 2 – `Index.aspx Source`

```
<%@ Page Language="C#" AutoEventWireup="false" CodeBehind="Index.aspx.cs" Inherits="MvcApplication1.Index" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
     <title>Index</title>
</head>
<body>
     <div>
          <form action="http://localhost:33102/" method="post">
               <label for="firstName">First Name:</label>

               <br />
               <input id="firstName" name="firstName" type="text" value="" />
               <br /><br />
               <label for="lastName">Last Name:</label>
               <br />
               <input id="lastName" name="lastName" type="text" value="" />
               <br /><br />

               <input id="btnRegister" name="btnRegister" type="submit" value="Register" />
          </form>
     </div>
</body>
</html>
```

Creating HTML Helpers with Static Methods

The easiest way to create a new HTML Helper is to create a static method that returns a string. Imagine, for example, that you decide to create a new HTML Helper that renders an HTML `<label>` tag. You can use the class in Listing 2 to render a `<label>` .

Listing 2 – `Helpers\LabelHelper.cs`

```
using System;
namespace MvcApplication1.Helpers
{
    public class LabelHelper
    {
        public static string Label(string target, string text)
        {
            return String.Format("<label for='{0}'>{1}</label>", target, text);
        }
    }
}
```

There is nothing special about the class in Listing 2. The `Label()` method simply returns a string.

The modified Index view in Listing 3 uses the `LabelHelper` to render HTML `<label>` tags. Notice that the view

includes an `<%@ imports %>` directive that imports the `Application1.Helpers` namespace.

Listing 2 – `Views\Home\Index2.aspx`

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Index2.aspx.cs"
Inherits="MvcApplication1.Views.Home.Index2"%>
<%@ Import Namespace="MvcApplication1.Helpers" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">

     <title>Index2</title>
</head>
<body>
     <div>
          <% using (Html.BeginForm())
          { %>
               <%= LabelHelper.Label("firstName", "First Name:") %>
               <br />

               <%= Html.TextBox("firstName")%>
               <br /><br />
               <%= LabelHelper.Label("lastName", "Last Name:") %>
               <br />
               <%= Html.TextBox("lastName")%>
               <br /><br />
               <input type="submit" value="Register" />
          <% } %>

     </div>
</body>
</html>
```

Creating HTML Helpers with Extension Methods

If you want to create HTML Helpers that work just like the standard HTML Helpers included in the ASP.NET MVC framework then you need to create extension methods. Extension methods enable you to add new methods to an existing class. When creating an HTML Helper method, you add new methods to the HtmlHelper class represented by a view's Html property.

The class in Listing 3 adds an extension method to the `HtmlHelper` class named `Label()`. There are a couple of things that you should notice about this class. First, notice that the class is a static class. You must define an extension method with a static class.

Second, notice that the first parameter of the `Label()` method is preceded by the keyword `this`. The first parameter of an extension method indicates the class that the extension method extends.

Listing 3 – `Helpers\LabelExtensions.cs`

```
using System;
using System.Web.Mvc;

namespace MvcApplication1.Helpers
{
     public static class LabelExtensions
     {
          public static string Label(this HtmlHelper helper, string target, string text)
          {
               return String.Format("<label for='{0}'>{1}</label>", target, text);
          }
     }
}
```

After you create an extension method, and build your application successfully, the extension method appears in Visual Studio Intellisense like all of the other methods of a class (see Figure 2). The only difference is that extension methods appear with a special symbol next to them (an icon of a downward arrow).

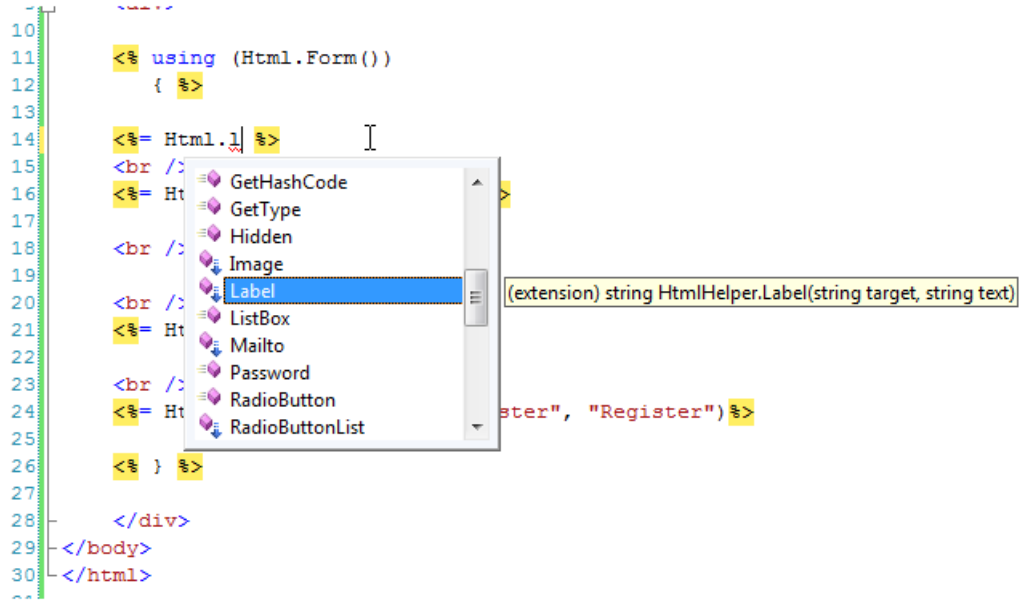


Figure 02: Using the Html.Label() extension method (Click to view full-size image)

The modified Index view in Listing 4 uses the Html.Label() extension method to render all of its `<label>` tags.

Listing 4 – `Views\Home\Index3.aspx`

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Index3.aspx.cs"
Inherits="MvcApplication1.Views.Home.Index3" %>

<%@ Import Namespace="MvcApplication1.Helpers" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
     <title>Index3</title>
</head>
<body>
     <div>

          <% using (Html.BeginForm())
          { %>
               <%= Html.Label("firstName", "First Name:") %>
               <br />
               <%= Html.TextBox("firstName")%>
               <br /><br />
               <%= Html.Label("lastName", "Last Name:") %>
               <br />
               <%= Html.TextBox("lastName")%>

               <br /><br />
               <input type="submit" value="Register" />
          <% } %>
     </div>
</body>
</html>
```

Summary

In this tutorial, you learned two methods of creating custom HTML Helpers. First, you learned how to create a custom `Label()` HTML Helper by creating a static method that returns a string. Next, you learned how to create a custom `Label()` HTML Helper method by creating an extension method on the `HtmlHelper` class.

In this tutorial, I focused on building an extremely simple HTML Helper method. Realize that an HTML Helper can be as complicated as you want. You can build HTML Helpers that render rich content such as tree views, menus, or tables of database data.

PREVIOUS NEXT

Using the TagBuilder Class to Build HTML Helpers (C#)

2/20/2020 • 3 minutes to read • Edit Online

by Stephen Walther

> Stephen Walther introduces you to a useful utility class in the ASP.NET MVC framework named the TagBuilder class. You can use the TagBuilder class to easily build HTML tags.

The ASP.NET MVC framework includes a useful utility class named the TagBuilder class that you can use when building HTML helpers. The TagBuilder class, as the name of the class suggests, enables you to easily build HTML tags. In this brief tutorial, you are provided with an overview of the TagBuilder class and you learn how to use this class when building a simple HTML helper that renders HTML <img> tags.

Overview of the TagBuilder Class

The TagBuilder class is contained in the System.Web.Mvc namespace. It has five methods:

- AddCssClass() - Enables you to add a new *class=""* attribute to a tag.
- GenerateId() - Enables you to add an id attribute to a tag. This method automatically replaces periods in the id (by default, periods are replaced by underscores)
- MergeAttribute() - Enables you to add attributes to a tag. There are multiple overloads of this method.
- SetInnerText() - Enables you to set the inner text of the tag. The inner text is HTML encode automatically.
- ToString() - Enables you to render the tag. You can specify whether you want to create a normal tag, a start tag, an end tag, or a self-closing tag.

The TagBuilder class has four important properties:

- Attributes - Represents all of the attributes of the tag.
- IdAttributeDotReplacement - Represents the character used by the GenerateId() method to replace periods (the default is an underscore).
- InnerHTML - Represents the inner contents of the tag. Assigning a string to this property *does not* HTML encode the string.
- TagName - Represents the name of the tag.

These methods and properties give you all of the basic methods and properties that you need to build up an HTML tag. You don't really need to use the TagBuilder class. You could use a StringBuilder class instead. However, the TagBuilder class makes your life a little easier.

Creating an Image HTML Helper

When you create an instance of the TagBuilder class, you pass the name of the tag that you want to build to the TagBuilder constructor. Next, you can call methods like the AddCssClass and MergeAttribute() methods to modify the attributes of the tag. Finally, you call the ToString() method to render the tag.

For example, Listing 1 contains an Image HTML helper. The Image helper is implemented internally with a TagBuilder that represents an HTML <img> tag.

Listing 1 - Helpers\ImageHelper.cs

```
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcApplication1.Helpers
{
    public static class ImageHelper
    {
        public static string Image(this HtmlHelper helper, string id, string url, string alternateText)
        {
            return Image(helper, id, url, alternateText, null);
        }

        public static string Image(this HtmlHelper helper, string id, string url, string alternateText, object
htmlAttributes)
        {
            // Create tag builder
            var builder = new TagBuilder("img");

            // Create valid id
            builder.GenerateId(id);

            // Add attributes
            builder.MergeAttribute("src", url);
            builder.MergeAttribute("alt", alternateText);
            builder.MergeAttributes(new RouteValueDictionary(htmlAttributes));

            // Render tag
            return builder.ToString(TagRenderMode.SelfClosing);
        }

    }
}
```

The class in Listing 1 contains two static overloaded methods named Image. When you call the Image() method, you can pass an object which represents a set of HTML attributes or not.

Notice how the TagBuilder.MergeAttribute() method is used to add individual attributes such as the src attribute to the TagBuilder. Notice, furthermore, how the TagBuilder.MergeAttributes() method is used to add a collection of attributes to the TagBuilder. The MergeAttributes() method accepts a Dictionary<string,object> parameter. The RouteValueDictionary class is used to convert the Object representing the collection of attributes into a Dictionary<string,object>.

After you create the Image helper, you can use the helper in your ASP.NET MVC views just like any of the other standard HTML helpers. The view in Listing 2 uses the Image helper to display the same image of an Xbox twice (see Figure 1). The Image() helper is called both with and without an HTML attribute collection.

Listing 2 - Home\Index.aspx

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage" %>
<%@ Import Namespace="MvcApplication1.Helpers" %>

<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent" runat="server">

    <!-- Calling helper without HTML attributes -->
    <%= Html.Image("img1", ResolveUrl("~/Content/XBox.jpg"), "XBox Console") %>

    <!-- Calling helper with HTML attributes -->
    <%= Html.Image("img1", ResolveUrl("~/Content/XBox.jpg"), "XBox Console", new {border="4px"})%>

</asp:Content>
```

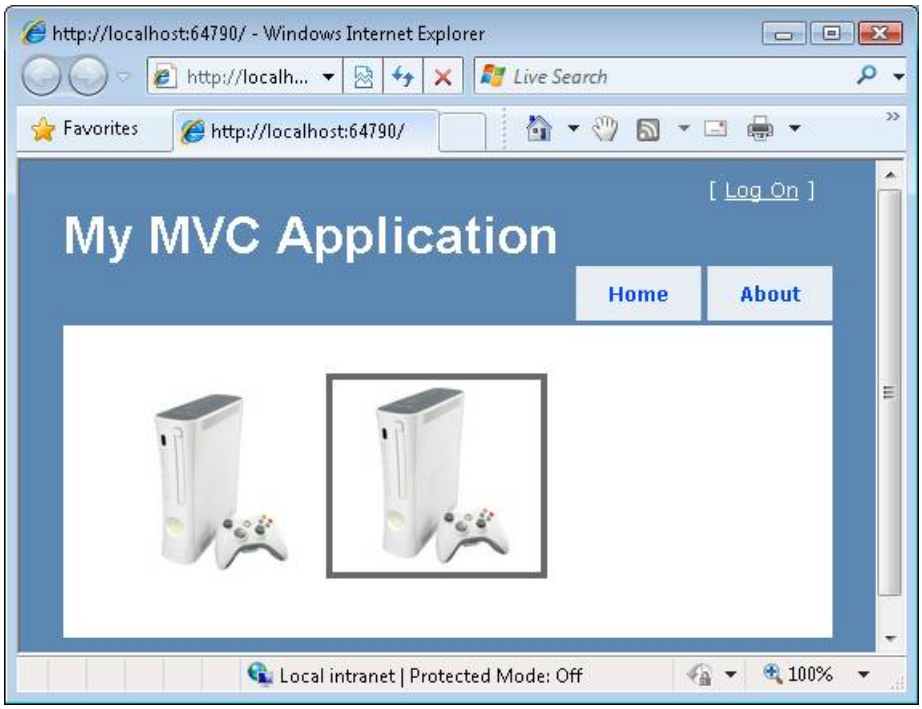


Figure 01: Using the Image helper(Click to view full-size image)

Notice that you must import the namespace associated with the Image helper at the top of the Index.aspx view. The helper is imported with the following directive:

```
<%@ Import Namespace="MvcApplication1.Helpers" %>
```

PREVIOUS NEXT

Creating Page Layouts with View Master Pages (C#)

4/10/2020 • 7 minutes to read • Edit Online

by Microsoft

Download PDF

> In this tutorial, you learn how to create a common page layout for multiple pages in your application by taking advantage of view master pages. You can use a view master page, for example, to define a two-column page layout and use the two-column layout for all of the pages in your web application.

Creating Page Layouts with View Master Pages

In this tutorial, you learn how to create a common page layout for multiple pages in your application by taking advantage of view master pages. You can use a view master page, for example, to define a two-column page layout and use the two-column layout for all of the pages in your web application.

You also can take advantage of view master pages to share common content across multiple pages in your application. For example, you can place your website logo, navigation links, and banner advertisements in a view master page. That way, every page in your application would display this content automatically.

In this tutorial, you learn how to create a new view master page and create a new view content page based on the master page.

Creating a View Master Page

Let's start by creating a view master page that defines a two-column layout. You add a new view master page to an MVC project by right-clicking the Views\Shared folder, selecting the menu option **Add, New Item**, and selecting the **MVC View Master Page** template (see Figure 1).

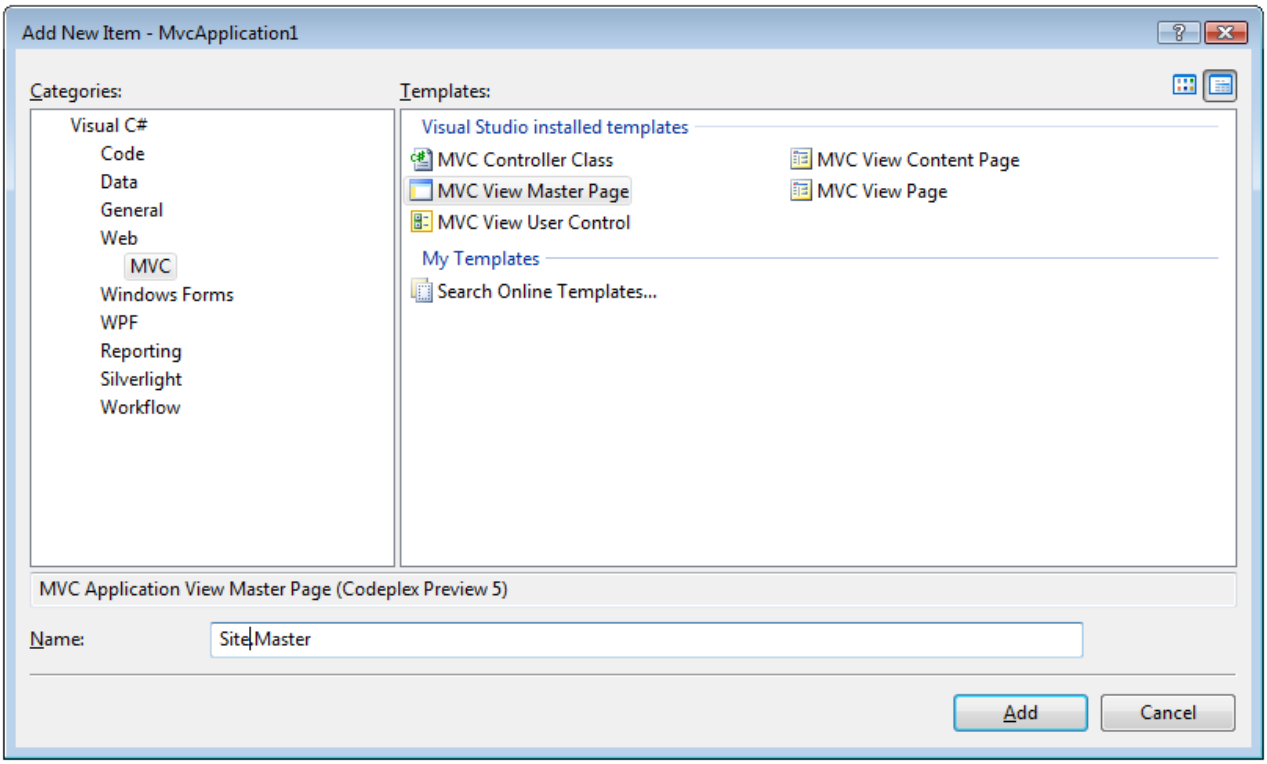


Figure 01: Adding a view master page (Click to view full-size image)

You can create more than one view master page in an application. Each view master page can define a different page layout. For example, you might want certain pages to have a two-column layout and other pages to have a three-column layout.

A view master page looks very much like a standard ASP.NET MVC view. However, unlike a normal view, a view master page contains one or more `<asp:ContentPlaceHolder>` tags. The `<contentplaceholder>` tags are used to mark the areas of the master page that can be overridden in an individual content page.

For example, the view master page in Listing 1 defines a two-column layout. It contains two `<contentplaceholder>` tags. One `<ContentPlaceHolder>` for each column.

Listing 1 – `Views\Shared\Site.master`

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site.Master.cs"
Inherits="MvcApplication1.Views.Shared.Main" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
     <title></title>

     <style type="text/css">

     html
     {
           background-color:gray;
     }

     .column
     {
          float:left;
          width:300px;
          border:solid 1px black;
          margin-right:10px;
          padding:5px;
          background-color:white;

          min-height:500px;
     }

     </style>

     <asp:ContentPlaceHolder ID="head" runat="server">
     </asp:ContentPlaceHolder>
</head>
<body>

     <h1>My Website</h1>

     <div class="column">
          <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
          </asp:ContentPlaceHolder>
     </div>
     <div class="column">
          <asp:ContentPlaceHolder ID="ContentPlaceHolder2" runat="server">
          </asp:ContentPlaceHolder>
     </div>

</body>
</html>
```

The body of the view master page in Listing 1 contains two `<div>` tags that correspond to the two columns. The Cascading Style Sheet column class is applied to both `<div>` tags. This class is defined in the style sheet declared at the top of the master page. You can preview how the view master page will be rendered by switching to Design

view. Click the Design tab at the bottom-left of the source code editor (see Figure 2).

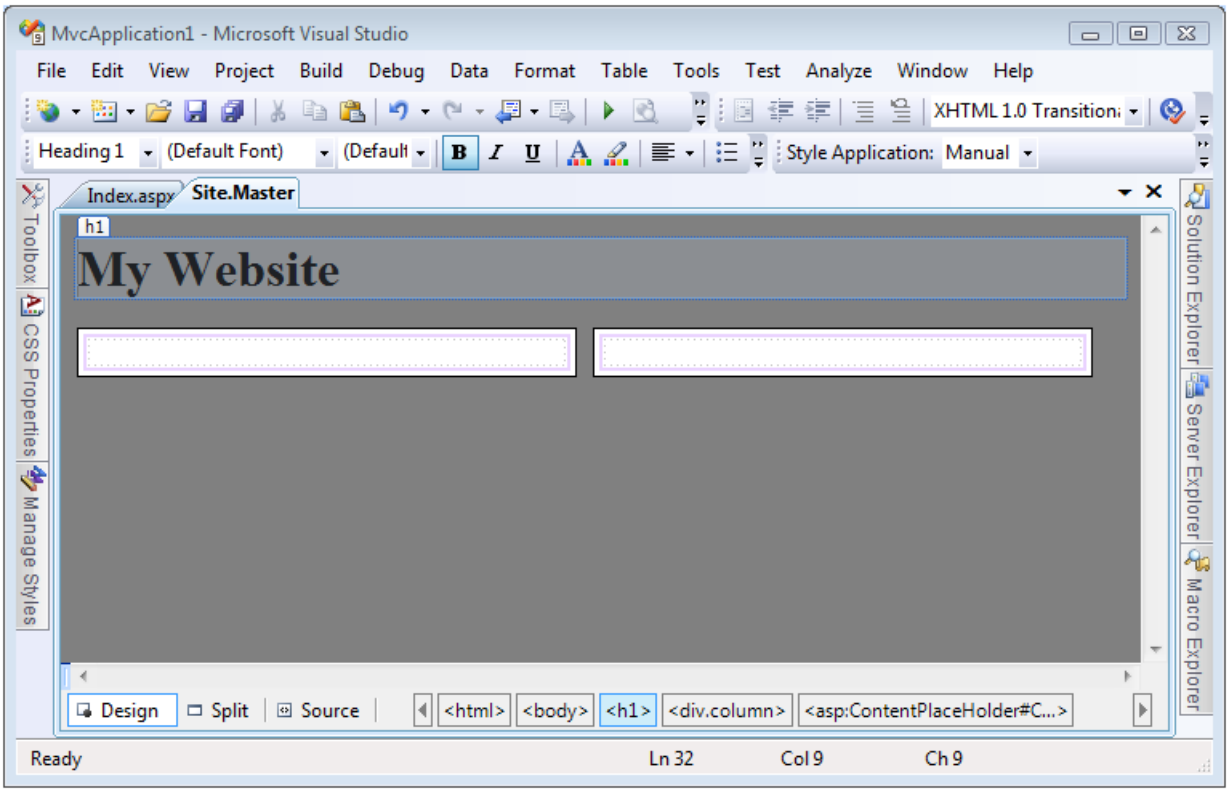


Figure 02: Previewing a master page in the designer (Click to view full-size image)

Creating a View Content Page

After you create a view master page, you can create one or more view content pages based on the view master page. For example, you can create an Index view content page for the Home controller by right-clicking the Views\Home folder, selecting **Add, New Item**, selecting the **MVC View Content Page** template, entering the name Index.aspx, and clicking the **Add** button (see Figure 3).

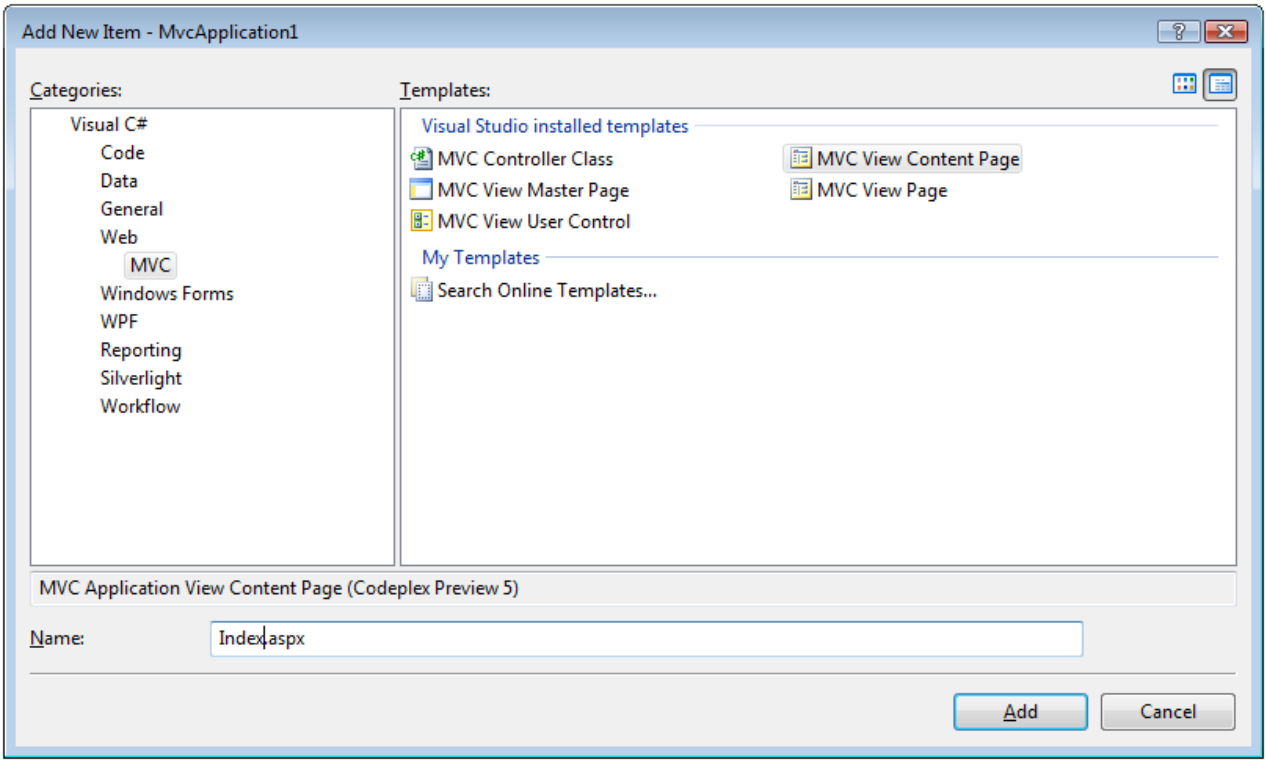


Figure 03: Adding a view content page (Click to view full-size image)

After you click the Add button, a new dialog appears that enables you to select a view master page to associate

with the view content page (see Figure 4). You can navigate to the Site.master view master page that we created in the previous section.

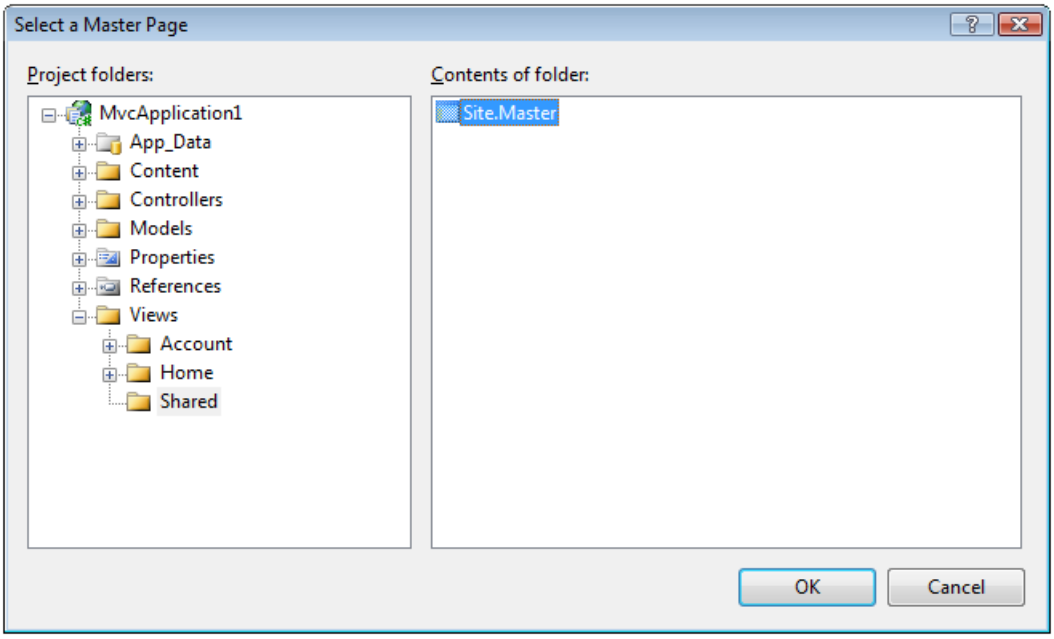


Figure 04: Selecting a master page (Click to view full-size image)

After you create a new view content page based on the Site.master master page, you get the file in Listing 2.

Listing 2 – `Views\Home\Index.aspx`

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="true"
CodeBehind="Index.aspx.cs" Inherits="MvcApplication1.Views.Home.Index" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">
</asp:Content>
<asp:Content ID="Content3" ContentPlaceHolderID="ContentPlaceHolder2" runat="server">
</asp:Content>
```

Notice that this view contains a `<asp:Content>` tag that corresponds to each of the `<asp:ContentPlaceHolder>` tags in the view master page. Each `<asp:Content>` tag includes a ContentPlaceHolderID attribute that points to the particular `<asp:ContentPlaceHolder>` that it overrides.

Notice, furthermore, that the content view page in Listing 2 does not contain any of the normal opening and closing HTML tags. For example, it does not contain the opening and closing `<html>` or `<head>` tags. All of the normal opening and closing tags are contained in the view master page.

Any content that you want to display in a view content page must be placed within a `<asp:Content>` tag. If you place any HTML or other content outside of these tags, then you will get an error when you attempt to view the page.

You don't need to override every `<asp:ContentPlaceHolder>` tag from a master page in a content view page. You only need to override a `<asp:ContentPlaceHolder>` tag when you want to replace the tag with particular content.

For example, the modified Index view in Listing 3 contains only two `<asp:Content>` tags. Each of the `<asp:Content>` tags includes some text.

Listing 3 – `Views\Home\Index.aspx (modified)`

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="true"
CodeBehind="Index.aspx.cs" Inherits="MvcApplication1.Views.Home.Index" %>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">

     <h1>Content in first column!</h1>

</asp:Content>
<asp:Content ID="Content3" ContentPlaceHolderID="ContentPlaceHolder2" runat="server">

     <h1>Content in second column!</h1>

</asp:Content>
```

When the view in Listing 3 is requested, it renders the page in Figure 5. Notice that the view renders a page with two columns. Notice, furthermore, that the content from the view content page is merged with the content from the view master page

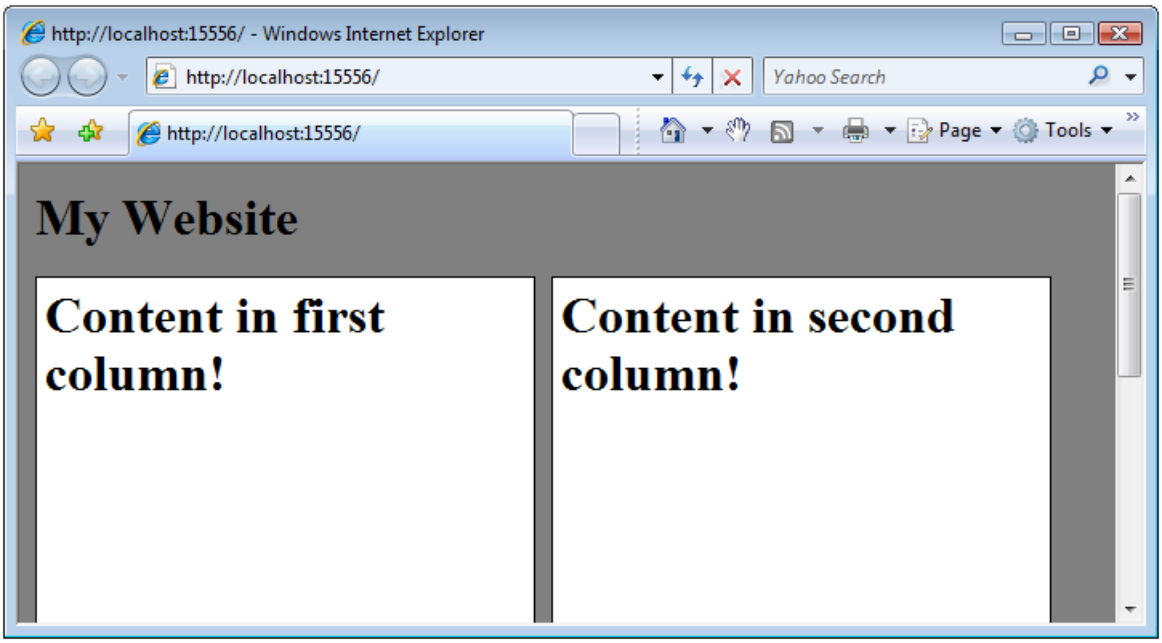


Figure 05: The Index view content page (Click to view full-size image)

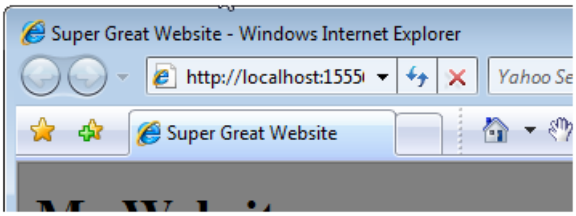
Modifying View Master Page Content

One issue that you encounter almost immediately when working with view master pages is the problem of modifying view master page content when different view content pages are requested. For example, you want each page in your web application to have a unique title. However, the title is declared in the view master page and not in the view content page. So, how do you customize the page title for each view content page?

There are two ways that you can modify the title displayed by a view content page. First, you can assign a page title to the title attribute of the `<%@ page %>` directive declared at the top of a view content page. For example, if you want to assign the page title "Super Great Website" to the Index view, then you can include the following directive at the top of the Index view:

```
<%@ page title="Super Great Website" language="C#" masterpagefile="~/Views/Shared/Site.Master"
            autoeventwireup="true" codebehind="Index.aspx.cs" inherits="MvcApplication1.Views.Home.Index"%>
```

When the Index view is rendered to the browser, the desired title appears in the browser title bar:



There is one important requirement that a master view page must satisfy in order for the title attribute to work. The view master page must contain a `<head runat="server">` tag instead of a normal `<head>` tag for its header. If the `<head>` tag does not include the runat="server" attribute then the title won't appear. The default view master page includes the required `<head runat="server">` tag.

An alternative approach to modifying master page content from an individual view content page is to wrap the region that you want to modify in a `<asp:ContentPlaceHolder>` tag. For example, imagine that you want to change not only the title, but also the meta tags, rendered by a master view page. The master view page in Listing 4 contains a `<asp:ContentPlaceHolder>` tag within its `<head>` tag.

Listing 4 – `Views\Shared\Site2.master`

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site2.Master.cs"
Inherits="MvcApplication1.Views.Shared.Site2" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>

     <asp:ContentPlaceHolder ID="head" runat="server">
          <title>Please change my title</title>
          <meta name="description" content="Please provide a description" />
          <meta name="keywords" content="keyword1,keyword2" />
     </asp:ContentPlaceHolder>
</head>
<body>
     <div>

          <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">

          </asp:ContentPlaceHolder>
     </div>
</body>
</html>
```

Notice that the `<asp:ContentPlaceHolder>` tag in Listing 4 includes default content: a default title and default meta tags. If you don't override this `<asp:ContentPlaceHolder>` tag in an individual view content page, then the default content will be displayed.

The content view page in Listing 5 overrides the `<asp:ContentPlaceHolder>` tag in order to display a custom title and custom meta tags.

Listing 5 – `Views\Home\Index2.aspx`

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site2.Master" AutoEventWireup="true"
CodeBehind="Index2.aspx.cs" Inherits="MvcApplication1.Views.Home.Index2" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
     <title>The Index2 Page</title>
     <meta name="description" content="Description of Index2 page" />
     <meta name="keywords" content="asp.net,mvc,cool,groovy" />
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">

     Just some content in the body of the page.

</asp:Content>
```

Summary

This tutorial provided you with a basic introduction to view master pages and view content pages. You learned how to create new view master pages and create view content pages based on them. We also examined how you can modify the content of a view master page from a particular view content page.

PREVIOUS NEXT

Passing Data to View Master Pages (C#)

4/10/2020 • 7 minutes to read • Edit Online

by Microsoft

Download PDF

The goal of this tutorial is to explain how you can pass data from a controller to a view master page. We examine two strategies for passing data to a view master page. First, we discuss an easy solution that results in an application that is difficult to maintain. Next, we examine a much better solution that requires a little more initial work but results in a much more maintainable application.

Passing Data to View Master Pages

The goal of this tutorial is to explain how you can pass data from a controller to a view master page. We examine two strategies for passing data to a view master page. First, we discuss an easy solution that results in an application that is difficult to maintain. Next, we examine a much better solution that requires a little more initial work but results in a much more maintainable application.

The Problem

Imagine that you are building a movie database application and you want to display the list of movie categories on every page in your application (see Figure 1). Imagine, furthermore, that the list of movie categories is stored in a database table. In that case, it would make sense to retrieve the categories from the database and render the list of movie categories within a view master page.

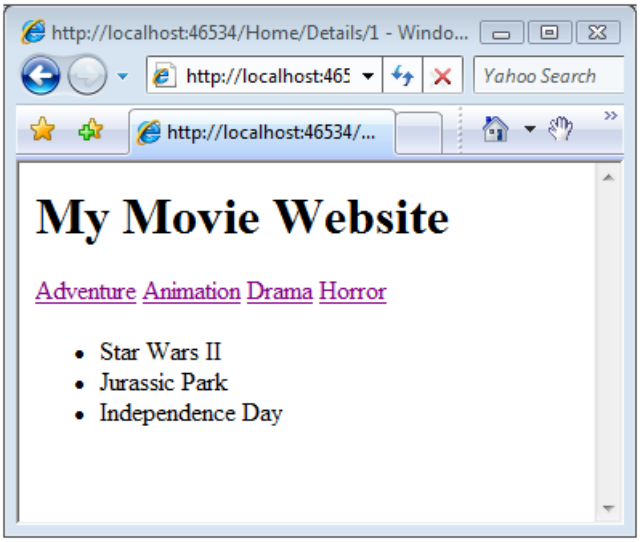


Figure 01: Displaying movie categories in a view master page (Click to view full-size image)

Here's the problem. How do you retrieve the list of movie categories in the master page? It is tempting to call methods of your model classes in the master page directly. In other words, it is tempting to include the code for retrieving the data from the database right in your master page. However, bypassing your MVC controllers to access the database would violate the clean separation of concerns that is one of the primary benefits of building an MVC application.

In an MVC application, you want all interaction between your MVC views and your MVC model to be handled by your MVC controllers. This separation of concerns results in a more maintainable, adaptable, and testable application.

In an MVC application, all data passed to a view – including a view master page – should be passed to a view by a controller action. Furthermore, the data should be passed by taking advantage of view data. In the remainder of this tutorial, I examine two methods of passing view data to a view master page.

The Simple Solution

Let's start with the simplest solution to passing view data from a controller to a view master page. The simplest solution is to pass the view data for the master page in each and every controller action.

Consider the controller in Listing 1. It exposes two actions named `Index()` and `Details()`. The `Index()` action method returns every movie in the Movies database table. The `Details()` action method returns every movie in a particular movie category.

Listing 1 – `Controllers\HomeController.cs`

```
using System.Linq;
using System.Web.Mvc;
using MvcApplication1.Models;

namespace MvcApplication1.Controllers
{
     [HandleError]
     public class HomeController : Controller
     {
          private MovieDataContext _dataContext = new MovieDataContext();

          /// <summary>

          /// Show list of all movies
          /// </summary>
          public ActionResult Index()
          {
               ViewData["categories"] = from c in _dataContext.MovieCategories
                         select c;
               ViewData["movies"] = from m in _dataContext.Movies
                         select m;
               return View();
          }

          /// <summary>
          /// Show list of movies in a category
          /// </summary>

          public ActionResult Details(int id)
          {
               ViewData["categories"] = from c in _dataContext.MovieCategories
                         select c;
               ViewData["movies"] = from m in _dataContext.Movies
                         where m.CategoryId == id
                         select m;
               return View();
          }
     }
}
```

Notice that both the Index() and the Details() actions add two items to view data. The Index() action adds two keys: categories and movies. The categories key represents the list of movie categories displayed by the view master page. The movies key represents the list of movies displayed by the Index view page.

The Details() action also adds two keys named categories and movies. The categories key, once again, represents the list of movie categories displayed by the view master page. The movies key represents the list of movies in a particular category displayed by the Details view page (see Figure 2).

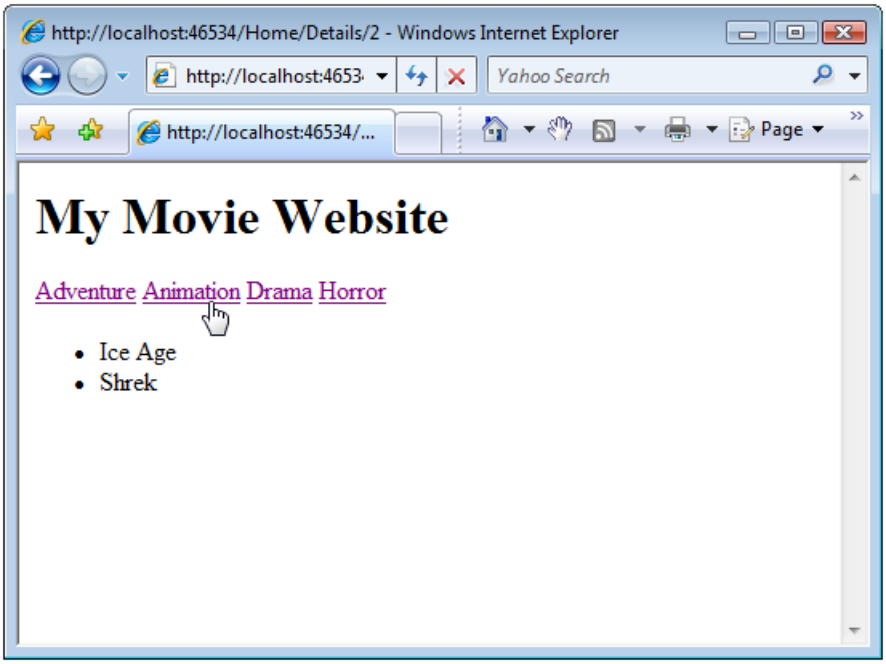


Figure 02: The Details view (Click to view full-size image)

The Index view is contained in Listing 2. It simply iterates through the list of movies represented by the movies item in view data.

Listing 2 – `Views\Home\Index.aspx`

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="true"
CodeBehind="Index.aspx.cs" Inherits="MvcApplication1.Views.Home.Index" %>

<%@ Import Namespace="MvcApplication1.Models" %>
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">

<ul>

<% foreach (var m in (IEnumerable<Movie>)ViewData["movies"])
     { %>

     <li><%= m.Title %></li>

<% } %>
</ul>

</asp:Content>
```

The view master page is contained in Listing 3. The view master page iterates and renders all of the movie categories represented by the categories item from view data.

Listing 3 – `Views\Shared\Site.master`

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="Site.Master.cs"
Inherits="MvcApplication1.Views.Shared.Site" %>
<%@ Import Namespace="MvcApplication1.Models" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
     <title></title>
     <asp:ContentPlaceHolder ID="head" runat="server">

     </asp:ContentPlaceHolder>
</head>
<body>
     <div>
          <h1>My Movie Website</h1>

          <% foreach (var c in (IEnumerable<MovieCategory>)ViewData["categories"])
                          {%>

               <%= Html.ActionLink(c.Name, "Details", new {id=c.Id} ) %>

          <% } %>


          <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">

          </asp:ContentPlaceHolder>
     </div>
</body>
</html>
```

All data is passed to the view and the view master page through view data. That is the correct way to pass data to the master page.

So, what's wrong with this solution? The problem is that this solution violates the DRY (Don't Repeat Yourself) principle. Each and every controller action must add the very same list of movie categories to view data. Having duplicate code in your application makes your application much more difficult to maintain, adapt, and modify.

The Good Solution

In this section, we examine an alternative, and better, solution to passing data from a controller action to a view master page. Instead of adding the movie categories for the master page in each and every controller action, we add the movie categories to the view data only once. All view data used by the view master page is added in an Application controller.

The ApplicationController class is contained in Listing 4.

Listing 4 – `Controllers\ApplicationController.cs`

```
using System.Linq;
using System.Web.Mvc;
using MvcApplication1.Models;

namespace MvcApplication1.Controllers
{
     public abstract class ApplicationController : Controller
     {
          private MovieDataContext _dataContext = new MovieDataContext();

          public MovieDataContext DataContext
          {
               get { return _dataContext; }
          }

          public ApplicationController()
          {
               ViewData["categories"] = from c in DataContext.MovieCategories
                         select c;
          }

     }
}
```

There are three things that you should notice about the Application controller in Listing 4. First, notice that the class inherits from the base System.Web.Mvc.Controller class. The Application controller is a controller class.

Second, notice that the Application controller class is an abstract class. An abstract class is a class that must be implemented by a concrete class. Because the Application controller is an abstract class, you cannot not invoke any methods defined in the class directly. If you attempt to invoke the Application class directly then you'll get a Resource Cannot Be Found error message.

Third, notice that the Application controller contains a constructor that adds the list of movie categories to view data. Every controller class that inherits from the Application controller calls the Application controller's constructor automatically. Whenever you call any action on any controller that inherits from the Application controller, the movie categories is included in the view data automatically.

The Movies controller in Listing 5 inherits from the Application controller.

Listing 5 – `Controllers\MoviesController.cs`

```
using System.Linq;
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
     public class MoviesController : ApplicationController
     {
          /// <summary>

          /// Show list of all movies
          /// </summary>
          public ActionResult Index()
          {
               ViewData["movies"] = from m in DataContext.Movies
                         select m;
               return View();
          }

          /// <summary>
          /// Show list of movies in a category
          /// </summary>

          public ActionResult Details(int id)
          {
               ViewData["movies"] = from m in DataContext.Movies
                         where m.CategoryId == id
                         select m;
               return View();
          }

     }
}
```

The Movies controller, just like the Home controller discussed in the previous section, exposes two action methods named `Index()` and `Details()`. Notice that the list of movie categories displayed by the view master page is not added to view data in either the `Index()` or `Details()` method. Because the Movies controller inherits from the Application controller, the list of movie categories is added to view data automatically.

Notice that this solution to adding view data for a view master page does not violate the DRY (Don't Repeat Yourself) principle. The code for adding the list of movie categories to view data is contained in only one location: the constructor for the Application controller.

Summary

In this tutorial, we discussed two approaches to passing view data from a controller to a view master page. First, we examined a simple, but difficult to maintain approach. In the first section, we discussed how you can add view data for a view master page in each every controller action in your application. We concluded that this was a bad approach because it violates the DRY (Don't Repeat Yourself) principle.

Next, we examined a much better strategy for adding data required by a view master page to view data. Instead of adding the view data in each and every controller action, we added the view data only once within an Application controller. That way, you can avoid duplicate code when passing data to a view master page in an ASP.NET MVC application.

PREVIOUS NEXT

ASP.NET MVC Views Overview (VB)

9/16/2020 • 7 minutes to read • Edit Online

by Stephen Walther

> What is an ASP.NET MVC View and how does it differ from a HTML page? In this tutorial, Stephen Walther introduces you to Views and demonstrates how you can take advantage of View Data and HTML Helpers within a View.

The purpose of this tutorial is to provide you with a brief introduction to ASP.NET MVC views, view data, and HTML Helpers. By the end of this tutorial, you should understand how to create new views, pass data from a controller to a view, and use HTML Helpers to generate content in a view.

Understanding Views

Unlike ASP.NET or Active Server Pages, ASP.NET MVC does not include anything that directly corresponds to a page. In an ASP.NET MVC application, there is not a page on disk that corresponds to the path in the URL that you type into the address bar of your browser. The closest thing to a page in an ASP.NET MVC application is something called a *view*.

In an ASP.NET MVC application, incoming browser requests are mapped to controller actions. A controller action might return a view. However, a controller action might perform some other type of action such as redirecting you to another controller action.

Listing 1 contains a simple controller named the HomeController. The HomeController exposes two controller actions named Index() and Details().

Listing 1 - HomeController.vb

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Function Index()
        Return View()
    End Function

    Function Details()
        Return RedirectToAction("Index")
    End Function
End Class
```

You can invoke the first action, the Index() action, by typing the following URL into your browser address bar:

/Home/Index

You can invoke the second action, the Details() action, by typing this address into your browser:

/Home/Details

The Index() action returns a view. Most actions that you create will return views. However, an action can return other types of action results. For example, the Details() action returns a RedirectToActionResult that redirects incoming request to the Index() action.

The Index() action contains the following single line of code:

View()

This line of code returns a view that must be located at the following path on your web server:

\Views\Home\Index.aspx

The path to the view is inferred from the name of the controller and the name of the controller action.

If you prefer, you can be explicit about the view. The following line of code returns a view named Fred :

View(Fred)

When this line of code is executed, a view is returned from the following path:

\Views\Home\Fred.aspx

> **NOTE**
> If you plan to create unit tests for your ASP.NET MVC application then it is a good idea to be explicit about view names. That way, you can create a unit test to verify that the expected view was returned by a controller action.

Adding Content to a View

A view is a standard (X)HTML document that can contain scripts. You use scripts to add dynamic content to a view.

For example, the view in Listing 2 displays the current date and time.

Listing 2 - \Views\Home\Index.aspx

```
<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Index</title>
</head>
<body>
    <div>

    The current date and time is
    <% Response.Write(DateTime.Now)%>

    </div>
</body>
</html>
```

Notice that the body of the HTML page in Listing 2 contains the following script:

<% Response.Write(DateTime.Now)%>

You use the script delimiters <% and %> to mark the beginning and end of a script. This script is written in Visual basic. It displays the current date and time by calling the Response.Write() method to render content to the browser. The script delimiters <% and %> can be used to execute one or more statements.

Since you call Response.Write() so often, Microsoft provides you with a shortcut for calling the Response.Write() method. The view in Listing 3 uses the delimiters <%= and %> as a shortcut for calling Response.Write().

Listing 3 - Views\Home\Index2.aspx

```
<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Index</title>
</head>
<body>
    <div>

    The current date and time is
    <%= DateTime.Now %>

    </div>
</body>
</html>
```

You can use any .NET language to generate dynamic content in a view. Normally, you'll use either Visual Basic .NET or C# to write your controllers and views.

Using HTML Helpers to Generate View Content

To make it easier to add content to a view, you can take advantage of something called an *HTML Helper*. An HTML Helper, typically, is a method that generates a string. You can use HTML Helpers to generate standard HTML elements such as textboxes, links, dropdown lists, and list boxes.

For example, the view in Listing 4 takes advantage of three HTML Helpers -- the BeginForm(), the TextBox() and Password() helpers -- to generate a Login form (see Figure 1).

Listing 4 -- \Views\Home\Login.aspx

```
<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Login Form</title>
</head>
<body>
    <div>

    <%  Using Html.BeginForm()%>

        <label for="UserName">User Name:</label>
        <br />
        <%= Html.TextBox("UserName") %>

        <br /><br />

        <label for="Password">Password:</label>
        <br />
        <%= Html.Password("Password") %>

        <br /><br />

        <input type="submit" value="Log in" />

    <% End Using%>

    </div>
</body>
</html>
```

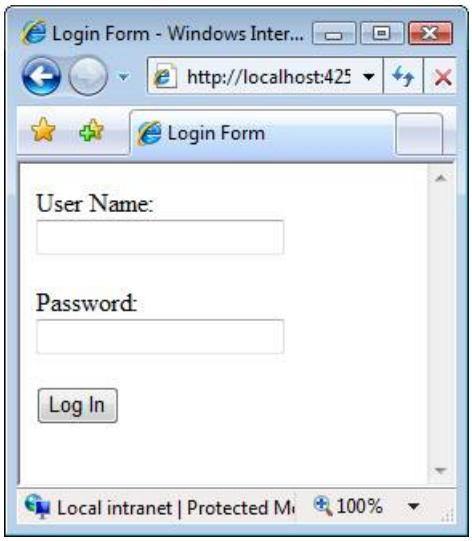


Figure 01: A standard Login form (Click to view full-size image)

All of the HTML Helpers methods are called on the Html property of the view. For example, you render a TextBox by calling the Html.TextBox() method.

Notice that you use the script delimiters <%= and %> when calling both the Html.TextBox() and Html.Password() helpers. These helpers simply return a string. You need to call Response.Write() in order to render the string to the browser.

Using HTML Helper methods is optional. They make your life easier by reducing the amount of HTML and script that you need to write. The view in Listing 5 renders the exact same form as the view in Listing 4 without using HTML Helpers.

Listing 5 -- \Views\Home\Login.aspx

```
<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Login Form</title>
</head>
<body>
    <div>

    <form method="post" action="/Home/Login">

    <label for="userName">User Name:</label>
    <br />
    <input name="userName" />

    <br /><br />

    <label for="password">Password:</label>
    <br />
    <input name="password" type="password" />

    <br /><br />
    <input type="submit" value="Log In" />

    </form>

    </div>
</body>
</html>
```

You also have the option of creating your own HTML Helpers. For example, you can create a GridView() helper method that displays a set of database records in an HTML table automatically. We explore this topic in the tutorial **Creating Custom HTML Helpers**.

Using View Data to Pass Data to a View

You use view data to pass data from a controller to a view. Think of view data like a package that you send through the mail. All data passed from a controller to a view must be sent using this package. For example, the controller in Listing 6 adds a message to view data.

Listing 6 - ProductController.vb

```
Public Class ProductController
    Inherits System.Web.Mvc.Controller

    Function Index()
        ViewData("message") = "Hello World!"
        Return View()
    End Function

End Class
```

The controller ViewData property represents a collection of name and value pairs. In Listing 6, the Index() method adds an item to the view data collection named message with the value Hello World!. When the view is returned by the Index() method, the view data is passed to the view automatically.

The view in Listing 7 retrieves the message from the view data and renders the message to the browser.

Listing 7 -- \Views\Product\Index.aspx

```
<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Product Index</title>
</head>
<body>
    <div>

    <%=Html.Encode(ViewData("message"))%>

    </div>
</body>
</html>
```

Notice that the view takes advantage of the Html.Encode() HTML Helper method when rendering the message. The Html.Encode() HTML Helper encodes special characters such as < and > into characters that are safe to display in a web page. Whenever you render content that a user submits to a website, you should encode the content to prevent JavaScript injection attacks.

(Because we created the message ourselves in the ProductController, we don't really need to encode the message. However, it is a good habit to always call the Html.Encode() method when displaying content retrieved from view data within a view.)

In Listing 7, we took advantage of view data to pass a simple string message from a controller to a view. You also can use view data to pass other types of data, such as a collection of database records, from a controller to a view. For example, if you want to display the contents of the Products database table in a view, then you would pass the collection of database records in view data.

You also have the option of passing strongly typed view data from a controller to a view. We explore this topic in the tutorial **Understanding Strongly Typed View Data and Views**.

Summary

This tutorial provided a brief introduction to ASP.NET MVC views, view data, and HTML Helpers. In the first section, you learned how to add new views to your project. You learned that you must add a view to the right folder in order to call it from a particular controller. Next, we discussed the topic of HTML Helpers. You learned how HTML Helpers enable you to easily generate standard HTML content. Finally, you learned how to take advantage of view data to pass data from a controller to a view.

PREVIOUS NEXT

Creating Custom HTML Helpers (VB)

4/10/2020 • 6 minutes to read • Edit Online

by Microsoft

Download PDF

> The goal of this tutorial is to demonstrate how you can create custom HTML Helpers that you can use within your MVC views. By taking advantage of HTML Helpers, you can reduce the amount of tedious typing of HTML tags that you must perform to create a standard HTML page.

The goal of this tutorial is to demonstrate how you can create custom HTML Helpers that you can use within your MVC views. By taking advantage of HTML Helpers, you can reduce the amount of tedious typing of HTML tags that you must perform to create a standard HTML page.

In the first part of this tutorial, I describe some of the existing HTML Helpers included with the ASP.NET MVC framework. Next, I describe two methods of creating custom HTML Helpers: I explain how to create custom HTML Helpers by creating a shared method and by creating an extension method.

Understanding HTML Helpers

An HTML Helper is just a method that returns a string. The string can represent any type of content that you want. For example, you can use HTML Helpers to render standard HTML tags like HTML `<input>` and `<img>` tags. You also can use HTML Helpers to render more complex content such as a tab strip or an HTML table of database data.

The ASP.NET MVC framework includes the following set of standard HTML Helpers (this is not a complete list):

- Html.ActionLink()
- Html.BeginForm()
- Html.CheckBox()
- Html.DropDownList()
- Html.EndForm()
- Html.Hidden()
- Html.ListBox()
- Html.Password()
- Html.RadioButton()
- Html.TextArea()
- Html.TextBox()

For example, consider the form in Listing 1. This form is rendered with the help of two of the standard HTML Helpers (see Figure 1). This form uses the `Html.BeginForm()` and `Html.TextBox()` Helper methods.

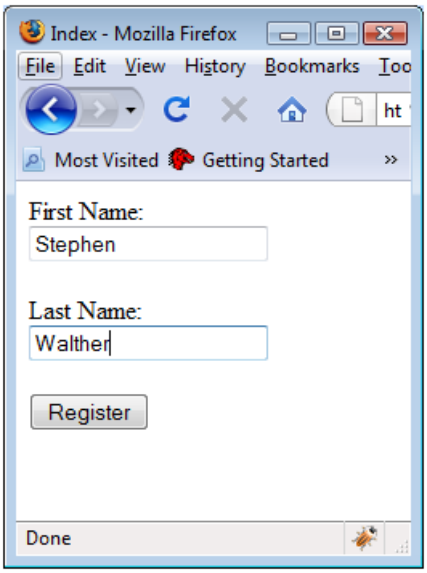


Figure 01: Page rendered with HTML Helpers (Click to view full-size image)

Listing 1 – `Views\Home\Index.aspx`

```
<%@ Page Language="VB" AutoEventWireup="false" CodeBehind="Index.aspx.vb" Inherits="MvcApplication1.Index"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns= "http://www.w3.org/1999/xhtml ">
<head id="Head1" runat="server">
     <title>Index</title>
</head>
<body>
      <div>
     <% Using Html.BeginForm()
          <label for="firstName">First Name:</label>

          <br />
          <%= Html.TextBox("firstName")%>
          <br /><br />
          <label for="lastName">Last Name:</label>
          <br />
          <%= Html.TextBox("lastName")%>
          <br /><br />

          <input type="submit" value="Register" />
     <% End Using %>
     </div>
</body>
</html>
```

The `Html.BeginForm()` Helper method is used to create the opening and closing HTML `<form>` tags. Notice that the `Html.BeginForm()` method is called within a using statement. The using statement ensures that the `<form>` tag gets closed at the end of the using block.

If you prefer, instead of creating a using block, you can call the Html.EndForm() Helper method to close the `<form>` tag. Use whichever approach to creating an opening and closing `<form>` tag that seems most intuitive to you.

The `Html.TextBox()` Helper methods are used in Listing 1 to render HTML `<input>` tags. If you select view source in your browser then you see the HTML source in Listing 2. Notice that the source contains standard HTML tags.

> **IMPORTANT**
> notice that the `Html.TextBox()` -HTML Helper is rendered with `<%= %>` tags instead of `<% %>` tags. If you don't include the equal sign, then nothing gets rendered to the browser.

The ASP.NET MVC framework contains a small set of helpers. Most likely, you will need to extend the MVC framework with custom HTML Helpers. In the remainder of this tutorial, you learn two methods of creating custom HTML Helpers.

Listing 2 – `Index.aspx Source`

```
<%@ Page Language="VB" AutoEventWireup="false" CodeBehind="Index.aspx.vb" Inherits="MvcApplication1.Index" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <title>Index</title>
</head>
<body>
     <div>
     <form action="http://localhost:33102/" method="post">

          <label for="firstName">First Name:</label>
          <br />
          <input id="firstName" name="firstName" type="text" value="" />
          <br /><br />
          <label for="lastName">Last Name:</label>
          <br />

          <input id="lastName" name="lastName" type="text" value="" />
          <br /><br />
          <input id="btnRegister" name="btnRegister" type="submit" value="Register" />
     </form>
     </div>
</body>
</html>
```

Creating HTML Helpers with Shared Methods

The easiest way to create a new HTML Helper is to create a shared method that returns a string. Imagine, for example, that you decide to create a new HTML Helper that renders an HTML `<label>` tag. You can use the class in Listing 2 to render a `<label>`.

Listing 2 – `Helpers\LabelHelper.vb`

```
Public Class LabelHelper
Public Shared Function Label(ByVal target As String, ByVal text As String) As String
     Return String.Format("<label for='{0}'>{1}</label>", target, text)
End Function
End Class
```

There is nothing special about the class in Listing 2. The `Label()` method simply returns a string.

The modified Index view in Listing 3 uses the `LabelHelper` to render HTML `<label>` tags. Notice that the view includes an `<%@ imports %>` directive that imports the Application1.Helpers namespace.

Listing 2 – `Views\Home\Index2.aspx`

```
<%@ Page Language="VB" AutoEventWireup="false" CodeBehind="Index2.aspx.vb" Inherits="MvcApplication1.Index2"%>
<%@ Import Namespace="MvcApplication1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
     <title>Index2</title>
</head>

<body>
     <div>
     <% Using Html.BeginForm()
          <%= LabelHelper.Label("firstName", "First Name:") %>
          <br />
          <%= Html.TextBox("firstName") %>
          <br /><br />
          <%= LabelHelper.Label("lastName", "Last Name:") %>
          <br />

          <%= Html.TextBox("lastName") %>
          <br /><br />
          <input type="submit" value="Register" />
     <% End Using %>
     </div>
</body>
</html>
```

Creating HTML Helpers with Extension Methods

If you want to create HTML Helpers that work just like the standard HTML Helpers included in the ASP.NET MVC framework then you need to create extension methods. Extension methods enable you to add new methods to an existing class. When creating an HTML Helper method, you add new methods to the `HtmlHelper` class represented by a view's Html property.

The Visual Basic module in Listing 3 adds an extension method named `Label()` to the `HtmlHelper` class. There are a couple of things that you should notice about this module. First, notice that the module is decorated with the `<Extension()>` attribute. In order to use this attribute, you must import the `System.Runtime.CompilerServices` namespace

Second, notice that the first parameter of the `Label()` method represents the `HtmlHelper` class. The first parameter of an extension method indicates the class that the extension method extends.

Listing 3 – `Helpers\LabelExtensions.vb`

```
Imports System.Runtime.CompilerServices

Public Module LabelExtensions
     <Extension()> _
     Public Function Label(ByVal helper As HtmlHelper, ByVal target As String, ByVal text As String) As String
          Return String.Format("<label for='{0}'> {1}</label>", target, text)

     End Function
End Module
```

After you create an extension method, and build your application successfully, the extension method appears in Visual Studio Intellisense like all of the other methods of a class (see Figure 2). The only difference is that extension methods appear with a special symbol next to them (an icon of a downward arrow).

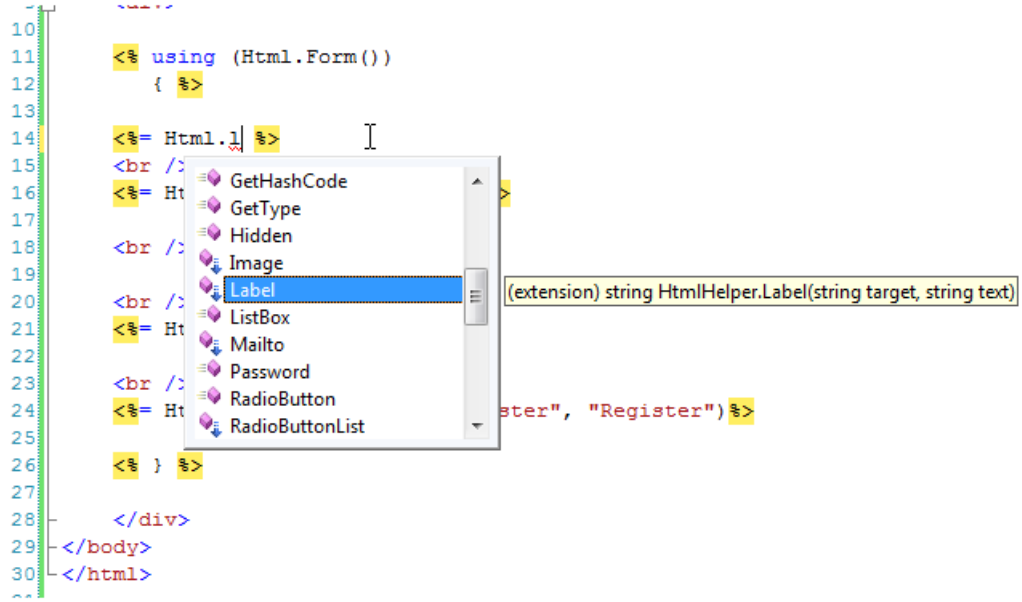


Figure 02: Using the Html.Label() extension method (Click to view full-size image)

The modified Index view in Listing 4 uses the Html.Label() extension method to render all of its <label> tags.

Listing 4 – `Views\Home\Index3.aspx`

```
<%@ Page Language="VB" AutoEventWireup="false" CodeBehind="Index3.aspx.vb" Inherits="MvcApplication1.Index3"
%>

<%@ Import Namespace="MvcApplication1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
     <title>Index3</title>
</head>
<body>
     <div>

     <% Using Html.BeginForm()%>
     <%=Html.Label("firstName", "First Name:")%>
     <br />
     <%= Html.TextBox("firstName")%>
     <br /><br />
     <%=Html.Label("lastName", "Last Name:")%>
     <br />
     <%= Html.TextBox("lastName")%>

     <br /><br />
     <input type="submit" value="Register" />
     <% End Using%>
     </div>
</body>
</html>
```

Summary

In this tutorial, you learned two methods of creating custom HTML Helpers. First, you learned how to create a custom `Label()` HTML Helper by creating a shared method that returns a string. Next, you learned how to create a custom `Label()` HTML Helper method by creating an extension method on the `HtmlHelper` class.

In this tutorial, I focused on building an extremely simple HTML Helper method. Realize that an HTML Helper can be as complicated as you want. You can build HTML Helpers that render rich content such as tree views, menus, or

tables of database data.

PREVIOUS NEXT

Using the TagBuilder Class to Build HTML Helpers (VB)

2/20/2020 • 3 minutes to read • Edit Online

by Stephen Walther

> Stephen Walther introduces you to a useful utility class in the ASP.NET MVC framework named the TagBuilder class. You can use the TagBuilder class to easily build HTML tags.

The ASP.NET MVC framework includes a useful utility class named the TagBuilder class that you can use when building HTML helpers. The TagBuilder class, as the name of the class suggests, enables you to easily build HTML tags. In this brief tutorial, you are provided with an overview of the TagBuilder class and you learn how to use this class when building a simple HTML helper that renders HTML <img> tags.

Overview of the TagBuilder Class

The TagBuilder class is contained in the System.Web.Mvc namespace. It has five methods:

- AddCssClass() – Enables you to add a new *class=""* attribute to a tag.
- GenerateId() – Enables you to add an id attribute to a tag. This method automatically replaces periods in the id (by default, periods are replaced by underscores)
- MergeAttribute() – Enables you to add attributes to a tag. There are multiple overloads of this method.
- SetInnerText() – Enables you to set the inner text of the tag. The inner text is HTML encode automatically.
- ToString() – Enables you to render the tag. You can specify whether you want to create a normal tag, a start tag, an end tag, or a self-closing tag.

The TagBuilder class has four important properties:

- Attributes – Represents all of the attributes of the tag.
- IdAttributeDotReplacement – Represents the character used by the GenerateId() method to replace periods (the default is an underscore).
- InnerHTML – Represents the inner contents of the tag. Assigning a string to this property *does not* HTML encode the string.
- TagName – Represents the name of the tag.

These methods and properties give you all of the basic methods and properties that you need to build up an HTML tag. You don't really need to use the TagBuilder class. You could use a StringBuilder class instead. However, the TagBuilder class makes your life a little easier.

Creating an Image HTML Helper

When you create an instance of the TagBuilder class, you pass the name of the tag that you want to build to the TagBuilder constructor. Next, you can call methods like the AddCssClass and MergeAttribute() methods to modify the attributes of the tag. Finally, you call the ToString() method to render the tag.

For example, Listing 1 contains an Image HTML helper. The Image helper is implemented internally with a TagBuilder that represents an HTML <img> tag.

Listing 1 – Helpers\ImageHelper.vb

```
Public Module ImageHelper

<System.Runtime.CompilerServices.Extension> _
Function Image(ByVal helper As HtmlHelper, ByVal id As String, ByVal url As String, ByVal alternateText As
String) As String
    Return Image(helper, id, url, alternateText, Nothing)
End Function

<System.Runtime.CompilerServices.Extension> _
Function Image(ByVal helper As HtmlHelper, ByVal id As String, ByVal url As String, ByVal alternateText As
String, ByVal htmlAttributes As Object) As String
    ' Create tag builder
    Dim builder = New TagBuilder("img")

    ' Create valid id
    builder.GenerateId(id)

    ' Add attributes
    builder.MergeAttribute("src", url)
    builder.MergeAttribute("alt", alternateText)
    builder.MergeAttributes(New RouteValueDictionary(htmlAttributes))

    ' Render tag
    Return builder.ToString(TagRenderMode.SelfClosing)
End Function

End Module
```

The module in Listing 1 contains two overloaded methods named Image(). When you call the Image() method, you can pass an object which represents a set of HTML attributes or not.

Notice how the TagBuilder.MergeAttribute() method is used to add individual attributes such as the src attribute to the TagBuilder. Notice, furthermore, how the TagBuilder.MergeAttributes() method is used to add a collection of attributes to the TagBuilder. The MergeAttributes() method accepts a Dictionary<string,object> parameter. The RouteValueDictionary class is used to convert the Object representing the collection of attributes into a Dictionary<string,object>.

After you create the Image helper, you can use the helper in your ASP.NET MVC views just like any of the other standard HTML helpers. The view in Listing 2 uses the Image helper to display the same image of an Xbox twice (see Figure 1). The Image() helper is called both with and without an HTML attribute collection.

Listing 2 – Home\Index.aspx

```
<%@ Page Language="VB" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage" %>
<%@ Import Namespace="MvcApplication1" %>
<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent" runat="server">

    <!-- Calling helper without HTML attributes -->
    <%= Html.Image("img1", ResolveUrl("~/Content/XBox.jpg"), "XBox Console") %>

    <!-- Calling helper with HTML attributes -->
    <%= Html.Image("img1", ResolveUrl("~/Content/XBox.jpg"), "XBox Console", New With {.border="4px"})%>

</asp:Content>
```

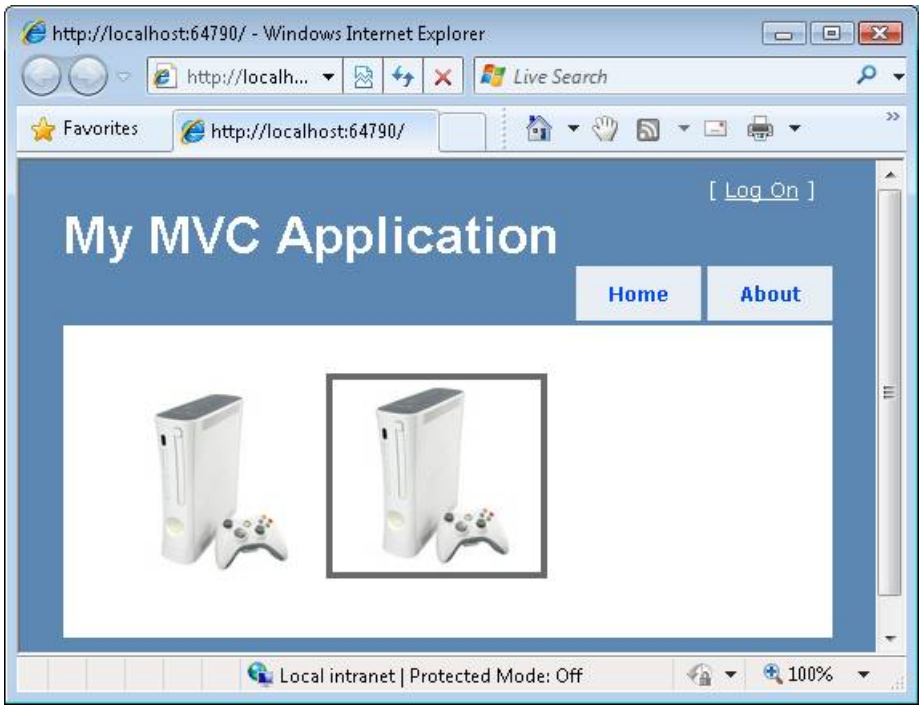


Figure 01: Using the Image helper(Click to view full-size image)

Notice that you must import the namespace associated with the Image helper at the top of the Index.aspx view. The helper is imported with the following directive:

```
<%@ Import Namespace="MvcApplication1" %>
```

In a Visual Basic application, the default namespace is the same as the name of the application.

PREVIOUS NEXT

Creating Page Layouts with View Master Pages (VB)

4/10/2020 • 7 minutes to read • Edit Online

by Microsoft

Download PDF

> In this tutorial, you learn how to create a common page layout for multiple pages in your application by taking advantage of view master pages. You can use a view master page, for example, to define a two-column page layout and use the two-column layout for all of the pages in your web application.

Creating Page Layouts with View Master Pages

In this tutorial, you learn how to create a common page layout for multiple pages in your application by taking advantage of view master pages. You can use a view master page, for example, to define a two-column page layout and use the two-column layout for all of the pages in your web application.

You also can take advantage of view master pages to share common content across multiple pages in your application. For example, you can place your website logo, navigation links, and banner advertisements in a view master page. That way, every page in your application would display this content automatically.

In this tutorial, you learn how to create a new view master page and create a new view content page based on the master page.

Creating a View Master Page

Let's start by creating a view master page that defines a two-column layout. You add a new view master page to an MVC project by right-clicking the Views\Shared folder, selecting the menu option **Add, New Item**, and selecting the MVC View Master Page template (see Figure 1).

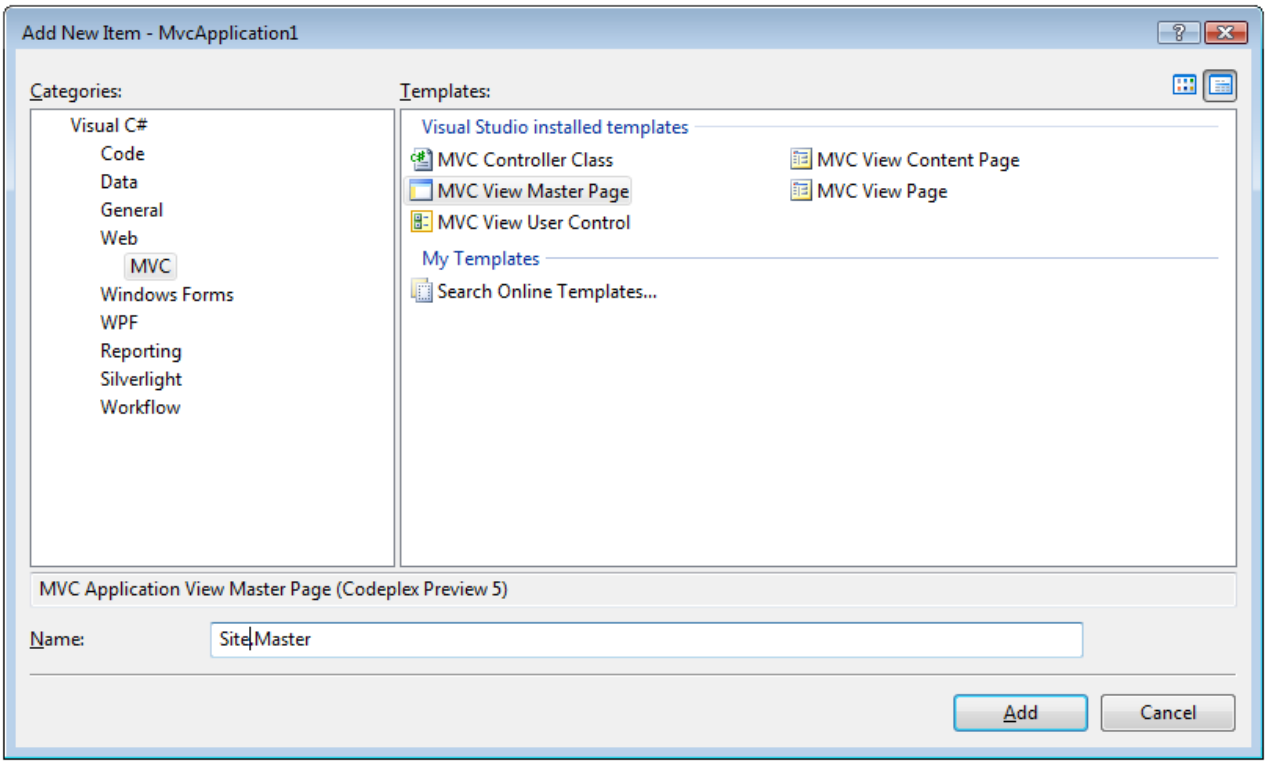


Figure 01: Adding a view master page (Click to view full-size image)

You can create more than one view master page in an application. Each view master page can define a different page layout. For example, you might want certain pages to have a two-column layout and other pages to have a three-column layout.

A view master page looks very much like a standard ASP.NET MVC view. However, unlike a normal view, a view master page contains one or more `<asp:ContentPlaceHolder>` tags. The `<contentplaceholder>` tags are used to mark the areas of the master page that can be overridden in an individual content page.

For example, the view master page in Listing 1 defines a two-column layout. It contains two `<contentplaceholder>` tags. One `<ContentPlaceHolder>` for each column.

Listing 1 – `Views\Shared\Site.master`

```
<%@ Master Language="VB" AutoEventWireup="false" CodeBehind="Site.Master.vb" Inherits="MvcApplication1.Site"
%>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
     <title></title>

     <style type="text/css">

     html
     {
           background-color:gray;

     }

     .column
     {
          float:left;
          width:300px;
          border:solid 1px black;
          margin-right:10px;
          padding:5px;
          background-color:white;
          min-height:500px;
     }

     </style>

     <asp:ContentPlaceHolder ID="head" runat="server">
     </asp:ContentPlaceHolder>

</head>
<body>

     <h1>My Website</h1>

     <div class="column">
          <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">
          </asp:ContentPlaceHolder>
     </div>

     <div class="column">
          <asp:ContentPlaceHolder ID="ContentPlaceHolder2" runat="server">
          </asp:ContentPlaceHolder>
     </div>

</body>
</html>
```

The body of the view master page in Listing 1 contains two `<div>` tags that correspond to the two columns. The Cascading Style Sheet column class is applied to both `<div>` tags. This class is defined in the style sheet declared at the top of the master page. You can preview how the view master page will be rendered by switching to Design view. Click the Design tab at the bottom-left of the source code editor (see Figure 2).

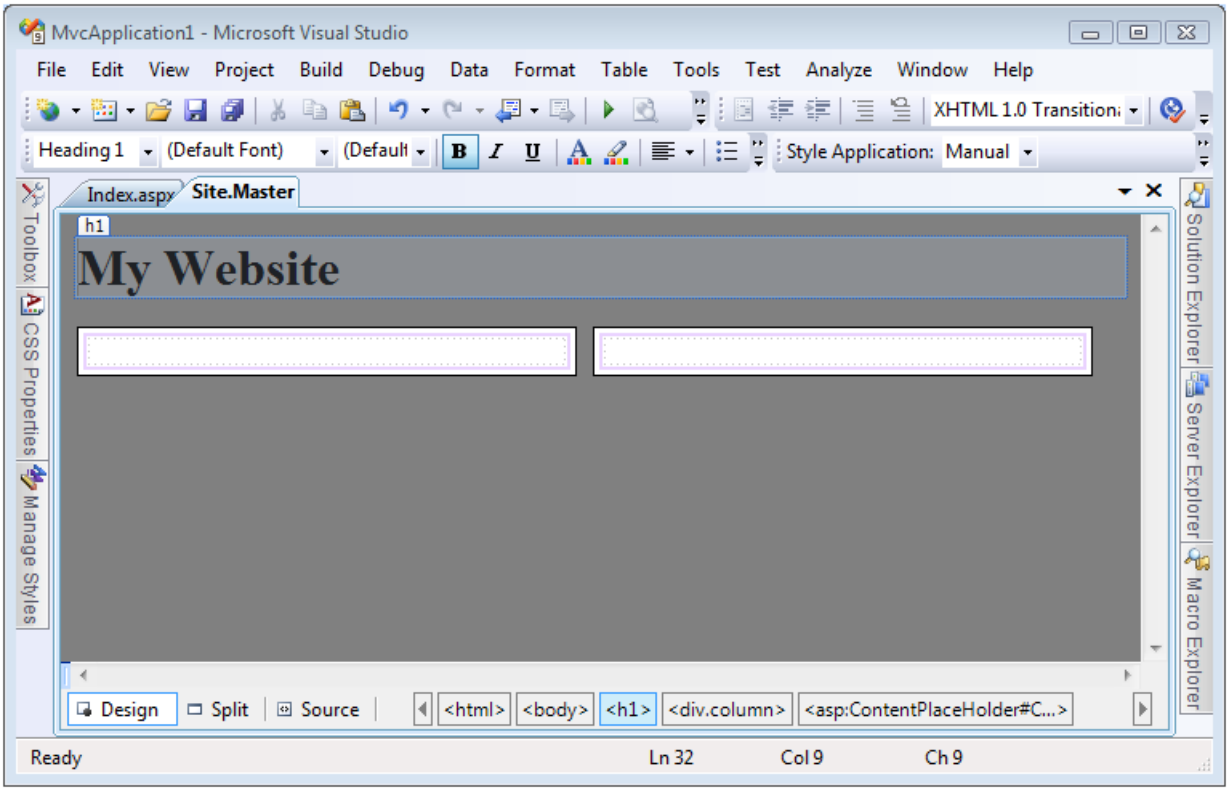


Figure 02: Previewing a master page in the designer (Click to view full-size image)

Creating a View Content Page

After you create a view master page, you can create one or more view content pages based on the view master page. For example, you can create an Index view content page for the Home controller by right-clicking the Views\Home folder, selecting **Add, New Item**, selecting the **MVC View Content Page** template, entering the name Index.aspx, and clicking the Add button (see Figure 3).

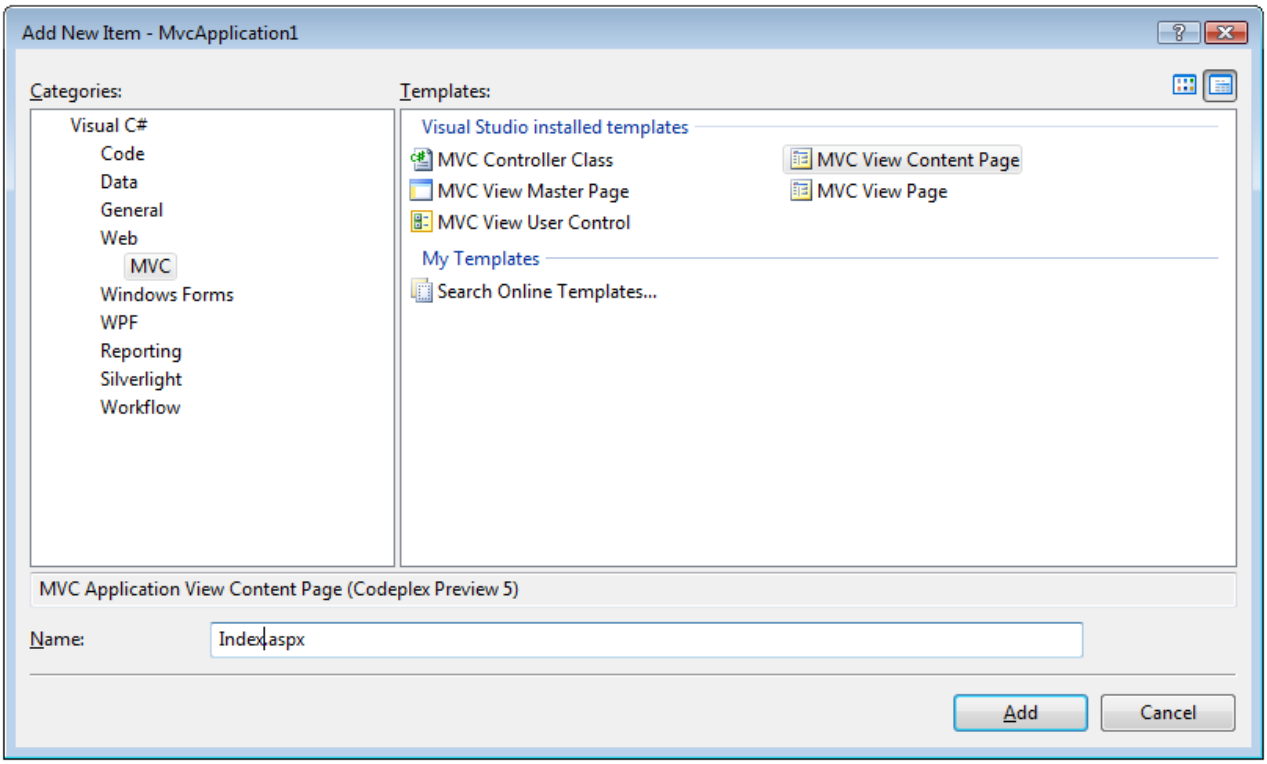


Figure 03: Adding a view content page (Click to view full-size image)

After you click the Add button, a new dialog appears that enables you to select a view master page to associate with the view content page (see Figure 4). You can navigate to the Site.master view master page that we created in the previous section.

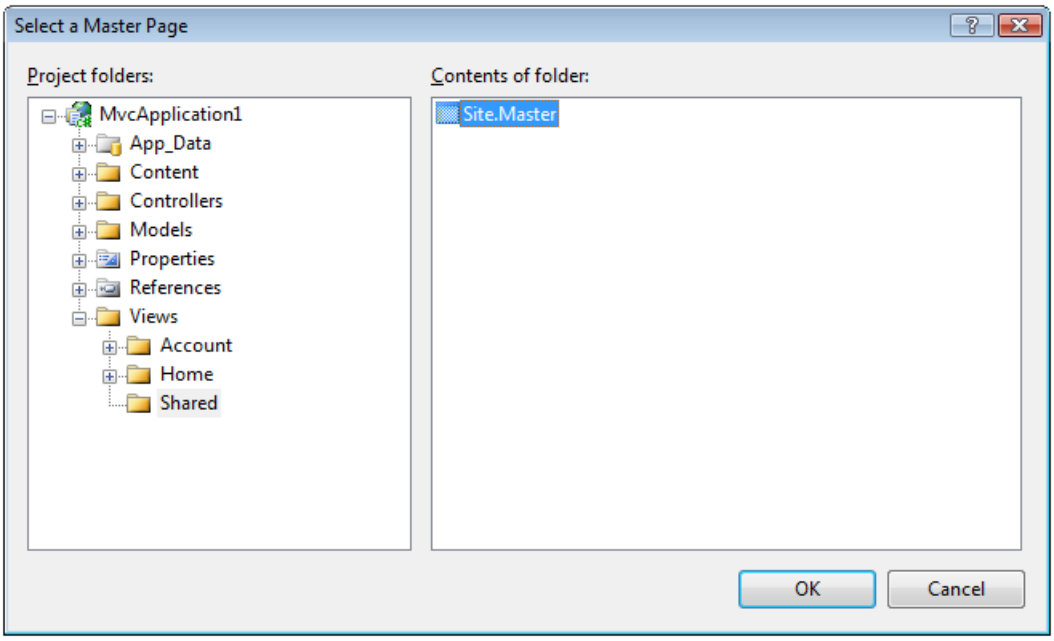


Figure 04: Selecting a master page (Click to view full-size image)

After you create a new view content page based on the Site.master master page, you get the file in Listing 2.

Listing 2 – `Views\Home\Index.aspx`

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="false"
CodeBehind="Index.aspx.vb" Inherits="MvcApplication1.Index" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">
</asp:Content>
<asp:Content ID="Content3" ContentPlaceHolderID="ContentPlaceHolder2" runat="server">
</asp:Content>
```

Notice that this view contains a `<asp:Content>` tag that corresponds to each of the `<asp:ContentPlaceHolder>` tags in the view master page. Each `<asp:Content>` tag includes a ContentPlaceHolderID attribute that points to the particular `<asp:ContentPlaceHolder>` that it overrides.

Notice, furthermore, that the content view page in Listing 2 does not contain any of the normal opening and closing HTML tags. For example, it does not contain the opening and closing `<html>` or `<head>` tags. All of the normal opening and closing tags are contained in the view master page.

Any content that you want to display in a view content page must be placed within a `<asp:Content>` tag. If you place any HTML or other content outside of these tags, then you will get an error when you attempt to view the page.

You don't need to override every `<asp:ContentPlaceHolder>` tag from a master page in a content view page. You only need to override a `<asp:ContentPlaceHolder>` tag when you want to replace the tag with particular content.

For example, the modified Index view in Listing 3 contains only two `<asp:Content>` tags. Each of the `<asp:Content>` tags includes some text.

Listing 3 – `Views\Home\Index.aspx (modified)`

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="false"
CodeBehind="Index.aspx.vb" Inherits="MvcApplication1.Index" %>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">

     <h1>Content in first column!</h1>

</asp:Content>
<asp:Content ID="Content3" ContentPlaceHolderID="ContentPlaceHolder2" runat="server">

     <h1>Content in second column!</h1>

</asp:Content>
```

When the view in Listing 3 is requested, it renders the page in Figure 5. Notice that the view renders a page with two columns. Notice, furthermore, that the content from the view content page is merged with the content from the view master page.

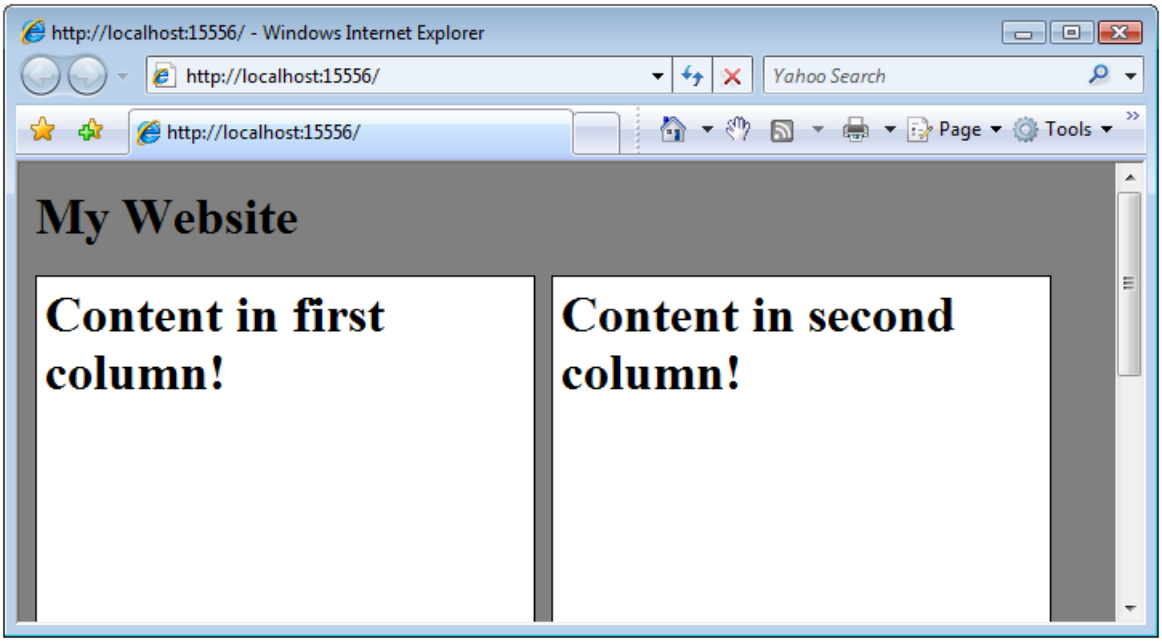


Figure 05: The Index view content page (Click to view full-size image)

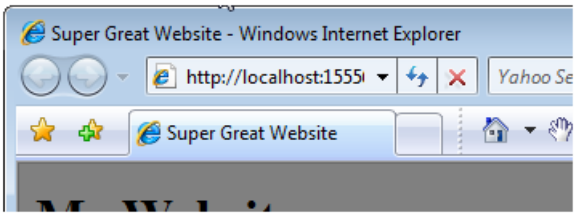
Modifying View Master Page Content

One issue that you encounter almost immediately when working with view master pages is the problem of modifying view master page content when different view content pages are requested. For example, you want each page in your web application to have a unique title. However, the title is declared in the view master page and not in the view content page. So, how do you customize the page title for each view content page?

There are two ways that you can modify the title displayed by a view content page. First, you can assign a page title to the title attribute of the `<%@ page %>` directive declared at the top of a view content page. For example, if you want to assign the page title "Super Great Website" to the Index view, then you can include the following directive at the top of the Index view:

```
<%@ page title="Super Great Website" language="VB" masterpagefile="~/Views/Shared/Site.Master"
            autoeventwireup="true" codebehind="Index.aspx.vb" inherits="MvcApplication1.Views.Home.Index"%>
```

When the Index view is rendered to the browser, the desired title appears in the browser title bar:



There is one important requirement that a master view page must satisfy in order for the title attribute to work. The view master page must contain a `<head runat="server">` tag instead of a normal `<head>` tag for its header. If the `<head>` tag does not include the runat="server" attribute then the title won't appear. The default view master page includes the required `<head runat="server">` tag.

An alternative approach to modifying master page content from an individual view content page is to wrap the region that you want to modify in a `<asp:ContentPlaceHolder>` tag. For example, imagine that you want to change not only the title, but also the meta tags, rendered by a master view page. The master view page in Listing 4 contains a `<asp:ContentPlaceHolder>` tag within its `<head>` tag.

Listing 4 – `Views\Shared\Site2.master`

```
<%@ Master Language="VB" AutoEventWireup="false" CodeBehind="Site2.Master.vb" Inherits="MvcApplication1.Site2"
%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>

     <asp:ContentPlaceHolder ID="head" runat="server">
          <title>Please change my title</title>
          <meta name="description" content="Please provide a description" />
          <meta name="keywords" content="keyword1,keyword2" />
     </asp:ContentPlaceHolder>
</head>
<body>
     <div>

          <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">

          </asp:ContentPlaceHolder>
     </div>
</body>
</html>
```

Notice that the `<asp:ContentPlaceHolder>` tag in Listing 4 includes default content: a default title and default meta tags. If you don't override this `<asp:ContentPlaceHolder>` tag in an individual view content page, then the default content will be displayed.

The content view page in Listing 5 overrides the `<asp:ContentPlaceHolder>` tag in order to display a custom title and custom meta tags.

Listing 5 – `Views\Home\Index2.aspx`

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site2.Master" AutoEventWireup="false"
CodeBehind="Index2.aspx.vb" Inherits="MvcApplication1.Index2" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
     <title>The Index2 Page</title>
     <meta name="description" content="Description of Index2 page" />
     <meta name="keywords" content="asp.net,mvc,cool,groovy" />
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">

     Just some content in the body of the page.

</asp:Content>
```

Summary

This tutorial provided you with a basic introduction to view master pages and view content pages. You learned how to create new view master pages and create view content pages based on them. We also examined how you can modify the content of a view master page from a particular view content page.

PREVIOUS NEXT

Passing Data to View Master Pages (VB)

4/10/2020 • 7 minutes to read • Edit Online

by Microsoft

Download PDF

The goal of this tutorial is to explain how you can pass data from a controller to a view master page. We examine two strategies for passing data to a view master page. First, we discuss an easy solution that results in an application that is difficult to maintain. Next, we examine a much better solution that requires a little more initial work but results in a much more maintainable application.

Passing Data to View Master Pages

The goal of this tutorial is to explain how you can pass data from a controller to a view master page. We examine two strategies for passing data to a view master page. First, we discuss an easy solution that results in an application that is difficult to maintain. Next, we examine a much better solution that requires a little more initial work but results in a much more maintainable application.

The Problem

Imagine that you are building a movie database application and you want to display the list of movie categories on every page in your application (see Figure 1). Imagine, furthermore, that the list of movie categories is stored in a database table. In that case, it would make sense to retrieve the categories from the database and render the list of movie categories within a view master page.

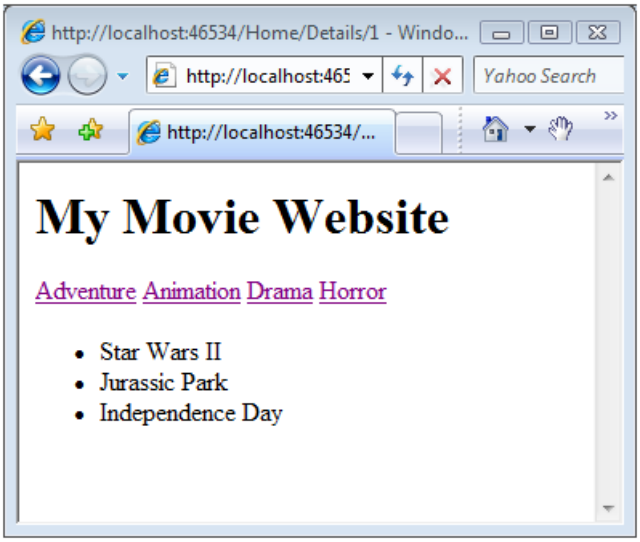


Figure 01: Displaying movie categories in a view master page (Click to view full-size image)

Here's the problem. How do you retrieve the list of movie categories in the master page? It is tempting to call methods of your model classes in the master page directly. In other words, it is tempting to include the code for retrieving the data from the database right in your master page. However, bypassing your MVC controllers to access the database would violate the clean separation of concerns that is one of the primary benefits of building an MVC application.

In an MVC application, you want all interaction between your MVC views and your MVC model to be handled by your MVC controllers. This separation of concerns results in a more maintainable, adaptable, and testable application.

In an MVC application, all data passed to a view – including a view master page – should be passed to a view by a controller action. Furthermore, the data should be passed by taking advantage of view data. In the remainder of this tutorial, I examine two methods of passing view data to a view master page.

The Simple Solution

Let's start with the simplest solution to passing view data from a controller to a view master page. The simplest solution is to pass the view data for the master page in each and every controller action.

Consider the controller in Listing 1. It exposes two actions named `Index()` and `Details()`. The `Index()` action method returns every movie in the Movies database table. The `Details()` action method returns every movie in a particular movie category.

Listing 1 – `Controllers\HomeController.vb`

```
<HandleError()> _
Public Class HomeController
     Inherits System.Web.Mvc.Controller

     Private _dataContext As New MovieDataContext()

     ''' <summary>

     ''' Show list of all movies
     ''' </summary>
     Function Index()
          ViewData("categories") = From c In _dataContext.MovieCategories _
                    Select c
          ViewData("movies") = From m In _dataContext.Movies _
                    Select m
          Return View()
     End Function

     ''' <summary>
     ''' Show list of movies in a category
     ''' </summary>

     Function Details(ByVal id As Integer)
          ViewData("categories") = From c In _dataContext.MovieCategories _
                    Select c
          ViewData("movies") = From m In _dataContext.Movies _
                    Where m.CategoryId = id _
                    Select m
          Return View()
     End Function

End Class
```

Notice that both the `Index()` and the `Details()` actions add two items to view data. The `Index()` action adds two keys: categories and movies. The categories key represents the list of movie categories displayed by the view master page. The movies key represents the list of movies displayed by the Index view page.

The `Details()` action also adds two keys named categories and movies. The categories key, once again, represents the list of movie categories displayed by the view master page. The movies key represents the list of movies in a particular category displayed by the Details view page (see Figure 2).

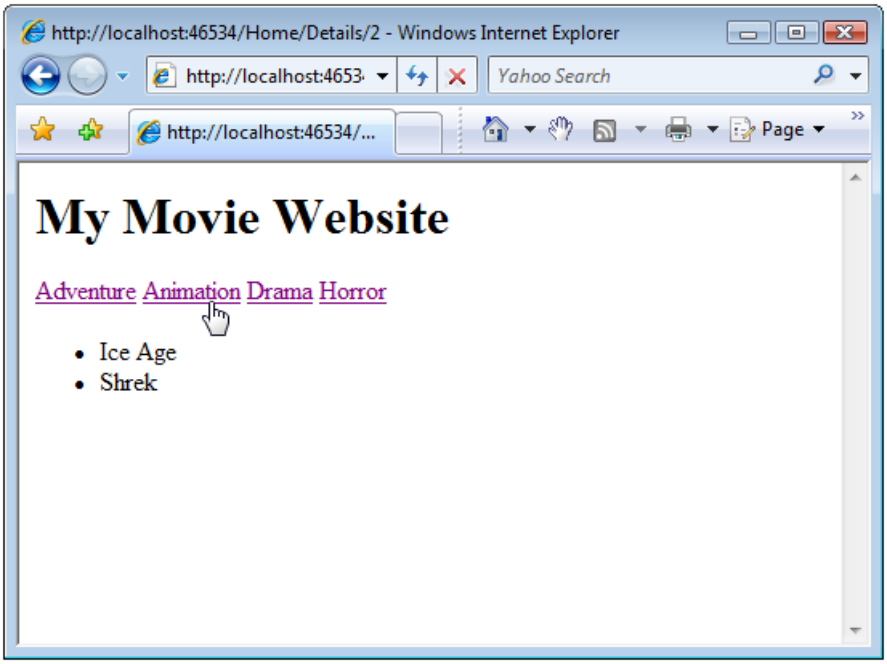


Figure 02: The Details view (Click to view full-size image)

The Index view is contained in Listing 2. It simply iterates through the list of movies represented by the movies item in view data.

Listing 2 – `Views\Home\Index.aspx`

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="false"
CodeBehind="Index.aspx.vb" Inherits="MvcApplication1.Index" %>

<%@ Import Namespace="MvcApplication1" %>
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1" runat="server">

<ul>

<%  For Each m In ViewData("movies")%>

     <li><%= m.Title %></li>

<% Next%>
</ul>

</asp:Content>
```

The view master page is contained in Listing 3. The view master page iterates and renders all of the movie categories represented by the categories item from view data.

Listing 3 – `Views\Shared\Site.master`

```
<%@ Master Language="VB" AutoEventWireup="false" CodeBehind="Site.Master.vb" Inherits="MvcApplication1.Site"
%>

<%@ Import Namespace="MvcApplication1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">

     <title></title>
     <asp:ContentPlaceHolder ID="head" runat="server">
     </asp:ContentPlaceHolder>
</head>
<body>
     <div>

          <h1>My Movie Website</h1>

          <% For Each c In ViewData("categories")%>

               <%=Html.ActionLink(c.Name, "Details", New With {.id = c.Id})%>

          <% Next%>


          <asp:ContentPlaceHolder ID="ContentPlaceHolder1" runat="server">

          </asp:ContentPlaceHolder>
     </div>
</body>
</html>
```

All data is passed to the view and the view master page through view data. That is the correct way to pass data to the master page.

So, what's wrong with this solution? The problem is that this solution violates the DRY (Don't Repeat Yourself) principle. Each and every controller action must add the very same list of movie categories to view data. Having duplicate code in your application makes your application much more difficult to maintain, adapt, and modify.

The Good Solution

In this section, we examine an alternative, and better, solution to passing data from a controller action to a view master page. Instead of adding the movie categories for the master page in each and every controller action, we add the movie categories to the view data only once. All view data used by the view master page is added in an Application controller.

The ApplicationController class is contained in Listing 4.

The ApplicationController class is contained in Listing 4.

Listing 4 – `Controllers\ApplicationController.vb`

```
Public MustInherit Class ApplicationController
     Inherits System.Web.Mvc.Controller

     Private _dataContext As New MovieDataContext()

     Public ReadOnly Property DataContext() As MovieDataContext
          Get

               Return _dataContext
          End Get
     End Property


     Sub New()
          ViewData("categories") = From c In DataContext.MovieCategories _
                    Select c
     End Sub

End Class
```

There are three things that you should notice about the Application controller in Listing 4. First, notice that the class inherits from the base System.Web.Mvc.Controller class. The Application controller is a controller class.

Second, notice that the Application controller class is a MustInherit class. An MustInherit class is a class that must be implemented by a concrete class. Because the Application controller is an MustInherit class, you cannot not invoke any methods defined in the class directly. If you attempt to invoke the Application class directly then you'll get a Resource Cannot Be Found error message.

Third, notice that the Application controller contains a constructor that adds the list of movie categories to view data. Every controller class that inherits from the Application controller calls the Application controller's constructor automatically. Whenever you call any action on any controller that inherits from the Application controller, the movie categories is included in the view data automatically.

The Movies controller in Listing 5 inherits from the Application controller.

Listing 5 – `Controllers\MoviesController.vb`

```
<HandleError()> _
Public Class MoviesController
     Inherits ApplicationController

     ''' <summary>
     ''' Show list of all movies
     ''' </summary>
     Function Index()
          ViewData("movies") = From m In DataContext.Movies _
                    Select m
          Return View()

      End Function

     ''' <summary>
     ''' Show list of movies in a category
     ''' </summary>
     Function Details(ByVal id As Integer)
          ViewData("movies") = From m In DataContext.Movies _
                    Where m.CategoryId = id _
                    Select m
          Return View()
     End Function

End Class
```

The Movies controller, just like the Home controller discussed in the previous section, exposes two action methods named `Index()` and `Details()`. Notice that the list of movie categories displayed by the view master page is not added to view data in either the `Index()` or `Details()` method. Because the Movies controller inherits from the Application controller, the list of movie categories is added to view data automatically.

Notice that this solution to adding view data for a view master page does not violate the DRY (Don't Repeat Yourself) principle. The code for adding the list of movie categories to view data is contained in only one location: the constructor for the Application controller.

Summary

In this tutorial, we discussed two approaches to passing view data from a controller to a view master page. First, we examined a simple, but difficult to maintain approach. In the first section, we discussed how you can add view data for a view master page in each every controller action in your application. We concluded that this was a bad approach because it violates the DRY (Don't Repeat Yourself) principle.

Next, we examined a much better strategy for adding data required by a view master page to view data. Instead of adding the view data in each and every controller action, we added the view data only once within an Application controller. That way, you can avoid duplicate code when passing data to a view master page in an ASP.NET MVC application.

PREVIOUS

Controllers and Routing

2/20/2020 • 2 minutes to read • Edit Online

In this tutorial set, you learn about ASP.NET routing, which maps browser requests to ASP.NET MVC controller actions.

- ASP.NET MVC Routing Overview (C#)
- Understanding Action Filters (C#)
- Improving Performance with Output Caching (C#)
- Adding Dynamic Content to a Cached Page (C#)
- Creating a Controller (C#)
- Creating an Action (C#)
- ASP.NET MVC Routing Overview (VB)
- Understanding Action Filters (VB)
- Improving Performance with Output Caching (VB)
- Adding Dynamic Content to a Cached Page (VB)
- Creating a Controller (VB)
- Creating an Action (VB)
- ASP.NET MVC Controller Overview (C#)
- Creating Custom Routes (C#)
- Creating a Route Constraint (C#)
- Creating a Custom Route Constraint (C#)
- ASP.NET MVC Controller Overview (VB)
- Creating Custom Routes (VB)
- Creating a Route Constraint (VB)
- Creating a Custom Route Constraint (VB)

ASP.NET MVC Routing Overview (C#)

2/20/2020 • 4 minutes to read • Edit Online

by Stephen Walther

> In this tutorial, Stephen Walther shows how the ASP.NET MVC framework maps browser requests to controller actions.

In this tutorial, you are introduced to an important feature of every ASP.NET MVC application called *ASP.NET Routing*. The ASP.NET Routing module is responsible for mapping incoming browser requests to particular MVC controller actions. By the end of this tutorial, you will understand how the standard route table maps requests to controller actions.

Using the Default Route Table

When you create a new ASP.NET MVC application, the application is already configured to use ASP.NET Routing. ASP.NET Routing is setup in two places.

First, ASP.NET Routing is enabled in your application's Web configuration file (Web.config file). There are four sections in the configuration file that are relevant to routing: the system.web.httpModules section, the system.web.httpHandlers section, the system.webserver.modules section, and the system.webserver.handlers section. Be careful not to delete these sections because without these sections routing will no longer work.

Second, and more importantly, a route table is created in the application's Global.asax file. The Global.asax file is a special file that contains event handlers for ASP.NET application lifecycle events. The route table is created during the Application Start event.

The file in Listing 1 contains the default Global.asax file for an ASP.NET MVC application.

Listing 1 - Global.asax.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcApplication1
{
    // Note: For instructions on enabling IIS6 or IIS7 classic mode,
    // visit https://go.microsoft.com/?LinkId=9394801

    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default",                                          // Route name
                "{controller}/{action}/{id}",                       // URL with parameters
                new { controller = "Home", action = "Index", id = "" }  // Parameter defaults
            );
        }

        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

When an MVC application first starts, the Application_Start() method is called. This method, in turn, calls the RegisterRoutes() method. The RegisterRoutes() method creates the route table.

The default route table contains a single route (named Default). The Default route maps the first segment of a URL to a controller name, the second segment of a URL to a controller action, and the third segment to a parameter named **id**.

Imagine that you enter the following URL into your web browser's address bar:

/Home/Index/3

The Default route maps this URL to the following parameters:

- controller = Home
- action = Index
- id = 3

When you request the URL /Home/Index/3, the following code is executed:

HomeController.Index(3)

The Default route includes defaults for all three parameters. If you don't supply a controller, then the controller parameter defaults to the value **Home**. If you don't supply an action, the action parameter defaults to the value **Index**. Finally, if you don't supply an id, the id parameter defaults to an empty string.

Let's look at a few examples of how the Default route maps URLs to controller actions. Imagine that you enter the following URL into your browser address bar:

/Home

Because of the Default route parameter defaults, entering this URL will cause the Index() method of the HomeController class in Listing 2 to be called.

Listing 2 - HomeController.cs

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index(string id)
        {
            return View();
        }
    }
}
```

In Listing 2, the HomeController class includes a method named Index() that accepts a single parameter named Id. The URL /Home causes the Index() method to be called with an empty string as the value of the Id parameter.

Because of the way that the MVC framework invokes controller actions, the URL /Home also matches the Index() method of the HomeController class in Listing 3.

Listing 3 - HomeController.cs (Index action with no parameter)

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

The Index() method in Listing 3 does not accept any parameters. The URL /Home will cause this Index() method to be called. The URL /Home/Index/3 also invokes this method (the Id is ignored).

The URL /Home also matches the Index() method of the HomeController class in Listing 4.

Listing 4 - HomeController.cs (Index action with nullable parameter)

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index(int? id)
        {
            return View();
        }
    }
}
```

In Listing 4, the Index() method has one Integer parameter. Because the parameter is a nullable parameter (can have the value Null), the Index() can be called without raising an error.

Finally, invoking the Index() method in Listing 5 with the URL /Home causes an exception since the Id parameter *is not* a nullable parameter. If you attempt to invoke the Index() method then you get the error displayed in Figure 1.

Listing 5 - HomeController.cs (Index action with Id parameter)

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index(int id)
        {
            return View();
        }
    }
}
```

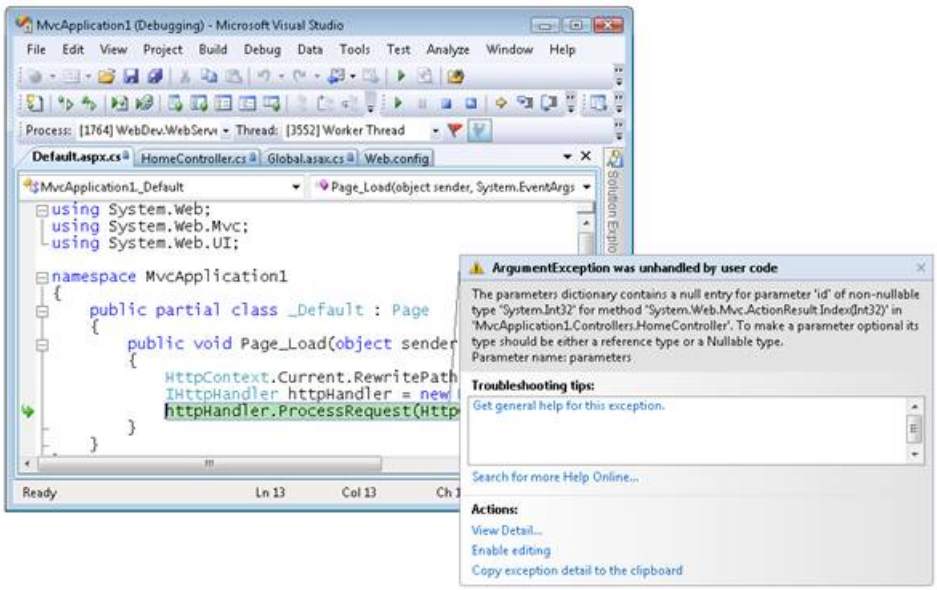


Figure 01: Invoking a controller action that expects a parameter value (Click to view full-size image)

The URL /Home/Index/3, on the other hand, works just fine with the Index controller action in Listing 5. The request /Home/Index/3 causes the Index() method to be called with an Id parameter that has the value 3.

Summary

The goal of this tutorial was to provide you with a brief introduction to ASP.NET Routing. We examined the default route table that you get with a new ASP.NET MVC application. You learned how the default route maps URLs to controller actions.

NEXT

Understanding Action Filters (C#)

4/10/2020 • 6 minutes to read • Edit Online

by Microsoft

Download PDF

> The goal of this tutorial is to explain action filters. An action filter is an attribute that you can apply to a controller action -- or an entire controller -- that modifies the way in which the action is executed.

Understanding Action Filters

The goal of this tutorial is to explain action filters. An action filter is an attribute that you can apply to a controller action -- or an entire controller -- that modifies the way in which the action is executed. The ASP.NET MVC framework includes several action filters:

- OutputCache – This action filter caches the output of a controller action for a specified amount of time.
- HandleError – This action filter handles errors raised when a controller action executes.
- Authorize – This action filter enables you to restrict access to a particular user or role.

You also can create your own custom action filters. For example, you might want to create a custom action filter in order to implement a custom authentication system. Or, you might want to create an action filter that modifies the view data returned by a controller action.

In this tutorial, you learn how to build an action filter from the ground up. We create a Log action filter that logs different stages of the processing of an action to the Visual Studio Output window.

Using an Action Filter

An action filter is an attribute. You can apply most action filters to either an individual controller action or an entire controller.

For example, the Data controller in Listing 1 exposes an action named `Index()` that returns the current time. This action is decorated with the `OutputCache` action filter. This filter causes the value returned by the action to be cached for 10 seconds.

Listing 1 – `Controllers\DataController.cs`

```
using System;
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
     public class DataController : Controller
     {
          [OutputCache(Duration=10)]
          public string Index()
          {
               return DateTime.Now.ToString("T");
          }
     }
}
```

If you repeatedly invoke the `Index()` action by entering the URL /Data/Index into the address bar of your browser and hitting the Refresh button multiple times, then you will see the same time for 10 seconds. The output of the

`Index()` action is cached for 10 seconds (see Figure 1).

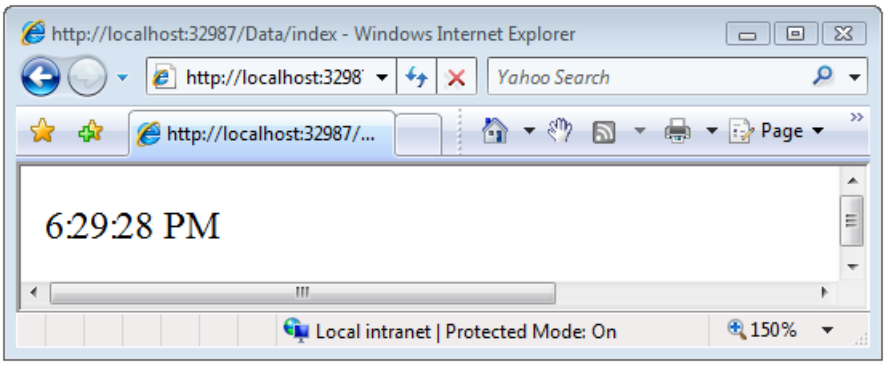


Figure 01: Cached time (Click to view full-size image)

In Listing 1, a single action filter – the `OutputCache` action filter – is applied to the `Index()` method. If you need, you can apply multiple action filters to the same action. For example, you might want to apply both the `OutputCache` and `HandleError` action filters to the same action.

In Listing 1, the `OutputCache` action filter is applied to the `Index()` action. You also could apply this attribute to the `DataController` class itself. In that case, the result returned by any action exposed by the controller would be cached for 10 seconds.

The Different Types of Filters

The ASP.NET MVC framework supports four different types of filters:

1. Authorization filters – Implements the `IAuthorizationFilter` attribute.
2. Action filters – Implements the `IActionFilter` attribute.
3. Result filters – Implements the `IResultFilter` attribute.
4. Exception filters – Implements the `IExceptionFilter` attribute.

Filters are executed in the order listed above. For example, authorization filters are always executed before action filters and exception filters are always executed after every other type of filter.

Authorization filters are used to implement authentication and authorization for controller actions. For example, the Authorize filter is an example of an Authorization filter.

Action filters contain logic that is executed before and after a controller action executes. You can use an action filter, for instance, to modify the view data that a controller action returns.

Result filters contain logic that is executed before and after a view result is executed. For example, you might want to modify a view result right before the view is rendered to the browser.

Exception filters are the last type of filter to run. You can use an exception filter to handle errors raised by either your controller actions or controller action results. You also can use exception filters to log errors.

Each different type of filter is executed in a particular order. If you want to control the order in which filters of the same type are executed then you can set a filter's Order property.

The base class for all action filters is the `System.Web.Mvc.FilterAttribute` class. If you want to implement a particular type of filter, then you need to create a class that inherits from the base Filter class and implements one or more of the `IAuthorizationFilter`, `IActionFilter`, `IResultFilter`, or `IExceptionFilter` interfaces.

The Base ActionFilterAttribute Class

In order to make it easier for you to implement a custom action filter, the ASP.NET MVC framework includes a base `ActionFilterAttribute` class. This class implements both the `IActionFilter` and `IResultFilter` interfaces and inherits from the `Filter` class.

The terminology here is not entirely consistent. Technically, a class that inherits from the ActionFilterAttribute class is both an action filter and a result filter. However, in the loose sense, the word action filter is used to refer to any type of filter in the ASP.NET MVC framework.

The base `ActionFilterAttribute` class has the following methods that you can override:

- OnActionExecuting – This method is called before a controller action is executed.
- OnActionExecuted – This method is called after a controller action is executed.
- OnResultExecuting – This method is called before a controller action result is executed.
- OnResultExecuted – This method is called after a controller action result is executed.

In the next section, we'll see how you can implement each of these different methods.

Creating a Log Action Filter

In order to illustrate how you can build a custom action filter, we'll create a custom action filter that logs the stages of processing a controller action to the Visual Studio Output window. Our `LogActionFilter` is contained in Listing 2.

Listing 2 – `ActionFilters\LogActionFilter.cs`

```
using System;
using System.Diagnostics;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcApplication1.ActionFilters
{
     public class LogActionFilter : ActionFilterAttribute

     {
          public override void OnActionExecuting(ActionExecutingContext filterContext)
          {
               Log("OnActionExecuting", filterContext.RouteData);
          }

          public override void OnActionExecuted(ActionExecutedContext filterContext)
          {
               Log("OnActionExecuted", filterContext.RouteData);
          }

          public override void OnResultExecuting(ResultExecutingContext filterContext)
          {
               Log("OnResultExecuting", filterContext.RouteData);
          }

          public override void OnResultExecuted(ResultExecutedContext filterContext)
          {
               Log("OnResultExecuted", filterContext.RouteData);
          }


          private void Log(string methodName, RouteData routeData)
          {
               var controllerName = routeData.Values["controller"];
               var actionName = routeData.Values["action"];
               var message = String.Format("{0} controller:{1} action:{2}", methodName, controllerName,
actionName);
               Debug.WriteLine(message, "Action Filter Log");
          }

     }
}
```

In Listing 2, the `OnActionExecuting()`, `OnActionExecuted()`, `OnResultExecuting()`, and `OnResultExecuted()` methods all call the `Log()` method. The name of the method and the current route data is passed to the `Log()` method. The `Log()` method writes a message to the Visual Studio Output window (see Figure 2).

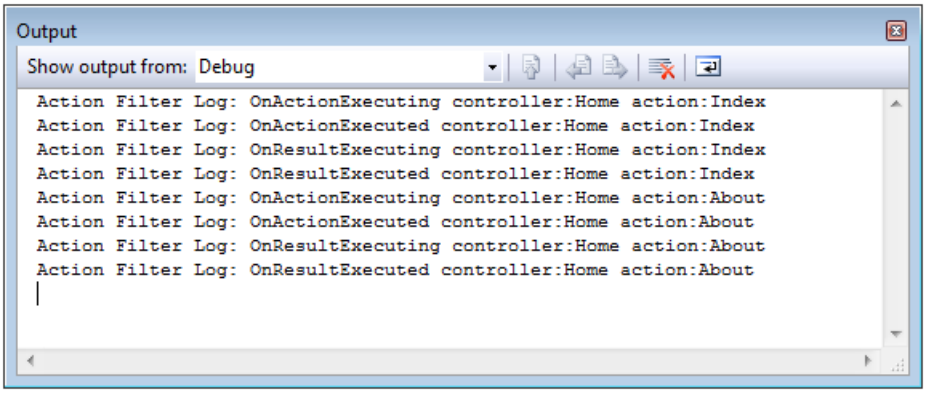


Figure 02: Writing to the Visual Studio Output window (Click to view full-size image)

The Home controller in Listing 3 illustrates how you can apply the Log action filter to an entire controller class. Whenever any of the actions exposed by the Home controller are invoked – either the `Index()` method or the `About()` method – the stages of processing the action are logged to the Visual Studio Output window.

Listing 3 – `Controllers\HomeController.cs`

```
using System.Web.Mvc;
using MvcApplication1.ActionFilters;

namespace MvcApplication1.Controllers
{
    [LogActionFilter]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult About()
        {
            return View();
        }
    }
}
```

Summary

In this tutorial, you were introduced to ASP.NET MVC action filters. You learned about the four different types of filters: authorization filters, action filters, result filters, and exception filters. You also learned about the base `ActionFilterAttribute` class.

Finally, you learned how to implement a simple action filter. We created a Log action filter that logs the stages of processing a controller action to the Visual Studio Output window.

PREVIOUS NEXT

Improving Performance with Output Caching (C#)

4/10/2020 • 9 minutes to read • Edit Online

by Microsoft

> In this tutorial, you learn how you can dramatically improve the performance of your ASP.NET MVC web applications by taking advantage of output caching. You learn how to cache the result returned from a controller action so that the same content does not need to be created each and every time a new user invokes the action.

The goal of this tutorial is to explain how you can dramatically improve the performance of an ASP.NET MVC application by taking advantage of the output cache. The output cache enables you to cache the content returned by a controller action. That way, the same content does not need to be generated each and every time the same controller action is invoked.

Imagine, for example, that your ASP.NET MVC application displays a list of database records in a view named Index. Normally, each and every time that a user invokes the controller action that returns the Index view, the set of database records must be retrieved from the database by executing a database query.

If, on the other hand, you take advantage of the output cache then you can avoid executing a database query every time any user invokes the same controller action. The view can be retrieved from the cache instead of being regenerated from the controller action. Caching enables you to avoid performing redundant work on the server.

Enabling Output Caching

You enable output caching by adding an [OutputCache] attribute to either an individual controller action or an entire controller class. For example, the controller in Listing 1 exposes an action named Index(). The output of the Index() action is cached for 10 seconds.

Listing 1 – Controllers\HomeController.cs

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        [OutputCache(Duration=10, VaryByParam="none")]
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

In the Beta versions of ASP.NET MVC, output caching does not work for a URL like http://www.MySite.com/. Instead, you must enter a URL like http://www.MySite.com/Home/Index.

In Listing 1, the output of the Index() action is cached for 10 seconds. If you prefer, you can specify a much longer cache duration. For example, if you want to cache the output of a controller action for one day then you can specify a cache duration of 86400 seconds (60 seconds * 60 minutes * 24 hours).

There is no guarantee that content will be cached for the amount of time that you specify. When memory resources become low, the cache starts evicting content automatically.

The Home controller in Listing 1 returns the Index view in Listing 2. There is nothing special about this view. The Index view simply displays the current time (see Figure 1).

Listing 2 – Views\Home\Index.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Index.aspx.cs"
Inherits="MvcApplication1.Views.Home.Index" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Index</title>
</head>
<body>
    <div>

    The current time is: <%= DateTime.Now.ToString("T") %>


    </div>
</body>
</html>
```

Figure 1 – Cached Index view

If you invoke the Index() action multiple times by entering the URL /Home/Index in the address bar of your browser and hitting the Refresh/Reload button in your browser repeatedly, then the time displayed by the Index view won't change for 10 seconds. The same time is displayed because the view is cached.

It is important to understand that the same view is cached for everyone who visits your application. Anyone who invokes the Index() action will get the same cached version of the Index view. This means that the amount of work that the web server must perform to serve the Index view is dramatically reduced.

The view in Listing 2 happens to be doing something really simple. The view just displays the current time. However, you could just as easily cache a view that displays a set of database records. In that case, the set of database records would not need to be retrieved from the database each and every time the controller action that returns the view is invoked. Caching can reduce the amount of work that both your web server and database server must perform.

Don't use the page <%@ OutputCache %> directive in an MVC view. This directive is bleeding over from the Web Forms world and should not be used in an ASP.NET MVC application.

Where Content is Cached

By default, when you use the [OutputCache] attribute, content is cached in three locations: the web server, any

proxy servers, and the web browser. You can control exactly where content is cached by modifying the Location property of the [OutputCache] attribute.

You can set the Location property to any one of the following values:

- Any
- Client
- Downstream
- Server
- None
- ServerAndClient

By default, the Location property has the value Any. However, there are situations in which you might want to cache only on the browser or only on the server. For example, if you are caching information that is personalized for each user then you should not cache the information on the server. If you are displaying different information to different users then you should cache the information only on the client.

For example, the controller in Listing 3 exposes an action named GetName() that returns the current user name. If Jack logs into the website and invokes the GetName() action then the action returns the string "Hi Jack". If, subsequently, Jill logs into the website and invokes the GetName() action then she also will get the string "Hi Jack". The string is cached on the web server for all users after Jack initially invokes the controller action.

Listing 3 – Controllers\BadUserController.cs

```
using System.Web.Mvc;
using System.Web.UI;

namespace MvcApplication1.Controllers
{
    public class BadUserController : Controller
    {
        [OutputCache(Duration = 3600, VaryByParam = "none")]
        public string GetName()
        {
            return "Hi " + User.Identity.Name;
        }
    }
}
```

Most likely, the controller in Listing 3 does not work the way that you want. You don't want to display the message "Hi Jack" to Jill.

You should never cache personalized content in the server cache. However, you might want to cache the personalized content in the browser cache to improve performance. If you cache content in the browser, and a user invokes the same controller action multiple times, then the content can be retrieved from the browser cache instead of the server.

The modified controller in Listing 4 caches the output of the GetName() action. However, the content is cached only on the browser and not on the server. That way, when multiple users invoke the GetName() method, each person gets their own user name and not another person's user name.

Listing 4 – Controllers\UserController.cs

```
using System.Web.Mvc;
using System.Web.UI;

namespace MvcApplication1.Controllers
{
    public class UserController : Controller
    {
        [OutputCache(Duration=3600, VaryByParam="none", Location=OutputCacheLocation.Client, NoStore=true)]
        public string GetName()
        {
            return "Hi " + User.Identity.Name;
        }
    }
}
```

Notice that the [OutputCache] attribute in Listing 4 includes a Location property set to the value OutputCacheLocation.Client. The [OutputCache] attribute also includes a NoStore property. The NoStore property is used to inform proxy servers and browser that they should not store a permanent copy of the cached content.

Varying the Output Cache

In some situations, you might want different cached versions of the very same content. Imagine, for example, that you are creating a master/detail page. The master page displays a list of movie titles. When you click a title, you get details for the selected movie.

If you cache the details page, then the details for the same movie will be displayed no matter which movie you click. The first movie selected by the first user will be displayed to all future users.

You can fix this problem by taking advantage of the VaryByParam property of the [OutputCache] attribute. This property enables you to create different cached versions of the very same content when a form parameter or query string parameter varies.

For example, the controller in Listing 5 exposes two actions named Master() and Details(). The Master() action returns a list of movie titles and the Details() action returns the details for the selected movie.

Listing 5 – Controllers\MoviesController.cs

```
using System.Linq;
using System.Web.Mvc;
using MvcApplication1.Models;

namespace MvcApplication1.Controllers
{
    public class MoviesController : Controller
    {
        private MovieDataContext _dataContext;

        public MoviesController()
        {
            _dataContext = new MovieDataContext();
        }

        [OutputCache(Duration=int.MaxValue, VaryByParam="none")]
        public ActionResult Master()
        {
            ViewData.Model = (from m in _dataContext.Movies
                              select m).ToList();
            return View();
        }

        [OutputCache(Duration = int.MaxValue, VaryByParam = "id")]
        public ActionResult Details(int id)
        {
            ViewData.Model = _dataContext.Movies.SingleOrDefault(m => m.Id == id);
            return View();
        }


    }
}
```

The Master() action includes a VaryByParam property with the value "none". When the Master() action is invoked, the same cached version of the Master view is returned. Any form parameters or query string parameters are ignored (see Figure 2).

Figure 2 – The /Movies/Master view

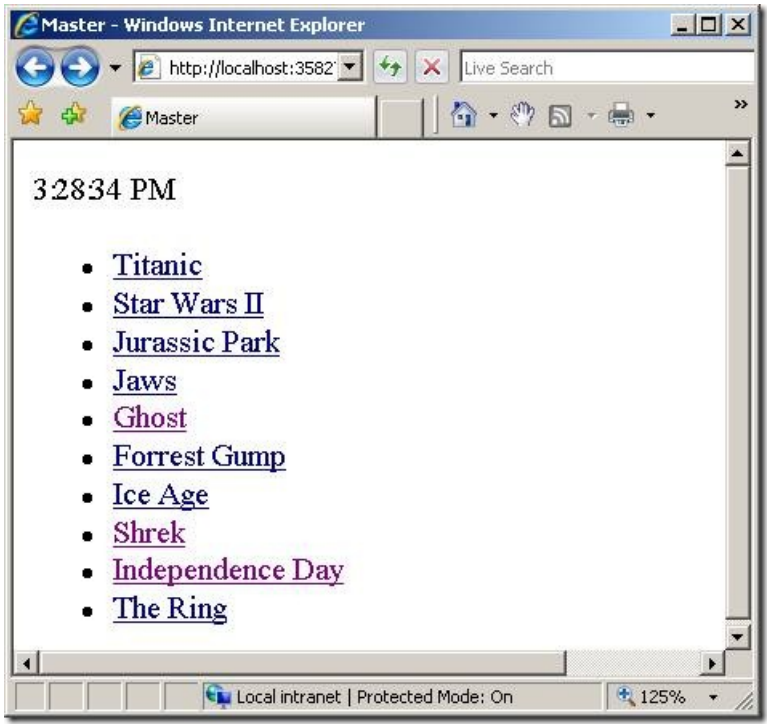
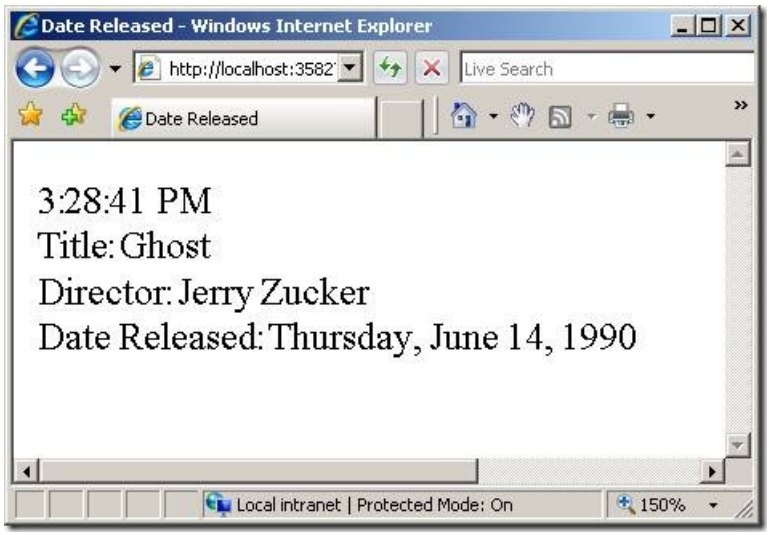


Figure 3 – The /Movies/Details view



The Details() action includes a VaryByParam property with the value "Id". When different values of the Id parameter are passed to the controller action, different cached versions of the Details view are generated.

It is important to understand that using the VaryByParam property results in more caching and not less. A different cached version of the Details view is created for each different version of the Id parameter.

You can set the VaryByParam property to the following values:

> * = Create a different cached version whenever a form or query string parameter varies.
>
> none = Never create different cached versions
>
> Semicolon list of parameters = Create different cached versions whenever any of the form or query string parameters in the list varies

Creating a Cache Profile

As an alternative to configuring output cache properties by modifying properties of the [OutputCache] attribute, you can create a cache profile in the web configuration (web.config) file. Creating a cache profile in the web configuration file offers a couple of important advantages.

First, by configuring output caching in the web configuration file, you can control how controller actions cache content in one central location. You can create one cache profile and apply the profile to several controllers or controller actions.

Second, you can modify the web configuration file without recompiling your application. If you need to disable caching for an application that has already been deployed to production, then you can simply modify the cache profiles defined in the web configuration file. Any changes to the web configuration file will be detected automatically and applied.

For example, the <caching> web configuration section in Listing 6 defines a cache profile named Cache1Hour. The <caching> section must appear within the <system.web> section of a web configuration file.

Listing 6 – Caching section for web.config

```
<caching>
<outputCacheSettings>
    <outputCacheProfiles>
        <add name="Cache1Hour" duration="3600" varyByParam="none"/>
    </outputCacheProfiles>
</outputCacheSettings>
</caching>
```

The controller in Listing 7 illustrates how you can apply the Cache1Hour profile to a controller action with the [OutputCache] attribute.

Listing 7 – Controllers\ProfileController.cs

```
using System;
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    public class ProfileController : Controller
    {
        [OutputCache(CacheProfile="Cache1Hour")]
        public string Index()
        {
            return DateTime.Now.ToString("T");
        }
    }
}
```

If you invoke the Index() action exposed by the controller in Listing 7 then the same time will be returned for 1 hour.

Summary

Output caching provides you with a very easy method of dramatically improving the performance of your ASP.NET MVC applications. In this tutorial, you learned how to use the [OutputCache] attribute to cache the output of controller actions. You also learned how to modify properties of the [OutputCache] attribute such as the Duration and VaryByParam properties to modify how content gets cached. Finally, you learned how to define cache profiles in the web configuration file.

PREVIOUS NEXT

Adding Dynamic Content to a Cached Page (C#)

4/10/2020 • 5 minutes to read • Edit Online

by Microsoft

Learn how to mix dynamic and cached content in the same page. Post-cache substitution enables you to display dynamic content, such as banner advertisements or news items, within a page that has been output cached.

By taking advantage of output caching, you can dramatically improve the performance of an ASP.NET MVC application. Instead of regenerating a page each and every time the page is requested, the page can be generated once and cached in memory for multiple users.

But there is a problem. What if you need to display dynamic content in the page? For example, imagine that you want to display a banner advertisement in the page. You don't want the banner advertisement to be cached so that every user sees the very same advertisement. You wouldn't make any money that way!

Fortunately, there is an easy solution. You can take advantage of a feature of the ASP.NET framework called *post-cache substitution*. Post-cache substitution enables you to substitute dynamic content in a page that has been cached in memory.

Normally, when you output cache a page by using the [OutputCache] attribute, the page is cached on both the server and the client (the web browser). When you use post-cache substitution, a page is cached only on the server.

Using Post-Cache Substitution

Using post-cache substitution requires two steps. First, you need to define a method that returns a string that represents the dynamic content that you want to display in the cached page. Next, you call the HttpResponse.WriteSubstitution() method to inject the dynamic content into the page.

Imagine, for example, that you want to randomly display different news items in a cached page. The class in Listing 1 exposes a single method, named RenderNews(), that randomly returns one news item from a list of three news items.

Listing 1 – Models\News.cs

```
using System;
using System.Collections.Generic;
using System.Web;

namespace MvcApplication1.Models
{
    public class News
    {
        public static string RenderNews(HttpContext context)
        {
            var news = new List<string>
                {
                    "Gas prices go up!",
                    "Life discovered on Mars!",
                    "Moon disappears!"
                };

            var rnd = new Random();
            return news[rnd.Next(news.Count)];
        }
    }
}
```

To take advantage of post-cache substitution, you call the HttpResponse.WriteSubstitution() method. The WriteSubstitution() method sets up the code to replace a region of the cached page with dynamic content. The WriteSubstitution() method is used to display the random news item in the view in Listing 2.

Listing 2 – Views\Home\Index.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Index.aspx.cs"
Inherits="MvcApplication1.Views.Home.Index" %>
<%@ Import Namespace="MvcApplication1.Models" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Index</title>
</head>
<body>
    <div>

    <% Response.WriteSubstitution(News.RenderNews); %>

    <hr />

    The content of this page is output cached.
    <%= DateTime.Now %>

    </div>
</body>
</html>
```

The RenderNews method is passed to the WriteSubstitution() method. Notice that the RenderNews method is not called (there are no parentheses). Instead a reference to the method is passed to WriteSubstitution().

The Index view is cached. The view is returned by the controller in Listing 3. Notice that the Index() action is decorated with an [OutputCache] attribute that causes the Index view to be cached for 60 seconds.

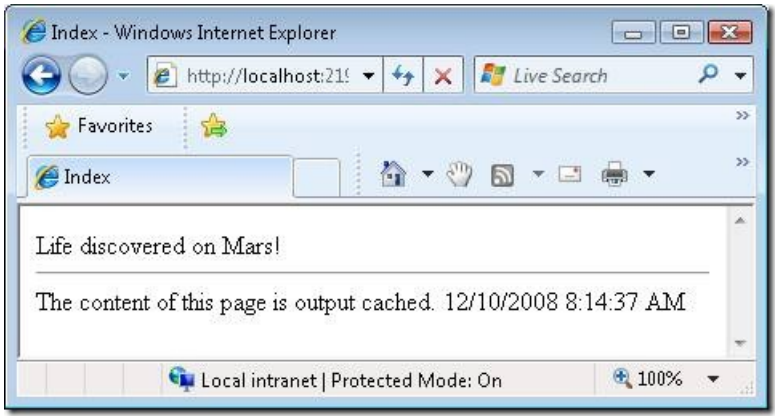
Listing 3 – Controllers\HomeController.cs

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        [OutputCache(Duration=60, VaryByParam="none")]
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Even though the Index view is cached, different random news items are displayed when you request the Index page. When you request the Index page, the time displayed by the page does not change for 60 seconds (see Figure 1). The fact that the time does not change proves that the page is cached. However, the content injected by the WriteSubstitution() method – the random news item – changes with each request .

Figure 1 – Injecting dynamic news items in a cached page



Using Post-Cache Substitution in Helper Methods

An easier way to take advantage of post-cache substitution is to encapsulate the call to the WriteSubstitution() method within a custom helper method. This approach is illustrated by the helper method in Listing 4.

Listing 4 – AdHelper.cs

```
using System;
using System.Collections.Generic;
using System.Web;
using System.Web.Mvc;

namespace MvcApplication1.Helpers
{
    public static class AdHelper
    {
        public static void RenderBanner(this HtmlHelper helper)
        {
            var context = helper.ViewContext.HttpContext;
            context.Response.WriteSubstitution(RenderBannerInternal);
        }

        private static string RenderBannerInternal(HttpContext context)
        {
            var ads = new List<string>
                {
                    "/ads/banner1.gif",
                    "/ads/banner2.gif",
                    "/ads/banner3.gif"
                };

            var rnd = new Random();
            var ad = ads[rnd.Next(ads.Count)];
            return String.Format("<img src='{0}' />", ad);
        }
    }
}
```

Listing 4 contains a static class that exposes two methods: RenderBanner() and RenderBannerInternal(). The RenderBanner() method represents the actual helper method. This method extends the standard ASP.NET MVC HtmlHelper class so that you can call Html.RenderBanner() in a view just like any other helper method.

The RenderBanner() method calls the HttpResponse.WriteSubstitution() method passing the RenderBannerInternal() method to the WriteSubstitution() method.

The RenderBannerInternal() method is a private method. This method won't be exposed as a helper method. The RenderBannerInternal() method randomly returns one banner advertisement image from a list of three banner advertisement images.

The modified Index view in Listing 5 illustrates how you can use the RenderBanner() helper method. Notice that an additional <%@ Import %> directive is included at the top of the view to import the MvcApplication1.Helpers namespace. If you neglect to import this namespace, then the RenderBanner() method won't appear as a method on the Html property.

Listing 5 – Views\Home\Index.aspx (with RenderBanner() method)

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Index.aspx.cs"
Inherits="MvcApplication1.Views.Home.Index" %>
<%@ Import Namespace="MvcApplication1.Models" %>
<%@ Import Namespace="MvcApplication1.Helpers" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Index</title>
</head>
<body>
    <div>

    <% Response.WriteSubstitution(News.RenderNews); %>

    <hr />

    <% Html.RenderBanner(); %>

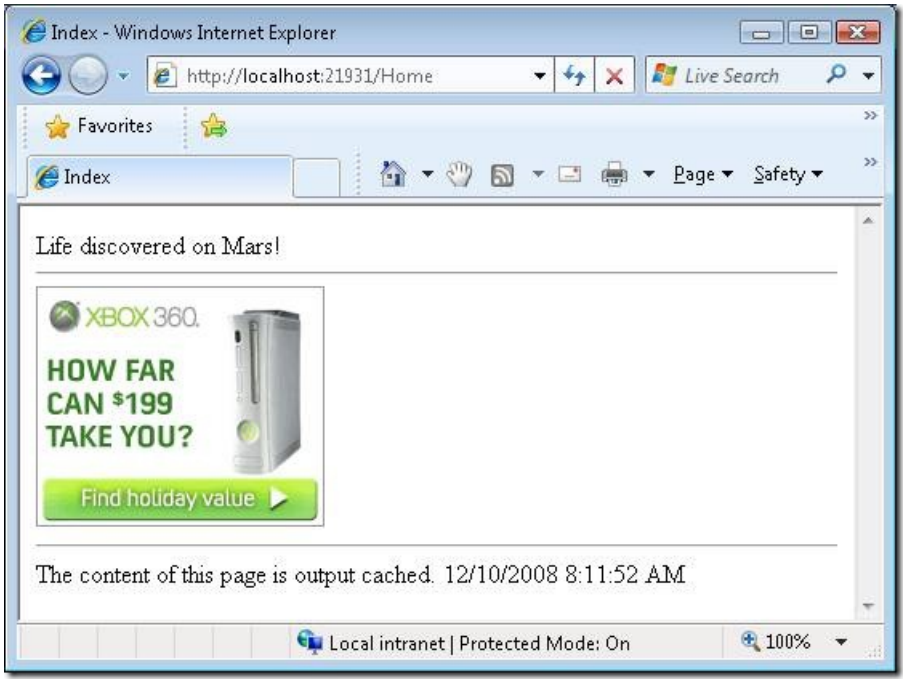
    <hr />

    The content of this page is output cached.
    <%= DateTime.Now %>

    </div>
</body>
</html>
```

When you request the page rendered by the view in Listing 5, a different banner advertisement is displayed with each request (see Figure 2). The page is cached, but the banner advertisement is injected dynamically by the RenderBanner() helper method.

Figure 2 – The Index view displaying a random banner advertisement



Summary

This tutorial explained how you can dynamically update content in a cached page. You learned how to use the HttpResponse.WriteSubstitution() method to enable dynamic content to be injected in a cached page. You also learned how to encapsulate the call to the WriteSubstitution() method within an HTML helper method.

Take advantage of caching whenever possible – it can have a dramatic impact on the performance of your web applications. As explained in this tutorial, you can take advantage of caching even when you need to display

dynamic content in your pages.

PREVIOUS NEXT

Creating a Controller (C#)

2/20/2020 • 3 minutes to read • Edit Online

by Stephen Walther

In this tutorial, Stephen Walther demonstrates how you can add a controller to an ASP.NET MVC application.

The goal of this tutorial is to explain how you can create new ASP.NET MVC controllers. You learn how to create controllers both by using the Visual Studio Add Controller menu option and by creating a class file by hand.

Using the Add Controller Menu Option

The easiest way to create a new controller is to right-click the Controllers folder in the Visual Studio Solution Explorer window and select the **Add, Controller** menu option (see Figure 1). Selecting this menu option opens the **Add Controller** dialog (see Figure 2).

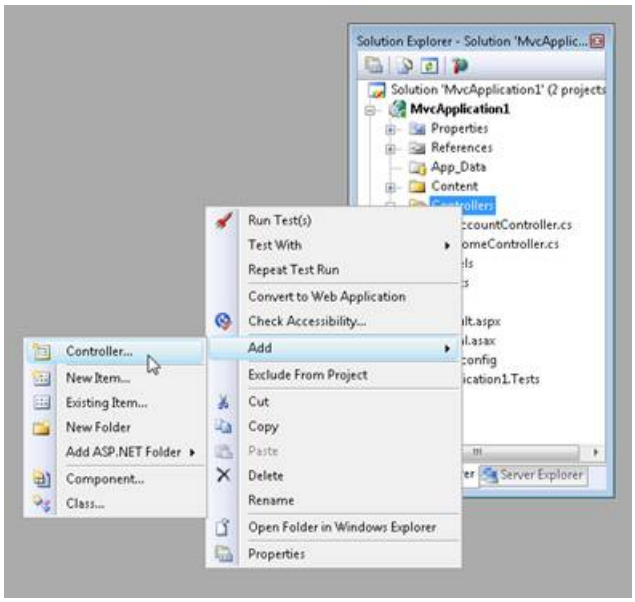


Figure 01: Adding a new controller(Click to view full-size image)

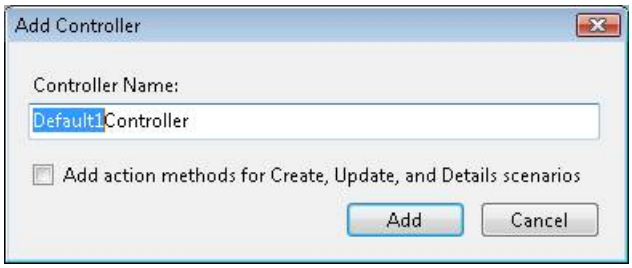


Figure 02: The Add Controller dialog (Click to view full-size image)

Notice that the first part of the controller name is highlighted in the **Add Controller** dialog. Every controller name must end with the suffix *Controller*. For example, you can create a controller named *ProductController* but not a controller named *Product*.

If you create a controller that is missing the *Controller* suffix then you won't be able to invoke the controller. Don't do this -- I've wasted countless hours of my life after making this mistake.

Listing 1 - Controllers\ProductController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Mvc.Ajax;

namespace MvcApplication1.Controllers
{
    public class ProductController : Controller
    {
        //
        // GET: /Product/

        public ActionResult Index()
        {
            return View();
        }
    }
}
```

You should always create controllers in the Controllers folder. Otherwise, you'll be violating the conventions of ASP.NET MVC and other developers will have a more difficult time understanding your application.

Scaffolding Action Methods

When you create a controller, you have the option to generate Create, Update, and Details action methods automatically (see Figure 3). If you select this option then the controller class in Listing 2 is generated.

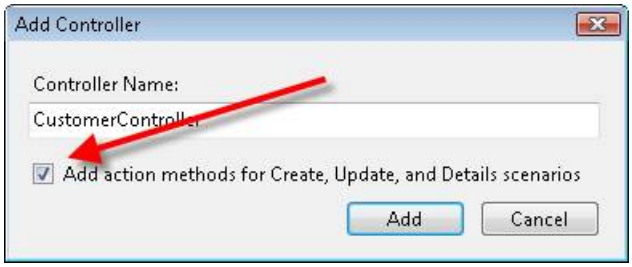


Figure 03: Creating action methods automatically (Click to view full-size image)

Listing 2 - Controllers\CustomerController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Mvc.Ajax;

namespace MvcApplication1.Controllers
{
    public class CustomerController : Controller
    {
        //
        // GET: /Customer/

        public ActionResult Index()
        {
            return View();
        }

        //
        // GET: /Customer/Details/5

        public ActionResult Details(int id)
```

```
        {
            return View();
        }

        //
        // GET: /Customer/Create

        public ActionResult Create()
        {
            return View();
        }

        //
        // POST: /Customer/Create

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create(FormCollection collection)
        {
            try
            {
                // TODO: Add insert logic here

                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }

        //
        // GET: /Customer/Edit/5

        public ActionResult Edit(int id)
        {
            return View();
        }

        //
        // POST: /Customer/Edit/5

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Edit(int id, FormCollection collection)
        {
            try
            {
                // TODO: Add update logic here

                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }
    }
}
```

These generated methods are stub methods. You must add the actual logic for creating, updating, and showing details for a customer yourself. But, the stub methods provide you with a nice starting point.

Creating a Controller Class

The ASP.NET MVC controller is just a class. If you prefer, you can ignore the convenient Visual Studio controller scaffolding and create a controller class by hand. Follow these steps:

1. Right-click the Controllers folder and select the menu option **Add, New Item** and select the **Class** template

(see Figure 4).

2. Name the new class PersonController.cs and click the **Add** button.
3. Modify the resulting class file so that the class inherits from the base System.Web.Mvc.Controller class (see Listing 3).

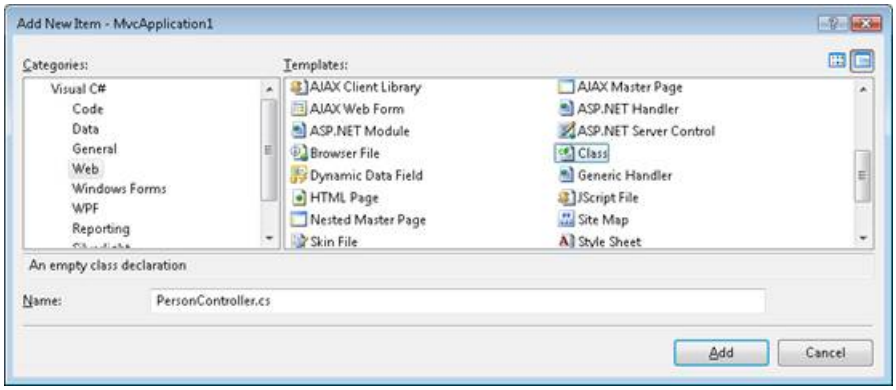


Figure 04: Creating a new class(Click to view full-size image)

Listing 3 - Controllers\PersonController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace MvcApplication1.Controllers
{
    public class PersonController : System.Web.Mvc.Controller
    {
        public string Index()
        {
            return "Hello World!";
        }
    }
}
```

The controller in Listing 3 exposes one action named Index() that returns the string "Hello World!". You can invoke this controller action by running your application and requesting a URL like the following:

```
http://localhost:40071/Person
```

NOTE

The ASP.NET Development Server uses a random port number (for example, 40071). When entering a URL to invoke a controller, you'll need to supply the right port number. You can determine the port number by hovering your mouse over the icon for the ASP.NET Development Server in the Windows Notification Area (bottom-right of your screen).

PREVIOUS NEXT

Creating an Action (C#)

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

> Learn how to add a new action to an ASP.NET MVC controller. Learn about the requirements for a method to be an action.

The goal of this tutorial is to explain how you can create a new controller action. You learn about the requirements of an action method. You also learn how to prevent a method from being exposed as an action.

Adding an Action to a Controller

You add a new action to a controller by adding a new method to the controller. For example, the controller in Listing 1 contains an action named Index() and an action named SayHello(). Both methods are exposed as actions.

Listing 1 - Controllers\HomeController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public string SayHello()
        {
            return "Hello!";
        }
    }
}
```

In order to be exposed to the universe as an action, a method must meet certain requirements:

- The method must be public.
- The method cannot be a static method.
- The method cannot be an extension method.
- The method cannot be a constructor, getter, or setter.
- The method cannot have open generic types.
- The method is not a method of the controller base class.
- The method cannot contain **ref** or **out** parameters.

Notice that there are no restrictions on the return type of a controller action. A controller action can return a string, a DateTime, an instance of the Random class, or void. The ASP.NET MVC framework will convert any return type that is not an action result into a string and render the string to the browser.

When you add any method that does not violate these requirements to a controller, the method is exposed as a controller action. Be careful here. A controller action can be invoked by anyone connected to the Internet. Do not, for example, create a DeleteMyWebsite() controller action.

Preventing a Public Method from Being Invoked

If you need to create a public method in a controller class and you don't want to expose the method as a controller action then you can prevent the method from being invoked by using the [NonAction] attribute. For example, the controller in Listing 2 contains a public method named CompanySecrets() that is decorated with the [NonAction] attribute.

Listing 2 - Controllers\WorkController.cs

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    public class WorkController : Controller
    {
        [NonAction]
        public string CompanySecrets()
        {
            return "This information is secret.";
        }
    }
}
```

If you attempt to invoke the CompanySecrets() controller action by typing /Work/CompanySecrets into the address bar of your browser then you'll get the error message in Figure 1.

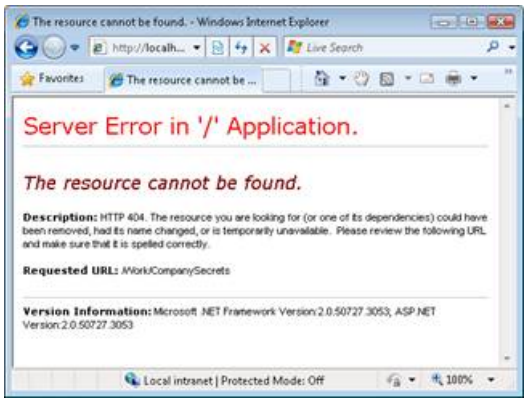


Figure 01: Invoking a NonAction method(Click to view full-size image)

PREVIOUS NEXT

ASP.NET MVC Routing Overview (VB)

2/20/2020 • 4 minutes to read • Edit Online

by Stephen Walther

> In this tutorial, Stephen Walther shows how the ASP.NET MVC framework maps browser requests to controller actions.

In this tutorial, you are introduced to an important feature of every ASP.NET MVC application called *ASP.NET Routing*. The ASP.NET Routing module is responsible for mapping incoming browser requests to particular MVC controller actions. By the end of this tutorial, you will understand how the standard route table maps requests to controller actions.

Using the Default Route Table

When you create a new ASP.NET MVC application, the application is already configured to use ASP.NET Routing. ASP.NET Routing is setup in two places.

First, ASP.NET Routing is enabled in your application's Web configuration file (Web.config file). There are four sections in the configuration file that are relevant to routing: the system.web.httpModules section, the system.web.httpHandlers section, the system.webserver.modules section, and the system.webserver.handlers section. Be careful not to delete these sections because without these sections routing will no longer work.

Second, and more importantly, a route table is created in the application's Global.asax file. The Global.asax file is a special file that contains event handlers for ASP.NET application lifecycle events. The route table is created during the Application Start event.

The file in Listing 1 contains the default Global.asax file for an ASP.NET MVC application.

Listing 1 - Global.asax.vb

```
' Note: For instructions on enabling IIS6 or IIS7 classic mode,
' visit https://go.microsoft.com/?LinkId=9394802

Public Class MvcApplication
    Inherits System.Web.HttpApplication

    Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}")

        ' MapRoute takes the following parameters, in order:
        ' (1) Route name
        ' (2) URL with parameters
        ' (3) Parameter defaults
        routes.MapRoute( _
            "Default", _
            "{controller}/{action}/{id}", _
            New With {.controller = "Home", .action = "Index", .id = ""} _
        )

    End Sub

    Sub Application_Start()
        RegisterRoutes(RouteTable.Routes)
    End Sub
End Class
```

When an MVC application first starts, the Application_Start() method is called. This method, in turn, calls the RegisterRoutes() method. The RegisterRoutes() method creates the route table.

The default route table contains a single route (named Default). The Default route maps the first segment of a URL to a controller name, the second segment of a URL to a controller action, and the third segment to a parameter named **id**.

Imagine that you enter the following URL into your web browser's address bar:

/Home/Index/3

The Default route maps this URL to the following parameters:

- controller = Home
- action = Index
- id = 3

When you request the URL /Home/Index/3, the following code is executed:

HomeController.Index(3)

The Default route includes defaults for all three parameters. If you don't supply a controller, then the controller parameter defaults to the value **Home**. If you don't supply an action, the action parameter defaults to the value **Index**. Finally, if you don't supply an id, the id parameter defaults to an empty string.

Let's look at a few examples of how the Default route maps URLs to controller actions. Imagine that you enter the following URL into your browser address bar:

/Home

Because of the Default route parameter defaults, entering this URL will cause the Index() method of the HomeController class in Listing 2 to be called.

Listing 2 - HomeController.vb

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Function Index(ByVal id As String)
        Return View()
    End Function

End Class
```

In Listing 2, the HomeController class includes a method named Index() that accepts a single parameter named Id. The URL /Home causes the Index() method to be called with the value Nothing as the value of the Id parameter.

Because of the way that the MVC framework invokes controller actions, the URL /Home also matches the Index() method of the HomeController class in Listing 3.

Listing 3 - HomeController.vb (Index action with no parameter)

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Function Index()
        Return View()
    End Function

End Class
```

The Index() method in Listing 3 does not accept any parameters. The URL /Home will cause this Index() method to be called. The URL /Home/Index/3 also invokes this method (the Id is ignored).

The URL /Home also matches the Index() method of the HomeController class in Listing 4.

Listing 4 - HomeController.vb (Index action with nullable parameter)

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Function Index(ByVal id? As Integer)
        Return View()
    End Function

End Class
```

In Listing 4, the Index() method has one Integer parameter. Because the parameter is a nullable parameter (can have the value Nothing), the Index() can be called without raising an error.

Finally, invoking the Index() method in Listing 5 with the URL /Home causes an exception since the Id parameter *is not* a nullable parameter. If you attempt to invoke the Index() method then you get the error displayed in Figure 1.

Listing 5 - HomeController.vb (Index action with Id parameter)

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Function Index(ByVal id As Integer)
        Return View()
    End Function

End Class
```

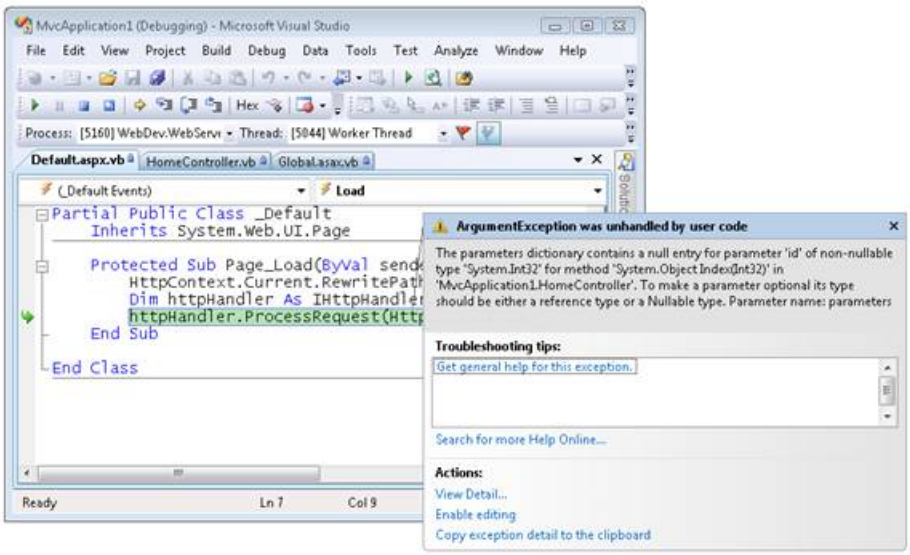


Figure 01: Invoking a controller action that expects a parameter value (Click to view full-size image)

The URL /Home/Index/3, on the other hand, works just fine with the Index controller action in Listing 5. The request /Home/Index/3 causes the Index() method to be called with an Id parameter that has the value 3.

Summary

The goal of this tutorial was to provide you with a brief introduction to ASP.NET Routing. We examined the default route table that you get with a new ASP.NET MVC application. You learned how the default route maps URLs to controller actions.

PREVIOUS NEXT

Understanding Action Filters (VB)

4/10/2020 • 6 minutes to read • Edit Online

by Microsoft

Download PDF

> The goal of this tutorial is to explain action filters. An action filter is an attribute that you can apply to a controller action -- or an entire controller -- that modifies the way in which the action is executed.

Understanding Action Filters

The goal of this tutorial is to explain action filters. An action filter is an attribute that you can apply to a controller action -- or an entire controller -- that modifies the way in which the action is executed. The ASP.NET MVC framework includes several action filters:

- OutputCache – This action filter caches the output of a controller action for a specified amount of time.
- HandleError – This action filter handles errors raised when a controller action executes.
- Authorize – This action filter enables you to restrict access to a particular user or role.

You also can create your own custom action filters. For example, you might want to create a custom action filter in order to implement a custom authentication system. Or, you might want to create an action filter that modifies the view data returned by a controller action.

In this tutorial, you learn how to build an action filter from the ground up. We create a Log action filter that logs different stages of the processing of an action to the Visual Studio Output window.

Using an Action Filter

An action filter is an attribute. You can apply most action filters to either an individual controller action or an entire controller.

For example, the Data controller in Listing 1 exposes an action named `Index()` that returns the current time. This action is decorated with the `OutputCache` action filter. This filter causes the value returned by the action to be cached for 10 seconds.

Listing 1 – `Controllers\DataController.vb`

```
Public Class DataController
     Inherits System.Web.Mvc.Controller

     <OutputCache(Duration:=10)> _
     Function Index()
          Return DateTime.Now.ToString("T")

     End Function

End Class
```

If you repeatedly invoke the `Index()` action by entering the URL /Data/Index into the address bar of your browser and hitting the Refresh button multiple times, then you will see the same time for 10 seconds. The output of the `Index()` action is cached for 10 seconds (see Figure 1).

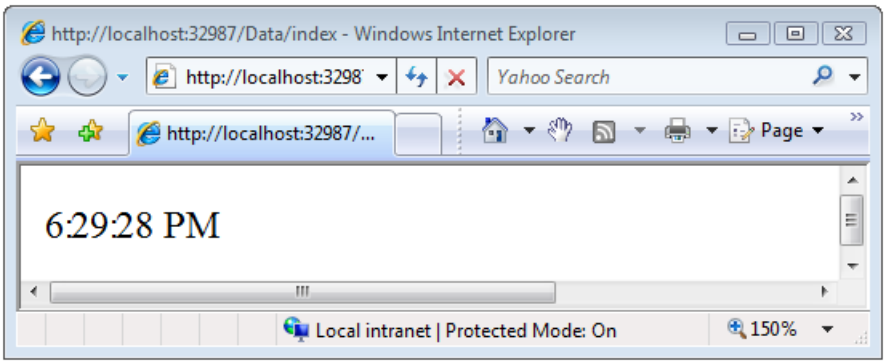


Figure 01: Cached time (Click to view full-size image)

In Listing 1, a single action filter – the `OutputCache` action filter – is applied to the `Index()` method. If you need, you can apply multiple action filters to the same action. For example, you might want to apply both the `OutputCache` and `HandleError` action filters to the same action.

In Listing 1, the `OutputCache` action filter is applied to the `Index()` action. You also could apply this attribute to the `DataController` class itself. In that case, the result returned by any action exposed by the controller would be cached for 10 seconds.

The Different Types of Filters

The ASP.NET MVC framework supports four different types of filters:

1. Authorization filters – Implements the `IAuthorizationFilter` attribute.
2. Action filters – Implements the `IActionFilter` attribute.
3. Result filters – Implements the `IResultFilter` attribute.
4. Exception filters – Implements the `IExceptionFilter` attribute.

Filters are executed in the order listed above. For example, authorization filters are always executed before action filters and exception filters are always executed after every other type of filter.

Authorization filters are used to implement authentication and authorization for controller actions. For example, the Authorize filter is an example of an Authorization filter.

Action filters contain logic that is executed before and after a controller action executes. You can use an action filter, for instance, to modify the view data that a controller action returns.

Result filters contain logic that is executed before and after a view result is executed. For example, you might want to modify a view result right before the view is rendered to the browser.

Exception filters are the last type of filter to run. You can use an exception filter to handle errors raised by either your controller actions or controller action results. You also can use exception filters to log errors.

Each different type of filter is executed in a particular order. If you want to control the order in which filters of the same type are executed then you can set a filter's Order property.

The base class for all action filters is the `System.Web.Mvc.FilterAttribute` class. If you want to implement a particular type of filter, then you need to create a class that inherits from the base Filter class and implements one or more of the IAuthorizationFilter, IActionFilter, IResultFilter, or ExceptionFilter interfaces.

The Base ActionFilterAttribute Class

In order to make it easier for you to implement a custom action filter, the ASP.NET MVC framework includes a base `ActionFilterAttribute` class. This class implements both the `IActionFilter` and `IResultFilter` interfaces and inherits from the `Filter` class.

The terminology here is not entirely consistent. Technically, a class that inherits from the ActionFilterAttribute class is both an action filter and a result filter. However, in the loose sense, the word action filter is used to refer to any

type of filter in the ASP.NET MVC framework.

The base ActionFilterAttribute class has the following methods that you can override:

- OnActionExecuting – This method is called before a controller action is executed.
- OnActionExecuted – This method is called after a controller action is executed.
- OnResultExecuting – This method is called before a controller action result is executed.
- OnResultExecuted – This method is called after a controller action result is executed.

In the next section, we'll see how you can implement each of these different methods.

Creating a Log Action Filter

In order to illustrate how you can build a custom action filter, we'll create a custom action filter that logs the stages of processing a controller action to the Visual Studio Output window. Our `LogActionFilter` is contained in Listing 2.

Listing 2 – `ActionFilters\LogActionFilter.vb`

```
Public Class LogActionFilter
     Inherits ActionFilterAttribute

     Public Overrides Sub OnActionExecuting(ByVal filterContext As ActionExecutingContext)
          Log("OnActionExecuting", filterContext.RouteData)

     End Sub

     Public Overrides Sub OnActionExecuted(ByVal filterContext As ActionExecutedContext)
          Log("OnActionExecuted", filterContext.RouteData)
     End Sub

     Public Overrides Sub OnResultExecuting(ByVal filterContext As ResultExecutingContext)
          Log("OnResultExecuting", filterContext.RouteData)
     End Sub

     Public Overrides Sub OnResultExecuted(ByVal filterContext As ResultExecutedContext)
          Log("OnResultExecuted", filterContext.RouteData)
     End Sub

     Private Sub Log(ByVal methodName As String, ByVal routeData As RouteData)
          Dim controllerName = routeData.Values("controller")
          Dim actionName = routeData.Values("action")
          Dim message = String.Format("{0} controller:{1} action:{2}", methodName, controllerName, actionName)
          Debug.WriteLine(message, "Action Filter Log")
     End Sub

End Class
```

In Listing 2, the `OnActionExecuting()`, `OnActionExecuted()`, `OnResultExecuting()`, and `OnResultExecuted()` methods all call the `Log()` method. The name of the method and the current route data is passed to the `Log()` method. The `Log()` method writes a message to the Visual Studio Output window (see Figure 2).

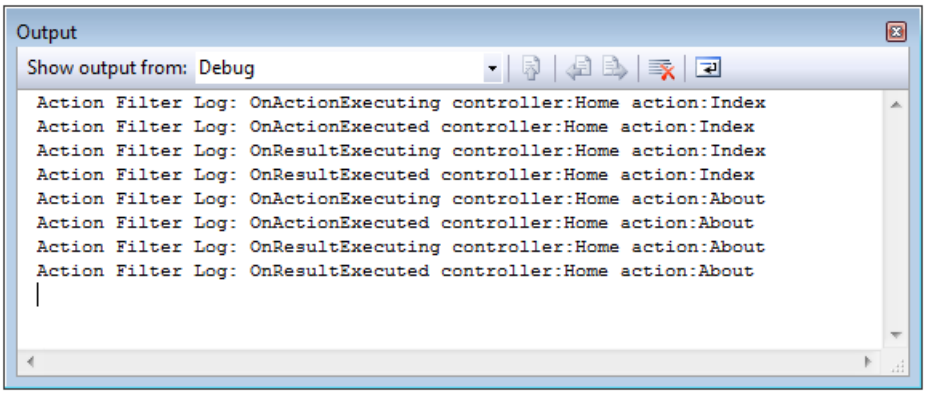


Figure 02: Writing to the Visual Studio Output window (Click to view full-size image)

The Home controller in Listing 3 illustrates how you can apply the Log action filter to an entire controller class. Whenever any of the actions exposed by the Home controller are invoked – either the `Index()` method or the `About()` method – the stages of processing the action are logged to the Visual Studio Output window.

Listing 3 – `Controllers\HomeController.vb`

```
<LogActionFilter()> _
Public Class HomeController
     Inherits System.Web.Mvc.Controller

     Function Index()
          Return View()
     End Function

     Function About()
          Return View()
     End Function

            End Class
```

Summary

In this tutorial, you were introduced to ASP.NET MVC action filters. You learned about the four different types of filters: authorization filters, action filters, result filters, and exception filters. You also learned about the base `ActionFilterAttribute` class.

Finally, you learned how to implement a simple action filter. We created a Log action filter that logs the stages of processing a controller action to the Visual Studio Output window.

PREVIOUS NEXT

Improving Performance with Output Caching (VB)

4/10/2020 • 9 minutes to read • Edit Online

by Microsoft

> In this tutorial, you learn how you can dramatically improve the performance of your ASP.NET MVC web applications by taking advantage of output caching. You learn how to cache the result returned from a controller action so that the same content does not need to be created each and every time a new user invokes the action.

The goal of this tutorial is to explain how you can dramatically improve the performance of an ASP.NET MVC application by taking advantage of the output cache. The output cache enables you to cache the content returned by a controller action. That way, the same content does not need to be generated each and every time the same controller action is invoked.

Imagine, for example, that your ASP.NET MVC application displays a list of database records in a view named Index. Normally, each and every time that a user invokes the controller action that returns the Index view, the set of database records must be retrieved from the database by executing a database query.

If, on the other hand, you take advantage of the output cache then you can avoid executing a database query every time any user invokes the same controller action. The view can be retrieved from the cache instead of being regenerated from the controller action. Caching enables you to avoid performing redundant work on the server.

Enabling Output Caching

You enable output caching by adding an <OutputCache> attribute to either an individual controller action or an entire controller class. For example, the controller in Listing 1 exposes an action named Index(). The output of the Index() action is cached for 10 seconds.

Listing 1 – Controllers\HomeController.vb

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    <OutputCache(Duration:=10, VaryByParam:="none")> _
    Function Index()
        Return View()
    End Function

End Class
```

In the Beta versions of ASP.NET MVC, output caching does not work for a URL like http://www.MySite.com/. Instead, you must enter a URL like http://www.MySite.com/Home/Index.

In Listing 1, the output of the Index() action is cached for 10 seconds. If you prefer, you can specify a much longer cache duration. For example, if you want to cache the output of a controller action for one day then you can specify a cache duration of 86400 seconds (60 seconds * 60 minutes * 24 hours).

There is no guarantee that content will be cached for the amount of time that you specify. When memory resources become low, the cache starts evicting content automatically.

The Home controller in Listing 1 returns the Index view in Listing 2. There is nothing special about this view. The Index view simply displays the current time (see Figure 1).

Listing 2 – Views\Home\Index.aspx

```
<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title></title>
</head>
<body>
    <div>

    The current time is: <%= DateTime.Now.ToString("T") %>

    </div>
</body>
</html>
```

Figure 1 – Cached Index view

If you invoke the Index() action multiple times by entering the URL /Home/Index in the address bar of your browser and hitting the Refresh/Reload button in your browser repeatedly, then the time displayed by the Index view won't change for 10 seconds. The same time is displayed because the view is cached.

It is important to understand that the same view is cached for everyone who visits your application. Anyone who invokes the Index() action will get the same cached version of the Index view. This means that the amount of work that the web server must perform to serve the Index view is dramatically reduced.

The view in Listing 2 happens to be doing something really simple. The view just displays the current time. However, you could just as easily cache a view that displays a set of database records. In that case, the set of database records would not need to be retrieved from the database each and every time the controller action that returns the view is invoked. Caching can reduce the amount of work that both your web server and database server must perform.

Don't use the page <%@ OutputCache %> directive in an MVC view. This directive is bleeding over from the Web Forms world and should not be used in an ASP.NET MVC application.

Where Content is Cached

By default, when you use the <OutputCache> attribute, content is cached in three locations: the web server, any proxy servers, and the web browser. You can control exactly where content is cached by modifying the Location property of the <OutputCache> attribute.

You can set the Location property to any one of the following values:

- Any
- Client

· Downstream

· Server

· None

· ServerAndClient

By default, the Location property has the value Any. However, there are situations in which you might want to cache only on the browser or only on the server. For example, if you are caching information that is personalized for each user then you should not cache the information on the server. If you are displaying different information to different users then you should cache the information only on the client.

For example, the controller in Listing 3 exposes an action named GetName() that returns the current user name. If Jack logs into the website and invokes the GetName() action then the action returns the string "Hi Jack". If, subsequently, Jill logs into the website and invokes the GetName() action then she also will get the string "Hi Jack". The string is cached on the web server for all users after Jack initially invokes the controller action.

Listing 3 – Controllers\BadUserController.vb

```
Public Class BadUserController
    Inherits System.Web.Mvc.Controller

    <OutputCache(Duration:=3600, VaryByParam:="none")> _
    Function Index()
        Return "Hi " & User.Identity.Name
    End Function

End Class
```

Most likely, the controller in Listing 3 does not work the way that you want. You don't want to display the message "Hi Jack" to Jill.

You should never cache personalized content in the server cache. However, you might want to cache the personalized content in the browser cache to improve performance. If you cache content in the browser, and a user invokes the same controller action multiple times, then the content can be retrieved from the browser cache instead of the server.

The modified controller in Listing 4 caches the output of the GetName() action. However, the content is cached only on the browser and not on the server. That way, when multiple users invoke the GetName() method, each person gets their own user name and not another person's user name.

Listing 4 – Controllers\UserController.vb

```
Public Class UserController
    Inherits System.Web.Mvc.Controller

    <OutputCache(Duration:=3600, VaryByParam:="none", Location:=OutputCacheLocation.Client, NoStore:=True)> _
    Function GetName()
        Return "Hi " & User.Identity.Name
    End Function

End Class
```

Notice that the <OutputCache> attribute in Listing 4 includes a Location property set to the value OutputCacheLocation.Client. The <OutputCache> attribute also includes a NoStore property. The NoStore property is used to inform proxy servers and browsers that they should not store a permanent copy of the cached content.

Varying the Output Cache

In some situations, you might want different cached versions of the very same content. Imagine, for example, that you are creating a master/detail page. The master page displays a list of movie titles. When you click a title, you get details for the selected movie.

If you cache the details page, then the details for the same movie will be displayed no matter which movie you click. The first movie selected by the first user will be displayed to all future users.

You can fix this problem by taking advantage of the VaryByParam property of the <OutputCache> attribute. This property enables you to create different cached versions of the very same content when a form parameter or query string parameter varies.

For example, the controller in Listing 5 exposes two actions named Master() and Details(). The Master() action returns a list of movie titles and the Details() action returns the details for the selected movie.

Listing 5 – Controllers\MoviesController.vb

```
Public Class MoviesController
    Inherits System.Web.Mvc.Controller

    Private _dataContext As MovieDataContext

    Public Sub New()
        _dataContext = New MovieDataContext()
    End Sub

    <OutputCache(Duration:=Integer.MaxValue, VaryByParam:="none")> _
    Public Function Master()
        ViewData.Model = (From m In _dataContext.Movies _
                          Select m).ToList()
        Return View()
    End Function

    <OutputCache(Duration:=Integer.MaxValue, VaryByParam:="id")> _
    Public Function Details(ByVal id As Integer)
        ViewData.Model = _dataContext.Movies.SingleOrDefault(Function(m) m.Id = id)
        Return View()
    End Function

End Class
```

The Master() action includes a VaryByParam property with the value "none". When the Master() action is invoked, the same cached version of the Master view is returned. Any form parameters or query string parameters are ignored (see Figure 2).

Figure 2 – The /Movies/Master view

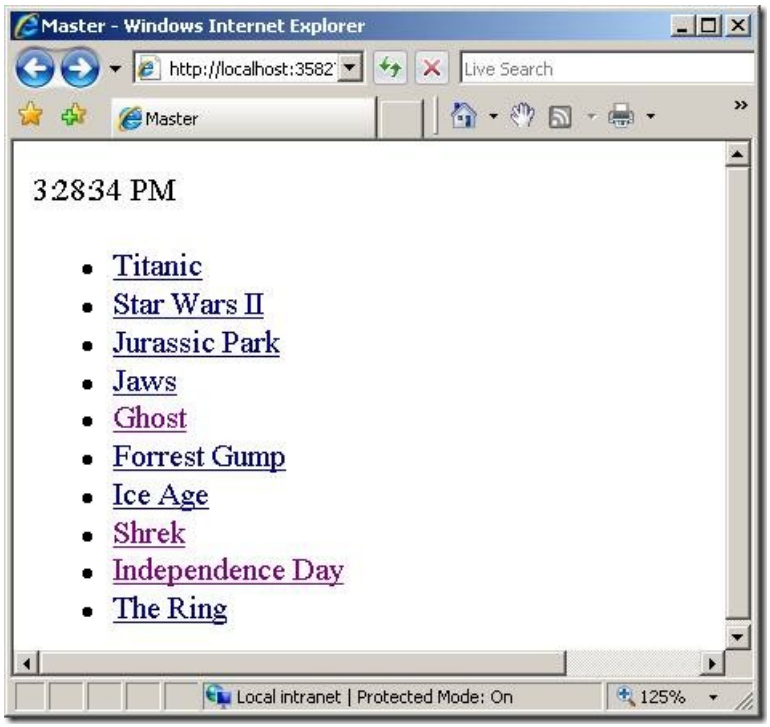
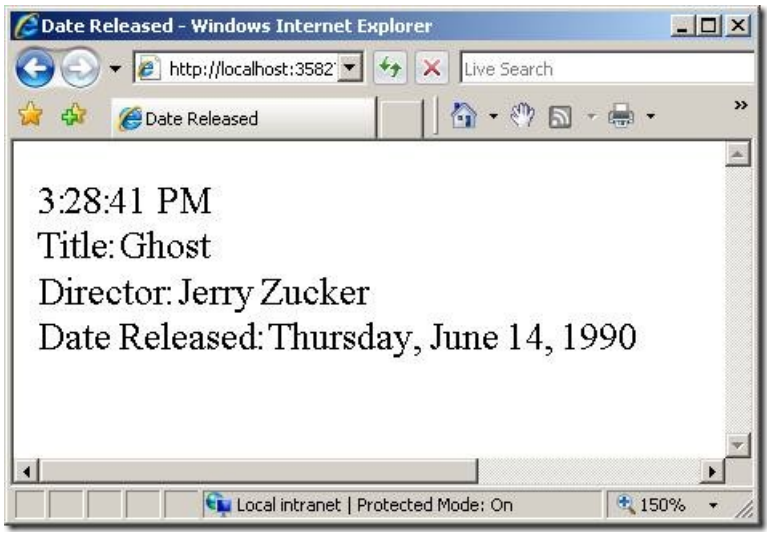


Figure 3 – The /Movies/Details view



The Details() action includes a VaryByParam property with the value "Id". When different values of the Id parameter are passed to the controller action, different cached versions of the Details view are generated.

It is important to understand that using the VaryByParam property results in more caching and not less. A different cached version of the Details view is created for each different version of the Id parameter.

You can set the VaryByParam property to the following values:

> * = Create a different cached version whenever a form or query string parameter varies.
>
> none = Never create different cached versions
>
> Semicolon list of parameters = Create different cached versions whenever any of the form or query string parameters in the list varies

Creating a Cache Profile

As an alternative to configuring output cache properties by modifying properties of the <OutputCache> attribute, you can create a cache profile in the web configuration (web.config) file. Creating a cache profile in the web configuration file offers a couple of important advantages.

First, by configuring output caching in the web configuration file, you can control how controller actions cache content in one central location. You can create one cache profile and apply the profile to several controllers or controller actions.

Second, you can modify the web configuration file without recompiling your application. If you need to disable caching for an application that has already been deployed to production, then you can simply modify the cache profiles defined in the web configuration file. Any changes to the web configuration file will be detected automatically and applied.

For example, the <caching> web configuration section in Listing 6 defines a cache profile named Cache1Hour. The <caching> section must appear within the <system.web> section of a web configuration file.

Listing 6 – Caching section for web.config

```
<caching>
<outputCacheSettings>
    <outputCacheProfiles>
        <add name="Cache1Hour" duration="3600" varyByParam="none"/>
    </outputCacheProfiles>
</outputCacheSettings>
</caching>
```

The controller in Listing 7 illustrates how you can apply the Cache1Hour profile to a controller action with the <OutputCache> attribute.

Listing 7 – Controllers\ProfileController.vb

```
Public Class ProfileController
    Inherits System.Web.Mvc.Controller

    <OutputCache(CacheProfile:="Cache1Hour")> _
    Function Index()
        Return DateTime.Now.ToString("T")
    End Function

End Class
```

If you invoke the Index() action exposed by the controller in Listing 7 then the same time will be returned for 1 hour.

Summary

Output caching provides you with a very easy method of dramatically improving the performance of your ASP.NET MVC applications. In this tutorial, you learned how to use the <OutputCache> attribute to cache the output of controller actions. You also learned how to modify properties of the <OutputCache> attribute such as the Duration and VaryByParam properties to modify how content gets cached. Finally, you learned how to define cache profiles in the web configuration file.

PREVIOUS NEXT

Adding Dynamic Content to a Cached Page (VB)

4/10/2020 • 5 minutes to read • Edit Online

by Microsoft

Learn how to mix dynamic and cached content in the same page. Post-cache substitution enables you to display dynamic content, such as banner advertisements or news items, within a page that has been output cached.

By taking advantage of output caching, you can dramatically improve the performance of an ASP.NET MVC application. Instead of regenerating a page each and every time the page is requested, the page can be generated once and cached in memory for multiple users.

But there is a problem. What if you need to display dynamic content in the page? For example, imagine that you want to display a banner advertisement in the page. You don't want the banner advertisement to be cached so that every user sees the very same advertisement. You wouldn't make any money that way!

Fortunately, there is an easy solution. You can take advantage of a feature of the ASP.NET framework called *post-cache substitution*. Post-cache substitution enables you to substitute dynamic content in a page that has been cached in memory.

Normally, when you output cache a page by using the <OutputCache> attribute, the page is cached on both the server and the client (the web browser). When you use post-cache substitution, a page is cached only on the server.

Using Post-Cache Substitution

Using post-cache substitution requires two steps. First, you need to define a method that returns a string that represents the dynamic content that you want to display in the cached page. Next, you call the HttpResponse.WriteSubstitution() method to inject the dynamic content into the page.

Imagine, for example, that you want to randomly display different news items in a cached page. The class in Listing 1 exposes a single method, named RenderNews(), that randomly returns one news item from a list of three news items.

Listing 1 – Models\News.vb

```
Public Class News

    Shared Function RenderNews(ByVal context As HttpContext) As String
        Dim newsItems As New List(Of String)
        newsItems.Add("Gas prices go up!")
        newsItems.Add("Life discovered on Mars!")
        newsItems.Add("Moon disappears!")

        Dim rnd As New Random()
        Return newsItems(rnd.Next(newsItems.Count))
    End Function

End Class
```

To take advantage of post-cache substitution, you call the HttpResponse.WriteSubstitution() method. The WriteSubstitution() method sets up the code to replace a region of the cached page with dynamic content. The WriteSubstitution() method is used to display the random news item in the view in Listing 2.

Listing 2 – Views\Home\Index.aspx

```
<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage" %>
<%@ Import Namespace="MvcApplication1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Index</title>
</head>
<body>
    <div>

    <% Response.WriteSubstitution(AddressOf News.RenderNews)%>

    <hr />

    The content of this page is output cached.
    <%= DateTime.Now %>

    </div>
</body>
</html>
```

The RenderNews method is passed to the WriteSubstitution() method. Notice that the RenderNews method is not called. Instead a reference to the method is passed to WriteSubstitution() with the help of the AddressOf operator.

The Index view is cached. The view is returned by the controller in Listing 3. Notice that the Index() action is decorated with an <OutputCache> attribute that causes the Index view to be cached for 60 seconds.

Listing 3 – Controllers\HomeController.vb

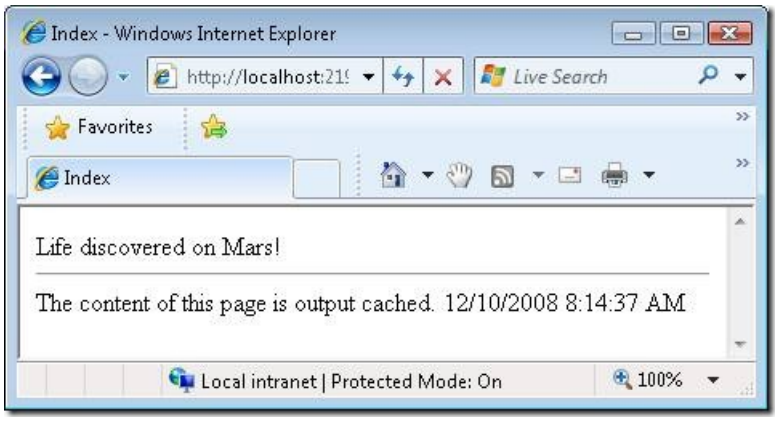
```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    <OutputCache(Duration:=60, VaryByParam:="none")> _
    Function Index()
        Return View()
    End Function

End Class
```

Even though the Index view is cached, different random news items are displayed when you request the Index page. When you request the Index page, the time displayed by the page does not change for 60 seconds (see Figure 1). The fact that the time does not change proves that the page is cached. However, the content injected by the WriteSubstitution() method – the random news item – changes with each request .

Figure 1 – Injecting dynamic news items in a cached page



Using Post-Cache Substitution in Helper Methods

An easier way to take advantage of post-cache substitution is to encapsulate the call to the WriteSubstitution() method within a custom helper method. This approach is illustrated by the helper method in Listing 4.

Listing 4 – Helpers\AdHelper.vb

```
Imports System.Runtime.CompilerServices

Public Module AdHelper

    <Extension()> _
    Sub RenderBanner(ByVal helper As HtmlHelper)
        Dim context = helper.ViewContext.HttpContext
        context.Response.WriteSubstitution(AddressOf RenderBannerInternal)
    End Sub

    Private Function RenderBannerInternal(ByVal context As HttpContext) As String
        Dim ads As New List(Of String)
        ads.Add("/ads/banner1.gif")
        ads.Add("/ads/banner2.gif")
        ads.Add("/ads/banner3.gif")

        Dim rnd As New Random()
        Dim ad = ads(rnd.Next(ads.Count))
        Return String.Format("<img src='{0}' />", ad)
    End Function

End Module
```

Listing 4 contains a Visual Basic module that exposes two methods: RenderBanner() and RenderBannerInternal(). The RenderBanner() method represents the actual helper method. This method extends the standard ASP.NET MVC HtmlHelper class so that you can call Html.RenderBanner() in a view just like any other helper method.

The RenderBanner() method calls the HttpResponse.WriteSubstitution() method passing the RenderBannerInternal() method to the WriteSubstitution() method.

The RenderBannerInternal() method is a private method. This method won't be exposed as a helper method. The RenderBannerInternal() method randomly returns one banner advertisement image from a list of three banner advertisement images.

The modified Index view in Listing 5 illustrates how you can use the RenderBanner() helper method. Notice that an additional <%@ Import %> directive is included at the top of the view to import the MvcApplication1.Helpers namespace. If you neglect to import this namespace, then the RenderBanner() method won't appear as a method on the Html property.

Listing 5 – Views\Home\Index.aspx (with RenderBanner() method)

```
<%@ Page Language="VB" Inherits="System.Web.Mvc.ViewPage" %>
<%@ Import Namespace="MvcApplication1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>Index</title>
</head>
<body>
    <div>

    <% Response.WriteSubstitution(AddressOf News.RenderNews)%>

    <hr />

    <% Html.RenderBanner()%>

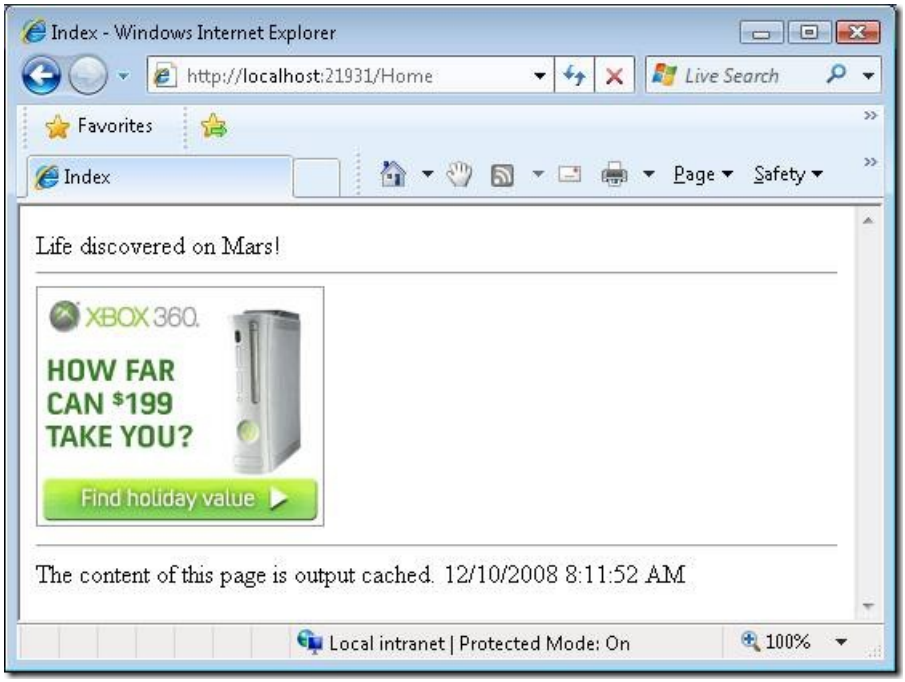
    <hr />

    The content of this page is output cached.
    <%= DateTime.Now %>

    </div>
</body>
</html>
```

When you request the page rendered by the view in Listing 5, a different banner advertisement is displayed with each request (see Figure 2). The page is cached, but the banner advertisement is injected dynamically by the RenderBanner() helper method.

Figure 2 – The Index view displaying a random banner advertisement



Summary

This tutorial explained how you can dynamically update content in a cached page. You learned how to use the HttpResponse.WriteSubstitution() method to enable dynamic content to be injected in a cached page. You also learned how to encapsulate the call to the WriteSubstitution() method within an HTML helper method.

Take advantage of caching whenever possible – it can have a dramatic impact on the performance of your web applications. As explained in this tutorial, you can take advantage of caching even when you need to display dynamic content in your pages.

Creating a Controller (VB)

2/20/2020 • 3 minutes to read • Edit Online

by Stephen Walther

In this tutorial, Stephen Walther demonstrates how you can add a controller to an ASP.NET MVC application.

The goal of this tutorial is to explain how you can create new ASP.NET MVC controllers. You learn how to create controllers both by using the Visual Studio Add Controller menu option and by creating a class file by hand.

Using the Add Controller Menu Option

The easiest way to create a new controller is to right-click the Controllers folder in the Visual Studio Solution Explorer window and select the **Add, Controller** menu option (see Figure 1). Selecting this menu option opens the **Add Controller** dialog (see Figure 2).

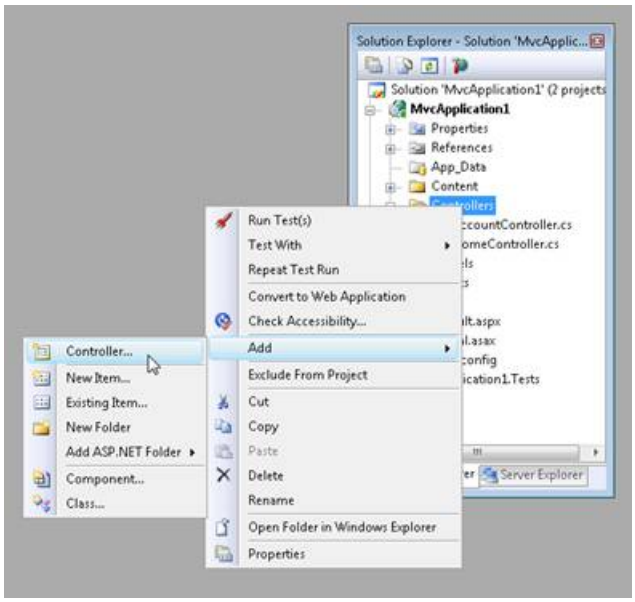


Figure 01: Adding a new controller(Click to view full-size image)

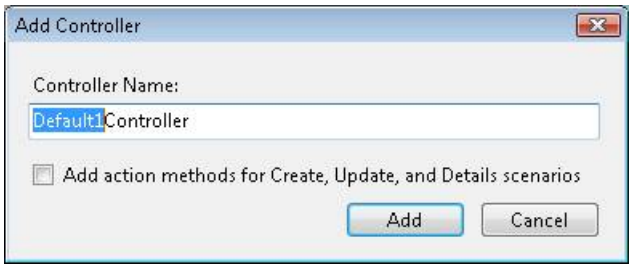


Figure 02: The Add Controller dialog (Click to view full-size image)

Notice that the first part of the controller name is highlighted in the **Add Controller** dialog. Every controller name must end with the suffix *Controller*. For example, you can create a controller named *ProductController* but not a controller named *Product*.

If you create a controller that is missing the *Controller* suffix then you won't be able to invoke the controller. Don't do this -- I've wasted countless hours of my life after making this mistake.

Listing 1 - Controllers\ProductController.vb

```
Public Class ProductController
    Inherits System.Web.Mvc.Controller

    '
    ' GET: /Product/

    Function Index() As ActionResult
        Return View()
    End Function

End Class
```

You should always create controllers in the Controllers folder. Otherwise, you'll be violating the conventions of ASP.NET MVC and other developers will have a more difficult time understanding your application.

Scaffolding Action Methods

When you create a controller, you have the option to generate Create, Update, and Details action methods automatically (see Figure 3). If you select this option then the controller class in Listing 2 is generated.

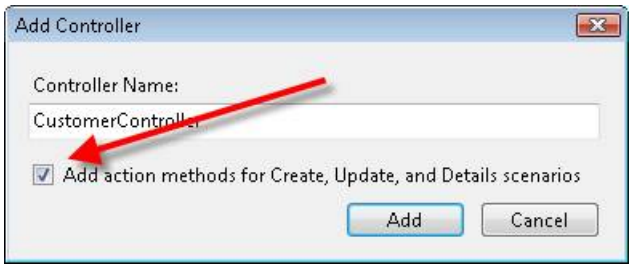


Figure 03: Creating action methods automatically (Click to view full-size image)

Listing 2 - Controllers\CustomerController.vb

```
Public Class CustomerController
    Inherits System.Web.Mvc.Controller

    '
    ' GET: /Customer/

    Function Index() As ActionResult
        Return View()
    End Function

    '
    ' GET: /Customer/Details/5

    Function Details(ByVal id As Integer) As ActionResult
        Return View()
    End Function

    '
    ' GET: /Customer/Create

    Function Create() As ActionResult
        Return View()
    End Function

    '
    ' POST: /Customer/Create

    <AcceptVerbs(HttpVerbs.Post)> _
    Function Create(ByVal collection As FormCollection) As ActionResult
        Try
            ' TODO: Add insert logic here
            Return RedirectToAction("Index")
        Catch
            Return View()
        End Try
    End Function

    '
    ' GET: /Customer/Edit/5

    Function Edit(ByVal id As Integer) As ActionResult
        Return View()
    End Function

    '
    ' POST: /Customer/Edit/5

    <AcceptVerbs(HttpVerbs.Post)> _
    Function Edit(ByVal id As Integer, ByVal collection As FormCollection) As ActionResult
        Try
            ' TODO: Add update logic here

            Return RedirectToAction("Index")
        Catch
            Return View()
        End Try
    End Function
End Class
```

These generated methods are stub methods. You must add the actual logic for creating, updating, and showing details for a customer yourself. But, the stub methods provide you with a nice starting point.

Creating a Controller Class

The ASP.NET MVC controller is just a class. If you prefer, you can ignore the convenient Visual Studio controller scaffolding and create a controller class by hand. Follow these steps:

1. Right-click the Controllers folder and select the menu option **Add, New Item** and select the **Class** template (see Figure 4).
2. Name the new class PersonController.vb and click the **Add** button.
3. Modify the resulting class file so that the class inherits from the base System.Web.Mvc.Controller class (see Listing 3).

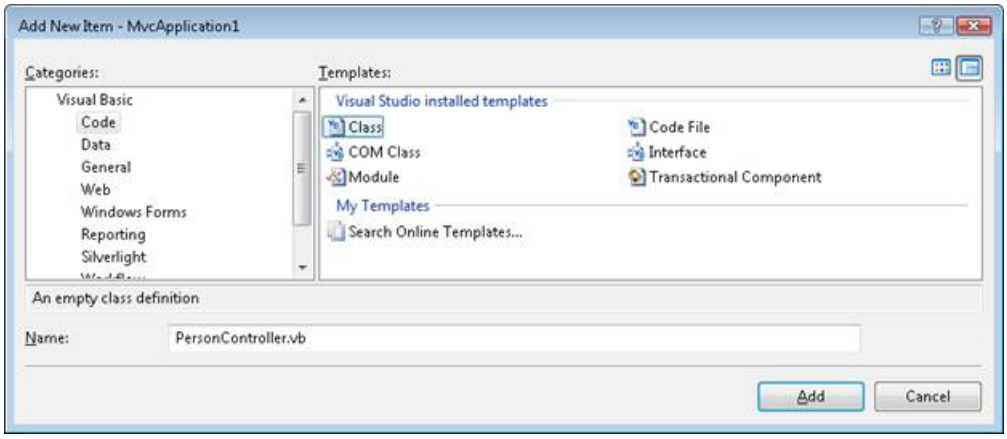


Figure 04: Creating a new class(Click to view full-size image)

Listing 3 - Controllers\PersonController.vb

```
Public Class PersonController
    inherits Controller

    Function Index AS String
        Return "Hello World!"
    End Function

End Class
```

The controller in Listing 3 exposes one action named Index() that returns the string "Hello World!". You can invoke this controller action by running your application and requesting a URL like the following:

```
http://localhost:40071/Person
```

NOTE

The ASP.NET Development Server uses a random port number (for example, 40071). When entering a URL to invoke a controller, you'll need to supply the right port number. You can determine the port number by hovering your mouse over the icon for the ASP.NET Development Server in the Windows Notification Area (bottom-right of your screen).

PREVIOUS NEXT

Creating an Action (VB)

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

> Learn how to add a new action to an ASP.NET MVC controller. Learn about the requirements for a method to be an action.

The goal of this tutorial is to explain how you can create a new controller action. You learn about the requirements of an action method. You also learn how to prevent a method from being exposed as an action.

Adding an Action to a Controller

You add a new action to a controller by adding a new method to the controller. For example, the controller in Listing 1 contains an action named Index() and an action named SayHello(). Both methods are exposed as actions.

Listing 1 - Controllers\HomeController.vb

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Function Index() As ActionResult
        Return View()
    End Function

    Function SayHello() As String
        Return "Hello!"
    End Function
End Class
```

In order to be exposed to the universe as an action, a method must meet certain requirements:

- The method must be public.
- The method cannot be a static method.
- The method cannot be an extension method.
- The method cannot be a constructor, getter, or setter.
- The method cannot have open generic types.
- The method is not a method of the controller base class.
- The method cannot contain **ref** or **out** parameters.

Notice that there are no restrictions on the return type of a controller action. A controller action can return a string, a DateTime, an instance of the Random class, or void. The ASP.NET MVC framework will convert any return type that is not an action result into a string and render the string to the browser.

When you add any method that does not violate these requirements to a controller, the method is exposed as a controller action. Be careful here. A controller action can be invoked by anyone connected to the Internet. Do not, for example, create a DeleteMyWebsite() controller action.

Preventing a Public Method from Being Invoked

If you need to create a public method in a controller class and you don't want to expose the method as a controller

action then you can prevent the method from being invoked by using the <NonAction> attribute. For example, the controller in Listing 2 contains a public method named CompanySecrets() that is decorated with the <NonAction> attribute.

Listing 2 - Controllers\WorkController.vb

```
Public Class WorkController
    Inherits System.Web.Mvc.Controller


     _
    Function CompanySecrets() As String
        Return "This information is secret."
    End Function

End Class
```

If you attempt to invoke the CompanySecrets() controller action by typing /Work/CompanySecrets into the address bar of your browser then you'll get the error message in Figure 1.

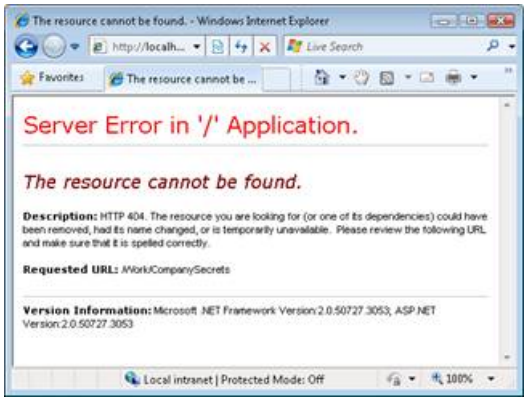


Figure 01: Invoking a NonAction method(Click to view full-size image)

PREVIOUS NEXT

ASP.NET MVC Controller Overview (C#)

2/20/2020 • 5 minutes to read • Edit Online

by Stephen Walther

> In this tutorial, Stephen Walther introduces you to ASP.NET MVC controllers. You learn how to create new controllers and return different types of action results.

This tutorial explores the topic of ASP.NET MVC controllers, controller actions, and action results. After you complete this tutorial, you will understand how controllers are used to control the way a visitor interacts with an ASP.NET MVC website.

Understanding Controllers

MVC controllers are responsible for responding to requests made against an ASP.NET MVC website. Each browser request is mapped to a particular controller. For example, imagine that you enter the following URL into the address bar of your browser:

```
http://localhost/Product/Index/3
```

In this case, a controller named ProductController is invoked. The ProductController is responsible for generating the response to the browser request. For example, the controller might return a particular view back to the browser or the controller might redirect the user to another controller.

Listing 1 contains a simple controller named ProductController.

Listing1 - Controllers\ProductController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Mvc.Ajax;

namespace MvcApplication1.Controllers
{
    public class ProductController : Controller
    {
        //
        // GET: /Products/

        public ActionResult Index()
        {
            // Add action logic here
            return View();
        }
    }
}
```

As you can see from Listing 1, a controller is just a class (a Visual Basic .NET or C# class). A controller is a class that derives from the base System.Web.Mvc.Controller class. Because a controller inherits from this base class, a controller inherits several useful methods for free (We discuss these methods in a moment).

Understanding Controller Actions

A controller exposes controller actions. An action is a method on a controller that gets called when you enter a particular URL in your browser address bar. For example, imagine that you make a request for the following URL:

```
http://localhost/Product/Index/3
```

In this case, the Index() method is called on the ProductController class. The Index() method is an example of a controller action.

A controller action must be a public method of a controller class. C# methods, by default, are private methods. Realize that any public method that you add to a controller class is exposed as a controller action automatically (You must be careful about this since a controller action can be invoked by anyone in the universe simply by typing the right URL into a browser address bar).

There are some additional requirements that must be satisfied by a controller action. A method used as a controller action cannot be overloaded. Furthermore, a controller action cannot be a static method. Other than that, you can use just about any method as a controller action.

Understanding Action Results

A controller action returns something called an *action result*. An action result is what a controller action returns in response to a browser request.

The ASP.NET MVC framework supports several types of action results including:

1. ViewResult - Represents HTML and markup.
2. EmptyResult - Represents no result.
3. RedirectResult - Represents a redirection to a new URL.
4. JsonResult - Represents a JavaScript Object Notation result that can be used in an AJAX application.
5. JavaScriptResult - Represents a JavaScript script.
6. ContentResult - Represents a text result.
7. FileContentResult - Represents a downloadable file (with the binary content).
8. FilePathResult - Represents a downloadable file (with a path).
9. FileStreamResult - Represents a downloadable file (with a file stream).

All of these action results inherit from the base ActionResult class.

In most cases, a controller action returns a ViewResult. For example, the Index controller action in Listing 2 returns a ViewResult.

Listing 2 - Controllers\BookController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Mvc.Ajax;

namespace MvcApplication1.Controllers
{
    public class BookController : Controller
    {
        public ActionResult Index()
        {
            // Add action logic here
            return View();
        }
    }
}
```

When an action returns a ViewResult, HTML is returned to the browser. The Index() method in Listing 2 returns a view named Index to the browser.

Notice that the Index() action in Listing 2 does not return a ViewResult(). Instead, the View() method of the Controller base class is called. Normally, you do not return an action result directly. Instead, you call one of the following methods of the Controller base class:

1. View - Returns a ViewResult action result.
2. Redirect - Returns a RedirectResult action result.
3. RedirectToAction - Returns a RedirectToRouteResult action result.
4. RedirectToRoute - Returns a RedirectToRouteResult action result.
5. Json - Returns a JsonResult action result.
6. JavaScriptResult - Returns a JavaScriptResult.
7. Content - Returns a ContentResult action result.
8. File - Returns a FileContentResult, FilePathResult, or FileStreamResult depending on the parameters passed to the method.

So, if you want to return a View to the browser, you call the View() method. If you want to redirect the user from one controller action to another, you call the RedirectToAction() method. For example, the Details() action in Listing 3 either displays a view or redirects the user to the Index() action depending on whether the Id parameter has a value.

Listing 3 - CustomerController.cs

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    public class CustomerController : Controller
    {
        public ActionResult Details(int? id)
        {
            if (!id.HasValue)
                return RedirectToAction("Index");

            return View();
        }

        public ActionResult Index()
        {
            return View();
        }

    }
}
```

The ContentResult action result is special. You can use the ContentResult action result to return an action result as plain text. For example, the Index() method in Listing 4 returns a message as plain text and not as HTML.

Listing 4 - Controllers\StatusController.cs

```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    public class StatusController : Controller
    {
        public ActionResult Index()
        {
            return Content("Hello World!");
        }
    }
}
```

When the StatusController.Index() action is invoked, a view is not returned. Instead, the raw text "Hello World!" is returned to the browser.

If a controller action returns a result that is not an action result - for example, a date or an integer - then the result is wrapped in a ContentResult automatically. For example, when the Index() action of the WorkController in Listing 5 is invoked, the date is returned as a ContentResult automatically.

Listing 5 - WorkController.cs

```
using System;
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    public class WorkController : Controller
    {
        public DateTime Index()
        {
            return DateTime.Now;
        }
    }
}
```

The Index() action in Listing 5 returns a DateTime object. The ASP.NET MVC framework converts the DateTime object to a string and wraps the DateTime value in a ContentResult automatically. The browser receives the date and time as plain text.

Summary

The purpose of this tutorial was to introduce you to the concepts of ASP.NET MVC controllers, controller actions, and controller action results. In the first section, you learned how to add new controllers to an ASP.NET MVC project. Next, you learned how public methods of a controller are exposed to the universe as controller actions. Finally, we discussed the different types of action results that can be returned from a controller action. In particular, we discussed how to return a ViewResult, RedirectToActionResult, and ContentResult from a controller action.

PREVIOUS NEXT

Creating Custom Routes (C#)

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

> Learn how to add custom routes to an ASP.NET MVC application. In this tutorial, you learn how to modify the default route table in the Global.asax file.

In this tutorial, you learn how to add a custom route to an ASP.NET MVC application. You learn how to modify the default route table in the Global.asax file with a custom route.

For many simple ASP.NET MVC applications, the default route table will work just fine. However, you might discover that you have specialized routing needs. In that case, you can create a custom route.

Imagine, for example, that you are building a blog application. You might want to handle incoming requests that look like this:

/Archive/12-25-2009

When a user enters this request, you want to return the blog entry that corresponds to the date 12/25/2009. In order to handle this type of request, you need to create a custom route.

The Global.asax file in Listing 1 contains a new custom route, named Blog, which handles requests that look like /Archive/*entry date*.

Listing 1 - Global.asax (with custom route)

```
using System.Web.Mvc;
using System.Web.Routing;
namespace MvcApplication1
{
    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(
                "Blog",                                           // Route name
                "Archive/{entryDate}",                            // URL with parameters
                new { controller = "Archive", action = "Entry" }  // Parameter defaults
            );
            routes.MapRoute(
                "Default",                                             // Route name
                "{controller}/{action}/{id}",                          // URL with parameters
                new { controller = "Home", action = "Index", id = "" }  // Parameter defaults
            );
        }
        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

The order of the routes that you add to the route table is important. Our new custom Blog route is added before the existing Default route. If you reversed the order, then the Default route always will get called instead of the custom route.

The custom Blog route matches any request that starts with /Archive/. So, it matches all of the following URLs:

- /Archive/12-25-2009
- /Archive/10-6-2004
- /Archive/apple

The custom route maps the incoming request to a controller named Archive and invokes the Entry() action. When the Entry() method is called, the entry date is passed as a parameter named entryDate.

You can use the Blog custom route with the controller in Listing 2.

Listing 2 - ArchiveController.cs

```
using System;
using System.Web.Mvc;
namespace MvcApplication1.Controllers
{
    public class ArchiveController : Controller
    {
        public string Entry(DateTime entryDate)
        {
            return "You requested the entry from " + entryDate.ToString();
        }
    }
}
```

Notice that the Entry() method in Listing 2 accepts a parameter of type DateTime. The MVC framework is smart enough to convert the entry date from the URL into a DateTime value automatically. If the entry date parameter from the URL cannot be converted to a DateTime, an error is raised (see Figure 1).

Figure 1 - Error from converting parameter

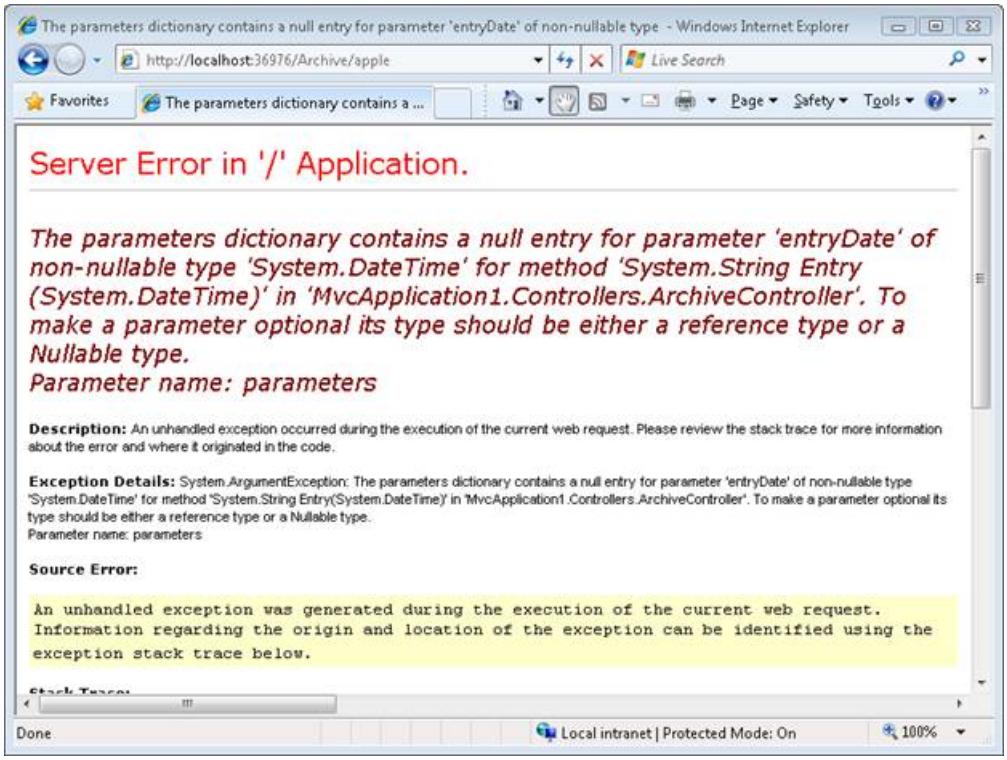


Figure 01: Error from converting parameter (Click to view full-size image)

Summary

The goal of this tutorial was to demonstrate how you can create a custom route. You learned how to add a custom route to the route table in the Global.asax file that represents blog entries. We discussed how to map requests for blog entries to a controller named ArchiveController and a controller action named Entry().

PREVIOUS NEXT

Creating a Route Constraint (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

> In this tutorial, Stephen Walther demonstrates how you can control how browser requests match routes by creating route constraints with regular expressions.

You use route constraints to restrict the browser requests that match a particular route. You can use a regular expression to specify a route constraint.

For example, imagine that you have defined the route in Listing 1 in your Global.asax file.

Listing 1 - Global.asax.cs

```
routes.MapRoute(
    "Product",
    "Product/{productId}",
    new {controller="Product", action="Details"}
);
```

Listing 1 contains a route named Product. You can use the Product route to map browser requests to the ProductController contained in Listing 2.

Listing 2 - Controllers\ProductController.cs

```
using System.Web.Mvc;
namespace MvcApplication1.Controllers
{
    public class ProductController : Controller
    {
        public ActionResult Details(int productId)
        {
            return View();
        }
    }
}
```

Notice that the Details() action exposed by the Product controller accepts a single parameter named productId. This parameter is an integer parameter.

The route defined in Listing 1 will match any of the following URLs:

- /Product/23
- /Product/7

Unfortunately, the route will also match the following URLs:

- /Product/blah
- /Product/apple

Because the Details() action expects an integer parameter, making a request that contains something other than an integer value will cause an error. For example, if you type the URL /Product/apple into your browser then you will get the error page in Figure 1.

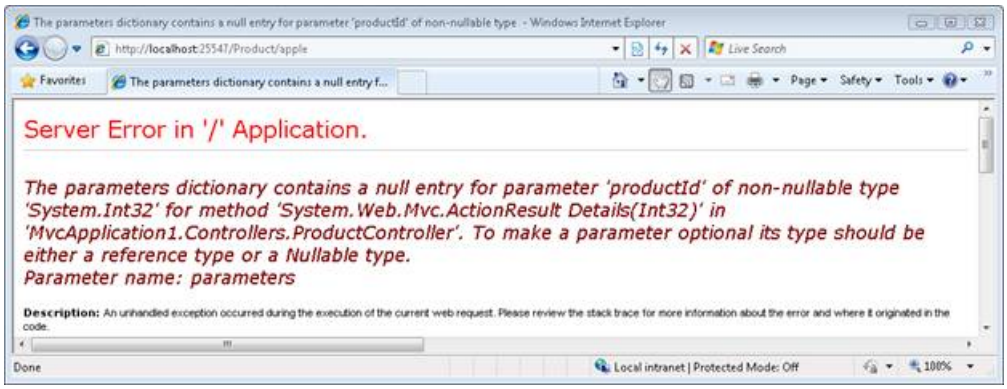


Figure 01: Seeing a page explode (Click to view full-size image)

What you really want to do is only match URLs that contain a proper integer productId. You can use a constraint when defining a route to restrict the URLs that match the route. The modified Product route in Listing 3 contains a regular expression constraint that only matches integers.

Listing 3 - Global.asax.cs

```
routes.MapRoute(
    "Product",
    "Product/{productId}",
    new {controller="Product", action="Details"},
    new {productId = @"\d+" }
 );
```

The regular expression \d+ matches one or more integers. This constraint causes the Product route to match the following URLs:

- /Product/3
- /Product/8999

But not the following URLs:

- /Product/apple
- /Product
- These browser requests will be handled by another route or, if there are no matching routes, a *The resource could not be found* error will be returned.

PREVIOUS NEXT

Creating a Custom Route Constraint (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

> Stephen Walther demonstrates how you can create a custom route constraint. We implement a simple custom constraint that prevents a route from being matched when a browser request is made from a remote computer.

The goal of this tutorial is to demonstrate how you can create a custom route constraint. A custom route constraint enables you to prevent a route from being matched unless some custom condition is matched.

In this tutorial, we create a Localhost route constraint. The Localhost route constraint only matches requests made from the local computer. Remote requests from across the Internet are not matched.

You implement a custom route constraint by implementing the IRouteConstraint interface. This is an extremely simple interface which describes a single method:

```
bool Match(
    HttpContextBase httpContext,
    Route route,
    string parameterName,
    RouteValueDictionary values,
    RouteDirection routeDirection
)
```

The method returns a Boolean value. If you return false, the route associated with the constraint won't match the browser request.

The Localhost constraint is contained in Listing 1.

Listing 1 - LocalhostConstraint.cs

```
using System.Web;
using System.Web.Routing;
namespace MvcApplication1.Constraints
{
    public class LocalhostConstraint : IRouteConstraint
    {
        public bool Match
            (
                HttpContextBase httpContext,
                Route route,
                string parameterName,
                RouteValueDictionary values,
                RouteDirection routeDirection
            )
        {
            return httpContext.Request.IsLocal;
        }
    }
}
```

The constraint in Listing 1 takes advantage of the IsLocal property exposed by the HttpRequest class. This property returns true when the IP address of the request is either 127.0.0.1 or when the IP of the request is the same as the

server's IP address.

You use a custom constraint within a route defined in the Global.asax file. The Global.asax file in Listing 2 uses the Localhost constraint to prevent anyone from requesting an Admin page unless they make the request from the local server. For example, a request for /Admin/DeleteAll will fail when made from a remote server.

Listing 2 - Global.asax

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using MvcApplication1.Constraints;
namespace MvcApplication1
{
    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(
                "Admin",
                "Admin/{action}",
                new {controller="Admin"},
                new {isLocal=new LocalhostConstraint()}
            );
            //routes.MapRoute(
            //    "Default",                                          // Route name
            //    "{controller}/{action}/{id}",                       // URL with parameters
            //    new { controller = "Home", action = "Index", id = "" }  // Parameter defaults
            //);
        }
        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

The Localhost constraint is used in the definition of the Admin route. This route won't be matched by a remote browser request. Realize, however, that other routes defined in Global.asax might match the same request. It is important to understand that a constraint prevents a particular route from matching a request and not all routes defined in the Global.asax file.

Notice that the Default route has been commented out from the Global.asax file in Listing 2. If you include the Default route, then the Default route would match requests for the Admin controller. In that case, remote users could still invoke actions of the Admin controller even though their requests wouldn't match the Admin route.

PREVIOUS NEXT

ASP.NET MVC Controller Overview (VB)

2/20/2020 • 5 minutes to read • Edit Online

by Stephen Walther

> In this tutorial, Stephen Walther introduces you to ASP.NET MVC controllers. You learn how to create new controllers and return different types of action results.

This tutorial explores the topic of ASP.NET MVC controllers, controller actions, and action results. After you complete this tutorial, you will understand how controllers are used to control the way a visitor interacts with an ASP.NET MVC website.

Understanding Controllers

MVC controllers are responsible for responding to requests made against an ASP.NET MVC website. Each browser request is mapped to a particular controller. For example, imagine that you enter the following URL into the address bar of your browser:

```
http://localhost/Product/Index/3
```

In this case, a controller named ProductController is invoked. The ProductController is responsible for generating the response to the browser request. For example, the controller might return a particular view back to the browser or the controller might redirect the user to another controller.

Listing 1 contains a simple controller named ProductController.

Listing1 - Controllers\ProductController.vb

```
Public Class ProductController
    Inherits System.Web.Mvc.Controller

    '
    ' GET: /Products/

    Function Index()
        ' Add action logic here
        Return View()
    End Function

End Class
```

As you can see from Listing 1, a controller is just a class (a Visual Basic .NET or C# class). A controller is a class that derives from the base System.Web.Mvc.Controller class. Because a controller inherits from this base class, a controller inherits several useful methods for free (We discuss these methods in a moment).

Understanding Controller Actions

A controller exposes controller actions. An action is a method on a controller that gets called when you enter a particular URL in your browser address bar. For example, imagine that you make a request for the following URL:

```
http://localhost/Product/Index/3
```

In this case, the Index() method is called on the ProductController class. The Index() method is an example of a controller action.

A controller action must be a public method of a controller class. Visual Basic.NET methods, by default, are public methods. Realize that any public method that you add to a controller class is exposed as a controller action automatically (You must be careful about this since a controller action can be invoked by anyone in the universe simply by typing the right URL into a browser address bar).

There are some additional requirements that must be satisfied by a controller action. A method used as a controller action cannot be overloaded. Furthermore, a controller action cannot be a static method. Other than that, you can use just about any method as a controller action.

Understanding Action Results

A controller action returns something called an *action result*. An action result is what a controller action returns in response to a browser request.

The ASP.NET MVC framework supports several types of action results including:

1. ViewResult - Represents HTML and markup.
2. EmptyResult - Represents no result.
3. RedirectResult - Represents a redirection to a new URL.
4. JsonResult - Represents a JavaScript Object Notation result that can be used in an AJAX application.
5. JavaScriptResult - Represents a JavaScript script.
6. ContentResult - Represents a text result.
7. FileContentResult - Represents a downloadable file (with the binary content).
8. FilePathResult - Represents a downloadable file (with a path).
9. FileStreamResult - Represents a downloadable file (with a file stream).

All of these action results inherit from the base ActionResult class.

In most cases, a controller action returns a ViewResult. For example, the Index controller action in Listing 2 returns a ViewResult.

Listing 2 - Controllers\BookController.vb

```
Public Class BookController
    Inherits System.Web.Mvc.Controller

    Function Index()
        Return View()
    End Function

End Class
```

When an action returns a ViewResult, HTML is returned to the browser. The Index() method in Listing 2 returns a view named Index to the browser.

Notice that the Index() action in Listing 2 does not return a ViewResult(). Instead, the View() method of the Controller base class is called. Normally, you do not return an action result directly. Instead, you call one of the following methods of the Controller base class:

1. View - Returns a ViewResult action result.
2. Redirect - Returns a RedirectResult action result.
3. RedirectToAction - Returns a RedirectToRouteResult action result.
4. RedirectToRoute - Returns a RedirectToRouteResult action result.
5. Json - Returns a JsonResult action result.
6. JavaScriptResult - Returns a JavaScriptResult.

7. Content - Returns a ContentResult action result.
8. File - Returns a FileContentResult, FilePathResult, or FileStreamResult depending on the parameters passed to the method.

So, if you want to return a View to the browser, you call the View() method. If you want to redirect the user from one controller action to another, you call the RedirectToAction() method. For example, the Details() action in Listing 3 either displays a view or redirects the user to the Index() action depending on whether the Id parameter has a value.

Listing 3 - CustomerController.vb

```
Public Class CustomerController
    Inherits System.Web.Mvc.Controller

    Function Details(ByVal id As Integer?)
        If Not id.HasValue Then
            Return RedirectToAction("Index")
        End If

        Return View()
    End Function
    Function Index()
        Return View()
    End Function

End Class
```

The ContentResult action result is special. You can use the ContentResult action result to return an action result as plain text. For example, the Index() method in Listing 4 returns a message as plain text and not as HTML.

Listing 4 - Controllers\StatusController.vb

StatusController

System.Web.Mvc.Controller

```
Public Class StatusController
    Inherits System.Web.Mvc.Controller

    Function Index()
        Return Content("Hello World!")
    End Function

End Class
```

When the StatusController.Index() action is invoked, a view is not returned. Instead, the raw text "Hello World!" is returned to the browser.

If a controller action returns a result that is not an action result - for example, a date or an integer - then the result is wrapped in a ContentResult automatically. For example, when the Index() action of the WorkController in Listing 5 is invoked, the date is returned as a ContentResult automatically.

Listing 5 - WorkController.vb

```
Public Class WorkController
    Inherits System.Web.Mvc.Controller

    Function Index()
        Return DateTime.Now
    End Function

End Class
```

The Index() action in Listing 5 returns a DateTime object. The ASP.NET MVC framework converts the DateTime object to a string and wraps the DateTime value in a ContentResult automatically. The browser receives the date and time as plain text.

Summary

The purpose of this tutorial was to introduce you to the concepts of ASP.NET MVC controllers, controller actions, and controller action results. In the first section, you learned how to add new controllers to an ASP.NET MVC project. Next, you learned how public methods of a controller are exposed to the universe as controller actions. Finally, we discussed the different types of action results that can be returned from a controller action. In particular, we discussed how to return a ViewResult, RedirectToActionResult, and ContentResult from a controller action.

PREVIOUS NEXT

Creating Custom Routes (VB)

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Learn how to add custom routes to an ASP.NET MVC application. In this tutorial, you learn how to modify the default route table in the Global.asax file.

In this tutorial, you learn how to add a custom route to an ASP.NET MVC application. You learn how to modify the default route table in the Global.asax file with a custom route.

In ASP.NET MVC applications, the default route table will work just fine. However, you might discover that you have specialized routing needs. In that case, you can create a custom route.

Imagine, for example, that you are building a blog application. You might want to handle incoming requests that look like this:

/Archive/12-25-2009

When a user enters this request, you want to return the blog entry that corresponds to the date 12/25/2009. In order to handle this type of request, you need to create a custom route.

The Global.asax file in Listing 1 contains a new custom route, named Blog, which handles requests that look like /Archive/*entry date*.

Listing 1 - Global.asax (with custom route)

```
Public Class MvcApplication
    Inherits System.Web.HttpApplication
    Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}")
        ' MapRoute takes the following parameters, in order:
        ' (1) Route name
        ' (2) URL with parameters
        ' (3) Parameter defaults
        routes.MapRoute( _
            "Blog", _
            "Archive/{entryDate}", _
            New With {.controller = "Archive", .action = "Entry"} _
        )
        routes.MapRoute( _
            "Default", _
            "{controller}/{action}/{id}", _
            New With {.controller = "Home", .action = "Index", .id = ""} _
        )
    End Sub
    Sub Application_Start()
        RegisterRoutes(RouteTable.Routes)
    End Sub
End Class
```

The order of the routes that you add to the route table is important. Our new custom Blog route is added before the existing Default route. If you reversed the order, then the Default route always will get called instead of the custom route.

The custom Blog route matches any request that starts with /Archive/. So, it matches all of the following URLs:

- /Archive/12-25-2009
- /Archive/10-6-2004
- /Archive/apple

The custom route maps the incoming request to a controller named Archive and invokes the Entry() action. When the Entry() method is called, the entry date is passed as a parameter named entryDate.

You can use the Blog custom route with the controller in Listing 2.

Listing 2 - ArchiveController.vb

```
Public Class ArchiveController
    Inherits System.Web.Mvc.Controller
    Function Entry(ByVal entryDate As DateTime)
        Return "You requested the entry from " & entryDate.ToString()
    End Function
End Class
```

Notice that the Entry() method in Listing 2 accepts a parameter of type DateTime. The MVC framework is smart enough to convert the entry date from the URL into a DateTime value automatically. If the entry date parameter from the URL cannot be converted to a DateTime, an error is raised (see Figure 1).

Figure 1 - Error from converting parameter

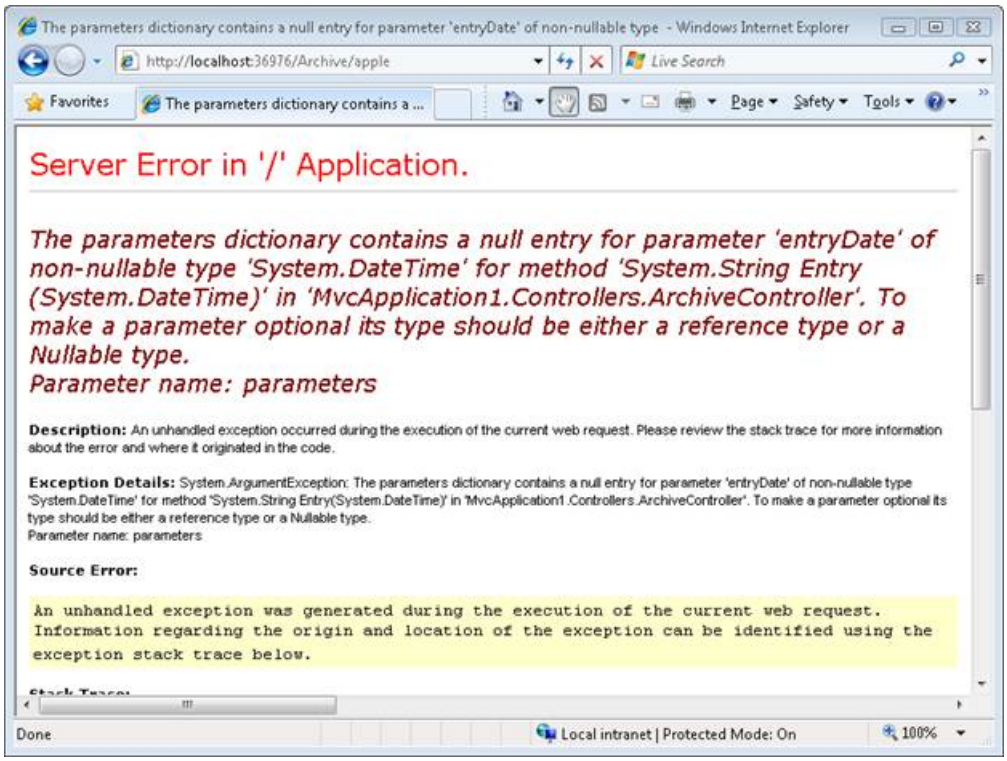


Figure 01: Error from converting parameter (Click to view full-size image)

Summary

The goal of this tutorial was to demonstrate how you can create a custom route. You learned how to add a custom route to the route table in the Global.asax file that represents blog entries. We discussed how to map requests for blog entries to a controller named ArchiveController and a controller action named Entry().

PREVIOUS NEXT

Creating a Route Constraint (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

> In this tutorial, Stephen Walther demonstrates how you can control how browser requests match routes by creating route constraints with regular expressions.

You use route constraints to restrict the browser requests that match a particular route. You can use a regular expression to specify a route constraint.

For example, imagine that you have defined the route in Listing 1 in your Global.asax file.

Listing 1 - Global.asax.vb

```
routes.MapRoute( _
    "Product", _
    "Product/{productId}", _
    New With {.controller = "Product", .action = "Details"} _
)
```

Listing 1 contains a route named Product. You can use the Product route to map browser requests to the ProductController contained in Listing 2.

Listing 2 - Controllers\ProductController.vb

```
Public Class ProductController
    Inherits System.Web.Mvc.Controller
    Function Details(ByVal productId As Integer) As ActionResult
        Return View()
    End Function
End Class
```

Notice that the Details() action exposed by the Product controller accepts a single parameter named productId. This parameter is an integer parameter.

The route defined in Listing 1 will match any of the following URLs:

- /Product/23
- /Product/7

Unfortunately, the route will also match the following URLs:

- /Product/blah
- /Product/apple

Because the Details() action expects an integer parameter, making a request that contains something other than an integer value will cause an error. For example, if you type the URL /Product/apple into your browser then you will get the error page in Figure 1.

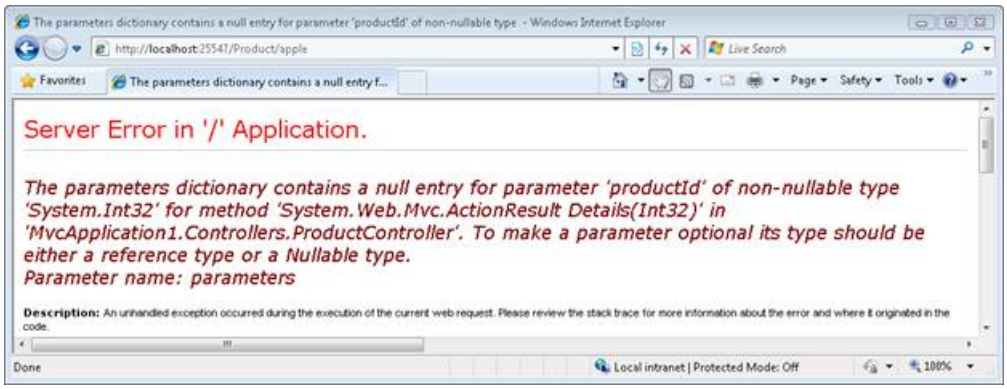


Figure 01: Seeing a page explode (Click to view full-size image)

What you really want to do is only match URLs that contain a proper integer productId. You can use a constraint when defining a route to restrict the URLs that match the route. The modified Product route in Listing 3 contains a regular expression constraint that only matches integers.

Listing 3 - Global.asax.vb

```
routes.MapRoute( _
   "Product", _
   "Product/{productId}", _
   New With {.controller = "Product", .action = "Details"}, _
   New With {.productId = "\d+"} _
)
```

The regular expression \d+ matches one or more integers. This constraint causes the Product route to match the following URLs:

- /Product/3
- /Product/8999

But not the following URLs:

- /Product/apple
- /Product

These browser requests will be handled by another route or, if there are no matching routes, a *The resource could not be found* error will be returned.

PREVIOUS NEXT

Creating a Custom Route Constraint (VB)

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

> Stephen Walther demonstrates how you can create a custom route constraint. We implement a simple custom constraint that prevents a route from being matched when a browser request is made from a remote computer.

The goal of this tutorial is to demonstrate how you can create a custom route constraint. A custom route constraint enables you to prevent a route from being matched unless some custom condition is matched.

In this tutorial, we create a Localhost route constraint. The Localhost route constraint only matches requests made from the local computer. Remote requests from across the Internet are not matched.

You implement a custom route constraint by implementing the IRouteConstraint interface. This is an extremely simple interface which describes a single method:

```
Function Match ( _
    httpContext As HttpContextBase, _
    route As Route, _
    parameterName As String, _
    values As RouteValueDictionary, _
    routeDirection As RouteDirection _
) As Boolean
```

The method returns a Boolean value. If you return False, the route associated with the constraint won't match the browser request.

The Localhost constraint is contained in Listing 1.

Listing 1 - LocalhostConstraint.vb

```
Public Class LocalhostConstraint
    Implements IRouteConstraint
    Public Function Match( _
                ByVal httpContext As HttpContextBase, _
                ByVal route As Route, _
                ByVal parameterName As String, _
                ByVal values As RouteValueDictionary, _
                ByVal routeDirection As RouteDirection _
            ) As Boolean Implements IRouteConstraint.Match
        Return httpContext.Request.IsLocal
    End Function
End Class
```

The constraint in Listing 1 takes advantage of the IsLocal property exposed by the HttpRequest class. This property returns true when the IP address of the request is either 127.0.0.1 or when the IP of the request is the same as the server's IP address.

You use a custom constraint within a route defined in the Global.asax file. The Global.asax file in Listing 2 uses the Localhost constraint to prevent anyone from requesting an Admin page unless they make the request from the local server. For example, a request for /Admin/DeleteAll will fail when made from a remote server.

Listing 2 - Global.asax

```
Public Class MvcApplication
    Inherits System.Web.HttpApplication
    Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}")
        routes.MapRoute( _
            "Admin", _
            "Admin/{action}", _
            New With {.controller = "Admin"}, _
            New With {.isLocal = New LocalhostConstraint()} _
        )
        'routes.MapRoute( _
        '    "Default", _
        '    "{controller}/{action}/{id}", _
        '    New With {.controller = "Home", .action = "Index", .id = ""} _
        ')
    End Sub
    Sub Application_Start()
        RegisterRoutes(RouteTable.Routes)
    End Sub
End Class
```

The Localhost constraint is used in the definition of the Admin route. This route won't be matched by a remote browser request. Realize, however, that other routes defined in Global.asax might match the same request. It is important to understand that a constraint prevents a particular route from matching a request and not all routes defined in the Global.asax file.

Notice that the Default route has been commented out from the Global.asax file in Listing 2. If you include the Default route, then the Default route would match requests for the Admin controller. In that case, remote users could still invoke actions of the Admin controller even though their requests wouldn't match the Admin route.

PREVIOUS

Deployment

2/20/2020 • 2 minutes to read • Edit Online

In this tutorial you learn how to use ASP.NET MVC and routing with different versions of IIS.

- Using ASP.NET MVC with Different Versions of IIS (C#)
- Using ASP.NET MVC with Different Versions of IIS (VB)

Using ASP.NET MVC with Different Versions of IIS (C#)

4/10/2020 • 11 minutes to read • Edit Online

by Microsoft

> In this tutorial, you learn how to use ASP.NET MVC, and URL Routing, with different versions of Internet Information Services. You learn different strategies for using ASP.NET MVC with IIS 7.0 (classic mode), IIS 6.0, and earlier versions of IIS.

The ASP.NET MVC framework depends on ASP.NET Routing to route browser requests to controller actions. In order to take advantage of ASP.NET Routing, you might have to perform additional configuration steps on your web server. It all depends on the version of Internet Information Services (IIS) and the request processing mode for your application.

Here's a summary of the different versions of IIS:

- IIS 7.0 (integrated mode) - No special configuration necessary to use ASP.NET Routing.
- IIS 7.0 (classic mode) - You need to perform special configuration to use ASP.NET Routing.
- IIS 6.0 or below - You need to perform special configuration to use ASP.NET Routing.

The latest version of IIS is version 7.5 (on Win7). IIS 7 of IIS is included with Windows Server 2008 AND VISTA/SP1 and higher. You also can install IIS 7.0 on any version of the Vista operating system except Home Basic (see https://technet.microsoft.com/library/cc731179%28WS.10%29.aspx).

IIS 7.0 supports two modes for processing requests. You can use integrated mode or classic mode. You don't need to perform any special configuration steps when using IIS 7.0 in integrated mode. However, you do need to perform additional configuration when using IIS 7.0 in classic mode.

Microsoft Windows Server 2003 includes IIS 6.0. You cannot upgrade IIS 6.0 to IIS 7.0 when using the Windows Server 2003 operating system. You must perform additional configuration steps when using IIS 6.0.

Microsoft Windows XP Professional includes IIS 5.1. You must perform additional configuration steps when using IIS 5.1.

Finally, Microsoft Windows 2000 and Microsoft Windows 2000 Professional includes IIS 5.0. You must perform additional configuration steps when using IIS 5.0.

Integrated versus Classic Mode

IIS 7.0 can process requests using two different request processing modes: integrated and classic. Integrated mode provides better performance and more features. Classic mode is included for backwards compatibility with earlier versions of IIS.

The request processing mode is determined by the application pool. You can determine which processing mode is being used by a particular web application by determining the application pool associated with the application. Follow these steps:

1. Launch the Internet Information Services Manager
2. In the Connections window, select an application
3. In the Actions window, click the **Basic Settings** link to open the Edit Application dialog box (see Figure 1)

4. Take note of the Application pool selected.

By default, IIS is configured to support two application pools: **DefaultAppPool** and **Classic .NET AppPool**. If DefaultAppPool is selected, then your application is running in integrated request processing mode. If Classic .NET AppPool is selected, your application is running in classic request processing mode.

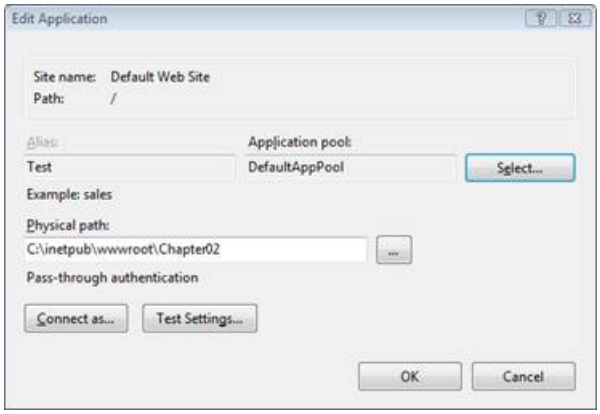


Figure 1: Detecting the request processing mode(Click to view full-size image)

Notice that you can modify the request processing mode within the Edit Application dialog box. Click the Select button and change the application pool associated with the application. Realize that there are compatibility issues when changing an ASP.NET application from classic to integrated mode. For more information, see the following articles:

- Upgrading ASP.NET 1.1 to IIS 7.0 on Windows Vista and Windows Server 2008 -- https://www.iis.net/learn/application-frameworks/building-and-running-aspnet-applications/upgrading-aspnet-11-to-iis-on-windows-vista-and-windows-server-2008
- ASP.NET Integration With IIS 7.0 - https://www.iis.net/learn/application-frameworks/building-and-running-aspnet-applications/aspnet-integration-with-iis

If an ASP.NET application is using the DefaultAppPool, then you don't need to perform any additional steps to get ASP.NET Routing (and therefore ASP.NET MVC) to work. However, if the ASP.NET application is configured to use the Classic .NET AppPool then keep reading, you have more work to do.

Using ASP.NET MVC with Older Versions of IIS

If you need to use ASP.NET MVC with an older version of IIS than IIS 7.0, or you need to use IIS 7.0 in classic mode, then you have two options. First, you can modify the route table to use file extensions. For example, instead of requesting a URL like /Store/Details, you would request a URL like /Store.aspx/Details.

The second option is to create something called a *wildcard script map*. A wildcard script map enables you to map every request into the ASP.NET framework.

If you don't have access to your web server (for example, your ASP.NET MVC application is being hosted by an Internet Service Provider) then you'll need to use the first option. If you don't want to modify the appearance of your URLs, and you have access to your web server, then you can use the second option.

We explore each option in detail in the following sections.

Adding Extensions to the Route Table

The easiest way to get ASP.NET Routing to work with older versions of IIS is to modify your route table in the Global.asax file. The default and unmodified Global.asax file in Listing 1 configures one route named the Default route.

Listing 1 - Global.asax (unmodified)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcApplication1
{
    public class GlobalApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default",                                          // Route name
                "{controller}/{action}/{id}",                       // URL with parameters
                new { controller = "Home", action = "Index", id = "" }  // Parameter defaults
            );
        }

        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

The Default route configured in Listing 1 enables you to route URLs that look like this:

/Home/Index

/Product/Details/3

/Product

Unfortunately, older versions of IIS won't pass these requests to the ASP.NET framework. Therefore, these requests won't get routed to a controller. For example, if you make a browser request for the URL /Home/Index then you'll get the error page in Figure 2.

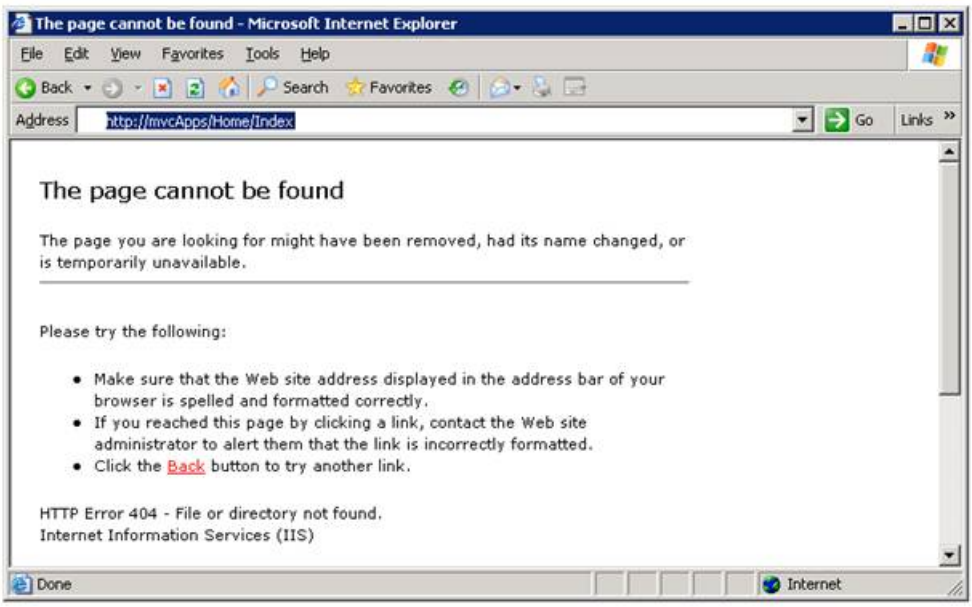


Figure 2: Receiving a 404 Not Found error(Click to view full-size image)

Older versions of IIS only map certain requests to the ASP.NET framework. The request must be for a URL with the

right file extension. For example, a request for /SomePage.aspx gets mapped to the ASP.NET framework. However, a request for /SomePage.htm does not.

Therefore, to get ASP.NET Routing to work, we must modify the Default route so that it includes a file extension that is mapped to the ASP.NET framework.

This is done using a script named `registermvc.wsf`. It was included with the ASP.NET MVC 1 release in `C:\Program Files\Microsoft ASP.NET\ASP.NET MVC\Scripts`, but as of ASP.NET 2 this script has been moved to the ASP.NET Futures, available at http://aspnet.codeplex.com/releases/view/39978.

Executing this script registers a new .mvc extension with IIS. After you register the .mvc extension, you can modify your routes in the Global.asax file so that the routes use the .mvc extension.

The modified Global.asax file in Listing 2 works with older versions of IIS.

Listing 2 - Global.asax (modified with extensions)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcApplication1
{
    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default",
                "{controller}.mvc/{action}/{id}",
                new { action = "Index", id = "" }
              );

            routes.MapRoute(
              "Root",
              "",
              new { controller = "Home", action = "Index", id = "" }
            );
        }

        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

Important: remember to build your ASP.NET MVC Application again after changing the Global.asax file.

There are two important changes to the Global.asax file in Listing 2. There are now two routes defined in the Global.asax. The URL pattern for the Default route, the first route, now looks like:

{controller}.mvc/{action}/{id}

The addition of the .mvc extension changes the type of files that the ASP.NET Routing module intercepts. With this change, the ASP.NET MVC application now routes requests like the following:

/Home.mvc/Index/

/Product.mvc/Details/3

/Product.mvc/

The second route, the Root route, is new. This URL pattern for the Root route is an empty string. This route is necessary for matching requests made against the root of your application. For example, the Root route will match a request that looks like this:

http://www.YourApplication.com/

After making these modifications to your route table, you'll need to make sure that all of the links in your application are compatible with these new URL patterns. In other words, make sure that all of your links include the .mvc extension. If you use the Html.ActionLink() helper method to generate your links, then you should not need to make any changes.

Instead of using the registermvc.wcf script, you can add a new extension to IIS that is mapped to the ASP.NET framework by hand. When adding a new extension yourself, make sure that the checkbox labeled **Verify that file exists** is not checked.

Hosted Server

You don't always have access to your web server. For example, if you are hosting your ASP.NET MVC application using an Internet Hosting Provider, then you won't necessarily have access to IIS.

In that case, you should use one of the existing file extensions that are mapped to the ASP.NET framework. Examples of file extensions mapped to ASP.NET include the .aspx, .axd, and .ashx extensions.

For example, the modified Global.asax file in Listing 3 uses the .aspx extension instead of the .mvc extension.

Listing 3 - Global.asax (modified with .aspx extensions)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcApplication1
{
    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default",
                "{controller}.aspx/{action}/{id}",
                new { action = "Index", id = "" }
              );

            routes.MapRoute(
              "Root",
              "",
              new { controller = "Home", action = "Index", id = "" }
            );
        }

        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

The Global.asax file in Listing 3 is exactly the same as the previous Global.asax file except for the fact that it uses the .aspx extension instead of the .mvc extension. You don't have to perform any setup on your remote web server to use the .aspx extension.

Creating a Wildcard Script Map

If you don't want to modify the URLs for your ASP.NET MVC application, and you have access to your web server, then you have an additional option. You can create a wildcard script map that maps all requests to the web server to the ASP.NET framework. That way, you can use the default ASP.NET MVC route table with IIS 7.0 (in classic mode) or IIS 6.0.

Be aware that this option causes IIS to intercept every request made against the web server. This includes requests for images, classic ASP pages, and HTML pages. Therefore, enabling a wildcard script map to ASP.NET does have performance implications.

Here's how you enable a wildcard script map for IIS 7.0:

1. Select your application in the Connections window
2. Make sure that the **Features** view is selected
3. Double-click the **Handler Mappings** button
4. Click the **Add Wildcard Script Map** link (see Figure 3)
5. Enter the path to the aspnet_isapi.dll file (You can copy this path from the PageHandlerFactory script map)
6. Enter the name MVC
7. Click the **OK** button

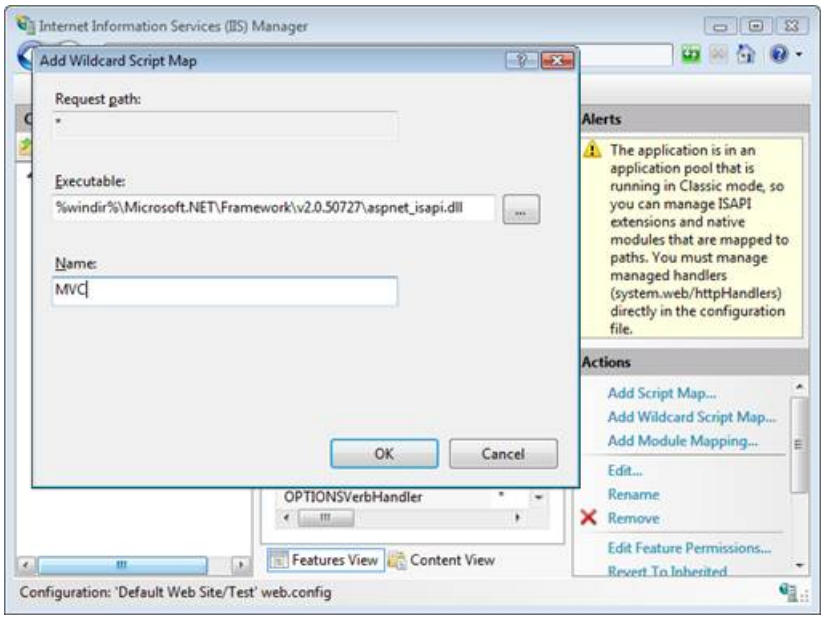


Figure 3: Creating a wildcard script map with IIS 7.0(Click to view full-size image)

Follow these steps to create a wildcard script map with IIS 6.0:

1. Right-click a website and select Properties
2. Select the **Home Directory** tab
3. Click the **Configuration** button
4. Select the **Mappings** tab
5. Click the **Insert** button (see Figure 4)
6. Paste the path to the aspnet_isapi.dll into the Executable field (you can copy this path from the script map for .aspx files)
7. Uncheck the checkbox labeled **Verify that file exists**
8. Click the **OK** button

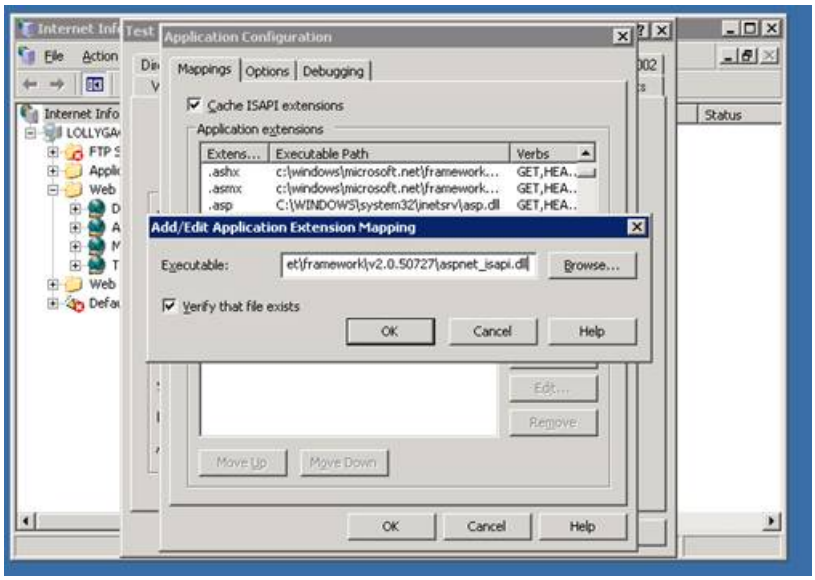


Figure 4: Creating a wildcard script map with IIS 6.0(Click to view full-size image)

After you enable wildcard script maps, you need to modify the route table in the Global.asax file so that it includes a Root route. Otherwise, you'll get the error page in Figure 5 when you make a request for the root page of your application. You can use the modified Global.asax file in Listing 4.

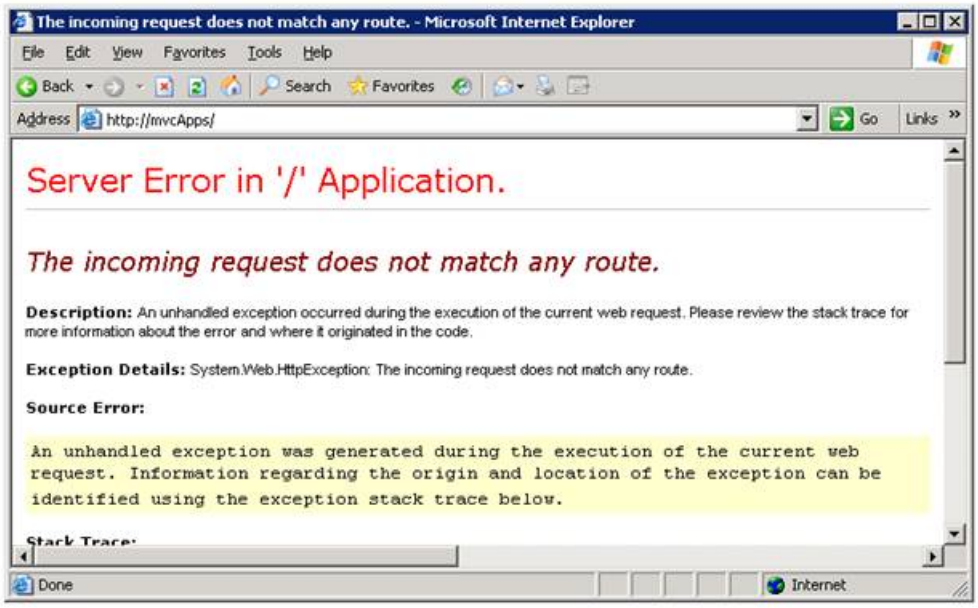


Figure 5: Missing Root route error(Click to view full-size image)

Listing 4 - Global.asax (modified with Root route)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcApplication1
{

    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default",
                "{controller}/{action}/{id}",
                new { action = "Index", id = "" }
              );

            routes.MapRoute(
              "Root",
              "",
              new { controller = "Home", action = "Index", id = "" }
            );

        }

        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

After you enable a wildcard script map for either IIS 7.0 or IIS 6.0, you can make requests that work with the default route table that look like this:

/

/Home/Index

/Product/Details/3

/Product

Summary

The goal of this tutorial was to explain how you can use ASP.NET MVC when using an older version of IIS (or IIS 7.0 in classic mode). We discussed two methods of getting ASP.NET Routing to work with older versions of IIS: Modify the default route table or create a wildcard script map.

The first option requires you to modify the URLs used in your ASP.NET MVC application. One very significant advantage of this first option is that you do not need access to a web server in order to modify the route table. That means that you can use this first option even when hosting your ASP.NET MVC application with an Internet hosting company.

The second option is to create a wildcard script map. The advantage of this second option is that you do not need to modify your URLs. The disadvantage of this second option is that it can impact the performance of your ASP.NET MVC application.

NEXT

Using ASP.NET MVC with Different Versions of IIS (VB)

4/10/2020 • 11 minutes to read • Edit Online

by Microsoft

> In this tutorial, you learn how to use ASP.NET MVC, and URL Routing, with different versions of Internet Information Services. You learn different strategies for using ASP.NET MVC with IIS 7.0 (classic mode), IIS 6.0, and earlier versions of IIS.

The ASP.NET MVC framework depends on ASP.NET Routing to route browser requests to controller actions. In order to take advantage of ASP.NET Routing, you might have to perform additional configuration steps on your web server. It all depends on the version of Internet Information Services (IIS) and the request processing mode for your application.

Here's a summary of the different versions of IIS:

- IIS 7.0 (integrated mode) - No special configuration necessary to use ASP.NET Routing.
- IIS 7.0 (classic mode) - You need to perform special configuration to use ASP.NET Routing.
- IIS 6.0 or below - You need to perform special configuration to use ASP.NET Routing.

The latest version of IIS is version 7.5 (on Win7). IIS 7 of IIS is included with Windows Server 2008 AND VISTA/SP1 and higher. You also can install IIS 7.0 on any version of the Vista operating system except Home Basic (see https://technet.microsoft.com/library/cc731179%28WS.10%29.aspx).

IIS 7.0 supports two modes for processing requests. You can use integrated mode or classic mode. You don't need to perform any special configuration steps when using IIS 7.0 in integrated mode. However, you do need to perform additional configuration when using IIS 7.0 in classic mode.

Microsoft Windows Server 2003 includes IIS 6.0. You cannot upgrade IIS 6.0 to IIS 7.0 when using the Windows Server 2003 operating system. You must perform additional configuration steps when using IIS 6.0.

Microsoft Windows XP Professional includes IIS 5.1. You must perform additional configuration steps when using IIS 5.1.

Finally, Microsoft Windows 2000 and Microsoft Windows 2000 Professional includes IIS 5.0. You must perform additional configuration steps when using IIS 5.0.

Integrated versus Classic Mode

IIS 7.0 can process requests using two different request processing modes: integrated and classic. Integrated mode provides better performance and more features. Classic mode is included for backwards compatibility with earlier versions of IIS.

The request processing mode is determined by the application pool. You can determine which processing mode is being used by a particular web application by determining the application pool associated with the application. Follow these steps:

1. Launch the Internet Information Services Manager
2. In the Connections window, select an application
3. In the Actions window, click the **Basic Settings** link to open the Edit Application dialog box (see Figure 1)

4. Take note of the Application pool selected.

By default, IIS is configured to support two application pools: **DefaultAppPool** and **Classic .NET AppPool**. If DefaultAppPool is selected, then your application is running in integrated request processing mode. If Classic .NET AppPool is selected, your application is running in classic request processing mode.

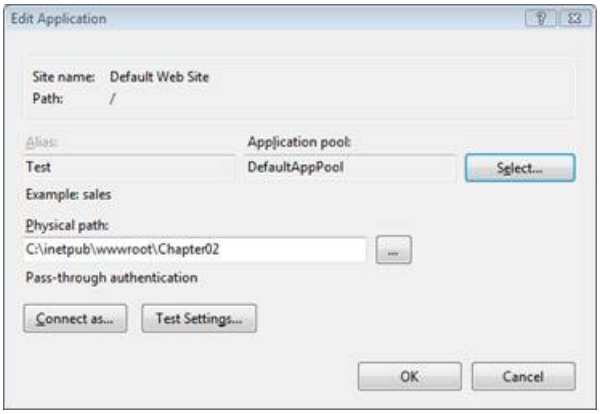


Figure 1: Detecting the request processing mode(Click to view full-size image)

Notice that you can modify the request processing mode within the Edit Application dialog box. Click the Select button and change the application pool associated with the application. Realize that there are compatibility issues when changing an ASP.NET application from classic to integrated mode. For more information, see the following articles:

- Upgrading ASP.NET 1.1 to IIS 7.0 on Windows Vista and Windows Server 2008 -- https://www.iis.net/learn/application-frameworks/building-and-running-aspnet-applications/upgrading-aspnet-11-to-iis-on-windows-vista-and-windows-server-2008
- ASP.NET Integration With IIS 7.0 - https://www.iis.net/learn/application-frameworks/building-and-running-aspnet-applications/aspnet-integration-with-iis

If an ASP.NET application is using the DefaultAppPool, then you don't need to perform any additional steps to get ASP.NET Routing (and therefore ASP.NET MVC) to work. However, if the ASP.NET application is configured to use the Classic .NET AppPool then keep reading, you have more work to do.

Using ASP.NET MVC with Older Versions of IIS

If you need to use ASP.NET MVC with an older version of IIS than IIS 7.0, or you need to use IIS 7.0 in classic mode, then you have two options. First, you can modify the route table to use file extensions. For example, instead of requesting a URL like /Store/Details, you would request a URL like /Store.aspx/Details.

The second option is to create something called a *wildcard script map*. A wildcard script map enables you to map every request into the ASP.NET framework.

If you don't have access to your web server (for example, your ASP.NET MVC application is being hosted by an Internet Service Provider) then you'll need to use the first option. If you don't want to modify the appearance of your URLs, and you have access to your web server, then you can use the second option.

We explore each option in detail in the following sections.

Adding Extensions to the Route Table

The easiest way to get ASP.NET Routing to work with older versions of IIS is to modify your route table in the Global.asax file. The default and unmodified Global.asax file in Listing 1 configures one route named the Default route.

Listing 1 - Global.asax (unmodified)

```
Public Class MvcApplication

        Inherits System.Web.HttpApplication

        Shared Sub RegisterRoutes(ByVal routes As RouteCollection)

            routes.IgnoreRoute("{resource}.axd/{*pathInfo}")

            ' MapRoute takes the following parameters, in order:

            ' (1) Route name

            ' (2) URL with parameters

            ' (3) Parameter defaults

            routes.MapRoute( _

                "Default", _

                "{controller}/{action}/{id}", _

                New With {.controller = "Home", .action = "Index", .id = ""} _

            )

        End Sub

        Sub Application_Start()

            RegisterRoutes(RouteTable.Routes)

        End Sub

    End Class
```

The Default route configured in Listing 1 enables you to route URLs that look like this:

/Home/Index

/Product/Details/3

/Product

Unfortunately, older versions of IIS won't pass these requests to the ASP.NET framework. Therefore, these requests won't get routed to a controller. For example, if you make a browser request for the URL /Home/Index then you'll get the error page in Figure 2.

Figure 2: Receiving a 404 Not Found error(Click to view full-size image)

Older versions of IIS only map certain requests to the ASP.NET framework. The request must be for a URL with the right file extension. For example, a request for /SomePage.aspx gets mapped to the ASP.NET framework. However, a request for /SomePage.htm does not.

Therefore, to get ASP.NET Routing to work, we must modify the Default route so that it includes a file extension that is mapped to the ASP.NET framework.

This is done using a script named `registermvc.wsf`. It was included with the ASP.NET MVC 1 release in `C:\Program Files\Microsoft ASP.NET\ASP.NET MVC\Scripts`, but as of ASP.NET 2 this script has been moved to the ASP.NET Futures, available at http://aspnet.codeplex.com/releases/view/39978.

Executing this script registers a new .mvc extension with IIS. After you register the .mvc extension, you can modify your routes in the Global.asax file so that the routes use the .mvc extension.

The modified Global.asax file in Listing 2 works with older versions of IIS.

Listing 2 - Global.asax (modified with extensions)

```
Public Class MvcApplication

        Inherits System.Web.HttpApplication

        Shared Sub RegisterRoutes(ByVal routes As RouteCollection)

            routes.IgnoreRoute("{resource}.axd/{*pathInfo}")

            ' MapRoute takes the following parameters, in order:

            ' (1) Route name

            ' (2) URL with parameters

            ' (3) Parameter defaults

            routes.MapRoute( _

                "Default", _

                "{controller}.mvc/{action}/{id}", _

                New With {.controller = "Home", .action = "Index", .id = ""} _

            )

            routes.MapRoute( _

                "Root", _

                "", _

                New With {.controller = "Home", .action = "Index", .id = ""} _

            )

        End Sub

        Sub Application_Start()

            RegisterRoutes(RouteTable.Routes)

        End Sub

    End Class
```

Important: remember to build your ASP.NET MVC Application again after changing the Global.asax file.

There are two important changes to the Global.asax file in Listing 2. There are now two routes defined in the Global.asax. The URL pattern for the Default route, the first route, now looks like:

{controller}.mvc/{action}/{id}

The addition of the .mvc extension changes the type of files that the ASP.NET Routing module intercepts. With this change, the ASP.NET MVC application now routes requests like the following:

/Home.mvc/Index/

/Product.mvc/Details/3

/Product.mvc/

The second route, the Root route, is new. This URL pattern for the Root route is an empty string. This route is necessary for matching requests made against the root of your application. For example, the Root route will match a request that looks like this:

http://www.YourApplication.com/

After making these modifications to your route table, you'll need to make sure that all of the links in your application are compatible with these new URL patterns. In other words, make sure that all of your links include the .mvc extension. If you use the Html.ActionLink() helper method to generate your links, then you should not need to make any changes.

Instead of using the registermvc.wcf script, you can add a new extension to IIS that is mapped to the ASP.NET framework by hand. When adding a new extension yourself, make sure that the checkbox labeled **Verify that file exists** is not checked.

Hosted Server

You don't always have access to your web server. For example, if you are hosting your ASP.NET MVC application using an Internet Hosting Provider, then you won't necessarily have access to IIS.

In that case, you should use one of the existing file extensions that are mapped to the ASP.NET framework. Examples of file extensions mapped to ASP.NET include the .aspx, .axd, and .ashx extensions.

For example, the modified Global.asax file in Listing 3 uses the .aspx extension instead of the .mvc extension.

Listing 3 - Global.asax (modified with .aspx extensions)

```
Public Class MvcApplication

        Inherits System.Web.HttpApplication

        Shared Sub RegisterRoutes(ByVal routes As RouteCollection)

            routes.IgnoreRoute("{resource}.axd/{*pathInfo}")

            ' MapRoute takes the following parameters, in order:

            ' (1) Route name

            ' (2) URL with parameters

            ' (3) Parameter defaults

            routes.MapRoute( _

                "Default", _

                "{controller}.aspx/{action}/{id}", _

                New With {.controller = "Home", .action = "Index", .id = ""} _

            )

            routes.MapRoute( _

                "Root", _

                "", _

                New With {.controller = "Home", .action = "Index", .id = ""} _

            )

        End Sub

        Sub Application_Start()

            RegisterRoutes(RouteTable.Routes)

        End Sub

    End Class
```

The Global.asax file in Listing 3 is exactly the same as the previous Global.asax file except for the fact that it uses the .aspx extension instead of the .mvc extension. You don't have to perform any setup on your remote web server to use the .aspx extension.

Creating a Wildcard Script Map

If you don't want to modify the URLs for your ASP.NET MVC application, and you have access to your web server, then you have an additional option. You can create a wildcard script map that maps all requests to the web server to the ASP.NET framework. That way, you can use the default ASP.NET MVC route table with IIS 7.0 (in classic mode) or IIS 6.0.

Be aware that this option causes IIS to intercept every request made against the web server. This includes requests for images, classic ASP pages, and HTML pages. Therefore, enabling a wildcard script map to ASP.NET does have performance implications.

Here's how you enable a wildcard script map for IIS 7.0:

1. Select your application in the Connections window
2. Make sure that the **Features** view is selected
3. Double-click the **Handler Mappings** button
4. Click the **Add Wildcard Script Map** link (see Figure 3)
5. Enter the path to the aspnet_isapi.dll file (You can copy this path from the PageHandlerFactory script map)
6. Enter the name MVC
7. Click the **OK** button

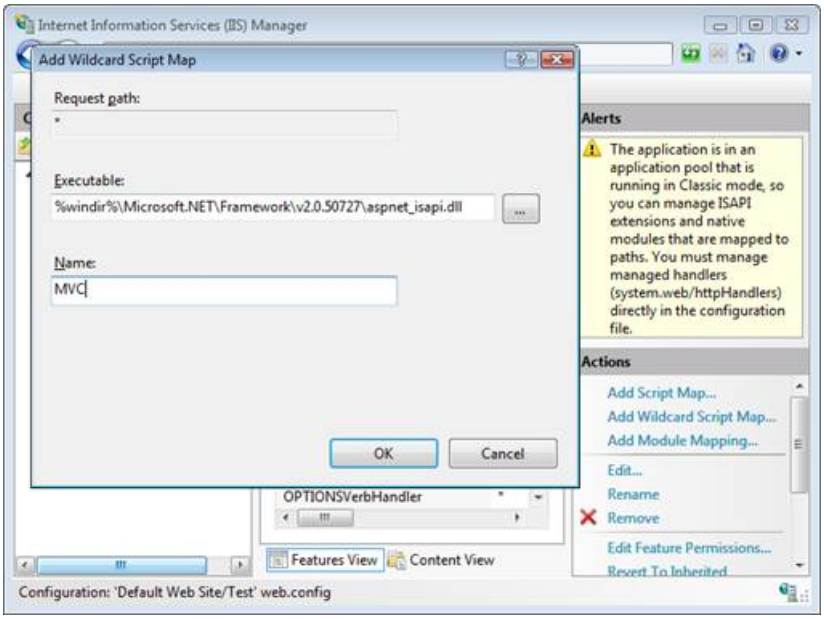


Figure 3: Creating a wildcard script map with IIS 7.0(Click to view full-size image)

Follow these steps to create a wildcard script map with IIS 6.0:

1. Right-click a website and select Properties
2. Select the **Home Directory** tab
3. Click the **Configuration** button
4. Select the **Mappings** tab
5. Click the **Insert** button (see Figure 4)
6. Paste the path to the aspnet_isapi.dll into the Executable field (you can copy this path from the script map for .aspx files)
7. Uncheck the checkbox labeled **Verify that file exists**
8. Click the **OK** button

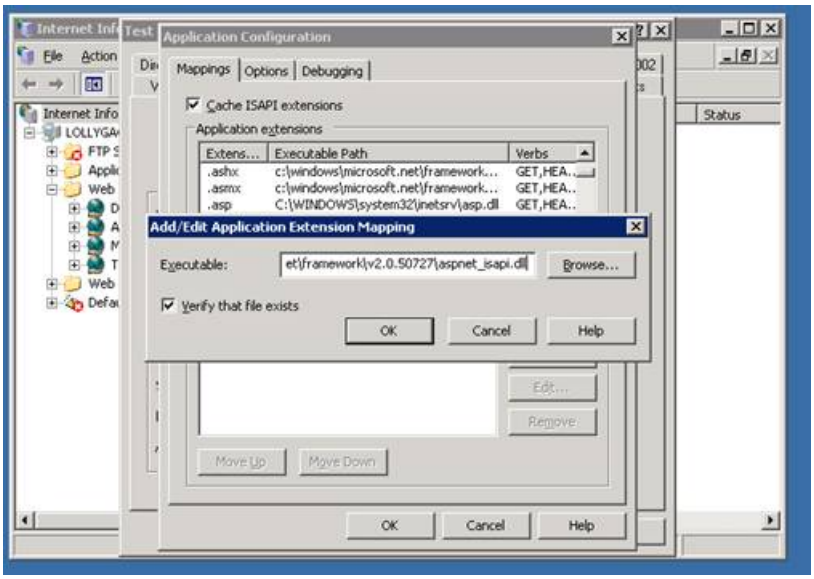


Figure 4: Creating a wildcard script map with IIS 6.0(Click to view full-size image)

After you enable wildcard script maps, you need to modify the route table in the Global.asax file so that it includes a Root route. Otherwise, you'll get the error page in Figure 5 when you make a request for the root page of your application. You can use the modified Global.asax file in Listing 4.

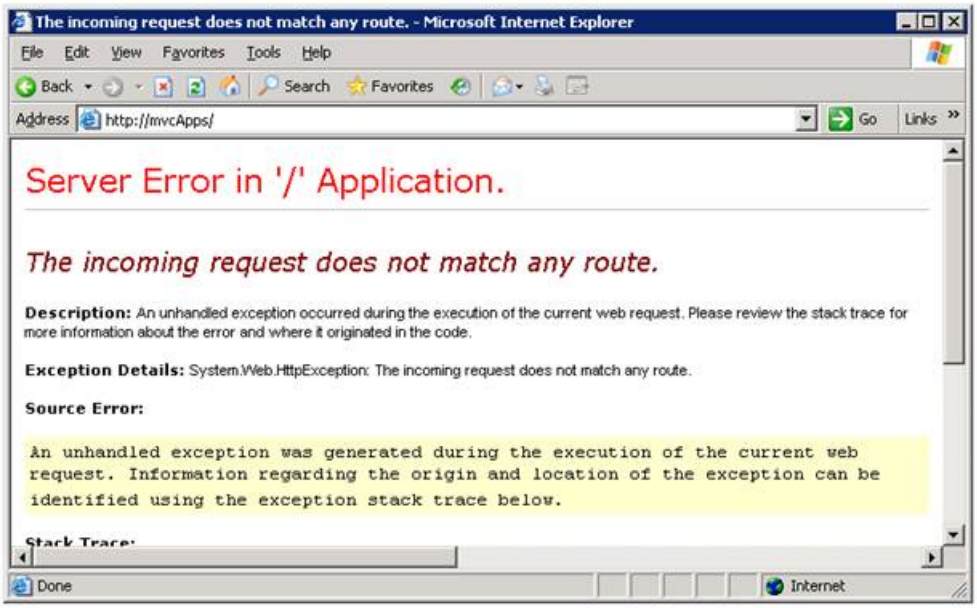


Figure 5: Missing Root route error(Click to view full-size image)

Listing 4 - Global.asax (modified with Root route)

```
Public Class MvcApplication

        Inherits System.Web.HttpApplication

        Shared Sub RegisterRoutes(ByVal routes As RouteCollection)

            routes.IgnoreRoute("{resource}.axd/{*pathInfo}")

            ' MapRoute takes the following parameters, in order:

            ' (1) Route name

            ' (2) URL with parameters

            ' (3) Parameter defaults

            routes.MapRoute( _

                "Default", _

                "{controller}/{action}/{id}", _

                New With {.controller = "Home", .action = "Index", .id = ""} _

            )

            routes.MapRoute( _

                "Root", _

                "", _

                New With {.controller = "Home", .action = "Index", .id = ""} _

            )

        End Sub

        Sub Application_Start()

            RegisterRoutes(RouteTable.Routes)

        End Sub

    End Class
```

After you enable a wildcard script map for either IIS 7.0 or IIS 6.0, you can make requests that work with the default route table that look like this:

/

/Home/Index

/Product/Details/3

/Product

Summary

The goal of this tutorial was to explain how you can use ASP.NET MVC when using an older version of IIS (or IIS 7.0 in classic mode). We discussed two methods of getting ASP.NET Routing to work with older versions of IIS: Modify the default route table or create a wildcard script map.

The first option requires you to modify the URLs used in your ASP.NET MVC application. One very significant

advantage of this first option is that you do not need access to a web server in order to modify the route table. That means that you can use this first option even when hosting your ASP.NET MVC application with an Internet hosting company.

The second option is to create a wildcard script map. The advantage of this second option is that you do not need to modify your URLs. The disadvantage of this second option is that it can impact the performance of your ASP.NET MVC application.

PREVIOUS

Security

2/20/2020 • 2 minutes to read • Edit Online

This tutorial set covers how to password protect some pages in your ASP.NET MVC application and how to create and manage users and roles.

- Authenticating Users with Forms Authentication (C#)
- Authenticating Users with Windows Authentication (C#)
- Preventing JavaScript Injection Attacks (C#)
- Authenticating Users with Forms Authentication (VB)
- Authenticating Users with Windows Authentication (VB)
- Preventing JavaScript Injection Attacks (VB)

Authenticating Users with Forms Authentication (C#)

4/10/2020 • 7 minutes to read • Edit Online

by Microsoft

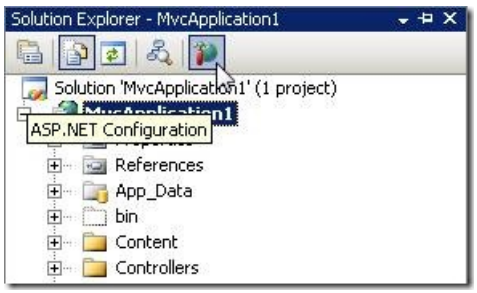
> Learn how to use the [Authorize] attribute to password protect particular pages in your MVC application. You learn how to use the Web Site Administration Tool to create and manage users and roles. You also learn how to configure where user account and role information is stored.

The goal of this tutorial is to explain how you can use Forms authentication to password protect the views in your ASP.NET MVC applications. You learn how to use the Web Site Administration Tool to create users and roles. You also learn how to prevent unauthorized users from invoking controller actions. Finally, you learn how to configure where user names and passwords are stored.

Using the Web Site Administration Tool

Before we do anything else, we should start by creating some users and roles. The easiest way to create new users and roles is to take advantage of the Visual Studio 2008 Web Site Administration Tool. You can launch this tool by selecting the menu option **Project, ASP.NET Configuration**. Alternatively, you can launch the Web Site Administration Tool by clicking the (somewhat scary) icon of the hammer hitting the world that appears at the top of the Solution Explorer window (see Figure 1).

Figure 1 – Launching the Web Site Administration Tool



Within the Web Site Administration Tool, you create new users and roles by selecting the Security tab. Click the **Create user** link to create a new user named Stephen (see Figure 2). Provide the Stephen user with any password that you want (for example, *secret*).

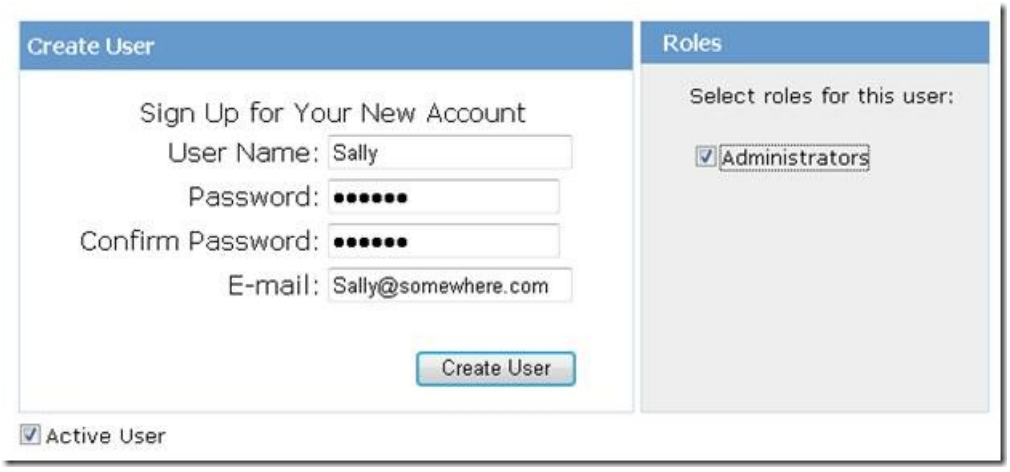
Figure 2 – Creating a new user

You create new roles by first enabling roles and defining one or more roles. Enable roles by clicking the **Enable roles** link. Next, create a role named *Administrators* by clicking the **Create or Manage roles** link (see Figure 3).

Figure 3 – Creating a new role

Finally, create a new user named Sally and associate Sally with the Administrators role by clicking the Create User link and selecting Administrators when creating Sally (see Figure 4).

Figure 4 – Adding a user to a role



When all is said and done, you should have two new users named Stephen and Sally. You should also have a new

role named Administrators. Sally is a member of the Administrators role and Stephen is not.

Requiring Authorization

You can require a user to be authenticated before the user invokes a controller action by adding the [Authorize] attribute to the action. You can apply the [Authorize] attribute to an individual controller action or you can apply this attribute to an entire controller class.

For example, the controller in Listing 1 exposes an action named CompanySecrets(). Because this action is decorated with the [Authorize] attribute, this action cannot be invoked unless a user is authenticated.

Listing 1 – Controllers\HomeController.cs

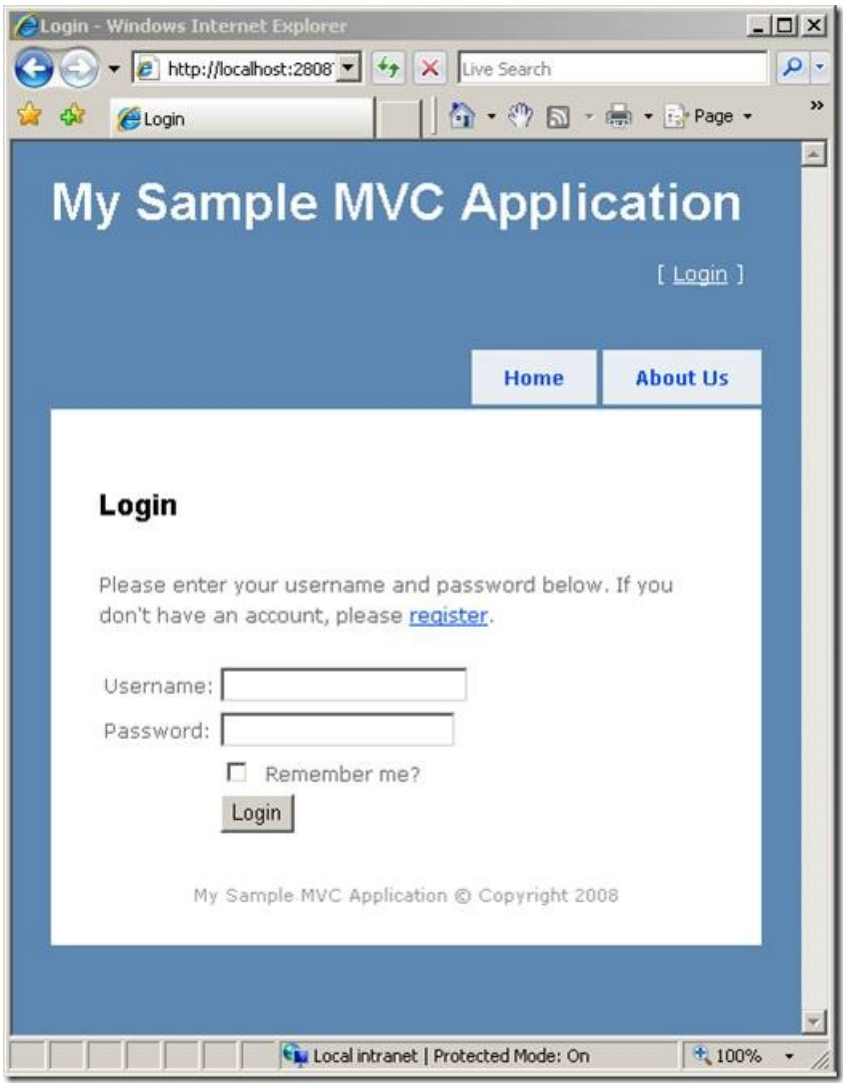
```
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        [Authorize]
        public ActionResult CompanySecrets()
        {
            return View();
        }
    }
}
```

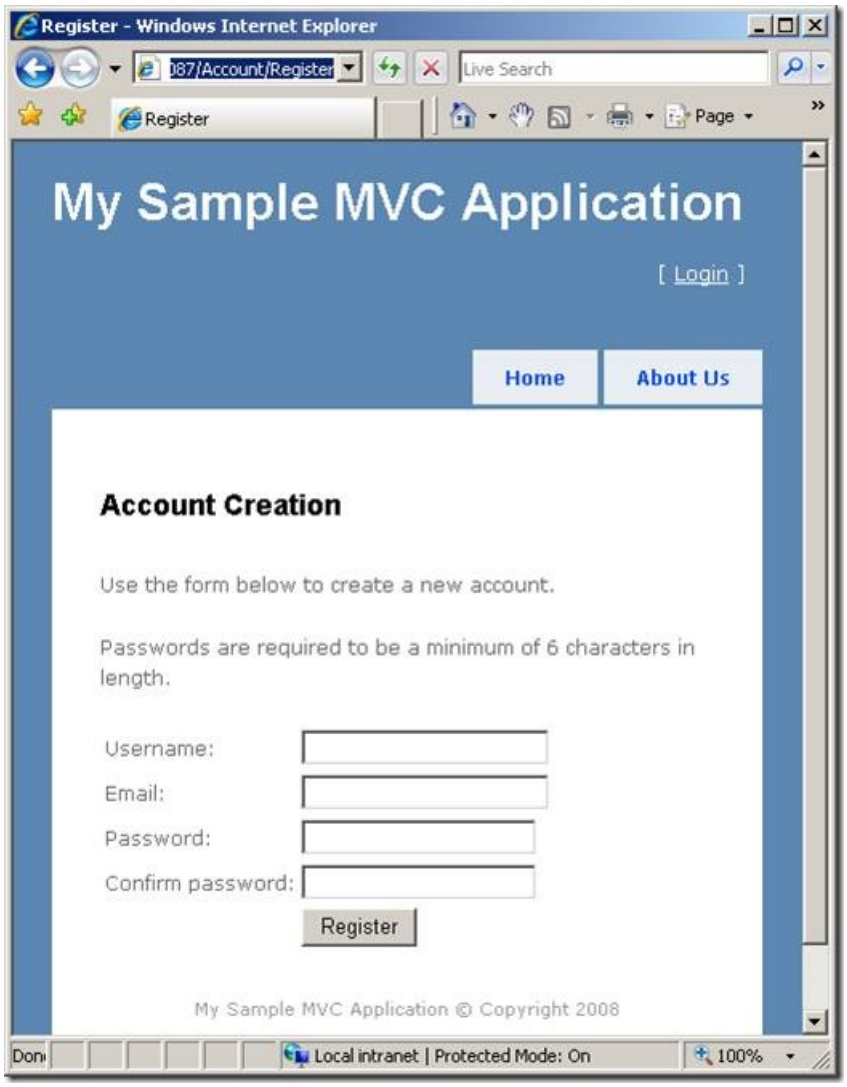
If you invoke the CompanySecrets() action by entering the URL /Home/CompanySecrets in the address bar of your browser, and you are not an authenticated user, then you will be redirected to the Login view automatically (see Figure 5).

Figure 5 – The Login view



You can use the Login view to enter your user name and password. If you are not a registered user then you can click the **register** link to navigate to the Register view (see Figure 6). You can use the Register view to create a new user account.

Figure 6 – The Register view



After you successfully log in, you can see the CompanySecrets view (see Figure 7). By default, you will continue to be logged in until you close your browser window.

Figure 7 – The CompanySecrets view

Authorizing by User Name or User Role

You can use the [Authorize] attribute to restrict access to a controller action to a particular set of users or a particular set of user roles. For example, the modified Home controller in Listing 2 contains two new actions named StephenSecrets() and AdministratorSecrets().

Listing 2 – Controllers\HomeController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Mvc.Ajax;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }


        [Authorize]
        public ActionResult CompanySecrets()
        {
            return View();
        }


        [Authorize(Users="Stephen")]
        public ActionResult StephenSecrets()
        {
            return View();
        }


        [Authorize(Roles = "Administrators")]
        public ActionResult AdministratorSecrets()
        {
            return View();
        }

    }
}
```

Only a user with the user name Stephen can invoke the StephenSecrets() action. All other users get redirected to the Login view. The Users property accepts a comma separated list of user account names.

Only users in the Administrators role can invoke the AdministratorSecrets() action. For example, because Sally is a member of the Administrators group, she can invoke the AdministratorSecrets() action. All other users get redirected to the Login view. The Roles property accepts a comma separated list of role names.

Configuring Authentication

At this point, you might be wondering where the user account and role information is being stored. By default, the information is stored in a (RANU) SQL Express database named ASPNETDB.mdf located in your MVC application's App_Data folder. This database is generated by the ASP.NET framework automatically when you start using membership.

In order to see the ASPNETDB.mdf database in the Solution Explorer window, you first need to select the menu option Project, Show All Files.

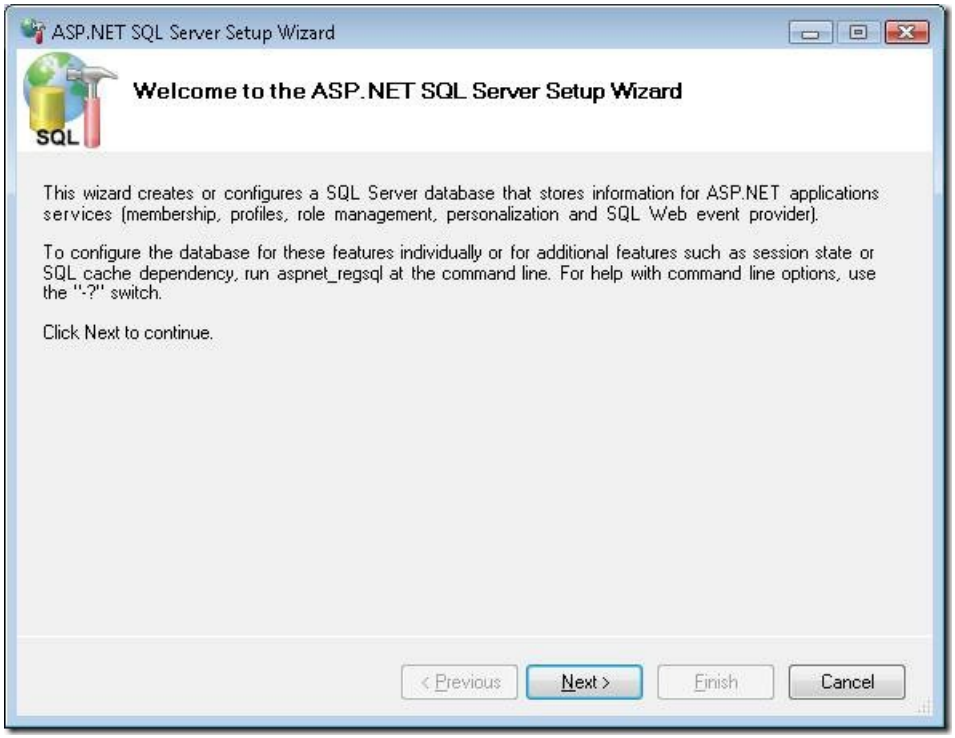
Using the default SQL Express database is fine when developing an application. Most likely, however, you won't want to use the default ASPNETDB.mdf database for a production application. In that case, you can change where user account information is stored by completing the following two steps:

1. Add the Application Services database objects to your production database - Change your application connection string to point to your production database

The first step is to add all of the necessary database objects (tables and stored procedures) to your production database. The easiest way to add these objects to a new database is to take advantage of the ASP.NET SQL Server Setup Wizard (see Figure 8). You can launch this tool by opening the Visual Studio 2008 Command Prompt from the Microsoft Visual Studio 2008 program group and executing the following command from the command prompt:

aspnet_regsql

Figure 8 – The ASP.NET SQL Server Setup Wizard



The ASP.NET SQL Server Setup Wizard enables you to select a SQL Server database on your network and install all of the database objects required by the ASP.NET application services. The database server is not required to be located on your local machine.

> **NOTE**
>
> If you don't want to use the ASP.NET SQL Server Setup Wizard, then you can find SQL scripts for adding the application services database objects in the following folder:
>
> C:\Windows\Microsoft.NET\Framework\v2.0.50727

After you create the necessary database objects, you need to modify the database connection used by your MVC application. Modify the ApplicationServices connection string in your web configuration (web.config) file so that it points to the production database. For example, the modified connection in Listing 3 points to a database named MyProductionDB (the original ApplicationServices connection string has been commented out).

Listing 3 – Web.config

```
<connectionStrings>
    <!--<add name="ApplicationServices" connectionString="data source=.\SQLEXPRESS;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User Instance=true"
providerName="System.Data.SqlClient"/>-->
  <add name="ApplicationServices" connectionString="data source=localhost;Integrated Security=SSPI;Initial
Catalog=MyProductionDB" />
</connectionStrings>
```

Configuring Database Permissions

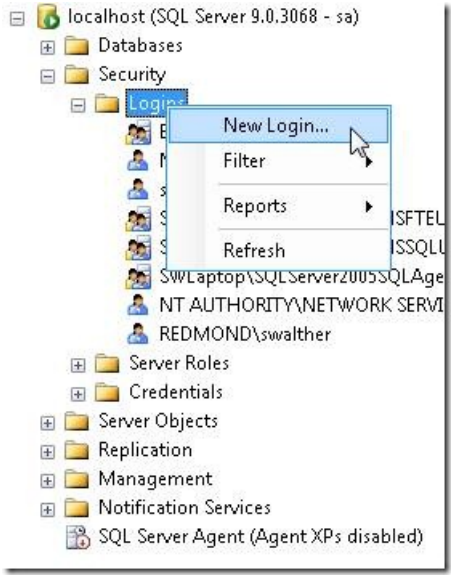
If you use Integrated Security to connect to your database then you will need to add the correct Windows user account as a login to your database. The correct account depends on whether you are using the ASP.NET Development Server or Internet Information Services as your web server. The correct user account also depends on your operating system.

If you are using the ASP.NET Development Server (the default web server used by Visual Studio) then your application executes within the context of your Windows user account. In that case, you need to add your Windows user account as a database server login.

Alternatively, if you are using Internet Information Services then you need to add either the ASPNET account or the NT AUTHORITY/NETWORK SERVICE account as a database server login. If you are using Windows XP then add the ASPNET account as a login to your database. If you are using a more recent operating system, such as Windows Vista or Windows Server 2008, then add the NT AUTHORITY/NETWORK SERVICE account as the database login.

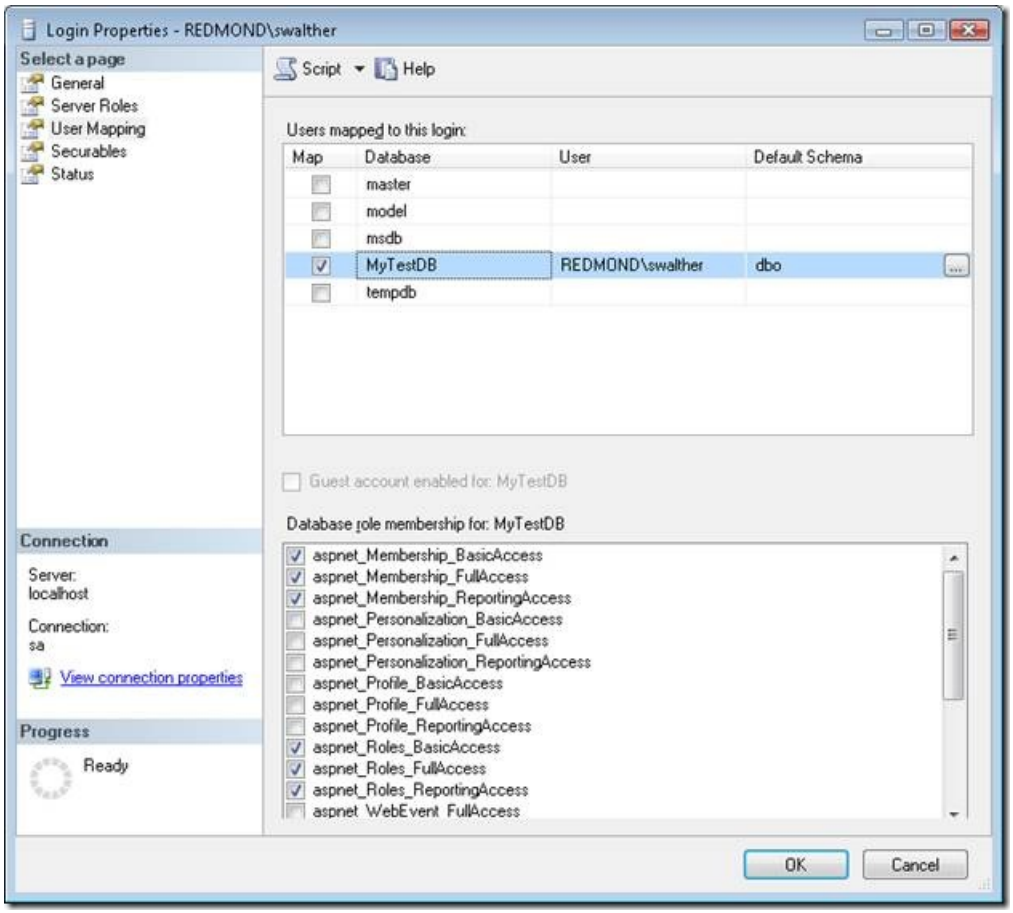
You can add a new user account to your database by using Microsoft SQL Server Management Studio (see Figure 9).

Figure 9 – Creating a new Microsoft SQL Server login



After you create the required login, you need to map the login to a database user with the right database roles. Double-click the login and select the User Mapping tab. Select one or more application services database roles. For example, in order to authenticate users, you need to enable the aspnet_Membership_BasicAccess database role. In order to create new users, you need to enable the aspnet_Membership_FullAccess database role (see Figure 10).

Figure 10 – Adding Application Services database roles



Summary

In this tutorial, you learned how to use Forms authentication when building an ASP.NET MVC application. First, you learned how to create new users and roles by taking advantage of the Web Site Administration Tool. Next, you learned how to use the [Authorize] attribute to prevent unauthorized users from invoking controller actions. Finally, you learned how to configure your MVC application to store user and role information in a production database.

NEXT

Authenticating Users with Windows Authentication (C#)

4/10/2020 • 4 minutes to read • Edit Online

by Microsoft

Learn how to use Windows authentication in the context of an MVC application. You learn how to enable Windows authentication within your application's web configuration file and how to configure authentication with IIS. Finally, you learn how to use the [Authorize] attribute to restrict access to controller actions to particular Windows users or groups.

The goal of this tutorial is to explain how you can take advantage of the security features built into Internet Information Services to password protect the views in your MVC applications. You learn how to allow controller actions to be invoked only by particular Windows users or users who are members of particular Windows groups.

Using Windows authentication makes sense when you are building an internal company website (an intranet site) and you want your users to be able to use their standard Windows user names and passwords when accessing the website. If you are building an outwards facing website (an Internet website) consider using Forms authentication instead.

Enabling Windows Authentication

When you create a new ASP.NET MVC application, Windows authentication is not enabled by default. Forms authentication is the default authentication type enabled for MVC applications. You must enable Windows authentication by modifying your MVC application's web configuration (web.config) file. Find the <authentication> section and modify it to use Windows instead of Forms authentication like this:

```
<authentication mode="Windows">

</authentication>
```

When you enable Windows authentication, your web server becomes responsible for authenticating users. Typically, there are two different types of web servers that you use when creating and deploying an ASP.NET MVC application.

First, while developing an MVC application, you use the ASP.NET Development Web Server included with Visual Studio. By default, the ASP.NET Development Web Server executes all pages in the context of the current Windows account (whatever account you used to log into Windows).

The ASP.NET Development Web Server also supports NTLM authentication. You can enable NTLM authentication by right-clicking the name of your project in the Solution Explorer window and selecting Properties. Next, select the Web tab and check the NTLM checkbox (see Figure 1).

Figure 1 – Enabling NTLM authentication for the ASP.NET Development Web Server

For a production web application, on the hand, you use IIS as your web server. IIS supports several types of authentication including:

- Basic Authentication – Defined as part of the HTTP 1.0 protocol. Sends user names and passwords in clear text (Base64 encoded) across the Internet. - Digest Authentication – Sends a hash of a password, instead of the password itself, across the internet. - Integrated Windows (NTLM) Authentication – The best type of authentication to use in intranet environments using windows. - Certificate Authentication – Enables authentication using a client-side certificate. The certificate maps to a Windows user account.

> **NOTE**
>
> For a more detailed overview of these different types of authentication, see https://msdn.microsoft.com/library/aa292114(VS.71).aspx.

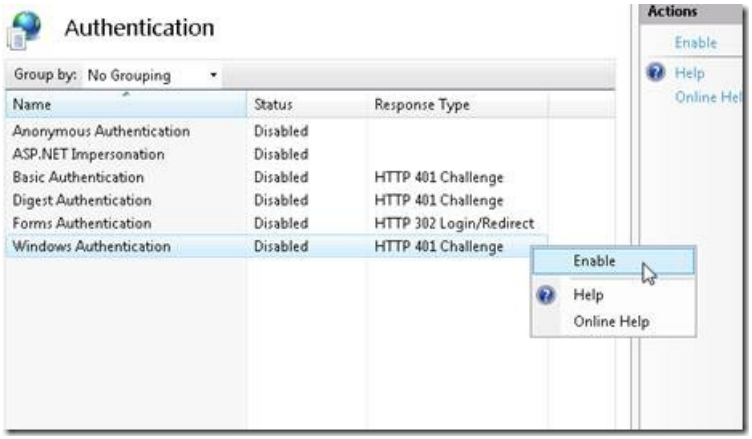
You can use Internet Information Services Manager to enable a particular type of authentication. Be aware that all types of authentication are not available in the case of every operating system. Furthermore, if you are using IIS 7.0 with Windows Vista, you will need to enable the different types of Windows authentication before they appear in the Internet Information Services Manager. Open **Control Panel, Programs, Programs and Features, Turn Windows features on or off**, and expand the Internet Information Services node (see Figure 2).

Figure 2 – Enabling Windows IIS features

Using Internet Information Services, you can enable or disable different types of authentication. For example,

Figure 3 illustrates disabling anonymous authentication and enabling Integrated Windows (NTLM) authentication when using IIS 7.0.

Figure 3 – Enabling Integrated Windows Authentication



Authorizing Windows Users and Groups

After you enable Windows authentication, you can use the [Authorize] attribute to control access to controllers or controller actions. This attribute can be applied to an entire MVC controller or a particular controller action.

For example, the Home controller in Listing 1 exposes three actions named Index(), CompanySecrets(), and StephenSecrets(). Anyone can invoke the Index() action. However, only members of the Windows local Managers group can invoke the CompanySecrets() action. Finally, only the Windows domain user named Stephen (in the Redmond domain) can invoke the StephenSecrets() action.

Listing 1 – Controllers\HomeController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Mvc.Ajax;

namespace MvcApplication1.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        [Authorize(Roles = "Managers")]
        public ActionResult CompanySecrets()
        {
            return View();
        }

        [Authorize(Users="redmond\\swalther")]
        public ActionResult StephenSecrets()
        {
            return View();
        }



    }
}
```

NOTE

Because of Windows User Account Control (UAC), when working with Windows Vista or Windows Server 2008, the local Administrators group will behave differently than other groups. The [Authorize] attribute won't correctly recognize a member of the local Administrators group unless you modify your computer's UAC settings.

Exactly what happens when you attempt to invoke a controller action without being the right permissions depends on the type of authentication enabled. By default, when using the ASP.NET Development Server, you simply get a blank page. The page is served with a **401 Not Authorized** HTTP Response Status.

If, on the other hand, you are using IIS with Anonymous authentication disabled and Basic authentication enabled, then you keep getting a login dialog prompt each time you request the protected page (see Figure 4).

Figure 4 – Basic authentication login dialog

Summary

This tutorial explained how you can use Windows authentication in the context of an ASP.NET MVC application. You learned how to enable Windows authentication within your application's web configuration file and how to configure authentication with IIS. Finally, you learned how to use the [Authorize] attribute to restrict access to controller actions to particular Windows users or groups.

PREVIOUS NEXT

Preventing JavaScript Injection Attacks (C#)

2/20/2020 • 6 minutes to read • Edit Online

by Stephen Walther

Download PDF

Prevent JavaScript Injection Attacks and Cross-Site Scripting Attacks from happening to you. In this tutorial, Stephen Walther explains how you can easily defeat these types of attacks by HTML encoding your content.

The goal of this tutorial is to explain how you can prevent JavaScript injection attacks in your ASP.NET MVC applications. This tutorial discusses two approaches to defending your website against a JavaScript injection attack. You learn how to prevent JavaScript injection attacks by encoding the data that you display. You also learn how to prevent JavaScript injection attacks by encoding the data that you accept.

What is a JavaScript Injection Attack?

Whenever you accept user input and redisplay the user input, you open your website to JavaScript injection attacks. Let's examine a concrete application that is open to JavaScript injection attacks.

Imagine that you have created a customer feedback website (see Figure 1). Customers can visit the website and enter feedback on their experience using your products. When a customer submits their feedback, the feedback is redisplayed on the feedback page.

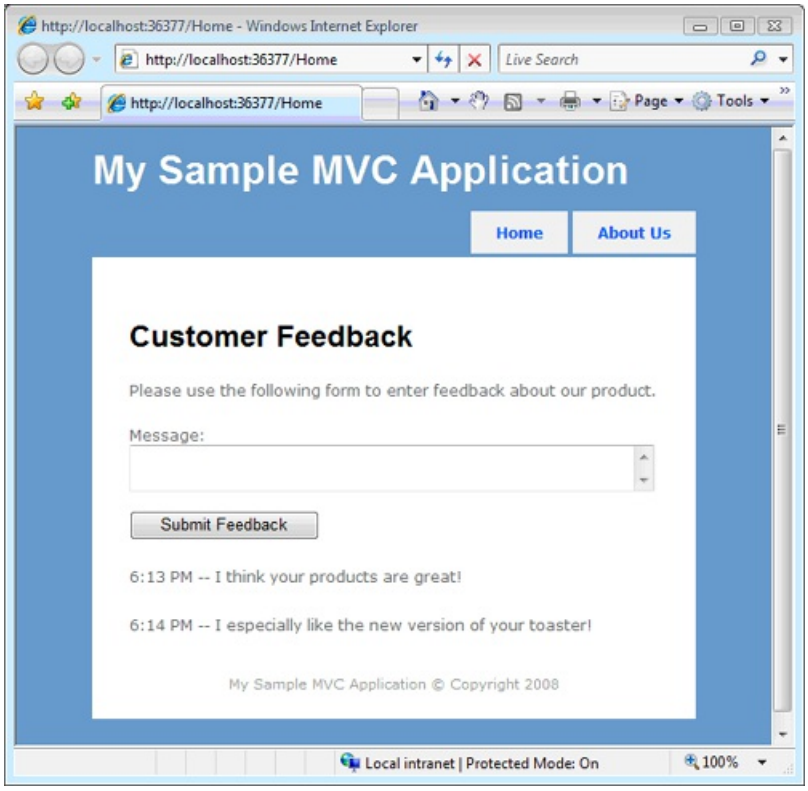


Figure 01: Customer Feedback Website (Click to view full-size image)

The customer feedback website uses the `controller` in Listing 1. This `controller` contains two actions named `Index()` and `Create()`.

Listing 1 – `HomeController.cs`

```
using System;
using System.Web.Mvc;
using CustomerFeedback.Models;

namespace CustomerFeedback.Controllers
{
     [HandleError]
     public class HomeController : Controller
     {
          private FeedbackDataContext db = new FeedbackDataContext();

          public ActionResult Index()
          {
               return View(db.Feedbacks);
          }

          public ActionResult Create(string message)

          {
               // Add feedback
               var newFeedback = new Feedback();
               newFeedback.Message = message;
               newFeedback.EntryDate = DateTime.Now;
               db.Feedbacks.InsertOnSubmit(newFeedback);
               db.SubmitChanges();

               // Redirect
               return RedirectToAction("Index");
          }
     }
}
```

The `Index()` method displays the `Index` view. This method passes all of the previous customer feedback to the `Index` view by retrieving the feedback from the database (using a LINQ to SQL query).

The `Create()` method creates a new Feedback item and adds it to the database. The message that the customer enters in the form is passed to the `Create()` method in the message parameter. A Feedback item is created and the message is assigned to the Feedback item's `Message` property. The Feedback item is submitted to the database with the `DataContext.SubmitChanges()` method call. Finally, the visitor is redirected back to the `Index` view where all of the feedback is displayed.

The `Index` view is contained in Listing 2.

Listing 2 – `Index.aspx`

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="true"
CodeBehind="Index.aspx.cs" Inherits="CustomerFeedback.Views.Home.Index"%>

<%@ Import Namespace="CustomerFeedback.Models" %>

<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent" runat="server">
     <h1>Customer Feedback</h1>
     <p>
          Please use the following form to enter feedback about our product.
     </p>

     <form method="post" action="/Home/Create">

          <label for="message">Message:</label>
          <br />
          <textarea name="message" cols="50" rows="2"></textarea>
          <br /><br />
          <input type="submit" value="Submit Feedback" />
     </form>

     <% foreach (Feedback feedback in ViewData.Model)
     {%>
          <p>
          <%=feedback.EntryDate.ToShortTimeString()%>
          --
          <%=feedback.Message%>
          </p>
     <% }%>

</asp:Content>
```

The `Index` view has two sections. The top section contains the actual customer feedback form. The bottom section contains a For..Each loop that loops through all of the previous customer feedback items and displays the EntryDate and Message properties for each feedback item.

The customer feedback website is a simple website. Unfortunately, the website is open to JavaScript injection attacks.

Imagine that you enter the following text into the customer feedback form:

```
<script>alert("Boo!")</script>
```

This text represents a JavaScript script that displays an alert message box. After someone submits this script into the feedback form, the message *Boo!* will appear whenever anyone visits the customer feedback website in the future (see Figure 2).

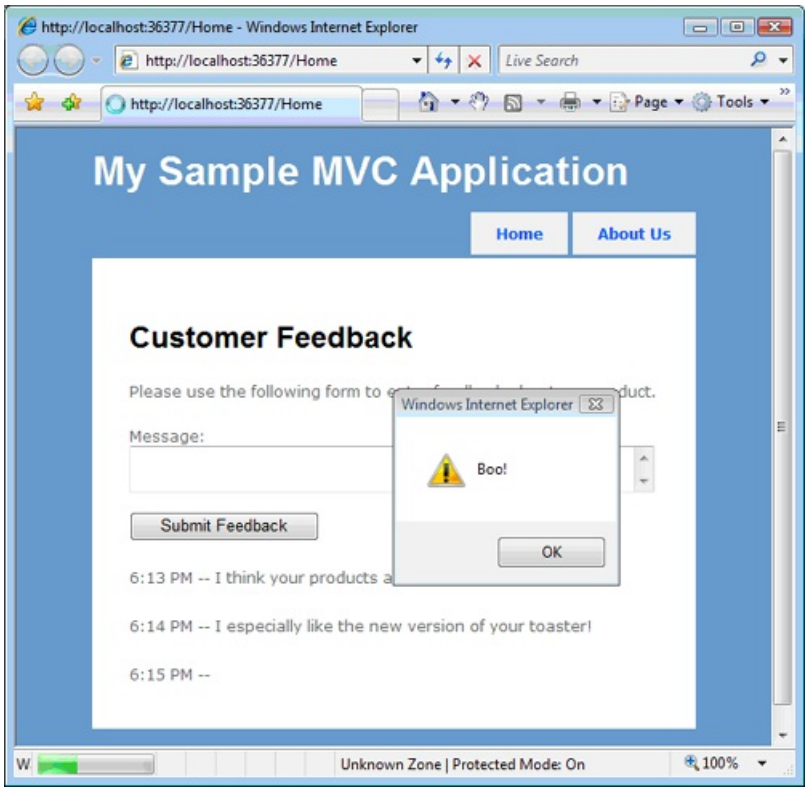


Figure 02: JavaScript Injection (Click to view full-size image)

Now, your initial response to JavaScript injection attacks might be apathy. You might think that JavaScript injection attacks are simply a type of *defacement* attack. You might believe that no one can do anything truly evil by committing a JavaScript injection attack.

Unfortunately, a hacker can do some really, really evil things by injecting JavaScript into a website. You can use a JavaScript injection attack to perform a Cross-Site Scripting (XSS) attack. In a Cross-Site Scripting attack, you steal confidential user information and send the information to another website.

For example, a hacker can use a JavaScript injection attack to steal the values of browser cookies from other users. If sensitive information -- such as passwords, credit card numbers, or social security numbers – is stored in the browser cookies, then a hacker can use a JavaScript injection attack to steal this information. Or, if a user enters sensitive information in a form field contained in a page that has been compromised with a JavaScript attack, then the hacker can use the injected JavaScript to grab the form data and send it to another website.

Please be scared. Take JavaScript injection attacks seriously and protect your user's confidential information. In the next two sections, we discuss two techniques that you can use to defend your ASP.NET MVC applications from JavaScript injection attacks.

Approach #1: HTML Encode in the View

One easy method of preventing JavaScript injection attacks is to HTML encode any data entered by website users when you redisplay the data in a view. The updated `Index` view in Listing 3 follows this approach.

Listing 3 – `Index.aspx` (HTML Encoded)

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="true"
CodeBehind="Index.aspx.cs" Inherits="CustomerFeedback.Views.Home.Index"%>

<%@ Import Namespace="CustomerFeedback.Models" %>

<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent" runat="server">
     <h1>Customer Feedback</h1>
     <p>
          Please use the following form to enter feedback about our product.
     </p>

     <form method="post" action="/Home/Create">
          <label for="message">Message:</label>
          <br />
          <textarea name="message" cols="50" rows="2"></textarea>
          <br /><br />
          <input type="submit" value="Submit Feedback" />

     </form>

     <% foreach (Feedback feedback in ViewData.Model)
     {%>
          <p>
          <%=feedback.EntryDate.ToShortTimeString()%>
          --
          <%=Html.Encode(feedback.Message)%>
          </p>
     <% }%>

</asp:Content>
```

Notice that the value of `feedback.Message` is HTML encoded before the value is displayed with the following code:

```
<%=Html.Encode(feedback.Message)%>
```

What does it mean to HTML encode a string? When you HTML encode a string, dangerous characters such as `<` and `>` are replaced by HTML entity references such as `<` and `>`. So when the string `<script>alert("Boo!")</script>` is HTML encoded, it gets converted to `<script>alert("Boo!")</script>`. The encoded string no longer executes as a JavaScript script when interpreted by a browser. Instead, you get the harmless page in Figure 3.

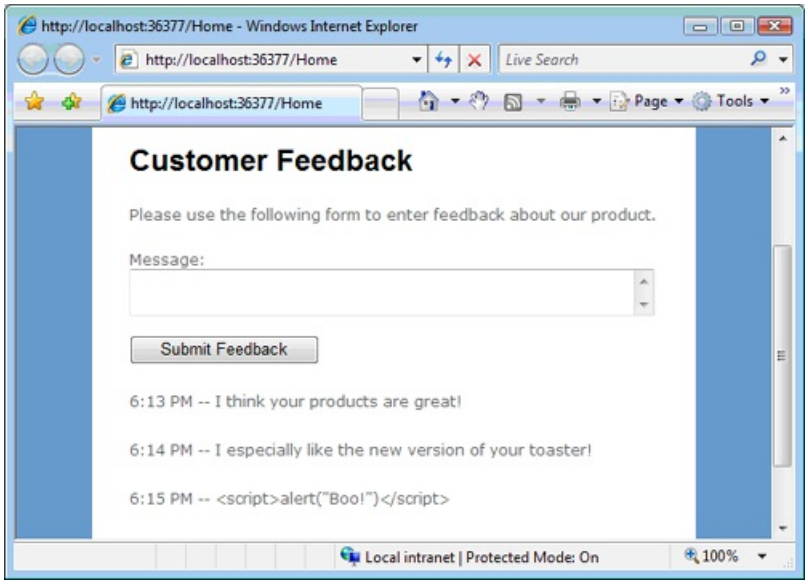


Figure 03: Defeated JavaScript Attack (Click to view full-size image)

Notice that in the `Index` view in Listing 3 only the value of `feedback.Message` is encoded. The value of `feedback.EntryDate` is not encoded. You only need to encode data entered by a user. Because the value of EntryDate was generated in the controller, you don't need to HTML encode this value.

Approach #2: HTML Encode in the Controller

Instead of HTML encoding data when you display the data in a view, you can HTML encode the data just before you submit the data to the database. This second approach is taken in the case of the `controller` in Listing 4.

Listing 4 – `HomeController.cs` (HTML Encoded)

```
using System;
using System.Web.Mvc;
using CustomerFeedback.Models;
namespace CustomerFeedback.Controllers
{
     [HandleError]
     public class HomeController : Controller
     {
          private FeedbackDataContext db = new FeedbackDataContext();

          public ActionResult Index()
          {
               return View(db.Feedbacks);
          }

          public ActionResult Create(string message)
          {
               // Add feedback
               var newFeedback = new Feedback();
               newFeedback.Message = Server.HtmlEncode(message);
               newFeedback.EntryDate = DateTime.Now;
               db.Feedbacks.InsertOnSubmit(newFeedback);

               db.SubmitChanges();

               // Redirect
               return RedirectToAction("Index");
          }
     }
}
```

Notice that the value of Message is HTML encoded before the value is submitted to the database within the `Create()` action. When the Message is redisplayed in the view, the Message is HTML encoded and any JavaScript injected in the Message is not executed.

Typically, you should favor the first approach discussed in this tutorial over this second approach. The problem with this second approach is that you end up with HTML encoded data in your database. In other words, your database data is dirtied with funny looking characters.

Why is this bad? If you ever need to display the database data in something other than a web page, then you will have problems. For example, you can no longer easily display the data in a Windows Forms application.

Summary

The purpose of this tutorial was to scare you about the prospect of a JavaScript injection attack. This tutorial discussed two approaches for defending your ASP.NET MVC applications against JavaScript injection attacks: you can either HTML encode user submitted data in the view or you can HTML encode user submitted data in the controller.

Authenticating Users with Forms Authentication (VB)

4/10/2020 • 7 minutes to read • Edit Online

by Microsoft

> Learn how to use the [Authorize] attribute to password protect particular pages in your MVC application. You learn how to use the Web Site Administration Tool to create and manage users and roles. You also learn how to configure where user account and role information is stored.

The goal of this tutorial is to explain how you can use Forms authentication to password protect the views in your ASP.NET MVC applications. You learn how to use the Web Site Administration Tool to create users and roles. You also learn how to prevent unauthorized users from invoking controller actions. Finally, you learn how to configure where user names and passwords are stored.

Using the Web Site Administration Tool

Before we do anything else, we should start by creating some users and roles. The easiest way to create new users and roles is to take advantage of the Visual Studio 2008 Web Site Administration Tool. You can launch this tool by selecting the menu option **Project, ASP.NET Configuration**. Alternatively, you can launch the Web Site Administration Tool by clicking the (somewhat scary) icon of the hammer hitting the world that appears at the top of the Solution Explorer window (see Figure 1).

Figure 1 – Launching the Web Site Administration Tool

Within the Web Site Administration Tool, you create new users and roles by selecting the Security tab. Click the **Create user** link to create a new user named Stephen (see Figure 2). Provide the Stephen user with any password that you want (for example, *secret*).

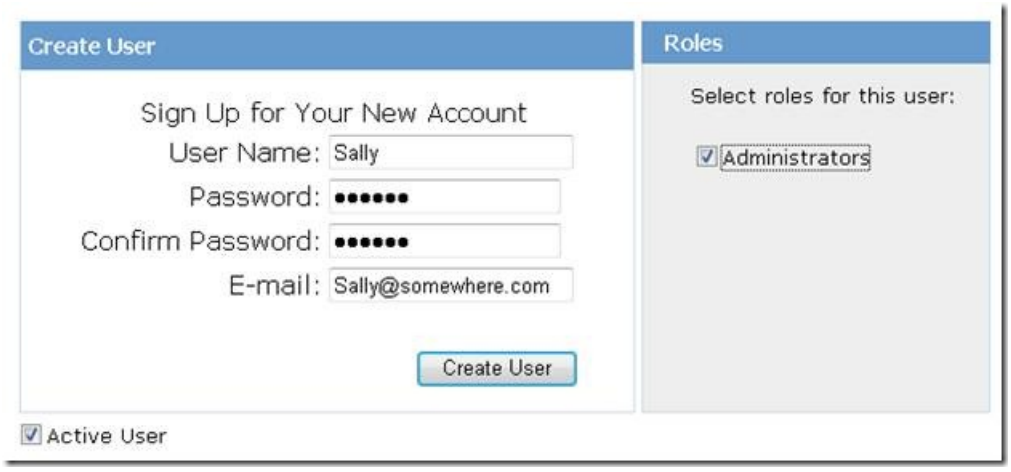
Figure 2 – Creating a new user

You create new roles by first enabling roles and defining one or more roles. Enable roles by clicking the **Enable roles** link. Next, create a role named *Administrators* by clicking the **Create or Manage roles** link (see Figure 3).

Figure 3 – Creating a new role

Finally, create a new user named Sally and associate Sally with the Administrators role by clicking the Create User link and selecting Administrators when creating Sally (see Figure 4).

Figure 4 – Adding a user to a role



When all is said and done, you should have two new users named Stephen and Sally. You should also have a new

role named Administrators. Sally is a member of the Administrators role and Stephen is not.

Requiring Authorization

You can require a user to be authenticated before the user invokes a controller action by adding the [Authorize] attribute to the action. You can apply the [Authorize] attribute to an individual controller action or you can apply this attribute to an entire controller class.

For example, the controller in Listing 1 exposes an action named CompanySecrets(). Because this action is decorated with the [Authorize] attribute, this action cannot be invoked unless a user is authenticated.

Listing 1 – Controllers\HomeController.vb

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

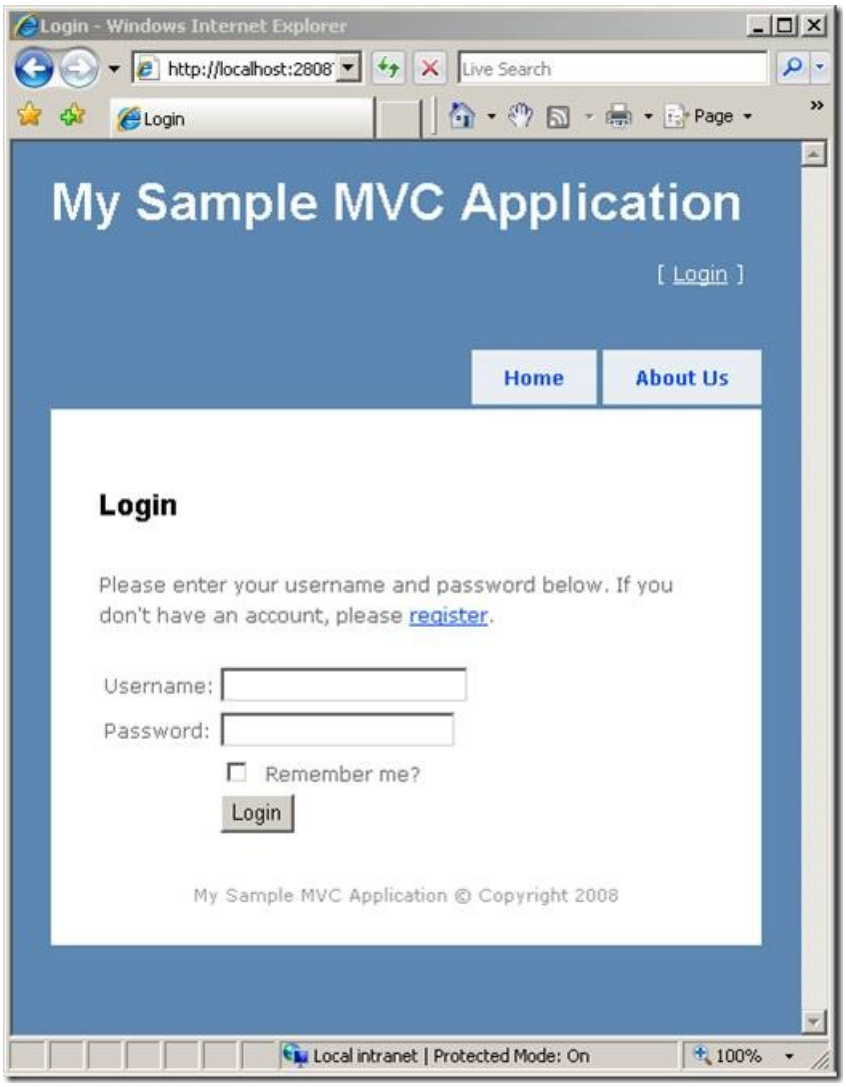
    Function Index()
        Return View()
    End Function

    <Authorize()> _
    Function CompanySecrets()
        Return View()
    End Function

End Class
```

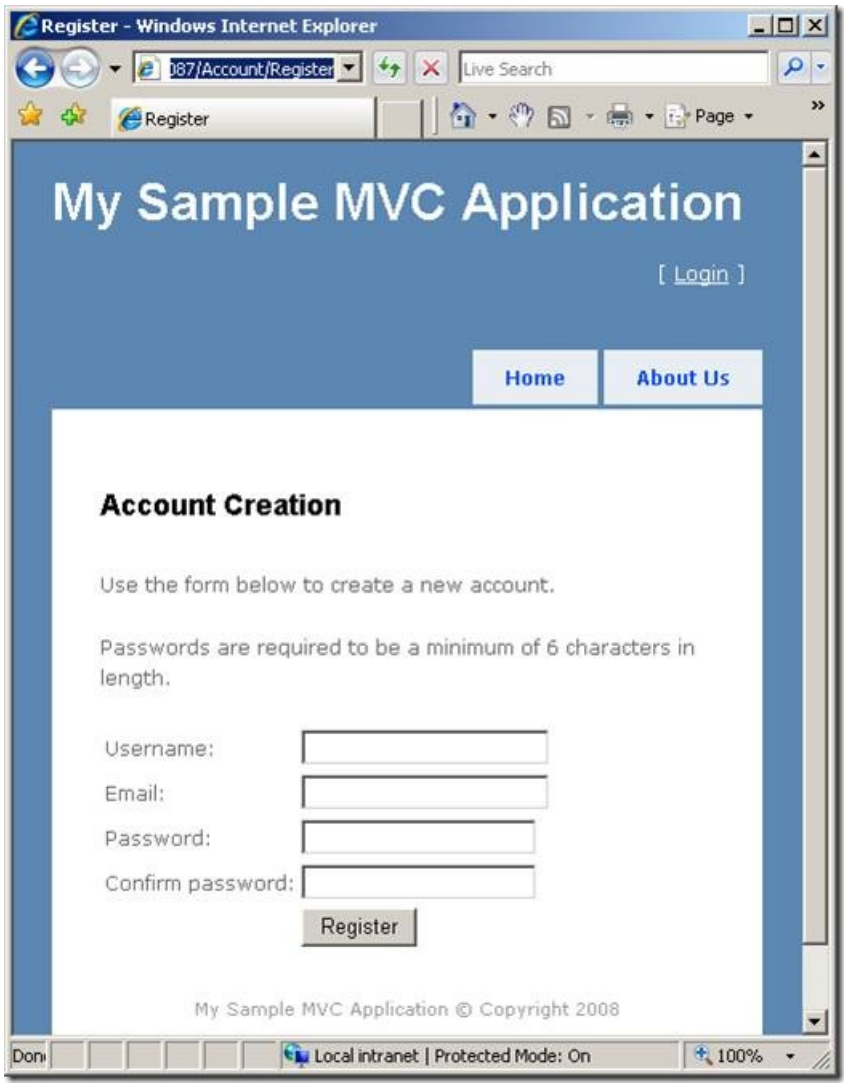
If you invoke the CompanySecrets() action by entering the URL /Home/CompanySecrets in the address bar of your browser, and you are not an authenticated user, then you will be redirected to the Login view automatically (see Figure 5).

Figure 5 – The Login view



You can use the Login view to enter your user name and password. If you are not a registered user then you can click the **register** link to navigate to the Register view (see Figure 6). You can use the Register view to create a new user account.

Figure 6 – The Register view



After you successfully log in, you can see the CompanySecrets view (see Figure 7). By default, you will continue to be logged in until you close your browser window.

Figure 7 – The CompanySecrets view

Authorizing by User Name or User Role

You can use the [Authorize] attribute to restrict access to a controller action to a particular set of users or a particular set of user roles. For example, the modified Home controller in Listing 2 contains two new actions named StephenSecrets() and AdministratorSecrets().

Listing 2 – Controllers\HomeController.vb

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Function Index()
        Return View()
    End Function

    <Authorize()> _
    Function CompanySecrets()
        Return View()
    End Function


    <Authorize(Users:="Stephen")> _
    Function StephenSecrets()
        Return View()
    End Function

    <Authorize(Roles:="Administrators")> _
    Function AdministratorSecrets()
        Return View()
    End Function

End Class
```

Only a user with the user name Stephen can invoke the StephenSecrets() action. All other users get redirected to the Login view. The Users property accepts a comma separated list of user account names.

Only users in the Administrators role can invoke the AdministratorSecrets() action. For example, because Sally is a member of the Administrators group, she can invoke the AdministratorSecrets() action. All other users get redirected to the Login view. The Roles property accepts a comma separated list of role names.

Configuring Authentication

At this point, you might be wondering where the user account and role information is being stored. By default, the information is stored in a (RANU) SQL Express database named ASPNETDB.mdf located in your MVC application's App_Data folder. This database is generated by the ASP.NET framework automatically when you start using membership.

In order to see the ASPNETDB.mdf database in the Solution Explorer window, you first need to select the menu option Project, Show All Files.

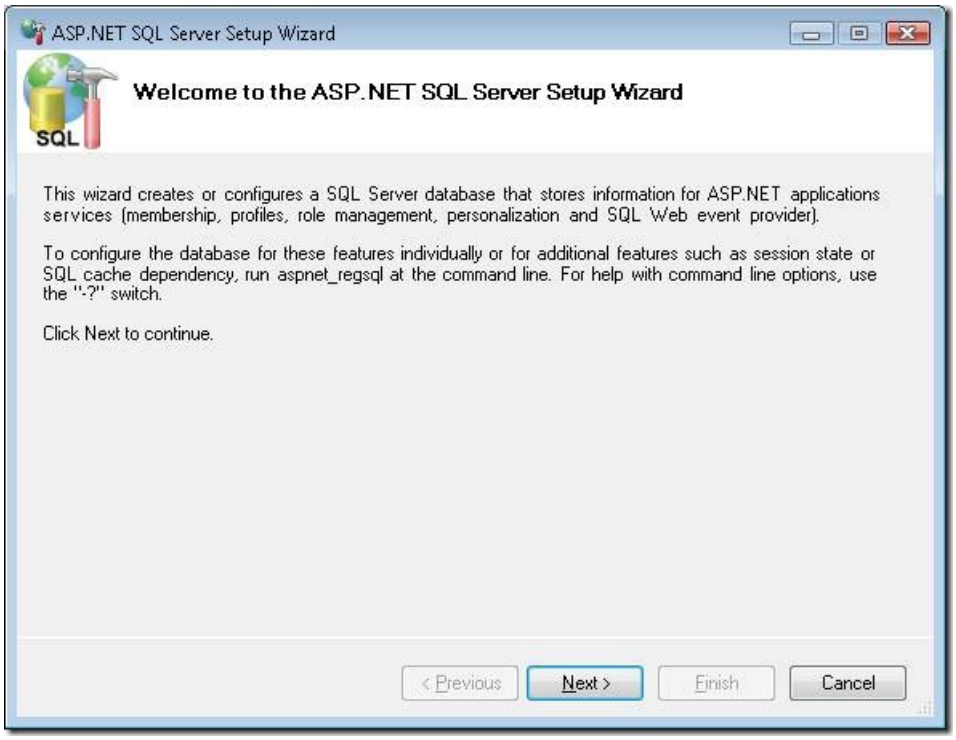
Using the default SQL Express database is fine when developing an application. Most likely, however, you won't want to use the default ASPNETDB.mdf database for a production application. In that case, you can change where user account information is stored by completing the following two steps:

1. Add the Application Services database objects to your production database - Change your application connection string to point to your production database

The first step is to add all of the necessary database objects (tables and stored procedures) to your production database. The easiest way to add these objects to a new database is to take advantage of the ASP.NET SQL Server Setup Wizard (see Figure 8). You can launch this tool by opening the Visual Studio 2008 Command Prompt from the Microsoft Visual Studio 2008 program group and executing the following command from the command prompt:

aspnet_regsql

Figure 8 – The ASP.NET SQL Server Setup Wizard



The ASP.NET SQL Server Setup Wizard enables you to select a SQL Server database on your network and install all of the database objects required by the ASP.NET application services. The database server is not required to be located on your local machine.

> **NOTE**
>
> If you don't want to use the ASP.NET SQL Server Setup Wizard, then you can find SQL scripts for adding the application services database objects in the following folder:
>
> C:\Windows\Microsoft.NET\Framework\v2.0.50727

After you create the necessary database objects, you need to modify the database connection used by your MVC application. Modify the ApplicationServices connection string in your web configuration (web.config) file so that it points to the production database. For example, the modified connection in Listing 3 points to a database named MyProductionDB (the original ApplicationServices connection string has been commented out).

Listing 3 – Web.config

```
<connectionStrings>

<!--<add name="ApplicationServices" connectionString="data source=.\SQLEXPRESS;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User Instance=true"
providerName="System.Data.SqlClient"/>-->

<add name="ApplicationServices" connectionString="data source=localhost;Integrated Security=SSPI;Initial
Catalog=MyProductionDB" />

</connectionStrings>
```

Configuring Database Permissions

If you use Integrated Security to connect to your database then you will need to add the correct Windows user account as a login to your database. The correct account depends on whether you are using the ASP.NET Development Server or Internet Information Services as your web server. The correct user account also depends on your operating system.

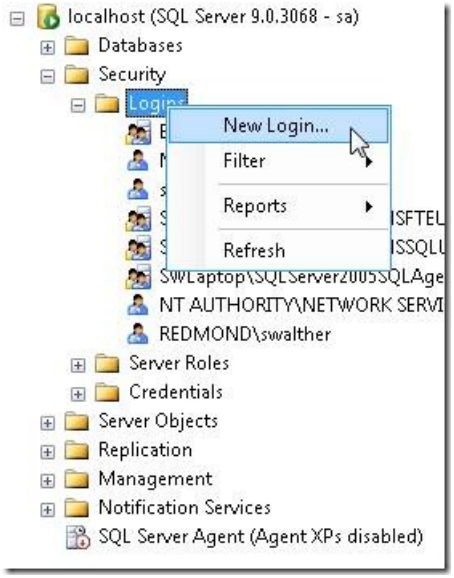
If you are using the ASP.NET Development Server (the default web server used by Visual Studio) then your

application executes within the context of your Windows user account. In that case, you need to add your Windows user account as a database server login.

Alternatively, if you are using Internet Information Services then you need to add either the ASPNET account or the NT AUTHORITY/NETWORK SERVICE account as a database server login. If you are using Windows XP then add the ASPNET account as a login to your database. If you are using a more recent operating system, such as Windows Vista or Windows Server 2008, then add the NT AUTHORITY/NETWORK SERVICE account as the database login.

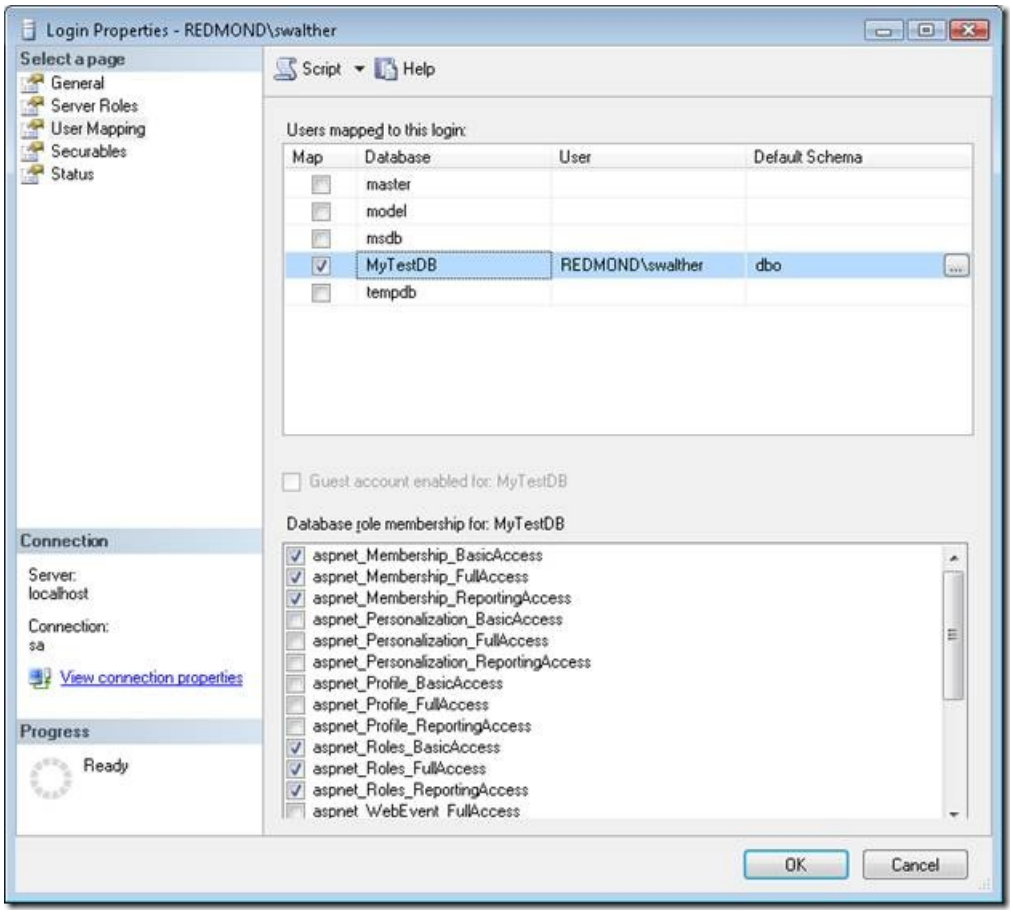
You can add a new user account to your database by using Microsoft SQL Server Management Studio (see Figure 9).

Figure 9 – Creating a new Microsoft SQL Server login



After you create the required login, you need to map the login to a database user with the right database roles. Double-click the login and select the User Mapping tab. Select one or more application services database roles. For example, in order to authenticate users, you need to enable the aspnet_Membership_BasicAccess database role. In order to create new users, you need to enable the aspnet_Membership_FullAccess database role (see Figure 10).

Figure 10 – Adding Application Services database roles



Summary

In this tutorial, you learned how to use Forms authentication when building an ASP.NET MVC application. First, you learned how to create new users and roles by taking advantage of the Web Site Administration Tool. Next, you learned how to use the [Authorize] attribute to prevent unauthorized users from invoking controller actions. Finally, you learned how to configure your MVC application to store user and role information in a production database.

PREVIOUS NEXT

Authenticating Users with Windows Authentication (VB)

4/10/2020 • 4 minutes to read • Edit Online

by Microsoft

Learn how to use Windows authentication in the context of an MVC application. You learn how to enable Windows authentication within your application's web configuration file and how to configure authentication with IIS. Finally, you learn how to use the [Authorize] attribute to restrict access to controller actions to particular Windows users or groups.

The goal of this tutorial is to explain how you can take advantage of the security features built into Internet Information Services to password protect the views in your MVC applications. You learn how to allow controller actions to be invoked only by particular Windows users or users who are members of particular Windows groups.

Using Windows authentication makes sense when you are building an internal company website (an intranet site) and you want your users to be able to use their standard Windows user names and passwords when accessing the website. If you are building an outwards facing website (an Internet website) consider using Forms authentication instead.

Enabling Windows Authentication

When you create a new ASP.NET MVC application, Windows authentication is not enabled by default. Forms authentication is the default authentication type enabled for MVC applications. You must enable Windows authentication by modifying your MVC application's web configuration (web.config) file. Find the <authentication> section and modify it to use Windows instead of Forms authentication like this:

```
<authentication mode="Windows">

</authentication>
```

When you enable Windows authentication, your web server becomes responsible for authenticating users. Typically, there are two different types of web servers that you use when creating and deploying an ASP.NET MVC application.

First, while developing an MVC application, you use the ASP.NET Development Web Server included with Visual Studio. By default, the ASP.NET Development Web Server executes all pages in the context of the current Windows account (whatever account you used to log into Windows).

The ASP.NET Development Web Server also supports NTLM authentication. You can enable NTLM authentication by right-clicking the name of your project in the Solution Explorer window and selecting Properties. Next, select the Web tab and check the NTLM checkbox (see Figure 1).

Figure 1 – Enabling NTLM authentication for the ASP.NET Development Web Server

For a production web application, on the hand, you use IIS as your web server. IIS supports several types of authentication including:

- Basic Authentication – Defined as part of the HTTP 1.0 protocol. Sends user names and passwords in clear text (Base64 encoded) across the Internet. - Digest Authentication – Sends a hash of a password, instead of the password itself, across the internet. - Integrated Windows (NTLM) Authentication – The best type of authentication to use in intranet environments using windows. - Certificate Authentication – Enables authentication using a client-side certificate. The certificate maps to a Windows user account.

> **NOTE**
>
> For a more detailed overview of these different types of authentication, see https://msdn.microsoft.com/library/aa292114(VS.71).aspx.

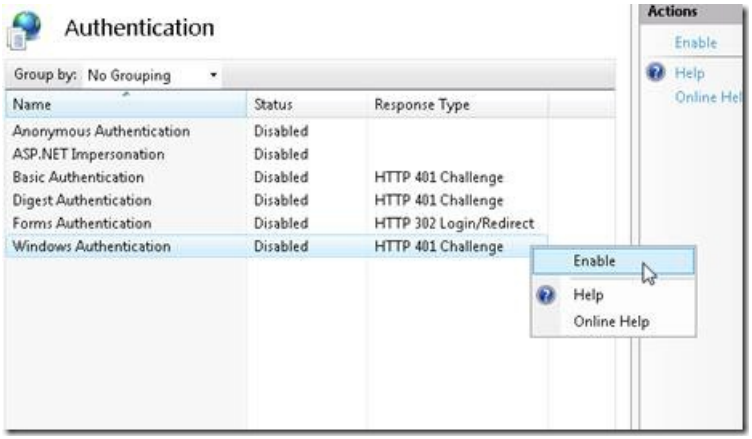
You can use Internet Information Services Manager to enable a particular type of authentication. Be aware that all types of authentication are not available in the case of every operating system. Furthermore, if you are using IIS 7.0 with Windows Vista, you will need to enable the different types of Windows authentication before they appear in the Internet Information Services Manager. Open **Control Panel, Programs, Programs and Features, Turn Windows features on or off**, and expand the Internet Information Services node (see Figure 2).

Figure 2 – Enabling Windows IIS features

Using Internet Information Services, you can enable or disable different types of authentication. For example,

Figure 3 illustrates disabling anonymous authentication and enabling Integrated Windows (NTLM) authentication when using IIS 7.0.

Figure 3 – Enabling Integrated Windows Authentication



Authorizing Windows Users and Groups

After you enable Windows authentication, you can use the <Authorize> attribute to control access to controllers or controller actions. This attribute can be applied to an entire MVC controller or a particular controller action.

For example, the Home controller in Listing 1 exposes three actions named Index(), CompanySecrets(), and StephenSecrets(). Anyone can invoke the Index() action. However, only members of the Windows local Managers group can invoke the CompanySecrets() action. Finally, only the Windows domain user named Stephen (in the Redmond domain) can invoke the StephenSecrets() action.

Listing 1 – Controllers\HomeController.vb

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Function Index()
        Return View()
    End Function

    <Authorize(Roles:="Managers")> _
    Function CompanySecrets()
        Return View()
    End Function


    <Authorize(Users:="redmond\swalther")> _
    Function StephenSecrets()
        Return View()
    End Function

End Class
```

> **NOTE**
>
> Because of Windows User Account Control (UAC), when working with Windows Vista or Windows Server 2008, the local Administrators group will behave differently than other groups. The <Authorize> attribute won't correctly recognize a member of the local Administrators group unless you modify your computer's UAC settings.

Exactly what happens when you attempt to invoke a controller action without being the right permissions depends on the type of authentication enabled. By default, when using the ASP.NET Development Server, you simply get a blank page. The page is served with a **401 Not Authorized** HTTP Response Status.

If, on the other hand, you are using IIS with Anonymous authentication disabled and Basic authentication enabled, then you keep getting a login dialog prompt each time you request the protected page (see Figure 4).

Figure 4 – Basic authentication login dialog

Summary

This tutorial explained how you can use Windows authentication in the context of an ASP.NET MVC application. You learned how to enable Windows authentication within your application's web configuration file and how to configure authentication with IIS. Finally, you learned how to use the <Authorize> attribute to restrict access to controller actions to particular Windows users or groups.

PREVIOUS NEXT

Preventing JavaScript Injection Attacks (VB)

2/20/2020 • 6 minutes to read • Edit Online

by Stephen Walther

Download PDF

Prevent JavaScript Injection Attacks and Cross-Site Scripting Attacks from happening to you. In this tutorial, Stephen Walther explains how you can easily defeat these types of attacks by HTML encoding your content.

The goal of this tutorial is to explain how you can prevent JavaScript injection attacks in your ASP.NET MVC applications. This tutorial discusses two approaches to defending your website against a JavaScript injection attack. You learn how to prevent JavaScript injection attacks by encoding the data that you display. You also learn how to prevent JavaScript injection attacks by encoding the data that you accept.

What is a JavaScript Injection Attack?

Whenever you accept user input and redisplay the user input, you open your website to JavaScript injection attacks. Let's examine a concrete application that is open to JavaScript injection attacks.

Imagine that you have created a customer feedback website (see Figure 1). Customers can visit the website and enter feedback on their experience using your products. When a customer submits their feedback, the feedback is redisplayed on the feedback page.

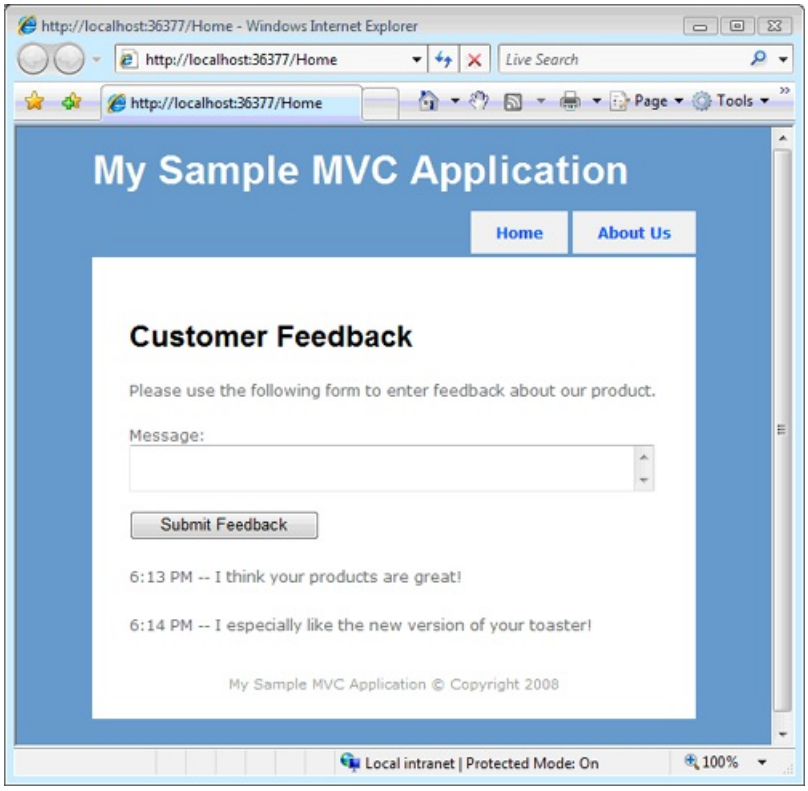


Figure 01: Customer Feedback Website (Click to view full-size image)

The customer feedback website uses the `controller` in Listing 1. This `controller` contains two actions named `Index()` and `Create()`.

Listing 1 – `HomeController.vb`

```
Public Class HomeController
     Inherits System.Web.Mvc.Controller

     Private db As New FeedbackDataContext()

     Function Index()
          Return View(db.Feedbacks)
     End Function

     Function Create(ByVal message As String)
          ' Add feedback
          Dim newFeedback As New Feedback()
          newFeedback.Message = Server.HtmlEncode(message)

          newFeedback.EntryDate = DateTime.Now
          db.Feedbacks.InsertOnSubmit(newFeedback)
          db.SubmitChanges()

          ' Redirect
          Return RedirectToAction("Index")
     End Function

End Class
```

The `Index()` method displays the `Index` view. This method passes all of the previous customer feedback to the `Index` view by retrieving the feedback from the database (using a LINQ to SQL query).

The `Create()` method creates a new Feedback item and adds it to the database. The message that the customer enters in the form is passed to the `Create()` method in the message parameter. A Feedback item is created and the message is assigned to the Feedback item's `Message` property. The Feedback item is submitted to the database with the `DataContext.SubmitChanges()` method call. Finally, the visitor is redirected back to the `Index` view where all of the feedback is displayed.

The `Index` view is contained in Listing 2.

Listing 2 – `Index.aspx`

```
<%@ Page Language="VB" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="false"
CodeBehind="Index.aspx.vb" Inherits="CustomerFeedback.Index"%>

<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent" runat="server">
     <h1>Customer Feedback</h1>
     <p>
          Please use the following form to enter feedback about our product.
     </p>
     <form method="post" action="/Home/Create">
          <label for="message">Message:</label>

          <br />
          <textarea name="message" cols="50" rows="2"></textarea>
          <br /><br />
          <input type="submit" value="Submit Feedback" />
     </form>

     <% For Each feedback As CustomerFeedback.Feedback In ViewData.Model%>
          <p>

          <%=feedback.EntryDate.ToShortTimeString()%>
          --
          <%=feedback.Message%>
          </p>
     <% Next %>

</asp:Content>
```

The `Index` view has two sections. The top section contains the actual customer feedback form. The bottom section contains a For..Each loop that loops through all of the previous customer feedback items and displays the EntryDate and Message properties for each feedback item.

The customer feedback website is a simple website. Unfortunately, the website is open to JavaScript injection attacks.

Imagine that you enter the following text into the customer feedback form:

```
<script>alert("Boo!")</script>
```

This text represents a JavaScript script that displays an alert message box. After someone submits this script into the feedback form, the message *Boo!* will appear whenever anyone visits the customer feedback website in the future (see Figure 2).

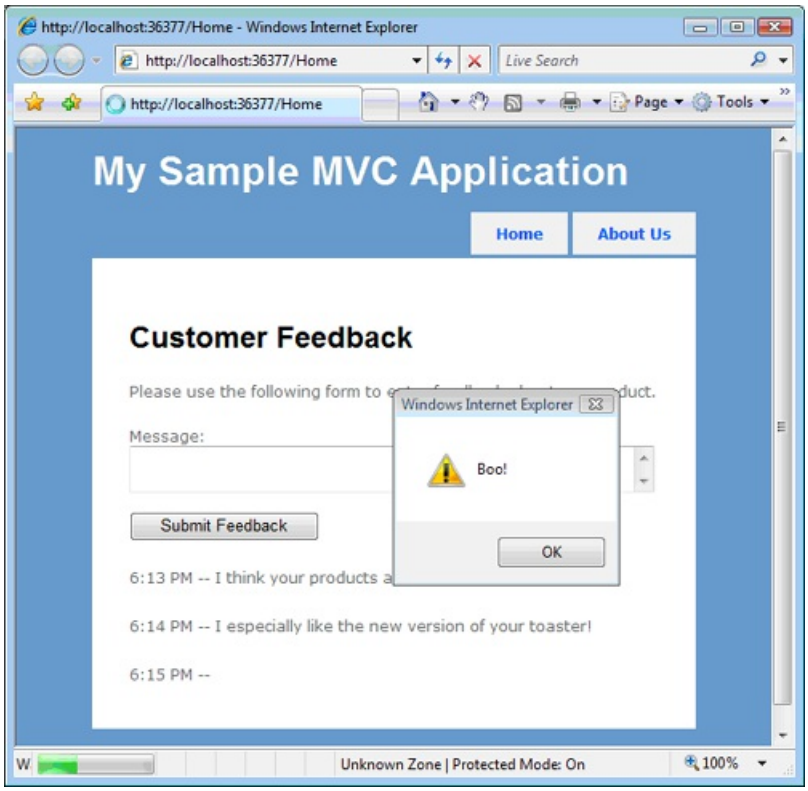


Figure 02: JavaScript Injection (Click to view full-size image)

Now, your initial response to JavaScript injection attacks might be apathy. You might think that JavaScript injection attacks are simply a type of *defacement* attack. You might believe that no one can do anything truly evil by committing a JavaScript injection attack.

Unfortunately, a hacker can do some really, really evil things by injecting JavaScript into a website. You can use a JavaScript injection attack to perform a Cross-Site Scripting (XSS) attack. In a Cross-Site Scripting attack, you steal confidential user information and send the information to another website.

For example, a hacker can use a JavaScript injection attack to steal the values of browser cookies from other users. If sensitive information -- such as passwords, credit card numbers, or social security numbers – is stored in the browser cookies, then a hacker can use a JavaScript injection attack to steal this information. Or, if a user enters sensitive information in a form field contained in a page that has been compromised with a JavaScript attack, then the hacker can use the injected JavaScript to grab the form data and send it to another website.

Please be scared. Take JavaScript injection attacks seriously and protect your user's confidential information. In the next two sections, we discuss two techniques that you can use to defend your ASP.NET MVC applications from JavaScript injection attacks.

Approach #1: HTML Encode in the View

One easy method of preventing JavaScript injection attacks is to HTML encode any data entered by website users when you redisplay the data in a view. The updated `Index` view in Listing 3 follows this approach.

Listing 3 – `Index.aspx` (HTML Encoded)

```
<%@ Page Language="VB" MasterPageFile="~/Views/Shared/Site.Master" AutoEventWireup="false"
CodeBehind="Index.aspx.vb" Inherits="CustomerFeedback.Index"%>

<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent" runat="server">
     <h1>Customer Feedback</h1>

     <p>
          Please use the following form to enter feedback about our product.
     </p>

     <form method="post" action="/Home/Create">
          <label for="message">Message:</label>
          <br />
          <textarea name="message" cols="50" rows="2"></textarea>

          <br /><br />
          <input type="submit" value="Submit Feedback" />
     </form>

     <% For Each feedback As CustomerFeedback.Feedback In ViewData.Model%>
          <p>
          <%=feedback.EntryDate.ToShortTimeString()%>
          --
          <%=Html.Encode(feedback.Message)%>

          </p>
     <% Next %>

</asp:Content>
```

Notice that the value of `feedback.Message` is HTML encoded before the value is displayed with the following code:

```
<%=Html.Encode(feedback.Message)%>
```

What does it mean to HTML encode a string? When you HTML encode a string, dangerous characters such as `<` and `>` are replaced by HTML entity references such as `<` and `>`. So when the string `<script>alert("Boo!")</script>` is HTML encoded, it gets converted to `<script>alert("Boo!")</script>`. The encoded string no longer executes as a JavaScript script when interpreted by a browser. Instead, you get the harmless page in Figure 3.

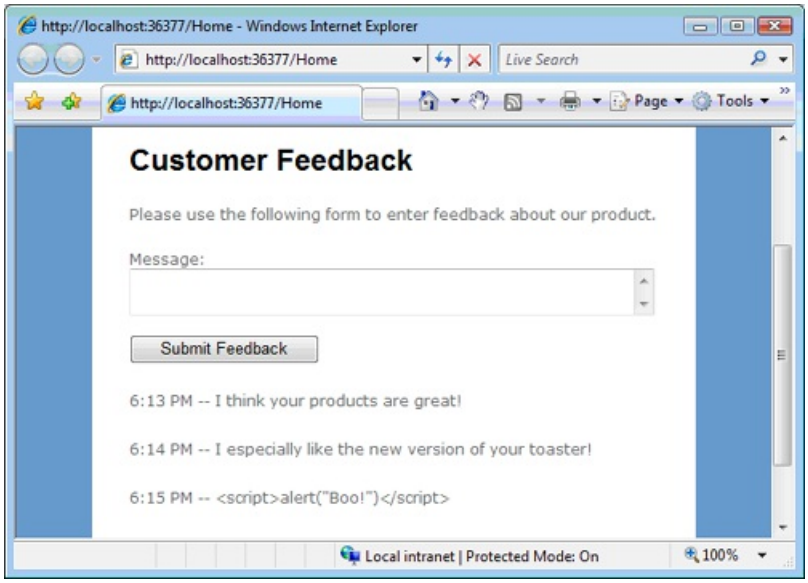


Figure 03: Defeated JavaScript Attack (Click to view full-size image)

Notice that in the `Index` view in Listing 3 only the value of `feedback.Message` is encoded. The value of

`feedback.EntryDate` is not encoded. You only need to encode data entered by a user. Because the value of EntryDate was generated in the controller, you don't need to HTML encode this value.

Approach #2: HTML Encode in the Controller

Instead of HTML encoding data when you display the data in a view, you can HTML encode the data just before you submit the data to the database. This second approach is taken in the case of the `controller` in Listing 4.

Listing 4 – `HomeController.cs` (HTML Encoded)

```
Public Class HomeController
     Inherits System.Web.Mvc.Controller

     Private db As New FeedbackDataContext()

     Function Index()
          Return View(db.Feedbacks)
     End Function

     Function Create(ByVal message As String)
          ' Add feedback
          Dim newFeedback As New Feedback()
          newFeedback.Message = Server.HtmlEncode(message)
          newFeedback.EntryDate = DateTime.Now
          db.Feedbacks.InsertOnSubmit(newFeedback)
          db.SubmitChanges()

          ' Redirect
          Return RedirectToAction("Index")
     End Function

End Class
```

Notice that the value of Message is HTML encoded before the value is submitted to the database within the `Create()` action. When the Message is redisplayed in the view, the Message is HTML encoded and any JavaScript injected in the Message is not executed.

Typically, you should favor the first approach discussed in this tutorial over this second approach. The problem with this second approach is that you end up with HTML encoded data in your database. In other words, your database data is dirtied with funny looking characters.

Why is this bad? If you ever need to display the database data in something other than a web page, then you will have problems. For example, you can no longer easily display the data in a Windows Forms application.

Summary

The purpose of this tutorial was to scare you about the prospect of a JavaScript injection attack. This tutorial discussed two approaches for defending your ASP.NET MVC applications against JavaScript injection attacks: you can either HTML encode user submitted data in the view or you can HTML encode user submitted data in the controller.

PREVIOUS

Unit Testing

2/20/2020 • 2 minutes to read • Edit Online

This tutorial set shows you how to create unit tests for ASP.NET MVC controller actions.

- Creating Unit Tests for ASP.NET MVC Applications (C#)
- Creating Unit Tests for ASP.NET MVC Applications (VB)

Creating Unit Tests for ASP.NET MVC Applications (C#)

2/20/2020 • 6 minutes to read • Edit Online

by Stephen Walther

Download PDF

Learn how to create unit tests for controller actions. In this tutorial, Stephen Walther demonstrates how to test whether a controller action returns a particular view, returns a particular set of data, or returns a different type of action result.

The goal of this tutorial is to demonstrate how you can write unit tests for the controllers in your ASP.NET MVC applications. We discuss how to build three different types of unit tests. You learn how to test the view returned by a controller action, how to test the View Data returned by a controller action, and how to test whether or not one controller action redirects you to a second controller action.

Creating the Controller under Test

Let's start by creating the controller that we intend to test. The controller, named the `ProductController`, is contained in Listing 1.

Listing 1 – `ProductController.cs`

```
using System;
using System.Web.Mvc;

namespace Store.Controllers
{
     public class ProductController : Controller
     {
          public ActionResult Index()
          {
               // Add action logic here
               throw new NotImplementedException();
          }

          public ActionResult Details(int Id)
          {

               return View("Details");
          }
     }
}
```

The `ProductController` contains two action methods named `Index()` and `Details()`. Both action methods return a view. Notice that the `Details()` action accepts a parameter named Id.

Testing the View returned by a Controller

Imagine that we want to test whether or not the `ProductController` returns the right view. We want to make sure that when the `ProductController.Details()` action is invoked, the Details view is returned. The test class in Listing 2 contains a unit test for testing the view returned by the `ProductController.Details()` action.

Listing 2 – `ProductControllerTest.cs`

```
using System.Web.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Store.Controllers;

namespace StoreTests.Controllers
{
     [TestClass]
     public class ProductControllerTest
     {
          [TestMethod]
          public void TestDetailsView()
          {
               var controller = new ProductController();
               var result = controller.Details(2) as ViewResult;
               Assert.AreEqual("Details", result.ViewName);

          }
     }
}
```

The class in Listing 2 includes a test method named `TestDetailsView()`. This method contains three lines of code. The first line of code creates a new instance of the `ProductController` class. The second line of code invokes the controller's `Details()` action method. Finally, the last line of code checks whether or not the view returned by the `Details()` action is the Details view.

The `ViewResult.ViewName` property represents the name of the view returned by a controller. One big warning about testing this property. There are two ways that a controller can return a view. A controller can explicitly return a view like this:

```
public ActionResult Details(int Id)
{
     return View("Details");
}
```

Alternatively, the name of the view can be inferred from the name of the controller action like this:

```
public ActionResult Details(int Id)
{
     return View();
}
```

This controller action also returns a view named `Details`. However, the name of the view is inferred from the action name. If you want to test the view name, then you must explicitly return the view name from the controller action.

You can run the unit test in Listing 2 by either entering the keyboard combination **Ctrl-R, A** or by clicking the **Run All Tests in Solution** button (see Figure 1). If the test passes, you'll see the Test Results window in Figure 2.

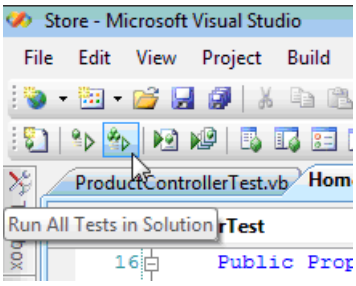


Figure 01: Run All Tests in Solution (Click to view full-size image)

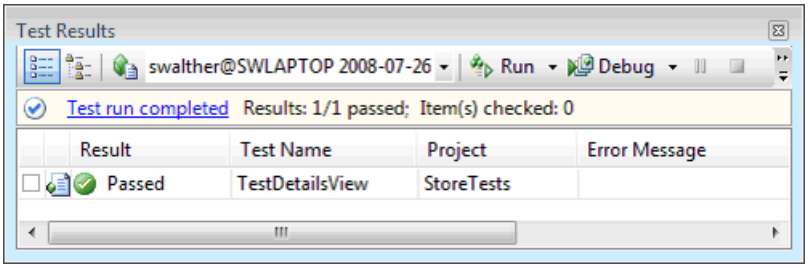


Figure 02: Success! (Click to view full-size image)

Testing the View Data returned by a Controller

An MVC controller passes data to a view by using something called `View Data`. For example, imagine that you want to display the details for a particular product when you invoke the `ProductController Details()` action. In that case, you can create an instance of a `Product` class (defined in your model) and pass the instance to the `Details` view by taking advantage of `View Data`.

The modified `ProductController` in Listing 3 includes an updated `Details()` action that returns a Product.

Listing 3 – `ProductController.cs`

```
using System;
using System.Web.Mvc;

namespace Store.Controllers
{
     public class ProductController : Controller
     {
          public ActionResult Index()
          {
               // Add action logic here
               throw new NotImplementedException();
          }

          public ActionResult Details(int Id)
          {
               var product = new Product(Id, "Laptop");
               return View("Details", product);
          }
     }
}
```

First, the `Details()` action creates a new instance of the `Product` class that represents a laptop computer. Next, the instance of the `Product` class is passed as the second parameter to the `View()` method.

You can write unit tests to test whether the expected data is contained in view data. The unit test in Listing 4 tests whether or not a Product representing a laptop computer is returned when you call the `ProductController Details()` action method.

Listing 4 – `ProductControllerTest.cs`

```
using System.Web.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Store.Controllers;

namespace StoreTests.Controllers
{
     [TestClass]
     public class ProductControllerTest
     {

          [TestMethod]
          public void TestDetailsViewData()
          {
               var controller = new ProductController();
               var result = controller.Details(2) as ViewResult;
               var product = (Product) result.ViewData.Model;
               Assert.AreEqual("Laptop", product.Name);
          }
     }
}
```

In Listing 4, the `TestDetailsView()` method tests the View Data returned by invoking the `Details()` method. The `ViewData` is exposed as a property on the `ViewResult` returned by invoking the `Details()` method. The `ViewData.Model` property contains the product passed to the view. The test simply verifies that the product contained in the View Data has the name Laptop.

Testing the Action Result returned by a Controller

A more complex controller action might return different types of action results depending on the values of the parameters passed to the controller action. A controller action can return a variety of types of action results including a `ViewResult`, `RedirectToRouteResult`, or `JsonResult`.

For example, the modified `Details()` action in Listing 5 returns the `Details` view when you pass a valid product Id to the action. If you pass an invalid product Id -- an Id with a value less than 1 -- then you are redirected to the `Index()` action.

Listing 5 – `ProductController.cs`

```
using System;
using System.Web.Mvc;
namespace Store.Controllers
{
     public class ProductController : Controller
     {
          public ActionResult Index()
          {
               // Add action logic here
               throw new NotImplementedException();
          }
          public ActionResult Details(int Id)
          {
               if (Id < 1)
                    return RedirectToAction("Index");
               var product = new Product(Id, "Laptop");
               return View("Details", product);

          }
     }
}
```

You can test the behavior of the `Details()` action with the unit test in Listing 6. The unit test in Listing 6 verifies

that you are redirected to the `Index` view when an Id with the value -1 is passed to the `Details()` method.

Listing 6 – `ProductControllerTest.cs`

```
using System.Web.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Store.Controllers;
namespace StoreTests.Controllers
{
     [TestClass]
     public class ProductControllerTest
     {
          [TestMethod]
          public void TestDetailsRedirect()
          {
               var controller = new ProductController();
               var result = (RedirectToRouteResult) controller.Details(-1);
               Assert.AreEqual("Index", result.Values["action"]);

          }
     }
}
```

When you call the `RedirectToAction()` method in a controller action, the controller action returns a `RedirectToRouteResult`. The test checks whether the `RedirectToRouteResult` will redirect the user to a controller action named `Index`.

Summary

In this tutorial, you learned how to build unit tests for MVC controller actions. First, you learned how to verify whether the right view is returned by a controller action. You learned how to use the `ViewResult.ViewName` property to verify the name of a view.

Next, we examined how you can test the contents of `View Data`. You learned how to check whether the right product was returned in `View Data` after calling a controller action.

Finally, we discussed how you can test whether different types of action results are returned from a controller action. You learned how to test whether a controller returns a `ViewResult` or a `RedirectToRouteResult`.

NEXT

Creating Unit Tests for ASP.NET MVC Applications (VB)

2/20/2020 • 6 minutes to read • Edit Online

by Stephen Walther

Download PDF

Learn how to create unit tests for controller actions. In this tutorial, Stephen Walther demonstrates how to test whether a controller action returns a particular view, returns a particular set of data, or returns a different type of action result.

The goal of this tutorial is to demonstrate how you can write unit tests for the controllers in your ASP.NET MVC applications. We discuss how to build three different types of unit tests. You learn how to test the view returned by a controller action, how to test the View Data returned by a controller action, and how to test whether or not one controller action redirects you to a second controller action.

Creating the Controller under Test

Let's start by creating the controller that we intend to test. The controller, named the `ProductController`, is contained in Listing 1.

Listing 1 – `ProductController.vb`

```
Public Class ProductController
     Inherits System.Web.Mvc.Controller

     Function Index()
          ' Add action logic here
          Throw New NotImplementedException()
     End Function

     Function Details(ByVal Id As Integer)
          Return View("Details")
     End Function
End Class
```

The `ProductController` contains two action methods named `Index()` and `Details()`. Both action methods return a view. Notice that the `Details()` action accepts a parameter named Id.

Testing the View returned by a Controller

Imagine that we want to test whether or not the `ProductController` returns the right view. We want to make sure that when the `ProductController.Details()` action is invoked, the Details view is returned. The test class in Listing 2 contains a unit test for testing the view returned by the `ProductController.Details()` action.

Listing 2 – `ProductControllerTest.vb`

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting
Imports System.Web.Mvc
Imports Store

<TestClass()> Public Class ProductControllerTest
     <TestMethod()> Public Sub TestDetailsView()
          Dim controller As New ProductController()
          Dim result As ViewResult = controller.Details(2)
          Assert.AreEqual("Details", result.ViewName)

     End Sub
End Class
```

The class in Listing 2 includes a test method named `TestDetailsView()`. This method contains three lines of code. The first line of code creates a new instance of the `ProductController` class. The second line of code invokes the controller's `Details()` action method. Finally, the last line of code checks whether or not the view returned by the `Details()` action is the Details view.

The `ViewResult.ViewName` property represents the name of the view returned by a controller. One big warning about testing this property. There are two ways that a controller can return a view. A controller can explicitly return a view like this:

```
Function Details(ByVal Id As Integer)
     Return View("Details")
End Function
```

Alternatively, the name of the view can be inferred from the name of the controller action like this:

```
Function Details(ByVal Id As Integer)
     Return View()
End Function
```

This controller action also returns a view named `Details`. However, the name of the view is inferred from the action name. If you want to test the view name, then you must explicitly return the view name from the controller action.

You can run the unit test in Listing 2 by either entering the keyboard combination **Ctrl-R, A** or by clicking the **Run All Tests in Solution** button (see Figure 1). If the test passes, you'll see the Test Results window in Figure 2.

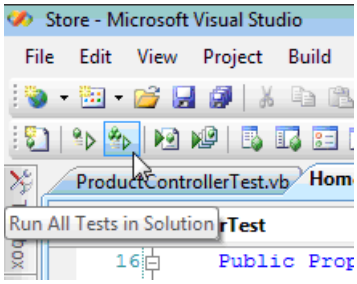


Figure 01: Run All Tests in Solution (Click to view full-size image)

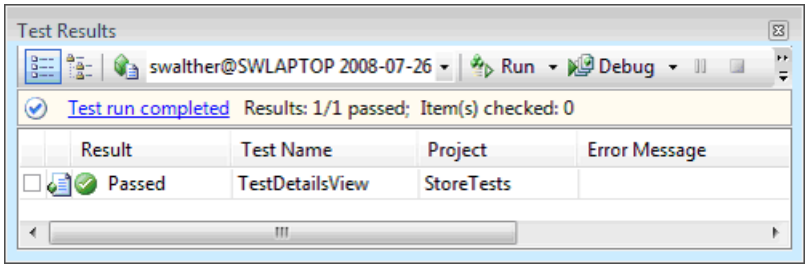


Figure 02: Success! (Click to view full-size image)

Testing the View Data returned by a Controller

An MVC controller passes data to a view by using something called `View Data`. For example, imagine that you want to display the details for a particular product when you invoke the `ProductController Details()` action. In that case, you can create an instance of a `Product` class (defined in your model) and pass the instance to the `Details` view by taking advantage of `View Data`.

The modified `ProductController` in Listing 3 includes an updated `Details()` action that returns a Product.

Listing 3 – `ProductController.vb`

```
Public Class ProductController
     Inherits System.Web.Mvc.Controller

     Function Index()
          ' Add action logic here
          Throw New NotImplementedException()
     End Function

     Function Details(ByVal Id As Integer)
          Dim product As New Product(Id, "Laptop")
          Return View("Details", product)
     End Function
End Class
```

First, the `Details()` action creates a new instance of the `Product` class that represents a laptop computer. Next, the instance of the `Product` class is passed as the second parameter to the `View()` method.

You can write unit tests to test whether the expected data is contained in view data. The unit test in Listing 4 tests whether or not a Product representing a laptop computer is returned when you call the `ProductController Details()` action method.

Listing 4 – `ProductControllerTest.vb`

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting
Imports System.Web.Mvc
Imports Store

<TestClass()> Public Class ProductControllerTest

     <TestMethod()> Public Sub TestDetailsViewData()
          Dim controller As New ProductController()
          Dim result As ViewResult = controller.Details(2)
          Dim product As Product = result.ViewData.Model
          Assert.AreEqual("Laptop", product.Name)
     End Sub
End Class
```

In Listing 4, the `TestDetailsView()` method tests the View Data returned by invoking the `Details()` method. The `ViewData` is exposed as a property on the `ViewResult` returned by invoking the `Details()` method. The `ViewData.Model` property contains the product passed to the view. The test simply verifies that the product contained in the View Data has the name Laptop.

Testing the Action Result returned by a Controller

A more complex controller action might return different types of action results depending on the values of the parameters passed to the controller action. A controller action can return a variety of types of action results

including a `ViewResult`, `RedirectToRouteResult`, or `JsonResult`.

For example, the modified `Details()` action in Listing 5 returns the `Details` view when you pass a valid product Id to the action. If you pass an invalid product Id -- an Id with a value less than 1 -- then you are redirected to the `Index()` action.

Listing 5 – `ProductController.vb`

```
Public Class ProductController
     Inherits System.Web.Mvc.Controller

     Function Index()
          ' Add action logic here
          Throw New NotImplementedException()
     End Function

     Function Details(ByVal Id As Integer)
          If Id < 1 Then
               Return RedirectToAction("Index")
          End If
          Dim product As New Product(Id, "Laptop")
          Return View("Details", product)
     End Function
End Class
```

You can test the behavior of the `Details()` action with the unit test in Listing 6. The unit test in Listing 6 verifies that you are redirected to the `Index` view when an Id with the value -1 is passed to the `Details()` method.

Listing 6 – `ProductControllerTest.vb`

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting
Imports System.Web.Mvc
Imports Store

<TestClass()> Public Class ProductControllerTest

     <TestMethod()> Public Sub TestDetailsRedirect()
          Dim controller As New ProductController()
          Dim result As RedirectToRouteResult = controller.Details(-1)
          Assert.AreEqual("Index", result.Values("action"))
     End Sub
End Class
```

When you call the `RedirectToAction()` method in a controller action, the controller action returns a `RedirectToRouteResult`. The test checks whether the `RedirectToRouteResult` will redirect the user to a controller action named `Index`.

Summary

In this tutorial, you learned how to build unit tests for MVC controller actions. First, you learned how to verify whether the right view is returned by a controller action. You learned how to use the `ViewResult.ViewName` property to verify the name of a view.

Next, we examined how you can test the contents of `View Data`. You learned how to check whether the right product was returned in `View Data` after calling a controller action.

Finally, we discussed how you can test whether different types of action results are returned from a controller action. You learned how to test whether a controller returns a `ViewResult` or a `RedirectToRouteResult`.

PREVIOUS

Contact Manager

2/20/2020 • 2 minutes to read • Edit Online

How to build an entire Contact Management application from start to finish.

- Iteration #1 – Create the Application (C#)
- Iteration #2 – Make the application look nice (C#)
- Iteration #3 – Add form validation (C#)
- Iteration #4 – Make the application loosely coupled (C#)
- Iteration #5 – Create unit tests (C#)
- Iteration #6 – Use test-driven development (C#)
- Iteration #7 – Add Ajax functionality (C#)
- Iteration #1 – Create the Application (VB)
- Iteration #2 – Make the application look nice (VB)
- Iteration #3 – Add form validation (VB)
- Iteration #4 – Make the application loosely coupled (VB)
- Iteration #5 – Create unit tests (VB)
- Iteration #6 – Use test-driven development (VB)
- Iteration #7 – Add Ajax functionality (VB)

Iteration #1 – Create the Application (C#)

4/10/2020 • 24 minutes to read • Edit Online

by Microsoft

Download Code

> In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).

Building a Contact Management ASP.NET MVC Application (VB)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information - names, phone numbers and email addresses - for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 - Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 - Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 - Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 - Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 - Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 - Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 - Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

In this first iteration, we build the basic application. The goal is to build the Contact Manager in the fastest and simplest way possible. In later iterations, we improve the design of the application.

The Contact Manager application is a basic database-driven application. You can use the application to create new contacts, edit existing contacts, and delete contacts.

In this iteration, we complete the following steps:

1. ASP.NET MVC application
2. Create a database to store our contacts
3. Generate a model class for our database with the Microsoft Entity Framework
4. Create a controller action and view that enables us to list all of the contacts in the database
5. Create controller actions and a view that enables us to create a new contact in the database
6. Create controller actions and a view that enables us to edit an existing contact in the database
7. Create controller actions and a view that enables us to delete an existing contact in the database

Software Prerequisites

In ASP.NET MVC applications, you must have either Visual Studio 2008 or Visual Web Developer 2008 installed on your computer (Visual Web Developer is a free version of Visual Studio that does not include all of the advanced features of Visual Studio). You can download either the trial version of Visual Studio 2008 or Visual Web Developer from the following address:

https://www.asp.net/downloads/essential/

> **NOTE**
> For ASP.NET MVC applications with Visual Web Developer, you must have Visual Web Developer Service Pack 1 installed. Without Service Pack 1, you cannot create Web Application Projects.

ASP.NET MVC framework. You can download the ASP.NET MVC framework from the following address:

https://www.asp.net/mvc

In this tutorial, we use the Microsoft Entity Framework to access a database. The Entity Framework is included with .NET Framework 3.5 Service Pack 1. You can download this service pack from the following location:

https://www.microsoft.com/downloads/details.aspx?familyid=ab99342f-5d1a-413d-8319-81da479ab0d7&displaylang=en

As an alternative to performing each of these downloads one by one, you can take advantage of the Web Platform Installer (Web PI). You can download the Web PI from the following address:

https://www.asp.net/downloads/essential/

ASP.NET MVC Project

ASP.NET MVC Web Application Project. Launch Visual Studio and select the menu option **File, New Project**. The **New Project** dialog appears (see Figure 1). Select the **Web** project type and the **ASP.NET MVC Web Application** template. Name your new project *ContactManager* and click the OK button.

Make sure that you have .NET Framework 3.5 selected from the dropdown list at the top right of the **New Project** dialog. Otherwise, the ASP.NET MVC Web Application template won't appear.

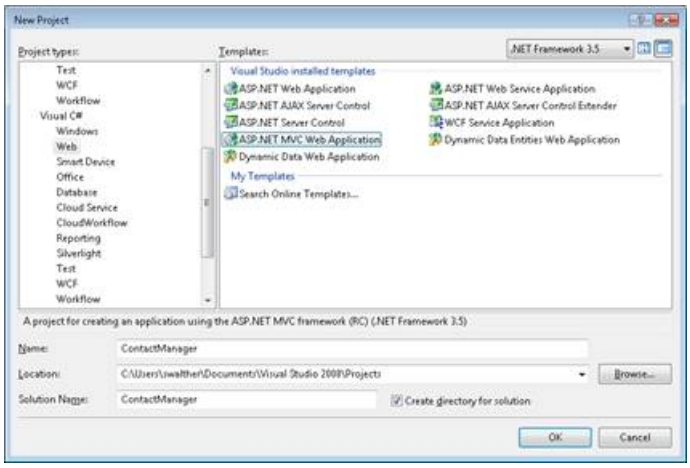


Figure 01: The New Project dialog(Click to view full-size image)

ASP.NET MVC application, the **Create Unit Test Project** dialog appears. You can use this dialog to indicate that you want to create and add a unit test project to your solution when you create your ASP.NET MVC application. Although we won't be building unit tests in this iteration, you should select the option **Yes, create a unit test project** because we plan to add unit tests in a later iteration. Adding a Test project when you first create a new ASP.NET MVC project is much easier than adding a Test project after the ASP.NET MVC project has been created.

> **NOTE**
>
> Because Visual Web Developer does not support Test projects, you do not get the Create Unit Test Project dialog when using Visual Web Developer.

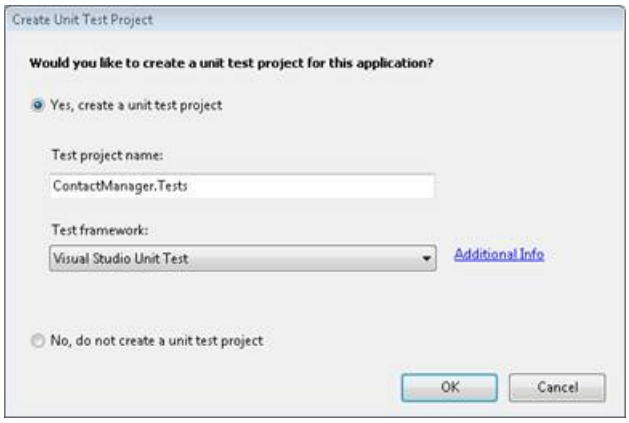


Figure 02: The Create Unit Test Project dialog (Click to view full-size image)

ASP.NET MVC application appears in the Visual Studio Solution Explorer window (see Figure 3). If you don t see the Solution Explorer window then you can open this window by selecting the menu option **View, Solution Explorer**. Notice that the solution contains two projects: the ASP.NET MVC project and the Test project. The ASP.NET MVC project is named ContactManager and the Test project is named ContactManager.Tests.

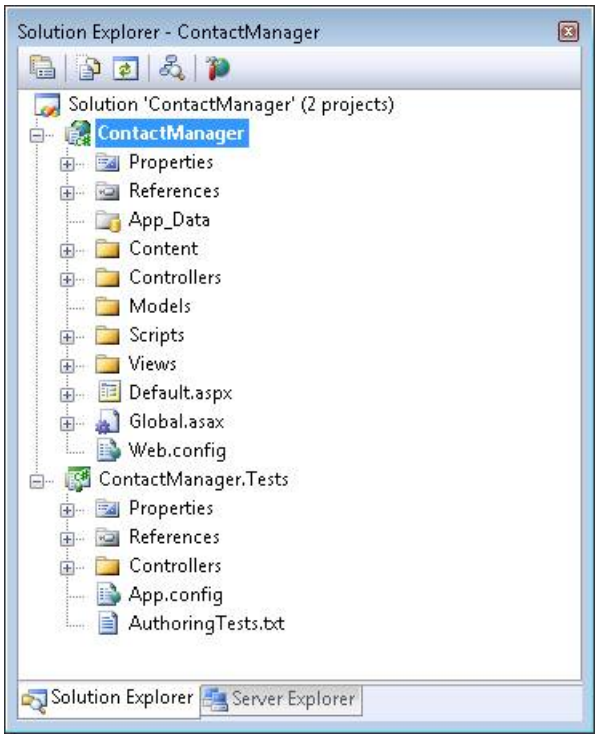


Figure 03: The Solution Explorer window (Click to view full-size image)

Deleting the Project Sample Files

The ASP.NET MVC project template includes sample files for controllers and views. Before creating a new ASP.NET MVC application, you should delete these files. You can delete files and folders in the Solution Explorer window by right-clicking a file or folder and selecting the menu option **Delete**.

You need to delete the following files from the ASP.NET MVC project:

- \Controllers\HomeController.cs
- \Views\Home\About.aspx
- \Views\Home\Index.aspx

And, you need to delete the following file from the Test project:

\Controllers\HomeControllerTest.cs

Creating the Database

The Contact Manager application is a database-driven web application. We use a database to store the contact information.

The ASP.NET MVC framework with any modern database including Microsoft SQL Server, Oracle, MySQL, and IBM DB2 databases. In this tutorial, we use a Microsoft SQL Server database. When you install Visual Studio, you are provided with the option of installing Microsoft SQL Server Express which is a free version of the Microsoft SQL Server database.

Create a new database by right-clicking the App_Data folder in the Solution Explorer window and selecting the menu option **Add, New Item**. In the **Add New Item** dialog, select the **Data** category and the **SQL Server Database** template (see Figure 4). Name the new database ContactManagerDB.mdf and click the OK button.

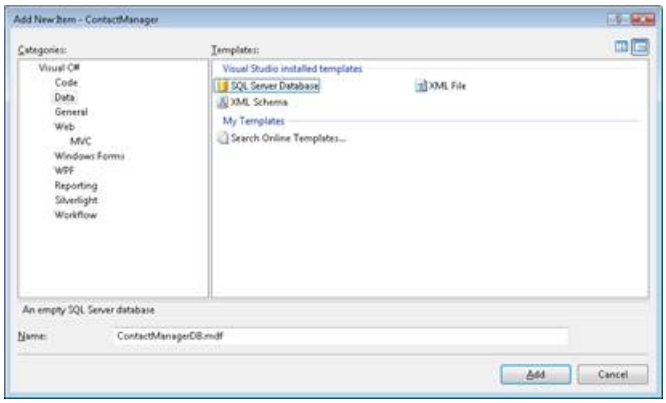


Figure 04: Creating a new Microsoft SQL Server Express database (Click to view full-size image)

After you create the new database, the database appears in the App_Data folder in the Solution Explorer window. Double-click the ContactManager.mdf file to open the Server Explorer window and connect to the database.

> **NOTE**
>
> The Server Explorer window is called the Database Explorer window in the case of Microsoft Visual Web Developer.

You can use the Server Explorer window to create new database objects such as database tables, views, triggers, and stored procedures. Right-click the Tables folder and select the menu option **Add New Table**. The Database Table Designer appears (see Figure 5).

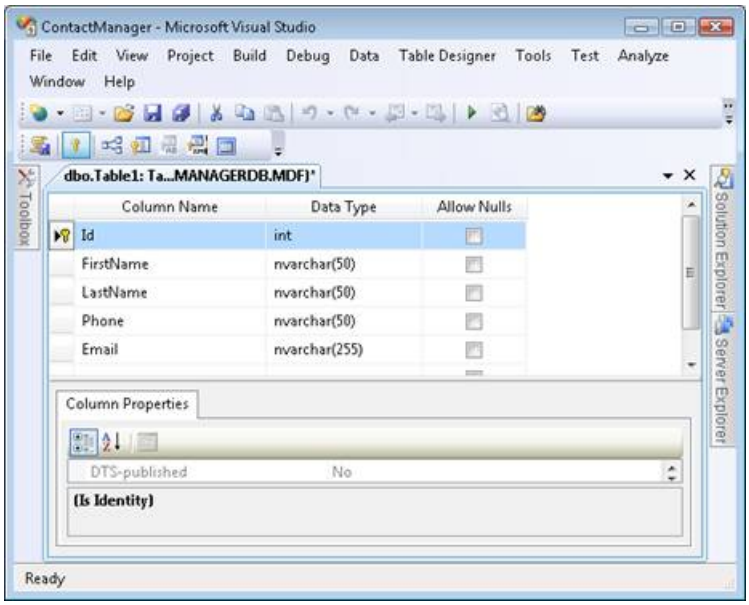


Figure 05: The Database Table Designer (Click to view full-size image)

We need to create a table that contains the following columns:

COLUMN NAME	DATA TYPE	ALLOW NULLS
Id	int	false
FirstName	nvarchar(50)	false
LastName	nvarchar(50)	false
Phone	nvarchar(50)	false

COLUMN NAME	DATA TYPE	ALLOW NULLS
Email	nvarchar(255)	false

The first column, the Id column, is special. You need to mark the Id column as an Identity column and a Primary Key column. You indicate that a column is an Identity column by expanding Column Properties (look at the bottom of Figure 6) and scrolling down to the Identity Specification property. Set the **(Is Identity)** property to the value **Yes**.

You mark a column as a Primary Key column by selecting the column and clicking the button with the icon of a key. After a column is marked as a Primary Key column, an icon of a key appears next to the column (see Figure 6).

After you finish creating the table, click the Save button (the button with an icon of a floppy) to save the new table. Give your new table the name *Contacts*.

After finish creating the Contacts database table, you should add some records to the table. Right-click the Contacts table in the Server Explorer window and select the menu option **Show Table Data**. Enter one or more contacts in the grid that appears.

Creating the Data Model

The ASP.NET MVC application consists of Models, Views, and Controllers. We start by creating a Model class that represents the Contacts table that we created in the previous section.

In this tutorial, we use the Microsoft Entity Framework to generate a model class from the database automatically.

> **NOTE**
> The ASP.NET MVC framework is not tied to the Microsoft Entity Framework in any way. You can use ASP.NET MVC with alternative database access technologies including NHibernate, LINQ to SQL, or ADO.NET.

Follow these steps to create the data model classes:

1. Right-click the Models folder in the Solution Explorer window and select **Add, New Item**. The **Add New Item** dialog appears (see Figure 6).
2. Select the **Data** category and the **ADO.NET Entity Data Model** template. Name your data model *ContactManagerModel.edmx* and click the **Add** button. The Entity Data Model wizard appears (see Figure 7).
3. In the **Choose Model Contents** step, select **Generate from database** (see Figure 7).
4. In the **Choose Your Data Connection** step, select the ContactManagerDB.mdf database and enter the name *ContactManagerDBEntities* for the Entity Connection Settings (see Figure 8).
5. In the **Choose Your Database Objects** step, select the checkbox labeled Tables (see Figure 9). The data model will include all tables contained in your database (there is just one, the Contacts table). Enter the namespace *Models*. Click the Finish button to complete the wizard.

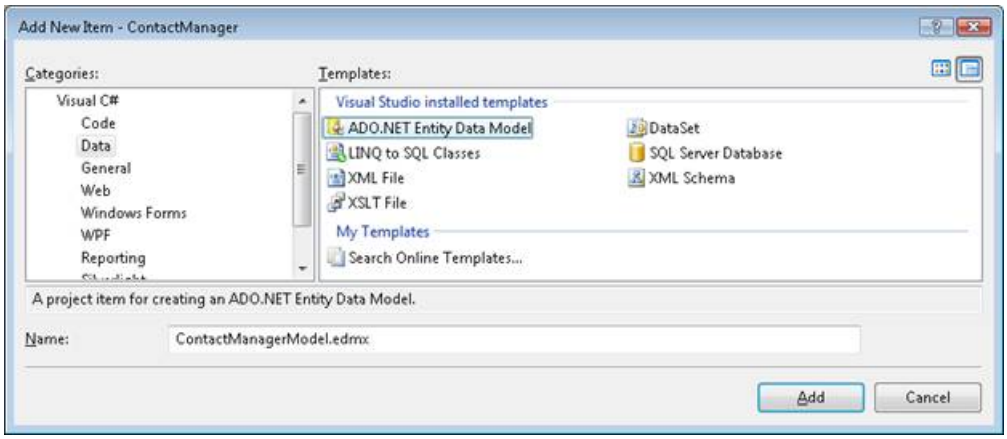


Figure 06: The Add New Item dialog (Click to view full-size image)

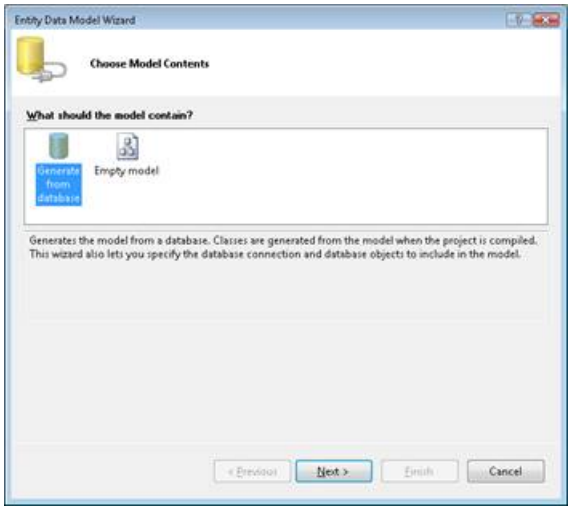


Figure 07: Choose Model Contents (Click to view full-size image)

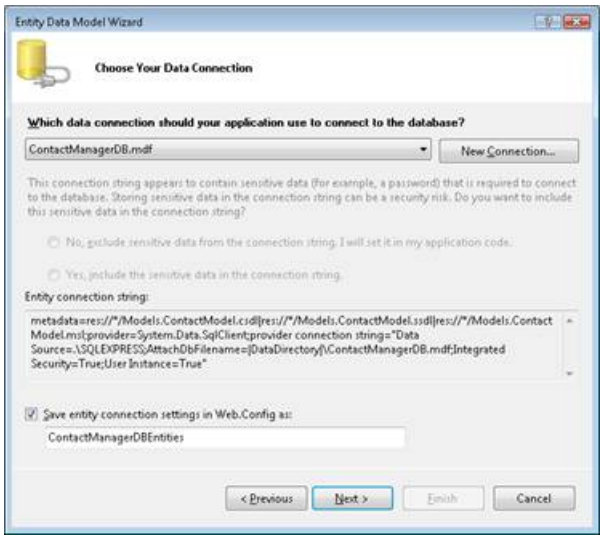


Figure 08: Choose Your Data Connection (Click to view full-size image)

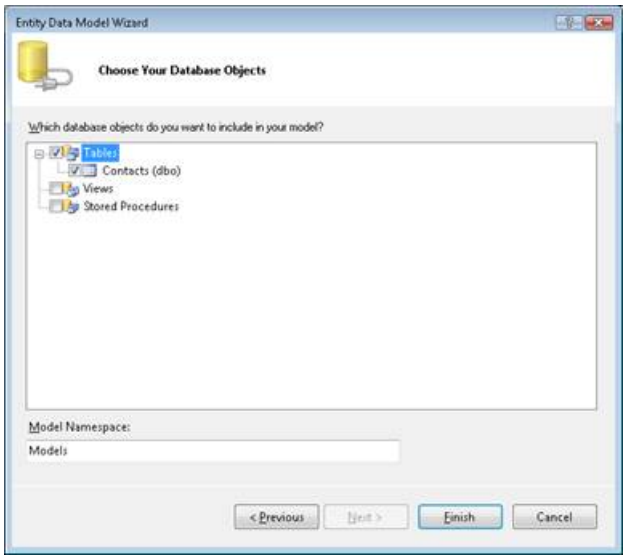


Figure 09: Choose Your Database Objects (Click to view full-size image)

After you complete the Entity Data Model Wizard, the Entity Data Model Designer appears. The designer displays a class that corresponds to each table being modeled. You should see one class named Contacts.

The Entity Data Model wizard generates class names based on database table names. You almost always need to change the name of the class generated by the wizard. Right-click the Contacts class in the designer and select the menu option **Rename**. Change the name of the class from Contacts (plural) to Contact (singular). After you change the class name, the class should appear like Figure 10.

Figure 10: The Contact class (Click to view full-size image)

At this point, we have created our database model. We can use the Contact class to represent a particular contact record in our database.

Creating the Home Controller

The next step is to create our Home controller. The Home controller is the default controller invoked in an ASP.NET MVC application.

Create the Home controller class by right-clicking the Controllers folder in the Solution Explorer window and selecting the menu option **Add, Controller** (see Figure 11). Notice the checkbox labeled **Add action methods for Create, Update, and Details scenarios**. Make sure that this checkbox is checked before clicking the **Add** button.

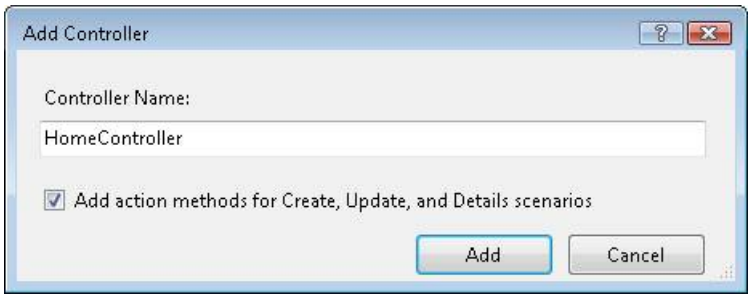


Figure 11: Adding the Home controller (Click to view full-size image)

When you create the Home controller, you get the class in Listing 1.

Listing 1 - Controllers\HomeController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Mvc.Ajax;

namespace ContactManager.Controllers
{
    public class HomeController : Controller
    {
        //
        // GET: /Home/

        public ActionResult Index()
        {
            return View();
        }

        //
        // GET: /Home/Details/5

        public ActionResult Details(int id)
        {
            return View();
        }

        //
        // GET: /Home/Create

        public ActionResult Create()
        {
            return View();
        }

        //
        // POST: /Home/Create

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create(FormCollection collection)
        {
            try
            {
                // TODO: Add insert logic here

                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
```

```
        }

        //
        // GET: /Home/Edit/5

        public ActionResult Edit(int id)
        {
            return View();
        }

        //
        // POST: /Home/Edit/5

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Edit(int id, FormCollection collection)
        {
            try
            {
                // TODO: Add update logic here

                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }
    }
}
```

Listing the Contacts

In order to display the records in the Contacts database table, we need to create an Index() action and an Index view.

The Home controller already contains an Index() action. We need to modify this method so that it looks like Listing 2.

Listing 2 - Controllers\HomeController.cs

```
public class HomeController : Controller
{

    private ContactManagerDBEntities _entities = new ContactManagerDBEntities();

    //
    // GET: /Home/

    public ActionResult Index()
    {
        return View(_entities.ContactSet.ToList());
    }
…
```

Notice that the Home controller class in Listing 2 contains a private field named _entities. The _entities field represents the entities from the data model. We use the _entities field to communicate with the database.

The Index() method returns a view that represents all of the contacts from the Contacts database table. The expression _entities.ContactSet.ToList() returns the list of contacts as a generic list.

Now that we ve created the Index controller, we next need to create the Index view. Before creating the Index view, compile your application by selecting the menu option **Build, Build Solution**. You should always compile your project before adding a view in order for the list of model classes to be displayed in the **Add View** dialog.

You create the Index view by right-clicking the Index() method and selecting the menu option **Add View** (see Figure 12). Selecting this menu option opens the **Add View** dialog (see Figure 13).

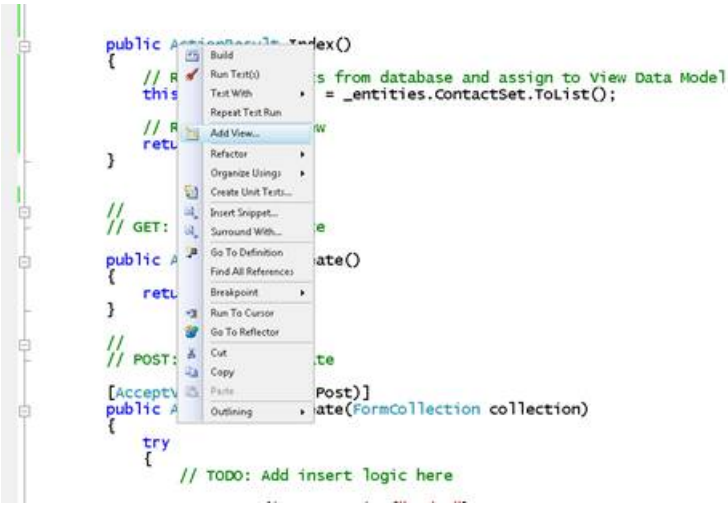


Figure 12: Adding the Index view (Click to view full-size image)

In the **Add View** dialog, check the checkbox labeled **Create a strongly-typed view**. Select the View data class ContactManager.Models.Contact and the View content List. Selecting these options generates a view that displays a list of Contact records.

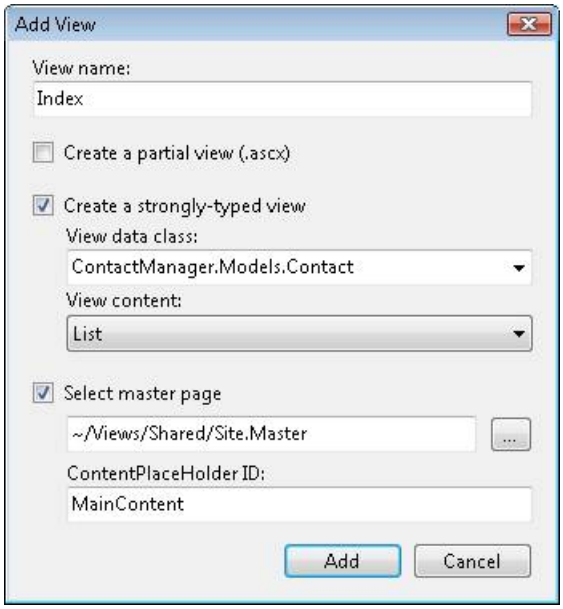


Figure 13: The Add View dialog (Click to view full-size image)

When you click the **Add** button, the Index view in Listing 3 is generated. Notice the <%@ Page %> directive that appears at the top of the file. The Index view inherits from the ViewPage<IEnumerable<ContactManager.Models.Contact>> class. In other words, the Model class in the view represents a list of Contact entities.

The body of the Index view contains a foreach loop that iterates through each of the contacts represented by the Model class. The value of each property of the Contact class is displayed within an HTML table.

Listing 3 - Views\Home\Index.aspx (unmodified)

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<IEnumerable<ContactManager.Models.Contact>>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Index</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Index</h2>

    <table>
        <tr>
            <th></th>
            <th>
                Id
            </th>
            <th>
                FirstName
            </th>
            <th>
                LastName
            </th>
            <th>
                Phone
            </th>
            <th>
                Email
            </th>
        </tr>

    <% foreach (var item in Model) { %>

        <tr>
            <td>
                <%= Html.ActionLink("Edit", "Edit", new { id=item.Id  }) %> |
                <%= Html.ActionLink("Details", "Details", new { id=item.Id  })%>
            </td>
            <td>
                <%= Html.Encode(item.Id) %>
            </td>
            <td>
                <%= Html.Encode(item.FirstName) %>
            </td>
            <td>
                <%= Html.Encode(item.LastName) %>
            </td>
            <td>
                <%= Html.Encode(item.Phone) %>
            </td>
            <td>
                <%= Html.Encode(item.Email) %>
            </td>
        </tr>

    <% } %>

    </table>

    <p>
        <%= Html.ActionLink("Create New", "Create") %>
    </p>

</asp:Content>
```

We need to make one modification to the Index view. Because we are not creating a Details view, we can remove

the Details link. Find and remove the following code from the Index view:

{ id=item.Id })%>

After you modify the Index view, you can run the Contact Manager application. Select the menu option Debug, Start Debugging or simply press F5. The first time you run the application, you get the dialog in Figure 14. Select the option **Modify the Web.config file to enable debugging** and click the OK button.

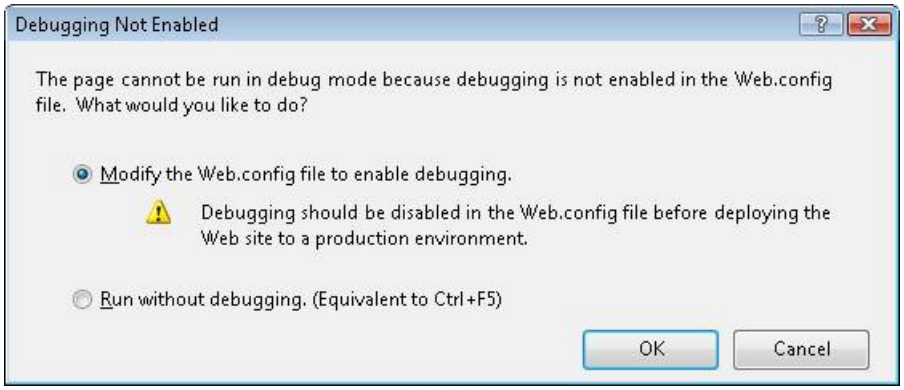


Figure 14: Enabling debugging (Click to view full-size image)

The Index view is returned by default. This view lists all of the data from the Contacts database table (see Figure 15).

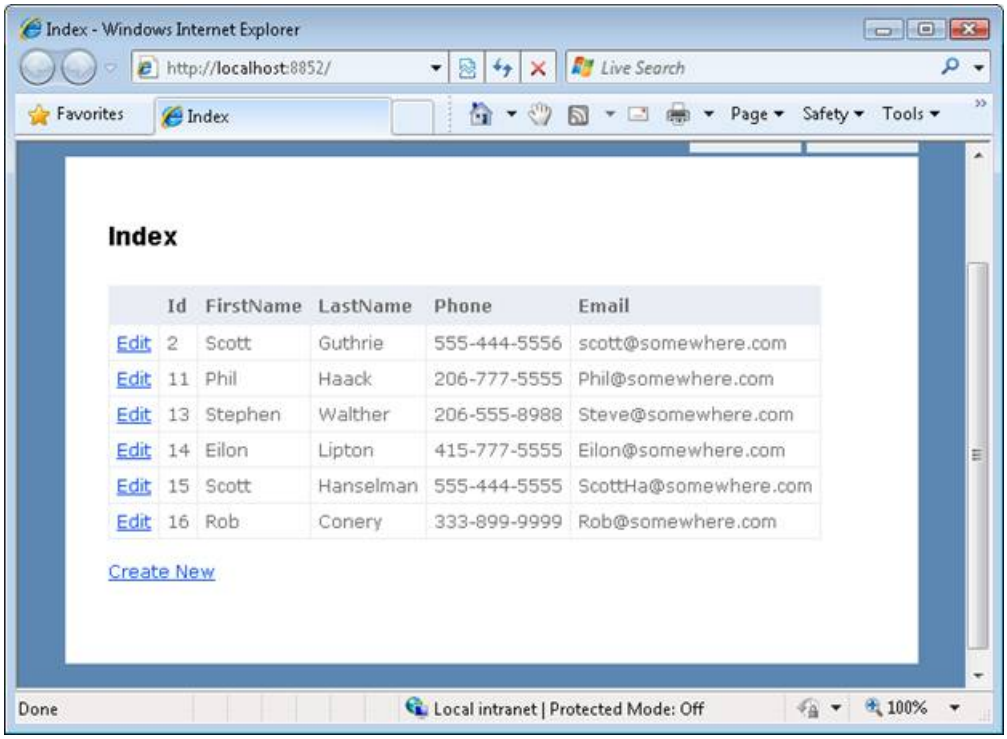


Figure 15: The Index view (Click to view full-size image)

Notice that the Index view includes a link labeled Create New at the bottom of the view. In the next section, you learn how to create new contacts.

Creating New Contacts

To enable users to create new contacts, we need to add two Create() actions to the Home controller. We need to create one Create() action that returns an HTML form for creating a new contact. We need to create a second Create() action that performs the actual database insert of the new contact.

The new Create() methods that we need to add to the Home controller are contained in Listing 4.

Listing 4 - Controllers\HomeController.cs (with Create methods)

```
//
// GET: /Home/Create

public ActionResult Create()
{
    return View();
}

//
// POST: /Home/Create

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create([Bind(Exclude = "Id")] Contact contactToCreate)
{
    if (!ModelState.IsValid)
        return View();

    try
    {
        _entities.AddToContactSet(contactToCreate);
        _entities.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

The first Create() method can be invoked with an HTTP GET while the second Create() method can be invoked only by an HTTP POST. In other words, the second Create() method can be invoked only when posting an HTML form. The first Create() method simply returns a view that contains the HTML form for creating a new contact. The second Create() method is much more interesting: it adds the new contact to the database.

Notice that the second Create() method has been modified to accept an instance of the Contact class. The form values posted from the HTML form are bound to this Contact class by the ASP.NET MVC framework automatically. Each form field from the HTML Create form is assigned to a property of the Contact parameter.

Notice that the Contact parameter is decorated with a [Bind] attribute. The [Bind] attribute is used to exclude the Contact Id property from binding. Because the Id property represents an Identity property, we don t want to set the Id property.

In the body of the Create() method, the Entity Framework is used to insert the new Contact into the database. The new Contact is added to the existing set of Contacts and the SaveChanges() method is called to push these changes back to the underlying database.

You can generate an HTML form for creating new Contacts by right-clicking either of the two Create() methods and selecting the menu option **Add View** (see Figure 16).

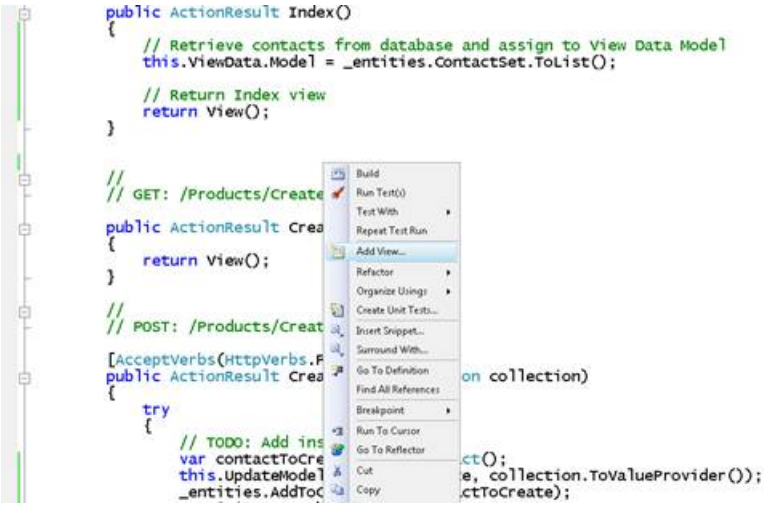


Figure 16: Adding the Create view (Click to view full-size image)

In the **Add View** dialog, select the **ContactManager.Models.Contact** class and the **Create** option for view content (see Figure 17). When you click the **Add** button, a Create view is generated automatically.

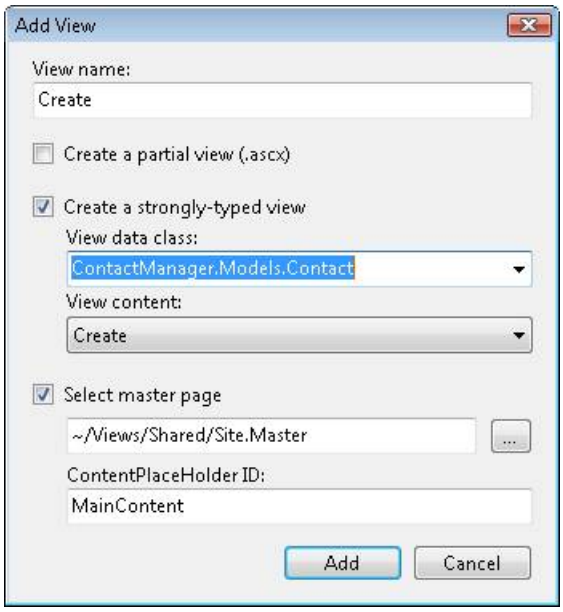


Figure 17: Seeing a page explode (Click to view full-size image)

The Create view contains form fields for each of the properties of the Contact class. The code for the Create view is included in Listing 5.

Listing 5 - Views\Home\Create.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.Contact>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Create</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Create</h2>

    <%= Html.ValidationSummary() %>

    <% using (Html.BeginForm()) {%>

        <fieldset>
            <legend>Fields</legend>
            <p>
                <label for="FirstName">FirstName:</label>
                <%= Html.TextBox("FirstName") %>
                <%= Html.ValidationMessage("FirstName", "*") %>
            </p>
            <p>
                <label for="LastName">LastName:</label>
                <%= Html.TextBox("LastName") %>
                <%= Html.ValidationMessage("LastName", "*") %>
            </p>
            <p>
                <label for="Phone">Phone:</label>
                <%= Html.TextBox("Phone") %>
                <%= Html.ValidationMessage("Phone", "*") %>
            </p>
            <p>
                <label for="Email">Email:</label>
                <%= Html.TextBox("Email") %>
                <%= Html.ValidationMessage("Email", "*") %>
            </p>
            <p>
                <input type="submit" value="Create" />
            </p>
        </fieldset>

    <% } %>

    <div>
        <%=Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

After you modify the Create() methods and add the Create view, you can run the Contact Manger application and create new contacts. Click the **Create New** link that appears in the Index view to navigate to the Create view. You should see the view in Figure 18.

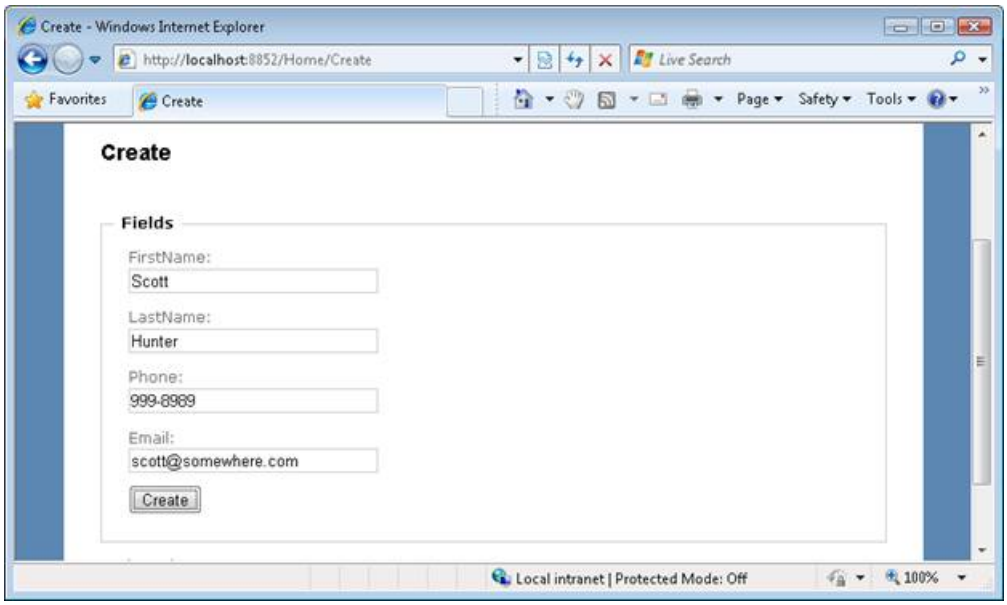


Figure 18: The Create View (Click to view full-size image)

Editing Contacts

Adding the functionality for editing a contact record is very similar to adding the functionality for creating new contact records. First, we need to add two new Edit methods to the Home controller class. These new Edit() methods are contained in Listing 6.

Listing 6 - Controllers\HomeController.cs (with Edit methods)

```
//
// GET: /Home/Edit/5

public ActionResult Edit(int id)
{
    var contactToEdit = (from c in _entities.ContactSet
                         where c.Id == id
                         select c).FirstOrDefault();

    return View(contactToEdit);
}

//
// POST: /Home/Edit/5

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(Contact contactToEdit)
{
    if (!ModelState.IsValid)
        return View();

    try
    {
        var originalContact = (from c in _entities.ContactSet
                               where c.Id == contactToEdit.Id
                               select c).FirstOrDefault();
        _entities.ApplyPropertyChanges(originalContact.EntityKey.EntitySetName, contactToEdit);
        _entities.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
}
```

The first Edit() method is invoked by an HTTP GET operation. An Id parameter is passed to this method which represents the Id of the contact record being edited. The Entity Framework is used to retrieve a contact that matches the Id. A view that contains an HTML form for editing a record is returned.

The second Edit() method performs the actual update to the database. This method accepts an instance of the Contact class as a parameter. The ASP.NET MVC framework binds the form fields from the Edit form to this class automatically. Notice that you don t include the[Bind] attribute when editing a Contact (we need the value of the Id property).

The Entity Framework is used to save the modified Contact to the database. The original Contact must be retrieved from the database first. Next, the Entity Framework ApplyPropertyChanges() method is called to record the changes to the Contact. Finally, the Entity Framework SaveChanges() method is called to persist the changes to the underlying database.

You can generate the view that contains the Edit form by right-clicking the Edit() method and selecting the menu option Add View. In the Add View dialog, select the **ContactManager.Models.Contact** class and the **Edit** view content (see Figure 19).

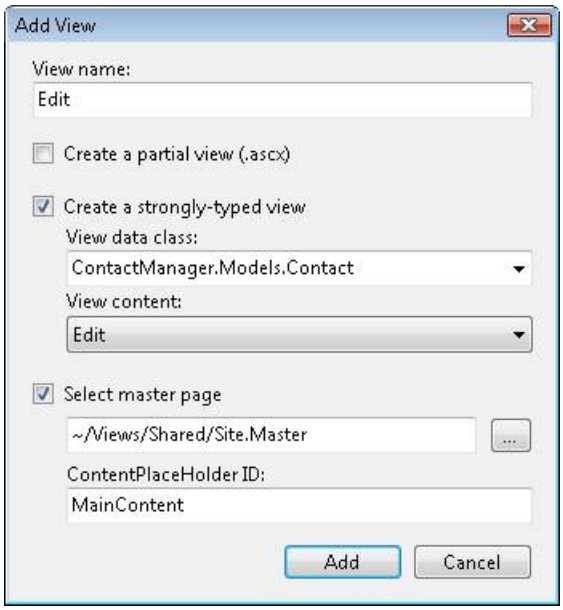


Figure 19: Adding an Edit View (Click to view full-size image)

When you click the Add button, a new Edit view is generated automatically. The HTML form that is generated contains fields that correspond to each of the properties of the Contact class (see Listing 7).

Listing 7 - Views\Home\Edit.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.Contact>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Edit</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Edit</h2>

    <%= Html.ValidationSummary() %>

    <% using (Html.BeginForm()) {%>

        <fieldset>
            <legend>Fields</legend>
            <p>
                <label for="FirstName">FirstName:</label>
                <%= Html.TextBox("FirstName") %>
                <%= Html.ValidationMessage("FirstName", "*") %>
            </p>
            <p>
                <label for="LastName">LastName:</label>
                <%= Html.TextBox("LastName") %>
                <%= Html.ValidationMessage("LastName", "*") %>
            </p>
            <p>
                <label for="Phone">Phone:</label>
                <%= Html.TextBox("Phone") %>
                <%= Html.ValidationMessage("Phone", "*") %>
            </p>
            <p>
                <label for="Email">Email:</label>
                <%= Html.TextBox("Email") %>
                <%= Html.ValidationMessage("Email", "*") %>
            </p>
            <p>
                <input type="submit" value="Save" />
            </p>
        </fieldset>

    <% } %>

    <div>
        <%=Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

Deleting Contacts

If you want to delete contacts then you need to add two Delete() actions to the Home controller class. The first Delete() action displays a delete confirmation form. The second Delete() action performs the actual delete.

> **NOTE**
>
> Later, in Iteration #7, we modify the Contact Manager so that it supports a one step Ajax delete.

The two new Delete() methods are contained in Listing 8.

Listing 8 - Controllers\HomeController.cs (Delete methods)

```
//
// GET: /Home/Delete/5

public ActionResult Delete(int id)
{
    var contactToDelete = (from c in _entities.ContactSet
                           where c.Id == id
                           select c).FirstOrDefault();

    return View(contactToDelete);
}

//
// POST: /Home/Delete/5

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Delete(Contact contactToDelete)
{
    try
    {
        var originalContact = (from c in _entities.ContactSet
                               where c.Id == contactToDelete.Id
                               select c).FirstOrDefault();

        _entities.DeleteObject(originalContact);
        _entities.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

The first Delete() method returns a confirmation form for deleting a contact record from the database (see Figure20). The second Delete() method performs the actual delete operation against the database. After the original contact has been retrieved from the database, the Entity Framework DeleteObject() and SaveChanges() methods are called to perform the database delete.

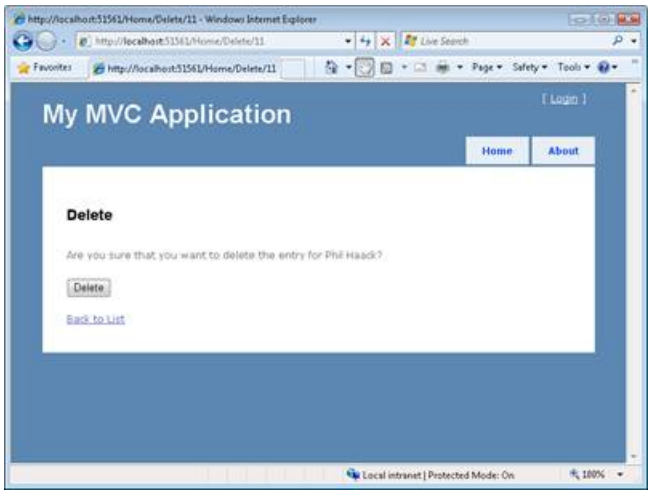


Figure 20: The delete confirmation view (Click to view full-size image)

We need to modify the Index view so that it contains a link for deleting contact records (see Figure 21). You need to add the following code to the same table cell that contains the Edit link:

Html.ActionLink({ id=item.Id }) %>

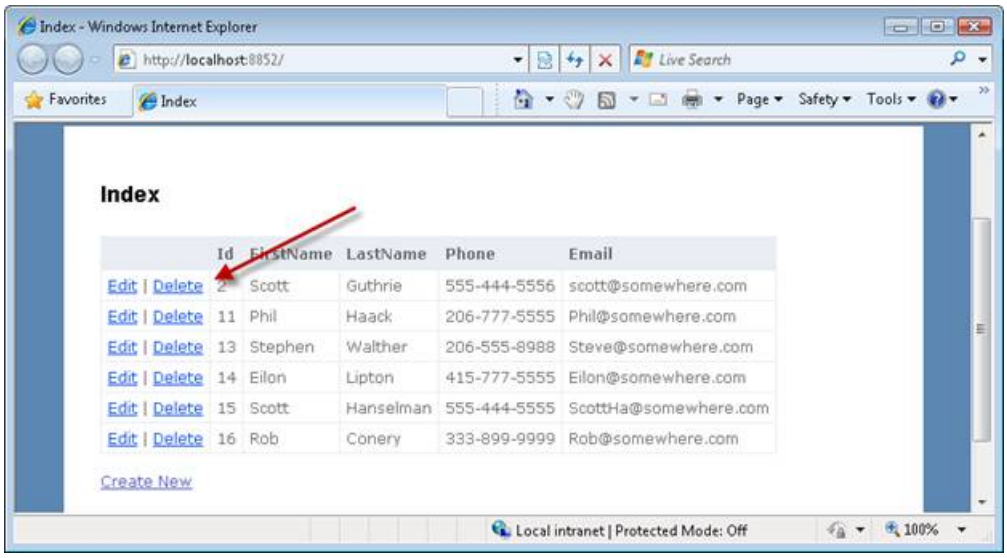


Figure 21: Index view with Edit link (Click to view full-size image)

Next, we need to create the delete confirmation view. Right-click the Delete() method in the Home controller class and select the menu option Add View. The Add View dialog appears (see Figure 22).

Unlike in the case of the List, Create, and Edit views, the Add View dialog does not contain an option to create a Delete view. Instead, select the **ContactManager.Models.Contact** data class and the **Empty** view content. Selecting the Empty view content option will require us to create the view ourselves.

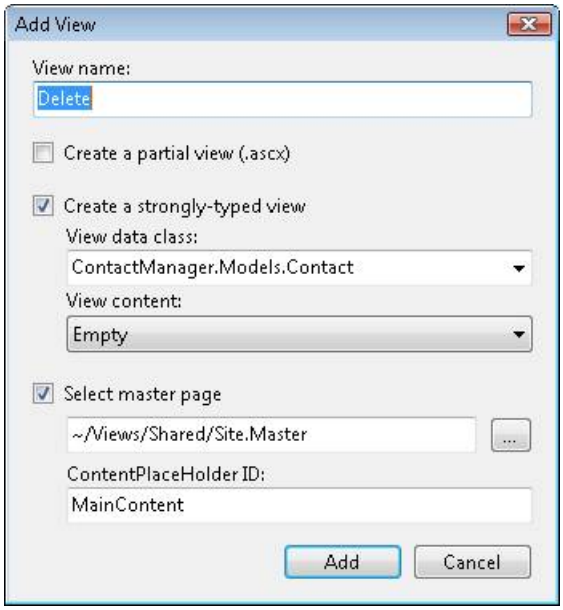


Figure 22: Adding the delete confirmation view (Click to view full-size image)

The content of the Delete view is contained in Listing 9. This view contains a form that confirms whether or not a particular contact should be deleted (see Figure 21).

Listing 9 - Views\Home\Delete.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.Contact>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Delete</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Delete</h2>

    <p>
    Are you sure that you want to delete the entry for
    <%= Model.FirstName %> <%= Model.LastName %>?
    </p>

    <% using (Html.BeginForm(new { Id = Model.Id }))
       { %>
       <p>
            <input type="submit" value="Delete" />
        </p>
    <% } %>

    <div>
        <%=Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

Changing the Name of the Default Controller

It might bother you that the name of our controller class for working with contacts is named the HomeController class. Shouldn't the controller be named ContactController?

This issue is easy enough to fix. First, we need to refactor the name of the Home controller. Open the HomeController class in the Visual Studio Code Editor, right click the name of the class and select the menu option **Refactor, Rename**. Selecting this menu option opens the Rename dialog.

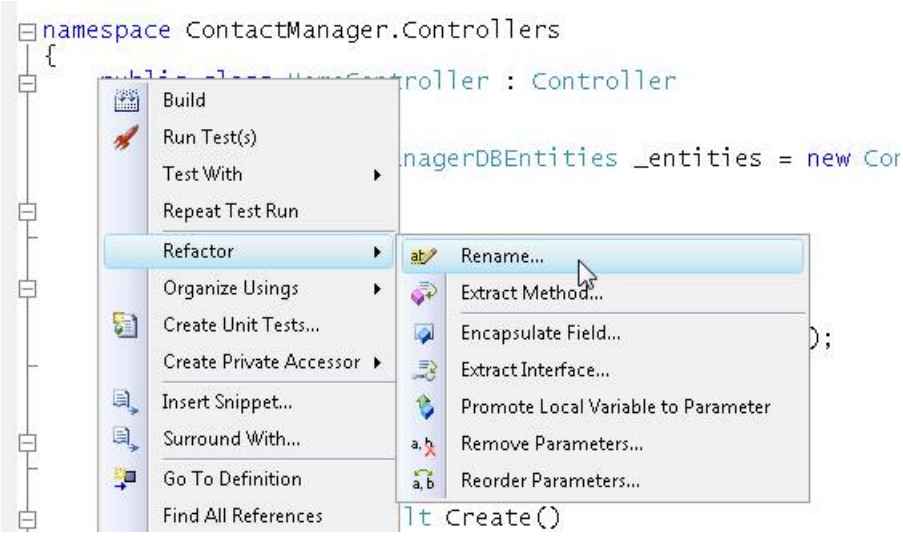


Figure 23: Refactoring a controller name (Click to view full-size image)

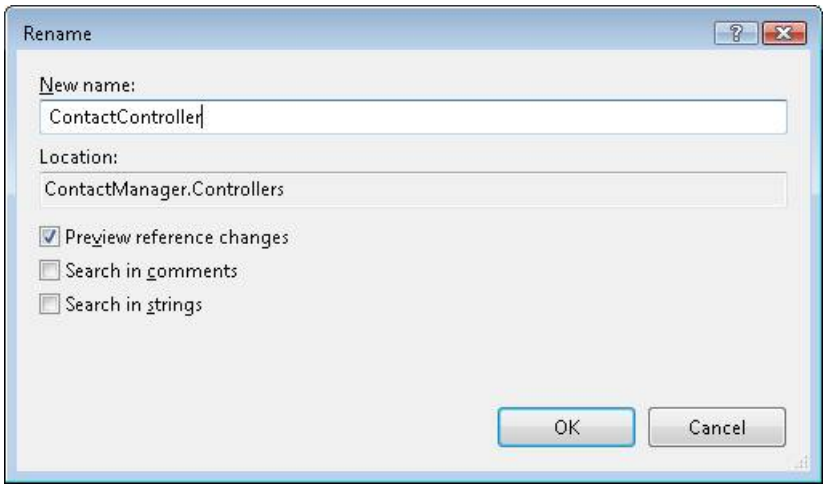


Figure 24: Using the Rename dialog (Click to view full-size image)

If you rename your controller class, Visual Studio will update the name of the folder in the Views folder as well. Visual Studio will rename the \Views\Home folder to the \Views\Contact folder.

After you make this change, your application will no longer have a Home controller. When you run your application, you'll get the error page in Figure 25.

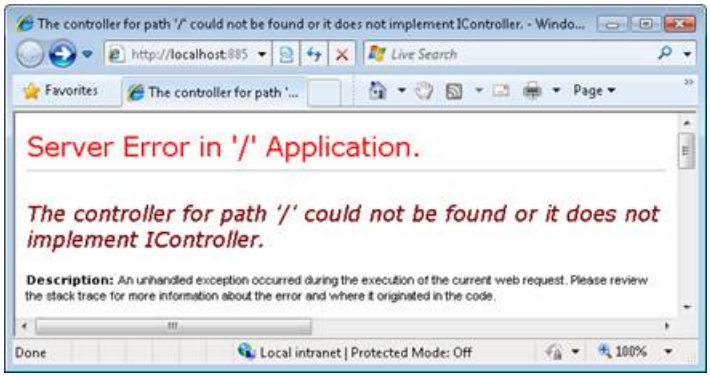


Figure 25: No default controller (Click to view full-size image)

We need to update the default route in the Global.asax file to use the Contact controller instead of the Home controller. Open the Global.asax file and modify the default controller used by the default route (see Listing 10).

Listing 10 - Global.asax.cs

```
using System.Web.Mvc;
using System.Web.Routing;

namespace ContactManager
{
    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default",                                          // Route name
                "{controller}/{action}/{id}",                       // URL with parameters
                new { controller = "Contact", action = "Index", id = "" }  // Parameter defaults
            );

        }

        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

After you make these changes, the Contact Manager will run correctly. Now, it will use the Contact controller class as the default controller.

Summary

In this first iteration, we created a basic Contact Manager application in the fastest way possible. We took advantage of Visual Studio to generate the initial code for our controllers and views automatically. We also took advantage of the Entity Framework to generate our database model classes automatically.

Currently, we can list, create, edit, and delete contact records with the Contact Manager application. In other words, we can perform all of the basic database operations required by a database-driven web application.

Unfortunately, our application has some problems. First and I hesitate to admit this, the Contact Manager application is not the most attractive application. It needs some design work. In the next iteration, we'll look at how we can change the default view master page and cascading style sheet to improve the appearance of the application.

Second, we have not implemented any form validation. For example, there is nothing to prevent you from submitting the Create contact form without entering values for any of the form fields. Furthermore, you can enter invalid phone numbers and email addresses. We start to tackle the problem of form validation in iteration #3.

Finally, and most importantly, the current iteration of the Contact Manager application cannot be easily modified or maintained. For example, the database access logic is baked right into the controller actions. This means that we cannot modify our data access code without modifying our controllers. In later iterations, we explore software design patterns that we can implement to make the Contact Manager more resilient to change.

NEXT

Iteration #2 – Make the application look nice (C#)

4/10/2020 • 8 minutes to read • Edit Online

by Microsoft

Download Code

> In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.

Building a Contact Management ASP.NET MVC Application (C#)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information - names, phone numbers and email addresses - for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 - Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 - Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 - Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 - Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 - Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 - Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 - Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

The goal of this iteration is to improve the appearance of the Contact Manager application. Currently, the Contact Manager uses the default ASP.NET MVC view master page and cascading style sheet (see Figure 1). These don t look bad, but I don t want the Contact Manager to look just like every other ASP.NET MVC website. I want to replace these files with custom files.

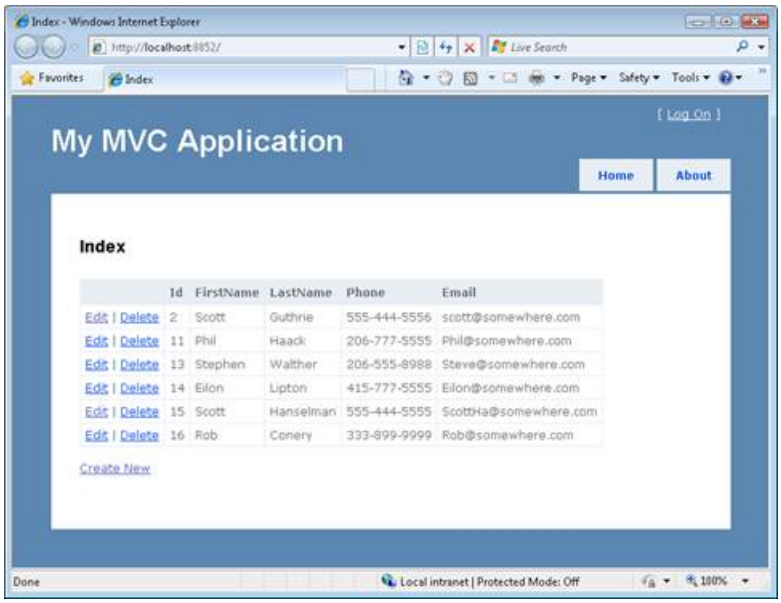


Figure 01: The default appearance of an ASP.NET MVC Application (Click to view full-size image)

In this iteration, I discuss two approaches to improving the visual design of our application. First, I show you how to take advantage of the ASP.NET MVC Design gallery to download a free ASP.NET MVC design template. The ASP.NET MVC Design gallery enables you to create a professional web application without doing any work.

I decided to not use a template from the ASP.NET MVC Design gallery for the Contact Manager application. Instead, I had a custom design created by a professional design firm. In the second part of this tutorial, I explain how I worked with a professional design company to create the final ASP.NET MVC design.

The ASP.NET MVC Design Gallery

The ASP.NET MVC Design Gallery is a free resource provided by Microsoft. The ASP.NET MVC Gallery is located at the following address:

https://www.asp.net/mvc/gallery

The ASP.NET MVC Design Gallery hosts a collection of free website designs that were created specifically for using in an ASP.NET MVC project. Designs are uploaded by members of the community. Visitors to the Gallery can vote for their favorite designs (see Figure 2).

Figure 02: The ASP.NET MVC Design Gallery (Click to view full-size image)

As I write this tutorial, the most popular design in the gallery is a design named October by David Hauser. You can use this design for an ASP.NET MVC project by completing the following steps:

1. Click the **Download** button to download the October.zip file to your computer.
2. Right-click the downloaded October.zip file and click the **Unblock** button (see Figure 3).
3. Unzip the file to a folder named October.
4. Select all of the files from the DesignTemplate folder contained in the October folder, right-click the files, and select the menu option **Copy**.

5. Right-click the ContactManager project node in the Visual Studio Solution Explorer window and select the menu option **Paste** (see Figure 4).
6. Select the Visual Studio menu option **Edit, Find and Replace, Quick Replace** and replace *[MyProjectName]* with *ContactManager* (see Figure 5).

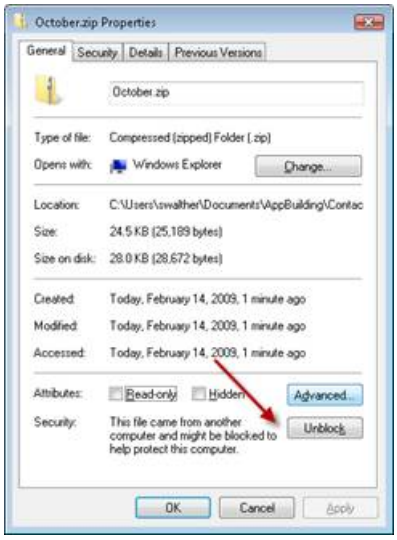


Figure 03: Unblocking a file downloaded from the web (Click to view full-size image)

Figure 04: Overwriting files in the Solution Explorer (Click to view full-size image)

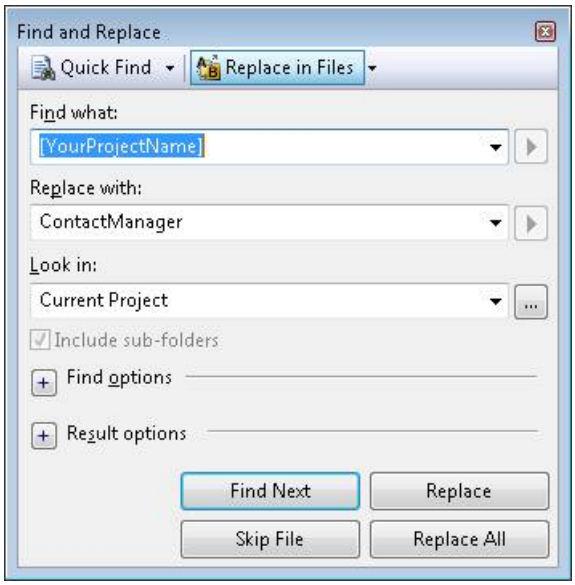


Figure 05: Replacing [ProjectName] with ContactManager (Click to view full-size image)

After you complete these steps, your web application will use the new design. The page in Figure 6 illustrates the appearance of the Contact Manager application with the October design.

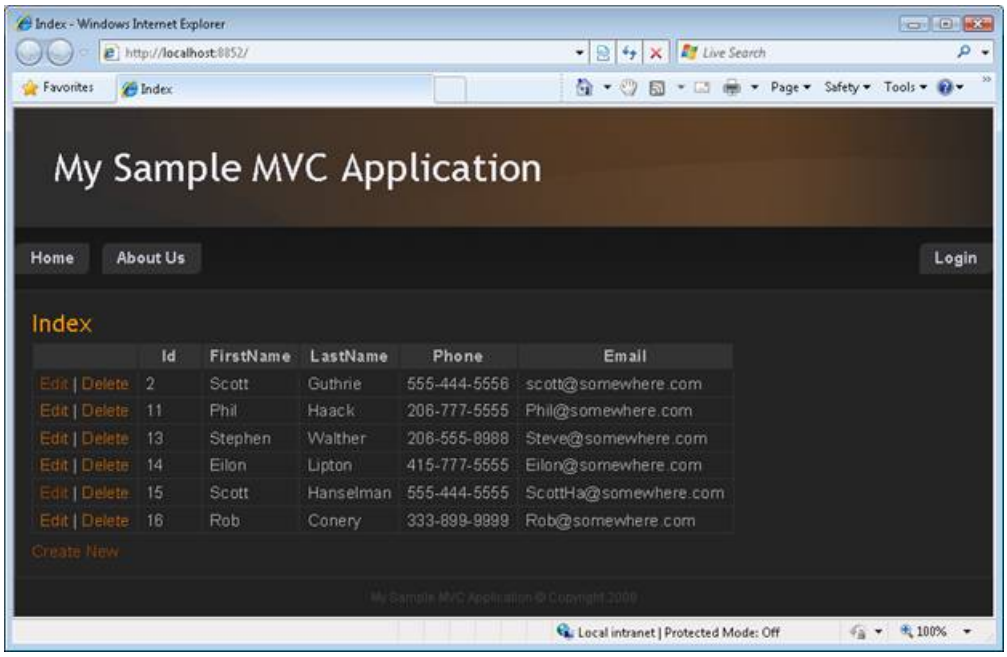


Figure 06: ContactManager with the October template (Click to view full-size image)

Creating a Custom ASP.NET MVC Design

The ASP.NET MVC Design Gallery has a good selection of different design styles. The Gallery provides you with a painless way to customize the appearance of your ASP.NET MVC applications. And, of course, the Gallery has the big advantage of being completely free.

However, you might need to create a completely unique design for your website. In that case, it makes sense to work with a website design company. I decided to take this approach for the design for the Contact Manager application.

I zipped up the Contact Manager from Iteration #1 and sent the project to the design company. They did not own Visual Studio (shame on them!), but that didn't present a problem. They were able to download Microsoft Visual Web Developer for free from the https://www.asp.net website and open the Contact Manager application in Visual Web Developer. In a couple of days, they had produced the design in Figure 7.

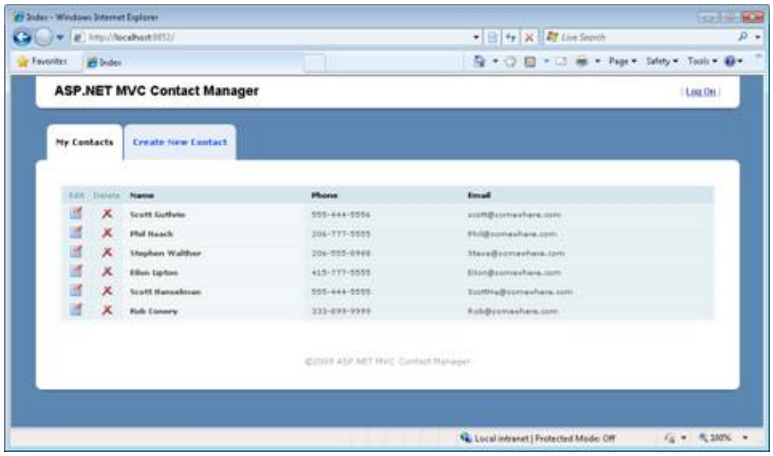


Figure 07: The ASP.NET MVC Contact Manager Design (Click to view full-size image)

The new design consisted of two main files: a new cascading style sheet file and a new view master page file. A view master page contains the layout and shared content for views in an ASP.NET MVC application. For example, the view master page includes the header, navigation tabs, and footer that appear in Figure 7. I overwrote the existing Site.Master view master page in the Views\Shared folder with the new Site.Master file from the design company,

The design company also created a new cascading style sheet and set of images. I placed these new files in the Content folder and overwrote the existing Site.css file. You should place all static content in the Content folder.

Notice that the new design for the Contact Manager includes images for editing and deleting contacts. An Edit and Delete image appear next to each contact in the HTML table of contacts.

Originally, these links that were rendered with the HTML.ActionLink() helper like this:

```
<td>
    <%= Html.ActionLink("Edit", "Edit", new { id=item.Id  }) %> |
    <%= Html.ActionLink("Delete", "Delete", new { id=item.Id  }) %>
</td>
```

The Html.ActionLink() method does not support images (the method HTML encodes the link text for security reasons). Therefore, I replaced the calls to Html.ActionLink() with calls to Url.Action() like this:

```
<td class="actions edit">
    <a href='<%= Url.Action("Edit", new {id=item.Id}) %>'><img src="../../Content/Edit.png" alt="Edit" /></a>
</td>
<td class="actions delete">
    <a href='<%= Url.Action("Delete", new {id=item.Id}) %>'><img src="../../Content/Delete.png" alt="Delete"
/></a>
</td>
```

The Html.ActionLink() method renders an entire HTML hyperlink. The Url.Action() method, on the other hand, renders just the URL without the <a> tag.

Notice, furthermore, that the new design includes both selected and unselected tabs. For example, in Figure 8, the **Create New Contact** tab is selected and the **My Contacts** tab is not selected.

Figure 08: Selected and unselected tabs(Click to view full-size image)

To support rendering both selected and unselected tabs, I created a custom HTML helper named the MenuItemHelper. This helper method renders either a <li> tag or a <li class="selected"> tag depending on whether the current controller and action corresponds to the controller and action name passed to the helper. The code for the MenuItemHelper is contained in Listing 1.

Listing 1 - Helpers\MenuItemHelper.cs

```
using System;
using System.Web.Mvc;
using System.Web.Mvc.Html;

namespace Helpers
{
    /// <summary>
    /// This helper method renders a link within an HTML LI tag.
    /// A class="selected" attribute is added to the tag when
    /// the link being rendered corresponds to the current
    /// controller and action.
    ///
    /// This helper method is used in the Site.Master View
    /// Master Page to display the website menu.
    /// </summary>
    public static class MenuItemHelper
    {
        public static string MenuItem(this HtmlHelper helper, string linkText, string actionName, string
controllerName)
        {
            string currentControllerName = (string)helper.ViewContext.RouteData.Values["controller"];
            string currentActionName = (string)helper.ViewContext.RouteData.Values["action"];

            var builder = new TagBuilder("li");

            // Add selected class
            if (currentControllerName.Equals(controllerName, StringComparison.CurrentCultureIgnoreCase) &&
currentActionName.Equals(actionName, StringComparison.CurrentCultureIgnoreCase))
                builder.AddCssClass("selected");

            // Add link
            builder.InnerHtml = helper.ActionLink(linkText, actionName, controllerName);

            // Render Tag Builder
            return builder.ToString(TagRenderMode.Normal);
        }

    }
}
```

The MenuItemHelper uses the TagBuilder class internally to build the <li> HTML tag. The TagBuilder class is a very useful utility class that you can use whenever you need to build up a new HTML tag. It includes methods for adding attributes, adding CSS classes, generating Ids, and modifying the tag s inner HTML.

Summary

In this iteration, we improved the visual design of our ASP.NET MVC application. First, you were introduced to the ASP.NET MVC Design Gallery. You learned how to download free design templates from the ASP.NET MVC Design Gallery that you can use in your ASP.NET MVC applications.

Next, we discussed how you can create a custom design by modifying the default cascading style sheet file and master view page file. In order to support the new design, we had to make some minor changes to our Contact Manager application. For example, we added a new HTML helper named the MenuItemHelper that displays selected and unselected tabs.

In the next iteration, we tackle the very important subject of validation. We add validation code to our application so that a user cannot create a new contact without supplying required values such as a person s first and last name.

Iteration #3 – Add form validation (C#)

4/10/2020 • 8 minutes to read • Edit Online

by Microsoft

Download Code

> In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.

Building a Contact Management ASP.NET MVC Application (C#)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information - names, phone numbers and email addresses - for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 - Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 - Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 - Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 - Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 - Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 - Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 - Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

In this second iteration of the Contact Manager application, we add basic form validation. We prevent people from submitting a contact without entering values for required form fields. We also validate phone numbers and email addresses (see Figure 1).

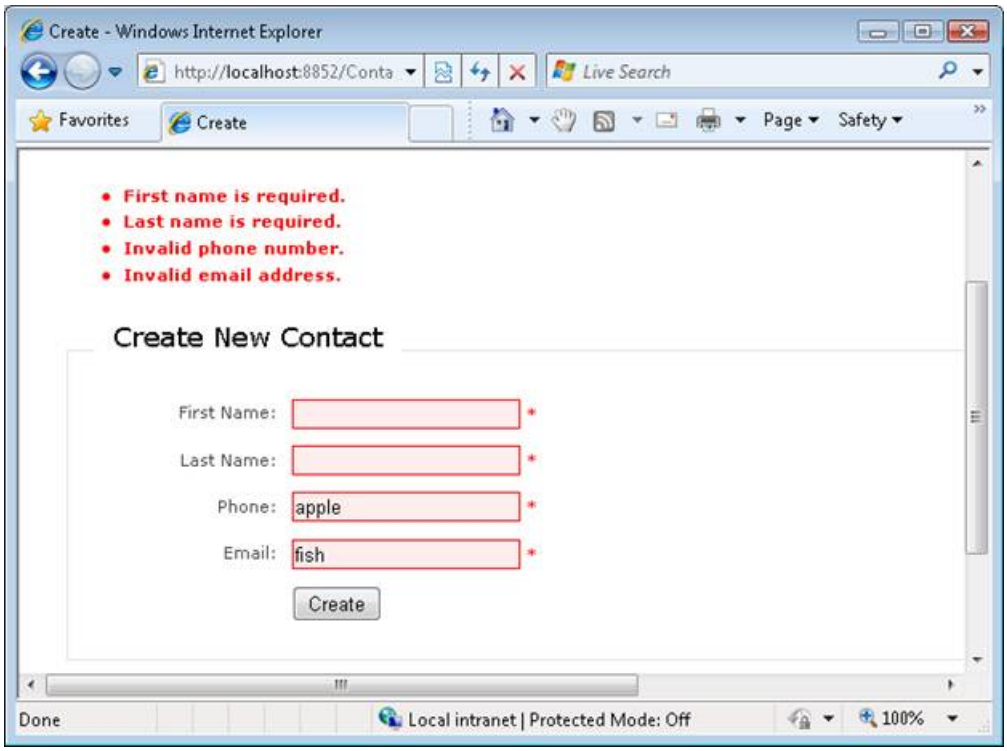


Figure 01: A form with validation (Click to view full-size image)

In this iteration, we add the validation logic directly to the controller actions. In general, this is not the recommended way to add validation to an ASP.NET MVC application. A better approach is to place an application s validation logic in a separate service layer. In the next iteration, we refactor the Contact Manager application to make the application more maintainable.

In this iteration, to keep things simple, we write all of the validation code by hand. Instead of writing the validation code ourselves, we could take advantage of a validation framework. For example, you can use the Microsoft Enterprise Library Validation Application Block (VAB) to implement the validation logic for your ASP.NET MVC application. To learn more about the Validation Application Block, see:

http://msdn.microsoft.com/library/dd203099.aspx

Adding Validation to the Create View

Let s start by adding validation logic to the Create view. Fortunately, because we generated the Create view with Visual Studio, the Create view already contains all of the necessary user interface logic to display validation messages. The Create view is contained in Listing 1.

Listing 1 - \Views\Contact\Create.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.Contact>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Create</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <%= Html.ValidationSummary() %>

    <% using (Html.BeginForm()) {%>

        <fieldset class="fields">
            <legend>Create New Contact</legend>
            <p>
                <label for="FirstName">First Name:</label>
                <%= Html.TextBox("FirstName") %>
                <%= Html.ValidationMessage("FirstName", "*") %>
            </p>
            <p>
                <label for="LastName">Last Name:</label>
                <%= Html.TextBox("LastName") %>
                <%= Html.ValidationMessage("LastName", "*") %>
            </p>
            <p>
                <label for="Phone">Phone:</label>
                <%= Html.TextBox("Phone") %>
                <%= Html.ValidationMessage("Phone", "*") %>
            </p>
            <p>
                <label for="Email">Email:</label>
                <%= Html.TextBox("Email") %>
                <%= Html.ValidationMessage("Email", "*") %>
            </p>
            <p class="submit">
                <input type="submit" value="Create" />
            </p>
        </fieldset>

    <% } %>

</asp:Content>
```

Notice the call to the Html.ValidationSummary() helper method that appears immediately above the HTML form. If there are validation error messages, then this method displays validation messages in a bulleted list.

Notice, furthermore, the calls to Html.ValidationMessage() that appear next to each form field. The ValidationMessage() helper displays an individual validation error message. In the case of Listing 1, an asterisk is displayed when there is a validation error.

Finally, the Html.TextBox() helper automatically renders a Cascading Style Sheet class when there is a validation error associated with the property displayed by the helper. The Html.TextBox() helper renders a class named **input-validation-error**.

When you create a new ASP.NET MVC application, a style sheet named Site.css is created in the Content folder automatically. This style sheet contains the following definitions for CSS classes related to the appearance of validation error messages:

```
.field-validation-error
{
    color: #ff0000;
}

.input-validation-error
{
    border: 1px solid #ff0000;
    background-color: #ffeeee;
}

.validation-summary-errors
{
    font-weight: bold;
    color: #ff0000;
}
```

The field-validation-error class is used to style the output rendered by the Html.ValidationMessage() helper. The input-validation-error class is used to style the textbox (input) rendered by the Html.TextBox() helper. The validation-summary-errors class is used to style the unordered list rendered by the Html.ValidationSummary() helper.

> **NOTE**
> You can modify the style sheet classes described in this section to customize the appearance of validation error messages.

Adding Validation Logic to the Create Action

Right now, the Create view never displays validation error messages because we have not written the logic to generate any messages. In order to display validation error messages, you need to add the error messages to ModelState.

> **NOTE**
> The UpdateModel() method adds error messages to ModelState automatically when there is an error assigning the value of a form field to a property. For example, if you attempt to assign the string "apple" to a BirthDate property that accepts DateTime values, then the UpdateModel() method adds an error to ModelState.

The modified Create() method in Listing 2 contains a new section that validates the properties of the Contact class before the new contact is inserted into the database.

Listing 2 - Controllers\ContactController.cs (Create with validation)

```
//
// POST: /Contact/Create

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create([Bind(Exclude = "Id")] Contact contactToCreate)
{
    // Validation logic
    if (contactToCreate.FirstName.Trim().Length == 0)
        ModelState.AddModelError("FirstName", "First name is required.");
    if (contactToCreate.LastName.Trim().Length == 0)
        ModelState.AddModelError("LastName", "Last name is required.");
    if (contactToCreate.Phone.Length > 0 && !Regex.IsMatch(contactToCreate.Phone, @"((\(\d{3}\) ?)|(\d{3}-))?
\d{3}-\d{4}"))
        ModelState.AddModelError("Phone", "Invalid phone number.");
    if (contactToCreate.Email.Length > 0 && !Regex.IsMatch(contactToCreate.Email, @"^[\w-\.]+@([\w-]+\.)+[\w-]
{2,4}$"))
        ModelState.AddModelError("Email", "Invalid email address.");
    if (!ModelState.IsValid)
        return View();

    // Database logic
    try
    {
        _entities.AddToContactSet(contactToCreate);
        _entities.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

The validate section enforces four distinct validation rules:

- The FirstName property must have a length greater than zero (and it cannot consist solely of spaces)
- The LastName property must have a length greater than zero (and it cannot consist solely of spaces)
- If the Phone property has a value (has a length greater than 0) then the Phone property must match a regular expression.
- If the Email property has a value (has a length greater than 0) then the Email property must match a regular expression.

When there is a validation rule violation, an error message is added to ModelState with the help of the AddModelError() method. When you add a message to ModelState, you provide the name of a property and the text of a validation error message. This error message is displayed in the view by the Html.ValidationSummary() and Html.ValidationMessage() helper methods.

After the validation rules are executed, the IsValid property of ModelState is checked. The IsValid property returns false when any validation error messages have been added to ModelState. If validation fails, the Create form is redisplayed with the error messages.

> **NOTE**
>
> I got the regular expressions for validating the phone number and email address from the regular expression repository at *http://regexlib.com*

Adding Validation Logic to the Edit Action

The Edit() action updates a Contact. The Edit() action needs to perform exactly the same validation as the Create()

action. Instead of duplicating the same validation code, we should refactor the Contact controller so that both the Create() and Edit() actions call the same validation method.

The modified Contact controller class is contained in Listing 3. This class has a new ValidateContact() method that is called within both the Create() and Edit() actions.

Listing 3 - Controllers\ContactController.cs

```
using System.Linq;
using System.Text.RegularExpressions;
using System.Web.Mvc;
using ContactManager.Models;

namespace ContactManager.Controllers
{
    public class ContactController : Controller
    {
        private ContactManagerDBEntities _entities = new ContactManagerDBEntities();

        protected void ValidateContact(Contact contactToValidate)
        {
            if (contactToValidate.FirstName.Trim().Length == 0)
                ModelState.AddModelError("FirstName", "First name is required.");
            if (contactToValidate.LastName.Trim().Length == 0)
                ModelState.AddModelError("LastName", "Last name is required.");
            if (contactToValidate.Phone.Length > 0 && !Regex.IsMatch(contactToValidate.Phone, @"((\(\d{3}\)
?)|(\d{3}-))?\d{3}-\d{4}"))
                ModelState.AddModelError("Phone", "Invalid phone number.");
            if (contactToValidate.Email.Length > 0 && !Regex.IsMatch(contactToValidate.Email, @"^[\w-
\.]+@([\w-]+\.)+[\w-]{2,4}$"))
                ModelState.AddModelError("Email", "Invalid email address.");
        }

        public ActionResult Index()
        {
            return View(_entities.ContactSet.ToList());
        }

        public ActionResult Create()
        {
            return View();
        }

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create([Bind(Exclude = "Id")] Contact contactToCreate)
        {
            // Validation logic
            ValidateContact(contactToCreate);
            if (!ModelState.IsValid)
                return View();

            // Database logic
            try
            {
                _entities.AddToContactSet(contactToCreate);
                _entities.SaveChanges();
                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }

        public ActionResult Edit(int id)
        {
            var contactToEdit = (from c in _entities.ContactSet
```

```
                                where c.Id == id
                                select c).FirstOrDefault();

            return View(contactToEdit);
        }

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Edit(Contact contactToEdit)
        {
            ValidateContact(contactToEdit);
            if (!ModelState.IsValid)
                return View();

            try
            {
                var originalContact = (from c in _entities.ContactSet
                                       where c.Id == contactToEdit.Id
                                       select c).FirstOrDefault();
                _entities.ApplyPropertyChanges(originalContact.EntityKey.EntitySetName, contactToEdit);
                _entities.SaveChanges();
                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }

        public ActionResult Delete(int id)
        {
            var contactToDelete = (from c in _entities.ContactSet
                                   where c.Id == id
                                   select c).FirstOrDefault();

            return View(contactToDelete);
        }

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Delete(Contact contactToDelete)
        {
            try
            {
                var originalContact = (from c in _entities.ContactSet
                                       where c.Id == contactToDelete.Id
                                       select c).FirstOrDefault();

                _entities.DeleteObject(originalContact);
                _entities.SaveChanges();
                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }

    }
}
```

Summary

In this iteration, we added basic form validation to our Contact Manager application. Our validation logic prevents users from submitting a new contact or editing an existing contact without supplying values for the FirstName and LastName properties. Furthermore, users must supply valid phone numbers and email addresses.

In this iteration, we added the validation logic to our Contact Manager application in the easiest way possible.

However, mixing our validation logic into our controller logic will create problems for us in the long term. Our application will be more difficult to maintain and modify over time.

In the next iteration, we will refactor our validation logic and database access logic out of our controllers. We'll take advantage of several software design principles to enable us to create a more loosely coupled, and more maintainable, application.

PREVIOUS NEXT

Iteration #4 – Make the application loosely coupled (C#)

4/10/2020 • 14 minutes to read • Edit Online

by Microsoft

Download Code

> In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.

Building a Contact Management ASP.NET MVC Application (C#)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information - names, phone numbers and email addresses - for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 - Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 - Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 - Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 - Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 - Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 - Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 - Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

In this fourth iteration of the Contact Manager application, we refactor the application to make the application more loosely coupled. When an application is loosely coupled, you can modify the code in one part of the application without needing to modify the code in other parts of the application. Loosely coupled applications are more resilient to change.

Currently, all of the data access and validation logic used by the Contact Manager application is contained in the controller classes. This is a bad idea. Whenever you need to modify one part of your application, you risk introducing bugs into another part of your application. For example, if you modify your validation logic, you risk introducing new bugs into your data access or controller logic.

> **NOTE**
> (SRP), a class should never have more than one reason to change. Mixing controller, validation, and database logic is a massive violation of the Single Responsibility Principle.

There are several reasons that you might need to modify your application. You might need to add a new feature to your application, you might need to fix a bug in your application, or you might need to modify how a feature of your application is implemented. Applications are rarely static. They tend to grow and mutate over time.

Imagine, for example, that you decide to change how you implement your data access layer. Right now, the Contact Manager application uses the Microsoft Entity Framework to access the database. However, you might decide to migrate to a new or alternative data access technology such as ADO.NET Data Services or NHibernate. However, because the data access code is not isolated from the validation and controller code, there is no way to modify the data access code in your application without modifying other code that is not directly related to data access.

When an application is loosely coupled, on the other hand, you can make changes to one part of an application without touching other parts of an application. For example, you can switch data access technologies without modifying your validation or controller logic.

In this iteration, we take advantage of several software design patterns that enable us to refactor our Contact Manager application into a more loosely coupled application. When we are done, the Contact Manager won t do anything that it didn't do before. However, we'll be able to change the application more easily in the future.

> **NOTE**
> Refactoring is the process of rewriting an application in such a way that it does not lose any existing functionality.

Using the Repository Software Design Pattern

Our first change is to take advantage of a software design pattern called the Repository pattern. We'll use the Repository pattern to isolate our data access code from the rest of our application.

Implementing the Repository pattern requires us to complete the following two steps:

1. Create an interface
2. Create a concrete class that implements the interface

First, we need to create an interface that describes all of the data access methods that we need to perform. The IContactManagerRepository interface is contained in Listing 1. This interface describes five methods: CreateContact(), DeleteContact(), EditContact(), GetContact, and ListContacts().

Listing 1 - Models\IContactManagerRepository.cs

```
using System;
using System.Collections.Generic;

namespace ContactManager.Models
{
    public interface IContactRepository
    {
        Contact CreateContact(Contact contactToCreate);
        void DeleteContact(Contact contactToDelete);
        Contact EditContact(Contact contactToUpdate);
        Contact GetContact(int id);
        IEnumerable<Contact> ListContacts();

    }
}
```

Next, we need to create a concrete class that implements the IContactManagerRepository interface. Because we are using the Microsoft Entity Framework to access the database, we'll create a new class named EntityContactManagerRepository. This class is contained in Listing 2.

Listing 2 - Models\EntityContactManagerRepository.cs

```
using System.Collections.Generic;
using System.Linq;

namespace ContactManager.Models
{
    public class EntityContactManagerRepository : ContactManager.Models.IContactManagerRepository
    {
        private ContactManagerDBEntities _entities = new ContactManagerDBEntities();

        public Contact GetContact(int id)
        {
            return (from c in _entities.ContactSet
                    where c.Id == id
                    select c).FirstOrDefault();
        }

        public IEnumerable ListContacts()
        {
            return _entities.ContactSet.ToList();
        }

        public Contact CreateContact(Contact contactToCreate)
        {
            _entities.AddToContactSet(contactToCreate);
            _entities.SaveChanges();
            return contactToCreate;
        }

        public Contact EditContact(Contact contactToEdit)
        {
            var originalContact = GetContact(contactToEdit.Id);
            _entities.ApplyPropertyChanges(originalContact.EntityKey.EntitySetName, contactToEdit);
            _entities.SaveChanges();
            return contactToEdit;
        }

        public void DeleteContact(Contact contactToDelete)
        {
            var originalContact = GetContact(contactToDelete.Id);
            _entities.DeleteObject(originalContact);
            _entities.SaveChanges();
        }

    }
}
```

Notice that the EntityContactManagerRepository class implements the IContactManagerRepository interface. The class implements all five of the methods described by that interface.

You might wonder why we need to bother with an interface. Why do we need to create both an interface and a class that implements it?

With one exception, the remainder of our application will interact with the interface and not the concrete class. Instead of calling the methods exposed by the EntityContactManagerRepository class, we'll call the methods exposed by the IContactManagerRepository interface.

That way, we can implement the interface with a new class without needing to modify the remainder of our application. For example, at some future date, we might want to implement an DataServicesContactManagerRepository class that implements the IContactManagerRepository interface. The DataServicesContactManagerRepository class might use ADO.NET Data Services to access a database instead of the Microsoft Entity Framework.

If our application code is programmed against the IContactManagerRepository interface instead of the concrete

EntityContactManagerRepository class then we can switch concrete classes without modifying any of the rest of our code. For example, we can switch from the EntityContactManagerRepository class to the DataServicesContactManagerRepository class without modifying our data access or validation logic.

Programming against interfaces (abstractions) instead of concrete classes makes our application more resilient to change.

> **NOTE**
> You can quickly create an interface from a concrete class within Visual Studio by selecting the menu option Refactor, Extract Interface. For example, you can create the EntityContactManagerRepository class first and then use Extract Interface to generate the IContactManagerRepository interface automatically.

Using the Dependency Injection Software Design Pattern

Now that we have migrated our data access code to a separate Repository class, we need to modify our Contact controller to use this class. We will take advantage of a software design pattern called Dependency Injection to use the Repository class in our controller.

The modified Contact controller is contained in Listing 3.

Listing 3 - Controllers\ContactController.cs

```
using System.Text.RegularExpressions;
using System.Web.Mvc;
using ContactManager.Models;

namespace ContactManager.Controllers
{
    public class ContactController : Controller
    {
        private IContactManagerRepository _repository;

        public ContactController()
            : this(new EntityContactManagerRepository())
        {}

        public ContactController(IContactManagerRepository repository)
        {
            _repository = repository;
        }

        protected void ValidateContact(Contact contactToValidate)
        {
            if (contactToValidate.FirstName.Trim().Length == 0)
                ModelState.AddModelError("FirstName", "First name is required.");
            if (contactToValidate.LastName.Trim().Length == 0)
                ModelState.AddModelError("LastName", "Last name is required.");
            if (contactToValidate.Phone.Length > 0 && !Regex.IsMatch(contactToValidate.Phone, @"((\(\d{3}\)
?)|(\d{3}-))?\d{3}-\d{4}"))
                ModelState.AddModelError("Phone", "Invalid phone number.");
            if (contactToValidate.Email.Length > 0 && !Regex.IsMatch(contactToValidate.Email, @"^[\w-
\.]+@([\w-]+\.)+[\w-]{2,4}$"))
                ModelState.AddModelError("Email", "Invalid email address.");
        }

        public ActionResult Index()
        {
            return View(_repository.ListContacts());
        }

        public ActionResult Create()
        {
```

```
            return View();
        }

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create([Bind(Exclude = "Id")] Contact contactToCreate)
        {
            // Validation logic
            ValidateContact(contactToCreate);
            if (!ModelState.IsValid)
                return View();

            // Database logic
            try
            {
                _repository.CreateContact(contactToCreate);
                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }

        public ActionResult Edit(int id)
        {
            return View(_repository.GetContact(id));
        }

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Edit(Contact contactToEdit)
        {
            // Validation logic
            ValidateContact(contactToEdit);
            if (!ModelState.IsValid)
                return View();

            // Database logic
            try
            {
                _repository.EditContact(contactToEdit);
                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }

        public ActionResult Delete(int id)
        {
            return View(_repository.GetContact(id));
        }

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Delete(Contact contactToDelete)
        {
            try
            {
                _repository.DeleteContact(contactToDelete);
                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }

    }
}
```

```
}
```

Notice that the Contact controller in Listing 3 has two constructors. The first constructor passes a concrete instance of the IContactManagerRepository interface to the second constructor. The Contact controller class uses *Constructor Dependency Injection*.

The one and only place that the EntityContactManagerRepository class is used is in the first constructor. The remainder of the class uses the IContactManagerRepository interface instead of the concrete EntityContactManagerRepository class.

This makes it easy to switch implementations of the IContactManagerRepository class in the future. If you want to use the DataServicesContactRepository class instead of the EntityContactManagerRepository class, just modify the first constructor.

Constructor Dependency injection also makes the Contact controller class very testable. In your unit tests, you can instantiate the Contact controller by passing a mock implementation of the IContactManagerRepository class. This feature of Dependency Injection will be very important to us in the next iteration when we build unit tests for the Contact Manager application.

> **NOTE**
> If you want to completely decouple the Contact controller class from a particular implementation of the IContactManagerRepository interface then you can take advantage of a framework that supports Dependency Injection such as StructureMap or the Microsoft Entity Framework (MEF). By taking advantage of a Dependency Injection framework, you never need to refer to a concrete class in your code.

Creating a Service Layer

You might have noticed that our validation logic is still mixed up with our controller logic in the modified controller class in Listing 3. For the same reason that it is a good idea to isolate our data access logic, it is a good idea to isolate our validation logic.

To fix this problem, we can create a separate *service layer*. The service layer is a separate layer that we can insert between our controller and repository classes. The service layer contains our business logic including all of our validation logic.

The ContactManagerService is contained in Listing 4. It contains the validation logic from the Contact controller class.

Listing 4 - Models\ContactManagerService.cs

```
using System.Collections.Generic;
using System.Text.RegularExpressions;
using System.Web.Mvc;
using ContactManager.Models.Validation;

namespace ContactManager.Models
{
    public class ContactManagerService : IContactManagerService
    {
        private IValidationDictionary _validationDictionary;
        private IContactManagerRepository _repository;

        public ContactManagerService(IValidationDictionary validationDictionary)
            : this(validationDictionary, new EntityContactManagerRepository())
        {}

        public ContactManagerService(IValidationDictionary validationDictionary, IContactManagerRepository
repository)
        {
```

```
        {
            _validationDictionary = validationDictionary;
            _repository = repository;
        }

        public bool ValidateContact(Contact contactToValidate)
        {
            if (contactToValidate.FirstName.Trim().Length == 0)
                _validationDictionary.AddError("FirstName", "First name is required.");
            if (contactToValidate.LastName.Trim().Length == 0)
                _validationDictionary.AddError("LastName", "Last name is required.");
            if (contactToValidate.Phone.Length > 0 && !Regex.IsMatch(contactToValidate.Phone, @"((\(\d{3}\)
?)|(\d{3}-))?\d{3}-\d{4}"))
                _validationDictionary.AddError("Phone", "Invalid phone number.");
            if (contactToValidate.Email.Length > 0 && !Regex.IsMatch(contactToValidate.Email, @"^[\w-
\.]+@([\w-]+\.)+[\w-]{2,4}$"))
                _validationDictionary.AddError("Email", "Invalid email address.");
            return _validationDictionary.IsValid;
        }

        #region IContactManagerService Members

        public bool CreateContact(Contact contactToCreate)
        {
            // Validation logic
            if (!ValidateContact(contactToCreate))
                return false;

            // Database logic
            try
            {
                _repository.CreateContact(contactToCreate);
            }
            catch
            {
                return false;
            }
            return true;
        }

        public bool EditContact(Contact contactToEdit)
        {
            // Validation logic
            if (!ValidateContact(contactToEdit))
                return false;

            // Database logic
            try
            {
                _repository.EditContact(contactToEdit);
            }
            catch
            {
                return false;
            }
            return true;
        }

        public bool DeleteContact(Contact contactToDelete)
        {
            try
            {
                _repository.DeleteContact(contactToDelete);
            }
            catch
            {
                return false;
            }
            return true;
```

```
        }

        public Contact GetContact(int id)
        {
            return _repository.GetContact(id);
        }

        public IEnumerable<Contact> ListContacts()
        {
            return _repository.ListContacts();
        }

        #endregion
    }
}
```

Notice that the constructor for the ContactManagerService requires a ValidationDictionary. The service layer communicates with the controller layer through this ValidationDictionary. We discuss the ValidationDictionary in detail in the following section when we discuss the Decorator pattern.

Notice, furthermore, that the ContactManagerService implements the IContactManagerService interface. You should always strive to program against interfaces instead of concrete classes. Other classes in the Contact Manager application do not interact with the ContactManagerService class directly. Instead, with one exception, the remainder of the Contact Manager application is programmed against the IContactManagerService interface.

The IContactManagerService interface is contained in Listing 5.

Listing 5 - Models\IContactManagerService.cs

```
using System.Collections.Generic;

namespace ContactManager.Models
{
    public interface IContactManagerService
    {
        bool CreateContact(Contact contactToCreate);
        bool DeleteContact(Contact contactToDelete);
        bool EditContact(Contact contactToEdit);
        Contact GetContact(int id);
        IEnumerable ListContacts();
    }
}
```

The modified Contact controller class is contained in Listing 6. Notice that the Contact controller no longer interacts with the ContactManager repository. Instead, the Contact controller interacts with the ContactManager service. Each layer is isolated as much as possible from other layers.

Listing 6 - Controllers\ContactController.cs

```
using System.Web.Mvc;
using ContactManager.Models;

namespace ContactManager.Controllers
{
    public class ContactController : Controller
    {
        private IContactManagerService _service;

        public ContactController()
        {
            _service = new ContactManagerService(new ModelStateWrapper(this.ModelState));

        }

        public ContactController(IContactManagerService service)
        {
            _service = service;
        }

        public ActionResult Index()
        {
            return View(_service.ListContacts());
        }

        public ActionResult Create()
        {
            return View();
        }

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create([Bind(Exclude = "Id")] Contact contactToCreate)
        {
            if (_service.CreateContact(contactToCreate))
                return RedirectToAction("Index");
            return View();
        }

        public ActionResult Edit(int id)
        {
            return View(_service.GetContact(id));
        }

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Edit(Contact contactToEdit)
        {
            if (_service.EditContact(contactToEdit))
                return RedirectToAction("Index");
            return View();
        }

        public ActionResult Delete(int id)
        {
            return View(_service.GetContact(id));
        }

        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Delete(Contact contactToDelete)
        {
            if (_service.DeleteContact(contactToDelete))
                return RedirectToAction("Index");
            return View();
        }

    }
}
```

Our application no longer runs afoul of the Single Responsibility Principle (SRP). The Contact controller in Listing 6 has been stripped of every responsibility other than controlling the flow of application execution. All the validation logic has been removed from the Contact controller and pushed into the service layer. All of the database logic has been pushed into the repository layer.

Using the Decorator Pattern

We want to be able to completely decouple our service layer from our controller layer. In principle, we should be able to compile our service layer in a separate assembly from our controller layer without needing to add a reference to our MVC application.

However, our service layer needs to be able to pass validation error messages back to the controller layer. How can we enable the service layer to communicate validation error messages without coupling the controller and service layer? We can take advantage of a software design pattern named the Decorator pattern.

A controller uses a ModelStateDictionary named ModelState to represent validation errors. Therefore, you might be tempted to pass ModelState from the controller layer to the service layer. However, using ModelState in the service layer would make your service layer dependent on a feature of the ASP.NET MVC framework. This would be bad because, someday, you might want to use the service layer with a WPF application instead of an ASP.NET MVC application. In that case, you wouldn't want to reference the ASP.NET MVC framework to use the ModelStateDictionary class.

The Decorator pattern enables you to wrap an existing class in a new class in order to implement an interface. Our Contact Manager project includes the ModelStateWrapper class contained in Listing 7. The ModelStateWrapper class implements the interface in Listing 8.

Listing 7 - Models\Validation\ModelStateWrapper.cs

```
using System.Web.Mvc;

namespace ContactManager.Models.Validation
{
    public class ModelStateWrapper : IValidationDictionary
    {
        private ModelStateDictionary _modelState;

        public ModelStateWrapper(ModelStateDictionary modelState)
        {
            _modelState = modelState;
        }

        public void AddError(string key, string errorMessage)
        {
            _modelState.AddModelError(key, errorMessage);
        }

        public bool IsValid
        {
            get { return _modelState.IsValid; }
        }
    }
}
```

Listing 8 - Models\Validation\IValidationDictionary.cs

```
namespace ContactManager.Models.Validation
{
    public interface IValidationDictionary
    {
        void AddError(string key, string errorMessage);
        bool IsValid {get;}
    }
}
```

If you take a close look at Listing 5 then you'll see that the ContactManager service layer uses the IValidationDictionary interface exclusively. The ContactManager service is not dependent on the ModelStateDictionary class. When the Contact controller creates the ContactManager service, the controller wraps its ModelState like this:

```
_service = new ContactManagerService(new ModelStateWrapper(this.ModelState));
```

Summary

In this iteration, we did not add any new functionality to the Contact Manager application. The goal of this iteration was to refactor the Contact Manager application so that is easier to maintain and modify.

First, we implemented the Repository software design pattern. We migrated all of the data access code to a separate ContactManager repository class.

We also isolated our validation logic from our controller logic. We created a separate service layer that contains all of our validation code. The controller layer interacts with the service layer, and the service layer interacts with the repository layer.

When we created the service layer, we took advantage of the Decorator pattern to isolate ModelState from our service layer. In our service layer, we programmed against the IValidationDictionary interface instead of ModelState.

Finally, we took advantage of a software design pattern named the Dependency Injection pattern. This pattern enables us to program against interfaces (abstractions) instead of concrete classes. Implementing the Dependency Injection design pattern also makes our code more testable. In the next iteration, we add unit tests to our project.

PREVIOUS NEXT

Iteration #5 – Create unit tests (C#)

4/10/2020 • 15 minutes to read • Edit Online

by Microsoft

Download Code

> In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.

Building a Contact Management ASP.NET MVC Application (C#)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information - names, phone numbers and email addresses - for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 - Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 - Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 - Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 - Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 - Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 - Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 - Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

In the previous iteration of the Contact Manager application, we refactored the application to be more loosely coupled. We separated the application into distinct controller, service, and repository layers. Each layer interacts with the layer beneath it through interfaces.

We refactored the application to make the application easier to maintain and modify. For example, if we need to use a new data access technology, we can simply change the repository layer without touching the controller or service layer. By making the Contact Manager loosely coupled, we ve made the application more resilient to

change.

But, what happens when we need to add a new feature to the Contact Manager application? Or, what happens when we fix a bug? A sad, but well proven, truth of writing code is that whenever you touch code you create the risk of introducing new bugs.

For example, one fine day, your manager might ask you to add a new feature to the Contact Manager. She wants you to add support for Contact Groups. She wants you to enable users to organize their contacts into groups such as Friends, Business, and so on.

In order to implement this new feature, you'll need to modify all three layers of the Contact Manager application. You'll need to add new functionality to the controllers, the service layer, and the repository. As soon as you start modifying code, you risk breaking functionality that worked before.

Refactoring our application into separate layers, as we did in the previous iteration, was a good thing. It was a good thing because it enables us to make changes to entire layers without touching the rest of the application. However, if you want to make the code within a layer easier to maintain and modify, you need to create unit tests for the code.

You use a unit test to test an individual unit of code. These units of code are smaller than entire application layers. Typically, you use a unit test to verify whether a particular method in your code behaves in the way that you expect. For example, you would create a unit test for the CreateContact() method exposed by the ContactManagerService class.

The unit tests for an application work just like a safety net. Whenever you modify code in an application, you can run a set of unit tests to check whether the modification breaks existing functionality. Unit tests make your code safe to modify. Unit tests make all of the code in your application more resilient to change.

In this iteration, we add unit tests to our Contact Manager application. That way, in the next iteration, we can add Contact Groups to our application without worrying about breaking existing functionality.

> **NOTE**
> There are a variety of unit testing frameworks including NUnit, xUnit.net, and MbUnit. In this tutorial, we use the unit testing framework included with Visual Studio. However, you could just as easily use one of these alternative frameworks.

What Gets Tested

In the perfect world, all of your code would be covered by unit tests. In the perfect world, you would have the perfect safety net. You would be able to modify any line of code in your application and know instantly, by executing your unit tests, whether the change broke existing functionality.

However, we don t live in a perfect world. In practice, when writing unit tests, you concentrate on writing tests for your business logic (for example, validation logic). In particular, you *do not* write unit tests for your data access logic or your view logic.

To be useful, unit tests must execute very quickly. You easily can accumulate hundreds (or even thousands) of unit tests for an application. If the unit tests take a long time to run then you'll avoid executing them. In other words, long running unit tests are useless for day to day coding purposes.

For this reason, you typically do not write unit tests for code that interacts with a database. Running hundreds of unit tests against a live database would be too slow. Instead, you mock your database and write code that interacts with the mock database (we discuss mocking a database below).

For a similar reason, you typically do not write unit tests for views. In order to test a view, you must spin up a web server. Because spinning up a web server is a relatively slow process, creating unit tests for your views is not recommended.

If your view contains complicated logic then you should consider moving the logic into Helper methods. You can write unit tests for Helper methods that execute without spinning up a web server.

> **NOTE**
> While writing tests for data access logic or view logic is not a good idea when writing unit tests, these tests can be very valuable when building functional or integration tests.

> **NOTE**
> ASP.NET MVC is the Web Forms View Engine. While the Web Forms View Engine is dependent on a web server, other view engines might not be.

Using a Mock Object Framework

When building unit tests, you almost always need to take advantage of a Mock Object framework. A Mock Object framework enables you to create mocks and stubs for the classes in your application.

For example, you can use a Mock Object framework to generate a mock version of your repository class. That way, you can use the mock repository class instead of the real repository class in your unit tests. Using the mock repository enables you to avoid executing database code when executing a unit test.

Visual Studio does not include a Mock Object framework. However, there are several commercial and open source Mock Object frameworks available for the .NET framework:

1. Moq - This framework is available under the open source BSD license. You can download Moq from https://code.google.com/p/moq/.
2. Rhino Mocks - This framework is available under the open source BSD license. You can download Rhino Mocks from http://ayende.com/projects/rhino-mocks.aspx.
3. Typemock Isolator - This is a commercial framework. You can download a trial version from http://www.typemock.com/.

In this tutorial, I decided to use Moq. However, you could just as easily use Rhino Mocks or Typemock Isolator to create the Mock objects for the Contact Manager application.

Before you can use Moq, you need to complete the following steps:

1. .
2. Before you unzip the download, make sure that you right-click the file and click the button labeled **Unblock** (see Figure 1).
3. Unzip the download.
4. Add a reference to the Moq assembly by right-clicking the References folder in the ContactManager.Tests project and selecting **Add Reference**. Under the Browse tab, browse to the folder where you unzipped Moq and select the Moq.dll assembly. Click the **OK** button.
5. After you complete these steps, your References folder should look like Figure 2.

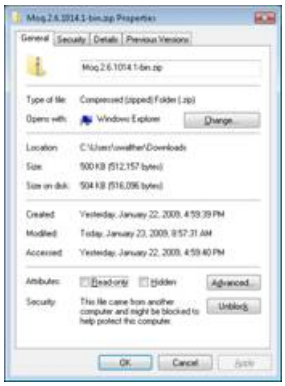


Figure 01: Unblocking Moq(Click to view full-size image)

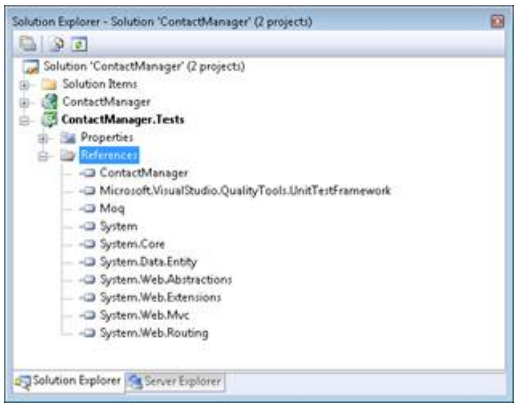


Figure 02: References after adding Moq(Click to view full-size image)

Creating Unit Tests for the Service Layer

Let s start by creating a set of unit tests for our Contact Manager application s service layer. We'll use these tests to verify our validation logic.

Create a new folder named Models in the ContactManager.Tests project. Next, right-click the Models folder and select **Add, New Test**. The **Add New Test** dialog shown in Figure 3 appears. Select the **Unit Test** template and name your new test ContactManagerServiceTest.cs. Click the **OK** button to add your new test to your Test Project.

> **NOTE**
>
> In general, you want the folder structure of your Test Project to match the folder structure of your ASP.NET MVC project. For example, you place controller tests in a Controllers folder, model tests in a Models folder, and so on.

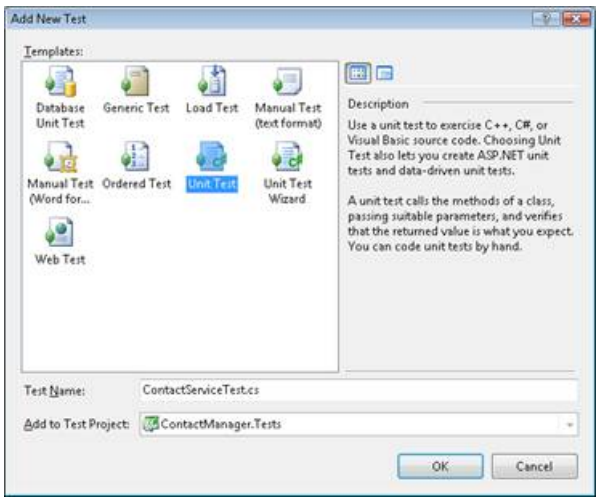


Figure 03: Models\ContactManagerServiceTest.cs(Click to view full-size image)

Initially, we want to test the CreateContact() method exposed by the ContactManagerService class. We'll create the following five tests:

- CreateContact() - Tests that CreateContact() returns the value true when a valid Contact is passed to the method.
- CreateContactRequiredFirstName() - Tests that an error message is added to model state when a Contact with a missing first name is passed to the CreateContact() method.
- CreateContactRequiredLastName() - Tests that an error message is added to model state when a Contact with a missing last name is passed to the CreateContact() method.
- CreateContactInvalidPhone() - Tests that an error message is added to model state when a Contact with an invalid phone number is passed to the CreateContact() method.
- CreateContactInvalidEmail() - Tests that an error message is added to model state when a Contact with an invalid email address is passed to the CreateContact() method..

The first test verifies that a valid Contact does not generate a validation error. The remaining tests check each of the validation rules.

The code for these tests is contained in Listing 1.

Listing 1 - Models\ContactManagerServiceTest.cs

```
using System.Web.Mvc;
using ContactManager.Models;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;

namespace ContactManager.Tests.Models
{
    [TestClass]
    public class ContactManagerServiceTest
    {
        private Mock<IContactManagerRepository> _mockRepository;
        private ModelStateDictionary _modelState;
        private IContactManagerService _service;

        [TestInitialize]
        public void Initialize()
        {
            _mockRepository = new Mock<IContactManagerRepository>();
            _modelState = new ModelStateDictionary();
            _service = new ContactManagerService(new ModelStateWrapper(_modelState), _mockRepository.Object);
        }

        [TestMethod]
        public void CreateContact()
        {
            // Arrange
            var contact = Contact.CreateContact(-1, "Stephen", "Walther", "555-5555", "steve@somewhere.com");

            // Act
            var result = _service.CreateContact(contact);

            // Assert
            Assert.IsTrue(result);
        }

        [TestMethod]
        public void CreateContactRequiredFirstName()
        {
            // Arrange
            var contact = Contact.CreateContact(-1, string.Empty, "Walther", "555-5555",
"steve@somewhere.com");
```

```
            // Act
            var result = _service.CreateContact(contact);

            // Assert
            Assert.IsFalse(result);
            var error = _modelState["FirstName"].Errors[0];
            Assert.AreEqual("First name is required.", error.ErrorMessage);
        }

        [TestMethod]
        public void CreateContactRequiredLastName()
        {
            // Arrange
            var contact = Contact.CreateContact(-1, "Stephen", string.Empty, "555-5555",
"steve@somewhere.com");

            // Act
            var result = _service.CreateContact(contact);

            // Assert
            Assert.IsFalse(result);
            var error = _modelState["LastName"].Errors[0];
            Assert.AreEqual("Last name is required.", error.ErrorMessage);
        }

        [TestMethod]
        public void CreateContactInvalidPhone()
        {
            // Arrange
            var contact = Contact.CreateContact(-1, "Stephen", "Walther", "apple", "steve@somewhere.com");

            // Act
            var result = _service.CreateContact(contact);

            // Assert
            Assert.IsFalse(result);
            var error = _modelState["Phone"].Errors[0];
            Assert.AreEqual("Invalid phone number.", error.ErrorMessage);
        }

        [TestMethod]
        public void CreateContactInvalidEmail()
        {
            // Arrange
            var contact = Contact.CreateContact(-1, "Stephen", "Walther", "555-5555", "apple");

            // Act
            var result = _service.CreateContact(contact);

            // Assert
            Assert.IsFalse(result);
            var error = _modelState["Email"].Errors[0];
            Assert.AreEqual("Invalid email address.", error.ErrorMessage);
        }
    }
}
```

Because we use the Contact class in Listing 1, we need to add a reference to the Microsoft Entity Framework to our Test project. Add a reference to the System.Data.Entity assembly.

Listing 1 contains a method named Initialize() that is decorated with the [TestInitialize] attribute. This method is called automatically before each of the unit tests is run (it is called 5 times right before each of the unit tests). The Initialize() method creates a mock repository with the following line of code:

```
_mockRepository = new Mock<IContactManagerRepository>();
```

This line of code uses the Moq framework to generate a mock repository from the IContactManagerRepository interface. The mock repository is used instead of the actual EntityContactManagerRepository to avoid accessing the database when each unit test is run. The mock repository implements the methods of the IContactManagerRepository interface, but the methods don t actually do anything.

> **NOTE**
>
> When using the Moq framework, there is a distinction between _mockRepository and _mockRepository.Object. The former refers to the Mock<IContactManagerRepository> class that contains methods for specifying how the mock repository will behave. The latter refers to the actual mock repository that implements the IContactManagerRepository interface.

The mock repository is used in the Initialize() method when creating an instance of the ContactManagerService class. All of the individual unit tests use this instance of the ContactManagerService class.

Listing 1 contains five methods that correspond to each of the unit tests. Each of these methods is decorated with the [TestMethod] attribute. When you run the unit tests, any method that has this attribute is called. In other words, any method that is decorated with the [TestMethod] attribute is a unit test.

The first unit test, named CreateContact(), verifies that calling CreateContact() returns the value true when a valid instance of the Contact class is passed to the method. The test creates an instance of the Contact class, calls the CreateContact() method, and verifies that CreateContact() returns the value true.

The remaining tests verify that when the CreateContact() method is called with an invalid Contact then the method returns false and the expected validation error message is added to model state. For example, the CreateContactRequiredFirstName() test creates an instance of the Contact class with an empty string for its FirstName property. Next, the CreateContact() method is called with the invalid Contact. Finally, the test verifies that CreateContact() returns false and that model state contains the expected validation error message "First name is required."

You can run the unit tests in Listing 1 by selecting the menu option **Test, Run, All Tests in Solution (CTRL+R, A)**. The results of the tests are displayed in the Test Results window (see Figure 4).

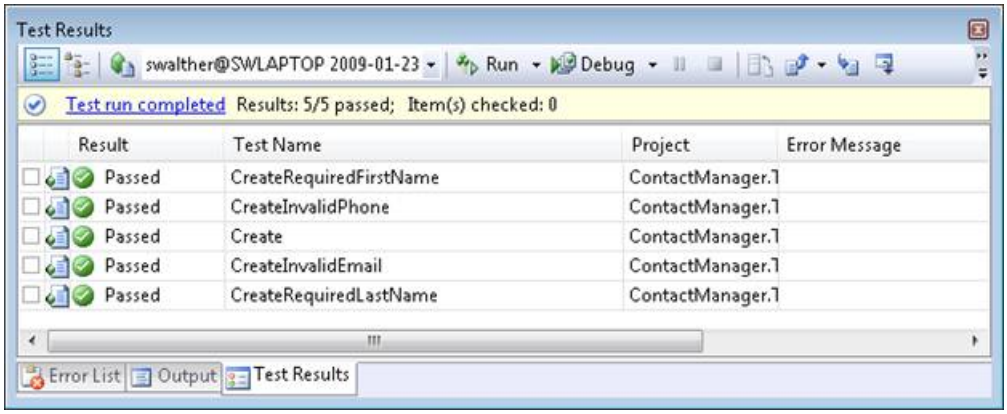


Figure 04: Test Results (Click to view full-size image)

Creating Unit Tests for Controllers

ASP.NETMVC application control the flow of user interaction. When testing a controller, you want to test whether the controller returns the right action result and view data. You also might want to test whether a controller interacts with model classes in the manner expected.

For example, Listing 2 contains two unit tests for the Contact controller Create() method. The first unit test verifies that when a valid Contact is passed to the Create() method then the Create() method redirects to the Index action. In other words, when passed a valid Contact, the Create() method should return a RedirectToRouteResult that represents the Index action.

We don t want to test the ContactManager service layer when we are testing the controller layer. Therefore, we mock the service layer with the following code in the Initialize method:

```
_service = new Mock();
```

In the CreateValidContact() unit test, we mock the behavior of calling the service layer CreateContact() method with the following line of code:

```
_service.Expect(s => s.CreateContact(contact)).Returns(true);
```

This line of code causes the mock ContactManager service to return the value true when its CreateContact() method is called. By mocking the service layer, we can test the behavior of our controller without needing to execute any code in the service layer.

The second unit test verifies that the Create() action returns the Create view when an invalid contact is passed to the method. We cause the service layer CreateContact() method to return the value false with the following line of code:

```
_service.Expect(s => s.CreateContact(contact)).Returns(false);
```

If the Create() method behaves as we expect then it should return the Create view when the service layer returns the value false. That way, the controller can display the validation error messages in the Create view and the user has a chance to correct that invalid Contact properties.

If you plan to build unit tests for your controllers then you need to return explicit view names from your controller actions. For example, do not return a view like this:

return View();

Instead, return the view like this:

return View("Create");

If you are not explicit when returning a view then the ViewResult.ViewName property returns an empty string.

Listing 2 - Controllers\ContactControllerTest.cs

```
using System.Web.Mvc;
using ContactManager.Controllers;
using ContactManager.Models;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;

namespace ContactManager.Tests.Controllers
{
    [TestClass]
    public class ContactControllerTest
    {
        private Mock<IContactManagerService> _service;

        [TestInitialize]
        public void Initialize()
        {
            _service = new Mock<IContactManagerService>();
        }

        [TestMethod]
        public void CreateValidContact()
        {
            // Arrange
            var contact = new Contact();
            _service.Expect(s => s.CreateContact(contact)).Returns(true);
            var controller = new ContactController(_service.Object);

            // Act
            var result = (RedirectToRouteResult)controller.Create(contact);

            // Assert
            Assert.AreEqual("Index", result.RouteValues["action"]);
        }

        [TestMethod]
        public void CreateInvalidContact()
        {
            // Arrange
            var contact = new Contact();
            _service.Expect(s => s.CreateContact(contact)).Returns(false);
            var controller = new ContactController(_service.Object);

            // Act
            var result = (ViewResult)controller.Create(contact);

            // Assert
            Assert.AreEqual("Create", result.ViewName);
        }

    }
}
```

Summary

In this iteration, we created unit tests for our Contact Manager application. We can run these unit tests at any time to verify that our application still behaves in the manner that we expect. The unit tests act as a safety net for our application enabling us to safely modify our application in the future.

We created two sets of unit tests. First, we tested our validation logic by creating unit tests for our service layer. Next, we tested our flow control logic by creating unit tests for our controller layer. When testing our service layer, we isolated our tests for our service layer from our repository layer by mocking our repository layer. When testing the controller layer, we isolated our tests for our controller layer by mocking the service layer.

In the next iteration, we modify the Contact Manager application so that it supports Contact Groups. We'll add this

new functionality to our application using a software design process called test-driven development.

PREVIOUS NEXT

Iteration #6 – Use test-driven development (C#)

4/10/2020 • 20 minutes to read • Edit Online

by Microsoft

Download Code

> In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.

Building a Contact Management ASP.NET MVC Application (C#)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information - names, phone numbers and email addresses - for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 - Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 - Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 - Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 - Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 - Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 - Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 - Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

In the previous iteration of the Contact Manager application, we created unit tests to provide a safety net for our code. The motivation for creating the unit tests was to make our code more resilient to change. With unit tests in place, we can happily make any change to our code and immediately know whether we have broken existing functionality.

In this iteration, we use unit tests for an entirely different purpose. In this iteration, we use unit tests as part of an application design philosophy called *test-driven development*. When you practice test-driven development, you

write tests first and then write code against the tests.

More precisely, when practicing test-driven development, there are three steps that you complete when creating code (Red/ Green/Refactor):

1. Write a unit test that fails (Red)
2. Write code that passes the unit test (Green)
3. Refactor your code (Refactor)

First, you write the unit test. The unit test should express your intention for how you expect your code to behave. When you first create the unit test, the unit test should fail. The test should fail because you have not yet written any application code that satisfies the test.

Next, you write just enough code in order for the unit test to pass. The goal is to write the code in the laziest, sloppiest and fastest possible way. You should not waste time thinking about the architecture of your application. Instead, you should focus on writing the minimal amount of code necessary to satisfy the intention expressed by the unit test.

Finally, after you have written enough code, you can step back and consider the overall architecture of your application. In this step, you rewrite (refactor) your code by taking advantage of software design patterns -- such as the repository pattern -- so that your code is more maintainable. You can fearlessly rewrite your code in this step because your code is covered by unit tests.

There are many benefits that result from practicing test-driven development. First, test-driven development forces you to focus on code that actually needs to be written. Because you are constantly focused on just writing enough code to pass a particular test, you are prevented from wandering into the weeds and writing massive amounts of code that you will never use.

Second, a "test first" design methodology forces you to write code from the perspective of how your code will be used. In other words, when practicing test-driven development, you are constantly writing your tests from a user perspective. Therefore, test-driven development can result in cleaner and more understandable APIs.

Finally, test-driven development forces you to write unit tests as part of the normal process of writing an application. As a project deadline approaches, testing is typically the first thing that goes out the window. When practicing test-driven development, on the other hand, you are more likely to be virtuous about writing unit tests because test-driven development makes unit tests central to the process of building an application.

> **NOTE**
> To learn more about test-driven development, I recommend that you read Michael Feathers book **Working Effectively with Legacy Code**.

In this iteration, we add a new feature to our Contact Manager application. We add support for Contact Groups. You can use Contact Groups to organize your contacts into categories such as Business and Friend groups.

We'll add this new functionality to our application by following a process of test-driven development. We'll write our unit tests first and we'll write all of our code against these tests.

What Gets Tested

As we discussed in the previous iteration, you typically do not write unit tests for data access logic or view logic. You don t write unit tests for data access logic because accessing a database is a relatively slow operation. You don t write unit tests for view logic because accessing a view requires spinning up a web server which is a relatively slow operation. You shouldn't write a unit test unless the test can be executed over and over again very fast

Because test-driven development is driven by unit tests, we focus initially on writing controller and business logic.

We avoid touching the database or views. We won't modify the database or create our views until the very end of this tutorial. We start with what can be tested.

Creating User Stories

When practicing test-driven development, you always start by writing a test. This immediately raises the question: How do you decide what test to write first? To answer this question, you should write a set of **user stories**.

A user story is a very brief (usually one sentence) description of a software requirement. It should be a non-technical description of a requirement written from a user perspective.

Here s the set of user stories that describe the features required by the new Contact Group functionality:

1. User can view a list of contact groups.
2. User can create a new contact group.
3. User can delete an existing contact group.
4. User can select a contact group when creating a new contact.
5. User can select a contact group when editing an existing contact.
6. A list of contact groups is displayed in the Index view.
7. When a user clicks a contact group, a list of matching contacts is displayed.

Notice that this list of user stories is completely understandable by a customer. There is no mention of technical implementation details.

While in the process of building your application, the set of user stories can become more refined. You might break a user story into multiple stories (requirements). For example, you might decide that creating a new contact group should involve validation. Submitting a contact group without a name should return a validation error.

After you create a list of user stories, you are ready to write your first unit test. We'll start by creating a unit test for viewing the list of contact groups.

Listing Contact Groups

Our first user story is that a user should be able to view a list of contact groups. We need to express this story with a test.

Create a new unit test by right-clicking the Controllers folder in the ContactManager.Tests project, selecting **Add, New Test**, and selecting the **Unit Test** template (see Figure 1). Name the new unit test GroupControllerTest.cs and click the **OK** button.

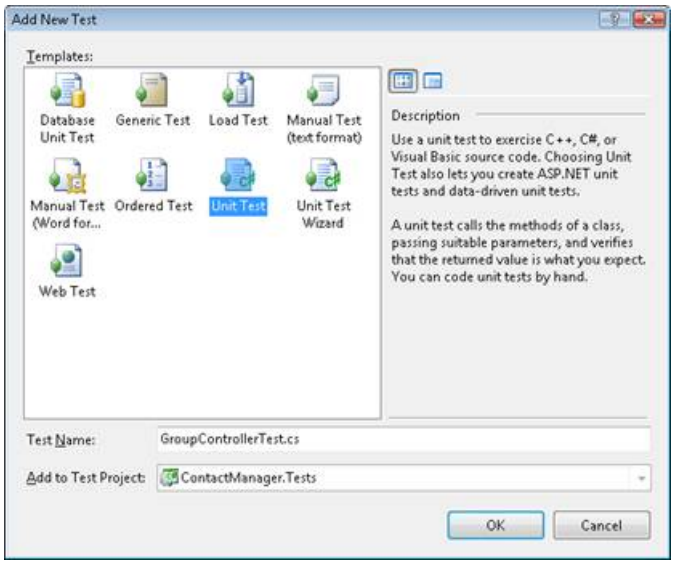


Figure 01: Adding the GroupControllerTest unit test(Click to view full-size image)

Our first unit test is contained in Listing 1. This test verifies that the Index() method of the Group controller returns a set of Groups. The test verifies that a collection of Groups is returned in view data.

Listing 1 - Controllers\GroupControllerTest.cs

```
using System;
using System.Text;
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Web.Mvc;
using ContactManager.Models;

namespace ContactManager.Tests.Controllers
{
    [TestClass]
    public class GroupControllerTest
    {

        [TestMethod]
        public void Index()
        {
            // Arrange
            var controller = new GroupController();

            // Act
            var result = (ViewResult)controller.Index();

            // Assert
            Assert.IsInstanceOfType(result.ViewData.Model, typeof(IEnumerable));
        }
    }
}
```

When you first type the code in Listing 1 in Visual Studio, you'll get a lot of red squiggly lines. We have not created the GroupController or Group classes.

At this point, we can t even build our application so we can t execute our first unit test. That s good. That counts as a failing test. Therefore, we now have permission to start writing application code. We need to write enough code to execute our test.

The Group controller class in Listing 2 contains the bare minimum of code required to pass the unit test. The Index() action returns a statically coded list of Groups (the Group class is defined in Listing 3).

Listing 2 - Controllers\GroupController.cs

```
using System.Collections.Generic;
using System.Web.Mvc;
using ContactManager.Models;

namespace ContactManager.Controllers
{
    public class GroupController : Controller
    {
        public ActionResult Index()
        {
            var groups = new List();
            return View(groups);
        }

    }
}
```

Listing 3 - Models\Group.cs

```
namespace ContactManager.Models
{
    public class Group
    {
    }
}
```

After we add the GroupController and Group classes to our project, our first unit test completes successfully (see Figure 2). We have done the minimum work required to pass the test. It is time to celebrate.

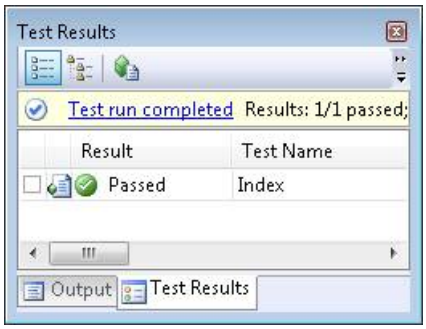


Figure 02: Success!(Click to view full-size image)

Creating Contact Groups

Now we can move on to the second user story. We need to be able to create new contact groups. We need to express this intention with a test.

The test in Listing 4 verifies that calling the Create() method with a new Group adds the Group to the list of Groups returned by the Index() method. In other words, if I create a new group then I should be able to get the new Group back from the list of Groups returned by the Index() method.

Listing 4 - Controllers\GroupControllerTest.cs

```
[TestMethod]
public void Create()
{
    // Arrange
    var controller = new GroupController();

    // Act
    var groupToCreate = new Group();
    controller.Create(groupToCreate);

    // Assert
    var result = (ViewResult)controller.Index();
    var groups = (IEnumerable<Group>)result.ViewData.Model;
    CollectionAssert.Contains(groups.ToList(), groupToCreate);
}
```

The test in Listing 4 calls the Group controller Create() method with a new contact Group. Next, the test verifies that calling the Group controller Index() method returns the new Group in view data.

The modified Group controller in Listing 5 contains the bare minimum of changes required to pass the new test.

Listing 5 - Controllers\GroupController.cs

```
using System.Collections.Generic;
using System.Web.Mvc;
using ContactManager.Models;
using System.Collections;

namespace ContactManager.Controllers
{
    public class GroupController : Controller
    {
        private IList<Group> _groups = new List<Group>();

        public ActionResult Index()
        {
            return View(_groups);
        }

        public ActionResult Create(Group groupToCreate)
        {
            _groups.Add(groupToCreate);
            return RedirectToAction("Index");
        }
    }
}
```

The Group controller in Listing 5 has a new Create() action. This action adds a Group to a collection of Groups. Notice that the Index() action has been modified to return the contents of the collection of Groups.

Once again, we have performed the bare minimum amount of work required to pass the unit test. After we make these changes to the Group controller, all of our unit tests pass.

Adding Validation

This requirement was not stated explicitly in the user story. However, it is reasonable to require that a Group have a name. Otherwise, organizing contacts into Groups would not be very useful.

Listing 6 contains a new test that expresses this intention. This test verifies that attempting to create a Group

without supplying a name results in a validation error message in model state.

Listing 6 - Controllers\GroupControllerTest.cs

```
[TestMethod]
public void CreateRequiredName()
{
    // Arrange
    var controller = new GroupController();

    // Act
    var groupToCreate = new Group();
    groupToCreate.Name = String.Empty;
    var result = (ViewResult)controller.Create(groupToCreate);

    // Assert
    var error = result.ViewData.ModelState["Name"].Errors[0];
    Assert.AreEqual("Name is required.", error.ErrorMessage);
}
```

In order to satisfy this test, we need to add a Name property to our Group class (see Listing 7). Furthermore, we need to add a tiny bit of validation logic to our Group controller s Create() action (see Listing 8).

Listing 7 - Models\Group.cs

```
namespace ContactManager.Models
{
    public class Group
    {
        public string Name { get; set; }
    }
}
```

Listing 8 - Controllers\GroupController.cs

```
public ActionResult Create(Group groupToCreate)
{
    // Validation logic
    if (groupToCreate.Name.Trim().Length == 0)
    {
        ModelState.AddModelError("Name", "Name is required.");
        return View("Create");
    }

    // Database logic
    _groups.Add(groupToCreate);
    return RedirectToAction("Index");
}
```

Notice that the Group controller Create() action now contains both validation and database logic. Currently, the database used by the Group controller consists of nothing more than an in-memory collection.

Time to Refactor

The third step in Red/Green/Refactor is the Refactor part. At this point, we need to step back from our code and consider how we can refactor our application to improve its design. The Refactor stage is the stage at which we think hard about the best way of implementing software design principles and patterns.

We are free to modify our code in any way that we choose to improve the design of the code. We have a safety net of unit tests that prevent us from breaking existing functionality.

Right now, our Group controller is a mess from the perspective of good software design. The Group controller contains a tangled mess of validation and data access code. To avoid violating the Single Responsibility Principle, we need to separate these concerns into different classes.

Our refactored Group controller class is contained in Listing 9. The controller has been modified to use the ContactManager service layer. This is the same service layer that we use with the Contact controller.

Listing 10 contains the new methods added to the ContactManager service layer to support validating, listing, and creating groups. The IContactManagerService interface was updated to include the new methods.

Listing 11 contains a new FakeContactManagerRepository class that implements the IContactManagerRepository interface. Unlike the EntityContactManagerRepository class that also implements the IContactManagerRepository interface, our new FakeContactManagerRepository class does not communicate with the database. The FakeContactManagerRepository class uses an in-memory collection as a proxy for the database. We'll use this class in our unit tests as a fake repository layer.

Listing 9 - Controllers\GroupController.cs

```
using System.Web.Mvc;
using ContactManager.Models;

namespace ContactManager.Controllers
{
    public class GroupController : Controller
    {

        private IContactManagerService _service;

        public GroupController()
        {
            _service = new ContactManagerService(new ModelStateWrapper(this.ModelState));
        }

        public GroupController(IContactManagerService service)
        {
            _service = service;
        }

        public ActionResult Index()
        {
            return View(_service.ListGroups());
        }

        public ActionResult Create(Group groupToCreate)
        {
            if (_service.CreateGroup(groupToCreate))
                return RedirectToAction("Index");
            return View("Create");
        }
    }
}
```

Listing 10 - Controllers\ContactManagerService.cs

```
public bool ValidateGroup(Group groupToValidate)
{
    if (groupToValidate.Name.Trim().Length == 0)
       _validationDictionary.AddError("Name", "Name is required.");
    return _validationDictionary.IsValid;
}

public bool CreateGroup(Group groupToCreate)
{
    // Validation logic
    if (!ValidateGroup(groupToCreate))
        return false;

    // Database logic
    try
    {
        _repository.CreateGroup(groupToCreate);
    }
    catch
    {
        return false;
    }
    return true;
}

public IEnumerable<Group> ListGroups()
{
    return _repository.ListGroups();
}
```

Listing 11 - Controllers\FakeContactManagerRepository.cs

```
using System;
using System.Collections.Generic;
using ContactManager.Models;

namespace ContactManager.Tests.Models
{
    public class FakeContactManagerRepository : IContactManagerRepository
    {
        private IList<Group> _groups = new List<Group>();

        #region IContactManagerRepository Members

        // Group methods

        public Group CreateGroup(Group groupToCreate)
        {
            _groups.Add(groupToCreate);
            return groupToCreate;
        }

        public IEnumerable<Group> ListGroups()
        {
            return _groups;
        }

        // Contact methods

        public Contact CreateContact(Contact contactToCreate)
        {
            throw new NotImplementedException();
        }

        public void DeleteContact(Contact contactToDelete)
        {
            throw new NotImplementedException();
        }

        public Contact EditContact(Contact contactToEdit)
        {
            throw new NotImplementedException();
        }

        public Contact GetContact(int id)
        {
            throw new NotImplementedException();
        }

        public IEnumerable<Contact> ListContacts()
        {
            throw new NotImplementedException();
        }

        #endregion
    }
}
```

Modifying the IContactManagerRepository interface requires use to implement the CreateGroup() and ListGroups() methods in the EntityContactManagerRepository class. The laziest and fastest way to do this is to add stub methods that look like this:

```
public Group CreateGroup(Group groupToCreate)
{
    throw new NotImplementedException();
}

public IEnumerable<Group> ListGroups()
{
    throw new NotImplementedException();
}
```

Finally, these changes to the design of our application require us to make some modifications to our unit tests. We now need to use the FakeContactManagerRepository when performing the unit tests. The updated GroupControllerTest class is contained in Listing 12.

Listing 12 - Controllers\GroupControllerTest.cs

```
using System.Collections.Generic;
using System.Web.Mvc;
using ContactManager.Controllers;
using ContactManager.Models;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Collections;
using System.Linq;
using System;
using ContactManager.Tests.Models;

namespace ContactManager.Tests.Controllers
{
    [TestClass]
    public class GroupControllerTest
    {
        private IContactManagerRepository _repository;
        private ModelStateDictionary _modelState;
        private IContactManagerService _service;

        [TestInitialize]
        public void Initialize()
        {
            _repository = new FakeContactManagerRepository();
            _modelState = new ModelStateDictionary();
            _service = new ContactManagerService(new ModelStateWrapper(_modelState), _repository);

        }

        [TestMethod]
        public void Index()
        {
            // Arrange
            var controller = new GroupController(_service);

            // Act
            var result = (ViewResult)controller.Index();

            // Assert
            Assert.IsInstanceOfType(result.ViewData.Model, typeof(IEnumerable));
        }

        [TestMethod]
        public void Create()
        {
            // Arrange
            var controller = new GroupController(_service);

            // Act
            var groupToCreate = new Group();
```

```
            groupToCreate.Name = "Business";
            controller.Create(groupToCreate);

            // Assert
            var result = (ViewResult)controller.Index();
            var groups = (IEnumerable)result.ViewData.Model;
            CollectionAssert.Contains(groups.ToList(), groupToCreate);
        }

        [TestMethod]
        public void CreateRequiredName()
        {
            // Arrange
            var controller = new GroupController(_service);

            // Act
            var groupToCreate = new Group();
            groupToCreate.Name = String.Empty;
            var result = (ViewResult)controller.Create(groupToCreate);

            // Assert
            var error = _modelState["Name"].Errors[0];
            Assert.AreEqual("Name is required.", error.ErrorMessage);
        }

    }
}
```

After we make all of these changes, once again, all of our unit tests pass. We have completed the entire cycle of Red/Green/Refactor. We have implemented the first two user stories. We now have supporting unit tests for the requirements expressed in the user stories. Implementing the remainder of the user stories involves repeating the same cycle of Red/Green/Refactor.

Modifying our Database

Unfortunately, even though we have satisfied all of the requirements expressed by our unit tests, our work is not done. We still need to modify our database.

We need to create a new Group database table. Follow these steps:

1. In the Server Explorer window, right-click the Tables folder and select the menu option **Add New Table**.
2. Enter the two columns described below in the Table Designer.
3. Mark the Id column as a primary key and Identity column.
4. Save the new table with the name Groups by clicking the icon of the floppy.

COLUMN NAME	DATA TYPE	ALLOW NULLS
Id	int	False
Name	nvarchar(50)	False

Next, we need to delete all of the data from the Contacts table (otherwise, we won't be able to create a relationship between the Contacts and Groups tables). Follow these steps:

1. Right-click the Contacts table and select the menu option **Show Table Data**.
2. Delete all of the rows.

Next, we need to define a relationship between the Groups database table and the existing Contacts database table. Follow these steps:

1. Double-click the Contacts table in the Server Explorer window to open the Table Designer.
2. Add a new integer column to the Contacts table named GroupId.
3. Click the Relationship button to open the Foreign Key Relationships dialog (see Figure 3).
4. Click the Add button.
5. Click the ellipsis button that appears next to the Table and Columns Specification button.
6. In the Tables and Columns dialog, select Groups as the primary key table and Id as the primary key column. Select Contacts as the foreign key table and GroupId as the foreign key column (see Figure 4). Click the OK button.
7. Under **INSERT and UPDATE Specification**, select the value **Cascade** for **Delete Rule**.
8. Click the Close button to close the Foreign Key Relationships dialog.
9. Click the Save button to save the changes to the Contacts table.

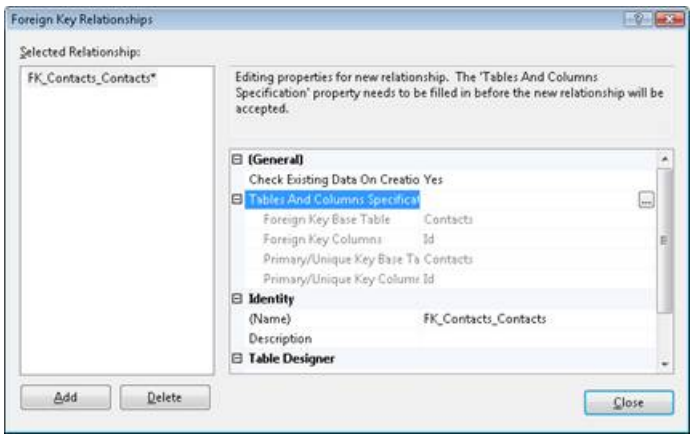


Figure 03: Creating a database table relationship (Click to view full-size image)

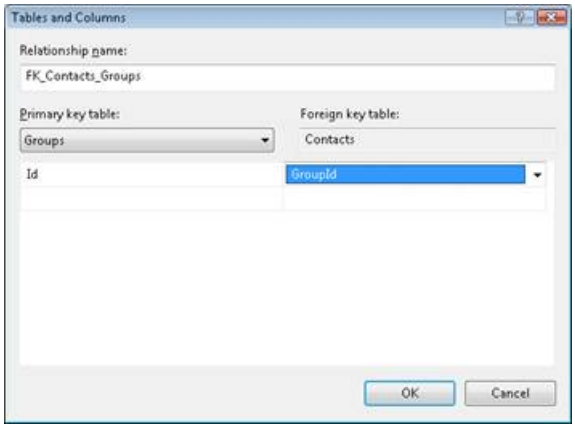


Figure 04: Specifying table relationships(Click to view full-size image)

Updating our Data Model

Next, we need to update our data model to represent the new database table. Follow these steps:

1. Double-click the ContactManagerModel.edmx file in the Models folder to open the Entity Designer.
2. Right-click the Designer surface and select the menu option **Update Model from Database**.
3. In the Update Wizard, select the Groups table and click the Finish button (see Figure 5).
4. Right-click the Groups entity and select the menu option **Rename**. Change the name of the *Groups* entity to *Group* (singular).
5. Right-click the Groups navigation property that appears at the bottom of the Contact entity. Change the name of the *Groups* navigation property to *Group* (singular).

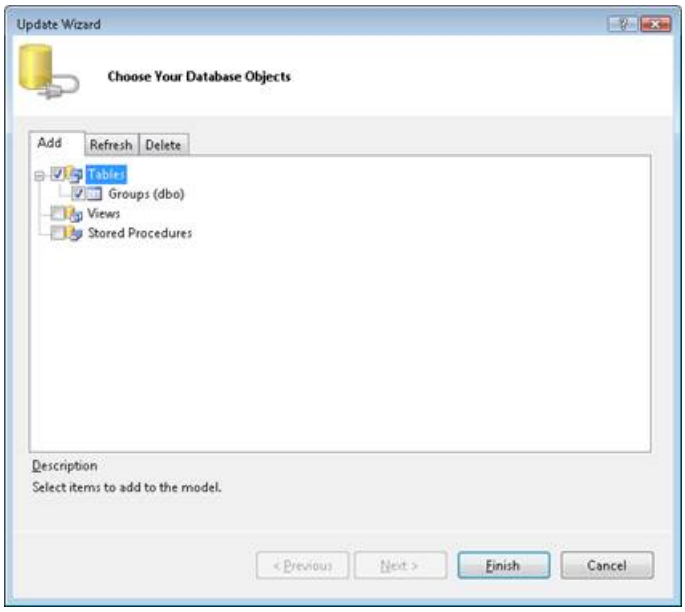


Figure 05: Updating an Entity Framework model from the database(Click to view full-size image)

After you complete these steps, your data model will represent both the Contacts and Groups tables. The Entity Designer should show both entities (see Figure 6).

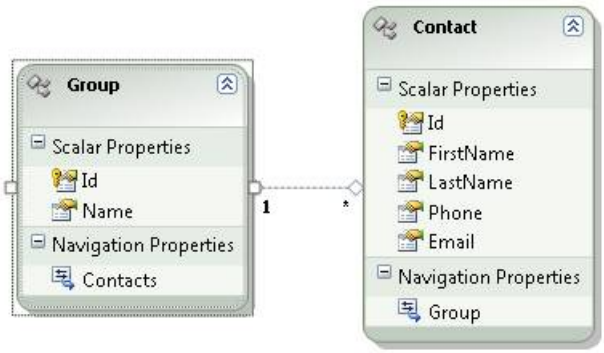


Figure 06: Entity Designer displaying Group and Contact(Click to view full-size image)

Creating our Repository Classes

Next, we need to implement our repository class. Over the course of this iteration, we added several new methods to the IContactManagerRepository interface while writing code to satisfy our unit tests. The final version of the IContactManagerRepository interface is contained in Listing 14.

Listing 14 - Models\IContactManagerRepository.cs

```
using System.Collections.Generic;

namespace ContactManager.Models
{
    public interface IContactManagerRepository
    {
        // Contact methods
        Contact CreateContact(int groupId, Contact contactToCreate);
        void DeleteContact(Contact contactToDelete);
        Contact EditContact(int groupId, Contact contactToEdit);
        Contact GetContact(int id);

        // Group methods
        Group CreateGroup(Group groupToCreate);
        IEnumerable<Group> ListGroups();
        Group GetGroup(int groupId);
        Group GetFirstGroup();
        void DeleteGroup(Group groupToDelete);
    }
}
```

We haven't actually implemented any of the methods related to working with contact groups. Currently, the EntityContactManagerRepository class has stub methods for each of the contact group methods listed in the IContactManagerRepository interface. For example, the ListGroups() method currently looks like this:

```
public IEnumerable<Group> ListGroups()
{
    throw new NotImplementedException();
}
```

The stub methods enabled us to compile our application and pass the unit tests. However, now it is time to actually implement these methods. The final version of the EntityContactManagerRepository class is contained in Listing 13.

Listing 13 - Models\EntityContactManagerRepository.cs

```
using System.Collections.Generic;
using System.Linq;
using System;

namespace ContactManager.Models
{
    public class EntityContactManagerRepository : ContactManager.Models.IContactManagerRepository
    {
        private ContactManagerDBEntities _entities = new ContactManagerDBEntities();

        // Contact methods

        public Contact GetContact(int id)
        {
            return (from c in _entities.ContactSet.Include("Group")
                    where c.Id == id
                    select c).FirstOrDefault();
        }

        public Contact CreateContact(int groupId, Contact contactToCreate)
        {
            // Associate group with contact
            contactToCreate.Group = GetGroup(groupId);

            // Save new contact
            _entities.AddToContactSet(contactToCreate);
            _entities.SaveChanges();
```

```
            return contactToCreate;
        }

        public Contact EditContact(int groupId, Contact contactToEdit)
        {
            // Get original contact
            var originalContact = GetContact(contactToEdit.Id);

            // Update with new group
            originalContact.Group = GetGroup(groupId);

            // Save changes
            _entities.ApplyPropertyChanges(originalContact.EntityKey.EntitySetName, contactToEdit);
            _entities.SaveChanges();
            return contactToEdit;
        }

        public void DeleteContact(Contact contactToDelete)
        {
            var originalContact = GetContact(contactToDelete.Id);
            _entities.DeleteObject(originalContact);
            _entities.SaveChanges();
        }

        public Group CreateGroup(Group groupToCreate)
        {
            _entities.AddToGroupSet(groupToCreate);
            _entities.SaveChanges();
            return groupToCreate;
        }

        // Group Methods

        public IEnumerable<Group> ListGroups()
        {
            return _entities.GroupSet.ToList();
        }

        public Group GetFirstGroup()
        {
            return _entities.GroupSet.Include("Contacts").FirstOrDefault();
        }

        public Group GetGroup(int id)
        {
            return (from g in _entities.GroupSet.Include("Contacts")
                        where g.Id == id
                        select g).FirstOrDefault();
        }

        public void DeleteGroup(Group groupToDelete)
        {
            var originalGroup = GetGroup(groupToDelete.Id);
            _entities.DeleteObject(originalGroup);
            _entities.SaveChanges();

        }

    }
}
```

Creating the Views

ASP.NET MVC application when you use the default ASP.NET view engine. So, you don t create views in response to a particular unit test. However, because an application would be useless without views, we can t complete this iteration without creating and modifying the views contained in the Contact Manager application.

We need to create the following new views for managing contact groups (see Figure 7):

- Views\Group\Index.aspx - Displays list of contact groups
- Views\Group\Delete.aspx - Displays confirmation form for deleting a contact group

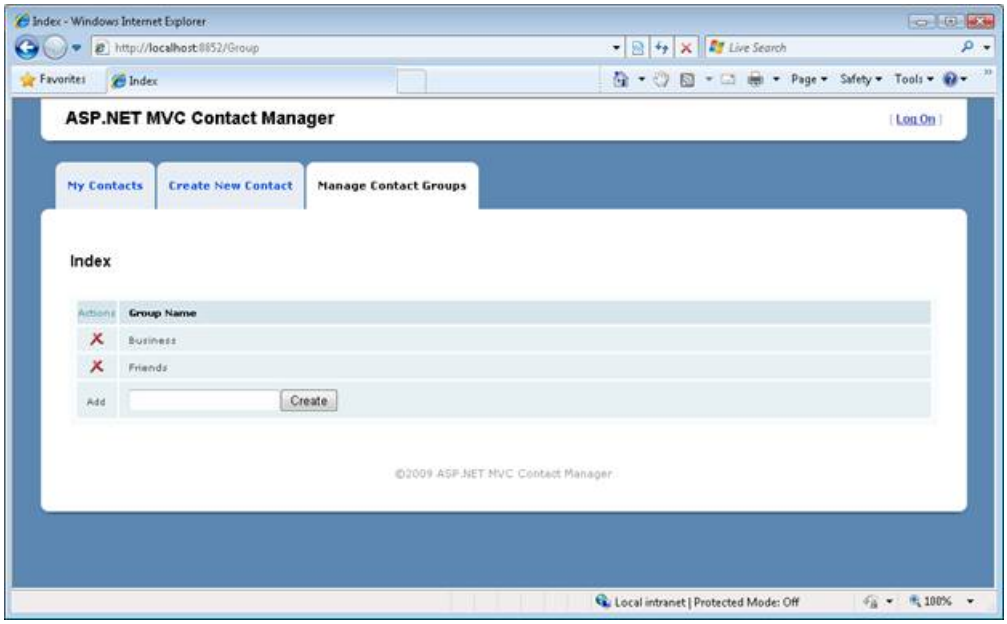


Figure 07: The Group Index view(Click to view full-size image)

We need to modify the following existing views so that they include contact groups:

- Views\Home\Create.aspx
- Views\Home\Edit.aspx
- Views\Home\Index.aspx

You can see the modified views by looking at the Visual Studio application that accompanies this tutorial. For example, Figure 8 illustrates the Contact Index view.

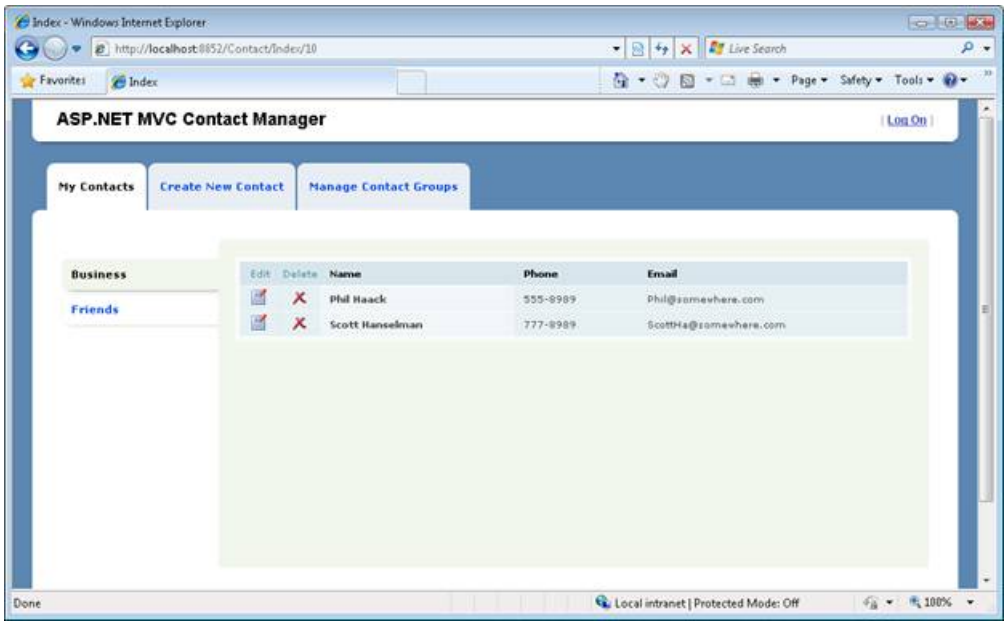


Figure 08: The Contact Index view(Click to view full-size image)

Summary

In this iteration, we added new functionality to our Contact Manager application by following a test-driven development application design methodology. We started by creating a set of user stories. We created a set of unit

tests that corresponds to the requirements expressed by the user stories. Finally, we wrote just enough code to satisfy the requirements expressed by the unit tests.

After we finished writing enough code to satisfy the requirements expressed by the unit tests, we updated our database and views. We added a new Groups table to our database and updated our Entity Framework Data Model. We also created and modified a set of views.

In the next iteration -- the final iteration -- we rewrite our application to take advantage of Ajax. By taking advantage of Ajax, we'll improve the responsiveness and performance of the Contact Manager application.

PREVIOUS NEXT

Iteration #7 – Add Ajax functionality (C#)

4/10/2020 • 18 minutes to read • Edit Online

by Microsoft

Download Code

> In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

Building a Contact Management ASP.NET MVC Application (C#)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information - names, phone numbers and email addresses - for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 - Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 - Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 - Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 - Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 - Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 - Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 - Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

In this iteration of the Contact Manager application, we refactor our application to make use of Ajax. By taking advantage of Ajax, we make our application more responsive. We can avoid rendering an entire page when we need to update only a certain region in a page.

We'll refactor our Index view so that we don t need to redisplay the entire page whenever someone selects a new contact group. Instead, when someone clicks a contact group, we'll just update the list of contacts and leave the rest of the page alone.

We'll also change the way our delete link works. Instead of displaying a separate confirmation page, we'll display a JavaScript confirmation dialog. If you confirm that you want to delete a contact, an HTTP DELETE operation is performed against the server to delete the contact record from the database.

Furthermore, we will take advantage of jQuery to add animation effects to our Index view. We'll display an animation when the new list of contacts is being fetched from the server.

Finally, we'll take advantage of the ASP.NET AJAX framework support for managing browser history. We'll create history points whenever we perform an Ajax call to update the contact list. That way, the browser backward and forward buttons will work.

Why use Ajax?

Using Ajax has many benefits. First, adding Ajax functionality to an application results in a better user experience. In a normal web application, the entire page must be posted back to the server each and every time a user performs an action. Whenever you perform an action, the browser locks and the user must wait until the entire page is fetched and redisplayed.

This would be an unacceptable experience in the case of a desktop application. But, traditionally, we lived with this bad user experience in the case of a web application because we did not know that we could do any better. We thought it was a limitation of web applications when, in actuality, it was just a limitation of our imaginations.

In an Ajax application, you don t need to bring the user experience to a halt just to update a page. Instead, you can perform an asynchronous request in the background to update the page. You don t force the user to wait while part of the page gets updated.

By taking advantage of Ajax, you also can improve the performance of your application. Consider how the Contact Manager application works right now without Ajax functionality. When you click a contact group, the entire Index view must be redisplayed. The list of contacts and list of contact groups must be retrieved from the database server. All of this data must be passed across the wire from web server to web browser.

After we add Ajax functionality to our application, however, we can avoid redisplaying the entire page when a user clicks a contact group. We no longer need to grab the contact groups from the database. We also don t need to push the entire Index view across the wire. By taking advantage of Ajax, we reduce the amount of work that our database server must perform and we reduce the amount of network traffic required by our application.

Don t be Afraid of Ajax

Some developers avoid using Ajax because they worry about downlevel browsers. They want to make sure that their web applications will still work when accessed by a browser that does not support JavaScript. Because Ajax depends on JavaScript, some developers avoid using Ajax.

However, if you are careful about how you implement Ajax then you can build applications that work with both uplevel and downlevel browsers. Our Contact Manager application will work with browsers that support JavaScript and browsers that do not.

If you use the Contact Manager application with a browser that supports JavaScript then you will have a better user experience. For example, when you click a contact group, only the region of the page that displays contacts will be updated.

If, on the other hand, you use the Contact Manager application with a browser that does not support JavaScript (or that has JavaScript disabled) then you will have a slightly less desirable user experience. For example, when you click a contact group, the entire Index view must be posted back to the browser in order to display the matching list of contacts.

Adding the Required JavaScript Files

We'll need to use three JavaScript files to add Ajax functionality to our application. All three of these files are included in the Scripts folder of a new ASP.NET MVC application.

If you plan to use Ajax in multiple pages in your application then it makes sense to include the required JavaScript files in your application s view master page. That way, the JavaScript files will be included in all of the pages in your application automatically.

Add the following JavaScript includes inside the <head> tag of your view master page:

```
<script src="../../Scripts/MicrosoftAjax.js" type="text/javascript"></script>
    <script src="../../Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>
    <script src="../../Scripts/jquery-1.2.6.min.js" type="text/javascript"></script>
```

Refactoring the Index View to use Ajax

Let s start by modifying our Index view so that clicking a contact group only updates the region of the view that displays contacts. The red box in Figure 1 contains the region that we want to update.

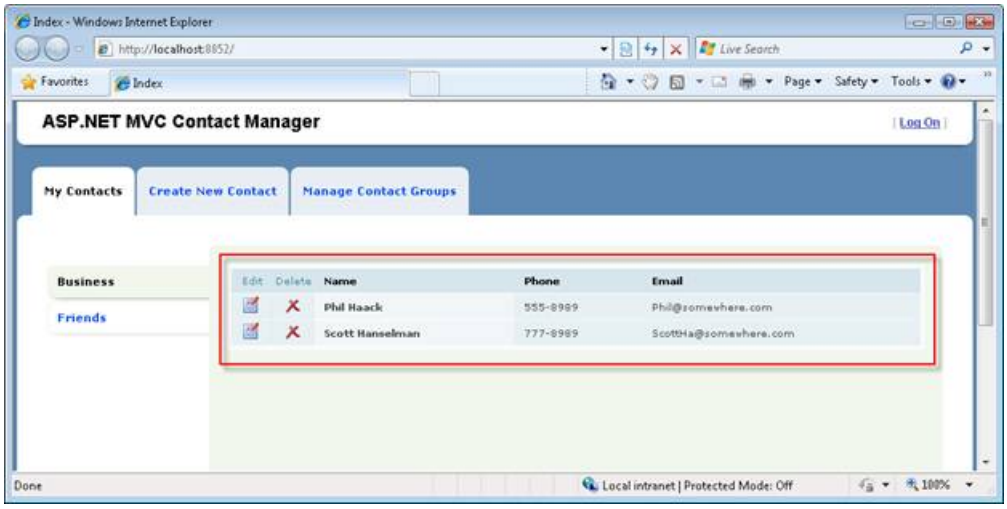


Figure 01: Updating only contacts(Click to view full-size image)

The first step is to separate the part of the view that we want to update asynchronously into a separate partial (view user control). The section of the Index view that displays the table of contacts has been moved into the partial in Listing 1.

Listing 1 - Views\Contact\ContactList.ascx

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<ContactManager.Models.Group>" %>
<%@ Import Namespace="Helpers" %>
<table class="data-table" cellpadding="0" cellspacing="0">
    <thead>
        <tr>
            <th class="actions edit">
                Edit
            </th>
            <th class="actions delete">
                Delete
            </th>
            <th>
                Name
            </th>
            <th>
                Phone
            </th>
            <th>
                Email
            </th>
        </tr>
    </thead>
    <tbody>
        <% foreach (var item in Model.Contacts)
           { %>
        <tr>
            <td class="actions edit">
                <a href='<%= Url.Action("Edit", new {id=item.Id}) %>'><img src="../../Content/Edit.png"
alt="Edit" /></a>
            </td>
            <td class="actions delete">
                <a href='<%= Url.Action("Delete", new {id=item.Id}) %>'><img src="../../Content/Delete.png"
alt="Edit" /></a>
            </td>
            <th>
                <%= Html.Encode(item.FirstName) %>
                <%= Html.Encode(item.LastName) %>
            </th>
            <td>
                <%= Html.Encode(item.Phone) %>
            </td>
            <td>
                <%= Html.Encode(item.Email) %>
            </td>
        </tr>
        <% } %>
    </tbody>
</table>
```

Notice that the partial in Listing 1 has a different model than the Index view. The *Inherits* attribute in the <%@ Page %> directive specifies that the partial inherits from the ViewUserControl<Group> class.

The updated Index view is contained in Listing 2.

Listing 2 - Views\Contact\Index.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.ViewData.IndexModel>" %>
<%@ Import Namespace="Helpers" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Index</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<ul id="leftColumn">
<% foreach (var item in Model.Groups) { %>
    <li <%= Html.Selected(item.Id, Model.SelectedGroup.Id) %>>
    <%= Ajax.ActionLink(item.Name, "Index", new { id = item.Id }, new AjaxOptions { UpdateTargetId =
"divContactList"})%>
    </li>
<% } %>
</ul>
<div id="divContactList">
    <% Html.RenderPartial("ContactList", Model.SelectedGroup); %>
</div>

<div class="divContactList-bottom"> </div>
</asp:Content>
```

There are two things that you should notice about the updated view in Listing 2. First, notice that all of the content moved into the partial is replaced with a call to Html.RenderPartial(). The Html.RenderPartial() method is called when the Index view is first requested in order to display the initial set of contacts.

Second, notice that the Html.ActionLink() used to display contact groups has been replaced with an Ajax.ActionLink(). The Ajax.ActionLink() is called with the following parameters:

```
<%= Ajax.ActionLink(item.Name, "Index", new { id = item.Id }, new AjaxOptions { UpdateTargetId =
"divContactList"})%>
```

The first parameter represents the text to display for the link, the second parameter represents the route values, and the third parameter represents the Ajax options. In this case, we use the UpdateTargetId Ajax option to point to the HTML tag that we want to update after the Ajax request completes. We want to update the tag with the new list of contacts.

The updated Index() method of the Contact controller is contained in Listing 3.

Listing 3 - Controllers\ContactController.cs (Index method)

```
public ActionResult Index(int? id)
{
    // Get selected group
    var selectedGroup = _service.GetGroup(id);
    if (selectedGroup == null)
        return RedirectToAction("Index", "Group");

    // Normal Request
    if (!Request.IsAjaxRequest())
    {
        var model = new IndexModel
        {
            Groups = _service.ListGroups(),
            SelectedGroup = selectedGroup
        };
        return View("Index", model);
    }

    // Ajax Request
    return PartialView("ContactList", selectedGroup);
}
```

The updated Index() action conditionally returns one of two things. If the Index() action is invoked by an Ajax request then the controller returns a partial. Otherwise, the Index() action returns an entire view.

Notice that the Index() action does not need to return as much data when invoked by an Ajax request. In the context of a normal request, the Index action returns a list of all of the contact groups and the selected contact group. In the context of an Ajax request, the Index() action returns only the selected group. Ajax means less work on your database server.

Our modified Index view works in the case of both uplevel and downlevel browsers. If you click a contact group, and your browser supports JavaScript, then only the region of the view that contains the list of contacts is updated. If, on the other hand, your browser does not support JavaScript, then the entire view is updated.

Our updated Index view has one problem. When you click a contact group, the selected group is not highlighted. Because the list of groups is displayed outside of the region that is updated during an Ajax request, the right group does not get highlighted. We'll fix this issue in the next section.

Adding jQuery Animation Effects

Normally, when you click a link in a web page, you can use the browser progress bar to detect whether or not the browser is actively fetching the updated content. When performing an Ajax request, on the other hand, the browser progress bar does not show any progress. This can make users nervous. How do you know whether the browser has frozen?

There are several ways that you can indicate to a user that work is being performed while performing an Ajax request. One approach is to display a simple animation. For example, you can fade out a region when an Ajax request begins and fade in the region when the request completes.

We'll use the jQuery library which is included with the Microsoft ASP.NET MVC framework, to create the animation effects. The updated Index view is contained in Listing 4.

Listing 4 - Views\Contact\Index.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.ViewData.IndexModel>" %>
<%@ Import Namespace="Helpers" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Index</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<script type="text/javascript">

    function beginContactList(args) {
        // Highlight selected group
        $('#leftColumn li').removeClass('selected');
        $(this).parent().addClass('selected');

        // Animate
        $('#divContactList').fadeOut('normal');
    }

    function successContactList() {
        // Animate
        $('#divContactList').fadeIn('normal');
    }

    function failureContactList() {
        alert("Could not retrieve contacts.");
    }

</script>

<ul id="leftColumn">
<% foreach (var item in Model.Groups) { %>
    <li <%= Html.Selected(item.Id, Model.SelectedGroup.Id) %>>
    <%= Ajax.ActionLink(item.Name, "Index", new { id = item.Id }, new AjaxOptions { UpdateTargetId =
"divContactList", OnBegin = "beginContactList", OnSuccess = "successContactList", OnFailure =
"failureContactList" })%>
    </li>
<% } %>
</ul>
<div id="divContactList">
    <% Html.RenderPartial("ContactList", Model.SelectedGroup); %>
</div>

<div class="divContactList-bottom"> </div>
</asp:Content>
```

Notice that the updated Index view contains three new JavaScript functions. The first two functions use jQuery to fade out and fade in the list of contacts when you click a new contact group. The third function displays an error message when an Ajax request results in an error (for example, network timeout).

The first function also takes care of highlighting the selected group. A class= selected attribute is added to the parent element (the LI element) of the element clicked. Again, jQuery makes it easy to select the right element and add the CSS class.

These scripts are tied to the group links with the help of the Ajax.ActionLink() AjaxOptions parameter. The updated Ajax.ActionLink() method call looks like this:

```
<%= Ajax.ActionLink(item.Name, "Index", new { id = item.Id }, new AjaxOptions { UpdateTargetId =
"divContactList", OnBegin = "beginContactList", OnSuccess = "successContactList", OnFailure =
"failureContactList" })%>
```

Adding Browser History Support

Normally, when you click a link to update a page, your browser history is updated. That way, you can click the browser Back button to move back in time to the previous state of the page. For example, if you click the Friends contact group and then click the Business contact group, you can click the browser Back button to navigate back to the state of the page when the Friends contact group was selected.

Unfortunately, performing an Ajax request does not update browser history automatically. If you click a contact group, and the list of matching contacts is retrieved with an Ajax request, then the browser history is not updated. You cannot use the browser Back button to navigate back to a contact group after selecting a new contact group.

If you want users to be able to use the browser Back button after performing Ajax requests then you need to perform a little more work. You need to take advantage of the browser history management functionality built in the ASP.NET AJAX Framework.

ASP.NET AJAX browser history, you need to do three things:

1. Enable Browser History by setting the enableBrowserHistory property to true.
2. Save history points when the state of a view changes by calling the addHistoryPoint() method.
3. Reconstruct the state of the view when the navigate event is raised.

The updated Index view is contained in Listing 5.

Listing 5 - Views\Contact\Index.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<ContactManager.Models.ViewData.IndexModel>" %>
<%@ Import Namespace="Helpers" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Index</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<script type="text/javascript">

    var _currentGroupId = -1;

    Sys.Application.add_init(pageInit);

    function pageInit() {
        // Enable history
        Sys.Application.set_enableHistory(true);

        // Add Handler for history
        Sys.Application.add_navigate(navigate);
    }

    function navigate(sender, e) {
        // Get groupId from address bar
        var groupId = e.get_state().groupId;

        // If groupId != currentGroupId then navigate
        if (groupId != _currentGroupId) {
            _currentGroupId = groupId;
            $("#divContactList").load("/Contact/Index/" + groupId);
            selectGroup(groupId);
        }
    }

    function selectGroup(groupId) {
        $('#leftColumn li').removeClass('selected');
        if (groupId)
            $('a[groupid=' + groupId + ']').parent().addClass('selected');
```

```
        else
            $('#leftColumn li:first').addClass('selected');
    }

    function beginContactList(args) {
        // Highlight selected group
        _currentGroupId = this.getAttribute("groupid");
        selectGroup(_currentGroupId);

        // Add history point
        Sys.Application.addHistoryPoint({ "groupId": _currentGroupId });

        // Animate
        $('#divContactList').fadeOut('normal');
    }

    function successContactList() {
        // Animate
        $('#divContactList').fadeIn('normal');
    }

    function failureContactList() {
        alert("Could not retrieve contacts.");
    }

</script>

<ul id="leftColumn">
<% foreach (var item in Model.Groups) { %>
    <li <%= Html.Selected(item.Id, Model.SelectedGroup.Id) %>>
    <%= Ajax.ActionLink(item.Name, "Index", new { id = item.Id }, new AjaxOptions { UpdateTargetId =
"divContactList", OnBegin = "beginContactList", OnSuccess = "successContactList", OnFailure =
"failureContactList" }, new { groupid = item.Id })%>
    </li>
<% } %>
</ul>
<div id="divContactList">
    <% Html.RenderPartial("ContactList", Model.SelectedGroup); %>
</div>

<div class="divContactList-bottom"> </div>
</asp:Content>
```

In Listing 5, Browser History is enabled in the pageInit() function. The pageInit() function is also used to set up the event handler for the navigate event. The navigate event is raised whenever the browser Forward or Back button causes the state of the page to change.

The beginContactList() method is called when you click a contact group. This method creates a new history point by calling the addHistoryPoint() method. The id of the contact group clicked is added to history.

The group id is retrieved from an expando attribute on the contact group link. The link is rendered with the following call to Ajax.ActionLink().

```
<%= Ajax.ActionLink(item.Name, "Index", new { id = item.Id }, new AjaxOptions { UpdateTargetId =
"divContactList", OnBegin = "beginContactList", OnSuccess = "successContactList", OnFailure =
"failureContactList" }, new {groupid=item.Id})%>
```

The last parameter passed to the Ajax.ActionLink() adds an expando attribute named groupid to the link (lowercase for XHTML compatibility).

When a user hits the browser Back or Forward button, the navigate event is raised and the navigate() method is called. This method updates the contacts displayed in the page to match the state of the page that corresponds to the browser history point passed to the navigate method.

Performing Ajax Deletes

Currently, in order to delete a contact, you need to click on the Delete link and then click the Delete button displayed in the delete confirmation page (see Figure 2). This seems like a lot of page requests to do something simple like deleting a database record.

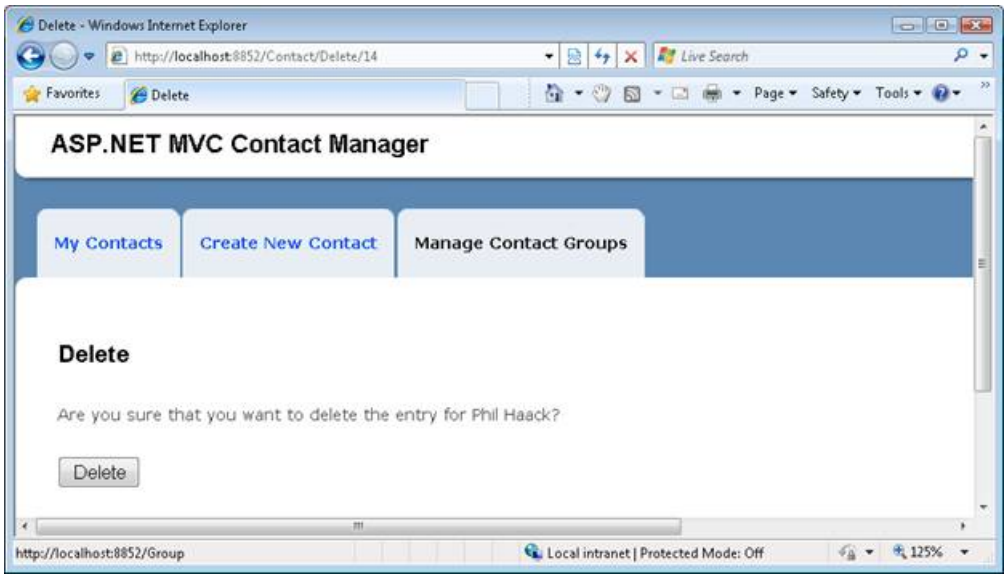


Figure 02: The delete confirmation page(Click to view full-size image)

It is tempting to skip the delete confirmation page and delete a contact directly from the Index view. You should avoid this temptation because taking this approach opens your application to security holes. In general, you don t want to perform an HTTP GET operation when invoking an action that modifies the state of your web application. When performing a delete, you want to perform an HTTP POST, or better yet, an HTTP DELETE operation.

The Delete link is contained in the ContactList partial. An updated version of the ContactList partial is contained in Listing 6.

Listing 6 - Views\Contact\ContactList.ascx

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<ContactManager.Models.Group>" %>
<%@ Import Namespace="Helpers" %>
<table class="data-table" cellpadding="0" cellspacing="0">
    <thead>
        <tr>
            <th class="actions edit">
                Edit
            </th>
            <th class="actions delete">
                Delete
            </th>
            <th>
                Name
            </th>
            <th>
                Phone
            </th>
            <th>
                Email
            </th>
        </tr>
    </thead>
    <tbody>
        <% foreach (var item in Model.Contacts)
           { %>
        <tr>
            <td class="actions edit">
                <a href='<%= Url.Action("Edit", new {id=item.Id}) %>'><img src="../../Content/Edit.png"
alt="Edit" /></a>
            </td>
            <td class="actions delete">
            <%= Ajax.ImageActionLink("../../Content/Delete.png", "Delete", "Delete", new { id = item.Id }, new
AjaxOptions { Confirm = "Delete contact?", HttpMethod = "Delete", UpdateTargetId = "divContactList" })%>
            </td>
            <th>
                <%= Html.Encode(item.FirstName) %>
                <%= Html.Encode(item.LastName) %>
            </th>
            <td>
                <%= Html.Encode(item.Phone) %>
            </td>
            <td>
                <%= Html.Encode(item.Email) %>
            </td>
        </tr>
        <% } %>
    </tbody>
</table>
```

The Delete link is rendered with the following call to the Ajax.ImageActionLink() method:

```
<%= Ajax.ImageActionLink("../../Content/Delete.png", "Delete", "Delete", new { id = item.Id }, new AjaxOptions
{ Confirm = "Delete contact?", HttpMethod = "Delete", UpdateTargetId = "divContactList" })%>
```

NOTE

The Ajax.ImageActionLink() is not a standard part of the ASP.NET MVC framework. The Ajax.ImageActionLink() is a custom helper methods included in the Contact Manager project.

The AjaxOptions parameter has two properties. First, the Confirm property is used to display a popup JavaScript confirmation dialog. Second, the HttpMethod property is used to perform an HTTP DELETE operation.

Listing 7 contains a new AjaxDelete() action that has been added to the Contact controller.

Listing 7 - Controllers\ContactController.cs (AjaxDelete)

```
[AcceptVerbs(HttpVerbs.Delete)]
[ActionName("Delete")]
public ActionResult AjaxDelete(int id)
{
    // Get contact and group
    var contactToDelete = _service.GetContact(id);
    var selectedGroup = _service.GetGroup(contactToDelete.Group.Id);

    // Delete from database
    _service.DeleteContact(contactToDelete);

    // Return Contact List
    return PartialView("ContactList", selectedGroup);
}
```

The AjaxDelete() action is decorated with an AcceptVerbs attribute. This attribute prevents the action from being invoked except by any HTTP operation other than an HTTP DELETE operation. In particular, you cannot invoke this action with an HTTP GET.

After you delete database record, you need to display the updated list of contacts that does not contain the deleted record. The AjaxDelete() method returns the ContactList partial and the updated list of contacts.

Summary

In this iteration, we added Ajax functionality to our Contact Manager application. We used Ajax to improve the responsiveness and performance of our application.

First, we refactored the Index view so that clicking a contact group does not update the entire view. Instead, clicking a contact group only updates the list of contacts.

Next, we used jQuery animation effects to fade out and fade in the list of contacts. Adding animation to an Ajax application can be used to provide users of the application with the equivalent of a browser progress bar.

We also added browser history support to our Ajax application. We enabled users to click the browser Back and Forward buttons to change the state of the Index view.

Finally, we created a delete link that supports HTTP DELETE operations. By performing Ajax deletes, we enable users to delete database records without requiring the user to request an additional delete confirmation page.

PREVIOUS NEXT

Iteration #1 – Create the Application (VB)

4/10/2020 • 24 minutes to read • Edit Online

by Microsoft

Download Code

> In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).

Building a Contact Management ASP.NET MVC Application (VB)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information - names, phone numbers and email addresses - for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 - Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 - Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 - Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 - Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 - Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 - Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 - Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

In this first iteration, we build the basic application. The goal is to build the Contact Manager in the fastest and simplest way possible. In later iterations, we improve the design of the application.

The Contact Manager application is a basic database-driven application. You can use the application to create new contacts, edit existing contacts, and delete contacts.

In this iteration, we complete the following steps:

1. ASP.NET MVC application
2. Create a database to store our contacts
3. Generate a model class for our database with the Microsoft Entity Framework
4. Create a controller action and view that enables us to list all of the contacts in the database
5. Create controller actions and a view that enables us to create a new contact in the database
6. Create controller actions and a view that enables us to edit an existing contact in the database
7. Create controller actions and a view that enables us to delete an existing contact in the database

Software Prerequisites

In ASP.NET MVC applications, you must have either Visual Studio 2008 or Visual Web Developer 2008 installed on your computer (Visual Web Developer is a free version of Visual Studio that does not include all of the advanced features of Visual Studio). You can download either the trial version of Visual Studio 2008 or Visual Web Developer from the following address:

https://www.asp.net/downloads/essential/

> **NOTE**
> For ASP.NET MVC applications with Visual Web Developer, you must have Visual Web Developer Service Pack 1 installed. Without Service Pack 1, you cannot create Web Application Projects.

ASP.NET MVC framework. You can download the ASP.NET MVC framework from the following address:

https://www.asp.net/mvc

In this tutorial, we use the Microsoft Entity Framework to access a database. The Entity Framework is included with .NET Framework 3.5 Service Pack 1. You can download this service pack from the following location:

https://www.microsoft.com/downloads/details.aspx?familyid=ab99342f-5d1a-413d-8319-81da479ab0d7&displaylang=en

As an alternative to performing each of these downloads one by one, you can take advantage of the Web Platform Installer (Web PI). You can download the Web PI from the following address:

https://www.asp.net/downloads/essential/

ASP.NET MVC Project

ASP.NET MVC Web Application Project. Launch Visual Studio and select the menu option **File, New Project**. The **New Project** dialog appears (see Figure 1). Select the **Web** project type and the **ASP.NET MVC Web Application** template. Name your new project *ContactManager* and click the OK button.

Make sure that you have .NET Framework 3.5 selected from the dropdown list at the top right of the **New Project** dialog. Otherwise, the ASP.NET MVC Web Application template won't appear.

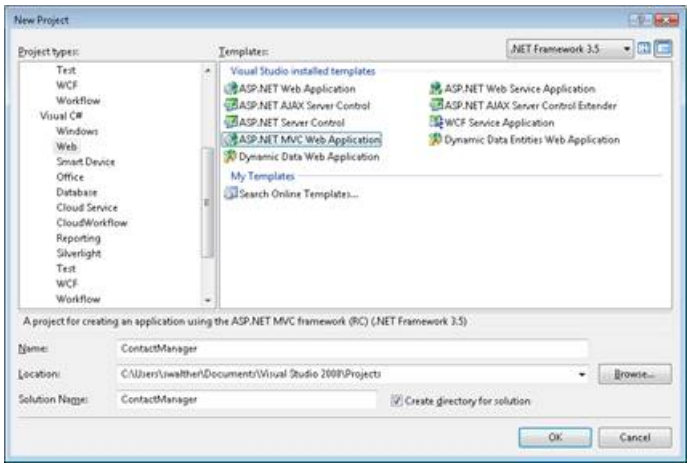


Figure 01: The New Project dialog(Click to view full-size image)

ASP.NET MVC application, the **Create Unit Test Project** dialog appears. You can use this dialog to indicate that you want to create and add a unit test project to your solution when you create your ASP.NET MVC application. Although we won't be building unit tests in this iteration, you should select the option **Yes, create a unit test project** because we plan to add unit tests in a later iteration. Adding a Test project when you first create a new ASP.NET MVC project is much easier than adding a Test project after the ASP.NET MVC project has been created.

> **NOTE**
>
> Because Visual Web Developer does not support Test projects, you do not get the Create Unit Test Project dialog when using Visual Web Developer.

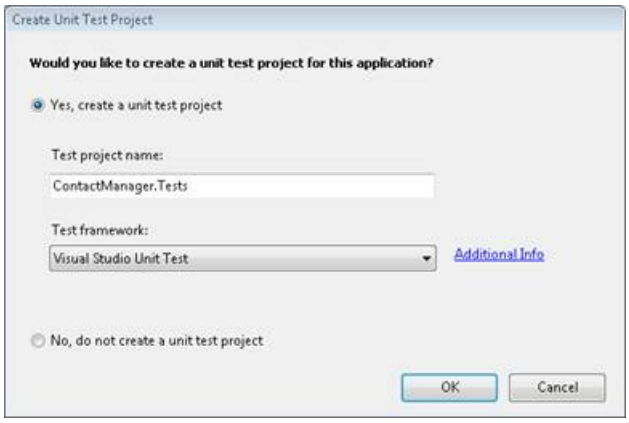


Figure 02: The Create Unit Test Project dialog (Click to view full-size image)

ASP.NET MVC application appears in the Visual Studio Solution Explorer window (see Figure 3). If you don t see the Solution Explorer window then you can open this window by selecting the menu option **View, Solution Explorer**. Notice that the solution contains two projects: the ASP.NET MVC project and the Test project. The ASP.NET MVC project is named ContactManager and the Test project is named ContactManager.Tests.

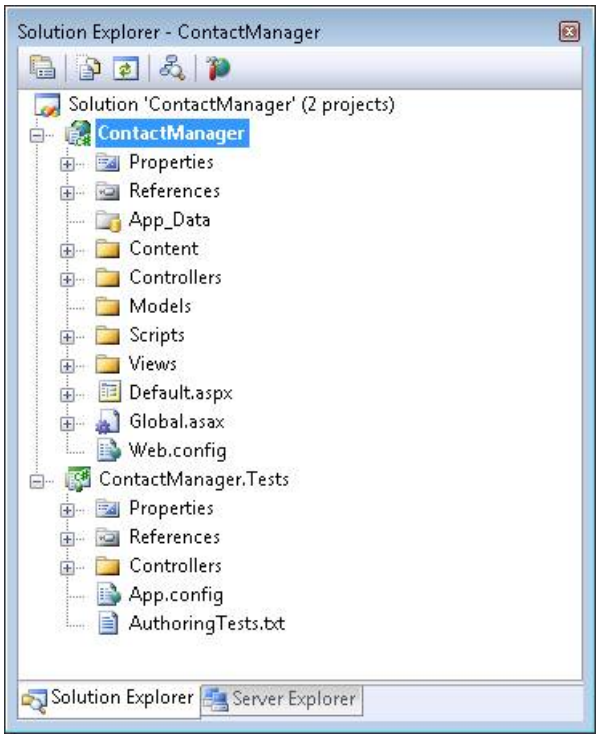


Figure 03: The Solution Explorer window (Click to view full-size image)

Deleting the Project Sample Files

The ASP.NET MVC project template includes sample files for controllers and views. Before creating a new ASP.NET MVC application, you should delete these files. You can delete files and folders in the Solution Explorer window by right-clicking a file or folder and selecting the menu option **Delete**.

You need to delete the following files from the ASP.NET MVC project:

- \Controllers\HomeController.vb
- \Views\Home\About.aspx
- \Views\Home\Index.aspx

And, you need to delete the following file from the Test project:

\Controllers\HomeControllerTest.vb

Creating the Database

The Contact Manager application is a database-driven web application. We use a database to store the contact information.

The ASP.NET MVC framework with any modern database including Microsoft SQL Server, Oracle, MySQL, and IBM DB2 databases. In this tutorial, we use a Microsoft SQL Server database. When you install Visual Studio, you are provided with the option of installing Microsoft SQL Server Express which is a free version of the Microsoft SQL Server database.

Create a new database by right-clicking the App_Data folder in the Solution Explorer window and selecting the menu option **Add, New Item**. In the **Add New Item** dialog, select the **Data** category and the **SQL Server Database** template (see Figure 4). Name the new database ContactManagerDB.mdf and click the OK button.

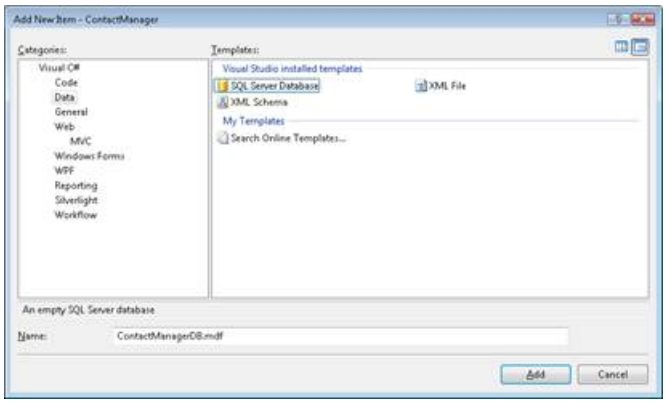


Figure 04: Creating a new Microsoft SQL Server Express database (Click to view full-size image)

After you create the new database, the database appears in the App_Data folder in the Solution Explorer window. Double-click the ContactManager.mdf file to open the Server Explorer window and connect to the database.

> **NOTE**
>
> The Server Explorer window is called the Database Explorer window in the case of Microsoft Visual Web Developer.

You can use the Server Explorer window to create new database objects such as database tables, views, triggers, and stored procedures. Right-click the Tables folder and select the menu option **Add New Table**. The Database Table Designer appears (see Figure 5).

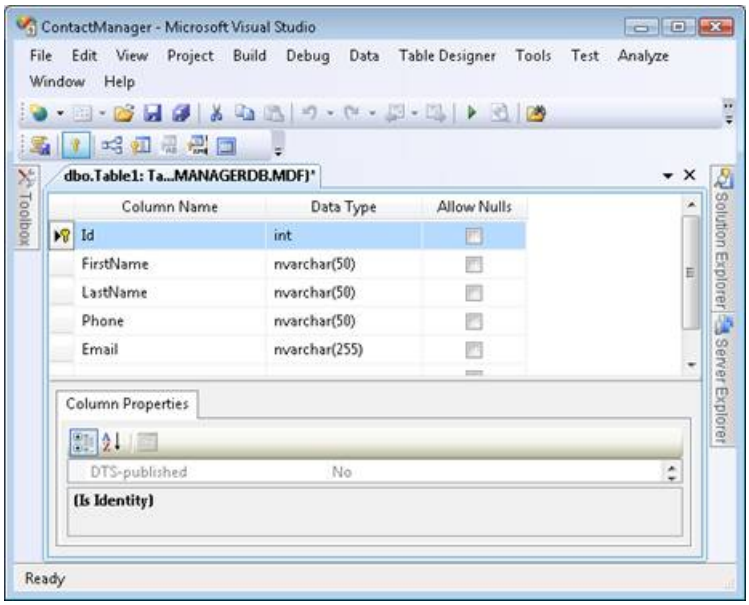


Figure 05: The Database Table Designer (Click to view full-size image)

We need to create a table that contains the following columns:

COLUMN NAME	DATA TYPE	ALLOW NULLS
Id	int	false
FirstName	nvarchar(50)	false
LastName	nvarchar(50)	false
Phone	nvarchar(50)	false

COLUMN NAME	DATA TYPE	ALLOW NULLS
Email	nvarchar(255)	false

The first column, the Id column, is special. You need to mark the Id column as an Identity column and a Primary Key column. You indicate that a column is an Identity column by expanding Column Properties (look at the bottom of Figure 6) and scrolling down to the Identity Specification property. Set the **(Is Identity)** property to the value **Yes**.

You mark a column as a Primary Key column by selecting the column and clicking the button with the icon of a key. After a column is marked as a Primary Key column, an icon of a key appears next to the column (see Figure 6).

After you finish creating the table, click the Save button (the button with an icon of a floppy) to save the new table. Give your new table the name *Contacts*.

After finish creating the Contacts database table, you should add some records to the table. Right-click the Contacts table in the Server Explorer window and select the menu option **Show Table Data**. Enter one or more contacts in the grid that appears.

Creating the Data Model

The ASP.NET MVC application consists of Models, Views, and Controllers. We start by creating a Model class that represents the Contacts table that we created in the previous section.

In this tutorial, we use the Microsoft Entity Framework to generate a model class from the database automatically.

> **NOTE**
> The ASP.NET MVC framework is not tied to the Microsoft Entity Framework in any way. You can use ASP.NET MVC with alternative database access technologies including NHibernate, LINQ to SQL, or ADO.NET.

Follow these steps to create the data model classes:

1. Right-click the Models folder in the Solution Explorer window and select **Add, New Item**. The **Add New Item** dialog appears (see Figure 6).
2. Select the **Data** category and the **ADO.NET Entity Data Model** template. Name your data model *ContactManagerModel.edmx* and click the **Add** button. The Entity Data Model wizard appears (see Figure 7).
3. In the **Choose Model Contents** step, select **Generate from database** (see Figure 7).
4. In the **Choose Your Data Connection** step, select the ContactManagerDB.mdf database and enter the name *ContactManagerDBEntities* for the Entity Connection Settings (see Figure 8).
5. In the **Choose Your Database Objects** step, select the checkbox labeled Tables (see Figure 9). The data model will include all tables contained in your database (there is just one, the Contacts table). Enter the namespace *Models*. Click the Finish button to complete the wizard.

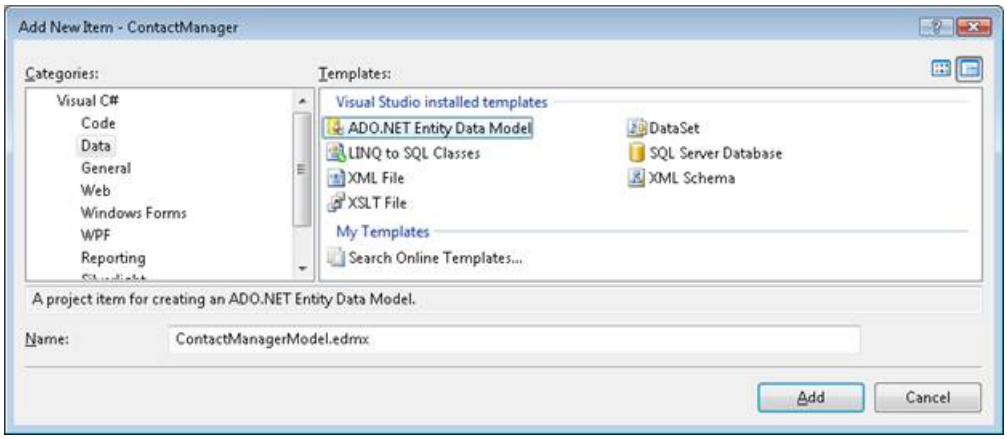


Figure 06: The Add New Item dialog (Click to view full-size image)

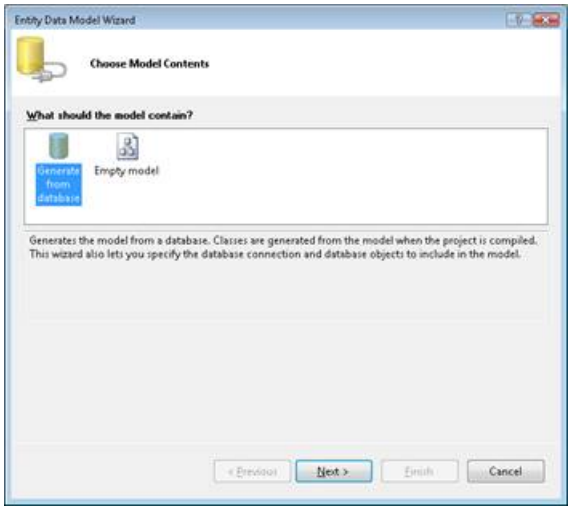


Figure 07: Choose Model Contents (Click to view full-size image)

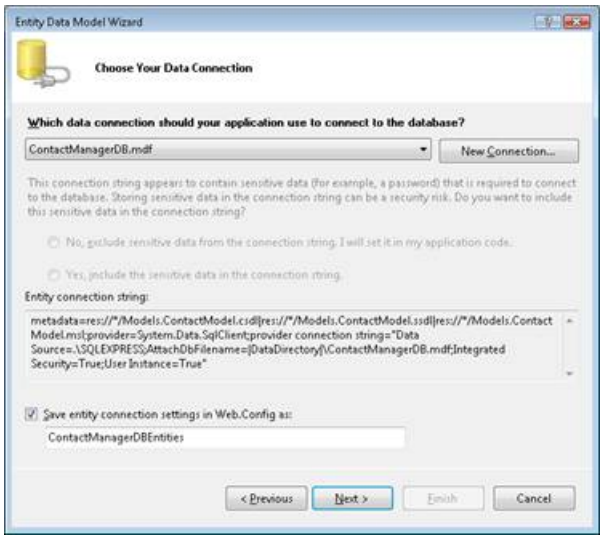


Figure 08: Choose Your Data Connection (Click to view full-size image)

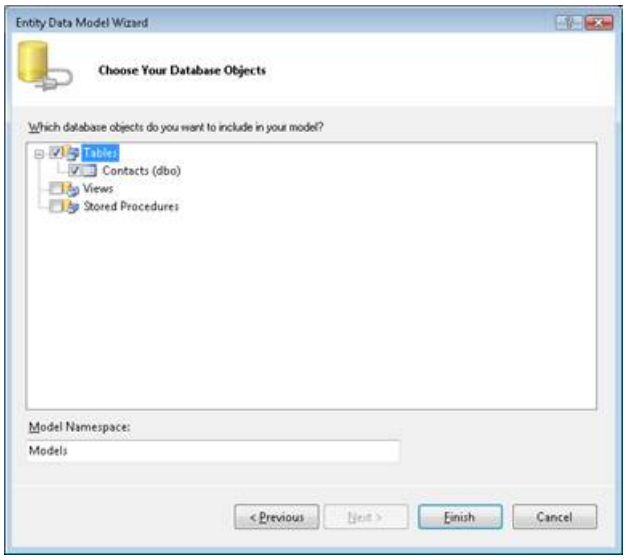


Figure 09: Choose Your Database Objects (Click to view full-size image)

After you complete the Entity Data Model Wizard, the Entity Data Model Designer appears. The designer displays a class that corresponds to each table being modeled. You should see one class named Contacts.

The Entity Data Model wizard generates class names based on database table names. You almost always need to change the name of the class generated by the wizard. Right-click the Contacts class in the designer and select the menu option **Rename**. Change the name of the class from Contacts (plural) to Contact (singular). After you change the class name, the class should appear like Figure 10.

Figure 10: The Contact class (Click to view full-size image)

At this point, we have created our database model. We can use the Contact class to represent a particular contact record in our database.

Creating the Home Controller

The next step is to create our Home controller. The Home controller is the default controller invoked in an ASP.NET MVC application.

Create the Home controller class by right-clicking the Controllers folder in the Solution Explorer window and selecting the menu option **Add, Controller** (see Figure 11). Notice the checkbox labeled **Add action methods for Create, Update, and Details scenarios**. Make sure that this checkbox is checked before clicking the **Add** button.

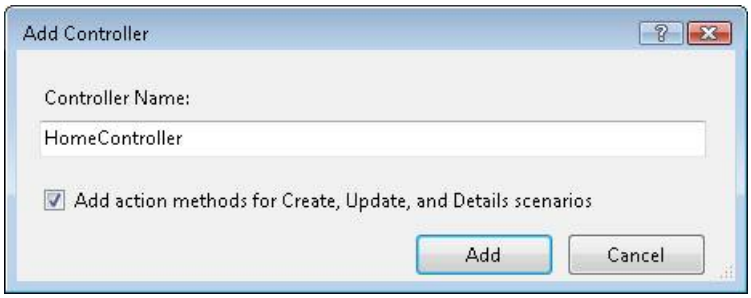


Figure 11: Adding the Home controller (Click to view full-size image)

When you create the Home controller, you get the class in Listing 1.

Listing 1 - Controllers\HomeController.vb

```
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    '
    ' GET: /Home/

    Function Index() As ActionResult
        Return View()
    End Function

    '
    ' GET: /Home/Details/5

    Function Details(ByVal id As Integer) As ActionResult
        Return View()
    End Function

    '
    ' GET: /Home/Create

    Function Create() As ActionResult
        Return View()
    End Function

    '
    ' POST: /Home/Create

    <AcceptVerbs(HttpVerbs.Post)> _
    Function Create(ByVal collection As FormCollection) As ActionResult
        Try
            ' TODO: Add insert logic here
            Return RedirectToAction("Index")
        Catch
            Return View()
        End Try
    End Function

    '
    ' GET: /Home/Edit/5

    Function Edit(ByVal id As Integer) As ActionResult
        Return View()
    End Function

    '
    ' POST: /Home/Edit/5

    <AcceptVerbs(HttpVerbs.Post)> _
    Function Edit(ByVal id As Integer, ByVal collection As FormCollection) As ActionResult
        Try
            ' TODO: Add update logic here

            Return RedirectToAction("Index")
        Catch
            Return View()
        End Try
    End Function
End Class
```

Listing the Contacts

In order to display the records in the Contacts database table, we need to create an Index() action and an Index view.

The Home controller already contains an Index() action. We need to modify this method so that it looks like Listing

2.

Listing 2 - Controllers\HomeController.vb

```
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Private _entities As New ContactManagerDBEntities()

    '
    ' GET: /Home/

    Function Index() As ActionResult
        Return View(_entities.ContactSet.ToList())
    End Function
…
```

Notice that the Home controller class in Listing 2 contains a private field named _entities. The _entities field represents the entities from the data model. We use the _entities field to communicate with the database.

The Index() method returns a view that represents all of the contacts from the Contacts database table. The expression _entities.ContactSet.ToList() returns the list of contacts as a generic list.

Now that we ve created the Index controller, we next need to create the Index view. Before creating the Index view, compile your application by selecting the menu option **Build, Build Solution**. You should always compile your project before adding a view in order for the list of model classes to be displayed in the **Add View** dialog.

You create the Index view by right-clicking the Index() method and selecting the menu option **Add View** (see Figure 12). Selecting this menu option opens the **Add View** dialog (see Figure 13).

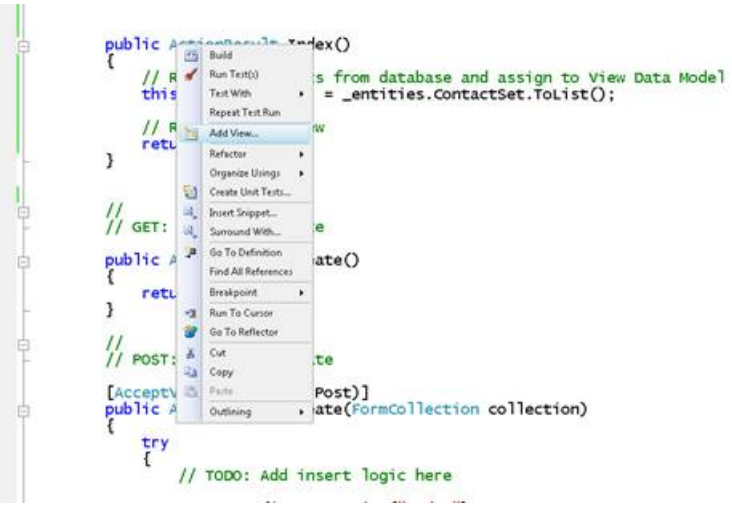


Figure 12: Adding the Index view (Click to view full-size image)

In the **Add View** dialog, check the checkbox labeled **Create a strongly-typed view**. Select the View data class ContactManager.Contact and the View content List. Selecting these options generates a view that displays a list of Contact records.

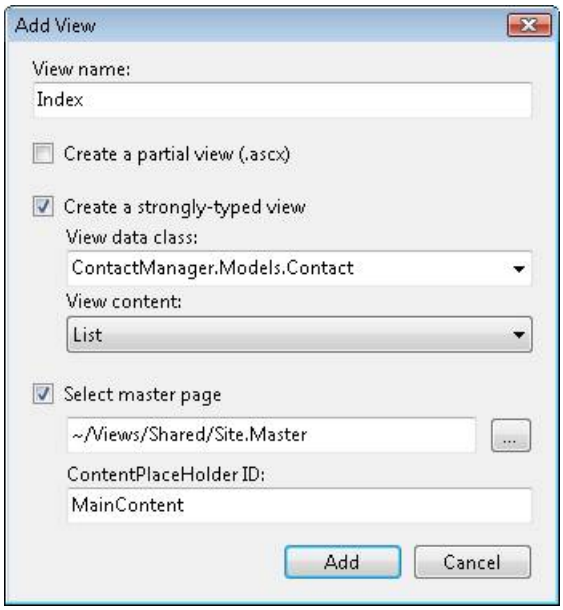


Figure 13: The Add View dialog (Click to view full-size image)

When you click the **Add** button, the Index view in Listing 3 is generated. Notice the <%@ Page %> directive that appears at the top of the file. The Index view inherits from the ViewPage<IEnumerable<ContactManager.Models.Contact>> class. In other words, the Model class in the view represents a list of Contact entities.

The body of the Index view contains a foreach loop that iterates through each of the contacts represented by the Model class. The value of each property of the Contact class is displayed within an HTML table.

Listing 3 - Views\Home\Index.aspx (unmodified)

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of IEnumerable(Of ContactManager.Contact))" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Index</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Index</h2>

    <p>
        <%=Html.ActionLink("Create New", "Create")%>
    </p>

    <table>
        <tr>
            <th></th>
            <th>
                Id
            </th>
            <th>
                FirstName
            </th>
            <th>
                LastName
            </th>
            <th>
                Phone
            </th>
            <th>
                Email
            </th>
        </tr>

    <% For Each item In Model%>

        <tr>
            <td>
                <%=Html.ActionLink("Edit", "Edit", New With {.id = item.Id})%> |
                <%=Html.ActionLink("Details", "Details", New With {.id = item.Id})%>
            </td>
            <td>
                <%=Html.Encode(item.Id)%>
            </td>
            <td>
                <%=Html.Encode(item.FirstName)%>
            </td>
            <td>
                <%=Html.Encode(item.LastName)%>
            </td>
            <td>
                <%=Html.Encode(item.Phone)%>
            </td>
            <td>
                <%=Html.Encode(item.Email)%>
            </td>
        </tr>

    <% Next%>

    </table>

</asp:Content>
```

We need to make one modification to the Index view. Because we are not creating a Details view, we can remove

the Details link. Find and remove the following code from the Index view:

{.id = item.Id})%>

After you modify the Index view, you can run the Contact Manager application. Select the menu option Debug, Start Debugging or simply press F5. The first time you run the application, you get the dialog in Figure 14. Select the option **Modify the Web.config file to enable debugging** and click the OK button.

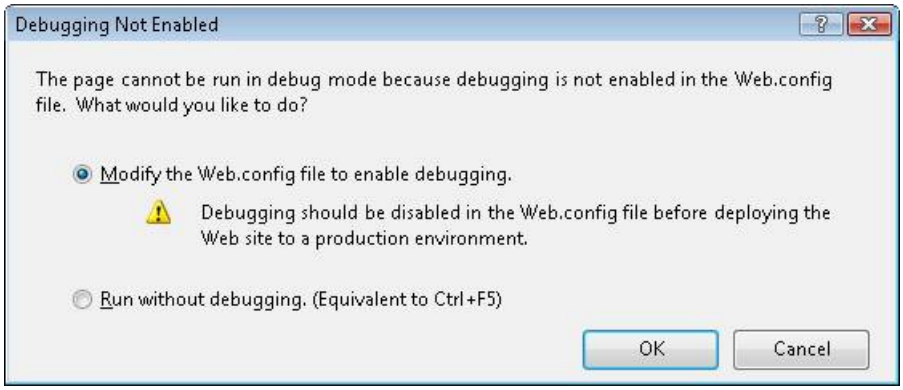


Figure 14: Enabling debugging (Click to view full-size image)

The Index view is returned by default. This view lists all of the data from the Contacts database table (see Figure 15).

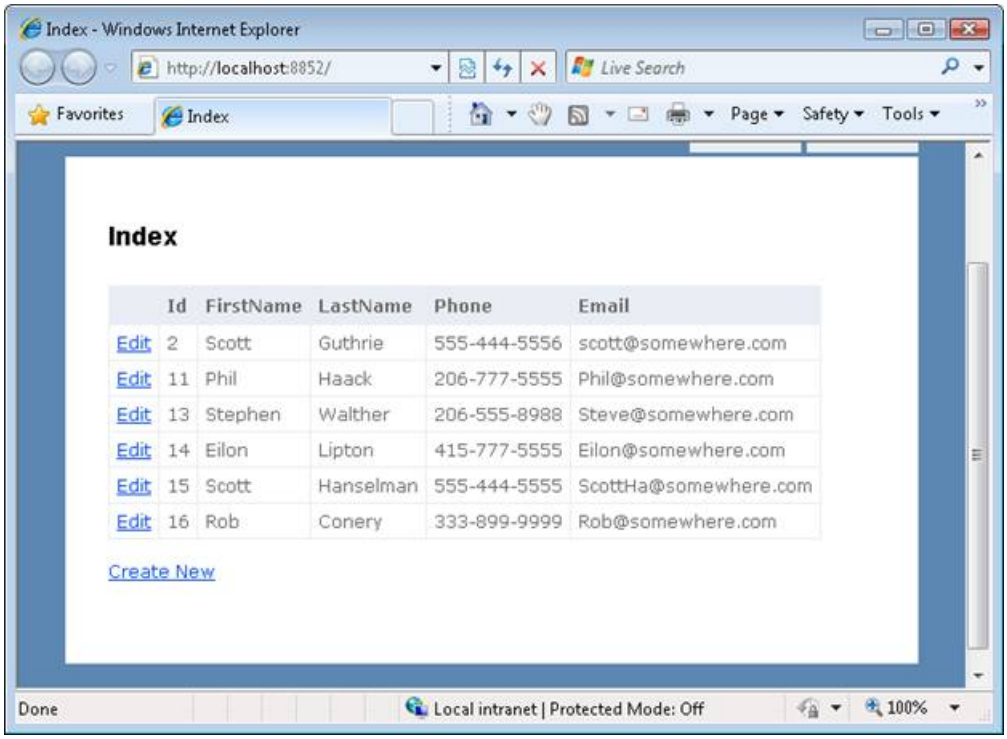


Figure 15: The Index view (Click to view full-size image)

Notice that the Index view includes a link labeled Create New at the bottom of the view. In the next section, you learn how to create new contacts.

Creating New Contacts

To enable users to create new contacts, we need to add two Create() actions to the Home controller. We need to create one Create() action that returns an HTML form for creating a new contact. We need to create a second Create() action that performs the actual database insert of the new contact.

The new Create() methods that we need to add to the Home controller are contained in Listing 4.

Listing 4 - Controllers\HomeController.vb (with Create methods)

```
'
' GET: /Home/Create

Function Create() As ActionResult
    Return View()
End Function

'
' POST: /Home/Create

<AcceptVerbs(HttpVerbs.Post)> _
Function Create(<Bind(Exclude:="Id")> ByVal contactToCreate As Contact) As ActionResult
    If Not ModelState.IsValid Then
        Return View()
    End If

    Try
        _entities.AddToContactSet(contactToCreate)
        _entities.SaveChanges()
        Return RedirectToAction("Index")
    Catch
        Return View()
    End Try
End Function
```

The first Create() method can be invoked with an HTTP GET while the second Create() method can be invoked only by an HTTP POST. In other words, the second Create() method can be invoked only when posting an HTML form. The first Create() method simply returns a view that contains the HTML form for creating a new contact. The second Create() method is much more interesting: it adds the new contact to the database.

Notice that the second Create() method has been modified to accept an instance of the Contact class. The form values posted from the HTML form are bound to this Contact class by the ASP.NET MVC framework automatically. Each form field from the HTML Create form is assigned to a property of the Contact parameter.

Notice that the Contact parameter is decorated with a [Bind] attribute. The [Bind] attribute is used to exclude the Contact Id property from binding. Because the Id property represents an Identity property, we don t want to set the Id property.

In the body of the Create() method, the Entity Framework is used to insert the new Contact into the database. The new Contact is added to the existing set of Contacts and the SaveChanges() method is called to push these changes back to the underlying database.

You can generate an HTML form for creating new Contacts by right-clicking either of the two Create() methods and selecting the menu option **Add View** (see Figure 16).

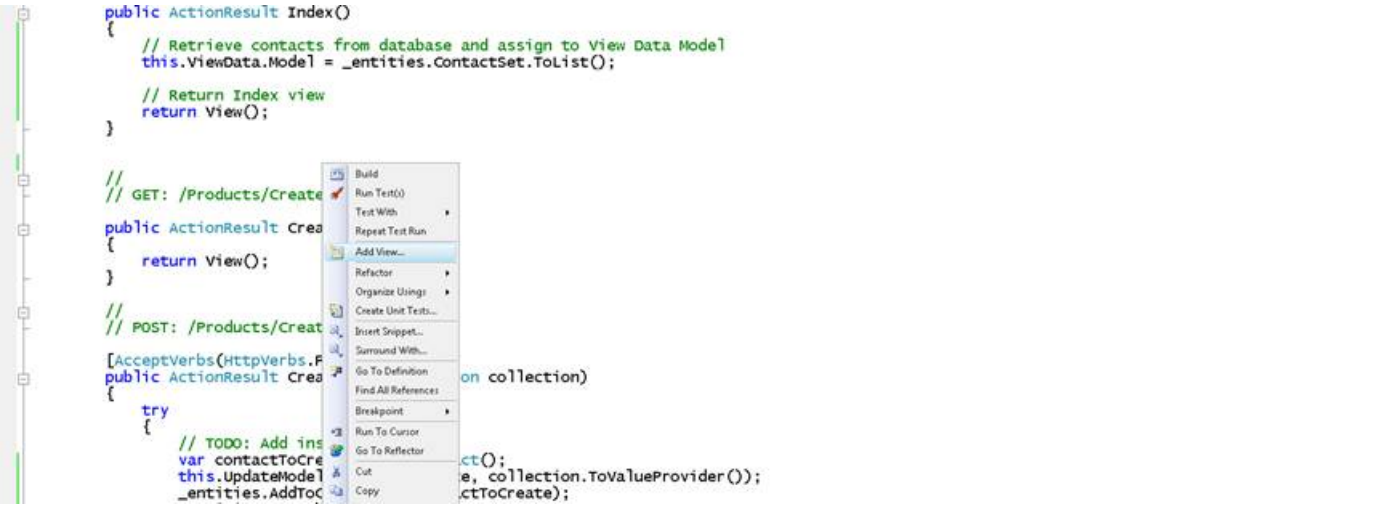


Figure 16: Adding the Create view (Click to view full-size image)

In the **Add View** dialog, select the **ContactManager.Contact** class and the **Create** option for view content (see Figure 17). When you click the **Add** button, a Create view is generated automatically.

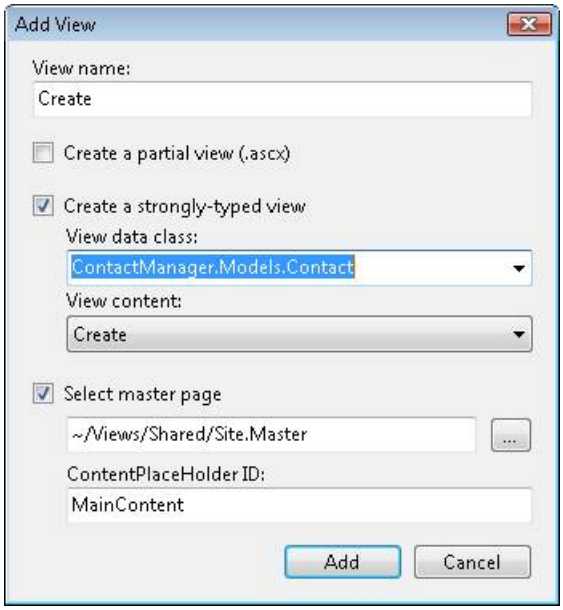


Figure 17: Seeing a page explode (Click to view full-size image)

The Create view contains form fields for each of the properties of the Contact class. The code for the Create view is included in Listing 5.

Listing 5 - Views\Home\Create.aspx

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of ContactManager.Contact)" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Create</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Create</h2>

    <%= Html.ValidationSummary() %>

    <% Using Html.BeginForm()%>

        <fieldset>
            <legend>Fields</legend>
            <p>
                <label for="FirstName">FirstName:</label>
                <%= Html.TextBox("FirstName") %>
                <%= Html.ValidationMessage("FirstName", "*") %>
            </p>
            <p>
                <label for="LastName">LastName:</label>
                <%= Html.TextBox("LastName") %>
                <%= Html.ValidationMessage("LastName", "*") %>
            </p>
            <p>
                <label for="Phone">Phone:</label>
                <%= Html.TextBox("Phone") %>
                <%= Html.ValidationMessage("Phone", "*") %>
            </p>
            <p>
                <label for="Email">Email:</label>
                <%= Html.TextBox("Email") %>
                <%= Html.ValidationMessage("Email", "*") %>
            </p>
            <p>
                <input type="submit" value="Create" />
            </p>
        </fieldset>

    <% End Using %>

    <div>
        <%=Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

After you modify the Create() methods and add the Create view, you can run the Contact Manger application and create new contacts. Click the **Create New** link that appears in the Index view to navigate to the Create view. You should see the view in Figure 18.

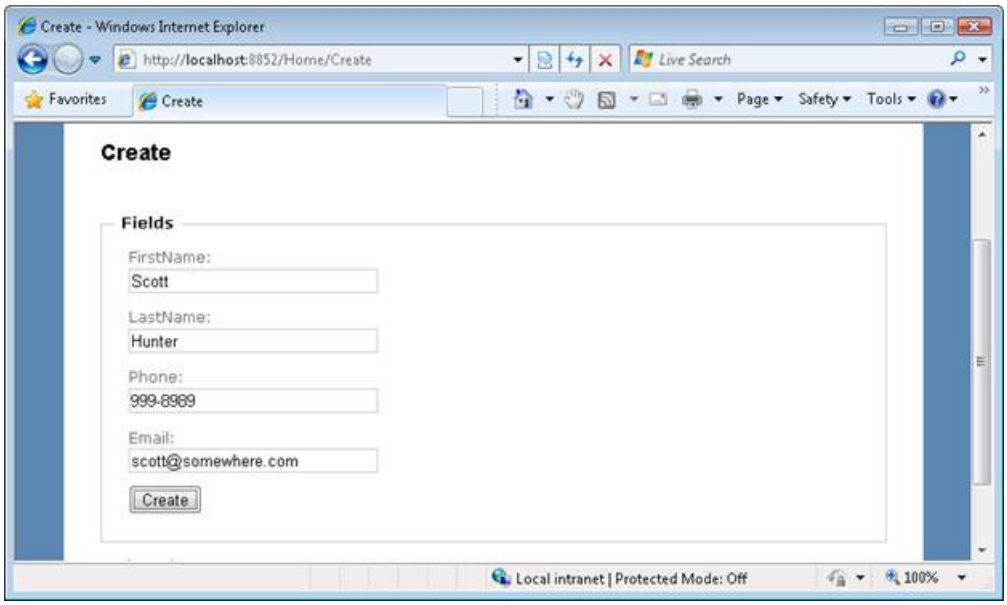


Figure 18: The Create View (Click to view full-size image)

Editing Contacts

Adding the functionality for editing a contact record is very similar to adding the functionality for creating new contact records. First, we need to add two new Edit methods to the Home controller class. These new Edit() methods are contained in Listing 6.

Listing 6 - Controllers\HomeController.vb (with Edit methods)

```
'
' GET: /Home/Edit/5

Function Edit(ByVal id As Integer) As ActionResult
    Dim contactToEdit = (from c in _entities.ContactSet _
                        where c.Id = id _
                        select c).FirstOrDefault()

    Return View(contactToEdit)
End Function

'
' POST: /Home/Edit/5

<AcceptVerbs(HttpVerbs.Post)> _
Function Edit(ByVal contactToEdit As Contact) As ActionResult
    If Not ModelState.IsValid Then
        Return View()
    End If

    Try
        Dim originalContact = (from c in _entities.ContactSet _
                              where c.Id = contactToEdit.Id _
                              select c).FirstOrDefault()
        _entities.ApplyPropertyChanges( originalContact.EntityKey.EntitySetName, contactToEdit)
        _entities.SaveChanges()
        Return RedirectToAction("Index")
    Catch
        Return View()
    End Try
End Function
```

The first Edit() method is invoked by an HTTP GET operation. An Id parameter is passed to this method which represents the Id of the contact record being edited. The Entity Framework is used to retrieve a contact that

matches the Id. A view that contains an HTML form for editing a record is returned.

The second Edit() method performs the actual update to the database. This method accepts an instance of the Contact class as a parameter. The ASP.NET MVC framework binds the form fields from the Edit form to this class automatically. Notice that you don t include the[Bind] attribute when editing a Contact (we need the value of the Id property).

The Entity Framework is used to save the modified Contact to the database. The original Contact must be retrieved from the database first. Next, the Entity Framework ApplyPropertyChanges() method is called to record the changes to the Contact. Finally, the Entity Framework SaveChanges() method is called to persist the changes to the underlying database.

You can generate the view that contains the Edit form by right-clicking the Edit() method and selecting the menu option Add View. In the Add View dialog, select the **ContactManager.Models.Contact** class and the **Edit** view content (see Figure 19).

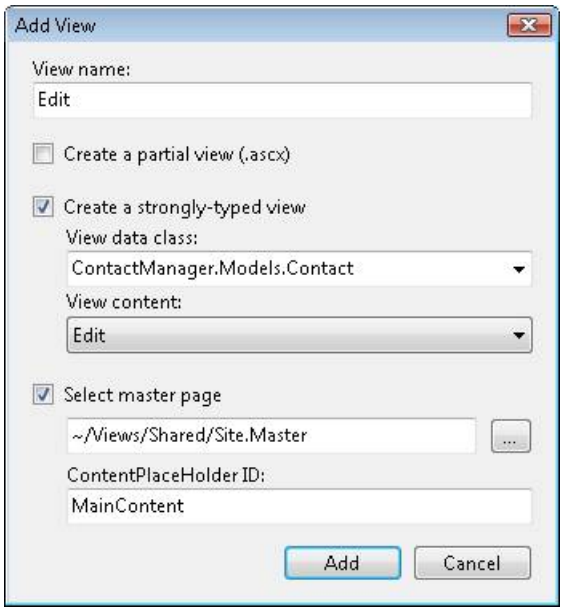


Figure 19: Adding an Edit View (Click to view full-size image)

When you click the Add button, a new Edit view is generated automatically. The HTML form that is generated contains fields that correspond to each of the properties of the Contact class (see Listing 7).

Listing 7 - Views\Home\Edit.aspx

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of ContactManager.Contact)" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Edit</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Edit</h2>

    <%=Html.ValidationSummary()%>

    <% Using Html.BeginForm() %>

        <fieldset>
            <legend>Fields</legend>
            <p>
                <label for="FirstName">FirstName:</label>
                <%= Html.TextBox("FirstName") %>
                <%= Html.ValidationMessage("FirstName", "*") %>
            </p>
            <p>
                <label for="LastName">LastName:</label>
                <%= Html.TextBox("LastName") %>
                <%= Html.ValidationMessage("LastName", "*") %>
            </p>
            <p>
                <label for="Phone">Phone:</label>
                <%= Html.TextBox("Phone") %>
                <%= Html.ValidationMessage("Phone", "*") %>
            </p>
            <p>
                <label for="Email">Email:</label>
                <%= Html.TextBox("Email") %>
                <%= Html.ValidationMessage("Email", "*") %>
            </p>
            <p>
                <input type="submit" value="Save" />
            </p>
        </fieldset>

    <% End Using %>

    <div>
        <%=Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

Deleting Contacts

If you want to delete contacts then you need to add two Delete() actions to the Home controller class. The first Delete() action displays a delete confirmation form. The second Delete() action performs the actual delete.

> **NOTE**
>
> Later, in Iteration #7, we modify the Contact Manager so that it supports a one step Ajax delete.

The two new Delete() methods are contained in Listing 8.

Listing 8 - Controllers\HomeController.vb (Delete methods)

```
'
' GET: /Home/Delete/5

Function Delete(ByVal id As Integer) As ActionResult
    Dim contactToDelete = (from c in _entities.ContactSet _
                        where c.Id = id _
                        select c).FirstOrDefault()

    Return View(contactToDelete)
End Function

'
' POST: /Home/Delete/5

<AcceptVerbs(HttpVerbs.Post)> _
Function Delete(ByVal contactToDelete As Contact) As ActionResult
    Try
        Dim originalContact = (from c in _entities.ContactSet _
                            where c.Id = contactToDelete.Id _
                            select c).FirstOrDefault()
        _entities.DeleteObject(originalContact)
        _entities.SaveChanges()
        Return RedirectToAction("Index")
    Catch
        Return View()
    End Try
End Function
```

The first Delete() method returns a confirmation form for deleting a contact record from the database (see Figure20). The second Delete() method performs the actual delete operation against the database. After the original contact has been retrieved from the database, the Entity Framework DeleteObject() and SaveChanges() methods are called to perform the database delete.

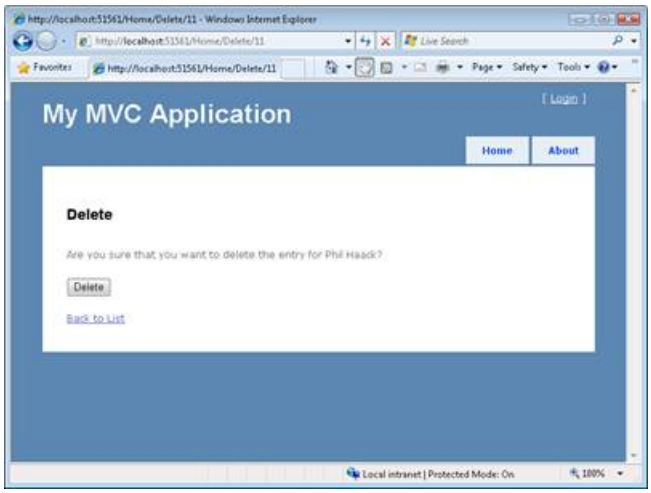


Figure 20: The delete confirmation view (Click to view full-size image)

We need to modify the Index view so that it contains a link for deleting contact records (see Figure 21). You need to add the following code to the same table cell that contains the Edit link:

{.id = item.Id})%>

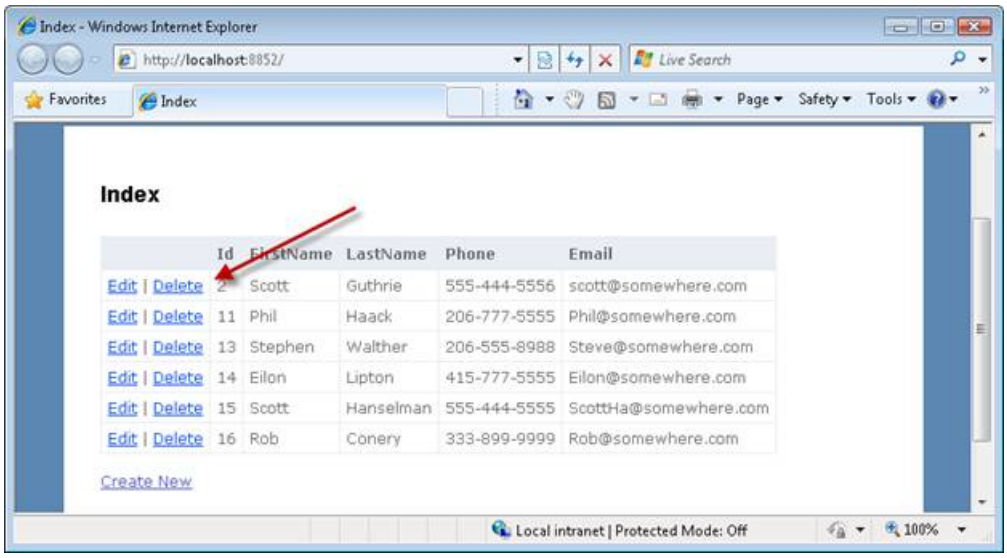


Figure 21: Index view with Edit link (Click to view full-size image)

Next, we need to create the delete confirmation view. Right-click the Delete() method in the Home controller class and select the menu option Add View. The Add View dialog appears (see Figure 22).

Unlike in the case of the List, Create, and Edit views, the Add View dialog does not contain an option to create a Delete view. Instead, select the **ContactManager.Models.Contact** data class and the **Empty** view content. Selecting the Empty view content option will require us to create the view ourselves.

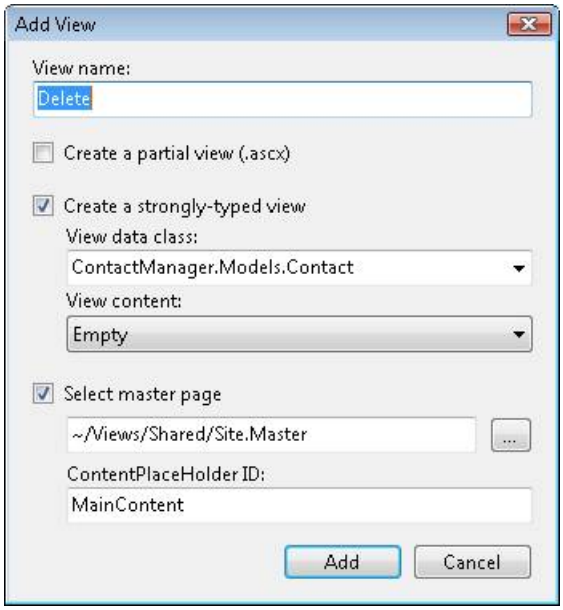


Figure 22: Adding the delete confirmation view (Click to view full-size image)

The content of the Delete view is contained in Listing 9. This view contains a form that confirms whether or not a particular contact should be deleted (see Figure 21).

Listing 9 - Views\Home\Delete.aspx

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of ContactManager.Contact)" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Delete</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Delete</h2>

    <p>
    Are you sure that you want to delete the entry for
    <%= Model.FirstName %> <%= Model.LastName %>?
    </p>

    <% Using Html.BeginForm(New With { .Id = Model.Id }) %>
       <p>
            <input type="submit" value="Delete" />
        </p>
    <% End Using %>

    <div>
        <%=Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

Changing the Name of the Default Controller

It might bother you that the name of our controller class for working with contacts is named the HomeController class. Shouldn't the controller be named ContactController?

This issue is easy enough to fix. First, we need to refactor the name of the Home controller. Open the HomeController class in the Visual Studio Code Editor, right click the name of the class and select the menu option **Rename**. Selecting this menu option opens the Rename dialog.

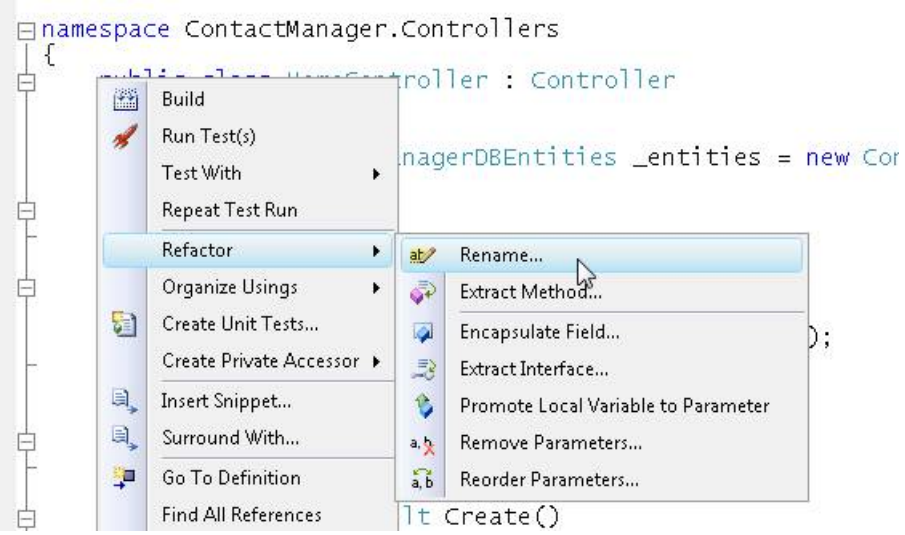


Figure 23: Refactoring a controller name (Click to view full-size image)

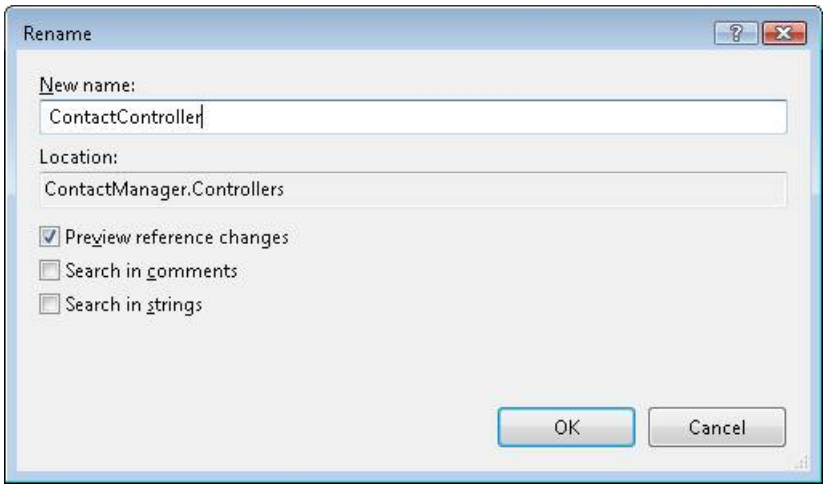


Figure 24: Using the Rename dialog (Click to view full-size image)

If you rename your controller class, Visual Studio will update the name of the folder in the Views folder as well. Visual Studio will rename the \Views\Home folder to the \Views\Contact folder.

After you make this change, your application will no longer have a Home controller. When you run your application, you'll get the error page in Figure 25.

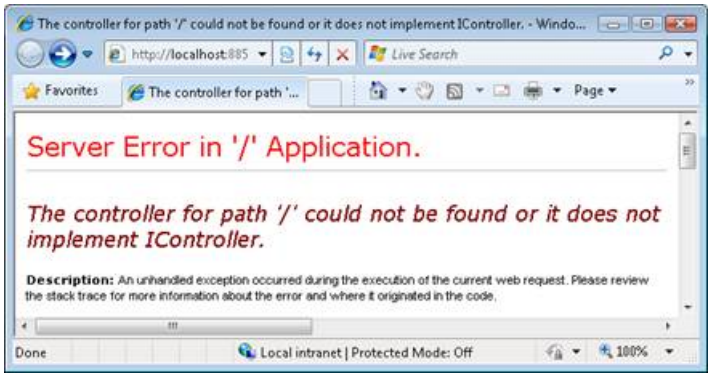


Figure 25: No default controller (Click to view full-size image)

We need to update the default route in the Global.asax file to use the Contact controller instead of the Home controller. Open the Global.asax file and modify the default controller used by the default route (see Listing 10).

Listing 10 - Global.asax.vb

```
Public Class MvcApplication
    Inherits System.Web.HttpApplication

    Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}")

        ' MapRoute takes the following parameters, in order:
        ' (1) Route name
        ' (2) URL with parameters
        ' (3) Parameter defaults
        routes.MapRoute( _
            "Default", _
            "{controller}/{action}/{id}", _
            New With {.controller = "Home", .action = "Index", .id = ""} _
        )

    End Sub

    Sub Application_Start()
        RegisterRoutes(RouteTable.Routes)
    End Sub
End Class
```

After you make these changes, the Contact Manager will run correctly. Now, it will use the Contact controller class as the default controller.

Summary

In this first iteration, we created a basic Contact Manager application in the fastest way possible. We took advantage of Visual Studio to generate the initial code for our controllers and views automatically. We also took advantage of the Entity Framework to generate our database model classes automatically.

Currently, we can list, create, edit, and delete contact records with the Contact Manager application. In other words, we can perform all of the basic database operations required by a database-driven web application.

Unfortunately, our application has some problems. First and I hesitate to admit this, the Contact Manager application is not the most attractive application. It needs some design work. In the next iteration, we'll look at how we can change the default view master page and cascading style sheet to improve the appearance of the application.

Second, we have not implemented any form validation. For example, there is nothing to prevent you from submitting the Create contact form without entering values for any of the form fields. Furthermore, you can enter invalid phone numbers and email addresses. We start to tackle the problem of form validation in iteration #3.

Finally, and most importantly, the current iteration of the Contact Manager application cannot be easily modified or maintained. For example, the database access logic is baked right into the controller actions. This means that we cannot modify our data access code without modifying our controllers. In later iterations, we explore software design patterns that we can implement to make the Contact Manager more resilient to change.

PREVIOUS NEXT

Iteration #2 – Make the application look nice (VB)

4/10/2020 • 8 minutes to read • Edit Online

by Microsoft

Download Code

> In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.

Building a Contact Management ASP.NET MVC Application (VB)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information – names, phone numbers and email addresses – for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 – Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 – Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 – Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 – Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 – Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 – Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 – Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

The goal of this iteration is to improve the appearance of the Contact Manager application. Currently, the Contact Manager uses the default ASP.NET MVC view master page and cascading style sheet (see Figure 1). These don t look bad, but I don t want the Contact Manager to look just like every other ASP.NET MVC website. I want to replace these files with custom files.

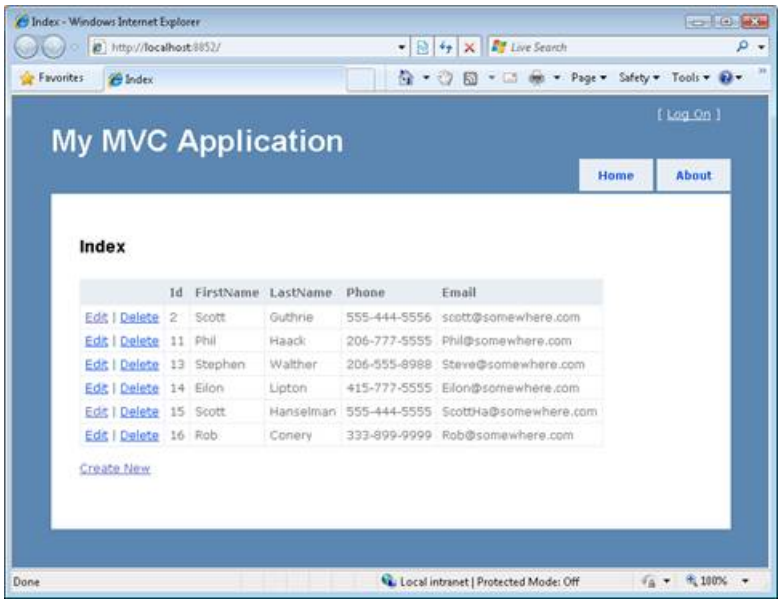


Figure 01: The default appearance of an ASP.NET MVC Application (Click to view full-size image)

In this iteration, I discuss two approaches to improving the visual design of our application. First, I show you how to take advantage of the ASP.NET MVC Design gallery to download a free ASP.NET MVC design template. The ASP.NET MVC Design gallery enables you to create a professional web application without doing any work.

I decided to not use a template from the ASP.NET MVC Design gallery for the Contact Manager application. Instead, I had a custom design created by a professional design firm. In the second part of this tutorial, I explain how I worked with a professional design company to create the final ASP.NET MVC design.

The ASP.NET MVC Design Gallery

The ASP.NET MVC Design Gallery is a free resource provided by Microsoft. The ASP.NET MVC Gallery is located at the following address:

https://www.asp.net/mvc/gallery

The ASP.NET MVC Design Gallery hosts a collection of free website designs that were created specifically for using in an ASP.NET MVC project. Designs are uploaded by members of the community. Visitors to the Gallery can vote for their favorite designs (see Figure 2).

Figure 02: The ASP.NET MVC Design Gallery (Click to view full-size image)

As I write this tutorial, the most popular design in the gallery is a design named October by David Hauser. You can use this design for an ASP.NET MVC project by completing the following steps:

1. Click the **Download** button to download the October.zip file to your computer.
2. Right-click the downloaded October.zip file and click the **Unblock** button (see Figure 3).
3. Unzip the file to a folder named October.
4. Select all of the files from the DesignTemplate folder contained in the October folder, right-click the files, and select the menu option **Copy**.

5. Right-click the ContactManager project node in the Visual Studio Solution Explorer window and select the menu option **Paste** (see Figure 4).
6. Select the Visual Studio menu option **Edit, Find and Replace, Quick Replace** and replace *[MyProjectName]* with *ContactManager* (see Figure 5).

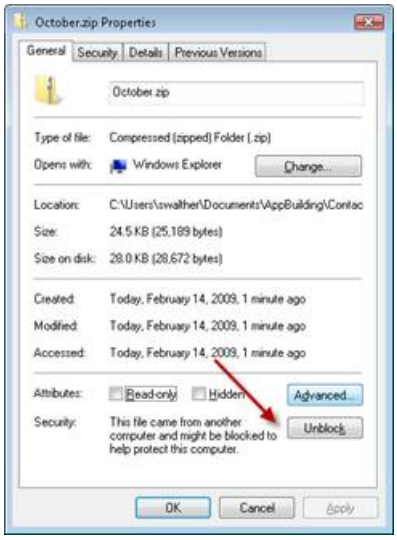


Figure 03: Unblocking a file downloaded from the web (Click to view full-size image)

Figure 04: Overwriting files in the Solution Explorer (Click to view full-size image)

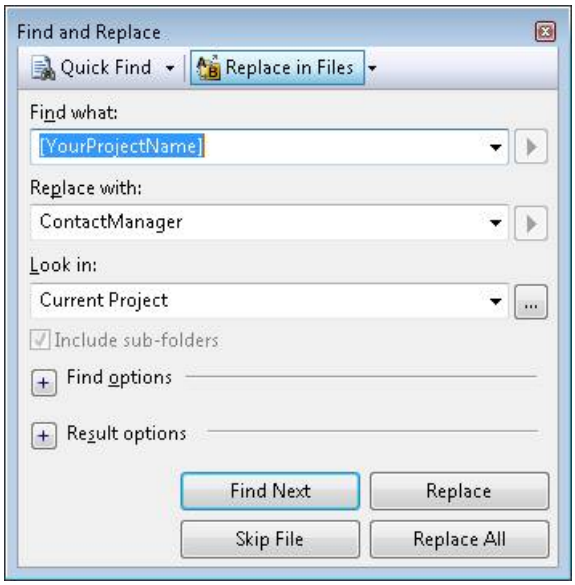


Figure 05: Replacing [ProjectName] with ContactManager (Click to view full-size image)

After you complete these steps, your web application will use the new design. The page in Figure 6 illustrates the appearance of the Contact Manager application with the October design.

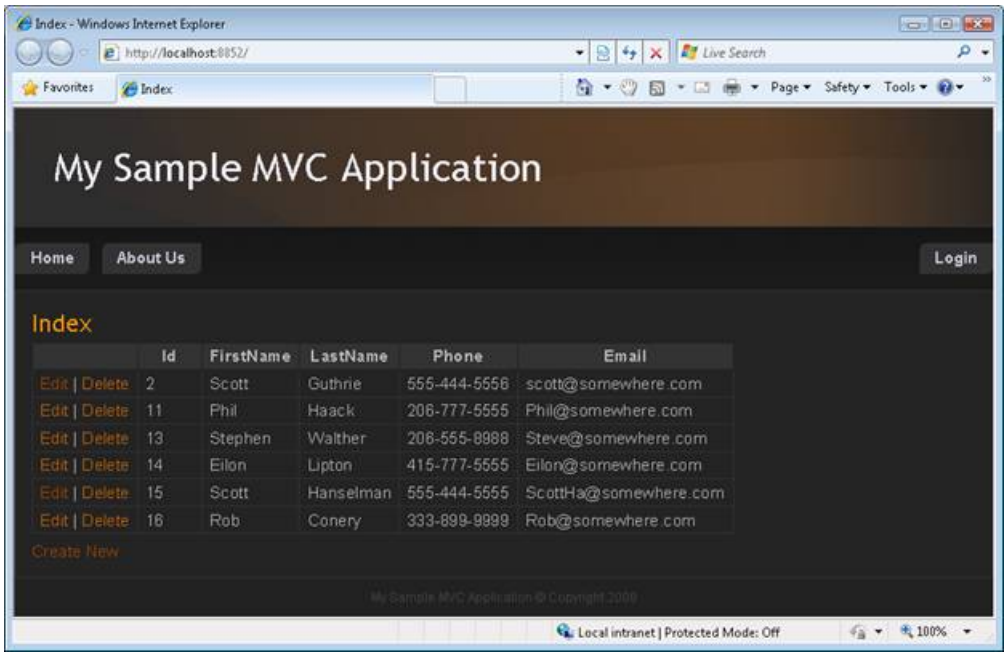


Figure 06: ContactManager with the October template (Click to view full-size image)

Creating a Custom ASP.NET MVC Design

The ASP.NET MVC Design Gallery has a good selection of different design styles. The Gallery provides you with a painless way to customize the appearance of your ASP.NET MVC applications. And, of course, the Gallery has the big advantage of being completely free.

However, you might need to create a completely unique design for your website. In that case, it makes sense to work with a website design company. I decided to take this approach for the design for the Contact Manager application.

I zipped up the Contact Manager from Iteration #1 and sent the project to the design company. They did not own Visual Studio (shame on them!), but that didn't present a problem. They were able to download Microsoft Visual Web Developer for free from the https://www.asp.net website and open the Contact Manager application in Visual Web Developer. In a couple of days, they had produced the design in Figure 7.

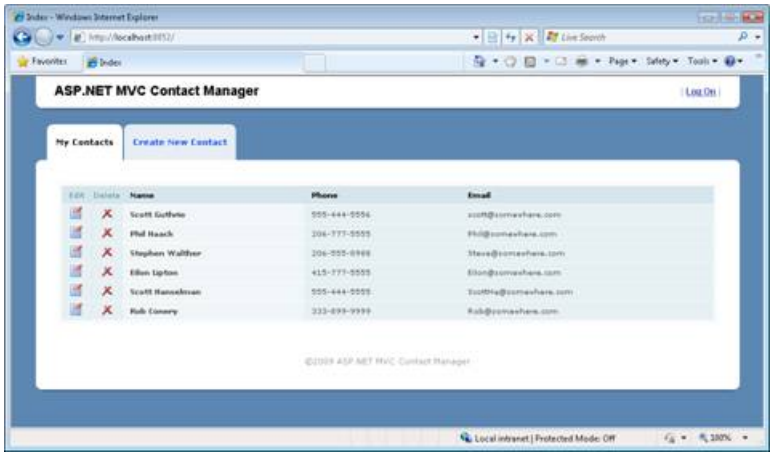


Figure 07: The ASP.NET MVC Contact Manager Design (Click to view full-size image)

The new design consisted of two main files: a new cascading style sheet file and a new view master page file. A view master page contains the layout and shared content for views in an ASP.NET MVC application. For example, the view master page includes the header, navigation tabs, and footer that appear in Figure 7. I overwrote the existing Site.Master view master page in the Views\Shared folder with the new Site.Master file from the design company,

The design company also created a new cascading style sheet and set of images. I placed these new files in the Content folder and overwrote the existing Site.css file. You should place all static content in the Content folder.

Notice that the new design for the Contact Manager includes images for editing and deleting contacts. An Edit and Delete image appear next to each contact in the HTML table of contacts.

Originally, these links that were rendered with the HTML.ActionLink() helper like this:

```
<td>
    <%= Html.ActionLink("Edit", "Edit", new { id=item.Id  }) %> |
    <%= Html.ActionLink("Delete", "Delete", new { id=item.Id  }) %>
</td>
```

The Html.ActionLink() method does not support images (the method HTML encodes the link text for security reasons). Therefore, I replaced the calls to Html.ActionLink() with calls to Url.Action() like this:

```
<td class="actions edit">
    <a href='<%= Url.Action("Edit", new {id=item.Id}) %>'><img src="../../Content/Edit.png" alt="Edit" /></a>
</td>
<td class="actions delete">
    <a href='<%= Url.Action("Delete", new {id=item.Id}) %>'><img src="../../Content/Delete.png" alt="Delete"
/></a>
</td>
```

The Html.ActionLink() method renders an entire HTML hyperlink. The Url.Action() method, on the other hand, renders just the URL without the <a> tag.

Notice, furthermore, that the new design includes both selected and unselected tabs. For example, in Figure 8, the **Create New Contact** tab is selected and the **My Contacts** tab is not selected.

Figure 08: Selected and unselected tabs(Click to view full-size image)

To support rendering both selected and unselected tabs, I created a custom HTML helper named the MenuItemHelper. This helper method renders either a <li> tag or a <li class="selected"> tag depending on whether the current controller and action corresponds to the controller and action name passed to the helper. The code for the MenuItemHelper is contained in Listing 1.

Listing 1 – Helpers\MenuItemHelper.vb

```
Public Module MenuItemHelper

   <System.Runtime.CompilerServices.Extension> _
   Function MenuItem(ByVal helper As HtmlHelper, ByVal linkText As String, ByVal actionName As String, ByVal
controllerName As String) As String
        Dim currentControllerName As String = helper.ViewContext.RouteData.Values("controller")
        Dim currentActionName As String = helper.ViewContext.RouteData.Values("action")

        Dim builder = New TagBuilder("li")

        ' Add selected class
        If currentControllerName.Equals(controllerName, StringComparison.CurrentCultureIgnoreCase) AndAlso
currentActionName.Equals(actionName, StringComparison.CurrentCultureIgnoreCase) Then
            builder.AddCssClass("selected")
        End If

        ' Add link
        builder.InnerHtml = helper.ActionLink(linkText, actionName, controllerName)

        ' Render Tag Builder
        Return builder.ToString(TagRenderMode.Normal)
   End Function

End Module
```

The MenuItemHelper uses the TagBuilder class internally to build the <li> HTML tag. The TagBuilder class is a very useful utility class that you can use whenever you need to build up a new HTML tag. It includes methods for adding attributes, adding CSS classes, generating Ids, and modifying the tag s inner HTML.

Summary

In this iteration, we improved the visual design of our ASP.NET MVC application. First, you were introduced to the ASP.NET MVC Design Gallery. You learned how to download free design templates from the ASP.NET MVC Design Gallery that you can use in your ASP.NET MVC applications.

Next, we discussed how you can create a custom design by modifying the default cascading style sheet file and master view page file. In order to support the new design, we had to make some minor changes to our Contact Manager application. For example, we added a new HTML helper named the MenuItemHelper that displays selected and unselected tabs.

In the next iteration, we tackle the very important subject of validation. We add validation code to our application so that a user cannot create a new contact without supplying required values such as a person s first and last name.

PREVIOUS NEXT

Iteration #3 – Add form validation (VB)

4/10/2020 • 8 minutes to read • Edit Online

by Microsoft

Download Code

> In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.

Building a Contact Management ASP.NET MVC Application (VB)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information - names, phone numbers and email addresses - for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 - Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 - Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 - Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 - Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 - Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 - Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 - Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

In this second iteration of the Contact Manager application, we add basic form validation. We prevent people from submitting a contact without entering values for required form fields. We also validate phone numbers and email addresses (see Figure 1).

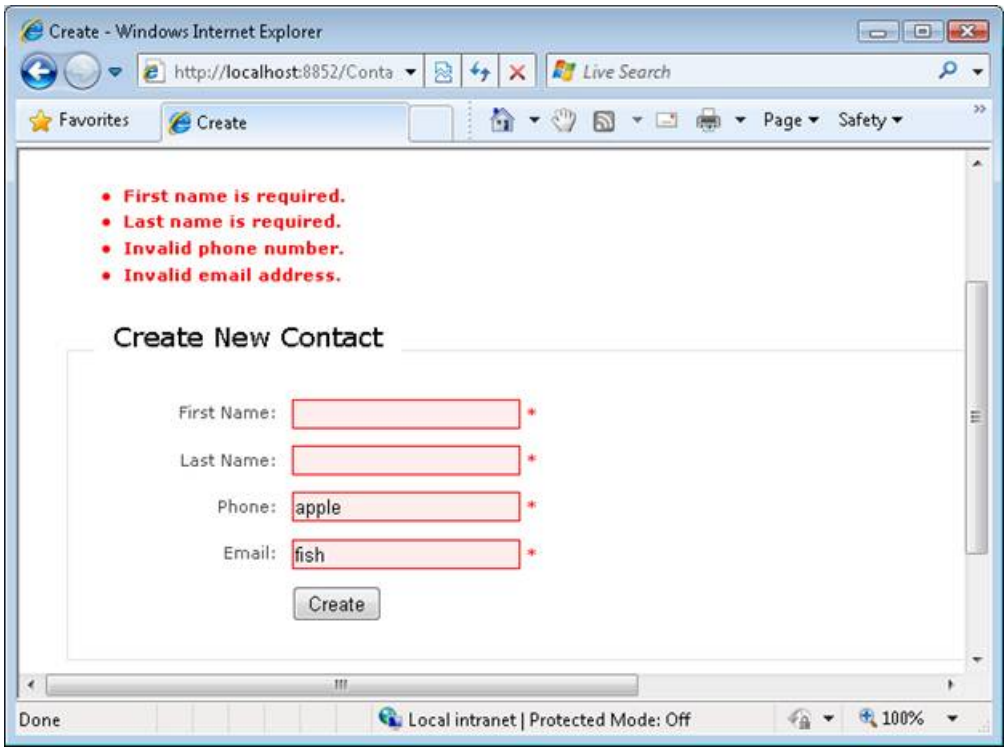


Figure 01: A form with validation (Click to view full-size image)

In this iteration, we add the validation logic directly to the controller actions. In general, this is not the recommended way to add validation to an ASP.NET MVC application. A better approach is to place an application s validation logic in a separate service layer. In the next iteration, we refactor the Contact Manager application to make the application more maintainable.

In this iteration, to keep things simple, we write all of the validation code by hand. Instead of writing the validation code ourselves, we could take advantage of a validation framework. For example, you can use the Microsoft Enterprise Library Validation Application Block (VAB) to implement the validation logic for your ASP.NET MVC application. To learn more about the Validation Application Block, see:

http://msdn.microsoft.com/library/dd203099.aspx

Adding Validation to the Create View

Let s start by adding validation logic to the Create view. Fortunately, because we generated the Create view with Visual Studio, the Create view already contains all of the necessary user interface logic to display validation messages. The Create view is contained in Listing 1.

Listing 1 - \Views\Contact\Create.aspx

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of ContactManager.Contact)" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Create</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <%= Html.ValidationSummary() %>

    <% Using Html.BeginForm()%>

        <fieldset>
            <legend>Create New Contact</legend>
            <p>
                <label for="FirstName">First Name:</label>
                <%= Html.TextBox("FirstName") %>
                <%= Html.ValidationMessage("FirstName", "*") %>
            </p>
            <p>
                <label for="LastName">Last Name:</label>
                <%= Html.TextBox("LastName") %>
                <%= Html.ValidationMessage("LastName", "*") %>
            </p>
            <p>
                <label for="Phone">Phone:</label>
                <%= Html.TextBox("Phone") %>
                <%= Html.ValidationMessage("Phone", "*") %>
            </p>
            <p>
                <label for="Email">Email:</label>
                <%= Html.TextBox("Email") %>
                <%= Html.ValidationMessage("Email", "*") %>
            </p>
            <p class="submit">
                <input type="submit" value="Create" />
            </p>
        </fieldset>

    <% End Using %>

</asp:Content>
```

The field-validation-error class is used to style the output rendered by the Html.ValidationMessage() helper. The input-validation-error class is used to style the textbox (input) rendered by the Html.TextBox() helper. The validation-summary-errors class is used to style the unordered list rendered by the Html.ValidationSummary() helper.

> **NOTE**
> You can modify the style sheet classes described in this section to customize the appearance of validation error messages.

Adding Validation Logic to the Create Action

Right now, the Create view never displays validation error messages because we have not written the logic to generate any messages. In order to display validation error messages, you need to add the error messages to ModelState.

> **NOTE**
> The UpdateModel() method adds error messages to ModelState automatically when there is an error assigning the value of a form field to a property. For example, if you attempt to assign the string "apple" to a BirthDate property that accepts DateTime values, then the UpdateModel() method adds an error to ModelState.

The modified Create() method in Listing 2 contains a new section that validates the properties of the Contact class before the new contact is inserted into the database.

Listing 2 - Controllers\ContactController.vb (Create with validation)

```
<AcceptVerbs(HttpVerbs.Post)> _
Function Create(<Bind(Exclude:="Id")> ByVal contactToCreate As Contact) As ActionResult
    ' Validation logic
    If contactToCreate.FirstName.Trim().Length = 0 Then
        ModelState.AddModelError("FirstName", "First name is required.")
    End If
    If contactToCreate.LastName.Trim().Length = 0 Then
        ModelState.AddModelError("LastName", "Last name is required.")
    End If
    If (contactToCreate.Phone.Length > 0 AndAlso Not Regex.IsMatch(contactToCreate.Phone, "((\(\d{3}\) ?)|
(\d{3}-))?\d{3}-\d{4}"))
        ModelState.AddModelError("Phone", "Invalid phone number.")
    End If
    If (contactToCreate.Email.Length > 0 AndAlso  Not Regex.IsMatch(contactToCreate.Email, "^[\w-\.]+@([\w-
]+\.)+[\w-]{2,4}$"))
        ModelState.AddModelError("Email", "Invalid email address.")
    End If
    If Not ModelState.IsValid Then
        Return View()
    End If

    ' Database logic
    Try
        _entities.AddToContactSet(contactToCreate)
        _entities.SaveChanges()
        Return RedirectToAction("Index")
    Catch
        Return View()
    End Try
End Function
```

The validate section enforces four distinct validation rules:

- The FirstName property must have a length greater than zero (and it cannot consist solely of spaces)
- The LastName property must have a length greater than zero (and it cannot consist solely of spaces)
- If the Phone property has a value (has a length greater than 0) then the Phone property must match a regular expression.
- If the Email property has a value (has a length greater than 0) then the Email property must match a regular expression.

When there is a validation rule violation, an error message is added to ModelState with the help of the AddModelError() method. When you add a message to ModelState, you provide the name of a property and the text of a validation error message. This error message is displayed in the view by the Html.ValidationSummary() and Html.ValidationMessage() helper methods.

After the validation rules are executed, the IsValid property of ModelState is checked. The IsValid property returns false when any validation error messages have been added to ModelState. If validation fails, the Create form is redisplayed with the error messages.

NOTE

I got the regular expressions for validating the phone number and email address from the regular expression repository at *http://regexlib.com*

Adding Validation Logic to the Edit Action

The Edit() action updates a Contact. The Edit() action needs to perform exactly the same validation as the Create() action. Instead of duplicating the same validation code, we should refactor the Contact controller so that both the Create() and Edit() actions call the same validation method.

The modified Contact controller class is contained in Listing 3. This class has a new ValidateContact() method that is called within both the Create() and Edit() actions.

Listing 3 - Controllers\ContactController.vb

```
Public Class ContactController
    Inherits System.Web.Mvc.Controller

    Private _entities As New ContactManagerDBEntities()

    Protected Sub ValidateContact(contactToValidate As Contact)
        If contactToValidate.FirstName.Trim().Length = 0 Then
            ModelState.AddModelError("FirstName", "First name is required.")
        End If
        If contactToValidate.LastName.Trim().Length = 0 Then
            ModelState.AddModelError("LastName", "Last name is required.")
        End If
        If (contactToValidate.Phone.Length > 0 AndAlso Not Regex.IsMatch(contactToValidate.Phone, "((\(\d{3}\)
?)|(\d{3}-))?\d{3}-\d{4}"))
            ModelState.AddModelError("Phone", "Invalid phone number.")
        End If
        If (contactToValidate.Email.Length > 0 AndAlso  Not Regex.IsMatch(contactToValidate.Email, "^[\w-
\.]+@([\w-]+\.)+[\w-]{2,4}$"))
            ModelState.AddModelError("Email", "Invalid email address.")
        End If
    End Sub

    '
    ' GET: /Contact

    Function Index() As ActionResult
        Return View(_entities.ContactSet.ToList())
    End Function

    '
    ' GET: /Contact/Create

    Function Create() As ActionResult
        Return View()
    End Function

    '
    ' POST: /Contact/Create

    <AcceptVerbs(HttpVerbs.Post)> _
    Function Create(<Bind(Exclude:="Id")> ByVal contactToCreate As Contact) As ActionResult
        ' Validation logic
        ValidateContact(contactToCreate)
        If Not ModelState.IsValid Then
            Return View()
        End If

        ' Database logic
```

```
        ' Database logic
        Try
            _entities.AddToContactSet(contactToCreate)
            _entities.SaveChanges()
            Return RedirectToAction("Index")
        Catch
            Return View()
        End Try
    End Function

    '
    ' GET: /Contact/Edit/5

    Function Edit(ByVal id As Integer) As ActionResult
        Dim contactToEdit = (from c in _entities.ContactSet _
                            where c.Id = id _
                            select c).FirstOrDefault()

        Return View(contactToEdit)
    End Function

    '
    ' POST: /Contact/Edit/5

    <AcceptVerbs(HttpVerbs.Post)> _
    Function Edit(ByVal contactToEdit As Contact) As ActionResult
        ' Validation logic
        ValidateContact(contactToEdit)
        If Not ModelState.IsValid Then
            Return View()
        End If

        ' Database logic
        Try
            Dim originalContact = (from c in _entities.ContactSet _
                                  where c.Id = contactToEdit.Id _
                                  select c).FirstOrDefault()
            _entities.ApplyPropertyChanges(originalContact.EntityKey.EntitySetName, contactToEdit)
            _entities.SaveChanges()
            Return RedirectToAction("Index")
        Catch
            Return View()
        End Try
    End Function

    '
    ' GET: /Contact/Delete/5

    Function Delete(ByVal id As Integer) As ActionResult
        Dim contactToDelete = (from c in _entities.ContactSet _
                            where c.Id = id _
                            select c).FirstOrDefault()

        Return View(contactToDelete)
    End Function

    '
    ' POST: /Contact/Delete/5

    <AcceptVerbs(HttpVerbs.Post)> _
    Function Delete(ByVal contactToDelete As Contact) As ActionResult
        Try
            Dim originalContact = (from c in _entities.ContactSet _
                                  where c.Id = contactToDelete.Id _
                                  select c).FirstOrDefault()
            _entities.DeleteObject(originalContact)
            _entities.SaveChanges()
            Return RedirectToAction("Index")
        Catch
            Return View()
```

```
            Return View()
        End Try
    End Function

End Class
```

Summary

In this iteration, we added basic form validation to our Contact Manager application. Our validation logic prevents users from submitting a new contact or editing an existing contact without supplying values for the FirstName and LastName properties. Furthermore, users must supply valid phone numbers and email addresses.

In this iteration, we added the validation logic to our Contact Manager application in the easiest way possible. However, mixing our validation logic into our controller logic will create problems for us in the long term. Our application will be more difficult to maintain and modify over time.

In the next iteration, we will refactor our validation logic and database access logic out of our controllers. We'll take advantage of several software design principles to enable us to create a more loosely coupled, and more maintainable, application.

PREVIOUS NEXT

Iteration #4 – Make the application loosely coupled (VB)

4/10/2020 • 15 minutes to read • Edit Online

by Microsoft

Download Code

> In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.

Building a Contact Management ASP.NET MVC Application (VB)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information - names, phone numbers and email addresses - for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 - Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 - Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 - Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 - Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 - Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 - Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 - Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

In this fourth iteration of the Contact Manager application, we refactor the application to make the application more loosely coupled. When an application is loosely coupled, you can modify the code in one part of the application without needing to modify the code in other parts of the application. Loosely coupled applications are more resilient to change.

Currently, all of the data access and validation logic used by the Contact Manager application is contained in the controller classes. This is a bad idea. Whenever you need to modify one part of your application, you risk introducing bugs into another part of your application. For example, if you modify your validation logic, you risk introducing new bugs into your data access or controller logic.

> **NOTE**
> (SRP), a class should never have more than one reason to change. Mixing controller, validation, and database logic is a massive violation of the Single Responsibility Principle.

There are several reasons that you might need to modify your application. You might need to add a new feature to your application, you might need to fix a bug in your application, or you might need to modify how a feature of your application is implemented. Applications are rarely static. They tend to grow and mutate over time.

Imagine, for example, that you decide to change how you implement your data access layer. Right now, the Contact Manager application uses the Microsoft Entity Framework to access the database. However, you might decide to migrate to a new or alternative data access technology such as ADO.NET Data Services or NHibernate. However, because the data access code is not isolated from the validation and controller code, there is no way to modify the data access code in your application without modifying other code that is not directly related to data access.

When an application is loosely coupled, on the other hand, you can make changes to one part of an application without touching other parts of an application. For example, you can switch data access technologies without modifying your validation or controller logic.

In this iteration, we take advantage of several software design patterns that enable us to refactor our Contact Manager application into a more loosely coupled application. When we are done, the Contact Manager won t do anything that it didn't do before. However, we'll be able to change the application more easily in the future.

> **NOTE**
> Refactoring is the process of rewriting an application in such a way that it does not lose any existing functionality.

Using the Repository Software Design Pattern

Our first change is to take advantage of a software design pattern called the Repository pattern. We'll use the Repository pattern to isolate our data access code from the rest of our application.

Implementing the Repository pattern requires us to complete the following two steps:

1. Create an interface
2. Create a concrete class that implements the interface

First, we need to create an interface that describes all of the data access methods that we need to perform. The IContactManagerRepository interface is contained in Listing 1. This interface describes five methods: CreateContact(), DeleteContact(), EditContact(), GetContact, and ListContacts().

Listing 1 - Models\IContactManagerRepository.vb

```
Public Interface IContactManagerRepository
Function CreateContact(ByVal contactToCreate As Contact) As Contact
Sub DeleteContact(ByVal contactToDelete As Contact)
Function EditContact(ByVal contactToUpdate As Contact) As Contact
Function GetContact(ByVal id As Integer) As Contact
Function ListContacts() As IEnumerable(Of Contact)
End Interface
```

Next, we need to create a concrete class that implements the IContactManagerRepository interface. Because we are using the Microsoft Entity Framework to access the database, we'll create a new class named EntityContactManagerRepository. This class is contained in Listing 2.

Listing 2 - Models\EntityContactManagerRepository.vb

```
Public Class EntityContactManagerRepository
Implements IContactManagerRepository

Private _entities As New ContactManagerDBEntities()

Public Function GetContact(ByVal id As Integer) As Contact Implements IContactManagerRepository.GetContact
    Return (From c In _entities.ContactSet _
            Where c.Id = id _
            Select c).FirstOrDefault()
End Function


Public Function ListContacts() As IEnumerable(Of Contact) Implements IContactManagerRepository.ListContacts
    Return _entities.ContactSet.ToList()
End Function


Public Function CreateContact(ByVal contactToCreate As Contact) As Contact Implements
IContactManagerRepository.CreateContact
    _entities.AddToContactSet(contactToCreate)
    _entities.SaveChanges()
    Return contactToCreate
End Function


Public Function EditContact(ByVal contactToEdit As Contact) As Contact Implements
IContactManagerRepository.EditContact
    Dim originalContact = GetContact(contactToEdit.Id)
    _entities.ApplyPropertyChanges(originalContact.EntityKey.EntitySetName, contactToEdit)
    _entities.SaveChanges()
    Return contactToEdit
End Function


Public Sub DeleteContact(ByVal contactToDelete As Contact) Implements IContactManagerRepository.DeleteContact
    Dim originalContact = GetContact(contactToDelete.Id)
    _entities.DeleteObject(originalContact)
    _entities.SaveChanges()
End Sub

End Class
```

Notice that the EntityContactManagerRepository class implements the IContactManagerRepository interface. The class implements all five of the methods described by that interface.

You might wonder why we need to bother with an interface. Why do we need to create both an interface and a class that implements it?

With one exception, the remainder of our application will interact with the interface and not the concrete class. Instead of calling the methods exposed by the EntityContactManagerRepository class, we'll call the methods exposed by the IContactManagerRepository interface.

That way, we can implement the interface with a new class without needing to modify the remainder of our application. For example, at some future date, we might want to implement an DataServicesContactManagerRepository class that implements the IContactManagerRepository interface. The DataServicesContactManagerRepository class might use ADO.NET Data Services to access a database instead of the Microsoft Entity Framework.

If our application code is programmed against the IContactManagerRepository interface instead of the concrete EntityContactManagerRepository class then we can switch concrete classes without modifying any of the rest of our code. For example, we can switch from the EntityContactManagerRepository class to the DataServicesContactManagerRepository class without modifying our data access or validation logic.

Programming against interfaces (abstractions) instead of concrete classes makes our application more resilient to change.

> **NOTE**
> You can quickly create an interface from a concrete class within Visual Studio by selecting the menu option Refactor, Extract Interface. For example, you can create the EntityContactManagerRepository class first and then use Extract Interface to generate the IContactManagerRepository interface automatically.

Using the Dependency Injection Software Design Pattern

Now that we have migrated our data access code to a separate Repository class, we need to modify our Contact controller to use this class. We will take advantage of a software design pattern called Dependency Injection to use the Repository class in our controller.

The modified Contact controller is contained in Listing 3.

Listing 3 - Controllers\ContactController.vb

```
Public Class ContactController
    Inherits System.Web.Mvc.Controller

    Private _repository As IContactManagerRepository

    Sub New()
        Me.New(new EntityContactManagerRepository())
    End Sub

    Sub New(repository As IContactManagerRepository)
        _repository = repository
    End Sub

    Protected Sub ValidateContact(contactToValidate As Contact)
        If contactToValidate.FirstName.Trim().Length = 0 Then
            ModelState.AddModelError("FirstName", "First name is required.")
        End If
        If contactToValidate.LastName.Trim().Length = 0 Then
            ModelState.AddModelError("LastName", "Last name is required.")
        End If
        If (contactToValidate.Phone.Length > 0 AndAlso Not Regex.IsMatch(contactToValidate.Phone, "((\(\d{3}\)
?)|(\d{3}-))?\d{3}-\d{4}"))
            ModelState.AddModelError("Phone", "Invalid phone number.")
        End If
        If (contactToValidate.Email.Length > 0 AndAlso  Not Regex.IsMatch(contactToValidate.Email, "^[\w-
\.]+@([\w-]+\.)+[\w-]{2,4}$"))
            ModelState.AddModelError("Email", "Invalid email address.")
        End If
    End Sub

    Function Index() As ActionResult
        Return View(_repository.ListContacts())
    End Function

    Function Create() As ActionResult
        Return View()
    End Function
```

```
    <AcceptVerbs(HttpVerbs.Post)> _
    Function Create(<Bind(Exclude:="Id")> ByVal contactToCreate As Contact) As ActionResult
        ' Validation logic
        ValidateContact(contactToCreate)
        If Not ModelState.IsValid Then
            Return View()
        End If

        ' Database logic
        Try
            _repository.CreateContact(contactToCreate)
            Return RedirectToAction("Index")
        Catch
            Return View()
        End Try
    End Function

    Function Edit(ByVal id As Integer) As ActionResult
        Return View(_repository.GetContact(id))
    End Function

    <AcceptVerbs(HttpVerbs.Post)> _
    Function Edit(ByVal contactToEdit As Contact) As ActionResult
        ' Validation logic
        ValidateContact(contactToEdit)
        If Not ModelState.IsValid Then
            Return View()
        End If

        ' Database logic
        Try
            _repository.EditContact(contactToEdit)
            Return RedirectToAction("Index")
        Catch
            Return View()
        End Try
    End Function

    Function Delete(ByVal id As Integer) As ActionResult
        Return View(_repository.GetContact(id))
    End Function

    <AcceptVerbs(HttpVerbs.Post)> _
    Function Delete(ByVal contactToDelete As Contact) As ActionResult
        Try
            _repository.DeleteContact(contactToDelete)
            Return RedirectToAction("Index")
        Catch
            Return View()
        End Try
    End Function

End Class
```

Notice that the Contact controller in Listing 3 has two constructors. The first constructor passes a concrete instance of the IContactManagerRepository interface to the second constructor. The Contact controller class uses *Constructor Dependency Injection*.

The one and only place that the EntityContactManagerRepository class is used is in the first constructor. The remainder of the class uses the IContactManagerRepository interface instead of the concrete EntityContactManagerRepository class.

This makes it easy to switch implementations of the IContactManagerRepository class in the future. If you want to use the DataServicesContactRepository class instead of the EntityContactManagerRepository class, just modify the first constructor.

Constructor Dependency injection also makes the Contact controller class very testable. In your unit tests, you can instantiate the Contact controller by passing a mock implementation of the IContactManagerRepository class. This feature of Dependency Injection will be very important to us in the next iteration when we build unit tests for the Contact Manager application.

> **NOTE**
>
> If you want to completely decouple the Contact controller class from a particular implementation of the IContactManagerRepository interface then you can take advantage of a framework that supports Dependency Injection such as StructureMap or the Microsoft Entity Framework (MEF). By taking advantage of a Dependency Injection framework, you never need to refer to a concrete class in your code.

Creating a Service Layer

You might have noticed that our validation logic is still mixed up with our controller logic in the modified controller class in Listing 3. For the same reason that it is a good idea to isolate our data access logic, it is a good idea to isolate our validation logic.

To fix this problem, we can create a separate service layer. The service layer is a separate layer that we can insert between our controller and repository classes. The service layer contains our business logic including all of our validation logic.

The ContactManagerService is contained in Listing 4. It contains the validation logic from the Contact controller class.

Listing 4 - Models\ContactManagerService.vb

```
Public Class ContactManagerService
Implements IContactManagerService

Private _validationDictionary As IValidationDictionary
Private _repository As IContactManagerRepository


Public Sub New(ByVal validationDictionary As IValidationDictionary)
    Me.New(validationDictionary, New EntityContactManagerRepository())
End Sub


Public Sub New(ByVal validationDictionary As IValidationDictionary, ByVal repository As
IContactManagerRepository)
    _validationDictionary = validationDictionary
    _repository = repository
End Sub


Public Function ValidateContact(ByVal contactToValidate As Contact) As Boolean
    If contactToValidate.FirstName.Trim().Length = 0 Then
        _validationDictionary.AddError("FirstName", "First name is required.")
    End If
    If contactToValidate.LastName.Trim().Length = 0 Then
        _validationDictionary.AddError("LastName", "Last name is required.")
    End If
    If contactToValidate.Phone.Length > 0 AndAlso (Not Regex.IsMatch(contactToValidate.Phone, "((\(\d{3}\) ?)|
(\d{3}-))?\d{3}-\d{4}")) Then
        _validationDictionary.AddError("Phone", "Invalid phone number.")
    End If
    If contactToValidate.Email.Length > 0 AndAlso (Not Regex.IsMatch(contactToValidate.Email, "^[\w-\.]+@([\w-
]+\.)+[\w-]{2,4}$")) Then
        _validationDictionary.AddError("Email", "Invalid email address.")
    End If
    Return _validationDictionary.IsValid
```

```
    Return _validationDictionary.IsValid
End Function


#Region "IContactManagerService Members"

Public Function CreateContact(ByVal contactToCreate As Contact) As Boolean Implements
IContactManagerService.CreateContact
    ' Validation logic
    If Not ValidateContact(contactToCreate) Then
        Return False
    End If

    ' Database logic
    Try
        _repository.CreateContact(contactToCreate)
    Catch
        Return False
    End Try
    Return True
End Function

Public Function EditContact(ByVal contactToEdit As Contact) As Boolean Implements
IContactManagerService.EditContact
    ' Validation logic
    If Not ValidateContact(contactToEdit) Then
        Return False
    End If

    ' Database logic
    Try
        _repository.EditContact(contactToEdit)
    Catch
        Return False
    End Try
    Return True
End Function

Public Function DeleteContact(ByVal contactToDelete As Contact) As Boolean Implements
IContactManagerService.DeleteContact
    Try
        _repository.DeleteContact(contactToDelete)
    Catch
        Return False
    End Try
    Return True
End Function

Public Function GetContact(ByVal id As Integer) As Contact Implements IContactManagerService.GetContact
    Return _repository.GetContact(id)
End Function

Public Function ListContacts() As IEnumerable(Of Contact) Implements IContactManagerService.ListContacts
    Return _repository.ListContacts()
End Function

#End Region
End Class
```

Notice that the constructor for the ContactManagerService requires a ValidationDictionary. The service layer communicates with the controller layer through this ValidationDictionary. We discuss the ValidationDictionary in detail in the following section when we discuss the Decorator pattern.

Notice, furthermore, that the ContactManagerService implements the IContactManagerService interface. You should always strive to program against interfaces instead of concrete classes. Other classes in the Contact Manager application do not interact with the ContactManagerService class directly. Instead, with one exception, the

remainder of the Contact Manager application is programmed against the IContactManagerService interface.

The IContactManagerService interface is contained in Listing 5.

Listing 5 - Models\IContactManagerService.vb

```
Public Interface IContactManagerService
Function CreateContact(ByVal contactToCreate As Contact) As Boolean
Function DeleteContact(ByVal contactToDelete As Contact) As Boolean
Function EditContact(ByVal contactToEdit As Contact) As Boolean
Function GetContact(ByVal id As Integer) As Contact
Function ListContacts() As IEnumerable(Of Contact)
End Interface
```

The modified Contact controller class is contained in Listing 6. Notice that the Contact controller no longer interacts with the ContactManager repository. Instead, the Contact controller interacts with the ContactManager service. Each layer is isolated as much as possible from other layers.

Listing 6 - Controllers\ContactController.vb

```
Public Class ContactController
    Inherits System.Web.Mvc.Controller

    Private _service As IContactManagerService

    Sub New()
        _service = new ContactManagerService(New ModelStateWrapper(ModelState))
    End Sub

    Sub New(service As IContactManagerService)
        _service = service
    End Sub

    Function Index() As ActionResult
        Return View(_service.ListContacts())
    End Function

    Function Create() As ActionResult
        Return View()
    End Function

    <AcceptVerbs(HttpVerbs.Post)> _
    Function Create(<Bind(Exclude:="Id")> ByVal contactToCreate As Contact) As ActionResult
        If _service.CreateContact(contactToCreate) Then
            Return RedirectToAction("Index")
        End If
        Return View()
    End Function

    Function Edit(ByVal id As Integer) As ActionResult
        Return View(_service.GetContact(id))
    End Function

    <AcceptVerbs(HttpVerbs.Post)> _
    Function Edit(ByVal contactToEdit As Contact) As ActionResult
        If _service.EditContact(contactToEdit) Then
            Return RedirectToAction("Index")
        End If
        Return View()
    End Function

    Function Delete(ByVal id As Integer) As ActionResult
        Return View(_service.GetContact(id))
    End Function

    <AcceptVerbs(HttpVerbs.Post)> _
    Function Delete(ByVal contactToDelete As Contact) As ActionResult
        If _service.DeleteContact(contactToDelete) Then
            return RedirectToAction("Index")
        End If
        Return View()
    End Function

End Class
```

Our application no longer runs afoul of the Single Responsibility Principle (SRP). The Contact controller in Listing 6 has been stripped of every responsibility other than controlling the flow of application execution. All the validation logic has been removed from the Contact controller and pushed into the service layer. All of the database logic has been pushed into the repository layer.

Using the Decorator Pattern

We want to be able to completely decouple our service layer from our controller layer. In principle, we should be able to compile our service layer in a separate assembly from our controller layer without needing to add a

reference to our MVC application.

However, our service layer needs to be able to pass validation error messages back to the controller layer. How can we enable the service layer to communicate validation error messages without coupling the controller and service layer? We can take advantage of a software design pattern named the Decorator pattern.

A controller uses a ModelStateDictionary named ModelState to represent validation errors. Therefore, you might be tempted to pass ModelState from the controller layer to the service layer. However, using ModelState in the service layer would make your service layer dependent on a feature of the ASP.NET MVC framework. This would be bad because, someday, you might want to use the service layer with a WPF application instead of an ASP.NET MVC application. In that case, you wouldn't want to reference the ASP.NET MVC framework to use the ModelStateDictionary class.

The Decorator pattern enables you to wrap an existing class in a new class in order to implement an interface. Our Contact Manager project includes the ModelStateWrapper class contained in Listing 7. The ModelStateWrapper class implements the interface in Listing 8.

Listing 7 - Models\Validation\ModelStateWrapper.vb

```
Public Class ModelStateWrapper
Implements IValidationDictionary

Private _modelState As ModelStateDictionary

Public Sub New(ByVal modelState As ModelStateDictionary)
    _modelState = modelState
End Sub

Public Sub AddError(ByVal key As String, ByVal errorMessage As String) Implements
IValidationDictionary.AddError
    _modelState.AddModelError(key, errorMessage)
End Sub

Public ReadOnly Property IsValid() As Boolean Implements IValidationDictionary.IsValid
    Get
        Return _modelState.IsValid
    End Get
End Property

End Class
```

Listing 8 - Models\Validation\IValidationDictionary.vb

```
Public Interface IValidationDictionary

Sub AddError(ByVal key As String, ByVal errorMessage As String)
ReadOnly Property IsValid() As Boolean

End Interface
```

If you take a close look at Listing 5 then you'll see that the ContactManager service layer uses the IValidationDictionary interface exclusively. The ContactManager service is not dependent on the ModelStateDictionary class. When the Contact controller creates the ContactManager service, the controller wraps its ModelState like this:

```
_service = new ContactManagerService(New ModelStateWrapper(ModelState))
```

Summary

In this iteration, we did not add any new functionality to the Contact Manager application. The goal of this iteration was to refactor the Contact Manager application so that is easier to maintain and modify.

First, we implemented the Repository software design pattern. We migrated all of the data access code to a separate ContactManager repository class.

We also isolated our validation logic from our controller logic. We created a separate service layer that contains all of our validation code. The controller layer interacts with the service layer, and the service layer interacts with the repository layer.

When we created the service layer, we took advantage of the Decorator pattern to isolate ModelState from our service layer. In our service layer, we programmed against the IValidationDictionary interface instead of ModelState.

Finally, we took advantage of a software design pattern named the Dependency Injection pattern. This pattern enables us to program against interfaces (abstractions) instead of concrete classes. Implementing the Dependency Injection design pattern also makes our code more testable. In the next iteration, we add unit tests to our project.

PREVIOUS NEXT

Iteration #5 – Create unit tests (VB)

4/10/2020 • 15 minutes to read • Edit Online

by Microsoft

Download Code

> In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.

Building a Contact Management ASP.NET MVC Application (VB)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information - names, phone numbers and email addresses - for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 - Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 - Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 - Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 - Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 - Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 - Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 - Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

In the previous iteration of the Contact Manager application, we refactored the application to be more loosely coupled. We separated the application into distinct controller, service, and repository layers. Each layer interacts with the layer beneath it through interfaces.

We refactored the application to make the application easier to maintain and modify. For example, if we need to use a new data access technology, we can simply change the repository layer without touching the controller or service layer. By making the Contact Manager loosely coupled, we ve made the application more resilient to

change.

But, what happens when we need to add a new feature to the Contact Manager application? Or, what happens when we fix a bug? A sad, but well proven, truth of writing code is that whenever you touch code you create the risk of introducing new bugs.

For example, one fine day, your manager might ask you to add a new feature to the Contact Manager. She wants you to add support for Contact Groups. She wants you to enable users to organize their contacts into groups such as Friends, Business, and so on.

In order to implement this new feature, you'll need to modify all three layers of the Contact Manager application. You'll need to add new functionality to the controllers, the service layer, and the repository. As soon as you start modifying code, you risk breaking functionality that worked before.

Refactoring our application into separate layers, as we did in the previous iteration, was a good thing. It was a good thing because it enables us to make changes to entire layers without touching the rest of the application. However, if you want to make the code within a layer easier to maintain and modify, you need to create unit tests for the code.

You use a unit test to test an individual unit of code. These units of code are smaller than entire application layers. Typically, you use a unit test to verify whether a particular method in your code behaves in the way that you expect. For example, you would create a unit test for the CreateContact() method exposed by the ContactManagerService class.

The unit tests for an application work just like a safety net. Whenever you modify code in an application, you can run a set of unit tests to check whether the modification breaks existing functionality. Unit tests make your code safe to modify. Unit tests make all of the code in your application more resilient to change.

In this iteration, we add unit tests to our Contact Manager application. That way, in the next iteration, we can add Contact Groups to our application without worrying about breaking existing functionality.

> **NOTE**
> There are a variety of unit testing frameworks including NUnit, xUnit.net, and MbUnit. In this tutorial, we use the unit testing framework included with Visual Studio. However, you could just as easily use one of these alternative frameworks.

What Gets Tested

In the perfect world, all of your code would be covered by unit tests. In the perfect world, you would have the perfect safety net. You would be able to modify any line of code in your application and know instantly, by executing your unit tests, whether the change broke existing functionality.

However, we don t live in a perfect world. In practice, when writing unit tests, you concentrate on writing tests for your business logic (for example, validation logic). In particular, you *do not* write unit tests for your data access logic or your view logic.

To be useful, unit tests must execute very quickly. You easily can accumulate hundreds (or even thousands) of unit tests for an application. If the unit tests take a long time to run then you'll avoid executing them. In other words, long running unit tests are useless for day to day coding purposes.

For this reason, you typically do not write unit tests for code that interacts with a database. Running hundreds of unit tests against a live database would be too slow. Instead, you mock your database and write code that interacts with the mock database (we discuss mocking a database below).

For a similar reason, you typically do not write unit tests for views. In order to test a view, you must spin up a web server. Because spinning up a web server is a relatively slow process, creating unit tests for your views is not recommended.

If your view contains complicated logic then you should consider moving the logic into Helper methods. You can write unit tests for Helper methods that execute without spinning up a web server.

> **NOTE**
> While writing tests for data access logic or view logic is not a good idea when writing unit tests, these tests can be very valuable when building functional or integration tests.

> **NOTE**
> ASP.NET MVC is the Web Forms View Engine. While the Web Forms View Engine is dependent on a web server, other view engines might not be.

Using a Mock Object Framework

When building unit tests, you almost always need to take advantage of a Mock Object framework. A Mock Object framework enables you to create mocks and stubs for the classes in your application.

For example, you can use a Mock Object framework to generate a mock version of your repository class. That way, you can use the mock repository class instead of the real repository class in your unit tests. Using the mock repository enables you to avoid executing database code when executing a unit test.

Visual Studio does not include a Mock Object framework. However, there are several commercial and open source Mock Object frameworks available for the .NET framework:

1. Moq - This framework is available under the open source BSD license. You can download Moq from https://code.google.com/p/moq/.
2. Rhino Mocks - This framework is available under the open source BSD license. You can download Rhino Mocks from http://ayende.com/projects/rhino-mocks.aspx.
3. Typemock Isolator - This is a commercial framework. You can download a trial version from http://www.typemock.com/.

In this tutorial, I decided to use Moq. However, you could just as easily use Rhino Mocks or Typemock Isolator to create the Mock objects for the Contact Manager application.

Before you can use Moq, you need to complete the following steps:

1. .
2. Before you unzip the download, make sure that you right-click the file and click the button labeled **Unblock** (see Figure 1).
3. Unzip the download.
4. Add a reference to the Moq assembly to your Test project by selecting the menu option **Project, Add Reference** to open the **Add Reference** dialog. Under the Browse tab, browse to the folder where you unzipped Moq and select the Moq.dll assembly. Click the **OK** button (see Figure 2).

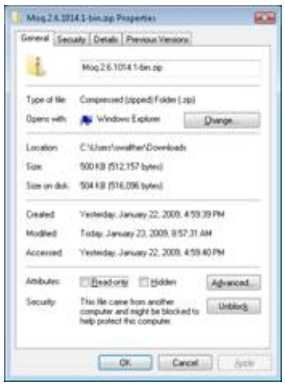


Figure 01: Unblocking Moq(Click to view full-size image)

Figure 02: References after adding Moq(Click to view full-size image)

Creating Unit Tests for the Service Layer

Let s start by creating a set of unit tests for our Contact Manager application s service layer. We'll use these tests to verify our validation logic.

Create a new folder named Models in the ContactManager.Tests project. Next, right-click the Models folder and select **Add, New Test**. The **Add New Test** dialog shown in Figure 3 appears. Select the **Unit Test** template and name your new test ContactManagerServiceTest.vb. Click the **OK** button to add your new test to your Test Project.

> **NOTE**
>
> In general, you want the folder structure of your Test Project to match the folder structure of your ASP.NET MVC project. For example, you place controller tests in a Controllers folder, model tests in a Models folder, and so on.

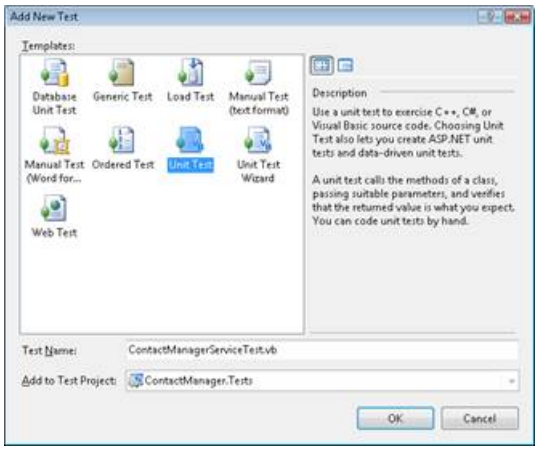


Figure 03: Models\ContactManagerServiceTest.cs(Click to view full-size image)

Initially, we want to test the CreateContact() method exposed by the ContactManagerService class. We'll create the following five tests:

- CreateContact() - Tests that CreateContact() returns the value true when a valid Contact is passed to the method.
- CreateContactRequiredFirstName() - Tests that an error message is added to model state when a Contact with a missing first name is passed to the CreateContact() method.
- CreateContactRequiredLastName() - Tests that an error message is added to model state when a Contact with a missing last name is passed to the CreateContact() method.
- CreateContactInvalidPhone() - Tests that an error message is added to model state when a Contact with an invalid phone number is passed to the CreateContact() method.
- CreateContactInvalidEmail() - Tests that an error message is added to model state when a Contact with an invalid email address is passed to the CreateContact() method..

The first test verifies that a valid Contact does not generate a validation error. The remaining tests check each of the validation rules.

The code for these tests is contained in Listing 1.

Listing 1 - Models\ContactManagerServiceTest.vb

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting
Imports Moq
Imports System.Web.Mvc

<TestClass()> _
Public Class ContactManagerServiceTest

    Private _mockRepository As Mock(Of IContactManagerRepository)
    Private _modelState As ModelStateDictionary
    Private _service As IContactManagerService

    <TestInitialize()> _
    Public Sub Initialize()
        _mockRepository = New Mock(Of IContactManagerRepository)()
        _modelState = New ModelStateDictionary()
        _service = New ContactManagerService(new ModelStateWrapper(_modelState), _mockRepository.Object)
    End Sub

    <TestMethod()> _
    Public Sub CreateContact()
        ' Arrange
        Dim contactToCreate = Contact.CreateContact(-1, "Stephen", "Walther", "555-5555",
"steve@somewhere.com")

        ' Act
        Dim result = _service.CreateContact(contactToCreate)

        ' Assert
        Assert.IsTrue(result)
    End Sub

    <TestMethod()> _
    Public Sub CreateContactRequiredFirstName()
        ' Arrange
        Dim contactToCreate = Contact.CreateContact(-1, String.Empty, "Walther", "555-5555",
"steve@somewhere.com")

        ' Act
        Dim result = _service.CreateContact(contactToCreate)
```

```
        ' Assert
        Assert.IsFalse(result)
        Dim [error] = _modelState("FirstName").Errors(0)
        Assert.AreEqual("First name is required.", [error].ErrorMessage)
    End Sub

    <TestMethod()> _
    Public Sub CreateContactRequiredLastName()
        ' Arrange
        Dim contactToCreate = Contact.CreateContact(-1, "Stephen", String.Empty, "555-5555",
"steve@somewhere.com")

        ' Act
        Dim result = _service.CreateContact(contactToCreate)

        ' Assert
        Assert.IsFalse(result)
        Dim [error] = _modelState("LastName").Errors(0)
        Assert.AreEqual("Last name is required.", [error].ErrorMessage)
    End Sub

    <TestMethod()> _
    Public Sub CreateContactInvalidPhone()
        ' Arrange
        Dim contactToCreate = Contact.CreateContact(-1, "Stephen", "Walther", "apple", "steve@somewhere.com")

        ' Act
        Dim result = _service.CreateContact(contactToCreate)

        ' Assert
        Assert.IsFalse(result)
        Dim [error] = _modelState("Phone").Errors(0)
        Assert.AreEqual("Invalid phone number.", [error].ErrorMessage)
    End Sub

    <TestMethod()> _
    Public Sub CreateContactInvalidEmail()
        ' Arrange
        Dim contactToCreate = Contact.CreateContact(-1, "Stephen", "Walther", "555-5555", "apple")

        ' Act
        Dim result = _service.CreateContact(contactToCreate)

        ' Assert
        Assert.IsFalse(result)
        Dim [error] = _modelState("Email").Errors(0)
        Assert.AreEqual("Invalid email address.", [error].ErrorMessage)
    End Sub
End Class
```

Because we use the Contact class in Listing 1, we need to add a reference to the Microsoft Entity Framework to our Test project. Add a reference to the System.Data.Entity assembly.

Listing 1 contains a method named Initialize() that is decorated with the [TestInitialize] attribute. This method is called automatically before each of the unit tests is run (it is called 5 times right before each of the unit tests). The Initialize() method creates a mock repository with the following line of code:

```
_mockRepository = New Mock(Of IContactManagerRepository)()
```

This line of code uses the Moq framework to generate a mock repository from the IContactManagerRepository interface. The mock repository is used instead of the actual EntityContactManagerRepository to avoid accessing the database when each unit test is run. The mock repository implements the methods of the IContactManagerRepository interface, but the methods don t actually do anything.

> **NOTE**
> When using the Moq framework, there is a distinction between _mockRepository and _mockRepository.Object. The former refers to the Mock(Of IContactManagerRepository) class that contains methods for specifying how the mock repository will behave. The latter refers to the actual mock repository that implements the IContactManagerRepository interface.

The mock repository is used in the Initialize() method when creating an instance of the ContactManagerService class. All of the individual unit tests use this instance of the ContactManagerService class.

Listing 1 contains five methods that correspond to each of the unit tests. Each of these methods is decorated with the [TestMethod] attribute. When you run the unit tests, any method that has this attribute is called. In other words, any method that is decorated with the [TestMethod] attribute is a unit test.

The first unit test, named CreateContact(), verifies that calling CreateContact() returns the value true when a valid instance of the Contact class is passed to the method. The test creates an instance of the Contact class, calls the CreateContact() method, and verifies that CreateContact() returns the value true.

The remaining tests verify that when the CreateContact() method is called with an invalid Contact then the method returns false and the expected validation error message is added to model state. For example, the CreateContactRequiredFirstName() test creates an instance of the Contact class with an empty string for its FirstName property. Next, the CreateContact() method is called with the invalid Contact. Finally, the test verifies that CreateContact() returns false and that model state contains the expected validation error message "First name is required."

You can run the unit tests in Listing 1 by selecting the menu option **Test, Run, All Tests in Solution (CTRL+R, A)**. The results of the tests are displayed in the Test Results window (see Figure 4).

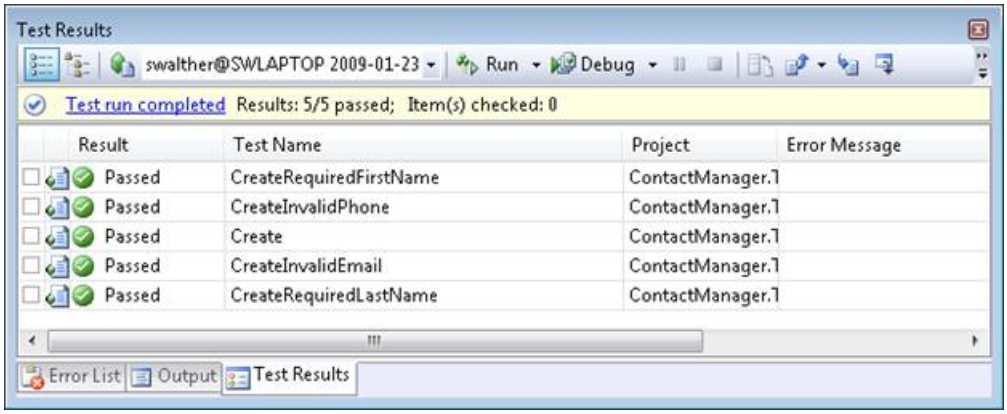


Figure 04: Test Results (Click to view full-size image)

Creating Unit Tests for Controllers

ASP.NET MVC application control the flow of user interaction. When testing a controller, you want to test whether the controller returns the right action result and view data. You also might want to test whether a controller interacts with model classes in the manner expected.

For example, Listing 2 contains two unit tests for the Contact controller Create() method. The first unit test verifies that when a valid Contact is passed to the Create() method then the Create() method redirects to the Index action. In other words, when passed a valid Contact, the Create() method should return a RedirectToRouteResult that represents the Index action.

We don t want to test the ContactManager service layer when we are testing the controller layer. Therefore, we mock the service layer with the following code in the Initialize method:

```
_service = New Mock(Of IContactManagerService)()
```

In the CreateValidContact() unit test, we mock the behavior of calling the service layer CreateContact() method with the following line of code:

```
_service.Expect( Function(s) s.CreateContact(contactToCreate) ).Returns(True)
```

This line of code causes the mock ContactManager service to return the value true when its CreateContact() method is called. By mocking the service layer, we can test the behavior of our controller without needing to execute any code in the service layer.

The second unit test verifies that the Create() action returns the Create view when an invalid contact is passed to the method. We cause the service layer CreateContact() method to return the value false with the following line of code:

```
_service.Expect( Function(s) s.CreateContact(contactToCreate) ).Returns(False)
```

If the Create() method behaves as we expect then it should return the Create view when the service layer returns the value false. That way, the controller can display the validation error messages in the Create view and the user has a chance to correct that invalid Contact properties.

If you plan to build unit tests for your controllers then you need to return explicit view names from your controller actions. For example, do not return a view like this:

Return View()

Instead, return the view like this:

Return View("Create")

If you are not explicit when returning a view then the ViewResult.ViewName property returns an empty string.

Listing 2 - Controllers\ContactControllerTest.vb

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting
Imports Moq
Imports System.Web.Mvc

<TestClass()> _
Public Class ContactControllerTest

    Private _service As Mock(Of IContactManagerService)

    <TestInitialize()> _
    Public Sub Initialize()
        _service = New Mock(Of IContactManagerService)()
    End Sub

    <TestMethod()> _
    Public Sub CreateValidContact()
        ' Arrange
        Dim contactToCreate = New Contact()
        _service.Expect(Function(s) s.CreateContact(contactToCreate)).Returns(True)
        Dim controller = New ContactController(_service.Object)

        ' Act
        Dim result = CType(controller.Create(contactToCreate), RedirectToRouteResult)

        ' Assert
        Assert.AreEqual("Index", result.RouteValues("action"))
    End Sub

    <TestMethod()> _
    Public Sub CreateInvalidContact()
        ' Arrange
        Dim contactToCreate = New Contact()
        _service.Expect(Function(s) s.CreateContact(contactToCreate)).Returns(False)
        Dim controller = New ContactController(_service.Object)

        ' Act
        Dim result = CType(controller.Create(contactToCreate), ViewResult)

        ' Assert
        Assert.AreEqual("Create", result.ViewName)
    End Sub

End Class
```

Summary

In this iteration, we created unit tests for our Contact Manager application. We can run these unit tests at any time to verify that our application still behaves in the manner that we expect. The unit tests act as a safety net for our application enabling us to safely modify our application in the future.

We created two sets of unit tests. First, we tested our validation logic by creating unit tests for our service layer. Next, we tested our flow control logic by creating unit tests for our controller layer. When testing our service layer, we isolated our tests for our service layer from our repository layer by mocking our repository layer. When testing the controller layer, we isolated our tests for our controller layer by mocking the service layer.

In the next iteration, we modify the Contact Manager application so that it supports Contact Groups. We'll add this new functionality to our application using a software design process called test-driven development.

PREVIOUS NEXT

Iteration #6 – Use test-driven development (VB)

4/10/2020 • 21 minutes to read • Edit Online

by Microsoft

Download Code

> In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.

Building a Contact Management ASP.NET MVC Application (VB)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information - names, phone numbers and email addresses - for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 - Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 - Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 - Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 - Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 - Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 - Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 - Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

In the previous iteration of the Contact Manager application, we created unit tests to provide a safety net for our code. The motivation for creating the unit tests was to make our code more resilient to change. With unit tests in place, we can happily make any change to our code and immediately know whether we have broken existing functionality.

In this iteration, we use unit tests for an entirely different purpose. In this iteration, we use unit tests as part of an application design philosophy called *test-driven development*. When you practice test-driven development, you

write tests first and then write code against the tests.

More precisely, when practicing test-driven development, there are three steps that you complete when creating code (Red/ Green/Refactor):

1. Write a unit test that fails (Red)
2. Write code that passes the unit test (Green)
3. Refactor your code (Refactor)

First, you write the unit test. The unit test should express your intention for how you expect your code to behave. When you first create the unit test, the unit test should fail. The test should fail because you have not yet written any application code that satisfies the test.

Next, you write just enough code in order for the unit test to pass. The goal is to write the code in the laziest, sloppiest and fastest possible way. You should not waste time thinking about the architecture of your application. Instead, you should focus on writing the minimal amount of code necessary to satisfy the intention expressed by the unit test.

Finally, after you have written enough code, you can step back and consider the overall architecture of your application. In this step, you rewrite (refactor) your code by taking advantage of software design patterns -- such as the repository pattern -- so that your code is more maintainable. You can fearlessly rewrite your code in this step because your code is covered by unit tests.

There are many benefits that result from practicing test-driven development. First, test-driven development forces you to focus on code that actually needs to be written. Because you are constantly focused on just writing enough code to pass a particular test, you are prevented from wandering into the weeds and writing massive amounts of code that you will never use.

Second, a "test first" design methodology forces you to write code from the perspective of how your code will be used. In other words, when practicing test-driven development, you are constantly writing your tests from a user perspective. Therefore, test-driven development can result in cleaner and more understandable APIs.

Finally, test-driven development forces you to write unit tests as part of the normal process of writing an application. As a project deadline approaches, testing is typically the first thing that goes out the window. When practicing test-driven development, on the other hand, you are more likely to be virtuous about writing unit tests because test-driven development makes unit tests central to the process of building an application.

> **NOTE**
> To learn more about test-driven development, I recommend that you read Michael Feathers book **Working Effectively with Legacy Code**.

In this iteration, we add a new feature to our Contact Manager application. We add support for Contact Groups. You can use Contact Groups to organize your contacts into categories such as Business and Friend groups.

We'll add this new functionality to our application by following a process of test-driven development. We'll write our unit tests first and we'll write all of our code against these tests.

What Gets Tested

As we discussed in the previous iteration, you typically do not write unit tests for data access logic or view logic. You don t write unit tests for data access logic because accessing a database is a relatively slow operation. You don t write unit tests for view logic because accessing a view requires spinning up a web server which is a relatively slow operation. You shouldn't write a unit test unless the test can be executed over and over again very fast

Because test-driven development is driven by unit tests, we focus initially on writing controller and business logic.

We avoid touching the database or views. We won't modify the database or create our views until the very end of this tutorial. We start with what can be tested.

Creating User Stories

When practicing test-driven development, you always start by writing a test. This immediately raises the question: How do you decide what test to write first? To answer this question, you should write a set of *user stories*.

A user story is a very brief (usually one sentence) description of a software requirement. It should be a non-technical description of a requirement written from a user perspective.

Here s the set of user stories that describe the features required by the new Contact Group functionality:

1. User can view a list of contact groups.
2. User can create a new contact group.
3. User can delete an existing contact group.
4. User can select a contact group when creating a new contact.
5. User can select a contact group when editing an existing contact.
6. A list of contact groups is displayed in the Index view.
7. When a user clicks a contact group, a list of matching contacts is displayed.

Notice that this list of user stories is completely understandable by a customer. There is no mention of technical implementation details.

While in the process of building your application, the set of user stories can become more refined. You might break a user story into multiple stories (requirements). For example, you might decide that creating a new contact group should involve validation. Submitting a contact group without a name should return a validation error.

After you create a list of user stories, you are ready to write your first unit test. We'll start by creating a unit test for viewing the list of contact groups.

Listing Contact Groups

Our first user story is that a user should be able to view a list of contact groups. We need to express this story with a test.

Create a new unit test by right-clicking the Controllers folder in the ContactManager.Tests project, selecting **Add, New Test**, and selecting the **Unit Test** template (see Figure 1). Name the new unit test GroupControllerTest.vb and click the **OK** button.

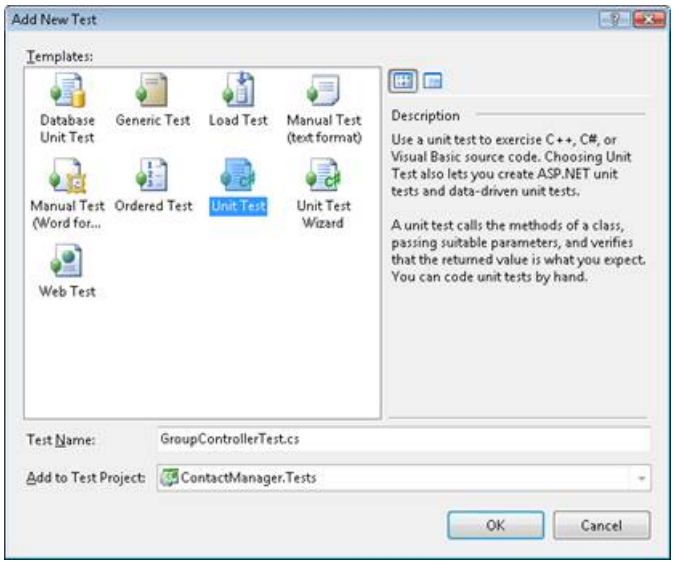


Figure 01: Adding the GroupControllerTest unit test(Click to view full-size image)

Our first unit test is contained in Listing 1. This test verifies that the Index() method of the Group controller returns a set of Groups. The test verifies that a collection of Groups is returned in view data.

Listing 1 - Controllers\GroupControllerTest.vb

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting
Imports System.Web.Mvc

<TestClass()> _
Public Class GroupControllerTest

    <TestMethod()> _
    Public Sub Index()
        ' Arrange
        Dim controller = New GroupController()

        ' Act
        Dim result = CType(controller.Index(), ViewResult)

        ' Assert
        Assert.IsInstanceOfType(result.ViewData.Model, GetType(IEnumerable(Of Group)))
    End Sub
End Class
```

When you first type the code in Listing 1 in Visual Studio, you'll get a lot of red squiggly lines. We have not created the GroupController or Group classes.

At this point, we can t even build our application so we can t execute our first unit test. That s good. That counts as a failing test. Therefore, we now have permission to start writing application code. We need to write enough code to execute our test.

The Group controller class in Listing 2 contains the bare minimum of code required to pass the unit test. The Index() action returns a statically coded list of Groups (the Group class is defined in Listing 3).

Listing 2 - Controllers\GroupController.vb

```
Public Class GroupController
    Inherits System.Web.Mvc.Controller

    Function Index() As ActionResult
        Dim groups = new List(Of Group)
        Return View(groups)
    End Function

End Class
```

Listing 3 - Models\Group.vb

```
Public Class Group

End Class
```

After we add the GroupController and Group classes to our project, our first unit test completes successfully (see Figure 2). We have done the minimum work required to pass the test. It is time to celebrate.

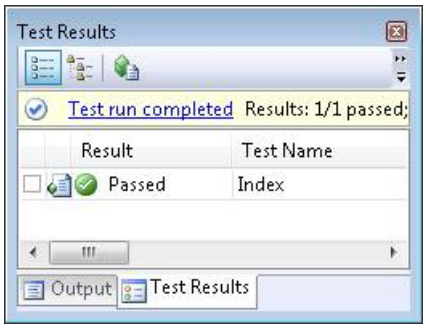


Figure 02: Success!(Click to view full-size image)

Creating Contact Groups

Now we can move on to the second user story. We need to be able to create new contact groups. We need to express this intention with a test.

The test in Listing 4 verifies that calling the Create() method with a new Group adds the Group to the list of Groups returned by the Index() method. In other words, if I create a new group then I should be able to get the new Group back from the list of Groups returned by the Index() method.

Listing 4 - Controllers\GroupControllerTest.vb

```
<TestMethod> _
Public Sub Create()
    ' Arrange
    Dim controller = New GroupController()

    ' Act
    Dim groupToCreate = New Group()
    controller.Create(groupToCreate)

    ' Assert
    Dim result = CType(controller.Index(), ViewResult)
    Dim groups = CType(result.ViewData.Model, IEnumerable(Of Group))
    CollectionAssert.Contains(groups.ToList(), groupToCreate)
End Sub
```

The test in Listing 4 calls the Group controller Create() method with a new contact Group. Next, the test verifies that calling the Group controller Index() method returns the new Group in view data.

The modified Group controller in Listing 5 contains the bare minimum of changes required to pass the new test.

Listing 5 - Controllers\GroupController.vb

```
Public Class GroupController
Inherits Controller

Private _groups As IList(Of Group) = New List(Of Group)()

Public Function Index() As ActionResult
    Return View(_groups)
End Function

Public Function Create(ByVal groupToCreate As Group) As ActionResult
    _groups.Add(groupToCreate)
    Return RedirectToAction("Index")

End Function
End Class
```

The Group controller in Listing 5 has a new Create() action. This action adds a Group to a collection of Groups. Notice that the Index() action has been modified to return the contents of the collection of Groups.

Once again, we have performed the bare minimum amount of work required to pass the unit test. After we make these changes to the Group controller, all of our unit tests pass.

Adding Validation

This requirement was not stated explicitly in the user story. However, it is reasonable to require that a Group have a name. Otherwise, organizing contacts into Groups would not be very useful.

Listing 6 contains a new test that expresses this intention. This test verifies that attempting to create a Group without supplying a name results in a validation error message in model state.

Listing 6 - Controllers\GroupControllerTest.vb

```
<TestMethod> _
Public Sub CreateRequiredName()
    ' Arrange
    Dim controller = New GroupController()

    ' Act
    Dim groupToCreate As New Group()
    groupToCreate.Name = String.Empty
    Dim result = CType(controller.Create(groupToCreate), ViewResult)

    ' Assert
    Dim [error] = result.ViewData.ModelState("Name").Errors(0)
    Assert.AreEqual("Name is required.", [error].ErrorMessage)
End Sub
```

In order to satisfy this test, we need to add a Name property to our Group class (see Listing 7). Furthermore, we need to add a tiny bit of validation logic to our Group controller s Create() action (see Listing 8).

Listing 7 - Models\Group.vb

```
Public Class Group

    Private _name As String

    Public Property Name() As String
    Get
        Return _name
    End Get
    Set(ByVal value As String)
        _name = value
    End Set
End Property

End Class
```

Listing 8 - Controllers\GroupController.vb

```
Public Function Create(ByVal groupToCreate As Group) As ActionResult
    ' Validation logic
    If groupToCreate.Name.Trim().Length = 0 Then
    ModelState.AddModelError("Name", "Name is required.")
    Return View("Create")
    End If

    ' Database logic
    _groups.Add(groupToCreate)
    Return RedirectToAction("Index")
End Function
```

Notice that the Group controller Create() action now contains both validation and database logic. Currently, the database used by the Group controller consists of nothing more than an in-memory collection.

Time to Refactor

The third step in Red/Green/Refactor is the Refactor part. At this point, we need to step back from our code and consider how we can refactor our application to improve its design. The Refactor stage is the stage at which we think hard about the best way of implementing software design principles and patterns.

We are free to modify our code in any way that we choose to improve the design of the code. We have a safety net of unit tests that prevent us from breaking existing functionality.

Right now, our Group controller is a mess from the perspective of good software design. The Group controller contains a tangled mess of validation and data access code. To avoid violating the Single Responsibility Principle, we need to separate these concerns into different classes.

Our refactored Group controller class is contained in Listing 9. The controller has been modified to use the ContactManager service layer. This is the same service layer that we use with the Contact controller.

Listing 10 contains the new methods added to the ContactManager service layer to support validating, listing, and creating groups. The IContactManagerService interface was updated to include the new methods.

Listing 11 contains a new FakeContactManagerRepository class that implements the IContactManagerRepository interface. Unlike the EntityContactManagerRepository class that also implements the IContactManagerRepository interface, our new FakeContactManagerRepository class does not communicate with the database. The FakeContactManagerRepository class uses an in-memory collection as a proxy for the database. We'll use this class in our unit tests as a fake repository layer.

Listing 9 - Controllers\GroupController.vb

```
Public Class GroupController
Inherits Controller

Private _service As IContactManagerService

Public Sub New()
    _service = New ContactManagerService(New ModelStateWrapper(Me.ModelState))

End Sub

Public Sub New(ByVal service As IContactManagerService)
    _service = service
End Sub

Public Function Index() As ActionResult
    Return View(_service.ListGroups())
End Function


Public Function Create(ByVal groupToCreate As Group) As ActionResult
    If _service.CreateGroup(groupToCreate) Then
        Return RedirectToAction("Index")
    End If
    Return View("Create")
End Function

End Class
```

Listing 10 - Controllers\ContactManagerService.vb

```
Public Function ValidateGroup(ByVal groupToValidate As Group) As Boolean
If groupToValidate.Name.Trim().Length = 0 Then
    _validationDictionary.AddError("Name", "Name is required.")
End If
Return _validationDictionary.IsValid
End Function

Public Function CreateGroup(ByVal groupToCreate As Group) As Boolean Implements
IContactManagerService.CreateGroup
    ' Validation logic
    If Not ValidateGroup(groupToCreate) Then
        Return False
    End If

    ' Database logic
    Try
        _repository.CreateGroup(groupToCreate)
    Catch
        Return False
    End Try
    Return True
End Function

Public Function ListGroups() As IEnumerable(Of Group) Implements IContactManagerService.ListGroups
    Return _repository.ListGroups()
End Function
```

Listing 11 - Controllers\FakeContactManagerRepository.vb

```
Public Class FakeContactManagerRepository
Implements IContactManagerRepository

Private _groups As IList(Of Group) = New List(Of Group)()

#Region "IContactManagerRepository Members"

' Group methods

Public Function CreateGroup(ByVal groupToCreate As Group) As Group Implements
IContactManagerRepository.CreateGroup
    _groups.Add(groupToCreate)
    Return groupToCreate
End Function

Public Function ListGroups() As IEnumerable(Of Group) Implements IContactManagerRepository.ListGroups
    Return _groups
End Function

' Contact methods

Public Function CreateContact(ByVal contactToCreate As Contact) As Contact Implements
IContactManagerRepository.CreateContact
    Throw New NotImplementedException()
End Function

Public Sub DeleteContact(ByVal contactToDelete As Contact) Implements IContactManagerRepository.DeleteContact
    Throw New NotImplementedException()
End Sub

Public Function EditContact(ByVal contactToEdit As Contact) As Contact Implements
IContactManagerRepository.EditContact
    Throw New NotImplementedException()
End Function

Public Function GetContact(ByVal id As Integer) As Contact Implements IContactManagerRepository.GetContact
    Throw New NotImplementedException()
End Function

Public Function ListContacts() As IEnumerable(Of Contact) Implements IContactManagerRepository.ListContacts
    Throw New NotImplementedException()
End Function

#End Region
End Class
```

Modifying the IContactManagerRepository interface requires use to implement the CreateGroup() and ListGroups() methods in the EntityContactManagerRepository class. The laziest and fastest way to do this is to add stub methods that look like this:

```
Public Function CreateGroup(groupToCreate As Group) As Group Implements IContactManagerRepository.CreateGroup

    throw New NotImplementedException()

End Function

Public Function ListGroups() As IEnumerable(Of Group) Implements IContactManagerRepository.ListGroups

    throw New NotImplementedException()

End Function
```

Finally, these changes to the design of our application require us to make some modifications to our unit tests. We now need to use the FakeContactManagerRepository when performing the unit tests. The updated

GroupControllerTest class is contained in Listing 12.

Listing 12 - Controllers\GroupControllerTest.vb

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting
Imports System.Web.Mvc

<TestClass()> _
Public Class GroupControllerTest

    Private _repository As IContactManagerRepository
    Private _modelState As ModelStateDictionary
    Private _service As IContactManagerService

    <TestInitialize()> _
    Public Sub Initialize()
        _repository = New FakeContactManagerRepository()
        _modelState = New ModelStateDictionary()
        _service = New ContactManagerService(New ModelStateWrapper(_modelState), _repository)
    End Sub

    <TestMethod()> _
    Public Sub Index()
        ' Arrange
        Dim controller = New GroupController(_service)

        ' Act
        Dim result = CType(controller.Index(), ViewResult)

        ' Assert
        Assert.IsInstanceOfType(result.ViewData.Model, GetType(IEnumerable(Of Group)))
    End Sub

    <TestMethod()> _
    Public Sub Create()
        ' Arrange
        Dim controller = New GroupController(_service)

        ' Act
        Dim groupToCreate = New Group()
        groupToCreate.Name = "Business"
        controller.Create(groupToCreate)

        ' Assert
        Dim result = CType(controller.Index(), ViewResult)
        Dim groups = CType(result.ViewData.Model, IEnumerable(Of Group))
        CollectionAssert.Contains(groups.ToList(), groupToCreate)
    End Sub

    <TestMethod()> _
    Public Sub CreateRequiredName()
        ' Arrange
        Dim controller = New GroupController(_service)

        ' Act
        Dim groupToCreate = New Group()
        groupToCreate.Name = String.Empty
        Dim result = CType(controller.Create(groupToCreate), ViewResult)

        ' Assert
        Dim nameError = _modelState("Name").Errors(0)
        Assert.AreEqual("Name is required.", nameError.ErrorMessage)
    End Sub

End Class
```

After we make all of these changes, once again, all of our unit tests pass. We have completed the entire cycle of Red/Green/Refactor. We have implemented the first two user stories. We now have supporting unit tests for the requirements expressed in the user stories. Implementing the remainder of the user stories involves repeating the same cycle of Red/Green/Refactor.

Modifying our Database

Unfortunately, even though we have satisfied all of the requirements expressed by our unit tests, our work is not done. We still need to modify our database.

We need to create a new Group database table. Follow these steps:

1. In the Server Explorer window, right-click the Tables folder and select the menu option **Add New Table**.
2. Enter the two columns described below in the Table Designer.
3. Mark the Id column as a primary key and Identity column.
4. Save the new table with the name Groups by clicking the icon of the floppy.

COLUMN NAME	DATA TYPE	ALLOW NULLS
Id	int	False
Name	nvarchar(50)	False

Next, we need to delete all of the data from the Contacts table (otherwise, we won't be able to create a relationship between the Contacts and Groups tables). Follow these steps:

1. Right-click the Contacts table and select the menu option **Show Table Data**.
2. Delete all of the rows.

Next, we need to define a relationship between the Groups database table and the existing Contacts database table. Follow these steps:

1. Double-click the Contacts table in the Server Explorer window to open the Table Designer.
2. Add a new integer column to the Contacts table named GroupId.
3. Click the Relationship button to open the Foreign Key Relationships dialog (see Figure 3).
4. Click the Add button.
5. Click the ellipsis button that appears next to the Table and Columns Specification button.
6. In the Tables and Columns dialog, select Groups as the primary key table and Id as the primary key column. Select Contacts as the foreign key table and GroupId as the foreign key column (see Figure 4). Click the OK button.
7. Under **INSERT and UPDATE Specification**, select the value **Cascade** for **Delete Rule**.
8. Click the Close button to close the Foreign Key Relationships dialog.
9. Click the Save button to save the changes to the Contacts table.

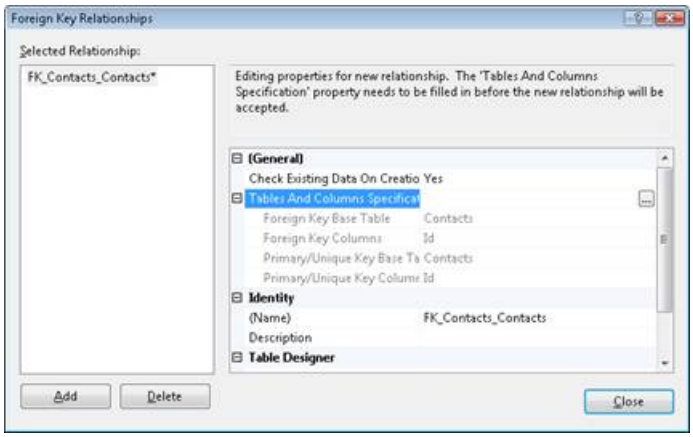


Figure 03: Creating a database table relationship (Click to view full-size image)

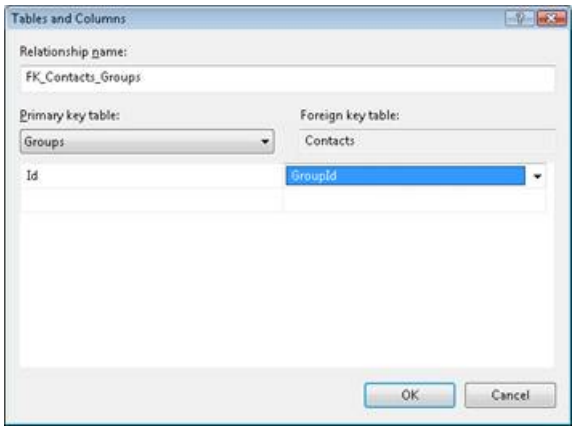


Figure 04: Specifying table relationships(Click to view full-size image)

Updating our Data Model

Next, we need to update our data model to represent the new database table. Follow these steps:

1. Double-click the ContactManagerModel.edmx file in the Models folder to open the Entity Designer.
2. Right-click the Designer surface and select the menu option **Update Model from Database**.
3. In the Update Wizard, select the Groups table and click the Finish button (see Figure 5).
4. Right-click the Groups entity and select the menu option **Rename**. Change the name of the *Groups* entity to *Group* (singular).
5. Right-click the Groups navigation property that appears at the bottom of the Contact entity. Change the name of the *Groups* navigation property to *Group* (singular).

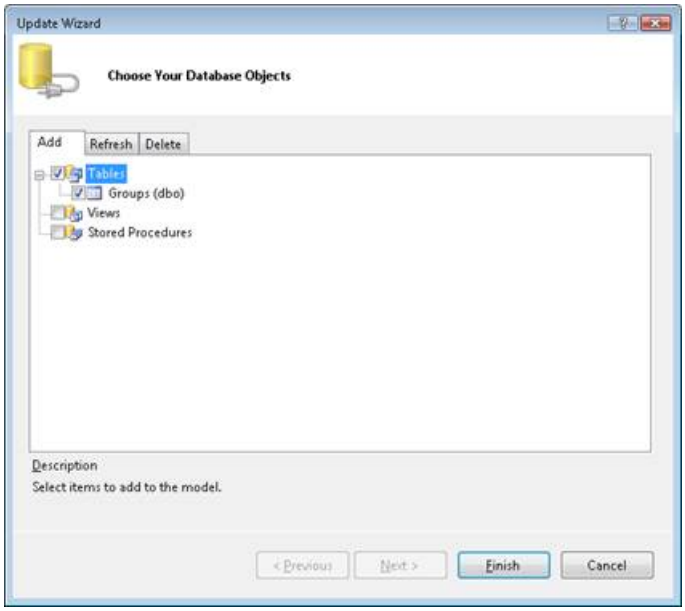


Figure 05: Updating an Entity Framework model from the database(Click to view full-size image)

After you complete these steps, your data model will represent both the Contacts and Groups tables. The Entity Designer should show both entities (see Figure 6).

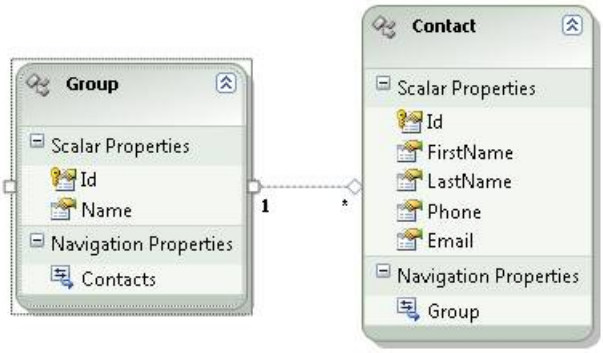


Figure 06: Entity Designer displaying Group and Contact(Click to view full-size image)

Creating our Repository Classes

Next, we need to implement our repository class. Over the course of this iteration, we added several new methods to the IContactManagerRepository interface while writing code to satisfy our unit tests. The final version of the IContactManagerRepository interface is contained in Listing 14.

Listing 14 - Models\IContactManagerRepository.vb

```
Public Interface IContactManagerRepository
' Contact methods
Function CreateContact(ByVal groupId As Integer, ByVal contactToCreate As Contact) As Contact
Sub DeleteContact(ByVal contactToDelete As Contact)
Function EditContact(ByVal groupId As Integer, ByVal contactToEdit As Contact) As Contact
Function GetContact(ByVal id As Integer) As Contact

' Group methods
Function CreateGroup(ByVal groupToCreate As Group) As Group
Function ListGroups() As IEnumerable(Of Group)
Function GetGroup(ByVal groupId As Integer) As Group
Function GetFirstGroup() As Group
Sub DeleteGroup(ByVal groupToDelete As Group)

End Interface
```

We haven't actually implemented any of the methods related to working with contact groups in our real EntityContactManagerRepository class. Currently, the EntityContactManagerRepository class has stub methods for each of the contact group methods listed in the IContactManagerRepository interface. For example, the ListGroups() method currently looks like this:

```
Public Function ListGroups() As IEnumerable(Of Group) Implements IContactManagerRepository.ListGroups

    throw New NotImplementedException()

End Function
```

The stub methods enabled us to compile our application and pass the unit tests. However, now it is time to actually implement these methods. The final version of the EntityContactManagerRepository class is contained in Listing 13.

Listing 13 - Models\EntityContactManagerRepository.vb

```
Public Class EntityContactManagerRepository
Implements IContactManagerRepository

Private _entities As New ContactManagerDBEntities()

' Contact methods

Public Function GetContact(ByVal id As Integer) As Contact Implements IContactManagerRepository.GetContact
    Return (From c In _entities.ContactSet.Include("Group") _
            Where c.Id = id _
            Select c).FirstOrDefault()
End Function

Public Function CreateContact(ByVal groupId As Integer, ByVal contactToCreate As Contact) As Contact
Implements IContactManagerRepository.CreateContact
    ' Associate group with contact
    contactToCreate.Group = GetGroup(groupId)

    ' Save new contact
    _entities.AddToContactSet(contactToCreate)
    _entities.SaveChanges()
    Return contactToCreate
End Function

Public Function EditContact(ByVal groupId As Integer, ByVal contactToEdit As Contact) As Contact Implements
IContactManagerRepository.EditContact
    ' Get original contact
    Dim originalContact = GetContact(contactToEdit.Id)

    ' Update with new group
    originalContact.Group = GetGroup(groupId)

    ' Save changes
    _entities.ApplyPropertyChanges(originalContact.EntityKey.EntitySetName, contactToEdit)
    _entities.SaveChanges()
    Return contactToEdit
End Function

Public Sub DeleteContact(ByVal contactToDelete As Contact) Implements IContactManagerRepository.DeleteContact
    Dim originalContact = GetContact(contactToDelete.Id)
    _entities.DeleteObject(originalContact)
    _entities.SaveChanges()
End Sub

    ' Group methods

Public Function CreateGroup(ByVal groupToCreate As Group) As Group Implements
IContactManagerRepository.CreateGroup
    _entities.AddToGroupSet(groupToCreate)
    _entities.SaveChanges()
    Return groupToCreate
End Function

Public Function ListGroups() As IEnumerable(Of Group) Implements IContactManagerRepository.ListGroups
    Return _entities.GroupSet.ToList()
End Function

Public Function GetFirstGroup() As Group Implements IContactManagerRepository.GetFirstGroup
    Return _entities.GroupSet.Include("Contacts").FirstOrDefault()
End Function

Public Function GetGroup(ByVal id As Integer) As Group Implements IContactManagerRepository.GetGroup
    Return (From g In _entities.GroupSet.Include("Contacts") _
            Where g.Id = id _
            Select g).FirstOrDefault()
End Function
```

```
Public Sub DeleteGroup(ByVal groupToDelete As Group) Implements IContactManagerRepository.DeleteGroup
    Dim originalGroup = GetGroup(groupToDelete.Id)
    _entities.DeleteObject(originalGroup)
    _entities.SaveChanges()
End Sub

End Class
```

Creating the Views

ASP.NET MVC application when you use the default ASP.NET view engine. So, you don t create views in response to a particular unit test. However, because an application would be useless without views, we can t complete this iteration without creating and modifying the views contained in the Contact Manager application.

We need to create the following new views for managing contact groups (see Figure 7):

- Views\Group\Index.aspx - Displays list of contact groups
- Views\Group\Delete.aspx - Displays confirmation form for deleting a contact group

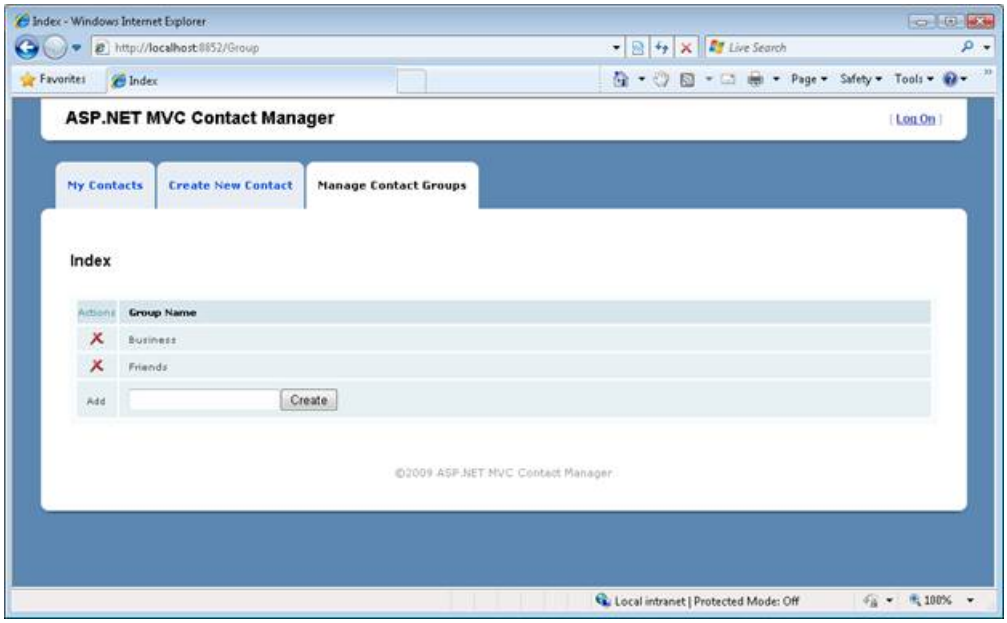


Figure 07: The Group Index view(Click to view full-size image)

We need to modify the following existing views so that they include contact groups:

- Views\Home\Create.aspx
- Views\Home\Edit.aspx
- Views\Home\Index.aspx

You can see the modified views by looking at the Visual Studio application that accompanies this tutorial. For example, Figure 8 illustrates the Contact Index view.

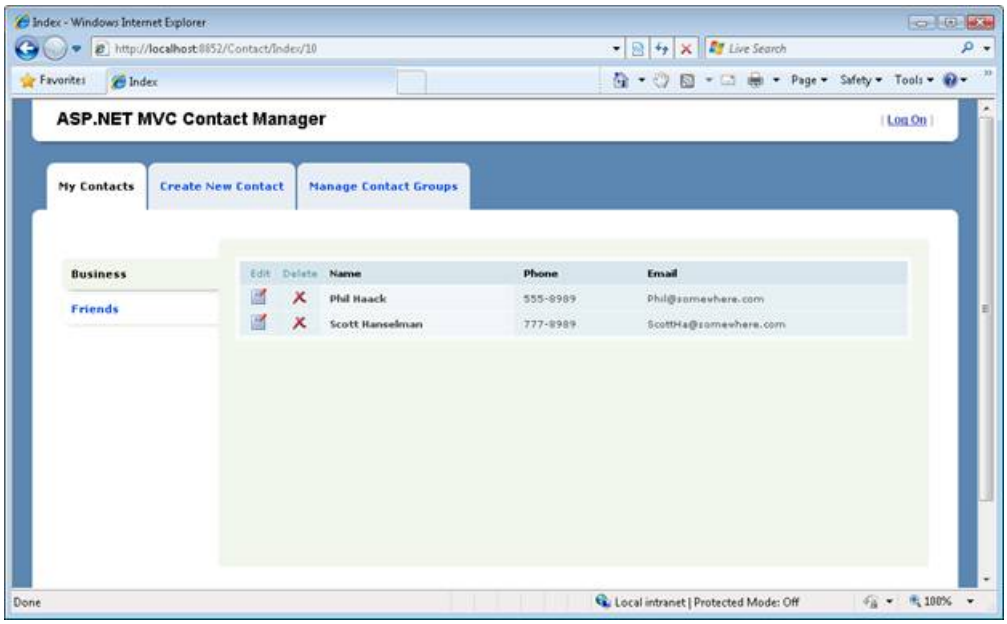


Figure 08: The Contact Index view(Click to view full-size image)

Summary

In this iteration, we added new functionality to our Contact Manager application by following a test-driven development application design methodology. We started by creating a set of user stories. We created a set of unit tests that corresponds to the requirements expressed by the user stories. Finally, we wrote just enough code to satisfy the requirements expressed by the unit tests.

After we finished writing enough code to satisfy the requirements expressed by the unit tests, we updated our database and views. We added a new Groups table to our database and updated our Entity Framework Data Model. We also created and modified a set of views.

In the next iteration -- the final iteration -- we rewrite our application to take advantage of Ajax. By taking advantage of Ajax, we'll improve the responsiveness and performance of the Contact Manager application.

PREVIOUS NEXT

Iteration #7 – Add Ajax functionality (VB)

4/10/2020 • 18 minutes to read • Edit Online

by Microsoft

Download Code

> In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

Building a Contact Management ASP.NET MVC Application (VB)

In this series of tutorials, we build an entire Contact Management application from start to finish. The Contact Manager application enables you to store contact information - names, phone numbers and email addresses - for a list of people.

We build the application over multiple iterations. With each iteration, we gradually improve the application. The goal of this multiple iteration approach is to enable you to understand the reason for each change.

- Iteration #1 - Create the application. In the first iteration, we create the Contact Manager in the simplest way possible. We add support for basic database operations: Create, Read, Update, and Delete (CRUD).
- Iteration #2 - Make the application look nice. In this iteration, we improve the appearance of the application by modifying the default ASP.NET MVC view master page and cascading style sheet.
- Iteration #3 - Add form validation. In the third iteration, we add basic form validation. We prevent people from submitting a form without completing required form fields. We also validate email addresses and phone numbers.
- Iteration #4 - Make the application loosely coupled. In this fourth iteration, we take advantage of several software design patterns to make it easier to maintain and modify the Contact Manager application. For example, we refactor our application to use the Repository pattern and the Dependency Injection pattern.
- Iteration #5 - Create unit tests. In the fifth iteration, we make our application easier to maintain and modify by adding unit tests. We mock our data model classes and build unit tests for our controllers and validation logic.
- Iteration #6 - Use test-driven development. In this sixth iteration, we add new functionality to our application by writing unit tests first and writing code against the unit tests. In this iteration, we add contact groups.
- Iteration #7 - Add Ajax functionality. In the seventh iteration, we improve the responsiveness and performance of our application by adding support for Ajax.

This Iteration

In this iteration of the Contact Manager application, we refactor our application to make use of Ajax. By taking advantage of Ajax, we make our application more responsive. We can avoid rendering an entire page when we need to update only a certain region in a page.

We'll refactor our Index view so that we don t need to redisplay the entire page whenever someone selects a new contact group. Instead, when someone clicks a contact group, we'll just update the list of contacts and leave the rest of the page alone.

We'll also change the way our delete link works. Instead of displaying a separate confirmation page, we'll display a JavaScript confirmation dialog. If you confirm that you want to delete a contact, an HTTP DELETE operation is performed against the server to delete the contact record from the database.

Furthermore, we will take advantage of jQuery to add animation effects to our Index view. We'll display an animation when the new list of contacts is being fetched from the server.

Finally, we'll take advantage of the ASP.NET AJAX framework support for managing browser history. We'll create history points whenever we perform an Ajax call to update the contact list. That way, the browser backward and forward buttons will work.

Why use Ajax?

Using Ajax has many benefits. First, adding Ajax functionality to an application results in a better user experience. In a normal web application, the entire page must be posted back to the server each and every time a user performs an action. Whenever you perform an action, the browser locks and the user must wait until the entire page is fetched and redisplayed.

This would be an unacceptable experience in the case of a desktop application. But, traditionally, we lived with this bad user experience in the case of a web application because we did not know that we could do any better. We thought it was a limitation of web applications when, in actuality, it was just a limitation of our imaginations.

In an Ajax application, you don t need to bring the user experience to a halt just to update a page. Instead, you can perform an asynchronous request in the background to update the page. You don t force the user to wait while part of the page gets updated.

By taking advantage of Ajax, you also can improve the performance of your application. Consider how the Contact Manager application works right now without Ajax functionality. When you click a contact group, the entire Index view must be redisplayed. The list of contacts and list of contact groups must be retrieved from the database server. All of this data must be passed across the wire from web server to web browser.

After we add Ajax functionality to our application, however, we can avoid redisplaying the entire page when a user clicks a contact group. We no longer need to grab the contact groups from the database. We also don t need to push the entire Index view across the wire. By taking advantage of Ajax, we reduce the amount of work that our database server must perform and we reduce the amount of network traffic required by our application.

Don t be Afraid of Ajax

Some developers avoid using Ajax because they worry about downlevel browsers. They want to make sure that their web applications will still work when accessed by a browser that does not support JavaScript. Because Ajax depends on JavaScript, some developers avoid using Ajax.

However, if you are careful about how you implement Ajax then you can build applications that work with both uplevel and downlevel browsers. Our Contact Manager application will work with browsers that support JavaScript and browsers that do not.

If you use the Contact Manager application with a browser that supports JavaScript then you will have a better user experience. For example, when you click a contact group, only the region of the page that displays contacts will be updated.

If, on the other hand, you use the Contact Manager application with a browser that does not support JavaScript (or that has JavaScript disabled) then you will have a slightly less desirable user experience. For example, when you click a contact group, the entire Index view must be posted back to the browser in order to display the matching list of contacts.

Adding the Required JavaScript Files

We'll need to use three JavaScript files to add Ajax functionality to our application. All three of these files are included in the Scripts folder of a new ASP.NET MVC application.

If you plan to use Ajax in multiple pages in your application then it makes sense to include the required JavaScript files in your application s view master page. That way, the JavaScript files will be included in all of the pages in your application automatically.

Add the following JavaScript includes inside the <head> tag of your view master page:

```
<script src="../../Scripts/MicrosoftAjax.js" type="text/javascript"></script>
    <script src="../../Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>
    <script src="../../Scripts/jquery-1.2.6.min.js" type="text/javascript"></script>
```

Refactoring the Index View to use Ajax

Let s start by modifying our Index view so that clicking a contact group only updates the region of the view that displays contacts. The red box in Figure 1 contains the region that we want to update.

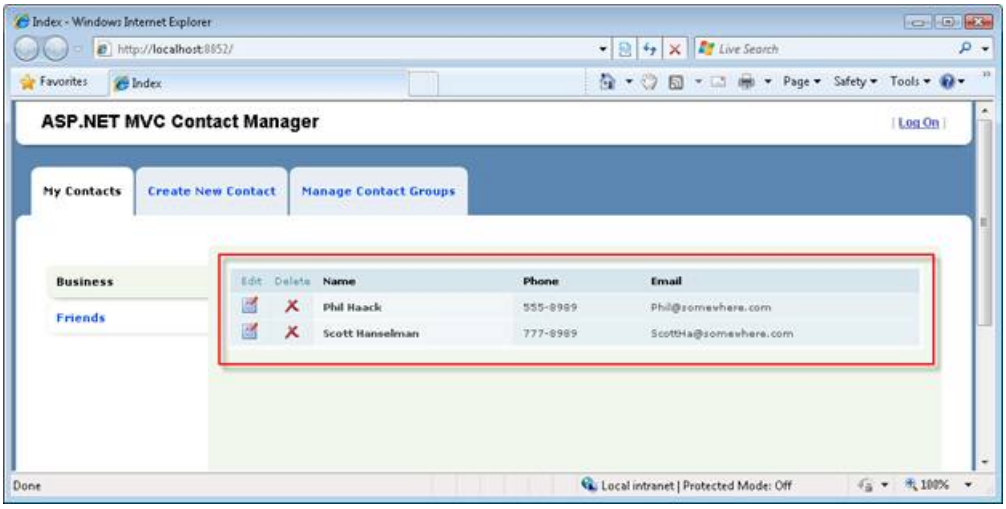


Figure 01: Updating only contacts(Click to view full-size image)

The first step is to separate the part of the view that we want to update asynchronously into a separate partial (view user control). The section of the Index view that displays the table of contacts has been moved into the partial in Listing 1.

Listing 1 - Views\Contact\ContactList.ascx

```
<%@ Control Language="VB" Inherits="System.Web.Mvc.ViewUserControl(Of ContactManager.Group)" %>
<table class="data-table" cellpadding="0" cellspacing="0">
    <thead>
        <tr>
            <th class="actions edit">
                Edit
            </th>
            <th class="actions delete">
                Delete
            </th>
            <th>
                Name
            </th>
            <th>
                Phone
            </th>
            <th>
                Email
            </th>
        </tr>
    </thead>
    <tbody>
        <% For Each item in Model.Contacts %>
        <tr>
            <td class="actions edit">
                <a href='<%= Url.Action("Edit", New With {.id=item.Id}) %>'><img src="../../Content/Edit.png"
alt="Edit" /></a>
            </td>
            <td class="actions delete">
                <a href='<%= Url.Action("Delete", New With {.id=item.Id}) %>'><img
src="../../Content/Delete.png" alt="Delete" /></a>
            </td>
            <th>
                <%= Html.Encode(item.FirstName) %>
                <%= Html.Encode(item.LastName) %>
            </th>
            <td>
                <%= Html.Encode(item.Phone) %>
            </td>
            <td>
                <%= Html.Encode(item.Email) %>
            </td>
        </tr>
        <% Next %>
    </tbody>
</table>
```

Notice that the partial in Listing 1 has a different model than the Index view. The *Inherits* attribute in the <%@ Page %> directive specifies that the partial inherits from the ViewUserControl<Group> class.

The updated Index view is contained in Listing 2.

Listing 2 - Views\Contact\Index.aspx

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of ContactManager.IndexModel)" %>
<%@ Import Namespace="ContactManager" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Index</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<ul id="leftColumn">
<% For Each item in Model.Groups %>
    <li <%= Html.Selected(item.Id, Model.SelectedGroup.Id) %>>
    <%= Ajax.ActionLink(item.Name, "Index", New With { .id = item.Id }, New AjaxOptions With { .UpdateTargetId
= "divContactList"})%>
    </li>
<% Next %>
</ul>
<div id="divContactList">
    <% Html.RenderPartial("ContactList", Model.SelectedGroup) %>
</div>

<div class="divContactList-bottom"> </div>
</asp:Content>
```

There are two things that you should notice about the updated view in Listing 2. First, notice that all of the content moved into the partial is replaced with a call to Html.RenderPartial(). The Html.RenderPartial() method is called when the Index view is first requested in order to display the initial set of contacts.

Second, notice that the Html.ActionLink() used to display contact groups has been replaced with an Ajax.ActionLink(). The Ajax.ActionLink() is called with the following parameters:

```
<%= Ajax.ActionLink(item.Name, "Index", New With { .id = item.Id }, New AjaxOptions With { .UpdateTargetId =
"divContactList"})%>
```

The first parameter represents the text to display for the link, the second parameter represents the route values, and the third parameter represents the Ajax options. In this case, we use the UpdateTargetId Ajax option to point to the HTML tag that we want to update after the Ajax request completes. We want to update the tag with the new list of contacts.

The updated Index() method of the Contact controller is contained in Listing 3.

Listing 3 - Controllers\ContactController.vb (Index method)

```
Public Function Index(ByVal id As Integer?) As ActionResult
    ' Get selected group
    Dim selectedGroup = _service.GetGroup(id)
    if IsNothing(selectedGroup) Then
        Return RedirectToAction("Index", "Group")
    End If

    ' Normal Request
    if Not Request.IsAjaxRequest() Then
        Dim model As new IndexModel With { _
            .Groups = _service.ListGroups(), _
            .SelectedGroup = selectedGroup _
        }
        Return View("Index", model)
    End If

    ' Ajax Request
    return PartialView("ContactList", selectedGroup)
End Function
```

The updated Index() action conditionally returns one of two things. If the Index() action is invoked by an Ajax request then the controller returns a partial. Otherwise, the Index() action returns an entire view.

Notice that the Index() action does not need to return as much data when invoked by an Ajax request. In the context of a normal request, the Index action returns a list of all of the contact groups and the selected contact group. In the context of an Ajax request, the Index() action returns only the selected group. Ajax means less work on your database server.

Our modified Index view works in the case of both uplevel and downlevel browsers. If you click a contact group, and your browser supports JavaScript, then only the region of the view that contains the list of contacts is updated. If, on the other hand, your browser does not support JavaScript, then the entire view is updated.

Our updated Index view has one problem. When you click a contact group, the selected group is not highlighted. Because the list of groups is displayed outside of the region that is updated during an Ajax request, the right group does not get highlighted. We'll fix this issue in the next section.

Adding jQuery Animation Effects

Normally, when you click a link in a web page, you can use the browser progress bar to detect whether or not the browser is actively fetching the updated content. When performing an Ajax request, on the other hand, the browser progress bar does not show any progress. This can make users nervous. How do you know whether the browser has frozen?

There are several ways that you can indicate to a user that work is being performed while performing an Ajax request. One approach is to display a simple animation. For example, you can fade out a region when an Ajax request begins and fade in the region when the request completes.

We'll use the jQuery library which is included with the Microsoft ASP.NET MVC framework, to create the animation effects. The updated Index view is contained in Listing 4.

Listing 4 - Views\Contact\Index.aspx

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of ContactManager.IndexModel)" %>
<%@ Import Namespace="ContactManager" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Index</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<script type="text/javascript">

    function beginContactList(args)
    {
        // Highlight selected group
        $('#leftColumn li').removeClass('selected');
        $(this).parent().addClass('selected');

        // Animate
        $('#divContactList').fadeOut('normal');
    }

    function successContactList()
    {
        // Animate
        $('#divContactList').fadeIn('normal');
    }

    function failureContactList()
    {
        alert("Could not retrieve contacts.");
    }

</script>

<ul id="leftColumn">
<% For Each item in Model.Groups %>
    <li <%= Html.Selected(item.Id, Model.SelectedGroup.Id) %>>
    <%= Ajax.ActionLink(item.Name, "Index", New With { .id = item.Id }, New AjaxOptions With { .UpdateTargetId
= "divContactList", .OnBegin = "beginContactList", .OnSuccess = "successContactList", .OnFailure =
"failureContactList" })%>
    </li>
<% Next %>
</ul>
<div id="divContactList">
    <% Html.RenderPartial("ContactList", Model.SelectedGroup) %>
</div>

<div class="divContactList-bottom"> </div>
</asp:Content>
```

Notice that the updated Index view contains three new JavaScript functions. The first two functions use jQuery to fade out and fade in the list of contacts when you click a new contact group. The third function displays an error message when an Ajax request results in an error (for example, network timeout).

The first function also takes care of highlighting the selected group. A class= selected attribute is added to the parent element (the LI element) of the element clicked. Again, jQuery makes it easy to select the right element and add the CSS class.

These scripts are tied to the group links with the help of the Ajax.ActionLink() AjaxOptions parameter. The updated Ajax.ActionLink() method call looks like this:

```
<%= Ajax.ActionLink(item.Name, "Index", New With { .id = item.Id }, New AjaxOptions With { .UpdateTargetId =
"divContactList", .OnBegin = "beginContactList", .OnSuccess = "successContactList", .OnFailure =
"failureContactList" })%>
```

Adding Browser History Support

Normally, when you click a link to update a page, your browser history is updated. That way, you can click the browser Back button to move back in time to the previous state of the page. For example, if you click the Friends contact group and then click the Business contact group, you can click the browser Back button to navigate back to the state of the page when the Friends contact group was selected.

Unfortunately, performing an Ajax request does not update browser history automatically. If you click a contact group, and the list of matching contacts is retrieved with an Ajax request, then the browser history is not updated. You cannot use the browser Back button to navigate back to a contact group after selecting a new contact group.

If you want users to be able to use the browser Back button after performing Ajax requests then you need to perform a little more work. You need to take advantage of the browser history management functionality built in the ASP.NET AJAX Framework.

ASP.NET AJAX browser history, you need to do three things:

1. Enable Browser History by setting the enableBrowserHistory property to true.
2. Save history points when the state of a view changes by calling the addHistoryPoint() method.
3. Reconstruct the state of the view when the navigate event is raised.

The updated Index view is contained in Listing 5.

Listing 5 - Views\Contact\Index.aspx

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of ContactManager.IndexModel)" %>
<%@ Import Namespace="ContactManager" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" runat="server">
<title>Index</title>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<script type="text/javascript">

    var _currentGroupId = -1;

    Sys.Application.add_init(pageInit);

    function pageInit() {
        // Enable history
        Sys.Application.set_enableHistory(true);

        // Add Handler for history
        Sys.Application.add_navigate(navigate);
    }

    function navigate(sender, e) {
        // Get groupId from address bar
        var groupId = e.get_state().groupId;

        // If groupId != currentGroupId then navigate
        if (groupId != _currentGroupId) {
            _currentGroupId = groupId;
            $("#divContactList").load("/Contact/Index/" + groupId);
            selectGroup(groupId);
```

```
        }
    }

    function selectGroup(groupId) {
        $('#leftColumn li').removeClass('selected');
        if (groupId)
            $('a[groupid=' + groupId + ']').parent().addClass('selected');
        else
            $('#leftColumn li:first').addClass('selected');
    }

    function beginContactList(args) {
        // Highlight selected group
        _currentGroupId = this.getAttribute("groupid");
        selectGroup(_currentGroupId);

        // Add history point
        Sys.Application.addHistoryPoint({ "groupId": _currentGroupId });

        // Animate
        $('#divContactList').fadeOut('normal');
    }

    function successContactList() {
        // Animate
        $('#divContactList').fadeIn('normal');
    }

    function failureContactList() {
        alert("Could not retrieve contacts.");
    }

</script>

<ul id="leftColumn">
<% For Each item in Model.Groups %>
    <li <%= Html.Selected(item.Id, Model.SelectedGroup.Id) %>>
    <%= Ajax.ActionLink(item.Name, "Index", New With { .id = item.Id }, New AjaxOptions With { .UpdateTargetId
= "divContactList", .OnBegin = "beginContactList", .OnSuccess = "successContactList", .OnFailure =
"failureContactList" }, New With { .groupid = item.Id })%>
    </li>
<% Next %>
</ul>
<div id="divContactList">
    <% Html.RenderPartial("ContactList", Model.SelectedGroup) %>
</div>

<div class="divContactList-bottom"> </div>
</asp:Content>
```

In Listing 5, Browser History is enabled in the pageInit() function. The pageInit() function is also used to set up the event handler for the navigate event. The navigate event is raised whenever the browser Forward or Back button causes the state of the page to change.

The beginContactList() method is called when you click a contact group. This method creates a new history point by calling the addHistoryPoint() method. The id of the contact group clicked is added to history.

The group id is retrieved from an expando attribute on the contact group link. The link is rendered with the following call to Ajax.ActionLink().

```
<%= Ajax.ActionLink(item.Name, "Index", New With { .id = item.Id }, New AjaxOptions With { .UpdateTargetId =
"divContactList", .OnBegin = "beginContactList", .OnSuccess = "successContactList", .OnFailure =
"failureContactList" }, New With { .groupid = item.Id })%>
```

The last parameter passed to the Ajax.ActionLink() adds an expando attribute named groupid to the link (lowercase

for XHTML compatibility).

When a user hits the browser Back or Forward button, the navigate event is raised and the navigate() method is called. This method updates the contacts displayed in the page to match the state of the page that corresponds to the browser history point passed to the navigate method.

Performing Ajax Deletes

Currently, in order to delete a contact, you need to click on the Delete link and then click the Delete button displayed in the delete confirmation page (see Figure 2). This seems like a lot of page requests to do something simple like deleting a database record.

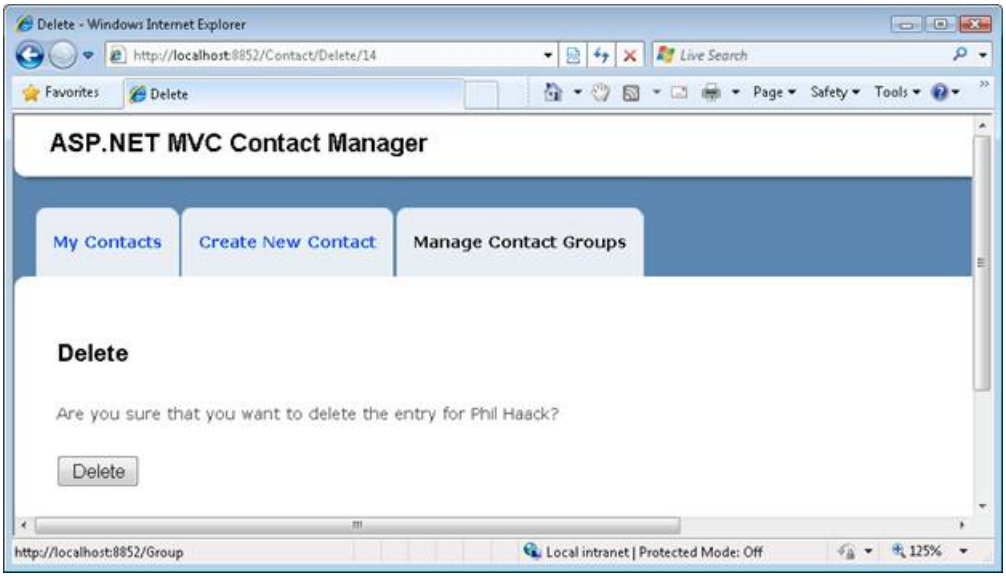


Figure 02: The delete confirmation page(Click to view full-size image)

It is tempting to skip the delete confirmation page and delete a contact directly from the Index view. You should avoid this temptation because taking this approach opens your application to security holes. In general, you don t want to perform an HTTP GET operation when invoking an action that modifies the state of your web application. When performing a delete, you want to perform an HTTP POST, or better yet, an HTTP DELETE operation.

The Delete link is contained in the ContactList partial. An updated version of the ContactList partial is contained in Listing 6.

Listing 6 - Views\Contact\ContactList.ascx

```
<%@ Control Language="VB" Inherits="System.Web.Mvc.ViewUserControl(Of ContactManager.Group)" %>
<%@ Import Namespace="ContactManager" %>
<table class="data-table" cellpadding="0" cellspacing="0">
    <thead>
        <tr>
            <th class="actions edit">
                Edit
            </th>
            <th class="actions delete">
                Delete
            </th>
            <th>
                Name
            </th>
            <th>
                Phone
            </th>
            <th>
                Email
            </th>
        </tr>
    </thead>
    <tbody>
        <% For Each item in Model.Contacts %>
        <tr>
            <td class="actions edit">
                <a href='<%= Url.Action("Edit", New With {.id=item.Id}) %>'><img src="../../Content/Edit.png"
alt="Edit" /></a>
            </td>
            <td class="actions delete">
                <%= Ajax.ImageActionLink("../../Content/Delete.png", "Delete", "Delete", New with { .id =
item.Id }, New AjaxOptions With { .Confirm = "Delete contact?", .HttpMethod = "Delete", .UpdateTargetId =
"divContactList" })%>
            </td>
            <th>
                <%= Html.Encode(item.FirstName) %>
                <%= Html.Encode(item.LastName) %>
            </th>
            <td>
                <%= Html.Encode(item.Phone) %>
            </td>
            <td>
                <%= Html.Encode(item.Email) %>
            </td>
        </tr>
        <% Next %>
    </tbody>
</table>
```

The Delete link is rendered with the following call to the Ajax.ImageActionLink() method:

```
<%= Ajax.ImageActionLink("../../Content/Delete.png", "Delete", "Delete", New with { .id = item.Id }, New
AjaxOptions With { .Confirm = "Delete contact?", .HttpMethod = "Delete", .UpdateTargetId = "divContactList"
})%<
```

NOTE

The Ajax.ImageActionLink() is not a standard part of the ASP.NET MVC framework. The Ajax.ImageActionLink() is a custom helper methods included in the Contact Manager project.

The AjaxOptions parameter has two properties. First, the Confirm property is used to display a popup JavaScript confirmation dialog. Second, the HttpMethod property is used to perform an HTTP DELETE operation.

Listing 7 contains a new AjaxDelete() action that has been added to the Contact controller.

Listing 7 - Controllers\ContactController.vb (AjaxDelete)

```
<AcceptVerbs(HttpVerbs.Delete), ActionName("Delete")> _
Public Function AjaxDelete(ByVal id As Integer) As ActionResult
    ' Get contact and group
    Dim contactToDelete = _service.GetContact(id)
    Dim selectedGroup = _service.GetGroup(contactToDelete.Group.Id)

    ' Delete from database
    _service.DeleteContact(contactToDelete)

    ' Return Contact List
    Return PartialView("ContactList", selectedGroup)
End Function
```

The AjaxDelete() action is decorated with an AcceptVerbs attribute. This attribute prevents the action from being invoked except by any HTTP operation other than an HTTP DELETE operation. In particular, you cannot invoke this action with an HTTP GET.

After you delete database record, you need to display the updated list of contacts that does not contain the deleted record. The AjaxDelete() method returns the ContactList partial and the updated list of contacts.

Summary

In this iteration, we added Ajax functionality to our Contact Manager application. We used Ajax to improve the responsiveness and performance of our application.

First, we refactored the Index view so that clicking a contact group does not update the entire view. Instead, clicking a contact group only updates the list of contacts.

Next, we used jQuery animation effects to fade out and fade in the list of contacts. Adding animation to an Ajax application can be used to provide users of the application with the equivalent of a browser progress bar.

We also added browser history support to our Ajax application. We enabled users to click the browser Back and Forward buttons to change the state of the Index view.

Finally, we created a delete link that supports HTTP DELETE operations. By performing Ajax deletes, we enable users to delete database records without requiring the user to request an additional delete confirmation page.

PREVIOUS

NerdDinner

2/20/2020 • 2 minutes to read • Edit Online

How to build the NerdDinner.com website using ASP.NET MVC 1. For the ASP.NET MVC version, please visit NerdDinner on GitHub.

- Introducing the NerdDinner Tutorial
- Create a New ASP.NET MVC Project
- Create a Database
- Build a Model with Business Rule Validations
- Use Controllers and Views to Implement a Listing/Details UI
- Provide CRUD (Create, Read, Update, Delete) Data Form Entry Support
- Use ViewData and Implement ViewModel Classes
- Re-use UI Using Master Pages and Partials
- Implement Efficient Data Paging
- Secure Applications Using Authentication and Authorization
- Use AJAX to Deliver Dynamic Updates
- Use AJAX to Implement Mapping Scenarios
- Enable Automated Unit Testing
- NerdDinner Wrap Up

Introducing the NerdDinner Tutorial

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

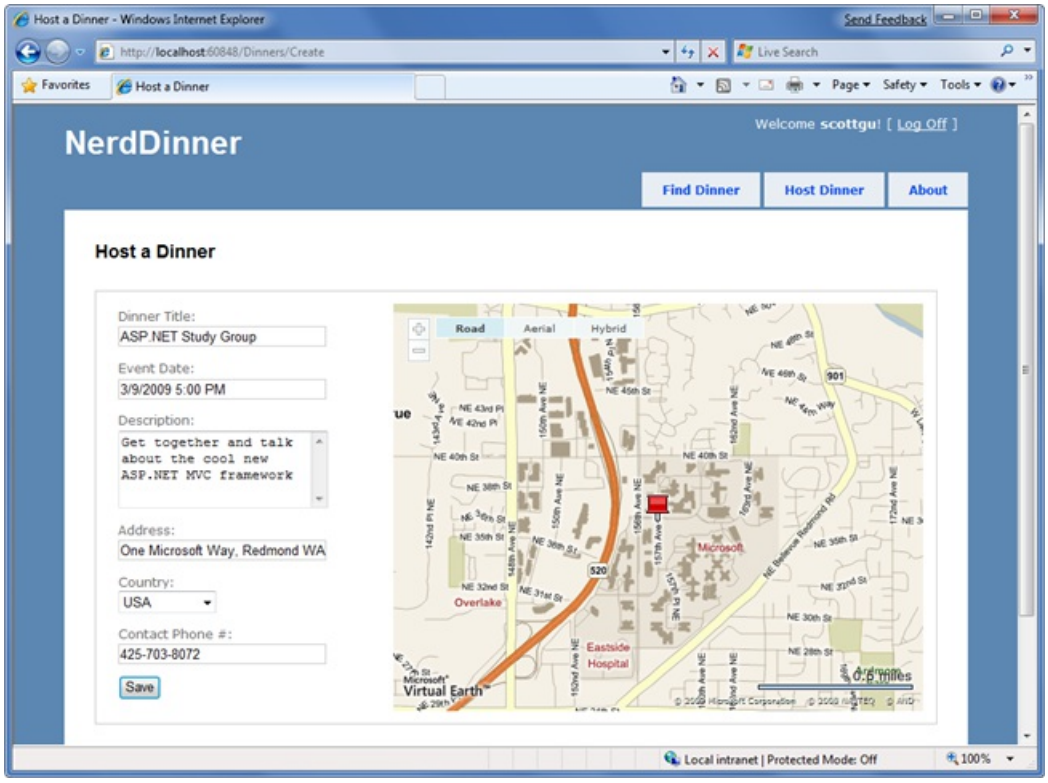
Download PDF

> The best way to learn a new framework is to build something with it. This tutorial walks through how to build a small, but complete, application using ASP.NET MVC 1, and introduces some of the core concepts behind it.
>
> If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

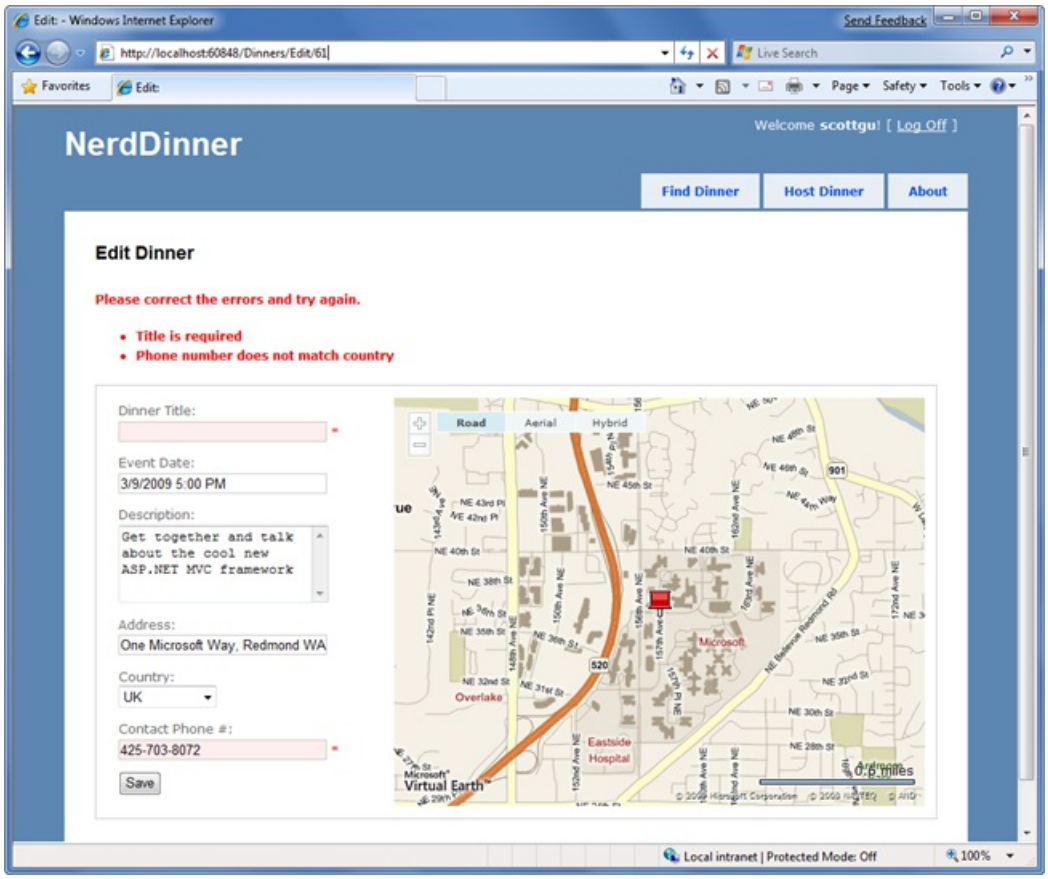
NerdDinner Tutorial

The best way to learn a new framework is to build something with it. This tutorial walks through how to build a small, but complete, application using ASP.NET MVC, and introduces some of the core concepts behind it.

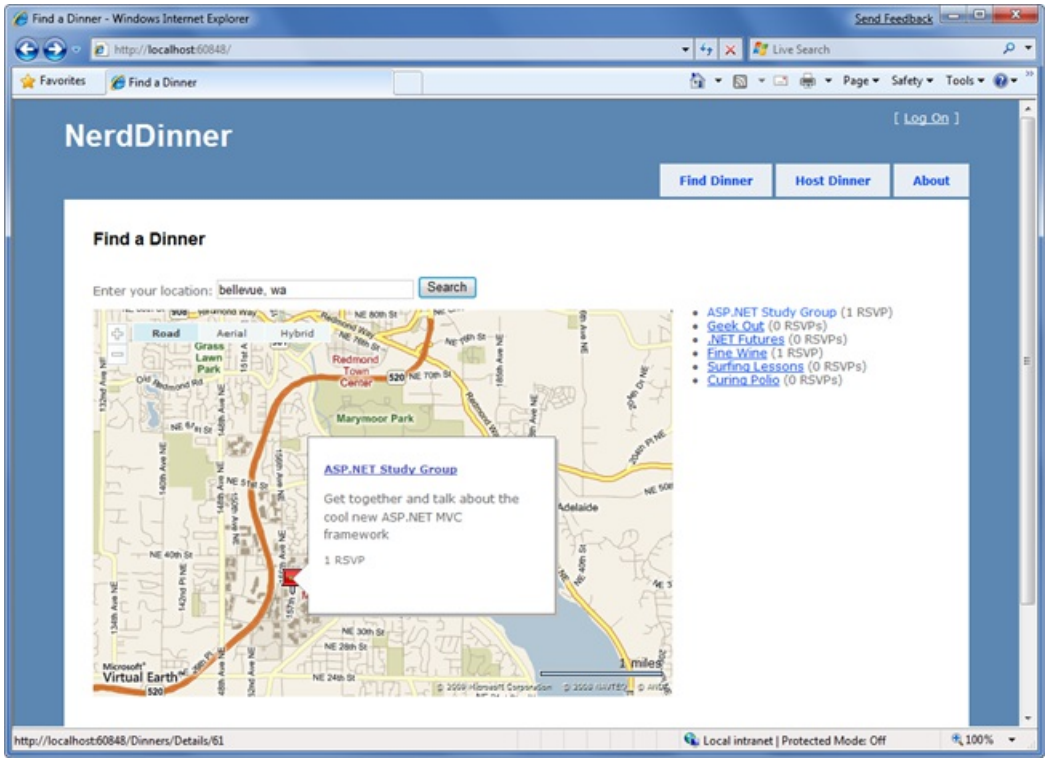
The application we are going to build is called "NerdDinner". NerdDinner provides an easy way for people to find and organize dinners online:



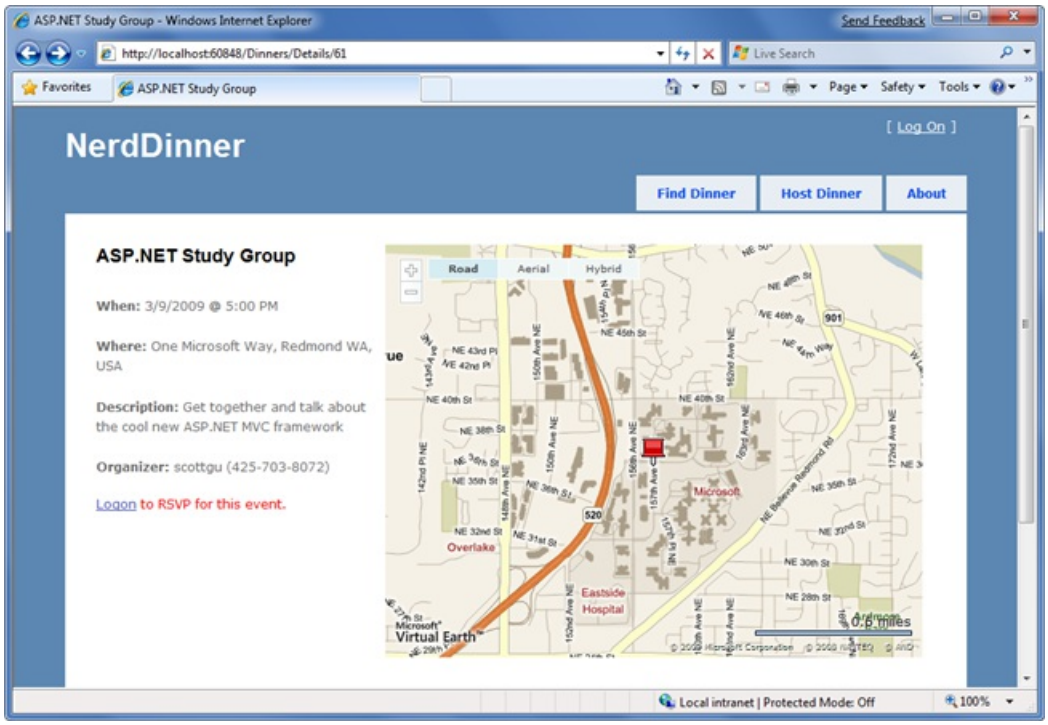
NerdDinner enables registered users to create, edit and delete dinners. It enforces a consistent set of validation and business rules across the application:



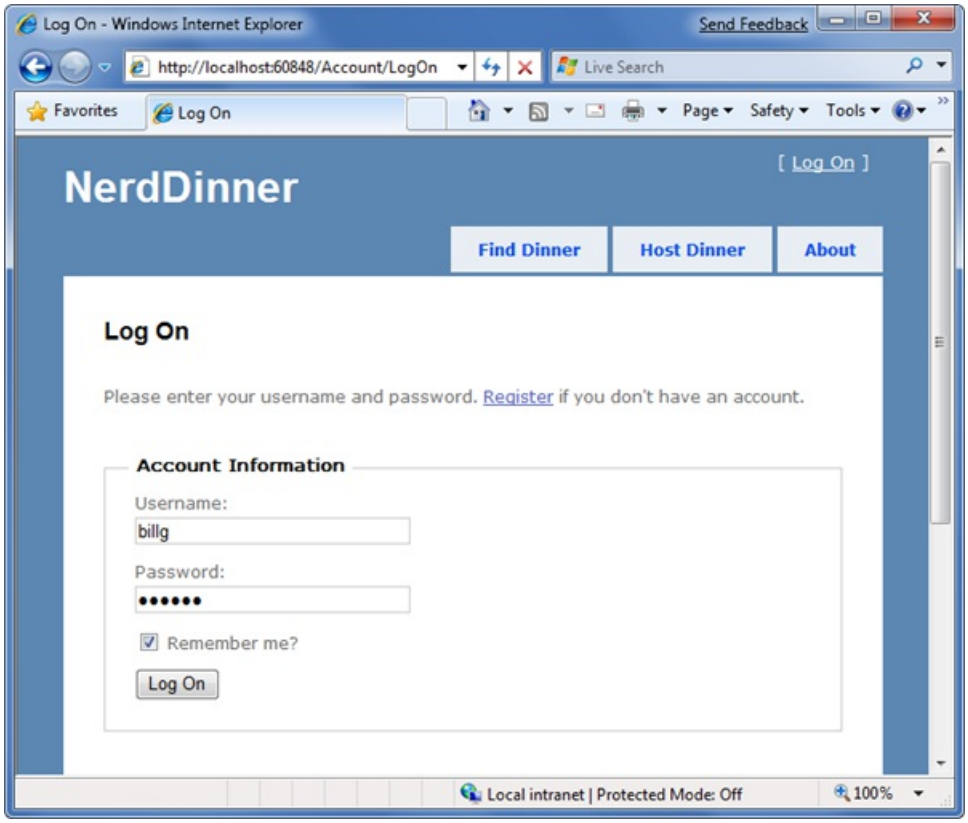
Visitors can use an AJAX-based map to search for upcoming dinners being held near them:



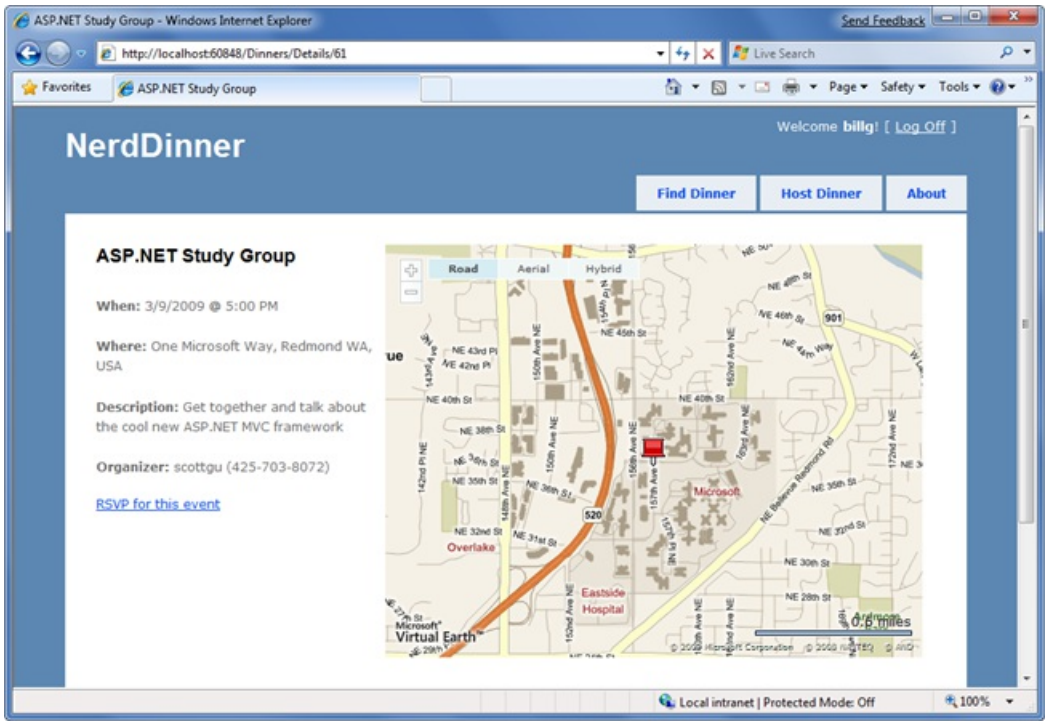
Clicking a dinner will take them to a details page where they can learn more about it:

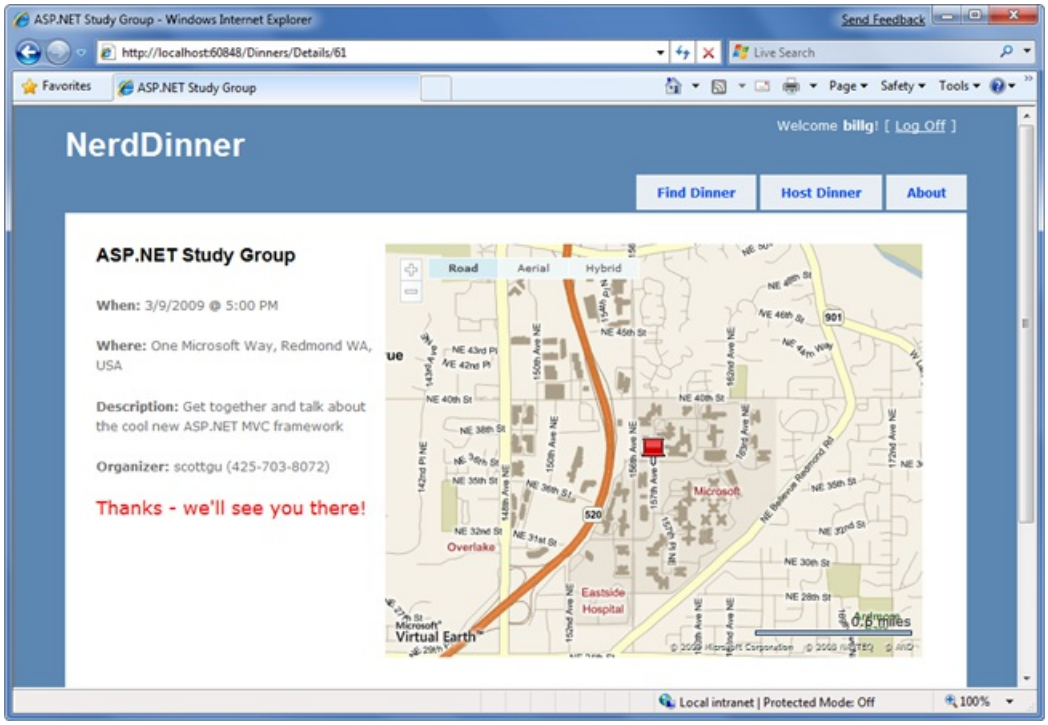


If they are interested in attending the dinner they can login or register on the site:



They can then click an AJAX-based RSVP link to attend the event:





Implementing NerdDinner

We are going to begin our NerdDinner application by using the File->New Project command within Visual Studio to create a brand new ASP.NET MVC project. We will then incrementally add functionality and features. Along the way we'll cover:

1. How to create a new ASP.NET MVC Project
2. How to create a database
3. How to build a model with business rule validations
4. How to use controllers and views to implement a listing/details UI
5. How to provide CRUD (create, read, update, delete) data form entry support
6. How to use ViewData and implement ViewModel classes
7. How to re-use UI using master pages and partials
8. How to implement efficient data paging
9. How to secure applications using authentication and authorization

10. How to use AJAX to deliver dynamic updates
11. How to use AJAX to implement mapping scenarios
12. How to enable automated unit testing

You can build your own copy of NerdDinner from scratch by completing each step we walkthrough in this chapter. Alternatively, you can download a completed version of the source code here: NerdDinner on GitHub. You can also optionally also download a free PDF version of this tutorial if you want to read the tutorial offline.

You can use either Visual Studio 2008 or the free Visual Web Developer 2008 Express to build the application. You can use either SQL Server or the free SQL Server Express for the database.

You can install ASP.NET MVC, Visual Web Developer 2008 Express, and SQL Server Express (all free) using V2 of the Microsoft Web Platform Installer

Now let's get started....

Now that we've covered what NerdDinner is, let's roll up our sleeves and write some code.

We'll begin by using File->New Project within Visual Studio to create the NerdDinner application.

NEXT

Create a New ASP.NET MVC Project

4/10/2020 • 4 minutes to read • Edit Online

by Microsoft

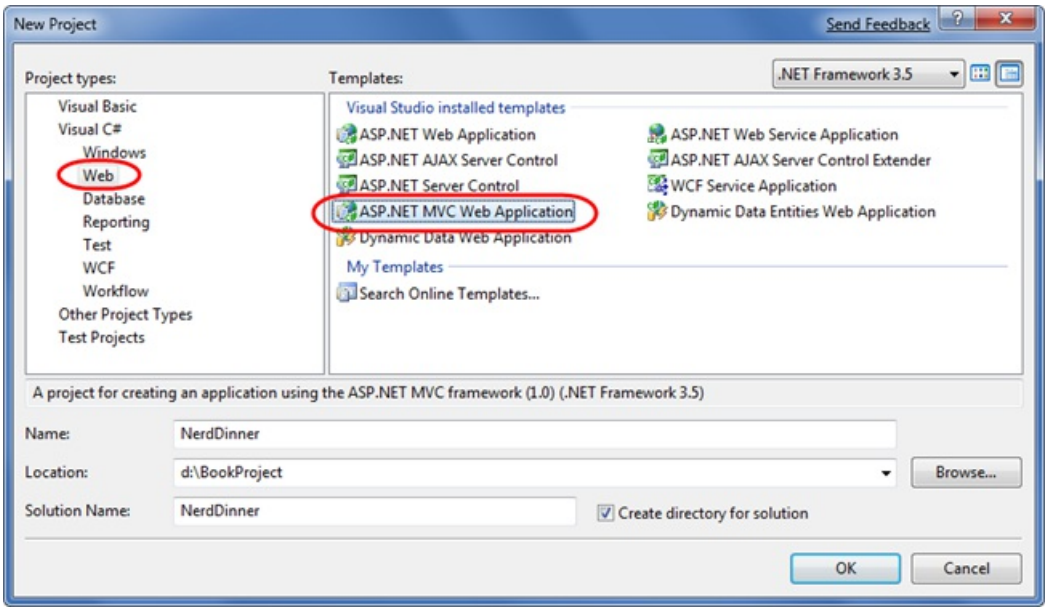
Download PDF

> This is step 1 of a free "NerdDinner" application tutorial that walks-through how to build a small, but complete, web application using ASP.NET MVC 1.
>
> Step 1 shows you how to put the basic NerdDinner application structure in place.
>
> If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

NerdDinner Step 1: File->New Project

We'll begin our NerdDinner application by selecting the **File->New Project** menu item within either Visual Studio 2008 or the free Visual Web Developer 2008 Express.

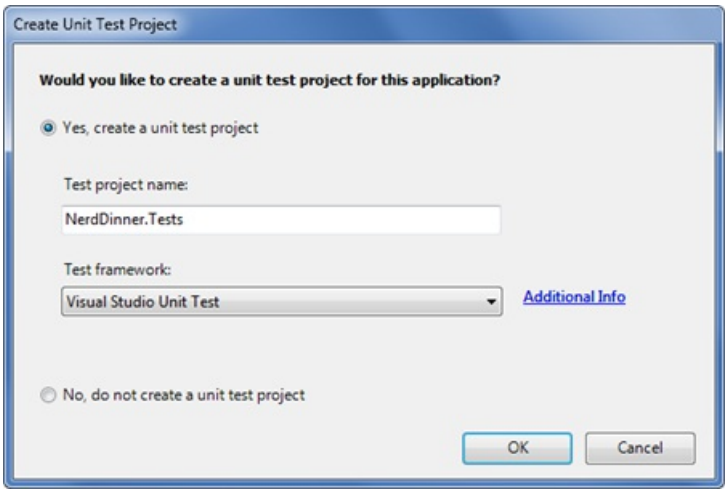
This will bring up the "New Project" dialog. To create a new ASP.NET MVC application, we'll select the "Web" node on the left-hand side of the dialog and then choose the "ASP.NET MVC Web Application" project template on the right:



Important: Make sure you've downloaded and installed ASP.NET MVC - otherwise it won't show up in the New Project dialog. You can use V2 of the Microsoft Web Platform Installer if you haven't installed it yet (ASP.NET MVC is available within the "Web Platform->Frameworks and Runtimes" section).

We'll name the new project we are going to create "NerdDinner" and then click the "ok" button to create it.

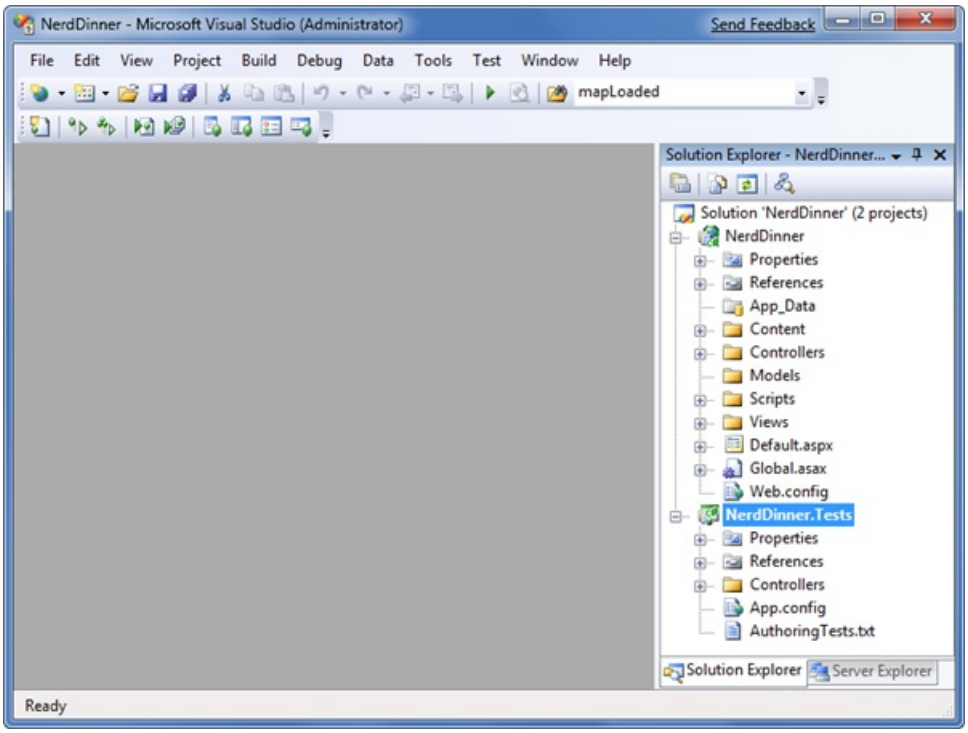
When we click "ok" Visual Studio will bring up an additional dialog that prompts us to optionally create a unit test project for the new application as well. This unit test project enables us to create automated tests that verify the functionality and behavior of our application (something we'll cover how to-do later in this tutorial).



The "Test framework" dropdown in the above dialog is populated with all available ASP.NET MVC unit test project templates installed on the machine. Versions can be downloaded for NUnit, MBUnit, and XUnit. The built-in Visual Studio Unit Test framework is also supported.

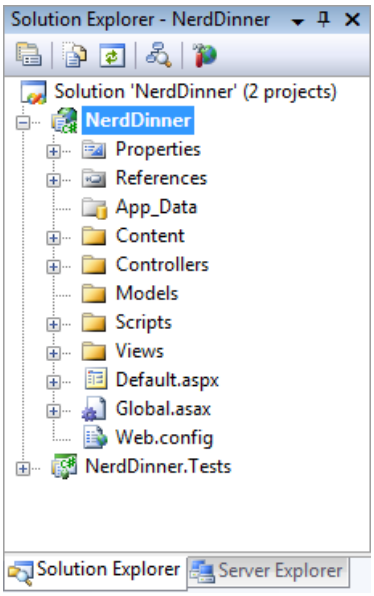
Note: The Visual Studio Unit Test Framework is only available with Visual Studio 2008 Professional and higher versions. If you are using VS 2008 Standard Edition or Visual Web Developer 2008 Express you will need to download and install the NUnit, MBUnit or XUnit extensions for ASP.NET MVC in order for this dialog to be shown. The dialog will not display if there aren't any test frameworks installed.

We'll use the default "NerdDinner.Tests" name for the test project we create, and use the "Visual Studio Unit Test" framework option. When we click the "ok" button Visual Studio will create a solution for us with two projects in it - one for our web application and one for our unit tests:



Examining the NerdDinner directory structure

When you create a new ASP.NET MVC application with Visual Studio, it automatically adds a number of files and directories to the project:

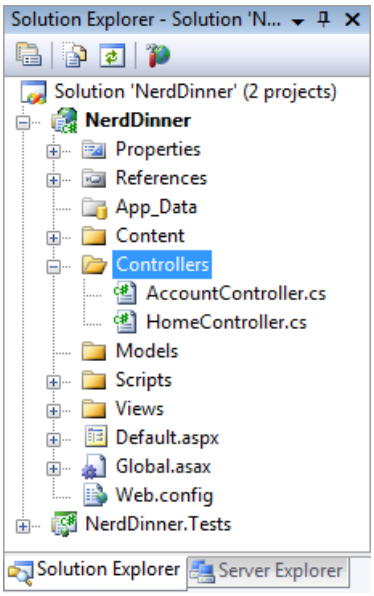


ASP.NET MVC projects by default have six top-level directories:

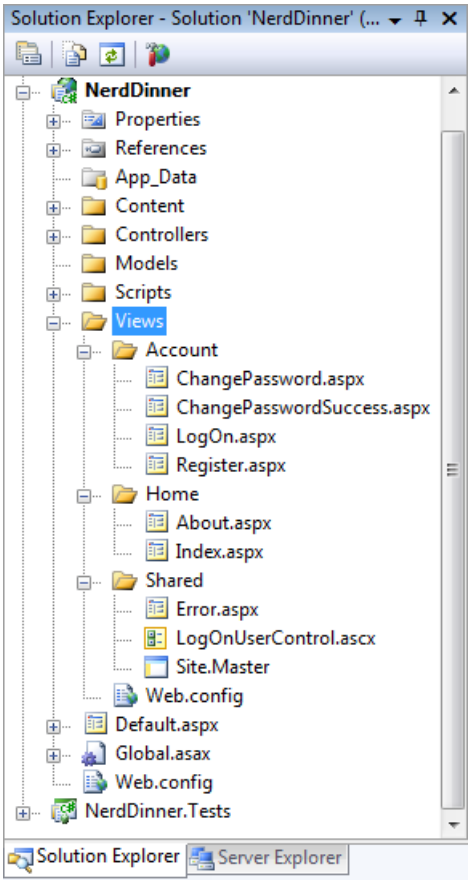
DIRECTORY	PURPOSE
/Controllers	Where you put Controller classes that handle URL requests
/Models	Where you put classes that represent and manipulate data
/Views	Where you put UI template files that are responsible for rendering output
/Scripts	Where you put JavaScript library files and scripts (.js)
/Content	Where you put CSS and image files, and other non-dynamic/non-JavaScript content
/App_Data	Where you store data files you want to read/write.

ASP.NET MVC does not require this structure. In fact, developers working on large applications will typically partition the application up across multiple projects to make it more manageable (for example: data model classes often go in a separate class library project from the web application). The default project structure, however, does provide a nice default directory convention that we can use to keep our application concerns clean.

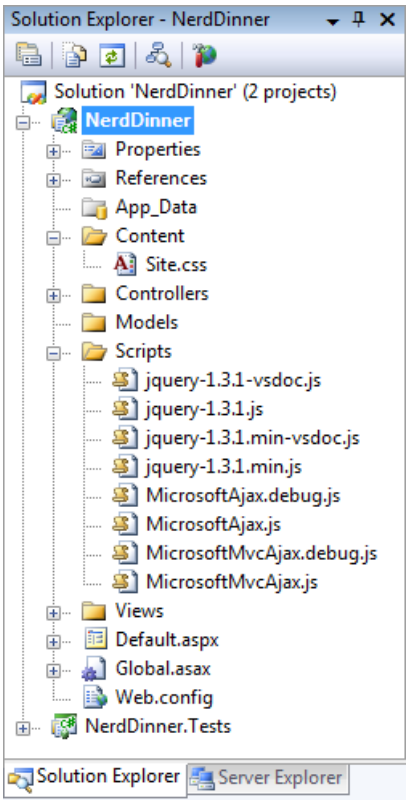
When we expand the /Controllers directory we'll find that Visual Studio added two controller classes – HomeController and AccountController – by default to the project:



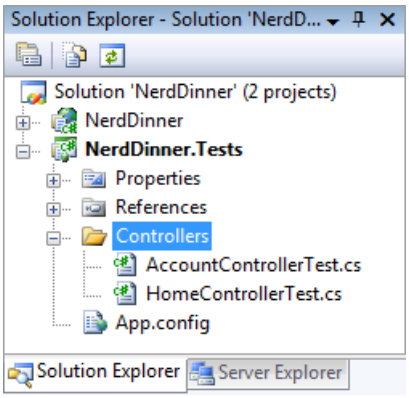
When we expand the /Views directory, we'll find three sub-directories – /Home, /Account and /Shared – as well as several template files within them were also added to the project by default:



When we expand the /Content and /Scripts directories, we'll find a Site.css file that is used to style all HTML on the site, as well as JavaScript libraries that can enable ASP.NET AJAX and jQuery support within the application:



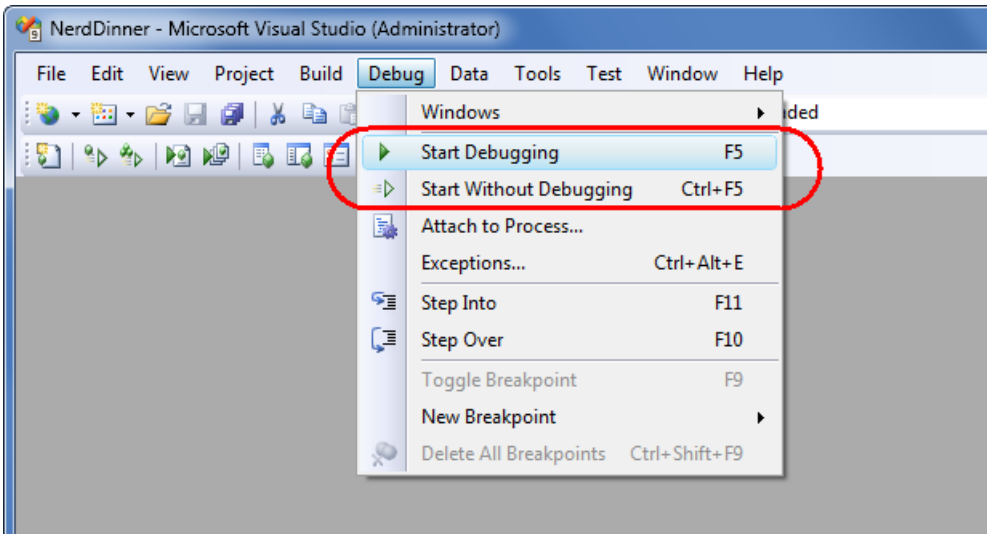
When we expand the NerdDinner.Tests project we'll find two classes that contain unit tests for our controller classes:



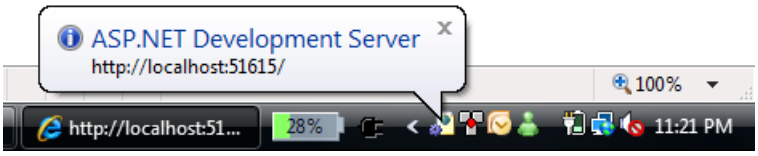
These default files added by Visual Studio provide us with a basic structure for a working application - complete with home page, about page, account login/logout/registration pages, and an unhandled error page (all wired-up and working out of the box).

Running the NerdDinner Application

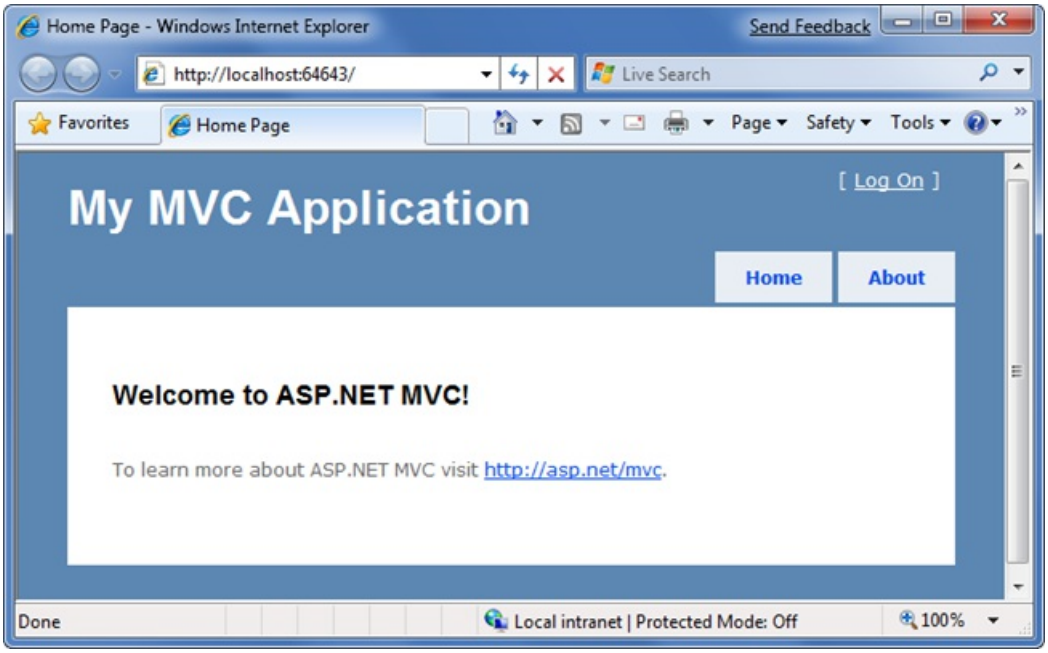
We can run the project by choosing either the **Debug->Start Debugging** or **Debug->Start Without Debugging** menu items:



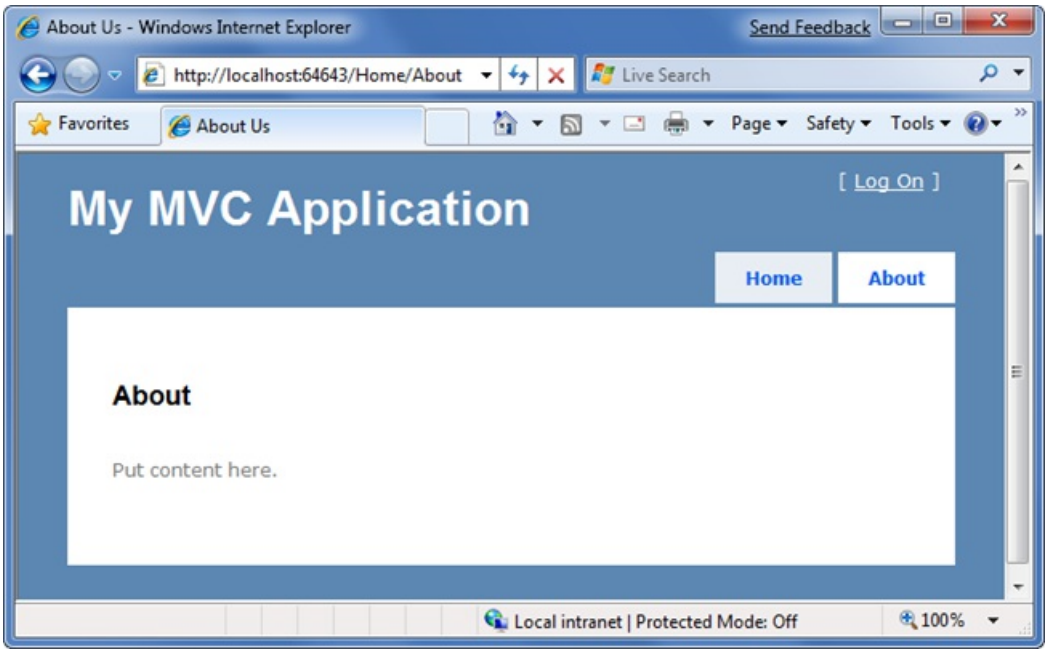
This will launch the built-in ASP.NET Web-server that comes with Visual Studio, and run our application:



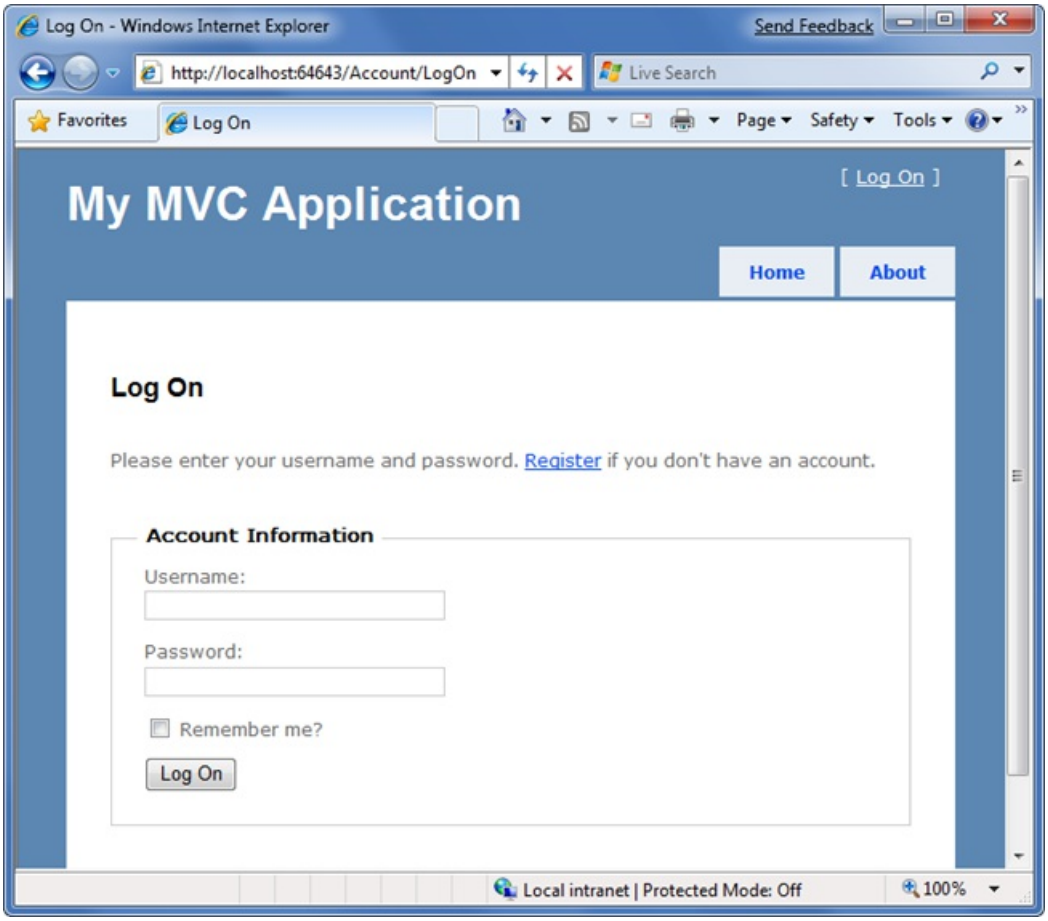
Below is the home page for our new project (URL: "/") when it runs:



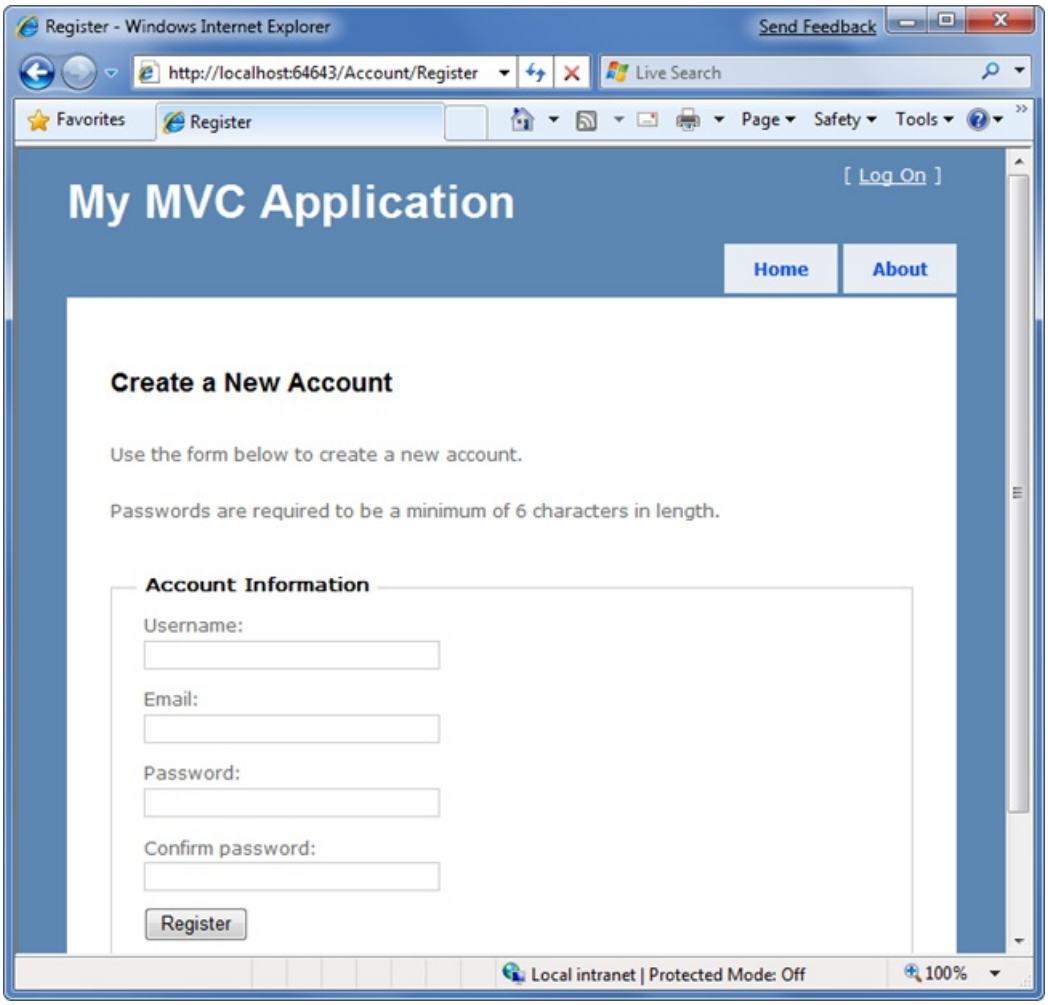
Clicking the "About" tab displays an about page (URL: "/Home/About"):



Clicking the "Log On" link on the top-right takes us to a Login page (URL: "/Account/LogOn")



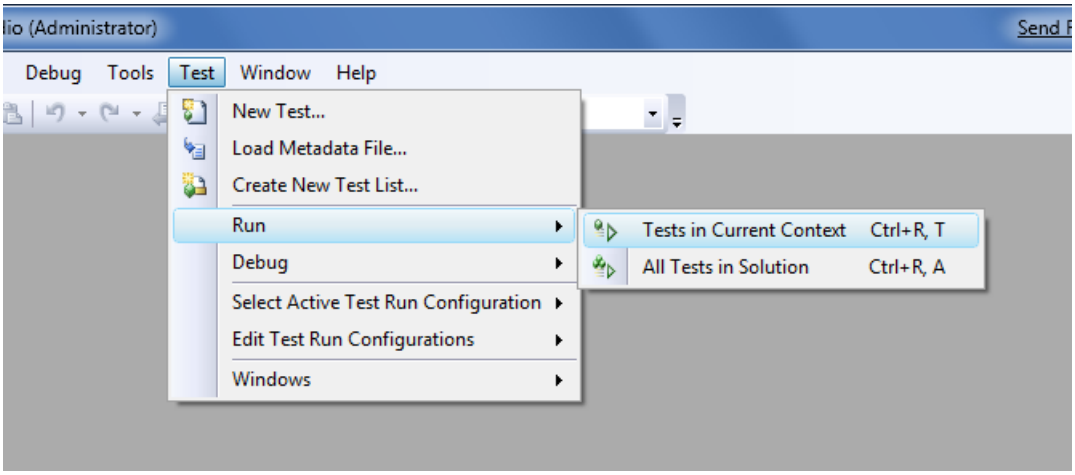
If we don't have a login account we can click the register link (URL: "/Account/Register") to create one:



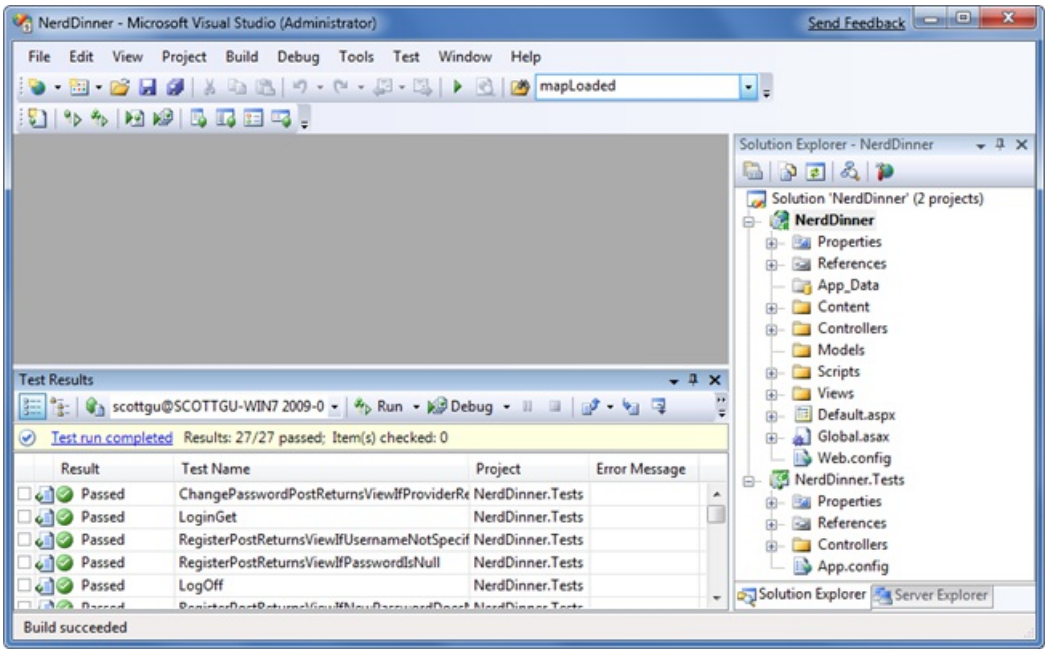
The code to implement the above home, about, and logout/ register functionality was added by default when we created our new project. We'll use it as the starting point of our application.

Testing the NerdDinner Application

If we are using the Professional Edition or higher version of Visual Studio 2008, we can use the built-in unit testing IDE support within Visual Studio to test the project:



Choosing one of the above options will open the "Test Results" pane within the IDE and provide us with pass/fail status on the 27 unit tests included in our new project that cover the built-in functionality:



Later in this tutorial we'll talk more about automated testing and add additional unit tests that cover the application functionality we implement.

Next Step

We've now got a basic application structure in place. Let's now create a database to store our application data.

PREVIOUS NEXT

Create a Database

4/10/2020 • 5 minutes to read • Edit Online

by Microsoft

Download PDF

This is step 2 of a free "NerdDinner" application tutorial that walks-through how to build a small, but complete, web application using ASP.NET MVC 1.

Step 2 shows the steps to create the database holding all of the dinner and RSVP data for our NerdDinner application.

If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

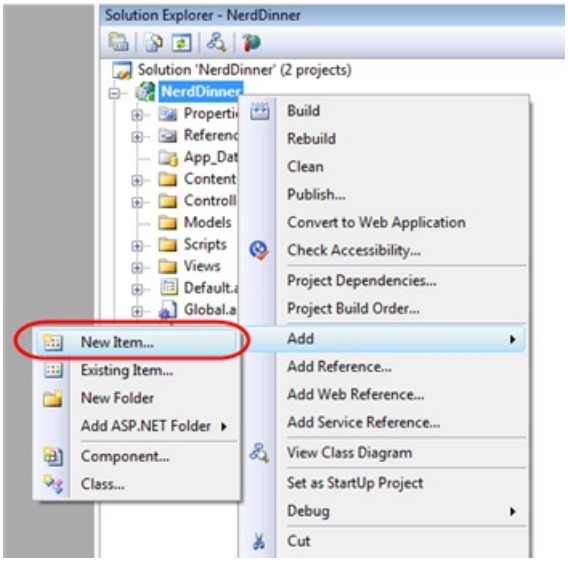
NerdDinner Step 2: Creating the Database

We'll be using a database to store all of the Dinner and RSVP data for our NerdDinner application.

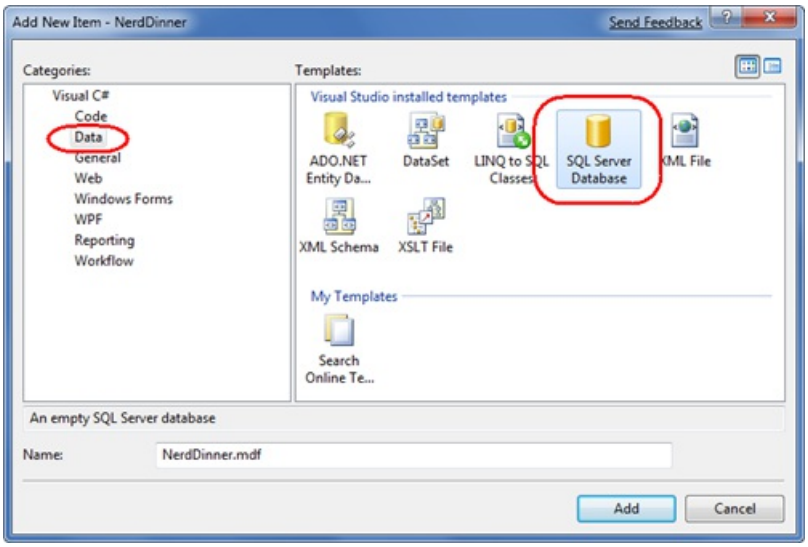
The steps below show creating the database using the free SQL Server Express edition (which you can easily install using V2 of the Microsoft Web Platform Installer). All of the code we'll write works with both SQL Server Express and the full SQL Server.

Creating a new SQL Server Express database

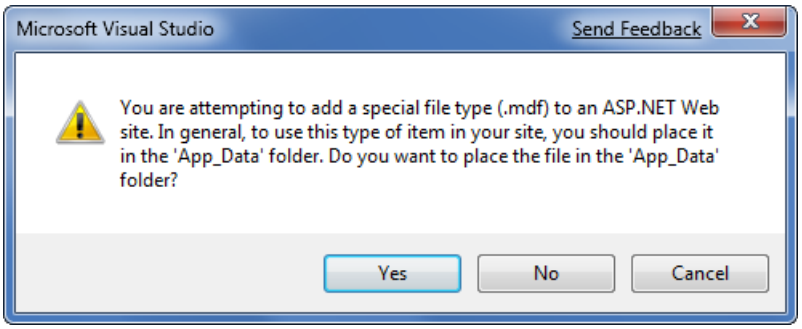
We'll begin by right-clicking on our web project, and then select the **Add->New Item** menu command:



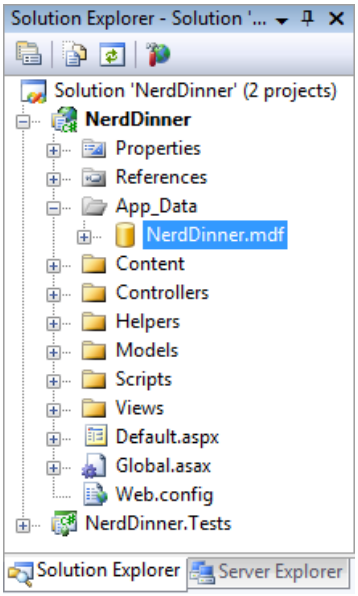
This will bring up Visual Studio's "Add New Item" dialog. We'll filter by the "Data" category and select the "SQL Server Database" item template:



We'll name the SQL Server Express database we want to create "NerdDinner.mdf" and hit ok. Visual Studio will then ask us if we want to add this file to our \App_Data directory (which is a directory already setup with both read and write security ACLs):



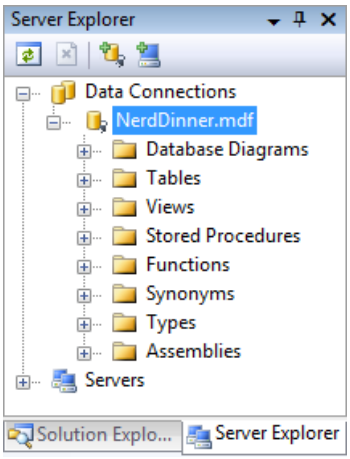
We'll click "Yes" and our new database will be created and added to our Solution Explorer:



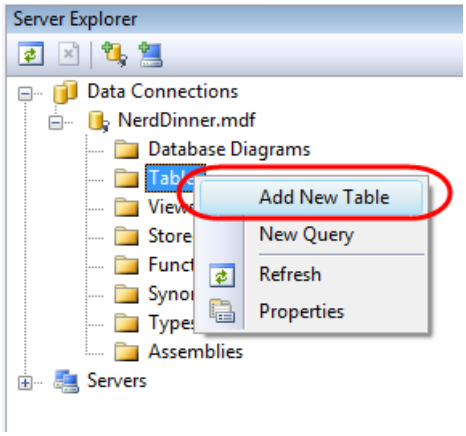
Creating Tables within our Database

We now have a new empty database. Let's add some tables to it.

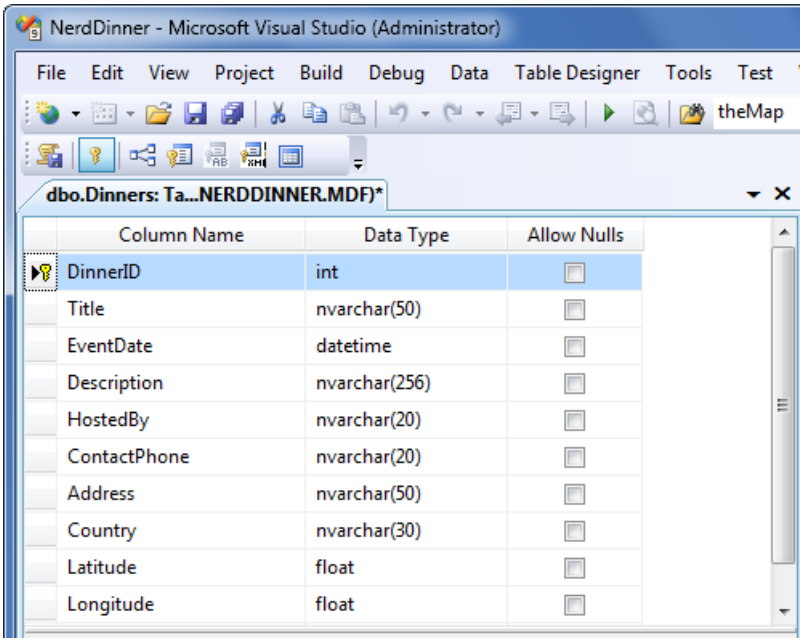
To do this we'll navigate to the "Server Explorer" tab window within Visual Studio, which enables us to manage databases and servers. SQL Server Express databases stored in the \App_Data folder of our application will automatically show up within the Server Explorer. We can optionally use the "Connect to Database" icon on the top of the "Server Explorer" window to add additional SQL Server databases (both local and remote) to the list as well:



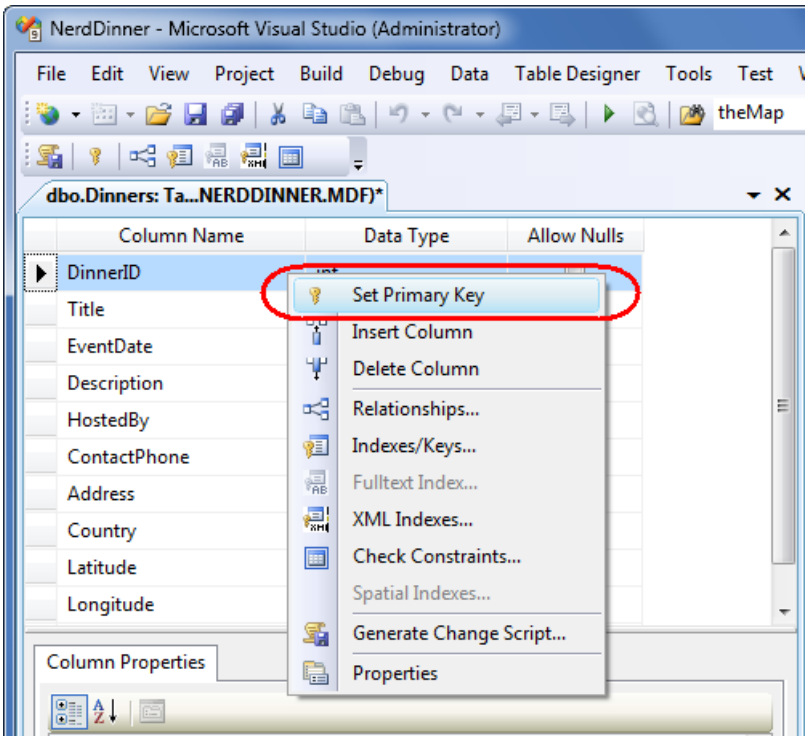
We will add two tables to our NerdDinner database – one to store our Dinners, and the other to track RSVP acceptances to them. We can create new tables by right-clicking on the "Tables" folder within our database and choosing the "Add New Table" menu command:



This will open up a table designer that allows us to configure the schema of our table. For our "Dinners" table we will add 10 columns of data:

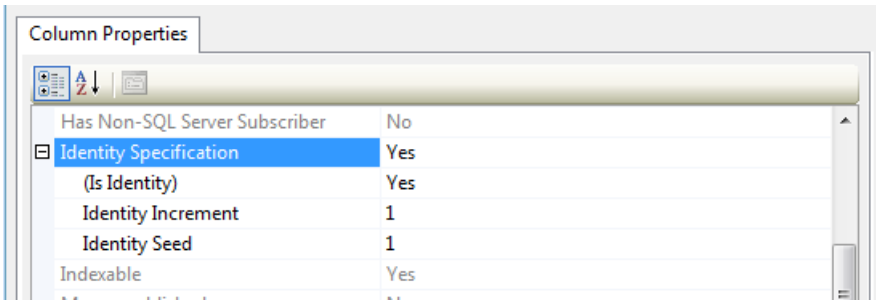


We want the "DinnerID" column to be a unique primary key for the table. We can configure this by right-clicking on the "DinnerID" column and choosing the "Set Primary Key" menu item:

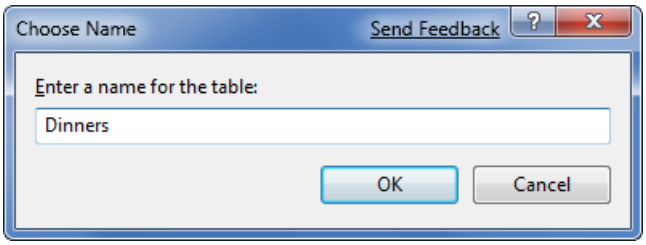


In addition to making DinnerID a primary key, we also want configure it as an "identity" column whose value is automatically incremented as new rows of data are added to the table (meaning the first inserted Dinner row will have a DinnerID of 1, the second inserted row will have a DinnerID of 2, etc).

We can do this by selecting the "DinnerID" column and then use the "Column Properties" editor to set the "(Is Identity)" property on the column to "Yes". We will use the standard identity defaults (start at 1 and increment 1 on each new Dinner row):

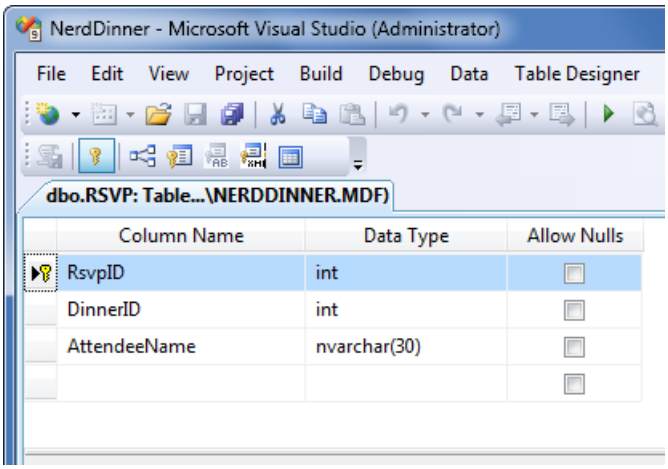


We'll then save our table by typing Ctrl-S or by using the **File->Save** menu command. This will prompt us to name the table. We'll name it "Dinners":



Our new Dinners table will then show up within our database in the server explorer.

We'll then repeat the above steps and create a "RSVP" table. This table with have 3 columns. We will setup the RsvpID column as the primary key, and also make it an identity column:

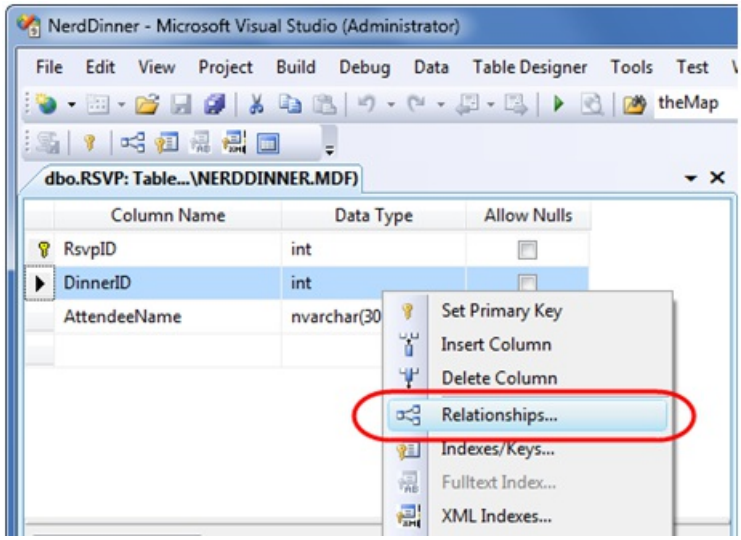


We'll save it and give it the name "RSVP".

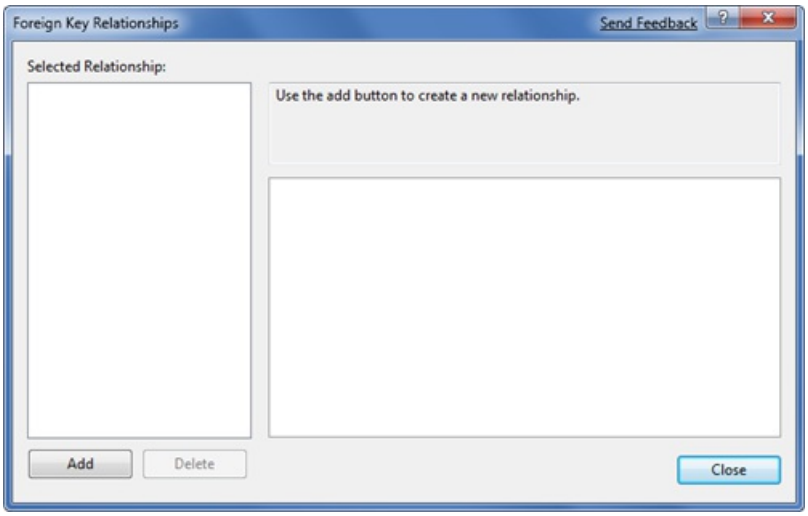
Setting up a Foreign Key Relationship between Tables

We now have two tables within our database. Our last schema design step will be to setup a "one-to-many" relationship between these two tables – so that we can associate each Dinner row with zero or more RSVP rows that apply to it. We will do this by configuring the RSVP table's "DinnerID" column to have a foreign-key relationship to the "DinnerID" column in the "Dinners" table.

To do this we'll open up the RSVP table within the table designer by double-clicking it in the server explorer. We'll then select the "DinnerID" column within it, right-click, and choose the "Relationships..." context menu command:

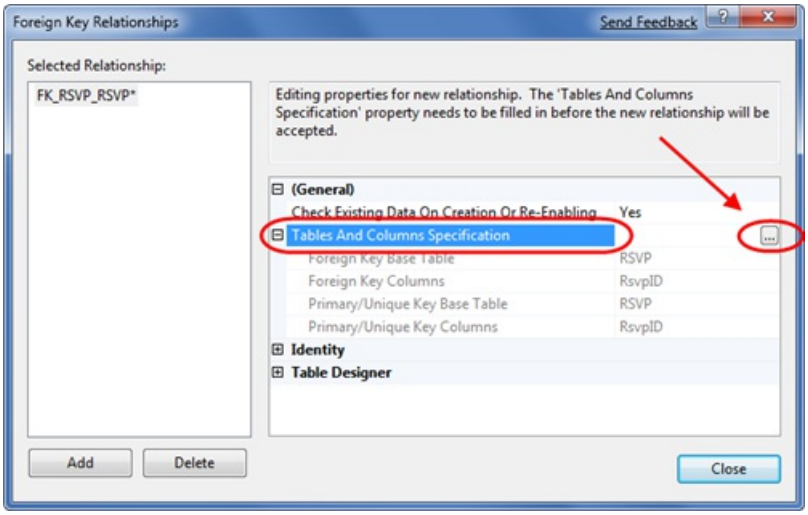


This will bring up a dialog that we can use to setup relationships between tables:



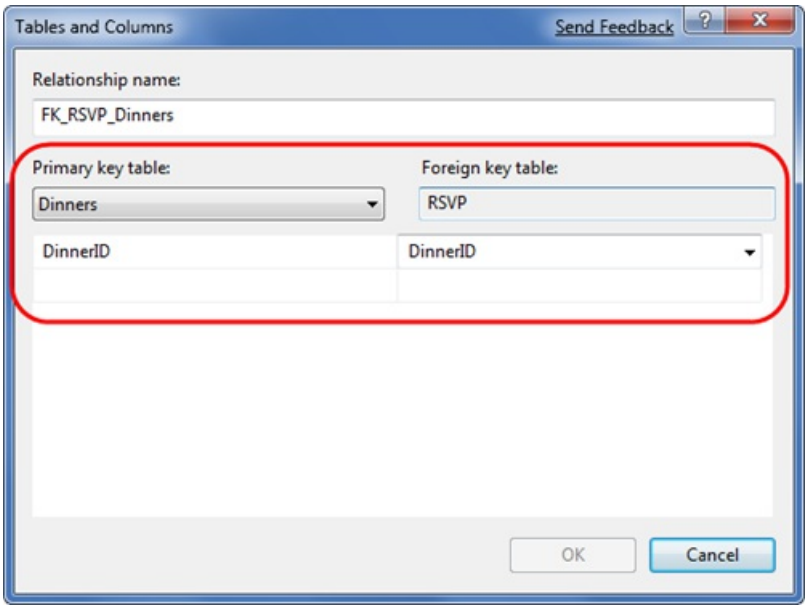
We'll click the "Add" button to add a new relationship to the dialog. Once a relationship has been added, we'll

expand the "Tables and Column Specification" tree-view node within the property grid to the right of the dialog, and then click the "..." button to the right of it:



Clicking the "..." button will bring up another dialog that allows us to specify which tables and columns are involved in the relationship, as well as allow us to name the relationship.

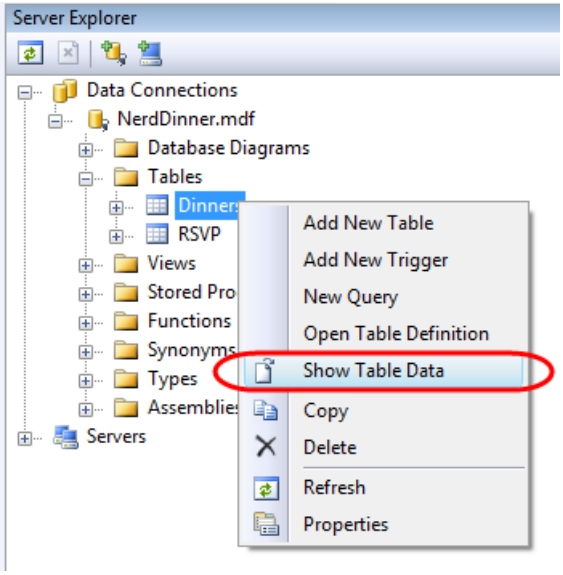
We will change the Primary Key Table to be "Dinners", and select the "DinnerID" column within the Dinners table as the primary key. Our RSVP table will be the foreign-key table, and the RSVP.DinnerID column will be associated as the foreign-key:



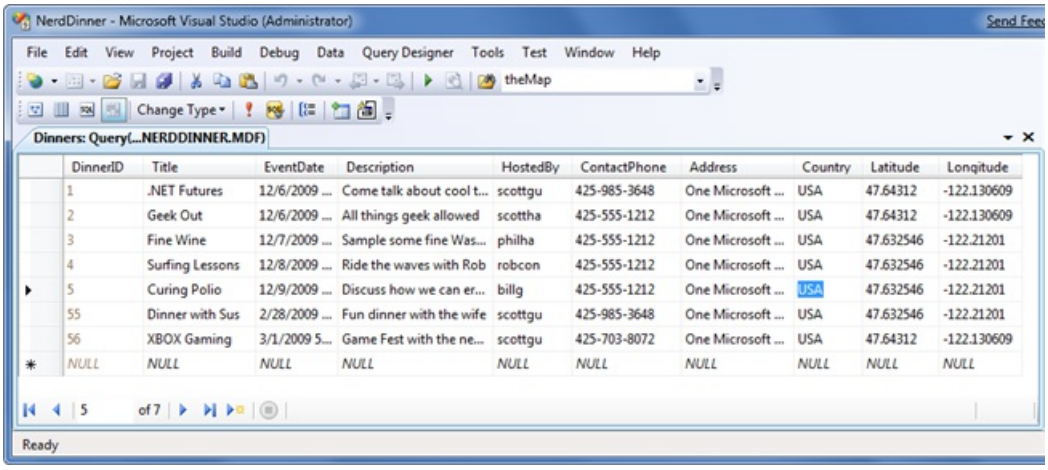
Now each row in the RSVP table will be associated with a row in the Dinner table. SQL Server will maintain referential integrity for us – and prevent us from adding a new RSVP row if it does not point to a valid Dinner row. It will also prevent us from deleting a Dinner row if there are still RSVP rows referring to it.

Adding Data to our Tables

Let's finish by adding some sample data to our Dinners table. We can add data to a table by right-clicking on it within the Server Explorer and choosing the "Show Table Data" command:



We'll add a few rows of Dinner data that we can use later as we start implementing the application:



Next Step

We've finished creating our database. Let's now create model classes that we can use to query and update it.

PREVIOUS NEXT

Build a Model with Business Rule Validations

4/10/2020 • 17 minutes to read • Edit Online

by Microsoft

Download PDF

> This is step 3 of a free "NerdDinner" application tutorial that walks-through how to build a small, but complete, web application using ASP.NET MVC 1.
>
> Step 3 shows how to create a model that we can use to both query and update the database for our NerdDinner application.
>
> If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

NerdDinner Step 3: Building the Model

In a model-view-controller framework the term "model" refers to the objects that represent the data of the application, as well as the corresponding domain logic that integrates validation and business rules with it. The model is in many ways the "heart" of an MVC-based application, and as we'll see later fundamentally drives the behavior of it.

The ASP.NET MVC framework supports using any data access technology, and developers can choose from a variety of rich .NET data options to implement their models including: LINQ to Entities, LINQ to SQL, NHibernate, LLBLGen Pro, SubSonic, WilsonORM, or just raw ADO.NET DataReaders or DataSets.

For our NerdDinner application we are going to use LINQ to SQL to create a simple model that corresponds fairly closely to our database design, and adds some custom validation logic and business rules. We will then implement a repository class that helps abstract away the data persistence implementation from the rest of the application, and enables us to easily unit test it.

LINQ to SQL

LINQ to SQL is an ORM (object relational mapper) that ships as part of .NET 3.5.

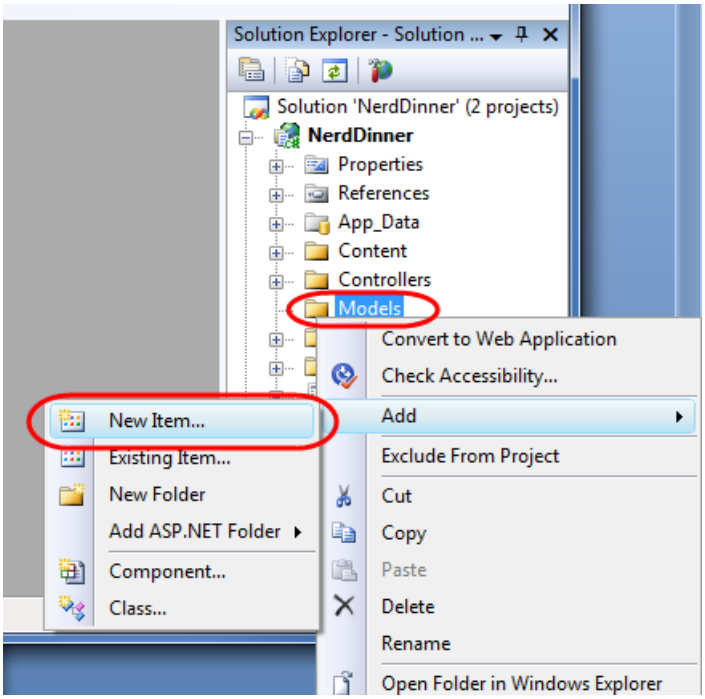
LINQ to SQL provides an easy way to map database tables to .NET classes we can code against. For our NerdDinner application we'll use it to map the Dinners and RSVP tables within our database to Dinner and RSVP classes. The columns of the Dinners and RSVP tables will correspond to properties on the Dinner and RSVP classes. Each Dinner and RSVP object will represent a separate row within the Dinners or RSVP tables in the database.

LINQ to SQL allows us to avoid having to manually construct SQL statements to retrieve and update Dinner and RSVP objects with database data. Instead, we'll define the Dinner and RSVP classes, how they map to/from the database, and the relationships between them. LINQ to SQL will then takes care of generating the appropriate SQL execution logic to use at runtime when we interact and use them.

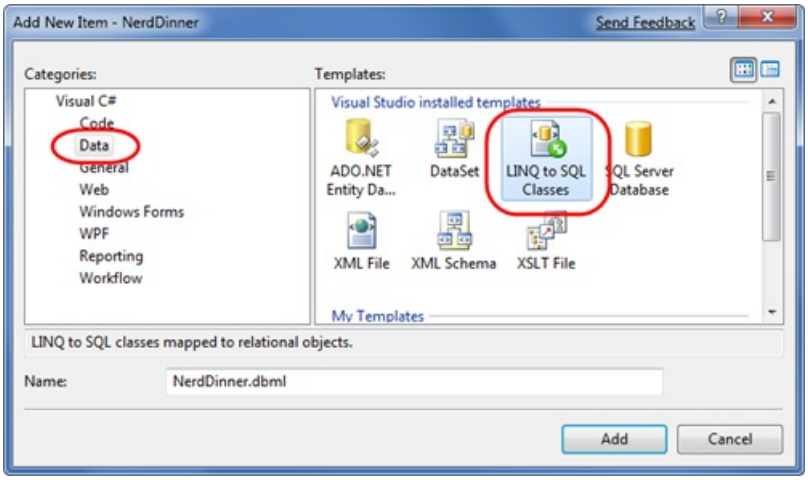
We can use the LINQ language support within VB and C# to write expressive queries that retrieve Dinner and RSVP objects from the database. This minimizes the amount of data code we need to write, and allows us to build really clean applications.

Adding LINQ to SQL Classes to our project

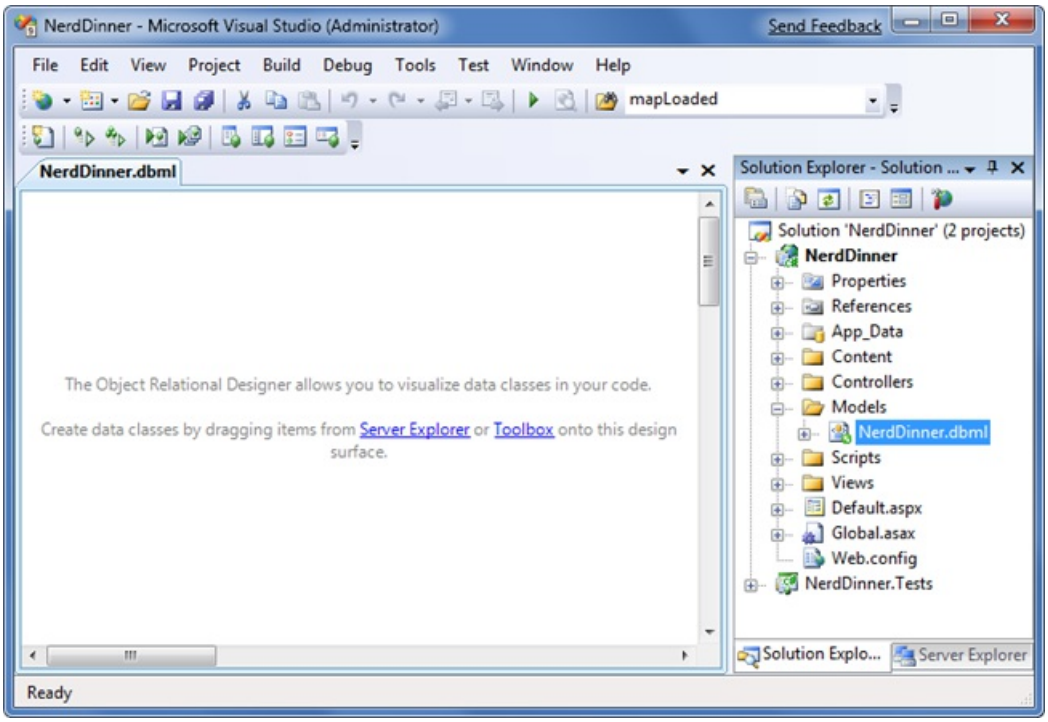
We'll begin by right-clicking on the "Models" folder within our project, and select the **Add->New Item** menu command:



This will bring up the "Add New Item" dialog. We'll filter by the "Data" category and select the "LINQ to SQL Classes" template within it:

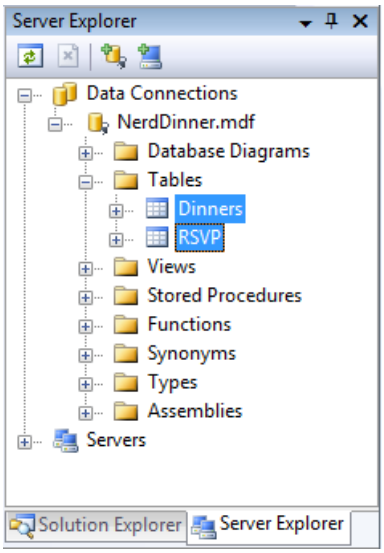


We'll name the item "NerdDinner" and click the "Add" button. Visual Studio will add a NerdDinner.dbml file under our \Models directory, and then open the LINQ to SQL object relational designer:

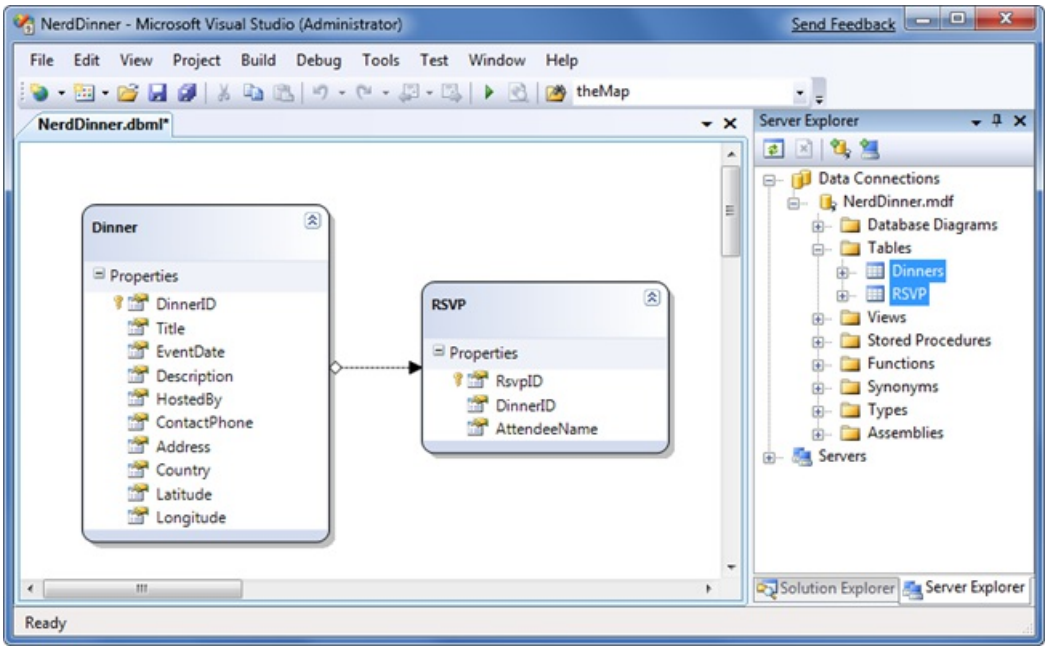


Creating Data Model Classes with LINQ to SQL

LINQ to SQL enables us to quickly create data model classes from existing database schema. To-do this we'll open the NerdDinner database in the Server Explorer, and select the Tables we want to model in it:

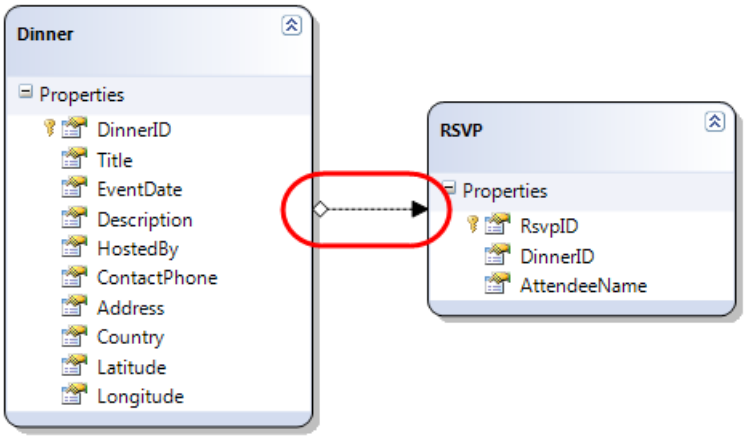


We can then drag the tables onto the LINQ to SQL designer surface. When we do this LINQ to SQL will automatically create Dinner and RSVP classes using the schema of the tables (with class properties that map to the database table columns):



By default the LINQ to SQL designer automatically "pluralizes" table and column names when it creates classes based on a database schema. For example: the "Dinners" table in our example above resulted in a "Dinner" class. This class naming helps make our models consistent with .NET naming conventions, and I usually find that having the designer fix this up convenient (especially when adding lots of tables). If you don't like the name of a class or property that the designer generates, though, you can always override it and change it to any name you want. You can do this either by editing the entity/property name in-line within the designer or by modifying it via the property grid.

By default the LINQ to SQL designer also inspects the primary key/foreign key relationships of the tables, and based on them automatically creates default "relationship associations" between the different model classes it creates. For example, when we dragged the Dinners and RSVP tables onto the LINQ to SQL designer a one-to-many relationship association between the two was inferred based on the fact that the RSVP table had a foreign-key to the Dinners table (this is indicated by the arrow in the designer):



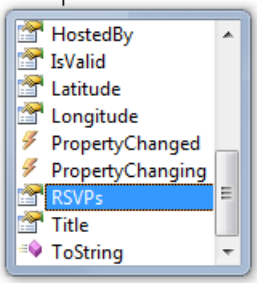
The above association will cause LINQ to SQL to add a strongly typed "Dinner" property to the RSVP class that developers can use to access the Dinner associated with a given RSVP. It will also cause the Dinner class to have a "RSVPs" collection property that enables developers to retrieve and update RSVP objects associated with a particular Dinner.

Below you can see an example of intellisense within Visual Studio when we create a new RSVP object and add it to a Dinner's RSVPs collection. Notice how LINQ to SQL automatically added a "RSVPs" collection on the Dinner object:

```
Dinner dinner = db.Dinners.Single(d => d.DinnerID == 1);

RSVP myRSVP = new RSVP();
myRSVP.AttendeeName = "ScottGu";

dinner.R
```



By adding the RSVP object to the Dinner's RSVPs collection we are telling LINQ to SQL to associate a foreign-key relationship between the Dinner and the RSVP row in our database:

```
Dinner dinner = db.Dinners.Single(d => d.DinnerID == 1);

RSVP myRSVP = new RSVP();
myRSVP.AttendeeName = "ScottGu";

dinner.RSVPs.Add(myRSVP);
```

If you don't like how the designer has modeled or named a table association, you can override it. Just click on the association arrow within the designer and access its properties via the property grid to rename, delete or modify it. For our NerdDinner application, though, the default association rules work well for the data model classes we are building and we can just use the default behavior.

NerdDinnerDataContext Class

Visual Studio will automatically create .NET classes that represent the models and database relationships defined using the LINQ to SQL designer. A LINQ to SQL DataContext class is also generated for each LINQ to SQL designer file added to the solution. Because we named our LINQ to SQL class item "NerdDinner", the DataContext class created will be called "NerdDinnerDataContext". This NerdDinnerDataContext class is the primary way we will interact with the database.

Our NerdDinnerDataContext class exposes two properties - "Dinners" and "RSVPs" - that represent the two tables we modeled within the database. We can use C# to write LINQ queries against those properties to query and retrieve Dinner and RSVP objects from the database.

The following code demonstrates how to instantiate a NerdDinnerDataContext object and perform a LINQ query against it to obtain a sequence of Dinners that occur in the future. Visual Studio provides full intellisense when writing the LINQ query, and the objects returned from it are strongly-typed and also support intellisense:

```
NerdDinnerDataContext db = new NerdDinnerDataContext();

var upcomingDinners = from dinner in db.Dinners
                      where dinner.EventDate > DateTime.Now
                      orderby dinner.EventDate
                      select dinner;

foreach (Dinner dinner in upcomingDinners) {
    dinner.
}
```

Description
DinnerID
Equals
EventDate
GetErrors
GetHashCode
GetType
HostedBy
Latitude

string Dinner.Description

In addition to allowing us to query for Dinner and RSVP objects, a NerdDinnerDataContext also automatically tracks any changes we subsequently make to the Dinner and RSVP objects we retrieve through it. We can use this functionality to easily save the changes back to the database - without having to write any explicit SQL update code.

For example, the code below demonstrates how to use a LINQ query to retrieve a single Dinner object from the database, update two of the Dinner properties, and then save the changes back to the database:

```
NerdDinnerDataContext db = new NerdDinnerDataContext();

// Retrieve Dinner object that reprents row with DinnerID of 1
Dinner dinner = db.Dinners.Single(d => d.DinnerID == 1);

// Update two properties on Dinner
dinner.Title = "Changed Title";
dinner.Description = "This dinner will be fun";

// Persist changes to database
db.SubmitChanges();
```

The NerdDinnerDataContext object in the code above automatically tracked the property changes made to the Dinner object we retrieved from it. When we called the "SubmitChanges()" method, it will execute an appropriate SQL "UPDATE" statement to the database to persist the updated values back.

Creating a DinnerRepository Class

For small applications it is sometimes fine to have Controllers work directly against a LINQ to SQL DataContext class, and embed LINQ queries within the Controllers. As applications get larger, though, this approach becomes cumbersome to maintain and test. It can also lead to us duplicating the same LINQ queries in multiple places.

One approach that can make applications easier to maintain and test is to use a "repository" pattern. A repository class helps encapsulate data querying and persistence logic, and abstracts away the implementation details of the data persistence from the application. In addition to making application code cleaner, using a repository pattern can make it easier to change data storage implementations in the future, and it can help facilitate unit testing an application without requiring a real database.

For our NerdDinner application we'll define a DinnerRepository class with the below signature:

```
public class DinnerRepository {

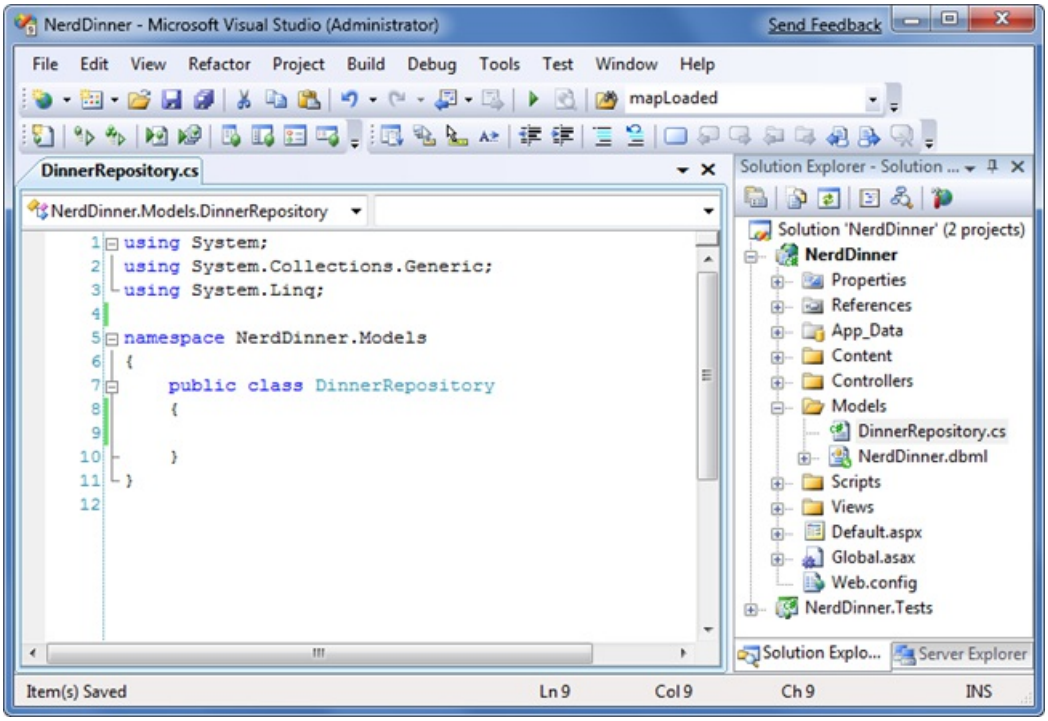
    // Query Methods
    public IQueryable<Dinner> FindAllDinners();
    public IQueryable<Dinner> FindUpcomingDinners();
    public Dinner             GetDinner(int id);

    // Insert/Delete
    public void Add(Dinner dinner);
    public void Delete(Dinner dinner);

    // Persistence
    public void Save();
}
```

Note: Later in this chapter we'll extract an IDinnerRepository interface from this class and enable dependency injection with it on our Controllers. To begin with, though, we are going to start simple and just work directly with the DinnerRepository class.

To implement this class we'll right-click on our "Models" folder and choose the **Add->New Item** menu command. Within the "Add New Item" dialog we'll select the "Class" template and name the file "DinnerRepository.cs":



We can then implement our DinnerRepository class using the code below:

```
public class DinnerRepository {

    private NerdDinnerDataContext db = new NerdDinnerDataContext();

    //
    // Query Methods

    public IQueryable<Dinner> FindAllDinners() {
        return db.Dinners;
    }

    public IQueryable<Dinner> FindUpcomingDinners() {
        return from dinner in db.Dinners
               where dinner.EventDate > DateTime.Now
               orderby dinner.EventDate
               select dinner;
    }

    public Dinner GetDinner(int id) {
        return db.Dinners.SingleOrDefault(d => d.DinnerID == id);
    }

    //
    // Insert/Delete Methods

    public void Add(Dinner dinner) {
        db.Dinners.InsertOnSubmit(dinner);
    }

    public void Delete(Dinner dinner) {
        db.RSVPs.DeleteAllOnSubmit(dinner.RSVPs);
        db.Dinners.DeleteOnSubmit(dinner);
    }

    //
    // Persistence

    public void Save() {
        db.SubmitChanges();
    }
}
```

Retrieving, Updating, Inserting and Deleting using the DinnerRepository class

Now that we've created our DinnerRepository class, let's look at a few code examples that demonstrate common tasks we can do with it:

Querying Examples

The code below retrieves a single Dinner using the DinnerID value:

```
DinnerRepository dinnerRepository = new DinnerRepository();

// Retrieve specific dinner by its DinnerID
Dinner dinner = dinnerRepository.GetDinner(5);
```

The code below retrieves all upcoming dinners and loops over them:

```
DinnerRepository dinnerRepository = new DinnerRepository();

// Retrieve all upcoming Dinners
var upcomingDinners = dinnerRepository.FindUpcomingDinners();

// Loop over each upcoming Dinner and print out its Title
foreach (Dinner dinner in upcomingDinners) {
   Response.Write("Title" + dinner.Title);
}
```

Insert and Update Examples

The code below demonstrates adding two new dinners. Additions/modifications to the repository aren't committed to the database until the "Save()" method is called on it. LINQ to SQL automatically wraps all changes in a database transaction – so either all changes happen or none of them do when our repository saves:

```
DinnerRepository dinnerRepository = new DinnerRepository();

// Create First Dinner
Dinner newDinner1 = new Dinner();
newDinner1.Title = "Dinner with Scott";
newDinner1.HostedBy = "ScotGu";
newDinner1.ContactPhone = "425-703-8072";

// Create Second Dinner
Dinner newDinner2 = new Dinner();
newDinner2.Title = "Dinner with Bill";
newDinner2.HostedBy = "BillG";
newDinner2.ContactPhone = "425-555-5151";

// Add Dinners to Repository
dinnerRepository.Add(newDinner1);
dinnerRepository.Add(newDinner2);

// Persist Changes
dinnerRepository.Save();
```

The code below retrieves an existing Dinner object, and modifies two properties on it. The changes are committed back to the database when the "Save()" method is called on our repository:

```
DinnerRepository dinnerRepository = new DinnerRepository();

// Retrieve specific dinner by its DinnerID
Dinner dinner = dinnerRepository.GetDinner(5);

// Update Dinner properties
dinner.Title = "Update Title";
dinner.HostedBy = "New Owner";

// Persist changes
dinnerRepository.Save();
```

The code below retrieves a dinner and then adds an RSVP to it. It does this using the RSVPs collection on the Dinner object that LINQ to SQL created for us (because there is a primary-key/foreign-key relationship between the two in the database). This change is persisted back to the database as a new RSVP table row when the "Save()" method is called on the repository:

```
DinnerRepository dinnerRepository = new DinnerRepository();

// Retrieve specific dinner by its DinnerID
Dinner dinner = dinnerRepository.GetDinner(5);

// Create a new RSVP object
RSVP myRSVP = new RSVP();
myRSVP.AttendeeName = "ScottGu";

// Add RSVP to Dinner's RSVP Collection
dinner.RSVPs.Add(myRSVP);

// Persist changes
dinnerRepository.Save();
```

Delete Example

The code below retrieves an existing Dinner object, and then marks it to be deleted. When the "Save()" method is called on the repository it will commit the delete back to the database:

```
DinnerRepository dinnerRepository = new DinnerRepository();

// Retrieve specific dinner by its DinnerID
Dinner dinner = dinnerRepository.GetDinner(5);

// Mark dinner to be deleted
dinnerRepository.Delete(dinner);

// Persist changes
dinnerRepository.Save();
```

Integrating Validation and Business Rule Logic with Model Classes

Integrating validation and business rule logic is a key part of any application that works with data.

Schema Validation

When model classes are defined using the LINQ to SQL designer, the datatypes of the properties in the data model classes correspond to the datatypes of the database table. For example: if the "EventDate" column in the Dinners table is a "datetime", the data model class created by LINQ to SQL will be of type "DateTime" (which is a built-in .NET datatype). This means you will get compile errors if you attempt to assign an integer or boolean to it from code, and it will raise an error automatically if you attempt to implicitly convert a non-valid string type to it at runtime.

LINQ to SQL will also automatically handles escaping SQL values for you when using strings - which helps protect you against SQL injection attacks when using it.

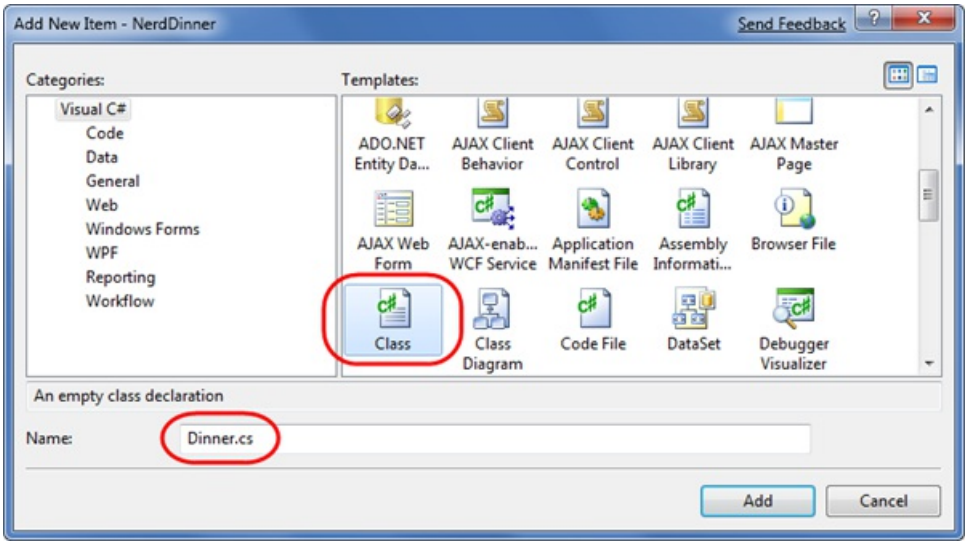
Validation and Business Rule Logic

Schema validation is useful as a first step, but is rarely sufficient. Most real-world scenarios require the ability to specify richer validation logic that can span multiple properties, execute code, and often have awareness of a model's state (for example: is it being created /updated/deleted, or within a domain-specific state like "archived"). There are a variety of different patterns and frameworks that can be used to define and apply validation rules to model classes, and there are several .NET based frameworks out there that can be used to help with this. You can use pretty much any of them within ASP.NET MVC applications.

For the purposes of our NerdDinner application, we'll use a relatively simple and straight-forward pattern where we expose an IsValid property and a GetRuleViolations() method on our Dinner model object. The IsValid property will return true or false depending on whether the validation and business rules are all valid. The GetRuleViolations() method will return a list of any rule errors.

We'll implement IsValid and GetRuleViolations() for our Dinner model by adding a "partial class" to our project.

Partial classes can be used to add methods/properties/events to classes maintained by a VS designer (like the Dinner class generated by the LINQ to SQL designer) and help avoid the tool from messing with our code. We can add a new partial class to our project by right-clicking on the \Models folder, and then select the "Add New Item" menu command. We can then choose the "Class" template within the "Add New Item" dialog and name it Dinner.cs.



Clicking the "Add" button will add a Dinner.cs file to our project and open it within the IDE. We can then implement a basic rule/validation enforcement framework using the below code:

```
public partial class Dinner {

    public bool IsValid {
        get { return (GetRuleViolations().Count() == 0); }
    }

    public IEnumerable<RuleViolation> GetRuleViolations() {
        yield break;
    }

    partial void OnValidate(ChangeAction action) {
        if (!IsValid)
            throw new ApplicationException("Rule violations prevent saving");
    }
}

public class RuleViolation {

    public string ErrorMessage { get; private set; }
    public string PropertyName { get; private set; }

    public RuleViolation(string errorMessage, string propertyName) {
        ErrorMessage = errorMessage;
        PropertyName = propertyName;
    }
}
```

A few notes about the above code:

- The Dinner class is prefaced with a "partial" keyword – which means the code contained within it will be combined with the class generated/maintained by the LINQ to SQL designer and compiled into a single class.
- The RuleViolation class is a helper class we'll add to the project that allows us to provide more details about a rule violation.
- The Dinner.GetRuleViolations() method causes our validation and business rules to be evaluated (we'll implement them shortly). It then returns back a sequence of RuleViolation objects that provide more details

about any rule errors.

- The Dinner.IsValid property provides a convenient helper property that indicates whether the Dinner object has any active RuleViolations. It can be proactively checked by a developer using the Dinner object at anytime (and does not raise an exception).
- The Dinner.OnValidate() partial method is a hook that LINQ to SQL provides that allows us to be notified anytime the Dinner object is about to be persisted within the database. Our OnValidate() implementation above ensures that the Dinner has no RuleViolations before it is saved. If it is in an invalid state it raises an exception, which will cause LINQ to SQL to abort the transaction.

This approach provides a simple framework that we can integrate validation and business rules into. For now let's add the below rules to our GetRuleViolations() method:

```
public IEnumerable<RuleViolation> GetRuleViolations() {

    if (String.IsNullOrEmpty(Title))
        yield return new RuleViolation("Title required","Title");

    if (String.IsNullOrEmpty(Description))
        yield return new RuleViolation("Description required","Description");

    if (String.IsNullOrEmpty(HostedBy))
        yield return new RuleViolation("HostedBy required", "HostedBy");

    if (String.IsNullOrEmpty(Address))
        yield return new RuleViolation("Address required", "Address");

    if (String.IsNullOrEmpty(Country))
        yield return new RuleViolation("Country required", "Country");

    if (String.IsNullOrEmpty(ContactPhone))
        yield return new RuleViolation("Phone# required", "ContactPhone");

    if (!PhoneValidator.IsValidNumber(ContactPhone, Country))
        yield return new RuleViolation("Phone# does not match country", "ContactPhone");

    yield break;
}
```

We are using the "yield return" feature of C# to return a sequence of any RuleViolations. The first six rule checks above simply enforce that string properties on our Dinner cannot be null or empty. The last rule is a little more interesting, and calls a PhoneValidator.IsValidNumber() helper method that we can add to our project to verify that the ContactPhone number format matches the Dinner's country.

We can use .NET's regular expression support to implement this phone validation support. Below is a simple PhoneValidator implementation that we can add to our project that enables us to add country-specific Regex pattern checks:

```
public class PhoneValidator {

    static IDictionary<string, Regex> countryRegex = new Dictionary<string, Regex>() {
           { "USA", new Regex("^[2-9]\\d{2}-\\d{3}-\\d{4}$")},
           { "UK", new Regex("(^1300\\d{6}$)|(^1800|1900|1902\\d{6}$)|(^0[2|3|7|8]{1}[0-9]{8}$)|(^13\\d{4}$)|
(^04\\d{2,3}\\d{6}$)")},
           { "Netherlands", new Regex("(^\\+[0-9]{2}|^\\+[0-9]{2}\\(0\\)|^\\(\\+[0-9]{2}\\)\\(0\\)|^00[0-9]
{2}|^0)([0-9]{9}$|[0-9\\-\\s]{10}$)")},
    };

    public static bool IsValidNumber(string phoneNumber, string country) {

        if (country != null && countryRegex.ContainsKey(country))
            return countryRegex[country].IsMatch(phoneNumber);
        else
            return false;
    }

    public static IEnumerable<string> Countries {
        get {
            return countryRegex.Keys;
        }
    }
}
```

Handling Validation and Business Logic Violations

Now that we've added the above validation and business rule code, any time we try to create or update a Dinner, our validation logic rules will be evaluated and enforced.

Developers can write code like below to proactively determine if a Dinner object is valid, and retrieve a list of all violations in it without raising any exceptions:

```
Dinner dinner = dinnerRepository.GetDinner(5);

dinner.Country = "USA";
dinner.ContactPhone = "425-555-BOGUS";

if (!dinner.IsValid) {

    var errors = dinner.GetRuleViolations();

    // do something to fix the errors
}
```

If we attempt to save a Dinner in an invalid state, an exception will be raised when we call the Save() method on the DinnerRepository. This occurs because LINQ to SQL automatically calls our Dinner.OnValidate() partial method before it saves the Dinner's changes, and we added code to Dinner.OnValidate() to raise an exception if any rule violations exist in the Dinner. We can catch this exception and reactively retrieve a list of the violations to fix:

```
Dinner dinner = dinnerRepository.GetDinner(5);

try {

    dinner.Country = "USA";
    dinner.ContactPhone = "425-555-BOGUS";

    dinnerRepository.Save();
}
catch {

    var errors = dinner.GetRuleViolations();

    // do something to fix errors
}
```

Because our validation and business rules are implemented within our model layer, and not within the UI layer, they will be applied and used across all scenarios within our application. We can later change or add business rules and have all code that works with our Dinner objects honor them.

Having the flexibility to change business rules in one place, without having these changes ripple throughout the application and UI logic, is a sign of a well-written application, and a benefit that an MVC framework helps encourage.

Next Step

We've now got a model that we can use to both query and update our database.

Let's now add some controllers and views to the project that we can use to build an HTML UI experience around it.

PREVIOUS NEXT

Use Controllers and Views to Implement a Listing/Details UI

4/10/2020 • 19 minutes to read • Edit Online

by Microsoft

Download PDF

> This is step 4 of a free "NerdDinner" application tutorial that walks-through how to build a small, but complete, web application using ASP.NET MVC 1.
>
> Step 4 shows how to add a Controller to the application that takes advantage of our model to provide users with a data listing/details navigation experience for dinners on our NerdDinner site.
>
> If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

NerdDinner Step 4: Controllers and Views

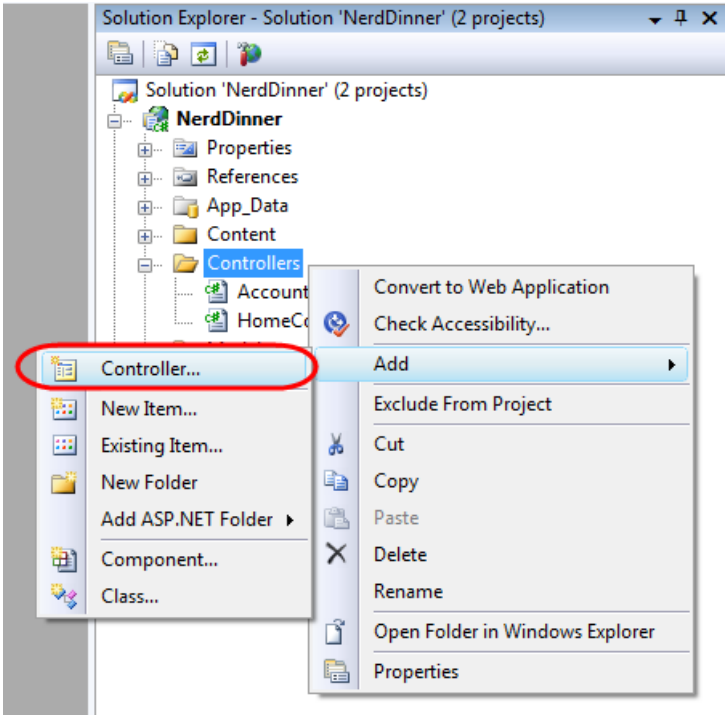
With traditional web frameworks (classic ASP, PHP, ASP.NET Web Forms, etc), incoming URLs are typically mapped to files on disk. For example: a request for a URL like "/Products.aspx" or "/Products.php" might be processed by a "Products.aspx" or "Products.php" file.

Web-based MVC frameworks map URLs to server code in a slightly different way. Instead of mapping incoming URLs to files, they instead map URLs to methods on classes. These classes are called "Controllers" and they are responsible for processing incoming HTTP requests, handling user input, retrieving and saving data, and determining the response to send back to the client (display HTML, download a file, redirect to a different URL, etc).

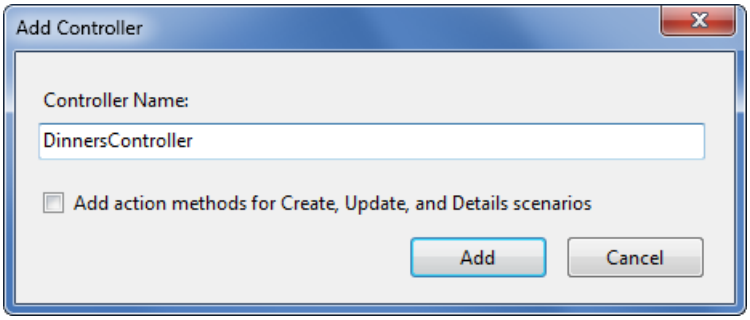
Now that we have built up a basic model for our NerdDinner application, our next step will be to add a Controller to the application that takes advantage of it to provide users with a data listing/details navigation experience for Dinners on our site.

Adding a DinnersController Controller

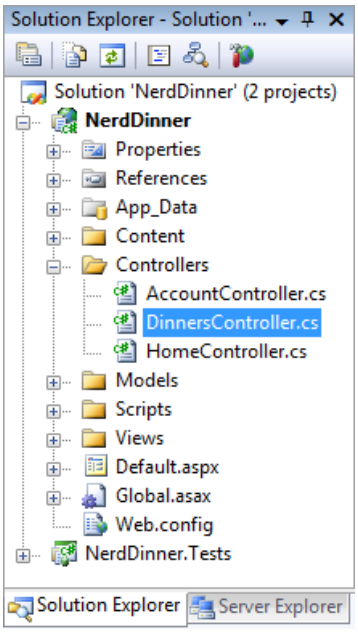
We'll begin by right-clicking on the "Controllers" folder within our web project, and then select the **Add->Controller** menu command (you can also execute this command by typing Ctrl-M, Ctrl-C):



This will bring up the "Add Controller" dialog:



We'll name the new controller "DinnersController" and click the "Add" button. Visual Studio will then add a DinnersController.cs file under our \Controllers directory:



It will also open up the new DinnersController class within the code-editor.

Adding Index() and Details() Action Methods to the DinnersController Class

We want to enable visitors using our application to browse a list of upcoming dinners, and allow them to click on any Dinner in the list to see specific details about it. We'll do this by publishing the following URLs from our

application:

URL	PURPOSE
/Dinners/	Display an HTML list of upcoming dinners
/Dinners/Details/[id]	Display details about a specific dinner indicated by an "id" parameter embedded within the URL – which will match the DinnerID of the dinner in the database. For example: /Dinners/Details/2 would display an HTML page with details about the Dinner whose DinnerID value is 2.

We will publish initial implementations of these URLs by adding two public "action methods" to our DinnersController class like below:

```
public class DinnersController : Controller {

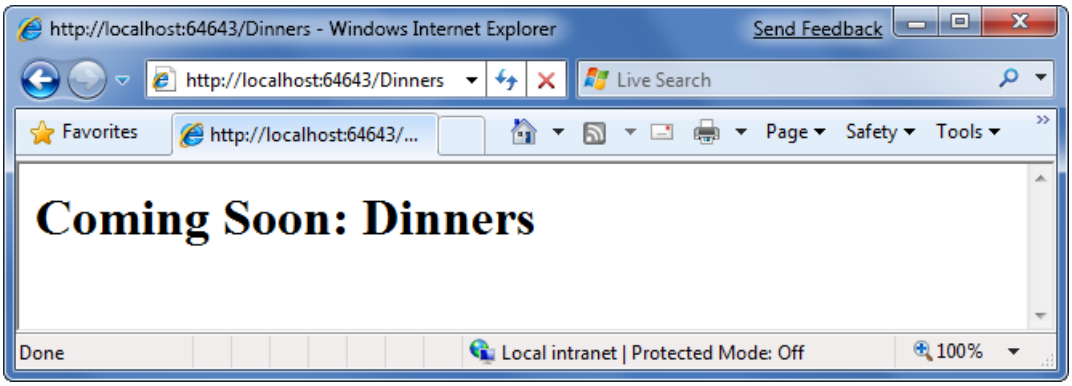
    //
    // HTTP-GET: /Dinners/

    public void Index() {
        Response.Write("<h1>Coming Soon: Dinners</h1>");
    }

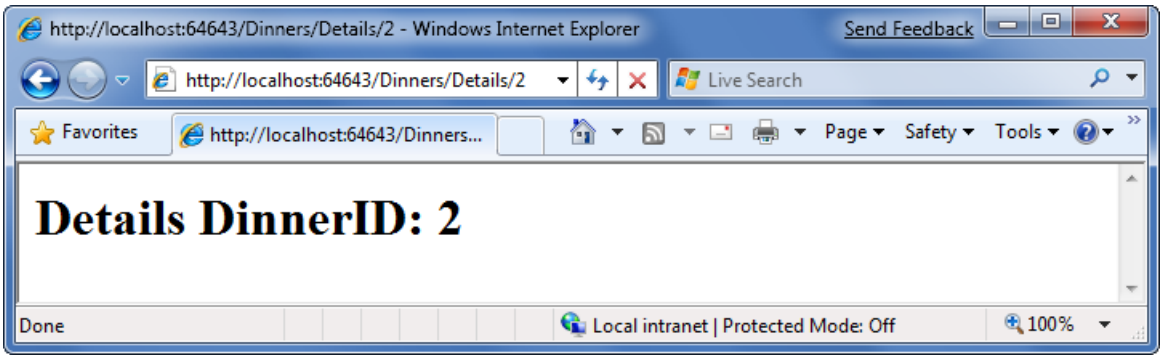
    //
    // HTTP-GET: /Dinners/Details/2

    public void Details(int id) {
        Response.Write("<h1>Details DinnerID: " + id + "</h1>");
    }
}
```

We'll then run the NerdDinner application and use our browser to invoke them. Typing in the *"/Dinners/"* URL will cause our *Index()* method to run, and it will send back the following response:



Typing in the *"/Dinners/Details/2"* URL will cause our *Details()* method to run, and send back the following response:

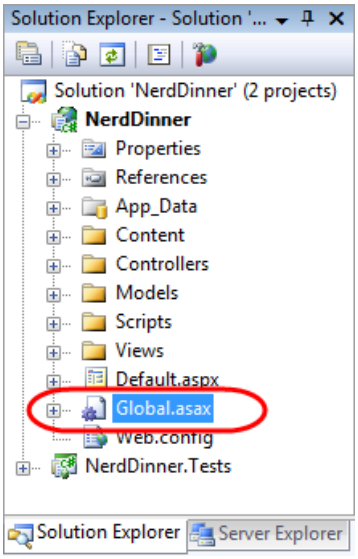


You might be wondering - how did ASP.NET MVC know to create our DinnersController class and invoke those methods? To understand that let's take a quick look at how routing works.

Understanding ASP.NET MVC Routing

ASP.NET MVC includes a powerful URL routing engine that provides a lot of flexibility in controlling how URLs are mapped to controller classes. It allows us to completely customize how ASP.NET MVC chooses which controller class to create, which method to invoke on it, as well as configure different ways that variables can be automatically parsed from the URL/Querystring and passed to the method as parameter arguments. It delivers the flexibility to totally optimize a site for SEO (search engine optimization) as well as publish any URL structure we want from an application.

By default, new ASP.NET MVC projects come with a preconfigured set of URL routing rules already registered. This enables us to easily get started on an application without having to explicitly configure anything. The default routing rule registrations can be found within the "Application" class of our projects – which we can open by double-clicking the "Global.asax" file in the root of our project:



The default ASP.NET MVC routing rules are registered within the "RegisterRoutes" method of this class:

```
public void RegisterRoutes(RouteCollection routes) {

    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        "Default",                                          // Route name
        "{controller}/{action}/{id}",                       // URL w/ params
        new { controller="Home", action="Index",id="" }  // Param defaults
    );
}
```

The "routes.MapRoute()" method call above registers a default routing rule that maps incoming URLs to controller classes using the URL format: "/{controller}/{action}/{id}" – where "controller" is the name of the controller class to instantiate, "action" is the name of a public method to invoke on it, and "id" is an optional parameter embedded within the URL that can be passed as an argument to the method. The third parameter passed to the "MapRoute()" method call is a set of default values to use for the controller/action/id values in the event that they are not present in the URL (Controller = "Home", Action="Index", Id="").

Below is a table that demonstrates how a variety of URLs are mapped using the default "*/{controllers}/{action}/{id}*" route rule:

URL	CONTROLLER CLASS	ACTION METHOD	PARAMETERS PASSED
/Dinners/Details/2	DinnersController	Details(id)	id=2
/Dinners/Edit/5	DinnersController	Edit(id)	id=5
/Dinners/Create	DinnersController	Create()	N/A
/Dinners	DinnersController	Index()	N/A
/Home	HomeController	Index()	N/A
/	HomeController	Index()	N/A

The last three rows show the default values (Controller = Home, Action = Index, Id = "") being used. Because the "Index" method is registered as the default action name if one isn't specified, the "/Dinners" and "/Home" URLs cause the Index() action method to be invoked on their Controller classes. Because the "Home" controller is registered as the default controller if one isn't specified, the "/" URL causes the HomeController to be created, and the Index() action method on it to be invoked.

If you don't like these default URL routing rules, the good news is that they are easy to change - just edit them within the RegisterRoutes method above. For our NerdDinner application, though, we aren't going to change any of the default URL routing rules – instead we'll just use them as-is.

Using the DinnerRepository from our DinnersController

Let's now replace our current implementation of the DinnersController's Index() and Details() action methods with implementations that use our model.

We'll use the DinnerRepository class we built earlier to implement the behavior. We'll begin by adding a "using" statement that references the "NerdDinner.Models" namespace, and then declare an instance of our DinnerRepository as a field on our DinnerController class.

Later in this chapter we'll introduce the concept of "Dependency Injection" and show another way for our Controllers to obtain a reference to a DinnerRepository that enables better unit testing – but for right now we'll just create an instance of our DinnerRepository inline like below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using NerdDinner.Models;

namespace NerdDinner.Controllers {

    public class DinnersController : Controller {

        DinnerRepository dinnerRepository = new DinnerRepository();

        //
        // GET: /Dinners/

        public void Index() {
            var dinners = dinnerRepository.FindUpcomingDinners().ToList();
        }

        //
        // GET: /Dinners/Details/2

        public void Details(int id) {
            Dinner dinner = dinnerRepository.GetDinner(id);
        }
    }
}
```

Now we are ready to generate a HTML response back using our retrieved data model objects.

Using Views with our Controller

While it is possible to write code within our action methods to assemble HTML and then use the *Response.Write()* helper method to send it back to the client, that approach becomes fairly unwieldy quickly. A much better approach is for us to only perform application and data logic inside our DinnersController action methods, and to then pass the data needed to render a HTML response to a separate "view" template that is responsible for outputting the HTML representation of it. As we'll see in a moment, a "view" template is a text file that typically contains a combination of HTML markup and embedded rendering code.

Separating our controller logic from our view rendering brings several big benefits. In particular it helps enforce a clear "separation of concerns" between the application code and UI formatting/rendering code. This makes it much easier to unit-test application logic in isolation from UI rendering logic. It makes it easier to later modify the UI rendering templates without having to make application code changes. And it can make it easier for developers and designers to collaborate together on projects.

We can update our DinnersController class to indicate that we want to use a view template to send back an HTML UI response by changing the method signatures of our two action methods from having a return type of "void" to instead have a return type of "ActionResult". We can then call the *View()* helper method on the Controller base class to return back a "ViewResult" object like below:

```
public class DinnersController : Controller {

    DinnerRepository dinnerRepository = new DinnerRepository();

    //
    // GET: /Dinners/

    public ActionResult Index() {

        var dinners = dinnerRepository.FindUpcomingDinners().ToList();

        return View("Index", dinners);
    }

    //
    // GET: /Dinners/Details/2

    public ActionResult Details(int id) {

        Dinner dinner = dinnerRepository.GetDinner(id);

        if (dinner == null)
            return View("NotFound");
        else
            return View("Details", dinner);
    }
}
```

The signature of the *View()* helper method we are using above looks like below:

```
ViewResult View(string viewName, object model);
```

The first parameter to the *View()* helper method is the name of the view template file we want to use to render the HTML response. The second parameter is a model object that contains the data that the view template needs in order to render the HTML response.

Within our Index() action method we are calling the *View()* helper method and indicating that we want to render an HTML listing of dinners using an "Index" view template. We are passing the view template a sequence of Dinner objects to generate the list from:

```
//
    // GET: /Dinners/

    public ActionResult Index() {

        var dinners = dinnerRepository.FindUpcomingDinners().ToList();

        return View("Index", dinners);
    }
```

Within our Details() action method we attempt to retrieve a Dinner object using the id provided within the URL. If a valid Dinner is found we call the *View()* helper method, indicating we want to use a "Details" view template to render the retrieved Dinner object. If an invalid dinner is requested, we render a helpful error message that indicates that the Dinner doesn't exist using a "NotFound" view template (and an overloaded version of the *View()* helper method that just takes the template name):

```
//
    // GET: /Dinners/Details/2

    public ActionResult Details(int id) {

        Dinner dinner = dinnerRepository.FindDinner(id);

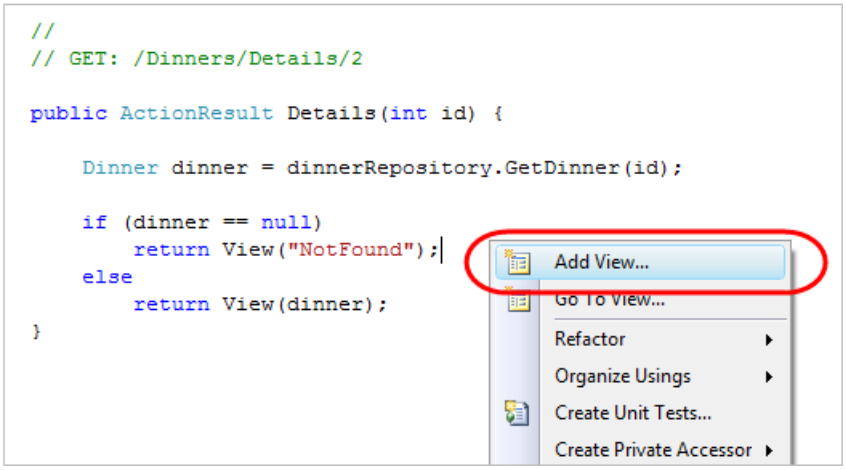
        if (dinner == null)
            return View("NotFound");
        else
            return View("Details", dinner);
    }
```

Let's now implement the "NotFound", "Details", and "Index" view templates.

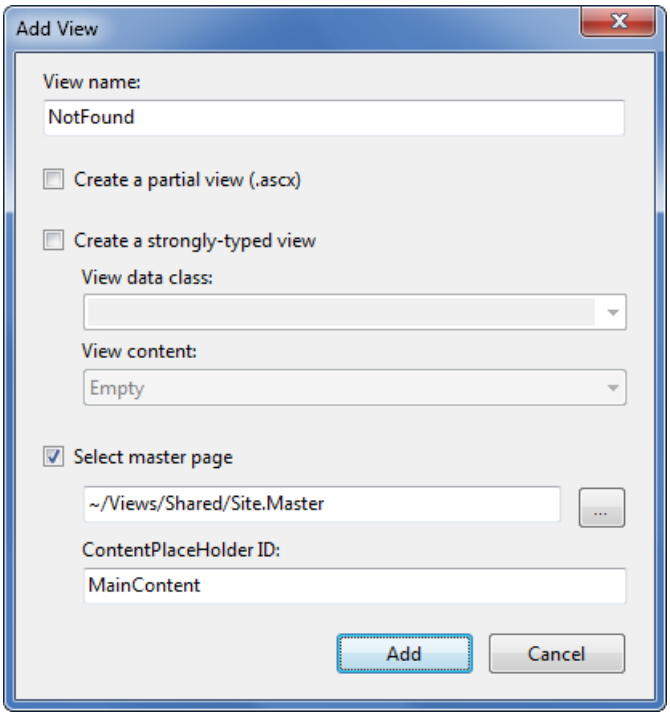
Implementing the "NotFound" View Template

We'll begin by implementing the "NotFound" view template – which displays a friendly error message indicating that the requested dinner can't be found.

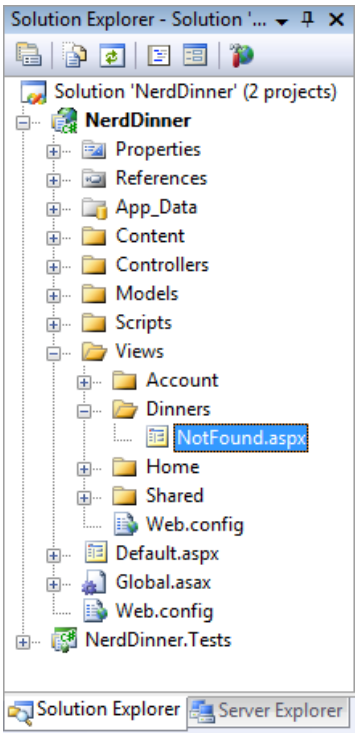
We'll create a new view template by positioning our text cursor within a controller action method, and then right click and choose the "Add View" menu command (we can also execute this command by typing Ctrl-M, Ctrl-V):



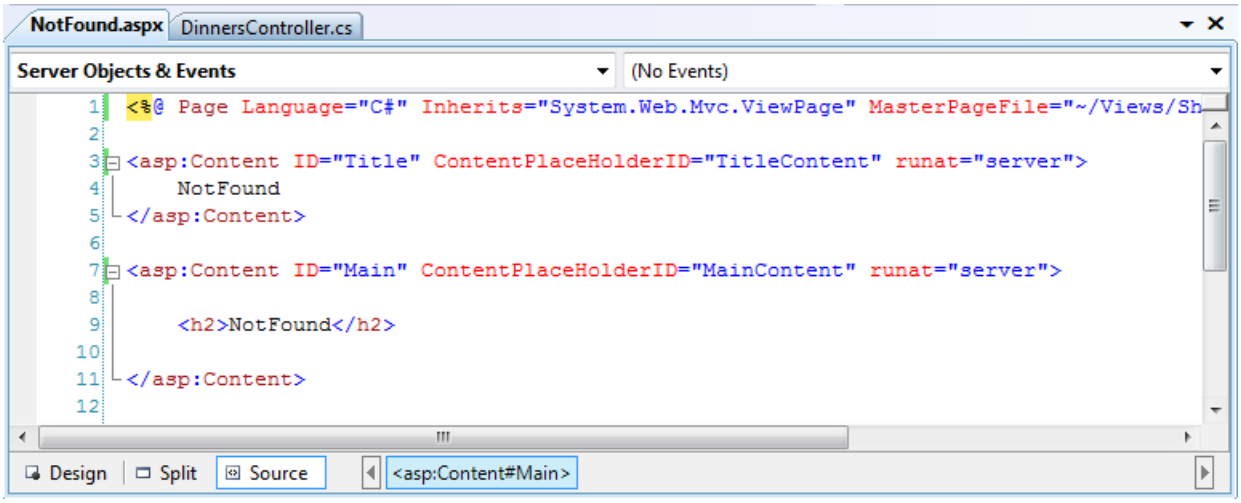
This will bring up an "Add View" dialog like below. By default the dialog will pre-populate the name of the view to create to match the name of the action method the cursor was in when the dialog was launched (in this case "Details"). Because we want to first implement the "NotFound" template, we'll override this view name and set it to instead be "NotFound":



When we click the "Add" button, Visual Studio will create a new "NotFound.aspx" view template for us within the "\Views\Dinners" directory (which it will also create if the directory doesn't already exist):



It will also open up our new "NotFound.aspx" view template within the code-editor:



View templates by default have two "content regions" where we can add content and code. The first allows us to customize the "title" of the HTML page sent back. The second allows us to customize the "main content" of the HTML page sent back.

To implement our "NotFound" view template we'll add some basic content:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    Dinner Not Found
</asp:Content>

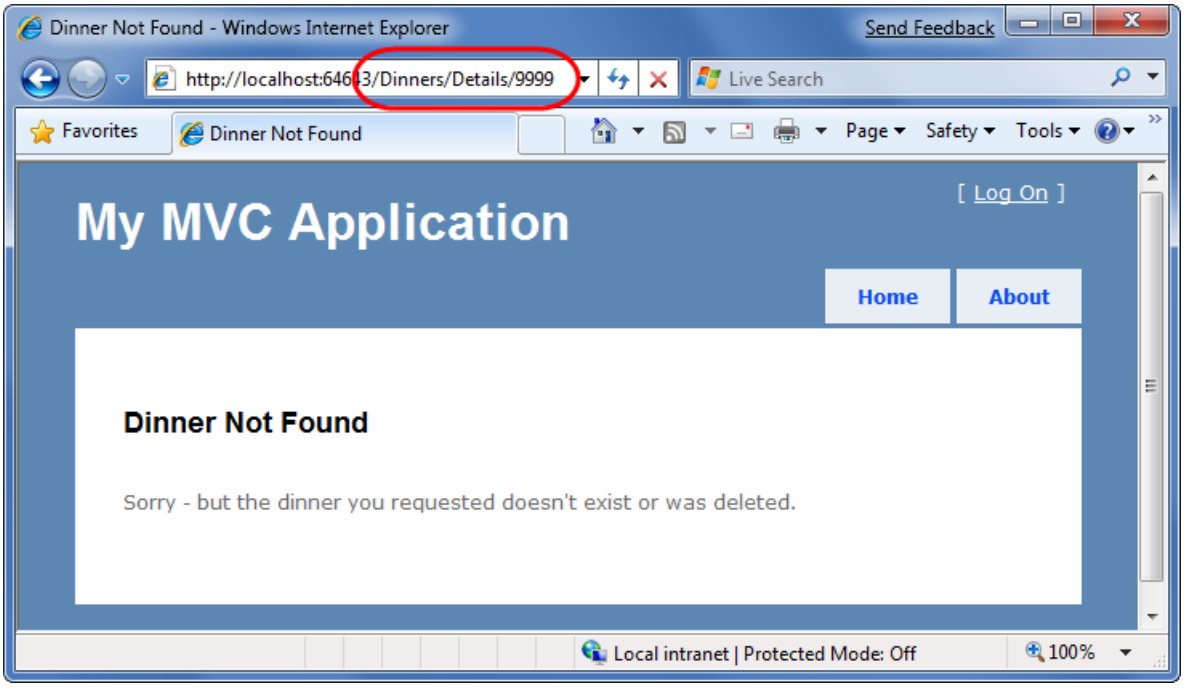
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Dinner Not Found</h2>

    <p>Sorry - but the dinner you requested doesn't exist or was deleted.</p>

</asp:Content>
```

We can then try it out within the browser. To do this let's request the *"/Dinners/Details/9999"* URL. This will refer to a dinner that doesn't currently exist in the database, and will cause our DinnersController.Details() action method to render our "NotFound" view template:



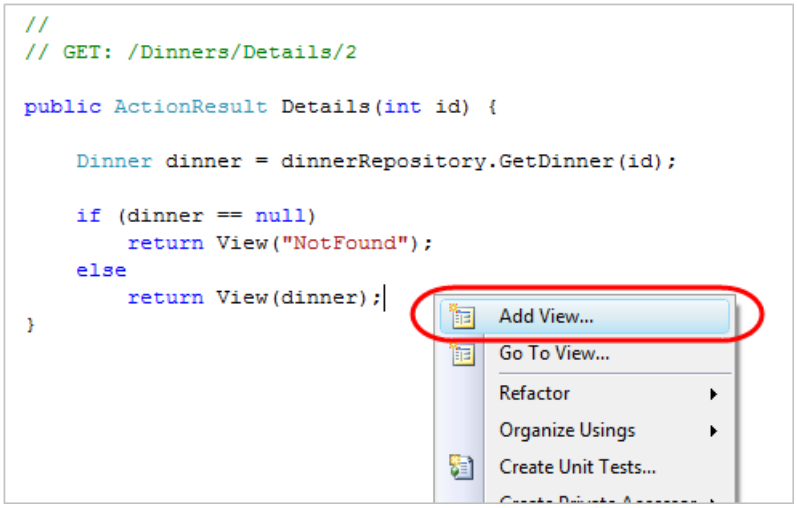
One thing you'll notice in the screen-shot above is that our basic view template has inherited a bunch of HTML that surrounds the main content on the screen. This is because our view-template is using a "master page"

template that enables us to apply a consistent layout across all views on the site. We'll discuss how master pages work more in a later part of this tutorial.

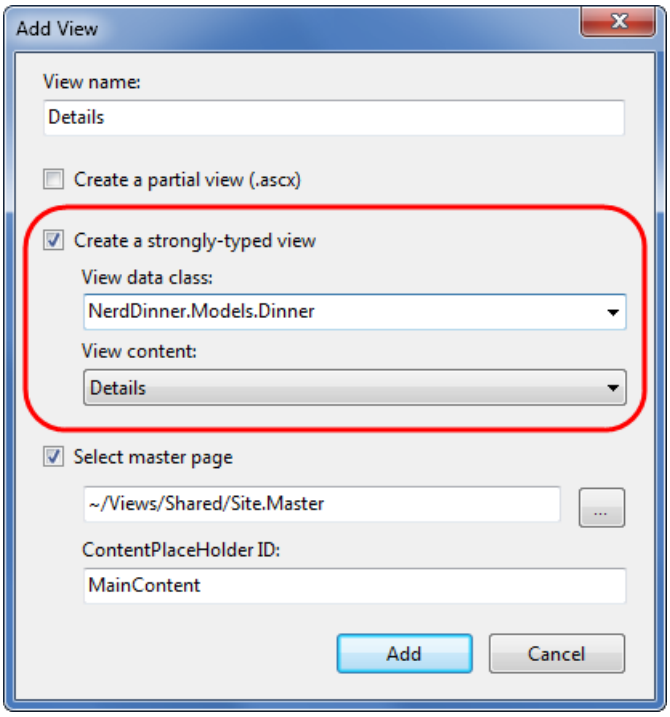
Implementing the "Details" View Template

Let's now implement the "Details" view template – which will generate HTML for a single Dinner model.

We'll do this by positioning our text cursor within the Details action method, and then right click and choose the "Add View" menu command (or press Ctrl-M, Ctrl-V):



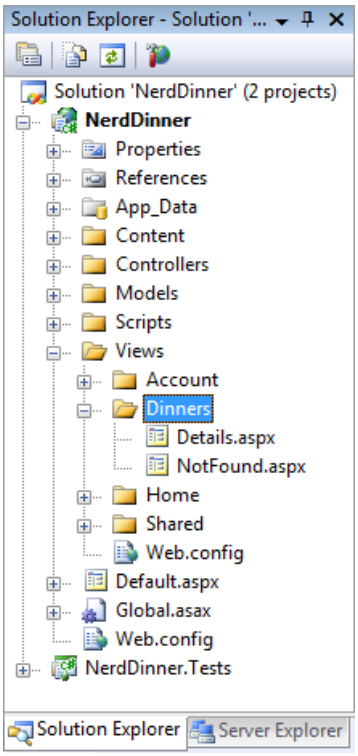
This will bring up the "Add View" dialog. We'll keep the default view name ("Details"). We'll also select the "Create a strongly-typed View" checkbox in the dialog and select (using the combobox dropdown) the name of the model type we are passing from the Controller to the View. For this view we are passing a Dinner object (the fully qualified name for this type is: "NerdDinner.Models.Dinner"):



Unlike the previous template, where we chose to create an "Empty View", this time we will choose to automatically "scaffold" the view using a "Details" template. We can indicate this by changing the "View content" drop-down in the dialog above.

"Scaffolding" will generate an initial implementation of our details view template based on the Dinner object we are passing to it. This provides an easy way for us to quickly get started on our view template implementation.

When we click the "Add" button, Visual Studio will create a new "Details.aspx" view template file for us within our "\Views\Dinners" directory:



It will also open up our new "Details.aspx" view template within the code-editor. It will contain an initial scaffold implementation of a details view based on a Dinner model. The scaffolding engine uses .NET reflection to look at the public properties exposed on the class passed it, and will add appropriate content based on each type it finds:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    Details
</asp:Content>

<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

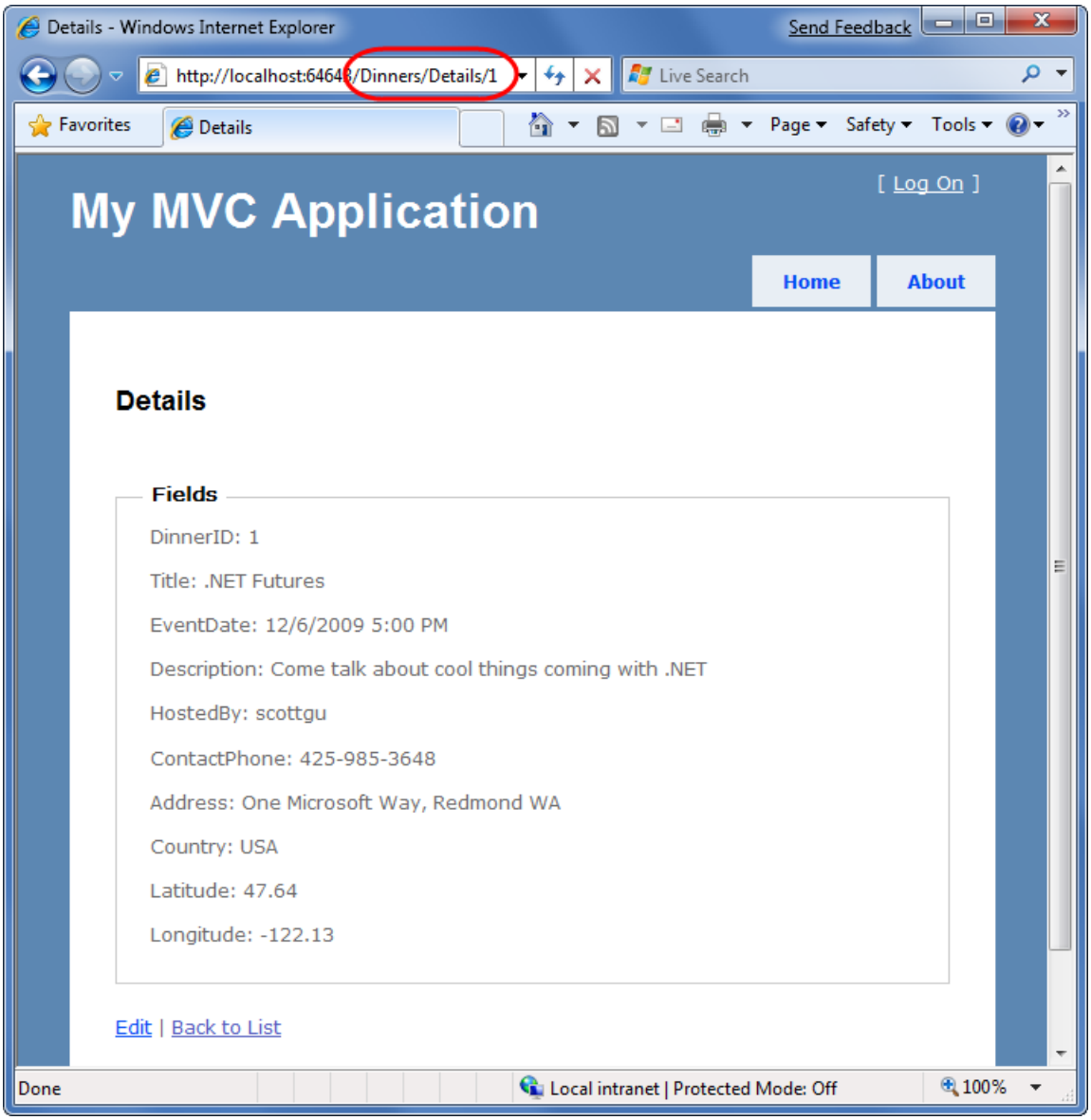
    <h2>Details</h2>

    <fieldset>
        <legend>Fields</legend>
        <p>
            DinnerID:
            <%=Html.Encode(Model.DinnerID) %>
        </p>
        <p>
            Title:
            <%=Html.Encode(Model.Title) %>
        </p>
        <p>
            EventDate:
            <%= Html.Encode(String.Format("{0:g}", Model.EventDate)) %>
        </p>
        <p>
            Description:
            <%=Html.Encode(Model.Description) %>
        </p>
        <p>
            HostedBy:
            <%=Html.Encode(Model.HostedBy) %>
        </p>
        <p>
            ContactPhone:
            <%=Html.Encode(Model.ContactPhone) %>
        </p>
        <p>
            Address:
            <%=Html.Encode(Model.Address) %>
        </p>
        <p>
            Country:
            <%=Html.Encode(Model.Country) %>
        </p>
        <p>
            Latitude:
            <%= Html.Encode(String.Format("{0:F}",Model.Latitude)) %>
        </p>
        <p>
            Longitude:
            <%= Html.Encode(String.Format("{0:F}",Model.Longitude)) %>
        </p>
    </fieldset>

    <p>
        <%=Html.ActionLink("Edit","Edit", new { id=Model.DinnerID }) %>|
        <%=Html.ActionLink("Back to List", "Index") %>
    </p>

</asp:Content>
```

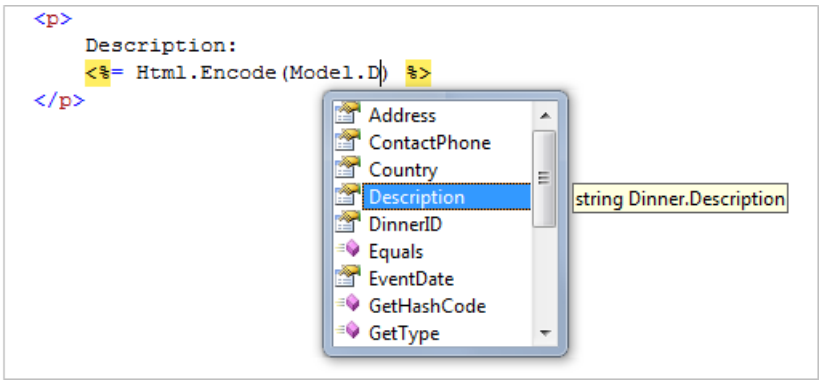
We can request the *"/Dinners/Details/1"* URL to see what this "details" scaffold implementation looks like in the browser. Using this URL will display one of the dinners we manually added to our database when we first created it:



This gets us up and running quickly, and provides us with an initial implementation of our Details.aspx view. We can then go and tweak it to customize the UI to our satisfaction.

When we look at the Details.aspx template more closely, we'll find that it contains static HTML as well as embedded rendering code. <% %> code nuggets execute code when the view template renders, and <%= %> code nuggets execute the code contained within them and then render the result to the output stream of the template.

We can write code within our View that accesses the "Dinner" model object that was passed from our controller using a strongly-typed "Model" property. Visual Studio provides us with full code-intellisense when accessing this "Model" property within the editor:



Let's make some tweaks so that the source for our final Details view template looks like below:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    Dinner: <%=Html.Encode(Model.Title) %>
</asp:Content>

<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

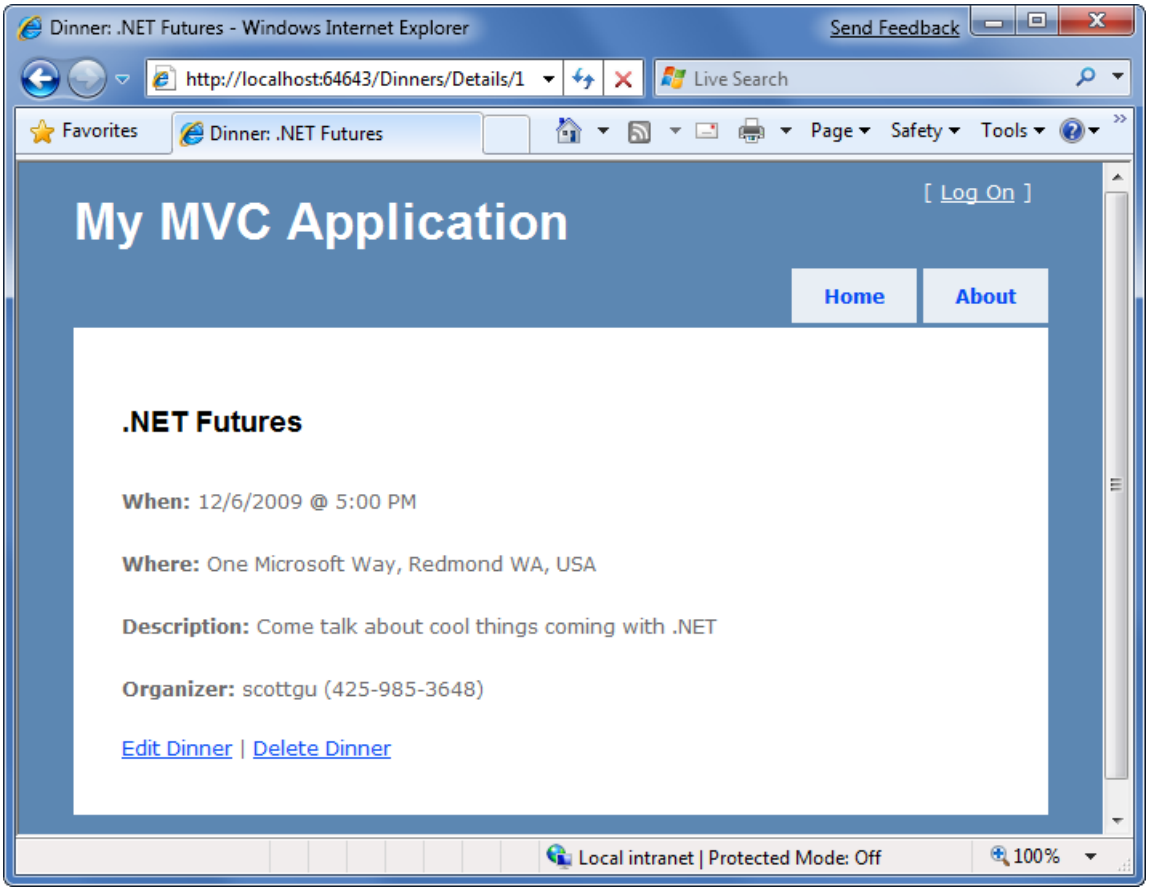
    <h2><%=Html.Encode(Model.Title) %></h2>
    <p>
        <strong>When:</strong>
        <%=Model.EventDate.ToShortDateString() %>

        <strong>@</strong>
        <%=Model.EventDate.ToShortTimeString() %>
    </p>
    <p>
        <strong>Where:</strong>
        <%=Html.Encode(Model.Address) %>,
        <%=Html.Encode(Model.Country) %>
    </p>
     <p>
        <strong>Description:</strong>
        <%=Html.Encode(Model.Description) %>
    </p>
    <p>
        <strong>Organizer:</strong>
        <%=Html.Encode(Model.HostedBy) %>
        (<%=Html.Encode(Model.ContactPhone) %>)
    </p>

    <%= Html.ActionLink("Edit Dinner", "Edit", new { id=Model.DinnerID })%> |
    <%= Html.ActionLink("Delete Dinner","Delete", new { id=Model.DinnerID})%>

</asp:Content>
```

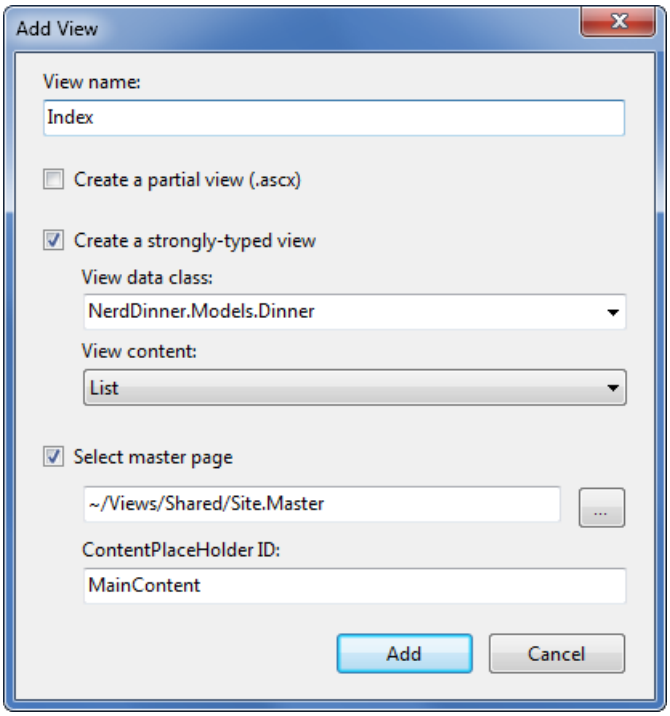
When we access the *"/Dinners/Details/1"* URL again it will now render like below:



Implementing the "Index" View Template

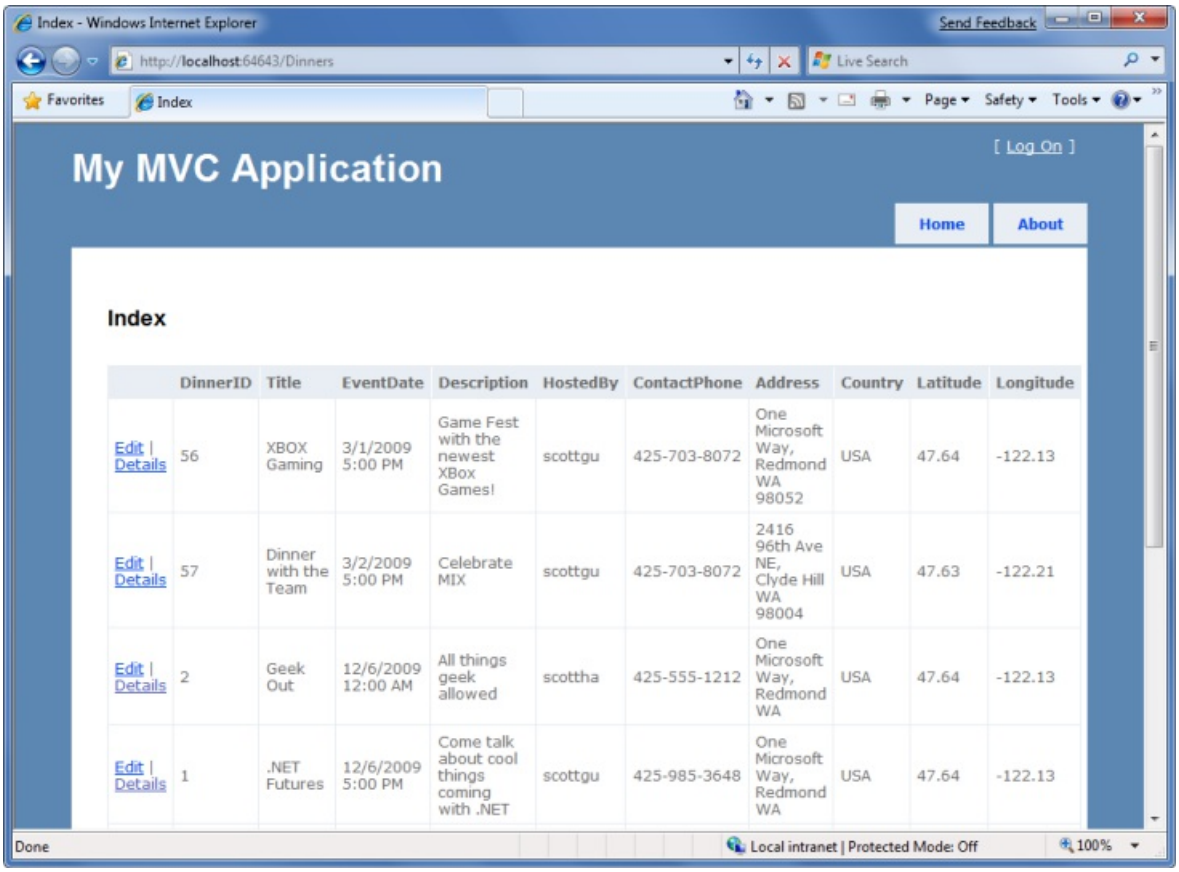
Let's now implement the "Index" view template – which will generate a listing of upcoming Dinners. To-do this we'll position our text cursor within the Index action method, and then right click and choose the "Add View" menu command (or press Ctrl-M, Ctrl-V).

Within the "Add View" dialog we'll keep the view template named "Index" and select the "Create a strongly-typed view" checkbox. This time we will choose to automatically generate a "List" view template, and select "NerdDinner.Models.Dinner" as the model type passed to the view (which because we have indicated we are creating a "List" scaffold will cause the Add View dialog to assume we are passing a sequence of Dinner objects from our Controller to the View):



When we click the "Add" button, Visual Studio will create a new "Index.aspx" view template file for us within our "\Views\Dinners" directory. It will "scaffold" an initial implementation within it that provides an HTML table listing of the Dinners we pass to the view.

When we run the application and access the *"/Dinners/"* URL it will render our list of dinners like so:



The table solution above gives us a grid-like layout of our Dinner data – which isn't quite what we want for our consumer facing Dinner listing. We can update the Index.aspx view template and modify it to list fewer columns of data, and use a <ul> element to render them instead of a table using the code below:

```
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Upcoming Dinners</h2>

    <ul>
        <% foreach (var dinner in Model) { %>

            <li>
                <%=Html.Encode(dinner.Title) %>
                on
                <%=Html.Encode(dinner.EventDate.ToShortDateString())%>
                @
                <%=Html.Encode(dinner.EventDate.ToShortTimeString())%>
            </li>

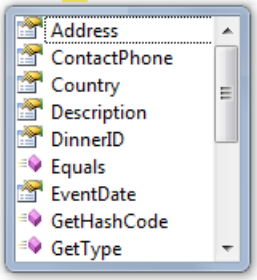
        <% } %>
    </ul>

</asp:Content>
```

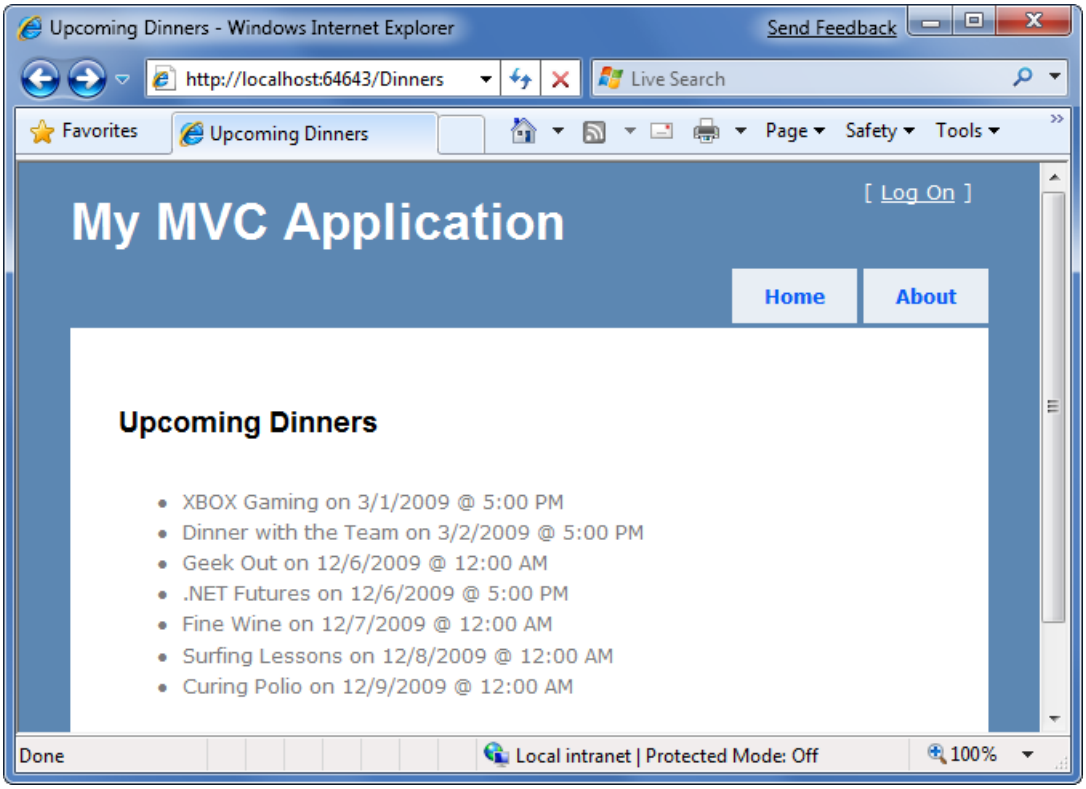
We are using the "var" keyword within the above foreach statement as we loop over each dinner in our Model. Those unfamiliar with C# 3.0 might think that using "var" means that the dinner object is late-bound. It instead means that the compiler is using type-inference against the strongly typed "Model" property (which is of type "IEnumerable<Dinner>") and compiling the local "dinner" variable as a Dinner type – which means we get full intellisense and compile-time checking for it within code blocks:

```
<ul>
    <% foreach (var dinner in Model) { %>

        <li>
            <%= Html.Encode(dinner. %>
```



When we hit refresh on the */Dinners* URL in our browser our updated view now looks like below:



This is looking better – but isn't entirely there yet. Our last step is to enable end-users to click individual Dinners in the list and see details about them. We'll implement this by rendering HTML hyperlink elements that link to the Details action method on our DinnersController.

We can generate these hyperlinks within our Index view in one of two ways. The first is to manually create HTML <a> elements like below, where we embed <% %> blocks within the <a> HTML element:

```
<% foreach (var dinner in Model) { %>

    <li>
        <a href="/Dinners/Details/<%=dinner.DinnerID %>">
            <%= Html.Encode(dinner.Title) %>
        </a>
        on
        <%= Html.Encode(dinner.EventDate.ToShortDateString()) %>
        @
        <%= Html.Encode(dinner.EventDate.ToShortTimeString()) %>
    </li>

<% } %>
```

An alternative approach we can use is to take advantage of the built-in "Html.ActionLink()" helper method within ASP.NET MVC that supports programmatically creating an HTML <a> element that links to another action method

on a Controller:

```
<%= Html.ActionLink(dinner.Title, "Details", new { id=dinner.DinnerID }) %>
```

The first parameter to the Html.ActionLink() helper method is the link-text to display (in this case the title of the dinner), the second parameter is the Controller action name we want to generate the link to (in this case the Details method), and the third parameter is a set of parameters to send to the action (implemented as an anonymous type with property name/values). In this case we are specifying the "id" parameter of the dinner we want to link to, and because the default URL routing rule in ASP.NET MVC is "{Controller}/{Action}/{id}" the Html.ActionLink() helper method will generate the following output:

```
<a href="/Dinners/Details/1">.NET Futures</a>
```

For our Index.aspx view we'll use the Html.ActionLink() helper method approach and have each dinner in the list link to the appropriate details URL:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    Upcoming Dinners
</asp:Content>

<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

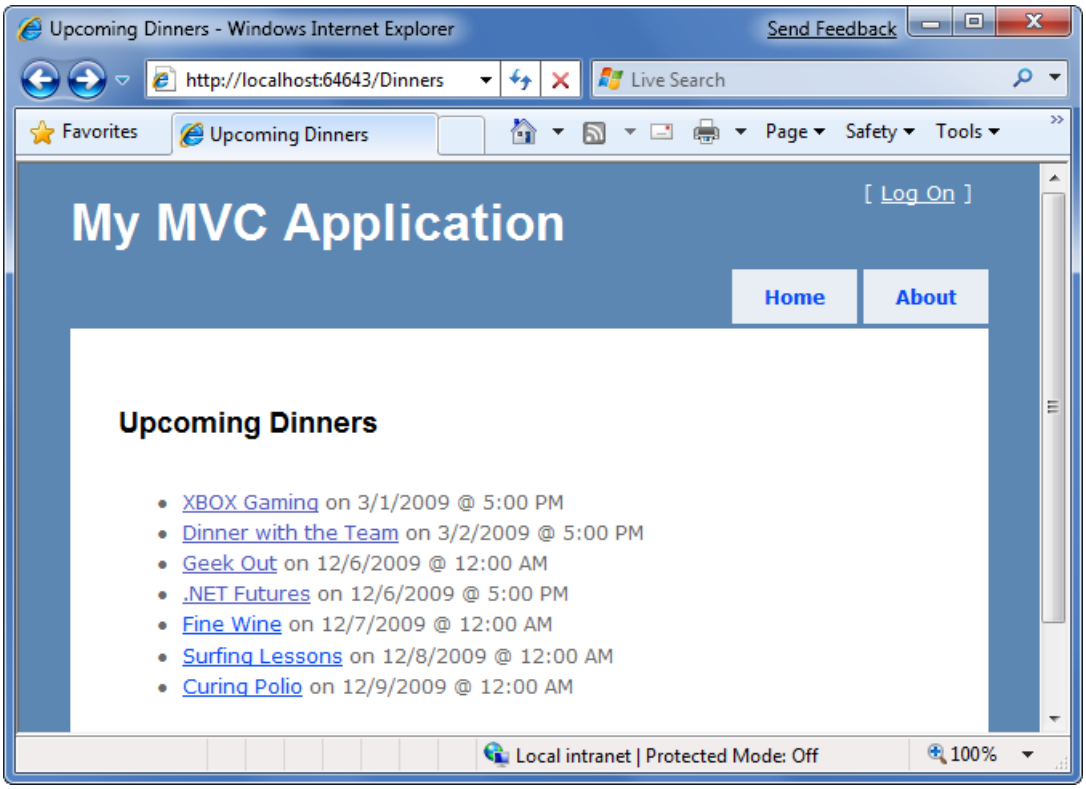
    <h2>Upcoming Dinners</h2>

    <ul>
        <% foreach (var dinner in Model) { %>

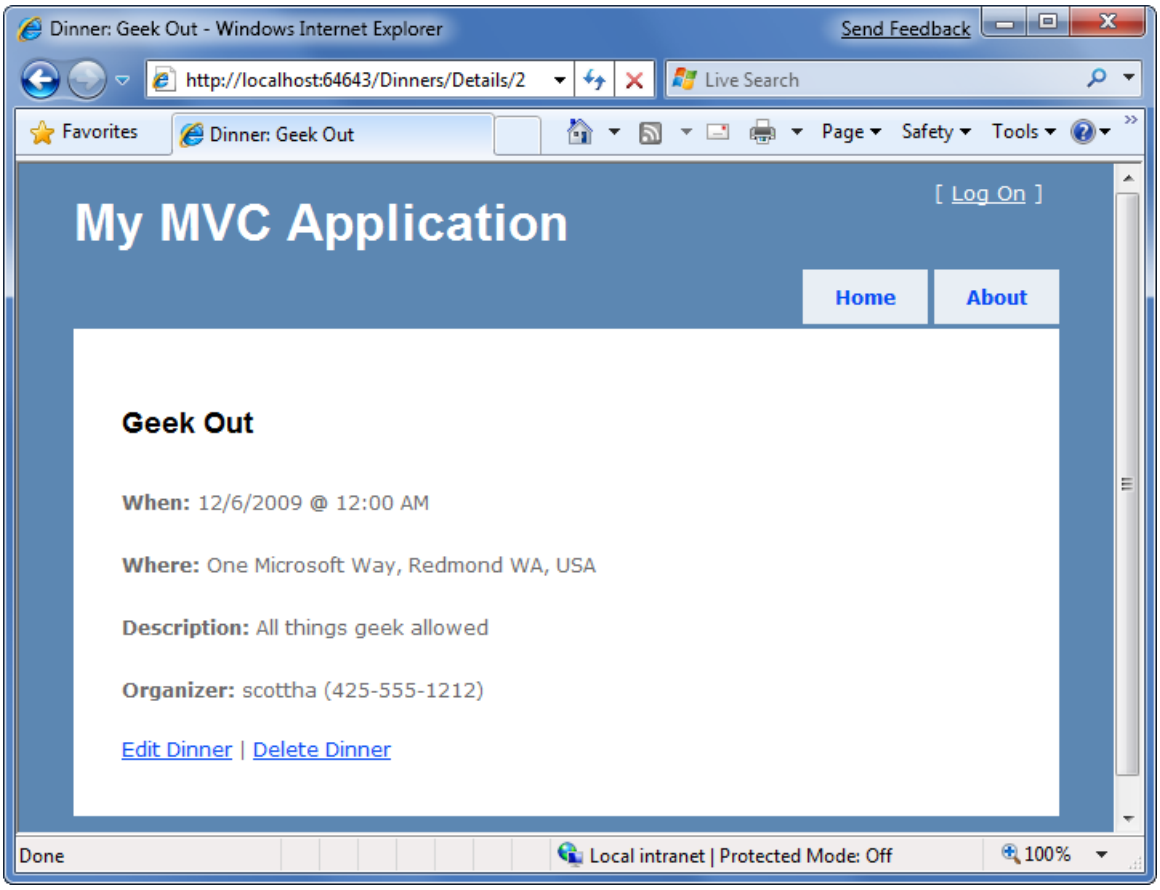
            <li>
                <%=Html.ActionLink(dinner.Title, "Details", new { id=dinner.DinnerID }) %>
                on
                <%=Html.Encode(dinner.EventDate.ToShortDateString())%>
                @
                <%=Html.Encode(dinner.EventDate.ToShortTimeString())%>
            </li>

        <% } %>
    </ul>
</asp:Content>
```

And now when we hit the */Dinners* URL our dinner list looks like below:



When we click any of the Dinners in the list we'll navigate to see details about it:

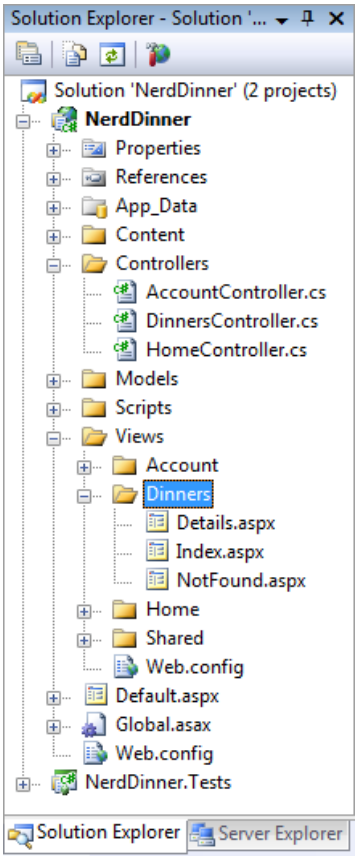


Convention-based naming and the \Views directory structure

ASP.NET MVC applications by default use a convention-based directory naming structure when resolving view templates. This allows developers to avoid having to fully-qualify a location path when referencing views from within a Controller class. By default ASP.NET MVC will look for the view template file within the *\Views[ControllerName]* directory underneath the application.

For example, we've been working on the DinnersController class – which explicitly references three view templates: "Index", "Details" and "NotFound". ASP.NET MVC will by default look for these views within the

Views\Dinners directory underneath our application root directory:



Notice above how there are currently three controller classes within the project (DinnersController, HomeController and AccountController – the last two were added by default when we created the project), and there are three sub-directories (one for each controller) within the \Views directory.

Views referenced from the Home and Accounts controllers will automatically resolve their view templates from the respective *\Views\Home* and *\Views\Account* directories. The *\Views\Shared* sub-directory provides a way to store view templates that are re-used across multiple controllers within the application. When ASP.NET MVC attempts to resolve a view template, it will first check within the *\Views[Controller]* specific directory, and if it can't find the view template there it will look within the *\Views\Shared* directory.

When it comes to naming individual view templates, the recommended guidance is to have the view template share the same name as the action method that caused it to render. For example, above our "Index" action method is using the "Index" view to render the view result, and the "Details" action method is using the "Details" view to render its results. This makes it easy to quickly see which template is associated with each action.

Developers do not need to explicitly specify the view template name when the view template has the same name as the action method being invoked on the controller. We can instead just pass the model object to the "View()" helper method (without specifying the view name), and ASP.NET MVC will automatically infer that we want to use the *\Views[ControllerName][ActionName]* view template on disk to render it.

This allows us to clean up our controller code a little, and avoid duplicating the name twice in our code:

```
public class DinnersController : Controller {

    DinnerRepository dinnerRepository = new DinnerRepository();

    //
    // GET: /Dinners/

    public ActionResult Index() {

        var dinners = dinnerRepository.FindUpcomingDinners().ToList();

        return View(dinners);
    }

    //
    // GET: /Dinners/Details/2

    public ActionResult Details(int id) {

        Dinner dinner = dinnerRepository.GetDinner(id);

        if (dinner == null)
            return View("NotFound");
        else
            return View(dinner);
    }
}
```

The above code is all that is needed to implement a nice Dinner listing/details experience for the site.

Next Step

We now have a nice Dinner browsing experience built.

Let's now enable CRUD (Create, Read, Update, Delete) data form editing support.

PREVIOUS NEXT

Provide CRUD (Create, Read, Update, Delete) Data Form Entry Support

4/10/2020 • 29 minutes to read • Edit Online

by Microsoft

Download PDF

> This is step 5 of a free "NerdDinner" application tutorial that walks-through how to build a small, but complete, web application using ASP.NET MVC 1.
>
> Step 5 shows how to take our DinnersController class further by enable support for editing, creating and deleting Dinners with it as well.
>
> If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

NerdDinner Step 5: Create, Update, Delete Form Scenarios

We've introduced controllers and views, and covered how to use them to implement a listing/details experience for Dinners on site. Our next step will be to take our DinnersController class further and enable support for editing, creating and deleting Dinners with it as well.

URLs handled by DinnersController

We previously added action methods to DinnersController that implemented support for two URLs: */Dinners* and */Dinners/Details/[id]*.

URL	VERB	PURPOSE
/Dinners/	GET	Display an HTML list of upcoming dinners.
/Dinners/Details/[id]	GET	Display details about a specific dinner.

We will now add action methods to implement three additional URLs: */Dinners/Edit/[id]*, */Dinners/Create*, and */Dinners/Delete/[id]*. These URLs will enable support for editing existing Dinners, creating new Dinners, and deleting Dinners.

We will support both HTTP GET and HTTP POST verb interactions with these new URLs. HTTP GET requests to these URLs will display the initial HTML view of the data (a form populated with the Dinner data in the case of "edit", a blank form in the case of "create", and a delete confirmation screen in the case of "delete"). HTTP POST requests to these URLs will save/update/delete the Dinner data in our DinnerRepository (and from there to the database).

URL	VERB	PURPOSE
/Dinners/Edit/[id]	GET	Display an editable HTML form populated with Dinner data.

URL	VERB	PURPOSE
POST	Save the form changes for a particular Dinner to the database.	
/Dinners/Create	GET	Display an empty HTML form that allows users to define new Dinners.
POST	Create a new Dinner and save it in the database.	
/Dinners/Delete/[id]	GET	Display delete confirmation screen.
POST	Deletes the specified dinner from the database.	

Edit Support

Let's begin by implementing the "edit" scenario.

The HTTP-GET Edit Action Method

We'll start by implementing the HTTP "GET" behavior of our edit action method. This method will be invoked when the */Dinners/Edit/[id]* URL is requested. Our implementation will look like:

```
//
// GET: /Dinners/Edit/2

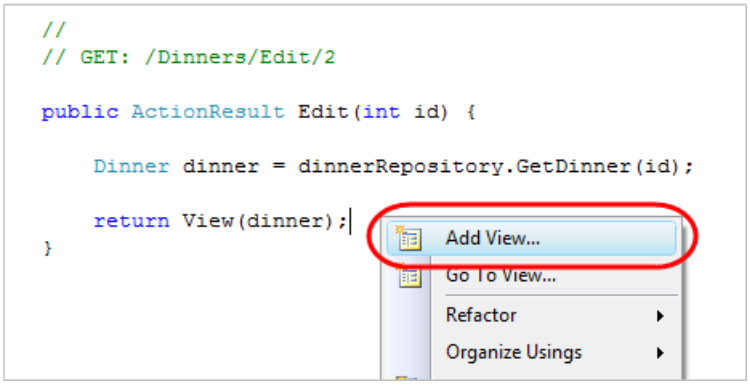
public ActionResult Edit(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

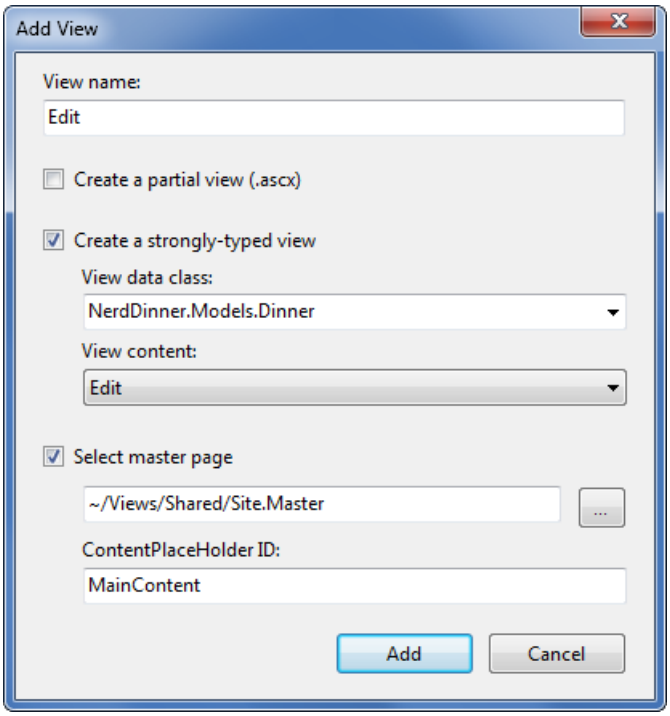
    return View(dinner);
}
```

The code above uses the DinnerRepository to retrieve a Dinner object. It then renders a View template using the Dinner object. Because we haven't explicitly passed a template name to the *View()* helper method, it will use the convention based default path to resolve the view template: /Views/Dinners/Edit.aspx.

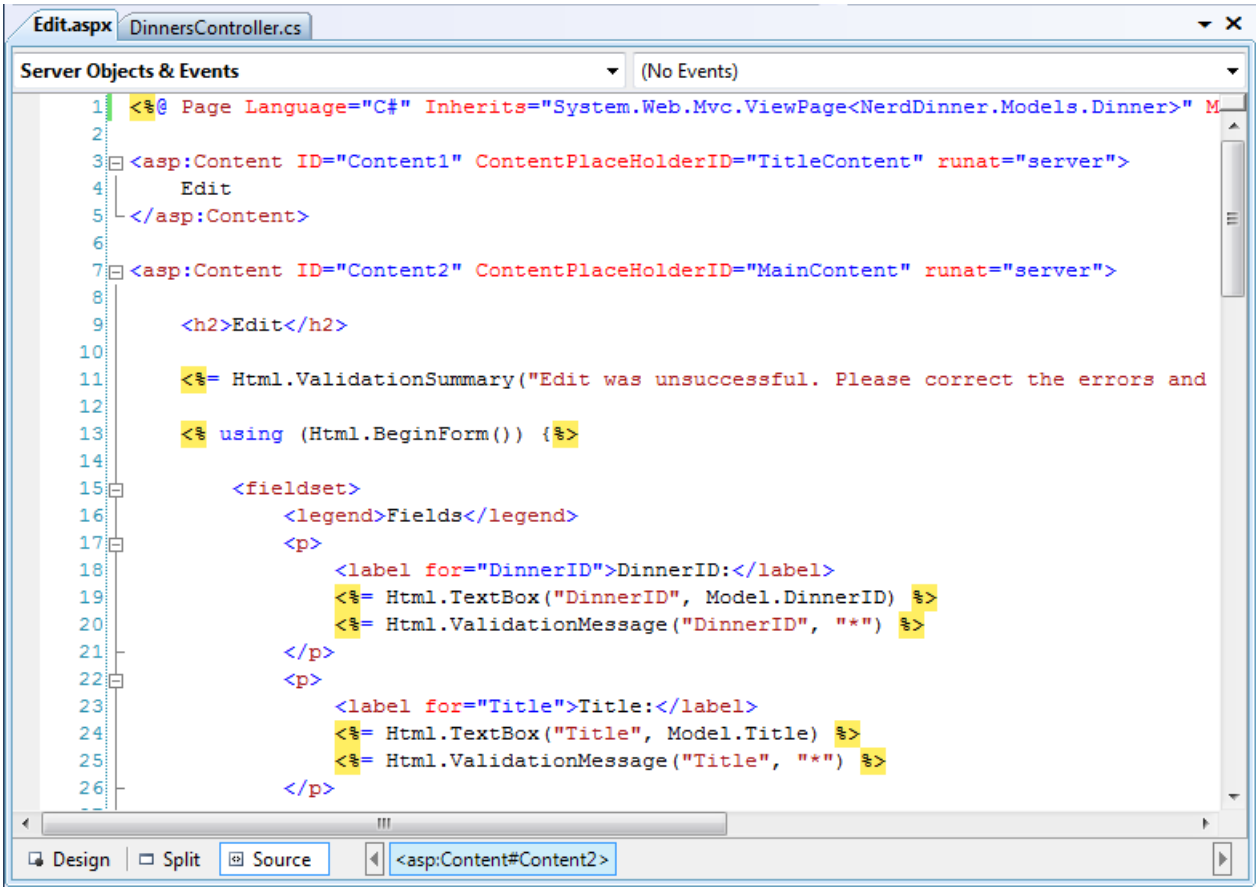
Let's now create this view template. We will do this by right-clicking within the Edit method and selecting the "Add View" context menu command:



Within the "Add View" dialog we'll indicate that we are passing a Dinner object to our view template as its model, and choose to auto-scaffold an "Edit" template:



When we click the "Add" button, Visual Studio will add a new "Edit.aspx" view template file for us within the "\Views\Dinners" directory. It will also open up the new "Edit.aspx" view template within the code-editor – populated with an initial "Edit" scaffold implementation like below:



Let's make a few changes to the default "edit" scaffold generated, and update the edit view template to have the content below (which removes a few of the properties we don't want to expose):

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    Edit: <%=Html.Encode(Model.Title)%>
</asp:Content>

<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Edit Dinner</h2>

    <%=Html.ValidationSummary("Please correct the errors and try again.") %>

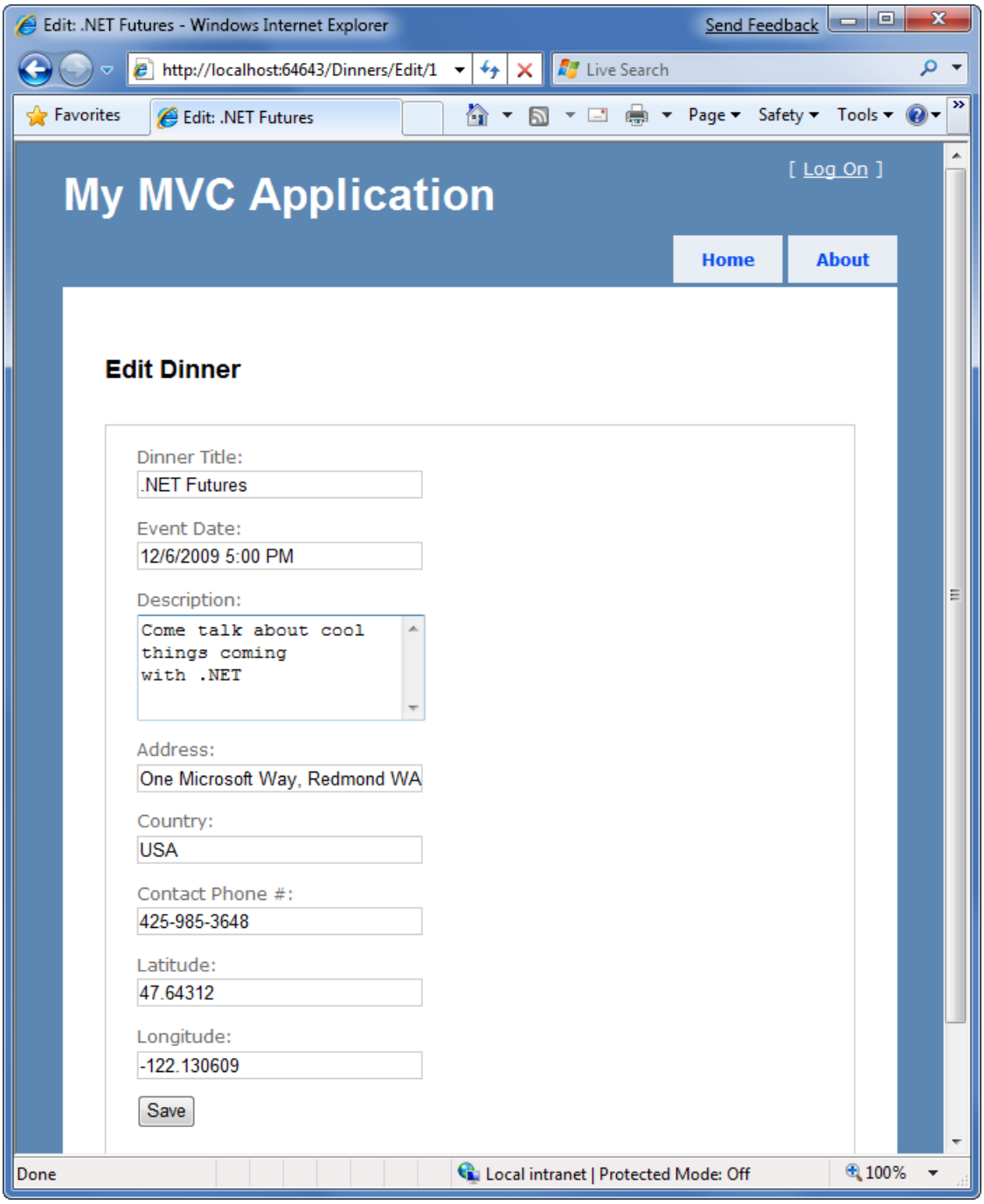
    <% using (Html.BeginForm()) { %>

        <fieldset>
            <p>
                <label for="Title">Dinner Title:</label>
                <%=Html.TextBox("Title") %>
                <%=Html.ValidationMessage("Title", "*") %>
            </p>
            <p>
                <label for="EventDate">EventDate:</label>
                <%=Html.TextBox("EventDate", String.Format("{0:g}", Model.EventDate))%>
                <%=Html.ValidationMessage("EventDate", "*") %>
            </p>
            <p>
                <label for="Description">Description:</label>
                <%=Html.TextArea("Description") %>
                <%=Html.ValidationMessage("Description", "*")%>
            </p>
            <p>
                <label for="Address">Address:</label>
                <%=Html.TextBox("Address") %>
                <%=Html.ValidationMessage("Address", "*") %>
            </p>
            <p>
                <label for="Country">Country:</label>
                <%=Html.TextBox("Country") %>
                <%=Html.ValidationMessage("Country", "*") %>
            </p>
            <p>
                <label for="ContactPhone">ContactPhone #:</label>
                <%=Html.TextBox("ContactPhone") %>
                <%=Html.ValidationMessage("ContactPhone", "*") %>
            </p>
            <p>
                <label for="Latitude">Latitude:</label>
                <%=Html.TextBox("Latitude") %>
                <%=Html.ValidationMessage("Latitude", "*") %>
            </p>
            <p>
                <label for="Longitude">Longitude:</label>
                <%=Html.TextBox("Longitude") %>
                <%=Html.ValidationMessage("Longitude", "*") %>
            </p>
            <p>
                <input type="submit" value="Save"/>
            </p>
        </fieldset>

    <% } %>

</asp:Content>
```

When we run the application and request the *"/Dinners/Edit/1"* URL we will see the following page:



The HTML markup generated by our view looks like below. It is standard HTML – with a <form> element that performs an HTTP POST to the */Dinners/Edit/1* URL when the "Save" <input type="submit"/> button is pushed. A HTML <input type="text"/> element has been output for each editable property:

```
<form action="/Dinners/Edit/1" method="post">

    <fieldset>
        <p>
            <label for="Title">Dinner Title:</label>
            <input id="Title" name="Title" type="text" value=".NET Futures" />
        </p>
        <p>
            <label for="EventDate">Event Date:</label>
            <input id="EventDate" name="EventDate" type="text" value="12/6/2009 5:00 PM" />
        </p>

        <!-- Some Fields Omitted for Brevity -->

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>

</form>
```

Html.BeginForm() and Html.TextBox() Html Helper Methods

Our "Edit.aspx" view template is using several "Html Helper" methods: Html.ValidationSummary(), Html.BeginForm(), Html.TextBox(), and Html.ValidationMessage(). In addition to generating HTML markup for us, these helper methods provide built-in error handling and validation support.

Html.BeginForm() helper method

The Html.BeginForm() helper method is what output the HTML <form> element in our markup. In our Edit.aspx view template you'll notice that we are applying a C# "using" statement when using this method. The open curly brace indicates the beginning of the <form> content, and the closing curly brace is what indicates the end of the </form> element:

```
<% using (Html.BeginForm()) { %>

   <fieldset>

      <!-- Fields Omitted for Brevity -->

      <p>
         <input type="submit" value="Save"/>
      </p>
   </fieldset>

<% } %>
```

Alternatively, if you find the "using" statement approach unnatural for a scenario like this, you can use a Html.BeginForm() and Html.EndForm() combination (which does the same thing):

```
<% Html.BeginForm();  %>

   <fieldset>

      <!-- Fields Omitted for Brevity -->

      <p>
          <input type="submit" value="Save"/>
      </p>
   </fieldset>

<% Html.EndForm(); %>
```

Calling Html.BeginForm() without any parameters will cause it to output a form element that does an HTTP-POST to the current request's URL. That is why our Edit view generates a *<form action="/Dinners/Edit/1"*

method="post"> element. We could have alternatively passed explicit parameters to Html.BeginForm() if we wanted to post to a different URL.

Html.TextBox() helper method

Our Edit.aspx view uses the Html.TextBox() helper method to output <input type="text"/> elements:

```
<%= Html.TextBox("Title") %>
```

The Html.TextBox() method above takes a single parameter – which is being used to specify both the id/name attributes of the <input type="text"/> element to output, as well as the model property to populate the textbox value from. For example, the Dinner object we passed to the Edit view had a "Title" property value of ".NET Futures", and so our Html.TextBox("Title") method call output: *<input id="Title" name="Title" type="text" value=".NET Futures" />*.

Alternatively, we can use the first Html.TextBox() parameter to specify the id/name of the element, and then explicitly pass in the value to use as a second parameter:

```
<%= Html.TextBox("Title", Model.Title)%>
```

Often we'll want to perform custom formatting on the value that is output. The String.Format() static method built-into .NET is useful for these scenarios. Our Edit.aspx view template is using this to format the EventDate value (which is of type DateTime) so that it doesn't show seconds for the time:

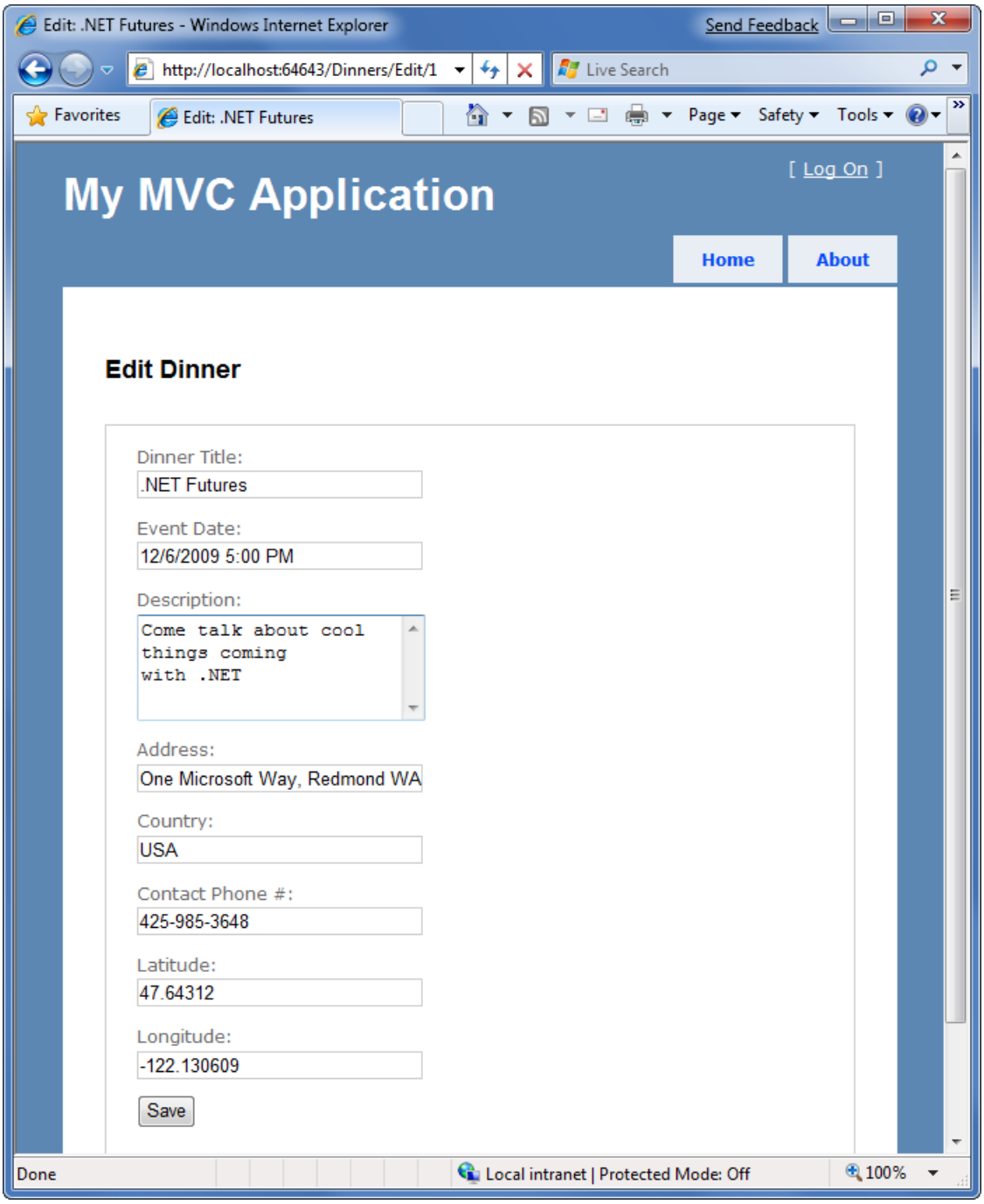
```
<%= Html.TextBox("EventDate", String.Format("{0:g}", Model.EventDate)) %>
```

A third parameter to Html.TextBox() can optionally be used to output additional HTML attributes. The code-snippet below demonstrates how to render an additional size="30" attribute and a class="mycssclass" attribute on the <input type="text"/> element. Note how we are escaping the name of the class attribute using a "@" character because "class" is a reserved keyword in C#:

```
<%= Html.TextBox("Title", Model.Title, new { size=30, @class="myclass" } )%>
```

Implementing the HTTP-POST Edit Action Method

We now have the HTTP-GET version of our Edit action method implemented. When a user requests the */Dinners/Edit/1* URL they receive an HTML page like the following:



Pressing the "Save" button causes a form post to the */Dinners/Edit/1* URL, and submits the HTML <input> form values using the HTTP POST verb. Let's now implement the HTTP POST behavior of our edit action method – which will handle saving the Dinner.

We'll begin by adding an overloaded "Edit" action method to our DinnersController that has an "AcceptVerbs" attribute on it that indicates it handles HTTP POST scenarios:

```
//
// POST: /Dinners/Edit/2

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {
   ...
}
```

When the [AcceptVerbs] attribute is applied to overloaded action methods, ASP.NET MVC automatically handles dispatching requests to the appropriate action method depending on the incoming HTTP verb. HTTP POST requests to */Dinners/Edit/[id]* URLs will go to the above Edit method, while all other HTTP verb requests to

/Dinners/Edit/[id] URLs will go to the first Edit method we implemented (which did not have an `[AcceptVerbs]` attribute).

SIDE TOPIC: WHY DIFFERENTIATE VIA HTTP VERBS?

You might ask – why are we using a single URL and differentiating its behavior via the HTTP verb? Why not just have two separate URLs to handle loading and saving edit changes? For example: /Dinners/Edit/[id] to display the initial form and /Dinners/Save/[id] to handle the form post to save it? The downside with publishing two separate URLs is that in cases where we post to /Dinners/Save/2, and then need to redisplay the HTML form because of an input error, the end-user will end up having the /Dinners/Save/2 URL in their browser's address bar (since that was the URL the form posted to). If the end-user bookmarks this redisplayed page to their browser favorites list, or copy/pastes the URL and emails it to a friend, they will end up saving a URL that won't work in the future (since that URL depends on post values). By exposing a single URL (like: /Dinners/Edit/[id]) and differentiating the processing of it by HTTP verb, it is safe for end-users to bookmark the edit page and/or send the URL to others.

Retrieving Form Post Values

There are a variety of ways we can access posted form parameters within our HTTP POST "Edit" method. One simple approach is to just use the Request property on the Controller base class to access the form collection and retrieve the posted values directly:

```
//
// POST: /Dinners/Edit/2

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {

    // Retrieve existing dinner
    Dinner dinner = dinnerRepository.GetDinner(id);

    // Update dinner with form posted values
    dinner.Title = Request.Form["Title"];
    dinner.Description = Request.Form["Description"];
    dinner.EventDate = DateTime.Parse(Request.Form["EventDate"]);
    dinner.Address = Request.Form["Address"];
    dinner.Country = Request.Form["Country"];
    dinner.ContactPhone = Request.Form["ContactPhone"];

    // Persist changes back to database
    dinnerRepository.Save();

    // Perform HTTP redirect to details page for the saved Dinner
    return RedirectToAction("Details", new { id = dinner.DinnerID });
}
```

The above approach is a little verbose, though, especially once we add error handling logic.

A better approach for this scenario is to leverage the built-in *UpdateModel()* helper method on the Controller base class. It supports updating the properties of an object we pass it using the incoming form parameters. It uses reflection to determine the property names on the object, and then automatically converts and assigns values to them based on the input values submitted by the client.

We could use the UpdateModel() method to simplify our HTTP-POST Edit Action using this code:

```
//
// POST: /Dinners/Edit/2

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {

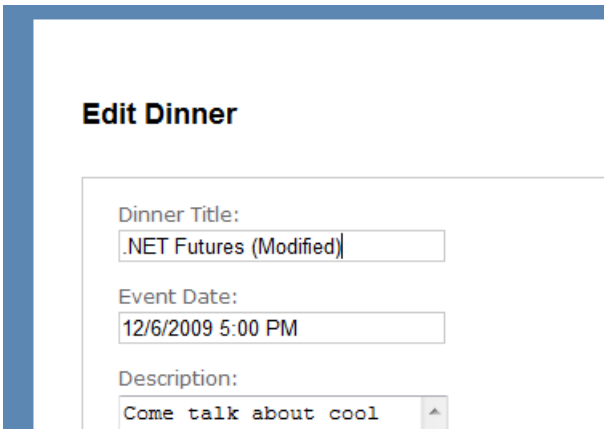
    Dinner dinner = dinnerRepository.GetDinner(id);

    UpdateModel(dinner);

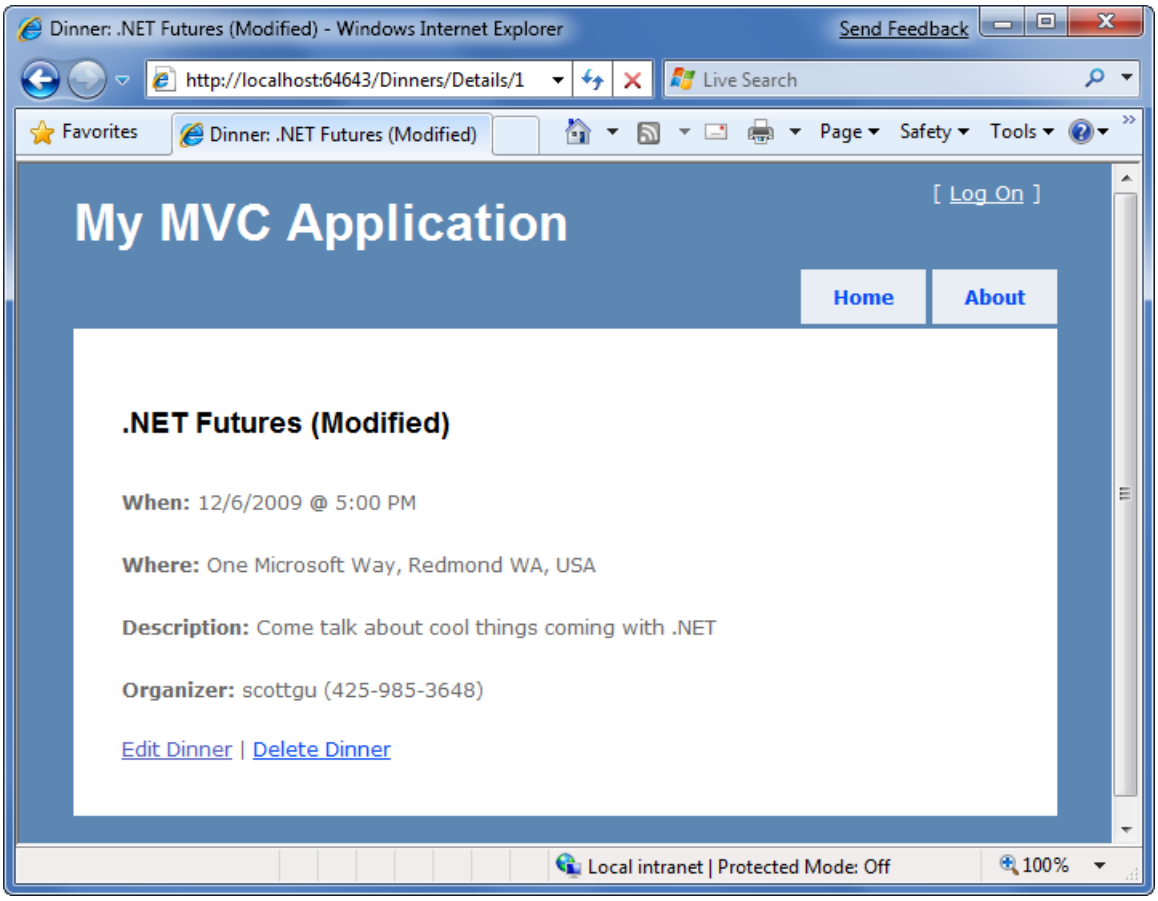
    dinnerRepository.Save();

    return RedirectToAction("Details", new { id = dinner.DinnerID });
}
```

We can now visit the */Dinners/Edit/1* URL, and change the title of our Dinner:



When we click the "Save" button, we'll perform a form post to our Edit action, and the updated values will be persisted in the database. We will then be redirected to the Details URL for the Dinner (which will display the newly saved values):



Handling Edit Errors

Our current HTTP-POST implementation works fine – except when there are errors.

When a user makes a mistake editing a form, we need to make sure that the form is redisplayed with an informative error message that guides them to fix it. This includes cases where an end-user posts incorrect input (for example: a malformed date string), as well as cases where the input format is valid but there is a business rule violation. When errors occur the form should preserve the input data the user originally entered so that they don't have to refill their changes manually. This process should repeat as many times as necessary until the form successfully completes.

ASP.NET MVC includes some nice built-in features that make error handling and form redisplay easy. To see these features in action let's update our Edit action method with the following code:

```
//
// POST: /Dinners/Edit/2

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    try {

        UpdateModel(dinner);

        dinnerRepository.Save();

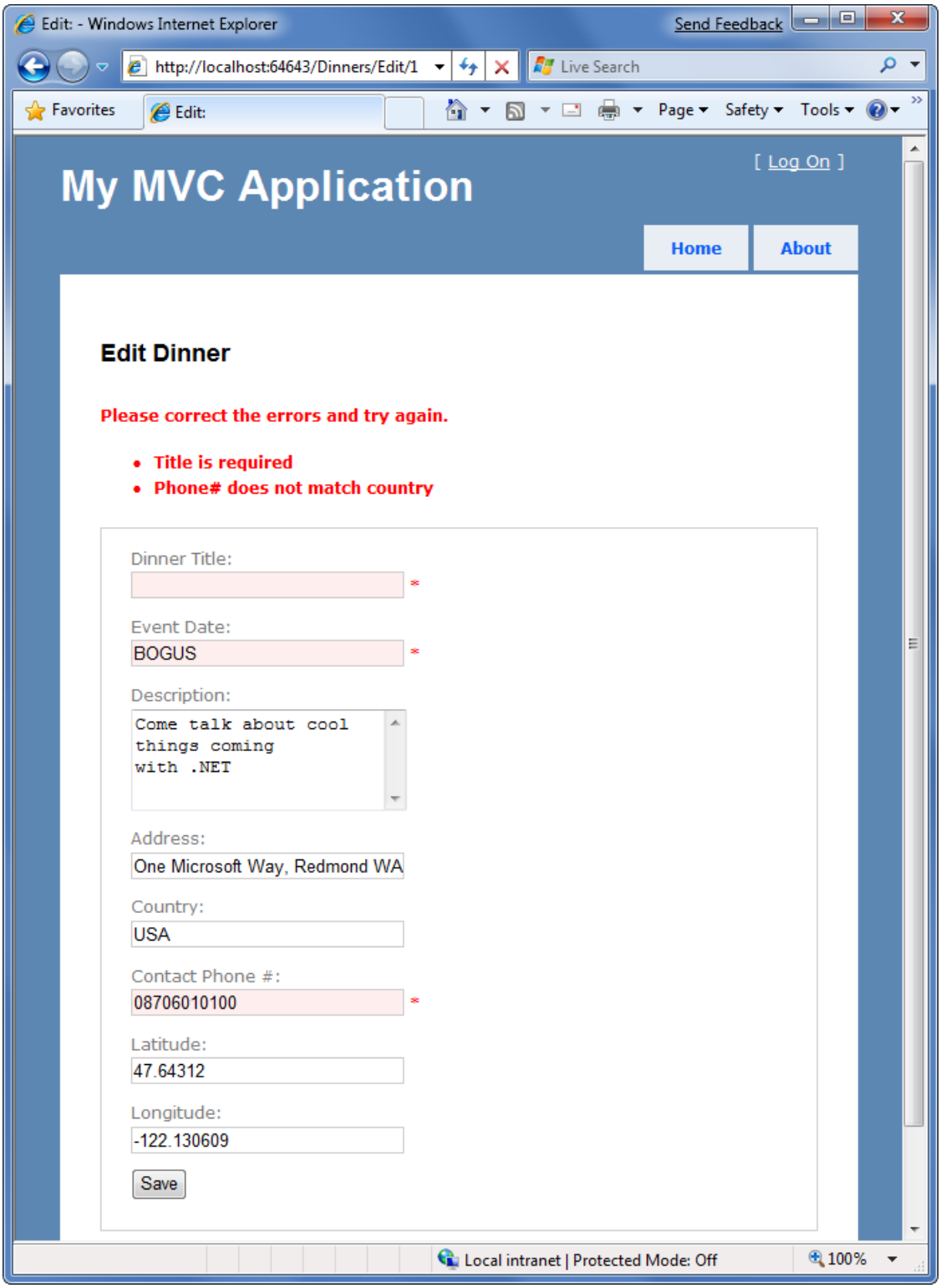
        return RedirectToAction("Details", new { id=dinner.DinnerID });
    }
    catch {

        foreach (var issue in dinner.GetRuleViolations()) {
            ModelState.AddModelError(issue.PropertyName, issue.ErrorMessage);
        }

        return View(dinner);
    }
}
```

The above code is similar to our previous implementation – except that we are now wrapping a try/catch error handling block around our work. If an exception occurs either when calling UpdateModel(), or when we try and save the DinnerRepository (which will raise an exception if the Dinner object we are trying to save is invalid because of a rule violation within our model), our catch error handling block will execute. Within it we loop over any rule violations that exist in the Dinner object and add them to a ModelState object (which we'll discuss shortly). We then redisplay the view.

To see this working let's re-run the application, edit a Dinner, and change it to have an empty Title, an EventDate of "BOGUS", and use a UK phone number with a country value of USA. When we press the "Save" button our HTTP POST Edit method will not be able to save the Dinner (because there are errors) and will redisplay the form:



Our application has a decent error experience. The text elements with the invalid input are highlighted in red, and validation error messages are displayed to the end user about them. The form is also preserving the input data the user originally entered – so that they don't have to refill anything.

How, you might ask, did this occur? How did the Title, EventDate, and ContactPhone textboxes highlight themselves in red and know to output the originally entered user values? And how did error messages get displayed in the list at the top? The good news is that this didn't occur by magic - rather it was because we used some of the built-in ASP.NET MVC features that make input validation and error handling scenarios easy.

Understanding ModelState and the Validation HTML Helper Methods

Controller classes have a "ModelState" property collection which provides a way to indicate that errors exist with a model object being passed to a View. Error entries within the ModelState collection identify the name of the model property with the issue (for example: "Title", "EventDate", or "ContactPhone"), and allow a human-friendly error message to be specified (for example: "Title is required").

The *UpdateModel()* helper method automatically populates the ModelState collection when it encounters errors while trying to assign form values to properties on the model object. For example, our Dinner object's EventDate property is of type DateTime. When the UpdateModel() method was unable to assign the string value "BOGUS" to it in the scenario above, the UpdateModel() method added an entry to the ModelState collection indicating an assignment error had occurred with that property.

Developers can also write code to explicitly add error entries into the ModelState collection like we are doing below within our "catch" error handling block, which is populating the ModelState collection with entries based on the active Rule Violations in the Dinner object:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    try {

        UpdateModel(dinner);

        dinnerRepository.Save();

        return RedirectToAction("Details", new { id=dinner.DinnerID });
    }
    catch {

        foreach (var issue in dinner.GetRuleViolations()) {
            ModelState.AddModelError(issue.PropertyName, issue.ErrorMessage);
        }

        return View(dinner);
    }
}
```

Html Helper Integration with ModelState

HTML helper methods - like Html.TextBox() - check the ModelState collection when rendering output. If an error for the item exists, they render the user-entered value and a CSS error class.

For example, in our "Edit" view we are using the Html.TextBox() helper method to render the EventDate of our Dinner object:

```
<%= Html.TextBox("EventDate", String.Format("{0:g}", Model.EventDate)) %>
```

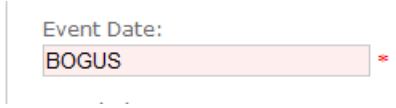
When the view was rendered in the error scenario, the Html.TextBox() method checked the ModelState collection to see if there were any errors associated with the "EventDate" property of our Dinner object. When it determined that there was an error it rendered the submitted user input ("BOGUS") as the value, and added a css error class to the <input type="textbox"/> markup it generated:

```
<input class="input-validation-error"id="EventDate" name="EventDate" type="text" value="BOGUS"/>
```

You can customize the appearance of the css error class to look however you want. The default CSS error class – "input-validation-error" – is defined in the *\content\site.css* stylesheet and looks like below:

```
.input-validation-error
{
    border: 1px solid #ff0000;
    background-color: #ffeeee;
}
```

This CSS rule is what caused our invalid input elements to be highlighted like below:



Html.ValidationMessage() Helper Method

The Html.ValidationMessage() helper method can be used to output the ModelState error message associated with a particular model property:

```
<%= Html.ValidationMessage("EventDate")%>
```

The above code outputs: * The value 'BOGUS' is invalid*

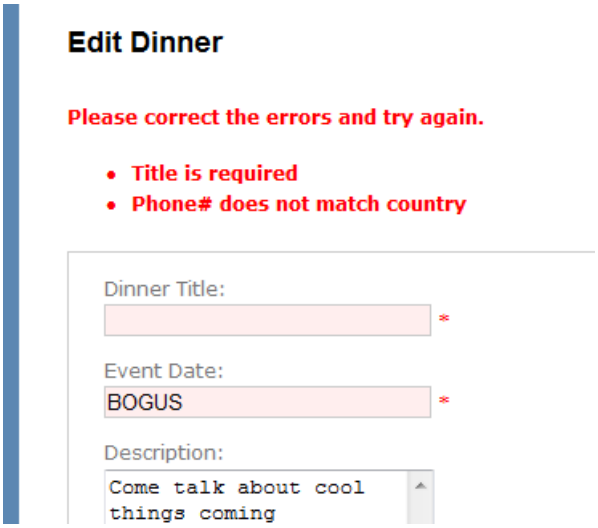
The Html.ValidationMessage() helper method also supports a second parameter that allows developers to override the error text message that is displayed:

```
<%= Html.ValidationMessage("EventDate","*") %>
```

The above code outputs: *** instead of the default error text when an error is present for the EventDate property.

Html.ValidationSummary() Helper Method

The Html.ValidationSummary() helper method can be used to render a summary error message, accompanied by a list of all detailed error messages in the ModelState collection:



The Html.ValidationSummary() helper method takes an optional string parameter – which defines a summary error message to display above the list of detailed errors:

```
<%= Html.ValidationSummary("Please correct the errors and try again.") %>
```

You can optionally use CSS to override what the error list looks like.

Using a AddRuleViolations Helper Method

Our initial HTTP-POST Edit implementation used a foreach statement within its catch block to loop over the Dinner object's Rule Violations and add them to the controller's ModelState collection:

```
catch {
        foreach (var issue in dinner.GetRuleViolations()) {
            ModelState.AddModelError(issue.PropertyName, issue.ErrorMessage);
        }

        return View(dinner);
    }
```

We can make this code a little cleaner by adding a "ControllerHelpers" class to the NerdDinner project, and implement an "AddRuleViolations" extension method within it that adds a helper method to the ASP.NET MVC ModelStateDictionary class. This extension method can encapsulate the logic necessary to populate the ModelStateDictionary with a list of RuleViolation errors:

```
public static class ControllerHelpers {

   public static void AddRuleViolations(this ModelStateDictionary modelState, IEnumerable<RuleViolation>
errors) {

       foreach (RuleViolation issue in errors) {
           modelState.AddModelError(issue.PropertyName, issue.ErrorMessage);
       }
   }
}
```

We can then update our HTTP-POST Edit action method to use this extension method to populate the ModelState collection with our Dinner Rule Violations.

Complete Edit Action Method Implementations

The code below implements all of the controller logic necessary for our Edit scenario:

```
//
// GET: /Dinners/Edit/2

public ActionResult Edit(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    return View(dinner);
}

//
// POST: /Dinners/Edit/2

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection formValues) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    try {

        UpdateModel(dinner);

        dinnerRepository.Save();

        return RedirectToAction("Details", new { id=dinner.DinnerID });
    }
    catch {

        ModelState.AddRuleViolations(dinner.GetRuleViolations());

        return View(dinner);
    }
}
```

The nice thing about our Edit implementation is that neither our Controller class nor our View template has to know anything about the specific validation or business rules being enforced by our Dinner model. We can add additional rules to our model in the future and do not have to make any code changes to our controller or view in order for them to be supported. This provides us with the flexibility to easily evolve our application requirements in the future with a minimum of code changes.

Create Support

We've finished implementing the "Edit" behavior of our DinnersController class. Let's now move on to implement the "Create" support on it – which will enable users to add new Dinners.

The HTTP-GET Create Action Method

We'll begin by implementing the HTTP "GET" behavior of our create action method. This method will be called when someone visits the */Dinners/Create* URL. Our implementation looks like:

```
//
// GET: /Dinners/Create

public ActionResult Create() {

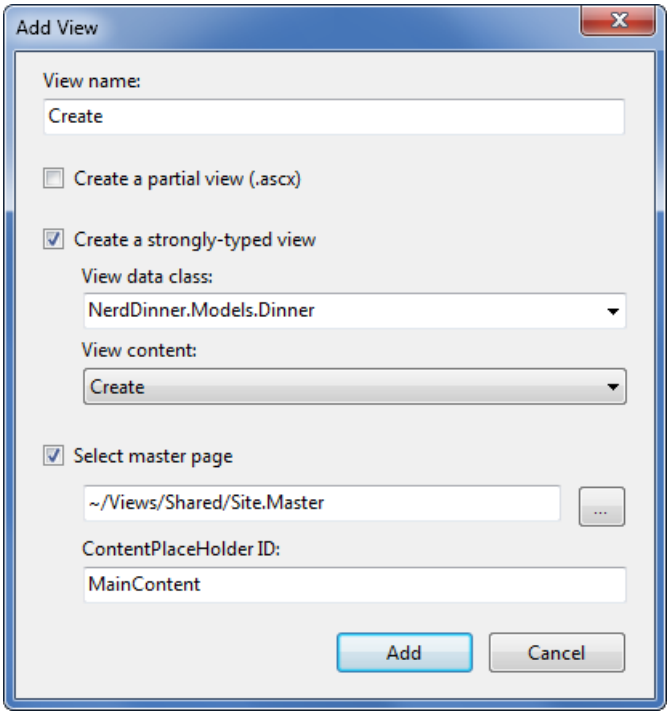
    Dinner dinner = new Dinner() {
        EventDate = DateTime.Now.AddDays(7)
    };

    return View(dinner);
}
```

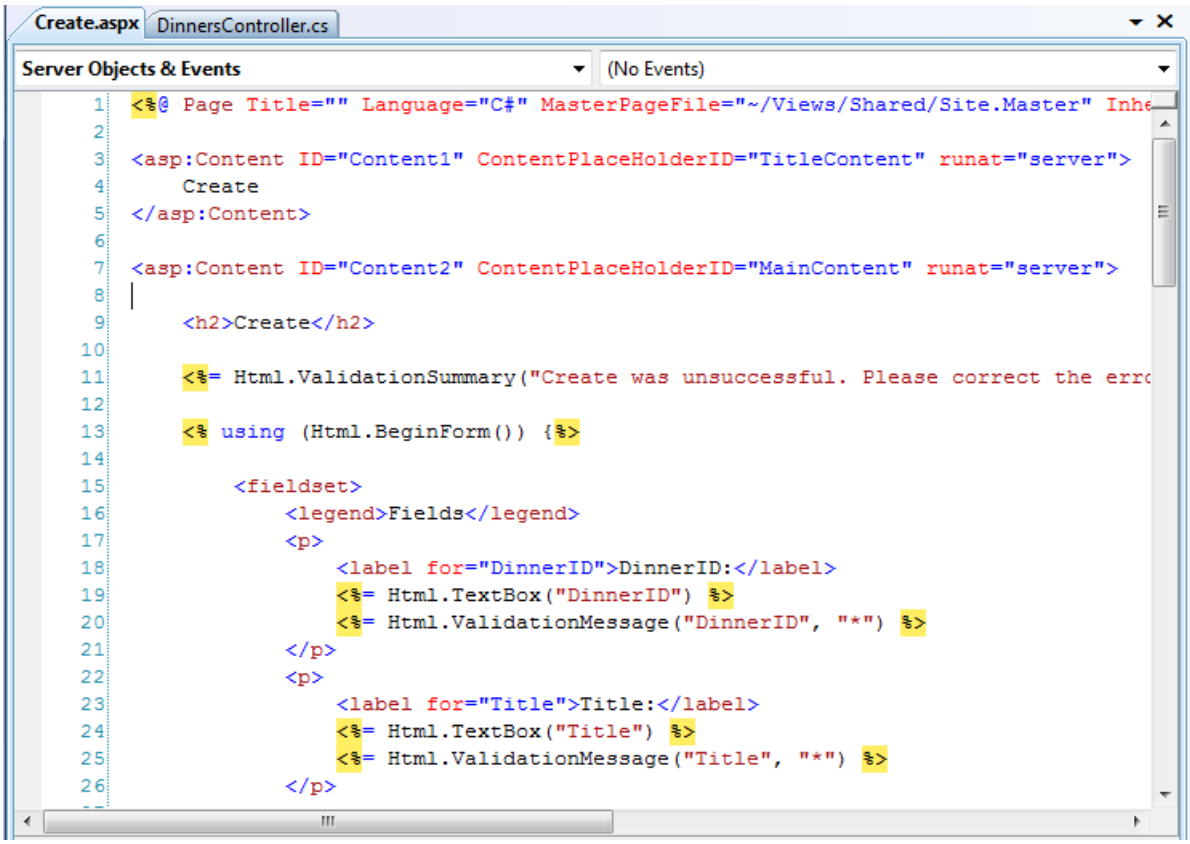
The code above creates a new Dinner object, and assigns its EventDate property to be one week in the future. It then renders a View that is based on the new Dinner object. Because we haven't explicitly passed a name to the

View() helper method, it will use the convention based default path to resolve the view template: /Views/Dinners/Create.aspx.

Let's now create this view template. We can do this by right-clicking within the Create action method and selecting the "Add View" context menu command. Within the "Add View" dialog we'll indicate that we are passing a Dinner object to the view template, and choose to auto-scaffold a "Create" template:



When we click the "Add" button, Visual Studio will save a new scaffold-based "Create.aspx" view to the "\Views\Dinners" directory, and open it up within the IDE:



Let's make a few changes to the default "create" scaffold file that was generated for us, and modify it up to look like below:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
     Host a Dinner
</asp:Content>

<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

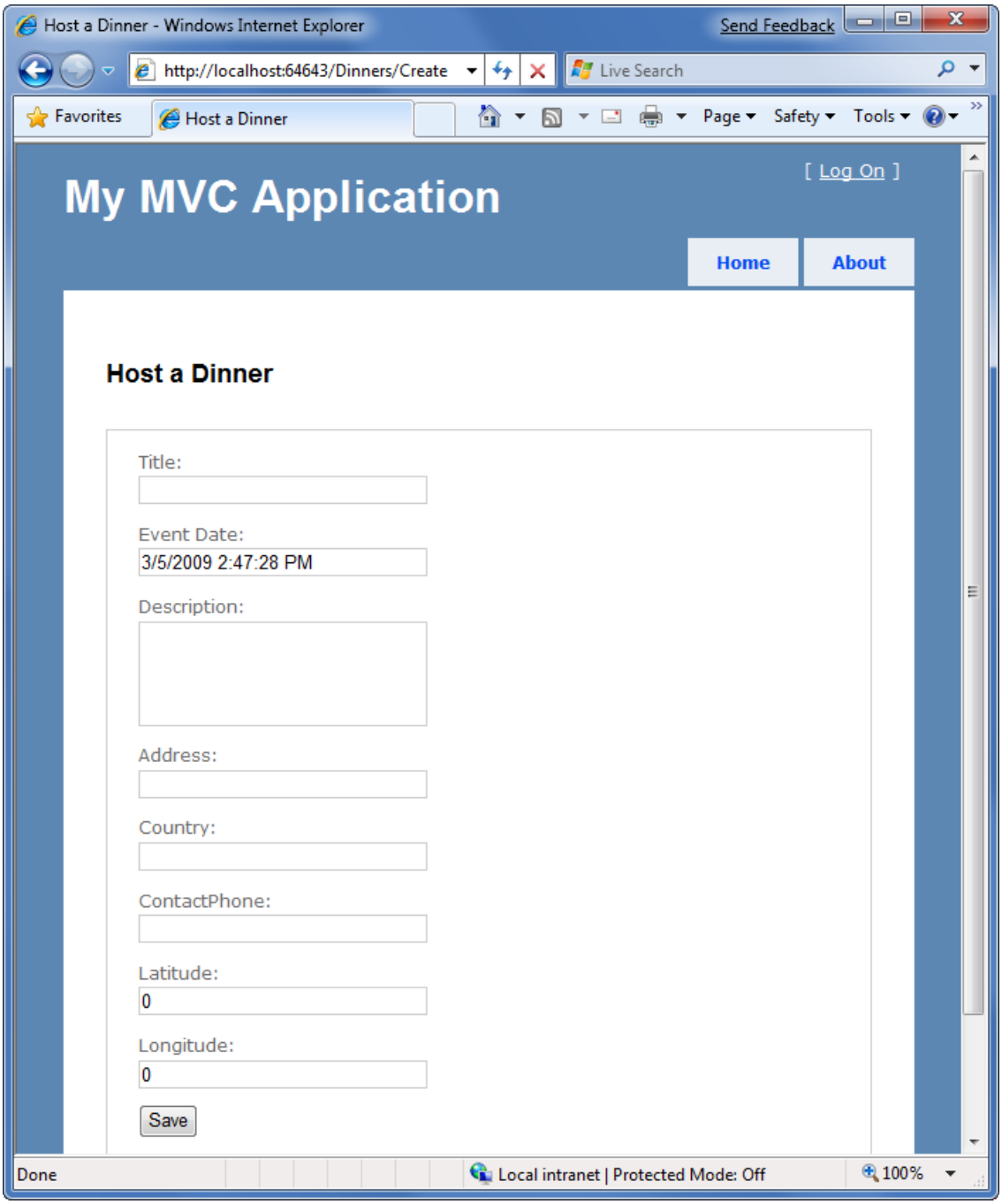
    <h2>Host a Dinner</h2>

    <%=Html.ValidationSummary("Please correct the errors and try again.") %>

    <% using (Html.BeginForm()) {%>

        <fieldset>
            <p>
                <label for="Title">Title:</label>
                <%= Html.TextBox("Title") %>
                <%= Html.ValidationMessage("Title", "*") %>
            </p>
            <p>
                <label for="EventDate">EventDate:</label>
                <%=Html.TextBox("EventDate") %>
                <%=Html.ValidationMessage("EventDate", "*") %>
            </p>
            <p>
                <label for="Description">Description:</label>
                <%=Html.TextArea("Description") %>
                <%=Html.ValidationMessage("Description", "*") %>
            </p>
            <p>
                <label for="Address">Address:</label>
                <%=Html.TextBox("Address") %>
                <%=Html.ValidationMessage("Address", "*") %>
            </p>
            <p>
                <label for="Country">Country:</label>
                <%=Html.TextBox("Country") %>
                <%=Html.ValidationMessage("Country", "*") %>
            </p>
            <p>
                <label for="ContactPhone">ContactPhone:</label>
                <%=Html.TextBox("ContactPhone") %>
                <%=Html.ValidationMessage("ContactPhone", "*") %>
            </p>
            <p>
                <label for="Latitude">Latitude:</label>
                <%=Html.TextBox("Latitude") %>
                <%=Html.ValidationMessage("Latitude", "*") %>
            </p>
            <p>
                <label for="Longitude">Longitude:</label>
                <%=Html.TextBox("Longitude") %>
                <%=Html.ValidationMessage("Longitude", "*") %>
            </p>
            <p>
                <input type="submit" value="Save"/>
            </p>
        </fieldset>
    <% }
%>
</asp:Content>
```

And now when we run our application and access the *"/Dinners/Create"* URL within the browser it will render UI like below from our Create action implementation:



Implementing the HTTP-POST Create Action Method

We have the HTTP-GET version of our Create action method implemented. When a user clicks the "Save" button it performs a form post to the */Dinners/Create* URL, and submits the HTML <input> form values using the HTTP POST verb.

Let's now implement the HTTP POST behavior of our create action method. We'll begin by adding an overloaded "Create" action method to our DinnersController that has an "AcceptVerbs" attribute on it that indicates it handles HTTP POST scenarios:

```
//
// POST: /Dinners/Create

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create() {
    ...
}
```

There are a variety of ways we can access the posted form parameters within our HTTP-POST enabled "Create" method.

One approach is to create a new Dinner object and then use the *UpdateModel()* helper method (like we did with the Edit action) to populate it with the posted form values. We can then add it to our DinnerRepository, persist it to the database, and redirect the user to our Details action to show the newly created Dinner using the code below:

```
//
// POST: /Dinners/Create

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create() {

    Dinner dinner = new Dinner();

    try {

        UpdateModel(dinner);

        dinnerRepository.Add(dinner);
        dinnerRepository.Save();

        return RedirectToAction("Details", new {id=dinner.DinnerID});
    }
    catch {

        ModelState.AddRuleViolations(dinner.GetRuleViolations());

        return View(dinner);
    }
}
```

Alternatively, we can use an approach where we have our Create() action method take a Dinner object as a method parameter. ASP.NET MVC will then automatically instantiate a new Dinner object for us, populate its properties using the form inputs, and pass it to our action method:

```
//
//
// POST: /Dinners/Create

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(Dinner dinner) {

    if (ModelState.IsValid) {

        try {
            dinner.HostedBy = "SomeUser";

            dinnerRepository.Add(dinner);
            dinnerRepository.Save();

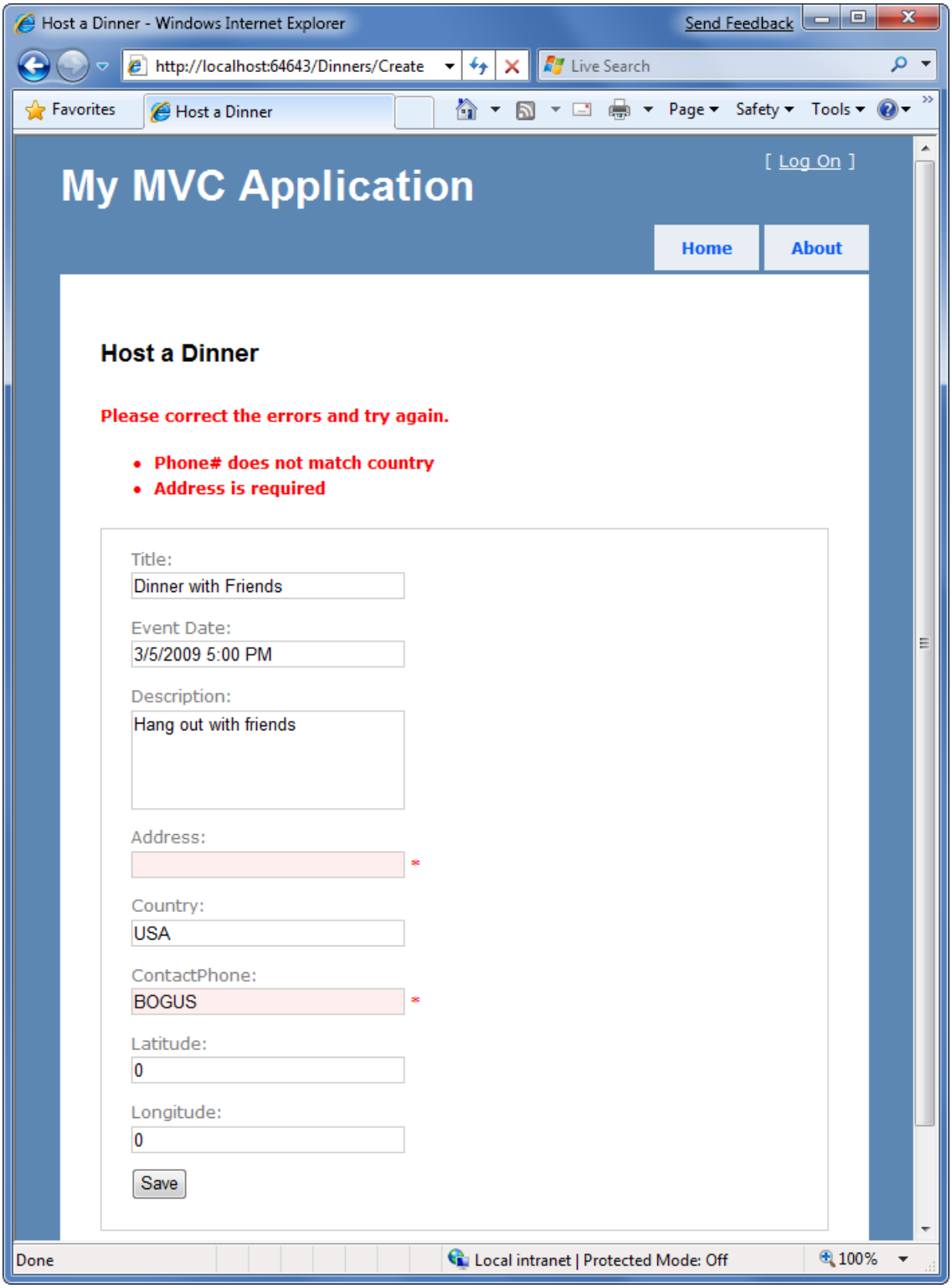
            return RedirectToAction("Details", new {id = dinner.DinnerID });
        }
        catch {
            ModelState.AddRuleViolations(dinner.GetRuleViolations());
        }
    }

    return View(dinner);
}
```

Our action method above verifies that the Dinner object has been successfully populated with the form post values by checking the ModelState.IsValid property. This will return false if there are input conversion issues (for example: a string of "BOGUS" for the EventDate property), and if there are any issues our action method redisplays the form.

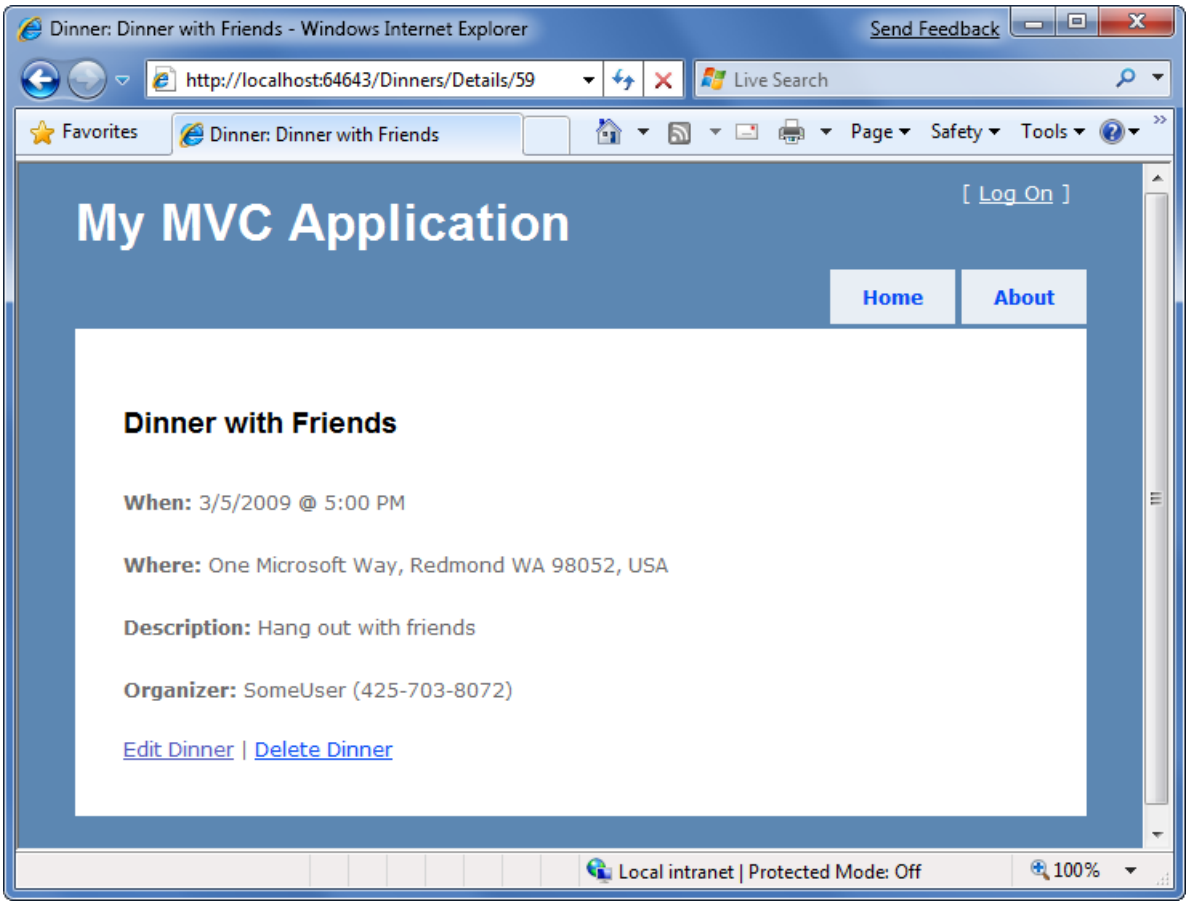
If the input values are valid, then the action method attempts to add and save the new Dinner to the DinnerRepository. It wraps this work within a try/catch block and redisplays the form if there are any business rule violations (which would cause the dinnerRepository.Save() method to raise an exception).

To see this error handling behavior in action, we can request the */Dinners/Create* URL and fill out details about a new Dinner. Incorrect input or values will cause the create form to be redisplayed with the errors highlighted like below:



Notice how our Create form is honoring the exact same validation and business rules as our Edit form. This is because our validation and business rules were defined in the model, and were not embedded within the UI or controller of the application. This means we can later change/evolve our validation or business rules in a single place and have them apply throughout our application. We will not have to change any code within either our Edit or Create action methods to automatically honor any new rules or modifications to existing ones.

When we fix the input values and click the "Save" button again, our addition to the DinnerRepository will succeed, and a new Dinner will be added to the database. We will then be redirected to the */Dinners/Details/[id]* URL – where we will be presented with details about the newly created Dinner:



Delete Support

Let's now add "Delete" support to our DinnersController.

The HTTP-GET Delete Action Method

We'll begin by implementing the HTTP GET behavior of our delete action method. This method will get called when someone visits the */Dinners/Delete/[id]* URL. Below is the implementation:

```
//
// HTTP GET: /Dinners/Delete/1

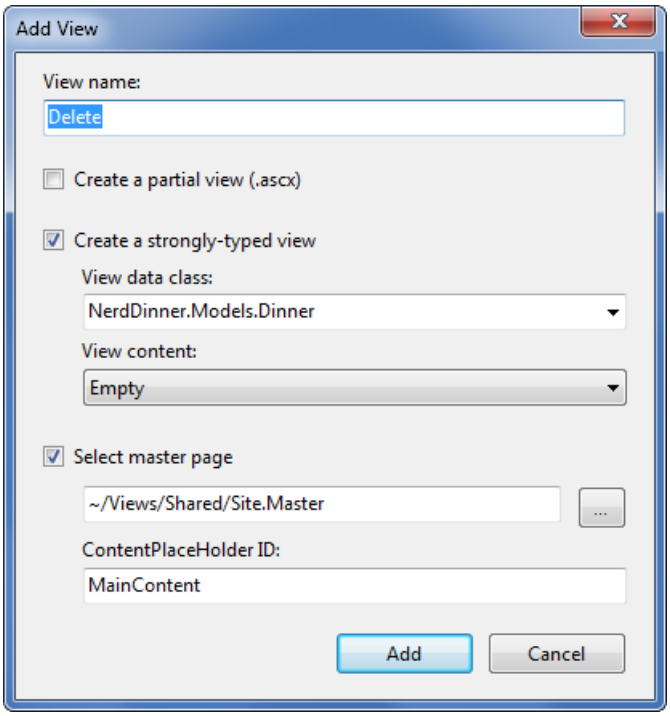
public ActionResult Delete(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (dinner == null)
         return View("NotFound");
    else
        return View(dinner);
}
```

The action method attempts to retrieve the Dinner to be deleted. If the Dinner exists it renders a View based on the Dinner object. If the object doesn't exist (or has already been deleted) it returns a View that renders the "NotFound" view template we created earlier for our "Details" action method.

We can create the "Delete" view template by right-clicking within the Delete action method and selecting the "Add View" context menu command. Within the "Add View" dialog we'll indicate that we are passing a Dinner object to our view template as its model, and choose to create an empty template:



When we click the "Add" button, Visual Studio will add a new "Delete.aspx" view template file for us within our "\Views\Dinners" directory. We'll add some HTML and code to the template to implement a delete confirmation screen like below:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    Delete Confirmation:  <%=Html.Encode(Model.Title) %>
</asp:Content>

<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

    <h2>
        Delete Confirmation
    </h2>

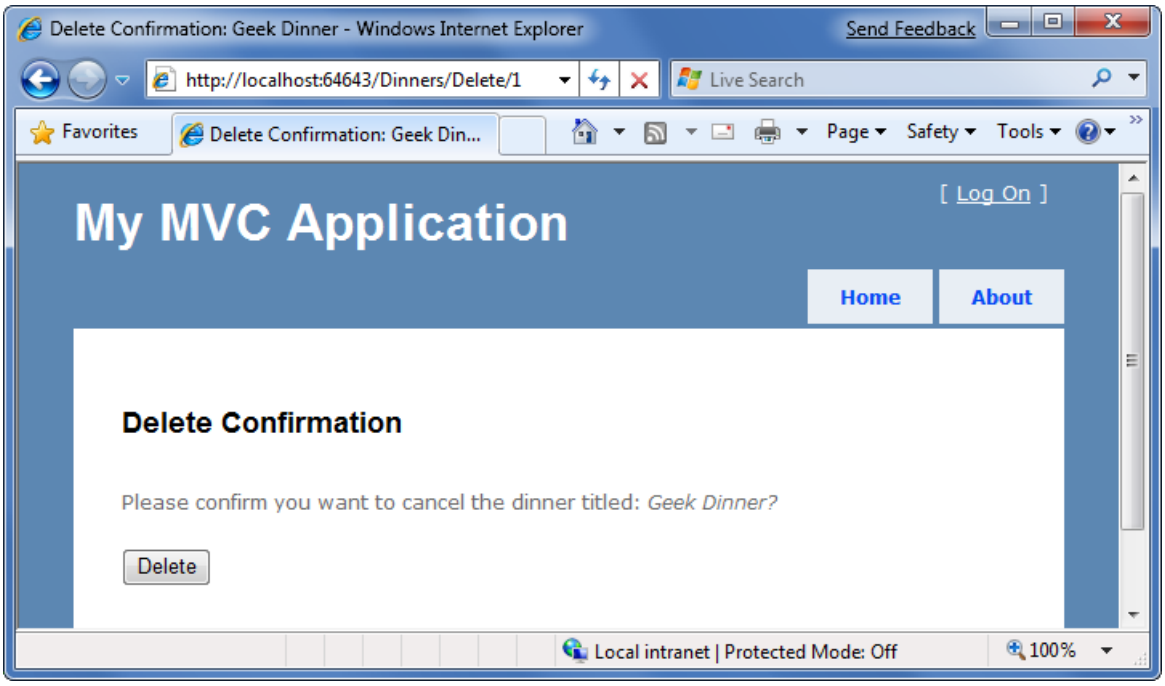
    <div>
        <p>Please confirm you want to cancel the dinner titled:
           <i> <%=Html.Encode(Model.Title) %>? </i>
        </p>
    </div>

    <% using (Html.BeginForm()) {  %>
        <input name="confirmButton" type="submit" value="Delete" />
    <% } %>

</asp:Content>
```

The code above displays the title of the Dinner to be deleted, and outputs a <form> element that does a POST to the /Dinners/Delete/[id] URL if the end-user clicks the "Delete" button within it.

When we run our application and access the *"/Dinners/Delete/[id]"* URL for a valid Dinner object it renders UI like below:



SIDE TOPIC: WHY ARE WE DOING A POST?

You might ask – why did we go through the effort of creating a <form> within our Delete confirmation screen? Why not just use a standard hyperlink to link to an action method that does the actual delete operation? The reason is because we want to be careful to guard against web-crawlers and search engines discovering our URLs and inadvertently causing data to be deleted when they follow the links. HTTP-GET based URLs are considered "safe" for them to access/crawl, and they are supposed to not follow HTTP-POST ones. A good rule is to make sure you always put destructive or data modifying operations behind HTTP-POST requests.

Implementing the HTTP-POST Delete Action Method

We now have the HTTP-GET version of our Delete action method implemented which displays a delete confirmation screen. When an end user clicks the "Delete" button it will perform a form post to the */Dinners/Dinner/[id]* URL.

Let's now implement the HTTP "POST" behavior of the delete action method using the code below:

```
//
// HTTP POST: /Dinners/Delete/1

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Delete(int id, string confirmButton) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (dinner == null)
        return View("NotFound");

    dinnerRepository.Delete(dinner);
    dinnerRepository.Save();

    return View("Deleted");
}
```

The HTTP-POST version of our Delete action method attempts to retrieve the dinner object to delete. If it can't find it (because it has already been deleted) it renders our "NotFound" template. If it finds the Dinner, it deletes it from the DinnerRepository. It then renders a "Deleted" template.

To implement the "Deleted" template we'll right-click in the action method and choose the "Add View" context menu. We'll name our view "Deleted" and have it be an empty template (and not take a strongly-typed model

object). We'll then add some HTML content to it:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    Dinner Deleted
</asp:Content>

<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

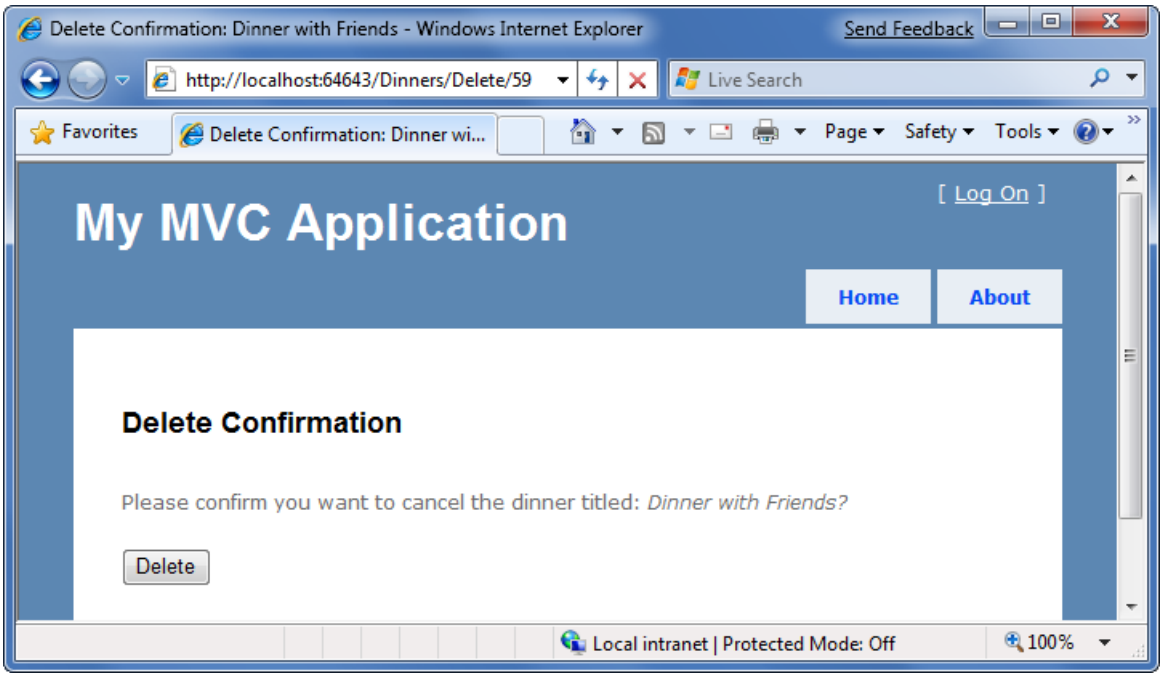
    <h2>Dinner Deleted</h2>

    <div>
        <p>Your dinner was successfully deleted.</p>
    </div>

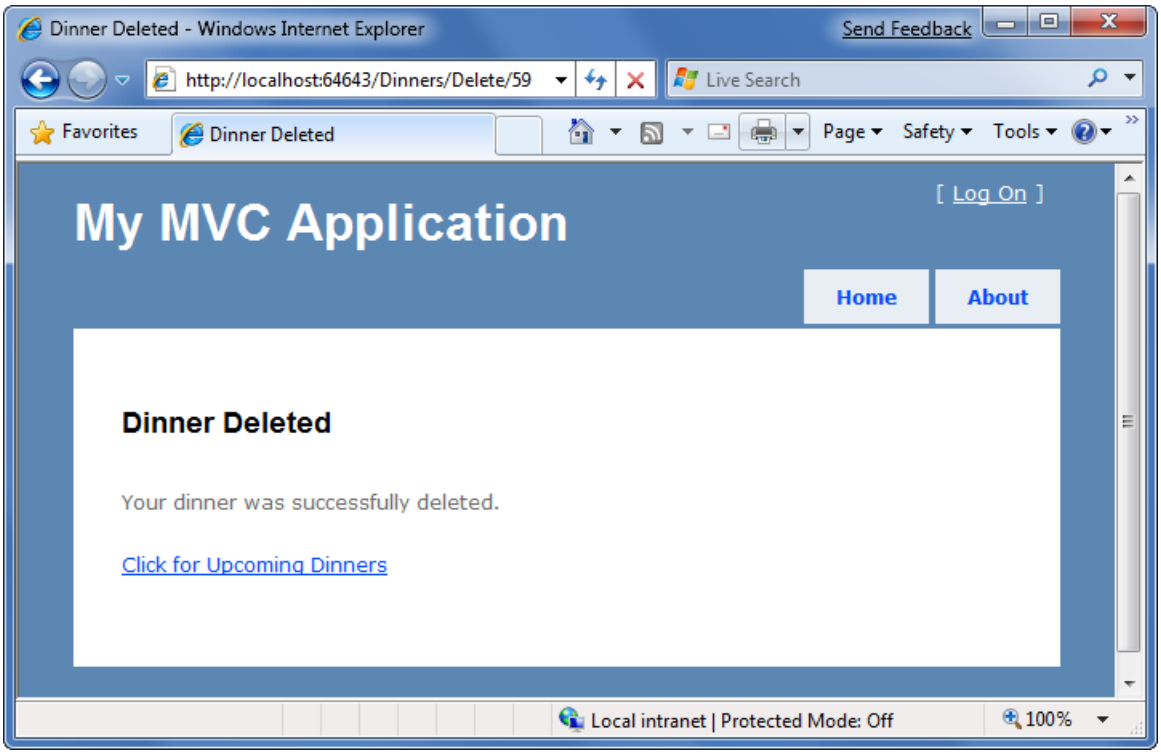
    <div>
        <p><a href="/dinners">Click for Upcoming Dinners</a></p>
    </div>

</asp:Content>
```

And now when we run our application and access the *"/Dinners/Delete/[id]"* URL for a valid Dinner object it will render our Dinner delete confirmation screen like below:



When we click the "Delete" button it will perform an HTTP-POST to the */Dinners/Delete/[id]* URL, which will delete the Dinner from our database, and display our "Deleted" view template:



Model Binding Security

We've discussed two different ways to use the built-in model-binding features of ASP.NET MVC. The first using the UpdateModel() method to update properties on an existing model object, and the second using ASP.NET MVC's support for passing model objects in as action method parameters. Both of these techniques are very powerful and extremely useful.

This power also brings with it responsibility. It is important to always be paranoid about security when accepting any user input, and this is also true when binding objects to form input. You should be careful to always HTML encode any user-entered values to avoid HTML and JavaScript injection attacks, and be careful of SQL injection attacks (note: we are using LINQ to SQL for our application, which automatically encodes parameters to prevent these types of attacks). You should never rely on client-side validation alone, and always employ server-side validation to guard against hackers attempting to send you bogus values.

One additional security item to make sure you think about when using the binding features of ASP.NET MVC is the scope of the objects you are binding. Specifically, you want to make sure you understand the security implications of the properties you are allowing to be bound, and make sure you only allow those properties that really should be updatable by an end-user to be updated.

By default, the UpdateModel() method will attempt to update all properties on the model object that match incoming form parameter values. Likewise, objects passed as action method parameters also by default can have all of their properties set via form parameters.

Locking down binding on a per-usage basis

You can lock down the binding policy on a per-usage basis by providing an explicit "include list" of properties that can be updated. This can be done by passing an extra string array parameter to the UpdateModel() method like below:

```
string[] allowedProperties = new[]{ "Title","Description",
                                    "ContactPhone", "Address",
                                    "EventDate", "Latitude",
                                    "Longitude"};

UpdateModel(dinner, allowedProperties);
```

Objects passed as action method parameters also support a [Bind] attribute that enables an "include list" of

allowed properties to be specified like below:

```
//
// POST: /Dinners/Create

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create( [Bind(Include="Title,Address")] Dinner dinner ) {
    ...
}
```

Locking down binding on a type basis

You can also lock down the binding rules on a per-type basis. This allows you to specify the binding rules once, and then have them apply in all scenarios (including both UpdateModel and action method parameter scenarios) across all controllers and action methods.

You can customize the per-type binding rules by adding a [Bind] attribute onto a type, or by registering it within the Global.asax file of the application (useful for scenarios where you don't own the type). You can then use the Bind attribute's Include and Exclude properties to control which properties are bindable for the particular class or interface.

We'll use this technique for the Dinner class in our NerdDinner application, and add a [Bind] attribute to it that restricts the list of bindable properties to the following:

```
[Bind(Include="Title,Description,EventDate,Address,Country,ContactPhone,Latitude,Longitude")]
public partial class Dinner {
   ...
}
```

Notice we are not allowing the RSVPs collection to be manipulated via binding, nor are we allowing the DinnerID or HostedBy properties to be set via binding. For security reasons we'll instead only manipulate these particular properties using explicit code within our action methods.

CRUD Wrap-Up

ASP.NET MVC includes a number of built-in features that help with implementing form posting scenarios. We used a variety of these features to provide CRUD UI support on top of our DinnerRepository.

We are using a model-focused approach to implement our application. This means that all our validation and business rule logic is defined within our model layer – and not within our controllers or views. Neither our Controller class nor our View templates know anything about the specific business rules being enforced by our Dinner model class.

This will keep our application architecture clean and make it easier to test. We can add additional business rules to our model layer in the future and *not have to make any code changes* to our Controller or View in order for them to be supported. This is going to provide us with a great deal of agility to evolve and change our application in the future.

Our DinnersController now enables Dinner listings/details, as well as create, edit and delete support. The complete code for the class can be found below:

```
public class DinnersController : Controller {

    DinnerRepository dinnerRepository = new DinnerRepository();

    //
    // GET: /Dinners/

    public ActionResult Index() {
```

```
        var dinners = dinnerRepository.FindUpcomingDinners().ToList();
        return View(dinners);
    }

    //
    // GET: /Dinners/Details/2

    public ActionResult Details(int id) {

        Dinner dinner = dinnerRepository.GetDinner(id);

        if (dinner == null)
            return View("NotFound");
        else
            return View(dinner);
    }

    //
    // GET: /Dinners/Edit/2

    public ActionResult Edit(int id) {

        Dinner dinner = dinnerRepository.GetDinner(id);
        return View(dinner);
    }

    //
    // POST: /Dinners/Edit/2

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Edit(int id, FormCollection formValues) {

        Dinner dinner = dinnerRepository.GetDinner(id);

        try {
            UpdateModel(dinner);

            dinnerRepository.Save();

            return RedirectToAction("Details", new { id= dinner.DinnerID });
        }
        catch {
            ModelState.AddRuleViolations(dinner.GetRuleViolations());

            return View(dinner);
        }
    }

    //
    // GET: /Dinners/Create

    public ActionResult Create() {

        Dinner dinner = new Dinner() {
            EventDate = DateTime.Now.AddDays(7)
        };
        return View(dinner);
    }

    //
    // POST: /Dinners/Create

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Create(Dinner dinner) {

        if (ModelState.IsValid) {

            try {
                dinner.HostedBy = "SomeUser";
```

```
                dinnerRepository.Add(dinner);
                dinnerRepository.Save();

                return RedirectToAction("Details", new{id=dinner.DinnerID});
            }
            catch {
                ModelState.AddRuleViolations(dinner.GetRuleViolations());
            }
        }

        return View(dinner);
    }

    //
    // HTTP GET: /Dinners/Delete/1

    public ActionResult Delete(int id) {

        Dinner dinner = dinnerRepository.GetDinner(id);

        if (dinner == null)
            return View("NotFound");
        else
            return View(dinner);
    }

    //
    // HTTP POST: /Dinners/Delete/1

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Delete(int id, string confirmButton) {

        Dinner dinner = dinnerRepository.GetDinner(id);

        if (dinner == null)
            return View("NotFound");

        dinnerRepository.Delete(dinner);
        dinnerRepository.Save();

        return View("Deleted");
    }
}
```

Next Step

We now have basic CRUD (Create, Read, Update and Delete) support implement within our DinnersController class.

Let's now look at how we can use ViewData and ViewModel classes to enable even richer UI on our forms.

PREVIOUS NEXT

Use ViewData and Implement ViewModel Classes

4/10/2020 • 7 minutes to read • Edit Online

by Microsoft

Download PDF

> This is step 6 of a free "NerdDinner" application tutorial that walks-through how to build a small, but complete, web application using ASP.NET MVC 1.
>
> Step 6 shows how enable support for richer form editing scenarios, and also discusses two approaches that can be used to pass data from controllers to views: ViewData and ViewModel.
>
> If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

NerdDinner Step 6: ViewData and ViewModel

We've covered a number of form post scenarios, and discussed how to implement create, update and delete (CRUD) support. We'll now take our DinnersController implementation further and enable support for richer form editing scenarios. While doing this we'll discuss two approaches that can be used to pass data from controllers to views: ViewData and ViewModel.

Passing Data from Controllers to View-Templates

One of the defining characteristics of the MVC pattern is the strict "separation of concerns" it helps enforce between the different components of an application. Models, Controllers and Views each have well defined roles and responsibilities, and they communicate amongst each other in well defined ways. This helps promote testability and code reuse.

When a Controller class decides to render an HTML response back to a client, it is responsible for explicitly passing to the view template all of the data needed to render the response. View templates should never perform any data retrieval or application logic – and should instead limit themselves to only have rendering code that is driven off of the model/data passed to it by the controller.

Right now the model data being passed by our DinnersController class to our view templates is simple and straight-forward – a list of Dinner objects in the case of Index(), and a single Dinner object in the case of Details(), Edit(), Create() and Delete(). As we add more UI capabilities to our application, we are often going to need to pass more than just this data to render HTML responses within our view templates. For example, we might want to change the "Country" field within our Edit and Create views from being an HTML textbox to a dropdownlist. Rather than hard-code the dropdown list of country names in the view template, we might want to generate it from a list of supported countries that we populate dynamically. We will need a way to pass both the Dinner object *and* the list of supported countries from our controller to our view templates.

Let's look at two ways we can accomplish this.

Using the ViewData Dictionary

The Controller base class exposes a "ViewData" dictionary property that can be used to pass additional data items from Controllers to Views.

For example, to support the scenario where we want to change the "Country" textbox within our Edit view from being an HTML textbox to a dropdownlist, we can update our Edit() action method to pass (in addition to a Dinner object) a SelectList object that can be used as the model of a countries dropdownlist.

```
//
// GET: /Dinners/Edit/5

[Authorize]
public ActionResult Edit(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    ViewData["Countries"] = new SelectList(PhoneValidator.AllCountries, dinner.Country);

    return View(dinner);
}
```

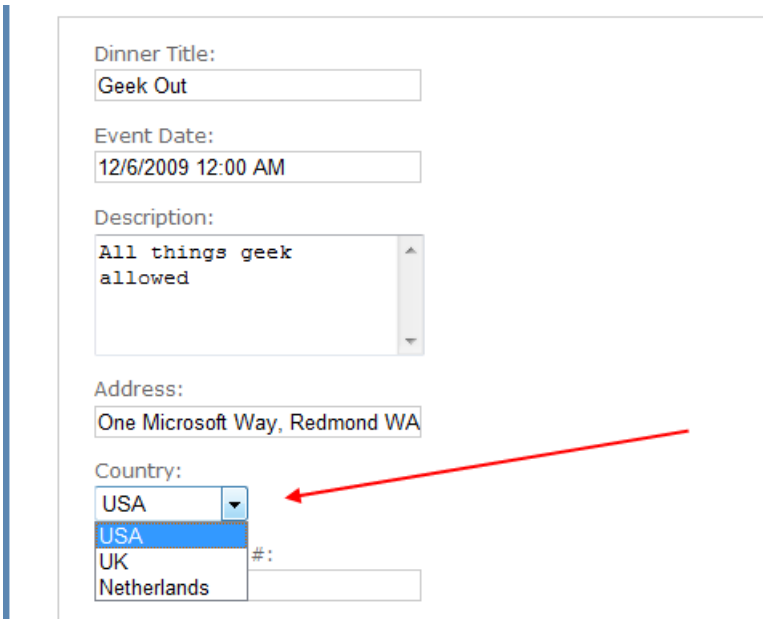
The constructor of the SelectList above is accepting a list of counties to populate the drop-downlist with, as well as the currently selected value.

We can then update our Edit.aspx view template to use the Html.DropDownList() helper method instead of the Html.TextBox() helper method we used previously:

```
<%= Html.DropDownList("Country", ViewData["Countries"] as SelectList) %>
```

The Html.DropDownList() helper method above takes two parameters. The first is the name of the HTML form element to output. The second is the "SelectList" model we passed via the ViewData dictionary. We are using the C# "as" keyword to cast the type within the dictionary as a SelectList.

And now when we run our application and access the */Dinners/Edit/1* URL within our browser we'll see that our edit UI has been updated to display a dropdownlist of countries instead of a textbox:



Because we also render the Edit view template from the HTTP-POST Edit method (in scenarios when errors occur), we'll want to make sure that we also update this method to add the SelectList to ViewData when the view template is rendered in error scenarios:

```
//
// POST: /Dinners/Edit/5

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection collection) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    try {

        UpdateModel(dinner);

        dinnerRepository.Save();

        return RedirectToAction("Details", new { id=dinner.DinnerID });
    }
    catch {

        ModelState.AddModelErrors(dinner.GetRuleViolations());

        ViewData["countries"] = new SelectList(PhoneValidator.AllCountries, dinner.Country);

        return View(dinner);
    }
}
```

And now our DinnersController edit scenario supports a DropDownList.

Using a ViewModel Pattern

The ViewData dictionary approach has the benefit of being fairly fast and easy to implement. Some developers don't like using string-based dictionaries, though, since typos can lead to errors that will not be caught at compile-time. The un-typed ViewData dictionary also requires using the "as" operator or casting when using a strongly-typed language like C# in a view template.

An alternative approach that we could use is one often referred to as the "ViewModel" pattern. When using this pattern we create strongly-typed classes that are optimized for our specific view scenarios, and which expose properties for the dynamic values/content needed by our view templates. Our controller classes can then populate and pass these view-optimized classes to our view template to use. This enables type-safety, compile-time checking, and editor intellisense within view templates.

For example, to enable dinner form editing scenarios we can create a "DinnerFormViewModel" class like below that exposes two strongly-typed properties: a Dinner object, and the SelectList model needed to populate the countries dropdownlist:

```
public class DinnerFormViewModel {

    // Properties
    public Dinner     Dinner    { get; private set; }
    public SelectList Countries { get; private set; }

    // Constructor
    public DinnerFormViewModel(Dinner dinner) {
        Dinner = dinner;
        Countries = new SelectList(PhoneValidator.AllCountries, dinner.Country);
    }
}
```

We can then update our Edit() action method to create the DinnerFormViewModel using the Dinner object we retrieve from our repository, and then pass it to our view template:

```
//
// GET: /Dinners/Edit/5

[Authorize]
public ActionResult Edit(int id) {

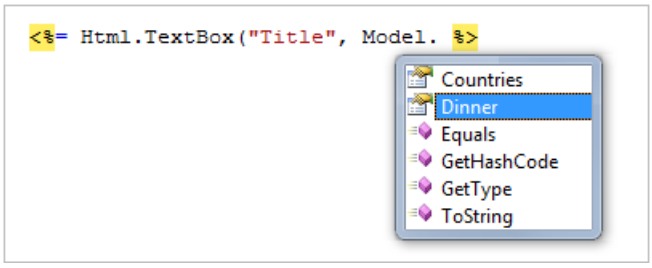
    Dinner dinner = dinnerRepository.GetDinner(id);

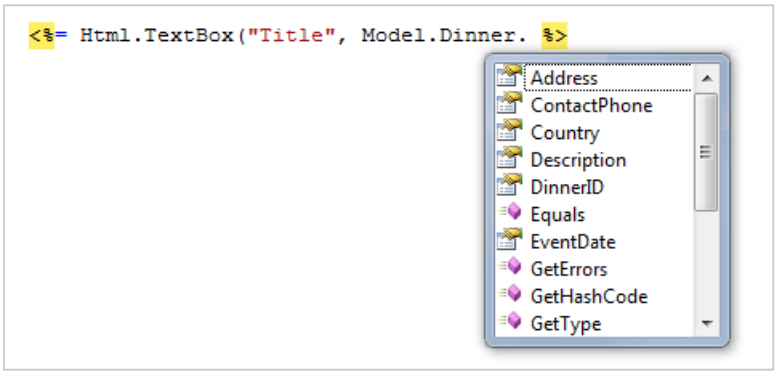
    return View(new DinnerFormViewModel(dinner));
}
```

We'll then update our view template so that it expects a "DinnerFormViewModel" instead of a "Dinner" object by changing the "inherits" attribute at the top of the edit.aspx page like so:

```
Inherits="System.Web.Mvc.ViewPage<NerdDinner.Controllers.DinnerFormViewModel>
```

Once we do this, the intellisense of the "Model" property within our view template will be updated to reflect the object model of the DinnerFormViewModel type we are passing it:





We can then update our view code to work off of it. Notice below how we are not changing the names of the input elements we are creating (the form elements will still be named "Title", "Country") – but we are updating the HTML Helper methods to retrieve the values using the DinnerFormViewModel class:

```
<p>
    <label for="Title">Dinner Title:</label>
    <%= Html.TextBox("Title", Model.Dinner.Title) %>
    <%=Html.ValidationMessage("Title", "*") %>
</p>

<p>
    <label for="Country">Country:</label>
    <%= Html.DropDownList("Country", Model.Countries) %>
    <%=Html.ValidationMessage("Country", "*") %>
</p>
```

We'll also update our Edit post method to use the DinnerFormViewModel class when rendering errors:

```
//
// POST: /Dinners/Edit/5

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, FormCollection collection) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    try {
        UpdateModel(dinner);

        dinnerRepository.Save();

        return RedirectToAction("Details", new { id=dinner.DinnerID });
    }
    catch {
        ModelState.AddModelErrors(dinner.GetRuleViolations());

        return View(new DinnerFormViewModel(dinner));
    }
}
```

We can also update our Create() action methods to re-use the exact same *DinnerFormViewModel* class to enable the countries DropDownList within those as well. Below is the HTTP-GET implementation:

```
//
// GET: /Dinners/Create

public ActionResult Create() {

    Dinner dinner = new Dinner() {
        EventDate = DateTime.Now.AddDays(7)
    };

    return View(new DinnerFormViewModel(dinner));
}
```

Below is the implementation of the HTTP-POST Create method:

```
//
// POST: /Dinners/Create

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(Dinner dinner) {

    if (ModelState.IsValid) {

        try {
            dinner.HostedBy = "SomeUser";

            dinnerRepository.Add(dinner);
            dinnerRepository.Save();

            return RedirectToAction("Details", new { id=dinner.DinnerID });
        }
        catch {
            ModelState.AddModelErrors(dinner.GetRuleViolations());
        }
    }

    return View(new DinnerFormViewModel(dinner));
}
```

And now both our Edit and Create screens support drop-downlists for picking the country.

Custom-shaped ViewModel classes

In the scenario above, our DinnerFormViewModel class directly exposes the Dinner model object as a property, along with a supporting SelectList model property. This approach works fine for scenarios where the HTML UI we want to create within our view template corresponds relatively closely to our domain model objects.

For scenarios where this isn't the case, one option that you can use is to create a custom-shaped ViewModel class whose object model is more optimized for consumption by the view – and which might look completely different from the underlying domain model object. For example, it could potentially expose different property names and/or aggregate properties collected from multiple model objects.

Custom-shaped ViewModel classes can be used both to pass data from controllers to views to render, as well as to help handle form data posted back to a controller's action method. For this later scenario, you might have the action method update a ViewModel object with the form-posted data, and then use the ViewModel instance to map or retrieve an actual domain model object.

Custom-shaped ViewModel classes can provide a great deal of flexibility, and are something to investigate any time you find the rendering code within your view templates or the form-posting code inside your action methods starting to get too complicated. This is often a sign that your domain models don't cleanly correspond to the UI you are generating, and that an intermediate custom-shaped ViewModel class can help.

Next Step

Let's now look at how we can use partials and master-pages to re-use and share UI across our application.

PREVIOUS NEXT

Re-use UI Using Master Pages and Partials

4/10/2020 • 8 minutes to read • Edit Online

by Microsoft

Download PDF

> This is step 7 of a free "NerdDinner" application tutorial that walks-through how to build a small, but complete, web application using ASP.NET MVC 1.
>
> Step 7 looks at ways we can apply the "DRY Principle" within our view templates to eliminate code duplication, using partial view templates and master pages.
>
> If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

NerdDinner Step 7: Partials and Master Pages

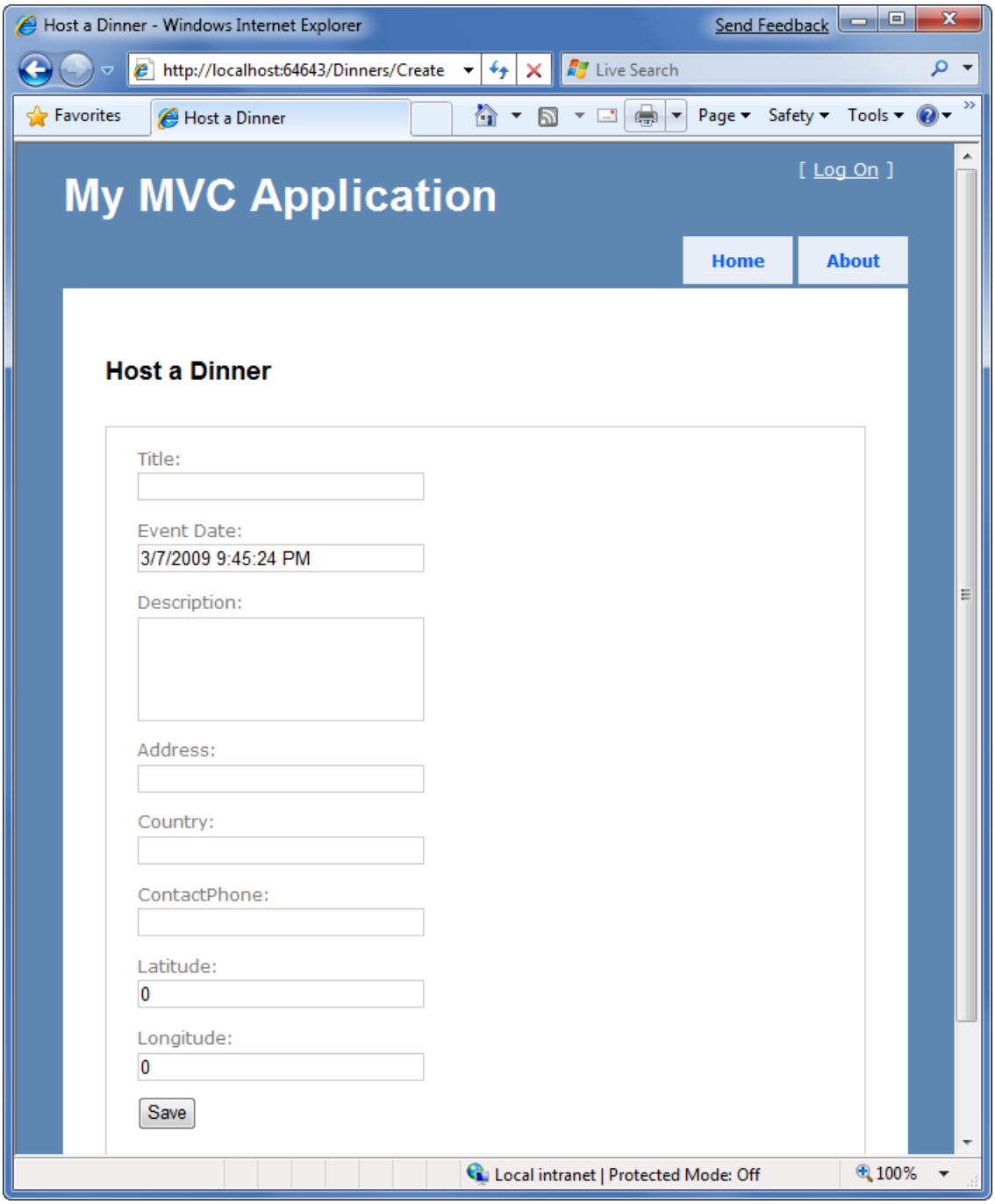
One of the design philosophies ASP.NET MVC embraces is the "Do Not Repeat Yourself" principle (commonly referred to as "DRY"). A DRY design helps eliminate the duplication of code and logic, which ultimately makes applications faster to build and easier to maintain.

We've already seen the DRY principle applied in several of our NerdDinner scenarios. A few examples: our validation logic is implemented within our model layer, which enables it to be enforced across both edit and create scenarios in our controller; we are re-using the "NotFound" view template across the Edit, Details and Delete action methods; we are using a convention- naming pattern with our view templates, which eliminates the need to explicitly specify the name when we call the View() helper method; and we are re-using the DinnerFormViewModel class for both Edit and Create action scenarios.

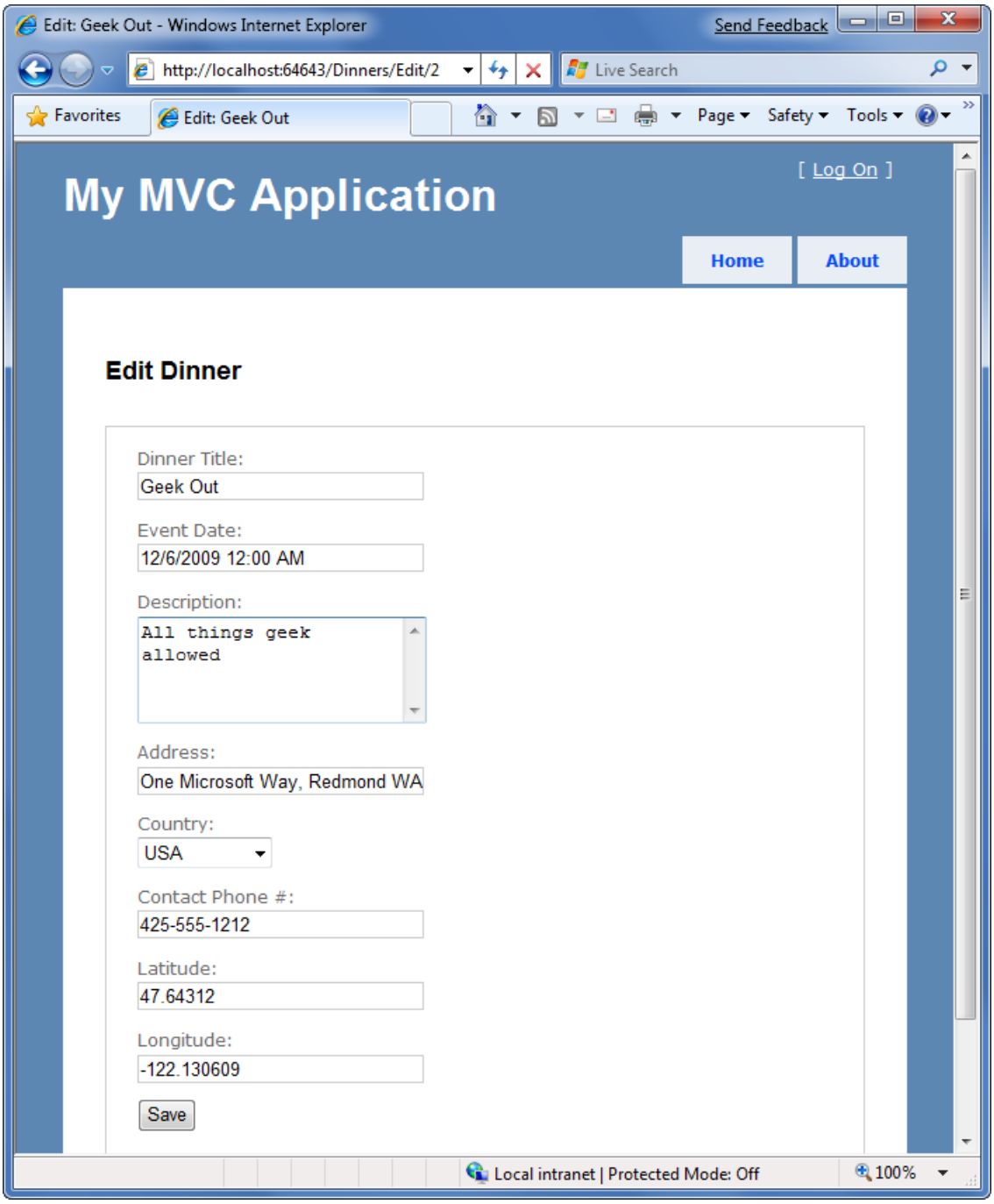
Let's now look at ways we can apply the "DRY Principle" within our view templates to eliminate code duplication there as well.

Re-visiting our Edit and Create View Templates

Currently we are using two different view templates – "Edit.aspx" and "Create.aspx" – to display our Dinner form UI. A quick visual comparison of them highlights how similar they are. Below is what the create form looks like:



And here is what our "Edit" form looks like:



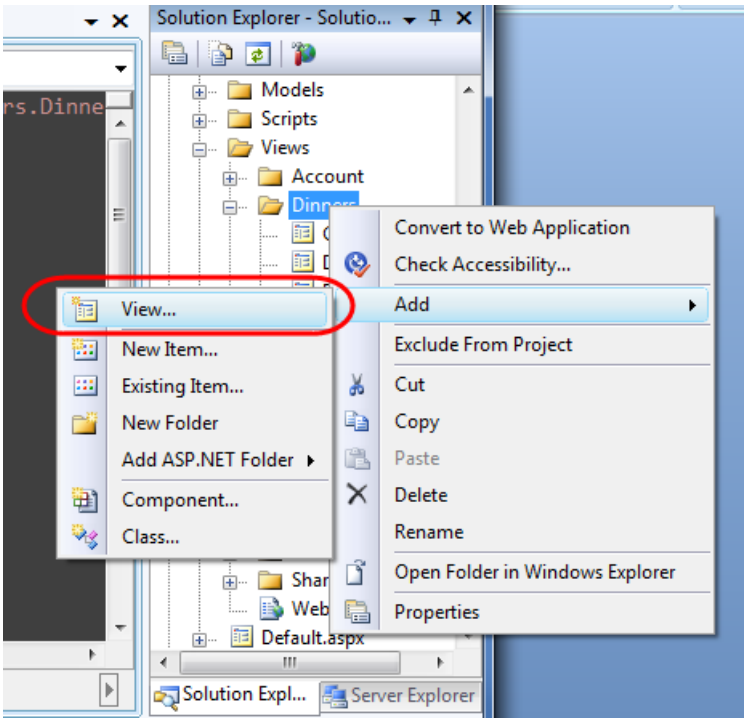
Not much of a difference is there? Other than the title and header text, the form layout and input controls are identical.

If we open up the "Edit.aspx" and "Create.aspx" view templates we'll find that they contain identical form layout and input control code. This duplication means we end up having to make changes twice anytime we introduce or change a new Dinner property - which is not good.

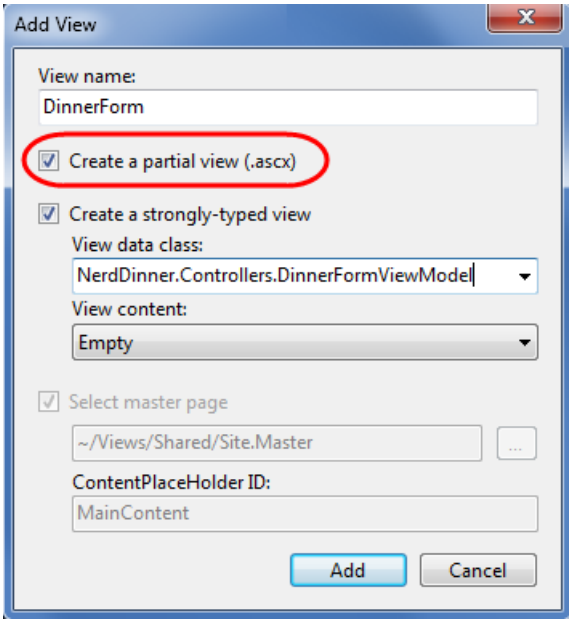
Using Partial View Templates

ASP.NET MVC supports the ability to define "partial view" templates that can be used to encapsulate view rendering logic for a sub-portion of a page. "Partials" provide a useful way to define view rendering logic once, and then re-use it in multiple places across an application.

To help "DRY-up" our Edit.aspx and Create.aspx view template duplication, we can create a partial view template named "DinnerForm.ascx" that encapsulates the form layout and input elements common to both. We'll do this by right-clicking on our /Views/Dinners directory and choosing the "Add->View" menu command:



This will display the "Add View" dialog. We'll name the new view we want to create "DinnerForm", select the "Create a partial view" checkbox within the dialog, and indicate that we will pass it a DinnerFormViewModel class:



When we click the "Add" button, Visual Studio will create a new "DinnerForm.ascx" view template for us within the "\Views\Dinners" directory.

We can then copy/paste the duplicate form layout / input control code from our Edit.aspx/ Create.aspx view templates into our new "DinnerForm.ascx" partial view template:

```
<%= Html.ValidationSummary("Please correct the errors and try again.") %>

<% using (Html.BeginForm()) { %>

    <fieldset>
        <p>
            <label for="Title">Dinner Title:</label>
            <%= Html.TextBox("Title", Model.Dinner.Title) %>
            <%=Html.ValidationMessage("Title", "*") %>
        </p>
        <p>
            <label for="EventDate">Event Date:</label>
            <%= Html.TextBox("EventDate", Model.Dinner.EventDate) %>
            <%= Html.ValidationMessage("EventDate", "*") %>
        </p>
        <p>
            <label for="Description">Description:</label>
            <%= Html.TextArea("Description", Model.Dinner.Description) %>
            <%= Html.ValidationMessage("Description", "*") %>
        </p>
        <p>
            <label for="Address">Address:</label>
            <%= Html.TextBox("Address", Model.Dinner.Address) %>
            <%= Html.ValidationMessage("Address", "*") %>
        </p>
        <p>
            <label for="Country">Country:</label>
            <%= Html.DropDownList("Country", Model.Countries) %>
            <%= Html.ValidationMessage("Country", "*") %>
        </p>
        <p>
            <label for="ContactPhone">Contact Phone #:</label>
            <%= Html.TextBox("ContactPhone", Model.Dinner.ContactPhone) %>
            <%= Html.ValidationMessage("ContactPhone", "*") %>
        </p>

        <p>
            <input type="submit" value="Save"/>
        </p>
    </fieldset>

<% } %>
```

We can then update our Edit and Create view templates to call the DinnerForm partial template and eliminate the form duplication. We can do this by calling Html.RenderPartial("DinnerForm") within our view templates:

Create.aspx

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    Host a Dinner
</asp:Content>

<asp:Content ID="Create" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Host a Dinner</h2>

    <% Html.RenderPartial("DinnerForm"); %>

</asp:Content>
```

Edit.aspx

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    Edit: <%=Html.Encode(Model.Dinner.Title) %>
</asp:Content>

<asp:Content ID="Edit" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Edit Dinner</h2>

    <% Html.RenderPartial("DinnerForm"); %>

</asp:Content>
```

You can explicitly qualify the path of the partial template you want when calling Html.RenderPartial (for example: ~Views/Dinners/DinnerForm.ascx"). In our code above, though, we are taking advantage of the convention-based naming pattern within ASP.NET MVC, and just specifying "DinnerForm" as the name of the partial to render. When we do this ASP.NET MVC will look first in the convention-based views directory (for DinnersController this would be /Views/Dinners). If it doesn't find the partial template there it will then look for it in the /Views/Shared directory.

When Html.RenderPartial() is called with just the name of the partial view, ASP.NET MVC will pass to the partial view the same Model and ViewData dictionary objects used by the calling view template. Alternatively, there are overloaded versions of Html.RenderPartial() that enable you to pass an alternate Model object and/or ViewData dictionary for the partial view to use. This is useful for scenarios where you only want to pass a subset of the full Model/ViewModel.

SIDE TOPIC: WHY <% %> INSTEAD OF <%= %>?

One of the subtle things you might have noticed with the code above is that we are using a <% %> block instead of a <%= %> block when calling Html.RenderPartial(). <%= %> blocks in ASP.NET indicate that a developer wants to render a specified value (for example: <%= "Hello" %> would render "Hello"). <% %> blocks instead indicate that the developer wants to execute code, and that any rendered output within them must be done explicitly (for example: <% Response.Write("Hello") %>. The reason we are using a <% %> block with our Html.RenderPartial code above is because the Html.RenderPartial() method doesn't return a string, and instead outputs the content directly to the calling view template's output stream. It does this for performance efficiency reasons, and by doing so it avoids the need to create a (potentially very large) temporary string object. This reduces memory usage and improves overall application throughput. One common mistake when using Html.RenderPartial() is to forget to add a semi-colon at the end of the call when it is within a <% %> block. For example, this code will cause a compiler error: <% Html.RenderPartial("DinnerForm") %> You instead need to write: <% Html.RenderPartial("DinnerForm"); %> This is because <% %> blocks are self-contained code statements, and when using C# code statements need to be terminated with a semi-colon.

Using Partial View Templates to Clarify Code

We created the "DinnerForm" partial view template to avoid duplicating view rendering logic in multiple places. This is the most common reason to create partial view templates.

Sometimes it still makes sense to create partial views even when they are only being called in a single place. Very complicated view templates can often become much easier to read when their view rendering logic is extracted and partitioned into one or more well named partial templates.

For example, consider the below code-snippet from the Site.master file in our project (which we will be looking at shortly). The code is relatively straight-forward to read – partly because the logic to display a login/logout link at the top right of the screen is encapsulated within the "LogOnUserControl" partial:

```
<div id="header">
    <div id="title">
        <h1>My MVC Application</h1>
    </div>

    <div id="logindisplay">
        <% Html.RenderPartial("LogOnUserControl"); %>
    </div>

    <div id="menucontainer">

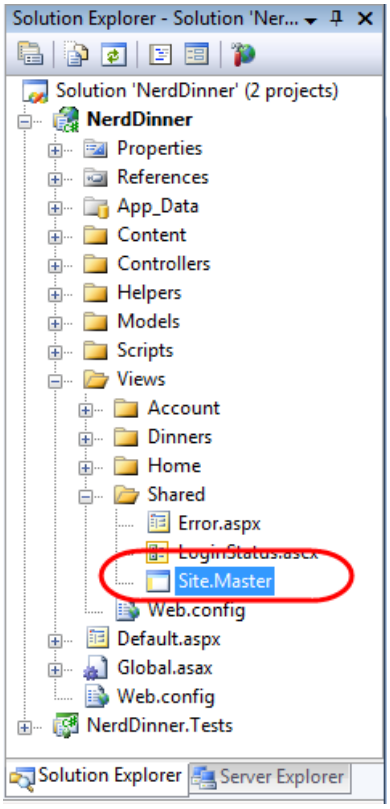
        <ul id="menu">
            <li><%=Html.ActionLink("Home", "Index", "Home")%></li>
            <li><%=Html.ActionLink("About", "About", "Home")%></li>
        </ul>
    </div>
</div>
```

Whenever you find yourself getting confused trying to understand the html/code markup within a view-template, consider whether it wouldn't be clearer if some of it was extracted and refactored into well-named partial views.

Master Pages

In addition to supporting partial views, ASP.NET MVC also supports the ability to create "master page" templates that can be used to define the common layout and top-level html of a site. Content placeholder controls can then be added to the master page to identify replaceable regions that can be overridden or "filled in" by views. This provides a very effective (and DRY) way to apply a common layout across an application.

By default, new ASP.NET MVC projects have a master page template automatically added to them. This master page is named "Site.master" and lives within the \Views\Shared\ folder:



The default Site.master file looks like below. It defines the outer html of the site, along with a menu for navigation at the top. It contains two replaceable content placeholder controls – one for the title, and the other for where the primary content of a page should be replaced:

```
<%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head runat="server">
    <title>
       <asp:ContentPlaceHolder ID="TitleContent" runat="server" />
    </title>
   <link href="../../Content/Site.css" rel="stylesheet" type="text/css" />
</head>

<body>
    <div class="page">

        <div id="header">
            <div id="title">
                <h1>My MVC Application</h1>
            </div>

            <div id="logindisplay">
                <% Html.RenderPartial("LogOnUserControl"); %>
            </div>

            <div id="menucontainer">

                <ul id="menu">
                    <li><%=Html.ActionLink("Home", "Index", "Home")%></li>
                    <li><%=Html.ActionLink("About", "About", "Home")%></li>
                </ul>

            </div>
        </div>

        <div id="main">
            <asp:ContentPlaceHolder ID="MainContent" runat="server" />
        </div>
    </div>
</body>
</html>
```

All of the view templates we've created for our NerdDinner application ("List", "Details", "Edit", "Create", "NotFound", etc) have been based on this Site.master template. This is indicated via the "MasterPageFile" attribute that was added by default to the top <% @ Page %> directive when we created our views using the "Add View" dialog:

```
<%@ Page Inherits="System.Web.Mvc.ViewPage<NerdDinner.Controllers.DinnerViewModel>"
MasterPageFile="~/Views/Shared/Site.Master" %>
```

What this means is that we can change the Site.master content, and have the changes automatically be applied and used when we render any of our view templates.

Let's update our Site.master's header section so that the header of our application is "NerdDinner" instead of "My MVC Application". Let's also update our navigation menu so that the first tab is "Find a Dinner" (handled by the HomeController's Index() action method), and let's add a new tab called "Host a Dinner" (handled by the DinnersController's Create() action method):

```
<div id="header">

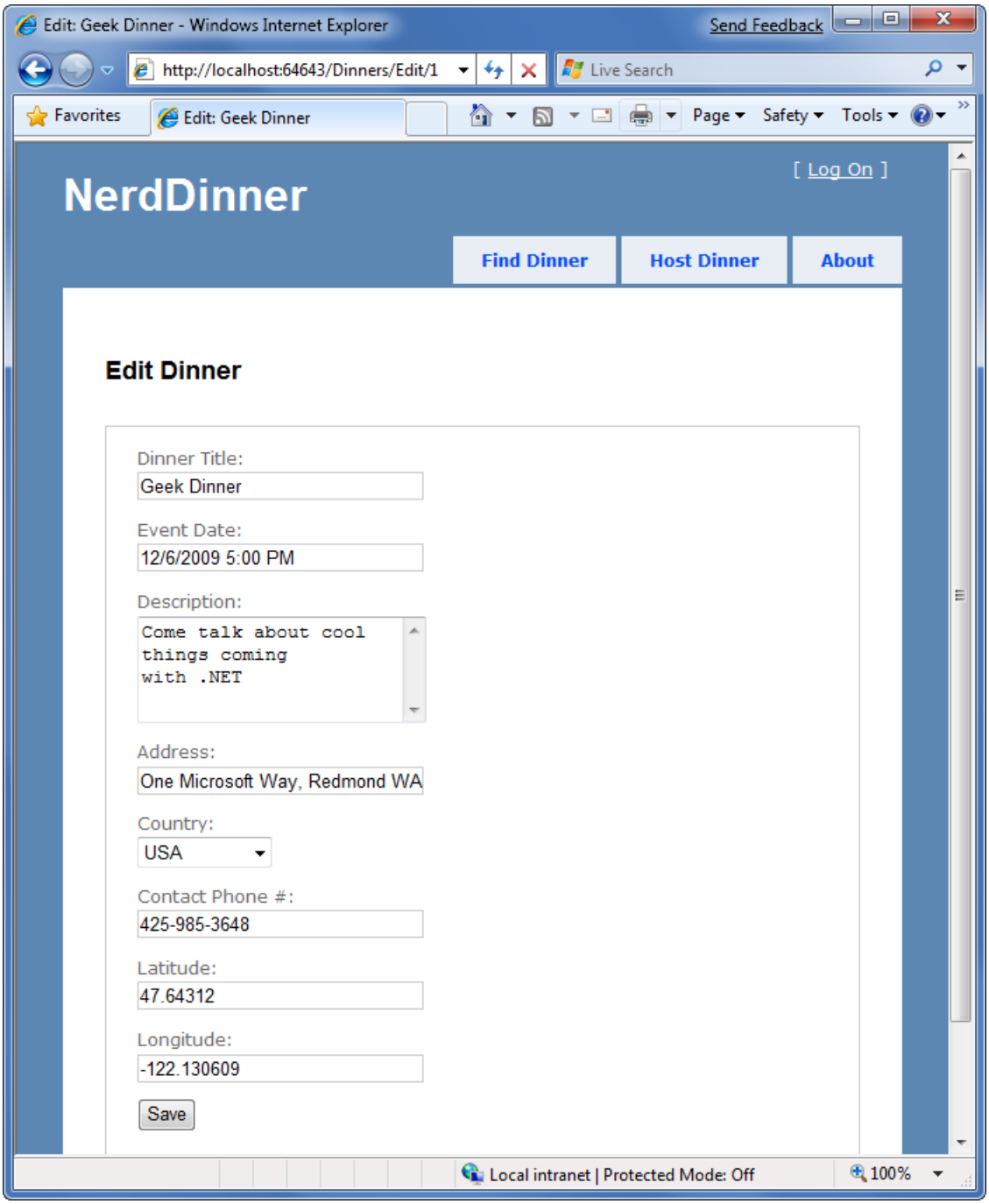
    <div id="title">
        <h1>NerdDinner</h1>
    </div>

    <div id="logindisplay">
        <% Html.RenderPartial("LoginStatus"); %>
    </div>

    <div id="menucontainer">
        <ul id="menu">
           <li><%=Html.ActionLink("Find Dinner", "Index", "Home")%></li>
           <li><%=Html.ActionLink("Host Dinner", "Create", "Dinners")%></li>
           <li><%=Html.ActionLink("About", "About", "Home")%></li>
        </ul>
    </div>
</div>
```

When we save the Site.master file and refresh our browser we'll see our header changes show up across all views within our application. For example:

And with the */Dinners/Edit/[id]* URL:



Next Step

Partials and master pages provide very flexible options that enable you to cleanly organize views. You'll find that they help you avoid duplicating view content/ code, and make your view templates easier to read and maintain.

Let's now revisit the listing scenario we built earlier and enable scalable paging support.

PREVIOUS NEXT

Implement Efficient Data Paging

4/10/2020 • 9 minutes to read • Edit Online

by Microsoft

Download PDF

> This is step 8 of a free "NerdDinner" application tutorial that walks-through how to build a small, but complete, web application using ASP.NET MVC 1.
>
> Step 8 shows how to add paging support to our /Dinners URL so that instead of displaying 1000s of dinners at once, we'll only display 10 upcoming dinners at a time - and allow end-users to page back and forward through the entire list in an SEO friendly way.
>
> If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

NerdDinner Step 8: Paging Support

If our site is successful, it will have thousands of upcoming dinners. We need to make sure that our UI scales to handle all of these dinners, and allows users to browse them. To enable this, we'll add paging support to our */Dinners* URL so that instead of displaying 1000s of dinners at once, we'll only display 10 upcoming dinners at a time - and allow end-users to page back and forward through the entire list in an SEO friendly way.

Index() Action Method Recap

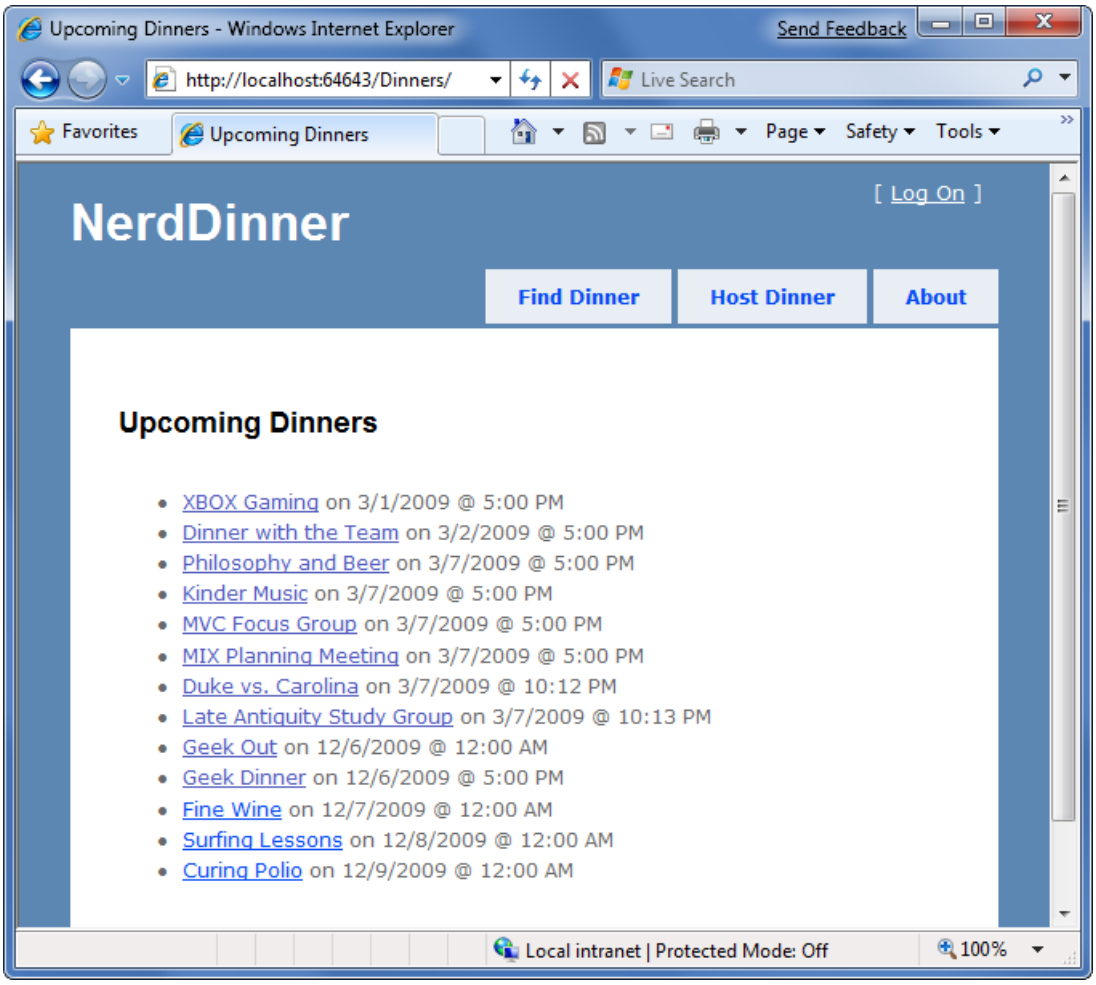
The Index() action method within our DinnersController class currently looks like below:

```
//
// GET: /Dinners/

public ActionResult Index() {

    var dinners = dinnerRepository.FindUpcomingDinners().ToList();
    return View(dinners);
}
```

When a request is made to the */Dinners* URL, it retrieves a list of all upcoming dinners and then renders a listing of all of them out:



Understanding IQueryable<T>

IQueryable<T> is an interface that was introduced with LINQ as part of .NET 3.5. It enables powerful "deferred execution" scenarios that we can take advantage of to implement paging support.

In our DinnerRepository we are returning an IQueryable<Dinner> sequence from our FindUpcomingDinners() method:

```
public class DinnerRepository {

    private NerdDinnerDataContext db = new NerdDinnerDataContext();

    //
    // Query Methods

    public IQueryable<Dinner> FindUpcomingDinners() {

        return from dinner in db.Dinners
               where dinner.EventDate > DateTime.Now
               orderby dinner.EventDate
               select dinner;
    }
```

The IQueryable<Dinner> object returned by our FindUpcomingDinners() method encapsulates a query to retrieve Dinner objects from our database using LINQ to SQL. Importantly, it won't execute the query against the database until we attempt to access/iterate over the data in the query, or until we call the ToList() method on it. The code calling our FindUpcomingDinners() method can optionally choose to add additional "chained" operations/filters to the IQueryable<Dinner> object before executing the query. LINQ to SQL is then smart enough to execute the combined query against the database when the data is requested.

To implement paging logic we can update our DinnersController's Index() action method so that it applies

additional "Skip" and "Take" operators to the returned IQueryable<Dinner> sequence before calling ToList() on it:

```
//
// GET: /Dinners/

public ActionResult Index() {

    var upcomingDinners = dinnerRepository.FindUpcomingDinners();
    var paginatedDinners = upcomingDinners.Skip(10).Take(20).ToList();

    return View(paginatedDinners);
}
```

The above code skips over the first 10 upcoming dinners in the database, and then returns back 20 dinners. LINQ to SQL is smart enough to construct an optimized SQL query that performs this skipping logic in the SQL database – and not in the web-server. This means that even if we have millions of upcoming Dinners in the database, only the 10 we want will be retrieved as part of this request (making it efficient and scalable).

Adding a "page" value to the URL

Instead of hard-coding a specific page range, we'll want our URLs to include a "page" parameter that indicates which Dinner range a user is requesting.

Using a Querystring value

The code below demonstrates how we can update our Index() action method to support a querystring parameter and enable URLs like */Dinners?page=2*:

```
//
// GET: /Dinners/
//      /Dinners?page=2

public ActionResult Index(int? page) {

    const int pageSize = 10;

    var upcomingDinners = dinnerRepository.FindUpcomingDinners();

    var paginatedDinners = upcomingDinners.Skip((page ?? 0) * pageSize)
                                          .Take(pageSize)
                                          .ToList();

    return View(paginatedDinners);
}
```

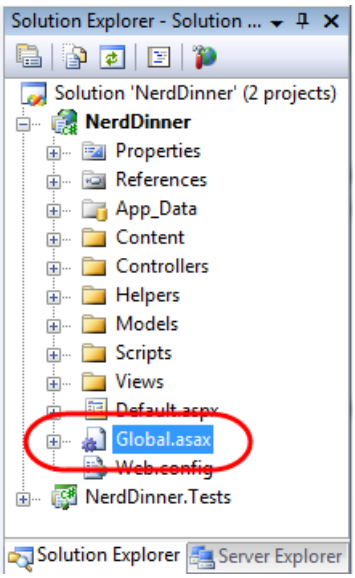
The Index() action method above has a parameter named "page". The parameter is declared as a nullable integer (that is what int? indicates). This means that the */Dinners?page=2* URL will cause a value of "2" to be passed as the parameter value. The */Dinners* URL (without a querystring value) will cause a null value to be passed.

We are multiplying the page value by the page size (in this case 10 rows) to determine how many dinners to skip over. We are using the C# null "coalescing" operator (??) which is useful when dealing with nullable types. The code above assigns page the value of 0 if the page parameter is null.

Using Embedded URL values

An alternative to using a querystring value would be to embed the page parameter within the actual URL itself. For example: */Dinners/Page/2* or */Dinners/2*. ASP.NET MVC includes a powerful URL routing engine that makes it easy to support scenarios like this.

We can register custom routing rules that map any incoming URL or URL format to any controller class or action method we want. All we need to-do is to open the Global.asax file within our project:



And then register a new mapping rule using the MapRoute() helper method like the first call to routes.MapRoute() below:

```
public void RegisterRoutes(RouteCollection routes) {

   routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        "UpcomingDinners",                              // Route name
        "Dinners/Page/{page}",                          // URL with params
        new { controller = "Dinners", action = "Index" } // Param defaults
    );

    routes.MapRoute(
        "Default",                                      // Route name
        "{controller}/{action}/{id}",                   // URL with params
        new { controller="Home", action="Index",id="" }  // Param defaults
    );
}

void Application_Start() {
    RegisterRoutes(RouteTable.Routes);
}
```

Above we are registering a new routing rule named "UpcomingDinners". We are indicating it has the URL format "Dinners/Page/{page}" – where {page} is a parameter value embedded within the URL. The third parameter to the MapRoute() method indicates that we should map URLs that match this format to the Index() action method on the DinnersController class.

We can use the exact same Index() code we had before with our Querystring scenario – except now our "page" parameter will come from the URL and not the querystring:

```
//
// GET: /Dinners/
//      /Dinners/Page/2

public ActionResult Index(int? page) {

    const int pageSize = 10;

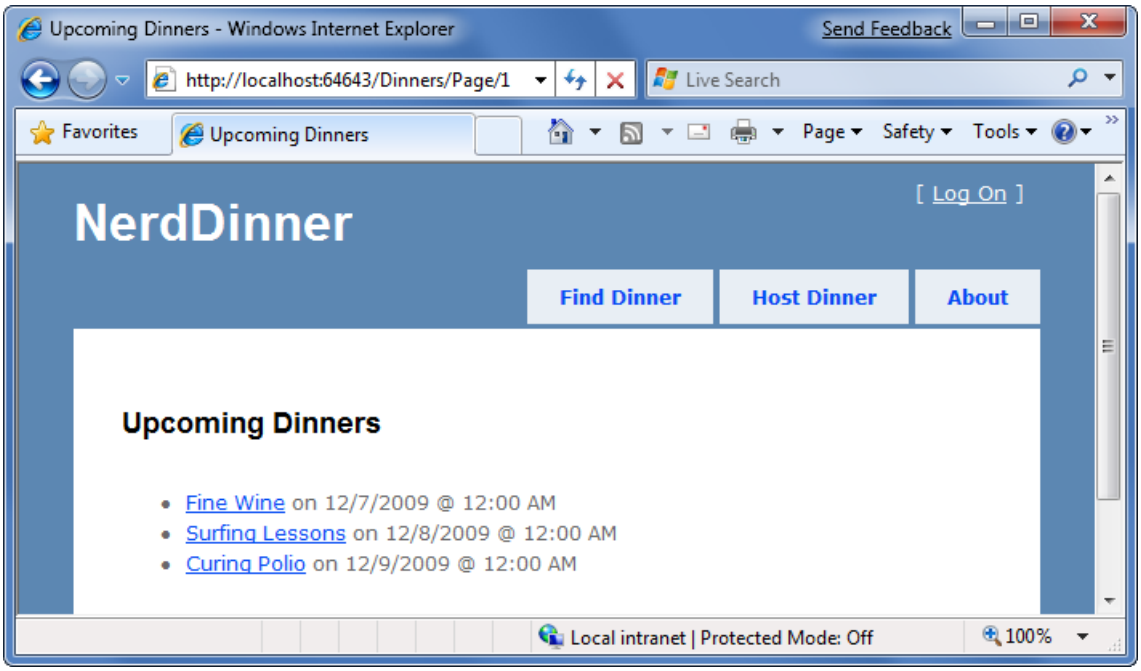
    var upcomingDinners = dinnerRepository.FindUpcomingDinners();

    var paginatedDinners = upcomingDinners.Skip((page ?? 0) * pageSize)
                                          .Take(pageSize)
                                          .ToList();

    return View(paginatedDinners);
}
```

And now when we run the application and type in */Dinners* we'll see the first 10 upcoming dinners:

And when we type in */Dinners/Page/1* we'll see the next page of dinners:



Adding page navigation UI

The last step to complete our paging scenario will be to implement "next" and "previous" navigation UI within our view template to enable users to easily skip over the Dinner data.

To implement this correctly, we'll need to know the total number of Dinners in the database, as well as how many pages of data this translates to. We'll then need to calculate whether the currently requested "page" value is at the beginning or end of the data, and show or hide the "previous" and "next" UI accordingly. We could implement this logic within our Index() action method. Alternatively we can add a helper class to our project that encapsulates this logic in a more re-usable way.

Below is a simple "PaginatedList" helper class that derives from the List<T> collection class built-into the .NET Framework. It implements a re-usable collection class that can be used to paginate any sequence of IQueryable data. In our NerdDinner application we'll have it work over IQueryable<Dinner> results, but it could just as easily be used against IQueryable<Product> or IQueryable<Customer> results in other application scenarios:

```
public class PaginatedList<T> : List<T> {

    public int PageIndex  { get; private set; }
    public int PageSize   { get; private set; }
    public int TotalCount { get; private set; }
    public int TotalPages { get; private set; }

    public PaginatedList(IQueryable<T> source, int pageIndex, int pageSize) {
        PageIndex = pageIndex;
        PageSize = pageSize;
        TotalCount = source.Count();
        TotalPages = (int) Math.Ceiling(TotalCount / (double)PageSize);

        this.AddRange(source.Skip(PageIndex * PageSize).Take(PageSize));
    }

    public bool HasPreviousPage {
        get {
            return (PageIndex > 0);
        }
    }

    public bool HasNextPage {
        get {
            return (PageIndex+1 < TotalPages);
        }
    }
}
```

Notice above how it calculates and then exposes properties like "PageIndex", "PageSize", "TotalCount", and "TotalPages". It also then exposes two helper properties "HasPreviousPage" and "HasNextPage" that indicate whether the page of data in the collection is at the beginning or end of the original sequence. The above code will cause two SQL queries to be run - the first to retrieve the count of the total number of Dinner objects (this doesn't return the objects – rather it performs a "SELECT COUNT" statement that returns an integer), and the second to retrieve just the rows of data we need from our database for the current page of data.

We can then update our DinnersController.Index() helper method to create a PaginatedList<Dinner> from our DinnerRepository.FindUpcomingDinners() result, and pass it to our view template:

```
//
// GET: /Dinners/
//      /Dinners/Page/2

public ActionResult Index(int? page) {

    const int pageSize = 10;

    var upcomingDinners = dinnerRepository.FindUpcomingDinners();
    var paginatedDinners = new PaginatedList<Dinner>(upcomingDinners, page ?? 0, pageSize);

    return View(paginatedDinners);
}
```

We can then update the \Views\Dinners\Index.aspx view template to inherit from ViewPage<NerdDinner.Helpers.PaginatedList<Dinner>> instead of ViewPage<IEnumerable<Dinner>>, and then add the following code to the bottom of our view-template to show or hide next and previous navigation UI:

```
<% if (Model.HasPreviousPage) { %>

    <%= Html.RouteLink("<<<", "UpcomingDinners", new { page = (Model.PageIndex-1) }) %>

<% } %>

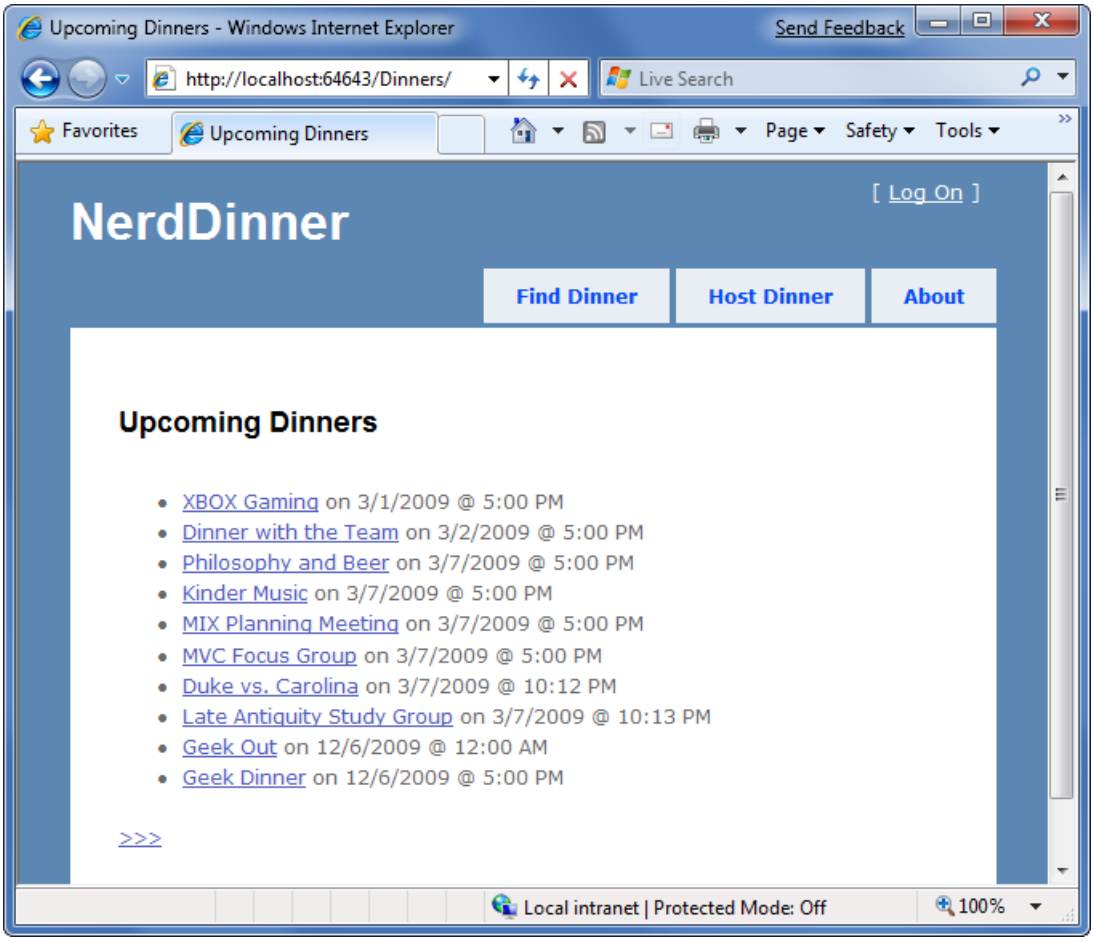
<% if (Model.HasNextPage) {  %>

    <%= Html.RouteLink(">>>", "UpcomingDinners", new { page = (Model.PageIndex + 1) }) %>

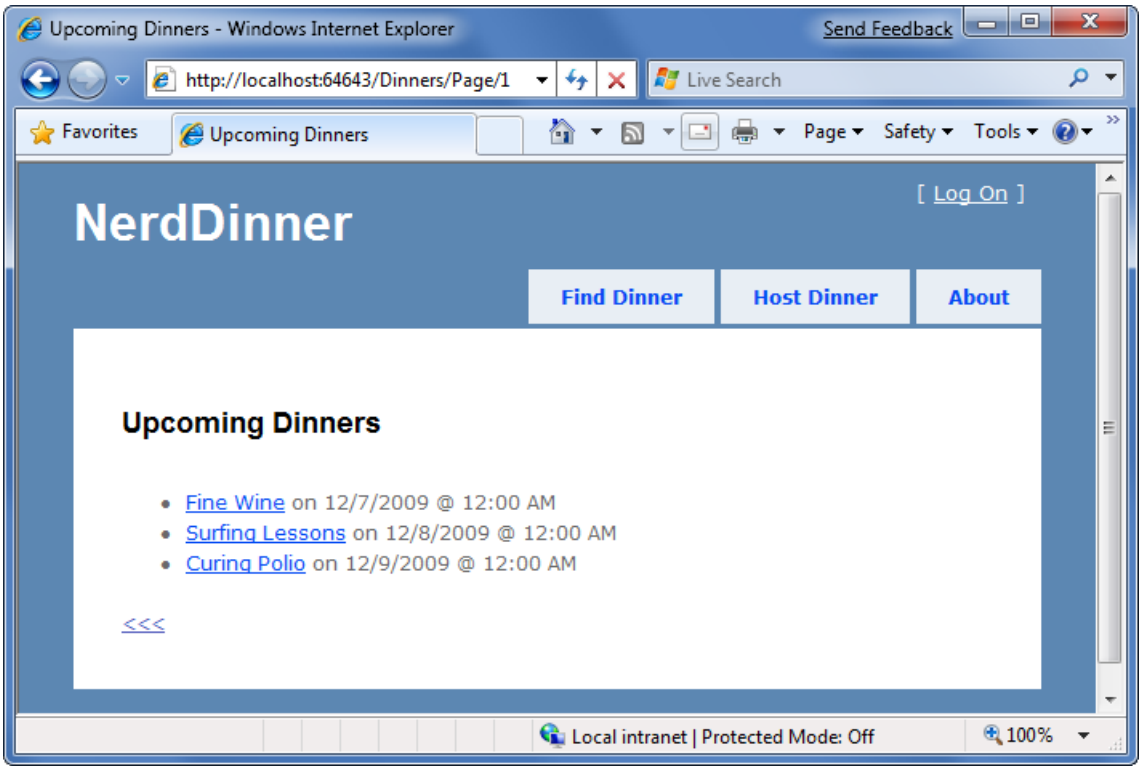
<% } %>
```

Notice above how we are using the Html.RouteLink() helper method to generate our hyperlinks. This method is similar to the Html.ActionLink() helper method we've used previously. The difference is that we are generating the URL using the "UpcomingDinners" routing rule we setup within our Global.asax file. This ensures that we'll generate URLs to our Index() action method that have the format: */Dinners/Page/{page}* – where the {page} value is a variable we are providing above based on the current PageIndex.

And now when we run our application again we'll see 10 dinners at a time in our browser:



We also have <<< and >>> navigation UI at the bottom of the page that allows us to skip forwards and backwards over our data using search engine accessible URLs:



SIDE TOPIC: UNDERSTANDING THE IMPLICATIONS OF IQUERYABLE<T>

IQueryable<T> is a very powerful feature that enables a variety of interesting deferred execution scenarios (like paging and composition based queries). As with all powerful features, you want to be careful with how you use it and make sure it is not abused. It is important to recognize that returning an IQueryable<T> result from your repository enables calling code to append on chained operator methods to it, and so participate in the ultimate query execution. If you do not want to provide calling code this ability, then you should return back IList<T> or IEnumerable<T> results - which contain the results of a query that has already executed. For pagination scenarios this would require you to push the actual data pagination logic into the repository method being called. In this scenario we might update our FindUpcomingDinners() finder method to have a signature that either returned a PaginatedList: PaginatedList< Dinner> FindUpcomingDinners(int pageIndex, int pageSize) { } Or return back an IList<Dinner>, and use a "totalCount" out param to return the total count of Dinners: IList<Dinner> FindUpcomingDinners(int pageIndex, int pageSize, out int totalCount) { }

Next Step

Let's now look at how we can add authentication and authorization support to our application.

PREVIOUS NEXT

Secure Applications Using Authentication and Authorization

4/10/2020 • 10 minutes to read • Edit Online

by Microsoft

Download PDF

> This is step 9 of a free "NerdDinner" application tutorial that walks-through how to build a small, but complete, web application using ASP.NET MVC 1.
>
> Step 9 shows how to add authentication and authorization to secure our NerdDinner application, so that users need to register and login to the site to create new dinners, and only the user who is hosting a dinner can edit it later.
>
> If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

NerdDinner Step 9: Authentication and Authorization

Right now our NerdDinner application grants anyone visiting the site the ability to create and edit the details of any dinner. Let's change this so that users need to register and login to the site to create new dinners, and add a restriction so that only the user who is hosting a dinner can edit it later.

To enable this we'll use authentication and authorization to secure our application.

Understanding Authentication and Authorization

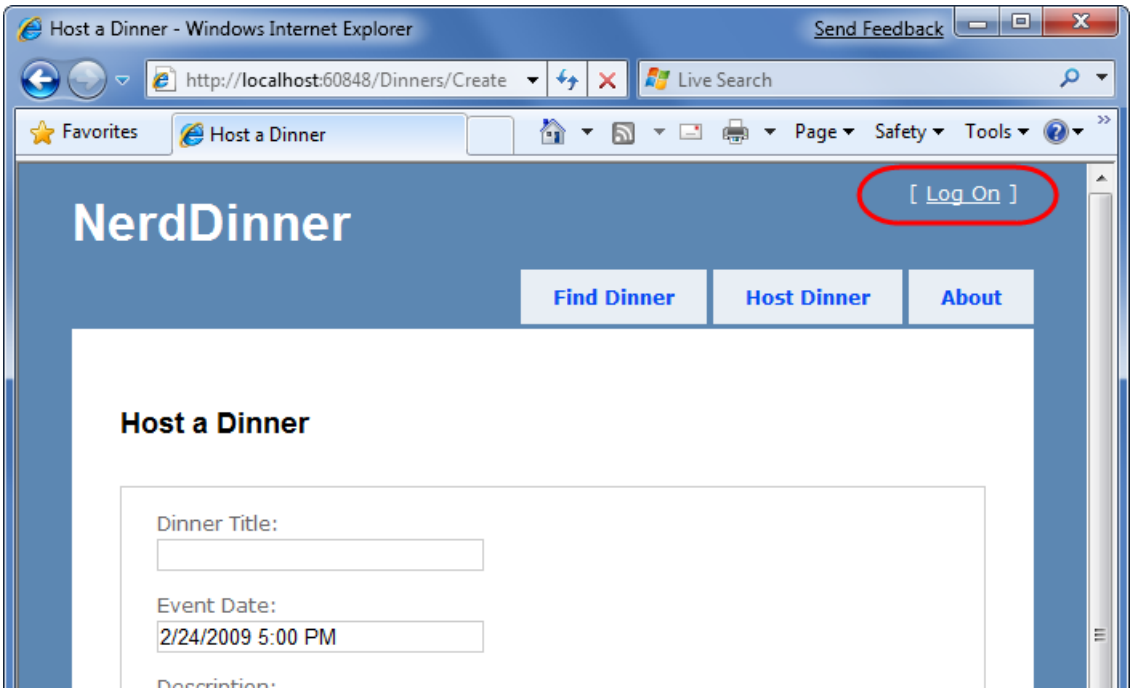
Authentication is the process of identifying and validating the identity of a client accessing an application. Put more simply, it is about identifying "who" the end-user is when they visit a website. ASP.NET supports multiple ways to authenticate browser users. For Internet web applications, the most common authentication approach used is called "Forms Authentication". Forms Authentication enables a developer to author an HTML login form within their application, and then validate the username/password an end-user submits against a database or other password credential store. If the username/password combination is correct, the developer can then ask ASP.NET to issue an encrypted HTTP cookie to identify the user across future requests. We'll by using forms authentication with our NerdDinner application.

Authorization is the process of determining whether an authenticated user has permission to access a particular URL/resource or to perform some action. For example, within our NerdDinner application we'll want to authorize that only users who are logged in can access the */Dinners/Create* URL and create new Dinners. We'll also want to add authorization logic so that only the user who is hosting a dinner can edit it – and deny edit access to all other users.

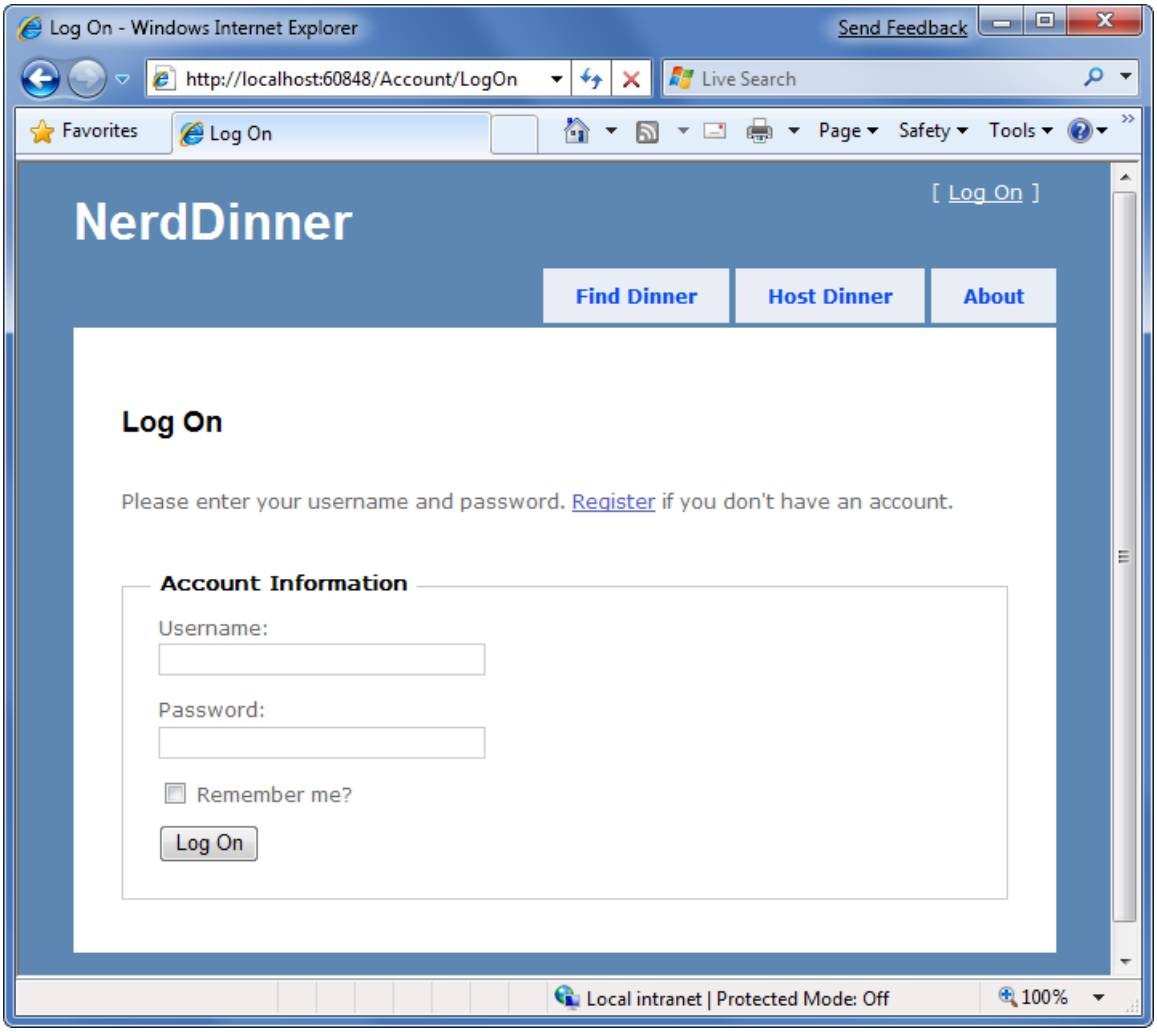
Forms Authentication and the AccountController

The default Visual Studio project template for ASP.NET MVC automatically enables forms authentication when new ASP.NET MVC applications are created. It also automatically adds a pre-built account login page implementation to the project – which makes it really easy to integrate security within a site.

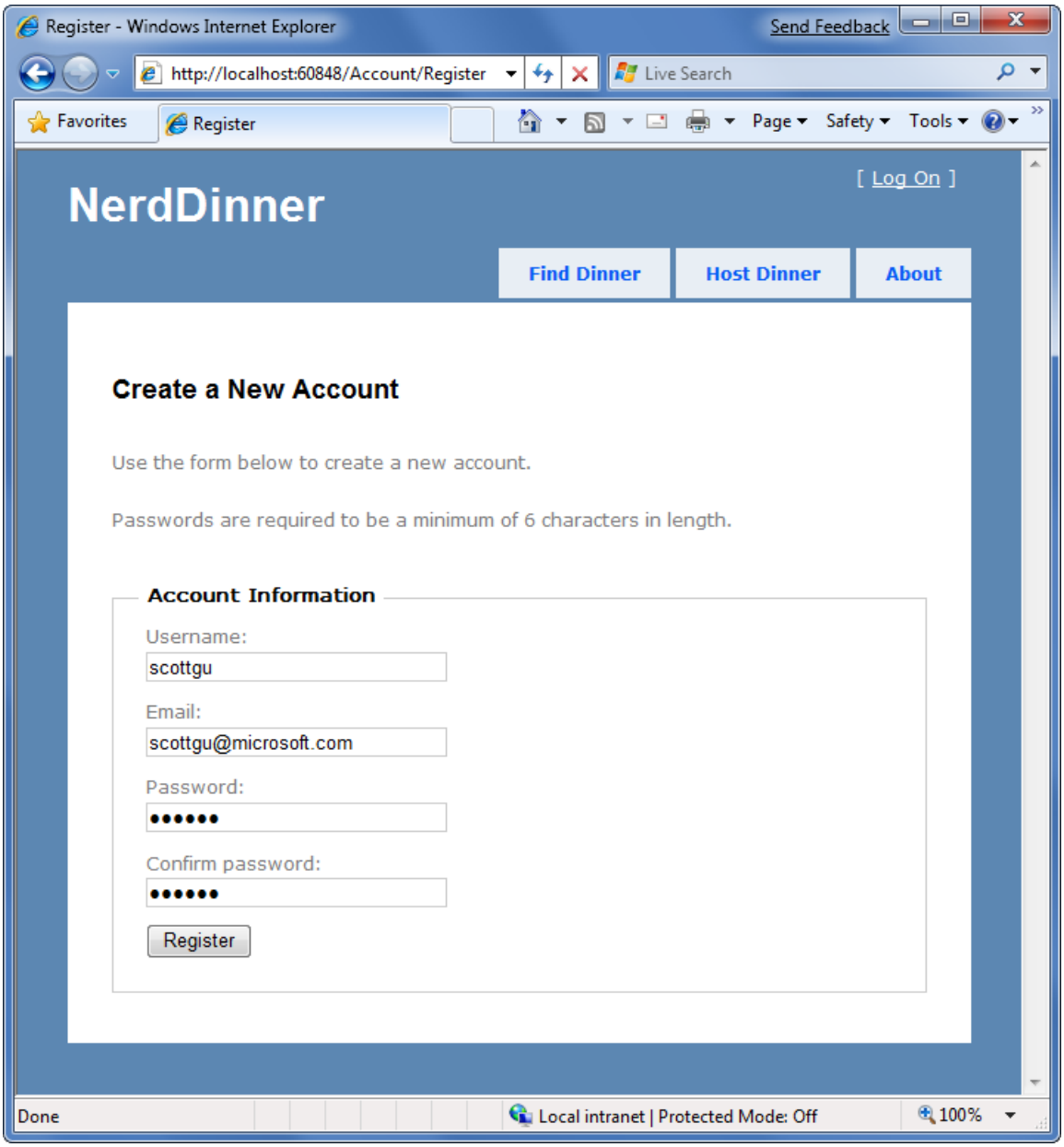
The default Site.master master page displays a "Log On" link at the top-right of the site when the user accessing it is not authenticated:



Clicking the "Log On" link takes a user to the */Account/LogOn* URL:

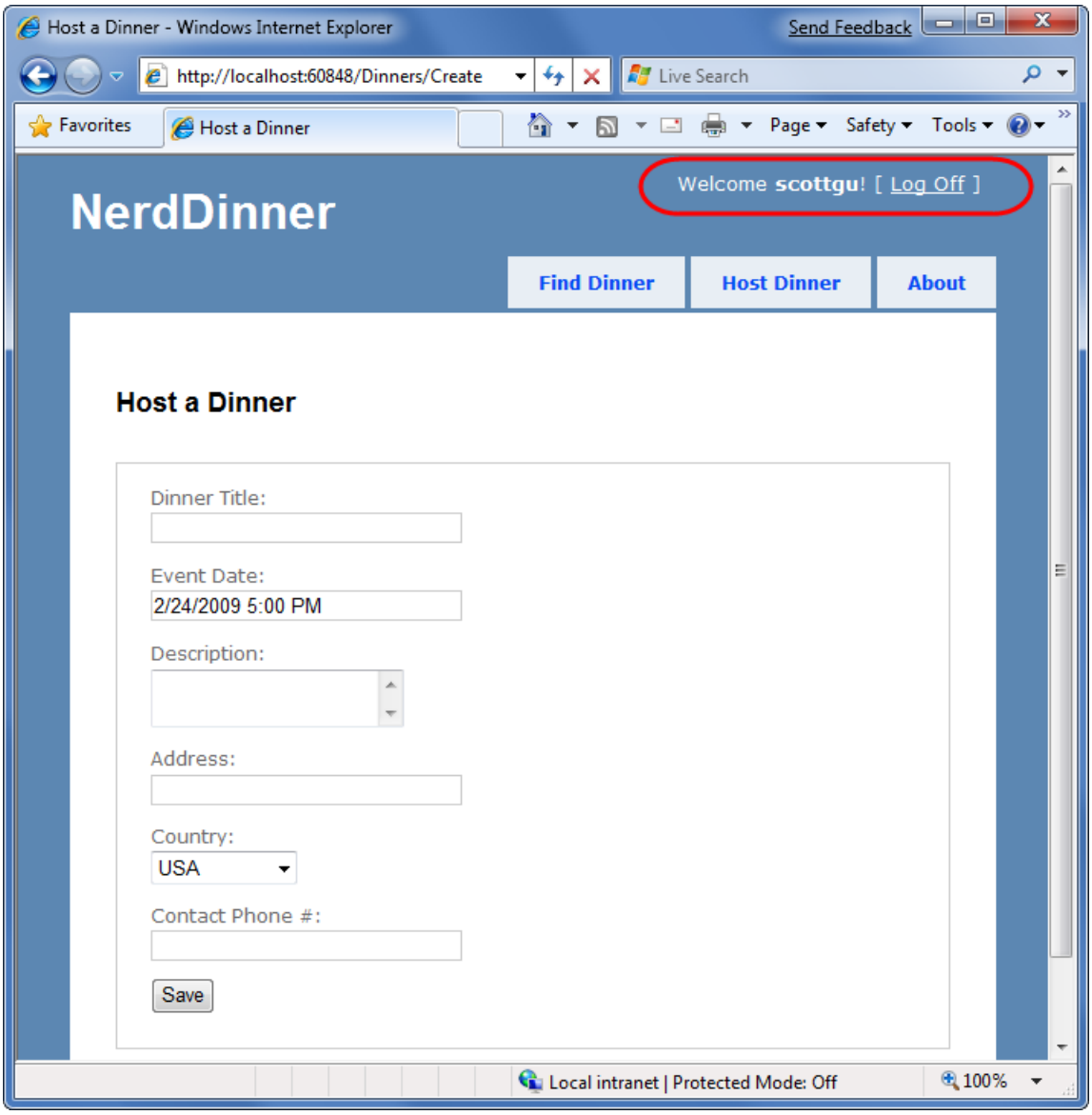


Visitors who haven't registered can do so by clicking the "Register" link – which will take them to the */Account/Register* URL and allow them to enter account details:

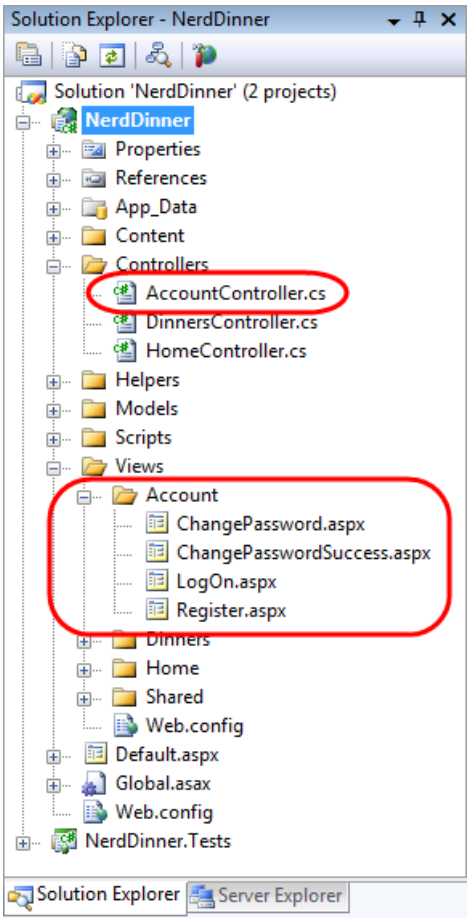


Clicking the "Register" button will create a new user within the ASP.NET Membership system, and authenticate the user onto the site using forms authentication.

When a user is logged-in, the Site.master changes the top-right of the page to output a "Welcome [username]!" message and renders a "Log Off" link instead of a "Log On" one. Clicking the "Log Off" link logs out the user:



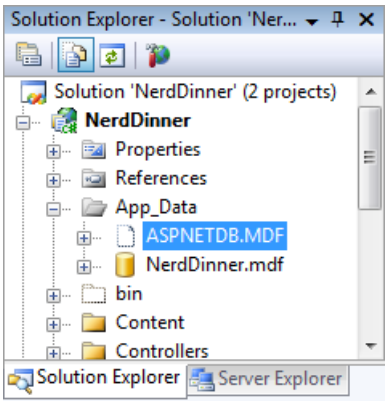
The above login, logout, and registration functionality is implemented within the AccountController class that was added to our project by Visual Studio when it created the project. The UI for the AccountController is implemented using view templates within the \Views\Account directory:



The AccountController class uses the ASP.NET Forms Authentication system to issue encrypted authentication cookies, and the ASP.NET Membership API to store and validate usernames/passwords. The ASP.NET Membership API is extensible and enables any password credential store to be used. ASP.NET ships with built-in membership provider implementations that store username/passwords within a SQL database, or within Active Directory.

We can configure which membership provider our NerdDinner application should use by opening the "web.config" file at the root of the project and looking for the <membership> section within it. The default web.config added when the project was created registers the SQL membership provider, and configures it to use a connection-string named "ApplicationServices" to specify the database location.

The default "ApplicationServices" connection string (which is specified within the <connectionStrings> section of the web.config file) is configured to use SQL Express. It points to a SQL Express database named "ASPNETDB.MDF" under the application's "App_Data" directory. If this database doesn't exist the first time the Membership API is used within the application, ASP.NET will automatically create the database and provision the appropriate membership database schema within it:



If instead of using SQL Express we wanted to use a full SQL Server instance (or connect to a remote database), all we'd need to-do is to update the "ApplicationServices" connection string within the web.config file and make sure that the appropriate membership schema has been added to the database it points at. You can run the

"aspnet_regsql.exe" utility within the \Windows\Microsoft.NET\Framework\v2.0.50727\ directory to add the appropriate schema for membership and the other ASP.NET application services to a database.

Authorizing the /Dinners/Create URL using the [Authorize] filter

We didn't have to write any code to enable a secure authentication and account management implementation for the NerdDinner application. Users can register new accounts with our application, and login/logout of the site.

Now we can add authorization logic to the application, and use the authentication status and username of visitors to control what they can and can't do within the site. Let's begin by adding authorization logic to the "Create" action methods of our DinnersController class. Specifically, we will require that users accessing the */Dinners/Create* URL must be logged in. If they aren't logged in we'll redirect them to the login page so that they can sign-in.

Implementing this logic is pretty easy. All we need to-do is to add an [Authorize] filter attribute to our Create action methods like so:

```
//
// GET: /Dinners/Create

[Authorize]
public ActionResult Create() {
   ...
}

//
// POST: /Dinners/Create

[AcceptVerbs(HttpVerbs.Post), Authorize]
public ActionResult Create(Dinner dinnerToCreate) {
   ...
}
```

ASP.NET MVC supports the ability to create "action filters" that can be used to implement re-usable logic that can be declaratively applied to action methods. The [Authorize] filter is one of the built-in action filters provided by ASP.NET MVC, and it enables a developer to declaratively apply authorization rules to action methods and controller classes.

When applied without any parameters (like above) the [Authorize] filter enforces that the user making the action method request must be logged in – and it will automatically redirect the browser to the login URL if they aren't. When doing this redirect the originally requested URL is passed as a querystring argument (for example: /Account/LogOn?ReturnUrl=%2fDinners%2fCreate). The AccountController will then redirect the user back to the originally requested URL once they login.

The [Authorize] filter optionally supports the ability to specify a "Users" or "Roles" property that can be used to require that the user is both logged in and within a list of allowed users or a member of an allowed security role. For example, the code below only allows two specific users, "scottgu" and "billg", to access the /Dinners/Create URL:

```
[Authorize(Users="scottgu,billg")]
public ActionResult Create() {
    ...
}
```

Embedding specific user names within code tends to be pretty un-maintainable though. A better approach is to define higher-level "roles" that the code checks against, and then to map users into the role using either a database or active directory system (enabling the actual user mapping list to be stored externally from the code). ASP.NET includes a built-in role management API as well as a built-in set of role providers (including ones for SQL

and Active Directory) that can help perform this user/role mapping. We could then update the code to only allow users within a specific "admin" role to access the /Dinners/Create URL:

```
[Authorize(Roles="admin")]
public ActionResult Create() {
   ...
}
```

Using the User.Identity.Name property when Creating Dinners

We can retrieve the username of the currently logged-in user of a request using the User.Identity.Name property exposed on the Controller base class.

Earlier when we implemented the HTTP-POST version of our Create() action method we had hardcoded the "HostedBy" property of the Dinner to a static string. We can now update this code to instead use the User.Identity.Name property, as well as automatically add an RSVP for the host creating the Dinner:

```
//
// POST: /Dinners/Create

[AcceptVerbs(HttpVerbs.Post), Authorize]
public ActionResult Create(Dinner dinner) {

    if (ModelState.IsValid) {

        try {
            dinner.HostedBy = User.Identity.Name;

            RSVP rsvp = new RSVP();
            rsvp.AttendeeName = User.Identity.Name;
            dinner.RSVPs.Add(rsvp);

            dinnerRepository.Add(dinner);
            dinnerRepository.Save();

            return RedirectToAction("Details", new { id=dinner.DinnerID });
        }
        catch {
            ModelState.AddModelErrors(dinner.GetRuleViolations());
        }
    }

    return View(new DinnerFormViewModel(dinner));
}
```

Because we have added an [Authorize] attribute to the Create() method, ASP.NET MVC ensures that the action method only executes if the user visiting the /Dinners/Create URL is logged in on the site. As such, the User.Identity.Name property value will always contain a valid username.

Using the User.Identity.Name property when Editing Dinners

Let's now add some authorization logic that restricts users so that they can only edit the properties of dinners they themselves are hosting.

To help with this, we'll first add an "IsHostedBy(username)" helper method to our Dinner object (within the Dinner.cs partial class we built earlier). This helper method returns true or false depending on whether a supplied username matches the Dinner HostedBy property, and encapsulates the logic necessary to perform a case-insensitive string comparison of them:

```
public partial class Dinner {

    public bool IsHostedBy(string userName) {
        return HostedBy.Equals(userName, StringComparison.InvariantCultureIgnoreCase);
    }
}
```

We'll then add an [Authorize] attribute to the Edit() action methods within our DinnersController class. This will ensure that users must be logged in to request a */Dinners/Edit/[id]* URL.

We can then add code to our Edit methods that uses the Dinner.IsHostedBy(username) helper method to verify that the logged-in user matches the Dinner host. If the user is not the host, we'll display an "InvalidOwner" view and terminate the request. The code to do this looks like below:

```
//
// GET: /Dinners/Edit/5

[Authorize]
public ActionResult Edit(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (!dinner.IsHostedBy(User.Identity.Name))
        return View("InvalidOwner");

    return View(new DinnerFormViewModel(dinner));
}

//
// POST: /Dinners/Edit/5

[AcceptVerbs(HttpVerbs.Post), Authorize]
public ActionResult Edit(int id, FormCollection collection) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (!dinner.IsHostedBy(User.Identity.Name))
        return View("InvalidOwner");

    try {
        UpdateModel(dinner);

        dinnerRepository.Save();

        return RedirectToAction("Details", new {id = dinner.DinnerID});
    }
    catch {
        ModelState.AddModelErrors(dinnerToEdit.GetRuleViolations());

        return View(new DinnerFormViewModel(dinner));
    }
}
```

We can then right-click on the \Views\Dinners directory and choose the Add->View menu command to create a new "InvalidOwner" view. We'll populate it with the below error message:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent" runat="server">
    You Don't Own This Dinner
</asp:Content>

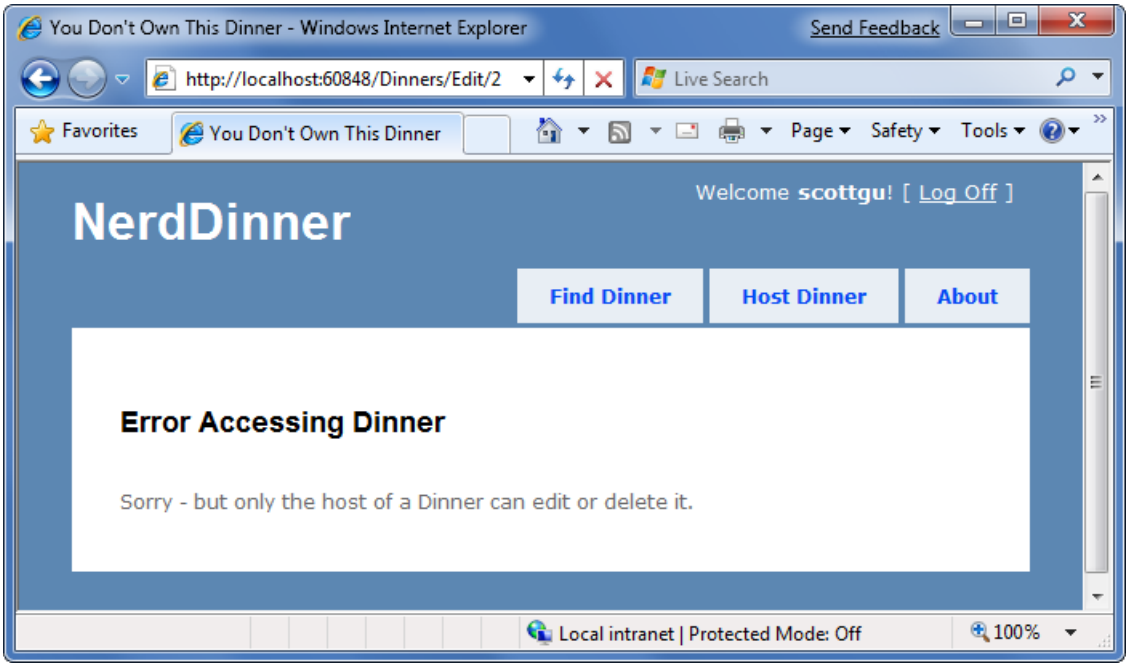
<asp:Content ID="Main" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Error Accessing Dinner</h2>

    <p>Sorry - but only the host of a Dinner can edit or delete it.</p>

</asp:Content>
```

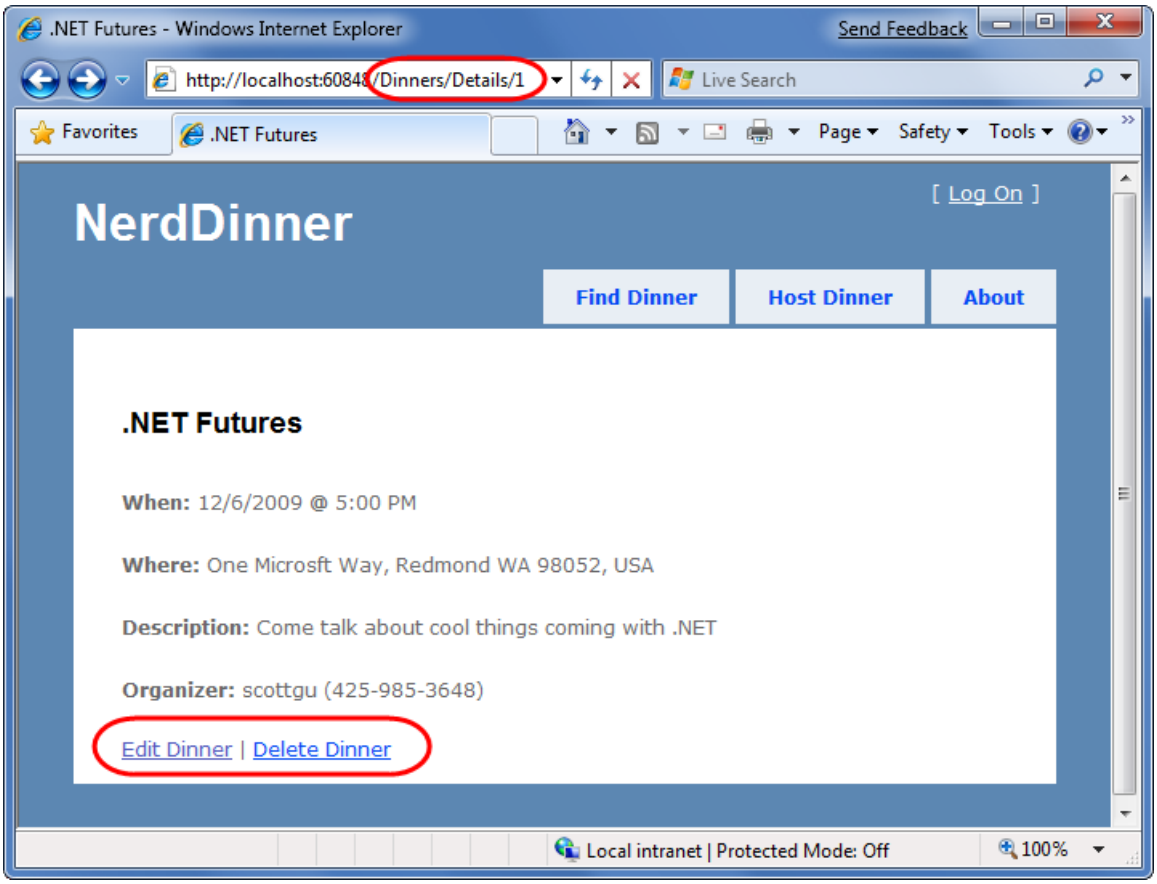
And now when a user attempts to edit a dinner they don't own, they'll get an error message:



We can repeat the same steps for the Delete() action methods within our controller to lock down permission to delete Dinners as well, and ensure that only the host of a Dinner can delete it.

Showing/Hiding Edit and Delete Links

We are linking to the Edit and Delete action method of our DinnersController class from our Details URL:



Currently we are showing the Edit and Delete action links regardless of whether the visitor to the details URL is the host of the dinner. Let's change this so that the links are only displayed if the visiting user is the owner of the dinner.

The Details() action method within our DinnersController retrieves a Dinner object and then passes it as the model object to our view template:

```
//
// GET: /Dinners/Details/5

public ActionResult Details(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (dinner == null)
        return View("NotFound");

    return View(dinner);
}
```

We can update our view template to conditionally show/hide the Edit and Delete links by using the Dinner.IsHostedBy() helper method like below:

```
<% if (Model.IsHostedBy(Context.User.Identity.Name)) { %>

   <%= Html.ActionLink("Edit Dinner", "Edit", new { id=Model.DinnerID }) %> |
   <%= Html.ActionLink("Delete Dinner", "Delete", new {id=Model.DinnerID}) %>

<% } %>
```

Next Steps

Let's now look at how we can enable authenticated users to RSVP for dinners using AJAX.

Use AJAX to Deliver Dynamic Updates

4/10/2020 • 8 minutes to read • Edit Online

by Microsoft

Download PDF

This is step 10 of a free "NerdDinner" application tutorial that walks-through how to build a small, but complete, web application using ASP.NET MVC 1.

Step 10 implements support for logged-in users to RSVP their interest in attending a dinner, using an Ajax-based approach integrated within the dinner details page.

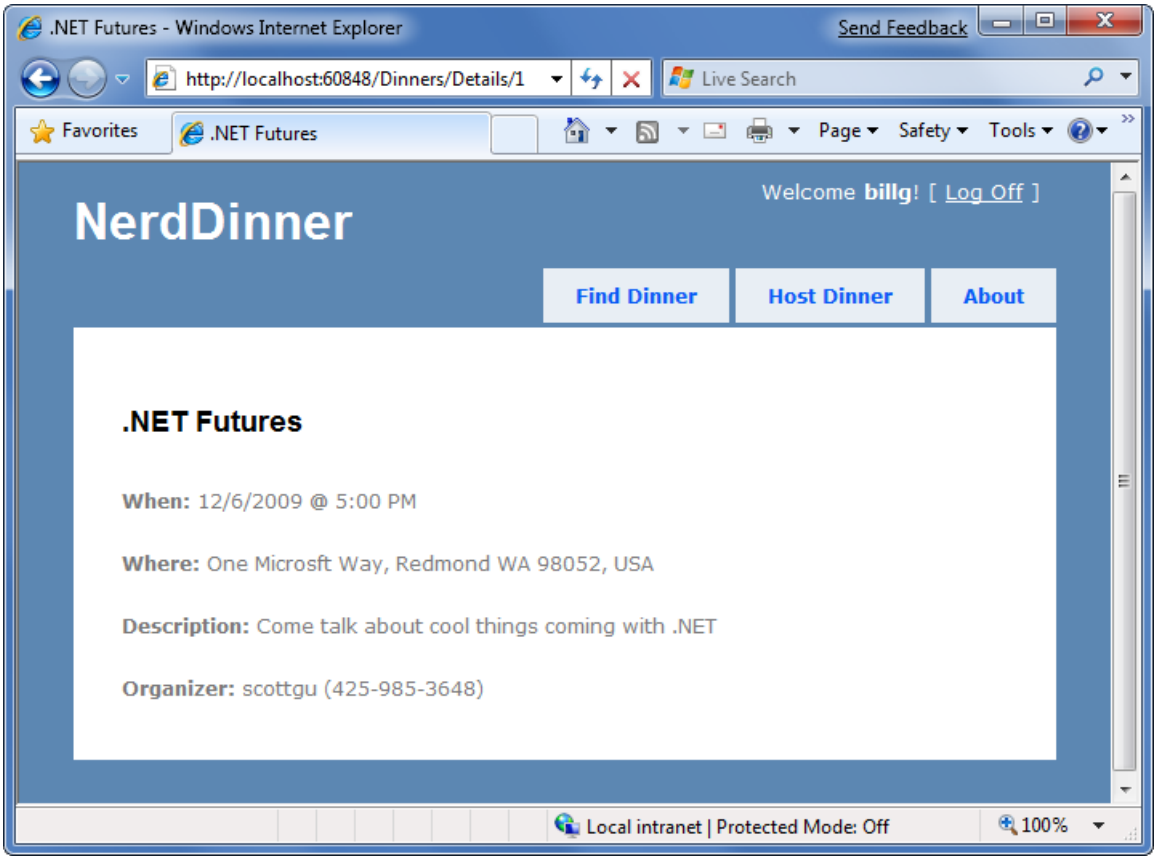
If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

NerdDinner Step 10: AJAX Enabling RSVPs Accepts

Let's now implement support for logged-in users to RSVP their interest in attending a dinner. We'll enable this using an AJAX-based approach integrated within the dinner details page.

Indicating whether the user is RSVP'd

Users can visit the */Dinners/Details/[id]* URL to see details about a particular dinner:



The Details() action method is implemented like so:

```
//
// GET: /Dinners/Details/2

public ActionResult Details(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (dinner == null)
        return View("NotFound");
    else
        return View(dinner);
}
```

Our first step to implement RSVP support will be to add an "IsUserRegistered(username)" helper method to our Dinner object (within the Dinner.cs partial class we built earlier). This helper method returns true or false depending on whether the user is currently RSVP'd for the Dinner:

```
public partial class Dinner {

    public bool IsUserRegistered(string userName) {
        return RSVPs.Any(r => r.AttendeeName.Equals(userName, StringComparison.InvariantCultureIgnoreCase));
    }
}
```

We can then add the following code to our Details.aspx view template to display an appropriate message indicating whether the user is registered or not for the event:

```
<% if (Request.IsAuthenticated) { %>

    <% if (Model.IsUserRegistered(Context.User.Identity.Name)) { %>

        <p>You are registred for this event!</p>

    <% } else {  %>

        <p>You are not registered for this event</p>

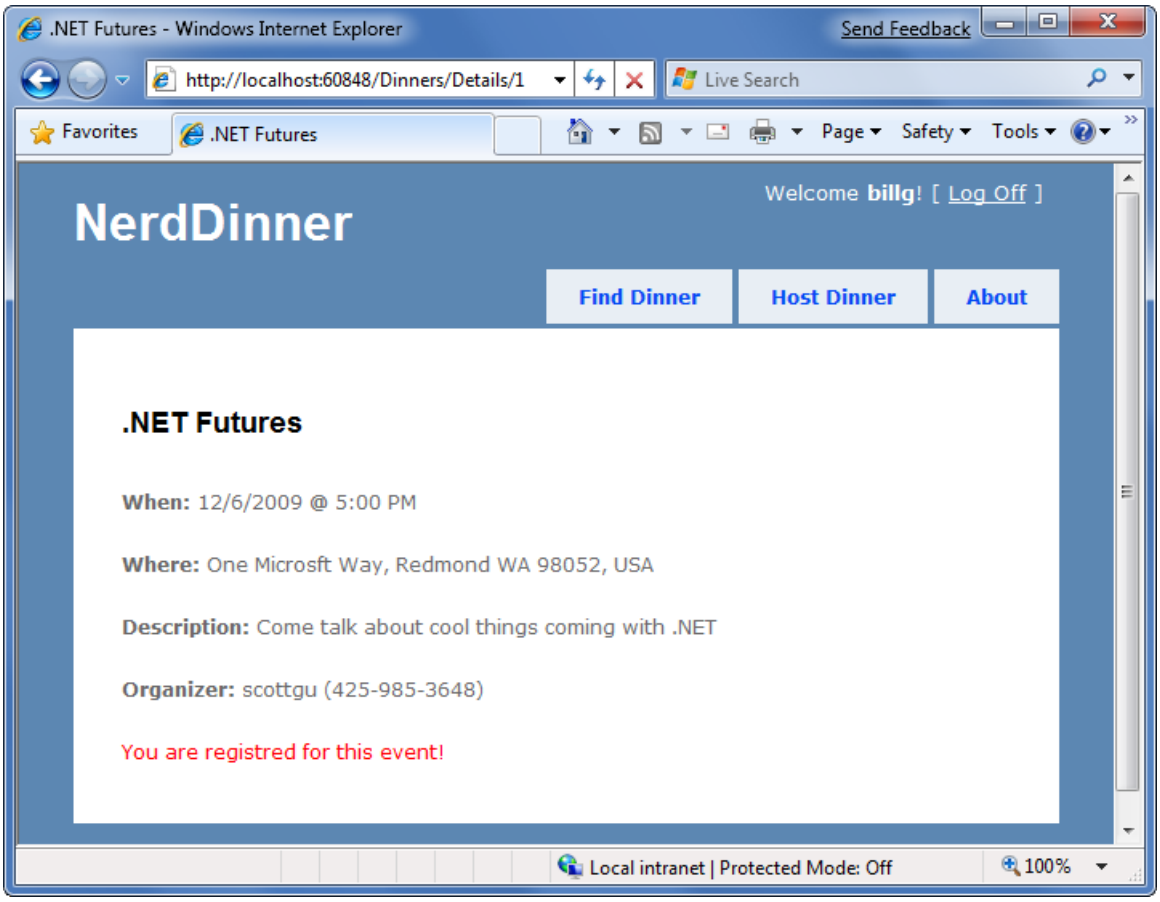
    <% }  %>

<% } else { %>

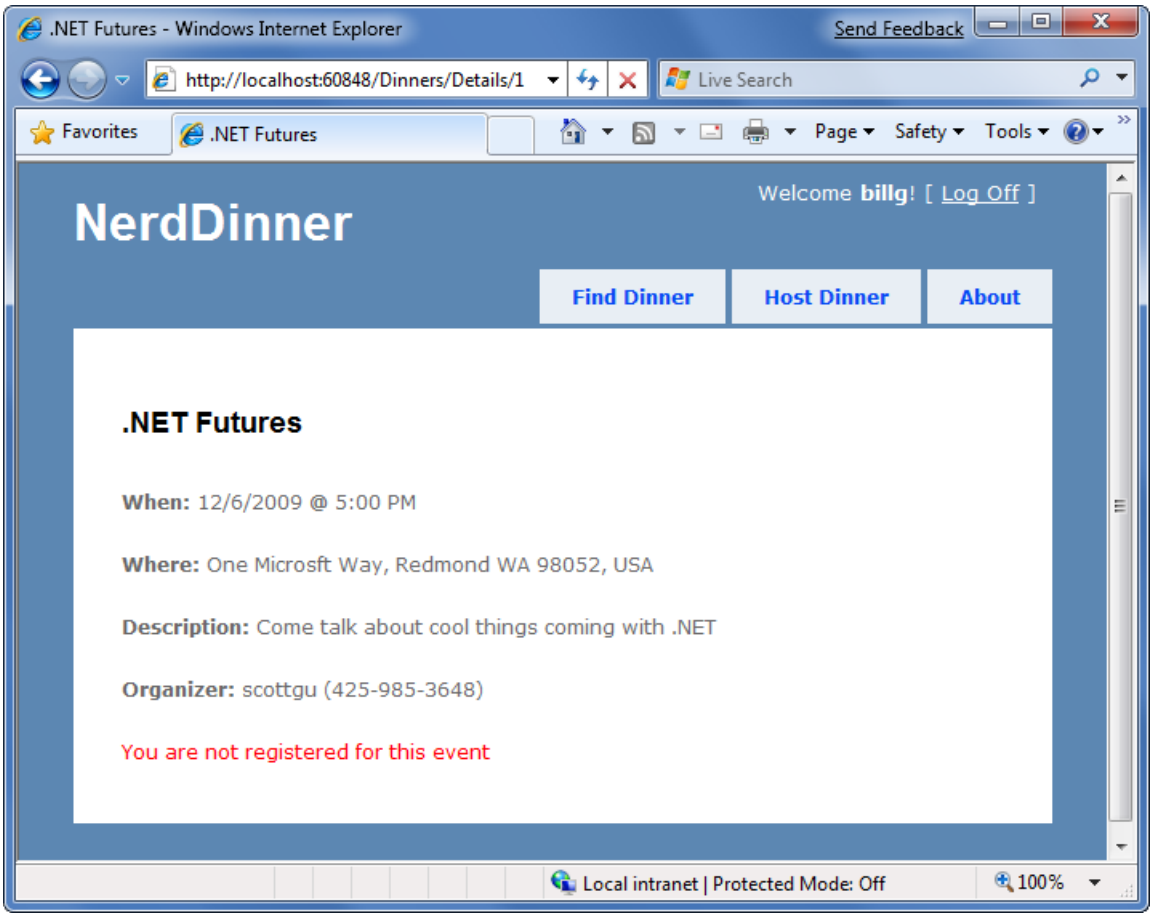
    <a href="/Account/Logon">Logon</a> to RSVP for this event.

<% } %>
```

And now when a user visits a Dinner they are registered for they'll see this message:



And when they visit a Dinner they are not registered for they'll see the below message:



Implementing the Register Action Method

Let's now add the functionality necessary to enable users to RSVP for a dinner from the details page.

To implement this, we'll create a new "RSVPController" class by right-clicking on the \Controllers directory and

choosing the Add->Controller menu command.

We'll implement a "Register" action method within the new RSVPController class that takes an id for a Dinner as an argument, retrieves the appropriate Dinner object, checks to see if the logged-in user is currently in the list of users who have registered for it, and if not adds an RSVP object for them:

```
public class RSVPController : Controller {

    DinnerRepository dinnerRepository = new DinnerRepository();

    //
    // AJAX: /Dinners/RSVPForEvent/1

    [Authorize, AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Register(int id) {

        Dinner dinner = dinnerRepository.GetDinner(id);

        if (!dinner.IsUserRegistered(User.Identity.Name)) {

            RSVP rsvp = new RSVP();
            rsvp.AttendeeName = User.Identity.Name;

            dinner.RSVPs.Add(rsvp);
            dinnerRepository.Save();
        }

        return Content("Thanks - we'll see you there!");
    }
}
```

Notice above how we are returning a simple string as the output of the action method. We could have embedded this message within a view template – but since it is so small we'll just use the Content() helper method on the controller base class and return a string message like above.

Calling the RSVPForEvent Action Method using AJAX

We'll use AJAX to invoke the Register action method from our Details view. Implementing this is pretty easy. First we'll add two script library references:

```
<script src="/Scripts/MicrosoftAjax.js" type="text/javascript"></script>
<script src="/Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>
```

The first library references the core ASP.NET AJAX client-side script library. This file is approximately 24k in size (compressed) and contains core client-side AJAX functionality. The second library contains utility functions that integrate with ASP.NET MVC's built-in AJAX helper methods (which we'll use shortly).

We can then update the view template code we added earlier so that instead of outputting a "You are not registered for this event" message, we instead render a link that when pushed performs an AJAX call that invokes our RSVPForEvent action method on our RSVP controller and RSVPs the user:

```
<div id="rsvpmsg">

<% if(Request.IsAuthenticated) { %>

    <% if(Model.IsUserRegistered(Context.User.Identity.Name)) { %>

        <p>You are registred for this event!</p>

    <% } else { %>

        <%= Ajax.ActionLink( "RSVP for this event",
                             "Register", "RSVP",
                             new { id=Model.DinnerID },
                             new AjaxOptions { UpdateTargetId="rsvpmsg"}) %>
    <% } %>

<% } else { %>

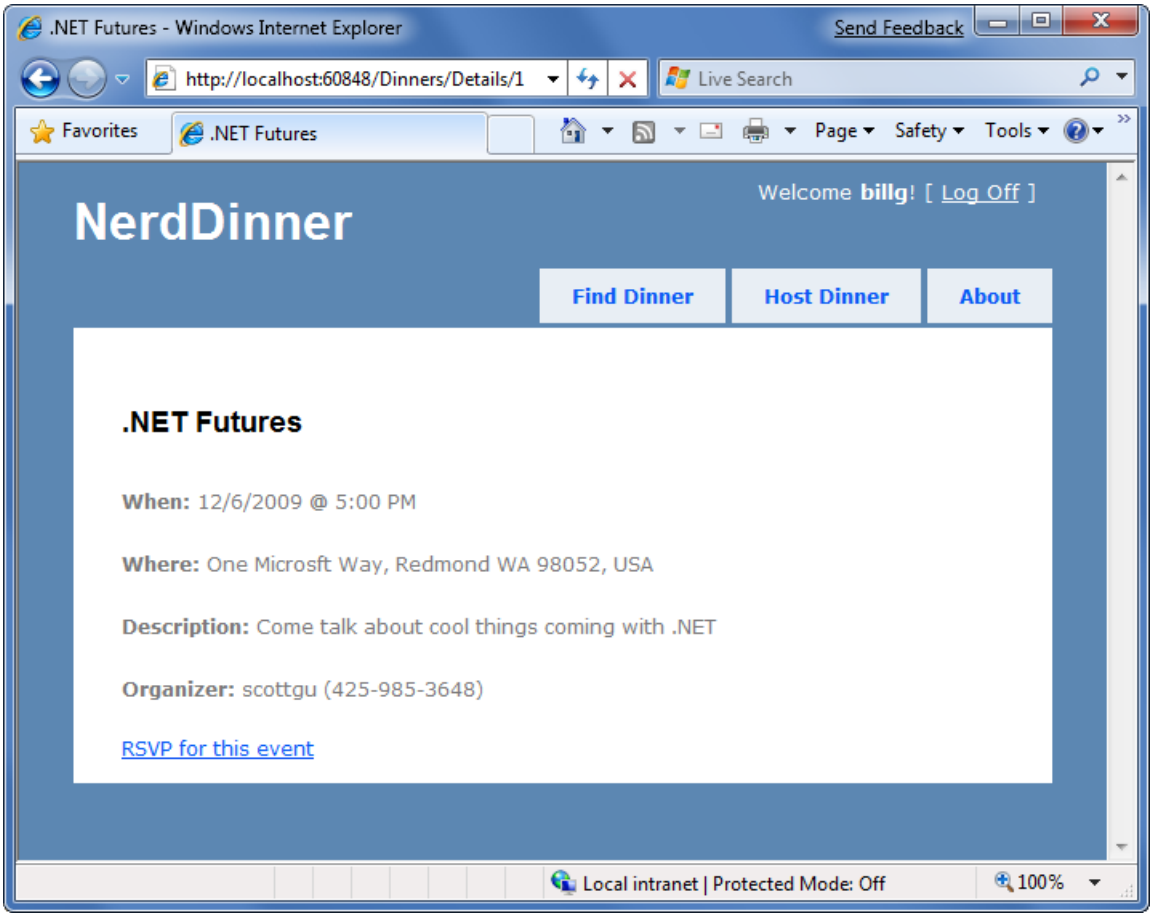
    <a href="/Account/Logon">Logon</a> to RSVP for this event.

<% } %>

</div>
```

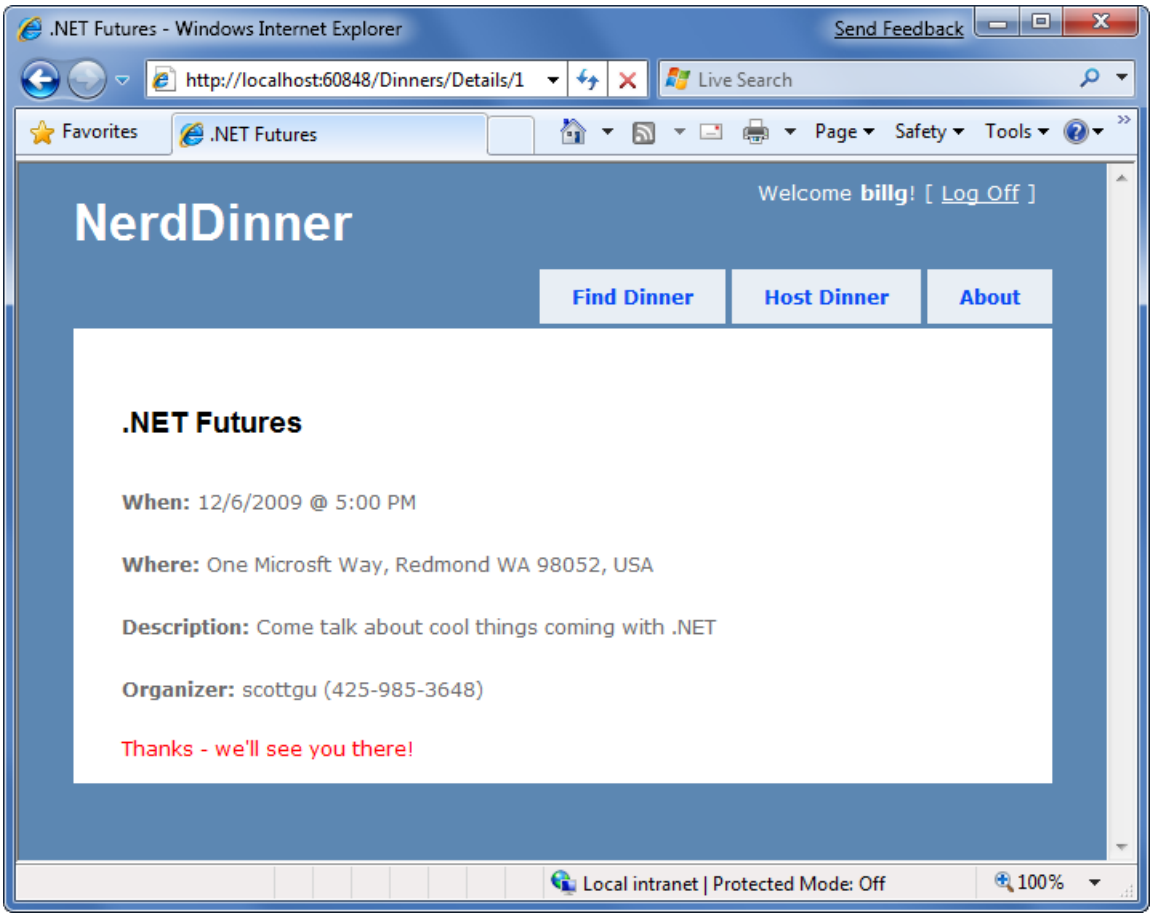
The Ajax.ActionLink() helper method used above is built-into ASP.NET MVC and is similar to the Html.ActionLink() helper method except that instead of performing a standard navigation it makes an AJAX call to the action method when the link is clicked. Above we are calling the "Register" action method on the "RSVP" controller and passing the DinnerID as the "id" parameter to it. The final AjaxOptions parameter we are passing indicates that we want to take the content returned from the action method and update the HTML element on the page whose id is "rsvpmsg".

And now when a user browses to a dinner they aren't registered for yet they'll see a link to RSVP for it:



If they click the "RSVP for this event" link they'll make an AJAX call to the Register action method on the RSVP

controller, and when it completes they'll see an updated message like below:



The network bandwidth and traffic involved when making this AJAX call is really lightweight. When the user clicks on the "RSVP for this event" link, a small HTTP POST network request is made to the */Dinners/Register/1* URL that looks like below on the wire:

```
POST /Dinners/Register/49 HTTP/1.1
X-Requested-With: XMLHttpRequest
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Referer: http://localhost:8080/Dinners/Details/49
```

And the response from our Register action method is simply:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 29
Thanks - we'll see you there!
```

This lightweight call is fast and will work even over a slow network.

Adding a jQuery Animation

The AJAX functionality we implemented works well and fast. Sometimes it can happen so fast, though, that a user might not notice that the RSVP link has been replaced with new text. To make the outcome a little more obvious we can add a simple animation to draw attention to the update message.

The default ASP.NET MVC project template includes jQuery – an excellent (and very popular) open source JavaScript library that is also supported by Microsoft. jQuery provides a number of features, including a nice HTML DOM selection and effects library.

To use jQuery we'll first add a script reference to it. Because we are going to be using jQuery within a variety of places within our site, we'll add the script reference within our Site.master master page file so that all pages can

use it.

```
<script src="/Scripts/jQuery-1.3.2.js" type="text/javascript"></script>
```

Tip: make sure you have installed the JavaScript intellisense hotfix for VS 2008 SP1 that enables richer intellisense support for JavaScript files (including jQuery). You can download it from: http://tinyurl.com/vs2008javascripthotfix

Code written using JQuery often uses a global "$()" JavaScript method that retrieves one or more HTML elements using a CSS selector. For example, *$("#rsvpmsg")* selects any HTML element with the id of rsvpmsg, while *$(".something")* would select all elements with the "something" CSS class name. You can also write more advanced queries like "return all of the checked radio buttons" using a selector query like: *$("input[@type=radio][@checked]")*.

Once you've selected elements, you can call methods on them to take action, like hiding them: *$("#rsvpmsg").hide();*

For our RSVP scenario, we'll define a simple JavaScript function named "AnimateRSVPMessage" that selects the "rsvpmsg" and animates the size of its text content. The below code starts the text small and then causes it to increase over a 400 milliseconds timeframe:

```
<script type="text/javascript">

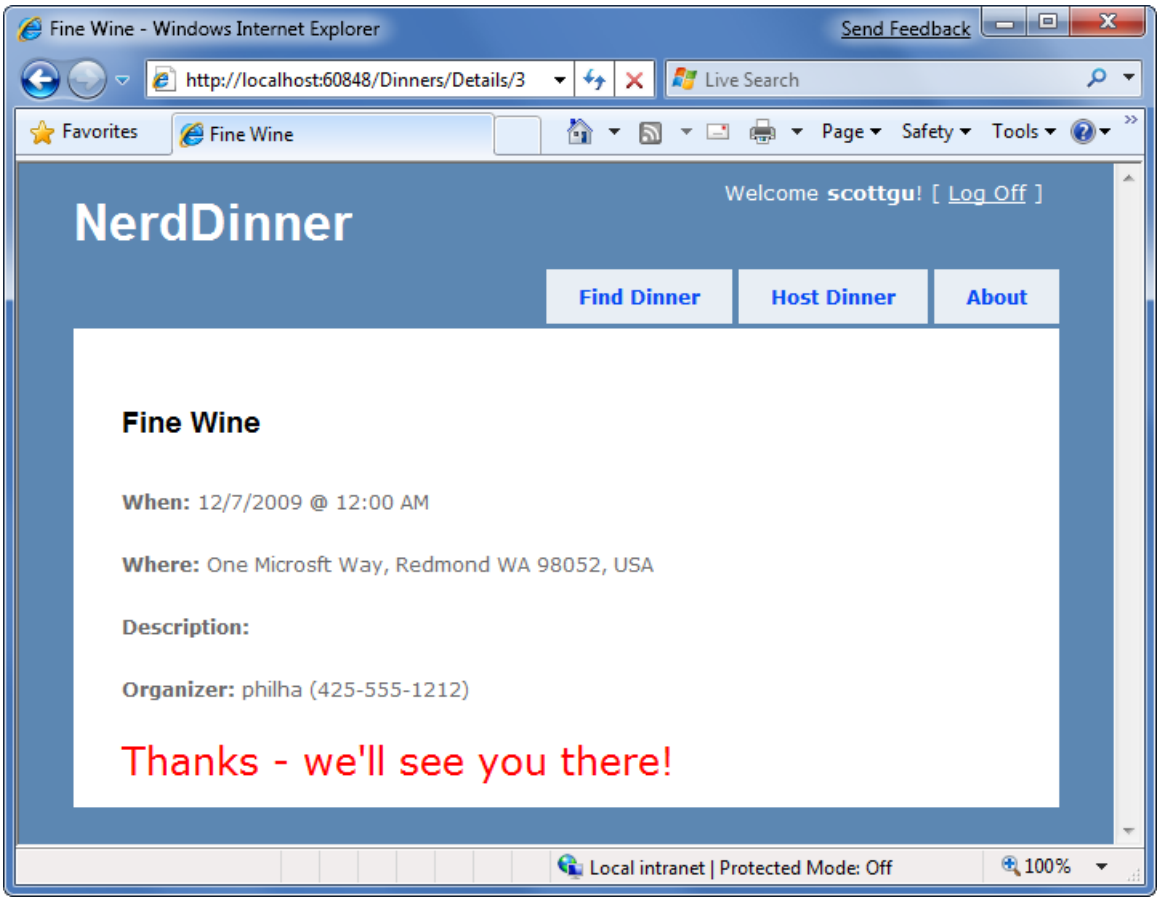
    function AnimateRSVPMessage() {
        $("#rsvpmsg").animate({fontSize: "1.5em"},400);
    }

</script>
```

We can then wire-up this JavaScript function to be called after our AJAX call successfully completes by passing its name to our Ajax.ActionLink() helper method (via the AjaxOptions "OnSuccess" event property):

```
<%= Ajax.ActionLink( "RSVP for this event",
                     "Register", "RSVP",
                     new { id=Model.DinnerID },
                     new AjaxOptions { UpdateTargetId="rsvpmsg",
                                       OnSuccess="AnimateRSVPMessage"}) %>
```

And now when the "RSVP for this event" link is clicked and our AJAX call completes successfully, the content message sent back will animate and grow large:



In addition to providing an "OnSuccess" event, the AjaxOptions object exposes OnBegin, OnFailure, and OnComplete events that you can handle (along with a variety of other properties and useful options).

Cleanup - Refactor out a RSVP Partial View

Our details view template is starting to get a little long, which overtime will make it a little harder to understand. To help improve the code readability, let's finish up by creating a partial view – RSVPStatus.ascx – that encapsulate all of the RSVP view code for our Details page.

We can do this by right-clicking on the \Views\Dinners folder and then choosing the Add->View menu command. We'll have it take a Dinner object as its strongly-typed ViewModel. We can then copy/paste the RSVP content from our Details.aspx view into it.

Once we've done that, let's also create another partial view – EditAndDeleteLinks.ascx - that encapsulates our Edit and Delete link view code. We'll also have it take a Dinner object as its strongly-typed ViewModel, and copy/paste the Edit and Delete logic from our Details.aspx view into it.

Our details view template can then just include two Html.RenderPartial() method calls at the bottom:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent"runat="server">
    <%= Html.Encode(Model.Title) %>
</asp:Content>

<asp:Content ID="details" ContentPlaceHolderID="MainContent" runat="server">

    <div id="dinnerDiv">

        <h2><%=Html.Encode(Model.Title) %></h2>
        <p>
            <strong>When:</strong>
            <%=Model.EventDate.ToShortDateString() %>

            <strong>@</strong>
            <%=Model.EventDate.ToShortTimeString() %>
        </p>
        <p>
            <strong>Where:</strong>
            <%=Html.Encode(Model.Address) %>,
            <%=Html.Encode(Model.Country) %>
        </p>
         <p>
            <strong>Description:</strong>
            <%=Html.Encode(Model.Description) %>
        </p>
        <p>
            <strong>Organizer:</strong>
            <%=Html.Encode(Model.HostedBy) %>
            (<%=Html.Encode(Model.ContactPhone) %>)
        </p>

        <% Html.RenderPartial("RSVPStatus"); %>
        <% Html.RenderPartial("EditAndDeleteLinks"); %>

    </div>

</asp:Content>
```

This makes the code cleaner to read and maintain.

Next Step

Let's now look at how we can use AJAX even further and add interactive mapping support to our application.

PREVIOUS NEXT

Use AJAX to Implement Mapping Scenarios

9/16/2020 • 14 minutes to read • Edit Online

by Microsoft

Download PDF

> This is step 11 of a free "NerdDinner" application tutorial that walks-through how to build a small, but complete, web application using ASP.NET MVC 1.
>
> Step 11 shows how to integrate AJAX mapping support into our NerdDinner application, enabling users who are creating, editing or viewing dinners to see the location of the dinner graphically.
>
> If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

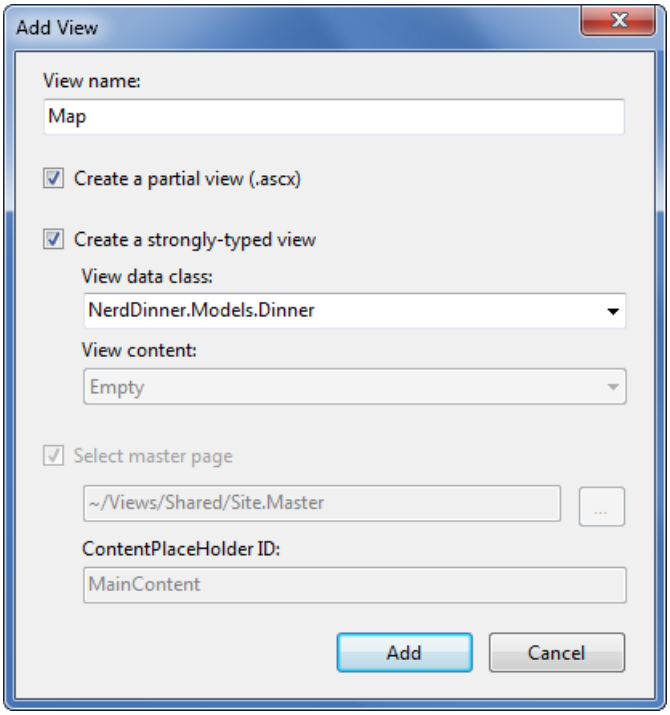
NerdDinner Step 11: Integrating an AJAX Map

We'll now make our application a little more visually exciting by integrating AJAX mapping support. This will enable users who are creating, editing or viewing dinners to see the location of the dinner graphically.

Creating a Map Partial View

We are going to use mapping functionality in several places within our application. To keep our code DRY we'll encapsulate the common map functionality within a single partial template that we can re-use across multiple controller actions and views. We'll name this partial view "map.ascx" and create it within the \Views\Dinners directory.

We can create the map.ascx partial by right-clicking on the \Views\Dinners directory and choosing the Add->View menu command. We'll name the view "Map.ascx", check it as a partial view, and indicate that we are going to pass it a strongly-typed "Dinner" model class:



When we click the "Add" button our partial template will be created. We'll then update the Map.ascx file to have the following content:

```
<script src="http://dev.virtualearth.net/mapcontrol/mapcontrol.ashx?v=6.2" type="text/javascript"></script>
<script src="/Scripts/Map.js" type="text/javascript"></script>

<div id="theMap">
</div>

<script type="text/javascript">

    $(document).ready(function() {
        var latitude = <%=Model.Latitude%>;
        var longitude = <%=Model.Longitude%>;

        if ((latitude == 0) || (longitude == 0))
            LoadMap();
        else
            LoadMap(latitude, longitude, mapLoaded);
    });

   function mapLoaded() {
        var title = "<%=Html.Encode(Model.Title) %>";
        var address = "<%=Html.Encode(Model.Address) %>";

        LoadPin(center, title, address);
        map.SetZoomLevel(14);
    }

</script>
```

The first <script> reference points to the Microsoft Virtual Earth 6.2 mapping library. The second <script> reference points to a map.js file that we will shortly create which will encapsulate our common Javascript mapping logic. The element is the HTML container that Virtual Earth will use to host the map.

We then have an embedded <script> block that contains two JavaScript functions specific to this view. The first function uses jQuery to wire-up a function that executes when the page is ready to run client-side script. It calls a LoadMap() helper function that we'll define within our Map.js script file to load the virtual earth map control. The second function is a callback event handler that adds a pin to the map that identifies a location.

Notice how we are using a server-side <%= %> block within the client-side script block to embed the latitude and longitude of the Dinner we want to map into the JavaScript. This is a useful technique to output dynamic values that can be used by client-side script (without requiring a separate AJAX call back to the server to retrieve the values – which makes it faster). The <%= %> blocks will execute when the view is rendering on the server – and so the output of the HTML will just end up with embedded JavaScript values (for example: var latitude = 47.64312;).

Creating a Map.js utility library

Let's now create the Map.js file that we can use to encapsulate the JavaScript functionality for our map (and implement the LoadMap and LoadPin methods above). We can do this by right-clicking on the \Scripts directory within our project, and then choose the "Add->New Item" menu command, select the JScript item, and name it "Map.js".

Below is the JavaScript code we'll add to the Map.js file that will interact with Virtual Earth to display our map and add locations pins to it for our dinners:

```
var map = null;
var points = [];
var shapes = [];
var center = null;

function LoadMap(latitude, longitude, onMapLoaded) {
    map = new VEMap('theMap');
```

```
    options = new VEMapOptions();
    options.EnableBirdseye = false;

    // Makes the control bar less obtrusize.
    map.SetDashboardSize(VEDashboardSize.Small);

    if (onMapLoaded != null)
        map.onLoadMap = onMapLoaded;

    if (latitude != null && longitude != null) {
        center = new VELatLong(latitude, longitude);
    }

    map.LoadMap(center, null, null, null, null, null, null, options);
}

function LoadPin(LL, name, description) {
    var shape = new VEShape(VEShapeType.Pushpin, LL);

    //Make a nice Pushpin shape with a title and description
    shape.SetTitle("<span class=\"pinTitle\"> " + escape(name) + "</span>");
    if (description !== undefined) {
        shape.SetDescription("<p class=\"pinDetails\">" +
        escape(description) + "</p>");
    }
    map.AddShape(shape);
    points.push(LL);
    shapes.push(shape);
}

function FindAddressOnMap(where) {
    var numberOfResults = 20;
    var setBestMapView = true;
    var showResults = true;

    map.Find("", where, null, null, null,
           numberOfResults, showResults, true, true,
           setBestMapView, callbackForLocation);
}

function callbackForLocation(layer, resultsArray, places,
            hasMore, VEErrorMessage) {

    clearMap();

    if (places == null)
        return;

    //Make a pushpin for each place we find
    $.each(places, function(i, item) {
        description = "";
        if (item.Description !== undefined) {
            description = item.Description;
        }
        var LL = new VELatLong(item.LatLong.Latitude,
                        item.LatLong.Longitude);

        LoadPin(LL, item.Name, description);
    });

    //Make sure all pushpins are visible
    if (points.length > 1) {
        map.SetMapView(points);
    }

    //If we've found exactly one place, that's our address.
    if (points.length === 1) {
        $("#Latitude").val(points[0].Latitude);
        $("#Longitude").val(points[0].Longitude);
```

```
    }
}

function clearMap() {
    map.Clear();
    points = [];
    shapes = [];
}
```

Integrating the Map with Create and Edit Forms

We'll now integrate the Map support with our existing Create and Edit scenarios. The good news is that this is pretty easy to-do, and doesn't require us to change any of our Controller code. Because our Create and Edit views share a common "DinnerForm" partial view to implement the dinner form UI, we can add the map in one place and have both our Create and Edit scenarios use it.

All we need to-do is to open the \Views\Dinners\DinnerForm.ascx partial view and update it to include our new map partial. Below is what the updated DinnerForm will look like once the map is added (note: the HTML form elements are omitted from the code snippet below for brevity):

```
<%= Html.ValidationSummary() %>

<% using (Html.BeginForm()) { %>

    <fieldset>

        <div id="dinnerDiv">
            <p>
               [HTML Form Elements Removed for Brevity]
            </p>
            <p>
               <input type="submit" value="Save"/>
            </p>
        </div>

        <div id="mapDiv">
            <%Html.RenderPartial("Map", Model.Dinner); %>
        </div>

    </fieldset>

    <script type="text/javascript">

        $(document).ready(function() {
            $("#Address").blur(function(evt) {
                $("#Latitude").val("");
                $("#Longitude").val("");

                var address = jQuery.trim($("#Address").val());
                if (address.length < 1)
                    return;

                FindAddressOnMap(address);
            });
        });

    </script>

<% } %>
```

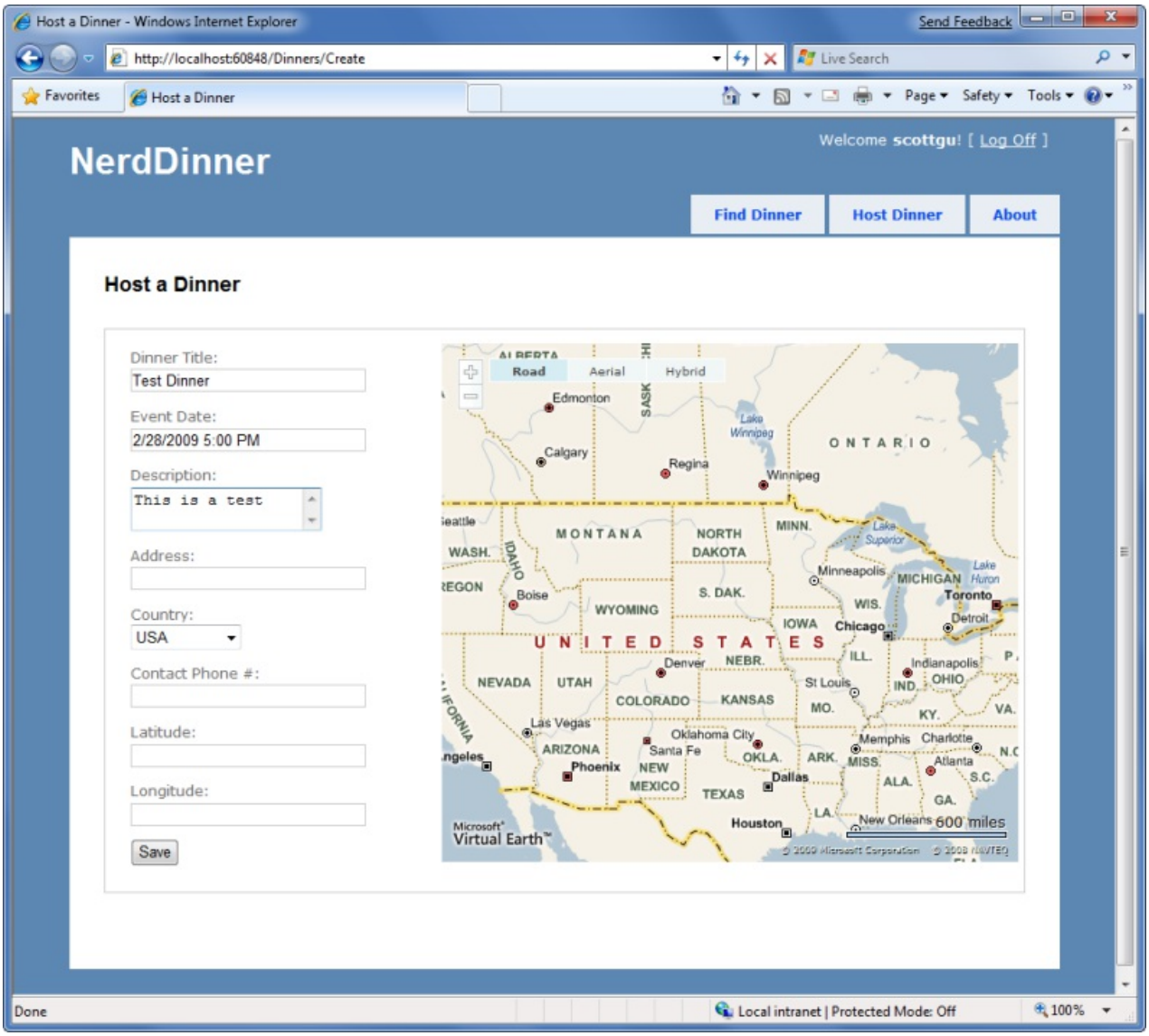
The DinnerForm partial above takes an object of type "DinnerFormViewModel" as its model type (because it needs both a Dinner object, as well as a SelectList to populate the dropdownlist of countries). Our Map partial just needs an object of type "Dinner" as its model type, and so when we render the map partial we are passing just the

Dinner sub-property of DinnerFormViewModel to it:

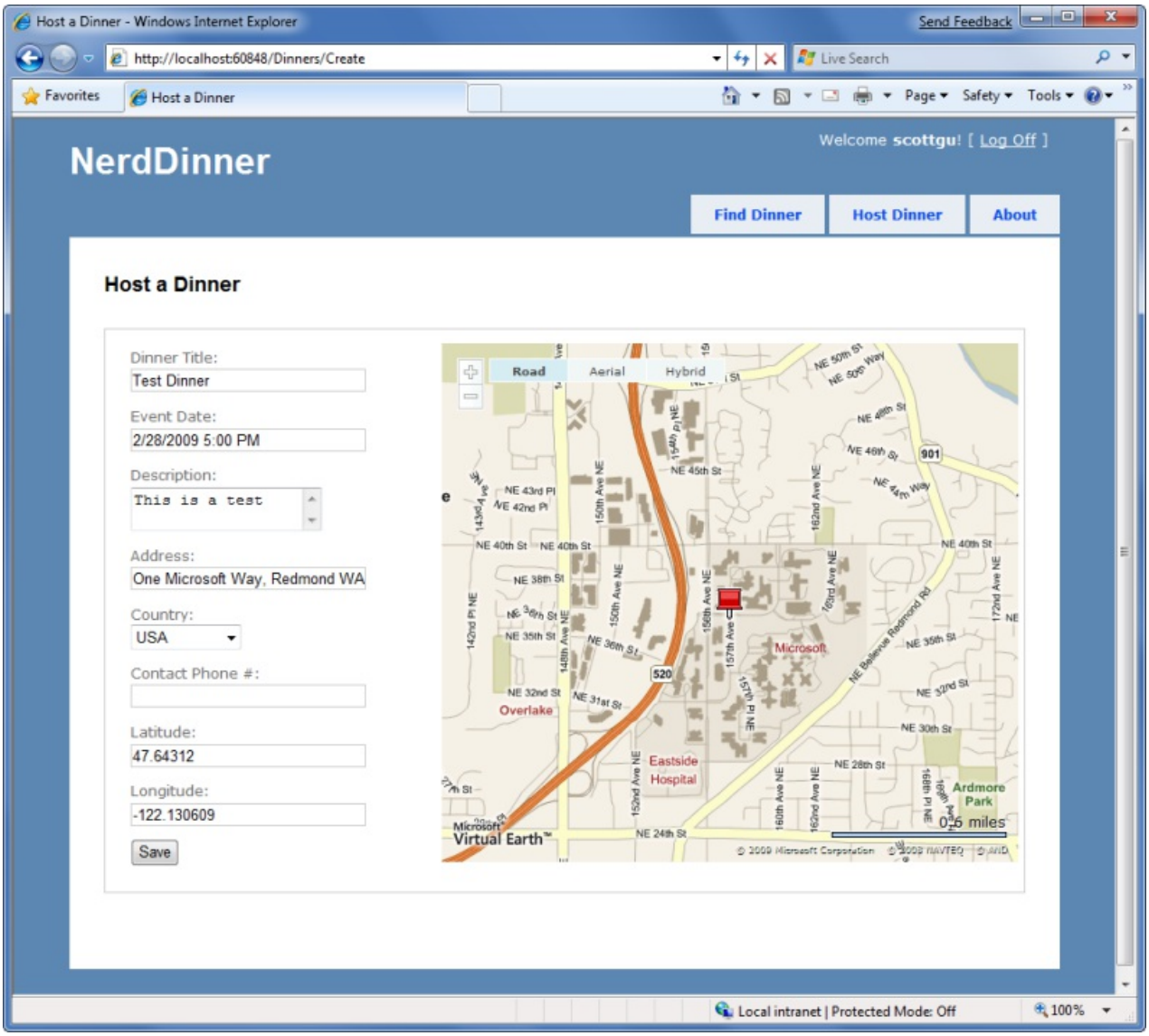
```
<% Html.RenderPartial("Map", Model.Dinner); %>
```

The JavaScript function we've added to the partial uses jQuery to attach a "blur" event to the "Address" HTML textbox. You've probably heard of "focus" events that fire when a user clicks or tabs into a textbox. The opposite is a "blur" event that fires when a user exits a textbox. The above event handler clears the latitude and longitude textbox values when this happens, and then plots the new address location on our map. A callback event handler that we defined within the map.js file will then update the longitude and latitude textboxes on our form using values returned by virtual earth based on the address we gave it.

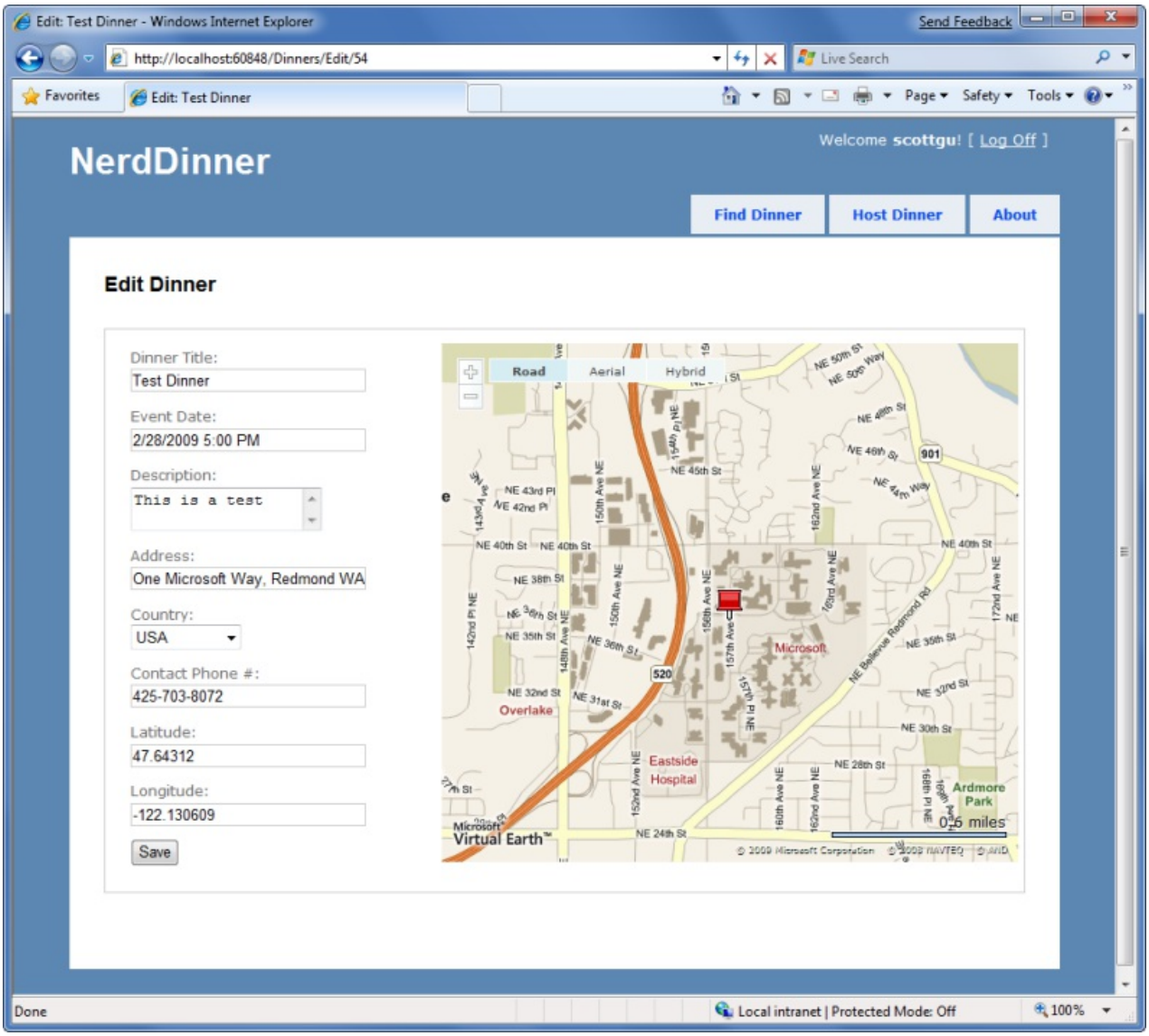
And now when we run our application again and click the "Host Dinner" tab we'll see a default map displayed along with our standard Dinner form elements:



When we type in an address, and then tab away, the map will dynamically update to display the location, and our event handler will populate the latitude/longitude textboxes with the location values:



If we save the new dinner and then open it again for editing, we'll find that the map location is displayed when the page loads:

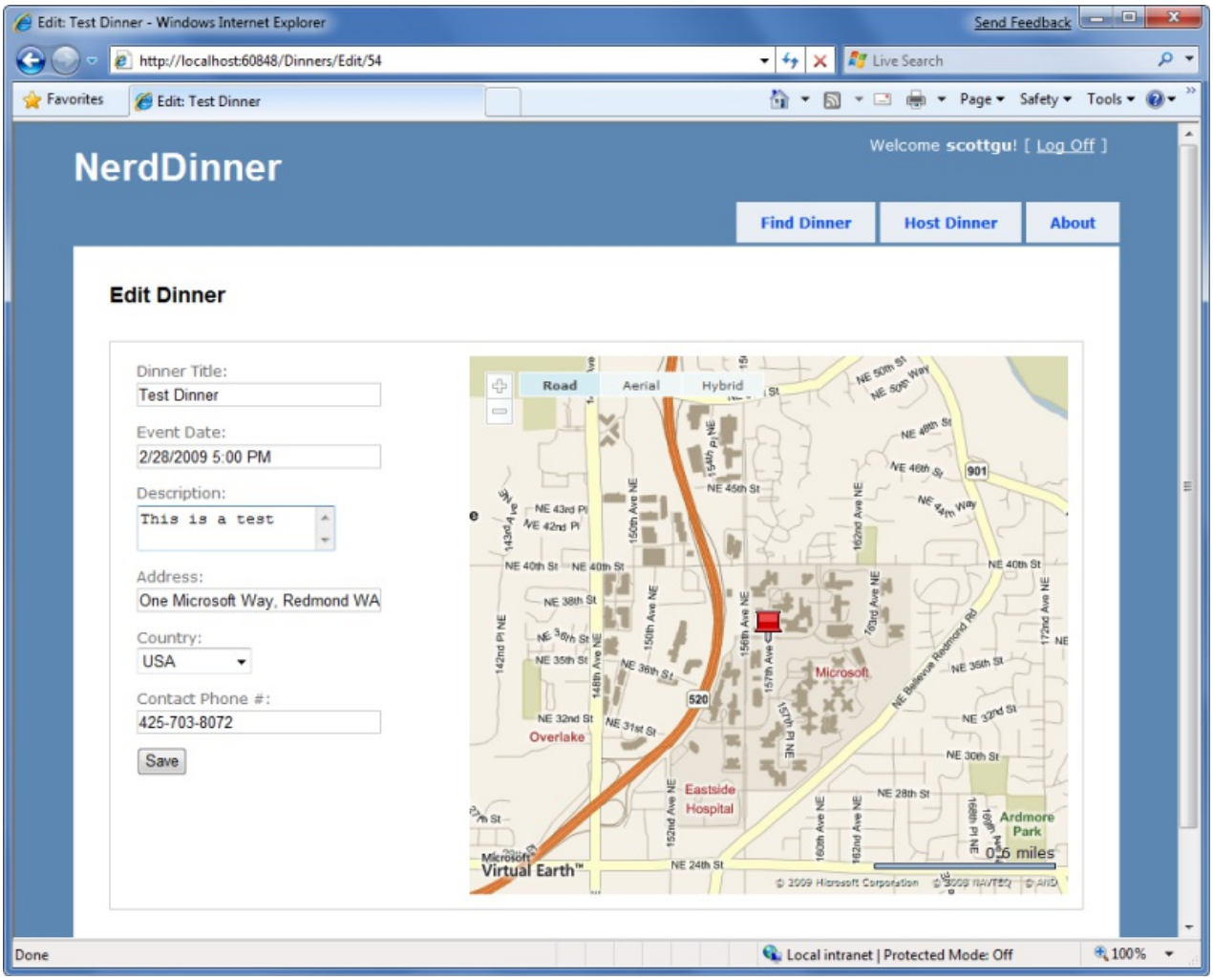


Every time the address field is changed, the map and the latitude/longitude coordinates will update.

Now that the map displays the Dinner location, we can also change the Latitude and Longitude form fields from being visible textboxes to instead be hidden elements (since the map is automatically updating them each time an address is entered). To-do this we'll switch from using the Html.TextBox() HTML helper to using the Html.Hidden() helper method:

```
<p>
    <%= Html.Hidden("Latitude", Model.Dinner.Latitude)%>
    <%= Html.Hidden("Longitude", Model.Dinner.Longitude)%>
</p>
```

And now our forms are a little more user-friendly and avoid displaying the raw latitude/longitude (while still storing them with each Dinner in the database):



Integrating the Map with the Details View

Now that we have the map integrated with our Create and Edit scenarios, let's also integrate it with our Details scenario. All we need to-do is to call <% Html.RenderPartial("map"); %> within the Details view.

Below is what the source code to the complete Details view (with map integration) looks like:

```
<asp:Content ID="Title" ContentPlaceHolderID="TitleContent"runat="server">
    <%= Html.Encode(Model.Title) %>
</asp:Content>

<asp:Content ID="details" ContentPlaceHolderID="MainContent" runat="server">

    <div id="dinnerDiv">

        <h2><%=Html.Encode(Model.Title) %></h2>
        <p>
            <strong>When:</strong>
            <%=Model.EventDate.ToShortDateString() %>

            <strong>@</strong>
            <%=Model.EventDate.ToShortTimeString() %>
        </p>
        <p>
            <strong>Where:</strong>
            <%=Html.Encode(Model.Address) %>,
            <%=Html.Encode(Model.Country) %>
        </p>
         <p>
            <strong>Description:</strong>
            <%=Html.Encode(Model.Description) %>
        </p>
        <p>
            <strong>Organizer:</strong>
            <%=Html.Encode(Model.HostedBy) %>
            (<%=Html.Encode(Model.ContactPhone) %>)
        </p>

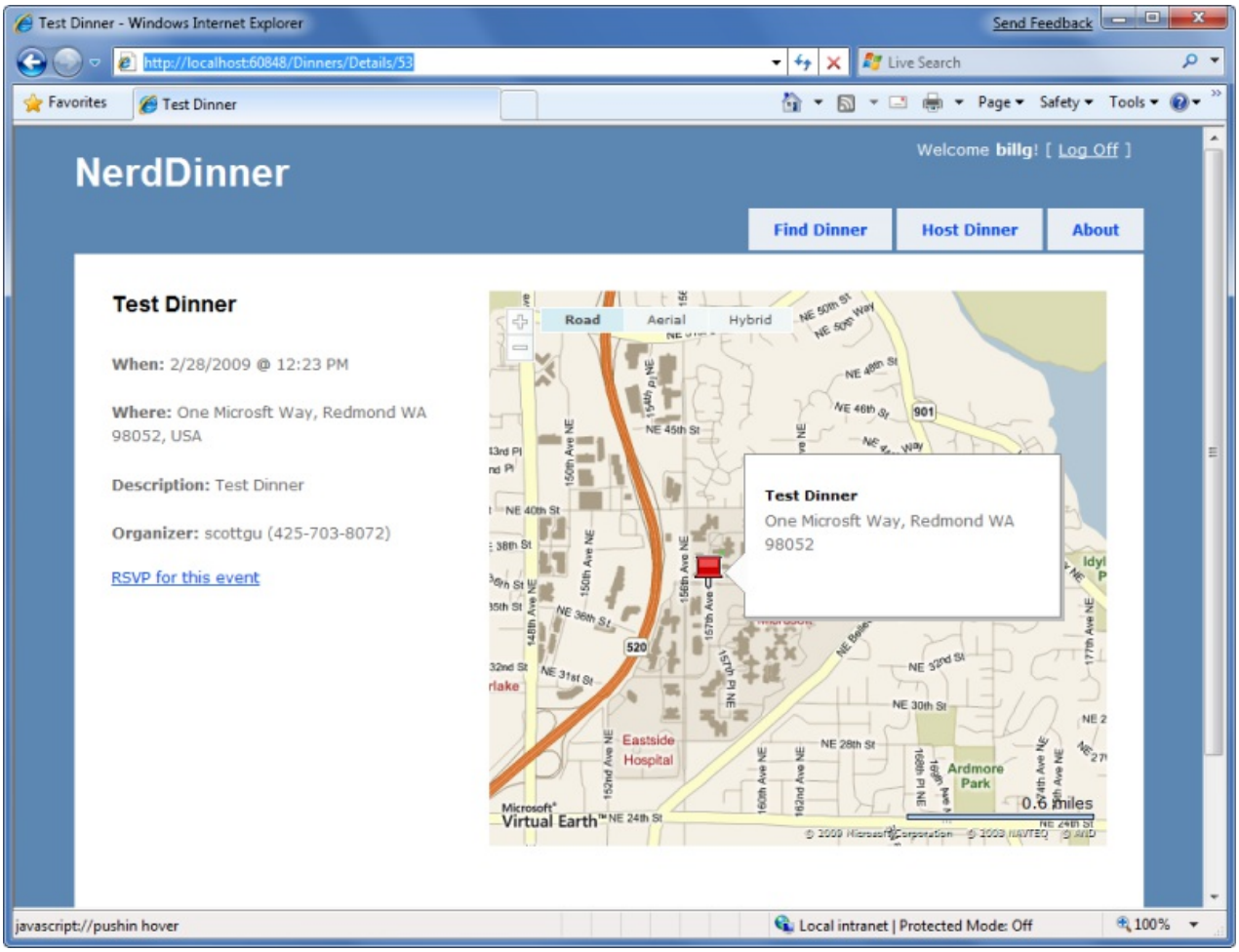
        <%Html.RenderPartial("RSVPStatus"); %>
        <%Html.RenderPartial("EditAndDeleteLinks"); %>

    </div>

    <div id="mapDiv">
        <%Html.RenderPartial("map"); %>
    </div>

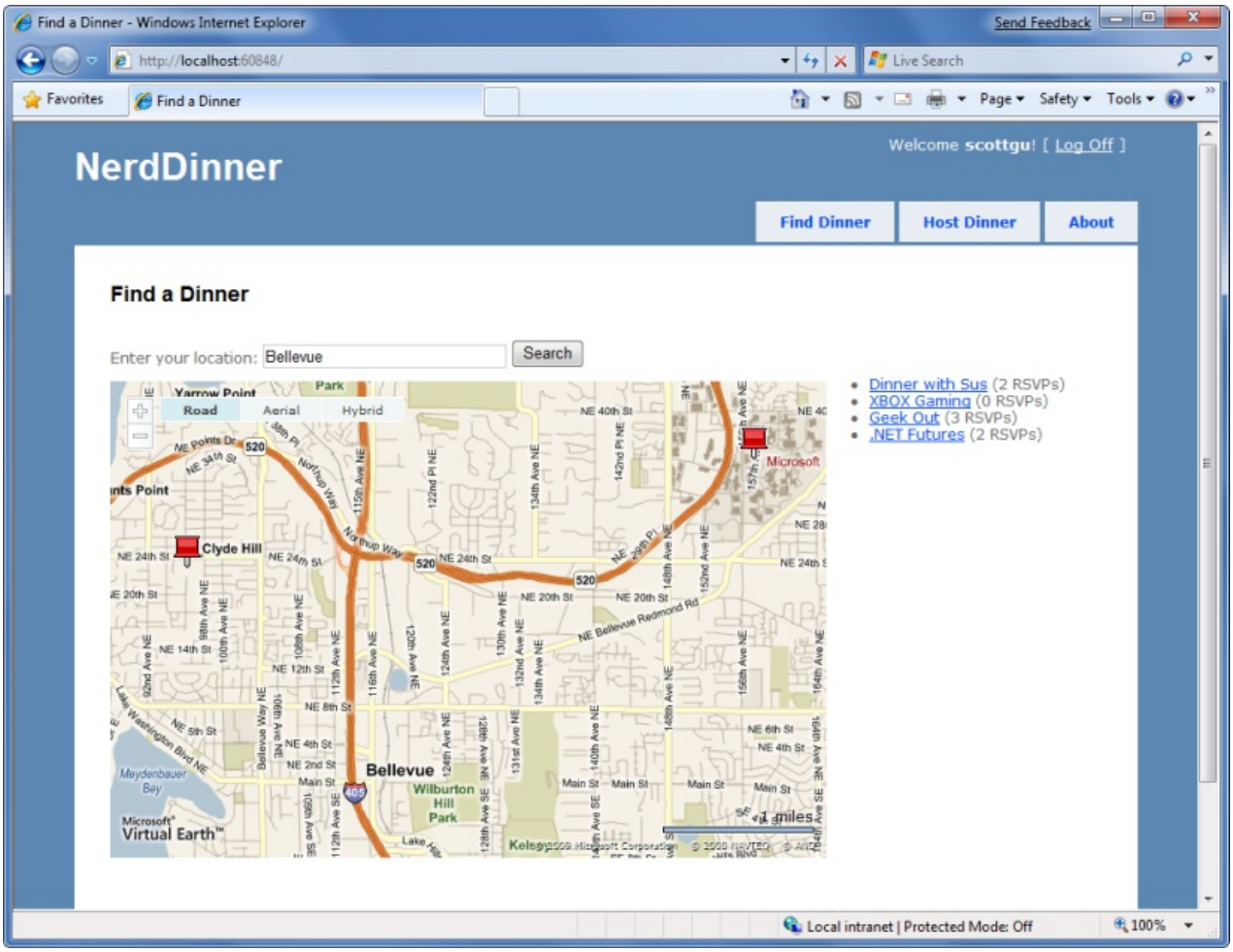
</asp:Content>
```

And now when a user navigates to a /Dinners/Details/[id] URL they'll see details about the dinner, the location of the dinner on the map (complete with a push-pin that when hovered over displays the title of the dinner and the address of it), and have an AJAX link to RSVP for it:



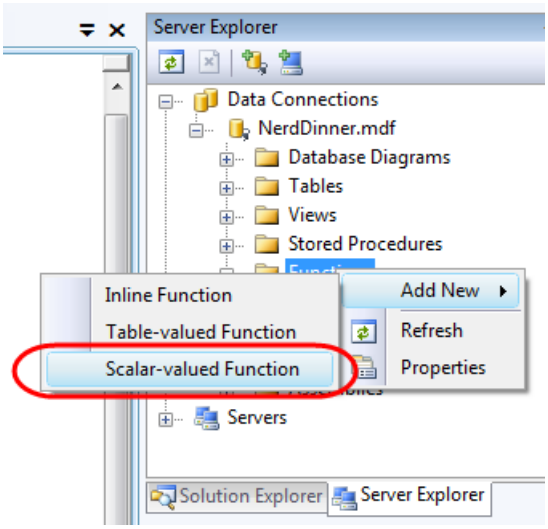
Implementing Location Search in our Database and Repository

To finish off our AJAX implementation, let's add a Map to the home page of the application that allows users to graphically search for dinners near them.



We'll begin by implementing support within our database and data repository layer to efficiently perform a location-based radius search for Dinners. We could use the new geospatial features of SQL 2008 to implement this, or alternatively we can use a SQL function approach that Gary Dryden discussed in article here: http://www.codeproject.com/KB/cs/distancebetweenlocations.aspx and Rob Conery blogged about using with LINQ to SQL here: http://blog.wekeroad.com/2007/08/30/linq-and-geocoding/

To implement this technique, we will open the "Server Explorer" within Visual Studio, select the NerdDinner database, and then right-click on the "functions" sub-node under it and choose to create a new "Scalar-valued function":



We'll then paste in the following DistanceBetween function:

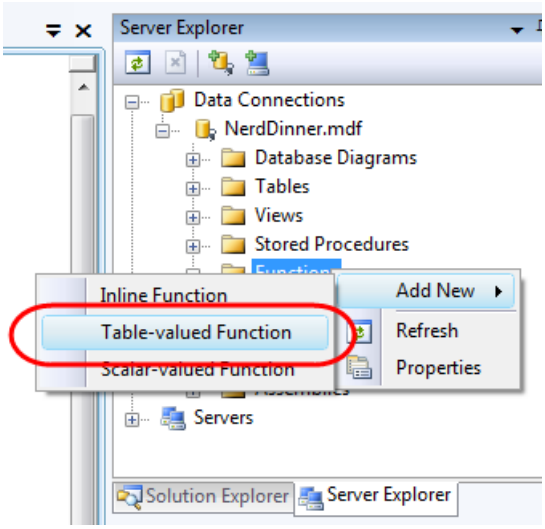
```
CREATE FUNCTION [dbo].[DistanceBetween](@Lat1 as real,
                @Long1 as real, @Lat2 as real, @Long2 as real)
RETURNS real
AS
BEGIN

DECLARE @dLat1InRad as float(53);
SET @dLat1InRad = @Lat1 * (PI()/180.0);
DECLARE @dLong1InRad as float(53);
SET @dLong1InRad = @Long1 * (PI()/180.0);
DECLARE @dLat2InRad as float(53);
SET @dLat2InRad = @Lat2 * (PI()/180.0);
DECLARE @dLong2InRad as float(53);
SET @dLong2InRad = @Long2 * (PI()/180.0);

DECLARE @dLongitude as float(53);
SET @dLongitude = @dLong2InRad - @dLong1InRad;
DECLARE @dLatitude as float(53);
SET @dLatitude = @dLat2InRad - @dLat1InRad;
/* Intermediate result a. */
DECLARE @a as float(53);
SET @a = SQUARE (SIN (@dLatitude / 2.0)) + COS (@dLat1InRad)
                 * COS (@dLat2InRad)
                 * SQUARE(SIN (@dLongitude / 2.0));
/* Intermediate result c (great circle distance in Radians). */
DECLARE @c as real;
SET @c = 2.0 * ATN2 (SQRT (@a), SQRT (1.0 - @a));
DECLARE @kEarthRadius as real;
/* SET kEarthRadius = 3956.0 miles */
SET @kEarthRadius = 6376.5;        /* kms */

DECLARE @dDistance as real;
SET @dDistance = @kEarthRadius * @c;
return (@dDistance);
END
```

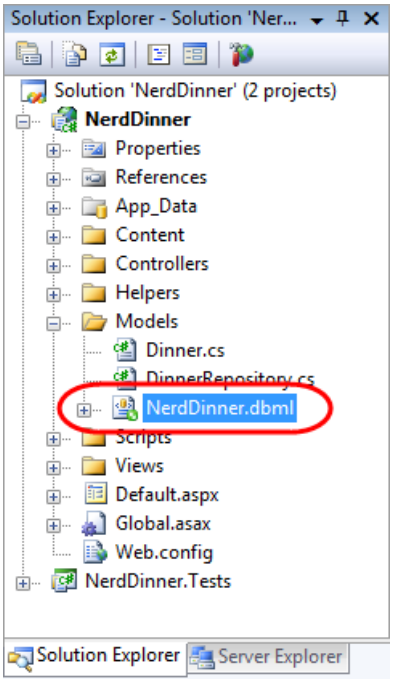
We'll then create a new table-valued function in SQL Server that we'll call "NearestDinners":



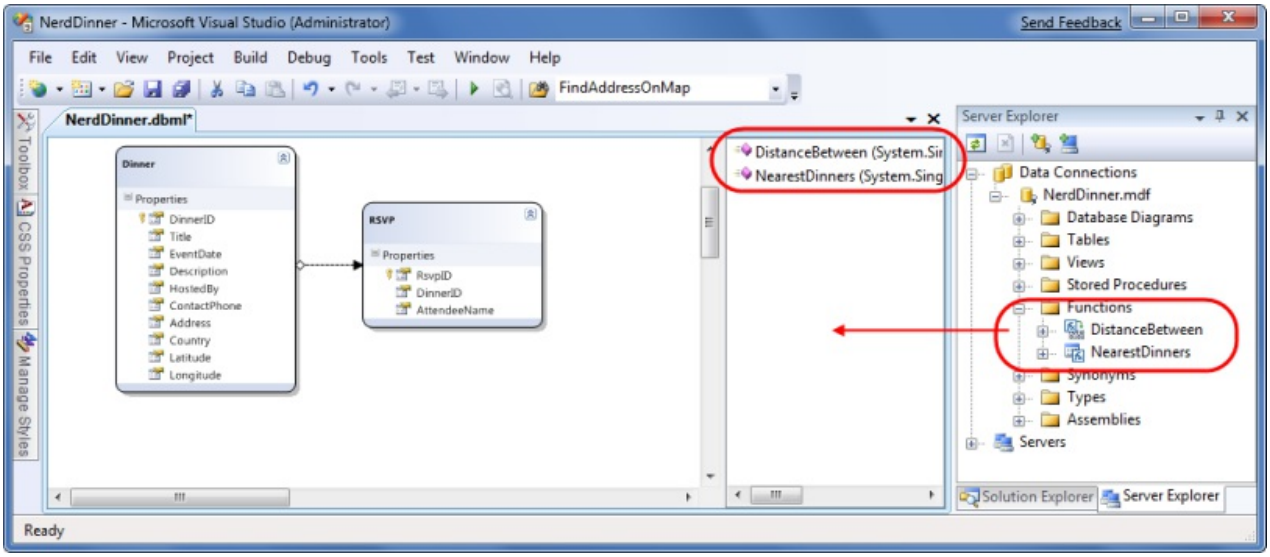
This "NearestDinners" table function uses the DistanceBetween helper function to return all Dinners within 100 miles of the latitude and longitude we supply it:

```
CREATE FUNCTION [dbo].[NearestDinners]
      (
      @lat real,
      @long real
      )
RETURNS  TABLE
AS
      RETURN
      SELECT Dinners.DinnerID
      FROM   Dinners
      WHERE  dbo.DistanceBetween(@lat, @long, Latitude, Longitude) <100
```

To call this function, we'll first open up the LINQ to SQL designer by double-clicking on the NerdDinner.dbml file within our \Models directory:



We'll then drag the NearestDinners and DistanceBetween functions onto the LINQ to SQL designer, which will cause them to be added as methods on our LINQ to SQL NerdDinnerDataContext class:



We can then expose a "FindByLocation" query method on our DinnerRepository class that uses the NearestDinner function to return upcoming Dinners that are within 100 miles of the specified location:

```
public IQueryable<Dinner> FindByLocation(float latitude, float longitude) {

   var dinners = from dinner in FindUpcomingDinners()
                 join i in db.NearestDinners(latitude, longitude)
                 on dinner.DinnerID equals i.DinnerID
                 select dinner;

   return dinners;
}
```

Implementing a JSON-based AJAX Search Action Method

We'll now implement a controller action method that takes advantage of the new FindByLocation() repository method to return back a list of Dinner data that can be used to populate a map. We'll have this action method return back the Dinner data in a JSON (JavaScript Object Notation) format so that it can be easily manipulated using JavaScript on the client.

To implement this, we'll create a new "SearchController" class by right-clicking on the \Controllers directory and choosing the Add->Controller menu command. We'll then implement a "SearchByLocation" action method within the new SearchController class like below:

```
public class JsonDinner {
    public int      DinnerID    { get; set; }
    public string   Title       { get; set; }
    public double   Latitude    { get; set; }
    public double   Longitude   { get; set; }
    public string   Description { get; set; }
    public int      RSVPCount   { get; set; }
}

public class SearchController : Controller {

    DinnerRepository dinnerRepository = new DinnerRepository();

    //
    // AJAX: /Search/SearchByLocation

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult SearchByLocation(float longitude, float latitude) {

        var dinners = dinnerRepository.FindByLocation(latitude,longitude);

        var jsonDinners = from dinner in dinners
                          select new JsonDinner {
                              DinnerID = dinner.DinnerID,
                              Latitude = dinner.Latitude,
                              Longitude = dinner.Longitude,
                              Title = dinner.Title,
                              Description = dinner.Description,
                              RSVPCount = dinner.RSVPs.Count
                          };

        return Json(jsonDinners.ToList());
    }
}
```

The SearchController's SearchByLocation action method internally calls the FindByLocation method on DinnerRepository to get a list of nearby dinners. Rather than return the Dinner objects directly to the client, though, it instead returns JsonDinner objects. The JsonDinner class exposes a subset of Dinner properties (for example: for security reasons it doesn't disclose the names of the people who have RSVP'd for a dinner). It also includes an RSVPCount property that doesn't exist on Dinner– and which is dynamically calculated by counting the number of RSVP objects associated with a particular dinner.

We are then using the Json() helper method on the Controller base class to return the sequence of dinners using a JSON-based wire format. JSON is a standard text format for representing simple data-structures. Below is an example of what a JSON-formatted list of two JsonDinner objects looks like when returned from our action method:

```
[{"DinnerID":53,"Title":"Dinner with the Family","Latitude":47.64312,"Longitude":-
122.130609,"Description":"Fun dinner","RSVPCount":2},
{"DinnerID":54,"Title":"Another Dinner","Latitude":47.632546,"Longitude":-122.21201,"Description":"Dinner with
Friends","RSVPCount":3}]
```

Calling the JSON-based AJAX method using jQuery

We are now ready to update the home page of the NerdDinner application to use the SearchController's SearchByLocation action method. To-do this, we'll open the /Views/Home/Index.aspx view template and update it to have a textbox, search button, our map, and a element named dinnerList:

```
<h2>Find a Dinner</h2>

<div id="mapDivLeft">

    <div id="searchBox">
        Enter your location: <%=Html.TextBox("Location") %>
        <input id="search" type="submit" value="Search"/>
    </div>

    <div id="theMap">
    </div>

</div>

<div id="mapDivRight">
    <div id="dinnerList"></div>
</div>
```

We can then add two JavaScript functions to the page:

```
<script type="text/javascript">

    $(document).ready(function() {
        LoadMap();
    });

    $("#search").click(function(evt) {
        var where = jQuery.trim($("#Location").val());
        if (where.length < 1)
            return;

        FindDinnersGivenLocation(where);
    });

</script>
```

The first JavaScript function loads the map when the page first loads. The second JavaScript function wires up a JavaScript click event handler on the search button. When the button is pressed it calls the FindDinnersGivenLocation() JavaScript function which we'll add to our Map.js file:

```
function FindDinnersGivenLocation(where) {
    map.Find("", where, null, null, null, null, null, false,
       null, null, callbackUpdateMapDinners);
}
```

This FindDinnersGivenLocation() function calls map.Find() on the Virtual Earth Control to center it on the entered location. When the virtual earth map service returns, the map.Find() method invokes the callbackUpdateMapDinners callback method we passed it as the final argument.

The callbackUpdateMapDinners() method is where the real work is done. It uses jQuery's $.post() helper method to perform an AJAX call to our SearchController's SearchByLocation() action method – passing it the latitude and longitude of the newly centered map. It defines an inline function that will be called when the $.post() helper method completes, and the JSON-formatted dinner results returned from the SearchByLocation() action method will be passed it using a variable called "dinners". It then does a foreach over each returned dinner, and uses the dinner's latitude and longitude and other properties to add a new pin on the map. It also adds a dinner entry to the HTML list of dinners to the right of the map. It then wires-up a hover event for both the pushpins and the HTML list so that details about the dinner are displayed when a user hovers over them:

```
function callbackUpdateMapDinners(layer, resultsArray, places, hasMore, VEErrorMessage) {

    $("#dinnerList").empty();
    clearMap();
    var center = map.GetCenter();

    $.post("/Search/SearchByLocation", { latitude: center.Latitude,
                                         longitude: center.Longitude },
    function(dinners) {
        $.each(dinners, function(i, dinner) {

            var LL = new VELatLong(dinner.Latitude,
                                   dinner.Longitude, 0, null);

            var RsvpMessage = "";

            if (dinner.RSVPCount == 1)
                RsvpMessage = "" + dinner.RSVPCount + "RSVP";
            else
                RsvpMessage = "" + dinner.RSVPCount + "RSVPs";

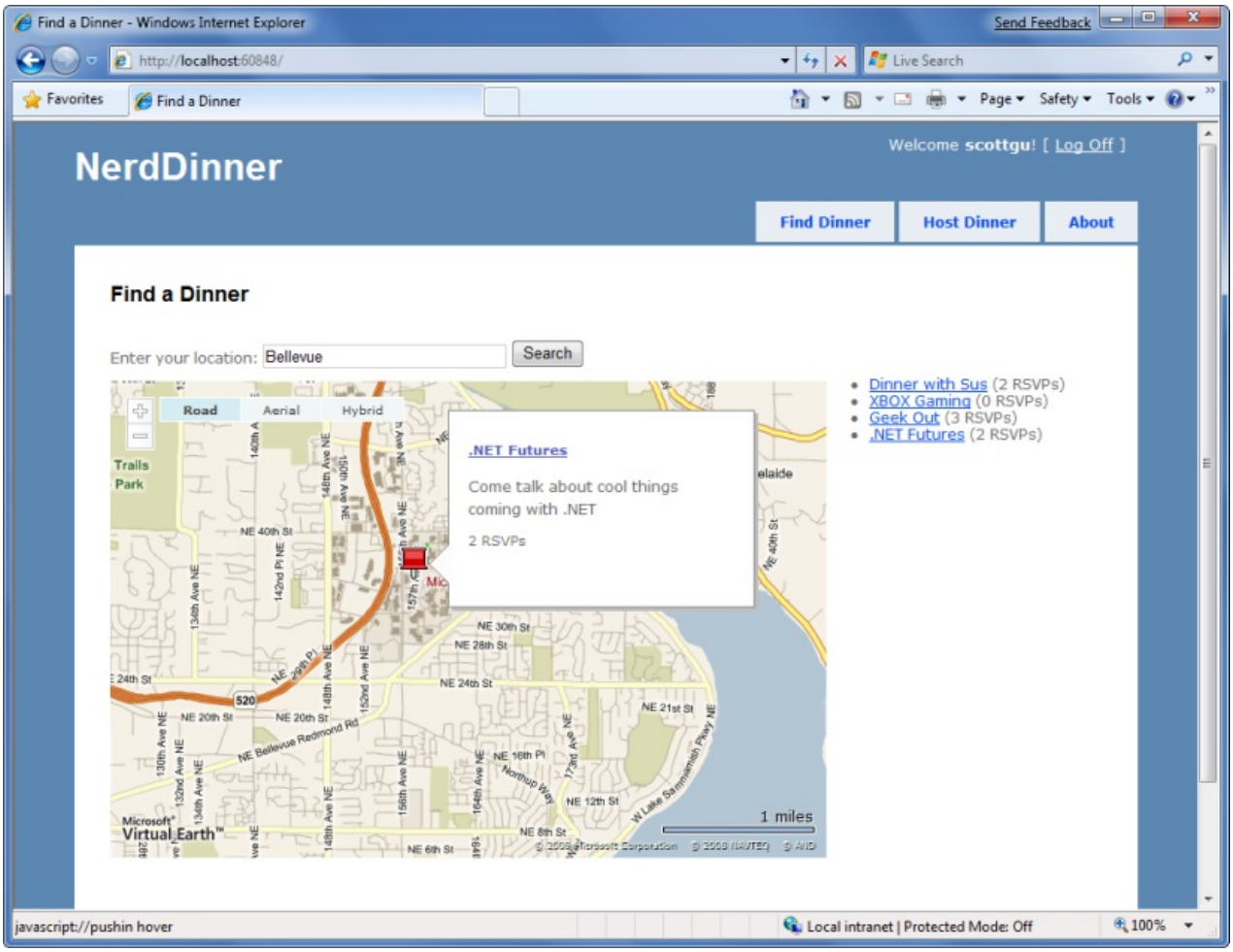
            // Add Pin to Map
            LoadPin(LL, '<a href="/Dinners/Details/' + dinner.DinnerID + '">'
                        + dinner.Title + '</a>',
                        "<p>" + dinner.Description + "</p>" + RsvpMessage);

            //Add a dinner to the <ul> dinnerList on the right
            $('#dinnerList').append($('<li/>')
                            .attr("class", "dinnerItem")
                            .append($('<a/>').attr("href",
                                      "/Dinners/Details/" + dinner.DinnerID)
                            .html(dinner.Title))
                            .append(" ("+RsvpMessage+")"));
        });

        // Adjust zoom to display all the pins we just added.
        map.SetMapView(points);

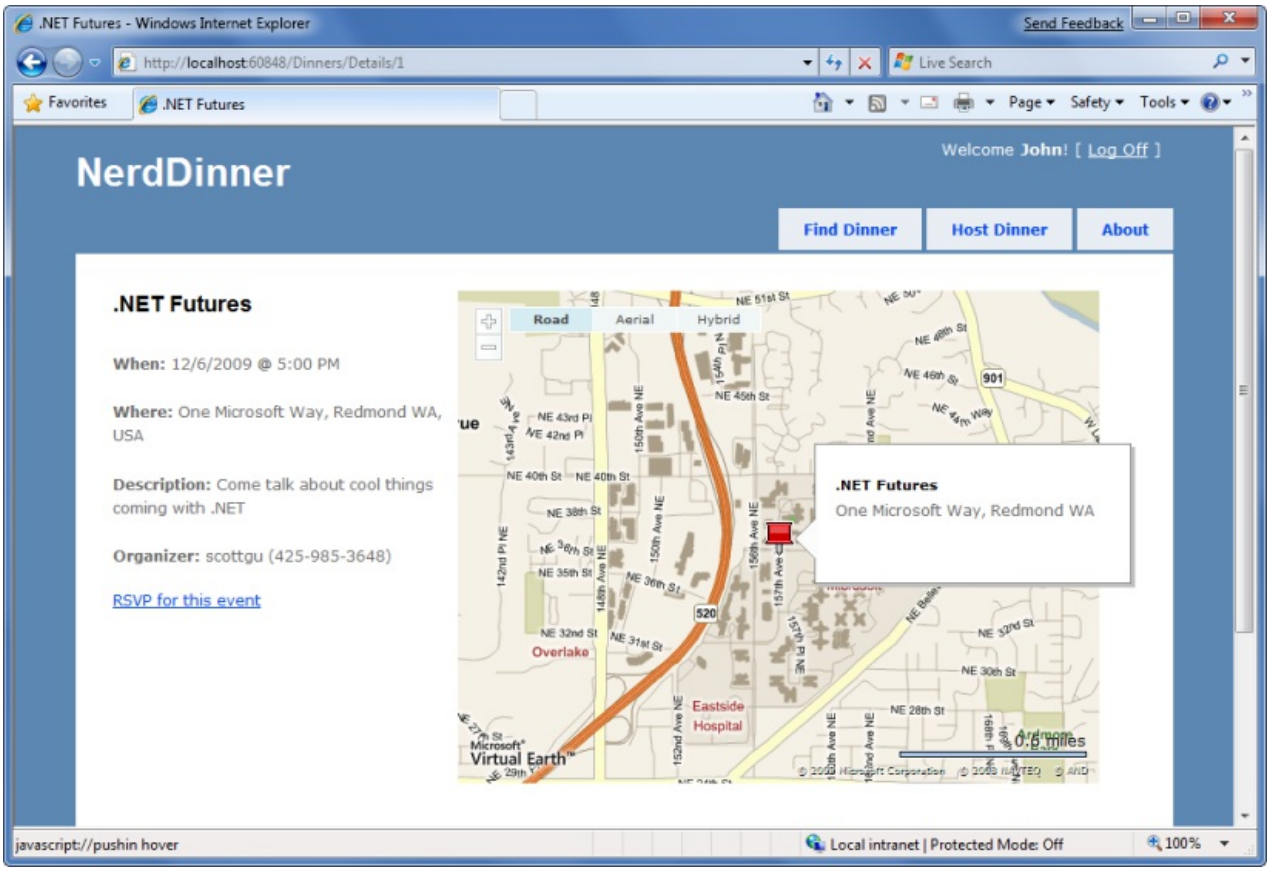
        // Display the event's pin-bubble on hover.
        $(".dinnerItem").each(function(i, dinner) {
            $(dinner).hover(
                function() { map.ShowInfoBox(shapes[i]); },
                function() { map.HideInfoBox(shapes[i]); }
            );
        });
    }, "json");
```

And now when we run the application and visit the home-page we'll be presented with a map. When we enter the name of a city the map will display the upcoming dinners near it:



Hovering over a dinner will display details about it.

Clicking the Dinner title either in the bubble or on the right-hand side in the HTML list will navigate us to the dinner – which we can then optionally RSVP for:



Next Step

We've now implemented all the application functionality of our NerdDinner application. Let's now look at how we can enable automated unit testing of it.

PREVIOUS NEXT

Enable Automated Unit Testing

4/10/2020 • 21 minutes to read • Edit Online

by Microsoft

Download PDF

> This is step 12 of a free "NerdDinner" application tutorial that walks-through how to build a small, but complete, web application using ASP.NET MVC 1.
>
> Step 12 shows how to develop a suite of automated unit tests that verify our NerdDinner functionality, and which will give us the confidence to make changes and improvements to the application in the future.
>
> If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

NerdDinner Step 12: Unit Testing

Let's develop a suite of automated unit tests that verify our NerdDinner functionality, and which will give us the confidence to make changes and improvements to the application in the future.

Why Unit Test?

On the drive into work one morning you have a sudden flash of inspiration about an application you are working on. You realize there is a change you can implement that will make the application dramatically better. It might be a refactoring that cleans up the code, adds a new feature, or fixes a bug.

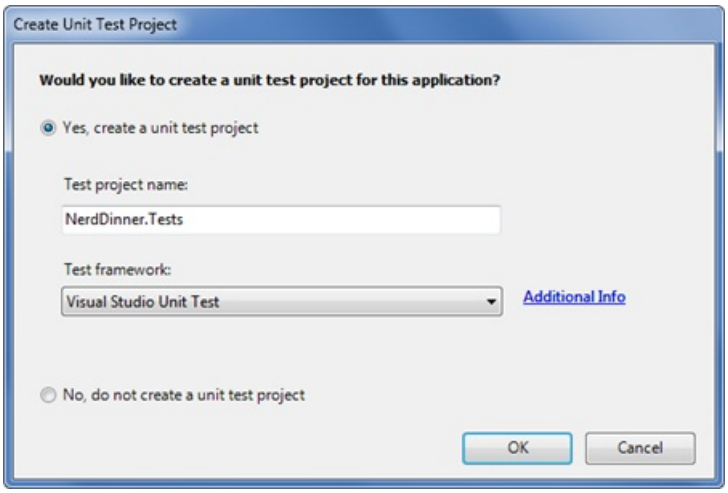
The question that confronts you when you arrive at your computer is – "how safe is it to make this improvement?" What if making the change has side effects or breaks something? The change might be simple and only take a few minutes to implement, but what if it takes hours to manually test out all of the application scenarios? What if you forget to cover a scenario and a broken application goes into production? Is making this improvement really worth all the effort?

Automated unit tests can provide a safety net that enables you to continually enhance your applications, and avoid being afraid of the code you are working on. Having automated tests that quickly verify functionality enables you to code with confidence – and empower you to make improvements you might otherwise not have felt comfortable doing. They also help create solutions that are more maintainable and have a longer lifetime - which leads to a much higher return on investment.

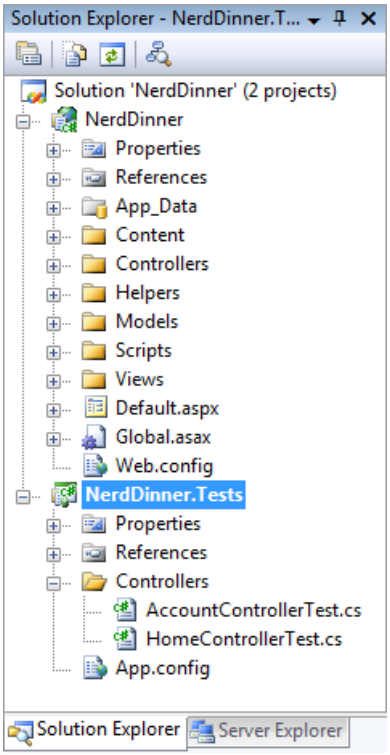
The ASP.NET MVC Framework makes it easy and natural to unit test application functionality. It also enables a Test Driven Development (TDD) workflow that enables test-first based development.

NerdDinner.Tests Project

When we created our NerdDinner application at the beginning of this tutorial, we were prompted with a dialog asking whether we wanted to create a unit test project to go along with the application project:



We kept the "Yes, create a unit test project" radio button selected – which resulted in a "NerdDinner.Tests" project being added to our solution:



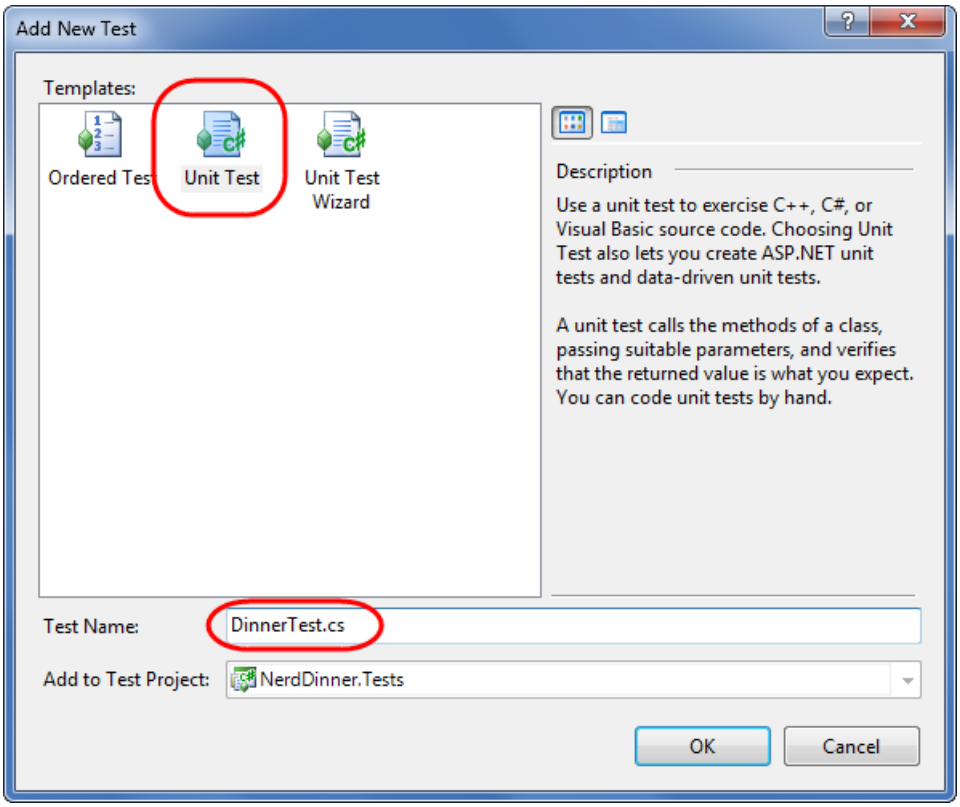
The NerdDinner.Tests project references the NerdDinner application project assembly, and enables us to easily add automated tests to it that verify the application functionality.

Creating Unit Tests for our Dinner Model Class

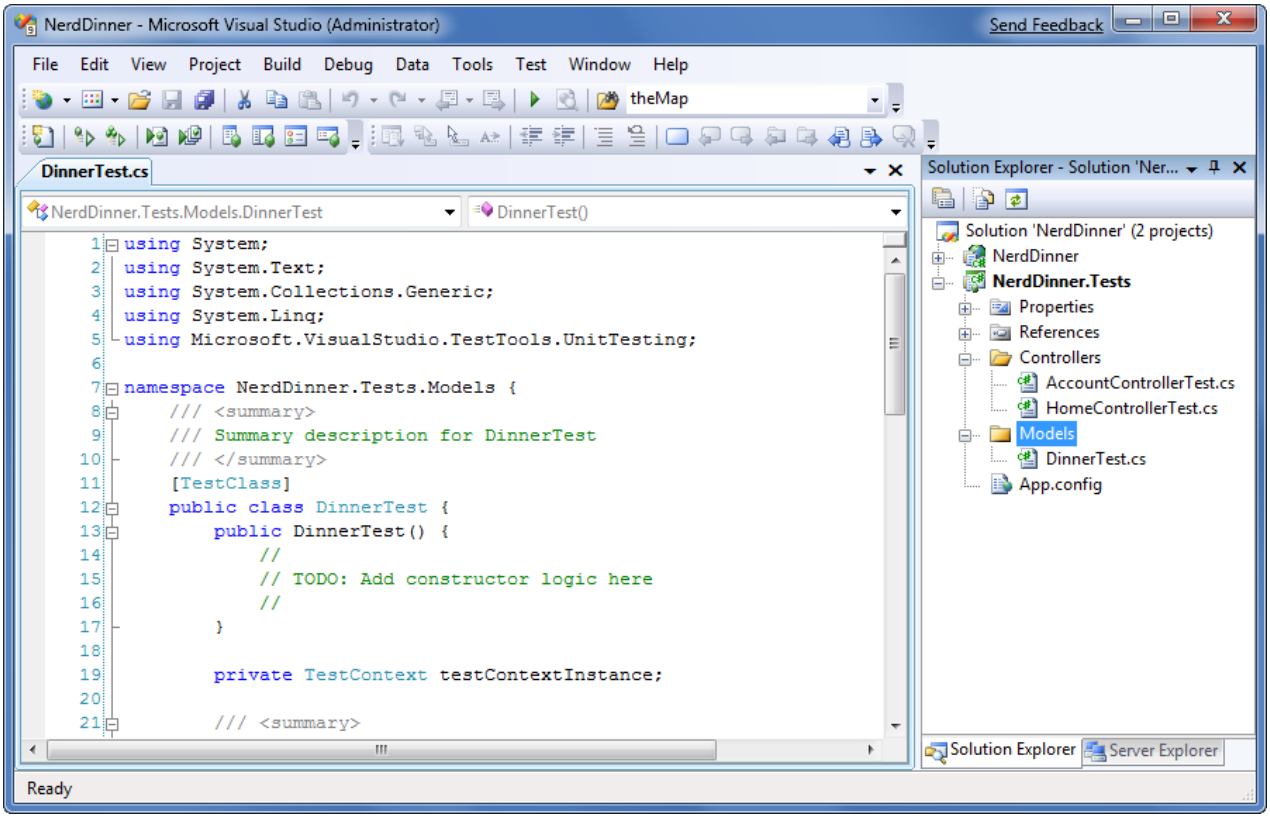
Let's add some tests to our NerdDinner.Tests project that verify the Dinner class we created when we built our model layer.

We'll start by creating a new folder within our test project called "Models" where we'll place our model-related tests. We'll then right-click on the folder and choose the **Add->New Test** menu command. This will bring up the "Add New Test" dialog.

We'll choose to create a "Unit Test" and name it "DinnerTest.cs":



When we click the "ok" button Visual Studio will add (and open) a DinnerTest.cs file to the project:



The default Visual Studio unit test template has a bunch of boiler-plate code within it that I find a little messy. Let's clean it up to just contain the code below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using NerdDinner.Models;

namespace NerdDinner.Tests.Models {

    [TestClass]
    public class DinnerTest {

    }
}
```

The [TestClass] attribute on the DinnerTest class above identifies it as a class that will contain tests, as well as optional test initialization and teardown code. We can define tests within it by adding public methods that have a [TestMethod] attribute on them.

Below are the first of two tests we'll add that exercise our Dinner class. The first test verifies that our Dinner is invalid if a new Dinner is created without all properties being set correctly. The second test verifies that our Dinner is valid when a Dinner has all properties set with valid values:

```
[TestClass]
public class DinnerTest {

    [TestMethod]
    public void Dinner_Should_Not_Be_Valid_When_Some_Properties_Incorrect() {

        //Arrange
        Dinner dinner = new Dinner() {
            Title = "Test title",
            Country = "USA",
            ContactPhone = "BOGUS"
        };

        // Act
        bool isValid = dinner.IsValid;

        //Assert
        Assert.IsFalse(isValid);
    }

    [TestMethod]
    public void Dinner_Should_Be_Valid_When_All_Properties_Correct() {

        //Arrange
        Dinner dinner = new Dinner {
            Title = "Test title",
            Description = "Some description",
            EventDate = DateTime.Now,
            HostedBy = "ScottGu",
            Address = "One Microsoft Way",
            Country = "USA",
            ContactPhone = "425-703-8072",
            Latitude = 93,
            Longitude = -92,
        };

        // Act
        bool isValid = dinner.IsValid;

        //Assert
        Assert.IsTrue(isValid);
    }
}
```

You'll notice above that our test names are very explicit (and somewhat verbose). We are doing this because we might end up creating hundreds or thousands of small tests, and we want to make it easy to quickly determine the intent and behavior of each of them (especially when we are looking through a list of failures in a test runner). The test names should be named after the functionality they are testing. Above we are using a "Noun_Should_Verb" naming pattern.

We are structuring the tests using the "AAA" testing pattern – which stands for "Arrange, Act, Assert":

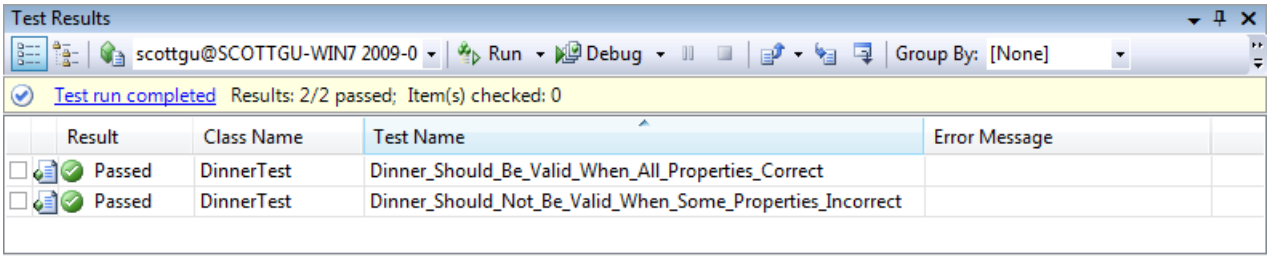
- Arrange: Setup the unit being tested
- Act: Exercise the unit under test and capture results
- Assert: Verify the behavior

When we write tests we want to avoid having the individual tests do too much. Instead each test should verify only a single concept (which will make it much easier to pinpoint the cause of failures). A good guideline is to try and only have a single assert statement for each test. If you have more than one assert statement in a test method, make sure they are all being used to test the same concept. When in doubt, make another test.

Running Tests

Visual Studio 2008 Professional (and higher editions) includes a built-in test runner that can be used to run Visual Studio Unit Test projects within the IDE. We can select the **Test->Run->All Tests in Solution** menu command (or type Ctrl R, A) to run all of our unit tests. Or alternatively we can position our cursor within a specific test class or test method and use the **Test->Run->Tests in Current Context** menu command (or type Ctrl R, T) to run a subset of the unit tests.

Let's position our cursor within the DinnerTest class and type "Ctrl R, T" to run the two tests we just defined. When we do this a "Test Results" window will appear within Visual Studio and we'll see the results of our test run listed within it:



Note: The VS test results window does not show the Class Name column by default. You can add this by right-clicking within the Test Results window and using the Add/Remove Columns menu command.

Our two tests took only a fraction of a second to run – and as you can see they both passed. We can now go on and augment them by creating additional tests that verify specific rule validations, as well as cover the two helper methods - IsUserHost() and IsUserRegistered() – that we added to the Dinner class. Having all these tests in place for the Dinner class will make it much easier and safer to add new business rules and validations to it in the future. We can add our new rule logic to Dinner, and then within seconds verify that it hasn't broken any of our previous logic functionality.

Notice how using a descriptive test name makes it easy to quickly understand what each test is verifying. I recommend using the **Tools->Options** menu command, opening the Test Tools->Test Execution configuration screen, and checking the "Double-clicking a failed or inconclusive unit test result displays the point of failure in the test" checkbox. This will allow you to double-click on a failure in the test results window and jump immediately to the assert failure.

Creating DinnersController Unit Tests

Let's now create some unit tests that verify our DinnersController functionality. We'll start by right-clicking on the "Controllers" folder within our Test project and then choose the **Add->New Test** menu command. We'll create a "Unit Test" and name it "DinnersControllerTest.cs".

We'll create two test methods that verify the Details() action method on the DinnersController. The first will verify that a View is returned when an existing Dinner is requested. The second will verify that a "NotFound" view is returned when a non-existent Dinner is requested:

```
[TestClass]
public class DinnersControllerTest {

    [TestMethod]
    public void DetailsAction_Should_Return_View_For_ExistingDinner() {

        // Arrange
        var controller = new DinnersController();

        // Act
        var result = controller.Details(1) as ViewResult;

        // Assert
        Assert.IsNotNull(result, "Expected View");
    }

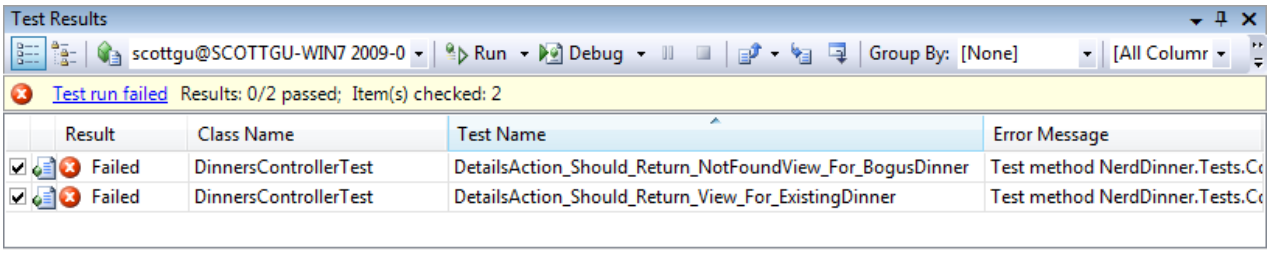
    [TestMethod]
    public void DetailsAction_Should_Return_NotFoundView_For_BogusDinner() {

        // Arrange
        var controller = new DinnersController();

        // Act
        var result = controller.Details(999) as ViewResult;

        // Assert
        Assert.AreEqual("NotFound", result.ViewName);
    }
}
```

The above code compiles clean. When we run the tests, though, they both fail:



If we look at the error messages, we'll see that the reason the tests failed was because our DinnersRepository class was unable to connect to a database. Our NerdDinner application is using a connection-string to a local SQL Server Express file which lives under the \App_Data directory of the NerdDinner application project. Because our NerdDinner.Tests project compiles and runs in a different directory then the application project, the relative path location of our connection-string is incorrect.

We *could* fix this by copying the SQL Express database file to our test project, and then add an appropriate test connection-string to it in the App.config of our test project. This would get the above tests unblocked and running.

Unit testing code using a real database, though, brings with it a number of challenges. Specifically:

- It significantly slows down the execution time of unit tests. The longer it takes to run tests, the less likely you are to execute them frequently. Ideally you want your unit tests to be able to be run in seconds – and have it be something you do as naturally as compiling the project.
- It complicates the setup and cleanup logic within tests. You want each unit test to be isolated and independent of others (with no side effects or dependencies). When working against a real database you have to be mindful of state and reset it between tests.

Let's look at a design pattern called "dependency injection" that can help us work around these issues and avoid the need to use a real database with our tests.

Dependency Injection

Right now DinnersController is tightly "coupled" to the DinnerRepository class. "Coupling" refers to a situation where a class explicitly relies on another class in order to work:

```
public class DinnersController : Controller {

    DinnerRepository dinnerRepository = new DinnerRepository();

    //
    // GET: /Dinners/Details/5

    public ActionResult Details(int id) {

        Dinner dinner = dinnerRepository.FindDinner(id);

        if (dinner == null)
            return View("NotFound");

        return View(dinner);
    }
```

Because the DinnerRepository class requires access to a database, the tightly coupled dependency the DinnersController class has on the DinnerRepository ends up requiring us to have a database in order for the DinnersController action methods to be tested.

We can get around this by employing a design pattern called "dependency injection" – which is an approach where dependencies (like repository classes that provide data access) are no longer implicitly created within classes that use them. Instead, dependencies can be explicitly passed to the class that uses them using constructor arguments. If the dependencies are defined using interfaces, we then have the flexibility to pass in "fake" dependency implementations for unit test scenarios. This enables us to create test-specific dependency implementations that do not actually require access to a database.

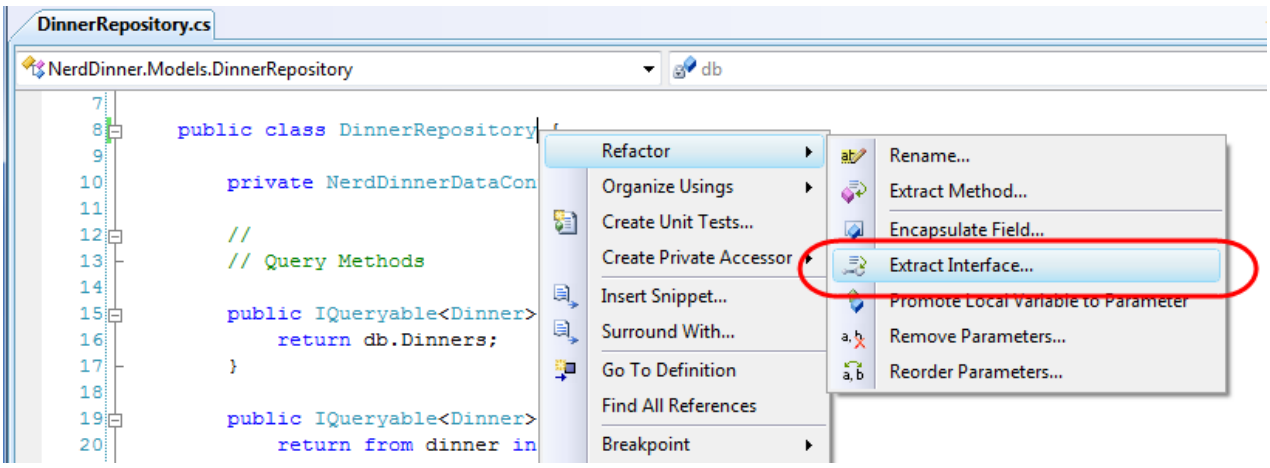
To see this in action, let's implement dependency injection with our DinnersController.

Extracting an IDinnerRepository interface

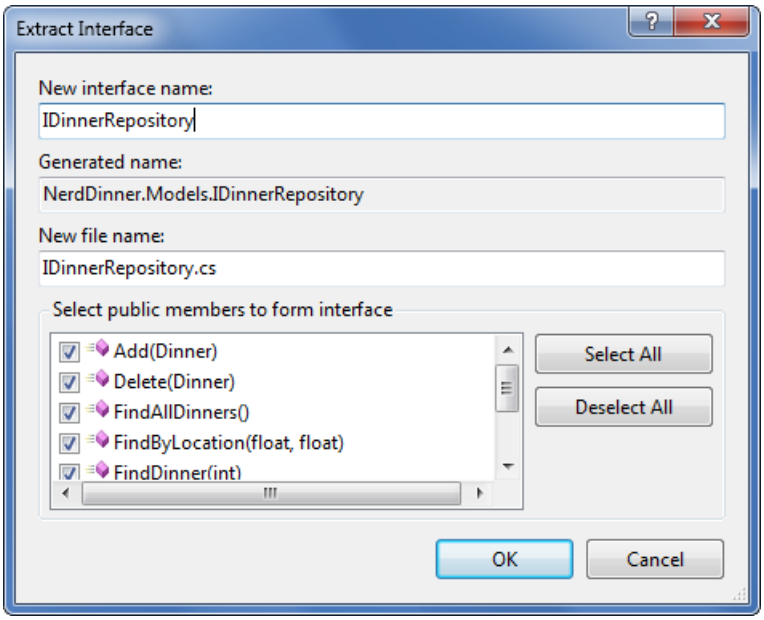
Our first step will be to create a new IDinnerRepository interface that encapsulates the repository contract our controllers require to retrieve and update Dinners.

We can define this interface contract manually by right-clicking on the \Models folder, and then choosing the **Add->New Item** menu command and creating a new interface named IDinnerRepository.cs.

Alternatively we can use the refactoring tools built-into Visual Studio Professional (and higher editions) to automatically extract and create an interface for us from our existing DinnerRepository class. To extract this interface using VS, simply position the cursor in the text editor on the DinnerRepository class, and then right-click and choose the **Refactor->Extract Interface** menu command:



This will launch the "Extract Interface" dialog and prompt us for the name of the interface to create. It will default to IDinnerRepository and automatically select all public methods on the existing DinnerRepository class to add to the interface:



When we click the "ok" button, Visual Studio will add a new IDinnerRepository interface to our application:

```
public interface IDinnerRepository {

    IQueryable<Dinner> FindAllDinners();
    IQueryable<Dinner> FindByLocation(float latitude, float longitude);
    IQueryable<Dinner> FindUpcomingDinners();
    Dinner             GetDinner(int id);

    void Add(Dinner dinner);
    void Delete(Dinner dinner);

    void Save();
}
```

And our existing DinnerRepository class will be updated so that it implements the interface:

```
public class DinnerRepository : IDinnerRepository {
   ...
}
```

Updating DinnersController to support constructor injection

We'll now update the DinnersController class to use the new interface.

Currently DinnersController is hard-coded such that its "dinnerRepository" field is always a DinnerRepository class:

```
public class DinnersController : Controller {

    DinnerRepository dinnerRepository = new DinnerRepository();

    ...
}
```

We'll change it so that the "dinnerRepository" field is of type IDinnerRepository instead of DinnerRepository. We'll then add two public DinnersController constructors. One of the constructors allows an IDinnerRepository to be

passed as an argument. The other is a default constructor that uses our existing DinnerRepository implementation:

```
public class DinnersController : Controller {

    IDinnerRepository dinnerRepository;

    public DinnersController()
        : this(new DinnerRepository()) {
    }

    public DinnersController(IDinnerRepository repository) {
        dinnerRepository = repository;
    }
    ...
}
```

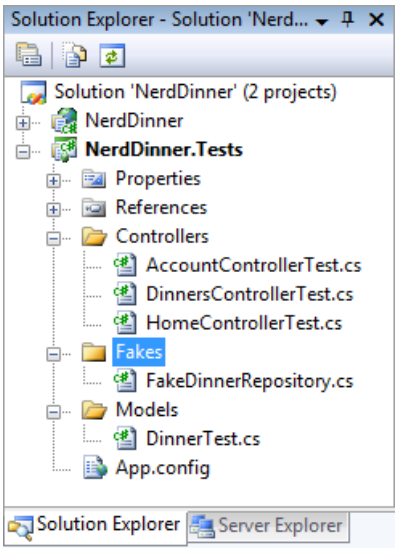
Because ASP.NET MVC by default creates controller classes using default constructors, our DinnersController at runtime will continue to use the DinnerRepository class to perform data access.

We can now update our unit tests, though, to pass in a "fake" dinner repository implementation using the parameter constructor. This "fake" dinner repository will not require access to a real database, and instead will use in-memory sample data.

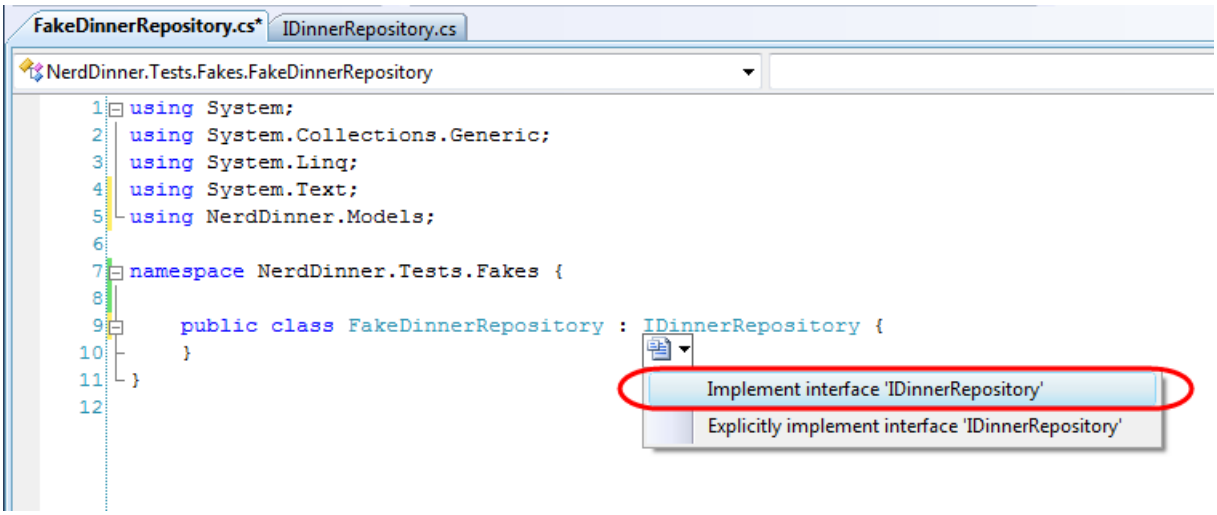
Creating the FakeDinnerRepository class

Let's create a FakeDinnerRepository class.

We'll begin by creating a "Fakes" directory within our NerdDinner.Tests project and then add a new FakeDinnerRepository class to it (right-click on the folder and choose **Add->New Class**):



We'll update the code so that the FakeDinnerRepository class implements the IDinnerRepository interface. We can then right-click on it and choose the "Implement interface IDinnerRepository" context menu command:



This will cause Visual Studio to automatically add all of the IDinnerRepository interface members to our FakeDinnerRepository class with default "stub out" implementations:

```
public class FakeDinnerRepository : IDinnerRepository {

    public IQueryable<Dinner> FindAllDinners() {
        throw new NotImplementedException();
    }

    public IQueryable<Dinner> FindByLocation(float lat, float long){
        throw new NotImplementedException();
    }

    public IQueryable<Dinner> FindUpcomingDinners() {
        throw new NotImplementedException();
    }

    public Dinner GetDinner(int id) {
        throw new NotImplementedException();
    }

    public void Add(Dinner dinner) {
        throw new NotImplementedException();
    }

    public void Delete(Dinner dinner) {
        throw new NotImplementedException();
    }

    public void Save() {
        throw new NotImplementedException();
    }
}
```

We can then update the FakeDinnerRepository implementation to work off of an in-memory List<Dinner> collection passed to it as a constructor argument:

```
public class FakeDinnerRepository : IDinnerRepository {

    private List<Dinner> dinnerList;

    public FakeDinnerRepository(List<Dinner> dinners) {
        dinnerList = dinners;
    }

    public IQueryable<Dinner> FindAllDinners() {
        return dinnerList.AsQueryable();
    }

    public IQueryable<Dinner> FindUpcomingDinners() {
        return (from dinner in dinnerList
                where dinner.EventDate > DateTime.Now
                select dinner).AsQueryable();
    }

    public IQueryable<Dinner> FindByLocation(float lat, float lon) {
        return (from dinner in dinnerList
                where dinner.Latitude == lat && dinner.Longitude == lon
                select dinner).AsQueryable();
    }

    public Dinner GetDinner(int id) {
        return dinnerList.SingleOrDefault(d => d.DinnerID == id);
    }

    public void Add(Dinner dinner) {
        dinnerList.Add(dinner);
    }

    public void Delete(Dinner dinner) {
        dinnerList.Remove(dinner);
    }

    public void Save() {
        foreach (Dinner dinner in dinnerList) {
            if (!dinner.IsValid)
                throw new ApplicationException("Rule violations");
        }
    }
}
```

We now have a fake IDinnerRepository implementation that does not require a database, and can instead work off an in-memory list of Dinner objects.

Using the FakeDinnerRepository with Unit Tests

Let's return to the DinnersController unit tests that failed earlier because the database wasn't available. We can update the test methods to use a FakeDinnerRepository populated with sample in-memory Dinner data to the DinnersController using the code below:

```
[TestClass]
public class DinnersControllerTest {

    List<Dinner> CreateTestDinners() {

        List<Dinner> dinners = new List<Dinner>();

        for (int i = 0; i < 101; i++) {

            Dinner sampleDinner = new Dinner() {
                DinnerID = i,
                Title = "Sample Dinner",
                HostedBy = "SomeUser",
                Address = "Some Address",
                Country = "USA",
                ContactPhone = "425-555-1212",
                Description = "Some description",
                EventDate = DateTime.Now.AddDays(i),
                Latitude = 99,
                Longitude = -99
            };

            dinners.Add(sampleDinner);
        }

        return dinners;
    }

    DinnersController CreateDinnersController() {
        var repository = new FakeDinnerRepository(CreateTestDinners());
        return new DinnersController(repository);
    }

    [TestMethod]
    public void DetailsAction_Should_Return_View_For_Dinner() {

        // Arrange
        var controller = CreateDinnersController();

        // Act
        var result = controller.Details(1);

        // Assert
        Assert.IsInstanceOfType(result, typeof(ViewResult));
    }

    [TestMethod]
    public void DetailsAction_Should_Return_NotFoundView_For_BogusDinner() {

        // Arrange
        var controller = CreateDinnersController();

        // Act
        var result = controller.Details(999) as ViewResult;

        // Assert
        Assert.AreEqual("NotFound", result.ViewName);
    }
}
```

And now when we run these tests they both pass:

Test Results

scottgu@SCOTTGU-WIN7 2009-0 | Run | Debug | Group By: [None] | [All Columr

Test run completed Results: 2/2 passed; Item(s) checked: 0

Result	Class Name	Test Name	Duration	Error Messag
Passed	DinnersControllerTest	DetailsAction_Should_Return_NotFoundView_For_BogusDinner	00:00:00.0003707	
Passed	DinnersControllerTest	DetailsAction_Should_Return_View_For_Dinner	00:00:00.0233735	

Best of all, they take only a fraction of a second to run, and do not require any complicated setup/cleanup logic. We can now unit test all of our DinnersController action method code (including listing, paging, details, create, update and delete) without ever needing to connect to a real database.

SIDE TOPIC: DEPENDENCY INJECTION FRAMEWORKS

Performing manual dependency injection (like we are above) works fine, but does become harder to maintain as the number of dependencies and components in an application increases. Several dependency injection frameworks exist for .NET that can help provide even more dependency management flexibility. These frameworks, also sometimes called "Inversion of Control" (IoC) containers, provide mechanisms that enable an additional level of configuration support for specifying and passing dependencies to objects at runtime (most often using constructor injection). Some of the more popular OSS Dependency Injection / IOC frameworks in .NET include: AutoFac, Ninject, Spring.NET, StructureMap, and Windsor. ASP.NET MVC exposes extensibility APIs that enable developers to participate in the resolution and instantiation of controllers, and which enables Dependency Injection / IoC frameworks to be cleanly integrated within this process. Using a DI/IOC framework would also enable us to remove the default constructor from our DinnersController – which would completely remove the coupling between it and the DinnerRepository. We won't be using a dependency injection / IOC framework with our NerdDinner application. But it is something we could consider for the future if the NerdDinner code-base and capabilities grew.

Creating Edit Action Unit Tests

Let's now create some unit tests that verify the Edit functionality of the DinnersController. We'll start by testing the HTTP-GET version of our Edit action:

```
//
// GET: /Dinners/Edit/5

[Authorize]
public ActionResult Edit(int id) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (!dinner.IsHostedBy(User.Identity.Name))
        return View("InvalidOwner");

    return View(new DinnerFormViewModel(dinner));
}
```

We'll create a test that verifies that a View backed by a DinnerFormViewModel object is rendered back when a valid dinner is requested:

```
[TestMethod]
public void EditAction_Should_Return_View_For_ValidDinner() {

    // Arrange
    var controller = CreateDinnersController();

    // Act
    var result = controller.Edit(1) as ViewResult;

    // Assert
    Assert.IsInstanceOfType(result.ViewData.Model, typeof(DinnerFormViewModel));
}
```

When we run the test, though, we'll find that it fails because a null reference exception is thrown when the Edit method accesses the User.Identity.Name property to perform the Dinner.IsHostedBy() check.

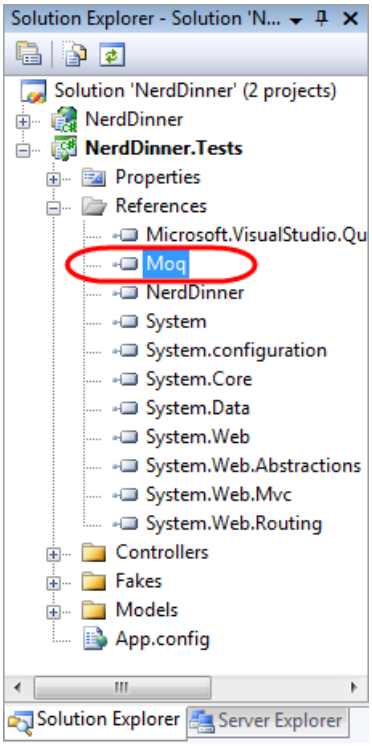
The User object on the Controller base class encapsulates details about the logged-in user, and is populated by ASP.NET MVC when it creates the controller at runtime. Because we are testing the DinnersController outside of a web-server environment, the User object isn't set (hence the null reference exception).

Mocking the User.Identity.Name property

Mocking frameworks make testing easier by enabling us to dynamically create fake versions of dependent objects that support our tests. For example, we can use a mocking framework in our Edit action test to dynamically create a User object that our DinnersController can use to lookup a simulated username. This will avoid a null reference from being thrown when we run our test.

There are many .NET mocking frameworks that can be used with ASP.NET MVC (you can see a list of them here: http://www.mockframeworks.com/). For testing our NerdDinner application we'll use an open source mocking framework called "Moq", which can be downloaded for free from http://www.mockframeworks.com/moq.

Once downloaded, we'll add a reference in our NerdDinner.Tests project to the Moq.dll assembly:



We'll then add a "CreateDinnersControllerAs(username)" helper method to our test class that takes a username as a parameter, and which then "mocks" the User.Identity.Name property on the DinnersController instance:

```
DinnersController CreateDinnersControllerAs(string userName) {

    var mock = new Mock<ControllerContext>();
    mock.SetupGet(p => p.HttpContext.User.Identity.Name).Returns(userName);
    mock.SetupGet(p => p.HttpContext.Request.IsAuthenticated).Returns(true);

    var controller = CreateDinnersController();
    controller.ControllerContext = mock.Object;

    return controller;
}
```

Above we are using Moq to create a Mock object that fakes a ControllerContext object (which is what ASP.NET MVC passes to Controller classes to expose runtime objects like User, Request, Response, and Session). We are calling the "SetupGet" method on the Mock to indicate that the HttpContext.User.Identity.Name property on

ControllerContext should return the username string we passed to the helper method.

We can mock any number of ControllerContext properties and methods. To illustrate this I've also added a SetupGet() call for the Request.IsAuthenticated property (which isn't actually needed for the tests below – but which helps illustrate how you can mock Request properties). When we are done we assign an instance of the ControllerContext mock to the DinnersController our helper method returns.

We can now write unit tests that use this helper method to test Edit scenarios involving different users:

```
[TestMethod]
public void EditAction_Should_Return_EditView_When_ValidOwner() {

    // Arrange
    var controller = CreateDinnersControllerAs("SomeUser");

    // Act
    var result = controller.Edit(1) as ViewResult;

    // Assert
    Assert.IsInstanceOfType(result.ViewData.Model, typeof(DinnerFormViewModel));
}

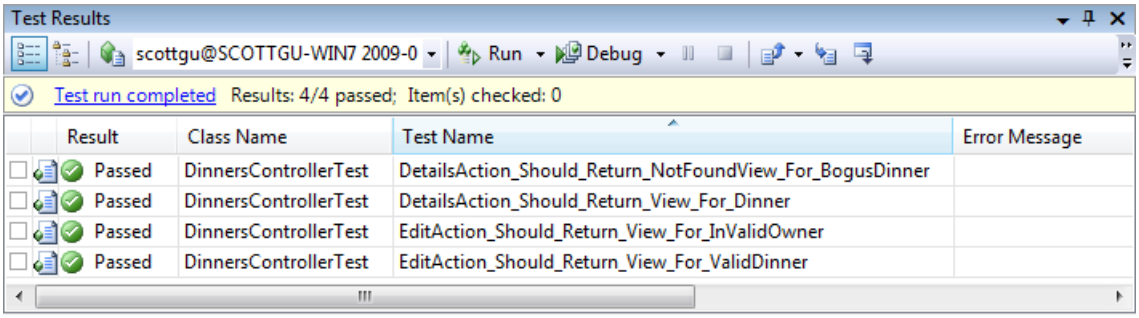
[TestMethod]
public void EditAction_Should_Return_InvalidOwnerView_When_InvalidOwner() {

    // Arrange
    var controller = CreateDinnersControllerAs("NotOwnerUser");

    // Act
    var result = controller.Edit(1) as ViewResult;

    // Assert
    Assert.AreEqual(result.ViewName, "InvalidOwner");
}
```

And now when we run the tests they pass:



Testing UpdateModel() scenarios

We've created tests that cover the HTTP-GET version of the Edit action. Let's now create some tests that verify the HTTP-POST version of the Edit action:

```
//
// POST: /Dinners/Edit/5

[AcceptVerbs(HttpVerbs.Post), Authorize]
public ActionResult Edit (int id, FormCollection collection) {

    Dinner dinner = dinnerRepository.GetDinner(id);

    if (!dinner.IsHostedBy(User.Identity.Name))
        return View("InvalidOwner");

    try {
        UpdateModel(dinner);

        dinnerRepository.Save();

        return RedirectToAction("Details", new { id=dinner.DinnerID });
    }
    catch {
        ModelState.AddModelErrors(dinner.GetRuleViolations());

        return View(new DinnerFormViewModel(dinner));
    }
}
```

The interesting new testing scenario for us to support with this action method is its usage of the UpdateModel() helper method on the Controller base class. We are using this helper method to bind form-post values to our Dinner object instance.

Below are two tests that demonstrates how we can supply form posted values for the UpdateModel() helper method to use. We'll do this by creating and populating a FormCollection object, and then assign it to the "ValueProvider" property on the Controller.

The first test verifies that on a successful save the browser is redirected to the details action. The second test verifies that when invalid input is posted the action redisplays the edit view again with an error message.

```
[TestMethod]
public void EditAction_Should_Redirect_When_Update_Successful() {

    // Arrange
    var controller = CreateDinnersControllerAs("SomeUser");

    var formValues = new FormCollection() {
        { "Title", "Another value" },
        { "Description", "Another description" }
    };

    controller.ValueProvider = formValues.ToValueProvider();

    // Act
    var result = controller.Edit(1, formValues) as RedirectToRouteResult;

    // Assert
    Assert.AreEqual("Details", result.RouteValues["Action"]);
}

[TestMethod]
public void EditAction_Should_Redisplay_With_Errors_When_Update_Fails() {

    // Arrange
    var controller = CreateDinnersControllerAs("SomeUser");

    var formValues = new FormCollection() {
        { "EventDate", "Bogus date value!!!"}
    };

    controller.ValueProvider = formValues.ToValueProvider();

    // Act
    var result = controller.Edit(1, formValues) as ViewResult;

    // Assert
    Assert.IsNotNull(result, "Expected redisplay of view");
    Assert.IsTrue(result.ViewData.ModelState.Count > 0, "Expected errors");
}
```

Testing Wrap-Up

We've covered the core concepts involved in unit testing controller classes. We can use these techniques to easily create hundreds of simple tests that verify the behavior of our application.

Because our controller and model tests do not require a real database, they are extremely fast and easy to run. We'll be able to execute hundreds of automated tests in seconds, and immediately get feedback as to whether a change we made broke something. This will help provide us the confidence to continually improve, refactor, and refine our application.

We covered testing as the last topic in this chapter – but not because testing is something you should do at the end of a development process! On the contrary, you should write automated tests as early as possible in your development process. Doing so enables you to get immediate feedback as you develop, helps you think thoughtfully about your application's use case scenarios, and guides you to design your application with clean layering and coupling in mind.

A later chapter in the book will discuss Test Driven Development (TDD), and how to use it with ASP.NET MVC. TDD is an iterative coding practice where you first write the tests that your resulting code will satisfy. With TDD you begin each feature by creating a test that verifies the functionality you are about to implement. Writing the unit test first helps ensure that you clearly understand the feature and how it is supposed to work. Only after the test is written (and you have verified that it fails) do you then implement the actual functionality the test verifies. Because you've already spent time thinking about the use case of how the feature is supposed to work, you will have a

better understanding of the requirements and how best to implement them. When you are done with the implementation you can re-run the test – and get immediate feedback as to whether the feature works correctly. We'll cover TDD more in Chapter 10.

Next Step

Some final wrap up comments.

PREVIOUS NEXT

NerdDinner Wrap Up

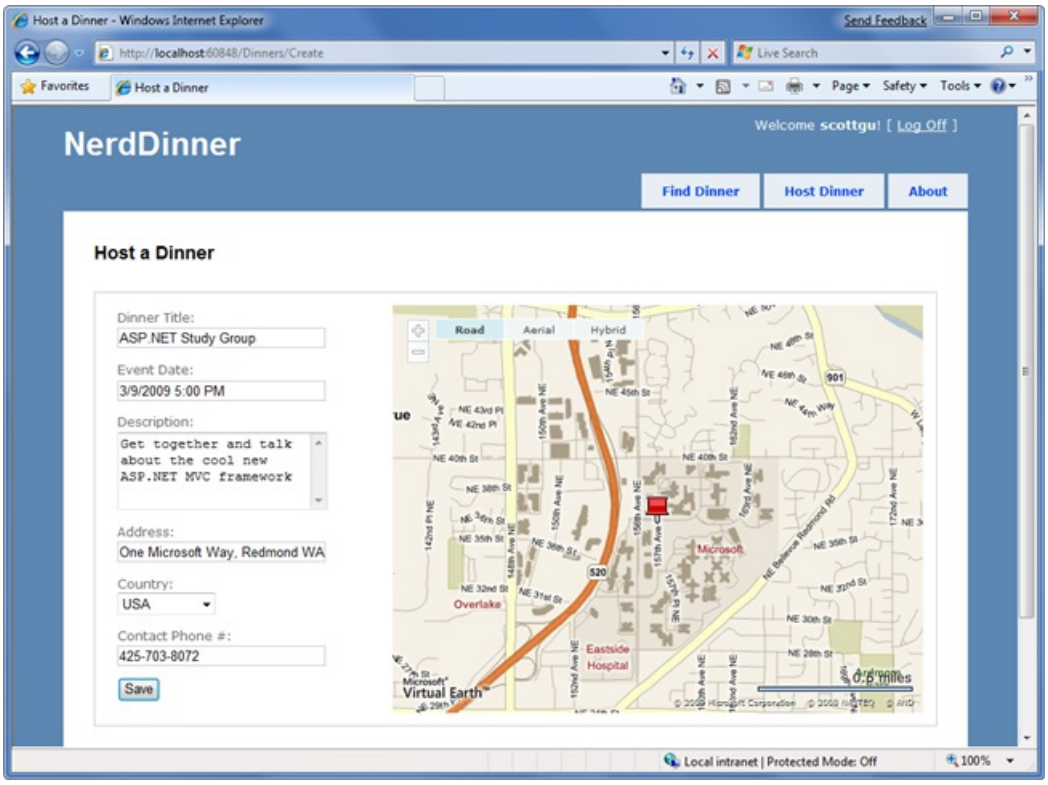
4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Download PDF

> This is the conclusion of the NerdDinner tutorial, and provides you with information on how to learn more.
>
> If you are using ASP.NET MVC 3, we recommend you follow the Getting Started With MVC 3 or MVC Music Store tutorials.

Our NerdDinner application is now complete and ready to deploy on the web.



We used a broad set of ASP.NET MVC features to build NerdDinner. Hopefully the process of developing it shed some light on how the core ASP.NET MVC features work, and provided context on how these features integrate together within an application.

Learning More

This NerdDinner tutorial is the first chapter of the new Professional ASP.NET MVC 1.0 book from Wrox Press.

You can read the rest of the book to learn even more about ASP.NET MVC.

You can also find great ASP.NET MVC tutorials and videos at: https://www.asp.net/mvc Other great ASP.NET MVC books will shortly be published. I will update this page to include links to additional books as they become available.

Hope this helps,

Scott

PREVIOUS

Movie Database

2/20/2020 • 2 minutes to read • Edit Online

How to build a Movie database application in 15 minutes.

- Create a Movie Database Application in 15 Minutes with ASP.NET MVC (C#)
- Create a Movie Database Application in 15 Minutes with ASP.NET MVC (VB)

Create a Movie Database Application in 15 Minutes with ASP.NET MVC (C#)

2/20/2020 • 17 minutes to read • Edit Online

by Stephen Walther

Download Code

> Stephen Walther builds an entire database-driven ASP.NET MVC application from start to finish. This tutorial is a great introduction for people who are new to the ASP.NET MVC Framework and who want to get a sense of the process of building an ASP.NET MVC application.

The purpose of this tutorial is to give you a sense of "what it is like" to build an ASP.NET MVC application. In this tutorial, I blast through building an entire ASP.NET MVC application from start to finish. I show you how to build a simple database-driven application that illustrates how you can list, create, and edit database records.

To simplify the process of building our application, we'll take advantage of the scaffolding features of Visual Studio 2008. We'll let Visual Studio generate the initial code and content for our controllers, models, and views.

If you have worked with Active Server Pages or ASP.NET, then you should find ASP.NET MVC very familiar. ASP.NET MVC views are very much like the pages in an Active Server Pages application. And, just like a traditional ASP.NET Web Forms application, ASP.NET MVC provides you with full access to the rich set of languages and classes provided by the .NET framework.

My hope is that this tutorial will give you a sense of how the experience of building an ASP.NET MVC application is both similar and different than the experience of building an Active Server Pages or ASP.NET Web Forms application.

Overview of the Movie Database Application

Because our goal is to keep things simple, we'll build a very simple Movie Database application. Our simple Movie Database application will allow us to do three things:

1. List a set of movie database records
2. Create a new movie database record
3. Edit an existing movie database record

Again, because we want to keep things simple, we'll take advantage of the minimum number of features of the ASP.NET MVC framework needed to build our application. For example, we won't be taking advantage of Test-Driven Development.

In order to create our application, we need to complete each of the following steps:

1. Create the ASP.NET MVC Web Application Project
2. Create the database
3. Create the database model
4. Create the ASP.NET MVC controller
5. Create the ASP.NET MVC views

Preliminaries

You'll need either Visual Studio 2008 or Visual Web Developer 2008 Express to build an ASP.NET MVC application. You also need to download the ASP.NET MVC framework.

If you don't own Visual Studio 2008, then you can download a 90 day trial version of Visual Studio 2008 from this website:

https://msdn.microsoft.com/vs2008/products/cc268305.aspx

Alternatively, you can create ASP.NET MVC applications with Visual Web Developer Express 2008. If you decide to use Visual Web Developer Express then you must have Service Pack 1 installed. You can download Visual Web Developer 2008 Express with Service Pack 1 from this website:

https://www.microsoft.com/downloads/details.aspx?FamilyId=BDB6391C-05CA-4036-9154-6DF4F6DEBD14&displaylang=en

After you install either Visual Studio 2008 or Visual Web Developer 2008, you need to install the ASP.NET MVC framework. You can download the ASP.NET MVC framework from the following website:

https://www.asp.net/mvc/

NOTE

Instead of downloading the ASP.NET framework and the ASP.NET MVC framework individually, you can take advantage of the Web Platform Installer. The Web Platform Installer is an application that enables you to easily manage the installed applications are your computer:

https://www.microsoft.com/web/gallery/Install.aspx

Creating an ASP.NET MVC Web Application Project

Let's start by creating a new ASP.NET MVC Web Application project in Visual Studio 2008. Select the menu option **File, New Project** and you will see the New Project dialog box in Figure 1. Select C# as the programming language and select the ASP.NET MVC Web Application project template. Give your project the name MovieApp and click the OK button.

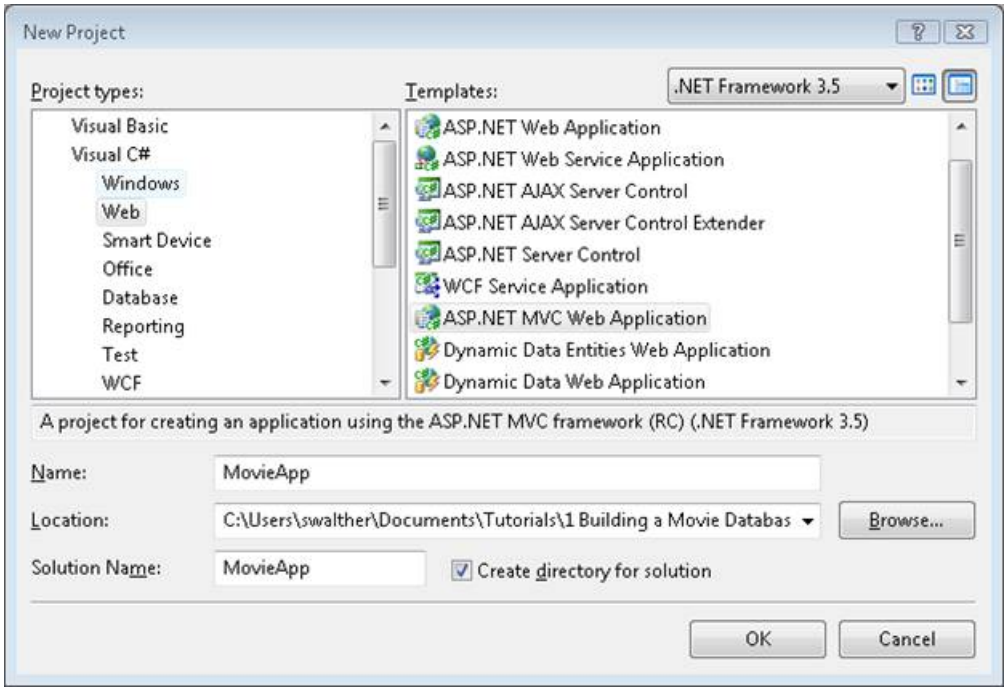


Figure 01: The New Project dialog box (Click to view full-size image)

Make sure that you select .NET Framework 3.5 from the dropdown list at the top of the New Project dialog or the

ASP.NET MVC Web Application project template won't appear.

Whenever you create a new MVC Web Application project, Visual Studio prompts you to create a separate unit test project. The dialog in Figure 2 appears. Because we won't be creating tests in this tutorial because of time constraints (and, yes, we should feel a little guilty about this) select the **No** option and click the **OK** button.

> **NOTE**
> Visual Web Developer does not support test projects.

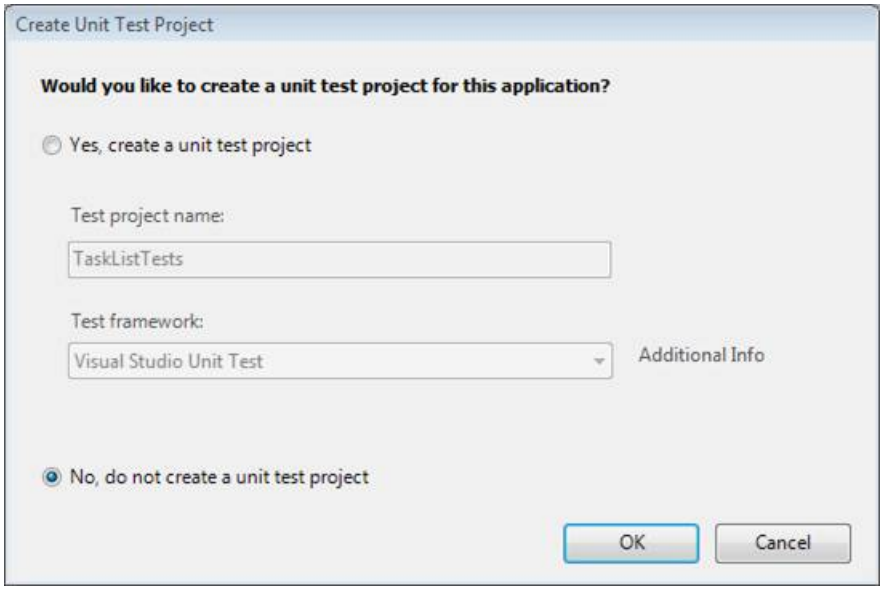


Figure 02: The Create Unit Test Project dialog (Click to view full-size image)

An ASP.NET MVC application has a standard set of folders: a Models, Views, and Controllers folder. You can see this standard set of folders in the Solution Explorer window. We'll need to add files to each of the Models, Views, and Controllers folders in order to build our Movie Database application.

When you create a new MVC application with Visual Studio, you get a sample application. Because we want to start from scratch, we need to delete the content for this sample application. You need to delete the following file and the following folder:

- Controllers\HomeController.cs
- Views\Home

Creating the Database

We need to create a database to hold our movie database records. Luckily, Visual Studio includes a free database named SQL Server Express. Follow these steps to create the database:

1. Right-click the App_Data folder in the Solution Explorer window and select the menu option **Add, New Item**.
2. Select the **Data** category and select the **SQL Server Database** template (see Figure 3).
3. Name your new database *MoviesDB.mdf* and click the **Add** button.

After you create your database, you can connect to the database by double-clicking the MoviesDB.mdf file located in the App_Data folder. Double-clicking the MoviesDB.mdf file opens the Server Explorer window.

> **NOTE**
> The Server Explorer window is named the Database Explorer window in the case of Visual Web Developer.

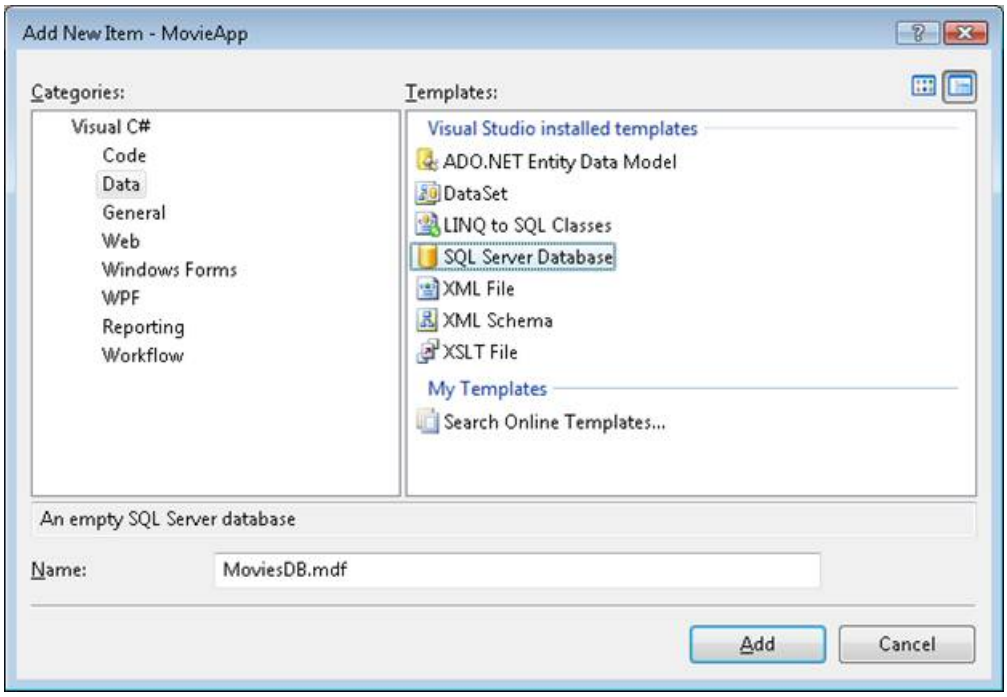


Figure 03: Creating a Microsoft SQL Server Database (Click to view full-size image)

Next, we need to create a new database table. From within the Sever Explorer window, right-click the Tables folder and select the menu option **Add New Table**. Selecting this menu option opens the database table designer. Create the following database columns:

COLUMN NAME	DATA TYPE	ALLOW NULLS
Id	Int	False
Title	Nvarchar(100)	False
Director	Nvarchar(100)	False
DateReleased	DateTime	False

The first column, the Id column, has two special properties. First, you need to mark the Id column as the primary key column. After selecting the Id column, click the **Set Primary Key** button (it is the icon that looks like a key). Second, you need to mark the Id column as an Identity column. In the Column Properties window, scroll down to the Identity Specification section and expand it. Change the **Is Identity** property to the value **Yes**. When you are finished, the table should look like Figure 4.

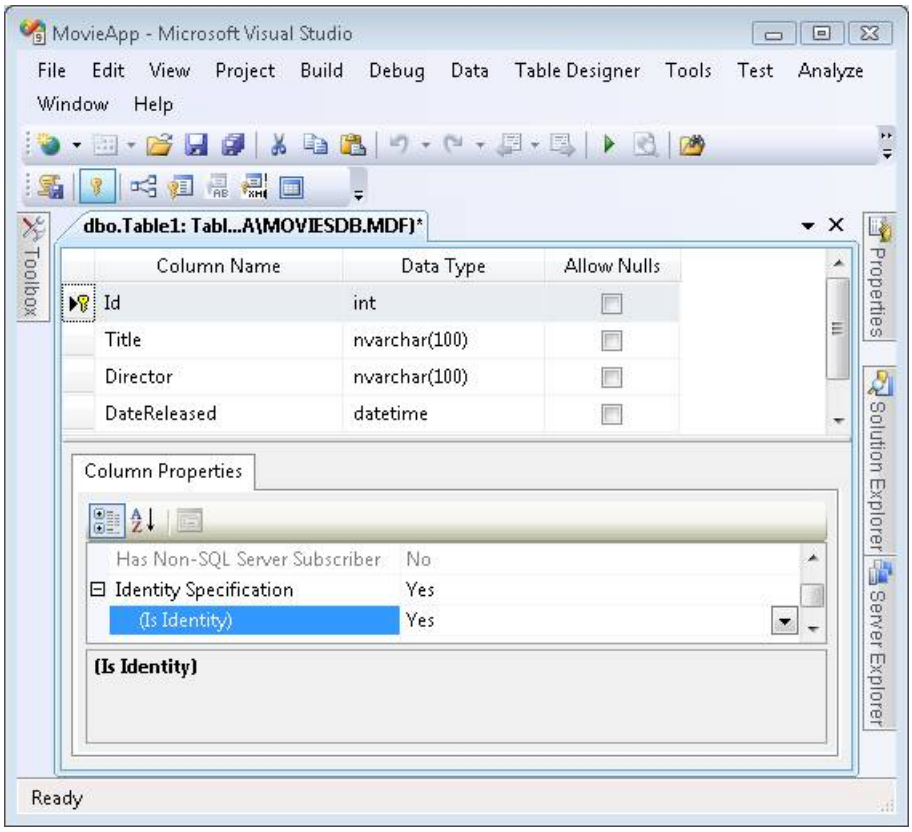


Figure 04: The Movies database table (Click to view full-size image)

The final step is to save the new table. Click the Save button (the icon of the floppy) and give the new table the name Movies.

After you finish creating the table, add some movie records to the table. Right-click the Movies table in the Server Explorer window and select the menu option **Show Table Data**. Enter a list of your favorite movies (see Figure 5).

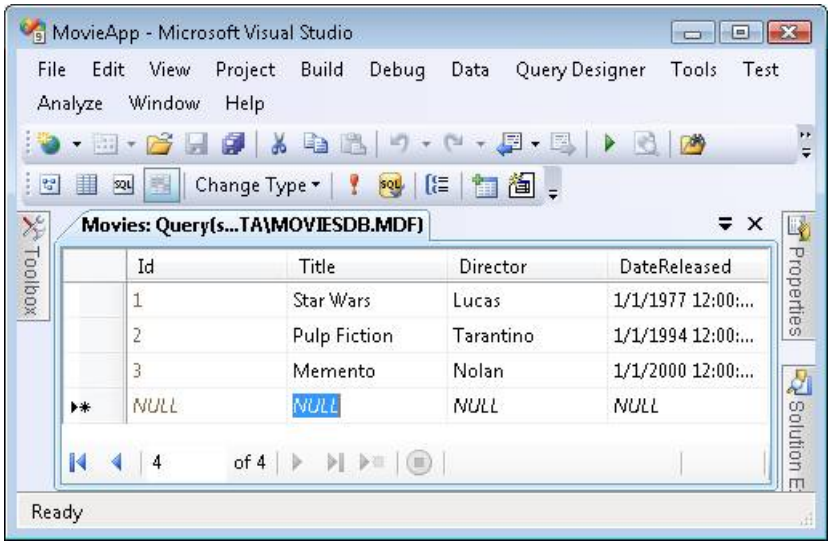


Figure 05: Entering movie records (Click to view full-size image)

Creating the Model

We next need to create a set of classes to represent our database. We need to create a database model. We'll take advantage of the Microsoft Entity Framework to generate the classes for our database model automatically.

> **NOTE**
> The ASP.NET MVC framework is not tied to the Microsoft Entity Framework. You can create your database model classes by taking advantage of a variety of Object Relational Mapping (OR/M) tools including LINQ to SQL, Subsonic, and NHibernate.

Follow these steps to launch the Entity Data Model Wizard:

1. Right-click the Models folder in the Solution Explorer window and the select the menu option **Add, New Item**.
2. Select the **Data** category and select the **ADO.NET Entity Data Model** template.
3. Give your data model the name *MoviesDBModel.edmx* and click the **Add** button.

After you click the Add button, the Entity Data Model Wizard appears (see Figure 6). Follow these steps to complete the wizard:

1. In the **Choose Model Contents** step, select the **Generate from database** option.
2. In the **Choose Your Data Connection** step, use the *MoviesDB.mdf* data connection and the name *MoviesDBEntities* for the connection settings. Click the **Next** button.
3. In the **Choose Your Database Objects** step, expand the Tables node, select the Movies table. Enter the namespace *MovieApp.Models* and click the **Finish** button.

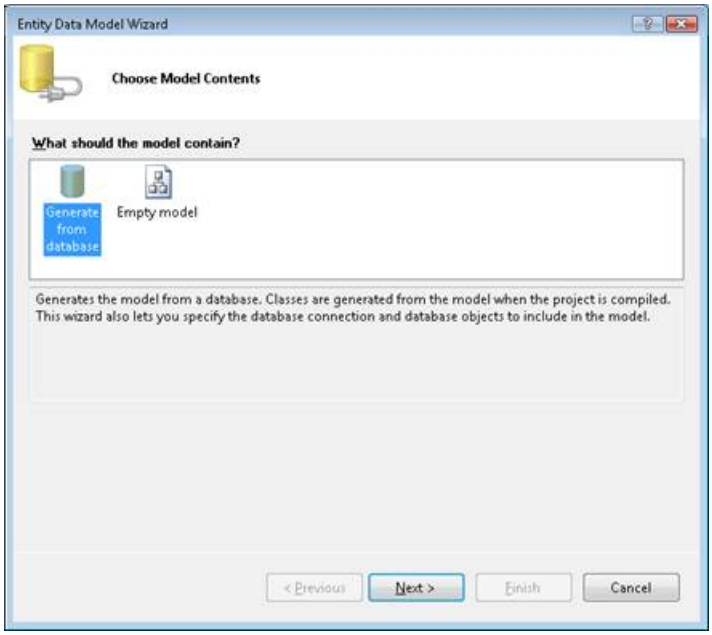


Figure 06: Generating a database model with the Entity Data Model Wizard (Click to view full-size image)

After you complete the Entity Data Model Wizard, the Entity Data Model Designer opens. The Designer should display the Movies database table (see Figure 7).

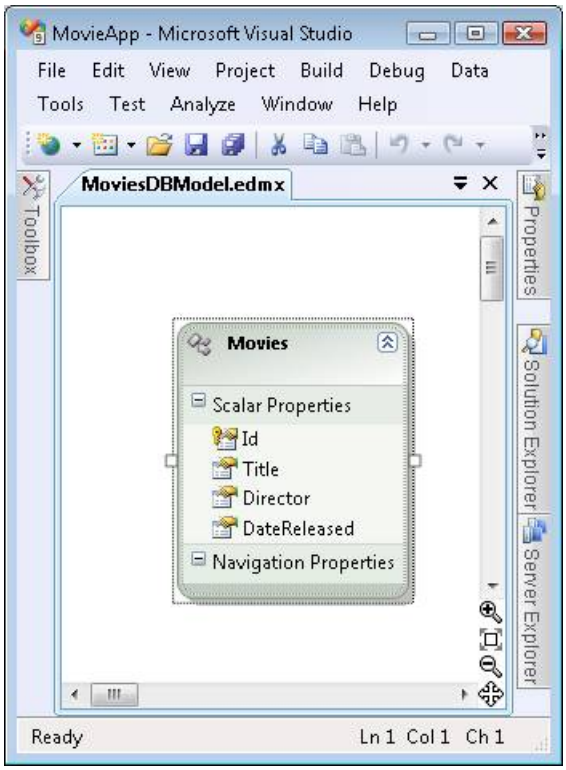


Figure 07: The Entity Data Model Designer (Click to view full-size image)

We need to make one change before we continue. The Entity Data Wizard generates a model class named *Movies* that represents the Movies database table. Because we'll use the Movies class to represent a particular movie, we need to modify the name of the class to be *Movie* instead of *Movies* (singular rather than plural).

Double-click the name of the class on the designer surface and change the name of the class from Movies to Movie. After making this change, click the **Save** button (the icon of the floppy disk) to generate the Movie class.

Creating the ASP.NET MVC Controller

The next step is to create the ASP.NET MVC controller. A controller is responsible for controlling how a user interacts with an ASP.NET MVC application.

Follow these steps:

1. In the Solution Explorer window, right-click the Controllers folder and select the menu option **Add, Controller**.
2. In the Add Controller dialog, enter the name *HomeController* and check the checkbox labeled **Add action methods for Create, Update, and Details scenarios** (see Figure 8).
3. Click the **Add** button to add the new controller to your project.

After you complete these steps, the controller in Listing 1 is created. Notice that it contains methods named Index, Details, Create, and Edit. In the following sections, we'll add the necessary code to get these methods to work.

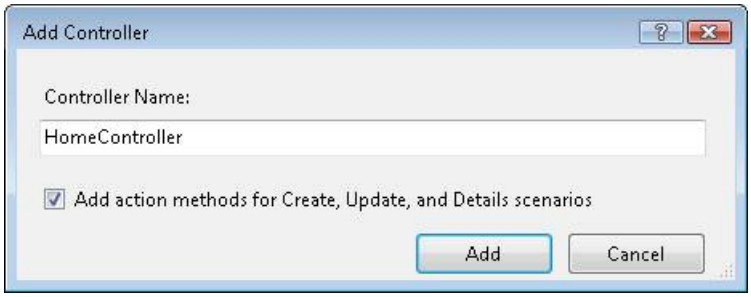


Figure 08: Adding a new ASP.NET MVC Controller (Click to view full-size image)

Listing 1 – Controllers\HomeController.cs

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Mvc.Ajax;

namespace MovieApp.Controllers

{

    public class HomeController : Controller

    {

        //

        // GET: /Home/

        public ActionResult Index()

        {

            return View();

        }

        //

        // GET: /Home/Details/5

        public ActionResult Details(int id)

        {

            return View();

        }

        //

        // GET: /Home/Create

        public ActionResult Create()

        {

            return View();

        }

        //

        // POST: /Home/Create

        [AcceptVerbs(HttpVerbs.Post)]

        public ActionResult Create(FormCollection collection)

        {

            try
```

```
            {
                // TODO: Add insert logic here
                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }
        //
        // GET: /Home/Edit/5
        public ActionResult Edit(int id)
        {
            return View();
        }
        //
        // POST: /Home/Edit/5
        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Edit(int id, FormCollection collection)
        {
            try
            {
                // TODO: Add update logic here
                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }
    }
}
```

Listing Database Records

The Index() method of the Home controller is the default method for an ASP.NET MVC application. When you run an ASP.NET MVC application, the Index() method is the first controller method that is called.

We'll use the Index() method to display the list of records from the Movies database table. We'll take advantage of the database model classes that we created earlier to retrieve the movie database records with the Index() method.

I've modified the HomeController class in Listing 2 so that it contains a new private field named _db. The MoviesDBEntities class represents our database model and we'll use this class to communicate with our database.

I've also modified the Index() method in Listing 2. The Index() method uses the MoviesDBEntities class to retrieve all of the movie records from the Movies database table. The expression *_db.MovieSet.ToList()* returns a list of all of the movie records from the Movies database table.

The list of movies is passed to the view. Anything that gets passed to the View() method gets passed to the view as view data.

Listing 2 – Controllers/HomeController.cs (modified Index method)

```
using System.Linq;

using System.Web.Mvc;

using MovieApp.Models;

namespace MovieApp.Controllers

{

    public class HomeController : Controller

    {

        private MoviesDBEntities _db = new MoviesDBEntities();

        public ActionResult Index()

        {

            return View(_db.MovieSet.ToList());

        }

    }

}
```

The Index() method returns a view named Index. We need to create this view to display the list of movie database records. Follow these steps:

You should build your project (select the menu option **Build, Build Solution**) before opening the **Add View** dialog or no classes will appear in the **View data class** dropdown list.

1. Right-click the Index() method in the code editor and select the menu option **Add View** (see Figure 9).
2. In the Add View dialog, verify that the checkbox labeled **Create a strongly-typed view** is checked.
3. From the **View content** dropdown list, select the value *List*.
4. From the **View data class** dropdown list, select the value *MovieApp.Models.Movie*.
5. Click the Add button to create the new view (see Figure 10).

After you complete these steps, a new view named Index.aspx is added to the Views\Home folder. The contents of the Index view are included in Listing 3.

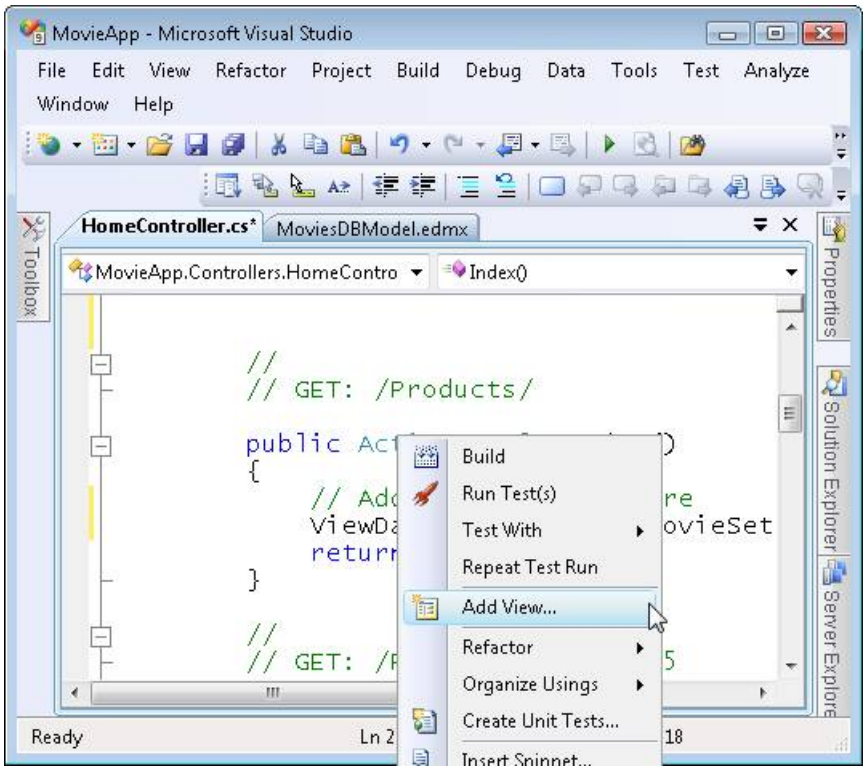


Figure 09: Adding a view from a controller action (Click to view full-size image)

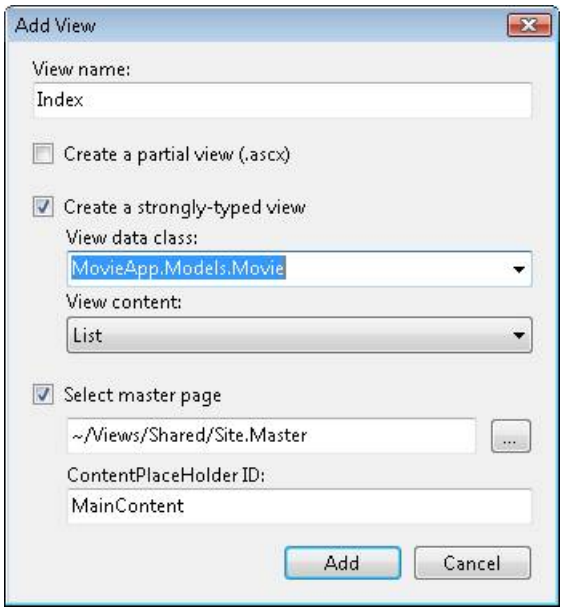


Figure 10: Creating a new view with the Add View dialog (Click to view full-size image)

Listing 3 – Views\Home\Index.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<IEnumerable<MovieApp.Models.Movie>>" %>

    <asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

       Index

    </asp:Content>

    <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

        <h2>Index</h2>

        <table>
```

```
            <tr>
                <th></th>
                <th>
                    Id
                </th>
                <th>
                    Title
                </th>
                <th>
                    Director
                </th>
                <th>
                    DateReleased
                </th>
            </tr>
        <% foreach (var item in Model) { %>
            <tr>
                <td>
                    <%= Html.ActionLink("Edit", "Edit", new { id=item.Id }) %> |
                    <%= Html.ActionLink("Details", "Details", new { id=item.Id })%>
                </td>
                <td>
                    <%= Html.Encode(item.Id) %>
                </td>
                <td>
                    <%= Html.Encode(item.Title) %>
                </td>
                <td>
                    <%= Html.Encode(item.Director) %>
                </td>
                <td>
                    <%= Html.Encode(String.Format("{0:g}", item.DateReleased)) %>
                </td>
            </tr>
        <% } %>
```

```
        %> }%>

        </table>

        <p>

            <%= Html.ActionLink("Create New", "Create") %>

        </p>

    </asp:Content>
```

The Index view displays all of the movie records from the Movies database table within an HTML table. The view contains a foreach loop that iterates through each movie represented by the ViewData.Model property. If you run your application by hitting the F5 key, then you'll see the web page in Figure 11.

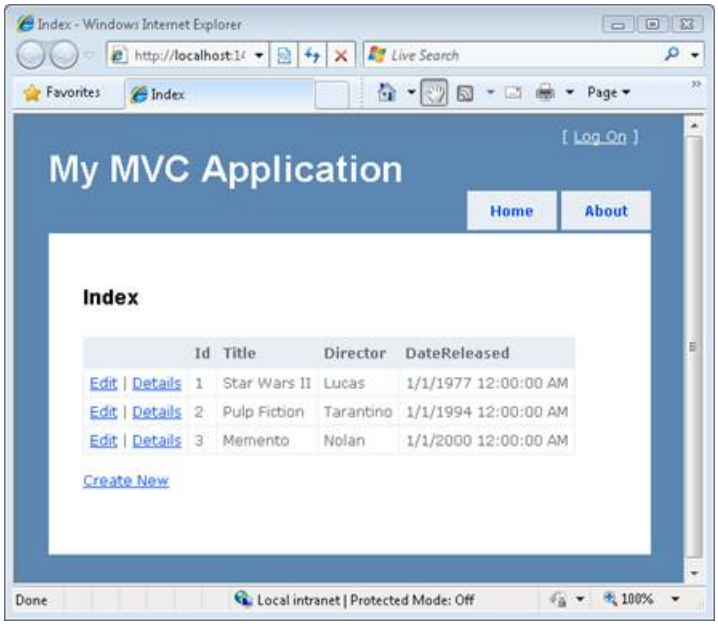


Figure 11: The Index view (Click to view full-size image)

Creating New Database Records

The Index view that we created in the previous section includes a link for creating new database records. Let's go ahead and implement the logic and create the view necessary for creating new movie database records.

The Home controller contains two methods named Create(). The first Create() method has no parameters. This overload of the Create() method is used to display the HTML form for creating a new movie database record.

The second Create() method has a FormCollection parameter. This overload of the Create() method is called when the HTML form for creating a new movie is posted to the server. Notice that this second Create() method has an AcceptVerbs attribute that prevents the method from being called unless an HTTP POST operation is performed.

This second Create() method has been modified in the updated HomeController class in Listing 4. The new version of the Create() method accepts a Movie parameter and contains the logic for inserting a new movie into the Movies database table.

> **NOTE**
>
> Notice the Bind attribute. Because we don't want to update the Movie Id property from HTML form, we need to explicitly exclude this property.

Listing 4 – Controllers\HomeController.cs (modified Create method)

```
//

// GET: /Home/Create

public ActionResult Create()

{

    return View();

}

//

// POST: /Home/Create

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Create([Bind(Exclude="Id")] Movie movieToCreate)

{

     if (!ModelState.IsValid)

        return View();

    _db.AddToMovieSet(movieToCreate);

    _db.SaveChanges();

    return RedirectToAction("Index");

}
```

Visual Studio makes it easy to create the form for creating a new movie database record (see Figure 12). Follow these steps:

1. Right-click the Create() method in the code editor and select the menu option **Add View**.
2. Verify that the checkbox labeled **Create a strongly-typed view** is checked.
3. From the **View content** dropdown list, select the value *Create*.
4. From the **View data class** dropdown list, select the value *MovieApp.Models.Movie*.
5. Click the **Add** button to create the new view.

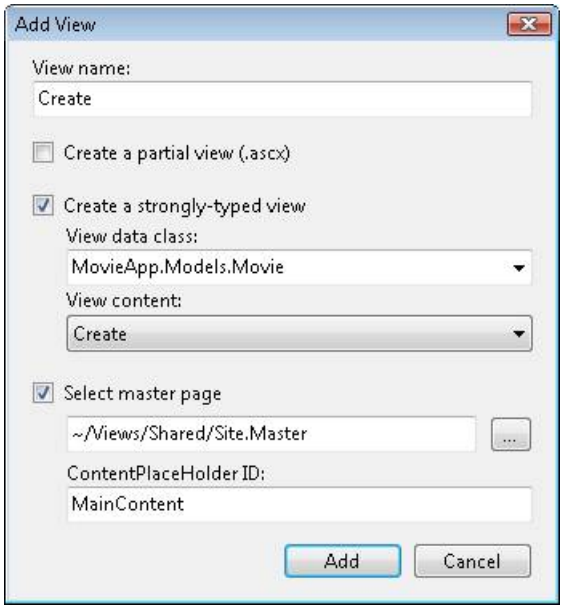


Figure 12: Adding the Create view (Click to view full-size image)

Visual Studio generates the view in Listing 5 automatically. This view contains an HTML form that includes fields that correspond to each of the properties of the Movie class.

Listing 5 – Views\Home\Create.aspx

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MovieApp.Models.Movie>" %>

    <asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

       Create

    </asp:Content>

    <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

        <h2>Create</h2>

        <%= Html.ValidationSummary("Create was unsuccessful. Please correct the errors and try again.") %>

        <% using (Html.BeginForm()) {%>

            <fieldset>

                <legend>Fields</legend>

                <p>

                    <label for="Id">Id:</label>

                    <%= Html.TextBox("Id") %>

                    <%= Html.ValidationMessage("Id", "*") %>

                </p>

                <p>

                    <label for="Title">Title:</label>

                    <%= Html.TextBox("Title") %>

                    <%= Html.ValidationMessage("Title", "*") %>

                </p>

                <p>

                    <label for="Director">Director:</label>

                    <%= Html.TextBox("Director") %>

                    <%= Html.ValidationMessage("Director", "*") %>

                </p>

                <p>

                    <label for="DateReleased">DateReleased:</label>

                    <%= Html.TextBox("DateReleased") %>

                    <%= Html.ValidationMessage("DateReleased", "*") %>

                </p>
```

```
                <p>
                    <input type="submit" value="Create" />
                </p>
            </fieldset>
        <% } %>
        <div>
            <%=Html.ActionLink("Back to List", "Index") %>
        </div>
    </asp:Content>
```

NOTE

The HTML form generated by the Add View dialog generates an Id form field. Because the Id column is an Identity column, we don't need this form field and you can safely remove it.

After you add the Create view, you can add new Movie records to the database. Run your application by pressing the F5 key and click the Create New link to see the form in Figure 13. If you complete and submit the form, a new movie database record is created.

Notice that you get form validation automatically. If you neglect to enter a release date for a movie, or you enter an invalid release date, then the form is redisplayed and the release date field is highlighted.

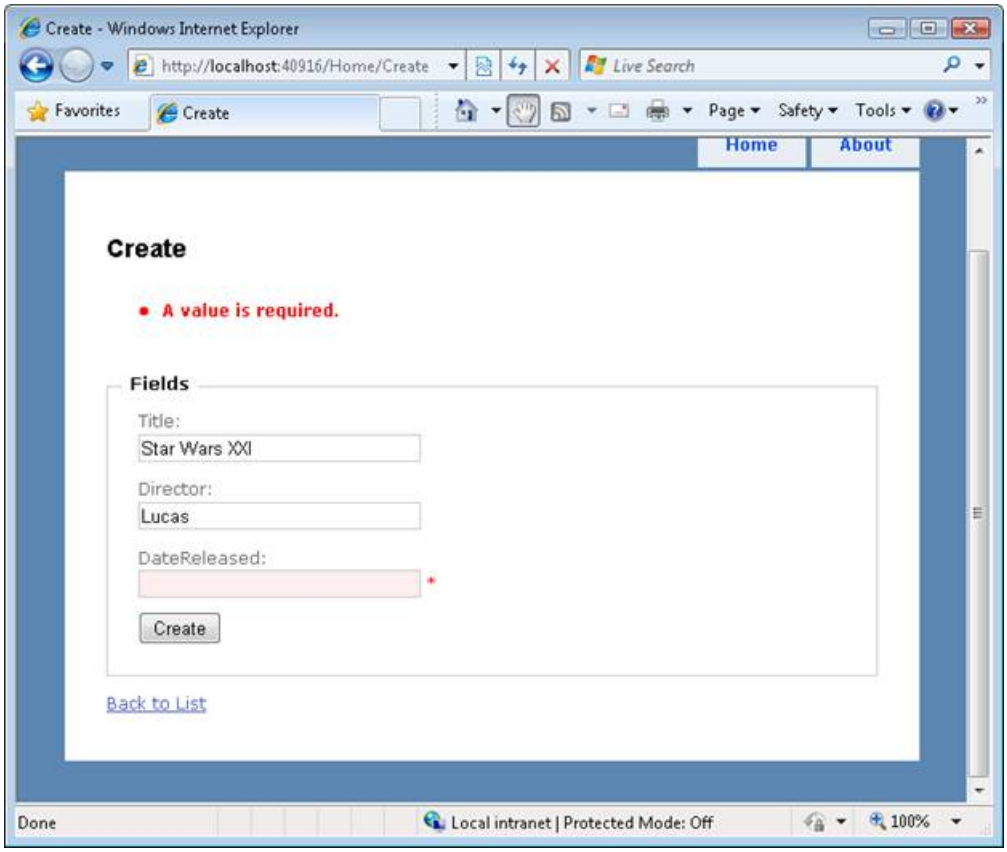


Figure 13: Creating a new movie database record (Click to view full-size image)

Editing Existing Database Records

In the previous sections, we discussed how you can list and create new database records. In this final section, we discuss how you can edit existing database records.

First, we need to generate the Edit form. This step is easy since Visual Studio will generate the Edit form for us automatically. Open the HomeController.cs class in the Visual Studio code editor and follow these steps:

1. Right-click the Edit() method in the code editor and select the menu option **Add View** (see Figure 14).
2. Check the checkbox labeled **Create a strongly-typed view**.
3. From the **View content** dropdown list, select the value *Edit*.
4. From the **View data class** dropdown list, select the value *MovieApp.Models.Movie*.
5. Click the **Add** button to create the new view.

Completing these steps adds a new view named Edit.aspx to the Views\Home folder. This view contains an HTML form for editing a movie record.

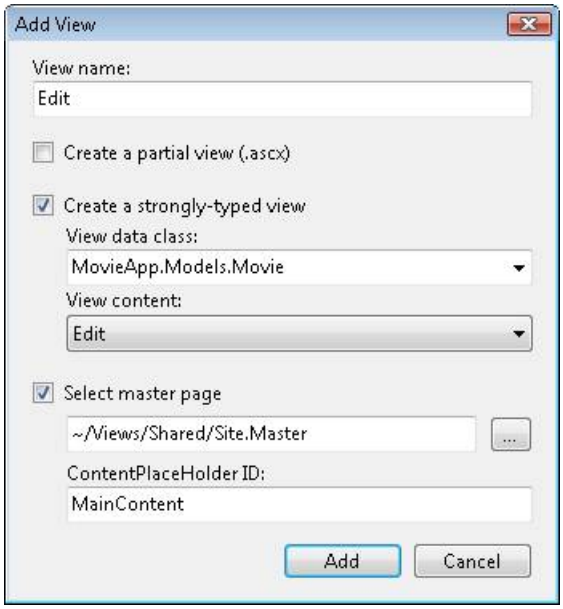


Figure 14: Adding the Edit view (Click to view full-size image)

> **NOTE**
> The Edit view contains an HTML form field that corresponds to the Movie Id property. Because you don't want people editing the value of the Id property, you should remove this form field.

Finally, we need to modify the Home controller so that it supports editing a database record. The updated HomeController class is contained in Listing 6.

Listing 6 – Controllers\HomeController.cs (Edit methods)

```
//

// GET: /Home/Edit/5

public ActionResult Edit(int id)

{

    var movieToEdit = (from m in _db.MovieSet

                       where m.Id == id

                       select m).First();

    return View(movieToEdit);

}

//

// POST: /Home/Edit/5

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Edit(Movie movieToEdit)

{

    var originalMovie = (from m in _db.MovieSet

                         where m.Id == movieToEdit.Id

                         select m).First();

    if (!ModelState.IsValid)

        return View(originalMovie);

        _db.ApplyPropertyChanges(originalMovie.EntityKey.EntitySetName, movieToEdit);

        _db.SaveChanges();

        return RedirectToAction("Index");

}
```

In Listing 6, I've added additional logic to both overloads of the Edit() method. The first Edit() method returns the movie database record that corresponds to the Id parameter passed to the method. The second overload performs the updates to a movie record in the database.

Notice that you must retrieve the original movie, and then call ApplyPropertyChanges(), to update the existing movie in the database.

Summary

The purpose of this tutorial was to give you a sense of the experience of building an ASP.NET MVC application. I hope that you discovered that building an ASP.NET MVC web application is very similar to the experience of building an Active Server Pages or ASP.NET application.

In this tutorial, we examined only the most basic features of the ASP.NET MVC framework. In future tutorials, we dive deeper into topics such as controllers, controller actions, views, view data, and HTML helpers.

NEXT

Create a Movie Database Application in 15 Minutes with ASP.NET MVC (VB)

2/20/2020 • 17 minutes to read • Edit Online

by Stephen Walther

Download Code

> Stephen Walther builds an entire database-driven ASP.NET MVC application from start to finish. This tutorial is a great introduction for people who are new to the ASP.NET MVC Framework and who want to get a sense of the process of building an ASP.NET MVC application.

The purpose of this tutorial is to give you a sense of "what it is like" to build an ASP.NET MVC application. In this tutorial, I blast through building an entire ASP.NET MVC application from start to finish. I show you how to build a simple database-driven application that illustrates how you can list, create, and edit database records.

To simplify the process of building our application, we'll take advantage of the scaffolding features of Visual Studio 2008. We'll let Visual Studio generate the initial code and content for our controllers, models, and views.

If you have worked with Active Server Pages or ASP.NET, then you should find ASP.NET MVC very familiar. ASP.NET MVC views are very much like the pages in an Active Server Pages application. And, just like a traditional ASP.NET Web Forms application, ASP.NET MVC provides you with full access to the rich set of languages and classes provided by the .NET framework.

My hope is that this tutorial will give you a sense of how the experience of building an ASP.NET MVC application is both similar and different than the experience of building an Active Server Pages or ASP.NET Web Forms application.

Overview of the Movie Database Application

Because our goal is to keep things simple, we'll build a very simple Movie Database application. Our simple Movie Database application will allow us to do three things:

1. List a set of movie database records
2. Create a new movie database record
3. Edit an existing movie database record

Again, because we want to keep things simple, we'll take advantage of the minimum number of features of the ASP.NET MVC framework needed to build our application. For example, we won't be taking advantage of Test-Driven Development.

In order to create our application, we need to complete each of the following steps:

1. Create the ASP.NET MVC Web Application Project
2. Create the database
3. Create the database model
4. Create the ASP.NET MVC controller
5. Create the ASP.NET MVC views

Preliminaries

You'll need either Visual Studio 2008 or Visual Web Developer 2008 Express to build an ASP.NET MVC application. You also need to download the ASP.NET MVC framework.

If you don't own Visual Studio 2008, then you can download a 90 day trial version of Visual Studio 2008 from this website:

https://msdn.microsoft.com/vs2008/products/cc268305.aspx

Alternatively, you can create ASP.NET MVC applications with Visual Web Developer Express 2008. If you decide to use Visual Web Developer Express then you must have Service Pack 1 installed. You can download Visual Web Developer 2008 Express with Service Pack 1 from this website:

https://www.microsoft.com/downloads/details.aspx?FamilyId=BDB6391C-05CA-4036-9154-6DF4F6DEBD14&displaylang=en

After you install either Visual Studio 2008 or Visual Web Developer 2008, you need to install the ASP.NET MVC framework. You can download the ASP.NET MVC framework from the following website:

https://www.asp.net/mvc/

> **NOTE**
>
> Instead of downloading the ASP.NET framework and the ASP.NET MVC framework individually, you can take advantage of the Web Platform Installer. The Web Platform Installer is an application that enables you to easily manage the installed applications are your computer:
>
> https://www.microsoft.com/web/gallery/Install.aspx

Creating an ASP.NET MVC Web Application Project

Let's start by creating a new ASP.NET MVC Web Application project in Visual Studio 2008. Select the menu option **File, New Project** and you will see the New Project dialog box in Figure 1. Select Visual Basic as the programming language and select the ASP.NET MVC Web Application project template. Give your project the name MovieApp and click the OK button.

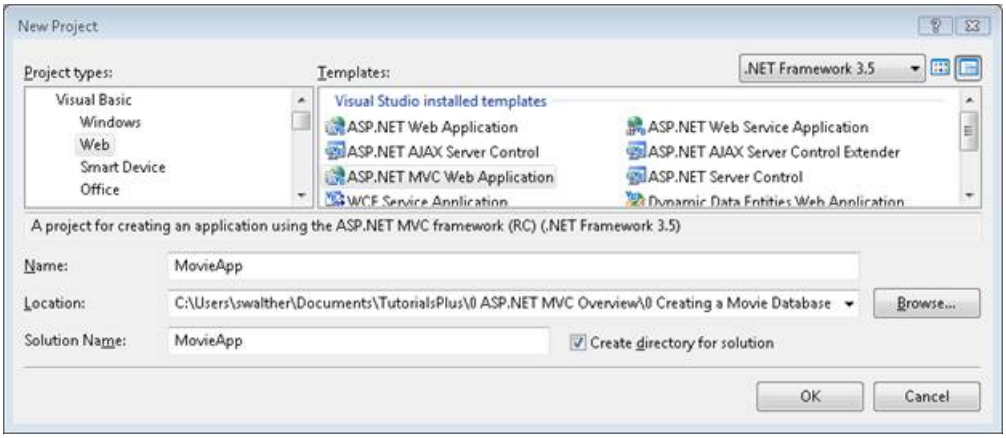


Figure 01: The New Project dialog box (Click to view full-size image)

Make sure that you select .NET Framework 3.5 from the dropdown list at the top of the New Project dialog or the ASP.NET MVC Web Application project template won't appear.

Whenever you create a new MVC Web Application project, Visual Studio prompts you to create a separate unit test project. The dialog in Figure 2 appears. Because we won't be creating tests in this tutorial because of time constraints (and, yes, we should feel a little guilty about this) select the **No** option and click the **OK** button.

NOTE

Visual Web Developer does not support test projects.

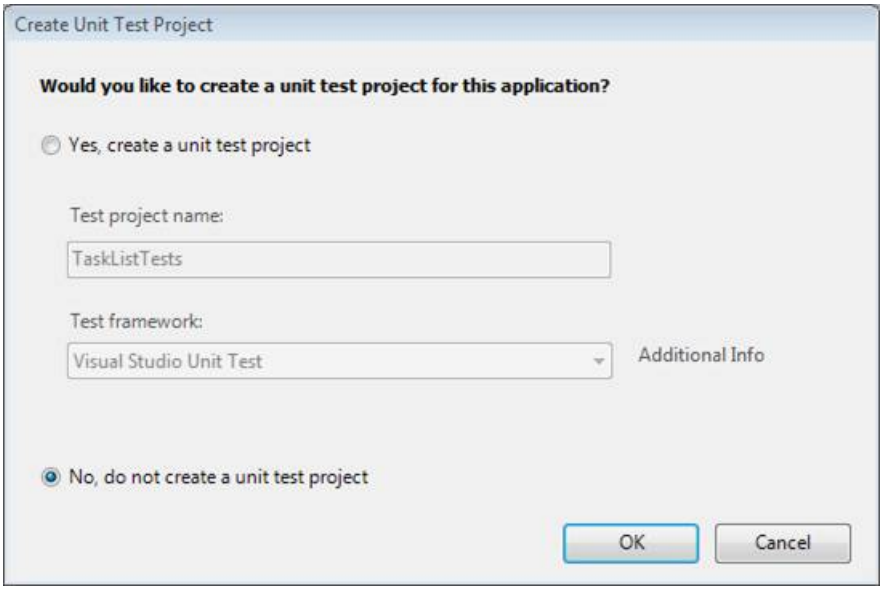


Figure 02: The Create Unit Test Project dialog (Click to view full-size image)

An ASP.NET MVC application has a standard set of folders: a Models, Views, and Controllers folder. You can see this standard set of folders in the Solution Explorer window. We'll need to add files to each of the Models, Views, and Controllers folders in order to build our Movie Database application.

When you create a new MVC application with Visual Studio, you get a sample application. Because we want to start from scratch, we need to delete the content for this sample application. You need to delete the following file and the following folder:

- Controllers\HomeController.vb
- Views\Home

Creating the Database

We need to create a database to hold our movie database records. Luckily, Visual Studio includes a free database named SQL Server Express. Follow these steps to create the database:

1. Right-click the App_Data folder in the Solution Explorer window and select the menu option **Add, New Item**.
2. Select the **Data** category and select the **SQL Server Database** template (see Figure 3).
3. Name your new database *MoviesDB.mdf* and click the **Add** button.

After you create your database, you can connect to the database by double-clicking the MoviesDB.mdf file located in the App_Data folder. Double-clicking the MoviesDB.mdf file opens the Server Explorer window.

NOTE

The Server Explorer window is named the Database Explorer window in the case of Visual Web Developer.

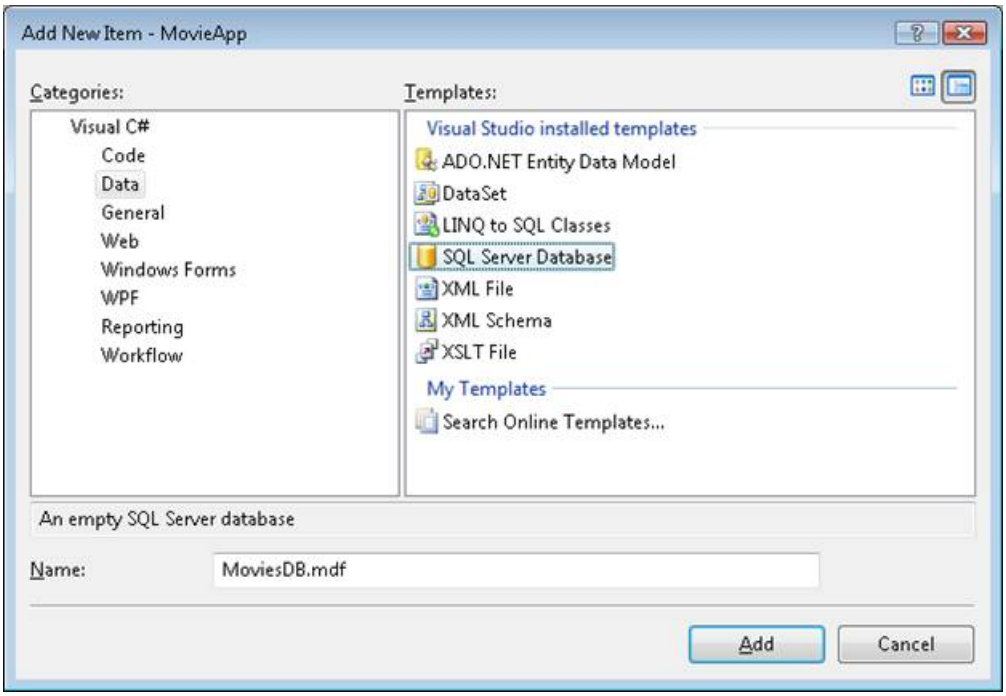


Figure 03: Creating a Microsoft SQL Server Database (Click to view full-size image)

Next, we need to create a new database table. From within the Sever Explorer window, right-click the Tables folder and select the menu option **Add New Table**. Selecting this menu option opens the database table designer. Create the following database columns:

COLUMN NAME	DATA TYPE	ALLOW NULLS
Id	Int	False
Title	Nvarchar(100)	False
Director	Nvarchar(100)	False
DateReleased	DateTime	False

The first column, the Id column, has two special properties. First, you need to mark the Id column as the primary key column. After selecting the Id column, click the **Set Primary Key** button (it is the icon that looks like a key). Second, you need to mark the Id column as an Identity column. In the Column Properties window, scroll down to the Identity Specification section and expand it. Change the **Is Identity** property to the value **Yes**. When you are finished, the table should look like Figure 4.

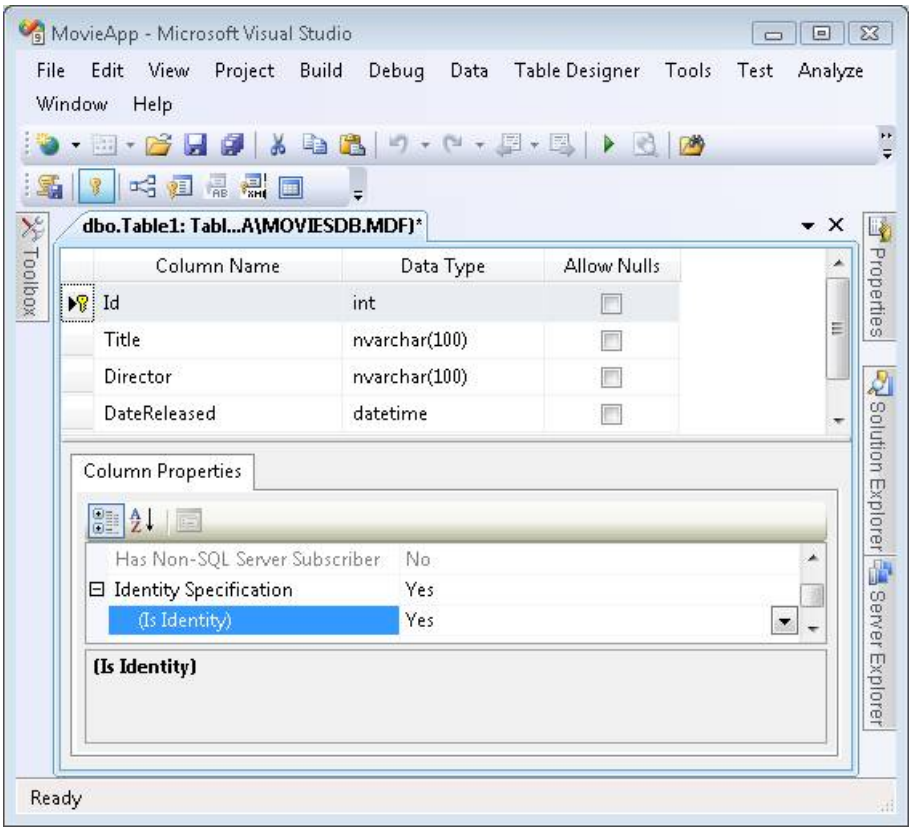


Figure 04: The Movies database table (Click to view full-size image)

The final step is to save the new table. Click the Save button (the icon of the floppy) and give the new table the name Movies.

After you finish creating the table, add some movie records to the table. Right-click the Movies table in the Server Explorer window and select the menu option **Show Table Data**. Enter a list of your favorite movies (see Figure 5).

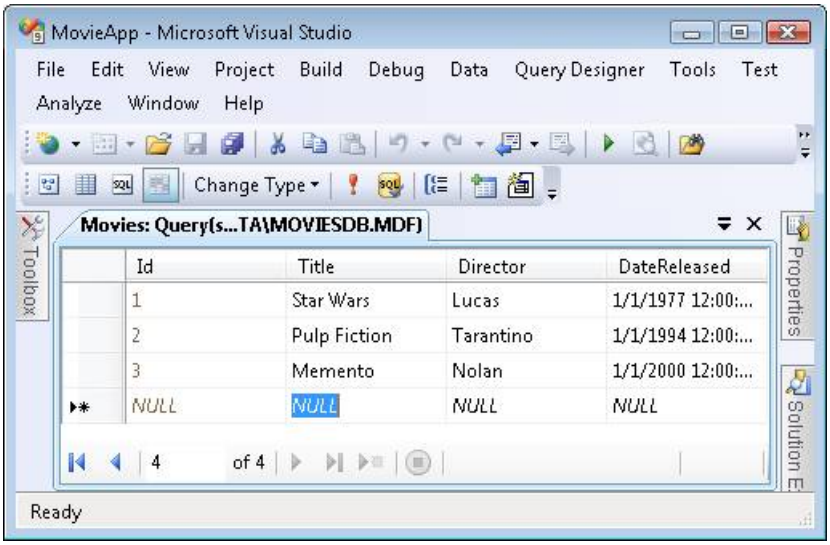


Figure 05: Entering movie records (Click to view full-size image)

Creating the Model

We next need to create a set of classes to represent our database. We need to create a database model. We'll take advantage of the Microsoft Entity Framework to generate the classes for our database model automatically.

> **NOTE**
> The ASP.NET MVC framework is not tied to the Microsoft Entity Framework. You can create your database model classes by taking advantage of a variety of Object Relational Mapping (OR/M) tools including LINQ to SQL, Subsonic, and NHibernate.

Follow these steps to launch the Entity Data Model Wizard:

1. Right-click the Models folder in the Solution Explorer window and the select the menu option **Add, New Item**.
2. Select the **Data** category and select the **ADO.NET Entity Data Model** template.
3. Give your data model the name *MoviesDBModel.edmx* and click the **Add** button.

After you click the Add button, the Entity Data Model Wizard appears (see Figure 6). Follow these steps to complete the wizard:

1. In the **Choose Model Contents** step, select the **Generate from database** option.
2. In the **Choose Your Data Connection** step, use the *MoviesDB.mdf* data connection and the name *MoviesDBEntities* for the connection settings. Click the **Next** button.
3. In the **Choose Your Database Objects** step, expand the Tables node, select the Movies table. Enter the namespace *MovieApp.Models* and click the **Finish** button.

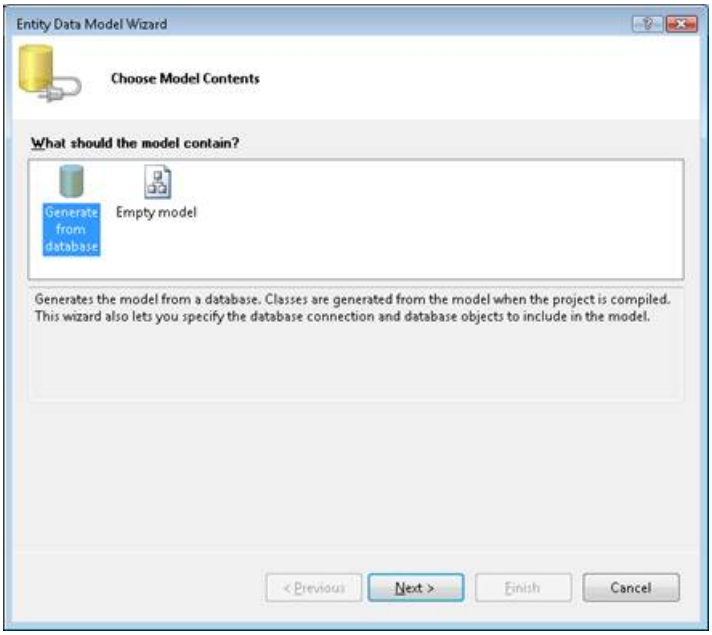


Figure 06: Generating a database model with the Entity Data Model Wizard (Click to view full-size image)

After you complete the Entity Data Model Wizard, the Entity Data Model Designer opens. The Designer should display the Movies database table (see Figure 7).

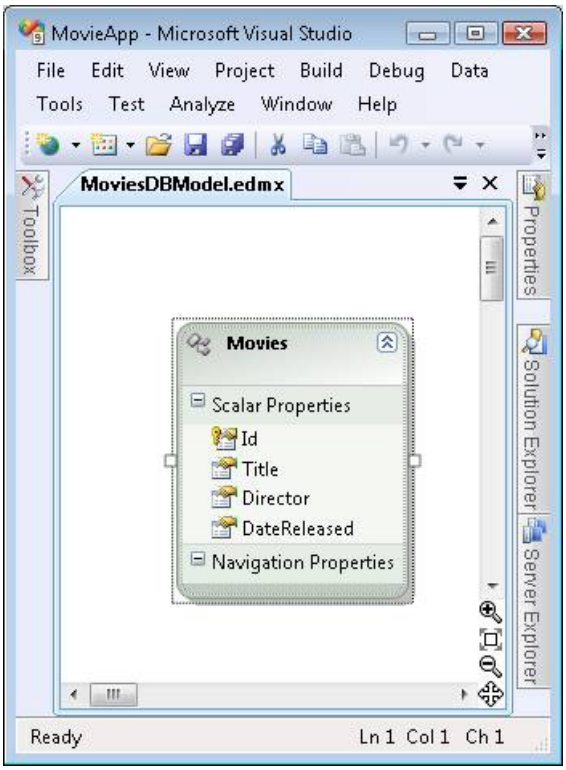


Figure 07: The Entity Data Model Designer (Click to view full-size image)

We need to make one change before we continue. The Entity Data Wizard generates a model class named Movies that represents the Movies database table. Because we'll use the Movies class to represent a particular movie, we need to modify the name of the class to be *Movie* instead of *Movies* (singular rather than plural).

Double-click the name of the class on the designer surface and change the name of the class from Movies to Movie. After making this change, click the **Save** button (the icon of the floppy disk) to generate the Movie class.

Creating the ASP.NET MVC Controller

The next step is to create the ASP.NET MVC controller. A controller is responsible for controlling how a user interacts with an ASP.NET MVC application.

Follow these steps:

1. In the Solution Explorer window, right-click the Controllers folder and select the menu option **Add, Controller**.
2. In the Add Controller dialog, enter the name *HomeController* and check the checkbox labeled **Add action methods for Create, Update, and Details scenarios** (see Figure 8).
3. Click the **Add** button to add the new controller to your project.

After you complete these steps, the controller in Listing 1 is created. Notice that it contains methods named Index, Details, Create, and Edit. In the following sections, we'll add the necessary code to get these methods to work.

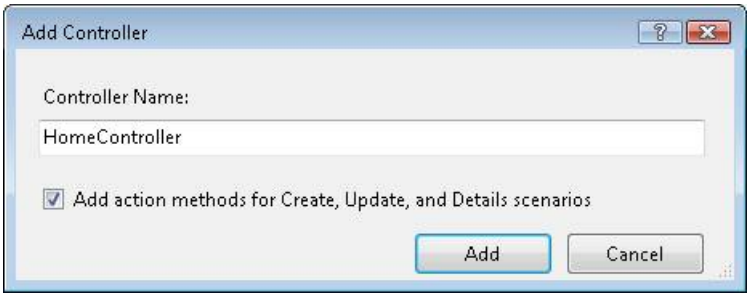


Figure 08: Adding a new ASP.NET MVC Controller (Click to view full-size image)

Listing 1 – Controllers\HomeController.vb

```
Public Class HomeController

        Inherits System.Web.Mvc.Controller

        '

        ' GET: /Home/

        Function Index() As ActionResult

            Return View()

        End Function

        '

        ' GET: /Home/Details/5

        Function Details(ByVal id As Integer) As ActionResult

            Return View()

        End Function

        '

        ' GET: /Home/Create

        Function Create() As ActionResult

            Return View()

        End Function

        '

        ' POST: /Home/Create

        <AcceptVerbs(HttpVerbs.Post)> _

        Function Create(ByVal collection As FormCollection) As ActionResult

            Try

                ' TODO: Add insert logic here

                Return RedirectToAction("Index")

            Catch

                Return View()

            End Try

        End Function

        '

        ' GET: /Home/Edit/5

        Function Edit(ByVal id As Integer) As ActionResult

            Return View()

        End Function

        '
```

```
        ' POST: /Home/Edit/5

        <AcceptVerbs(HttpVerbs.Post)> _

        Function Edit(ByVal id As Integer, ByVal collection As FormCollection) As ActionResult

            Try

                ' TODO: Add update logic here

                Return RedirectToAction("Index")

            Catch

                Return View()

            End Try

        End Function

    End Class
```

Listing Database Records

The Index() method of the Home controller is the default method for an ASP.NET MVC application. When you run an ASP.NET MVC application, the Index() method is the first controller method that is called.

We'll use the Index() method to display the list of records from the Movies database table. We'll take advantage of the database model classes that we created earlier to retrieve the movie database records with the Index() method.

I've modified the HomeController class in Listing 2 so that it contains a new private field named _db. The MoviesDBEntities class represents our database model and we'll use this class to communicate with our database.

I've also modified the Index() method in Listing 2. The Index() method uses the MoviesDBEntities class to retrieve all of the movie records from the Movies database table. The expression *_db.MovieSet.ToList()* returns a list of all of the movie records from the Movies database table.

The list of movies is passed to the view. Anything that gets passed to the View() method gets passed to the view as view data.

Listing 2 – Controllers/HomeController.vb (modified Index method)

```
Public Class HomeController

        Inherits System.Web.Mvc.Controller

        Private _db As New MoviesDBEntities()

        Function Index() As ActionResult

            Return View(_db.MovieSet.ToList())

        End Function

    End Class
```

The Index() method returns a view named Index. We need to create this view to display the list of movie database records. Follow these steps:

You should build your project (select the menu option **Build, Build Solution**) before opening the **Add View** dialog or no classes will appear in the **View data class** dropdown list.

1. Right-click the Index() method in the code editor and select the menu option **Add View** (see Figure 9).
2. In the Add View dialog, verify that the checkbox labeled **Create a strongly-typed view** is checked.
3. From the **View content** dropdown list, select the value *List*.
4. From the **View data class** dropdown list, select the value *MovieApp.Movie*.
5. Click the Add button to create the new view (see Figure 10).

After you complete these steps, a new view named Index.aspx is added to the Views\Home folder. The contents of the Index view are included in Listing 3.

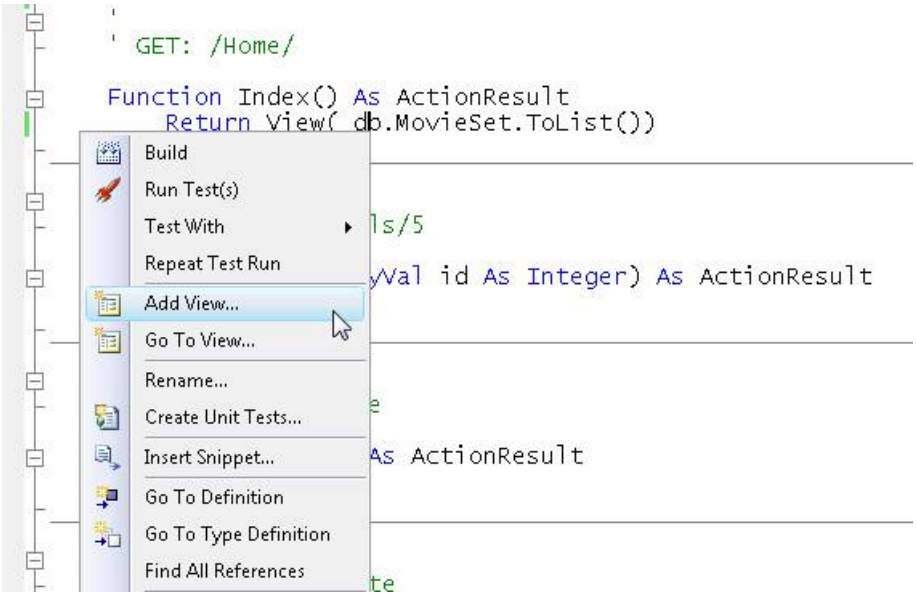


Figure 09: Adding a view from a controller action (Click to view full-size image)

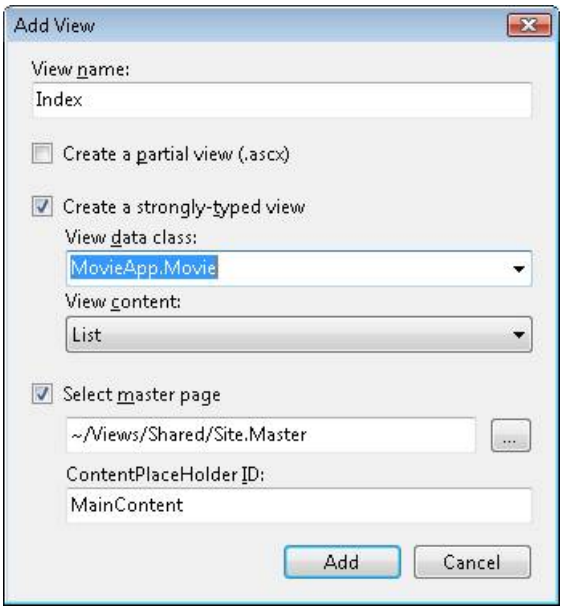


Figure 10: Creating a new view with the Add View dialog (Click to view full-size image)

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of IEnumerable (Of MovieApp.Movie))" %>

    <asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

        Index

    </asp:Content>

    <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
```

```
        <h2>Index</h2>

        <p>

            <%=Html.ActionLink("Create New", "Create")%>

        </p>

        <table>

            <tr>

                <th></th>

                <th>

                    Id

                </th>

                <th>

                    Title

                </th>

                <th>

                    Director

                </th>

                <th>

                    DateReleased

                </th>

            </tr>

        <% For Each item In Model%>

            <tr>

                <td>

                    <%=Html.ActionLink("Edit", "Edit", New With {.id = item.Id})%> |

                    <%=Html.ActionLink("Details", "Details", New With {.id = item.Id})%>

                </td>

                <td>

                    <%= Html.Encode(item.Id) %>

                </td>

                <td>

                    <%= Html.Encode(item.Title) %>

                </td>

                <td>

                    <%= Html.Encode(item.Director) %>

                </td>
```

```
                <td>
                    <%= Html.Encode(String.Format("{0:g}", item.DateReleased)) %>
                </td>
            </tr>
        <% Next%>
        </table>
    </asp:Content>
```

The Index view displays all of the movie records from the Movies database table within an HTML table. The view contains a For Each loop that iterates through each movie represented by the ViewData.Model property. If you run your application by hitting the F5 key, then you'll see the web page in Figure 11.

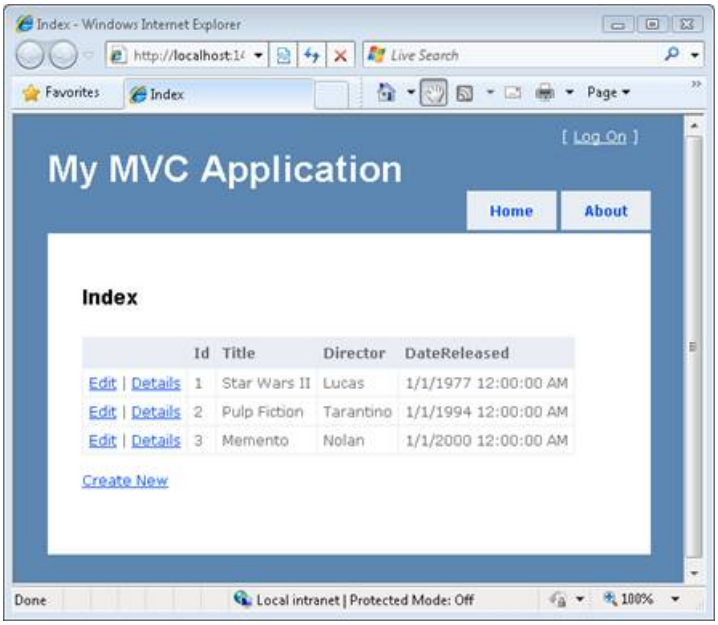


Figure 11: The Index view (Click to view full-size image)

Creating New Database Records

The Index view that we created in the previous section includes a link for creating new database records. Let's go ahead and implement the logic and create the view necessary for creating new movie database records.

The Home controller contains two methods named Create(). The first Create() method has no parameters. This overload of the Create() method is used to display the HTML form for creating a new movie database record.

The second Create() method has a FormCollection parameter. This overload of the Create() method is called when the HTML form for creating a new movie is posted to the server. Notice that this second Create() method has an AcceptVerbs attribute that prevents the method from being called unless an HTTP Post operation is performed.

This second Create() method has been modified in the updated HomeController class in Listing 4. The new version of the Create() method accepts a Movie parameter and contains the logic for inserting a new movie into the Movies database table.

NOTE

Notice the Bind attribute. Because we don't want to update the Movie Id property from HTML form, we need to explicitly exclude this property.

Listing 4 – Controllers\HomeController.vb (modified Create method)

```
Function Create() As ActionResult

            Return View()

        End Function

        <AcceptVerbs(HttpVerbs.Post)> _

        Function Create(<_bind28_exclude3a_3d_22_id22_29_> ByVal movieToCreate As Movie) As ActionResult

            If Not ModelState.IsValid Then

                Return View()

            End If

            _db.AddToMovieSet(movieToCreate)

            _db.SaveChanges()

            Return RedirectToAction("Index")

        End Function
<!--_bind28_exclude3a_3d_22_id22_29_-->
```

Visual Studio makes it easy to create the form for creating a new movie database record (see Figure 12). Follow these steps:

1. Right-click the Create() method in the code editor and select the menu option **Add View**.
2. Verify that the checkbox labeled **Create a strongly-typed view** is checked.
3. From the **View content** dropdown list, select the value *Create*.
4. From the **View data class** dropdown list, select the value *MovieApp.Movie*.
5. Click the **Add** button to create the new view.

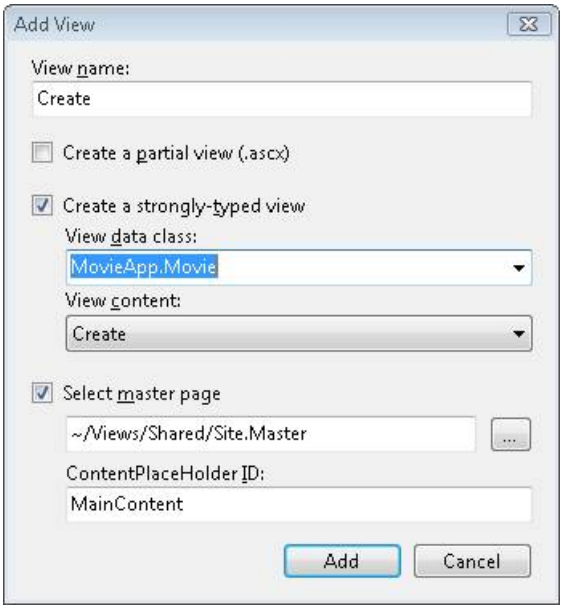


Figure 12: Adding the Create view (Click to view full-size image)

Visual Studio generates the view in Listing 5 automatically. This view contains an HTML form that includes fields that correspond to each of the properties of the Movie class.

Listing 5 – Views\Home\Create.aspx

```
<%@ Page Title="" Language="VB" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage(Of MovieApp.Movie)" %>

    <asp:Content ID="Content3" ContentPlaceHolderID="TitleContent" runat="server">

       Create

    </asp:Content>

    <asp:Content ID="Content4" ContentPlaceHolderID="MainContent" runat="server">

        <h2>Create</h2>

        <%= Html.ValidationSummary("Create was unsuccessful. Please correct the errors and try again.") %>

        <% Using Html.BeginForm()%>

            <fieldset>

                <legend>Fields</legend>

                <p>

                    <label for="Id">Id:</label>

                    <%= Html.TextBox("Id") %>

                    <%= Html.ValidationMessage("Id", "*") %>

                </p>

                <p>

                    <label for="Title">Title:</label>

                    <%= Html.TextBox("Title") %>

                    <%= Html.ValidationMessage("Title", "*") %>

                </p>

                <p>

                    <label for="Director">Director:</label>

                    <%= Html.TextBox("Director") %>

                    <%= Html.ValidationMessage("Director", "*") %>

                </p>

                <p>

                    <label for="DateReleased">DateReleased:</label>

                    <%= Html.TextBox("DateReleased") %>

                    <%= Html.ValidationMessage("DateReleased", "*") %>

                </p>
```

```
            <p>
                <input type="submit" value="Create" />
            </p>
        </fieldset>
    <% End Using %>
    <div>
        <%=Html.ActionLink("Back to List", "Index") %>
    </div>
</asp:Content>
```

NOTE

The HTML form generated by the Add View dialog generates an Id form field. Because the Id column is an Identity column, we don't need this form field and you can safely remove it.

After you add the Create view, you can add new Movie records to the database. Run your application by pressing the F5 key and click the Create New link to see the form in Figure 13. If you complete and submit the form, a new movie database record is created.

Notice that you get form validation automatically. If you neglect to enter a release date for a movie, or you enter an invalid release date, then the form is redisplayed and the release date field is highlighted.

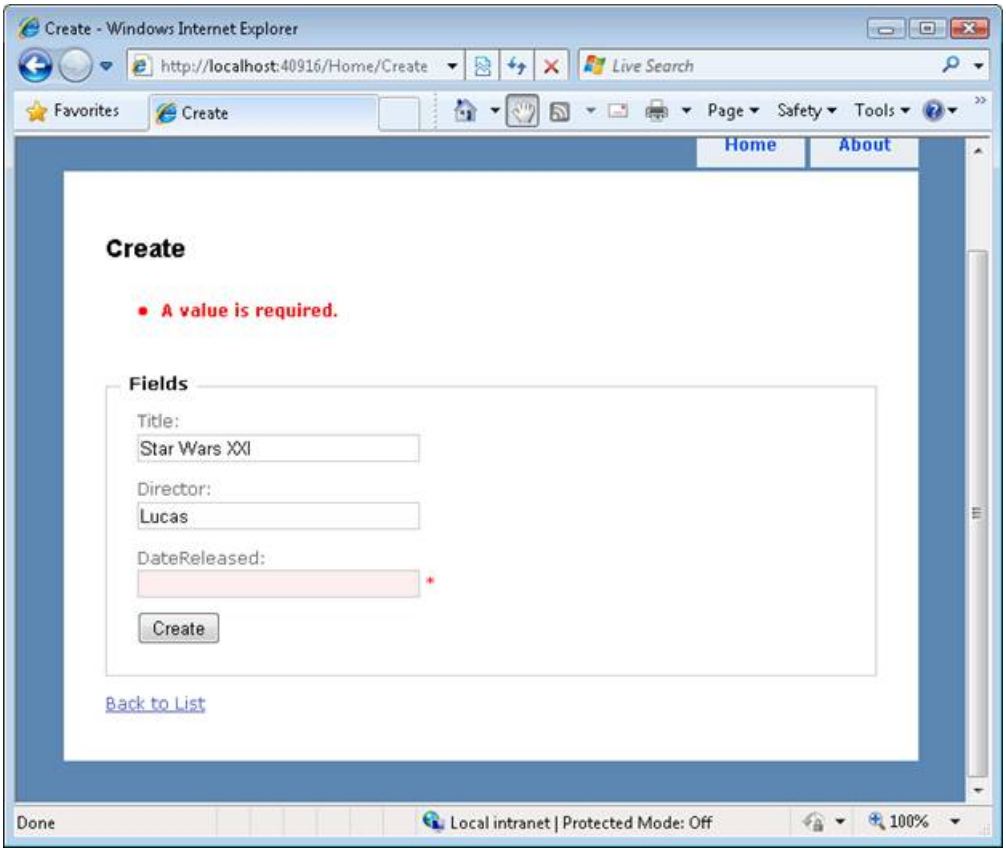


Figure 13: Creating a new movie database record (Click to view full-size image)

Editing Existing Database Records

In the previous sections, we discussed how you can list and create new database records. In this final section, we discuss how you can edit existing database records.

First, we need to generate the Edit form. This step is easy since Visual Studio will generate the Edit form for us automatically. Open the HomeController.vb class in the Visual Studio code editor and follow these steps:

1. Right-click the Edit() method in the code editor and select the menu option **Add View** (see Figure 14).
2. Check the checkbox labeled **Create a strongly-typed view**.
3. From the **View content** dropdown list, select the value *Edit*.
4. From the **View data class** dropdown list, select the value *MovieApp.Movie*.
5. Click the **Add** button to create the new view.

Completing these steps adds a new view named Edit.aspx to the Views\Home folder. This view contains an HTML form for editing a movie record.

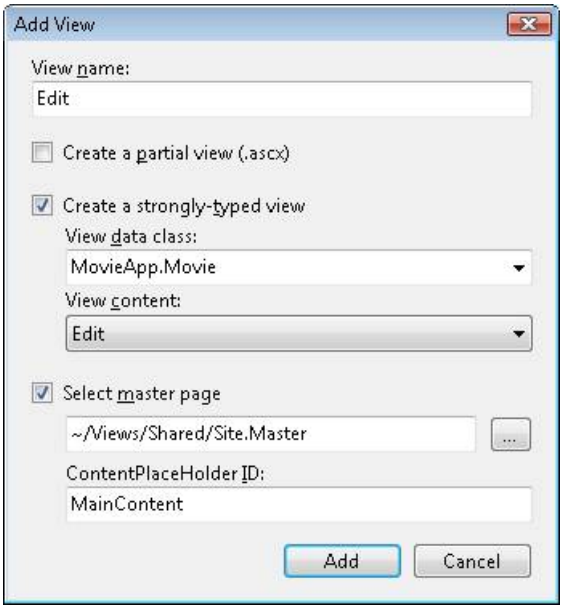


Figure 14: Adding the Edit view (Click to view full-size image)

> **NOTE**
> The Edit view contains an HTML form field that corresponds to the Movie Id property. Because you don't want people editing the value of the Id property, you should remove this form field.

Finally, we need to modify the Home controller so that it supports editing a database record. The updated HomeController class is contained in Listing 6.

Listing 6 – Controllers\HomeController.vb (Edit methods)

```
Function Edit(ByVal id As Integer) As ActionResult

            Dim movieToEdit = (From m In _db.MovieSet _

                           Where m.Id = id _

                           Select m).First()

            Return View(movieToEdit)

        End Function

        <AcceptVerbs(HttpVerbs.Post)> _

        Function Edit(ByVal movieToEdit As Movie) As ActionResult

            Dim originalMovie = (From m In _db.MovieSet _

                       Where m.Id = movieToEdit.Id _

                       Select m).First()

            If Not ModelState.IsValid Then

                Return View(originalMovie)

            End If

            _db.ApplyPropertyChanges(originalMovie.EntityKey.EntitySetName, movieToEdit)

            _db.SaveChanges()

            Return RedirectToAction("Index")

        End Function
```

In Listing 6, I've added additional logic to both overloads of the Edit() method. The first Edit() method returns the movie database record that corresponds to the Id parameter passed to the method. The second overload performs the updates to a movie record in the database.

Notice that you must retrieve the original movie, and then call ApplyPropertyChanges(), to update the existing movie in the database.

Summary

The purpose of this tutorial was to give you a sense of the experience of building an ASP.NET MVC application. I hope that you discovered that building an ASP.NET MVC web application is very similar to the experience of building an Active Server Pages or ASP.NET application.

In this tutorial, we examined only the most basic features of the ASP.NET MVC framework. In future tutorials, we dive deeper into topics such as controllers, controller actions, views, view data, and HTML helpers.

PREVIOUS

Videos

2/20/2020 • 2 minutes to read • Edit Online

- ASP.NET MVC 4
  - Using Page Inspector in ASP.NET MVC
  - ASP.NET MVC 4
- ASP.NET MVC 3
  - MVC 3 - Razor View Engine
  - MVC 3 - Razor Helpers
- ASP.NET MVC 2
  - How-Do-I
    - Creating a Tasklist Application with ASP.NET MVC
    - Creating a Movie Database Application in 15 minutes with ASP.NET MVC
    - Understanding Models, Views, and Controllers
    - ASP.NET MVC Controller Overview
    - Understanding Controllers, Controller Actions, and Action Results
    - Understanding Views, View Data, and HTML Helpers
    - Intro to URL Routing
    - Preventing JavaScript Injection Attacks
    - Creating Unit Tests for ASP.NET MVC Applications
    - Creating Custom HTML Helpers
    - Creating Model Classes with LINQ to SQL
    - Displaying a Table of Database Data
    - What is ASP.NET MVC 80 minute technical video for developers, building NerdDinner
    - Why ASP.NET MVC 3 minute overview video for decision makers
    - ASP.NET MVC: How 10 minute technical video for developers
    - Return JSON Formatted Data for an AJAX Call in an ASP.NET MVC Web Application
    - Work with Data in ASP.NET MVC Partial Views
    - Implement View Models to Manage Data for ASP.NET MVC Views
    - Create a Custom HTML Helper for an MVC Application
    - Work with Model Binders in an MVC Application
    - Use HttpVerbs Attributes in an MVC Application
    - MVC2 - Html Encoding
    - MVC2 - StronglyTyped Helpers
    - MVC2 - Model Validation
    - MVC2 - Template Customization
    - ASP.NET MVC 2 - Areas
    - ASP.NET MVC 2 - Render Action
    - 5 Minute Intro to ASP.NET MVC
    - Best Learn ASP.NET MVC
  - ASP.NET MVC For the Rest of Us

    - ASP.NET MVC For the Rest of Us: Part 1
    - ASP.NET MVC For the Rest of Us: Part 2
    - ASP.NET MVC For the Rest of Us: Part 3
    - ASP.NET MVC For the Rest of Us: Part 4
  - Music Store
    - MVC Music Store Part 1: Intro, Tools, and Project Structure
    - MVC Music Store Part 2: Controllers
    - MVC Music Store Part 3: Views and ViewModels
- ASP.NET MVC 1
  - ASP.NET MVC Storefront
    - ASP.NET MVC Storefront Part 1: Architectural Discussion and Overview
    - ASP.NET MVC Storefront Part 2: Repository Pattern
    - ASP.NET MVC Storefront Part 3: Pipes and Filters
    - ASP.NET MVC Storefront Part 4: Linq To Sql Spike
    - ASP.NET MVC Storefront Part 5: Globalization
    - ASP.NET MVC Storefront Part 6: Finishing Repository, and Initial UI Work
    - ASP.NET MVC Storefront Part 7: Routing and UI Work
    - ASP.NET MVC Storefront Part 8: Testing Controllers, Iteration 1 Complete
    - ASP.NET MVC Storefront Part 9: Shopping Cart
    - ASP.NET MVC Storefront Part 10: Shopping Cart Refactor and Authorization
    - ASP.NET MVC Storefront Part 11: Hooking Up Shopping Cart and Using Components
    - ASP.NET MVC Storefront Part 12: Mocking
    - ASP.NET MVC Storefront Part 13: Dependency Injection
    - ASP.NET MVC Storefront Part 14: Rich Client Interaction
    - ASP.NET MVC Storefront Part 15: Public Code review
    - ASP.NET MVC Storefront Part 16: Membership Redo With OpenID
    - ASP.NET MVC Storefront Part 17: Checkout With Jeff Atwood
    - ASP.NET MVC Storefront Part 18: Creating Experience
    - ASP.NET MVC Storefront Part 19: Processing Orders With Windows Workflow
    - ASP.NET MVC Storefront Part 19a: Windows Workflow Followup
    - ASP.NET MVC Storefront Part 20: Logging
    - ASP.NET MVC Storefront Part 21: Order Manager and Personalization
    - ASP.NET MVC Storefront Part 22: Restructuring, Rerouting, and PayPal
    - ASP.NET MVC Storefront Part 23: Getting Started With Domain-Driven Design
    - ASP.NET MVC Storefront Part 24: Finis
  - Talks
    - Creating NerdDinner.com with Microsoft ASP.NET Model View Controller (MVC)
    - America's Next Top Model View Controller Framework
    - Ninja on Fire Black Belt Tips

ASP.NET MVC 4

2/20/2020 • 2 minutes to read • Edit Online

ASP.NET MVC 4

- Using Page Inspector in ASP.NET MVC
- ASP.NET MVC 4

Using Page Inspector in ASP.NET MVC

2/20/2020 • 2 minutes to read • Edit Online

by Tim Ammann

The videos explores the usage of Page Inspector in ASP.NET MVC web applications. Read the tutorial

Watch video (6 minutes)

ASP.NET MVC 4

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Watch video (7 minutes)

ASP.NET MVC 3

2/20/2020 • 2 minutes to read • Edit Online

ASP.NET MVC 3 Quick Hits

- MVC 3 - Razor View Engine
- MVC 3 - Razor Helpers

MVC 3 - Razor View Engine

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

In this video you will learn the basics of the new Razor view engine included in ASP.NET MVC 3.

Watch video (3 minutes)

MVC 3 - Razor Helpers

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

In this video, you will learn how to use both extension methods and declarative @helper syntax in Razor views.

Watch video (8 minutes)

ASP.NET MVC 2

2/20/2020 • 2 minutes to read • Edit Online

ASP.NET MVC 2

- How-Do-I
  - Creating a Tasklist Application with ASP.NET MVC
  - Creating a Movie Database Application in 15 minutes with ASP.NET MVC
  - Understanding Models, Views, and Controllers
  - ASP.NET MVC Controller Overview
  - Understanding Controllers, Controller Actions, and Action Results
  - Understanding Views, View Data, and HTML Helpers
  - Intro to URL Routing
  - Preventing JavaScript Injection Attacks
  - Creating Unit Tests for ASP.NET MVC Applications
  - Creating Custom HTML Helpers
  - Creating Model Classes with LINQ to SQL
  - Displaying a Table of Database Data
  - What is ASP.NET MVC 80 minute technical video for developers, building NerdDinner
  - Why ASP.NET MVC 3 minute overview video for decision makers
  - ASP.NET MVC: How 10 minute technical video for developers
  - Return JSON Formatted Data for an AJAX Call in an ASP.NET MVC Web Application
  - Work with Data in ASP.NET MVC Partial Views
  - Implement View Models to Manage Data for ASP.NET MVC Views
  - Create a Custom HTML Helper for an MVC Application
  - Work with Model Binders in an MVC Application
  - Use HttpVerbs Attributes in an MVC Application
  - MVC2 - Html Encoding
  - MVC2 - StronglyTyped Helpers
  - MVC2 - Model Validation
  - MVC2 - Template Customization
  - ASP.NET MVC 2 - Areas
  - ASP.NET MVC 2 - Render Action
  - 5 Minute Intro to ASP.NET MVC
  - Best Learn ASP.NET MVC
- ASP.NET MVC For the Rest of Us
  - ASP.NET MVC For the Rest of Us: Part 1
  - ASP.NET MVC For the Rest of Us: Part 2
  - ASP.NET MVC For the Rest of Us: Part 3
  - ASP.NET MVC For the Rest of Us: Part 4
- Music Store
  - MVC Music Store Part 1: Intro, Tools, and Project Structure

- MVC Music Store Part 2: Controllers
- MVC Music Store Part 3: Views and ViewModels

How-Do-I

2/20/2020 • 2 minutes to read • Edit Online

Videos in this section cover ASP.NET MVC, including how to build "The Storefront" - an entire e-commerce application from start to finish.

- Creating a Tasklist Application with ASP.NET MVC
- Creating a Movie Database Application in 15 minutes with ASP.NET MVC
- Understanding Models, Views, and Controllers
- ASP.NET MVC Controller Overview
- Understanding Controllers, Controller Actions, and Action Results
- Understanding Views, View Data, and HTML Helpers
- Intro to URL Routing
- Preventing JavaScript Injection Attacks
- Creating Unit Tests for ASP.NET MVC Applications
- Creating Custom HTML Helpers
- Creating Model Classes with LINQ to SQL
- Displaying a Table of Database Data
- What is ASP.NET MVC 80 minute technical video for developers, building NerdDinner
- Why ASP.NET MVC 3 minute overview video for decision makers
- ASP.NET MVC: How 10 minute technical video for developers
- Return JSON Formatted Data for an AJAX Call in an ASP.NET MVC Web Application
- Work with Data in ASP.NET MVC Partial Views
- Implement View Models to Manage Data for ASP.NET MVC Views
- Create a Custom HTML Helper for an MVC Application
- Work with Model Binders in an MVC Application
- Use HttpVerbs Attributes in an MVC Application
- MVC2 - Html Encoding
- MVC2 - StronglyTyped Helpers
- MVC2 - Model Validation
- MVC2 - Template Customization
- ASP.NET MVC 2 - Areas
- ASP.NET MVC 2 - Render Action
- 5 Minute Intro to ASP.NET MVC
- Best Learn ASP.NET MVC

Creating a Tasklist Application with ASP.NET MVC

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

Stephen Walther builds an entire ASP.NET MVC application from start to finish. This tutorial is a great introduction for people who are new to the ASP.NET MVC Framework and who want to get a sense of the process of building an ASP.NET MVC application.

Watch video (38 minutes)

NEXT

Creating a Movie Database Application in 15 minutes with ASP.NET MVC

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

Stephen Walther builds an entire ASP.NET MVC application from start to finish. This video is a great introduction for people who are new to the ASP.NET MVC Framework and who want to get a sense of the process of building an ASP.NET MVC application.

Watch video (12 minutes)

PREVIOUS NEXT

Understanding Models, Views, and Controllers

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

Confused about Models, Views, and Controllers? In this tutorial, Stephen Walther introduces you to the different parts of an ASP.NET MVC application. For additional information about this video, read the "Understanding Models, Views, and Controllers" tutorial (C#, VB).

Watch video (13 minutes)

PREVIOUS NEXT

ASP.NET MVC Controller Overview

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

In this video, Stephen Walther introduces you to ASP.NET MVC controllers and controller actions. You learn how to return different types of action results.

Watch video (5 minutes)

PREVIOUS NEXT

Understanding Controllers, Controller Actions, and Action Results

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

In this tutorial, Stephen Walther introduces you to ASP.NET MVC controllers. You learn how to create new controllers and return different types of action results. For additional information about this video, read the "Understanding Controllers, Controller Actions, and Action Results" tutorial (C#, VB).

Watch video (11 minutes)

PREVIOUS NEXT

Understanding Views, View Data, and HTML Helpers

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

What is an ASP.NET MVC View and how does it differ from a HTML page? In this tutorial, Stephen Walther introduces you to Views and demonstrates how you can take advantage of View Data and HTML Helpers within a View. For additional information about this video, read the "Understanding Views, View Data, and HTML Helpers" tutorial (C#, VB).

Watch video (11 minutes)

PREVIOUS NEXT

An Introduction to URL Routing

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

In this tutorial, Stephen Walther demonstrates how browser requests get mapped to controller actions through URL Routing. You also learn how to create a custom route that displays different blog entries depending on the date passed in a URL. For additional information about this video, read the "An Introduction to URL Routing" tutorial (C#, VB).

Watch video (14 minutes)

PREVIOUS NEXT

Preventing JavaScript Injection Attacks

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

Prevent JavaScript Injection Attacks and Cross-Site Scripting Attacks from happening to you. In this tutorial, Stephen Walther explains how you can easily defeat these types of attacks by HTML encoding your content. For additional information about this video, read the "Preventing JavaScript Injection Attacks" tutorial (C#, VB).

Watch video (11 minutes)

PREVIOUS NEXT

Creating Unit Tests for ASP.NET MVC Applications

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

Learn how to create unit tests for controller actions. In this tutorial, Stephen Walther demonstrates how to test whether a controller action returns a particular view, returns a particular set of data, or returns a different type of action result. For additional information about this video, read the "Creating Unit Tests for ASP.NET MVC Applications" tutorial (C#, VB).

Watch video (15 minutes)

PREVIOUS NEXT

Creating Custom HTML Helpers

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

Create new HTML Helpers to make it easier to generate view content. Stephen Walther demonstrates two simple methods of creating new HTML Helpers.

Watch video (14 minutes)

PREVIOUS NEXT

Creating Model Classes with LINQ to SQL

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

Learn how to create data access classes by taking advantage of Microsoft LINQ to SQL. In this video tutorial, you learn how to use a LINQ to SQL DataContext class directly within a controller. You also learn how to use the Repository Pattern to create testable and flexible database-driven applications.

Watch video (24 minutes)

PREVIOUS NEXT

Displaying a Table of Database Data

2/20/2020 • 2 minutes to read • Edit Online

by Stephen Walther

Need to display a set of database records in an ASP.NET MVC view? Learn two methods of displaying records in an HTML table. You learn how to perform all of the formatting inline. You also learn how to create a template with a ASP.NET MVC user control.

Watch video (15 minutes)

PREVIOUS NEXT

What is ASP.NET MVC? 80 minute technical video for developers, building NerdDinner

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

An in-depth tutorial on building an application with ASP.NET MVC. In this video we build the basics of Nerddinner.com and cover issues such as Unit Testing, Javascript (using jQuery), and how to use the new tooling features for Visual Studio 2008 which are installed with the ASP.NET MVC project templates.

Watch video (80 minutes)

PREVIOUS NEXT

Why ASP.NET MVC? 3 minute overview video for decision makers

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

A high-level overview of ASP.NET MVC for decision-makers who would like to know more about benefits of ASP.NET MVC.

Watch video (3 minutes)

PREVIOUS NEXT

ASP.NET MVC: How? 10 minute technical video for developers

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

A short overview of what it's like to work with ASP.NET MVC.

Watch video (11 minutes)

PREVIOUS NEXT

How Do I: Return JSON Formatted Data for an AJAX Call in an ASP.NET MVC Web Application?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to use the JsonResult and Json classes to return instances of classes as JSON formatted data. First, a sample MVC web application is created and a controller and associated view are added for customer information. Then a simple customer class is created as part of the data model. A GetCustomer() function is then added to the customer controller to retrieve data for a single customer in JSON format. Finally, script is added to the customer view that uses jQuery to make an AJAX call to GetCustomer() to retrieve the customer data in JSON format

Watch video (15 minutes)

PREVIOUS NEXT

How Do I: Work with Data in ASP.NET MVC Partial Views?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to organize and pass data to partial views in an MVC application. A sample MVC application is created that contains domain models for several business entities as well as a view model that packages instances of the classes. A demo view is then created along with several partial views, all of which are strongly typed and make use of the view model class. The partial views are then added to the demo view and the instance of the view model is used to easily pass the appropriate data down to the partial views. Finally, the uses of the concepts implemented in the sample in a more extensive MVC application are discussed.

Watch video (20 minutes)

PREVIOUS NEXT

How Do I: Implement View Models to Manage Data for ASP.NET MVC Views?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to use the concept of a "View Model" for managing data for ASP.NET MVC views. First, a sample MVC project is created that uses a ViewDataDictionary to package and pass data from a controller to a view. Next, a class is created which represents a view model for a view in the sample MVC project. As part of this implementation, the advantages of the view model approach are discussed. Finally, there is a discussion of how to partition the application logic when using a view model implementation.

Watch video (21 minutes)

PREVIOUS NEXT

How Do I: Create a Custom HTML Helper for an MVC Application?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to create a custom HtmlHelper that is not available in the standard set in an MVC application. First, a sample MVC application is created with a demo controller and view to test the custom HtmlHelper. Next, a module is created with a public function that is an extension method which represents the implementation of the custom HtmlHelper. The custom helper is for creating `<img>` tags in a page and receives several inbound parameters including the id, url, and alt text for the image tag. The logic is then added to the function for returning the completed `<img>` tag with the specified information. Then the custom HtmlHelper is used on the demo page to display an image. Finally, the custom HtmlHelper is expanded to include multiple constructor overrides which provide flexibility for more easily creating different `<img>` tags.

Watch video (18 minutes)

PREVIOUS NEXT

How Do I: Work with Model Binders in an MVC Application?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to use model binders to bind form data to complex business objects in models in an MVC application. First, an MVC application is created along with a simple model for a Customer class along with a controller and view for editing the customer data. Then, the automatic model binding of MVC is demonstrated with the customer class. Next, the Bind attribute is discussed and demonstrated as a mechanism for controlling the details of the binding process to include and exclude specific properties of a complex object. Finally, the use of namespace prefixes for segmenting form data on a page is discussed.

Watch video (17 minutes)

PREVIOUS NEXT

How Do I: Use HttpVerbs Attributes in an MVC Application?

2/20/2020 • 2 minutes to read • Edit Online

by Chris Pels

In this video Chris Pels shows how to use the HttpVerbs attributes to control access to MVC actions. First, a sample application is created with a default controller and view for editing the information. Next, a second Index action is added to the controller which has an HttpPost attribute which restricts it to being called only when an HTTP POST is used. As a follow-up, the AcceptVerbs() attribute is implemented as an alternative syntax for Visual Studio 2008. A use of the HttpVerbs for preventing the security risk associated with using an HTTP GET to perform a delete from a link is then discussed.

Watch video (16 minutes)

PREVIOUS NEXT

MVC2 - Html Encoding

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

In this video you will learn how to use the new HTML Encoding shortcut syntax in ASP.NET to guard against script injection, and to use the MvcHtmlString when you want to include HTML in your strings.

Watch video (2 minutes)

PREVIOUS NEXT

MVC2 - StronglyTyped Helpers

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

In this video you will learn about how you can use strongly typed HTML Helpers to gain compile-type checking when creating HTML UI Elements.

Watch video (2 minutes)

PREVIOUS NEXT

MVC2 - Model Validation

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

In this video you will learn about how to use Data Annotation Attributes to define your validation rules, and how to enable client-side (AJAX) validation.

Watch video (4 minutes)

PREVIOUS NEXT

MVC2 - Template Customization

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

In this video you will learn about the template customization features in ASP.NET MVC 2. We'll look at Html.EditorForModel, Html.DisplayForModel, Model Template customization, and DataType customization.

Watch video (3 minutes)

PREVIOUS NEXT

ASP.NET MVC 2 - Areas

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

In this video you will learn how to use the Areas feature to make it easier to organize larger ASP.NET MVC applications by breaking them into different functional areas.

Watch video (3 minutes)

PREVIOUS NEXT

ASP.NET MVC 2 - Render Action

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

In this video you'll learn how to use Html.RenderAction to write the results of a Controller Action to a View. We'll look at how Html.RenderAction differs from Html.RenderPartial, and how Html.Action differs from Html.RenderAction.

Watch video (3 minutes)

PREVIOUS NEXT

5 Minute Introduction to ASP.NET MVC

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Scott Hanselman presents a 5 minute introduction to ASP.NET MVC.

Watch video (8 minutes)

PREVIOUS NEXT

How to Best Learn ASP.NET MVC

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Microsoft's Scott Hanselman shows you what you need to learn how to use ASP.NET MVC to develop your web applications.

Watch video (3 minutes)

PREVIOUS

ASP.NET MVC For the Rest of Us

2/20/2020 • 2 minutes to read • Edit Online

ASP.NET MVC For the Rest of Us

- ASP.NET MVC For the Rest of Us: Part 1
- ASP.NET MVC For the Rest of Us: Part 2
- ASP.NET MVC For the Rest of Us: Part 3
- ASP.NET MVC For the Rest of Us: Part 4

ASP.NET MVC For the Rest of Us: Part 1

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this Video Microsoft's Joe Stagner builds a "Contact Us" feature for an ASP.NET MVC application by adding a View, Controller Logic and a Custom Model to the default ASP.NET Application. You'll see some of the fundamental differences between working with ASP.NET MVC versus ASP.NET Web Forms.

Watch video (36 minutes)

NEXT

ASP.NET MVC For the Rest of Us: Part 2

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this Video Microsoft's Joe Stagner builds a "Contact Us" feature for an ASP.NET MVC application by adding a View, Controller Logic and a Custom Model to the default ASP.NET Application. You'll see some of the fundamental differences between working with ASP.NET MVC versus ASP.NET Web Forms.

Watch video (31 minutes)

PREVIOUS NEXT

ASP.NET MVC For the Rest of Us: Part 3

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this Video Microsoft's Joe Stagner rebuilds the "Contact Us" feature for an ASP.NET MVC application using the Microsoft Entity Framework and a strongly typed view. Here we start to get a real feel for the power of working with data in ASP.NET MVC.

Watch video (16 minutes)

PREVIOUS NEXT

ASP.NET MVC For the Rest of Us: Part 4

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

In this video Microsoft's Joe Stagner adds a controller to an ASP.NET MVC application.

Watch video (5 minutes)

PREVIOUS

Music Store

2/20/2020 • 2 minutes to read • Edit Online

Build the ASP.NET MVC Music Store sample application

- MVC Music Store Part 1: Intro, Tools, and Project Structure
- MVC Music Store Part 2: Controllers
- MVC Music Store Part 3: Views and ViewModels

MVC Music Store Part 1: Intro, Tools, and Project Structure

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

Jon Galloway begins the MVC Music Store tutorial with an introduction, overview of the application, and review of the directory structure of an MVC application.

Watch video (6 minutes)

NEXT

MVC Music Store Part 2: Controllers

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

Jon continues the MVC Music Store tutorial by creating some basic controller actions.

Watch video (6 minutes)

PREVIOUS NEXT

MVC Music Store Part 3: Views and ViewModels

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

Jon continues the MVC Music Store tutorial with an introduction to Views, ViewModels, and MasterPages.

Watch video (13 minutes)

PREVIOUS

ASP.NET MVC 1

2/20/2020 • 2 minutes to read • Edit Online

ASP.NET MVC 1

- ASP.NET MVC Storefront
  - ASP.NET MVC Storefront Part 1: Architectural Discussion and Overview
  - ASP.NET MVC Storefront Part 2: Repository Pattern
  - ASP.NET MVC Storefront Part 3: Pipes and Filters
  - ASP.NET MVC Storefront Part 4: Linq To Sql Spike
  - ASP.NET MVC Storefront Part 5: Globalization
  - ASP.NET MVC Storefront Part 6: Finishing Repository, and Initial UI Work
  - ASP.NET MVC Storefront Part 7: Routing and UI Work
  - ASP.NET MVC Storefront Part 8: Testing Controllers, Iteration 1 Complete
  - ASP.NET MVC Storefront Part 9: Shopping Cart
  - ASP.NET MVC Storefront Part 10: Shopping Cart Refactor and Authorization
  - ASP.NET MVC Storefront Part 11: Hooking Up Shopping Cart and Using Components
  - ASP.NET MVC Storefront Part 12: Mocking
  - ASP.NET MVC Storefront Part 13: Dependency Injection
  - ASP.NET MVC Storefront Part 14: Rich Client Interaction
  - ASP.NET MVC Storefront Part 15: Public Code review
  - ASP.NET MVC Storefront Part 16: Membership Redo With OpenID
  - ASP.NET MVC Storefront Part 17: Checkout With Jeff Atwood
  - ASP.NET MVC Storefront Part 18: Creating Experience
  - ASP.NET MVC Storefront Part 19: Processing Orders With Windows Workflow
  - ASP.NET MVC Storefront Part 19a: Windows Workflow Followup
  - ASP.NET MVC Storefront Part 20: Logging
  - ASP.NET MVC Storefront Part 21: Order Manager and Personalization
  - ASP.NET MVC Storefront Part 22: Restructuring, Rerouting, and PayPal
  - ASP.NET MVC Storefront Part 23: Getting Started With Domain-Driven Design
  - ASP.NET MVC Storefront Part 24: Finis
- Talks
  - Creating NerdDinner.com with Microsoft ASP.NET Model View Controller (MVC)
  - America's Next Top Model View Controller Framework
  - Ninja on Fire Black Belt Tips

ASP.NET MVC Storefront

2/20/2020 • 2 minutes to read • Edit Online

ASP.NET MVC Storefront Starter Kit

- ASP.NET MVC Storefront Part 1: Architectural Discussion and Overview
- ASP.NET MVC Storefront Part 2: Repository Pattern
- ASP.NET MVC Storefront Part 3: Pipes and Filters
- ASP.NET MVC Storefront Part 4: Linq To Sql Spike
- ASP.NET MVC Storefront Part 5: Globalization
- ASP.NET MVC Storefront Part 6: Finishing Repository, and Initial UI Work
- ASP.NET MVC Storefront Part 7: Routing and UI Work
- ASP.NET MVC Storefront Part 8: Testing Controllers, Iteration 1 Complete
- ASP.NET MVC Storefront Part 9: Shopping Cart
- ASP.NET MVC Storefront Part 10: Shopping Cart Refactor and Authorization
- ASP.NET MVC Storefront Part 11: Hooking Up Shopping Cart and Using Components
- ASP.NET MVC Storefront Part 12: Mocking
- ASP.NET MVC Storefront Part 13: Dependency Injection
- ASP.NET MVC Storefront Part 14: Rich Client Interaction
- ASP.NET MVC Storefront Part 15: Public Code review
- ASP.NET MVC Storefront Part 16: Membership Redo With OpenID
- ASP.NET MVC Storefront Part 17: Checkout With Jeff Atwood
- ASP.NET MVC Storefront Part 18: Creating Experience
- ASP.NET MVC Storefront Part 19: Processing Orders With Windows Workflow
- ASP.NET MVC Storefront Part 19a: Windows Workflow Followup
- ASP.NET MVC Storefront Part 20: Logging
- ASP.NET MVC Storefront Part 21: Order Manager and Personalization
- ASP.NET MVC Storefront Part 22: Restructuring, Rerouting, and PayPal
- ASP.NET MVC Storefront Part 23: Getting Started With Domain-Driven Design
- ASP.NET MVC Storefront Part 24: Finis

ASP.NET MVC Storefront Part 1: Architectural Discussion and Overview

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

Rob Conery covers the initial architecture here, in part 1, where he discusses the Repository Pattern with Ayende Rahien and Steve Harman. He also asks Phil Haack what's first: The Test Chicken or the Model Egg.

Watch video (10 minutes)

NEXT

ASP.NET MVC Storefront Part 2: The Repository Pattern

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

Rob Conery walks through Repository implementation and writes out the first main set of unit tests. He will also structure up the initial service layer methods and interfaces.

Watch video (10 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 3: Pipes and Filters

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

Rob Conery discusses the feedback from Parts 1 and 2, and then dive into the Pipes and Filters implementation on the IQueryable repository.

Watch video (16 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 4: Linq To Sql Spike

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

Rob creates a "spike" - a bit of functionality to test a thought or inspiration - to make sure that Linq To Sql will work for my data access pattern.

Watch video (14 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 5: Globalization

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

Rob Conery works with Damien Guard to refactor the database to deal with test data and globalization issues.

Watch video (11 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 6: Finishing The Repository, and Initial UI Work

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

Rob talks with Jon Galloway about CSS strategies and begin working in the Front end of the application.

Watch video (20 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 7: Routing and UI Work

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

Rob talks with Scott Hanselman about how to setup Routing, and then he implements the user interface.

Watch video (25 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 8: Testing Controllers, Iteration 1 Complete

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

Rob finishes up the initial iteration of the site and work on Controller Tests as well.

Watch video (30 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 9: The Shopping Cart

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

Rob Conery cooks up a simple cart system so he can spike.

Watch video (28 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 10: Shopping Cart Refactor and Authorization

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

Rob refactors the initial go at the Shopping Cart, and implements the core of a Membership System using ASP.NET Membership.

Watch video (25 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 11: Hooking Up The Shopping Cart and Using Components

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

In this episode I hook the site up to the shopping cart and run some tests on my Controllers. I also build a Component with a little-known feature of MVC, the System.Web.MVC.ComponentController.

Watch video (31 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 12: Mocking

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

In this episode I implement mocking so I can test my Authentication action for my UserController.

Watch video (18 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 13: Dependency Injection

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

This is a joint development session with Jeremy Miller of CodeBetter/StructureMap, where he shows me the ins and outs of Dependency Injection.

Watch video (58 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 14: Rich Client Interaction

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

Rich Client interaction specifically focused on AJAX. Rob Conery has documented each step in the development process on his blog.

Watch video (33 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 15: Public Code review

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

In this episode Rob Conery does a public code review of the Storefront Application based on user feedback. Rob Conery has documented each step in the development process on his blog.

Watch video (37 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 16: Membership Redo With OpenID

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

Rob changes the Membership system and splits it out into three separate services/parts. Rob then sits with Jon Galloway to talk about OpenID and how to implement.

Watch video (40 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 17: Checkout With Jeff Atwood

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

In this episode I ready up the checkout process and talk with Mr. CodingHorror himself, Jeff Atwood. We talk about all kinds of things but mostly discuss Open ID, CardSpace, and how Jeff would like to see the checkout process work in an ideal world. I then walk you through setting up CardSpace on your local machine.

Watch video (34 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 18: Creating An Experience

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

In this episode I work on pulling all of the technical pieces together to form a coherent "experience" for the shopper.

Watch video (21 minutes)

PREVIOUS NEXT

ASP.NET MVC MVC Storefront Part 19: Processing Orders With Windows Workflow

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

In this episode I introduce Windows Workflow and spike it to see how well it will work for various processes in the site.

Watch video (43 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 19a: Windows Workflow Followup

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

In this episode I followup with more information about using Windows Workflow, and I touch on some of the feedback I received with episode 19.

Watch video (13 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 20: Logging

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

In this episode I plug in logging - something that probably should have been implemented sooner.

Watch video (21 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 21: Order Manager and Personalization

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

In this episode I create the Order Manager page and talk about various issues relating to order fulfillment. In addition I introduce personalization features to make the experience a bit more friendly for the user.

Watch video (43 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 22: Restructuring, Rerouting, and PayPal

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

In this episode I restructure a bit in order to clean things up some. I also change the routes to be a lot simpler. Finally, I add PayPal as a checkout option.

Watch video (45 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 23: Getting Started With Domain-Driven Design

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

I don't like the complexity that's crept into my application. Not sure how else to put it - but the architecture needs to be shaken a bit so in this episode I tackle this complexity head on using Domain-Driven Design (DDD).

Watch video (46 minutes)

PREVIOUS NEXT

ASP.NET MVC Storefront Part 24: Finis

2/20/2020 • 2 minutes to read • Edit Online

by Rob Conery

This is the last episode in the MVC Storefront series and this might seem a bit odd. It's definitely not the last screencast I'll do with respect to ASP.NET MVC, but I think I've gone about as far as I can go with this series.

Watch video (27 minutes)

PREVIOUS

Talks

2/20/2020 • 2 minutes to read • Edit Online

ASP.NET MVC 1 Talks

- Creating NerdDinner.com with Microsoft ASP.NET Model View Controller (MVC)
- America's Next Top Model View Controller Framework
- Ninja on Fire Black Belt Tips

Creating NerdDinner.com with Microsoft ASP.NET Model View Controller (MVC)

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

No slides. Just code. Start with File-New and build a real Web site with ASP.NET, ASP.NET AJAX, Authentication, Authorization, MVC, Microsoft SQL Server and jQuery in sixty-five minutes or less.

Watch video (71 minutes)

America's Next Top Model View Controller Framework

2/20/2020 • 2 minutes to read • Edit Online

by Phil Haack

See how to build a simple application that highlights some of the key capabilities of ASP.NET MVC. Also, learn how to improve productivity using some of the cool new tooling features.

Watch video (70 minutes)

Ninja on Fire Black Belt Tips

2/20/2020 • 2 minutes to read • Edit Online

by Phil Haack

See a walkthrough of several tips and tricks for getting the most out of the ASP.NET MVC framework.

Watch video (77 minutes)

ASP.NET MVC 4

2/20/2020 • 2 minutes to read • Edit Online

Top Features

- ASP.NET Web API
- Refreshed and modernized default project templates
- New mobile project template
- Many new features to support mobile apps
- Enhanced support for asynchronous methods
- *Read the full feature list in the release notes*

What's New in ASP.NET MVC 4

- ASP.NET Web API
- Tutorial: ASP.NET MVC 4 Mobile Features
- Release Notes for ASP.NET MVC 4
- New ASP.NET MVC 4 tutorial series

Install ASP.NET MVC 4 for Visual Studio 2010

- Download the standalone ASP.NET MVC 4 installer executable

Install Visual Studio 2012 (includes ASP.NET MVC 4)

- Visual Studio 2012 Ultimate

Getting Help and Reporting Issues

- Known issues and breaking changes list
- Get help and discuss ASP.NET MVC 4 in the forums
- Report a bug in ASP.NET MVC 4
- Make a feature request

ASP.NET MVC 3

4/6/2020 • 14 minutes to read • Edit Online

(includes April 2011 Tools Update)

ASP.NET MVC 3 is a framework for building scalable, standards-based web applications using well-established design patterns and the power of ASP.NET and the .NET Framework.

It installs side-by-side with ASP.NET MVC 2, so get started using it today!

Download the installer here

Top Features

- Integrated Scaffolding system extensible via NuGet
- HTML 5 enabled project templates
- Expressive Views including the new Razor View Engine
- Powerful hooks with Dependency Injection and Global Action Filters
- Rich JavaScript support with unobtrusive JavaScript, jQuery Validation, and JSON binding
- *Read the full feature list below*

Top Links

What's New in ASP.NET MVC 3

- Phil Haack: ASP.NET MVC 3 Released
- Scott Hanselman: ASP.NET MVC3, WebMatrix, NuGet, IIS Express and Orchard released - The Microsoft January Web Release in Context
- Scott Guthrie: Announcing release of ASP.NET MVC 3, IIS Express, SQL CE 4, Web Farm Framework, Orchard, WebMatrix
- Release Notes for ASP.NET MVC 3

Installation and Help

- Install ASP.NET MVC 3 using the Web Platform Installer (recommended)
- Install ASP.NET MVC 3 using the installer executable
- Install ASP.NET MVC 3 for Visual Studio 11 Developer Preview
- Read the Intro to ASP.NET MVC 3 tutorial
- Get help and discuss ASP.NET MVC 3 in the forums

ASP.NET MVC 3 Overview

ASP.NET MVC 3 builds on ASP.NET MVC 1 and 2, adding great features that both simplify your code and allow deeper extensibility. This topic provides an overview of many of the new features that are included in this release, organized into the following sections:

- Extensible Scaffolding with MvcScaffold integration
- HTML 5 enabled project templates
- The Razor View Engine
- Support for Multiple View Engines

- Controller Improvements
- JavaScript and Ajax
- Model Validation Improvements
- Dependency Injection Improvements
- Other New Features

Extensible Scaffolding with MvcScaffold integration

The new Scaffolding system makes it easier to pick up and start using productively if you're entirely new to the framework, and to automate common development tasks if you're experienced and already know what you're doing.

This is supported by new NuGet *scaffolding* package called **MvcScaffolding**. The term "Scaffolding" is used by many software technologies to mean "quickly generating a basic outline of your software that you can then edit and customize". The scaffolding package we're creating for ASP.NET MVC is greatly beneficial in several scenarios:

- **If you're learning ASP.NET MVC for the first time**, because it gives you a fast way to get some useful, working code, that you can then edit and adapt according to your needs. It saves you from the trauma of looking at a blank page and having no idea where to start!
- **If you know ASP.NET MVC well and are now exploring some new add-on technology** such as an object-relational mapper, a view engine, a testing library, etc., because the creator of that technology may have also created a scaffolding package for it.
- **If your work involves repeatedly creating similar classes or files of some sort**, because you can create custom scaffolders that output test fixtures, deployment scripts, or whatever else you need. Everyone on your team can use your custom scaffolders, too.

Other features in MvcScaffolding include:

- Support for C# and VB projects
- Support for the Razor and ASPX view engines
- Supports scaffolding into ASP.NET MVC areas and using custom view layouts/masters
- You can easily customize the output by editing T4 templates
- You can add entirely new scaffolders using custom PowerShell logic and custom T4 templates. These (and any custom parameters you've given them) automatically appear in the console tab-completion list.
- You can get NuGet packages containing additional scaffolders for different technologies (e.g., there's a proof-of-concept one for LINQ to SQL now) and mix and match them together

The ASP.NET MVC 3 Tools Update includes great Visual Studio support for this scaffolding system, such as:

- Add Controller Dialog now supports full automatic scaffolding of Create, Read, Update, and Delete controller actions and corresponding views. By default, this scaffolds data access code using EF Code First.
- Add Controller Dialog supports *extensible scaffolds* via NuGet packages such as *MvcScaffolding*. This allows plugging in custom scaffolds into the dialog which would allow you to create scaffolds for other data access technologies such as NHibernate or even JET with ODBCDirect if you're so inclined!

For more information about Scaffolding in ASP.NET MVC 3, see the following resources:

- Steve Sanderson's post series, including:
  1. Introduction: Scaffold your ASP.NET MVC 3 project with the MvcScaffolding package
  2. Standard usage: Typical use cases and options
  3. One-to-Many Relationships
  4. Scaffolding Actions and Unit Tests
  5. Overriding the T4 templates

6. This post: Creating custom scaffolders

- Scott Hanselman's post from his PDC 2010 session Building a Blog with Microsoft "Unnamed Package of Web Love"
- MVC 3 Release Notes

HTML 5 Project Templates

The New Project dialog includes a checkbox enable HTML 5 versions of project templates. These templates leverage Modernizr 1.7 to provide compatibility support for HTML 5 and CSS 3 in down-level browsers.

The Razor View Engine

ASP.NET MVC 3 comes with a new view engine named Razor that offers the following benefits:

- Razor syntax is clean and concise, requiring a minimum number of keystrokes.
- Razor is easy to learn, in part because it's based on existing languages like C# and Visual Basic.
- Visual Studio includes IntelliSense and code colorization for Razor syntax.
- Razor views can be unit tested without requiring that you run the application or launch a web server.

Some new Razor features include the following:

- `@model` syntax for specifying the type being passed to the view.
- `@* *@` comment syntax.
- The ability to specify defaults (such as `layoutpage`) once for an entire site.
- The `Html.Raw` method for displaying text without HTML-encoding it.
- Support for sharing code among multiple views (*_viewstart.cshtml* or *_viewstart.vbhtml* files).

Razor also includes new HTML helpers, such as the following:

- `Chart`. Renders a chart, offering the same features as the chart control in ASP.NET 4.
- `WebGrid`. Renders a data grid, complete with paging and sorting functionality.
- `Crypto`. Uses hashing algorithms to create properly salted and hashed passwords.
- `WebImage`. Renders an image.
- `WebMail`. Sends an email message.

For more information about Razor, see the following resources:

- Scott Guthrie's blog post introducing Razor
- Scott Guthrie's blog post introducing the @model keyword
- Scott Guthrie's blog post introducing Razor layouts
- Razor API Quick Reference
- MVC 3 Release Notes

Support for Multiple View Engines

The **Add View** dialog box in ASP.NET MVC 3 lets you choose the view engine you want to work with, and the **New Project** dialog box lets you specify the default view engine for a project. You can choose the Web Forms view engine (ASPX), Razor, or an open-source view engine such as Spark, NHaml, or NDjango.

Controller Improvements

Global Action Filters

Sometimes you want to perform logic either before an action method runs or after an action method runs. To support this, ASP.NET MVC 2 provided action filters. Action filters are custom attributes that provide a declarative means to add pre-action and post-action behavior to specific controller action methods. However, in some cases you might want to specify pre-action or post-action behavior that applies to all action methods. MVC 3 lets you specify global filters by adding them to the `GlobalFilters` collection. For more information about global action filters, see the following resources:

- Scott Guthrie's blog on the MVC 3 Preview
- Filtering in ASP.NET MVC

New "ViewBag" Property

MVC 2 controllers support a `ViewData` property that enables you to pass data to a view template using a late-bound dictionary API. In MVC 3, you can also use somewhat simpler syntax with the `ViewBag` property to accomplish the same purpose. For example, instead of writing `ViewData["Message"]="text"`, you can write `ViewBag.Message="text"`. You do not need to define any strongly-typed classes to use the `ViewBag` property. Because it is a dynamic property, you can instead just get or set properties and it will resolve them dynamically at run time. Internally, `ViewBag` properties are stored as name/value pairs in the `ViewData` dictionary. (Note: in most pre-release versions of MVC 3, the `ViewBag` property was named the `ViewModel` property.)

New "ActionResult" Types

The following `ActionResult` types and corresponding helper methods are new or enhanced in MVC 3:

- HttpNotFoundResult. Returns a 404 HTTP status code to the client.
- RedirectResult. Returns a temporary redirect (HTTP 302 status code) or a permanent redirect (HTTP 301 status code), depending on a Boolean parameter. In conjunction with this change, the Controller class now has three methods for performing permanent redirects: `RedirectPermanent`, `RedirectToRoutePermanent`, and `RedirectToActionPermanent`. These methods return an instance of `RedirectResult` with the `Permanent` property set to `true`.
- HttpStatusCodeResult. Returns a user-specified HTTP status code.

JavaScript and Ajax Improvements

By default, Ajax and validation helpers in MVC 3 use an unobtrusive JavaScript approach. Unobtrusive JavaScript avoids injecting inline JavaScript into HTML. This makes your HTML smaller and less cluttered, and makes it easier to swap out or customize JavaScript libraries. Validation helpers in MVC 3 also use the `jQueryValidate` plugin by default. If you want MVC 2 behavior, you can disable unobtrusive JavaScript using a *web.config* file setting. For more information about JavaScript and Ajax improvements, see the following resources:

- Basic introduction to unobtrusive JavaScript on the Wikipedia site
- Brad Wilson's Unobtrusive JavaScript Post
- Brad Wilson's Unobtrusive JavaScript Validation Post
- Creating a MVC 3 Application with Razor and Unobtrusive JavaScript (tutorial on the ASP.NET site)
- MVC 3 Release Notes

Client-Side Validation Enabled by Default

In earlier versions of MVC, you need to explicitly call the `Html.EnableClientValidation` method from a view in order to enable client-side validation. In MVC 3 this is no longer required because client-side validation is enabled by default. (You can disable this using a setting in the *web.config* file.)

In order for client-side validation to work, you still need to reference the appropriate jQuery and jQuery Validation libraries in your site. You can host those libraries on your own server or reference them from a content delivery network (CDN) like the CDNs from Microsoft or Google.

Remote Validator

ASP.NET MVC 3 supports the new RemoteAttribute class that enables you to take advantage of the jQuery Validation plug-in's remote validator support. This enables the client-side validation library to automatically call a custom method that you define on the server in order to perform validation logic that can only be done server-side.

In the following example, the `Remote` attribute specifies that client validation will call an action named `UserNameAvailable` on the `UsersController` class in order to validate the `UserName` field.

```
public class User
{
    [Remote("UserNameAvailable", "Users")]
    public string UserName { get; set; }
}
```

The following example shows the corresponding controller.

```
public class UsersController
{
    public bool UserNameAvailable(string username)
    {
        if(MyRepository.UserNameExists(username))
        {
            return "false";
        }
        return "true";
    }
}
```

For more information about how to use the `Remote` attribute, see How to: Implement Remote Validation in ASP.NET MVC in the MSDN library.

JSON Binding Support

ASP.NET MVC 3 includes built-in JSON binding support that enables action methods to receive JSON-encoded data and model-bind it to action-method parameters. This capability is useful in scenarios involving client templates and data binding. (Client templates enable you to format and display a single data item or set of data items by using templates that execute on the client.) MVC 3 enables you to easily connect client templates with action methods on the server that send and receive JSON data. For more information about JSON binding support, see the **JavaScript and AJAX Improvements** section of Scott Guthrie's MVC 3 Preview blog post.

Model Validation Improvements

"DataAnnotations" Metadata Attributes

ASP.NET MVC 3 supports `DataAnnotations` metadata attributes such as `DisplayAttribute`.

"ValidationAttribute" Class

The `ValidationAttribute` class was improved in the .NET Framework 4 to support a new `IsValid` overload that provides more information about the current validation context, such as what object is being validated. This enables richer scenarios where you can validate the current value based on another property of the model. For example, the new `CompareAttribute` attribute lets you compare the values of two properties of a model. In the following example, the `ComparePassword` property must match the `Password` field in order to be valid.

```
public class User
{
    [Required]
    public string Password { get; set; }
    [Required, Compare("Password")]
    public string ComparePassword { get; set; }
}
```

Validation Interfaces

The IValidatableObject interface enables you to perform model-level validation, and it enables you to provide validation error messages that are specific to the state of the overall model, or between two properties within the model. MVC 3 now retrieves errors from the `IValidatableObject` interface when model binding, and automatically flags or highlights affected fields within a view using the built-in HTML form helpers.

The IClientValidatable interface enables ASP.NET MVC to discover at run time whether a validator has support for client validation. This interface has been designed so that it can be integrated with a variety of validation frameworks.

For more information about validation interfaces, see the **Model Validation Improvements** section of Scott Guthrie's MVC 3 Preview blog post. (However, note that the reference to "IValidateObject" in the blog should be "IValidatableObject".)

Dependency Injection Improvements

ASP.NET MVC 3 provides better support for applying Dependency Injection (DI) and for integrating with Dependency Injection or Inversion of Control (IOC) containers. Support for DI has been added in the following areas:

- Controllers (registering and injecting controller factories, injecting controllers).
- Views (registering and injecting view engines, injecting dependencies into view pages).
- Action filters (locating and injecting filters).
- Model binders (registering and injecting).
- Model validation providers (registering and injecting).
- Model metadata providers (registering and injecting).
- Value providers (registering and injecting).

MVC 3 supports the Common Service Locator library and any DI container that supports that library's `IServiceLocator` interface. It also supports a new `IDependencyResolver` interface that makes it easier to integrate DI frameworks.

For more information about DI in MVC 3, see the following resources:

- Brad Wilson's series of blog posts on Service Location
- MVC 3 Release Notes

Other New Features

NuGet Integration

ASP.NET MVC 3 automatically installs and enables NuGet as part of its setup. NuGet is a free open-source package manager that makes it easy to find, install, and use .NET libraries and tools in your projects. It works with all Visual Studio project types (including ASP.NET Web Forms and ASP.NET MVC).

NuGet enables developers who maintain open source projects (for example, projects like Moq, NHibernate, Ninject, StructureMap, NUnit, Windsor, RhinoMocks, and Elmah) to package their libraries and register them in an online

gallery. It is then easy for .NET developers who want to use one of these libraries to find the package and install it in projects they are working on.

With the ASP.NET 3 Tools Update, project templates include JavaScript libraries pre-installed NuGet packages, so they are updatable via NuGet. Entity Framework Code First is also pre-installed as a NuGet package.

For more information about NuGet, see the NuGet documentation.

Partial-Page Output Caching

ASP.NET MVC has supported output caching of full page responses since version 1. MVC 3 also supports partial-page output caching, which allows you to easily cache regions or fragments of a response. For more information about caching, see the **Partial Page Output Caching** section of Scott Guthrie's blog post on the MVC 3 release candidate and the **Child Action Output Caching** section of the MVC 3 Release Notes.

Granular Control over Request Validation

ASP.NET MVC has built-in request validation that automatically helps protect against XSS and HTML injection attacks. However, sometimes you want to explicitly disable request validation, such as if you want to let users post HTML content (for example, in blog entries or CMS content). You can now add an AllowHtml attribute to models or view models to disable request validation on a per-property basis during model binding. For more information about request validation, see the following resources:

- The **Unobtrusive JavaScript and Validation** section in Scott Guthrie's blog post on the MVC 3 release candidate.
- MVC 3 Release Notes

Extensible "New Project" Dialog Box

In ASP.NET MVC 3 you can add project templates, view engines, and unit test project frameworks to the **New Project** dialog box.

Template Scaffolding Improvements

ASP.NET MVC 3 scaffolding templates do a better job of identifying primary-key properties on models and handling them appropriately than in earlier versions of MVC. (For example, the scaffolding templates now make sure that the primary key is not scaffolded as an editable form field.)

By default, the Create and Edit scaffolds now use the `Html.EditorFor` helper instead of the `Html.TextBoxFor` helper. This improves support for metadata on the model in the form of data annotation attributes when the **Add View** dialog box generates a view.

New Overloads for "Html.LabelFor" and "Html.LabelForModel"

New method overloads have been added for the `LabelFor` and `LabelForModel` helper methods. The new overloads enable you to specify or override the label text.

Sessionless Controller Support

In ASP.NET MVC 3 you can indicate whether you want a controller class to use session state, and if so, whether session state should be read/write or read-only. For more information about sessionless controller support, see MVC 3 Release Notes.

New "AdditionalMetadataAttribute" Class

You can use the AdditionalMetadata attribute to populate the `ModelMetadata.AdditionalValues` dictionary for a model property. For example, if a view model has a property that should be displayed only to an administrator, you can annotate that property as shown in the following example:

```
public class ProductViewModel
{
    [AdditionalMetadata("AdminOnly", true)]
    public string RefundCode {get; set;}
}
```

This metadata is made available to any display or editor template when a product view model is rendered. It is up to you to interpret the metadata information.

AccountController improvements

The AccountController in the Internet project template has been greatly improved.

New Intranet Project Template

A new Intranet Project Template is included which enables Windows Authentication and removes the AccountController.

MVC Video Training from Pluralsight

2/20/2020 • 2 minutes to read • Edit Online

Videos

- Introduction and Prerequisites
- OWIN and Katana
- Identity and Security
- Bootstrap
- Web API 2
- Entity Framework 6
- SignalR
- Web Developer Tools and Visual Studio 2013

> This MVC 5 course recorded by K. Scott Allen for Pluralsight provides a fantastic way to quickly get up to speed on the latest release of ASP.NET MVC. Scott's presentation style is easy to follow and technically compelling, and the course format, video player, and overall cadence are really great. Plus – it's free, so there's no excuse not to learn MVC 5 today!
>
> —Scott Guthrie, Executive Vice President, Microsoft Cloud and Enterprise group, Microsoft

Unlock access to 3,000+ dev, IT, and creative courses that you can watch anytime, anywhere.

- ASP.NET MVC 5 Fundamentals
- Pluralsight Pricing and plans

ASP.NET Web API

2/20/2020 • 3 minutes to read • Edit Online

- Guidance
  - Getting Started
    - Getting Started with Web API 2 (C#)
    - Action Results in Web API 2
    - Using Web API with ASP.NET Web Forms
    - Creating Web API Help Pages
    - Hands On Lab: Build a Single Page Application (SPA) with ASP.NET Web API and Angular.js
  - Routing
    - Routing in Web API
    - Routing and Action Selection in Web API
    - Attribute Routing in Web API 2
    - Create a REST API with Attribute Routing in Web API 2
  - Working with Data
    - Using Web API 2 with Entity Framework 6
      - Create the Project
      - Add Models and Controllers
      - Seed the Database
      - Handling Entity Relations
      - Create Data Transfer Objects (DTOs)
      - Create the JavaScript Client
      - Create the View (UI)
      - Display Item Details
      - Add a New Item to the Database
      - Publish the App to Azure Websites
  - Mobile Clients
    - Calling Web API from a Windows Phone 8 Application (C#)
  - OData
    - OData v4
      - Create an OData v4 Endpoint
      - Create an OData v4 Client App (C#)
      - Entity Relations in OData v4
      - Actions and Functions in OData v4
      - Containment in OData v4
      - Create a Singleton in OData v4
      - Open Types in OData v4
      - Complex Type Inheritance in OData v4
    - OData v3

    - Creating an OData v3 Endpoint
    - Supporting Entity Relations in OData v3
    - OData Actions in Web API 2
    - Calling an OData Service From a .NET Client (C#)
  - Supporting OData Query Options in Web API 2
  - Using $select, $expand, and $value in Web API 2 OData
  - Routing Conventions in Web API 2 OData
  - Security Guidance for Web API 2 OData
- Serialization and Model Binding
  - Media Formatters in Web API 2
  - JSON and XML Serialization in Web API
  - BSON Support in Web API 2.1
  - Content Negotiation in Web API
  - Model Validation in Web API
  - Parameter Binding in Web API
- Error Handling
  - Exception Handling in Web API
  - Global Error Handling in Web API 2
- Testing and Debugging
  - Tracing in Web API 2
  - Unit Testing Web API 2
  - Unit Testing Controllers in ASP.NET Web API 2
  - Mocking Entity Framework when Unit Testing Web API 2
  - Troubleshooting HTTP 405 Errors after Publishing Web API 2 Applications
- Security, Authentication, and Authorization
  - Authentication and Authorization in Web API
  - Secure a Web API with Individual Accounts in Web API 2.2
  - External Authentication Services with Web API (C#)
  - Preventing Cross-Site Request Forgery (CSRF) Attacks in Web API
  - Enabling Cross-Origin Requests in Web API 2
  - Authentication Filters in Web API 2
  - Basic Authentication in Web API
  - Forms Authentication in Web API
  - Integrated Windows Authentication
  - Working with SSL
- Deployment and Hosting
  - Use OWIN to Self-Host Web API 2
  - Host Web API 2 in an Azure Worker Role
- Advanced Topics
  - Configuring Web API 2
  - Dependency Injection in Web API 2
  - Calling a Web API From a .NET Client (C#)

    - HTTP Message Handlers in Web API
    - HttpClient Message Handlers in Web API
    - Sending HTML Form Data in Web API: Form-urlencoded Data
    - Sending HTML Form Data in Web API: File Upload and Multipart MIME
    - HTTP Cookies in ASP.NET Web API
  - Releases
    - What's New in Web API OData 5.3
    - What's New in ASP.NET Web API 2.2
    - What's New in ASP.NET Web API 2.1
  - Older Versions
    - Enabling CRUD Operations in Web API 1
    - Using ASP.NET Web API 1 with Entity Framework 5
      - Overview and Creating the Project
      - Creating the Domain Models
      - Creating an Admin Controller
      - Adding an Admin View
      - Creating a Dynamic UI with Knockout.js
      - Creating Product and Order Controllers
      - Creating the Main Page
    - Self-Host Web API 1 (C#)
    - Build RESTful API's with ASP.NET Web API
- Videos
  - Overview and Getting Started
    - ASP.NET Web API
    - ASP.NET Web API, Part 1: Your First Web API
    - ASP.NET Web API, Part 2: Getting Data
    - ASP.NET Web API, Part 3: Delete and Update
    - ASP.NET Web API, Part 4: Paging and Querying
    - ASP.NET Web API, Part 5: Custom Validation
    - ASP.NET Web API, Part 6: Authorization
- Web API Samples List

Web API Guidance

2/20/2020 • 2 minutes to read • Edit Online

- Getting Started
  - Getting Started with Web API 2 (C#)
  - Action Results in Web API 2
  - Using Web API with ASP.NET Web Forms
  - Creating Web API Help Pages
  - Hands On Lab: Build a Single Page Application (SPA) with ASP.NET Web API and Angular.js
- Routing
  - Routing in Web API
  - Routing and Action Selection in Web API
  - Attribute Routing in Web API 2
  - Create a REST API with Attribute Routing in Web API 2
- Working with Data
  - Using Web API 2 with Entity Framework 6
    - Create the Project
    - Add Models and Controllers
    - Seed the Database
    - Handling Entity Relations
    - Create Data Transfer Objects (DTOs)
    - Create the JavaScript Client
    - Create the View (UI)
    - Display Item Details
    - Add a New Item to the Database
    - Publish the App to Azure Websites
- Mobile Clients
  - Calling Web API from a Windows Phone 8 Application (C#)
- OData
  - OData v4
    - Create an OData v4 Endpoint
    - Create an OData v4 Client App (C#)
    - Entity Relations in OData v4
    - Actions and Functions in OData v4
    - Containment in OData v4
    - Create a Singleton in OData v4
    - Open Types in OData v4
    - Complex Type Inheritance in OData v4
  - OData v3
    - Creating an OData v3 Endpoint

    - Supporting Entity Relations in OData v3
    - OData Actions in Web API 2
    - Calling an OData Service From a .NET Client (C#)
  - Supporting OData Query Options in Web API 2
  - Using $select, $expand, and $value in Web API 2 OData
  - Routing Conventions in Web API 2 OData
  - Security Guidance for Web API 2 OData
- Serialization and Model Binding
  - Media Formatters in Web API 2
  - JSON and XML Serialization in Web API
  - BSON Support in Web API 2.1
  - Content Negotiation in Web API
  - Model Validation in Web API
  - Parameter Binding in Web API
- Error Handling
  - Exception Handling in Web API
  - Global Error Handling in Web API 2
- Testing and Debugging
  - Tracing in Web API 2
  - Unit Testing Web API 2
  - Unit Testing Controllers in ASP.NET Web API 2
  - Mocking Entity Framework when Unit Testing Web API 2
  - Troubleshooting HTTP 405 Errors after Publishing Web API 2 Applications
- Security, Authentication, and Authorization
  - Authentication and Authorization in Web API
  - Secure a Web API with Individual Accounts in Web API 2.2
  - External Authentication Services with Web API (C#)
  - Preventing Cross-Site Request Forgery (CSRF) Attacks in Web API
  - Enabling Cross-Origin Requests in Web API 2
  - Authentication Filters in Web API 2
  - Basic Authentication in Web API
  - Forms Authentication in Web API
  - Integrated Windows Authentication
  - Working with SSL
- Deployment and Hosting
  - Use OWIN to Self-Host Web API 2
  - Host Web API 2 in an Azure Worker Role
- Advanced Topics
  - Configuring Web API 2
  - Dependency Injection in Web API 2
  - Calling a Web API From a .NET Client (C#)
  - HTTP Message Handlers in Web API

- HttpClient Message Handlers in Web API
- Sending HTML Form Data in Web API: Form-urlencoded Data
- Sending HTML Form Data in Web API: File Upload and Multipart MIME
- HTTP Cookies in ASP.NET Web API

- Releases
  - What's New in Web API OData 5.3
  - What's New in ASP.NET Web API 2.2
  - What's New in ASP.NET Web API 2.1
- Older Versions
  - Enabling CRUD Operations in Web API 1
  - Using ASP.NET Web API 1 with Entity Framework 5
    - Overview and Creating the Project
    - Creating the Domain Models
    - Creating an Admin Controller
    - Adding an Admin View
    - Creating a Dynamic UI with Knockout.js
    - Creating Product and Order Controllers
    - Creating the Main Page
  - Self-Host Web API 1 (C#)
  - Build RESTful API's with ASP.NET Web API

Getting Started with ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

The basics of building an HTTP service using ASP.NET Web API

- Getting Started with Web API 2 (C#)
- Action Results in Web API 2
- Using Web API with ASP.NET Web Forms
- Creating Web API Help Pages
- Hands On Lab: Build a Single Page Application (SPA) with ASP.NET Web API and Angular.js

Get Started with ASP.NET Web API 2 (C#)

3/11/2020 • 8 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

In this tutorial, you will use ASP.NET Web API to create a web API that returns a list of products.

HTTP is not just for serving up web pages. HTTP is also a powerful platform for building APIs that expose services and data. HTTP is simple, flexible, and ubiquitous. Almost any platform that you can think of has an HTTP library, so HTTP services can reach a broad range of clients, including browsers, mobile devices, and traditional desktop applications.

ASP.NET Web API is a framework for building web APIs on top of the .NET Framework.

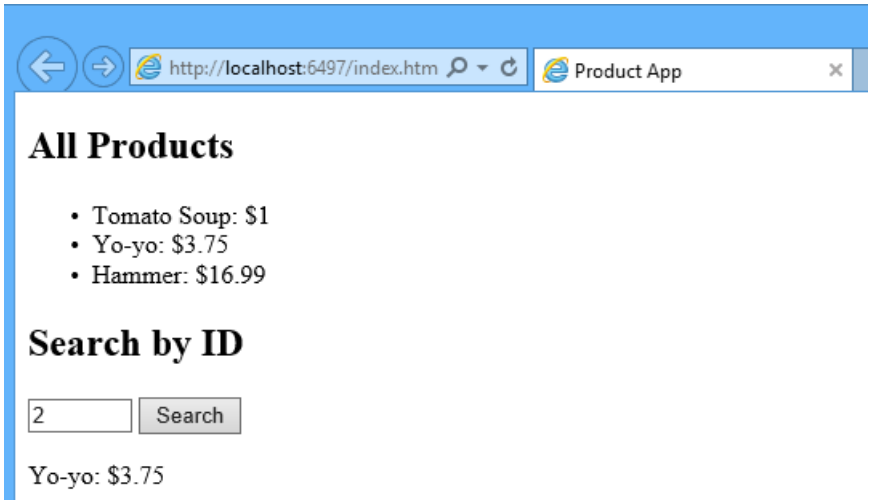
Software versions used in the tutorial

- Visual Studio 2017
- Web API 2

See Create a web API with ASP.NET Core and Visual Studio for Windows for a newer version of this tutorial.

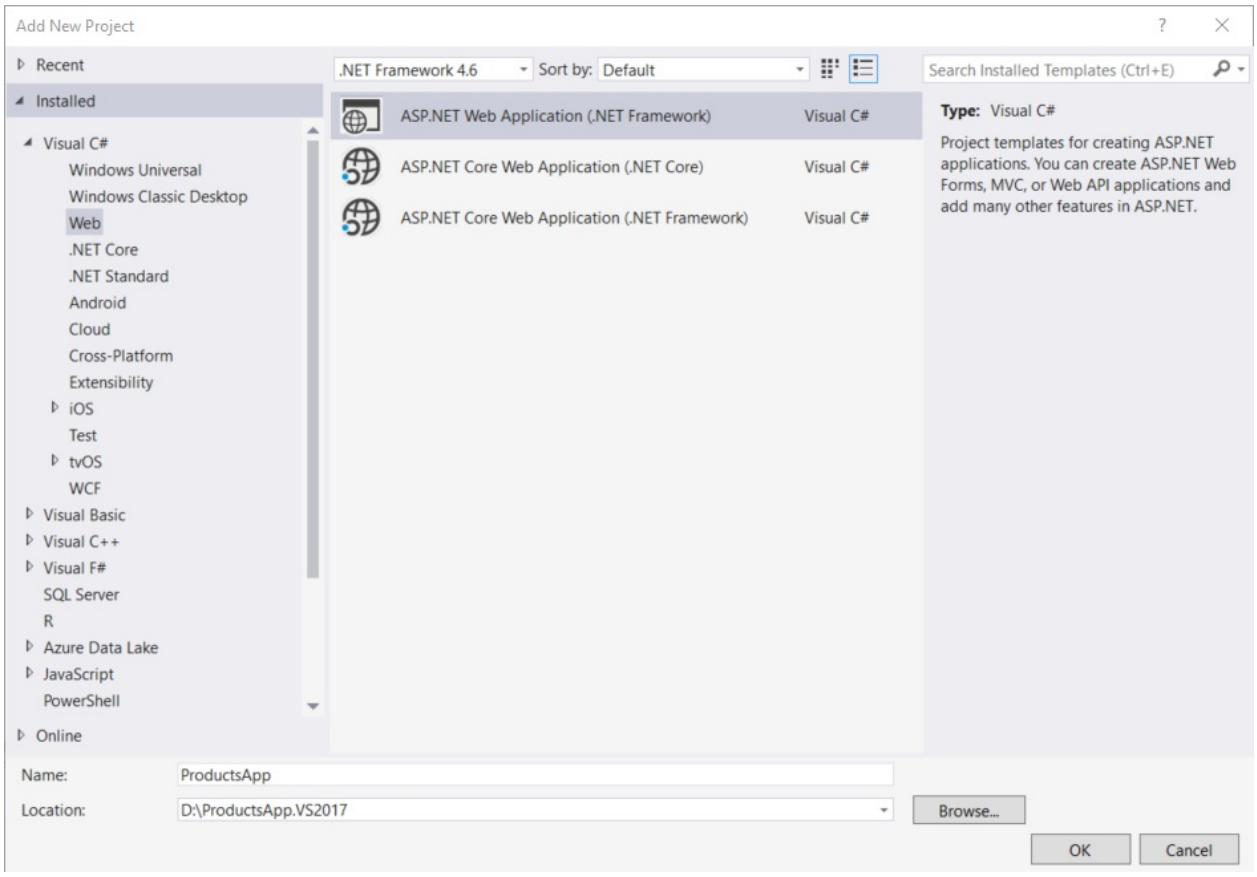
Create a Web API Project

In this tutorial, you will use ASP.NET Web API to create a web API that returns a list of products. The front-end web page uses jQuery to display the results.

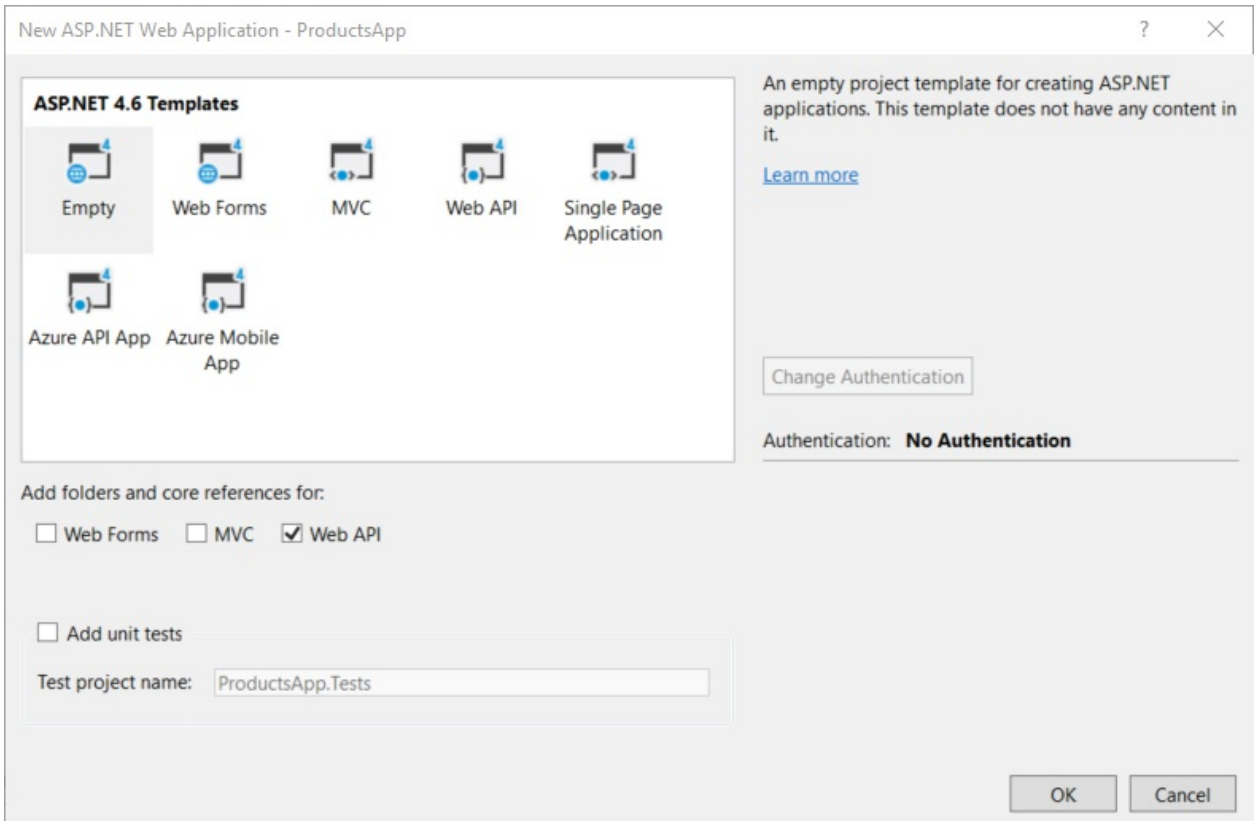


Start Visual Studio and select **New Project** from the **Start** page. Or, from the **File** menu, select **New** and then **Project**.

In the **Templates** pane, select **Installed Templates** and expand the **Visual C#** node. Under **Visual C#**, select **Web**. In the list of project templates, select **ASP.NET Web Application**. Name the project "ProductsApp" and click **OK**.



In the **New ASP.NET Project** dialog, select the **Empty** template. Under "Add folders and core references for", check **Web API**. Click **OK**.



NOTE

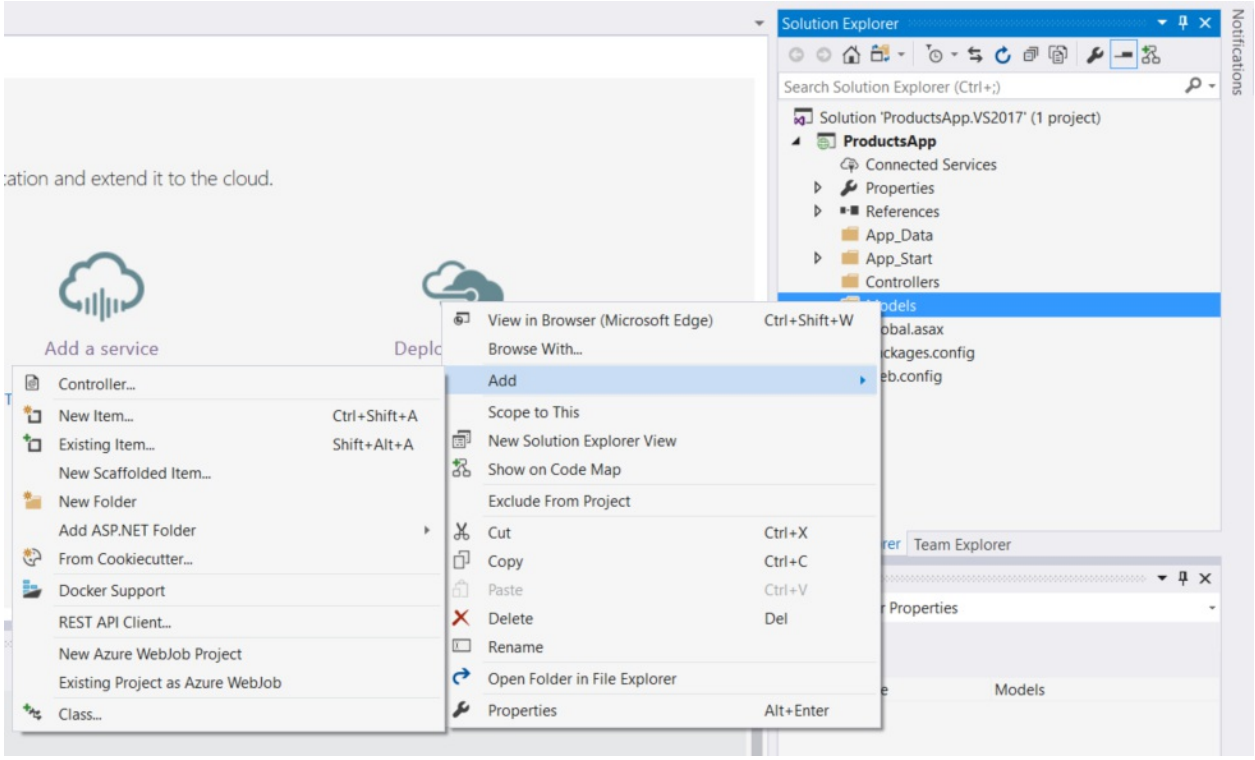
You can also create a Web API project using the "Web API" template. The Web API template uses ASP.NET MVC to provide API help pages. I'm using the Empty template for this tutorial because I want to show Web API without MVC. In general, you don't need to know ASP.NET MVC to use Web API.

Adding a Model

A *model* is an object that represents the data in your application. ASP.NET Web API can automatically serialize your model to JSON, XML, or some other format, and then write the serialized data into the body of the HTTP response message. As long as a client can read the serialization format, it can deserialize the object. Most clients can parse either XML or JSON. Moreover, the client can indicate which format it wants by setting the Accept header in the HTTP request message.

Let's start by creating a simple model that represents a product.

If Solution Explorer is not already visible, click the **View** menu and select **Solution Explorer**. In Solution Explorer, right-click the Models folder. From the context menu, select **Add** then select **Class**.



Name the class "Product". Add the following properties to the `Product` class.

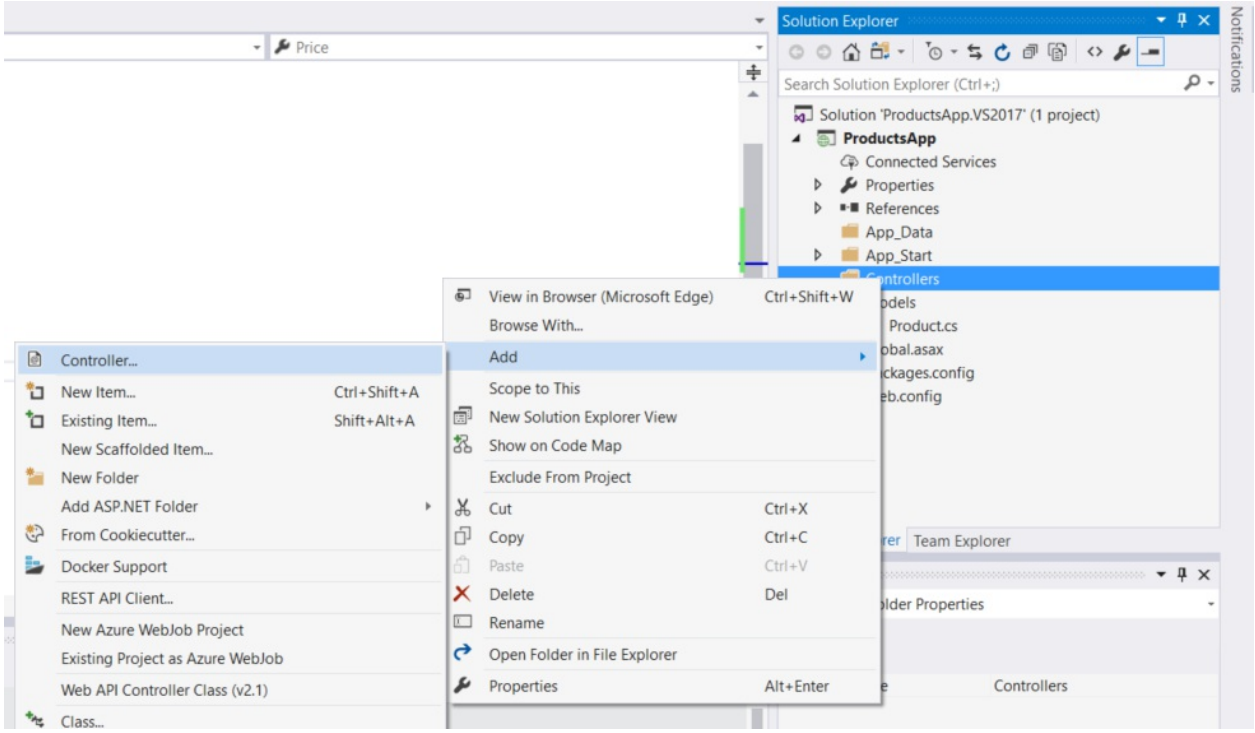
```
namespace ProductsApp.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public decimal Price { get; set; }
    }
}
```

Adding a Controller

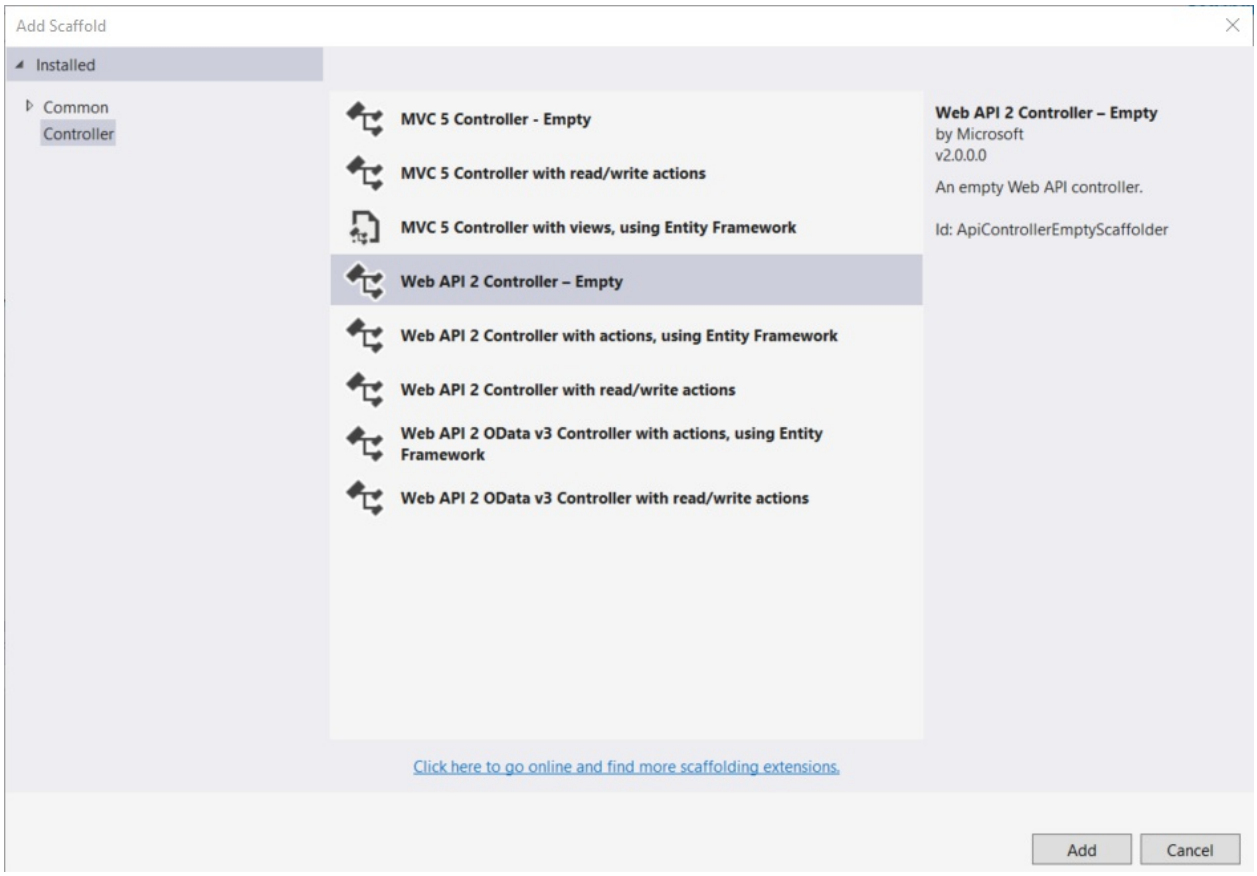
In Web API, a *controller* is an object that handles HTTP requests. We'll add a controller that can return either a list of products or a single product specified by ID.

> **NOTE**
>
> If you have used ASP.NET MVC, you are already familiar with controllers. Web API controllers are similar to MVC controllers, but inherit the **ApiController** class instead of the **Controller** class.

In **Solution Explorer**, right-click the Controllers folder. Select **Add** and then select **Controller**.

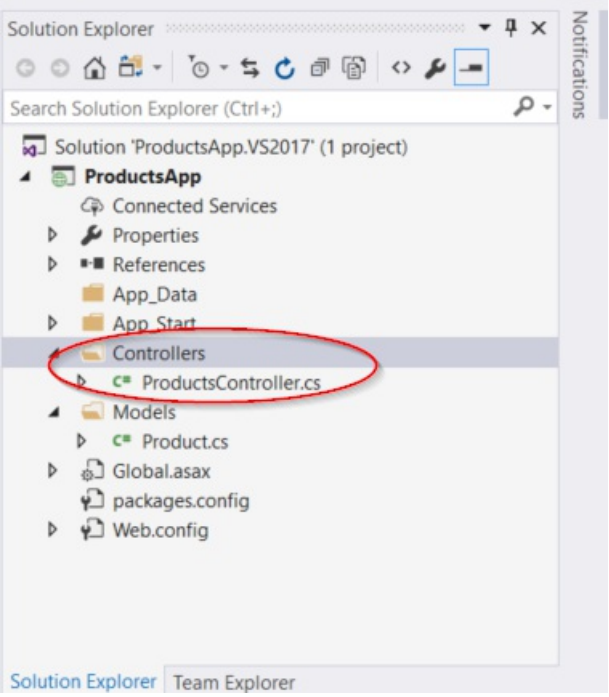


In the **Add Scaffold** dialog, select **Web API Controller - Empty**. Click **Add**.



In the **Add Controller** dialog, name the controller "ProductsController". Click **Add**.

The scaffolding creates a file named ProductsController.cs in the Controllers folder.



> **NOTE**
>
> You don't need to put your controllers into a folder named Controllers. The folder name is just a convenient way to organize your source files.

If this file is not open already, double-click the file to open it. Replace the code in this file with the following:

```
using ProductsApp.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Web.Http;

namespace ProductsApp.Controllers
{
    public class ProductsController : ApiController
    {
        Product[] products = new Product[]
        {
            new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
            new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
            new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
        };

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public IHttpActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product);
        }
    }
}
```

To keep the example simple, products are stored in a fixed array inside the controller class. Of course, in a real application, you would query a database or use some other external data source.

The controller defines two methods that return products:

- The `GetAllProducts` method returns the entire list of products as an **IEnumerable<Product>** type.
- The `GetProduct` method looks up a single product by its ID.

That's it! You have a working web API. Each method on the controller corresponds to one or more URIs:

CONTROLLER METHOD	URI
GetAllProducts	/api/products
GetProduct	/api/products/*id*

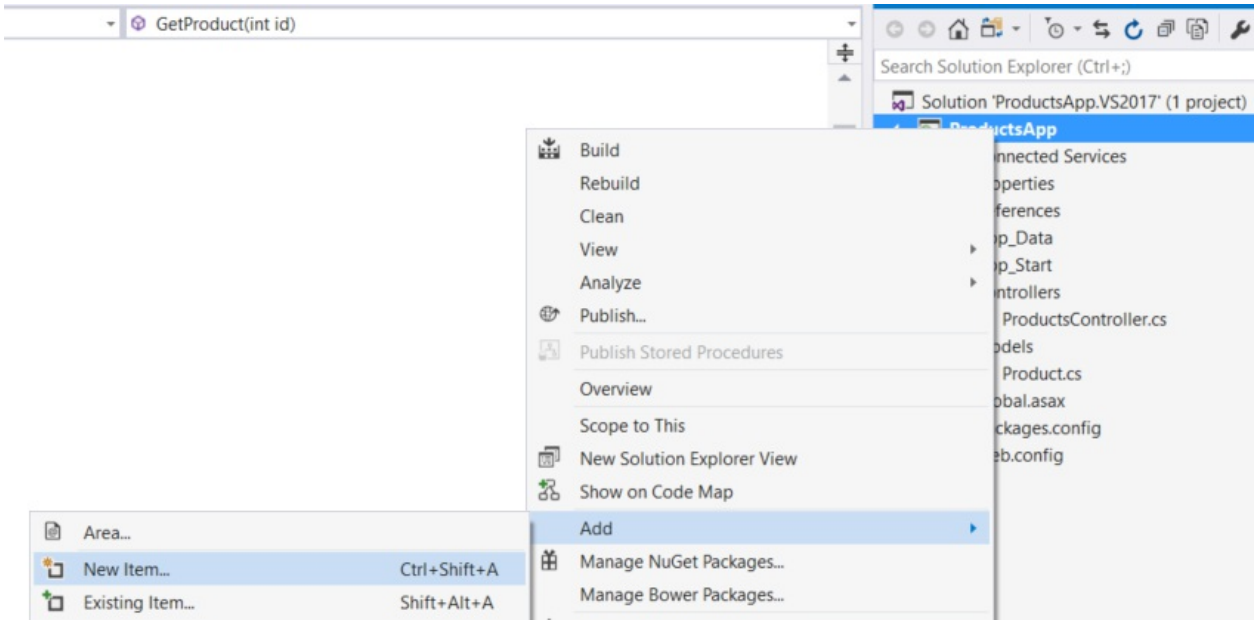
For the `GetProduct` method, the *id* in the URI is a placeholder. For example, to get the product with ID of 5, the URI is `api/products/5`.

For more information about how Web API routes HTTP requests to controller methods, see Routing in ASP.NET Web API.

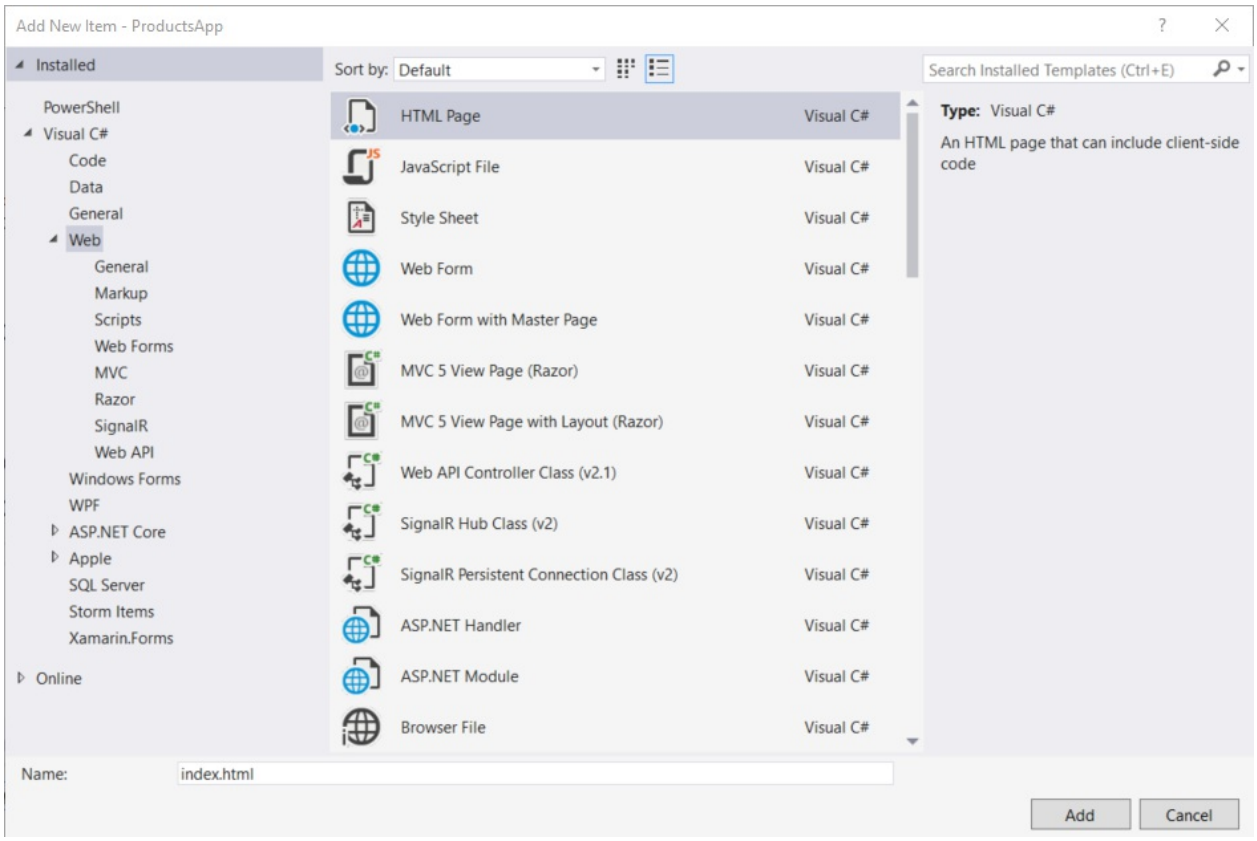
Calling the Web API with Javascript and jQuery

In this section, we'll add an HTML page that uses AJAX to call the web API. We'll use jQuery to make the AJAX calls and also to update the page with the results.

In Solution Explorer, right-click the project and select **Add**, then select **New Item**.



In the **Add New Item** dialog, select the **Web** node under **Visual C#**, and then select the **HTML Page** item. Name the page "index.html".



Replace everything in this file with the following:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Product App</title>
</head>
<body>

  <div>
    <h2>All Products</h2>
    <ul id="products" />
  </div>
  <div>
    <h2>Search by ID</h2>
    <input type="text" id="prodId" size="5" />
    <input type="button" value="Search" onclick="find();" />
    <p id="product" />
  </div>

  <script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.3.min.js"></script>
  <script>
    var uri = 'api/products';

    $(document).ready(function () {
      // Send an AJAX request
      $.getJSON(uri)
          .done(function (data) {
            // On success, 'data' contains a list of products.
            $.each(data, function (key, item) {
              // Add a list item for the product.
              $('<li>', { text: formatItem(item) }).appendTo($('#products'));
            });
          });
    });

    function formatItem(item) {
      return item.Name + ': $' + item.Price;
    }

    function find() {
      var id = $('#prodId').val();
      $.getJSON(uri + '/' + id)
          .done(function (data) {
            $('#product').text(formatItem(data));
          })
          .fail(function (jqXHR, textStatus, err) {
            $('#product').text('Error: ' + err);
          });
    }
  </script>
</body>
</html>
```

There are several ways to get jQuery. In this example, I used the Microsoft Ajax CDN. You can also download it from http://jquery.com/, and the ASP.NET "Web API" project template includes jQuery as well.

Getting a List of Products

To get a list of products, send an HTTP GET request to "/api/products".

The jQuery getJSON function sends an AJAX request. The response contains array of JSON objects. The `done` function specifies a callback that is called if the request succeeds. In the callback, we update the DOM with the product information.

```
$(document).ready(function () {
    // Send an AJAX request
    $.getJSON(apiUrl)
        .done(function (data) {
            // On success, 'data' contains a list of products.
            $.each(data, function (key, item) {
                // Add a list item for the product.
                $('<li>', { text: formatItem(item) }).appendTo($('#products'));
            });
        });
});
```

Getting a Product By ID

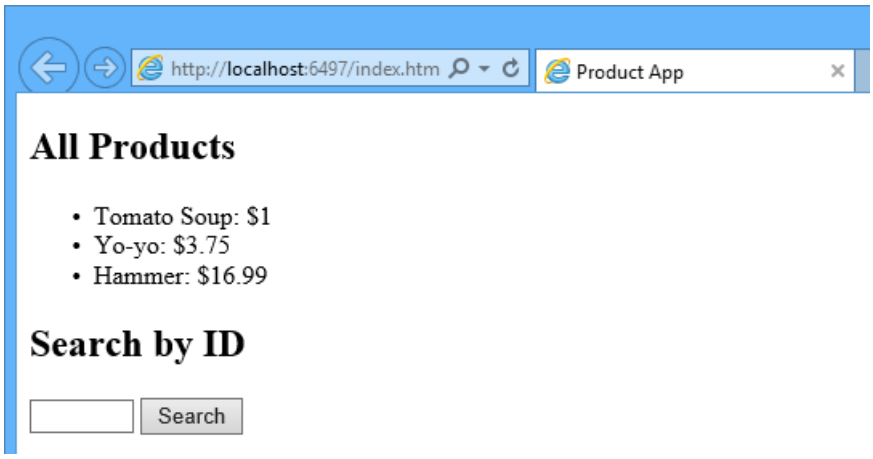
To get a product by ID, send an HTTP GET request to "/api/products/*id*", where *id* is the product ID.

```
function find() {
    var id = $('#prodId').val();
    $.getJSON(apiUrl + '/' + id)
        .done(function (data) {
            $('#product').text(formatItem(data));
        })
        .fail(function (jqXHR, textStatus, err) {
            $('#product').text('Error: ' + err);
        });
}
```

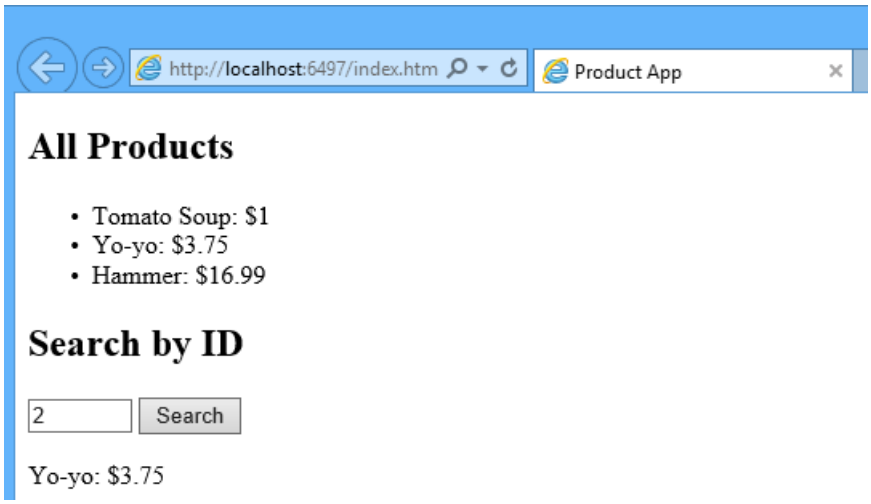
We still call `getJSON` to send the AJAX request, but this time we put the ID in the request URI. The response from this request is a JSON representation of a single product.

Running the Application

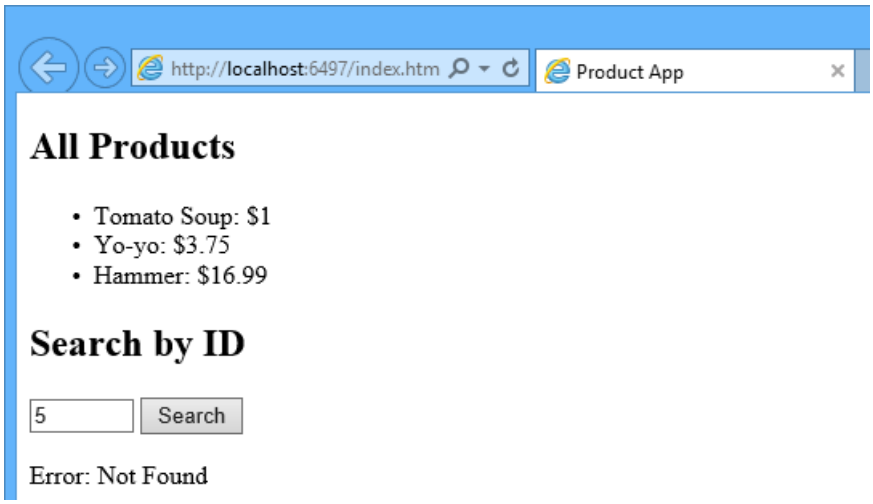
Press F5 to start debugging the application. The web page should look like the following:



To get a product by ID, enter the ID and click Search:

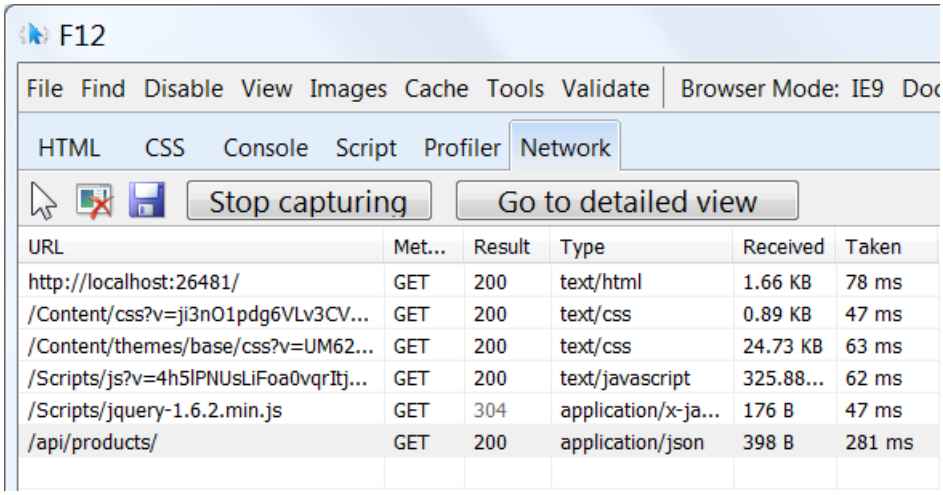


If you enter an invalid ID, the server returns an HTTP error:

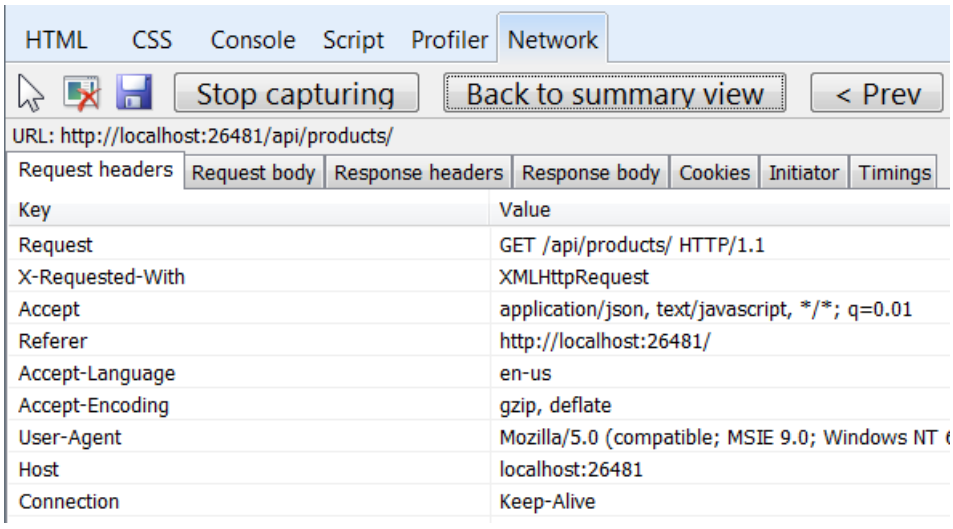


Using F12 to View the HTTP Request and Response

When you are working with an HTTP service, it can be very useful to see the HTTP request and response messages. You can do this by using the F12 developer tools in Internet Explorer 9. From Internet Explorer 9, press **F12** to open the tools. Click the **Network** tab and press **Start Capturing**. Now go back to the web page and press **F5** to reload the web page. Internet Explorer will capture the HTTP traffic between the browser and the web server. The summary view shows all the network traffic for a page:



Locate the entry for the relative URI "api/products/". Select this entry and click **Go to detailed view**. In the detail view, there are tabs to view the request and response headers and bodies. For example, if you click the **Request headers** tab, you can see that the client requested "application/json" in the Accept header.



If you click the Response body tab, you can see how the product list was serialized to JSON. Other browsers have similar functionality. Another useful tool is Fiddler, a web debugging proxy. You can use Fiddler to view your HTTP traffic, and also to compose HTTP requests, which gives you full control over the HTTP headers in the request.

See this App Running on Azure

Would you like to see the finished site running as a live web app? You can deploy a complete version of the app to your Azure account by simply clicking the following button.

Deploy to Azure

You need an Azure account to deploy this solution to Azure. If you do not already have an account, you have the following options:

- Open an Azure account for free - You get credits you can use to try out paid Azure services, and even after they're used up you can keep the account and use free Azure services.
- Activate MSDN subscriber benefits - Your MSDN subscription gives you credits every month that you can use for paid Azure services.

Next Steps

- For a more complete example of an HTTP service that supports POST, PUT, and DELETE actions and writes to a database, see Using Web API 2 with Entity Framework 6.
- For more about creating fluid and responsive web applications on top of an HTTP service, see ASP.NET Single Page Application.
- For information about how to deploy a Visual Studio web project to Azure App Service, see Create an ASP.NET web app in Azure App Service.

Action Results in Web API 2

2/20/2020 • 4 minutes to read • Edit Online

Consider using ASP.NET Core web API. It has the following advantages over ASP.NET 4.x Web API:

- ASP.NET Core is an open-source, cross-platform framework for building modern, cloud-based web apps on Windows, macOS, and Linux.
- The ASP.NET Core MVC controllers and web API controllers are unified.
- Architected for testability.
- Ability to develop and run on Windows, macOS, and Linux.
- Open-source and community-focused.
- Integration of modern, client-side frameworks and development workflows.
- A cloud-ready, environment-based configuration system.
- Built-in dependency injection.
- A lightweight, high-performance, and modular HTTP request pipeline.
- Ability to host on Kestrel, IIS, HTTP.sys, Nginx, Apache, and Docker.
- Side-by-side versioning.
- Tooling that simplifies modern web development.

This topic describes how ASP.NET Web API converts the return value from a controller action into an HTTP response message.

A Web API controller action can return any of the following:

1. void
2. **HttpResponseMessage**
3. **IHttpActionResult**
4. Some other type

Depending on which of these is returned, Web API uses a different mechanism to create the HTTP response.

RETURN TYPE	HOW WEB API CREATES THE RESPONSE
void	Return empty 204 (No Content)
HttpResponseMessage	Convert directly to an HTTP response message.
IHttpActionResult	Call **ExecuteAsync** to create an **HttpResponseMessage**, then convert to an HTTP response message.
Other type	Write the serialized return value into the response body; return 200 (OK).

The rest of this topic describes each option in more detail.

void

If the return type is `void`, Web API simply returns an empty HTTP response with status code 204 (No Content).

Example controller:

```
public class ValuesController : ApiController
{
    public void Post()
    {
    }
}
```

HTTP response:

```
HTTP/1.1 204 No Content
Server: Microsoft-IIS/8.0
Date: Mon, 27 Jan 2014 02:13:26 GMT
```

HttpResponseMessage

If the action returns an HttpResponseMessage, Web API converts the return value directly into an HTTP response message, using the properties of the **HttpResponseMessage** object to populate the response.

This option gives you a lot of control over the response message. For example, the following controller action sets the Cache-Control header.

```
public class ValuesController : ApiController
{
    public HttpResponseMessage Get()
    {
        HttpResponseMessage response = Request.CreateResponse(HttpStatusCode.OK, "value");
        response.Content = new StringContent("hello", Encoding.Unicode);
        response.Headers.CacheControl = new CacheControlHeaderValue()
        {
            MaxAge = TimeSpan.FromMinutes(20)
        };
        return response;
    }
}
```

Response:

```
HTTP/1.1 200 OK
Cache-Control: max-age=1200
Content-Length: 10
Content-Type: text/plain; charset=utf-16
Server: Microsoft-IIS/8.0
Date: Mon, 27 Jan 2014 08:53:35 GMT

hello
```

If you pass a domain model to the **CreateResponse** method, Web API uses a media formatter to write the serialized model into the response body.

```
public HttpResponseMessage Get()
{
    // Get a list of products from a database.
    IEnumerable<Product> products = GetProductsFromDB();

    // Write the list to the response body.
    HttpResponseMessage response = Request.CreateResponse(HttpStatusCode.OK, products);
    return response;
}
```

Web API uses the Accept header in the request to choose the formatter. For more information, see Content Negotiation.

IHttpActionResult

The **IHttpActionResult** interface was introduced in Web API 2. Essentially, it defines an **HttpResponseMessage** factory. Here are some advantages of using the **IHttpActionResult** interface:

- Simplifies unit testing your controllers.
- Moves common logic for creating HTTP responses into separate classes.
- Makes the intent of the controller action clearer, by hiding the low-level details of constructing the response.

IHttpActionResult contains a single method, **ExecuteAsync**, which asynchronously creates an **HttpResponseMessage** instance.

```
public interface IHttpActionResult
{
    Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken);
}
```

If a controller action returns an **IHttpActionResult**, Web API calls the **ExecuteAsync** method to create an **HttpResponseMessage**. Then it converts the **HttpResponseMessage** into an HTTP response message.

Here is a simple implementation of **IHttpActionResult** that creates a plain text response:

```
public class TextResult : IHttpActionResult
{
    string _value;
    HttpRequestMessage _request;

    public TextResult(string value, HttpRequestMessage request)
    {
        _value = value;
        _request = request;
    }
    public Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
    {
        var response = new HttpResponseMessage()
        {
            Content = new StringContent(_value),
            RequestMessage = _request
        };
        return Task.FromResult(response);
    }
}
```

Example controller action:

```
public class ValuesController : ApiController
{
    public IHttpActionResult Get()
    {
        return new TextResult("hello", Request);
    }
}
```

Response:

```
HTTP/1.1 200 OK
Content-Length: 5
Content-Type: text/plain; charset=utf-8
Server: Microsoft-IIS/8.0
Date: Mon, 27 Jan 2014 08:53:35 GMT

hello
```

More often, you use the **IHttpActionResult** implementations defined in the **System.Web.Http.Results** namespace. The **ApiController** class defines helper methods that return these built-in action results.

In the following example, if the request does not match an existing product ID, the controller calls ApiController.NotFound to create a 404 (Not Found) response. Otherwise, the controller calls ApiController.OK, which creates a 200 (OK) response that contains the product.

```
public IHttpActionResult Get (int id)
{
    Product product = _repository.Get (id);
    if (product == null)
    {
        return NotFound(); // Returns a NotFoundResult
    }
    return Ok(product);  // Returns an OkNegotiatedContentResult
}
```

Other Return Types

For all other return types, Web API uses a media formatter to serialize the return value. Web API writes the serialized value into the response body. The response status code is 200 (OK).

```
public class ProductsController : ApiController
{
    public IEnumerable<Product> Get()
    {
        return GetAllProductsFromDB();
    }
}
```

A disadvantage of this approach is that you cannot directly return an error code, such as 404. However, you can throw an **HttpResponseException** for error codes. For more information, see Exception Handling in ASP.NET Web API.

Web API uses the Accept header in the request to choose the formatter. For more information, see Content Negotiation.

Example request

```
GET http://localhost/api/products HTTP/1.1
User-Agent: Fiddler
Host: localhost:24127
Accept: application/json
```

Example response

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Server: Microsoft-IIS/8.0
Date: Mon, 27 Jan 2014 08:53:35 GMT
Content-Length: 56

[{"Id":1,"Name":"Yo-yo","Category":"Toys","Price":6.95}]
```

Using Web API with ASP.NET Web Forms

2/20/2020 • 3 minutes to read • Edit Online

by Mike Wasson

This tutorial walks you through the steps to add Web API to a traditional ASP.NET Web Forms application in ASP.NET 4.x.

Overview

Although ASP.NET Web API is packaged with ASP.NET MVC, it is easy to add Web API to a traditional ASP.NET Web Forms application.

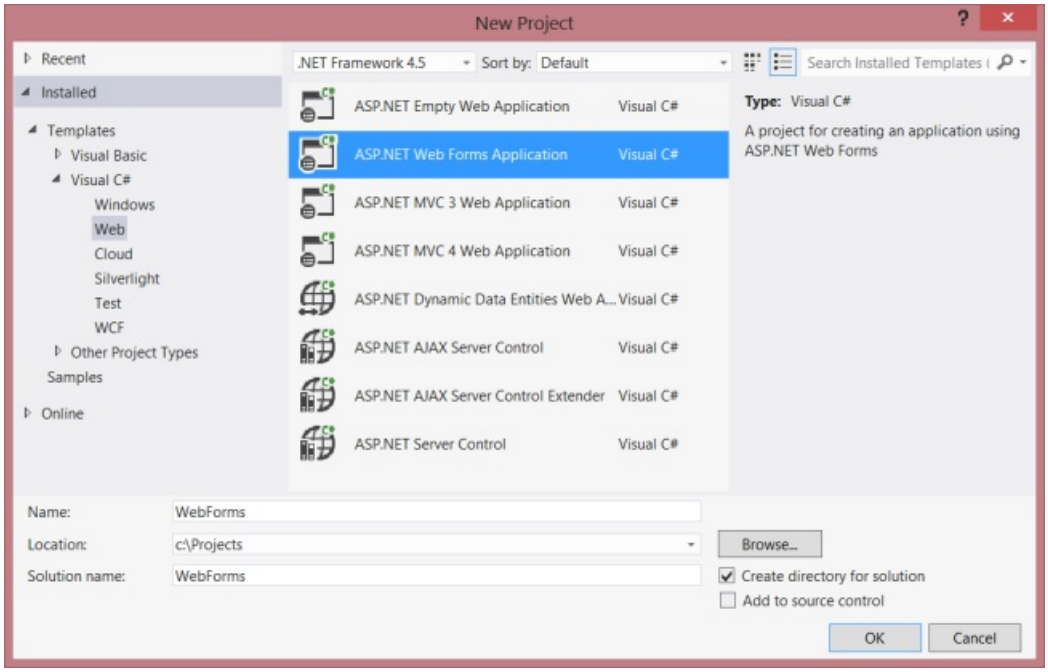
To use Web API in a Web Forms application, there are two main steps:

- Add a Web API controller that derives from the **ApiController** class.
- Add a route table to the **Application_Start** method.

Create a Web Forms Project

Start Visual Studio and select **New Project** from the **Start** page. Or, from the **File** menu, select **New** and then **Project**.

In the **Templates** pane, select **Installed Templates** and expand the **Visual C#** node. Under **Visual C#**, select **Web**. In the list of project templates, select **ASP.NET Web Forms Application**. Enter a name for the project and click **OK**.



Create the Model and Controller

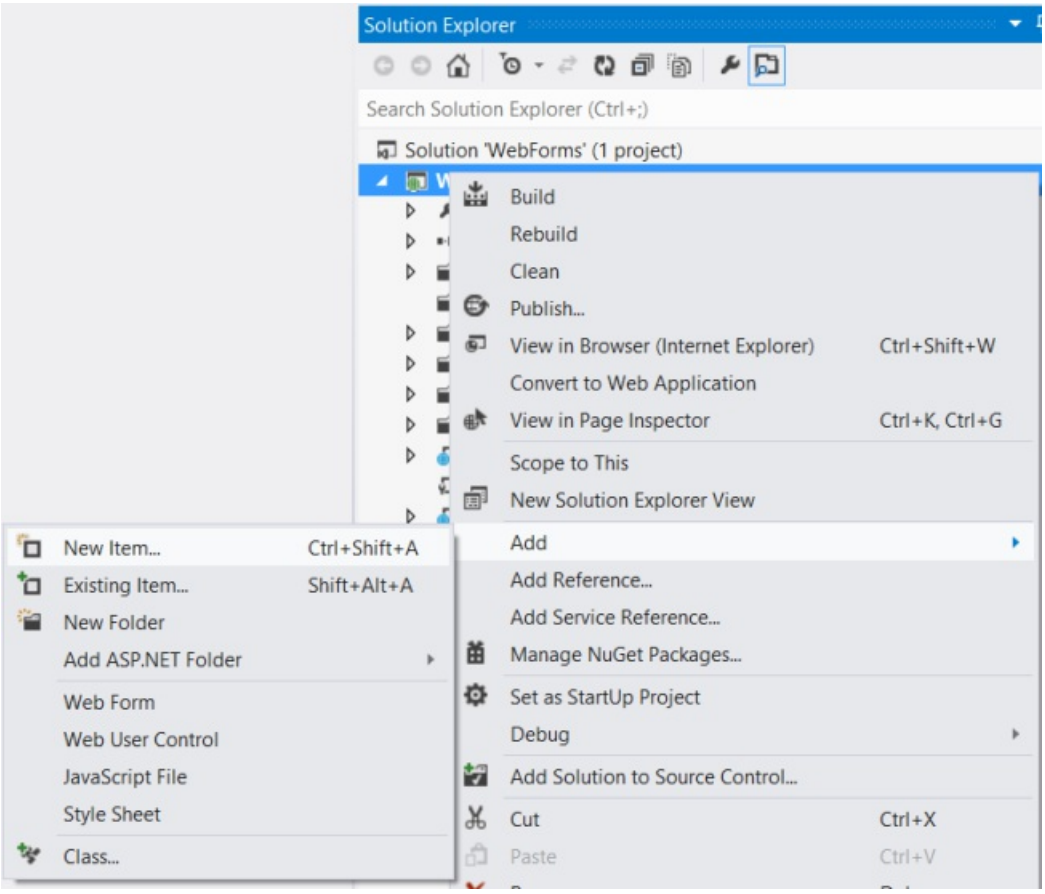
This tutorial uses the same model and controller classes as the Getting Started tutorial.

First, add a model class. In **Solution Explorer**, right-click the project and select **Add Class**. Name the class Product, and add the following implementation:

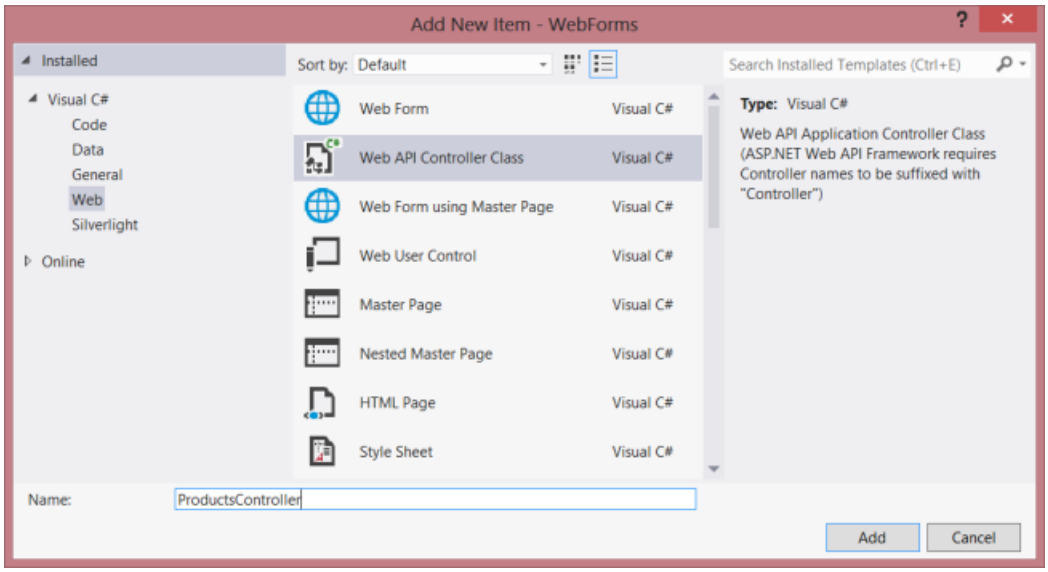
```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Category { get; set; }
}
```

Next, add a Web API controller to the project., A *controller* is the object that handles HTTP requests for Web API.

In **Solution Explorer**, right-click the project. Select **Add New Item**.



Under **Installed Templates**, expand **Visual C#** and select **Web**. Then, from the list of templates, select **Web API Controller Class**. Name the controller "ProductsController" and click **Add**.



The **Add New Item** wizard will create a file named ProductsController.cs. Delete the methods that the wizard

included and add the following methods:

```
namespace WebForms
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Net;
    using System.Net.Http;
    using System.Web.Http;

    public class ProductsController : ApiController
    {

        Product[] products = new Product[]
        {
            new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
            new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
            new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
        };

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public Product GetProductById(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                throw new HttpResponseException(HttpStatusCode.NotFound);
            }
            return product;
        }

        public IEnumerable<Product> GetProductsByCategory(string category)
        {
            return products.Where(
                (p) => string.Equals(p.Category, category,
                    StringComparison.OrdinalIgnoreCase));
        }
    }
}
```

For more information about the code in this controller, see the Getting Started tutorial.

Add Routing Information

Next, we'll add a URI route so that URIs of the form "/api/products/" are routed to the controller.

In **Solution Explorer**, double-click Global.asax to open the code-behind file Global.asax.cs. Add the following **using** statement.

```
using System.Web.Http;
```

Then add the following code to the **Application_Start** method:

```
RouteTable.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = System.Web.Http.RouteParameter.Optional }
);
```

For more information about routing tables, see Routing in ASP.NET Web API.

Add Client-Side AJAX

That's all you need to create a web API that clients can access. Now let's add an HTML page that uses jQuery to call the API.

Make sure your master page (for example, *Site.Master*) includes a `ContentPlaceHolder` with `ID="HeadContent"`:

```
<asp:ContentPlaceHolder runat="server" ID="HeadContent"></asp:ContentPlaceHolder>
```

Open the file Default.aspx. Replace the boilerplate text that is in the main content section, as shown:

```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.Master"
    AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="WebForms._Default" %>

<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
</asp:Content>

<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <h2>Products</h2>
    <table>
    <thead>
        <tr><th>Name</th><th>Price</th></tr>
    </thead>
    <tbody id="products">
    </tbody>
    </table>
</asp:Content>
```

Next, add a reference to the jQuery source file in the `HeaderContent` section:

```
<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
    <script src="Scripts/jquery-1.10.2.min.js" type="text/javascript"></script>
</asp:Content>
```

Note: You can easily add the script reference by dragging and dropping the file from **Solution Explorer** into the code editor window.

Below the jQuery script tag, add the following script block:

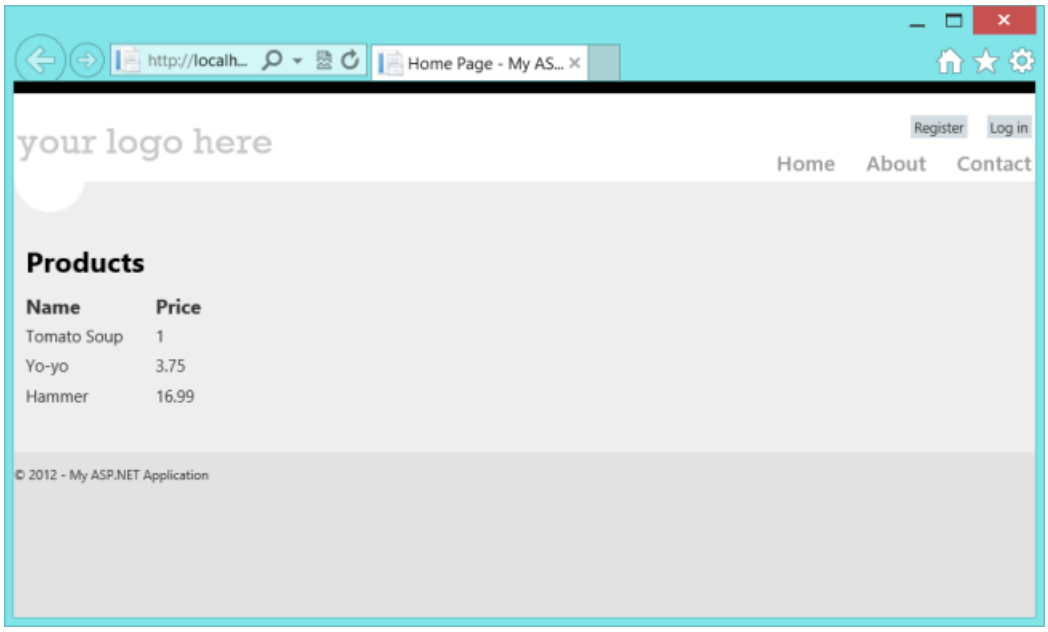
```
<script type="text/javascript">
    function getProducts() {
        $.getJSON("api/products",
            function (data) {
                $('#products').empty(); // Clear the table body.

                // Loop through the list of products.
                $.each(data, function (key, val) {
                    // Add a table row for the product.
                    var row = '<td>' + val.Name + '</td><td>' + val.Price + '</td>';
                    $('<tr/>', { html: row })  // Append the name.
                        .appendTo($('#products'));
                });
            });
        }

        $(document).ready(getProducts);
</script>
```

When the document loads, this script makes an AJAX request to "api/products". The request returns a list of products in JSON format. The script adds the product information to the HTML table.

When you run the application, it should look like this:



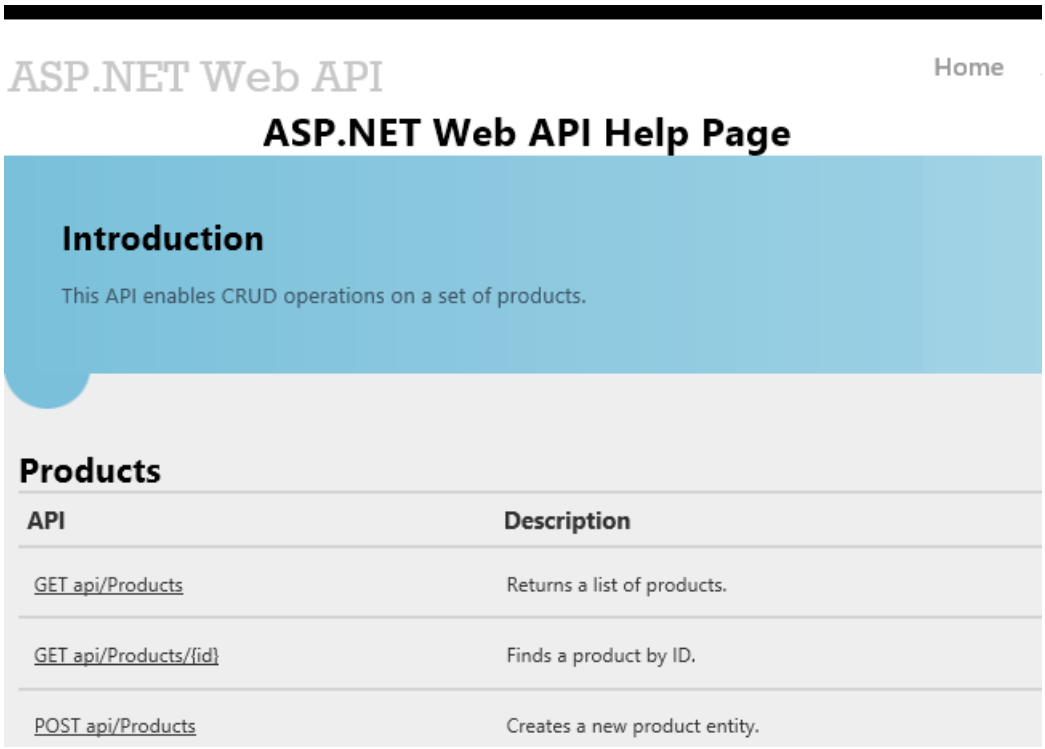
Creating Help Pages for ASP.NET Web API

2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson

This tutorial with code shows how to create help pages for ASP.NET Web API in ASP.NET 4.x.

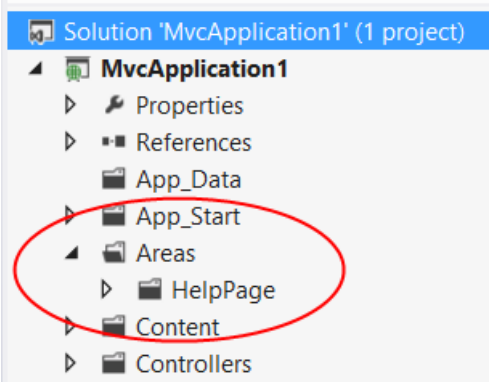
When you create a web API, it is often useful to create a help page, so that other developers will know how to call your API. You could create all of the documentation manually, but it is better to autogenerate as much as possible. To make this task easier, ASP.NET Web API provides a library for auto-generating help pages at run time.



Creating API Help Pages

Install ASP.NET and Web Tools 2012.2 Update. This update integrates help pages into the Web API project template.

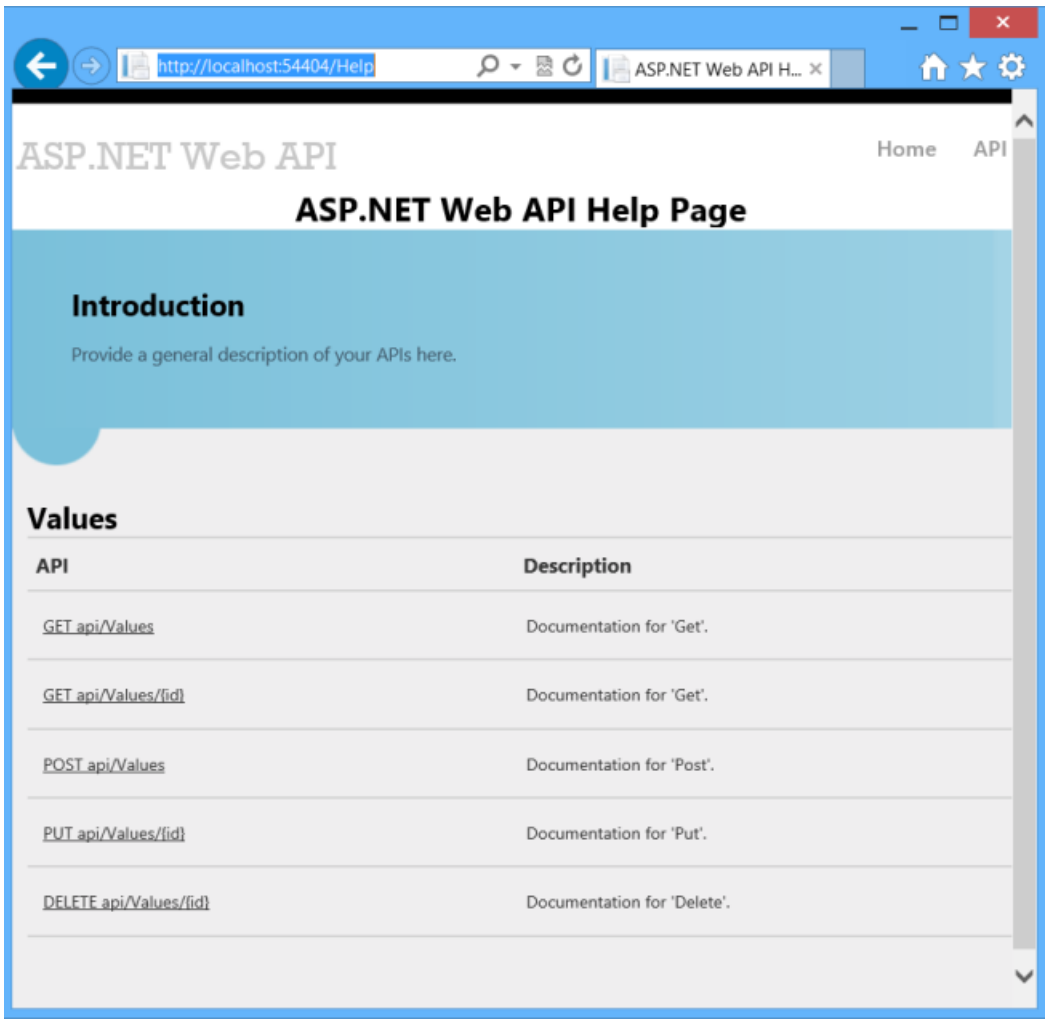
Next, create a new ASP.NET MVC 4 project and select the Web API project template. The project template creates an example API controller named `ValuesController`. The template also creates the API help pages. All of the code files for the help page are placed in the Areas folder of the project.



When you run the application, the home page contains a link to the API help page. From the home page, the

relative path is /Help.

This link brings you to an API summary page.



The MVC view for this page is defined in Areas/HelpPage/Views/Help/Index.cshtml. You can edit this page to modify the layout, introduction, title, styles, and so forth.

The main part of the page is a table of APIs, grouped by controller. The table entries are generated dynamically, using the **IApiExplorer** interface. (I'll talk more about this interface later.) If you add a new API controller, the table is automatically updated at run time.

The "API" column lists the HTTP method and relative URI. The "Description" column contains documentation for each API. Initially, the documentation is just placeholder text. In the next section, I'll show you how to add documentation from XML comments.

Each API has a link to a page with more detailed information, including example request and response bodies.

Adding Help Pages to an Existing Project

You can add help pages to an existing Web API project by using NuGet Package Manager. This option is useful you start from a different project template than the "Web API" template.

From the **Tools** menu, select **NuGet Package Manager**, and then select **Package Manager Console**. In the Package Manager Console window, type one of the following commands:

For a **C#** application: `Install-Package Microsoft.AspNet.WebApi.HelpPage`

For a **Visual Basic** application: `Install-Package Microsoft.AspNet.WebApi.HelpPage.VB`

There are two packages, one for C# and one for Visual Basic. Make sure to use the one that matches your project.

This command installs the necessary assemblies and adds the MVC views for the help pages (located in the Areas/HelpPage folder). You'll need to manually add a link to the Help page. The URI is /Help. To create a link in a razor view, add the following:

```
@Html.ActionLink("API", "Index", "Help", new { area = "" }, null)
```

Also, make sure to register areas. In the Global.asax file, add the following code to the **Application_Start** method, if it is not there already:

```
protected void Application_Start()
{
    // Add this code, if not present.
    AreaRegistration.RegisterAllAreas();

    // ...
}
```

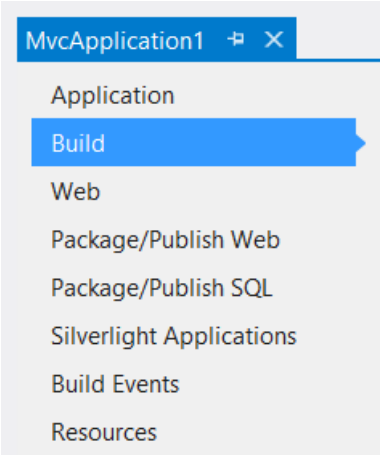
Adding API Documentation

By default, the help pages have placeholder strings for documentation. You can use XML documentation

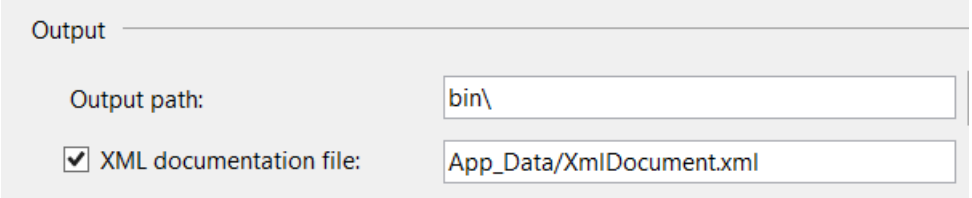
comments to create the documentation. To enable this feature, open the file Areas/HelpPage/App_Start/HelpPageConfig.cs and uncomment the following line:

```
config.SetDocumentationProvider(new XmlDocumentationProvider(
    HttpContext.Current.Server.MapPath("~/App_Data/XmlDocument.xml")));
```

Now enable XML documentation. In Solution Explorer, right-click the project and select **Properties**. Select the **Build** page.



Under **Output**, check **XML documentation file**. In the edit box, type "App_Data/XmlDocument.xml".



Next, open the code for the `ValuesController` API controller, which is defined in /Controllers/ValuesController.cs. Add some documentation comments to the controller methods. For example:

```
/// <summary>
/// Gets some very important data from the server.
/// </summary>
public IEnumerable<string> Get()
{
    return new string[] { "value1", "value2" };
}

/// <summary>
/// Looks up some data by ID.
/// </summary>
/// <param name="id">The ID of the data.</param>
public string Get(int id)
{
    return "value";
}
```

> **NOTE**
>
> Tip: If you position the caret on the line above the method and type three forward slashes, Visual Studio automatically inserts the XML elements. Then you can fill in the blanks.

Now build and run the application again, and navigate to the help pages. The documentation strings should

appear in the API table.

API	Description
GET api/Values	Gets some very important data from the server.
GET api/Values/{id}	Looks up some data by ID.

The help page reads the strings from the XML file at run time. (When you deploy the application, make sure to deploy the XML file.)

Under the Hood

The help pages are built on top of the **ApiExplorer** class, which is part of the Web API framework. The **ApiExplorer** class provides the raw material for creating a help page. For each API, **ApiExplorer** contains an **ApiDescription** that describes the API. For this purpose, an "API" is defined as the combination of HTTP method and relative URI. For example, here are some distinct APIs:

- GET /api/Products
- GET /api/Products/{id}
- POST /api/Products

If a controller action supports multiple HTTP methods, the **ApiExplorer** treats each method as a distinct API.

To hide an API from the **ApiExplorer**, add the **ApiExplorerSettings** attribute to the action and set *IgnoreApi* to true.

```
[ApiExplorerSettings(IgnoreApi=true)]
public HttpResponseMessage Get(int id) {  }
```

You can also add this attribute to the controller, to exclude the entire controller.

The ApiExplorer class gets documentation strings from the **IDocumentationProvider** interface. As you saw earlier, the Help Pages library provides an **IDocumentationProvider** that gets documentation from XML documentation strings. The code is located in /Areas/HelpPage/XmlDocumentationProvider.cs. You can get documentation from another source by writing your own **IDocumentationProvider**. To wire it up, call the **SetDocumentationProvider** extension method, defined in **HelpPageConfigurationExtensions**

ApiExplorer automatically calls into the **IDocumentationProvider** interface to get documentation strings for each API. It stores them in the **Documentation** property of the **ApiDescription** and **ApiParameterDescription** objects.

Next Steps

You aren't limited to the help pages shown here. In fact, **ApiExplorer** is not limited to creating help pages. Yao Huang Lin has written some great blog posts to get you thinking out of the box:

- Adding a simple Test Client to ASP.NET Web API Help Page
- Making ASP.NET Web API Help Page work on self-hosted services
- Design-time generation of help page (or client) for ASP.NET Web API
- Advanced Help Page customizations

Hands On Lab: Build a Single Page Application (SPA) with ASP.NET Web API and Angular.js

2/20/2020 • 22 minutes to read • Edit Online

by Web Camps Team

Download Web Camps Training Kit

This hands on lab shows you how to build a Single Page Application (SPA) with ASP.NET Web API and Angular.js for ASP.NET 4.x.

In this hand-on lab, you will take advantage of those technologies to implement Geek Quiz, a trivia website based on the SPA concept. You will first implement the service layer with ASP.NET Web API to expose the required endpoints to retrieve the quiz questions and store the answers. Then, you will build a rich and responsive UI using AngularJS and CSS3 transformation effects.

In traditional web applications, the client (browser) initiates the communication with the server by requesting a page. The server then processes the request and sends the HTML of the page to the client. In subsequent interactions with the page –e.g. the user navigates to a link or submits a form with data– a new request is sent to the server, and the flow starts again: the server processes the request and sends a new page to the browser in response to the new action requested by the client.

> In Single-Page Applications (SPAs) the entire page is loaded in the browser after the initial request, but subsequent interactions take place through Ajax requests. This means that the browser has to update only the portion of the page that has changed; there is no need to reload the entire page. The SPA approach reduces the time taken by the application to respond to user actions, resulting in a more fluid experience.
>
> The architecture of a SPA involves certain challenges that are not present in traditional web applications. However, emerging technologies like ASP.NET Web API, JavaScript frameworks like AngularJS and new styling features provided by CSS3 make it really easy to design and build SPAs.
>
> All sample code and snippets are included in the Web Camps Training Kit, available at https://aka.ms/webcamps-training-kit.

Overview

Objectives

In this hands-on lab, you will learn how to:

- Create an ASP.NET Web API service to send and receive JSON data
- Create a responsive UI using AngularJS
- Enhance the UI experience with CSS3 transformations

Prerequisites

The following is required to complete this hands-on lab:

- Visual Studio Express 2013 for Web or greater

Setup

In order to run the exercises in this hands-on lab, you will need to set up your environment first.

1. Open Windows Explorer and browse to the lab's **Source** folder.

2. Right-click on **Setup.cmd** and select **Run as administrator** to launch the setup process that will configure your environment and install the Visual Studio code snippets for this lab.
3. If the User Account Control dialog box is shown, confirm the action to proceed.

> **NOTE**
> Make sure you have checked all the dependencies for this lab before running the setup.

Using the Code Snippets

Throughout the lab document, you will be instructed to insert code blocks. For your convenience, most of this code is provided as Visual Studio Code Snippets, which you can access from within Visual Studio 2013 to avoid having to add it manually.

> **NOTE**
> Each exercise is accompanied by a starting solution located in the **Begin** folder of the exercise that allows you to follow each exercise independently of the others. Please be aware that the code snippets that are added during an exercise are missing from these starting solutions and may not work until you have completed the exercise. Inside the source code for an exercise, you will also find an **End** folder containing a Visual Studio solution with the code that results from completing the steps in the corresponding exercise. You can use these solutions as guidance if you need additional help as you work through this hands-on lab.

Exercises

This hands-on lab includes the following exercises:

1. Creating a Web API
2. Creating a SPA Interface

Estimated time to complete this lab: **60 minutes**

> **NOTE**
> When you first start Visual Studio, you must select one of the predefined settings collections. Each predefined collection is designed to match a particular development style and determines window layouts, editor behavior, IntelliSense code snippets, and dialog box options. The procedures in this lab describe the actions necessary to accomplish a given task in Visual Studio when using the **General Development Settings** collection. If you choose a different settings collection for your development environment, there may be differences in the steps that you should take into account.

Exercise 1: Creating a Web API

One of the key parts of a SPA is the service layer. It is responsible for processing the Ajax calls sent by the UI and returning data in response to that call. The data retrieved should be presented in a machine-readable format in order to be parsed and consumed by the client.

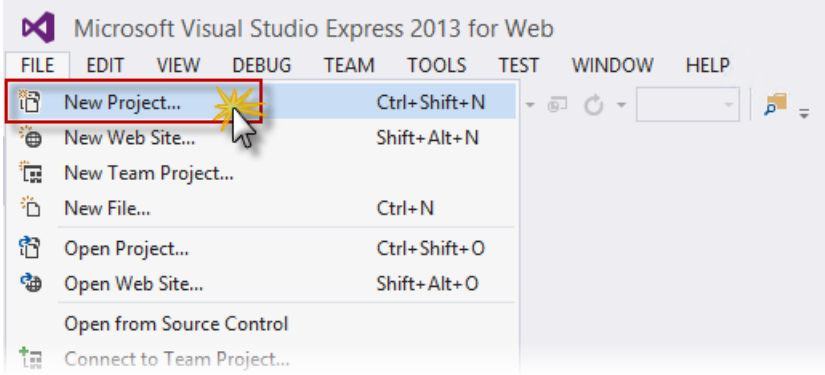
The Web API framework is part of the ASP.NET Stack and is designed to make it easy to implement HTTP services, generally sending and receiving JSON- or XML-formatted data through a RESTful API. In this exercise you will create the Web site to host the Geek Quiz application and then implement the back-end service to expose and persist the quiz data using ASP.NET Web API.

Task 1 – Creating the Initial Project for Geek Quiz

In this task you will start creating a new ASP.NET MVC project with support for ASP.NET Web API based on the **One ASP.NET** project type that comes with Visual Studio. **One ASP.NET** unifies all ASP.NET technologies and gives you the option to mix and match them as desired. You will then add the Entity Framework's model classes

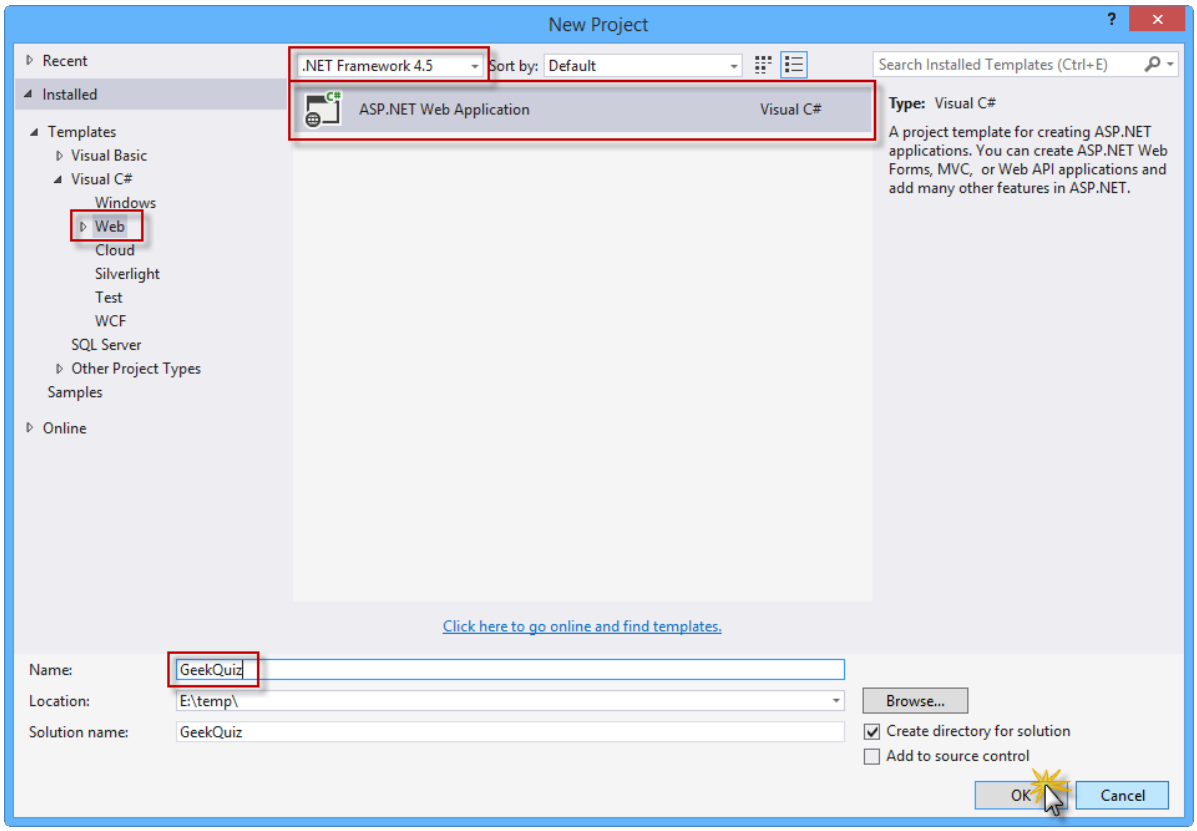
and the database initializer to insert the quiz questions.

1. Open **Visual Studio Express 2013 for Web** and select **File | New Project...** to start a new solution.



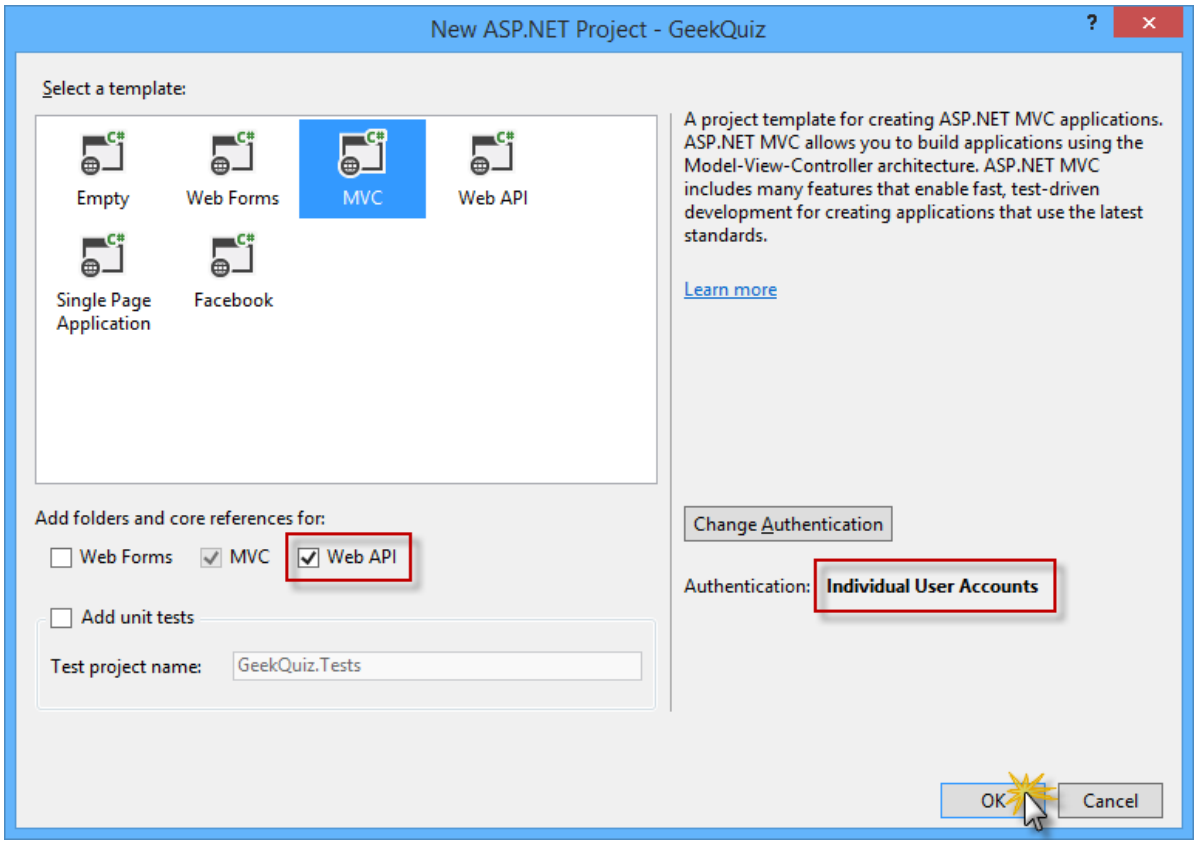
Creating a New Project

2. In the **New Project** dialog box, select **ASP.NET Web Application** under the **Visual C# | Web** tab. Make sure **.NET Framework 4.5** is selected, name it *GeekQuiz*, choose a **Location** and click **OK**.



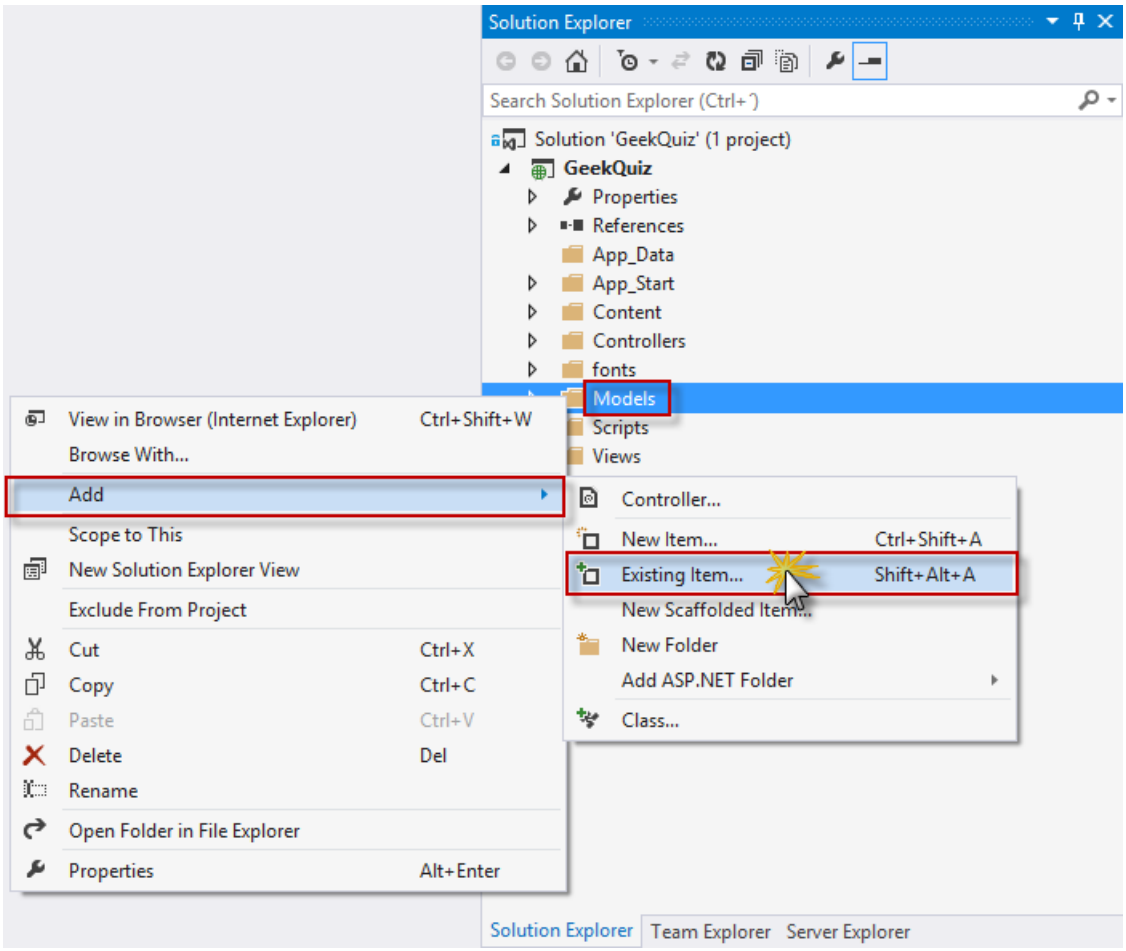
Creating a new ASP.NET Web Application project

3. In the **New ASP.NET Project** dialog box, select the **MVC** template and select the **Web API** option. Also, make sure that the **Authentication** option is set to **Individual User Accounts**. Click **OK** to continue.



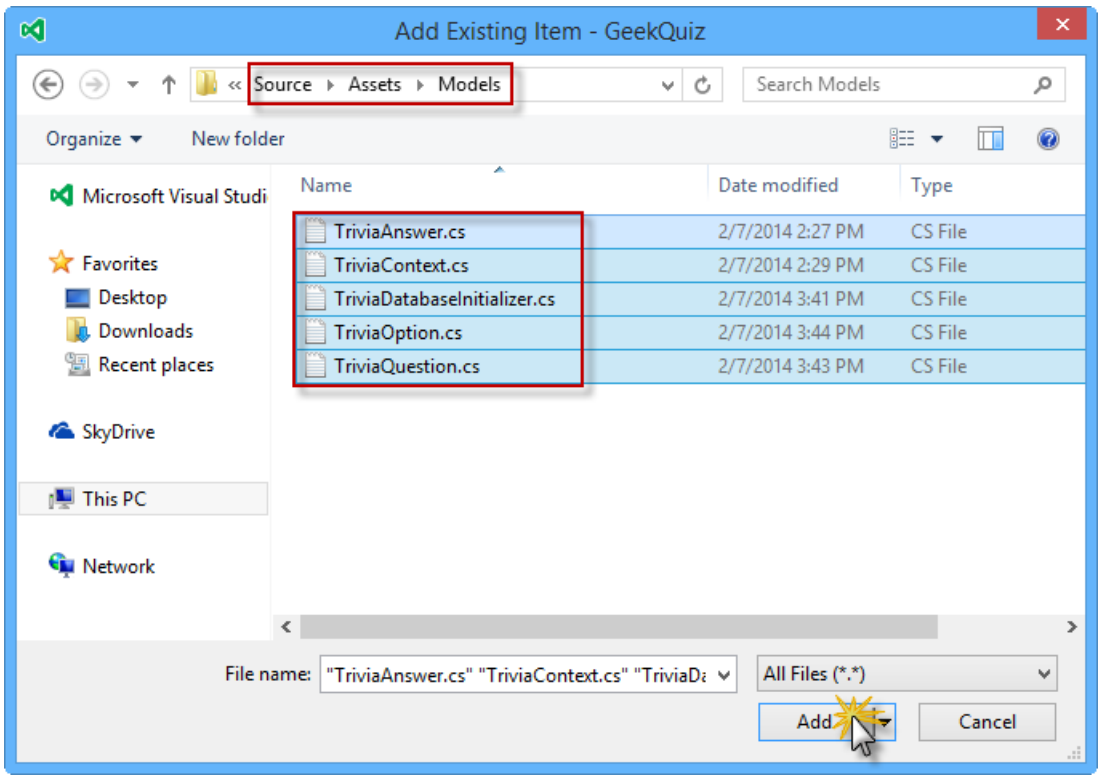
Creating a new project with the MVC template, including Web API components

4. In **Solution Explorer**, right-click the **Models** folder of the **GeekQuiz** project and select **Add | Existing Item...**.



Adding an existing item

5. In the **Add Existing Item** dialog box, navigate to the **Source/Assets/Models** folder and select all the files. Click **Add**.



Adding the model assets

> **NOTE**
>
> By adding these files, you are adding the data model, the Entity Framework's database context and the database initializer for the Geek Quiz application.
>
> **Entity Framework (EF)** is an object-relational mapper (ORM) that enables you to create data access applications by programming with a conceptual application model instead of programming directly using a relational storage schema. You can learn more about Entity Framework here.
>
> The following is a description of the classes you just added:
>
> - **TriviaOption:** represents a single option associated with a quiz question
> - **TriviaQuestion:** represents a quiz question and exposes the associated options through the **Options** property
> - **TriviaAnswer:** represents the option selected by the user in response to a quiz question
> - **TriviaContext:** represents the Entity Framework's database context of the Geek Quiz application. This class derives from **DContext** and exposes **DbSet** properties that represent collections of the entities described above.
> - **TriviaDatabaseInitializer:** the implementation of the Entity Framework initializer for the **TriviaContext** class which inherits from **CreateDatabaseIfNotExists**. The default behavior of this class is to create the database only if it does not exist, inserting the entities specified in the **Seed** method.

6. Open the **Global.asax.cs** file and add the following using statement.

```
using GeekQuiz.Models;
```

7. Add the following code at the beginning of the **Application_Start** method to set the **TriviaDatabaseInitializer** as the database initializer.

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        System.Data.Entity.Database.SetInitializer(new TriviaDatabaseInitializer());

        AreaRegistration.RegisterAllAreas();
        GlobalConfiguration.Configure(WebApiConfig.Register);
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

8. Modify the **Home** controller to restrict access to authenticated users. To do this, open the **HomeController.cs** file inside the **Controllers** folder and add the **Authorize** attribute to the **HomeController** class definition.

```
namespace GeekQuiz.Controllers
{
    [Authorize]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        ...
    }
}
```

> **NOTE**
>
> The **Authorize** filter checks to see if the user is authenticated. If the user is not authenticated, it returns HTTP status code 401 (Unauthorized) without invoking the action. You can apply the filter globally, at the controller level, or at the level of individual actions.

9. You will now customize the layout of the web pages and the branding. To do this, open the **_Layout.cshtml** file inside the **Views | Shared** folder and update the content of the **<title>** element by replacing *My ASP.NET Application* with *Geek Quiz*.

```
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - Geek Quiz</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")

</head>
```

10. In the same file, update the navigation bar by removing the *About* and *Contact* links and renaming the *Home* link to *Play*. Additionally, rename the *Application name* link to *Geek Quiz*. The HTML for the navigation bar should look like the following code.

```
<div class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            @Html.ActionLink("Geek Quiz", "Index", "Home", null, new { @class = "navbar-brand" })
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li>@Html.ActionLink("Play", "Index", "Home")</li>
            </ul>
            @Html.Partial("_LoginPartial")
        </div>
    </div>
</div>
```

11. Update the footer of the layout page by replacing *My ASP.NET Application* with *Geek Quiz*. To do this, replace the content of the **<footer>** element with the following highlighted code.

```
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; @DateTime.Now.Year - Geek Quiz</p>
    </footer>
</div>
```

Task 2 – Creating the TriviaController Web API

In the previous task, you created the initial structure of the Geek Quiz web application. You will now build a simple Web API service that interacts with the quiz data model and exposes the following actions:

- **GET /api/trivia**: Retrieves the next question from the quiz list to be answered by the authenticated user.
- **POST /api/trivia**: Stores the quiz answer specified by the authenticated user.

You will use the ASP.NET Scaffolding tools provided by Visual Studio to create the baseline for the Web API controller class.

1. Open the **WebApiConfig.cs** file inside the **App_Start** folder. This file defines the configuration of the Web API service, like how routes are mapped to Web API controller actions.

2. Add the following using statement at the beginning of the file.

```
using Newtonsoft.Json.Serialization;
```

3. Add the following highlighted code to the **Register** method to globally configure the formatter for the JSON data retrieved by the Web API action methods.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // Web API configuration and services

        // Use camel case for JSON data.
        config.Formatters.JsonFormatter.SerializerSettings.ContractResolver = new
CamelCasePropertyNamesContractResolver();

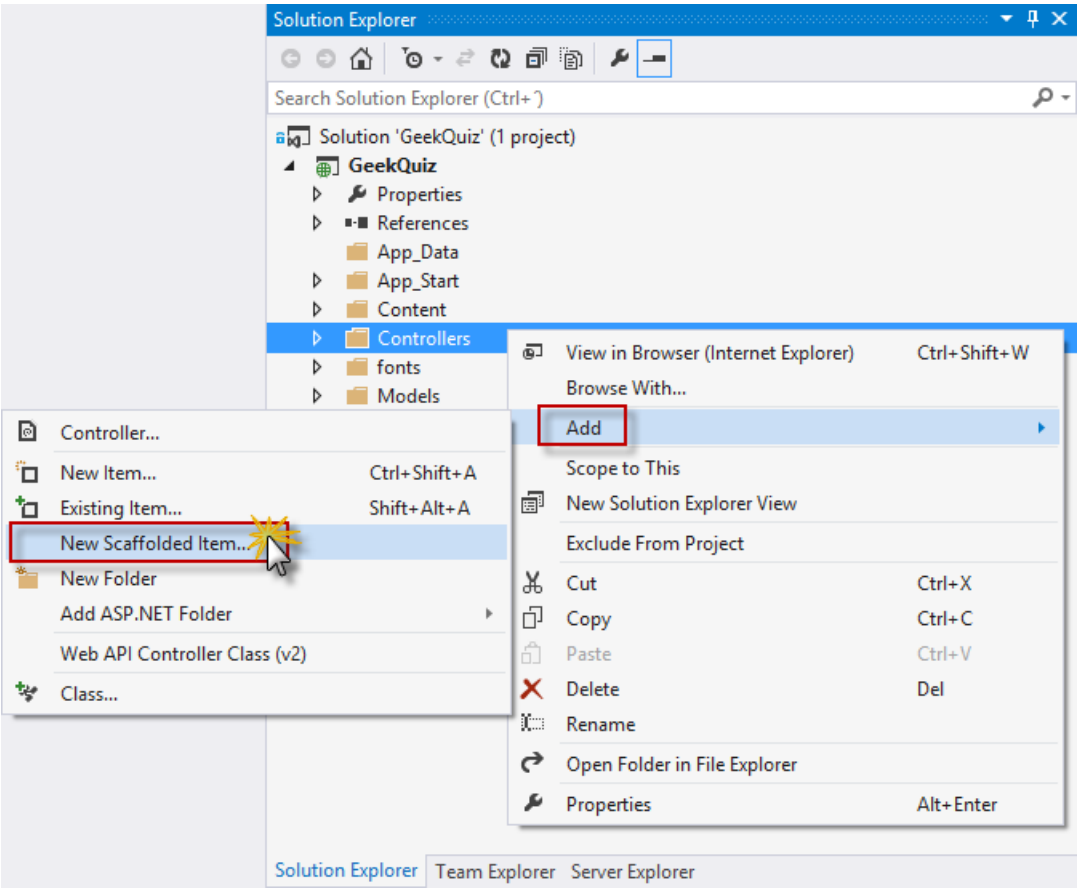
        // Web API routes
        config.MapHttpAttributeRoutes();

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

NOTE

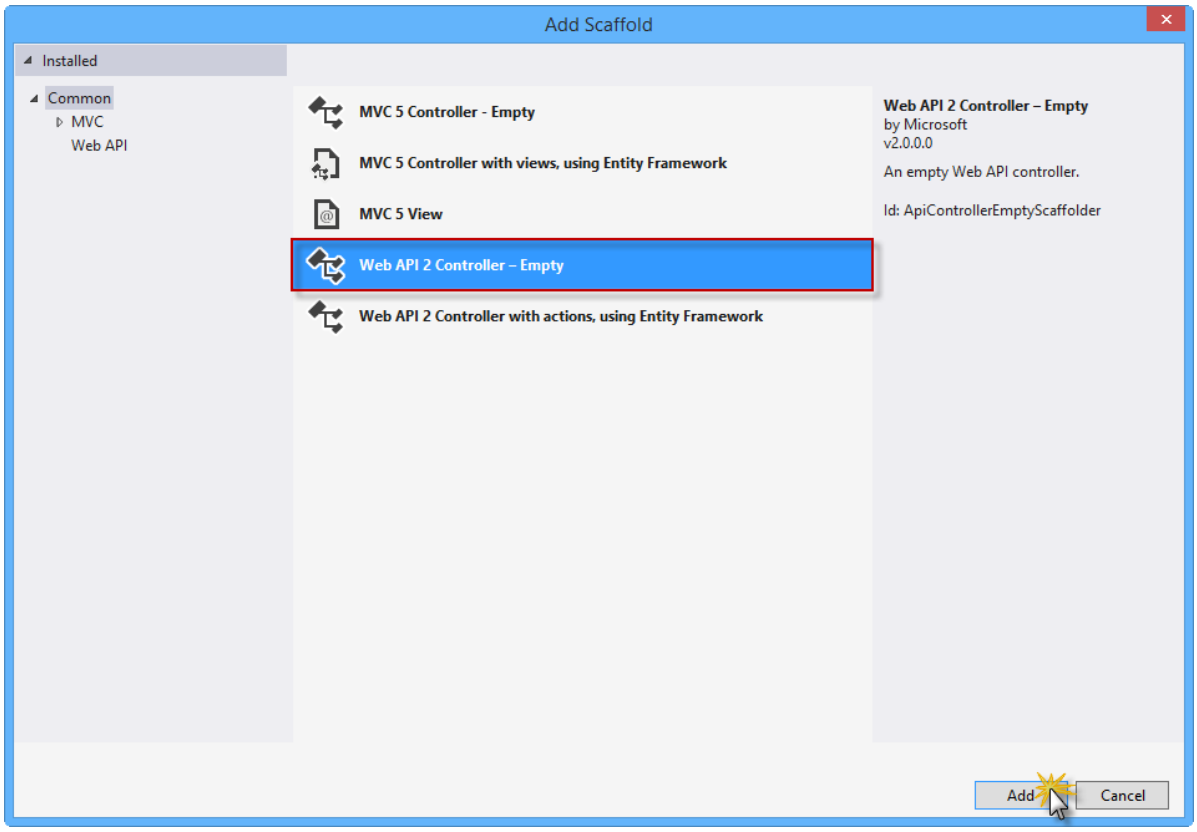
The **CamelCasePropertyNamesContractResolver** automatically converts property names to *camel* case, which is the general convention for property names in JavaScript.

4. In **Solution Explorer**, right-click the **Controllers** folder of the **GeekQuiz** project and select **Add | New Scaffolded Item...**.



Creating a new scaffolded item

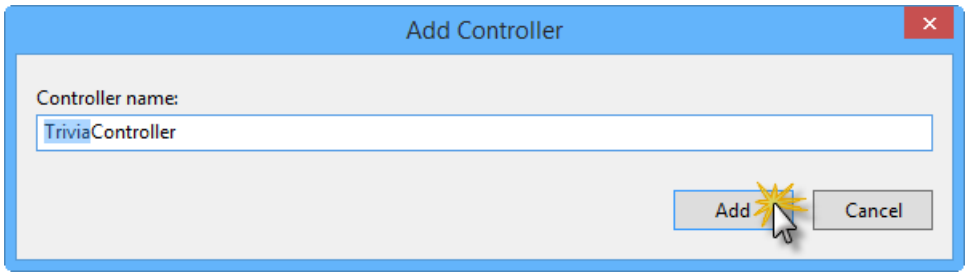
5. In the **Add Scaffold** dialog box, make sure that the **Common** node is selected in the left pane. Then, select the **Web API 2 Controller - Empty** template in the center pane and click **Add**.



Selecting the Web API 2 Controller Empty template

> **NOTE**
>
> **ASP.NET Scaffolding** is a code generation framework for ASP.NET Web applications. Visual Studio 2013 includes pre-installed code generators for MVC and Web API projects. You should use scaffolding in your project when you want to quickly add code that interacts with data models in order to reduce the amount of time required to develop standard data operations.
>
> The scaffolding process also ensures that all the required dependencies are installed in the project. For example, if you start with an empty ASP.NET project and then use scaffolding to add a Web API controller, the required Web API NuGet packages and references are added to your project automatically.

6. In the **Add Controller** dialog box, type *TriviaController* in the **Controller name** text box and click **Add**.



Adding the Trivia Controller

7. The **TriviaController.cs** file is then added to the **Controllers** folder of the **GeekQuiz** project, containing an empty **TriviaController** class. Add the following using statements at the beginning of the file.

   (Code Snippet - *AspNetWebApiSpa - Ex1 - TriviaControllerUsings*)

```
using System.Data.Entity;
using System.Threading;
using System.Threading.Tasks;
using System.Web.Http.Description;
using GeekQuiz.Models;
```

8. Add the following code at the beginning of the **TriviaController** class to define, initialize and dispose the **TriviaContext** instance in the controller.

   (Code Snippet - *AspNetWebApiSpa - Ex1 - TriviaControllerContext*)

```
public class TriviaController : ApiController
{
    private TriviaContext db = new TriviaContext();

    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            this.db.Dispose();
        }

        base.Dispose(disposing);
    }
}
```

> **NOTE**
> The **Dispose** method of **TriviaController** invokes the **Dispose** method of the **TriviaContext** instance, which ensures that all the resources used by the context object are released when the **TriviaContext** instance is disposed or garbage-collected. This includes closing all database connections opened by Entity Framework.

9. Add the following helper method at the end of the **TriviaController** class. This method retrieves the following quiz question from the database to be answered by the specified user.

   (Code Snippet - *AspNetWebApiSpa - Ex1 - TriviaControllerNextQuestion*)

```
private async Task<TriviaQuestion> NextQuestionAsync(string userId)
{
    var lastQuestionId = await this.db.TriviaAnswers
        .Where(a => a.UserId == userId)
        .GroupBy(a => a.QuestionId)
        .Select(g => new { QuestionId = g.Key, Count = g.Count() })
        .OrderByDescending(q => new { q.Count, QuestionId = q.QuestionId })
        .Select(q => q.QuestionId)
        .FirstOrDefaultAsync();

    var questionsCount = await this.db.TriviaQuestions.CountAsync();

    var nextQuestionId = (lastQuestionId % questionsCount) + 1;
    return await this.db.TriviaQuestions.FindAsync(CancellationToken.None, nextQuestionId);
}
```

10. Add the following **Get** action method to the **TriviaController** class. This action method calls the **NextQuestionAsync** helper method defined in the previous step to retrieve the next question for the authenticated user.

    (Code Snippet - *AspNetWebApiSpa - Ex1 - TriviaControllerGetAction*)

```
// GET api/Trivia
[ResponseType(typeof(TriviaQuestion))]
public async Task<IHttpActionResult> Get()
{
    var userId = User.Identity.Name;

    TriviaQuestion nextQuestion = await this.NextQuestionAsync(userId);

    if (nextQuestion == null)
    {
        return this.NotFound();
    }

    return this.Ok(nextQuestion);
}
```

11. Add the following helper method at the end of the **TriviaController** class. This method stores the specified answer in the database and returns a Boolean value indicating whether or not the answer is correct.

    (Code Snippet - *AspNetWebApiSpa - Ex1 - TriviaControllerStoreAsync*)

```
private async Task<bool> StoreAsync(TriviaAnswer answer)
{
    this.db.TriviaAnswers.Add(answer);

    await this.db.SaveChangesAsync();
    var selectedOption = await this.db.TriviaOptions.FirstOrDefaultAsync(o => o.Id == answer.OptionId
        && o.QuestionId == answer.QuestionId);

    return selectedOption.IsCorrect;
}
```

12. Add the following **Post** action method to the **TriviaController** class. This action method associates the answer to the authenticated user and calls the **StoreAsync** helper method. Then, it sends a response with the Boolean value returned by the helper method.

    (Code Snippet - *AspNetWebApiSpa - Ex1 - TriviaControllerPostAction*)

```
// POST api/Trivia
[ResponseType(typeof(TriviaAnswer))]
public async Task<IHttpActionResult> Post(TriviaAnswer answer)
{
    if (!ModelState.IsValid)
    {
        return this.BadRequest(this.ModelState);
    }

    answer.UserId = User.Identity.Name;

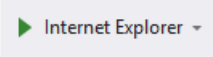
    var isCorrect = await this.StoreAsync(answer);
    return this.Ok<bool>(isCorrect);
}
```

13. Modify the Web API controller to restrict access to authenticated users by adding the **Authorize** attribute to the **TriviaController** class definition.

```
[Authorize]
public class TriviaController : ApiController
{
    ...
}
```

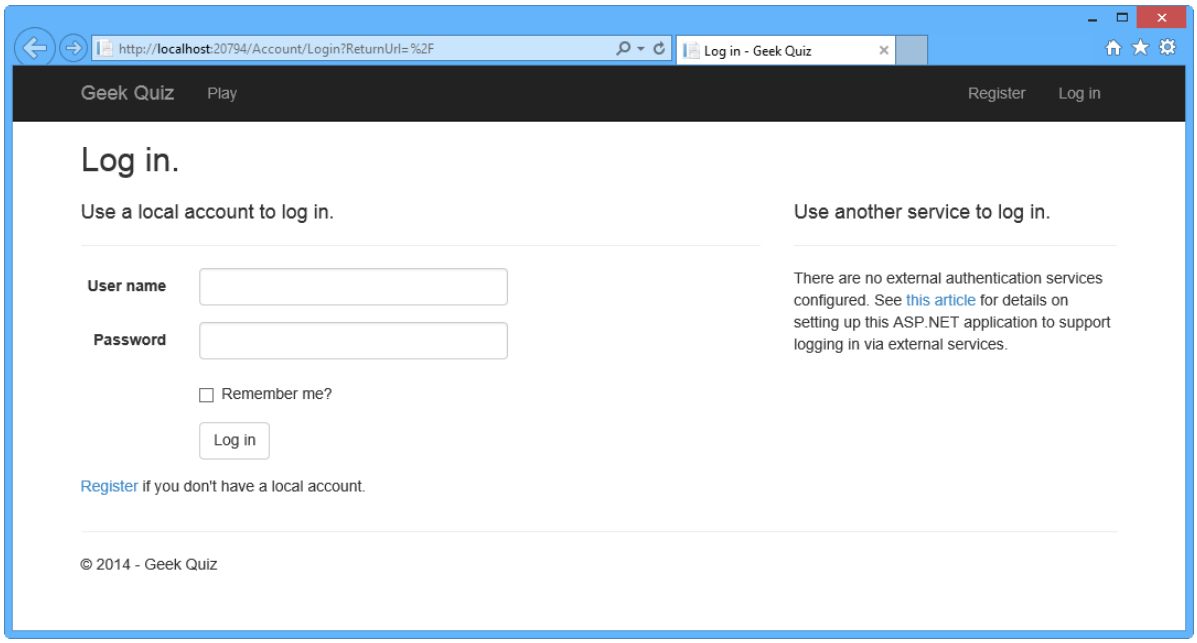
Task 3 – Running the Solution

In this task you will verify that the Web API service you built in the previous task is working as expected. You will use the Internet Explorer **F12 Developer Tools** to capture the network traffic and inspect the full response from the Web API service.

> **NOTE**
>
> Make sure that **Internet Explorer** is selected in the **Start** button located on the Visual Studio toolbar.
>
> 
>

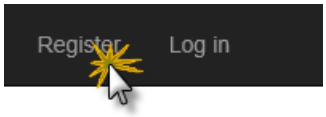
1. Press **F5** to run the solution. The **Log in** page should appear in the browser.

> **NOTE**
>
> When the application starts, the default MVC route is triggered, which by default is mapped to the **Index** action of the **HomeController** class. Since **HomeController** is restricted to authenticated users (remember that you decorated that class with the **Authorize** attribute in Exercise 1) and there is no user authenticated yet, the application redirects the original request to the log in page.



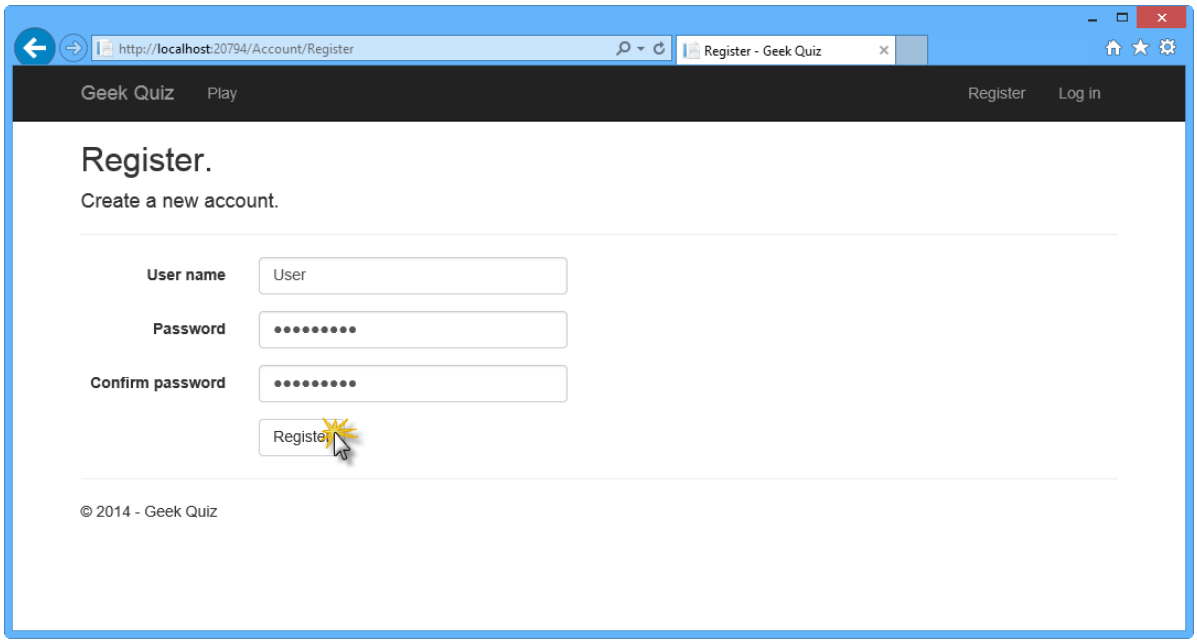
Running the solution

2. Click **Register** to create a new user.



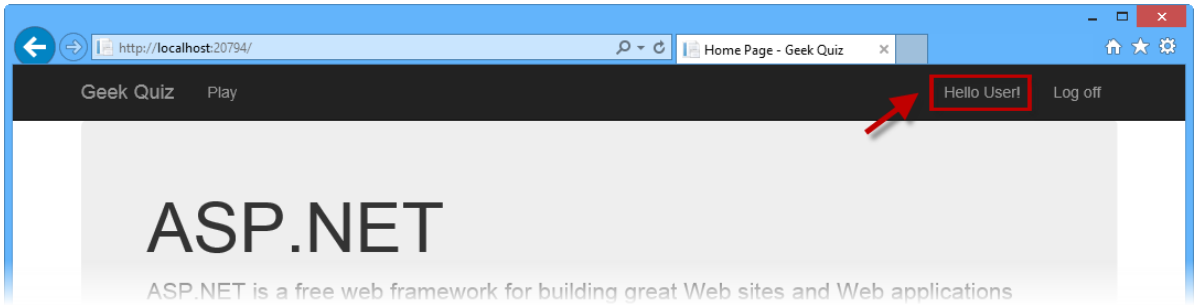
Registering a new user

3. In the **Register** page, enter a **User name** and **Password**, and then click **Register**.



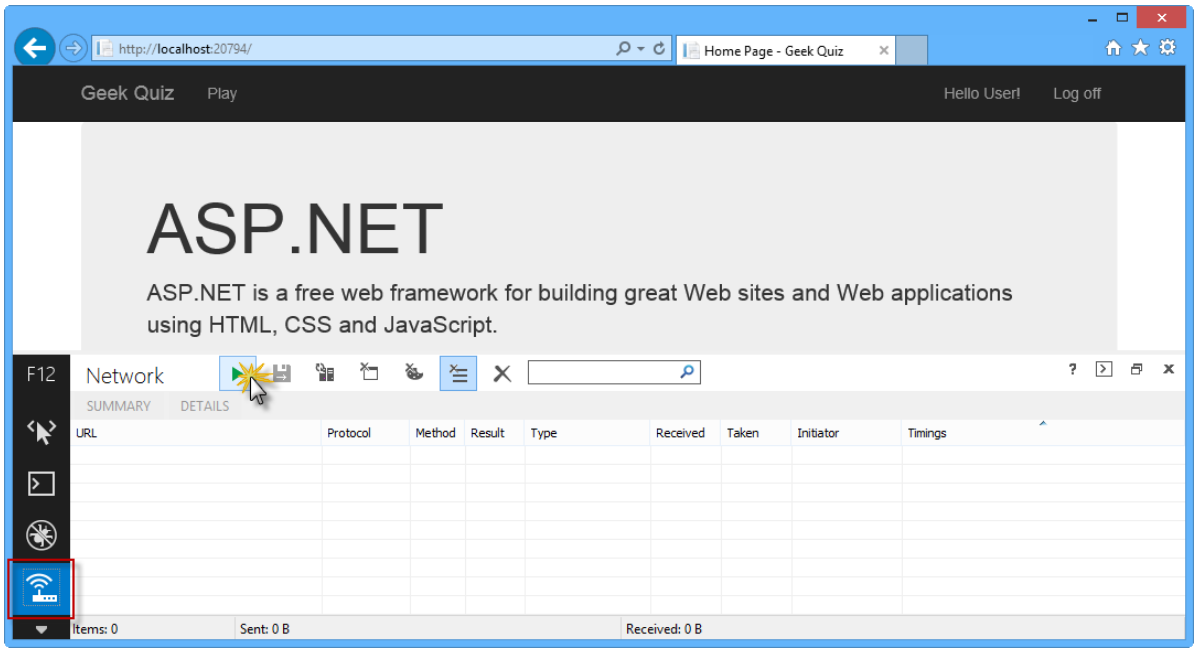
Register page

4. The application registers the new account and the user is authenticated and redirected back to the home page.



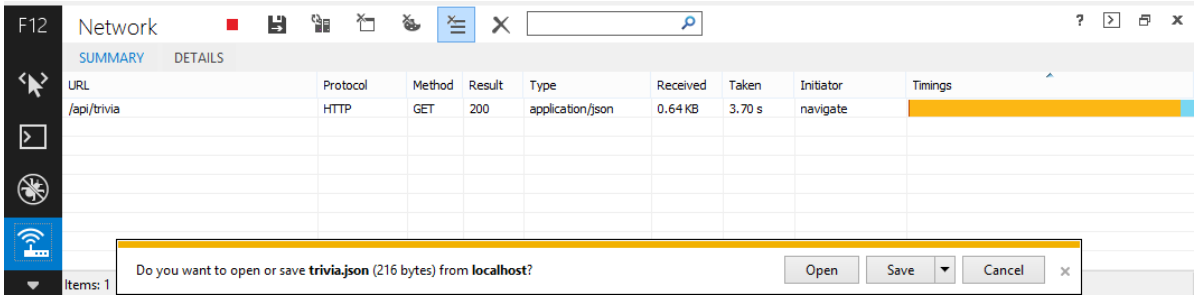
User is authenticated

5. In the browser, press **F12** to open the **Developer Tools** panel. Press **CTRL + 4** or click the **Network** icon, and then click the green arrow button to begin capturing network traffic.



Initiating Web API network capture

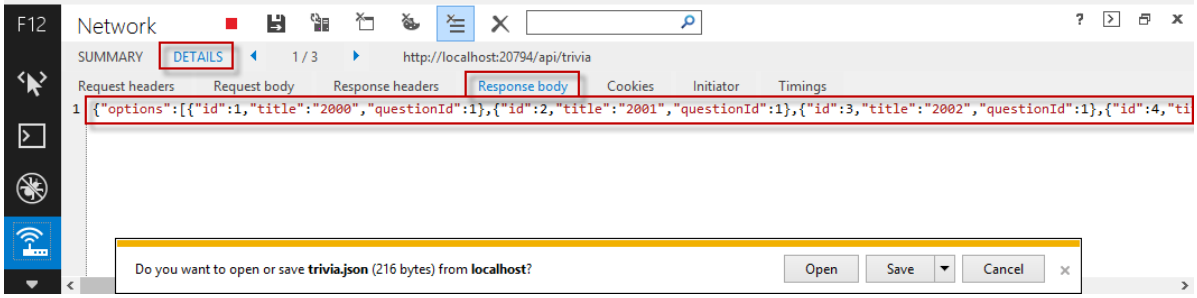
6. Append **api/trivia** to the URL in the browser's address bar. You will now inspect the details of the response from the **Get** action method in **TriviaController**.



Retrieving the next question data through Web API

> **NOTE**
>
> Once the download finishes, you will be prompted to make an action with the downloaded file. Leave the dialog box open in order to be able to watch the response content through the Developers Tool window.

7. Now you will inspect the body of the response. To do this, click the **Details** tab and then click **Response body**. You can check that the downloaded data is an object with the properties **options** (which is a list of **TriviaOption** objects), **id** and **title** that correspond to the **TriviaQuestion** class.



Viewing Web API Response Body

8. Go back to Visual Studio and press **SHIFT + F5** to stop debugging.

Exercise 2: Creating the SPA Interface

In this exercise you will first build the web front-end portion of Geek Quiz, focusing on the Single-Page Application interaction using **AngularJS**. You will then enhance the user experience with CSS3 to perform rich animations and provide a visual effect of context switching when transitioning from one question to the next.

Task 1 – Creating the SPA Interface Using AngularJS

In this task you will use **AngularJS** to implement the client side of the Geek Quiz application. **AngularJS** is an open-source JavaScript framework that augments browser-based applications with *Model-View-Controller* (MVC) capability, facilitating both development and testing.

You will start by installing AngularJS from Visual Studio's Package Manager Console. Then, you will create the controller to provide the behavior of the Geek Quiz app and the view to render the quiz questions and answers using the AngularJS template engine.

> **NOTE**
>
> For more information about AngularJS, refer to http://angularjs.org/.

1. Open **Visual Studio Express 2013 for Web** and open the **GeekQuiz.sln** solution located in the **Source/Ex2-CreatingASPAInterface/Begin** folder. Alternatively, you can continue with the solution that

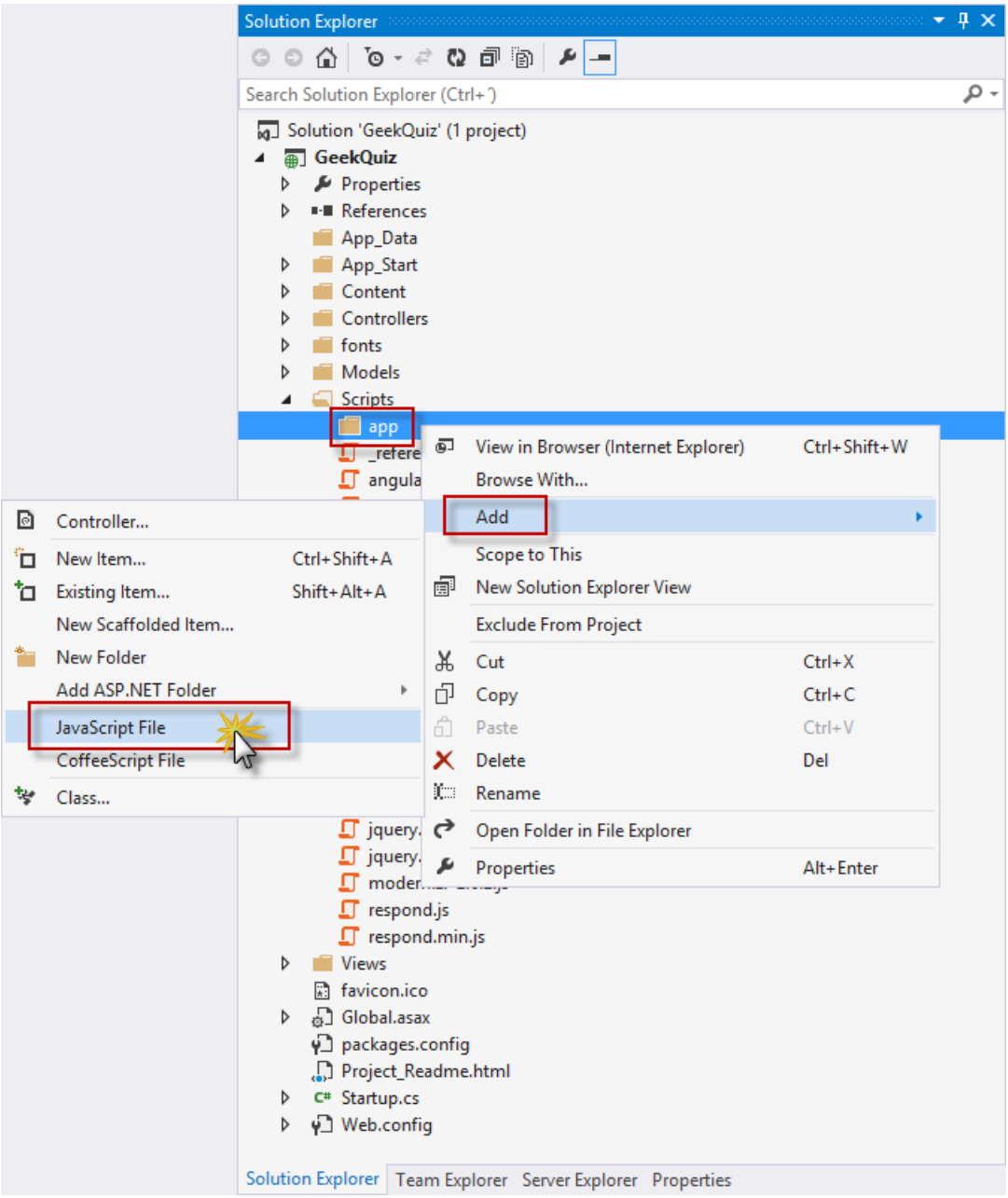
you obtained in the previous exercise.

2. Open the **Package Manager Console** from **Tools** > **NuGet Package Manager**. Type the following command to install the **AngularJS.Core** NuGet package.

```
Install-Package AngularJS.Core
```

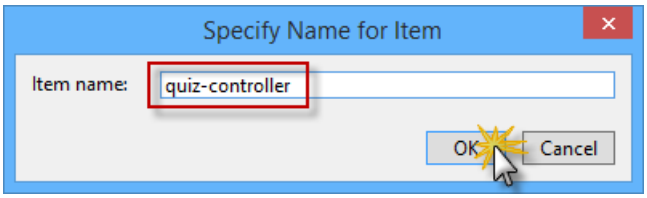
3. In **Solution Explorer**, right-click the **Scripts** folder of the **GeekQuiz** project and select **Add | New Folder**. Name the folder **app** and press **Enter**.

4. Right-click the **app** folder you just created and select **Add | JavaScript File**.



Creating a new JavaScript file

5. In the **Specify Name for Item** dialog box, type *quiz-controller* in the **Item name** text box and click **OK**.



Naming the new JavaScript file

6. In the **quiz-controller.js** file, add the following code to declare and initialize the AngularJS **QuizCtrl** controller.

   (Code Snippet - *AspNetWebApiSpa - Ex2 - AngularQuizController*)

```
angular.module('QuizApp', [])
    .controller('QuizCtrl', function ($scope, $http) {
        $scope.answered = false;
        $scope.title = "loading question...";
        $scope.options = [];
        $scope.correctAnswer = false;
        $scope.working = false;

        $scope.answer = function () {
            return $scope.correctAnswer ? 'correct' : 'incorrect';
        };
    });
```

> **NOTE**
>
> The constructor function of the **QuizCtrl** controller expects an injectable parameter named **$scope**. The initial state of the scope should be set up in the constructor function by attaching properties to the **$scope** object. The properties contain the **view model**, and will be accessible to the template when the controller is registered.
>
> The **QuizCtrl** controller is defined inside a module named **QuizApp**. Modules are units of work that let you break your application into separate components. The main advantages of using modules is that the code is easier to understand and facilitates unit testing, reusability and maintainability.

7. You will now add behavior to the scope in order to react to events triggered from the view. Add the following code at the end of the **QuizCtrl** controller to define the **nextQuestion** function in the **$scope** object.

   (Code Snippet - *AspNetWebApiSpa - Ex2 - AngularQuizControllerNextQuestion*)

```
.controller('QuizCtrl', function ($scope, $http) {
    ...

    $scope.nextQuestion = function () {
        $scope.working = true;
        $scope.answered = false;
        $scope.title = "loading question...";
        $scope.options = [];

        $http.get("/api/trivia").success(function (data, status, headers, config) {
            $scope.options = data.options;
            $scope.title = data.title;
            $scope.answered = false;
            $scope.working = false;
        }).error(function (data, status, headers, config) {
            $scope.title = "Oops... something went wrong";
            $scope.working = false;
        });
    };
};
```

> **NOTE**
>
> This function retrieves the next question from the **Trivia** Web API created in the previous exercise and attaches the question data to the **$scope** object.

8. Insert the following code at the end of the **QuizCtrl** controller to define the **sendAnswer** function in the **$scope** object.

   (Code Snippet - *AspNetWebApiSpa - Ex2 - AngularQuizControllerSendAnswer*)

```
.controller('QuizCtrl', function ($scope, $http) {
    ...

    $scope.sendAnswer = function (option) {
        $scope.working = true;
        $scope.answered = true;

        $http.post('/api/trivia', { 'questionId': option.questionId, 'optionId': option.id
}).success(function (data, status, headers, config) {
            $scope.correctAnswer = (data === true);
            $scope.working = false;
        }).error(function (data, status, headers, config) {
            $scope.title = "Oops... something went wrong";
            $scope.working = false;
        });
    };
};
```

> **NOTE**
>
> This function sends the answer selected by the user to the **Trivia** Web API and stores the result –i.e. if the answer is correct or not– in the **$scope** object.
>
> The **nextQuestion** and **sendAnswer** functions from above use the AngularJS **$http** object to abstract the communication with the Web API via the XMLHttpRequest JavaScript object from the browser. AngularJS supports another service that brings a higher level of abstraction to perform CRUD operations against a resource through RESTful APIs. The AngularJS **$resource** object has action methods which provide high-level behaviors without the need to interact with the **$http** object. Consider using the **$resource** object in scenarios that requires the CRUD model (fore information, see the $resource documentation).

9. The next step is to create the AngularJS template that defines the view for the quiz. To do this, open the **Index.cshtml** file inside the **Views | Home** folder and replace the content with the following code.

   (Code Snippet - *AspNetWebApiSpa - Ex2 - GeekQuizView*)

```
@{
    ViewBag.Title = "Play";
}

<div id="bodyContainer" ng-app="QuizApp">
    <section id="content">
        <div class="container" >
            <div class="row">
                <div class="flip-container text-center col-md-12" ng-controller="QuizCtrl" ng-
init="nextQuestion()">
                    <div class="back" ng-class="{flip: answered, correct: correctAnswer,
incorrect:!correctAnswer}">
                        <p class="lead">{{answer()}}</p>
                        <p>
                            <button class="btn btn-info btn-lg next option" ng-click="nextQuestion()"
ng-disabled="working">Next Question</button>
                        </p>
                    </div>
                    <div class="front" ng-class="{flip: answered}">
                        <p class="lead">{{title}}</p>
                        <div class="row text-center">
                            <button class="btn btn-info btn-lg option" ng-repeat="option in options"
ng-click="sendAnswer(option)" ng-disabled="working">{{option.title}}</button>
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </section>
</div>

@section scripts {
    @Scripts.Render("~/Scripts/angular.js")
    @Scripts.Render("~/Scripts/app/quiz-controller.js")
}
```

NOTE

The AngularJS template is a declarative specification that uses information from the model and the controller to transform static markup into the dynamic view that the user sees in the browser. The following are examples of AngularJS elements and element attributes that can be used in a template:

- The **ng-app** directive tells AngularJS the DOM element that represents the root element of the application.
- The **ng-controller** directive attaches a controller to the DOM at the point where the directive is declared.
- The curly brace notation **{{ }}** denotes bindings to the scope properties defined in the controller.
- The **ng-click** directive is used to invoke the functions defined in the scope in response to user clicks.

10. Open the **Site.css** file inside the **Content** folder and add the following highlighted styles at the end of the file to provide a look and feel for the quiz view.

    (Code Snippet - *AspNetWebApiSpa - Ex2 - GeekQuizStyles*)

```
.validation-summary-valid {
     display: none;
}

/* Geek Quiz styles */
.flip-container .back,
.flip-container .front {
     border: 5px solid #00bcf2;
     padding-bottom: 30px;
     padding-top: 30px;
}

#content {
    position:relative;
    background:#fff;
    padding:50px 0 0 0;
}

.option {
     width:140px;
     margin: 5px;
}

div.correct p {
     color: green;
}

div.incorrect p {
     color: red;
}

.btn {
     border-radius: 0;
}

.flip-container div.front, .flip-container div.back.flip {
    display: block;
}

.flip-container div.front.flip, .flip-container div.back {
    display: none;
}
```

Task 2 – Running the Solution

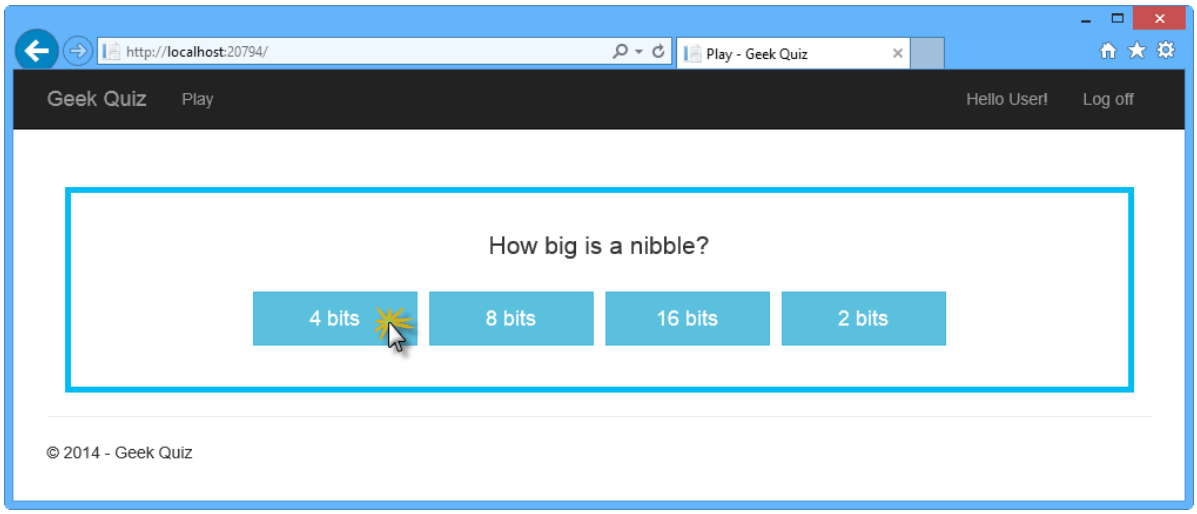
In this task you will execute the solution using the new user interface you built with AngularJS to answer some of the quiz questions.

1. Press **F5** to run the solution.

2. Register a new user account. To do this, follow the registration steps described in Exercise 1, Task 3.

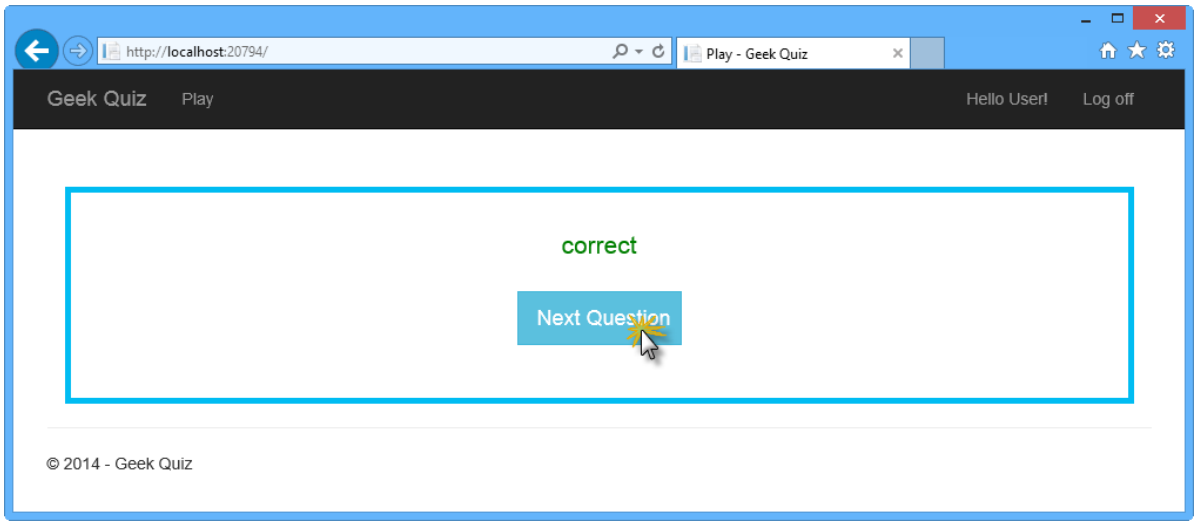
> **NOTE**
>
> If you are using the solution from the previous exercise, you can log in with the user account you created before.

3. The **Home** page should appear, showing the first question of the quiz. Answer the question by clicking one of the options. This will trigger the **sendAnswer** function defined earlier, which sends the selected option to the **Trivia** Web API.



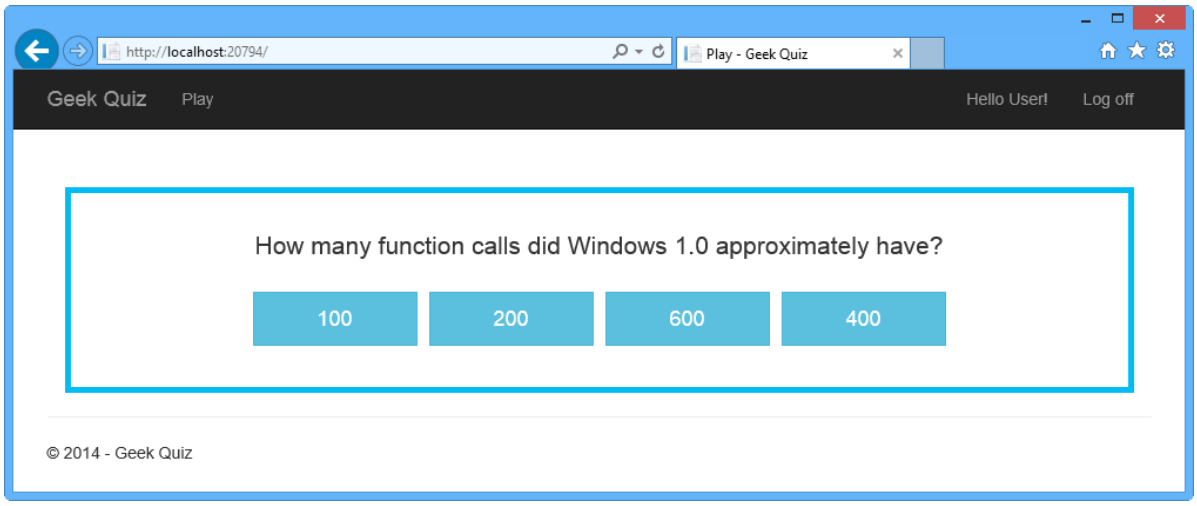
Answering a question

4. After clicking one of the buttons, the answer should appear. Click **Next Question** to show the following question. This will trigger the **nextQuestion** function defined in the controller.



Requesting the next question

5. The next question should appear. Continue answering questions as many times as you want. After completing all the questions you should return to the first question.



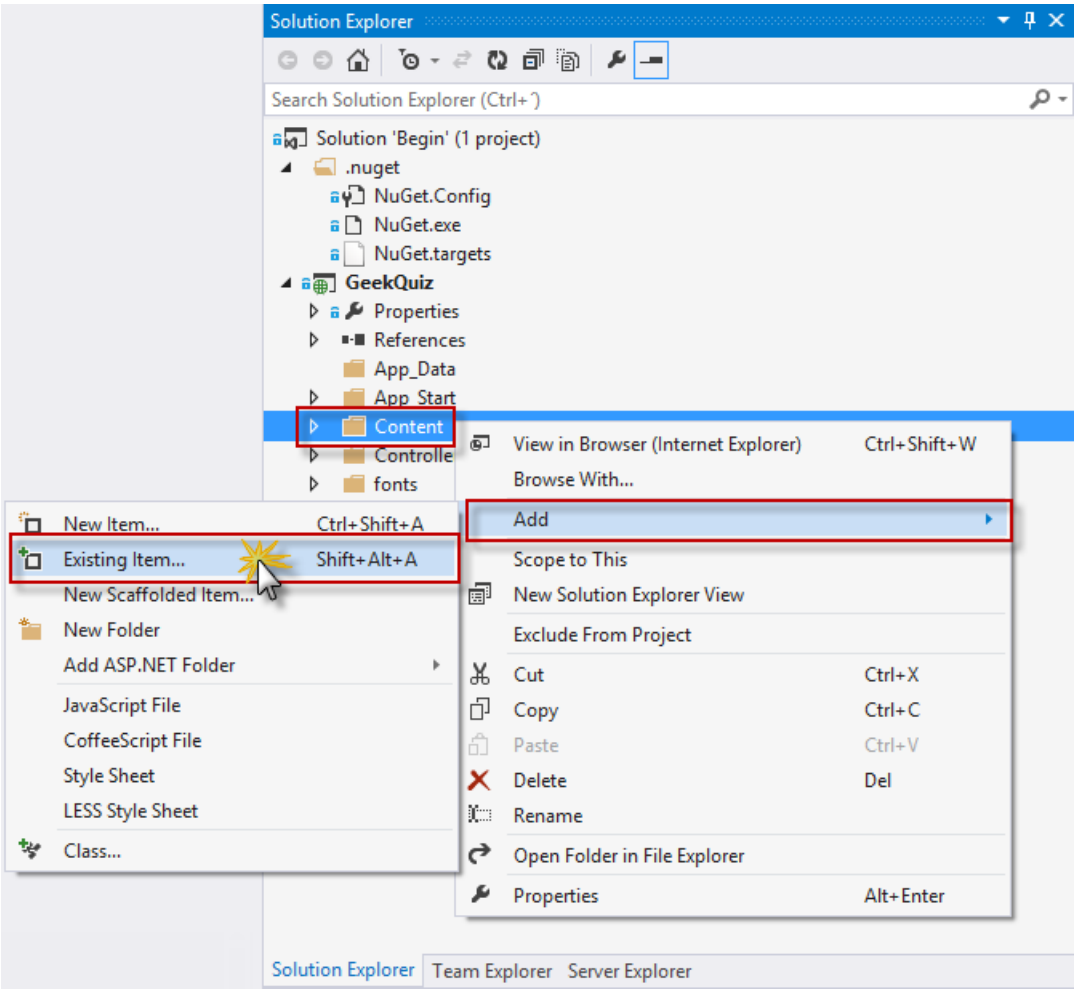
Next question

6. Go back to Visual Studio and press **SHIFT + F5** to stop debugging.

Task 3 – Creating a Flip Animation Using CSS3

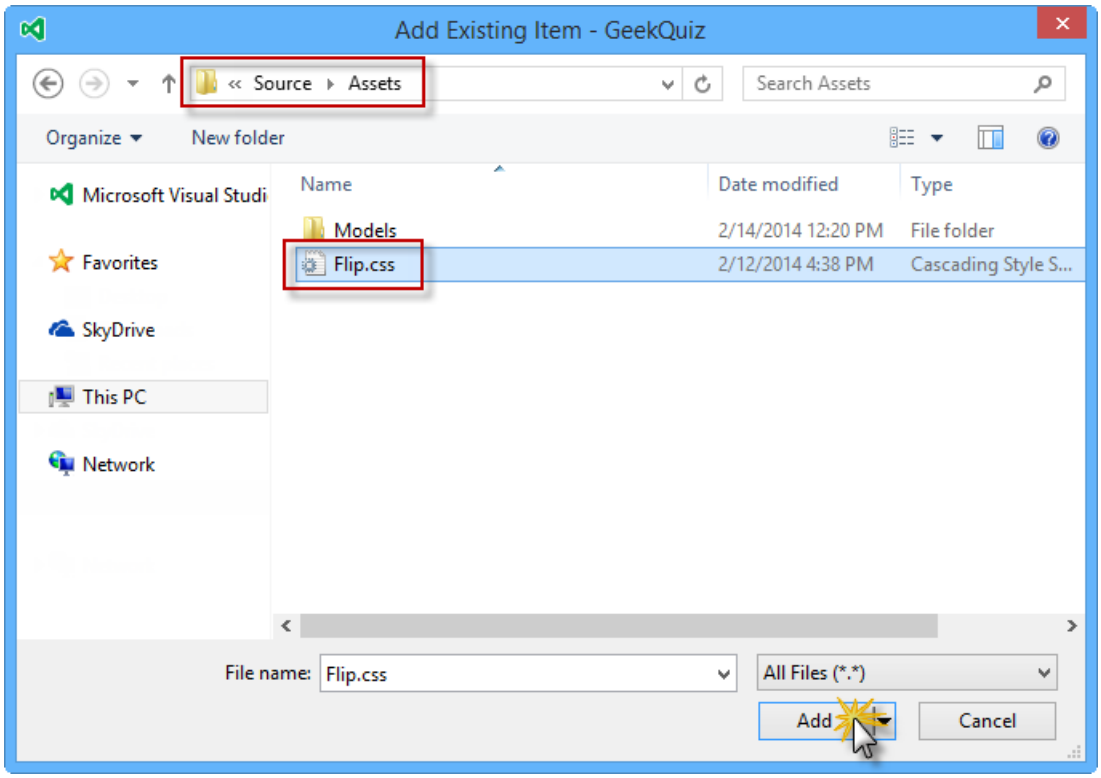
In this task you will use CSS3 properties to perform rich animations by adding a flip effect when a question is answered and when the next question is retrieved.

1. In **Solution Explorer**, right-click the **Content** folder of the **GeekQuiz** project and select **Add | Existing Item...**.



Adding an existing item to the Content folder

2. In the **Add Existing Item** dialog box, navigate to the **Source/Assets** folder and select **Flip.css**. Click **Add**.



Adding the Flip.css file from Assets

3. Open the **Flip.css** file you just added and inspect its content.

4. Locate the **flip transformation** comment. The styles below that comment use the CSS **perspective** and **rotateY** transformations to generate a "card flip" effect.

```
/* flip transformation */
.flip-container div.front {
    -moz-transform: perspective(2000px) rotateY(0deg);
    -webkit-transform: perspective(2000px) rotateY(0deg);
    -o-transform: perspective(2000px) rotateY(0deg);
    transform: perspective(2000px) rotateY(0deg);
}

    .flip-container div.front.flip {
        -moz-transform: perspective(2000px) rotateY(179.9deg);
        -webkit-transform: perspective(2000px) rotateY(179.9deg);
        -o-transform: perspective(2000px) rotateY(179.9deg);
        transform: perspective(2000px) rotateY(179.9deg);
    }

.flip-container div.back {
    -moz-transform: perspective(2000px) rotateY(-180deg);
    -webkit-transform: perspective(2000px) rotateY(-180deg);
    -o-transform: perspective(2000px) rotateY(-180deg);
    transform: perspective(2000px) rotateY(-180deg);
}

    .flip-container div.back.flip {
        -moz-transform: perspective(2000px) rotateY(0deg);
        -webkit-transform: perspective(2000px) rotateY(0deg);
        -ms-transform: perspective(2000px) rotateY(0);
        -o-transform: perspective(2000px) rotateY(0);
        transform: perspective(2000px) rotateY(0);
    }
```

5. Locate the **hide back of pane during flip** comment. The style below that comment hides the back-side of the faces when they are facing away from the viewer by setting the **backface-visibility** CSS property to

hidden.

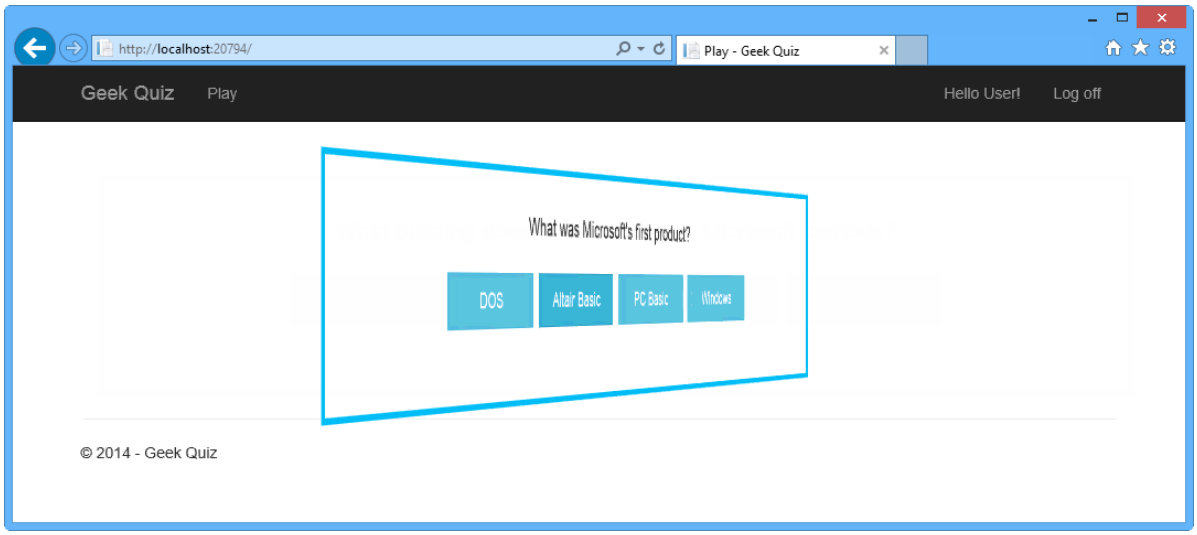
```
/* hide back of pane during flip */
.front, .back {
    -moz-backface-visibility: hidden;
    -webkit-backface-visibility: hidden;
    backface-visibility: hidden;
}
```

6. Open the **BundleConfig.cs** file inside the **App_Start** folder and add the reference to the **Flip.css** file in the **"~/Content/css"** style bundle

```
bundles.Add(new StyleBundle("~/Content/css").Include(
    "~/Content/bootstrap.css",
    "~/Content/site.css",
    "~/Content/Flip.css"));
```

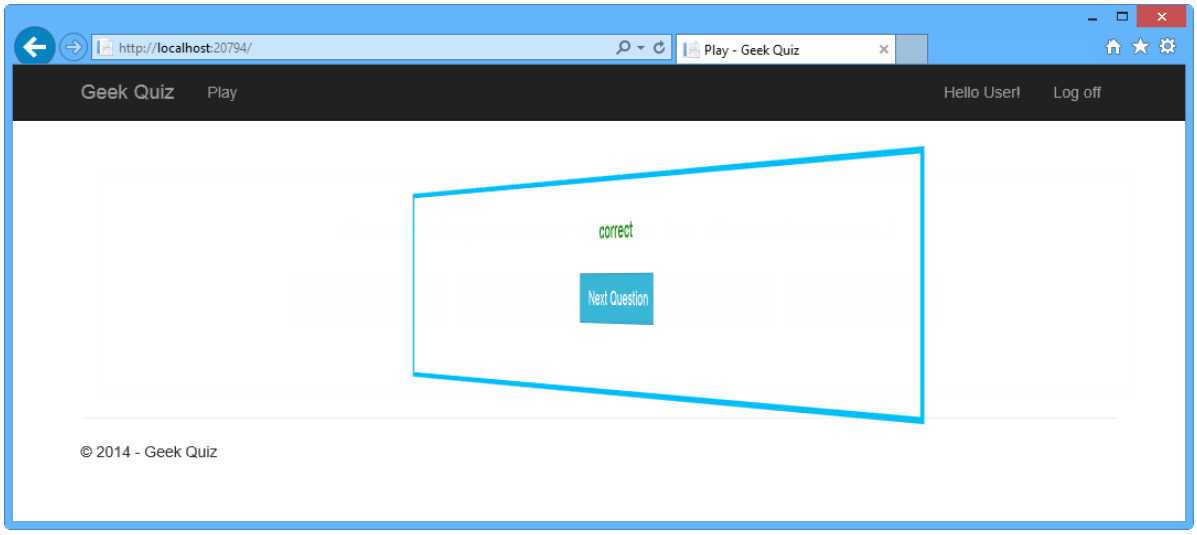
7. Press **F5** to run the solution and log in with your credentials.

8. Answer a question by clicking one of the options. Notice the flip effect when transitioning between views.



Answering a question with the flip effect

9. Click **Next Question** to retrieve the following question. The flip effect should appear again.



Retrieving the following question with the flip effect

Summary

By completing this hands-on lab you have learned how to:

- Create an ASP.NET Web API controller using ASP.NET Scaffolding
- Implement a Web API Get action to retrieve the next quiz question
- Implement a Web API Post action to store the quiz answers
- Install AngularJS from the Visual Studio Package Manager Console
- Implement AngularJS templates and controllers
- Use CSS3 transitions to perform animation effects

Web API Routing

2/20/2020 • 2 minutes to read • Edit Online

How ASP.NET Web API routes HTTP requests to controllers and actions.

- Routing in Web API
- Routing and Action Selection in Web API
- Attribute Routing in Web API 2
- Create a REST API with Attribute Routing in Web API 2

Routing in ASP.NET Web API

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson

This article describes how ASP.NET Web API routes HTTP requests to controllers.

> **NOTE**
>
> If you are familiar with ASP.NET MVC, Web API routing is very similar to MVC routing. The main difference is that Web API uses the HTTP verb, not the URI path, to select the action. You can also use MVC-style routing in Web API. This article does not assume any knowledge of ASP.NET MVC.

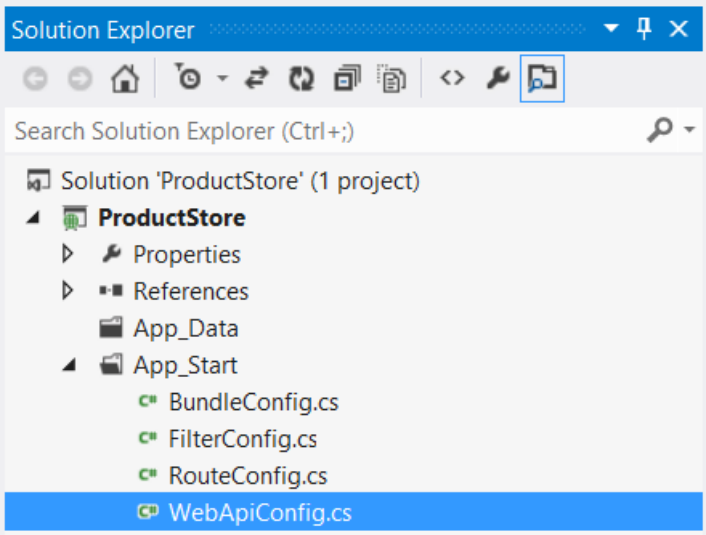
Routing Tables

In ASP.NET Web API, a *controller* is a class that handles HTTP requests. The public methods of the controller are called *action methods* or simply *actions*. When the Web API framework receives a request, it routes the request to an action.

To determine which action to invoke, the framework uses a *routing table*. The Visual Studio project template for Web API creates a default route:

```
routes.MapHttpRoute(
    name: "API Default",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

This route is defined in the *WebApiConfig.cs* file, which is placed in the *App_Start* directory:



For more information about the `WebApiConfig` class, see Configuring ASP.NET Web API.

If you self-host Web API, you must set the routing table directly on the `HttpSelfHostConfiguration` object. For more information, see Self-Host a Web API.

Each entry in the routing table contains a *route template*. The default route template for Web API is "api/{controller}/{id}". In this template, "api" is a literal path segment, and {controller} and {id} are placeholder

variables.

When the Web API framework receives an HTTP request, it tries to match the URI against one of the route templates in the routing table. If no route matches, the client receives a 404 error. For example, the following URIs match the default route:

- /api/contacts
- /api/contacts/1
- /api/products/gizmo1

However, the following URI does not match, because it lacks the "api" segment:

- /contacts/1

> **NOTE**
> The reason for using "api" in the route is to avoid collisions with ASP.NET MVC routing. That way, you can have "/contacts" go to an MVC controller, and "/api/contacts" go to a Web API controller. Of course, if you don't like this convention, you can change the default route table.

Once a matching route is found, Web API selects the controller and the action:

- To find the controller, Web API adds "Controller" to the value of the *{controller}* variable.
- To find the action, Web API looks at the HTTP verb, and then looks for an action whose name begins with that HTTP verb name. For example, with a GET request, Web API looks for an action prefixed with "Get", such as "GetContact" or "GetAllContacts". This convention applies only to GET, POST, PUT, DELETE, HEAD, OPTIONS, and PATCH verbs. You can enable other HTTP verbs by using attributes on your controller. We'll see an example of that later.
- Other placeholder variables in the route template, such as *{id},* are mapped to action parameters.

Let's look at an example. Suppose that you define the following controller:

```
public class ProductsController : ApiController
{
    public IEnumerable<Product> GetAllProducts() { }
    public Product GetProductById(int id) { }
    public HttpResponseMessage DeleteProduct(int id){ }
}
```

Here are some possible HTTP requests, along with the action that gets invoked for each:

HTTP VERB	URI PATH	ACTION	PARAMETER
GET	api/products	GetAllProducts	*(none)*
GET	api/products/4	GetProductById	4
DELETE	api/products/4	DeleteProduct	4
POST	api/products	*(no match)*	

Notice that the *{id}* segment of the URI, if present, is mapped to the *id* parameter of the action. In this example, the controller defines two GET methods, one with an *id* parameter and one with no parameters.

Also, note that the POST request will fail, because the controller does not define a "Post..." method.

Routing Variations

The previous section described the basic routing mechanism for ASP.NET Web API. This section describes some variations.

HTTP verbs

Instead of using the naming convention for HTTP verbs, you can explicitly specify the HTTP verb for an action by decorating the action method with one of the following attributes:

- `[HttpGet]`
- `[HttpPut]`
- `[HttpPost]`
- `[HttpDelete]`
- `[HttpHead]`
- `[HttpOptions]`
- `[HttpPatch]`

In the following example, the `FindProduct` method is mapped to GET requests:

```
public class ProductsController : ApiController
{
    [HttpGet]
    public Product FindProduct(id) {}
}
```

To allow multiple HTTP verbs for an action, or to allow HTTP verbs other than GET, PUT, POST, DELETE, HEAD, OPTIONS, and PATCH, use the `[AcceptVerbs]` attribute, which takes a list of HTTP verbs.

```
public class ProductsController : ApiController
{
    [AcceptVerbs("GET", "HEAD")]
    public Product FindProduct(id) { }

    // WebDAV method
    [AcceptVerbs("MKCOL")]
    public void MakeCollection() { }
}
```

Routing by Action Name

With the default routing template, Web API uses the HTTP verb to select the action. However, you can also create a route where the action name is included in the URI:

```
routes.MapHttpRoute(
    name: "ActionApi",
    routeTemplate: "api/{controller}/{action}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

In this route template, the *{action}* parameter names the action method on the controller. With this style of routing, use attributes to specify the allowed HTTP verbs. For example, suppose your controller has the following method:

```
public class ProductsController : ApiController
{
    [HttpGet]
    public string Details(int id);
}
```

In this case, a GET request for "api/products/details/1" would map to the `Details` method. This style of routing is similar to ASP.NET MVC, and may be appropriate for an RPC-style API.

You can override the action name by using the `[ActionName]` attribute. In the following example, there are two actions that map to "api/products/thumbnail/*id*. One supports GET and the other supports POST:

```
public class ProductsController : ApiController
{
    [HttpGet]
    [ActionName("Thumbnail")]
    public HttpResponseMessage GetThumbnailImage(int id);

    [HttpPost]
    [ActionName("Thumbnail")]
    public void AddThumbnailImage(int id);
}
```

Non-Actions

To prevent a method from getting invoked as an action, use the `[NonAction]` attribute. This signals to the framework that the method is not an action, even if it would otherwise match the routing rules.

```
// Not an action method.
[NonAction]
public string GetPrivateData() { ... }
```

Further Reading

This topic provided a high-level view of routing. For more detail, see Routing and Action Selection, which describes exactly how the framework matches a URI to a route, selects a controller, and then selects the action to invoke.

Routing and Action Selection in ASP.NET Web API

2/20/2020 • 9 minutes to read • Edit Online

by Mike Wasson

This article describes how ASP.NET Web API routes an HTTP request to a particular action on a controller.

> **NOTE**
>
> For a high-level overview of routing, see Routing in ASP.NET Web API.

This article looks at the details of the routing process. If you create a Web API project and find that some requests don't get routed the way you expect, hopefully this article will help.

Routing has three main phases:

1. Matching the URI to a route template.
2. Selecting a controller.
3. Selecting an action.

You can replace some parts of the process with your own custom behaviors. In this article, I describe the default behavior. At the end, I note the places where you can customize the behavior.

Route Templates

A route template looks similar to a URI path, but it can have placeholder values, indicated with curly braces:

```
"api/{controller}/public/{category}/{id}"
```

When you create a route, you can provide default values for some or all of the placeholders:

```
defaults: new { category = "all" }
```

You can also provide constraints, which restrict how a URI segment can match a placeholder:

```
constraints: new { id = @"\d+" }   // Only matches if "id" is one or more digits.
```

The framework tries to match the segments in the URI path to the template. Literals in the template must match exactly. A placeholder matches any value, unless you specify constraints. The framework does not match other parts of the URI, such as the host name or the query parameters. The framework selects the first route in the route table that matches the URI.

There are two special placeholders: "{controller}" and "{action}".

- "{controller}" provides the name of the controller.
- "{action}" provides the name of the action. In Web API, the usual convention is to omit "{action}".

Defaults

If you provide defaults, the route will match a URI that is missing those segments. For example:

```
routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{category}",
    defaults: new { category = "all" }
);
```

The URIs `http://localhost/api/products/all` and `http://localhost/api/products` match the preceding route. In the latter URI, the missing `{category}` segment is assigned the default value `all`.

Route Dictionary

If the framework finds a match for a URI, it creates a dictionary that contains the value for each placeholder. The keys are the placeholder names, not including the curly braces. The values are taken from the URI path or from the defaults. The dictionary is stored in the **IHttpRouteData** object.

During this route-matching phase, the special "{controller}" and "{action}" placeholders are treated just like the other placeholders. They are simply stored in the dictionary with the other values.

A default can have the special value **RouteParameter.Optional**. If a placeholder gets assigned this value, the value is not added to the route dictionary. For example:

```
routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{category}/{id}",
    defaults: new { category = "all", id = RouteParameter.Optional }
);
```

For the URI path "api/products", the route dictionary will contain:

- controller: "products"
- category: "all"

For "api/products/toys/123", however, the route dictionary will contain:

- controller: "products"
- category: "toys"
- id: "123"

The defaults can also include a value that does not appear anywhere in the route template. If the route matches, that value is stored in the dictionary. For example:

```
routes.MapHttpRoute(
    name: "Root",
    routeTemplate: "api/root/{id}",
    defaults: new { controller = "customers", id = RouteParameter.Optional }
);
```

If the URI path is "api/root/8", the dictionary will contain two values:

- controller: "customers"
- id: "8"

Selecting a Controller

Controller selection is handled by the **IHttpControllerSelector.SelectController** method. This method takes an **HttpRequestMessage** instance and returns an **HttpControllerDescriptor**. The default implementation is provided by the **DefaultHttpControllerSelector** class. This class uses a straightforward algorithm:

1. Look in the route dictionary for the key "controller".
2. Take the value for this key and append the string "Controller" to get the controller type name.
3. Look for a Web API controller with this type name.

For example, if the route dictionary contains the key-value pair "controller" = "products", then the controller type is "ProductsController". If there is no matching type, or multiple matches, the framework returns an error to the client.

For step 3, **DefaultHttpControllerSelector** uses the **IHttpControllerTypeResolver** interface to get the list of Web API controller types. The default implementation of **IHttpControllerTypeResolver** returns all public classes that (a) implement **IHttpController**, (b) are not abstract, and (c) have a name that ends in "Controller".

Action Selection

After selecting the controller, the framework selects the action by calling the **IHttpActionSelector.SelectAction** method. This method takes an **HttpControllerContext** and returns an **HttpActionDescriptor**.

The default implementation is provided by the **ApiControllerActionSelector** class. To select an action, it looks at the following:

- The HTTP method of the request.
- The "{action}" placeholder in the route template, if present.
- The parameters of the actions on the controller.

Before looking at the selection algorithm, we need to understand some things about controller actions.

Which methods on the controller are considered "actions"? When selecting an action, the framework only looks at public instance methods on the controller. Also, it excludes "special name" methods (constructors, events, operator overloads, and so forth), and methods inherited from the **ApiController** class.

HTTP Methods. The framework only chooses actions that match the HTTP method of the request, determined as follows:

1. You can specify the HTTP method with an attribute: **AcceptVerbs**, **HttpDelete**, **HttpGet**, **HttpHead**, **HttpOptions**, **HttpPatch**, **HttpPost**, or **HttpPut**.
2. Otherwise, if the name of the controller method starts with "Get", "Post", "Put", "Delete", "Head", "Options", or "Patch", then by convention the action supports that HTTP method.
3. If none of the above, the method supports POST.

Parameter Bindings. A parameter binding is how Web API creates a value for a parameter. Here is the default rule for parameter binding:

- Simple types are taken from the URI.
- Complex types are taken from the request body.

Simple types include all of the .NET Framework primitive types, plus **DateTime**, **Decimal**, **Guid**, **String**, and **TimeSpan**. For each action, at most one parameter can read the request body.

> **NOTE**
>
> It is possible to override the default binding rules. See WebAPI Parameter binding under the hood.

With that background, here is the action selection algorithm.

1. Create a list of all actions on the controller that match the HTTP request method.

2. If the route dictionary has an "action" entry, remove actions whose name does not match this value.

3. Try to match action parameters to the URI, as follows:
    a. For each action, get a list of the parameters that are a simple type, where the binding gets the parameter from the URI. Exclude optional parameters.
    b. From this list, try to find a match for each parameter name, either in the route dictionary or in the URI query string. Matches are case insensitive and do not depend on the parameter order.
    c. Select an action where every parameter in the list has a match in the URI.
    d. If more that one action meets these criteria, pick the one with the most parameter matches.
4. Ignore actions with the **[NonAction]** attribute.

Step #3 is probably the most confusing. The basic idea is that a parameter can get its value either from the URI, from the request body, or from a custom binding. For parameters that come from the URI, we want to ensure that the URI actually contains a value for that parameter, either in the path (via the route dictionary) or in the query string.

For example, consider the following action:

```
public void Get(int id)
```

The *id* parameter binds to the URI. Therefore, this action can only match a URI that contains a value for "id", either in the route dictionary or in the query string.

Optional parameters are an exception, because they are optional. For an optional parameter, it's OK if the binding can't get the value from the URI.

Complex types are an exception for a different reason. A complex type can only bind to the URI through a custom binding. But in that case, the framework cannot know in advance whether the parameter would bind to a particular URI. To find out, it would need to invoke the binding. The goal of the selection algorithm is to select an action from the static description, before invoking any bindings. Therefore, complex types are excluded from the matching algorithm.

After the action is selected, all parameter bindings are invoked.

Summary:

- The action must match the HTTP method of the request.
- The action name must match the "action" entry in the route dictionary, if present.
- For every parameter of the action, if the parameter is taken from the URI, then the parameter name must be found either in the route dictionary or in the URI query string. (Optional parameters and parameters with complex types are excluded.)
- Try to match the most number of parameters. The best match might be a method with no parameters.

Extended Example

Routes:

```
routes.MapHttpRoute(
    name: "ApiRoot",
    routeTemplate: "api/root/{id}",
    defaults: new { controller = "products", id = RouteParameter.Optional }
);
routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

Controller:

```
public class ProductsController : ApiController
{
    public IEnumerable<Product> GetAll() {}
    public Product GetById(int id, double version = 1.0) {}
    [HttpGet]
    public void FindProductsByName(string name) {}
    public void Post(Product value) {}
    public void Put(int id, Product value) {}
}
```

HTTP request:

```
GET http://localhost:34701/api/products/1?version=1.5&details=1
```

Route Matching

The URI matches the route named "DefaultApi". The route dictionary contains the following entries:

- controller: "products"
- id: "1"

The route dictionary does not contain the query string parameters, "version" and "details", but these will still be considered during action selection.

Controller Selection

From the "controller" entry in the route dictionary, the controller type is `ProductsController`.

Action Selection

The HTTP request is a GET request. The controller actions that support GET are `GetAll`, `GetById`, and `FindProductsByName`. The route dictionary does not contain an entry for "action", so we don't need to match the action name.

Next, we try to match parameter names for the actions, looking only at the GET actions.

ACTION	PARAMETERS TO MATCH
`GetAll`	none
`GetById`	"id"
`FindProductsByName`	"name"

Notice that the *version* parameter of `GetById` is not considered, because it is an optional parameter.

The `GetAll` method matches trivially. The `GetById` method also matches, because the route dictionary contains "id". The `FindProductsByName` method does not match.

The `GetById` method wins, because it matches one parameter, versus no parameters for `GetAll`. The method is invoked with the following parameter values:

- *id* = 1
- *version* = 1.5

Notice that even though *version* was not used in the selection algorithm, the value of the parameter comes from the URI query string.

Extension Points

Web API provides extension points for some parts of the routing process.

INTERFACE	DESCRIPTION
IHttpControllerSelector	Selects the controller.
IHttpControllerTypeResolver	Gets the list of controller types. The **DefaultHttpControllerSelector** chooses the controller type from this list.
IAssembliesResolver	Gets the list of project assemblies. The **IHttpControllerTypeResolver** interface uses this list to find the controller types.
IHttpControllerActivator	Creates new controller instances.
IHttpActionSelector	Selects the action.
IHttpActionInvoker	Invokes the action.

To provide your own implementation for any of these interfaces, use the **Services** collection on the **HttpConfiguration** object:

```
var config = GlobalConfiguration.Configuration;
config.Services.Replace(typeof(IHttpControllerSelector), new MyControllerSelector(config));
```

Attribute Routing in ASP.NET Web API 2

4/6/2020 • 10 minutes to read • Edit Online

by Mike Wasson

Routing is how Web API matches a URI to an action. Web API 2 supports a new type of routing, called *attribute routing*. As the name implies, attribute routing uses attributes to define routes. Attribute routing gives you more control over the URIs in your web API. For example, you can easily create URIs that describe hierarchies of resources.

The earlier style of routing, called convention-based routing, is still fully supported. In fact, you can combine both techniques in the same project.

This topic shows how to enable attribute routing and describes the various options for attribute routing. For an end-to-end tutorial that uses attribute routing, see Create a REST API with Attribute Routing in Web API 2.

Prerequisites

Visual Studio 2017 Community, Professional, or Enterprise edition

Alternatively, use NuGet Package Manager to install the necessary packages. From the **Tools** menu in Visual Studio, select **NuGet Package Manager**, then select **Package Manager Console**. Enter the following command in the Package Manager Console window:

```
Install-Package Microsoft.AspNet.WebApi.WebHost
```

Why Attribute Routing?

The first release of Web API used *convention-based* routing. In that type of routing, you define one or more route templates, which are basically parameterized strings. When the framework receives a request, it matches the URI against the route template. For more information about convention-based routing, see Routing in ASP.NET Web API.

One advantage of convention-based routing is that templates are defined in a single place, and the routing rules are applied consistently across all controllers. Unfortunately, convention-based routing makes it hard to support certain URI patterns that are common in RESTful APIs. For example, resources often contain child resources: Customers have orders, movies have actors, books have authors, and so forth. It's natural to create URIs that reflect these relations:

```
/customers/1/orders
```

This type of URI is difficult to create using convention-based routing. Although it can be done, the results don't scale well if you have many controllers or resource types.

With attribute routing, it's trivial to define a route for this URI. You simply add an attribute to the controller action:

```
[Route("customers/{customerId}/orders")]
public IEnumerable<Order> GetOrdersByCustomer(int customerId) { ... }
```

Here are some other patterns that attribute routing makes easy.

API versioning

In this example, "/api/v1/products" would be routed to a different controller than "/api/v2/products".

`/api/v1/products` `/api/v2/products`

Overloaded URI segments

In this example, "1" is an order number, but "pending" maps to a collection.

`/orders/1` `/orders/pending`

Multiple parameter types

In this example, "1" is an order number, but "2013/06/16" specifies a date.

`/orders/1` `/orders/2013/06/16`

Enabling Attribute Routing

To enable attribute routing, call **MapHttpAttributeRoutes** during configuration. This extension method is defined in the **System.Web.Http.HttpConfigurationExtensions** class.

```
using System.Web.Http;

namespace WebApplication
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Web API routes
            config.MapHttpAttributeRoutes();

            // Other Web API configuration not shown.
        }
    }
}
```

Attribute routing can be combined with convention-based routing. To define convention-based routes, call the **MapHttpRoute** method.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // Attribute routing.
        config.MapHttpAttributeRoutes();

        // Convention-based routing.
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

For more information about configuring Web API, see Configuring ASP.NET Web API 2.

Note: Migrating From Web API 1

Prior to Web API 2, the Web API project templates generated code like this:

```
protected void Application_Start()
{
    // WARNING - Not compatible with attribute routing.
    WebApiConfig.Register(GlobalConfiguration.Configuration);
}
```

If attribute routing is enabled, this code will throw an exception. If you upgrade an existing Web API project to use attribute routing, make sure to update this configuration code to the following:

```
protected void Application_Start()
{
    // Pass a delegate to the Configure method.
    GlobalConfiguration.Configure(WebApiConfig.Register);
}
```

> **NOTE**
>
> For more information, see Configuring Web API with ASP.NET Hosting.

Adding Route Attributes

Here is an example of a route defined using an attribute:

```
public class OrdersController : ApiController
{
    [Route("customers/{customerId}/orders")]
    [HttpGet]
    public IEnumerable<Order> FindOrdersByCustomer(int customerId) { ... }
}
```

The string "customers/{customerId}/orders" is the URI template for the route. Web API tries to match the request URI to the template. In this example, "customers" and "orders" are literal segments, and "{customerId}" is a variable parameter. The following URIs would match this template:

- `http://localhost/customers/1/orders`
- `http://localhost/customers/bob/orders`
- `http://localhost/customers/1234-5678/orders`

You can restrict the matching by using constraints, described later in this topic.

Notice that the "{customerId}" parameter in the route template matches the name of the *customerId* parameter in the method. When Web API invokes the controller action, it tries to bind the route parameters. For example, if the URI is `http://example.com/customers/1/orders`, Web API tries to bind the value "1" to the *customerId* parameter in the action.

A URI template can have several parameters:

```
[Route("customers/{customerId}/orders/{orderId}")]
public Order GetOrderByCustomer(int customerId, int orderId) { ... }
```

Any controller methods that do not have a route attribute use convention-based routing. That way, you can combine both types of routing in the same project.

HTTP Methods

Web API also selects actions based on the HTTP method of the request (GET, POST, etc). By default, Web API looks for a case-insensitive match with the start of the controller method name. For example, a controller method named `PutCustomers` matches an HTTP PUT request.

You can override this convention by decorating the method with any the following attributes:

- **[HttpDelete]**
- **[HttpGet]**
- **[HttpHead]**
- **[HttpOptions]**
- **[HttpPatch]**
- **[HttpPost]**
- **[HttpPut]**

The following example maps the CreateBook method to HTTP POST requests.

```
[Route("api/books")]
[HttpPost]
public HttpResponseMessage CreateBook(Book book) { ... }
```

For all other HTTP methods, including non-standard methods, use the **AcceptVerbs** attribute, which takes a list of HTTP methods.

```
// WebDAV method
[Route("api/books")]
[AcceptVerbs("MKCOL")]
public void MakeCollection() { }
```

Route Prefixes

Often, the routes in a controller all start with the same prefix. For example:

```
public class BooksController : ApiController
{
    [Route("api/books")]
    public IEnumerable<Book> GetBooks() { ... }

    [Route("api/books/{id:int}")]
    public Book GetBook(int id) { ... }

    [Route("api/books")]
    [HttpPost]
    public HttpResponseMessage CreateBook(Book book) { ... }
}
```

You can set a common prefix for an entire controller by using the **[RoutePrefix]** attribute:

```
[RoutePrefix("api/books")]
public class BooksController : ApiController
{
    // GET api/books
    [Route("")]
    public IEnumerable<Book> Get() { ... }

    // GET api/books/5
    [Route("{id:int}")]
    public Book Get(int id) { ... }

    // POST api/books
    [Route("")]
    public HttpResponseMessage Post(Book book) { ... }
}
```

Use a tilde (~) on the method attribute to override the route prefix:

```
[RoutePrefix("api/books")]
public class BooksController : ApiController
{
    // GET /api/authors/1/books
    [Route("~/api/authors/{authorId:int}/books")]
    public IEnumerable<Book> GetByAuthor(int authorId) { ... }

    // ...
}
```

The route prefix can include parameters:

```
[RoutePrefix("customers/{customerId}")]
public class OrdersController : ApiController
{
    // GET customers/1/orders
    [Route("orders")]
    public IEnumerable<Order> Get(int customerId) { ... }
}
```

Route Constraints

Route constraints let you restrict how the parameters in the route template are matched. The general syntax is "{parameter:constraint}". For example:

```
[Route("users/{id:int}")]
public User GetUserById(int id) { ... }

[Route("users/{name}")]
public User GetUserByName(string name) { ... }
```

Here, the first route will only be selected if the "id" segment of the URI is an integer. Otherwise, the second route will be chosen.

The following table lists the constraints that are supported.

CONSTRAINT	DESCRIPTION	EXAMPLE
alpha	Matches uppercase or lowercase Latin alphabet characters (a-z, A-Z)	{x:alpha}

CONSTRAINT	DESCRIPTION	EXAMPLE
bool	Matches a Boolean value.	{x:bool}
datetime	Matches a **DateTime** value.	{x:datetime}
decimal	Matches a decimal value.	{x:decimal}
double	Matches a 64-bit floating-point value.	{x:double}
float	Matches a 32-bit floating-point value.	{x:float}
guid	Matches a GUID value.	{x:guid}
int	Matches a 32-bit integer value.	{x:int}
length	Matches a string with the specified length or within a specified range of lengths.	{x:length(6)} {x:length(1,20)}
long	Matches a 64-bit integer value.	{x:long}
max	Matches an integer with a maximum value.	{x:max(10)}
maxlength	Matches a string with a maximum length.	{x:maxlength(10)}
min	Matches an integer with a minimum value.	{x:min(10)}
minlength	Matches a string with a minimum length.	{x:minlength(10)}
range	Matches an integer within a range of values.	{x:range(10,50)}
regex	Matches a regular expression.	{x:regex(^\d{3}-\d{3}-\d{4}$)}

Notice that some of the constraints, such as "min", take arguments in parentheses. You can apply multiple constraints to a parameter, separated by a colon.

```
[Route("users/{id:int:min(1)}")]
public User GetUserById(int id) { ... }
```

Custom Route Constraints

You can create custom route constraints by implementing the **IHttpRouteConstraint** interface. For example, the following constraint restricts a parameter to a non-zero integer value.

```
public class NonZeroConstraint : IHttpRouteConstraint
{
    public bool Match(HttpRequestMessage request, IHttpRoute route, string parameterName,
        IDictionary<string, object> values, HttpRouteDirection routeDirection)
    {
        object value;
        if (values.TryGetValue(parameterName, out value) && value != null)
        {
            long longValue;
            if (value is long)
            {
                longValue = (long)value;
                return longValue != 0;
            }

            string valueString = Convert.ToString(value, CultureInfo.InvariantCulture);
            if (Int64.TryParse(valueString, NumberStyles.Integer,
                CultureInfo.InvariantCulture, out longValue))
            {
                return longValue != 0;
            }
        }
        return false;
    }
}
```

The following code shows how to register the constraint:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        var constraintResolver = new DefaultInlineConstraintResolver();
        constraintResolver.ConstraintMap.Add("nonzero", typeof(NonZeroConstraint));

        config.MapHttpAttributeRoutes(constraintResolver);
    }
}
```

Now you can apply the constraint in your routes:

```
[Route("{id:nonzero}")]
public HttpResponseMessage GetNonZero(int id) { ... }
```

You can also replace the entire **DefaultInlineConstraintResolver** class by implementing the **IInlineConstraintResolver** interface. Doing so will replace all of the built-in constraints, unless your implementation of **IInlineConstraintResolver** specifically adds them.

Optional URI Parameters and Default Values

You can make a URI parameter optional by adding a question mark to the route parameter. If a route parameter is optional, you must define a default value for the method parameter.

```
public class BooksController : ApiController
{
    [Route("api/books/locale/{lcid:int?}")]
    public IEnumerable<Book> GetBooksByLocale(int lcid = 1033) { ... }
}
```

In this example, `/api/books/locale/1033` and `/api/books/locale` return the same resource.

Alternatively, you can specify a default value inside the route template, as follows:

```
public class BooksController : ApiController
{
    [Route("api/books/locale/{lcid:int=1033}")]
    public IEnumerable<Book> GetBooksByLocale(int lcid) { ... }
}
```

This is almost the same as the previous example, but there is a slight difference of behavior when the default value is applied.

- In the first example ("{lcid:int?}"), the default value of 1033 is assigned directly to the method parameter, so the parameter will have this exact value.
- In the second example ("{lcid:int=1033}"), the default value of "1033" goes through the model-binding process. The default model-binder will convert "1033" to the numeric value 1033. However, you could plug in a custom model binder, which might do something different.

(In most cases, unless you have custom model binders in your pipeline, the two forms will be equivalent.)

Route Names

In Web API, every route has a name. Route names are useful for generating links, so that you can include a link in an HTTP response.

To specify the route name, set the **Name** property on the attribute. The following example shows how to set the route name, and also how to use the route name when generating a link.

```
public class BooksController : ApiController
{
    [Route("api/books/{id}", Name="GetBookById")]
    public BookDto GetBook(int id)
    {
        // Implementation not shown...
    }

    [Route("api/books")]
    public HttpResponseMessage Post(Book book)
    {
        // Validate and add book to database (not shown)

        var response = Request.CreateResponse(HttpStatusCode.Created);

        // Generate a link to the new book and set the Location header in the response.
        string uri = Url.Link("GetBookById", new { id = book.BookId });
        response.Headers.Location = new Uri(uri);
        return response;
    }
}
```

Route Order

When the framework tries to match a URI with a route, it evaluates the routes in a particular order. To specify the order, set the **Order** property on the route attribute. Lower values are evaluated first. The default order value is zero.

Here is how the total ordering is determined:

1. Compare the **Order** property of the route attribute.
2. Look at each URI segment in the route template. For each segment, order as follows:
   a. Literal segments.
   b. Route parameters with constraints.
   c. Route parameters without constraints.
   d. Wildcard parameter segments with constraints.
   e. Wildcard parameter segments without constraints.
3. In the case of a tie, routes are ordered by a case-insensitive ordinal string comparison (OrdinalIgnoreCase) of the route template.

Here is an example. Suppose you define the following controller:

```
[RoutePrefix("orders")]
public class OrdersController : ApiController
{
    [Route("{id:int}")] // constrained parameter
    public HttpResponseMessage Get(int id) { ... }

    [Route("details")]  // literal
    public HttpResponseMessage GetDetails() { ... }

    [Route("pending", RouteOrder = 1)]
    public HttpResponseMessage GetPending() { ... }

    [Route("{customerName}")]  // unconstrained parameter
    public HttpResponseMessage GetByCustomer(string customerName) { ... }

    [Route("{*date:datetime}")]  // wildcard
    public HttpResponseMessage Get(DateTime date) { ... }
}
```

These routes are ordered as follows.

1. orders/details
2. orders/{id}
3. orders/{customerName}
4. orders/{*date}
5. orders/pending

Notice that "details" is a literal segment and appears before "{id}", but "pending" appears last because the **Order** property is 1. (This example assumes there are no customers named "details" or "pending". In general, try to avoid ambiguous routes. In this example, a better route template for `GetByCustomer` is "customers/{customerName}")

Create a REST API with Attribute Routing in ASP.NET Web API 2

9/16/2020 • 11 minutes to read • Edit Online

by Mike Wasson

Web API 2 supports a new type of routing, called *attribute routing*. For a general overview of attribute routing, see Attribute Routing in Web API 2. In this tutorial, you will use attribute routing to create a REST API for a collection of books. The API will support the following actions:

ACTION	EXAMPLE URI
Get a list of all books.	/api/books
Get a book by ID.	/api/books/1
Get the details of a book.	/api/books/1/details
Get a list of books by genre.	/api/books/fantasy
Get a list of books by publication date.	/api/books/date/2013-02-16 /api/books/date/2013/02/16 (alternate form)
Get a list of books by a particular author.	/api/authors/1/books

All methods are read-only (HTTP GET requests).

For the data layer, we'll use Entity Framework. Book records will have the following fields:

- ID
- Title
- Genre
- Publication date
- Price
- Description
- AuthorID (foreign key to an Authors table)

For most requests, however, the API will return a subset of this data (title, author, and genre). To get the complete record, the client requests `/api/books/{id}/details`.

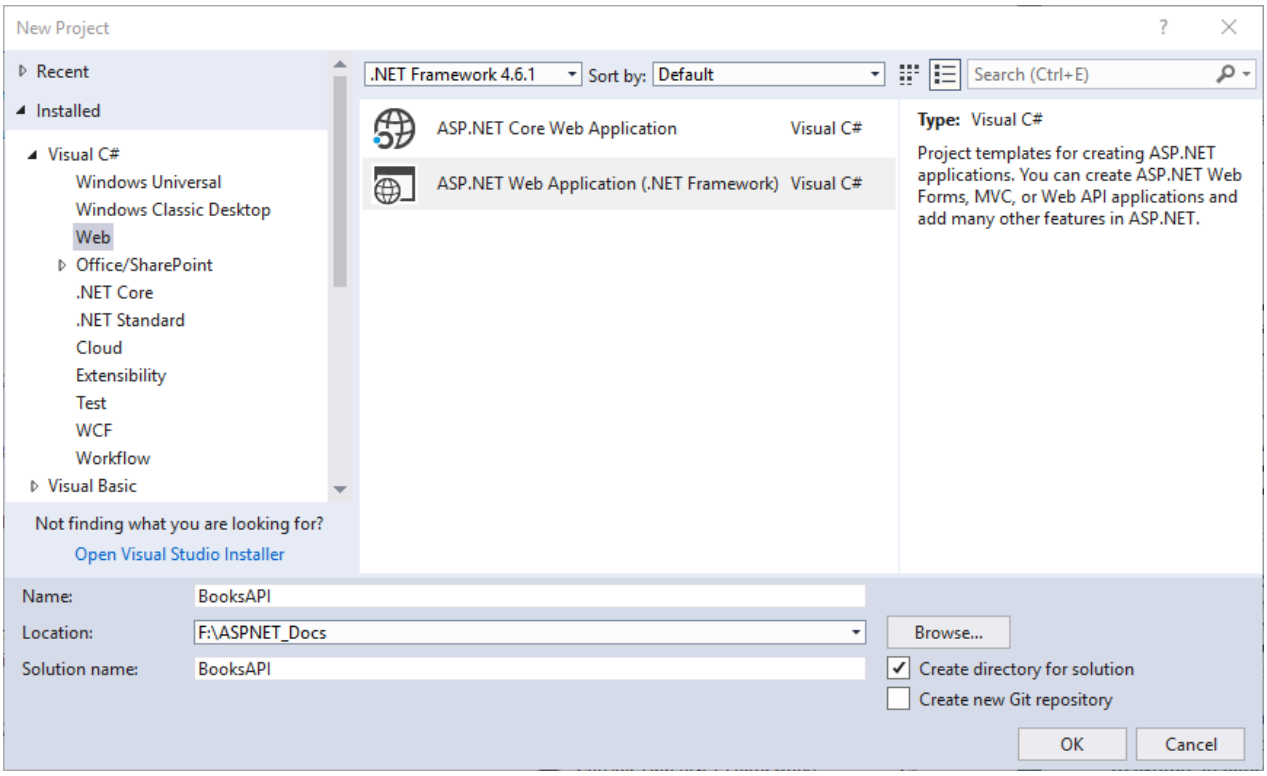
Prerequisites

Visual Studio 2017 Community, Professional or Enterprise edition.

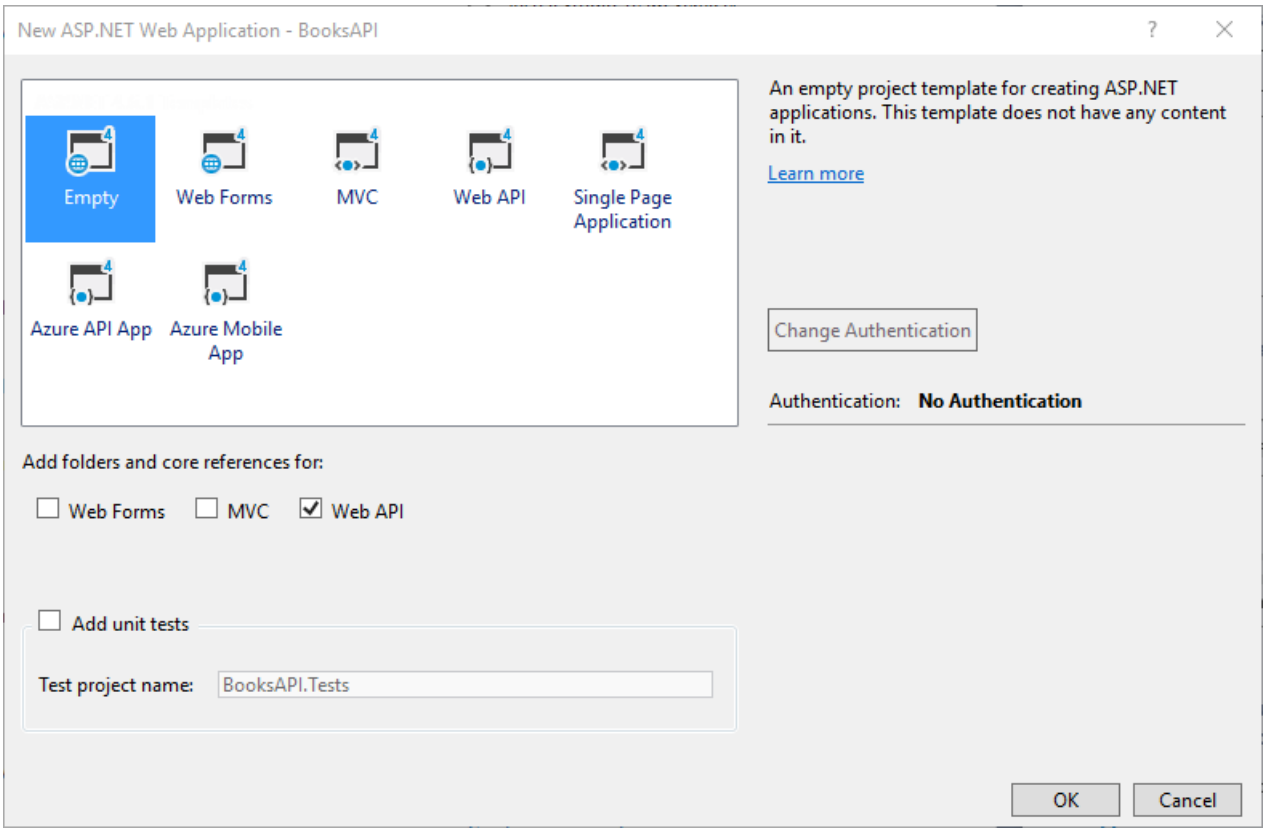
Create the Visual Studio Project

Start by running Visual Studio. From the **File** menu, select **New** and then select **Project**.

Expand the **Installed** > **Visual C#** category. Under **Visual C#**, select **Web**. In the list of project templates, select **ASP.NET Web Application (.NET Framework)**. Name the project "BooksAPI".



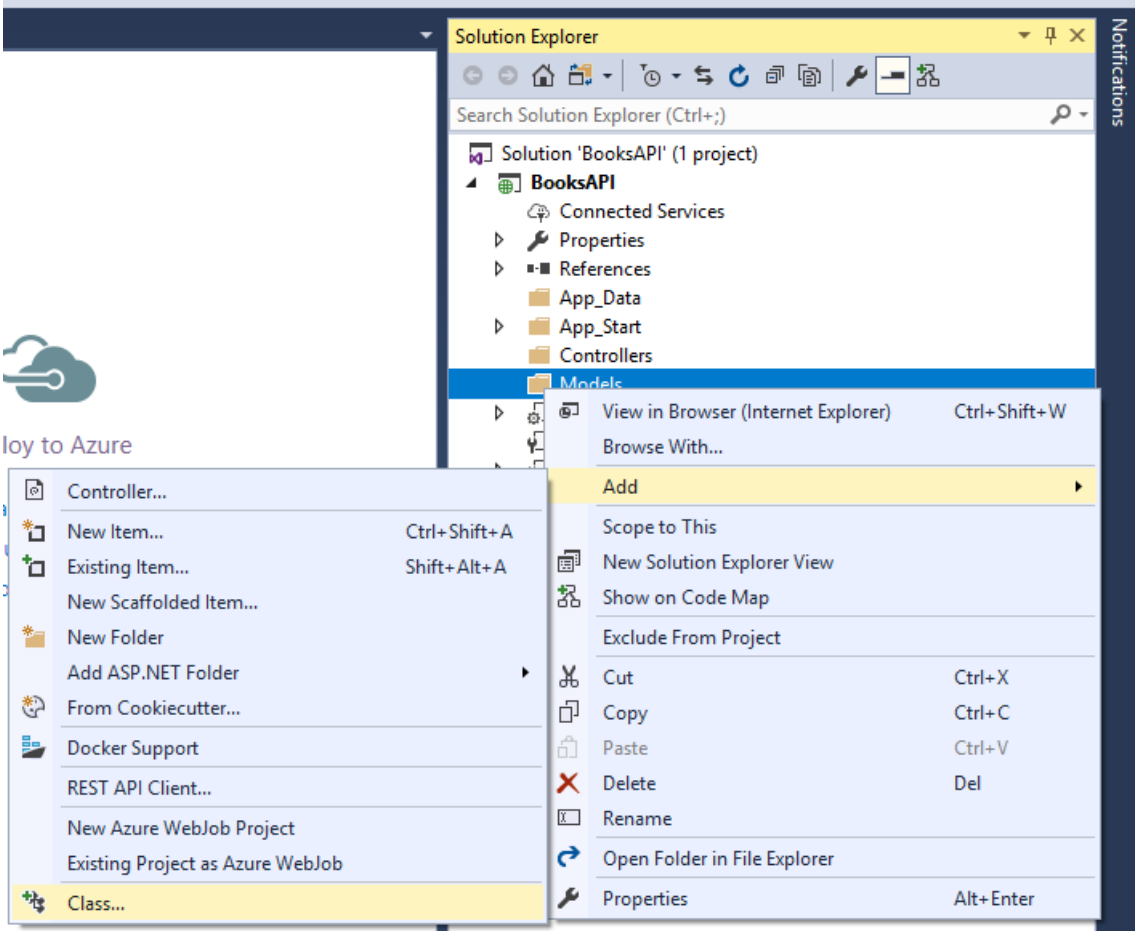
In the **New ASP.NET Web Application** dialog, select the **Empty** template. Under "Add folders and core references for", select the **Web API** checkbox. Click **OK**.



This creates a skeleton project that is configured for Web API functionality.

Domain Models

Next, add classes for domain models. In Solution Explorer, right-click the Models folder. Select **Add**, then select **Class**. Name the class `Author`.



Replace the code in Author.cs with the following:

```
using System.ComponentModel.DataAnnotations;

namespace BooksAPI.Models
{
    public class Author
    {
        public int AuthorId { get; set; }
        [Required]
        public string Name { get; set; }
    }
}
```

Now add another class named `Book`.

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

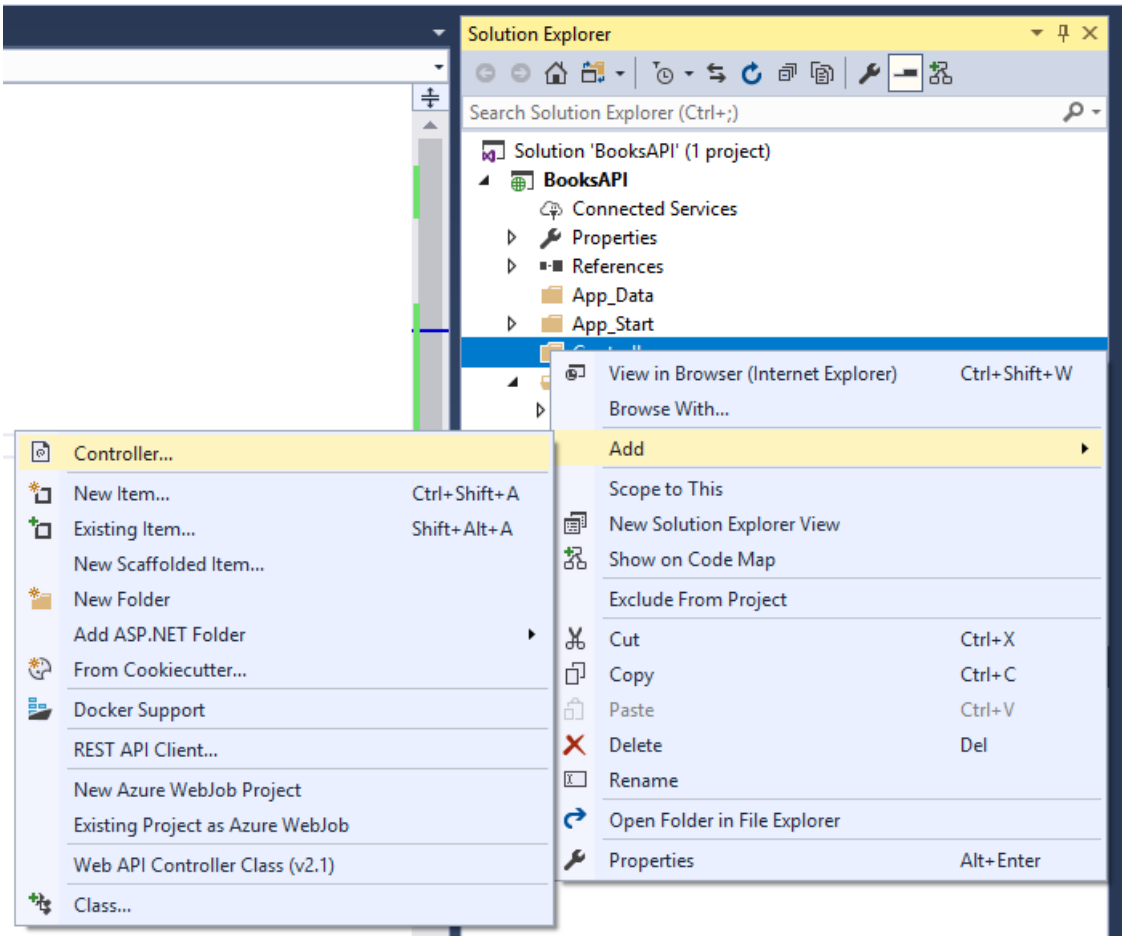
namespace BooksAPI.Models
{
    public class Book
    {
        public int BookId { get; set; }
        [Required]
        public string Title { get; set; }
        public decimal Price { get; set; }
        public string Genre { get; set; }
        public DateTime PublishDate { get; set; }
        public string Description { get; set; }
        public int AuthorId { get; set; }
        [ForeignKey("AuthorId")]
        public Author Author { get; set; }
    }
}
```

Add a Web API Controller

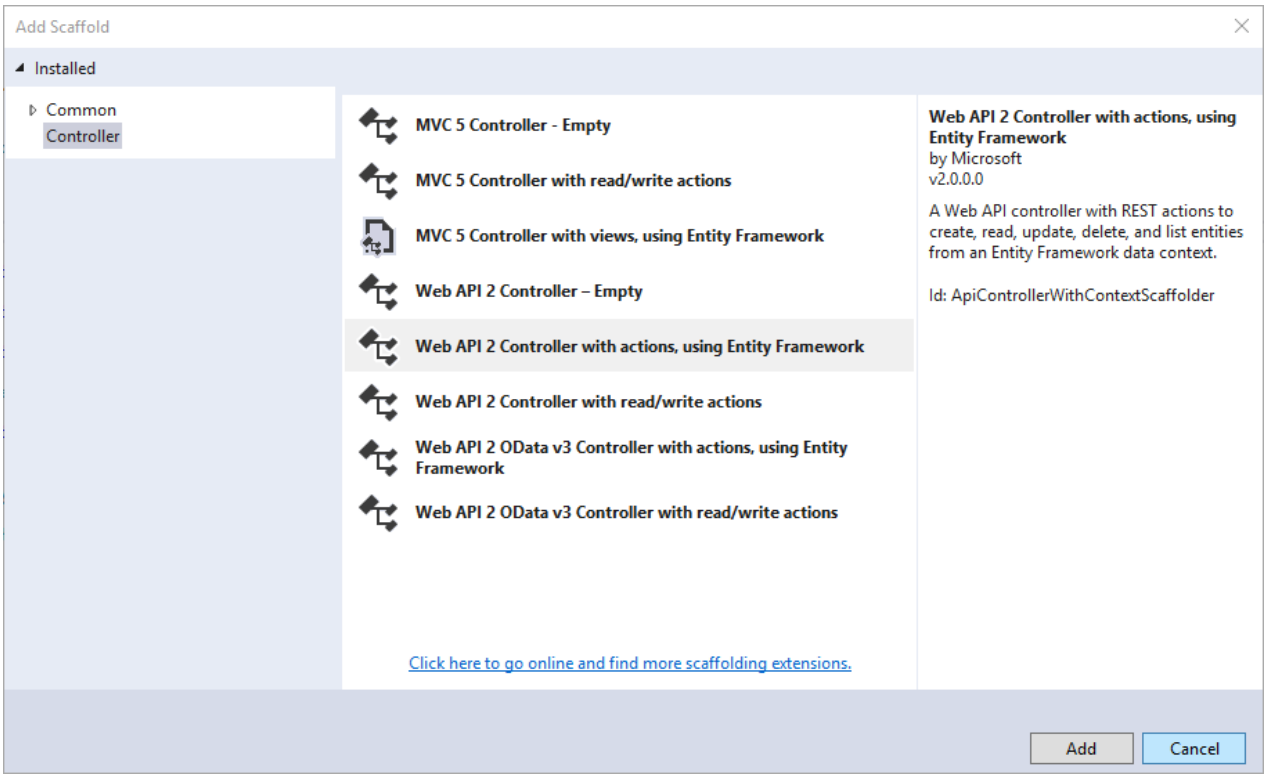
In this step, we'll add a Web API controller that uses Entity Framework as the data layer.

Press CTRL+SHIFT+B to build the project. Entity Framework uses reflection to discover the properties of the models, so it requires a compiled assembly to create the database schema.

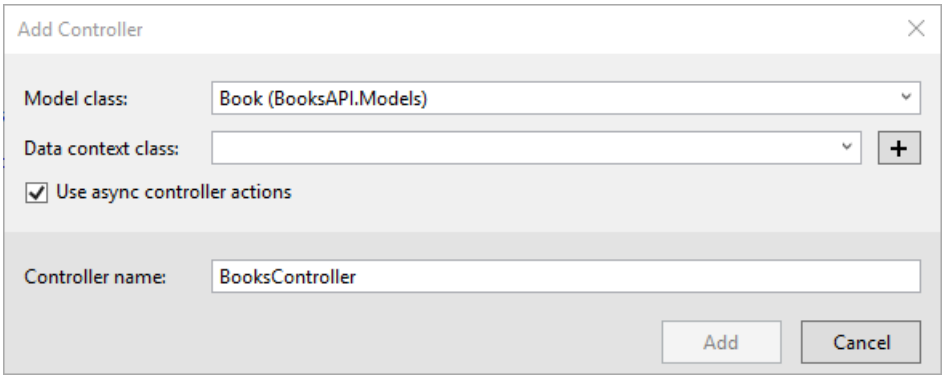
In Solution Explorer, right-click the Controllers folder. Select **Add**, then select **Controller**.



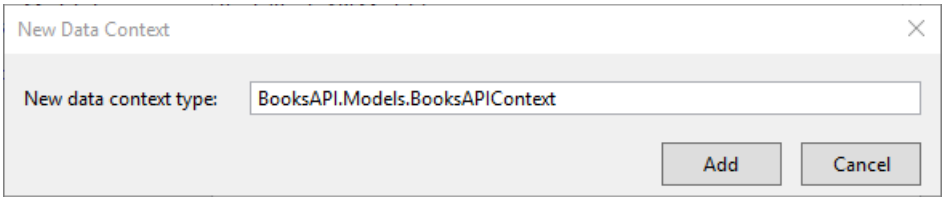
In the **Add Scaffold** dialog, select **Web API 2 Controller with actions, using Entity Framework**.



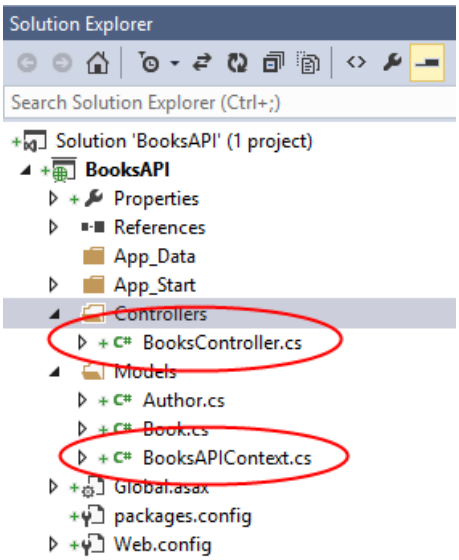
In the **Add Controller** dialog, for **Controller name**, enter "BooksController". Select the "Use async controller actions" checkbox. For **Model class**, select "Book". (If you don't see the `Book` class listed in the dropdown, make sure that you built the project.) Then click the "+" button.



Click **Add** in the **New Data Context** dialog.



Click **Add** in the **Add Controller** dialog. The scaffolding adds a class named `BooksController` that defines the API controller. It also adds a class named `BooksAPIContext` in the Models folder, which defines the data context for Entity Framework.



Seed the Database

From the Tools menu, select **NuGet Package Manager**, and then select **Package Manager Console**.

In the Package Manager Console window, enter the following command:

```
Add-Migration
```

This command creates a Migrations folder and adds a new code file named Configuration.cs. Open this file and add the following code to the `Configuration.Seed` method.

```
protected override void Seed(BooksAPI.Models.BooksAPIContext context)
{
    context.Authors.AddOrUpdate(new Author[] {
        new Author() { AuthorId = 1, Name = "Ralls, Kim" },
        new Author() { AuthorId = 2, Name = "Corets, Eva" },
        new Author() { AuthorId = 3, Name = "Randall, Cynthia" },
        new Author() { AuthorId = 4, Name = "Thurman, Paula" }
        });

    context.Books.AddOrUpdate(new Book[] {
        new Book() { BookId = 1,  Title= "Midnight Rain", Genre = "Fantasy",
        PublishDate = new DateTime(2000, 12, 16), AuthorId = 1, Description =
        "A former architect battles an evil sorceress.", Price = 14.95M },

        new Book() { BookId = 2, Title = "Maeve Ascendant", Genre = "Fantasy",
            PublishDate = new DateTime(2000, 11, 17), AuthorId = 2, Description =
            "After the collapse of a nanotechnology society, the young" +
            "survivors lay the foundation for a new society.", Price = 12.95M },

        new Book() { BookId = 3, Title = "The Sundered Grail", Genre = "Fantasy",
            PublishDate = new DateTime(2001, 09, 10), AuthorId = 2, Description =
            "The two daughters of Maeve battle for control of England.", Price = 12.95M },

        new Book() { BookId = 4, Title = "Lover Birds", Genre = "Romance",
            PublishDate = new DateTime(2000, 09, 02), AuthorId = 3, Description =
            "When Carla meets Paul at an ornithology conference, tempers fly.", Price = 7.99M },

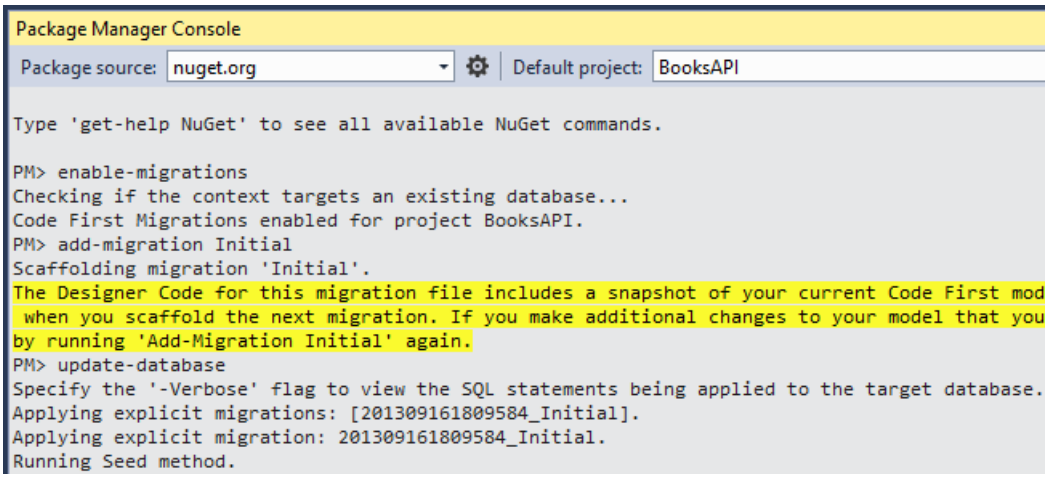
        new Book() { BookId = 5, Title = "Splish Splash", Genre = "Romance",
            PublishDate = new DateTime(2000, 11, 02), AuthorId = 4, Description =
            "A deep sea diver finds true love 20,000 leagues beneath the sea.", Price = 6.99M},
    });
}
```

In the Package Manager Console window, type the following commands.

```
add-migration Initial

update-database
```

These commands create a local database and invoke the Seed method to populate the database.



Add DTO Classes

If you run the application now and send a GET request to /api/books/1, the response looks similar to the following. (I added indentation for readability.)

```
{
  "BookId": 1,
  "Title": "Midnight Rain",
  "Genre": "Fantasy",
  "PublishDate": "2000-12-16T00:00:00",
  "Description": "A former architect battles an evil sorceress.",
  "Price": 14.95,
  "AuthorId": 1,
  "Author": null
}
```

Instead, I want this request to return a subset of the fields. Also, I want it to return the author's name, rather than the author ID. To accomplish this, we'll modify the controller methods to return a *data transfer object* (DTO) instead of the EF model. A DTO is an object that is designed only to carry data.

In Solution Explorer, right-click the project and select **Add | New Folder**. Name the folder "DTOs". Add a class named `BookDto` to the DTOs folder, with the following definition:

```
namespace BooksAPI.DTOs
{
    public class BookDto
    {
        public string Title { get; set; }
        public string Author { get; set; }
        public string Genre { get; set; }
    }
}
```

Add another class named `BookDetailDto`.

```
using System;

namespace BooksAPI.DTOs
{
    public class BookDetailDto
    {
        public string Title { get; set; }
        public string Genre { get; set; }
        public DateTime PublishDate { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Author { get; set; }
    }
}
```

Next, update the `BooksController` class to return `BookDto` instances. We'll use the Queryable.Select method to project `Book` instances to `BookDto` instances. Here is the updated code for the controller class.

```
using BooksAPI.DTOs;
using BooksAPI.Models;
using System;
using System.Data.Entity;
using System.Linq;
using System.Linq.Expressions;
using System.Threading.Tasks;
using System.Web.Http;
using System.Web.Http.Description;

namespace BooksAPI.Controllers
{
    public class BooksController : ApiController
    {
        private BooksAPIContext db = new BooksAPIContext();

        // Typed lambda expression for Select() method.
        private static readonly Expression<Func<Book, BookDto>> AsBookDto =
            x => new BookDto
            {
                Title = x.Title,
                Author = x.Author.Name,
                Genre = x.Genre
            };

        // GET api/Books
        public IQueryable<BookDto> GetBooks()
        {
            return db.Books.Include(b => b.Author).Select(AsBookDto);
        }

        // GET api/Books/5
        [ResponseType(typeof(BookDto))]
        public async Task<IHttpActionResult> GetBook(int id)
        {
            BookDto book = await db.Books.Include(b => b.Author)
                .Where(b => b.BookId == id)
                .Select(AsBookDto)
                .FirstOrDefaultAsync();
            if (book == null)
            {
                return NotFound();
            }

            return Ok(book);
        }

        protected override void Dispose(bool disposing)
        {
            db.Dispose();
            base.Dispose(disposing);
        }
    }
}
```

NOTE

I deleted the `PutBook`, `PostBook`, and `DeleteBook` methods, because they aren't needed for this tutorial.

Now if you run the application and request /api/books/1, the response body should look like this:

```
{"Title":"Midnight Rain","Author":"Ralls, Kim","Genre":"Fantasy"}
```

Add Route Attributes

Next, we'll convert the controller to use attribute routing. First, add a **RoutePrefix** attribute to the controller. This attribute defines the initial URI segments for all methods on this controller.

```
[RoutePrefix("api/books")]
public class BooksController : ApiController
{
    // ...
```

Then add **[Route]** attributes to the controller actions, as follows:

```
[Route("")]
public IQueryable<BookDto> GetBooks()
{
    // ...
}

[Route("{id:int}")]
[ResponseType(typeof(BookDto))]
public async Task<IHttpActionResult> GetBook(int id)
{
    // ...
}
```

The route template for each controller method is the prefix plus the string specified in the **Route** attribute. For the `GetBook` method, the route template includes the parameterized string "{id:int}", which matches if the URI segment contains an integer value.

METHOD	ROUTE TEMPLATE	EXAMPLE URI
`GetBooks`	"api/books"	`http://localhost/api/books`
`GetBook`	"api/books/{id:int}"	`http://localhost/api/books/5`

Get Book Details

To get book details, the client will send a GET request to `/api/books/{id}/details`, where *{id}* is the ID of the book.

Add the following method to the `BooksController` class.

```
[Route("{id:int}/details")]
[ResponseType(typeof(BookDetailDto))]
public async Task<IHttpActionResult> GetBookDetail(int id)
{
    var book = await (from b in db.Books.Include(b => b.Author)
                where b.BookId == id
                select new BookDetailDto
                {
                    Title = b.Title,
                    Genre = b.Genre,
                    PublishDate = b.PublishDate,
                    Price = b.Price,
                    Description = b.Description,
                    Author = b.Author.Name
                }).FirstOrDefaultAsync();

    if (book == null)
    {
        return NotFound();
    }
    return Ok(book);
}
```

If you request `/api/books/1/details`, the response looks like this:

```
{
  "Title": "Midnight Rain",
  "Genre": "Fantasy",
  "PublishDate": "2000-12-16T00:00:00",
  "Description": "A former architect battles an evil sorceress.",
  "Price": 14.95,
  "Author": "Ralls, Kim"
}
```

Get Books By Genre

To get a list of books in a specific genre, the client will send a GET request to `/api/books/genre`, where *genre* is the name of the genre. (For example, `/api/books/fantasy`.)

Add the following method to `BooksController`.

```
[Route("{genre}")]
public IQueryable<BookDto> GetBooksByGenre(string genre)
{
    return db.Books.Include(b => b.Author)
        .Where(b => b.Genre.Equals(genre, StringComparison.OrdinalIgnoreCase))
        .Select(AsBookDto);
}
```

Here we are defining a route that contains a {genre} parameter in the URI template. Notice that Web API is able to distinguish these two URIs and route them to different methods:

`/api/books/1`

`/api/books/fantasy`

That's because the `GetBook` method includes a constraint that the "id" segment must be an integer value:

```
[Route("{id:int}")]
public BookDto GetBook(int id)
{
    // ...
}
```

If you request /api/books/fantasy, the response looks like this:

```
[ { "Title": "Midnight Rain", "Author": "Ralls, Kim", "Genre": "Fantasy" }, { "Title": "Maeve Ascendant",
"Author": "Corets, Eva", "Genre": "Fantasy" }, { "Title": "The Sundered Grail", "Author": "Corets, Eva",
"Genre": "Fantasy" } ]
```

Get Books By Author

To get a list of a books for a particular author, the client will send a GET request to `/api/authors/id/books`, where *id* is the ID of the author.

Add the following method to `BooksController`.

```
[Route("~/api/authors/{authorId:int}/books")]
public IQueryable<BookDto> GetBooksByAuthor(int authorId)
{
    return db.Books.Include(b => b.Author)
        .Where(b => b.AuthorId == authorId)
        .Select(AsBookDto);
}
```

This example is interesting because "books" is treated a child resource of "authors". This pattern is quite common in RESTful APIs.

The tilde (~) in the route template overrides the route prefix in the **RoutePrefix** attribute.

Get Books By Publication Date

To get a list of books by publication date, the client will send a GET request to `/api/books/date/yyyy-mm-dd`, where *yyyy-mm-dd* is the date.

Here is one way to do this:

```
[Route("date/{pubdate:datetime}")]
public IQueryable<BookDto> GetBooks(DateTime pubdate)
{
    return db.Books.Include(b => b.Author)
        .Where(b => DbFunctions.TruncateTime(b.PublishDate)
            == DbFunctions.TruncateTime(pubdate))
        .Select(AsBookDto);
}
```

The `{pubdate:datetime}` parameter is constrained to match a **DateTime** value. This works, but it's actually more permissive than we'd like. For example, these URIs will also match the route:

`/api/books/date/Thu, 01 May 2008`

`/api/books/date/2000-12-16T00:00:00`

There's nothing wrong with allowing these URIs. However, you can restrict the route to a particular format by adding a regular-expression constraint to the route template:

```
[Route("date/{pubdate:datetime:regex(\\d{4}-\\d{2}-\\d{2})}")]
public IQueryable<BookDto> GetBooks(DateTime pubdate)
{
    // ...
}
```

Now only dates in the form "yyyy-mm-dd" will match. Notice that we don't use the regex to validate that we got a real date. That is handled when Web API tries to convert the URI segment into a **DateTime** instance. An invalid date such as '2012-47-99' will fail to be converted, and the client will get a 404 error.

You can also support a slash separator (`/api/books/date/yyyy/mm/dd`) by adding another **[Route]** attribute with a different regex.

```
[Route("date/{pubdate:datetime:regex(\\d{4}-\\d{2}-\\d{2})}")]
[Route("date/{*pubdate:datetime:regex(\\d{4}/\\d{2}/\\d{2})}")]  // new
public IQueryable<BookDto> GetBooks(DateTime pubdate)
{
    // ...
}
```

There is a subtle but important detail here. The second route template has a wildcard character (*) at the start of the {pubdate} parameter:

```
{*pubdate: ... }
```

This tells the routing engine that {pubdate} should match the rest of the URI. By default, a template parameter matches a single URI segment. In this case, we want {pubdate} to span several URI segments:

`/api/books/date/2013/06/17`

Controller Code

Here is the complete code for the BooksController class.

```
using BooksAPI.DTOs;
using BooksAPI.Models;
using System;
using System.Data.Entity;
using System.Linq;
using System.Linq.Expressions;
using System.Threading.Tasks;
using System.Web.Http;
using System.Web.Http.Description;

namespace BooksAPI.Controllers
{
    [RoutePrefix("api/books")]
    public class BooksController : ApiController
    {
        private BooksAPIContext db = new BooksAPIContext();

        // Typed lambda expression for Select() method.
        private static readonly Expression<Func<Book, BookDto>> AsBookDto =
            x => new BookDto
            {
                Title = x.Title,
                Author = x.Author.Name,
                Genre = x.Genre
            };
```

```
        // GET api/Books
        [Route("")]
        public IQueryable<BookDto> GetBooks()
        {
            return db.Books.Include(b => b.Author).Select(AsBookDto);
        }

        // GET api/Books/5
        [Route("{id:int}")]
        [ResponseType(typeof(BookDto))]
        public async Task<IHttpActionResult> GetBook(int id)
        {
            BookDto book = await db.Books.Include(b => b.Author)
                .Where(b => b.BookId == id)
                .Select(AsBookDto)
                .FirstOrDefaultAsync();
            if (book == null)
            {
                return NotFound();
            }

            return Ok(book);
        }

        [Route("{id:int}/details")]
        [ResponseType(typeof(BookDetailDto))]
        public async Task<IHttpActionResult> GetBookDetail(int id)
        {
            var book = await (from b in db.Books.Include(b => b.Author)
                              where b.AuthorId == id
                              select new BookDetailDto
                              {
                                  Title = b.Title,
                                  Genre = b.Genre,
                                  PublishDate = b.PublishDate,
                                  Price = b.Price,
                                  Description = b.Description,
                                  Author = b.Author.Name
                              }).FirstOrDefaultAsync();

            if (book == null)
            {
                return NotFound();
            }
            return Ok(book);
        }

        [Route("{genre}")]
        public IQueryable<BookDto> GetBooksByGenre(string genre)
        {
            return db.Books.Include(b => b.Author)
                .Where(b => b.Genre.Equals(genre, StringComparison.OrdinalIgnoreCase))
                .Select(AsBookDto);
        }

        [Route("~/api/authors/{authorId}/books")]
        public IQueryable<BookDto> GetBooksByAuthor(int authorId)
        {
            return db.Books.Include(b => b.Author)
                .Where(b => b.AuthorId == authorId)
                .Select(AsBookDto);
        }

        [Route("date/{pubdate:datetime:regex(\\d{4}-\\d{2}-\\d{2})}")]
        [Route("date/{*pubdate:datetime:regex(\\d{4}/\\d{2}/\\d{2})}")]
        public IQueryable<BookDto> GetBooks(DateTime pubdate)
        {
            return db.Books.Include(b => b.Author)
                .Where(b => DbFunctions.TruncateTime(b.PublishDate)
```

```
                .Where(b => DbFunctions.TruncateTime(b.PublishDate)
                    == DbFunctions.TruncateTime(pubdate))
                .Select(AsBookDto);
        }

        protected override void Dispose(bool disposing)
        {
            db.Dispose();
            base.Dispose(disposing);
        }
    }
}
```

Summary

Attribute routing gives you more control and greater flexibility when designing the URIs for your API.

Working with Data in ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

How to program CRUD (create, read, update, delete) operations in Web API.

- Using Web API 2 with Entity Framework 6
  - Create the Project
  - Add Models and Controllers
  - Seed the Database
  - Handling Entity Relations
  - Create Data Transfer Objects (DTOs)
  - Create the JavaScript Client
  - Create the View (UI)
  - Display Item Details
  - Add a New Item to the Database
  - Publish the App to Azure Websites

Using Web API 2 with Entity Framework 6

2/20/2020 • 2 minutes to read • Edit Online

How to create a Web API application that uses Entity Framework for database persistence.

- Create the Project
- Add Models and Controllers
- Seed the Database
- Handling Entity Relations
- Create Data Transfer Objects (DTOs)
- Create the JavaScript Client
- Create the View (UI)
- Display Item Details
- Add a New Item to the Database
- Publish the App to Azure Websites

Using Web API 2 with Entity Framework 6

2/20/2020 • 2 minutes to read • Edit Online

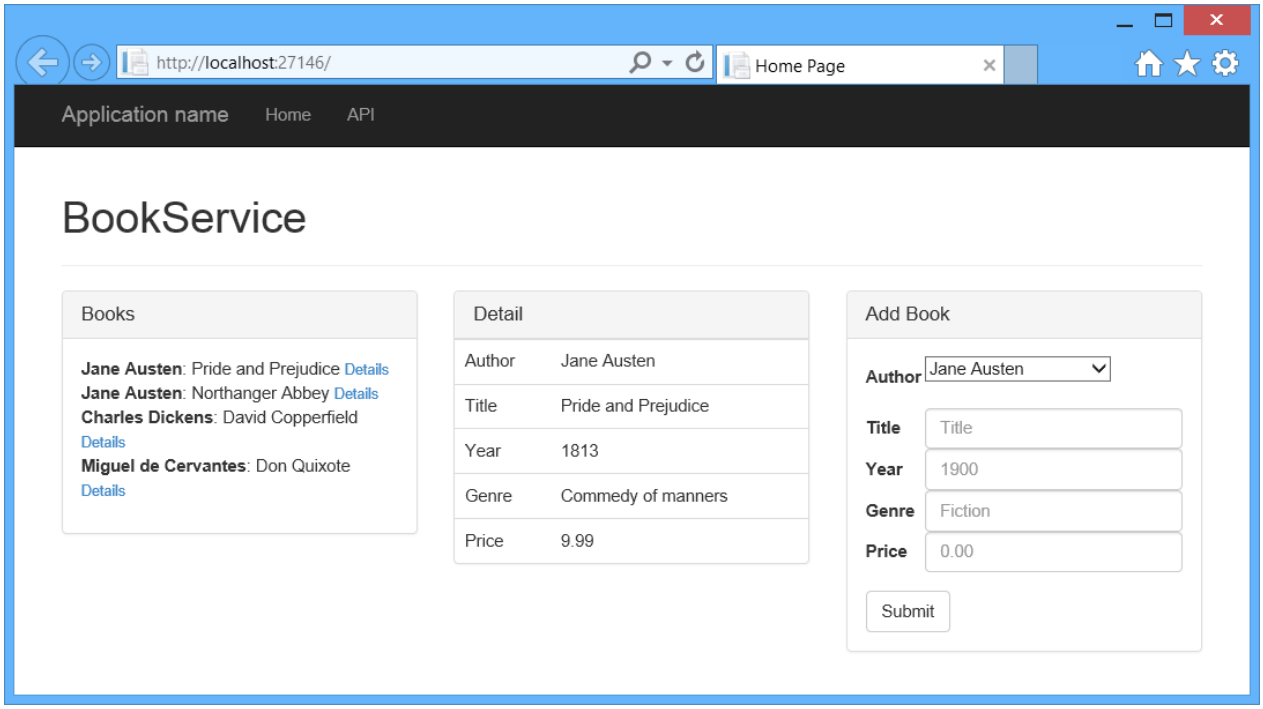
Download Completed Project

This tutorial teaches you the basics of creating a web application with an ASP.NET Web API back end. The tutorial uses Entity Framework 6 for the data layer, and Knockout.js for the client-side JavaScript application. The tutorial also shows how to deploy the app to Azure App Service Web Apps.

Software versions used in the tutorial

- Web API 2.1
- Visual Studio 2017 (download Visual Studio 2017 here)
- Entity Framework 6
- .NET 4.7
- Knockout.js 3.1

This tutorial uses ASP.NET Web API 2 with Entity Framework 6 to create a web application that manipulates a back-end database. Here is a screen shot of the application that you will create.



The app uses a single-page application (SPA) design. "Single-page application" is the general term for a web application that loads a single HTML page and then updates the page dynamically, instead of loading new pages. After the initial page load, the app talks with the server through AJAX requests. The AJAX requests return JSON data, which the app uses to update the UI.

AJAX isn't new, but today there are JavaScript frameworks that make it easier to build and maintain a large sophisticated SPA application. This tutorial uses Knockout.js, but you can use any JavaScript client framework.

Here are the main building blocks for this app:

- ASP.NET MVC creates the HTML page.

- ASP.NET Web API handles the AJAX requests and returns JSON data.
- Knockout.js data-binds the HTML elements to the JSON data.
- Entity Framework talks to the database.

See this app running on Azure

Would you like to see the finished site running as a live web app? You can deploy a complete version of the app to your Azure account by selecting the following button.

Deploy to Azure

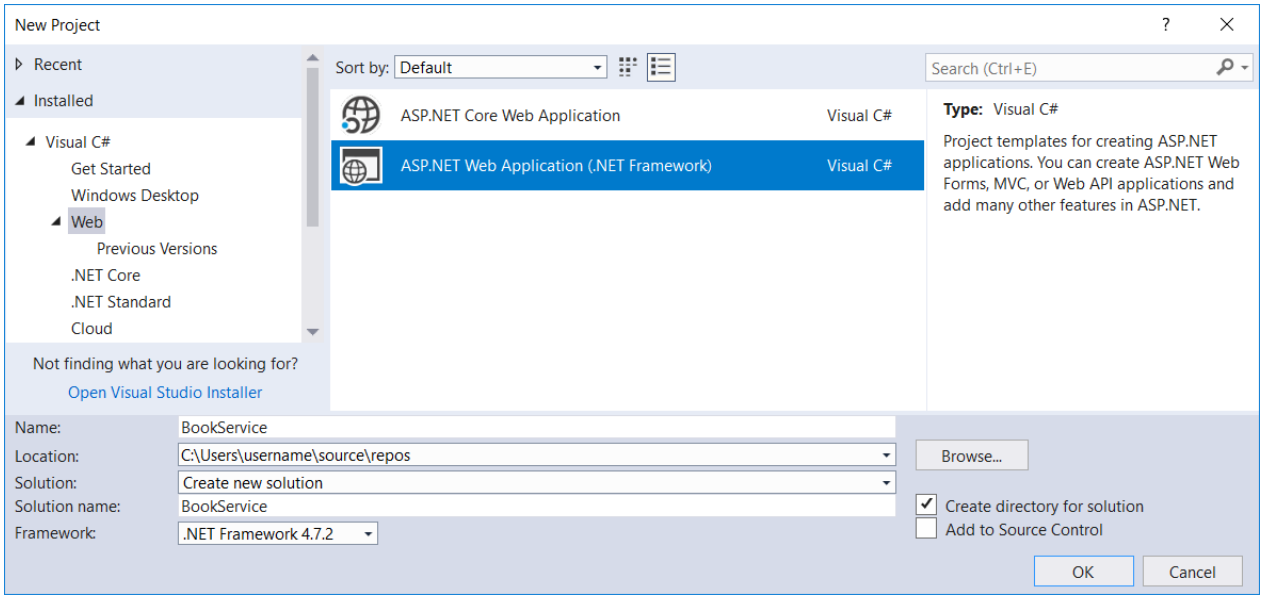
You need an Azure account to deploy this solution to Azure. If you do not already have an account, you have the following options:

- Open an Azure account for free - You get credits you can use to try out paid Azure services, and even after they're used up you can keep the account and use free Azure services.
- Activate MSDN subscriber benefits - Your MSDN subscription gives you credits every month that you can use for paid Azure services.

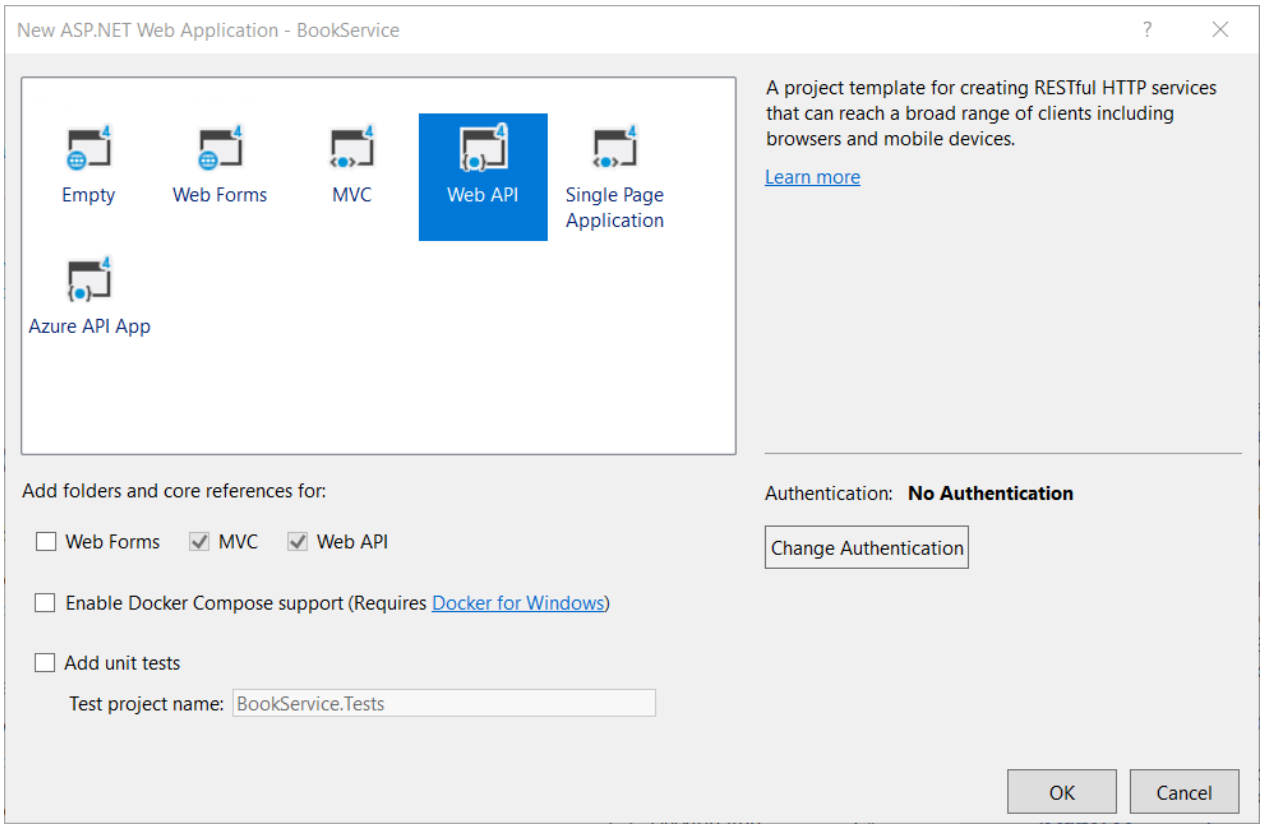
Create the project

Open Visual Studio. From the **File** menu, select **New**, then select **Project**. (Or select **New Project** on the Start page.)

In the **New Project** dialog, select **Web** in the left pane and **ASP.NET Web Application (.NET Framework)** in the middle pane. Name the project **BookService** and select **OK**.



In the **New ASP.NET Project** dialog, select the **Web API** template.

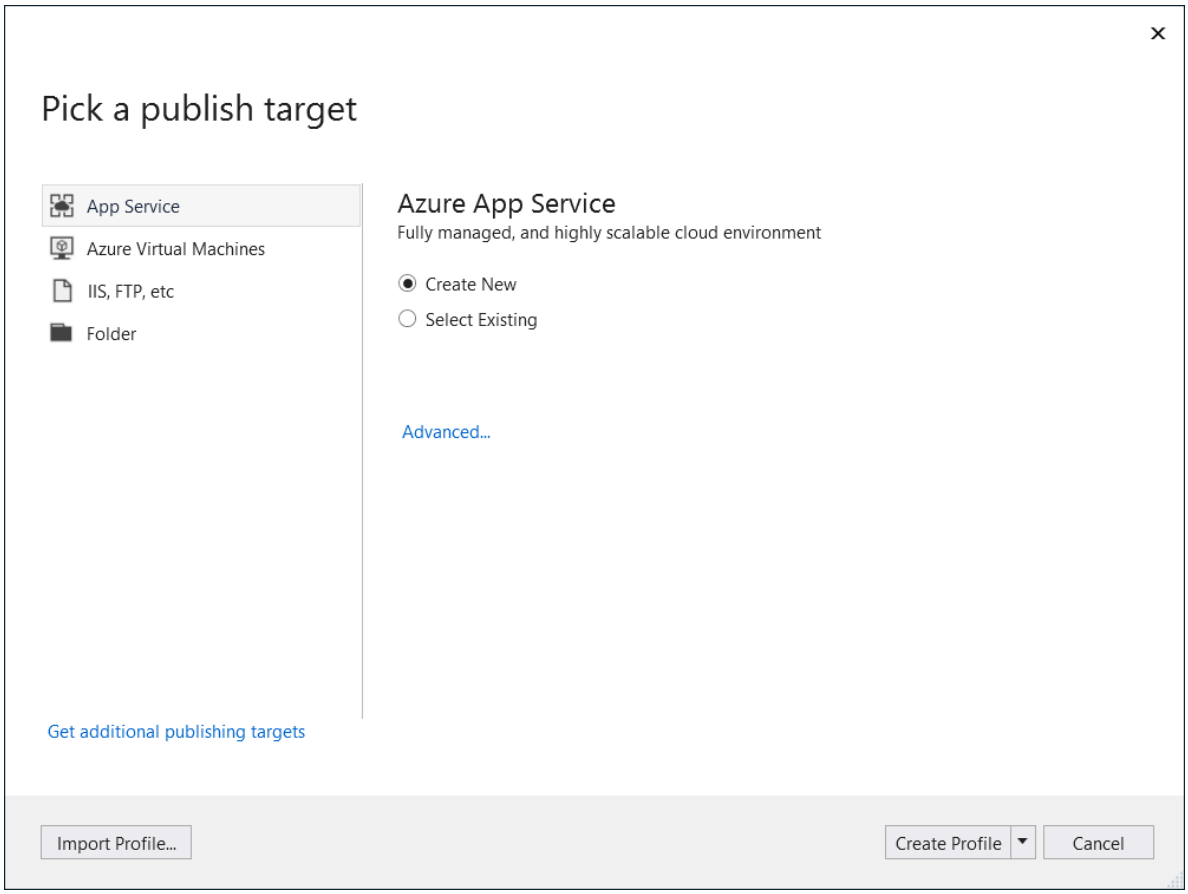


Select **OK** to create the project.

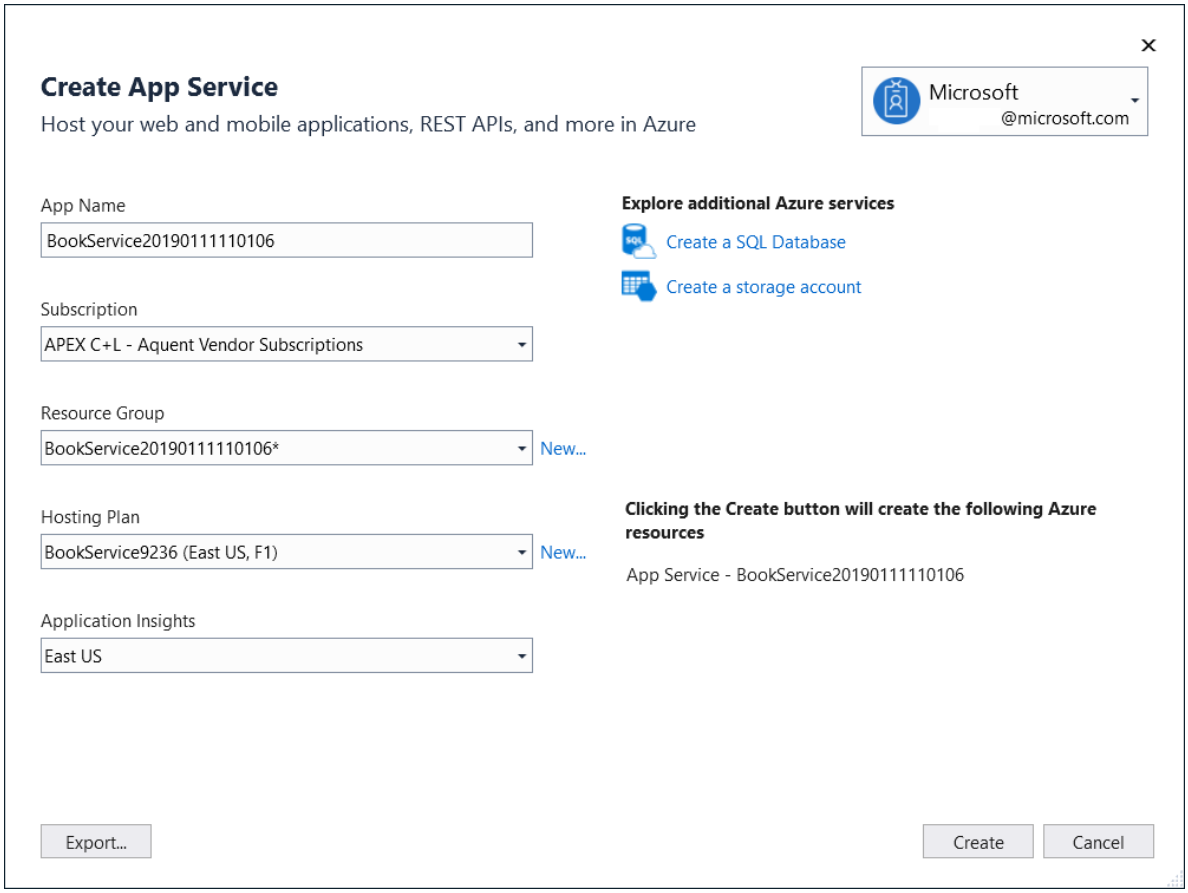
Configure Azure settings (optional)

After you create the project, you can choose to deploy to Azure App Service Web Apps at any time.

1. In Solution Explorer, right-click on your project and select **Publish**.
2. In the window that appears, select **Start**. The **Pick a publish target** dialog box appears.

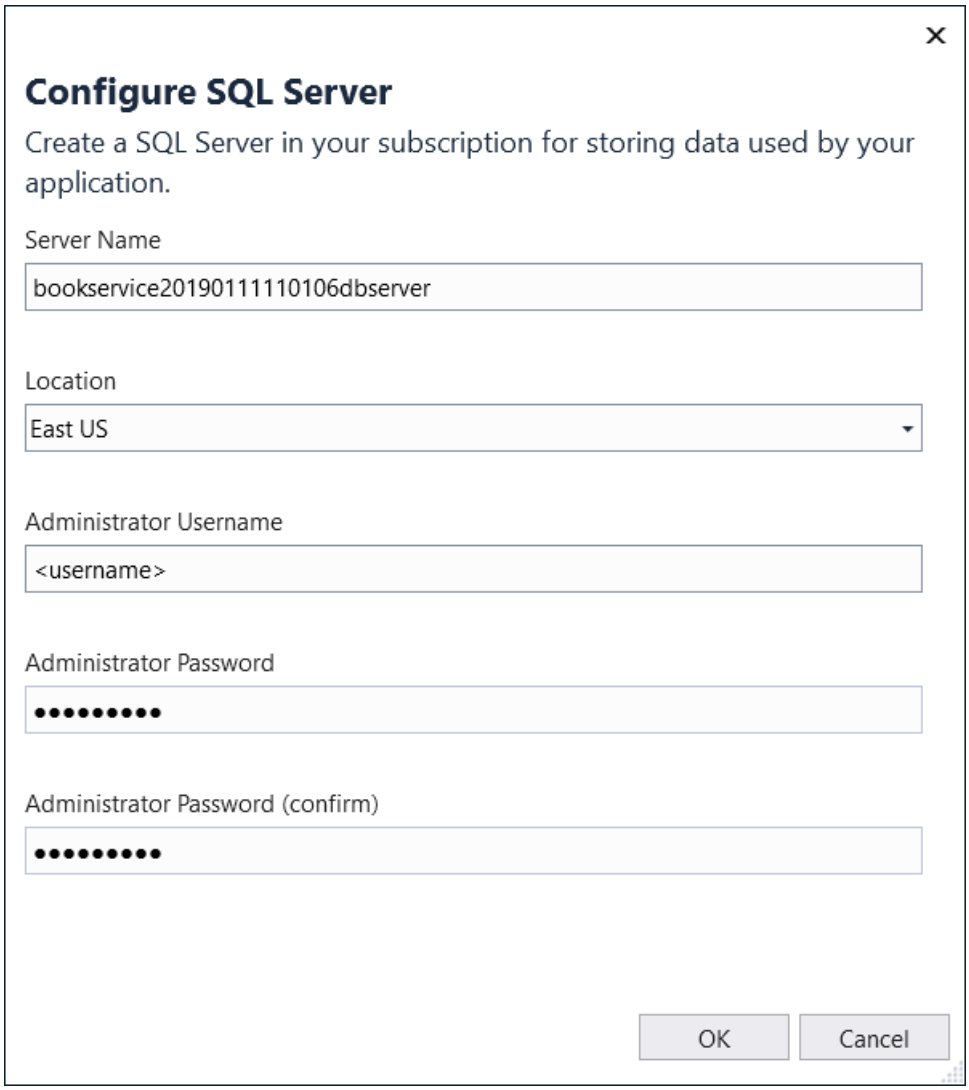


3. Select **Create Profile**. The **Create App Service** dialog box appears.



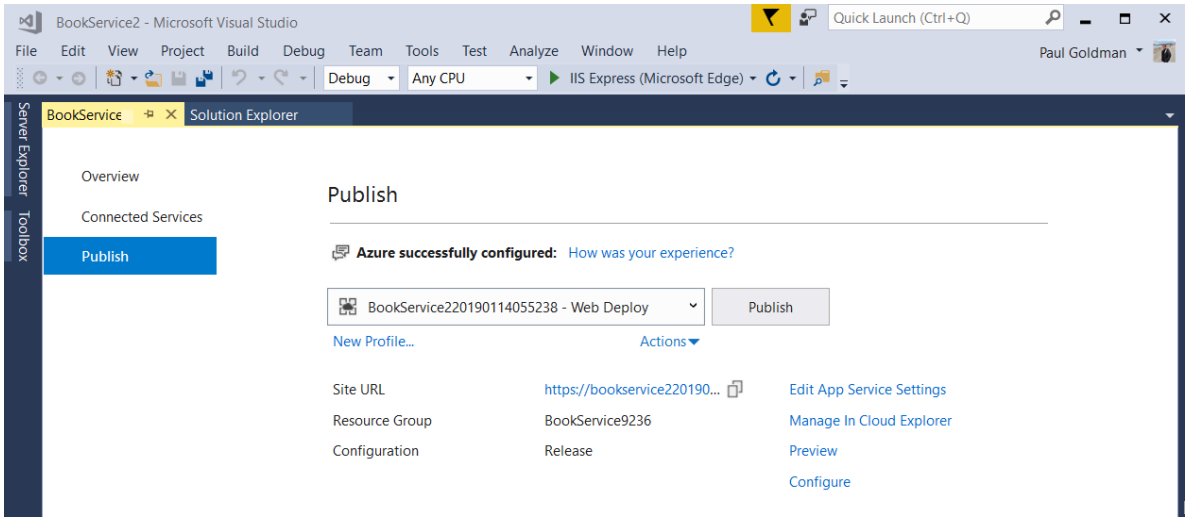
Accept the defaults, or enter different values for the application name, resource group, hosting plan, Azure subscription, and geographical region.

4. Select **Create a SQL database**. The **Configure SQL Server** dialog box appears.



Accept the defaults or enter different values. Enter an **Administrator Username** and **Administrator Password** for your new database. Select **OK** when you're done. The **Create App Service** page reappears.

5. Select **Create** to create your profile. A message appears in the lower-right corner indicating that deployment is in progress. After a short while, the **Publish** window reappears.



The profile you created to deploy the app is now available.

NEXT

Add Models and Controllers

2/20/2020 • 3 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

In this section, you will add model classes that define the database entities. Then you will add Web API controllers that perform CRUD operations on those entities.

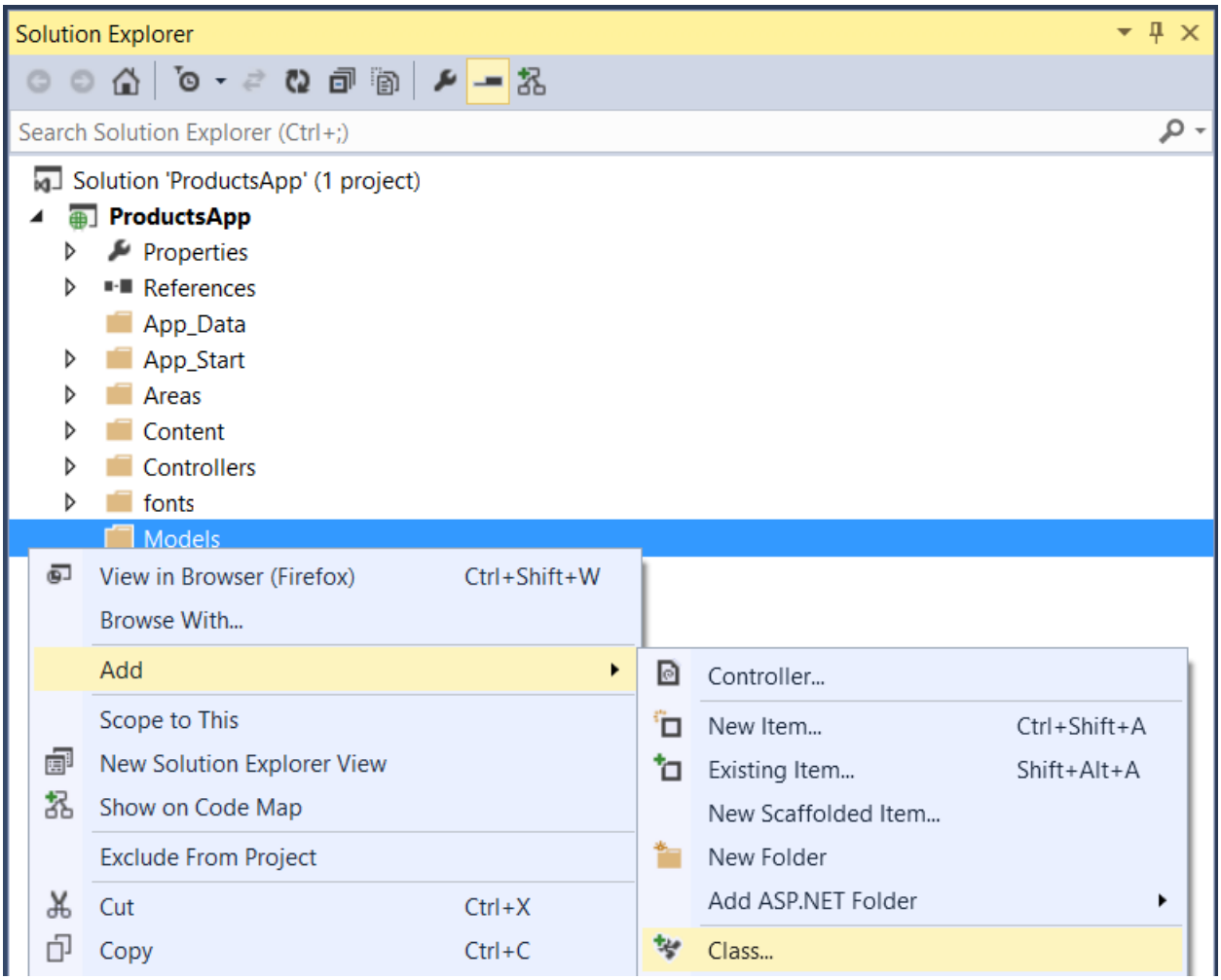
Add Model Classes

In this tutorial, we'll create the database by using the "Code First" approach to Entity Framework (EF). With Code First, you write C# classes that correspond to database tables, and EF creates the database. (For more information, see Entity Framework Development Approaches.)

We start by defining our domain objects as POCOs (plain-old CLR objects). We will create the following POCOs:

- Author
- Book

In Solution Explorer, right click the Models folder. Select **Add**, then select **Class**. Name the class `Author`.



Replace all of the boilerplate code in Author.cs with the following code.

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace BookService.Models
{
    public class Author
    {
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
    }
}
```

Add another class named `Book`, with the following code.

```
using System.ComponentModel.DataAnnotations;

namespace BookService.Models
{
    public class Book
    {
        public int Id { get; set; }
        [Required]
        public string Title { get; set; }
        public int Year { get; set; }
        public decimal Price { get; set; }
        public string Genre { get; set; }

        // Foreign Key
        public int AuthorId { get; set; }
        // Navigation property
        public Author Author { get; set; }
    }
}
```

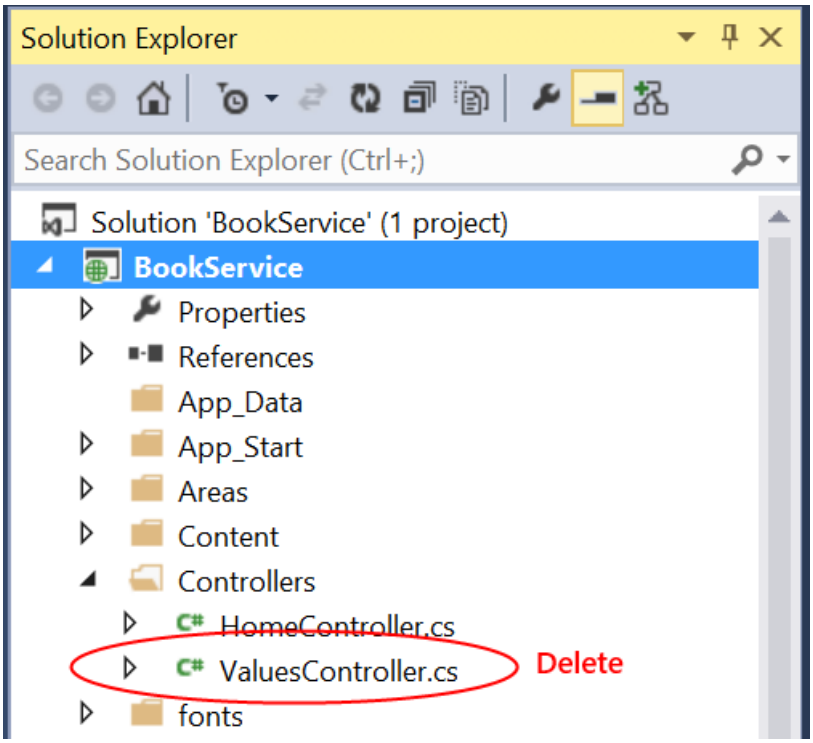
Entity Framework will use these models to create database tables. For each model, the `Id` property will become the primary key column of the database table.

In the Book class, the `AuthorId` defines a foreign key into the `Author` table. (For simplicity, I'm assuming that each book has a single author.) The book class also contains a navigation property to the related `Author`. You can use the navigation property to access the related `Author` in code. I say more about navigation properties in part 4, Handling Entity Relations.

Add Web API Controllers

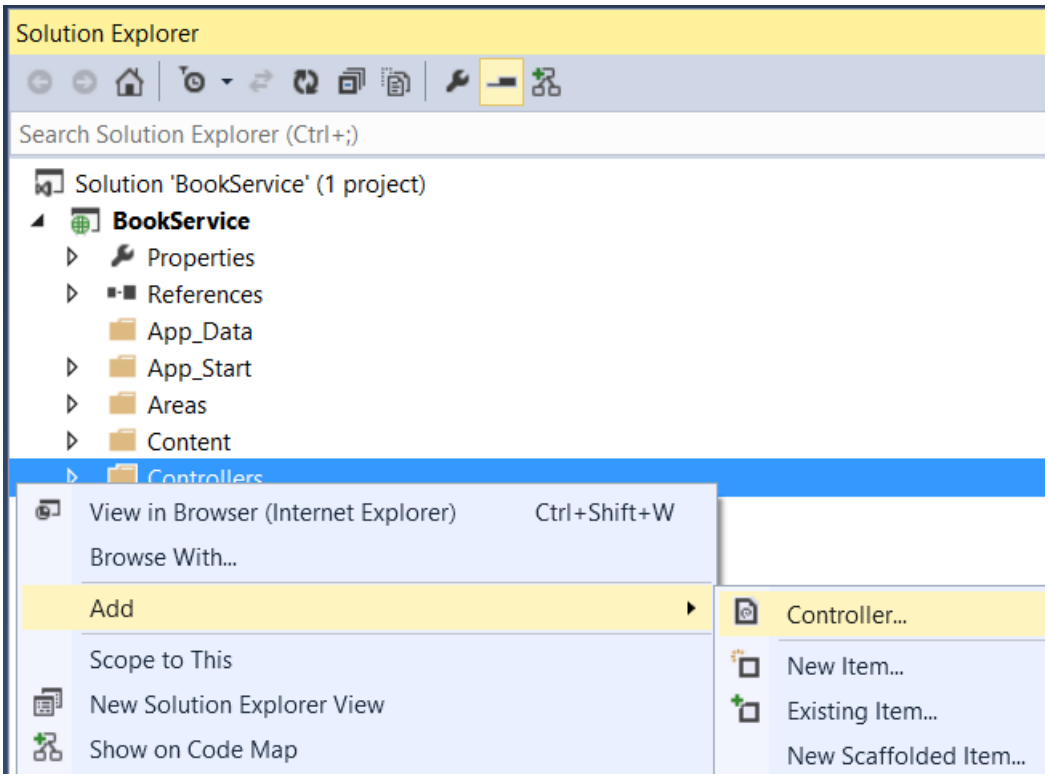
In this section, we'll add Web API controllers that support CRUD operations (create, read, update, and delete). The controllers will use Entity Framework to communicate with the database layer.

First, you can delete the file Controllers/ValuesController.cs. This file contains an example Web API controller, but you don't need it for this tutorial.

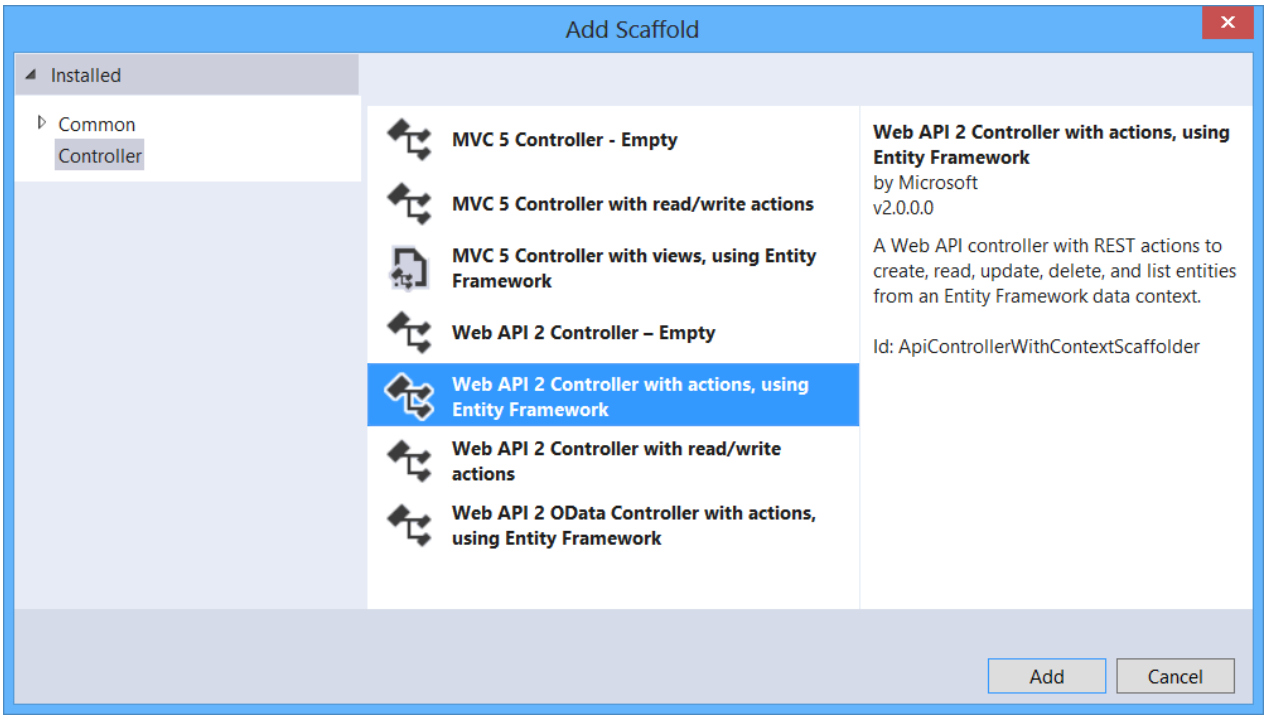


Next, build the project. The Web API scaffolding uses reflection to find the model classes, so it needs the compiled assembly.

In Solution Explorer, right-click the Controllers folder. Select **Add**, then select **Controller**.

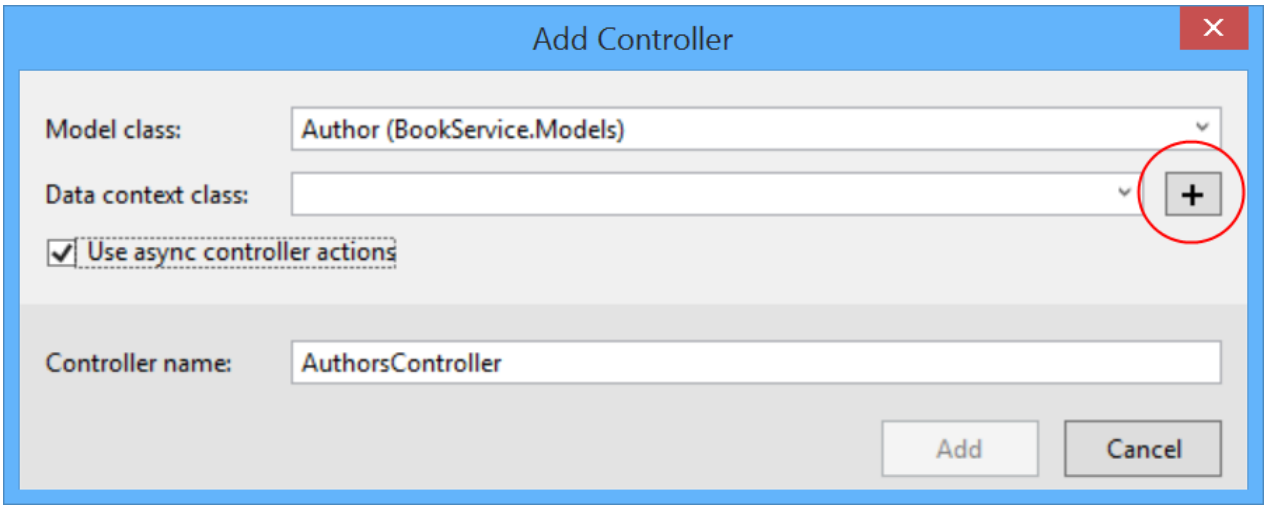


In the **Add Scaffold** dialog, select "Web API 2 Controller with actions, using Entity Framework". Click **Add**.

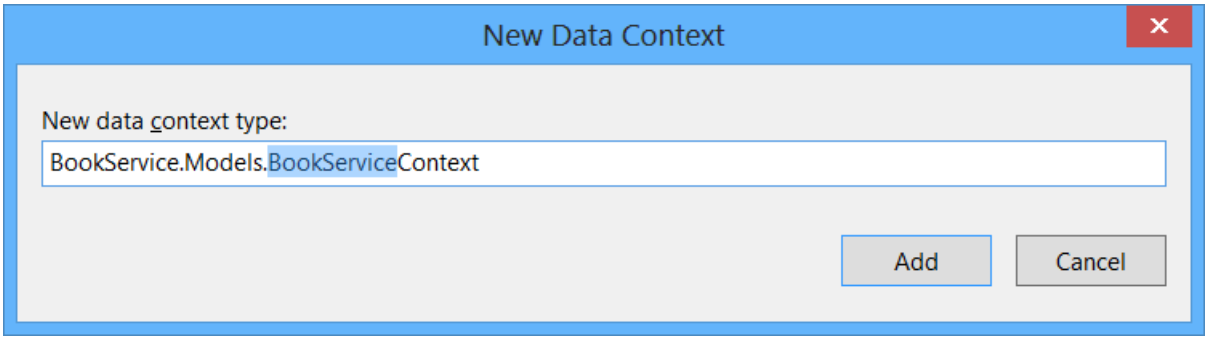


In the **Add Controller** dialog, do the following:

1. In the **Model class** dropdown, select the `Author` class. (If you don't see it listed in the dropdown, make sure that you built the project.)
2. Check "Use async controller actions".
3. Leave the controller name as "AuthorsController".
4. Click plus (+) button next to **Data Context Class**.

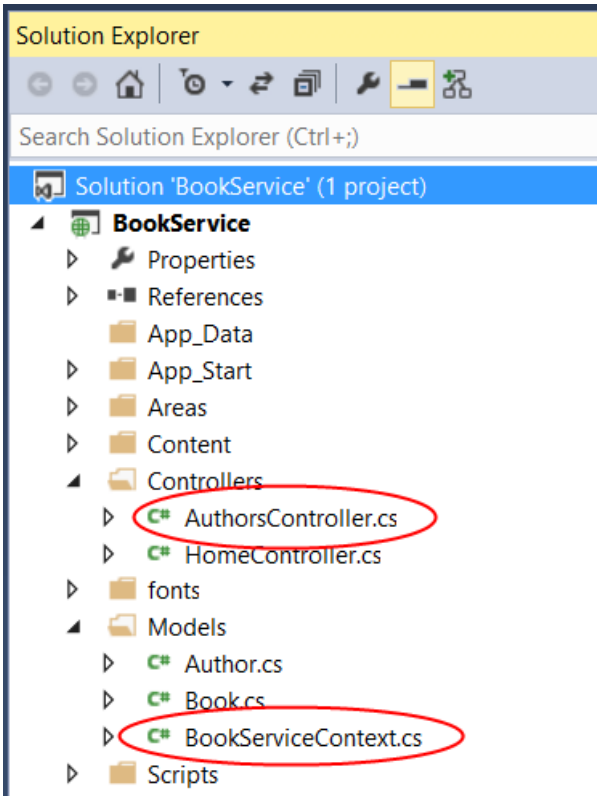


In the **New Data Context** dialog, leave the default name and click **Add**.

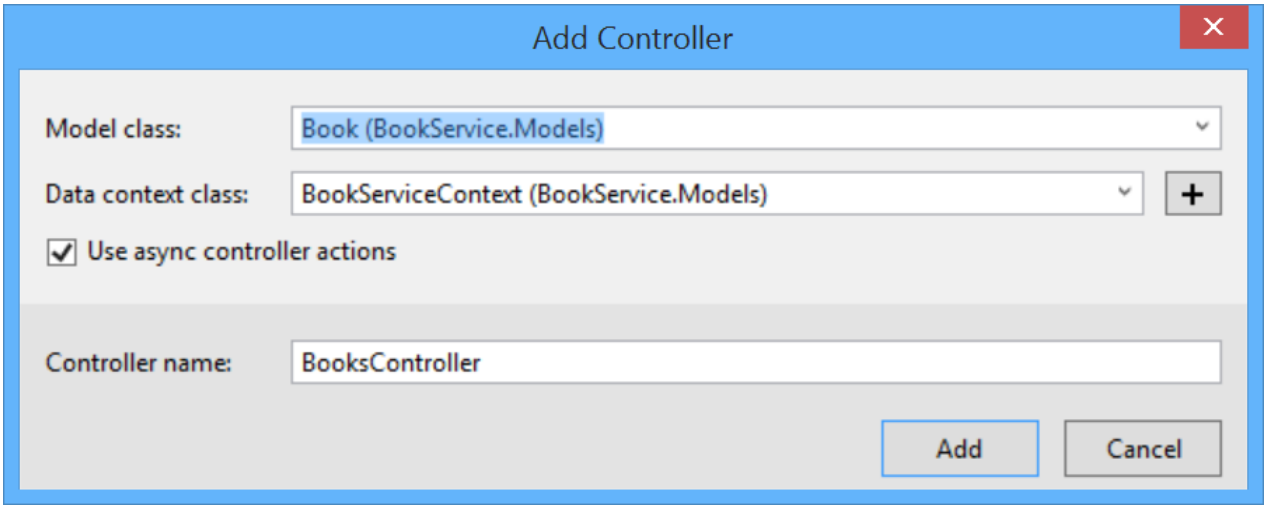


Click **Add** to complete the **Add Controller** dialog. The dialog adds two classes to your project:

- `AuthorsController` defines a Web API controller. The controller implements the REST API that clients use to perform CRUD operations on the list of authors.
- `BookServiceContext` manages entity objects during run time, which includes populating objects with data from a database, change tracking, and persisting data to the database. It inherits from `DbContext`.



At this point, build the project again. Now go through the same steps to add an API controller for `Book` entities. This time, select `Book` for the model class, and select the existing `BookServiceContext` class for the data context class. (Don't create a new data context.) Click **Add** to add the controller.



PREVIOUS NEXT

Use Code First Migrations to Seed the Database

2/20/2020 • 3 minutes to read • Edit Online

by Mike Wasson

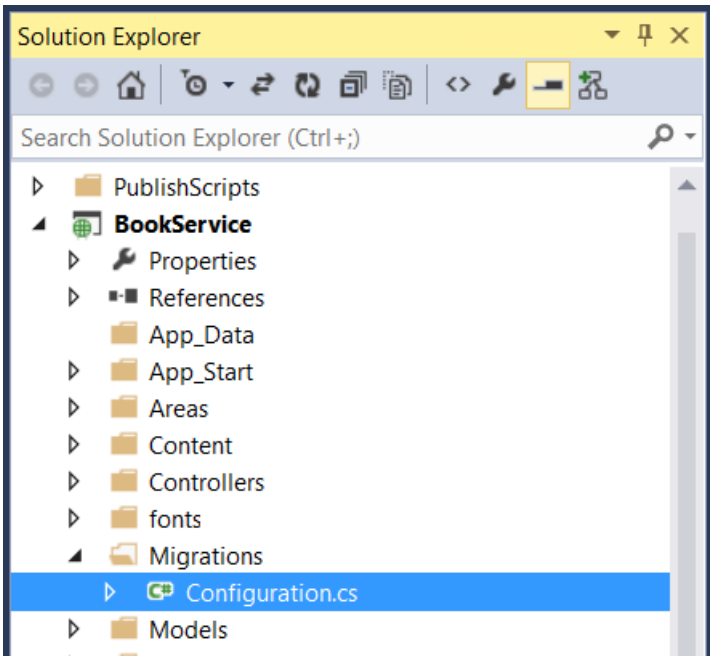
Download Completed Project

In this section, you will use Code First Migrations in EF to seed the database with test data.

From the **Tools** menu, select **NuGet Package Manager**, then select **Package Manager Console**. In the Package Manager Console window, enter the following command:

```
Enable-Migrations
```

This command adds a folder named Migrations to your project, plus a code file named Configuration.cs in the Migrations folder.



Open the Configuration.cs file. Add the following **using** statement.

```
using BookService.Models;
```

Then add the following code to the **Configuration.Seed** method:

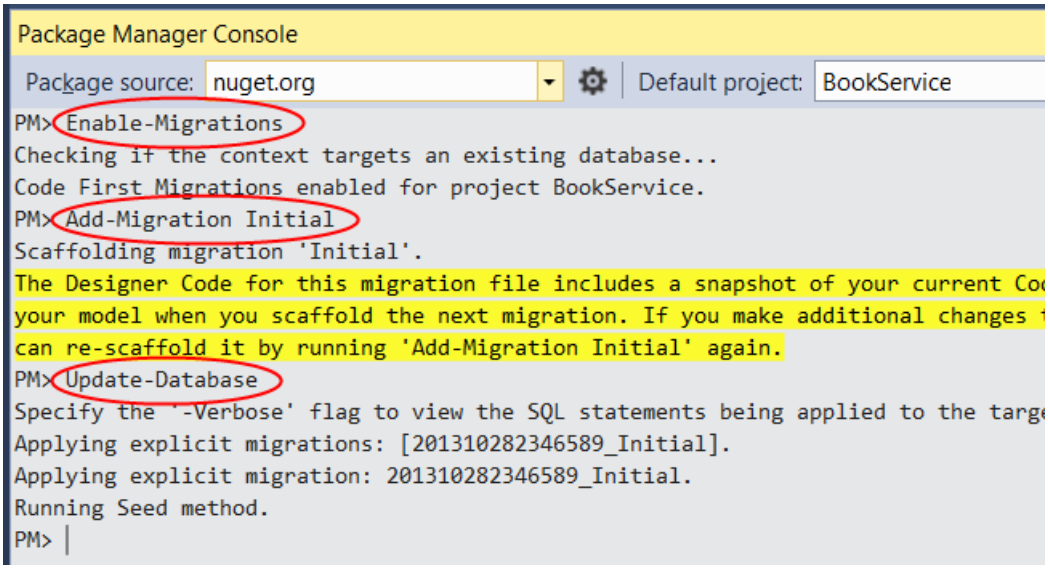
```
protected override void Seed(BookService.Models.BookServiceContext context)
{
    context.Authors.AddOrUpdate(x => x.Id,
        new Author() { Id = 1, Name = "Jane Austen" },
        new Author() { Id = 2, Name = "Charles Dickens" },
        new Author() { Id = 3, Name = "Miguel de Cervantes" }
        );

    context.Books.AddOrUpdate(x => x.Id,
        new Book() { Id = 1, Title = "Pride and Prejudice", Year = 1813, AuthorId = 1,
            Price = 9.99M, Genre = "Comedy of manners" },
        new Book() { Id = 2, Title = "Northanger Abbey", Year = 1817, AuthorId = 1,
            Price = 12.95M, Genre = "Gothic parody" },
        new Book() { Id = 3, Title = "David Copperfield", Year = 1850, AuthorId = 2,
            Price = 15, Genre = "Bildungsroman" },
        new Book() { Id = 4, Title = "Don Quixote", Year = 1617, AuthorId = 3,
            Price = 8.95M, Genre = "Picaresque" }
        );
}
```

In the Package Manager Console window, type the following commands:

```
Add-Migration Initial
Update-Database
```

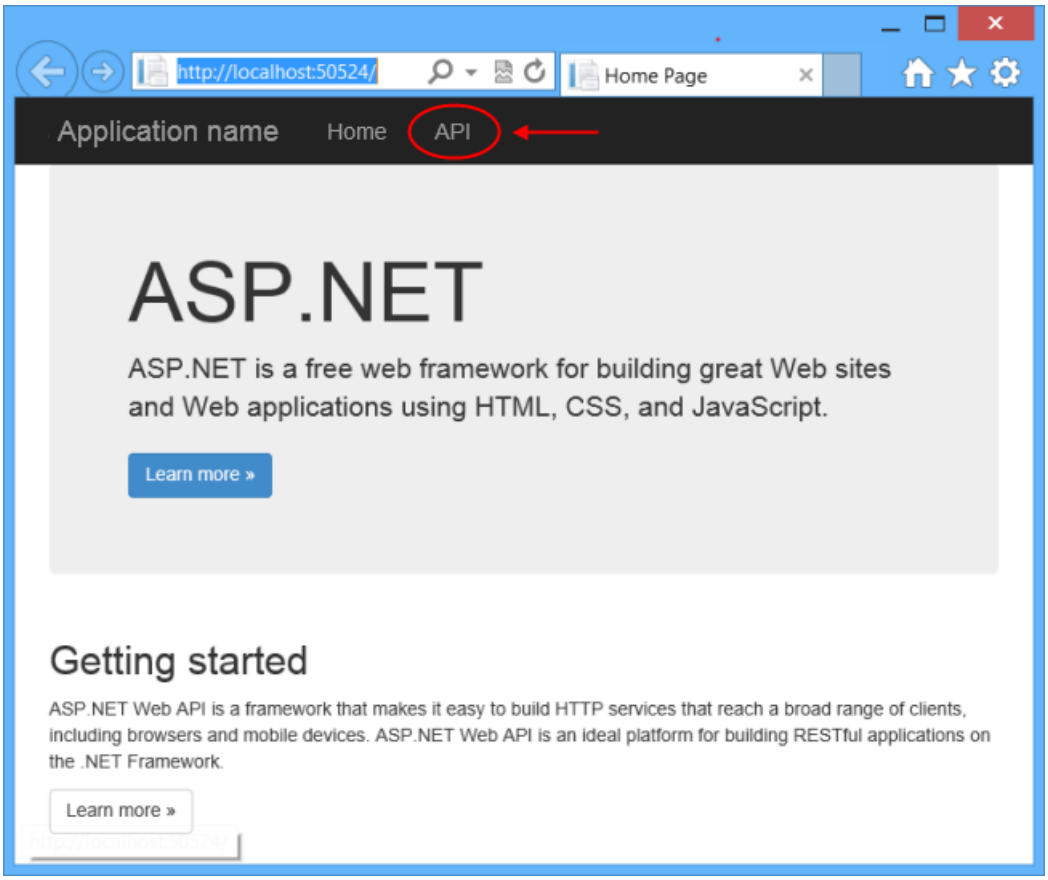
The first command generates code that creates the database, and the second command executes that code. The database is created locally, using LocalDB.



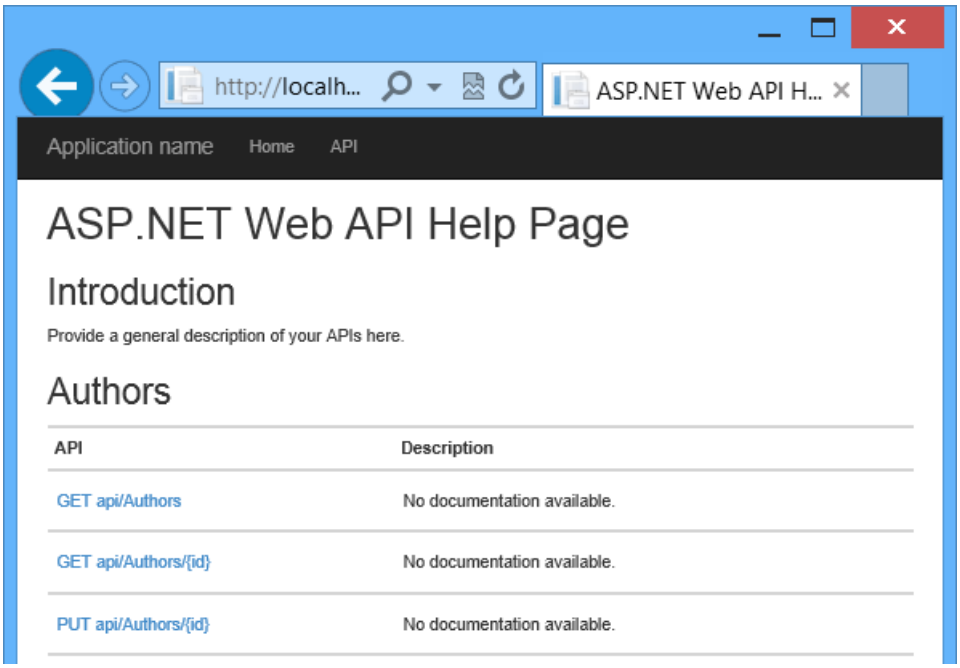
Explore the API (Optional)

Press F5 to run the application in debug mode. Visual Studio starts IIS Express and runs your web app. Visual Studio then launches a browser and opens the app's home page.

When Visual Studio runs a web project, it assigns a port number. In the image below, the port number is 50524. When you run the application, you'll see a different port number.



The home page is implemented using ASP.NET MVC. At the top of the page, there is a link that says "API". This link brings you to an auto-generated help page for the web API. (To learn how this help page is generated, and how you can add your own documentation to the page, see Creating Help Pages for ASP.NET Web API.) You can click on the help page links to see details about the API, including the request and response format.



The API enables CRUD operations on the database. The following summarizes the API.

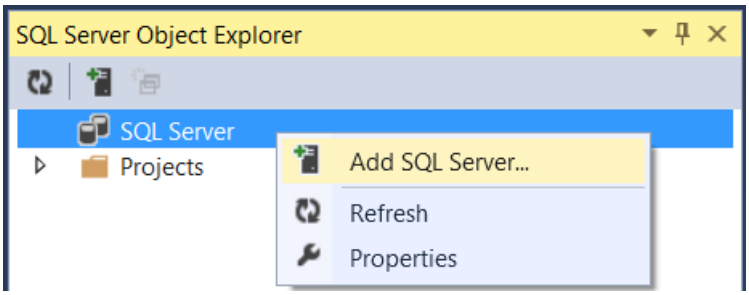
AUTHORS	
GET api/authors	Get all authors.
GET api/authors/{id}	Get an author by ID.

AUTHORS	
POST /api/authors	Create a new author.
PUT /api/authors/{id}	Update an existing author.
DELETE /api/authors/{id}	Delete an author.

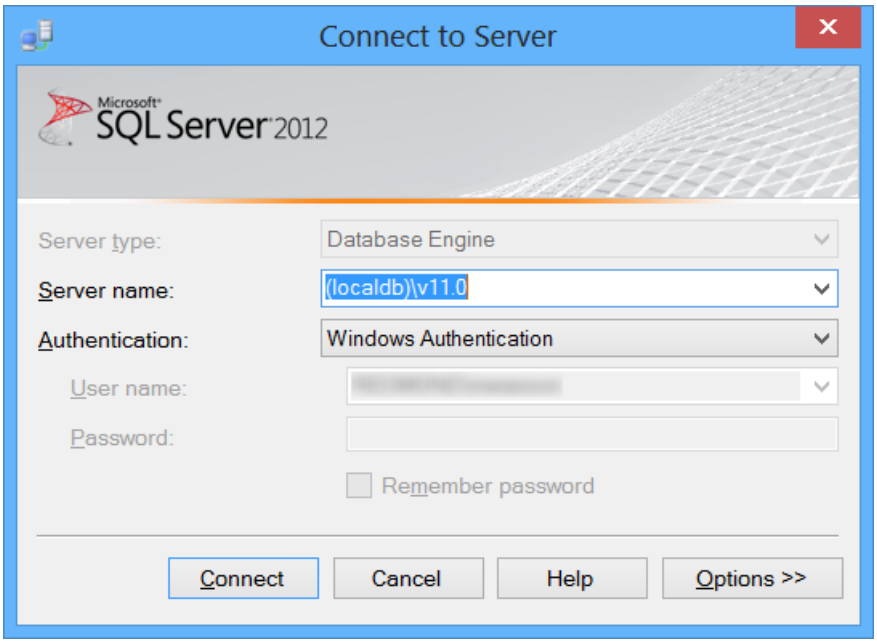
BOOKS	
GET /api/books	Get all books.
GET /api/books/{id}	Get a book by ID.
POST /api/books	Create a new book.
PUT /api/books/{id}	Update an existing book.
DELETE /api/books/{id}	Delete a book.

View the Database (Optional)

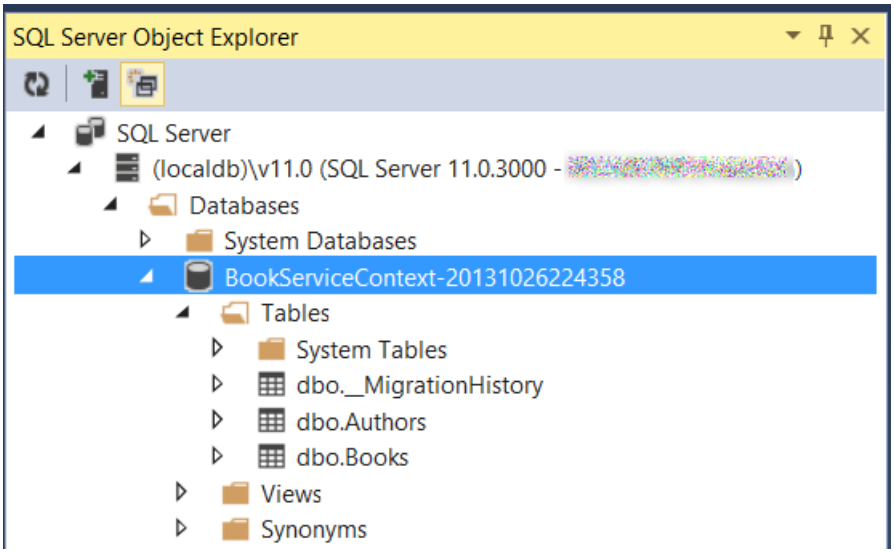
When you ran the Update-Database command, EF created the database and called the `Seed` method. When you run the application locally, EF uses LocalDB. You can view the database in Visual Studio. From the **View** menu, select **SQL Server Object Explorer**.



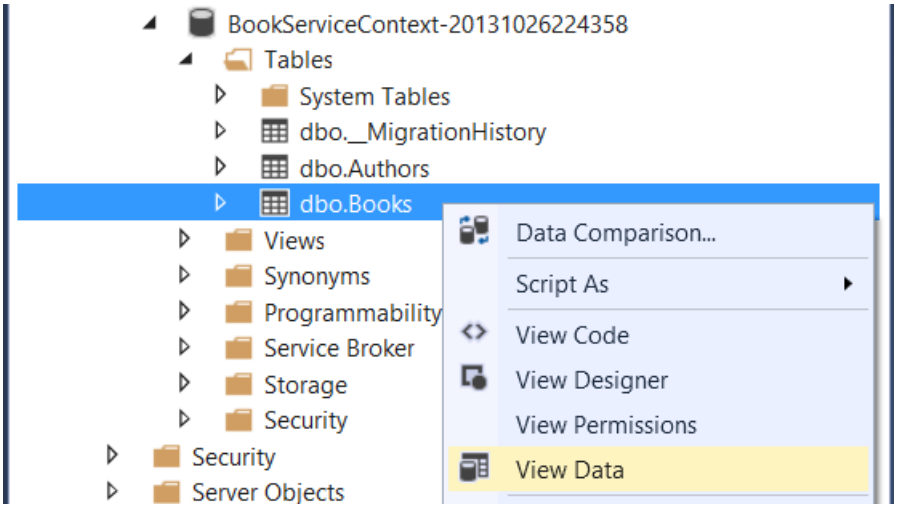
In the **Connect to Server** dialog, in the **Server Name** edit box, type "(localdb)\v11.0". Leave the **Authentication** option as "Windows Authentication". Click **Connect**.



Visual Studio connects to LocalDB and shows your existing databases in the SQL Server Object Explorer window. You can expand the nodes to see the tables that EF created.



To view the data, right-click a table and select **View Data**.



The following screenshot shows the results for the Books table. Notice that EF populated the database with the seed data, and the table contains the foreign key to the Authors table.

SQLQuery1.sql | dbo.Books [Data]

Max Rows: 1000

BookId	Title	Year	AuthorId
1	Pride and Pr...	1813	1
2	Emma	1815	1
3	David Copp...	1850	2
4	Don Quixote	1617	3
NULL	NULL	NULL	NULL

PREVIOUS NEXT

Handling Entity Relations

2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

This section describes some details of how EF loads related entities, and how to handle circular navigation properties in your model classes. (This section provides background knowledge, and is not required to complete the tutorial. If you prefer, skip to Part 5..)

Eager Loading versus Lazy Loading

When using EF with a relational database, it's important to understand how EF loads related data.

It's also useful to see the SQL queries that EF generates. To trace the SQL, add the following line of code to the `BookServiceContext` constructor:

```
public BookServiceContext() : base("name=BookServiceContext")
{
    // New code:
    this.Database.Log = s => System.Diagnostics.Debug.WriteLine(s);
}
```

If you send a GET request to /api/books, it returns JSON like the following:

```
[
  {
    "BookId": 1,
    "Title": "Pride and Prejudice",
    "Year": 1813,
    "Price": 9.99,
    "Genre": "Comedy of manners",
    "AuthorId": 1,
    "Author": null
  },
  ...
```

You can see that the Author property is null, even though the book contains a valid AuthorId. That's because EF is not loading the related Author entities. The trace log of the SQL query confirms this:

```
SELECT
    [Extent1].[BookId] AS [BookId],
    [Extent1].[Title] AS [Title],
    [Extent1].[Year] AS [Year],
    [Extent1].[Price] AS [Price],
    [Extent1].[Genre] AS [Genre],
    [Extent1].[AuthorId] AS [AuthorId]
    FROM [dbo].[Books] AS [Extent1]
```

The SELECT statement takes from the Books table, and does not reference the Author table.

For reference, here is the method in the `BooksController` class that returns the list of books.

```
public IQueryable<Book> GetBooks()
{
    return db.Books;
}
```

Let's see how we can return the Author as part of the JSON data. There are three ways to load related data in Entity Framework: eager loading, lazy loading, and explicit loading. There are trade-offs with each technique, so it's important to understand how they work.

Eager Loading

With *eager loading*, EF loads related entities as part of the initial database query. To perform eager loading, use the **System.Data.Entity.Include** extension method.

```
public IQueryable<Book> GetBooks()
{
    return db.Books
        // new code:
        .Include(b => b.Author);
}
```

This tells EF to include the Author data in the query. If you make this change and run the app, now the JSON data looks like this:

```
[
  {
    "BookId": 1,
    "Title": "Pride and Prejudice",
    "Year": 1813,
    "Price": 9.99,
    "Genre": "Comedy of manners",
    "AuthorId": 1,
    "Author": {
      "AuthorId": 1,
      "Name": "Jane Austen"
    }
  },
  ...
```

The trace log shows that EF performed a join on the Book and Author tables.

```
SELECT
    [Extent1].[BookId] AS [BookId],
    [Extent1].[Title] AS [Title],
    [Extent1].[Year] AS [Year],
    [Extent1].[Price] AS [Price],
    [Extent1].[Genre] AS [Genre],
    [Extent1].[AuthorId] AS [AuthorId],
    [Extent2].[AuthorId] AS [AuthorId1],
    [Extent2].[Name] AS [Name]
    FROM  [dbo].[Books] AS [Extent1]
    INNER JOIN [dbo].[Authors] AS [Extent2] ON [Extent1].[AuthorId] = [Extent2].[AuthorId]
```

Lazy Loading

With lazy loading, EF automatically loads a related entity when the navigation property for that entity is dereferenced. To enable lazy loading, make the navigation property virtual. For example, in the Book class:

```
public class Book
{
    // (Other properties)

    // Virtual navigation property
    public virtual Author Author { get; set; }
}
```

Now consider the following code:

```
var books = db.Books.ToList();  // Does not load authors
var author = books[0].Author;   // Loads the author for books[0]
```

When lazy loading is enabled, accessing the `Author` property on `books[0]` causes EF to query the database for the author.

Lazy loading requires multiple database trips, because EF sends a query each time it retrieves a related entity. Generally, you want lazy loading disabled for objects that you serialize. The serializer has to read all of the properties on the model, which triggers loading the related entities. For example, here are the SQL queries when EF serializes the list of books with lazy loading enabled. You can see that EF makes three separate queries for the three authors.

```
SELECT
    [Extent1].[BookId] AS [BookId],
    [Extent1].[Title] AS [Title],
    [Extent1].[Year] AS [Year],
    [Extent1].[Price] AS [Price],
    [Extent1].[Genre] AS [Genre],
    [Extent1].[AuthorId] AS [AuthorId]
    FROM [dbo].[Books] AS [Extent1]

SELECT
    [Extent1].[AuthorId] AS [AuthorId],
    [Extent1].[Name] AS [Name]
    FROM [dbo].[Authors] AS [Extent1]
    WHERE [Extent1].[AuthorId] = @EntityKeyValue1

SELECT
    [Extent1].[AuthorId] AS [AuthorId],
    [Extent1].[Name] AS [Name]
    FROM [dbo].[Authors] AS [Extent1]
    WHERE [Extent1].[AuthorId] = @EntityKeyValue1

SELECT
    [Extent1].[AuthorId] AS [AuthorId],
    [Extent1].[Name] AS [Name]
    FROM [dbo].[Authors] AS [Extent1]
    WHERE [Extent1].[AuthorId] = @EntityKeyValue1
```

There are still times when you might want to use lazy loading. Eager loading can cause EF to generate a very complex join. Or you might need related entities for a small subset of the data, and lazy loading would be more efficient.

One way to avoid serialization problems is to serialize data transfer objects (DTOs) instead of entity objects. I'll show this approach later in the article.

Explicit Loading

Explicit loading is similar to lazy loading, except that you explicitly get the related data in code; it doesn't happen automatically when you access a navigation property. Explicit loading gives you more control over when to load

related data, but requires extra code. For more information about explicit loading, see Loading Related Entities.

Navigation Properties and Circular References

When I defined the Book and Author models, I defined a navigation property on the `Book` class for the Book-Author relationship, but I did not define a navigation property in the other direction.

What happens if you add the corresponding navigation property to the `Author` class?

```
public class Author
{
    public int AuthorId { get; set; }
    [Required]
    public string Name { get; set; }

    public ICollection<Book> Books { get; set; }
}
```

Unfortunately, this creates a problem when you serialize the models. If you load the related data, it creates a circular object graph.

When the JSON or XML formatter tries to serialize the graph, it will throw an exception. The two formatters throw different exception messages. Here is an example for the JSON formatter:

```
{
  "Message": "An error has occurred.",
  "ExceptionMessage": "The 'ObjectContent`1' type failed to serialize the response body for content type
      'application/json; charset=utf-8'.",
  "ExceptionType": "System.InvalidOperationException",
  "StackTrace": null,
  "InnerException": {
    "Message": "An error has occurred.",
    "ExceptionMessage": "Self referencing loop detected with type 'BookService.Models.Book'.
        Path '[0].Author.Books'.",
    "ExceptionType": "Newtonsoft.Json.JsonSerializationException",
    "StackTrace": "..."
     }
}
```

Here is the XML formatter:

```
<Error>
  <Message>An error has occurred.</Message>
  <ExceptionMessage>The 'ObjectContent`1' type failed to serialize the response body for content type
    'application/xml; charset=utf-8'.</ExceptionMessage>
  <ExceptionType>System.InvalidOperationException</ExceptionType>
  <StackTrace />
  <InnerException>
    <Message>An error has occurred.</Message>
    <ExceptionMessage>Object graph for type 'BookService.Models.Author' contains cycles and cannot be
      serialized if reference tracking is disabled.</ExceptionMessage>
    <ExceptionType>System.Runtime.Serialization.SerializationException</ExceptionType>
    <StackTrace> ... </StackTrace>
  </InnerException>
</Error>
```

One solution is to use DTOs, which I describe in the next section. Alternatively, you can configure the JSON and XML formatters to handle graph cycles. For more information, see Handling Circular Object References.

For this tutorial, you don't need the `Author.Book` navigation property, so you can leave it out.

PREVIOUS NEXT

Create Data Transfer Objects (DTOs)

2/20/2020 • 2 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

Right now, our web API exposes the database entities to the client. The client receives data that maps directly to your database tables. However, that's not always a good idea. Sometimes you want to change the shape of the data that you send to client. For example, you might want to:

- Remove circular references (see previous section).
- Hide particular properties that clients are not supposed to view.
- Omit some properties in order to reduce payload size.
- Flatten object graphs that contain nested objects, to make them more convenient for clients.
- Avoid "over-posting" vulnerabilities. (See Model Validation for a discussion of over-posting.)
- Decouple your service layer from your database layer.

To accomplish this, you can define a *data transfer object* (DTO). A DTO is an object that defines how the data will be sent over the network. Let's see how that works with the Book entity. In the Models folder, add two DTO classes:

```
namespace BookService.Models
{
    public class BookDto
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string AuthorName { get; set; }
    }
}

namespace BookService.Models
{
    public class BookDetailDto
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public int Year { get; set; }
        public decimal Price { get; set; }
        public string AuthorName { get; set; }
        public string Genre { get; set; }
    }
}
```

The `BookDetailDto` class includes all of the properties from the Book model, except that `AuthorName` is a string that will hold the author name. The `BookDto` class contains a subset of properties from `BookDetailDto`.

Next, replace the two GET methods in the `BooksController` class, with versions that return DTOs. We'll use the LINQ **Select** statement to convert from Book entities into DTOs.

```
// GET api/Books
public IQueryable<BookDto> GetBooks()
{
    var books = from b in db.Books
                select new BookDto()
                {
                    Id = b.Id,
                    Title = b.Title,
                    AuthorName = b.Author.Name
                };

    return books;
}

// GET api/Books/5
[ResponseType(typeof(BookDetailDto))]
public async Task<IHttpActionResult> GetBook(int id)
{
    var book = await db.Books.Include(b => b.Author).Select(b =>
        new BookDetailDto()
        {
            Id = b.Id,
            Title = b.Title,
            Year = b.Year,
            Price = b.Price,
            AuthorName = b.Author.Name,
            Genre = b.Genre
        }).SingleOrDefaultAsync(b => b.Id == id);
    if (book == null)
    {
        return NotFound();
    }

    return Ok(book);
}
```

Here is the SQL generated by the new `GetBooks` method. You can see that EF translates the LINQ **Select** into a SQL SELECT statement.

```
SELECT
    [Extent1].[Id] AS [Id],
    [Extent1].[Title] AS [Title],
    [Extent2].[Name] AS [Name]
    FROM  [dbo].[Books] AS [Extent1]
    INNER JOIN [dbo].[Authors] AS [Extent2] ON [Extent1].[AuthorId] = [Extent2].[Id]
```

Finally, modify the `PostBook` method to return a DTO.

```
[ResponseType(typeof(BookDto))]
public async Task<IHttpActionResult> PostBook(Book book)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    db.Books.Add(book);
    await db.SaveChangesAsync();

    // New code:
    // Load author name
    db.Entry(book).Reference(x => x.Author).Load();

    var dto = new BookDto()
    {
        Id = book.Id,
        Title = book.Title,
        AuthorName = book.Author.Name
    };

    return CreatedAtRoute("DefaultApi", new { id = book.Id }, dto);
}
```

NOTE

In this tutorial, we're converting to DTOs manually in code. Another option is to use a library like AutoMapper that handles the conversion automatically.

PREVIOUS NEXT

Create the JavaScript Client

2/20/2020 • 2 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

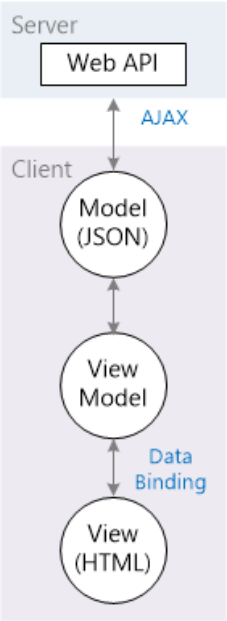
In this section, you will create the client for the application, using HTML, JavaScript, and the Knockout.js library. We'll build the client app in stages:

- Showing a list of books.
- Showing a book detail.
- Adding a new book.

The Knockout library uses the Model-View-ViewModel (MVVM) pattern:

- The **model** is the server-side representation of the data in the business domain (in our case, books and authors).
- The **view** is the presentation layer (HTML).
- The **view model** is a JavaScript object that holds the models. The view model is a code abstraction of the UI. It has no knowledge of the HTML representation. Instead, it represents abstract features of the view, such as "a list of books".

The view is data-bound to the view model. Updates to the view model are automatically reflected in the view. The view model also gets events from the view, such as button clicks.



This approach makes it easy to change the layout and UI of your app, because you can change the bindings, without rewriting any code. For example, you might show a list of items as a `<ul>`, then change it later to a table.

Add the Knockout Library

In Visual Studio, from the **Tools** menu, select **NuGet Package Manager**. Then select **Package Manager Console**. In the Package Manager Console window, enter the following command:

```
Install-Package knockoutjs
```

This command adds the Knockout files to the Scripts folder.

Create the View Model

Add a JavaScript file named app.js to the Scripts folder. (In Solution Explorer, right-click the Scripts folder, select **Add**, then select **JavaScript File**.) Paste in the following code:

```
var ViewModel = function () {
    var self = this;
    self.books = ko.observableArray();
    self.error = ko.observable();

    var booksUri = '/api/books/';

    function ajaxHelper(uri, method, data) {
        self.error(''); // Clear error message
        return $.ajax({
            type: method,
            url: uri,
            dataType: 'json',
            contentType: 'application/json',
            data: data ? JSON.stringify(data) : null
        }).fail(function (jqXHR, textStatus, errorThrown) {
            self.error(errorThrown);
        });
    }

    function getAllBooks() {
        ajaxHelper(booksUri, 'GET').done(function (data) {
            self.books(data);
        });
    }

    // Fetch the initial data.
    getAllBooks();
};

ko.applyBindings(new ViewModel());
```

In Knockout, the `observable` class enables data-binding. When the contents of an observable change, the observable notifies all of the data-bound controls, so they can update themselves. (The `observableArray` class is the array version of *observable*.) To start with, our view model has two observables:

- `books` holds the list of books.
- `error` contains an error message if an AJAX call fails.

The `getAllBooks` method makes an AJAX call to get the list of books. Then it pushes the result onto the `books` array.

The `ko.applyBindings` method is part of the Knockout library. It takes the view model as a parameter and sets up the data binding.

Add a Script Bundle

Bundling is a feature in ASP.NET 4.5 that makes it easy to combine or bundle multiple files into a single file. Bundling reduces the number of requests to the server, which can improve page load time.

Open the file App_Start/BundleConfig.cs. Add the following code to the RegisterBundles method.

```
public static void RegisterBundles(BundleCollection bundles)
{
    // ...

    // New code:
    bundles.Add(new ScriptBundle("~/bundles/app").Include(
              "~/Scripts/knockout-{version}.js",
              "~/Scripts/app.js"));
}
```

PREVIOUS NEXT

Create the View (UI)

2/20/2020 • 2 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

In this section, you will start to define the HTML for the app, and add data binding between the HTML and the view model.

Open the file Views/Home/Index.cshtml. Replace the entire contents of that file with the following.

```
@section scripts {
  @Scripts.Render("~/bundles/app")
}

<div class="page-header">
  <h1>BookService</h1>
</div>

<div class="row">

  <div class="col-md-4">
    <div class="panel panel-default">
      <div class="panel-heading">
        <h2 class="panel-title">Books</h2>
      </div>
      <div class="panel-body">
        <ul class="list-unstyled" data-bind="foreach: books">
          <li>
            <strong><span data-bind="text: AuthorName"></span></strong>: <span data-bind="text: Title">
</span>
            <small><a href="#">Details</a></small>
          </li>
        </ul>
      </div>
    </div>
    <div class="alert alert-danger" data-bind="visible: error"><p data-bind="text: error"></p></div>
  </div>

  <div class="col-md-4">
    <!-- TODO: Book details -->
  </div>

  <div class="col-md-4">
    <!-- TODO: Add new book -->
  </div>
</div>
```

Most of the `div` elements are there for Bootstrap styling. The important elements are the ones with `data-bind` attributes. This attribute links the HTML to the view model.

For example:

```
<p data-bind="text: error">
```

In this example, the "`text`" binding causes the `<p>` element to show the value of the `error` property from the view model. Recall that `error` was declared as a `ko.observable`:

```
self.error = ko.observable();
```

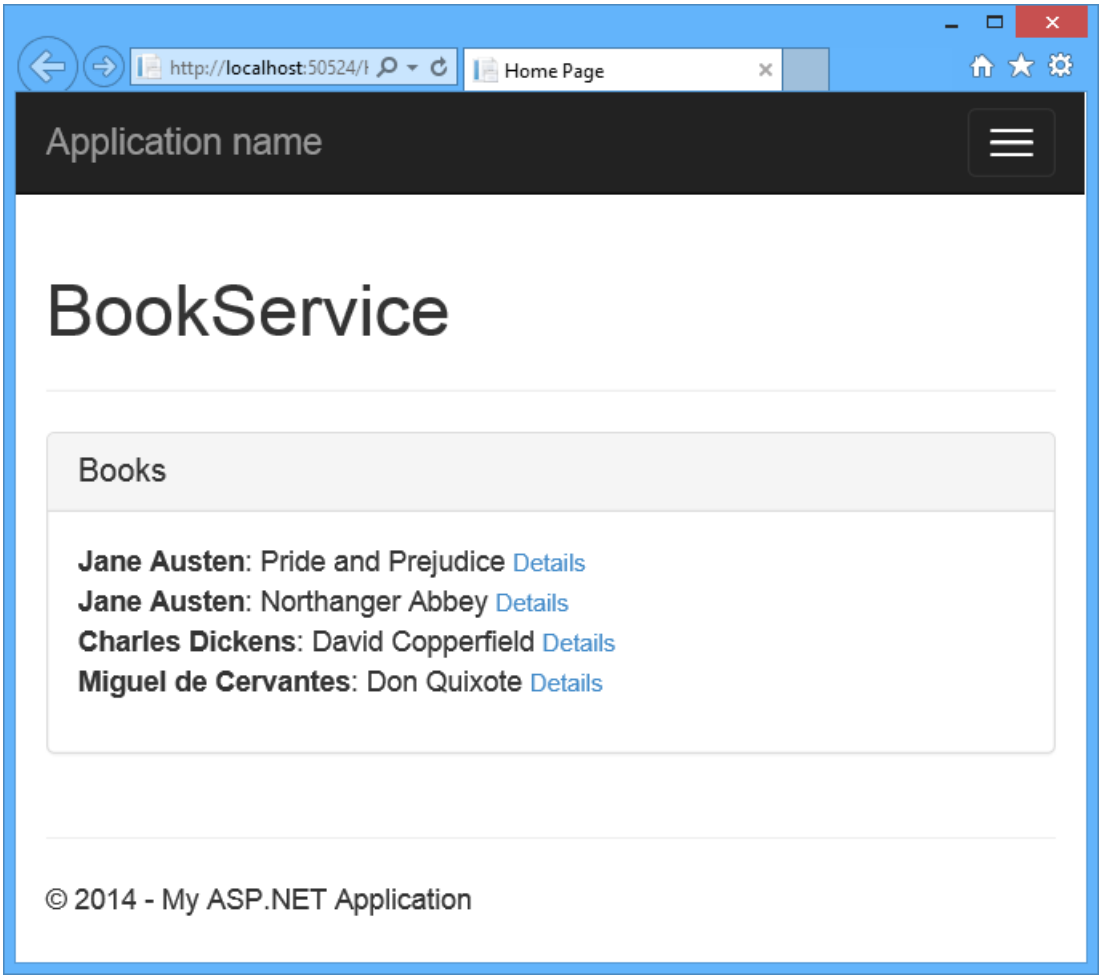
Whenever a new value is assigned to `error`, Knockout updates the text in the `<p>` element.

The `foreach` binding tells Knockout to loop through the contents of the `books` array. For each item in the array, Knockout creates a new <li> element. Bindings inside the context of the `foreach` refer to properties on the array item. For example:

```
<span data-bind="text: Author"></span>
```

Here the `text` binding reads the Author property of each book.

If you run the application now, it should look like this:



The list of books loads asynchronously, after the page loads. Right now, the "Details" links are not functional. We'll add this functionality in the next section.

Display Item Details

2/20/2020 • 2 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

In this section, you will add the ability to view details for each book. In app.js, add to the following code to the view model:

```
self.detail = ko.observable();

self.getBookDetail = function (item) {
    ajaxHelper(booksUri + item.Id, 'GET').done(function (data) {
        self.detail(data);
    });
}
```

In Views/Home/Index.cshtml, add a data-bind element to the Details link:

```
<ul class="list-unstyled" data-bind="foreach: books">
  <li>
    <strong><span data-bind="text: AuthorName"></span></strong>: <span data-bind="text: Title"></span>
    <!-- New code -->
    <small><a href="#" data-bind="click: $parent.getBookDetail">Details</a></small>
  </li>
</ul>
```

This binds the click handler for the <a> element to the `getBookDetail` function on the view model.

In the same file, replace the following mark-up:

```
<div class="col-md-4">
    <!-- TODO: Book details -->
</div>
```

with this:

```
<!-- ko if:detail() -->

<div class="col-md-4">
<div class="panel panel-default">
  <div class="panel-heading">
    <h2 class="panel-title">Detail</h2>
  </div>
  <table class="table">
    <tr><td>Author</td><td data-bind="text: detail().AuthorName"></td></tr>
    <tr><td>Title</td><td data-bind="text: detail().Title"></td></tr>
    <tr><td>Year</td><td data-bind="text: detail().Year"></td></tr>
    <tr><td>Genre</td><td data-bind="text: detail().Genre"></td></tr>
    <tr><td>Price</td><td data-bind="text: detail().Price"></td></tr>
  </table>
</div>
</div>

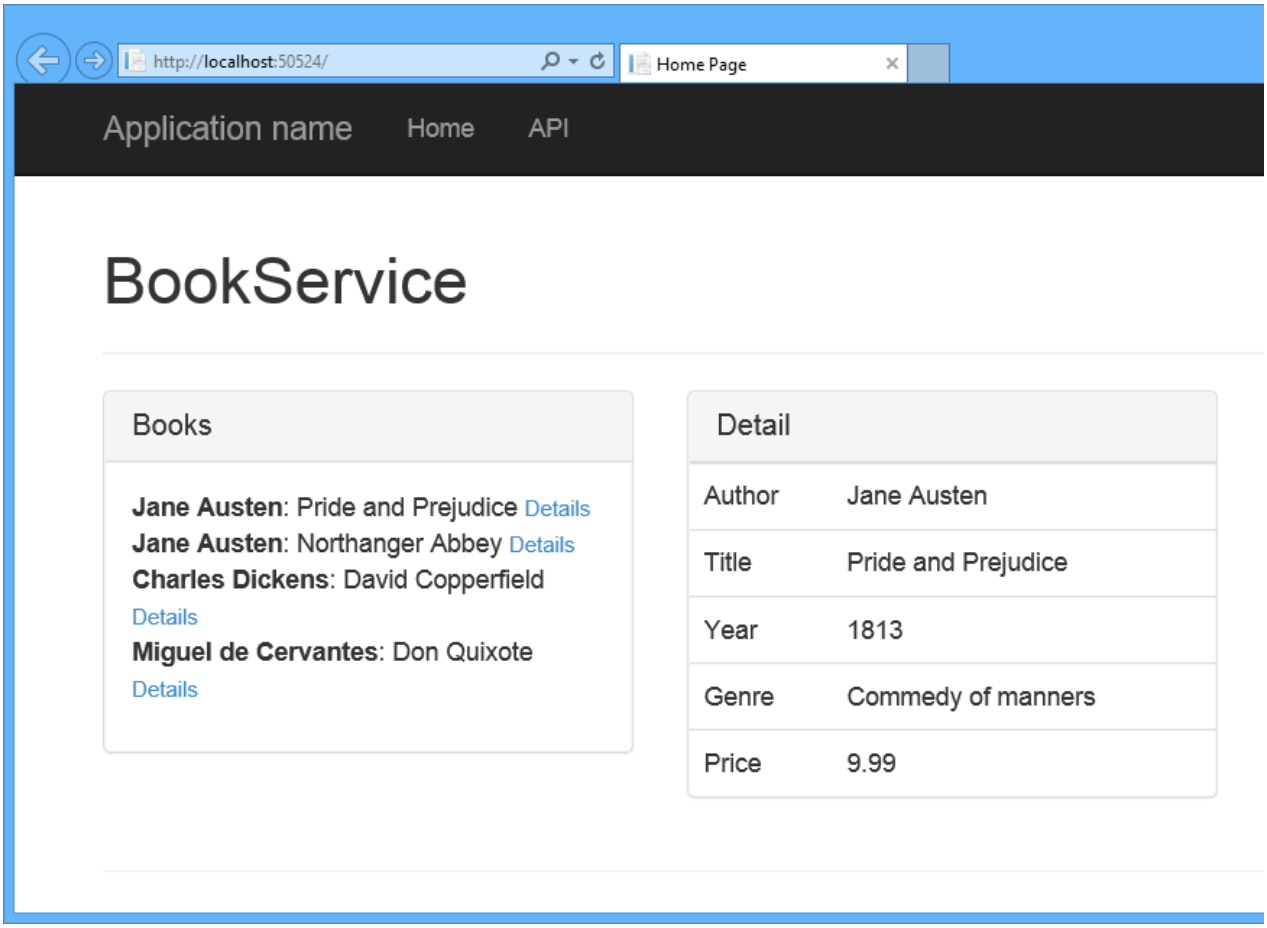
<!-- /ko -->
```

This markup creates a table that is data-bound to the properties of the `detail` observable in the view model.

The "<!-- ko -->" syntax lets you include a Knockout binding outside of a DOM element. In this case, the `if` binding causes this section of markup to be displayed only when `details` is non-null.

```
<!-- ko if:detail() -->
```

Now if you run the app and click one of the "Detail" links, the app will display the book details.



Add a New Item to the Database

2/20/2020 • 2 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

In this section, you will add the ability for users to create a new book. In app.js, add the following code to the view model:

```
self.authors = ko.observableArray();
self.newBook = {
    Author: ko.observable(),
    Genre: ko.observable(),
    Price: ko.observable(),
    Title: ko.observable(),
    Year: ko.observable()
}

var authorsUri = '/api/authors/';

function getAuthors() {
    ajaxHelper(authorsUri, 'GET').done(function (data) {
        self.authors(data);
    });
}

self.addBook = function (formElement) {
    var book = {
        AuthorId: self.newBook.Author().Id,
        Genre: self.newBook.Genre(),
        Price: self.newBook.Price(),
        Title: self.newBook.Title(),
        Year: self.newBook.Year()
    };

    ajaxHelper(booksUri, 'POST', book).done(function (item) {
        self.books.push(item);
    });
}

getAuthors();
```

In Index.cshtml, replace the following markup:

```
<div class="col-md-4">
    <!-- TODO: Add new book -->
</div>
```

With:

```
<div class="col-md-4">
<div class="panel panel-default">
  <div class="panel-heading">
    <h2 class="panel-title">Add Book</h2>
  </div>

  <div class="panel-body">
    <form class="form-horizontal" data-bind="submit: addBook">
      <div class="form-group">
        <label for="inputAuthor" class="col-sm-2 control-label">Author</label>
        <div class="col-sm-10">
          <select data-bind="options:authors, optionsText: 'Name', value: newBook.Author"></select>
        </div>
      </div>

      <div class="form-group" data-bind="with: newBook">
        <label for="inputTitle" class="col-sm-2 control-label">Title</label>
        <div class="col-sm-10">
          <input type="text" class="form-control" id="inputTitle" data-bind="value:Title"/>
        </div>

        <label for="inputYear" class="col-sm-2 control-label">Year</label>
        <div class="col-sm-10">
          <input type="number" class="form-control" id="inputYear" data-bind="value:Year"/>
        </div>

        <label for="inputGenre" class="col-sm-2 control-label">Genre</label>
        <div class="col-sm-10">
          <input type="text" class="form-control" id="inputGenre" data-bind="value:Genre"/>
        </div>

        <label for="inputPrice" class="col-sm-2 control-label">Price</label>
        <div class="col-sm-10">
          <input type="number" step="any" class="form-control" id="inputPrice" data-bind="value:Price"/>
        </div>
      </div>
      <button type="submit" class="btn btn-default">Submit</button>
    </form>
  </div>
</div>
</div>
```

This markup creates a form for submitting a new author. The values for the author drop-down list are data-bound to the `authors` observable in the view model. For the other form inputs, the values are data-bound to the `newBook` property of the view model.

The submit handler on the form is bound to the `addBook` function:

```
<form class="form-horizontal" data-bind="submit: addBook">
```

The `addBook` function reads the current values of the data-bound form inputs to create a JSON object. Then it POSTs the JSON object to `/api/books`.

PREVIOUS NEXT

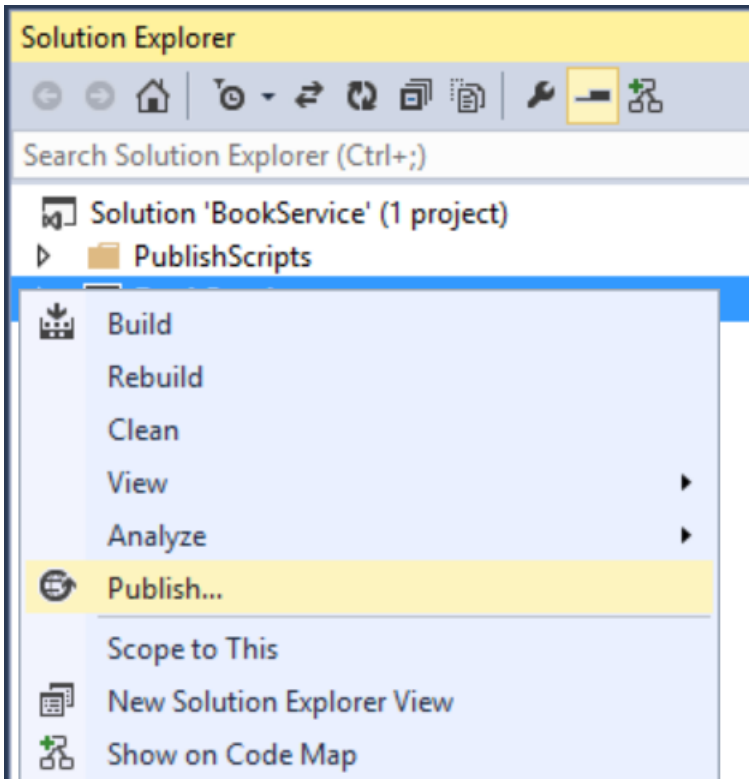
Publish the App to Azure Azure App Service

2/20/2020 • 2 minutes to read • Edit Online

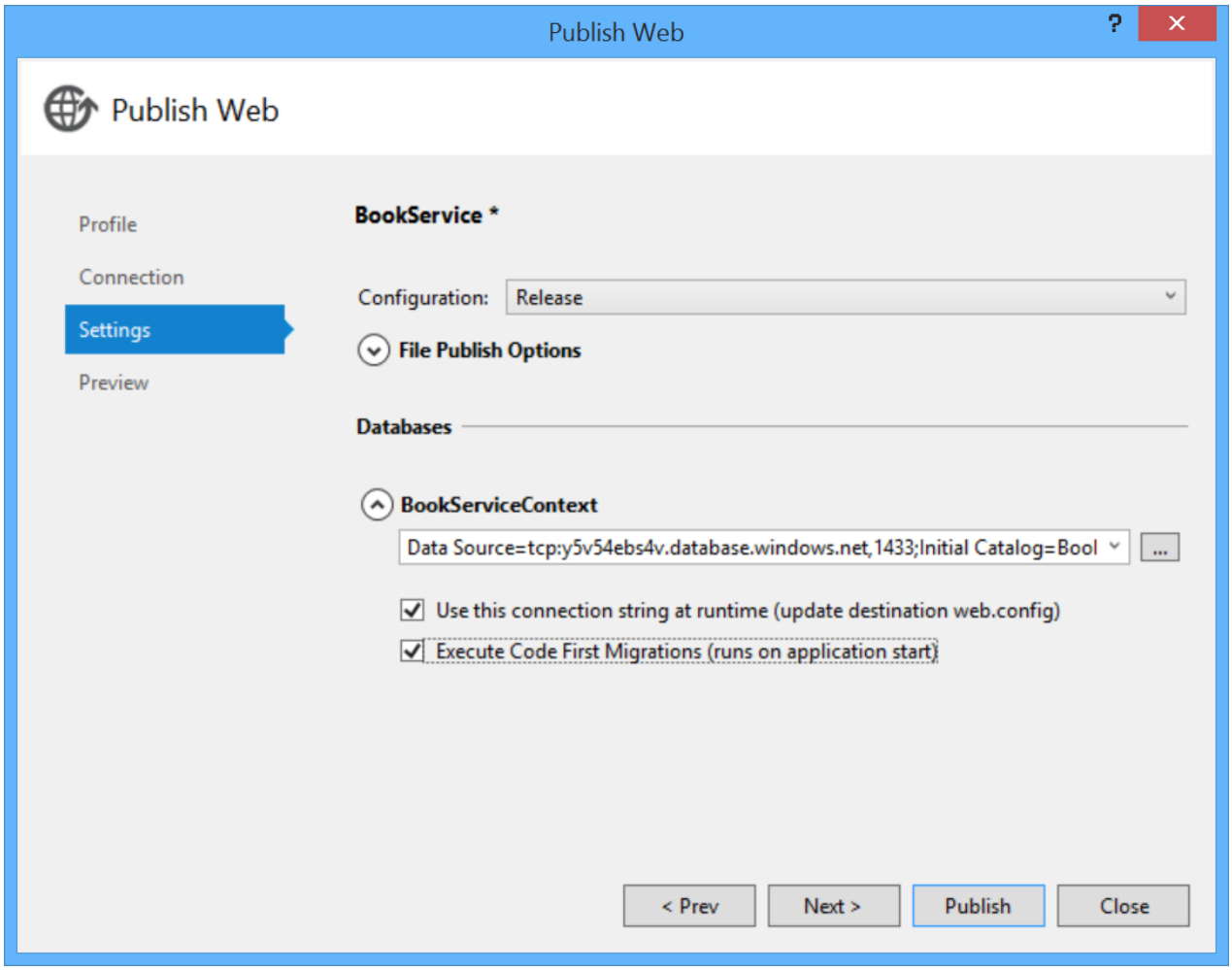
by Mike Wasson

Download Completed Project

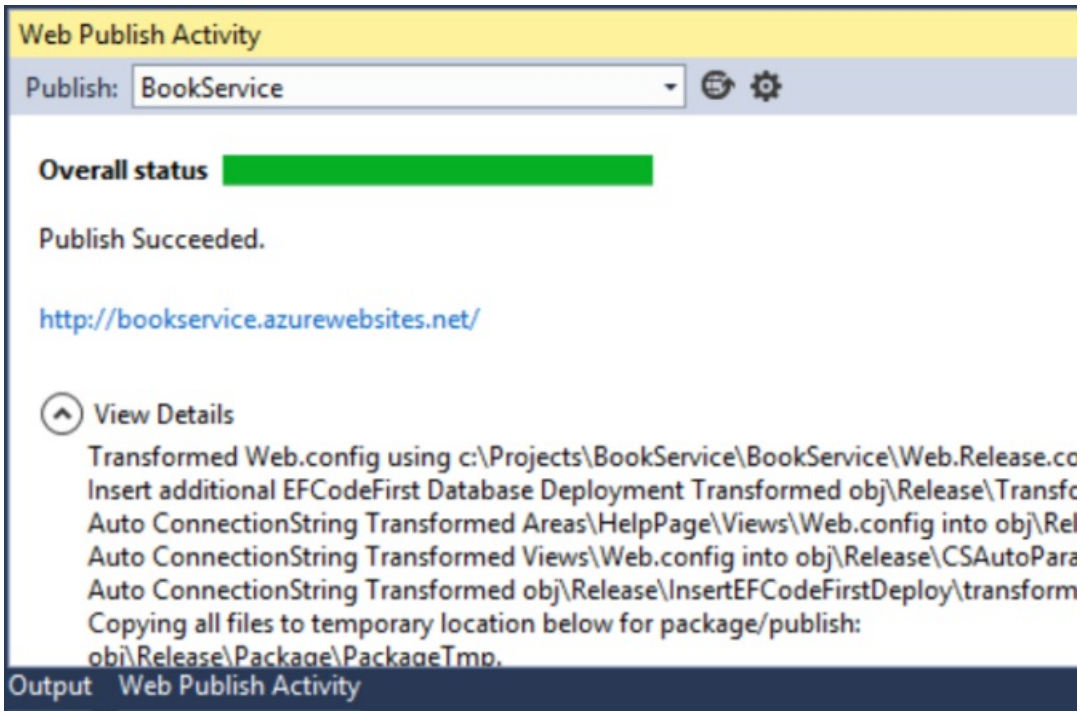
As the last step, you will publish the application to Azure. In Solution Explorer, right-click the project and select **Publish**.



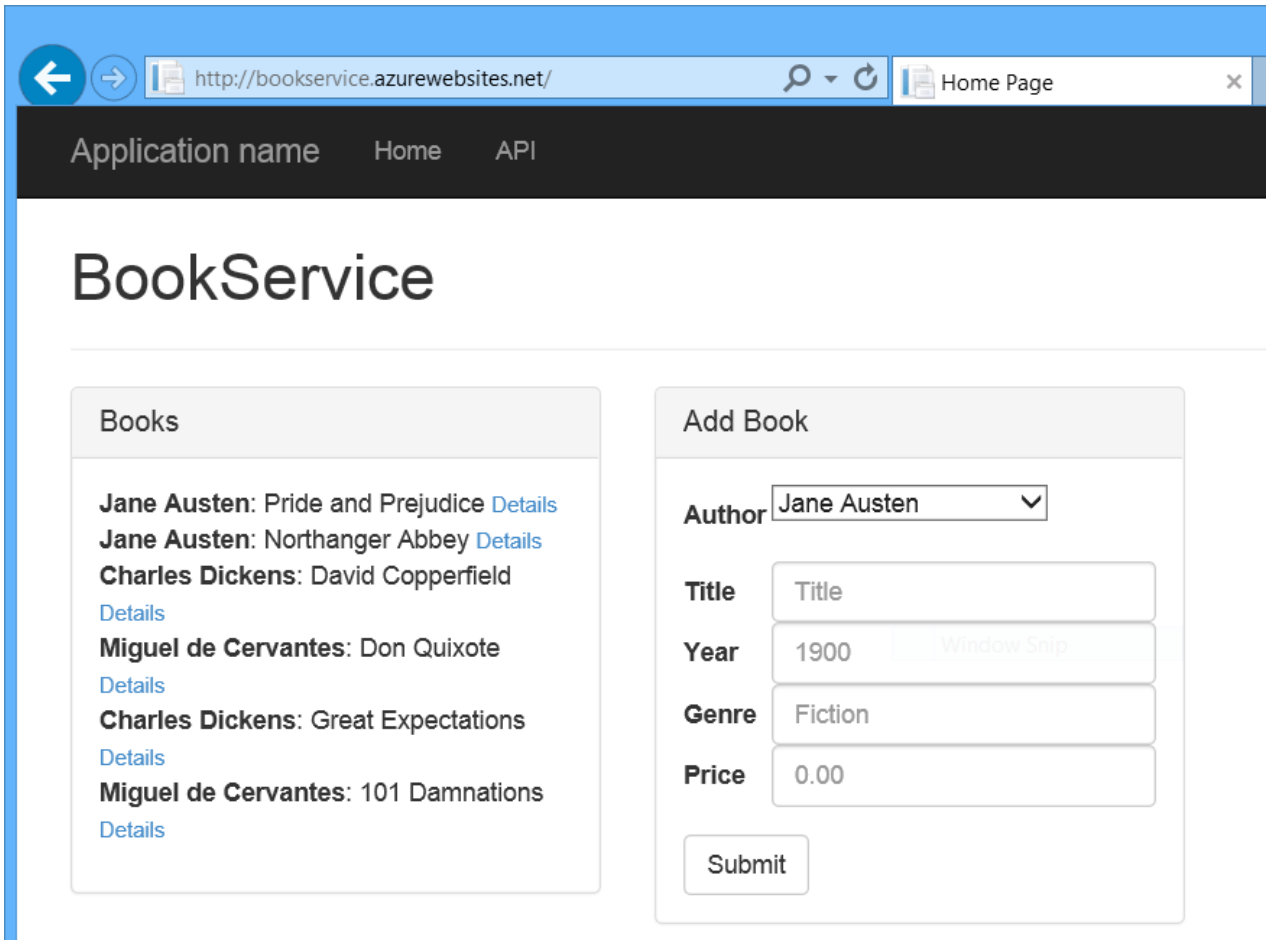
Clicking **Publish** invokes the **Publish Web** dialog. If you checked **Host in Cloud** when you first created the project, then the connection and settings are already configured. In that case, just click the **Settings** tab and check "Execute Code First Migrations". (If you didn't check **Host in Cloud** at the beginning, then follow the steps in the next section.)



To deploy the app, click **Publish**. You can view the publishing progress in the **Web Publish Activity** window. (From the **View** menu, select **Other Windows**, then select **Web Publish Activity**.)

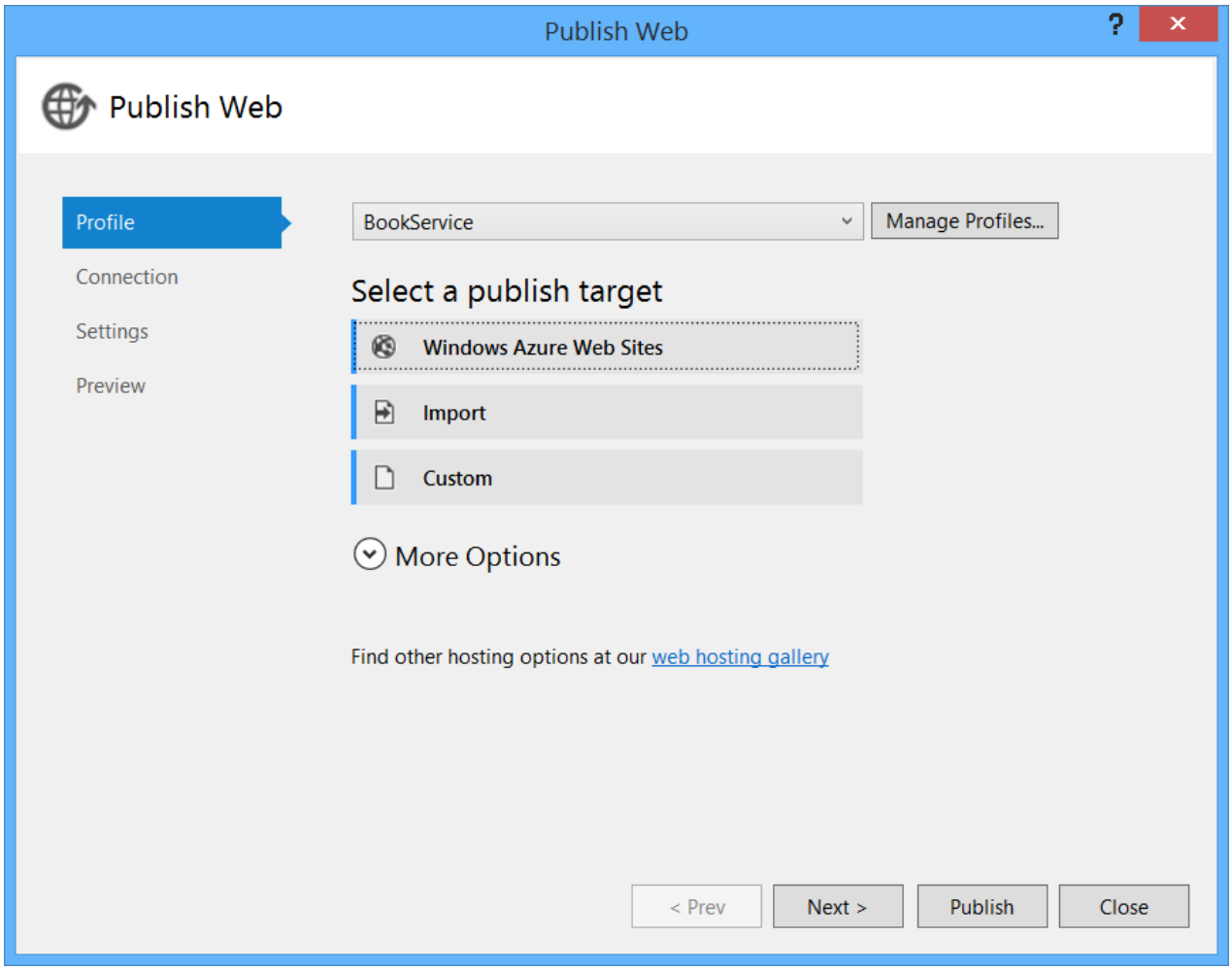


When Visual Studio finishes deploying the app, the default browser automatically opens to the URL of the deployed website, and the application that you created is now running in the cloud. The URL in the browser address bar shows that the site is being loaded from the Internet.

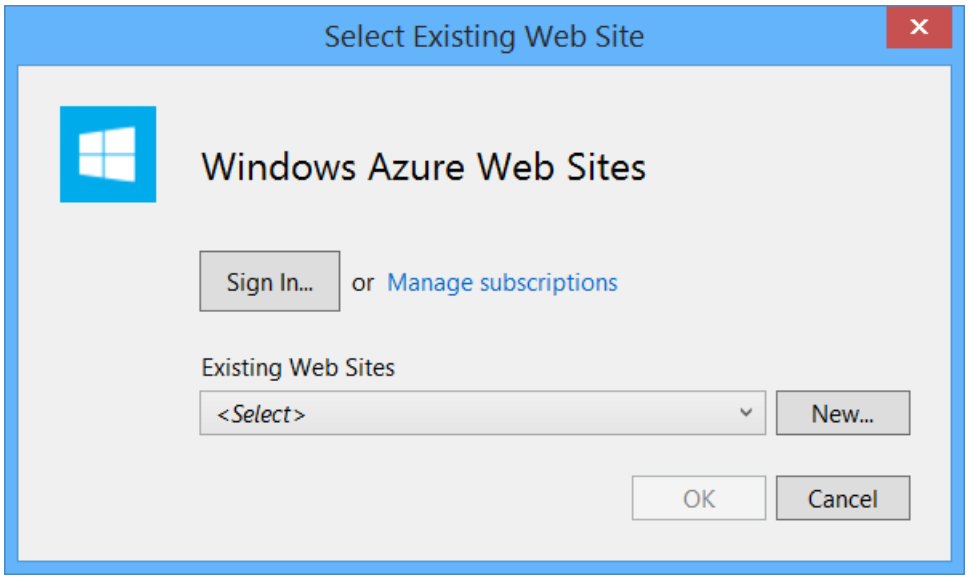


Deploying to a New Website

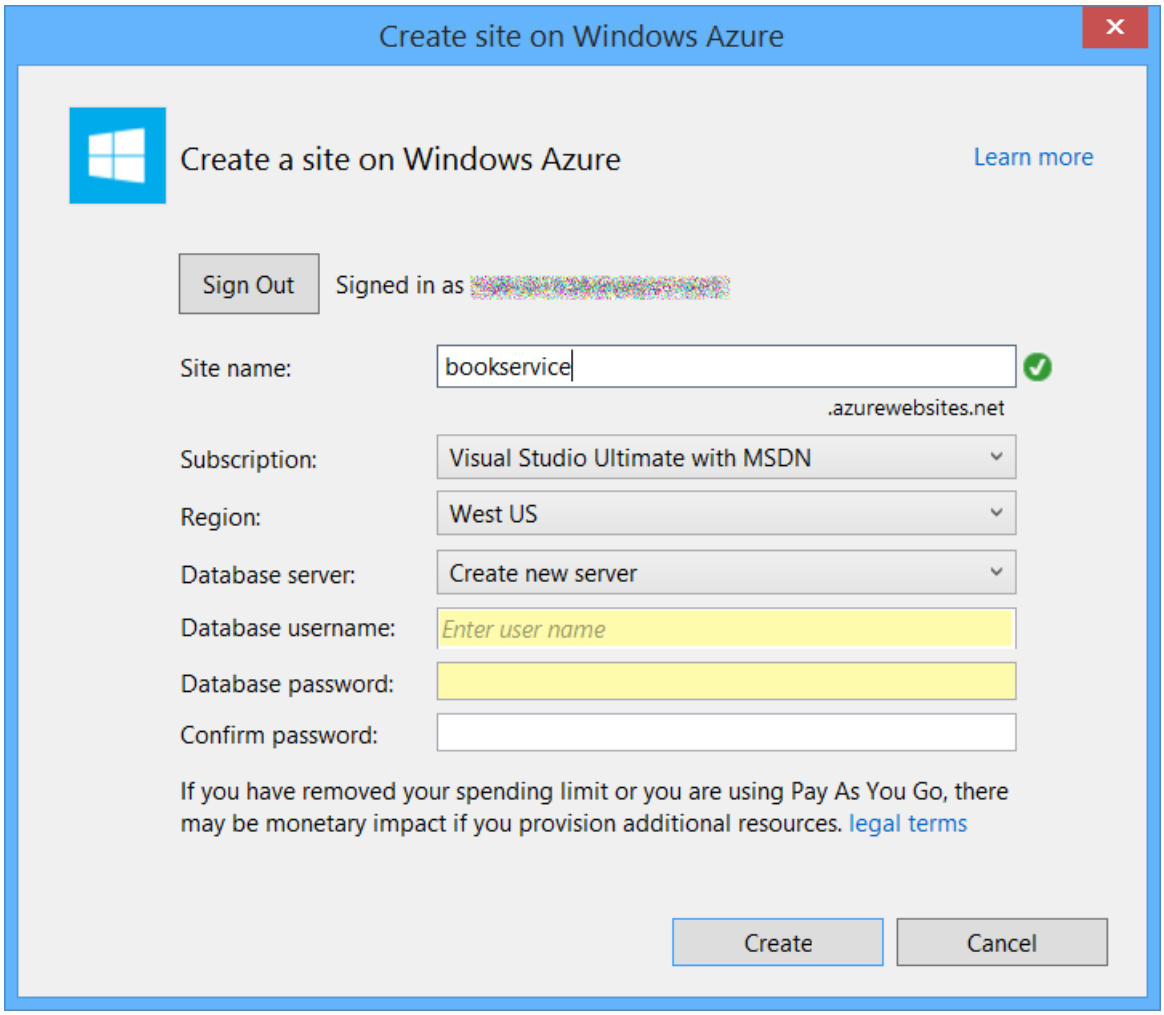
If you did not check **Host in Cloud** when you first created the project, you can configure a new web app now. In Solution Explorer, right-click the project and select **Publish**. Select the **Profile** tab and click **Microsoft Azure Websites**. If you aren't currently signed in to Azure, you will be prompted to sign in.



In the **Existing Websites** dialog, click **New**.



Enter a site name. Select your Azure subscription and the region. Under **Database server**, select **Create New Server**, or select an existing server. Click **Create**.



Click the **Settings** tab and check "Execute Code First Migrations". Then click **Publish**.

PREVIOUS

Web API and Mobile Clients

2/20/2020 • 2 minutes to read • Edit Online

Using ASP.NET Web API to reach native mobile clients.

- Calling Web API from a Windows Phone 8 Application (C#)

Calling Web API from a Windows Phone 8 Application (C#)

2/20/2020 • 16 minutes to read • Edit Online

by Robert McMurray

In this tutorial, you will learn how to create a complete end-to-end scenario consisting of an ASP.NET Web API application that provides a catalog of books to a Windows Phone 8 application.

Overview

RESTful services like ASP.NET Web API simplify the creation of HTTP-based applications for developers by abstracting the architecture for server-side and client-side applications. Instead of creating a proprietary socket-based protocol for communication, Web API developers simply need to publish the requisite HTTP methods for their application, (for example: GET, POST, PUT, DELETE), and client application developers only need to consume the HTTP methods that are necessary for their application.

In this end-to-end tutorial, you will learn how to use Web API to create the following projects:

- In the first part of this tutorial, you will create an ASP.NET Web API application that supports all of the Create, Read, Update, and Delete (CRUD) operations to manage a book catalog. This application will use the Sample XML File (books.xml) from MSDN.
- In the second part of this tutorial, you will create an interactive Windows Phone 8 application that retrieves the data from your Web API application.

Prerequisites

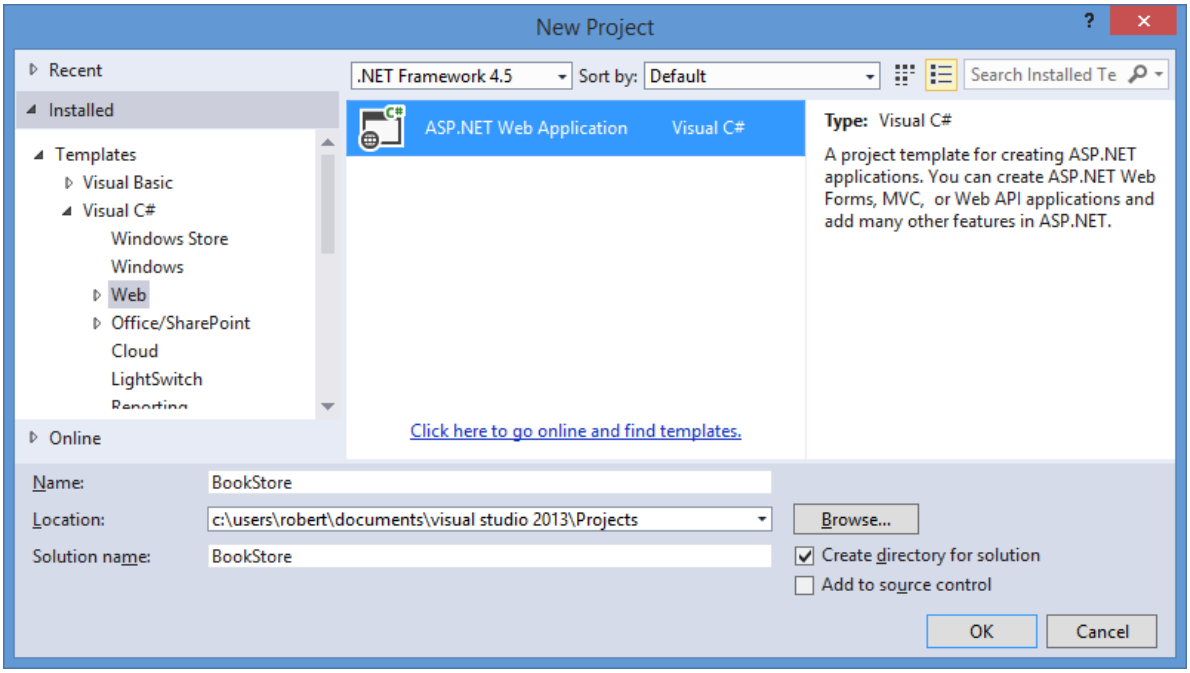
- Visual Studio 2013 with the Windows Phone 8 SDK installed
- Windows 8 or later on a 64-bit system with Hyper-V installed
- For a list of additional requirements, see the *System Requirements* section on the Windows Phone SDK 8.0 download page.

> **NOTE**
>
> If you are going to test the connectivity between Web API and Windows Phone 8 projects on your local system, you will need to follow the instructions in the *Connecting the Windows Phone 8 Emulator to Web API Applications on a Local Computer* article to set up your testing environment.

Step 1: Creating the Web API Bookstore Project

The first step of this end-to-end tutorial is to create a Web API project that supports all of the CRUD operations; note that you will add the Windows Phone application project to this solution in Step 2 of this tutorial.

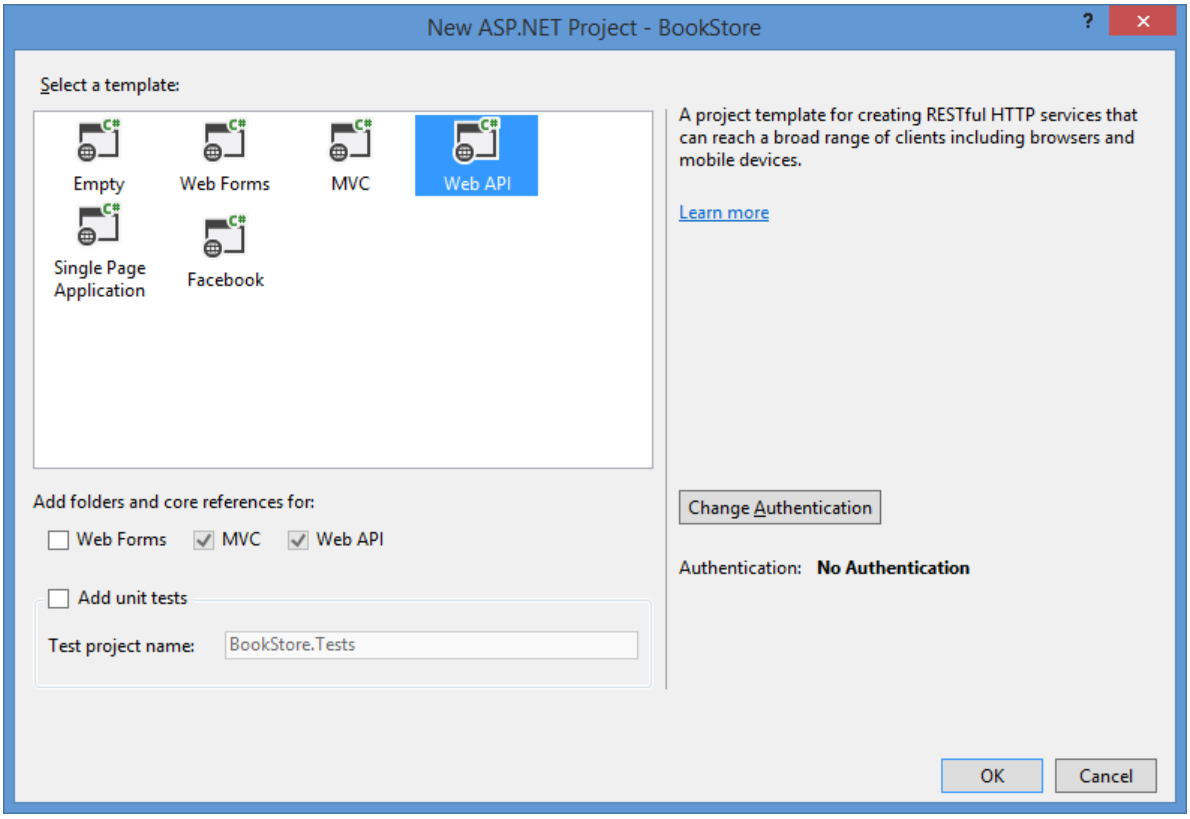
1. Open **Visual Studio 2013**.
2. Click **File**, then **New**, and then **Project**.
3. When the **New Project** dialog box is displayed, expand **Installed**, then **Templates**, then **Visual C#**, and then **Web**.



Click image to expand

4. Highlight **ASP.NET Web Application**, enter **BookStore** for the project name, and then click **OK**.

5. When the **New ASP.NET Project** dialog box is displayed, select the **Web API** template, and then click **OK**.



Click image to expand

6. When the Web API project opens, remove the sample controller from the project:

   a. Expand the **Controllers** folder in the solution explorer.
   b. Right-click the **ValuesController.cs** file, and then click **Delete**.
   c. Click **OK** when prompted to confirm the deletion.

7. Add an XML data file to the Web API project; this file contains the contents of the bookstore catalog:

   a. Right-click the **App_Data** folder in the solution explorer, then click **Add**, and then click **New Item**.

   b. When the **Add New Item** dialog box is displayed, highlight the **XML File** template.

   c. Name the file **Books.xml**, and then click **Add**.

   d. When the **Books.xml** file is opened, replace the code in the file with the XML from the sample **books.xml** file on MSDN:

```
<?xml version="1.0" encoding="utf-8"?>
<catalog>
  <book id="bk101">
    <author>Gambardella, Matthew</author>
    <title>XML Developer's Guide</title>
    <genre>Computer</genre>
    <price>44.95</price>
    <publish_date>2000-10-01</publish_date>
    <description>
      An in-depth look at creating applications
      with XML.
    </description>
  </book>
  <book id="bk102">
    <author>Ralls, Kim</author>
    <title>Midnight Rain</title>
    <genre>Fantasy</genre>
    <price>5.95</price>
    <publish_date>2000-12-16</publish_date>
    <description>
      A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.
    </description>
  </book>
  <book id="bk103">
    <author>Corets, Eva</author>
    <title>Maeve Ascendant</title>
    <genre>Fantasy</genre>
    <price>5.95</price>
    <publish_date>2000-11-17</publish_date>
    <description>
      After the collapse of a nanotechnology
      society in England, the young survivors lay the
      foundation for a new society.
    </description>
  </book>
  <book id="bk104">
    <author>Corets, Eva</author>
    <title>Oberon's Legacy</title>
    <genre>Fantasy</genre>
    <price>5.95</price>
    <publish_date>2001-03-10</publish_date>
    <description>
      In post-apocalypse England, the mysterious
      agent known only as Oberon helps to create a new life
      for the inhabitants of London. Sequel to Maeve
      Ascendant.
    </description>
  </book>
  <book id="bk105">
    <author>Corets, Eva</author>
    <title>The Sundered Grail</title>
    <genre>Fantasy</genre>
    <price>5.95</price>
    <publish_date>2001-09-10</publish_date>
```

```
    <description>
      The two daughters of Maeve, half-sisters,
      battle one another for control of England. Sequel to
      Oberon's Legacy.
    </description>
  </book>
  <book id="bk106">
    <author>Randall, Cynthia</author>
    <title>Lover Birds</title>
    <genre>Romance</genre>
    <price>4.95</price>
    <publish_date>2000-09-02</publish_date>
    <description>
      When Carla meets Paul at an ornithology
      conference, tempers fly as feathers get ruffled.
    </description>
  </book>
  <book id="bk107">
    <author>Thurman, Paula</author>
    <title>Splish Splash</title>
    <genre>Romance</genre>
    <price>4.95</price>
    <publish_date>2000-11-02</publish_date>
    <description>
      A deep sea diver finds true love twenty
      thousand leagues beneath the sea.
    </description>
  </book>
  <book id="bk108">
    <author>Knorr, Stefan</author>
    <title>Creepy Crawlies</title>
    <genre>Horror</genre>
    <price>4.95</price>
    <publish_date>2000-12-06</publish_date>
    <description>
      An anthology of horror stories about roaches,
      centipedes, scorpions  and other insects.
    </description>
  </book>
  <book id="bk109">
    <author>Kress, Peter</author>
    <title>Paradox Lost</title>
    <genre>Science Fiction</genre>
    <price>6.95</price>
    <publish_date>2000-11-02</publish_date>
    <description>
      After an inadvertant trip through a Heisenberg
      Uncertainty Device, James Salway discovers the problems
      of being quantum.
    </description>
  </book>
  <book id="bk110">
    <author>O'Brien, Tim</author>
    <title>Microsoft .NET: The Programming Bible</title>
    <genre>Computer</genre>
    <price>36.95</price>
    <publish_date>2000-12-09</publish_date>
    <description>
      Microsoft's .NET initiative is explored in
      detail in this deep programmer's reference.
    </description>
  </book>
  <book id="bk111">
    <author>O'Brien, Tim</author>
    <title>MSXML3: A Comprehensive Guide</title>
    <genre>Computer</genre>
    <price>36.95</price>
    <publish_date>2000-12-01</publish_date>
    <description>
```

```
      The Microsoft MSXML3 parser is covered in
      detail, with attention to XML DOM interfaces, XSLT processing,
      SAX and more.
    </description>
  </book>
  <book id="bk112">
    <author>Galos, Mike</author>
    <title>Visual Studio 7: A Comprehensive Guide</title>
    <genre>Computer</genre>
    <price>49.95</price>
    <publish_date>2001-04-16</publish_date>
    <description>
      Microsoft Visual Studio 7 is explored in depth,
      looking at how Visual Basic, Visual C++, C#, and ASP+ are
      integrated into a comprehensive development
      environment.
    </description>
  </book>
</catalog>
```

e. Save and close the XML file.

8. Add the bookstore model to the Web API project; this model contains the Create, Read, Update, and Delete (CRUD) logic for the bookstore application:

   a. Right-click the **Models** folder in the solution explorer, then click **Add**, and then click **Class**.

   b. When the **Add New Item** dialog box is displayed, name the class file **BookDetails.cs**, and then click **Add**.

   c. When the **BookDetails.cs** file is opened, replace the code in the file with the following:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Xml;
using System.Xml.Linq;
using System.Xml.XPath;
using System.Web;

namespace BookStore.Models
{
    /// <summary>
    /// Define a class that will hold the detailed information for a book.
    /// </summary>
    public class BookDetails
    {
        [Required]
        public String Id { get; set; }
        [Required]
        public String Title { get; set; }
        public String Author { get; set; }
        public String Genre { get; set; }
        public Decimal Price { get; set; }
        public DateTime PublishDate { get; set; }
        public String Description { get; set; }
    }

    /// <summary>
    /// Define an interface which contains the methods for the book repository.
    /// </summary>
    public interface IBookRepository
    {
        BookDetails CreateBook(BookDetails book);
        IEnumerable<BookDetails> ReadAllBooks();
```

```
        BookDetails ReadBook(String id);
        BookDetails UpdateBook(String id, BookDetails book);
        Boolean DeleteBook(String id);
    }

    /// <summary>
    /// Define a class based on the book repository interface which contains the method
implementations.
    /// </summary>
    public class BookRepository : IBookRepository
    {
        private string xmlFilename = null;
        private XDocument xmlDocument = null;

        /// <summary>
        /// Define the class constructor.
        /// </summary>
        public BookRepository()
        {
            try
            {
                // Determine the path to the books.xml file.
                xmlFilename = HttpContext.Current.Server.MapPath("~/app_data/books.xml");
                // Load the contents of the books.xml file into an XDocument object.
                xmlDocument = XDocument.Load(xmlFilename);
            }
            catch (Exception ex)
            {
                // Rethrow the exception.
                throw ex;
            }
        }

        /// <summary>
        /// Method to add a new book to the catalog.
        /// Defines the implementation of the POST method.
        /// </summary>
        public BookDetails CreateBook(BookDetails book)
        {
            try
            {
                // Retrieve the book with the highest ID from the catalog.
                var highestBook = (
                    from bookNode in xmlDocument.Elements("catalog").Elements("book")
                    orderby bookNode.Attribute("id").Value descending
                    select bookNode).Take(1);
                // Extract the ID from the book data.
                string highestId = highestBook.Attributes("id").First().Value;
                // Create an ID for the new book.
                string newId = "bk" + (Convert.ToInt32(highestId.Substring(2)) + 1).ToString();
                // Verify that this book ID does not currently exist.
                if (this.ReadBook(newId) == null)
                {
                    // Retrieve the parent element for the book catalog.
                    XElement bookCatalogRoot = xmlDocument.Elements("catalog").Single();
                    // Create a new book element.
                    XElement newBook = new XElement("book", new XAttribute("id", newId));
                    // Create elements for each of the book's data items.
                    XElement[] bookInfo = FormatBookData(book);
                    // Add the element to the book element.
                    newBook.ReplaceNodes(bookInfo);
                    // Append the new book to the XML document.
                    bookCatalogRoot.Add(newBook);
                    // Save the XML document.
                    xmlDocument.Save(xmlFilename);
                    // Return an object for the newly-added book.
                    return this.ReadBook(newId);
                }
            }
```

```
            catch (Exception ex)
            {
                // Rethrow the exception.
                throw ex;
            }
            // Return null to signify failure.
            return null;
        }

        /// <summary>
        /// Method to retrieve all of the books in the catalog.
        /// Defines the implementation of the non-specific GET method.
        /// </summary>
        public IEnumerable<BookDetails> ReadAllBooks()
        {
            try
            {
                // Return a list that contains the catalog of book ids/titles.
                return (
                    // Query the catalog of books.
                    from book in xmlDocument.Elements("catalog").Elements("book")
                    // Sort the catalog based on book IDs.
                    orderby book.Attribute("id").Value ascending
                    // Create a new instance of the detailed book information class.
                    select new BookDetails
                    {
                        // Populate the class with data from each of the book's elements.
                        Id = book.Attribute("id").Value,
                        Author = book.Element("author").Value,
                        Title = book.Element("title").Value,
                        Genre = book.Element("genre").Value,
                        Price = Convert.ToDecimal(book.Element("price").Value),
                        PublishDate = Convert.ToDateTime(book.Element("publish_date").Value),
                        Description = book.Element("description").Value
                    }).ToList();
            }
            catch (Exception ex)
            {
                // Rethrow the exception.
                throw ex;
            }
        }

        /// <summary>
        /// Method to retrieve a specific book from the catalog.
        /// Defines the implementation of the ID-specific GET method.
        /// </summary>
        public BookDetails ReadBook(String id)
        {
            try
            {
                // Retrieve a specific book from the catalog.
                return (
                    // Query the catalog of books.
                    from book in xmlDocument.Elements("catalog").Elements("book")
                    // Specify the specific book ID to query.
                    where book.Attribute("id").Value.Equals(id)
                    // Create a new instance of the detailed book information class.
                    select new BookDetails
                    {
                        // Populate the class with data from each of the book's elements.
                        Id = book.Attribute("id").Value,
                        Author = book.Element("author").Value,
                        Title = book.Element("title").Value,
                        Genre = book.Element("genre").Value,
                        Price = Convert.ToDecimal(book.Element("price").Value),
                        PublishDate = Convert.ToDateTime(book.Element("publish_date").Value),
                        Description = book.Element("description").Value
                    }).Single();
```

```
            }
            catch
            {
                // Return null to signify failure.
                return null;
            }
        }

        /// <summary>
        /// Populates a book BookDetails class with the data for a book.
        /// </summary>
        private XElement[] FormatBookData(BookDetails book)
        {
            XElement[] bookInfo =
            {
                new XElement("author", book.Author),
                new XElement("title", book.Title),
                new XElement("genre", book.Genre),
                new XElement("price", book.Price.ToString()),
                new XElement("publish_date", book.PublishDate.ToString()),
                new XElement("description", book.Description)
            };
            return bookInfo;
        }

        /// <summary>
        /// Method to update an existing book in the catalog.
        /// Defines the implementation of the PUT method.
        /// </summary>
        public BookDetails UpdateBook(String id, BookDetails book)
        {
            try
            {
                // Retrieve a specific book from the catalog.
                XElement updateBook =
xmlDocument.XPathSelectElement(String.Format("catalog/book[@id='{0}']", id));
                // Verify that the book exists.
                if (updateBook != null)
                {
                    // Create elements for each of the book's data items.
                    XElement[] bookInfo = FormatBookData(book);
                    // Add the element to the book element.
                    updateBook.ReplaceNodes(bookInfo);
                    // Save the XML document.
                    xmlDocument.Save(xmlFilename);
                    // Return an object for the updated book.
                    return this.ReadBook(id);
                }
            }
            catch (Exception ex)
            {
                // Rethrow the exception.
                throw ex;
            }
            // Return null to signify failure.
            return null;
        }

        /// <summary>
        /// Method to remove an existing book from the catalog.
        /// Defines the implementation of the DELETE method.
        /// </summary>
        public Boolean DeleteBook(String id)
        {
            try
            {
                if (this.ReadBook(id) != null)
                {
                    // Remove the specific child node from the catalog.
```

```
                    xmlDocument
                        .Elements("catalog")
                        .Elements("book")
                        .Where(x => x.Attribute("id").Value.Equals(id))
                        .Remove();
                    // Save the XML document.
                    xmlDocument.Save(xmlFilename);
                    // Return a success status.
                    return true;
                }
                else
                {
                    // Return a failure status.
                    return false;
                }
            }
            catch (Exception ex)
            {
                // Rethrow the exception.
                throw ex;
            }
        }
    }
}
```

d. Save and close the **BookDetails.cs** file.

9. Add the bookstore controller to the Web API project:

   a. Right-click the **Controllers** folder in the solution explorer, then click **Add**, and then click **Controller**.

   b. When the **Add Scaffold** dialog box is displayed, highlight **Web API 2 Controller - Empty**, and then click **Add**.

   c. When the **Add Controller** dialog box is displayed, name the controller **BooksController**, and then click **Add**.

   d. When the **BooksController.cs** file is opened, replace the code in the file with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using BookStore.Models;

namespace BookStore.Controllers
{
    public class BooksController : ApiController
    {
        private BookRepository repository = null;

        // Define the class constructor.
        public BooksController()
        {
            this.repository = new BookRepository();
        }

        /// <summary>
        /// Method to retrieve all of the books in the catalog.
        /// Example: GET api/books
        /// </summary>
        [HttpGet]
        public HttpResponseMessage Get()
        {
```

```
            IEnumerable<BookDetails> books = this.repository.ReadAllBooks();
            if (books != null)
            {
                return Request.CreateResponse<IEnumerable<BookDetails>>(HttpStatusCode.OK,
books);
            }
            else
            {
                return Request.CreateResponse(HttpStatusCode.NotFound);
            }
        }

        /// <summary>
        /// Method to retrieve a specific book from the catalog.
        /// Example: GET api/books/5
        /// </summary>
        [HttpGet]
        public HttpResponseMessage Get(String id)
        {
            BookDetails book = this.repository.ReadBook(id);
            if (book != null)
            {
                return Request.CreateResponse<BookDetails>(HttpStatusCode.OK, book);
            }
            else
            {
                return Request.CreateResponse(HttpStatusCode.NotFound);
            }
        }

        /// <summary>
        /// Method to add a new book to the catalog.
        /// Example: POST api/books
        /// </summary>
        [HttpPost]
        public HttpResponseMessage Post(BookDetails book)
        {
            if ((this.ModelState.IsValid) && (book != null))
            {
                BookDetails newBook = this.repository.CreateBook(book);
                if (newBook != null)
                {
                    var httpResponse = Request.CreateResponse<BookDetails>
(HttpStatusCode.Created, newBook);
                    string uri = Url.Link("DefaultApi", new { id = newBook.Id });
                    httpResponse.Headers.Location = new Uri(uri);
                    return httpResponse;
                }
            }
            return Request.CreateResponse(HttpStatusCode.BadRequest);
        }

        /// <summary>
        /// Method to update an existing book in the catalog.
        /// Example: PUT api/books/5
        /// </summary>
        [HttpPut]
        public HttpResponseMessage Put(String id, BookDetails book)
        {
            if ((this.ModelState.IsValid) && (book != null) && (book.Id.Equals(id)))
            {
                BookDetails modifiedBook = this.repository.UpdateBook(id, book);
                if (modifiedBook != null)
                {
                    return Request.CreateResponse<BookDetails>(HttpStatusCode.OK, modifiedBook);
                }
                else
                {
                    return Request.CreateResponse(HttpStatusCode.NotFound);
```

```
                }
            }
            return Request.CreateResponse(HttpStatusCode.BadRequest);
        }

        /// <summary>
        /// Method to remove an existing book from the catalog.
        /// Example: DELETE api/books/5
        /// </summary>
        [HttpDelete]
        public HttpResponseMessage Delete(String id)
        {
            BookDetails book = this.repository.ReadBook(id);
            if (book != null)
            {
                if (this.repository.DeleteBook(id))
                {
                    return Request.CreateResponse(HttpStatusCode.OK);
                }
            }
            else
            {
                return Request.CreateResponse(HttpStatusCode.NotFound);
            }
            return Request.CreateResponse(HttpStatusCode.BadRequest);
        }
    }
}
```

e. Save and close the **BooksController.cs** file.

10. Build the Web API application to check for errors.

Step 2: Adding the Windows Phone 8 Bookstore Catalog Project

The next step of this end-to-end scenario is to create the catalog application for Windows Phone 8. This application will use the *Windows Phone Databound App* template for the default user interface, and it will use the Web API application that you created in Step 1 of this tutorial as the data source.

1. Right-click the **BookStore** solution in the in the solution explorer, then click **Add**, and then **New Project**.
2. When the **New Project** dialog box is displayed, expand **Installed**, then **Visual C#**, and then **Windows Phone**.
3. Highlight **Windows Phone Databound App**, enter **BookCatalog** for the name, and then click **OK**.
4. Add the Json.NET NuGet package to the **BookCatalog** project:
   a. Right-click **References** for the **BookCatalog** project in the solution explorer, and then click **Manage NuGet Packages**.
   b. When the **Manage NuGet Packages** dialog box is displayed, expand the **Online** section, and highlight **nuget.org**.
   c. Enter **Json.NET** in the search field and click the search icon.
   d. Highlight **Json.NET** in the search results, and then click **Install**.
   e. When the installation has completed, click **Close**.
5. Add the **BookDetails** model to the **BookCatalog** project; this contains a generic model of the bookstore class:
   a. Right-click the **BookCatalog** project in the solution explorer, then click **Add**, and then click **New Folder**.
   b. Name the new folder **Models**.

c. Right-click the **Models** folder in the solution explorer, then click **Add**, and then click **Class**.

d. When the **Add New Item** dialog box is displayed, name the class file **BookDetails.cs**, and then click **Add**.

e. When the **BookDetails.cs** file is opened, replace the code in the file with the following:

```
using System;
using System.Text;

namespace BookCatalog.Models
{
    /// <summary>
    /// Define a class that will hold the detailed information for a book.
    /// </summary>
    public class BookDetails
    {
        public String Id { get; set; }
        public String Title { get; set; }
        public String Author { get; set; }
        public String Genre { get; set; }
        public Decimal Price { get; set; }
        public DateTime PublishDate { get; set; }
        public String Description { get; set; }
    }
}
```

f. Save and close the **BookDetails.cs** file.

6. Update the **MainViewModel.cs** class to include the functionality to communicate with the BookStore Web API application:

a. Expand the **ViewModels** folder in the solution explorer, and then double-click the **MainViewModel.cs** file.

b. When the **MainViewModel.cs** file is opened, replace the code in the file with the following; note that you will need to update the value of the `apiUrl` constant with the actual URL of your Web API:

```
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Net;
using System.Net.NetworkInformation;
using BookCatalog.Resources;
using System.Collections.Generic;
using Newtonsoft.Json;
using BookCatalog.Models;

namespace BookCatalog.ViewModels
{
    public class MainViewModel : INotifyPropertyChanged
    {
        const string apiUrl = @"http://www.contoso.com/api/Books";

        public MainViewModel()
        {
            this.Items = new ObservableCollection<ItemViewModel>();
        }

        /// <summary>
        /// A collection for ItemViewModel objects.
        /// </summary>
        public ObservableCollection<ItemViewModel> Items { get; private set; }
```

```
        public bool IsDataLoaded
        {
            get;
            private set;
        }

        /// <summary>
        /// Creates and adds a few ItemViewModel objects into the Items collection.
        /// </summary>
        public void LoadData()
        {
            if (this.IsDataLoaded == false)
            {
                this.Items.Clear();
                this.Items.Add(new ItemViewModel() { ID = "0", LineOne = "Please Wait...",
LineTwo = "Please wait while the catalog is downloaded from the server.", LineThree = null });
                WebClient webClient = new WebClient();
                webClient.Headers["Accept"] = "application/json";
                webClient.DownloadStringCompleted += new
DownloadStringCompletedEventHandler(webClient_DownloadCatalogCompleted);
                webClient.DownloadStringAsync(new Uri(apiUrl));
            }
        }

        private void webClient_DownloadCatalogCompleted(object sender,
DownloadStringCompletedEventArgs e)
        {
            try
            {
                this.Items.Clear();
                if (e.Result != null)
                {
                    var books = JsonConvert.DeserializeObject<BookDetails[]>(e.Result);
                    int id = 0;
                    foreach (BookDetails book in books)
                    {
                        this.Items.Add(new ItemViewModel()
                        {
                            ID = (id++).ToString(),
                            LineOne = book.Title,
                            LineTwo = book.Author,
                            LineThree = book.Description.Replace("\n", " ")
                        });
                    }
                    this.IsDataLoaded = true;
                }
            }
            catch (Exception ex)
            {
                this.Items.Add(new ItemViewModel()
                {
                    ID = "0",
                    LineOne = "An Error Occurred",
                    LineTwo = String.Format("The following exception occured: {0}", ex.Message),
                    LineThree = String.Format("Additional inner exception information: {0}",
ex.InnerException.Message)
                });
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;
        private void NotifyPropertyChanged(String propertyName)
        {
            PropertyChangedEventHandler handler = PropertyChanged;
            if (null != handler)
            {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }
```

```
    }
}
```

c. Save and close the **MainViewModel.cs** file.

7. Update the **MainPage.xaml** file to customize the application name:

   a. Double-click the **MainPage.xaml** file in the solution explorer.

   b. When the **MainPage.xaml** file is opened, locate the following lines of code:

```
<StackPanel Grid.Row="0" Margin="12,17,0,28">
    <TextBlock Text="MY APPLICATION" Style="{StaticResource PhoneTextNormalStyle}"/>
    <TextBlock Text="page name" Margin="9,-7,0,0" Style="{StaticResource
PhoneTextTitle1Style}"/>
</StackPanel>
```

   c. Replace those lines with the following:

```
<StackPanel Grid.Row="0" Margin="12,17,0,28">
    <TextBlock Text="Book Store" Style="{StaticResource PhoneTextTitle1Style}"/>
    <TextBlock Text="Current Catalog" Margin="9,-7,0,0" Style="{StaticResource
PhoneTextTitle2Style}"/>
</StackPanel>
```

   d. Save and close the **MainPage.xaml** file.

8. Update the **DetailsPage.xaml** file to customize the displayed items:

   a. Double-click the **DetailsPage.xaml** file in the solution explorer.

   b. When the **DetailsPage.xaml** file is opened, locate the following lines of code:

```
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
    <TextBlock Text="MY APPLICATION" Style="{StaticResource PhoneTextNormalStyle}"/>
    <TextBlock Text="{Binding LineOne}" Margin="9,-7,0,0" Style="{StaticResource
PhoneTextTitle1Style}"/>
</StackPanel>
```

   c. Replace those lines with the following:

```
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
    <TextBlock Text="Book Store" Style="{StaticResource PhoneTextTitle1Style}"/>
    <TextBlock Text="{Binding LineOne}" Margin="9,-7,0,0" Style="{StaticResource
PhoneTextTitle2Style}"/>
</StackPanel>
```

   d. Save and close the **DetailsPage.xaml** file.

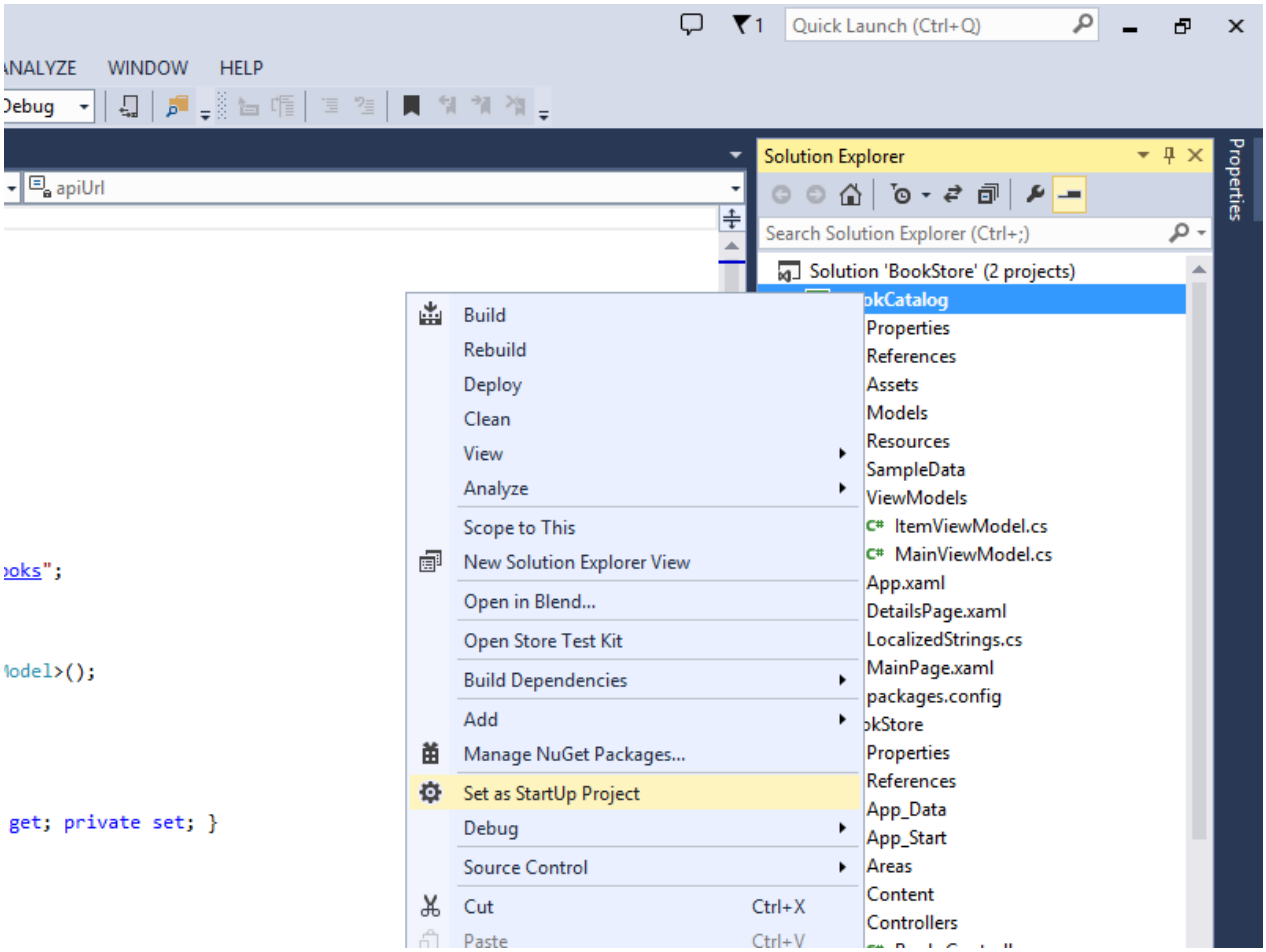
9. Build the Windows Phone application to check for errors.

Step 3: Testing the End-to-End Solution

As mentioned in the *Prerequisites* section of this tutorial, when you are testing the connectivity between Web API and Windows Phone 8 projects on your local system, you will need to follow the instructions in the *Connecting the Windows Phone 8 Emulator to Web API Applications on a Local Computer* article to set up your testing environment.

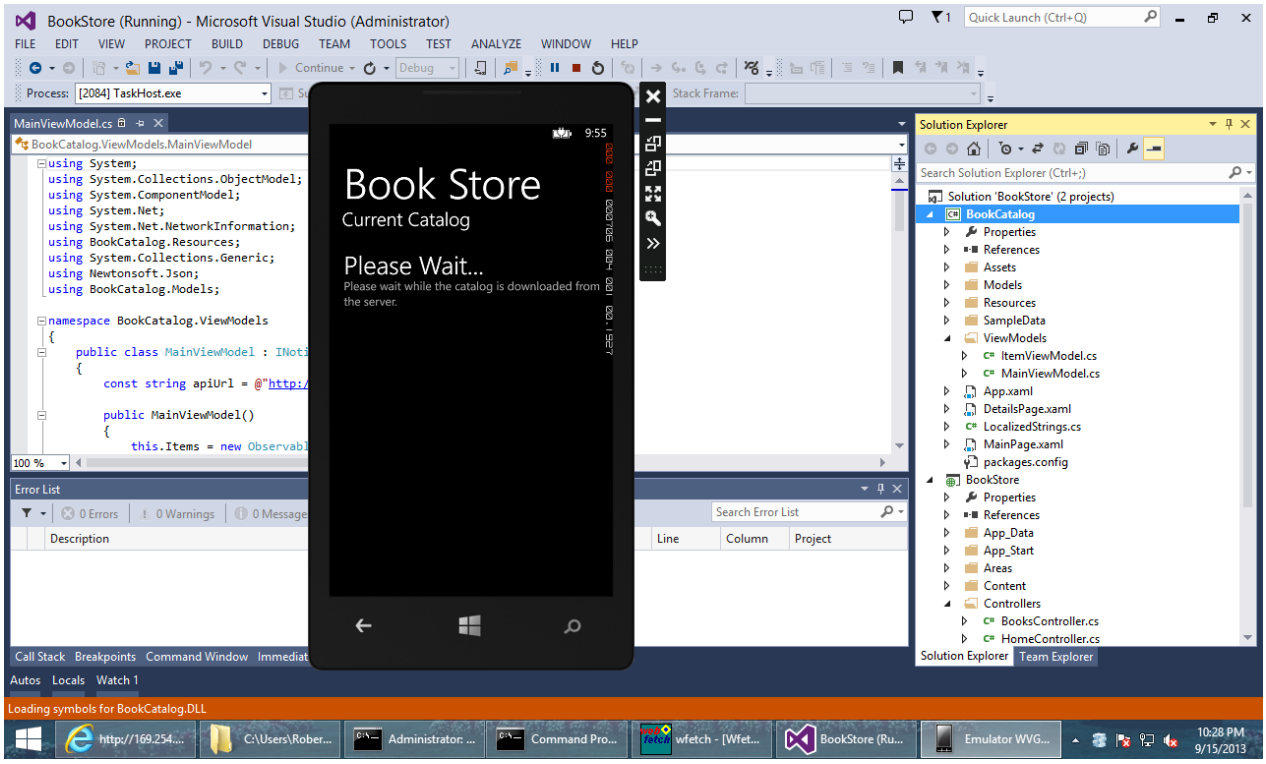
Once you have the testing environment configured, you will need to set the Windows Phone application as the

startup project. To do so, highlight the **BookCatalog** application in the solution explorer, and then click **Set as StartUp Project**:



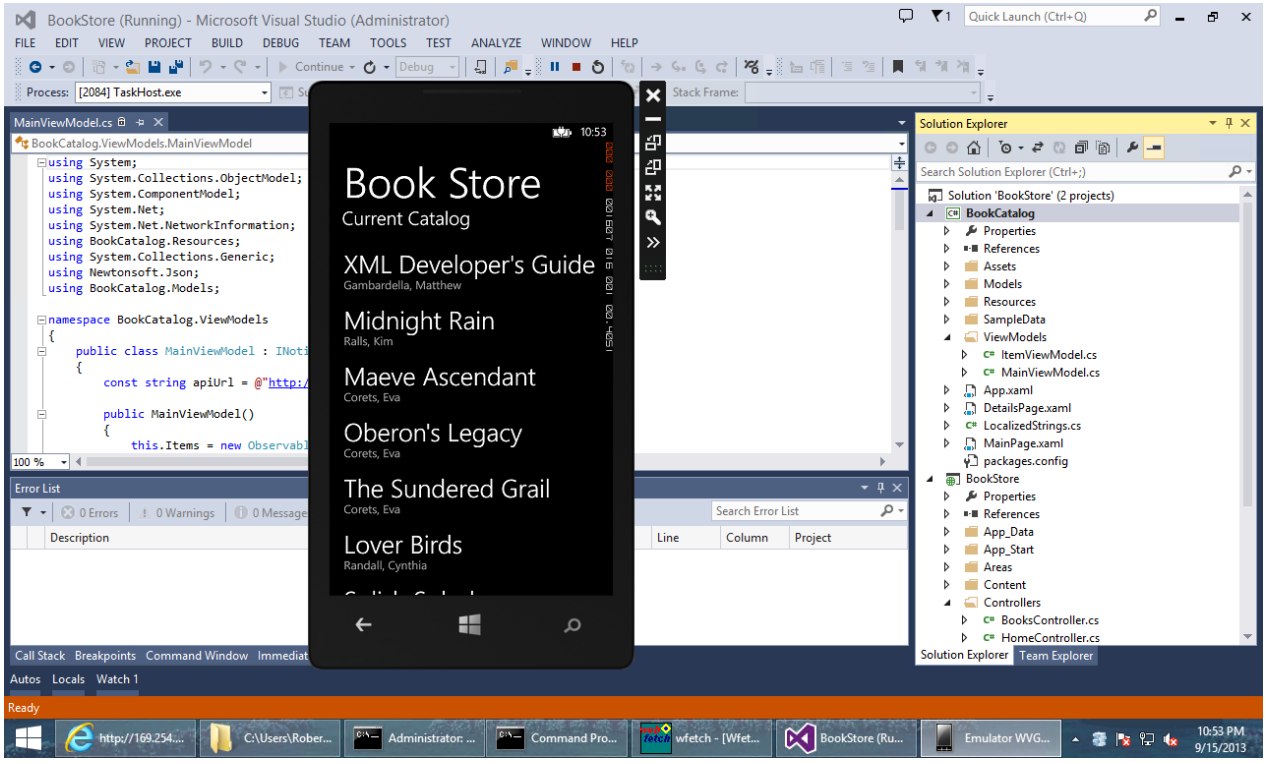
Click image to expand

When you press F5, Visual Studio will start both the Windows Phone Emulator, which will display a "Please Wait" message while the application data is retrieved from your Web API:



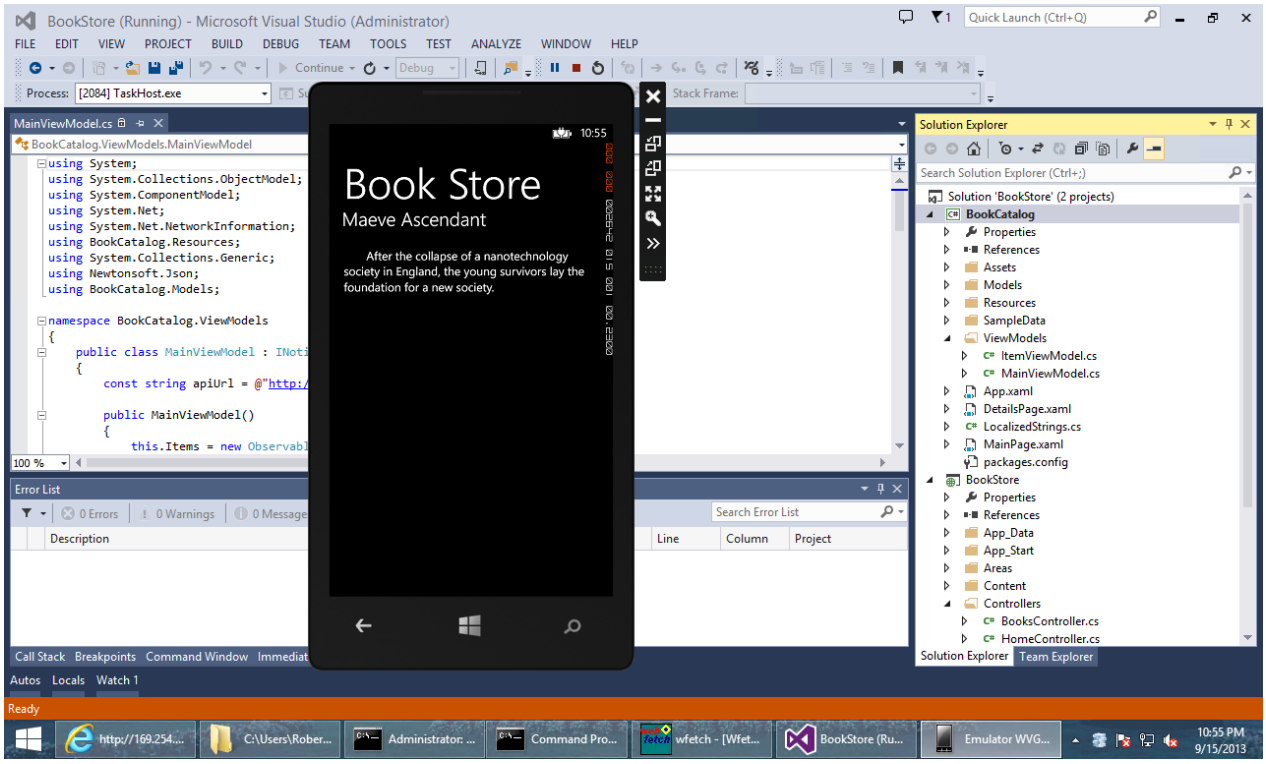
Click image to expand

If everything is successful, you should see the catalog displayed:



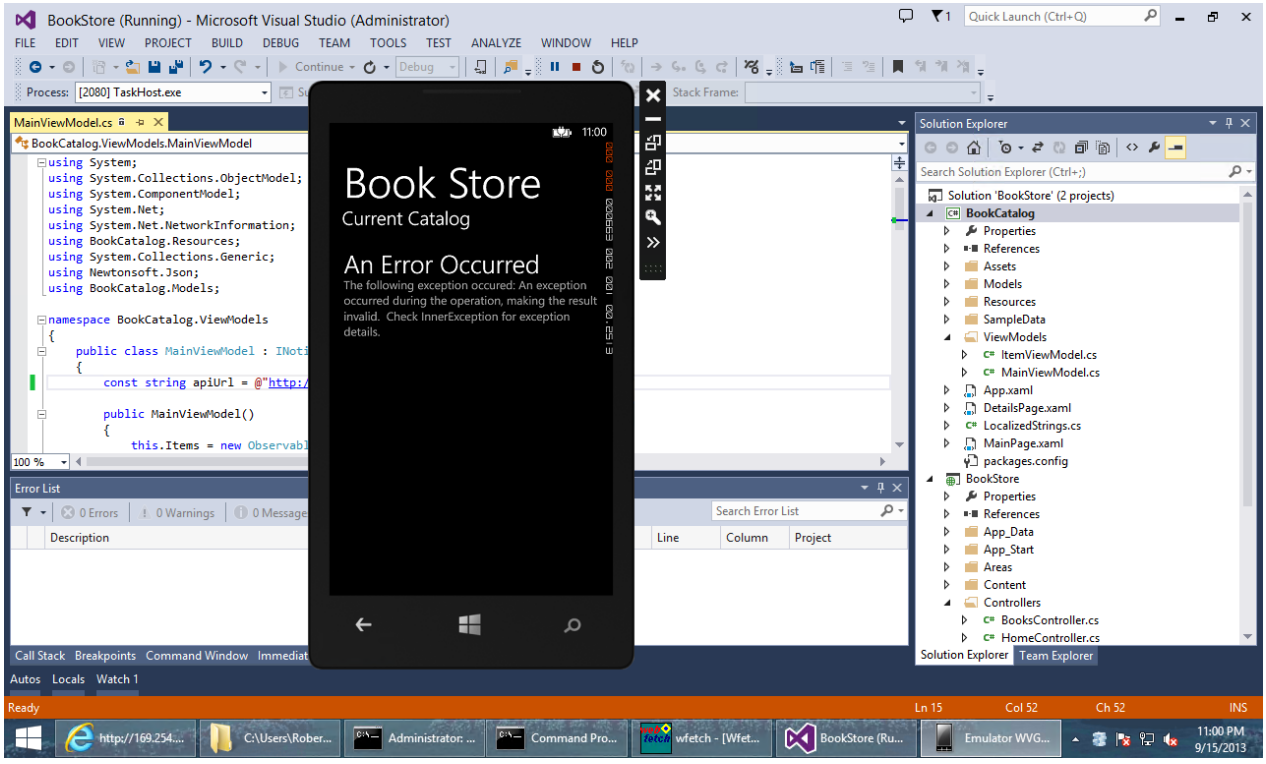
Click image to expand

If you tap on any book title, the application will display the book description:



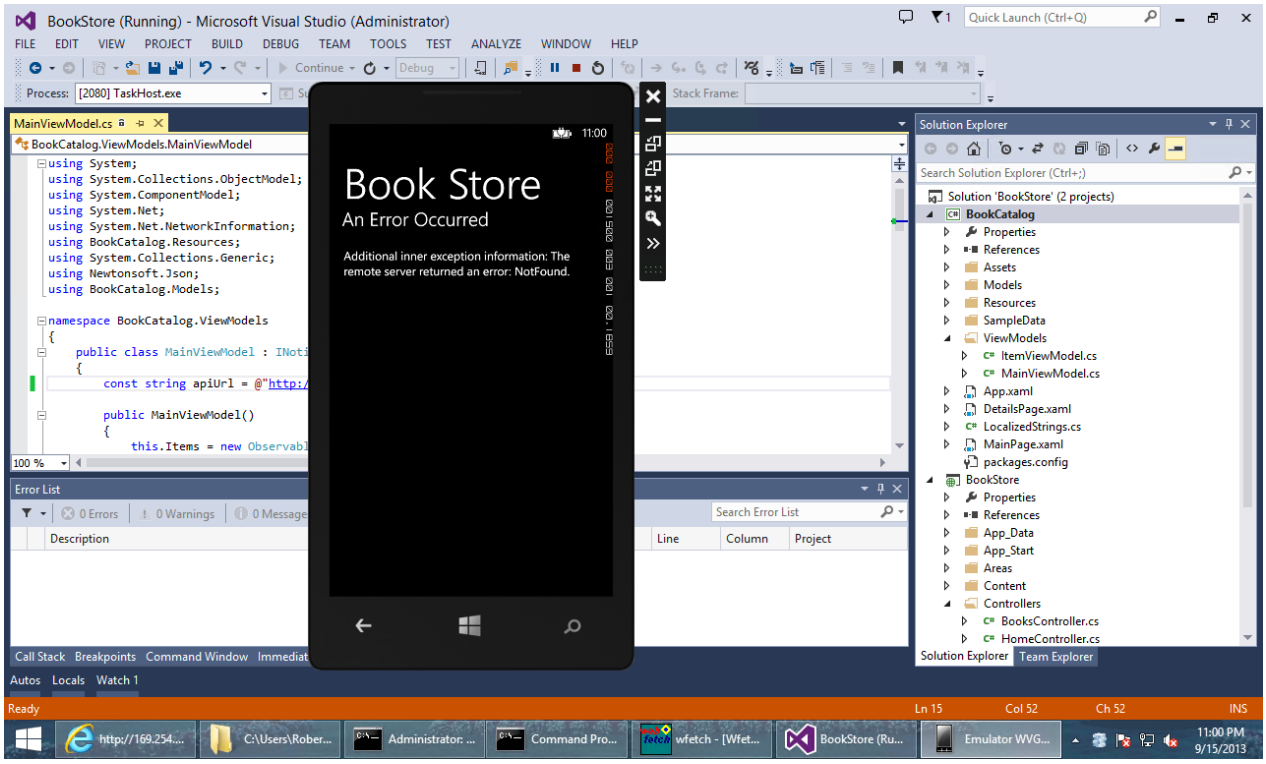
Click image to expand

If the application cannot communicate with your Web API, an error message will be displayed:



Click image to expand

If you tap on the error message, any additional details about the error will be displayed:



Click image to expand

OData in ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

How to create OData endpoints in ASP.NET Web API for ASP.NET 4.x. OData is a data access protocol for the web. It provides a uniform way to query and manipulate data sets. Web API supports both Version 3 and Version 4 of the OData protocol.

- OData v4
  - Create an OData v4 Endpoint
  - Create an OData v4 Client App (C#)
  - Entity Relations in OData v4
  - Actions and Functions in OData v4
  - Containment in OData v4
  - Create a Singleton in OData v4
  - Open Types in OData v4
  - Complex Type Inheritance in OData v4
- OData v3
  - Creating an OData v3 Endpoint
  - Supporting Entity Relations in OData v3
  - OData Actions in Web API 2
  - Calling an OData Service From a .NET Client (C#)
- Supporting OData Query Options in Web API 2
- Using $select, $expand, and $value in Web API 2 OData
- Routing Conventions in Web API 2 OData
- Security Guidance for Web API 2 OData

Supporting OData v4 in ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

Use ASP.NET Web API to create endpoints that support OData Version 4.

- Create an OData v4 Endpoint
- Create an OData v4 Client App (C#)
- Entity Relations in OData v4
- Actions and Functions in OData v4
- Containment in OData v4
- Create a Singleton in OData v4
- Open Types in OData v4
- Complex Type Inheritance in OData v4

Create an OData v4 Endpoint Using ASP.NET Web API

2/20/2020 • 6 minutes to read • Edit Online

The Open Data Protocol (OData) is a data access protocol for the web. OData provides a uniform way to query and manipulate data sets through CRUD operations (create, read, update, and delete).

ASP.NET Web API supports both v3 and v4 of the protocol. You can even have a v4 endpoint that runs side-by-side with a v3 endpoint.

This tutorial shows how to create an OData v4 endpoint that supports CRUD operations.

Software versions used in the tutorial

- Web API 5.2
- OData v4
- Visual Studio 2017 (download Visual Studio 2017 here)
- Entity Framework 6
- .NET 4.7.2

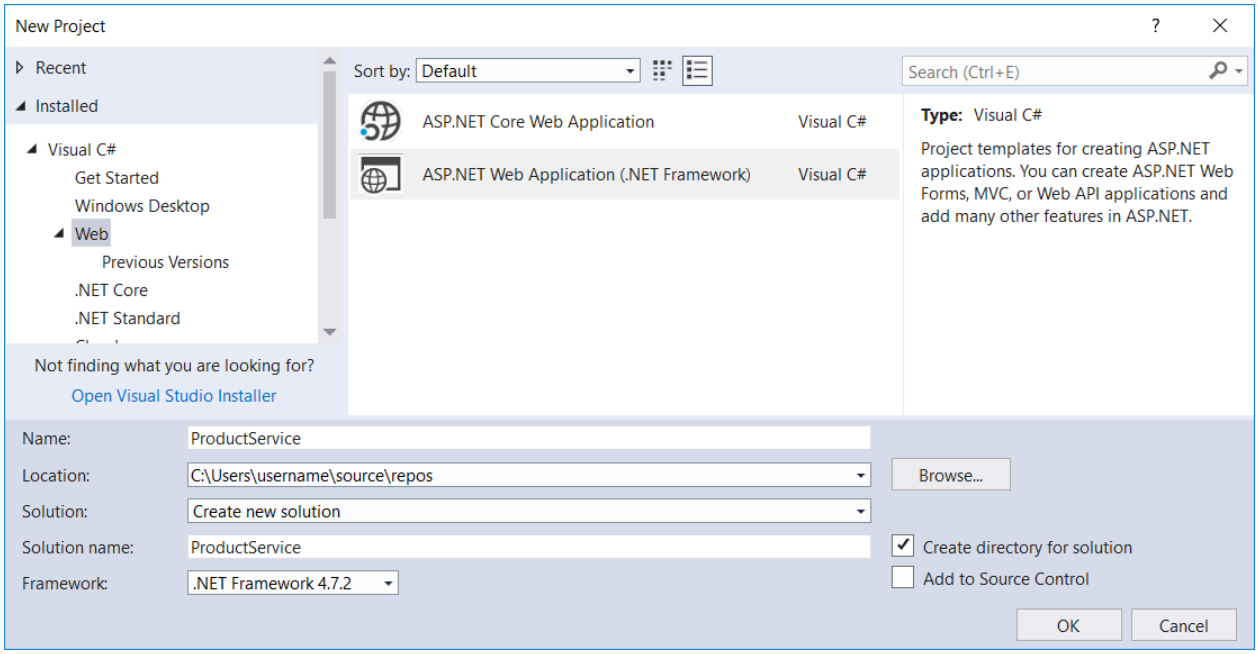
Tutorial versions

For the OData Version 3, see Creating an OData v3 Endpoint.

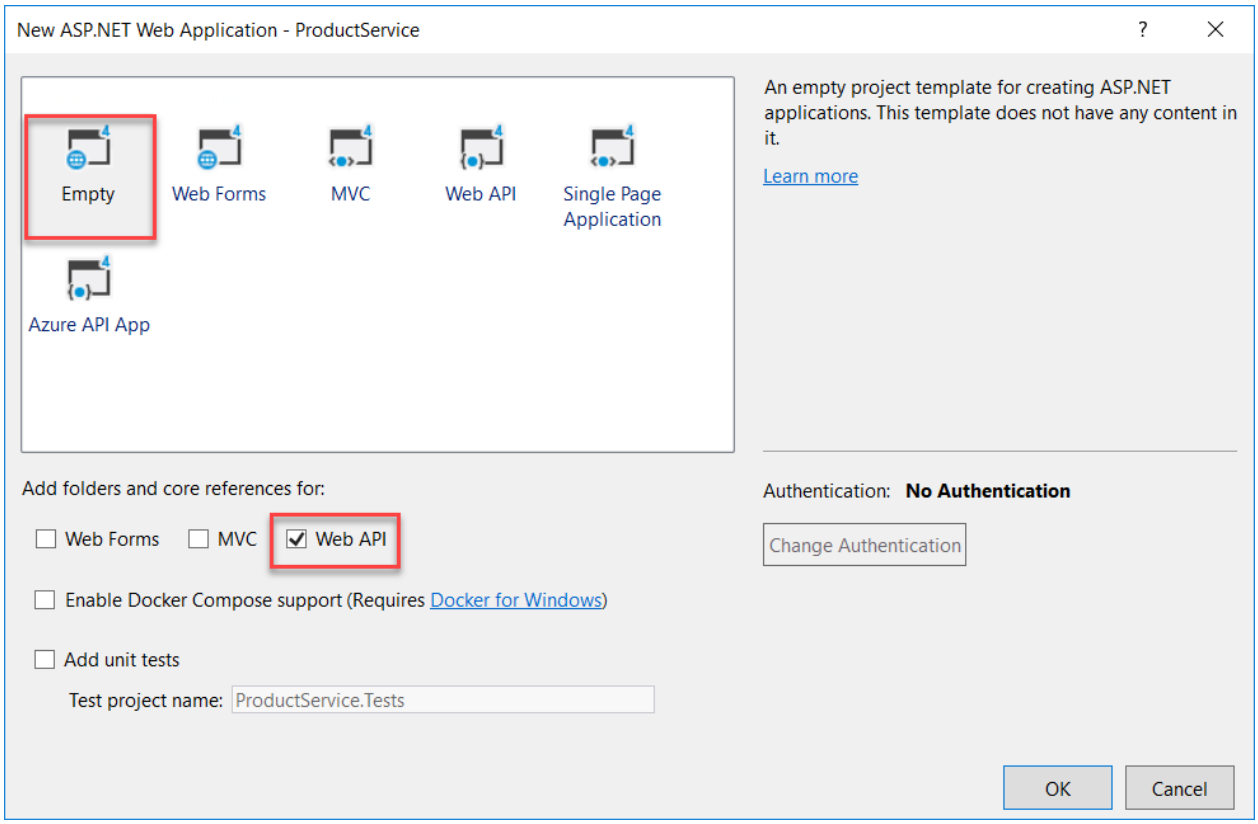
Create the Visual Studio Project

In Visual Studio, from the **File** menu, select **New** > **Project**.

Expand **Installed** > **Visual C#** > **Web**, and select the **ASP.NET Web Application (.NET Framework)** template. Name the project "ProductService".



Select **OK**.



Select the **Empty** template. Under **Add folders and core references for:**, select **Web API**. Select **OK**.

Install the OData packages

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**. In the Package Manager Console window, type:

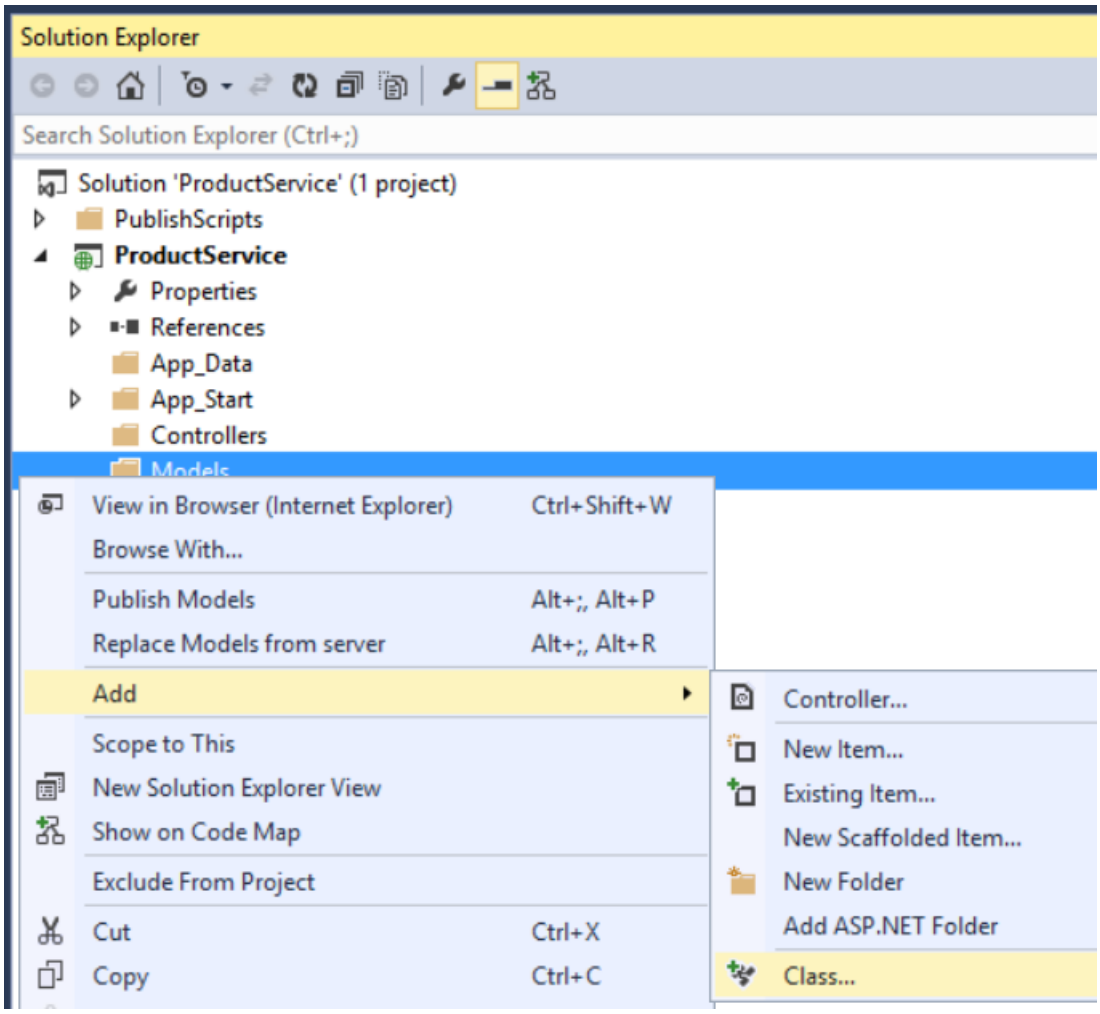
```
Install-Package Microsoft.AspNet.Odata
```

This command installs the latest OData NuGet packages.

Add a model class

A *model* is an object that represents a data entity in your application.

In Solution Explorer, right-click the Models folder. From the context menu, select **Add** > **Class**.



> **NOTE**
>
> By convention, model classes are placed in the Models folder, but you don't have to follow this convention in your own projects.

Name the class `Product`. In the Product.cs file, replace the boilerplate code with the following:

```
namespace ProductService.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
        public string Category { get; set; }
    }
}
```

The `Id` property is the entity key. Clients can query entities by key. For example, to get the product with ID of 5, the URI is `/Products(5)`. The `Id` property will also be the primary key in the back-end database.

Enable Entity Framework

For this tutorial, we'll use Entity Framework (EF) Code First to create the back-end database.

> **NOTE**
> Web API OData does not require EF. Use any data-access layer that can translate database entities into models.

First, install the NuGet package for EF. From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**. In the Package Manager Console window, type:

```
Install-Package EntityFramework
```

Open the Web.config file, and add the following section inside the **configuration** element, after the **configSections** element.

```
<configuration>
  <configSections>
    <!-- ... -->
  </configSections>

  <!-- Add this: -->
  <connectionStrings>
    <add name="ProductsContext" connectionString="Data Source=(localdb)\mssqllocaldb;
        Initial Catalog=ProductsContext; Integrated Security=True; MultipleActiveResultSets=True;
        AttachDbFilename=|DataDirectory|ProductsContext.mdf"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
```

This setting adds a connection string for a LocalDB database. This database will be used when you run the app locally.

Next, add a class named `ProductsContext` to the Models folder:

```
using System.Data.Entity;
namespace ProductService.Models
{
    public class ProductsContext : DbContext
    {
        public ProductsContext()
                : base("name=ProductsContext")
        {
        }
        public DbSet<Product> Products { get; set; }
    }
}
```

In the constructor, `"name=ProductsContext"` gives the name of the connection string.

Configure the OData endpoint

Open the file App_Start/WebApiConfig.cs. Add the following **using** statements:

```
using ProductService.Models;
using Microsoft.AspNet.OData.Builder;
using Microsoft.AspNet.OData.Extensions;
```

Then add the following code to the **Register** method:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // New code:
        ODataModelBuilder builder = new ODataConventionModelBuilder();
        builder.EntitySet<Product>("Products");
        config.MapODataServiceRoute(
            routeName: "ODataRoute",
            routePrefix: null,
            model: builder.GetEdmModel());
    }
}
```

This code does two things:

- Creates an Entity Data Model (EDM).
- Adds a route.

An EDM is an abstract model of the data. The EDM is used to create the service metadata document. The **ODataConventionModelBuilder** class creates an EDM by using default naming conventions. This approach requires the least code. If you want more control over the EDM, you can use the **ODataModelBuilder** class to create the EDM by adding properties, keys, and navigation properties explicitly.

A *route* tells Web API how to route HTTP requests to the endpoint. To create an OData v4 route, call the **MapODataServiceRoute** extension method.

If your application has multiple OData endpoints, create a separate route for each. Give each route a unique route name and prefix.

Add the OData controller

A *controller* is a class that handles HTTP requests. You create a separate controller for each entity set in your OData service. In this tutorial, you will create one controller, for the `Product` entity.

In Solution Explorer, right-click the Controllers folder and select **Add** > **Class**. Name the class `ProductsController`.

> **NOTE**
> The version of this tutorial for OData v3 uses the **Add Controller** scaffolding. Currently, there is no scaffolding for OData v4.

Replace the boilerplate code in ProductsController.cs with the following.

```
using ProductService.Models;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Net;
using System.Threading.Tasks;
using System.Web.Http;
using System.Web.OData;
namespace ProductService.Controllers
{
    public class ProductsController : ODataController
    {
        ProductsContext db = new ProductsContext();
        private bool ProductExists(int key)
        {
            return db.Products.Any(p => p.Id == key);
        }
        protected override void Dispose(bool disposing)
        {
            db.Dispose();
            base.Dispose(disposing);
        }
    }
}
```

The controller uses the `ProductsContext` class to access the database using EF. Notice that the controller overrides the **Dispose** method to dispose of the **ProductsContext**.

This is the starting point for the controller. Next, we'll add methods for all of the CRUD operations.

Query the entity set

Add the following methods to `ProductsController`.

```
[EnableQuery]
public IQueryable<Product> Get()
{
    return db.Products;
}
[EnableQuery]
public SingleResult<Product> Get([FromODataUri] int key)
{
    IQueryable<Product> result = db.Products.Where(p => p.Id == key);
    return SingleResult.Create(result);
}
```

The parameterless version of the `Get` method returns the entire Products collection. The `Get` method with a *key* parameter looks up a product by its key (in this case, the `Id` property).

The **[EnableQuery]** attribute enables clients to modify the query, by using query options such as $filter, $sort, and $page. For more information, see Supporting OData Query Options.

Add an entity to the entity set

To enable clients to add a new product to the database, add the following method to `ProductsController`.

```
public async Task<IHttpActionResult> Post(Product product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    db.Products.Add(product);
    await db.SaveChangesAsync();
    return Created(product);
}
```

Update an entity

OData supports two different semantics for updating an entity, PATCH and PUT.

- PATCH performs a partial update. The client specifies just the properties to update.
- PUT replaces the entire entity.

The disadvantage of PUT is that the client must send values for all of the properties in the entity, including values that are not changing. The OData spec states that PATCH is preferred.

In any case, here is the code for both PATCH and PUT methods:

```
public async Task<IHttpActionResult> Patch([FromODataUri] int key, Delta<Product> product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    var entity = await db.Products.FindAsync(key);
    if (entity == null)
    {
        return NotFound();
    }
    product.Patch(entity);
    try
    {
        await db.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!ProductExists(key))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }
    return Updated(entity);
}
public async Task<IHttpActionResult> Put([FromODataUri] int key, Product update)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    if (key != update.Id)
    {
        return BadRequest();
    }
    db.Entry(update).State = EntityState.Modified;
    try
    {
        await db.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!ProductExists(key))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }
    return Updated(update);
}
```

In the case of PATCH, the controller uses the **Delta<T>** type to track the changes.

Delete an entity

To enable clients to delete a product from the database, add the following method to `ProductsController`.

```
public async Task<IHttpActionResult> Delete([FromODataUri] int key)
{
    var product = await db.Products.FindAsync(key);
    if (product == null)
    {
        return NotFound();
    }
    db.Products.Remove(product);
    await db.SaveChangesAsync();
    return StatusCode(HttpStatusCode.NoContent);
}
```

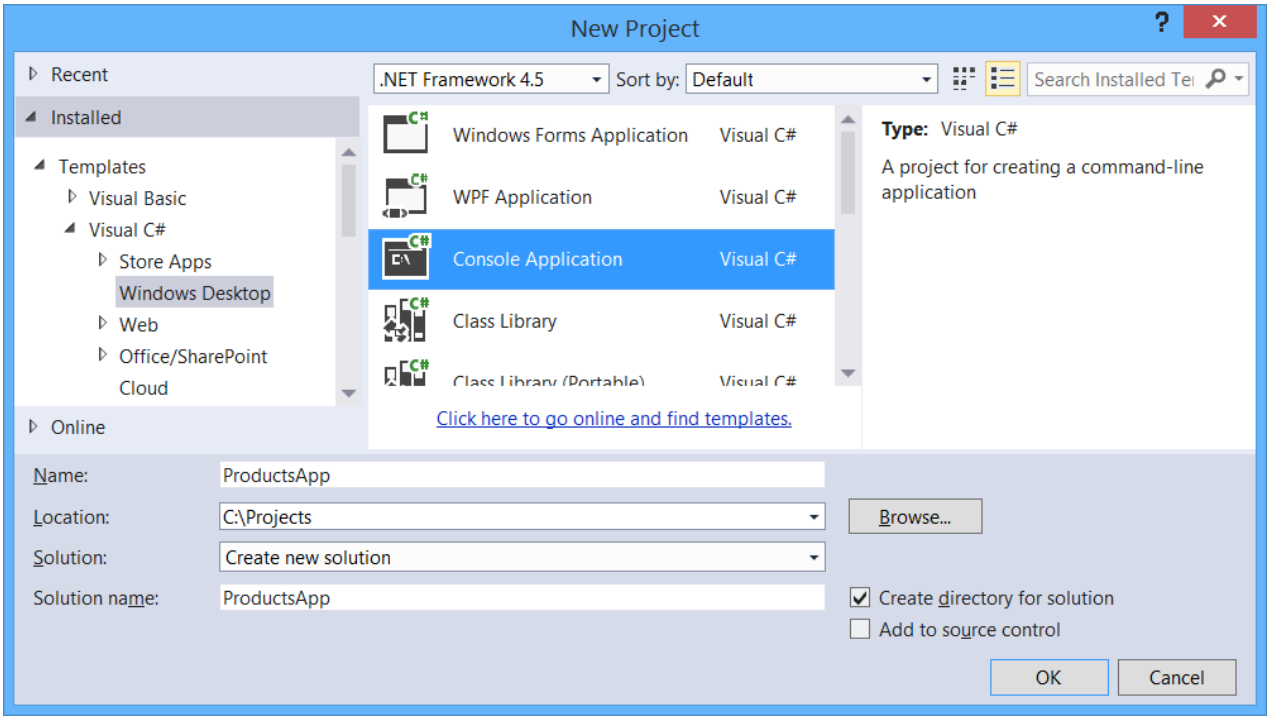
Create an OData v4 Client App (C#)

2/20/2020 • 2 minutes to read • Edit Online

by Mike Wasson

In the previous tutorial, you created a basic OData service that supports CRUD operations. Now let's create a client for the service.

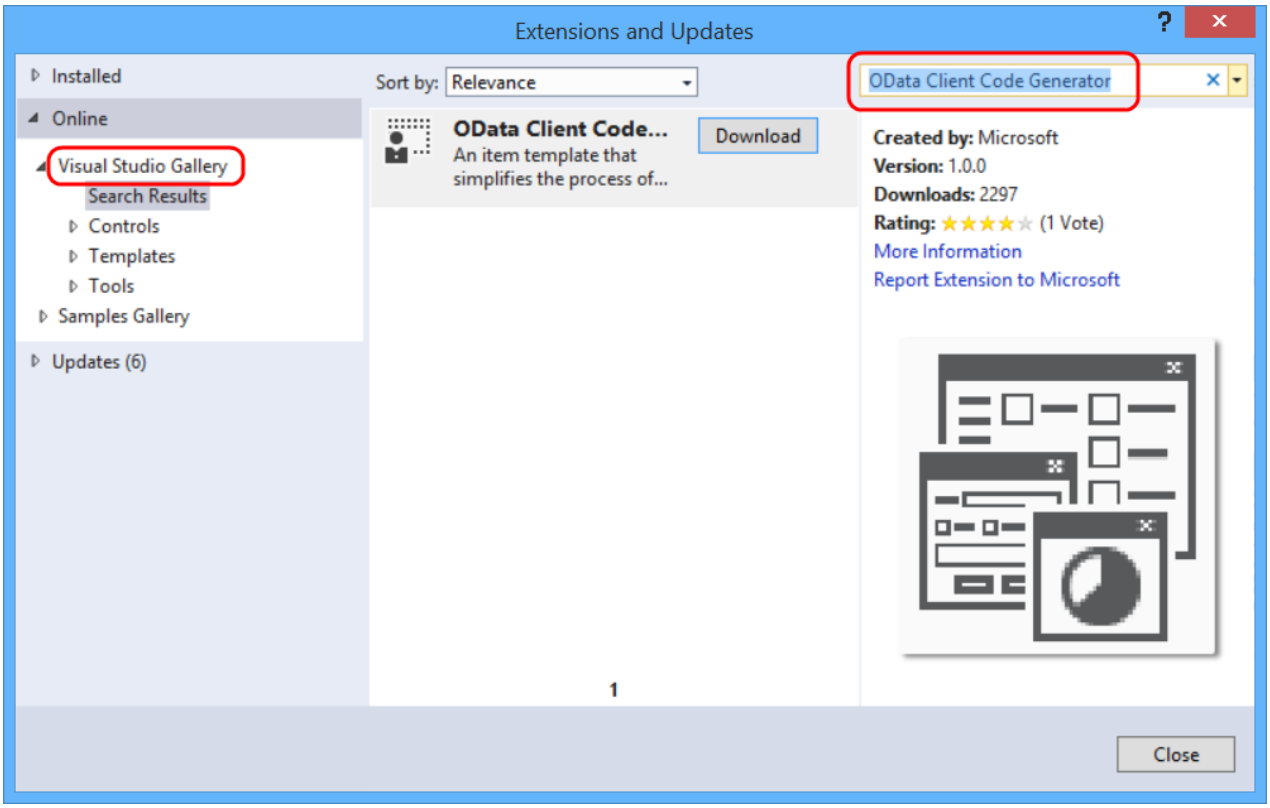
Start a new instance of Visual Studio and create a new console application project. In the **New Project** dialog, select **Installed** > **Templates** > **Visual C#** > **Windows Desktop**, and select the **Console Application** template. Name the project "ProductsApp".



> **NOTE**
>
> You can also add the console app to the same Visual Studio solution that contains the OData service.

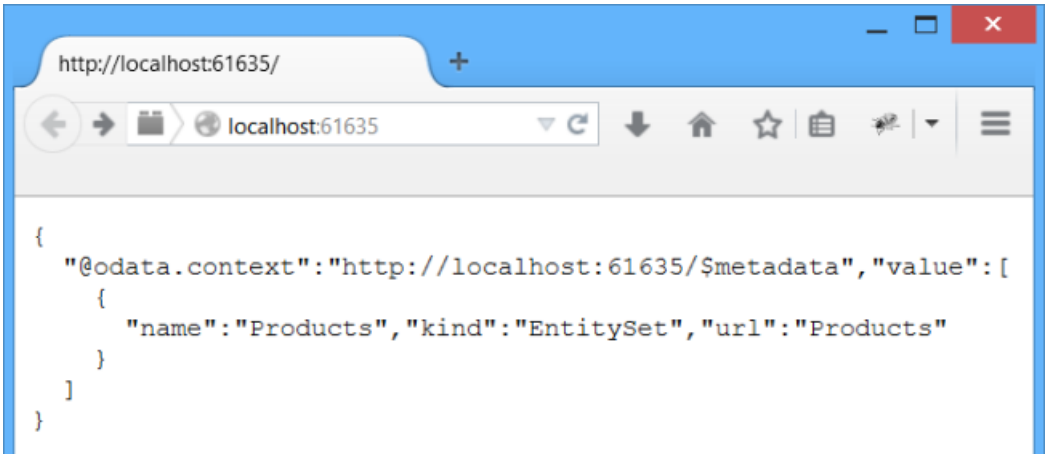
Install the OData Client Code Generator

From the **Tools** menu, select **Extensions and Updates**. Select **Online** > **Visual Studio Gallery**. In the search box, search for "OData Client Code Generator". Click **Download** to install the VSIX. You might be prompted to restart Visual Studio.



Run the OData Service Locally

Run the ProductService project from Visual Studio. By default, Visual Studio launches a browser to the application root. Note the URI; you will need this in the next step. Leave the application running.

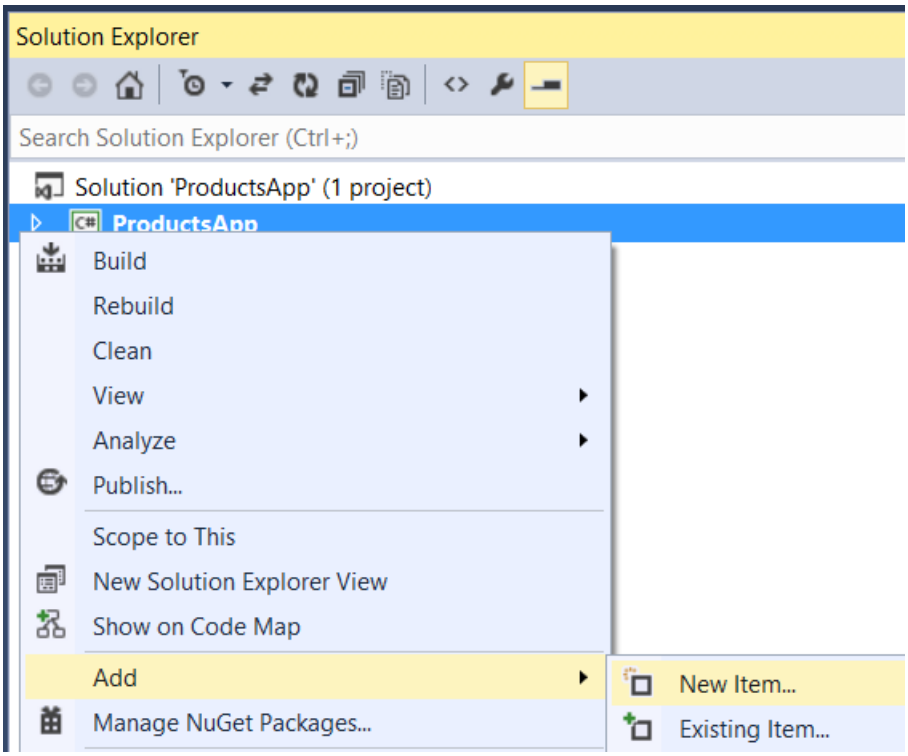


> **NOTE**
>
> If you put both projects in the same solution, make sure to run the ProductService project without debugging. In the next step, you will need to keep the service running while you modify the console application project.

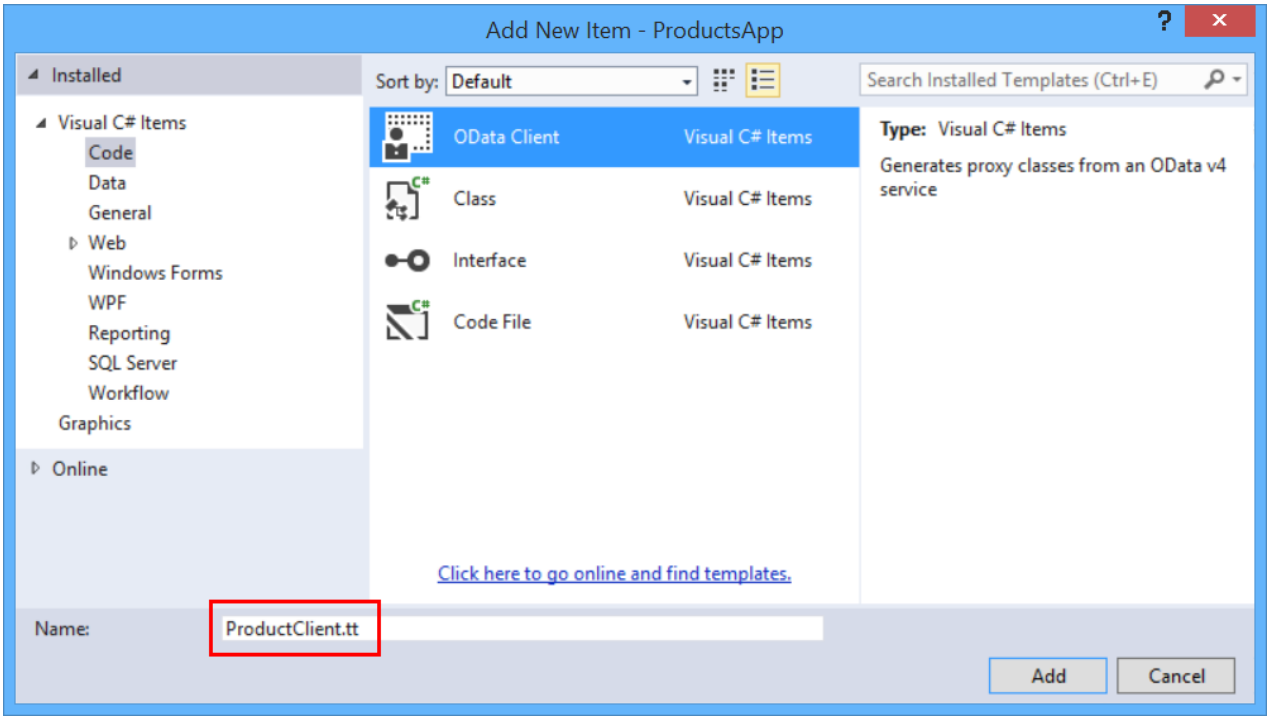
Generate the Service Proxy

The service proxy is a .NET class that defines methods for accessing the OData service. The proxy translates method calls into HTTP requests. You will create the proxy class by running a T4 template.

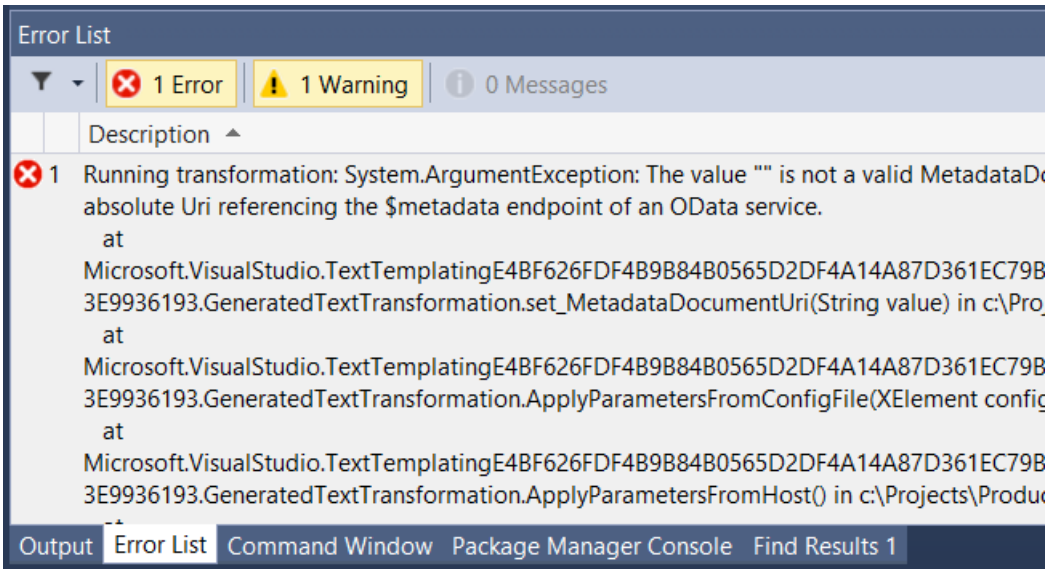
Right-click the project. Select **Add** > **New Item**.



In the **Add New Item** dialog, select **Visual C# Items** > **Code** > **OData Client**. Name the template "ProductClient.tt". Click **Add** and click through the security warning.

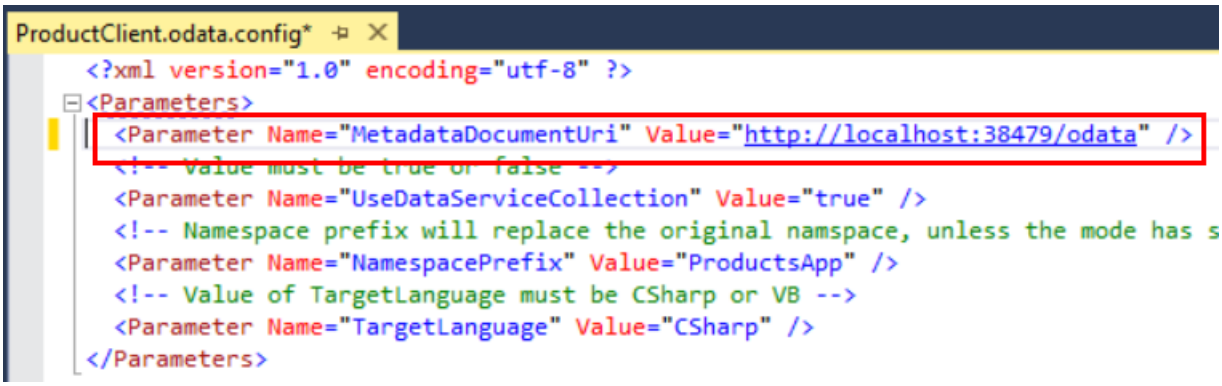


At this point, you'll get an error, which you can ignore. Visual Studio automatically runs the template, but the template needs some configuration settings first.



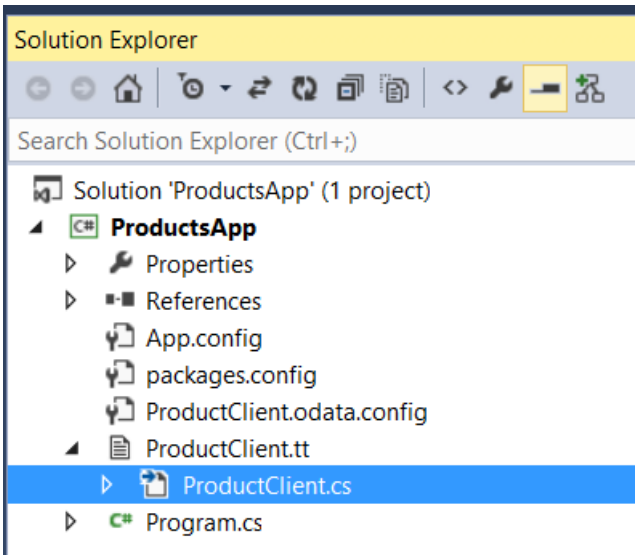
Open the file ProductClient.odata.config. In the `Parameter` element, paste in the URI from the ProductService project (previous step). For example:

```
<Parameter Name="MetadataDocumentUri" Value="http://localhost:61635/" />
```



Run the template again. In Solution Explorer, right click the ProductClient.tt file and select **Run Custom Tool**.

The template creates a code file named ProductClient.cs that defines the proxy. As you develop your app, if you change the OData endpoint, run the template again to update the proxy.



Use the Service Proxy to Call the OData Service

Open the file Program.cs and replace the boilerplate code with the following.

```
using System;

namespace ProductsApp
{
    class Program
    {
        // Get an entire entity set.
        static void ListAllProducts(Default.Container container)
        {
            foreach (var p in container.Products)
            {
                Console.WriteLine("{0} {1} {2}", p.Name, p.Price, p.Category);
            }
        }

        static void AddProduct(Default.Container container, ProductService.Models.Product product)
        {
            container.AddToProducts(product);
            var serviceResponse = container.SaveChanges();
            foreach (var operationResponse in serviceResponse)
            {
                Console.WriteLine("Response: {0}", operationResponse.StatusCode);
            }
        }

        static void Main(string[] args)
        {
            // TODO: Replace with your local URI.
            string serviceUri = "http://localhost:port/";
            var container = new Default.Container(new Uri(serviceUri));

            var product = new ProductService.Models.Product()
            {
                Name = "Yo-yo",
                Category = "Toys",
                Price = 4.95M
            };

            AddProduct(container, product);
            ListAllProducts(container);
        }
    }
}
```

Replace the value of *serviceUri* with the service URI from earlier.

```
// TODO: Replace with your local URI.
string serviceUri = "http://localhost:port/";
```

When you run the app, it should output the following:

```
Response: 201
Yo-yo 4.95 Toys
```

Entity Relations in OData v4 Using ASP.NET Web API 2.2

2/20/2020 • 7 minutes to read • Edit Online

by Mike Wasson

Most data sets define relations between entities: Customers have orders; books have authors; products have suppliers. Using OData, clients can navigate over entity relations. Given a product, you can find the supplier. You can also create or remove relationships. For example, you can set the supplier for a product.

This tutorial shows how to support these operations in OData v4 using ASP.NET Web API. The tutorial builds on the tutorial Create an OData v4 Endpoint Using ASP.NET Web API 2.

Software versions used in the tutorial

- Web API 2.1
- OData v4
- Visual Studio 2013 (download Visual Studio 2017 here)
- Entity Framework 6
- .NET 4.5

Tutorial versions

For the OData Version 3, see Supporting Entity Relations in OData v3.

Add a Supplier Entity

> **NOTE**
>
> The tutorial builds on the tutorial Create an OData v4 Endpoint Using ASP.NET Web API 2.

First, we need a related entity. Add a class named `Supplier` in the Models folder.

```
using System.Collections.Generic;

namespace ProductService.Models
{
    public class Supplier
    {
        public int Id { get; set; }
        public string Name { get; set; }

        public ICollection<Product> Products { get; set; }
    }
}
```

Add a navigation property to the `Product` class:

```
using System.ComponentModel.DataAnnotations.Schema;

namespace ProductService.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
        public string Category { get; set; }

        // New code:
        [ForeignKey("Supplier")]
        public int? SupplierId { get; set; }
        public virtual Supplier Supplier { get; set; }
    }
}
```

Add a new **DbSet** to the `ProductsContext` class, so that Entity Framework will include the Supplier table in the database.

```
public class ProductsContext : DbContext
{
    static ProductsContext()
    {
        Database.SetInitializer(new ProductInitializer());
    }

    public DbSet<Product> Products { get; set; }
    // New code:
    public DbSet<Supplier> Suppliers { get; set; }
}
```

In WebApiConfig.cs, add a "Suppliers" entity set to the entity data model:

```
public static void Register(HttpConfiguration config)
{
    ODataModelBuilder builder = new ODataConventionModelBuilder();
    builder.EntitySet<Product>("Products");
    // New code:
    builder.EntitySet<Supplier>("Suppliers");
    config.MapODataServiceRoute("ODataRoute", null, builder.GetEdmModel());
}
```

Add a Suppliers Controller

Add a `SuppliersController` class to the Controllers folder.

```
using ProductService.Models;
using System.Linq;
using System.Web.OData;

namespace ProductService.Controllers
{
    public class SuppliersController : ODataController
    {
        ProductsContext db = new ProductsContext();

        protected override void Dispose(bool disposing)
        {
            db.Dispose();
            base.Dispose(disposing);
        }
    }
}
```

I won't show how to add CRUD operations for this controller. The steps are the same as for the Products controller (see Create an OData v4 Endpoint).

Getting Related Entities

To get the supplier for a product, the client sends a GET request:

```
GET /Products(1)/Supplier
```

To support this request, add the following method to the `ProductsController` class:

```
public class ProductsController : ODataController
{
    // GET /Products(1)/Supplier
    [EnableQuery]
    public SingleResult<Supplier> GetSupplier([FromODataUri] int key)
    {
        var result = db.Products.Where(m => m.Id == key).Select(m => m.Supplier);
        return SingleResult.Create(result);
    }

   // Other controller methods not shown.
}
```

This method uses a default naming convention

- Method name: GetX, where X is the navigation property.
- Parameter name: *key*

If you follow this naming convention, Web API automatically maps the HTTP request to the controller method.

Example HTTP request:

```
GET http://myproductservice.example.com/Products(1)/Supplier HTTP/1.1
User-Agent: Fiddler
Host: myproductservice.example.com
```

Example HTTP response:

```
HTTP/1.1 200 OK
Content-Length: 125
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true
Server: Microsoft-IIS/8.0
OData-Version: 4.0
Date: Tue, 08 Jul 2014 00:44:27 GMT

{
  "@odata.context":"http://myproductservice.example.com/$metadata#Suppliers/$entity","Id":2,"Name":"Wingtip
Toys"
}
```

Getting a related collection

In the previous example, a product has one supplier. A navigation property can also return a collection. The following code gets the products for a supplier:

```
public class SuppliersController : ODataController
{
    // GET /Suppliers(1)/Products
    [EnableQuery]
    public IQueryable<Product> GetProducts([FromODataUri] int key)
    {
        return db.Suppliers.Where(m => m.Id.Equals(key)).SelectMany(m => m.Products);
    }

    // Other controller methods not shown.
}
```

In this case, the method returns an **IQueryable** instead of a **SingleResult<T>**

Example HTTP request:

```
GET http://myproductservice.example.com/Suppliers(2)/Products HTTP/1.1
User-Agent: Fiddler
Host: myproductservice.example.com
```

Example HTTP response:

```
HTTP/1.1 200 OK
Content-Length: 372
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true
Server: Microsoft-IIS/8.0
OData-Version: 4.0
Date: Tue, 08 Jul 2014 01:06:54 GMT

{
  "@odata.context":"http://myproductservice.example.com/$metadata#Products","value":[
    {
      "Id":1,"Name":"Hat","Price":14.95,"Category":"Clothing","SupplierId":2
    },{
      "Id":2,"Name":"Socks","Price":6.95,"Category":"Clothing","SupplierId":2
    },{
      "Id":4,"Name":"Pogo Stick","Price":29.99,"Category":"Toys","SupplierId":2
    }
  ]
}
```

Creating a Relationship Between Entities

OData supports creating or removing relationships between two existing entities. In OData v4 terminology, the relationship is a "reference". (In OData v3, the relationship was called a *link*. The protocol differences don't matter for this tutorial.)

A reference has its own URI, with the form `/Entity/NavigationProperty/$ref`. For example, here is the URI to address the reference between a product and its supplier:

```
http:/host/Products(1)/Supplier/$ref
```

To add a relationship, the client sends a POST or PUT request to this address.

- PUT if the navigation property is a single entity, such as `Product.Supplier`.
- POST if the navigation property is a collection, such as `Supplier.Products`.

The body of the request contains the URI of the other entity in the relation. Here is an example request:

```
PUT http://myproductservice.example.com/Products(6)/Supplier/$ref HTTP/1.1
OData-Version: 4.0;NetFx
OData-MaxVersion: 4.0;NetFx
Accept: application/json;odata.metadata=minimal
Accept-Charset: UTF-8
Content-Type: application/json;odata.metadata=minimal
User-Agent: Microsoft ADO.NET Data Services
Host: myproductservice.example.com
Content-Length: 70
Expect: 100-continue

{"@odata.id":"http://myproductservice.example.com/Suppliers(4)"}
```

In this example, the client sends a PUT request to `/Products(6)/Supplier/$ref`, which is the $ref URI for the `Supplier` of the product with ID = 6. If the request succeeds, the server sends a 204 (No Content) response:

```
HTTP/1.1 204 No Content
Server: Microsoft-IIS/8.0
Date: Tue, 08 Jul 2014 06:35:59 GMT
```

Here is the controller method to add a relationship to a `Product`:

```
public class ProductsController : ODataController
{
    [AcceptVerbs("POST", "PUT")]
    public async Task<IHttpActionResult> CreateRef([FromODataUri] int key,
        string navigationProperty, [FromBody] Uri link)
    {
        var product = await db.Products.SingleOrDefaultAsync(p => p.Id == key);
        if (product == null)
        {
            return NotFound();
        }
        switch (navigationProperty)
        {
            case "Supplier":
                // Note: The code for GetKeyFromUri is shown later in this topic.
                var relatedKey = Helpers.GetKeyFromUri<int>(Request, link);
                var supplier = await db.Suppliers.SingleOrDefaultAsync(f => f.Id == relatedKey);
                if (supplier == null)
                {
                    return NotFound();
                }

                product.Supplier = supplier;
                break;

            default:
                return StatusCode(HttpStatusCode.NotImplemented);
        }
        await db.SaveChangesAsync();
        return StatusCode(HttpStatusCode.NoContent);
    }

    // Other controller methods not shown.
}
```

The *navigationProperty* parameter specifies which relationship to set. (If there is more than one navigation property on the entity, you can add more `case` statements.)

The *link* parameter contains the URI of the supplier. Web API automatically parses the request body to get the value for this parameter.

To look up the supplier, we need the ID (or key), which is part of the *link* parameter. To do this, use the following helper method:

```
using Microsoft.OData.Core;
using Microsoft.OData.Core.UriParser;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Web.Http.Routing;
using System.Web.OData.Extensions;
using System.Web.OData.Routing;

namespace ProductService
{
    public static class Helpers
    {
        public static TKey GetKeyFromUri<TKey>(HttpRequestMessage request, Uri uri)
        {
            if (uri == null)
            {
                throw new ArgumentNullException("uri");
            }

            var urlHelper = request.GetUrlHelper() ?? new UrlHelper(request);

            string serviceRoot = urlHelper.CreateODataLink(
                request.ODataProperties().RouteName,
                request.ODataProperties().PathHandler, new List<ODataPathSegment>());
            var odataPath = request.ODataProperties().PathHandler.Parse(
                request.ODataProperties().Model,
                serviceRoot, uri.LocalPath);

            var keySegment = odataPath.Segments.OfType<KeyValuePathSegment>().FirstOrDefault();
            if (keySegment == null)
            {
                throw new InvalidOperationException("The link does not contain a key.");
            }

            var value = ODataUriUtils.ConvertFromUriLiteral(keySegment.Value, ODataVersion.V4);
            return (TKey)value;
        }

    }
}
```

Basically, this method uses the OData library to split the URI path into segments, find the segment that contains the key, and convert the key into the correct type.

Deleting a Relationship Between Entities

To delete a relationship, the client sends an HTTP DELETE request to the $ref URI:

```
DELETE http://host/Products(1)/Supplier/$ref
```

Here is the controller method to delete the relationship between a Product and a Supplier:

```
public class ProductsController : ODataController
{
    public async Task<IHttpActionResult> DeleteRef([FromODataUri] int key,
        string navigationProperty, [FromBody] Uri link)
    {
        var product = db.Products.SingleOrDefault(p => p.Id == key);
        if (product == null)
        {
            return NotFound();
        }

        switch (navigationProperty)
        {
            case "Supplier":
                product.Supplier = null;
                break;

            default:
                return StatusCode(HttpStatusCode.NotImplemented);
        }
        await db.SaveChangesAsync();

        return StatusCode(HttpStatusCode.NoContent);
    }

    // Other controller methods not shown.
}
```

In this case, `Product.Supplier` is the "1" end of a 1-to-many relation, so you can remove the relationship just by setting `Product.Supplier` to `null`.

In the "many" end of a relationship, the client must specify which related entity to remove. To do so, the client sends the URI of the related entity in the query string of the request. For example, to remove "Product 1" from "Supplier 1":

```
DELETE http://host/Suppliers(1)/Products/$ref?$id=http://host/Products(1)
```

To support this in Web API, we need to include an extra parameter in the `DeleteRef` method. Here is the controller method to remove a product from the `Supplier.Products` relation.

```
public class SuppliersController : ODataController
{
    public async Task<IHttpActionResult> DeleteRef([FromODataUri] int key,
        [FromODataUri] string relatedKey, string navigationProperty)
    {
        var supplier = await db.Suppliers.SingleOrDefaultAsync(p => p.Id == key);
        if (supplier == null)
        {
            return StatusCode(HttpStatusCode.NotFound);
        }

        switch (navigationProperty)
        {
            case "Products":
                var productId = Convert.ToInt32(relatedKey);
                var product = await db.Products.SingleOrDefaultAsync(p => p.Id == productId);

                if (product == null)
                {
                    return NotFound();
                }
                product.Supplier = null;
                break;
            default:
                return StatusCode(HttpStatusCode.NotImplemented);

        }
        await db.SaveChangesAsync();

        return StatusCode(HttpStatusCode.NoContent);
    }

    // Other controller methods not shown.
}
```

The *key* parameter is the key for the supplier, and the *relatedKey* parameter is the key for the product to remove from the `Products` relationship. Note that Web API automatically gets the key from the query string.

Actions and Functions in OData v4 Using ASP.NET Web API 2.2

2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson

In OData, actions and functions are a way to add server-side behaviors that are not easily defined as CRUD operations on entities. This tutorial shows how to add actions and functions to an OData v4 endpoint, using Web API 2.2. The tutorial builds on the tutorial Create an OData v4 Endpoint Using ASP.NET Web API 2

Software versions used in the tutorial

- Web API 2.2
- OData v4
- Visual Studio 2013 (download Visual Studio 2017 here)
- .NET 4.5

Tutorial versions

For OData Version 3, see OData Actions in ASP.NET Web API 2.

The difference between *actions* and *functions* is that actions can have side effects, and functions do not. Both actions and functions can return data. Some uses for actions include:

- Complex transactions.
- Manipulating several entities at once.
- Allowing updates only to certain properties of an entity.
- Sending data that is not an entity.

Functions are useful for returning information that does not correspond directly to an entity or collection.

An action (or function) can target a single entity or a collection. In OData terminology, this is the *binding*. You can also have "unbound" actions/functions, which are called as static operations on the service.

Example: Adding an Action

Let's define an action to rate a product.

> **NOTE**
>
> This tutorial builds on the tutorial Create an OData v4 Endpoint Using ASP.NET Web API 2

First, add a `ProductRating` model to represent the ratings.

```
namespace ProductService.Models
{
    public class ProductRating
    {
        public int ID { get; set; }
        public int Rating { get; set; }
        public int ProductID { get; set; }
        public virtual Product Product { get; set; }
    }
}
```

Also add a **DbSet** to the `ProductsContext` class, so that EF will create a Ratings table in the database.

```
public class ProductsContext : DbContext
{
    public ProductsContext()
            : base("name=ProductsContext")
    {
    }

    public DbSet<Product> Products { get; set; }
    public DbSet<Supplier> Suppliers { get; set; }
    // New code:
    public DbSet<ProductRating> Ratings { get; set; }
}
```

Add the Action to the EDM

In WebApiConfig.cs, add the following code:

```
ODataModelBuilder builder = new ODataConventionModelBuilder();
builder.EntitySet<Product>("Products");

// New code:
builder.Namespace = "ProductService";
builder.EntityType<Product>()
    .Action("Rate")
    .Parameter<int>("Rating");
```

The **EntityTypeConfiguration.Action** method adds an action to the entity data model (EDM). The **Parameter** method specifies a typed parameter for the action.

This code also sets the namespace for the EDM. The namespace matters because the URI for the action includes the fully-qualified action name:

```
http://localhost/Products(1)/ProductService.Rate
```

> **NOTE**
>
> In a typical IIS configuration, the dot in this URL will cause IIS to return error 404. You can resolve this by adding the following section to your Web.Config file:

```
<system.webServer>
    <handlers>
      <clear/>
      <add name="ExtensionlessUrlHandler-Integrated-4.0" path="/*"
          verb="*" type="System.Web.Handlers.TransferRequestHandler"
          preCondition="integratedMode,runtimeVersionv4.0" />
    </handlers>
</system.webServer>
```

Add a Controller Method for the Action

To enable the "Rate" action, add the following method to `ProductsController`:

```
[HttpPost]
public async Task<IHttpActionResult> Rate([FromODataUri] int key, ODataActionParameters parameters)
{
    if (!ModelState.IsValid)
    {
        return BadRequest();
    }

    int rating = (int)parameters["Rating"];
    db.Ratings.Add(new ProductRating
    {
        ProductID = key,
        Rating = rating
    });

    try
    {
        await db.SaveChangesAsync();
    }
    catch (DbUpdateException e)
    {
        if (!ProductExists(key))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return StatusCode(HttpStatusCode.NoContent);
}
```

Notice that the method name matches the action name. The **[HttpPost]** attribute specifies the method is an HTTP POST method.

To invoke the action, the client sends an HTTP POST request like the following:

```
POST http://localhost/Products(1)/ProductService.Rate HTTP/1.1
Content-Type: application/json
Content-Length: 12

{"Rating":5}
```

The "Rate" action is bound to Product instances, so the URI for the action is the fully-qualified action name appended to the entity URI. (Recall that we set the EDM namespace to "ProductService", so the fully-qualified action name is "ProductService.Rate".)

The body of the request contains the action parameters as a JSON payload. Web API automatically converts the JSON payload to an **ODataActionParameters** object, which is just a dictionary of parameter values. Use this dictionary to access the parameters in your controller method.

If the client sends the action parameters in the wrong format, the value of **ModelState.IsValid** is false. Check this flag in your controller method and return an error if **IsValid** is false.

```
if (!ModelState.IsValid)
{
    return BadRequest();
}
```

Example: Adding a Function

Now let's add an OData function that returns the most expensive product. As before, the first step is adding the function to the EDM. In WebApiConfig.cs, add the following code.

```
ODataModelBuilder builder = new ODataConventionModelBuilder();
builder.EntitySet<Product>("Products");
builder.EntitySet<Supplier>("Suppliers");

// New code:
builder.Namespace = "ProductService";
builder.EntityType<Product>().Collection
    .Function("MostExpensive")
    .Returns<double>();
```

In this case, the function is bound to the Products collection, rather than individual Product instances. Clients invoke the function by sending a GET request:

```
GET http://localhost:38479/Products/ProductService.MostExpensive
```

Here is the controller method for this function:

```
public class ProductsController : ODataController
{
    [HttpGet]
    public IHttpActionResult MostExpensive()
    {
        var product = db.Products.Max(x => x.Price);
        return Ok(product);
    }

    // Other controller methods not shown.
}
```

Notice that the method name matches the function name. The **[HttpGet]** attribute specifies the method is an HTTP GET method.

Here is the HTTP response:

```
HTTP/1.1 200 OK
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true
OData-Version: 4.0
Date: Sat, 28 Jun 2014 00:44:07 GMT
Content-Length: 85

{
  "@odata.context":"http://localhost:38479/$metadata#Edm.Decimal","value":50.00
}
```

Example: Adding an Unbound Function

The previous example was a function bound to a collection. In this next example, we'll create an *unbound* function. Unbound functions are called as static operations on the service. The function in this example will return the sales tax for a given postal code.

In the WebApiConfig file, add the function to the EDM:

```
ODataModelBuilder builder = new ODataConventionModelBuilder();
builder.EntitySet<Product>("Products");

// New code:
builder.Function("GetSalesTaxRate")
    .Returns<double>()
    .Parameter<int>("PostalCode");
```

Notice that we are calling **Function** directly on the **ODataModelBuilder**, instead of the entity type or collection. This tells the model builder that the function is unbound.

Here is the controller method that implements the function:

```
[HttpGet]
[ODataRoute("GetSalesTaxRate(PostalCode={postalCode})")]
public IHttpActionResult GetSalesTaxRate([FromODataUri] int postalCode)
{
    double rate = 5.6;  // Use a fake number for the sample.
    return Ok(rate);
}
```

It does not matter which Web API controller you place this method in. You could put it in `ProductsController`, or define a separate controller. The **[ODataRoute]** attribute defines the URI template for the function.

Here is an example client request:

```
GET http://localhost:38479/GetSalesTaxRate(PostalCode=10) HTTP/1.1
```

The HTTP response:

```
HTTP/1.1 200 OK
Content-Type: application/json; odata.metadata=minimal; odata.streaming=true
OData-Version: 4.0
Date: Sat, 28 Jun 2014 01:05:32 GMT
Content-Length: 82

{
  "@odata.context":"http://localhost:38479/$metadata#Edm.Double","value":5.6
}
```

Containment in OData v4 Using Web API 2.2

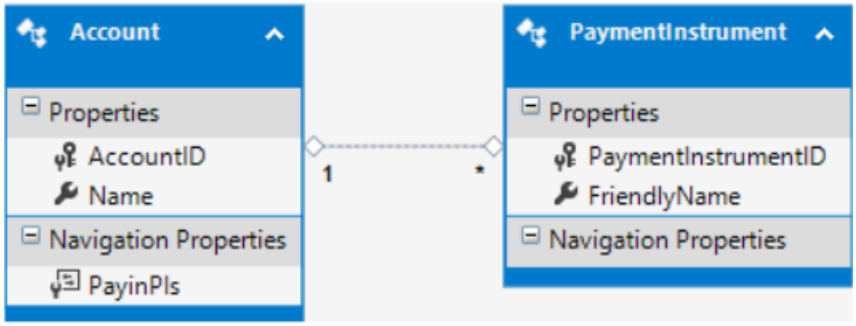
9/16/2020 • 3 minutes to read • Edit Online

by Jinfu Tan

> Traditionally, an entity could only be accessed if it were encapsulated inside an entity set. But OData v4 provides two additional options, Singleton and Containment, both of which WebAPI 2.2 supports.

This topic shows how to define a containment in an OData endpoint in WebApi 2.2. For more information about containment, see Containment is coming with OData v4. To create an OData V4 endpoint in Web API, see Create an OData v4 Endpoint Using ASP.NET Web API 2.2.

First, we'll create a containment domain model in the OData service, using this data model:



An account contains many PaymentInstruments (PI), but we don't define an entity set for a PI. Instead, the PIs can only be accessed through an Account.

You can download the solution used in this topic from CodePlex.

Defining the data model

1. Define the CLR types.

```
public class Account
{
    public int AccountID { get; set; }
    public string Name { get; set; }
    [Contained]
    public IList<PaymentInstrument> PayinPIs { get; set; }
}

public class PaymentInstrument
{
    public int PaymentInstrumentID { get; set; }
    public string FriendlyName { get; set; }
}
```

   The `Contained` attribute is used for containment navigation properties.

2. Generate the EDM model based on the CLR types.

```
public static IEdmModel GetModel()
{
    ODataConventionModelBuilder builder = new ODataConventionModelBuilder();
    builder.EntitySet<Account>("Accounts");
    var paymentInstrumentType = builder.EntityType<PaymentInstrument>();
    var functionConfiguration =
        paymentInstrumentType.Collection.Function("GetCount");
    functionConfiguration.Parameter<string>("NameContains");
    functionConfiguration.Returns<int>();
    builder.Namespace = typeof(Account).Namespace;
    return builder.GetEdmModel();
}
```

The `ODataConventionModelBuilder` will handle building the EDM model if the `Contained` attribute is added to the corresponding navigation property. If the property is a collection type, a `GetCount(string NameContains)` function will also be created.

The generated metadata will look like the following:

```
<EntityType Name="Account">
  <Key>
    <PropertyRef Name="AccountID" />
  </Key>
  <Property Name="AccountID" Type="Edm.Int32" Nullable="false" />
  <Property Name="Name" Type="Edm.String" />
  <NavigationProperty
    Name="PayinPIs"
    Type="Collection(ODataContrainmentSample.PaymentInstrument)"
    ContainsTarget="true" />
</EntityType>
```

The `ContainsTarget` attribute indicates that the navigation property is a containment.

Define the containing entity set controller

Contained entities don't have their own controller; the action is defined in the containing entity set controller. In this sample, there is an AccountsController, but no PaymentInstrumentsController.

```
public class AccountsController : ODataController
{
    private static IList<Account> _accounts = null;
    public AccountsController()
    {
        if (_accounts == null)
        {
            _accounts = InitAccounts();
        }
    }
    // PUT ~/Accounts(100)/PayinPIs
    [EnableQuery]
    public IHttpActionResult GetPayinPIs(int key)
    {
        var payinPIs = _accounts.Single(a => a.AccountID == key).PayinPIs;
        return Ok(payinPIs);
    }
    [EnableQuery]
    [ODataRoute("Accounts({accountId})/PayinPIs({paymentInstrumentId})")]
    public IHttpActionResult GetSinglePayinPI(int accountId, int paymentInstrumentId)
    {
        var payinPIs = _accounts.Single(a => a.AccountID == accountId).PayinPIs;
        var payinPI = payinPIs.Single(pi => pi.PaymentInstrumentID == paymentInstrumentId);
        return Ok(payinPI);
```

```
    }
    // PUT ~/Accounts(100)/PayinPIs(101)
    [ODataRoute("Accounts({accountId})/PayinPIs({paymentInstrumentId})")]
    public IHttpActionResult PutToPayinPI(int accountId, int paymentInstrumentId, [FromBody]PaymentInstrument
paymentInstrument)
    {
        var account = _accounts.Single(a => a.AccountID == accountId);
        var originalPi = account.PayinPIs.Single(p => p.PaymentInstrumentID == paymentInstrumentId);
        originalPi.FriendlyName = paymentInstrument.FriendlyName;
        return Ok(paymentInstrument);
    }
    // DELETE ~/Accounts(100)/PayinPIs(101)
    [ODataRoute("Accounts({accountId})/PayinPIs({paymentInstrumentId})")]
    public IHttpActionResult DeletePayinPIFromAccount(int accountId, int paymentInstrumentId)
    {
        var account = _accounts.Single(a => a.AccountID == accountId);
        var originalPi = account.PayinPIs.Single(p => p.PaymentInstrumentID == paymentInstrumentId);
        if (account.PayinPIs.Remove(originalPi))
        {
            return StatusCode(HttpStatusCode.NoContent);
        }
        else
        {
            return StatusCode(HttpStatusCode.InternalServerError);
        }
    }
    // GET ~/Accounts(100)/PayinPIs/Namespace.GetCount()
    [ODataRoute("Accounts({accountId})/PayinPIs/ODataContrainmentSample.GetCount(NameContains={name})")]
    public IHttpActionResult GetPayinPIsCountWhoseNameContainsGivenValue(int accountId, [FromODataUri]string
name)
    {
        var account = _accounts.Single(a => a.AccountID == accountId);
        var count = account.PayinPIs.Where(pi => pi.FriendlyName.Contains(name)).Count();
        return Ok(count);
    }
    private static IList<Account> InitAccounts()
    {
        var accounts = new List<Account>()
        {
            new Account()
            {
                AccountID = 100,
                Name="Name100",
                PayinPIs = new List<PaymentInstrument>()
                {
                    new PaymentInstrument()
                    {
                        PaymentInstrumentID = 101,
                        FriendlyName = "101 first PI",
                    },
                    new PaymentInstrument()
                    {
                        PaymentInstrumentID = 102,
                        FriendlyName = "102 second PI",
                    },
                },
            },
        };
        return accounts;
    }
}
```

If the OData path is 4 or more segments, only attribute routing works, such as `[ODataRoute("Accounts({accountId})/PayinPIs({paymentInstrumentId})")]` in the above controller. Otherwise, both attribute and conventional routing works: for instance, `GetPayInPIs(int key)` matches `GET ~/Accounts(1)/PayinPIs`.

Thanks to Leo Hu for the original content of this article.

Create a Singleton in OData v4 Using Web API 2.2

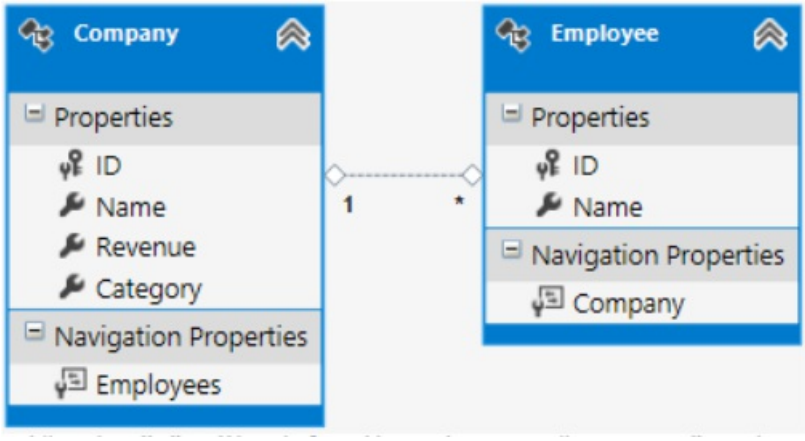
2/20/2020 • 3 minutes to read • Edit Online

by Zoe Luo

> Traditionally, an entity could only be accessed if it were encapsulated inside an entity set. But OData v4 provides two additional options, Singleton and Containment, both of which WebAPI 2.2 supports.

This article shows how to define a singleton in an OData endpoint in Web API 2.2. For information on what a singleton is and how you can benefit from using it, see Using a singleton to define your special entity. To create an OData V4 endpoint in Web API, see Create an OData v4 Endpoint Using ASP.NET Web API 2.2.

We'll create a singleton in your Web API project using the following data model:



A singleton named `Umbrella` will be defined based on type `Company`, and an entity set named `Employees` will be defined based on type `Employee`.

The solution used in this tutorial can be downloaded from CodePlex.

Define the data model

1. Define the CLR types.

```
/// <summary>
/// Present the EntityType "Employee"
/// </summary>
public class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }

    [Singleton]
    public Company Company { get; set; }
}
/// <summary>
/// Present company category, which is an enum type
/// </summary>
public enum CompanyCategory
{
    IT = 0,
    Communication = 1,
    Electronics = 2,
    Others = 3
}
/// <summary>
/// Present the EntityType "Company"
/// </summary>
public class Company
{
     public int ID { get; set; }
     public string Name { get; set; }
     public Int64 Revenue { get; set; }
     public CompanyCategory Category { get; set; }
     public List<Employee> Employees { get; set; }
}
```

2. Generate the EDM model based on the CLR types.

```
public static IEdmModel GetEdmModel()
{
    ODataConventionModelBuilder builder = new ODataConventionModelBuilder();
    builder.EntitySet<Employee>("Employees"); builder.Singleton<Company>("Umbrella");
    builder.Namespace = typeof(Company).Namespace;
    return builder.GetEdmModel();
}
```

Here, `builder.Singleton<Company>("Umbrella")` tells the model builder to create a singleton named `Umbrella` in the EDM model.

The generated metadata will look like the following:

```
<EntityContainer Name="Container">
  <EntitySet Name="Employees" EntityType="ODataSingletonSample.Employee">
    <NavigationPropertyBinding Path="Company" Target="Umbrella"/>
  </EntitySet>
  <Singleton Name="Umbrella" Type="ODataSingletonSample.Company">
    <NavigationPropertyBinding Path="Employees" Target="Employees"/>
  </Singleton>
</EntityContainer>
```

From the metadata we can see that the navigation property `Company` in the `Employees` entity set is bound to the singleton `Umbrella`. The binding is done automatically by `ODataConventionModelBuilder`, since only `Umbrella` has the `Company` type. If there is any ambiguity in the model, you can use `HasSingletonBinding` to explicitly bind a navigation property to a singleton; `HasSingletonBinding` has the same effect as using the

`Singleton` attribute in the CLR type definition:

```
EntitySetConfiguration<Employee> employeesConfiguration =
    builder.EntitySet<Employee>("Employees");
employeesConfiguration.HasSingletonBinding(c => c.Company, "Umbrella");
```

Define the singleton controller

Like the EntitySet controller, the singleton controller inherits from `ODataController`, and the singleton controller name should be `[singletonName]Controller`.

```
public class UmbrellaController : ODataController
{
    public static Company Umbrella;
    static UmbrellaController()
    {
        InitData();
    }
    private static void InitData()
    {
        Umbrella = new Company()
        {
            ID = 1,
            Name = "Umbrella",
            Revenue = 1000,
            Category = CompanyCategory.Communication,
            Employees = new List<Employee>()
        };
    }
}
```

In order to handle different kinds of requests, actions are required to be pre-defined in the controller. **Attribute routing** is enabled by default in WebApi 2.2. For example, to define an action to handle querying `Revenue` from `Company` using attribute routing, use the following:

```
[ODataRoute("Umbrella/Revenue")]
public IHttpActionResult GetCompanyRevenue()
{
     return Ok(Umbrella.Revenue);
}
```

If you are not willing to define attributes for each action, just define your actions following OData Routing Conventions. Since a key is not required for querying a singleton, the actions defined in the singleton controller are slightly different from actions defined in the entityset controller.

For reference, method signatures for every action definition in the singleton controller are listed below.

```
// Get Singleton
// ~/singleton
public IHttpActionResult Get()
public IHttpActionResult GetUmbrella()

// Get Singleton
// ~/singleton/cast
public IHttpActionResult GetFromSubCompany()
public IHttpActionResult GetUmbrellaFromSubCompany()

// Get Singleton Property
// ~/singleton/property
public IHttpActionResult GetName()
public IHttpActionResult GetNameFromCompany()

// Get Singleton Navigation Property
// ~/singleton/navigation
public IHttpActionResult GetEmployees()
public IHttpActionResult GetEmployeesFromCompany()

// Update singleton by PUT
// PUT ~/singleton
public IHttpActionResult Put(Company newCompany)
public IHttpActionResult PutUmbrella(Company newCompany)

// Update singleton by Patch
// PATCH ~/singleton
public IHttpActionResult Patch(Delta<Company> item)
public IHttpActionResult PatchUmbrella(Delta<Company> item)

// Add navigation link to singleton
// POST ~/singleton/navigation/$ref
public IHttpActionResult CreateRef(string navigationProperty, [FromBody] Uri link)

// Delete navigation link from singleton
// DELETE ~/singleton/navigation/$ref?$id=~/relatedKey
public IHttpActionResult DeleteRef(string relatedKey, string navigationProperty)

// Add a new entity to singleton navigation property
// POST ~/singleton/navigation
public IHttpActionResult PostToEmployees([FromBody] Employee employee)

// Call function bounded to singleton
// GET ~/singleton/function()
public IHttpActionResult GetEmployeesCount()
```

Basically, this is all you need to do on the service side. The sample project contains all of the code for the solution and the OData client that shows how to use the singleton. The client is built by following the steps in Create an OData v4 Client App.

.

Thanks to Leo Hu for the original content of this article.

Open Types in OData v4 with ASP.NET Web API

4/10/2020 • 5 minutes to read • Edit Online

by Microsoft

In OData v4, an *open type* is a structured type that contains dynamic properties, in addition to any properties that are declared in the type definition. Open types let you add flexibility to your data models. This tutorial shows how to use open types in ASP.NET Web API OData.

This tutorial assumes that you already know how to create an OData endpoint in ASP.NET Web API. If not, start by reading Create an OData v4 Endpoint first.

Software versions used in the tutorial

- Web API OData 5.3
- OData v4

First, some OData terminology:

- Entity type: A structured type with a key.
- Complex type: A structured type without a key.
- Open type: A type with dynamic properties. Both entity types and complex types can be open.

The value of a dynamic property can be a primitive type, complex type, or enumeration type; or a collection of any of those types. For more information about open types, see the OData v4 specification.

Install the Web OData Libraries

Use NuGet Package Manager to install the latest Web API OData libraries. From the Package Manager Console window:

```
Install-Package Microsoft.AspNet.OData
Install-Package Microsoft.AspNet.WebApi.OData
```

Define the CLR Types

Start by defining the EDM models as CLR types.

```
public enum Category
{
    Book,
    Magazine,
    EBook
}

public class Address
{
    public string City { get; set; }
    public string Street { get; set; }
}

public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Address Address { get; set; }
}

public class Press
{
    public string Name { get; set; }
    public string Email { get; set; }
    public Category Category { get; set; }
    public IDictionary<string, object> DynamicProperties { get; set; }
}

public class Book
{
    [Key]
    public string ISBN { get; set; }
    public string Title { get; set; }
    public Press Press { get; set; }
    public IDictionary<string, object> Properties { get; set; }
}
```

When the Entity Data Model (EDM) is created,

- `Category` is an enumeration type.
- `Address` is a complex type. (It does not have a key, so it is not an entity type.)
- `Customer` is an entity type. (It has a key.)
- `Press` is an open complex type.
- `Book` is an open entity type.

To create an open type, the CLR type must have a property of type `IDictionary<string, object>`, which holds the dynamic properties.

Build the EDM Model

If you use **ODataConventionModelBuilder** to create the EDM, `Press` and `Book` are automatically added as open types, based on the presence of a `IDictionary<string, object>` property.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        ODataConventionModelBuilder builder = new ODataConventionModelBuilder();
        builder.EntitySet<Book>("Books");
        builder.EntitySet<Customer>("Customers");
        var model = builder.GetEdmModel();

        config.MapODataServiceRoute(
            routeName: "ODataRoute",
            routePrefix: null,
            model: model);

    }
}
```

You can also build the EDM explicitly, using **ODataModelBuilder**.

```
ODataModelBuilder builder = new ODataModelBuilder();

ComplexTypeConfiguration<Press> pressType = builder.ComplexType<Press>();
pressType.Property(c => c.Name);
// ...
pressType.HasDynamicProperties(c => c.DynamicProperties);

EntityTypeConfiguration<Book> bookType = builder.EntityType<Book>();
bookType.HasKey(c => c.ISBN);
bookType.Property(c => c.Title);
// ...
bookType.ComplexProperty(c => c.Press);
bookType.HasDynamicProperties(c => c.Properties);

// ...
builder.EntitySet<Book>("Books");
IEdmModel model = builder.GetEdmModel();
```

Add an OData Controller

Next, add an OData controller. For this tutorial, we'll use a simplified controller that just supports GET and POST requests, and uses an in-memory list to store entities.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;
using System.Web.OData;

namespace MyApp.Controllers
{
    public class BooksController : ODataController
    {
        private IList<Book> _books = new List<Book>
        {
            new Book
            {
                ISBN = "978-0-7356-8383-9",
                Title = "SignalR Programming in Microsoft ASP.NET",
                Press = new Press
                {
                    Name = "Microsoft Press",
                    Category = Category.Book
                }
```

```
            },

            new Book
            {
                ISBN = "978-0-7356-7942-9",
                Title = "Microsoft Azure SQL Database Step by Step",
                Press = new Press
                {
                    Name = "Microsoft Press",
                    Category = Category.EBook,
                    DynamicProperties = new Dictionary<string, object>
                    {
                        { "Blog", "https://blogs.msdn.com/b/microsoft_press/" },
                        { "Address", new Address {
                              City = "Redmond", Street = "One Microsoft Way" }
                        }
                    }
                },
                Properties = new Dictionary<string, object>
                {
                    { "Published", new DateTimeOffset(2014, 7, 3, 0, 0, 0, 0, new TimeSpan(0))},
                    { "Authors", new [] { "Leonard G. Lobel", "Eric D. Boyd" }},
                    { "OtherCategories", new [] {Category.Book, Category.Magazine}}
                }
            }
        };

        [EnableQuery]
        public IQueryable<Book> Get()
        {
            return _books.AsQueryable();
        }

        public IHttpActionResult Get([FromODataUri]string key)
        {
            Book book = _books.FirstOrDefault(e => e.ISBN == key);
            if (book == null)
            {
                return NotFound();
            }

            return Ok(book);
        }

        public IHttpActionResult Post(Book book)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            // For this sample, we aren't enforcing unique keys.
            _books.Add(book);
            return Created(book);
        }
    }
}
```

Notice that the first `Book` instance has no dynamic properties. The second `Book` instance has the following dynamic properties:

- "Published": Primitive type
- "Authors": Collection of primitive types
- "OtherCategories": Collection of enumeration types.

Also, the `Press` property of that `Book` instance has the following dynamic properties:

- "Blog": Primitive type
- "Address": Complex type

Query the Metadata

To get the OData metadata document, send a GET request to `~/$metadata`. The response body should look similar to this:

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="4.0" xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx">
  <edmx:DataServices>
    <Schema Namespace="MyApp.Models" xmlns="http://docs.oasis-open.org/odata/ns/edm">
      <EntityType Name="Book" OpenType="true">
        <Key>
          <PropertyRef Name="ISBN" />
        </Key>
        <Property Name="ISBN" Type="Edm.String" Nullable="false" />
        <Property Name="Title" Type="Edm.String" />
        <Property Name="Press" Type="MyApp.Models.Press" />
      </EntityType>
      <EntityType Name="Customer">
        <Key>
          <PropertyRef Name="Id" />
        </Key>
        <Property Name="Id" Type="Edm.Int32" Nullable="false" />
        <Property Name="Name" Type="Edm.String" />
        <Property Name="Address" Type="MyApp.Models.Address" />
      </EntityType>
      <ComplexType Name="Press" OpenType="true">
        <Property Name="Name" Type="Edm.String" />
        <Property Name="Category" Type="MyApp.Models.Category" Nullable="false" />
      </ComplexType>
      <ComplexType Name="Address">
        <Property Name="City" Type="Edm.String" />
        <Property Name="Street" Type="Edm.String" />
      </ComplexType>
      <EnumType Name="Category">
        <Member Name="Book" Value="0" />
        <Member Name="Magazine" Value="1" />
        <Member Name="EBook" Value="2" />
      </EnumType>
    </Schema>
    <Schema Namespace="Default" xmlns="http://docs.oasis-open.org/odata/ns/edm">
      <EntityContainer Name="Container">
        <EntitySet Name="Books" EntityType="MyApp.Models.Book" />
        <EntitySet Name="Customers" EntityType="MyApp.Models.Customer" />
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

From the metadata document, you can see that:

- For the `Book` and `Press` types, the value of the `OpenType` attribute is true. The `Customer` and `Address` types don't have this attribute.
- The `Book` entity type has three declared properties: ISBN, Title, and Press. The OData metadata does not include the `Book.Properties` property from the CLR class.
- Similarly, the `Press` complex type has only two declared properties: Name and Category. The metadata does not include the `Press.DynamicProperties` property from the CLR class.

Query an Entity

To get the book with ISBN equal to "978-0-7356-7942-9", send a GET request to `~/Books('978-0-7356-7942-9')`. The response body should look similar to the following. (Indented to make it more readable.)

```
{
  "@odata.context":"http://localhost:37141/$metadata#Books/$entity",
    "ISBN":"978-0-7356-7942-9",
    "Title":"Microsoft Azure SQL Database Step by Step",
    "Press":{
      "Name":"Microsoft Press",
      "Category":"EBook",
      "Blog":"https://blogs.msdn.com/b/microsoft_press/",
      "Address":{
        "@odata.type":"#MyApp.Models.Address",
        "City":"Redmond",
        "Street":"One Microsoft Way"
      }
  },
  "Published":"2014-07-03T00:00:00Z",
  "Authors@odata.type":"#Collection(String)",
  "Authors":[
    "Leonard G. Lobel","Eric D. Boyd"
  ],
  "OtherCategories@odata.type":"#Collection(MyApp.Models.Category)",
  "OtherCategories":[
    "Book","Magazine"
  ]
}
```

Notice that the dynamic properties are included inline with the declared properties.

POST an Entity

To add a Book entity, send a POST request to `~/Books`. The client can set dynamic properties in the request payload.

Here is an example request. Note the "Price" and "Published" properties.

```
POST http://localhost:37141/Books HTTP/1.1
User-Agent: Fiddler
Host: localhost:37141
Content-Type: application/json
Content-Length: 191

{
  "ISBN":"978-0-7356-8383-9","Title":"Programming Microsoft ASP.NET MVC","Press":{
  "Name":"Microsoft Press","Category":"Book"
   }, "Price": 49.99, "Published":"2014-02-15T00:00:00Z"
}
```

If you set a breakpoint in the controller method, you can see that Web API added these properties to the `Properties` dictionary.

```
public IHttpActionResult Post(Book book)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    _books.Add(book);
    return Create
}

}
}
```

book {WebApplication1.Models.Book}
ISBN "978-0-7356-8383-9"
Press {WebApplication1.Models.Press}
Properties Count = 2
[0] {[Price, 49.99]}
[1] {[Published, 2014-02-15T00:00:00Z]}
Raw View

Additional Resources

OData Open Type Sample

Complex Type Inheritance in OData v4 with ASP.NET Web API

4/10/2020 • 3 minutes to read • Edit Online

by Microsoft

According to the OData v4 specification, a complex type can inherit from another complex type. (A *complex* type is a structured type without a key.) Web API OData 5.3 supports complex type inheritance.

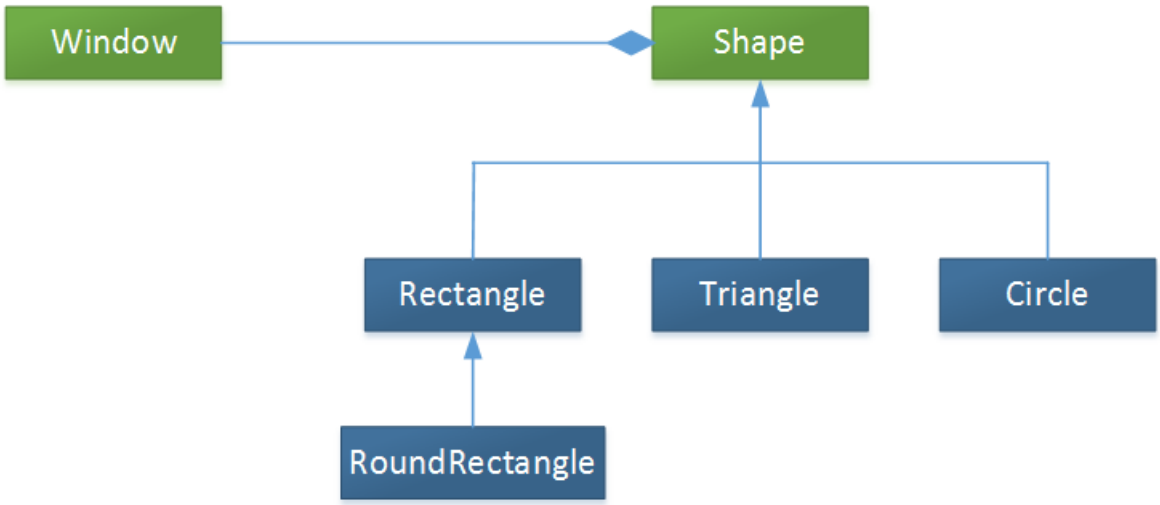
This topic shows how to build an entity data model (EDM) with complex inheritance types. For the complete source code, see OData Complex Type Inheritance Sample.

Software versions used in the tutorial

- Web API OData 5.3
- OData v4

Model Hierarchy

To illustrate complex type inheritance, we'll use the following class hierarchy.



`Shape` is an abstract complex type. `Rectangle`, `Triangle`, and `Circle` are complex types derived from `Shape`, and `RoundRectangle` derives from `Rectangle`. `Window` is an entity type and contains a `Shape` instance.

Here are the CLR classes that define these types.

```
public class Window
{
    public int Id { get; set; }
    public string Title { get; set; }
    public Shape Shape { get; set; }
}

public abstract class Shape
{
    public bool HasBorder { get; set; }
    public Color Color { get; set; }
}

public class Rectangle : Shape
{
    public Point LeftTop { get; set; }
    public int Height { get; set; }
    public int Weight { get; set; }
}

public class RoundRectangle : Rectangle
{
    public double Round { get; set; }
}

public class Triangle : Shape
{
    public Point P1 { get; set; }
    public Point P2 { get; set; }
    public Point P3 { get; set; }
}

public class Circle : Shape
{
    public Point Center { get; set; }
    public int Radius { get; set; }
}

public class Point
{
    public int X { get; set; }
    public int Y { get; set; }
}

public enum Color
{
    Red,
    Blue,
    Green,
    Yellow
}
```

Build the EDM Model

To create the EDM, you can use **ODataConventionModelBuilder**, which infers the inheritance relationships from the CLR types.

```
private IEdmModel GetEdmModel()
{
    ODataConventionModelBuilder builder = new ODataConventionModelBuilder();
    builder.EntitySet<Window>("Windows");
    return builder.GetEdmModel();
}
```

You can also build the EDM explicitly, using **ODataModelBuilder**. This takes more code, but gives you more control over the EDM.

```
private IEdmModel GetExplicitEdmModel()
{
  ODataModelBuilder builder = new ODataModelBuilder();

  EnumTypeConfiguration<Color> color = builder.EnumType<Color>();
  color.Member(Color.Red);
  color.Member(Color.Blue);
  color.Member(Color.Green);
  color.Member(Color.Yellow);

  ComplexTypeConfiguration<Point> point = builder.ComplexType<Point>();
  point.Property(c => c.X);
  point.Property(c => c.Y);

  ComplexTypeConfiguration<Shape> shape = builder.ComplexType<Shape>();
  shape.EnumProperty(c => c.Color);
  shape.Property(c => c.HasBorder);
  shape.Abstract();

  ComplexTypeConfiguration<Triangle> triangle = builder.ComplexType<Triangle>();
    triangle.ComplexProperty(c => c.P1);
    triangle.ComplexProperty(c => c.P2);
    triangle.ComplexProperty(c => c.P2);
    triangle.DerivesFrom<Shape>();

    ComplexTypeConfiguration<Rectangle> rectangle = builder.ComplexType<Rectangle>();
    rectangle.ComplexProperty(c => c.LeftTop);
    rectangle.Property(c => c.Height);
    rectangle.Property(c => c.Weight);
    rectangle.DerivesFrom<Shape>();

  ComplexTypeConfiguration<RoundRectangle> roundRectangle = builder.ComplexType<RoundRectangle>();
    roundRectangle.Property(c => c.Round);
    roundRectangle.DerivesFrom<Rectangle>();

    ComplexTypeConfiguration<Circle> circle = builder.ComplexType<Circle>();
    circle.ComplexProperty(c => c.Center);
    circle.Property(c => c.Radius);
    circle.DerivesFrom<Shape>();

    EntityTypeConfiguration<Window> window = builder.EntityType<Window>();
    window.HasKey(c => c.Id);
    window.Property(c => c.Title);
    window.ComplexProperty(c => c.Shape);

    builder.EntitySet<Window>("Windows");
    return builder.GetEdmModel();
}
```

These two examples create the same EDM schema.

Metadata Document

Here is the OData metadata document, showing complex type inheritance.

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="4.0" xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx">
  <edmx:DataServices>
    <Schema Namespace="NS" xmlns="http://docs.oasis-open.org/odata/ns/edm">
      <EntityType Name="Window">
        <Key>
          <PropertyRef Name="Id" />
        </Key>
        <Property Name="Id" Type="Edm.Int32" Nullable="false" />
        <Property Name="Title" Type="Edm.String" />
        <Property Name="Shape" Type="BookStore.Shape" />
      </EntityType>
      <ComplexType Name="Shape" Abstract="true">
        <Property Name="HasBorder" Type="Edm.Boolean" Nullable="false" />
        <Property Name="Color" Type="BookStore.Color" Nullable="false" />
      </ComplexType>
      <ComplexType Name="Circle" BaseType="BookStore.Shape">
        <Property Name="Center" Type="BookStore.Point" />
        <Property Name="Radius" Type="Edm.Int32" Nullable="false" />
      </ComplexType>
      <ComplexType Name="Point">
        <Property Name="X" Type="Edm.Int32" Nullable="false" />
        <Property Name="Y" Type="Edm.Int32" Nullable="false" />
      </ComplexType>
      <ComplexType Name="Rectangle" BaseType="BookStore.Shape">
        <Property Name="LeftTop" Type="BookStore.Point" />
        <Property Name="Height" Type="Edm.Int32" Nullable="false" />
        <Property Name="Weight" Type="Edm.Int32" Nullable="false" />
      </ComplexType>
      <ComplexType Name="RoundRectangle" BaseType="BookStore.Rectangle">
        <Property Name="Round" Type="Edm.Double" Nullable="false" />
      </ComplexType>
      <ComplexType Name="Triangle" BaseType="BookStore.Shape">
        <Property Name="P1" Type="BookStore.Point" />
        <Property Name="P2" Type="BookStore.Point" />
        <Property Name="P3" Type="BookStore.Point" />
      </ComplexType>
      <EnumType Name="Color">
        <Member Name="Red" Value="0" />
        <Member Name="Blue" Value="1" />
        <Member Name="Green" Value="2" />
        <Member Name="Yellow" Value="3" />
      </EnumType>
    </Schema>
    <Schema Namespace="Default" xmlns="http://docs.oasis-open.org/odata/ns/edm">
      <EntityContainer Name="Container">
        <EntitySet Name="Windows" EntityType="BookStore.Window" />
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

From the metadata document, you can see that:

- The `Shape` complex type is abstract.
- The `Rectangle`, `Triangle`, and `Circle` complex type have the base type `Shape`.
- The `RoundRectangle` type has the base type `Rectangle`.

Casting Complex Types

Casting on complex types is now supported. For example, the following query casts a `Shape` to a `Rectangle`.

```
GET ~/odata/Windows(1)/Shape/NS.Rectangle/LeftTop
```

Here's the response payload:

```
{
   "@odata.context":"http://localhost/odata/$metadata#Windows(1)/Shape/NS.Rectangle/LeftTop",
    "X":100,"Y":100
}
```

Supporting OData v3 in ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

Use ASP.NET Web API to create endpoints that support OData Version 3.

- Creating an OData v3 Endpoint
- Supporting Entity Relations in OData v3
- OData Actions in Web API 2
- Calling an OData Service From a .NET Client (C#)

Creating an OData v3 Endpoint with Web API 2

2/20/2020 • 10 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

The Open Data Protocol (OData) is a data access protocol for the web. OData provides a uniform way to structure data, query the data, and manipulate the data set through CRUD operations (create, read, update, and delete). OData supports both AtomPub (XML) and JSON formats. OData also defines a way to expose metadata about the data. Clients can use the metadata to discover the type information and relationships for the data set.

ASP.NET Web API makes it easy to create an OData endpoint for a data set. You can control exactly which OData operations the endpoint supports. You can host multiple OData endpoints, alongside non-OData endpoints. You have full control over your data model, back-end business logic, and data layer.

Software versions used in the tutorial

- Visual Studio 2013
- Web API 2
- OData Version 3
- Entity Framework 6
- Fiddler Web Debugging Proxy (Optional)

Web API OData support was added in ASP.NET and Web Tools 2012.2 Update. However, this tutorial uses scaffolding that was added in Visual Studio 2013.

In this tutorial, you will create a simple OData endpoint that clients can query. You will also create a C# client for the endpoint. After you complete this tutorial, the next set of tutorials show how to add more functionality, including entity relations, actions, and $expand/$select.

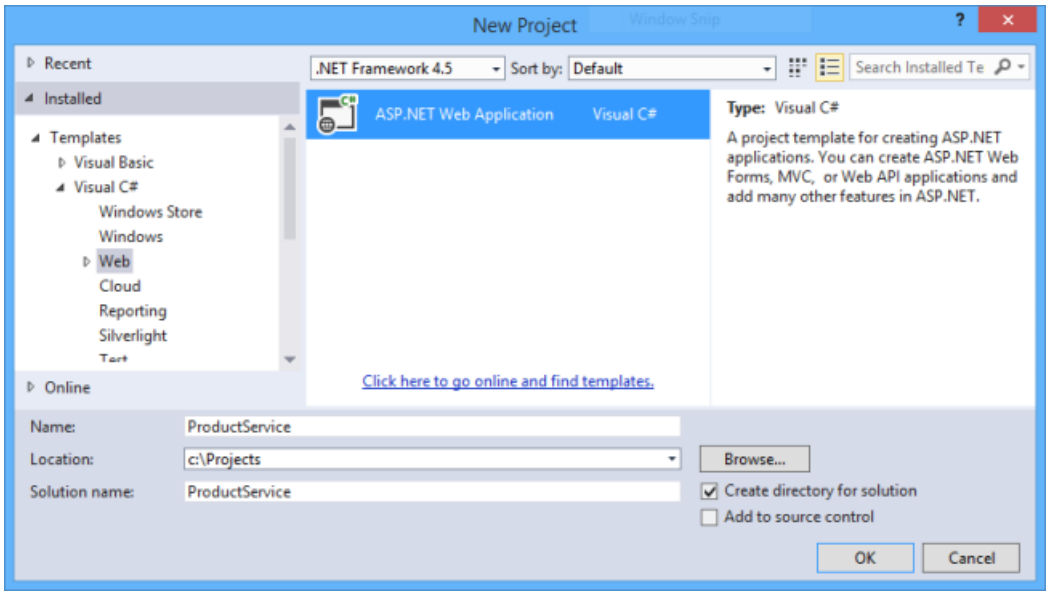
- Create the Visual Studio Project
- Add an Entity Model
- Add an OData Controller
- Add the EDM and Route
- Seed the Database (Optional)
- Exploring the OData Endpoint
- OData Serialization Formats

Create the Visual Studio Project

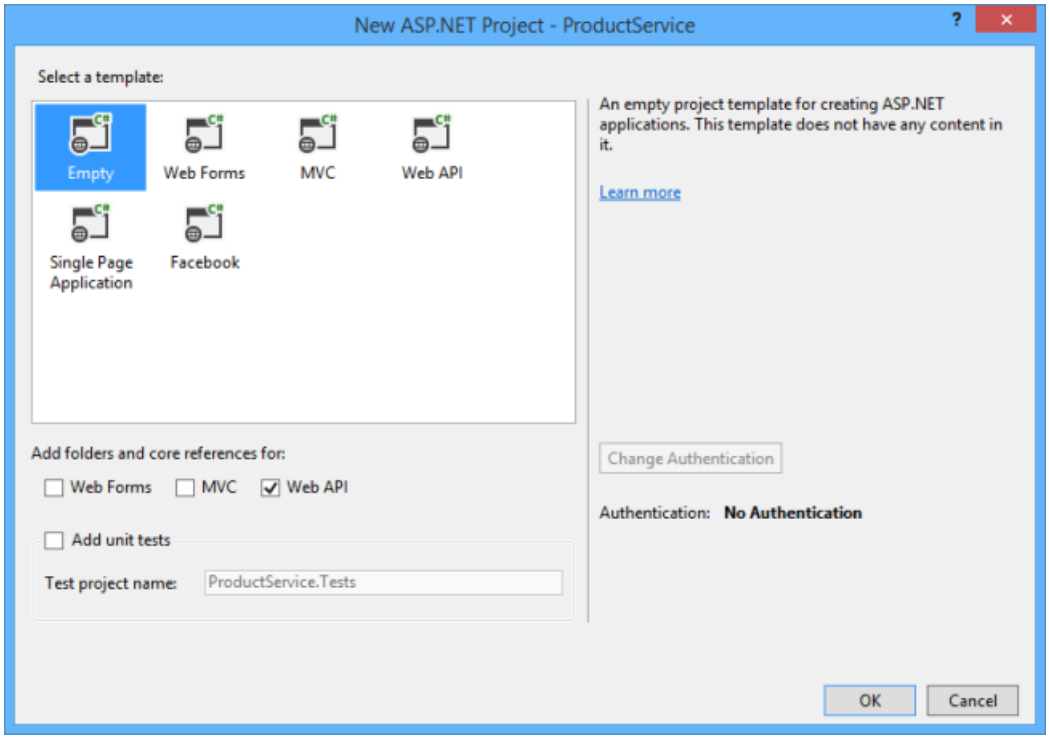
In this tutorial, you will create an OData endpoint that supports basic CRUD operations. The endpoint will expose a single resource, a list of products. Later tutorials will add more features.

Start Visual Studio and select **New Project** from the Start page. Or, from the **File** menu, select **New** and then **Project**.

In the **Templates** pane, select **Installed Templates** and expand the Visual C# node. Under **Visual C#**, select **Web**. Select **the ASP.NET Web Application** template.



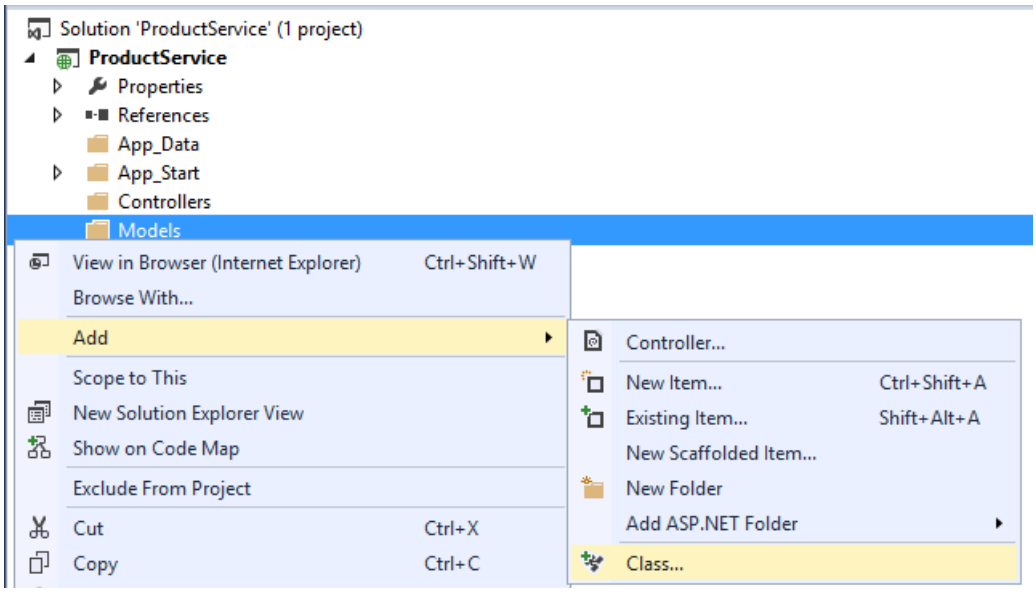
In the **New ASP.NET Project** dialog, select the **Empty** template. Under "Add folders and core references for...", check **Web API**. Click **OK**.



Add an Entity Model

A *model* is an object that represents the data in your application. For this tutorial, we need a model that represents a product. The model corresponds to our OData entity type.

In Solution Explorer, right-click the Models folder. From the context menu, select **Add** then select **Class**.



In the **Add New** Item dialog, name the class "Product".

> **NOTE**
>
> By convention, model classes are placed in the Models folder. You don't have to follow this convention in your own projects, but we'll use it for this tutorial.

In the Product.cs file, add the following class definition:

```
public class Product
{
    public int ID { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Category { get; set; }
}
```

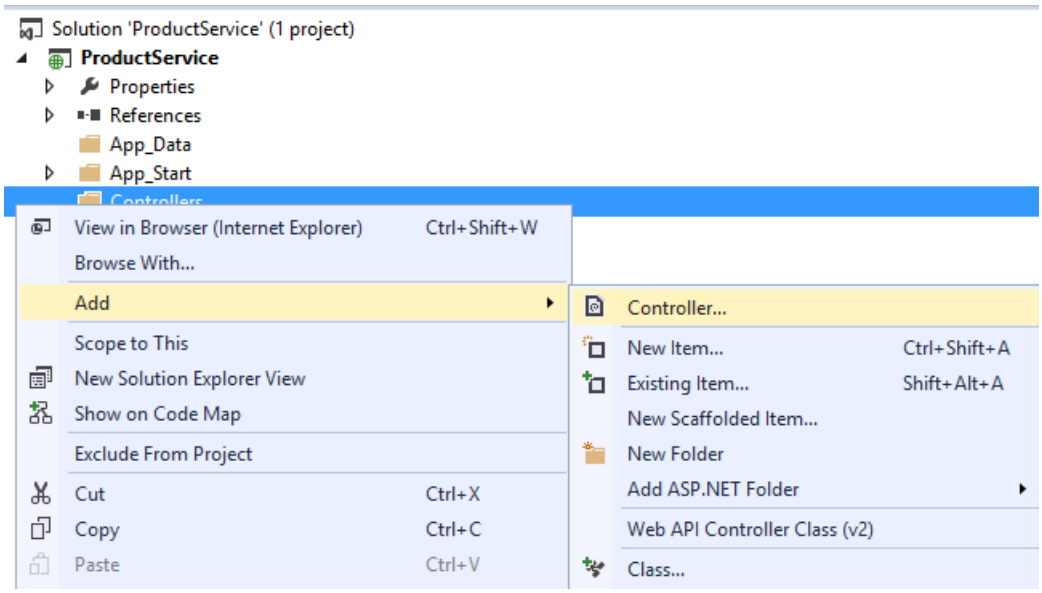
The ID property will be the entity key. Clients can query products by ID. This field would also be the primary key in the back-end database.

Build the project now. In the next step, we'll use some Visual Studio scaffolding that uses reflection to find the Product type.

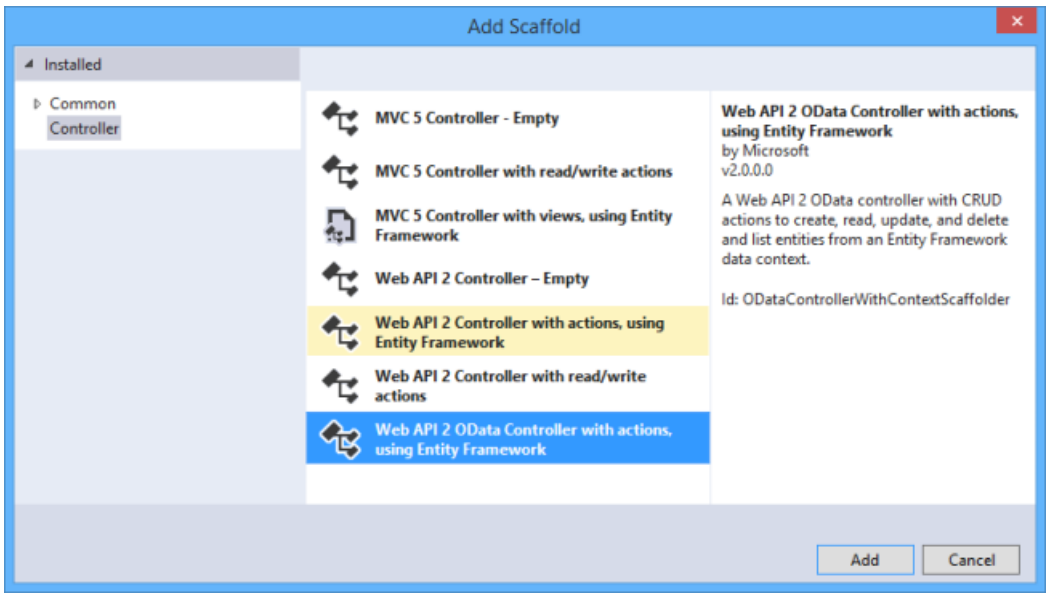
Add an OData Controller

A *controller* is a class that handles HTTP requests. You define a separate controller for each entity set in you OData service. In this tutorial, we'll create a single controller.

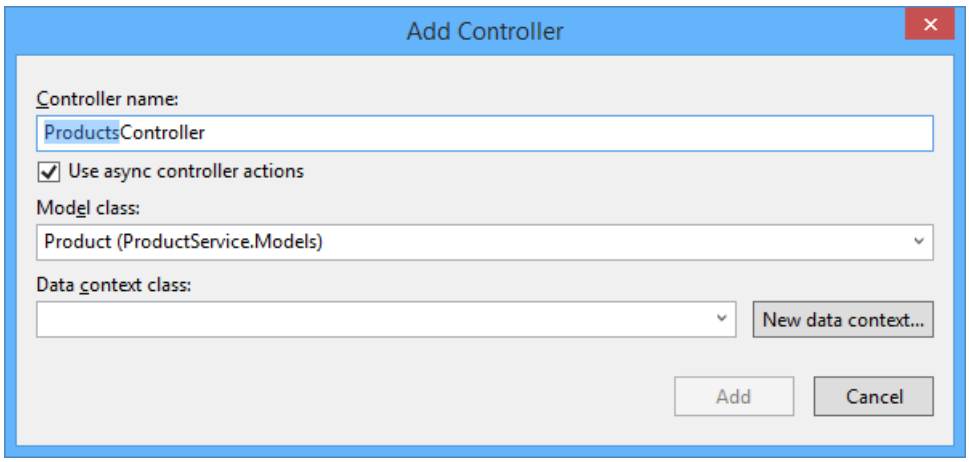
In Solution Explorer, right-click the Controllers folder. Select **Add** and then select **Controller**.



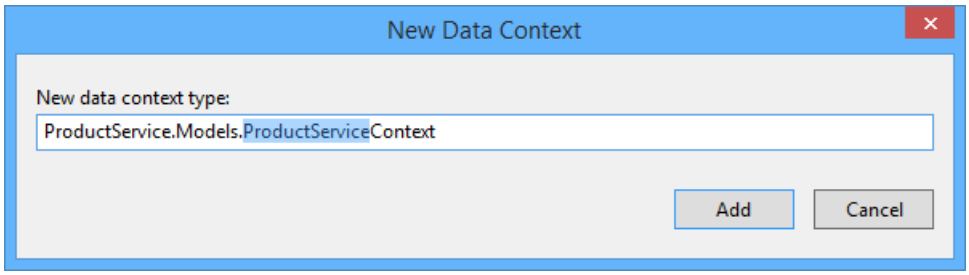
In the **Add Scaffold** dialog, select "Web API 2 OData Controller with actions, using Entity Framework".



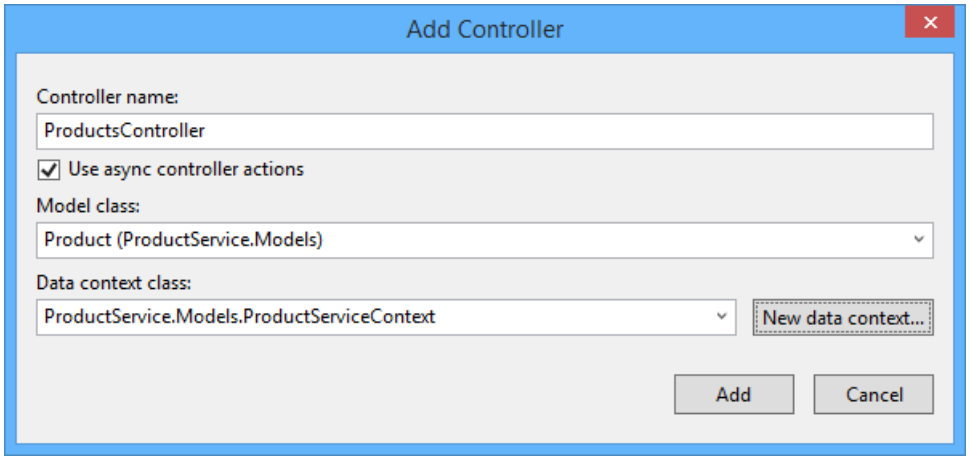
In the **Add Controller** dialog, name the controller "ProductsController". Select the "Use async controller actions" checkbox. In the **Model** drop-down list, select the Product class.



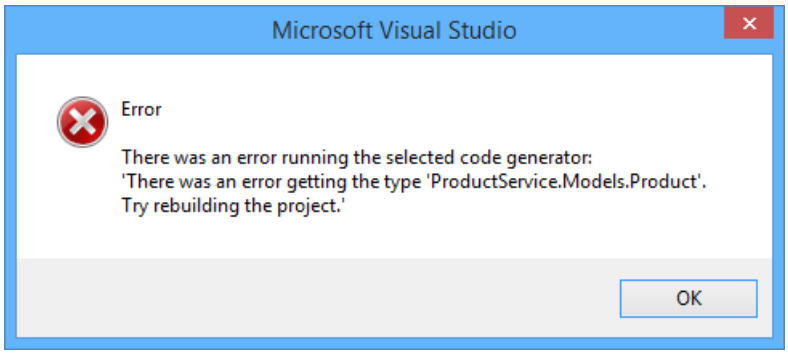
Click the **New data context...** button. Leave the default name for the data context type, and click **Add**.



Click Add in the Add Controller dialog to add the controller.

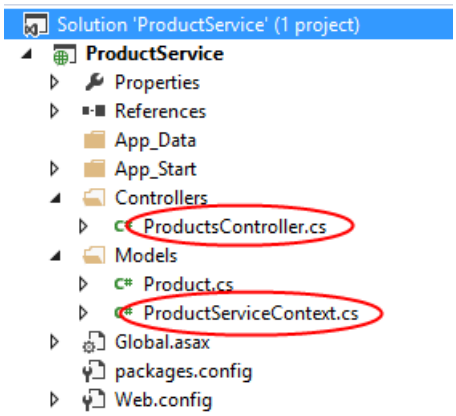


Note: If you get an error message that says "There was an error getting the type...", make sure that you built the Visual Studio project after you added the Product class. The scaffolding uses reflection to find the class.



The scaffolding adds two code files to the project:

- Products.cs defines the Web API controller that implements the OData endpoint.
- ProductServiceContext.cs provides methods to query the underlying database, using Entity Framework.



Add the EDM and Route

In Solution Explorer, expand the App_Start folder and open the file named WebApiConfig.cs. This class holds configuration code for Web API. Replace this code with the following:

```
using ProductService.Models;
using System.Web.Http;
using System.Web.Http.OData.Builder;

namespace ProductService
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            ODataConventionModelBuilder builder = new ODataConventionModelBuilder();
            builder.EntitySet<Product>("Products");
            config.Routes.MapODataRoute("odata", "odata", builder.GetEdmModel());
        }
    }
}
```

This code does two things:

- Creates an Entity Data Model (EDM) for the OData endpoint.
- Adds a route for the endpoint.

An EDM is an abstract model of the data. The EDM is used to create the metadata document and define the URIs for the service. The **ODataConventionModelBuilder** creates an EDM by using a set of default naming conventions EDM. This approach requires the least code. If you want more control over the EDM, you can use the **ODataModelBuilder** class to create the EDM by adding properties, keys, and navigation properties explicitly.

The **EntitySet** method adds an entity set to the EDM:

```
modelBuilder.EntitySet<Product>("Products");
```

The string "Products" defines the name of the entity set. The name of the controller must match the name of the entity set. In this tutorial, the entity set is named "Products" and the controller is named `ProductsController`. If you named the entity set "ProductSet", you would name the controller `ProductSetController`. Note that an endpoint can have multiple entity sets. Call **EntitySet<T>** for each entity set, and then define a corresponding controller.

The **MapODataRoute** method adds a route for the OData endpoint.

```
config.Routes.MapODataRoute("ODataRoute", "odata", model);
```

The first parameter is a friendly name for the route. Clients of your service do not see this name. The second parameter is the URI prefix for the endpoint. Given this code, the URI for the Products entity set is http://*hostname*/odata/Products. Your application can have more than one OData endpoint. For each endpoint, call **MapODataRoute** and provide a unique route name and a unique URI prefix.

Seed the Database (Optional)

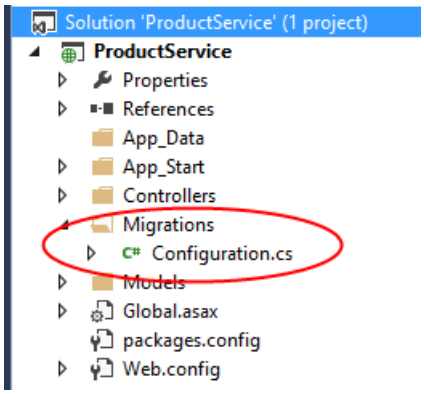
In this step, you will use Entity Framework to seed the database with some test data. This step is optional, but it lets you test out your OData endpoint right away.

From the **Tools** menu, select **NuGet Package Manager**, then select **Package Manager Console**. In the

Package Manager Console window, enter the following command:

```
Enable-Migrations
```

This adds a folder named Migrations and a code file named Configuration.cs.



Open this file and add the following code to the `Configuration.Seed` method.

```
protected override void Seed(ProductService.Models.ProductServiceContext context)
{
    // New code
    context.Products.AddOrUpdate(new Product[] {
        new Product() { ID = 1, Name = "Hat", Price = 15, Category = "Apparel" },
        new Product() { ID = 2, Name = "Socks", Price = 5, Category = "Apparel" },
        new Product() { ID = 3, Name = "Scarf", Price = 12, Category = "Apparel" },
        new Product() { ID = 4, Name = "Yo-yo", Price = 4.95M, Category = "Toys" },
        new Product() { ID = 5, Name = "Puzzle", Price = 8, Category = "Toys" },
    });
}
```

In the Package Manager Console Window, enter the following commands:

```
Add-Migration Initial
Update-Database
```

These commands generate code that creates the database, and then executes that code.

Exploring the OData Endpoint

In this section, we'll use the Fiddler Web Debugging Proxy to send requests to the endpoint and examine the response messages. This will help you to understand the capabilities of an OData endpoint.

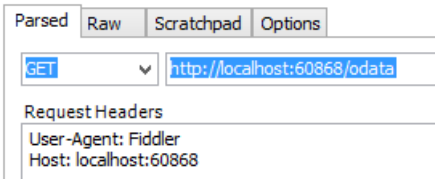
In Visual Studio, press F5 to start debugging. By default, Visual Studio opens your browser to `http://localhost:*port*`, where *port* is the port number configured in the project settings.

You can change the port number in the project settings. In Solution Explorer, right-click the project and select **Properties**. In the properties window, select **Web**. Enter the port number under **Project Url**.

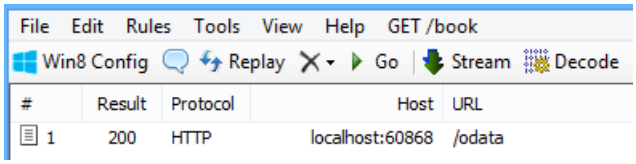
Service Document

The *service document* contains a list of the entity sets for the OData endpoint. To get the service document, send a GET request to the root URI of the service.

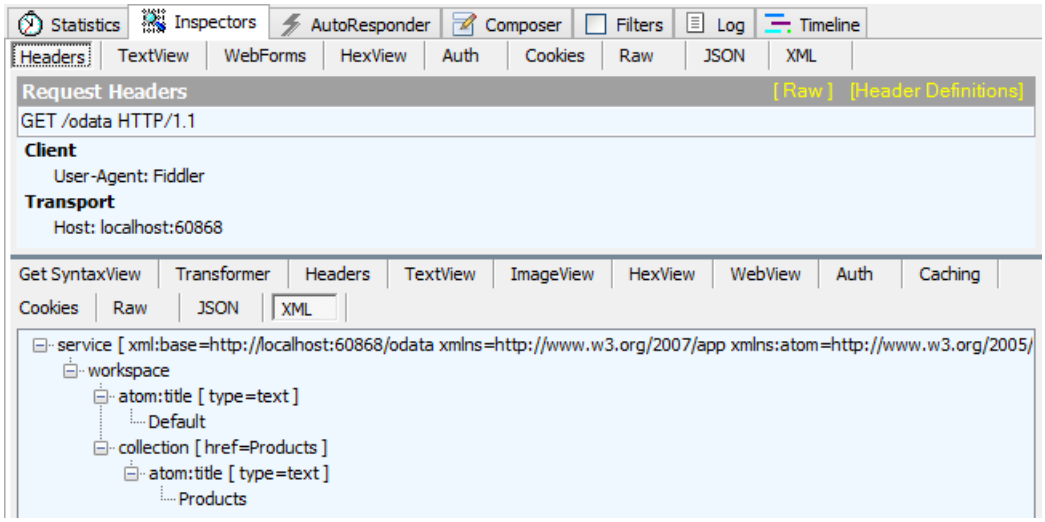
Using Fiddler, enter the following URI in the **Composer** tab: `http://localhost:port/odata/`, where *port* is the port number.



Click the **Execute** button. Fiddler sends an HTTP GET request to your application. You should see the response in the Web Sessions list. If everything is working, the status code will be 200.



Double-click the response in the Web Sessions list to see the details of the response message in the Inspectors tab.



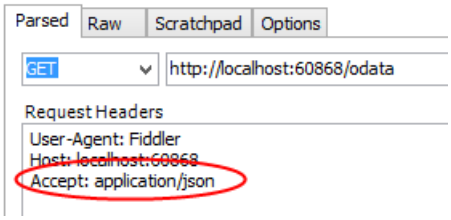
The raw HTTP response message should look similar to the following:

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/atomsvc+xml; charset=utf-8
Expires: -1
Server: Microsoft-IIS/8.0
DataServiceVersion: 3.0
Date: Mon, 23 Sep 2013 17:51:01 GMT
Content-Length: 364

<?xml version="1.0" encoding="utf-8"?>
<service xml:base="http://localhost:60868/odata"
    xmlns="http://www.w3.org/2007/app" xmlns:atom="http://www.w3.org/2005/Atom">
  <workspace>
    <atom:title type="text">Default</atom:title>
    <collection href="Products">
        <atom:title type="text">Products</atom:title>
    </collection>
    </workspace>
</service></pre>
```

By default, Web API returns the service document in AtomPub format. To request JSON, add the following header to the HTTP request:

```
Accept: application/json
```



Now the HTTP response contains a JSON payload:

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/8.0
DataServiceVersion: 3.0
Date: Mon, 23 Sep 2013 22:59:28 GMT
Content-Length: 136

{
  "odata.metadata":"http://localhost:60868/odata/$metadata","value":[
    {
      "name":"Products","url":"Products"
    }
  ]
}
```

Service Metadata Document

The *service metadata document* describes the data model of the service, using an XML language called the Conceptual Schema Definition Language (CSDL). The metadata document shows the structure of the data in the service, and can be used to generate client code.

To get the metadata document, send a GET request to `http://localhost:port/odata/$metadata`. Here is the metadata for the endpoint shown in this tutorial.

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/xml; charset=utf-8
Expires: -1
Server: Microsoft-IIS/8.0
DataServiceVersion: 3.0
Date: Mon, 23 Sep 2013 23:05:52 GMT
Content-Length: 1086

<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="1.0" xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx">
  <edmx:DataServices m:DataServiceVersion="3.0" m:MaxDataServiceVersion="3.0"
    xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
    <Schema Namespace="ProductService.Models" xmlns="http://schemas.microsoft.com/ado/2009/11/edm">
      <EntityType Name="Product">
        <Key>
          <PropertyRef Name="ID" />
        </Key>
        <Property Name="ID" Type="Edm.Int32" Nullable="false" />
        <Property Name="Name" Type="Edm.String" />
        <Property Name="Price" Type="Edm.Decimal" Nullable="false" />
        <Property Name="Category" Type="Edm.String" />
      </EntityType>
    </Schema>
    <Schema Namespace="Default" xmlns="http://schemas.microsoft.com/ado/2009/11/edm">
      <EntityContainer Name="Container" m:IsDefaultEntityContainer="true">
        <EntitySet Name="Products" EntityType="ProductService.Models.Product" />
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

Entity Set

To get the Products entity set, send a GET request to `http://localhost:port/odata/Products`.

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/8.0
DataServiceVersion: 3.0
Date: Mon, 23 Sep 2013 23:01:31 GMT
Content-Length: 459

{
  "odata.metadata":"http://localhost:60868/odata/$metadata#Products","value":[
    {
      "ID":1,"Name":"Hat","Price":"15.00","Category":"Apparel"
    },{
      "ID":2,"Name":"Socks","Price":"5.00","Category":"Apparel"
    },{
      "ID":3,"Name":"Scarf","Price":"12.00","Category":"Apparel"
    },{
      "ID":4,"Name":"Yo-yo","Price":"4.95","Category":"Toys"
    },{
      "ID":5,"Name":"Puzzle","Price":"8.00","Category":"Toys"
    }
  ]
}
```

Entity

To get an individual product, send a GET request to `http://localhost:port/odata/Products(1)`, where "1" is the product ID.

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/8.0
DataServiceVersion: 3.0
Date: Mon, 23 Sep 2013 23:04:29 GMT
Content-Length: 140

{
  "odata.metadata":"http://localhost:60868/odata/$metadata#Products/@Element","ID":1,
      "Name":"Hat","Price":"15.00","Category":"Apparel"
}
```

OData Serialization Formats

OData supports several serialization formats:

- Atom Pub (XML)
- JSON "light" (introduced in OData v3)
- JSON "verbose" (OData v2)

By default, Web API uses AtomPubJSON "light" format.

To get AtomPub format, set the Accept header to "application/atom+xml". Here is an example response body:

```
<?xml version="1.0" encoding="utf-8"?>
<entry xml:base="http://localhost:60868/odata" xmlns="http://www.w3.org/2005/Atom"
xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns:georss="http://www.georss.org/georss" xmlns:gml="http://www.opengis.net/gml">
  <id>http://localhost:60868/odata/Products(1)</id>
  <category term="ProductService.Models.Product"
scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
  <link rel="edit" href="http://localhost:60868/odata/Products(1)" />
  <link rel="self" href="http://localhost:60868/odata/Products(1)" />
  <title />
  <updated>2013-09-23T23:42:11Z</updated>
  <author>
    <name />
  </author>
  <content type="application/xml">
    <m:properties>
      <d:ID m:type="Edm.Int32">1</d:ID>
      <d:Name>Hat</d:Name>
      <d:Price m:type="Edm.Decimal">15.00</d:Price>
      <d:Category>Apparel</d:Category>
    </m:properties>
  </content>
</entry>
```

You can see one obvious disadvantage of the Atom format: It's a lot more verbose than the JSON light. However, if you have a client that understands AtomPub, the client might prefer that format over JSON.

Here is the JSON light version of the same entity:

```
{
  "odata.metadata":"http://localhost:60868/odata/$metadata#Products/@Element",
  "ID":1,
  "Name":"Hat",
  "Price":"15.00",
  "Category":"Apparel"
}
```

The JSON light format was introduced in version 3 of the OData protocol. For backward compatibility, a client can request the older "verbose" JSON format. To request verbose JSON, set the Accept header to `application/json;odata=verbose`. Here is the verbose version:

```
{
  "d":{
    "__metadata":{
      "id":"http://localhost:18285/odata/Products(1)",
      "uri":"http://localhost:18285/odata/Products(1)",
      "type":"ProductService.Models.Product"
    },"ID":1,"Name":"Hat","Price":"15.00","Category":"Apparel"
  }
}
```

This format conveys more metadata in the response body, which can add considerable overhead over an entire session. Also, it adds a level of indirection by wrapping the object in a property named "d".

Next Steps

- Add Entity Relations
- Add OData Actions
- Call the OData Service From a .NET Client

Supporting Entity Relations in OData v3 with Web API 2

2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

Most data sets define relations between entities: Customers have orders; books have authors; products have suppliers. Using OData, clients can navigate over entity relations. Given a product, you can find the supplier. You can also create or remove relationships. For example, you can set the supplier for a product.

This tutorial shows how to support these operations in ASP.NET Web API. The tutorial builds on the tutorial Creating an OData v3 Endpoint with Web API 2.

Software versions used in the tutorial

- Web API 2
- OData Version 3
- Entity Framework 6

Add a Supplier Entity

First we need to add a new entity type to our OData feed. We'll add a `Supplier` class.

```
using System.ComponentModel.DataAnnotations;

namespace ProductService.Models
{
    public class Supplier
    {
        [Key]
        public string Key { get; set; }
        public string Name { get; set; }
    }
}
```

This class uses a string for the entity key. In practice, that might be less common than using an integer key. But it's worth seeing how OData handles other key types besides integers.

Next, we'll create a relation by adding a `Supplier` property to the `Product` class:

```
public class Product
{
    public int ID { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Category { get; set; }

    // New code
    [ForeignKey("Supplier")]
    public string SupplierId { get; set; }
    public virtual Supplier Supplier { get; set; }
}
```

Add a new **DbSet** to the `ProductServiceContext` class, so that Entity Framework will include the `Supplier` table in the database.

```
public class ProductServiceContext : DbContext
{
    public ProductServiceContext() : base("name=ProductServiceContext")
    {
    }

    public System.Data.Entity.DbSet<ProductService.Models.Product> Products { get; set; }
    // New code:
    public System.Data.Entity.DbSet<ProductService.Models.Supplier> Suppliers { get; set; }
}
```

In WebApiConfig.cs, add a "Suppliers" entity to the EDM model:

```
ODataConventionModelBuilder builder = new ODataConventionModelBuilder();
builder.EntitySet<Product>("Products");
// New code:
builder.EntitySet<Supplier>("Suppliers");
```

Navigation Properties

To get the supplier for a product, the client sends a GET request:

```
GET /Products(1)/Supplier
```

Here "Supplier" is a navigation property on the `Product` type. In this case, `Supplier` refers to a single item, but a navigation property can also return a collection (one-to-many or many-to-many relation).

To support this request, add the following method to the `ProductsController` class:

```
// GET /Products(1)/Supplier
public Supplier GetSupplier([FromODataUri] int key)
{
    Product product = _context.Products.FirstOrDefault(p => p.ID == key);
    if (product == null)
    {
        throw new HttpResponseException(HttpStatusCode.NotFound);
    }
    return product.Supplier;
}
```

The *key* parameter is the key of the product. The method returns the related entity—in this case, a `Supplier`

instance. The method name and parameter name are both important. In general, if the navigation property is named "X", you need to add a method named "GetX". The method must take a parameter named "*key*" that matches the data type of the parent's key.

It is also important to include the **[FromODataUri]** attribute in the *key* parameter. This attribute tells Web API to use OData syntax rules when it parses the key from the request URI.

Creating and Deleting Links

OData supports creating or removing relationships between two entities. In OData terminology, the relationship is a "link." Each link has a URI with the form *entity*/$links/*entity*. For example, the link from product to supplier looks like this:

```
/Products(1)/$links/Supplier
```

To create a new link, the client sends a POST request to the link URI. The body of the request is the URI of the target entity. For example, suppose there is a supplier with the key "CTSO". To create a link from "Product(1)" to "Supplier('CTSO')", the client sends a request like the following:

```
POST http://localhost/odata/Products(1)/$links/Supplier
Content-Type: application/json
Content-Length: 50

{"url":"http://localhost/odata/Suppliers('CTSO')"}
```

To delete a link, the client sends a DELETE request to the link URI.

Creating Links

To enable a client to create product-supplier links, add the following code to the `ProductsController` class:

```
[AcceptVerbs("POST", "PUT")]
public async Task<IHttpActionResult> CreateLink([FromODataUri] int key, string navigationProperty, [FromBody]
Uri link)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    Product product = await db.Products.FindAsync(key);
    if (product == null)
    {
        return NotFound();
    }

    switch (navigationProperty)
    {
        case "Supplier":
            string supplierKey = GetKeyFromLinkUri<string>(link);
            Supplier supplier = await db.Suppliers.FindAsync(supplierKey);
            if (supplier == null)
            {
                return NotFound();
            }
            product.Supplier = supplier;
            await db.SaveChangesAsync();
            return StatusCode(HttpStatusCode.NoContent);

        default:
            return NotFound();
    }
}
```

This method takes three parameters:

- *key*: The key to the parent entity (the product)
- *navigationProperty*: The name of the navigation property. In this example, the only valid navigation property is "Supplier".
- *link*: The OData URI of the related entity. This value is taken from the request body. For example, the link URI might be "`http://localhost/odata/Suppliers('CTSO')`, meaning the supplier with ID = 'CTSO'.

The method uses the link to look up the supplier. If the matching supplier is found, the method sets the `Product.Supplier` property and saves the result to the database.

The hardest part is parsing the link URI. Basically, you need to simulate the result of sending a GET request to that URI. The following helper method shows how to do this. The method invokes the Web API routing process and gets back an **ODataPath** instance that represents the parsed OData path. For a link URI, one of the segments should be the entity key. (If not, the client sent a bad URI.)

```
// Helper method to extract the key from an OData link URI.
private TKey GetKeyFromLinkUri<TKey>(Uri link)
{
    TKey key = default(TKey);

    // Get the route that was used for this request.
    IHttpRoute route = Request.GetRouteData().Route;

    // Create an equivalent self-hosted route.
    IHttpRoute newRoute = new HttpRoute(route.RouteTemplate,
        new HttpRouteValueDictionary(route.Defaults),
        new HttpRouteValueDictionary(route.Constraints),
        new HttpRouteValueDictionary(route.DataTokens), route.Handler);

    // Create a fake GET request for the link URI.
    var tmpRequest = new HttpRequestMessage(HttpMethod.Get, link);

    // Send this request through the routing process.
    var routeData = newRoute.GetRouteData(
        Request.GetConfiguration().VirtualPathRoot, tmpRequest);

    // If the GET request matches the route, use the path segments to find the key.
    if (routeData != null)
    {
        ODataPath path = tmpRequest.GetODataPath();
        var segment = path.Segments.OfType<KeyValuePathSegment>().FirstOrDefault();
        if (segment != null)
        {
            // Convert the segment into the key type.
            key = (TKey)ODataUriUtils.ConvertFromUriLiteral(
                segment.Value, ODataVersion.V3);
        }
    }
    return key;
}
```

Deleting Links

To delete a link, add the following code to the `ProductsController` class:

```
public async Task<IHttpActionResult> DeleteLink([FromODataUri] int key, string navigationProperty)
{
    Product product = await db.Products.FindAsync(key);
    if (product == null)
    {
        return NotFound();
    }

    switch (navigationProperty)
    {
        case "Supplier":
            product.Supplier = null;
            await db.SaveChangesAsync();
            return StatusCode(HttpStatusCode.NoContent);

        default:
            return NotFound();

    }
}
```

In this example, the navigation property is a single `Supplier` entity. If the navigation property is a collection, the URI to delete a link must include a key for the related entity. For example:

```
DELETE /odata/Customers(1)/$links/Orders(1)
```

This request removes order 1 from customer 1. In this case, the DeleteLink method will have the following signature:

```
void DeleteLink([FromODataUri] int key, string relatedKey, string navigationProperty);
```

The *relatedKey* parameter gives the key for the related entity. So in your `DeleteLink` method, look up the primary entity by the *key* parameter, find the related entity by the *relatedKey* parameter, and then remove the association. Depending on your data model, you might need to implement both versions of `DeleteLink`. Web API will call the correct version based on the request URI.

Supporting OData Actions in ASP.NET Web API 2

2/20/2020 • 6 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

In OData, *actions* are a way to add server-side behaviors that are not easily defined as CRUD operations on entities. Some uses for actions include:

- Implementing complex transactions.
- Manipulating several entities at once.
- Allowing updates only to certain properties of an entity.
- Sending information to the server that is not defined in an entity.

Software versions used in the tutorial

- Web API 2
- OData Version 3
- Entity Framework 6

Example: Rating a Product

In this example, we want to let users rate products, and then expose the average ratings for each product. On the database, we will store a list of ratings, keyed to products.

Here is the model we might use to represent the ratings in Entity Framework:

```
public class ProductRating
{
    public int ID { get; set; }

    [ForeignKey("Product")]
    public int ProductID { get; set; }
    public virtual Product Product { get; set; }  // Navigation property

    public int Rating { get; set; }
}
```

But we don't want clients to POST a `ProductRating` object to a "Ratings" collection. Intuitively, the rating is associated with the Products collection, and the client should only need to post the rating value.

Therefore, instead of using the normal CRUD operations, we define an action that a client can invoke on a Product. In OData terminology, the action is *bound* to Product entities.

Actions have side-effects on the server. For this reason, they are invoked using HTTP POST requests. Actions can have parameters and return types, which are described in the service metadata. The client sends the parameters in the request body, and the server sends the return value in the response body. To invoke the "Rate Product" action, the client sends a POST to a URI like the following:

```
http://localhost/odata/Products(1)/RateProduct
```

The data in the POST request is simply the product rating:

```
{"Rating":2}
```

Declare the Action in the Entity Data Model

In your Web API configuration, add the action to the entity data model (EDM):

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        ODataConventionModelBuilder builder = new ODataConventionModelBuilder();
        builder.EntitySet<Product>("Products");
        builder.EntitySet<Supplier>("Suppliers");
        builder.EntitySet<ProductRating>("Ratings");

        // New code: Add an action to the EDM, and define the parameter and return type.
        ActionConfiguration rateProduct = builder.Entity<Product>().Action("RateProduct");
        rateProduct.Parameter<int>("Rating");
        rateProduct.Returns<double>();

        config.Routes.MapODataRoute("odata", "odata", builder.GetEdmModel());
    }
}
```

This code defines "RateProduct" as an action that can be performed on Product entities. It also declares that the action takes an **int** parameter named "Rating", and returns an **int** value.

Add the Action to the Controller

The "RateProduct" action is bound to Product entities. To implement the action, add a method named `RateProduct` to the Products controller:

```
[HttpPost]
public async Task<IHttpActionResult> RateProduct([FromODataUri] int key, ODataActionParameters parameters)
{
    if (!ModelState.IsValid)
    {
        return BadRequest();
    }

    int rating = (int)parameters["Rating"];

    Product product = await db.Products.FindAsync(key);
    if (product == null)
    {
        return NotFound();
    }

    product.Ratings.Add(new ProductRating() { Rating = rating });
    db.SaveChanges();

    double average = product.Ratings.Average(x => x.Rating);

    return Ok(average);
}
```

Notice that the method name matches the name of the action in the EDM. The method has two parameters:

- *key*: The key for the product to rate.
- *parameters*: A dictionary of action parameter values.

If you are using the default routing conventions, the key parameter must be named "key". It is also important to include the **[FromOdataUri]** attribute, as shown. This attribute tells Web API to use OData syntax rules when it parses the key from the request URI.

Use the *parameters* dictionary to get the action parameters:

```
if (!ModelState.IsValid)
{
    return BadRequest();
}
int rating = (int)parameters["Rating"];
```

If the client sends the action parameters in the correct format, the value of **ModelState.IsValid** is true. In that case, you can use the **ODataActionParameters** dictionary to get the parameter values. In this example, the `RateProduct` action takes a single parameter named "Rating".

Action Metadata

To view the service metadata, send a GET request to /odata/$metadata. Here is the portion of the metadata that declares the `RateProduct` action:

```
<FunctionImport Name="RateProduct" m:IsAlwaysBindable="true" IsBindable="true" ReturnType="Edm.Double">
  <Parameter Name="bindingParameter" Type="ProductService.Models.Product"/>
  <Parameter Name="Rating" Nullable="false" Type="Edm.Int32"/>
</FunctionImport>
```

The **FunctionImport** element declares the action. Most of the fields are self-explanatory, but two are worth noting:

- **IsBindable** means the action can be invoked on the target entity, at least some of the time.

- **IsAlwaysBindable** means the action can always be invoked on the target entity.

The difference is that some actions are always available to clients, but other actions might depend on the state of the entity. For example, suppose you define a "Purchase" action. You can only purchase an item that is in stock. If the item is out of stock, a client cannot invoke that action.

When you define the EDM, the **Action** method creates an always-bindable action:

```
builder.Entity<Product>().Action("RateProduct"); // Always bindable
```

I'll talk about not-always-bindable actions (also called *transient* actions) later in this topic.

Invoking the Action

Now let's see how a client would invoke this action. Suppose the client wants to give a rating of 2 to the product with ID = 4. Here is an example request message, using JSON format for the request body:

```
POST http://localhost/odata/Products(4)/RateProduct HTTP/1.1
Content-Type: application/json
Content-Length: 12

{"Rating":2}
```

Here is the response message:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
DataServiceVersion: 3.0
Date: Tue, 22 Oct 2013 19:04:00 GMT
Content-Length: 89

{
  "odata.metadata":"http://localhost:21900/odata/$metadata#Edm.Double","value":2.75
}
```

Binding an Action to an Entity Set

In the previous example, the action is bound to a single entity: The client rates a single product. You can also bind an action to a collection of entities. Just make the following changes:

In the EDM, add the action to the entity's **Collection** property.

```
var rateAllProducts = builder.Entity<Product>().Collection.Action("RateAllProducts");
```

In the controller method, omit the *key* parameter.

```
[HttpPost]
public int RateAllProducts(ODataActionParameters parameters)
{
    // ....
}
```

Now the client invokes the action on the Products entity set:

```
http://localhost/odata/Products/RateAllProducts
```

Actions with Collection Parameters

Actions can have parameters that take a collection of values. In the EDM, use **CollectionParameter<T>** to declare the parameter.

```
rateAllProducts.CollectionParameter<int>("Ratings");
```

This declares a parameter named "Ratings" that takes a collection of **int** values. In the controller method, you still get the parameter value from the **ODataActionParameters** object, but now the value is an **ICollection<int>** value:

```
[HttpPost]
public void RateAllProducts(ODataActionParameters parameters)
{
    if (!ModelState.IsValid)
    {
        throw new HttpResponseException(HttpStatusCode.BadRequest);
    }

    var ratings = parameters["Ratings"] as ICollection<int>;

    // ...
}
```

Transient Actions

In the "RateProduct" example, users can always rate a product, so the action is always available. But some actions depend on the state of the entity. For example, in a video rental service, the "CheckOut" action is not always available. (It depends whether a copy of that video is available.) This type of action is called a *transient* action.

In the service metadata, a transient action has **IsAlwaysBindable** equal to false. That's actually the default value, so the metadata will look like this:

```
<FunctionImport Name="CheckOut" IsBindable="true">
    <Parameter Name="bindingParameter" Type="ProductsService.Models.Product" />
</FunctionImport>
```

Here's why this matters: If an action is transient, the server needs to tell the client when the action is available. It does this by including a link to the action in the entity. Here is an example for a Movie entity:

```
{
  "odata.metadata":"http://localhost:17916/odata/$metadata#Movies/@Element",
  "#CheckOut":{ "target":"http://localhost:17916/odata/Movies(1)/CheckOut" },
  "ID":1,"Title":"Sudden Danger 3","Year":2012,"Genre":"Action"
}
```

The "#CheckOut" property contains a link to the CheckOut action. If the action is not available, the server omits the link.

To declare a transient action in the EDM, call the **TransientAction** method:

```
var checkoutAction = builder.Entity<Movie>().TransientAction("CheckOut");
```

Also, you must provide a function that returns an action link for a given entity. Set this function by calling **HasActionLink**. You can write the function as a lambda expression:

```
checkoutAction.HasActionLink(ctx =>
{
    var movie = ctx.EntityInstance as Movie;
    if (movie.IsAvailable) {
        return new Uri(ctx.Url.ODataLink(
            new EntitySetPathSegment(ctx.EntitySet),
            new KeyValuePathSegment(movie.ID.ToString()),
            new ActionPathSegment(checkoutAction.Name)));
    }
    else
    {
        return null;
    }
}, followsConventions: true);
```

If the action is available, the lambda expression returns a link to the action. The OData serializer includes this link when it serializes the entity. When the action is not available, the function returns `null`.

Additional Resources

OData Actions Sample

Calling an OData Service From a .NET Client (C#)

2/20/2020 • 8 minutes to read • Edit Online

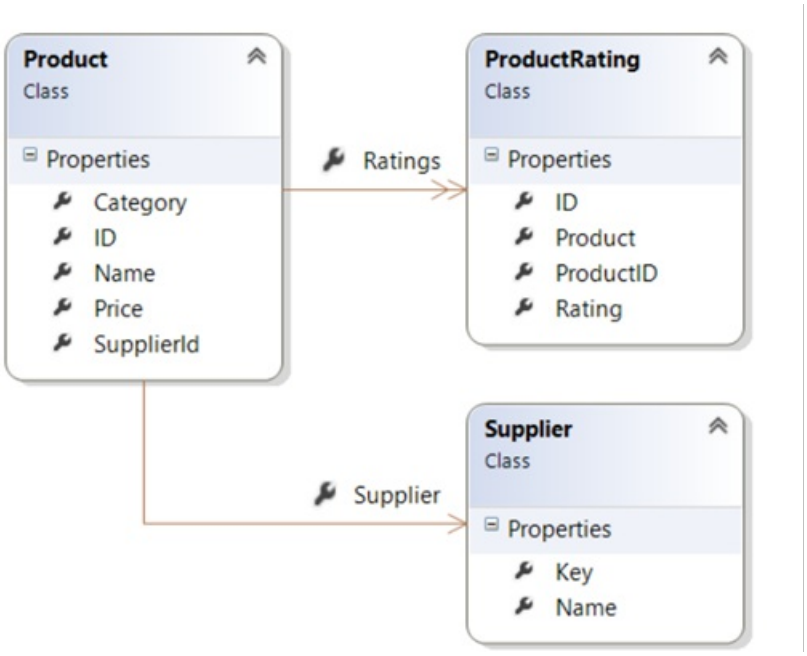
by Mike Wasson

Download Completed Project

> This tutorial shows how to call an OData service from a C# client application.
>
> ## Software versions used in the tutorial
>
> - Visual Studio 2013 (works with Visual Studio 2012)
> - WCF Data Services Client Library
> - Web API 2. (The example OData service is built using Web API 2, but the client application does not depend on Web API.)

In this tutorial, I'll walk through creating a client application that calls an OData service. The OData service exposes the following entities:

- `Product`
- `Supplier`
- `ProductRating`



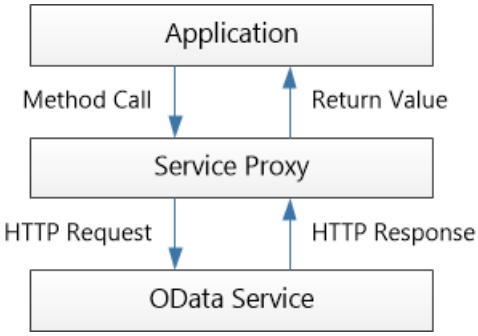
The following articles describe how to implement the OData service in Web API. (You don't need to read them to understand this tutorial, however.)

- Creating an OData Endpoint in Web API 2
- OData Entity Relations in Web API 2
- OData Actions in Web API 2

Generate the Service Proxy

The first step is to generate a service proxy. The service proxy is a .NET class that defines methods for accessing

the OData service. The proxy translates method calls into HTTP requests.

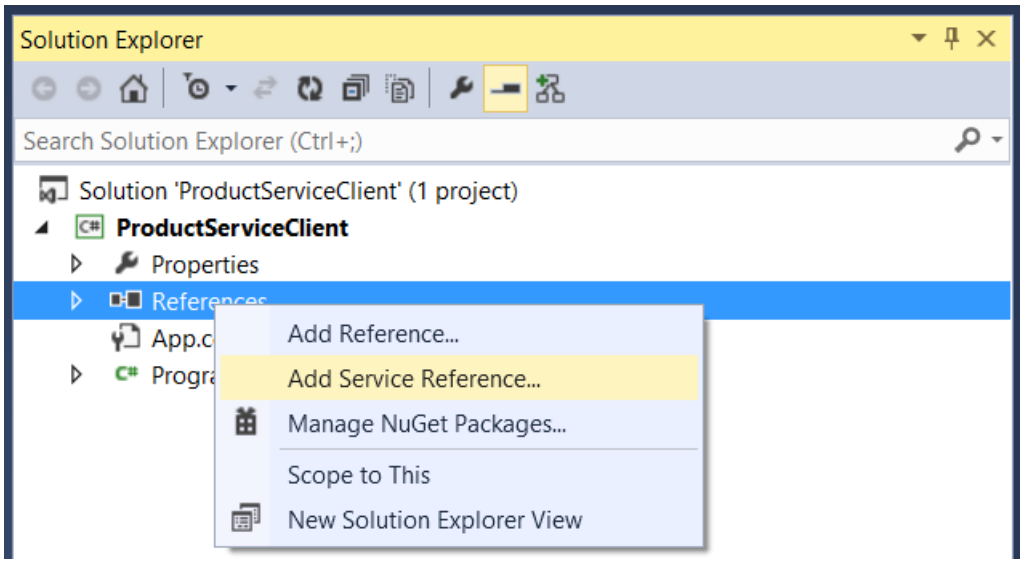


Start by opening the OData service project in Visual Studio. Press CTRL+F5 to run the service locally in IIS Express. Note the local address, including the port number that Visual Studio assigns. You will need this address when you create the proxy.

Next, open another instance of Visual Studio and create a console application project. The console application will be our OData client application. (You can also add the project to the same solution as the service.)

> **NOTE**
>
> The remaining steps refer the console project.

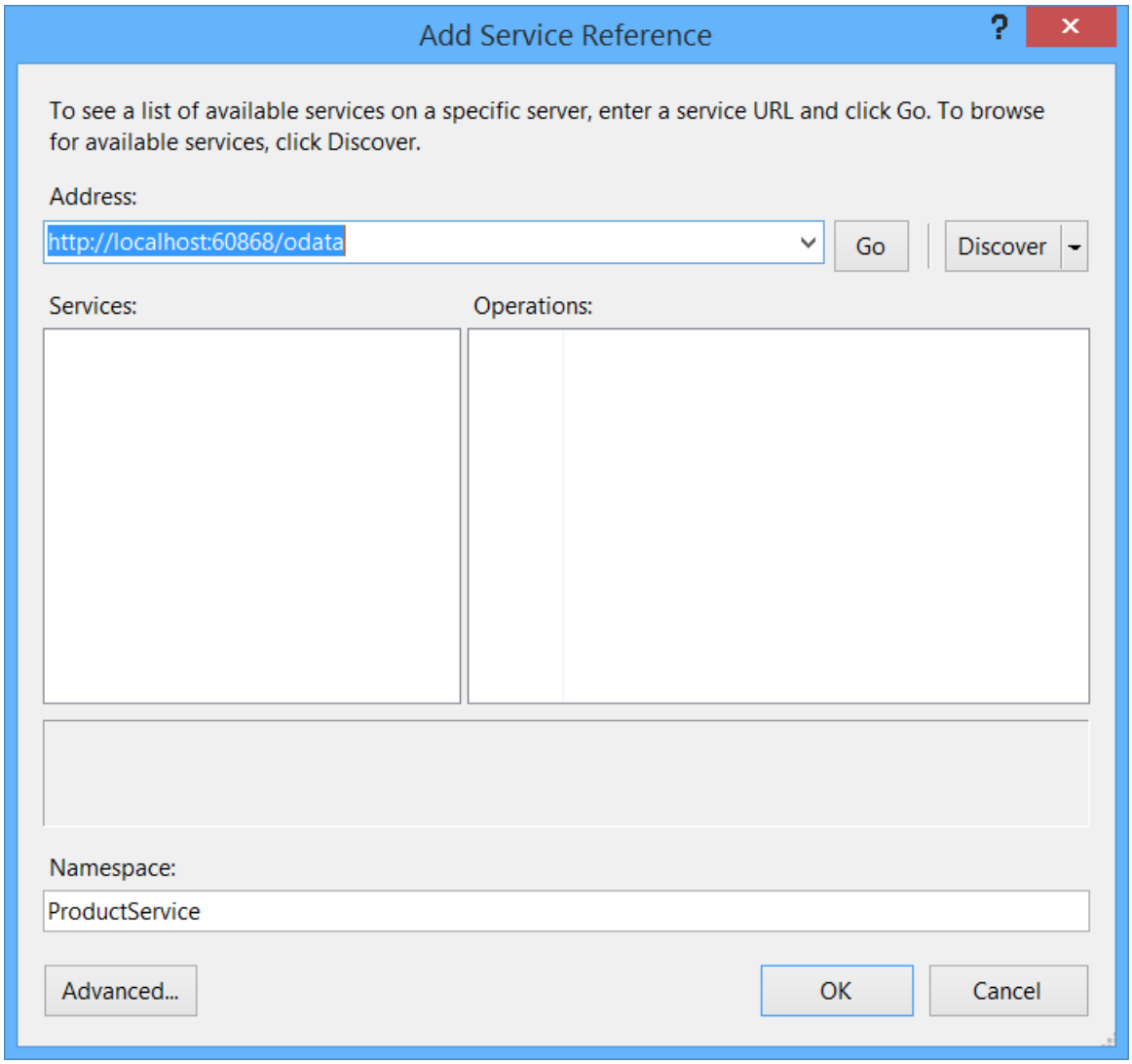
In Solution Explorer, right-click **References** and select **Add Service Reference**.



In the **Add Service Reference** dialog, type the address of the OData service:

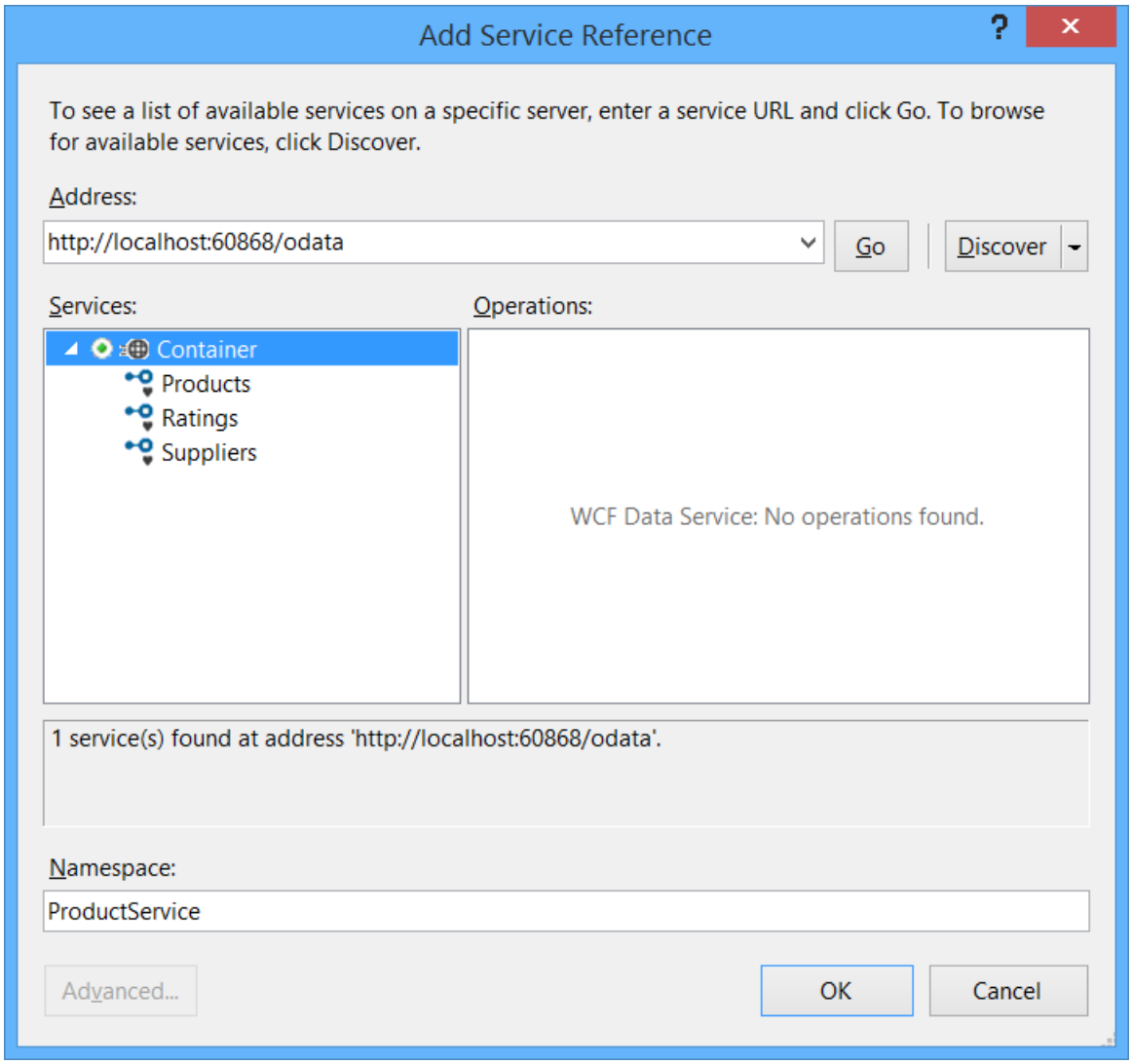
```
http://localhost:port/odata
```

where *port* is the port number.

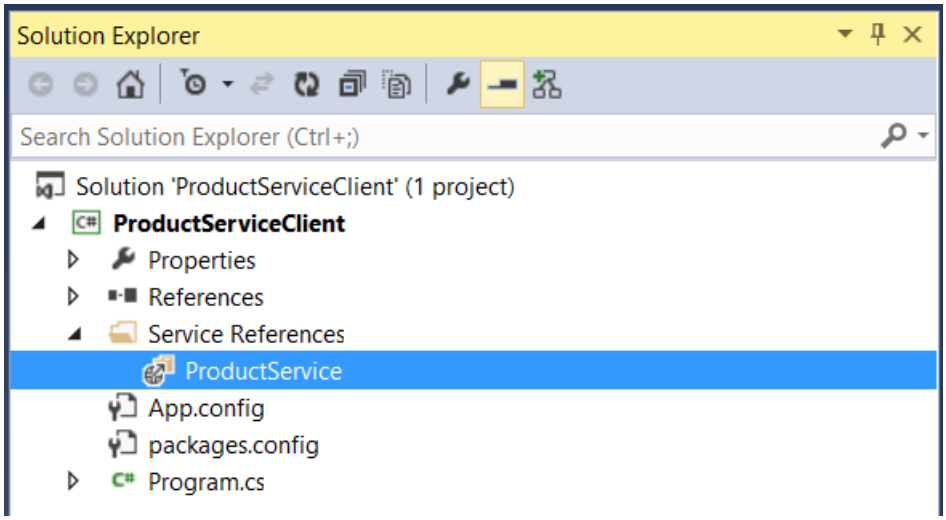


For **Namespace**, type "ProductService". This option defines the namespace of the proxy class.

Click **Go**. Visual Studio reads the OData metadata document to discover the entities in the service.



Click **OK** to add the proxy class to your project.



Create an Instance of the Service Proxy Class

Inside your `Main` method, create a new instance of the proxy class, as follows:

```
using System;
using System.Data.Services.Client;
using System.Linq;

namespace Client
{
    class Program
    {
        static void Main(string[] args)
        {
            Uri uri = new Uri("http://localhost:1234/odata/");
            var container = new ProductService.Container(uri);

            // ...
        }
    }
}
```

Again, use the actual port number where your service is running. When you deploy your service, you will use the URI of the live service. You don't need to update the proxy.

The following code adds an event handler that prints the request URIs to the console window. This step isn't required, but it's interesting to see the URIs for each query.

```
container.SendingRequest2 += (s, e) =>
{
    Console.WriteLine("{0} {1}", e.RequestMessage.Method, e.RequestMessage.Url);
};
```

Query the Service

The following code gets the list of products from the OData service.

```
class Program
{
    static void DisplayProduct(ProductService.Product product)
    {
        Console.WriteLine("{0} {1} {2}", product.Name, product.Price, product.Category);
    }

    // Get an entire entity set.
    static void ListAllProducts(ProductService.Container container)
    {
        foreach (var p in container.Products)
        {
            DisplayProduct(p);
        }
    }

    static void Main(string[] args)
    {
        Uri uri = new Uri("http://localhost:18285/odata/");
        var container = new ProductService.Container(uri);
        container.SendingRequest2 += (s, e) =>
        {
            Console.WriteLine("{0} {1}", e.RequestMessage.Method, e.RequestMessage.Url);
        };

        // Get the list of products
        ListAllProducts(container);
    }
}
```

Notice that you don't need to write any code to send the HTTP request or parse the response. The proxy class does this automatically when you enumerate the `Container.Products` collection in the **foreach** loop.

When you run the application, the output should look like the following:

```
GET http://localhost:60868/odata/Products
Hat 15.00   Apparel
Scarf   12.00   Apparel
Socks   5.00    Apparel
Yo-yo   4.95    Toys
Puzzle  8.00    Toys
```

To get an entity by ID, use a `where` clause.

```
// Get a single entity.
static void ListProductById(ProductService.Container container, int id)
{
    var product = container.Products.Where(p => p.ID == id).SingleOrDefault();
    if (product != null)
    {
        DisplayProduct(product);
    }
}
```

For the rest of this topic, I won't show the entire `Main` function, just the code needed to call the service.

Apply Query Options

OData defines query options that can be used to filter, sort, page data, and so forth. In the service proxy, you can apply these options by using various LINQ expressions.

In this section, I'll show brief examples. For more details, see the topic LINQ Considerations (WCF Data Services) on MSDN.

Filtering ($filter)

To filter, use a `where` clause. The following example filters by product category.

```
// Use the $filter option.
static void ListProductsInCategory(ProductService.Container container, string category)
{
    var products =
        from p in container.Products
        where p.Category == category
        select p;
    foreach (var p in products)
    {
        DisplayProduct(p);
    }
}
```

This code corresponds to the following OData query.

```
GET http://localhost/odata/Products()?$filter=Category eq 'apparel'
```

Notice that the proxy converts the `where` clause into an OData `$filter` expression.

Sorting ($orderby)

To sort, use an `orderby` clause. The following example sorts by price, from highest to lowest.

```
// Use the $orderby option
static void ListProductsSorted(ProductService.Container container)
{
    // Sort by price, highest to lowest.
    var products =
        from p in container.Products
        orderby p.Price descending
        select p;

    foreach (var p in products)
    {
        DisplayProduct(p);
    }
}
```

Here is the corresponding OData request.

```
GET http://localhost/odata/Products()?$orderby=Price desc
```

Client-Side Paging ($skip and $top)

For large entity sets, the client might want to limit the number of results. For example, a client might show 10 entries at a time. This is called *client-side paging*. (There is also server-side paging, where the server limits the number of results.) To perform client-side paging, use the LINQ **Skip** and **Take** methods. The following example skips the first 40 results and takes the next 10.

```
// Use $skip and $top options.
static void ListProductsPaged(ProductService.Container container)
{
    var products =
        (from p in container.Products
          orderby p.Price descending
          select p).Skip(40).Take(10);

    foreach (var p in products)
    {
        DisplayProduct(p);
    }
}
```

Here is the corresponding OData request:

```
GET http://localhost/odata/Products()?$orderby=Price desc&$skip=40&$top=10
```

Select ($select) and Expand ($expand)

To include related entities, use the `DataServiceQuery<t>.Expand` method. For example, to include the `Supplier` for each `Product`:

```
// Use the $expand option.
static void ListProductsAndSupplier(ProductService.Container container)
{
    var products = container.Products.Expand(p => p.Supplier);
    foreach (var p in products)
    {
        Console.WriteLine("{0}\t{1}\t{2}", p.Name, p.Price, p.Supplier.Name);
    }
}
```

Here is the corresponding OData request:

```
GET http://localhost/odata/Products()?$expand=Supplier
```

To change the shape of the response, use the LINQ **select** clause. The following example gets just the name of each product, with no other properties.

```
// Use the $select option.
static void ListProductNames(ProductService.Container container)
{

    var products = from p in container.Products select new { Name = p.Name };
    foreach (var p in products)
    {
        Console.WriteLine(p.Name);
    }
}
```

Here is the corresponding OData request:

```
GET http://localhost/odata/Products()?$select=Name
```

A select clause can include related entities. In that case, do not call **Expand**; the proxy automatically includes the expansion in this case. The following example gets the name and supplier of each product.

```
// Use $expand and $select options
static void ListProductNameSupplier(ProductService.Container container)
{
    var products =
        from p in container.Products
        select new
        {
            Name = p.Name,
            Supplier = p.Supplier.Name
        };
    foreach (var p in products)
    {
        Console.WriteLine("{0}\t{1}", p.Name, p.Supplier);
    }
}
```

Here is the corresponding OData request. Notice that it includes the **$expand** option.

```
GET http://localhost/odata/Products()?$expand=Supplier&$select=Name,Supplier/Name
```

For more information about $select and $expand, see Using $select, $expand, and $value in Web API 2.

Add a New Entity

To add a new entity to an entity set, call `AddToEntitySet`, where *EntitySet* is the name of the entity set. For example, `AddToProducts` adds a new `Product` to the `Products` entity set. When you generate the proxy, WCF Data Services automatically creates these strongly-typed **AddTo** methods.

```
// Add an entity.
static void AddProduct(ProductService.Container container, ProductService.Product product)
{
    container.AddToProducts(product);
    var serviceResponse = container.SaveChanges();
    foreach (var operationResponse in serviceResponse)
    {
        Console.WriteLine(operationResponse.StatusCode);
    }
}
```

To add a link between two entities, use the **AddLink** and **SetLink** methods. The following code adds a new supplier and a new product, and then creates links between them.

```
// Add entities with links.
static void AddProductWithSupplier(ProductService.Container container,
    ProductService.Product product, ProductService.Supplier supplier)
{
    container.AddToSuppliers(supplier);
    container.AddToProducts(product);
    container.AddLink(supplier, "Products", product);
    container.SetLink(product, "Supplier", supplier);
    var serviceResponse = container.SaveChanges();
    foreach (var operationResponse in serviceResponse)
    {
        Console.WriteLine(operationResponse.StatusCode);
    }
}
```

Use **AddLink** when the navigation property is a collection. In this example, we are adding a product to the `Products` collection on the supplier.

Use **SetLink** when the navigation property is a single entity. In this example, we are setting the `Supplier` property on the product.

Update / Patch

To update an entity, call the **UpdateObject** method.

```
static void UpdatePrice(ProductService.Container container, int id, decimal price)
{
    var product = container.Products.Where(p => p.ID == id).SingleOrDefault();
    if (product != null)
    {
        product.Price = price;
        container.UpdateObject(product);
        container.SaveChanges(SaveChangesOptions.PatchOnUpdate);
    }
}
```

The update is performed when you call **SaveChanges**. By default, WCF sends an HTTP MERGE request. The **PatchOnUpdate** option tells WCF to send an HTTP PATCH instead.

> **NOTE**
> Why PATCH versus MERGE? The original HTTP 1.1 specification (RCF 2616) did not define any HTTP method with "partial update" semantics. To support partial updates, the OData specification defined the MERGE method. In 2010, RFC 5789 defined the PATCH method for partial updates. You can read some of the history in this blog post on the WCF Data Services Blog. Today, PATCH is preferred over MERGE. The OData controller created by the Web API scaffolding supports both methods.

If you want to replace the entire entity (PUT semantics), specify the **ReplaceOnUpdate** option. This causes WCF to send an HTTP PUT request.

```
container.SaveChanges(SaveChangesOptions.ReplaceOnUpdate);
```

Delete an Entity

To delete an entity, call **DeleteObject**.

```
static void DeleteProduct(ProductService.Container container, int id)
{
    var product = container.Products.Where(p => p.ID == id).SingleOrDefault();
    if (product != null)
    {
        container.DeleteObject(product);
        container.SaveChanges();
    }
}
```

Invoke an OData Action

In OData, actions are a way to add server-side behaviors that are not easily defined as CRUD operations on entities.

Although the OData metadata document describes the actions, the proxy class does not create any strongly-typed methods for them. You can still invoke an OData action by using the generic **Execute** method. However, you will

need to know the data types of the parameters and the return value.

For example, the `RateProduct` action takes parameter named "Rating" of type `Int32` and returns a `double`. The following code shows how to invoke this action.

```
int rating = 2;
Uri actionUri = new Uri(uri, "Products(5)/RateProduct");
var averageRating = container.Execute<double>(
    actionUri, "POST", true, new BodyOperationParameter("Rating", rating)).First();
```

For more information, see Calling Service Operations and Actions.

One option is to extend the **Container** class to provide a strongly typed method that invokes the action:

```
namespace ProductServiceClient.ProductService
{
    public partial class Container
    {
        public double RateProduct(int productID, int rating)
        {
            Uri actionUri = new Uri(this.BaseUri,
                String.Format("Products({0})/RateProduct", productID)
                );

            return this.Execute<double>(actionUri,
                "POST", true, new BodyOperationParameter("Rating", rating)).First();
        }
    }
}
```

Supporting OData Query Options in ASP.NET Web API 2

2/20/2020 • 6 minutes to read • Edit Online

by Mike Wasson

This overview with code examples demonstrates the supporting OData Query Options in ASP.NET Web API 2 for ASP.NET 4.x.

OData defines parameters that can be used to modify an OData query. The client sends these parameters in the query string of the request URI. For example, to sort the results, a client uses the $orderby parameter:

```
http://localhost/Products?$orderby=Name
```

The OData specification calls these parameters *query options*. You can enable OData query options for any Web API controller in your project — the controller does not need to be an OData endpoint. This gives you a convenient way to add features such as filtering and sorting to any Web API application.

Before enabling query options, please read the topic OData Security Guidance.

- Enabling OData Query Options
- Example Queries
- Server-Driven Paging
- Limiting the Query Options
- Invoking Query Options Directly
- Query Validation

Enabling OData Query Options

Web API supports the following OData query options:

OPTION	DESCRIPTION
$expand	Expands related entities inline.
$filter	Filters the results, based on a Boolean condition.
$inlinecount	Tells the server to include the total count of matching entities in the response. (Useful for server-side paging.)
$orderby	Sorts the results.
$select	Selects which properties to include in the response.
$skip	Skips the first n results.
$top	Returns only the first n the results.

To use OData query options, you must enable them explicitly. You can enable them globally for the entire application, or enable them for specific controllers or specific actions.

To enable OData query options globally, call **EnableQuerySupport** on the **HttpConfiguration** class at startup:

```
public static void Register(HttpConfiguration config)
{
    // ...

    config.EnableQuerySupport();

    // ...
}
```

The **EnableQuerySupport** method enables query options globally for any controller action that returns an **IQueryable** type. If you don't want query options enabled for the entire application, you can enable them for specific controller actions by adding the **[Queryable]** attribute to the action method.

```
public class ProductsController : ApiController
{
    [Queryable]
    IQueryable<Product> Get() {}
}
```

Example Queries

This section shows the types of queries that are possible using the OData query options. For specific details about the query options, refer to the OData documentation at www.odata.org.

For information about $expand and $select, see Using $select, $expand, and $value in ASP.NET Web API OData.

Client-Driven Paging

For large entity sets, the client might want to limit the number of results. For example, a client might show 10 entries at a time, with "next" links to get the next page of results. To do this, the client uses the $top and $skip options.

`http://localhost/Products?$top=10&$skip=20`

The $top option gives the maximum number of entries to return, and the $skip option gives the number of entries to skip. The previous example fetches entries 21 through 30.

Filtering

The $filter option lets a client filter the results by applying a Boolean expression. The filter expressions are quite powerful; they include logical and arithmetic operators, string functions, and date functions.

RETURN ALL PRODUCTS WITH CATEGORY EQUAL TO "TOYS".	`HTTP://LOCALHOST/PRODUCTS?$FILTER=CATEGORY` EQ 'TOYS'
Return all products with price less than 10.	`http://localhost/Products?$filter=Price` lt 10
Logical operators: Return all products where price >= 5 and price <= 15.	`http://localhost/Products?$filter=Price` ge 5 and Price le 15
String functions: Return all products with "zz" in the name.	`http://localhost/Products?$filter=substringof('zz',Name)`
Date functions: Return all products with ReleaseDate after 2005.	`http://localhost/Products?$filter=year(ReleaseDate)` gt 2005

Sorting

To sort the results, use the $orderby filter.

SORT BY PRICE.	`HTTP://LOCALHOST/PRODUCTS?$ORDERBY=PRICE`
Sort by price in descending order (highest to lowest).	`http://localhost/Products?$orderby=Price desc`
Sort by category, then sort by price in descending order within categories.	`http://localhost/odata/Products?$orderby=Category,Price desc`

Server-Driven Paging

If your database contains millions of records, you don't want to send them all in one payload. To prevent this, the server can limit the number of entries that it sends in a single response. To enable server paging, set the **PageSize** property in the **Queryable** attribute. The value is the maximum number of entries to return.

```
[Queryable(PageSize=10)]
public IQueryable<Product> Get()
{
    return products.AsQueryable();
}
```

If your controller returns OData format, the response body will contain a link to the next page of data:

```
{
  "odata.metadata":"http://localhost/$metadata#Products",
  "value":[
    { "ID":1,"Name":"Hat","Price":"15","Category":"Apparel" },
    { "ID":2,"Name":"Socks","Price":"5","Category":"Apparel" },
    // Others not shown
  ],
  "odata.nextLink":"http://localhost/Products?$skip=10"
}
```

The client can use this link to fetch the next page. To learn the total number of entries in the result set, the client can set the $inlinecount query option with the value "allpages".

`http://localhost/Products?$inlinecount=allpages`

The value "allpages" tells the server to include the total count in the response:

```
{
  "odata.metadata":"http://localhost/$metadata#Products",
  "odata.count":"50",
  "value":[
    { "ID":1,"Name":"Hat","Price":"15","Category":"Apparel" },
    { "ID":2,"Name":"Socks","Price":"5","Category":"Apparel" },
    // Others not shown
  ]
}
```

NOTE

Next-page links and inline count both require OData format. The reason is that OData defines special fields in the response body to hold the link and count.

For non-OData formats, it is still possible to support next-page links and inline count, by wrapping the query results in a **PageResult<T>** object. However, it requires a bit more code. Here is an example:

```
public PageResult<Product> Get(ODataQueryOptions<Product> options)
{
    ODataQuerySettings settings = new ODataQuerySettings()
    {
        PageSize = 5
    };

    IQueryable results = options.ApplyTo(_products.AsQueryable(), settings);

    return new PageResult<Product>(
        results as IEnumerable<Product>,
        Request.GetNextPageLink(),
        Request.GetInlineCount());
}
```

Here is an example JSON response:

```
{
  "Items": [
    { "ID":1,"Name":"Hat","Price":"15","Category":"Apparel" },
    { "ID":2,"Name":"Socks","Price":"5","Category":"Apparel" },

    // Others not shown

  ],
  "NextPageLink": "http://localhost/api/values?$inlinecount=allpages&$skip=10",
  "Count": 50
}
```

Limiting the Query Options

The query options give the client a lot of control over the query that is run on the server. In some cases, you might want to limit the available options for security or performance reasons. The **[Queryable]** attribute has some built in properties for this. Here are some examples.

Allow only $skip and $top, to support paging and nothing else:

```
[Queryable(AllowedQueryOptions=
    AllowedQueryOptions.Skip | AllowedQueryOptions.Top)]
```

Allow ordering only by certain properties, to prevent sorting on properties that are not indexed in the database:

```
[Queryable(AllowedOrderByProperties="Id")] // comma-separated list of properties
```

Allow the "eq" logical function but no other logical functions:

```
[Queryable(AllowedLogicalOperators=AllowedLogicalOperators.Equal)]
```

Do not allow any arithmetic operators:

```
[Queryable(AllowedArithmeticOperators=AllowedArithmeticOperators.None)]
```

You can restrict options globally by constructing a **QueryableAttribute** instance and passing it to the **EnableQuerySupport** function:

```
var queryAttribute = new QueryableAttribute()
{
    AllowedQueryOptions = AllowedQueryOptions.Top | AllowedQueryOptions.Skip,
    MaxTop = 100
};

config.EnableQuerySupport(queryAttribute);
```

Invoking Query Options Directly

Instead of using the **[Queryable]** attribute, you can invoke the query options directly in your controller. To do so, add an **ODataQueryOptions** parameter to the controller method. In this case, you don't need the **[Queryable]** attribute.

```
public IQueryable<Product> Get(ODataQueryOptions opts)
{
    var settings = new ODataValidationSettings()
    {
        // Initialize settings as needed.
        AllowedFunctions = AllowedFunctions.AllMathFunctions
    };

    opts.Validate(settings);

    IQueryable results = opts.ApplyTo(products.AsQueryable());
    return results as IQueryable<Product>;
}
```

Web API populates the **ODataQueryOptions** from the URI query string. To apply the query, pass an **IQueryable** to the **ApplyTo** method. The method returns another **IQueryable**.

For advanced scenarios, if you do not have an **IQueryable** query provider, you can examine the **ODataQueryOptions** and translate the query options into another form. (For example, see RaghuRam Nadiminti's blog post Translating OData queries to HQL, which also includes a sample.)

Query Validation

The **[Queryable]** attribute validates the query before executing it. The validation step is performed in the **QueryableAttribute.ValidateQuery** method. You can also customize the validation process.

Also see OData Security Guidance.

First, override one of the validator classes that is defined in the **Web.Http.OData.Query.Validators** namespace. For example, the following validator class disables the 'desc' option for the $orderby option.

```
public class MyOrderByValidator : OrderByQueryValidator
{
    // Disallow the 'desc' parameter for $orderby option.
    public override void Validate(OrderByQueryOption orderByOption,
                                  ODataValidationSettings validationSettings)
    {
        if (orderByOption.OrderByNodes.Any(
                node => node.Direction == OrderByDirection.Descending))
        {
            throw new ODataException("The 'desc' option is not supported.");
        }
        base.Validate(orderByOption, validationSettings);
    }
}
```

Subclass the **[Queryable]** attribute to override the **ValidateQuery** method.

```
public class MyQueryableAttribute : QueryableAttribute
{
    public override void ValidateQuery(HttpRequestMessage request,
        ODataQueryOptions queryOptions)
    {
        if (queryOptions.OrderBy != null)
        {
            queryOptions.OrderBy.Validator = new MyOrderByValidator();
        }
        base.ValidateQuery(request, queryOptions);
    }
}
```

Then set your custom attribute either globally or per-controller:

```
// Globally:
config.EnableQuerySupport(new MyQueryableAttribute());

// Per controller:
public class ValuesController : ApiController
{
    [MyQueryable]
    public IQueryable<Product> Get()
    {
        return products.AsQueryable();
    }
}
```

If you are using **ODataQueryOptions** directly, set the validator on the options:

```
public IQueryable<Product> Get(ODataQueryOptions opts)
{
    if (opts.OrderBy != null)
    {
        opts.OrderBy.Validator = new MyOrderByValidator();
    }

    var settings = new ODataValidationSettings()
    {
        // Initialize settings as needed.
        AllowedFunctions = AllowedFunctions.AllMathFunctions
    };

    // Validate
    opts.Validate(settings);

    IQueryable results = opts.ApplyTo(products.AsQueryable());
    return results as IQueryable<Product>;
}
```

Using $select, $expand, and $value in ASP.NET Web API 2 OData

2/20/2020 • 5 minutes to read • Edit Online

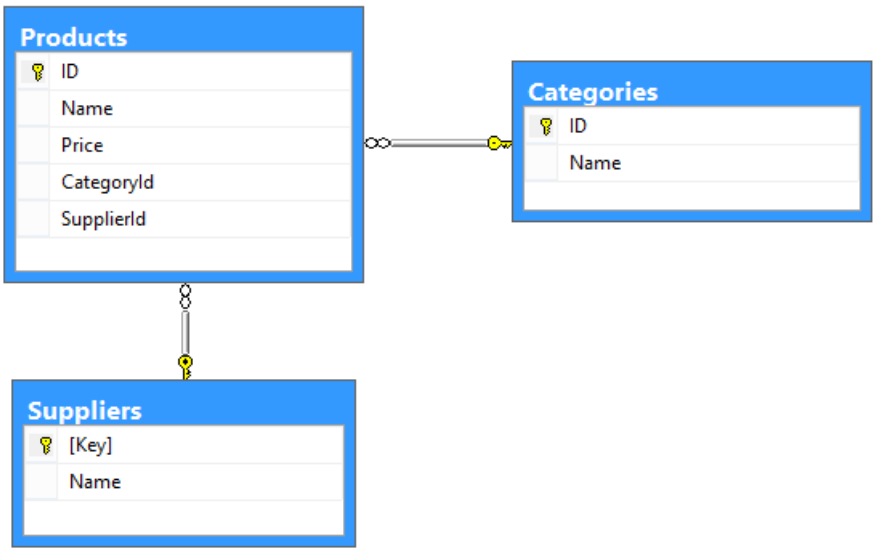
by Mike Wasson

Overview and code samples for the $expand, $select, and $value options in OData Web API 2 for ASP.NET 4.x. These options allow a client to control the representation that it gets back from the server.

- **$expand** causes related entities to be included inline in the response.
- **$select** selects a subset of properties to include in the response.
- **$value** gets the raw value of a property.

Example Schema

For this article, I'll use an OData service that defines three entities: Product, Supplier, and Category. Each product has one category and one supplier.



Here are the C# classes that define the entity models:

```
public class Supplier
{
    [Key]
    public string Key {get; set; }
    public string Name { get; set; }
}
public class Category
{
    public int ID { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Product> Products { get; set; }
}

public class Product
{
    public int ID { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }

    [ForeignKey("Category")]
    public int CategoryId { get; set; }
    public Category Category { get; set; }

    [ForeignKey("Supplier")]
    public string SupplierId { get; set; }
    public virtual Supplier Supplier { get; set; }
}
```

Notice that the `Product` class defines navigation properties for the `Supplier` and `Category`. The `Category` class defines a navigation property for the products in each category.

To create an OData endpoint for this schema, use the Visual Studio 2013 scaffolding, as described in Creating an OData Endpoint in ASP.NET Web API. Add separate controllers for Product, Category, and Supplier.

Enabling $expand and $select

In Visual Studio 2013, the Web API OData scaffolding creates a controller that automatically supports $expand and $select. For reference, here are the requirements to support $expand and $select in a controller.

For collections, the controller's `Get` method must return an **IQueryable**.

```
[Queryable]
public IQueryable<Category> GetCategories()
{
    return db.Categories;
}
```

For single entities, return a **SingleResult<T>**, where T is an **IQueryable** that contains zero or one entities.

```
[Queryable]
public SingleResult<Category> GetCategory([FromODataUri] int key)
{
    return SingleResult.Create(db.Categories.Where(c => c.ID == key));
}
```

Also, decorate your `Get` methods with the **[Queryable]** attribute, as shown in the previous code snippets. Alternatively, call **EnableQuerySupport** on the **HttpConfiguration** object at startup. (For more information, see Enabling OData Query Options.)

Using $expand

When you query an OData entity or collection, the default response does not include related entities. For example, here is the default response for the Categories entity set:

```
{
  "odata.metadata":"http://localhost/odata/$metadata#Categories",
  "value":[
    {"ID":1,"Name":"Apparel"},
    {"ID":2,"Name":"Toys"}
  ]
}
```

As you can see, the response does not include any products, even though the Category entity has a Products navigation link. However, the client can use $expand to get the list of products for each category. The $expand option goes in the query string of the request:

```
GET http://localhost/odata/Categories?$expand=Products
```

Now the server will include the products for each category, inline with the categories. Here is the response payload:

```
{
  "odata.metadata":"http://localhost/odata/$metadata#Categories",
  "value":[
    {
      "Products":[
        {"ID":1,"Name":"Hat","Price":"15.00","CategoryId":1,"SupplierId":"CTSO"},
        {"ID":2,"Name":"Scarf","Price":"12.00","CategoryId":1,"SupplierId":"CTSO"},
        {"ID":3,"Name":"Socks","Price":"5.00","CategoryId":1,"SupplierId":"FBRK"}
      ],
      "ID":1,
      "Name":"Apparel"
    },
    {
      "Products":[
        {"ID":4,"Name":"Yo-yo","Price":"4.95","CategoryId":2,"SupplierId":"WING"},
        {"ID":5,"Name":"Puzzle","Price":"8.00","CategoryId":2,"SupplierId":"WING"}
      ],
      "ID":2,
      "Name":"Toys"
    }
  ]
}
```

Notice that each entry in the "value" array contains a Products list.

The $expand option takes a comma-separated list of navigation properties to expand. The following request expands both the category and the supplier for a product.

```
GET http://localhost/odata/Products(1)?$expand=Category,Supplier
```

Here is the response body:

```
{
  "odata.metadata":"http://localhost/odata/$metadata#Products/@Element",
  "Category": {"ID":1,"Name":"Apparel"},
  "Supplier":{"Key":"CTSO","Name":"Contoso, Ltd."},
  "ID":1,
  "Name":"Hat",
  "Price":"15.00",
  "CategoryId":1,
  "SupplierId":"CTSO"
}
```

You can expand more than one level of navigation property. The following example includes all the products for a category and also the supplier for each product.

```
GET http://localhost/odata/Categories(1)?$expand=Products/Supplier
```

Here is the response body:

```
{
  "odata.metadata":"http://localhost/odata/$metadata#Categories/@Element",
  "Products":[
    {
      "Supplier":{"Key":"CTSO","Name":"Contoso, Ltd."},
      "ID":1,"Name":"Hat","Price":"15.00","CategoryId":1,"SupplierId":"CTSO"
    },
    {
      "Supplier":{"Key":"CTSO","Name":"Contoso, Ltd."},
      "ID":2,"Name":"Scarf","Price":"12.00","CategoryId":1,"SupplierId":"CTSO"
    },{
      "Supplier":{
        "Key":"FBRK","Name":"Fabrikam, Inc."
      },"ID":3,"Name":"Socks","Price":"5.00","CategoryId":1,"SupplierId":"FBRK"
    }
  ],"ID":1,"Name":"Apparel"
}
```

By default, Web API limits the maximum expansion depth to 2. That prevents the client from sending complex requests like `$expand=Orders/OrderDetails/Product/Supplier/Region`, which might be inefficient to query and create large responses. To override the default, set the **MaxExpansionDepth** property on the **[Queryable]** attribute.

```
[Queryable(MaxExpansionDepth=4)]
public IQueryable<Category> GetCategories()
{
    return db.Categories;
}
```

For more information about the $expand option, see Expand System Query Option ($expand) in the official OData documentation.

Using $select

The $select option specifies a subset of properties to include in the response body. For example, to get only the name and price of each product, use the following query:

```
GET http://localhost/odata/Products?$select=Price,Name
```

Here is the response body:

```
{
  "odata.metadata":"http://localhost/odata/$metadata#Products&$select=Price,Name",
  "value":[
    {"Price":"15.00","Name":"Hat"},
    {"Price":"12.00","Name":"Scarf"},
    {"Price":"5.00","Name":"Socks"},
    {"Price":"4.95","Name":"Yo-yo"},
    {"Price":"8.00","Name":"Puzzle"}
  ]
}
```

You can combine $select and $expand in the same query. Make sure to include the expanded property in the $select option. For example, the following request gets the product name and supplier.

```
GET http://localhost/odata/Products?$select=Name,Supplier&$expand=Supplier
```

Here is the response body:

```
{
  "odata.metadata":"http://localhost/odata/$metadata#Products&$select=Name,Supplier",
  "value":[
    {
      "Supplier":{"Key":"CTSO","Name":"Contoso, Ltd."},
      "Name":"Hat"
    },
    {
      "Supplier":{"Key":"CTSO","Name":"Contoso, Ltd."},
      "Name":"Scarf"
    },
    {
      "Supplier":{"Key":"FBRK","Name":"Fabrikam, Inc."},
      "Name":"Socks"
    },
    {
      "Supplier":{"Key":"WING","Name":"Wingtip Toys"},
      "Name":"Yo-yo"
    },
    {
      "Supplier":{"Key":"WING","Name":"Wingtip Toys"},
      "Name":"Puzzle"
   }
  ]
}
```

You can also select the properties within an expanded property. The following request expands Products and selects category name plus product name.

```
GET http://localhost/odata/Categories?$expand=Products&$select=Name,Products/Name
```

Here is the response body:

```
{
  "odata.metadata":"http://localhost/odata/$metadata#Categories&$select=Name,Products/Name",
  "value":[
    {
      "Products":[ {"Name":"Hat"},{"Name":"Scarf"},{"Name":"Socks"} ],
      "Name":"Apparel"
    },
    {
      "Products":[ {"Name":"Yo-yo"},{"Name":"Puzzle"} ],
      "Name":"Toys"
    }
  ]
}
```

For more information about the $select option, see Select System Query Option ($select) in the official OData documentation.

Getting Individual Properties of an Entity ($value)

There are two ways for an OData client to get an individual property from an entity. The client can either get the value in OData format, or get the raw value of the property.

The following request gets a property in OData format.

```
GET http://localhost/odata/Products(1)/Name
```

Here is an example response in JSON format:

```
HTTP/1.1 200 OK
Content-Type: application/json; odata=minimalmetadata; streaming=true; charset=utf-8
DataServiceVersion: 3.0
Content-Length: 90

{
  "odata.metadata":"http://localhost:14239/odata/$metadata#Edm.String",
  "value":"Hat"
}
```

To get the raw value of the property, append $value to the URI:

```
GET http://localhost/odata/Products(1)/Name/$value
```

Here is the response. Notice that the content type is "text/plain", not JSON.

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
DataServiceVersion: 3.0
Content-Length: 3

Hat
```

To support these queries in your OData controller, add a method named `GetProperty`, where `Property` is the name of the property. For example, the method to get the Name property would be named `GetName`. The method should return the value of that property:

```
public async Task<IHttpActionResult> GetName(int key)
{
    Product product = await db.Products.FindAsync(key);
    if (product == null)
    {
        return NotFound();
    }
    return Ok(product.Name);
}
```

Routing Conventions in ASP.NET Web API 2 Odata

2/20/2020 • 7 minutes to read • Edit Online

by Mike Wasson

> This article describes the routing conventions that Web API 2 in ASP.NET 4.x uses for OData endpoints.

When Web API gets an OData request, it maps the request to a controller name and an action name. The mapping is based on the HTTP method and the URI. For example, `GET /odata/Products(1)` maps to `ProductsController.GetProduct`.

In part 1 of this article, I describe the built-in OData routing conventions. These conventions are designed specifically for OData endpoints, and they replace the default Web API routing system. (The replacement happens when you call **MapODataRoute**.)

In part 2, I show how to add custom routing conventions. Currently the built-in conventions do not cover the entire range of OData URIs, but you can extend them to handle additional cases.

- Built-in Routing Conventions
- Custom Routing Conventions

Built-in Routing Conventions

Before I describe the OData routing conventions in Web API, it's helpful to understand OData URIs. An OData URI consists of:

- The service root
- The resource path
- Query options

http://example.com/odata	/Products(1)/Supplier	?$top=2
Service root	Resource path	Query options

For routing, the important part is the resource path. The resource path is divided into segments. For example, `/Products(1)/Supplier` has three segments:

- `Products` refers to an entity set named "Products".
- `1` is an entity key, selecting a single entity from the set.
- `Supplier` is a navigation property that selects a related entity.

So this path picks out the supplier of product 1.

> **NOTE**
>
> OData path segments do not always correspond to URI segments. For example, "1" is considered a path segment.

Controller Names. The controller name is always derived from the entity set at the root of the resource path. For example, if the resource path is `/Products(1)/Supplier`, Web API looks for a controller named `ProductsController`.

Action Names. Action names are derived from the path segments plus the entity data model (EDM), as listed in the following tables. In some cases, you have two choices for the action name. For example, "Get" or

"GetProducts".

Querying Entities

REQUEST	EXAMPLE URI	ACTION NAME	EXAMPLE ACTION
GET /entityset	/Products	GetEntitySet or Get	GetProducts
GET /entityset(key)	/Products(1)	GetEntityType or Get	GetProduct
GET /entityset(key)/cast	/Products(1)/Models.Book	GetEntityType or Get	GetBook

For more information, see Create a Read-Only OData Endpoint.

Creating, Updating, and Deleting Entities

REQUEST	EXAMPLE URI	ACTION NAME	EXAMPLE ACTION
POST /entityset	/Products	PostEntityType or Post	PostProduct
PUT /entityset(key)	/Products(1)	PutEntityType or Put	PutProduct
PUT /entityset(key)/cast	/Products(1)/Models.Book	PutEntityType or Put	PutBook
PATCH /entityset(key)	/Products(1)	PatchEntityType or Patch	PatchProduct
PATCH /entityset(key)/cast	/Products(1)/Models.Book	PatchEntityType or Patch	PatchBook
DELETE /entityset(key)	/Products(1)	DeleteEntityType or Delete	DeleteProduct
DELETE /entityset(key)/cast	/Products(1)/Models.Book	DeleteEntityType or Delete	DeleteBook

Querying a Navigation Property

REQUEST	EXAMPLE URI	ACTION NAME	EXAMPLE ACTION
GET /entityset(key)/navigation	/Products(1)/Supplier	GetNavigationFromEntityType or GetNavigation	GetSupplierFromProduct
GET /entityset(key)/cast/navigation	/Products(1)/Models.Book/Author	GetNavigationFromEntityType or GetNavigation	GetAuthorFromBook

For more information, see Working with Entity Relations.

Creating and Deleting Links

REQUEST	EXAMPLE URI	ACTION NAME
POST /entityset(key)/$links/navigation	/Products(1)/$links/Supplier	CreateLink
PUT /entityset(key)/$links/navigation	/Products(1)/$links/Supplier	CreateLink
DELETE /entityset(key)/$links/navigation	/Products(1)/$links/Supplier	DeleteLink

REQUEST	EXAMPLE URI	ACTION NAME
DELETE /entityset(key)/$links/navigation(relatedKey)	/Products/(1)/$links/Suppliers(1)	DeleteLink

For more information, see Working with Entity Relations.

Properties

Requires Web API 2

REQUEST	EXAMPLE URI	ACTION NAME	EXAMPLE ACTION
GET /entityset(key)/property	/Products(1)/Name	GetPropertyFromEntityType or GetProperty	GetNameFromProduct
GET /entityset(key)/cast/property	/Products(1)/Models.Book/Author	GetPropertyFromEntityType or GetProperty	GetTitleFromBook

Actions

REQUEST	EXAMPLE URI	ACTION NAME	EXAMPLE ACTION
POST /entityset(key)/action	/Products(1)/Rate	ActionNameOnEntityType or ActionName	RateOnProduct
POST /entityset(key)/cast/action	/Products(1)/Models.Book/CheckOut	ActionNameOnEntityType or ActionName	CheckOutOnBook

For more information, see OData Actions.

Method Signatures

Here are some rules for the method signatures:

- If the path contains a key, the action should have a parameter named *key*.
- If the path contains a key into a navigation property, the action should have a parameter named *relatedKey*.
- Decorate *key* and *relatedKey* parameters with the **[FromODataUri]** parameter.
- POST and PUT requests take a parameter of the entity type.
- PATCH requests take a parameter of type **Delta<T>**, where *T* is the entity type.

For reference, here is an example that shows method signatures for every built-in OData routing convention.

```
public class ProductsController : ODataController
{
    // GET /odata/Products
    public IQueryable<Product> Get()

    // GET /odata/Products(1)
    public Product Get([FromODataUri] int key)

    // GET /odata/Products(1)/ODataRouting.Models.Book
    public Book GetBook([FromODataUri] int key)

    // POST /odata/Products
    public HttpResponseMessage Post(Product item)

    // PUT /odata/Products(1)
    public HttpResponseMessage Put([FromODataUri] int key, Product item)

    // PATCH /odata/Products(1)
    public HttpResponseMessage Patch([FromODataUri] int key, Delta<Product> item)

    // DELETE /odata/Products(1)
    public HttpResponseMessage Delete([FromODataUri] int key)

    // PUT /odata/Products(1)/ODataRouting.Models.Book
    public HttpResponseMessage PutBook([FromODataUri] int key, Book item)

    // PATCH /odata/Products(1)/ODataRouting.Models.Book
    public HttpResponseMessage PatchBook([FromODataUri] int key, Delta<Book> item)

    // DELETE /odata/Products(1)/ODataRouting.Models.Book
    public HttpResponseMessage DeleteBook([FromODataUri] int key)

    //  GET /odata/Products(1)/Supplier
    public Supplier GetSupplierFromProduct([FromODataUri] int key)

    // GET /odata/Products(1)/ODataRouting.Models.Book/Author
    public Author GetAuthorFromBook([FromODataUri] int key)

    // POST /odata/Products(1)/$links/Supplier
    public HttpResponseMessage CreateLink([FromODataUri] int key,
        string navigationProperty, [FromBody] Uri link)

    // DELETE /odata/Products(1)/$links/Supplier
    public HttpResponseMessage DeleteLink([FromODataUri] int key,
        string navigationProperty, [FromBody] Uri link)

    // DELETE /odata/Products(1)/$links/Parts(1)
    public HttpResponseMessage DeleteLink([FromODataUri] int key, string relatedKey, string
navigationProperty)

    // GET odata/Products(1)/Name
    // GET odata/Products(1)/Name/$value
    public HttpResponseMessage GetNameFromProduct([FromODataUri] int key)

    // GET /odata/Products(1)/ODataRouting.Models.Book/Title
    // GET /odata/Products(1)/ODataRouting.Models.Book/Title/$value
    public HttpResponseMessage GetTitleFromBook([FromODataUri] int key)
}
```

Custom Routing Conventions

Currently the built-in conventions do not cover all possible OData URIs. You can add new conventions by implementing the **IODataRoutingConvention** interface. This interface has two methods:

```
string SelectController(ODataPath odataPath, HttpRequestMessage request);
string SelectAction(ODataPath odataPath, HttpControllerContext controllerContext,
    ILookup<string, HttpActionDescriptor> actionMap);
```

- **SelectController** returns the name of the controller.
- **SelectAction** returns the name of the action.

For both methods, if the convention does not apply to that request, the method should return null.

The **ODataPath** parameter represents the parsed OData resource path. It contains a list of **ODataPathSegment** instances, one for each segment of the resource path. **ODataPathSegment** is an abstract class; each segment type is represented by a class that derives from **ODataPathSegment**.

The **ODataPath.TemplatePath** property is a string that represents the concatenation all of the path segments. For example, if the URI is `/Products(1)/Supplier`, the path template is "~/entityset/key/navigation". Notice that the segments don't correspond directly to URI segments. For example, the entity key (1) is represented as its own **ODataPathSegment**.

Typically, an implementation of **IODataRoutingConvention** does the following:

1. Compare the path template to see if this convention applies to the current request. If it does not apply, return null.
2. If the convention applies, use properties of the **ODataPathSegment** instances to derive controller and action names.
3. For actions, add any values to the route dictionary that should bind to the action parameters (typically entity keys).

Let's look at a specific example. The built-in routing conventions do not support indexing into a navigation collection. In other words, there is no convention for URIs like the following:

```
/odata/Products(1)/Suppliers(1)
```

Here is a custom routing convention to handle this type of query.

```
using Microsoft.Data.Edm;
using System.Linq;
using System.Net.Http;
using System.Web.Http.Controllers;
using System.Web.Http.OData.Routing;
using System.Web.Http.OData.Routing.Conventions;

namespace ODataRouting
{
    public class NavigationIndexRoutingConvention : EntitySetRoutingConvention
    {
        public override string SelectAction(ODataPath odataPath, HttpControllerContext context,
            ILookup<string, HttpActionDescriptor> actionMap)
        {
            if (context.Request.Method == HttpMethod.Get &&
                odataPath.PathTemplate == "~/entityset/key/navigation/key")
            {
                NavigationPathSegment navigationSegment = odataPath.Segments[2] as NavigationPathSegment;
                IEdmNavigationProperty navigationProperty = navigationSegment.NavigationProperty.Partner;
                IEdmEntityType declaringType = navigationProperty.DeclaringType as IEdmEntityType;

                string actionName = "Get" + declaringType.Name;
                if (actionMap.Contains(actionName))
                {
                    // Add keys to route data, so they will bind to action parameters.
                    KeyValuePathSegment keyValueSegment = odataPath.Segments[1] as KeyValuePathSegment;
                    context.RouteData.Values[ODataRouteConstants.Key] = keyValueSegment.Value;

                    KeyValuePathSegment relatedKeySegment = odataPath.Segments[3] as KeyValuePathSegment;
                    context.RouteData.Values[ODataRouteConstants.RelatedKey] = relatedKeySegment.Value;

                    return actionName;
                }
            }
            // Not a match.
            return null;
        }
    }
}
```

Notes:

1. I derive from **EntitySetRoutingConvention**, because the **SelectController** method in that class is appropriate for this new routing convention. That means I don't need to re-implement **SelectController**.
2. The convention applies only to GET requests, and only when the path template is "~/entityset/key/navigation/key".
3. The action name is "Get{EntityType}", where *{EntityType}* is the type of the navigation collection. For example, "GetSupplier". You can use any naming convention that you like — just make sure your controller actions match.
4. The action takes two parameters named *key* and *relatedKey*. (For a list of some predefined parameter names, see ODataRouteConstants.)

The next step is adding the new convention to the list of routing conventions. This happens during configuration, as shown in the following code:

```
using ODataRouting.Models;
using System.Web.Http;
using System.Web.Http.OData.Builder;
using System.Web.Http.OData.Routing;
using System.Web.Http.OData.Routing.Conventions;

namespace ODataRouting
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            ODataModelBuilder modelBuilder = new ODataConventionModelBuilder();
            // Create EDM (not shown).

            // Create the default collection of built-in conventions.
            var conventions = ODataRoutingConventions.CreateDefault();
            // Insert the custom convention at the start of the collection.
            conventions.Insert(0, new NavigationIndexRoutingConvention());

            config.Routes.MapODataRoute(routeName: "ODataRoute",
                routePrefix: "odata",
                model: modelBuilder.GetEdmModel(),
                pathHandler: new DefaultODataPathHandler(),
                routingConventions: conventions);

        }
    }
}
```

Here are some other sample routing conventions that be useful to study:

- CompositeKeyRoutingConvention
- CustomNavigationRoutingConvention
- NonBindableActionRoutingConvention
- ODataVersionRouteConstraint

And of course Web API itself is open-source, so you can see the source code for the built-in routing conventions. These are defined in the **System.Web.Http.OData.Routing.Conventions** namespace.

Security Guidance for ASP.NET Web API 2 OData

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson

This topic describes some of the security issues that you should consider when exposing a dataset through OData for ASP.NET Web API 2 on ASP.NET 4.x.

EDM Security

The query semantics are based on the entity data model (EDM), not the underlying model types. You can exclude a property from the EDM and it will not be visible to the query. For example, suppose your model includes an Employee type with a Salary property. You might want to exclude this property from the EDM to hide it from clients.

There are two ways to exclude a property from the EDM. You can set the **[IgnoreDataMember]** attribute on the property in the model class:

```
public class Employee
{
    public string Name { get; set; }
    public string Title { get; set; }
    [IgnoreDataMember]
    public decimal Salary { get; set; } // Not visible in the EDM
}
```

You can also remove the property from the EDM programmatically:

```
var employees = modelBuilder.EntitySet<Employee>("Employees");
employees.EntityType.Ignore(emp => emp.Salary);
```

Query Security

A malicious or naive client may be able to construct a query that takes a very long time to execute. In the worst case this can disrupt access to your service.

The **[Queryable]** attribute is an action filter that parses, validates, and applies the query. The filter converts the query options into a LINQ expression. When the OData controller returns an **IQueryable** type, the **IQueryable** LINQ provider converts the LINQ expression into a query. Therefore, performance depends on the LINQ provider that is used, and also on the particular characteristics of your dataset or database schema.

For more information about using OData query options in ASP.NET Web API, see Supporting OData Query Options.

If you know that all clients are trusted (for example, in an enterprise environment), or if your dataset is small, query performance might not be an issue. Otherwise, you should consider the following recommendations.

- Test your service with various queries and profile the DB.
- Enable server-driven paging, to avoid returning a large data set in one query. For more information, see Server-Driven Paging.

```
// Enable server-driven paging.
[Queryable(PageSize=10)]
```

- Do you need $filter and $orderby? Some applications might allow client paging, using $top and $skip, but disable the other query options.

```
// Allow client paging but no other query options.
[Queryable(AllowedQueryOptions=AllowedQueryOptions.Skip |
                               AllowedQueryOptions.Top)]
```

- Consider restricting $orderby to properties in a clustered index. Sorting large data without a clustered index is slow.

```
// Set the allowed $orderby properties.
[Queryable(AllowedOrderByProperties="Id,Name")] // Comma separated list
```

- Maximum node count: The **MaxNodeCount** property on **[Queryable]** sets the maximum number nodes allowed in the $filter syntax tree. The default value is 100, but you may want to set a lower value, because a large number of nodes can be slow to compile. This is particularly true if you are using LINQ to Objects (i.e., LINQ queries on a collection in memory, without the use of an intermediate LINQ provider).

```
// Set the maximum node count.
[Queryable(MaxNodeCount=20)]
```

- Consider disabling the any() and all() functions, as these can be slow.

```
// Disable any() and all() functions.
[Queryable(AllowedFunctions= AllowedFunctions.AllFunctions &
    ~AllowedFunctions.All & ~AllowedFunctions.Any)]
```

- If any string properties contain large strings—for example, a product description or a blog entry—consider disabling the string functions.

```
// Disable string functions.
[Queryable(AllowedFunctions=AllowedFunctions.AllFunctions &
    ~AllowedFunctions.AllStringFunctions)]
```

- Consider disallowing filtering on navigation properties. Filtering on navigation properties can result in a join, which might be slow, depending on your database schema. The following code shows a query validator that prevents filtering on navigation properties. For more information about query validators, see Query Validation.

```
// Validator to prevent filtering on navigation properties.
public class MyFilterQueryValidator : FilterQueryValidator
{
    public override void ValidateNavigationPropertyNode(
        Microsoft.Data.OData.Query.SemanticAst.QueryNode sourceNode,
        Microsoft.Data.Edm.IEdmNavigationProperty navigationProperty,
        ODataValidationSettings settings)
    {
        throw new ODataException("No navigation properties");
    }
}
```

- Consider restricting $filter queries by writing a validator that is customized for your database. For example, consider these two queries:
  - All movies with actors whose last name starts with 'A'.
  - All movies released in 1994.

    Unless movies are indexed by actors, the first query might require the DB engine to scan the entire list of movies. Whereas the second query might be acceptable, assuming movies are indexed by release year.

    The following code shows a validator that allows filtering on the "ReleaseYear" and "Title" properties but no other properties.

```
// Validator to restrict which properties can be used in $filter expressions.
public class MyFilterQueryValidator : FilterQueryValidator
{
    static readonly string[] allowedProperties = { "ReleaseYear", "Title" };

    public override void ValidateSingleValuePropertyAccessNode(
        SingleValuePropertyAccessNode propertyAccessNode,
        ODataValidationSettings settings)
    {
        string propertyName = null;
        if (propertyAccessNode != null)
        {
            propertyName = propertyAccessNode.Property.Name;
        }

        if (propertyName != null && !allowedProperties.Contains(propertyName))
        {
            throw new ODataException(
                String.Format("Filter on {0} not allowed", propertyName));
        }
        base.ValidateSingleValuePropertyAccessNode(propertyAccessNode, settings);
    }
}
```

- In general, consider which $filter functions you need. If your clients do not need the full expressiveness of $filter, you can limit the allowed functions.

Serialization and Model Binding in ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

How ASP.NET Web API binds HTTP messages to domain models, and how to work with media formats in Web API.

- Media Formatters in Web API 2
- JSON and XML Serialization in Web API
- BSON Support in Web API 2.1
- Content Negotiation in Web API
- Model Validation in Web API
- Parameter Binding in Web API

Media Formatters in ASP.NET Web API 2

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson

This tutorial shows how to support additional media formats in ASP.NET Web API.

Internet Media Types

A media type, also called a MIME type, identifies the format of a piece of data. In HTTP, media types describe the format of the message body. A media type consists of two strings, a type and a subtype. For example:

- text/html
- image/png
- application/json

When an HTTP message contains an entity-body, the Content-Type header specifies the format of the message body. This tells the receiver how to parse the contents of the message body.

For example, if an HTTP response contains a PNG image, the response might have the following headers.

```
HTTP/1.1 200 OK
Content-Length: 95267
Content-Type: image/png
```

When the client sends a request message, it can include an Accept header. The Accept header tells the server which media type(s) the client wants from the server. For example:

```
Accept: text/html,application/xhtml+xml,application/xml
```

This header tells the server that the client wants either HTML, XHTML, or XML.

The media type determines how Web API serializes and deserializes the HTTP message body. Web API has built-in support for XML, JSON, BSON, and form-urlencoded data, and you can support additional media types by writing a *media formatter*.

To create a media formatter, derive from one of these classes:

- MediaTypeFormatter. This class uses asynchronous read and write methods.
- BufferedMediaTypeFormatter. This class derives from **MediaTypeFormatter** but uses synchronous read/write methods.

Deriving from **BufferedMediaTypeFormatter** is simpler, because there is no asynchronous code, but it also means the calling thread can block during I/O.

Example: Creating a CSV Media Formatter

The following example shows a media type formatter that can serialize a Product object to a comma-separated values (CSV) format. This example uses the Product type defined in the tutorial Creating a Web API that Supports CRUD Operations. Here is the definition of the Product object:

```
namespace ProductStore.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public decimal Price { get; set; }
    }
}
```

To implement a CSV formatter, define a class that derives from **BufferedMediaTypeFormatter**:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net.Http;
using System.Net.Http.Formatting;
using System.Net.Http.Headers;
using ProductStore.Models;

namespace ProductStore.Formatters
{
    public class ProductCsvFormatter : BufferedMediaTypeFormatter
    {
    }
}
```

In the constructor, add the media types that the formatter supports. In this example, the formatter supports a single media type, "text/csv":

```
public ProductCsvFormatter()
{
    // Add the supported media type.
    SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/csv"));
}
```

Override the **CanWriteType** method to indicate which types the formatter can serialize:

```
public override bool CanWriteType(System.Type type)
{
    if (type == typeof(Product))
    {
        return true;
    }
    else
    {
        Type enumerableType = typeof(IEnumerable<Product>);
        return enumerableType.IsAssignableFrom(type);
    }
}
```

In this example, the formatter can serialize single `Product` objects as well as collections of `Product` objects.

Similarly, override the **CanReadType** method to indicate which types the formatter can deserialize. In this example, the formatter does not support deserialization, so the method simply returns **false**.

```
public override bool CanReadType(Type type)
{
    return false;
}
```

Finally, override the **WriteToStream** method. This method serializes a type by writing it to a stream. If your formatter supports deserialization, also override the **ReadFromStream** method.

```
public override void WriteToStream(Type type, object value, Stream writeStream, HttpContent content)
{
    using (var writer = new StreamWriter(writeStream))
    {
        var products = value as IEnumerable<Product>;
        if (products != null)
        {
            foreach (var product in products)
            {
                WriteItem(product, writer);
            }
        }
        else
        {
            var singleProduct = value as Product;
            if (singleProduct == null)
            {
                throw new InvalidOperationException("Cannot serialize type");
            }
            WriteItem(singleProduct, writer);
        }
    }
}

// Helper methods for serializing Products to CSV format.
private void WriteItem(Product product, StreamWriter writer)
{
    writer.WriteLine("{0},{1},{2},{3}", Escape(product.Id),
        Escape(product.Name), Escape(product.Category), Escape(product.Price));
}

static char[] _specialChars = new char[] { ',', '\n', '\r', '"' };

private string Escape(object o)
{
    if (o == null)
    {
        return "";
    }
    string field = o.ToString();
    if (field.IndexOfAny(_specialChars) != -1)
    {
        // Delimit the entire field with quotes and replace embedded quotes with "".
        return String.Format("\"{0}\"", field.Replace("\"", "\"\""));
    }
    else return field;
}
```

Adding a Media Formatter to the Web API Pipeline

To add a media type formatter to the Web API pipeline, use the **Formatters** property on the **HttpConfiguration** object.

```
public static void ConfigureApis(HttpConfiguration config)
{
    config.Formatters.Add(new ProductCsvFormatter());
}
```

Character Encodings

Optionally, a media formatter can support multiple character encodings, such as UTF-8 or ISO 8859-1.

In the constructor, add one or more System.Text.Encoding types to the **SupportedEncodings** collection. Put the default encoding first.

```
public ProductCsvFormatter()
{
    SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/csv"));

    // New code:
    SupportedEncodings.Add(new UTF8Encoding(encoderShouldEmitUTF8Identifier: false));
    SupportedEncodings.Add(Encoding.GetEncoding("iso-8859-1"));
}
```

In the **WriteToStream** and **ReadFromStream** methods, call MediaTypeFormatter.SelectCharacterEncoding to select the preferred character encoding. This method matches the request headers against the list of supported encodings. Use the returned **Encoding** when you read or write from the stream:

```
public override void WriteToStream(Type type, object value, Stream writeStream, HttpContent content)
{
    Encoding effectiveEncoding = SelectCharacterEncoding(content.Headers);

    using (var writer = new StreamWriter(writeStream, effectiveEncoding))
    {
        // Write the object (code not shown)
    }
}
```

JSON and XML Serialization in ASP.NET Web API

4/6/2020 • 9 minutes to read • Edit Online

by Mike Wasson

This article describes the JSON and XML formatters in ASP.NET Web API.

In ASP.NET Web API, a *media-type formatter* is an object that can:

- Read CLR objects from an HTTP message body
- Write CLR objects into an HTTP message body

Web API provides media-type formatters for both JSON and XML. The framework inserts these formatters into the pipeline by default. Clients can request either JSON or XML in the Accept header of the HTTP request.

Contents

- JSON Media-Type Formatter
  - Read-Only Properties
  - Dates
  - Indenting
  - Camel Casing
  - Anonymous and Weakly-Typed Objects
- XML Media-Type Formatter
  - Read-Only Properties
  - Dates
  - Indenting
  - Setting Per-Type XML Serializers
- Removing the JSON or XML Formatter
- Handling Circular Object References
- Testing Object Serialization

JSON Media-Type Formatter

JSON formatting is provided by the **JsonMediaTypeFormatter** class. By default, **JsonMediaTypeFormatter** uses the Json.NET library to perform serialization. Json.NET is a third-party open source project.

If you prefer, you can configure the **JsonMediaTypeFormatter** class to use the **DataContractJsonSerializer** instead of Json.NET. To do so, set the **UseDataContractJsonSerializer** property to **true**:

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.UseDataContractJsonSerializer = true;
```

JSON Serialization

This section describes some specific behaviors of the JSON formatter, using the default Json.NET serializer. This is not meant to be comprehensive documentation of the Json.NET library; for more information, see the Json.NET Documentation.

What Gets Serialized?

By default, all public properties and fields are included in the serialized JSON. To omit a property or field, decorate it with the **JsonIgnore** attribute.

```
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    [JsonIgnore]
    public int ProductCode { get; set; } // omitted
}
```

If you prefer an "opt-in" approach, decorate the class with the **DataContract** attribute. If this attribute is present, members are ignored unless they have the **DataMember**. You can also use **DataMember** to serialize private members.

```
[DataContract]
public class Product
{
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public decimal Price { get; set; }
    public int ProductCode { get; set; }  // omitted by default
}
```

Read-Only Properties

Read-only properties are serialized by default.

Dates

By default, Json.NET writes dates in ISO 8601 format. Dates in UTC (Coordinated Universal Time) are written with a "Z" suffix. Dates in local time include a time-zone offset. For example:

```
2012-07-27T18:51:45.53403Z         // UTC
2012-07-27T11:51:45.53403-07:00    // Local
```

By default, Json.NET preserves the time zone. You can override this by setting the DateTimeZoneHandling property:

```
// Convert all dates to UTC
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.DateTimeZoneHandling = Newtonsoft.Json.DateTimeZoneHandling.Utc;
```

If you prefer to use Microsoft JSON date format (`"\/Date(ticks)\/"`) instead of ISO 8601, set the **DateFormatHandling** property on the serializer settings:

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.DateFormatHandling
= Newtonsoft.Json.DateFormatHandling.MicrosoftDateFormat;
```

Indenting

To write indented JSON, set the **Formatting** setting to **Formatting.Indented**:

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.Formatting = Newtonsoft.Json.Formatting.Indented;
```

Camel Casing

To write JSON property names with camel casing, without changing your data model, set the **CamelCasePropertyNamesContractResolver** on the serializer:

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();
```

Anonymous and Weakly-Typed Objects

An action method can return an anonymous object and serialize it to JSON. For example:

```
public object Get()
{
    return new {
        Name = "Alice",
        Age = 23,
        Pets = new List<string> { "Fido", "Polly", "Spot" }
    };
}
```

The response message body will contain the following JSON:

```
{"Name":"Alice","Age":23,"Pets":["Fido","Polly","Spot"]}
```

If your web API receives loosely structured JSON objects from clients, you can deserialize the request body to a **Newtonsoft.Json.Linq.JObject** type.

```
public void Post(JObject person)
{
    string name = person["Name"].ToString();
    int age = person["Age"].ToObject<int>();
}
```

However, it is usually better to use strongly typed data objects. Then you don't need to parse the data yourself, and you get the benefits of model validation.

The XML serializer does not support anonymous types or **JObject** instances. If you use these features for your JSON data, you should remove the XML formatter from the pipeline, as described later in this article.

XML Media-Type Formatter

XML formatting is provided by the **XmlMediaTypeFormatter** class. By default, **XmlMediaTypeFormatter** uses the **DataContractSerializer** class to perform serialization.

If you prefer, you can configure the **XmlMediaTypeFormatter** to use the **XmlSerializer** instead of the **DataContractSerializer**. To do so, set the **UseXmlSerializer** property to **true**:

```
var xml = GlobalConfiguration.Configuration.Formatters.XmlFormatter;
xml.UseXmlSerializer = true;
```

The **XmlSerializer** class supports a narrower set of types than **DataContractSerializer**, but gives more control

over the resulting XML. Consider using **XmlSerializer** if you need to match an existing XML schema.

XML Serialization

This section describes some specific behaviors of the XML formatter, using the default **DataContractSerializer**.

By default, the DataContractSerializer behaves as follows:

- All public read/write properties and fields are serialized. To omit a property or field, decorate it with the **IgnoreDataMember** attribute.
- Private and protected members are not serialized.
- Read-only properties are not serialized. (However, the contents of a read-only collection property are serialized.)
- Class and member names are written in the XML exactly as they appear in the class declaration.
- A default XML namespace is used.

If you need more control over the serialization, you can decorate the class with the **DataContract** attribute. When this attribute is present, the class is serialized as follows:

- "Opt in" approach: Properties and fields are not serialized by default. To serialize a property or field, decorate it with the **DataMember** attribute.
- To serialize a private or protected member, decorate it with the **DataMember** attribute.
- Read-only properties are not serialized.
- To change how the class name appears in the XML, set the *Name* parameter in the **DataContract** attribute.
- To change how a member name appears in the XML, set the *Name* parameter in the **DataMember** attribute.
- To change the XML namespace, set the *Namespace* parameter in the **DataContract** class.

Read-Only Properties

Read-only properties are not serialized. If a read-only property has a backing private field, you can mark the private field with the **DataMember** attribute. This approach requires the **DataContract** attribute on the class.

```
[DataContract]
public class Product
{
    [DataMember]
    private int pcode;  // serialized

    // Not serialized (read-only)
    public int ProductCode { get { return pcode; } }
}
```

Dates

Dates are written in ISO 8601 format. For example, "2012-05-23T20:21:37.9116538Z".

Indenting

To write indented XML, set the **Indent** property to **true**:

```
var xml = GlobalConfiguration.Configuration.Formatters.XmlFormatter;
xml.Indent = true;
```

Setting Per-Type XML Serializers

You can set different XML serializers for different CLR types. For example, you might have a particular data object that requires **XmlSerializer** for backward compatibility. You can use **XmlSerializer** for this object and continue to use **DataContractSerializer** for other types.

To set an XML serializer for a particular type, call **SetSerializer**.

```
var xml = GlobalConfiguration.Configuration.Formatters.XmlFormatter;
// Use XmlSerializer for instances of type "Product":
xml.SetSerializer<Product>(new XmlSerializer(typeof(Product)));
```

You can specify an **XmlSerializer** or any object that derives from **XmlObjectSerializer**.

Removing the JSON or XML Formatter

You can remove the JSON formatter or the XML formatter from the list of formatters, if you do not want to use them. The main reasons to do this are:

- To restrict your web API responses to a particular media type. For example, you might decide to support only JSON responses, and remove the XML formatter.
- To replace the default formatter with a custom formatter. For example, you could replace the JSON formatter with your own custom implementation of a JSON formatter.

The following code shows how to remove the default formatters. Call this from your **Application_Start** method, defined in Global.asax.

```
void ConfigureApi(HttpConfiguration config)
{
    // Remove the JSON formatter
    config.Formatters.Remove(config.Formatters.JsonFormatter);

    // or

    // Remove the XML formatter
    config.Formatters.Remove(config.Formatters.XmlFormatter);
}
```

Handling Circular Object References

By default, the JSON and XML formatters write all objects as values. If two properties refer to the same object, or if the same object appears twice in a collection, the formatter will serialize the object twice. This is a particular problem if your object graph contains cycles, because the serializer will throw an exception when it detects a loop in the graph.

Consider the following object models and controller.

```
public class Employee
{
    public string Name { get; set; }
    public Department Department { get; set; }
}

public class Department
{
    public string Name { get; set; }
    public Employee Manager { get; set; }
}

public class DepartmentsController : ApiController
{
    public Department Get(int id)
    {
        Department sales = new Department() { Name = "Sales" };
        Employee alice = new Employee() { Name = "Alice", Department = sales };
        sales.Manager = alice;
        return sales;
    }
}
```

Invoking this action will cause the formatter to throw an exception, which translates to a status code 500 (Internal Server Error) response to the client.

To preserve object references in JSON, add the following code to **Application_Start** method in the Global.asax file:

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.PreserveReferencesHandling =
    Newtonsoft.Json.PreserveReferencesHandling.All;
```

Now the controller action will return JSON that looks like this:

```
{"$id":"1","Name":"Sales","Manager":{"$id":"2","Name":"Alice","Department":{"$ref":"1"}}}
```

Notice that the serializer adds an "$id" property to both objects. Also, it detects that the Employee.Department property creates a loop, so it replaces the value with an object reference: {"$ref":"1"}.

> **NOTE**
> Object references are not standard in JSON. Before using this feature, consider whether your clients will be able to parse the results. It might be better simply to remove cycles from the graph. For example, the link from Employee back to Department is not really needed in this example.

To preserve object references in XML, you have two options. The simpler option is to add `[DataContract(IsReference=true)]` to your model class. The *IsReference* parameter enables object references. Remember that **DataContract** makes serialization opt-in, so you will also need to add **DataMember** attributes to the properties:

```
[DataContract(IsReference=true)]
public class Department
{
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public Employee Manager { get; set; }
}
```

Now the formatter will produce XML similar to following:

```
<Department xmlns:i="http://www.w3.org/2001/XMLSchema-instance" z:Id="i1"
            xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
            xmlns="http://schemas.datacontract.org/2004/07/Models">
  <Manager>
    <Department z:Ref="i1" />
    <Name>Alice</Name>
  </Manager>
  <Name>Sales</Name>
</Department>
```

If you want to avoid attributes on your model class, there is another option: Create a new type-specific **DataContractSerializer** instance and set *preserveObjectReferences* to **true** in the constructor. Then set this instance as a per-type serializer on the XML media-type formatter. The following code show how to do this:

```
var xml = GlobalConfiguration.Configuration.Formatters.XmlFormatter;
var dcs = new DataContractSerializer(typeof(Department), null, int.MaxValue,
    false, /* preserveObjectReferences: */ true, null);
xml.SetSerializer<Department>(dcs);
```

Testing Object Serialization

As you design your web API, it is useful to test how your data objects will be serialized. You can do this without creating a controller or invoking a controller action.

```
string Serialize<T>(MediaTypeFormatter formatter, T value)
{
    // Create a dummy HTTP Content.
    Stream stream = new MemoryStream();
    var content = new StreamContent(stream);
    /// Serialize the object.
    formatter.WriteToStreamAsync(typeof(T), value, stream, content, null).Wait();
    // Read the serialized string.
    stream.Position = 0;
    return content.ReadAsStringAsync().Result;
}

T Deserialize<T>(MediaTypeFormatter formatter, string str) where T : class
{
    // Write the serialized string to a memory stream.
    Stream stream = new MemoryStream();
    StreamWriter writer = new StreamWriter(stream);
    writer.Write(str);
    writer.Flush();
    stream.Position = 0;
    // Deserialize to an object of type T
    return formatter.ReadFromStreamAsync(typeof(T), stream, null, null).Result as T;
}

// Example of use
void TestSerialization()
{
    var value = new Person() { Name = "Alice", Age = 23 };

    var xml = new XmlMediaTypeFormatter();
    string str = Serialize(xml, value);

    var json = new JsonMediaTypeFormatter();
    str = Serialize(json, value);

    // Round trip
    Person person2 = Deserialize<Person>(json, str);
}
```

BSON Support in ASP.NET Web API 2.1

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson

This topic shows how to use BSON in your Web API controller (server side) and in a .NET client app. Web API 2.1 introduces support for BSON.

What is BSON?

BSON is a binary serialization format. "BSON" stands for "Binary JSON", but BSON and JSON are serialized very differently. BSON is "JSON-like", because objects are represented as name-value pairs, similar to JSON. Unlike JSON, numeric data types are stored as bytes, not strings

BSON was designed to be lightweight, easy to scan, and fast to encode/decode.

- BSON is comparable in size to JSON. Depending on the data, a BSON payload may be smaller or larger than a JSON payload. For serializing binary data, such as an image file, BSON is smaller than JSON, because the binary data is not base64-encoded.
- BSON documents are easy to scan because elements are prefixed with a length field, so a parser can skip elements without decoding them.
- Encoding and decoding are efficient, because numeric data types are stored as numbers, not strings.

Native clients, such as .NET client apps, can benefit from using BSON in place of text-based formats such as JSON or XML. For browser clients, you will probably want to stick with JSON, because JavaScript can directly convert the JSON payload.

Fortunately, Web API uses content negotiation, so your API can support both formats and let the client choose.

Enabling BSON on the Server

In your Web API configuration, add the **BsonMediaTypeFormatter** to the formatters collection.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Formatters.Add(new BsonMediaTypeFormatter());

        // Other Web API configuration not shown...
    }
}
```

Now if the client requests "application/bson", Web API will use the BSON formatter.

To associate BSON with other media types, add them to the SupportedMediaTypes collection. The following code adds "application/vnd.contoso" to the supported media types:

```
var bson = new BsonMediaTypeFormatter();
bson.SupportedMediaTypes.Add(new MediaTypeHeaderValue("application/vnd.contoso"));
config.Formatters.Add(bson);
```

Example HTTP Session

For this example, we'll use the following model class plus a simple Web API controller:

```
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Author { get; set; }
    public decimal Price { get; set; }
    public DateTime PublicationDate { get; set; }
}

public class BooksController : ApiController
{
    public IHttpActionResult GetBook(int id)
    {
        var book = new Book()
        {
            Id = id,
            Author = "Charles Dickens",
            Title = "Great Expectations",
            Price = 9.95M,
            PublicationDate = new DateTime(2014, 1, 20)
        };

        return Ok(book);
    }
}
```

A client might send the following HTTP request:

```
GET http://localhost:15192/api/books/1 HTTP/1.1
User-Agent: Fiddler
Host: localhost:15192
Accept: application/bson
```

Here is the response:

```
HTTP/1.1 200 OK
Content-Type: application/bson; charset=utf-8
Date: Fri, 17 Jan 2014 01:05:40 GMT
Content-Length: 111

.....Id......Title.....Great Expectations..Author.....Charles Dickens..Price..........PublicationDate.........
```

Here I've replaced the binary data with "." characters. The following screen shot from Fiddler shows the raw hex values.

```
48 54 54 50 2F 31 2E 31 20 32 30 30 20 4F 4B 0D 0A 43 61 63 68   HTTP/1.1 200 OK..Cach
65 2D 43 6F 6E 74 72 6F 6C 3A 20 6E 6F 2D 63 61 63 68 65 0D 0A   e-Control: no-cache..
50 72 61 67 6D 61 3A 20 6E 6F 2D 63 61 63 68 65 0D 0A 43 6F 6E   Pragma: no-cache..Con
74 65 6E 74 2D 54 79 70 65 3A 20 61 70 70 6C 69 63 61 74 69 6F   tent-Type: applicatio
6E 2F 62 73 6F 6E 3B 20 63 68 61 72 73 65 74 3D 75 74 66 2D 38   n/bson; charset=utf-8
0D 0A 45 78 70 69 72 65 73 3A 20 2D 31 0D 0A 53 65 72 76 65 72   ..Expires: -1..Server
3A 20 4D 69 63 72 6F 73 6F 66 74 2D 49 49 53 2F 38 2E 30 0D 0A   : Microsoft-IIS/8.0..
58 2D 41 73 70 4E 65 74 2D 56 65 72 73 69 6F 6E 3A 20 34 2E 30   X-AspNet-Version: 4.0
2E 33 30 33 31 39 0D 0A 58 2D 53 6F 75 72 63 65 46 69 6C 65 73   .30319..X-SourceFiles
3A 20 3D 3F 55 54 46 2D 38 3F 42 3F 51 7A 70 63 56 58 4E 6C 63   : =?UTF-8?B?QzpcVXNlc
6E 4E 63 62 58 64 68 63 33 4E 76 62 6C 78 45 62 32 4E 31 62 57   nNcbXdhc3NvblxEb2N1bW
56 75 64 48 4E 63 56 6D 6C 7A 64 57 46 73 49 46 4E 30 64 57 52   VudHNcVmlzdWFsIFN0dWR
70 62 79 41 79 4D 44 45 7A 58 46 42 79 62 32 70 6C 59 33 52 7A   pbyAyMDEzXFByb2plY3Rz
58 45 4A 7A 62 32 35 46 65 47 46 74 63 47 78 6C 58 45 4A 7A 62   XEJzb25FeGFtcGxlXEJzb
32 35 46 65 47 46 74 63 47 78 6C 58 47 46 77 61 56 78 69 62 32   25FeGFtcGxlXGFwaVxib2
39 72 63 31 77 78 3F 3D 0D 0A 58 2D 50 6F 77 65 72 65 64 2D 42   9rc1wx?=..X-Powered-B
79 3A 20 41 53 50 2E 4E 45 54 0D 0A 44 61 74 65 3A 20 46 72 69   y: ASP.NET..Date: Fri
2C 20 31 37 20 4A 61 6E 20 32 30 31 34 20 30 31 3A 30 35 3A 34   , 17 Jan 2014 01:05:4
30 20 47 4D 54 0D 0A 43 6F 6E 74 65 6E 74 2D 4C 65 6E 67 74 68   0 GMT..Content-Length
3A 20 31 31 31 0D 0A 0D 0A 6F 00 00 00 10 49 64 00 01 00 00 00   : 111....o....Id.....
02 54 69 74 6C 65 00 13 00 00 00 47 72 65 61 74 20 45 78 70 65   .Title.....Great Expe
63 74 61 74 69 6F 6E 73 00 02 41 75 74 68 6F 72 00 10 00 00 00   ctations..Author.....
43 68 61 72 6C 65 73 20 44 69 63 6B 65 6E 73 00 01 50 72 69 63   Charles Dickens..Pric
65 00 66 66 66 66 66 E6 23 40 09 50 75 62 6C 69 63 61 74 69 6F   e.fffffæ#@.Publicatio
6E 44 61 74 65 00 00 CC A9 AE 43 01 00 00 00                     nDate..Ì©®C....
```

Using BSON with HttpClient

.NET clients apps can use the BSON formatter with **HttpClient**. For more information about **HttpClient**, see Calling a Web API From a .NET Client.

The following code sends a GET request that accepts BSON, and then deserializes the BSON payload in the response.

```
static async Task RunAsync()
{
    using (HttpClient client = new HttpClient())
    {
        client.BaseAddress = new Uri("http://localhost");

        // Set the Accept header for BSON.
        client.DefaultRequestHeaders.Accept.Clear();
        client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/bson"));

        // Send GET request.
        result = await client.GetAsync("api/books/1");
        result.EnsureSuccessStatusCode();

        // Use BSON formatter to deserialize the result.
        MediaTypeFormatter[] formatters = new MediaTypeFormatter[] {
            new BsonMediaTypeFormatter()
        };

        var book = await result.Content.ReadAsAsync<Book>(formatters);
    }
}
```

To request BSON from the server, set the Accept header to "application/bson":

```
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(new
    MediaTypeWithQualityHeaderValue("application/bson"));
```

To deserialize the response body, use the **BsonMediaTypeFormatter**. This formatter is not in the default formatters collection, so you have to specify it when you read the response body:

```
MediaTypeFormatter[] formatters = new MediaTypeFormatter[] {
    new BsonMediaTypeFormatter()
};

var book = await result.Content.ReadAsAsync<Book>(formatters);
```

The next example shows how to send a POST request that contains BSON.

```
static async Task RunAsync()
{
    using (HttpClient client = new HttpClient())
    {
        client.BaseAddress = new Uri("http://localhost:15192");

        // Set the Accept header for BSON.
        client.DefaultRequestHeaders.Accept.Clear();
        client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/bson"));

        var book = new Book()
        {
            Author = "Jane Austen",
            Title = "Emma",
            Price = 9.95M,
            PublicationDate = new DateTime(1815, 1, 1)
        };

        // POST using the BSON formatter.
        MediaTypeFormatter bsonFormatter = new BsonMediaTypeFormatter();
        var result = await client.PostAsync("api/books", book, bsonFormatter);
        result.EnsureSuccessStatusCode();
    }
}
```

Much of this code is the same as the previous example. But in the **PostAsync** method, specify **BsonMediaTypeFormatter** as the formatter:

```
MediaTypeFormatter bsonFormatter = new BsonMediaTypeFormatter();
var result = await client.PostAsync("api/books", book, bsonFormatter);
```

Serializing Top-Level Primitive Types

Every BSON document is a list of key/value pairs.The BSON specification does not define a syntax for serializing a single raw value, such as an integer or string.

To work around this limitation, the **BsonMediaTypeFormatter** treats primitive types as a special case. Before serializing, it converts the value into a key/value pair with the key "Value". For example, suppose your API controller returns an integer:

```
public class ValuesController : ApiController
{
    public IHttpActionResult Get()
    {
        return Ok(42);
    }
}
```

Before serializing, the BSON formatter converts this to the following key/value pair:

```
{ "Value": 42 }
```

When you deserialize, the formatter converts the data back to the original value. However, clients using a different BSON parser will need to handle this case, if your web API returns raw values. In general, you should consider returning structured data, rather than raw values.

Additional Resources

Web API BSON Sample

Media Formatters

Content Negotiation in ASP.NET Web API

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson

This article describes how ASP.NET Web API implements content negotiation for ASP.NET 4.x.

The HTTP specification (RFC 2616) defines content negotiation as "the process of selecting the best representation for a given response when there are multiple representations available." The primary mechanism for content negotiation in HTTP are these request headers:

- **Accept:** Which media types are acceptable for the response, such as "application/json," "application/xml," or a custom media type such as "application/vnd.example+xml"
- **Accept-Charset:** Which character sets are acceptable, such as UTF-8 or ISO 8859-1.
- **Accept-Encoding:** Which content encodings are acceptable, such as gzip.
- **Accept-Language:** The preferred natural language, such as "en-us".

The server can also look at other portions of the HTTP request. For example, if the request contains an X-Requested-With header, indicating an AJAX request, the server might default to JSON if there is no Accept header.

In this article, we'll look at how Web API uses the Accept and Accept-Charset headers. (At this time, there is no built-in support for Accept-Encoding or Accept-Language.)

Serialization

If a Web API controller returns a resource as CLR type, the pipeline serializes the return value and writes it into the HTTP response body.

For example, consider the following controller action:

```
public Product GetProduct(int id)
{
    var item = _products.FirstOrDefault(p => p.ID == id);
    if (item == null)
    {
        throw new HttpResponseException(HttpStatusCode.NotFound);
    }
    return item;
}
```

A client might send this HTTP request:

```
GET http://localhost.:21069/api/products/1 HTTP/1.1
Host: localhost.:21069
Accept: application/json, text/javascript, */*; q=0.01
```

In response, the server might send:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 57
Connection: Close

{"Id":1,"Name":"Gizmo","Category":"Widgets","Price":1.99}
```

In this example, the client requested either JSON, Javascript, or "anything" (*/*). The server responded with a JSON representation of the `Product` object. Notice that the Content-Type header in the response is set to "application/json".

A controller can also return an **HttpResponseMessage** object. To specify a CLR object for the response body, call the **CreateResponse** extension method:

```
public HttpResponseMessage GetProduct(int id)
{
    var item = _products.FirstOrDefault(p => p.ID == id);
    if (item == null)
    {
        throw new HttpResponseException(HttpStatusCode.NotFound);
    }
    return Request.CreateResponse(HttpStatusCode.OK, item);
}
```

This option gives you more control over the details of the response. You can set the status code, add HTTP headers, and so forth.

The object that serializes the resource is called a *media formatter*. Media formatters derive from the **MediaTypeFormatter** class. Web API provides media formatters for XML and JSON, and you can create custom formatters to support other media types. For information about writing a custom formatter, see Media Formatters.

How Content Negotiation Works

First, the pipeline gets the **IContentNegotiator** service from the **HttpConfiguration** object. It also gets the list of media formatters from the **HttpConfiguration.Formatters** collection.

Next, the pipeline calls **IContentNegotiator.Negotiate**, passing in:

- The type of object to serialize
- The collection of media formatters
- The HTTP request

The **Negotiate** method returns two pieces of information:

- Which formatter to use
- The media type for the response

If no formatter is found, the **Negotiate** method returns **null**, and the client receives HTTP error 406 (Not Acceptable).

The following code shows how a controller can directly invoke content negotiation:

```
public HttpResponseMessage GetProduct(int id)
{
    var product = new Product()
        { Id = id, Name = "Gizmo", Category = "Widgets", Price = 1.99M };

    IContentNegotiator negotiator = this.Configuration.Services.GetContentNegotiator();

    ContentNegotiationResult result = negotiator.Negotiate(
        typeof(Product), this.Request, this.Configuration.Formatters);
    if (result == null)
    {
        var response = new HttpResponseMessage(HttpStatusCode.NotAcceptable);
        throw new HttpResponseException(response));
    }

    return new HttpResponseMessage()
    {
        Content = new ObjectContent<Product>(
            product,          // What we are serializing
            result.Formatter,           // The media formatter
            result.MediaType.MediaType  // The MIME type
        )
    };
}
```

This code is equivalent to the what the pipeline does automatically.

Default Content Negotiator

The **DefaultContentNegotiator** class provides the default implementation of **IContentNegotiator**. It uses several criteria to select a formatter.

First, the formatter must be able to serialize the type. This is verified by calling **MediaTypeFormatter.CanWriteType**.

Next, the content negotiator looks at each formatter and evaluates how well it matches the HTTP request. To evaluate the match, the content negotiator looks at two things on the formatter:

- The **SupportedMediaTypes** collection, which contains a list of supported media types. The content negotiator tries to match this list against the request Accept header. Note that the Accept header can include ranges. For example, "text/plain" is a match for text/* or */*.
- The **MediaTypeMappings** collection, which contains a list of **MediaTypeMapping** objects. The **MediaTypeMapping** class provides a generic way to match HTTP requests with media types. For example, it could map a custom HTTP header to a particular media type.

If there are multiple matches, the match with the highest quality factor wins. For example:

```
Accept: application/json, application/xml; q=0.9, */*; q=0.1
```

In this example, application/json has an implied quality factor of 1.0, so it is preferred over application/xml.

If no matches are found, the content negotiator tries to match on the media type of the request body, if any. For example, if the request contains JSON data, the content negotiator looks for a JSON formatter.

If there are still no matches, the content negotiator simply picks the first formatter that can serialize the type.

Selecting a Character Encoding

After a formatter is selected, the content negotiator chooses the best character encoding by looking at the

SupportedEncodings property on the formatter, and matching it against the Accept-Charset header in the request (if any).

Model Validation in ASP.NET Web API

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson

This article shows how to annotate your models, use the annotations for data validation, and handle validation errors in your web API. When a client sends data to your web API, often you want to validate the data before doing any processing.

Data Annotations

In ASP.NET Web API, you can use attributes from the System.ComponentModel.DataAnnotations namespace to set validation rules for properties on your model. Consider the following model:

```
using System.ComponentModel.DataAnnotations;

namespace MyApi.Models
{
    public class Product
    {
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
        public decimal Price { get; set; }
        [Range(0, 999)]
        public double Weight { get; set; }
    }
}
```

If you have used model validation in ASP.NET MVC, this should look familiar. The **Required** attribute says that the `Name` property must not be null. The **Range** attribute says that `Weight` must be between zero and 999.

Suppose that a client sends a POST request with the following JSON representation:

```
{ "Id":4, "Price":2.99, "Weight":5 }
```

You can see that the client did not include the `Name` property, which is marked as required. When Web API converts the JSON into a `Product` instance, it validates the `Product` against the validation attributes. In your controller action, you can check whether the model is valid:

```
using MyApi.Models;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace MyApi.Controllers
{
    public class ProductsController : ApiController
    {
        public HttpResponseMessage Post(Product product)
        {
            if (ModelState.IsValid)
            {
                // Do something with the product (not shown).

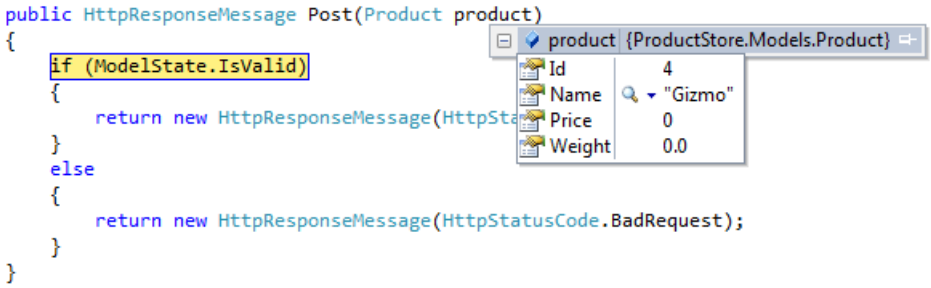
                return new HttpResponseMessage(HttpStatusCode.OK);
            }
            else
            {
                return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);
            }
        }
    }
}
```

Model validation does not guarantee that client data is safe. Additional validation might be needed in other layers of the application. (For example, the data layer might enforce foreign key constraints.) The tutorial Using Web API with Entity Framework explores some of these issues.

"Under-Posting": Under-posting happens when the client leaves out some properties. For example, suppose the client sends the following:

```
{"Id":4, "Name":"Gizmo"}
```

Here, the client did not specify values for `Price` or `Weight`. The JSON formatter assigns a default value of zero to the missing properties.



The model state is valid, because zero is a valid value for these properties. Whether this is a problem depends on your scenario. For example, in an update operation, you might want to distinguish between "zero" and "not set." To force clients to set a value, make the property nullable and set the **Required** attribute:

```
[Required]
public decimal? Price { get; set; }
```

"Over-Posting": A client can also send *more* data than you expected. For example:

```
{"Id":4, "Name":"Gizmo", "Color":"Blue"}
```

Here, the JSON includes a property ("Color") that does not exist in the `Product` model. In this case, the JSON formatter simply ignores this value. (The XML formatter does the same.) Over-posting causes problems if your model has properties that you intended to be read-only. For example:

```
public class UserProfile
{
    public string Name { get; set; }
    public Uri Blog { get; set; }
    public bool IsAdmin { get; set; }  // uh-oh!
}
```

You don't want users to update the `IsAdmin` property and elevate themselves to administrators! The safest strategy is to use a model class that exactly matches what the client is allowed to send:

```
public class UserProfileDTO
{
    public string Name { get; set; }
    public Uri Blog { get; set; }
    // Leave out "IsAdmin"
}
```

NOTE

Brad Wilson's blog post "Input Validation vs. Model Validation in ASP.NET MVC" has a good discussion of under-posting and over-posting. Although the post is about ASP.NET MVC 2, the issues are still relevant to Web API.

Handling Validation Errors

Web API does not automatically return an error to the client when validation fails. It is up to the controller action to check the model state and respond appropriately.

You can also create an action filter to check the model state before the controller action is invoked. The following code shows an example:

```
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http.Controllers;
using System.Web.Http.Filters;
using System.Web.Http.ModelBinding;

namespace MyApi.Filters
{
    public class ValidateModelAttribute : ActionFilterAttribute
    {
        public override void OnActionExecuting(HttpActionContext actionContext)
        {
            if (actionContext.ModelState.IsValid == false)
            {
                actionContext.Response = actionContext.Request.CreateErrorResponse(
                    HttpStatusCode.BadRequest, actionContext.ModelState);
            }
        }
    }
}
```

If model validation fails, this filter returns an HTTP response that contains the validation errors. In that case, the

controller action is not invoked.

```
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Date: Tue, 16 Jul 2013 21:02:29 GMT
Content-Length: 331

{
  "Message": "The request is invalid.",
  "ModelState": {
    "product": [
      "Required property 'Name' not found in JSON. Path '', line 1, position 17."
    ],
    "product.Name": [
      "The Name field is required."
    ],
    "product.Weight": [
      "The field Weight must be between 0 and 999."
    ]
  }
}
```

To apply this filter to all Web API controllers, add an instance of the filter to the **HttpConfiguration.Filters** collection during configuration:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Filters.Add(new ValidateModelAttribute());

        // ...
    }
}
```

Another option is to set the filter as an attribute on individual controllers or controller actions:

```
public class ProductsController : ApiController
{
    [ValidateModel]
    public HttpResponseMessage Post(Product product)
    {
        // ...
    }
}
```

Parameter Binding in ASP.NET Web API

2/20/2020 • 12 minutes to read • Edit Online

by Mike Wasson

Consider using ASP.NET Core web API. It has the following advantages over ASP.NET 4.x Web API:

- ASP.NET Core is an open-source, cross-platform framework for building modern, cloud-based web apps on Windows, macOS, and Linux.
- The ASP.NET Core MVC controllers and web API controllers are unified.
- Architected for testability.
- Ability to develop and run on Windows, macOS, and Linux.
- Open-source and community-focused.
- Integration of modern, client-side frameworks and development workflows.
- A cloud-ready, environment-based configuration system.
- Built-in dependency injection.
- A lightweight, high-performance, and modular HTTP request pipeline.
- Ability to host on Kestrel, IIS, HTTP.sys, Nginx, Apache, and Docker.
- Side-by-side versioning.
- Tooling that simplifies modern web development.

This article describes how Web API binds parameters, and how you can customize the binding process. When Web API calls a method on a controller, it must set values for the parameters, a process called *binding*.

By default, Web API uses the following rules to bind parameters:

- If the parameter is a "simple" type, Web API tries to get the value from the URI. Simple types include the .NET primitive types (**int**, **bool**, **double**, and so forth), plus **TimeSpan**, **DateTime**, **Guid**, **decimal**, and **string**, *plus* any type with a type converter that can convert from a string. (More about type converters later.)
- For complex types, Web API tries to read the value from the message body, using a media-type formatter.

For example, here is a typical Web API controller method:

```
HttpResponseMessage Put(int id, Product item) { ... }
```

The *id* parameter is a "simple" type, so Web API tries to get the value from the request URI. The *item* parameter is a complex type, so Web API uses a media-type formatter to read the value from the request body.

To get a value from the URI, Web API looks in the route data and the URI query string. The route data is populated when the routing system parses the URI and matches it to a route. For more information, see Routing and Action Selection.

In the rest of this article, I'll show how you can customize the model binding process. For complex types, however, consider using media-type formatters whenever possible. A key principle of HTTP is that resources are sent in the message body, using content negotiation to specify the representation of the resource. Media-type formatters were designed for exactly this purpose.

Using [FromUri]

To force Web API to read a complex type from the URI, add the **[FromUri]** attribute to the parameter. The

following example defines a `GeoPoint` type, along with a controller method that gets the `GeoPoint` from the URI.

```
public class GeoPoint
{
    public double Latitude { get; set; }
    public double Longitude { get; set; }
}

public ValuesController : ApiController
{
    public HttpResponseMessage Get([FromUri] GeoPoint location) { ... }
}
```

The client can put the Latitude and Longitude values in the query string and Web API will use them to construct a `GeoPoint`. For example:

`http://localhost/api/values/?Latitude=47.678558&Longitude=-122.130989`

Using [FromBody]

To force Web API to read a simple type from the request body, add the **[FromBody]** attribute to the parameter:

```
public HttpResponseMessage Post([FromBody] string name) { ... }
```

In this example, Web API will use a media-type formatter to read the value of *name* from the request body. Here is an example client request.

```
POST http://localhost:5076/api/values HTTP/1.1
User-Agent: Fiddler
Host: localhost:5076
Content-Type: application/json
Content-Length: 7

"Alice"
```

When a parameter has [FromBody], Web API uses the Content-Type header to select a formatter. In this example, the content type is "application/json" and the request body is a raw JSON string (not a JSON object).

At most one parameter is allowed to read from the message body. So this will not work:

```
// Caution: Will not work!
public HttpResponseMessage Post([FromBody] int id, [FromBody] string name) { ... }
```

The reason for this rule is that the request body might be stored in a non-buffered stream that can only be read once.

Type Converters

You can make Web API treat a class as a simple type (so that Web API will try to bind it from the URI) by creating a **TypeConverter** and providing a string conversion.

The following code shows a `GeoPoint` class that represents a geographical point, plus a **TypeConverter** that converts from strings to `GeoPoint` instances. The `GeoPoint` class is decorated with a **[TypeConverter]** attribute to specify the type converter. (This example was inspired by Mike Stall's blog post How to bind to custom objects in action signatures in MVC/WebAPI.)

```
[TypeConverter(typeof(GeoPointConverter))]
public class GeoPoint
{
    public double Latitude { get; set; }
    public double Longitude { get; set; }

    public static bool TryParse(string s, out GeoPoint result)
    {
        result = null;

        var parts = s.Split(',');
        if (parts.Length != 2)
        {
            return false;
        }

        double latitude, longitude;
        if (double.TryParse(parts[0], out latitude) &&
            double.TryParse(parts[1], out longitude))
        {
            result = new GeoPoint() { Longitude = longitude, Latitude = latitude };
            return true;
        }
        return false;
    }
}

class GeoPointConverter : TypeConverter
{
    public override bool CanConvertFrom(ITypeDescriptorContext context, Type sourceType)
    {
        if (sourceType == typeof(string))
        {
            return true;
        }
        return base.CanConvertFrom(context, sourceType);
    }

    public override object ConvertFrom(ITypeDescriptorContext context,
        CultureInfo culture, object value)
    {
        if (value is string)
        {
            GeoPoint point;
            if (GeoPoint.TryParse((string)value, out point))
            {
                return point;
            }
        }
        return base.ConvertFrom(context, culture, value);
    }
}
```

Now Web API will treat `GeoPoint` as a simple type, meaning it will try to bind `GeoPoint` parameters from the URI. You don't need to include **[FromUri]** on the parameter.

```
public HttpResponseMessage Get(GeoPoint location) { ... }
```

The client can invoke the method with a URI like this:

`http://localhost/api/values/?location=47.678558,-122.130989`

Model Binders

A more flexible option than a type converter is to create a custom model binder. With a model binder, you have access to things like the HTTP request, the action description, and the raw values from the route data.

To create a model binder, implement the **IModelBinder** interface. This interface defines a single method, **BindModel**:

```
bool BindModel(HttpActionContext actionContext, ModelBindingContext bindingContext);
```

Here is a model binder for `GeoPoint` objects.

```
public class GeoPointModelBinder : IModelBinder
{
    // List of known locations.
    private static ConcurrentDictionary<string, GeoPoint> _locations
        = new ConcurrentDictionary<string, GeoPoint>(StringComparer.OrdinalIgnoreCase);

    static GeoPointModelBinder()
    {
        _locations["redmond"] = new GeoPoint() { Latitude = 47.67856, Longitude = -122.131 };
        _locations["paris"] = new GeoPoint() { Latitude = 48.856930, Longitude = 2.3412 };
        _locations["tokyo"] = new GeoPoint() { Latitude = 35.683208, Longitude = 139.80894 };
    }

    public bool BindModel(HttpActionContext actionContext, ModelBindingContext bindingContext)
    {
        if (bindingContext.ModelType != typeof(GeoPoint))
        {
            return false;
        }

        ValueProviderResult val = bindingContext.ValueProvider.GetValue(
            bindingContext.ModelName);
        if (val == null)
        {
            return false;
        }

        string key = val.RawValue as string;
        if (key == null)
        {
            bindingContext.ModelState.AddModelError(
                bindingContext.ModelName, "Wrong value type");
            return false;
        }

        GeoPoint result;
        if (_locations.TryGetValue(key, out result) || GeoPoint.TryParse(key, out result))
        {
            bindingContext.Model = result;
            return true;
        }

        bindingContext.ModelState.AddModelError(
            bindingContext.ModelName, "Cannot convert value to GeoPoint");
        return false;
    }
}
```

A model binder gets raw input values from a *value provider*. This design separates two distinct functions:

- The value provider takes the HTTP request and populates a dictionary of key-value pairs.
- The model binder uses this dictionary to populate the model.

The default value provider in Web API gets values from the route data and the query string. For example, if the URI is `http://localhost/api/values/1?location=48,-122`, the value provider creates the following key-value pairs:

- id = "1"
- location = "48,122"

(I'm assuming the default route template, which is "api/{controller}/{id}".)

The name of the parameter to bind is stored in the **ModelBindingContext.ModelName** property. The model binder looks for a key with this value in the dictionary. If the value exists and can be converted into a `GeoPoint`, the model binder assigns the bound value to the **ModelBindingContext.Model** property.

Notice that the model binder is not limited to a simple type conversion. In this example, the model binder first looks in a table of known locations, and if that fails, it uses type conversion.

Setting the Model Binder

There are several ways to set a model binder. First, you can add a **[ModelBinder]** attribute to the parameter.

```
public HttpResponseMessage Get([ModelBinder(typeof(GeoPointModelBinder))] GeoPoint location)
```

You can also add a **[ModelBinder]** attribute to the type. Web API will use the specified model binder for all parameters of that type.

```
[ModelBinder(typeof(GeoPointModelBinder))]
public class GeoPoint
{
    // ....
}
```

Finally, you can add a model-binder provider to the **HttpConfiguration**. A model-binder provider is simply a factory class that creates a model binder. You can create a provider by deriving from the ModelBinderProvider class. However, if your model binder handles a single type, it's easier to use the built-in **SimpleModelBinderProvider**, which is designed for this purpose. The following code shows how to do this.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        var provider = new SimpleModelBinderProvider(
            typeof(GeoPoint), new GeoPointModelBinder());
        config.Services.Insert(typeof(ModelBinderProvider), 0, provider);

        // ...
    }
}
```

With a model-binding provider, you still need to add the **[ModelBinder]** attribute to the parameter, to tell Web API that it should use a model binder and not a media-type formatter. But now you don't need to specify the type of model binder in the attribute:

```
public HttpResponseMessage Get([ModelBinder] GeoPoint location) { ... }
```

Value Providers

I mentioned that a model binder gets values from a value provider. To write a custom value provider, implement

the **IValueProvider** interface. Here is an example that pulls values from the cookies in the request:

```
public class CookieValueProvider : IValueProvider
{
    private Dictionary<string, string> _values;

    public CookieValueProvider(HttpActionContext actionContext)
    {
        if (actionContext == null)
        {
            throw new ArgumentNullException("actionContext");
        }

        _values = new Dictionary<string, string>(StringComparer.OrdinalIgnoreCase);
        foreach (var cookie in actionContext.Request.Headers.GetCookies())
        {
            foreach (CookieState state in cookie.Cookies)
            {
                _values[state.Name] = state.Value;
            }
        }
    }

    public bool ContainsPrefix(string prefix)
    {
        return _values.Keys.Contains(prefix);
    }

    public ValueProviderResult GetValue(string key)
    {
        string value;
        if (_values.TryGetValue(key, out value))
        {
            return new ValueProviderResult(value, value, CultureInfo.InvariantCulture);
        }
        return null;
    }
}
```

You also need to create a value provider factory by deriving from the **ValueProviderFactory** class.

```
public class CookieValueProviderFactory : ValueProviderFactory
{
    public override IValueProvider GetValueProvider(HttpActionContext actionContext)
    {
        return new CookieValueProvider(actionContext);
    }
}
```

Add the value provider factory to the **HttpConfiguration** as follows.

```
public static void Register(HttpConfiguration config)
{
    config.Services.Add(typeof(ValueProviderFactory), new CookieValueProviderFactory());

    // ...
}
```

Web API composes all of the value providers, so when a model binder calls **ValueProvider.GetValue**, the model binder receives the value from the first value provider that is able to produce it.

Alternatively, you can set the value provider factory at the parameter level by using the **ValueProvider** attribute,

as follows:

```
public HttpResponseMessage Get(
    [ValueProvider(typeof(CookieValueProviderFactory))] GeoPoint location)
```

This tells Web API to use model binding with the specified value provider factory, and not to use any of the other registered value providers.

HttpParameterBinding

Model binders are a specific instance of a more general mechanism. If you look at the **[ModelBinder]** attribute, you will see that it derives from the abstract **ParameterBindingAttribute** class. This class defines a single method, **GetBinding**, which returns an **HttpParameterBinding** object:

```
public abstract class ParameterBindingAttribute : Attribute
{
    public abstract HttpParameterBinding GetBinding(HttpParameterDescriptor parameter);
}
```

An **HttpParameterBinding** is responsible for binding a parameter to a value. In the case of **[ModelBinder]**, the attribute returns an **HttpParameterBinding** implementation that uses an **IModelBinder** to perform the actual binding. You can also implement your own **HttpParameterBinding**.

For example, suppose you want to get ETags from `if-match` and `if-none-match` headers in the request. We'll start by defining a class to represent ETags.

```
public class ETag
{
    public string Tag { get; set; }
}
```

We'll also define an enumeration to indicate whether to get the ETag from the `if-match` header or the `if-none-match` header.

```
public enum ETagMatch
{
    IfMatch,
    IfNoneMatch
}
```

Here is an **HttpParameterBinding** that gets the ETag from the desired header and binds it to a parameter of type ETag:

```
public class ETagParameterBinding : HttpParameterBinding
{
    ETagMatch _match;

    public ETagParameterBinding(HttpParameterDescriptor parameter, ETagMatch match)
        : base(parameter)
    {
        _match = match;
    }

    public override Task ExecuteBindingAsync(ModelMetadataProvider metadataProvider,
        HttpActionContext actionContext, CancellationToken cancellationToken)
    {
        EntityTagHeaderValue etagHeader = null;
        switch (_match)
        {
            case ETagMatch.IfNoneMatch:
                etagHeader = actionContext.Request.Headers.IfNoneMatch.FirstOrDefault();
                break;

            case ETagMatch.IfMatch:
                etagHeader = actionContext.Request.Headers.IfMatch.FirstOrDefault();
                break;
        }

        ETag etag = null;
        if (etagHeader != null)
        {
            etag = new ETag { Tag = etagHeader.Tag };
        }
        actionContext.ActionArguments[Descriptor.ParameterName] = etag;

        var tsc = new TaskCompletionSource<object>();
        tsc.SetResult(null);
        return tsc.Task;
    }
}
```

The **ExecuteBindingAsync** method does the binding. Within this method, add the bound parameter value to the **ActionArgument** dictionary in the **HttpActionContext**.

> **NOTE**
>
> If your **ExecuteBindingAsync** method reads the body of the request message, override the **WillReadBody** property to return true. The request body might be an unbuffered stream that can only be read once, so Web API enforces a rule that at most one binding can read the message body.

To apply a custom **HttpParameterBinding**, you can define an attribute that derives from **ParameterBindingAttribute**. For `ETagParameterBinding`, we'll define two attributes, one for `if-match` headers and one for `if-none-match` headers. Both derive from an abstract base class.

```
public abstract class ETagMatchAttribute : ParameterBindingAttribute
{
    private ETagMatch _match;

    public ETagMatchAttribute(ETagMatch match)
    {
        _match = match;
    }

    public override HttpParameterBinding GetBinding(HttpParameterDescriptor parameter)
    {
        if (parameter.ParameterType == typeof(ETag))
        {
            return new ETagParameterBinding(parameter, _match);
        }
        return parameter.BindAsError("Wrong parameter type");
    }
}

public class IfMatchAttribute : ETagMatchAttribute
{
    public IfMatchAttribute()
        : base(ETagMatch.IfMatch)
    {
    }
}

public class IfNoneMatchAttribute : ETagMatchAttribute
{
    public IfNoneMatchAttribute()
        : base(ETagMatch.IfNoneMatch)
    {
    }
}
```

Here is a controller method that uses the `[IfNoneMatch]` attribute.

```
public HttpResponseMessage Get([IfNoneMatch] ETag etag) { ... }
```

Besides **ParameterBindingAttribute**, there is another hook for adding a custom **HttpParameterBinding**. On the **HttpConfiguration** object, the **ParameterBindingRules** property is a collection of anonymous functions of type (**HttpParameterDescriptor** -> **HttpParameterBinding**). For example, you could add a rule that any ETag parameter on a GET method uses `ETagParameterBinding` with `if-none-match`:

```
config.ParameterBindingRules.Add(p =>
{
    if (p.ParameterType == typeof(ETag) &&
        p.ActionDescriptor.SupportedHttpMethods.Contains(HttpMethod.Get))
    {
        return new ETagParameterBinding(p, ETagMatch.IfNoneMatch);
    }
    else
    {
        return null;
    }
});
```

The function should return `null` for parameters where the binding is not applicable.

IActionValueBinder

The entire parameter-binding process is controlled by a pluggable service, **IActionValueBinder**. The default implementation of **IActionValueBinder** does the following:

1. Look for a **ParameterBindingAttribute** on the parameter. This includes **[FromBody]**, **[FromUri]**, and **[ModelBinder]**, or custom attributes.
2. Otherwise, look in **HttpConfiguration.ParameterBindingRules** for a function that returns a non-null **HttpParameterBinding**.
3. Otherwise, use the default rules that I described previously.
   - If the parameter type is "simple"or has a type converter, bind from the URI. This is equivalent to putting the **[FromUri]** attribute on the parameter.
   - Otherwise, try to read the parameter from the message body. This is equivalent to putting **[FromBody]** on the parameter.

If you wanted, you could replace the entire **IActionValueBinder** service with a custom implementation.

Additional Resources

Custom Parameter Binding Sample

Mike Stall wrote a good series of blog posts about Web API parameter binding:

- How Web API does Parameter Binding
- MVC Style parameter binding for Web API
- How to bind to custom objects in action signatures in MVC/Web API
- How to create a custom value provider in Web API
- Web API Parameter binding under the hood

Error Handling in ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

How to handle exceptions in ASP.NET Web API.

- Exception Handling in Web API
- Global Error Handling in Web API 2

Exception Handling in ASP.NET Web API

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson

This article describes error and exception handling in ASP.NET Web API.

- HttpResponseException
- Exception Filters
- Registering Exception Filters
- HttpError

HttpResponseException

What happens if a Web API controller throws an uncaught exception? By default, most exceptions are translated into an HTTP response with status code 500, Internal Server Error.

The **HttpResponseException** type is a special case. This exception returns any HTTP status code that you specify in the exception constructor. For example, the following method returns 404, Not Found, if the *id* parameter is not valid.

```
public Product GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        throw new HttpResponseException(HttpStatusCode.NotFound);
    }
    return item;
}
```

For more control over the response, you can also construct the entire response message and include it with the **HttpResponseException**:

```
public Product GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        var resp = new HttpResponseMessage(HttpStatusCode.NotFound)
        {
            Content = new StringContent(string.Format("No product with ID = {0}", id)),
            ReasonPhrase = "Product ID Not Found"
        };
        throw new HttpResponseException(resp);
    }
    return item;
}
```

Exception Filters

You can customize how Web API handles exceptions by writing an *exception filter*. An exception filter is executed when a controller method throws any unhandled exception that is *not* an **HttpResponseException** exception.

The **HttpResponseException** type is a special case, because it is designed specifically for returning an HTTP response.

Exception filters implement the **System.Web.Http.Filters.IExceptionFilter** interface. The simplest way to write an exception filter is to derive from the **System.Web.Http.Filters.ExceptionFilterAttribute** class and override the **OnException** method.

> **NOTE**
> Exception filters in ASP.NET Web API are similar to those in ASP.NET MVC. However, they are declared in a separate namespace and function separately. In particular, the **HandleErrorAttribute** class used in MVC does not handle exceptions thrown by Web API controllers.

Here is a filter that converts **NotImplementedException** exceptions into HTTP status code 501, Not Implemented:

```
namespace ProductStore.Filters
{
    using System;
    using System.Net;
    using System.Net.Http;
    using System.Web.Http.Filters;

    public class NotImplExceptionFilterAttribute : ExceptionFilterAttribute
    {
        public override void OnException(HttpActionExecutedContext context)
        {
            if (context.Exception is NotImplementedException)
            {
                context.Response = new HttpResponseMessage(HttpStatusCode.NotImplemented);
            }
        }
    }
}
```

The **Response** property of the **HttpActionExecutedContext** object contains the HTTP response message that will be sent to the client.

Registering Exception Filters

There are several ways to register a Web API exception filter:

- By action
- By controller
- Globally

To apply the filter to a specific action, add the filter as an attribute to the action:

```
public class ProductsController : ApiController
{
    [NotImplExceptionFilter]
    public Contact GetContact(int id)
    {
        throw new NotImplementedException("This method is not implemented");
    }
}
```

To apply the filter to all of the actions on a controller, add the filter as an attribute to the controller class:

```
[NotImplExceptionFilter]
public class ProductsController : ApiController
{
    // ...
}
```

To apply the filter globally to all Web API controllers, add an instance of the filter to the **GlobalConfiguration.Configuration.Filters** collection. Exception filters in this collection apply to any Web API controller action.

```
GlobalConfiguration.Configuration.Filters.Add(
    new ProductStore.NotImplExceptionFilterAttribute());
```

If you use the "ASP.NET MVC 4 Web Application" project template to create your project, put your Web API configuration code inside the `WebApiConfig` class, which is located in the App_Start folder:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Filters.Add(new ProductStore.NotImplExceptionFilterAttribute());

        // Other configuration code...
    }
}
```

HttpError

The **HttpError** object provides a consistent way to return error information in the response body. The following example shows how to return HTTP status code 404 (Not Found) with an **HttpError** in the response body.

```
public HttpResponseMessage GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        var message = string.Format("Product with id = {0} not found", id);
        return Request.CreateErrorResponse(HttpStatusCode.NotFound, message);
    }
    else
    {
        return Request.CreateResponse(HttpStatusCode.OK, item);
    }
}
```

CreateErrorResponse is an extension method defined in the **System.Net.Http.HttpRequestMessageExtensions** class. Internally, **CreateErrorResponse** creates an **HttpError** instance and then creates an **HttpResponseMessage** that contains the **HttpError**.

In this example, if the method is successful, it returns the product in the HTTP response. But if the requested product is not found, the HTTP response contains an **HttpError** in the request body. The response might look like the following:

```
HTTP/1.1 404 Not Found
Content-Type: application/json; charset=utf-8
Date: Thu, 09 Aug 2012 23:27:18 GMT
Content-Length: 51

{
  "Message": "Product with id = 12 not found"
}
```

Notice that the **HttpError** was serialized to JSON in this example. One advantage of using **HttpError** is that it goes through the same content-negotiation and serialization process as any other strongly-typed model.

HttpError and Model Validation

For model validation, you can pass the model state to **CreateErrorResponse**, to include the validation errors in the response:

```
public HttpResponseMessage PostProduct(Product item)
{
    if (!ModelState.IsValid)
    {
        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);
    }

    // Implementation not shown...
}
```

This example might return the following response:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Content-Length: 320

{
  "Message": "The request is invalid.",
  "ModelState": {
    "item": [
      "Required property 'Name' not found in JSON. Path '', line 1, position 14."
    ],
    "item.Name": [
      "The Name field is required."
    ],
    "item.Price": [
      "The field Price must be between 0 and 999."
    ]
  }
}
```

For more information about model validation, see Model Validation in ASP.NET Web API.

Using HttpError with HttpResponseException

The previous examples return an **HttpResponseMessage** message from the controller action, but you can also use **HttpResponseException** to return an **HttpError**. This lets you return a strongly-typed model in the normal success case, while still returning **HttpError** if there is an error:

```
public Product GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        var message = string.Format("Product with id = {0} not found", id);
        throw new HttpResponseException(
            Request.CreateErrorResponse(HttpStatusCode.NotFound, message));
    }
    else
    {
        return item;
    }
}
```

Global Error Handling in ASP.NET Web API 2

4/6/2020 • 9 minutes to read • Edit Online

by David Matson, Rick Anderson

This topic provides an overview of global error handling in ASP.NET Web API 2 for ASP.NET 4.x. Today there's no easy way in Web API to log or handle errors globally. Some unhandled exceptions can be processed via exception filters, but there are a number of cases that exception filters can't handle. For example:

1. Exceptions thrown from controller constructors.
2. Exceptions thrown from message handlers.
3. Exceptions thrown during routing.
4. Exceptions thrown during response content serialization.

We want to provide a simple, consistent way to log and handle (where possible) these exceptions.

There are two major cases for handling exceptions, the case where we are able to send an error response and the case where all we can do is log the exception. An example for the latter case is when an exception is thrown in the middle of streaming response content; in that case it is too late to send a new response message since the status code, headers, and partial content have already gone across the wire, so we simply abort the connection. Even though the exception can't be handled to produce a new response message, we still support logging the exception. In cases where we can detect an error, we can return an appropriate error response as shown in the following:

```
public IHttpActionResult GetProduct(int id)
{
    var product = products.FirstOrDefault((p) => p.Id == id);
    if (product == null)
    {
        return NotFound();
    }
    return Ok(product);
}
```

Existing Options

In addition to exception filters, message handlers can be used today to observe all 500-level responses, but acting on those responses is difficult, as they lack context about the original error. Message handlers also have some of the same limitations as exception filters regarding the cases they can handle. While Web API does have tracing infrastructure that captures error conditions the tracing infrastructure is for diagnostics purposes and is not designed or suited for running in production environments. Global exception handling and logging should be services that can run during production and be plugged into existing monitoring solutions (for example, ELMAH).

Solution Overview

We provide two new user-replaceable services, IExceptionLogger and IExceptionHandler, to log and handle unhandled exceptions. The services are very similar, with two main differences:

1. We support registering multiple exception loggers but only a single exception handler.
2. Exception loggers always get called, even if we're about to abort the connection. Exception handlers only get called when we're still able to choose which response message to send.

Both services provide access to an exception context containing relevant information from the point where the exception was detected, particularly the HttpRequestMessage, the HttpRequestContext, the thrown exception and the exception source (details below).

Design Principles

1. **No breaking changes** Because this functionality is being added in a minor release, one important constraint impacting the solution is that there be no breaking changes, either to type contracts or to behavior. This constraint ruled out some cleanup we would like to have done in terms of existing catch blocks turning exceptions into 500 responses. This additional cleanup is something we might consider for a subsequent major release. If this is important to you please vote on it at ASP.NET Web API user voice.
2. **Maintaining consistency with Web API constructs** Web API's filter pipeline is a great way to handle cross-cutting concerns with the flexibility of applying the logic at an action-specific, controller-specific or global scope. Filters, including exception filters, always have action and controller contexts, even when registered at the global scope. That contract makes sense for filters, but it means that exception filters, even globally scoped ones, aren't a good fit for some exception handling cases, such as exceptions from message handlers, where no action or controller context exists. If we want to use the flexible scoping afforded by filters for exception handling, we still need exception filters. But if we need to handle exception outside of a controller context, we also need a separate construct for full global error handling (something without the controller context and action context constraints).

When to Use

- Exception loggers are the solution to seeing all unhandled exception caught by Web API.
- Exception handlers are the solution for customizing all possible responses to unhandled exceptions caught by Web API.
- Exception filters are the easiest solution for processing the subset unhandled exceptions related to a specific action or controller.

Service Details

The exception logger and handler service interfaces are simple async methods taking the respective contexts:

```
public interface IExceptionLogger
{
   Task LogAsync(ExceptionLoggerContext context,
                 CancellationToken cancellationToken);
}

public interface IExceptionHandler
{
   Task HandleAsync(ExceptionHandlerContext context,
                    CancellationToken cancellationToken);
}
```

We also provide base classes for both of these interfaces. Overriding the core (sync or async) methods is all that is required to log or handle at the recommended times. For logging, the `ExceptionLogger` base class will ensure that the core logging method is only called once for each exception (even if it later propagates further up the call stack and is caught again). The `ExceptionHandler` base class will call the core handling method only for exceptions at the top of the call stack, ignoring legacy nested catch blocks. (Simplified versions of these base classes are in the appendix below.) Both `IExceptionLogger` and `IExceptionHandler` receive information about the exception via an `ExceptionContext`.

```
public class ExceptionContext
{
   public Exception Exception { get; set; }

   public HttpRequestMessage Request { get; set; }

   public HttpRequestContext RequestContext { get; set; }

   public HttpControllerContext ControllerContext { get; set; }

   public HttpActionContext ActionContext { get; set; }

   public HttpResponseMessage Response { get; set; }

   public string CatchBlock { get; set; }

   public bool IsTopLevelCatchBlock { get; set; }
}
```

When the framework calls an exception logger or an exception handler, it will always provide an `Exception` and a `Request`. Except for unit testing, it will also always provide a `RequestContext`. It will rarely provide a `ControllerContext` and `ActionContext` (only when calling from the catch block for exception filters). It will very rarely provide a `Response` (only in certain IIS cases when in the middle of trying to write the response). Note that because some of these properties may be `null` it is up to the consumer to check for `null` before accessing members of the exception class. `CatchBlock` is a string indicating which catch block saw the exception. The catch block strings are as follows:

- HttpServer (SendAsync method)
- HttpControllerDispatcher (SendAsync method)
- HttpBatchHandler (SendAsync method)
- IExceptionFilter (ApiController's processing of the exception filter pipeline in ExecuteAsync)
- OWIN host:
  - HttpMessageHandlerAdapter.BufferResponseContentAsync (for buffering output)
  - HttpMessageHandlerAdapter.CopyResponseContentAsync (for streaming output)
- Web host:
  - HttpControllerHandler.WriteBufferedResponseContentAsync (for buffering output)
  - HttpControllerHandler.WriteStreamedResponseContentAsync (for streaming output)
  - HttpControllerHandler.WriteErrorResponseContentAsync (for failures in error recovery under buffered output mode)

The list of catch block strings is also available via static readonly properties. (The core catch block string are on the static ExceptionCatchBlocks; the remainder appear on one static class each for OWIN and web host). `IsTopLevelCatchBlock` is helpful for following the recommended pattern of handling exceptions only at the top of the call stack. Rather than turning exceptions into 500 responses anywhere a nested catch block occurs, an exception handler can let exceptions propagate until they are about to be seen by the host.

In addition to the `ExceptionContext`, a logger gets one more piece of information via the full `ExceptionLoggerContext`:

```
public class ExceptionLoggerContext
{
   public ExceptionContext ExceptionContext { get; set; }
   public bool CanBeHandled { get; set; }
}
```

The second property, `CanBeHandled`, allows a logger to identify an exception that cannot be handled. When the connection is about to be aborted and no new response message can be sent, the loggers will be called but the handler will *not* be called, and the loggers can identify this scenario from this property.

In additional to the `ExceptionContext`, a handler gets one more property it can set on the full `ExceptionHandlerContext` to handle the exception:

```
public class ExceptionHandlerContext
{
   public ExceptionContext ExceptionContext { get; set; }
   public IHttpActionResult Result { get; set; }
}
```

An exception handler indicates that it has handled an exception by setting the `Result` property to an action result (for example, an ExceptionResult, InternalServerErrorResult, StatusCodeResult, or a custom result). If the `Result` property is null, the exception is unhandled and the original exception will be re-thrown.

For exceptions at the top of the call stack, we took an extra step to ensure the response is appropriate for API callers. If the exception propagates up to the host, the caller would see the yellow screen of death or some other host provided response which is typically HTML and not usually an appropriate API error response. In these cases, the Result starts out non-null, and only if a custom exception handler explicitly sets it back to `null` (unhandled) will the exception propagate to the host. Setting `Result` to `null` in such cases can be useful for two scenarios:

1. OWIN hosted Web API with custom exception handling middleware registered before/outside Web API.
2. Local debugging via a browser, where the yellow screen of death is actually a helpful response for an unhandled exception.

For both exception loggers and exception handlers, we don't do anything to recover if the logger or handler itself throws an exception. (Other than letting the exception propagate, leave feedback at the bottom of this page if you have a better approach.) The contract for exception loggers and handlers is that they should not let exceptions propagate up to their callers; otherwise, the exception will just propagate, often all the way to the host resulting in an HTML error (like ASP.NET's yellow screen) being sent back to the client (which usually isn't the preferred option for API callers that expect JSON or XML).

Examples

Tracing Exception Logger

The exception logger below sends exception data to configured Trace sources (including the Debug output window in Visual Studio).

```
class TraceExceptionLogger : ExceptionLogger
{
    public override void LogCore(ExceptionLoggerContext context)
    {
        Trace.TraceError(context.ExceptionContext.Exception.ToString());
    }
}
```

Custom Error Message Exception Handler

The exception handler below produces a custom error response to clients, including an email address for contacting support.

```
class OopsExceptionHandler : ExceptionHandler
{
    public override void HandleCore(ExceptionHandlerContext context)
    {
        context.Result = new TextPlainErrorResult
        {
            Request = context.ExceptionContext.Request,
            Content = "Oops! Sorry! Something went wrong." +
                      "Please contact support@contoso.com so we can try to fix it."
        };
    }

    private class TextPlainErrorResult : IHttpActionResult
    {
        public HttpRequestMessage Request { get; set; }

        public string Content { get; set; }

        public Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
        {
            HttpResponseMessage response =
                              new HttpResponseMessage(HttpStatusCode.InternalServerError);
            response.Content = new StringContent(Content);
            response.RequestMessage = Request;
            return Task.FromResult(response);
        }
    }
}
```

Registering Exception Filters

If you use the "ASP.NET MVC 4 Web Application" project template to create your project, put your Web API configuration code inside the `WebApiConfig` class, in the *App_Start* folder:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Filters.Add(new ProductStore.NotImplExceptionFilterAttribute());

        // Other configuration code...
    }
}
```

Appendix: Base Class Details

```
public class ExceptionLogger : IExceptionLogger
{
    public virtual Task LogAsync(ExceptionLoggerContext context,
                                 CancellationToken cancellationToken)
    {
        if (!ShouldLog(context))
        {
            return Task.FromResult(0);
        }

        return LogAsyncCore(context, cancellationToken);
    }
```

```
    public virtual Task LogAsyncCore(ExceptionLoggerContext context,
                                     CancellationToken cancellationToken)
    {
        LogCore(context);
        return Task.FromResult(0);
    }

    public virtual void LogCore(ExceptionLoggerContext context)
    {
    }

    public virtual bool ShouldLog(ExceptionLoggerContext context)
    {
        IDictionary exceptionData = context.ExceptionContext.Exception.Data;

        if (!exceptionData.Contains("MS_LoggedBy"))
        {
            exceptionData.Add("MS_LoggedBy", new List<object>());
        }

        ICollection<object> loggedBy = ((ICollection<object>)exceptionData[LoggedByKey]);

        if (!loggedBy.Contains(this))
        {
            loggedBy.Add(this);
            return true;
        }
        else
        {
            return false;
        }
    }
}

public class ExceptionHandler : IExceptionHandler
{
    public virtual Task HandleAsync(ExceptionHandlerContext context,
                                    CancellationToken cancellationToken)
    {
        if (!ShouldHandle(context))
        {
            return Task.FromResult(0);
        }

        return HandleAsyncCore(context, cancellationToken);
    }

    public virtual Task HandleAsyncCore(ExceptionHandlerContext context,
                                        CancellationToken cancellationToken)
    {
        HandleCore(context);
        return Task.FromResult(0);
    }

    public virtual void HandleCore(ExceptionHandlerContext context)
    {
    }

    public virtual bool ShouldHandle(ExceptionHandlerContext context)
    {
        return context.ExceptionContext.IsOutermostCatchBlock;
    }
}
```

Testing and Debugging ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

Testing and debugging ASP.NET Web API

- Tracing in Web API 2
- Unit Testing Web API 2
- Unit Testing Controllers in ASP.NET Web API 2
- Mocking Entity Framework when Unit Testing Web API 2
- Troubleshooting HTTP 405 Errors after Publishing Web API 2 Applications

Tracing in ASP.NET Web API 2

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson

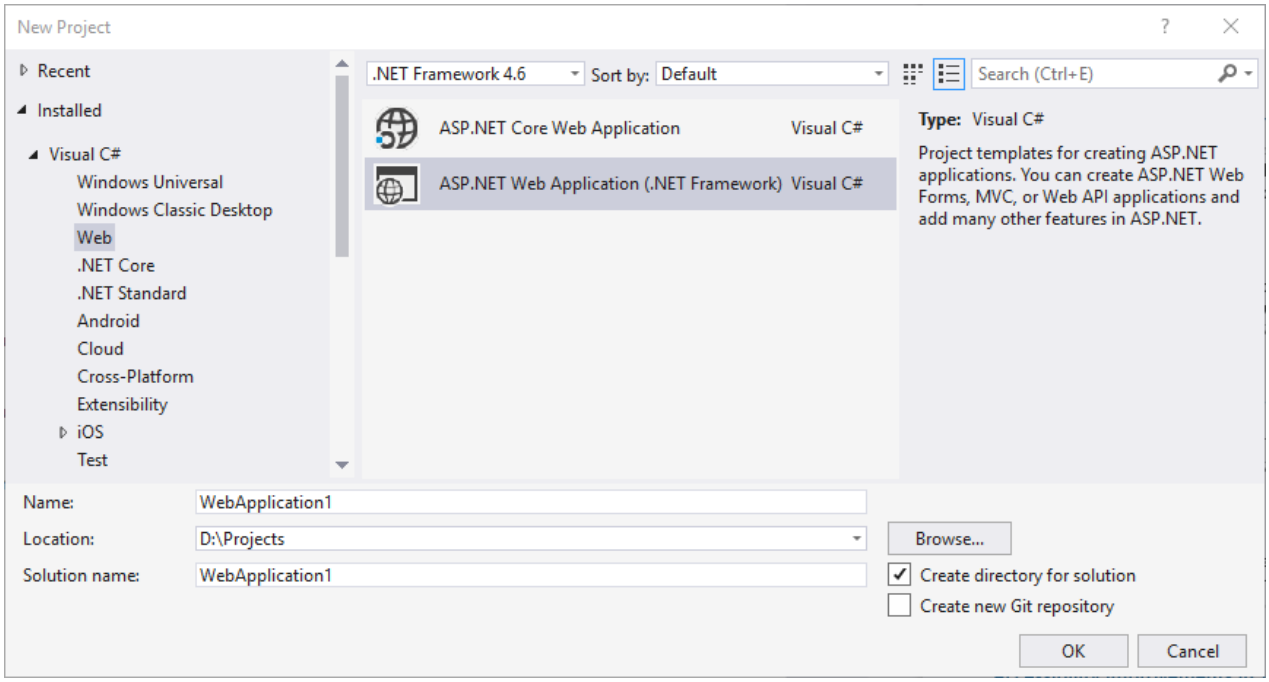
When you are trying to debug a web-based application, there is no substitute for a good set of trace logs. This tutorial shows how to enable tracing in ASP.NET Web API. You can use this feature to trace what the Web API framework does before and after it invokes your controller. You can also use it to trace your own code.

Software versions used in the tutorial

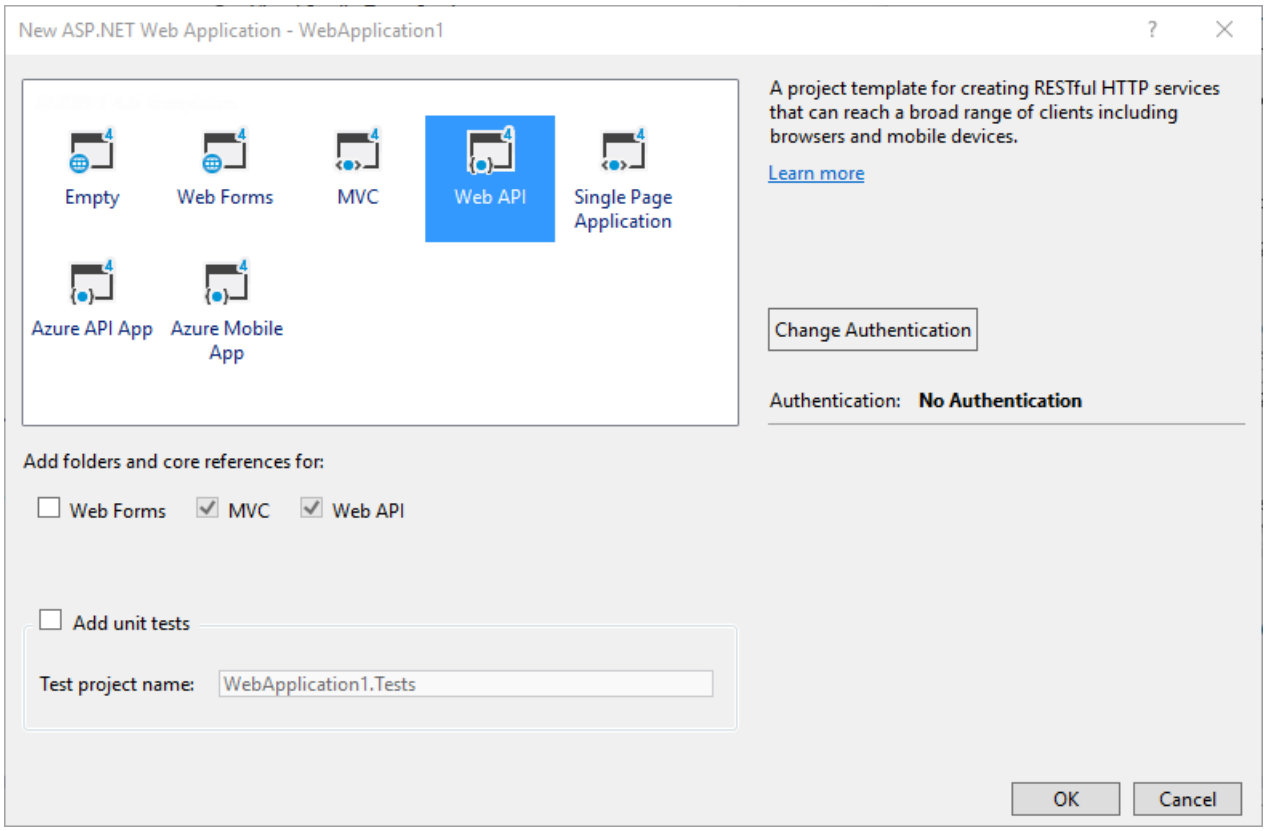
- Visual Studio 2017 (also works with Visual Studio 2015)
- Web API 2
- Microsoft.AspNet.WebApi.Tracing

Enable System.Diagnostics Tracing in Web API

First, we'll create a new ASP.NET Web Application project. In Visual Studio, from the **File** menu, select **New** > **Project**. Under **Templates**, **Web**, select **ASP.NET Web Application**.



Choose the Web API project template.



From the **Tools** menu, select **NuGet Package Manager**, then **Package Manage Console**.

In the Package Manager Console window, type the following commands.

```
Install-Package Microsoft.AspNet.WebApi.Tracing
Update-Package Microsoft.AspNet.WebApi.WebHost
```

The first command installs the latest Web API tracing package. It also updates the core Web API packages. The second command updates the WebApi.WebHost package to the latest version.

> **NOTE**
> If you want to target a specific version of Web API, use the -Version flag when you install the tracing package.

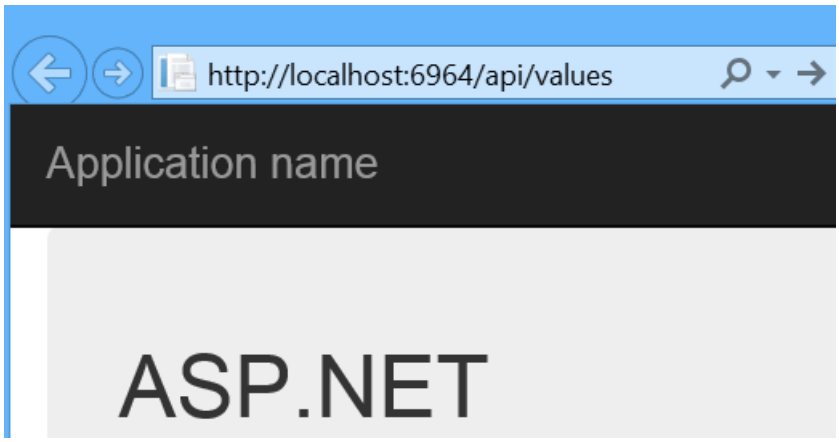
Open the file WebApiConfig.cs in the App_Start folder. Add the following code to the **Register** method.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // New code
        config.EnableSystemDiagnosticsTracing();

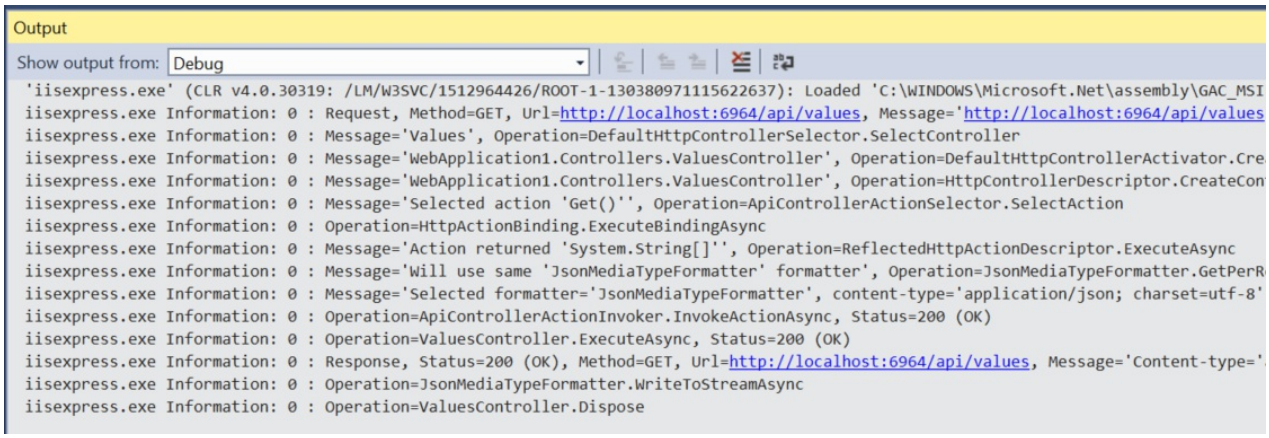
        // Other configuration code not shown.
    }
}
```

This code adds the SystemDiagnosticsTraceWriter class to the Web API pipeline. The **SystemDiagnosticsTraceWriter** class writes traces to System.Diagnostics.Trace.

To see the traces, run the application in the debugger. In the browser, navigate to `/api/values`.



The trace statements are written to the Output window in Visual Studio. (From the **View** menu, select **Output**).



Because **SystemDiagnosticsTraceWriter** writes traces to **System.Diagnostics.Trace**, you can register additional trace listeners; for example, to write traces to a log file. For more information about trace writers, see the Trace Listeners topic on MSDN.

Configuring SystemDiagnosticsTraceWriter

The following code shows how to configure the trace writer.

```
SystemDiagnosticsTraceWriter traceWriter = config.EnableSystemDiagnosticsTracing();
traceWriter.IsVerbose = true;
traceWriter.MinimumLevel = TraceLevel.Debug;
```

There are two settings that you can control:

- IsVerbose: If false, each trace contains minimal information. If true, traces include more information.
- MinimumLevel: Sets the minimum trace level. Trace levels, in order, are Debug, Info, Warn, Error, and Fatal.

Adding Traces to Your Web API Application

Adding a trace writer gives you immediate access to the traces created by the Web API pipeline. You can also use the trace writer to trace your own code:

```
using System.Web.Http.Tracing;

public class ProductsController : ApiController
{
    public HttpResponseMessage GetAllProducts()
    {
        Configuration.Services.GetTraceWriter().Info(
            Request, "ProductsController", "Get the list of products.");

        // ...
    }
}
```

To get the trace writer, call **HttpConfiguration.Services.GetTraceWriter**. From a controller, this method is accessible through the **ApiController.Configuration** property.

To write a trace, you can call the **ITraceWriter.Trace** method directly, but the ITraceWriterExtensions class defines some extension methods that are more friendly. For example, the **Info** method shown above creates a trace with trace level **Info**.

Web API Tracing Infrastructure

This section describes how to write a custom trace writer for Web API.

The Microsoft.AspNet.WebApi.Tracing package is built on top of a more general tracing infrastructure in Web API. Instead of using Microsoft.AspNet.WebApi.Tracing, you can also plug in some other tracing/logging library, such as NLog or log4net.

To collect traces, implement the **ITraceWriter** interface. Here is a simple example:

```
public class SimpleTracer : ITraceWriter
{
    public void Trace(HttpRequestMessage request, string category, TraceLevel level,
        Action<TraceRecord> traceAction)
    {
        TraceRecord rec = new TraceRecord(request, category, level);
        traceAction(rec);
        WriteTrace(rec);
    }

    protected void WriteTrace(TraceRecord rec)
    {
        var message = string.Format("{0};{1};{2}",
            rec.Operator, rec.Operation, rec.Message);
        System.Diagnostics.Trace.WriteLine(message, rec.Category);
    }
}
```

The **ITraceWriter.Trace** method creates a trace. The caller specifies a category and trace level. The category can be any user-defined string. Your implementation of **Trace** should do the following:

1. Create a new **TraceRecord**. Initialize it with the request, category, and trace level, as shown. These values are provided by the caller.
2. Invoke the *traceAction* delegate. Inside this delegate, the caller is expected to fill in the rest of the **TraceRecord**.
3. Write the **TraceRecord**, using any logging technique that you like. The example shown here simply calls into **System.Diagnostics.Trace**.

Setting the Trace Writer

To enable tracing, you must configure Web API to use your **ITraceWriter** implementation. You do this through the **HttpConfiguration** object, as shown in the following code:

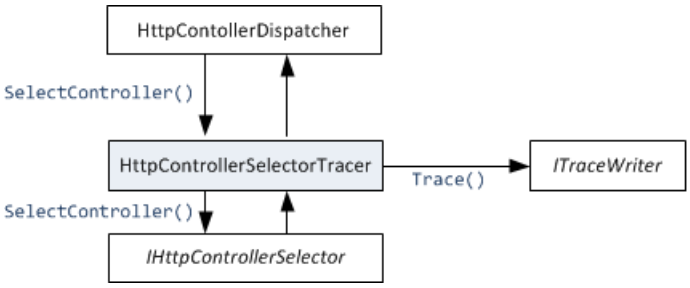
```
public static void Register(HttpConfiguration config)
{
    config.Services.Replace(typeof(ITraceWriter), new SimpleTracer());
}
```

Only one trace writer can be active. By default, Web API sets a "no-op" tracer that does nothing. (The "no-op" tracer exists so that tracing code does not have to check whether the trace writer is **null** before writing a trace.)

How Web API Tracing Works

Tracing in Web API uses a *facade* pattern: When tracing is enabled, Web API wraps various parts of the request pipeline with classes that perform trace calls.

For example, when selecting a controller, the pipeline uses the **IHttpControllerSelector** interface. With tracing enabled, the pipeline inserts a class that implements **IHttpControllerSelector** but calls through to the real implementation:



The benefits of this design include:

- If you do not add a trace writer, the tracing components are not instantiated and have no performance impact.
- If you replace default services such as **IHttpControllerSelector** with your own custom implementation, tracing is not affected, because tracing is done by the wrapper object.

You can also replace the entire Web API trace framework with your own custom framework, by replacing the default **ITraceManager** service:

```
config.Services.Replace(typeof(ITraceManager), new MyTraceManager());
```

Implement **ITraceManager.Initialize** to initialize your tracing system. Be aware that this replaces the *entire* trace framework, including all of the tracing code that is built into Web API.

Unit Testing ASP.NET Web API 2

2/20/2020 • 6 minutes to read • Edit Online

by Tom FitzMacken

Download Completed Project

This guidance and application demonstrate how to create simple unit tests for your Web API 2 application. This tutorial shows how to include a unit test project in your solution, and write test methods that check the returned values from a controller method.

This tutorial assumes you are familiar with the basic concepts of ASP.NET Web API. For an introductory tutorial, see Getting Started with ASP.NET Web API 2.

The unit tests in this topic are intentionally limited to simple data scenarios. For unit testing more advanced data scenarios, see Mocking Entity Framework when Unit Testing ASP.NET Web API 2.

Software versions used in the tutorial

- Visual Studio 2017
- Web API 2

In this topic

This topic contains the following sections:

- Prerequisites
- Download code
- Create application with unit test project
  - Add unit test project when creating the application
  - Add unit test project to an existing application
- Set up the Web API 2 application
- Install NuGet packages in test project
- Create tests
- Run tests

Prerequisites

Visual Studio 2017 Community, Professional or Enterprise edition

Download code

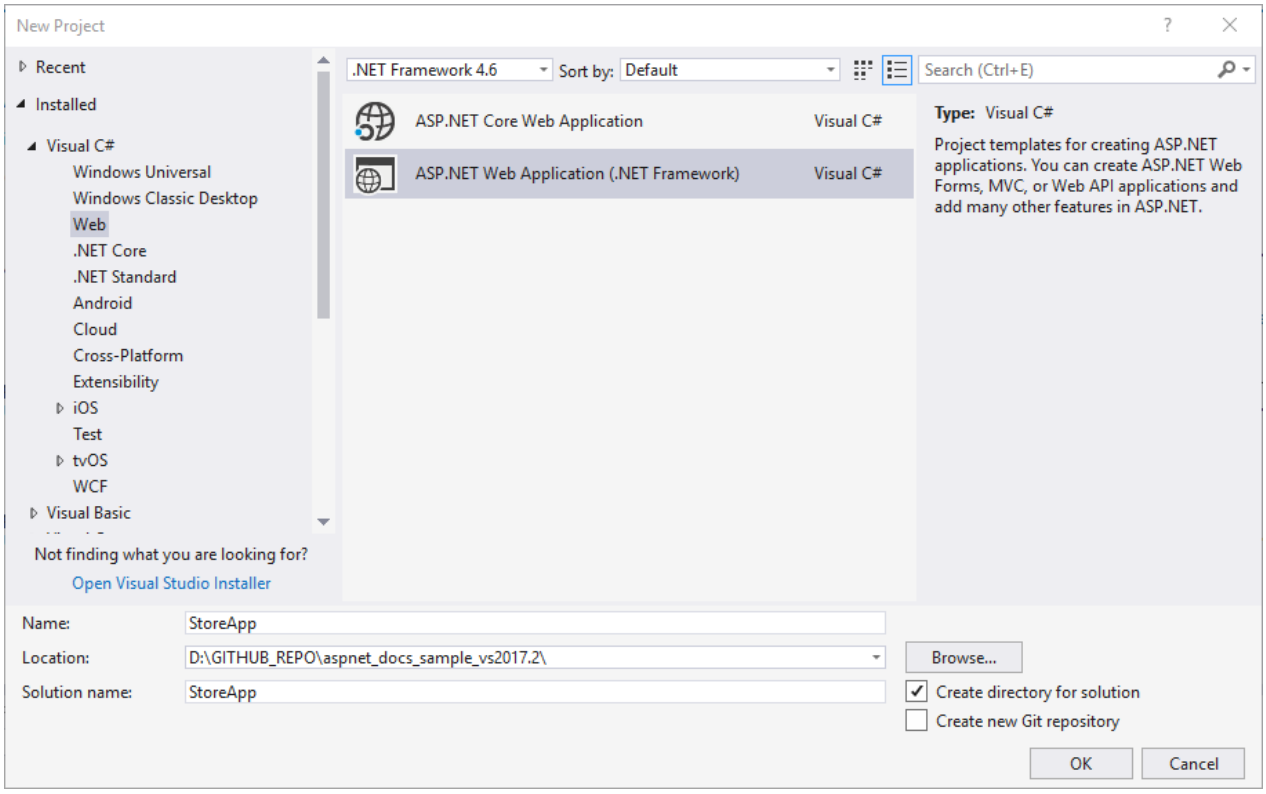
Download the completed project. The downloadable project includes unit test code for this topic and for the Mocking Entity Framework when Unit Testing ASP.NET Web API topic.

Create application with unit test project

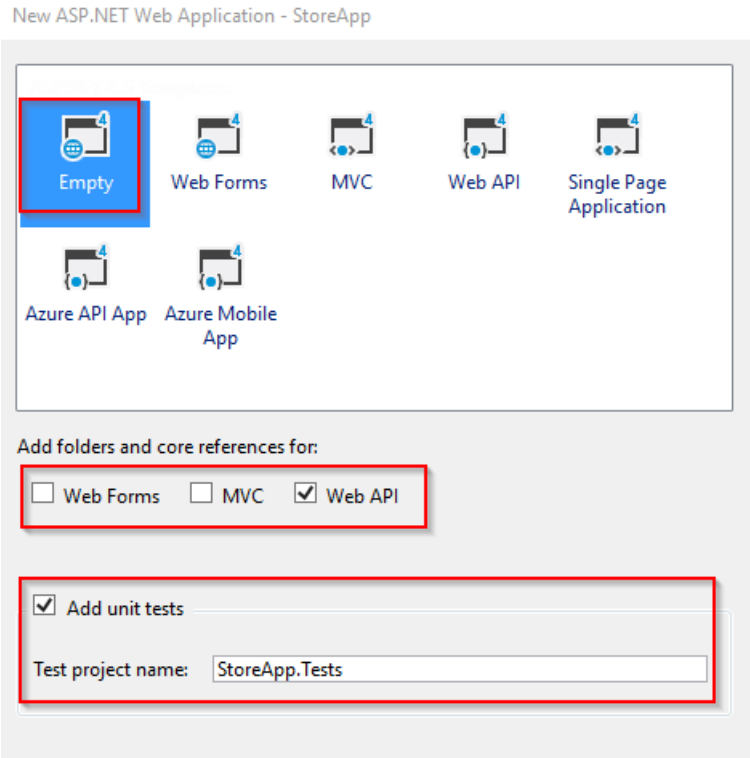
You can either create a unit test project when creating your application or add a unit test project to an existing application. This tutorial shows both methods for creating a unit test project. To follow this tutorial, you can use either approach.

Add unit test project when creating the application

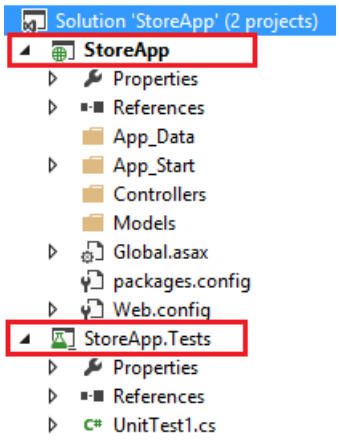
Create a new ASP.NET Web Application named **StoreApp**.



In the New ASP.NET Project windows, select the **Empty** template and add folders and core references for Web API. Select the **Add unit tests** option. The unit test project is automatically named **StoreApp.Tests**. You can keep this name.

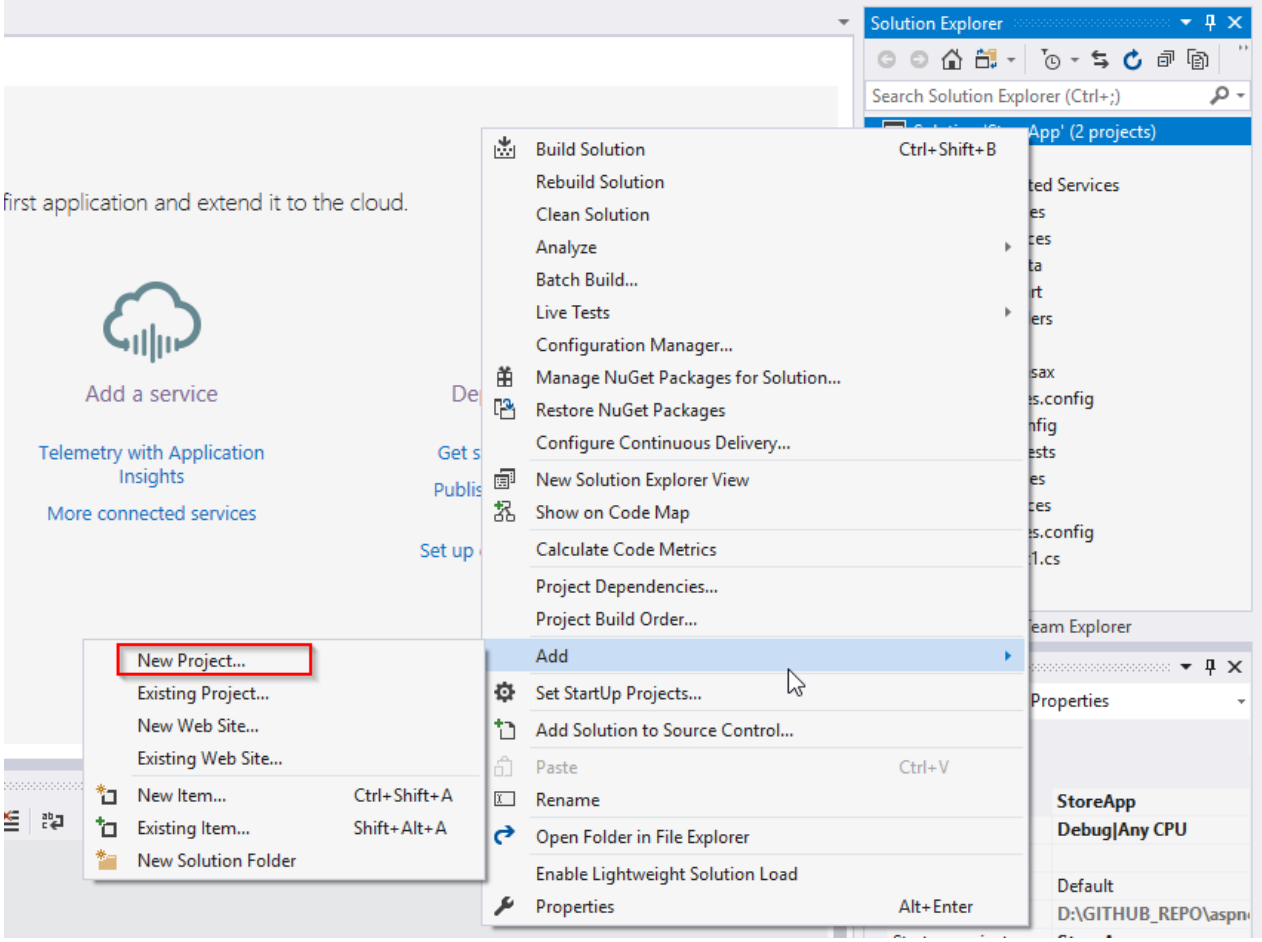


After creating the application, you will see it contains two projects.

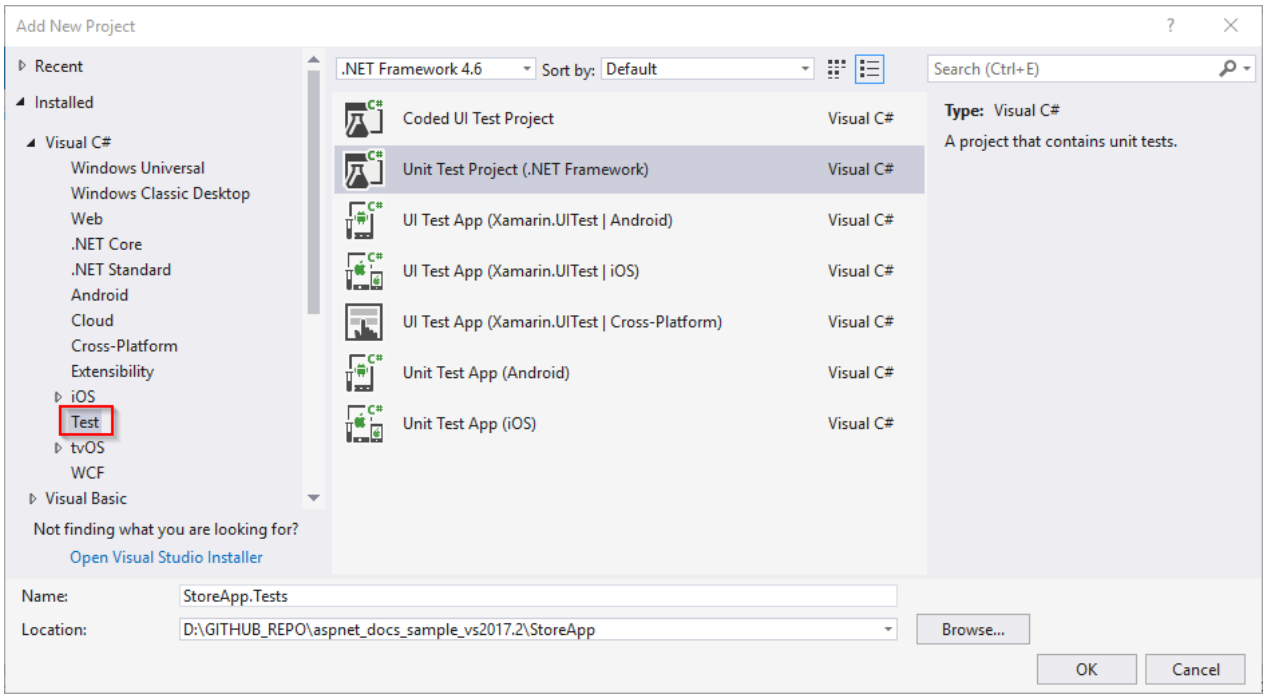


Add unit test project to an existing application

If you did not create the unit test project when you created your application, you can add one at any time. For example, suppose you already have an application named StoreApp, and you want to add unit tests. To add a unit test project, right-click your solution and select **Add** and **New Project**.



Select **Test** in the left pane, and select **Unit Test Project** for the project type. Name the project **StoreApp.Tests**.



You will see the unit test project in your solution.

In the unit test project, add a project reference to the original project.

Set up the Web API 2 application

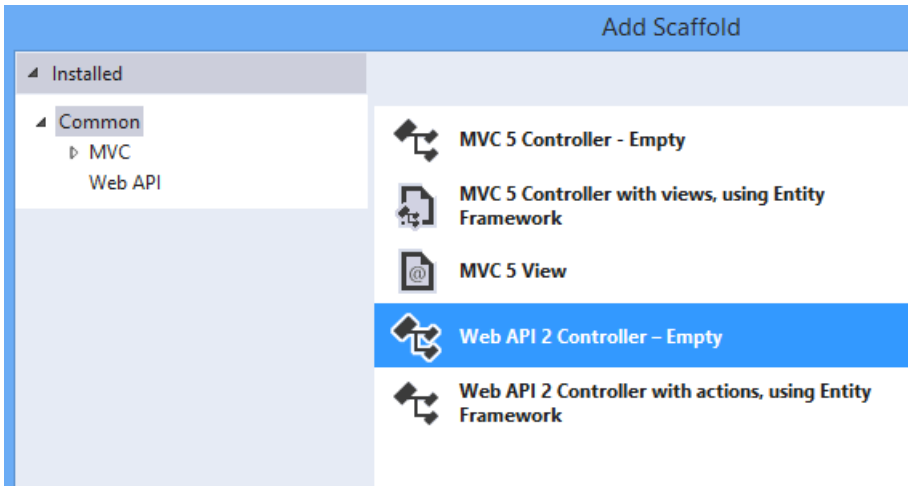
In your StoreApp project, add a class file to the **Models** folder named **Product.cs**. Replace the contents of the file with the following code.

```
using System;

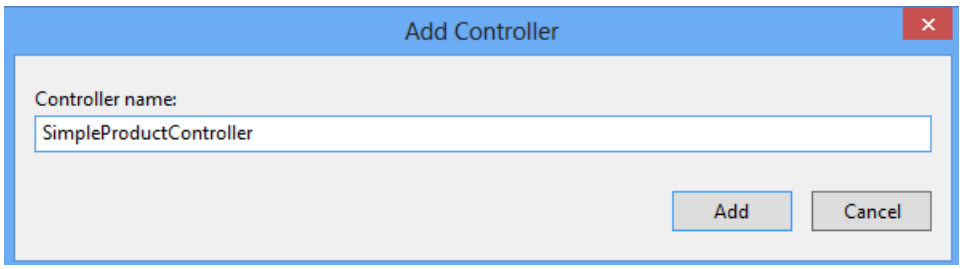
namespace StoreApp.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

Build the solution.

Right-click the Controllers folder and select **Add** and **New Scaffolded Item**. Select **Web API 2 Controller - Empty**.



Set the controller name to **SimpleProductController**, and click **Add**.



Replace the existing code with the following code. To simplify this example, the data is stored in a list rather than a database. The list defined in this class represents the production data. Notice that the controller includes a constructor that takes as a parameter a list of Product objects. This constructor enables you to pass test data when unit testing. The controller also includes two **async** methods to illustrate unit testing asynchronous methods. These async methods were implemented by calling **Task.FromResult** to minimize extraneous code, but normally the methods would include resource-intensive operations.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Web.Http;
using StoreApp.Models;

namespace StoreApp.Controllers
{
    public class SimpleProductController : ApiController
    {
        List<Product> products = new List<Product>();

        public SimpleProductController() { }

        public SimpleProductController(List<Product> products)
        {
            this.products = products;
        }

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public async Task<IEnumerable<Product>> GetAllProductsAsync()
        {
            return await Task.FromResult(GetAllProducts());
        }

        public IHttpActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product);
        }

        public async Task<IHttpActionResult> GetProductAsync(int id)
        {
            return await Task.FromResult(GetProduct(id));
        }
    }
}
```

The GetProduct method returns an instance of the **IHttpActionResult** interface. IHttpActionResult is one of the new features in Web API 2, and it simplifies unit test development. Classes that implement the IHttpActionResult interface are found in the System.Web.Http.Results namespace. These classes represent possible responses from an action request, and they correspond to HTTP status codes.

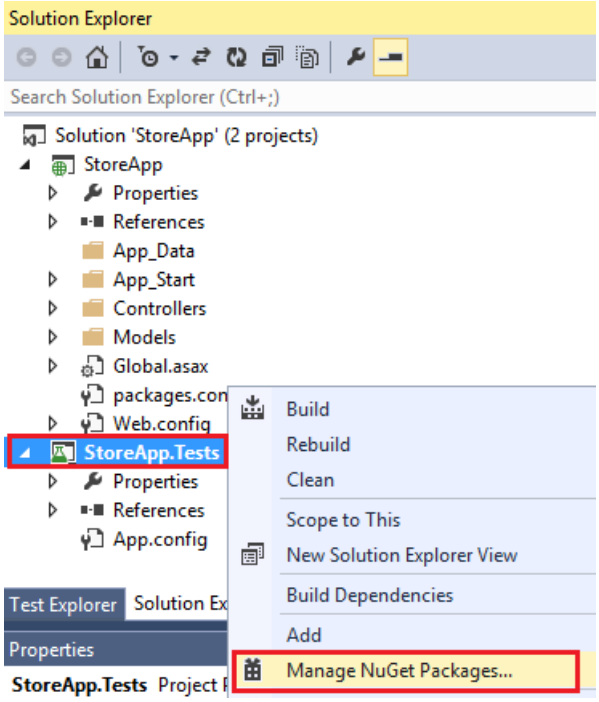
Build the solution.

You are now ready to set up the test project.

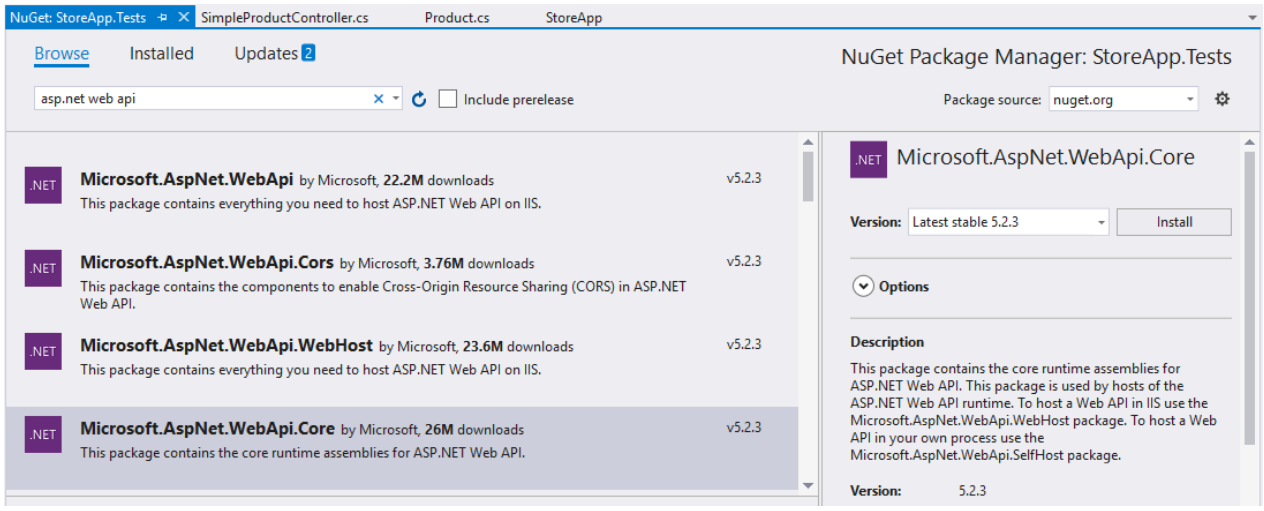
Install NuGet packages in test project

When you use the Empty template to create an application, the unit test project (StoreApp.Tests) does not include any installed NuGet packages. Other templates, such as the Web API template, include some NuGet packages in the unit test project. For this tutorial, you must include the Microsoft ASP.NET Web API 2 Core package to the test project.

Right-click the StoreApp.Tests project and select **Manage NuGet Packages**. You must select the StoreApp.Tests project to add the packages to that project.



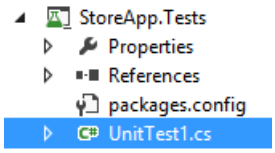
Find and install Microsoft ASP.NET Web API 2 Core package.



Close the Manage NuGet Packages window.

Create tests

By default, your test project includes an empty test file named UnitTest1.cs. This file shows the attributes you use to create test methods. For your unit tests, you can either use this file or create your own file.



For this tutorial, you will create your own test class. You can delete the UnitTest1.cs file. Add a class named **TestSimpleProductController.cs**, and replace the code with the following code.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
```

```
using System.Web.Http.Results;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using StoreApp.Controllers;
using StoreApp.Models;

namespace StoreApp.Tests
{
    [TestClass]
    public class TestSimpleProductController
    {
        [TestMethod]
        public void GetAllProducts_ShouldReturnAllProducts()
        {
            var testProducts = GetTestProducts();
            var controller = new SimpleProductController(testProducts);

            var result = controller.GetAllProducts() as List<Product>;
            Assert.AreEqual(testProducts.Count, result.Count);
        }

        [TestMethod]
        public async Task GetAllProductsAsync_ShouldReturnAllProducts()
        {
            var testProducts = GetTestProducts();
            var controller = new SimpleProductController(testProducts);

            var result = await controller.GetAllProductsAsync() as List<Product>;
            Assert.AreEqual(testProducts.Count, result.Count);
        }

        [TestMethod]
        public void GetProduct_ShouldReturnCorrectProduct()
        {
            var testProducts = GetTestProducts();
            var controller = new SimpleProductController(testProducts);

            var result = controller.GetProduct(4) as OkNegotiatedContentResult<Product>;
            Assert.IsNotNull(result);
            Assert.AreEqual(testProducts[3].Name, result.Content.Name);
        }

        [TestMethod]
        public async Task GetProductAsync_ShouldReturnCorrectProduct()
        {
            var testProducts = GetTestProducts();
            var controller = new SimpleProductController(testProducts);

            var result = await controller.GetProductAsync(4) as OkNegotiatedContentResult<Product>;
            Assert.IsNotNull(result);
            Assert.AreEqual(testProducts[3].Name, result.Content.Name);
        }

        [TestMethod]
        public void GetProduct_ShouldNotFindProduct()
        {
            var controller = new SimpleProductController(GetTestProducts());

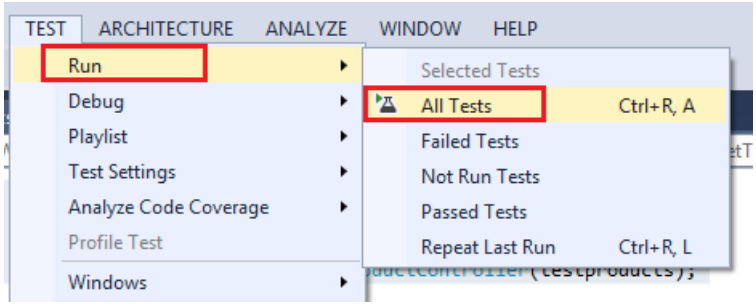
            var result = controller.GetProduct(999);
            Assert.IsInstanceOfType(result, typeof(NotFoundResult));
        }

        private List<Product> GetTestProducts()
        {
            var testProducts = new List<Product>();
            testProducts.Add(new Product { Id = 1, Name = "Demo1", Price = 1 });
            testProducts.Add(new Product { Id = 2, Name = "Demo2", Price = 3.75M });
            testProducts.Add(new Product { Id = 3, Name = "Demo3", Price = 16.99M });
            testProducts.Add(new Product { Id = 4, Name = "Demo4", Price = 11.00M });
```

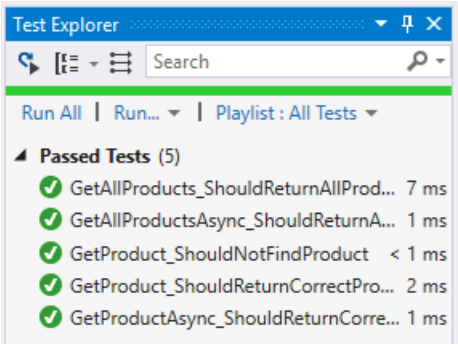
```
            return testProducts;
        }
    }
}
```

Run tests

You are now ready to run the tests. All of the method that are marked with the **TestMethod** attribute will be tested. From the **Test** menu item, run the tests.



Open the **Test Explorer** window, and notice the results of the tests.



Summary

You have completed this tutorial. The data in this tutorial was intentionally simplified to focus on unit testing conditions. For unit testing more advanced data scenarios, see Mocking Entity Framework when Unit Testing ASP.NET Web API 2.

Unit Testing Controllers in ASP.NET Web API 2

9/16/2020 • 6 minutes to read • Edit Online

by Mike Wasson

This topic describes some specific techniques for unit testing controllers in Web API 2. Before reading this topic, you might want to read the tutorial Unit Testing ASP.NET Web API 2, which shows how to add a unit-test project to your solution.

Software versions used in the tutorial

- Visual Studio 2017
- Web API 2
- Moq 4.5.30

> **NOTE**
>
> I used Moq, but the same idea applies to any mocking framework. Moq 4.5.30 (and later) supports Visual Studio 2017, Roslyn and .NET 4.5 and later versions.

A common pattern in unit tests is "arrange-act-assert":

- Arrange: Set up any prerequisites for the test to run.
- Act: Perform the test.
- Assert: Verify that the test succeeded.

In the arrange step, you will often use mock or stub objects. That minimizes the number of dependencies, so the test is focused on testing one thing.

Here are some things that you should unit test in your Web API controllers:

- The action returns the correct type of response.
- Invalid parameters return the correct error response.
- The action calls the correct method on the repository or service layer.
- If the response includes a domain model, verify the model type.

These are some of the general things to test, but the specifics depend on your controller implementation. In particular, it makes a big difference whether your controller actions return **HttpResponseMessage** or **IHttpActionResult**. For more information about these result types, see Action Results in Web Api 2.

Testing Actions that Return HttpResponseMessage

Here is an example of a controller whose actions return **HttpResponseMessage**.

```
public class ProductsController : ApiController
{
    IProductRepository _repository;

    public ProductsController(IProductRepository repository)
    {
        _repository = repository;
    }

    public HttpResponseMessage Get(int id)
    {
        Product product = _repository.GetById(id);
        if (product == null)
        {
            return Request.CreateResponse(HttpStatusCode.NotFound);
        }
        return Request.CreateResponse(product);
    }

    public HttpResponseMessage Post(Product product)
    {
        _repository.Add(product);

        var response = Request.CreateResponse(HttpStatusCode.Created, product);
        string uri = Url.Link("DefaultApi", new { id = product.Id });
        response.Headers.Location = new Uri(uri);

        return response;
    }
}
```

Notice the controller uses dependency injection to inject an `IProductRepository`. That makes the controller more testable, because you can inject a mock repository. The following unit test verifies that the `Get` method writes a `Product` to the response body. Assume that `repository` is a mock `IProductRepository`.

```
[TestMethod]
public void GetReturnsProduct()
{
    // Arrange
    var controller = new ProductsController(repository);
    controller.Request = new HttpRequestMessage();
    controller.Configuration = new HttpConfiguration();

    // Act
    var response = controller.Get(10);

    // Assert
    Product product;
    Assert.IsTrue(response.TryGetContentValue<Product>(out product));
    Assert.AreEqual(10, product.Id);
}
```

It's important to set **Request** and **Configuration** on the controller. Otherwise, the test will fail with an **ArgumentNullException** or **InvalidOperationException**.

Testing Link Generation

The `Post` method calls **UrlHelper.Link** to create links in the response. This requires a little more setup in the unit test:

```
[TestMethod]
public void PostSetsLocationHeader()
{
    // Arrange
    ProductsController controller = new ProductsController(repository);

    controller.Request = new HttpRequestMessage {
        RequestUri = new Uri("http://localhost/api/products")
    };
    controller.Configuration = new HttpConfiguration();
    controller.Configuration.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional });

    controller.RequestContext.RouteData = new HttpRouteData(
        route: new HttpRoute(),
        values: new HttpRouteValueDictionary { { "controller", "products" } });

    // Act
    Product product = new Product() { Id = 42, Name = "Product1" };
    var response = controller.Post(product);

    // Assert
    Assert.AreEqual("http://localhost/api/products/42", response.Headers.Location.AbsoluteUri);
}
```

The **UrlHelper** class needs the request URL and route data, so the test has to set values for these. Another option is mock or stub **UrlHelper**. With this approach, you replace the default value of ApiController.Url with a mock or stub version that returns a fixed value.

Let's rewrite the test using the Moq framework. Install the `Moq` NuGet package in the test project.

```
[TestMethod]
public void PostSetsLocationHeader_MockVersion()
{
    // This version uses a mock UrlHelper.

    // Arrange
    ProductsController controller = new ProductsController(repository);
    controller.Request = new HttpRequestMessage();
    controller.Configuration = new HttpConfiguration();

    string locationUrl = "http://location/";

    // Create the mock and set up the Link method, which is used to create the Location header.
    // The mock version returns a fixed string.
    var mockUrlHelper = new Mock<UrlHelper>();
    mockUrlHelper.Setup(x => x.Link(It.IsAny<string>(), It.IsAny<object>())).Returns(locationUrl);
    controller.Url = mockUrlHelper.Object;

    // Act
    Product product = new Product() { Id = 42 };
    var response = controller.Post(product);

    // Assert
    Assert.AreEqual(locationUrl, response.Headers.Location.AbsoluteUri);
}
```

In this version, you don't need to set up any route data, because the mock **UrlHelper** returns a constant string.

Testing Actions that Return IHttpActionResult

In Web API 2, a controller action can return **IHttpActionResult**, which is analogous to **ActionResult** in ASP.NET MVC. The **IHttpActionResult** interface defines a command pattern for creating HTTP responses. Instead of creating the response directly, the controller returns an **IHttpActionResult**. Later, the pipeline invokes the **IHttpActionResult** to create the response. This approach makes it easier to write unit tests, because you can skip a lot of the setup that is needed for **HttpResponseMessage**.

Here is an example controller whose actions return **IHttpActionResult**.

```
public class Products2Controller : ApiController
{
    IProductRepository _repository;

    public Products2Controller(IProductRepository repository)
    {
        _repository = repository;
    }

    public IHttpActionResult Get(int id)
    {
        Product product = _repository.GetById(id);
        if (product == null)
        {
            return NotFound();
        }
        return Ok(product);
    }

    public IHttpActionResult Post(Product product)
    {
        _repository.Add(product);
        return CreatedAtRoute("DefaultApi", new { id = product.Id }, product);
    }

    public IHttpActionResult Delete(int id)
    {
        _repository.Delete(id);
        return Ok();
    }

    public IHttpActionResult Put(Product product)
    {
        // Do some work (not shown).
        return Content(HttpStatusCode.Accepted, product);
    }
}
```

This example shows some common patterns using **IHttpActionResult**. Let's see how to unit test them.

Action returns 200 (OK) with a response body

The `Get` method calls `Ok(product)` if the product is found. In the unit test, make sure the return type is **OkNegotiatedContentResult** and the returned product has the right ID.

```
[TestMethod]
public void GetReturnsProductWithSameId()
{
    // Arrange
    var mockRepository = new Mock<IProductRepository>();
    mockRepository.Setup(x => x.GetById(42))
        .Returns(new Product { Id = 42 });

    var controller = new Products2Controller(mockRepository.Object);

    // Act
    IHttpActionResult actionResult = controller.Get(42);
    var contentResult = actionResult as OkNegotiatedContentResult<Product>;

    // Assert
    Assert.IsNotNull(contentResult);
    Assert.IsNotNull(contentResult.Content);
    Assert.AreEqual(42, contentResult.Content.Id);
}
```

Notice that the unit test doesn't execute the action result. You can assume the action result creates the HTTP response correctly. (That's why the Web API framework has its own unit tests!)

Action returns 404 (Not Found)

The `Get` method calls `NotFound()` if the product is not found. For this case, the unit test just checks if the return type is **NotFoundResult**.

```
[TestMethod]
public void GetReturnsNotFound()
{
    // Arrange
    var mockRepository = new Mock<IProductRepository>();
    var controller = new Products2Controller(mockRepository.Object);

    // Act
    IHttpActionResult actionResult = controller.Get(10);

    // Assert
    Assert.IsInstanceOfType(actionResult, typeof(NotFoundResult));
}
```

Action returns 200 (OK) with no response body

The `Delete` method calls `Ok()` to return an empty HTTP 200 response. Like the previous example, the unit test checks the return type, in this case **OkResult**.

```
[TestMethod]
public void DeleteReturnsOk()
{
    // Arrange
    var mockRepository = new Mock<IProductRepository>();
    var controller = new Products2Controller(mockRepository.Object);

    // Act
    IHttpActionResult actionResult = controller.Delete(10);

    // Assert
    Assert.IsInstanceOfType(actionResult, typeof(OkResult));
}
```

Action returns 201 (Created) with a Location header

The `Post` method calls `CreatedAtRoute` to return an HTTP 201 response with a URI in the Location header. In the unit test, verify that the action sets the correct routing values.

```
[TestMethod]
public void PostMethodSetsLocationHeader()
{
    // Arrange
    var mockRepository = new Mock<IProductRepository>();
    var controller = new Products2Controller(mockRepository.Object);

    // Act
    IHttpActionResult actionResult = controller.Post(new Product { Id = 10, Name = "Product1" });
    var createdResult = actionResult as CreatedAtRouteNegotiatedContentResult<Product>;

    // Assert
    Assert.IsNotNull(createdResult);
    Assert.AreEqual("DefaultApi", createdResult.RouteName);
    Assert.AreEqual(10, createdResult.RouteValues["id"]);
}
```

Action returns another 2xx with a response body

The `Put` method calls `Content` to return an HTTP 202 (Accepted) response with a response body. This case is similar to returning 200 (OK), but the unit test should also check the status code.

```
[TestMethod]
public void PutReturnsContentResult()
{
    // Arrange
    var mockRepository = new Mock<IProductRepository>();
    var controller = new Products2Controller(mockRepository.Object);

    // Act
    IHttpActionResult actionResult = controller.Put(new Product { Id = 10, Name = "Product" });
    var contentResult = actionResult as NegotiatedContentResult<Product>;

    // Assert
    Assert.IsNotNull(contentResult);
    Assert.AreEqual(HttpStatusCode.Accepted, contentResult.StatusCode);
    Assert.IsNotNull(contentResult.Content);
    Assert.AreEqual(10, contentResult.Content.Id);
}
```

Additional Resources

- Mocking Entity Framework when Unit Testing ASP.NET Web API 2
- Writing tests for an ASP.NET Web API service (blog post by Youssef Moussaoui).
- Debugging ASP.NET Web API with Route Debugger

Mocking Entity Framework when Unit Testing ASP.NET Web API 2

2/20/2020 • 8 minutes to read • Edit Online

by Tom FitzMacken

Download Completed Project

This guidance and application demonstrate how to create unit tests for your Web API 2 application that uses the Entity Framework. It shows how to modify the scaffolded controller to enable passing a context object for testing, and how to create test objects that work with Entity Framework.

For an introduction to unit testing with ASP.NET Web API, see Unit Testing with ASP.NET Web API 2.

This tutorial assumes you are familiar with the basic concepts of ASP.NET Web API. For an introductory tutorial, see Getting Started with ASP.NET Web API 2.

Software versions used in the tutorial

- Visual Studio 2017
- Web API 2

In this topic

This topic contains the following sections:

- Prerequisites
- Download code
- Create application with unit test project
- Create the model class
- Add the controller
- Add dependency injection
- Install NuGet packages in test project
- Create test context
- Create tests
- Run tests

If you have already completed the steps in Unit Testing with ASP.NET Web API 2, you can skip to the section Add the controller.

Prerequisites

Visual Studio 2017 Community, Professional or Enterprise edition

Download code

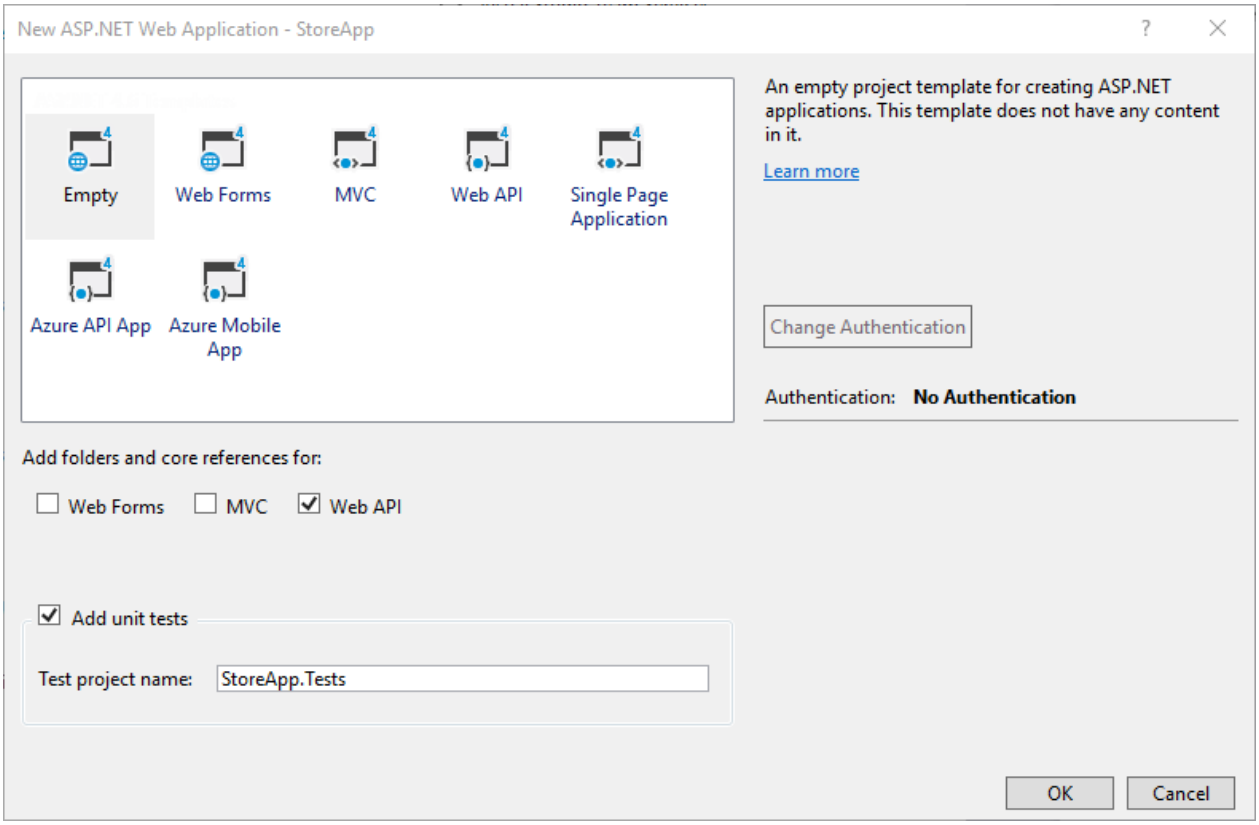
Download the completed project. The downloadable project includes unit test code for this topic and for the Unit Testing ASP.NET Web API 2 topic.

Create application with unit test project

You can either create a unit test project when creating your application or add a unit test project to an existing application. This tutorial shows creating a unit test project when creating the application.

Create a new ASP.NET Web Application named **StoreApp**.

In the New ASP.NET Project windows, select the **Empty** template and add folders and core references for Web API. Select the **Add unit tests** option. The unit test project is automatically named **StoreApp.Tests**. You can keep this name.



After creating the application, you will see it contains two projects - **StoreApp** and **StoreApp.Tests**.

Create the model class

In your StoreApp project, add a class file to the **Models** folder named **Product.cs**. Replace the contents of the file with the following code.

```
using System;

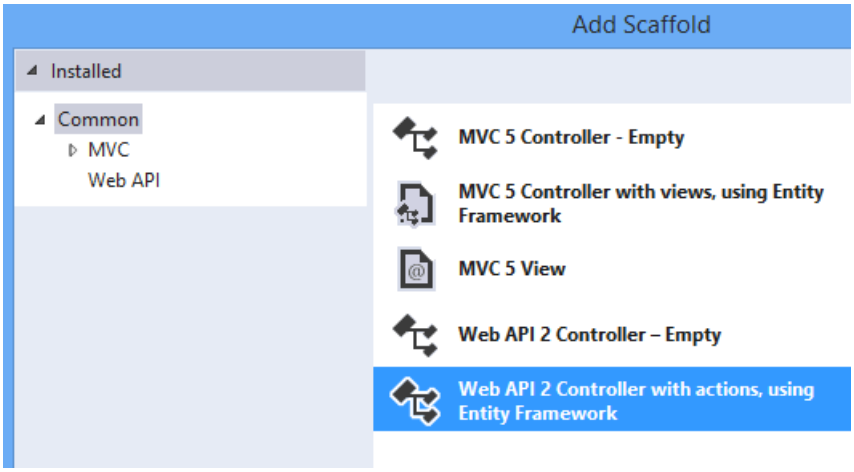
namespace StoreApp.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

Build the solution.

Add the controller

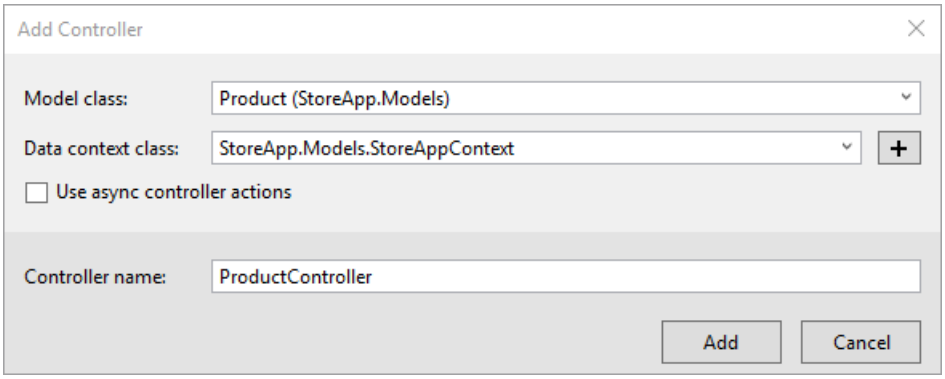
Right-click the Controllers folder and select **Add** and **New Scaffolded Item**. Select Web API 2 Controller with

actions, using Entity Framework.



Set the following values:

- Controller name: **ProductController**
- Model class: **Product**
- Data context class: [Select **New data context** button which fills in the values seen below]



Click **Add** to create the controller with automatically-generated code. The code includes methods for creating, retrieving, updating and deleting instances of the Product class. The following code shows the method for add a Product. Notice that the method returns an instance of **IHttpActionResult**.

```
// POST api/Product
[ResponseType(typeof(Product))]
public IHttpActionResult PostProduct(Product product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    db.Products.Add(product);
    db.SaveChanges();

    return CreatedAtRoute("DefaultApi", new { id = product.Id }, product);
}
```

IHttpActionResult is one of the new features in Web API 2, and it simplifies unit test development.

In the next section, you will customize the generated code to facilitate passing test objects to the controller.

Add dependency injection

Currently, the ProductController class is hard-coded to use an instance of the StoreAppContext class. You will use

a pattern called dependency injection to modify your application and remove that hard-coded dependency. By breaking this dependency, you can pass in a mock object when testing.

Right-click the **Models** folder, and add a new interface named **IStoreAppContext**.

Replace the code with the following code.

```
using System;
using System.Data.Entity;

namespace StoreApp.Models
{
    public interface IStoreAppContext : IDisposable
    {
        DbSet<Product> Products { get; }
        int SaveChanges();
        void MarkAsModified(Product item);
    }
}
```

Open the StoreAppContext.cs file and make the following highlighted changes. The important changes to note are:

- StoreAppContext class implements IStoreAppContext interface
- MarkAsModified method is implemented

```
using System;
using System.Data.Entity;

namespace StoreApp.Models
{
    public class StoreAppContext : DbContext, IStoreAppContext
    {
        public StoreAppContext() : base("name=StoreAppContext")
        {
        }

        public DbSet<Product> Products { get; set; }

        public void MarkAsModified(Product item)
        {
            Entry(item).State = EntityState.Modified;
        }
    }
}
```

Open the ProductController.cs file. Change the existing code to match the highlighted code. These changes break the dependency on StoreAppContext and enable other classes to pass in a different object for the context class. This change will enable you to pass in a test context during unit tests.

```
public class ProductController : ApiController
{
    // modify the type of the db field
    private IStoreAppContext db = new StoreAppContext();

    // add these constructors
    public ProductController() { }

    public ProductController(IStoreAppContext context)
    {
        db = context;
    }
    // rest of class not shown
}
```

There is one more change you must make in ProductController. In the **PutProduct** method, replace the line that sets the entity state to modified with a call to the MarkAsModified method.

```
// PUT api/Product/5
public IHttpActionResult PutProduct(int id, Product product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    if (id != product.Id)
    {
        return BadRequest();
    }

    //db.Entry(product).State = EntityState.Modified;
    db.MarkAsModified(product);

    // rest of method not shown
}
```

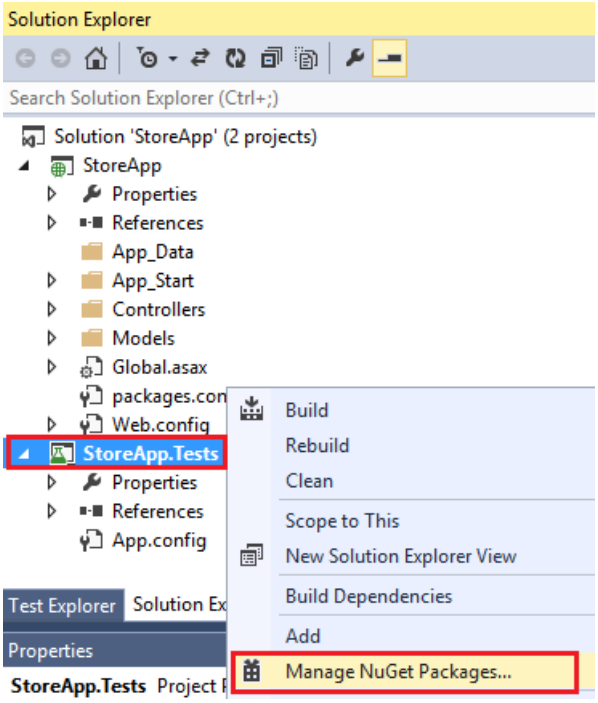
Build the solution.

You are now ready to set up the test project.

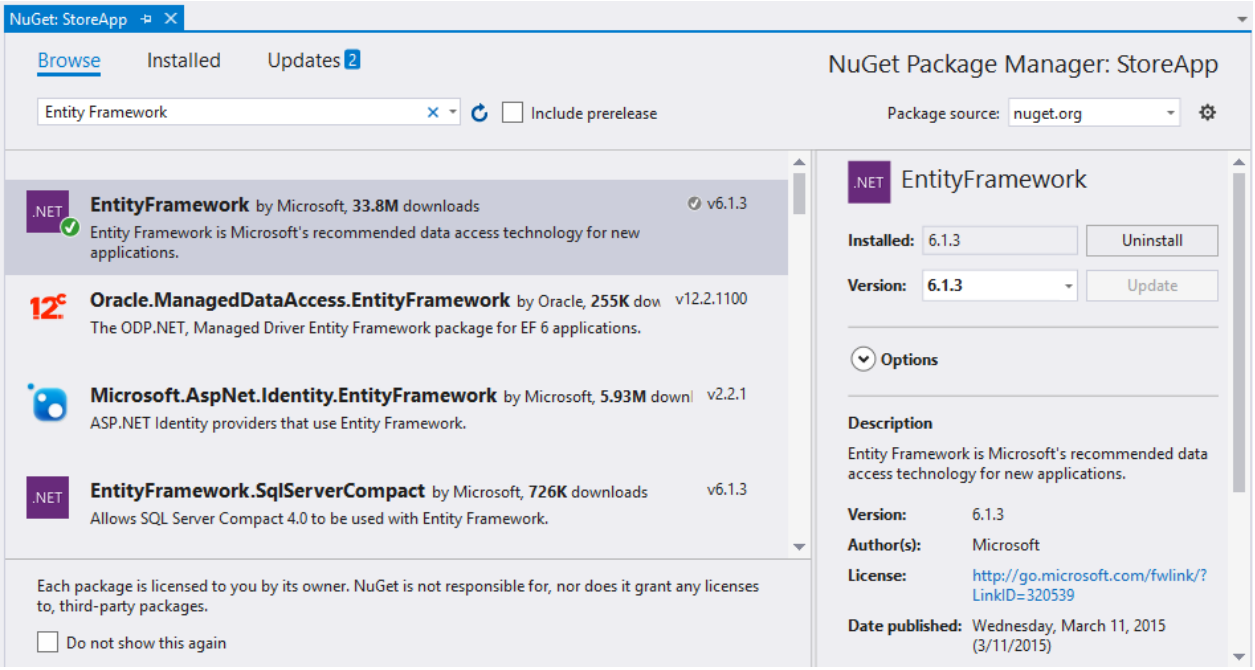
Install NuGet packages in test project

When you use the Empty template to create an application, the unit test project (StoreApp.Tests) does not include any installed NuGet packages. Other templates, such as the Web API template, include some NuGet packages in the unit test project. For this tutorial, you must include the Entity Framework package and the Microsoft ASP.NET Web API 2 Core package to the test project.

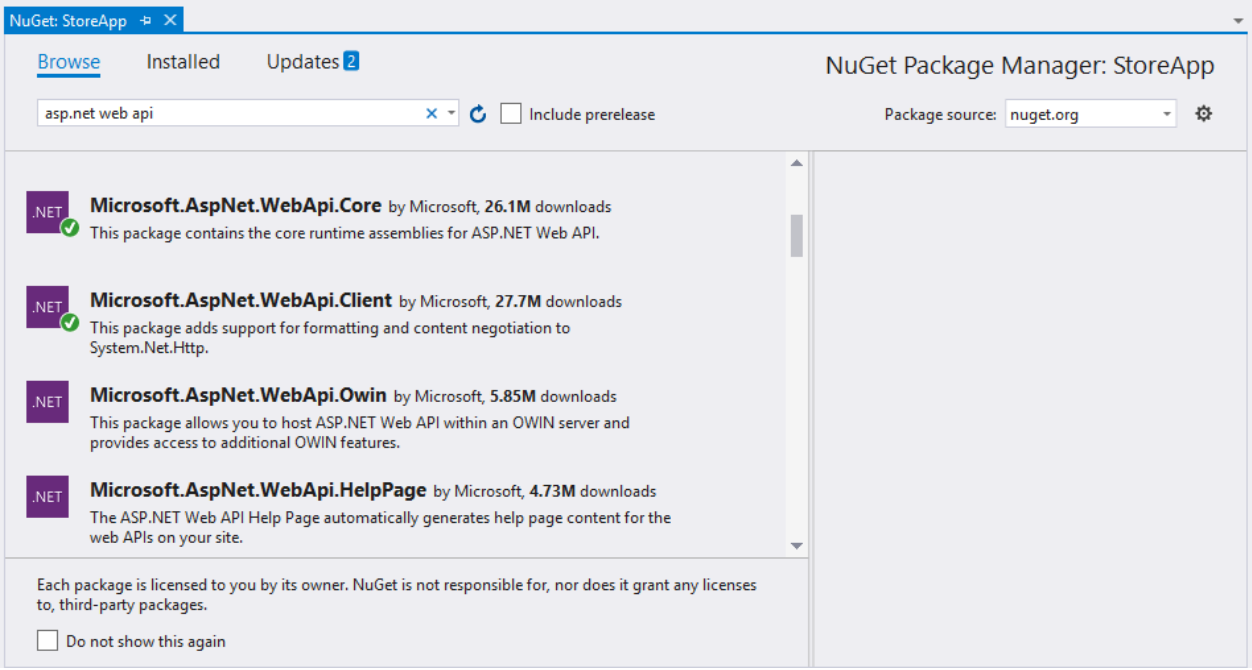
Right-click the StoreApp.Tests project and select **Manage NuGet Packages**. You must select the StoreApp.Tests project to add the packages to that project.



From the Online packages, find and install the EntityFramework package (version 6.0 or later). If it appears that the EntityFramework package is already installed, you may have selected the StoreApp project instead of the StoreApp.Tests project.



Find and install Microsoft ASP.NET Web API 2 Core package.



Close the Manage NuGet Packages window.

Create test context

Add a class named **TestDbSet** to the test project. This class serves as the base class for your test data set. Replace the code with the following code.

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Data.Entity;
using System.Linq;

namespace StoreApp.Tests
{
    public class TestDbSet<T> : DbSet<T>, IQueryable, IEnumerable<T>
        where T : class
    {
        ObservableCollection<T> _data;
        IQueryable _query;

        public TestDbSet()
        {
            _data = new ObservableCollection<T>();
            _query = _data.AsQueryable();
        }

        public override T Add(T item)
        {
            _data.Add(item);
            return item;
        }

        public override T Remove(T item)
        {
            _data.Remove(item);
            return item;
        }

        public override T Attach(T item)
        {
            _data.Add(item);
            return item;
```

```
        }

        public override T Create()
        {
            return Activator.CreateInstance<T>();
        }

        public override TDerivedEntity Create<TDerivedEntity>()
        {
            return Activator.CreateInstance<TDerivedEntity>();
        }

        public override ObservableCollection<T> Local
        {
            get { return new ObservableCollection<T>(_data); }
        }

        Type IQueryable.ElementType
        {
            get { return _query.ElementType; }
        }

        System.Linq.Expressions.Expression IQueryable.Expression
        {
            get { return _query.Expression; }
        }

        IQueryProvider IQueryable.Provider
        {
            get { return _query.Provider; }
        }

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
        {
            return _data.GetEnumerator();
        }

        IEnumerator<T> IEnumerable<T>.GetEnumerator()
        {
            return _data.GetEnumerator();
        }
    }
}
```

Add a class named **TestProductDbSet** to the test project which contains the following code.

```
using System;
using System.Linq;
using StoreApp.Models;

namespace StoreApp.Tests
{
    class TestProductDbSet : TestDbSet<Product>
    {
        public override Product Find(params object[] keyValues)
        {
            return this.SingleOrDefault(product => product.Id == (int)keyValues.Single());
        }
    }
}
```

Add a class named **TestStoreAppContext** and replace the existing code with the following code.

```
using System;
using System.Data.Entity;
using StoreApp.Models;

namespace StoreApp.Tests
{
    public class TestStoreAppContext : IStoreAppContext
    {
        public TestStoreAppContext()
        {
            this.Products = new TestProductDbSet();
        }

        public DbSet<Product> Products { get; set; }

        public int SaveChanges()
        {
            return 0;
        }

        public void MarkAsModified(Product item) { }
        public void Dispose() { }
    }
}
```

Create tests

By default, your test project includes an empty test file named **UnitTest1.cs**. This file shows the attributes you use to create test methods. For this tutorial, you can delete this file because you will add a new test class.

Add a class named **TestProductController** to the test project. Replace the code with the following code.

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Web.Http.Results;
using System.Net;
using StoreApp.Models;
using StoreApp.Controllers;

namespace StoreApp.Tests
{
    [TestClass]
    public class TestProductController
    {
        [TestMethod]
        public void PostProduct_ShouldReturnSameProduct()
        {
            var controller = new ProductController(new TestStoreAppContext());

            var item = GetDemoProduct();

            var result =
                controller.PostProduct(item) as CreatedAtRouteNegotiatedContentResult<Product>;

            Assert.IsNotNull(result);
            Assert.AreEqual(result.RouteName, "DefaultApi");
            Assert.AreEqual(result.RouteValues["id"], result.Content.Id);
            Assert.AreEqual(result.Content.Name, item.Name);
        }

        [TestMethod]
        public void PutProduct_ShouldReturnStatusCode()
        {
            var controller = new ProductController(new TestStoreAppContext());
```

```
            var item = GetDemoProduct();

            var result = controller.PutProduct(item.Id, item) as StatusCodeResult;
            Assert.IsNotNull(result);
            Assert.IsInstanceOfType(result, typeof(StatusCodeResult));
            Assert.AreEqual(HttpStatusCode.NoContent, result.StatusCode);
        }

        [TestMethod]
        public void PutProduct_ShouldFail_WhenDifferentID()
        {
            var controller = new ProductController(new TestStoreAppContext());

            var badresult = controller.PutProduct(999, GetDemoProduct());
            Assert.IsInstanceOfType(badresult, typeof(BadRequestResult));
        }

        [TestMethod]
        public void GetProduct_ShouldReturnProductWithSameID()
        {
            var context = new TestStoreAppContext();
            context.Products.Add(GetDemoProduct());

            var controller = new ProductController(context);
            var result = controller.GetProduct(3) as OkNegotiatedContentResult<Product>;

            Assert.IsNotNull(result);
            Assert.AreEqual(3, result.Content.Id);
        }

        [TestMethod]
        public void GetProducts_ShouldReturnAllProducts()
        {
            var context = new TestStoreAppContext();
            context.Products.Add(new Product { Id = 1, Name = "Demo1", Price = 20 });
            context.Products.Add(new Product { Id = 2, Name = "Demo2", Price = 30 });
            context.Products.Add(new Product { Id = 3, Name = "Demo3", Price = 40 });

            var controller = new ProductController(context);
            var result = controller.GetProducts() as TestProductDbSet;

            Assert.IsNotNull(result);
            Assert.AreEqual(3, result.Local.Count);
        }

        [TestMethod]
        public void DeleteProduct_ShouldReturnOK()
        {
            var context = new TestStoreAppContext();
            var item = GetDemoProduct();
            context.Products.Add(item);

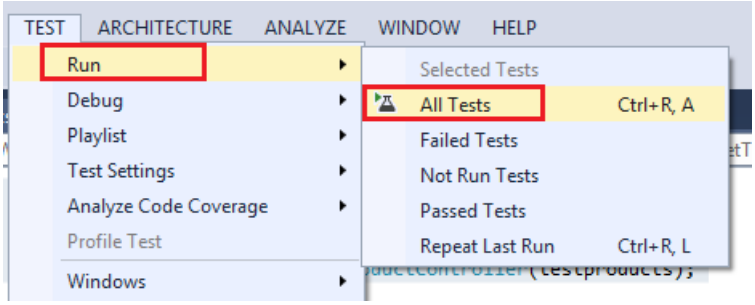
            var controller = new ProductController(context);
            var result = controller.DeleteProduct(3) as OkNegotiatedContentResult<Product>;

            Assert.IsNotNull(result);
            Assert.AreEqual(item.Id, result.Content.Id);
        }

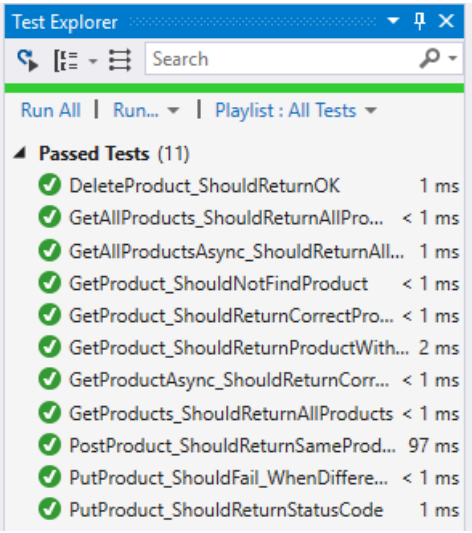
        Product GetDemoProduct()
        {
            return new Product() { Id = 3, Name = "Demo name", Price = 5 };
        }
    }
}
```

Run tests

You are now ready to run the tests. All of the method that are marked with the **TestMethod** attribute will be tested. From the **Test** menu item, run the tests.



Open the **Test Explorer** window, and notice the results of the tests.



Troubleshoot Web API2 apps that work in Visual Studio and fail on a production IIS server

2/20/2020 • 8 minutes to read • Edit Online

This doc explains how to troubleshoot Web API2 apps that are deployed to a production IIS server. It addresses common HTTP 405 and 501 errors.

Software used in this tutorial

- Internet Information Services (IIS) (version 7 or later)
- Web API

Web API apps typically use several HTTP verbs: GET, POST, PUT, DELETE, and sometimes PATCH. That being said, developers may run into situations where those verbs are implemented by another IIS module on their production IIS server, which leads to a situation where a Web API controller that works correctly in Visual Studio or on a development server will return an HTTP 405 error when it is deployed to a production IIS server.

What causes HTTP 405 errors

The first step toward learning how to troubleshoot HTTP 405 errors is to understand what an HTTP 405 error actually means. The primary governing document for HTTP is RFC 2616, which defines the HTTP 405 status code as ***Method Not Allowed***, and further describes this status code as a situation where "the method specified in the Request-Line is not allowed for the resource identified by the Request-URI." In other words, the HTTP verb is not allowed for the specific URL that an HTTP client has requested.

As a brief review, here are several of the most-used HTTP methods as defined in RFC 2616, RFC 4918, and RFC 5789:

HTTP METHOD	DESCRIPTION
GET	This method is used to retrieve data from a URI, and it probably the most-used HTTP method.
HEAD	This method is much like the GET method, except that it doesn't actually retrieve the data from the request URI - it simply retrieves the HTTP status.
POST	This method is typically used to send new data to the URI; POST is often used to submit form data.
PUT	This method is typically used to send raw data to the URI; PUT is often used to submit JSON or XML data to Web API applications.
DELETE	This method is used to remove data from a URI.
OPTIONS	This method is typically used to retrieve the list of HTTP methods that are supported for a URI.

HTTP METHOD	DESCRIPTION
COPY MOVE	These two methods are used with WebDAV, and their purpose is self-explanatory.
MKCOL	This method is used with WebDAV, and it is used to create a collection (e.g. a directory) at the specified URI.
PROPFIND PROPPATCH	These two methods are used with WebDAV, and they are used to query or set properties for a URI.
LOCK UNLOCK	These two methods are used with WebDAV, and they are used to lock/unlock the resource identified by the request URI when authoring.
PATCH	This method is used to modify an existing HTTP resource.

When one of these HTTP methods is configured for use on the server, the server will respond with the HTTP status and other data that is appropriate for the request. (For example, a GET method might receive an HTTP 200 ***OK*** response, and a PUT method might receive an HTTP 201 ***Created*** response.)

If the HTTP method is not configured for use on the server, the server will respond with an HTTP 501 ***Not Implemented*** error.

However, when an HTTP method is configured for use on the server, but it has been disabled for a given URI, the server will respond with an HTTP 405 ***Method Not Allowed*** error.

Example HTTP 405 error

The following example HTTP request and response illustrate a situation where an HTTP client is attempting to PUT value to a Web API app on a web server, and the server returns an HTTP error which states that the PUT method is not allowed:

HTTP Request:

```
PUT /api/values/1 HTTP/1.1
Content-type: application/json
Host: localhost
Accept: */*
Content-Length: 12

"Some Value"
```

HTTP Response:

```
HTTP/1.1 405 Method Not Allowed
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/8.0
X-Powered-By: ASP.NET
Date: Wed, 15 May 2013 02:38:57 GMT
Content-Length: 72

{"Message":"The requested resource does not support http method 'PUT'."}
```

In this example, the HTTP client sent a valid JSON request to the URL for a Web API application on a web server, but the server returned an HTTP 405 error message which indicates that the PUT method was not allowed for the URL. In contrast, if the request URI did not match a route for the Web API application, the server would return an HTTP 404 ***Not Found*** error.

Resolve HTTP 405 errors

There are several reasons why a specific HTTP verb may not be allowed, but there is one primary scenario that is the leading cause of this error in IIS: multiple handlers are defined for the same verb/method, and one of the handlers is blocking the expected handler from processing the request. By way of explanation, IIS processes handlers from first to last based on the order handler entries in the *applicationHost.config* and *web.config* files, where the first matching combination of path, verb, resource, etc., will be used to handle the request.

The following example is an excerpt from an *applicationHost.config* file for an IIS server that was returning an HTTP 405 error when using the PUT method to submit data to a Web API application. In this excerpt, several HTTP handlers are defined, and each handler has a different set of HTTP methods for which it is configured - the last entry in the list is the static content handler, which is the default handler that is used after the other handlers have had a chance to examine the request:

```
<handlers accessPolicy="Read, Script">
   <add name="WebDAV"
      path="*"
      verb="PROPFIND,PROPPATCH,MKCOL,PUT,COPY,DELETE,MOVE,LOCK,UNLOCK"
      modules="WebDAVModule"
      resourceType="Unspecified"
      requireAccess="None" />
   <add name="ISAPI-dll"
      path="*.dll"
      verb="*"
      modules="IsapiModule"
      resourceType="File"
      requireAccess="Execute"
      allowPathInfo="true" />
   <add name="ExtensionlessUrlHandler-ISAPI-4.0_64bit"
      path="*."
      verb="GET,HEAD,POST,DEBUG"
      modules="IsapiModule"
      scriptProcessor="%windir%\Microsoft.NET\Framework64\v4.0.30319\aspnet_isapi.dll"
      preCondition="classicMode,runtimeVersionv4.0,bitness64"
      responseBufferLimit="0" />

   <!-- Additional handlers will be defined here. -->

   <add name="StaticFile"
      path="*"
      verb="*"
      modules="StaticFileModule,DefaultDocumentModule,DirectoryListingModule"
      resourceType="Either"
      requireAccess="Read" />
</handlers>
```

In the preceding example, the WebDAV handler and the Extension-less URL Handler for ASP.NET (which is used for Web API) are clearly defined for separate lists of HTTP methods. Note that the ISAPI DLL handler is configured for all HTTP methods, although this configuration will not necessarily cause an error. However, configuration settings like this need to be considered when troubleshooting HTTP 405 errors.

In the preceding example, the ISAPI DLL handler was not the problem; in fact, the problem was not defined in the *applicationHost.config* file for the IIS server - the problem was caused by an entry that was made in the *web.config* file when the Web API application was created in Visual Studio. The following excerpt from the application's *web.config* file shows the location of the problem:

```
<handlers accessPolicy="Read, Script">
   <remove name="ExtensionlessUrlHandler-ISAPI-4.0_64bit" />
   <add name="ExtensionlessUrlHandler-ISAPI-4.0_64bit"
      path="*."
      verb="GET,HEAD,POST,DEBUG,PUT,DELETE,PATCH,OPTIONS"
      modules="IsapiModule"
      scriptProcessor="%windir%\Microsoft.NET\Framework64\v4.0.30319\aspnet_isapi.dll"
      preCondition="classicMode,runtimeVersionv4.0,bitness64"
      responseBufferLimit="0" />
</handlers>
```

In this excerpt, the Extension-less URL Handler for ASP.NET is redefined to include additional HTTP methods that will be used with the Web API application. However, since a similar set of HTTP methods is defined for the WebDAV handler, a conflict occurs. In this specific case, the WebDAV handler is defined and loaded by IIS, even though WebDAV is disabled for the website that includes the Web API application. During the processing of an HTTP PUT request, IIS calls the WebDAV module since it is defined for the PUT verb. When the WebDAV module is called, it checks its configuration and sees that it is disabled, so it will return an HTTP 405 ***Method Not Allowed*** error for any request that resembles a WebDAV request. To resolve this issue, you should remove WebDAV from the list of HTTP modules for the website where your Web API application is defined. The following example shows what that might look like:

```
<handlers accessPolicy="Read, Script">
   <remove name="WebDAV" />
   <remove name="ExtensionlessUrlHandler-ISAPI-4.0_64bit" />
   <add name="ExtensionlessUrlHandler-ISAPI-4.0_64bit"
      path="*."
      verb="GET,HEAD,POST,DEBUG,PUT,DELETE,PATCH,OPTIONS"
      modules="IsapiModule"
      scriptProcessor="%windir%\Microsoft.NET\Framework64\v4.0.30319\aspnet_isapi.dll"
      preCondition="classicMode,runtimeVersionv4.0,bitness64"
      responseBufferLimit="0" />
</handlers>
```

This scenario is often encountered after an application is published from a development environment to an IIS production environment, and this occurs because the list of handlers/modules is different between your development and production environments. For example, if you are using Visual Studio 2012 or later to develop a Web API application, IIS Express is the default web server for testing. This development web server is a scaled-down version of the full IIS functionality that ships in a server product, and this development web server contains a few changes that were added for development scenarios. For example, the WebDAV module is often installed on a production web server that is running the full version of IIS, although it may not be in use. The development version of IIS, (IIS Express), installs the WebDAV module, but the entries for the WebDAV module are intentionally commented out, so the WebDAV module is never loaded on IIS Express unless you specifically alter your IIS Express configuration settings to add WebDAV functionality to your IIS Express installation. As a result, your web application may work correctly on your development computer, but you may encounter HTTP 405 errors when you publish your Web API application to your production IIS web server.

HTTP 501 errors

- Indicates the specific functionality has not been implemented on the server.
- Typically means that there is no handler defined in your IIS settings which matches the HTTP request:
  - Probably indicates that something was not installed correctly on IIS or
  - Something has modified your IIS settings so that there are no handlers defined that support the specific HTTP method.

To resolve that issue, you would need to reinstall any application that is attempting to use an HTTP method for

which it has no corresponding module or handler definitions.

Summary

HTTP 405 errors are caused when an HTTP method is not allowed by a web server for a requested URL. This condition is often seen when a particular handler has been defined for a specific verb, and that handler is overriding the handler that you expect to process the request.

Security, Authentication, and Authorization in ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

Security issues for Web API.

- Authentication and Authorization in Web API
- Secure a Web API with Individual Accounts in Web API 2.2
- External Authentication Services with Web API (C#)
- Preventing Cross-Site Request Forgery (CSRF) Attacks in Web API
- Enabling Cross-Origin Requests in Web API 2
- Authentication Filters in Web API 2
- Basic Authentication in Web API
- Forms Authentication in Web API
- Integrated Windows Authentication
- Working with SSL

Authentication and Authorization in ASP.NET Web API

2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson

You've created a web API, but now you want to control access to it. In this series of articles, we'll look at some options for securing a web API from unauthorized users. This series will cover both authentication and authorization.

- *Authentication* is knowing the identity of the user. For example, Alice logs in with her username and password, and the server uses the password to authenticate Alice.
- *Authorization* is deciding whether a user is allowed to perform an action. For example, Alice has permission to get a resource but not create a resource.

The first article in the series gives a general overview of authentication and authorization in ASP.NET Web API. Other topics describe common authentication scenarios for Web API.

> **NOTE**
>
> Thanks to the people who reviewed this series and provided valuable feedback: Rick Anderson, Levi Broderick, Barry Dorrans, Tom Dykstra, Hongmei Ge, David Matson, Daniel Roth, Tim Teebken.

Authentication

Web API assumes that authentication happens in the host. For web-hosting, the host is IIS, which uses HTTP modules for authentication. You can configure your project to use any of the authentication modules built in to IIS or ASP.NET, or write your own HTTP module to perform custom authentication.

When the host authenticates the user, it creates a *principal*, which is an IPrincipal object that represents the security context under which code is running. The host attaches the principal to the current thread by setting **Thread.CurrentPrincipal**. The principal contains an associated **Identity** object that contains information about the user. If the user is authenticated, the **Identity.IsAuthenticated** property returns **true**. For anonymous requests, **IsAuthenticated** returns **false**. For more information about principals, see Role-Based Security.

HTTP Message Handlers for Authentication

Instead of using the host for authentication, you can put authentication logic into an HTTP message handler. In that case, the message handler examines the HTTP request and sets the principal.

When should you use message handlers for authentication? Here are some tradeoffs:

- An HTTP module sees all requests that go through the ASP.NET pipeline. A message handler only sees requests that are routed to Web API.
- You can set per-route message handlers, which lets you apply an authentication scheme to a specific route.
- HTTP modules are specific to IIS. Message handlers are host-agnostic, so they can be used with both web-hosting and self-hosting.
- HTTP modules participate in IIS logging, auditing, and so on.
- HTTP modules run earlier in the pipeline. If you handle authentication in a message handler, the principal does not get set until the handler runs. Moreover, the principal reverts back to the previous principal when the response leaves the message handler.

Generally, if you don't need to support self-hosting, an HTTP module is a better option. If you need to support self-hosting, consider a message handler.

Setting the Principal

If your application performs any custom authentication logic, you must set the principal on two places:

- **Thread.CurrentPrincipal**. This property is the standard way to set the thread's principal in .NET.
- **HttpContext.Current.User**. This property is specific to ASP.NET.

The following code shows how to set the principal:

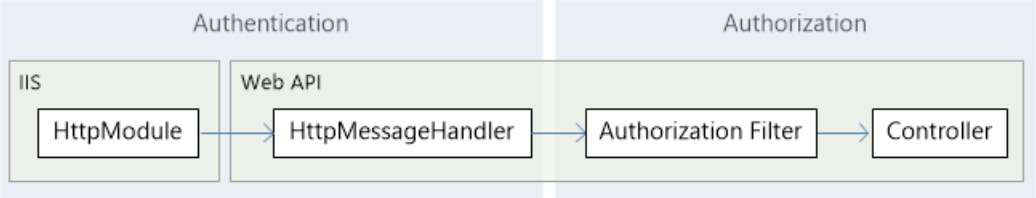
```
private void SetPrincipal(IPrincipal principal)
{
    Thread.CurrentPrincipal = principal;
    if (HttpContext.Current != null)
    {
        HttpContext.Current.User = principal;
    }
}
```

For web-hosting, you must set the principal in both places; otherwise the security context may become inconsistent. For self-hosting, however, **HttpContext.Current** is null. To ensure your code is host-agnostic, therefore, check for null before assigning to **HttpContext.Current**, as shown.

Authorization

Authorization happens later in the pipeline, closer to the controller. That lets you make more granular choices when you grant access to resources.

- *Authorization filters* run before the controller action. If the request is not authorized, the filter returns an error response, and the action is not invoked.
- Within a controller action, you can get the current principal from the **ApiController.User** property. For example, you might filter a list of resources based on the user name, returning only those resources that belong to that user.



Using the [Authorize] Attribute

Web API provides a built-in authorization filter, AuthorizeAttribute. This filter checks whether the user is authenticated. If not, it returns HTTP status code 401 (Unauthorized), without invoking the action.

You can apply the filter globally, at the controller level, or at the level of individual actions.

Globally: To restrict access for every Web API controller, add the **AuthorizeAttribute** filter to the global filter list:

```
public static void Register(HttpConfiguration config)
{
    config.Filters.Add(new AuthorizeAttribute());
}
```

Controller: To restrict access for a specific controller, add the filter as an attribute to the controller:

```
// Require authorization for all actions on the controller.
[Authorize]
public class ValuesController : ApiController
{
    public HttpResponseMessage Get(int id) { ... }
    public HttpResponseMessage Post() { ... }
}
```

Action: To restrict access for specific actions, add the attribute to the action method:

```
public class ValuesController : ApiController
{
    public HttpResponseMessage Get() { ... }

    // Require authorization for a specific action.
    [Authorize]
    public HttpResponseMessage Post() { ... }
}
```

Alternatively, you can restrict the controller and then allow anonymous access to specific actions, by using the `[AllowAnonymous]` attribute. In the following example, the `Post` method is restricted, but the `Get` method allows anonymous access.

```
[Authorize]
public class ValuesController : ApiController
{
    [AllowAnonymous]
    public HttpResponseMessage Get() { ... }

    public HttpResponseMessage Post() { ... }
}
```

In the previous examples, the filter allows any authenticated user to access the restricted methods; only anonymous users are kept out. You can also limit access to specific users or to users in specific roles:

```
// Restrict by user:
[Authorize(Users="Alice,Bob")]
public class ValuesController : ApiController
{
}

// Restrict by role:
[Authorize(Roles="Administrators")]
public class ValuesController : ApiController
{
}
```

> **NOTE**
>
> The **AuthorizeAttribute** filter for Web API controllers is located in the **System.Web.Http** namespace. There is a similar filter for MVC controllers in the **System.Web.Mvc** namespace, which is not compatible with Web API controllers.

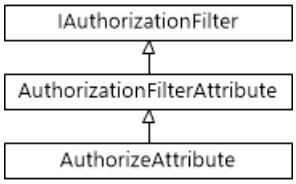
Custom Authorization Filters

To write a custom authorization filter, derive from one of these types:

- **AuthorizeAttribute**. Extend this class to perform authorization logic based on the current user and the user's roles.

- **AuthorizationFilterAttribute**. Extend this class to perform synchronous authorization logic that is not necessarily based on the current user or role.
- **IAuthorizationFilter**. Implement this interface to perform asynchronous authorization logic; for example, if your authorization logic makes asynchronous I/O or network calls. (If your authorization logic is CPU-bound, it is simpler to derive from **AuthorizationFilterAttribute**, because then you don't need to write an asynchronous method.)

The following diagram shows the class hierarchy for the **AuthorizeAttribute** class.



Authorization Inside a Controller Action

In some cases, you might allow a request to proceed, but change the behavior based on the principal. For example, the information that you return might change depending on the user's role. Within a controller method, you can get the current principal from the **ApiController.User** property.

```
public HttpResponseMessage Get()
{
    if (User.IsInRole("Administrators"))
    {
        // ...
    }
}
```

Secure a Web API with Individual Accounts and Local Login in ASP.NET Web API 2.2

2/20/2020 • 11 minutes to read • Edit Online

by Mike Wasson

Download Sample App

This topic shows how to secure a web API using OAuth2 to authenticate against a membership database.

Software versions used in the tutorial

- Visual Studio 2013 Update 3
- Web API 2.2
- ASP.NET Identity 2.1

In Visual Studio 2013, the Web API project template gives you three options for authentication:

- **Individual accounts.** The app uses a membership database.
- **Organizational accounts.** Users sign in with their Azure Active Directory, Office 365, or on-premise Active Directory credentials.
- **Windows authentication.** This option is intended for Intranet applications, and uses the Windows Authentication IIS module.

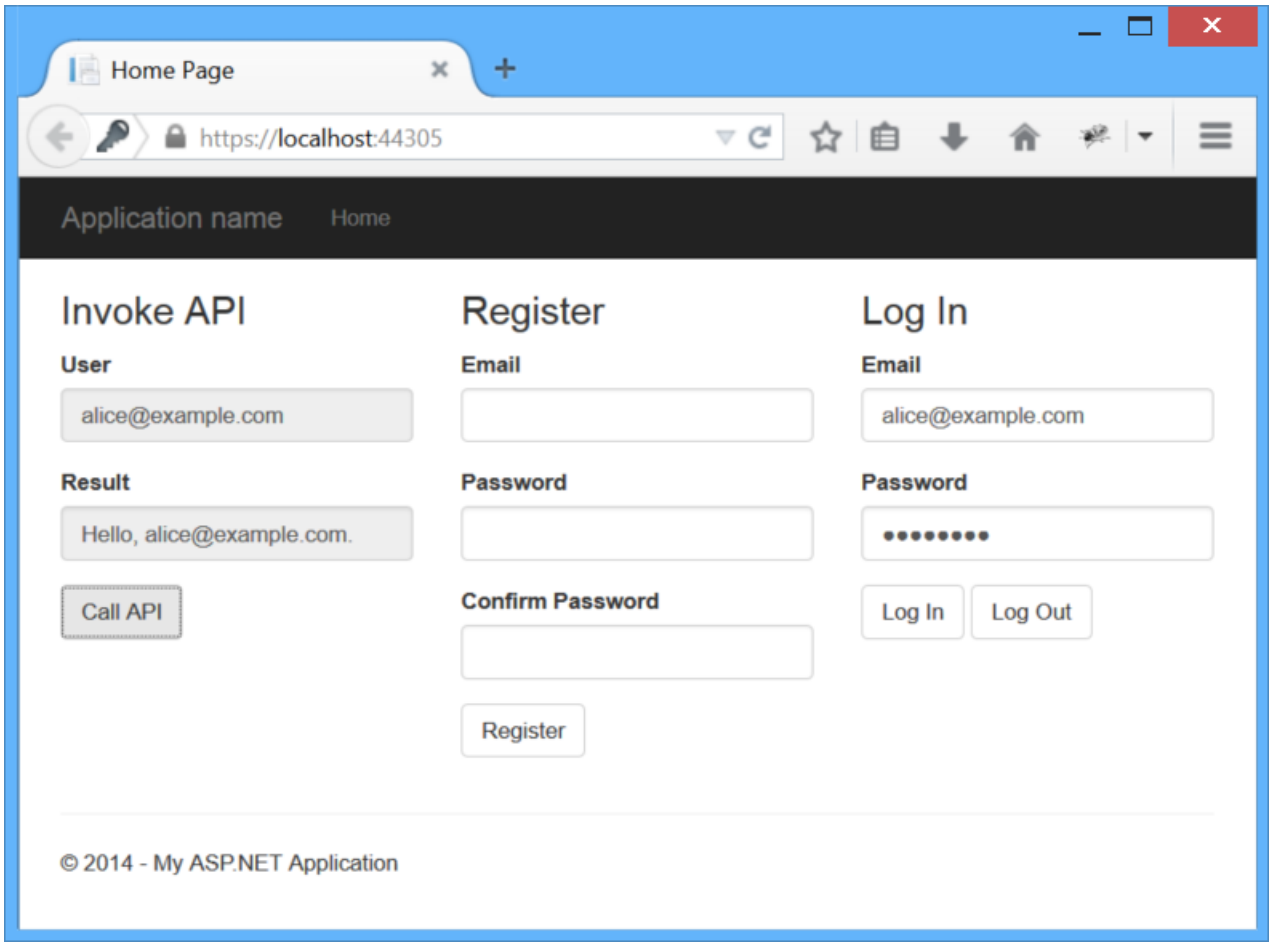
For more details about these options, see Creating ASP.NET Web Projects in Visual Studio 2013.

Individual accounts provide two ways for a user to log in:

- **Local login**. The user registers at the site, entering a username and password. The app stores the password hash in the membership database. When the user logs in, the ASP.NET Identity system verifies the password.
- **Social login**. The user signs in with an external service, such as Facebook, Microsoft, or Google. The app still creates an entry for the user in the membership database, but does not store any credentials. The user authenticates by signing into the external service.

This article looks at the local login scenario. For both local and social login, Web API uses OAuth2 to authenticate requests. However, the credential flows are different for local and social login.

In this article, I'll demonstrate a simple app that lets the user log in and send authenticated AJAX calls to a web API. You can download the sample code here. The readme describes how to create the sample from scratch in Visual Studio.



The sample app uses Knockout.js for data-binding and jQuery for sending AJAX requests. I'll be focusing on the AJAX calls, so you don't need to know Knockout.js for this article.

Along the way, I'll describe:

- What the app is doing on the client side.
- What's happening on the server.
- The HTTP traffic in the middle.

First, we need to define some OAuth2 terminology.

- *Resource*. Some piece of data that can be protected.
- *Resource server*. The server that hosts the resource.
- *Resource owner*. The entity that can grant permission to access a resource. (Typically the user.)
- *Client*. The app that wants access to the resource. In this article, the client is a web browser.
- *Access token*. A token that grants access to a resource.
- *Bearer token*. A particular type of access token, with the property that anyone can use the token. In other words, a client doesn't need a cryptographic key or other secret to use a bearer token. For that reason, bearer tokens should only be used over a HTTPS, and should have relatively short expiration times.
- *Authorization server*. A server that gives out access tokens.

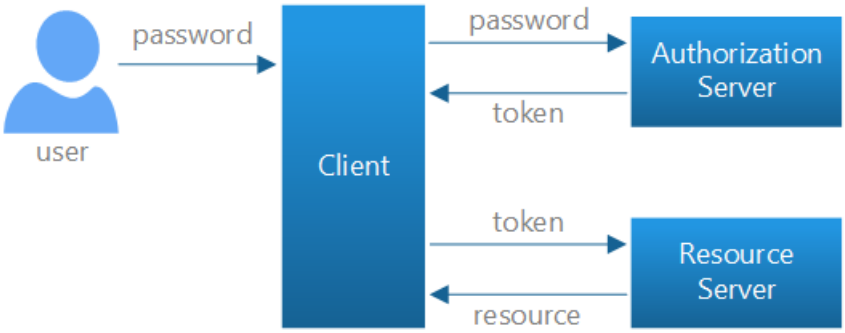
An application can act as both authorization server and resource server. The Web API project template follows this pattern.

Local Login Credential Flow

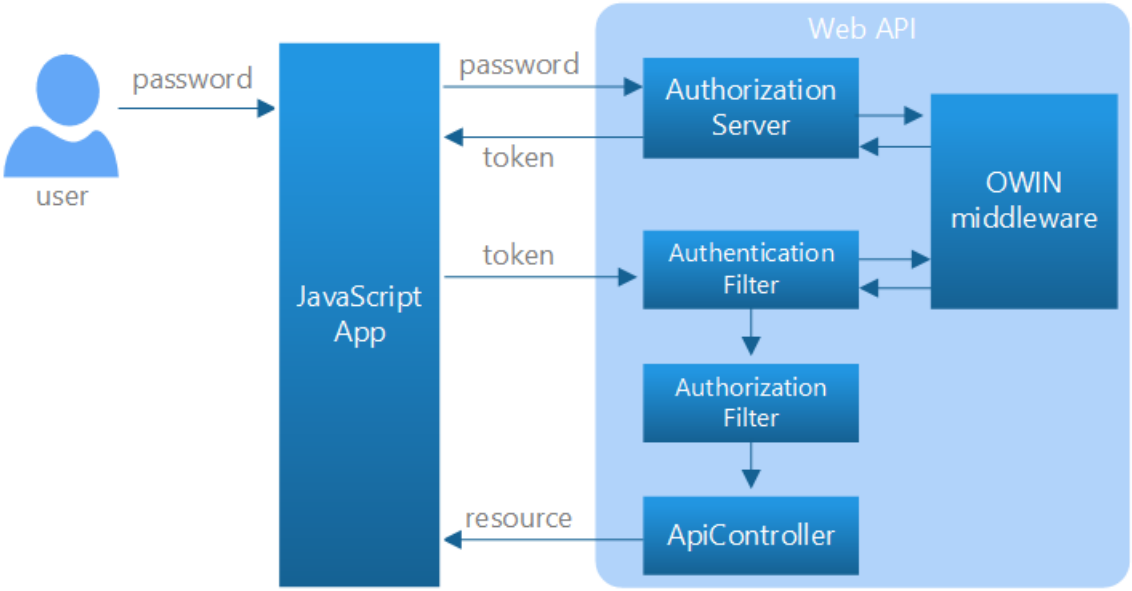
For local login, Web API uses the resource owner password flow defined in OAuth2.

1. The user enters a name and password into the client.

2. The client sends these credentials to the authorization server.
3. The authorization server authenticates the credentials and returns an access token.
4. To access a protected resource, the client includes the access token in the Authorization header of the HTTP request.



When you select **Individual accounts** in the Web API project template, the project includes an authorization server that validates user credentials and issues tokens. The following diagram shows the same credential flow in terms of Web API components.

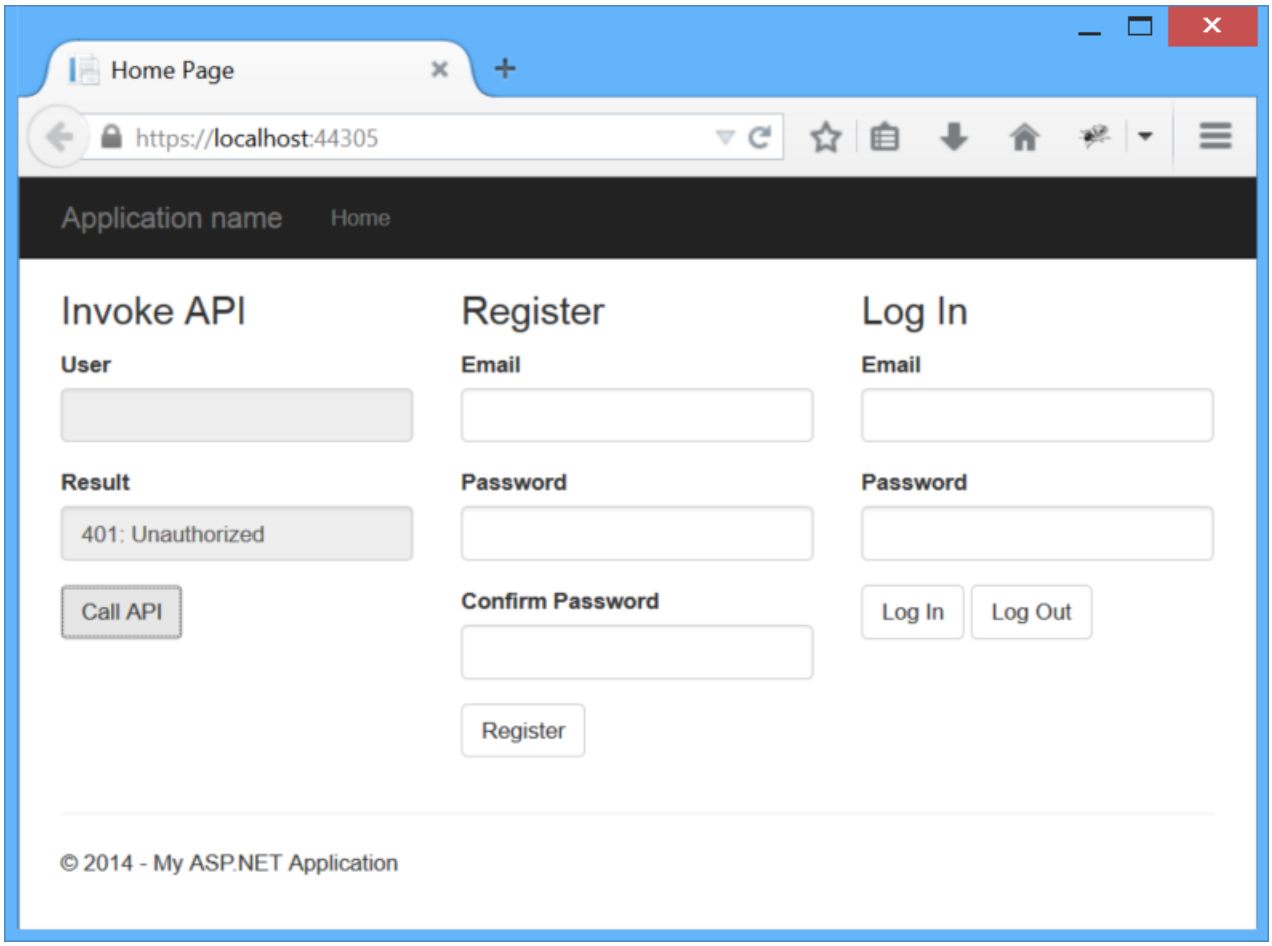


In this scenario, Web API controllers act as resource servers. An authentication filter validates access tokens, and the **[Authorize]** attribute is used to protect a resource. When a controller or action has the **[Authorize]** attribute, all requests to that controller or action must be authenticated. Otherwise, authorization is denied, and Web API returns a 401 (Unauthorized) error.

The authorization server and the authentication filter both call into an OWIN middleware component that handles the details of OAuth2. I'll describe the design in more detail later in this tutorial.

Sending an Unauthorized Request

To get started, run the app and click the **Call API** button. When the request completes, you should see an error message in the **Result** box. That's because the request does not contain an access token, so the request is unauthorized.



The **Call API** button sends an AJAX request to ~/api/values, which invokes a Web API controller action. Here is the section of JavaScript code that sends the AJAX request. In the sample app, all of the JavaScript app code is located in the Scripts\app.js file.

```
// If we already have a bearer token, set the Authorization header.
var token = sessionStorage.getItem(tokenKey);
var headers = {};
if (token) {
    headers.Authorization = 'Bearer ' + token;
}

$.ajax({
    type: 'GET',
    url: 'api/values/1',
    headers: headers
}).done(function (data) {
    self.result(data);
}).fail(showError);
```

Until the user logs in, there is no bearer token, and therefore no Authorization header in the request. This causes the request to return a 401 error.

Here is the HTTP request. (I used Fiddler to capture the HTTP traffic.)

```
GET https://localhost:44305/api/values HTTP/1.1
Host: localhost:44305
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:32.0) Gecko/20100101 Firefox/32.0
Accept: */*
Accept-Language: en-US,en;q=0.5
X-Requested-With: XMLHttpRequest
Referer: https://localhost:44305/
```

HTTP response:

```
HTTP/1.1 401 Unauthorized
Content-Type: application/json; charset=utf-8
Server: Microsoft-IIS/8.0
WWW-Authenticate: Bearer
Date: Tue, 30 Sep 2014 21:54:43 GMT
Content-Length: 61

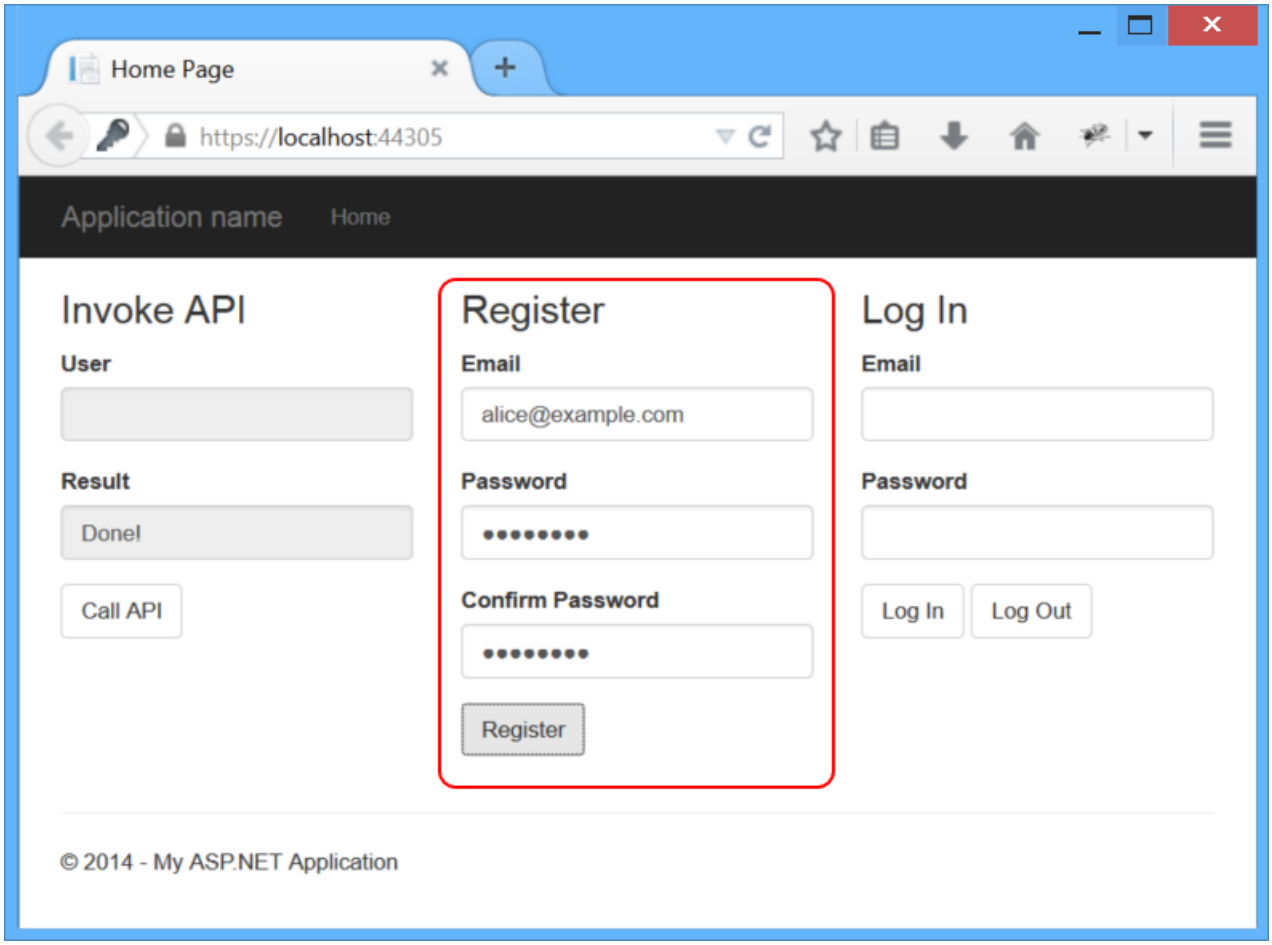
{"Message":"Authorization has been denied for this request."}
```

Notice that the response includes a Www-Authenticate header with the challenge set to Bearer. That indicates the server expects a bearer token.

Register a User

In the **Register** section of the app, enter an email and password, and click the **Register** button.

You don't need to use a valid email address for this sample, but a real app would confirm the address. (See Create a secure ASP.NET MVC 5 web app with log in, email confirmation and password reset.) For the password, use something like "Password1!", with an upper case letter, lower case letter, number, and non-alpha-numeric character. To keep the app simple, I left out client-side validation, so if there is a problem with the password format, you'll get a 400 (Bad Request) error.



The **Register** button sends a POST request to ~/api/Account/Register/. The request body is a JSON object that holds the name and password. Here is the JavaScript code that sends the request:

```
var data = {
    Email: self.registerEmail(),
    Password: self.registerPassword(),
    ConfirmPassword: self.registerPassword2()
};

$.ajax({
    type: 'POST',
    url: '/api/Account/Register',
    contentType: 'application/json; charset=utf-8',
    data: JSON.stringify(data)
}).done(function (data) {
    self.result("Done!");
}).fail(showError);
```

HTTP request:

```
POST https://localhost:44305/api/Account/Register HTTP/1.1
Host: localhost:44305
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:32.0) Gecko/20100101 Firefox/32.0
Accept: */*
Content-Type: application/json; charset=utf-8
X-Requested-With: XMLHttpRequest
Referer: https://localhost:44305/
Content-Length: 84

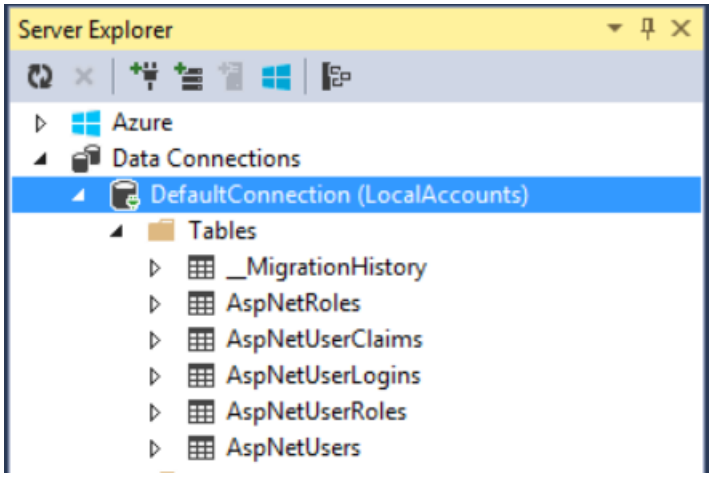
{"Email":"alice@example.com","Password":"Password1!","ConfirmPassword":"Password1!"}
```

HTTP response:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/8.0
Date: Wed, 01 Oct 2014 00:57:58 GMT
Content-Length: 0
```

This request is handled by the `AccountController` class. Internally, `AccountController` uses ASP.NET Identity to manage the membership database.

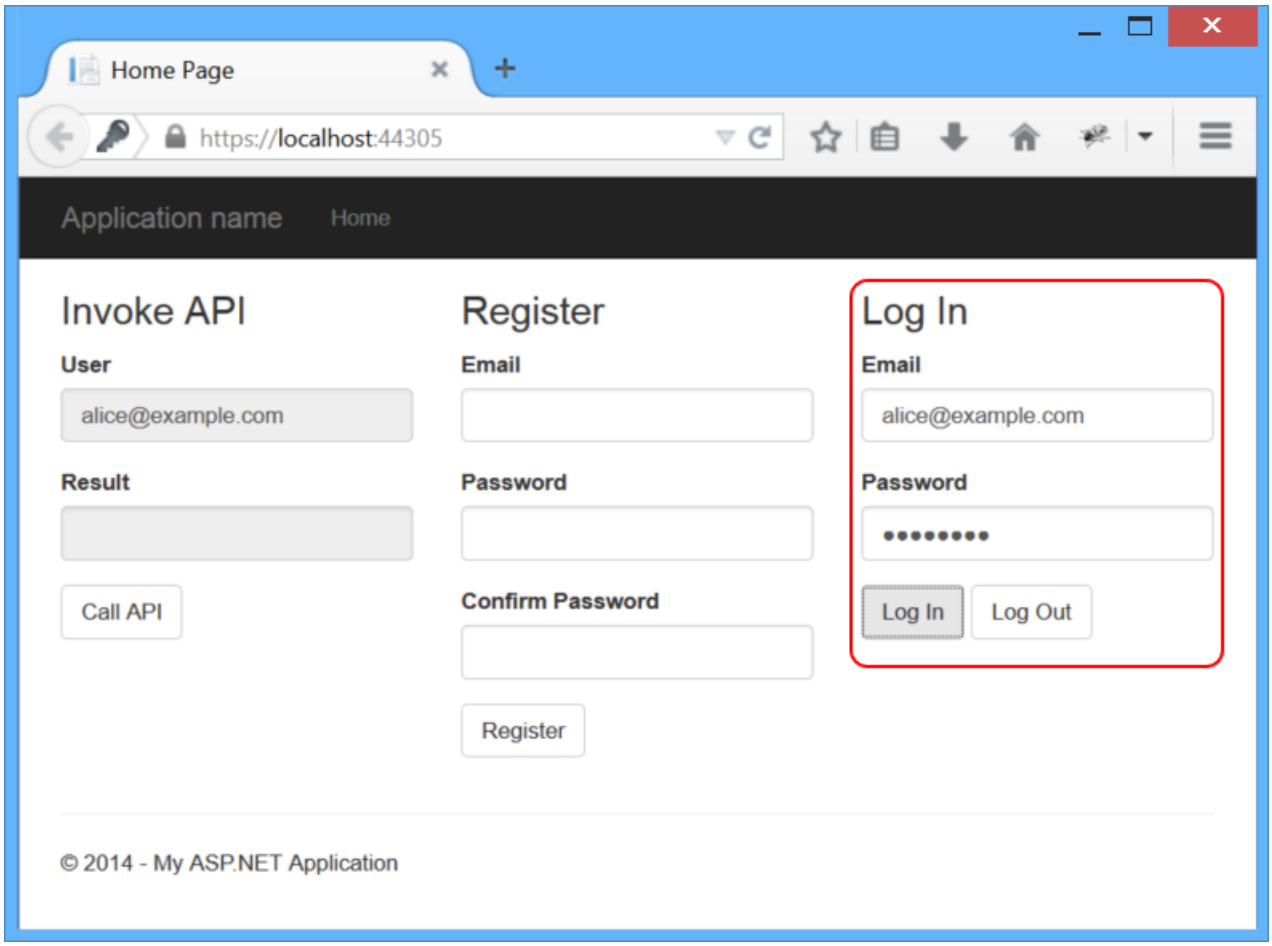
If you run the app locally from Visual Studio, user accounts are stored in LocalDB, in the AspNetUsers table. To view the tables in Visual Studio, click the **View** menu, select **Server Explorer**, then expand **Data Connections**.



Get an Access Token

So far we have not done any OAuth, but now we'll see the OAuth authorization server in action, when we request

an access token. In the **Log In** area of the sample app, enter the email and password and click **Log In**.



The **Log In** button sends a request to the token endpoint. The body of the request contains the following form-url-encoded data:

- grant_type: "password"
- username: <the user's email>
- password: <password>

Here is the JavaScript code that sends the AJAX request:

```
var loginData = {
    grant_type: 'password',
    username: self.loginEmail(),
    password: self.loginPassword()
};

$.ajax({
    type: 'POST',
    url: '/Token',
    data: loginData
}).done(function (data) {
    self.user(data.userName);
    // Cache the access token in session storage.
    sessionStorage.setItem(tokenKey, data.access_token);
}).fail(showError);
```

If the request succeeds, the authorization server returns an access token in the response body. Notice that we store the token in session storage, to use later when sending requests to the API. Unlike some forms of authentication (such as cookie-based authentication), the browser will not automatically include the access token in subsequent requests. The application must do so explicitly. That's a good thing, because it limits CSRF vulnerabilities.

HTTP request:

```
POST https://localhost:44305/Token HTTP/1.1
Host: localhost:44305
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:32.0) Gecko/20100101 Firefox/32.0
Accept: */*
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Referer: https://localhost:44305/
Content-Length: 68

grant_type=password&username=alice%40example.com&password=Password1!
```

You can see that the request contains the user's credentials. You *must* use HTTPS to provide transport layer security.

HTTP response:

```
HTTP/1.1 200 OK
Content-Length: 669
Content-Type: application/json;charset=UTF-8
Server: Microsoft-IIS/8.0
Date: Wed, 01 Oct 2014 01:22:36 GMT

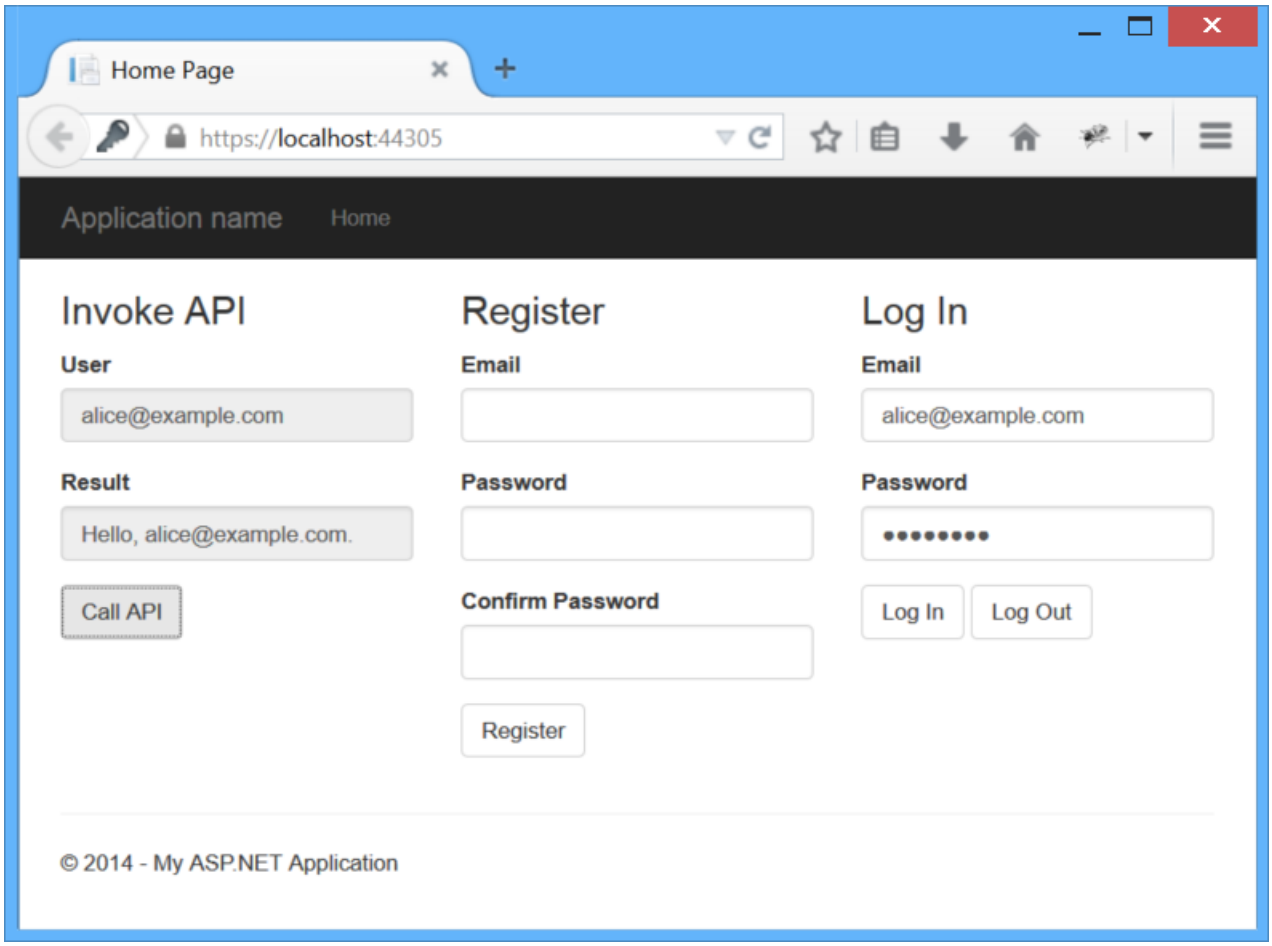
{
  "access_token":"imSXTs2OqSrGWzsFQhIXziFCO3rF...",
  "token_type":"bearer",
  "expires_in":1209599,
  "userName":"alice@example.com",
  ".issued":"Wed, 01 Oct 2014 01:22:33 GMT",
  ".expires":"Wed, 15 Oct 2014 01:22:33 GMT"
}
```

For readability, I indented the JSON and truncated the access token, which is a quite long.

The `access_token`, `token_type`, and `expires_in` properties are defined by the OAuth2 spec. The other properties (`userName`, `.issued`, and `.expires`) are just for informational purposes. You can find the code that adds those additional properties in the `TokenEndpoint` method, in the /Providers/ApplicationOAuthProvider.cs file.

Send an Authenticated Request

Now that we have a bearer token, we can make an authenticated request to the API. This is done by setting the Authorization header in the request. Click the **Call API** button again to see this.



HTTP request:

```
GET https://localhost:44305/api/values/1 HTTP/1.1
Host: localhost:44305
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:32.0) Gecko/20100101 Firefox/32.0
Accept: */*
Authorization: Bearer imSXTs2OqSrGWzsFQhIXziFCO3rF...
X-Requested-With: XMLHttpRequest
```

HTTP response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Server: Microsoft-IIS/8.0
Date: Wed, 01 Oct 2014 01:41:29 GMT
Content-Length: 27

"Hello, alice@example.com."
```

Log Out

Because the browser does not cache the credentials or the access token, logging out is simply a matter of "forgetting" the token, by removing it from session storage:

```
self.logout = function () {
    sessionStorage.removeItem(tokenKey)
}
```

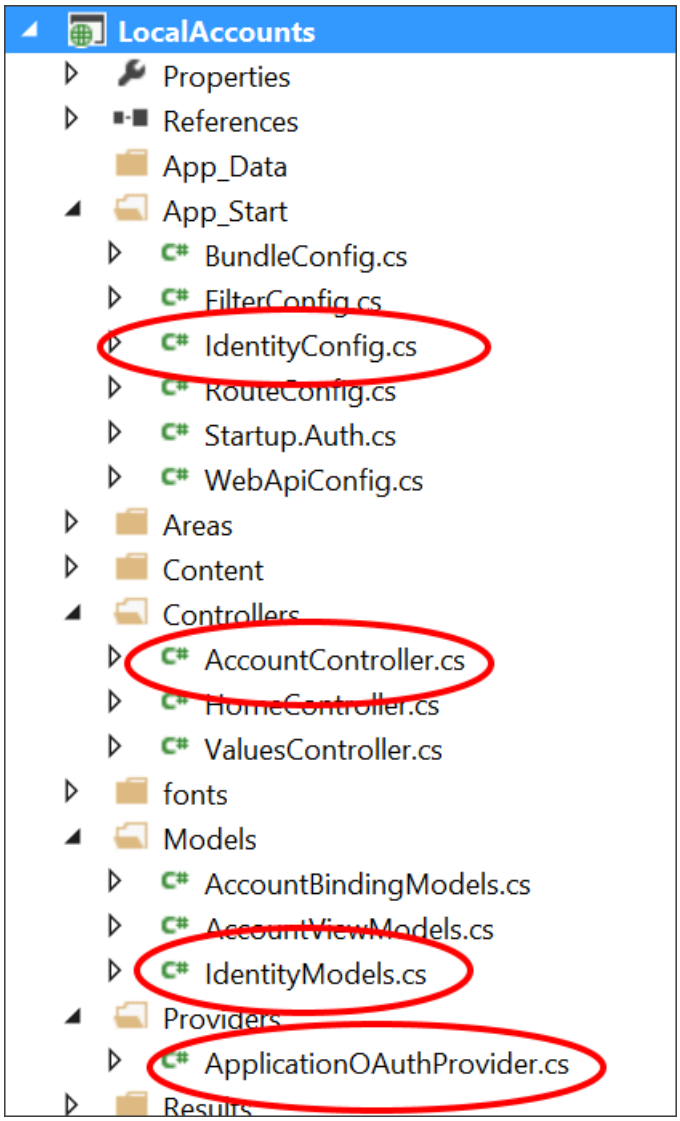
Understanding the Individual Accounts Project Template

When you select **Individual Accounts** in the ASP.NET Web Application project template, the project includes:

- An OAuth2 authorization server.
- A Web API endpoint for managing user accounts
- An EF model for storing user accounts.

Here are the main application classes that implement these features:

- `AccountController`. Provides a Web API endpoint for managing user accounts. The `Register` action is the only one that we used in this tutorial. Other methods on the class support password reset, social logins, and other functionality.
- `ApplicationUser`, defined in /Models/IdentityModels.cs. This class is the EF model for user accounts in the membership database.
- `ApplicationUserManager`, defined in /App_Start/IdentityConfig.cs This class derives from UserManager and performs operations on user accounts, such as creating a new user, verifying passwords, and so forth, and automatically persists changes to the database.
- `ApplicationOAuthProvider`. This object plugs into the OWIN middleware, and processes events raised by the middleware. It derives from OAuthAuthorizationServerProvider.



Configuring the Authorization Server

In StartupAuth.cs, the following code configures the OAuth2 authorization server.

```
PublicClientId = "self";
OAuthOptions = new OAuthAuthorizationServerOptions
{
    TokenEndpointPath = new PathString("/Token"),
    Provider = new ApplicationOAuthProvider(PublicClientId),
    AuthorizeEndpointPath = new PathString("/api/Account/ExternalLogin"),
    AccessTokenExpireTimeSpan = TimeSpan.FromDays(14),
    // Note: Remove the following line before you deploy to production:
    AllowInsecureHttp = true
};

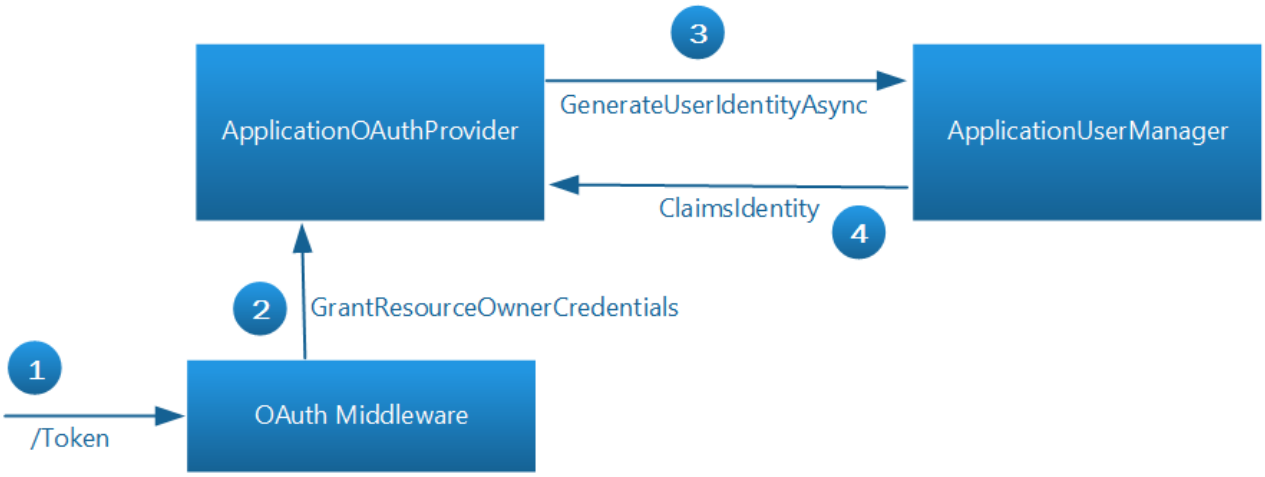
// Enable the application to use bearer tokens to authenticate users
app.UseOAuthBearerTokens(OAuthOptions);
```

The `TokenEndpointPath` property is the URL path to the authorization server endpoint. That's the URL that app uses to get the bearer tokens.

The `Provider` property specifies a provider that plugs into the OWIN middleware, and processes events raised by the middleware.

Here is the basic flow when the app wants to get a token:

1. To get an access token, the app sends a request to ~/Token.
2. The OAuth middleware calls `GrantResourceOwnerCredentials` on the provider.
3. The provider calls the `ApplicationUserManager` to validate the credentials and create a claims identity.
4. If that succeeds, the provider creates an authentication ticket, which is used to generate the token.



The OAuth middleware doesn't know anything about the user accounts. The provider communicates between the middleware and ASP.NET Identity. For more information about implementing the authorization server, see OWIN OAuth 2.0 Authorization Server.

Configuring Web API to use Bearer Tokens

In the `WebApiConfig.Register` method, the following code sets up authentication for the Web API pipeline:

```
config.SuppressDefaultHostAuthentication();
config.Filters.Add(new HostAuthenticationFilter(OAuthDefaults.AuthenticationType));
```

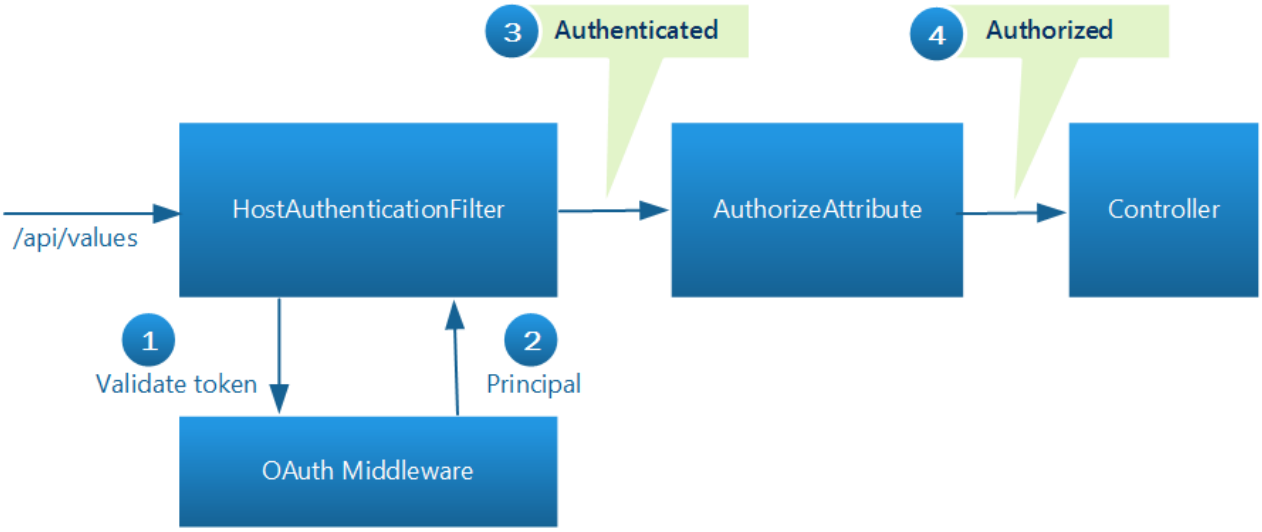
The **HostAuthenticationFilter** class enables authentication using bearer tokens.

The **SuppressDefaultHostAuthentication** method tells Web API to ignore any authentication that happens before the request reaches the Web API pipeline, either by IIS or by OWIN middleware. That way, we can restrict Web API to authenticate only using bearer tokens.

> **NOTE**
> In particular, the MVC portion of your app might use forms authentication, which stores credentials in a cookie. Cookie-based authentication requires the use of anti-forgery tokens, to prevent CSRF attacks. That's a problem for web APIs, because there is no convenient way for the web API to send the anti-forgery token to the client. (For more background on this issue, see Preventing CSRF Attacks in Web API.) Calling **SuppressDefaultHostAuthentication** ensures that Web API is not vulnerable to CSRF attacks from credentials stored in cookies.

When the client requests a protected resource, here is what happens in the Web API pipeline:

1. The **HostAuthentication** filter calls the OAuth middleware to validate the token.
2. The middleware converts the token into a claims identity.
3. At this point, the request is *authenticated* but not *authorized*.
4. The authorization filter examines the claims identity. If the claims authorize the user for that resource, the request is authorized. By default, the **[Authorize]** attribute will authorize any request that is authenticated. However, you can authorize by role or by other claims. For more information, see Authentication and Authorization in Web API.
5. If the previous steps are successful, the controller returns the protected resource. Otherwise, the client receives a 401 (Unauthorized) error.



Additional Resources

- ASP.NET Identity
- Understanding Security Features in the SPA Template for VS2013 RC. MSDN blog post by Hongye Sun.
- Dissecting the Web API Individual Accounts Template–Part 2: Local Accounts. Blog post by Dominick Baier.
- Host authentication and Web API with OWIN. A good explanation of `SuppressDefaultHostAuthentication` and `HostAuthenticationFilter` by Brock Allen.
- Customizing profile information in ASP.NET Identity in VS 2013 templates. MSDN blog post by Pranav Rastogi.
- Per request lifetime management for UserManager class in ASP.NET Identity. MSDN blog post by Suhas Joshi, with a good explanation of the `UserManager` class.

External Authentication Services with ASP.NET Web API (C#)

2/20/2020 • 16 minutes to read • Edit Online

Visual Studio 2017 and ASP.NET 4.7.2 expand the security options for Single Page Applications (SPA) and Web API services to integrate with external authentication services, which include several OAuth/OpenID and social media authentication services: Microsoft Accounts, Twitter, Facebook, and Google.

In this Walkthrough

- Using External Authentication Services
- Creating the Sample Web Application
- Enabling Facebook Authentication
- Enabling Google Authentication
- Enabling Microsoft Authentication
- Enabling Twitter Authentication
- Additional Information
  - Combining External Authentication Services
  - Configuring IIS Express to use a Fully Qualified Domain Name
  - How to Obtain your Application Settings for Microsoft Authentication
  - Optional: Disable Local Registration

Prerequisites

To follow the examples in this walkthrough, you need to have the following:

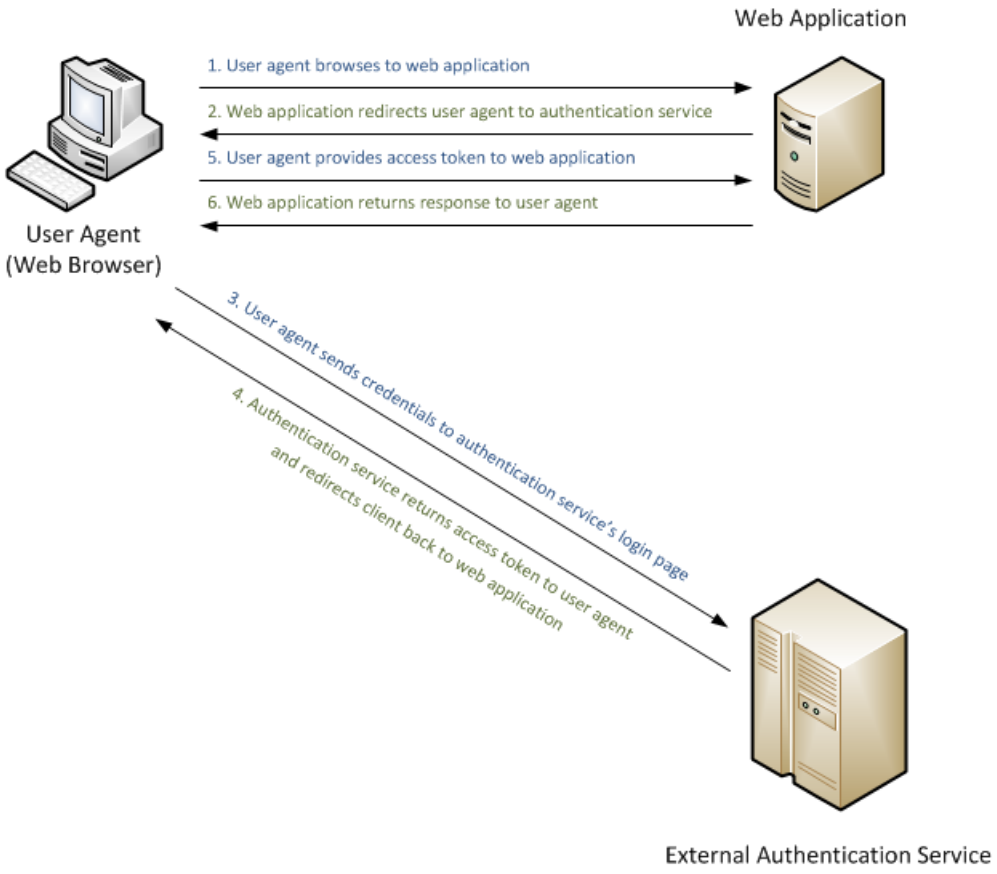
- Visual Studio 2017
- A developer account with the application identifier and secret key for one of the following social media authentication services:
  - Microsoft Accounts (https://go.microsoft.com/fwlink/?LinkID=144070)
  - Twitter (https://dev.twitter.com/)
  - Facebook (https://developers.facebook.com/)
  - Google (https://developers.google.com/)

Using External Authentication Services

The abundance of external authentication services that are currently available to web developers help to reduce development time when creating new web applications. Web users typically have several existing accounts for popular web services and social media websites, therefore when a web application implements the authentication services from an external web service or social media website, it saves the development time that would have been spent creating an authentication implementation. Using an external authentication service saves end users from having to create another account for your web application, and also from having to remember another username and password.

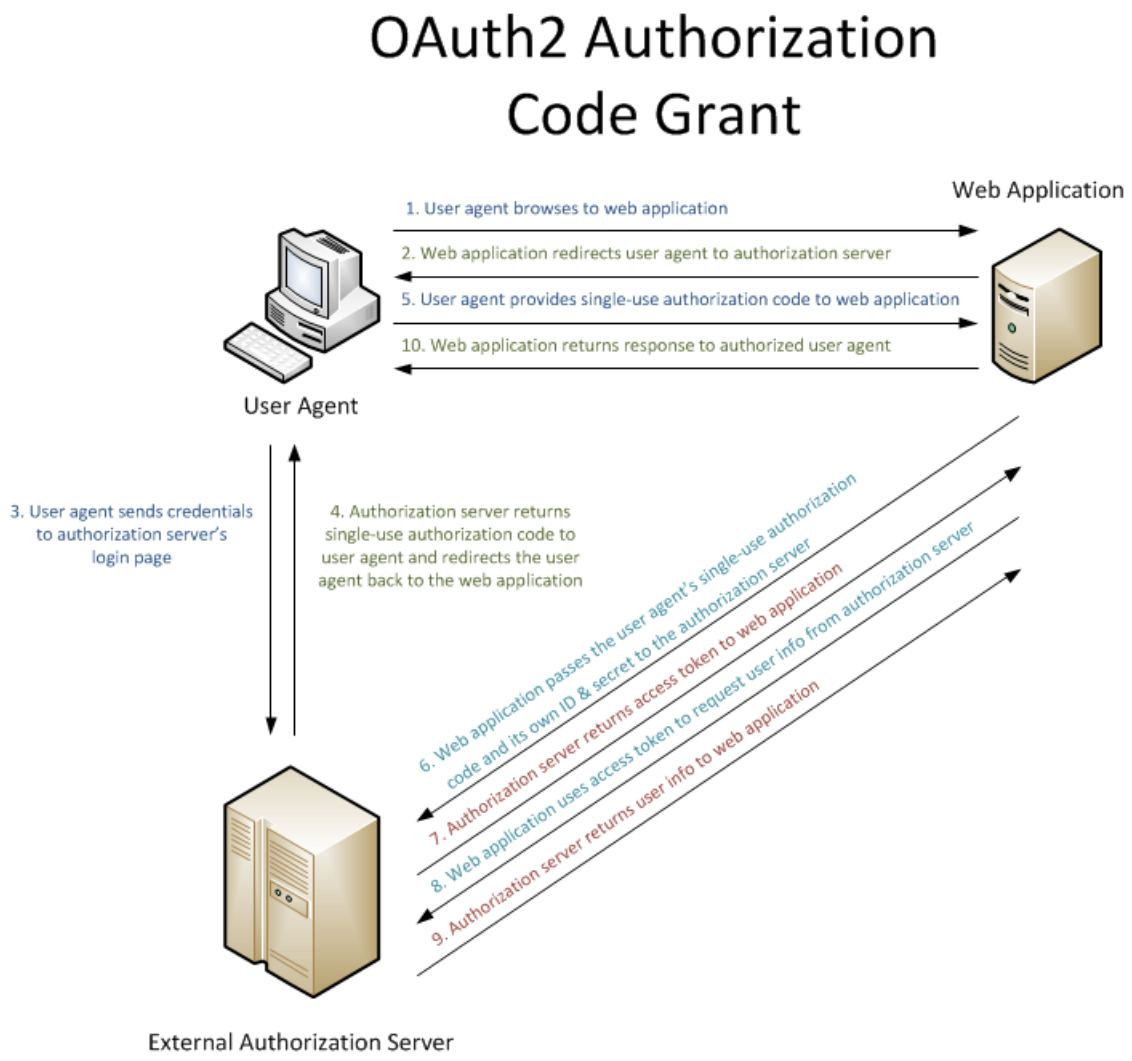
In the past, developers have had two choices: create their own authentication implementation, or learn how to

integrate an external authentication service into their applications. At the most basic level, the following diagram illustrates a simple request flow for a user agent (web browser) that is requesting information from a web application that is configured to use an external authentication service:



In the preceding diagram, the user agent (or web browser in this example) makes a request to a web application, which redirects the web browser to an external authentication service. The user agent sends its credentials to the external authentication service, and if the user agent has successfully authenticated, the external authentication service will redirect the user agent to the original web application with some form of token which the user agent will send to the web application. The web application will use the token to verify that the user agent has been successfully authenticated by the external authentication service, and the web application may use the token to gather more information about the user agent. Once the application is done processing the user agent's information, the web application will return the appropriate response to the user agent based on its authorization settings.

In this second example, the user agent negotiates with the web application and external authorization server, and the web application performs additional communication with the external authorization server to retrieve additional information about the user agent:



Visual Studio 2017 and ASP.NET 4.7.2 make integration with external authentication services easier for developers by providing built-in integration for the following authentication services:

- Facebook
- Google
- Microsoft Accounts (Windows Live ID accounts)
- Twitter

The examples in this walkthrough will demonstrate how to configure each of the supported external authentication services by using the new ASP.NET Web Application template that ships with Visual Studio 2017.

> **NOTE**
>
> If necessary, you may need to add your FQDN to the settings for your external authentication service. This requirement is based on security constraints for some external authentication services which require the FQDN in your application settings to match the FQDN that is used by your clients. (The steps for this will vary greatly for each external authentication service; you will need to consult the documentation for each external authentication service to see if this is required and how to configure these settings.) If you need to configure IIS Express to use an FQDN for testing this environment, see the Configuring IIS Express to use a Fully Qualified Domain Name section later in this walkthrough.

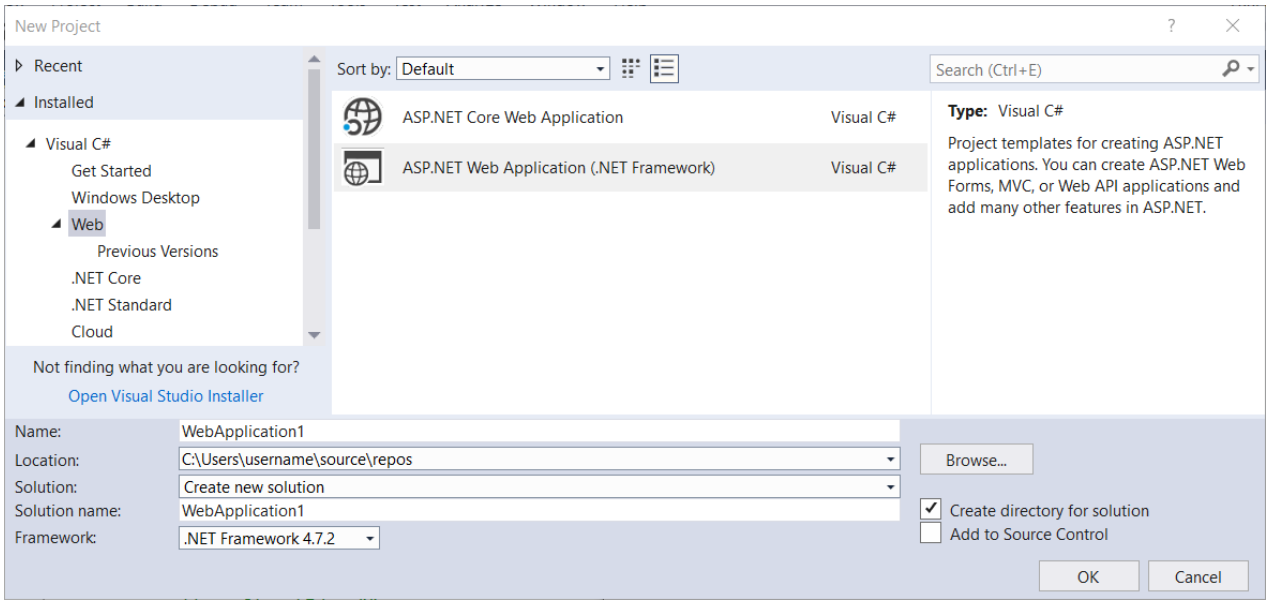
Create a Sample Web Application

The following steps will lead you through creating a sample application by using the ASP.NET Web Application template, and you will use this sample application for each of the external authentication services later in this

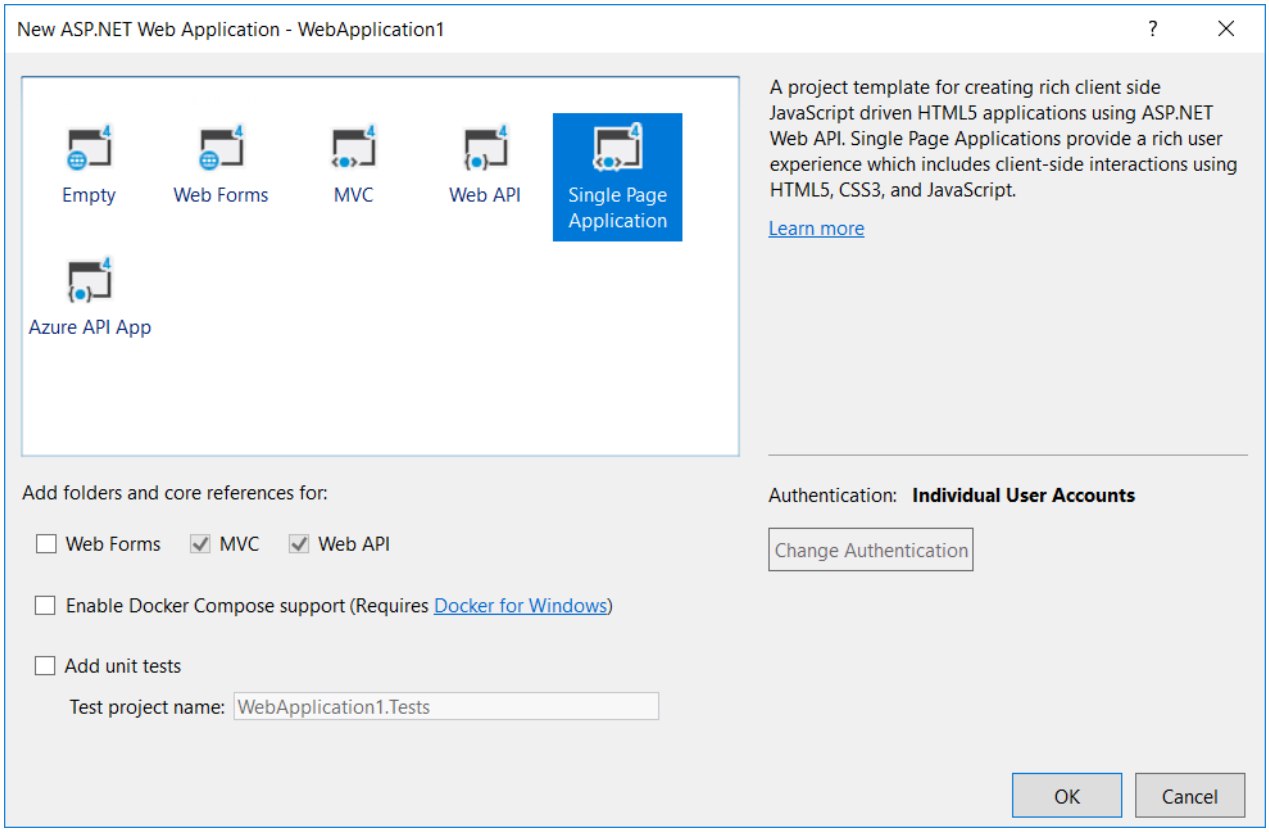
walkthrough.

Start Visual Studio 2017 and select **New Project** from the Start page. Or, from the **File** menu, select **New** and then **Project**.

When the **New Project** dialog box is displayed, select **Installed** and expand **Visual C#**. Under **Visual C#**, select **Web**. In the list of project templates, select **ASP.NET Web Application (.Net Framework)**. Enter a name for your project and click **OK**.



When the **New ASP.NET Project** is displayed, select the **Single Page Application** template and click **Create Project**.



Wait as Visual Studio 2017 creates your project.

When Visual Studio 2017 has finished creating your project, open the *Startup.Auth.cs* file that is located in the **App_Start** folder.

When you first create the project, none of the external authentication services are enabled in *Startup.Auth.cs* file;

the following illustrates what your code might resemble, with the sections highlighted for where you would enable an external authentication service and any relevant settings in order to use Microsoft Accounts, Twitter, Facebook, or Google authentication with your ASP.NET application:

```
using System;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.DataProtection;
using Microsoft.Owin.Security.Google;
using Microsoft.Owin.Security.OAuth;
using Owin;
using WebApplication1.Models;
using WebApplication1.Providers;

namespace WebApplication1
{
    public partial class Startup
    {
        // Enable the application to use OAuthAuthorization. You can then secure your Web APIs
        static Startup()
        {
            PublicClientId = "web";

            OAuthOptions = new OAuthAuthorizationServerOptions
            {
                TokenEndpointPath = new PathString("/Token"),
                AuthorizeEndpointPath = new PathString("/Account/Authorize"),
                Provider = new ApplicationOAuthProvider(PublicClientId),
                AccessTokenExpireTimeSpan = TimeSpan.FromDays(14),
                AllowInsecureHttp = true
            };
        }

        public static OAuthAuthorizationServerOptions OAuthOptions { get; private set; }

        public static string PublicClientId { get; private set; }

        // For more information on configuring authentication, please visit https://go.microsoft.com/fwlink/?
LinkId=301864
        public void ConfigureAuth(IAppBuilder app)
        {
            // Configure the db context, user manager and signin manager to use a single instance per request
            app.CreatePerOwinContext(ApplicationDbContext.Create);
            app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Create);
            app.CreatePerOwinContext<ApplicationSignInManager>(ApplicationSignInManager.Create);

            // Enable the application to use a cookie to store information for the signed in user
            app.UseCookieAuthentication(new CookieAuthenticationOptions
            {
                AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
                LoginPath = new PathString("/Account/Login"),
                Provider = new CookieAuthenticationProvider
                {
                    // Enables the application to validate the security stamp when the user logs in.
                    // This is a security feature which is used when you change a password or add an external
login to your account.
                    OnValidateIdentity = SecurityStampValidator.OnValidateIdentity<ApplicationUserManager,
ApplicationUser>(
                        validateInterval: TimeSpan.FromMinutes(20),
                        regenerateIdentity: (manager, user) => user.GenerateUserIdentityAsync(manager))
                }
            });
            // Use a cookie to temporarily store information about a user logging in with a third party login
provider
```

```
            app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);

            // Enables the application to temporarily store user information when they are verifying the
second factor in the two-factor authentication process.
            app.UseTwoFactorSignInCookie(DefaultAuthenticationTypes.TwoFactorCookie,
TimeSpan.FromMinutes(5));

            // Enables the application to remember the second login verification factor such as phone or
email.
            // Once you check this option, your second step of verification during the login process will be
remembered on the device where you logged in from.
            // This is similar to the RememberMe option when you log in.
            app.UseTwoFactorRememberBrowserCookie(DefaultAuthenticationTypes.TwoFactorRememberBrowserCookie);

            // Enable the application to use bearer tokens to authenticate users
            app.UseOAuthBearerTokens(OAuthOptions);

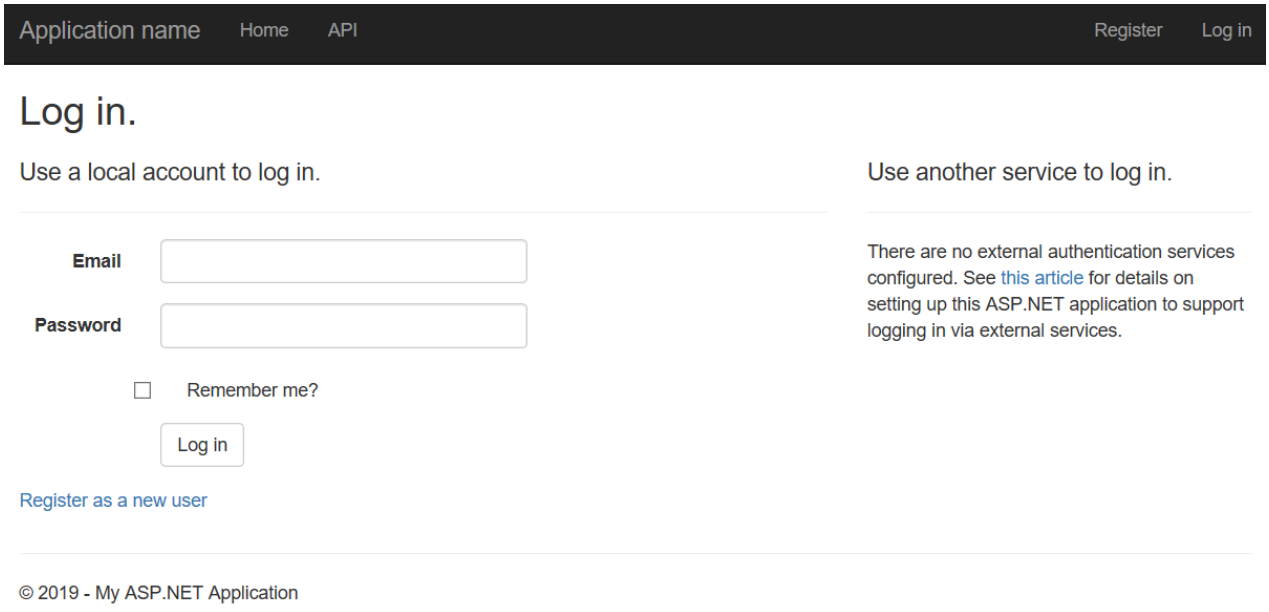
            // Uncomment the following lines to enable logging in with third party login providers
            //app.UseMicrosoftAccountAuthentication(
            //    clientId: "",
            //    clientSecret: "");

            //app.UseTwitterAuthentication(
            //    consumerKey: "",
            //    consumerSecret: "");

            //app.UseFacebookAuthentication(
            //    appId: "",
            //    appSecret: "");

            //app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
            //{
            //    ClientId = "",
            //    ClientSecret = ""
            //});
        }
    }
}
```

When you press F5 to build and debug your web application, it will display a login screen where you will see that no external authentication services have been defined.



In the following sections, you will learn how to enable each of the external authentication services that are

provided with ASP.NET in Visual Studio 2017.

Enabling Facebook authentication

Using Facebook authentication requires you to create a Facebook developer account, and your project will require an application ID and secret key from Facebook in order to function. For information about creating a Facebook developer account and obtaining your application ID and secret key, see https://go.microsoft.com/fwlink/?LinkID=252166.

Once you have obtained your application ID and secret key, use the following steps to enable Facebook authentication for your web application:

1. When your project is open in Visual Studio 2017, open the *Startup.Auth.cs* file.
2. Locate the Facebook authentication section of code:

```
// Uncomment the following lines to enable logging in with third party login providers
//app.UseMicrosoftAccountAuthentication(
//    clientId: "",
//    clientSecret: "");

//app.UseTwitterAuthentication(
//    consumerKey: "",
//    consumerSecret: "");

//app.UseFacebookAuthentication(
//    appId: "",
//    appSecret: "");

//app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
//{
//    ClientId = "",
//    ClientSecret = ""
//});
```

3. Remove the "//" characters to uncomment the highlighted lines of code, and then add your application ID and secret key. Once you have added those parameters, you can recompile your project:

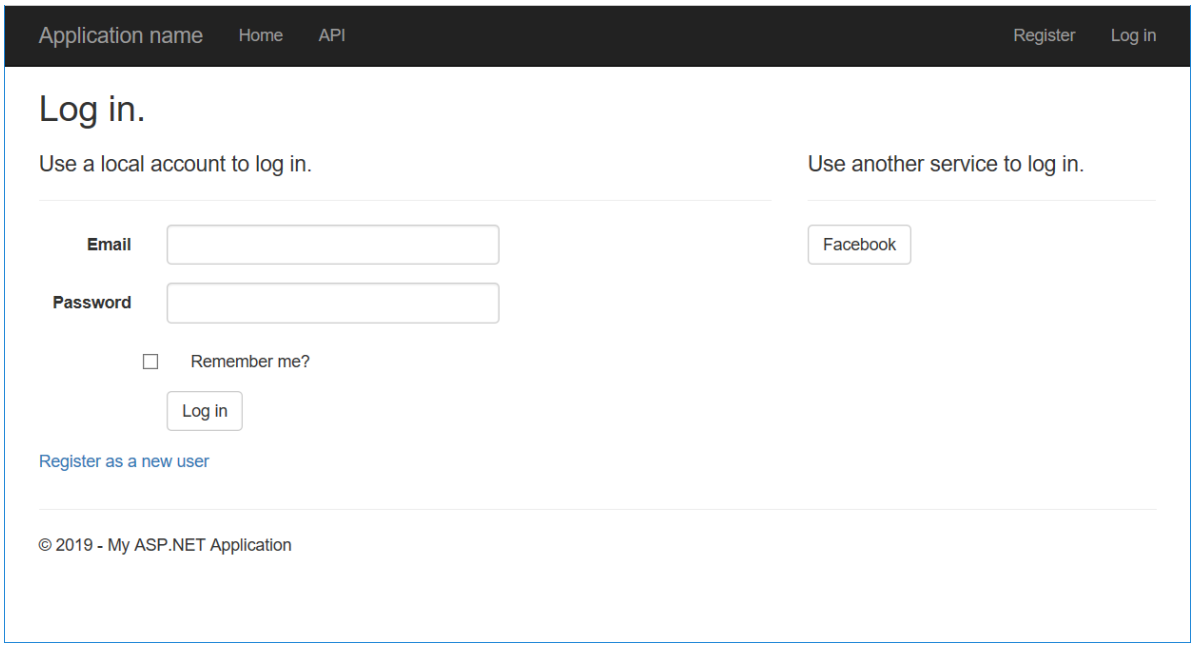
```
// Uncomment the following lines to enable logging in with third party login providers
//app.UseMicrosoftAccountAuthentication(
//    clientId: "",
//    clientSecret: "");

//app.UseTwitterAuthentication(
//   consumerKey: "",
//   consumerSecret: "");

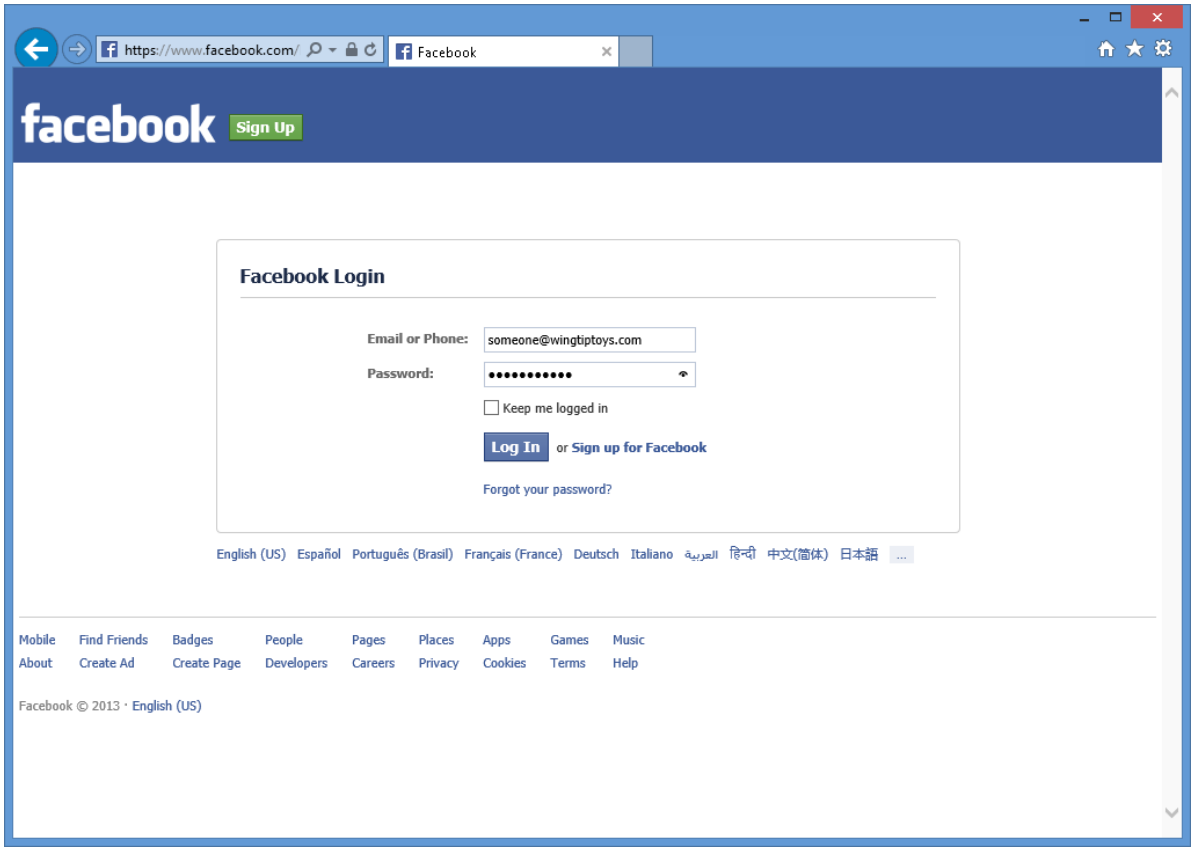
//app.UseFacebookAuthentication(
//   appId: "426f62526f636b73",
//   appSecret: "");

//app.UseGoogleAuthentication();
```

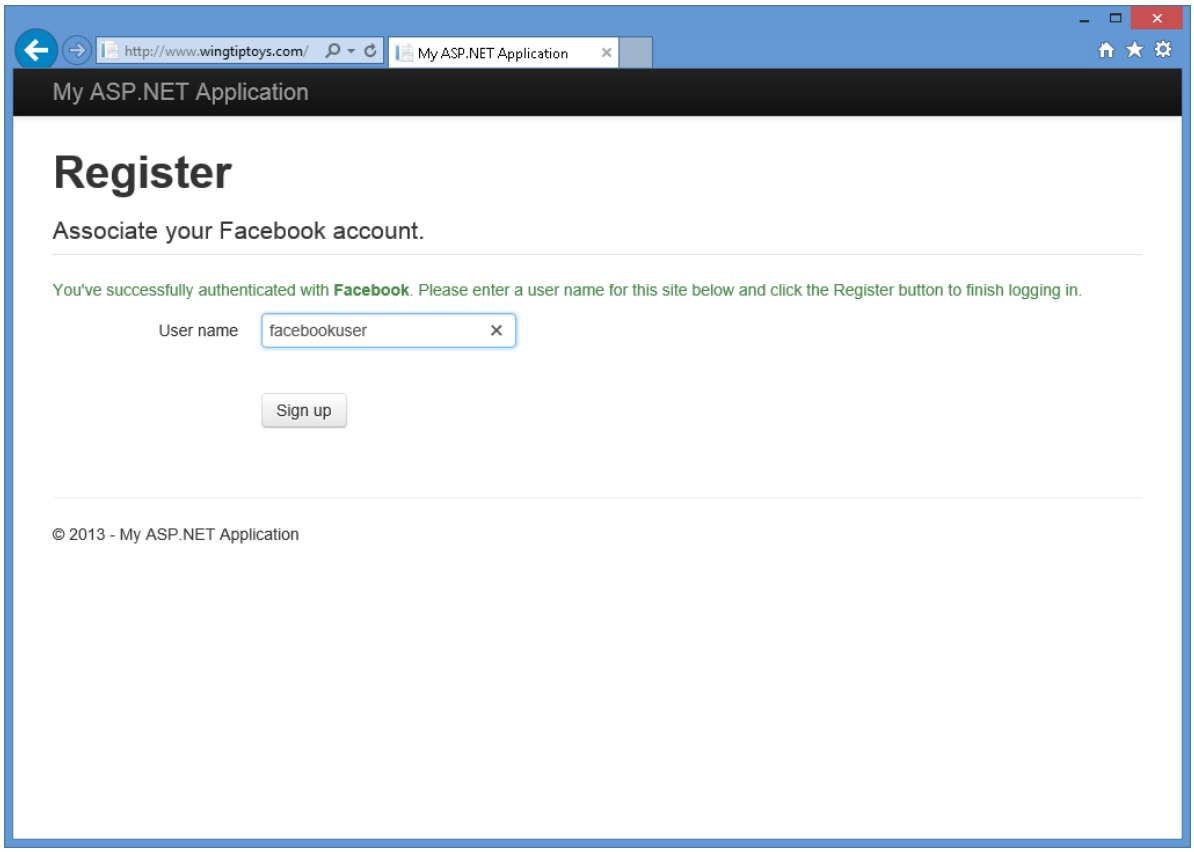
4. When you press F5 to open your web application in your web browser, you will see that Facebook has been defined as an external authentication service:



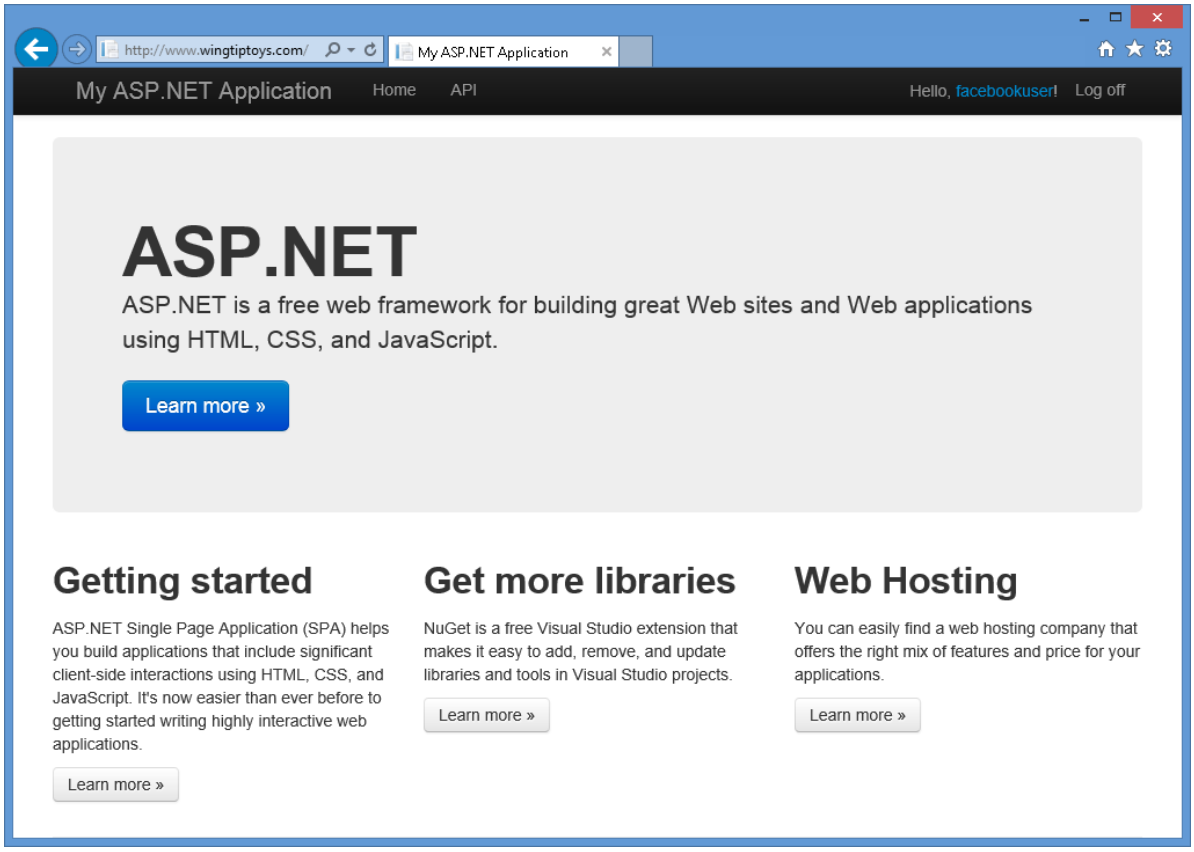
5. When you click the **Facebook** button, your browser will be redirected to the Facebook login page:



6. After you enter your Facebook credentials and click **Log in**, your web browser will be redirected back to your web application, which will prompt you for the **User name** that you want to associate with your Facebook account:



7. After you have entered your user name and clicked the **Sign up** button, your web application will display the default **home page** for your Facebook account:



Enabling Google Authentication

Using Google authentication requires you to create a Google developer account, and your project will require an application ID and secret key from Google in order to function. For information about creating a Google developer account and obtaining your application ID and secret key, see https://developers.google.com.

To enable Google authentication for your web application, use the following steps:

1. When your project is open in Visual Studio 2017, open the *Startup.Auth.cs* file.

2. Locate the Google authentication section of code:

```
// Uncomment the following lines to enable logging in with third party login providers
//app.UseMicrosoftAccountAuthentication(
//    clientId: "",
//    clientSecret: "");

//app.UseTwitterAuthentication(
//    consumerKey: "",
//    consumerSecret: "");

//app.UseFacebookAuthentication(
//    appId: "",
//    appSecret: "");

//app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
//{
//    ClientId = "",
//    ClientSecret = ""
//});
```

3. Remove the "//" characters to uncomment the highlighted lines of code, and then add your application ID and secret key. Once you have added those parameters, you can recompile your project:

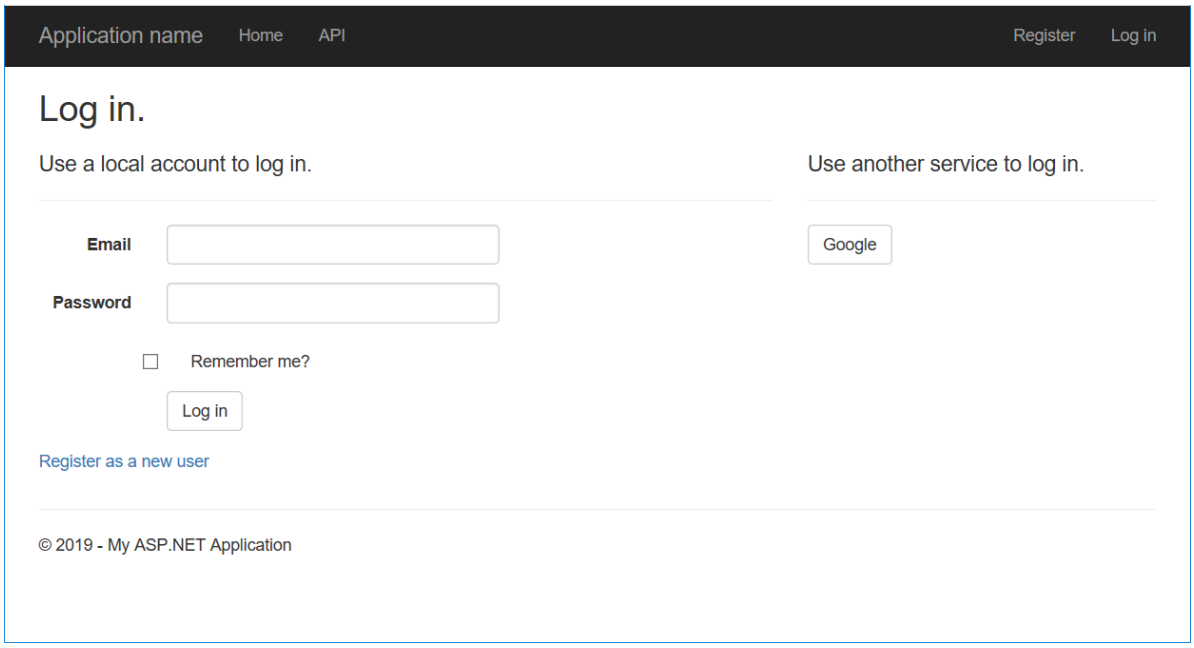
```
// Uncomment the following lines to enable logging in with third party login providers
//app.UseMicrosoftAccountAuthentication(
//    clientId: "",
//    clientSecret: "");

//app.UseTwitterAuthentication(
//   consumerKey: "",
//   consumerSecret: "");

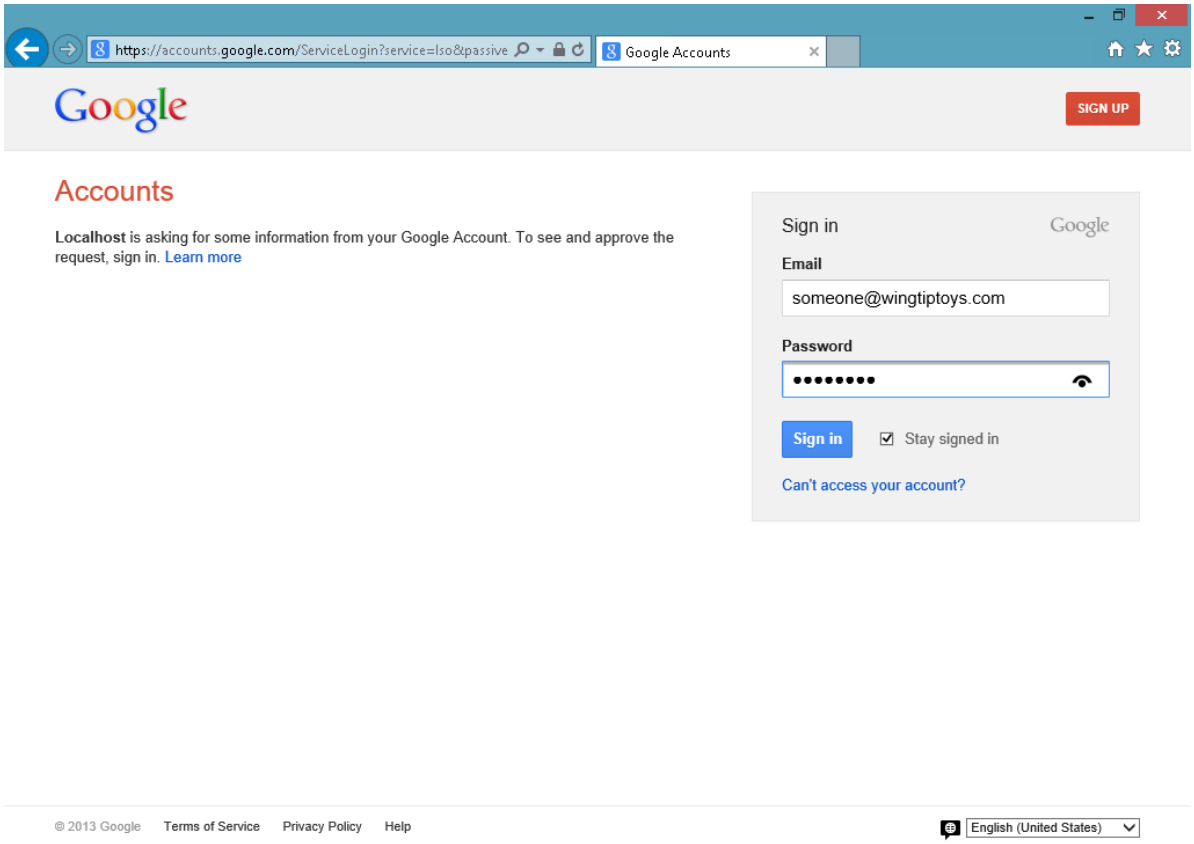
//app.UseFacebookAuthentication(
//   appId: "",
//   appSecret: "");

app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
{
    ClientId = "477522346600.apps.googleusercontent.com",
    ClientSecret = "gobkdpbocikdfbnfahjladnetpdkvmic"
});
```

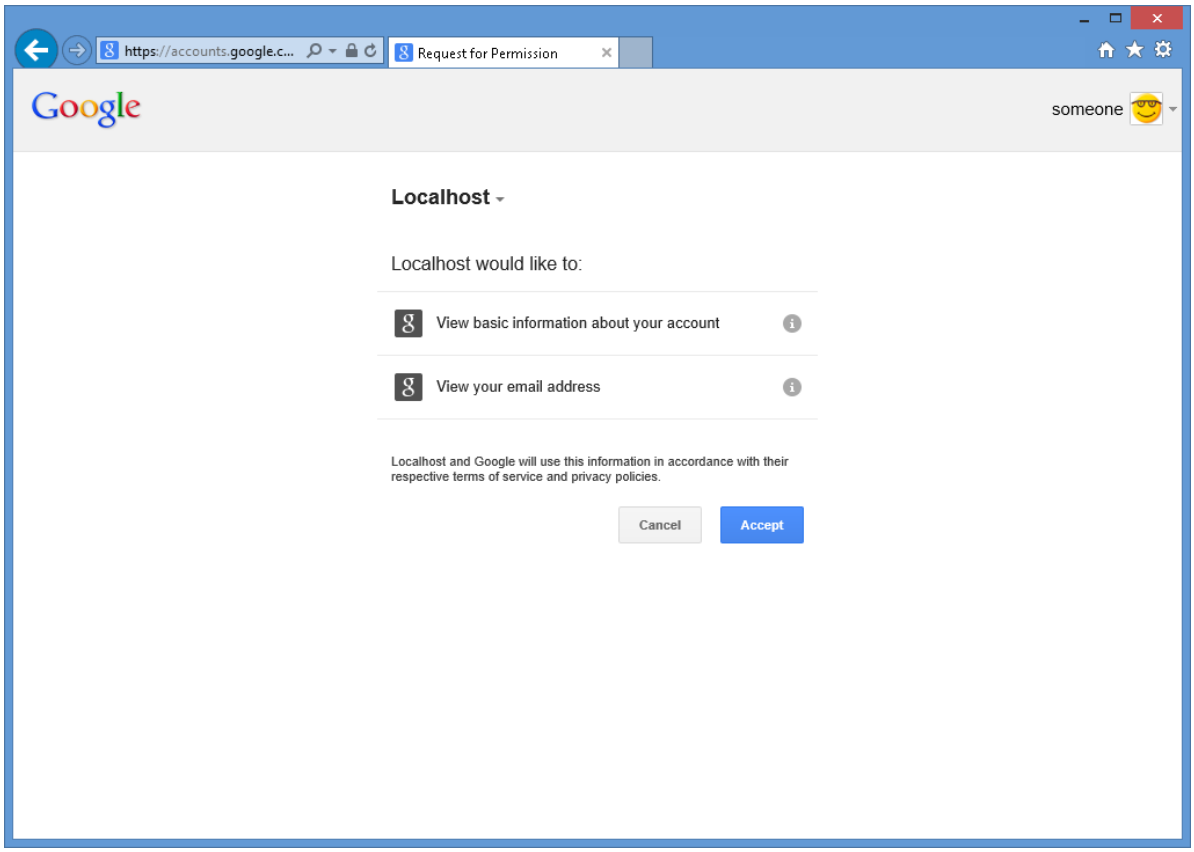
4. When you press F5 to open your web application in your web browser, you will see that Google has been defined as an external authentication service:



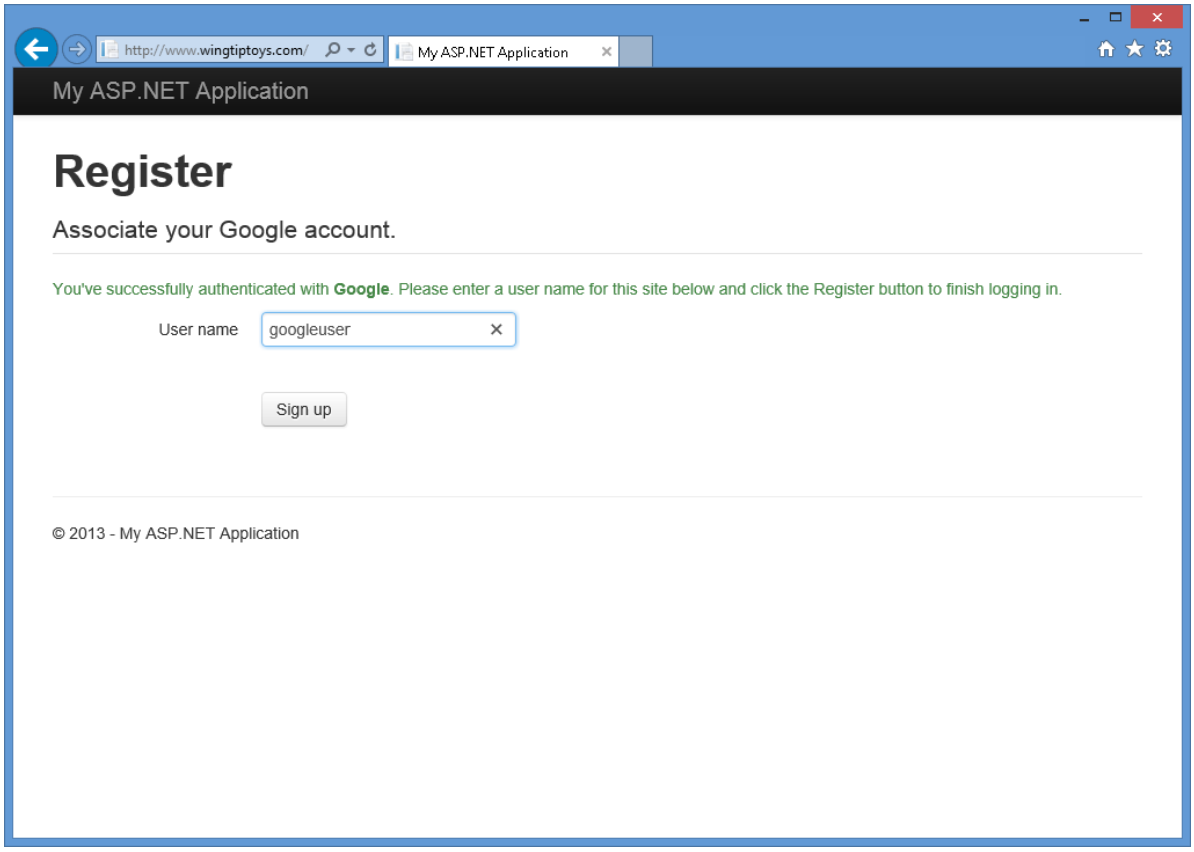
5. When you click the **Google** button, your browser will be redirected to the Google login page:



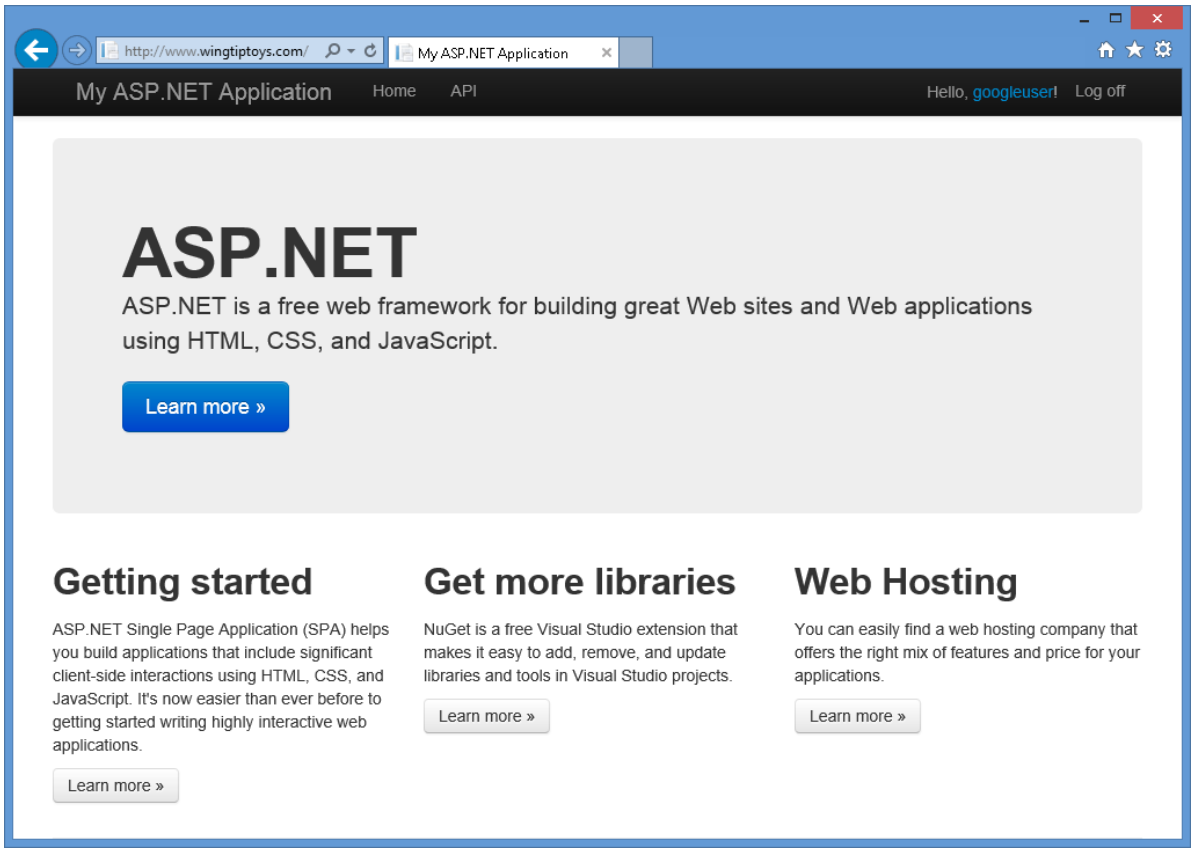
6. After you enter your Google credentials and click **Sign in**, Google will prompt you to verify that your web application has permissions to access your Google account:



7. When you click **Accept**, your web browser will be redirected back to your web application, which will prompt you for the **User name** that you want to associate with your Google account:



8. After you have entered your user name and clicked the **Sign up** button, your web application will display the default **home page** for your Google account:



Enabling Microsoft Authentication

Microsoft authentication requires you to create a developer account, and it requires a client ID and client secret in order to function. For information about creating a Microsoft developer account and obtaining your client ID and client secret, see https://go.microsoft.com/fwlink/?LinkID=144070.

Once you have obtained your consumer key and consumer secret, use the following steps to enable Microsoft authentication for your web application:

1. When your project is open in Visual Studio 2017, open the *Startup.Auth.cs* file.
2. Locate the Microsoft authentication section of code:

```
// Uncomment the following lines to enable logging in with third party login providers
//app.UseMicrosoftAccountAuthentication(
//    clientId: "",
//    clientSecret: "");

//app.UseTwitterAuthentication(
//   consumerKey: "",
//   consumerSecret: "");

//app.UseFacebookAuthentication(
//   appId: "",
//   appSecret: "");

//app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
//{
//    ClientId = "",
//    ClientSecret = ""
//});
```

3. Remove the "//" characters to uncomment the highlighted lines of code, and then add your client ID and client secret. Once you have added those parameters, you can recompile your project:

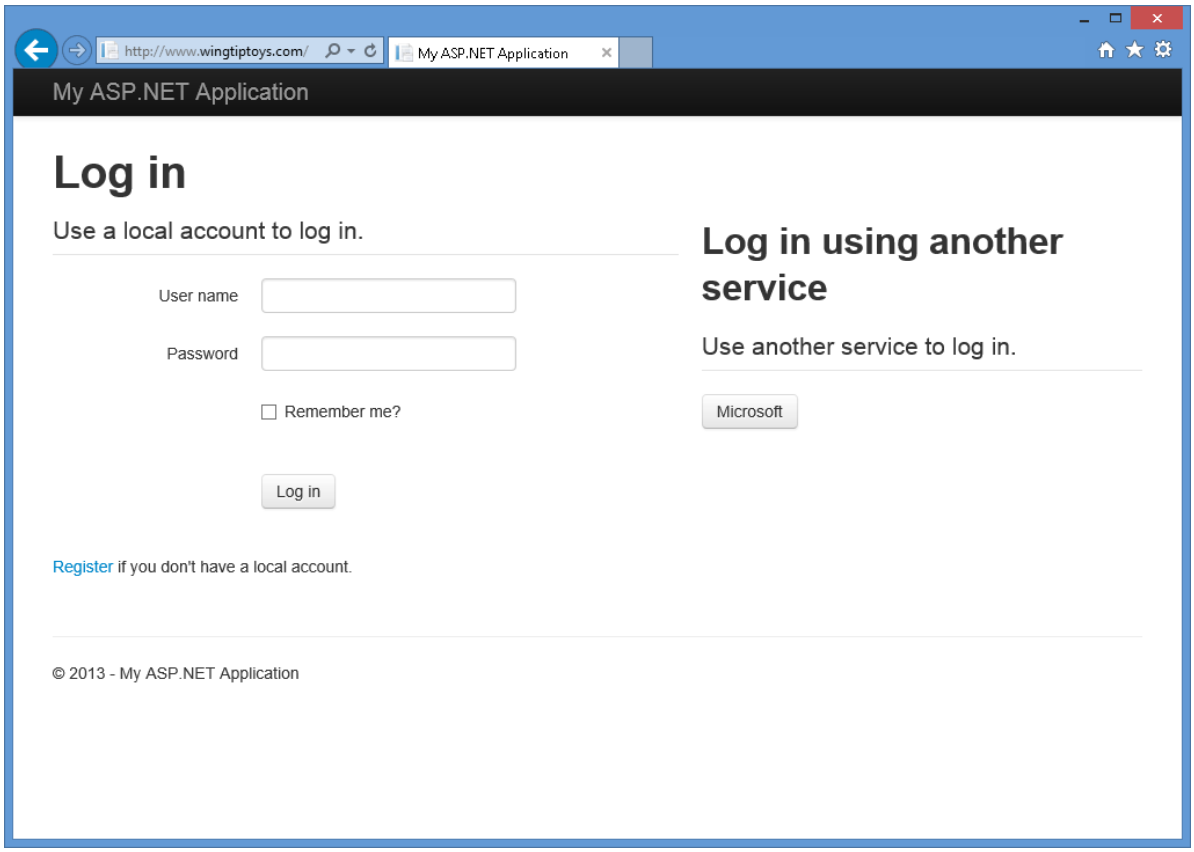
```
// Uncomment the following lines to enable logging in with third party login providers
app.UseMicrosoftAccountAuthentication(
    clientId: "426f62526f636b73",
    clientSecret: "57686f6120447564652c2049495320526f636b73");

//app.UseTwitterAuthentication(
//   consumerKey: "",
//   consumerSecret: "");

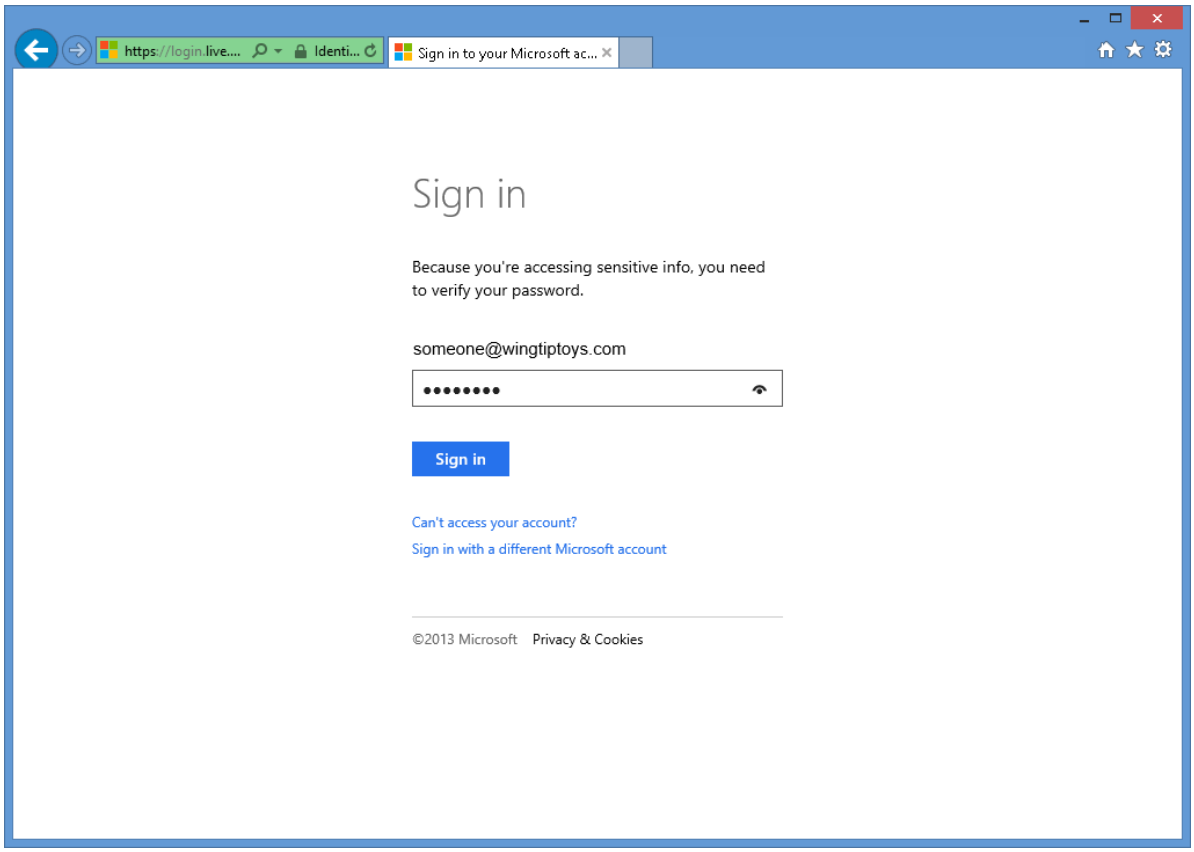
//app.UseFacebookAuthentication(
//   appId: "",
//   appSecret: "");

//app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
//{
//    ClientId = "",
//    ClientSecret = ""
//});
```

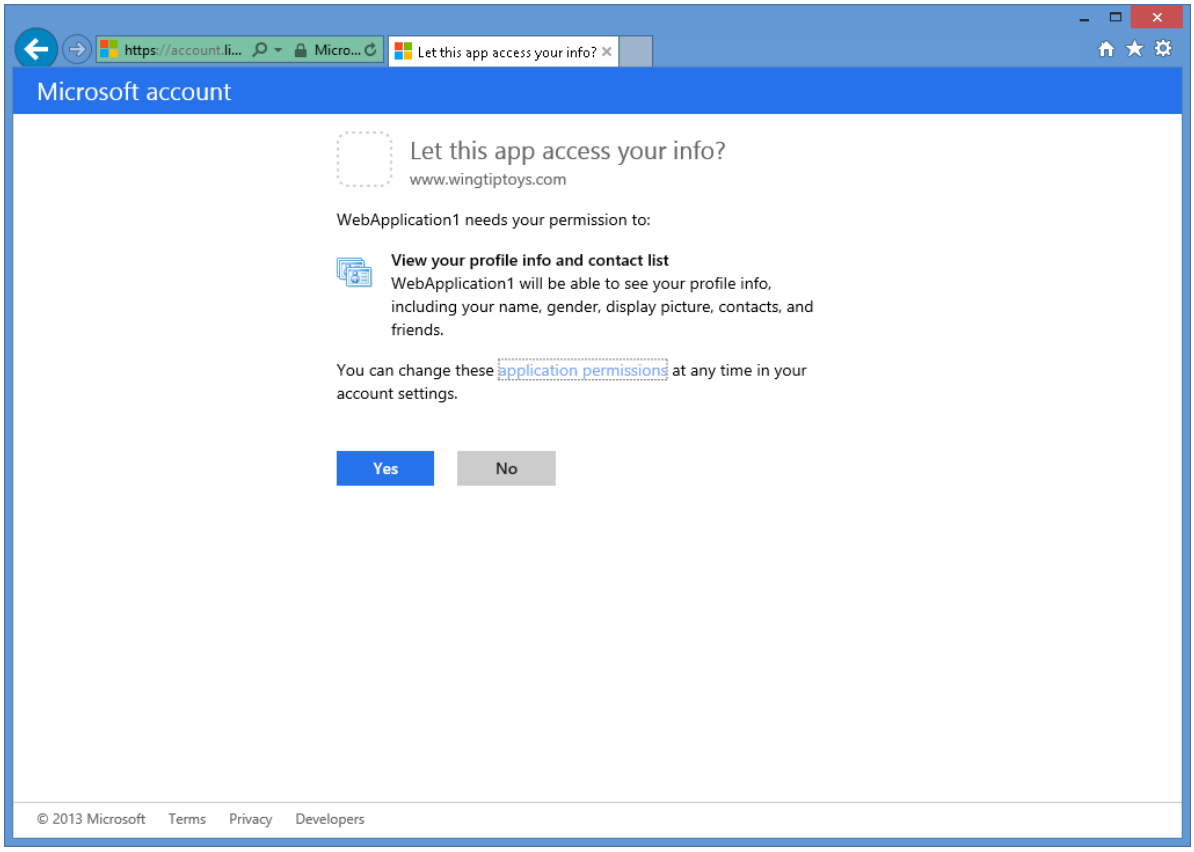
4. When you press F5 to open your web application in your web browser, you will see that Microsoft has been defined as an external authentication service:



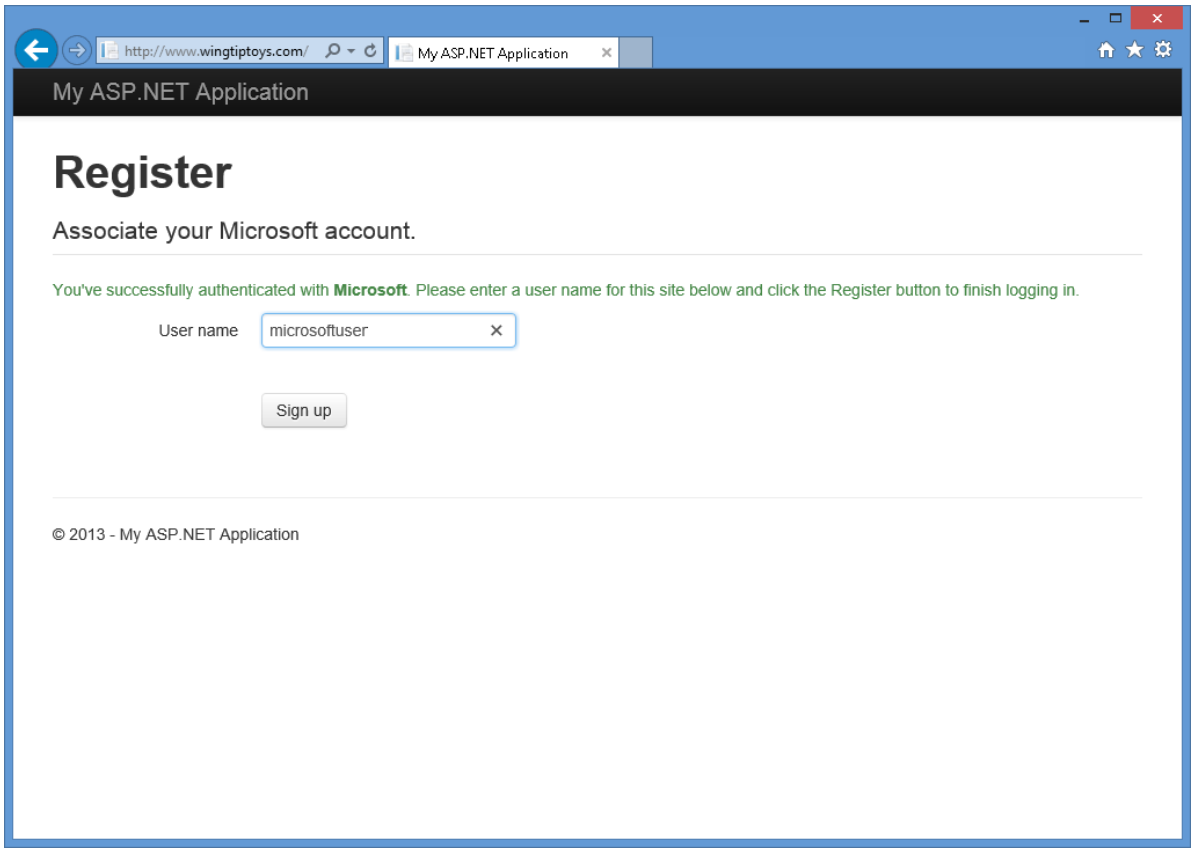
5. When you click the **Microsoft** button, your browser will be redirected to the Microsoft login page:



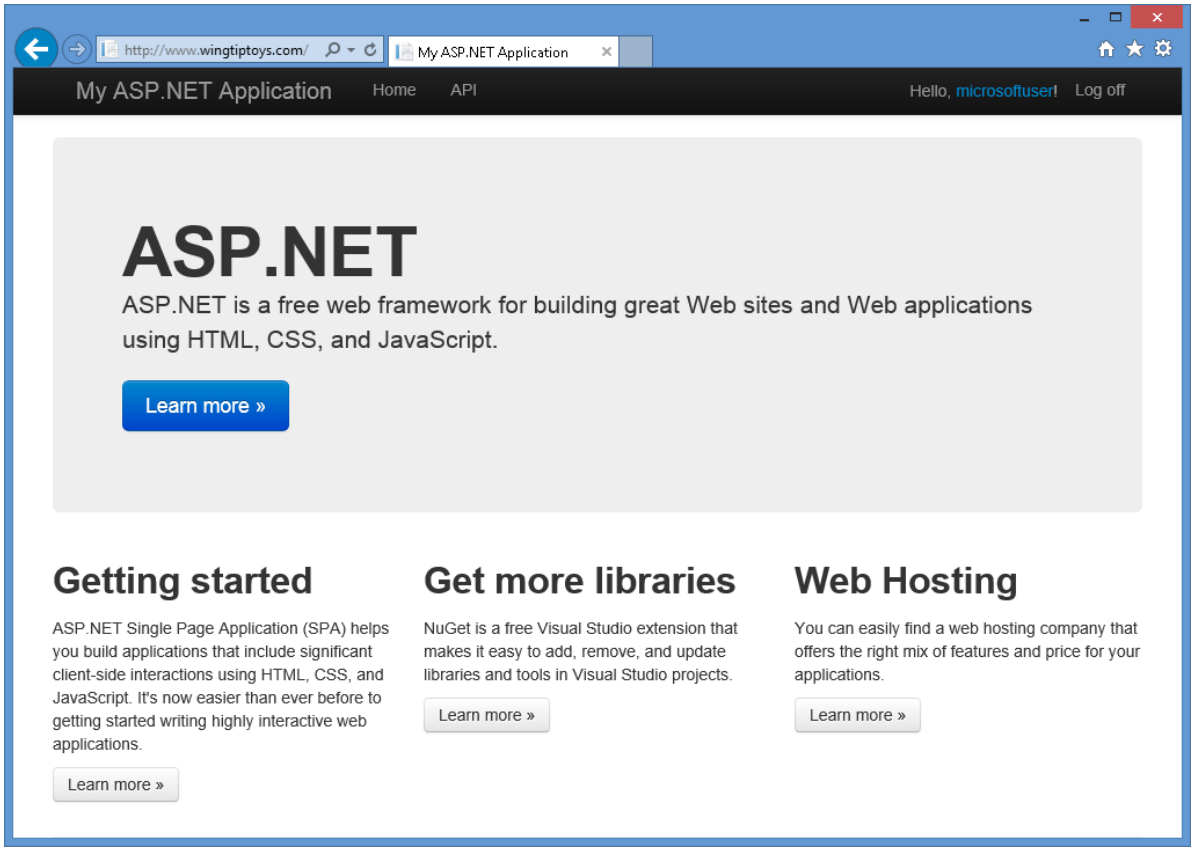
6. After you enter your Microsoft credentials and click **Sign in**, you will be prompted to verify that your web application has permissions to access your Microsoft account:



7. When you click **Yes**, your web browser will be redirected back to your web application, which will prompt you for the **User name** that you want to associate with your Microsoft account:



8. After you have entered your user name and clicked the **Sign up** button, your web application will display the default **home page** for your Microsoft account:



Enabling Twitter Authentication

Twitter authentication requires you to create a developer account, and it requires a consumer key and consumer secret in order to function. For information about creating a Twitter developer account and obtaining your consumer key and consumer secret, see https://go.microsoft.com/fwlink/?LinkID=252166.

Once you have obtained your consumer key and consumer secret, use the following steps to enable Twitter authentication for your web application:

1. When your project is open in Visual Studio 2017, open the *Startup.Auth.cs* file.

2. Locate the Twitter authentication section of code:

```
// Uncomment the following lines to enable logging in with third party login providers
//app.UseMicrosoftAccountAuthentication(
//    clientId: "",
//    clientSecret: "");

//app.UseTwitterAuthentication(
//   consumerKey: "",
//   consumerSecret: "");

//app.UseFacebookAuthentication(
//   appId: "",
//   appSecret: "");

//app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
//{
//    ClientId = "",
//    ClientSecret = ""
//});
```

3. Remove the "//" characters to uncomment the highlighted lines of code, and then add your consumer key and consumer secret. Once you have added those parameters, you can recompile your project:

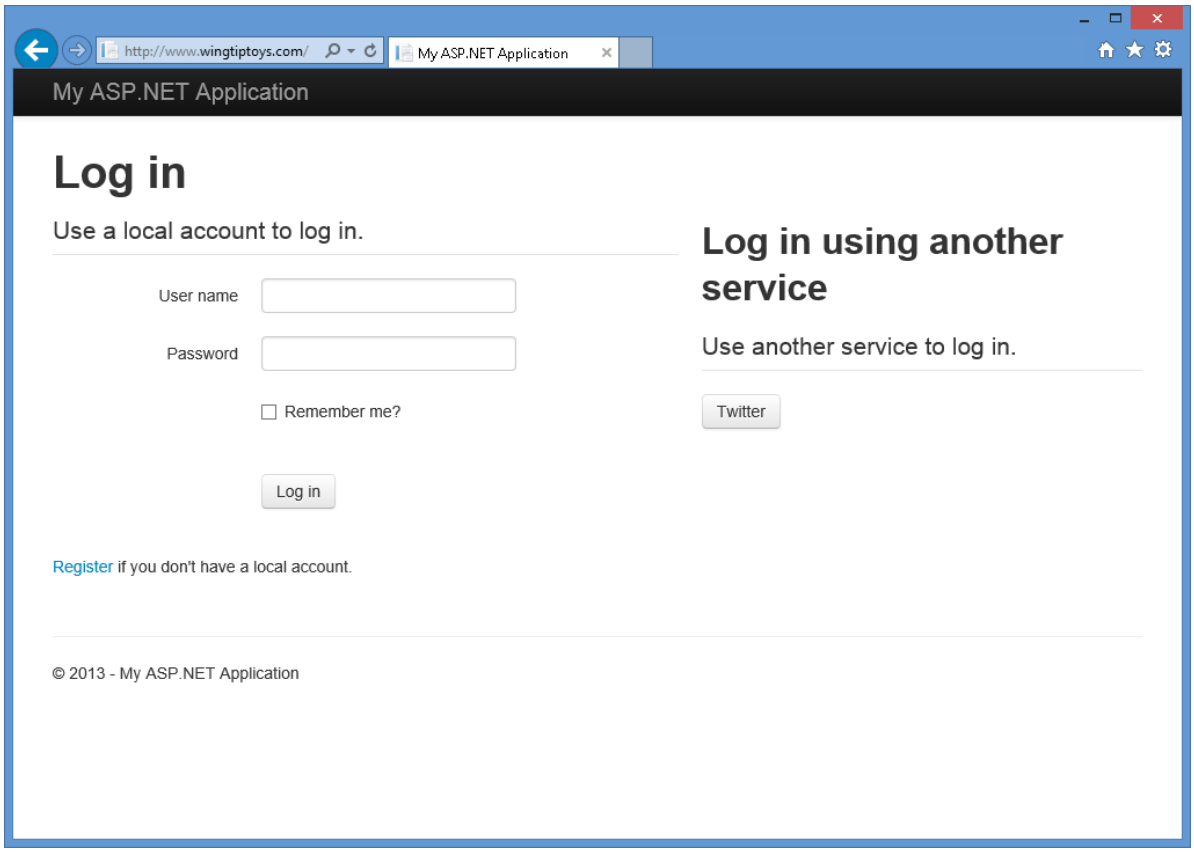
```
// Uncomment the following lines to enable logging in with third party login providers
//app.UseMicrosoftAccountAuthentication(
//    clientId: "",
//    clientSecret: "");

app.UseTwitterAuthentication(
   consumerKey: "426f62526f636b73",
   consumerSecret: "57686f6120447564652c2049495320526f636b73");

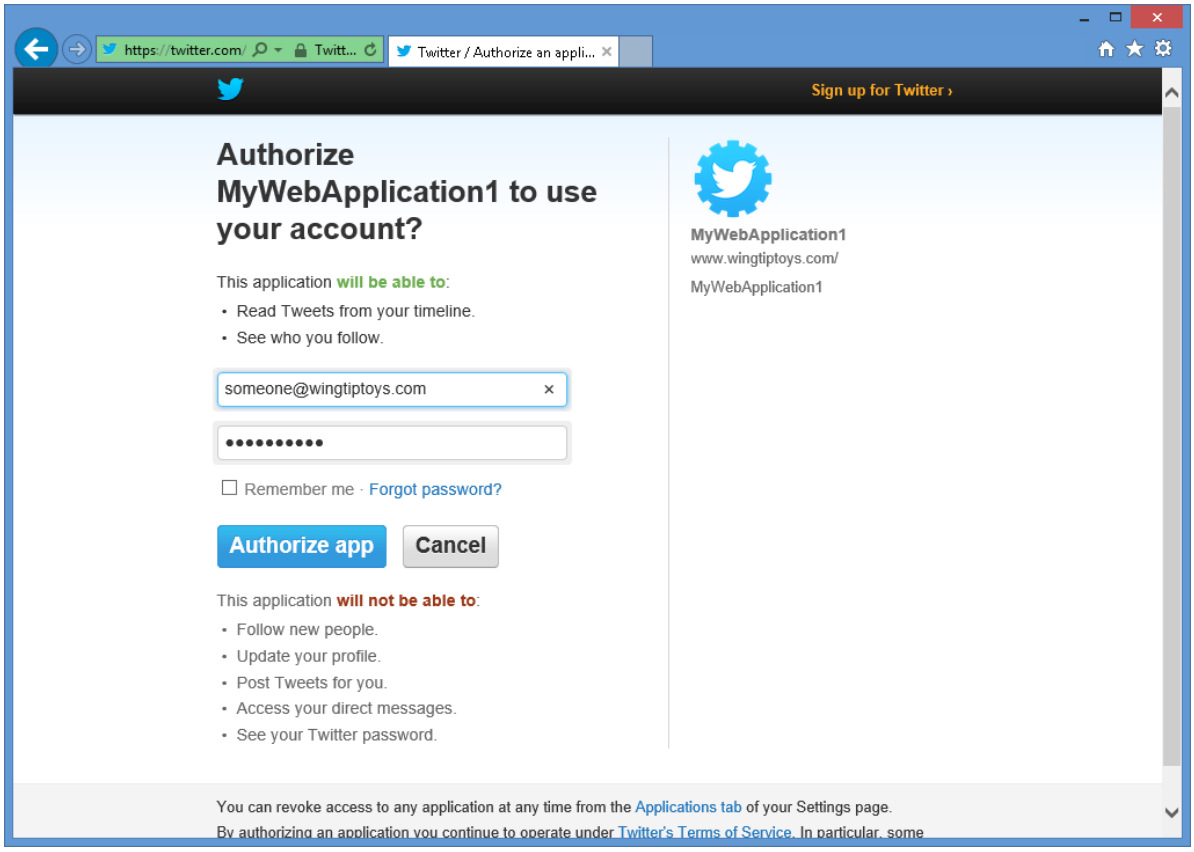
//app.UseFacebookAuthentication(
//   appId: "",
//   appSecret: "");

//app.UseGoogleAuthentication(new GoogleOAuth2AuthenticationOptions()
//{
//    ClientId = "",
//    ClientSecret = ""
//});
```

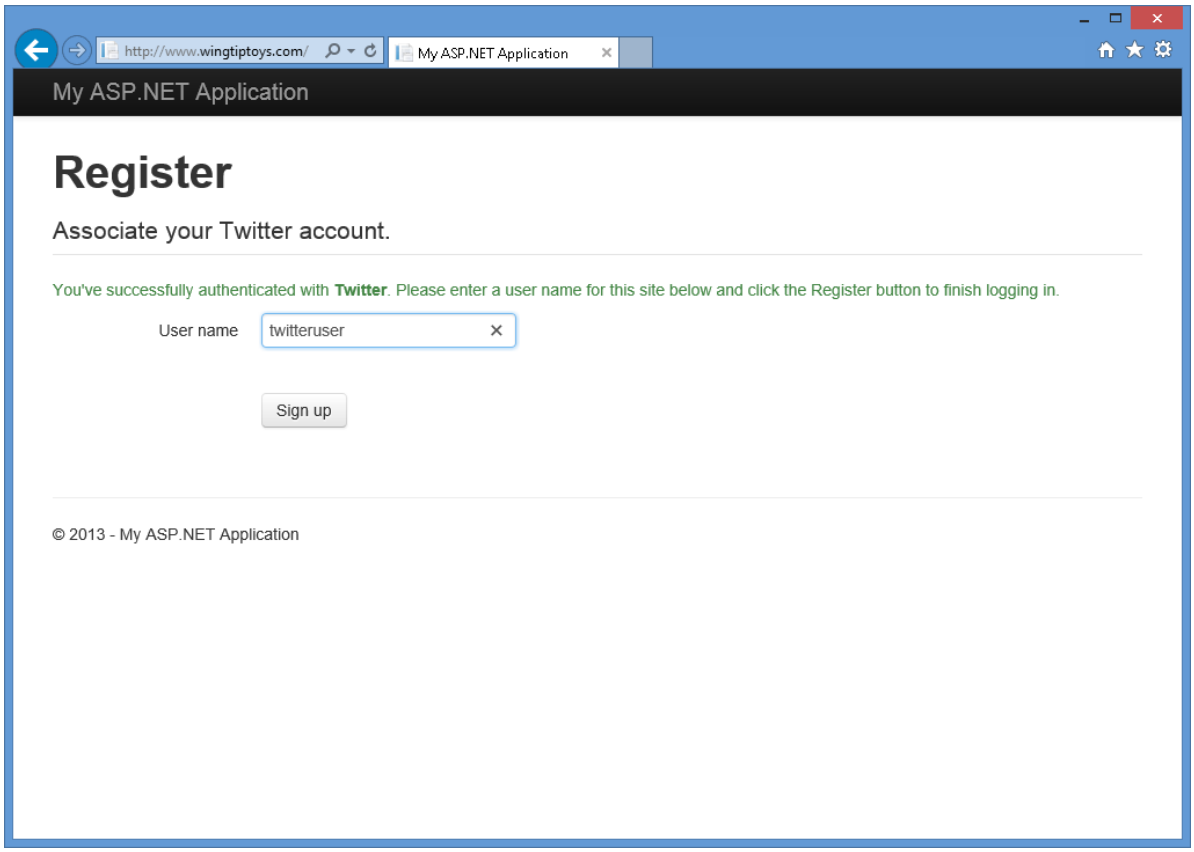
4. When you press F5 to open your web application in your web browser, you will see that Twitter has been defined as an external authentication service:



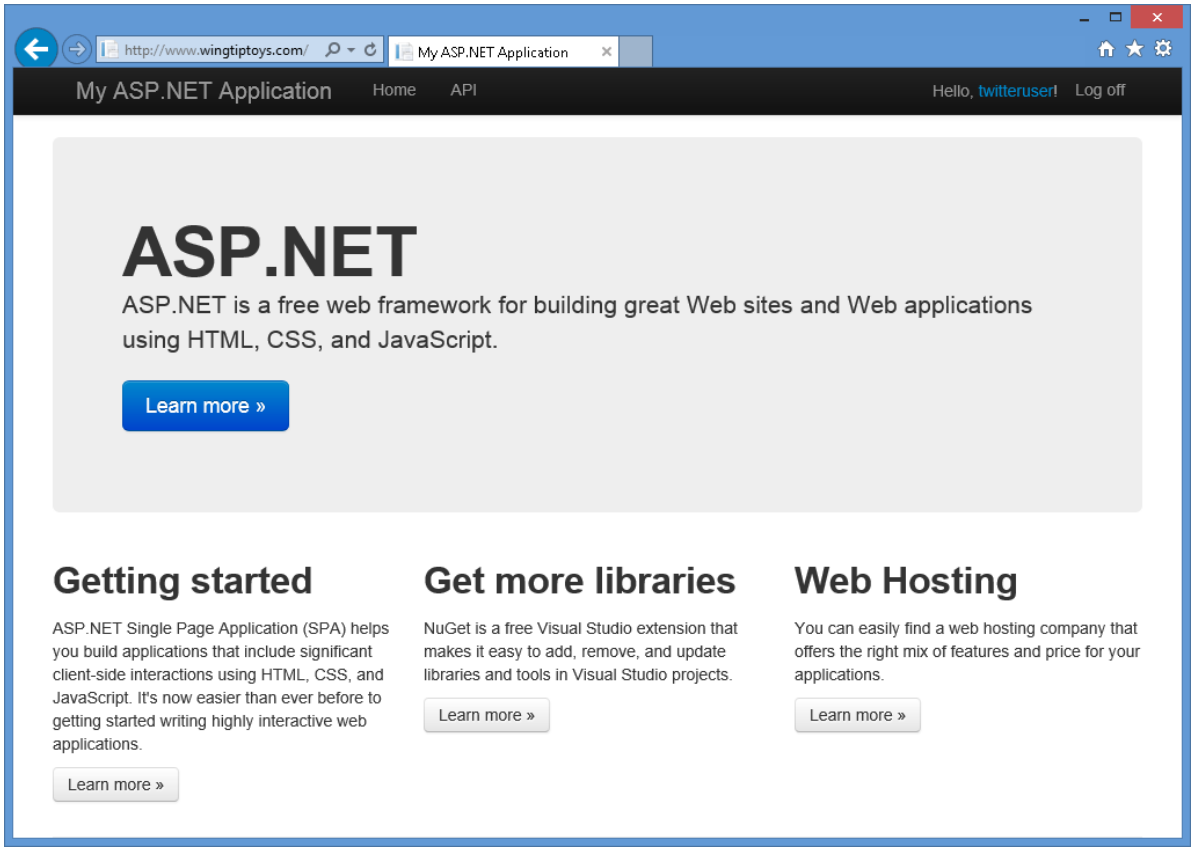
5. When you click the **Twitter** button, your browser will be redirected to the Twitter login page:



6. After you enter your Twitter credentials and click **Authorize app**, your web browser will be redirected back to your web application, which will prompt you for the **User name** that you want to associate with your Twitter account:



7. After you have entered your user name and clicked the **Sign up** button, your web application will display the default **home page** for your Twitter account:



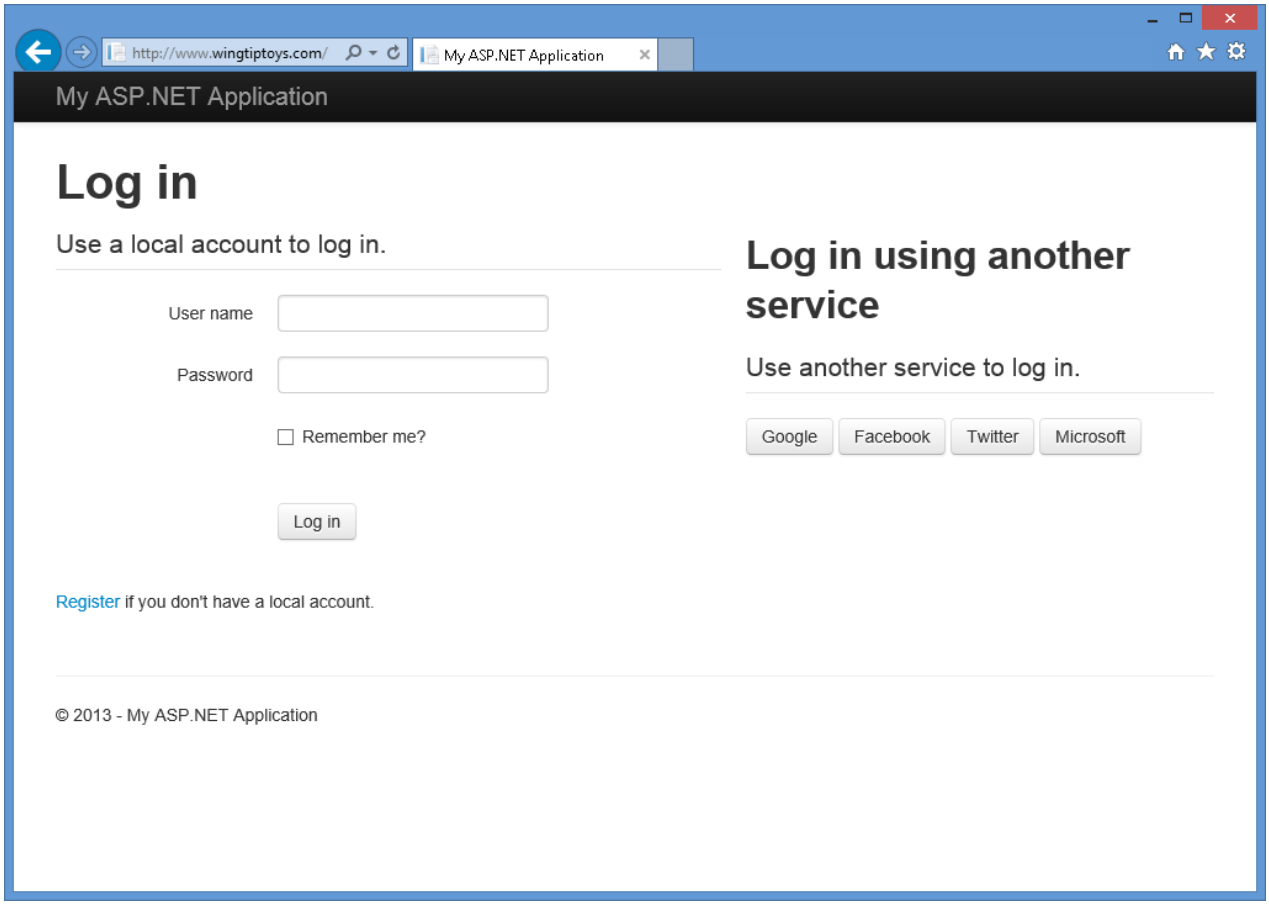
Additional Information

For additional information about creating applications that use OAuth and OpenID, see the following URLs:

- https://go.microsoft.com/fwlink/?LinkID=252166
- https://go.microsoft.com/fwlink/?LinkID=243995

Combining External Authentication Services

For greater flexibility, you can define multiple external authentication services at the same time - this allows your web application's users to use an account from any of the enabled external authentication services:



Configure IIS Express to use a fully qualified domain name

Some external authentication providers do not support testing your application by using an HTTP address like `http://localhost:port/`. To work around this issue, you can add a static Fully Qualified Domain Name (FQDN) mapping to your HOSTS file and configure your project options in Visual Studio 2017 to use the FQDN for testing/debugging. To do so, use the following steps:

- Add a static FQDN mapping your HOSTS file:
  1. Open an elevated command prompt in Windows.
  2. Type the following command:

     ```
     notepad %WinDir%\system32\drivers\etc\hosts
     ```

  3. Add an entry like the following to the HOSTS file:

     ```
     127.0.0.1 www.wingtiptoys.com
     ```

  4. Save and close your HOSTS file.
- Configure your Visual Studio project to use the FQDN:
  1. When your project is open in Visual Studio 2017, click the **Project** menu, and then select your project's properties. For example, you might select **WebApplication1 Properties**.
  2. Select the **Web** tab.
  3. Enter your FQDN for the **Project Url**. For example, you would enter `http://www.wingtiptoys.com` if that was the FQDN mapping that you added to your HOSTS file.
- Configure IIS Express to use the FQDN for your application:

1. Open an elevated command prompt in Windows.
2. Type the following command to change to your IIS Express folder:

   ```
   cd /d "%ProgramFiles%\IIS Express"
   ```

3. Type the following command to add the FQDN to your application:

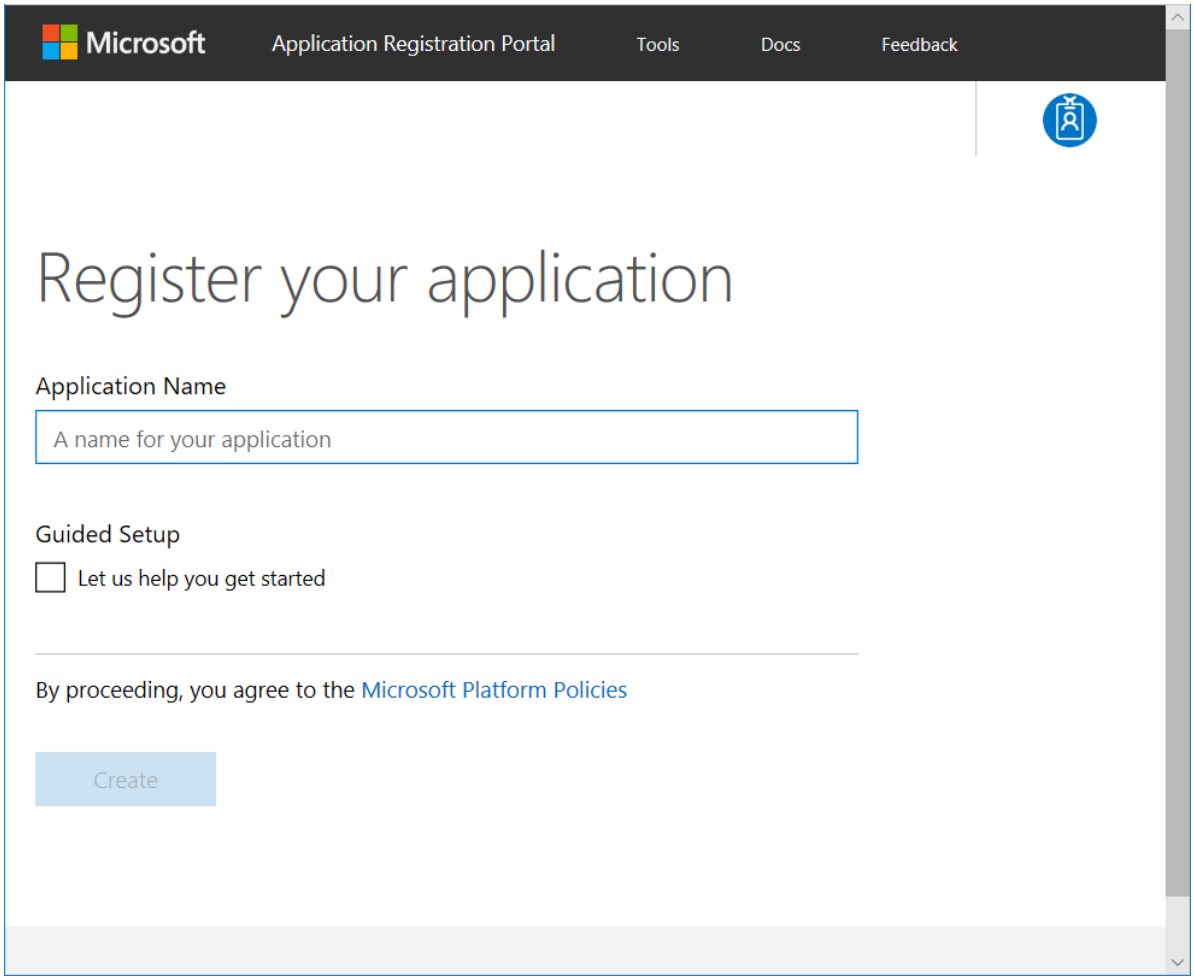
   ```
   appcmd.exe set config -section:system.applicationHost/sites /+"
   [name='WebApplication1'].bindings.
   [protocol='http',bindingInformation='*:80:www.wingtiptoys.com']"
   /commit:apphost
   ```

   Where **WebApplication1** is the name of your project and **bindingInformation** contains the port number and FQDN that you want to use for your testing.

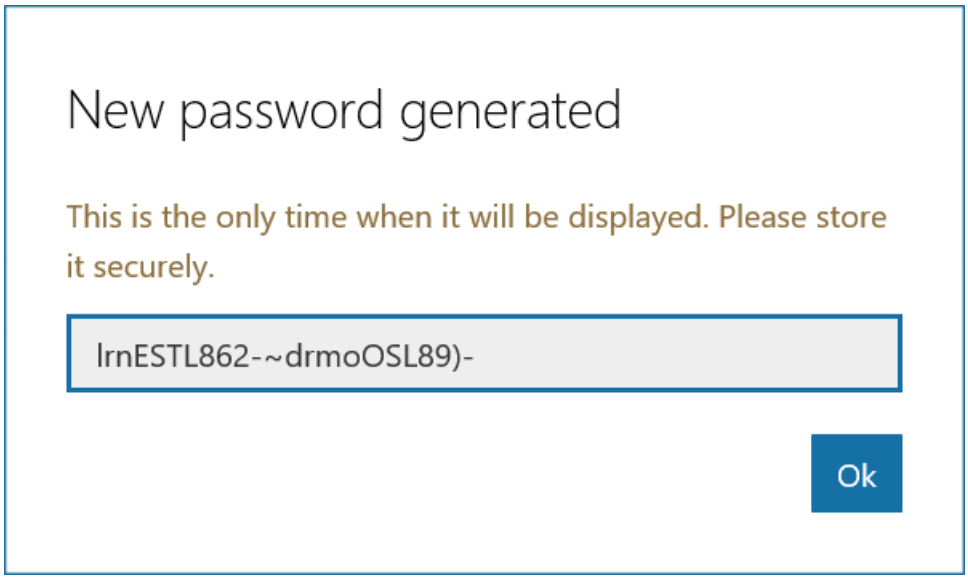
How to Obtain your Application Settings for Microsoft Authentication

Linking an application to Windows Live for Microsoft Authentication is a simple process. If you have not already linked an application to Windows Live, you can use the following steps:

1. Browse to https://go.microsoft.com/fwlink/?LinkID=144070 and enter your Microsoft account name and password when prompted, then click **Sign in**:
2. Select **Add an app** and enter the name of your application when prompted, and then click **Create**:



3. Select your app under **Name** and its application properties page appears.
4. Enter the redirect domain for your application. Copy the **Application ID** and, under **Application Secrets**, select **Generate Password**. Copy the password that appears. The application ID and password are your client ID and client secret. Select **Ok** and then **Save**.



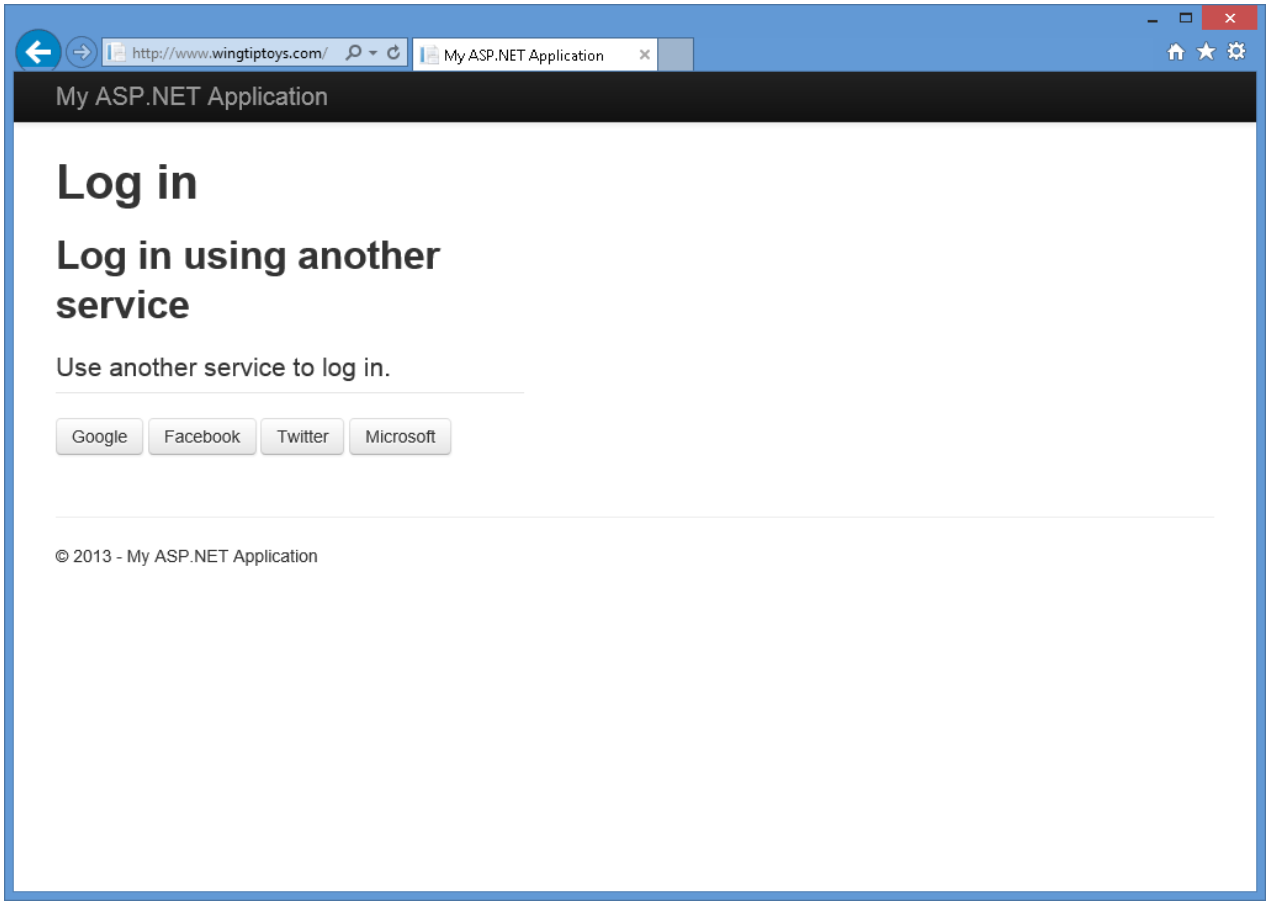
Optional: Disable Local Registration

The current ASP.NET local registration functionality does not prevent automated programs (bots) from creating member accounts; for example, by using a bot-prevention and validation technology like CAPTCHA. Because of this, you should remove the local login form and registration link on the login page. To do so, open the *_Login.cshtml* page in your project, and then comment out the lines for the local login panel and the registration link. The resulting page should look like the following code sample:

```
<!-- ko with: login -->
<hgroup class="title">
    <h1>Log in</h1>
</hgroup>
<div class="row-fluid">
    @*<section class="span7">
        <form>
            <fieldset class="form-horizontal">
                <legend>Use a local account to log in.</legend>
                <ul class="text-error" data-bind="foreach: errors">
                    <li data-bind="text: $data"></li>
                </ul>
                <div class="control-group">
                    <label for="UserName" class="control-label">User name</label>
                    <div class="controls">
                        <input type="text" name="UserName" data-bind="value: userName, hasFocus: true" />
                        <span class="text-error" data-bind="visible: userName.hasError, text:
userName.errorMessage"></span>
                    </div>
                </div>
                <div class="control-group">
                    <label for="Password" class="control-label">Password</label>
                    <div class="controls">
                        <input type="password" name="Password" data-bind="value: password" />
                        <span class="text-error" data-bind="visible: password.hasError, text:
password.errorMessage"></span>
                    </div>
                </div>
                <div class="control-group">
                    <div class="controls">
                        <label class="checkbox">
                            <input type="checkbox" name="RememberMe" data-bind="checked: rememberMe" />
                            <label for="RememberMe">Remember me?</label>
                        </label>
                    </div>
                </div>
                <div class="form-actions no-color">
                    <button type="submit" class="btn" data-bind="click: login, disable: loggingIn">Log
in</button>
```

```
                </div>
                <p><a href="#" data-bind="click: register">Register</a> if you don't have a local account.
</p>
            </fieldset>
        </form>
    </section>*@
    <section class="span5">
        <h2>Log in using another service</h2>
        <div data-bind="visible: loadingExternalLogin">Loading...</div>
        <div data-bind="visible: !loadingExternalLogin()">
            <div class="message-info" data-bind="visible: !hasExternalLogin()">
                <p>
                    There are no external authentication services configured. See <a
href="https://go.microsoft.com/fwlink/?LinkId=252166">this article</a>
                    for details on setting up this ASP.NET application to support logging in via external
services.
                </p>
            </div>
            <form data-bind="visible: hasExternalLogin">
                <fieldset class="form-horizontal">
                    <legend>Use another service to log in.</legend>
                    <p data-bind="foreach: externalLoginProviders">
                        <button type="submit" class="btn" data-bind="text: name, attr: { title: 'Log in using
your ' + name() + ' account' }, click: login"></button>
                    </p>
                </fieldset>
            </form>
        </div>
    </section>
</div>
<!-- /ko -->
```

Once the local login panel and the registration link have been disabled, your login page will only display the external authentication providers that you have enabled:



Preventing Cross-Site Request Forgery (CSRF) Attacks in ASP.NET MVC Application

2/20/2020 • 3 minutes to read • Edit Online

by Mike Wasson

Cross-Site Request Forgery (CSRF) is an attack where a malicious site sends a request to a vulnerable site where the user is currently logged in

Here is an example of a CSRF attack:

1. A user logs into `www.example.com` using forms authentication.
2. The server authenticates the user. The response from the server includes an authentication cookie.
3. Without logging out, the user visits a malicious web site. This malicious site contains the following HTML form:

   ```
   <h1>You Are a Winner!</h1>
     <form action="http://example.com/api/account" method="post">
       <input type="hidden" name="Transaction" value="withdraw" />
       <input type="hidden" name="Amount" value="1000000" />
     <input type="submit" value="Click Me"/>
   </form>
   ```

   Notice that the form action posts to the vulnerable site, not to the malicious site. This is the "cross-site" part of CSRF.
4. The user clicks the submit button. The browser includes the authentication cookie with the request.
5. The request runs on the server with the user's authentication context, and can do anything that an authenticated user is allowed to do.

Although this example requires the user to click the form button, the malicious page could just as easily run a script that submits the form automatically. Moreover, using SSL does not prevent a CSRF attack, because the malicious site can send an "https://" request.

Typically, CSRF attacks are possible against web sites that use cookies for authentication, because browsers send all relevant cookies to the destination web site. However, CSRF attacks are not limited to exploiting cookies. For example, Basic and Digest authentication are also vulnerable. After a user logs in with Basic or Digest authentication. the browser automatically sends the credentials until the session ends.

Anti-Forgery Tokens

To help prevent CSRF attacks, ASP.NET MVC uses anti-forgery tokens, also called *request verification tokens*.

1. The client requests an HTML page that contains a form.
2. The server includes two tokens in the response. One token is sent as a cookie. The other is placed in a hidden form field. The tokens are generated randomly so that an adversary cannot guess the values.
3. When the client submits the form, it must send both tokens back to the server. The client sends the cookie token as a cookie, and it sends the form token inside the form data. (A browser client automatically does this when the user submits the form.)
4. If a request does not include both tokens, the server disallows the request.

Here is an example of an HTML form with a hidden form token:

```
<form action="/Home/Test" method="post">
    <input name="__RequestVerificationToken" type="hidden"
           value="6fGBtLZmVBZ59oUad1Fr33BuPxANKY9q3Srr5y[...]" />
    <input type="submit" value="Submit" />
</form>
```

Anti-forgery tokens work because the malicious page cannot read the user's tokens, due to same-origin policies. (Same-origin policies prevent documents hosted on two different sites from accessing each other's content. So in the earlier example, the malicious page can send requests to example.com, but it cannot read the response.)

To prevent CSRF attacks, use anti-forgery tokens with any authentication protocol where the browser silently sends credentials after the user logs in. This includes cookie-based authentication protocols, such as forms authentication, as well as protocols such as Basic and Digest authentication.

You should require anti-forgery tokens for any nonsafe methods (POST, PUT, DELETE). Also, make sure that safe methods (GET, HEAD) do not have any side effects. Moreover, if you enable cross-domain support, such as CORS or JSONP, then even safe methods like GET are potentially vulnerable to CSRF attacks, allowing the attacker to read potentially sensitive data.

Anti-Forgery Tokens in ASP.NET MVC

To add the anti-forgery tokens to a Razor page, use the **HtmlHelper.AntiForgeryToken** helper method:

```
@using (Html.BeginForm("Manage", "Account")) {
    @Html.AntiForgeryToken()
}
```

This method adds the hidden form field and also sets the cookie token.

Anti-CSRF and AJAX

The form token can be a problem for AJAX requests, because an AJAX request might send JSON data, not HTML form data. One solution is to send the tokens in a custom HTTP header. The following code uses Razor syntax to generate the tokens, and then adds the tokens to an AJAX request. The tokens are generated at the server by calling **AntiForgery.GetTokens**.

```
<script>
    @functions{
        public string TokenHeaderValue()
        {
            string cookieToken, formToken;
            AntiForgery.GetTokens(null, out cookieToken, out formToken);
            return cookieToken + ":" + formToken;
        }
    }

    $.ajax("api/values", {
        type: "post",
        contentType: "application/json",
        data: {  }, // JSON data goes here
        dataType: "json",
        headers: {
            'RequestVerificationToken': '@TokenHeaderValue()'
        }
    });
</script>
```

When you process the request, extract the tokens from the request header. Then call the **AntiForgery.Validate** method to validate the tokens. The **Validate** method throws an exception if the tokens are not valid.

```
void ValidateRequestHeader(HttpRequestMessage request)
{
    string cookieToken = "";
    string formToken = "";

    IEnumerable<string> tokenHeaders;
    if (request.Headers.TryGetValues("RequestVerificationToken", out tokenHeaders))
    {
        string[] tokens = tokenHeaders.First().Split(':');
        if (tokens.Length == 2)
        {
            cookieToken = tokens[0].Trim();
            formToken = tokens[1].Trim();
        }
    }
    AntiForgery.Validate(cookieToken, formToken);
}
```

Enable cross-origin requests in ASP.NET Web API 2

2/20/2020 • 12 minutes to read • Edit Online

by Mike Wasson

Browser security prevents a web page from making AJAX requests to another domain. This restriction is called the *same-origin policy*, and prevents a malicious site from reading sensitive data from another site. However, sometimes you might want to let other sites call your web API.

Cross Origin Resource Sharing (CORS) is a W3C standard that allows a server to relax the same-origin policy. Using CORS, a server can explicitly allow some cross-origin requests while rejecting others. CORS is safer and more flexible than earlier techniques such as JSONP. This tutorial shows how to enable CORS in your Web API application.

Software used in the tutorial

- Visual Studio
- Web API 2.2

Introduction

This tutorial demonstrates CORS support in ASP.NET Web API. We'll start by creating two ASP.NET projects – one called "WebService", which hosts a Web API controller, and the other called "WebClient", which calls WebService. Because the two applications are hosted at different domains, an AJAX request from WebClient to WebService is a cross-origin request.

What is "same origin"?

Two URLs have the same origin if they have identical schemes, hosts, and ports. (RFC 6454)

These two URLs have the same origin:

- `http://example.com/foo.html`
- `http://example.com/bar.html`

These URLs have different origins than the previous two:

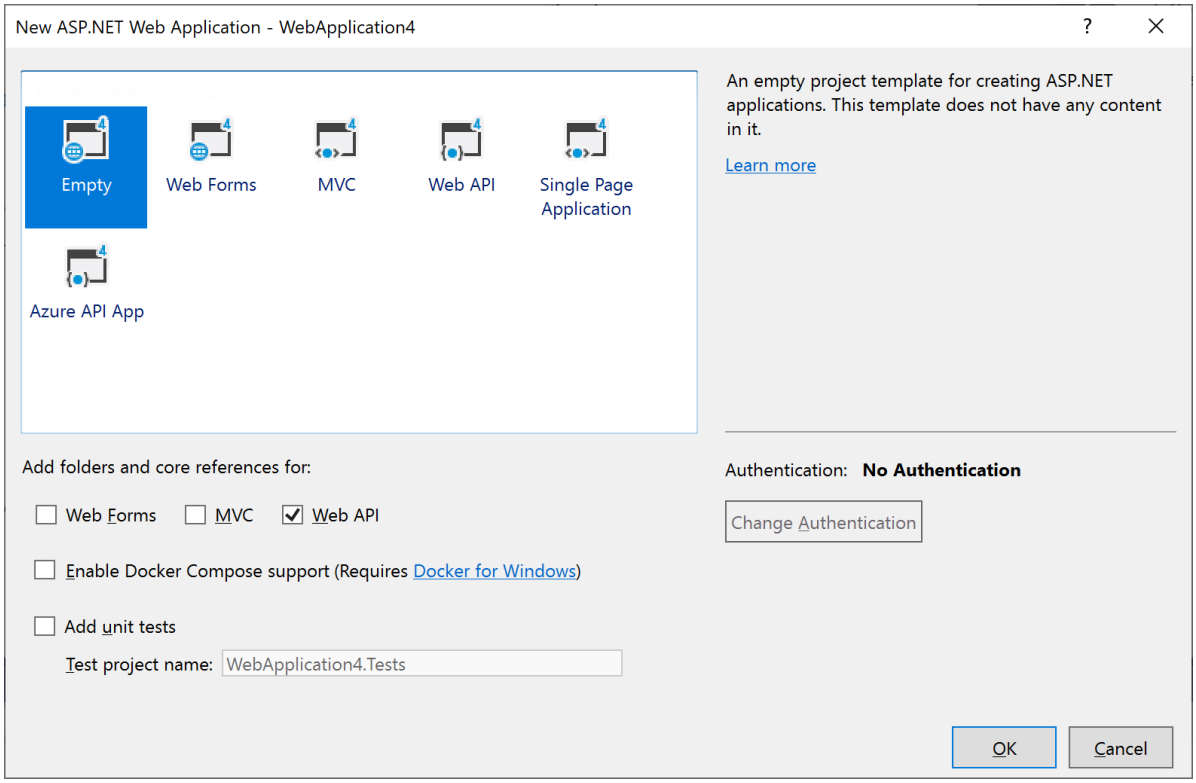
- `http://example.net` - Different domain
- `http://example.com:9000/foo.html` - Different port
- `https://example.com/foo.html` - Different scheme
- `http://www.example.com/foo.html` - Different subdomain

> **NOTE**
>
> Internet Explorer does not consider the port when comparing origins.

Create the WebService project

> **NOTE**
>
> This section assumes you already know how to create Web API projects. If not, see Getting Started with ASP.NET Web API.

1. Start Visual Studio and create a new **ASP.NET Web Application (.NET Framework)** project.
2. In the **New ASP.NET Web Application** dialog box, select the **Empty** project template. Under **Add folders and core references for**, select the **Web API** checkbox.



3. Add a Web API controller named `TestController` with the following code:

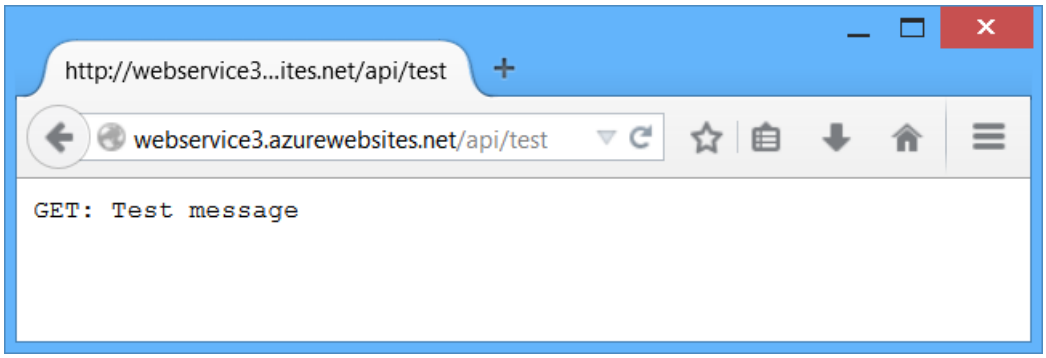
```
using System.Net.Http;
using System.Web.Http;

namespace WebService.Controllers
{
    public class TestController : ApiController
    {
        public HttpResponseMessage Get()
        {
            return new HttpResponseMessage()
            {
                Content = new StringContent("GET: Test message")
            };
        }

        public HttpResponseMessage Post()
        {
            return new HttpResponseMessage()
            {
                Content = new StringContent("POST: Test message")
            };
        }

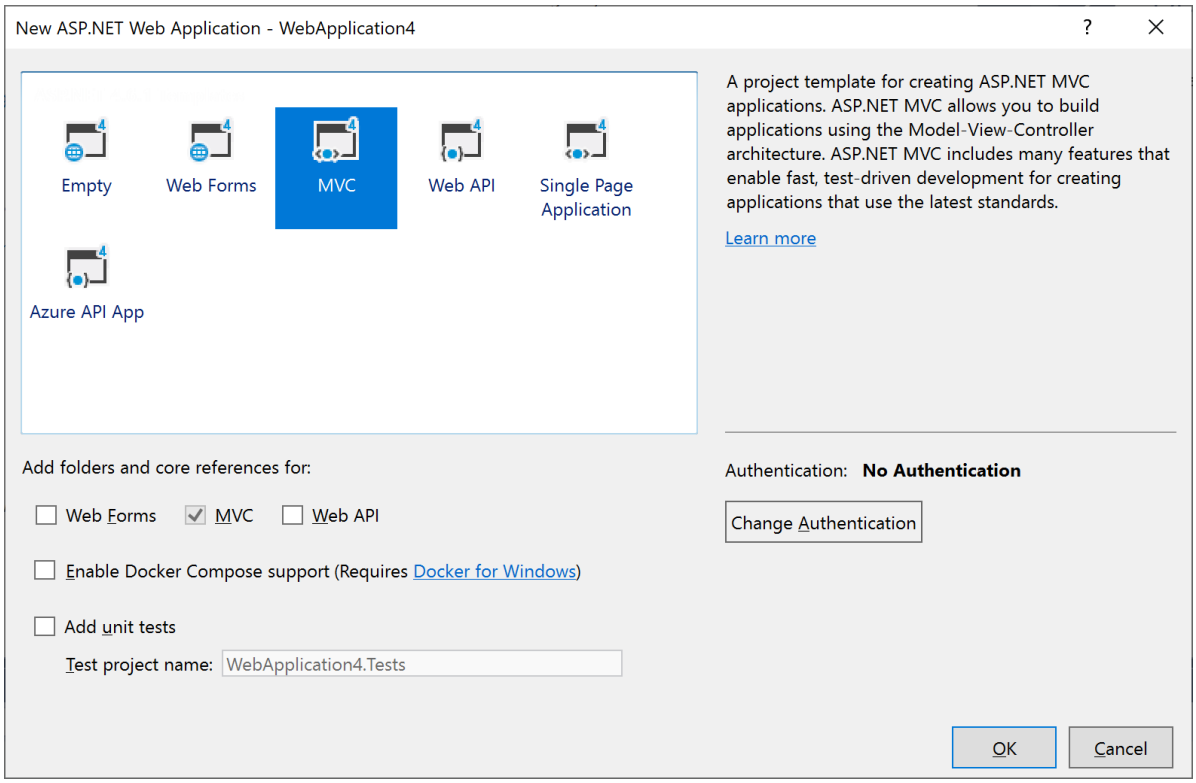
        public HttpResponseMessage Put()
        {
            return new HttpResponseMessage()
            {
                Content = new StringContent("PUT: Test message")
            };
        }
    }
}
```

4. You can run the application locally or deploy to Azure. (For the screenshots in this tutorial, the app deploys to Azure App Service Web Apps.) To verify that the web API is working, navigate to `http://hostname/api/test/`, where *hostname* is the domain where you deployed the application. You should see the response text, "GET: Test Message".



Create the WebClient project

1. Create another **ASP.NET Web Application (.NET Framework)** project and select the **MVC** project template. Optionally, select **Change Authentication** > **No Authentication**. You don't need authentication for this tutorial.



2. In **Solution Explorer**, open the file *Views/Home/Index.cshtml*. Replace the code in this file with the following:

```
<div>
    <select id="method">
        <option value="get">GET</option>
        <option value="post">POST</option>
        <option value="put">PUT</option>
    </select>
    <input type="button" value="Try it" onclick="sendRequest()" />
    <span id='value1'>(Result)</span>
</div>

@section scripts {
<script>
    // TODO: Replace with the URL of your WebService app
    var serviceUrl = 'http://mywebservice/api/test';

    function sendRequest() {
        var method = $('#method').val();

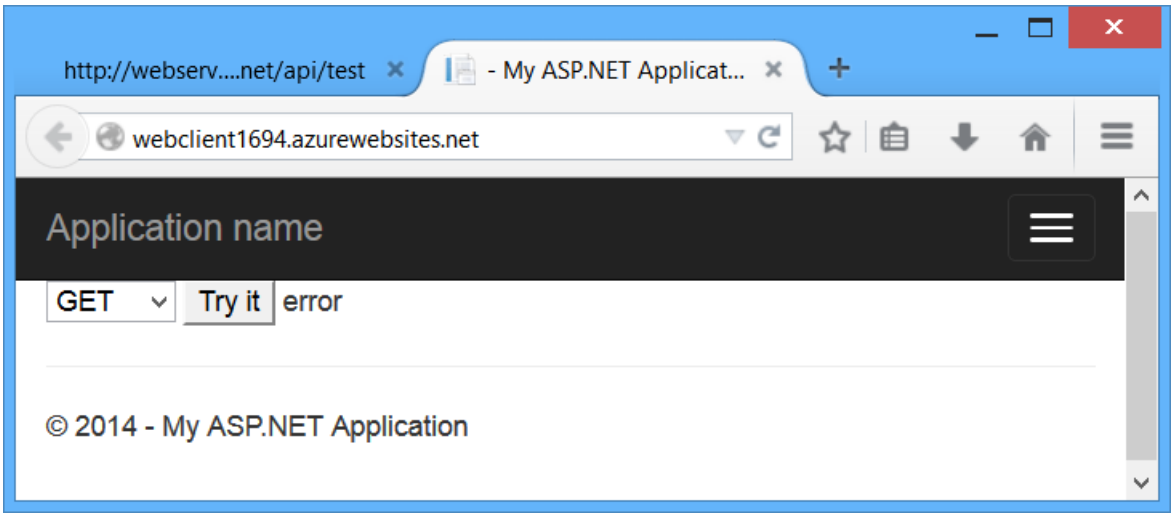
        $.ajax({
            type: method,
            url: serviceUrl
        }).done(function (data) {
            $('#value1').text(data);
        }).fail(function (jqXHR, textStatus, errorThrown) {
            $('#value1').text(jqXHR.responseText || textStatus);
        });
    }
</script>
}
```

For the *serviceUrl* variable, use the URI of the WebService app.

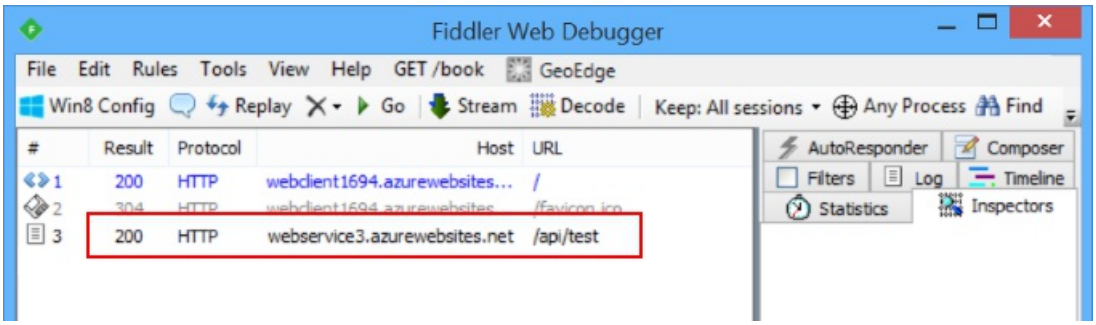
3. Run the WebClient app locally or publish it to another website.

When you click the "Try It" button, an AJAX request is submitted to the WebService app using the HTTP method listed in the dropdown box (GET, POST, or PUT). This lets you examine different cross-origin requests. Currently,

the WebService app does not support CORS, so if you click the button you'll get an error.



> **NOTE**
>
> If you watch the HTTP traffic in a tool like Fiddler, you'll see that the browser does send the GET request, and the request succeeds, but the AJAX call returns an error. It's important to understand that same-origin policy does not prevent the browser from *sending* the request. Instead, it prevents the application from seeing the *response*.



Enable CORS

Now let's enable CORS in the WebService app. First, add the CORS NuGet package. In Visual Studio, from the **Tools** menu, select **NuGet Package Manager**, then select **Package Manager Console**. In the Package Manager Console window, type the following command:

```
Install-Package Microsoft.AspNet.WebApi.Cors
```

This command installs the latest package and updates all dependencies, including the core Web API libraries. Use the `-Version` flag to target a specific version. The CORS package requires Web API 2.0 or later.

Open the file *App_Start/WebApiConfig.cs*. Add the following code to the **WebApiConfig.Register** method:

```
using System.Web.Http;
namespace WebService
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // New code
            config.EnableCors();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

Next, add the **[EnableCors]** attribute to the `TestController` class:

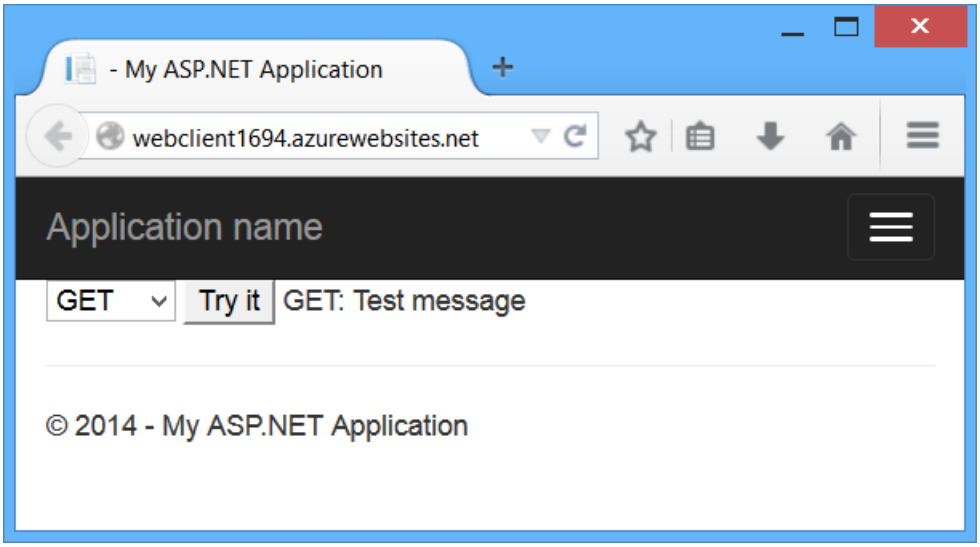
```
using System.Net.Http;
using System.Web.Http;
using System.Web.Http.Cors;

namespace WebService.Controllers
{
    [EnableCors(origins: "http://mywebclient.azurewebsites.net", headers: "*", methods: "*")]
    public class TestController : ApiController
    {
        // Controller methods not shown...
    }
}
```

For the *origins* parameter, use the URI where you deployed the WebClient application. This allows cross-origin requests from WebClient, while still disallowing all other cross-domain requests. Later, I'll describe the parameters for **[EnableCors]** in more detail.

Do not include a forward slash at the end of the *origins* URL.

Redeploy the updated WebService application. You don't need to update WebClient. Now the AJAX request from WebClient should succeed. The GET, PUT, and POST methods are all allowed.



How CORS Works

This section describes what happens in a CORS request, at the level of the HTTP messages. It's important to understand how CORS works, so that you can configure the **[EnableCors]** attribute correctly and troubleshoot if things don't work as you expect.

The CORS specification introduces several new HTTP headers that enable cross-origin requests. If a browser supports CORS, it sets these headers automatically for cross-origin requests; you don't need to do anything special in your JavaScript code.

Here is an example of a cross-origin request. The "Origin" header gives the domain of the site that is making the request.

```
GET http://myservice.azurewebsites.net/api/test HTTP/1.1
Referer: http://myclient.azurewebsites.net/
Accept: */*
Accept-Language: en-US
Origin: http://myclient.azurewebsites.net
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
```

If the server allows the request, it sets the Access-Control-Allow-Origin header. The value of this header either matches the Origin header, or is the wildcard value "*", meaning that any origin is allowed.

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: text/plain; charset=utf-8
Access-Control-Allow-Origin: http://myclient.azurewebsites.net
Date: Wed, 05 Jun 2013 06:27:30 GMT
Content-Length: 17

GET: Test message
```

If the response does not include the Access-Control-Allow-Origin header, the AJAX request fails. Specifically, the browser disallows the request. Even if the server returns a successful response, the browser does not make the response available to the client application.

Preflight Requests

For some CORS requests, the browser sends an additional request, called a "preflight request", before it sends the actual request for the resource.

The browser can skip the preflight request if the following conditions are true:

- The request method is GET, HEAD, or POST, *and*
- The application does not set any request headers other than Accept, Accept-Language, Content-Language, Content-Type, or Last-Event-ID, *and*
- The Content-Type header (if set) is one of the following:
  - application/x-www-form-urlencoded
  - multipart/form-data
  - text/plain

The rule about request headers applies to headers that the application sets by calling **setRequestHeader** on the **XMLHttpRequest** object. (The CORS specification calls these "author request headers".) The rule does not apply to headers the *browser* can set, such as User-Agent, Host, or Content-Length.

Here is an example of a preflight request:

```
OPTIONS http://myservice.azurewebsites.net/api/test HTTP/1.1
Accept: */*
Origin: http://myclient.azurewebsites.net
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: accept, x-my-custom-header
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
Content-Length: 0
```

The pre-flight request uses the HTTP OPTIONS method. It includes two special headers:

- Access-Control-Request-Method: The HTTP method that will be used for the actual request.
- Access-Control-Request-Headers: A list of request headers that the *application* set on the actual request. (Again, this does not include headers that the browser sets.)

Here is an example response, assuming that the server allows the request:

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 0
Access-Control-Allow-Origin: http://myclient.azurewebsites.net
Access-Control-Allow-Headers: x-my-custom-header
Access-Control-Allow-Methods: PUT
Date: Wed, 05 Jun 2013 06:33:22 GMT
```

The response includes an Access-Control-Allow-Methods header that lists the allowed methods, and optionally an Access-Control-Allow-Headers header, which lists the allowed headers. If the preflight request succeeds, the browser sends the actual request, as described earlier.

Tools commonly used to test endpoints with preflight OPTIONS requests (for example, Fiddler and Postman) don't send the required OPTIONS headers by default. Confirm that the `Access-Control-Request-Method` and `Access-Control-Request-Headers` headers are sent with the request and that OPTIONS headers reach the app through IIS.

To configure IIS to allow an ASP.NET app to receive and handle OPTION requests, add the following configuration to the app's *web.config* file in the `<system.webServer><handlers>` section:

```
<system.webServer>
  <handlers>
    <remove name="ExtensionlessUrlHandler-Integrated-4.0" />
    <remove name="OPTIONSVerbHandler" />
    <add name="ExtensionlessUrlHandler-Integrated-4.0" path="*." verb="*"
type="System.Web.Handlers.TransferRequestHandler" preCondition="integratedMode,runtimeVersionv4.0" />
  </handlers>
</system.webServer>
```

The removal of `OPTIONSVerbHandler` prevents IIS from handling OPTIONS requests. The replacement of `ExtensionlessUrlHandler-Integrated-4.0` allows OPTIONS requests to reach the app because the default module registration only allows GET, HEAD, POST, and DEBUG requests with extensionless URLs.

Scope Rules for [EnableCors]

You can enable CORS per action, per controller, or globally for all Web API controllers in your application.

Per Action

To enable CORS for a single action, set the **[EnableCors]** attribute on the action method. The following example enables CORS for the `GetItem` method only.

```
public class ItemsController : ApiController
{
    public HttpResponseMessage GetAll() { ... }

    [EnableCors(origins: "http://www.example.com", headers: "*", methods: "*")]
    public HttpResponseMessage GetItem(int id) { ... }

    public HttpResponseMessage Post() { ... }
    public HttpResponseMessage PutItem(int id) { ... }
}
```

Per Controller

If you set **[EnableCors]** on the controller class, it applies to all the actions on the controller. To disable CORS for an action, add the **[DisableCors]** attribute to the action. The following example enables CORS for every method except `PutItem`.

```
[EnableCors(origins: "http://www.example.com", headers: "*", methods: "*")]
public class ItemsController : ApiController
{
    public HttpResponseMessage GetAll() { ... }
    public HttpResponseMessage GetItem(int id) { ... }
    public HttpResponseMessage Post() { ... }

    [DisableCors]
    public HttpResponseMessage PutItem(int id) { ... }
}
```

Globally

To enable CORS for all Web API controllers in your application, pass an **EnableCorsAttribute** instance to the **EnableCors** method:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        var cors = new EnableCorsAttribute("www.example.com", "*", "*");
        config.EnableCors(cors);
        // ...
    }
}
```

If you set the attribute at more than one scope, the order of precedence is:

1. Action
2. Controller
3. Global

Set the allowed origins

The *origins* parameter of the **[EnableCors]** attribute specifies which origins are allowed to access the resource. The value is a comma-separated list of the allowed origins.

```
[EnableCors(origins: "http://www.contoso.com,http://www.example.com",
    headers: "*", methods: "*")]
```

You can also use the wildcard value "*" to allow requests from any origins.

Consider carefully before allowing requests from any origin. It means that literally any website can make AJAX calls to your web API.

```
// Allow CORS for all origins. (Caution!)
[EnableCors(origins: "*", headers: "*", methods: "*")]
```

Set the allowed HTTP methods

The *methods* parameter of the **[EnableCors]** attribute specifies which HTTP methods are allowed to access the resource. To allow all methods, use the wildcard value "*". The following example allows only GET and POST requests.

```
[EnableCors(origins: "http://www.example.com", headers: "*", methods: "get,post")]
public class TestController : ApiController
{
    public HttpResponseMessage Get() { ... }
    public HttpResponseMessage Post() { ... }
    public HttpResponseMessage Put() { ... }
}
```

Set the allowed request headers

This article described earlier how a preflight request might include an Access-Control-Request-Headers header, listing the HTTP headers set by the application (the so-called "author request headers"). The *headers* parameter of the **[EnableCors]** attribute specifies which author request headers are allowed. To allow any headers, set *headers* to "*". To whitelist specific headers, set *headers* to a comma-separated list of the allowed headers:

```
[EnableCors(origins: "http://example.com",
    headers: "accept,content-type,origin,x-my-header", methods: "*")]
```

However, browsers are not entirely consistent in how they set Access-Control-Request-Headers. For example, Chrome currently includes "origin". FireFox does not include standard headers such as "Accept", even when the application sets them in script.

If you set *headers* to anything other than "*", you should include at least "accept", "content-type", and "origin", plus any custom headers that you want to support.

Set the allowed response headers

By default, the browser does not expose all of the response headers to the application. The response headers that are available by default are:

- Cache-Control
- Content-Language
- Content-Type
- Expires
- Last-Modified

- Pragma

The CORS spec calls these simple response headers. To make other headers available to the application, set the *exposedHeaders* parameter of **[EnableCors]**.

In the following example, the controller's `Get` method sets a custom header named 'X-Custom-Header'. By default, the browser will not expose this header in a cross-origin request. To make the header available, include 'X-Custom-Header' in *exposedHeaders*.

```
[EnableCors(origins: "*", headers: "*", methods: "*", exposedHeaders: "X-Custom-Header")]
public class TestController : ApiController
{
    public HttpResponseMessage Get()
    {
        var resp = new HttpResponseMessage()
        {
            Content = new StringContent("GET: Test message")
        };
        resp.Headers.Add("X-Custom-Header", "hello");
        return resp;
    }
}
```

Pass credentials in cross-origin requests

Credentials require special handling in a CORS request. By default, the browser does not send any credentials with a cross-origin request. Credentials include cookies as well as HTTP authentication schemes. To send credentials with a cross-origin request, the client must set **XMLHttpRequest.withCredentials** to true.

Using **XMLHttpRequest** directly:

```
var xhr = new XMLHttpRequest();
xhr.open('get', 'http://www.example.com/api/test');
xhr.withCredentials = true;
```

In jQuery:

```
$.ajax({
    type: 'get',
    url: 'http://www.example.com/api/test',
    xhrFields: {
        withCredentials: true
    }
```

In addition, the server must allow the credentials. To allow cross-origin credentials in Web API, set the **SupportsCredentials** property to true on the **[EnableCors]** attribute:

```
[EnableCors(origins: "http://myclient.azurewebsites.net", headers: "*",
    methods: "*", SupportsCredentials = true)]
```

If this property is true, the HTTP response will include an Access-Control-Allow-Credentials header. This header tells the browser that the server allows credentials for a cross-origin request.

If the browser sends credentials, but the response does not include a valid Access-Control-Allow-Credentials header, the browser will not expose the response to the application, and the AJAX request fails.

Be careful about setting **SupportsCredentials** to true, because it means a website at another domain can send a

logged-in user's credentials to your Web API on the user's behalf, without the user being aware. The CORS spec also states that setting *origins* to "*" is invalid if **SupportsCredentials** is true.

Custom CORS policy providers

The **[EnableCors]** attribute implements the **ICorsPolicyProvider** interface. You can provide your own implementation by creating a class that derives from **Attribute** and implements **ICorsPolicyProvider**.

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, AllowMultiple = false)]
public class MyCorsPolicyAttribute : Attribute, ICorsPolicyProvider
{
    private CorsPolicy _policy;

    public MyCorsPolicyAttribute()
    {
        // Create a CORS policy.
        _policy = new CorsPolicy
        {
            AllowAnyMethod = true,
            AllowAnyHeader = true
        };

        // Add allowed origins.
        _policy.Origins.Add("http://myclient.azurewebsites.net");
        _policy.Origins.Add("http://www.contoso.com");
    }

    public Task<CorsPolicy> GetCorsPolicyAsync(HttpRequestMessage request)
    {
        return Task.FromResult(_policy);
    }
}
```

Now you can apply the attribute any place that you would put **[EnableCors]**.

```
[MyCorsPolicy]
public class TestController : ApiController
{
    .. //
```

For example, a custom CORS policy provider could read the settings from a configuration file.

As an alternative to using attributes, you can register an **ICorsPolicyProviderFactory** object that creates **ICorsPolicyProvider** objects.

```
public class CorsPolicyFactory : ICorsPolicyProviderFactory
{
    ICorsPolicyProvider _provider = new MyCorsPolicyProvider();

    public ICorsPolicyProvider GetCorsPolicyProvider(HttpRequestMessage request)
    {
        return _provider;
    }
}
```

To set the **ICorsPolicyProviderFactory**, call the **SetCorsPolicyProviderFactory** extension method at startup, as follows:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.SetCorsPolicyProviderFactory(new CorsPolicyFactory());
        config.EnableCors();

        // ...
    }
}
```

Browser support

The Web API CORS package is a server-side technology. The user's browser also needs to support CORS. Fortunately, the current versions of all major browsers include support for CORS.

Authentication Filters in ASP.NET Web API 2

2/20/2020 • 8 minutes to read • Edit Online

by Mike Wasson

> An authentication filter is a component that authenticates an HTTP request. Web API 2 and MVC 5 both support authentication filters, but they differ slightly, mostly in the naming conventions for the filter interface. This topic describes Web API authentication filters.

Authentication filters let you set an authentication scheme for individual controllers or actions. That way, your app can support different authentication mechanisms for different HTTP resources.

In this article, I'll show code from the Basic Authentication sample on https://github.com/aspnet/samples. The sample shows an authentication filter that implements the HTTP Basic Access Authentication scheme (RFC 2617). The filter is implemented in a class named `IdentityBasicAuthenticationAttribute`. I won't show all of the code from the sample, just the parts that illustrate how to write an authentication filter.

Setting an Authentication Filter

Like other filters, authentication filters can be applied per-controller, per-action, or globally to all Web API controllers.

To apply an authentication filter to a controller, decorate the controller class with the filter attribute. The following code sets the `[IdentityBasicAuthentication]` filter on a controller class, which enables Basic Authentication for all of the controller's actions.

```
[IdentityBasicAuthentication] // Enable Basic authentication for this controller.
[Authorize] // Require authenticated requests.
public class HomeController : ApiController
{
    public IHttpActionResult Get() { . . . }
    public IHttpActionResult Post() { . . . }
}
```

To apply the filter to one action, decorate the action with the filter. The following code sets the `[IdentityBasicAuthentication]` filter on the controller's `Post` method.

```
[Authorize] // Require authenticated requests.
public class HomeController : ApiController
{
    public IHttpActionResult Get() { . . . }

    [IdentityBasicAuthentication] // Enable Basic authentication for this action.
    public IHttpActionResult Post() { . . . }
}
```

To apply the filter to all Web API controllers, add it to **GlobalConfiguration.Filters**.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Filters.Add(new IdentityBasicAuthenticationAttribute());

        // Other configuration code not shown...
    }
}
```

Implementing a Web API Authentication Filter

In Web API, authentication filters implement the System.Web.Http.Filters.IAuthenticationFilter interface. They should also inherit from **System.Attribute**, in order to be applied as attributes.

The **IAuthenticationFilter** interface has two methods:

- **AuthenticateAsync** authenticates the request by validating credentials in the request, if present.
- **ChallengeAsync** adds an authentication challenge to the HTTP response, if needed.

These methods correspond to the authentication flow defined in RFC 2612 and RFC 2617:

1. The client sends credentials in the Authorization header. This typically happens after the client receives a 401 (Unauthorized) response from the server. However, a client can send credentials with any request, not just after getting a 401.
2. If the server does not accept the credentials, it returns a 401 (Unauthorized) response. The response includes a Www-Authenticate header that contains one or more challenges. Each challenge specifies an authentication scheme recognized by the server.

The server can also return 401 from an anonymous request. In fact, that's typically how the authentication process is initiated:

1. The client sends an anonymous request.
2. The server returns 401.
3. The clients resends the request with credentials.

This flow includes both *authentication* and *authorization* steps.

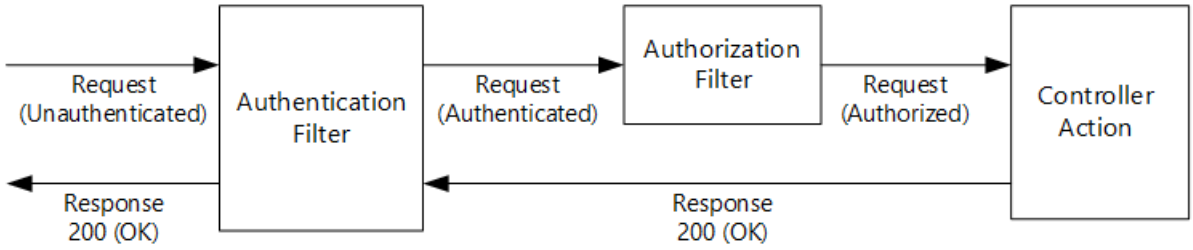
- Authentication proves the identity of the client.
- Authorization determines whether the client can access a particular resource.

In Web API, authentication filters handle authentication, but not authorization. Authorization should be done by an authorization filter or inside the controller action.

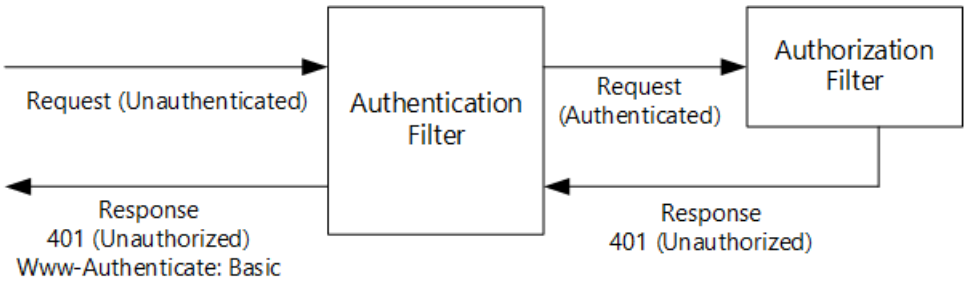
Here is the flow in the Web API 2 pipeline:

1. Before invoking an action, Web API creates a list of the authentication filters for that action. This includes filters with action scope, controller scope, and global scope.
2. Web API calls **AuthenticateAsync** on every filter in the list. Each filter can validate credentials in the request. If any filter successfully validates credentials, the filter creates an **IPrincipal** and attaches it to the request. A filter can also trigger an error at this point. If so, the rest of the pipeline does not run.
3. Assuming there is no error, the request flows through the rest of the pipeline.
4. Finally, Web API calls every authentication filter's **ChallengeAsync** method. Filters use this method to add a challenge to the response, if needed. Typically (but not always) that would happen in response to a 401 error.

The following diagrams show two possible cases. In the first, the authentication filter successfully authenticates the request, an authorization filter authorizes the request, and the controller action returns 200 (OK).



In the second example, the authentication filter authenticates the request, but the authorization filter returns 401 (Unauthorized). In this case, the controller action is not invoked. The authentication filter adds a Www-Authenticate header to the response.



Other combinations are possible—for example, if the controller action allows anonymous requests, you might have an authentication filter but no authorization.

Implementing the AuthenticateAsync Method

The **AuthenticateAsync** method tries to authenticate the request. Here is the method signature:

```
Task AuthenticateAsync(
    HttpAuthenticationContext context,
    CancellationToken cancellationToken
)
```

The **AuthenticateAsync** method must do one of the following:

1. Nothing (no-op).
2. Create an **IPrincipal** and set it on the request.
3. Set an error result.

Option (1) means the request did not have any credentials that the filter understands. Option (2) means the filter successfully authenticated the request. Option (3) means the request had invalid credentials (like the wrong password), which triggers an error response.

Here is a general outline for implementing **AuthenticateAsync**.

1. Look for credentials in the request.
2. If there are no credentials, do nothing and return (no-op).
3. If there are credentials but the filter does not recognize the authentication scheme, do nothing and return (no-op). Another filter in the pipeline might understand the scheme.
4. If there are credentials that the filter understands, try to authenticate them.
5. If the credentials are bad, return 401 by setting `context.ErrorResult`.
6. If the credentials are valid, create an **IPrincipal** and set `context.Principal`.

The follow code shows the **AuthenticateAsync** method from the Basic Authentication sample. The comments indicate each step. The code shows several types of error: An Authorization header with no credentials, malformed credentials, and bad username/password.

```
public async Task AuthenticateAsync(HttpAuthenticationContext context, CancellationToken cancellationToken)
{
    // 1. Look for credentials in the request.
    HttpRequestMessage request = context.Request;
    AuthenticationHeaderValue authorization = request.Headers.Authorization;

    // 2. If there are no credentials, do nothing.
    if (authorization == null)
    {
        return;
    }

    // 3. If there are credentials but the filter does not recognize the
    //    authentication scheme, do nothing.
    if (authorization.Scheme != "Basic")
    {
        return;
    }

    // 4. If there are credentials that the filter understands, try to validate them.
    // 5. If the credentials are bad, set the error result.
    if (String.IsNullOrEmpty(authorization.Parameter))
    {
        context.ErrorResult = new AuthenticationFailureResult("Missing credentials", request);
        return;
    }

    Tuple<string, string> userNameAndPassword = ExtractUserNameAndPassword(authorization.Parameter);
    if (userNameAndPassword == null)
    {
        context.ErrorResult = new AuthenticationFailureResult("Invalid credentials", request);
    }

    string userName = userNameAndPassword.Item1;
    string password = userNameAndPassword.Item2;

    IPrincipal principal = await AuthenticateAsync(userName, password, cancellationToken);
    if (principal == null)
    {
        context.ErrorResult = new AuthenticationFailureResult("Invalid username or password", request);
    }

    // 6. If the credentials are valid, set principal.
    else
    {
        context.Principal = principal;
    }

}
```

Setting an Error Result

If the credentials are invalid, the filter must set `context.ErrorResult` to an **IHttpActionResult** that creates an error response. For more information about **IHttpActionResult**, see Action Results in Web API 2.

The Basic Authentication sample includes an `AuthenticationFailureResult` class that is suitable for this purpose.

```
public class AuthenticationFailureResult : IHttpActionResult
{
    public AuthenticationFailureResult(string reasonPhrase, HttpRequestMessage request)
    {
        ReasonPhrase = reasonPhrase;
        Request = request;
    }

    public string ReasonPhrase { get; private set; }

    public HttpRequestMessage Request { get; private set; }

    public Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
    {
        return Task.FromResult(Execute());
    }

    private HttpResponseMessage Execute()
    {
        HttpResponseMessage response = new HttpResponseMessage(HttpStatusCode.Unauthorized);
        response.RequestMessage = Request;
        response.ReasonPhrase = ReasonPhrase;
        return response;
    }
}
```

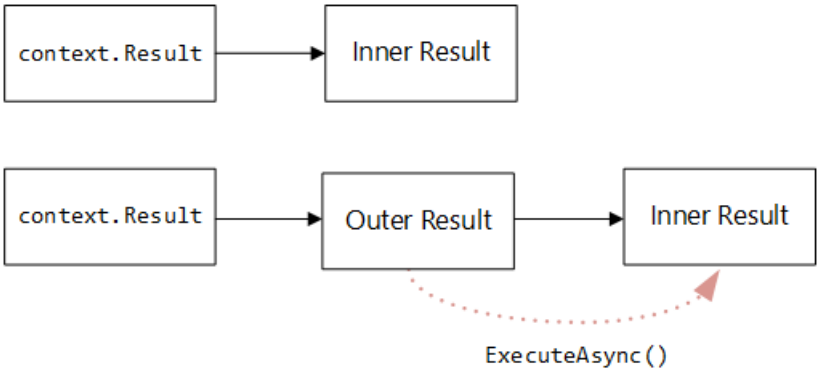
Implementing ChallengeAsync

The purpose of the **ChallengeAsync** method is to add authentication challenges to the response, if needed. Here is the method signature:

```
Task ChallengeAsync(
    HttpAuthenticationChallengeContext context,
    CancellationToken cancellationToken
)
```

The method is called on every authentication filter in the request pipeline.

It's important to understand that **ChallengeAsync** is called *before* the HTTP response is created, and possibly even before the controller action runs. When **ChallengeAsync** is called, `context.Result` contains an **IHttpActionResult**, which is used later to create the HTTP response. So when **ChallengeAsync** is called, you don't know anything about the HTTP response yet. The **ChallengeAsync** method should replace the original value of `context.Result` with a new **IHttpActionResult**. This **IHttpActionResult** must wrap the original `context.Result`.



I'll call the original **IHttpActionResult** the *inner result*, and the new **IHttpActionResult** the *outer result*. The outer result must do the following:

1. Invoke the inner result to create the HTTP response.
2. Examine the response.
3. Add an authentication challenge to the response, if needed.

The following example is taken from the Basic Authentication sample. It defines an **IHttpActionResult** for the outer result.

```
public class AddChallengeOnUnauthorizedResult : IHttpActionResult
{
    public AddChallengeOnUnauthorizedResult(AuthenticationHeaderValue challenge, IHttpActionResult
innerResult)
    {
        Challenge = challenge;
        InnerResult = innerResult;
    }

    public AuthenticationHeaderValue Challenge { get; private set; }

    public IHttpActionResult InnerResult { get; private set; }

    public async Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
    {
        HttpResponseMessage response = await InnerResult.ExecuteAsync(cancellationToken);

        if (response.StatusCode == HttpStatusCode.Unauthorized)
        {
            // Only add one challenge per authentication scheme.
            if (!response.Headers.WwwAuthenticate.Any((h) => h.Scheme == Challenge.Scheme))
            {
                response.Headers.WwwAuthenticate.Add(Challenge);
            }
        }

        return response;
    }
}
```

The `InnerResult` property holds the inner **IHttpActionResult**. The `Challenge` property represents a Www-Authentication header. Notice that **ExecuteAsync** first calls `InnerResult.ExecuteAsync` to create the HTTP response, and then adds the challenge if needed.

Check the response code before adding the challenge. Most authentication schemes only add a challenge if the response is 401, as shown here. However, some authentication schemes do add a challenge to a success response. For example, see Negotiate (RFC 4559).

Given the `AddChallengeOnUnauthorizedResult` class, the actual code in **ChallengeAsync** is simple. You just create the result and attach it to `context.Result`.

```
public Task ChallengeAsync(HttpAuthenticationChallengeContext context, CancellationToken cancellationToken)
{
    var challenge = new AuthenticationHeaderValue("Basic");
    context.Result = new AddChallengeOnUnauthorizedResult(challenge, context.Result);
    return Task.FromResult(0);
}
```

Note: The Basic Authentication sample abstracts this logic a bit, by placing it in an extension method.

Combining Authentication Filters with Host-Level Authentication

"Host-level authentication" is authentication performed by the host (such as IIS), before the request reaches the

Web API framework.

Often, you may want to enable host-level authentication for the rest of your application, but disable it for your Web API controllers. For example, a typical scenario is to enable Forms Authentication at the host level, but use token-based authentication for Web API.

To disable host-level authentication inside the Web API pipeline, call `config.SuppressHostPrincipal()` in your configuration. This causes Web API to remove the **IPrincipal** from any request that enters the Web API pipeline. Effectively, it "un-authenticates" the request.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.SuppressHostPrincipal();

        // Other configuration code not shown...
    }
}
```

Additional Resources

ASP.NET Web API Security Filters (MSDN Magazine)

Basic Authentication in ASP.NET Web API

2/20/2020 • 3 minutes to read • Edit Online

by Mike Wasson

Basic authentication is defined in RFC 2617, HTTP Authentication: Basic and Digest Access Authentication.

Disadvantages

- User credentials are sent in the request.
- Credentials are sent as plaintext.
- Credentials are sent with every request.
- No way to log out, except by ending the browser session.
- Vulnerable to cross-site request forgery (CSRF); requires anti-CSRF measures.

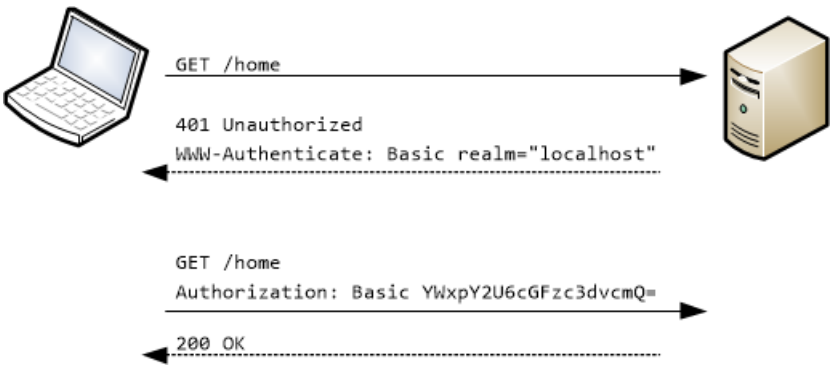
Advantages

- Internet standard.
- Supported by all major browsers.
- Relatively simple protocol.

Basic authentication works as follows:

1. If a request requires authentication, the server returns 401 (Unauthorized). The response includes a WWW-Authenticate header, indicating the server supports Basic authentication.
2. The client sends another request, with the client credentials in the Authorization header. The credentials are formatted as the string "name:password", base64-encoded. The credentials are not encrypted.

Basic authentication is performed within the context of a "realm." The server includes the name of the realm in the WWW-Authenticate header. The user's credentials are valid within that realm. The exact scope of a realm is defined by the server. For example, you might define several realms in order to partition resources.



Because the credentials are sent unencrypted, Basic authentication is only secure over HTTPS. See Working with SSL in Web API.

Basic authentication is also vulnerable to CSRF attacks. After the user enters credentials, the browser automatically sends them on subsequent requests to the same domain, for the duration of the session. This includes AJAX requests. See Preventing Cross-Site Request Forgery (CSRF) Attacks.

Basic Authentication with IIS

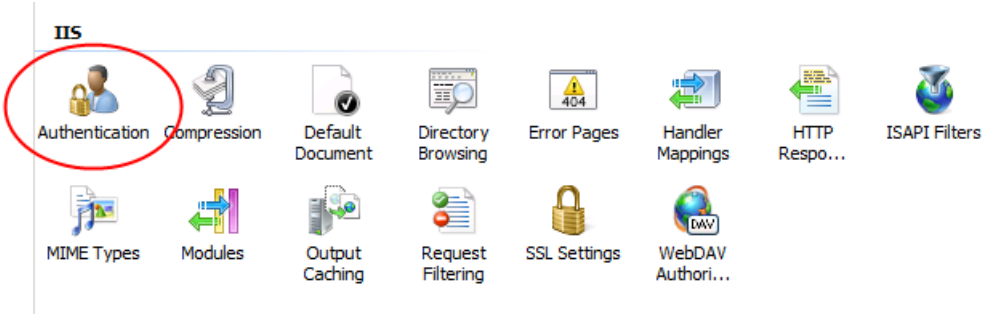
IIS supports Basic authentication, but there is a caveat: The user is authenticated against their Windows credentials.

That means the user must have an account on the server's domain. For a public-facing web site, you typically want to authenticate against an ASP.NET membership provider.

To enable Basic authentication using IIS, set the authentication mode to "Windows" in the Web.config of your ASP.NET project:

```
<system.web>
    <authentication mode="Windows" />
</system.web>
```

In this mode, IIS uses Windows credentials to authenticate. In addition, you must enable Basic authentication in IIS. In IIS Manager, go to Features View, select Authentication, and enable Basic authentication.



In your Web API project, add the `[Authorize]` attribute for any controller actions that need authentication.

A client authenticates itself by setting the Authorization header in the request. Browser clients perform this step automatically. Nonbrowser clients will need to set the header.

Basic Authentication with Custom Membership

As mentioned, the Basic Authentication built into IIS uses Windows credentials. That means you need to create accounts for your users on the hosting server. But for an internet application, user accounts are typically stored in an external database.

The following code how an HTTP module that performs Basic Authentication. You can easily plug in an ASP.NET membership provider by replacing the `CheckPassword` method, which is a dummy method in this example.

In Web API 2, you should consider writing an authentication filter or OWIN middleware, instead of an HTTP module.

```
namespace WebHostBasicAuth.Modules
{
    public class BasicAuthHttpModule : IHttpModule
    {
        private const string Realm = "My Realm";

        public void Init(HttpApplication context)
        {
            // Register event handlers
            context.AuthenticateRequest += OnApplicationAuthenticateRequest;
            context.EndRequest += OnApplicationEndRequest;
        }

        private static void SetPrincipal(IPrincipal principal)
        {
            Thread.CurrentPrincipal = principal;
            if (HttpContext.Current != null)
            {
                HttpContext.Current.User = principal;
            }
        }
```

```
        // TODO: Here is where you would validate the username and password.
        private static bool CheckPassword(string username, string password)
        {
            return username == "user" && password == "password";
        }

        private static void AuthenticateUser(string credentials)
        {
            try
            {
                var encoding = Encoding.GetEncoding("iso-8859-1");
                credentials = encoding.GetString(Convert.FromBase64String(credentials));

                int separator = credentials.IndexOf(':');
                string name = credentials.Substring(0, separator);
                string password = credentials.Substring(separator + 1);

                if (CheckPassword(name, password))
                {
                    var identity = new GenericIdentity(name);
                    SetPrincipal(new GenericPrincipal(identity, null));
                }
                else
                {
                    // Invalid username or password.
                    HttpContext.Current.Response.StatusCode = 401;
                }
            }
            catch (FormatException)
            {
                // Credentials were not formatted correctly.
                HttpContext.Current.Response.StatusCode = 401;
            }
        }

        private static void OnApplicationAuthenticateRequest(object sender, EventArgs e)
        {
            var request = HttpContext.Current.Request;
            var authHeader = request.Headers["Authorization"];
            if (authHeader != null)
            {
                var authHeaderVal = AuthenticationHeaderValue.Parse(authHeader);

                // RFC 2617 sec 1.2, "scheme" name is case-insensitive
                if (authHeaderVal.Scheme.Equals("basic",
                        StringComparison.OrdinalIgnoreCase) &&
                    authHeaderVal.Parameter != null)
                {
                    AuthenticateUser(authHeaderVal.Parameter);
                }
            }
        }

        // If the request was unauthorized, add the WWW-Authenticate header
        // to the response.
        private static void OnApplicationEndRequest(object sender, EventArgs e)
        {
            var response = HttpContext.Current.Response;
            if (response.StatusCode == 401)
            {
                response.Headers.Add("WWW-Authenticate",
                    string.Format("Basic realm=\"{0}\"", Realm));
            }
        }

        public void Dispose()
        {
        }
```

```
    }
}
```

To enable the HTTP module, add the following to your web.config file in the **system.webServer** section:

```
<system.webServer>
    <modules>
      <add name="BasicAuthHttpModule"
        type="WebHostBasicAuth.Modules.BasicAuthHttpModule, YourAssemblyName"/>
    </modules>
```

Replace "YourAssemblyName" with the name of the assembly (not including the "dll" extension).

You should disable other authentication schemes, such as Forms or Windows auth.

Forms Authentication in ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

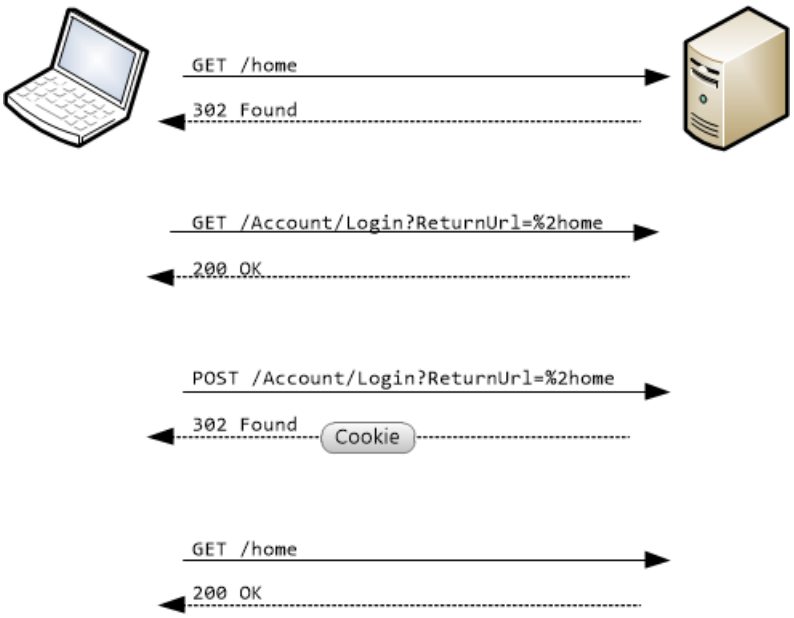
by Mike Wasson

Forms authentication uses an HTML form to send the user's credentials to the server. It is not an Internet standard. Forms authentication is only appropriate for web APIs that are called from a web application, so that the user can interact with the HTML form.

ADVANTAGES	DISADVANTAGES
- Easy to implement: Built into ASP.NET. - Uses ASP.NET membership provider, which makes it easy to manage user accounts.	- Not a standard HTTP authentication mechanism; uses HTTP cookies instead of the standard Authorization header. - Requires a browser client. - Credentials are sent as plaintext. - Vulnerable to cross-site request forgery (CSRF); requires anti-CSRF measures. - Difficult to use from nonbrowser clients. Login requires a browser. - User credentials are sent in the request. - Some users disable cookies.

Briefly, forms authentication in ASP.NET works like this:

1. The client requests a resource that requires authentication.
2. If the user is not authenticated, the server returns HTTP 302 (Found) and redirects to a login page.
3. The user enters credentials and submits the form.
4. The server returns another HTTP 302 that redirects back to the original URI. This response includes an authentication cookie.
5. The client requests the resource again. The request includes the authentication cookie, so the server grants the request.



For more information, see An Overview of Forms Authentication.

Using Forms Authentication with Web API

To create an application that uses forms authentication, select the "Internet Application" template in the MVC 4 project wizard. This template creates MVC controllers for account management. You can also use the "Single Page

Application" template, available in the ASP.NET Fall 2012 Update.

In your web API controllers, you can restrict access by using the `[Authorize]` attribute, as described in Using the [Authorize] Attribute.

Forms-authentication uses a session cookie to authenticate requests. Browsers automatically send all relevant cookies to the destination web site. This feature makes forms authentication potentially vulnerable to cross-site request forgery (CSRF) attacks See Preventing Cross-Site Request Forgery (CSRF) Attacks.

Forms authentication does not encrypt the user's credentials. Therefore, forms authentication is not secure unless used with SSL. See Working with SSL in Web API.

Integrated Windows Authentication

9/16/2020 • 2 minutes to read • Edit Online

by Mike Wasson

Integrated Windows authentication enables users to log in with their Windows credentials, using Kerberos or NTLM. The client sends credentials in the Authorization header. Windows authentication is best suited for an intranet environment. For more information, see Windows Authentication.

ADVANTAGES	DISADVANTAGES
Built into IIS.	Not recommended for Internet applications.
Does not send the user credentials in the request.	Requires Kerberos or NTLM support in the client.
If the client computer belongs to the domain (for example, intranet application), the user does not need to enter credentials.	Client must be in the Active Directory domain.

NOTE

If your application is hosted on Azure and you have an on-premise Active Directory domain, consider federating your on-premise AD with Azure Active Directory. That way, users can log in with their on-premise credentials, but the authentication is performed by Azure AD. For more information, see Azure Authentication.

To create an application that uses Integrated Windows authentication, select the "Intranet Application" template in the MVC 4 project wizard. This project template puts the following setting in the Web.config file:

```
<system.web>
    <authentication mode="Windows" />
</system.web>
```

On the client side, Integrated Windows authentication works with any browser that supports the Negotiate authentication scheme, which includes most major browsers. For .NET client applications, the **HttpClient** class supports Windows authentication:

```
HttpClientHandler handler = new HttpClientHandler()
{
    UseDefaultCredentials = true
};

HttpClient client = new HttpClient(handler);
```

Windows authentication is vulnerable to cross-site request forgery (CSRF) attacks. See Preventing Cross-Site Request Forgery (CSRF) Attacks.

Working with SSL in Web API

2/20/2020 • 3 minutes to read • Edit Online

by Mike Wasson

Several common authentication schemes are not secure over plain HTTP. In particular, Basic authentication and forms authentication send unencrypted credentials. To be secure, these authentication schemes *must* use SSL. In addition, SSL client certificates can be used to authenticate clients.

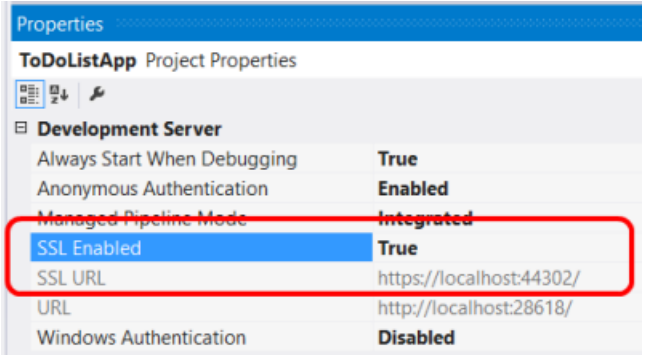
Enabling SSL on the Server

To set up SSL in IIS 7 or later:

- Create or get a certificate. For testing, you can create a self-signed certificate.
- Add an HTTPS binding.

For details, see How to Set Up SSL on IIS 7.

For local testing, you can enable SSL in IIS Express from Visual Studio. In the Properties window, set **SSL Enabled** to **True**. Note the value of **SSL URL**; use this URL for testing HTTPS connections.



Enforcing SSL in a Web API Controller

If you have both an HTTPS and an HTTP binding, clients can still use HTTP to access the site. You might allow some resources to be available through HTTP, while other resources require SSL. In that case, use an action filter to require SSL for the protected resources. The following code shows a Web API authentication filter that checks for SSL:

```
public class RequireHttpsAttribute : AuthorizationFilterAttribute
{
    public override void OnAuthorization(HttpActionContext actionContext)
    {
        if (actionContext.Request.RequestUri.Scheme != Uri.UriSchemeHttps)
        {
            actionContext.Response = new HttpResponseMessage(System.Net.HttpStatusCode.Forbidden)
            {
                ReasonPhrase = "HTTPS Required"
            };
        }
        else
        {
            base.OnAuthorization(actionContext);
        }
    }
}
```

Add this filter to any Web API actions that require SSL:

```
public class ValuesController : ApiController
{
    [RequireHttps]
    public HttpResponseMessage Get() { ... }
}
```

SSL Client Certificates

SSL provides authentication by using Public Key Infrastructure certificates. The server must provide a certificate that authenticates the server to the client. It is less common for the client to provide a certificate to the server, but this is one option for authenticating clients. To use client certificates with SSL, you need a way to distribute signed certificates to your users. For many application types, this will not be a good user experience, but in some environments (for example, enterprise) it may be feasible.

ADVANTAGES	DISADVANTAGES
- Certificate credentials are stronger than username/password. - SSL provides a complete secure channel, with authentication, message integrity, and message encryption.	- You must obtain and manage PKI certificates. - The client platform must support SSL client certificates.

To configure IIS to accept client certificates, open IIS Manager and perform the following steps:

1. Click the site node in the tree view.
2. Double-click the **SSL Settings** feature in the middle pane.
3. Under **Client Certificates**, select one of these options:
   - **Accept**: IIS will accept a certificate from the client, but does not require one.
   - **Require**: Require a client certificate. (To enable this option, you must also select "Require SSL")

You can also set these options in the ApplicationHost.config file:

```
<system.webServer>
    <security>
        <access sslFlags="Ssl, SslNegotiateCert" />
        <!-- To require a client cert: -->
        <!-- <access sslFlags="Ssl, SslRequireCert" /> -->
    </security>
</system.webServer>
```

The **SslNegotiateCert** flag means IIS will accept a certificate from the client, but does not require one (equivalent to the "Accept" option in IIS Manager). To require a certificate, set the **SslRequireCert** flag. For testing, you can also set these options in IIS Express, in the local applicationhost.Config file, located in "Documents\IISExpress\config".

Creating a Client Certificate for Testing

For testing purposes, you can use MakeCert.exe to create a client certificate. First, create a test root authority:

```
makecert.exe -n "CN=Development CA" -r -sv TempCA.pvk TempCA.cer
```

Makecert will prompt you to enter a password for the private key.

Next, add the certificate to the test server's "Trusted Root Certification Authorities" store, as follows:

1. Open MMC.
2. Under **File**, select **Add/Remove Snap-In**.
3. Select **Computer Account**.
4. Select **Local computer** and complete the wizard.
5. Under the navigation pane, expand the "Trusted Root Certification Authorities" node.
6. On the **Action** menu, point to **All Tasks**, and then click **Import** to start the Certificate Import Wizard.
7. Browse to the certificate file, TempCA.cer.
8. Click **Open**, then click **Next** and complete the wizard. (You will be prompted to re-enter the password.)

Now create a client certificate that is signed by the first certificate:

```
makecert.exe -pe -ss My -sr CurrentUser -a sha1 -sky exchange -n "CN=name"
     -eku 1.3.6.1.5.5.7.3.2 -sk SignedByCA -ic TempCA.cer -iv TempCA.pvk
```

Using Client Certificates in Web API

On the server side, you can get the client certificate by calling GetClientCertificate on the request message. The method returns null if there is no client certificate. Otherwise, it returns an **X509Certificate2** instance. Use this object to get information from the certificate, such as the issuer and subject. Then you can use this information for authentication and/or authorization.

```
X509Certificate2 cert = Request.GetClientCertificate();
string issuer = cert.Issuer;
string subject = cert.Subject;
```

ASP.NET Web API: Deployment and Hosting

2/20/2020 • 2 minutes to read • Edit Online

How to host your web API.

- Use OWIN to Self-Host Web API 2
- Host Web API 2 in an Azure Worker Role

Use OWIN to Self-Host ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

This tutorial shows how to host ASP.NET Web API in a console application, using OWIN to self-host the Web API framework.

Open Web Interface for .NET (OWIN) defines an abstraction between .NET web servers and web applications. OWIN decouples the web application from the server, which makes OWIN ideal for self-hosting a web application in your own process, outside of IIS.

Software versions used in the tutorial

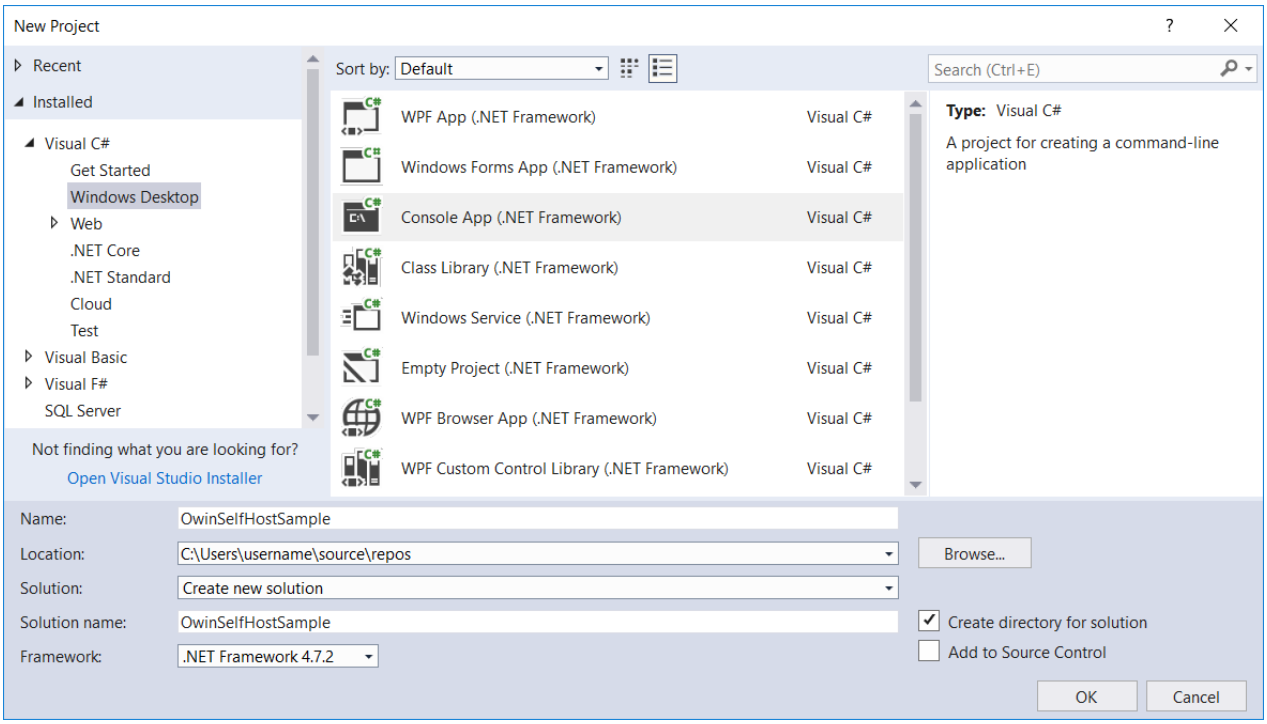
- Visual Studio 2017
- Web API 5.2.7

NOTE

You can find the complete source code for this tutorial at github.com/aspnet/samples.

Create a console application

On the **File** menu, **New**, then select **Project**. From **Installed**, under **Visual C#**, select **Windows Desktop** and then select **Console App (.Net Framework)**. Name the project "OwinSelfhostSample" and select **OK**.



Add the Web API and OWIN packages

From the **Tools** menu, select **NuGet Package Manager**, then select **Package Manager Console**. In the Package Manager Console window, enter the following command:

```
Install-Package Microsoft.AspNet.WebApi.OwinSelfHost
```

This will install the WebAPI OWIN selfhost package and all the required OWIN packages.

```
Package Manager Console
Package source: nuget.org    Default project: OwinSelfHostSample
Each package is licensed to you by its owner. Microsoft is not responsible for, no
which are governed by additional licenses. Follow the package source (feed) URL to

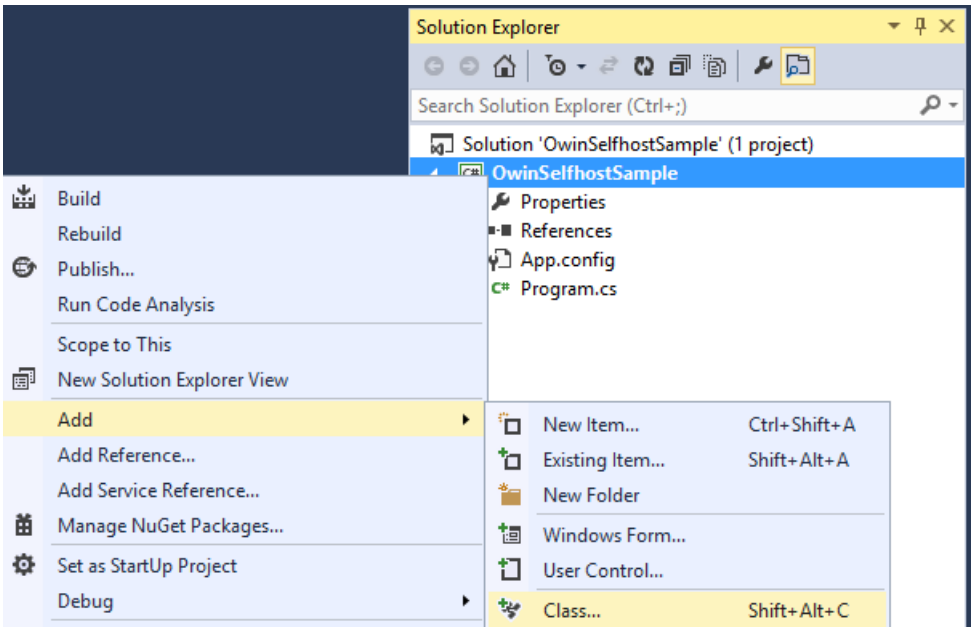
Package Manager Console Host Version 2.7.40911.287

Type 'get-help NuGet' to see all available NuGet commands.

PM> Install-Package Microsoft.AspNet.WebApi.OwinSelfHost
```

Configure Web API for self-host

In Solution Explorer, right-click the project and select **Add** / **Class** to add a new class. Name the class `Startup`.



Replace all of the boilerplate code in this file with the following:

```
using Owin;
using System.Web.Http;

namespace OwinSelfhostSample
{
    public class Startup
    {
        // This code configures Web API. The Startup class is specified as a type
        // parameter in the WebApp.Start method.
        public void Configuration(IAppBuilder appBuilder)
        {
            // Configure Web API for self-host.
            HttpConfiguration config = new HttpConfiguration();
            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );

            appBuilder.UseWebApi(config);
        }
    }
}
```

Add a Web API controller

Next, add a Web API controller class. In Solution Explorer, right-click the project and select **Add** / **Class** to add a new class. Name the class `ValuesController`.

Replace all of the boilerplate code in this file with the following:

```
using System.Collections.Generic;
using System.Web.Http;

namespace OwinSelfhostSample
{
    public class ValuesController : ApiController
    {
        // GET api/values
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET api/values/5
        public string Get(int id)
        {
            return "value";
        }

        // POST api/values
        public void Post([FromBody]string value)
        {
        }

        // PUT api/values/5
        public void Put(int id, [FromBody]string value)
        {
        }

        // DELETE api/values/5
        public void Delete(int id)
        {
        }
    }
}
```

Start the OWIN Host and make a request with HttpClient

Replace all of the boilerplate code in the Program.cs file with the following:

```
using Microsoft.Owin.Hosting;
using System;
using System.Net.Http;

namespace OwinSelfhostSample
{
    public class Program
    {
        static void Main()
        {
            string baseAddress = "http://localhost:9000/";

            // Start OWIN host
            using (WebApp.Start<Startup>(url: baseAddress))
            {
                // Create HttpClient and make a request to api/values
                HttpClient client = new HttpClient();

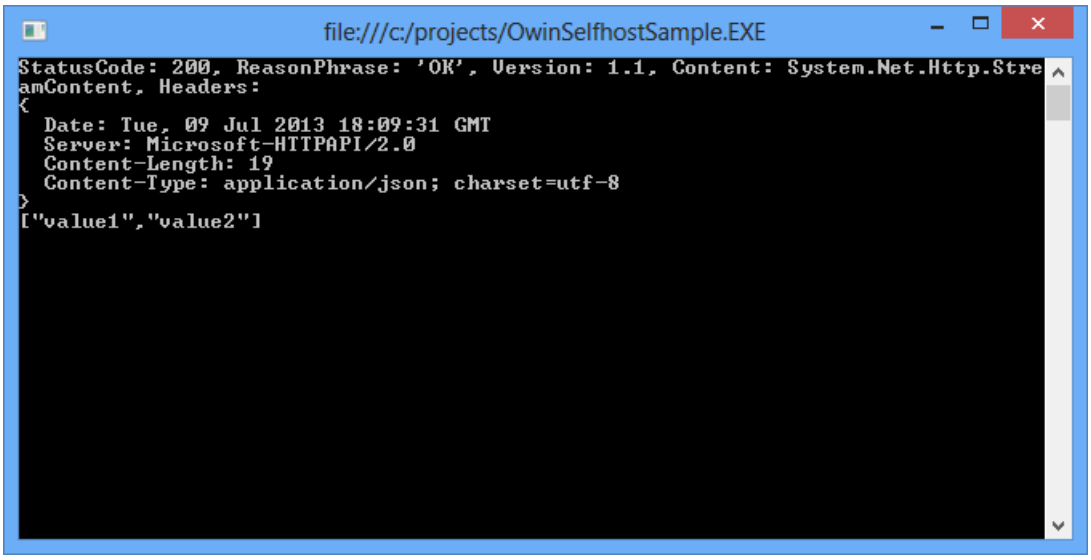
                var response = client.GetAsync(baseAddress + "api/values").Result;

                Console.WriteLine(response);
                Console.WriteLine(response.Content.ReadAsStringAsync().Result);
                Console.ReadLine();
            }
        }
    }
 }
```

Run the application

To run the application, press F5 in Visual Studio. The output should look like the following:

```
StatusCode: 200, ReasonPhrase: 'OK', Version: 1.1, Content: System.Net.Http.StreamContent, Headers:
{
  Date: Tue, 09 Jul 2013 18:10:15 GMT
  Server: Microsoft-HTTPAPI/2.0
  Content-Length: 19
  Content-Type: application/json; charset=utf-8
}
["value1","value2"]
```



Additional resources

An Overview of Project Katana

Host ASP.NET Web API in an Azure Worker Role

Host ASP.NET Web API 2 in an Azure Worker Role

2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson

This tutorial shows how to host ASP.NET Web API in an Azure Worker Role, using OWIN to self-host the Web API framework.

Open Web Interface for .NET (OWIN) defines an abstraction between .NET web servers and web applications. OWIN decouples the web application from the server, which makes OWIN ideal for self-hosting a web application in your own process, outside of IIS–for example, inside an Azure worker role.

In this tutorial, you'll use the Microsoft.Owin.Host.HttpListener package, which provides an HTTP server that be used to self-host OWIN applications.

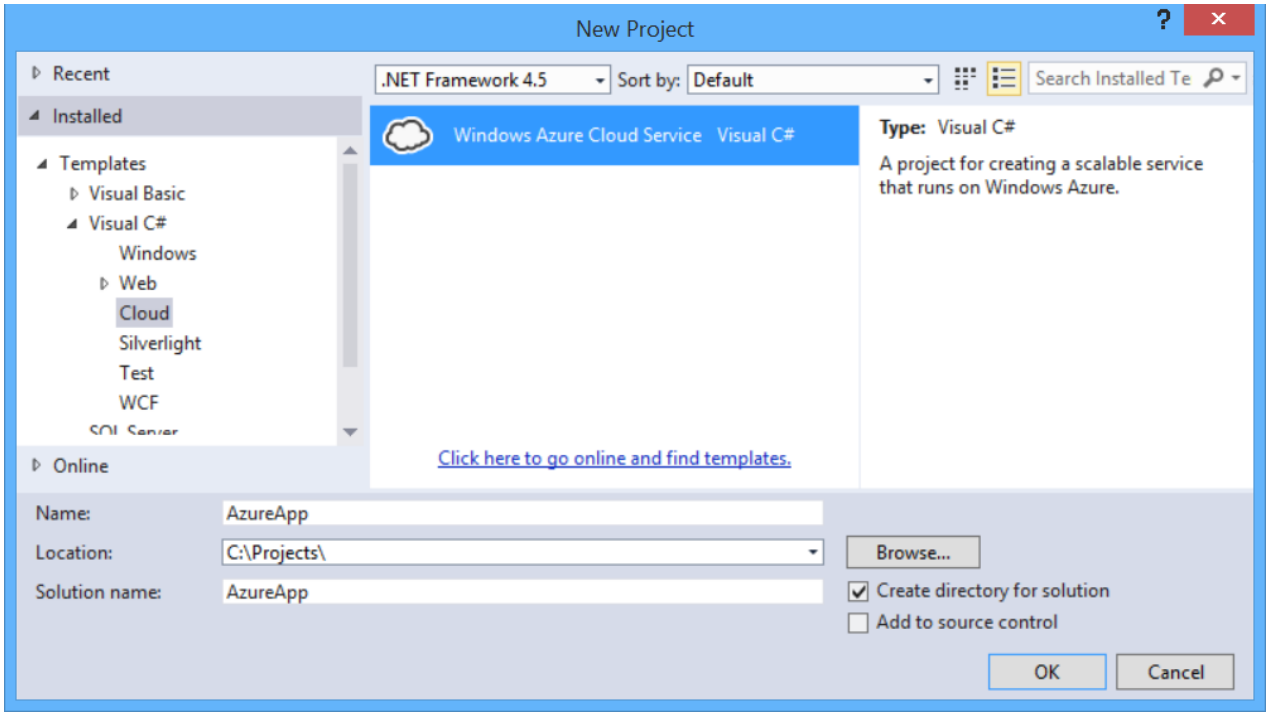
Software versions used in the tutorial

- Visual Studio 2013
- Web API 2
- Azure SDK for .NET 2.3

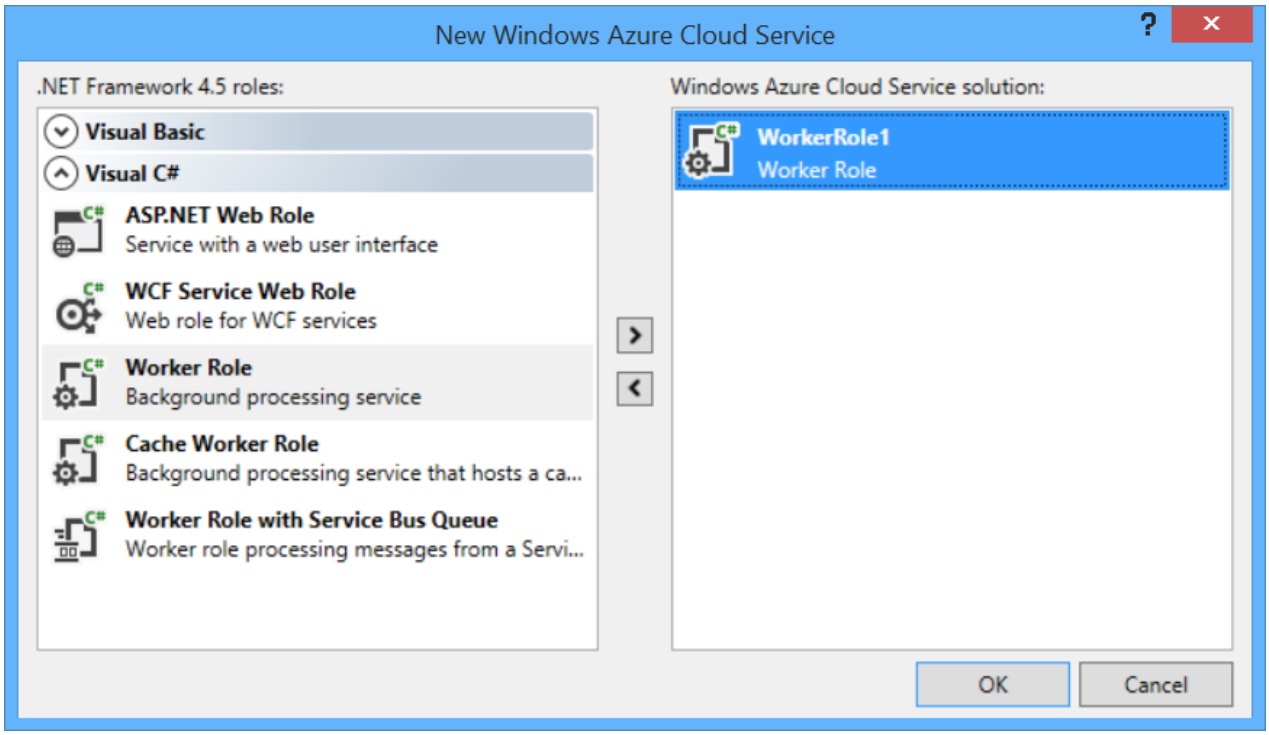
Create a Microsoft Azure Project

Start Visual Studio with administrator privileges. Administrator privileges are needed to debug the application locally, using the Azure compute emulator.

On the **File** menu, click **New**, then click **Project**. From **Installed Templates**, under Visual C#, click **Cloud** and then click **Windows Azure Cloud Service**. Name the project "AzureApp" and click **OK**.



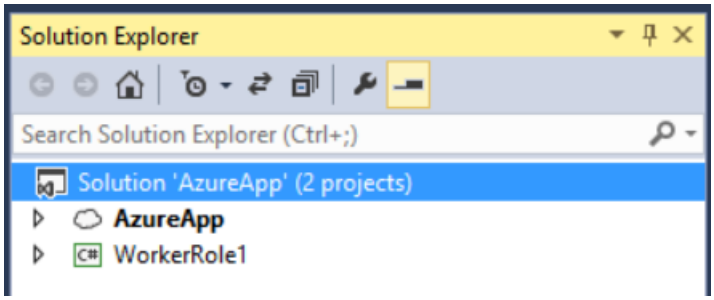
In the **New Windows Azure Cloud Service** dialog, double-click **Worker Role**. Leave the default name ("WorkerRole1"). This step adds a worker role to the solution. Click **OK**.



The Visual Studio solution that is created contains two projects:

- "AzureApp" defines the roles and configuration for the Azure application.
- "WorkerRole1" contains the code for the worker role.

In general, an Azure application can contain multiple roles, although this tutorial uses a single role.



Add the Web API and OWIN Packages

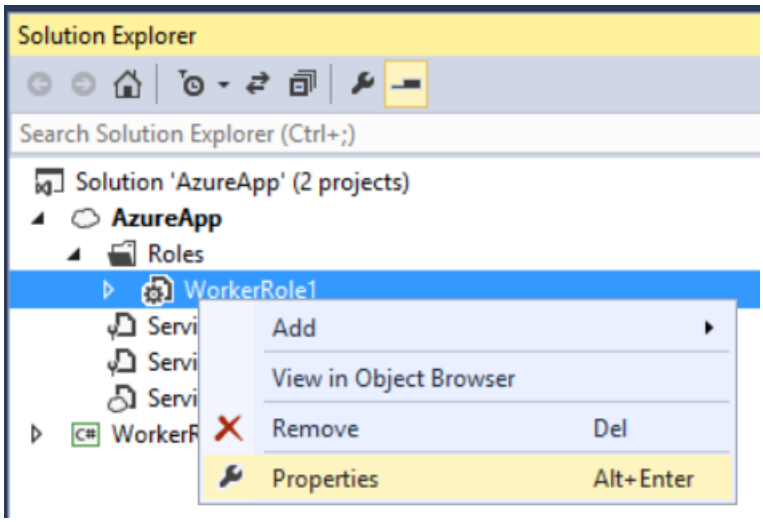
From the **Tools** menu, click **NuGet Package Manager**, then click **Package Manager Console**.

In the Package Manager Console window, enter the following command:

```
Install-Package Microsoft.AspNet.WebApi.OwinSelfHost
```

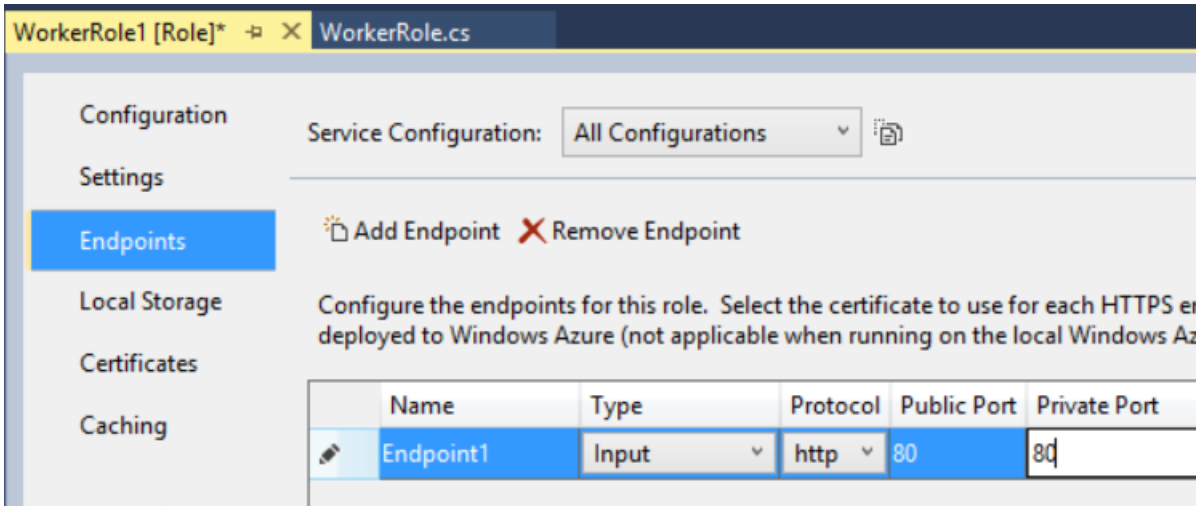
Add an HTTP Endpoint

In Solution Explorer, expand the AzureApp project. Expand the Roles node, right-click WorkerRole1, and select **Properties**.



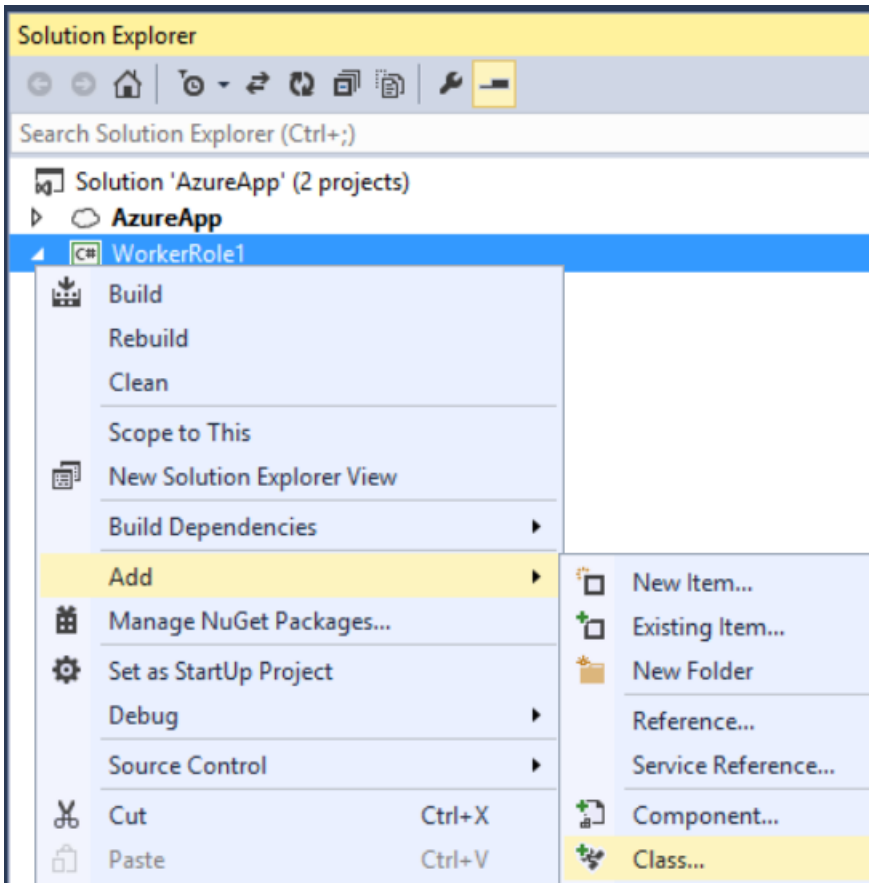
Click **Endpoints**, and then click **Add Endpoint**.

In the **Protocol** dropdown list, select "http". In **Public Port** and **Private Port**, type 80. These port numbers can be different. The public port is what clients use when they send a request to the role.



Configure Web API for Self-Host

In Solution Explorer, right click the WorkerRole1 project and select **Add** / **Class** to add a new class. Name the class `Startup`.



Replace all of the boilerplate code in this file with the following:

```
using Owin;
using System.Web.Http;

namespace WorkerRole1
{
    class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            HttpConfiguration config = new HttpConfiguration();
            config.Routes.MapHttpRoute(
                "Default",
                "{controller}/{id}",
                new { id = RouteParameter.Optional });

            app.UseWebApi(config);
        }
    }
}
```

Add a Web API Controller

Next, add a Web API controller class. Right-click the WorkerRole1 project and select **Add** / **Class**. Name the class TestController. Replace all of the boilerplate code in this file with the following:

```
using System;
using System.Net.Http;
using System.Web.Http;

namespace WorkerRole1
{
    public class TestController : ApiController
    {
        public HttpResponseMessage Get()
        {
            return new HttpResponseMessage()
            {
                Content = new StringContent("Hello from OWIN!")
            };
        }

        public HttpResponseMessage Get(int id)
        {
            string msg = String.Format("Hello from OWIN (id = {0})", id);
            return new HttpResponseMessage()
            {
                Content = new StringContent(msg)
            };
        }
    }
}
```

For simplicity, this controller just defines two GET methods that return plain text.

Start the OWIN Host

Open the WorkerRole.cs file. This class defines the code that runs when the worker role is started and stopped.

Add the following using statement:

```
using Microsoft.Owin.Hosting;
```

Add an **IDisposable** member to the `WorkerRole` class:

```
public class WorkerRole : RoleEntryPoint
{
    private IDisposable _app = null;

    // ....
}
```

In the `OnStart` method, add the following code to start the host:

```
public override bool OnStart()
{
    ServicePointManager.DefaultConnectionLimit = 12;

    // New code:
    var endpoint = RoleEnvironment.CurrentRoleInstance.InstanceEndpoints["Endpoint1"];
    string baseUri = String.Format("{0}://{1}",
        endpoint.Protocol, endpoint.IPEndpoint);

    Trace.TraceInformation(String.Format("Starting OWIN at {0}", baseUri),
        "Information");

    _app = WebApp.Start<Startup>(new StartOptions(url: baseUri));
    return base.OnStart();
}
```

The **WebApp.Start** method starts the OWIN host. The name of the `Startup` class is a type parameter to the method. By convention, the host will call the `Configure` method of this class.

Override the `OnStop` to dispose of the *_app* instance:

```
public override void OnStop()
{
    if (_app != null)
    {
        _app.Dispose();
    }
    base.OnStop();
}
```

Here is the complete code for WorkerRole.cs:

```
using Microsoft.Owin.Hosting;
using Microsoft.WindowsAzure.ServiceRuntime;
using System;
using System.Diagnostics;
using System.Net;
using System.Threading;

namespace WorkerRole1
{
    public class WorkerRole : RoleEntryPoint
    {
        private IDisposable _app = null;

        public override void Run()
        {
            Trace.TraceInformation("WebApiRole entry point called", "Information");

            while (true)
            {
                Thread.Sleep(10000);
                Trace.TraceInformation("Working", "Information");
            }
        }

        public override bool OnStart()
        {
            ServicePointManager.DefaultConnectionLimit = 12;

            var endpoint = RoleEnvironment.CurrentRoleInstance.InstanceEndpoints["Endpoint1"];
            string baseUri = String.Format("{0}://{1}",
                endpoint.Protocol, endpoint.IPEndpoint);

            Trace.TraceInformation(String.Format("Starting OWIN at {0}", baseUri),
                "Information");

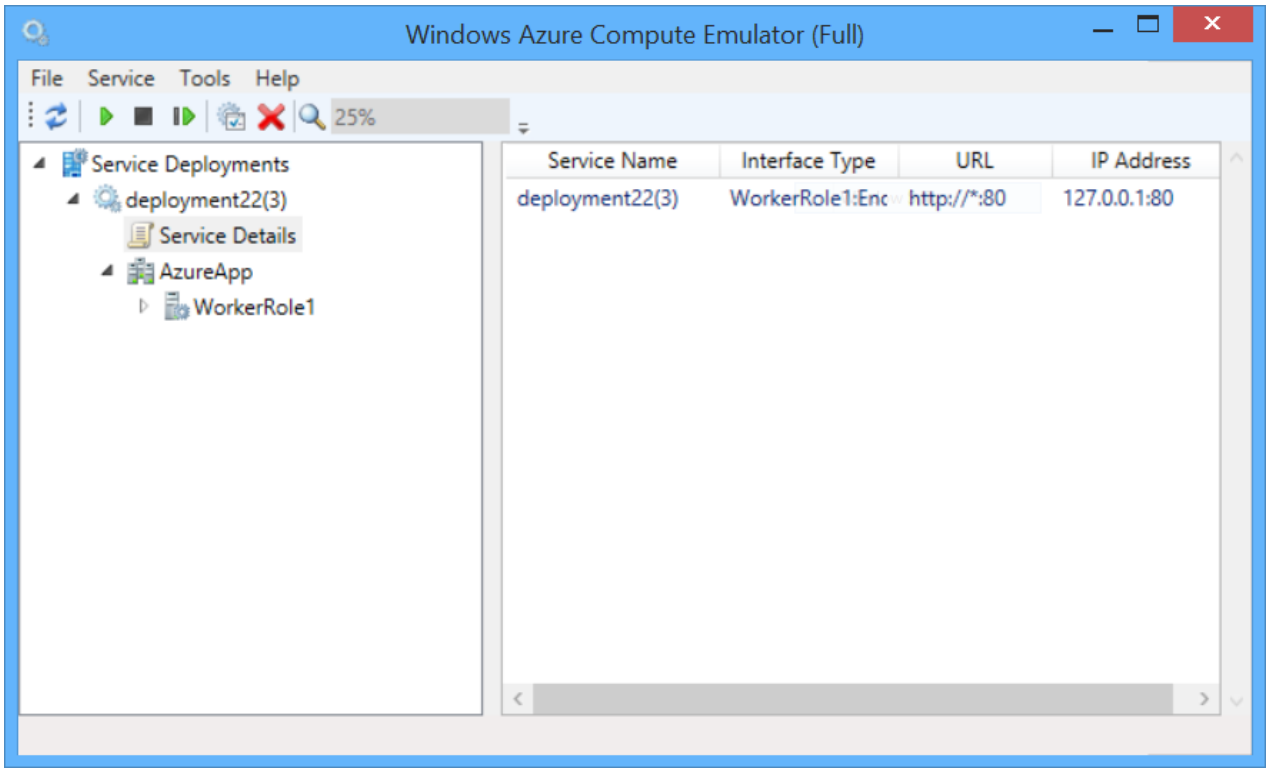
            _app = WebApp.Start<Startup>(new StartOptions(url: baseUri));
           return base.OnStart();
        }

        public override void OnStop()
        {
            if (_app != null)
            {
                _app.Dispose();
            }
            base.OnStop();
        }
    }
}
```

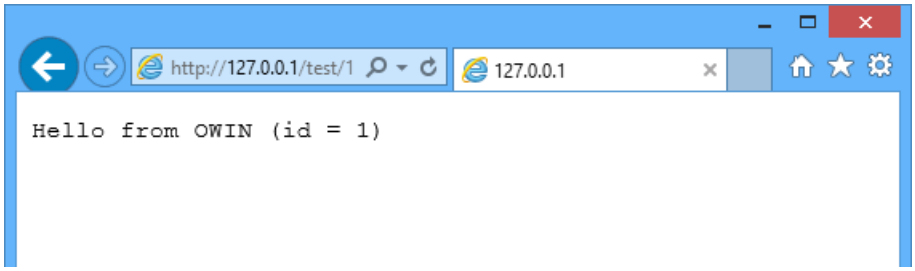
Build the solution, and press F5 to run the application locally in the Azure Compute Emulator. Depending on your firewall settings, you might need to allow the emulator through your firewall.

> **NOTE**
>
> If you get an exception like the following, please see this blog post for a workaround. "Could not load file or assembly 'Microsoft.Owin, Version=2.0.2.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35' or one of its dependencies. The located assembly's manifest definition does not match the assembly reference. (Exception from HRESULT: 0x80131040)"

The compute emulator assigns a local IP address to the endpoint. You can find the IP address by viewing the Compute Emulator UI. Right-click the emulator icon in the task bar notification area, and select **Show Compute Emulator UI**.



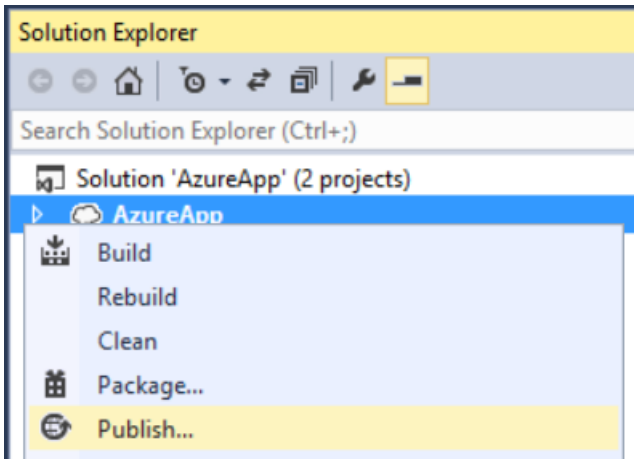
Find the IP address under Service Deployments, deployment [id], Service Details. Open a web browser and navigate to http://*address*/test/1, where *address* is the IP address assigned by the compute emulator; for example, `http://127.0.0.1:80/test/1`. You should see the response from the Web API controller:



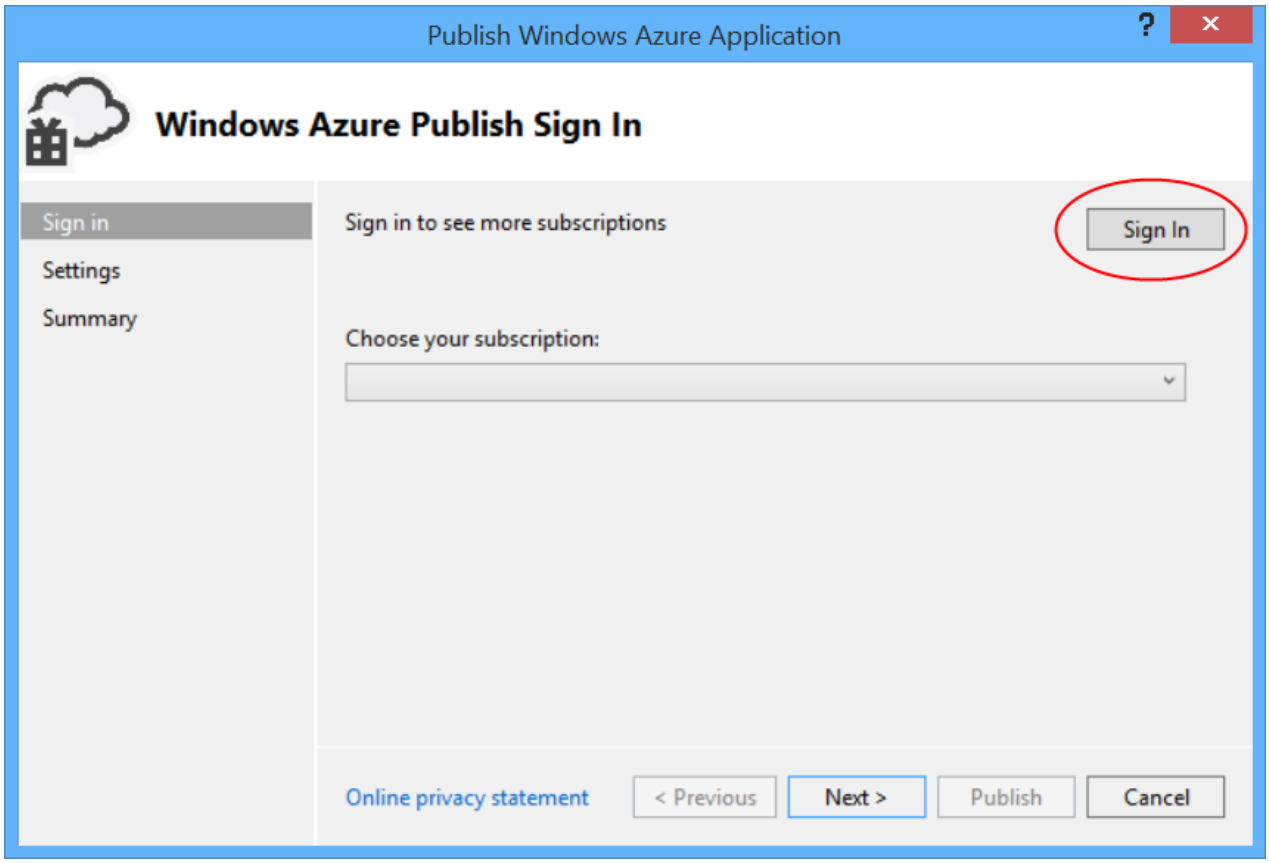
Deploy to Azure

For this step, you must have an Azure account. If you don't already have one, you can create a free trial account in just a couple of minutes. For details, see Microsoft Azure Free Trial.

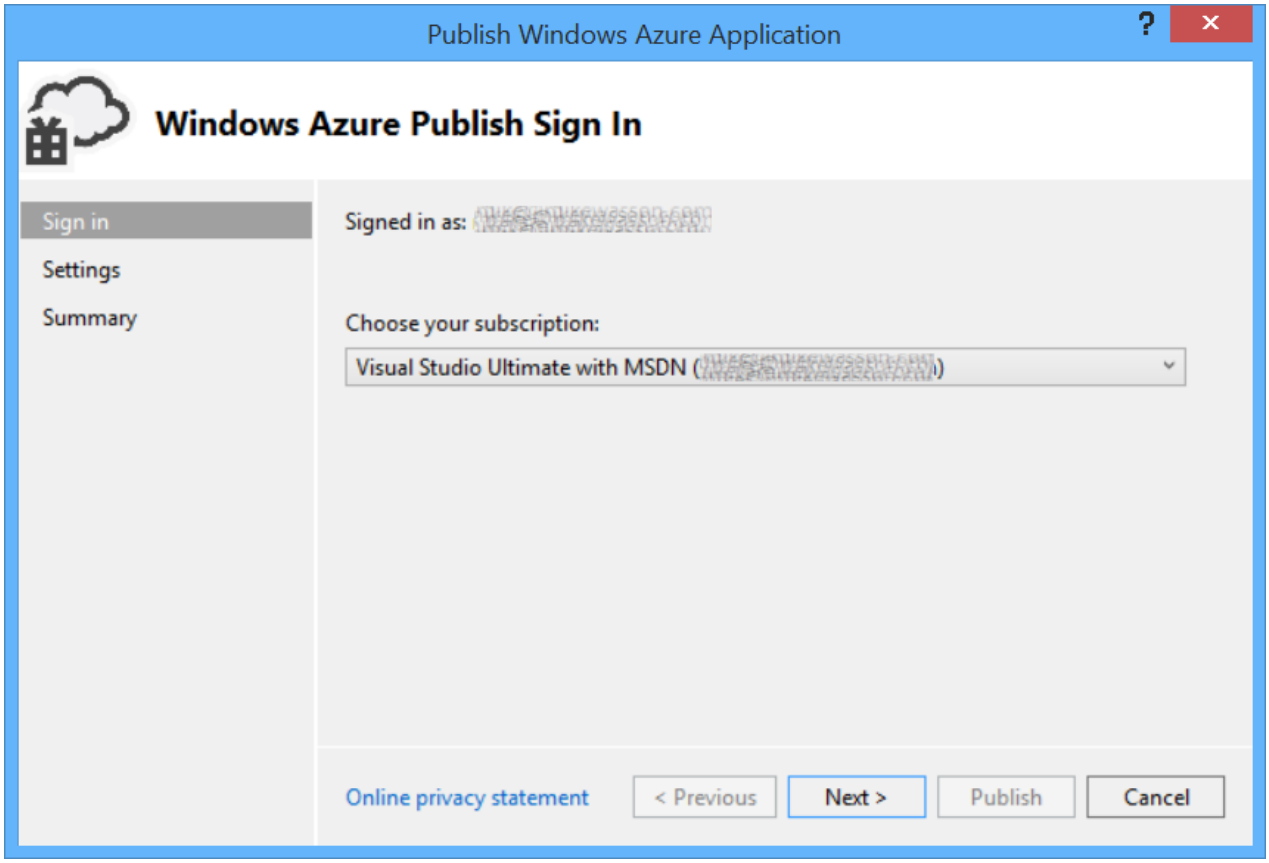
In Solution Explorer, right-click the AzureApp project. Select **Publish**.



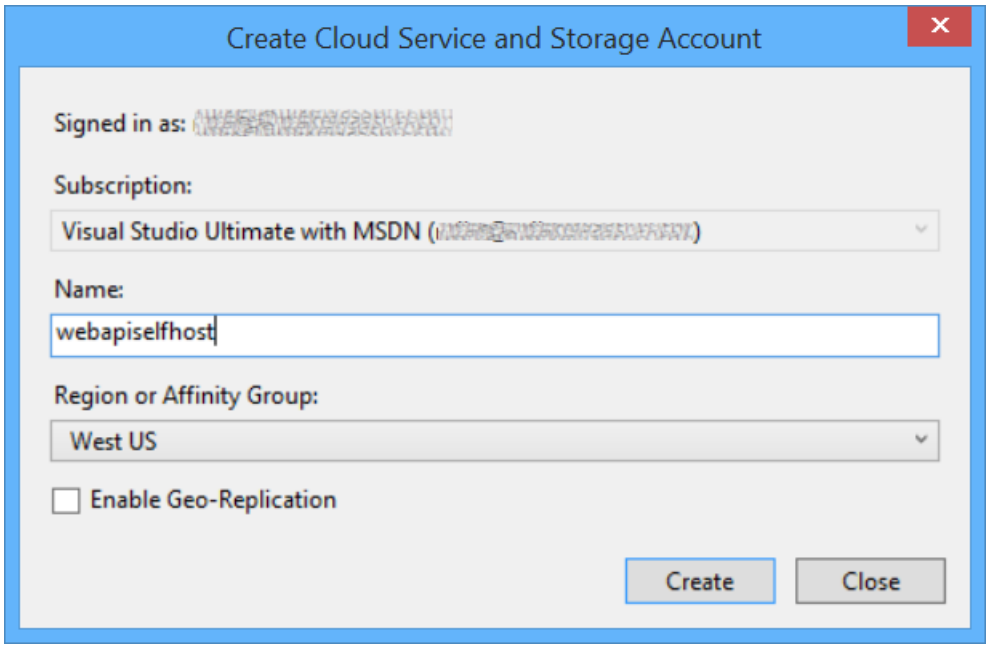
If you are not signed in to your Azure account, click **Sign In**.



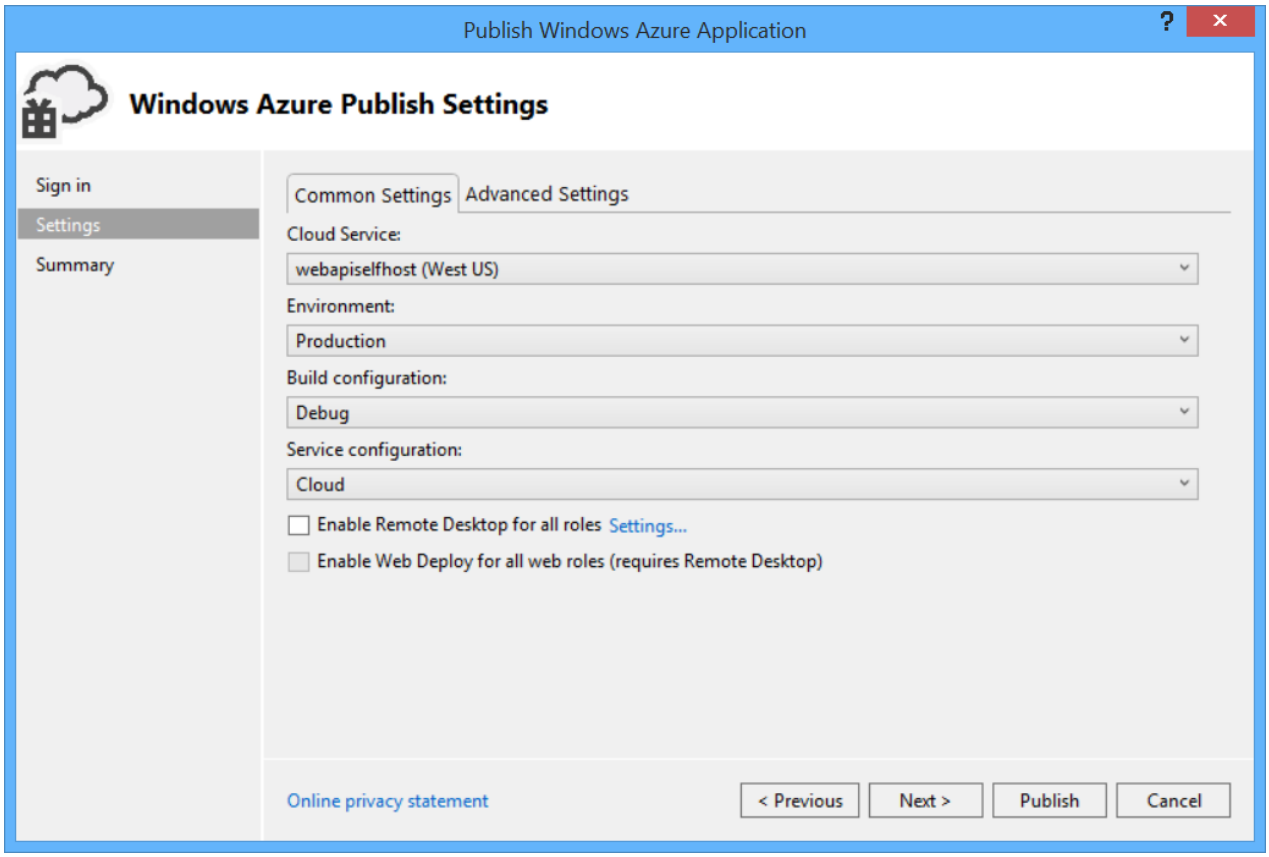
After you are signed in, choose a subscription and click **Next**.



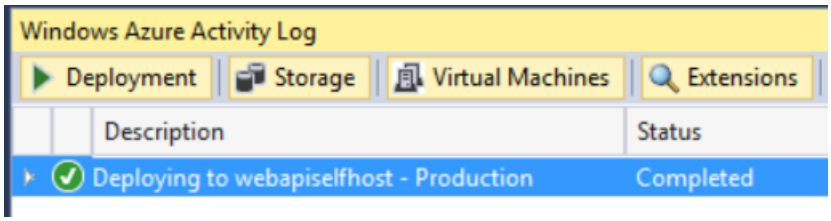
Enter a name for the cloud service and choose a region. Click **Create**.



Click **Publish**.



The Azure Activity Log window shows the progress of the deployment. When the app is deployed, browse to http://appname.cloudapp.net/test/1.



Additional Resources

- An Overview of Project Katana

- Katana Project on GitHub

Advanced Topics for ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

More advanced topics for ASP.NET Web API development, after you've learned the basics.

- Configuring Web API 2
- Dependency Injection in Web API 2
- Calling a Web API From a .NET Client (C#)
- HTTP Message Handlers in Web API
- HttpClient Message Handlers in Web API
- Sending HTML Form Data in Web API: Form-urlencoded Data
- Sending HTML Form Data in Web API: File Upload and Multipart MIME
- HTTP Cookies in ASP.NET Web API

Configuring ASP.NET Web API 2

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson

This topic describes how to configure ASP.NET Web API.

- Configuration Settings
- Configuring Web API with ASP.NET Hosting
- Configuring Web API with OWIN Self-Hosting
- Global Web API Services
- Per-Controller Configuration

Configuration Settings

Web API configuration settings are defined in the HttpConfiguration class.

MEMBER	DESCRIPTION
DependencyResolver	Enables dependency injection for controllers. See Using the Web API Dependency Resolver.
Filters	Action filters.
Formatters	Media-type formatters.
IncludeErrorDetailPolicy	Specifies whether the server should include error details, such as exception messages and stack traces, in HTTP response messages. See IncludeErrorDetailPolicy.
Initializer	A function that performs final initialization of the **HttpConfiguration**.
MessageHandlers	HTTP message handlers.
ParameterBindingRules	A collection of rules for binding parameters on controller actions.
Properties	A generic property bag.
Routes	The collection of routes. See Routing in ASP.NET Web API.
Services	The collection of services. See Services.

Prerequisites

Visual Studio 2017 Community, Professional, or Enterprise edition.

Configuring Web API with ASP.NET Hosting

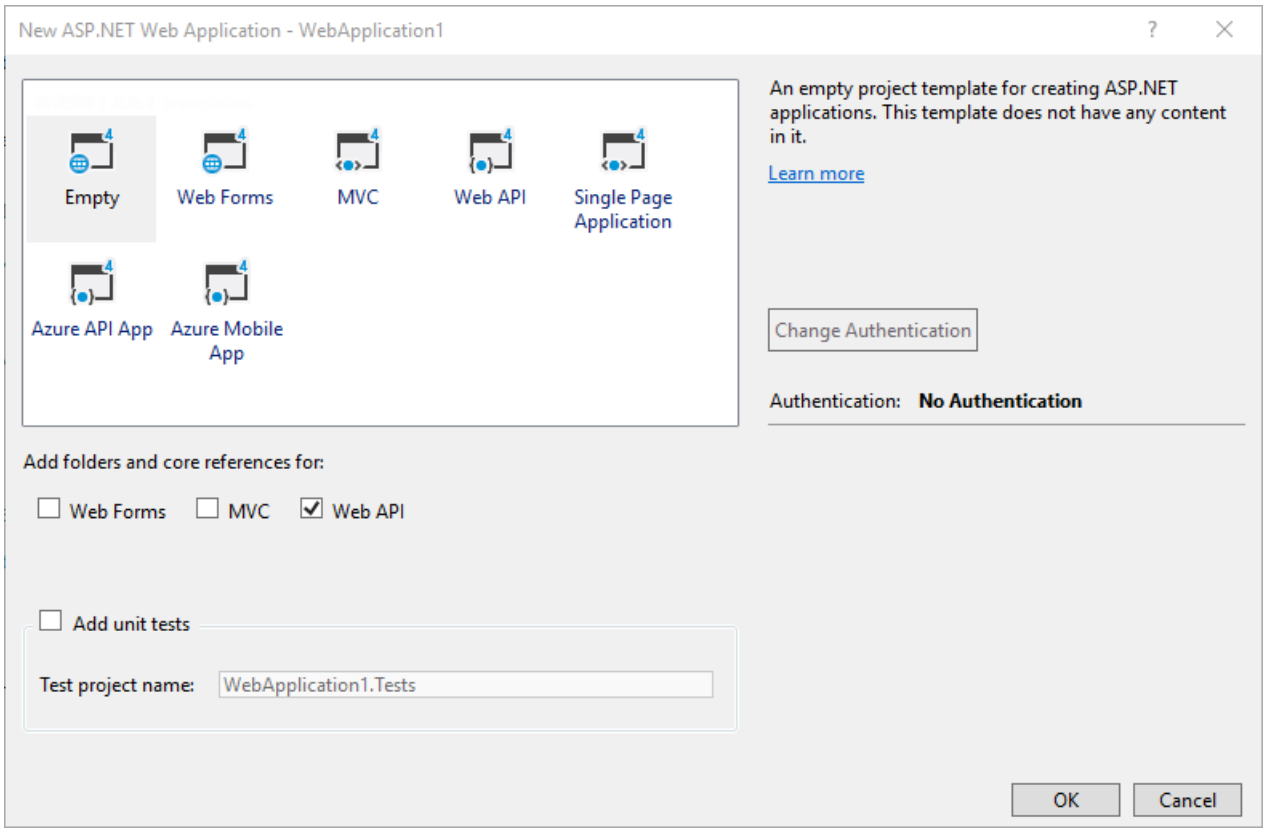
In an ASP.NET application, configure Web API by calling GlobalConfiguration.Configure in the **Application_Start** method. The **Configure** method takes a delegate with a single parameter of type **HttpConfiguration**. Perform all of your configuration inside the delegate.

Here is an example using an anonymous delegate:

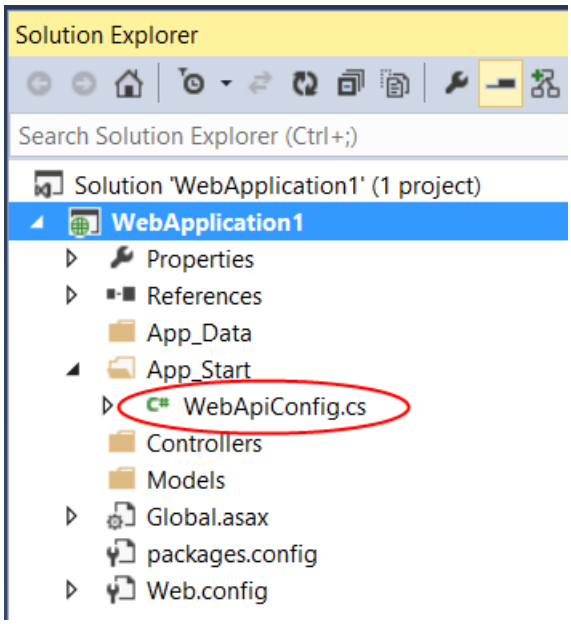
```
using System.Web.Http;
namespace WebApplication1
{
    public class WebApiApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(config =>
            {
                config.MapHttpAttributeRoutes();

                config.Routes.MapHttpRoute(
                    name: "DefaultApi",
                    routeTemplate: "api/{controller}/{id}",
                    defaults: new { id = RouteParameter.Optional }
                );
            });
        }
    }
}
```

In Visual Studio 2017, the "ASP.NET Web Application" project template automatically sets up the configuration code, if you select "Web API" in the **New ASP.NET Project** dialog.



The project template creates a file named WebApiConfig.cs inside the App_Start folder. This code file defines the delegate where you should put your Web API configuration code.




```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;

namespace WebApplication1
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // TODO: Add any additional configuration code.

            // Web API routes
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

The project template also adds the code that calls the delegate from **Application_Start**.

```
public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        GlobalConfiguration.Configure(WebApiConfig.Register);
    }
}
```

Configuring Web API with OWIN Self-Hosting

If you are self-hosting with OWIN, create a new **HttpConfiguration** instance. Perform any configuration on this instance, and then pass the instance to the **Owin.UseWebApi** extension method.

```
public class Startup
{
    public void Configuration(IAppBuilder appBuilder)
    {
        HttpConfiguration config = new HttpConfiguration();

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );

        appBuilder.UseWebApi(config);
    }
}
```

The tutorial Use OWIN to Self-Host ASP.NET Web API 2 shows the complete steps.

Global Web API Services

The **HttpConfiguration.Services** collection contains a set of global services that Web API uses to perform various tasks, such as controller selection and content negotiation.

> **NOTE**
> The **Services** collection is not a general-purpose mechanism for service discovery or dependency injection. It only stores service types that are known to the Web API framework.

The **Services** collection is initialized with a default set of services, and you can provide your own custom implementations. Some services support multiple instances, while others can have only one instance. (However, you can also provide services at the controller level; see Per-Controller Configuration.

Single-Instance Services

SERVICE	DESCRIPTION
IActionValueBinder	Gets a binding for a parameter.
IApiExplorer	Gets descriptions of the APIs exposed by the application. See Creating a Help Page for a Web API.
IAssembliesResolver	Gets a list of the assemblies for the application. See Routing and Action Selection.
IBodyModelValidator	Validates a model that is read from the request body by a media-type formatter.
IContentNegotiator	Performs content negotiation.
IDocumentationProvider	Provides documentation for APIs. The default is **null**. See Creating a Help Page for a Web API.
IHostBufferPolicySelector	Indicates whether the host should buffer HTTP message entity bodies.
IHttpActionInvoker	Invokes a controller action. See Routing and Action Selection.

SERVICE	DESCRIPTION
IHttpActionSelector	Selects a controller action. See Routing and Action Selection.
IHttpControllerActivator	Activates a controller. See Routing and Action Selection.
IHttpControllerSelector	Selects a controller. See Routing and Action Selection.
IHttpControllerTypeResolver	Provides a list of the Web API controller types in the application. See Routing and Action Selection.
ITraceManager	Initializes the tracing framework. See Tracing in ASP.NET Web API.
ITraceWriter	Provides a trace writer. The default is a "no-op" trace writer. See Tracing in ASP.NET Web API.
IModelValidatorCache	Provides a cache of model validators.

Multiple-Instance Services

SERVICE	DESCRIPTION
IFilterProvider	Returns a list of filters for a controller action.
ModelBinderProvider	Returns a model binder for a given type.
ModelMetadataProvider	Provides metadata for a model.
ModelValidatorProvider	Provides a validator for a model.
ValueProviderFactory	Creates a value provider. For more information, see Mike Stall's blog post How to create a custom value provider in WebAPI

To add a custom implementation to a multi-instance service, call **Add** or **Insert** on the **Services** collection:

```
config.Services.Add(typeof(IFilterProvider), new MyFilterProvider());
```

To replace a single-instance service with a custom implementation, call **Replace** on the **Services** collection:

```
config.Services.Replace(typeof(ITraceWriter), new MyTraceWriter());
```

Per-Controller Configuration

You can override the following settings on a per-controller basis:

- Media-type formatters
- Parameter binding rules
- Services

To do so, define a custom attribute that implements the **IControllerConfiguration** interface. Then apply the attribute to the controller.

The following example replaces the default media-type formatters with a custom formatter.

```
using System;
using System.Web.Http;
using System.Web.Http.Controllers;

namespace WebApplication1.Controllers
{

    public class UseMyFormatterAttribute : Attribute, IControllerConfiguration
    {
        public void Initialize(HttpControllerSettings settings,
            HttpControllerDescriptor descriptor)
        {
            // Clear the formatters list.
            settings.Formatters.Clear();

            // Add a custom media-type formatter.
            settings.Formatters.Add(new MyFormatter());
        }
    }

    [UseMyFormatter]
    public class ValuesController : ApiController
    {
        // Controller methods not shown...
    }
}
```

The **IControllerConfiguration.Initialize** method takes two parameters:

- An **HttpControllerSettings** object
- An **HttpControllerDescriptor** object

The **HttpControllerDescriptor** contains a description of the controller, which you can examine for informational purposes (say, to distinguish between two controllers).

Use the **HttpControllerSettings** object to configure the controller. This object contains the subset of configuration parameters that you can override on a per-controller basis. Any settings that you don't change default to the global **HttpConfiguration** object.

Dependency Injection in ASP.NET Web API 2

2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

This tutorial shows how to inject dependencies into your ASP.NET Web API controller.

Software versions used in the tutorial

- Web API 2
- Unity Application Block
- Entity Framework 6 (version 5 also works)

What is Dependency Injection?

A *dependency* is any object that another object requires. For example, it's common to define a repository that handles data access. Let's illustrate with an example. First, we'll define a domain model:

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

Here is a simple repository class that stores items in a database, using Entity Framework.

```
public class ProductsContext : DbContext
{
    public ProductsContext()
        : base("name=ProductsContext")
    {
    }
    public DbSet<Product> Products { get; set; }
}

public class ProductRepository : IDisposable
{
    private ProductsContext db = new ProductsContext();

    public IEnumerable<Product> GetAll()
    {
        return db.Products;
    }
    public Product GetByID(int id)
    {
        return db.Products.FirstOrDefault(p => p.Id == id);
    }
    public void Add(Product product)
    {
        db.Products.Add(product);
        db.SaveChanges();
    }

    protected void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (db != null)
            {
                db.Dispose();
                db = null;
            }
        }
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

Now let's define a Web API controller that supports GET requests for `Product` entities. (I'm leaving out POST and other methods for simplicity.) Here is a first attempt:

```
public class ProductsController : ApiController
{
    // This line of code is a problem!
    ProductRepository _repository = new ProductRepository();

    public IEnumerable<Product> Get()
    {
        return _repository.GetAll();
    }

    public IHttpActionResult Get(int id)
    {
        var product = _repository.GetByID(id);
        if (product == null)
        {
            return NotFound();
        }
        return Ok(product);
    }
}
```

Notice that the controller class depends on `ProductRepository`, and we are letting the controller create the `ProductRepository` instance. However, it's a bad idea to hard code the dependency in this way, for several reasons.

- If you want to replace `ProductRepository` with a different implementation, you also need to modify the controller class.
- If the `ProductRepository` has dependencies, you must configure these inside the controller. For a large project with multiple controllers, your configuration code becomes scattered across your project.
- It is hard to unit test, because the controller is hard-coded to query the database. For a unit test, you should use a mock or stub repository, which is not possible with the current design.

We can address these problems by *injecting* the repository into the controller. First, refactor the `ProductRepository` class into an interface:

```
public interface IProductRepository
{
    IEnumerable<Product> GetAll();
    Product GetById(int id);
    void Add(Product product);
}

public class ProductRepository : IProductRepository
{
    // Implementation not shown.
}
```

Then provide the `IProductRepository` as a constructor parameter:

```
public class ProductsController : ApiController
{
    private IProductRepository _repository;

    public ProductsController(IProductRepository repository)
    {
        _repository = repository;
    }

    // Other controller methods not shown.
}
```

This example uses constructor injection. You can also use *setter injection*, where you set the dependency through a setter method or property.

But now there is a problem, because your application doesn't create the controller directly. Web API creates the controller when it routes the request, and Web API doesn't know anything about `IProductRepository`. This is where the Web API dependency resolver comes in.

The Web API Dependency Resolver

Web API defines the **IDependencyResolver** interface for resolving dependencies. Here is the definition of the interface:

```
public interface IDependencyResolver : IDependencyScope, IDisposable
{
    IDependencyScope BeginScope();
}

public interface IDependencyScope : IDisposable
{
    object GetService(Type serviceType);
    IEnumerable<object> GetServices(Type serviceType);
}
```

The **IDependencyScope** interface has two methods:

- **GetService** creates one instance of a type.
- **GetServices** creates a collection of objects of a specified type.

The **IDependencyResolver** method inherits **IDependencyScope** and adds the **BeginScope** method. I'll talk about scopes later in this tutorial.

When Web API creates a controller instance, it first calls **IDependencyResolver.GetService**, passing in the controller type. You can use this extensibility hook to create the controller, resolving any dependencies. If **GetService** returns null, Web API looks for a parameterless constructor on the controller class.

Dependency Resolution with the Unity Container

Although you could write a complete **IDependencyResolver** implementation from scratch, the interface is really designed to act as bridge between Web API and existing IoC containers.

An IoC container is a software component that is responsible for managing dependencies. You register types with the container, and then use the container to create objects. The container automatically figures out the dependency relations. Many IoC containers also allow you to control things like object lifetime and scope.

> **NOTE**
> "IoC" stands for "inversion of control", which is a general pattern where a framework calls into application code. An IoC container constructs your objects for you, which "inverts" the usual flow of control.

For this tutorial, we'll use Unity from Microsoft Patterns & Practices. (Other popular libraries include Castle Windsor, Spring.Net, Autofac, Ninject, and StructureMap.) You can use NuGet Package Manager to install Unity. From the **Tools** menu in Visual Studio, select **NuGet Package Manager**, then select **Package Manager Console**. In the Package Manager Console window, type the following command:

```
Install-Package Unity
```

Here is an implementation of **IDependencyResolver** that wraps a Unity container.

```
using Microsoft.Practices.Unity;
using System;
using System.Collections.Generic;
using System.Web.Http.Dependencies;

public class UnityResolver : IDependencyResolver
{
    protected IUnityContainer container;

    public UnityResolver(IUnityContainer container)
    {
        if (container == null)
        {
            throw new ArgumentNullException("container");
        }
        this.container = container;
    }

    public object GetService(Type serviceType)
    {
        try
        {
            return container.Resolve(serviceType);
        }
        catch (ResolutionFailedException)
        {
            return null;
        }
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        try
        {
            return container.ResolveAll(serviceType);
        }
        catch (ResolutionFailedException)
        {
            return new List<object>();
        }
    }

    public IDependencyScope BeginScope()
    {
        var child = container.CreateChildContainer();
        return new UnityResolver(child);
    }

    public void Dispose()
    {
        Dispose(true);
    }

    protected virtual void Dispose(bool disposing)
    {
        container.Dispose();
    }
}
```

> **NOTE**
> If the **GetService** method cannot resolve a type, it should return **null**. If the **GetServices** method cannot resolve a type, it should return an empty collection object. Don't throw exceptions for unknown types.

Configuring the Dependency Resolver

Set the dependency resolver on the **DependencyResolver** property of the global **HttpConfiguration** object.

The following code registers the `IProductRepository` interface with Unity and then creates a `UnityResolver`.

```
public static void Register(HttpConfiguration config)
{
    var container = new UnityContainer();
    container.RegisterType<IProductRepository, ProductRepository>(new HierarchicalLifetimeManager());
    config.DependencyResolver = new UnityResolver(container);

    // Other Web API configuration not shown.
}
```

Dependency Scope and Controller Lifetime

Controllers are created per request. To manage object lifetimes, **IDependencyResolver** uses the concept of a *scope*.

The dependency resolver attached to the **HttpConfiguration** object has global scope. When Web API creates a controller, it calls **BeginScope**. This method returns an **IDependencyScope** that represents a child scope.

Web API then calls **GetService** on the child scope to create the controller. When request is complete, Web API calls **Dispose** on the child scope. Use the **Dispose** method to dispose of the controller's dependencies.

How you implement **BeginScope** depends on the IoC container. For Unity, scope corresponds to a child container:

```
public IDependencyScope BeginScope()
{
    var child = container.CreateChildContainer();
    return new UnityResolver(child);
}
```

Most IoC containers have similar equivalents.

Call a Web API From a .NET Client (C#)

5/1/2020 • 7 minutes to read • Edit Online

by Mike Wasson and Rick Anderson

Download Completed Project. Download instructions.

This tutorial shows how to call a web API from a .NET application, using System.Net.Http.HttpClient.

In this tutorial, a client app is written that consumes the following web API:

ACTION	HTTP METHOD	RELATIVE URI
Get a product by ID	GET	/api/products/*id*
Create a new product	POST	/api/products
Update a product	PUT	/api/products/*id*
Delete a product	DELETE	/api/products/*id*

To learn how to implement this API with ASP.NET Web API, see Creating a Web API that Supports CRUD Operations.

For simplicity, the client application in this tutorial is a Windows console application. **HttpClient** is also supported for Windows Phone and Windows Store apps. For more information, see Writing Web API Client Code for Multiple Platforms Using Portable Libraries

NOTE: If you pass base URLs and relative URIs as hard-coded values, be mindful of the rules for utilizing the `HttpClient` API. The `HttpClient.BaseAddress` property should be set to an address with a trailing forward slash (`/`). For example, when passing hard-coded resource URIs to the `HttpClient.GetAsync` method, don't include a leading forward slash. To get a `Product` by ID:

1. Set `client.BaseAddress = new Uri("https://localhost:5001/");`
2. Request a `Product`. For example, `client.GetAsync<Product>("api/products/4");`.

Create the Console Application

In Visual Studio, create a new Windows console app named **HttpClientSample** and paste in the following code:

```
using System;
using System.Net;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading.Tasks;

namespace HttpClientSample
{
    public class Product
    {
        public string Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
        public string Category { get; set; }
    }
```

```
class Program
{
    static HttpClient client = new HttpClient();

    static void ShowProduct(Product product)
    {
        Console.WriteLine($"Name: {product.Name}\tPrice: " +
            $"{product.Price}\tCategory: {product.Category}");
    }

    static async Task<Uri> CreateProductAsync(Product product)
    {
        HttpResponseMessage response = await client.PostAsJsonAsync(
            "api/products", product);
        response.EnsureSuccessStatusCode();

        // return URI of the created resource.
        return response.Headers.Location;
    }

    static async Task<Product> GetProductAsync(string path)
    {
        Product product = null;
        HttpResponseMessage response = await client.GetAsync(path);
        if (response.IsSuccessStatusCode)
        {
            product = await response.Content.ReadAsAsync<Product>();
        }
        return product;
    }

    static async Task<Product> UpdateProductAsync(Product product)
    {
        HttpResponseMessage response = await client.PutAsJsonAsync(
            $"api/products/{product.Id}", product);
        response.EnsureSuccessStatusCode();

        // Deserialize the updated product from the response body.
        product = await response.Content.ReadAsAsync<Product>();
        return product;
    }

    static async Task<HttpStatusCode> DeleteProductAsync(string id)
    {
        HttpResponseMessage response = await client.DeleteAsync(
            $"api/products/{id}");
        return response.StatusCode;
    }

    static void Main()
    {
        RunAsync().GetAwaiter().GetResult();
    }

    static async Task RunAsync()
    {
        // Update port # in the following line.
        client.BaseAddress = new Uri("http://localhost:64195/");
        client.DefaultRequestHeaders.Accept.Clear();
        client.DefaultRequestHeaders.Accept.Add(
            new MediaTypeWithQualityHeaderValue("application/json"));

        try
        {
            // Create a new product
            Product product = new Product
            {
                Name = "Gizmo",
```

```
                Price = 100,
                Category = "Widgets"
            };

            var url = await CreateProductAsync(product);
            Console.WriteLine($"Created at {url}");

            // Get the product
            product = await GetProductAsync(url.PathAndQuery);
            ShowProduct(product);

            // Update the product
            Console.WriteLine("Updating price...");
            product.Price = 80;
            await UpdateProductAsync(product);

            // Get the updated product
            product = await GetProductAsync(url.PathAndQuery);
            ShowProduct(product);

            // Delete the product
            var statusCode = await DeleteProductAsync(product.Id);
            Console.WriteLine($"Deleted (HTTP Status = {(int)statusCode})");

        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }

        Console.ReadLine();
    }
  }
}
```

The preceding code is the complete client app.

`RunAsync` runs and blocks until it completes. Most **HttpClient** methods are async, because they perform network I/O. All of the async tasks are done inside `RunAsync`. Normally an app doesn't block the main thread, but this app doesn't allow any interaction.

```
static async Task RunAsync()
{
    // Update port # in the following line.
    client.BaseAddress = new Uri("http://localhost:64195/");
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/json"));

    try
    {
        // Create a new product
        Product product = new Product
        {
            Name = "Gizmo",
            Price = 100,
            Category = "Widgets"
        };

        var url = await CreateProductAsync(product);
        Console.WriteLine($"Created at {url}");

        // Get the product
        product = await GetProductAsync(url.PathAndQuery);
        ShowProduct(product);

        // Update the product
        Console.WriteLine("Updating price...");
        product.Price = 80;
        await UpdateProductAsync(product);

        // Get the updated product
        product = await GetProductAsync(url.PathAndQuery);
        ShowProduct(product);

        // Delete the product
        var statusCode = await DeleteProductAsync(product.Id);
        Console.WriteLine($"Deleted (HTTP Status = {(int)statusCode})");

    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }

    Console.ReadLine();
}
```

Install the Web API Client Libraries

Use NuGet Package Manager to install the Web API Client Libraries package.

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**. In the Package Manager Console (PMC), type the following command:

```
Install-Package Microsoft.AspNet.WebApi.Client
```

The preceding command adds the following NuGet packages to the project:

- Microsoft.AspNet.WebApi.Client
- Newtonsoft.Json

Netwonsoft.Json (also known as Json.NET) is a popular high-performance JSON framework for .NET.

Add a Model Class

Examine the `Product` class:

```
public class Product
{
    public string Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Category { get; set; }
}
```

This class matches the data model used by the web API. An app can use **HttpClient** to read a `Product` instance from an HTTP response. The app doesn't have to write any deserialization code.

Create and Initialize HttpClient

Examine the static **HttpClient** property:

```
static HttpClient client = new HttpClient();
```

HttpClient is intended to be instantiated once and reused throughout the life of an application. The following conditions can result in **SocketException** errors:

- Creating a new **HttpClient** instance per request.
- Server under heavy load.

Creating a new **HttpClient** instance per request can exhaust the available sockets.

The following code initializes the **HttpClient** instance:

```
static async Task RunAsync()
{
    // Update port # in the following line.
    client.BaseAddress = new Uri("http://localhost:64195/");
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/json"));
```

The preceding code:

- Sets the base URI for HTTP requests. Change the port number to the port used in the server app. The app won't work unless port for the server app is used.
- Sets the Accept header to "application/json". Setting this header tells the server to send data in JSON format.

Send a GET request to retrieve a resource

The following code sends a GET request for a product:

```
static async Task<Product> GetProductAsync(string path)
{
    Product product = null;
    HttpResponseMessage response = await client.GetAsync(path);
    if (response.IsSuccessStatusCode)
    {
        product = await response.Content.ReadAsAsync<Product>();
    }
    return product;
}
```

The **GetAsync** method sends the HTTP GET request. When the method completes, it returns an **HttpResponseMessage** that contains the HTTP response. If the status code in the response is a success code, the response body contains the JSON representation of a product. Call **ReadAsAsync** to deserialize the JSON payload to a `Product` instance. The **ReadAsAsync** method is asynchronous because the response body can be arbitrarily large.

HttpClient does not throw an exception when the HTTP response contains an error code. Instead, the **IsSuccessStatusCode** property is **false** if the status is an error code. If you prefer to treat HTTP error codes as exceptions, call HttpResponseMessage.EnsureSuccessStatusCode on the response object. `EnsureSuccessStatusCode` throws an exception if the status code falls outside the range 200–299. Note that **HttpClient** can throw exceptions for other reasons — for example, if the request times out.

Media-Type Formatters to Deserialize

When **ReadAsAsync** is called with no parameters, it uses a default set of *media formatters* to read the response body. The default formatters support JSON, XML, and Form-url-encoded data.

Instead of using the default formatters, you can provide a list of formatters to the **ReadAsAsync** method. Using a list of formatters is useful if you have a custom media-type formatter:

```
var formatters = new List<MediaTypeFormatter>() {
    new MyCustomFormatter(),
    new JsonMediaTypeFormatter(),
    new XmlMediaTypeFormatter()
};
resp.Content.ReadAsAsync<IEnumerable<Product>>(formatters);
```

For more information, see Media Formatters in ASP.NET Web API 2

Sending a POST Request to Create a Resource

The following code sends a POST request that contains a `Product` instance in JSON format:

```
static async Task<Uri> CreateProductAsync(Product product)
{
    HttpResponseMessage response = await client.PostAsJsonAsync(
        "api/products", product);
    response.EnsureSuccessStatusCode();

    // return URI of the created resource.
    return response.Headers.Location;
}
```

The **PostAsJsonAsync** method:

- Serializes an object to JSON.
- Sends the JSON payload in a POST request.

If the request succeeds:

- It should return a 201 (Created) response.
- The response should include the URL of the created resources in the Location header.

Sending a PUT Request to Update a Resource

The following code sends a PUT request to update a product:

```
static async Task<Product> UpdateProductAsync(Product product)
{
    HttpResponseMessage response = await client.PutAsJsonAsync(
        $"api/products/{product.Id}", product);
    response.EnsureSuccessStatusCode();

    // Deserialize the updated product from the response body.
    product = await response.Content.ReadAsAsync<Product>();
    return product;
}
```

The **PutAsJsonAsync** method works like **PostAsJsonAsync**, except that it sends a PUT request instead of POST.

Sending a DELETE Request to Delete a Resource

The following code sends a DELETE request to delete a product:

```
static async Task<HttpStatusCode> DeleteProductAsync(string id)
{
    HttpResponseMessage response = await client.DeleteAsync(
        $"api/products/{id}");
    return response.StatusCode;
}
```

Like GET, a DELETE request does not have a request body. You don't need to specify JSON or XML format with DELETE.

Test the sample

To test the client app:

1. Download and run the server app. Download instructions. Verify the server app is working. For example, `http://localhost:64195/api/products` should return a list of products.

2. Set the base URI for HTTP requests. Change the port number to the port used in the server app.

   ```
   static async Task RunAsync()
   {
       // Update port # in the following line.
       client.BaseAddress = new Uri("http://localhost:64195/");
       client.DefaultRequestHeaders.Accept.Clear();
       client.DefaultRequestHeaders.Accept.Add(
           new MediaTypeWithQualityHeaderValue("application/json"));
   ```

3. Run the client app. The following output is produced:

```
Created at http://localhost:64195/api/products/4
Name: Gizmo     Price: 100.0    Category: Widgets
Updating price...
Name: Gizmo     Price: 80.0     Category: Widgets
Deleted (HTTP Status = 204)
```

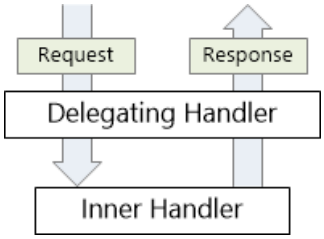
HTTP Message Handlers in ASP.NET Web API

2/20/2020 • 7 minutes to read • Edit Online

by Mike Wasson

A *message handler* is a class that receives an HTTP request and returns an HTTP response. Message handlers derive from the abstract **HttpMessageHandler** class.

Typically, a series of message handlers are chained together. The first handler receives an HTTP request, does some processing, and gives the request to the next handler. At some point, the response is created and goes back up the chain. This pattern is called a *delegating* handler.



Server-Side Message Handlers

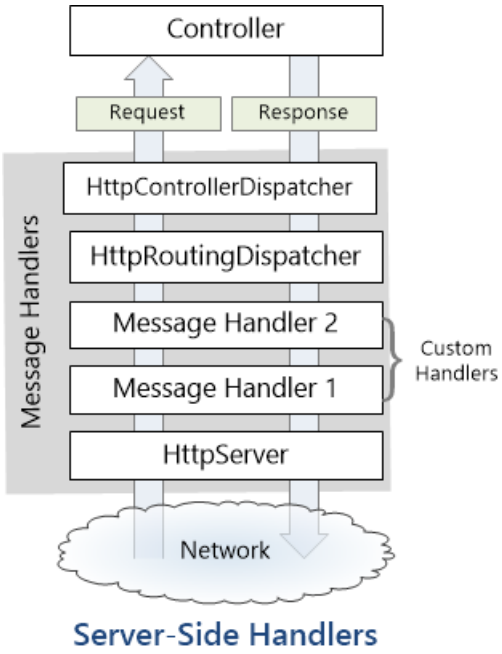
On the server side, the Web API pipeline uses some built-in message handlers:

- **HttpServer** gets the request from the host.
- **HttpRoutingDispatcher** dispatches the request based on the route.
- **HttpControllerDispatcher** sends the request to a Web API controller.

You can add custom handlers to the pipeline. Message handlers are good for cross-cutting concerns that operate at the level of HTTP messages (rather than controller actions). For example, a message handler might:

- Read or modify request headers.
- Add a response header to responses.
- Validate requests before they reach the controller.

This diagram shows two custom handlers inserted into the pipeline:



> **NOTE**
>
> On the client side, HttpClient also uses message handlers. For more information, see HttpClient Message Handlers.

Custom Message Handlers

To write a custom message handler, derive from **System.Net.Http.DelegatingHandler** and override the **SendAsync** method. This method has the following signature:

```
Task<HttpResponseMessage> SendAsync(
    HttpRequestMessage request, CancellationToken cancellationToken);
```

The method takes an **HttpRequestMessage** as input and asynchronously returns an **HttpResponseMessage**. A typical implementation does the following:

1. Process the request message.
2. Call `base.SendAsync` to send the request to the inner handler.
3. The inner handler returns a response message. (This step is asynchronous.)
4. Process the response and return it to the caller.

Here is a trivial example:

```
public class MessageHandler1 : DelegatingHandler
{
    protected async override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        Debug.WriteLine("Process request");
        // Call the inner handler.
        var response = await base.SendAsync(request, cancellationToken);
        Debug.WriteLine("Process response");
        return response;
    }
}
```

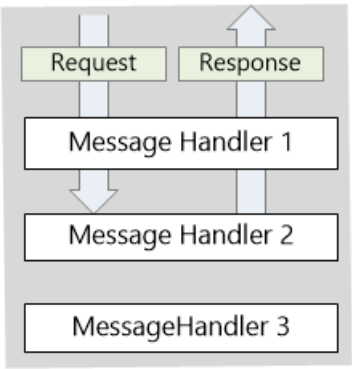
> **NOTE**
> The call to `base.SendAsync` is asynchronous. If the handler does any work after this call, use the **await** keyword, as shown.

A delegating handler can also skip the inner handler and directly create the response:

```
public class MessageHandler2 : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        // Create the response.
        var response = new HttpResponseMessage(HttpStatusCode.OK)
        {
            Content = new StringContent("Hello!")
        };

        // Note: TaskCompletionSource creates a task that does not contain a delegate.
        var tsc = new TaskCompletionSource<HttpResponseMessage>();
        tsc.SetResult(response);   // Also sets the task state to "RanToCompletion"
        return tsc.Task;
    }
}
```

If a delegating handler creates the response without calling `base.SendAsync`, the request skips the rest of the pipeline. This can be useful for a handler that validates the request (creating an error response).



Adding a Handler to the Pipeline

To add a message handler on the server side, add the handler to the **HttpConfiguration.MessageHandlers** collection. If you used the "ASP.NET MVC 4 Web Application" template to create the project, you can do this inside the **WebApiConfig** class:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.MessageHandlers.Add(new MessageHandler1());
        config.MessageHandlers.Add(new MessageHandler2());

        // Other code not shown...
    }
}
```

Message handlers are called in the same order that they appear in **MessageHandlers** collection. Because they are nested, the response message travels in the other direction. That is, the last handler is the first to get the

response message.

Notice that you don't need to set the inner handlers; the Web API framework automatically connects the message handlers.

If you are self-hosting, create an instance of the **HttpSelfHostConfiguration** class and add the handlers to the **MessageHandlers** collection.

```
var config = new HttpSelfHostConfiguration("http://localhost");
config.MessageHandlers.Add(new MessageHandler1());
config.MessageHandlers.Add(new MessageHandler2());
```

Now let's look at some examples of custom message handlers.

Example: X-HTTP-Method-Override

X-HTTP-Method-Override is a non-standard HTTP header. It is designed for clients that cannot send certain HTTP request types, such as PUT or DELETE. Instead, the client sends a POST request and sets the X-HTTP-Method-Override header to the desired method. For example:

```
X-HTTP-Method-Override: PUT
```

Here is a message handler that adds support for X-HTTP-Method-Override:

```
public class MethodOverrideHandler : DelegatingHandler
{
    readonly string[] _methods = { "DELETE", "HEAD", "PUT" };
    const string _header = "X-HTTP-Method-Override";

    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        // Check for HTTP POST with the X-HTTP-Method-Override header.
        if (request.Method == HttpMethod.Post && request.Headers.Contains(_header))
        {
            // Check if the header value is in our methods list.
            var method = request.Headers.GetValues(_header).FirstOrDefault();
            if (_methods.Contains(method, StringComparer.InvariantCultureIgnoreCase))
            {
                // Change the request method.
                request.Method = new HttpMethod(method);
            }
        }
        return base.SendAsync(request, cancellationToken);
    }
}
```

In the **SendAsync** method, the handler checks whether the request message is a POST request, and whether it contains the X-HTTP-Method-Override header. If so, it validates the header value, and then modifies the request method. Finally, the handler calls `base.SendAsync` to pass the message to the next handler.

When the request reaches the **HttpControllerDispatcher** class, **HttpControllerDispatcher** will route the request based on the updated request method.

Example: Adding a Custom Response Header

Here is a message handler that adds a custom header to every response message:

```
// .Net 4.5
public class CustomHeaderHandler : DelegatingHandler
{
    async protected override Task<HttpResponseMessage> SendAsync(
            HttpRequestMessage request, CancellationToken cancellationToken)
    {
        HttpResponseMessage response = await base.SendAsync(request, cancellationToken);
        response.Headers.Add("X-Custom-Header", "This is my custom header.");
        return response;
    }
}
```

First, the handler calls `base.SendAsync` to pass the request to the inner message handler. The inner handler returns a response message, but it does so asynchronously using a **Task<T>** object. The response message is not available until `base.SendAsync` completes asynchronously.

This example uses the **await** keyword to perform work asynchronously after `SendAsync` completes. If you are targeting .NET Framework 4.0, use the **Task**<T>.**ContinueWith** method:

```
public class CustomHeaderHandler : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        return base.SendAsync(request, cancellationToken).ContinueWith(
            (task) =>
            {
                HttpResponseMessage response = task.Result;
                response.Headers.Add("X-Custom-Header", "This is my custom header.");
                return response;
            }
        );
    }
}
```

Example: Checking for an API Key

Some web services require clients to include an API key in their request. The following example shows how a message handler can check requests for a valid API key:

```
public class ApiKeyHandler : DelegatingHandler
{
    public string Key { get; set; }

    public ApiKeyHandler(string key)
    {
        this.Key = key;
    }

    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        if (!ValidateKey(request))
        {
            var response = new HttpResponseMessage(HttpStatusCode.Forbidden);
            var tsc = new TaskCompletionSource<HttpResponseMessage>();
            tsc.SetResult(response);
            return tsc.Task;
        }
        return base.SendAsync(request, cancellationToken);
    }

    private bool ValidateKey(HttpRequestMessage message)
    {
        var query = message.RequestUri.ParseQueryString();
        string key = query["key"];
        return (key == Key);
    }
}
```

This handler looks for the API key in the URI query string. (For this example, we assume that the key is a static string. A real implementation would probably use more complex validation.) If the query string contains the key, the handler passes the request to the inner handler.

If the request does not have a valid key, the handler creates a response message with status 403, Forbidden. In this case, the handler does not call `base.SendAsync`, so the inner handler never receives the request, nor does the controller. Therefore, the controller can assume that all incoming requests have a valid API key.

> **NOTE**
> If the API key applies only to certain controller actions, consider using an action filter instead of a message handler. Action filters run after URI routing is performed.

Per-Route Message Handlers

Handlers in the **HttpConfiguration.MessageHandlers** collection apply globally.

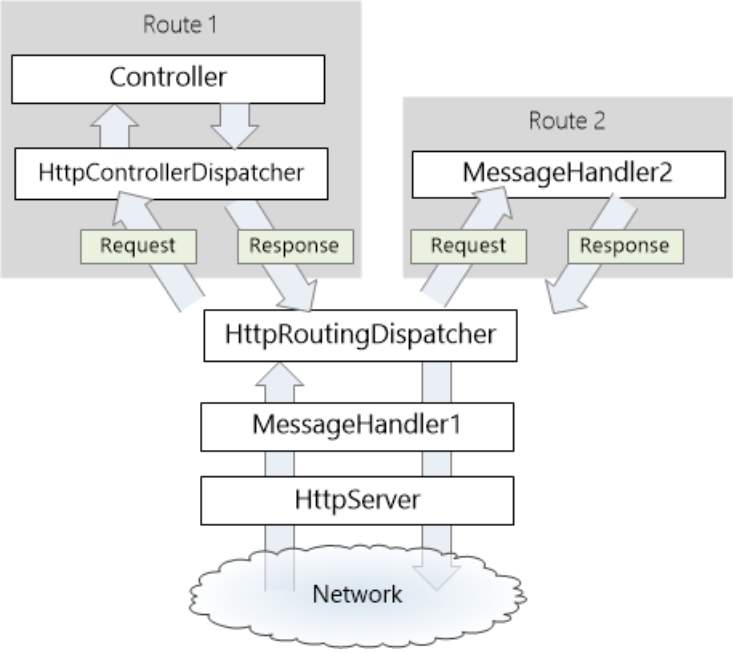
Alternatively, you can add a message handler to a specific route when you define the route:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.MapHttpRoute(
            name: "Route1",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );

        config.Routes.MapHttpRoute(
            name: "Route2",
            routeTemplate: "api2/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional },
            constraints: null,
            handler: new MessageHandler2()  // per-route message handler
        );

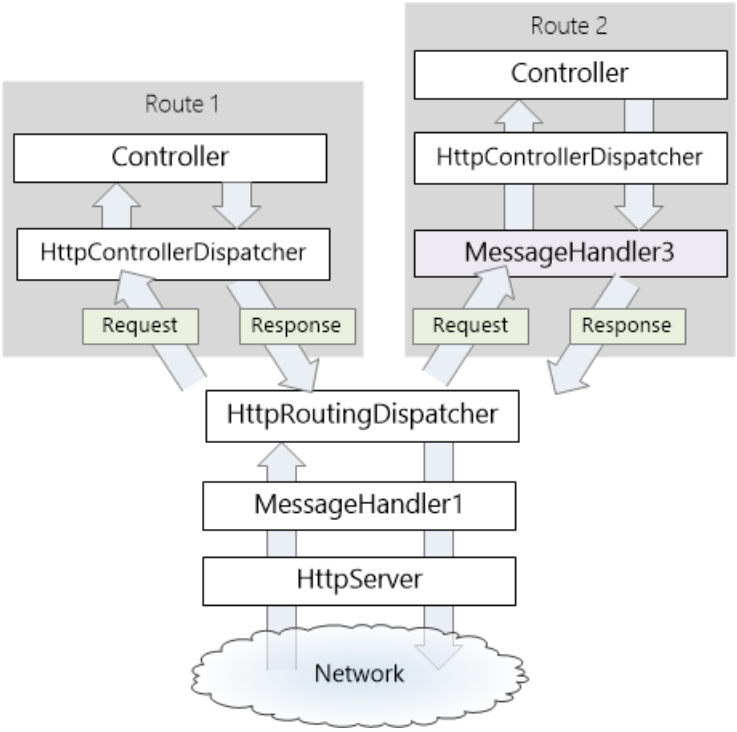
        config.MessageHandlers.Add(new MessageHandler1());  // global message handler
    }
}
```

In this example, if the request URI matches "Route2", the request is dispatched to `MessageHandler2`. The following diagram shows the pipeline for these two routes:



Notice that `MessageHandler2` replaces the default **HttpControllerDispatcher**. In this example, `MessageHandler2` creates the response, and requests that match "Route2" never go to a controller. This lets you replace the entire Web API controller mechanism with your own custom endpoint.

Alternatively, a per-route message handler can delegate to **HttpControllerDispatcher**, which then dispatches to a controller.



The following code shows how to configure this route:

```
// List of delegating handlers.
DelegatingHandler[] handlers = new DelegatingHandler[] {
    new MessageHandler3()
};

// Create a message handler chain with an end-point.
var routeHandlers = HttpClientFactory.CreatePipeline(
    new HttpControllerDispatcher(config), handlers);

config.Routes.MapHttpRoute(
    name: "Route2",
    routeTemplate: "api2/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional },
    constraints: null,
    handler: routeHandlers
);
```

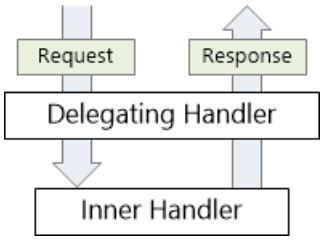
HttpClient Message Handlers in ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

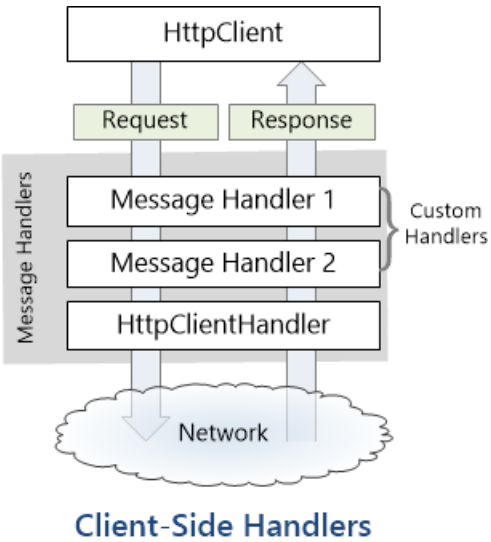
by Mike Wasson

A *message handler* is a class that receives an HTTP request and returns an HTTP response.

Typically, a series of message handlers are chained together. The first handler receives an HTTP request, does some processing, and gives the request to the next handler. At some point, the response is created and goes back up the chain. This pattern is called a *delegating* handler.



On the client side, the **HttpClient** class uses a message handler to process requests. The default handler is **HttpClientHandler**, which sends the request over the network and gets the response from the server. You can insert custom message handlers into the client pipeline:



> **NOTE**
>
> ASP.NET Web API also uses message handlers on the server side. For more information, see HTTP Message Handlers.

Custom Message Handlers

To write a custom message handler, derive from **System.Net.Http.DelegatingHandler** and override the **SendAsync** method. Here is the method signature:

```
Task<HttpResponseMessage> SendAsync(
    HttpRequestMessage request, CancellationToken cancellationToken);
```

The method takes an **HttpRequestMessage** as input and asynchronously returns an **HttpResponseMessage**. A

typical implementation does the following:

1. Process the request message.
2. Call `base.SendAsync` to send the request to the inner handler.
3. The inner handler returns a response message. (This step is asynchronous.)
4. Process the response and return it to the caller.

The following example shows a message handler that adds a custom header to the outgoing request:

```
class MessageHandler1 : DelegatingHandler
{
    private int _count = 0;

    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, System.Threading.CancellationToken cancellationToken)
    {
        System.Threading.Interlocked.Increment(ref _count);
        request.Headers.Add("X-Custom-Header", _count.ToString());
        return base.SendAsync(request, cancellationToken);
    }
}
```

The call to `base.SendAsync` is asynchronous. If the handler does any work after this call, use the **await** keyword to resume execution after the method completes. The following example shows a handler that logs error codes. The logging itself is not very interesting, but the example shows how to get at the response inside the handler.

```
class LoggingHandler : DelegatingHandler
{
    StreamWriter _writer;

    public LoggingHandler(Stream stream)
    {
        _writer = new StreamWriter(stream);
    }

    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, System.Threading.CancellationToken cancellationToken)
    {
        var response = await base.SendAsync(request, cancellationToken);

        if (!response.IsSuccessStatusCode)
        {
            _writer.WriteLine("{0}\t{1}\t{2}", request.RequestUri,
                (int)response.StatusCode, response.Headers.Date);
        }
        return response;
    }

    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            _writer.Dispose();
        }
        base.Dispose(disposing);
    }
}
```

Adding Message Handlers to the Client Pipeline

To add custom handlers to **HttpClient**, use the **HttpClientFactory.Create** method:

```
HttpClient client = HttpClientFactory.Create(new Handler1(), new Handler2(), new Handler3());
```

Message handlers are called in the order that you pass them into the `Create` method. Because handlers are nested, the response message travels in the other direction. That is, the last handler is the first to get the response message.

Sending HTML Form Data in ASP.NET Web API: Form-urlencoded Data

2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson

Part 1: Form-urlencoded Data

This article shows how to post form-urlencoded data to a Web API controller.

- Overview of HTML Forms
- Sending Complex Types
- Sending Form Data via AJAX
- Sending Simple Types

> **NOTE**
>
> Download the completed project.

Overview of HTML Forms

HTML forms use either GET or POST to send data to the server. The **method** attribute of the **form** element gives the HTTP method:

```
<form action="api/values" method="post">
```

The default method is GET. If the form uses GET, the form data is encoded in the URI as a query string. If the form uses POST, the form data is placed in the request body. For POSTed data, the **enctype** attribute specifies the format of the request body:

ENCTYPE	DESCRIPTION
application/x-www-form-urlencoded	Form data is encoded as name/value pairs, similar to a URI query string. This is the default format for POST.
multipart/form-data	Form data is encoded as a multipart MIME message. Use this format if you are uploading a file to the server.

Part 1 of this article looks at x-www-form-urlencoded format. Part 2 describes multipart MIME.

Sending Complex Types

Typically, you will send a complex type, composed of values taken from several form controls. Consider the following model that represents a status update:

```
namespace FormEncode.Models
{
    using System;
    using System.ComponentModel.DataAnnotations;

    public class Update
    {
        [Required]
        [MaxLength(140)]
        public string Status { get; set; }

        public DateTime Date { get; set; }
    }
}
```

Here is a Web API controller that accepts an `Update` object via POST.

```
namespace FormEncode.Controllers
{
    using FormEncode.Models;
    using System;
    using System.Collections.Generic;
    using System.Net;
    using System.Net.Http;
    using System.Web;
    using System.Web.Http;

    public class UpdatesController : ApiController
    {
        static readonly Dictionary<Guid, Update> updates = new Dictionary<Guid, Update>();

        [HttpPost]
        [ActionName("Complex")]
        public HttpResponseMessage PostComplex(Update update)
        {
            if (ModelState.IsValid && update != null)
            {
                // Convert any HTML markup in the status text.
                update.Status = HttpUtility.HtmlEncode(update.Status);

                // Assign a new ID.
                var id = Guid.NewGuid();
                updates[id] = update;

                // Create a 201 response.
                var response = new HttpResponseMessage(HttpStatusCode.Created)
                {
                    Content = new StringContent(update.Status)
                };
                response.Headers.Location =
                    new Uri(Url.Link("DefaultApi", new { action = "status", id = id }));
                return response;
            }
            else
            {
                return Request.CreateResponse(HttpStatusCode.BadRequest);
            }
        }

        [HttpGet]
        public Update Status(Guid id)
        {
            Update update;
            if (updates.TryGetValue(id, out update))
            {
                return update;
            }
            else
            {
                throw new HttpResponseException(HttpStatusCode.NotFound);
            }
        }

    }
}
```

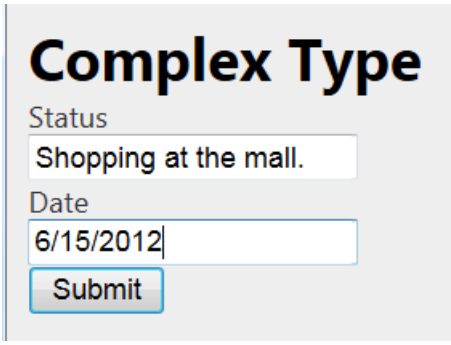
NOTE

This controller uses action-based routing, so the route template is "api/{controller}/{action}/{id}". The client will post the data to "/api/updates/complex".

Now let's write an HTML form for users to submit a status update.

```
<h1>Complex Type</h1>
<form id="form1" method="post" action="api/updates/complex"
    enctype="application/x-www-form-urlencoded">
    <div>
        <label for="status">Status</label>
    </div>
    <div>
        <input name="status" type="text" />
    </div>
    <div>
        <label for="date">Date</label>
    </div>
    <div>
        <input name="date" type="text" />
    </div>
    <div>
        <input type="submit" value="Submit" />
    </div>
</form>
```

Notice that the **action** attribute on the form is the URI of our controller action. Here is the form with some values entered in:



When the user clicks Submit, the browser sends an HTTP request similar to the following:

```
POST http://localhost:38899/api/updates/complex HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0)
Content-Type: application/x-www-form-urlencoded
Content-Length: 47

status=Shopping+at+the+mall.&date=6%2F15%2F2012
```

Notice that the request body contains the form data, formatted as name/value pairs. Web API automatically converts the name/value pairs into an instance of the `Update` class.

Sending Form Data via AJAX

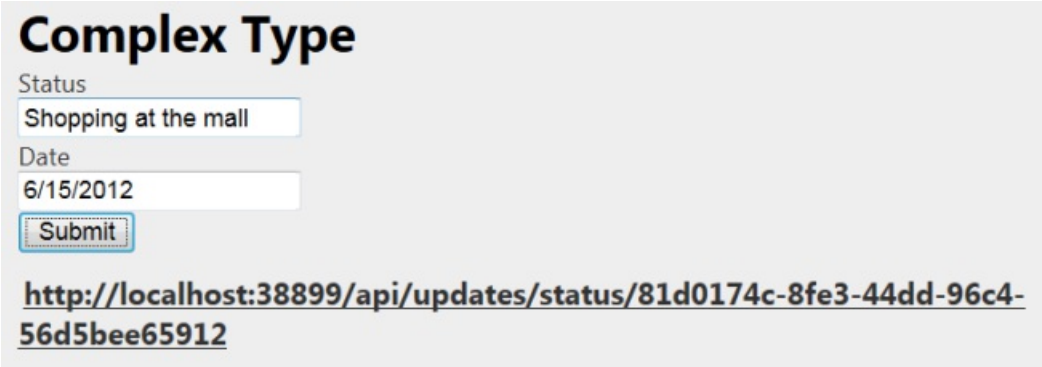
When a user submits a form, the browser navigates away from the current page and renders the body of the response message. That's OK when the response is an HTML page. With a web API, however, the response body is usually either empty or contains structured data, such as JSON. In that case, it makes more sense to send the form data using an AJAX request, so that the page can process the response.

The following code shows how to post form data using jQuery.

```
<script type="text/javascript">
    $("#form1").submit(function () {
        var jqxhr = $.post('api/updates/complex', $('#form1').serialize())
            .success(function () {
                var loc = jqxhr.getResponseHeader('Location');
                var a = $('<a/>', { href: loc, text: loc });
                $('#message').html(a);
            })
            .error(function () {
                $('#message').html("Error posting the update.");
            });
        return false;
    });
</script>
```

The jQuery **submit** function replaces the form action with a new function. This overrides the default behavior of the Submit button. The **serialize** function serializes the form data into name/value pairs. To send the form data to the server, call `$.post()`.

When the request completes, the `.success()` or `.error()` handler displays an appropriate message to the user.



Sending Simple Types

In the previous sections, we sent a complex type, which Web API deserialized to an instance of a model class. You can also send simple types, such as a string.

> **NOTE**
> Before sending a simple type, consider wrapping the value in a complex type instead. This gives you the benefits of model validation on the server side, and makes it easier to extend your model if needed.

The basic steps to send a simple type are the same, but there are two subtle differences. First, in the controller, you must decorate the parameter name with the **FromBody** attribute.

```
[HttpPost]
[ActionName("Simple")]
public HttpResponseMessage PostSimple([FromBody] string value)
{
    if (value != null)
    {
        Update update = new Update()
        {
            Status = HttpUtility.HtmlEncode(value),
            Date = DateTime.UtcNow
        };

        var id = Guid.NewGuid();
        updates[id] = update;

        var response = new HttpResponseMessage(HttpStatusCode.Created)
        {
            Content = new StringContent(update.Status)
        };
        response.Headers.Location =
            new Uri(Url.Link("DefaultApi", new { action = "status", id = id }));
        return response;
    }
    else
    {
        return Request.CreateResponse(HttpStatusCode.BadRequest);
    }
```

By default, Web API tries to get simple types from the request URI. The **FromBody** attribute tells Web API to read the value from the request body.

> **NOTE**
> Web API reads the response body at most once, so only one parameter of an action can come from the request body. If you need to get multiple values from the request body, define a complex type.

Second, the client needs to send the value with the following format:

```
=value
```

Specifically, the name portion of the name/value pair must be empty for a simple type. Not all browsers support this for HTML forms, but you create this format in script as follows:

```
$.post('api/updates/simple', { "": $('#status1').val() });
```

Here is an example form:

```
<h1>Simple Type</h1>
<form id="form2">
    <div>
        <label for="status">Status</label>
    </div>
    <div>
        <input id="status1" type="text" />
    </div>
    <div>
        <input type="submit" value="Submit" />
    </div>
</form>
```

And here is the script to submit the form value. The only difference from the previous script is the argument passed into the **post** function.

```
$('#form2').submit(function () {
    var jqxhr = $.post('api/updates/simple', { "": $('#status1').val() })
        .success(function () {
            var loc = jqxhr.getResponseHeader('Location');
            var a = $('<a/>', { href: loc, text: loc });
            $('#message').html(a);
        })
        .error(function () {
            $('#message').html("Error posting the update.");
        });
    return false;
});
```

You can use the same approach to send an array of simple types:

```
$.post('api/updates/postlist', { "": ["update one", "update two", "update three"] });
```

Additional Resources

Part 2: File Upload and Multipart MIME

Sending HTML Form Data in ASP.NET Web API: File Upload and Multipart MIME

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson

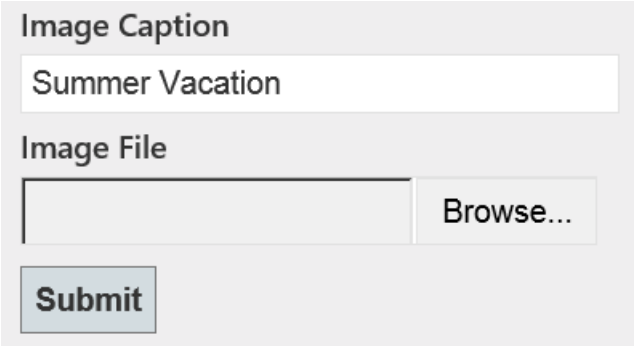
Part 2: File Upload and Multipart MIME

This tutorial shows how to upload files to a web API. It also describes how to process multipart MIME data.

> **NOTE**
>
> Download the completed project.

Here is an example of an HTML form for uploading a file:

```
<form name="form1" method="post" enctype="multipart/form-data" action="api/upload">
    <div>
        <label for="caption">Image Caption</label>
        <input name="caption" type="text" />
    </div>
    <div>
        <label for="image1">Image File</label>
        <input name="image1" type="file" />
    </div>
    <div>
        <input type="submit" value="Submit" />
    </div>
</form>
```



This form contains a text input control and a file input control. When a form contains a file input control, the **enctype** attribute should always be "multipart/form-data", which specifies that the form will be sent as a multipart MIME message.

The format of a multipart MIME message is easiest to understand by looking at an example request:

```
POST http://localhost:50460/api/values/1 HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:12.0) Gecko/20100101 Firefox/12.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data; boundary=---------------------------41184676334
Content-Length: 29278

-----------------------------41184676334
Content-Disposition: form-data; name="caption"

Summer vacation
-----------------------------41184676334
Content-Disposition: form-data; name="image1"; filename="GrandCanyon.jpg"
Content-Type: image/jpeg

(Binary data not shown)
-----------------------------41184676334--
```

This message is divided into two *parts*, one for each form control. Part boundaries are indicated by the lines that start with dashes.

> **NOTE**
> The part boundary includes a random component ("41184676334") to ensure that the boundary string does not accidentally appear inside a message part.

Each message part contains one or more headers, followed by the part contents.

- The Content-Disposition header includes the name of the control. For files, it also contains the file name.
- The Content-Type header describes the data in the part. If this header is omitted, the default is text/plain.

In the previous example, the user uploaded a file named GrandCanyon.jpg, with content type image/jpeg; and the value of the text input was "Summer Vacation".

File Upload

Now let's look at a Web API controller that reads files from a multipart MIME message. The controller will read the files asynchronously. Web API supports asynchronous actions using the task-based programming model. First, here is the code if you are targeting .NET Framework 4.5, which supports the **async** and **await** keywords.

```
using System.Diagnostics;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.Web;
using System.Web.Http;

public class UploadController : ApiController
{
    public async Task<HttpResponseMessage> PostFormData()
    {
        // Check if the request contains multipart/form-data.
        if (!Request.Content.IsMimeMultipartContent())
        {
            throw new HttpResponseException(HttpStatusCode.UnsupportedMediaType);
        }

        string root = HttpContext.Current.Server.MapPath("~/App_Data");
        var provider = new MultipartFormDataStreamProvider(root);

        try
        {
            // Read the form data.
            await Request.Content.ReadAsMultipartAsync(provider);

            // This illustrates how to get the file names.
            foreach (MultipartFileData file in provider.FileData)
            {
                Trace.WriteLine(file.Headers.ContentDisposition.FileName);
                Trace.WriteLine("Server file path: " + file.LocalFileName);
            }
            return Request.CreateResponse(HttpStatusCode.OK);
        }
        catch (System.Exception e)
        {
            return Request.CreateErrorResponse(HttpStatusCode.InternalServerError, e);
        }
    }

}
```

Notice that the controller action does not take any parameters. That's because we process the request body inside the action, without invoking a media-type formatter.

The **IsMultipartContent** method checks whether the request contains a multipart MIME message. If not, the controller returns HTTP status code 415 (Unsupported Media Type).

The **MultipartFormDataStreamProvider** class is a helper object that allocates file streams for uploaded files. To read the multipart MIME message, call the **ReadAsMultipartAsync** method. This method extracts all of the message parts and writes them into the streams provided by the **MultipartFormDataStreamProvider**.

When the method completes, you can get information about the files from the **FileData** property, which is a collection of **MultipartFileData** objects.

- **MultipartFileData.FileName** is the local file name on the server, where the file was saved.
- **MultipartFileData.Headers** contains the part header (*not* the request header). You can use this to access the Content_Disposition and Content-Type headers.

As the name suggests, **ReadAsMultipartAsync** is an asynchronous method. To perform work after the method completes, use a continuation task (.NET 4.0) or the **await** keyword (.NET 4.5).

Here is the .NET Framework 4.0 version of the previous code:

```
public Task<HttpResponseMessage> PostFormData()
{
    // Check if the request contains multipart/form-data.
    if (!Request.Content.IsMimeMultipartContent())
    {
        throw new HttpResponseException(HttpStatusCode.UnsupportedMediaType);
    }

    string root = HttpContext.Current.Server.MapPath("~/App_Data");
    var provider = new MultipartFormDataStreamProvider(root);

    // Read the form data and return an async task.
    var task = Request.Content.ReadAsMultipartAsync(provider).
        ContinueWith<HttpResponseMessage>(t =>
        {
            if (t.IsFaulted || t.IsCanceled)
            {
                Request.CreateErrorResponse(HttpStatusCode.InternalServerError, t.Exception);
            }

            // This illustrates how to get the file names.
            foreach (MultipartFileData file in provider.FileData)
            {
                Trace.WriteLine(file.Headers.ContentDisposition.FileName);
                Trace.WriteLine("Server file path: " + file.LocalFileName);
            }
            return Request.CreateResponse(HttpStatusCode.OK);
        });

    return task;
}
```

Reading Form Control Data

The HTML form that I showed earlier had a text input control.

```
<div>
        <label for="caption">Image Caption</label>
        <input name="caption" type="text" />
    </div>
```

You can get the value of the control from the **FormData** property of the **MultipartFormDataStreamProvider**.

```
public async Task<HttpResponseMessage> PostFormData()
{
    if (!Request.Content.IsMimeMultipartContent())
    {
        throw new HttpResponseException(HttpStatusCode.UnsupportedMediaType);
    }

    string root = HttpContext.Current.Server.MapPath("~/App_Data");
    var provider = new MultipartFormDataStreamProvider(root);

    try
    {
        await Request.Content.ReadAsMultipartAsync(provider);

        // Show all the key-value pairs.
        foreach (var key in provider.FormData.AllKeys)
        {
            foreach (var val in provider.FormData.GetValues(key))
            {
                Trace.WriteLine(string.Format("{0}: {1}", key, val));
            }
        }

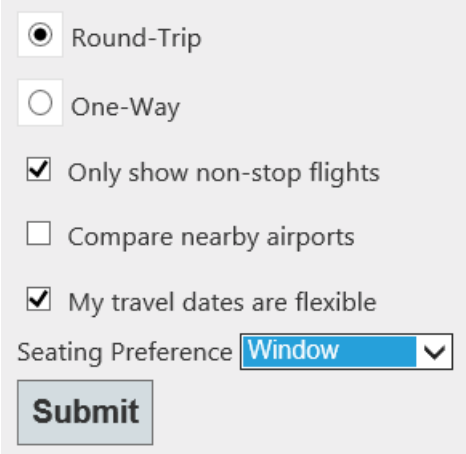
        return Request.CreateResponse(HttpStatusCode.OK);
    }
    catch (System.Exception e)
    {
        return Request.CreateErrorResponse(HttpStatusCode.InternalServerError, e);
    }
}
```

FormData is a **NameValueCollection** that contains name/value pairs for the form controls. The collection can contain duplicate keys. Consider this form:

```
<form name="trip_search" method="post" enctype="multipart/form-data" action="api/upload">
    <div>
        <input type="radio" name="trip" value="round-trip"/>
        Round-Trip
    </div>
    <div>
        <input type="radio" name="trip" value="one-way"/>
        One-Way
    </div>

    <div>
        <input type="checkbox" name="options" value="nonstop" />
        Only show non-stop flights
    </div>
    <div>
        <input type="checkbox" name="options" value="airports" />
        Compare nearby airports
    </div>
    <div>
        <input type="checkbox" name="options" value="dates" />
        My travel dates are flexible
    </div>

    <div>
        <label for="seat">Seating Preference</label>
        <select name="seat">
            <option value="aisle">Aisle</option>
            <option value="window">Window</option>
            <option value="center">Center</option>
            <option value="none">No Preference</option>
        </select>
    </div>
</form>
```



The request body might look like this:

```
-----------------------------7dc1d13623304d6
Content-Disposition: form-data; name="trip"

round-trip
-----------------------------7dc1d13623304d6
Content-Disposition: form-data; name="options"

nonstop
-----------------------------7dc1d13623304d6
Content-Disposition: form-data; name="options"

dates
-----------------------------7dc1d13623304d6
Content-Disposition: form-data; name="seat"

window
-----------------------------7dc1d13623304d6--
```

In that case, the **FormData** collection would contain the following key/value pairs:

- trip: round-trip
- options: nonstop
- options: dates
- seat: window

HTTP Cookies in ASP.NET Web API

2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson

This topic describes how to send and receive HTTP cookies in Web API.

Background on HTTP Cookies

This section gives a brief overview of how cookies are implemented at the HTTP level. For details, consult RFC 6265.

A cookie is a piece of data that a server sends in the HTTP response. The client (optionally) stores the cookie and returns it on subsequent requests. This allows the client and server to share state. To set a cookie, the server includes a Set-Cookie header in the response. The format of a cookie is a name-value pair, with optional attributes. For example:

```
Set-Cookie: session-id=1234567
```

Here is an example with attributes:

```
Set-Cookie: session-id=1234567; max-age=86400; domain=example.com; path=/;
```

To return a cookie to the server, the client includes a Cookie header in later requests.

```
Cookie: session-id=1234567
```

An HTTP response can include multiple Set-Cookie headers.

```
Set-Cookie: session-token=abcdef;
Set-Cookie: session-id=1234567;
```

The client returns multiple cookies using a single Cookie header.

```
Cookie: session-id=1234567; session-token=abcdef;
```

The scope and duration of a cookie are controlled by following attributes in the Set-Cookie header:

- **Domain**: Tells the client which domain should receive the cookie. For example, if the domain is "example.com", the client returns the cookie to every subdomain of example.com. If not specified, the domain is the origin server.
- **Path**: Restricts the cookie to the specified path within the domain. If not specified, the path of the request URI is used.
- **Expires**: Sets an expiration date for the cookie. The client deletes the cookie when it expires.
- **Max-Age**: Sets the maximum age for the cookie. The client deletes the cookie when it reaches the maximum age.

If both `Expires` and `Max-Age` are set, `Max-Age` takes precedence. If neither is set, the client deletes the cookie when the current session ends. (The exact meaning of "session" is determined by the user-agent.)

However, be aware that clients may ignore cookies. For example, a user might disable cookies for privacy reasons. Clients may delete cookies before they expire, or limit the number of cookies stored. For privacy reasons, clients often reject "third party" cookies, where the domain does not match the origin server. In short, the server should not rely on getting back the cookies that it sets.

Cookies in Web API

To add a cookie to an HTTP response, create a **CookieHeaderValue** instance that represents the cookie. Then call the **AddCookies** extension method, which is defined in the **System.Net.Http.HttpResponseHeadersExtensions** class, to add the cookie.

For example, the following code adds a cookie within a controller action:

```
public HttpResponseMessage Get()
{
    var resp = new HttpResponseMessage();

    var cookie = new CookieHeaderValue("session-id", "12345");
    cookie.Expires = DateTimeOffset.Now.AddDays(1);
    cookie.Domain = Request.RequestUri.Host;
    cookie.Path = "/";

    resp.Headers.AddCookies(new CookieHeaderValue[] { cookie });
    return resp;
}
```

Notice that **AddCookies** takes an array of **CookieHeaderValue** instances.

To extract the cookies from a client request, call the **GetCookies** method:

```
string sessionId = "";

CookieHeaderValue cookie = Request.Headers.GetCookies("session-id").FirstOrDefault();
if (cookie != null)
{
    sessionId = cookie["session-id"].Value;
}
```

A **CookieHeaderValue** contains a collection of **CookieState** instances. Each **CookieState** represents one cookie. Use the indexer method to get a **CookieState** by name, as shown.

Structured Cookie Data

Many browsers limit how many cookies they will store—both the total number, and the number per domain. Therefore, it can be useful to put structured data into a single cookie, instead of setting multiple cookies.

> **NOTE**
> RFC 6265 does not define the structure of cookie data.

Using the **CookieHeaderValue** class, you can pass a list of name-value pairs for the cookie data. These name-value pairs are encoded as URL-encoded form data in the Set-Cookie header:

```
var resp = new HttpResponseMessage();

var nv = new NameValueCollection();
nv["sid"] = "12345";
nv["token"] = "abcdef";
nv["theme"] = "dark blue";
var cookie = new CookieHeaderValue("session", nv);

resp.Headers.AddCookies(new CookieHeaderValue[] { cookie });
```

The previous code produces the following Set-Cookie header:

```
Set-Cookie: session=sid=12345&token=abcdef&theme=dark+blue;
```

The **CookieState** class provides an indexer method to read the sub-values from a cookie in the request message:

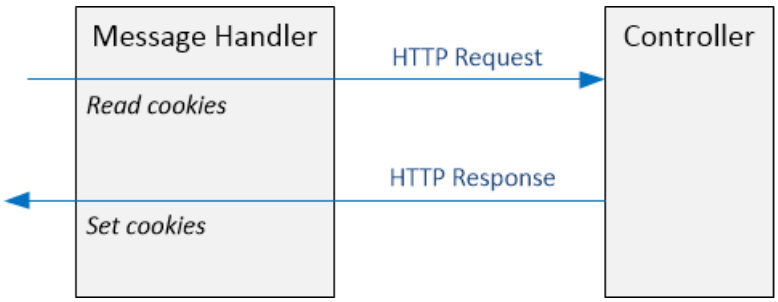
```
string sessionId = "";
string sessionToken = "";
string theme = "";

CookieHeaderValue cookie = Request.Headers.GetCookies("session").FirstOrDefault();
if (cookie != null)
{
    CookieState cookieState = cookie["session"];

    sessionId = cookieState["sid"];
    sessionToken = cookieState["token"];
    theme = cookieState["theme"];
}
```

Example: Set and Retrieve Cookies in a Message Handler

The previous examples showed how to use cookies from within a Web API controller. Another option is to use message handlers. Message handlers are invoked earlier in the pipeline than controllers. A message handler can read cookies from the request before the request reaches the controller, or add cookies to the response after the controller generates the response.



The following code shows a message handler for creating session IDs. The session ID is stored in a cookie. The handler checks the request for the session cookie. If the request does not include the cookie, the handler generates a new session ID. In either case, the handler stores the session ID in the **HttpRequestMessage.Properties**

property bag. It also adds the session cookie to the HTTP response.

This implementation does not validate that the session ID from the client was actually issued by the server. Don't use it as a form of authentication! The point of the example is to show HTTP cookie management.

```
using System;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading;
using System.Threading.Tasks;
using System.Web.Http;

public class SessionIdHandler : DelegatingHandler
{
    public static string SessionIdToken = "session-id";

    async protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        string sessionId;

        // Try to get the session ID from the request; otherwise create a new ID.
        var cookie = request.Headers.GetCookies(SessionIdToken).FirstOrDefault();
        if (cookie == null)
        {
            sessionId = Guid.NewGuid().ToString();
        }
        else
        {
            sessionId = cookie[SessionIdToken].Value;
            try
            {
                Guid guid = Guid.Parse(sessionId);
            }
            catch (FormatException)
            {
                // Bad session ID. Create a new one.
                sessionId = Guid.NewGuid().ToString();
            }
        }

        // Store the session ID in the request property bag.
        request.Properties[SessionIdToken] = sessionId;

        // Continue processing the HTTP request.
        HttpResponseMessage response = await base.SendAsync(request, cancellationToken);

        // Set the session ID as a cookie in the response message.
        response.Headers.AddCookies(new CookieHeaderValue[] {
            new CookieHeaderValue(SessionIdToken, sessionId)
        });

        return response;
    }
}
```

A controller can get the session ID from the **HttpRequestMessage.Properties** property bag.

```
public HttpResponseMessage Get()
{
    string sessionId = Request.Properties[SessionIdHandler.SessionIdToken] as string;

    return new HttpResponseMessage()
    {
        Content = new StringContent("Your session ID = " + sessionId)
    };
}
```

ASP.NET Web API Releases

2/20/2020 • 2 minutes to read • Edit Online

What's new in recent Web API releases.

- What's New in Web API OData 5.3
- What's New in ASP.NET Web API 2.2
- What's New in ASP.NET Web API 2.1

What's New in ASP.NET Web API OData 5.3

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

This topic describes what's new for ASP.NET Web API OData 5.3.

- Download
- Documentation
- OData Core Libraries
- New Features
- Known Issues and Breaking Changes
- Bug Fixes
- ASP.NET Web API OData 5.3.1

Download

The runtime features are released as NuGet packages on the NuGet gallery. You can install or update to the released NuGet packages by using the NuGet Package Manager Console:

```
Install-Package Microsoft.AspNet.OData -Version 5.3.0
Install-Package Microsoft.AspNet.WebApi.OData -Version 5.3.0
```

Documentation

You can find tutorials and other documentation about ASP.NET Web API OData at the ASP.NET web site.

OData Core Libraries

For OData v4, Web API now uses ODataLib version 6.5.0

New Features in ASP.NET Web API OData 5.3

Support for $levels in $expand

You can use the $levels query option in $expand queries. For example:

```
http://example.com/Employees?$expand=Manager($levels=2)
```

This query is equivalent to:

```
http://example.com/Employees?$expand=Manager($expand=Manager))
```

Support for Open Entity Types

An *open type* is a structured type that contains dynamic properties, in addition to any properties that are declared in the type definition. Open types let you add flexibility to your data models. For more information, see xxxx.

Support for dynamic collection properties in open types

Previously, a dynamic property had to be a single value. In 5.3, dynamic properties can have collection values. For

example, in the following JSON payload, the `Emails` property is a dynamic property and is of collection of string type:

```
{
   "Id": 1,
   "Name": "Ben",
   "Emails@odata.type": "#Collection(Edm.String)",
   "Emails": [
      "a@a.com",
      "b@b.com"
   ]
}
```

Support for inheritance for complex types

Now complex types can inherit from a base type. For example, an OData service could define the following complex types:

```
public abstract class Shape
{
    public bool HasBorder { get; set; }
}

public class Point
{
    public int X { get; set; }
    public int Y { get; set; }
}

public class Circle : Shape
{
    public Point Center { get; set; }
    public int Radius { get; set; }

    public override string ToString()
    {
        return "{" + Center.X + "," + Center.Y + "," + Radius + "}";
    }
}

public class Polygon : Shape
{
    public IList<Point> Vertexes { get; set; }
    public Polygon()
    {
        Vertexes = new List<Point>();
    }
}
```

Here is the EDM for this example:

```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="4.0" xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx">
  <edmx:DataServices>
    <Schema Namespace="ODataComplexTypeInheritanceSample" xmlns="http://docs.oasis-open.org/odata/ns/edm">
      <ComplexType Name="Shape" Abstract="true">
        <Property Name="HasBorder" Type="Edm.Boolean" Nullable="false" />
      </ComplexType>
      <ComplexType Name="Polygon" BaseType="ODataComplexTypeInheritanceSample.Shape">
        <Property Name="Vertexes" Type="Collection(ODataComplexTypeInheritanceSample.Point)" />
      </ComplexType>
      <ComplexType Name="Point">
        <Property Name="X" Type="Edm.Int32" Nullable="false" />
        <Property Name="Y" Type="Edm.Int32" Nullable="false" />
      </ComplexType>
      <ComplexType Name="Circle" BaseType="ODataComplexTypeInheritanceSample.Shape">
        <Property Name="Center" Type="ODataComplexTypeInheritanceSample.Point" />
        <Property Name="Radius" Type="Edm.Int32" Nullable="false" />
      </ComplexType>
      <EntityContainer Name="Container">
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

For more information, see OData Complex Type Inheritance Sample.

Known Issues and Breaking Changes

This section describes known issues and breaking changes in the ASP.NET Web API OData 5.3.

OData v4

Query Options

Issue: Using nested $expand with $levels=max results in an incorrect expansion depth.

For example, given the following request:

```
~/Entities(6)?$expand=P($levels=2;$expand=D($levels=max))
```

If `MaxExpansionDepth` is 5, this query would result in an expansion depth of 6.

Bug Fixes and Minor Feature Updates

This release also includes several bug fixes and minor feature updates. You can find the complete list here:

- Bug fixes

ASP.NET Web API OData 5.3.1

In this release we made a bug fix to some of the AllowedFunctions enums. This release doesn't have any other bug fixes or new features.

What's New in ASP.NET Web API 2.2

4/10/2020 • 6 minutes to read • Edit Online

by Microsoft

This topic describes what's new for ASP.NET Web API 2.2.

- Download
- Documentation
- New Features in ASP.NET Web API 2.2
  - OData v4
  - Attribute Routing Improvements
  - Web API Client support for Windows Phone 8.1
- Known Issues and Breaking Changes
- Bug Fixes
- Microsoft.AspNet.OData 5.2.1
- Microsoft.AspNet.WebAPI 5.2.2
- Microsoft.AspNet.WebAPI 5.2.3 Beta

Download

The runtime features are released as NuGet packages on the NuGet gallery. All the runtime packages follow the Semantic Versioning specification. The latest ASP.NET Web API 2.2 package has the following version: "5.2.0". You can install or update these packages through NuGet. The release also includes corresponding localized packages on NuGet.

You can install or update to the released NuGet packages by using the NuGet Package Manager Console:

```
Install-Package Microsoft.AspNet.WebApi -Version 5.2.0
```

Documentation

Tutorials and other information about ASP.NET Web API 2.2 are available from the ASP.NET web site (https://www.asp.net/web-api).

New Features in ASP.NET Web API 2.2

OData v4

This release adds support for the OData v4 protocol. For more information, see the Web API OData v4 documentation.

Here are some of the key features and changes for OData v4:

- Support for aliasing properties in OData model
- Support for ComplexTypeAttribute, AssociationAttribute, TimesTampAttribute and ConcurrencyCheckAttribute in ODataConventionModelBuilder

- Provide ability to supply friendly Title for actions
- Integrate with ODL UriParser
- Support for enum, containment and singleton
- Support cast for primitive types
- Added OData function support
- Support parameter aliases for function calls
- Support camel case naming convention in model
- Support for cast() in $filter
- Support for open complex type
- Removed EntitySetController and AsyncEntitySetController
- Changed $link to $ref
- Added Attribute routing support
- Uses OData Core Libraries 6.4.0

Attribute Routing Improvements

Attribute Routing now provides an extensibility point called IDirectRouteProvider, which allows full control over how attribute routes are discovered and configured. An IDirectRouteProvider is responsible for providing a list of actions and controllers along with associated route information to specify exactly what routing configuration is desired for those actions. An IDirectRouteProvider implementation may be specified when calling MapAttributes/MapHttpAttributeRoutes.

Customizing IDirectRouteProvider will be easiest by extending our default implementation, DefaultDirectRouteProvider. This class provides separate overridable virtual methods to change the logic for discovering attributes, creating route entries, and discovering route prefix and area prefix.

Following are some examples on what you could do with this new extensibility point:

1. Support inheritance of Route attributes

   Example:

   Here a request like "/api/values/10" would successfully return "Success:10"

   ```
   public class BaseController : ApiController
   {
       [Route("{id:int}")]
       public string Get(int id)
       {
           return "Success:" + id;
       }
   }
   [RoutePrefix("api/values")]
   public class ValuesController : BaseController
   {
   }

   config.MapHttpAttributeRoutes(new CustomDirectRouteProvider());
   public class CustomDirectRouteProvider : DefaultDirectRouteProvider
   {
       protected override IReadOnlyList<IDirectRouteFactory>
       GetActionRouteFactories(HttpActionDescriptor actionDescriptor)
       {
           return actionDescriptor.GetCustomAttributes<IDirectRouteFactory>
           (inherit: true);
       }
   }
   ```

2. Provide a default route name for your attribute routes by following some convention you like. By default,

attribute routing doesn't automatically create names for attribute routes.

3. Modify attribute routes' route template at one central place before they end up in the route table.

Web API Client Support for Windows Phone 8.1

You can now use the Web API Client NuGet package to implement your Web API client logic when targeting Windows Phone 8.1 or from within a Universal App.

Known Issues and Breaking Changes

This section describes known issues and breaking changes in the ASP.NET Web API 2.2.

OData v4

Model builder

Issue: Overloaded Functions could not be exposed as FunctionImport

If there are 2 overloaded functions and they are also FunctionImport as shown below then requesting ~/GetAllConventionCustomers(CustomerName={customerName}) results in System.InvalidOperationException.

```
<Function Name="GetAllConventionCustomers"
ReturnType="Collection(WebStack.QA.Test.OData.UnboundOperation.ConventionCustomer)"
IsComposable="true" />
<Function Name="GetAllConventionCustomers"
ReturnType="Collection(WebStack.QA.Test.OData.UnboundOperation.ConventionCustomer)"
IsComposable="true">
<Parameter Name="CustomerName" Type="Edm.String" FixedLength="false"
Unicode="false" />
</Function>
...
<FunctionImport Name="GetAllConventionCustomers"
Function="WebStack.QA.Test.OData.UnboundOperation.GetAllConventionCustomers"
EntitySet="ConventionCustomers" IncludeInServiceDocument="true" />
```

Workaround: The workaround for this issue is to add both the function overloads as FunctionImports.

OData Routing

String literals that include the URL encoded slash (%2F), and backslash(%5C) cause a 404 error when they are used in the OData resource paths.

For example, string literals can be used in the OData resource paths as parameters of functions or key values of entity sets.

/Employees/Total.GetCount(Name='Name%2F')

/Employees('Name%5C')

When services receive such requests the hosts will un-escape those escape sequences before passing them to the Web API runtime. This protects against attacks like the following:

```
http://www.contoso.com/..%2F..%2F/Windows/System32/cmd.exe?/c+dir+c:
```

This causes the Web API OData stack to return a 404 error (Not Found). To prevent this error, your client should use the double escape sequences for slash (%252F) and backslash (%255C). This does not happen for query strings such as /Employees?$filter=Name eq 'Name%2F'

Note un-escaped slashes ('/') and backslashes ('') are not legal in OData resource path string literals. Slashes should appear only as path separators and backslashes should not appear in the OData resource path at all. (Both are usable in some portions of an OData query string.)

Workaround: You could override the Parse method of DefaultODataPathHandler to escape the slash and backslash

in string literals before actually parsing them. You can find a sample of this approach here.

OData v3

[Queryable]

The [Queryable] attribute is deprecated. New OData v3 applications should use **System.Web.Http.OData.EnableQueryAttribute**.

The **ODataHttpConfigurationExtensions.EnableQuerySupport** extension method now adds an **EnableQueryAttribute** to the global filter collection. If any controllers have the **[Queryable]** attribute, calling `config.EnableQuerySupport()` will cause the **[Queryable]** attribute to fail

The recommended way to resolve this issue is to replace all instances of **QueryableAttribute** with **System.Web.Http.OData.EnableQueryAttribute**.

An alternative workaround is to use the following code in your Web API configuration:

```
config.EnableQuerySupport(new QueryableAttribute());
```

Attribute Routing

Issue: Model binding of complex type which is decorated with FromUri attribute behaves differently when using Attribute Routing.

Following link is tracking the issue and also has details about a workaround.
http://aspnetwebstack.codeplex.com/workitem/1944

Issue: Scaffolding MVC/Web API into a project with 5.2.0 packages results in 5.1.2 packages for ones that don't already exist in the project

Updating NuGet packages for ASP.NET MVC 5.2 does not update the Visual Studio tools such as ASP.NET scaffolding or the ASP.NET Web Application project template. They use the previous version of the ASP.NET runtime packages (e.g. 5.1.2 in Update 2). As a result, the ASP.NET scaffolding will install the previous version (e.g. 5.1.2 in Update 2) of the required packages, if they are not already available in your projects. However, the ASP.NET scaffolding in Visual Studio 2013 RTM or Update 1 does not overwrite the latest packages in your projects. If you use ASP.NET scaffolding after updating the packages of your projects to Web API 2.2 or ASP.NET MVC 5.2, make sure the versions of Web API and ASP.NET MVC are consistent.

Bug Fixes and Minor Feature Updates

This release also includes several bug fixes and minor feature updates. You can find the complete list here:

- 5.2 package

Microsoft.AspNet.OData 5.2.1

The Microsoft.AspNet.OData 5.2.1 package contains NuGet dependency updates but no bug fixes. With this update, there is no longer a strict dependency on Microsoft.OData.Core 6.4.0, but one can upgrade to any version between 6.4.0 and 7.0.0.

Microsoft.AspNet.WebAPI 5.2.2

In this release we have made a dependency change for `Json.Net 6.0.4`. For more information on what is new in this release of `Json.NET`, see Json.NET 6.0 Release 4 - JSON Merge, Dependency Injection. This release doesn't have any other new features or bug fixes in Web API. We have subsequently updated all other dependent packages we own to depend on this new version of Web API.

Microsoft.AspNet.WebAPI 5.2.3 Beta

You can read about the release here. This release contains only bug fixes. You can use this query to see the list of issues fixed in this release.

What's New in ASP.NET Web API 2.1

4/10/2020 • 5 minutes to read • Edit Online

by Microsoft

This topic describes what's new for ASP.NET Web API 2.1.

- Download
- Documentation
- New Features in ASP.NET Web API 2.1
  - Global Error Handling
  - Attribute Routing Improvements
  - Help Page Improvements
  - IgnoreRoute Support
  - BSON Media-Type Formatter
  - Better Support for Async Filters
  - Query Parsing for the Client Formatting Library
- Known Issues and Breaking Changes
- Bug Fixes

Download

The runtime features are released as NuGet packages on the NuGet gallery. All the runtime packages follow the Semantic Versioning specification. The latest ASP.NET Web API 2.1 RTM package has the following version: "5.1.2". You can install or update these packages through NuGet. The release also includes corresponding localized packages on NuGet.

You can install or update to the released NuGet packages by using the NuGet Package Manager Console:

```
Install-Package Microsoft.AspNet.WebApi -Version 5.1.2
```

Documentation

Tutorials and other information about ASP.NET Web API 2.1 RTM are available from the ASP.NET web site (https://www.asp.net/web-api).

New Features in ASP.NET Web API 2.1

Global Error Handling

All unhandled exceptions can now be logged through one central mechanism, and the behavior for unhandled exceptions can be customized.

The framework supports multiple exception loggers, which all see the unhandled exception and information about the context in which it occurred, such as the request being processed at the time.

For example, the following code uses System.Diagnostics.TraceSource to log all unhandled exceptions:

```
public class TraceSourceExceptionLogger : ExceptionLogger
{
    private readonly TraceSource _traceSource;

    public TraceSourceExceptionLogger(TraceSource traceSource)
    {
        _traceSource = traceSource;
    }

    public override void Log(ExceptionLoggerContext context)
    {
        _traceSource.TraceEvent(TraceEventType.Error, 1,
            "Unhandled exception processing {0} for {1}: {2}",
            context.Request.Method,
            context.Request.RequestUri,
            context.Exception);
    }
}

config.Services.Add(typeof(IExceptionLogger),
    new TraceSourceExceptionLogger(new
    TraceSource("MyTraceSource", SourceLevels.All)));
```

You can also replace the default exception handler, so that you can fully customize the HTTP response message that is sent when an unhandled exception occurs.

We have provided a sample that logs all unhandled exceptions via the popular ELMAH framework.

Attribute Routing Improvements

Attribute routing now supports constraints, enabling versioning and header-based route selection. Also, many aspects of attribute routes are now customizable via the **IDirectRouteFactory** interface and **RouteFactoryAttribute** class. The route prefix is now extensible via the **IRoutePrefix** interface and **RoutePrefixAttribute** class.

We have provided a sample that uses constraints to dynamically filter controllers by an 'api-version' HTTP header.

Help Page Improvements

Web API 2.1 includes the following enhancements to API Help Pages:

- Documentation of individual properties of parameters or return types of actions.
- Documentation of data model annotations.

The UI design of the help pages was also updated, to accommodate these changes.

IgnoreRoute Support

Web API 2.1 supports ignoring URL patterns in Web API routing, through a set of **IgnoreRoute** extension methods on **HttpRouteCollection**. These methods cause Web API to ignore any URLs that match a specified template, and allow the host to apply additional processing if appropriate.

The following example ignores URIs that start with a "content" segment:

```
routes.IgnoreRoute("IgnoreContent", "content/{*paths}");
routes.MapHttpRoute("Default", "{controller}/{id}");
```

BSON Media-Type Formatter

Web API now supports the BSON wire format, both on the client and on the server.

To enable BSON on the server side, add the **BsonMediaTypeFormatter** to the formatters collection:

```
config.Formatters.Add(new BsonMediaTypeFormatter());
```

Here is how a .NET client can consume BSON format:

```
// Add Accept header.
client.DefaultRequestHeaders.Accept.Add(
    new MediaTypeWithQualityHeaderValue("application/bson"));

// POST data in BSON format.
HttpResponseMessage response = await client.PostAsync<MyData>("api/MyData", data, new
BsonMediaTypeFormatter());

// GET data in BSON format.
data = await response.Content.ReadAsAsync<MyData>(new MediaTypeFormatter[] {
  new BsonMediaTypeFormatter() });
```

We have provided a sample that shows both the client and server side.

For more information, see BSON Support in Web API 2.1

Better Support for Async Filters

Web API now supports an easy way to create filters that execute asynchronously. This feature is useful is your filter needs to perform an async action, such as access a database. Previously, to create an async filter, you had to implement the filter interface yourself, because the filter base classes only exposed synchronous methods. Now you can override the virtual `On*Async` methods of the filter base class.

For example:

```
public class AsyncLoggingFilter : ActionFilterAttribute
{
    public override async Task OnActionExecutingAsync(HttpActionContext actionContext, CancellationToken
cancellationToken)
    {
        await Trace.WriteAsync("Executing action named {0} for request {1}.",
            actionContext.ActionDescriptor.ActionName,
            actionContext.Request.GetCorrelationId());
    }
}
```

The **AuthorizationFilterAttribute**, **ActionFilterAttribute**, and **ExceptionFilterAttribute** classes all support async in Web API 2.1.

Query Parsing for the Client Formatting Library

Previously, **System.Net.Http.Formatting** supported parsing and updating URI queries for server-side code, but the equivalent portable library was missing this feature. In Web API 2.1, a client application can now easily parse and update a query string.

The following examples show how to parse, modify, and generate URI queries. (The examples show a console application for simplicity.)

```
// Query parsing
HttpValueCollection collection = new Uri("http://api/something?catId=3&catId=4&dogId=1,2").ParseQueryString();

Console.WriteLine(collection["catId"]); // output: 3,4
Console.WriteLine(collection["dogId"]); // output: 1,2

// Modify the query
collection.Add("dogId", "7");

// Index into the values
Console.WriteLine(collection["catId"]); // output: 3,4
Console.WriteLine(collection["dogId"]); // output: 1,2,7

// Recreate the query string
Console.WriteLine(collection.ToString()); // output: catId=3&catId=4&dogId=1%2C2&dogId=7

// Query generation
HttpValueCollection newCollection = new HttpValueCollection();

newCollection.Add("catId", "1");
newCollection.Add("dogId", "7");

// Index into the values
Console.WriteLine(newCollection["catId"]); // output: 1
Console.WriteLine(newCollection["dogId"]); // output: 7

// Create the query string
Console.WriteLine(newCollection.ToString()); // catId=1&dogId=7
```

Known Issues and Breaking Changes

This section describes known issues and breaking changes in the ASP.NET Web API 2.1 RTM.

Attribute Routing

Ambiguities in attribute routing matches now report an error rather than choosing the first match.

Attribute routes are prohibited from using the *{controller}* parameter, and from using the *{action}* parameter on routes placed on actions. These parameters would very likely cause ambiguities.

Scaffolding MVC/Web API into a project with 5.1 packages results in 5.0 packages for ones that don't already exist in the project

Updating NuGet packages for ASP.NET Web API 2.1 RTM does not update the Visual Studio tools, such as ASP.NET scaffolding or the ASP.NET Web Application project template. They use the previous version of the ASP.NET runtime packages (5.0.0.0). As a result, the ASP.NET scaffolding will install the previous version (5.0.0.0) of the required packages, if they are not already available in your projects. However, the ASP.NET scaffolding in Visual Studio 2013 RTM or Update 1 does not overwrite the latest packages in your projects.

If you use ASP.NET scaffolding after updating the packages to Web API 2.1 or ASP.NET MVC 5.1, make sure the versions of Web API and MVC are consistent.

Type Renames

Some of the types used for attribute routing extensibility were renamed from the RC to the 2.1 RTM.

OLD TYPE NAME (2.1 RC)	NEW TYPE NAME (2.1 RTM)
IDirectRouteProvider	IDirectRouteFactory
RouteProviderAttribute	RouteFactoryAttribute

OLD TYPE NAME (2.1 RC)	NEW TYPE NAME (2.1 RTM)
DirectRouteProviderContext	DirectRouteFactoryContext

Exception filters do not unwrap aggregate exceptions thrown in async actions

Previously, if an async action threw an **AggregateException**, an exception filter would unwrap the exception, and **OnException** would get the base exception. In 2.1, the exception filter does not unwrap it, and **OnException** gets the original **AggregateException**.

Bug Fixes

This release also includes several bug fixes. You can find the complete list here:

- 5.1.0 package
- 5.1.1 package

The 5.1.2 package contains IntelliSense updates but no bug fixes.

Documentation on Older Versions of Web API

2/20/2020 • 2 minutes to read • Edit Online

Documentation on older versions of Web API for ASP.NET 4.x.

- Enabling CRUD Operations in Web API 1
- Using ASP.NET Web API 1 with Entity Framework 5
  - Overview and Creating the Project
  - Creating the Domain Models
  - Creating an Admin Controller
  - Adding an Admin View
  - Creating a Dynamic UI with Knockout.js
  - Creating Product and Order Controllers
  - Creating the Main Page
- Self-Host Web API 1 (C#)
- Build RESTful API's with ASP.NET Web API

Enabling CRUD Operations in ASP.NET Web API 1

2/20/2020 • 9 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

> This tutorial shows how to support CRUD operations in an HTTP service using ASP.NET Web API for ASP.NET 4.x.
>
> ## Software versions used in the tutorial
>
> - Visual Studio 2012
> - Web API 1 (also works with Web API 2)

CRUD stands for "Create, Read, Update, and Delete," which are the four basic database operations. Many HTTP services also model CRUD operations through REST or REST-like APIs.

In this tutorial, you will build a very simple web API to manage a list of products. Each product will contain a name, price, and category (such as "toys" or "hardware"), plus a product ID.

The products API will expose following methods.

ACTION	HTTP METHOD	RELATIVE URI
Get a list of all products	GET	/api/products
Get a product by ID	GET	/api/products/*id*
Get a product by category	GET	/api/products?category=*category*
Create a new product	POST	/api/products
Update a product	PUT	/api/products/*id*
Delete a product	DELETE	/api/products/*id*

Notice that some of the URIs include the product ID in path. For example, to get the product whose ID is 28, the client sends a GET request for `http://hostname/api/products/28`.

Resources

The products API defines URIs for two resource types:

RESOURCE	URI
The list of all the products.	/api/products
An individual product.	/api/products/*id*

Methods

The four main HTTP methods (GET, PUT, POST, and DELETE) can be mapped to CRUD operations as follows:

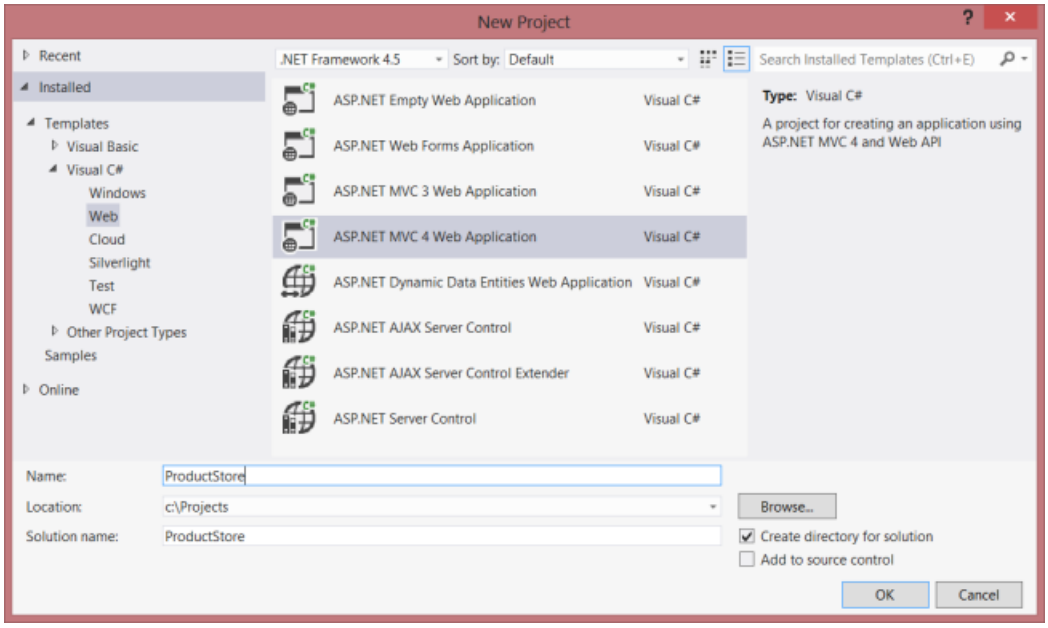
- GET retrieves the representation of the resource at a specified URI. GET should have no side effects on the server.
- PUT updates a resource at a specified URI. PUT can also be used to create a new resource at a specified URI, if the server allows clients to specify new URIs. For this tutorial, the API will not support creation through PUT.
- POST creates a new resource. The server assigns the URI for the new object and returns this URI as part of the response message.
- DELETE deletes a resource at a specified URI.

Note: The PUT method replaces the entire product entity. That is, the client is expected to send a complete representation of the updated product. If you want to support partial updates, the PATCH method is preferred. This tutorial does not implement PATCH.

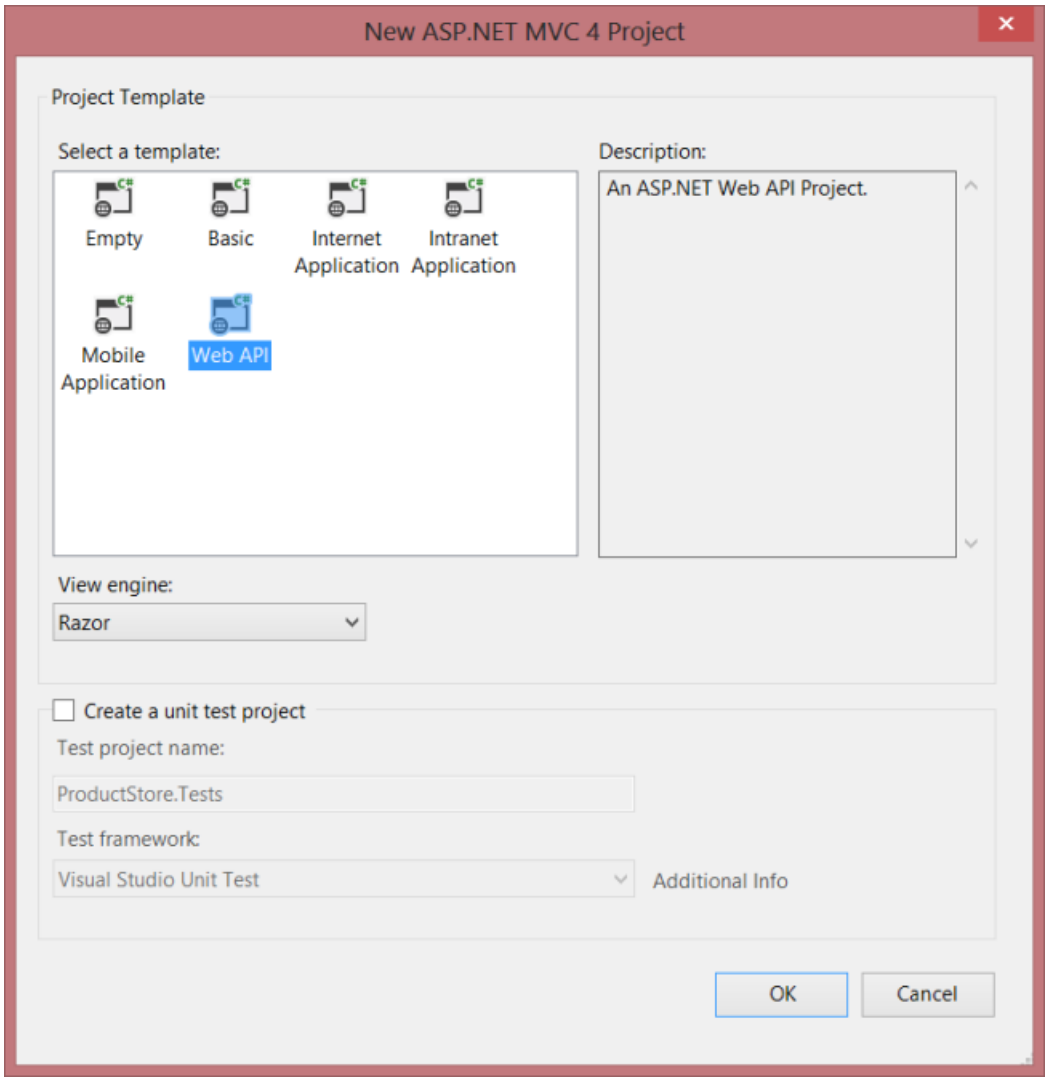
Create a New Web API Project

Start by running Visual Studio and select **New Project** from the **Start** page. Or, from the **File** menu, select **New** and then **Project**.

In the **Templates** pane, select **Installed Templates** and expand the **Visual C#** node. Under **Visual C#**, select **Web**. In the list of project templates, select **ASP.NET MVC 4 Web Application**. Name the project "ProductStore" and click **OK**.



In the **New ASP.NET MVC 4 Project** dialog, select **Web API** and click **OK**.

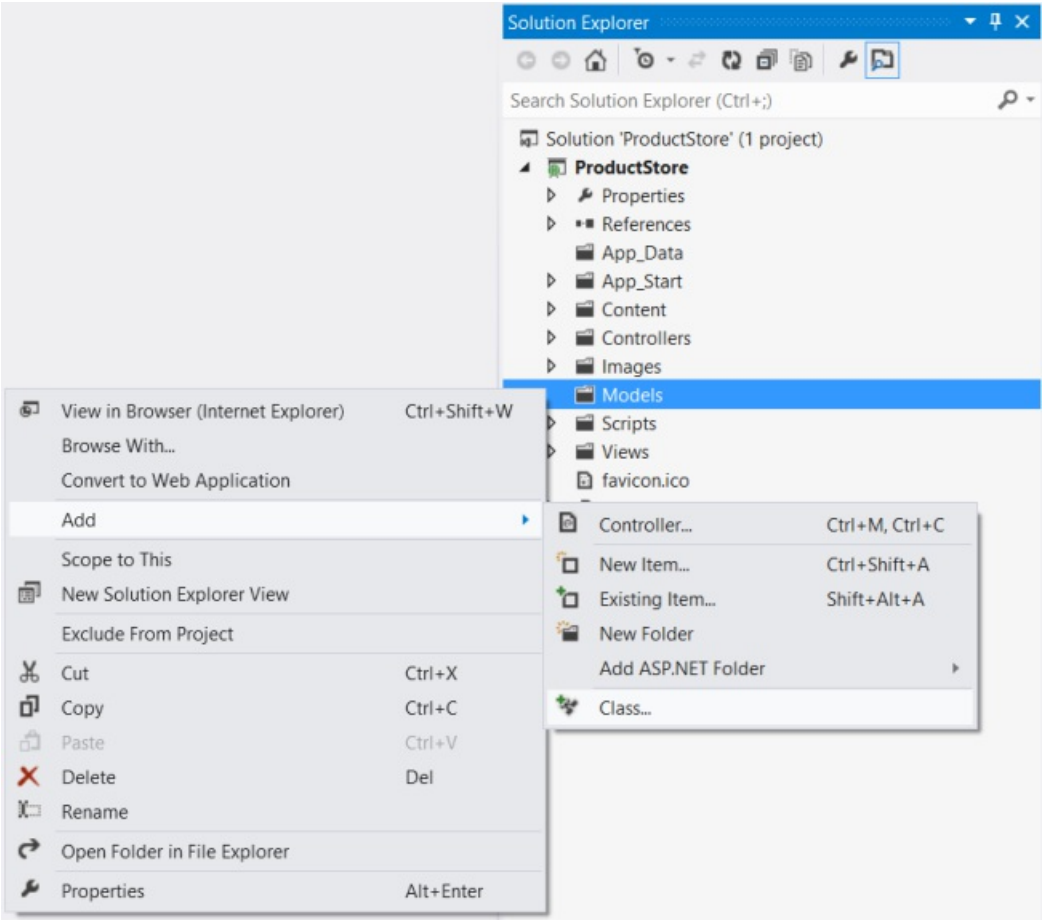


Adding a Model

A *model* is an object that represents the data in your application. In ASP.NET Web API, you can use strongly typed CLR objects as models, and they will automatically be serialized to XML or JSON for the client.

For the ProductStore API, our data consists of products, so we'll create a new class named `Product`.

If Solution Explorer is not already visible, click the **View** menu and select **Solution Explorer**. In Solution Explorer, right-click the **Models** folder. From the context menu, select **Add**, then select **Class**. Name the class "Product".



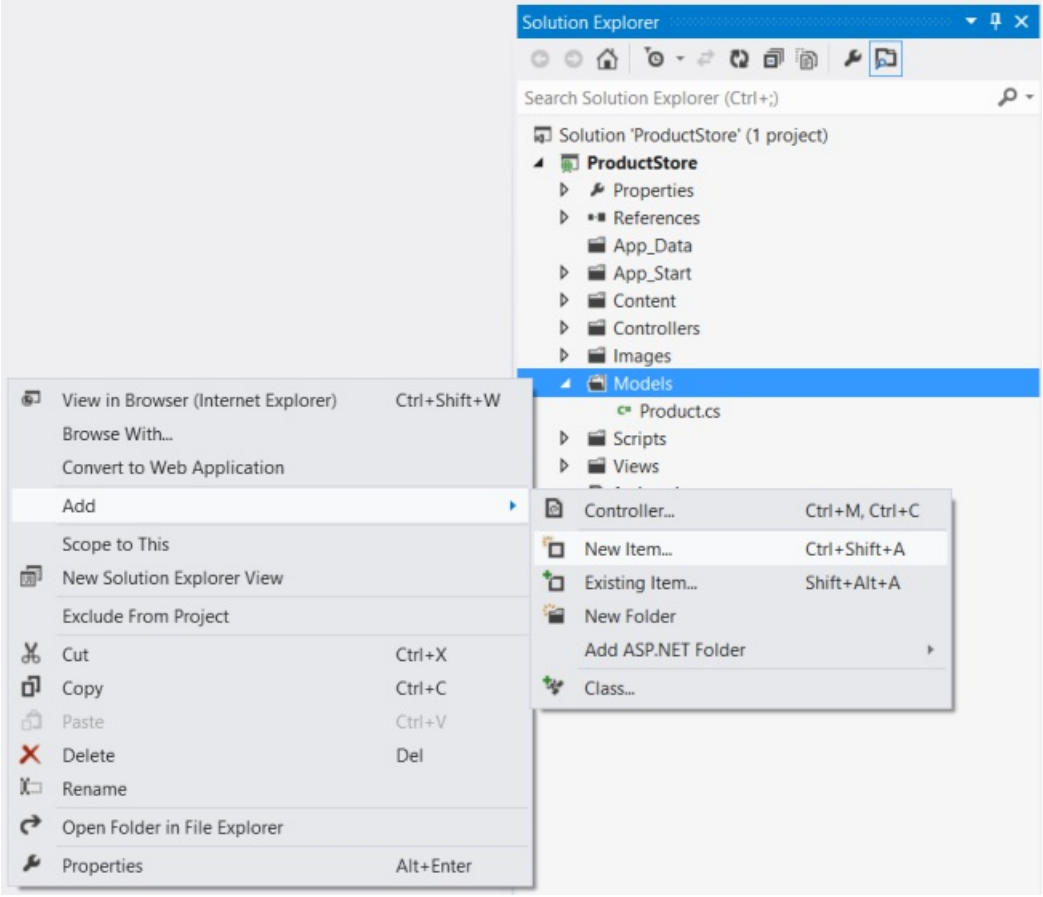
Add the following properties to the `Product` class.

```
namespace ProductStore.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public decimal Price { get; set; }
    }
}
```

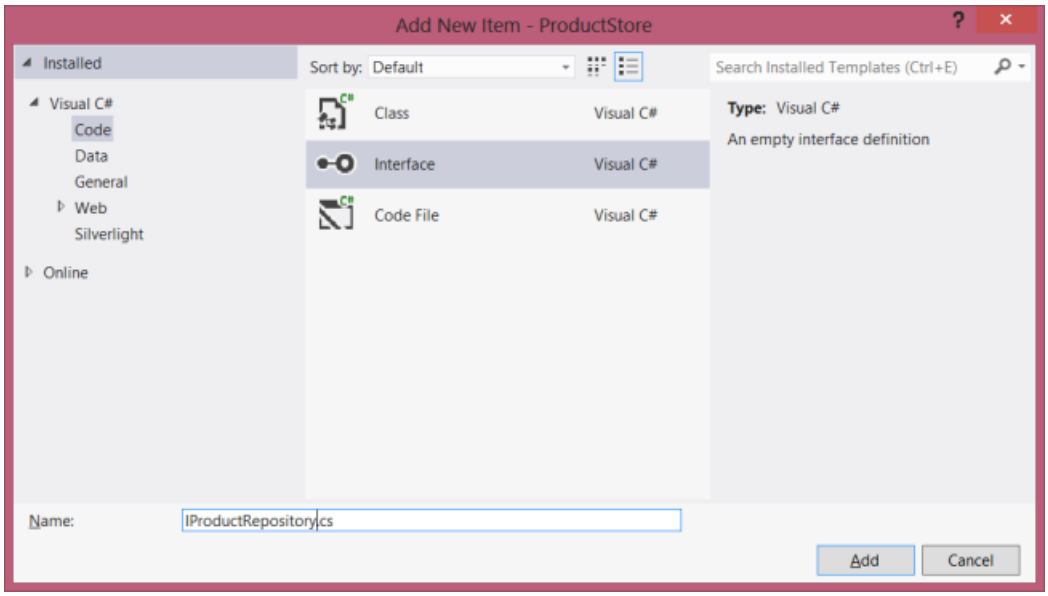
Adding a Repository

We need to store a collection of products. It's a good idea to separate the collection from our service implementation. That way, we can change the backing store without rewriting the service class. This type of design is called the *repository* pattern. Start by defining a generic interface for the repository.

In Solution Explorer, right-click the **Models** folder. Select **Add**, then select **New Item**.



In the **Templates** pane, select **Installed Templates** and expand the C# node. Under C#, select **Code**. In the list of code templates, select **Interface**. Name the interface "IProductRepository".



Add the following implementation:

```
namespace ProductStore.Models
{
    public interface IProductRepository
    {
        IEnumerable<Product> GetAll();
        Product Get(int id);
        Product Add(Product item);
        void Remove(int id);
        bool Update(Product item);
    }
}
```

Now add another class to the Models folder, named "ProductRepository." This class will implement the `IProductRepository` interface. Add the following implementation:

```
namespace ProductStore.Models
{
    public class ProductRepository : IProductRepository
    {
        private List<Product> products = new List<Product>();
        private int _nextId = 1;

        public ProductRepository()
        {
            Add(new Product { Name = "Tomato soup", Category = "Groceries", Price = 1.39M });
            Add(new Product { Name = "Yo-yo", Category = "Toys", Price = 3.75M });
            Add(new Product { Name = "Hammer", Category = "Hardware", Price = 16.99M });
        }

        public IEnumerable<Product> GetAll()
        {
            return products;
        }

        public Product Get(int id)
        {
            return products.Find(p => p.Id == id);
        }

        public Product Add(Product item)
        {
            if (item == null)
            {
                throw new ArgumentNullException("item");
            }
            item.Id = _nextId++;
            products.Add(item);
            return item;
        }

        public void Remove(int id)
        {
            products.RemoveAll(p => p.Id == id);
        }

        public bool Update(Product item)
        {
            if (item == null)
            {
                throw new ArgumentNullException("item");
            }
            int index = products.FindIndex(p => p.Id == item.Id);
            if (index == -1)
            {
                return false;
            }
            products.RemoveAt(index);
            products.Add(item);
            return true;
        }
    }
}
```

The repository keeps the list in local memory. This is OK for a tutorial, but in a real application, you would store the data externally, either a database or in cloud storage. The repository pattern will make it easier to change the implementation later.

Adding a Web API Controller

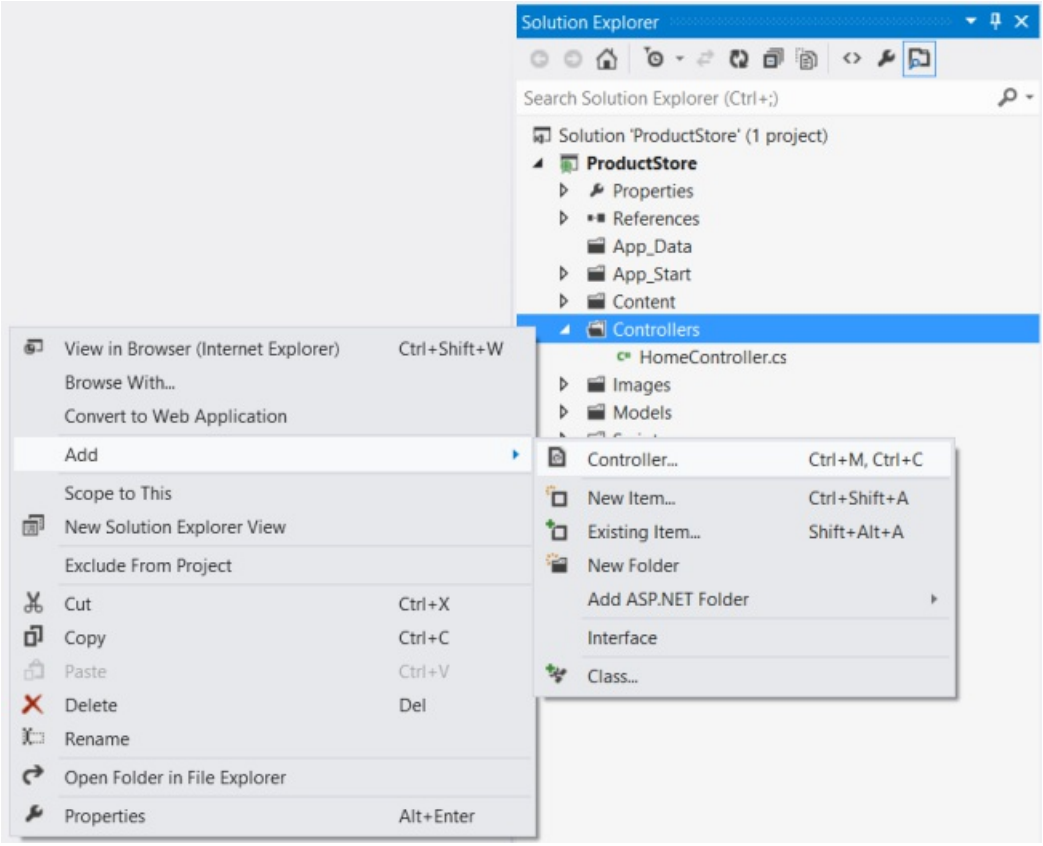
If you have worked with ASP.NET MVC, then you are already familiar with controllers. In ASP.NET Web API, a

controller is a class that handles HTTP requests from the client. The New Project wizard created two controllers for you when it created the project. To see them, expand the Controllers folder in Solution Explorer.

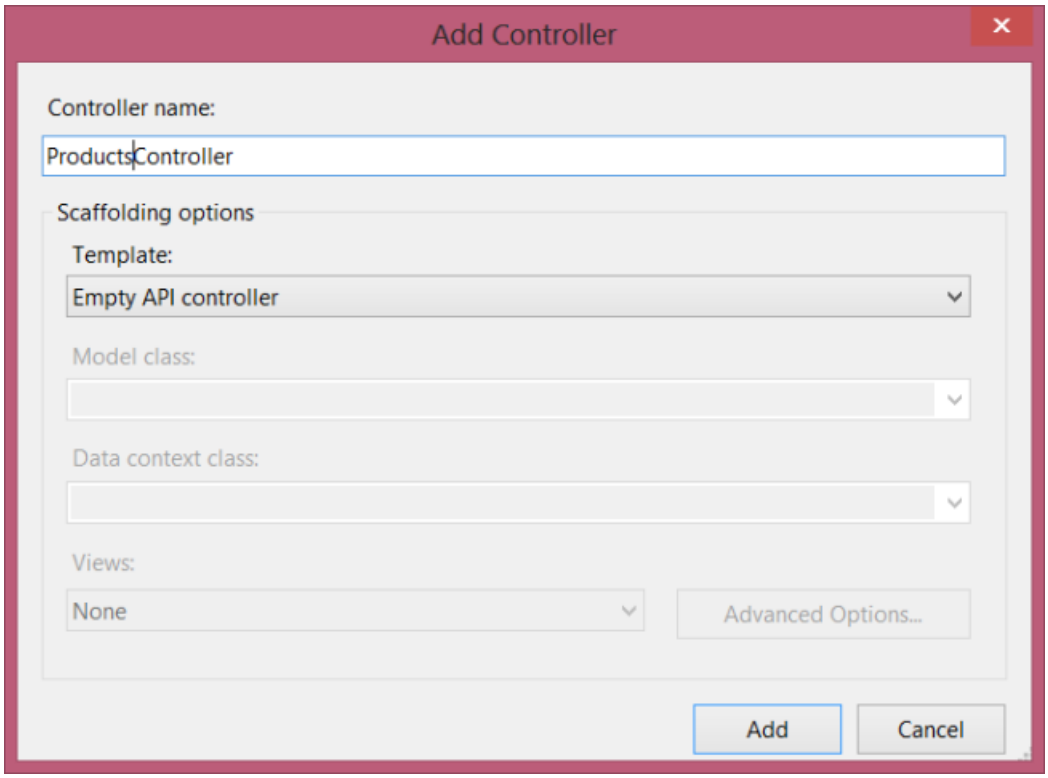
- HomeController is a traditional ASP.NET MVC controller. It is responsible for serving HTML pages for the site, and is not directly related to our web API.
- ValuesController is an example WebAPI controller.

Go ahead and delete ValuesController, by right-clicking the file in Solution Explorer and selecting **Delete.** Now add a new controller, as follows:

In **Solution Explorer**, right-click the Controllers folder. Select **Add** and then select **Controller**.



In the **Add Controller** wizard, name the controller "ProductsController". In the **Template** drop-down list, select **Empty API Controller**. Then click **Add**.



> **NOTE**
> It is not necessary to put your controllers into a folder named Controllers. The folder name is not important; it is simply a convenient way to organize your source files.

The **Add Controller** wizard will create a file named ProductsController.cs in the Controllers folder. If this file is not open already, double-click the file to open it. Add the following **using** statement:

```
using ProductStore.Models;
```

Add a field that holds an **IProductRepository** instance.

```
public class ProductsController : ApiController
{
    static readonly IProductRepository repository = new ProductRepository();
}
```

> **NOTE**
> Calling `new ProductRepository()` in the controller is not the best design, because it ties the controller to a particular implementation of `IProductRepository`. For a better approach, see Using the Web API Dependency Resolver.

Getting a Resource

The ProductStore API will expose several "read" actions as HTTP GET methods. Each action will correspond to a method in the `ProductsController` class.

ACTION	HTTP METHOD	RELATIVE URI
Get a list of all products	GET	/api/products

ACTION	HTTP METHOD	RELATIVE URI
Get a product by ID	GET	/api/products/*id*
Get a product by category	GET	/api/products?category=*category*

To get the list of all products, add this method to the `ProductsController` class:

```
public class ProductsController : ApiController
{
    public IEnumerable<Product> GetAllProducts()
    {
        return repository.GetAll();
    }
    // ....
}
```

The method name starts with "Get", so by convention it maps to GET requests. Also, because the method has no parameters, it maps to a URI that does not contain an *"id"* segment in the path.

To get a product by ID, add this method to the `ProductsController` class:

```
public Product GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        throw new HttpResponseException(HttpStatusCode.NotFound);
    }
    return item;
}
```

This method name also starts with "Get", but the method has a parameter named *id*. This parameter is mapped to the "id" segment of the URI path. The ASP.NET Web API framework automatically converts the ID to the correct data type (**int**) for the parameter.

The GetProduct method throws an exception of type **HttpResponseException** if *id* is not valid. This exception will be translated by the framework into a 404 (Not Found) error.

Finally, add a method to find products by category:

```
public IEnumerable<Product> GetProductsByCategory(string category)
{
    return repository.GetAll().Where(
        p => string.Equals(p.Category, category, StringComparison.OrdinalIgnoreCase));
}
```

If the request URI has a query string, Web API tries to match the query parameters to parameters on the controller method. Therefore, a URI of the form "api/products?category=*category*" will map to this method.

Creating a Resource

Next, we'll add a method to the `ProductsController` class to create a new product. Here is a simple implementation of the method:

```
// Not the final implementation!
public Product PostProduct(Product item)
{
    item = repository.Add(item);
    return item;
}
```

Note two things about this method:

- The method name starts with "Post...". To create a new product, the client sends an HTTP POST request.
- The method takes a parameter of type Product. In Web API, parameters with complex types are deserialized from the request body. Therefore, we expect the client to send a serialized representation of a product object, in either XML or JSON format.

This implementation will work, but it is not quite complete. Ideally, we would like the HTTP response to include the following:

- **Response code:** By default, the Web API framework sets the response status code to 200 (OK). But according to the HTTP/1.1 protocol, when a POST request results in the creation of a resource, the server should reply with status 201 (Created).
- **Location:** When the server creates a resource, it should include the URI of the new resource in the Location header of the response.

ASP.NET Web API makes it easy to manipulate the HTTP response message. Here is the improved implementation:

```
public HttpResponseMessage PostProduct(Product item)
{
    item = repository.Add(item);
    var response = Request.CreateResponse<Product>(HttpStatusCode.Created, item);

    string uri = Url.Link("DefaultApi", new { id = item.Id });
    response.Headers.Location = new Uri(uri);
    return response;
}
```

Notice that the method return type is now **HttpResponseMessage**. By returning an **HttpResponseMessage** instead of a Product, we can control the details of the HTTP response message, including the status code and the Location header.

The **CreateResponse** method creates an **HttpResponseMessage** and automatically writes a serialized representation of the Product object into the body fo the response message.

> **NOTE**
>
> This example does not validate the `Product`. For information about model validation, see Model Validation in ASP.NET Web API.

Updating a Resource

Updating a product with PUT is straightforward:

```
public void PutProduct(int id, Product product)
{
    product.Id = id;
    if (!repository.Update(product))
    {
        throw new HttpResponseException(HttpStatusCode.NotFound);
    }
}
```

The method name starts with "Put...", so Web API matches it to PUT requests. The method takes two parameters, the product ID and the updated product. The *id* parameter is taken from the URI path, and the *product* parameter is deserialized from the request body. By default, the ASP.NET Web API framework takes simple parameter types from the route and complex types from the request body.

Deleting a Resource

To delete a resource, define a "Delete..." method.

```
public void DeleteProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        throw new HttpResponseException(HttpStatusCode.NotFound);
    }

    repository.Remove(id);
}
```

If a DELETE request succeeds, it can return status 200 (OK) with an entity-body that describes the status; status 202 (Accepted) if the deletion is still pending; or status 204 (No Content) with no entity body. In this case, the `DeleteProduct` method has a `void` return type, so ASP.NET Web API automatically translates this into status code 204 (No Content).

Using ASP.NET Web API 1 with Entity Framework 5

2/20/2020 • 2 minutes to read • Edit Online

How to create a Web API 1 application that uses Entity Framework for database persistence.

- Overview and Creating the Project
- Creating the Domain Models
- Creating an Admin Controller
- Adding an Admin View
- Creating a Dynamic UI with Knockout.js
- Creating Product and Order Controllers
- Creating the Main Page

Part 1: Overview and Creating the Project

2/20/2020 • 2 minutes to read • Edit Online

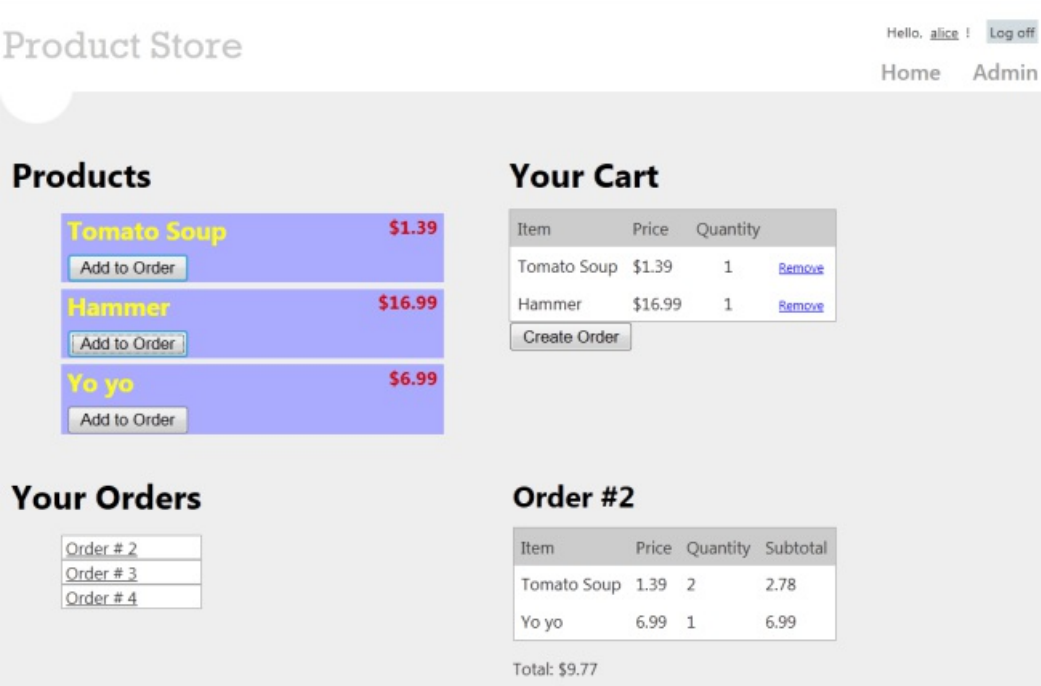
by Mike Wasson

Download Completed Project

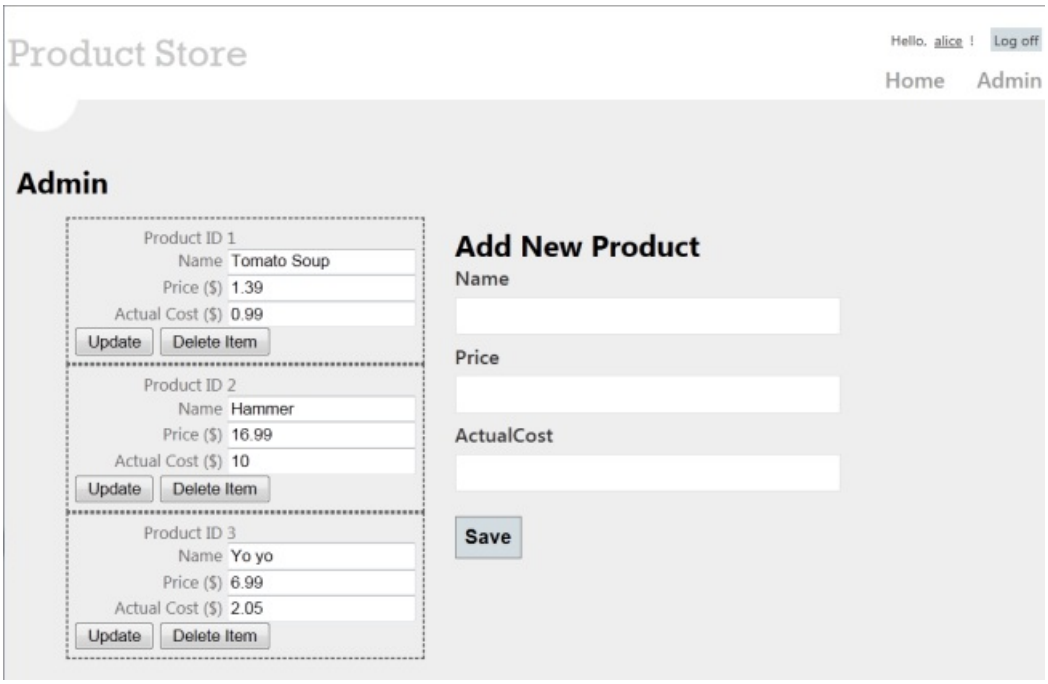
Entity Framework is an object/relational mapping framework. It maps the domain objects in your code to entities in a relational database. For the most part, you do not have to worry about the database layer, because Entity Framework takes care of it for you. Your code manipulates the objects, and changes are persisted to a database.

About the Tutorial

In this tutorial, you will create a simple store application. There are two main parts to the application. Normal users can view products and create orders:



Administrators can create, delete, or edit products:



Skills You'll Learn

Here's what you'll learn:

- How to use Entity Framework with ASP.NET Web API.
- How to use knockout.js to create a dynamic client UI.
- How to use forms authentication with Web API to authenticate users.

Although this tutorial is self-contained, you might want to read the following tutorials first:

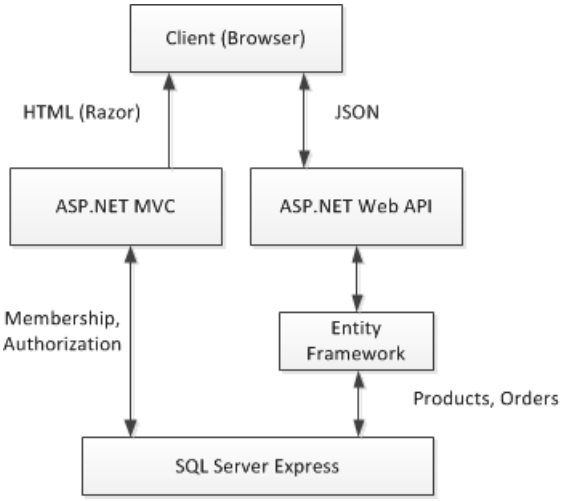
- Your First ASP.NET Web API
- Creating a Web API that Supports CRUD Operations

Some knowledge of ASP.NET MVC is also helpful.

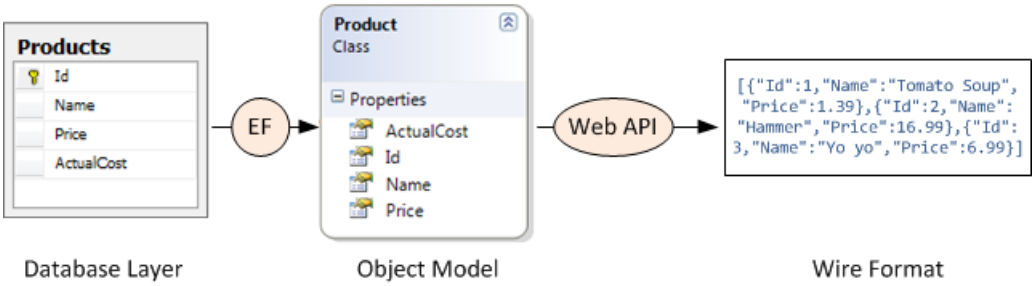
Overview

At a high level, here is the architecture of the application:

- ASP.NET MVC generates the HTML pages for the client.
- ASP.NET Web API exposes CRUD operations on the data (products and orders).
- Entity Framework translates the C# models used by Web API into database entities.



The following diagram shows how the domain objects are represented at various layers of the application: The database layer, the object model, and finally the wire format, which is used to transmit data to the client via HTTP.

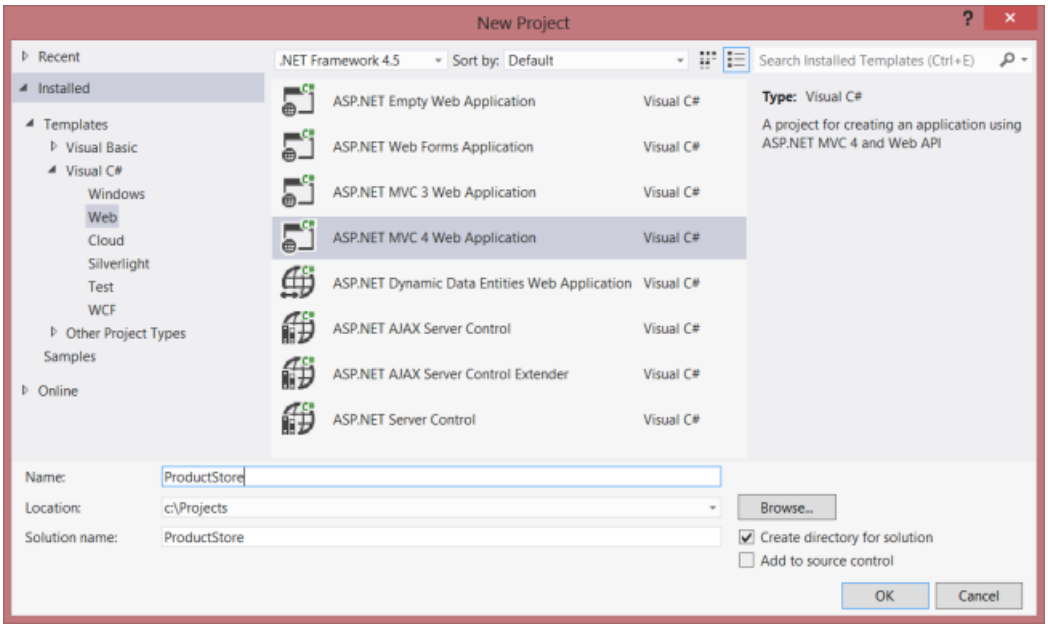


Create the Visual Studio Project

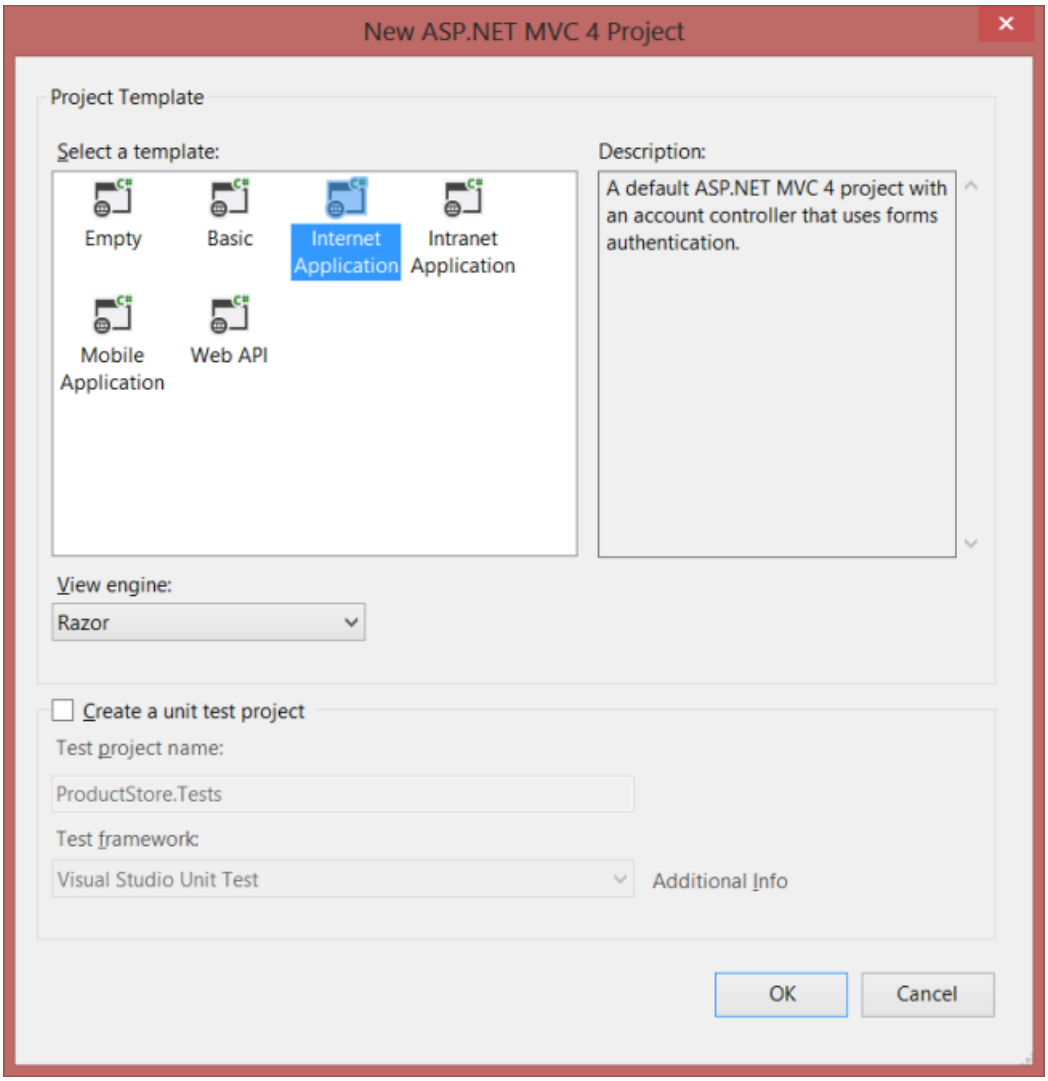
You can create the tutorial project using either Visual Web Developer Express or the full version of Visual Studio.

From the **Start** page, click **New Project**.

In the **Templates** pane, select **Installed Templates** and expand the **Visual C#** node. Under **Visual C#**, select **Web**. In the list of project templates, select **ASP.NET MVC 4 Web Application**. Name the project "ProductStore" and click **OK**.



In the **New ASP.NET MVC 4 Project** dialog, select **Internet Application** and click **OK**.



The "Internet Application" template creates an ASP.NET MVC application that supports forms authentication. If you run the application now, it already has some features:

- New users can register by clicking the "Register" link in the upper right corner.
- Registered users can log in by clicking the "Log in" link.

Membership information is persisted in a database that gets created automatically. For more information about forms authentication in ASP.NET MVC, see Walkthrough: Using Forms Authentication in ASP.NET MVC.

Update the CSS File

This step is cosmetic, but it will make the pages render like the earlier screen shots.

In Solution Explorer, expand the Content folder and open the file named Site.css. Add the following CSS styles:

```
.content {
    clear: both;
    width: 90%;
}

li {
    list-style-type: none;
}

#products li {
    width: 300px;
    background-color: #aaf;
    font-size: 1.5em;
    font-weight: bold;
    color: #ff0;
    margin: 0 0 5px 0;
    padding: 0 5px 0 5px;
}

.price  {
    float: right;
    color: #c00;
    font-size: 0.75em;
}

.details thead td {
    background-color: #CCCCCC;
    color: #333333;
}

.details td {
padding: 6px;
}

.details td.qty {
text-align: center;
}

#cart a {
color: Blue;
font-size: .75em;
}

#update-products li {
    padding: 5px;
    color: #666;
    border-style: dashed;
    border-width: 2px;
    border-color: #666;
}

#update-products li .item {
width: 120px;
display: inline-block;
text-align: right;
}
```

NEXT

Part 2: Creating the Domain Models

2/20/2020 • 3 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

Add Models

There are three ways to approach Entity Framework:

- Database-first: You start with a database, and Entity Framework generates the code.
- Model-first: You start with a visual model, and Entity Framework generates both the database and code.
- Code-first: You start with code, and Entity Framework generates the database.

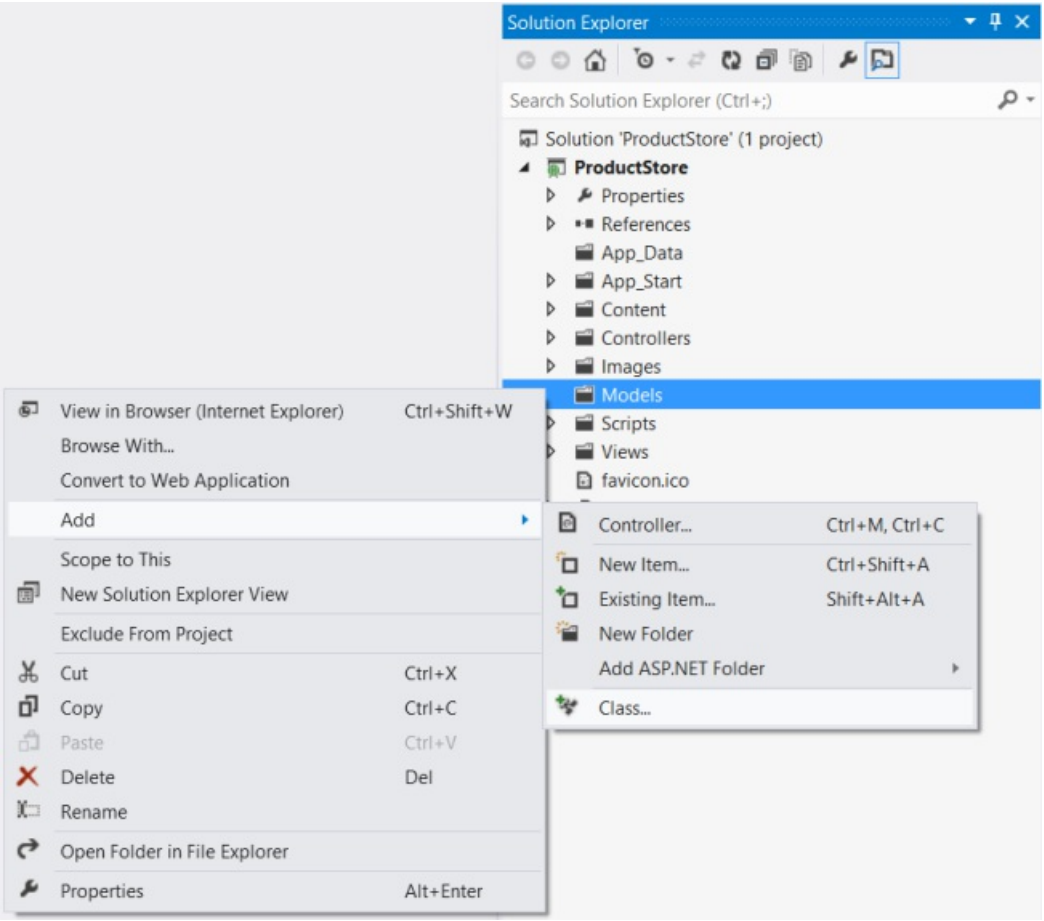
We are using the code-first approach, so we start by defining our domain objects as POCOs (plain-old CLR objects). With the code-first approach, domain objects don't need any extra code to support the database layer, such as transactions or persistence. (Specifically, they do not need to inherit from the EntityObject class.) You can still use data annotations to control how Entity Framework creates the database schema.

Because POCOs do not carry any extra properties that describe database state, they can easily be serialized to JSON or XML. However, that does not mean you should always expose your Entity Framework models directly to clients, as we'll see later in the tutorial.

We will create the following POCOs:

- Product
- Order
- OrderDetail

To create each class, right-click the Models folder in Solution Explorer. From the context menu, select **Add** and then select **Class.**



Add a `Product` class with the following implementation:

```
namespace ProductStore.Models
{
    using System.ComponentModel.DataAnnotations;

    public class Product
    {
        [ScaffoldColumn(false)]
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
        public decimal Price { get; set; }
        public decimal ActualCost { get; set; }
    }
}
```

By convention, Entity Framework uses the `Id` property as the primary key and maps it to an identity column in the database table. When you create a new `Product` instance, you won't set a value for `Id`, because the database generates the value.

The **ScaffoldColumn** attribute tells ASP.NET MVC to skip the `Id` property when generating an editor form. The **Required** attribute is used to validate the model. It specifies that the `Name` property must be a non-empty string.

Add the `Order` class:

```
namespace ProductStore.Models
{
    using System.Collections.Generic;
    using System.ComponentModel.DataAnnotations;

    public class Order
    {
        public int Id { get; set; }
        [Required]
        public string Customer { get; set; }

        // Navigation property
        public  ICollection<OrderDetail> OrderDetails { get; set; }
    }
}
```

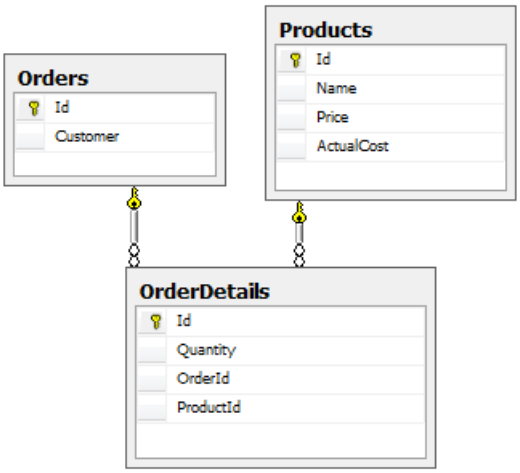
Add the `OrderDetail` class:

```
namespace ProductStore.Models
{
    public class OrderDetail
    {
        public int Id { get; set; }
        public int Quantity { get; set; }
        public int OrderId { get; set; }
        public int ProductId { get; set; }

        // Navigation properties
        public Product Product { get; set; }
        public Order Order { get; set; }
    }
}
```

Foreign Key Relations

An order contains many order details, and each order detail refers to a single product. To represent these relations, the `OrderDetail` class defines properties named `OrderId` and `ProductId`. Entity Framework will infer that these properties represent foreign keys, and will add foreign-key constraints to the database.



The `Order` and `OrderDetail` classes also include "navigation" properties, which contain references to the related objects. Given an order, you can navigate to the products in the order by following the navigation properties.

Compile the project now. Entity Framework uses reflection to discover the properties of the models, so it requires a compiled assembly to create the database schema.

Configure the Media-Type Formatters

A media-type formatter is an object that serializes your data when Web API writes the HTTP response body. The built-in formatters support JSON and XML output. By default, both of these formatters serialize all objects by value.

Serialization by value creates a problem if an object graph contains circular references. That's exactly the case with the `Order` and `OrderDetail` classes, because each holds a reference to the other. The formatter will follow the references, writing each object by value, and go in circles. Therefore, we need to change the default behavior.

In Solution Explorer, expand the App_Start folder and open the file named WebApiConfig.cs. Add the following code to the `WebApiConfig` class:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );

        // New code:
        var json = config.Formatters.JsonFormatter;
        json.SerializerSettings.PreserveReferencesHandling =
            Newtonsoft.Json.PreserveReferencesHandling.Objects;

        config.Formatters.Remove(config.Formatters.XmlFormatter);
    }
}
```

This code sets the JSON formatter to preserve object references, and removes the XML formatter from the pipeline entirely. (You can configure the XML formatter to preserve object references, but it's a little more work, and we only need JSON for this application. For more information, see Handling Circular Object References.)

PREVIOUS NEXT

Part 3: Creating an Admin Controller

2/20/2020 • 4 minutes to read • Edit Online

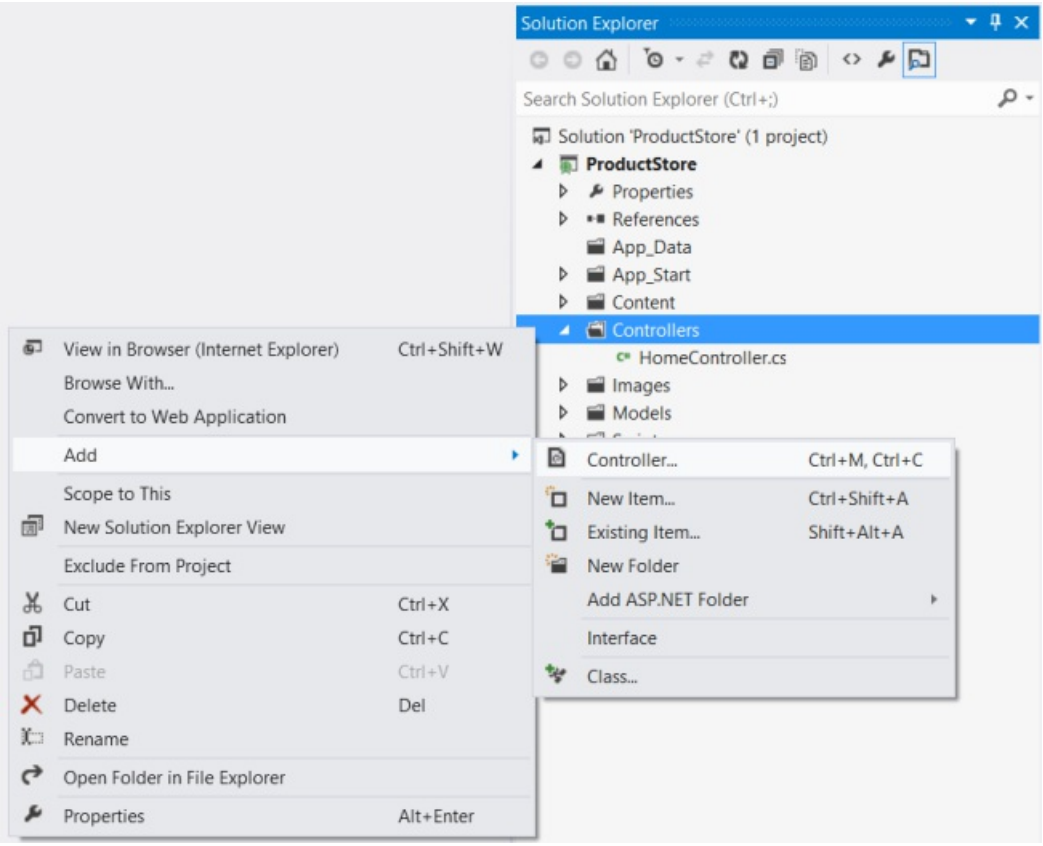
by Mike Wasson

Download Completed Project

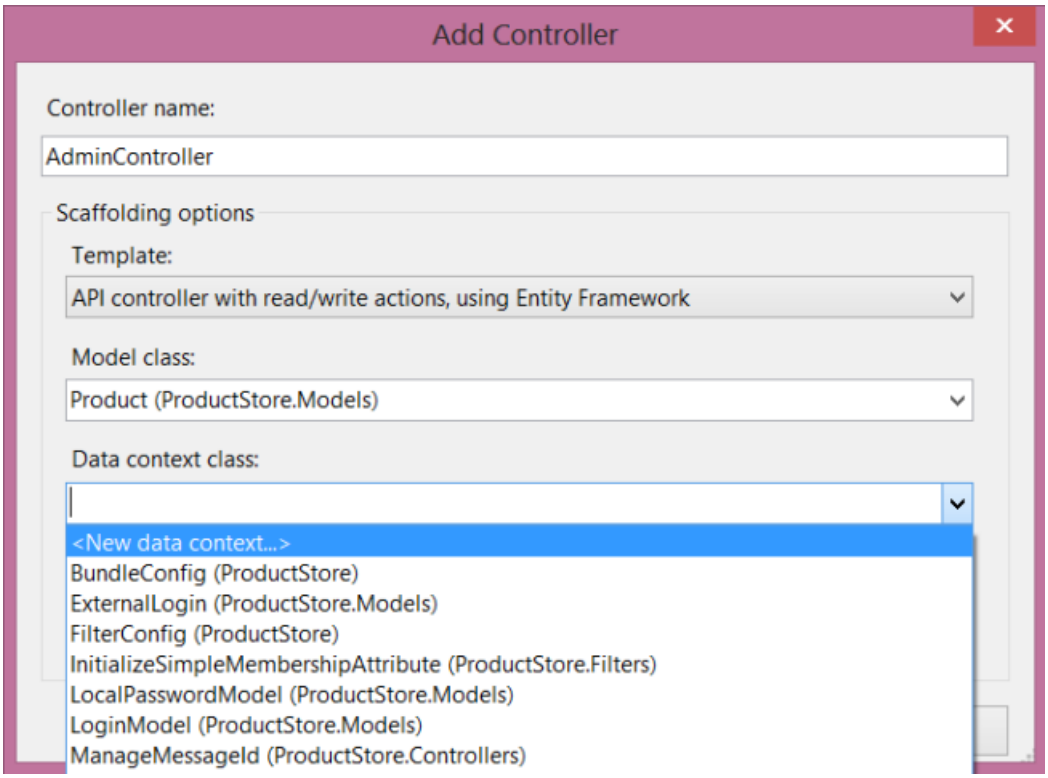
Add an Admin Controller

In this section, we'll add a Web API controller that supports CRUD (create, read, update, and delete) operations on products. The controller will use Entity Framework to communicate with the database layer. Only administrators will be able to use this controller. Customers will access the products through another controller.

In Solution Explorer, right-click the Controllers folder. Select **Add** and then **Controller**.

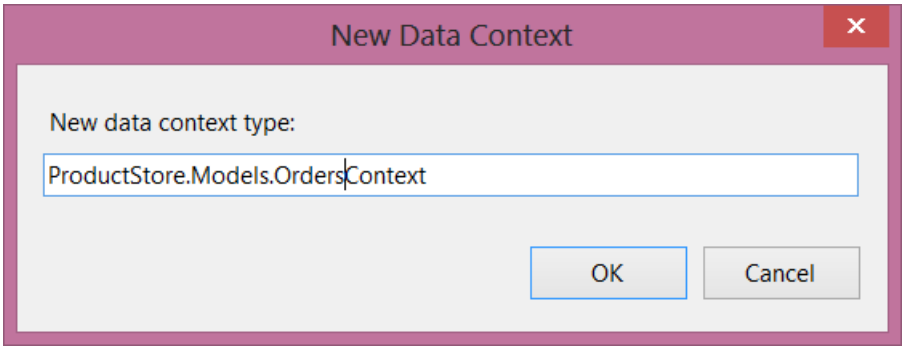


In the **Add Controller** dialog, name the controller `AdminController`. Under **Template**, select "API controller with read/write actions, using Entity Framework". Under **Model class**, select "Product (ProductStore.Models)". Under **Data Context**, select "<New Data Context>".



> **NOTE**
>
> If the **Model class** drop-down does not show any model classes, make sure you compiled the project. Entity Framework uses reflection, so it needs the compiled assembly.

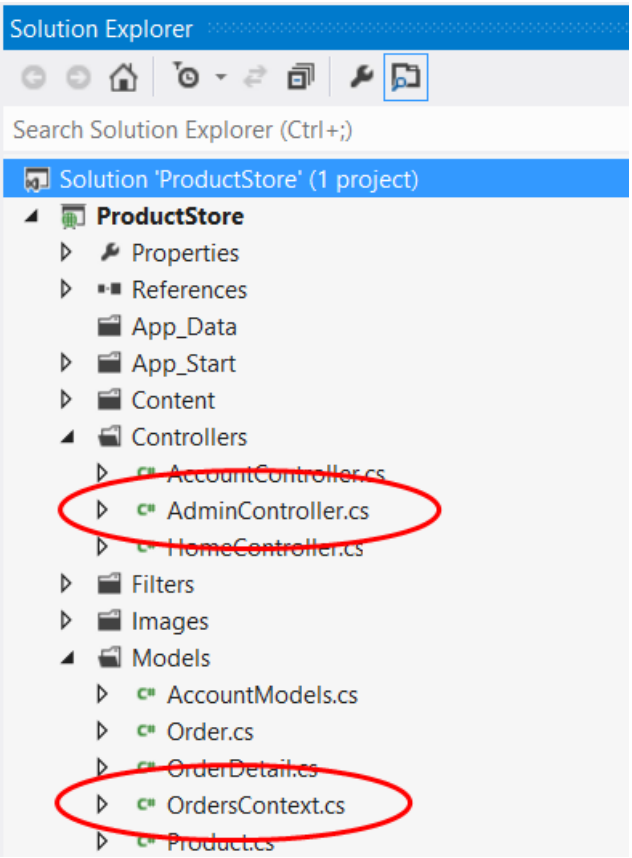
Selecting "<New Data Context>" will open the **New Data Context** dialog. Name the data context `ProductStore.Models.OrdersContext`.



Click **OK** to dismiss the **New Data Context** dialog. In the **Add Controller** dialog, click **Add**.

Here's what got added to the project:

- A class named `OrdersContext` that derives from **DbContext**. This class provides the glue between the POCO models and the database.
- A Web API controller named `AdminController`. This controller supports CRUD operations on `Product` instances. It uses the `OrdersContext` class to communicate with Entity Framework.
- A new database connection string in the Web.config file.



Open the OrdersContext.cs file. Notice that the constructor specifies the name of the database connection string. This name refers to the connection string that was added to Web.config.

```
public OrdersContext() : base("name=OrdersContext")
```

Add the following properties to the `OrdersContext` class:

```
public DbSet<Order> Orders { get; set; }
public DbSet<OrderDetail> OrderDetails { get; set; }
```

A **DbSet** represents a set of entities that can be queried. Here is the complete listing for the `OrdersContext` class:

```
public class OrdersContext : DbContext
{
    public OrdersContext() : base("name=OrdersContext")
    {
    }

    public DbSet<Order> Orders { get; set; }
    public DbSet<OrderDetail> OrderDetails { get; set; }
    public DbSet<Product> Products { get; set; }
}
```

The `AdminController` class defines five methods that implement basic CRUD functionality. Each method corresponds to a URI that the client can invoke:

CONTROLLER METHOD	DESCRIPTION	URI	HTTP METHOD
GetProducts	Gets all products.	api/products	GET
GetProduct	Finds a product by ID.	api/products/*id*	GET

CONTROLLER METHOD	DESCRIPTION	URI	HTTP METHOD
PutProduct	Updates a product.	api/products/*id*	PUT
PostProduct	Creates a new product.	api/products	POST
DeleteProduct	Deletes a product.	api/products/*id*	DELETE

Each method calls into `OrdersContext` to query the database. The methods that modify the collection (PUT, POST, and DELETE) call `db.SaveChanges` to persist the changes to the database. Controllers are created per HTTP request and then disposed, so it is necessary to persist changes before a method returns.

Add a Database Initializer

Entity Framework has a nice feature that lets you populate the database on startup, and automatically recreate the database whenever the models change. This feature is useful during development, because you always have some test data, even if you change the models.

In Solution Explorer, right-click the Models folder and create a new class named `OrdersContextInitializer`. Paste in the following implementation:

```
namespace ProductStore.Models
{
    using System;
    using System.Collections.Generic;
    using System.Data.Entity;

    public class OrdersContextInitializer : DropCreateDatabaseIfModelChanges<OrdersContext>
    {
        protected override void Seed(OrdersContext context)
        {
            var products = new List<Product>()
            {
                new Product() { Name = "Tomato Soup", Price = 1.39M, ActualCost = .99M },
                new Product() { Name = "Hammer", Price = 16.99M, ActualCost = 10 },
                new Product() { Name = "Yo yo", Price = 6.99M, ActualCost = 2.05M }
            };

            products.ForEach(p => context.Products.Add(p));
            context.SaveChanges();

            var order = new Order() { Customer = "Bob" };
            var od = new List<OrderDetail>()
            {
                new OrderDetail() { Product = products[0], Quantity = 2, Order = order},
                new OrderDetail() { Product = products[1], Quantity = 4, Order = order }
            };
            context.Orders.Add(order);
            od.ForEach(o => context.OrderDetails.Add(o));

            context.SaveChanges();
        }
    }
}
```

By inheriting from the **DropCreateDatabaseIfModelChanges** class, we are telling Entity Framework to drop the database whenever we modify the model classes. When Entity Framework creates (or recreates) the database, it calls the **Seed** method to populate the tables. We use the **Seed** method to add some example products plus an

example order.

This feature is great for testing, but don't use the **DropCreateDatabaseIfModelChanges** class in production,, because you could lose your data if someone changes a model class.

Next, open Global.asax and add the following code to the **Application_Start** method:

```
System.Data.Entity.Database.SetInitializer(
    new ProductStore.Models.OrdersContextInitializer());
```

Send a Request to the Controller

At this point, we haven't written any client code, but you can invoke the web API using a web browser or an HTTP debugging tool such as Fiddler. In Visual Studio, press F5 to start debugging. Your web browser will open to `http://localhost:*portnum*/`, where *portnum* is some port number.

Send an HTTP request to "`http://localhost:*portnum*/api/admin`. The first request may be slow to complete, because Entity Framework needs to create and seed the database. The response should something similar to the following:

```
HTTP/1.1 200 OK
Server: ASP.NET Development Server/10.0.0.0
Date: Mon, 18 Jun 2012 04:30:33 GMT
X-AspNet-Version: 4.0.30319
Cache-Control: no-cache
Pragma: no-cache
Expires: -1
Content-Type: application/json; charset=utf-8
Content-Length: 175
Connection: Close

[{"Id":1,"Name":"Tomato Soup","Price":1.39,"ActualCost":0.99},{"Id":2,"Name":"Hammer",
"Price":16.99,"ActualCost":10.00},{"Id":3,"Name":"Yo yo","Price":6.99,"ActualCost":
2.05}]
```

PREVIOUS NEXT

Part 4: Adding an Admin View

2/20/2020 • 3 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

Add an Admin View

Now we'll turn to the client side, and add a page that can consume data from the Admin controller. The page will allow users to create, edit, or delete products, by sending AJAX requests to the controller.

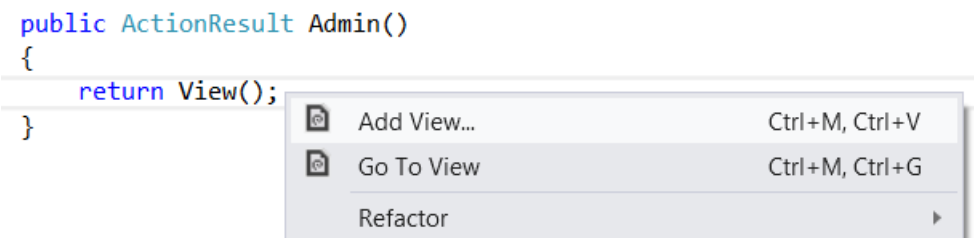
In Solution Explorer, expand the Controllers folder and open the file named HomeController.cs. This file contains an MVC controller. Add a method named `Admin`:

```
public ActionResult Admin()
{
    string apiUri= Url.HttpRouteUrl("DefaultApi", new { controller = "admin", });
    ViewBag.ApiUrl = new Uri(Request.Url, apiUri).AbsoluteUri.ToString();

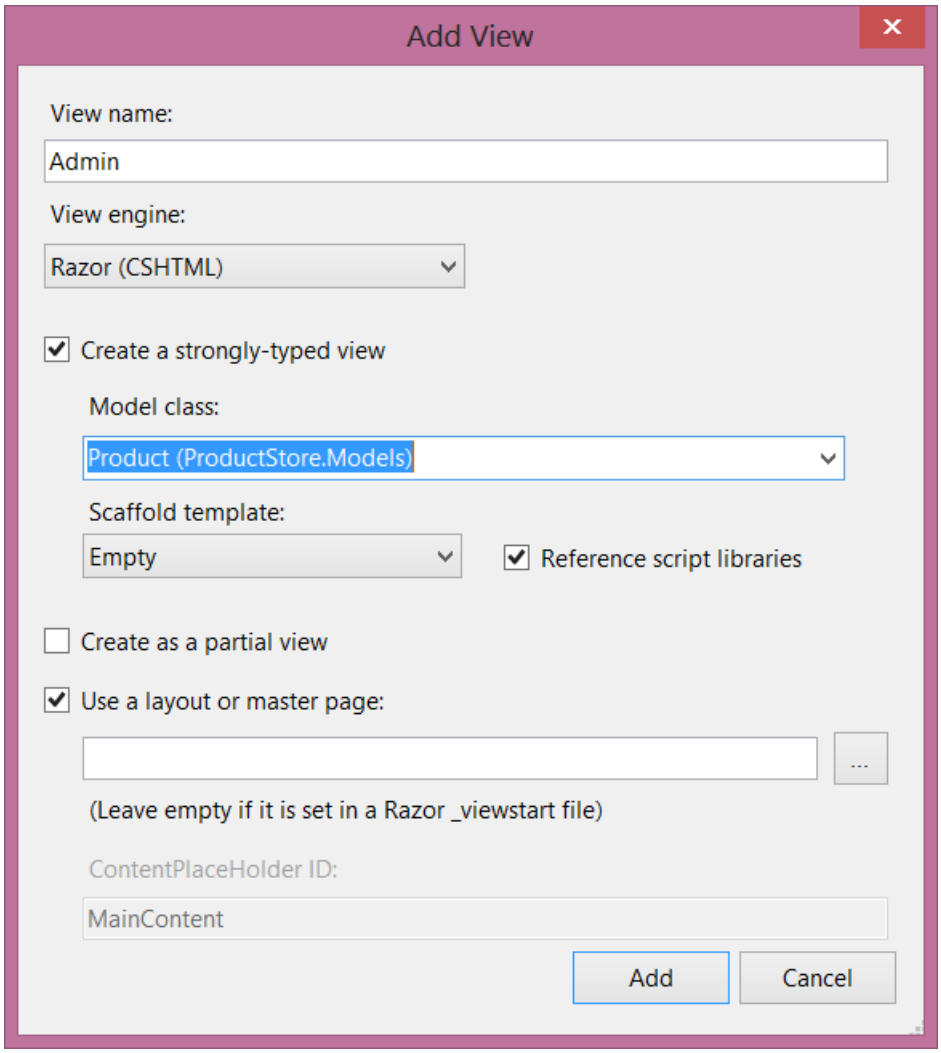
    return View();
}
```

The **HttpRouteUrl** method creates the URI to the web API, and we store this in the view bag for later.

Next, position the text cursor within the `Admin` action method, then right-click and select **Add View**. This will bring up the **Add View** dialog.



In the **Add View** dialog, name the view "Admin". Select the check box labeled **Create a strongly-typed view**. Under **Model Class**, select "Product (ProductStore.Models)". Leave all the other options as their default values.



Clicking **Add** adds a file named Admin.cshtml under Views/Home. Open this file and add the following HTML. This HTML defines the structure of the page, but no functionality is wired up yet.

```
<div class="content">
    <div class="float-left">
        <ul id="update-products">
            <li>
                <div><div class="item">Product ID</div><span></span></div>
                <div><div class="item">Name</div> <input type="text" /></div>
                <div><div class="item">Price ($)</div> <input type="text" /></div>
                <div><div class="item">Actual Cost ($)</div> <input type="text" /></div>
                <div>
                    <input type="button" value="Update" />
                    <input type="button" value="Delete Item" />
                </div>
         </li>
        </ul>
    </div>

    <div class="float-right">
    <h2>Add New Product</h2>
    <form id="product">
        @Html.ValidationSummary(true)
        <fieldset>
            <legend>Contact</legend>
            @Html.EditorForModel()
            <p>
                <input type="submit" value="Save" />
            </p>
        </fieldset>
    </form>
    </div>
</div>
```

Create a Link to the Admin Page

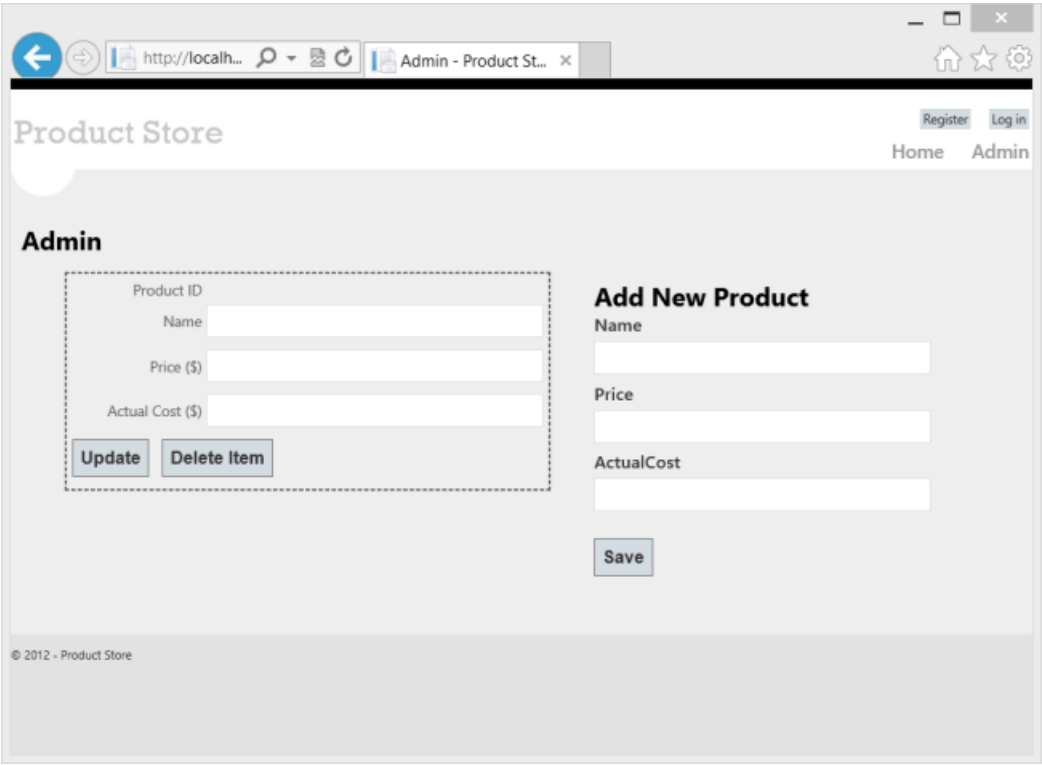
In Solution Explorer, expand the Views folder and then expand the Shared folder. Open the file named _Layout.cshtml. Locate the **ul** element with id = "menu", and an action link for the Admin view:

```
<li>@Html.ActionLink("Admin", "Admin", "Home")</li>
```

NOTE

In the sample project, I made a few other cosmetic changes, such as replacing the string "Your logo here". These don't affect the functionality of the application. You can download the project and compare the files.

Run the application and click the "Admin" link that appears at the top of the home page. The Admin page should look like the following:



Right now, the page doesn't do anything. In the next section, we'll use Knockout.js to create a dynamic UI.

Add Authorization

The Admin page is currently accessible to anyone visiting the site. Let's change this to restrict permission to administrators.

Start by adding an "Administrator" role and an administrator user. In Solution Explorer, expand the Filters folder and open the file named InitializeSimpleMembershipAttribute.cs. Locate the `SimpleMembershipInitializer` constructor. After the call to **WebSecurity.InitializeDatabaseConnection**, add the following code:

```
const string adminRole = "Administrator";
const string adminName = "Administrator";

if (!Roles.RoleExists(adminRole))
{
    Roles.CreateRole(adminRole);
}
if (!WebSecurity.UserExists(adminName))
{
    WebSecurity.CreateUserAndAccount(adminName, "password");
    Roles.AddUserToRole(adminName, adminRole);
}
```

This is a quick-and-dirty way to add the "Administrator" role and create a user for the role.

In Solution Explorer, expand the Controllers folder and open the HomeController.cs file. Add the **Authorize** attribute to the `Admin` method.

```
[Authorize(Roles="Administrator")]
public ActionResult Admin()
{
    return View();
}
```

Open the AdminController.cs file and add the **Authorize** attribute to the entire `AdminController` class.

```
[Authorize(Roles="Administrator")]
public class AdminController : ApiController
{
    // ...
```

NOTE

MVC and Web API both define **Authorize** attributes, in different namespaces. MVC uses **System.Web.Mvc.AuthorizeAttribute**, while Web API uses **System.Web.Http.AuthorizeAttribute**.

Now only administrators can view the Admin page. Also, if you send an HTTP request to the Admin controller, the request must contain an authentication cookie. If not, the server sends an HTTP 401 (Unauthorized) response. You can see this in Fiddler by sending a GET request to `http://localhost:*port*/api/admin`.

PREVIOUS NEXT

Part 5: Creating a Dynamic UI with Knockout.js

2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

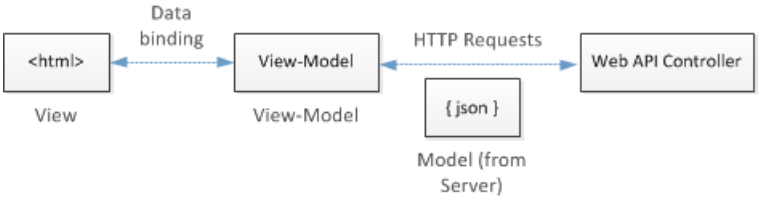
Creating a Dynamic UI with Knockout.js

In this section, we'll use Knockout.js to add functionality to the Admin view.

Knockout.js is a Javascript library that makes it easy to bind HTML controls to data. Knockout.js uses the Model-View-ViewModel (MVVM) pattern.

- The *model* is the server-side representation of the data in the business domain (in our case, products and orders).
- The *view* is the presentation layer (HTML).
- The *view-model* is a Javascript object that holds the model data. The view-model is a code abstraction of the UI. It has no knowledge of the HTML representation. Instead, it represents abstract features of the view, such as "a list of items".

The view is data-bound to the view-model. Updates to the view-model are automatically reflected in the view. The view-model also gets events from the view, such as button clicks, and performs operations on the model, such as creating an order.



First we'll define the view-model. After that, we will bind the HTML markup to the view-model.

Add the following Razor section to Admin.cshtml:

```
@section Scripts {
  @Scripts.Render("~/bundles/jqueryval")
  <script type="text/javascript" src="@Url.Content("~/Scripts/knockout-2.1.0.js")"></script>
  <script type="text/javascript">
  // View-model will go here
  </script>
}
```

You can add this section anywhere in the file. When the view is rendered, the section appears at the bottom of the HTML page, right before the closing </body> tag.

All of the script for this page will go inside the script tag indicated by the comment:

```
<script type="text/javascript">
  // View-model will go here
  </script>
```

First, define a view-model class:

```
function ProductsViewModel() {
    var self = this;
    self.products = ko.observableArray();
}
```

ko.observableArray is a special kind of object in Knockout, called an *observable*. From the Knockout.js documentation: An observable is a "JavaScript object that can notify subscribers about changes." When the contents of an observable change, the view is automatically updated to match.

To populate the `products` array, make an AJAX request to the web API. Recall that we stored the base URI for the API in the view bag (see Part 4 of the tutorial).

```
function ProductsViewModel() {
    var self = this;
    self.products = ko.observableArray();

    // New code
    var baseUri = '@ViewBag.ApiUrl';
    $.getJSON(baseUri, self.products);
}
```

Next, add functions to the view-model to create, update, and delete products. These functions submit AJAX calls to the web API and use the results to update the view-model.

```
function ProductsViewModel() {
    var self = this;
    self.products = ko.observableArray();

    var baseUri = '@ViewBag.ApiUrl';

    // New code
    self.create = function (formElement) {
        // If the form data is valid, post the serialized form data to the web API.
        $(formElement).validate();
        if ($(formElement).valid()) {
            $.post(baseUri, $(formElement).serialize(), null, "json")
                .done(function (o) {
                    // Add the new product to the view-model.
                    self.products.push(o);
                });
        }
    }

    self.update = function (product) {
        $.ajax({ type: "PUT", url: baseUri + '/' + product.Id, data: product });
    }

    self.remove = function (product) {
        // First remove from the server, then from the view-model.
        $.ajax({ type: "DELETE", url: baseUri + '/' + product.Id })
            .done(function () { self.products.remove(product); });
    }

    $.getJSON(baseUri, self.products);
}
```

Now the most important part: When the DOM is fulled loaded, call the **ko.applyBindings** function and pass in a new instance of the `ProductsViewModel`:

```
$(document).ready(function () {
    ko.applyBindings(new ProductsViewModel());
})
```

The **ko.applyBindings** method activates Knockout and wires up the view-model to the view.

Now that we have a view-model, we can create the bindings. In Knockout.js, you do this by adding `data-bind` attributes to HTML elements. For example, to bind an HTML list to an array, use the `foreach` binding:

```
<ul id="update-products" data-bind="foreach: products">
```

The `foreach` binding iterates through the array and creates child elements for each object in the array. Bindings on the child elements can refer to properties on the array objects.

Add the following bindings to the "update-products" list:

```
<ul id="update-products" data-bind="foreach: products">
    <li>
        <div>
            <div class="item">Product ID</div> <span data-bind="text: $data.Id"></span>
        </div>
        <div>
            <div class="item">Name</div>
            <input type="text" data-bind="value: $data.Name"/>
        </div>
        <div>
            <div class="item">Price ($)</div>
            <input type="text" data-bind="value: $data.Price"/>
        </div>
        <div>
            <div class="item">Actual Cost ($)</div>
            <input type="text" data-bind="value: $data.ActualCost"/>
        </div>
        <div>
            <input type="button" value="Update" data-bind="click: $root.update"/>
            <input type="button" value="Delete Item" data-bind="click: $root.remove"/>
        </div>
    </li>
</ul>
```

The `<li>` element occurs within the scope of the **foreach** binding. That means Knockout will render the element once for each product in the `products` array. All of the bindings within the `<li>` element refer to that product instance. For example, `$data.Name` refers to the `Name` property on the product.

To set the values of the text inputs, use the `value` binding. The buttons are bound to functions on the model-view, using the `click` binding. The product instance is passed as a parameter to each function. For more information, the Knockout.js documentation has good descriptions of the various bindings.

Next, add a binding for the **submit** event on the Add Product form:

```
<form id="addProduct" data-bind="submit: create">
```

This binding calls the `create` function on the view-model to create a new product.

Here is the complete code for the Admin view:

```
@model ProductStore.Models.Product
```

```
@{
    ViewBag.Title = "Admin";
}

@section Scripts {
  @Scripts.Render("~/bundles/jqueryval")
  <script type="text/javascript" src="@Url.Content("~/Scripts/knockout-2.0.0.js")"></script>
  <script type="text/javascript">
      function ProductsViewModel() {
          var self = this;
          self.products = ko.observableArray();

          var baseUri = '@ViewBag.ApiUrl';

          self.create = function (formElement) {
              // If valid, post the serialized form data to the web api
              $(formElement).validate();
              if ($(formElement).valid()) {
                  $.post(baseUri, $(formElement).serialize(), null, "json")
                      .done(function (o) { self.products.push(o); });
              }
          }

          self.update = function (product) {
              $.ajax({ type: "PUT", url: baseUri + '/' + product.Id, data: product });
          }

          self.remove = function (product) {
              // First remove from the server, then from the UI
              $.ajax({ type: "DELETE", url: baseUri + '/' + product.Id })
                  .done(function () { self.products.remove(product); });
          }

          $.getJSON(baseUri, self.products);
      }

      $(document).ready(function () {
          ko.applyBindings(new ProductsViewModel());
      })
  </script>
}

<h2>Admin</h2>
<div class="content">
    <div class="float-left">
    <ul id="update-products" data-bind="foreach: products">
        <li>
            <div>
                <div class="item">Product ID</div> <span data-bind="text: $data.Id"></span>
            </div>
            <div>
                <div class="item">Name</div>
                <input type="text" data-bind="value: $data.Name"/>
            </div>
            <div>
                <div class="item">Price ($)</div>
                <input type="text" data-bind="value: $data.Price"/>
            </div>
            <div>
                <div class="item">Actual Cost ($)</div>
                <input type="text" data-bind="value: $data.ActualCost"/>
            </div>
            <div>
                <input type="button" value="Update" data-bind="click: $root.update"/>
                <input type="button" value="Delete Item" data-bind="click: $root.remove"/>
            </div>
        </li>
    </ul>
    </div>
```

```
    <div class="float-right">
    <h2>Add New Product</h2>
    <form id="addProduct" data-bind="submit: create">
        @Html.ValidationSummary(true)
        <fieldset>
            <legend>Contact</legend>
            @Html.EditorForModel()
            <p>
                <input type="submit" value="Save" />
            </p>
        </fieldset>
    </form>
    </div>
</div>
```

Run the application, log in with the Administrator account, and click the "Admin" link. You should see the list of products, and be able to create, update, or delete products.

PREVIOUS NEXT

Part 6: Creating Product and Order Controllers

2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

Add a Products Controller

The Admin controller is for users who have administrator privileges. Customers, on the other hand, can view products but cannot create, update, or delete them.

We can easily restrict access to the Post, Put, and Delete methods, while leaving the Get methods open. But look at the data that is returned for a product:

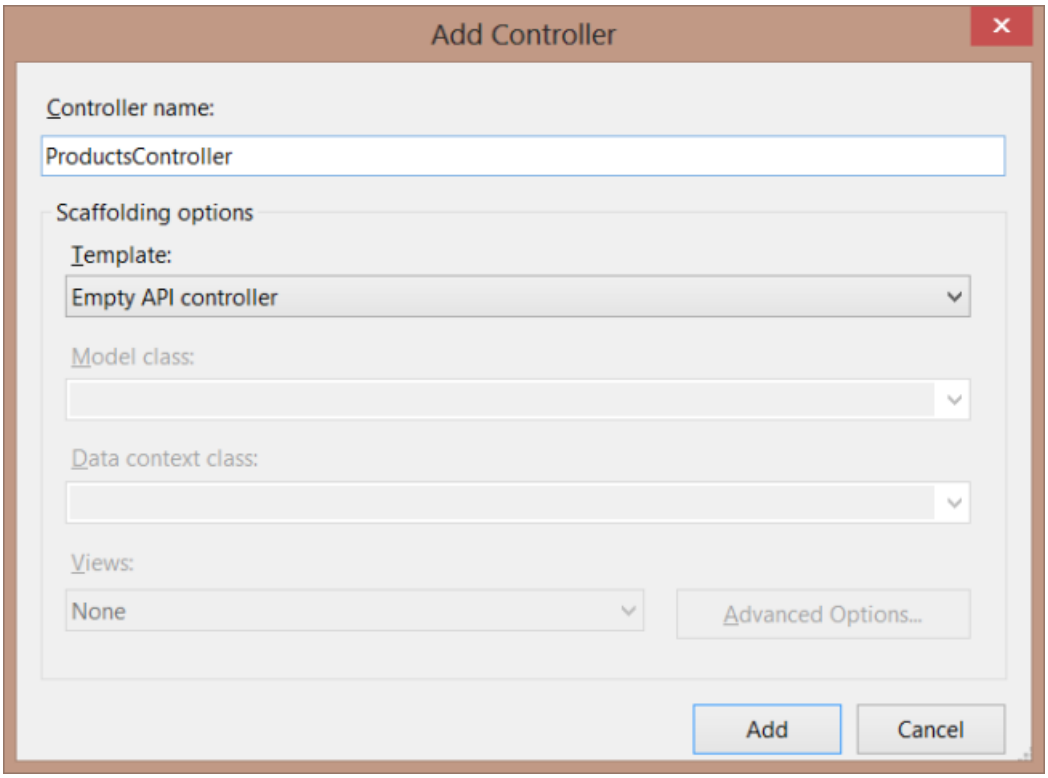
```
{"Id":1,"Name":"Tomato Soup","Price":1.39,"ActualCost":0.99}
```

The `ActualCost` property should not be visible to customers! The solution is to define a *data transfer object* (DTO) that includes a subset of properties that should be visible to customers. We will use LINQ to project `Product` instances to `ProductDTO` instances.

Add a class named `ProductDTO` to the Models folder.

```
namespace ProductStore.Models
{
    public class ProductDTO
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

Now add the controller. In Solution Explorer, right-click the Controllers folder. Select **Add**, then select **Controller**. In the **Add Controller** dialog, name the controller "ProductsController". Under **Template**, select **Empty API controller**.



Replace everything in the source file with the following code:

```
namespace ProductStore.Controllers
{
    using System.Collections.Generic;
    using System.Linq;
    using System.Net;
    using System.Net.Http;
    using System.Web.Http;
    using ProductStore.Models;

    public class ProductsController : ApiController
    {
        private OrdersContext db = new OrdersContext();

        // Project products to product DTOs.
        private IQueryable<ProductDTO> MapProducts()
        {
            return from p in db.Products select new ProductDTO()
                { Id = p.Id, Name = p.Name, Price = p.Price };
        }

        public IEnumerable<ProductDTO> GetProducts()
        {
            return MapProducts().AsEnumerable();
        }

        public ProductDTO GetProduct(int id)
        {
            var product = (from p in MapProducts()
                           where p.Id == 1
                           select p).FirstOrDefault();
            if (product == null)
            {
                throw new HttpResponseException(
                    Request.CreateResponse(HttpStatusCode.NotFound));
            }
            return product;
        }

        protected override void Dispose(bool disposing)
        {
            db.Dispose();
            base.Dispose(disposing);
        }
    }
}
```

The controller still uses the `OrdersContext` to query the database. But instead of returning `Product` instances directly, we call `MapProducts` to project them onto `ProductDTO` instances:

```
return from p in db.Products select new ProductDTO()
    { Id = p.Id, Name = p.Name, Price = p.Price };
```

The `MapProducts` method returns an **IQueryable**, so we can compose the result with other query parameters. You can see this in the `GetProduct` method, which adds a **where** clause to the query:

```
var product = (from p in MapProducts()
    where p.Id == 1
    select p).FirstOrDefault();
```

Add an Orders Controller

Next, add a controller that lets users create and view orders.

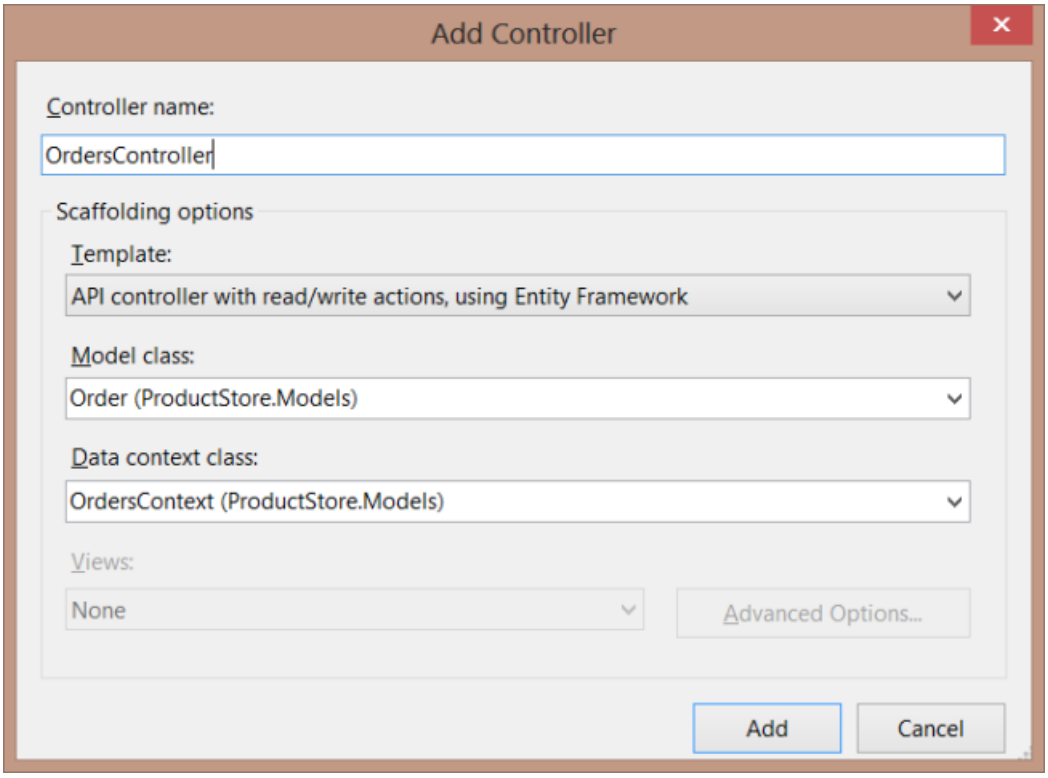
We'll start with another DTO. In Solution Explorer, right-click the Models folder and add a class named `OrderDTO`. Use the following implementation:

```
namespace ProductStore.Models
{
    using System.Collections.Generic;

    public class OrderDTO
    {
        public class Detail
        {
            public int ProductID { get; set; }
            public string Product { get; set; }
            public decimal Price { get; set; }
            public int Quantity { get; set; }
        }
        public IEnumerable<Detail> Details { get; set; }
    }
}
```

Now add the controller. In Solution Explorer, right-click the Controllers folder. Select **Add**, then select **Controller**. In the **Add Controller** dialog, set the following options:

- Under **Controller Name**, enter "OrdersController".
- Under **Template**, select "API controller with read/write actions, using Entity Framework".
- Under **Model class**, select "Order (ProductStore.Models)".
- Under **Data context class**, select "OrdersContext (ProductStore.Models)".



Click **Add**. This adds a file named OrdersController.cs. Next, we need to modify the default implementation of the controller.

First, delete the `PutOrder` and `DeleteOrder` methods. For this sample, customers cannot modify or delete existing orders. In a real application, you would need lots of back-end logic to handle these cases. (For example, was the order already shipped?)

Change the `GetOrders` method to return just the orders that belong to the user:

```
public IEnumerable<Order> GetOrders()
{
    return db.Orders.Where(o => o.Customer == User.Identity.Name);
}
```

Change the `GetOrder` method as follows:

```
public OrderDTO GetOrder(int id)
{
    Order order = db.Orders.Include("OrderDetails.Product")
        .First(o => o.Id == id && o.Customer == User.Identity.Name);
    if (order == null)
    {
        throw new HttpResponseException(Request.CreateResponse(HttpStatusCode.NotFound));
    }

    return new OrderDTO()
    {
        Details = from d in order.OrderDetails
                  select new OrderDTO.Detail()
                      {
                          ProductID = d.Product.Id,
                          Product = d.Product.Name,
                          Price = d.Product.Price,
                          Quantity = d.Quantity
                      }
    };
}
```

Here are the changes that we made to the method:

- The return value is an `OrderDTO` instance, instead of an `Order`.
- When we query the database for the order, we use the DbQuery.Include method to fetch the related `OrderDetail` and `Product` entities.
- We flatten the result by using a projection.

The HTTP response will contain an array of products with quantities:

```
{"Details":[{"ProductID":1,"Product":"Tomato Soup","Price":1.39,"Quantity":2},
{"ProductID":3,"Product":"Yo yo","Price":6.99,"Quantity":1}]}
```

This format is easier for clients to consume than the original object graph, which contains nested entities (order, details, and products).

The last method to consider it `PostOrder`. Right now, this method takes an `Order` instance. But consider what happens if a client sends a request body like this:

```
{"Customer":"Alice","OrderDetails":[{"Quantity":1,"Product":{"Name":"Koala bears",
"Price":5,"ActualCost":1}}]}
```

This is a well-structured order, and Entity Framework will happily insert it into the database. But it contains a Product entity that did not exist previously. The client just created a new product in our database! This will be a surprise to the order fulfillment department, when they see an order for koala bears. The moral is, be really careful about the data you accept in a POST or PUT request.

To avoid this problem, change the `PostOrder` method to take an `OrderDTO` instance. Use the `OrderDTO` to create the `Order`.

```
var order = new Order()
{
    Customer = User.Identity.Name,
    OrderDetails = (from item in dto.Details select new OrderDetail()
        { ProductId = item.ProductID, Quantity = item.Quantity }).ToList()
};
```

Notice that we use the `ProductID` and `Quantity` properties, and we ignore any values that the client sent for either product name or price. If the product ID is not valid, it will violate the foreign key constraint in the database, and the insert will fail, as it should.

Here is the complete `PostOrder` method:

```
public HttpResponseMessage PostOrder(OrderDTO dto)
{
    if (ModelState.IsValid)
    {
        var order = new Order()
        {
            Customer = User.Identity.Name,
            OrderDetails = (from item in dto.Details select new OrderDetail()
                { ProductId = item.ProductID, Quantity = item.Quantity }).ToList()
        };

        db.Orders.Add(order);
        db.SaveChanges();

        HttpResponseMessage response = Request.CreateResponse(HttpStatusCode.Created, order);
        response.Headers.Location = new Uri(Url.Link("DefaultApi", new { id = order.Id }));
        return response;
    }
    else
    {
        return Request.CreateResponse(HttpStatusCode.BadRequest);
    }
}
```

Finally, add the **Authorize** attribute to the controller:

```
[Authorize]
public class OrdersController : ApiController
{
    // ...
```

Now only registered users can create or view orders.

PREVIOUS NEXT

Part 7: Creating the Main Page

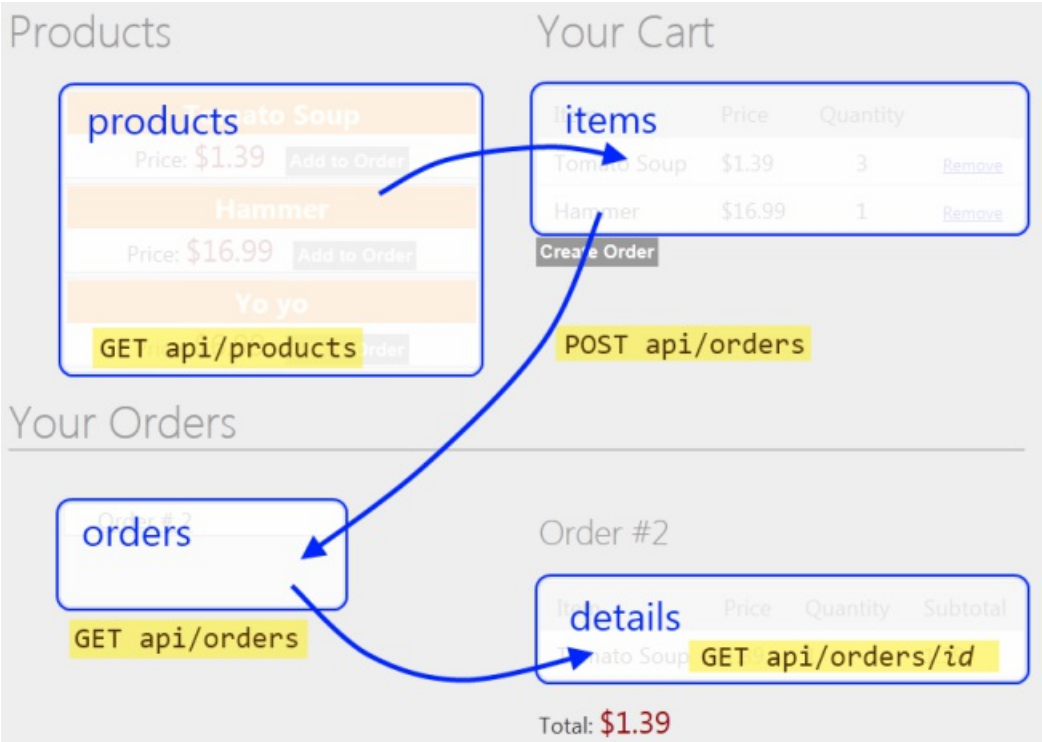
2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson

Download Completed Project

Creating the Main Page

In this section, you will create the main application page. This page will be more complex than the Admin page, so we'll approach it in several steps. Along the way, you'll see some more advanced Knockout.js techniques. Here is the basic layout of the page:



- "Products" holds an array of products.
- "Cart" holds an array of products with quantities. Clicking "Add to Cart" updates the cart.
- "Orders" holds an array of order IDs.
- "Details" holds an order detail, which is an array of items (products with quantities)

We'll start by defining some basic layout in HTML, with no data binding or script. Open the file Views/Home/Index.cshtml and replace all of the contents with the following:

```
<div class="content">
    <!-- List of products -->
    <div class="float-left">
    <h1>Products</h1>
    <ul id="products">
    </ul>
    </div>

    <!-- Cart -->
    <div id="cart" class="float-right">
    <h1>Your Cart</h1>
        <table class="details ui-widget-content">
    </table>
    <input type="button" value="Create Order"/>
    </div>
</div>

<div id="orders-area" class="content" >
    <!-- List of orders -->
    <div class="float-left">
    <h1>Your Orders</h1>
    <ul id="orders">
    </ul>
    </div>

   <!-- Order Details -->
    <div id="order-details" class="float-right">
    <h2>Order #<span></span></h2>
    <table class="details ui-widget-content">
    </table>
    <p>Total: <span></span></p>
    </div>
</div>
```

Next, add a Scripts section and create an empty view-model:

```
@section Scripts {
  <script type="text/javascript" src="@Url.Content("~/Scripts/knockout-2.1.0.js")"></script>
  <script type="text/javascript">

    function AppViewModel() {
        var self = this;
        self.loggedIn = @(Request.IsAuthenticated ? "true" : "false");
    }

    $(document).ready(function () {
        ko.applyBindings(new AppViewModel());
    });

  </script>
}
```

Based on the design sketched earlier, our view model needs observables for products, cart, orders, and details. Add the following variables to the `AppViewModel` object:

```
self.products = ko.observableArray();
self.cart = ko.observableArray();
self.orders = ko.observableArray();
self.details = ko.observable();
```

Users can add items from the products list into the cart, and remove items from the cart. To encapsulate these functions, we'll create another view-model class that represents a product. Add the following code to

`AppViewModel`:

```
function AppViewModel() {
    // ...

    // NEW CODE
    function ProductViewModel(root, product) {
        var self = this;
        self.ProductId = product.Id;
        self.Name = product.Name;
        self.Price = product.Price;
        self.Quantity = ko.observable(0);

        self.addItemToCart = function () {
            var qty = self.Quantity();
            if (qty == 0) {
                root.cart.push(self);
            }
            self.Quantity(qty + 1);
        };

        self.removeAllFromCart = function () {
            self.Quantity(0);
            root.cart.remove(self);
        };
    }
}
```

The `ProductViewModel` class contains two functions that are used to move the product to and from the cart: `addItemToCart` adds one unit of the product to the cart, and `removeAllFromCart` removes all quantities of the product.

Users can select an existing order and get the order details. We'll encapsulate this functionality into another view-model:

```
function AppViewModel() {
    // ...

    // NEW CODE
    function OrderDetailsViewModel(order) {
        var self = this;
        self.items = ko.observableArray();
        self.Id = order.Id;

        self.total = ko.computed(function () {
            var sum = 0;
            $.each(self.items(), function (index, item) {
                sum += item.Price * item.Quantity;
            });
            return '$' + sum.toFixed(2);
        });

        $.getJSON("/api/orders/" + order.Id, function (order) {
            $.each(order.Details, function (index, item) {
                self.items.push(item);
            })
        });
    };
}
```

The `OrderDetailsViewModel` is initialized with an order, and it fetches the order details by sending an AJAX request to the server.

Also, notice the `total` property on the `OrderDetailsViewModel`. This property is a special kind of observable called a computed observable. As the name implies, a computed observable lets you data bind to a computed value—in this case, the total cost of the order.

Next, add these functions to `AppViewModel`:

- `resetCart` removes all items from the cart.
- `getDetails` gets the details for an order (by pushing a new `OrderDetailsViewModel` onto the `details` list).
- `createOrder` creates a new order and empties the cart.

```
function AppViewModel() {
    // ...

    // NEW CODE
    self.resetCart = function() {
        var items = self.cart.removeAll();
        $.each(items, function (index, product) {
            product.Quantity(0);
        });
    }

    self.getDetails = function (order) {
        self.details(new OrderDetailsViewModel(order));
    }

    self.createOrder = function () {
        var jqxhr = $.ajax({
            type: 'POST',
            url: "api/orders",
            contentType: 'application/json; charset=utf-8',
            data: ko.toJSON({ Details: self.cart }),
            dataType: "json",
            success: function (newOrder) {
                self.resetCart();
                self.orders.push(newOrder);
            },
            error: function (jqXHR, textStatus, errorThrown) {
                self.errorMessage(errorThrown);
            }
        });
    };
};
```

Finally, initialize the view model by making AJAX requests for the products and orders:

```
function AppViewModel() {
    // ...

    // NEW CODE
    // Initialize the view-model.
    $.getJSON("/api/products", function (products) {
        $.each(products, function (index, product) {
            self.products.push(new ProductViewModel(self, product));
        })
    });

    $.getJSON("api/orders", self.orders);
};
```

OK, that's a lot of code, but we built it up step-by-step, so hopefully the design is clear. Now we can add some Knockout.js bindings to the HTML.

Products

Here are the bindings for the product list:

```
<ul id="products" data-bind="foreach: products">
    <li>
        <div>
            <span data-bind="text: Name"></span>
            <span class="price" data-bind="text: '$' + Price"></span>
        </div>
        <div data-bind="if: $parent.loggedIn">
            <button data-bind="click: addItemToCart">Add to Order</button>
        </div>
    </li>
</ul>
```

This iterates over the products array and displays the name and price. The "Add to Order" button is visible only when the user is logged in.

The "Add to Order" button calls `addItemToCart` on the `ProductViewModel` instance for the product. This demonstrates a nice feature of Knockout.js: When a view-model contains other view-models, you can apply the bindings to the inner model. In this example, the bindings within the `foreach` are applied to each of the `ProductViewModel` instances. This approach is much cleaner than putting all of the functionality into a single view-model.

Cart

Here are the bindings for the cart:

```
<div id="cart" class="float-right" data-bind="visible: cart().length > 0">
<h1>Your Cart</h1>
    <table class="details ui-widget-content">
    <thead>
        <tr><td>Item</td><td>Price</td><td>Quantity</td><td></td></tr>
    </thead>
    <tbody data-bind="foreach: cart">
        <tr>
            <td><span data-bind="text: $data.Name"></span></td>
            <td>$<span data-bind="text: $data.Price"></span></td>
            <td class="qty"><span data-bind="text: $data.Quantity()"></span></td>
            <td><a href="#" data-bind="click: removeAllFromCart">Remove</a></td>
        </tr>
    </tbody>
</table>
<input type="button" data-bind="click: createOrder" value="Create Order"/>
```

This iterates over the cart array and displays the name, price, and quantity. Note that the "Remove" link and the "Create Order" button are bound to view-model functions.

Orders

Here are the bindings for the orders list:

```
<h1>Your Orders</h1>
<ul id="orders" data-bind="foreach: orders">
<li class="ui-widget-content">
    <a href="#" data-bind="click: $root.getDetails">
        Order # <span data-bind="text: $data.Id"></span></a>
</li>
</ul>
```

This iterates over the orders and shows the order ID. The click event on the link is bound to the `getDetails` function.

Order Details

Here are the bindings for the order details:

```
<div id="order-details" class="float-right" data-bind="if: details()">
<h2>Order #<span data-bind="text: details().Id"></span></h2>
<table class="details ui-widget-content">
    <thead>
        <tr><td>Item</td><td>Price</td><td>Quantity</td><td>Subtotal</td></tr>
    </thead>
    <tbody data-bind="foreach: details().items">
        <tr>
            <td><span data-bind="text: $data.Product"></span></td>
            <td><span data-bind="text: $data.Price"></span></td>
            <td><span data-bind="text: $data.Quantity"></span></td>
            <td>
                <span data-bind="text: ($data.Price * $data.Quantity).toFixed(2)"></span>
            </td>
        </tr>
    </tbody>
</table>
<p>Total: <span data-bind="text: details().total"></span></p>
</div>
```

This iterates over the items in the order and displays the product, price, and quantity. The surrounding div is visible only if the details array contains one or more items.

Conclusion

In this tutorial, you created an application that uses Entity Framework to communicate with the database, and ASP.NET Web API to provide a public-facing interface on top of the data layer. We use ASP.NET MVC 4 to render the HTML pages, and Knockout.js plus jQuery to provide dynamic interactions without page reloads.

Additional resources:

- ASP.NET Data Access Content Map
- Entity Framework Developer Center

PREVIOUS

Self-Host ASP.NET Web API 1 (C#)

2/20/2020 • 6 minutes to read • Edit Online

by Mike Wasson

This tutorial shows how to host a web API inside a console application. ASP.NET Web API does not require IIS. You can self-host a web API in your own host process.

New applications should use OWIN to self-host Web API. See Use OWIN to Self-Host ASP.NET Web API 2.

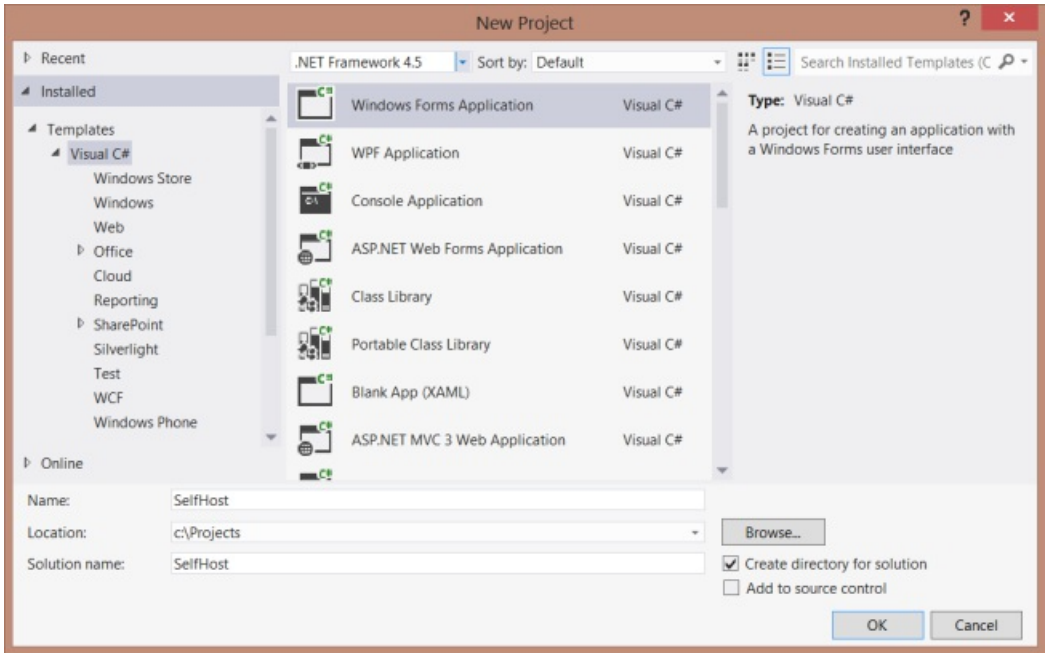
Software versions used in the tutorial

- Web API 1
- Visual Studio 2012

Create the Console Application Project

Start Visual Studio and select **New Project** from the **Start** page. Or, from the **File** menu, select **New** and then **Project**.

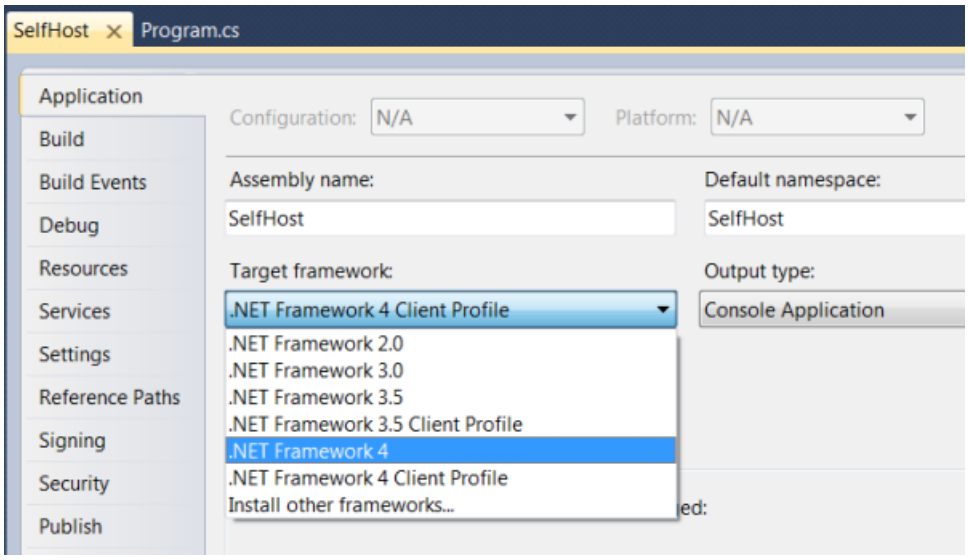
In the **Templates** pane, select **Installed Templates** and expand the **Visual C#** node. Under **Visual C#**, select **Windows**. In the list of project templates, select **Console Application**. Name the project "SelfHost" and click **OK**.



Set the Target Framework (Visual Studio 2010)

If you are using Visual Studio 2010, change the target framework to .NET Framework 4.0. (By default, the project template targets the .Net Framework Client Profile.)

In Solution Explorer, right-click the project and select **Properties**. In the **Target framework** dropdown list, change the target framework to .NET Framework 4.0. When prompted to apply the change, click **Yes**.



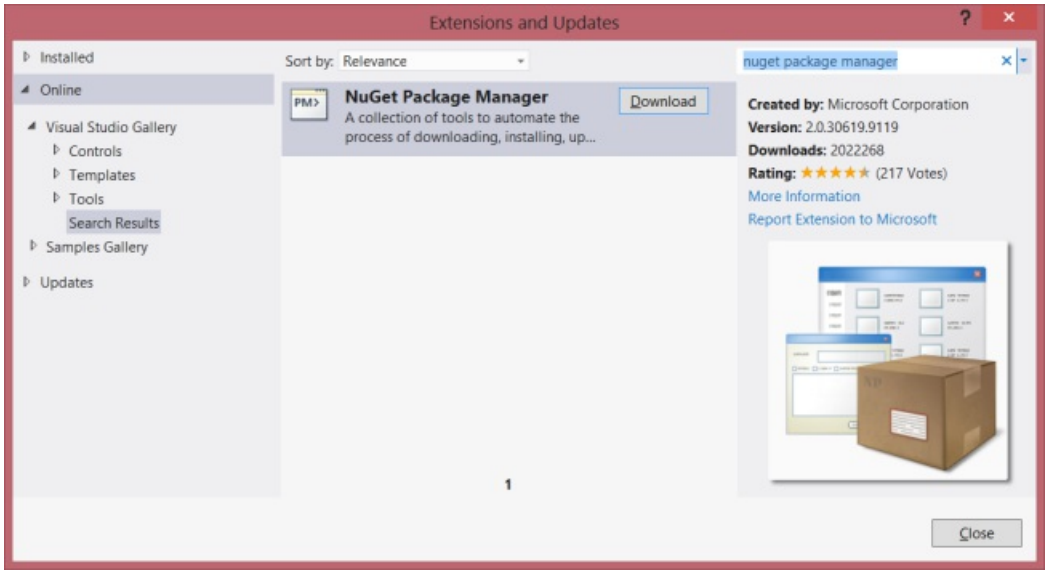
Install NuGet Package Manager

The NuGet Package Manager is the easiest way to add the Web API assemblies to a non-ASP.NET project.

To check if NuGet Package Manager is installed, click the **Tools** menu in Visual Studio. If you see a menu item called **NuGet Package Manager**, then you have NuGet Package Manager.

To install NuGet Package Manager:

1. Start Visual Studio.
2. From the **Tools** menu, select **Extensions and Updates**.
3. In the **Extensions and Updates** dialog, select **Online**.
4. If you don't see "NuGet Package Manager", type "nuget package manager" in the search box.
5. Select the NuGet Package Manager and click **Download**.
6. After the download completes, you will be prompted to install.
7. After the installation completes, you might be prompted to restart Visual Studio.



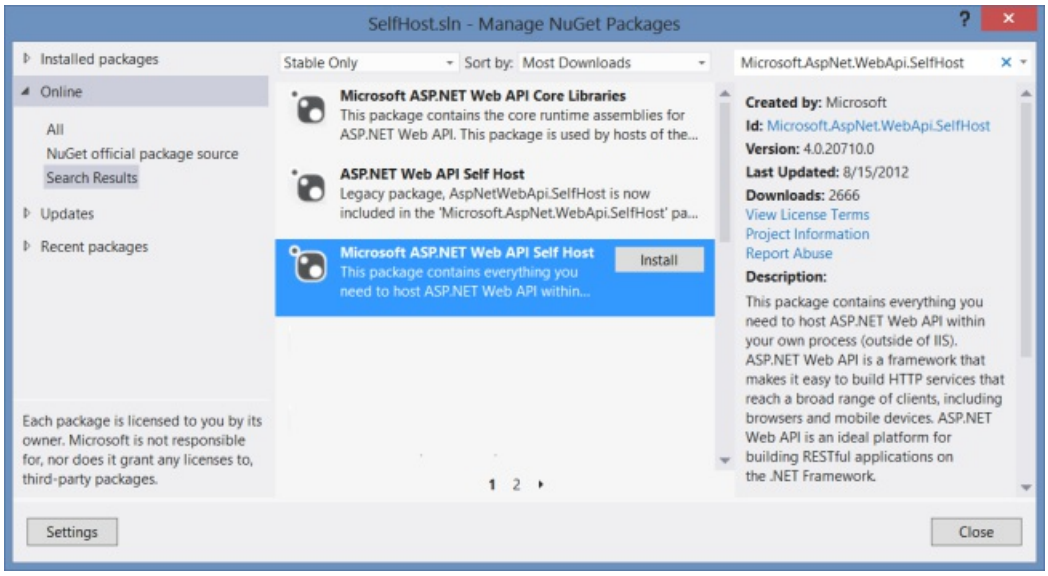
Add the Web API NuGet Package

After NuGet Package Manager is installed, add the Web API Self-Host package to your project.

1. From the **Tools** menu, select **NuGet Package Manager**. *Note*: If do you not see this menu item, make sure that NuGet Package Manager installed correctly.

2. Select **Manage NuGet Packages for Solution**
3. In the **Manage NugGet Packages** dialog, select **Online**.
4. In the search box, type "Microsoft.AspNet.WebApi.SelfHost".
5. Select the ASP.NET Web API Self Host package and click **Install**.
6. After the package installs, click **Close** to close the dialog.

> **NOTE**
>
> Make sure to install the package named Microsoft.AspNet.WebApi.SelfHost, not AspNetWebApi.SelfHost.



Create the Model and Controller

This tutorial uses the same model and controller classes as the Getting Started tutorial.

Add a public class named `Product`.

```
namespace SelfHost
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public decimal Price { get; set; }
    }
}
```

Add a public class named `ProductsController`. Derive this class from **System.Web.Http.ApiController**.

```
namespace SelfHost
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Net;
    using System.Web.Http;

    public class ProductsController : ApiController
    {
        Product[] products = new Product[]
        {
            new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
            new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
            new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
        };

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public Product GetProductById(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                throw new HttpResponseException(HttpStatusCode.NotFound);
            }
            return product;
        }

        public IEnumerable<Product> GetProductsByCategory(string category)
        {
            return products.Where(p => string.Equals(p.Category, category,
                    StringComparison.OrdinalIgnoreCase));
        }
    }
}
```

For more information about the code in this controller, see the Getting Started tutorial. This controller defines three GET actions:

URI	DESCRIPTION
/api/products	Get a list of all products.
/api/products/*id*	Get a product by ID.
/api/products/?category=*category*	Get a list of products by category.

Host the Web API

Open the file Program.cs and add the following using statements:

```
using System.Web.Http;
using System.Web.Http.SelfHost;
```

Add the following code to the **Program** class.

```
var config = new HttpSelfHostConfiguration("http://localhost:8080");

config.Routes.MapHttpRoute(
    "API Default", "api/{controller}/{id}",
    new { id = RouteParameter.Optional });

using (HttpSelfHostServer server = new HttpSelfHostServer(config))
{
    server.OpenAsync().Wait();
    Console.WriteLine("Press Enter to quit.");
    Console.ReadLine();
}
```

(Optional) Add an HTTP URL Namespace Reservation

This application listens to `http://localhost:8080/`. By default, listening at a particular HTTP address requires administrator privileges. When you run the tutorial, therefore, you may get this error: "HTTP could not register URL http://+:8080/" There are two ways to avoid this error:

- Run Visual Studio with elevated administrator permissions, or
- Use Netsh.exe to give your account permissions to reserve the URL.

To use Netsh.exe, open a command prompt with administrator privileges and enter the following command:following command:

```
netsh http add urlacl url=http://+:8080/ user=machine\username
```

where *machine**username* is your user account.

When you are finished self-hosting, be sure to delete the reservation:

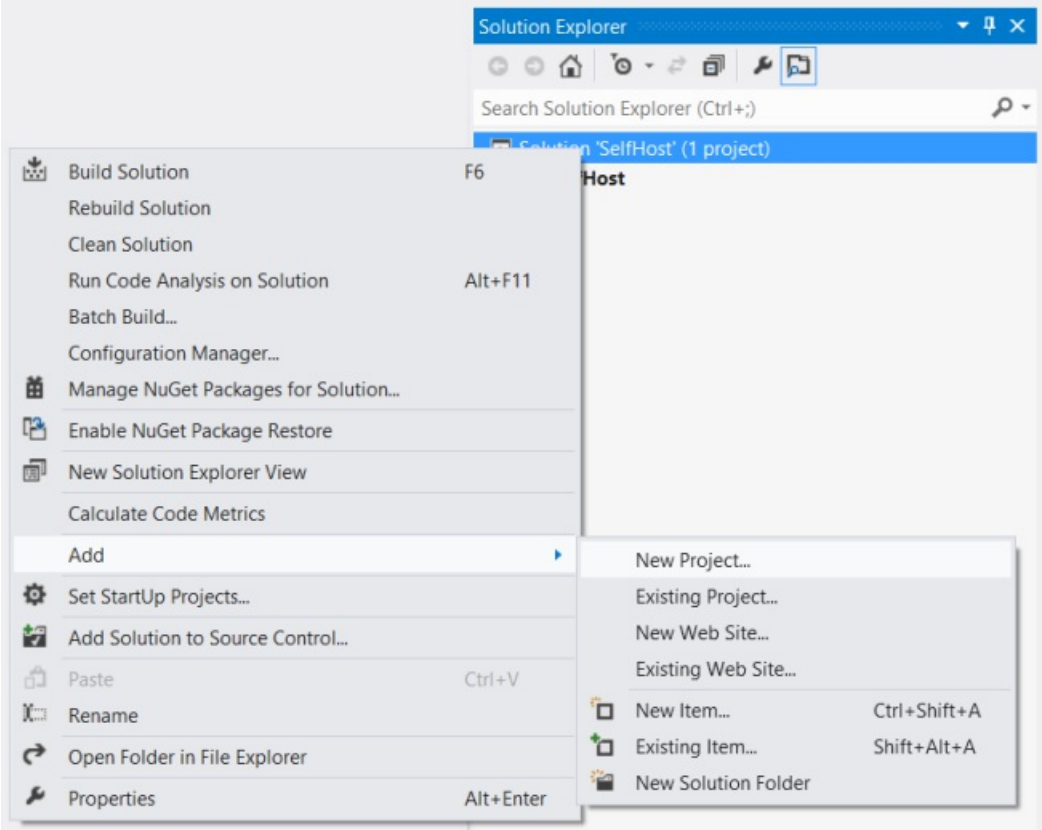
```
netsh http delete urlacl url=http://+:8080/
```

Call the Web API from a Client Application (C#)

Let's write a simple console application that calls the web API.

Add a new console application project to the solution:

- In Solution Explorer, right-click the solution and select **Add New Project**.
- Create a new console application named "ClientApp".

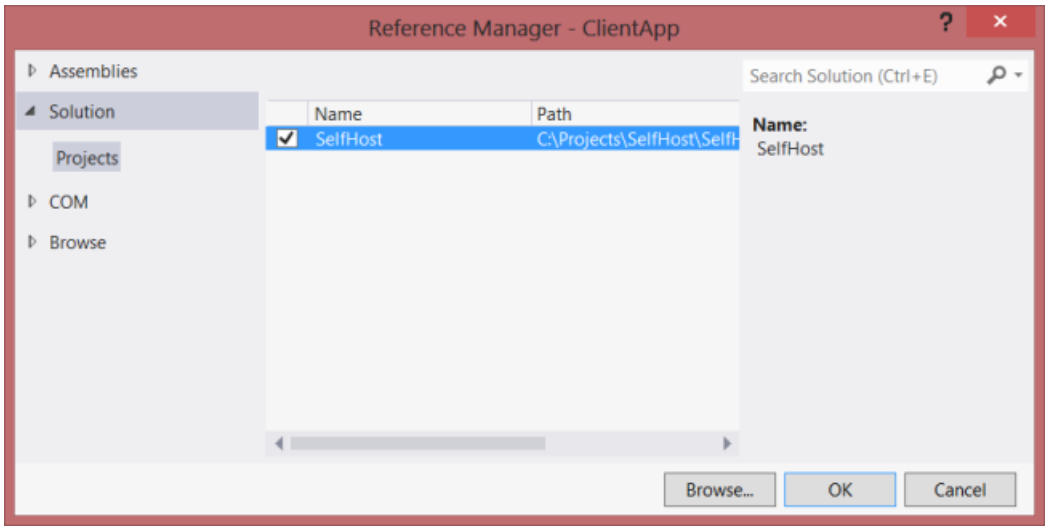


Use NuGet Package Manager to add the ASP.NET Web API Core Libraries package:

- From the Tools menu, select **NuGet Package Manager**.
- Select **Manage NuGet Packages for Solution**
- In the **Manage NuGet Packages** dialog, select **Online**.
- In the search box, type "Microsoft.AspNet.WebApi.Client".
- Select the Microsoft ASP.NET Web API Client Libraries package and click **Install**.

Add a reference in ClientApp to the SelfHost project:

- In Solution Explorer, right-click the ClientApp project.
- Select **Add Reference**.
- In the **Reference Manager** dialog, under **Solution**, select **Projects**.
- Select the SelfHost project.
- Click **OK**.



Open the Client/Program.cs file. Add the following **using** statement:

```
using System.Net.Http;
```

Add a static **HttpClient** instance:

```
namespace Client
{
    class Program
    {
        static HttpClient client = new HttpClient();
    }
}
```

Add the following methods to list all products, list a product by ID, and list products by category.

```
static void ListAllProducts()
{
    HttpResponseMessage resp = client.GetAsync("api/products").Result;
    resp.EnsureSuccessStatusCode();

    var products = resp.Content.ReadAsAsync<IEnumerable<SelfHost.Product>>().Result;
    foreach (var p in products)
    {
        Console.WriteLine("{0} {1} {2} ({3})", p.Id, p.Name, p.Price, p.Category);
    }
}

static void ListProduct(int id)
{
    var resp = client.GetAsync(string.Format("api/products/{0}", id)).Result;
    resp.EnsureSuccessStatusCode();

    var product = resp.Content.ReadAsAsync<SelfHost.Product>().Result;
    Console.WriteLine("ID {0}: {1}", id, product.Name);
}

static void ListProducts(string category)
{
    Console.WriteLine("Products in '{0}':", category);

    string query = string.Format("api/products?category={0}", category);

    var resp = client.GetAsync(query).Result;
    resp.EnsureSuccessStatusCode();

    var products = resp.Content.ReadAsAsync<IEnumerable<SelfHost.Product>>().Result;
    foreach (var product in products)
    {
        Console.WriteLine(product.Name);
    }
}
```

Each of these methods follows the same pattern:

1. Call **HttpClient.GetAsync** to send a GET request to the appropriate URI.
2. Call **HttpResponseMessage.EnsureSuccessStatusCode**. This method throws an exception if the HTTP response status is an error code.
3. Call **ReadAsAsync<T>** to deserialize a CLR type from the HTTP response. This method is an extension method, defined in **System.Net.Http.HttpContentExtensions**.

The **GetAsync** and **ReadAsAsync** methods are both asynchronous. They return **Task** objects that represent the asynchronous operation. Getting the **Result** property blocks the thread until the operation completes.

For more information about using HttpClient, including how to make non-blocking calls, see Calling a Web API From a .NET Client.

Before calling these methods, set the BaseAddress property on the HttpClient instance to "`http://localhost:8080`". For example:

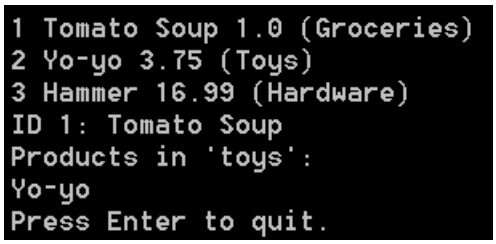
```
static void Main(string[] args)
{
    client.BaseAddress = new Uri("http://localhost:8080");

    ListAllProducts();
    ListProduct(1);
    ListProducts("toys");

    Console.WriteLine("Press Enter to quit.");
    Console.ReadLine();
}
```

This should output the following. (Remember to run the SelfHost application first.)

```
1 Tomato Soup 1.0 (Groceries)
2 Yo-yo 3.75 (Toys)
3 Hammer 16.99 (Hardware)
ID 1: Tomato Soup
Products in 'toys':
Yo-yo
Press Enter to quit.
```



Build RESTful APIs with ASP.NET Web API

2/20/2020 • 24 minutes to read • Edit Online

by Web Camps Team

> Hands on lab: Use Web API in ASP.NET 4.x to build a simple REST API for a contact manager application. You will also build a client to consume the API.

In recent years, it has become clear that HTTP is not just for serving up HTML pages. It is also a powerful platform for building Web APIs, using a handful of verbs (GET, POST, and so forth) plus a few simple concepts such as *URIs* and *headers*. ASP.NET Web API is a set of components that simplify HTTP programming. Because it is built on top of the ASP.NET MVC runtime, Web API automatically handles the low-level transport details of HTTP. At the same time, Web API naturally exposes the HTTP programming model. In fact, one goal of Web API is to *not* abstract away the reality of HTTP. As a result, Web API is both flexible and easy to extend. The REST architectural style has proven to be an effective way to leverage HTTP - although it is certainly not the only valid approach to HTTP. The contact manager will expose the RESTful for listing, adding and removing contacts, among others.

This lab requires a basic understanding of HTTP, REST, and assumes you have a basic working knowledge of HTML, JavaScript, and jQuery.

> **NOTE**
>
> The ASP.NET Web site has an area dedicated to the ASP.NET Web API framework at https://asp.net/web-api. This site will continue to provide late-breaking information, samples, and news related to Web API, so check it frequently if you'd like to delve deeper into the art of creating custom Web APIs available to virtually any device or development framework.
>
> ASP.NET Web API, similar to ASP.NET MVC 4, has great flexibility in terms of separating the service layer from the controllers allowing you to use several of the available Dependency Injection frameworks fairly easy. There is a good sample in MSDN that shows how to use Ninject for dependency injection in an ASP.NET Web API project that you can download it from here.

All sample code and snippets are included in the Web Camps Training Kit, available at https://go.microsoft.com/fwlink/?LinkID=248297&clcid=0x409.

Objectives

In this hands-on lab, you will learn how to:

- Implement a RESTful Web API
- Call the API from an HTML client

Prerequisites

The following is required to complete this hands-on lab:

- Microsoft Visual Studio Express 2012 for Web or superior (read Appendix B for instructions on how to install it).

Setup

Installing Code Snippets

For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. To install the code snippets run `.\Source\Setup\CodeSnippets.vsi` file.

If you are not familiar with the Visual Studio Code Snippets, and want to learn how to use them, you can refer to the appendix from this document "Appendix A: Using Code Snippets".

Exercises

This hands-on lab includes the following exercise:

1. Exercise 1: Create a Read-Only Web API
2. Exercise 2: Create a Read/Write Web API
3. Exercise 3: Consume the Web API from an HTML Client

> **NOTE**
>
> Each exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

Estimated time to complete this lab: **60 minutes**.

Exercise 1: Create a Read-Only Web API

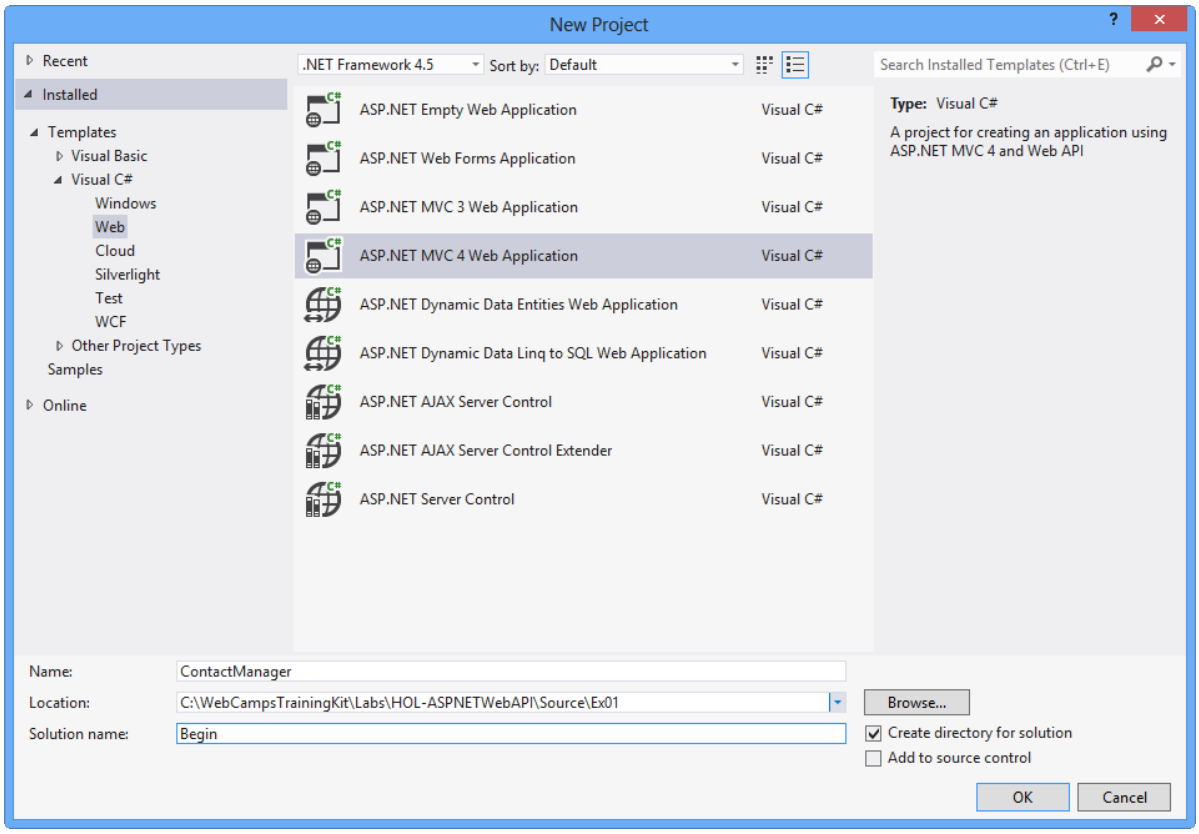
In this exercise, you will implement the read-only GET methods for the contact manager.

Task 1 - Creating the API Project

In this task, you will use the new ASP.NET web project templates to create a Web API web application.

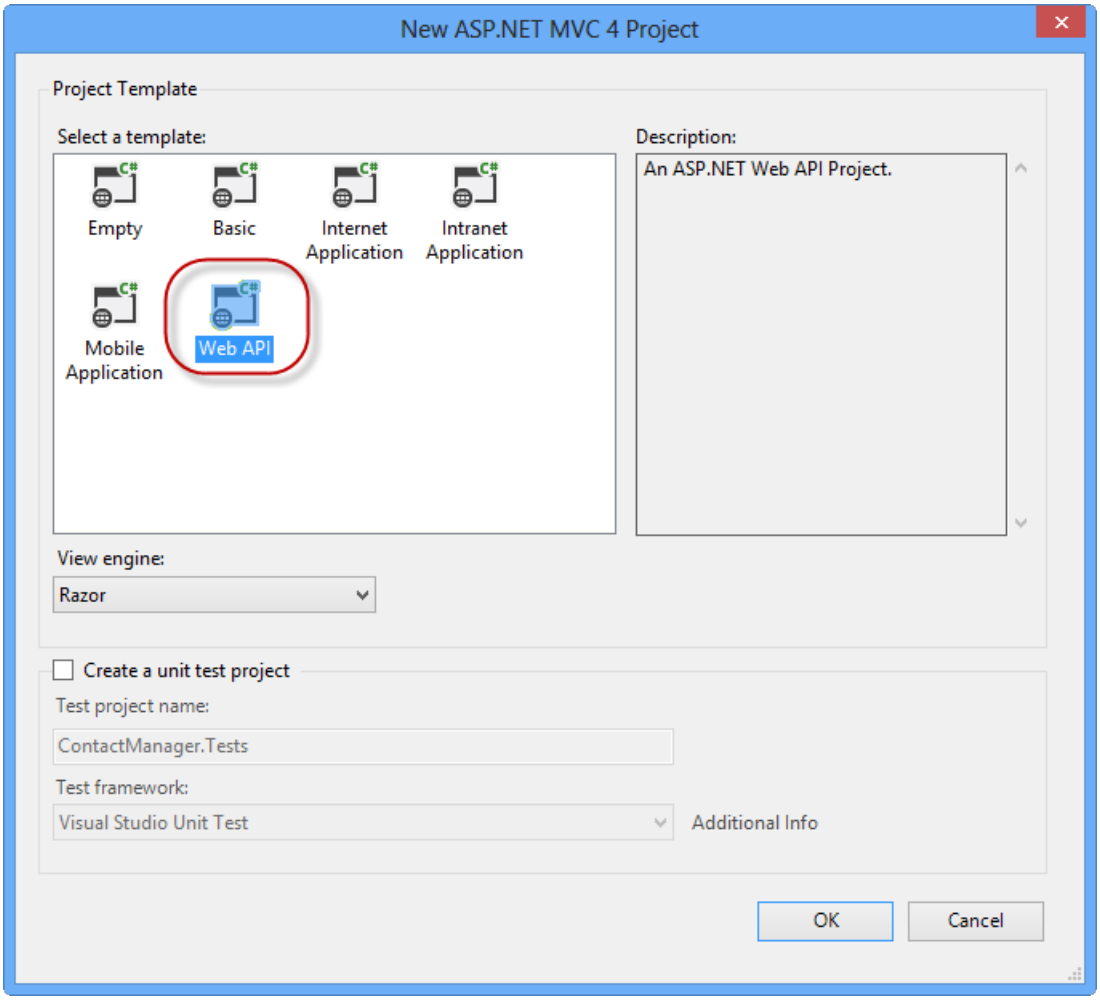
1. Run **Visual Studio 2012 Express for Web**, to do this go to **Start** and type **VS Express for Web** then press **Enter**.

2. From the **File** menu, select **New Project**. Select the **Visual C# | Web** project type from the project type tree view, then select the **ASP.NET MVC 4 Web Application** project type. Set the project's **Name** to *ContactManager* and the **Solution name** to *Begin*, then click **OK**.



Creating a new ASP.NET MVC 4.0 Web Application Project

3. In the ASP.NET MVC 4 project type dialog, select the **Web API** project type. Click **OK**.



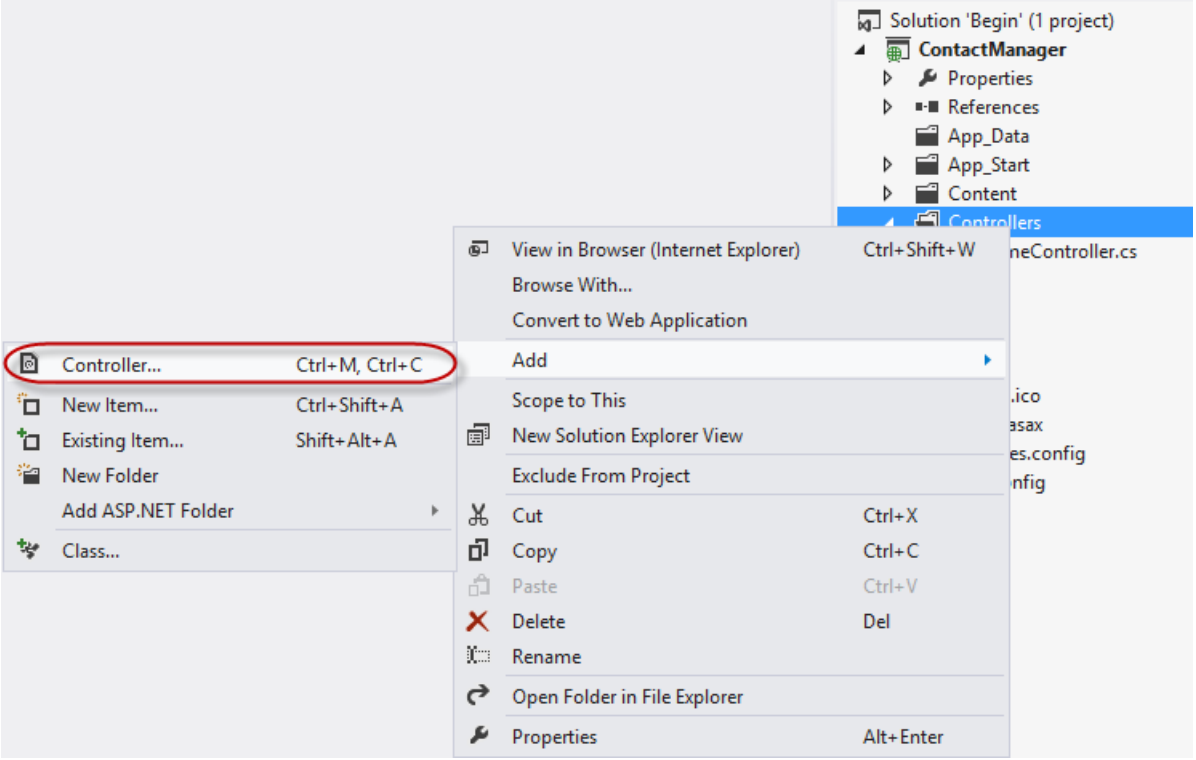
Specifying the Web API project type

Task 2 - Creating the Contact Manager API Controllers

In this task, you will create the controller classes in which API methods will reside.

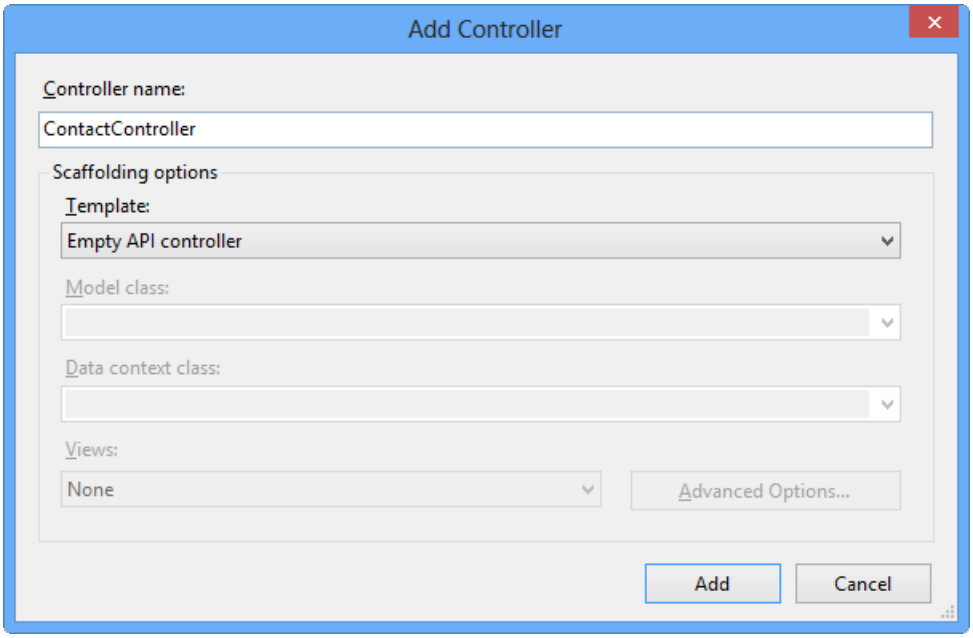
1. Delete the file named **ValuesController.cs** within **Controllers** folder from the project.

2. Right-click the **Controllers** folder in the project and select **Add | Controller** from the context menu.



Adding a new controller to the project

3. In the **Add Controller** dialog that appears, select **Empty API Controller** from the Template menu. Name the controller class **ContactController**. Then, click **Add.**



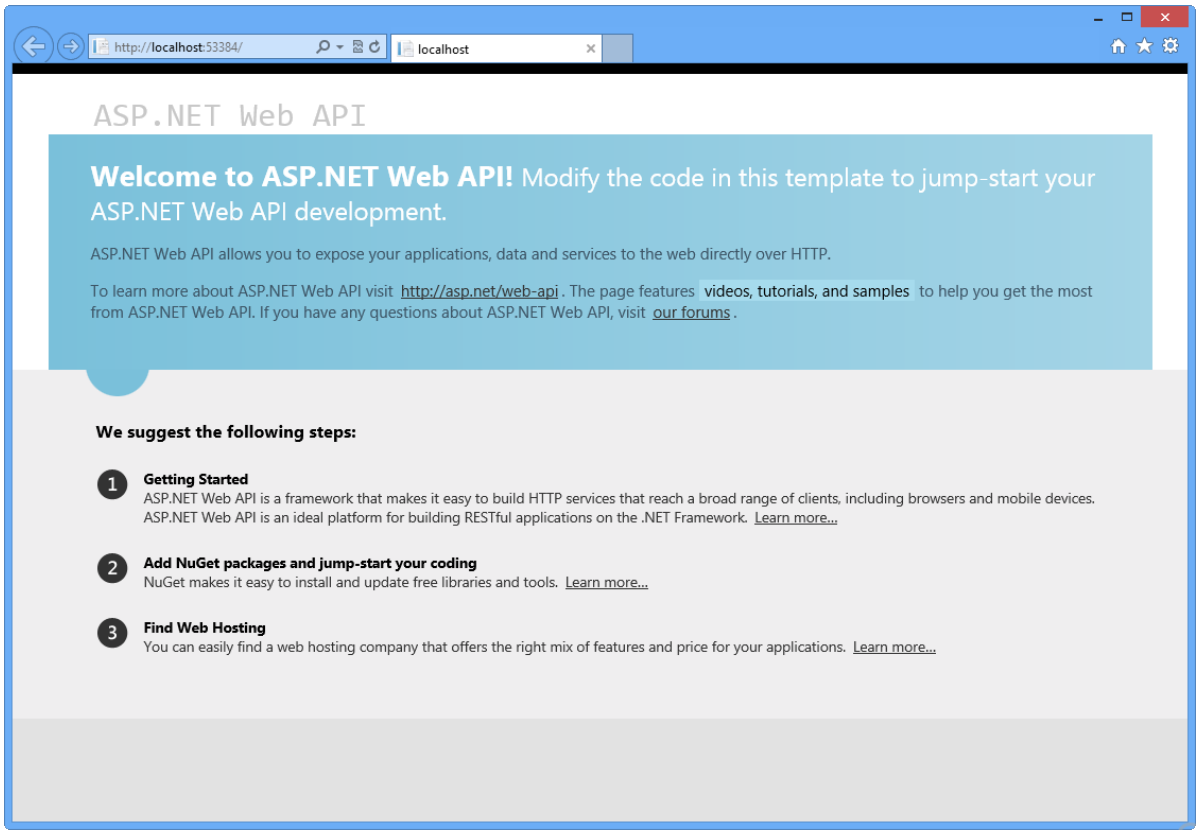
Using the Add Controller dialog to create a new Web API controller

4. Add the following code to the **ContactController**.

(Code Snippet - *Web API Lab - Ex01 - Get API Method*)

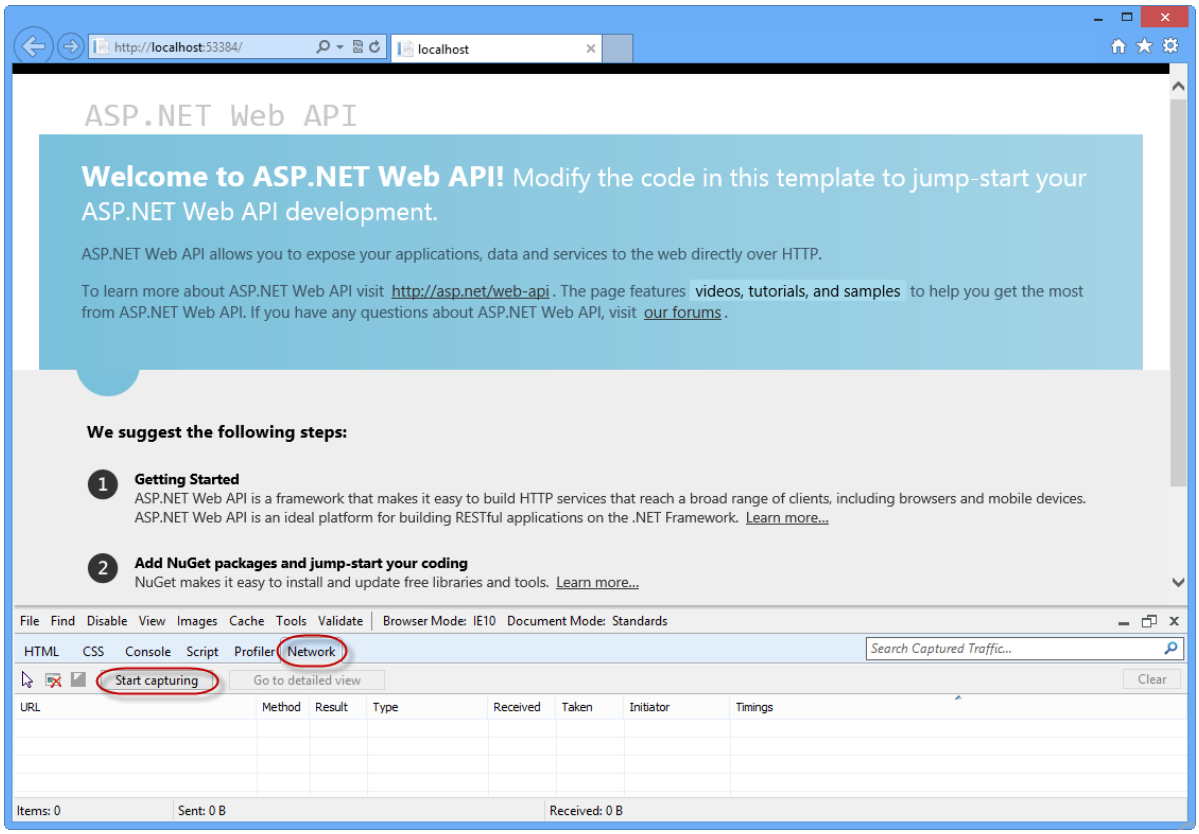
```
public string[] Get()
{
    return new string[]
    {
        "Hello",
        "World"
    };
}
```

5. Press **F5** to debug the application. The default home page for a Web API project should appear.



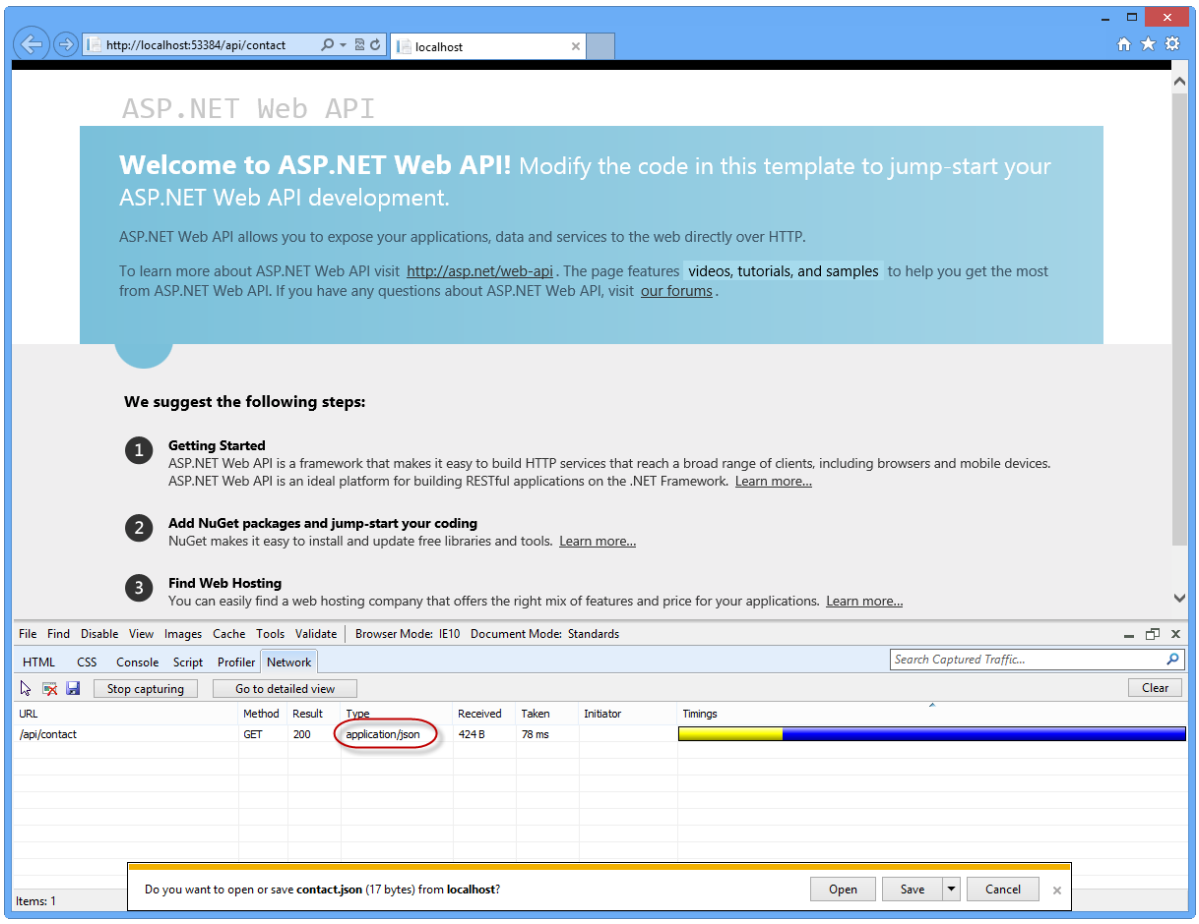
The default home page of an ASP.NET Web API application

6. In the Internet Explorer window, press the **F12** key to open the **Developer Tools** window. Click the **Network** tab, and then click the **Start Capturing** button to begin capturing network traffic into the window.



Opening the network tab and initiating network capture

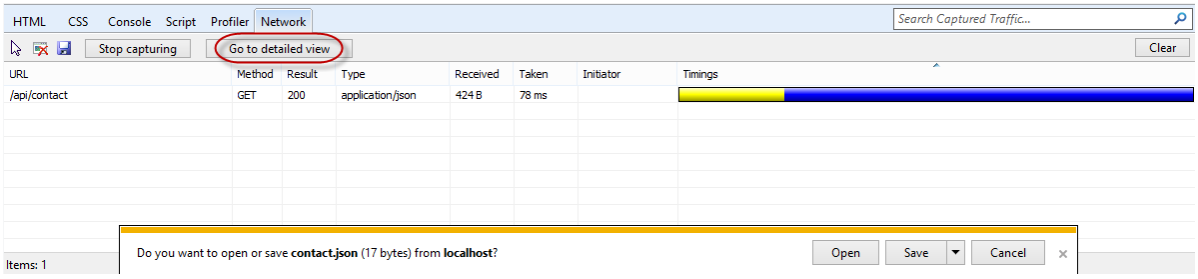
7. Append the URL in the browser's address bar with **/api/contact** and press enter. The transmission details will appear in the network capture window. Note that the response's MIME type is **application/json**. This demonstrates how the default output format is JSON.



Viewing the output of the Web API request in the Network view

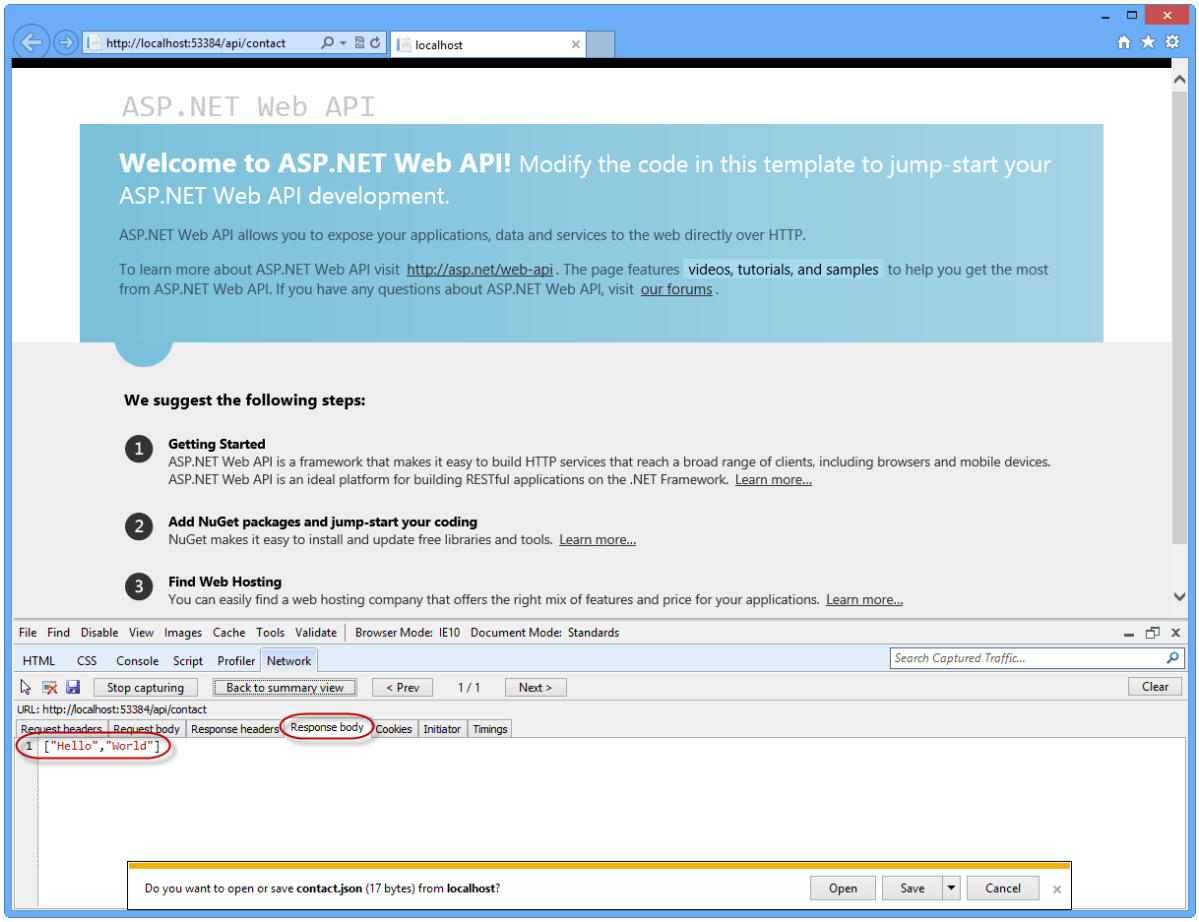
> **NOTE**
>
> Internet Explorer 10's default behavior at this point will be to ask if the user would like to save or open the stream resulting from the Web API call. The output will be a text file containing the JSON result of the Web API URL call. Do not cancel the dialog in order to be able to watch the response's content through Developers Tool window.

8. Click the **Go to detailed view** button to see more details about the response of this API call.



Switch to Detailed View

9. Click the **Response body** tab to view the actual JSON response text.

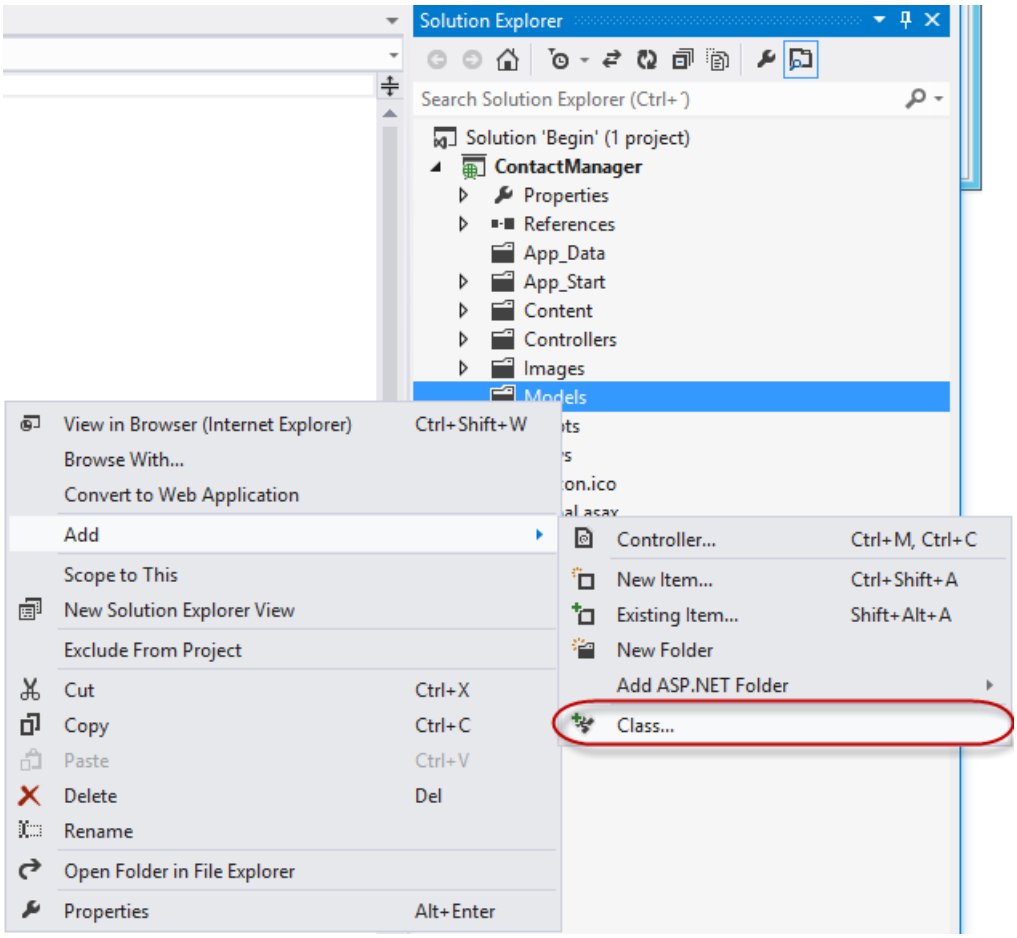


Viewing the JSON output text in the network monitor

Task 3 - Creating the Contact Models and Augment the Contact Controller

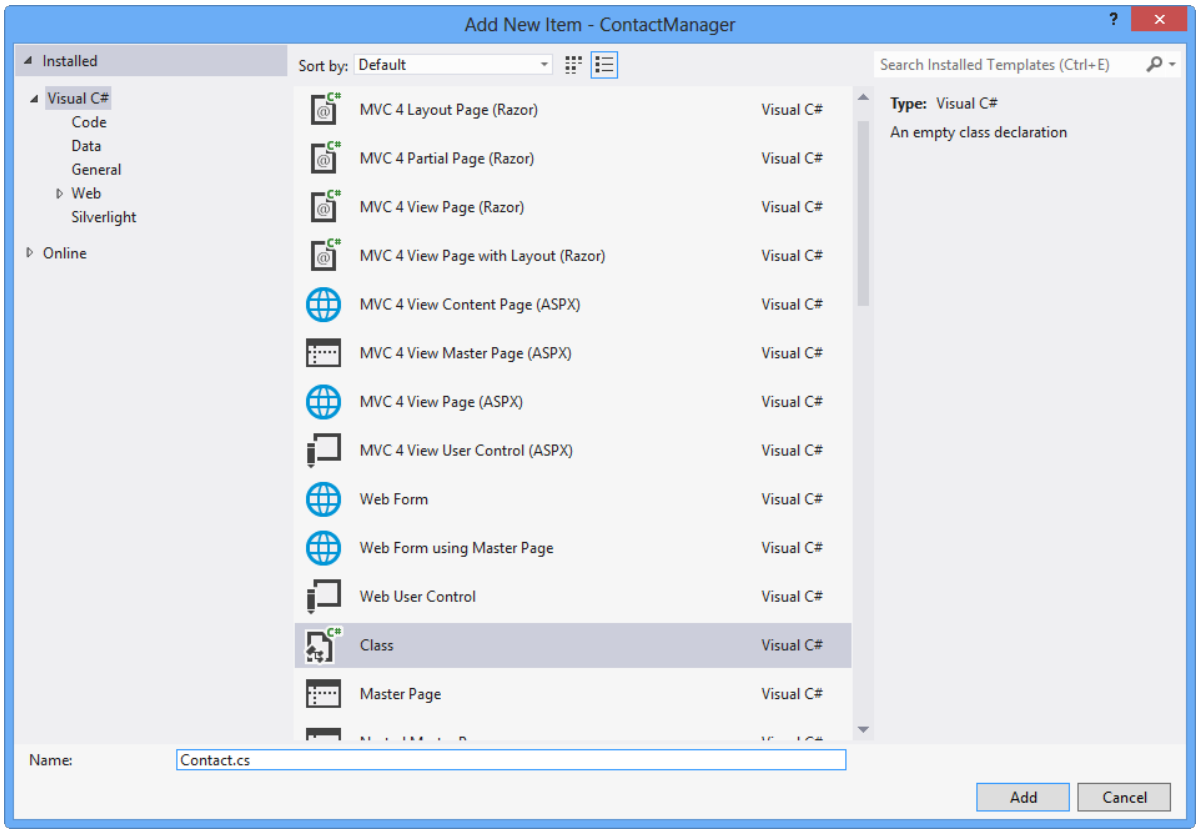
In this task, you will create the controller classes in which API methods will reside.

1. Right-click the **Models** folder and select **Add | Class...** from the context menu.



Adding a new model to the web application

2. In the **Add New Item** dialog, name the new file **Contact.cs** and click **Add.**



Creating the new Contact class file

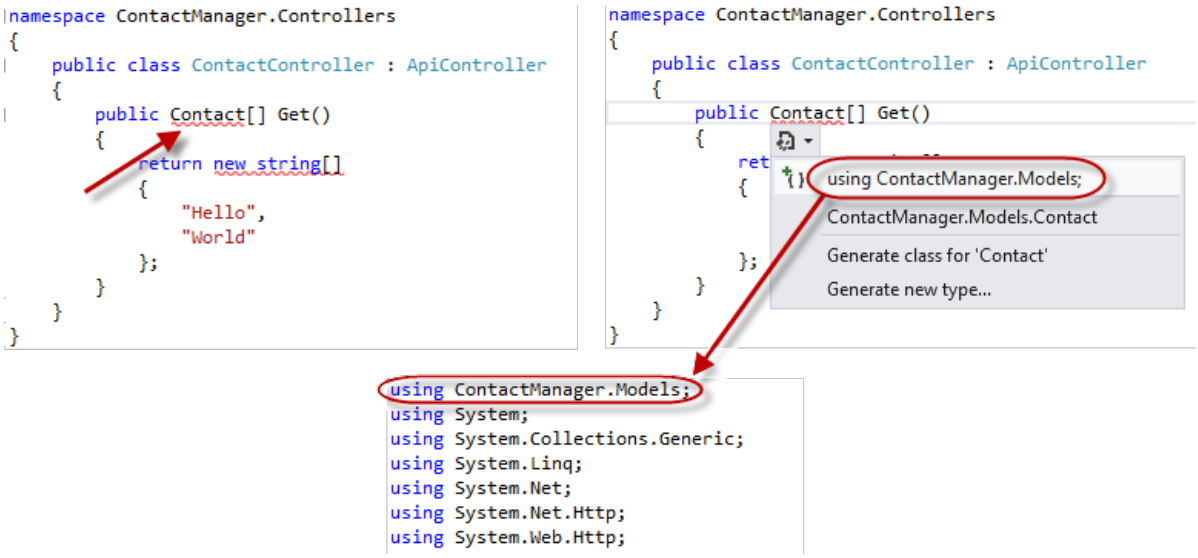
3. Add the following highlighted code to the **Contact** class.

(Code Snippet - *Web API Lab - Ex01 - Contact Class*)

```
public class Contact
{
    public int Id { get; set; }

    public string Name { get; set; }
}
```

4. In the **ContactController** class, select the word **string** in method definition of the **Get** method, and type the word *Contact*. Once the word is typed in, an indicator will appear at the beginning of the word **Contact**. Either hold down the **Ctrl** key and press the period (.) key or click the icon using your mouse to open up the assistance dialog in the code editor, to automatically fill in the **using** directive for the Models namespace.



Using Intellisense assistance for namespace declarations

5. Modify the code for the **Get** method so that it returns an array of Contact model instances.

(Code Snippet - *Web API Lab - Ex01 - Returning a list of contacts*)

```
public Contact[] Get()
{
    return new Contact[]
    {
        new Contact
        {
            Id = 1,
            Name = "Glenn Block"
        },
        new Contact
        {
            Id = 2,
            Name = "Dan Roth"
        }
    };
}
```

6. Press **F5** to debug the web application in the browser. To view the changes made to the response output of the API, perform the following steps.

   a. Once the browser opens, press **F12** if the developer tools are not open yet.

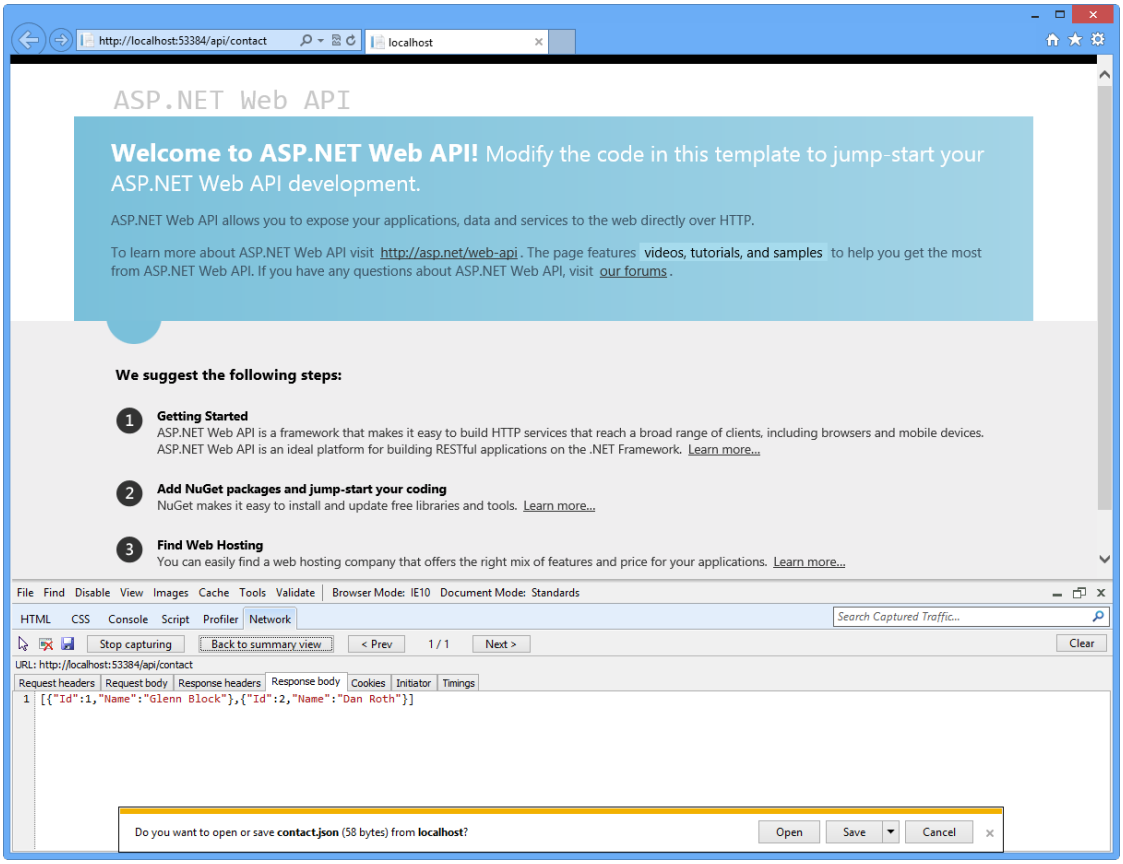
   b. Click the **Network** tab.

c. Press the **Start Capturing** button.

d. Add the URL suffix **/api/contact** to the URL in the address bar and press the **Enter** key.

e. Press the **Go to detailed view** button.

f. Select the **Response body** tab. You should see a JSON string representing the serialized form of an array of Contact instances.

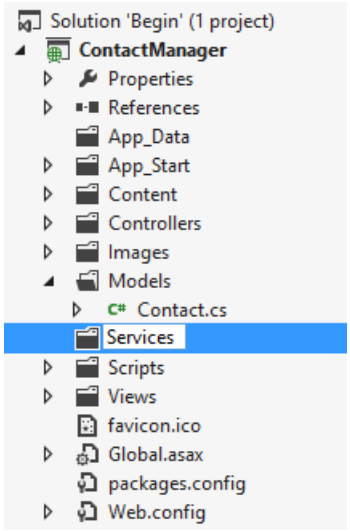


JSON serialized output of a complex Web API method call

Task 4 - Extracting Functionality into a Service Layer

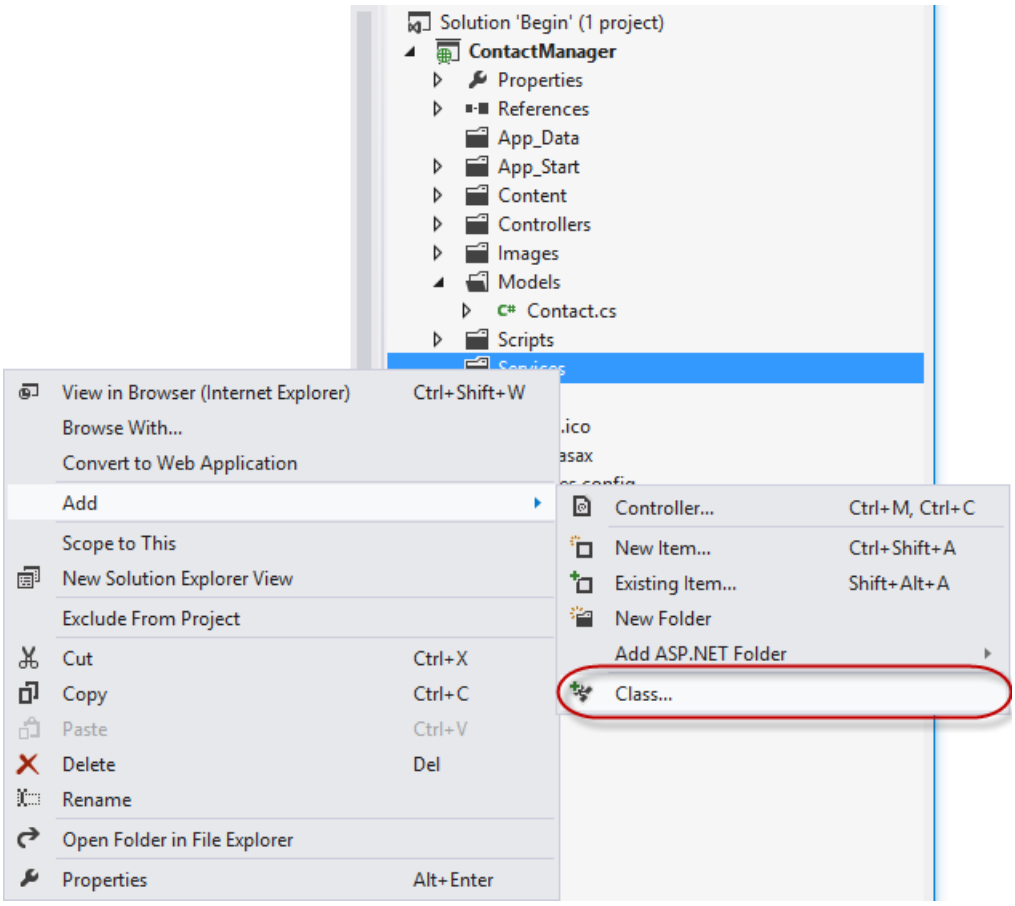
This task will demonstrate how to extract functionality into a Service layer to make it easy for developers to separate their service functionality from the controller layer, thereby allowing reusability of the services that actually do the work.

1. Create a new folder in the solution root and name it **Services**. To do this, right-click **ContactManager** project, select **Add | New Folder**, name it *Services*.



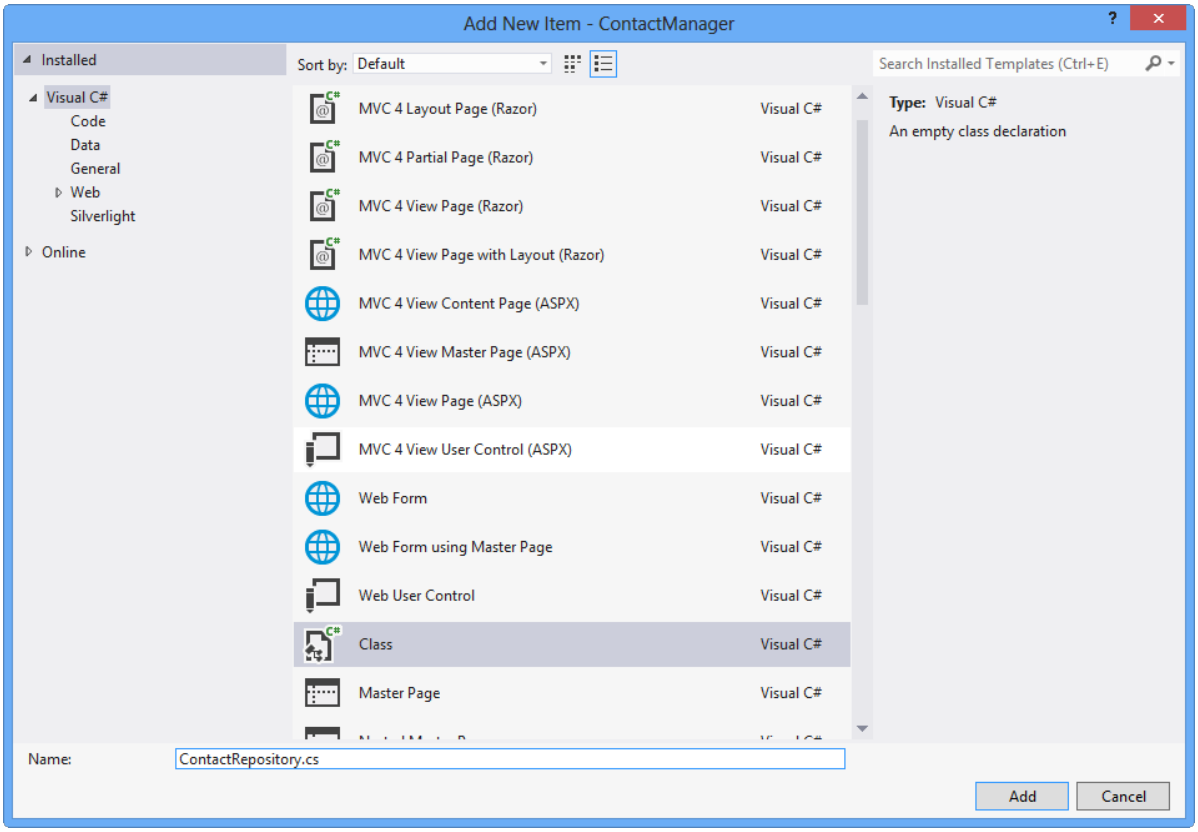
Creating Services folder

2. Right-click the **Services** folder and select **Add | Class...** from the context menu.



Adding a new class to the Services folder

3. When the **Add New Item** dialog appears, name the new class **ContactRepository** and click **Add**.



Creating a class file to contain the code for the Contact Repository service layer

4. Add a using directive to the **ContactRepository.cs** file to include the models namespace.

```
using ContactManager.Models;
```

5. Add the following highlighted code to the **ContactRepository.cs** file to implement GetAllContacts method.

   (Code Snippet - *Web API Lab - Ex01 - Contact Repository*)

```
public class ContactRepository
{
    public Contact[] GetAllContacts()
    {
        return new Contact[]
        {
            new Contact
            {
                Id = 1,
                Name = "Glenn Block"
            },
            new Contact
            {
                Id = 2,
                Name = "Dan Roth"
            }
        };
    }
}
```

6. Open the **ContactController.cs** file if it is not already open.

7. Add the following using statement to the namespace declaration section of the file.

```
using ContactManager.Services;
```

8. Add the following highlighted code to the **ContactController.cs** class to add a private field to represent the instance of the repository, so that the rest of the class members can make use of the service implementation.

   (Code Snippet - *Web API Lab - Ex01 - Contact Controller*)

```
public class ContactController : ApiController
{
    private ContactRepository contactRepository;

    public ContactController()
    {
        this.contactRepository = new ContactRepository();
    }
    ...
}
```

9. Change the **Get** method so that it makes use of the contact repository service.

   (Code Snippet - *Web API Lab - Ex01 - Returning a list of contacts via the repository*)

```
public Contact[] Get()
{
    return contactRepository.GetAllContacts();
}
```

10. Put a breakpoint on the **ContactController**'s **Get** method definition.

```
namespace ContactManager.Controllers
{
    public class ContactController : ApiController
    {
        private ContactRepository contactRepository;

        public ContactController()
        {
            this.contactRepository = new ContactRepository();
        }

        public Contact[] Get()
        {
            return this.contactRepository.GetAllContacts();
        }
    }
}
```

Adding breakpoints to the contact controller 11. Press **F5** to run the application. 12. When the browser opens, press **F12** to open the developer tools. 13. Click the **Network** tab. 14. Click the **Start Capturing** button. 15. Append the URL in the address bar with the suffix **/api/contact** and press **Enter** to load the API controller. 16. Visual Studio 2012 should break once **Get** method begins execution.

```
public Contact[] Get()
{
    return this.contactRepository.GetAllContacts();
}
```

this.contactRepository {ContactManager.Services.ContactRepository}

Breaking within the Get method 17. Press **F5** to continue. 18. Go back to Internet Explorer if it is not already in focus. Note the network capture window.

```
![Network view in Internet Explorer showing results of the Web API call](build-restful-apis-with-aspnet-web-
api/_static/image19.png "Network view in Internet Explorer showing results of the Web API call")

*Network view in Internet Explorer showing results of the Web API call*
```

19. Click the **Go to detailed view** button.

20. Click the **Response body** tab. Note the JSON output of the API call, and how it represents the two contacts retrieved by the service layer.

HTML CSS Console Script Profiler Network Search Captured Traffic...

Stop capturing | Back to summary view | < Prev | 1 / 1 | Next > | Clear

URL: http://localhost:53384/api/contact

Request headers | Request body | Response headers | Response body | Cookies | Initiator | Timings

```
1 [{"Id":1,"Name":"Glenn Block"},{"Id":2,"Name":"Dan Roth"}]
```

Do you want to open or save **contact.json** (58 bytes) from **localhost**? Open | Save | Cancel

Viewing the JSON output from the Web API in the developer tools window

Exercise 2: Create a Read/Write Web API

In this exercise, you will implement POST and PUT methods for the contact manager to enable it with data-editing features.

Task 1 - Opening the Web API Project

In this task, you will prepare to enhance the Web API project created in Exercise 1 so that it can accept user input.

1. Run **Visual Studio 2012 Express for Web**, to do this go to **Start** and type **VS Express for Web** then press **Enter**.
2. Open the **Begin** solution located at **Source/Ex02-ReadWriteWebAPI/Begin/** folder. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.
   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.
   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.
   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

3. Open the **Services/ContactRepository.cs** file.

Task 2 - Adding Data-Persistence Features to the Contact Repository Implementation

In this task, you will augment the ContactRepository class of the Web API project created in Exercise 1 so that it can persist and accept user input and new Contact instances.

1. Add the following constant to the **ContactRepository** class to represent the name of the web server cache item key name later in this exercise.

```
private const string CacheKey = "ContactStore";
```

2. Add a constructor to the **ContactRepository** containing the following code.

   (Code Snippet - *Web API Lab - Ex02 - Contact Repository Constructor*)

```
public ContactRepository()
{
    var ctx = HttpContext.Current;

    if (ctx != null)
    {
        if (ctx.Cache[CacheKey] == null)
        {
            var contacts = new Contact[]
            {
                new Contact
                {
                    Id = 1, Name = "Glenn Block"
                },
                new Contact
                {
                    Id = 2, Name = "Dan Roth"
                }
            };

            ctx.Cache[CacheKey] = contacts;
        }
    }
}
```

3. Modify the code for the **GetAllContacts** method as demonstrated below.

   (Code Snippet - *Web API Lab - Ex02 - Get All Contacts*)

```
public Contact[] GetAllContacts()
{
    var ctx = HttpContext.Current;

    if (ctx != null)
    {
        return (Contact[])ctx.Cache[CacheKey];
    }

    return new Contact[]
        {
            new Contact
            {
                Id = 0,
                Name = "Placeholder"
            }
        };
}
```

> **NOTE**
>
> This example is for demonstration purposes and will use the web server's cache as a storage medium, so that the values will be available to multiple clients simultaneously, rather than use a Session storage mechanism or a Request storage lifetime. One could use Entity Framework, XML storage, or any other variety in place of the web server cache.

4. Implement a new method named **SaveContact** to the **ContactRepository** class to do the work of saving a contact. The **SaveContact** method should take a single **Contact** parameter and return a Boolean value indicating success or failure.

   (Code Snippet - *Web API Lab - Ex02 - Implementing the SaveContact Method*)

```
public bool SaveContact(Contact contact)
{
    var ctx = HttpContext.Current;

    if (ctx != null)
    {
         try
         {
              var currentData = ((Contact[])ctx.Cache[CacheKey]).ToList();
              currentData.Add(contact);
              ctx.Cache[CacheKey] = currentData.ToArray();

              return true;
         }
         catch (Exception ex)
         {
              Console.WriteLine(ex.ToString());
              return false;
         }
    }

    return false;
}
```

Exercise 3: Consume the Web API from an HTML Client

In this exercise, you will create an HTML client to call the Web API. This client will facilitate data exchange with the Web API using JavaScript and will display the results in a web browser using HTML markup.

Task 1 - Modifying the Index View to Provide a GUI for Displaying Contacts

In this task, you will modify the default Index view of the web application to support the requirement of displaying the list of existing contacts in an HTML browser.

1. Open **Visual Studio 2012 Express for Web** if it is not already open.
2. Open the **Begin** solution located at **Source/Ex03-ConsumingWebAPI/Begin/** folder. Otherwise, you might continue using the **End** solution obtained by completing the previous exercise.
   a. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.
   b. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.
   c. Finally, build the solution by clicking **Build | Build Solution**.

   > **NOTE**
   > One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

3. Open the **Index.cshtml** file located at **Views/Home** folder.
4. Replace the HTML code within the div element with id **body** so that it looks like the following code.

```
<div id="body">
    <ul id="contacts"></ul>
</div>
```

5. Add the following Javascript code at the bottom of the file to perform the HTTP request to the Web API.

```
@section scripts{
<script type="text/javascript">
$(function()
{
        $.getJSON('/api/contact', function(contactsJsonPayload)
        {
            $(contactsJsonPayload).each(function(i, item)
            {
                $('#contacts').append('<li>' + item.Name + '</li>');
            });
        });
});
</script>
}
```

6. Open the **ContactController.cs** file if it is not already open.

7. Place a breakpoint on the **Get** method of the **ContactController** class.

```
public Contact[] Get()
{
    return this.contactRepository.GetAllContacts();
}
```

Placing a breakpoint on the Get method of the API controller

8. Press **F5** to run the project. The browser will load the HTML document.

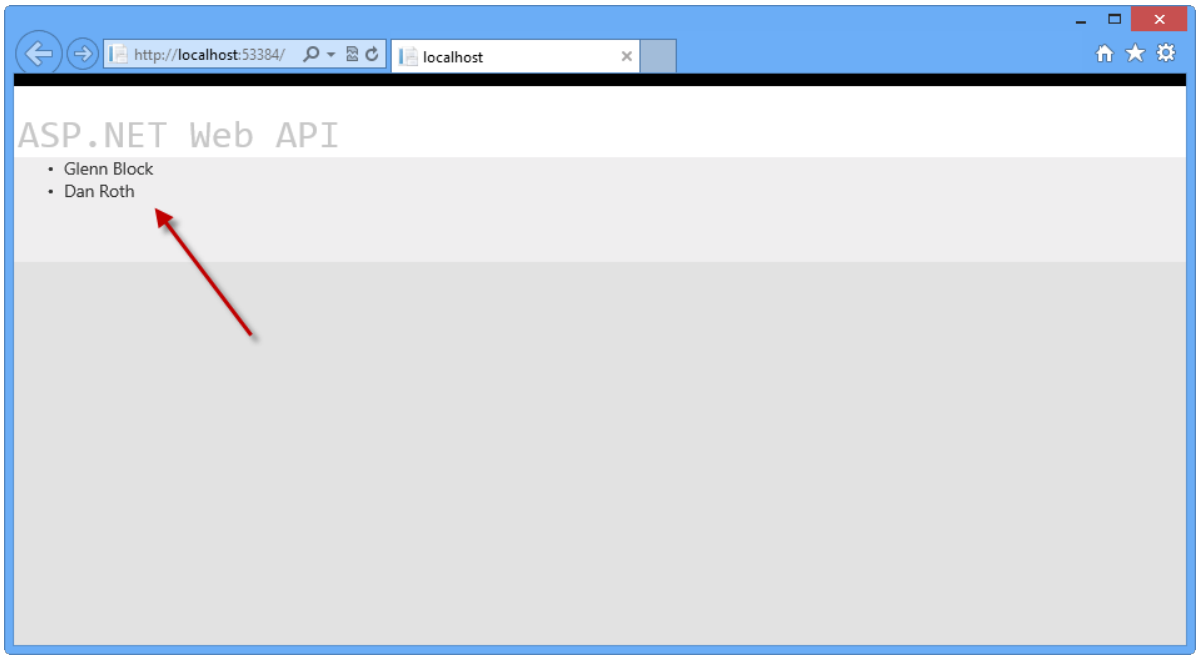
> **NOTE**
>
> Ensure that you are browsing to the root URL of your application.

9. Once the page loads and the JavaScript executes, the breakpoint will be hit and the code execution will pause in the controller.

```
public Contact[] Get()
{
    return this.contactRepository.GetAllContacts();
}
```

Debugging into the Web API call using Visual Studio 2012 Express for Web

10. Remove the breakpoint and press **F5** or the debugging toolbar's **Continue** button to continue loading the view in the browser. Once the Web API call completes you should see the contacts returned from the Web API call displayed as list items in the browser.



Results of the API call displayed in the browser as list items

11. Stop debugging.

Task 2 - Modifying the Index View to Provide a GUI for Creating Contacts

In this task, you will continue to modify the Index view of the MVC application. A form will be added to the HTML page that will capture user input and send it to the Web API to create a new Contact, and a new Web API controller method will be created to collect date from the GUI.

1. Open the **ContactController.cs** file.

2. Add a new method to the controller class named **Post** as shown in the following code.

   (Code Snippet - *Web API Lab - Ex03 - Post Method*)

```
public HttpResponseMessage Post(Contact contact)
{
    this.contactRepository.SaveContact(contact);

    var response = Request.CreateResponse<Contact>(System.Net.HttpStatusCode.Created, contact);

    return response;
}
```

3. Open the **Index.cshtml** file in Visual Studio if it is not already open.

4. Add the HTML code below to the file just after the unordered list you added in the previous task.

```
<form id="saveContactForm" method="post">
<h3>Create a new Contact</h3>
<p>
    <label for="contactId">Contact Id:</label>
    <input type="text" name="Id" />
</p>
<p>
    <label for="contactName">Contact Name:</label>
    <input type="text" name="Name" />
</p>
<input type="button" id="saveContact" value="Save" />
</form>
```

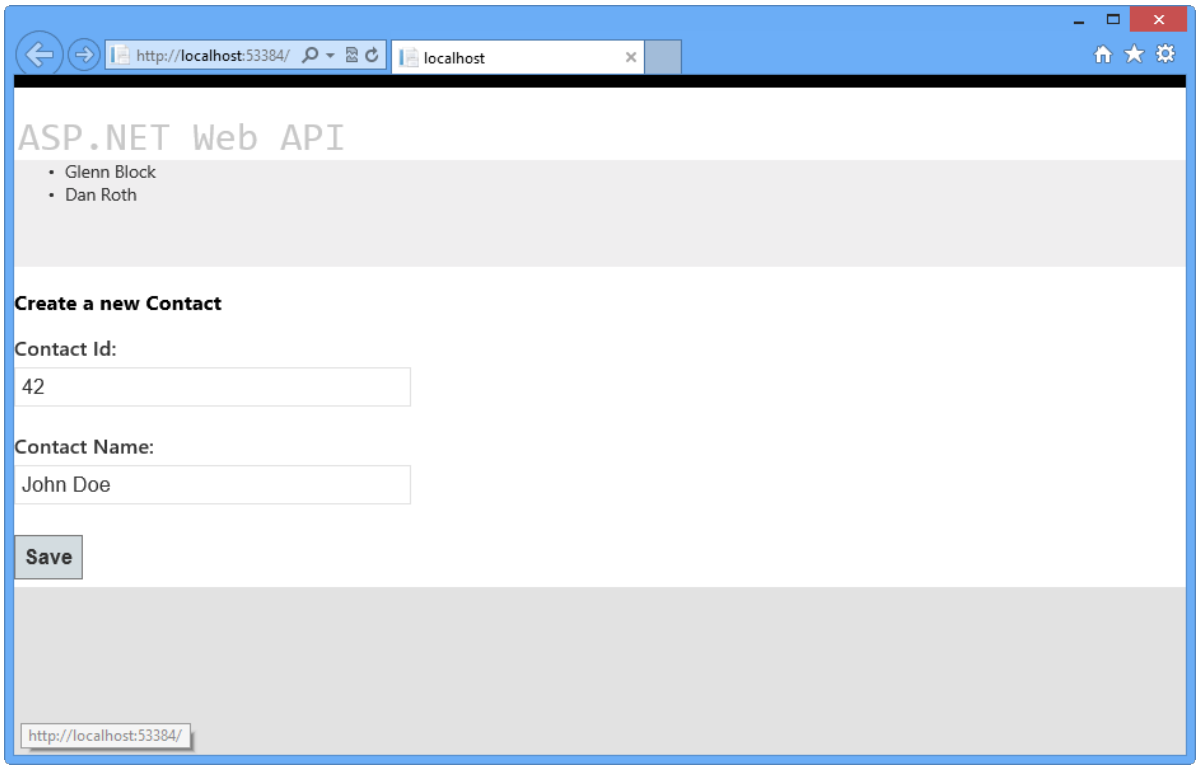
5. Within the script element at the bottom of the document, add the following highlighted code to handle button-click events, which will post the data to the Web API using an HTTP POST call.

```
<script type="text/javascript">
...
$('#saveContact').click(function()
        {
             $.post("api/contact",
                   $("#saveContactForm").serialize(),
                   function(value)
                   {
                        $('#contacts').append('<li>' + value.Name + '</li>');
                   },
                   "json"
             );
        });
</script>
```

6. In **ContactController.cs**, place a breakpoint on the **Post** method.

7. Press **F5** to run the application in the browser.

8. Once the page is loaded in the browser, type in a new contact name and Id and click the **Save** button.



The client HTML document loaded in the browser

9. When the debugger window breaks in the **Post** method, take a look at the properties of the **contact** parameter. The values should match the data you entered in the form.

```
public HttpResponseMessage Post(Contact contact)
{
    this.contactRepository.SaveContact(contact);

    var response = Request.CreateResponse<Contact>(System.Net.HttpStatusCode.Created, contact);

    return response;
}
}
}
```

100 %

Locals

Name	Value
this	{ContactManager.Controllers.ContactController}
contact	{ContactManager.Models.Contact}
Id	42
Name	"John Doe"
response	null

The Contact object being sent to the Web API from the client

10. Step through the method in the debugger until the **response** variable has been created. Upon inspection in the **Locals** window in the debugger, you'll see that all the properties have been set.

```
public HttpResponseMessage Post(Contact contact)
{
    this.contactRepository.SaveContact(contact);

    var response = Request.CreateResponse<Contact>(System.Net.HttpStatusCode.Created, contact);

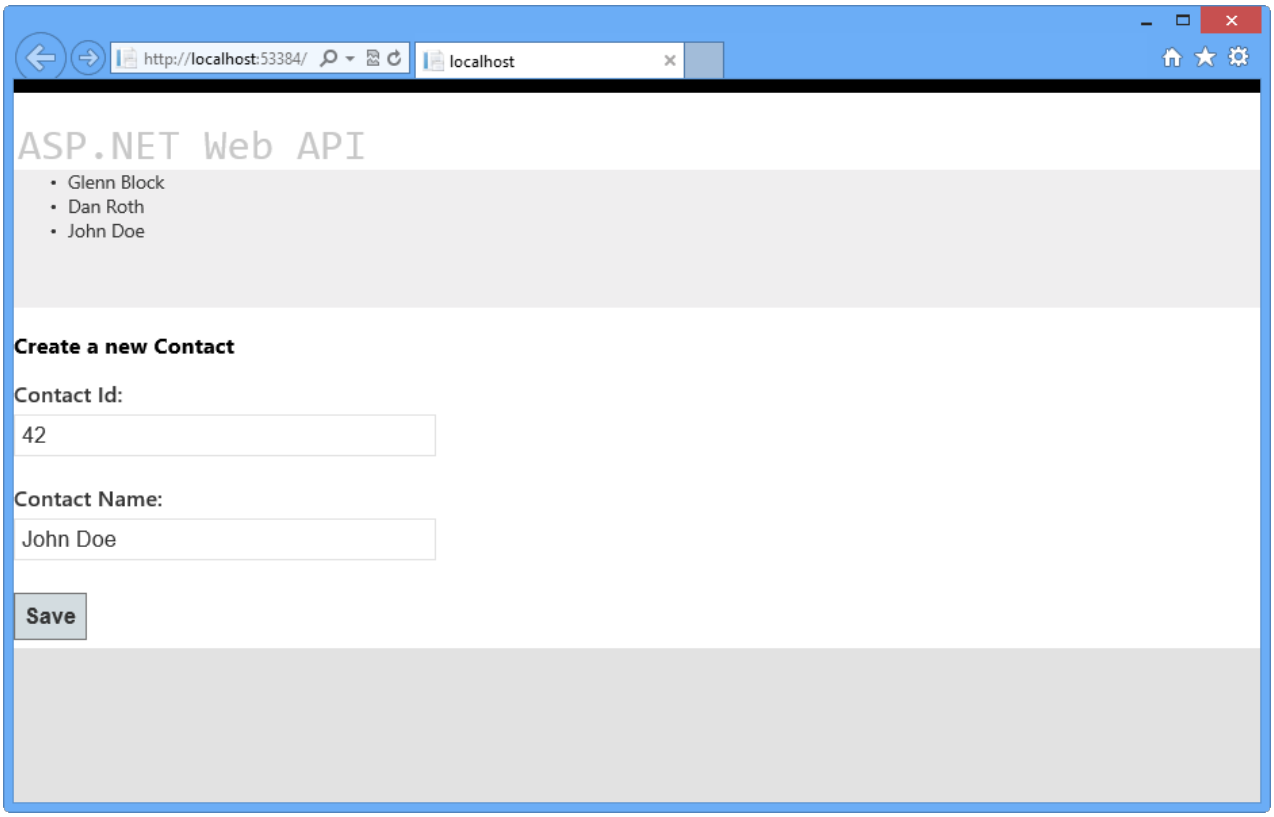
    return response;
}
}
}
```

100 %

Locals

Name	Value
contact	{ContactManager.Models.Contact}
Id	42
Name	"John Doe"
response	{StatusCode: 201, ReasonPhrase: 'Created', Version: 1.1, Content: System.Net.Http.ObjectContent`1[[ContactManager.Models.Contact, Co
Content	{System.Net.Http.ObjectContent<ContactManager.Models.Contact>}
Headers	{}
IsSuccessStatusCode	true
ReasonPhrase	"Created"
RequestMessage	{Method: POST, RequestUri: 'http://localhost:53384/api/contact', Version: 1.1, Content: System.Net.Http.StreamContent, Headers:{ Cache
StatusCode	Created
Version	{1.1}

The response following creation in the debugger 11. If you press **F5** or click **Continue** in the debugger the request will complete. Once you switch back to the browser, the new contact has been added to the list of contacts stored by the **ContactRepository** implementation.



The browser reflects successful creation of the new contact instance

> **NOTE**
>
> Additionally, you can deploy this application to Azure following Appendix C: Publishing an ASP.NET MVC 4 Application using Web Deploy.

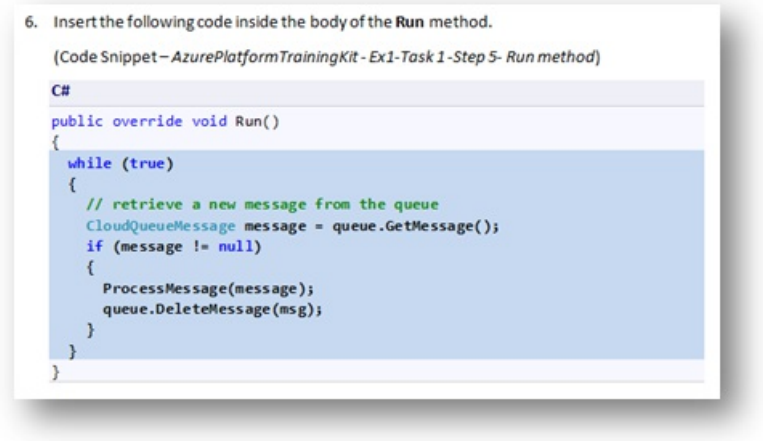
Summary

This lab has introduced you to the new ASP.NET Web API framework and to the implementation of RESTful Web APIs using the framework. From here, you could create a new repository that facilitates data persistence using any number of mechanisms and wire that service up rather than the simple one provided as an example in this lab. Web API supports a number of additional features, such as enabling communication from non-HTML clients written in any language that supports HTTP and JSON or XML. The ability to host a Web API outside of a typical web application is also possible, as well as is the ability to create your own serialization formats.

The ASP.NET Web site has an area dedicated to the ASP.NET Web API framework at [https://asp.net/web-api] (https://asp.net/web-api). This site will continue to provide late-breaking information, samples, and news related to Web API, so check it frequently if you'd like to delve deeper into the art of creating custom Web APIs available to virtually any device or development framework.

Appendix A: Using Code Snippets

With code snippets, you have all the code you need at your fingertips. The lab document will tell you exactly when you can use them, as shown in the following figure.



6. Insert the following code inside the body of the **Run** method.

(Code Snippet – *AzurePlatformTrainingKit - Ex1-Task 1 -Step 5- Run method*)

C#

```
public override void Run()
{
  while (true)
  {
    // retrieve a new message from the queue
    CloudQueueMessage message = queue.GetMessage();
    if (message != null)
    {
      ProcessMessage(message);
      queue.DeleteMessage(msg);
    }
  }
}
```

Using Visual Studio code snippets to insert code into your project

To add a code snippet using the keyboard (C# only)

1. Place the cursor where you would like to insert the code.
2. Start typing the snippet name (without spaces or hyphens).
3. Watch as IntelliSense displays matching snippets' names.
4. Select the correct snippet (or keep typing until the entire snippet's name is selected).
5. Press the Tab key twice to insert the snippet at the cursor location.

```
static void Main(string[] args)
{
    Azure
}
```

AzureTrainingKitAsyncWorkloadHandling
AzureTrainingKitIntro
AzureTrainingKitRoleCommunication
AzureTrainingKitServiceBus

Start typing the snippet name

```
static void Main(string[] args)
{
    AzureTrainingKitServiceBus
}
```

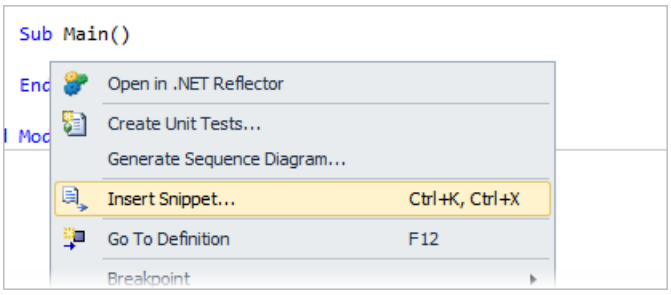
Press Tab to select the highlighted snippet

```
static void Main(string[] args)
{
    request.Headers.Add("$accountKey", accountKey);
    request.Headers.Add("$uniqueUserID", uniqueUserId);
}
```

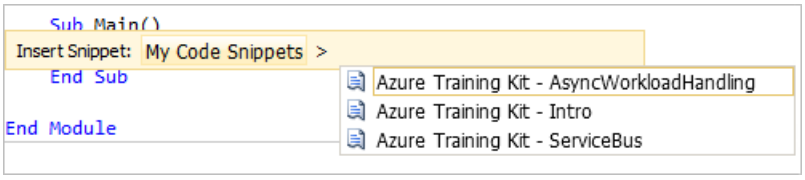
Press Tab again and the snippet will expand

To add a code snippet using the mouse (C#, Visual Basic and XML)

1. Right-click where you want to insert the code snippet.
2. Select **Insert Snippet** followed by **My Code Snippets**.
3. Pick the relevant snippet from the list, by clicking on it.



Right-click where you want to insert the code snippet and select Insert Snippet



Pick the relevant snippet from the list, by clicking on it

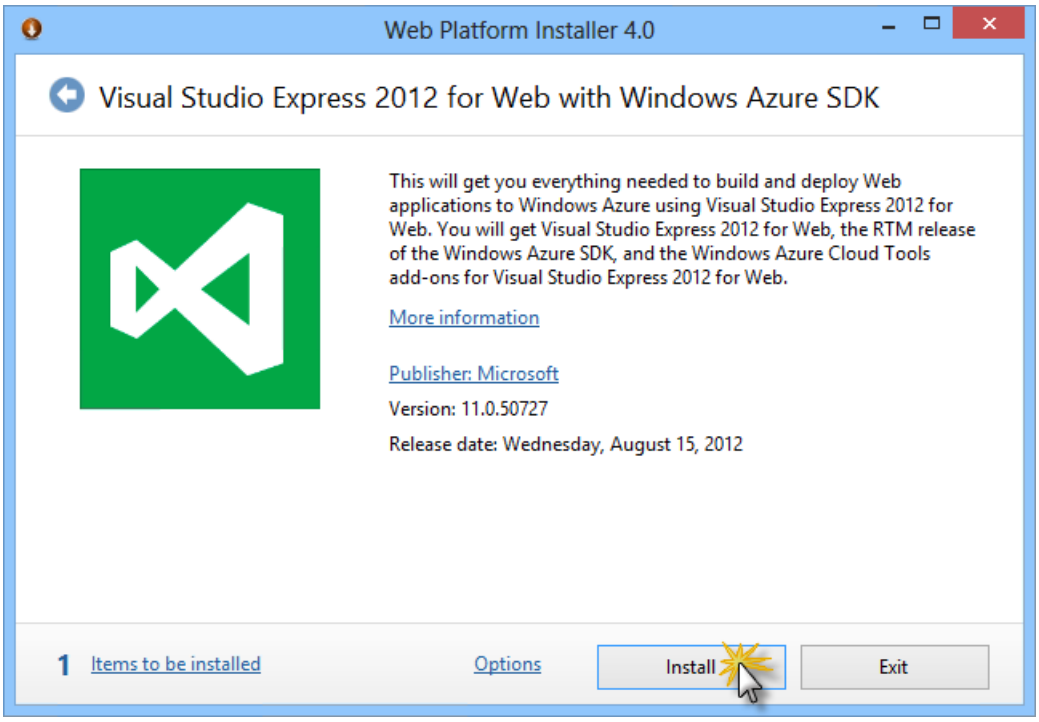
Appendix B: Installing Visual Studio Express 2012 for Web

You can install **Microsoft Visual Studio Express 2012 for Web** or another "Express" version using the **Microsoft Web Platform Installer**. The following instructions guide you through the steps required to install *Visual studio Express 2012 for Web* using *Microsoft Web Platform Installer*.

1. Go to https://go.microsoft.com/?linkid=9810169. Alternatively, if you already have installed Web Platform Installer, you can open it and search for the product "*Visual Studio Express 2012 for Web with Azure SDK*".

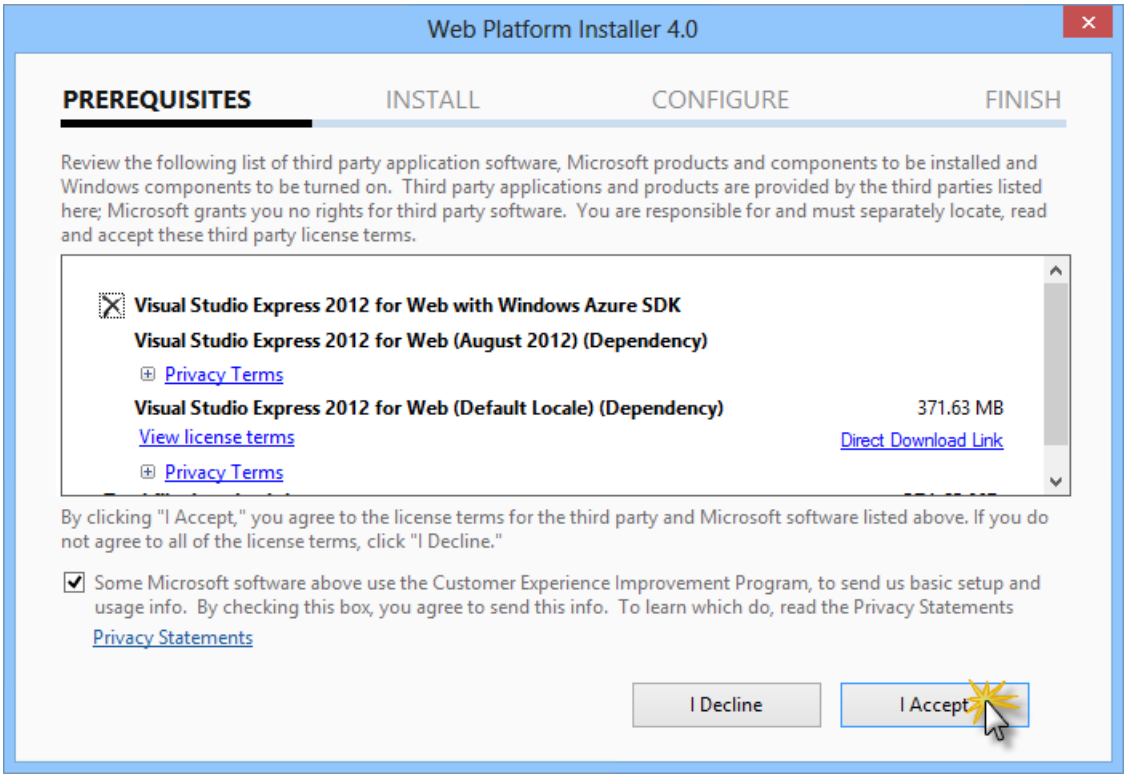
2. Click on **Install Now**. If you do not have **Web Platform Installer** you will be redirected to download and install it first.

3. Once **Web Platform Installer** is open, click **Install** to start the setup.



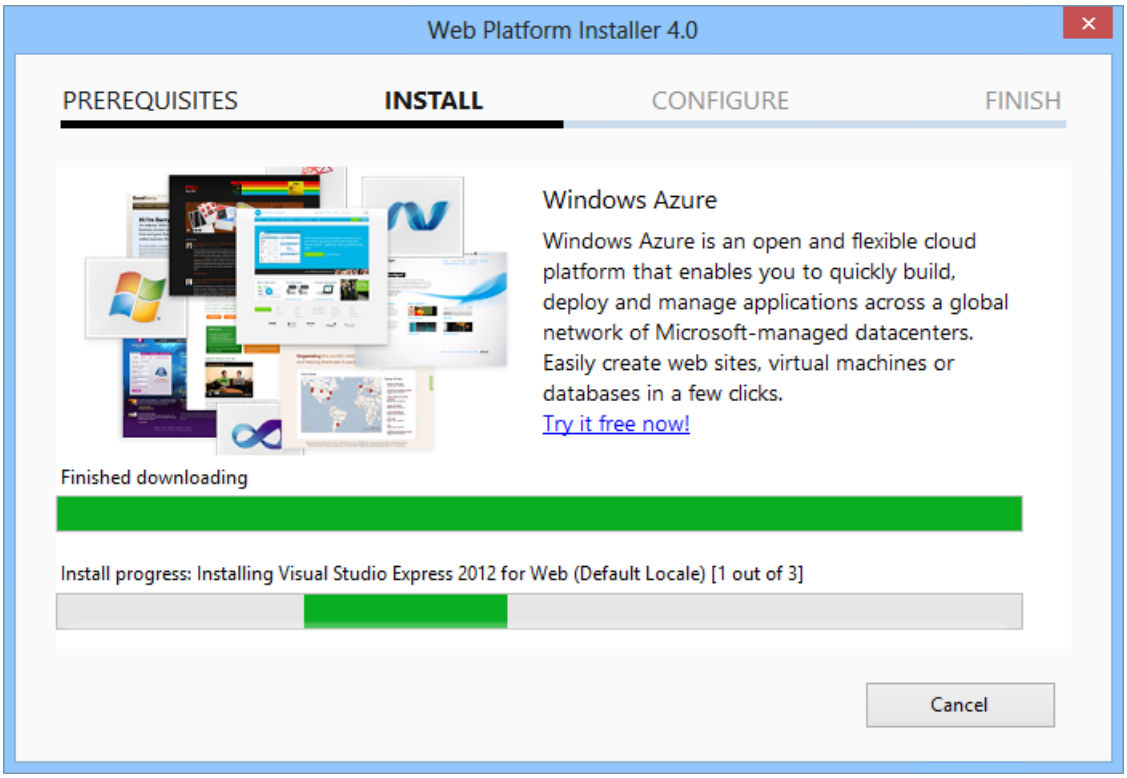
Install Visual Studio Express

4. Read all the products' licenses and terms and click **I Accept** to continue.



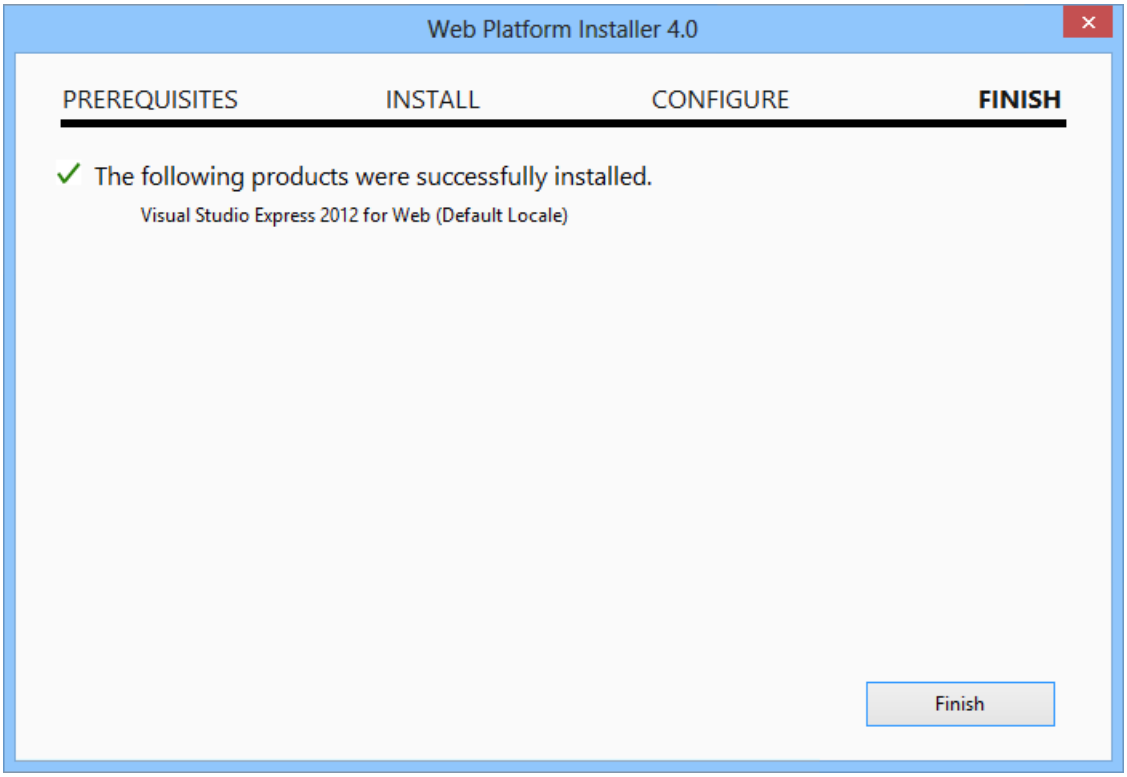
Accepting the license terms

5. Wait until the downloading and installation process completes.



Installation progress

6. When the installation completes, click **Finish**.



Installation completed

7. Click **Exit** to close Web Platform Installer.

8. To open Visual Studio Express for Web, go to the **Start** screen and start writing "**VS Express**", then click on the **VS Express for Web** tile.

VS Express for Web tile

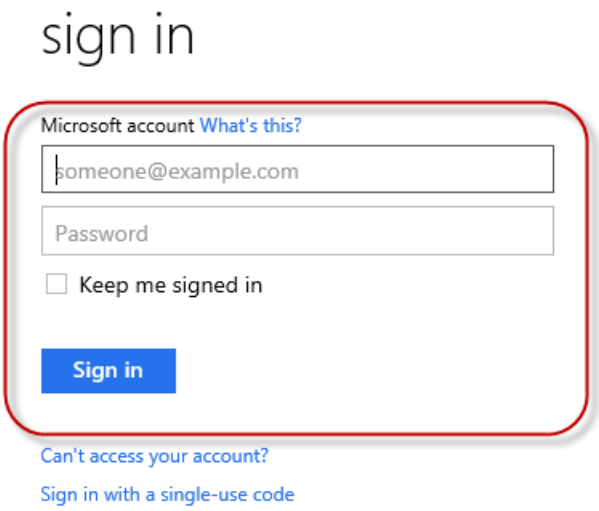
Appendix C: Publishing an ASP.NET MVC 4 Application using Web Deploy

This appendix will show you how to create a new web site from the Azure Portal and publish the application you obtained by following the lab, taking advantage of the Web Deploy publishing feature provided by Azure.

Task 1 - Creating a New Web Site from the Azure Portal

1. Go to the Azure Management Portal and sign in using the Microsoft credentials associated with your subscription.

> **NOTE**
> With Azure you can host 10 ASP.NET Web Sites for free and then scale as your traffic grows. You can sign up here.



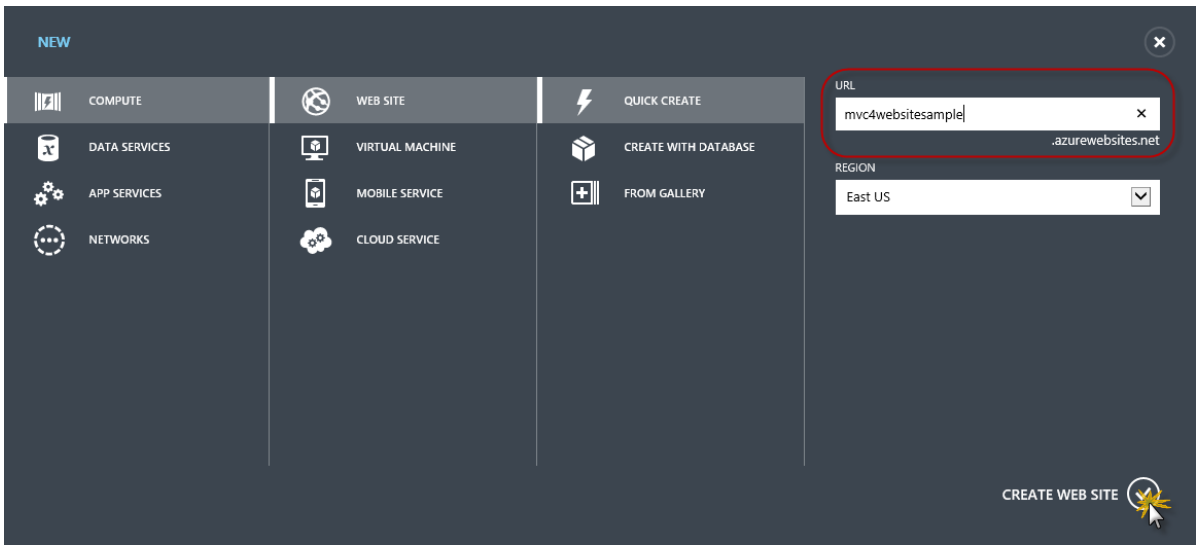
Log on to Portal

2. Click **New** on the command bar.

Creating a new Web Site

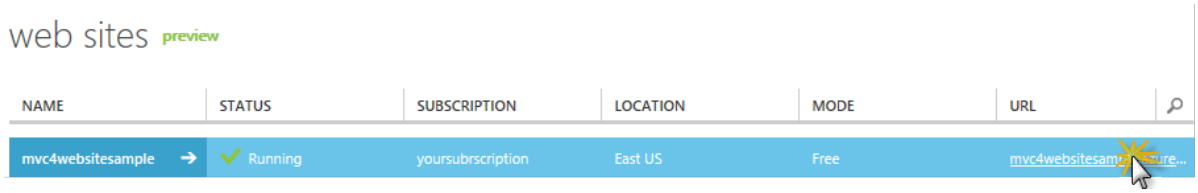
3. Click **Compute | Web Site**. Then select **Quick Create** option. Provide an available URL for the new web site and click **Create Web Site**.

> **NOTE**
>
> Azure is the host for a web application running in the cloud that you can control and manage. The Quick Create option allows you to deploy a completed web application to the Azure from outside the portal. It does not include steps for setting up a database.

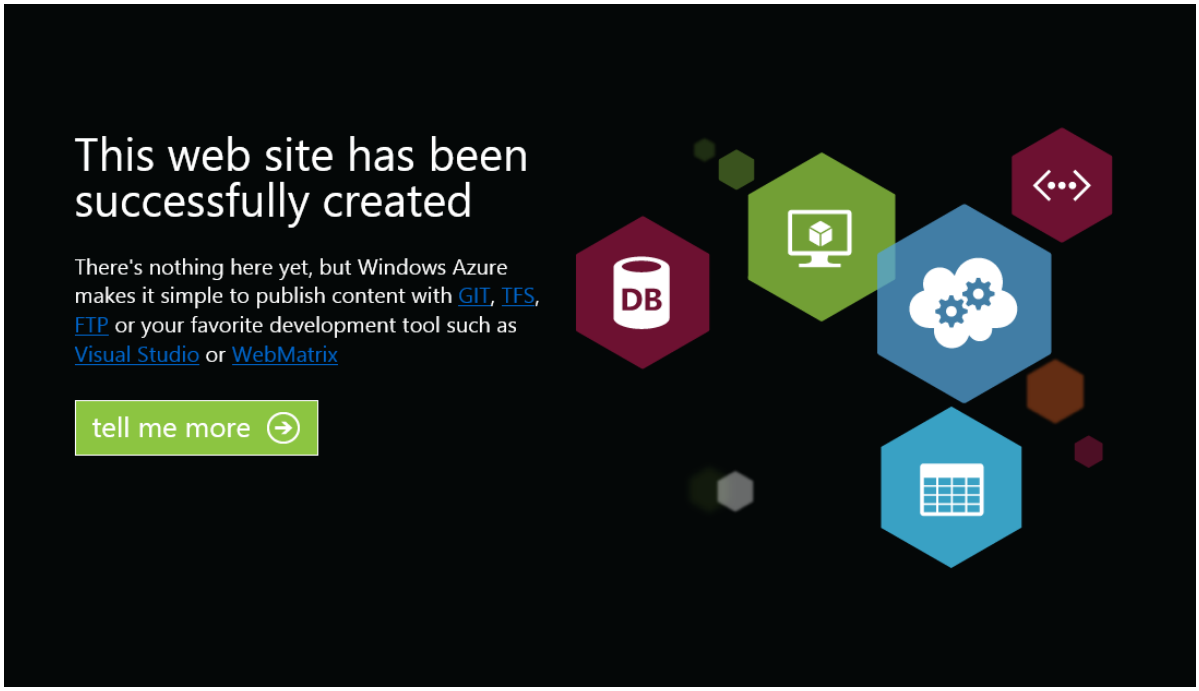


Creating a new Web Site using Quick Create

4. Wait until the new **Web Site** is created.
5. Once the Web Site is created click the link under the **URL** column. Check that the new Web Site is working.

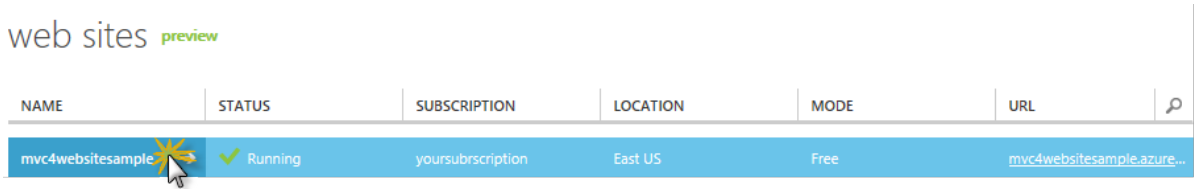


Browsing to the new web site



Web site running

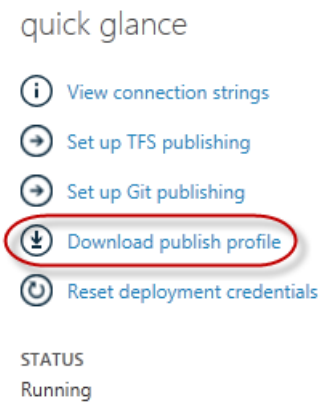
6. Go back to the portal and click the name of the web site under the **Name** column to display the management pages.



Opening the Web Site management pages

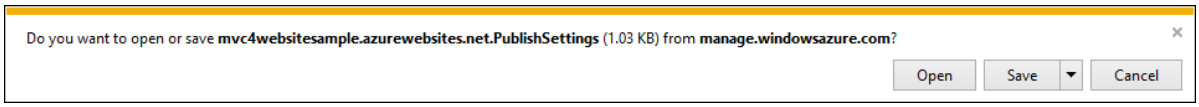
7. In the **Dashboard** page, under the **quick glance** section, click the **Download publish profile** link.

> **NOTE**
>
> The *publish profile* contains all of the information required to publish a web application to a Azure for each enabled publication method. The publish profile contains the URLs, user credentials and database strings required to connect to and authenticate against each of the endpoints for which a publication method is enabled. **Microsoft WebMatrix 2**, **Microsoft Visual Studio Express for Web** and **Microsoft Visual Studio 2012** support reading publish profiles to automate configuration of these programs for publishing web applications to Azure.



Downloading the Web Site publish profile

8. Download the publish profile file to a known location. Further in this exercise you will see how to use this file to publish a web application to Azure from Visual Studio.

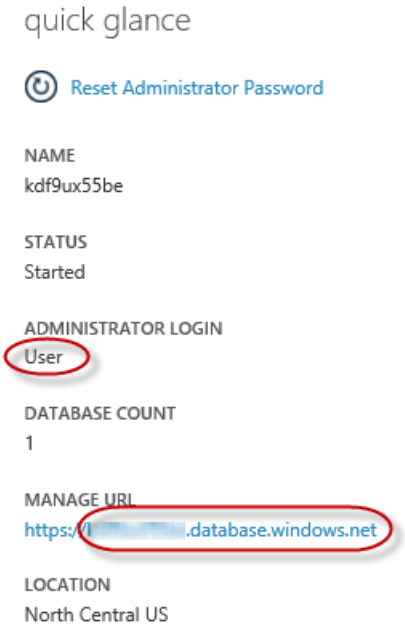


Saving the publish profile file

Task 2 - Configuring the Database Server

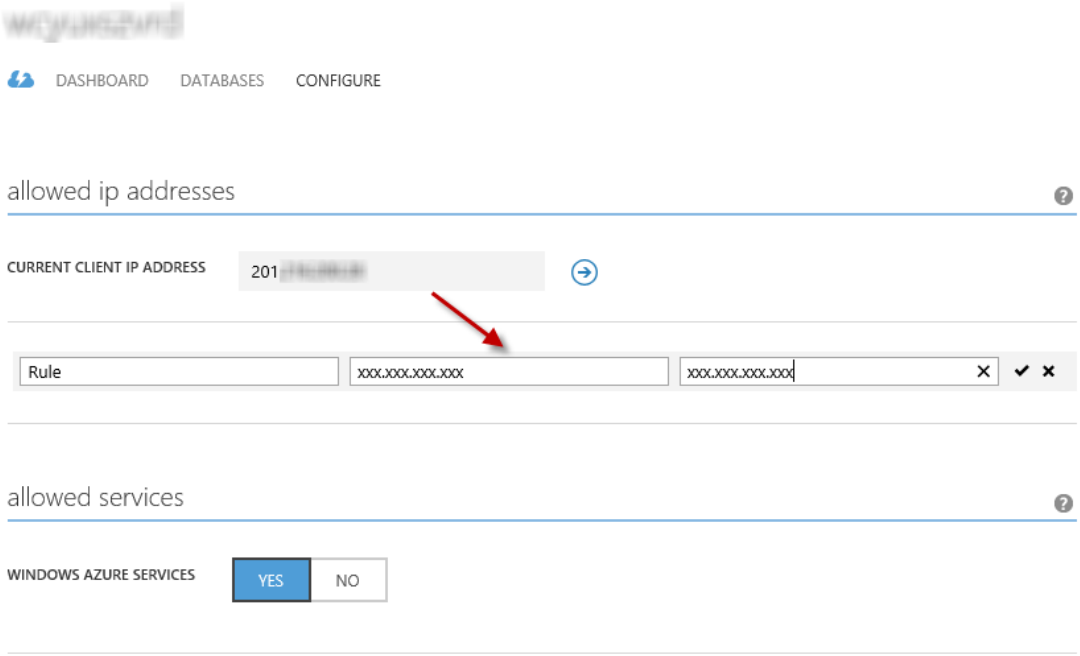
If your application makes use of SQL Server databases you will need to create a SQL Database server. If you want to deploy a simple application that does not use SQL Server you might skip this task.

1. You will need a SQL Database server for storing the application database. You can view the SQL Database servers from your subscription in the Azure Management portal at **Sql Databases | Servers | Server's Dashboard**. If you do not have a server created, you can create one using the **Add** button on the command bar. Take note of the **server name and URL, administrator login name and password**, as you will use them in the next tasks. Do not create the database yet, as it will be created in a later stage.



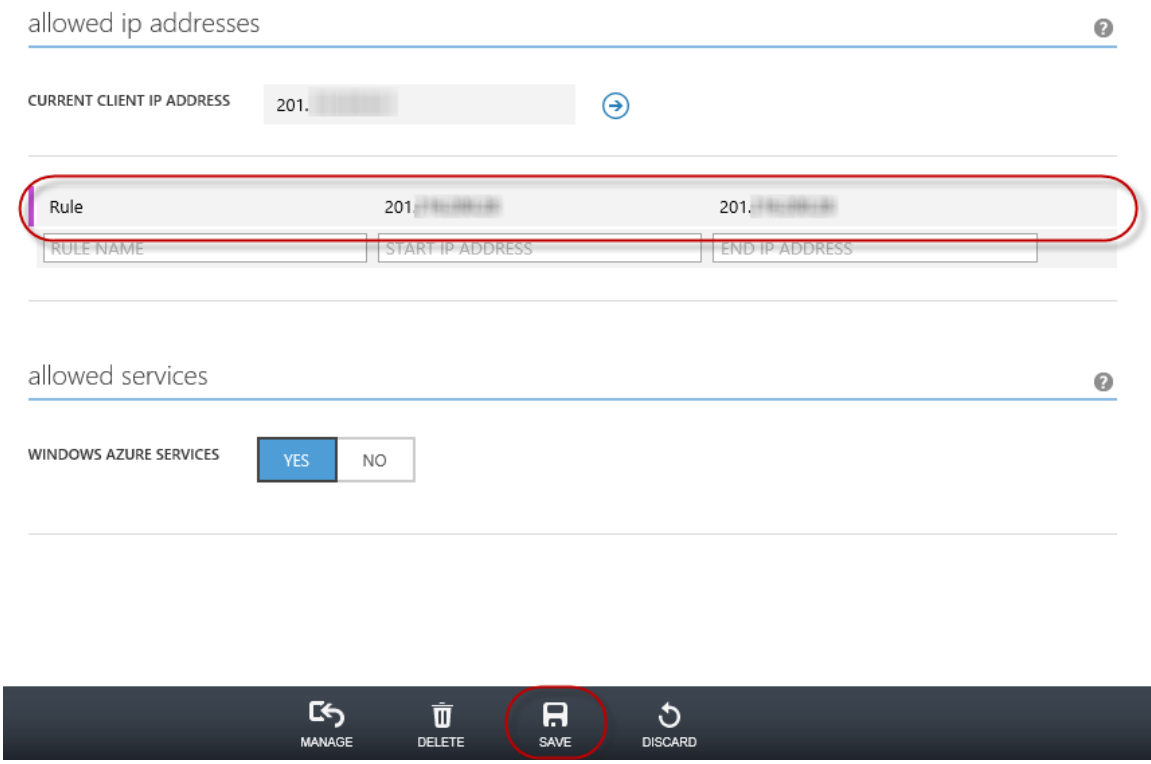
SQL Database Server Dashboard

2. In the next task you will test the database connection from Visual Studio, for that reason you need to include your local IP address in the server's list of **Allowed IP Addresses**. To do that, click **Configure**, select the IP address from **Current Client IP Address** and paste it on the **Start IP Address** and **End IP Address** text boxes and click the ✔ button.



Adding Client IP Address

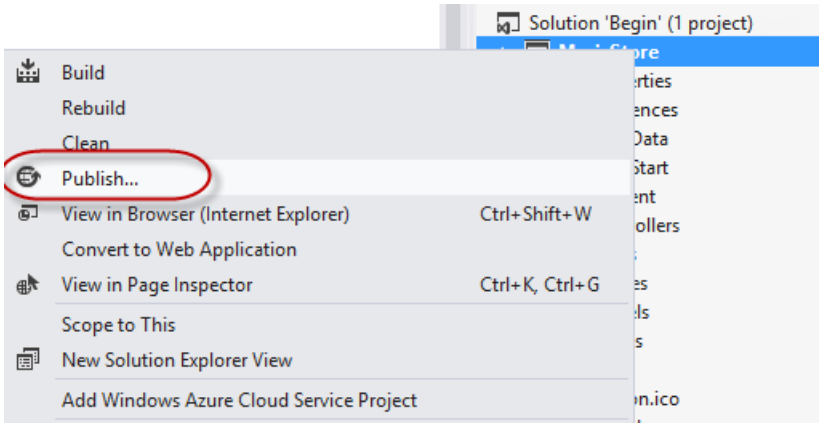
3. Once the **Client IP Address** is added to the allowed IP addresses list, click on **Save** to confirm the changes.



Confirm Changes

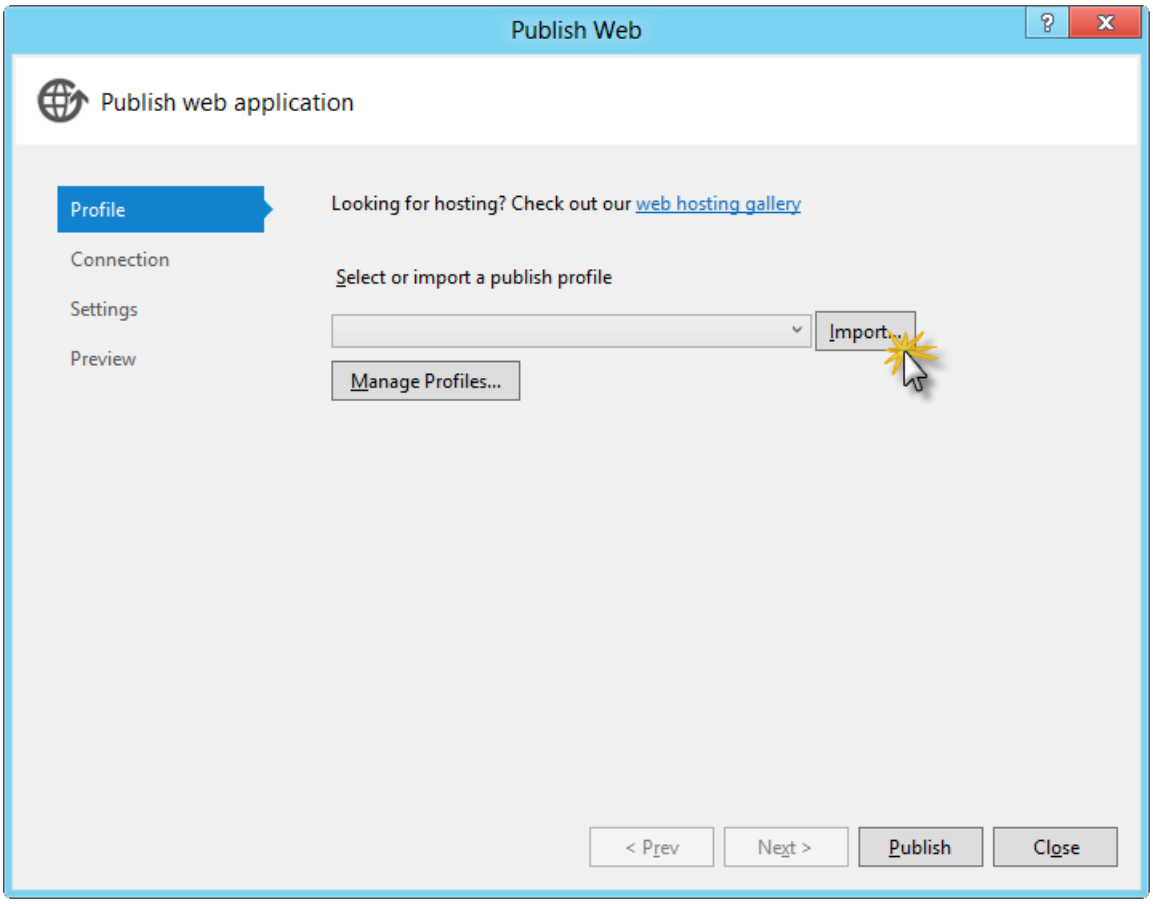
Task 3 - Publishing an ASP.NET MVC 4 Application using Web Deploy

1. Go back to the ASP.NET MVC 4 solution. In the **Solution Explorer**, right-click the web site project and select **Publish**.



Publishing the web site

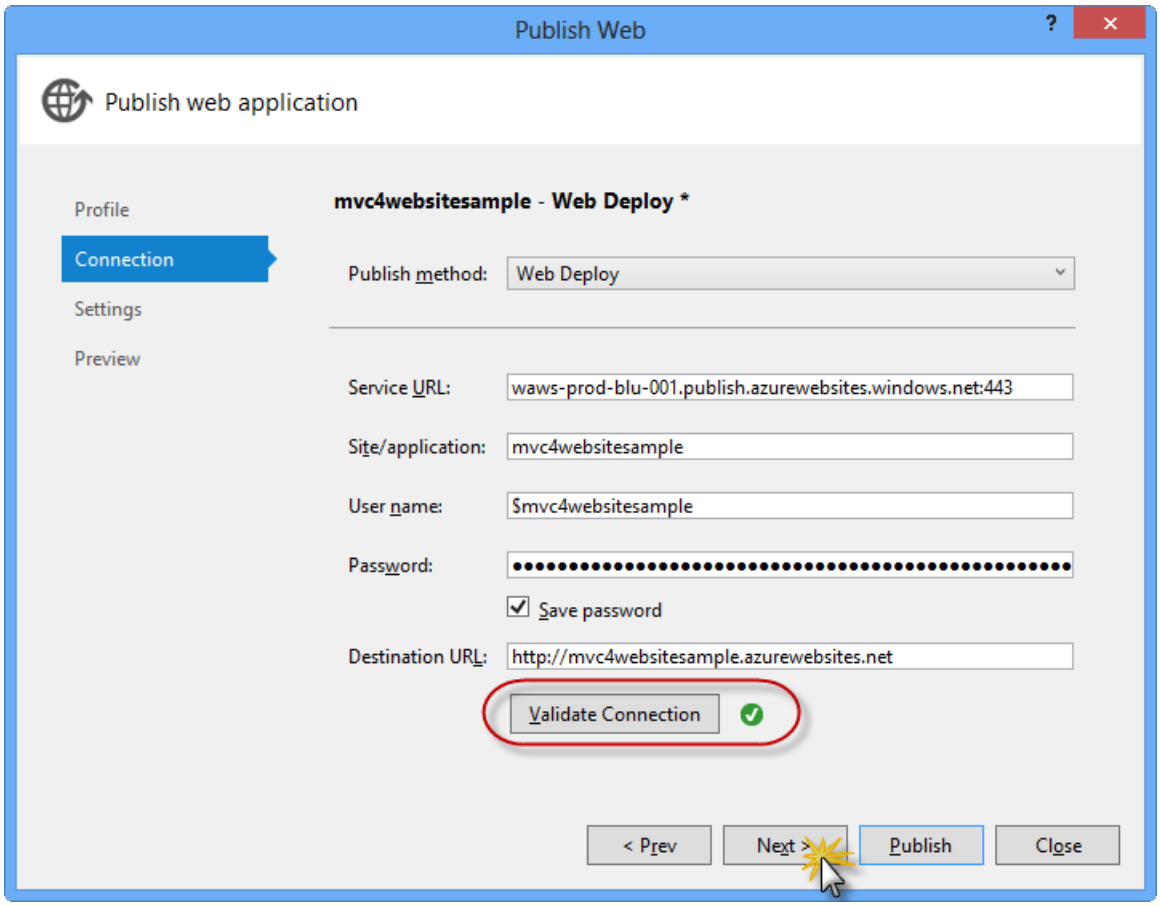
2. Import the publish profile you saved in the first task.



Importing publish profile

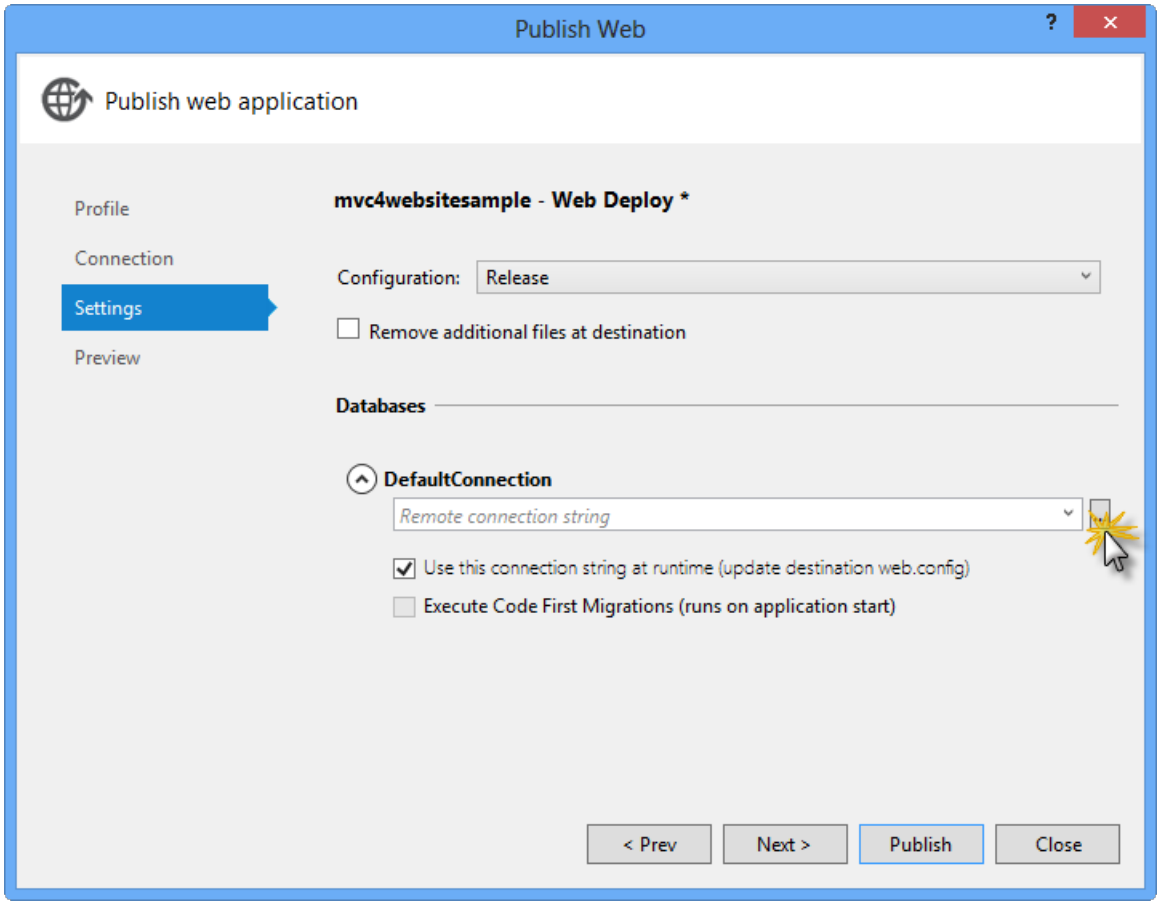
3. Click **Validate Connection**. Once Validation is complete click **Next**.

> **NOTE**
>
> Validation is complete once you see a green checkmark appear next to the Validate Connection button.



Validating connection

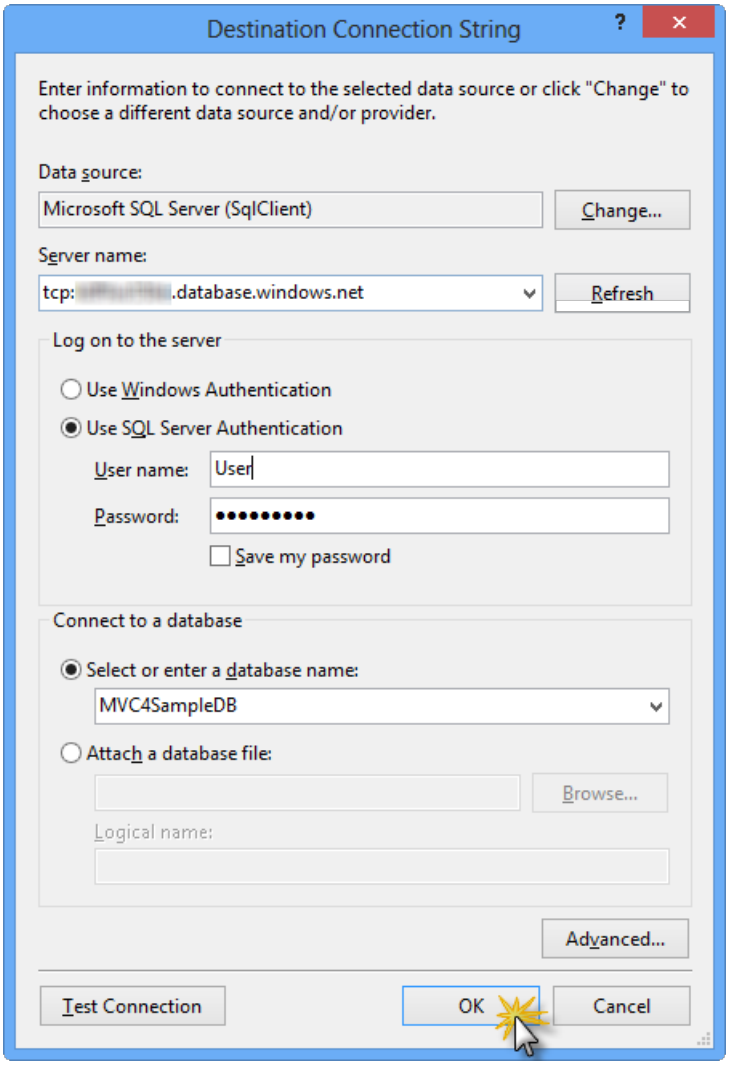
4. In the **Settings** page, under the **Databases** section, click the button next to your database connection's textbox (i.e. **DefaultConnection**).



Web deploy configuration

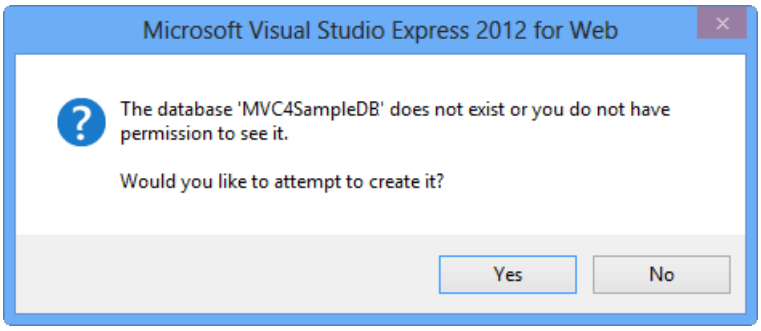
5. Configure the database connection as follows:

   - In the **Server name** type your SQL Database server URL using the *tcp:* prefix.
   - In **User name** type your server administrator login name.
   - In **Password** type your server administrator login password.
   - Type a new database name, for example: *MVC4SampleDB*.



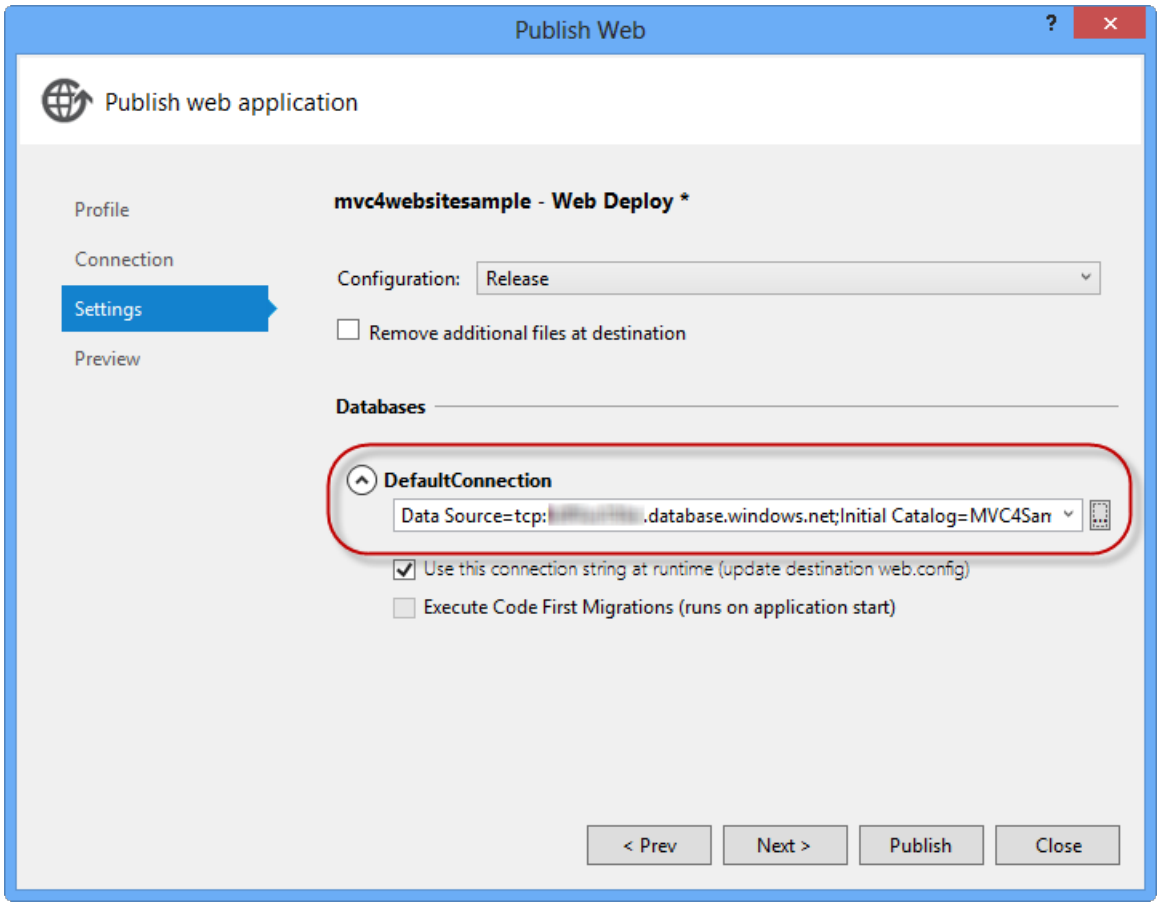
Configuring destination connection string

6. Then click **OK**. When prompted to create the database click **Yes**.



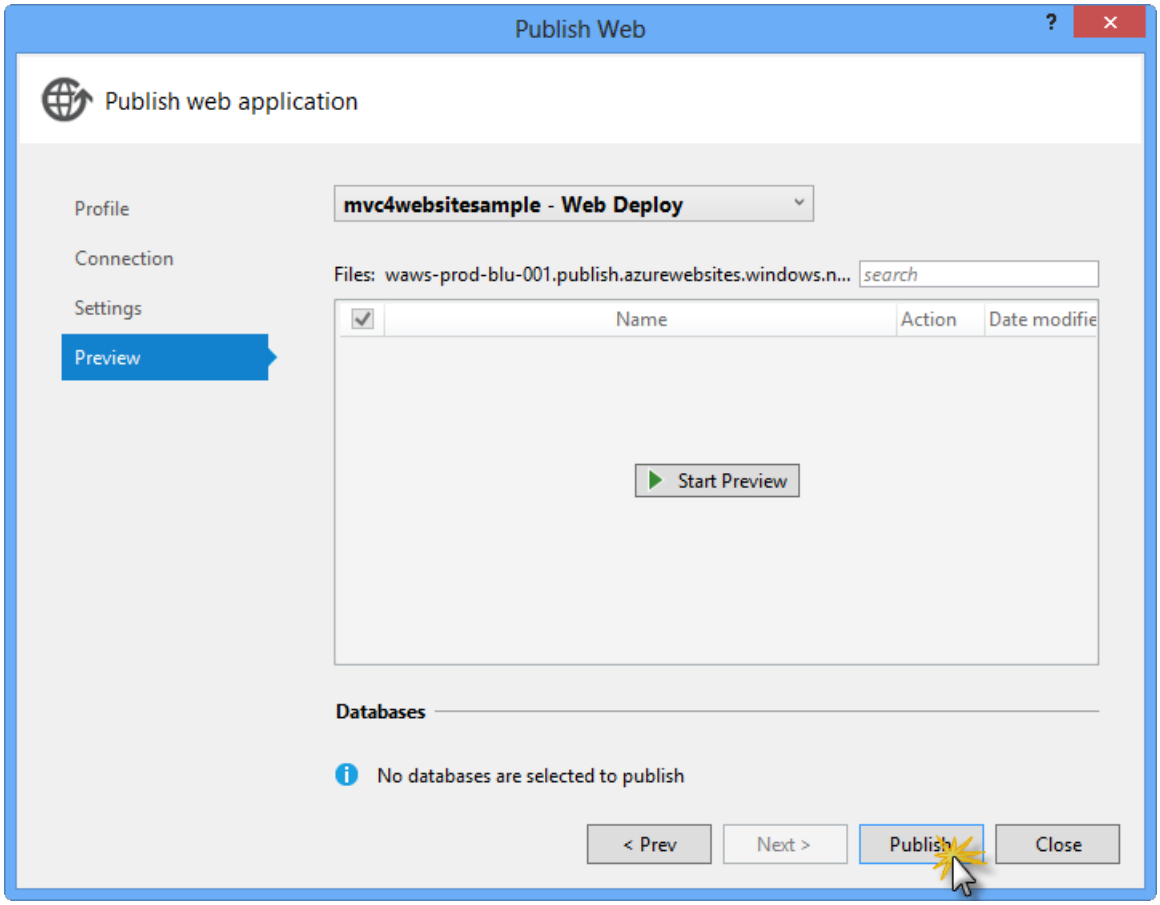
Creating the database

7. The connection string you will use to connect to SQL Database in Windows Azure is shown within Default Connection textbox. Then click **Next**.



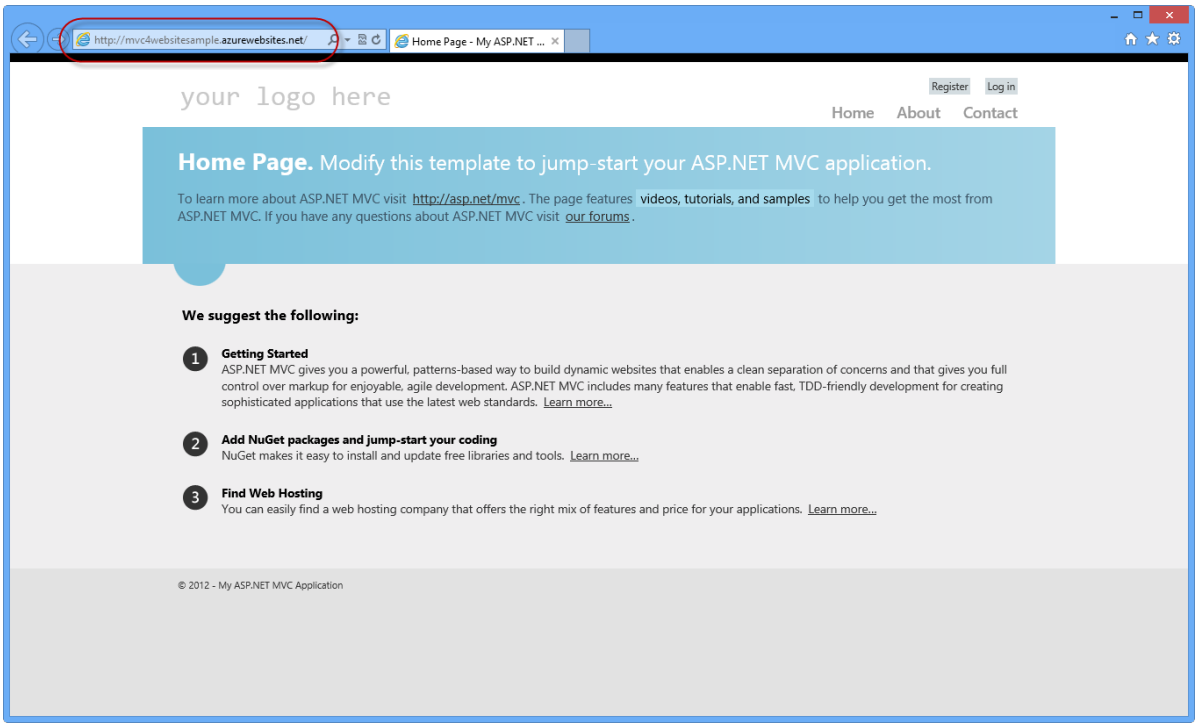
Connection string pointing to SQL Database

8. In the **Preview** page, click **Publish**.



Publishing the web application

9. Once the publishing process finishes, your default browser will open the published web site.



Application published to Azure

ASP.NET Web API Videos

2/20/2020 • 2 minutes to read • Edit Online

- Overview and Getting Started
  - ASP.NET Web API
  - ASP.NET Web API, Part 1: Your First Web API
  - ASP.NET Web API, Part 2: Getting Data
  - ASP.NET Web API, Part 3: Delete and Update
  - ASP.NET Web API, Part 4: Paging and Querying
  - ASP.NET Web API, Part 5: Custom Validation
  - ASP.NET Web API, Part 6: Authorization

ASP.NET Web API Overview and Getting Started Videos

2/20/2020 • 2 minutes to read • Edit Online

This is a series of videos to get you started with ASP.NET Web API.

- ASP.NET Web API
- ASP.NET Web API, Part 1: Your First Web API
- ASP.NET Web API, Part 2: Getting Data
- ASP.NET Web API, Part 3: Delete and Update
- ASP.NET Web API, Part 4: Paging and Querying
- ASP.NET Web API, Part 5: Custom Validation
- ASP.NET Web API, Part 6: Authorization

ASP.NET Web API

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Watch video (6 minutes)

ASP.NET Web API, Part 1: Your First Web API

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

NOTE

This video contains information from a pre-release version of ASP.NET Web API. For information on what changed, see the updated sample code at https://code.msdn.microsoft.com/ASPNET-Web-API-JavaScript-d0d64dd7

Watch video (3 minutes)

ASP.NET Web API, Part 2: Getting Data

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

NOTE

This video contains information from a pre-release version of ASP.NET Web API. For information on what changed, see the updated sample code at https://code.msdn.microsoft.com/ASPNET-Web-API-JavaScript-d0d64dd7

Watch video (4 minutes)

ASP.NET Web API, Part 3: Delete and Update

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

NOTE

This video contains information from a pre-release version of ASP.NET Web API. For information on what changed, see the updated sample code at https://code.msdn.microsoft.com/ASPNET-Web-API-JavaScript-d0d64dd7

Watch video (5 minutes)

ASP.NET Web API, Part 4: Paging and Querying

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

NOTE

This video contains information from a pre-release version of ASP.NET Web API. For information on what changed, see the updated sample code at https://code.msdn.microsoft.com/ASPNET-Web-API-JavaScript-d0d64dd7

Watch video (4 minutes)

ASP.NET Web API, Part 5: Custom Validation

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

NOTE

This video contains information from a pre-release version of ASP.NET Web API. For information on what changed, see the updated sample code at https://code.msdn.microsoft.com/ASPNET-Web-API-JavaScript-d0d64dd7

Watch video (3 minutes)

ASP.NET Web API, Part 6: Authorization

2/20/2020 • 2 minutes to read • Edit Online

by Jon Galloway

NOTE

This video contains information from a pre-release version of ASP.NET Web API. For information on what changed, see the updated sample code at https://code.msdn.microsoft.com/ASPNET-Web-API-JavaScript-d0d64dd7

Watch video (2 minutes)

Web API Samples List

2/20/2020 • 6 minutes to read • Edit Online

HttpClient Samples

Bing Translate Sample | VS 2012 source

Shows how to call the Microsoft Translator service using the **HttpClient** class. The Microsoft Translator service API requires an OAuth token, which the application obtains by sending a request to the Azure token server for each request to the translator service. The result from the token server is fed into the request sent to the translation service. Before running this sample, you must obtain an application key from Azure Marketplace and fill in the information in the AccessTokenMessageHandler sample class.

Google Maps Sample | detailed description | VS 2012 source

Uses **HttpClient** to download a map of Redmond, WA from Google Maps API, saves it as a local file, and opens the default image viewer.

Twitter Client Sample | detailed description | VS 2012 source

Shows how to write a simple Twitter client using **HttpClient**. The sample uses an **HttpMessageHandler** to insert OAuth authentication information into the outgoing **HttpRequestMessage**. The result from Twitter is read using JSON.NET. Before running this sample, you must obtain an application key from Twitter, and fill in the information in the OAuthMessageHandler sample class.

World Bank Sample | detailed description | VS 2010 source | VS 2012 source

Shows how to retrieve data from the World Bank data site, using JSON.NET to parse the result.

Web API Samples

Getting Started with ASP.NET Web API | VS 2012 source

Shows how to create a basic web API that supports HTTP GET requests. Contains the source code for the tutorial Your First ASP.NET Web API.

ASP.NET Web API JavaScript Scenarios – Comments | VS 2012 source

Shows how to use ASP.NET Web API to build web APIs that support browser clients and can be easily called using jQuery.

Contact Manager | VS 2010 source

This sample uses ASP.NET Web API to build a simple contact manager application. The application consists of a contact manager web API that is used by an ASP.NET MVC application and a Windows Phone application to display and manage a list of contacts.

Batching Sample | detailed description | VS 2012 source

Shows how to implement HTTP batching within ASP.NET. The batching consists of putting multiple HTTP requests within a single MIME multipart entity body, which is then sent to the server as an HTTP POST. The requests are processed individually, and the responses are put into another MIME multipart entity body, which is returned to the client.

Content Controller Sample | detailed description | VS 2010 source | VS 2012 source

Shows how to read and write request and response entities asynchronously using streams. The sample controller has two actions: a PUT action that reads the request entity body asynchronously and stores it in a local file, and a GET action that returns the contents of the local file.

Custom Assembly Resolver Sample | VS 2012 source

Shows how to modify ASP.NET Web API to support discovery of controllers from a dynamically loaded library assembly. The sample implements a custom **IAssembliesResolver** which calls the default implementation and then adds the library assembly to the default results.

Custom Media Type Formatter Sample | detailed description | VS 2010 source

Shows how to create a custom media type formatter using the **BufferedMediaTypeFormatter** base class. This base class is intended for formatters which primarily use synchronous read and write operations. In addition to showing the media type formatter, the sample shows how to hook it up by registering it as part of the **HttpConfiguration** for your application. Note that it is also possible to use the **MediaTypeFormatter** base class directly, for formatters which primarily use asynchronous read and write operations.

Custom Parameter Binding Sample | detailed description | VS 2010 source

Shows how to customize the parameter binding process, which is the process that determines how information from a request is bound to action parameters. In this sample, the Home controller has four actions:

1. BindPrincipal shows how to bind an IPrincipal parameter from a custom generic principal, not from an HTTP GET message;
2. BindCustomComplexTypeFromUriOrBody shows how to bind a complex-type parameter, which could come either from the message body or from the request URI of an HTTP POST message;
3. BindCustomComplexTypeFromUriWithRenamedProperty shows how to bind a complex-type parameter with a renamed property which comes from the request URI of an HTTP POST message;
4. PostMultipleParametersFromBody shows how to bind multiple parameters from the body for a POST message;

File Upload Sample | detailed description | VS 2012 source

Shows how to upload files to an **ApiController** using MIME Multipart File Upload, and how to set up progress notifications with **HttpClient** using **ProgressNotificationHandler**. The controller reads the contents of an HTML file upload asynchronously and writes one or more body parts to a local file. The response contains information about the uploaded file (or files).

File Upload to Azure Blob Store Sample | detailed description | VS 2012 source

This sample is similar to the File Upload Sample, but instead of saving the uploaded files on local disk, it asynchronously uploads the files to Azure Blob Store using Windows Azure SDK for .NET. It also provides a mechanism for listing the blobs currently present in an Azure Blob Storage Container. You can try out the sample running against **Azure Storage Emulator** that comes with the Azure SDK. If you have an Azure Storage Account, you can run against the real storage service as well.

Http Message Handler Pipeline Sample | detailed description | VS 2010 source

Shows how to wire up **HttpMessageHandler** instances on both the client (**HttpClient**) and server (ASP.NET Web API). In the sample, the same handler is used on both the client and server. While it is rare that the exact same handler would run in both places, the object model is the same on client and server side.

JSON Upload Sample | VS 2012 source

Shows how to upload and download JSON to and from an **ApiController**. The sample uses a minimal **ApiController** and accesses it using **HttpClient**.

Mashup Sample | detailed description | VS 2012 source

Shows how to access multiple remote sites asynchronously from within an **ApiController** action. Each time the action is hit, the requests are performed asynchronously, so that no threads are blocked.

Memory Tracing Sample | detailed description | VS 2010 source

This sample project creates a Nuget package that will install a custom in-memory trace writer into ASP.NET Web API applications.

MongoDB Sample | detailed description | VS 2012 source

Shows how to use MongoDB as the persistent store for an **ApiController**, using a repository pattern.

Response Body Processor Sample | VS 2012 source

Shows how to copy a response entity (that is, an HTTP response body) to a local file before it is transmitted to the client, and perform additional processing on that file asynchronously. The sample implements an **HttpMessageHandler** that wraps the response entity with one that both writes itself to the output as normal and to a local file.

Upload XDocument Sample | detailed description | VS 2012 source

Shows how to upload an XDocument to an **ApiController** using **PushStreamContent** and **HttpClient**.

Validation Sample | VS 2010 source

Shows how you can use validation attributes on your models in ASP.NET WebAPI to validate the contents of the HTTP request. Demonstrates how to mark properties as required, how to use both framework-defined and custom validation attributes to annotate your model, and how to return error responses for invalid model states.

Web Form Sample | detailed description | VS 2010 source

Shows an ApiController added to a Web Forms project.

RestBugs Sample

RestBugs is a simple bug tracking application that shows how to use ASP.NET Web API and the new HTTP Client library to create a hypermedia-driven system. The sample includes both client and server implementations, using ASP.NET Web API. The server uses a custom Razor formatter to generate resource representations. The sample also provides a node.js server to illustrate the benefits that come from using a hypermedia design to decouple clients and servers.

ASP.NET Web Pages

4/10/2020 • 2 minutes to read • Edit Online

NOTE

WebMatrix is no longer recommended as an integrated development environment for ASP.NET Web Pages. Use Visual Studio or Visual Studio Code.

- Guidance
  - Getting Started
    - Program ASP.NET Web Pages in Visual Studio
    - Intro to ASP.NET Web Programming Razor Syntax
    - ASP.NET Web Pages Visual Basic
    - ASP.NET Web Pages (Razor) FAQ
    - Sending Email from Your Site
    - Adding Social Networking to Your Website
    - Introducing ASP.NET Web Pages
      - Getting Started
      - Programming Basics
      - Displaying Data
      - HTML Form Basics
      - Entering Database Data by Using Forms
      - Updating Database Data
      - Deleting Database Data
      - Creating a Consistent Layout
      - Publishing a Site by Using WebMatrix
  - Routing
    - Creating Readable URLs in ASP.NET Web Pages Sites
  - UI, Layouts, and Themes
    - Working with Forms
    - Validating User Input in ASP.NET Web Pages Sites
    - Creating a Consistent Look
    - Customizing Site-Wide Behavior
    - Working with Images
    - Working with Video
    - Creating and Using a Helper in an ASP.NET Web Pages Site
    - Installing a Helper
    - Twitter Helper with ASP.NET Web Pages
    - Displaying Maps in an ASP.NET Web Pages Site

- Mobile
  - Rendering ASP.NET Web Pages Sites for Mobile Devices
- Working with Data
  - Working with Data
  - Displaying Data in a Chart
  - Working with Files
- Testing and Debugging
  - Intro to Debugging
  - ASP.NET Web Pages (Razor) Troubleshooting Guide
- Security
  - Adding Security and Membership
  - Enabling Login from External Sites in an ASP.NET Web Pages Site
  - Using a CAPTCHA to Prevent Automated Programs (Bots) from Using Your ASP.NET Web Site
- Performance
  - Caching to Improve the Performance of Your Website
  - Analyzing Traffic
  - Bundling and Minifying Assets in an ASP.NET Web Pages (Razor) Site
- API Reference
  - ASP.NET API Quick Reference
- Releases
  - What's New in ASP.NET Web Pages 3.2
  - What's New in ASP.NET Web Pages 3.1
  - Top Features in ASP.NET Web Pages 2
  - ASP.NET Web Pages 2 Developer Preview ReadMe
  - Running Different Versions of ASP.NET Web Pages Side by Side

- Videos
  - Introduction
    - 5 Minute Intro to ASP.NET Web Pages
    - Install and Use the Microsoft WebMatrix Tool
    - Create a Website Using WebMatrix
    - Change the Visual Style of a Web Page
    - Use the Layout Features in WebMatrix
    - Create a Data-Driven Dynamic Web Page
    - Create a Web Interface in WebMatrix
    - Create an Edit Data Page in WebMatrix
    - Create a Delete Data Page in WebMatrix
    - Publish a Website to the Internet Using WebMatrix
  - ASP.NET Razor Pages
    - Getting Started with WebMatrix and ASP.NET Web Pages
    - Intro to ASP.NET Web Programming Razor Syntax
    - Creating a Consistent Look (Part 1)
    - Creating a Consistent Look (Part 2)

    - Working with Forms (Part 1)
    - Working with Forms (Part 2)
    - Working with Data (Part 1)
    - Working with Data (Part 2)
    - Displaying Data in a Grid
    - Displaying Data in a Chart (Part 1)
    - Displaying Data in a Chart (Part 2)
    - Working with Files
    - Working with Images
    - Working with Video
    - Adding Email to Your Web Site
    - Adding Search to Your Web Site
    - Adding Social Networking to Your Website
- WebMatrix and Web Pages Video Training from Pluralsight
- WebMatrix Readme
    - WebMatrix Readme
    - Web Matrix and ASP.NET Web Pages (Razor) Beta 3 Release Readme
- WebMatrix Content Guide

Web Pages Guidance

2/20/2020 • 2 minutes to read • Edit Online

- Getting Started
  - Program ASP.NET Web Pages in Visual Studio
  - Intro to ASP.NET Web Programming Razor Syntax
  - ASP.NET Web Pages Visual Basic
  - ASP.NET Web Pages (Razor) FAQ
  - Sending Email from Your Site
  - Adding Social Networking to Your Website
  - Introducing ASP.NET Web Pages
    - Getting Started
    - Programming Basics
    - Displaying Data
    - HTML Form Basics
    - Entering Database Data by Using Forms
    - Updating Database Data
    - Deleting Database Data
    - Creating a Consistent Layout
    - Publishing a Site by Using WebMatrix
- Routing
  - Creating Readable URLs in ASP.NET Web Pages Sites
- UI, Layouts, and Themes
  - Working with Forms
  - Validating User Input in ASP.NET Web Pages Sites
  - Creating a Consistent Look
  - Customizing Site-Wide Behavior
  - Working with Images
  - Working with Video
  - Creating and Using a Helper in an ASP.NET Web Pages Site
  - Installing a Helper
  - Twitter Helper with ASP.NET Web Pages
  - Displaying Maps in an ASP.NET Web Pages Site
- Mobile
  - Rendering ASP.NET Web Pages Sites for Mobile Devices
- Working with Data
  - Working with Data
  - Displaying Data in a Chart
  - Working with Files

- Testing and Debugging
  - Intro to Debugging
  - ASP.NET Web Pages (Razor) Troubleshooting Guide
- Security
  - Adding Security and Membership
  - Enabling Login from External Sites in an ASP.NET Web Pages Site
  - Using a CAPTCHA to Prevent Automated Programs (Bots) from Using Your ASP.NET Web Site
- Performance
  - Caching to Improve the Performance of Your Website
  - Analyzing Traffic
  - Bundling and Minifying Assets in an ASP.NET Web Pages (Razor) Site
- API Reference
  - ASP.NET API Quick Reference
- Releases
  - What's New in ASP.NET Web Pages 3.2
  - What's New in ASP.NET Web Pages 3.1
  - Top Features in ASP.NET Web Pages 2
  - ASP.NET Web Pages 2 Developer Preview ReadMe
  - Running Different Versions of ASP.NET Web Pages Side by Side

Getting Started with ASP.NET Web Pages

2/20/2020 • 2 minutes to read • Edit Online

What Web Pages and WebMatrix are, plus how to install everything, get started, and deploy sites.

- Program ASP.NET Web Pages in Visual Studio
- Intro to ASP.NET Web Programming Razor Syntax
- ASP.NET Web Pages Visual Basic
- ASP.NET Web Pages (Razor) FAQ
- Sending Email from Your Site
- Adding Social Networking to Your Website
- Introducing ASP.NET Web Pages
  - Getting Started
  - Programming Basics
  - Displaying Data
  - HTML Form Basics
  - Entering Database Data by Using Forms
  - Updating Database Data
  - Deleting Database Data
  - Creating a Consistent Layout
  - Publishing a Site by Using WebMatrix

Programming ASP.NET Web Pages (Razor) Using Visual Studio

4/10/2020 • 5 minutes to read • Edit Online

by Tom FitzMacken

This article explains how you can use Visual Studio or Visual Web Developer Express to program ASP.NET Web Pages (Razor) websites.

What you'll learn

- What you need to install (if anything) to work with ASP.NET Web Pages in your version of Visual Studio.
- How to add support for ASP.NET Web Pages to Visual Web Developer 2010 Express.
- How to use features in Visual Studio to work with ASP.NET Razor pages, including IntelliSense and the debugger.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 3
- Visual Studio 2013
- WebMatrix 3

This tutorial also works with ASP.NET Web Pages 2, Visual Studio 2012, Visual Studio 2010, and WebMatrix 2.

You can program ASP.NET Web pages with Razor syntax using WebMatrix or many other code editors. You can also use Microsoft Visual Studio which is a full-featured integrated development environment (IDE) that provides a powerful set of tools for creating many types of applications (not just websites). To work with ASP.NET Razor pages, you can use Visual Studio 2017.

Two particularly useful features that Visual Studio provides for programming with ASP.NET Razor web pages are:

- *IntelliSense*. The IntelliSense feature built into Visual Studio is more comprehensive than IntelliSense in WebMatrix.
- *Debugger*. The debugger lets you troubleshoot your code by stopping a program while it's running, examining variables, and stepping through the code line by line.

Using Visual Studio with Different Versions of ASP.NET Web Pages

To develop ASP.NET web apps in Visual Studio 2017, install the **ASP.NET and web development** workload.

Visual Studio 2012 and Visual Studio 2013 include support for ASP.NET Web Pages. (The packages that are required in order to support ASP.NET Web Pages are installed when you install Visual Studio.)

Visual Studio 2010 does not include support by default for ASP.NET Web Pages. To use ASP.NET Web Pages with Visual Studio 2010, you must install the ASP.NET MVC package. To get ASP.NET Web Pages 2, you install ASP.NET MVC 4.

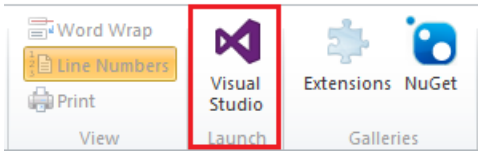
The following table summarizes the support for ASP.NET Web Pages in different versions of Visual Studio.

	VISUAL STUDIO 2010	VISUAL STUDIO 2012	VISUAL STUDIO 2013
ASP.NET Web Pages 2	Install ASP.NET MVC 4	(Included)	(Included)
ASP.NET Web Pages 3		Update to ASP.NET Web Pages 3 through NuGet	(Included)

To work with Visual Studio 2010, see Installing Support for ASP.NET Web Pages in Visual Studio 2010.

Launching Visual Studio from WebMatrix

If you have started a project in WebMatrix and want to switch to Visual Studio, WebMatrix provides a button to easily open the project in Visual Studio. You must have Visual Studio installed on your computer for this button to be enabled. The following image shows the button in WebMatrix.

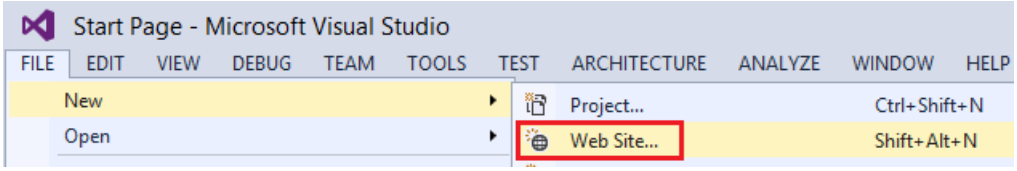


When you click the button, the project is opened in Visual Studio. You can switch back and forth between WebMatrix and Visual Studio without any problems. You will be notified if any files have changed in the other environment and need to be reloaded to get the latest changes.

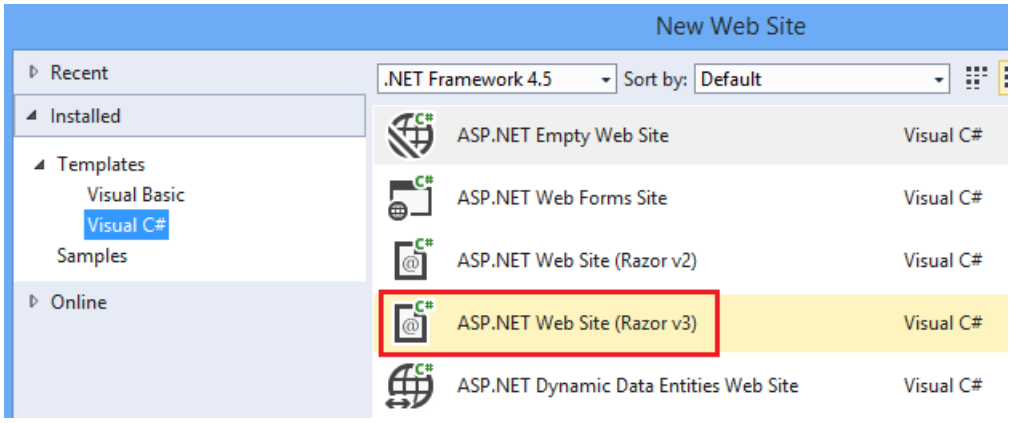
Creating ASP.NET Razor Site in Visual Studio

To create an ASP.NET Razor website in Visual Studio:

1. Open Visual Studio.
2. In the **File** menu, click **New Web Site**.



3. In the **New Web Site** dialog box, select the language to use (Visual C# or Visual Basic).
4. Select the **ASP.NET Web Site (Razor)** template.



5. Click **OK**.

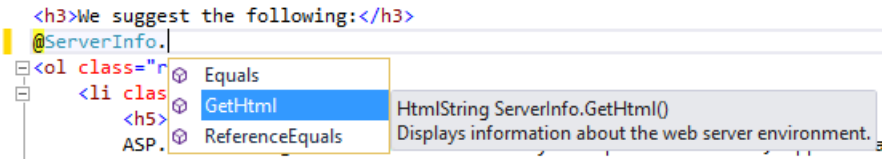
Your new project exists and is populated with some default web pages to help you get started.

Using IntelliSense

Now that you've created a site, you can see how IntelliSense works in Visual Studio.

1. In the website you just created, open the *Default.cshtml* page.

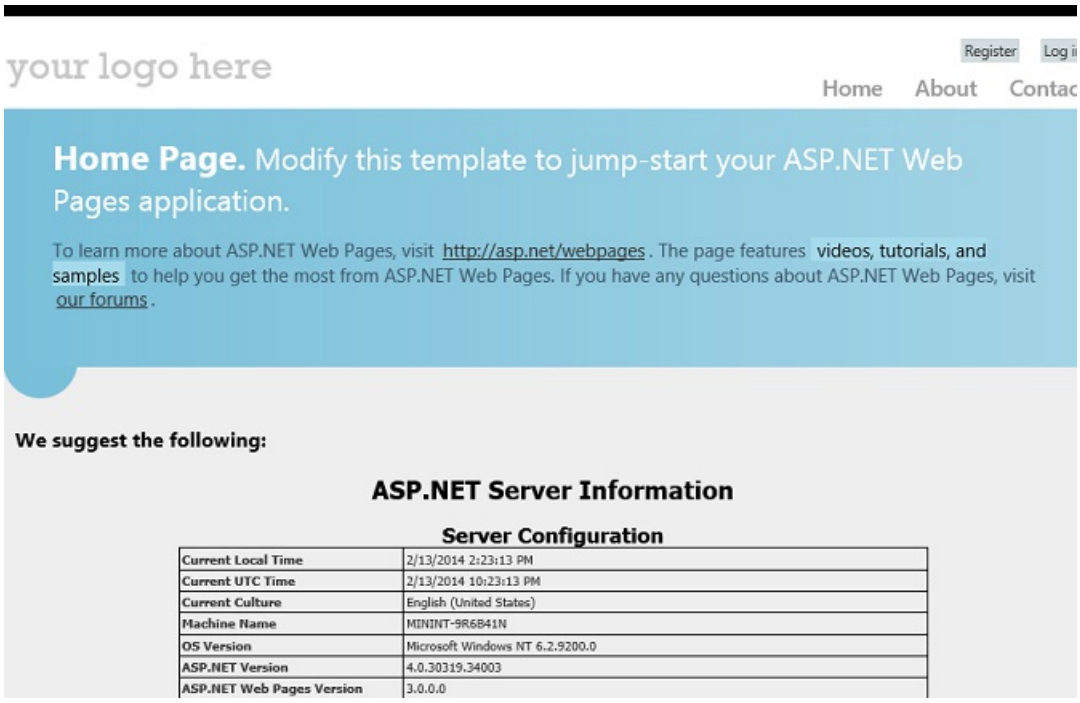
2. After the `<h3>` tags in the page, type `@ServerInfo.` (including the dot). Notice how IntelliSense displays the available methods for the `ServerInfo` helper in a drop-down list.



3. Select the `GetHtml` method from the list and then press Enter. IntelliSense automatically fills in the method. (As with any method in C#, you must add `()` characters after the method.) The completed code for the `GetHtml` method looks like the following example:

```
@ServerInfo.GetHtml()
```

4. Press Ctrl+F5 to run the page. This is what the page looks like when displayed in a browser:



5. Close the browser.

Using the Debugger

1. At the top of the *Default.cshtml* page, after the line that begins with `Page.Title`, add the following line of code:

```
var myTime = DateTime.Now.TimeOfDay;
```

2. In the gray margin of the editor to the left of the code, click next to this new line in order to add a *breakpoint*. A breakpoint is a marker that tells the debugger to stop running the program at that point so you can see what's happening.

```
@{
    Layout = "~/_SiteLayout.cshtml";
    Page.Title = "Home Page";
    var myTime = DateTime.Now.TimeOfDay;
}
```

3. Remove the call to the `ServerInfo.GetHtml` method, and add a call to the `@myTime` variable in its place. This call displays the current time value that's returned by the new line of code.

4. Press F5 to run the page in the debugger. The page stops on the breakpoint that you set. The following image shows what the page looks like in the editor with the breakpoint (in yellow).

```
@{
    Layout = "~/_SiteLayout.cshtml";
    Page.Title = "Home Page";
    var myTime = DateTime.Now.TimeOfDay;
}
```

5. In the Debug toolbar, click the **Step Into** button (or press F11) to run the next line of code. Each time you click this button, you advance the execution to the next line of code.

6. Examine the value of the `myTime` variable by holding your mouse pointer over it or by inspecting the values displayed in the **Locals** and **Call Stack** windows. Visual Studio display the value of the variable.

```
@{
    Layout = "~/_SiteLayout.cshtml";
    Page.Title = "Home Page";
    var myTime = DateTime.Now.TimeOfDay;
}
```

myTime {14:54:10.4713083}

7. When you're done examining the variable and stepping through code, press F5 to continue running the page without stopping at each line. When you've finished stepping through all the code, the browser displays the page.

To learn more about the debugger and about how to debug code in Visual Studio, see Walkthrough: Debugging Web Pages in Visual Web Developer.

Using Razor in ASP.NET MVC projects with Visual Studio

The Razor syntax is also used extensively in ASP.NET MVC projects. MVC is a powerful, patterns-based way to build dynamic websites. If your ASP.NET Web Pages site becomes difficult to maintain, you might want to consider converting it to an ASP.NET MVC application. For an example of creating an MVC application, see Getting Started with ASP.NET MVC 5.

Installing Support for ASP.NET Web Pages in Visual Studio 2010

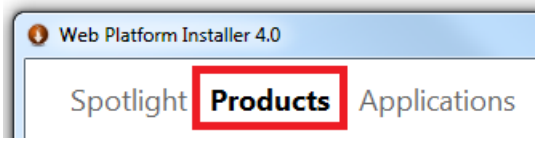
This section shows how to install Visual Web Developer Express 2010 and the ASP.NET Web Pages Tools for Visual Studio.

1. If you don't already have the Web Platform Installer, download it from the following URL:

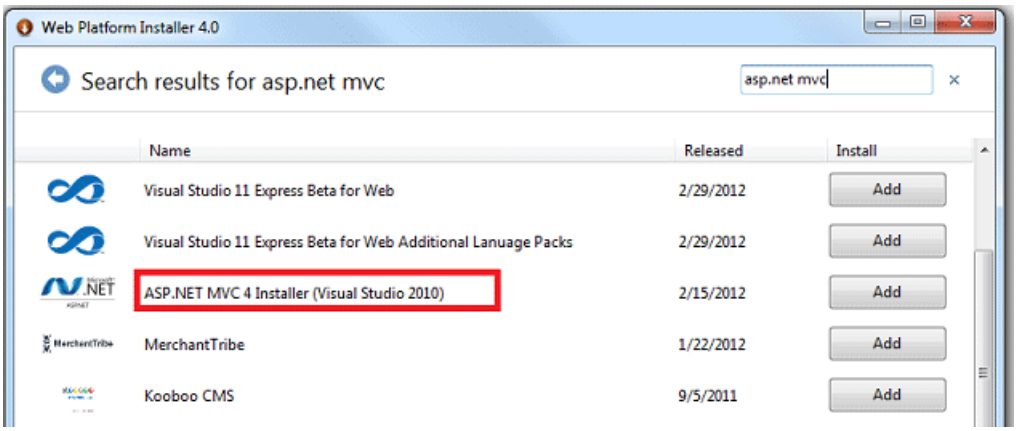
   https://www.microsoft.com/web/downloads/platform.aspx

2. Run the Web Platform Installer.

3. Click the **Products** tab.



4. Search for **ASP.NET MVC 4** (for ASP.NET Web Pages 2) and then click **Add**. These products include Visual Studio tools for building ASP.NET Razor websites.



5. Click **Install** to complete the installation.

Introduction to ASP.NET Web Programming Using the Razor Syntax (C#)

2/20/2020 • 40 minutes to read • Edit Online

by Tom FitzMacken

This article gives you an overview of programming with ASP.NET Web Pages using the Razor syntax. ASP.NET is Microsoft's technology for running dynamic web pages on web servers. This articles focuses on using the C# programming language.

What you'll learn:

- The top 8 programming tips for getting started with programming ASP.NET Web Pages using Razor syntax.
- Basic programming concepts you'll need.
- What ASP.NET server code and the Razor syntax is all about.

Software versions

- ASP.NET Web Pages (Razor) 3

This tutorial also works with ASP.NET Web Pages 2.

The Top 8 Programming Tips

This section lists a few tips that you absolutely need to know as you start writing ASP.NET server code using the Razor syntax.

> **NOTE**
>
> The Razor syntax is based on the C# programming language, and that's the language that's used most often with ASP.NET Web Pages. However, the Razor syntax also supports the Visual Basic language, and everything you see you can also do in Visual Basic. For details, see the appendix Visual Basic Language and Syntax.

You can find more details about most of these programming techniques later in the article.

1. You add code to a page using the @ character

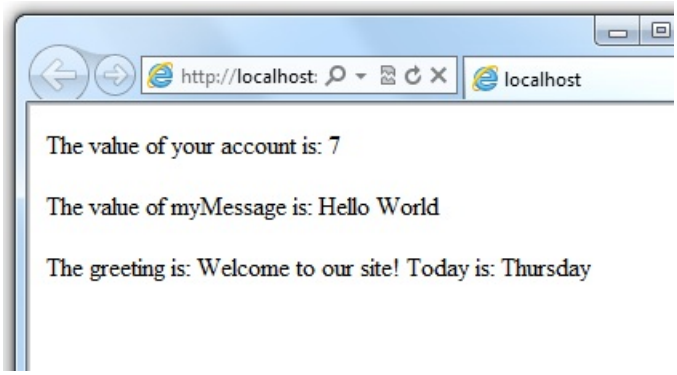
The `@` character starts inline expressions, single statement blocks, and multi-statement blocks:

```
<!-- Single statement blocks  -->
@{ var total = 7; }
@{ var myMessage = "Hello World"; }

<!-- Inline expressions -->
<p>The value of your account is: @total </p>
<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->
@{
    var greeting = "Welcome to our site!";
    var weekDay = DateTime.Now.DayOfWeek;
    var greetingMessage = greeting + " Today is: " + weekDay;
}
<p>The greeting is: @greetingMessage</p>
```

This is what these statements look like when the page runs in a browser:



TIP

HTML Encoding

When you display content in a page using the `@` character, as in the preceding examples, ASP.NET HTML-encodes the output. This replaces reserved HTML characters (such as `<` and `>` and `&`) with codes that enable the characters to be displayed as characters in a web page instead of being interpreted as HTML tags or entities. Without HTML encoding, the output from your server code might not display correctly, and could expose a page to security risks.

If your goal is to output HTML markup that renders tags as markup (for example `<p></p>` for a paragraph or `<em></em>` to emphasize text), see the section Combining Text, Markup, and Code in Code Blocks later in this article.

You can read more about HTML encoding in Working with Forms.

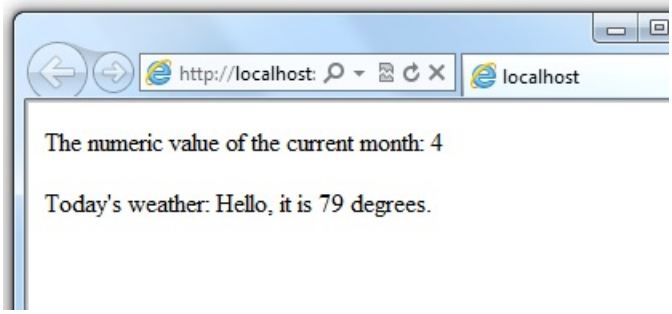
2. You enclose code blocks in braces

A *code block* includes one or more code statements and is enclosed in braces.

```
<!-- Single statement block.  -->
@{ var theMonth = DateTime.Now.Month; }
<p>The numeric value of the current month: @theMonth</p>

<!-- Multi-statement block. -->
@{
    var outsideTemp = 79;
    var weatherMessage = "Hello, it is " + outsideTemp + " degrees.";
}
<p>Today's weather: @weatherMessage</p>
```

The result displayed in a browser:



3. Inside a block, you end each code statement with a semicolon

Inside a code block, each complete code statement must end with a semicolon. Inline expressions don't end with a semicolon.

```
<!-- Single-statement block -->
@{ var theMonth = DateTime.Now.Month; }

<!-- Multi-statement block -->
@{
    var outsideTemp = 79;
    var weatherMessage = "Hello, it is " + outsideTemp + " degrees.";
}

<!-- Inline expression, so no semicolon -->
<p>Today's weather: @weatherMessage</p>
```

4. You use variables to store values

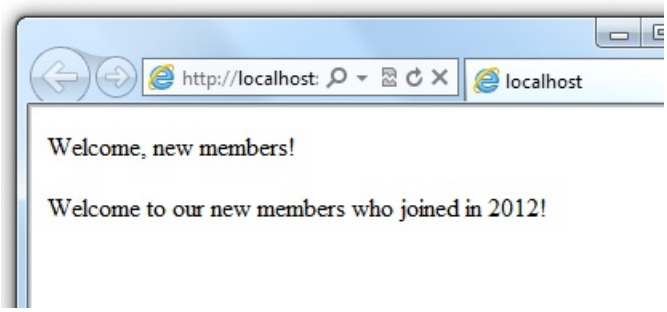
You can store values in a *variable*, including strings, numbers, and dates, etc. You create a new variable using the `var` keyword. You can insert variable values directly in a page using `@`.

```
<!-- Storing a string -->
@{ var welcomeMessage = "Welcome, new members!"; }
<p>@welcomeMessage</p>

<!-- Storing a date -->
@{ var year = DateTime.Now.Year; }

<!-- Displaying a variable -->
<p>Welcome to our new members who joined in @year!</p>
```

The result displayed in a browser:



5. You enclose literal string values in double quotation marks

A *string* is a sequence of characters that are treated as text. To specify a string, you enclose it in double quotation marks:

```
@{ var myString = "This is a string literal"; }
```

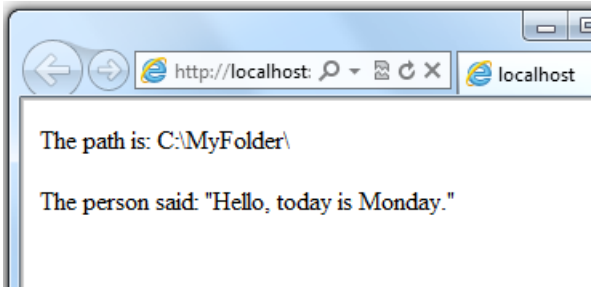
If the string that you want to display contains a backslash character (`\`) or double quotation marks (`"`), use a *verbatim string literal* that's prefixed with the `@` operator. (In C#, the \ character has special meaning unless you use a verbatim string literal.)

```
<!-- Embedding a backslash in a string -->
@{ var myFilePath = @"C:\MyFolder\"; }
<p>The path is: @myFilePath</p>
```

To embed double quotation marks, use a verbatim string literal and repeat the quotation marks:

```
<!-- Embedding double quotation marks in a string -->
@{ var myQuote = @"The person said: ""Hello, today is Monday."""; }
<p>@myQuote</p>
```

Here's the result of using both of these examples in a page:



> **NOTE**
> Notice that the `@` character is used both to mark verbatim string literals in C# and to mark code in ASP.NET pages.

6. Code is case sensitive

In C#, keywords (like `var`, `true`, and `if`) and variable names are case sensitive. The following lines of code create two different variables, `lastName` and `LastName.`

```
@{
    var lastName = "Smith";
    var LastName = "Jones";
}
```

If you declare a variable as `var lastName = "Smith";` and try to reference that variable in your page as `@LastName`, you would get the value `"Jones"` instead of `"Smith"`.

> **NOTE**
> In Visual Basic, keywords and variables are *not* case sensitive.

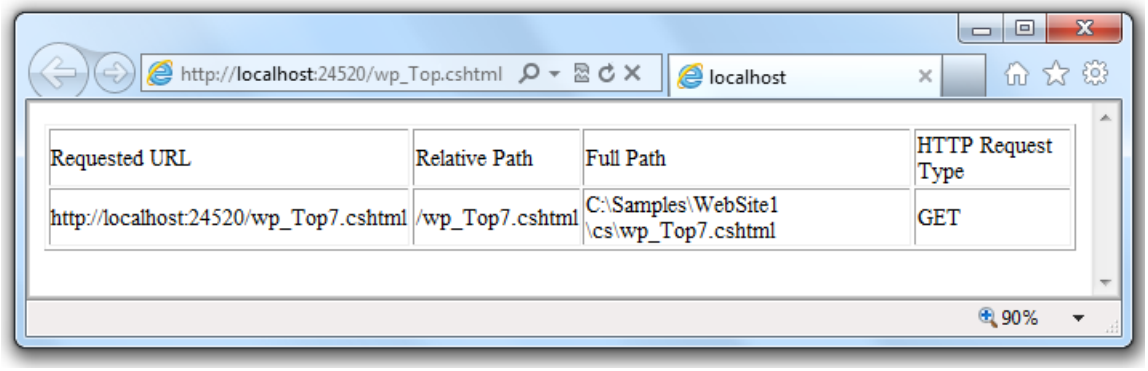
7. Much of your coding involves objects

An *object* represents a thing that you can program with — a page, a text box, a file, an image, a web request, an email message, a customer record (database row), etc. Objects have properties that describe their characteristics and that you can read or change — a text box object has a `Text` property (among others), a request object has a `Url` property, an email message has a `From` property, and a customer object has a `FirstName` property. Objects also have methods that are the "verbs" they can perform. Examples include a file object's `Save` method, an image object's `Rotate` method, and an email object's `Send` method.

You'll often work with the `Request` object, which gives you information like the values of text boxes (form fields) on the page, what type of browser made the request, the URL of the page, the user identity, etc. The following example shows how to access properties of the `Request` object and how to call the `MapPath` method of the `Request` object, which gives you the absolute path of the page on the server:

```
<table border="1">
<tr>
    <td>Requested URL</td>
    <td>Relative Path</td>
    <td>Full Path</td>
    <td>HTTP Request Type</td>
</tr>
<tr>
    <td>@Request.Url</td>
    <td>@Request.FilePath</td>
    <td>@Request.MapPath(Request.FilePath)</td>
    <td>@Request.RequestType</td>
</tr>
</table>
```

The result displayed in a browser:



8. You can write code that makes decisions

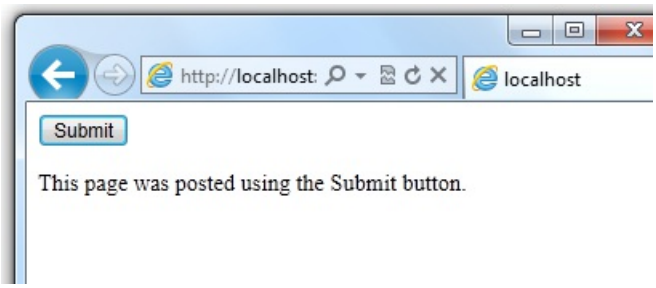
A key feature of dynamic web pages is that you can determine what to do based on conditions. The most common way to do this is with the `if` statement (and optional `else` statement).

```
@{
   var result = "";
   if(IsPost)
   {
      result = "This page was posted using the Submit button.";
   }
   else
   {
      result = "This was the first request for this page.";
   }
}

<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
<body>
<form method="POST" action="" >
  <input type="Submit" name="Submit" value="Submit"/>
  <p>@result</p>
</form>
</body>
</html>
```

The statement `if(IsPost)` is a shorthand way of writing `if(IsPost == true)`. Along with `if` statements, there are a variety of ways to test conditions, repeat blocks of code, and so on, which are described later in this article.

The result displayed in a browser (after clicking **Submit**):



TIP

HTTP GET and POST Methods and the IsPost Property

The protocol used for web pages (HTTP) supports a very limited number of methods (verbs) that are used to make requests to the server. The two most common ones are GET, which is used to read a page, and POST, which is used to submit a page. In general, the first time a user requests a page, the page is requested using GET. If the user fills in a form and then clicks a submit button, the browser makes a POST request to the server.

In web programming, it's often useful to know whether a page is being requested as a GET or as a POST so that you know how to process the page. In ASP.NET Web Pages, you can use the `IsPost` property to see whether a request is a GET or a POST. If the request is a POST, the `IsPost` property will return true, and you can do things like read the values of text boxes on a form. Many examples you'll see show you how to process the page differently depending on the value of `IsPost`.

A Simple Code Example

This procedure shows you how to create a page that illustrates basic programming techniques. In the example, you create a page that lets users enter two numbers, then it adds them and displays the result.

1. In your editor, create a new file and name it *AddNumbers.cshtml*.

2. Copy the following code and markup into the page, replacing anything already in the page.

```
@{
    var total = 0;
    var totalMessage = "";
    if(IsPost) {

        // Retrieve the numbers that the user entered.
        var num1 = Request["text1"];
        var num2 = Request["text2"];

        // Convert the entered strings into integers numbers and add.
        total = num1.AsInt() + num2.AsInt();
        totalMessage = "Total = " + total;
    }
}

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Add Numbers</title>
    <meta charset="utf-8" />
    <style type="text/css">
      body {background-color: beige; font-family: Verdana, Arial;
            margin: 50px; }
      form {padding: 10px; border-style: solid; width: 250px;}
    </style>
  </head>
<body>
  <p>Enter two whole numbers and then click <strong>Add</strong>.</p>
  <form action="" method="post">
    <p><label for="text1">First Number:</label>
      <input type="text" name="text1" />
    </p>
    <p><label for="text2">Second Number:</label>
      <input type="text" name="text2" />
    </p>
    <p><input type="submit" value="Add" /></p>
  </form>

  <p>@totalMessage</p>

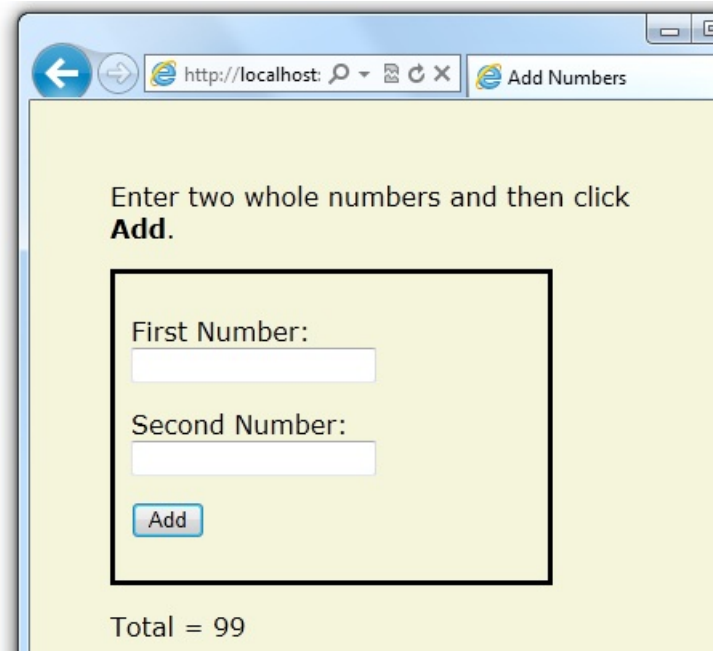
</body>
</html>
```

Here are some things for you to note:

- The `@` character starts the first block of code in the page, and it precedes the `totalMessage` variable that's embedded near the bottom of the page.
- The block at the top of the page is enclosed in braces.
- In the block at the top, all lines end with a semicolon.
- The variables `total`, `num1`, `num2`, and `totalMessage` store several numbers and a string.
- The literal string value assigned to the `totalMessage` variable is in double quotation marks.
- Because the code is case-sensitive, when the `totalMessage` variable is used near the bottom of the page, its name must match the variable at the top exactly.
- The expression `num1.AsInt() + num2.AsInt()` shows how to work with objects and methods. The `AsInt` method on each variable converts the string entered by a user to a number (an integer) so that you can perform arithmetic on it.
- The `<form>` tag includes a `method="post"` attribute. This specifies that when the user clicks **Add**, the page will be sent to the server using the HTTP POST method. When the page is submitted, the `if(IsPost)` test evaluates to true and the conditional code runs, displaying the result of adding the

numbers.

3. Save the page and run it in a browser. (Make sure the page is selected in the **Files** workspace before you run it.) Enter two whole numbers and then click the **Add** button.



Basic Programming Concepts

This article provides you with an overview of ASP.NET web programming. It isn't an exhaustive examination, just a quick tour through the programming concepts you'll use most often. Even so, it covers almost everything you'll need to get started with ASP.NET Web Pages.

But first, a little technical background.

The Razor Syntax, Server Code, and ASP.NET

Razor syntax is a simple programming syntax for embedding server-based code in a web page. In a web page that uses the Razor syntax, there are two kinds of content: client content and server code. Client content is the stuff you're used to in web pages: HTML markup (elements), style information such as CSS, maybe some client script such as JavaScript, and plain text.

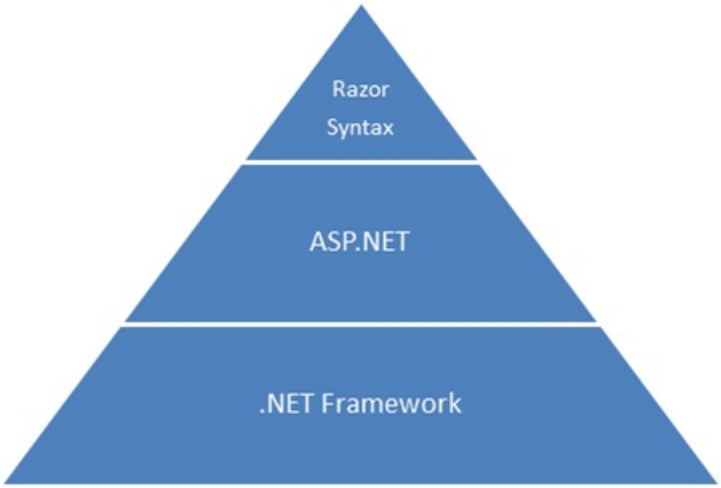
Razor syntax lets you add server code to this client content. If there's server code in the page, the server runs that code first, before it sends the page to the browser. By running on the server, the code can perform tasks that can be a lot more complex to do using client content alone, like accessing server-based databases. Most importantly, server code can dynamically create client content — it can generate HTML markup or other content on the fly and then send it to the browser along with any static HTML that the page might contain. From the browser's perspective, client content that's generated by your server code is no different than any other client content. As you've already seen, the server code that's required is quite simple.

ASP.NET web pages that include the Razor syntax have a special file extension (*.cshtml* or *.vbhtml*). The server recognizes these extensions, runs the code that's marked with Razor syntax, and then sends the page to the browser.

Where does ASP.NET fit in?

Razor syntax is based on a technology from Microsoft called ASP.NET, which in turn is based on the Microsoft .NET Framework. The.NET Framework is a big, comprehensive programming framework from Microsoft for developing virtually any type of computer application. ASP.NET is the part of the .NET Framework that's specifically designed for creating web applications. Developers have used ASP.NET to create many of the largest and highest-traffic websites in the world. (Any time you see the file-name extension *.aspx* as part of the URL in a site, you'll know that the site was written using ASP.NET.)

The Razor syntax gives you all the power of ASP.NET, but using a simplified syntax that's easier to learn if you're a beginner and that makes you more productive if you're an expert. Even though this syntax is simple to use, its family relationship to ASP.NET and the .NET Framework means that as your websites become more sophisticated, you have the power of the larger frameworks available to you.



TIP

Classes and Instances

ASP.NET server code uses objects, which are in turn built on the idea of classes. The class is the definition or template for an object. For example, an application might contain a `Customer` class that defines the properties and methods that any customer object needs.

When the application needs to work with actual customer information, it creates an instance of (or *instantiates*) a customer object. Each individual customer is a separate instance of the `Customer` class. Every instance supports the same properties and methods, but the property values for each instance are typically different, because each customer object is unique. In one customer object, the `LastName` property might be "Smith"; in another customer object, the `LastName` property might be "Jones."

Similarly, any individual web page in your site is a `Page` object that's an instance of the `Page` class. A button on the page is a `Button` object that is an instance of the `Button` class, and so on. Each instance has its own characteristics, but they all are based on what's specified in the object's class definition.

Basic Syntax

Earlier you saw a basic example of how to create an ASP.NET Web Pages page, and how you can add server code to HTML markup. Here you'll learn the basics of writing ASP.NET server code using the Razor syntax — that is, the programming language rules.

If you're experienced with programming (especially if you've used C, C++, C#, Visual Basic, or JavaScript), much of what you read here will be familiar. You'll probably need to familiarize yourself only with how server code is added to markup in *.cshtml* files.

Combining Text, Markup, and Code in Code Blocks

In server code blocks, you often want to output text or markup (or both) to the page. If a server code block contains text that's not code and that instead should be rendered as is, ASP.NET needs to be able to distinguish that text from code. There are several ways to do this.

- Enclose the text in an HTML element like `<p></p>` or `<em></em>`:

```
@if(IsPost) {
    // This line has all content between matched <p> tags.
    <p>Hello, the time is @DateTime.Now and this page is a postback!</p>
} else {
    // All content between matched tags, followed by server code.
    <p>Hello <em>stranger</em>, today is: <br /> </p>  @DateTime.Now
}
```

The HTML element can include text, additional HTML elements, and server-code expressions. When ASP.NET sees the opening HTML tag (for example, `<p>`), it renders everything including the element and its content as is to the browser, resolving server-code expressions as it goes.

- Use the `@:` operator or the `<text>` element. The `@:` outputs a single line of content containing plain text or unmatched HTML tags; the `<text>` element encloses multiple lines to output. These options are useful when you don't want to render an HTML element as part of the output.

```
@if(IsPost) {
    // Plain text followed by an unmatched HTML tag and server code.
    @: The time is: <br /> @DateTime.Now
    <br/>
    // Server code and then plain text, matched tags, and more text.
    @DateTime.Now @:is the <em>current</em> time.
}
```

If you want to output multiple lines of text or unmatched HTML tags, you can precede each line with `@:`, or you can enclose the line in a `<text>` element. Like the `@:` operator, `<text>` tags are used by ASP.NET to identify text content and are never rendered in the page output.

```
@if(IsPost) {
    // Repeat the previous example, but use <text> tags.
    <text>
    The time is: <br /> @DateTime.Now
    <br/>
    @DateTime.Now is the <em>current</em> time.
    </text>
}

@{
    var minTemp = 75;
    <text>It is the month of @DateTime.Now.ToString("MMMM"), and
    it's a <em>great</em> day! <br /><p>You can go swimming if it's at
    least @minTemp degrees. </p></text>
}
```

The first example repeats the previous example but uses a single pair of `<text>` tags to enclose the text to render. In the second example, the `<text>` and `</text>` tags enclose three lines, all of which have some uncontained text and unmatched HTML tags (`<br />`), along with server code and matched HTML tags. Again, you could also precede each line individually with the `@:` operator; either way works.

> **NOTE**
>
> When you output text as shown in this section — using an HTML element, the `@:` operator, or the `<text>` element — ASP.NET doesn't HTML-encode the output. (As noted earlier, ASP.NET does encode the output of server code expressions and server code blocks that are preceded by `@`, except in the special cases noted in this section.)

Whitespace

Extra spaces in a statement (and outside of a string literal) don't affect the statement:

```
@{ var lastName =    "Smith"; }
```

A line break in a statement has no effect on the statement, and you can wrap statements for readability. The following statements are the same:

```
@{ var theName =
"Smith"; }

@{
    var
    personName
    =
    "Smith"
    ;
}
```

However, you can't wrap a line in the middle of a string literal. The following example doesn't work:

```
@{ var test = "This is a long
    string"; }  // Does not work!
```

To combine a long string that wraps to multiple lines like the above code, there are two options. You can use the concatenation operator (`+`), which you'll see later in this article. You can also use the `@` character to create a verbatim string literal, as you saw earlier in this article. You can break verbatim string literals across lines:

```
@{ var longString = @"This is a
    long
    string";
}
```

Code (and Markup) Comments

Comments let you leave notes for yourself or others. They also allow you to disable (*comment out*) a section of code or markup that you don't want to run but want to keep in your page for the time being.

There's different commenting syntax for Razor code and for HTML markup. As with all Razor code, Razor comments are processed (and then removed) on the server before the page is sent to the browser. Therefore, the Razor commenting syntax lets you put comments into the code (or even into the markup) that you can see when you edit the file, but that users don't see, even in the page source.

For ASP.NET Razor comments, you start the comment with `@*` and end it with `*@`. The comment can be on one line or multiple lines:

```
@*  A one-line code comment. *@

@*
    This is a multiline code comment.
    It can continue for any number of lines.
*@
```

Here is a comment within a code block:

```
@{
    @* This is a comment. *@
    var theVar = 17;
}
```

Here is the same block of code, with the line of code commented out so that it won't run:

```
@{
    @* This is a comment. *@
    @* var theVar = 17;  *@
}
```

Inside a code block, as an alternative to using Razor comment syntax, you can use the commenting syntax of the programming language you're using, such as C#:

```
@{
    // This is a comment.
    var myVar = 17;
    /* This is a multi-line comment
    that uses C# commenting syntax. */
}
```

In C#, single-line comments are preceded by the `//` characters, and multi-line comments begin with `/*` and end with `*/`. (As with Razor comments, C# comments are not rendered to the browser.)

For markup, as you probably know, you can create an HTML comment:

```
<!-- This is a comment.  -->
```

HTML comments start with `<!--` characters and end with `-->`. You can use HTML comments to surround not only text, but also any HTML markup that you may want to keep in the page but don't want to render. This HTML comment will hide the entire content of the tags and the text they contain:

```
<!-- <p>This is my paragraph.</p>  -->
```

Unlike Razor comments, HTML comments *are* rendered to the page and the user can see them by viewing the page source.

Razor has limitations on nested blocks of C#. For more information see Named C# Variables and Nested Blocks Generate Broken Code

Variables

A variable is a named object that you use to store data. You can name variables anything, but the name must begin with an alphabetic character and it cannot contain whitespace or reserved characters.

Variables and Data Types

A variable can have a specific data type, which indicates what kind of data is stored in the variable. You can have string variables that store string values (like "Hello world"), integer variables that store whole-number values (like 3 or 79), and date variables that store date values in a variety of formats (like 4/12/2012 or March 2009). And there are many other data types you can use.

However, you generally don't have to specify a type for a variable. Most of the time, ASP.NET can figure out the type based on how the data in the variable is being used. (Occasionally you must specify a type; you'll see

examples where this is true.)

You declare a variable using the `var` keyword (if you don't want to specify a type) or by using the name of the type:

```
@{
    // Assigning a string to a variable.
    var greeting = "Welcome!";

    // Assigning a number to a variable.
    var theCount = 3;

    // Assigning an expression to a variable.
    var monthlyTotal = theCount + 5;

    // Assigning a date value to a variable.
    var today = DateTime.Today;

    // Assigning the current page's URL to a variable.
    var myPath = this.Request.Url;

    // Declaring variables using explicit data types.
    string name = "Joe";
    int count = 5;
    DateTime tomorrow = DateTime.Now.AddDays(1);
}
```

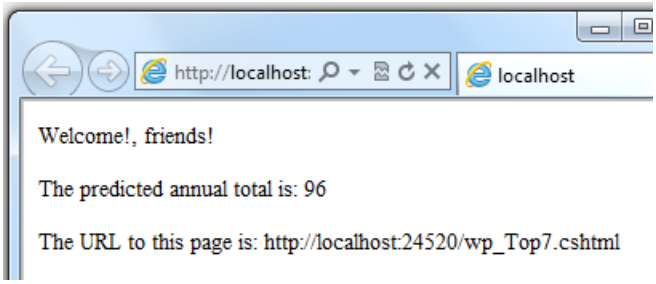
The following example shows some typical uses of variables in a web page:

```
@{
    // Embedding the value of a variable into HTML markup.
    <p>@greeting, friends!</p>

    // Using variables as part of an inline expression.
    <p>The predicted annual total is: @( monthlyTotal * 12)</p>

    // Displaying the page URL with a variable.
    <p>The URL to this page is: @myPath</p>
}
```

If you combine the previous examples in a page, you see this displayed in a browser:



Converting and Testing Data Types

Although ASP.NET can usually determine a data type automatically, sometimes it can't. Therefore, you might need to help ASP.NET out by performing an explicit conversion. Even if you don't have to convert types, sometimes it's helpful to test to see what type of data you might be working with.

The most common case is that you have to convert a string to another type, such as to an integer or date. The following example shows a typical case where you must convert a string to a number.

```
@{
    var total = 0;

    if(IsPost) {
        // Retrieve the numbers that the user entered.
        var num1 = Request["text1"];
        var num2 = Request["text2"];
        // Convert the entered strings into integers numbers and add.
        total = num1.AsInt() + num2.AsInt();
    }
}
```

As a rule, user input comes to you as strings. Even if you've prompted users to enter a number, and even if they've entered a digit, when user input is submitted and you read it in code, the data is in string format. Therefore, you must convert the string to a number. In the example, if you try to perform arithmetic on the values without converting them, the following error results, because ASP.NET cannot add two strings:

Cannot implicitly convert type 'string' to 'int'.

To convert the values to integers, you call the `AsInt` method. If the conversion is successful, you can then add the numbers.

The following table lists some common conversion and test methods for variables.

Method

Description

Example

`AsInt(), IsInt()`

Converts a string that represents a whole number (like "593") to an integer.

```
var myIntNumber = 0;
var myStringNum = "539";
if(myStringNum.IsInt()==true){
    myIntNumber = myStringNum.AsInt();
}
```

`AsBool(), IsBool()`

Converts a string like "true" or "false" to a Boolean type.

```
var myStringBool = "True";
var myVar = myStringBool.AsBool();
```

`AsFloat(), IsFloat()`

Converts a string that has a decimal value like "1.3" or "7.439" to a floating-point number.

```
var myStringFloat = "41.432895";
var myFloatNum = myStringFloat.AsFloat();
```

`AsDecimal(), IsDecimal()`

Converts a string that has a decimal value like "1.3" or "7.439" to a decimal number. (In ASP.NET, a decimal

number is more precise than a floating-point number.)

```
var myStringDec = "10317.425";
var myDecNum = myStringDec.AsDecimal();
```

`AsDateTime(), IsDateTime()`

Converts a string that represents a date and time value to the ASP.NET `DateTime` type.

```
var myDateString = "12/27/2012";
var newDate = myDateString.AsDateTime();
```

`ToString()`

Converts any other data type to a string.

```
int num1 = 17;
int num2 = 76;
// myString is set to 1776
string myString = num1.ToString() +
  num2.ToString();
```

Operators

An operator is a keyword or character that tells ASP.NET what kind of command to perform in an expression. The C# language (and the Razor syntax that's based on it) supports many operators, but you only need to recognize a few to get started. The following table summarizes the most common operators.

Operator

Description

Examples

`+` `-` `*` `/`

Math operators used in numerical expressions.

```
@(5 + 13)
@{ var netWorth = 150000; }
@{ var newTotal = netWorth * 2; }
@(newTotal / 2)
```

`=`

Assignment. Assigns the value on the right side of a statement to the object on the left side.

```
var age = 17;
```

`==`

Equality. Returns `true` if the values are equal. (Notice the distinction between the `=` operator and the `==` operator.)

```
var myNum = 15;
if (myNum == 15) {
    // Do something.
}
```

`!=`

Inequality. Returns `true` if the values are not equal.

```
var theNum = 13;
if (theNum != 15) {
    // Do something.
}
```

`< > <= >=`

Less-than, greater-than, less-than-or-equal, and greater-than-or-equal.

```
if (2 < 3) {
    // Do something.
}
var currentCount = 12;
if(currentCount >= 12) {
    // Do something.
}
```

`+`

Concatenation, which is used to join strings. ASP.NET knows the difference between this operator and the addition operator based on the data type of the expression.

```
// The displayed result is "abcdef".
@("abc" + "def")
```

`+=` `-=`

The increment and decrement operators, which add and subtract 1 (respectively) from a variable.

```
int theCount = 0;
theCount += 1; // Adds 1 to count
```

`.`

Dot. Used to distinguish objects and their properties and methods.

```
var myUrl = Request.Url;
var count = Request["Count"].AsInt();
```

`()`

Parentheses. Used to group expressions and to pass parameters to methods.

```
@(3 + 7)
@Request.MapPath(Request.FilePath);
```

`[]`

Brackets. Used for accessing values in arrays or collections.

```
var income = Request["AnnualIncome"];
```

`!`

Not. Reverses a `true` value to `false` and vice versa. Typically used as a shorthand way to test for `false` (that is, for not `true`).

```
bool taskCompleted = false;
// Processing.
if(!taskCompleted) {
    // Continue processing
}
```

`&&` `||`

Logical AND and OR, which are used to link conditions together.

```
bool myTaskCompleted = false;
int totalCount = 0;
// Processing.
if(!myTaskCompleted && totalCount < 12) {
    // Continue processing.
}
```

Working with File and Folder Paths in Code

You'll often work with file and folder paths in your code. Here is an example of physical folder structure for a website as it might appear on your development computer:

`C:\WebSites\MyWebSite default.cshtml datafile.txt \images Logo.jpg \styles Styles.css`

Here are some essential details about URLs and paths:

- A URL begins with either a domain name (`http://www.example.com`) or a server name (`http://localhost`, `http://mycomputer`).
- A URL corresponds to a physical path on a host computer. For example, `http://myserver` might correspond to the folder *C:\websites\mywebsite* on the server.
- A virtual path is shorthand to represent paths in code without having to specify the full path. It includes the portion of a URL that follows the domain or server name. When you use virtual paths, you can move your code to a different domain or server without having to update the paths.

Here's an example to help you understand the differences:

COMPLETE URL	`HTTP://MYCOMPANYSERVER/HUMANRESOURCES/COMPANYPOLICY.HTM`
Server name	*mycompanyserver*

COMPLETE URL	`HTTP://MYCOMPANYSERVER/HUMANRESOURCES/COMPANYPOLICY.HTM`
Virtual path	*/humanresources/CompanyPolicy.htm*
Physical path	*C:\mywebsites\humanresources\CompanyPolicy.htm*

The virtual root is /, just like the root of your C: drive is . (Virtual folder paths always use forward slashes.) The virtual path of a folder doesn't have to have the same name as the physical folder; it can be an alias. (On production servers, the virtual path rarely matches an exact physical path.)

When you work with files and folders in code, sometimes you need to reference the physical path and sometimes a virtual path, depending on what objects you're working with. ASP.NET gives you these tools for working with file and folder paths in code: the `Server.MapPath` method, and the `~` operator and `Href` method.

Converting virtual to physical paths: the Server.MapPath method

The `Server.MapPath` method converts a virtual path (like */default.cshtml*) to an absolute physical path (like *C:\WebSites\MyWebSiteFolder\default.cshtml*). You use this method any time you need a complete physical path. A typical example is when you're reading or writing a text file or image file on the web server.

You typically don't know the absolute physical path of your site on a hosting site's server, so this method can convert the path you do know — the virtual path — to the corresponding path on the server for you. You pass the virtual path to a file or folder to the method, and it returns the physical path:

```
@{
    var dataFilePath = "~/dataFile.txt";
}
<!-- Displays a physical path C:\Websites\MyWebSite\datafile.txt  -->
<p>@Server.MapPath(dataFilePath)</p>
```

Referencing the virtual root: the ~ operator and Href method

In a *.cshtml* or *.vbhtml* file, you can reference the virtual root path using the `~` operator. This is very handy because you can move pages around in a site, and any links they contain to other pages won't be broken. It's also handy in case you ever move your website to a different location. Here are some examples:

```
@{
    var myImagesFolder = "~/images";
    var myStyleSheet = "~/styles/StyleSheet.css";
}
```

If the website is `http://myserver/myapp`, here's how ASP.NET will treat these paths when the page runs:

- `myImagesFolder` : `http://myserver/myapp/images`
- `myStyleSheet` : `http://myserver/myapp/styles/Stylesheet.css`

(You won't actually see these paths as the values of the variable, but ASP.NET will treat the paths as if that's what they were.)

You can use the `~` operator both in server code (as above) and in markup, like this:

```
<!-- Examples of using the ~ operator in markup in ASP.NET Web Pages -->

<a href="~/Default">Home</a>
<img src="~/images/MyImages.png" />
```

In markup, you use the `~` operator to create paths to resources like image files, other web pages, and CSS files. When the page runs, ASP.NET looks through the page (both code and markup) and resolves all the `~` references to the appropriate path.

Conditional Logic and Loops

ASP.NET server code lets you perform tasks based on conditions and write code that repeats statements a specific number of times (that is, code that runs a loop).

Testing Conditions

To test a simple condition you use the `if` statement, which returns true or false based on a test you specify:

```
@{
  var showToday = true;
  if(showToday)
  {
    @DateTime.Today;
  }
}
```

The `if` keyword starts a block. The actual test (condition) is in parentheses and returns true or false. The statements that run if the test is true are enclosed in braces. An `if` statement can include an `else` block that specifies statements to run if the condition is false:

```
@{
  var showToday = false;
  if(showToday)
  {
    @DateTime.Today;
  }
  else
  {
    <text>Sorry!</text>
  }
}
```

You can add multiple conditions using an `else if` block:

```
@{
    var theBalance = 4.99;
    if(theBalance == 0)
    {
        <p>You have a zero balance.</p>
    }
    else if (theBalance  > 0 && theBalance <= 5)
    {
        <p>Your balance of $@theBalance is very low.</p>
    }
    else
    {
        <p>Your balance is: $@theBalance</p>
    }
}
```

In this example, if the first condition in the if block is not true, the `else if` condition is checked. If that condition is met, the statements in the `else if` block are executed. If none of the conditions are met, the statements in the `else` block are executed. You can add any number of else if blocks, and then close with an `else` block as the "everything else" condition.

To test a large number of conditions, use a `switch` block:

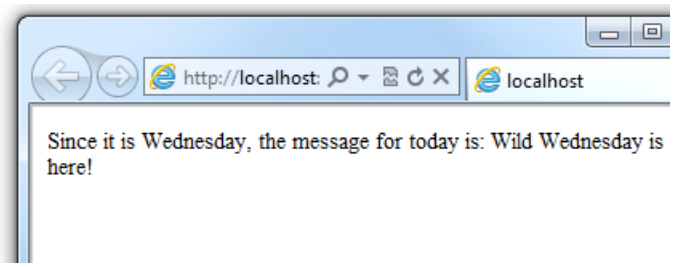
```
@{
    var weekday = "Wednesday";
    var greeting = "";

    switch(weekday)
    {
        case "Monday":
            greeting = "Ok, it's a marvelous Monday";
            break;
        case "Tuesday":
            greeting = "It's a tremendous Tuesday";
            break;
        case "Wednesday":
            greeting = "Wild Wednesday is here!";
            break;
        default:
            greeting = "It's some other day, oh well.";
            break;
    }

    <p>Since it is @weekday, the message for today is: @greeting</p>
}
```

The value to test is in parentheses (in the example, the `weekday` variable). Each individual test uses a `case` statement that ends with a colon (:). If the value of a `case` statement matches the test value, the code in that case block is executed. You close each case statement with a `break` statement. (If you forget to include break in each `case` block, the code from the next `case` statement will run also.) A `switch` block often has a `default` statement as the last case for an "everything else" option that runs if none of the other cases are true.

The result of the last two conditional blocks displayed in a browser:



Looping Code

You often need to run the same statements repeatedly. You do this by looping. For example, you often run the same statements for each item in a collection of data. If you know exactly how many times you want to loop, you can use a `for` loop. This kind of loop is especially useful for counting up or counting down:

```
@for(var i = 10; i < 21; i++)
{
    <p>Line #: @i</p>
}
```

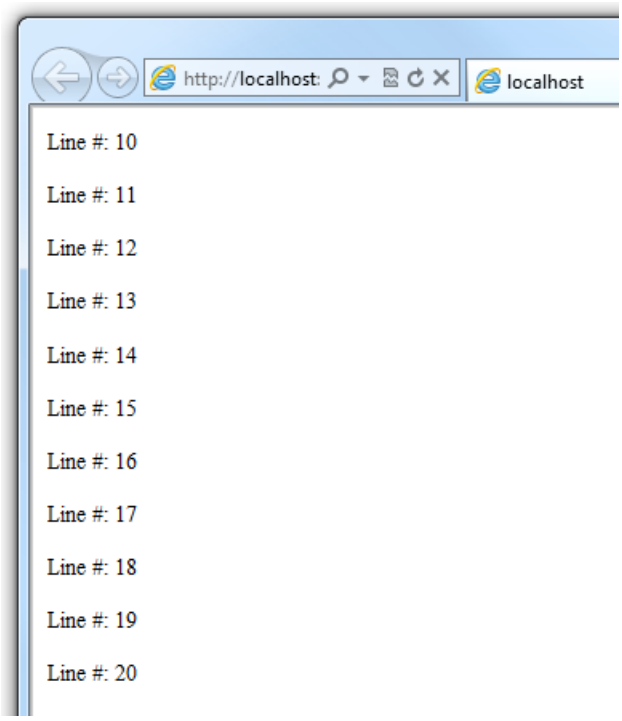
The loop begins with the `for` keyword, followed by three statements in parentheses, each terminated with a semicolon.

- Inside the parentheses, the first statement (`var i=10;`) creates a counter and initializes it to 10. You don't have to name the counter `i` — you can use any variable. When the `for` loop runs, the counter is automatically incremented.
- The second statement (`i < 21;`) sets the condition for how far you want to count. In this case, you want it to

go to a maximum of 20 (that is, keep going while the counter is less than 21).

- The third statement (`i++`) uses an increment operator, which simply specifies that the counter should have 1 added to it each time the loop runs.

Inside the braces is the code that will run for each iteration of the loop. The markup creates a new paragraph (`<p>` element) each time and adds a line to the output, displaying the value of `i` (the counter). When you run this page, the example creates 11 lines displaying the output, with the text in each line indicating the item number.

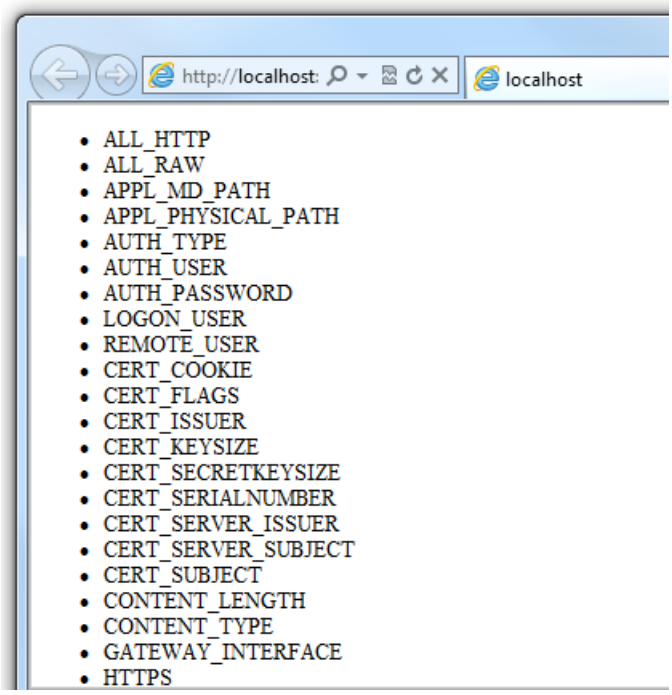


If you're working with a collection or array, you often use a `foreach` loop. A collection is a group of similar objects, and the `foreach` loop lets you carry out a task on each item in the collection. This type of loop is convenient for collections, because unlike a `for` loop, you don't have to increment the counter or set a limit. Instead, the `foreach` loop code simply proceeds through the collection until it's finished.

For example, the following code returns the items in the `Request.ServerVariables` collection, which is an object that contains information about your web server. It uses a `foreac` h loop to display the name of each item by creating a new `<li>` element in an HTML bulleted list.

```
<ul>
@foreach (var myItem in Request.ServerVariables)
{
    <li>@myItem</li>
}
</ul>
```

The `foreach` keyword is followed by parentheses where you declare a variable that represents a single item in the collection (in the example, `var item`), followed by the `in` keyword, followed by the collection you want to loop through. In the body of the `foreach` loop, you can access the current item using the variable that you declared earlier.



To create a more general-purpose loop, use the `while` statement:

```
@{
    var countNum = 0;
    while (countNum < 50)
    {
        countNum += 1;
        <p>Line #@countNum: </p>
    }
}
```

A `while` loop begins with the `while` keyword, followed by parentheses where you specify how long the loop continues (here, for as long as `countNum` is less than 50), then the block to repeat. Loops typically increment (add to) or decrement (subtract from) a variable or object used for counting. In the example, the `+=` operator adds 1 to `countNum` each time the loop runs. (To decrement a variable in a loop that counts down, you would use the decrement operator `-=`).

Objects and Collections

Nearly everything in an ASP.NET website is an object, including the web page itself. This section discusses some important objects you'll work with frequently in your code.

Page Objects

The most basic object in ASP.NET is the page. You can access properties of the page object directly without any qualifying object. The following code gets the page's file path, using the `Request` object of the page:

```
@{
    var path = Request.FilePath;
}
```

To make it clear that you're referencing properties and methods on the current page object, you can optionally use the keyword `this` to represent the page object in your code. Here is the previous code example, with `this` added to represent the page:

```
@{
    var path = this.Request.FilePath;
}
```

You can use properties of the `Page` object to get a lot of information, such as:

- `Request`. As you've already seen, this is a collection of information about the current request, including what type of browser made the request, the URL of the page, the user identity, etc.
- `Response`. This is a collection of information about the response (page) that will be sent to the browser when the server code has finished running. For example, you can use this property to write information into the response.

  ```
  @{
      // Access the page's Request object to retrieve the Url.
      var pageUrl = this.Request.Url;
  }
  <a href="@pageUrl">My page</a>
  ```

Collection Objects (Arrays and Dictionaries)

A *collection* is a group of objects of the same type, such as a collection of `Customer` objects from a database. ASP.NET contains many built-in collections, like the `Request.Files` collection.

You'll often work with data in collections. Two common collection types are the *array* and the *dictionary*. An array is useful when you want to store a collection of similar items but don't want to create a separate variable to hold each item:

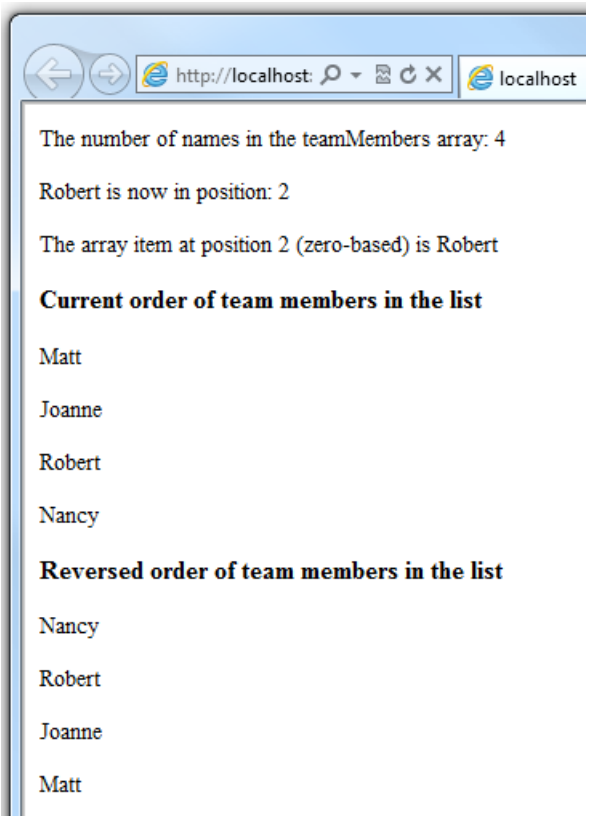
```
@* Array block 1: Declaring a new array using braces. *@
@{
    <h3>Team Members</h3>
    string[] teamMembers = {"Matt", "Joanne", "Robert", "Nancy"};
    foreach (var person in teamMembers)
    {
        <p>@person</p>
    }
}
```

With arrays, you declare a specific data type, such as `string`, `int`, or `DateTime`. To indicate that the variable can contain an array, you add brackets to the declaration (such as `string[]` or `int[]`). You can access items in an array using their position (index) or by using the `foreach` statement. Array indexes are zero-based — that is, the first item is at position 0, the second item is at position 1, and so on.

```
@{
    string[] teamMembers = {"Matt", "Joanne", "Robert", "Nancy"};
    <p>The number of names in the teamMembers array: @teamMembers.Length </p>
    <p>Robert is now in position: @Array.IndexOf(teamMembers, "Robert")</p>
    <p>The array item at position 2 (zero-based) is @teamMembers[2]</p>
    <h3>Current order of team members in the list</h3>
    foreach (var name in teamMembers)
    {
        <p>@name</p>
    }
    <h3>Reversed order of team members in the list</h3>
    Array.Reverse(teamMembers);
    foreach (var reversedItem in teamMembers)
    {
        <p>@reversedItem</p>
    }
}
```

You can determine the number of items in an array by getting its `Length` property. To get the position of a specific item in the array (to search the array), use the `Array.IndexOf` method. You can also do things like reverse the contents of an array (the `Array.Reverse` method) or sort the contents (the `Array.Sort` method).

The output of the string array code displayed in a browser:



A dictionary is a collection of key/value pairs, where you provide the key (or name) to set or retrieve the corresponding value:

```
@{
    var myScores = new Dictionary<string, int>();
    myScores.Add("test1", 71);
    myScores.Add("test2", 82);
    myScores.Add("test3", 100);
    myScores.Add("test4", 59);
}
<p>My score on test 3 is: @myScores["test3"]%</p>
@{myScores["test4"] = 79;}
<p>My corrected score on test 4 is: @myScores["test4"]%</p>
```

To create a dictionary, you use the `new` keyword to indicate that you're creating a new dictionary object. You can assign a dictionary to a variable using the `var` keyword. You indicate the data types of the items in the dictionary using angle brackets (`< >`). At the end of the declaration, you must add a pair of parentheses, because this is actually a method that creates a new dictionary.

To add items to the dictionary, you can call the `Add` method of the dictionary variable (`myScores` in this case), and then specify a key and a value. Alternatively, you can use square brackets to indicate the key and do a simple assignment, as in the following example:

```
myScores["test4"] = 79;
```

To get a value from the dictionary, you specify the key in brackets:

```
var testScoreThree = myScores["test3"];
```

Calling Methods with Parameters

As you read earlier in this article, the objects that you program with can have methods. For example, a `Database` object might have a `Database.Connect` method. Many methods also have one or more parameters. A *parameter* is a value that you pass to a method to enable the method to complete its task. For example, look at a declaration for the `Request.MapPath` method, which takes three parameters:

```
public string MapPath(string virtualPath, string baseVirtualDir,
    bool allowCrossAppMapping);
```

(The line has been wrapped to make it more readable. Remember that you can put line breaks almost any place except inside strings that are enclosed in quotation marks.)

This method returns the physical path on the server that corresponds to a specified virtual path. The three parameters for the method are `virtualPath`, `baseVirtualDir`, and `allowCrossAppMapping`. (Notice that in the declaration, the parameters are listed with the data types of the data that they'll accept.) When you call this method, you must supply values for all three parameters.

The Razor syntax gives you two options for passing parameters to a method: *positional parameters* and *named parameters*. To call a method using positional parameters, you pass the parameters in a strict order that's specified in the method declaration. (You would typically know this order by reading documentation for the method.) You must follow the order, and you can't skip any of the parameters — if necessary, you pass an empty string (`""`) or `null` for a positional parameter that you don't have a value for.

The following example assumes you have a folder named *scripts* on your website. The code calls the `Request.MapPath` method and passes values for the three parameters in the correct order. It then displays the resulting mapped path.

```
@{
    // Pass parameters to a method using positional parameters.
    var myPathPositional = Request.MapPath("/scripts", "/", true);
}
<p>@myPathPositional</p>
```

When a method has many parameters, you can keep your code more readable by using named parameters. To call a method using named parameters, you specify the parameter name followed by a colon (:), and then the value. The advantage of named parameters is that you can pass them in any order you want. (A disadvantage is that the method call is not as compact.)

The following example calls the same method as above, but uses named parameters to supply the values:

```
@{
    // Pass parameters to a method using named parameters.
    var myPathNamed = Request.MapPath(baseVirtualDir: "/",
        allowCrossAppMapping: true, virtualPath: "/scripts");
}
<p>@myPathNamed</p>
```

As you can see, the parameters are passed in a different order. However, if you run the previous example and this example, they'll return the same value.

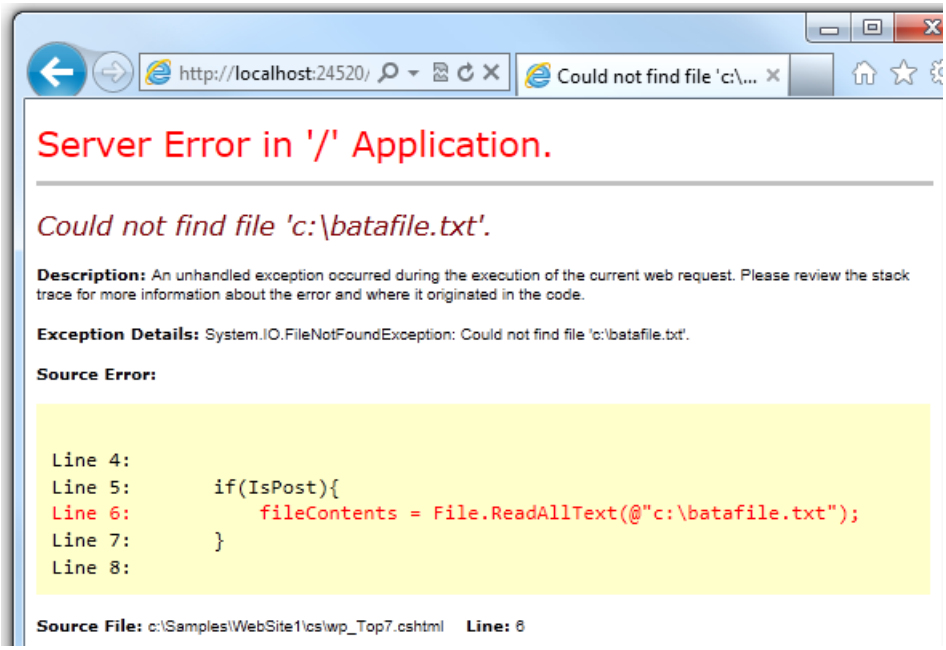
Handling Errors

Try-Catch Statements

You'll often have statements in your code that might fail for reasons outside your control. For example:

- If your code tries to create or access a file, all sorts of errors might occur. The file you want might not exist, it might be locked, the code might not have permissions, and so on.
- Similarly, if your code tries to update records in a database, there can be permissions issues, the connection to the database might be dropped, the data to save might be invalid, and so on.

In programming terms, these situations are called *exceptions*. If your code encounters an exception, it generates (throws) an error message that's, at best, annoying to users:



In situations where your code might encounter exceptions, and in order to avoid error messages of this type, you

can use `try/catch` statements. In the `try` statement, you run the code that you're checking. In one or more `catch` statements, you can look for specific errors (specific types of exceptions) that might have occurred. You can include as many `catch` statements as you need to look for errors that you are anticipating.

> **NOTE**
> We recommend that you avoid using the `Response.Redirect` method in `try/catch` statements, because it can cause an exception in your page.

The following example shows a page that creates a text file on the first request and then displays a button that lets the user open the file. The example deliberately uses a bad file name so that it will cause an exception. The code includes `catch` statements for two possible exceptions: `FileNotFoundException`, which occurs if the file name is bad, and `DirectoryNotFoundException`, which occurs if ASP.NET can't even find the folder. (You can uncomment a statement in the example in order to see how it runs when everything works properly.)

If your code didn't handle the exception, you would see an error page like the previous screen shot. However, the `try/catch` section helps prevent the user from seeing these types of errors.

```
@{
    var dataFilePath = "~/dataFile.txt";
    var fileContents = "";
    var physicalPath = Server.MapPath(dataFilePath);
    var userMessage = "Hello world, the time is " + DateTime.Now;
    var userErrMsg = "";
    var errMsg = "";

    if(IsPost)
    {
        // When the user clicks the "Open File" button and posts
        // the page, try to open the created file for reading.
        try {
            // This code fails because of faulty path to the file.
            fileContents = File.ReadAllText(@"c:\batafile.txt");

            // This code works. To eliminate error on page,
            // comment the above line of code and uncomment this one.
            //fileContents = File.ReadAllText(physicalPath);
        }
        catch (FileNotFoundException ex) {
            // You can use the exception object for debugging, logging, etc.
            errMsg = ex.Message;
            // Create a friendly error message for users.
            userErrMsg = "A file could not be opened, please contact "
                + "your system administrator.";
        }
        catch (DirectoryNotFoundException ex) {
            // Similar to previous exception.
            errMsg = ex.Message;
            userErrMsg = "A directory was not found, please contact "
                + "your system administrator.";
        }
    }
    else
    {
        // The first time the page is requested, create the text file.
        File.WriteAllText(physicalPath, userMessage);
    }
}

<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Try-Catch Statements</title>
    </head>
    <body>
    <form method="POST" action="" >
      <input type="Submit" name="Submit" value="Open File"/>
    </form>

    <p>@fileContents</p>
    <p>@userErrMsg</p>

    </body>
</html>
```

Additional Resources

Programming with Visual Basic

Appendix: Visual Basic Language and Syntax

Reference Documentation

ASP.NET

C# Language

Introduction to ASP.NET Web Programming Using the Razor Syntax (Visual Basic)

2/20/2020 • 36 minutes to read • Edit Online

by Tom FitzMacken

This article gives you an overview of programming with ASP.NET Web Pages using the Razor syntax and Visual Basic. ASP.NET is Microsoft's technology for running dynamic web pages on web servers.

What you'll learn:

- The top 8 programming tips for getting started with programming ASP.NET Web Pages using Razor syntax.
- Basic programming concepts you'll need.
- What ASP.NET server code and the Razor syntax is all about.

Software versions

- ASP.NET Web Pages (Razor) 3

This tutorial also works with ASP.NET Web Pages 2.

Most examples of using ASP.NET Web Pages with Razor syntax use C#. But the Razor syntax also supports Visual Basic. To program an ASP.NET web page in Visual Basic, you create a web page with a *.vbhtml* filename extension, and then add Visual Basic code. This article gives you an overview of working with the Visual Basic language and syntax to create ASP.NET Webpages.

> **NOTE**
>
> The default website templates for Microsoft WebMatrix (**Bakery**, **Photo Gallery**, and **Starter Site**, etc.) are available in C# and Visual Basic versions. You can install the Visual Basic templates by as NuGet packages. Website templates are installed in the root folder of your site in a folder named *Microsoft Templates*.

The Top 8 Programming Tips

This section lists a few tips that you absolutely need to know as you start writing ASP.NET server code using the Razor syntax.

1. You add code to a page using the @ character

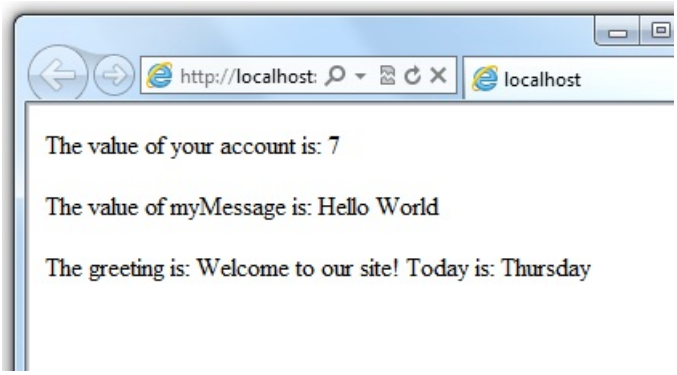
The `@` character starts inline expressions, single-statement blocks, and multi-statement blocks:

```
<!-- Single statement blocks  -->
@Code  Dim total = 7  End Code
@Code  Dim myMessage = "Hello World" End Code

<!-- Inline expressions -->
<p>The value of your account is: @total </p>
<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->
@Code
    Dim greeting = "Welcome to our site!"
    Dim weekDay = DateTime.Now.DayOfWeek
    Dim greetingMessage = greeting & " Today is: " & weekDay.ToString()
End Code
<p>The greeting is: @greetingMessage</p>
```

The result displayed in a browser:



TIP

HTML Encoding

When you display content in a page using the `@` character, as in the preceding examples, ASP.NET HTML-encodes the output. This replaces reserved HTML characters (such as `<` and `>` and `&`) with codes that enable the characters to be displayed as characters in a web page instead of being interpreted as HTML tags or entities. Without HTML encoding, the output from your server code might not display correctly, and could expose a page to security risks.

If your goal is to output HTML markup that renders tags as markup (for example `<p></p>` for a paragraph or `<em></em>` to emphasize text), see the section Combining Text, Markup, and Code in Code Blocks later in this article.

You can read more about HTML encoding in Working with HTML Forms in ASP.NET Web Pages Sites.

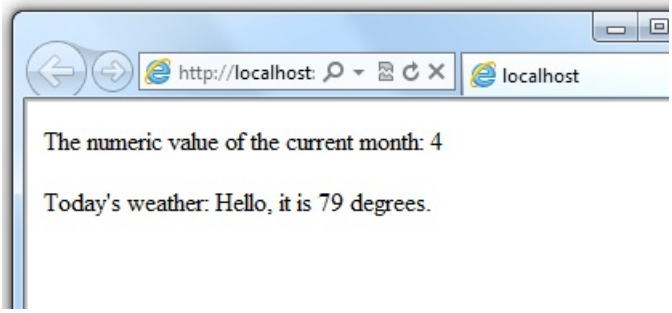
2. You enclose code blocks with Code...End Code

A code block includes one or more code statements and is enclosed with the keywords `Code` and `End Code`. Place the opening `Code` keyword immediately after the `@` character — there can't be whitespace between them.

```
<!-- Single statement block.  -->
@Code
    Dim theMonth = DateTime.Now.Month
End Code
<p>The numeric value of the current month: @theMonth</p>

<!-- Multi-statement block. -->
@Code
    Dim outsideTemp = 79
    Dim weatherMessage = "Hello, it is " & outsideTemp & " degrees."
End Code
<p>Today's weather: @weatherMessage</p>
```

The result displayed in a browser:



3. Inside a block, you end each code statement with a line break

In a Visual Basic code block, each statement ends with a line break. (Later in the article you'll see a way to wrap a long code statement into multiple lines if needed.)

```
<!-- Single statement block. -->
@Code
    Dim theMonth = DateTime.Now.Month
End Code

<!-- Multi-statement block. -->
@Code
    Dim outsideTemp = 79
    Dim weatherMessage = "Hello, it is " & outsideTemp & " degrees."
End Code

<!-- An inline expression, so no line break needed. -->
<p>Today's weather: @weatherMessage</p>
```

4. You use variables to store values

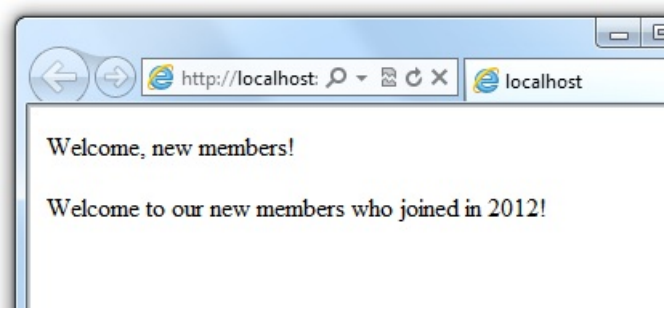
You can store values in a *variable*, including strings, numbers, and dates, etc. You create a new variable using the `Dim` keyword. You can insert variable values directly in a page using `@`.

```
<!-- Storing a string -->
@Code
    Dim welcomeMessage = "Welcome, new members!"
End Code
<p>@welcomeMessage</p>

<!-- Storing a date -->
@Code
    Dim year = DateTime.Now.Year
End Code

<!-- Displaying a variable -->
<p>Welcome to our new members who joined in @year!</p>
```

The result displayed in a browser:



5. You enclose literal string values in double quotation marks

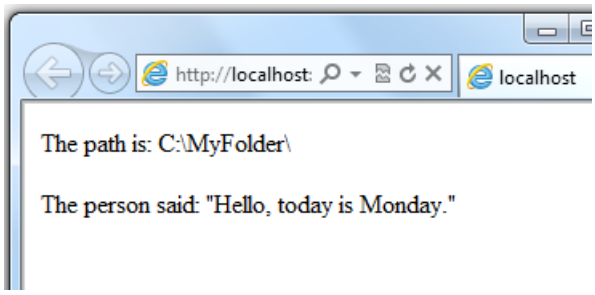
A *string* is a sequence of characters that are treated as text. To specify a string, you enclose it in double quotation marks:

```
@Code
    Dim myString = "This is a string literal"
End Code
```

To embed double quotation marks within a string value, insert two double quotation mark characters. If you want the double quotation character to appear once in the page output, enter it as `""` within the quoted string, and if you want it to appear twice, enter it as `""""` within the quoted string.

```
<!-- Embedding double quotation marks in a string -->
@Code
    Dim myQuote = "The person said: ""Hello, today is Monday."""
End Code
<p>@myQuote</p>
```

The result displayed in a browser:



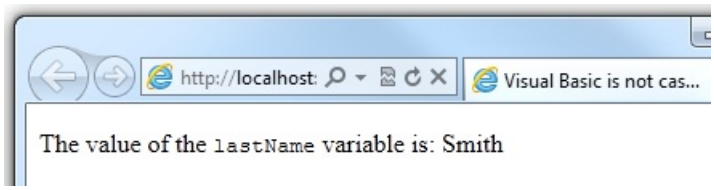
6. Visual Basic code is not case sensitive

The Visual Basic language is not case sensitive. Programming keywords (like `Dim`, `If`, and `True`) and variable names (like `myString`, or `subTotal`) can be written in any case.

The following lines of code assign a value to the variable `lastname` using a lowercase name, and then output the variable value to the page using an uppercase name.

```
@Code
    Dim lastName = "Smith"
    ' Keywords like dim are also not case sensitive.
    DIM someNumber = 7
End Code
<p>The value of the <code>lastName</code> variable is: @LASTNAME</p>
```

The result displayed in a browser:



7. Much of your coding involves working with objects

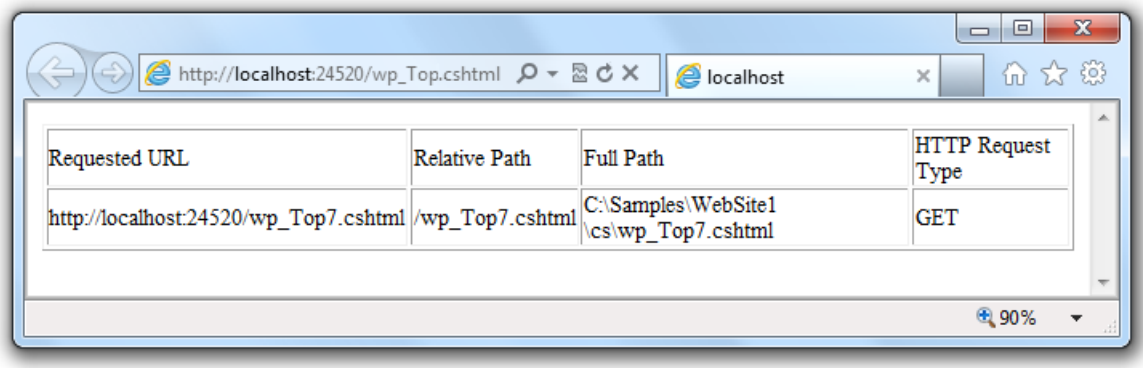
An object represents a thing that you can program with — a page, a text box, a file, an image, a web request, an email message, a customer record (database row), etc. Objects have properties that describe their characteristics — a text box object has a `Text` property, a request object has a `Url` property, an email message has a `From` property, and a customer object has a `FirstName` property. Objects also have methods that are the "verbs" they

can perform. Examples include a file object's `Save` method, an image object's `Rotate` method, and an email object's `Send` method.

You'll often work with the `Request` object, which gives you information like the values of form fields on the page (text boxes, etc.), what type of browser made the request, the URL of the page, the user identity, etc. This example shows how to access properties of the `Request` object and how to call the `MapPath` method of the `Request` object, which gives you the absolute path of the page on the server:

```
<table border="1">
    <tr>
        <td>Requested URL</td>
        <td>Relative Path</td>
        <td>Full Path</td>
        <td>HTTP Request Type</td>
    </tr>
    <tr>
        <td>@Request.Url</td>
        <td>@Request.FilePath</td>
        <td>@Request.MapPath(Request.FilePath)</td>
        <td>@Request.RequestType</td>
    </tr>
</table>
```

The result displayed in a browser:



8. You can write code that makes decisions

A key feature of dynamic web pages is that you can determine what to do based on conditions. The most common way to do this is with the `If` statement (and optional `Else` statement).

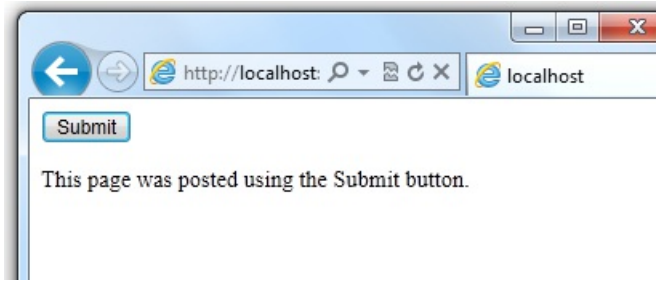
```
@Code
   Dim result = ""
   If IsPost Then
      result = "This page was posted using the Submit button."
   Else
      result = "This was the first request for this page."
   End If
End Code
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Write Code that Makes Decisions</title>
    </head>
<body>

    <form method="POST" action="" >
        <input type="Submit" name="Submit" value="Submit"/>
        <p>@result</p>
    </form>

</body>
</html>
```

The statement `If IsPost` is a shorthand way of writing `If IsPost = True`. Along with `If` statements, there are a variety of ways to test conditions, repeat blocks of code, and so on, which are described later in this article.

The result displayed in a browser (after clicking **Submit**):



TIP

HTTP GET and POST Methods and the IsPost Property

The protocol used for web pages (HTTP) supports a very limited number of methods ("verbs") that are used to make requests to the server. The two most common ones are GET, which is used to read a page, and POST, which is used to submit a page. In general, the first time a user requests a page, the page is requested using GET. If the user fills in a form and then clicks **Submit**, the browser makes a POST request to the server.

In web programming, it's often useful to know whether a page is being requested as a GET or as a POST so that you know how to process the page. In ASP.NET Web Pages, you can use the `IsPost` property to see whether a request is a GET or a POST. If the request is a POST, the `IsPost` property will return true, and you can do things like read the values of text boxes on a form. Many examples you'll see show you how to process the page differently depending on the value of `IsPost`.

A Simple Code Example

This procedure shows you how to create a page that illustrates basic programming techniques. In the example, you create a page that lets users enter two numbers, then it adds them and displays the result.

1. In your editor, create a new file and name it *AddNumbers.vbhtml*.

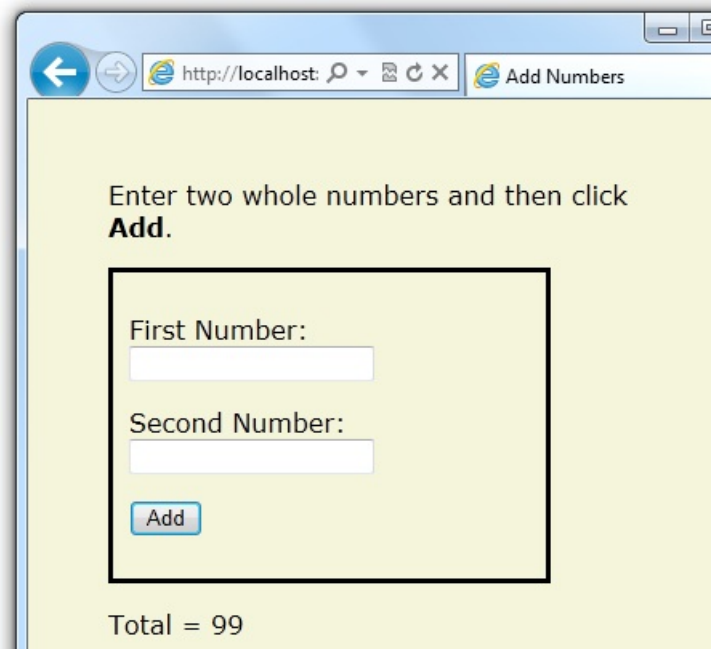
2. Copy the following code and markup into the page, replacing anything already in the page.

```
@Code
    Dim total = 0
    Dim totalMessage = ""
    if IsPost Then
        ' Retrieve the numbers that the user entered.
        Dim num1 = Request("text1")
        Dim num2 = Request("text2")
        ' Convert the entered strings into integers numbers and add.
        total = num1.AsInt() + num2.AsInt()
        totalMessage = "Total = " & total
    End If
End Code
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Adding Numbers</title>
        <style type="text/css">
            body {background-color: beige; font-family: Verdana, Ariel;
                margin: 50px;
                }
            form {padding: 10px; border-style: solid; width: 250px;}
        </style>
    </head>
<body>
    <p>Enter two whole numbers and click <strong>Add</strong> to display the result.</p>
    <p></p>
    <form action="" method="post">
    <p><label for="text1">First Number:</label>
    <input type="text" name="text1" />
    </p>
    <p><label for="text2">Second Number:</label>
    <input type="text" name="text2" />
    </p>
    <p><input type="submit" value="Add" /></p>
    </form>
    <p>@totalMessage</p>
</body>
</html>
```

Here are some things for you to note:

- The `@` character starts the first block of code in the page, and it precedes the `totalMessage` variable embedded near the bottom.
- The block at the top of the page is enclosed in `Code...End Code`.
- The variables `total`, `num1`, `num2`, and `totalMessage` store several numbers and a string.
- The literal string value assigned to the `totalMessage` variable is in double quotation marks.
- Because Visual Basic code is not case sensitive, when the `totalMessage` variable is used near the bottom of the page, its name only needs to match the spelling of the variable declaration at the top of the page. The casing doesn't matter.
- The expression `num1.AsInt()` + `num2.AsInt()` shows how to work with objects and methods. The `AsInt` method on each variable converts the string entered by a user to a whole number (an integer) that can be added.
- The `<form>` tag includes a `method="post"` attribute. This specifies that when the user clicks **Add**, the page will be sent to the server using the HTTP POST method. When the page is submitted, the code `If IsPost` evaluates to true and the conditional code runs, displaying the result of adding the numbers.

3. Save the page and run it in a browser. (Make sure the page is selected in the **Files** workspace before you run it.) Enter two whole numbers and then click the **Add** button.



Visual Basic Language and Syntax

Earlier you saw a basic example of how to create an ASP.NET web page, and how you can add server code to HTML markup. Here you'll learn the basics of using Visual Basic to write ASP.NET server code using the Razor syntax — that is, the programming language rules.

If you're experienced with programming (especially if you've used C, C++, C#, Visual Basic, or JavaScript), much of what you read here will be familiar. You'll probably need to familiarize yourself only with how WebMatrix code is added to markup in *.vbhtml* files.

Combining text, markup, and code in code blocks

In server code blocks, you'll often want to output text and markup to the page. If a server code block contains text that's not code and that instead should be rendered as is, ASP.NET needs to be able to distinguish that text from code. There are several ways to do this.

- Enclose the text in an HTML block element like `<p></p>` or `<em></em>`:

  ```
  @If IsPost Then
      ' This line has all content between matched <p> tags.
      @<p>Hello, the time is @DateTime.Now and this page is a postback!</p>
  Else
      ' All content between matched tags, followed by server code.
      @<p>Hello, <em>Stranger!</em> today is: </p> @DateTime.Now
  End If
  ```

  The HTML element can include text, additional HTML elements, and server-code expressions. When ASP.NET sees the opening HTML tag (for example, `<p>`), it renders everything the element and its content as is to the browser (and resolves the server-code expressions).

- Use the `@:` operator or the `<text>` element. The `@:` outputs a single line of content containing plain text or unmatched HTML tags; the `<text>` element encloses multiple lines to output. These options are useful when you don't want to render an HTML element as part of the output.

```
@If IsPost Then
    ' Plain text followed by an unmatched HTML tag and server code.
    @:The time is: <br /> @DateTime.Now
    ' Server code and then plain text, matched tags, and more text.
    @DateTime.Now @:is the <em>current</em> time.
End If
```

The following example repeats the previous example but uses a single pair of `<text>` tags to enclose the text to render.

```
@If IsPost Then
    @<text>
    The time is: <br /> @DateTime.Now
    @DateTime.Now is the <em>current</em> time.
    </text>
End If
```

In the following example, the `<text>` and `</text>` tags enclose three lines, all of which have some uncontained text and unmatched HTML tags (`<br />`), along with server code and matched HTML tags. Again, you could also precede each line individually with the `@:` operator; either way works.

```
@Code
    dim minTemp = 75
    @<text>It is the month of @DateTime.Now.ToString("MMMM"), and
    it's a <em>great</em> day! <p>You can go swimming if it's at
    least @minTemp degrees.</p></text>
End Code
```

> **NOTE**
>
> When you output text as shown in this section — using an HTML element, the `@:` operator, or the `<text>` element — ASP.NET doesn't HTML-encode the output. (As noted earlier, ASP.NET does encode the output of server code expressions and server code blocks that are preceded by `@`, except in the special cases noted in this section.)

Whitespace

Extra spaces in a statement (and outside of a string literal) don't affect the statement:

```
@Code Dim personName =    "Smith"    End Code
```

Breaking long statements into multiple lines

You can break a long code statement into multiple lines by using the underscore character `_` (which in Visual Basic is called the *continuation character*) after each line of code. To break a statement onto the next line, at the end of the line add a space and then the continuation character. Continue the statement on the next line. You can wrap statements onto as many lines as you need to improve readability. The following statements are the same:

```
@Code
    Dim familyName _
    =  "Smith"
End Code

@Code
    Dim _
    theName _
    = _
    "Smith"
End Code
```

However, you can't wrap a line in the middle of a string literal. The following example doesn't work:

```
@Code
    ' Doesn't work.
    Dim test = "This is a long _
      string"
End Code
```

To combine a long string that wraps to multiple lines like the above code, you would need to use the *concatenation operator* (`&`), which you'll see later in this article.

Code comments

Comments let you leave notes for yourself or others. Razor syntax comments are prefixed with `@*` and end with `*@`.

```
@* A single-line comment is added like this example. *@

@*
    This is a multiline code comment.
    It can continue for any number of lines.
*@
```

Within code blocks you can use the Razor syntax comments, or you can use ordinary Visual Basic comment character, which is a single quote (`'`) prefixed to each line.

```
@Code
    ' You can make comments in blocks by just using ' before each line.
End Code

@Code
    ' There is no multi-line comment character in Visual Basic.
    ' You use a ' before each line you want to comment.
End Code
```

Variables

A variable is a named object that you use to store data. You can name variables anything, but the name must begin with an alphabetic character and it cannot contain whitespace or reserved characters. In Visual Basic, as you saw earlier, the case of the letters in a variable name doesn't matter.

Variables and data types

A variable can have a specific data type, which indicates what kind of data is stored in the variable. You can have string variables that store string values (like "Hello world"), integer variables that store whole-number values (like 3 or 79), and date variables that store date values in a variety of formats (like 4/12/2012 or March 2009). And

there are many other data types you can use.

However, you don't have to specify a type for a variable. In most cases ASP.NET can figure out the type based on how the data in the variable is being used. (Occasionally you must specify a type; you'll see examples where this is true.)

To declare a variable without specifying a type, use `Dim` plus the variable name (for instance, `Dim myVar`). To declare a variable with a type, use `Dim` plus the variable name, followed by `As` and then the type name (for instance, `Dim myVar As String`).

```
@Code
    ' Assigning a string to a variable.
    Dim greeting = "Welcome"

    ' Assigning a number to a variable.
    Dim theCount = 3

    ' Assigning an expression to a variable.
    Dim monthlyTotal = theCount + 5

    ' Assigning a date value to a variable.
    Dim today = DateTime.Today

    ' Assigning the current page's URL to a variable.
    Dim myPath = Request.Url

    ' Declaring variables using explicit data types.
    Dim name as String = "Joe"
    Dim count as Integer = 5
    Dim tomorrow as DateTime = DateTime.Now.AddDays(1)
End Code
```

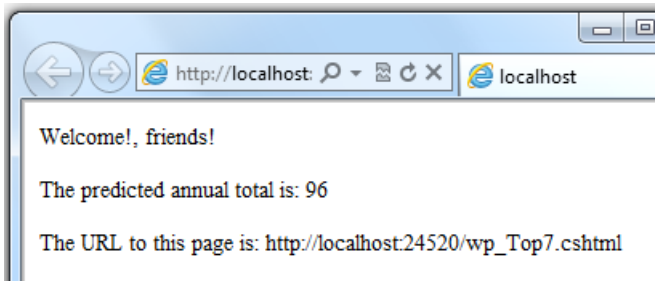
The following example shows some inline expressions that use the variables in a web page.

```
@Code
    ' Embedding the value of a variable into HTML markup.
    ' Precede the markup with @ because we are in a code block.
    @<p>@greeting, friends!</p>
End Code

<!-- Using a variable with an inline expression in HTML. -->
<p>The predicted annual total is: @( monthlyTotal * 12)</p>

<!--  Displaying the page URL with a variable. -->
<p>The URL to this page is:  @myPath</p>
```

The result displayed in a browser:



Converting and testing data types

Although ASP.NET can usually determine a data type automatically, sometimes it can't. Therefore, you might need to help ASP.NET out by performing an explicit conversion. Even if you don't have to convert types, sometimes it's helpful to test to see what type of data you might be working with.

The most common case is that you have to convert a string to another type, such as to an integer or date. The following example shows a typical case where you must convert a string to a number.

```
@Code
    Dim total = 0
    Dim totalMessage = ""
    if IsPost Then
        ' Retrieve the numbers that the user entered.
        Dim num1 = Request("text1")
        Dim num2 = Request("text2")
        ' Convert the entered strings into integers numbers and add.
        total = num1.AsInt() + num2.AsInt()
        totalMessage = "Total = " & total
    End If
End Code
```

As a rule, user input comes to you as strings. Even if you've prompted the user to enter a number, and even if they've entered a digit, when user input is submitted and you read it in code, the data is in string format. Therefore, you must convert the string to a number. In the example, if you try to perform arithmetic on the values without converting them, the following error results, because ASP.NET cannot add two strings:

`Cannot implicitly convert type 'string' to 'int'.`

To convert the values to integers, you call the `AsInt` method. If the conversion is successful, you can then add the numbers.

The following table lists some common conversion and test methods for variables.

Method	Description	Example

`AsInt(), IsInt()`	Converts a string that represents a whole number (like "593") to an integer.	`Dim myIntNumber = 0`
		`Dim myStringNum = "539"`
		`If myStringNum.IsInt() Then`
		`  myIntNumber = myStringNum.AsInt()`
		`End If`

`AsBool(), IsBool()`	Converts a string like "true" or "false" to a Boolean type.	`Dim myStringBool = "True"` `Dim myVar = myStringBool.AsBool()`
`AsFloat(), IsFloat()`	Converts a string that has a decimal value like "1.3" or "7.439" to a floating-point number.	`Dim myStringFloat = "41.432895"` `Dim myFloatNum = myStringFloat.AsFloat()`

`AsDecimal(), IsDecimal()`

Converts a string that has a decimal value like "1.3" or "7.439" to a decimal number. (In ASP.NET, a decimal number is more precise than a floating-point number.)

```
Dim myStringDec = "10317.425"
Dim myDecNum = myStringDec.AsDecimal()
```

`AsDateTime(), IsDateTime()`

Converts a string that represents a date and time value to the ASP.NET `DateTime` type.

```
Dim myDateString = "12/27/2012"
Dim newDate = myDateString.AsDateTime()
```

`ToString()`

Converts any other data type to a string.

```
Dim num1 As Integer = 17
Dim num2 As Integer = 76

' myString is set to 1776
Dim myString as String = num1.ToString() & _
    num2.ToString()
```

Operators

An operator is a keyword or character that tells ASP.NET what kind of command to perform in an expression. Visual Basic supports many operators, but you only need to recognize a few to get started developing ASP.NET web pages. The following table summarizes the most common operators.

Operator

Description

Examples

`+ - * /`

Math operators used in numerical expressions.

```
@(5 + 13)

Dim netWorth = 150000
Dim newTotal = netWorth * 2
@(newTotal / 2)
```

`=`

Assignment and equality. Depending on context, either assigns the value on the right side of a statement to the object on the left side, or checks the values for equality.

```
Dim age = 17

Dim income = Request("AnnualIncome")
```

`<>`

Inequality. Returns `True` if the values are not equal.

```
Dim theNum = 13
If theNum <> 15 Then
    ' Do something.
End If
```

`< > <= >=`

Less than, greater than, less than or equal, and greater than or equal.

```
If 2 < 3 Then
    ' Do something.
End If

Dim currentCount = 12
If currentCount >= 12 Then
    ' Do something.
End If
```

`&`

Concatenation, which is used to join strings.

```
' The displayed result is "abcdef".
@("abc" & "def")
```

`+= -=`

The increment and decrement operators, which add and subtract 1 (respectively) from a variable.

```
Dim theCount As Integer = 0
theCount += 1 ' Adds 1 to count
```

`.`

Dot. Used to distinguish objects and their properties and methods.

```
Dim myUrl = Request.Url
Dim count = Request("Count").AsInt()
```

`()`

Parentheses. Used to group expressions, to pass parameters to methods, and to access members of arrays and collections.

```
@(3 + 7)

@Request.MapPath(Request.FilePath)
```

`Not`

Not. Reverses a true value to false and vice versa. Typically used as a shorthand way to test for `False` (that is, for not `True`).

```
Dim taskCompleted As Boolean = False
' Processing.
If Not taskCompleted Then
    ' Continue processing
End If
```

`AndAlso OrElse`

Logical AND and OR, which are used to link conditions together.

```
Dim myTaskCompleted As Boolean = false
Dim totalCount As Integer = 0
' Processing.
If (Not myTaskCompleted) AndAlso _
         totalCount < 12 Then
    ' Continue processing.
End If
```

Working with File and Folder Paths in Code

You'll often work with file and folder paths in your code. Here is an example of physical folder structure for a website as it might appear on your development computer:

`C:\WebSites\MyWebSite default.cshtml datafile.txt \images Logo.jpg \styles Styles.css`

Here are some essential details about URLs and paths:

- A URL begins with either a domain name (`http://www.example.com`) or a server name (`http://localhost`, `http://mycomputer`).
- A URL corresponds to a physical path on a host computer. For example, `http://myserver` might correspond to the folder *C:\websites\mywebsite* on the server.
- A virtual path is shorthand to represent paths in code without having to specify the full path. It includes the portion of a URL that follows the domain or server name. When you use virtual paths, you can move your code to a different domain or server without having to update the paths.

Here's an example to help you understand the differences:

COMPLETE URL	`HTTP://MYCOMPANYSERVER/HUMANRESOURCES/COMPANYPOLICY.HTM`
Server name	*mycompanyserver*
Virtual path	*/humanresources/CompanyPolicy.htm*
Physical path	*C:\mywebsites\humanresources\CompanyPolicy.htm*

The virtual root is /, just like the root of your C: drive is . (Virtual folder paths always use forward slashes.) The virtual path of a folder doesn't have to have the same name as the physical folder; it can be an alias. (On production servers, the virtual path rarely matches an exact physical path.)

When you work with files and folders in code, sometimes you need to reference the physical path and sometimes a virtual path, depending on what objects you're working with. ASP.NET gives you these tools for working with file and folder paths in code: the `Server.MapPath` method, and the `~` operator and `Href` method.

Converting virtual to physical paths: the Server.MapPath method

The `Server.MapPath` method converts a virtual path (like */default.cshtml*) to an absolute physical path (like *C:\WebSites\MyWebSiteFolder\default.cshtml*). You use this method any time you need a complete physical path. A typical example is when you're reading or writing a text file or image file on the web server.

You typically don't know the absolute physical path of your site on a hosting site's server, so this method can convert the path you do know — the virtual path — to the corresponding path on the server for you. You pass the virtual path to a file or folder to the method, and it returns the physical path:

```
@Code
    Dim dataFilePath = "~/dataFile.txt"
End Code

<!-- Displays a physical path C:\Websites\MyWebSite\datafile.txt  -->
<p>@Server.MapPath(dataFilePath)</p>
```

Referencing the virtual root: the ~ operator and Href method

In a *.cshtml* or *.vbhtml* file, you can reference the virtual root path using the `~` operator. This is very handy because you can move pages around in a site, and any links they contain to other pages won't be broken. It's also handy in case you ever move your website to a different location. Here are some examples:

```
@Code
    Dim myImagesFolder = "~/images"
    Dim myStyleSheet = "~/styles/StyleSheet.css"
End Code
```

If the website is `http://myserver/myapp`, here's how ASP.NET will treat these paths when the page runs:

- `myImagesFolder` : `http://myserver/myapp/images`
- `myStyleSheet` : `http://myserver/myapp/styles/Stylesheet.css`

(You won't actually see these paths as the values of the variable, but ASP.NET will treat the paths as if that's what they were.)

You can use the `~` operator both in server code (as above) and in markup, like this:

```
<!-- Examples of using the ~ operator in markup in ASP.NET Web Pages -->

<a href="~/Default">Home</a>
<img src="~/images/MyImages.png" />
```

In markup, you use the `~` operator to create paths to resources like image files, other web pages, and CSS files. When the page runs, ASP.NET looks through the page (both code and markup) and resolves all the `~` references to the appropriate path.

Conditional Logic and Loops

ASP.NET server code lets you perform tasks based on conditions and write code that repeats statements a specific number of times that is, code that runs a loop).

Testing conditions

To test a simple condition you use the `If...Then` statement, which returns `True` or `False` based on a test you specify:

```
@Code
    Dim showToday = True
    If showToday Then
        DateTime.Today
    End If
End Code
```

The `If` keyword starts a block. The actual test (condition) follows the `If` keyword and returns true or false. The `If` statement ends with `Then`. The statements that will run if the test is true are enclosed by `If` and `End If`. An `If` statement can include an `Else` block that specifies statements to run if the condition is false:

```
@Code
    Dim showToday = False
    If showToday Then
        DateTime.Today
    Else
        @<text>Sorry!</text>
    End If
End Code
```

If an `If` statement starts a code block, you don't have to use the normal `Code...End Code` statements to include the blocks. You can just add `@` to the block, and it will work. This approach works with `If` as well as other Visual Basic programming keywords that are followed by code blocks, including `For`, `For Each`, `Do While`, etc.

```
@If showToday Then
    DateTime.Today
Else
    @<text>Sorry!</text>
End If
```

You can add multiple conditions using one or more `ElseIf` blocks:

```
@Code
    Dim theBalance = 4.99
    If theBalance = 0 Then
        @<p>You have a zero balance.</p>
    ElseIf theBalance > 0 AndAlso theBalance <= 5 Then
        ' If the balance is above 0 but less than
        ' or equal to $5, display this message.
        @<p>Your balance of $@theBalance is very low.</p>
    Else
        ' For balances greater than $5, display balance.
        @<p>Your balance is: $@theBalance</p>
    End If
End Code
```

In this example, if the first condition in the `If` block is not true, the `ElseIf` condition is checked. If that condition is met, the statements in the `ElseIf` block are executed. If none of the conditions are met, the statements in the `Else` block are executed. You can add any number of `ElseIf` blocks, and then close with an `Else` block as the "everything else" condition.

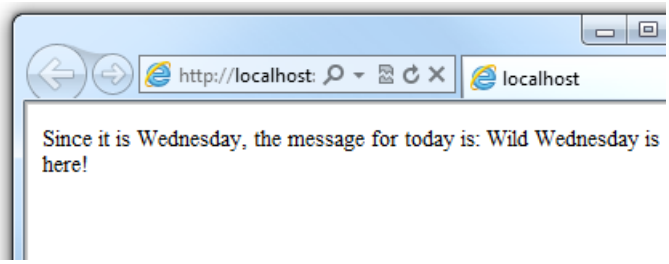
To test a large number of conditions, use a `Select Case` block:

```
@Code
    Dim weekday = "Wednesday"
    Dim greeting = ""

    Select Case weekday
        Case "Monday"
            greeting = "Ok, it's a marvelous Monday."
        Case "Tuesday"
            greeting = "It's a tremendous Tuesday."
        Case "Wednesday"
            greeting = "Wild Wednesday is here!"
        Case Else
            greeting = "It's some other day, oh well."
    End Select
End Code
<p>Since it is @weekday, the message for today is: @greeting</p>
```

The value to test is in parentheses (in the example, the weekday variable). Each individual test uses a `Case` statement that lists a value. If the value of a `Case` statement matches the test value, the code in that `Case` block is executed.

The result of the last two conditional blocks displayed in a browser:



Looping code

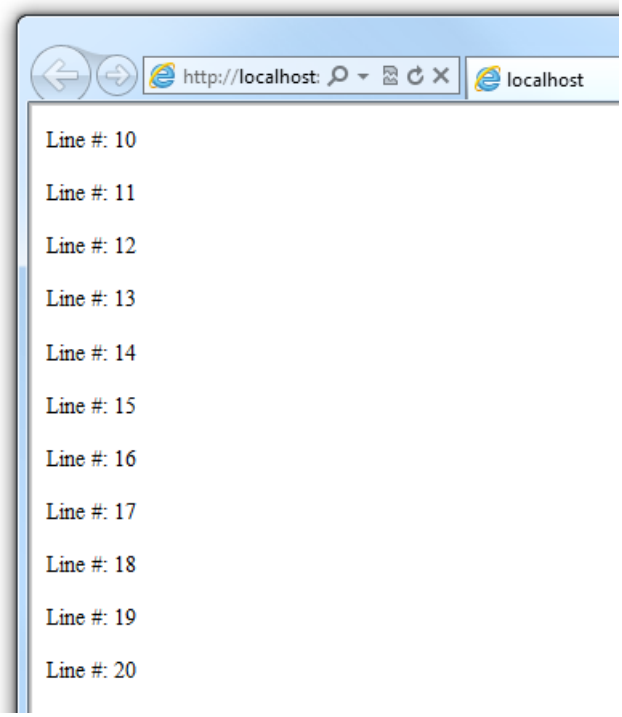
You often need to run the same statements repeatedly. You do this by looping. For example, you often run the same statements for each item in a collection of data. If you know exactly how many times you want to loop, you can use a `For` loop. This kind of loop is especially useful for counting up or counting down:

```
@For i = 10 To 20
    @<p>Item #: @i</p>
Next i
```

The loop begins with the `For` keyword, followed by three elements:

- Immediately after the `For` statement, you declare a counter variable (you don't have to use `Dim`) and then indicate the range, as in `i = 10 to 20`. This means the variable `i` will start counting at 10 and continue until it reaches 20 (inclusive).
- Between the `For` and `Next` statements is the content of the block. This can contain one or more code statements that execute with each loop.
- The `Next i` statement ends the loop. It increments the counter and starts the next iteration of the loop.

The line of code between the `For` and `Next` lines contains the code that runs for each iteration of the loop. The markup creates a new paragraph (`<p>` element) each time and adds a line to the output, displaying the value of i (the counter). When you run this page, the example creates 11 lines displaying the output, with the text in each line indicating the item number.

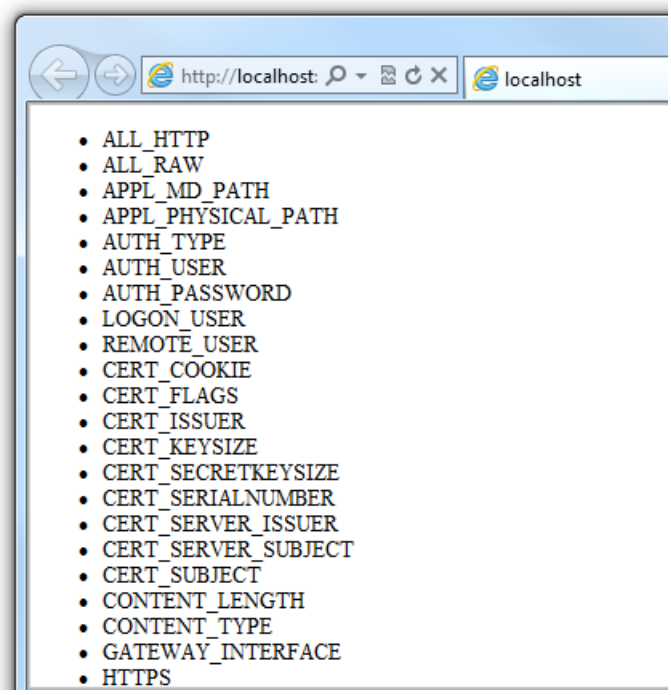


If you're working with a collection or array, you often use a `For Each` loop. A collection is a group of similar objects, and the `For Each` loop lets you carry out a task on each item in the collection. This type of loop is convenient for collections, because unlike a `For` loop, you don't have to increment the counter or set a limit. Instead, the `For Each` loop code simply proceeds through the collection until it's finished.

This example returns the items in the `Request.ServerVariables` collection (which contains information about your web server). It uses a `For Each` loop to display the name of each item by creating a new `<li>` element in an HTML bulleted list.

```
<ul>
@For Each myItem In Request.ServerVariables
    @<li>@myItem</li>
Next myItem
</ul>
```

The `For Each` keyword is followed by a variable that represents a single item in the collection (in the example, `myItem`), followed by the `In` keyword, followed by the collection you want to loop through. In the body of the `For Each` loop, you can access the current item using the variable that you declared earlier.



To create a more general-purpose loop, use the `Do While` statement:

```
@Code
    Dim countNum = 0
    Do While countNum < 50
        countNum += 1
        @<p>Line #@countNum: </p>
    Loop
End Code
```

This loop begins with the `Do While` keyword, followed by a condition, followed by the block to repeat. Loops typically increment (add to) or decrement (subtract from) a variable or object used for counting. In the example, the `+=` operator adds 1 to the value of a variable each time the loop runs. (To decrement a variable in a loop that counts down, you would use the decrement operator `-=`.)

Objects and Collections

Nearly everything in an ASP.NET website is an object, including the web page itself. This section discusses some important objects you'll work with frequently in your code.

Page objects

The most basic object in ASP.NET is the page. You can access properties of the page object directly without any qualifying object. The following code gets the page's file path, using the `Request` object of the page:

```
@Code
    Dim path = Request.FilePath
End Code
```

You can use properties of the `Page` object to get a lot of information, such as:

- `Request`. As you've already seen, this is a collection of information about the current request, including what type of browser made the request, the URL of the page, the user identity, etc.
- `Response`. This is a collection of information about the response (page) that will be sent to the browser when the server code has finished running. For example, you can use this property to write information into the response.

```
@Code
    ' Access the page's Request object to retrieve the URL.
    Dim pageUrl = Request.Url
End Code
    <a href="@pageUrl">My page</a>
```

Collection objects (arrays and dictionaries)

A collection is a group of objects of the same type, such as a collection of `Customer` objects from a database. ASP.NET contains many built-in collections, like the `Request.Files` collection.

You'll often work with data in collections. Two common collection types are the *array* and the *dictionary*. An array is useful when you want to store a collection of similar items but don't want to create a separate variable to hold each item:

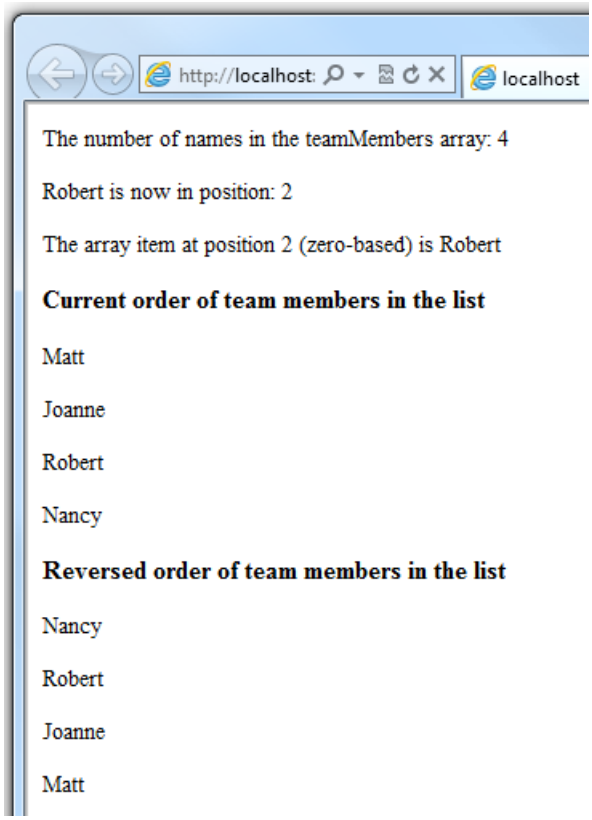
```
<h3>Team Members</h3>
@Code
    Dim teamMembers() As String = {"Matt", "Joanne", "Robert", "Nancy"}
    For Each name In teamMembers
        @<p>@name</p>
    Next name
End Code
```

With arrays, you declare a specific data type, such as `String`, `Integer`, or `DateTime`. To indicate that the variable can contain an array, you add parentheses to the variable name in the declaration (such as `Dim myVar() As String`). You can access items in an array using their position (index) or by using the `For Each` statement. Array indexes are zero-based — that is, the first item is at position 0, the second item is at position 1, and so on.

```
@Code
    Dim teamMembers() As String = {"Matt", "Joanne", "Robert", "Nancy"}
    @<p>The number of names in the teamMembers array: @teamMembers.Length </p>
    @<p>Robert is now in position: @Array.IndexOf(teamMembers, "Robert")</p>
    @<p>The array item at position 2 (zero-based) is @teamMembers(2)</p>
    @<h3>Current order of team members in the list</h3>
    For Each name In teamMembers
        @<p>@name</p>
    Next name
    @<h3>Reversed order of team members in the list</h3>
    Array.Reverse(teamMembers)
    For Each reversedItem In teamMembers
        @<p>@reversedItem</p>
    Next reversedItem
End Code
```

You can determine the number of items in an array by getting its `Length` property. To get the position of a specific item in the array (that is, to search the array), use the `Array.IndexOf` method. You can also do things like reverse the contents of an array (the `Array.Reverse` method) or sort the contents (the `Array.Sort` method).

The output of the string array code displayed in a browser:



A dictionary is a collection of key/value pairs, where you provide the key (or name) to set or retrieve the corresponding value:

```
@Code
    Dim myScores = New Dictionary(Of String, Integer)()
    myScores.Add("test1", 71)
    myScores.Add("test2", 82)
    myScores.Add("test3", 100)
    myScores.Add("test4", 59)
End Code
<p>My score on test 3 is: @myScores("test3")%</p>
@Code
    myScores("test4") = 79
End Code
<p>My corrected score on test 4 is: @myScores("test4")%</p>
```

To create a dictionary, you use the `New` keyword to indicate that you're creating a new `Dictionary` object. You can assign a dictionary to a variable using the `Dim` keyword. You indicate the data types of the items in the dictionary using parentheses (`( )`). At the end of the declaration, you must add another pair of parentheses, because this is actually a method that creates a new dictionary.

To add items to the dictionary, you can call the `Add` method of the dictionary variable (`myScores` in this case), and then specify a key and a value. Alternatively, you can use parentheses to indicate the key and do a simple assignment, as in the following example:

```
@Code
    myScores("test4") = 79
End Code
```

To get a value from the dictionary, you specify the key in parentheses:

```
@myScores("test4")
```

Calling Methods with Parameters

As you saw earlier in this article, the objects that you program with have methods. For example, a `Database` object might have a `Database.Connect` method. Many methods also have one or more parameters. A *parameter* is a value that you pass to a method to enable the method to complete its task. For example, look at a declaration for the `Request.MapPath` method, which takes three parameters:

```
Public Overridable Function MapPath (virtualPath As String, _
    baseVirtualDir As String, _
    allowCrossAppMapping As Boolean)
```

This method returns the physical path on the server that corresponds to a specified virtual path. The three parameters for the method are `virtualPath`, `baseVirtualDir`, and `allowCrossAppMapping`. (Notice that in the declaration, the parameters are listed with the data types of the data that they'll accept.) When you call this method, you must supply values for all three parameters.

When you're using Visual Basic with the Razor syntax, you have two options for passing parameters to a method: *positional parameters* or *named parameters*. To call a method using positional parameters, you pass the parameters in a strict order that's specified in the method declaration. (You would typically know this order by reading documentation for the method.) You must follow the order, and you can't skip any of the parameters — if necessary, you pass an empty string (`""`) or null for a positional parameter that you don't have a value for.

The following example assumes you have a folder named *scripts* on your website. The code calls the `Request.MapPath` method and passes values for the three parameters in the correct order. It then displays the resulting mapped path.

```
@Code
    ' Pass parameters to a method using positional parameters.
    Dim myPathPositional = Request.MapPath("/scripts", "/", true)
End Code
<p>@myPathPositional</p>
```

When there are many parameters for a method, you can keep your code cleaner and more readable by using named parameters. To call a method using named parameters, specify the parameter name followed by `:=` and then provide the value. An advantage of named parameters is that you can add them in any order you want. (A disadvantage is that the method call is not as compact.)

The following example calls the same method as above, but uses named parameters to supply the values:

```
@Code
    ' Pass parameters to a method using named parameters.
    Dim myPathNamed = Request.MapPath(baseVirtualDir:= "/", allowCrossAppMapping:= true, virtualPath:=
"/scripts")
End Code
<p>@myPathNamed</p>
```

As you can see, the parameters are passed in a different order. However, if you run the previous example and this example, they'll return the same value.

Handling Errors

Try-Catch statements

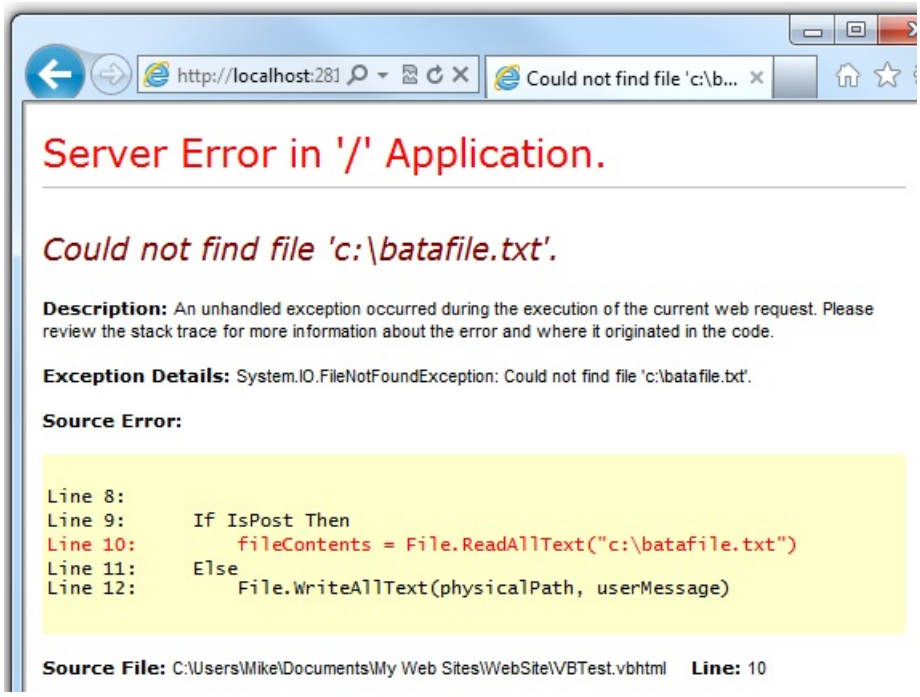
You'll often have statements in your code that might fail for reasons outside your control. For example:

- If your code tries to open, create, read, or write a file, all sorts of errors might occur. The file you want might not

exist, it might be locked, the code might not have permissions, and so on.

- Similarly, if your code tries to update records in a database, there can be permissions issues, the connection to the database might be dropped, the data to save might be invalid, and so on.

In programming terms, these situations are called *exceptions*. If your code encounters an exception, it generates (throws) an error message that is, at best, annoying to users.



In situations where your code might encounter exceptions, and in order to avoid error messages of this type, you can use `Try/Catch` statements. In the `Try` statement, you run the code that you're checking. In one or more `Catch` statements, you can look for specific errors (specific types of exceptions) that might have occurred. You can include as many `Catch` statements as you need to look for errors that you're anticipating.

> **NOTE**
>
> We recommend that you avoid using the `Response.Redirect` method in `Try/Catch` statements, because it can cause an exception in your page.

The following example shows a page that creates a text file on the first request and then displays a button that lets the user open the file. The example deliberately uses a bad file name so that it will cause an exception. The code includes `Catch` statements for two possible exceptions: `FileNotFoundException`, which occurs if the file name is bad, and `DirectoryNotFoundException`, which occurs if ASP.NET can't even find the folder. (You can uncomment a statement in the example in order to see how it runs when everything works properly.)

If your code didn't handle the exception, you would see an error page like the previous screen shot. However, the `Try/Catch` section helps prevent the user from seeing these types of errors.

```
@Code
    Dim dataFilePath = "~/dataFile.txt"
    Dim fileContents = ""
    Dim physicalPath = Server.MapPath(dataFilePath)
    Dim userMessage = "Hello world, the time is " + DateTime.Now
    Dim userErrMsg = ""
    Dim errMsg = ""

    If IsPost Then
        ' When the user clicks the "Open File" button and posts
        ' the page, try to open the file.
        Try
            ' This code fails because of faulty path to the file.
            fileContents = File.ReadAllText("c:\batafile.txt")

            ' This code works. To eliminate error on page,
            ' comment the above line of code and uncomment this one.
            ' fileContents = File.ReadAllText(physicalPath)

        Catch ex As FileNotFoundException
            ' You can use the exception object for debugging, logging, etc.
            errMsg = ex.Message
            ' Create a friendly error message for users.
            userErrMsg = "The file could not be opened, please contact " _
                & "your system administrator."

        Catch ex As DirectoryNotFoundException
            ' Similar to previous exception.
            errMsg = ex.Message
            userErrMsg = "The file could not be opened, please contact " _
                & "your system administrator."
        End Try
    Else
        ' The first time the page is requested, create the text file.
        File.WriteAllText(physicalPath, userMessage)
    End If
End Code
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Try-Catch Statements</title>
    </head>
    <body>
    <form method="POST" action="" >
      <input type="Submit" name="Submit" value="Open File"/>
    </form>

    <p>@fileContents</p>
    <p>@userErrMsg</p>

    </body>
</html>
```

Additional Resources

Reference Documentation

- ASP.NET
- Visual Basic Language

ASP.NET Web Pages (Razor) FAQ

4/10/2020 • 5 minutes to read • Edit Online

by Tom FitzMacken

> **NOTE**
> WebMatrix is no longer recommended as an integrated development environment for ASP.NET Web Pages. Use Visual Studio or Visual Studio Code.

This article lists some frequently asked questions about ASP.NET Web Pages (Razor) and WebMatrix.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 3
- Visual Studio 2013
- WebMatrix 3

This tutorial also works with ASP.NET Web Pages 2, WebMatrix 2, and Visual Studio 2012.

- What's the difference between ASP.NET Web Pages, ASP.NET Web Forms, and ASP.NET MVC?
- Do I need WebMatrix in order to work with Web Pages?
- Can I use ASP.NET Web Forms controls on a Web Pages page?
- Can I deploy an ASP.NET Web Pages site without using WebMatrix?
- Do I have to use the WebSecurity helper to support logins?
- Does ASP.NET Web Pages support HTML5?
- Can I use JavaScript and jQuery with Web Pages?
- Additional Resources

For questions about errors and other issues, see the ASP.NET Web Pages (Razor) Troubleshooting Guide.

What's the difference between ASP.NET Web Pages, ASP.NET Web Forms, and ASP.NET MVC?

All three are ASP.NET technologies for creating dynamic web applications:

- ASP.NET Web Pages focuses on adding dynamic (server-side) code and database access to HTML pages, and features simple and lightweight syntax.
- ASP.NET Web Forms is based on a page object model and traditional window-type controls (buttons, lists, etc.). Web Forms uses an event-based model that's familiar to those who've worked with client-based (Windows forms) development.
- ASP.NET MVC implements the model-view-controller pattern for ASP.NET. The emphasis is on "separation of concerns" (processing, data, and UI layers).

All three frameworks are fully supported and continue to be developed by the ASP.NET team. In general, the choice of which framework to use depends on your background and experience with ASP.NET.

ASP.NET Web Pages in particular was designed to make it easy for people who already know HTML to add server processing to their pages. It's a good choice for students, hobbyists, people in general who are new to

programming. It can also be a good choice for developers who have experience with non-ASP.NET web technologies.

Do I need WebMatrix in order to work with Web Pages?

No. WebMatrix is no longer recommended as an integrated development environment for ASP.NET Web Pages. Use Visual Studio or Visual Studio Code.

If you don't want to use either Visual Studio or Visual Studio Code, you can install the component products individually using Microsoft Web Platform Installer. You need the following products:

- Microsoft .NET Framework 4.5
- ASP.NET MVC 5 (which installs the ASP.NET Web Pages framework as well)
- IIS Express (the web server)
- Microsoft SQL Server Compact 4.0 (the database)

You can use a text editor to edit *.cshtml* (or *.vbhtml*) pages.

Managing SQL Server Compact databases (*.sdf* files) without a tool is a bit harder. Visual Studio containds tools for managing *.sdf* databases. You can also run SQL commands in code to perform many SQL Server management tasks.

To test *.cshtml* pages without using an integrated development environment (IDE), you can deploy them to a live server. (See Can I deploy an ASP.NET Web Pages site without using WebMatrix?)

Running IIS Express without using an IDE

If you install IIS Express on your computer as a web server, you can use that to test the pages. You can run IIS Express from the command line and associate it with a specific port number. You then specify that port when you request *.cshtml* files in your browser.

In Windows, open a command prompt with administrator privileges and change to *C:\Program Files\IIS Express.* (For 64-bit systems, use the folder *C:\Program Files (x86)\IIS Express.)* Then enter the following command, using the actual path to your site:

```
iisexpress.exe /port:35896 /path:C:\BasicWebSite
```

You can use any port number that isn't already reserved by some other process. (Port numbers above 1024 are typically free.) For the `path` value, use the path of the website folder where the *.cshtml* files are.

After you run this command to set up IIS Express to serve your pages, you can open a browser and browse to a *.cshtml* file. Use a URL like the following:

```
http://localhost:35896/default.cshtml
```

For help with IIS Express command line options, enter `iisexpress.exe /?` at the command line.

Can I use ASP.NET Web Forms controls on a Web Pages page?

No. Web Forms controls like the CheckBox control, the validation controls, and the GridView control only work in Web Forms pages (*.aspx* files). These controls require the Web Forms page framework.

Can I deploy an ASP.NET Web Pages site without using WebMatrix?

Yes. You can manually copy website files to a server (typically by using FTP). If you perform a manual copy, you also have to copy the files that support SQL Server Compact (the database). For details, see the blog entry Deploying Web Pages applications without a tool.

Do I have to use the WebSecurity helper to support logins?

No. The `SimpleMembership` provider that is part of ASP.NET Web Pages is one option. The security providers that are part of ASP.NET (that you might be used to working with in Web Forms) are also available. For example, you can use forms authentication in ASP.NET Web Pages just as you would in Web Forms. For one example of how to use forms authentication, see the Microsoft Support article How To Implement Forms-Based Authentication in Your ASP.NET Application by Using C#.NET. To download a simple example, see ASP.NET version of "Login & Password.

For information about how to use Windows authentication, see the blog post Using Windows authentication in ASP.NET Web Pages.

Does ASP.NET Web Pages support HTML5?

Yes. The pages you create with ASP.NET Web Pages (*.cshtml* or *.vbhtml* pages) are essentially HTML pages that also contain code that runs on the server, before the page is rendered. As long as the user's browser supports HTML5, you can use HTML5 elements in a *.cshtml* or *.vbhtml* page.

Can I use JavaScript and jQuery with Web Pages?

Absolutely. The pages you create with ASP.NET Web Pages (*.cshtml* or *.vbhtml* pages) are just HTML pages with server code in them. Therefore, anything you can do in a normal HTML page by using JavaScript or jQuery you can also do in a *.cshtml* or *.vbhtml* page.

The **Starter Site** template in WebMatrix contains a number of jQuery libraries. If you create a site by using that template, the *Scripts* folder contains a jQuery core library (*jquery-1.6.2.js)* and libraries for jQuery validation (*jquery.validate.js*, etc.).

Here are some blog posts that illustrate ways to use jQuery with ASP.NET Web Pages:

- Adding jQuery Goodness to ASP.NET Web Pages using WebMatrix by Rachel Appel
- 5 min: WebMatrix + jQuery UI + json + jQuery templates by Jonas Eriksson
- WebMatrix And jQuery Forms by Mike Brind

Additional Resources

ASP.NET Web Pages (Razor) Troubleshooting Guide

WebMatrix and ASP.NET Web Pages forum on the ASP.NET website

Sending Email from an ASP.NET Web Pages (Razor) Site

2/20/2020 • 9 minutes to read • Edit Online

by Tom FitzMacken

This article explains how to send an email message from a website when you use ASP.NET Web Pages (Razor).

What you'll learn:

- How to send an email message from your website.
- How to attach a file to an email message.

This is the ASP.NET feature introduced in the article:

- The `WebMail` helper.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 3

This tutorial also works with ASP.NET Web Pages 2.

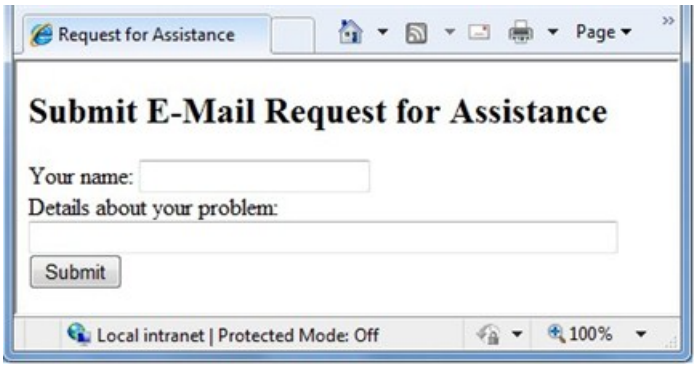
Sending Email Messages from Your Website

There are all sorts of reasons why you might need to send email from your website. You might send confirmation messages to users, or you might send notifications to yourself (for example, that a new user has registered.) The `WebMail` helper makes it easy for you to send email.

To use the `WebMail` helper, you have to have access to an SMTP server. (SMTP stands for *Simple Mail Transfer Protocol*.) An SMTP server is an email server that only forwards messages to the recipient's server — it's the outbound side of email. If you use a hosting provider for your website, they probably set you up with email and they can tell you what your SMTP server name is. If you're working inside a corporate network, an administrator or your IT department can usually give you the information about an SMTP server that you can use. If you're working at home, you might even be able to test using your ordinary email provider, who can tell you the name of their SMTP server. You typically need:

- The name of the SMTP server.
- The port number. This is almost always 25. However, your ISP may require you to use port 587. If you are using secure sockets layer (SSL) for email, you might need a different port. Check with your email provider.
- Credentials (user name, password).

In this procedure, you create two pages. The first page has a form that lets users enter a description, as if they were filling in a technical-support form. The first page submits its information to a second page. In the second page, code extracts the user's information and sends an email message. It also displays a message confirming that the problem report has been received.



NOTE

To keep this example simple, the code initializes the `WebMail` helper right in the page where you use it. However, for real websites, it's a better idea to put initialization code like this in a global file, so that you initialize the `WebMail` helper for all files in your website. For more information, see Customizing Site-Wide Behavior for ASP.NET Web Pages.

1. Create a new website.

2. Add a new page named *EmailRequest.cshtml* and add the following markup:

```
<!DOCTYPE html>
<html>
<head>
    <title>Request for Assistance</title>
</head>
<body>
  <h2>Submit Email Request for Assistance</h2>
  <form method="post" action="ProcessRequest.cshtml">
    <div>
        Your name:
        <input type="text" name="customerName" />
    </div>

    <div>
        Your email address:
        <input type="text" name="customerEmail" />
    </div>

    <div>
        Details about your problem: <br />
        <textarea name="customerRequest" cols="45" rows="4"></textarea>
    </div>

    <div>
        <input type="submit" value="Submit" />
    </div>
  </form>
</body>
</html>
```

   Notice that the `action` attribute of the form element has been set to *ProcessRequest.cshtml*. This means that the form will be submitted to that page instead of back to the current page.

3. Add a new page named *ProcessRequest.cshtml* to the website and add the following code and markup:

```
@{
    var customerName = Request["customerName"];
    var customerEmail = Request["customerEmail"];
    var customerRequest = Request["customerRequest"];
    var errorMessage = "";
    var debuggingFlag = false;
    try {
        // Initialize WebMail helper
        WebMail.SmtpServer = "your-SMTP-host";
        WebMail.SmtpPort = 25;
        WebMail.UserName = "your-user-name-here";
        WebMail.Password = "your-account-password";
        WebMail.From = "your-email-address-here";

        // Send email
        WebMail.Send(to: customerEmail,
            subject: "Help request from - " + customerName,
            body: customerRequest
        );
    }
    catch (Exception ex ) {
        errorMessage = ex.Message;
    }
}
<!DOCTYPE html>
<html>
<head>
  <title>Request for Assistance</title>
</head>
<body>
  <p>Sorry to hear that you are having trouble, <b>@customerName</b>.</p>
    @if(errorMessage == ""){
      <p>An email message has been sent to our customer service
         department regarding the following problem:</p>
      <p><b>@customerRequest</b></p>
    }
    else{
        <p><b>The email was <em>not</em> sent.</b></p>
        <p>Please check that the code in the ProcessRequest page has
           correct settings for the SMTP server name, a user name,
           a password, and a "from" address.
        </p>
        if(debuggingFlag){
            <p>The following error was reported:</p>
            <p><em>@errorMessage</em></p>
        }
    }
</body>
</html>
```

In the code, you get the values of the form fields that were submitted to the page. You then call the `WebMail` helper's `Send` method to create and send the email message. In this case, the values to use are made up of text that you concatenate with the values that were submitted from the form.

The code for this page is inside a `try/catch` block. If for any reason the attempt to send an email doesn't work (for example, the settings aren't right), the code in the `catch` block runs and sets the `errorMessage` variable to the error that has occurred. (For more information about `try/catch` blocks or the `<text>` tag, see Introduction to ASP.NET Web Pages Programming Using the Razor Syntax.)

In the body of the page, if the `errorMessage` variable is empty (the default), the user sees a message that the email message has been sent. If the `errorMessage` variable is set to true, the user sees a message that there's been a problem sending the message.

Notice that in the portion of the page that displays an error message, there's an additional test:

`if(debuggingFlag)`. This is a variable that you can set to true if you're having trouble sending email. When `debuggingFlag` is true, and if there's a problem sending email, an additional error message is displayed that shows whatever ASP.NET has reported when it tried to send the email message. Fair warning, though: the error messages that ASP.NET reports when it can't send an email message can be generic. For example, if ASP.NET can't contact the SMTP server (for example, because you made an error in the server name), the error is `Failure sending mail`.

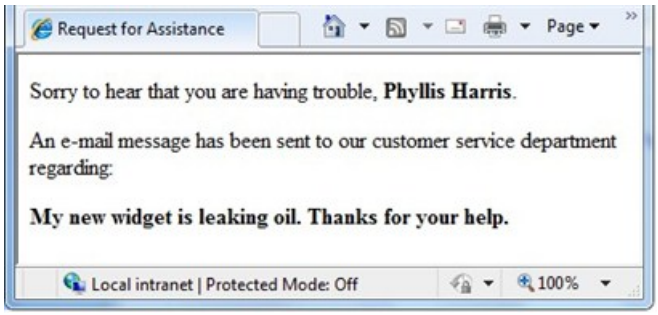
> **NOTE**
>
> **Important** When you get an error message from an exception object (`ex` in the code), do *not* routinely pass that message through to users. Exception objects often include information that users should not see and that can even be a security vulnerability. That's why this code includes the variable `debuggingFlag` that's used as a switch to display the error message, and why the variable by default is set to false. You should set that variable to true (and therefore display the error message) *only* if you're having a problem with sending email and you need to debug. Once you have fixed any problems, set `debuggingFlag` back to false.

Modify the following email related settings in the code:

- Set `your-SMTP-host` to the name of the SMTP server that you have access to.
- Set `your-user-name-here` to the user name for your SMTP server account.
- Set `your-account-password` to the password for your SMTP server account.
- Set `your-email-address-here` to your own email address. This is the email address that the message is sent from. (Some email providers don't let you specify a different `From` address and will use your user name as the `From` address.)

> **TIP**
>
> **Configuring Email Settings**
>
> It can be a challenge sometimes to make sure you have the right settings for the SMTP server, port number, and so on. Here are a few tips:
>
> - The SMTP server name is often something like `smtp.provider.com` or `smtp.provider.net`. However, if you publish your site to a hosting provider, the SMTP server name at that point might be `localhost`. This is because after you've published and your site is running on the provider's server, the email server might be local from the perspective of your application. This change in server names might mean you have to change the SMTP server name as part of your publishing process.
> - The port number is usually 25. However, some providers require you to use port 587 or some other port.
> - Make sure that you use the right credentials. If you've published your site to a hosting provider, use the credentials that the provider has specifically indicated are for email. These might be different from the credentials you use to publish.
> - Sometimes you don't need credentials at all. If you're sending email using your personal ISP, your email provider might already know your credentials. After you publish, you might need to use different credentials than when you test on your local computer.
> - If your email provider uses encryption, you have to set `WebMail.EnableSsl` to `true`.

4. Run the *EmailRequest.cshtml* page in a browser. (Make sure the page is selected in the **Files** workspace before you run it.)
5. Enter your name and a problem description, and then click the **Submit** button. You're redirected to the *ProcessRequest.cshtml* page, which confirms your message and which sends you an email message.



Sending a File Using Email

You can also send files that are attached to email messages. In this procedure, you create a text file and two HTML pages. You'll use the text file as an email attachment.

1. In the website, add a new text file and name it *MyFile.txt*.

2. Copy the following text and paste it in the file:

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et
dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip
ex ea commodo consequat.
```

3. Create a page named *SendFile.cshtml* and add the following markup:

```
<!DOCTYPE html>
<html>
<head>
    <title>Attach File</title>
</head>
<body>
  <h2>Submit Email with Attachment</h2>
  <form method="post" action="ProcessFile.cshtml">
    <div>
        Your name:
        <input type="text" name="customerName" />
    </div>

    <div>
        Your email address:
        <input type="text" name="customerEmail" />
    </div>

    <div>
        Subject line: <br />
        <input type="text" size= 30 name="subjectLine" />
    </div>

    <div>
        File to attach: <br />
        <input type="text" size=60 name="fileAttachment" />
    </div>

    <div>
        <input type="submit" value="Submit" />
    </div>
  </form>
</body>
</html>
```

4. Create a page named *ProcessFile.cshtml* and add the following markup:

```
@{
    var customerName = Request["customerName"];
    var customerEmail = Request["customerEmail"];
    var customerRequest = Request["customerRequest"];
    var subjectLine = Request["subjectLine"];
    var fileAttachment = Request["fileAttachment"];
    var errorMessage = "";
    var debuggingFlag = false;

    try {
        // Initialize WebMail helper
        WebMail.SmtpServer = "your-SMTP-host";
        WebMail.SmtpPort = 25;
        WebMail.UserName = "your-user-name-here";
        WebMail.Password = "your-account-password";
        WebMail.From = "your-email-address-here";

        // Create array containing file name
        var filesList = new string [] { fileAttachment };

        // Attach file and send email
        WebMail.Send(to: customerEmail,
            subject: subjectLine,
            body: "File attached. <br />From: " + customerName,
            filesToAttach: filesList);
    }
    catch (Exception ex ) {
        errorMessage = ex.Message;
    }
}
<!DOCTYPE html>
<html>
<head>
    <title>Request for Assistance</title>
</head>
<body>
  <p>Sorry to hear that you are having trouble, <b>@customerName</b>.</p>
    @if(errorMessage == ""){
        <p><b>@customerName</b>, thank you for your interest.</p>
        <p>An email message has been sent to our customer service
           department with the <b>@fileAttachment</b> file attached.</p>
    }
    else{
        <p><b>The email was <em>not</em> sent.</b></p>
        <p>Please check that the code in the ProcessRequest page has
           correct settings for the SMTP server name, a user name,
           a password, and a "from" address.
        </p>
        if(debuggingFlag){
            <p>The following error was reported:</p>
            <p><em>@errorMessage</em></p>
        }
    }
</body>
</html>
```

5. Modify the following email related settings in the code from the example:

- Set `your-SMTP-host` to the name of an SMTP server that you have access to.
- Set `your-user-name-here` to the user name for your SMTP server account.
- Set `your-email-address-here` to your own email address. This is the email address that the message is sent from.
- Set `your-account-password` to the password for your SMTP server account.
- Set `target-email-address-here` to your own email address. (As before, you'd normally send an email to

someone else, but for testing, you can send it to yourself.)

6. Run the *SendFile.cshtml* page in a browser.

7. Enter your name, a subject line, and the name of the text file to attach (*MyFile.txt*).

8. Click the `Submit` button. As before, you're redirected to the *ProcessFile.cshtml* page, which confirms your message and which sends you an email message with the attached file.

Additional Resources

- ASP.NET Web Pages (Razor) Troubleshooting Guide
- Simple Mail Transfer Protocol
- Customizing Site-Wide Behavior for ASP.NET Web Pages

Adding Social Networking to ASP.NET Web Pages (Razor) Sites

2/20/2020 • 5 minutes to read • Edit Online

by Tom FitzMacken

This article explains how to add social networking links for Facebook, Twitter, Reddit, and Digg to pages in an ASP.NET Web Pages (Razor) website, and how to include Twitter feeds, Xbox gamer cards, and Gravatar images.

What you'll learn:

- How to let people bookmark/link your site.
- How to add a Twitter feed.
- How to add a Facebook **Like** button to pages.
- How to render Gravatar.com images.
- How to display an Xbox gamer card on your site.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 2
- ASP.NET Web Helper Library (NuGet package)

This tutorial also works with ASP.NET Web Pages 3, except for parts that use the ASP.NET Web Helper Library.

Linking Your Website on Social Networking Sites

If people like something on your site, they often want to share it with friends. You can make this easy by displaying glyphs (icons) that people can click to share a page on Digg, Reddit, Facebook, Twitter, or similar sites.

To display these glyphs, add the `LinkSharecode` helper to a page. People who visit your page can click an individual glyph. If they have an account with that social networking site, they can then post a link to your page on that site.

1. Add the ASP.NET Web Helpers Library to your website as described in Installing Helpers in an ASP.NET Web Pages Site, if you haven't already added it.- Create a page named *ListLinkShare.cshtml* and add the following markup:

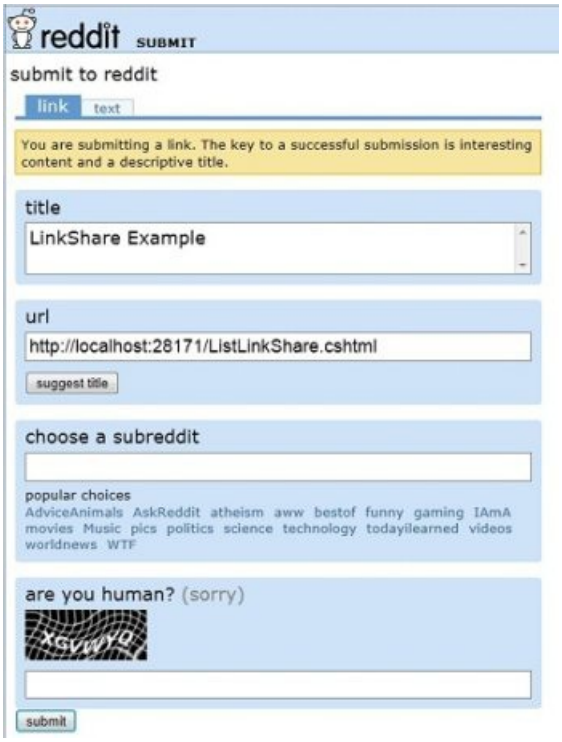
```
@using Microsoft.Web.Helpers;

<!DOCTYPE html>
<html>
  <head>
    <title>LinkShare Example</title>
  </head>
  <body>
    <h1>LinkShare Example</h1>
    Share: @LinkShare.GetHtml("LinkShare Example", linkSites: new[]{
        LinkShareSite.Reddit, LinkShareSite.Facebook, LinkShareSite.Twitter})
  </body>
</html>
```

In this example, when the `LinkShare` helper runs, the page title is passed as a parameter, which in turn passes the page title to the social networking site. However, you could pass in any string you want. This example also specifies which social networking sites to include in the list. You can specify the social networking sites that are relevant to your site.

2. Run the *ListLinkShare.cshtml* page in a browser. (Make sure the page is selected in the **Files** workspace before you run it.)

3. Click a glyph for one of the sites that you're signed up for. The link takes you to the page on the selected social network site where you can share a link. For example, if you click the Reddit link, you're taken to the `submit to reddit` page on the Reddit website.



Adding a Twitter Feed

For information about using a Twitter helper that is compatible with the current version of the Twitter API, see Twitter helper. This example shows how to write your own helper so you can easily reuse the code from many pages.

Displaying a Facebook "Like" Button

In some cases, your best option is to get the code directly from the social networking provider rather than relying on a helper. This is especially true if the social network provider updates its options more quickly than the helper is

updated.

To add Facebook features (such as the Like button) to your site, you can retrieve code snippets from the developers.facebook.com site. On the Facebook site, you use their tools to generate a code snippet that is relevant to your site.

The following highlighted code is the code that was retrieved from the Like Button tool on the developers.facebook.com site. You must provide your own app ID.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Facebook Like Example</title>
  </head>
  <body>
    <div id="fb-root"></div>
    <script>(function(d, s, id) {
        var js, fjs = d.getElementsByTagName(s)[0];
        if (d.getElementById(id)) return;
        js = d.createElement(s); js.id = id;
        js.src = "//connect.facebook.net/en_US/all.js#xfbml=1&appId=<provide appId>";
        fjs.parentNode.insertBefore(js, fjs);
      }(document, 'script', 'facebook-jssdk'));</script>
    <h1>Facebook Like Example</h1>
    <div class="fb-like" data-href="/web-pages/overview" data-layout="standard"
        data-action="like" data-show-faces="true" data-share="true"></div>
  </body>
</html>
```

Rendering a Gravatar Image

A *Gravatar* (a "globally recognized avatar") is an image that can be used on multiple websites as your avatar — that is, an image that represents you. For example, a Gravatar can identify a person in a forum post, in a blog comment, and so on. (You can register your own Gravatar at the Gravatar website at http://www.gravatar.com/.) If you want to display images next to people's names or email addresses on your website, you can use the Gravatar helper.

In this example, you're using a single Gravatar that represents yourself. Another way to use a Gravatar is to let people specify their Gravatar address when they register on your site. (You can learn how to let people register in Adding Security and Membership to an ASP.NET Web Pages Site.) Then whenever you display information for that user, you can just add the Gravatar to where you display the user's name.

1. Add the ASP.NET Web Helpers Library to your website as described in Installing Helpers in an ASP.NET Web Pages Site, if you haven't already.

2. Create a new web page named *Gravatar.cshtml*.

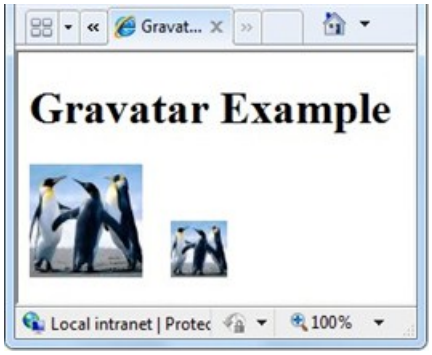
3. Add the following markup to the file:

```
@using Microsoft.Web.Helpers;
<!DOCTYPE html>
<html>
    <head>
        <title>Gravatar Example</title>
    </head>
    <body>
        <h1>Gravatar Example</h1>
        @Gravatar.GetHtml("<Your Gravatar account here>")
           
        @Gravatar.GetHtml("<Your Gravatar account here>", 40)
    </body>
</html>
```

The `Gravatar.GetHtml` method displays the Gravatar image on the page. To change the size of the image, you can include a number as a second parameter. The default size is 80. Numbers less than 80 make the image smaller. Numbers greater than 80 make the image larger.

4. In the `Gravatar.GetHtml` methods, replace `<Your Gravatar account here>` with the email address that you use for your Gravatar account. (If you don't have a Gravatar account, you can use the email address of someone who does.)

5. Run the page in your browser. The page displays two Gravatar images for the email address you specified. The second image is smaller than the first.



Displaying an Xbox Gamer Card

When people play Microsoft Xbox games online, each user has a unique ID. Statistics are kept for each player in the form of a gamer card, which shows their reputation, gamer score, and recently played games. If you're an Xbox gamer, you can show your gamer card on pages in your site by using the `GamerCard` helper.

1. Add the ASP.NET Web Helpers Library to your website as described in Installing Helpers in an ASP.NET Web Pages Site, if you haven't already.

2. Create a new page named *XboxGamer.cshtml* and add the following markup.

```
@using Microsoft.Web.Helpers;
<!DOCTYPE html>
<html>
  <head>
    <title>Xbox Gamer Card</title>
  </head>
  <body>
    <h1>Xbox Gamer Card</h1>
    @GamerCard.GetHtml("major nelson")
  </body>
</html>
```

You use the `GamerCard.GetHtml` property to specify the alias for the gamer card to be displayed.

3. Run the page in your browser. The page displays the Xbox gamer card that you specified.

Introducing ASP.NET Web Pages

2/20/2020 • 2 minutes to read • Edit Online

Introducing ASP.NET Web Pages 2

- Getting Started
- Programming Basics
- Displaying Data
- HTML Form Basics
- Entering Database Data by Using Forms
- Updating Database Data
- Deleting Database Data
- Creating a Consistent Layout
- Publishing a Site by Using WebMatrix

Getting Started

9/16/2020 • 13 minutes to read • Edit Online

by Tom FitzMacken

For new web application development, we recommend Razor Pages. For more information, see Get started with Razor Pages.

> **NOTE**
> WebMatrix is no longer recommended as an integrated development environment for ASP.NET Web Pages. Use Visual Studio or Visual Studio Code.

This guidance and application gives you an overview of ASP.NET Web Pages (version 2 or later) and Razor syntax, a lightweight framework for creating dynamic websites. It also introduces WebMatrix, a tool for creating pages and sites.

Level: New to ASP.NET Web Pages.
Skills assumed: HTML, basic cascading style sheets (CSS).

What you'll learn in the first tutorial of the set:

- What ASP.NET Web Pages technology is and what it's for.
- What WebMatrix is.
- How to install everything.
- How to create a website by using WebMatrix.

Features/technologies discussed:

- Microsoft Web Platform Installer.
- WebMatrix.
- *.cshtml* pages

Mike Pope originally wrote this tutorial. Tom FitzMacken updated it for Microsoft WebMatrix 3.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 2
- WebMatrix 3

What Should You Know?

We're assuming that you're familiar with:

- **HTML**. No in-depth expertise is required. We won't explain HTML, but we also don't use anything complex. We'll provide links to HTML tutorials where we think they're useful.
- **Cascading style sheets (CSS)**. Same as with HTML.
- **Basic database ideas**. If you've used a spreadsheet for data and sorted and filtered the data, that's the level of expertise we're generally assuming for this tutorial set.

We're also assuming that you're interested in learning basic programming. ASP.NET Web Pages use a

programming language called C#. You don't have to have any background in programming, just an interest in it. If you've ever written any JavaScript in a web page before, you've got plenty of background.

Note that if you are familiar with programming, you might find that this tutorial set initially moves slowly while we bring new programmers up to speed. As we get past the first few tutorials, though, there will be less basic programming to explain and things will move along at a faster clip.

What Do You Need?

Here's what you'll need:

- A computer that is running Windows 8, Windows 7, Windows Server 2008, or Windows Server 2012.
- A live internet connection.
- Administrator privileges (required for the installation process).

What Is ASP.NET Web Pages?

ASP.NET Web Pages is a framework that you can use to create dynamic web pages. A simple HTML web page is static; its content is determined by the fixed HTML markup that's in the page. Dynamic pages like those you create with ASP.NET Web Pages let you create the page content on the fly, by using code.

Dynamic pages let you do all sorts of things. You can ask a user for input by using a form and then change what the page displays or how it looks. You can take information from a user, save it in a database, and then list it later. You can send email from your site. You can interact with other services on the web (for example, a mapping service) and produce pages that integrate information from those sources.

What is WebMatrix?

WebMatrix is a tool that integrates a web page editor, a database utility, a web server for testing pages, and features for publishing your website to the Internet. WebMatrix is free, and it's easy to install and easy to use. (It also works for just plain HTML pages, as well as for other technologies like PHP.)

You don't actually *have* to use WebMatrix to work with ASP.NET Web Pages. You can create pages by using a text editor, for example, and test pages by using a web server that you have access to. However, WebMatrix makes it all very easy, so these tutorials will use WebMatrix throughout.

About These Tutorials

This tutorial set is an introduction to how to use ASP.NET Web Pages. There are 9 tutorials total in this introductory tutorial set. It's part of a series of tutorial sets that takes you from ASP.NET Web Pages novice to creating real, professional-looking websites.

This first tutorial set concentrates on showing you the basics of how to work with ASP.NET Web Pages. When you're done, you can work with additional tutorial sets that pick up where this one ends and that explore Web Pages in more depth.

We deliberately go easy on the in-depth explanations. And whenever we show something, for this tutorial set we always choose the way that we think is easiest to understand. Later tutorial sets go into more depth and show you more efficient or more flexible approaches (also more fun). But those tutorials require you to understand the basics first.

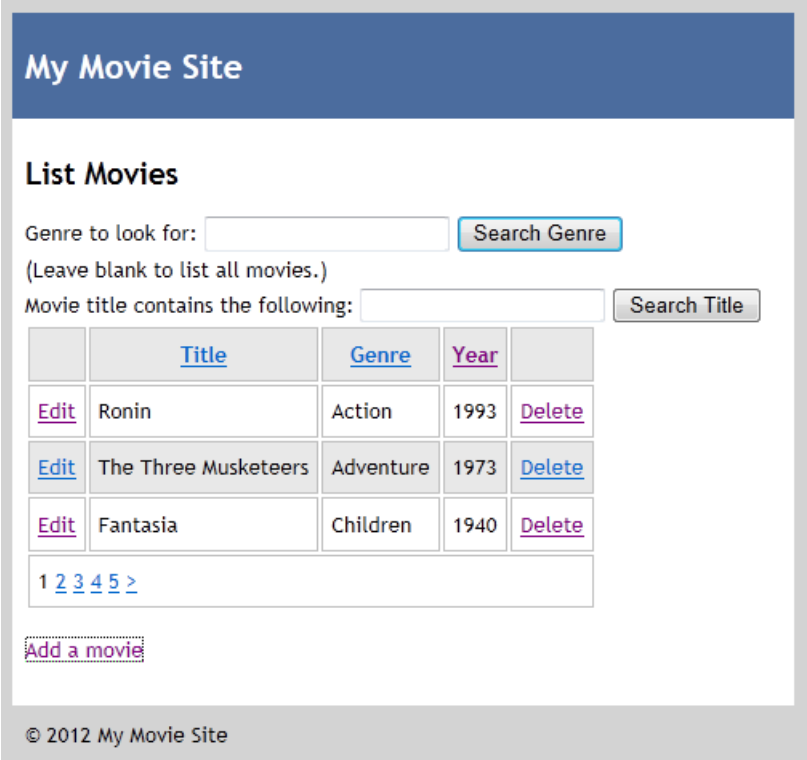
The tutorial set you've just started covers these features of ASP.NET Web Pages:

- Introduction and getting everything installed. (That's in the tutorial you're reading.)
- The basics of ASP.NET Web Pages programming.
- Creating a database.

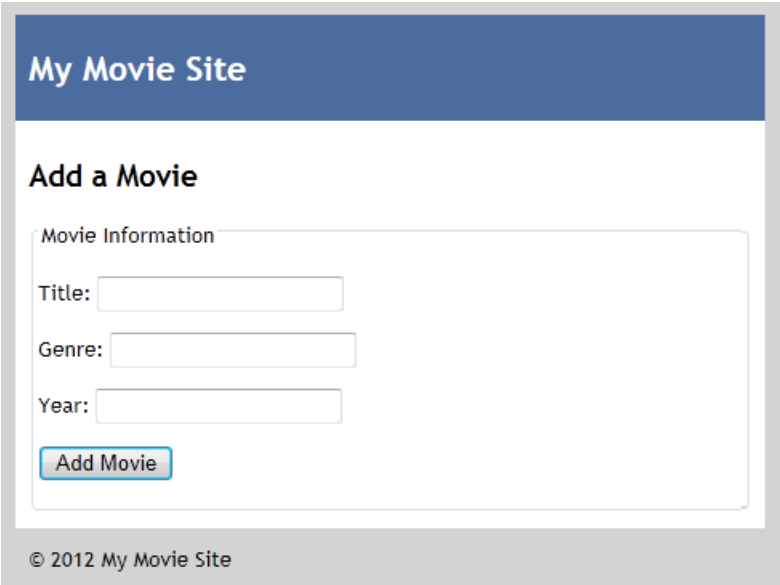
- Creating and processing a user input form.
- Adding, updating, and deleting data in the database.

What Will You Create?

This tutorial set and subsequent ones revolve around a website where you can list movies that you like. You'll be able to enter movies, edit them, and list them. Here are a couple of the pages you'll create in this tutorial set. The first one shows the movie listing page that you'll create:



And here's the page that lets you add new movie information to your site:



Subsequent tutorial sets build on this set and add more functionality, like uploading pictures, letting people log in, sending email, and integrating with social media.

See this App Running on Azure

Would you like to see the finished site running as a live web app? You can deploy a complete version of the app to your Azure account by simply clicking the following button.

Deploy to Azure

You need an Azure account to deploy this solution to Azure. If you do not already have an account, you have the following options:

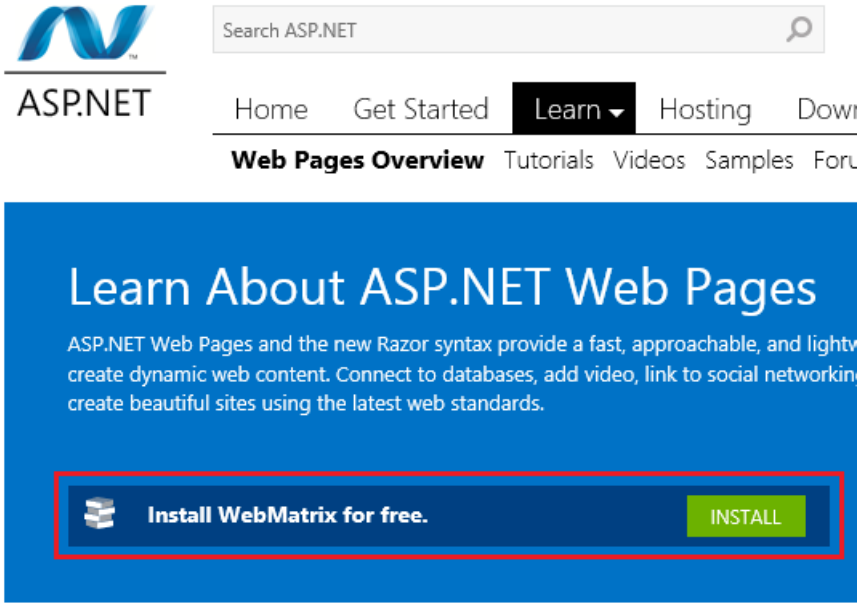
- Open an Azure account for free - You get credits you can use to try out paid Azure services, and even after they're used up you can keep the account and use free Azure services.
- Activate MSDN subscriber benefits - Your MSDN subscription gives you credits every month that you can use for paid Azure services.

Installing Everything

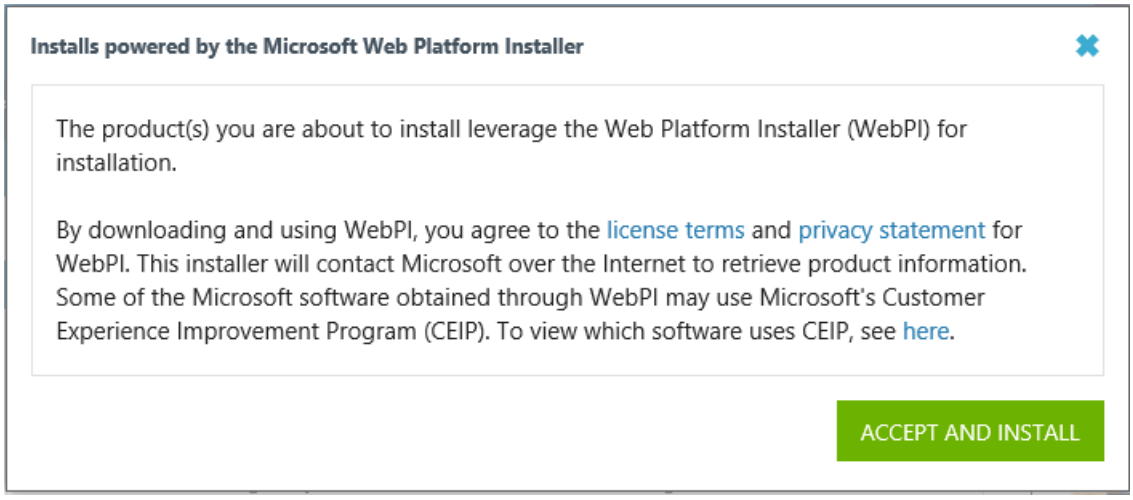
You can install everything by using the Web Platform Installer from Microsoft. In effect, you install the installer, and then use it to install everything else.

To use Web Pages, you have to be have at least Windows XP with SP3 installed, or Windows Server 2008 or later.

On the Web Pages page of the ASP.NET website, click **Install**.

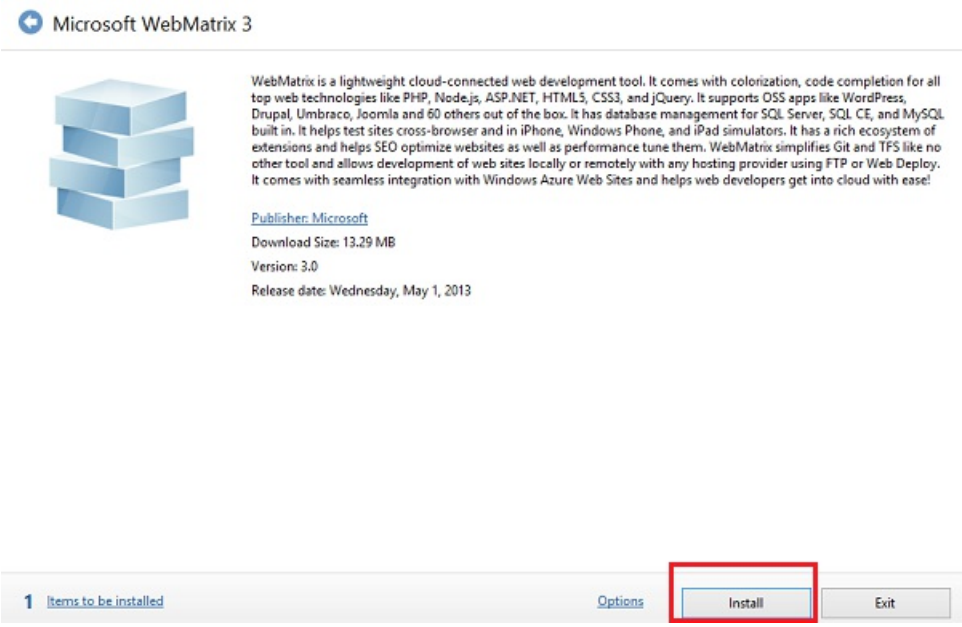


You are asked to accept the license terms and privacy statement before installing WebMatrix.



Click **Run** to start the installation. (If you want to save the installer, click **Save** and then run the installer from the folder where you saved it.)

The Web Platform Installer appears, ready to install WebMatrix. Click **Install**.



The installation process figures out what it has to install on your computer and starts the process. Depending on what exactly has to be installed, the process can take anywhere from a few moments to several minutes. Select **I Accept** to accept the license terms.

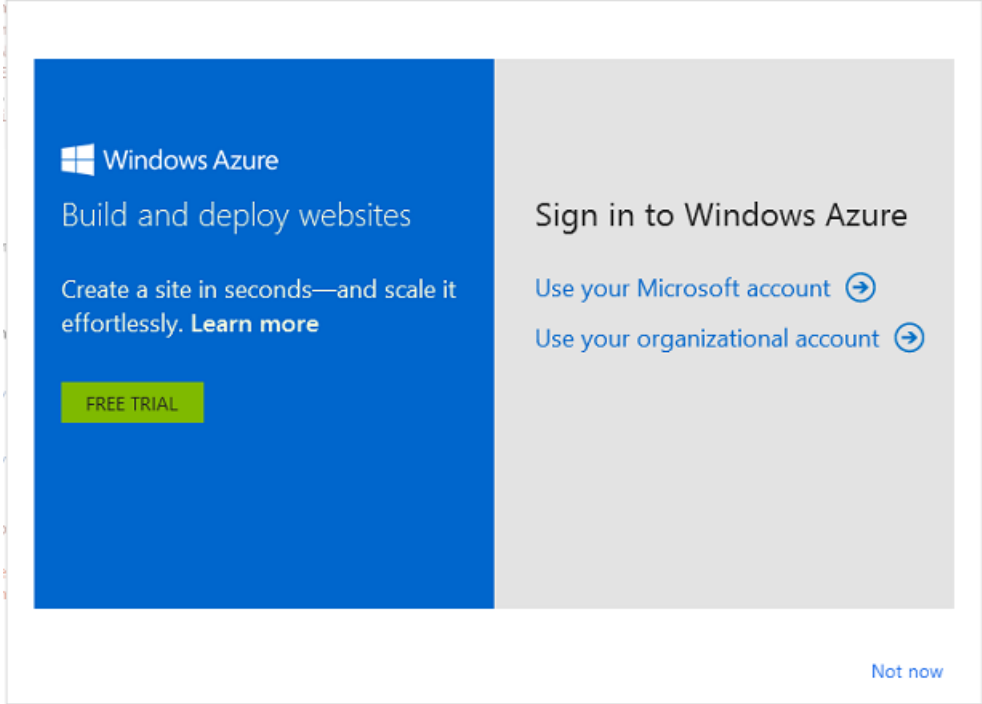
Hello, WebMatrix

When it's done, the installation process can launch WebMatrix automatically. If it doesn't, in Windows, from the **Start** menu, launch **Microsoft WebMatrix**.

When you launch WebMatrix for the first time, you are given a chance to sign in to Microsoft Azure with your Microsoft account. By signing in, you will receive 10 free web apps through Azure. These free web apps provide a convenient way to test your apps. If you don't already have an Azure account, but you do have an MSDN subscription, you can activate your MSDN subscription benefits. Otherwise, you can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

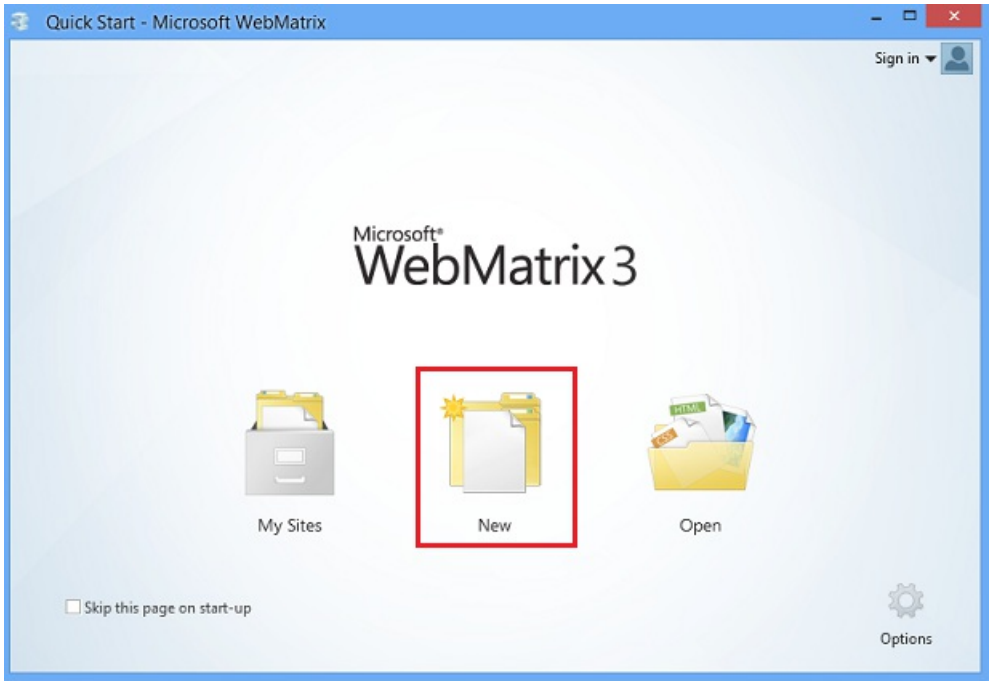
You do not have to sign in right now to continue with this tutorial. If you do not sign in now, you will still have the option to sign in later. The last topic in this tutorial series covers how to deploy your website to Azure; therefore, you would need to sign in to complete that topic.

At this point, either sign in with your Microsoft account or select **Not Now** in the lower right corner.



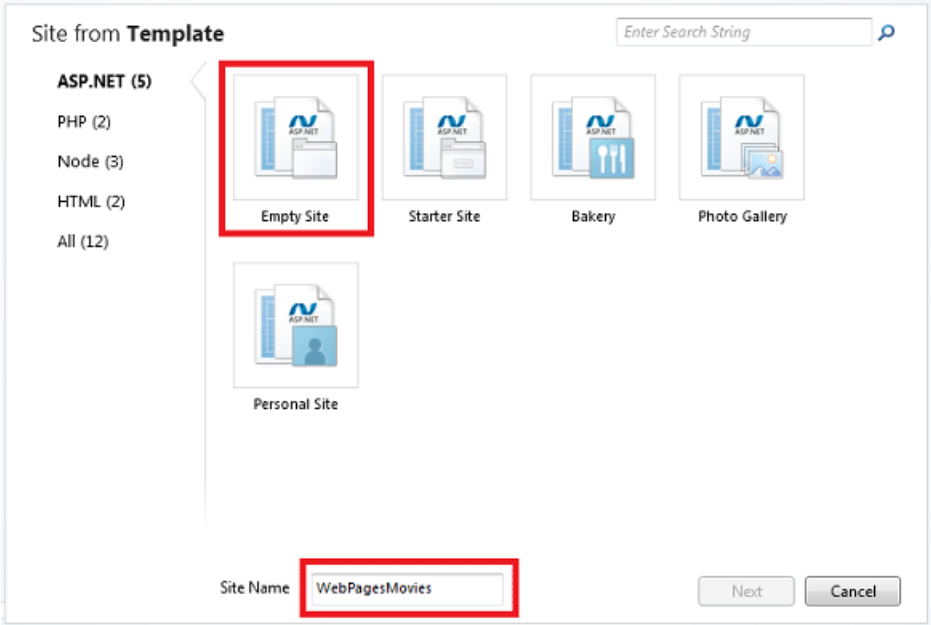
To begin, you'll create a blank website and add a page. In a later tutorial in this set you'll play with one of the built-in website templates.

In the start window, click **New**.



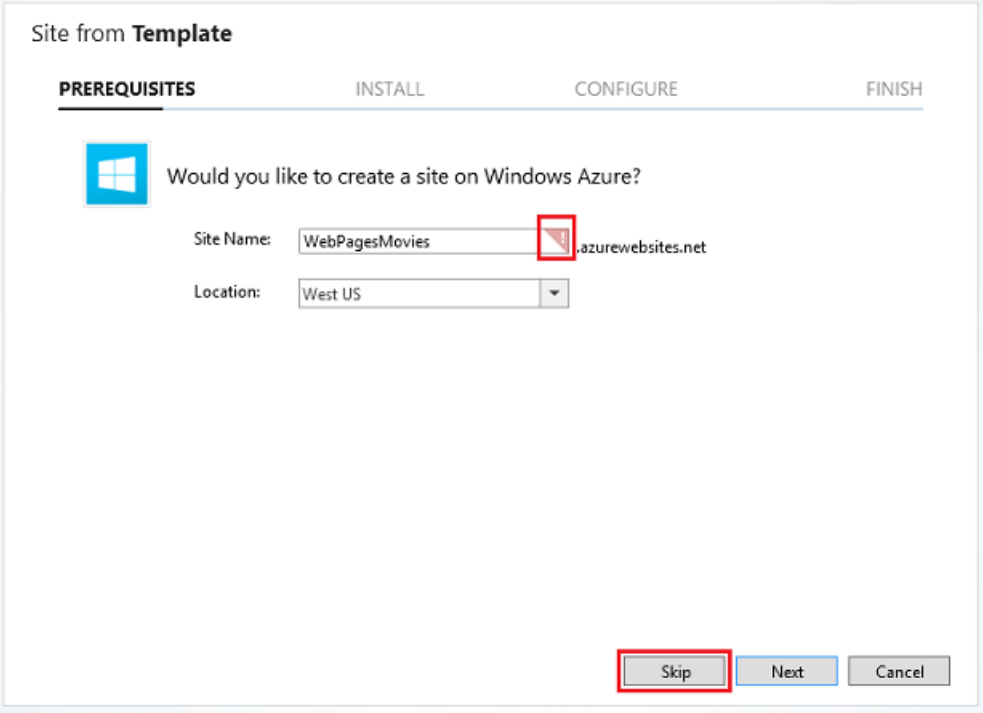
Templates are prebuilt files and pages for different types of websites. To see all of the templates that are available by default, select the Template Gallery option.

In the **Quick Start** window, select **Empty Site** from the **ASP.NET** group and name the new site "WebPagesMovies".

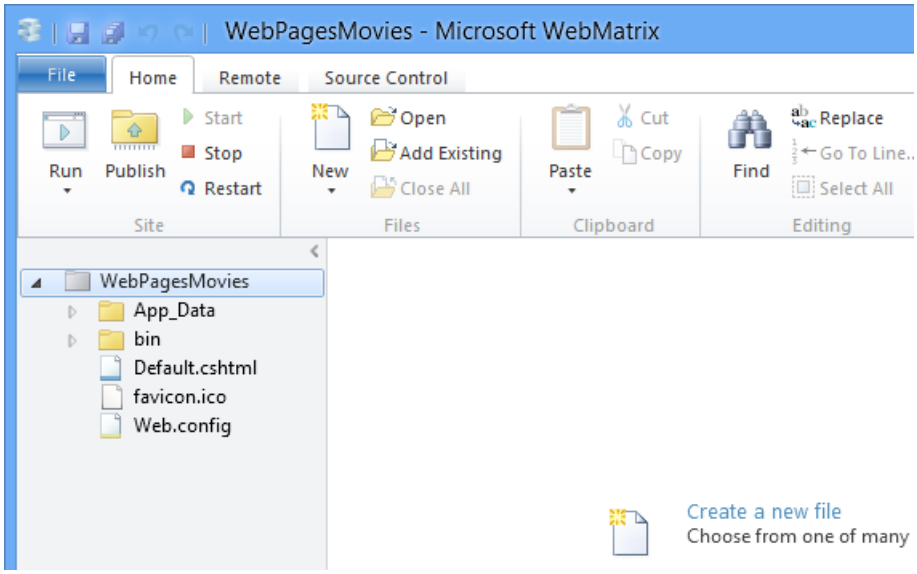


Click **Next**.

If you have signed in to your Microsoft account, you will be given the opportunity to create the site on Azure. Based on the name of your site, the default name of **WebPagesMovies.azurewebsites.net** is suggested; however, the exclamation point indicates that this name is not available on Windows Azure. For simplicity, select **Skip** to bypass creating the web site on Azure right now. Later in this series, we will publish the site to Azure.



WebMatrix creates and opens the site:



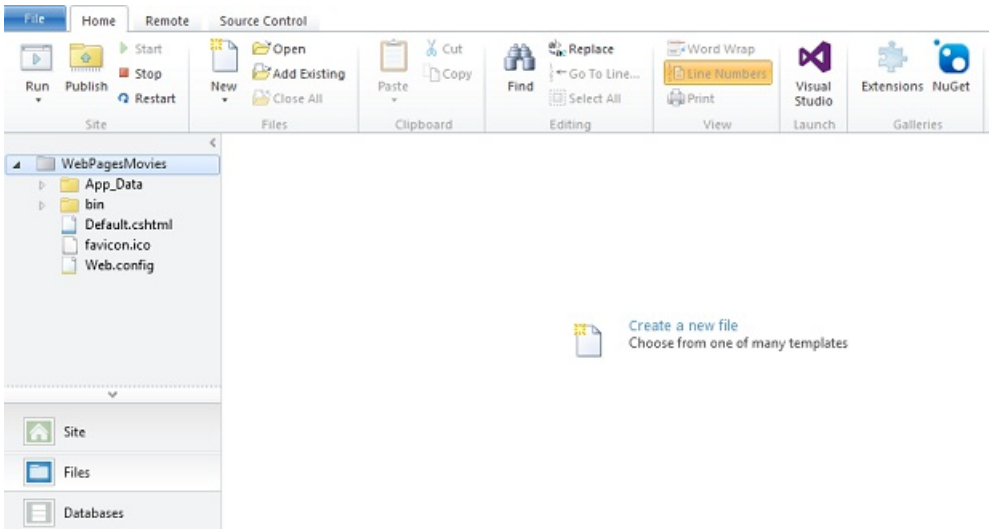
At the top, there's a Quick Access Toolbar and a ribbon. At the bottom left, you see the workspace selector where you switch between tasks (**Site**, **Files**, **Databases**, **Reports**). On the right is the content pane for the editor and for reports. And across the bottom you'll occasionally see a notification bar for messages.

You'll learn more about WebMatrix and its features as you go through these tutorials.

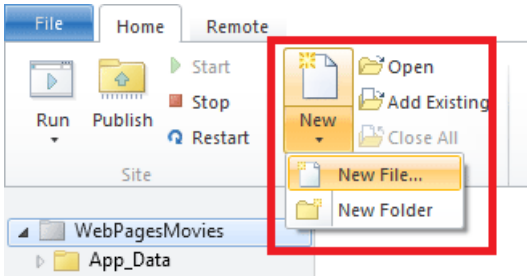
Creating a Web Page

To become familiar with WebMatrix and ASP.NET Web Pages, you'll create a simple page.

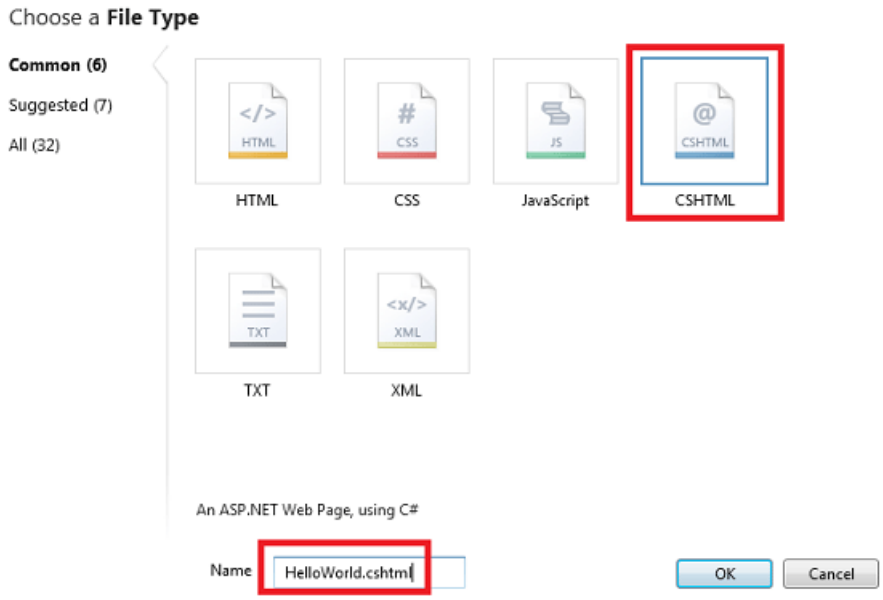
In the workspace selector, select the **Files** workspace. This workspace lets you work with files and folders. The left pane shows the file structure of your site. The ribbon changes to show file-related tasks.



In the ribbon, click the arrow under **New** and then click **New File**.

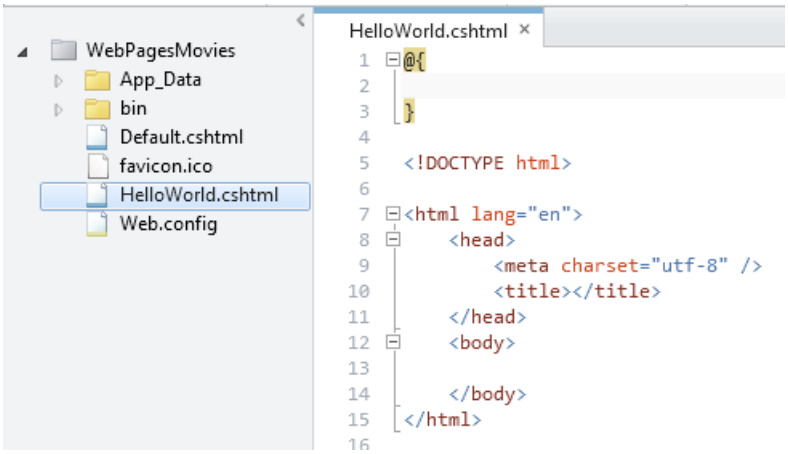


WebMatrix displays a list of file types. Select **CSHTML**, and in the **Name** box, type "HelloWorld". A CSHTML page is an ASP.NET Web Pages page.



Click **OK**.

WebMatrix creates the page and opens it in the editor.



As you can see, the page contains mostly ordinary HTML markup, except for a block at the top that looks like this:

```
@{

}
```

That's for adding code, as you'll see shortly.

Notice that the different parts of the page — the element names, attributes, and text, plus the block at the top — are all in different colors. This is called *syntax highlighting*, and it makes it easier to keep everything clear. It's one of the features that makes it easy to work with web pages in WebMatrix.

Add content for the `<head>` and `<body>` elements like in the following example. (If you want, you can just copy the following block and replace the entire existing page with this code.)

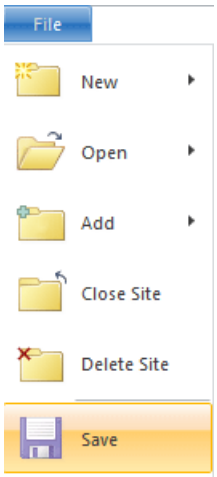
```
@{

}

<!DOCTYPE html>

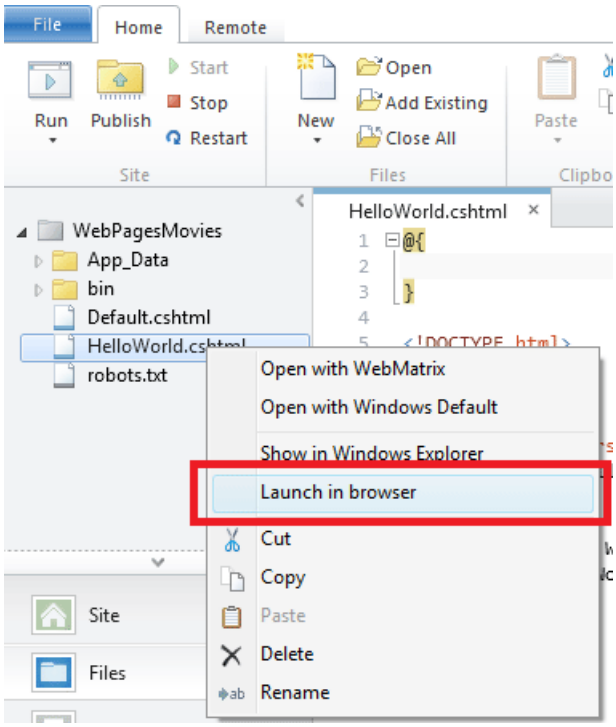
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Hello World Page</title>
    </head>
    <body>
        <h1>Hello World Page</h1>
        <p>Hello World!</p>
    </body>
</html>
```

In the Quick Access Toolbar or in the **File** menu, click **Save**.

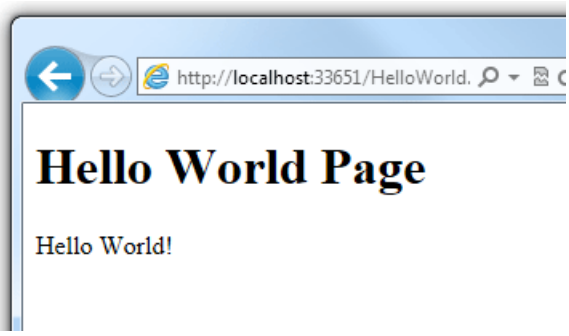


Testing the Page

In the **Files** workspace, right-click the *HelloWorld.cshtml* page and then click **Launch in browser**.



WebMatrix starts a built-in web server (IIS Express) that you can use to test pages on your computer. (Without IIS Express in WebMatrix, you'd have to publish your page to a web server somewhere before you could test it.) The page is displayed in your default browser.



Notice that when you test a page in WebMatrix, the URL in the browser is something like `http://localhost:33651/HelloWorld.cshtml.` The name *localhost* refers to a local server, meaning that the page is served by a web server that's on your own computer. As noted, WebMatrix includes a web server program named IIS Express that runs when you launch a page.

The number after *localhost* (for example, *localhost:33651*) refers to a *port number* on your computer. This is the number of the "channel" that IIS Express uses for this particular website. The port number is selected at random from the range 1024 through 65536 when you create your site, and it's different for every site that you create. (When you test your own site, the port number will almost certainly be a different number than 33561.) By using a different port for each website, IIS Express can keep straight which of your sites it's talking to.

Later when you publish your site to a public web server, you no longer see *localhost* in the URL. At that point, you'll see a more typical URL like `http://myhostingsite/mywebsite/HelloWorld.cshtml` or whatever the page is. You'll learn more about publishing a site later in this tutorial series.

Adding Some Server-Side Code

Close the browser and go back to the page in WebMatrix.

Add a line to the code block so that it looks like the following code:

```
@{
   var currentDateTime = DateTime.Now;
}
```

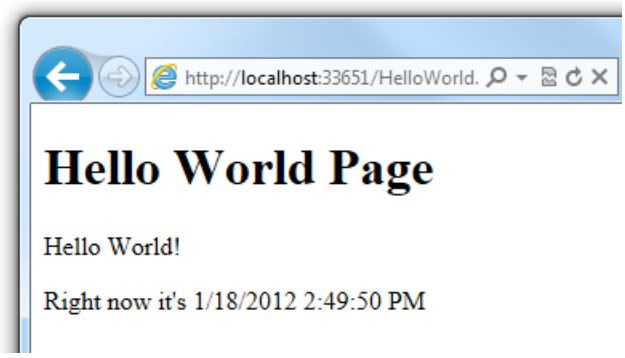
This is a little bit of Razor code. It's probably clear that it gets the current date and time and puts that value into a *variable* named `currentDateTime`. You'll read more about Razor syntax in the next tutorial.

In the body of the page, after the `<p>Hello World!</p>` element, add the following:

```
<p>Right now it's @currentDateTime</p>
```

This code gets the value that you put into the `currentDateTime` variable at the top and inserts it into the markup of the page. The `@` character marks the ASP.NET Web Pages code in the page.

Run the page again (WebMatrix saves the changes for you before it runs the page). This time you see the date and time in the page.



Wait a few moments and then refresh the page in the browser. The date and time display is updated.

In the browser, look at the page source. It looks like the following markup:

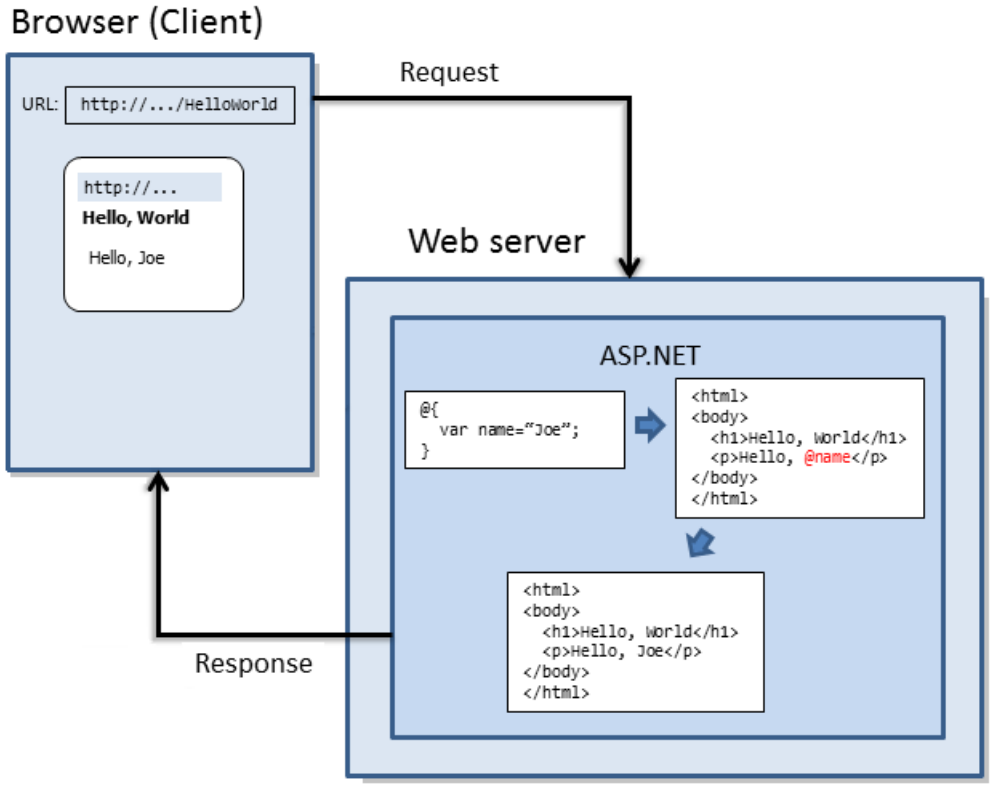
```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Hello World Page</title>
    </head>
    <body>
        <h1>Hello World Page</h1>
        <p>Hello World!</p>
        <p>Right now it's 1/18/2012 2:49:50 PM</p>
    </body>
</html>
```

Notice that the `@{ }` block at the top isn't there. Also notice that the date and time display shows an actual string of characters (`1/18/2012 2:49:50 PM` or whatever), not `@currentDateTime` like you had in the *.cshtml* page. What happened here is that when you ran the page, ASP.NET processed all the code (very little in this case) that was marked with `@`. The code produces output, and that output was inserted into the page.

This Is What ASP.NET Web Pages Are About

When you read that ASP.NET Web Pages produces dynamic web content, what you've seen here is the idea. The page you just created contains the same HTML markup that you've seen before. It can also contain code that can perform all sorts of tasks. In this example, it did the trivial task of getting the current date and time. As you saw, you can intersperse code with the HTML to produce output in the page. When someone requests a *.cshtml* page in the browser, ASP.NET processes the page while it's still in the hands of the web server. ASP.NET inserts the output of the code (if any) into the page as HTML. When the code processing is done, ASP.NET sends the resulting page to the browser. All the browser ever gets is HTML. Here's a diagram:



The idea is simple, but there are many interesting tasks that the code can perform, and there are many interesting ways in which you can dynamically add HTML content to the page. And ASP.NET *.cshtml* pages, like any HTML page, can also include code that runs in the browser itself (JavaScript and jQuery code). You'll explore all of these things in this tutorial set and in subsequent ones.

Coming Up Next

In the next tutorial in this series, you explore ASP.NET Web Pages programming a little more.

Additional Resources

Create an ASP.NET website from scratch. This is a tutorial that's specifically about using WebMatrix (not ASP.NET Web Pages). It goes into a little more detail about some of the additional features of WebMatrix that we won't cover in this tutorial set.

NEXT

Introducing ASP.NET Web Pages - Programming Basics

2/20/2020 • 24 minutes to read • Edit Online

by Tom FitzMacken

This tutorial gives you an overview of how to program in ASP.NET Web Pages with Razor syntax.

What you'll learn:

- The basic "Razor" syntax that you use for programming in ASP.NET Web Pages.
- Some basic C#, which is the programming language you'll use.
- Some fundamental programming concepts for Web Pages.
- How to install packages (components that contain prebuilt code) to use with your site.
- How to use *helpers* to perform common programming tasks.

Features/technologies discussed:

- NuGet and the package manager.
- The `Gravatar` helper.

This tutorial is primarily an exercise in introducing you to the programming syntax that you'll use for ASP.NET Web Pages. You'll learn about *Razor syntax* and code that's written in the C# programming language. You got a glimpse of this syntax in the previous tutorial; in this tutorial we'll explain the syntax more.

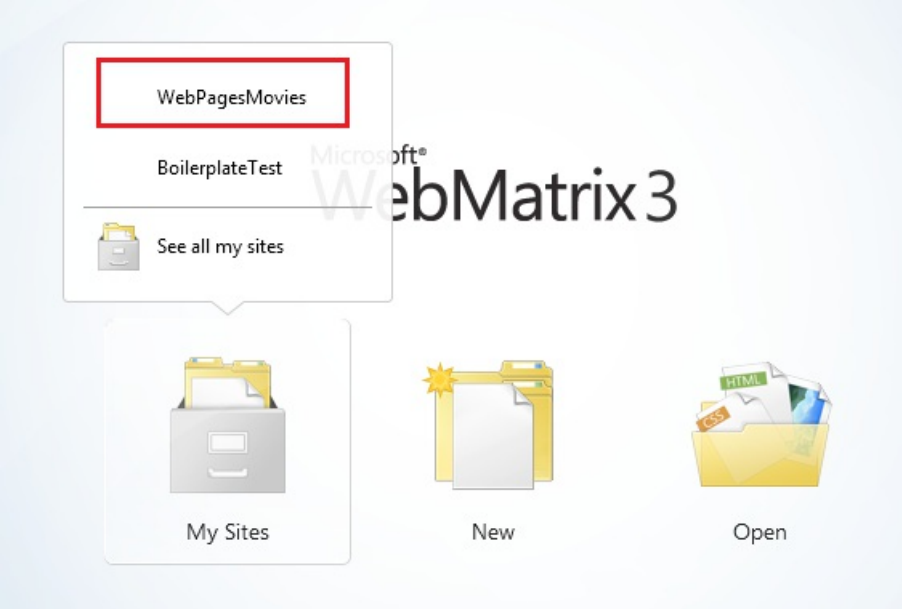
We promise that this tutorial involves the most programming that you'll see in a single tutorial, and that it's the only tutorial that is *only* about programming. In the remaining tutorials in this set, you'll actually create pages that do interesting things.

You'll also learn about *helpers*. A helper is a component — a packaged-up piece of code — that you can add to a page. The helper performs work for you that otherwise might be tedious or complex to do by hand.

Creating a Page to Play with Razor

In this section you'll play a bit with Razor so you can get a sense of the basic syntax.

Start WebMatrix if it's not already running. You'll use the website you created in the previous tutorial (Getting Started With Web Pages). To reopen it, click **My Sites** and choose **WebPageMovies**:



Select the **Files** workspace.

In the ribbon, click **New** to create a page. Select **CSHTML** and name the new page *TestRazor.cshtml*.

Click **OK**.

Copy the following into the file, completely replacing what's there already.

> **NOTE**
> When you copy code or markup from the examples into a page, the indentation and alignment might not be the same as in the tutorial. Indentation and alignment don't affect how the code runs, though.

```
@{
   // Working with numbers
   var a = 4;
   var b = 5;
   var theSum = a + b;

   // Working with characters (strings)
   var technology = "ASP.NET";
   var product ="Web Pages";

   // Working with objects
   var rightNow = DateTime.Now;
}

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Testing Razor Syntax</title>
    <meta charset="utf-8" />
    <style>
    body {font-family:Verdana; margin-left:50px; margin-top:50px;}
    div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
    span.bright {color:red;}
    </style>
  </head>
<body>
  <h1>Testing Razor Syntax</h1>
  <form method="post">

    <div>
      <p>The value of <em>a</em> is @a.  The value of <em>b</em> is @b.
      <p>The sum of <em>a</em> and <em>b</em> is <strong>@theSum</strong>.</p>
      <p>The product of <em>a</em> and <em>b</em> is <strong>@(a*b)</strong>.</p>
    </div>

    <div>
      <p>The technology is @technology, and the product is @product.</p>
      <p>Together they are <span class="bright">@(technology + " " + product)</span></p>
   </div>

   <div>
     <p>The current date and time is: @rightNow</p>
     <p>The URL of the current page is<br/><br/><code>@Request.Url</code></p>
   </div>

  </form>
</body>
</html>
```

Examining the Example Page

Most of what you see is ordinary HTML. However, at the top there's this code block:

```
@{
   // Working with numbers.
   var a = 4;
   var b = 5;
   var theSum = a + b;

   // Working with characters (strings).
   var technology = "ASP.NET";
   var product ="Web Pages";

   // Working with objects.
   var rightNow = DateTime.Now;
}
```

Notice the following things about this code block:

- The @ character tells ASP.NET that what follows is Razor code, not HTML. ASP.NET will treat everything after the @ character as code until it runs into some HTML again. (In this case, that's the <!DOCTYPE> element.
- The braces ({ and }) enclose a block of Razor code if the code has more than one line. The braces tell ASP.NET where the code for that block starts and ends.
- The // characters mark a comment — that is, a part of the code that won't execute.
- Each statement has to end with a semicolon (;). (Not comments, though.)
- You can store values in *variables*, which you create (*declare*) with the keyword var. When you create a variable, you give it a name, which can include letters, numbers, and underscore (_). Variable names can't start with a number and can't use the name of a programming keyword (like var).
- You enclose character strings (like "ASP.NET" and "Web Pages") in quotation marks. (They must be double quotation marks.) Numbers are not in quotation marks.
- Whitespace outside of quotation marks doesn't matter. Line breaks mostly don't matter; the exception is that you can't split a string in quotation marks across lines. Indentation and alignment don't matter.

Something that's not obvious from this example is that all code is case sensitive. This means that the variable TheSum is a different variable than variables that might be named theSum or thesum. Similarly, var is a keyword, but Var is not.

Objects and properties and methods

Then there's the expression DateTime.Now. In simple terms, DateTime is an *object*. An object is a thing that you can program with—a page, a text box, a file, an image, a web request, an email message, a customer record, etc. Objects have one or more *properties* that describe their characteristics. A text box object has a Text property (among others), a request object has a Url property (and others), an email message has a From property and a To property, and so on. Objects also have *methods* that are the "verbs" they can perform. You'll be working with objects a lot.

As you can see from the example, DateTime is an object that lets you program dates and times. It has a property named Now that returns the current date and time.

Using code to render markup in the page

In the body of the page, notice the following:

```
<div>
  <p>The value of <em>a</em> is @a.  The value of <em>b</em> is @b.
  <p>The sum of <em>a</em> and <em>b</em> is <strong>@theSum</strong>.</p>
  <p>The product of <em>a</em> and <em>b</em> is <strong>@(a*b)</strong>.</p>
</div>

<div>
  <p>The technology is @technology, and the product is @product.</p>
  <p>Together they are <span class="bright">@(technology + " " + product)</span></p>
</div>

<div>
  <p>The current date and time is: @rightNow</p>
     <p>The URL of the current page is<br/><br/><code>@Request.Url</code></p>
</div>
```

Again, the @ character tells ASP.NET that what follows is code, not HTML. In the markup you can add @ followed by a code expression, and ASP.NET will render the value of that expression right at that point. In the example, @a will render whatever the value is of the variable named a, @product renders whatever is in the variable named product, and so on.

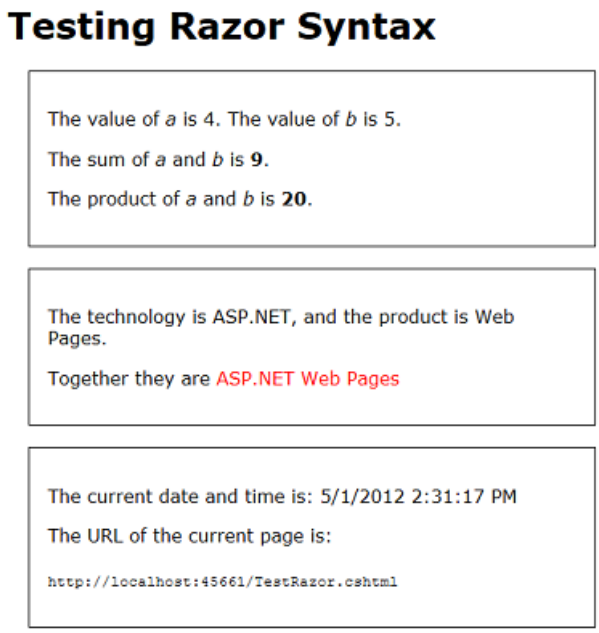
You're not limited to variables, though. In a few instances here, the @ character precedes an expression:

- @(a*b) renders the product of whatever is in the variables a and b. (The * operator means multiplication.)
- @(technology + " " + product) renders the values in the variables technology and product after concatenating them and adding a space in between. The operator (+) for concatenating strings is the same as the operator for adding numbers. ASP.NET can usually tell whether you're working with numbers or with strings and does the right thing with the + operator.
- @Request.Url renders the Url property of the Request object. The Request object contains information about the current request from the browser, and of course the Url property contains the URL of that current request.

The example is also designed to show you that you can do work in different ways. You can do calculations in the code block at the top, put the results into a variable, and then render the variable in markup. Or you can do calculations in an expression right in the markup. The approach you use depends on what you're doing and, to some extent, on your own preference.

Seeing the code in action

Right-click the name of the file and then choose **Launch in browser**. You see the page in the browser with all the values and expressions resolved in the page.



Look at the source in the browser.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Testing Razor Syntax</title>
    <meta charset="utf-8" />
    <style>
    body {font-family:Verdana; margin-left:50px; margin-top:50px;}
    div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
    span.bright {color:red;}
    </style>
  </head>
<body>
  <h1>Testing Razor Syntax</h1>
  <form method="post">

    <div>
      <p>The value of <em>a</em> is 4.  The value of <em>b</em> is 5.
      <p>The sum of <em>a</em> and <em>b</em> is <strong>9</strong></p>
      <p>The product of <em>a</em> and <em>b</em> is <strong>20</strong></p>
    </div>

    <div>
      <p>The technology is ASP.NET, and the product is Web Pages.</p>
      <p>Together they are <span class="bright">ASP.NET Web Pages</span></p>
   </div>

   <div>
     <p>The current date and time is: 4/29/2012 11:44:02 PM</p>
     <p>The URL of the current page is <code>http://localhost:45661/TestRazor.cshtml</code></p>
   </div>

  </form>
</body>
</html>
```

As you expect from your experience in the previous tutorial, none of the Razor code is in the page. All you see are the actual display values. When you run a page, you're actually making a request to the web server that's built into WebMatrix. When the request is received, ASP.NET resolves all the values and expressions and renders their values into the page. It then sends the page to the browser.

TIP

Razor and C#

Up to now we've said that you're working with Razor syntax. That's true, but it's not the complete story. The actual programming language you're using is called *C#*. C# was created by Microsoft over a decade ago and has become one of the primary programming languages for creating Windows apps. All the rules you've seen about how to name a variable and how to create statements and so on are actually all rules of the C# language.

Razor refers more specifically to the small set of conventions for how you embed this code into a page. For example, the convention of using @ to mark code in the page and using @{ } to embed a code block is the Razor aspect of a page. Helpers are also considered to be part of Razor. Razor syntax is used in more places than just in ASP.NET Web Pages. (For example, it's used in ASP.NET MVC views as well.)

We mention this because if you look for information about programming ASP.NET Web Pages, you'll find lots of references to Razor. However, a lot of those references don't apply to what you're doing and might therefore be confusing. And in fact, many of your programming questions are really going to be about either working with C# or working with ASP.NET. So if you look specifically for information about Razor, you might not find the answers you need.

Adding Some Conditional Logic

One of the great features about using code in a page is that you can change what happens based on various conditions. In this part of the tutorial, you'll play around with some ways to change what's displayed in the page.

The example will be simple and somewhat contrived so that we can concentrate on the conditional logic. The page you'll create will do this:

- Show different text on the page depending on whether it's the first time the page is displayed or whether you've clicked a button to submit the page. That will be the first conditional test.
- Display the message only if a certain value is passed in the query string of the URL (http://...?show=true). That will be the second conditional test.

In WebMatrix, create a page and name it *TestRazorPart2.cshtml*. (In the ribbon, click **New**, choose **CSHTML**, name the file, and then click **OK**.)

Replace the contents of that page with the following:

```
@{
   var message = "This is the first time you've requested the page.";
}
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Testing Razor Syntax - Part 2</title>
    <meta charset="utf-8" />
    <style>
      body {font-family:Verdana; margin-left:50px; margin-top:50px;}
      div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
    </style>
  </head>
  <body>
  <h1>Testing Razor Syntax - Part 2</h1>
  <form method="post">
    <div>
      <p>@message</p>
      <p><input type="submit" value="Submit" /></p>
  </div>
  </form>
</body>
</html>
```

The code block at the top initializes a variable named message with some text. In the body of the page, the contents of the message variable are displayed inside a <p> element. The markup also contains an <input> element to create a **Submit** button.

Run the page to see how it works now. For now, it's basically a static page, even if you click the **Submit** button.

Go back to WebMatrix. Inside the code block, add the following highlighted code *after* the line that initializes message:

```
@{
    var message = "This is the first time you've requested the page.";

    if(IsPost) {
        message = "Now you've submitted the page.";
    }
}
```

The if { } block

What you just added was an if condition. In code, the if condition has a structure like this:

```
if(some condition){
    One or more statements here that run if the condition is true;
}
```

The condition to test is in parentheses. It has to be a value or an expression that returns true or false. If the condition is true, ASP.NET runs the statement or statements that are inside the braces. (Those are the *then* part of the *if-then* logic.) If the condition is false, the block of code is skipped.

Here are a few examples of conditions you can test in an if statement:

```
if(currentValue > 12) { ... }

if(dueDate <= DateTime.Today) { ... }

if(IsDone == true) { ... }

if(IsPost) { ... }

if(!IsPost) { ... }

if(a != 0) { ... }

if(fileProcessingIsDone != true && displayMessage == false) { ... }
```

You can test variables against values or against expressions by using a *logical operator* or *comparison operator*: equal to (==), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=). The != operator means not equal to — for example, if(a != 0) means *if a is not equal to 0*.

> **NOTE**
> Make sure you notice that the comparison operator for equals to (==) is not the same as =. The = operator is used only to assign values (var a=2). If you mix these operators up, you'll either get an error or you'll get some strange results.

To test whether something is true, the complete syntax is if(IsDone == true). But you can also use the shortcut if(IsDone). If there's no comparison operator, ASP.NET assumes that you're testing for true.

The ! operator by itself means a logical NOT. For example, the condition if(!IsPost) means *if IsPost is not true*.

You can combine conditions by using a logical AND (&& operator) or logical OR (|| operator). For example, the last of the if conditions in the preceding examples means *if FileProcessingIsDone is not set to true AND displayMessage is set to false*.

The else block

One final thing about if blocks: an if block can be followed by an else block. An else block is useful is you have to execute different code when the condition is false. Here's a simple example:

```
var message = "";
if(errorOccurred == true)
{
    message = "Sorry, an error occurred.";
}
else
{
    message = "The process finished without errors!";
}
```

You'll see some examples in later tutorials in this series where using an else block is useful.

Testing whether the request is a submit (post)

There's more, but let's get back to the example, which has the condition if(IsPost){ ... }. IsPost is actually a property of the current page. The first time the page is requested, IsPost returns false. However, if you click a button or otherwise submit the page — that is, you post it — IsPost returns true. So IsPost lets you determine whether you're dealing with a form submission. (In terms of HTTP verbs, if the request is a GET operation, IsPost returns false. If the request is a POST operation, IsPost returns true.) In a later tutorial you'll work with input forms, where this test becomes particularly useful.

Run the page. Because this is the first time you're requested the page, you see "This is the first time you've

requested the page". That string is the value that you initialized the message variable to. There's an if(IsPost) test, but that returns false at the moment, so the code inside the if block doesn't run.

Click the **Submit** button. The page is requested again. As before, the message variable is set to "This is the first time ...". But this time, the test if(IsPost) returns true, so the code inside the if block runs. The code changes the value of the message variable to a different value, which is what's rendered in the markup.

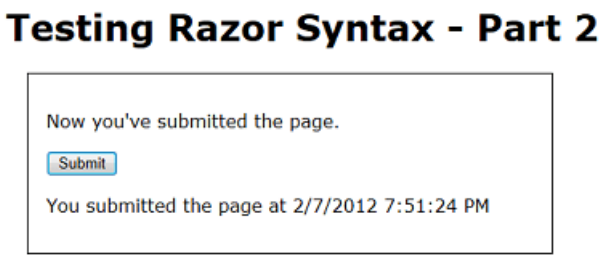
Now add an if condition in the markup. Below the <p> element that contains the **Submit** button, add the following markup:

```
@if(IsPost){
  <p>You submitted the page at @DateTime.Now</p>
}
```

You're adding code inside the markup, so you have to start with @. Then there's an if test similar to the one you added earlier up in the code block. Inside the braces, though, you're adding ordinary HTML — at least, it's ordinary until it gets to @DateTime.Now. This is another little bit of Razor code, so again you have to add @ in front of it.

The point here is that you can add if conditions in both the code block at the top and in the markup. If you use an if condition in the body of the page, the lines inside the block can be markup or code. In that case, and as is true anytime you mix markup and code, you have to use @ to make it clear to ASP.NET where the code is.

Run the page and click **Submit**. This time you not only see a different message when you submit ("Now you've submitted ..."), but you see a new message that lists the date and time.



Testing the value of a query string

One more test. This time, you'll add an if block that tests a value named show that might be passed in the query string. (Like this: `http://localhost:43097/TestRazorPart2.cshtml?show=true`) You'll change the page so that the message you've been displaying ("This is the first time ...", etc.) is only displayed if the value of show is true.

At the bottom (but inside) the code block at the top of the page, add the following:

```
var showMessage = false;
if(Request.QueryString["show"].AsBool() == true) {
    showMessage = true;
}
```

The complete code block now look like the following example. (Remember that when you copy the code into your page, the indentation might look different. But that doesn't affect how the code runs.)

```
@{
   var message = "This is the first time you've requested the page.";

   if(IsPost) {
      message = "Now you've submitted the page.";
   }

   var showMessage = false;
   if(Request.QueryString["show"].AsBool() == true) {
     showMessage = true;
   }
}
```

The new code in the block initializes a variable named showMessage to false. It then does an if test to look for a value in the query string. When you first request the page, it has a URL like this one:

`http://localhost:43097/TestRazorPart2.cshtml`

The code determines whether the URL contains a variable named show in the query string, like this version of the URL:

`http://localhost:43097/TestRazorPart2.cshtml`?show=true

The test itself looks at the QueryString property of the Request object. If the query string contains an item named show, and if that item is set to true, the if block runs and sets the showMessage variable to true.

There's a trick here, as you can see. Like the name says, the query string is a string. However, you can only test for true and false if the value you're testing is a Boolean (true/false) value. Before you can test the value of the show variable in the query string, you have to convert it to a Boolean value. That's what the AsBool method does — it takes a string as input and converts it to a Boolean value. Clearly, if the string is "true", the AsBool method converts that value to true. If the value of the string is anything else, AsBool returns false.

TIP

Data Types and As() Methods

We've only said so far that when you create a variable, you use the keyword var. That's not the entire story, though. In order to manipulate values — to add numbers, or concatenate strings, or compare dates, or test for true/false — C# has to work with an appropriate internal representation of the value. C# can *usually* figure out what that representation should be (that is, what *type* the data is) based on what you're doing with the values. Now and then, though, it can't do that. If not, you have to help out by explicitly indicating how C# should represent the data. The AsBool method does that — it tells C# that a string value of "true" or "false" should be treated as a Boolean value. Similar methods exist to represent strings as other types as well, like AsInt (treat as an integer), AsDateTime (treat as a date/time), AsFloat (treat as a floating-point number), and so on. When you use these As( ) methods, if C# can't represent the string value as requested, you'll see an error.

In the markup of the page, remove or comment out this element (here it's shown commented out):

```
<!-- <p>@message</p> -->
```

Right where you removed or commented out that text, add the following:

```
@if(showMessage) {
  <p>@message</p>
}
```

The if test says that if the showMessage variable is true, render a <p> element with the value of the message variable.

Summary of your conditional logic

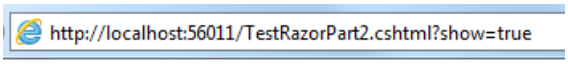
In case you're not entirely sure of what you've just done, here's a summary.

- The message variable is initialized to a default string ("This is the first time ...").
- If the page request is the result of a submit (post), the value of message is changed to "Now you've submitted ..."
- The showMessage variable is initialized to false.
- If the query string contains ?show=true, the showMessage variable is set to true.
- In the markup, if showMessage is true, a <p> element is rendered that shows the value of message. (If showMessage is false, nothing is rendered at that point in the markup.)
- In the markup, if the request is a post, a <p> element is rendered that displays the date and time.

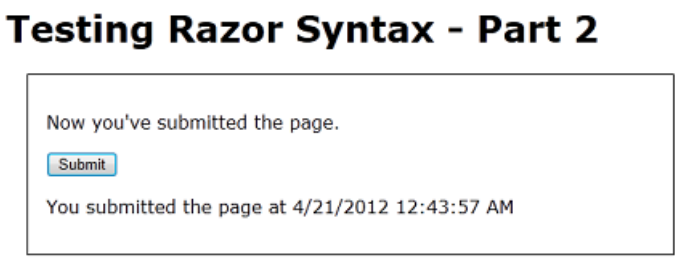
Run the page. There's no message, because showMessage is false, so in the markup the if(showMessage) test returns false.

Click **Submit**. You see the date and time, but still no message.

In your browser, go to the URL box and add the following to the end of the URL: ?show=true and then press Enter.



The page is displayed again. (Because you changed the URL, this is a new request, not a submit.) Click **Submit** again. The message is displayed again, as is the date and time.



In the URL, change ?show=true to ?show=false and press Enter. Submit the page again. The page is back to how you started — no message.

As noted earlier, the logic of this example is a little contrived. However, if is going to come up in many of your pages, and it will take one or more of the forms you've seen here.

Installing a Helper (Displaying a Gravatar Image)

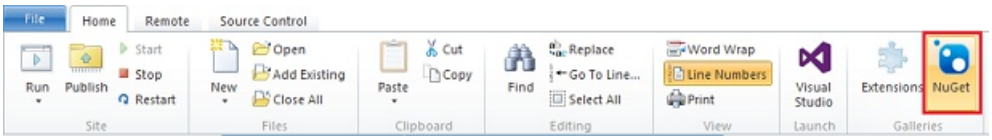
Some tasks that people often want to do on web pages require a lot of code or require extra knowledge. Examples: displaying a chart for data; putting a Facebook "Like" button on a page; sending email from your website; cropping or resizing images; using PayPal for your site. To make it easy to do these kinds of things, ASP.NET Web Pages lets you use *helpers*. Helpers are components that you install for a site and that let you perform typical tasks by using just a few lines of Razor code.

ASP.NET Web Pages has a few helpers built in. However, many helpers are available in packages (add-ins) that are provided using the NuGet package manager. NuGet lets you select a package to install and then it takes care of all the details of the installation.

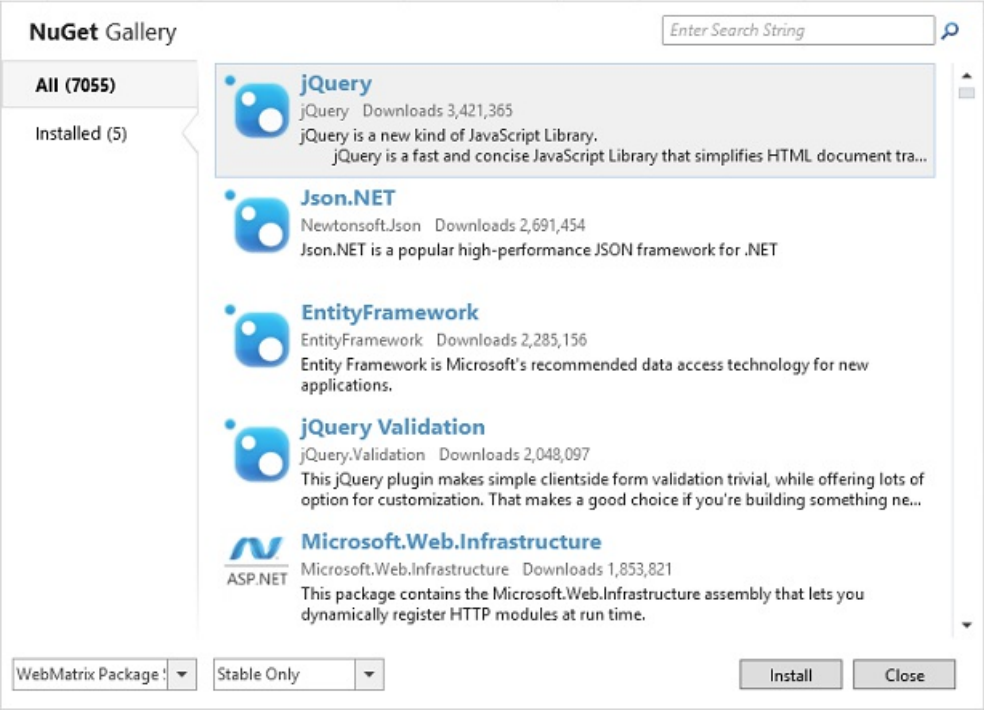
In this part of the tutorial, you'll install a helper that lets you display a Gravatar ("globally recognized avatar") image. You'll learn two things. One is how to find and install a helper. You'll also learn how a helper makes it easy to do something you'd otherwise need to do by using a lot of code you'd have to write yourself.

You can register your own Gravatar at the Gravatar website at http://www.gravatar.com/, but it is not essential to create a Gravatar account to perform this part of the tutorial.

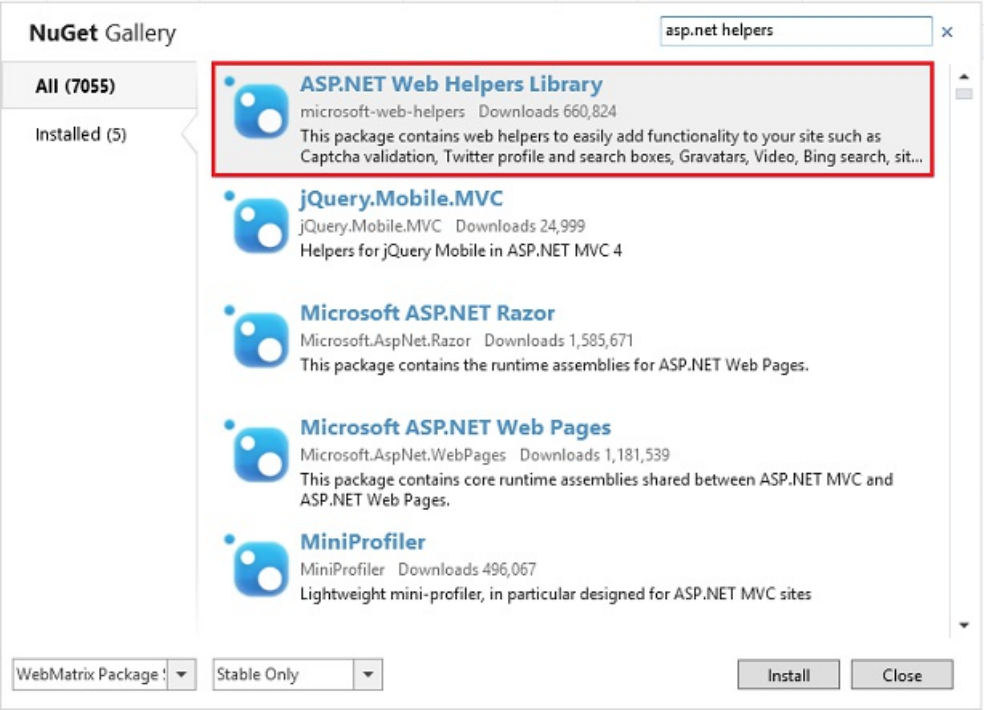
In WebMatrix, click the **NuGet** button.



This launches the NuGet package manager and displays available packages. (Not all of the packages are helpers; some add functionality to WebMatrix itself, some are additional templates, and so on.) You may get an error message about version incompatibility. You can ignore this error message by clicking **OK** and proceeding with this tutorial.



In the search box, enter "asp.net helpers". NuGet shows the packages that match the search terms.



The ASP.NET Web Helpers Library contains code to simplify many common tasks, including the use of Gravatar

images. Select the **ASP.NET Web Helpers Library** package and then click **Install** to launch the installer. Select **Yes** when asked if you want to install the package, and accept the terms to complete the installation.

That's it. NuGet downloads and installs everything, including any additional components that might be required (*dependencies*).

If for some reason you have to uninstall a helper, the process is very similar. Click the **NuGet** button, click the **Installed** tab, and pick the package you want to uninstall.

Using a Helper in a Page

Now you'll use the helper that you just installed. The process for adding a helper to a page is similar for most helpers.

In WebMatrix, create a page and name it *GravatarTest.cshml*. (You're creating a special page to test the helper, but you can use helpers in any page in your site.)

Inside the <body> element, add a element. Inside the element, type this:

@Gravatar.

The @ character is the same character you've been using to mark Razor code. **Gravatar** is the helper object that you're working with.

As soon as you type the period (.), WebMatrix displays a list of *methods* (functions) that the Gravatar helper makes available:

```
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title></title>
    </head>
    <body>
        <div>
            @Gravatar.
        </div>
    </body>
</html>
```

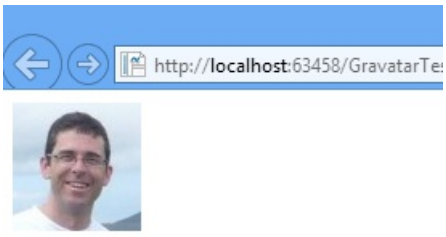
Equals
GetHtml
GetUrl
ReferenceEquals

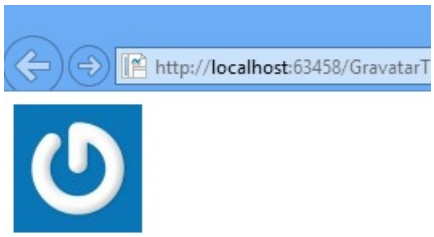
This feature is known as *IntelliSense*. It helps you code by providing context-appropriate choices. IntelliSense works with HTML, CSS, ASP.NET code, JavaScript, and other languages that are supported in WebMatrix. It's another feature that makes it easier to develop web pages in WebMatrix.

Press G on the keyboard, and you see that IntelliSense finds the GetHtml method. Press Tab. IntelliSense inserts the selected method (GetHtml) for you. Type an open parenthesis, and notice that the closing parenthesis is automatically added. Type your email address in quotation marks between the two parenthesis. If you have a Gravatar account, your profile picture will be returned. If you do not have a Gravatar account, a default image is returned. When you're done, the line looks like this:

```
@Gravatar.GetHtml("john@contoso.com")
```

Now view the page in a browser. Either your picture or the default image is displayed, depending on whether you have a Gravatar account.



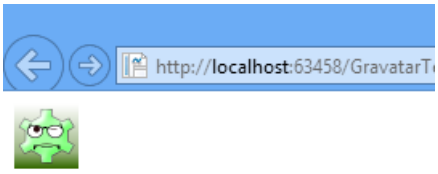


To get an idea of what the helper is doing for you, view the source of the page in the browser. Along with the HTML that you had in your page, you see an image element that includes an identifier. This is code that the helper rendered into the page at the place where you had @Gravatar.GetHtml. The helper took the information you provided and generated the code that talks directly to Gravatar in order to get back the correct image for supplied account.

The GetHtml method also enables you to customize the image by providing other parameters. The following code shows how to request an image has a width and height of 40 pixels, and uses a specified default image named **wavatar** if the specified account does not exist.

```
@Gravatar.GetHtml("john@contoso.com", 40, "wavatar")
```

This code produces something like the following result (the default image will randomly vary).



Coming Up Next

To keep this tutorial short, we had to focus on only a few basics. Naturally, there's a *lot* more to Razor and C#. You'll learn more as you go through these tutorials. If you're interested in learning more about the programming aspects of Razor and C# right now, you can read a more thorough introduction here: Introduction to ASP.NET Web Programming Using the Razor Syntax.

The next tutorial introduces you to working with a database. In that tutorial, you'll begin creating the sample application that lets you list your favorite movies.

Complete Listing for TestRazor Page

```
@{
   // Working with numbers
   var a = 4;
   var b = 5;
   var theSum = a + b;

   // Working with characters (strings)
   var technology = "ASP.NET";
   var product ="Web Pages";

   // Working with objects
   var rightNow = DateTime.Now;
}

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Testing Razor Syntax</title>
    <meta charset="utf-8" />
    <style>
    body {font-family:Verdana; margin-left:50px; margin-top:50px;}
    div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
    span.bright {color:red;}
    </style>
  </head>
<body>
  <h1>Testing Razor Syntax</h1>
  <form method="post">

    <div>
      <p>The value of <em>a</em> is @a.  The value of <em>b</em> is @b.
      <p>The sum of <em>a</em> and <em>b</em> is <strong>@theSum</strong>.</p>
      <p>The product of <em>a</em> and <em>b</em> is <strong>@(a*b)</strong>.</p>
    </div>

    <div>
      <p>The technology is @technology, and the product is @product.</p>
      <p>Together they are <span class="bright">@(technology + " " + product)</span></p>
   </div>

   <div>
     <p>The current date and time is: @rightNow</p>
     <p>The URL of the current page is<br/><br/><code>@Request.Url</code></p>
   </div>

  </form>
</body>
</html>
```

Complete Listing for TestRazorPart2 Page

```
@{
   var message = "This is the first time you've requested the page.";

   if (IsPost) {
      message = "Now you've submitted the page.";
   }

   var showMessage = false;
   if (Request.QueryString["show"].AsBool() == true) {
     showMessage = true;
   }
}

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Testing Razor Syntax - Part 2</title>
    <meta charset="utf-8" />
    <style>
      body {font-family:Verdana; margin-left:50px; margin-top:50px;}
      div {border: 1px solid black; width:50%; margin:1.2em;padding:1em;}
    </style>
  </head>
  <body>
  <h1>Testing Razor Syntax - Part 2</h1>
  <form method="post">
    <div>
      <!--<p>@message</p>-->
      @if(showMessage){
        <p>@message</p>
      }
      <p><input type="submit" value="Submit" /></p>
      @if (IsPost) {
        <p>You submitted the page at @DateTime.Now</p>
      }
    </div>
  </form>
</body>
</html>
```

Complete Listing for GravatarTest Page

```
@{
}
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title></title>
    </head>
    <body>
        <div>
          @Gravatar.GetHtml("john@contoso.com")
          @Gravatar.GetHtml("john@contoso.com", 40, "wavatar")
        </div>
    </body>
</html>
```

Additional Resources

- Introduction to ASP.NET Web Programming Using the Razor Syntax
- Twitter helper

Introducing ASP.NET Web Pages - Displaying Data

2/20/2020 • 18 minutes to read • Edit Online

by Tom FitzMacken

This tutorial shows you how to create a database in WebMatrix and how to display database data in a page when you use ASP.NET Web Pages (Razor). It assumes you have completed the series through Introduction to ASP.NET Web Pages Programming.

What you'll learn:

- How to use WebMatrix tools to create a database and database tables.
- How to use WebMatrix tools to add data to a database.
- How to display data from the database on a page.
- How to run SQL commands in ASP.NET Web Pages.
- How to customize the `WebGrid` helper to change the data display and to add paging and sorting.

Features/technologies discussed:

- WebMatrix database tools.
- `WebGrid` helper.

What You'll Build

In the previous tutorial, you were introduced to ASP.NET Web Pages (*.cshtml* files), to the basics of Razor syntax, and to helpers. In this tutorial, you'll begin creating the actual web application that you'll use for the rest of the series. The app is a simple movie application that lets you view, add, change, and delete information about movies.

When you're done with this tutorial, you'll be able to view a movie listing that looks like this page:

Movies

Title	Genre	Year
When Harry Met Sally	Romantic Comedy	1989
Gone With The Wind	Drama	1939
Ghostbusters	Comedy	1984
Fantasia	Children	1939
Clueless	Comedy	1995
Silverado	Western	1985
The Three Musketeers	Adventure	1973
Ronin	Action	1998

But to begin, you have to create a database.

A Very Brief Introduction to Databases

This tutorial will provide only the briefest introduction to databases. If you have database experience, you can skip this short section.

A database contains one or more tables that contain information — for example, tables for customers, orders, and vendors, or for students, teachers, classes, and grades. Structurally, a database table is like a spreadsheet. Imagine

a typical address book. For each entry in the address book (that is, for each person) you have several pieces of information such as first name, last name, address, email address, and phone number.

ID	FirstName	LastName	Address	Email	Phone
1	Lisa	Miller	4567 Main St	lisa@adatum.com	555-0199
2	Patrick	Hines	123 Elm St	patrickh@tailspintoys.com	555-0111
3	Julia	Ilyina	89 Oak Ave	julia@fourthcoffee.com	555-0166
4	Mark	Alexieff	678 8th St	marka@proseware.com	555-0129
5	Jim	Daly	345 Broad Blvd	jdaly@contoso.com	555-0188

(Rows are sometimes referred to as *records*, and columns are sometimes referred to as *fields*.)

For most database tables, the table has to have a column that contains a unique value, like a customer number, account number, and so on. This value is known as the table's *primary key*, and you use it to identify each row in the table. In the example, the ID column is the primary key for the address book shown in the previous example.

Much of the work you do in web applications consists of reading information out of the database and displaying it on a page. You'll also often gather information from users and add it to a database, or you'll modify records that are already in the database. (We'll cover all of these operations in the course of this tutorial set.)

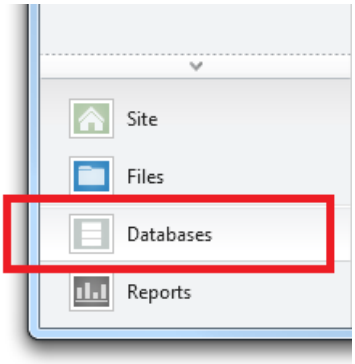
Database work can be enormously complex and can require specialized knowledge. For this tutorial set, though, you have to understand only basic concepts, which will all be explained as you go.

Creating a Database

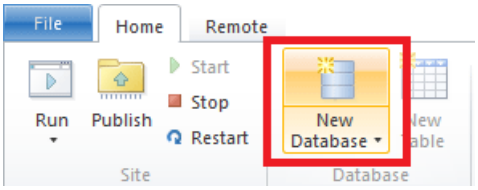
WebMatrix includes tools that make it easy to create a database and to create tables in the database. (The structure of a database is referred to as the database's *schema*.) For this tutorial set, you'll create a database that has only one table in it — Movies.

Open WebMatrix if you haven't already done so, and open the WebPagesMovies site that you created in the previous tutorial.

In the left pane, click the **Database** workspace.



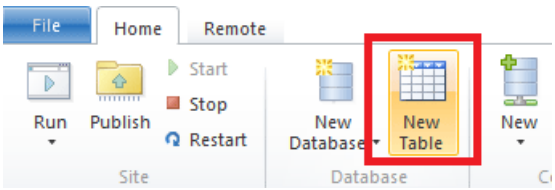
The ribbon changes to show database-related tasks. In the ribbon, click **New Database**.



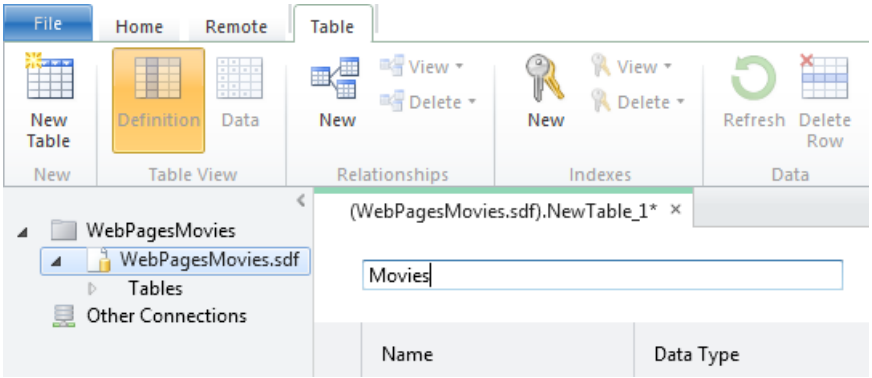
WebMatrix creates a SQL Server CE database (an *.sdf* file) that has the same name as your site — *WebPagesMovies.sdf*. (You won't do this here, but you can rename the file to anything you like, as long as it has an *.sdf* extension.)

Creating a Table

In the ribbon, click **New Table**. WebMatrix opens the table designer in a new tab. (If the New Table option isn't available, make sure that the new database is selected in the tree view on the left.)



In the text box at the top (where the watermark says "Enter table name"), enter "Movies".



The pane underneath the table name is where you define individual columns. For the *Movies* table in this tutorial, you'll create only a few columns: *ID*, *Title*, *Genre*, and *Year*.

In the **Name** box, enter "ID". Entering a value here activates all the controls for the new column.

Tab to the **Data Type** list and choose **int**. This value specifies that the ID column will contain integer (number) data.

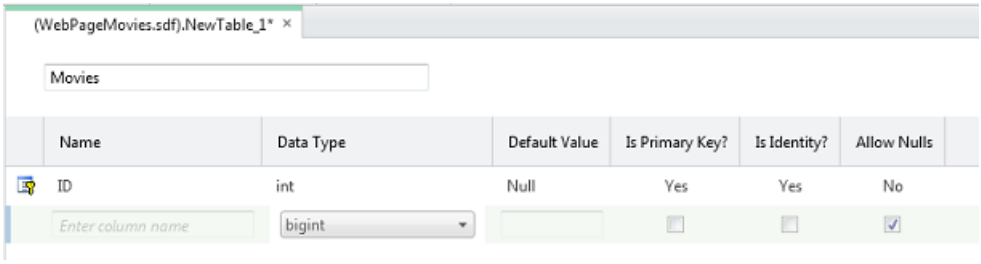
> **NOTE**
>
> We won't call it out any more here (much), but you can use standard Windows keyboard gestures to navigate in this grid. For example, you can tab between fields, you can just start typing in order to select an item in a list, and so on.

Tab past the **Default Value** box (that is, leave it blank). Tab to the **Is Primary Key** check box and select it. This option tells the database that the *ID* column will contain the data that identifies individual rows. (That is, each row will have a unique value in the ID column that you can use to find that row.)

Choose the **Is Identity** option. This option tells the database that it should automatically generate the next sequential number for each new row. (The **Is Identity** option works only if you've also selected the **Is Primary Key** option.)

Click in the next grid row, or press Tab twice to finish the current row. Either gesture saves the current row and starts the next one. Notice that the **Default Value** column now says **Null**. (Null is the default value for the default value, so to speak.)

When you've finished defining the new **ID** column, the designer will look like this illustration:



To create the next column, click in the box in the **Name** column. Enter "Title" for the column and then select

nvarchar for the **Data Type** value. The "var" part of **nvarchar** tells the database that the data for this column will be a string whose size might vary from record to record. (The "n" prefix represents "national," which indicates that the field can hold character data for any alphabet or writing system — that is, the field holds Unicode data.)

When you choose **nvarchar**, another box appears where you can enter the maximum length for the field. Enter 50, on the assumption that no movie title that you'll work with in this tutorial will be longer than 50 characters.

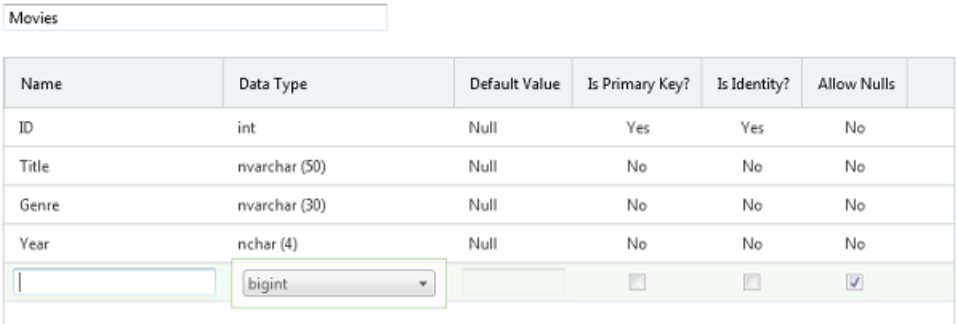
Skip **Default Value** and clear the **Allow Nulls** option. You don't want the database to allow any movies to be entered into the database that don't have a title.

When you're done and move to the next row, the designer looks like this illustration:

Repeat these steps to create a column named "Genre", except for the length, set it to just 30.

Create another column named "Year." For the data type, choose **nchar** (not **nvarchar**) and set the length to 4. For the year, you're going to use a 4-digit number like "1995" or "2010", so you don't require a variable-sized column.

Here's what the finished design looks like:

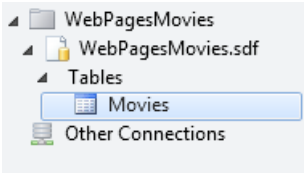


Press Ctrl+S or click the **Save** button in the Quick Access toolbar. Close the database designer by closing the tab.

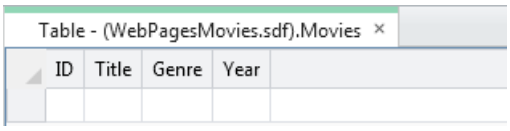
Adding Some Example Data

Later in this tutorial series you'll create a page where you can enter new movies in a form. For now, however, you can add some example data that you can then display on a page.

In the **Database** workspace in WebMatrix, notice that there's a tree that shows you the *.sdf* file you created earlier. Open the node for your new *.sdf* file, and then open the **Tables** node.



Right-click the **Movies** node and then choose **Data**. WebMatrix opens a grid where you can enter data for the *Movies* table:



Click the **Title** column and enter "When Harry Met Sally". Move to the **Genre** column (you can use the Tab key) and enter "Romantic Comedy". Move to the **Year** column and enter "1989":

Table - (WebPagesMovies.sdf).Movies

ID	Title	Genre	Year
NULL	When Harry Met Sally	Romantic Comedy	1989

Press Enter, and WebMatrix saves the new movie. Notice that the **ID** column has been filled in.

Table - (WebPagesMovies.sdf).Movies

ID	Title	Genre	Year
1	When Harry Met Sally	Romantic Comedy	1989

Enter another movie (for example, "Gone with the Wind", "Drama", "1939"). The ID column is filled in again:

Table - (WebPagesMovies.sdf).Movies

ID	Title	Genre	Year
1	When Harry Met Sally	Romantic Comedy	1989
2	Gone With The Wind	Drama	1939

Enter a third movie (for example, "Ghostbusters", "Comedy"). As an experiment, leave the **Year** column blank and then press Enter. Because you unselected the **Allow Nulls** option, the database shows an error:

Invalid data

Your change could not be committed to the database. After you click OK you can fix the invalid entry or press the Esc key to cancel your changes.

The column cannot contain null values. [Column name = Year,Table name = Movies]

Copy full details to clipboard

OK

Click **OK** to go back and fix the entry (the year for "Ghostbusters" is 1984), and then press Enter.

Fill in several movies until you have 8 or so. (Entering 8 makes it easier to work with paging later. But if that's too many, enter just a few for now.) The actual data doesn't matter.

Table - (WebPagesMovies.sdf).Movies

ID	Title	Genre	Year
1	When Harry Met Sally	Romantic Comedy	1989
2	Gone With The Wind	Drama	1939
4	Ghostbusters	Comedy	1984
5	Fantasia	Children	1939
8	Clueless	Comedy	1995
9	Silverado	Western	1985
10	The Three Musketeers	Adventure	1973
11	Ronin	Action	1998

If you entered all the movies without any errors, the ID values are sequential. If you tried to save an incomplete movie record, the ID numbers might not be sequential. If so, that's okay. The numbers don't have any inherent meaning, and the only thing that's important is that they're unique in the *Movies* table.

Close the tab that contains the database designer.

Now you can turn to displaying this data on a web page.

Displaying Data in a Page by Using the WebGrid Helper

To display data in a page, you're going to use the `WebGrid` helper. This helper produces a display in a grid or table (rows and columns). As you'll see, you'll be able refine the grid with formatting and other features.

To run the grid, you'll have to write a few lines of code. These few lines will serve as a kind of pattern for almost all of the data access that you do in this tutorial.

> **NOTE**
> You actually have many options for displaying data on a page; the `WebGrid` helper is just one. We chose it for this tutorial because it's the easiest way to display data and because it's reasonably flexible. In the next tutorial set, you'll see how to use a more "manual" way to work with data in the page, which gives you more direct control over how to display the data.

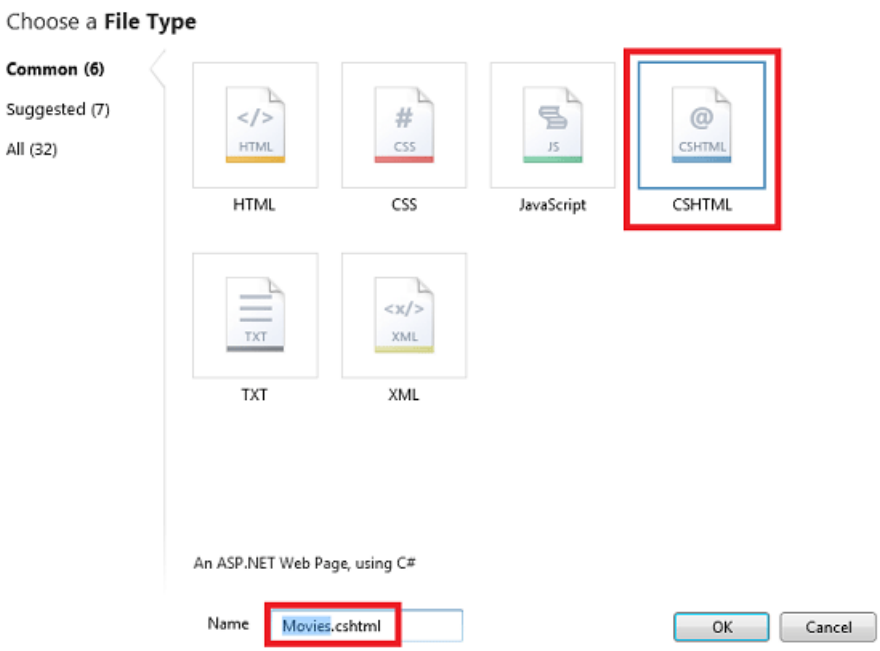
In the left pane in WebMatrix, click the **Files** workspace.

The new database you created is in the *App_Data* folder. If the folder didn't already exist, WebMatrix created it for your new database. (The folder might have existed if you'd previously installed helpers.)

In the tree view, select the root of the website. You must select the root of the website; otherwise, the new file might be added to the App_Data folder.

In the ribbon, click **New**. In the **Choose a File Type** box, choose **CSHTML**.

In the **Name** box, name the new page "Movies.cshtml":



Click the **OK** button. WebMatrix opens a new file with some skeleton elements in it. First you'll write some code to go get the data from the database. Then you'll add markup to the page to actually display the data.

Writing the Data Query Code

At the top of the page, between the `@{` and `}` characters, enter the following code. (Make sure that you enter this code between the opening and closing braces.)

```
var db = Database.Open("WebPagesMovies");
var selectedData = db.Query("SELECT * FROM Movies");
var grid = new WebGrid(source: selectedData);
```

The first line opens the database that you created earlier, which is always the first step before doing something with the database. You tell the `Database.Open` method name of the database to open. Notice that you don't include

.sdf in the name. The `Open` method assumes that it's looking for an *.sdf* file (that is, *WebPagesMovies.sdf*) and that the *.sdf* file is in the *App_Data* folder. (Earlier we noted that the *App_Data* folder is reserved; this scenario is one of the places where ASP.NET makes assumptions about that name.)

When the database is opened, a reference to it is put into the variable named `db`. (Which could be named anything.) The `db` variable is how you'll end up interacting with the database.

The second line actually fetches the database data by using the `Query` method. Notice how this code works: the `db` variable has a reference to the opened database, and you invoke the `Query` method by using the `db` variable (`db.Query`).

The query itself is a SQL `Select` statement. (For a little background about SQL, see the explanation later.) In the statement, `Movies` identifies the table to query. The `*` character specifies that the query should return all the columns from the table. (You could also list columns individually, separated by commas.)

The results of the query, if any, are returned and made available in the `selectedData` variable. Again, the variable could be named anything.

Finally, the third line tells ASP.NET that you want to use an instance of the `WebGrid` helper. You create (*instantiate*) the helper object by using the `new` keyword and pass it the query results via the `selectedData` variable. The new `WebGrid` object, along with the results of the database query, are made available in the `grid` variable. You'll need that result in a moment to actually display the data in the page.

At this stage, the database has been opened, you've gotten the data you want, and you've prepared the `WebGrid` helper with that data. Next is to create the markup in the page.

TIP

Structured Query Language (SQL)

SQL is a language that's used in most relational databases for managing data in a database. It includes commands that let you retrieve data and update it, and that let you create, modify, and manage data in database tables. SQL is different than a programming language (like C#). With SQL, you tell the database what you want, and it's the database's job to figure out how to get the data or perform the task. Here are examples of some SQL commands and what they do:

```
Select * From Movies
```

```
SELECT ID, Name, Price FROM Product WHERE Price > 10.00 ORDER BY Name
```

The first `Select` statement gets all the columns (specified by `*`) from the *Movies* table.

The second `Select` statement fetches the ID, Name, and Price columns from records in the *Product* table whose Price column value is more than 10. The command returns the results in alphabetical order based on the values of the Name column. If no records match the price criteria, the command returns an empty set.

```
INSERT INTO Product (Name, Description, Price) VALUES ('Croissant', 'A flaky delight', 1.99)
```

This command inserts a new record into the *Product* table, setting the Name column to "Croissant", the Description column to "A flaky delight", and the price to 1.99.

Notice that when you're specifying a non-numeric value, the value is enclosed in single quotation marks (not double quotation marks, as in C#). You use these quotation marks around text or date values, but not around numbers.

```
DELETE FROM Product WHERE ExpirationDate < '01/01/2008'
```

This command deletes records in the *Product* table whose expiration date column is earlier than January 1, 2008. (The command assumes that the *Product* table has such a column, of course.) The date is entered here in MM/DD/YYYY format, but it should be entered in the format that's used for your locale.

The `Insert` and `Delete` commands don't return result sets. Instead, they return a number that tells you how many records were affected by the command.

For some of these operations (like inserting and deleting records), the process that's requesting the operation has to have appropriate permissions in the database. That's why for production databases you often have to supply a user name and password when you connect to the database.

There are dozens of SQL commands, but they all follow a pattern like the commands you see here. You can use SQL commands to create database tables, count the number of records in a table, calculate prices, and perform many more operations.

Adding Markup to Display the Data

Inside the `<head>` element, set contents of the `<title>` element to "Movies":

```
<head>
    <meta charset="utf-8" />
    <title>Movies</title>
</head>
```

Inside the `<body>` element of the page, add the following:

```
<h1>Movies</h1>
<div>
    @grid.GetHtml()
</div>
```

That's it. The `grid` variable is the value you created when you created the `WebGrid` object in code earlier.

In the WebMatrix tree view, right-click the page and select **Launch in browser**. You'll see something like this page:

Movies

ID	Title	Genre	Year
1	When Harry Met Sally	Romantic Comedy	1989
2	Gone With The Wind	Drama	1939
4	Ghostbusters	Comedy	1984
5	Fantasia	Children	1939
8	Clueless	Comedy	1995
9	Silverado	Western	1985
10	The Three Musketeers	Adventure	1973
11	Ronin	Action	1998

Click a column heading link to sort by that column. Being able to sort by clicking a heading is a feature that's built into the **WebGrid** helper.

The `GetHtml` method, as its name suggests, generates markup that displays the data. By default, the `GetHtml` method generates an HTML `<table>` element. (If you want, you can verify the rendering by looking at the source of the page in the browser.)

Modifying the Look of the Grid

Using the `WebGrid` helper like you just did is easy, but the resulting display is plain. The `WebGrid` helper has all sorts of options that let you control how the data is displayed. There are far too many to explore in this tutorial, but this section will give you an idea of some of those options. A few additional options will be covered in later tutorials in this series.

Specifying Individual Columns to Display

To start, you can specify that you want to display only certain columns. By default, as you've seen, the grid shows all four of the columns from the *Movies* table.

In the *Movies.cshtml* file, replace the `@grid.GetHtml()` markup that you just added with the following:

```
@grid.GetHtml(
    columns: grid.Columns(
        grid.Column("Title"),
        grid.Column("Genre"),
        grid.Column("Year")
    )
)
```

To tell the helper which columns to display, you include a `columns` parameter for the `GetHtml` method and pass in a collection of columns. In the collection, you specify each column to include. You specify an individual column to display by including a `grid.Column` object, and pass in the name of the data column you want. (These columns must be included in the SQL query results — the `WebGrid` helper cannot display columns that were not returned by the query.)

Launch the *Movies.cshtml* page in the browser again, and this time you get a display like the following one (notice that no ID column is displayed):

Movies

Title	Genre	Year
When Harry Met Sally	Romantic Comedy	1989
Gone With The Wind	Drama	1939
Ghostbusters	Comedy	1984
Fantasia	Children	1939
Clueless	Comedy	1995
Silverado	Western	1985
The Three Musketeers	Adventure	1973
Ronin	Action	1998

Changing the Look of the Grid

There are quite a few more options for displaying columns, some of which will be explored in later tutorials in this set. For now, this section will introduce you to ways in which you can style the grid as a whole.

Inside the `<head>` section of the page, just before the closing `</head>` tag, add the following `<style>` element:

```
<style type="text/css">
  .grid { margin: 4px; border-collapse: collapse; width: 600px; }
  .grid th, .grid td { border: 1px solid #C0C0C0; padding: 5px; }
  .head { background-color: #E8E8E8; font-weight: bold; color: #FFF; }
  .alt { background-color: #E8E8E8; color: #000; }
</style>
```

This CSS markup defines classes named `grid`, `head`, and so on. You could also put these style definitions in a separate *.css* file and link that to the page. (In fact, you'll do that later in this tutorial set.) But to make things easy for this tutorial, they're inside the same page that displays the data.

Now you can get the `WebGrid` helper to use these style classes. The helper has a number of properties (for example, `tableStyle`) for just this purpose — you assign a CSS style class name to them, and that class name is rendered as part of the markup that's rendered by the helper.

Change the `grid.GetHtml` markup so that it now looks like this code:

```
@grid.GetHtml(
    tableStyle: "grid",
    headerStyle: "head",
    alternatingRowStyle: "alt",
    columns: grid.Columns(
        grid.Column("Title"),
        grid.Column("Genre"),
        grid.Column("Year")
    )
)
```

What's new here is that you've added `tableStyle`, `headerStyle`, and `alternatingRowStyle` parameters to the `GetHtml` method. These parameters have been set to the names of the CSS styles that you added a moment ago.

Run the page, and this time you see a grid that looks much less plain than before:

Movies

Title	Genre	Year
When Harry Met Sally	Romantic Comedy	1989
Gone With The Wind	Drama	1939
Ghostbusters	Comedy	1984
Fantasia	Children	1939
Clueless	Comedy	1995
Silverado	Western	1985
The Three Musketeers	Adventure	1973
Ronin	Action	1998

To see what the `GetHtml` method generated, you can look at the source of the page in the browser. We won't go into detail here, but the important point is that by specifying parameters like `tableStyle`, you caused the grid to generate HTML tags like the following:

```
<table class="grid">
```

The `<table>` tag has had a `class` attribute added to it that references one of the CSS rules that you added earlier. This code shows you the basic pattern — different parameters for the `GetHtml` method let you reference CSS classes that the method then generates along with the markup. What you do with the CSS classes is up to you.

Adding Paging

As the last task for this tutorial, you'll add paging to the grid. Right now it's no problem to display all your movies at once. But if you added hundreds of movies, the page display would get long.

In the page code, change the line that creates the `WebGrid` object to the following code:

```
var grid = new WebGrid(source: selectedData, rowsPerPage: 3);
```

The only difference from before is that you've added a `rowsPerPage` parameter that's set to 3.

Run the page. The grid displays 3 rows at a time, plus navigation links that let you page through the movies in your database:

Movies

Title	Genre	Year
When Harry Met Sally	Romantic Comedy	1989
Gone With The Wind	Drama	1939
Ghostbusters	Comedy	1984
1 2 3 >		

Coming Up Next

In the next tutorial, you'll learn how to use Razor and C# code to get user input in a form. You'll add a search box to the Movies page so that you can find movies by title or genre.

Complete Listing for Movies Page

```
@{
    var db = Database.Open("WebPagesMovies");
    var selectedData = db.Query("SELECT * FROM Movies");
    var grid = new WebGrid(source: selectedData, rowsPerPage: 3);
}

<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Movies</title>
        <style type="text/css">
          .grid { margin: 4px; border-collapse: collapse; width: 600px; }
          .grid th, .grid td { border: 1px solid #C0C0C0; padding: 5px; }
          .head { background-color: #E8E8E8; font-weight: bold; color: #FFF; }
          .alt { background-color: #E8E8E8; color: #000; }
        </style>
    </head>
    <body>
        <h1>Movies</h1>
        <div>
            @grid.GetHtml(
                tableStyle: "grid",
                headerStyle: "head",
                alternatingRowStyle: "alt",
                columns: grid.Columns(
                    grid.Column("Title"),
                    grid.Column("Genre"),
                    grid.Column("Year")
                )
            )
        </div>
    </body>
</html>
```

Additional Resources

- Introduction to ASP.NET Web Programming Using the Razor Syntax

PREVIOUS NEXT

Introducing ASP.NET Web Pages - HTML Form Basics

2/20/2020 • 25 minutes to read • Edit Online

by Tom FitzMacken

This tutorial shows you the basics of how to create an input form and how to handle the user's input when you use ASP.NET Web Pages (Razor). And now that you've got a database, you'll use your form skills to let users find specific movies in the database. It assumes you have completed the series through Introduction to Displaying Data Using ASP.NET Web Pages.

What you'll learn:

- How to create a form by using standard HTML elements.
- How to read the user's input in a form.
- How to create a SQL query that selectively gets data by using a search term that the user supplies.
- How to have fields in the page "remember" what the user entered.

Features/technologies discussed:

- The `Request` object.
- The SQL `Where` clause.

What You'll Build

In the previous tutorial, you created a database, added data to it, and then used the `WebGrid` helper to display the data. In this tutorial, you'll add a search box that lets you find movies of a specific genre or whose title contains whatever word you enter. (For example, you'll be able to find all movies whose genre is "Action" or whose title contains "Harry" or "Adventure.")

When you're done with this tutorial, you'll have a page like this one:

The listing part of the page is the same as in the last tutorial — a grid. The difference will be that the grid will show only the movies that you searched for.

About HTML Forms

(If you've got experience with creating HTML forms and with the difference between `GET` and `POST`, you can skip this section.)

A form has user input elements — text boxes, buttons, radio buttons, check boxes, drop-down lists, and so on. Users fill in these controls or make selections and then submit the form by clicking a button.

The basic HTML syntax of a form is illustrated by this example:

```
<form method="post">
  <input type="text" name="name" value="" />
  <br/>
  <input type="submit" name="submit" value="Submit" />
</form>
```

When this markup runs in a page, it creates a simple form that looks like this illustration:

The `<form>` element encloses HTML elements to be submitted. (An easy mistake to make is to add elements to the page but then forget to put them inside a `<form>` element. In that case, nothing is submitted.) The `method` attribute tells the browser how to submit the user input. You set this to `post` if you're performing an update on the server or to `get` if you're just fetching data from the server.

TIP

GET, POST, and HTTP Verb Safety

HTTP, the protocol that browsers and servers use to exchange information, is remarkably simple in its basic operations. Browsers use only a few verbs to make requests to servers. When you write code for the web, it's helpful to understand these verbs and how the browser and server use them. Far and away the most commonly used verbs are these:

- `GET`. The browser uses this verb to fetch something from the server. For example, when you type a URL into your browser, the browser performs a `GET` operation to request the page you want. If the page includes graphics, the browser performs additional `GET` operations to get the images. If the `GET` operation has to pass information to the server, the information is passed as part of the URL in the query string.
- `POST`. The browser sends a `POST` request in order to submit data to be added or changed on the server. For example, the `POST` verb is used to create records in a database or change existing ones. Most of the time, when you fill in a form and click the submit button, the browser performs a `POST` operation. In a `POST` operation, the data being passed to the server is in the body of the page.

An important distinction between these verbs is that a `GET` operation is not supposed to change anything on the server — or to put it in a slightly more abstract way, a `GET` operation does not result in a change in state on the server. You can perform a `GET` operation on the same resources as many times as you like, and those resources don't change. (A `GET` operation is often said to be "safe," or to use a technical term, is *idempotent*.) In contrast, of course, a `POST` request changes something on the server each time you perform the operation.

Two examples will help illustrate this distinction. When you perform a search using an engine like Bing or Google, you fill in a form that consists of one text box, and then you click the search button. The browser performs a `GET` operation, with the value you entered into the box passed as part of the URL. Using a `GET` operation for this type of form is fine, because a search operation doesn't change any resources on the server, it just fetches information.

Now consider the process of ordering something online. You fill in the order details and then click the submit button. This operation will be a `POST` request, because the operation will result in changes on the server, such as a new order record, a change in your account information, and perhaps many other changes. Unlike the `GET` operation, you cannot repeat your `POST` request — if you did, each time you resubmitted the request, you'd generate a new order on the server. (In cases like this, websites will often warn you not to click a submit button more than once, or will disable the submit button so that you don't resubmit the form accidentally.)

In the course of this tutorial, you'll use both a `GET` operation and a `POST` operation to work with HTML forms. We'll explain in each case why the verb you use is the appropriate one.

(To learn more about HTTP verbs, see the Method Definitions article on the W3C site.)

Most user input elements are HTML `<input>` elements. They look like `<input type="type" name="name">,` where *type* indicates the kind of user input control you want. These elements are the common ones:

- Text box: `<input type="text">`
- Check box: `<input type="check">`
- Radio button: `<input type="radio">`
- Button: `<input type="button">`
- Submit button: `<input type="submit">`

You can also use the `<textarea>` element to create a multiline text box and the `<select>` element to create a drop-down list or scrollable list. (For more about HTML form elements, see HTML Forms and Input on the W3Schools site.)

The `name` attribute is very important, because the name is how you'll get the value of the element later, as you'll see shortly.

The interesting part is what you, the page developer, do with the user's input. There's no built-in behavior

associated with these elements. Instead, you have to get the values that the user has entered or selected and do something with them. That's what you'll learn in this tutorial.

> **TIP**
>
> **HTML5 and Input Forms**
>
> As you might know, HTML is in transition and the latest version (HTML5) includes support for more intuitive ways for users to enter information. For example, in HTML5, you (the page developer) can tell the page that you want the user to enter a date. The browser can then automatically display a calendar rather than requiring the user to enter a date manually. However, HTML5 is new and is not supported in all browsers yet.
>
> ASP.NET Web Pages supports HTML5 input to the extent that the user's browser does. For an idea of the new attributes for the `<input>` element in HTML5, see HTML <input> type Attribute on the W3Schools site.

Creating the Form

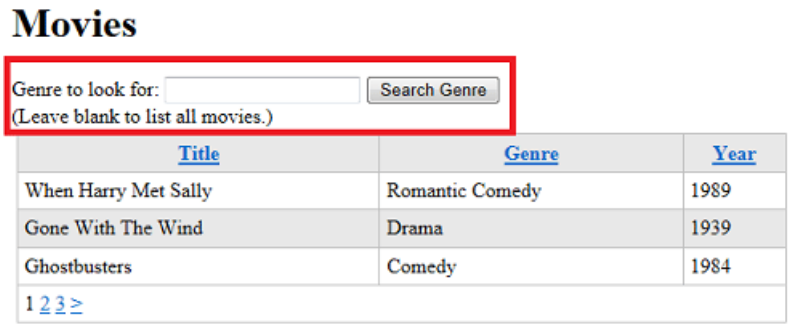
In WebMatrix, in the **Files** workspace, open the *Movies.cshtml* page.

After the closing `</h1>` tag and before the opening `<div>` tag of the `grid.GetHtml` call, add the following markup:

```
<form method="get">
  <div>
    <label for="searchGenre">Genre to look for:</label>
    <input type="text" name="searchGenre" value="" />
    <input type="Submit" value="Search Genre" /><br/>
    (Leave blank to list all movies.)<br/>
    </div>
</form>
```

This markup creates a form that has a text box named `searchGenre` and a submit button. The text box and submit button are enclosed in a `<form>` element whose `method` attribute is set to `get`. (Remember that if you don't put the text box and submit button inside a `<form>` element, nothing will be submitted when you click the button.) You use the `GET` verb here because you're creating a form that does not make any changes on the server — it just results in a search. (In the previous tutorial, you used a `post` method, which is how you submit changes to the server. You'll see that in the next tutorial again.)

Run the page. Although you haven't defined any behavior for the form, you can see what it looks like:



Enter a value into the text box, like "Comedy." Then click **Search Genre**.

Take note of the URL of the page. Because you set the `<form>` element's `method` attribute to `get`, the value you entered is now part of the query string in the URL, like this:

```
http://localhost:45661/Movies.cshtml?searchGenre=Comedy
```

Reading Form Values

The page already contains some code that gets database data and displays the results in a grid. Now you have to add some code that reads the value of the text box so you can run a SQL query that includes the search term.

Because you set the form's method to `get`, you can read the value that was entered into the text box by using code like the following:

```
var searchTerm = Request.QueryString["searchGenre"];
```

The `Request.QueryString` object (the `QueryString` property of the `Request` object) includes the values of elements that were submitted as part of the `GET` operation. The `Request.QueryString` property contains a *collection* (a list) of the values that are submitted in the form. To get any individual value, you specify the name of the element that you want. That's why you have to have a `name` attribute on the `<input>` element (`searchTerm`) that creates the text box. (For more about the `Request` object, see the sidebar later.)

It's simple enough to read the value of the text box. But if the user didn't enter anything at all in the text box but clicked **Search** anyway, you can ignore that click, since there's nothing to search.

The following code is an example that shows how to implement these conditions. (You don't have to add this code yet; you'll do that in a moment.)

```
if(!Request.QueryString["searchGenre"].IsEmpty() ) {
     // Do something here
}
```

The test breaks down in this way:

- Get the value of `Request.QueryString["searchGenre"]`, namely the value that was entered into the `<input>` element named `searchGenre`.
- Find out if it's empty by using the `IsEmpty` method. This method is the standard way to determine whether something (for example, a form element) contains a value. But really, you care only if it's *not* empty, therefore ...
- Add the `!` operator in front of the `IsEmpty` test. (The `!` operator means logical NOT).

In plain English, the entire `if` condition translates into the following: *If the form's searchGenre element is not empty, then ...*

This block sets the stage for creating a query that uses the search term. You'll do that in the next section.

TIP

The Request Object

The `Request` object contains all the information that the browser sends to your application when a page is requested or submitted. This object includes any information that the user provides, like text box values or a file to upload. It also includes all sorts of additional information, like cookies, values in the URL query string (if any), the file path of the page that is running, the type of browser that the user is using, the list of languages that are set in the browser, and much more.

The `Request` object is a *collection* (list) of values. You get an individual value out of the collection by specifying its name:

```
var someValue = Request["name"];
```

The `Request` object actually exposes several subsets. For example:

- `Request.Form` gives you values from elements inside the submitted `<form>` element if the request is a `POST` request.
- `Request.QueryString` gives you just the values in the URL's query string. (In a URL like `http://mysite/myapp/page?searchGenre=action&page=2`, the `?searchGenre=action&page=2` section of the URL is the query string.)
- `Request.Cookies` collection gives you access to cookies that the browser has sent.

To get a value that you know is in the submitted form, you can use `Request["name"]`. Alternatively, you can use the more specific versions `Request.Form["name"]` (for `POST` requests) or `Request.QueryString["name"]` (for `GET` requests). Of course, *name* is the name of the item to get.

The name of the item you want to get has to be unique within the collection you're using. That's why the `Request` object provides the subsets like `Request.Form` and `Request.QueryString`. Suppose that your page contains a form element named `userName` and *also* contains a cookie named `userName`. If you get `Request["userName"]`, it's ambiguous whether you want the form value or the cookie. However, if you get `Request.Form["userName"]` or `Request.Cookie["userName"]`, you're being explicit about which value to get.

It's a good practice to be specific and use the subset of `Request` that you're interested in, like `Request.Form` or `Request.QueryString`. For the simple pages that you're creating in this tutorial, it probably doesn't really make any difference. However, as you create more complex pages, using the explicit version `Request.Form` or `Request.QueryString` can help you avoid problems that can arise when the page contains a form (or multiple forms), cookies, query string values, and so on.

Creating a Query by Using a Search Term

Now that you know how to get the search term that the user entered, you can create a query that uses it. Remember that to get all the movie items out of the database, you're using a SQL query that looks like this statement:

```
SELECT * FROM Movies
```

To get only certain movies, you have to use a query that includes a `Where` clause. This clause lets you set a condition on which rows are returned by the query. Here's an example:

```
SELECT * FROM Movies WHERE Genre = 'Action'
```

The basic format is `WHERE column = value`. You can use different operators besides just `=`, like `>` (greater than), `<` (less than), `<>` (not equal to), `<=` (less than or equal to), etc., depending on what you're looking for.

In case you're wondering, SQL statements are not case sensitive — `SELECT` is the same as `Select` (or even `select`). However, people often capitalize keywords in a SQL statement, like `SELECT` and `WHERE`, to make it easier to read.

Passing the search term as a parameter

Searching for a specific genre is easy enough (`WHERE Genre = 'Action'`), but you want to be able to search for any genre that the user enters. To do that, you create as SQL query that includes a placeholder for the value to search. It will look like this command:

```
SELECT * FROM Movies WHERE Genre = @0
```

The placeholder is the `@` character followed by zero. As you might guess, a query can contain multiple placeholders, and they'd be named `@0`, `@1`, `@2`, etc.

To set up the query and actually pass it the value, you use the code like the following:

```
selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
selectedData = db.Query(selectCommand, Request.QueryString["searchGenre"]);
```

This code is similar to what you've already done to display data in the grid. The only differences are:

- The query contains a placeholder (`WHERE Genre = @0"`).
- The query is put into a variable (`selectCommand`); before, you passed the query directly to the `db.Query` method.
- When you call the `db.Query` method, you pass both the query and the value to use for the placeholder. (If the query had multiple placeholders, you'd pass them all as separate values to the method.)

If you put all these elements together, you get the following code:

```
if(!Request.QueryString["searchGenre"].IsEmpty() ) {
    searchTerm = Request.QueryString["searchGenre"];
    selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
    selectedData = db.Query(selectCommand, searchTerm);
}
```

NOTE

Important! Using placeholders (like `@0`) to pass values to a SQL command is *extremely important* for security. The way you see it here, with placeholders for variable data, is the only way you should construct SQL commands.

Never construct a SQL statement by putting together (concatenating) literal text and values you get from the user. Concatenating user input into a SQL statement opens your site to a *SQL injection attack* where a malicious user submits values to your page that hack your database. (You can read more in the article SQL Injection the MSDN website.)

Updating the Movies Page with Search Code

Now you can update the code in the *Movies.cshtml* file. To begin, replace the code in the code block at the top of the page with this code:

```
var db = Database.Open("WebPagesMovies");
var selectCommand = "SELECT * FROM Movies";
var searchTerm = "";
```

The difference here is that you've put the query into the `selectCommand` variable, which you'll pass to `db.Query` later. Putting the SQL statement into a variable lets you change the statement, which is what you'll do to perform the search.

You've also removed these two lines, which you'll put back in later:

```
var selectedData = db.Query("SELECT * FROM Movies");
var grid = new WebGrid(source: selectedData, rowsPerPage: 3);
```

You don't want to run the query yet (that is, call `db.Query`) and you don't want to initialize the `WebGrid` helper yet either. You'll do those things after you've figured out which SQL statement has to run.

After this rewritten block, you can add the new logic for handling the search. The completed code will look like the following. Update the code in your page so it matches this example:

```
@{
    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
        searchTerm = Request.QueryString["searchGenre"];
    }

    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre", rowsPerPage:3);
}
```

The page now works like this. Every time the page runs, the code opens the database and the `selectCommand` variable is set to the SQL statement that gets all the records from the `Movies` table. The code also initializes the `searchTerm` variable.

However, if the current request includes a value for the `searchGenre` element, the code sets `selectCommand` to a different query — namely, to one that includes the `Where` clause to search for a genre. It also sets `searchTerm` to whatever was passed for the search box (which might be nothing).

Regardless of which SQL statement is in `selectCommand`, the code then calls `db.Query` to run the query, passing it whatever is in `searchTerm`. If there's nothing in `searchTerm`, it doesn't matter, because in that case there's no parameter to pass the value to `selectCommand` anyway.

Finally, the code initializes the `WebGrid` helper by using the query results, just like before.

You can see that by putting the SQL statement and the search term into variables, you've added flexibility to the code. As you'll see later in this tutorial, you can use this basic framework and keep adding logic for different types of searches.

Testing the Search-by-Genre Feature

In WebMatrix, run the *Movies.cshtml* page. You see the page with the text box for genre.

Enter a genre that you've entered for one of your test records, then click **Search**. This time you see a listing of just the movies that match that genre:

Enter a different genre and search again. Try entering the genre by using all lowercase or all uppercase letters so

that you can see that the search is not case sensitive.

"Remembering" What the User Entered

You might have noticed that after you entered a genre and clicked **Search Genre**, you saw a listing for that genre. However, the search text box was empty — in other words, it didn't remember what you'd entered.

It's important to understand why this behavior occurs. When you submit a page, the browser sends a request to the web server. When ASP.NET gets the request, it creates a brand-new instance of the page, runs the code in it, and then renders the page to the browser again. In effect, though, the page doesn't know that you were just working with a previous version of itself. All it knows is that it got a request that had some form data in it.

Every time you request a page — whether for the first time or by submitting it — you're getting a new page. The web server has no memory of your last request. Neither does ASP.NET, and neither does the browser. The only connection between these separate instances of the page is any data that you transmit between them. If you submit a page, for example, the new page instance can get the form data that was sent by the earlier instance. (Another way to pass data between pages is to use cookies.)

A formal way to describe this situation is to say that web pages are *stateless*. Web servers and the pages themselves and the elements in the page do not maintain any information about the previous state of a page. The web was designed this way because maintaining state for individual requests would quickly exhaust the resources of web servers, which often handle thousands, maybe even hundreds of thousands, of requests per second.

So that's why the text box was empty. After you submitted the page, ASP.NET created a new instance of the page and ran through the code and markup. There was nothing in that code that told ASP.NET to put a value into the text box. So ASP.NET didn't do anything, and the text box was rendered without a value in it.

There's actually an easy way to get around this issue. The genre that you entered into the text box *is* available to you in code — it's in `Request.QueryString["searchGenre"]`.

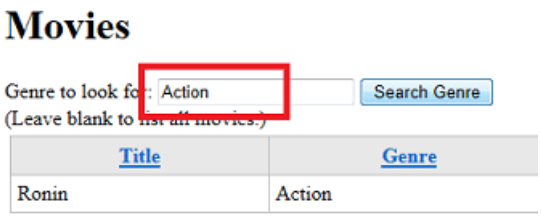
Update the markup for the text box so that the `value` attribute gets its value from `searchTerm`, like this example:

```
<input type="text" name="searchGenre" value="@Request.QueryString["searchGenre"]" />
```

In this page, you could have also set the `value` attribute to the `searchTerm` variable, since that variable also contains the genre you entered. But using the `Request` object to set the `value` attribute as shown here is the standard way to accomplish this task. (Assuming you even want to do this — in some situations, you might want to render the page *without* values in the fields. It all depends on what's going on with your app.)

> **NOTE**
> You can't "remember" the value of a text box that's used for passwords. It would be a security hole to allow people to fill in a password field by using code.

Run the page again, enter a genre, and click **Search Genre**. This time not only do you see the results of the search, but the text box remembers what you entered last time:



Searching for Any Word in the Title

You can now search for any genre, but you might also want to search for a title. It's hard to get a title exactly right when you search, so instead you can search for a word that appears anywhere inside a title. To do that in SQL, you use the `LIKE` operator and syntax like the following:

```
SELECT * FROM Movies WHERE Title LIKE '%adventure%'
```

This command gets all the movies whose titles contain "adventure". When you use the `LIKE` operator, you include the wildcard character `%` as part of the search term. The search `LIKE 'adventure%'` means "starting with 'adventure'". (Technically, it means "The string 'adventure' followed by anything.") Similarly, the search term `LIKE '%adventure'` means "anything followed by the string 'adventure'", which is another way to say "ending with 'adventure'".

The search term `LIKE '%adventure%'` therefore means "with 'adventure' anywhere in the title." (Technically, "anything in the title, followed by 'adventure', followed by anything.")

Inside the `<form>` element, add the following markup right under the closing `</div>` tag for the genre search (just before the closing `</form>` element):

```
<div>
  <label for="SearchTitle">Movie title contains the following:</label>
  <input type="text" name="searchTitle" value="@Request.QueryString["searchTitle"]" />
  <input type="Submit" value="Search Title" /><br/>
</div>
```

The code to handle this search is similar to the code for the genre search, except that you have to assemble the `LIKE` search. Inside the code block at the top of the page, add this `if` block just after the `if` block for the genre search:

```
if(!Request.QueryString["searchTitle"].IsEmpty() ) {
    selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
    searchTerm = "%" + Request["searchTitle"] + "%";
}
```

This code uses the same logic you saw earlier, except that the search uses a `LIKE` operator and the code puts "`%`" before and after the search term.

Notice how it was easy to add another search to the page. All you had to do was:

- Create an `if` block that tested to see whether the relevant search box had a value.
- Set the `selectCommand` variable to a new SQL statement.
- Set the `searchTerm` variable to the value to pass to the query.

Here's the complete code block, which contains the new logic for a title search:

```
@{
    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
        searchTerm = Request.QueryString["searchGenre"];
    }

   if(!Request.QueryString["searchTitle"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
        searchTerm = "%" + Request["searchTitle"] + "%";
    }

    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre", rowsPerPage:8);
}
```

Here's a summary of what this code does:

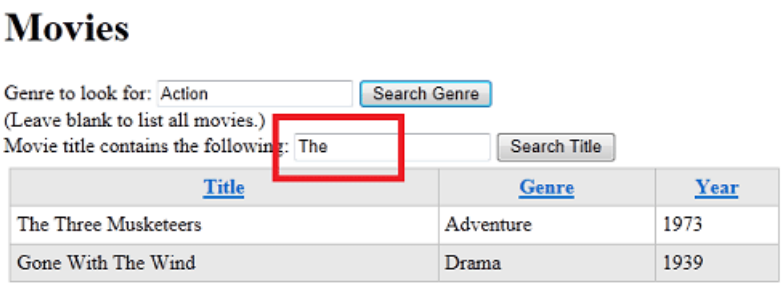
- The variables `searchTerm` and `selectCommand` are initialized at the top. You're going to set these variables to the appropriate search term (if any) and appropriate SQL command based on what the user does in the page. The default search is the simple case of getting all the movies from the database.
- In the tests for `searchGenre` and `searchTitle`, the code sets `searchTerm` to the value you want to search for. Those code blocks also set `selectCommand` to an appropriate SQL command for that search.
- The `db.Query` method is invoked only once, using whatever SQL command is in `selectedCommand` and whatever value is in `searchTerm`. If there is no search term (no genre and no title word), the value of `searchTerm` is an empty string. However, that doesn't matter, because in that case the query doesn't require a parameter.

Testing the Title Search Feature

Now you can test your completed search page. Run *Movies.cshtml*.

Enter a genre and click **Search Genre**. The grid displays movies of that genre, like before.

Enter a title word and click **Search Title**. The grid displays movies that have that word in the title.



Leave both text boxes blank and click either button. The grid displays all the movies.

Combining the Queries

You might notice that the searches you can perform are exclusive. You can't search the title and the genre at the same time, even if both search boxes have values in them. For example, you can't search for all action movies whose title contains "Adventure". (As the page is coded now, if you enter values for both genre and title, the title search gets precedence.) To create a search that combines the conditions, you would have to create a SQL query that has syntax like the following:

```
SELECT * FROM Movies WHERE Genre = @0 AND Title LIKE @1
```

And you'd have to run the query by using a statement like the following (roughly speaking):

```
var selectedData = db.Query(selectCommand, searchGenre, searchTitle);
```

Creating logic to allow many permutations of search criteria can get a bit involved, as you can see. Therefore, we'll stop here.

Coming Up Next

In the next tutorial, you'll create a page that uses a form to let users add movies to the database.

Complete Listing for Movie Page (Updated with Search)

```
@{
    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
        searchTerm = Request.QueryString["searchGenre"];
    }

    if(!Request.QueryString["searchTitle"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
        searchTerm = "%" + Request["searchTitle"] + "%";
    }

    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre", rowsPerPage:3);
}
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Movies</title>
    <style type="text/css">
      .grid { margin: 4px; border-collapse: collapse; width: 600px; }
      .grid th, .grid td { border: 1px solid #C0C0C0; padding: 5px; }
      .head { background-color: #E8E8E8; font-weight: bold; color: #FFF; }
      .alt { background-color: #E8E8E8; color: #000; }
    </style>
  </head>
  <body>
    <h1>Movies</h1>
      <form method="get">
        <div>
        <label for="searchGenre">Genre to look for:</label>
        <input type="text" name="searchGenre" value="@Request.QueryString["searchGenre"]" />
        <input type="Submit" value="Search Genre" /><br/>
        (Leave blank to list all movies.)<br/>
        </div>

        <div>
          <label for="SearchTitle">Movie title contains the following:</label>
          <input type="text" name="searchTitle" value="@Request.QueryString["searchTitle"]" />
          <input type="Submit" value="Search Title" /><br/>
        </div>
      </form>

    <div>
      @grid.GetHtml(
        tableStyle: "grid",
        headerStyle: "head",
        alternatingRowStyle: "alt",
        columns: grid.Columns(
          grid.Column("Title"),
          grid.Column("Genre"),
          grid.Column("Year")
        )
      )
    </div>
  </body>
</html>
```

Additional Resources

- Introduction to ASP.NET Web Programming Using the Razor Syntax

- SQL WHERE Clause on the W3Schools site
- Method Definitions article on the W3C site

PREVIOUS NEXT

Introducing ASP.NET Web Pages - Entering Database Data by Using Forms

2/20/2020 • 19 minutes to read • Edit Online

by Tom FitzMacken

This tutorial shows you how to create an entry form and then enter the data that you get from the form into a database table when you use ASP.NET Web Pages (Razor). It assumes you have completed the series through Basics of HTML Forms in ASP.NET Web Pages.

What you'll learn:

- More about how to process entry forms.
- How to add (insert) data in a database.
- How to make sure that users have entered a required value in a form (how to validate user input).
- How to display validation errors.
- How to jump to another page from the current page.

Features/technologies discussed:

- The `Database.Execute` method.
- The SQL `Insert Into` statement
- The `Validation` helper.
- The `Response.Redirect` method.

What You'll Build

In the tutorial earlier that showed you how to create a database, you entered database data by editing the database directly in WebMatrix, working in the **Database** workspace. In most apps, that's not a practical way to put data into the database, though. So in this tutorial, you'll create a web-based interface that lets you or anyone enter data and save it to the database.

You'll create a page where you can enter new movies. The page will contain an entry form that has fields (text boxes) where you can enter a movie title, genre, and year. The page will look like this page:

The text boxes will be HTML `<input>` elements that will look like this markup:

```
<input type="text" name="genre" value="" />
```

Creating the Basic Entry Form

Create a page named *AddMovie.cshtml*.

Replace what's in the file with the following markup. Overwrite everything; you'll add a code block at the top shortly.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Add a Movie</title>
</head>
<body>
  <h1>Add a Movie</h1>
  <form method="post">
    <fieldset>
      <legend>Movie Information</legend>
      <p><label for="title">Title:</label>
         <input type="text" name="title" value="@Request.Form["title"]" />
      </p>

      <p><label for="genre">Genre:</label>
         <input type="text" name="genre" value="@Request.Form["genre"]" />
      </p>

      <p><label for="year">Year:</label>
         <input type="text" name="year" value="@Request.Form["year"]" />
      </p>

      <p><input type="submit" name="buttonSubmit" value="Add Movie" /></p>
    </fieldset>
  </form>
</body>
</html>
```

This example shows typical HTML for creating a form. It uses `<input>` elements for the text boxes and for the submit button. The captions for the text boxes are created by using standard `<label>` elements. The `<fieldset>` and `<legend>` elements put a nice box around the form.

Notice that in this page, the `<form>` element uses `post` as the value for the `method` attribute. In the previous tutorial, you created a form that used the `get` method. That was correct, because although the form submitted values to the server, the request did not make any changes. All it did was fetch data in different ways. However, in this page you *will* make changes—you're going to add new database records. Therefore, this form should use the `post` method. (For more about the difference between `GET` and `POST` operations, see theGET, POST, and HTTP Verb Safety sidebar in the previous tutorial.)

Note that each text box has a `name` element (`title`, `genre`, `year`). As you saw in the previous tutorial, these names are important because you must have those names so you can get the user's input later. You can use any names. It's helpful to use meaningful names that help you remember what data you're working with.

The `value` attribute of each `<input>` element contains a bit of Razor code (for example, `Request.Form["title"]`). You learned a version of this trick in the previous tutorial to preserve the value entered into the text box (if any) after the form has been submitted.

Getting the Form Values

Next, you add code that processes the form. In outline, you'll do the following:

1. Check whether the page is being posted (was submitted). You want to your code to run only when users have clicked the button, not when the page first runs.
2. Get the values that the user entered into the text boxes. In this case, because the form is using the `POST` verb,

you get the form values from the `Request.Form` collection.

3. Insert the values as a new record in the *Movies* database table.

At the top of the file, add the following code:

```
@{
    var title = "";
    var genre = "";
    var year = "";

    if(IsPost){
        title = Request.Form["title"];
        genre = Request.Form["genre"];
        year = Request.Form["year"];
    }
}
```

The first few lines create variables (`title`, `genre`, and `year`) to hold the values from the text boxes. The line `if(IsPost)` makes sure that the variables are set *only* when users click the **Add Movie** button — that is, when the form has been posted.

As you saw in an earlier tutorial, you get the value of a text box by using an expression like `Request.Form["name"]`, where *name* is the name of the `<input>` element.

The names of the variables (`title`, `genre`, and `year`) are arbitrary. Like the names that you assign to `<input>` elements, you can call them anything you like. (The names of the variables don't have to match the name attributes of `<input>` elements on the form.) But as with the `<input>` elements, it's a good idea to use variable names that reflect the data that they contain. When you write code, consistent names make it easier for you to remember what data you're working with.

Adding Data to the Database

In the code block you just added, just *inside* the closing brace (`}`) of the `if` block (not just inside the code block), add the following code:

```
var db = Database.Open("WebPagesMovies");
var insertCommand = "INSERT INTO Movies (Title, Genre, Year) VALUES(@0, @1, @2)";
db.Execute(insertCommand, title, genre, year);
```

This example is similar to the code you used in a previous tutorial to fetch and display data. The line that starts with `db =` opens the database, like before, and the next line defines a SQL statement, again as you saw before. However, this time it defines a SQL `Insert Into` statement. The following example shows the general syntax of the `Insert Into` statement:

`INSERT INTO table (column1, column2, column3, ...) VALUES (value1, value2, value3, ...)`

In other words, you specify the table to insert into, then list the columns to insert into, and then list the values to insert. (As noted before, SQL is not case sensitive but some people capitalize the keywords to make it easier to read the command.)

The columns that you're inserting into are already listed in the command — `(Title, Genre, Year)`. The interesting part is how you get the values from the text boxes into the `VALUES` part of the command. Instead of actual values, you see `@0`, `@1`, and `@2`, which are of course placeholders. When you run the command (on the `db.Execute` line), you pass the values that you got from the text boxes.

Important! Remember that the only way you should ever include data entered online by a user in a SQL

statement is to use placeholders, as you see here (`VALUES(@0, @1, @2)`). If you concatenate user input into a SQL statement, you open yourself to a SQL injection attack, as explained in Form Basics in ASP.NET Web Pages (the previous tutorial).

Still inside the `if` block, add the following line after the `db.Execute` line:

```
Response.Redirect("~/Movies");
```

After the new movie has been inserted into the database, this line jumps you (redirects) to the *Movies* page so you can see the movie you just entered. The `~` operator means "root of the website." (The `~` operator works only in ASP.NET pages, not in HTML generally.)

The complete code block looks like this example:

```
@{
    var title = "";
    var genre = "";
    var year = "";

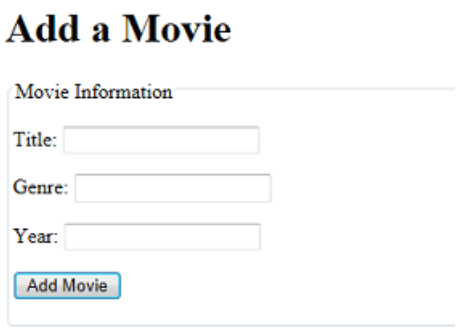
    if(IsPost){
        title = Request.Form["title"];
        genre = Request.Form["genre"];
        year = Request.Form["year"];

        var db = Database.Open("WebPagesMovies");
        var insertCommand = "INSERT INTO Movies (Title, Genre, Year) Values(@0, @1, @2)";
        db.Execute(insertCommand, title, genre, year);
        Response.Redirect("~/Movies");
    }
}
```

Testing the Insert Command (So Far)

You're not done yet, but now is a good time to test.

In the tree view of files in WebMatrix, right-click the *AddMovie.cshtml* page and then click **Launch in browser**.



(If you end up with a different page in the browser, make sure that the URL is `http://localhost:nnnnn/AddMovie`), where *nnnnn* is the port number that you're using.)

Did you get an error page? If so, read it carefully and make sure that the code looks exactly what was listed earlier.

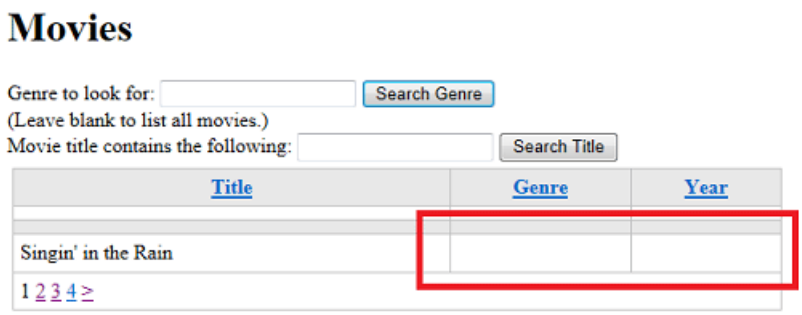
Enter a movie in the form — for example, use "Citizen Kane", "Drama", and "1941". (Or whatever.) Then click **Add Movie**.

If all goes well, you're redirected to the *Movies* page. Make sure that your new movie is listed.

Validating User Input

Go back to the *AddMovie* page, or run it again. Enter another movie, but this time, enter only the title — for example, enter "Singin' in the Rain". Then click **Add Movie**.

You're redirected to the *Movies* page again. You can find the new movie, but it's incomplete.



When you created the *Movies* table, you explicitly said that none of the fields could be null. Here you have an entry form for new movies, and you're leaving fields blank. That's an error.

In this case, the database didn't actually raise (or *throw*) an error. You didn't supply a genre or year, so the code in the *AddMovie* page treated those values as so-called *empty strings*. When the SQL `Insert Into` command ran, the genre and year fields didn't have useful data in them, but they weren't null.

Obviously, you don't want to let users enter half-empty movie information into the database. The solution is to validate the user's input. Initially, the validation will simply make sure that the user has entered a value for all of the fields (that is, that none of them contains an empty string).

TIP

Null and Empty Strings

In programming, there's a distinction between different notions of "no value." In general, a value is *null* if it has never been set or initialized in any way. In contrast, a variable that expects character data (strings) can be set to an *empty string*. In that case, the value is not null; it's just been explicitly set to a string of characters whose length is zero. These two statements show the difference:

```
var firstName;       // Not set, so its value is null
var firstName = "";  // Explicitly set to an empty string -- not null
```

It's a little more complicated than that, but the important point is that `null` represents a sort of undetermined state.

Now and then it's important to understand exactly when a value is null and when it's just an empty string. In the code for the *AddMovie* page, you get the values of the text boxes by using `Request.Form["title"]` and so on. When the page first runs (before you click the button), the value of `Request.Form["title"]` is null. But when you submit the form, `Request.Form["title"]` gets the value of the `title` text box. It's not obvious, but an empty text box is not null; it just has an empty string in it. So when the code runs in response to the button click, `Request.Form["title"]` has an empty string in it.

Why is this distinction important? When you created the *Movies* table, you explicitly said that none of the fields could be null. But here you have an entry form for new movies, and you're leaving fields blank. You would reasonably expect the database to complain when you tried to save new movies that didn't have values for genre or year. But that's the point — even if you leave those text boxes blank, the values aren't null; they're empty strings. As a result, you're able to save new movies to the database with these columns empty — but not null! — values. Therefore, you have to make sure that users don't submit an empty string, which you can do by validating the user's input.

The Validation Helper

ASP.NET Web Pages includes a helper — the `Validation` helper — that you can use to make sure that users enter data that meets your requirements. The `Validation` helper is one of the helpers that's built in to ASP.NET Web Pages, so you don't have to install it as a package by using NuGet, the way you installed the Gravatar helper in an earlier tutorial.

To validate the user's input, you'll do the following:

- Use code to specify that you want to require values in the text boxes on the page.
- Put a test into the code so that the movie information is added to the database only if everything validates properly.
- Add code into the markup to display error messages.

In the code block in the *AddMovie* page, right up at the top before the variable declarations, add the following code:

```
Validation.RequireField("title", "You must enter a title");
Validation.RequireField("genre", "Genre is required");
Validation.RequireField("year", "You haven't entered a year");
```

You call `Validation.RequireField` once for each field (`<input>` element) where you want to require an entry. You can also add a custom error message for each call, like you see here. (We varied the messages just to show that you can put anything you like there.)

If there's a problem, you want to prevent the new movie information from being inserted into the database. In the `if(IsPost)` block, use `&&` (logical AND) to add another condition that tests `Validation.IsValid()`. When you're done, the whole `if(IsPost)` block looks like this code:

```
if(IsPost && Validation.IsValid()){
    title = Request.Form["title"];
    genre = Request.Form["genre"];
    year = Request.Form["year"];

    var db = Database.Open("WebPagesMovies");
    var insertCommand = "INSERT INTO Movies (Title, Genre, Year) Values(@0, @1, @2)";
    db.Execute(insertCommand, title, genre, year);
    Response.Redirect("~/Movies");
}
```

If there's a validation error with any of the fields that you registered by using the `Validation` helper, the `Validation.IsValid` method returns false. And in that case, none of the code in that block will run, so no invalid movie entries will be inserted into the database. And of course you're not redirected to the *Movies* page.

The complete code block, including the validation code, now looks like this example:

```
@{
    Validation.RequireField("title", "You must enter a title");
    Validation.RequireField("genre", "Genre is required");
    Validation.RequireField("year", "You haven't entered a year");

    var title = "";
    var genre = "";
    var year = "";

    if(IsPost && Validation.IsValid()){
       title = Request.Form["title"];
       genre = Request.Form["genre"];
       year = Request.Form["year"];

       var db = Database.Open("WebPagesMovies");
       var insertCommand = "INSERT INTO Movies (Title, Genre, Year) Values(@0, @1, @2)";
       db.Execute(insertCommand, title, genre, year);
       Response.Redirect("~/Movies");
    }
}
```

Displaying Validation Errors

The last step is to display any error messages. You can display individual messages for each validation error, or you can display a summary, or both. For this tutorial, you'll do both so that you can see how it works.

Next to each `<input>` element that you're validating, call the `Html.ValidationMessage` method and pass it the name of the `<input>` element you're validating. You put the `Html.ValidationMessage` method right where you want the error message to appear. When the page runs, the `Html.ValidationMessage` method renders a `<span>` element where the validation error will go. (If there's no error, the `<span>` element is rendered, but there's no text in it.)

Change the markup in the page so that it includes an `Html.ValidationMessage` method for each of the three `<input>` elements on the page, like this example:

```
<p><label for="title">Title:</label>
     <input type="text" name="title" value="@Request.Form["title"]" />
      @Html.ValidationMessage("title")
  </p>

  <p><label for="genre">Genre:</label>
     <input type="text" name="genre" value="@Request.Form["genre"]" />
      @Html.ValidationMessage("genre")
  </p>

  <p><label for="year">Year:</label>
     <input type="text" name="year" value="@Request.Form["year"]" />
      @Html.ValidationMessage("year")
  </p>
```

To see how the summary works, also add the following markup and code right after the `<h1>Add a Movie</h1>` element on the page:

```
@Html.ValidationSummary()
```

By default, the `Html.ValidationSummary` method displays all the validation messages in a list (a `<ul>` element that's inside a `<div>` element). As with the `Html.ValidationMessage` method, the markup for the validation summary is always rendered; if there are no errors, no list items are rendered.

The summary can be an alternative way to display validation messages instead of by using the `Html.ValidationMessage` method to display each field-specific error. Or you can use both a summary and the details. Or you can use the `Html.ValidationSummary` method to display a generic error and then use individual `Html.ValidationMessage` calls to display details.

The complete page now looks like this example:

```
@{
    Validation.RequireField("title", "You must enter a title");
    Validation.RequireField("genre", "Genre is required");
    Validation.RequireField("year", "You haven't entered a year");

    var title = "";
    var genre = "";
    var year = "";

    if(IsPost && Validation.IsValid()){
       title = Request.Form["title"];
       genre = Request.Form["genre"];
       year = Request.Form["year"];

       var db = Database.Open("WebPagesMovies");
       var insertCommand = "INSERT INTO Movies (Title, Genre, Year) Values(@0, @1, @2)";
       db.Execute(insertCommand, title, genre, year);
       Response.Redirect("~/Movies");
    }
}

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Add a Movie</title>
</head>
<body>
  <h1>Add a Movie</h1>
  @Html.ValidationSummary()
  <form method="post">
    <fieldset>
      <legend>Movie Information</legend>
      <p><label for="title">Title:</label>
         <input type="text" name="title" value="@Request.Form["title"]" />
          @Html.ValidationMessage("title")
      </p>

      <p><label for="genre">Genre:</label>
         <input type="text" name="genre" value="@Request.Form["genre"]" />
          @Html.ValidationMessage("genre")
      </p>

      <p><label for="year">Year:</label>
         <input type="text" name="year" value="@Request.Form["year"]" />
          @Html.ValidationMessage("year")
      </p>

      <p><input type="submit" name="buttonSubmit" value="Add Movie" /></p>
    </fieldset>
  </form>
</body>
</html>
```

That's it. You can now test the page by adding a movie but leaving out one or more of the fields. When you do, you see the following error display:

Styling the Validation Error Messages

You can see that there are error messages, but they don't really stand out very well. There's an easy way to style the error messages, though.

To style the individual error messages that are displayed by `Html.ValidationMessage`, create a CSS style class named `field-validation-error`. To define the look for the validation summary, create a CSS style class named `validation-summary-errors`.

To see how this technique works, add a `<style>` element inside the `<head>` section of the page. Then define style classes named `field-validation-error` and `validation-summary-errors` that contain the following rules:

```
<head>
  <meta charset="utf-8" />
  <title>Add a Movie</title>
  <style type="text/css">
    .field-validation-error {
      font-weight:bold;
      color:red;
      background-color:yellow;
     }
    .validation-summary-errors{
      border:2px dashed red;
      color:red;
      background-color:yellow;
      font-weight:bold;
      margin:12px;
    }
  </style>
</head>
```

Normally you'd probably put style information into a separate *.css* file, but for simplicity you can put them in the page for now. (Later in this tutorial set, you'll move the CSS rules to a separate *.css* file.)

If there's a validation error, the `Html.ValidationMessage` method renders a `<span>` element that includes `class="field-validation-error"`. By adding a style definition for that class, you can configure what the message looks like. If there are errors, the `ValidationSummary` method likewise dynamically renders the attribute `class="validation-summary-errors"`.

Run the page again and deliberately leave out a couple of the fields. The errors are now more noticeable. (In fact, they're overdone, but that's just to show what you can do.)

Adding a Link to the Movies Page

One final step is to make it convenient to get to the *AddMovie* page from the original movie listing.

Open the *Movies* page again. After the closing `</div>` tag that follows the `WebGrid` helper, add the following markup:

```
<p>
  <a href="~/AddMovie">Add a movie</a>
</p>
```

As you saw before, ASP.NET interprets the `~` operator as the root of the website. You don't have to use the `~` operator; you could use the markup `<a href="./AddMovie">Add a movie</a>` or some other way to define the path that HTML understands. But the `~` operator is a good general approach when you create links for Razor pages, because it makes the site more flexible — if you move the current page to a subfolder, the link will still go to the *AddMovie* page. (Remember that the `~` operator only works in *.cshtml* pages. ASP.NET understands it, but it's not standard HTML.)

When you're done, run the *Movies* page. It will look like this page:

Click the **Add a movie** link to make sure that it goes to the *AddMovie* page.

Coming Up Next

In the next tutorial, you'll learn how to let users edit data that's already in the database.

Complete Listing for AddMovie Page

```
@{
    Validation.RequireField("title", "You must enter a title");
```

```
    Validation.RequireField("title", "You must enter a title");
    Validation.RequireField("genre", "Genre is required");
    Validation.RequireField("year", "You haven't entered a year");

    var title = "";
    var genre = "";
    var year = "";

    if(IsPost && Validation.IsValid()){
       title = Request.Form["title"];
       genre = Request.Form["genre"];
       year = Request.Form["year"];

       var db = Database.Open("WebPagesMovies");
       var insertCommand = "INSERT INTO Movies (Title, Genre, Year) Values(@0, @1, @2)";
       db.Execute(insertCommand, title, genre, year);
       Response.Redirect("~/Movies");
    }
}

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Add a Movie</title>
      <style type="text/css">
    .field-validation-error {
      font-weight:bold;
      color:red;
      background-color:yellow;
     }
    .validation-summary-errors{
      border:2px dashed red;
      color:red;
      background-color:yellow;
      font-weight:bold;
      margin:12px;
    }
  </style>
</head>
<body>
  <h1>Add a Movie</h1>
  @Html.ValidationSummary()
  <form method="post">
    <fieldset>
      <legend>Movie Information</legend>
      <p><label for="title">Title:</label>
         <input type="text" name="title" value="@Request.Form["title"]" />
          @Html.ValidationMessage("title")
      </p>

      <p><label for="genre">Genre:</label>
         <input type="text" name="genre" value="@Request.Form["genre"]" />
         @Html.ValidationMessage("genre")
      </p>

      <p><label for="year">Year:</label>
         <input type="text" name="year" value="@Request.Form["year"]" />
          @Html.ValidationMessage("year")
      </p>

      <p><input type="submit" name="buttonSubmit" value="Add Movie" /></p>
    </fieldset>
  </form>
</body>
</html>
```

Additional Resources

- Introduction to ASP.NET Web Programming Using the Razor Syntax
- SQL INSERT INTO Statement on the W3Schools site
- Validating User Input in ASP.NET Web Pages Sites. More information about working with the `Validation` helper.

PREVIOUS NEXT

Introducing ASP.NET Web Pages - Updating Database Data

2/20/2020 • 21 minutes to read • Edit Online

by Tom FitzMacken

This tutorial shows you how to update (change) an existing database entry when you use ASP.NET Web Pages (Razor). It assumes you have completed the series through Entering Data by Using Forms Using ASP.NET Web Pages.

What you'll learn:

- How to select an individual record in the `WebGrid` helper.
- How to read a single record from a database.
- How to preload a form with values from the database record.
- How to update an existing record in a database.
- How to store information in the page without displaying it.
- How to use a hidden field to store information.

Features/technologies discussed:

- The `WebGrid` helper.
- The SQL `Update` command.
- The `Database.Execute` method.
- Hidden fields (`<input type="hidden">`).

What You'll Build

In the previous tutorial, you learned how to add a record to a database. Here, you'll learn how to display a record for editing. In the *Movies* page, you'll update the `WebGrid` helper so that it displays an **Edit** link next to each movie:

When you click the **Edit** link, it takes you to a different page, where the movie information is already in a form:

You can change any of the values. When you submit the changes, the code in the page updates the database and takes you back to the movie listing.

This part of the process works almost exactly like the *AddMovie.cshtml* page you created in the previous tutorial, so much of this tutorial will be familiar.

There are several ways you could implement a way to edit an individual movie. The approach shown was chosen because it's easy to implement and easy to understand.

Adding an Edit Link to the Movie Listing

To begin, you'll update the *Movies* page so that each movie listing also contains an **Edit** link.

Open the *Movies.cshtml* file.

In the body of the page, change the `WebGrid` markup by adding a column. Here's the modified markup:

```
@grid.GetHtml(
    tableStyle: "grid",
    headerStyle: "head",
    alternatingRowStyle: "alt",
    columns: grid.Columns(
        grid.Column(format: @<a href="~/EditMovie?id=@item.ID">Edit</a>),
        grid.Column("Title"),
        grid.Column("Genre"),
        grid.Column("Year")
    )
)
```

The new column is this one:

```
grid.Column(format: @<a href="~/EditMovie?id=@item.ID)">Edit</a>)
```

The point of this column is to show a link (`<a>` element) whose text says "Edit". What we're after is to create a link that looks like the following when the page runs, with the `id` value different for each movie:

```
http://localhost:43097/EditMovie?id=7
```

This link will invoke a page named *EditMovie*, and it will pass the query string `?id=7` to that page.

The syntax for the new column might look a bit complex, but that's only because it puts together several elements. Each individual element is straightforward. If you concentrate on just the `<a>` element, you see this markup:

```
<a href="~/EditMovie?id=@item.ID)">Edit</a>
```

Some background about how the grid works: the grid displays rows, one for each database record, and it displays

columns for each field in the database record. While each grid row is being constructed, the `item` object contains the database record (item) for that row. This arrangement gives you a way in code to get at the data for that row. That's what you see here: the expression `item.ID` is getting the ID value of the current database item. You could get any of the database values (title, genre, or year) the same way by using `item.Title`, `item.Genre`, or `item.Year`.

The expression `"~/EditMovie?id=@item.ID` combines the hard-coded part of the target URL (`~/EditMovie?id=`) with this dynamically derived ID. (You saw the `~` operator in the previous tutorial; it's an ASP.NET operator that represents the current website root.)

The result is that this part of the markup in the column simply produces something like the following markup at run time:

```
href="/EditMovie?id=2"
```

Naturally, the actual value of `id` will be different for each row.

Creating a Custom Display for a Grid Column

Now back to the grid column. The three columns you originally had in the grid displayed only data values (title, genre, and year). You specified this display by passing the name of the database column — for example, `grid.Column("Title")`.

This new **Edit** link column is different. Instead of specifying a column name, you're passing a `format` parameter. This parameter lets you define markup that the `WebGrid` helper will render along with the `item` value to display the column data as bold or green or in whatever format that you want. For example, if you wanted the title to appear bold, you could create a column like this example:

```
grid.Column(format:@<strong>@item.Title</strong>)
```

(The various `@` characters you see in the `format` property mark the transition between markup and a code value.)

Once you know about the `format` property, it's easier to understand how the new **Edit** link column is put together:

```
grid.Column(format: @<a href="~/EditMovie?id=@item.ID">Edit</a>),
```

The column consists *only* of the markup that renders the link, plus some information (the ID) that's extracted from the database record for the row.

TIP

Named Parameters and Positional Parameters for a Method

Many times when you've called a method and passed parameters to it, you've simply listed the parameter values separated by commas. Here are a couple of examples:

`db.Execute(insertCommand, title, genre, year)`

`Validation.RequireField("title", "You must enter a title")`

We didn't mention the issue when you first saw this code, but in each case, you're passing parameters to the methods in a specific order — namely, the order in which the parameters are defined in that method. For `db.Execute` and `Validation.RequireFields`, if you mixed up the order of the values you pass, you'd get an error message when the page runs, or at least some strange results. Clearly, you have to know the order to pass the parameters in. (In WebMatrix, IntelliSense can help you learn figure out the name, type, and order of the parameters.)

As an alternative to passing values in order, you can use *named parameters*. (Passing parameters in order is known as using *positional parameters*.) For named parameters, you explicitly include the name of the parameter when passing its value. You've used named parameters already a number of times in these tutorials. For example:

```
var grid = new WebGrid(source: selectedData, defaultSort: "Genre", rowsPerPage:3)
```

and

```
@grid.GetHtml(
    tableStyle: "grid",
    headerStyle: "head",
    alternatingRowStyle: "alt",
    columns: grid.Columns(
       grid.Column("Title"),
       grid.Column("Genre"),
       grid.Column("Year")
    )
)
```

Named parameters are handy for a couple of situations, especially when a method takes many parameters. One is when you want to pass only one or two parameters, but the values you want to pass are not among the first positions in the parameter list. Another situation is when you want to make your code more readable by passing the parameters in the order that makes the most sense to you.

Obviously, to use named parameters, you have to know the names of the parameters. WebMatrix IntelliSense can *show* you the names, but it cannot currently fill them in for you.

Creating the Edit Page

Now you can create the *EditMovie* page. When users click the **Edit** link, they'll end up on this page.

Create a page named *EditMovie.cshtml* and replace what's in the file with the following markup:

```
<!DOCTYPE html>
<html>
  <head>
   <meta charset="utf-8" />
   <title>Edit a Movie</title>
    <style>
      .validation-summary-errors{
        border:2px dashed red;
        color:red;
        font-weight:bold;
        margin:12px;
      }
    </style>
  </head>
  <body>
    <h1>Edit a Movie</h1>
    @Html.ValidationSummary()
    <form method="post">
      <fieldset>
        <legend>Movie Information</legend>

        <p><label for="title">Title:</label>
           <input type="text" name="title" value="@title" /></p>

        <p><label for="genre">Genre:</label>
           <input type="text" name="genre" value="@genre" /></p>

        <p><label for="year">Year:</label>
           <input type="text" name="year" value="@year" /></p>

        <input type="hidden" name="movieid" value="@movieId" />

        <p><input type="submit" name="buttonSubmit" value="Submit Changes" /></p>
      </fieldset>
    </form>
  </body>
</html>
```

This markup and code is similar to what you have in the *AddMovie* page. There's a small difference in the text for the submit button. As with the *AddMovie* page, there's an `Html.ValidationSummary` call that will display validation errors if there are any. This time we're leaving out calls to `Validation.Message`, since errors will be displayed in the validation summary. As noted in the previous tutorial, you can use the validation summary and the individual error messages in various combinations.

Notice again that the `method` attribute of the `<form>` element is set to `post`. As with the *AddMovie.cshtml* page, this page makes changes to the database. Therefore, this form should perform a `POST` operation. (For more about the difference between `GET` and `POST` operations, see the GET, POST, and HTTP Verb Safety sidebar in the tutorial on HTML forms.)

As you saw in an earlier tutorial, the `value` attributes of the text boxes are being set with Razor code in order to preload them. This time, though, you're using variables like `title` and `genre` for that task instead of `Request.Form["title"]`:

```
<input type="text" name="title" value="@title" />
```

As before, this markup will preload the text box values with the movie values. You'll see in a moment why it's handy to use variables this time instead of using the `Request` object.

There's also a `<input type="hidden">` element on this page. This element stores the movie ID without making it visible on the page. The ID is initially passed to the page by using a query string value (`?id=7` or similar in the URL). By putting the ID value into a hidden field, you can make sure that it's available when the form is submitted,

even if you no longer have access to the original URL that the page was invoked with.

Unlike the *AddMovie* page, the code for the *EditMovie* page has two distinct functions. The first function is that when the page is displayed for the first time (and *only* then), the code gets the movie ID from the query string. The code then uses the ID to read the corresponding movie out of the database and display (preload) it in the text boxes.

The second function is that when the user clicks the **Submit Changes** button, the code has to read the values of the text boxes and validate them. The code also has to update the database item with the new values. This technique is similar to adding a record, as you saw in *AddMovie*.

Adding Code to Read a Single Movie

To perform the first function, add this code to the top of the page:

```
@{
    var title = "";
    var genre = "";
    var year = "";
    var movieId = "";

    if(!IsPost){
        if(!Request.QueryString["ID"].IsEmpty()){
            movieId = Request.QueryString["ID"];
            var db = Database.Open("WebPagesMovies");
            var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
            var row = db.QuerySingle(dbCommand, movieId);
            title = row.Title;
            genre = row.Genre;
            year = row.Year;
        }
        else{
            Validation.AddFormError("No movie was selected.");
        }
    }
}
```

Most of this code is inside a block that starts `if(!IsPost)`. The `!` operator means "not," so the expression means *if this request is not a post submission*, which is an indirect way of saying *if this request is the first time that this page has been run*. As noted earlier, this code should run *only* the first time the page runs. If you didn't enclose the code in `if(!IsPost)`, it would run every time the page is invoked, whether the first time or in response to a button click.

Notice that the code includes an `else` block this time. As we said when we introduced `if` blocks, sometimes you want to run alternative code if the condition you're testing isn't true. That's the case here. If the condition passes (that is, if the ID passed to the page is ok), you read a row from the database. However, if the condition doesn't pass, the `else` block runs and the code sets an error message.

Validating a Value Passed to the Page

The code uses `Request.QueryString["id"]` to get the ID that's passed to the page. The code makes sure that a value was actually passed for the ID. If no value was passed, the code sets a validation error.

This code shows a different way to validate information. In the previous tutorial, you worked with the `Validation` helper. You registered fields to validate, and ASP.NET automatically did the validation and displayed errors by using `Html.ValidationMessage` and `Html.ValidationSummary`. In this case, however, you're not really validating user input. Instead, you're validating a value that was passed to the page from elsewhere. The `Validation` helper doesn't do that for you.

Therefore, you check the value yourself, by testing it with `if(!Request.QueryString["ID"].IsEmpty()`). If there's a problem, you can display the error by using `Html.ValidationSummary`, as you did with the `Validation` helper. To do that, you call `Validation.AddFormError` and pass it a message to display. `Validation.AddFormError` is a built-in method that lets you define custom messages that tie in with the validation system you're already familiar with. (Later in this tutorial we'll talk about how to make this validation process a little more robust.)

After making sure that there's an ID for the movie, the code reads the database, looking for only a single database item. (You probably have noticed the general pattern for database operations: open the database, define a SQL statement, and run the statement.) This time, the SQL `Select` statement includes `WHERE ID = @0`. Because the ID is unique, only one record can be returned.

The query is performed by using `db.QuerySingle` (not `db.Query`, as you used for the movie listing), and the code puts the result into the `row` variable. The name `row` is arbitrary; you can name the variables anything you like. The variables initialized at the top are then filled with the movie details so that these values can be displayed in the text boxes.

Testing the Edit Page (So Far)

If you'd like to test your page, run the *Movies* page now and click an **Edit** link next to any movie. You'll see the *EditMovie* page with the details filled in for the movie you selected:

Notice that the URL of the page includes something like `?id=10` (or some other number). So far you've tested that **Edit** links in the *Movie* page work, that your page is reading the ID from the query string, and that the database query to get a single movie record is working.

You can change the movie information, but nothing happens when you click **Submit Changes**.

Adding Code to Update the Movie with the User's Changes

In the *EditMovie.cshtml* file, to implement the second function (saving changes), add the following code just inside the closing brace of the `@` block. (If you're not sure exactly where to put the code, you can look at the complete code listing for the Edit Movie page that appears at the end of this tutorial.)

```
if(IsPost){
    Validation.RequireField("title", "You must enter a title");
    Validation.RequireField("genre", "Genre is required");
    Validation.RequireField("year", "You haven't entered a year");
    Validation.RequireField("movieid", "No movie ID was submitted!");

    title = Request.Form["title"];
    genre = Request.Form["genre"];
    year = Request.Form["year"];
    movieId = Request.Form["movieId"];

    if(Validation.IsValid()){
        var db = Database.Open("WebPagesMovies");
        var updateCommand = "UPDATE Movies SET Title=@0, Genre=@1, Year=@2 WHERE Id=@3";
        db.Execute(updateCommand, title, genre, year, movieId);
        Response.Redirect("~/Movies");
   }
}
```

Again, this markup and code is similar to the code in *AddMovie*. The code is in an `if(IsPost)` block, because this code runs only when the user clicks the **Submit Changes** button — that is, when (and only when) the form has been posted. In this case, you're not using a test like `if(IsPost && Validation.IsValid())` — that is, you're not combining both tests by using AND. In this page, you first determine whether there's a form submission (`if(IsPost)`), and only then register the fields for validation. Then you can test the validation results (`if(Validation.IsValid()`). The flow is slightly different than in the *AddMovie.cshtml* page, but the effect is the same.

You get the values of the text boxes by using `Request.Form["title"]` and similar code for the other `<input>` elements. Notice that this time, the code gets the movie ID out of the hidden field (`<input type="hidden">`). When the page ran the first time, the code got the ID out of the query string. You get the value from the hidden field to make sure that you're getting the ID of the movie that was originally displayed, in case the query string was somehow altered since then.

The really important difference between the *AddMovie* code and this code is that in this code you use the SQL `Update` statement instead of the `Insert Into` statement. The following example shows the syntax of the SQL `Update` statement:

```
UPDATE table SET col1="value", col2="value", col3="value" ... WHERE ID = value
```

You can specify any columns in any order, and you don't necessarily have to update every column during an `Update` operation. (You cannot update the ID itself, because that would in effect save the record as a new record, and that's not allowed for an `Update` operation.)

> **NOTE**
>
> **Important** The `Where` clause with the ID is very important, because that's how the database knows which database record you want to update. If you left off the `Where` clause, the database would update *every* record in the database. In most cases, that would be a disaster.

In the code, the values to update are passed to the SQL statement by using placeholders. To repeat what we've said before: for security reasons, *only* use placeholders to pass values to a SQL statement.

After the code uses `db.Execute` to run the `Update` statement, it redirects back to the listing page, where you can see the changes.

TIP

Different SQL Statements, Different Methods

You might have noticed that you use slightly different methods to run different SQL statements. To run a `Select` query that potentially returns multiple records, you use the `Query` method. To run a `Select` query that you know will return only one database item, you use the `QuerySingle` method. To run commands that make changes but that don't return database items, you use the `Execute` method.

You have to have different methods because each of them returns different results, as you saw already in the difference between `Query` and `QuerySingle`. (The `Execute` method actually returns a value also — namely, the number of database rows that were affected by the command — but you've been ignoring that so far.)

Of course, the `Query` method might return only one database row. However, ASP.NET always treats the results of the `Query` method as a collection. Even if the method returns just one row, you have to extract that single row from the collection. Therefore, in situations where you *know* you'll get back only one row, it's a bit more convenient to use `QuerySingle`.

There are a few other methods that perform specific types of database operations. You can find a listing of database methods in the ASP.NET Web Pages API Quick Reference.

Making Validation for the ID More Robust

The first time that the page runs, you get the movie ID from the query string so that you can go get that movie from the database. You made sure that there actually was a value to go look for, which you did by using this code:

```
if(!IsPost){
    if(!Request.QueryString["ID"].IsEmpty()){
        // Etc.
    }
}
```

You used this code to make sure that if a user gets to the *EditMovies* page without first selecting a movie in the *Movies* page, the page would display a user-friendly error message. (Otherwise, users would see an error that would probably just confuse them.)

However, this validation isn't very robust. The page might also be invoked with these errors:

- The ID isn't a number. For example, the page could be invoked with a URL like `http://localhost:nnnnn/EditMovie?id=abc`.
- The ID is a number, but it references a movie that doesn't exist (for example, `http://localhost:nnnnn/EditMovie?id=100934`).

If you're curious to see the errors that result from these URLs, run the *Movies* page. Select a movie to edit, and then change the URL of the *EditMovie* page to a URL that contains an alphabetic ID or the ID of a non-existent movie.

So what should you do? The first fix is to make sure that not only is an ID passed to the page, but that the ID is an integer. Change the code for the `!IsPost` test to look like this example:

```
if(!IsPost){
    if(!Request.QueryString["ID"].IsEmpty() && Request.QueryString["ID"].IsInt()) {
       // Etc.
```

You've added a second condition to the `IsEmpty` test, linked with `&&` (logical AND):

```
Request.QueryString["ID"].IsInt()
```

You might remember from the Introduction to ASP.NET Web Pages Programming tutorial that methods like `AsBool` an `AsInt` convert a character string to some other data type. The `IsInt` method (and others, like `IsBool` and `IsDateTime`) are similar. However, they test only whether you *can* convert the string, without actually performing the conversion. So here you're essentially saying *If the query string value can be converted to an integer ....*

The other potential problem is looking for a movie that doesn't exist. The code to get a movie looks like this code:

```
var row = db.QuerySingle(dbCommand, movieId);
```

If you pass a `movieId` value to the `QuerySingle` method that doesn't correspond to an actual movie, nothing is returned and the statements that follow (for example, `title=row.Title`) result in errors.

Again there's an easy fix. If the `db.QuerySingle` method returns no results, the `row` variable will be null. So you can check whether the `row` variable is null before you try to get values from it. The following code adds an `if` block around the statements that get the values out of the `row` object:

```
if(row != null) {
    title = row.Title;
    genre = row.Genre;
    year = row.Year;
}
else{
    Validation.AddFormError("No movie was found for that ID.");
}
```

With these two additional validation tests, the page becomes more bullet-proof. The complete code for the `!IsPost` branch now looks like this example:

```
if(!IsPost){
    if(!Request.QueryString["ID"].IsEmpty() && Request.QueryString["ID"].IsInt()) {
        movieId = Request.QueryString["ID"];
        var db = Database.Open("WebPagesMovies");
        var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
        var row = db.QuerySingle(dbCommand, movieId);

        if(row != null) {
            title = row.Title;
            genre = row.Genre;
            year = row.Year;
        }
        else {
            Validation.AddFormError("No movie was found for that ID.");
        }
    }
    else {
        Validation.AddFormError("No movie was selected.");
    }
}
```

We'll note once more that this task is a good use for an `else` block. If the tests don't pass, the `else` blocks set error messages.

Adding a Link to Return to the Movies Page

A final and helpful detail is to add a link back to the *Movies* page. In the ordinary flow of events, users will start at the *Movies* page and click an **Edit** link. That brings them to the *EditMovie* page, where they can edit the movie and click the button. After the code processes the change, it redirects back to the *Movies* page.

However:

- The user might decide not to change anything.
- The user might have gotten to this page without first clicking an **Edit** link in the *Movies* page.

Either way, you want to make it easy for them to return to the main listing. It's an easy fix — add the following markup just after the closing `</form>` tag in the markup:

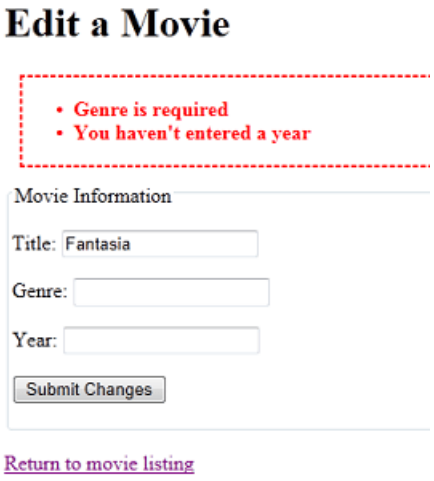
```
<p><a href="~/Movies">Return to movie listing</a></p>
```

This markup uses the same syntax for an `<a>` element that you've seen elsewhere. The URL includes `~` to mean "root of the website."

Testing the Movie Update Process

Now you can test. Run the *Movies* page, and click **Edit** next to a movie. When the *EditMovie* page appears, make changes to the movie and click **Submit Changes**. When the movie listing appears, make sure that your changes are shown.

To make sure that validation is working, click **Edit** for another movie. When you get to the *EditMovie* page, clear the **Genre** field (or **Year** field, or both) and try to submit your changes. You'll see an error, as you'd expect:



Click the **Return to movie listing** link to abandon your changes and return to the *Movies* page.

Coming Up Next

In the next tutorial, you'll see how to delete a movie record.

Complete Listing for Movie Page (Updated with Edit Links)

```
@{
    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
        searchTerm = Request.QueryString["searchGenre"];
    }

    if(!Request.QueryString["searchTitle"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
        searchTerm = "%" + Request.QueryString["searchTitle"] + "%";
    }

    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre", rowsPerPage:3);
}

<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Movies</title>
        <style type="text/css">
          .grid { margin: 4px; border-collapse: collapse; width: 600px; }
          .grid th, .grid td { border: 1px solid #C0C0C0; padding: 5px; }
          .head { background-color: #E8E8E8; font-weight: bold; color: #FFF; }
          .alt { background-color: #E8E8E8; color: #000; }
        </style>
    </head>
    <body>
        <h1>Movies</h1>
          <form method="get">
              <div>
                <label for="searchGenre">Genre to look for:</label>
                <input type="text" name="searchGenre" value="@Request.QueryString["searchGenre"]" />
                <input type="Submit" value="Search Genre" /><br/>
                (Leave blank to list all movies.)<br/>
                </div>

              <div>
                  <label for="SearchTitle">Movie title contains the following:</label>
                  <input type="text" name="searchTitle" value="@Request.QueryString["searchTitle"]" />
                  <input type="Submit" value="Search Title" /><br/>
                </div>
            </form>

        <div>
             @grid.GetHtml(
                tableStyle: "grid",
                headerStyle: "head",
                alternatingRowStyle: "alt",
                columns: grid.Columns(
                    grid.Column(format: @<a href="~/EditMovie?id=@item.ID">Edit</a>),
                    grid.Column("Title"),
                    grid.Column("Genre"),
                    grid.Column("Year")
                )
            )
        </div>
    <p>
        <a href="~/AddMovie">Add a movie</a>
    </p>
    </body>
</html>
```

Complete Page Listing for Edit Movie Page

```
@{
    var title = "";
    var genre = "";
    var year = "";
    var movieId = "";

    if(!IsPost){
        if(!Request.QueryString["ID"].IsEmpty() && Request.QueryString["ID"].IsInt()) {
            movieId = Request.QueryString["ID"];
            var db = Database.Open("WebPagesMovies");
            var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
            var row = db.QuerySingle(dbCommand, movieId);

            if(row != null) {
                title = row.Title;
                genre = row.Genre;
                year = row.Year;
            }
            else{
                Validation.AddFormError("No movie was selected.");
            }
        }
        else{
            Validation.AddFormError("No movie was selected.");
        }
    }

    if(IsPost){
        Validation.RequireField("title", "You must enter a title");
        Validation.RequireField("genre", "Genre is required");
        Validation.RequireField("year", "You haven't entered a year");
        Validation.RequireField("movieid", "No movie ID was submitted!");

        title = Request.Form["title"];
        genre = Request.Form["genre"];
        year = Request.Form["year"];
        movieId = Request.Form["movieId"];

        if(Validation.IsValid()){
            var db = Database.Open("WebPagesMovies");
            var updateCommand = "UPDATE Movies SET Title=@0, Genre=@1, Year=@2 WHERE Id=@3";
            db.Execute(updateCommand, title, genre, year, movieId);
            Response.Redirect("~/Movies");
        }
    }
}

<!DOCTYPE html>
<html>
  <head>
   <meta charset="utf-8" />
   <title>Edit a Movie</title>
    <style>
      .validation-summary-errors{
        border:2px dashed red;
        color:red;
        font-weight:bold;
        margin:12px;
      }
    </style>
  </head>
  <body>
    <h1>Edit a Movie</h1>
      @Html.ValidationSummary()
      <form method="post">
```

```
      <fieldset>
        <legend>Movie Information</legend>

        <p><label for="title">Title:</label>
           <input type="text" name="title" value="@title" /></p>

        <p><label for="genre">Genre:</label>
           <input type="text" name="genre" value="@genre" /></p>

        <p><label for="year">Year:</label>
           <input type="text" name="year" value="@year" /></p>

        <input type="hidden" name="movieid" value="@movieId" />

        <p><input type="submit" name="buttonSubmit" value="Submit Changes" /></p>
      </fieldset>
    </form>
    <p><a href="~/Movies">Return to movie listing</a></p>
  </body>
</html>
```

Additional Resources

- Introduction to ASP.NET Web Programming by Using the Razor Syntax
- SQL UPDATE Statement on the W3Schools site

PREVIOUS NEXT

Introducing ASP.NET Web Pages - Deleting Database Data

2/20/2020 • 12 minutes to read • Edit Online

by Tom FitzMacken

This tutorial shows you how to delete an individual database entry. It assumes you have completed the series through Updating Database Data in ASP.NET Web Pages.

What you'll learn:

- How to select an individual record from a listing of records.
- How to delete a single record from a database.
- How to check that a specific button was clicked in a form.

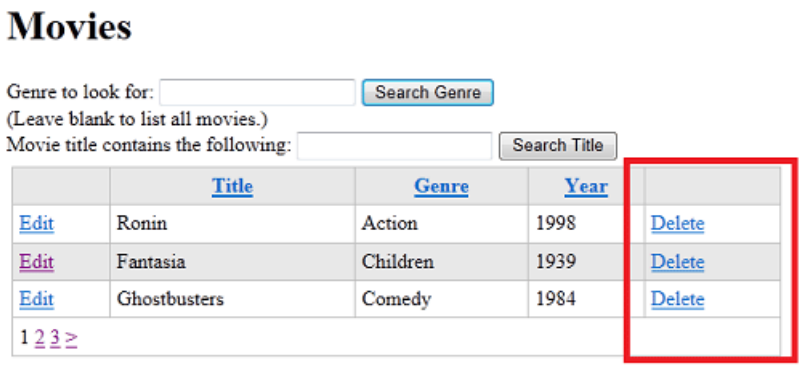
Features/technologies discussed:

- The `WebGrid` helper.
- The SQL `Delete` command.
- The `Database.Execute` method to run a SQL `Delete` command.

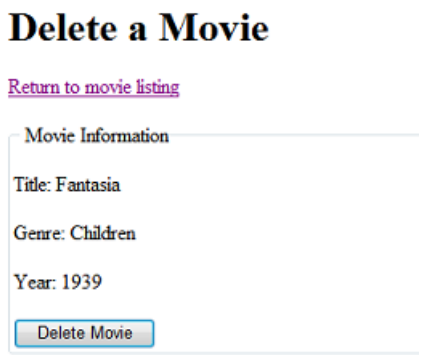
What You'll Build

In the previous tutorial, you learned how to update an existing database record. This tutorial is similar, except that instead of updating the record, you'll delete it. The processes are much the same, except that deleting is simpler, so this tutorial will be short.

In the *Movies* page, you'll update the `WebGrid` helper so that it displays a **Delete** link next to each movie to accompany the **Edit** link you added earlier.



As with editing, when you click the **Delete** link, it takes you to a different page, where the movie information is already in a form:



You can then click the button to delete the record permanently.

Adding a Delete Link to the Movie Listing

You'll start by adding a **Delete** link to the `WebGrid` helper. This link is similar to the **Edit** link you added in a previous tutorial.

Open the *Movies.cshtml* file.

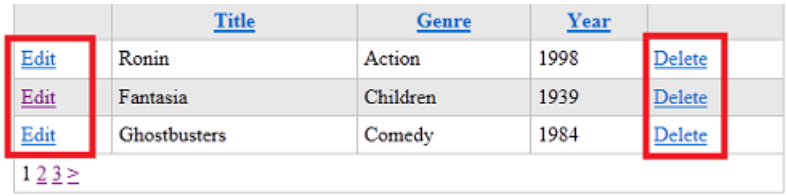
Change the `WebGrid` markup in the body of the page by adding a column. Here's the modified markup:

```
@grid.GetHtml(
    tableStyle: "grid",
    headerStyle: "head",
    alternatingRowStyle: "alt",
    columns: grid.Columns(
grid.Column(format: @<a href="~/EditMovie?id=@item.ID">Edit</a>),
grid.Column("Title"),
grid.Column("Genre"),
grid.Column("Year"),
grid.Column(format: @<a href="~/DeleteMovie?id=@item.ID">Delete</a>)
    )
)
```

The new column is this one:

```
grid.Column(format: @<a href="~/DeleteMovie?id=@item.ID">Delete</a>)
```

The way the grid is configured, the **Edit** column is leftmost in the grid and the **Delete** column is rightmost. (There's a comma after the `Year` column now, in case you didn't notice that.) There's nothing special about where these link columns go, and you could as easily put them next to each other. In this case, they're separate to make them harder to get mixed up.



The new column shows a link (`<a>` element) whose text says "Delete". The target of the link (its `href` attribute) is code that ultimately resolves to something like this URL, with the `id` value different for each movie:

```
http://localhost:43097/DeleteMovie?id=7
```

This link will invoke a page named *DeleteMovie* and pass it the ID of the movie you've selected.

This tutorial won't go into detail about how this link is constructed, because it's almost identical to the **Edit** link from the previous tutorial (Updating Database Data in ASP.NET Web Pages).

Creating the Delete Page

Now you can create the page that will be the target for the **Delete** link in the grid.

> **NOTE**
>
> **Important** The technique of first selecting a record to delete and then using a separate page and button to confirm the process is extremely important for security. As you've read in previous tutorials, making *any* sort of change to your website should *always* be done using a form — that is, using an HTTP POST operation. If you made it possible to change the site just by clicking a link (that is, using a GET operation), people could make simple requests to your site and delete your data. Even a search-engine crawler that's indexing your site could inadvertently delete data just by following links.
>
> When your app lets people change a record, you have to present the record to the user for editing anyway. But you might be tempted to skip this step for deleting a record. Don't skip that step, though. (It's also helpful for users to see the record and confirm that they're deleting the record that they intended.)
>
> In a subsequent tutorial set, you'll see how to add login functionality so a user would have to log in before deleting a record.

Create a page named *DeleteMovie.cshtml* and replace what's in the file with the following markup:

```
<html>
<head>
  <title>Delete a Movie</title>
</head>
<body>
      <h1>Delete a Movie</h1>
        @Html.ValidationSummary()
      <p><a href="~/Movies">Return to movie listing</a></p>

      <form method="post">
        <fieldset>
        <legend>Movie Information</legend>

        <p><span>Title:</span>
         <span>@title</span></p>

        <p><span>Genre:</span>
         <span>@genre</span></p>

        <p><span>Year:</span>
          <span>@year</span></p>

        <input type="hidden" name="movieid" value="@movieId" />
        <p><input type="submit" name="buttonDelete" value="Delete Movie" /></p>
        </fieldset>
      </form>
    </body>
</html>
```

This markup is like the *EditMovie* pages, except that instead of using text boxes (`<input type="text">`), the markup includes `<span>` elements. There's nothing here to edit. All you have to do is display the movie details so that users can make sure that they're deleting the right movie.

The markup already contains a link that lets the user return to the movie listing page.

As in the *EditMovie* page, the ID of the selected movie is stored in a hidden field. (It's passed into the page in the

first place as a query string value.) There's an `Html.ValidationSummary` call that will display validation errors. In this case, the error might be that no movie ID was passed to the page or that the movie ID is invalid. This situation could occur if someone ran this page without first selecting a movie in the *Movies* page.

The button caption is **Delete Movie**, and its name attribute is set to `buttonDelete`. The `name` attribute will be used in the code to identify the button that submitted the form.

You'll have to write code to 1) read the movie details when the page is first displayed and 2) actually delete the movie when the user clicks the button.

Adding Code to Read a Single Movie

At the top of the *DeleteMovie.cshtml* page, add the following code block:

```
@{
    var title = "";
    var genre = "";
    var year = "";
    var movieId = "";

    if(!IsPost){
        if(!Request.QueryString["ID"].IsEmpty() && Request.QueryString["ID"].IsInt()){
            movieId = Request.QueryString["ID"];
            var db = Database.Open("WebPagesMovies");
            var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
            var row = db.QuerySingle(dbCommand, movieId);
            if(row != null) {
                title = row.Title;
                genre = row.Genre;
                year = row.Year;
            }
            else{
                Validation.AddFormError("No movie was found for that ID.");
            }
        }
        else{
            Validation.AddFormError("No movie was found for that ID.");
        }
    }
}
```

This markup is the same as the corresponding code in the *EditMovie* page. It gets the movie ID out of the query string and uses the ID to read a record from the database. The code includes the validation test (`IsInt()` and `row != null`) to make sure that the movie ID being passed to the page is valid.

Remember that this code should only run the first time the page runs. You don't want to re-read the movie record from the database when the user clicks the **Delete Movie** button. Therefore, code to read the movie is inside a test that says `if(!IsPost)` — that is, *if the request is not a post operation (form submission)*.

Adding Code to Delete the Selected Movie

To delete the movie when the user clicks the button, add the following code just inside the closing brace of the `@` block:

```
if(IsPost && !Request["buttonDelete"].IsEmpty()){
    movieId = Request.Form["movieId"];
    var db = Database.Open("WebPagesMovies");
    var deleteCommand = "DELETE FROM Movies WHERE ID = @0";
    db.Execute(deleteCommand, movieId);
    Response.Redirect("~/Movies");
}
```

This code is similar to the code for updating an existing record, but simpler. The code basically runs a SQL `Delete` statement.

As in the *EditMovie* page, the code is in an `if(IsPost)` block. This time, the `if()` condition is a little more complicated:

```
if(IsPost && !Request["buttonDelete"].IsEmpty())
```

There are two conditions here. The first is that the page is being submitted, as you've seen before — `if(IsPost)`.

The second condition is `!Request["buttonDelete"].IsEmpty()`, meaning that the request has an object named `buttonDelete`. Admittedly, it's an indirect way of testing which button submitted the form. If a form contains multiple submit buttons, only the name of the button that was clicked appears in the request. Therefore, logically, if the name of a particular button appears in the request — or as stated in the code, if that button isn't empty — that's the button that submitted the form.

The `&&` operator means "and" (logical AND). Therefore the entire `if` condition is ...

This request is a post (not a first-time request)

AND

The `buttonDelete` *button was the button that submitted the form.*

This form (in fact, this page) contains only one button, so the additional test for `buttonDelete` is technically not required. Still, you're about to perform an operation that will permanently remove data. So you want to be as sure as possible that you're performing the operation only when the user has explicitly requested it. For example, suppose that you expanded this page later and added other buttons to it. Even then, the code that deletes the movie will run only if the `buttonDelete` button was clicked.

As in the *EditMovie* page, you get the ID from the hidden field and then run the SQL command. The syntax for the `Delete` statement is:

`DELETE FROM table WHERE ID = value`

It's vital to include the `WHERE` clause and the ID. If you leave out the WHERE clause, *all the records in the table will be deleted*. As you have seen, you pass the ID value to the SQL command by using a placeholder.

Testing the Movie Delete Process

Now you can test. Run the *Movies* page, and click **Delete** next to a movie. When the *DeleteMovie* page appears, click **Delete Movie**.

When you click the button, the code deletes the movies and returns to the movie listing. There you can search for the deleted movie and confirm that it's been deleted.

Coming Up Next

The next tutorial shows you how to give all the pages on your site a common look and layout.

Complete Listing for Movie Page (Updated with Delete Links)

```
@{
    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
        searchTerm = Request.QueryString["searchGenre"];
    }

    if(!Request.QueryString["searchTitle"].IsEmpty() ) {
      selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
      searchTerm = "%" + Request.QueryString["searchTitle"] + "%";
    }

    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre", rowsPerPage:3);
}
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Movies</title>
      <style type="text/css">
        .grid { margin: 4px; border-collapse: collapse; width: 600px; }
        .grid th, .grid td { border: 1px solid #C0C0C0; padding: 5px; }
        .head { background-color: #E8E8E8; font-weight: bold; color: #FFF; }
        .alt { background-color: #E8E8E8; color: #000; }
      </style>
    </head>
    <body>
      <h1>Movies</h1>
      <form method="get">
        <div>
          <label for="searchGenre">Genre to look for:</label>
          <input type="text" name="searchGenre" value="@Request.QueryString["searchGenre"]" />
          <input type="Submit" value="Search Genre" /><br/>
          (Leave blank to list all movies.)<br/>
          </div>

        <div>
          <label for="SearchTitle">Movie title contains the following:</label>
          <input type="text" name="searchTitle" value="@Request.QueryString["searchTitle"]" />
          <input type="Submit" value="Search Title" /><br/>
        </div>

      </form>
        <div>
          @grid.GetHtml(
            tableStyle: "grid",
            headerStyle: "head",
            alternatingRowStyle: "alt",
            columns: grid.Columns(
                grid.Column(format: @<a href="~/EditMovie?id=@item.ID">Edit</a>),
                grid.Column("Title"),
                grid.Column("Genre"),
                grid.Column("Year"),
                grid.Column(format: @<a href="~/DeleteMovie?id=@item.ID">Delete</a>)
            )
        )
      </div>
      <p>
        <a href="~/AddMovie">Add a movie</a>
      </p>
    </body>
</html>
```

Complete Listing for DeleteMovie Page

```
@{
    var title = "";
    var genre = "";
    var year = "";
    var movieId = "";

    if(!IsPost){
        if(!Request.QueryString["ID"].IsEmpty() && Request.QueryString["ID"].IsInt()){
            movieId = Request.QueryString["ID"];
            var db = Database.Open("WebPagesMovies");
            var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
            var row = db.QuerySingle(dbCommand, movieId);
            if(row != null) {
                title = row.Title;
                genre = row.Genre;
                year = row.Year;
            }
            else{
                Validation.AddFormError("No movie was found for that ID.");
            }
        }
        else{
            Validation.AddFormError("No movie was found for that ID.");
        }
    }

    if(IsPost && !Request["buttonDelete"].IsEmpty()){
        movieId = Request.Form["movieId"];
        var db = Database.Open("WebPagesMovies");
        var deleteCommand = "DELETE FROM Movies WHERE ID = @0";
        db.Execute(deleteCommand, movieId);
        Response.Redirect("~/Movies");
    }
}
<html>
<head>
  <title>Delete a Movie</title>
</head>
<body>
      <h1>Delete a Movie</h1>
        @Html.ValidationSummary()
      <p><a href="~/Movies">Return to movie listing</a></p>

      <form method="post">
        <fieldset>
        <legend>Movie Information</legend>

        <p><span>Title:</span>
         <span>@title</span></p>

        <p><span>Genre:</span>
         <span>@genre</span></p>

        <p><span>Year:</span>
          <span>@year</span></p>

        <input type="hidden" name="movieid" value="@movieId" />
        <p><input type="submit" name="buttonDelete" value="Delete Movie" /></p>
        </fieldset>
        <p><a href="~/Movies">Return to movie listing</a></p>
      </form>
    </body>
</html>
```

Additional Resources

- Introduction to ASP.NET Web Programming by Using the Razor Syntax
- SQL DELETE Statement on the W3Schools site

PREVIOUS NEXT

Introducing ASP.NET Web Pages - Creating a Consistent Layout

2/20/2020 • 16 minutes to read • Edit Online

by Tom FitzMacken

This tutorial shows you how to use *layouts* to create a consistent look for the pages on a site that uses ASP.NET Web Pages. It assumes you have completed the series through Deleting Database Data in ASP.NET Web Pages.

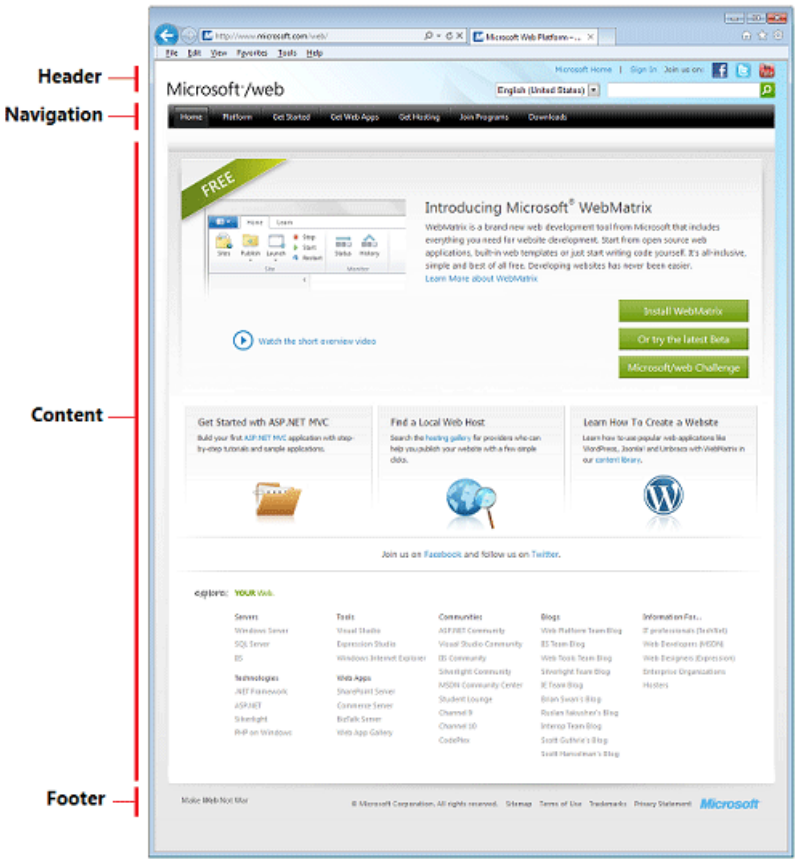
What you'll learn:

- What a layout page is.
- How to combine layout pages with dynamic content.
- How to pass values to a layout page.

About Layouts

The pages you've created so far have all been complete, standalone pages. They all belong to the same site, but they don't have any common elements or a standard look.

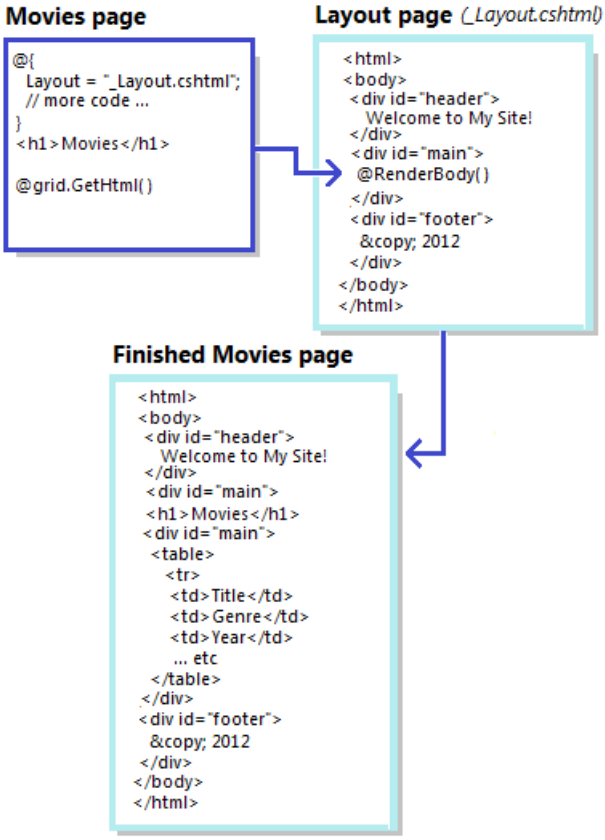
Most sites do have a consistent look and layout. For example, if you go to the Microsoft.com/web site and look around, you see that the pages all adhere to an overall layout and to a visual theme:



An *inefficient* way to create this layout would be to define a header, navigation bar, and footer separately on each of your pages. You'd be duplicating the same markup each time. If you wanted to change something (for example, update the footer), you'd have to change each page separately.

That's where *layout pages* come in. In ASP.NET Web Pages, you can define a layout page that provides an overall container for pages on your site. For example, the layout page can contain the header, navigation area, and footer. The layout page includes a placeholder where the main content goes.

You can then define individual content pages that contain the markup and the code for only that page. Content pages don't have to be complete HTML pages; they don't even have to have a `<body>` element. They also have a line of code that tells ASP.NET what layout page you want to display the content in. Here's a picture that shows roughly how this relationship works:



This interaction is easy to understand when you see it in action. In this tutorial, you'll change your movies pages to use a layout.

Adding a Layout Page

You'll start by creating a layout page that defines a typical page layout with a header, footer, and an area for the main content. In the WebPagesMovies site, add a CSHTML page named *_Layout.cshtml*.

The leading underscore (`_`) character is significant. If a page's name starts with an underscore, ASP.NET won't directly send that page to the browser. This convention lets you define pages that are required for your site but that users shouldn't be able to request directly.

Replace the content in the page with the following:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Movie Site</title>
    <link href="~/Styles/Movies.css" rel="stylesheet" type="text/css" />
  </head>
  <body>
    <div id="container">
        <div id="header">
          <h1>My Movie Site</h1>
        </div>
        <div id="main">
          @RenderBody()
        </div>
        <div id="footer">
          &copy; @DateTime.Now.Year My Movie Site
        </div>
    </div>
  </body>
</html>
```

As you can see, this markup is just HTML that uses `<div>` elements to define three sections in the page plus one more `<div>` element to hold the three sections. The footer contains a bit of Razor code: `@DateTime.Now.Year`, which will render the current year at that location in the page.

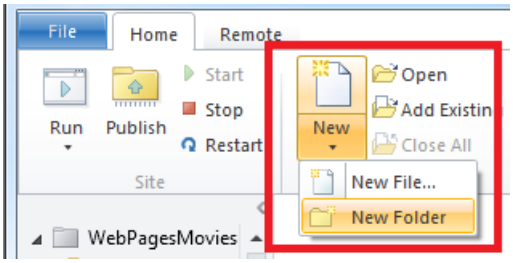
Notice that there's a link to a style sheet named *Movies.css*. The style sheet is where the details of the physical layout of the elements will be defined. You'll create that in a moment.

The only unusual feature in this *_Layout.cshtml* page is the `@Render.Body()` line. That's the placeholder where the content will go when this layout is merged with another page.

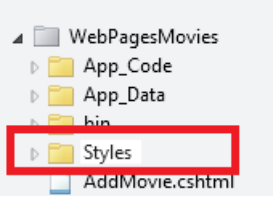
Adding a .css File

The preferred way to define the actual arrangement (that is, appearance) of elements on the page is to use cascading style sheet (CSS) rules. So you'll create a *.css* file that has the rules for your new layout.

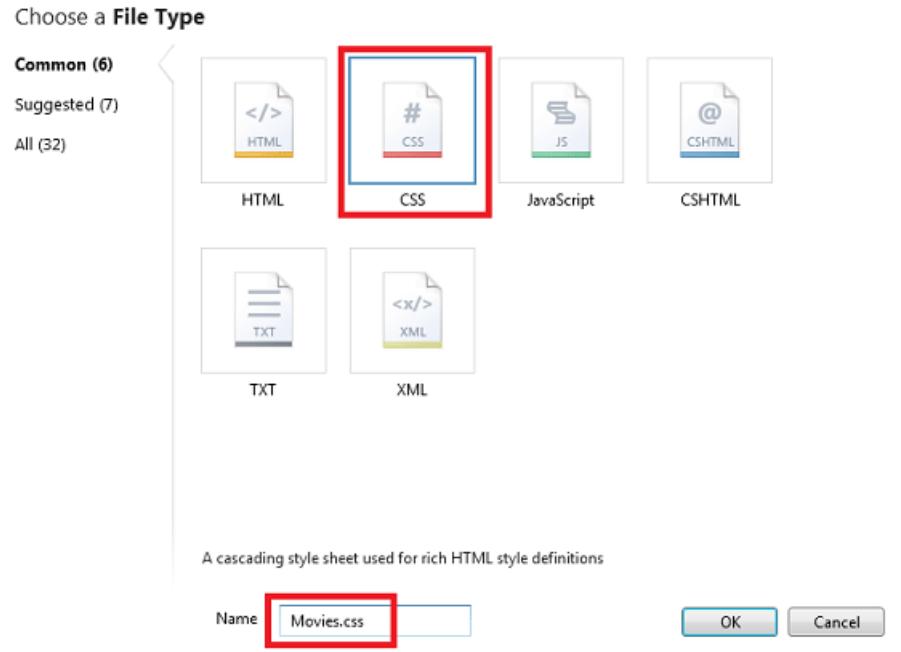
In WebMatrix, select the root of your site. Then in the **Files** tab of the ribbon, click the arrow under the **New** button and then click **New Folder**.



Name the new folder *Styles*.



Inside the new *Styles* folder, create a file named *Movies.css*.



Replace the contents of the new *.css* file with the following:

```
html{ height:100%; margin:0; padding:0; }

body {
  height:60%;
  font-family:'Trebuchet MS',  'Arial', 'Helvetica', 'sans-serif';
  font-size:10pt;
  background-color: LightGray;
  line-height:1.6em;
}

h1{ font-size:1.6em; }
h2{ font-size:1.4em; }

#container{
   min-height:100%;
   position:relative;
   left:10%;
}

#header{
  padding:8px;
  width:80%;
  background-color:#4b6c9e;
  color:White;
}

#main{
  width:80%;
  padding: 8px;
  padding-bottom:4em;
  background-color:White;
}

#footer{
  width:80%;
  height:2em;
  padding:8px;
  margin-top:-2em;
  background-color:LightGray;
}

.head { background-color: #E8E8E8; font-weight: bold; color: #FFF; }
.grid th, .grid td { border: 1px solid #C0C0C0; padding: 5px; }
.alt { background-color: #E8E8E8; color: #000; }
.selected {background-color:Yellow;}
span.caption {width:100px;}
span.dataDisplay {font-weight:bold;}
```

We won't say much about these CSS rules, except to note two things. One is that in addition to setting fonts and sizes, the rules use absolute positioning to establish the location of the header, footer, and main content area. If you're new to positioning in CSS, you can read the CSS Positioning tutorial at the W3Schools site.

The other thing to note is that at the bottom, we've copied the style rules that were originally defined individually in the *Movies.cshtml* file. These rules were used in the Introduction to Displaying Data by Using ASP.NET Web Pages tutorial to make the `WebGrid` helper render markup that added stripes to the table. (If you're going to use a *.css* file for style definitions, you might as well put the style rules for the whole site in it.)

Updating the Movies File to Use the Layout

Now you can update the existing files in your site to use the new layout. Open the *Movies.cshtml* file. At the top, as the first line of code, add the following:

```
Layout = "~/_Layout.cshtml";
```

The page now starts out this way:

```
@{
    Layout = "~/_Layout.cshtml";

    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    // Etc.
```

This one line of code tells ASP.NET that when the *Movies* page runs, it should be merged with the *_Layout.cshtml* file.

Since the *Movies.cshtml* file now uses a layout page, you can remove the markup from the *Movies.cshtml* page that's taken care of by the *_Layout.cshtml* file. Take out the `<!DOCTYPE>`, `<html>`, and `<body>` opening and closing tags. Take out the entire `<head>` element and its contents, which includes the style rules for the grid, since you've now got those rules in a *.css* file. While you're at it, change the existing `<h1>` element to an `<h2>` element; you have an `<h1>` element in the layout page already. Change the `<h2>` text to "List Movies".

Normally you wouldn't have to make these sorts of changes in a content page. When you start your site out with a layout page, you create content pages without all these elements to begin with. In this case, though, you're converting a standalone page to one that uses a layout, so there's a bit of cleanup.

When you're finished, the *Movies.cshtml* page will look like the following:

```
@{
    Layout = "~/_Layout.cshtml";

    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

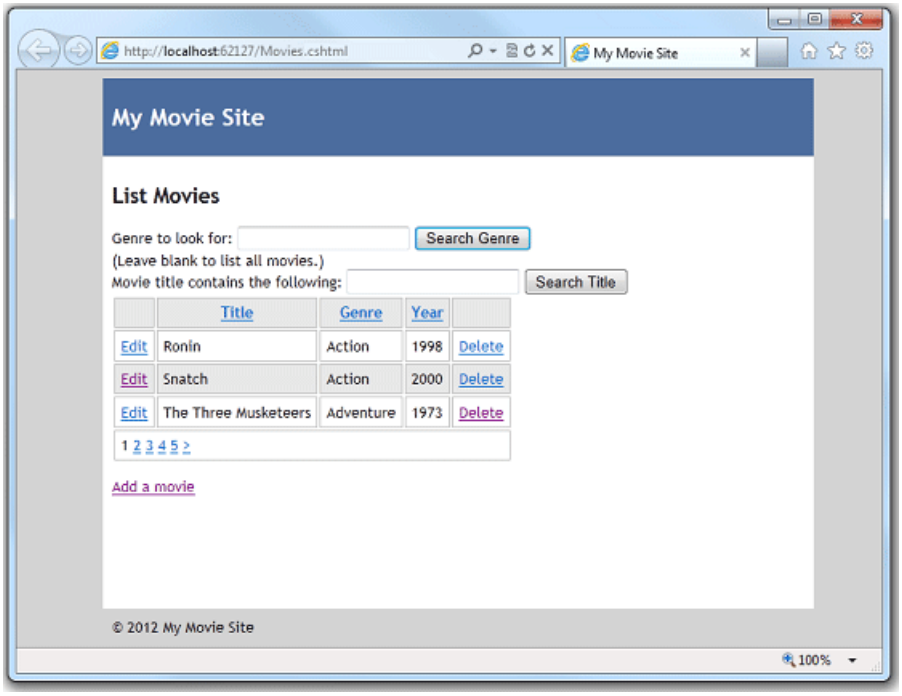
    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
        searchTerm = Request.QueryString["searchGenre"];
    }

    if(!Request.QueryString["searchTitle"].IsEmpty() ) {
      selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
      searchTerm = "%" + Request.QueryString["searchTitle"] + "%";
    }

    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre", rowsPerPage:3);
}
  <h2>List Movies</h2>
  <form method="get">
    <div>
      <label for="searchGenre">Genre to look for:</label>
      <input type="text" name="searchGenre"
         value="@Request.QueryString["searchGenre"]" />
      <input type="Submit" value="Search Genre" /><br/>
      (Leave blank to list all movies.)<br/>
    </div>
    <div>
       <label for="SearchTitle">Movie title contains the following:</label>
       <input type="text" name="searchTitle" value="@Request.QueryString["searchTitle"]" />
       <input type="Submit" value="Search Title" /><br/>
    </div>
  </form>
  <div>
    @grid.GetHtml(
        tableStyle: "grid",
        headerStyle: "head",
        alternatingRowStyle: "alt",
        columns: grid.Columns(
            grid.Column(format: @<a href="~/EditMovie?id=@item.ID">Edit</a>),
            grid.Column("Title"),
            grid.Column("Genre"),
            grid.Column("Year"),
            grid.Column(format: @<a href="~/DeleteMovie?id=@item.ID">Delete</a>)
       )
    )
  </div>
  <p><a href="~/AddMovie">Add a movie</a></p>
```

Testing the Layout

Now you can see what the layout looks like. In WebMatrix, right-click the *Movies.cshtml* page and select **Launch in browser**. When the browser displays the page, it looks like this page:



ASP.NET has merged the content of the Movies.cshtml page into the *_Layout.cshtml* page right where the `RenderBody` method is. And of course the *_Layout.cshtml* page references a *.css* file that defines the look of the page.

Updating the AddMovie Page to Use the Layout

The real benefit of layouts is that you can use them for all the pages in your site. Open the *AddMovie.cshtml* page.

You might remember that the *AddMovie.cshtml* page originally had some CSS rules in it to define the look of validation error messages. Since you have a *.css* file for your site now, you can move those rules to the *.css* file. Remove them from the *AddMovie.cshtml* file and add them to the bottom of the *Movies.css* file. You are moving the following rules:

```
.field-validation-error {
  font-weight:bold;
  color:red;
  background-color:yellow;
 }
.validation-summary-errors{
  border:2px dashed red;
  color:red;
  background-color:yellow;
  font-weight:bold;
  margin:12px;
}
```

Now make the same sorts of changes in *AddMovie.cshtml* that you did for *Movies.cshtml* — add `Layout="~/_Layout.cshtml;` and remove the HTML markup that's now extraneous. Change the `<h1>` element to `<h2>`. When you're done, the page will look like this example:

```
@{
    Layout = "~/_Layout.cshtml";
    Validation.RequireField("title", "You must enter a title");
    Validation.RequireField("genre", "Genre is required");
    Validation.RequireField("year", "You haven't entered a year");

    var title = "";
    var genre = "";
    var year = "";

    if(IsPost){
        if(Validation.IsValid()){
            title = Request.Form["title"];
            genre = Request.Form["genre"];
            year = Request.Form["year"];

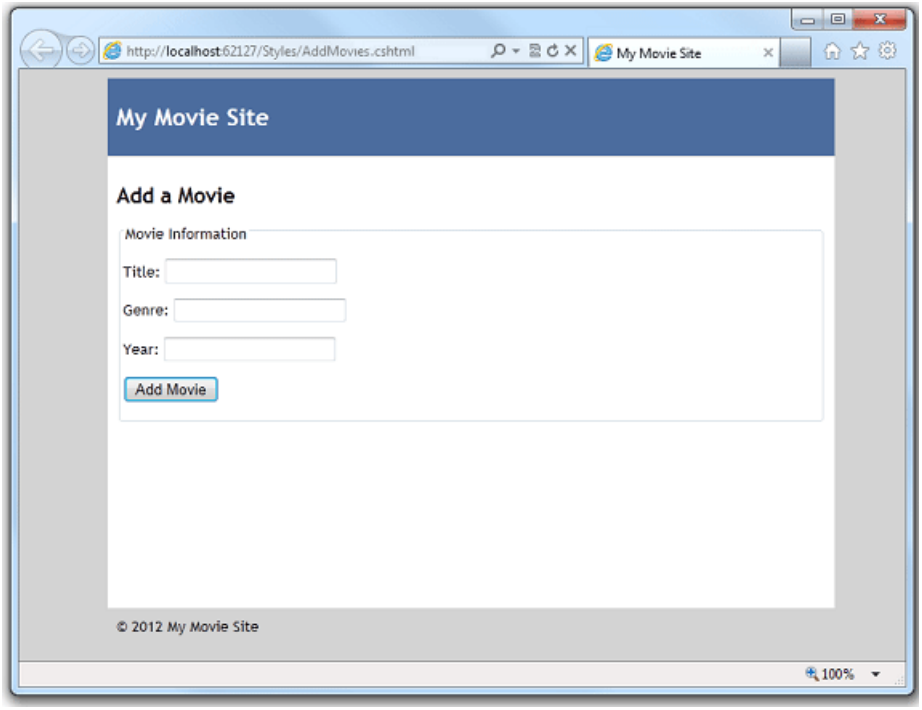
            var db = Database.Open("WebPagesMovies");
            var insertCommand =
                "INSERT INTO Movies (Title, Genre, Year) Values(@0, @1, @2)";
            db.Execute(insertCommand, title, genre, year);
            Response.Redirect("~/Movies");
        }
    }
}
  <h2>Add a Movie</h2>
    @Html.ValidationSummary()
 <form method="post">
  <fieldset>
    <legend>Movie Information</legend>
    <p><label for="title">Title:</label>
      <input type="text" name="title" value="@Request.Form["title"]" />
      @Html.ValidationMessage("title")
    </p>

    <p><label for="genre">Genre:</label>
      <input type="text" name="genre" value="@Request.Form["genre"]" />
      @Html.ValidationMessage("genre")
    </p>

    <p><label for="year">Year:</label>
      <input type="text" name="year" value="@Request.Form["year"]" />
      @Html.ValidationMessage("year")
    </p>

    <p><input type="submit" name="buttonSubmit" value="Add Movie" /></p>
  </fieldset>
  </form>
```

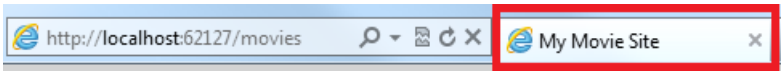
Run the page. Now it looks like this illustration:



You want to make similar changes to the pages in the site — *EditMovie.cshtml* and *DeleteMovie.cshtml*. However, before you do, you can make another change to the layout that makes it a little more flexible.

Passing Title Information to the Layout Page

The *_Layout.cshtml* page that you created has a `<title>` element that's set to "My Movie Site". Most browsers display the content of this element as the text on a tab:



This title information is generic. Suppose that you want the title text to be more specific to the current page. (The title text is also used by search engines to determine what your page is about.) You can pass information from a content page like *Movies.cshtml* or *AddMovie.cshtml* to the layout page, and then use that information to customize what the layout page renders.

Open the *Movies.cshtml* page again. In the code at the top, add the following line:

```
Page.Title = "List Movies";
```

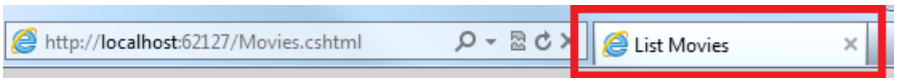
The `Page` object is available on all *.cshtml* pages and is for this purpose, namely to share information between a page and its layout.

Open the *_Layout.cshtml* page. Change the `<title>` element so that it looks like this markup:

```
<title>@Page.Title</title>
```

This code renders whatever is in the `Page.Title` property right at that location in the page.

Run the *Movies.cshtml* page. This time the browser tab shows what you passed as the value of `Page.Title`:



If you want, view the page source in the browser. You can see that the `<title>` element is rendered as `<title>List Movies</title>`.

TIP

The Page Object

A useful feature of `Page` is that it's a dynamic object — the `Title` property is not a fixed or reserved name. You can use *any* name for a value of the `Page` object. For example, you could as easily have passed the title by using a property named `Page.CurrentName` or `Page.MyPage`. The only restriction is that the name has to follow the normal rules for what properties can be named. (For example, the name can't contain a space.)

You can pass any number of values by using the `Page` object. If you wanted to pass movie information to the layout page, you could pass values by using something like `Page.MovieTitle` and `Page.Genre` and `Page.MovieYear`. (Or any other names that you invented to store the information.) The only requirement — which is probably obvious — is that you have to use the same names in the content page and the layout page.

The information you pass by using the `Page` object isn't limited to just text to display on the layout page. You can pass a value to the layout page, and then code in the layout page can use the value to decide whether to display a section of the page, what *.css* file to use, and so on. The values you pass in the `Page` object are like any other values that you use in code. It's just that the values originate in the content page and are passed to the layout page.

Open the *AddMovie.cshtml* page and add a line to the top of the code that provides a title for the *AddMovie.cshtml* page:

```
Page.Title = "Add a Movie";
```

Run the *AddMovie.cshtml* page. You see the new title there:

Updating the Remaining Pages to Use the Layout

Now you can finish the remaining pages in your site so that they use the new layout. Open *EditMovie.cshtml* and *DeleteMovie.cshtml* in turn and make the same changes in each.

Add the line of code that links to the layout page:

```
Layout = "~/_Layout.cshtml";
```

Add a line to set the title of the page:

```
Page.Title = "Edit a Movie";
```

or:

```
Page.Title = "Delete a Movie";
```

Remove all the extraneous HTML markup — basically, leave only the bits that are inside the `<body>` element (plus the code block at the top).

Change the `<h1>` element to be an `<h2>` element.

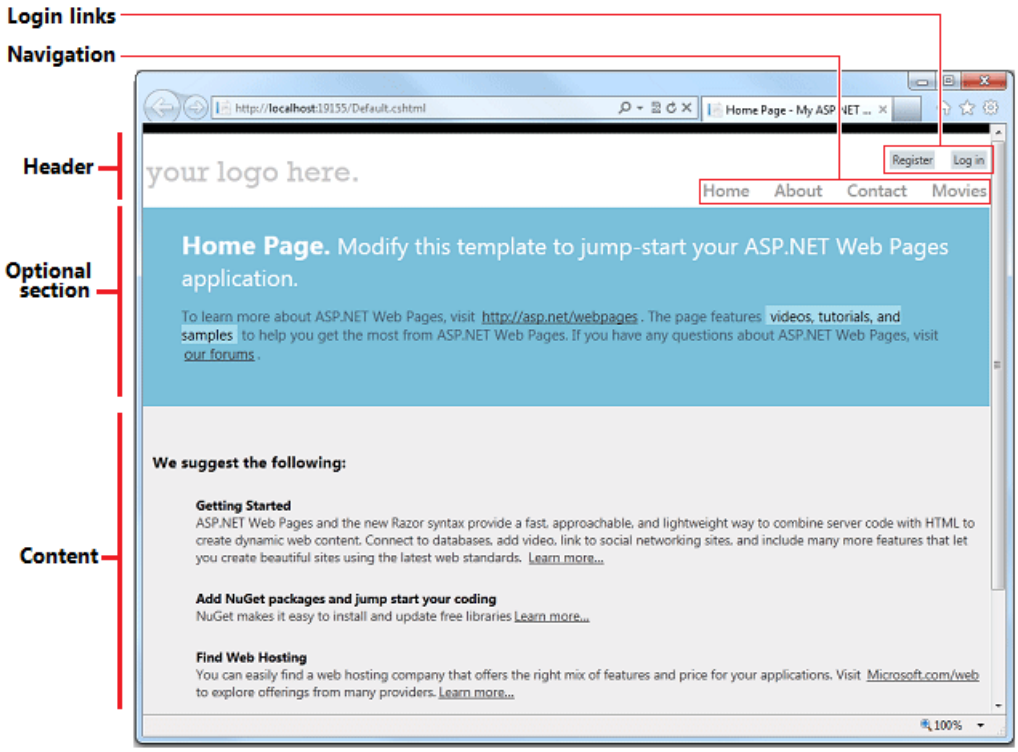
When you've made these changes, test each and make sure that it's displaying properly and that the title is correct.

Parting Thoughts About Layout Pages

In this tutorial you created a *_Layout.cshtml* page and used the `RenderBody` method to merge content from another page. That's the basic pattern for using layouts in Web Pages.

Layout pages have additional features that we didn't cover here. For example, you can nest layout pages — one layout page can in turn reference another. Nested layouts can be useful if you're working with subsections of a site that require different layouts. You can also use additional methods (for example, `RenderSection`) to set up named sections in the layout page.

The combination of layout pages and *.css* files is powerful. As you'll see in the next tutorial series, in WebMatrix you can create a site based on a *template*, which gives you a site that has prebuilt functionality in it. The templates make good use of layout pages and CSS to create sites that look great and that have features like menus. Here's a screenshot of the home page from a site based on a template, showing features that use layout pages and CSS:



Complete Listing for Movie Page (Updated to Use a Layout Page)

```
@{
    Layout = "~/_Layout.cshtml";
    Page.Title = "List Movies";

    var db = Database.Open("WebPagesMovies") ;
    var selectCommand = "SELECT * FROM Movies";
    var searchTerm = "";

    if(!Request.QueryString["searchGenre"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Genre = @0";
        searchTerm = Request.QueryString["searchGenre"];
    }

    if(!Request.QueryString["searchTitle"].IsEmpty() ) {
        selectCommand = "SELECT * FROM Movies WHERE Title LIKE @0";
        searchTerm = "%" + Request.QueryString["searchTitle"] + "%";
    }

    var selectedData = db.Query(selectCommand, searchTerm);
    var grid = new WebGrid(source: selectedData, defaultSort: "Genre", rowsPerPage:3);
}

<h2>List Movies</h2>
    <form method="get">
      <div>
        <label for="searchGenre">Genre to look for:</label>
        <input type="text" name="searchGenre" value="@Request.QueryString["searchGenre"]" />
        <input type="Submit" value="Search Genre" /><br/>
        (Leave blank to list all movies.)<br/>
      </div>

      <div>
        <label for="SearchTitle">Movie title contains the following:</label>
        <input type="text" name="searchTitle" value="@Request.QueryString["searchTitle"]" />
        <input type="Submit" value="Search Title" /><br/>
      </div>
    </form>

<div>
    @grid.GetHtml(
        tableStyle: "grid",
        headerStyle: "head",
        alternatingRowStyle: "alt",
        columns: grid.Columns(
    grid.Column(format: @<a href="~/EditMovie?id=@item.ID">Edit</a>),
    grid.Column("Title"),
    grid.Column("Genre"),
    grid.Column("Year"),
    grid.Column(format: @<a href="~/DeleteMovie?id=@item.ID">Delete</a>)
        )
    )
</div>
<p><a href="~/AddMovie">Add a movie</a></p>
```

Complete Page Listing for Add Movie Page (Updated for Layout)

```
@{
    Layout = "~/_Layout.cshtml";
    Page.Title = "Add a Movie";

    Validation.RequireField("title", "You must enter a title");
    Validation.RequireField("genre", "Genre is required");
    Validation.RequireField("year", "You haven't entered a year");

    var title = "";
    var genre = "";
    var year = "";

    if(IsPost){
        if(Validation.IsValid()){
            title = Request.Form["title"];
            genre = Request.Form["genre"];
            year = Request.Form["year"];

            var db = Database.Open("WebPagesMovies");
            var insertCommand = "INSERT INTO Movies (Title, Genre, Year) VALUES(@0, @1, @2)";
            db.Execute(insertCommand, title, genre, year);

            Response.Redirect("~/Movies");
        }
    }
}

<h2>Add a Movie</h2>
@Html.ValidationSummary()
<form method="post">
<fieldset>
    <legend>Movie Information</legend>
    <p><label for="title">Title:</label>
        <input type="text" name="title" value="@Request.Form["title"]" />
        @Html.ValidationMessage("title")

    <p><label for="genre">Genre:</label>
        <input type="text" name="genre" value="@Request.Form["genre"]" />
        @Html.ValidationMessage("genre")

    <p><label for="year">Year:</label>
        <input type="text" name="year" value="@Request.Form["year"]" />
        @Html.ValidationMessage("year")

    <p><input type="submit" name="buttonSubmit" value="Add Movie" /></p>
</fieldset>
</form>
```

Complete Page Listing for Delete Movie Page (Updated for Layout)

```
@{
    Layout = "~/_Layout.cshtml";
    Page.Title = "Delete a Movie";

    var title = "";
    var genre = "";
    var year = "";
    var movieId = "";

    if(!IsPost){
        if(!Request.QueryString["ID"].IsEmpty() && Request.QueryString["ID"].AsInt() > 0){
            movieId = Request.QueryString["ID"];
            var db = Database.Open("WebPagesMovies");
            var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
            var row = db.QuerySingle(dbCommand, movieId);
```

```
            if(row != null) {
                title = row.Title;
                genre = row.Genre;
                year = row.Year;
            }
            else{
                Validation.AddFormError("No movie was found for that ID.");
                // If you are using a version of ASP.NET Web Pages 2 that's
                // earlier than the RC release, comment out the preceding
                // statement and uncomment the following one.
                //ModelState.AddFormError("No movie was found for that ID.");
            }
        }
        else{
            Validation.AddFormError("No movie was found for that ID.");
            // If you are using a version of ASP.NET Web Pages 2 that's
            // earlier than the RC release, comment out the preceding
            // statement and uncomment the following one.
            //ModelState.AddFormError("No movie was found for that ID.");
        }
    }

    if(IsPost && !Request["buttonDelete"].IsEmpty()){
        movieId = Request.Form["movieId"];
        var db = Database.Open("WebPagesMovies");
        var deleteCommand = "DELETE FROM Movies WHERE ID = @0";
        db.Execute(deleteCommand, movieId);
        Response.Redirect("~/Movies");
    }

}

<h2>Delete a Movie</h2>
@Html.ValidationSummary()
<p><a href="~/Movies">Return to movie listing</a></p>

<form method="post">
<fieldset>
<legend>Movie Information</legend>

<p><span>Title:</span>
    <span>@title</span></p>

<p><span>Genre:</span>
    <span>@genre</span></p>

<p><span>Year:</span>
    <span>@year</span></p>

<input type="hidden" name="movieid" value="@movieId" />
<p><input type="submit" name="buttonDelete" value="Delete Movie" /></p>
</fieldset>
</form>
```

Complete Page Listing for Edit Movie Page (Updated for Layout)

```
@{
    Layout = "~/_Layout.cshtml";
    Page.Title = "Edit a Movie";

    var title = "";
    var genre = "";
    var year = "";
    var movieId = "";

    if(!IsPost){
```

```
        if(!Request.QueryString["ID"].IsEmpty() && Request.QueryString["ID"].IsInt()) {
            movieId = Request.QueryString["ID"];
            var db = Database.Open("WebPagesMovies");
            var dbCommand = "SELECT * FROM Movies WHERE ID = @0";
            var row = db.QuerySingle(dbCommand, movieId);

            if(row != null) {
                title = row.Title;
                genre = row.Genre;
                year = row.Year;
            }
            else{
                Validation.AddFormError("No movie was selected.");
                // If you are using a version of ASP.NET Web Pages 2 that's
                // earlier than the RC release, comment out the preceding
                // statement and uncomment the following one.
                //ModelState.AddFormError("No movie was selected.");
            }
        }
        else{
            Validation.AddFormError("No movie was selected.");
            // If you are using a version of ASP.NET Web Pages 2 that's
            // earlier than the RC release, comment out the preceding
            // statement and uncomment the following one.
            //ModelState.AddFormError("No movie was selected.");
        }
    }

    if(IsPost){
        Validation.RequireField("title", "You must enter a title");
        Validation.RequireField("genre", "Genre is required");
        Validation.RequireField("year", "You haven't entered a year");
        Validation.RequireField("movieid", "No movie ID was submitted!");

        title = Request.Form["title"];
        genre = Request.Form["genre"];
        year = Request.Form["year"];
        movieId = Request.Form["movieId"];

        if(Validation.IsValid()){
            var db = Database.Open("WebPagesMovies");
            var updateCommand = "UPDATE Movies SET Title=@0, Genre=@1, Year=@2 WHERE Id=@3";
            db.Execute(updateCommand, title, genre, year, movieId);
            Response.Redirect("~/Movies");
        }
    }
}

<h2>Edit a Movie</h2>
@Html.ValidationSummary()
<form method="post">
<fieldset>
    <legend>Movie Information</legend>

    <p><label for="title">Title:</label>
        <input type="text" name="title" value="@title" /></p>

    <p><label for="genre">Genre:</label>
        <input type="text" name="genre" value="@genre" /></p>

    <p><label for="year">Year:</label>
        <input type="text" name="year" value="@year" /></p>

    <input type="hidden" name="movieid" value="@movieId" />

    <p><input type="submit" name="buttonSubmit" value="Submit Changes" /></p>
</fieldset>
</form>
<p><a href="~/Movies">Return to movie listing</a></p>
```

Coming Up Next

In the next tutorial, you'll learn how to publish your site to the Internet so everyone can see it.

Additional Resources

- Creating a Consistent Look — An article that provides some more detail on working with layouts. It also describes how to pass a value to a layout page that shows or hides some of the content.
- Nested Layout Pages with Razor — Mike Brind blogs an example of how to nest layout pages. (Includes a download of the pages.)

PREVIOUS NEXT

Introducing ASP.NET Web Pages - Publishing a Site by Using WebMatrix

9/16/2020 • 10 minutes to read • Edit Online

by Tom FitzMacken

This tutorial is the final installment in the tutorial set that introduces ASP.NET Web Pages and Microsoft WebMatrix. It discusses how to publish your site to the Internet so that others can work with it. It assumes you have completed the series through Creating a Consistent Look for ASP.NET Web Pages Sites.

You'll learn how to publish your site using:

- Microsoft Azure
- Web Hosting Company

About Publishing Your Site

Up to now, you've done all your work on a local computer, including testing your pages. To run your *.cshtml* pages, you've used the web server that's built into WebMatrix, namely IIS Express. But of course no one can see the site you've created except you. To let others work with your site, you have to publish it to the Internet.

Unless you have access to a public web server already, publishing means that you have to have an account with a *cloud platform* or a *hosting provider*. A cloud platform, such as Microsoft Azure, provides on-demand infrastructure for your applications. A hosting provider is a company that owns publicly accessible web servers and that will rent you space for your site. Hosting plans run from a few dollars a month (or even free) for small sites to many hundreds of dollars a month for high-volume commercial websites.

> **NOTE**
>
> You might have access to a public web server via the internet service provider (ISP) that you use to get internet service at home. However, your hosting provider must support ASP.NET Web Pages. Many ISPs don't, but it's always worth checking.

In this tutorial, we'll give you an overview of how to publish. It's not practical to provide exact details for everything, because the process differs a bit for every hosting provider. But you'll get a good idea of how the process works.

This tutorial contains four sections:

1. Setting up the default page
2. Publishing (choose one of the following)
   a. Publishing Your Site to Microsoft Azure
   b. Publishing Your Site to a Web Hosting Company
3. Updating the Live Site: Republishing

Setting up the default page

When a user navigates to the base address for your web site, the default page for your site is displayed to the user. For example, when *Default.htm* is set as the default page for the site at `www.contoso.com`, then navigating to `www.contoso.com` is the same as navigating to `www.contoso.com/Default.htm`.

Currently, your site uses **Default.cshtml** as the default page. This page is fine for your default page, but in this tutorial you have not added any content to that page so it would display a blank page. Open Default.cshtml and replace the content with the following code.

```
@{
    Layout = "~/_Layout.cshtml";
    Page.Title = "Welcome to Movie site";
}

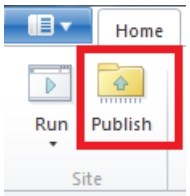
<p><a href="~/Movies">Go to movie listing</a></p>
```

Now your site is ready for publication. First, you will see how to deploy the site to Azure, and then how to deploy it to a web hosting company. Either option works for your web site, and you only need to follow one of the deployment options.

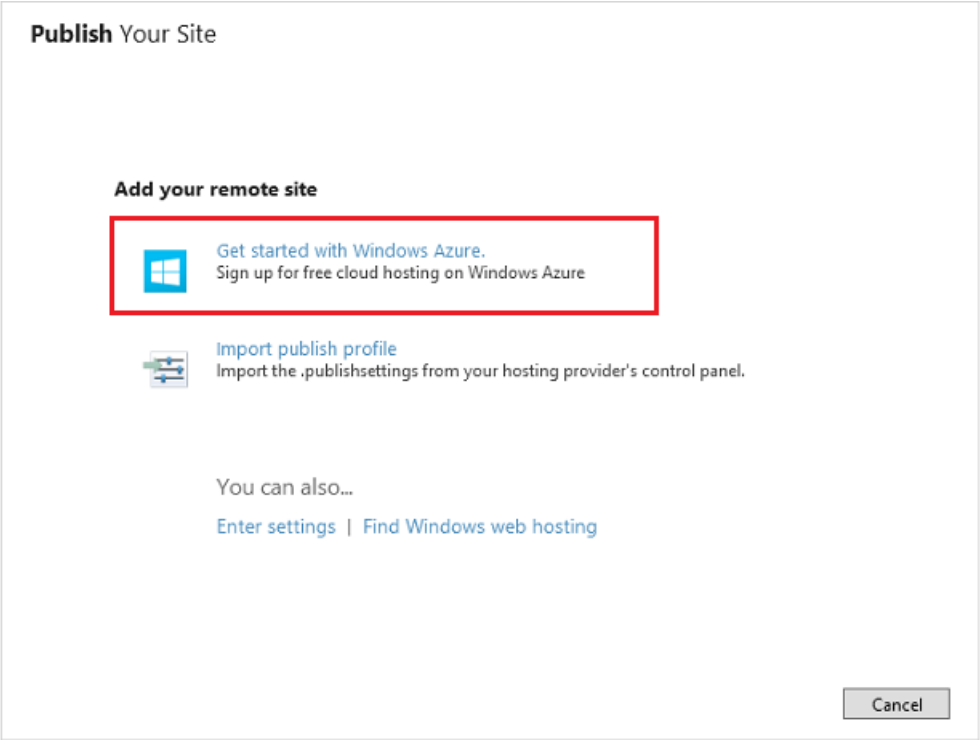
Publishing Your Site to Microsoft Azure

This tutorial will first show you how to deploy your site to Microsoft Azure. By signing in with a Microsoft account, you can create up to 10 free sites on Azure. These free sites provide a convenient way to test your sites. You can always delete this example site later to avoid using all of your free sites. You can create a free trial account in just a couple of minutes. For details, see Azure Free Trial.

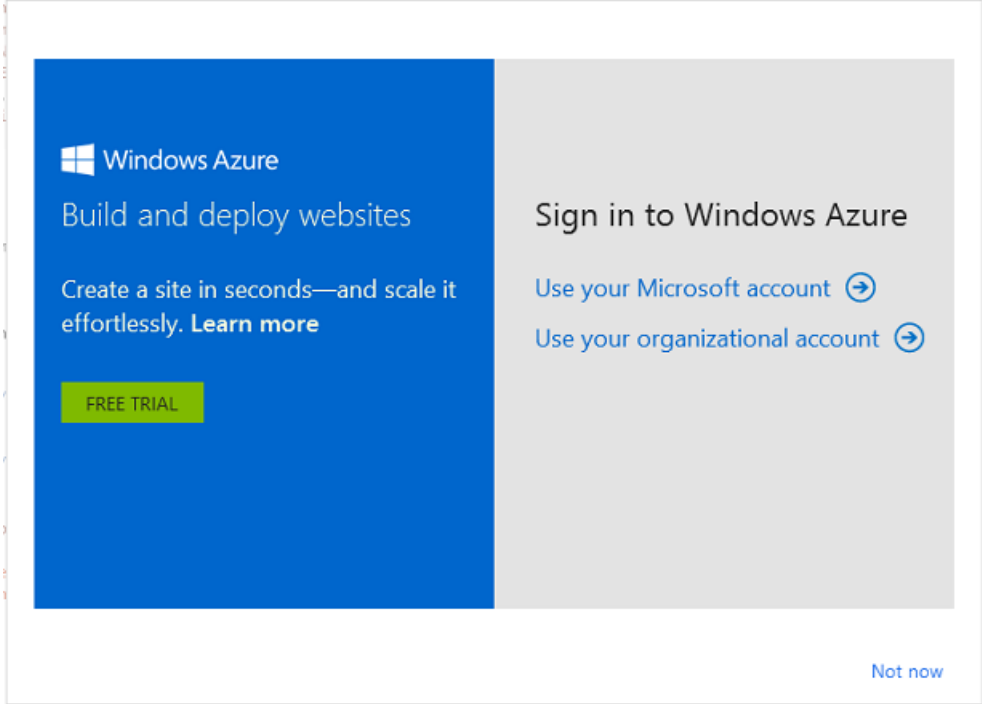
In the WebMatrix ribbon, click the **Publish** button.



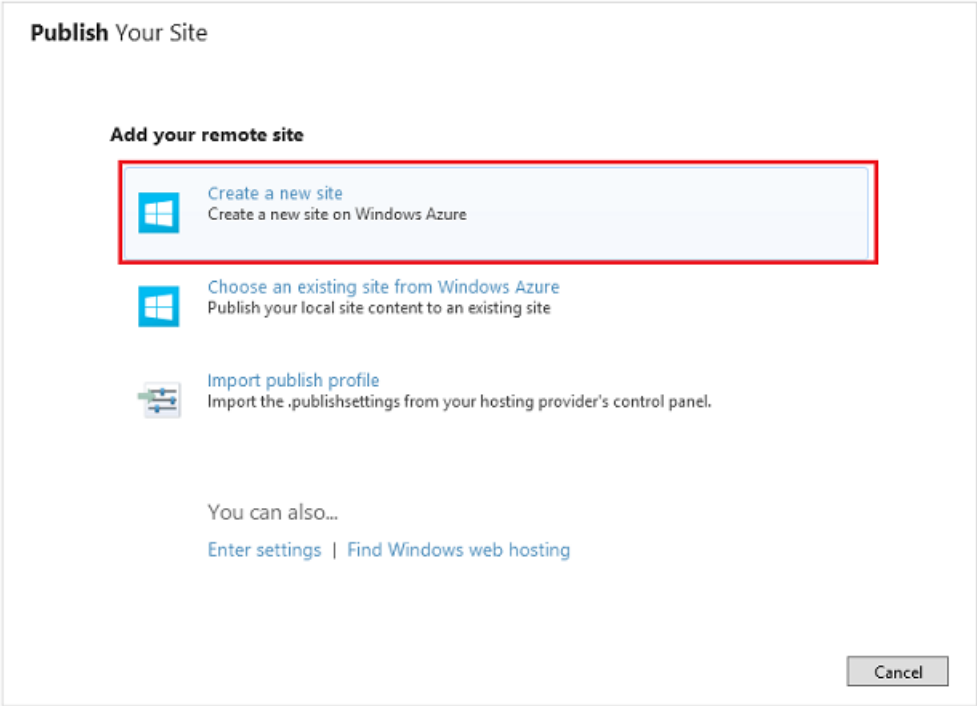
The **Publish Your Site** dialog box is displayed. If you have not signed in to your Microsoft account, the dialog box will contain a **Get started with Azure** link. Click this link.



If you have not signed in to a Microsoft account, you are again given the opportunity to sign in. You must sign in to a Microsoft account to publish your site on Azure.

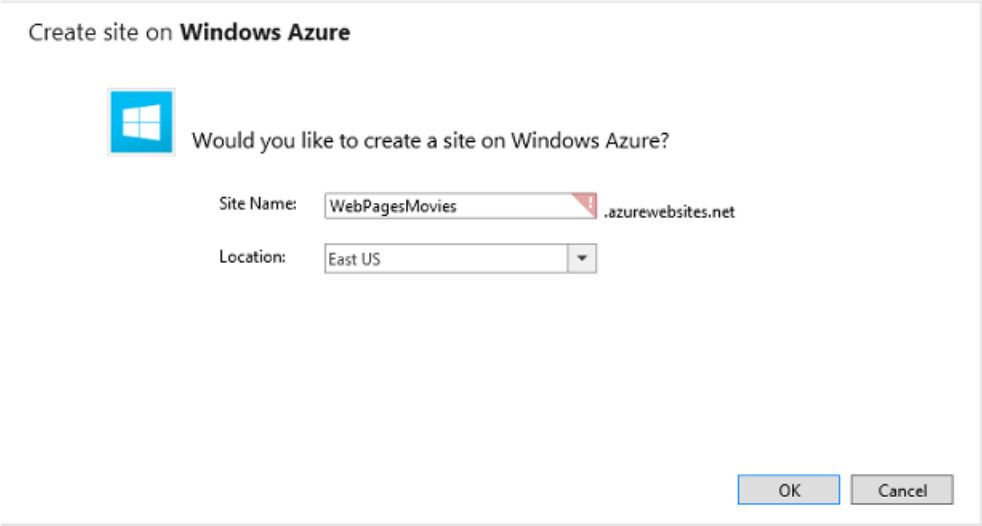


After signing in to your Microsoft account, the dialog box contains links to create a new site on Azure or connect to one of your existing sites on Azure.

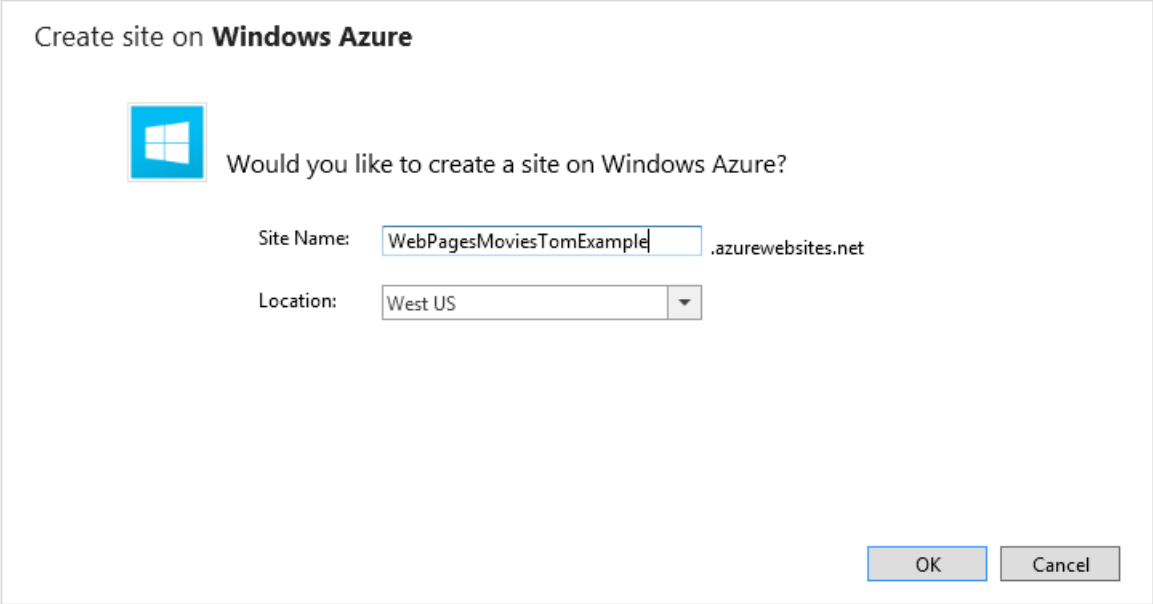


Select **Create a new site**.

If you named your project **WebPagesMovies**, the default name for your site will be **webpagesmovies.azurewebsites.net**. This default name is most likely not available, as indicated by the red exclamation mark.

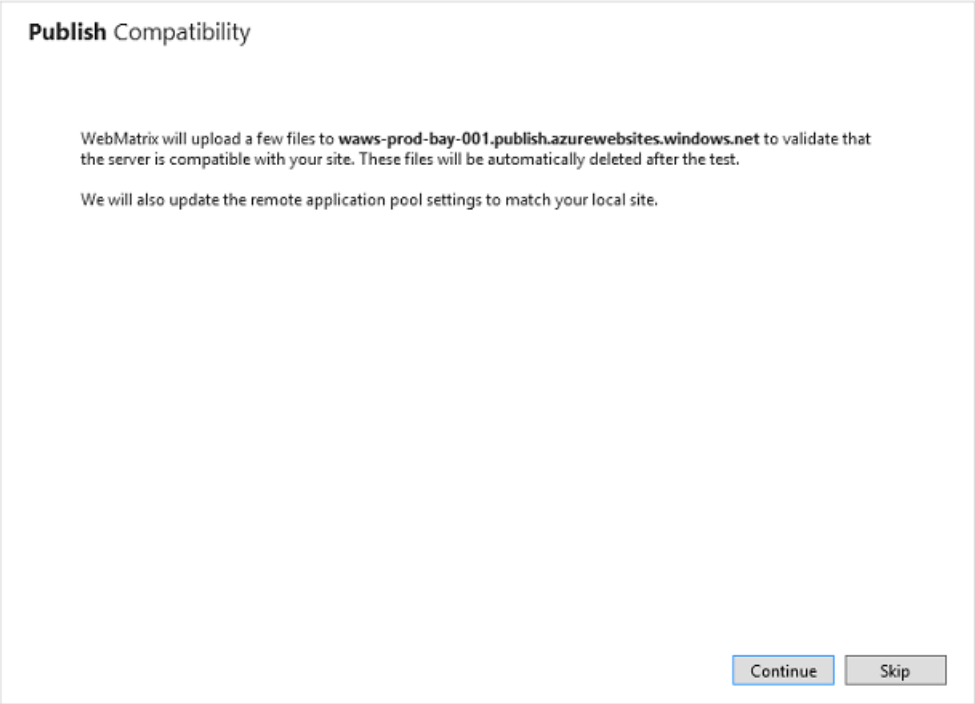


Change the site name to something that is available, and select a location that is close to your location.



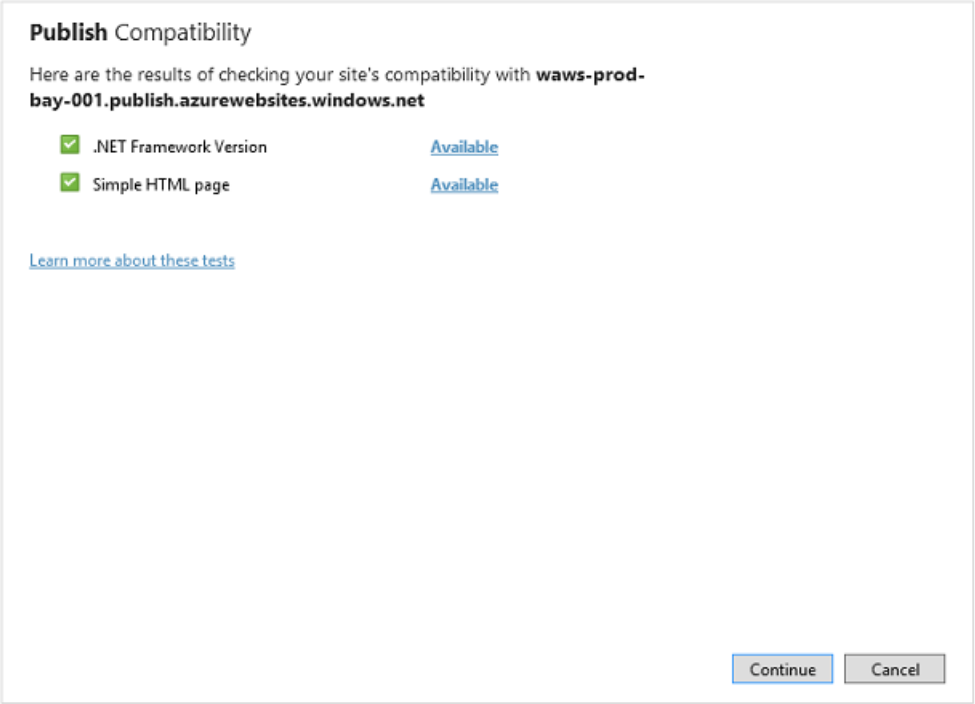
Click **OK**.

WebMatrix performss a test to determine if the server is compatible with your site.



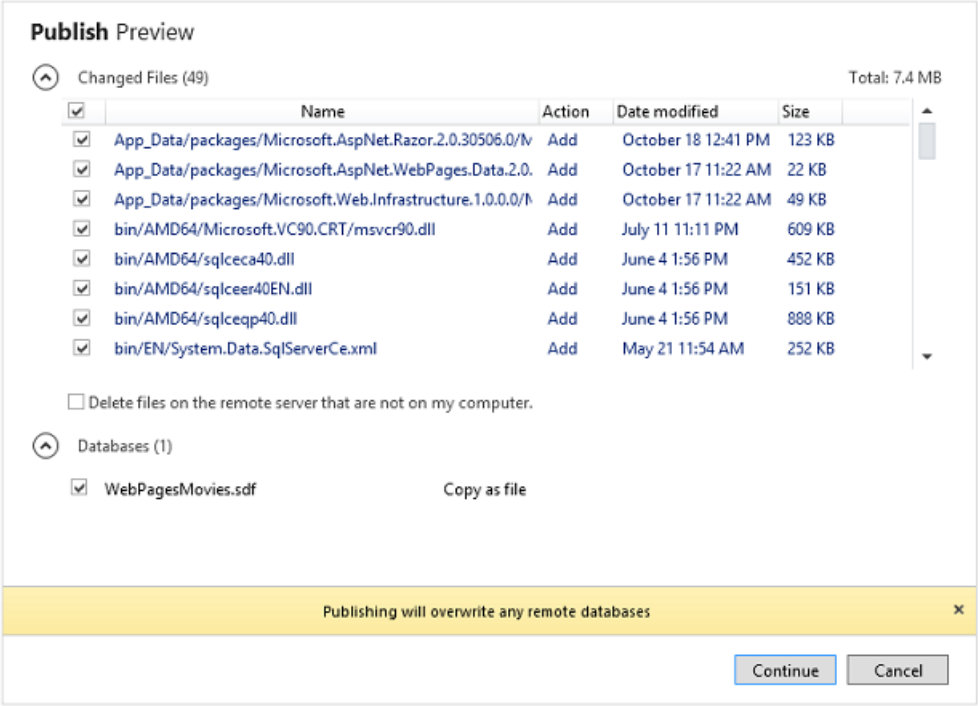
Select **Continue**.

The results of the compatibility test are displayed.



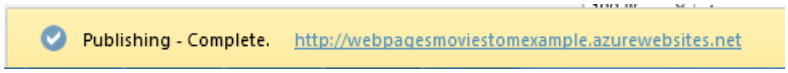
Select **Continue**.

WebMatrix displays the files and databases that will be published to the site. Since this is the first time you are publishing the site, all of the files are listed. You can uncheck a file that is not ready to be published. In the subsequent publications, only the files that have changed will be displayed. See Updating the Live Site: Republishing.

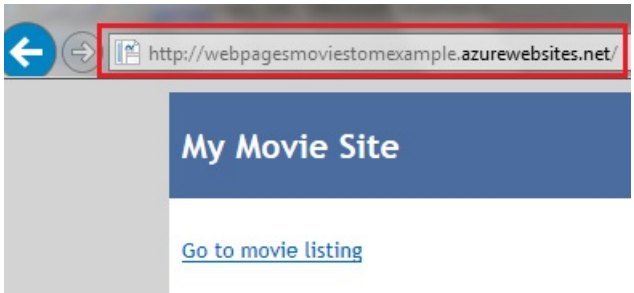


Select **Continue**.

After the site has been deployed to Azure, a message is displayed that indicates the deployment has completed.



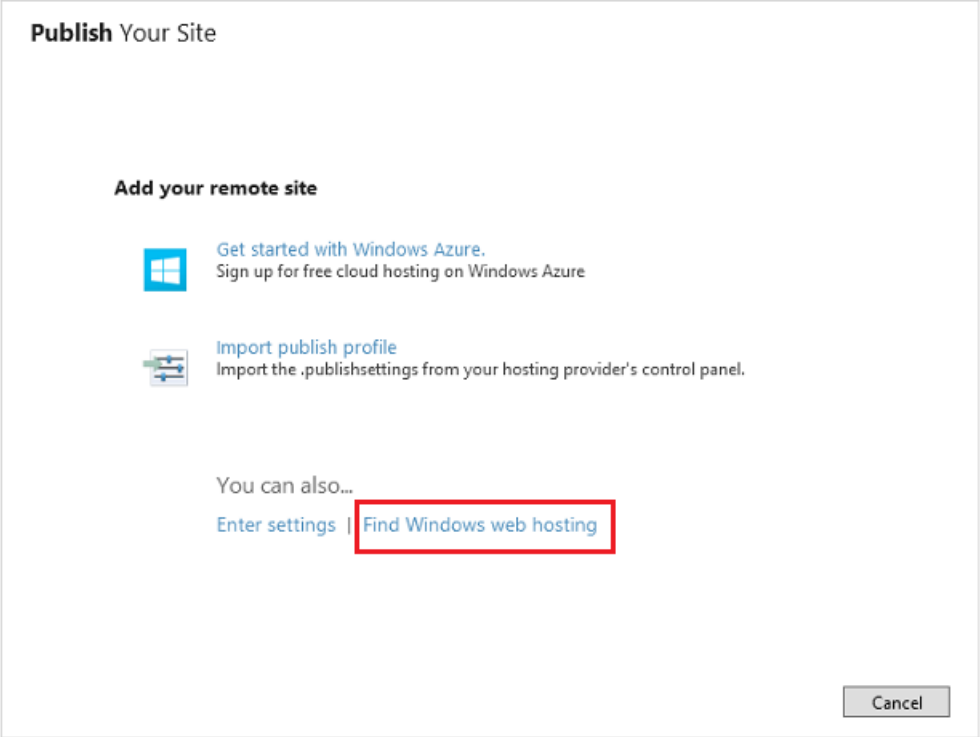
Your site and database have been published to Azure, and are now available to the public. Click the link in the message indicating publishing has completed, and you will now see your deployed site. You or anyone with Internet access can add or modify records in the database.



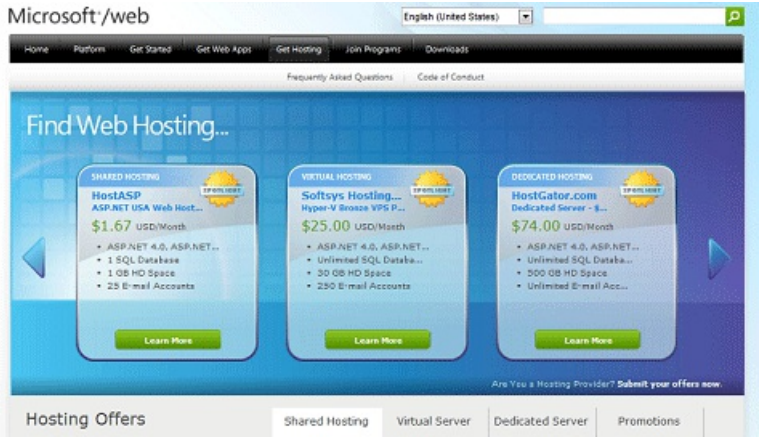
Publishing Your Site to a Web Hosting Company

If you decide to not publish to Azure, you can instead publish your site to a web hosting company.

Click the **Find web hosting** link.



You go to a page on the Microsoft site that lists hosting providers that support ASP.NET.



Obviously, it can be difficult to know now exactly what hosting features you might require over the long term. Here are a couple of things to consider:

- For purposes of the WebPagesMovies site, you don't have to have a separate add-on for SQL Server, which often costs extra. In your site, you're using SQL Server Compact Edition, which is self-contained. However, you might need SQL Server access for some future website work you do. If you think you might, make sure that you can add SQL Server capability later.
- Check whether the hosting provider supports the Web Deploy publishing protocol. You can publish by using FTP protocol, but it's more convenient to use Web Deploy.

Some sites offer a free trial period. A free trial is a good way to try publishing and hosting while you're still experimenting with WebMatrix and ASP.NET Web Pages.

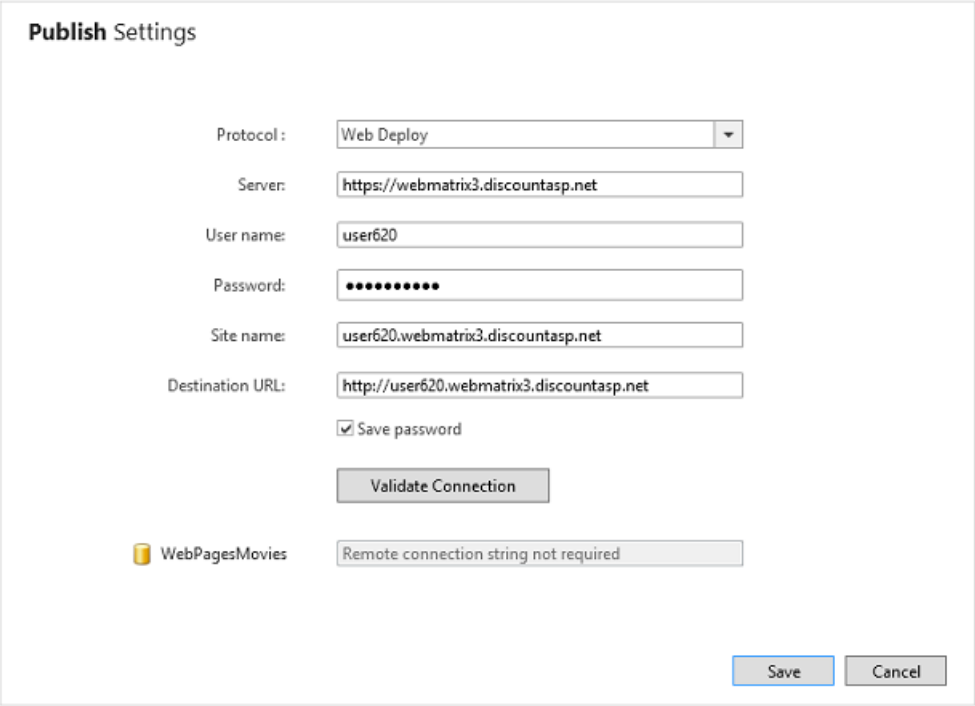
Pick one that you like. For this tutorial, we selected DiscountASP.NET, because while we were creating the tutorial, that company had a promotion that let people host a site free for a few months.

> **NOTE**
> Our choice of a hosting provider for this tutorial shouldn't be interpreted as an endorsement of that company over any other. But we had to pick one for illustration, and DiscountASP.NET is one of the many companies that supports ASP.NET Web Pages and the Web Deploy protocol for publishing.

Typically, after you've signed up with the hosting provider, the company sends you an email that contains a user name and password, the URL of the web server, and so on. If the hosting company supports Web Deploy protocol, they might send you a file that contains publish settings, or let you download one. A publish settings file simplifies the process for you.

When you've signed up and are ready to publish, click the **Publish** button in the WebMatrix ribbon. The **Publish Settings** dialog box is displayed.

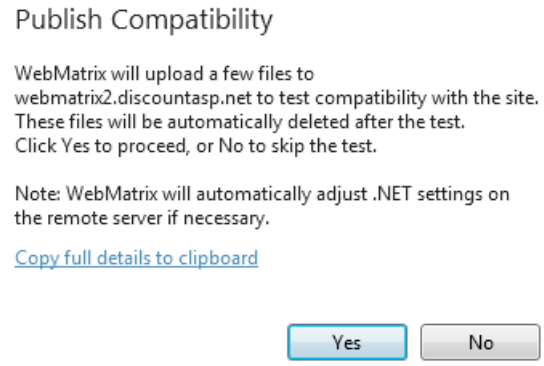
If the hosting provider sent you a publish settings file, click the **Import publish settings** link and import the file. If you don't have a publish settings file, fill in the fields by using the values that the hosting company sent you in email. Here's what the **Publish Settings** dialog box might look like when you're done:



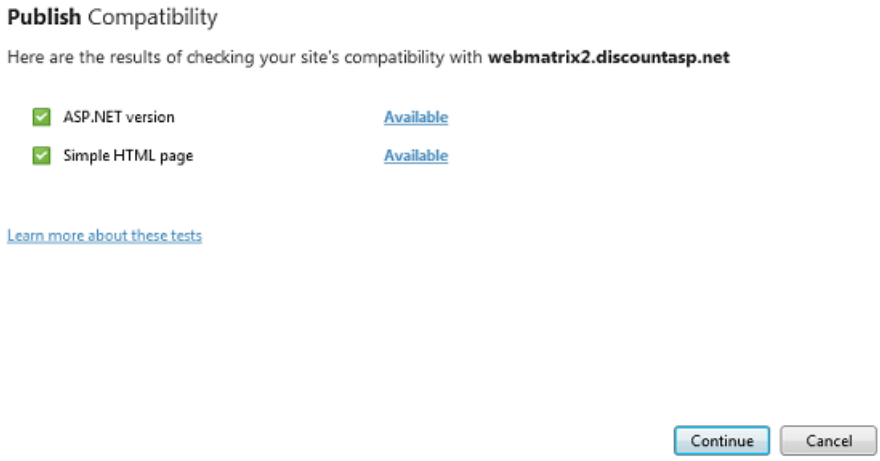
Click **Validate Connection**. If everything is ok, the dialog box reports **Connected successfully**, which means it can communicate with the hosting provider's server.

If there's a problem, WebMatrix does its best to tell you what the problem is:

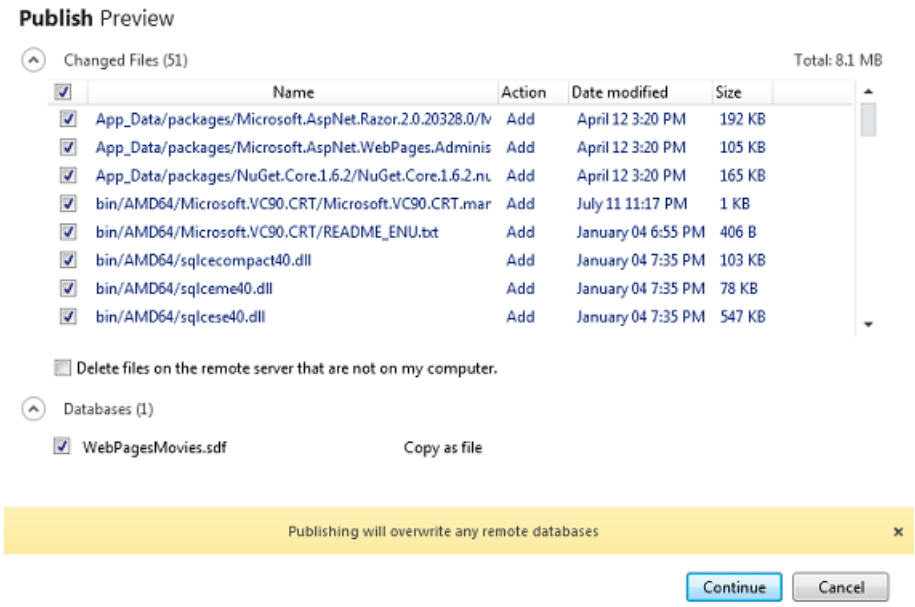
Click **Save** to save your settings. WebMatrix offers to perform a test to make sure that it can communicate correctly with the hosting site:



Click **Yes**. WebMatrix uploads some sample files to the hosting provider. When the compatibility test is done, WebMatrix reports the results:

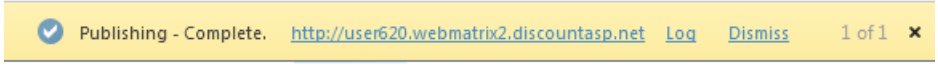


If you're ready to go, go ahead and click **Continue** to start the publish process for real. WebMatrix figures out what files are in your site and are already on the host server (right now, none) and gives you a preview of the publish process:

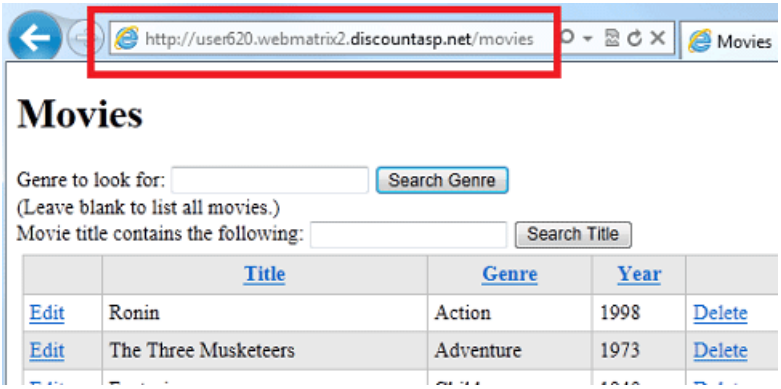


The list of files to publish includes the web pages that you've created like *Movies.cshtml*. The list also includes files for helpers that you've installed, the files to run SQL Server Compact Edition for your database, and so on. As a result, the initial publish process can be substantial.

Click **Continue**. WebMatrix copies your files to the hosting provider's server. When it's done, the results are reported in the status bar:



To see your live site, click the link in the status bar. Add *Movies* to the URL, and you'll see the *Movies.cshtml* file that you created:

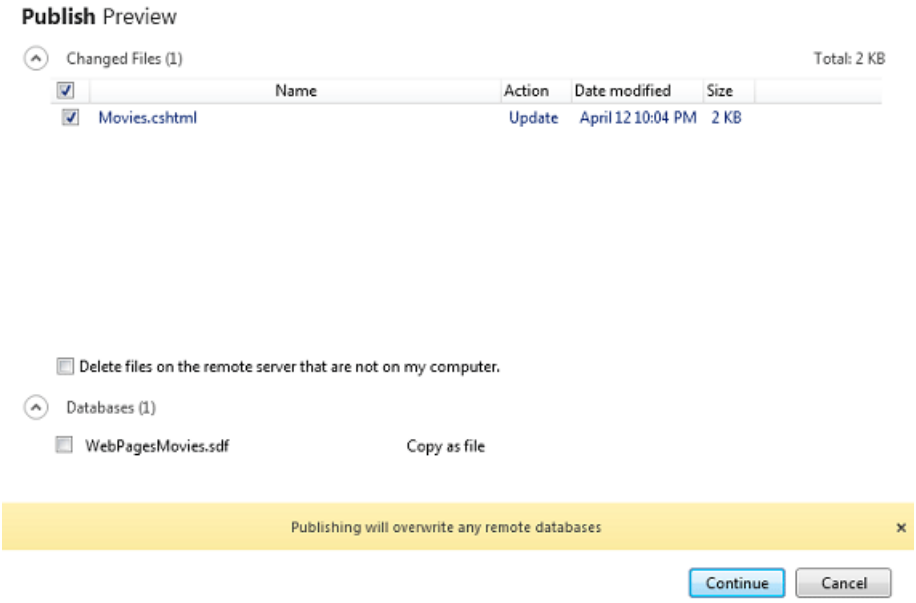


Updating the Live Site: Republishing

Once you've published your site (to either Azure or a web hosting company), there are two copies of it — the version on your computer and the version on the service provider. You'll probably want to continue developing the site (if nothing else, as part of the next tutorial set). When you do, you have to republish your site in order to copy changes from your computer to the service provider. The publish process in WebMatrix can determine what files have changed on your site and publish just those files.

To see how republishing works, open the *Movies.cshtml* site, make some small change, and then save the file. For example, change the title to `Movies - Updated`.

Click the **Publish** button in the ribbon. WebMatrix determines what's changed and shows you a preview of the files it will publish.



IMPORTANT

By default, WebMatrix publishes your database (*.sdf* file) only the first time you publish the site. Once your site is published and people are interacting with the website, the database on the live site typically has the site's real data. You have to be very careful not to overwrite the live database with the *.sdf* file that's on your computer, which usually contains only test data. That's why you see the warning **Publishing will overwrite any remote databases**, and why the check box for *WebPagesMovies.sdf* is cleared by default.

Click **Continue**. WebMatrix publishes the changed files and shows you a success message, like it did the first time you published.

Go to the live site (you can click the link in the success message if it's still showing) and verify that your change has been published.

> **TIP**
>
> **Editing Files Remotely**
>
> As an alternative to changing your site and then republishing, you can edit remote files directly in WebMatrix. In this scenario, you open a file that's on the service provider, and WebMatrix downloads a copy of it for you to edit. Every time you save the file, WebMatrix sends the changes to the site.
>
> Remote editing is an easy way to make changes to your live site. However, the changes you make this way aren't synchronized with the files in your local site. To synchronize the local files with the remote site, you can download the remote files. This process works much like publishing, except in reverse.
>
> We won't describe more about the remote-editing and remote-download facilities of WebMatrix here. They're quite useful if multiple people have to work on the same site on different computers. For more information, see Publish and Edit a Remote Site with WebMatrix 2 Beta.

Additional Resources

- ASP.NET WebMatrix ASP.NET Web Pages forum, a great place to post questions and get answers.

PREVIOUS

Routing

2/20/2020 • 2 minutes to read • Edit Online

How to customize the URLs used by a Web Pages application.

- Creating Readable URLs in ASP.NET Web Pages Sites

Creating Readable URLs in ASP.NET Web Pages (Razor) Sites

2/20/2020 • 3 minutes to read • Edit Online

by Tom FitzMacken

This article describes routing in an ASP.NET Web Pages (Razor) website, and how this lets you use URLs that are more readable and better for SEO.

What you'll learn:

- How ASP.NET uses routing to let you use more readable and searchable URLs.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 3

This tutorial also works with ASP.NET Web Pages 2.

About Routing

The URLs for the pages in your site can have an impact on how well the site works. A URL that's "friendly" can make it easier for people to use the site. It can also help with search-engine optimization (SEO) for the site. ASP.NET websites include the ability to use friendly URLs automatically.

ASP.NET lets you create meaningful URLs that describe user actions instead of just pointing to a file on the server. Consider these URLs for a fictional blog:

- `http://www.contoso.com/Blog/blog.cshtml?categories=hardware`
- `http://www.contoso.com//Blog/blog.cshtml?startdate=2009-11-01&enddate=2009-11-30`

Compare those URLs to the following ones:

- `http://www.contoso.com/Blog/categories/hardware/`
- `http://www.contoso.com/Blog/2009/November`

In the first pair, a user would have to know that the blog is displayed using the *blog.cshtml* page, and would then have to construct a query string that gets the right category or date range. The second set of examples is much easier to comprehend and create.

The URLs for the first example also point directly to a specific file (*blog.cshtml*). If for some reason the blog were moved to another folder on the server, or if the blog were rewritten to use a different page, the links would be wrong. The second set of URLs doesn't point to a specific page, so even if the blog implementation or location changes, the URLs would still be valid.

In ASP.NET Web Pages, you can create friendlier URLs like those in the above examples because ASP.NET uses *routing*. Routing creates logical mapping from a URL to a page (or pages) that can fulfill the request. Because the mapping is logical (not physical, to a specific file), routing provides great flexibility in how you define the URLs for your site.

How Routing Works

When ASP.NET processes a request, it reads the URL to determine how to route it. ASP.NET tries to match individual segments of the URL to files on disk, going from left to right. If there's a match, anything remaining in the URL is passed to the page as *path information*.

Imagine that someone makes a request using this URL:

`http://www.contoso.com/a/b/c`

The search goes like this:

1. Is there a file with the path and name of */a/b/c.cshtml*? If so, run that page and pass no information to it. Otherwise ...
2. Is there a file with the path and name of */a/b.cshtml*? If so, run that page and pass the value `c` to it. Otherwise ...
3. Is there a file with the path and name of */a.cshtml*? If so, run that page and pass the value `b/c` to it.

If the search found no exact matches for *.cshtml* files in their specified folders, ASP.NET continues looking for these files in turn:

1. */a/b/c/default.cshtml* (no path information).
2. */a/b/c/index.cshtml* (no path information).

> **NOTE**
> To be clear, requests for specific pages (that is, requests that include the *.cshtml* filename extension) work just like you'd expect. A request like `http://www.contoso.com/a/b.cshtml` will run the page *b.cshtml* just fine.

Inside a page, you can get the path information via the page's `UrlData` property, which is a dictionary. Imagine that you have a file named *ViewCustomers.cshtml* and your site gets this request:

`http://mysite.com/myWebSite/ViewCustomers/1000`

As described in the rules above, the request will go to your page. Inside the page, you can use code like the following to get and display the path information (in this case, the value "1000"):

```
<!DOCTYPE html>
<html>
    <head>
        <title>URLData</title>
    </head>
    <body>
      Customer ID: @UrlData[0].ToString()
    </body>
</html>
```

> **NOTE**
> Because routing doesn't involve complete file names, there can be ambiguity if you have pages that have the same name but different file-name extensions (for example, *MyPage.cshtml* and *MyPage.html*). In order to avoid problems with routing, it's best to make sure that you don't have pages in your site whose names differ only in their extension.

Additional Resources

WebMatrix - URLs, UrlData and Routing for SEO. This blog entry by Mike Brind provides some additional details on how routing works in ASP.NET Web Pages.

UI, Layouts, and Themes

2/20/2020 • 2 minutes to read • Edit Online

How to customize the site UI in an ASP.NET Web Pages application.

- Working with Forms
- Validating User Input in ASP.NET Web Pages Sites
- Creating a Consistent Look
- Customizing Site-Wide Behavior
- Working with Images
- Working with Video
- Creating and Using a Helper in an ASP.NET Web Pages Site
- Installing a Helper
- Twitter Helper with ASP.NET Web Pages
- Displaying Maps in an ASP.NET Web Pages Site

Working with HTML Forms in ASP.NET Web Pages (Razor) Sites

2/20/2020 • 10 minutes to read • Edit Online

by Tom FitzMacken

This article describes how to process an HTML form (with text boxes and buttons) when you are working in an ASP.NET Web Pages (Razor) website.

What you'll learn:

- How to create an HTML form.
- How to read user input from the form.
- How to validate user input.
- How to restore form values after the page is submitted.

These are the ASP.NET programming concepts introduced in the article:

- The `Request` object.
- Input validation.
- HTML encoding.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 3

This tutorial also works with ASP.NET Web Pages 2.

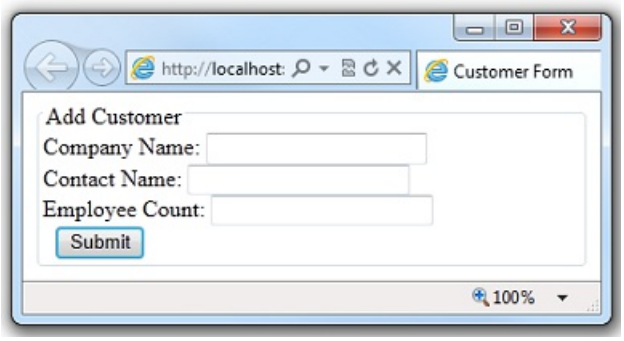
Creating a Simple HTML Form

1. Create a new website.

2. In the root folder, create a web page named *Form.cshtml* and enter the following markup:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Customer Form</title>
    </head>
    <body>
      <form method="post" >
        <fieldset>
          <legend>Add Customer</legend>
          <div>
            <label for="CompanyName">Company Name:</label>
            <input type="text" name="CompanyName" value="" />
          </div>
          <div>
            <label for="ContactName">Contact Name:</label>
            <input type="text" name="ContactName" value="" />
          </div>
          <div>
            <label for="Employees">Employee Count:</label>
            <input type="text" name="Employees" value="" />
          </div>
          <div>
            <label> </label>
            <input type="submit" value="Submit" class="submit" />
          </div>
        </fieldset>
      </form>
    </body>
</html>
```

3. Launch the page in your browser. (In WebMatrix, in the **Files** workspace, right-click the file and then select **Launch in browser**.) A simple form with three input fields and a **Submit** button is displayed.



At this point, if you click the **Submit** button, nothing happens. To make the form useful, you have to add some code that will run on the server.

Reading User Input from the Form

To process the form, you add code that reads the submitted field values and does something with them. This procedure shows you how to read the fields and display the user input on the page. (In a production application, you generally do more interesting things with user input. You'll do that in the article about working with databases.)

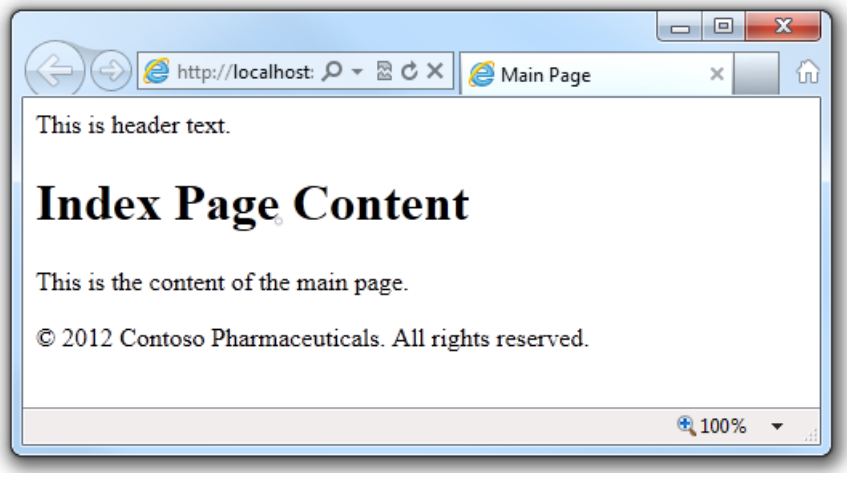
1. At the top of the *Form.cshtml* file, enter the following code:

```
@{
    if (IsPost) {
        string companyname = Request.Form["companyname"];
        string contactname = Request.Form["contactname"];
        int employeecount = Request.Form["employees"].AsInt();

        <text>
          You entered: <br />
          Company Name: @companyname <br />
          Contact Name: @contactname <br />
          Employee Count: @employeecount <br />
        </text>
    }
}
```

When the user first requests the page, only the empty form is displayed. The user (which will be you) fills in the form and then clicks **Submit**. This submits (posts) the user input to the server. By default, the request goes to the same page (namely, *Form.cshtml*).

When you submit the page this time, the values you entered are displayed just above the form:



Look at the code for the page. You first use the `IsPost` method to determine whether the page is being posted — that is, whether a user clicked the **Submit** button. If this is a post, `IsPost` returns true. This is the standard way in ASP.NET Web Pages to determine whether you're working with an initial request (a GET request) or a postback (a POST request). (For more information about GET and POST, see the sidebar "HTTP GET and POST and the IsPost Property" in Introduction to ASP.NET Web Pages Programming Using the Razor Syntax.)

Next, you get the values that the user filled in from the `Request.Form` object, and you put them in variables for later. The `Request.Form` object contains all the values that were submitted with the page, each identified by a key. The key is the equivalent to the `name` attribute of the form field that you want to read. For example, to read the `companyname` field (text box), you use `Request.Form["companyname"]`.

Form values are stored in the `Request.Form` object as strings. Therefore, when you have to work with a value as a number or a date or some other type, you have to convert it from a string to that type. In the example, the `AsInt` method of the `Request.Form` is used to convert the value of the employees field (which contains an employee count) to an integer.

2. Launch the page in your browser, fill in the form fields, and click **Submit**. The page displays the values you entered.

TIP

HTML Encoding for Appearance and Security

HTML has special uses for characters like `<`, `>`, and `&`. If these special characters appear where they're not expected, they can ruin the appearance and functionality of your web page. For example, the browser interprets the `<` character (unless it's followed by a space) as the beginning of an HTML element, like `<b>` or `<input ...>`. If the browser doesn't recognize the element, it simply discards the string that begins with `<` until it reaches something that it again recognizes. Obviously, this can result in some weird rendering in the page.

HTML encoding replaces these reserved characters with a code that browsers interpret as the correct symbol. For example, the `<` character is replaced with `<` and the `>` character is replaced with `>`. The browser renders these replacement strings as the characters that you want to see.

It's a good idea to use HTML encoding any time you display strings (input) that you got from a user. If you don't, a user can try to get your web page to run a malicious script or do something else that compromises your site security or that's just not what you intend. (This is particularly important if you take user input, store it someplace, and then display it later — for example, as a blog comment, user review, or something like that.)

To help prevent these problems, ASP.NET Web Pages automatically HTML-encodes any text content that you output from your code. For example, when you display the content of a variable or an expression using code such as `@MyVar`, ASP.NET Web Pages automatically encodes the output.

Validating User Input

Users make mistakes. You ask them to fill in a field, and they forget to, or you ask them to enter the number of employees and they type a name instead. To make sure that a form has been filled in correctly before you process it, you validate the user's input.

This procedure shows how to validate all three form fields to make sure the user didn't leave them blank. You also check that the employee count value is a number. If there are errors, you'll display an error message that tells the user what values didn't pass validation.

1. In the *Form.cshtml* file, replace the first block of code with the following code:

```
@{
    Validation.RequireField("companyname", "Company name is required.");
    Validation.RequireField("contactname", "Contact name is required.");
    Validation.RequireField("employees", "Employee count is required.");
    Validation.Add("employees", Validator.Integer());

    if (IsPost)  {
        var companyname = Request.Form["companyname"];
        var contactname = Request.Form["contactname"];
        var employeecount = Request.Form["employees"];
        if(Validation.IsValid()){
            <text>
              You entered: <br />
              Company Name: @companyname <br />
              Contact Name: @contactname <br />
              Employee Count: @employeecount <br />
            </text>
        }
    }
}
```

   To validate user input, you use the `Validation` helper. You register required fields by calling `Validation.RequireField`. You register other types of validation by calling `Validation.Add` and specifying the field to validate and the type of validation to perform.

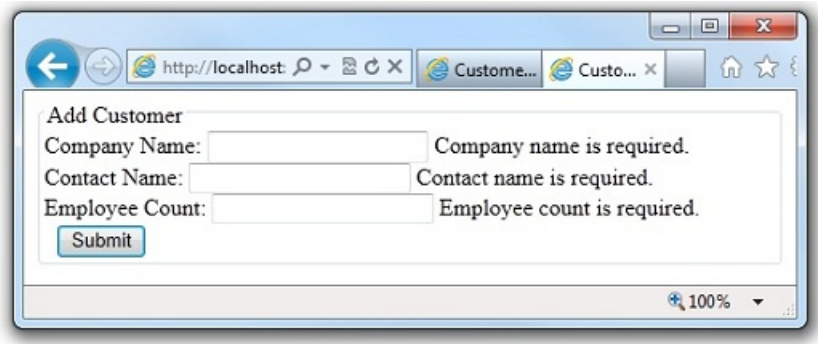
When the page runs, ASP.NET does all the validation for you. You can check the results by calling `Validation.IsValid`, which returns true if everything passed and false if any field failed validation. Typically, you call `Validation.IsValid` before you perform any processing on the user input.

2. Update the `<body>` element by adding three calls to the `Html.ValidationMessage` method, like this:

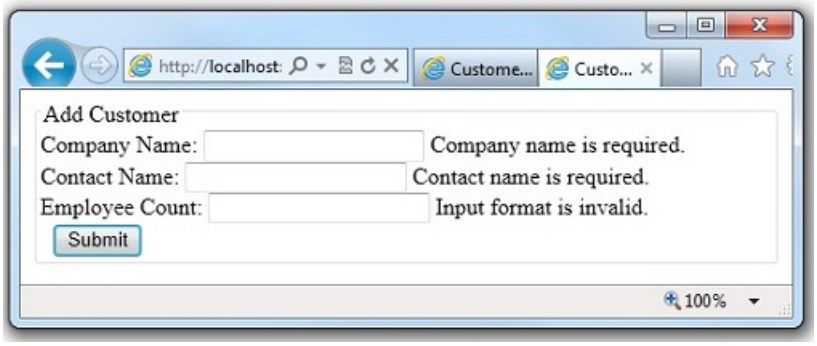
```
<body>
  <form method="post">
    <fieldset>
      <legend>Add Customer</legend>
      <div>
         <label for="CompanyName">Company Name:</label>
         <input type="text" name="CompanyName" value="" />
         @Html.ValidationMessage("companyname")
      </div>
      <div>
         <label for="ContactName">Contact Name:</label>
         <input type="text" name="ContactName" value="" />
         @Html.ValidationMessage("contactname")
      </div>
      <div>
         <label for="Employees">Employee Count:</label>
         <input type="text" name="Employees" value="" />
         @Html.ValidationMessage("employees")
      </div>
      <div>
         <label> </label>
         <input type="submit" value="Submit" class="submit" />
      </div>
    </fieldset>
  </form>
</body>
```

To display validation error messages, you can call Html.`ValidationMessage` and pass it the name of the field that you want the message for.

3. Run the page. Leave the fields blank and click **Submit**. You see error messages.



4. Add a string (for example, "ABC") to the **Employee Count** field and click **Submit** again. This time you see an error that indicates that the string isn't in the right format, namely, an integer.



ASP.NET Web Pages provides more options for validating user input, including the ability to automatically perform validation using client script, so that users get immediate feedback in the browser. See the Additional Resources section later for more information.

Restoring Form Values After Postbacks

When you tested the page in the previous section, you might have noticed that if you had a validation error, everything you entered (not just the invalid data) was gone, and you had to re-enter values for all the fields. This illustrates an important point: when you submit a page, process it, and then render the page again, the page is re-created from scratch. As you saw, this means that any values that were in the page when it was submitted are lost.

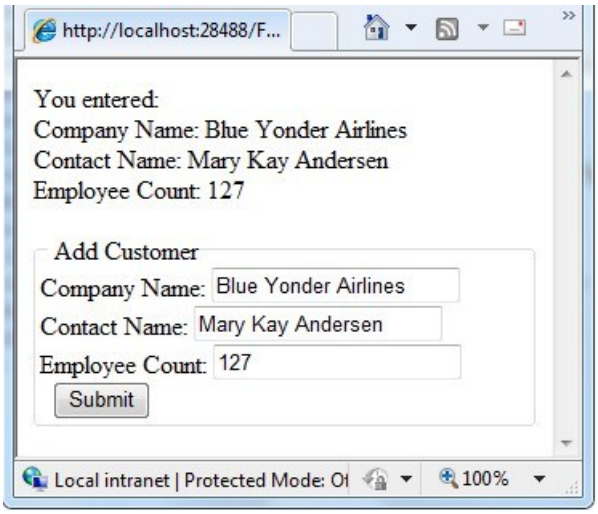
You can fix this easily, however. You have access to the values that were submitted (in the `Request.Form` object, so you can fill those values back into the form fields when the page is rendered.

1. In the *Form.cshtml* file, replace the `value` attributes of the `<input>` elements using the `value` attribute.:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Customer Form</title>
  </head>
  <body>
    <form method="post">
      <fieldset>
        <legend>Add Customer</legend>
        <div>
          <label for="CompanyName">Company Name:</label>
          <input type="text" name="CompanyName"
                 value="@Request.Form["companyname"]"  />
          @Html.ValidationMessage("companyname")
        </div>
        <div>
         <label for="ContactName">Contact Name:</label>
         <input type="text" name="ContactName"
                value="@Request.Form["contactname"]" />
         @Html.ValidationMessage("contactname")
        </div>
        <div>
         <label for="Employees">Employee Count:</label>
         <input type="text" name="Employees"
                value="@Request.Form["employees"]" />
         @Html.ValidationMessage("employees")
        </div>
        <div>
          <label> </label>
          <input type="submit" value="Submit" class="submit" />
        </div>
    </fieldset>
  </form>
</body>
```

   The `value` attribute of the `<input>` elements has been set to dynamically read the field value out of the `Request.Form` object. The first time that the page is requested, the values in the `Request.Form` object are all empty. This is fine, because that way the form is blank.

2. Launch the page in your browser, fill in the form fields or leave them blank, and click **Submit**. A page that shows the submitted values is displayed.



Additional Resources

- 1,001 Ways to Get Input from Web Users
- Using Forms and Processing User Input
- Validating User Input in ASP.NET Web Pages Sites
- Using AutoComplete in HTML Forms

Validating User Input in ASP.NET Web Pages (Razor) Sites

2/20/2020 • 10 minutes to read • Edit Online

by Tom FitzMacken

This article discusses how to validate information you get from users — that is, to make sure that users enter valid information in HTML forms in an ASP.NET Web Pages (Razor) site.

What you'll learn:

- How to check that a user's input matches validation criteria that you define.
- How to determine whether all validation tests have passed.
- How to display validation errors (and how to format them).
- How to validate data that doesn't come directly from users.

These are the ASP.NET programming concepts introduced in the article:

- The `Validation` helper.
- The `Html.ValidationSummary` and `Html.ValidationMessage` methods.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 3

This tutorial also works with ASP.NET Web Pages 2.

This article contains the following sections:

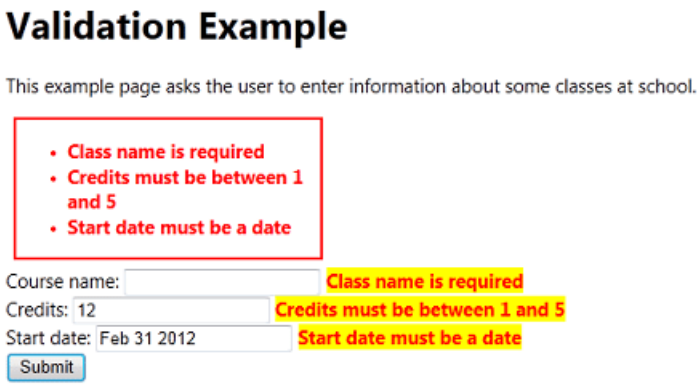
- Overview of User Input Validation
- Validating User Input
- Adding Client-Side Validation
- Formatting Validation Errors
- Validating Data That Doesn't Come Directly from Users

Overview of User Input Validation

If you ask users to enter information in a page — for example, into a form — it's important to make sure that the values that they enter are valid. For example, you don't want to process a form that's missing critical information.

When users enter values into an HTML form, the values that they enter are strings. In many cases, the values you need are some other data types, like integers or dates. Therefore, you also have to make sure that the values that users enter can be correctly converted to the appropriate data types.

You might also have certain restrictions on the values. Even if users correctly enter an integer, for example, you might need to make sure that the value falls within a certain range.



> **NOTE**
>
> **Important** Validating user input is also important for security. When you restrict the values that users can enter in forms, you reduce the chance that someone can enter a value that can compromise the security of your site.

Validating User Input

In ASP.NET Web Pages 2, you can use the `Validator` helper to test user input. The basic approach is to do the following:

1. Determine which input elements (fields) you want to validate.

   You typically validate values in `<input>` elements in a form. However, it's a good practice to validate all input, even input that comes from a constrained element like a `<select>` list. This helps to make sure that users don't bypass the controls on a page and submit a form.

2. In the page code, add individual validation checks for each input element by using methods of the `Validation` helper.

   To check for required fields, use `Validation.RequireField(field, [error message])` (for an individual field) or `Validation.RequireFields(field1, field2, ...))` (for a list of fields). For other types of validation, use `Validation.Add(field, ValidationType)`. For `ValidationType`, you can use these options:

   `Validator.DateTime ([error message])`
   `Validator.Decimal([error message])`
   `Validator.EqualsTo(otherField [, error message])`
   `Validator.Float([error message])`
   `Validator.Integer([error message])`
   `Validator.Range(min, max [, error message])`
   `Validator.RegEx(pattern [, error message])`
   `Validator.Required([error message])`
   `Validator.StringLength(length)`
   `Validator.Url([error message])`

3. When the page is submitted, check whether validation has passed by checking `Validation.IsValid`:

   ```
   if(IsPost && Validation.IsValid()){
       // Process form submit
   }
   ```

   If there are any validation errors, you skip normal page processing. For example, if the purpose of the page is to update a database, you don't do that until all validation errors have been fixed.

4. If there are validation errors, display error messages in the page's markup by using `Html.ValidationSummary` or `Html.ValidationMessage`, or both.

The following example shows a page that illustrates these steps.

```
@{
    var message="";
    // Specify validation requirements for different fields.
    Validation.RequireField("coursename", "Class name is required");
    Validation.RequireField("credits", "Credits is required");
    Validation.Add("coursename", Validator.StringLength(5));
    Validation.Add("credits", Validator.Integer("Credits must be an integer"));
    Validation.Add("credits", Validator.Range(1, 5, "Credits must be between 1 and 5"));
    Validation.Add("startDate", Validator.DateTime("Start date must be a date"));

    if (IsPost)  {
        // Before processing anything, make sure that all user input is valid.
        if (Validation.IsValid()) {
            var coursename = Request["coursename"];
            var credits = Request["credits"].AsInt();
            var startDate = Request["startDate"].AsDateTime();
            message += @"For Class, you entered " + coursename;
            message += @"<br/>For Credits, you entered " + credits.ToString();
            message += @"<br/>For Start Date, you entered " + startDate.ToString("dd-MMM-yyyy");

            // Further processing here
        }
    }
}
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Validation Example</title>
  <style>
      body {margin: 1in; font-family: 'Segoe UI'; font-size: 11pt; }
   </style>
</head>
<body>
  <h1>Validation Example</h1>
  <p>This example page asks the user to enter information about some classes at school.</p>
  <form method="post">
    @Html.ValidationSummary()
    <div>
      <label for="coursename">Course name: </label>
      <input type="text"
         name="coursename"
         value="@Request["coursename"]"
      />
      @Html.ValidationMessage("coursename")
    </div>

    <div>
      <label for="credits">Credits: </label>
      <input type="text"
         name="credits"
         value="@Request["credits"]"
      />
      @Html.ValidationMessage("credits")
    </div>

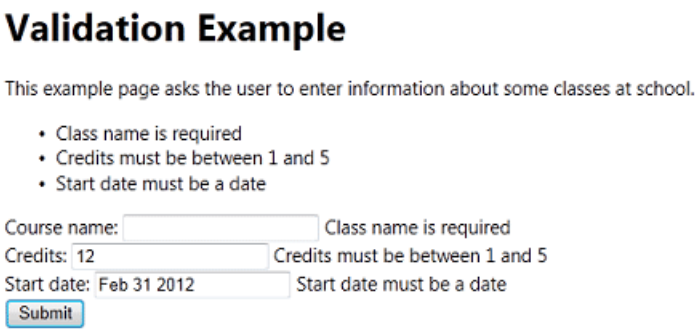
    <div>
      <label for="startDate">Start date: </label>
      <input type="text"
         name="startDate"
         value="@Request["startDate"]"
      />
      @Html.ValidationMessage("startDate")
```

```
    </div>

   <div>
      <input type="submit" value="Submit" class="submit" />
    </div>

    <div>
      @if(IsPost){
        <p>@Html.Raw(message)</p>
      }
    </div>
  </form>
</body>
</html>
```

To see how validation works, run this page and deliberately make mistakes. For example, here's what the page looks like if you forget to enter a course name, if you enter an, and if you enter an invalid date:



Adding Client-Side Validation

By default, user input is validated after users submit the page — that is, the validation is performed in server code. A disadvantage of this approach is that users don't know that they've made an error until after they submit the page. If a form is long or complex, reporting errors only after the page is submitted can be inconvenient to the user.

You can add support to perform validation in client script. In that case, the validation is performed as users work in the browser. For example, suppose you specify that a value should be an integer. If a user enters a non-integer value, the error is reported as soon as the user leaves the entry field. Users get immediate feedback, which is convenient for them. Client-based validation can also reduce the number of times that the user has to submit the form to correct multiple errors.

> **NOTE**
> Even if you use client-side validation, validation is always also performed in server code. Performing validation in server code is a security measure, in case users bypass client-based validation.

1. Register the following JavaScript libraries in the page:

```
<script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.2.js">
</script>
<script
src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8.1/jquery.validate.js">
</script>
<script src="~/Scripts/jquery.validate.unobtrusive.js">
</script>
```

Two of the libraries are loadable from a content delivery network (CDN), so you don't necessarily have to have them on your computer or server. However, you must have a local copy of *jquery.validate.unobtrusive.js*. If you are not already working with a WebMatrix template (like **Starter Site**) that includes the library, create a Web Pages site that's based on **Starter Site**. Then copy the *.js* file to your current site.

2. In markup, for each element that you're validating, add a call to `Validation.For(field)`. This method emits attributes that are used by client-side validation. (Rather than emitting actual JavaScript code, the method emits attributes like `data-val-...`. These attributes support unobtrusive client validation that uses jQuery to do the work.)

The following page shows how to add client validation features to the example shown earlier.

```
@{
    // Note that client validation as implemented here will work only with
    // ASP.NET Web Pages 2.

    var message="";
    // Specify validation requirements for different fields.
    Validation.RequireField("coursename", "Class name is required");
    Validation.RequireField("credits", "Credits is required");
    Validation.Add("coursename", Validator.StringLength(5));
    Validation.Add("credits", Validator.Integer("Credits must be an integer"));
    Validation.Add("credits", Validator.Range(1, 5, "Credits must be between 1 and 5"));
    Validation.Add("startDate", Validator.DateTime("Start date must be a date"));

    if (IsPost)  {
        // Before processing anything, make sure that all user input is valid.
        if (Validation.IsValid()) {
            var coursename = Request["coursename"];
            var credits = Request["credits"].AsInt();
            var startDate = Request["startDate"].AsDateTime();
            message += @"For Class, you entered " + coursename;
            message += @"<br/>For Credits, you entered " + credits.ToString();
            message += @"<br/>For Start Date, you entered " + startDate.ToString("dd-MMM-yyyy");

            // Further processing here
        }
    }
}
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Validation Example with Client Validation</title>
  <style>
      body {margin: 1in; font-family: 'Segoe UI'; font-size: 11pt; }
   </style>
    <script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.2.js"></script>
    <script
        src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8.1/jquery.validate.js">
    </script>
    <script src="~/Scripts/jquery.validate.unobtrusive.js"></script>
</head>
<body>
  <h1>Validation Example with Client Validation</h1>
  <p>This example page asks the user to enter information about some classes at school.</p>
  <form method="post">
    @Html.ValidationSummary()
    <div>
      <label for="coursename">Course name: </label>
      <input type="text"
         name="coursename"
         value="@Request["coursename"]"
         @Validation.For("coursename")
      />
```

```
      @Html.ValidationMessage("coursename")
    </div>

    <div>
      <label for="credits">Credits: </label>
      <input type="text"
         name="credits"
         value="@Request["credits"]"
         @Validation.For("credits")
      />
      @Html.ValidationMessage("credits")
    </div>

    <div>
      <label for="startDate">Start date: </label>
      <input type="text"
         name="startDate"
         value="@Request["startDate"]"
         @Validation.For("startDate")
      />
      @Html.ValidationMessage("startDate")
    </div>

   <div>
      <input type="submit" value="Submit" class="submit" />
    </div>

    <div>
      @if(IsPost){
        <p>@Html.Raw(message)</p>
      }
    </div>
  </form>
</body>
</html>
```

Not all validation checks run on the client. In particular, data-type validation (integer, date, and so on) don't run on the client. The following checks work on both the client and server:

- `Required`
- `Range(minValue, maxValue)`
- `StringLength(maxLength[, minLength])`
- `Regex(pattern)`
- `EqualsTo(otherField)`

In this example, the test for a valid date won't work in client code. However, the test will be performed in server code.

Formatting Validation Errors

You can control how validation errors are displayed by defining CSS classes that have the following reserved names:

- `field-validation-error`. Defines the output of the `Html.ValidationMessage` method when it's displaying an error.
- `field-validation-valid`. Defines the output of the `Html.ValidationMessage` method when there is no error.
- `input-validation-error`. Defines how `<input>` elements are rendered when there's an error. (For example, you can use this class to set the background color of an <input> element to a different color if its value is invalid.) This CSS class is used only during client validation (in ASP.NET Web Pages 2).
- `input-validation-valid`. Defines the appearance of `<input>` elements when there is no error.

- `validation-summary-errors`. Defines the output of the `Html.ValidationSummary` method it's displaying a list of errors.
- `validation-summary-valid`. Defines the output of the `Html.ValidationSummary` method when there is no error.

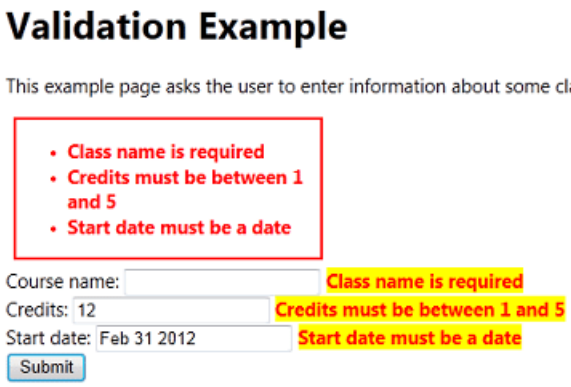
The following `<style>` block shows rules for error conditions.

```
<style>
.validation-summary-errors {
  border:2px solid red;
  color:red;
  font-weight:bold;
  margin:6px;
  width:30%;
}

.field-validation-error{
  color:red;
   font-weight:bold;
   background-color:yellow;
}

.input-validation-error{
  color:red;
  font-weight:bold;
  background-color:pink;
}
</style>
```

If you include this style block in the example pages from earlier in the article, the error display will look like the following illustration:



> **NOTE**
> If you're not using client validation in ASP.NET Web Pages 2, the CSS classes for the `<input>` elements (`input-validation-error` and `input-validation-valid` don't have any effect.

Static and Dynamic Error Display

The CSS rules come in pairs, such as `validation-summary-errors` and `validation-summary-valid`. These pairs let you define rules for both conditions: an error condition and a "normal" (non-error) condition. It's important to understand that the markup for the error display is always rendered, even if there are no errors. For example, if a page has an `Html.ValidationSummary` method in the markup, the page source will contain the following markup even when the page is requested for the first time:

```
<div class="validation-summary-valid" data-valmsg-summary="true"><ul></ul></div>
```

In other words, the `Html.ValidationSummary` method always renders a `<div>` element and a list, even if the error

list is empty. Similarly, the `Html.ValidationMessage` method always renders a `<span>` element as a placeholder for an individual field error, even if there is no error.

In some situations, displaying an error message can cause the page to reflow and can cause elements on the page to move around. The CSS rules that end in `-valid` let you define a layout that can help prevent this problem. For example, you can define `field-validation-error` and `field-validation-valid` to both have the same fixed size. That way, the display area for the field is static and won't change the page flow if an error message is displayed.

Validating Data That Doesn't Come Directly from Users

Sometimes you have to validate information that doesn't come directly from an HTML form. A typical example is a page where a value is passed in a query string, as in the following example:

`http://server/myapp/EditClassInformation?classid=1022`

In this case, you want to make sure that the value that's passed to the page (here, 1022 for the value of `classid`) is valid. You can't directly use the `Validation` helper to perform this validation. However, you can use other features of the validation system, like the ability to display validation error messages.

> **NOTE**
>
> **Important** Always validate values that you get from *any* source, including form-field values, query-string values, and cookie values. It's easy for people to change these values (perhaps for malicious purposes). So you must check these values in order to protect your application.

The following example shows how you might validate a value that's passed in a query string. The code tests that the value is not empty and that it's an integer.

```
if(!IsPost){
    if(!Request.QueryString["classid"].IsEmpty() && Request.QueryString["classid"].IsInt()) {
        // Process the value
    }
    else{
        Validation.AddFormError("No class was selected.");
    }
}
```

Notice that the test is performed when the request is not a form submission (`if(!IsPost)`). This test would pass the first time that the page is requested, but not when the request is a form submission.

To display this error, you can add the error to the list of validation errors by calling `Validation.AddFormError("message")`. If the page contains a call to the `Html.ValidationSummary` method, the error is displayed there, just like a user-input validation error.

Additional Resources

Working with HTML Forms in ASP.NET Web Pages Sites

Creating a Consistent Layout in ASP.NET Web Pages (Razor) Sites

2/20/2020 • 14 minutes to read • Edit Online

by Tom FitzMacken

This article explains how you can use layout pages in an ASP.NET Web Pages (Razor) website to create reusable blocks of content (like headers and footers) and to create a consistent look for all the pages in the site.

What you'll learn:

- How to create reusable blocks of content like headers and footers.
- How to create a consistent look for all the pages in your site using a layout.
- How to pass data at run time to a layout page.

These are the ASP.NET features introduced in the article:

- Content blocks, which are files that contain HTML-formatted content to be inserted in multiple pages.
- Layout pages, which are pages that contain HTML-formatted content that can be shared by pages on the website.
- The `RenderPage`, `RenderBody`, and `RenderSection` methods, which tell ASP.NET where to insert page elements.
- The `PageData` dictionary that lets you share data between content blocks and layout pages.

Software versions used in the tutorial

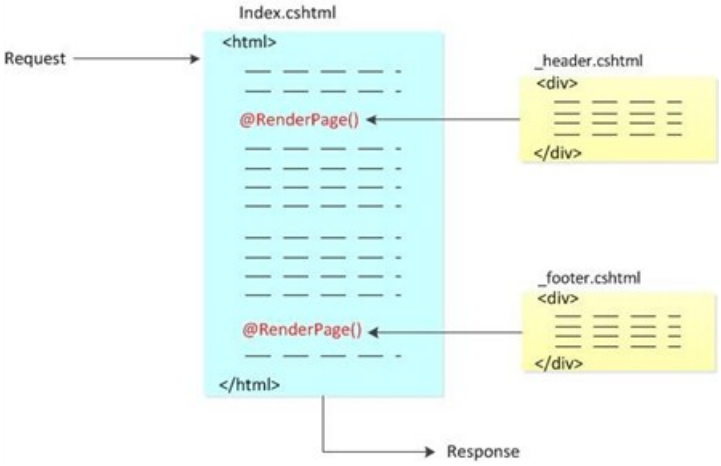
- ASP.NET Web Pages (Razor) 3

This tutorial also works with ASP.NET Web Pages 2.

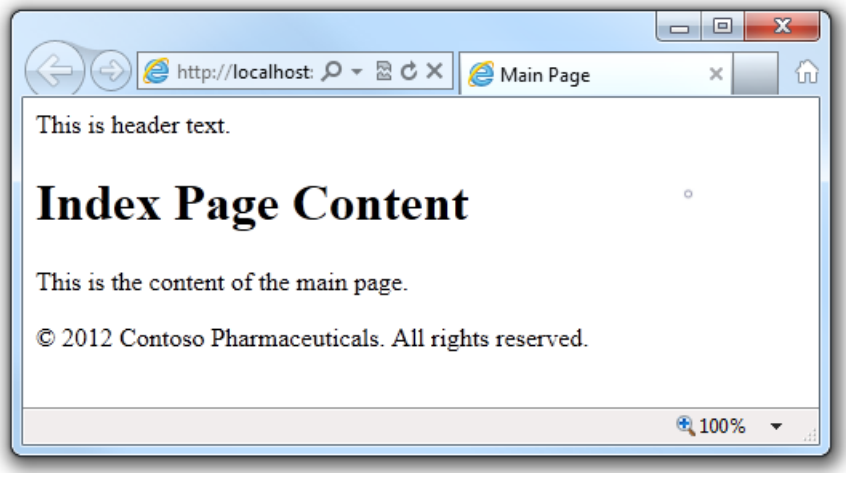
About Layout Pages

Many websites have content that's displayed on every page, like a header and footer, or a box that tells users that they're logged in. ASP.NET lets you create a separate file with a content block that can contain text, markup, and code, just like a regular web page. You can then insert the content block in other pages on the site where you want the information to appear. That way you don't have to copy and paste the same content into every page. Creating common content like this also makes it easier to update your site. If you need to change the content, you can just update a single file, and the changes are then reflected everywhere the content has been inserted.

The following diagram shows how content blocks work. When a browser requests a page from the web server, ASP.NET inserts the content blocks at the point where the `RenderPage` method is called in the main page. The finished (merged) page is then sent to the browser.



In this procedure, you'll create a page that references two content blocks (a header and a footer) that are located in separate files. You can use these same content blocks in any page in your site. When you're done, you'll get a page like this:



1. In the root folder of your website, create a file named *Index.cshtml*.

2. Replace the existing markup with the following:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Main Page</title>
  </head>
  <body>

    <h1>Index Page Content</h1>
    <p>This is the content of the main page.</p>

  </body>
</html>
```

3. In the root folder, create a folder named *Shared*.

> **NOTE**
>
> It's common practice to store files that are shared among web pages in a folder named *Shared*.

4. In the *Shared* folder, create a file named *_Header.cshtml*.

5. Replace any existing content with the following:

```
<div class="header">This is header text.</div>
```

Notice that the file name is *_Header.cshtml*, with an underscore (_) as a prefix. ASP.NET won't send a page to the browser if its name starts with an underscore. This prevents people from requesting (inadvertently or otherwise) these pages directly. It's a good idea to use an underscore to name pages that have content blocks in them, because you don't really want users to be able to request these pages — they exist strictly to be inserted into other pages.

6. In the *Shared* folder, create a file named *_Footer.cshtml* and replace the content with the following:

```
<div class="footer">&copy; 2012 Contoso Pharmaceuticals. All rights reserved.
</div>
```

7. In the *Index.cshtml* page, add two calls to the `RenderPage` method, as shown here:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Main Page</title>
  </head>
  <body>

    @RenderPage("~/Shared/_Header.cshtml")

    <h1>Index Page Content</h1>
    <p>This is the content of the main page.</p>

    @RenderPage("~/Shared/_Footer.cshtml")

  </body>
</html>
```

This shows how to insert a content block into a web page. You call the `RenderPage` method and pass it the name of the file whose contents you want to insert at that point. Here, you're inserting the contents of the *_Header.cshtml* and *_Footer.cshtml* files into the *Index.cshtml* file.

8. Run the *Index.cshtml* page in a browser. (In WebMatrix, in the **Files** workspace, right-click the file and then select **Launch in browser**.)

9. In the browser, view the page source. (For example, in Internet Explorer, right-click the page and then click **View Source**.)

   This lets you see the web page markup that's sent to the browser, which combines the index page markup with the content blocks. The following example shows the page source that's rendered for *Index.cshtml*. The calls to `RenderPage` that you inserted into *Index.cshtml* have been replaced with the actual contents of the header and footer files.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Main Page</title>
  </head>
  <body>

  <div class="header">
    This is header text.
  </div>

    <h1>Index Page Content</h1>
    <p>This is the content of the main page.</p>

  <div class="footer">
    &copy; 2012 Contoso Pharmaceuticals. All rights reserved.
  </div>

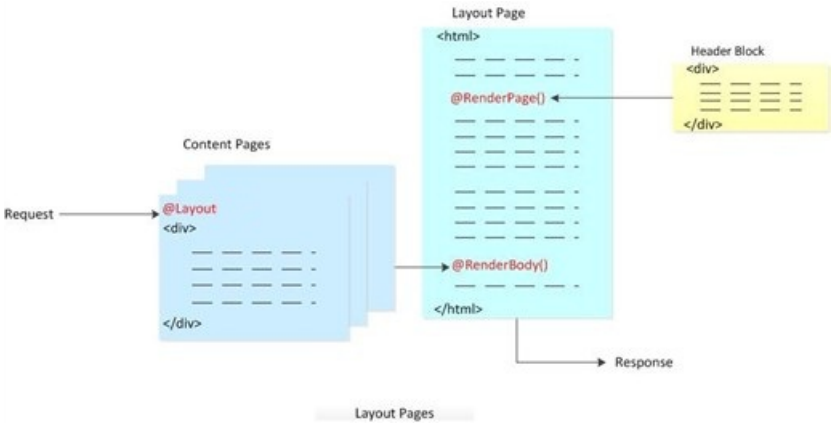
  </body>
</html>
```

Creating a Consistent Look Using Layout Pages

So far you've seen that it's easy to include the same content on multiple pages. A more structured approach to creating a consistent look for a site is to use layout pages. A layout page defines the structure of a web page, but doesn't contain any actual content. After you've created a layout page, you can create web pages that contain the content and then link them to the layout page. When these pages are displayed, they'll be formatted according to the layout page. (In this sense, a layout page acts as a kind of template for content that's defined in other pages.)

The layout page is just like any HTML page, except that it contains a call to the `RenderBody` method. The position of the `RenderBody` method in the layout page determines where the information from the content page will be included.

The following diagram shows how content pages and layout pages are combined at run time to produce the finished web page. The browser requests a content page. The content page has code in it that specifies the layout page to use for the page's structure. In the layout page, the content is inserted at the point where the `RenderBody` method is called. Content blocks can also be inserted into the layout page by calling the `RenderPage` method, the way you did in the previous section. When the web page is complete, it's sent to the browser.



The following procedure shows how to create a layout page and link content pages to it.

1. In the *Shared* folder of your website, create a file named *_Layout1.cshtml*.

2. Replace any existing content with the following:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Structured Content </title>
    <link href="~/Styles/Site.css" rel="stylesheet" type="text/css" />
  </head>
  <body>
    @RenderPage("~/Shared/_Header2.cshtml")
    <div id="main">
      @RenderBody()
    </div>
    <div id="footer">
      &copy; 2012 Contoso Pharmaceuticals. All rights reserved.
    </div>
  </body>
</html>
```

You use the `RenderPage` method in a layout page to insert content blocks. A layout page can contain only one call to the `RenderBody` method.

3. In the *Shared* folder, create a file named *_Header2.cshtml* and replace any existing content with the following:

```
<div id="header">Creating a Consistent Look</div>
```

4. In the root folder, create a new folder and name it *Styles*.

5. In the *Styles* folder, create a file named *Site.css* and add the following style definitions:

```
h1 {
    border-bottom: 3px solid #cc9900;
    font: 2.75em/1.75em Georgia, serif;
    color: #996600;
}

ul {
    list-style-type: none;
}

body {
    margin: 0;
    padding: 1em;
    background-color: #ffffff;
    font: 75%/1.75em "Trebuchet MS", Verdana, sans-serif;
    color: #006600;
}

#list {
    margin: 1em 0 7em -3em;
    padding: 1em 0 0 0;
    background-color: #ffffff;
    color: #996600;
    width: 25%;
    float: left;
}

#header, #footer {
    margin: 0;
    padding: 0;
    color: #996600;
}
```

These style definitions are here only to show how style sheets can be used with layout pages. If you want, you can define your own styles for these elements.

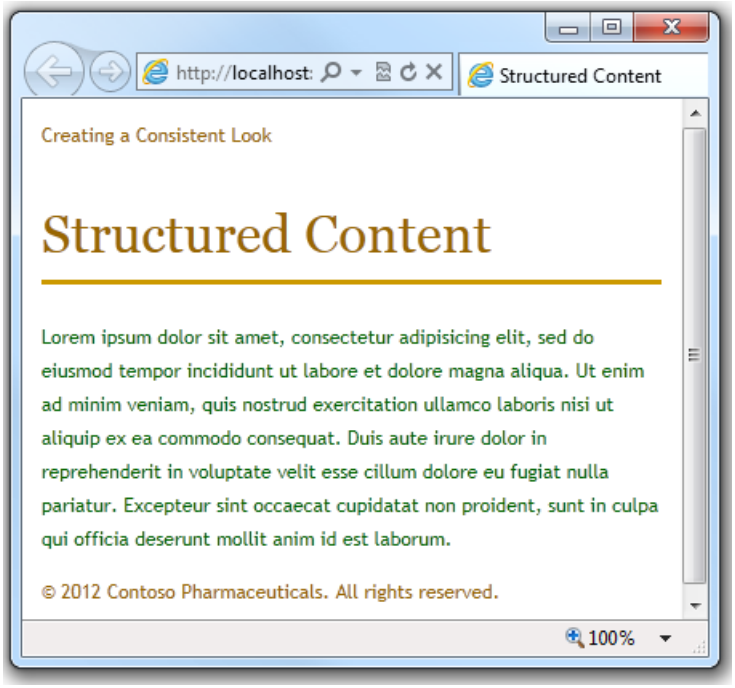
6. In the root folder, create a file named *Content1.cshtml* and replace any existing content with the following:

```
@{
    Layout = "~/Shared/_Layout1.cshtml";
}

<h1> Structured Content </h1>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum.</p>
```

This is a page that will use a layout page. The code block at the top of the page indicates which layout page to use to format this content.

7. Run *Content1.cshtml* in a browser. The rendered page uses the format and style sheet defined in *_Layout1.cshtml* and the text (content) defined in *Content1.cshtml*.



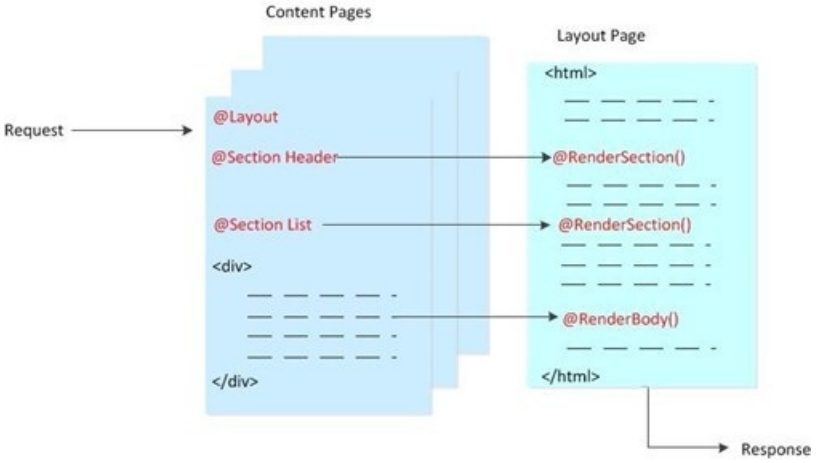
You can repeat step 6 to create additional content pages that can then share the same layout page.

> **NOTE**
> You can set up your site so that you can automatically use the same layout page for all the content pages in a folder. For details, see Customizing Site-Wide Behavior for ASP.NET Web Pages.

Designing Layout Pages That Have Multiple Content Sections

A content page can have multiple sections, which is useful if you want to use layouts that have multiple areas with replaceable content. In the content page, you give each section a unique name. (The default section is left unnamed.) In the layout page, you add a `RenderBody` method to specify where the unnamed (default) section should appear. You then add separate `RenderSection` methods in order to render named sections individually.

The following diagram shows how ASP.NET handles content that's divided into multiple sections. Each named section is contained in a section block in the content page. (They're named `Header` and `List` in the example.) The framework inserts content section into the layout page at the point where the `RenderSection` method is called. The unnamed (default) section is inserted at the point where the `RenderBody` method is called, as you saw earlier.



This procedure shows how to create a content page that has multiple content sections and how to render it using a layout page that supports multiple content sections.

1. In the *Shared* folder, create a file named *_Layout2.cshtml*.

2. Replace any existing content with the following:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Multisection Content</title>
    <link href="~/Styles/Site.css" rel="stylesheet" type="text/css" />
  </head>
  <body>
    <div id="header">
      @RenderSection("header")
    </div>
    <div id="list">
      @RenderSection("list")
    </div>
    <div id="main">
      @RenderBody()
    </div>
    <div id="footer">
      &copy; 2012 Contoso Pharmaceuticals. All rights reserved.
    </div>
  </body>
</html>
```

   You use the `RenderSection` method to render both the header and list sections.

3. In the root folder, create a file named *Content2.cshtml* and replace any existing content with the following:

```
@{
    Layout = "~/Shared/_Layout2.cshtml";
}

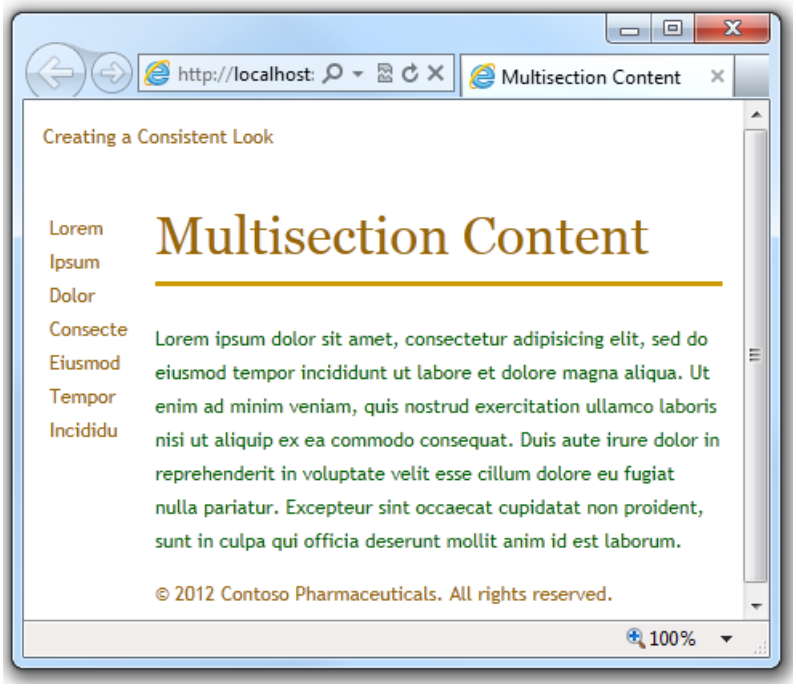
@section header {
    <div id="header">
        Creating a Consistent Look
    </div>
}

@section list {
    <ul>
        <li>Lorem</li>
        <li>Ipsum</li>
        <li>Dolor</li>
        <li>Consecte</li>
        <li>Eiusmod</li>
        <li>Tempor</li>
        <li>Incididu</li>
    </ul>
}

<h1>Multisection Content</h1>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum.</p>
```

This content page contains a code block at the top of the page. Each named section is contained in a section block. The rest of the page contains the default (unnamed) content section.

4. Run *Content2.cshtml* in a browser.



Making Content Sections Optional

Normally, the sections that you create in a content page have to match sections that are defined in the layout page. You can get errors if any of the following occur:

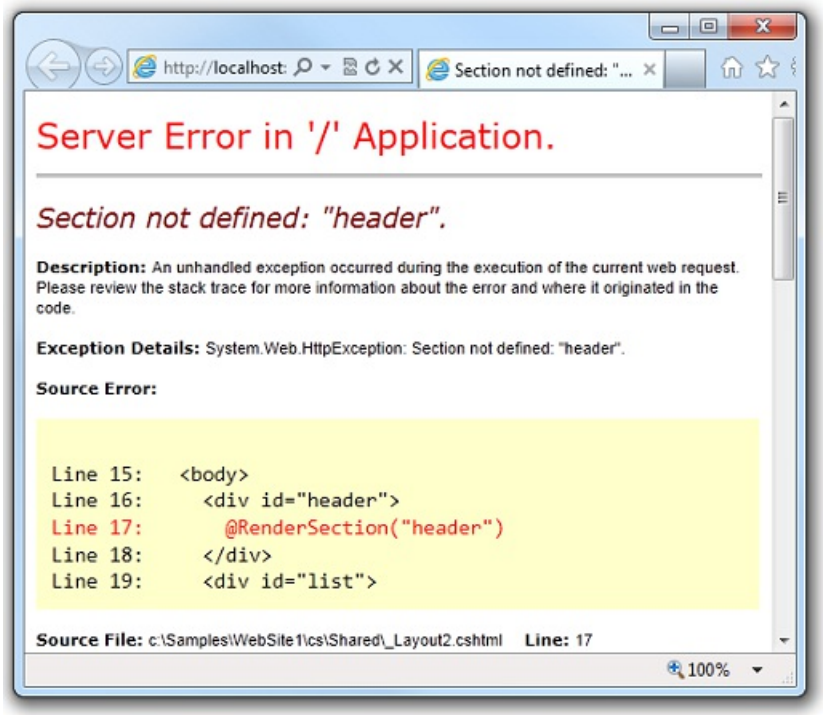
- The content page contains a section that has no corresponding section in the layout page.
- The layout page contains a section for which there's no content.
- The layout page includes method calls that try to render the same section more than once.

However, you can override this behavior for a named section by declaring the section to be optional in the layout page. This lets you define multiple content pages that can share a layout page but that might or might not have content for a specific section.

1. Open *Content2.cshtml* and remove the following section:

```
@section header {
  <div id="header">
    Creating a Consistent Look
  </div>
}
```

2. Save the page and then run it in a browser. An error message is displayed, because the content page doesn't provide content for a section defined in the layout page, namely the header section.



3. In the *Shared* folder, open the *_Layout2.cshtml* page and replace this line:

```
@RenderSection("header")
```

with the following code:

```
@RenderSection("header", required: false)
```

As an alternative, you could replace the previous line of code with the following code block, which produces the same results:

```
@if (IsSectionDefined("header")) {
    @RenderSection("header")
}
```

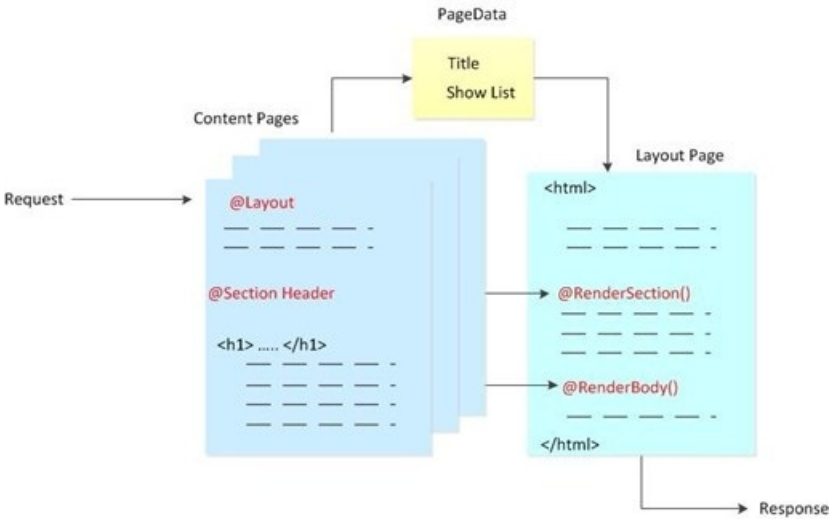
4. Run the *Content2.cshtml* page in a browser again. (If you still have this page open in the browser, you can just refresh it.) This time the page is displayed with no error, even though it has no header.

Passing Data to Layout Pages

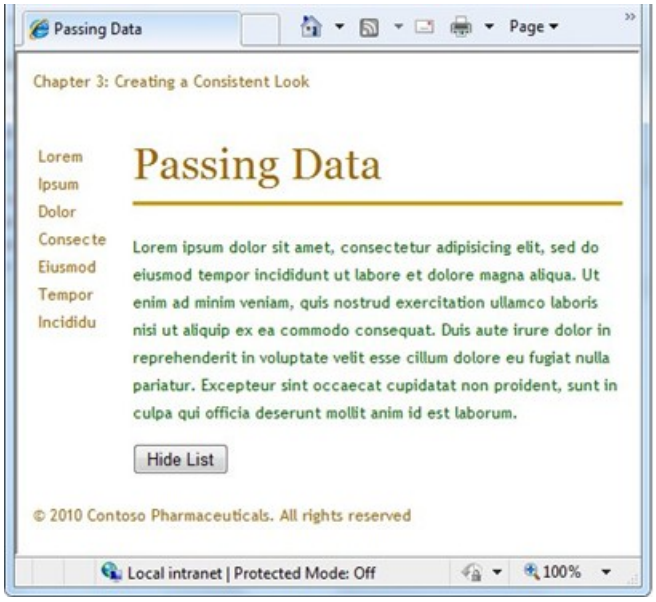
You might have data defined in the content page that you need to refer to in a layout page. If so, you need to pass the data from the content page to the layout page. For example, you might want to display the login status of a user, or you might want to show or hide content areas based on user input.

To pass data from a content page to a layout page, you can put values into the `PageData` property of the content page. The `PageData` property is a collection of name/value pairs that hold the data that you want to pass between pages. In the layout page, you can then read values out of the `PageData` property.

Here's another diagram. This one shows how ASP.NET can use the `PageData` property to pass values from a content page to the layout page. When ASP.NET begins building the web page, it creates the `PageData` collection. In the content page, you write code to put data in the `PageData` collection. Values in the `PageData` collection can also be accessed by other sections in the content page or by additional content blocks.



The following procedure shows how to pass data from a content page to a layout page. When the page runs, it displays a button that lets the user hide or show a list that's defined in the layout page. When users click the button, it sets a true/false (Boolean) value in the `PageData` property. The layout page reads that value, and if it's false, hides the list. The value is also used in the content page to determine whether to display the **Hide List** button or the **Show List** button.



1. In the root folder, create a file named *Content3.cshtml* and replace any existing content with the following:

```
@{
    Layout = "~/Shared/_Layout3.cshtml";

    PageData["Title"] = "Passing Data";
    PageData["ShowList"] = true;

    if (IsPost) {
        if (Request.Form["list"] == "off") {
            PageData["ShowList"] = false;
        }
    }
}

@section header {
  <div id="header">
    Creating a Consistent Look
  </div>
}

<h1>@PageData["Title"]</h1>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id est laborum.</p>

@if (PageData["ShowList"] == true) {
    <form method="post" action="">
      <input type="hidden" name="list" value="off" />
      <input type="submit" value="Hide List" />
    </form>
}
else {
    <form method="post" action="">
      <input type="hidden" name="list" value="on" />
      <input type="submit" value="Show List" />
    </form>
}
```

   The code stores two pieces of data in the `PageData` property — the title of the web page and true or false to specify whether to display a list.

   Notice that ASP.NET lets you put HTML markup into the page conditionally using a code block. For example, the `if/else` block in the body of the page determines which form to display depending on whether `PageData["ShowList"]` is set to true.

2. In the *Shared* folder, create a file named *_Layout3.cshtml* and replace any existing content with the following:

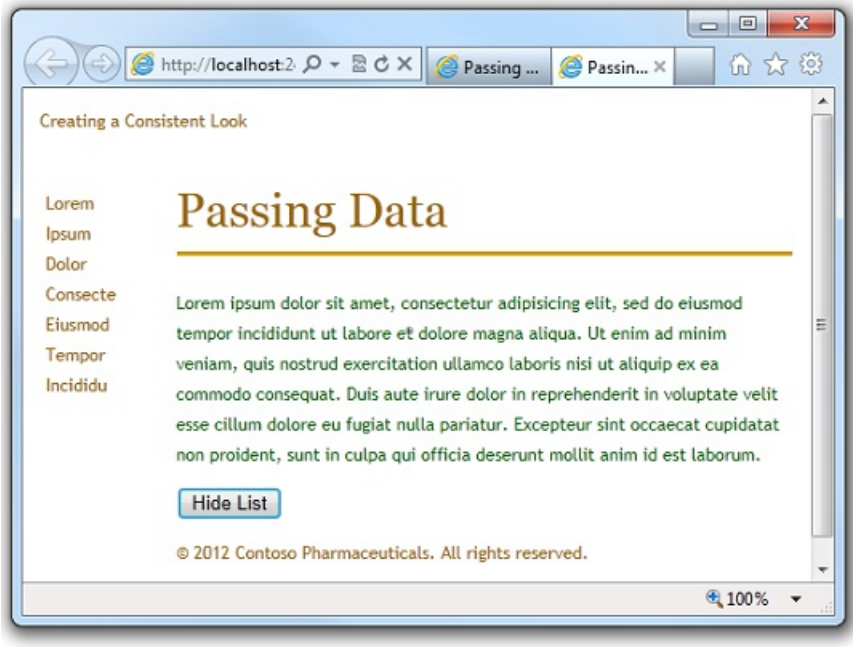
```
<!DOCTYPE html>
<html>
  <head>
    <title>@PageData["Title"]</title>
    <link href="~/Styles/Site.css" rel="stylesheet" type="text/css" />
  </head>
  <body>
    <div id="header">
      @RenderSection("header")
    </div>
      @if (PageData["ShowList"] == true) {
          <div id="list">
            @RenderPage("~/Shared/_List.cshtml")
          </div>
      }
    <div id="main">
      @RenderBody()
    </div>
    <div id="footer">
      <p>&copy; 2012 Contoso Pharmaceuticals. All rights reserved.</p>
    </div>
  </body>
</html>
```

The layout page includes an expression in the `<title>` element that gets the title value from the `PageData` property. It also uses the `ShowList` value of the `PageData` property to determine whether to display the list content block.

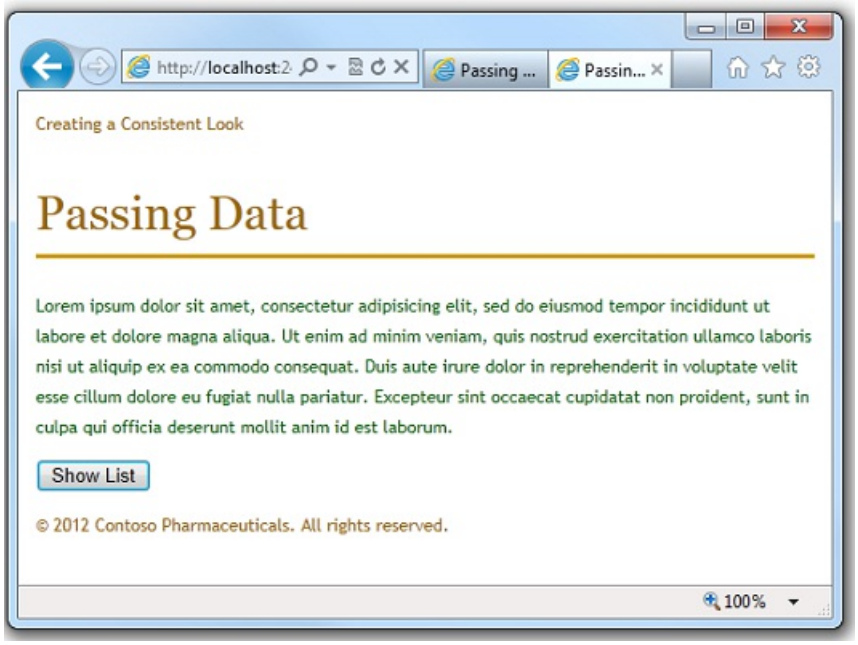
3. In the *Shared* folder, create a file named *_List.cshtml* and replace any existing content with the following:

```
<ul>
  <li>Lorem</li>
  <li>Ipsum</li>
  <li>Dolor</li>
  <li>Consecte</li>
  <li>Eiusmod</li>
  <li>Tempor</li>
  <li>Incididu</li>
</ul>
```

4. Run the *Content3.cshtml* page in a browser. The page is displayed with the list visible on the left side of the page and a **Hide List** button at the bottom.



5. Click **Hide List**. The list disappears and the button changes to **Show List**.



6. Click the **Show List** button, and the list is displayed again.

Additional Resources

Customizing Site-Wide Behavior for ASP.NET Web Pages

Customizing Site-Wide Behavior for ASP.NET Web Pages (Razor) Sites

2/20/2020 • 12 minutes to read • Edit Online

by Tom FitzMacken

This article explains how to make site-side settings for pages in an ASP.NET Web Pages (Razor) website.

What you'll learn:

- How to run code that lets you set values (global values or helper settings) for all pages in a site.
- How to run code that lets you set values for all pages in a folder.
- How to run code before and after a page loads.
- How to send errors to a central error page.
- How to add authentication to all pages in a folder.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 2
- WebMatrix 3
- ASP.NET Web Helpers Library (NuGet package)

This tutorial also works with ASP.NET Web Pages 3 and Visual Studio 2013 (or Visual Studio Express 2013 for Web), except you cannot use the ASP.NET Web Helpers Library.

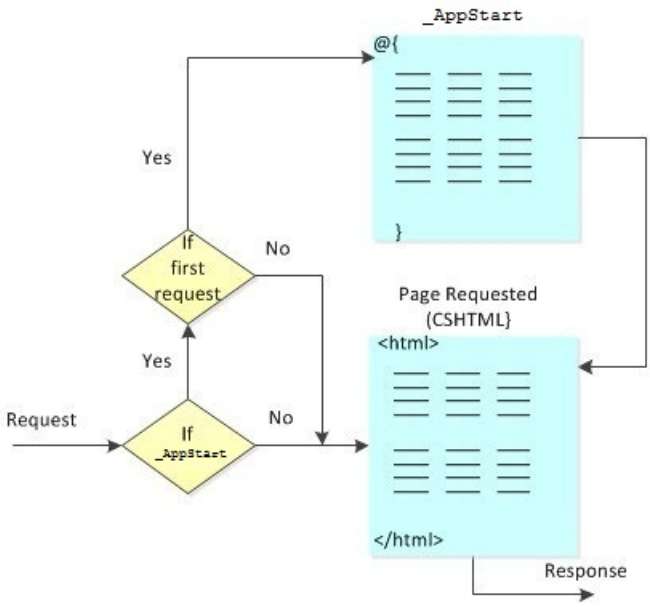
Adding Website Startup Code for ASP.NET Web Pages

For much of the code that you write in ASP.NET Web Pages, an individual page can contain all the code that's required for that page. For example, if a page sends an email message, it's possible to put all the code for that operation in a single page. This can include the code to initialize the settings for sending email (that is, for the SMTP server) and for sending the email message.

However, in some situations, you might want to run some code before any page on the site runs. This is useful for setting values that can be used anywhere in the site (referred to as *global values*.) For example, some helpers require you to provide values like email settings or account keys. It can be handy to keep these settings in global values.

You can do this by creating a page named *_AppStart.cshtml* in the root of the site. If this page exists, it runs the first time any page in the site is requested. Therefore, it's a good place to run code to set global values. (Because *_AppStart.cshtml* has an underscore prefix, ASP.NET won't send the page to a browser even if users request it directly.)

The following diagram shows how the *_AppStart.cshtml* page works. When a request comes in for a page, and if this is the first request for any page in the site, ASP.NET first checks whether a *_AppStart.cshtml* page exists. If so, any code in the *_AppStart.cshtml* page runs, and then the requested page runs.



Setting Global Values for Your Website

1. In the root folder of a WebMatrix website, create a file named *_AppStart.cshtml*. The file must be in the root of the site.

2. Replace the existing content with the following:

```
@{
  AppState["customAppName"] = "Application Name";
}
```

   This code stores a value in the `AppState` dictionary, which is automatically available to all pages in the site. Notice that the *_AppStart.cshtml* file does not have any markup in it. The page will run the code and then redirect to the page that was originally requested.

   > **NOTE**
   > Be careful when you put code in the *_AppStart.cshtml* file. If any errors occur in code in the *_AppStart.cshtml* file, the website won't start.

3. In the root folder, create a new page named *AppName.cshtml*.

4. Replace the default markup and code with the following:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Show Application Name</title>
    </head>
    <body>
        <h1>@AppState["customAppName"]</h1>
    </body>
</html>
```

   This code extracts the value from the `AppState` object that you set in the *_AppStart.cshtml* page.

5. Run the *AppName.cshtml* page in a browser. (Make sure the page is selected in the **Files** workspace before you run it.) The page displays the global value.

Setting Values for Helpers

A good use for the *_AppStart.cshtml* file is to set values for helpers that you use in your site and that have to be initialized. Typical examples are email settings for the `WebMail` helper and the private and public keys for the `ReCaptcha` helper. In cases like these, you can set the values once in the *_AppStart.cshtml* and then they're already set for all the pages in your site.

This procedure shows you how to set `WebMail` settings globally. (For more information about using the `WebMail` helper, see Adding Email to an ASP.NET Web Pages Site.)

1. Add the ASP.NET Web Helpers Library to your website as described in Installing Helpers in an ASP.NET Web Pages Site, if you haven't already added it.

2. If you don't already have a *_AppStart.cshtml* file, in the root folder of a website create a file named *_AppStart.cshtml*.

3. Add the following `WebMail` settings to the *_AppStart.cshtml* file:

```
@{
     // Initialize WebMail helper
     WebMail.SmtpServer = "your-SMTP-host";
     WebMail.SmtpPort = 25;
     WebMail.UserName = "your-user-name-here";
     WebMail.Password = "your-account-password";
     WebMail.From = "your-email-address-here";
}
```

   Modify the following email related settings in the code:

   - Set `your-SMTP-host` to the name of the SMTP server that you have access to.
   - Set `your-user-name-here` to the user name for your SMTP server account.
   - Set `your-account-password` to the password for your SMTP server account.
   - Set `your-email-address-here` to your own email address. This is the email address that the message is sent from. (Some email providers don't let you specify a different `From` address and will use your user name as the `From` address.)

     For more information about SMTP settings, see Configuring Email Settings in the article Sending Email from an ASP.NET Web Pages (Razor) Site and Issues with Sending Email in the ASP.NET Web Pages (Razor) Troubleshooting Guide.

4. Save the *_AppStart.cshtml* file and close it.

5. In the root folder of a website, create new page named *TestEmail.cshtml*.

6. Replace the existing content with the following:

```
@{
    var message = "";
    try{
        if(IsPost){
            WebMail.Send(
                to: Request.Form["emailAddress"],
                subject: Request.Form["emailSubject"],
                body:Request.Form["emailBody"]
           );
           message = "Email sent!";
        }
    }
    catch(Exception ex){
        message = "Email could not be sent!";
    }
}
<!DOCTYPE html>
<html lang="en">
  <head>
     <meta charset="utf-8" />
     <title>Test Email</title>
  </head>
  <body>
    <h1>Test Email</h1>
    <form method="post">
      <p>
        <label for="emailAddress">Email address:</label>
        <input type="text" name="emailAddress" />
      </p>
      <p>
        <label for="emailSubject">Subject:</label>
        <input type="text" name="emailSubject" />
      </p>
      <p>
        <label for="emailBody">Text to send:</label><br/>
        <textarea name="emailBody" rows="6"></textarea>
      </p>
    <p><input type="submit" value="Send!" /></p>
    @if(IsPost){
        <p>@message</p>
    }
    </form>
  </body>
</html>
```

7. Run the *TestEmail.cshtml* page in a browser.

8. Fill in the fields to send yourself an email message and then click **Send**.

9. Check your email to make sure you've gotten the message.

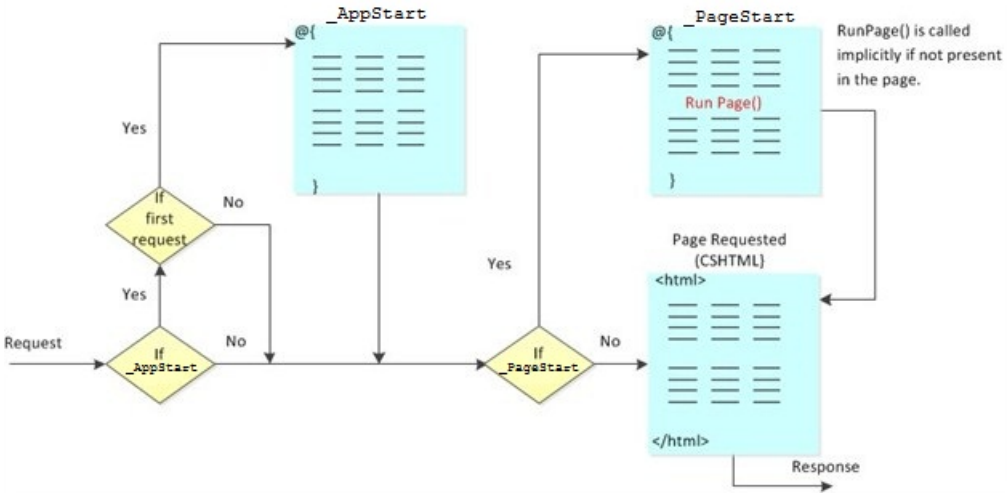
The important part of this example is that the settings that you don't usually change — like the name of your SMTP server and your email credentials — are set in the *_AppStart.cshtml* file. That way you don't need to set them again in each page where you send email. (Although if for some reason you need to change those settings, you can set them individually in a page.) In the page, you only set the values that typically change each time, like the recipient and the body of the email message.

Running Code Before and After Files in a Folder

Just like you can use *_AppStart.cshtml* to write code before pages in the site run, you can write code that runs before (and after) any page in a particular folder run. This is useful for things like setting the same layout page for all the pages in a folder, or for checking that a user is logged in before running a page in the folder.

For pages in particular folders, you can create code in a file named *_PageStart.cshtml*. The following diagram shows how the *_PageStart.cshtml* page works. When a request comes in for a page, ASP.NET first checks for a *_AppStart.cshtml* page and runs that. Then ASP.NET checks whether there's a *_PageStart.cshtml* page, and if so, runs that. It then runs the requested page.

Inside the *_PageStart.cshtml* page, you can specify where during processing you want the requested page to run by including a `RunPage` method. This lets you run code before the requested page runs and then again after it. If you don't include `RunPage`, all the code in *_PageStart.cshtml* runs, and then the requested page runs automatically.



ASP.NET lets you create a hierarchy of *_PageStart.cshtml* files. You can put a *_PageStart.cshtml* file in the root of the site and in any subfolder. When a page is requested, the *_PageStart.cshtml* file at the top-most level (nearest to the site root) runs, followed by the *_PageStart.cshtml* file in the next subfolder, and so on down the subfolder structure until the request reaches the folder that contains the requested page. After all the applicable *_PageStart.cshtml* files have run, the requested page runs.

For example, you might have the following combination of *_PageStart.cshtml* files and *Default.cshtml* file:

```
@* ~/_PageStart.cshtml *@
@{
  PageData["Color1"] = "Red";
  PageData["Color2"] = "Blue";
}
```

```
@* ~/myfolder/_PageStart.cshtml *@
@{
  PageData["Color2"] = "Yellow";
  PageData["Color3"] = "Green";
}
```

```
@* ~/myfolder/default.cshtml *@
@PageData["Color1"]
<br/>
@PageData["Color2"]
<br/>
@PageData["Color3"]
```

When you run */myfolder/default.cshtml*, you'll see the following:

```
Red

Yellow

Green
```

Running Initialization Code for All Pages in a Folder

A good use for *_PageStart.cshtml* files is to initialize the same layout page for all files in a single folder.

1. In the root folder, create a new folder named *InitPages*.
2. In the *InitPages* folder of your website, create a file named *_PageStart.cshtml* and replace the default markup and code with the following:

```
@{
    // Sets the layout page for all pages in the folder.
    Layout = "~/Shared/_Layout1.cshtml";

    // Sets a variable available to all pages in the folder.
    PageData["MyBackground"] = "Yellow";
}
```

3. In the root of the website, create a folder named *Shared*.
4. In the *Shared* folder, create a file named *_Layout1.cshtml* and replace the default markup and code with the following:

```
@{
  var backgroundColor = PageData["MyBackground"];
}
<!DOCTYPE html>
<html>
<head>
  <title>Page Title</title>
  <link type="text/css" href="/Styles/Site.css" rel="stylesheet" />
</head>
<body>
  <div id="header">
    Using the _PageStart.cshtml file
  </div>
  <div id="main" style="background-color:@backgroundColor">
    @RenderBody()
  </div>
<div id="footer">
  &copy; 2012 Contoso. All rights reserved
</div>
</body>
</html>
```

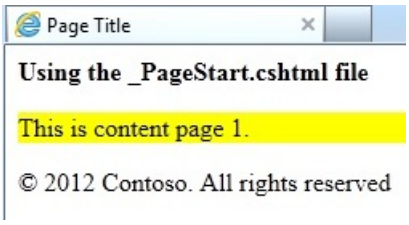
5. In the *InitPages* folder, create a file named *Content1.cshtml* and replace the existing content with the following:

```
<p>This is content page 1.</p>
```

6. In the *InitPages* folder, create another file named *Content2.cshtml* and replace the default markup with the following:

```
<p>This is content page 2.</p>
```

7. Run *Content1.cshtml* in a browser.



   When the *Content1.cshtml* page runs, the *_PageStart.cshtml* file sets `Layout` and also sets `PageData["MyBackground"]` to a color. In *Content1.cshtml*, the layout and color are applied.

8. Display *Content2.cshtml* in a browser.

   The layout is the same, because both pages use the same layout page and color as initialized in *_PageStart.cshtml*.

Using _PageStart.cshtml to Handle Errors

Another good use for the *_PageStart.cshtml* file is to create a way to handle programming errors (exceptions) that might occur in any *.cshtml* page in a folder. This example shows you one way to do this.

1. In the root folder, create a folder named *InitCatch*.

2. In the *InitCatch* folder of your website, create a file named *_PageStart.cshtml* and replace the existing markup and code with the following:

```
@{
    try
    {
        RunPage();
    }
    catch (Exception ex)
    {
        Response.Redirect("~/Error.cshtml?source=" +
            HttpUtility.UrlEncode(Request.AppRelativeCurrentExecutionFilePath));
    }
}
```

   In this code, you try running the requested page explicitly by calling the `RunPage` method inside a `try` block. If any programming errors occur in the requested page, the code inside the `catch` block runs. In this case, the code redirects to a page (*Error.cshtml*) and passes the name of the file that experienced the error as part of the URL. (You'll create the page shortly.)

3. In the *InitCatch* folder of your website, create a file named *Exception.cshtml* and replace the existing markup and code with the following:

```
@{
    var db = Database.Open("invalidDatabaseFile");
}
```

   For purposes of this example, what you're doing in this page is deliberately creating an error by trying to open a database file that doesn't exist.

4. In the root folder, create a file named *Error.cshtml* and replace the existing markup and code with the

following:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Error Page</title>
    </head>
    <body>
<h1>Error report</h1>
<p>An error occurred while running the following file: @Request["source"]</p>
    </body>
</html>
```

In this page, the expression `@Request["source"]` gets the value out of the URL and displays it.

5. In the toolbar, click **Save**.

6. Run *Exception.cshtml* in a browser.

Because an error occurs in *Exception.cshtml*, the *_PageStart.cshtml* page redirects to the *Error.cshtml* file, which displays the message.

For more information about exceptions, see Introduction to ASP.NET Web Pages Programming Using the Razor Syntax.

Using _PageStart.cshtml to Restrict Folder Access

You can also use the *_PageStart.cshtml* file to restrict access to all the files in a folder.

1. In WebMatrix, create a new website using the **Site From Template** option.

2. From the available templates, select **Starter Site**.

3. In the root folder, create a folder named *AuthenticatedContent*.

4. In the *AuthenticatedContent* folder, create a file named *_PageStart.cshtml* and replace the existing markup and code with the following:

```
@{
    Response.CacheControl = "no-cache";
    if (!WebSecurity.IsAuthenticated) {
        var returnUrl = "~/Account/Login?ReturnUrl=" + Request.Url.LocalPath;
        Response.Redirect(returnUrl);
    }
}
```

The code starts by preventing all files in the folder from being cached. (This is required for scenarios like public computers, where you don't want one user's cached pages to be available to the next user.) Next, the code determines whether the user has signed in to the site before they can view any of the pages in the folder. If the user is not signed in, the code redirects to the login page. The login page can return the user to the page that was originally requested if you include a query string value named `ReturnUrl`.

5. Create a new page in the *AuthenticatedContent* folder named *Page.cshtml*.

6. Replace the default markup with the following:

```
@{
    Layout = "~/_SiteLayout.cshtml";
    Page.Title = "Authenticated Content";
}
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
  </head>
  <body>
    Thank you for authenticating!
  </body>
</html>
```

7. Run *Page.cshtml* in a browser. The code redirects you to a login page. You must register before logging in. After you've registered and logged in, you can navigate to the page and view its contents.

Additional Resources

Introduction to ASP.NET Web Pages Programming Using the Razor Syntax

Working with Images in an ASP.NET Web Pages (Razor) Site

2/20/2020 • 15 minutes to read • Edit Online

by Tom FitzMacken

This article shows you how to add, display, and manipulate images (resize, flip, and add watermarks) in an ASP.NET Web Pages (Razor) website.

What you'll learn:

- How to add an image to a page dynamically.
- How to let users upload an image.
- How to resize an image.
- How to flip or rotate an image.
- How to add a watermark to an image.
- How to use an image as a watermark.

These are the ASP.NET programming features introduced in the article:

- The `WebImage` helper.
- The `Path` object, which provides methods that let you manipulate path and file names.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 2
- WebMatrix 2

This tutorial also works with WebMatrix 3.

Adding an Image to a Web Page Dynamically

You can add images to your website and to individual pages while you're developing the website. You can also let users upload images, which might be useful for tasks like letting them add a profile photo.

If an image is already available on your site and you just want to display it on a page, you use an HTML `<img>` element like this:

```
<img src="images/Photo1.jpg" alt="Sample Photo" />
```

Sometimes, though, you need to be able to display images dynamically — that is, you don't know what image to display until the page is running.

The procedure in this section shows how to display an image on the fly where users specify the image file name from a list of image names. They select the name of the image from a drop-down list, and when they submit the page, the image they selected is displayed.

1. In WebMatrix, create a new website.
2. Add a new page named *DynamicImage.cshtml*.
3. In the root folder of the website, add a new folder and name it *images*.
4. Add four images to the *images* folder you just created. (Any images you have handy will do, but they should fit onto a page.) Rename the images *Photo1.jpg*, *Photo2.jpg*, *Photo3.jpg*, and *Photo4.jpg*. (You won't use *Photo4.jpg* in this procedure, but you'll use it later in the article.)
5. Verify that the four images are not marked as read-only.
6. Replace the existing content in the page with the following:

```
@{  var imagePath= "";
    if( Request["photoChoice"] != null){
        imagePath = @"images\" + Request["photoChoice"];
   }
}
<!DOCTYPE html>
<html>
<head>
  <title>Display Image on the Fly</title>
</head>
<body>
<h1>Displaying an Image On the Fly</h1>
<form method="post" action="">
    <div>
        I want to see:
        <select name="photoChoice">
            <option value="Photo1.jpg">Photo 1</option>
            <option value="Photo2.jpg">Photo 2</option>
            <option value="Photo3.jpg">Photo 3</option>
        </select>
         
        <input type="submit" value="Submit" />
    </div>
    <div style="padding:10px;">
        @if(imagePath != ""){
            <img src="@imagePath" alt="Sample Image" width="300px" />
        }
    </div>
</form>
</body>
</html>
```

The body of the page has a drop-down list (a `<select>` element) that's named `photoChoice`. The list has

three options, and the `value` attribute of each list option has the name of one of the images that you put in the *images* folder. Essentially, the list lets the user select a friendly name like "Photo 1", and it then passes the *.jpg* file name when the page is submitted.

In the code, you can get the user's selection (in other words, the image file name) from the list by reading `Request["photoChoice"]`. You first see if there's a selection at all. If there is, you construct a path for the image that consists of the name of the folder for the images and the user's image file name. (If you tried to construct a path but there was nothing in `Request["photoChoice"]`, you'd get an error.) This results in a relative path like this:

images/Photo1.jpg

The path is stored in variable named `imagePath` that you'll need later in the page.

In the body, there's also an `<img>` element that's used to display the image that the user picked. The `src` attribute isn't set to a file name or URL, like you'd do to display a static element. Instead, it's set to `@imagePath`, meaning that it gets its value from the path you set in code.

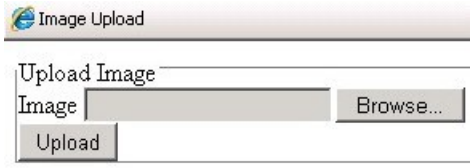
The first time that the page runs, though, there's no image to display, because the user hasn't selected anything. This would normally mean that the `src` attribute would be empty and the image would show up as a red "x" (or whatever the browser renders when it can't find an image). To prevent this, you put the `<img>` element in an `if` block that tests to see whether the `imagePath` variable has anything in it. If the user made a selection, `imagePath` contains the path. If the user didn't pick an image or if this is the first time the page is displayed, the `<img>` element isn't even rendered.

7. Save the file and run the page in a browser. (Make sure the page is selected in the **Files** workspace before you run it.)

8. Select an image from the drop-down list and then click **Sample Image**. Make sure that you see different images for different choices.

Uploading an Image

The previous example showed you how to display an image dynamically, but it worked only with images that were already on your website. This procedure shows how to let users upload an image, which is then displayed on the page. In ASP.NET, you can manipulate images on the fly using the `WebImage` helper, which has methods that let you create, manipulate, and save images. The `WebImage` helper supports all the common web image file types, including *.jpg*, *.png*, and *.bmp*. Throughout this article, you'll use *.jpg* images, but you can use any of the image types.



1. Add a new page and name it *UploadImage.cshtml*.

2. Replace the existing content in the page with the following:

```
@{  WebImage photo = null;
    var newFileName = "";
    var imagePath = "";

    if(IsPost){
        photo = WebImage.GetImageFromRequest();
        if(photo != null){
            newFileName = Guid.NewGuid().ToString() + "_" +
                Path.GetFileName(photo.FileName);
            imagePath = @"images\" + newFileName;

            photo.Save(@"~\" + imagePath);
        }
    }
}
<!DOCTYPE html>
<html>
<head>
  <title>Image Upload</title>
</head>
<body>
  <form action="" method="post" enctype="multipart/form-data">
    <fieldset>
      <legend> Upload Image </legend>
      <label for="Image">Image</label>
      <input type="file" name="Image" />
      <br/>
      <input type="submit" value="Upload" />
    </fieldset>
  </form>
  <h1>Uploaded Image</h1>
    @if(imagePath != ""){
    <div class="result">
        <img src="@imagePath" alt="image" />

    </div>
    }
</body>
</html>
```

The body of the text has an `<input type="file">` element, which lets users select a file to upload. When they click **Submit**, the file they picked is submitted along with the form.

To get the uploaded image, you use the `WebImage` helper, which has all sorts of useful methods for working with images. Specifically, you use `WebImage.GetImageFromRequest` to get the uploaded image (if any) and store it in a variable named `photo`.

A lot of the work in this example involves getting and setting file and path names. The issue is that you want to get the name (and just the name) of the image that the user uploaded, and then create a new path for where you're going to store the image. Because users could potentially upload multiple images that have the same name, you use a bit of extra code to create unique names and make sure that users don't overwrite existing pictures.

If an image actually has been uploaded (the test `if (photo != null)`), you get the image name from the image's `FileName` property. When the user uploads the image, `FileName` contains the user's original name, which includes the path from the user's computer. It might look like this:

C:\Users\Joe\Pictures\SamplePhoto1.jpg

You don't want all that path information, though — you just want the actual file name (*SamplePhoto1.jpg*).

You can strip out just the file from a path by using the `Path.GetFileName` method, like this:

```
Path.GetFileName(photo.FileName)
```

You then create a new unique file name by adding a GUID to the original name. (For more about GUIDs, see About GUIDs later in this article.) Then you construct a complete path that you can use to save the image. The save path is made up of the new file name, the folder (images), and the current website location.

> **NOTE**
> In order for your code to save files in the *images* folder, the application needs read-write permissions for that folder. On your development computer this is not typically an issue. However, when you publish your site to a hosting provider's web server, you might need to explicitly set those permissions. If you run this code on a hosting provider's server and get errors, check with the hosting provider to find out how to set those permissions.

Finally, you pass the save path to the `Save` method of the `WebImage` helper. This stores the uploaded image under its new name. The save method looks like this: `photo.Save(@"~\" + imagePath)`. The complete path is appended to `@"~\"`, which is the current website location. (For information about the `~` operator, see Introduction to ASP.NET Web Programming Using the Razor Syntax.)

As in the previous example, the body of the page contains an `<img>` element to display the image. If `imagePath` has been set, the `<img>` element is rendered and its `src` attribute is set to the `imagePath` value.

3. Run the page in a browser.
4. Upload an image and make sure it's displayed in the page.
5. In your site, open the *images* folder. You see that a new file has been added whose file name looks something like this::

   45ea4527-7ddd-4965-b9ca-c6444982b342_MyPhoto.png

   This is the image that you uploaded with a GUID prefixed to the name. (Your own file will have a different GUID and probably is named something different than *MyPhoto.png*.)

> **TIP**
>
> **About GUIDs**
> A GUID (globally-unique ID) is an identifier that's usually rendered in a format like this: `936DA01F-9ABD-4d9d-80C7-02AF85C822A8`. The numbers and letters (from A-F) differ for each GUID, but they all follow the pattern of using groups of 8-4-4-4-12 characters. (Technically, a GUID is a 16-byte/128-bit number.) When you need a GUID, you can call specialized code that generates a GUID for you. The idea behind GUIDs is that between the enormous size of the number (3.4×10^{38}) and the algorithm for generating it, the resulting number is virtually guaranteed to be one of a kind. GUIDs therefore are a good way to generate names for things when you must guarantee that you won't use the same name twice. The downside, of course, is that GUIDs aren't particularly user friendly, so they tend to be used when the name is used only in code.

Resizing an Image

If your website accepts images from a user, you might want to resize the images before you display or save them. You can again use the `WebImage` helper for this.

This procedure shows how to resize an uploaded image to create a thumbnail and then save the thumbnail and original image in the website. You display the thumbnail on the page and use a hyperlink to redirect users to the

full-sized image.

1. Add a new page named *Thumbnail.cshtml*.
2. In the *images* folder, create a subfolder named *thumbs*.
3. Replace the existing content in the page with the following:

```
@{
    WebImage photo = null;
    var newFileName = "";
    var imagePath = "";
    var imageThumbPath  = "";

    if(IsPost){
        photo = WebImage.GetImageFromRequest();
        if(photo != null){
             newFileName = Guid.NewGuid().ToString() + "_" +
                 Path.GetFileName(photo.FileName);
             imagePath = @"images\" + newFileName;
             photo.Save(@"~\" + imagePath);

            imageThumbPath = @"images\thumbs\" + newFileName;
            photo.Resize(width: 60, height: 60, preserveAspectRatio: true,
               preventEnlarge: true);
            photo.Save(@"~\" + imageThumbPath);        }
    }
}
<!DOCTYPE html>
<html>
<head>
  <title>Resizing Image</title>
</head>
<body>
<h1>Thumbnail Image</h1>
  <form action="" method="post" enctype="multipart/form-data">
    <fieldset>
      <legend> Creating Thumbnail Image </legend>
      <label for="Image">Image</label>
      <input type="file" name="Image" />
      <br/>
      <input type="submit" value="Submit" />
    </fieldset>
  </form>
    @if(imagePath != ""){
    <div class="result">
        <img src="@imageThumbPath" alt="Thumbnail image" />
        <a href="@Html.AttributeEncode(imagePath)" target="_Self">
            View full size
        </a>
    </div>

    }
</body>
</html>
```

This code is similar to the code from the previous example. The difference is that this code saves the image twice, once normally and once after you create a thumbnail copy of the image. First you get the uploaded image and save it in the *images* folder. You then construct a new path for the thumbnail image. To actually create the thumbnail, you call the `WebImage` helper's `Resize` method to create a 60-pixel by 60-pixel image. The example shows how you preserve the aspect ratio and how you can prevent the image from being enlarged (in case the new size would actually make the image larger). The resized image is then saved in the *thumbs* subfolder.

At the end of the markup, you use the same `<img>` element with the dynamic `src` attribute that you've seen in the previous examples to conditionally show the image. In this case, you display the thumbnail. You also use an `<a>` element to create a hyperlink to the big version of the image. As with the `src` attribute of the `<img>` element, you set the `href` attribute of the `<a>` element dynamically to whatever is in `imagePath`. To make sure that the path can work as a URL, you pass `imagePath` to the `Html.AttributeEncode` method, which converts reserved characters in the path to characters that are ok in a URL.

4. Run the page in a browser.
5. Upload a photo and verify that the thumbnail is shown.
6. Click the thumbnail to see the full-size image.
7. In the *images* and *images/thumbs*, note that new files have been added.

Rotating and Flipping an Image

The `WebImage` helper also lets you flip and rotate images. This procedure shows how to get an image from the server, flip the image upside down (vertically), save it, and then display the flipped image on the page. In this example, you're just using a file you already have on the server (*Photo2.jpg*). In a real application, you'd probably flip an image whose name you get dynamically, like you did in previous examples.

1. Add a new page named *FlipImage.cshtml*.
2. Replace the existing content in the page with the following:

```
@{  var imagePath= "";
    WebImage photo = new WebImage(@"~\Images\Photo2.jpg");
    if(photo != null){
        imagePath = @"images\Photo2.jpg";
        photo.FlipVertical();
        photo.Save(@"~\" + imagePath);
     }
}
<!DOCTYPE html>
<html>
<head>
  <title>Get Image From File</title>
  <meta http-equiv="content-type" content="text/html;charset=utf-8" />
</head>
<body>
<h1>Flip Image Vertically</h1>
@if(imagePath != ""){
  <div class="result">
    <img src="@imagePath" alt="Image" />
  </div>
}
</body>
</html>
```

The code uses the `WebImage` helper to get an image from the server. You create the path to the image using the same technique you used in earlier examples for saving images, and you pass that path when you create an image using `WebImage`:

```
WebImage photo = new WebImage(@"~\Images\Photo2.jpg");
```

If an image is found, you construct a new path and file name, like you did in earlier examples. To flip the image, you call the `FlipVertical` method, and then you save the image again.

The image is again displayed on the page by using the `<img>` element with the `src` attribute set to `imagePath`.

3. Run the page in a browser. The image for *Photo2.jpg* is shown upside down.
4. Refresh the page or request the page again to see the image is flipped right side up again.

To rotate an image, you use the same code, except that instead of calling the `FlipVertical` or `FlipHorizontal`, you call `RotateLeft` or `RotateRight`.

Adding a Watermark to an Image

When you add images to your website, you might want to add a watermark to the image before you save it or display it on a page. People often use watermarks to add copyright information to an image or to advertise their business name.

1. Add a new page named *Watermark.cshtml*.
2. Replace the existing content in the page with the following:

```
@{  var imagePath= "";
    WebImage photo = new WebImage(@"~\Images\Photo3.jpg");
    if(photo != null){
        imagePath = @"images\Photo3.jpg";
        photo.AddTextWatermark("My Watermark", fontColor:"Yellow", fontFamily:
            "Arial");
        photo.Save(@"~\" + imagePath);    }
}
<!DOCTYPE html>
<html>
<head>
  <title>Water Mark</title>
  <meta http-equiv="content-type" content="text/html;charset=utf-8" />
</head>
<body>
<h1>Adding a Watermark to an Image</h1>
@if(imagePath != ""){
  <div class="result">
    <img src="@imagePath" alt="Image" />
  </div>
}
</body>
</html>
```

This code is like the code in the *FlipImage.cshtml* page from earlier (although this time it uses the *Photo3.jpg* file). To add the watermark, you call the `WebImage` helper's `AddTextWatermark` method before you save the image. In the call to `AddTextWatermark`, you pass the text "My Watermark", set the font color to yellow, and set the font family to Arial. (Although it's not shown here, the `WebImage` helper also lets you specify opacity, font family and font size, and the position of the watermark text.) When you save the image it must not be read-only.

As you've seen before, the image is displayed on the page by using the `<img>` element with the src attribute set to `@imagePath`.

3. Run the page in a browser. Notice the text "My Watermark" at the bottom-right corner of the image.

Using an Image As a Watermark

Instead of using text for a watermark, you can use another image. People sometimes use images like a company logo as a watermark, or they use a watermark image instead of text for copyright information.

1. Add a new page named *ImageWatermark.cshtml*.
2. Add an image to the *images* folder that you can use as a logo, and rename the image *MyCompanyLogo.jpg*. This image should be an image that you can see clearly when it's set to 80 pixels wide and 20 pixels high.
3. Replace the existing content in the page with the following:

```
@{  var imagePath = "";
   WebImage WatermarkPhoto = new WebImage(@"~\" +
        @"\Images\MyCompanyLogo.jpg");
    WebImage photo = new WebImage(@"~\Images\Photo4.jpg");
    if(photo != null){
        imagePath = @"images\Photo4.jpg";
        photo.AddImageWatermark(WatermarkPhoto, width: 80, height: 20,
           horizontalAlign:"Center", verticalAlign:"Bottom",
           opacity:100,  padding:10);
      photo.Save(@"~\" + imagePath);
   }
}
<!DOCTYPE html>
<html>
<head>
  <title>Image Watermark</title>
  <meta http-equiv="content-type" content="text/html;charset=utf-8" />
</head>
<body>
  <h1>Using an Image as a Watermark</h1>
  @if(imagePath != ""){
    <div class="result">
      <img src="@imagePath" alt="Image" />
    </div>
  }
</body>
</html>
```

This is another variation on the code from earlier examples. In this case, you call `AddImageWatermark` to add the watermark image to the target image (*Photo3.jpg*) before you save the image. When you call `AddImageWatermark`, you set its width to 80 pixels and the height to 20 pixels. The *MyCompanyLogo.jpg* image is horizontally aligned in the center and vertically aligned at the bottom of the target image. The opacity is set to 100% and the padding is set to 10 pixels. If the watermark image is bigger than the target image, nothing will happen. If the watermark image is bigger than the target image and you set the padding for the image watermark to zero, the watermark is ignored.

As before, you display the image using the `<img>` element and a dynamic `src` attribute.

4. Run the page in a browser. Notice that the watermark image appears at the bottom of the main image.

Additional Resources

Working with Files in an ASP.NET Web Pages Site

Introduction to ASP.NET Web Pages Programming Using the Razor Syntax

Displaying Video in an ASP.NET Web Pages (Razor) Site

2/20/2020 • 7 minutes to read • Edit Online

by Tom FitzMacken

This article explains how to use a video (media) player in an ASP.NET Web Pages (Razor) website to let users view videos that are stored on the site. ASP.NET Web Pages with Razor syntax lets you play Flash (*.swf*), Media Player (*.wmv*), and Silverlight (*.xap*) videos.

What you'll learn:

- How to choose a video player.
- How to add video to a web page.
- How to set video player attributes.

These are the ASP.NET Razor pages features introduced in the article:

- The `Video` helper.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 2
- WebMatrix 2

This tutorial also works with WebMatrix 3.

Introduction

You might want to display a video on your site. One way to do that is to link to a site that already has the video, like YouTube. If you want to embed a video from these sites directly in your own pages, you can usually get HTML markup from the site and then copy it into your page. For example, the following example shows how to embed a YouTube video:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Embedded Video Example</title>
    </head>
    <body>
        <h1>Embedded Video Example</h1>
        <p>The following video provides an introduction to WebMatrix:</p>
        <iframe width="560"
                height="315"
                src="http://www.youtube.com/embed/fxCEcPxUbYA"
                frameborder="0"
                allowfullscreen></iframe>
</body>
</html>
```

If you want to play a video that's on your own website (not on a public video-sharing site), you can't directly link to it using embedded markup like this. However, you can play videos from your site by using the `Video` helper, which

renders a media player directly in a page.

Choosing a Video Player

There are lots of formats for video files, and each format typically requires a different player and a different way to configure the player. In ASP.NET Razor pages, you can play a video in a web page using the `Video` helper. The `Video` helper simplifies the process of embedding videos in a web page because it automatically generates the `object` and `embed` HTML elements that are normally used to add video to the page.

The `Video` helper supports the following media players:

- Adobe Flash
- Windows MediaPlayer
- Microsoft Silverlight

The Flash Player

The `Flash` player of the `Video` helper let you play Flash videos (*.swf* files) in a web page. At a minimum, you have to provide a path to the video file. If you specify nothing but the path, the player uses default values that are set by the current version of Flash. Typical default settings are:

- The video is displayed using its default width and height and without a background color.
- The video plays automatically when the page loads.
- The video loops continuously until it's explicitly stopped.
- The video is scaled to show all of the video, rather than cropping the video to fit a specific size.
- The video plays in a window.

The MediaPlayer Player

The `MediaPlayer` player of the `Video` helper lets you play Windows Media videos (*.wmv* files), Windows Media audio (*.wma* files), and MP3 (*.mp3* files) in a web page. You must include path of the media file to play; all other parameters are optional. If you specify only a path, the player uses default settings set by the current version of MediaPlayer, such as:

- The video is displayed using its default width and height.
- The video plays automatically when the page loads.
- The video plays once (it doesn't loop).
- The player displays the full set of controls in the user interface.
- The video plays in a window.

The Silverlight Player

The `Silverlight` player of the `Video` helper lets you play Windows Media Video (*.wmv* files), Windows Media Audio (*.wma* files), and MP3 (*.mp3* files). You must set the path parameter to point to a Silverlight-based application package (*.xap* file). You also must set the width and height parameters. All other parameters are optional. When you use the Silverlight player for video, if you set only the required parameters, the Silverlight player displays the video without a background color.

> **NOTE**
>
> In case you don't already know Silverlight: the *.xap* file is a compressed file that contains layout instructions in a *.xaml* file, managed code in assemblies, and optional resources. You can create a *.xap* file in Visual Studio as a Silverlight application project.

The `Silverlight` video player uses both the settings that you provide for the player and the settings that are provided in the *.xap* file.

TIP

MIME Types

When a browser downloads a file, the browser makes sure that the file type matches the MIME type that's specified for the document that's being rendered. The MIME type is the content type or media type of a file. The `Video` helper uses the following MIME types:

- `application/x-shockwave-flash`
- `application/x-mplayer2`
- `application/x-silverlight-2`

Playing Flash (.swf) Videos

This procedure shows you how to play a Flash video named *sample.swf*. The procedure assumes that you've got a folder named *Media* on your site and that the *.swf* file is in that folder.

1. Add the ASP.NET Web Helpers Library to your website as described in Installing Helpers in an ASP.NET Web Pages Site, if you haven't already added it.

2. In the website, add a page and name it *FlashVideo.cshtml*.

3. Add the following markup to the page:

```
<!DOCTYPE html>
<html>
<head>
    <title>Flash Video</title>
</head>
<body>
    @Video.Flash(path: "Media/sample.swf",
                 width: "400",
                 height: "600",
                 play: true,
                 loop: true,
                 menu:  false,
                 bgColor: "red",
                 quality: "medium",
                 scale: "exactfit",
                 windowMode: "transparent")
</body>
</html>
```

4. Run the page in a browser. (Make sure the page is selected in the **Files** workspace before you run it.) The page is displayed and the video plays automatically.

You can set the `quality` parameter for a Flash video to `low`, `autolow`, `autohigh`, `medium`, `high`, and `best`:

```
<!-- Set the Flash video quality -->
@Video.Flash(path: "Media/sample.swf", quality: "autohigh")
```

You can change the Flash video to play at a specific size using the `scale` parameter, which you can set to the following:

- `showall`. This makes the entire video visible while maintaining the original aspect ratio. However, you might end up with borders on each side.
- `noorder`. This scales the video while maintaining the original aspect ratio, but it might be cropped.
- `exactfit`. This makes the entire video visible without preserving the original aspect ratio, but distortion may occur.

If you don't specify a `scale` parameter, the entire video will be visible and the original aspect ratio will be maintained without any cropping. The following example shows how to use the `scale` parameter:

```
<!-- Set the Flash video to an exact size -->
@Video.Flash(path: "Media/sample.swf", width: "1000", height: "100",
    scale: "exactfit")
```

The Flash player supports a video mode setting named `windowMode`. You can set this to `window`, `opaque`, and `transparent`. By default, the `windowMode` is set to `window`, which displays the video in a separate window on the web page. The `opaque` setting hides everything behind the video on the web page. The `transparent` setting lets the background of the web page show through the video, assuming any part of the video is transparent.

Playing MediaPlayer (*.wmv*) Videos

The following procedure shows you how to play a Window Media video named *sample.wmv* that's in the *Media* folder.

1. Add the ASP.NET Web Helpers Library to your website as described in Installing Helpers in an ASP.NET Web Pages Site, if you haven't already.

2. Create a new page named *MediaPlayerVideo.cshtml*.

3. Add the following markup to the page:

```
<!DOCTYPE html>
<html>
<head>
  <title>MediaPlayer Video</title>
</head>
<body>
    @Video.MediaPlayer(
        path: "Media/sample.wmv",
        width: "400",
        height: "600",
        autoStart: true,
        playCount: 2,
        uiMode:  "full",
        stretchToFit: true,
        enableContextMenu: true,
        mute: false,
        volume: 75)
</body>
</html>
```

4. Run the page in a browser. The video loads and plays automatically.

You can set `playCount` to an integer that indicates how many times to play the video automatically:

```
<!-- Set the MediaPlayer video playCount -->
@Video.MediaPlayer(path: "Media/sample.wmv", playCount: 2)
```

The `uiMode` parameter lets you specify which controls show up in the user interface. You can set `uiMode` to `invisible`, `none`, `mini`, or `full`. If you don't specify a `uiMode` parameter, the video will be displayed with the status window, seek bar, control buttons, and volume controls in addition to the video window. These controls will also be displayed if you use the player to play an audio file. Here's an example of how to use the `uiMode` parameter:

```
<!-- Set the MediaPlayer control UI -->
@Video.MediaPlayer(path: "Media/sample.wmv", uiMode: "mini")
```

By default, audio is on when the video plays. You can mute the audio by setting the `mute` parameter to true:

```
<!-- Play the MediaPlayer video without audio -->
@Video.MediaPlayer(path: "Media/sample.wmv", mute: true)
```

You can control the audio level of the MediaPlayer video by setting the `volume` parameter to a value between 0 and 100. The default value is 50. Here's an example:

```
<!-- Play the MediaPlayer video without audio -->
@Video.MediaPlayer(path: "Media/sample.wmv", volume: 75)
```

Playing Silverlight Videos

This procedure shows you how to play video contained in a Silverlight *.xap* page that's in a folder named *Media*.

1. Add the ASP.NET Web Helpers Library to your website as described in Installing Helpers in an ASP.NET Web Pages Site, if you haven't already .
2. Create a new page named *SilverlightVideo.cshtml*.
3. Add the following markup to the page:

```
<!DOCTYPE html>
<html>
<head>
  <title>Silverlight Video</title>
</head>
<body>
    @Video.Silverlight(
        path: "Media/sample.xap",
        width: "400",
        height: "600",
        bgColor: "red",
        autoUpgrade: true)
</body>
</html>
```

4. Run the page in a browser.

Additional Resources

Silverlight Overview

Flash OBJECT and EMBED tag attributes

Windows Media Player 11 SDK PARAM Tags

Creating and Using a Helper in an ASP.NET Web Pages (Razor) Site

2/20/2020 • 3 minutes to read • Edit Online

by Tom FitzMacken

This article describes how to create a helper in an ASP.NET Web Pages (Razor) website. A *helper* is a reusable component that includes code and markup to perform a task that might be tedious or complex.

What you'll learn:

- How to create and use a simple helper.

These are the ASP.NET features introduced in the article:

- The `@helper` syntax.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 3

This tutorial also works with ASP.NET Web Pages 2.

Overview of Helpers

If you need to perform the same tasks on different pages in your site, you can use a helper. ASP.NET Web Pages includes a number of helpers, and there are many more that you can download and install. (A list of the built-in helpers in ASP.NET Web Pages is listed in the ASP.NET API Quick Reference.) If none of the existing helpers meet your needs, you can create your own helper.

A helper lets you use a common block of code across multiple pages. Suppose that in your page you often want to create a note item that's set apart from normal paragraphs. Perhaps the note is created as a `<div>` element that's styled as a box with a border. Rather than add this same markup to a page every time you want to display a note, you can package the markup as a helper. You can then insert the note with a single line of code anywhere you need it.

Using a helper like this makes the code in each of your pages simpler and easier to read. It also makes it easier to maintain your site, because if you need to change how the notes look, you can change the markup in one place.

Creating a Helper

This procedure shows you how to create the helper that creates the note, as just described. This is a simple example, but the custom helper can include any markup and ASP.NET code that you need.

1. In the root folder of the website, create a folder named *App_Code*. This is a reserved folder name in ASP.NET where you can put code for components like helpers.
2. In the *App_Code* folder create a new *.cshtml* file and name it *MyHelpers.cshtml*.
3. Replace the existing content with the following:

```
@helper MakeNote(string content) {
  <div class="note"
       style="border: 1px solid black; width: 90%; padding: 5px; margin-left: 15px;">
    <p>
      <strong>Note</strong>   @content
    </p>
  </div>
}
```

The code uses the `@helper` syntax to declare a new helper named `MakeNote`. This particular helper lets you pass a parameter named `content` that can contain a combination of text and markup. The helper inserts the string into the note body using the `@content` variable.

Notice that the file is named *MyHelpers.cshtml*, but the helper is named `MakeNote`. You can put multiple custom helpers into a single file.

4. Save and close the file.

Using the Helper in a Page

1. In the root folder, create a new blank file called *TestHelper.cshtml*.

2. Add the following code to the file:

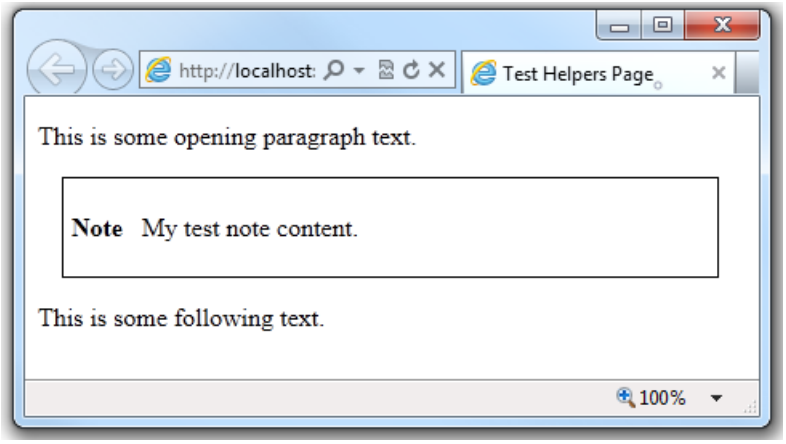
```
<!DOCTYPE html>
  <head>
    <title>Test Helpers Page</title>
  </head>
  <body>
    <p>This is some opening paragraph text.</p>

    <!-- Insert the call to your note helper here. -->
    @MyHelpers.MakeNote("My test note content.")

    <p>This is some following text.</p>
  </body>
</html>
```

To call the helper you created, use `@` followed by the file name where the helper is, a dot, and then the helper name. (If you had multiple folders in the *App_Code* folder, you could use the syntax `@FolderName.FileName.HelperName` to call your helper within any nested folder level). The text that you add in quotation marks within the parentheses is the text that the helper will display as part of the note in the web page.

3. Save the page and run it in a browser. The helper generates the note item right where you called the helper: between the two paragraphs.



Additional Resources

Horizontal menu as a Razor helper. This blog entry by Mike Pope shows how to create a horizontal menu as a helper using markup, CSS, and code.

Leveraging HTML5 in ASP.NET Web Pages Helpers for WebMatrix and ASP.NET MVC3. This blog entry by Sam Abraham shows a helper that renders an HTML5 `Canvas` element.

The Difference Between @Helpers and @Functions in WebMatrix. This blog entry by Mike Brind describes `@helper` syntax and `@function` syntax and when to use each.

Installing a Helper in an ASP.NET Web Pages (Razor) Site

2/20/2020 • 2 minutes to read • Edit Online

by Tom FitzMacken

This article describes how to install a helper in an ASP.NET Web Pages (Razor) website. A *helper* is a reusable component that includes code and markup to perform a task that might be tedious or complex.

What you'll learn:

- How to install a helper in a website created using WebMatrix 3.

Software versions used in the tutorial

- WebMatrix 3

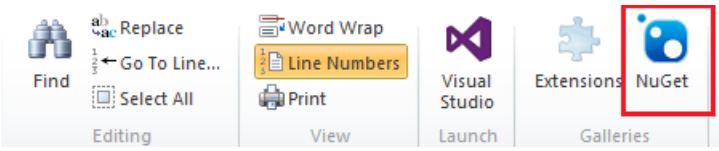
Overview of Helpers

Some tasks that people often want to do on web pages require a lot of code or require extra knowledge. Examples include displaying a chart for data; putting a Twitter "Follow" button on a page; sending email from your website; cropping or resizing images; using PayPal for your site. To make it easy to do these kinds of things, ASP.NET Web Pages lets you use *helpers*. Helpers are components that you install for a site and that let you perform typical tasks by using just a line or two of Razor code.

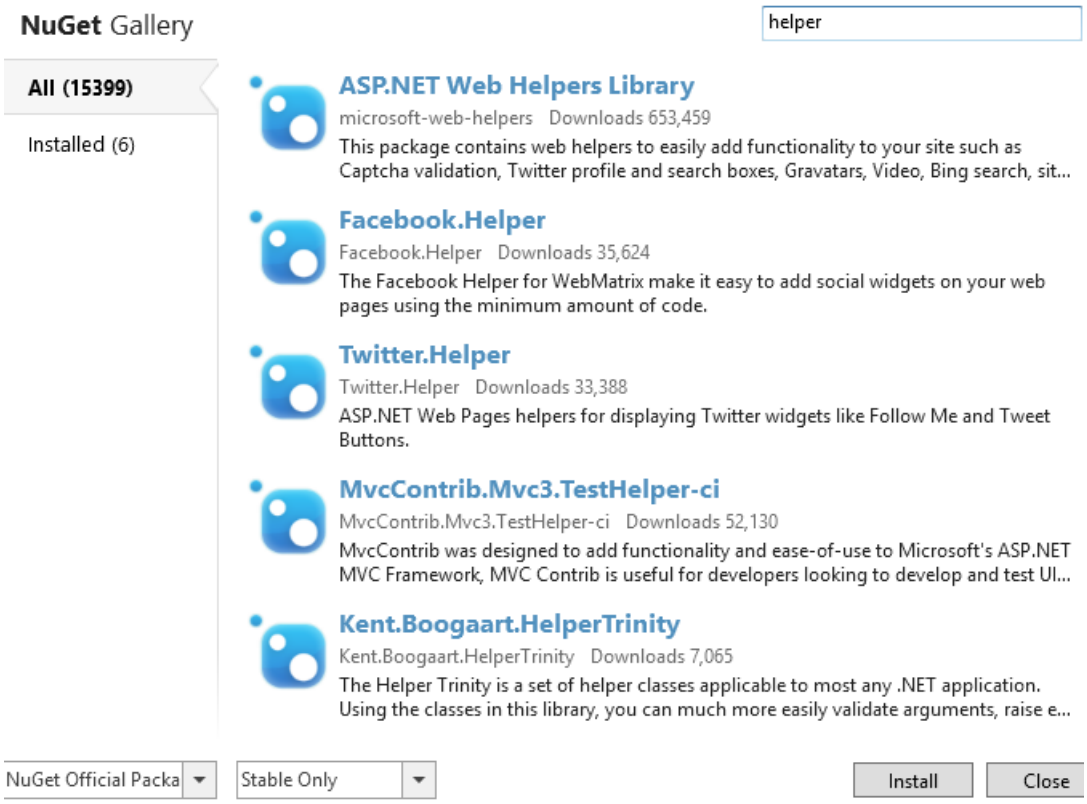
ASP.NET Web Pages has a few helpers built in. However, many helpers are available in packages (add-ins) that are provided using the NuGet package manager. NuGet lets you select a package to install and then it takes care of all the details of the installation.

Installing a Helper in WebMatrix 3

1. In WebMatrix 3, click the **NuGet** button.



2. This launches the NuGet package manager and displays available packages. In the search box, enter a keyword for the helper you want to install.



3. Select the package and then click **Install**. Click **Yes** when asked if you want to install the package and indicate that you accept the terms.

   If this is the first time you've installed a helper, NuGet creates folders in your website for the code that makes up the helper.

4. To uninstall a helper, click the **Gallery** button, click the **Installed** tab, and pick the package you want to uninstall.

Installing the Twitter helper

The latest version of the Twitter API is not compatible with the Twitter helper you install through NuGet. Instead, see the Twitter Helper with WebMatrix topic for information about how to set up the Twitter helper in your project.

Additional Resources

Introducing ASP.NET Web Pages 2 - Programming Basics

Twitter Helper with WebMatrix

Twitter Helper with ASP.NET Web Pages

2/20/2020 • 7 minutes to read • Edit Online

by Tom FitzMacken

IMPORTANT

Twitter Helpers are obsolete. For Twitter's latest engagement tools for websites, see Twitter for Websites Overview.

This topic and application show how to add a Twitter Helper to your WebMatrix 3 project. It contains the Twitter Helper code and shows how to call the helper methods.

This code for the Twitter.cshtml file was developed by **Tian Pan** of Microsoft.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 3

This tutorial also works with ASP.NET Web Pages 2.

Introduction

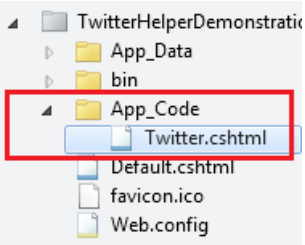
This topic demonstrates how to add a Twitter Helper to your application and use Razor syntax to call the helper methods. The Twitter Helper makes it easy to incorporate Twitter buttons and widgets in your application. To use a Twitter widget, such as a user's timeline or the search results for a hashtag, you must first create the widget on Twitter. After creating your widget, you will receive a widget id. You pass this widget id as a parameter when calling the helper methods that show widget.

This topic was written for version 1.1 of the Twitter API. By directly adding the Twitter Helper code to your project, you can update the helper code if the Twitter API changes.

For information about installing WebMatrix, see Introducing ASP.NET Web Pages 2 - Getting Started.

Add Twitter Helper to your project

To add the Twitter Helper, first, add a folder named **App_Code** to your project. Then, create a file named **Twitter.cshtml**.



Replace the default code in Twitter.cshtml with the following code.

```
@* This Twitter helper is compatible with version 1.1 of the Twitter API. *@

@using System.Globalization

@* For more about the twitter follow button, please visit
```

```
   https://dev.twitter.com/docs/follow-button *@
@helper FollowButton(
            string userName,
            bool showCount = false,
            bool showUserName = true,
            bool largeButton = false,
            bool optOutOfTailoring = false,
            string language = "",
            bool alignLeft = true)
{
    var showCountAttribute = new HtmlString(showCount ? "" : "data-show-count=\"false\"");
    var showUserNameAttribute = new HtmlString(showUserName ? "" : "data-show-screen-name=\"false\"");
    var largeButtonAttribute = new HtmlString(largeButton ? "data-size=\"large\"" : "");
    var optOutOfTailoringAttribute = new HtmlString(optOutOfTailoring ? "data-dnt=\"true\"" : "");
    var languageAttribute = new HtmlString(!language.IsEmpty() && !language.Equals("en",
StringComparison.OrdinalIgnoreCase) ? String.Format(CultureInfo.InvariantCulture, " data-lang=\"" +
HttpUtility.HtmlAttributeEncode(language) + "\"") : "");
    var alignAttribute = new HtmlString(alignLeft ? "" : "data-align=\"right\"");
    <a href="https://twitter.com/@HttpUtility.UrlEncode(userName)" class="twitter-follow-button"
@showCountAttribute @showUserNameAttribute @largeButtonAttribute @optOutOfTailoringAttribute
@languageAttribute @alignAttribute>Follow @@@userName)</a>
    <script>!function (d, s, id) { var js, fjs = d.getElementsByTagName(s)[0], p = /^http:/.test(d.location)
? 'http' : 'https'; if (!d.getElementById(id)) { js = d.createElement(s); js.id = id; js.src = p +
'://platform.twitter.com/widgets.js'; fjs.parentNode.insertBefore(js, fjs); } }(document, 'script', 'twitter-
wjs');</script>
}

@* For more about the tweet button, please visit https://dev.twitter.com/docs/tweet-button *@
@helper TweetButton(string url = "",
            string tweetText = "",
            bool showCount = true,
            string via = "",
            string recommend = "",
            string hashtag = "",
            bool largeButton = false,
            bool optOutOfTailoring = false,
            string language = "")
{
    var urlAttribute = new HtmlString(url.IsEmpty() ? "" : String.Format(CultureInfo.InvariantCulture, "
data-url=\"" + HttpUtility.HtmlAttributeEncode(url) + "\""));
    var tweetTextAttribute = new HtmlString(tweetText.IsEmpty() ? "" : "data-text=\"" +
HttpUtility.HtmlAttributeEncode(tweetText) + "\"");
    var showCountAttribute = new HtmlString(showCount ? "" : "data-show-count=\"false\"");
    var viaAttribute = new HtmlString(via.IsEmpty() ? "" : "data-via=\"" +
HttpUtility.HtmlAttributeEncode(via) + "\"");
    var recommendAttribute = new HtmlString(recommend.IsEmpty() ? "" : "data-related=\"" +
HttpUtility.HtmlAttributeEncode(recommend) + "\"");
    var hashtagAttribute = new HtmlString(hashtag.IsEmpty() ? "" : "data-hashtags=\"" +
HttpUtility.HtmlAttributeEncode(hashtag) + "\"");
    var largeButtonAttribute = new HtmlString(largeButton ? "data-size=\"large\"" : "");
    var optOutOfTailoringAttribute = new HtmlString(optOutOfTailoring ? "data-dnt=\"true\"" : "");
    var languageAttribute = new HtmlString(!language.IsEmpty() && !language.Equals("en",
StringComparison.OrdinalIgnoreCase) ? String.Format(CultureInfo.InvariantCulture, " data-lang=\"{0}\"",
HttpUtility.HtmlAttributeEncode(language)) : "");
    <a href="https://twitter.com/share" class="twitter-share-button" @urlAttribute @tweetTextAttribute
@showCountAttribute @viaAttribute @recommendAttribute @hashtagAttribute @largeButtonAttribute
@optOutOfTailoringAttribute @languageAttribute>Tweet</a>
    <script>!function (d, s, id) { var js, fjs = d.getElementsByTagName(s)[0], p = /^http:/.test(d.location)
? 'http' : 'https'; if (!d.getElementById(id)) { js = d.createElement(s); js.id = id; js.src = p +
'://platform.twitter.com/widgets.js'; fjs.parentNode.insertBefore(js, fjs); } }(document, 'script', 'twitter-
wjs');</script>
}

@helper TimeLine(string userName,
            string widgetId,
            string theme = "",
            string linkColor = "",
            string language = "",
            int? tweetLimit = null,
```

```
            string relatedUsers = "",
            string ariaPolite = "",
            int? width = null,
            int? height = null,
            string title = "Tweets")
{
    var themeAttribute = new HtmlString(theme.IsEmpty() ? "" : "data-theme=\"" + theme + "\"");
    var linkColorAttribute = new HtmlString(linkColor.IsEmpty() ? "" : "data-link-color=\"" + linkColor +
"\"");
    var languageAttribute = new HtmlString(!language.IsEmpty() && !language.Equals("en",
StringComparison.OrdinalIgnoreCase) ? String.Format(CultureInfo.InvariantCulture, " data-lang=\"{0}\"",
HttpUtility.HtmlAttributeEncode(language)) : "");
    var relatedUsersAttribute = new HtmlString(relatedUsers.IsEmpty() ? "" : "data-related=\"" + relatedUsers
+ "\"");
    var ariaPoliteAttribute = new HtmlString(ariaPolite.IsEmpty() ? "" : "data-aria-polite=\"" + ariaPolite +
"\"");
    <a class="twitter-timeline" href="https://twitter.com/@userName" width="@width.ToString()"
height="@height.ToString()" data-widget-id="@HttpUtility.HtmlEncode(widgetId)" @themeAttribute
@linkColorAttribute @languageAttribute data-tweet-limit="@tweetLimit.ToString()" @relatedUsersAttribute
@ariaPoliteAttribute>@title</a>
    <script>!function (d, s, id) { var js, fjs = d.getElementsByTagName(s)[0], p = /^http:/.test(d.location)
? 'http' : 'https'; if (!d.getElementById(id)) { js = d.createElement(s); js.id = id; js.src = p +
"://platform.twitter.com/widgets.js"; fjs.parentNode.insertBefore(js, fjs); } }(document, "script", "twitter-
wjs");</script>
}

@helper Profile(string userName,
            string widgetId,
            string theme = "",
            string linkColor = "",
            string language = "",
            int? tweetLimit = null,
            string relatedUsers = "",
            string ariaPolite = "",
            int? width = null,
            int? height = null)
{
    @Twitter.TimeLine(userName, widgetId, theme, linkColor, language, tweetLimit, relatedUsers, ariaPolite,
width, height, "Tweets by @" + HttpUtility.HtmlEncode(userName));
}

@helper Faves(string userName,
            string widgetId,
            string theme = "",
            string linkColor = "",
            string language = "",
            int? tweetLimit = null,
            string relatedUsers = "",
            string ariaPolite = "",
            int? width = null,
            int? height = null)
{
    @Twitter.TimeLine(userName + "/favorites", widgetId, theme, linkColor, language, tweetLimit,
relatedUsers, ariaPolite, width, height, "Favorite Tweets by @" + HttpUtility.HtmlEncode(userName));
}

@helper List(string userName,
            string list,
            string widgetId,
            string theme = "",
            string linkColor = "",
            string language = "",
            int? tweetLimit = null,
            string relatedUsers = "",
            string ariaPolite = "",
            int? width = null,
            int? height = null)
{
    @Twitter.TimeLine(userName + "/" + list, widgetId, theme, linkColor, language, tweetLimit, relatedUsers,
```

```
    @Twitter.TimeLine(userName + "/" + list, widgetId, theme, linkColor, language, tweetLimit, relatedUsers,
ariaPolite, width, height, "Tweets from @" + HttpUtility.HtmlEncode(userName) + "/" +
HttpUtility.HtmlEncode(list));
}

@helper Search(string query,
            string widgetId,
            string theme = "",
            string linkColor = "",
            string language = "",
            int? tweetLimit = null,
            string relatedUsers = "",
            string ariaPolite = "",
            int? width = null,
            int? height = null)
{
    @Twitter.TimeLine("search?q=" + HttpUtility.UrlEncode(query), widgetId, theme, linkColor, language,
tweetLimit, relatedUsers, ariaPolite, width, height, "Tweets about \"" + HttpUtility.HtmlEncode(query) +
"\"");
}
```

Call Twitter methods from your web pages

The following example shows how to use the Twitter Helper methods from a page in your project. In your project, you will want to replace the parameter values with values that are relevant to your needs. You can use the provided widget ids to explore how the methods work, but you will want to generate your own widgets for your project.

Not all of the parameters shown below are required. The optional parameters are used to customize how the button or widget is displayed. For example, the Follow Button only requires the user name to follow, but the example shows how to include the number of followers, and how specify the size of the button and the language.

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Show the Twitter Helper</title>
    </head>
    <body>
        <h1>Twitter Buttons and Widgets</h1>

        <h2>1. Follow Button</h2>
        <p>@Twitter.FollowButton(userName: "aspnet", showCount: true, largeButton: false, language: "es")</p>

        <h2>2. Tweet Button</h2>
        <p>@Twitter.TweetButton(url: "http://www.asp.net/mvc", largeButton: true, hashtag: "awesome")</p>

        <h2>3. User Timeline (Profile)</h2>
        <p>@Twitter.Profile("aspnet", "370310677673422848")</p>

        <h2>4. Favorites</h2>
        <p>@Twitter.Faves("Microsoft", "372486349795753984")</p>

        <h2>5. List</h2>
        <p>@Twitter.List("MS_Consumer_Bands", "Microsoft", "372485258974748672")</p>

        <h2>6. Search</h2>
        <p>@Twitter.Search("#asp.net", "370310477957435392")</p>
    </body>
</html>
```

See the results

The above code produces the following buttons and widgets. These buttons and widgets are fully-functional, not screenshots. The Follow Button is displayed in Spanish because the language parameter was set to **es**.

Follow Button

Follow @aspnet)

```
<script>!function (d, s, id) { var js, fjs = d.getElementsByTagName(s)[0], p = /^http:/.test(d.location) ?
'http' : 'https'; if (!d.getElementById(id)) { js = d.createElement(s); js.id = id; js.src = p +
'://platform.twitter.com/widgets.js'; fjs.parentNode.insertBefore(js, fjs); } }(document, 'script', 'twitter-
wjs');</script>
```

Tweet Button

Tweet

```
<script>!function (d, s, id) { var js, fjs = d.getElementsByTagName(s)[0], p = /^http:/.test(d.location) ?
'http' : 'https'; if (!d.getElementById(id)) { js = d.createElement(s); js.id = id; js.src = p +
'://platform.twitter.com/widgets.js'; fjs.parentNode.insertBefore(js, fjs); } }(document, 'script', 'twitter-
wjs');</script>
```

User Timeline (Profile)

Tweets by @aspnet

```
<script>!function (d, s, id) { var js, fjs = d.getElementsByTagName(s)[0], p = /^http:/.test(d.location) ?
'http' : 'https'; if (!d.getElementById(id)) { js = d.createElement(s); js.id = id; js.src = p +
"://platform.twitter.com/widgets.js"; fjs.parentNode.insertBefore(js, fjs); } }(document, "script", "twitter-
wjs");</script>
```

Favorites

Favorite Tweets by Microsoft

```
<script>!function (d, s, id) { var js, fjs = d.getElementsByTagName(s)[0], p = /^http:/.test(d.location) ?
'http' : 'https'; if (!d.getElementById(id)) { js = d.createElement(s); js.id = id; js.src = p +
"://platform.twitter.com/widgets.js"; fjs.parentNode.insertBefore(js, fjs); } }(document, "script", "twitter-
wjs");</script>
```

List

Tweets from @Microsoft/MS_Consumer_Bands

```
<script>!function (d, s, id) { var js, fjs = d.getElementsByTagName(s)[0], p = /^http:/.test(d.location) ?
'http' : 'https'; if (!d.getElementById(id)) { js = d.createElement(s); js.id = id; js.src = p +
"://platform.twitter.com/widgets.js"; fjs.parentNode.insertBefore(js, fjs); } }(document, "script", "twitter-
wjs");</script>
```

Search

Tweets about "#asp.net"

```
<script>!function (d, s, id) { var js, fjs = d.getElementsByTagName(s)[0], p = /^http:/.test(d.location) ?
'http' : 'https'; if (!d.getElementById(id)) { js = d.createElement(s); js.id = id; js.src = p +
"://platform.twitter.com/widgets.js"; fjs.parentNode.insertBefore(js, fjs); } }(document, "script", "twitter-
wjs");</script>
```

Displaying Maps in an ASP.NET Web Pages (Razor) Site

2/20/2020 • 4 minutes to read • Edit Online

by Tom FitzMacken

This article explains how to display interactive maps on pages in an ASP.NET Web Pages (Razor) website based on mapping services provided by Bing, Google, MapQuest, and Yahoo.

What you'll learn:

- How to generate a map based on an address.
- How to generate a map based on latitude and longitude coordinates.
- How to register a Bing Maps Developer Account and get a key to use with Bing Maps.

This is the ASP.NET feature introduced in the article:

- The `Maps` helper.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 2
- WebMatrix 2

This tutorial also works with WebMatrix 3.

In Web Pages, you can display maps on a page by using `Maps` helper. You can generate maps based either on an address or on a set of longitude and latitude coordinates. The `Maps` class lets you call into popular map engines including Bing, Google, MapQuest, and Yahoo.

The steps for adding mapping to a page are the same regardless of which of the map engines you call. You just add a JavaScript file reference that makes available methods to display the map, and then you call methods of the `Maps` helper.

You choose a map service based on which `Maps` helper method you use. You can use any of these:

- `Maps.GetBingHtml`
- `Maps.GetGoogleHtml`
- `Maps.GetYahooHtml`
- `Maps.GetMapQuestHtml`

Installing the Pieces You Need

To display maps, you need these pieces:

- The `Maps` helper. This helper is in version 2 of the ASP.NET Web Helpers Library. If you haven't already added the library, you can install it in your site as a NuGet package. For details, see Installing Helpers in an ASP.NET Web Pages Site. (In the Gallery, search for the `microsoft-web-helpers` package.)
- The jQuery library. Several of the WebMatrix site templates already include jQuery libraries in their *Script* folders. If you do not have these libraries, you can download the latest jQuery library directly from the

jQuery.org site. Or you can create a new site using a template (for example, the **Starter Site** template) and then copy the jQuery files from that site to your current site.

Finally, if you want to use Bing maps, you must first create a (free) account and get a key. To get a key, follow these steps:

1. Create an account on the Bing Maps Developer Account. You must have a Microsoft account (Windows Live ID) as well.

   You can specify that you want to use the key for **Evaluation/Test**. If you are testing the mapping function on your own computer using WebMatrix and IIS Express, go the **Site** workspace and note the URL of your site (for example, `http://localhost:50408`, although your port number will probably be different). You can use this *localhost* address as the site when you register.

2. After you have registered for an account, go to the Bing Maps Account Center and click **Create or view keys**:

3. Record the key that Bing creates.

Creating a Map Based on an Address (Using Google)

The following example shows how to create a page that renders a map based on an address. This example shows how to use Google Maps.

1. Create a file named *MapAddress.cshtml* in the root of the site. This page will generate a map based on an address that you pass to it.

2. Copy the following code into the file, overwriting the existing content.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Map an Address</title>
  <script src="~/Scripts/jquery-1.6.4.min.js" type="text/javascript"></script>
</head>

<body>
    <h1>Map an Address</h1>
    <form method="post">
        <fieldset>
          <div>
            <label for="address">Address:</label>
            <input style="width: 300px" type="text" name="address" value="@Request["address"]"/>
            <input type="submit" value="Map It!" />
           </div>
        </fieldset>
    </form>
    @if(IsPost) {
        @Maps.GetGoogleHtml(Request.Form["address"],
            width: "400",
            height: "400")
    }
</body>
</html>
```

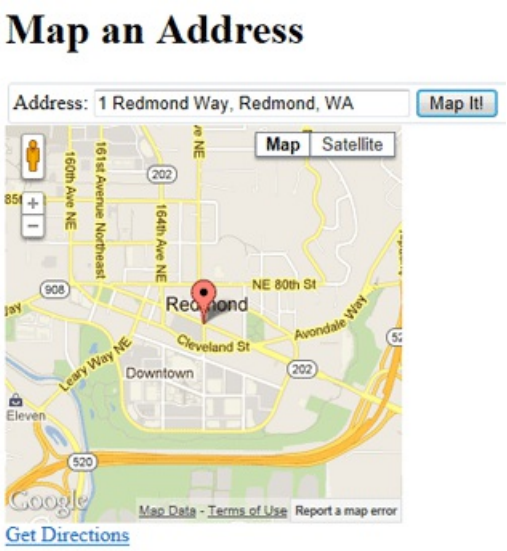
Notice the following features of the page:

- The `<script>` element in the `<head>` element. In the example, the `<script>` element references the *jquery-1.6.4.min.js* file, which is a minified (compressed) version of the jQuery library, version 1.6.4. Note that the reference assumes that the *.js* file is in the *Scripts* folder of your site.

  > **NOTE**
  > If you're using a different version of the jQuery library, just make sure that you're pointing to that version correctly.

- The call to the `@Maps.GetGoogleHtml` in the body of the page. To map an address, you must pass an address string. The methods for the other map engines work in a similar way (`@Maps.GetYahooHtml`, `@Maps.GetMapQuestHtml`).

3. Run the page and enter an address. The page displays a map, based on Google Maps, that shows the location that you specified.



Creating a Map Based on Latitude and Longitude Coordinates (Using Bing)

This example shows how to create a map based on coordinates. This example shows how to use Bing maps and how to include your Bing key. (You can create a map based on coordinates using the other map engines also, without using a Bing key.)

1. Create a file named *MapCoordinates.cshtml* in the root of the site and replace the existing content with the following code and markup:

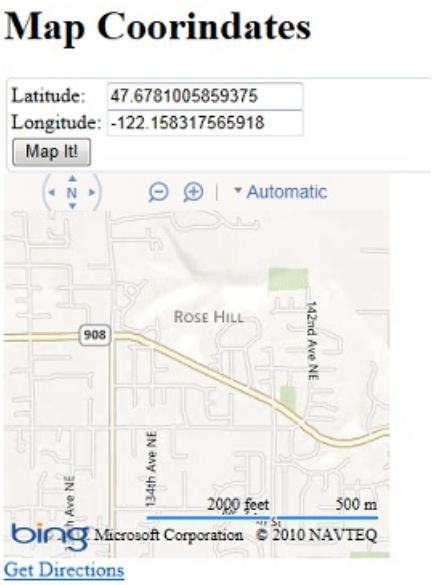
```
<!DOCTYPE html>
<html lang="en"><head>
  <title>Map Coordinates</title>
  <script src="~/Scripts/jquery-1.6.4.min.js" type="text/javascript"></script>
</head>
<body>
  <h1>Map Coordinates</h1>
  <form method="post">
    <fieldset>
    <div>
      <label for="latitude">Latitude:   </label>
        <input type="text" name="latitude" value="@Request["latitude"]"/>
    </div>
    <div>
      <label for="longitude">Longitude:</label>
      <input type="text" name="longitude" value="@Request["longitude"]"/>
    </div>
    <div>
      <input type="submit" value="Map It!" />
    </div>
    </fieldset>
  </form>
  @if(IsPost) {
      @Maps.GetBingHtml(key: "your-key-here",
          latitude: Request["latitude"],
          longitude: Request["longitude"],
          width: "300",
          height: "300"
       )
    }
</body>
</html>
```

2. Replace `your-key-here` with the Bing Maps key that you generated earlier.

3. Run the *MapCoordinates.cshtml* page, enter latitude and longitude coordinates, and then click the **Map It!** button. (If you don't know any coordinates, try the following. This is a location on the Microsoft Redmond campus.)

   - Latitude: 47.6781005859375
   - Longitude: -122.158317565918

   The page is displayed using the coordinates that you specified.



Additional Resources

Microsoft.Maps API Reference

Mobile

2/20/2020 • 2 minutes to read • Edit Online

How to work with Mobile devices in ASP.NET Web Pages applications.

- Rendering ASP.NET Web Pages Sites for Mobile Devices

Rendering ASP.NET Web Pages (Razor) Sites for Mobile Devices

2/20/2020 • 3 minutes to read • Edit Online

by Tom FitzMacken

This article describes how to create pages in an ASP.NET Web Pages (Razor) site that will render appropriately on mobile devices.

What you'll learn:

- How to use a naming convention to specify that a page is designed specifically for mobile devices.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 3

This tutorial also works with ASP.NET Web Pages 2.

ASP.NET Web Pages lets you create custom displays for rendering content on mobile or other devices.

The simplest way to create device-specific page in an ASP.NET Web Pages site is by using a file-naming pattern like this: *FileName.Mobile.cshtml*. You can create two versions of a page (for example, one named *MyFile.cshtml* and one named *MyFile.Mobile.cshtml*). At run time, when a mobile device requests *MyFile.cshtml*, ASP.NET renders the content from *MyFile.Mobile.cshtml*. Otherwise, *MyFile.cshtml* is rendered.

The following example shows how to enable mobile rendering by adding a content page for mobile devices. *Page1.cshtml* contains content plus a navigation sidebar. *Page1.Mobile.cshtml* contains the same content, but omits the sidebar.

1. In an ASP.NET Web Pages site, create a file named *Page1.cshtml* and replace the current content with following markup.

```
<!DOCTYPE html>

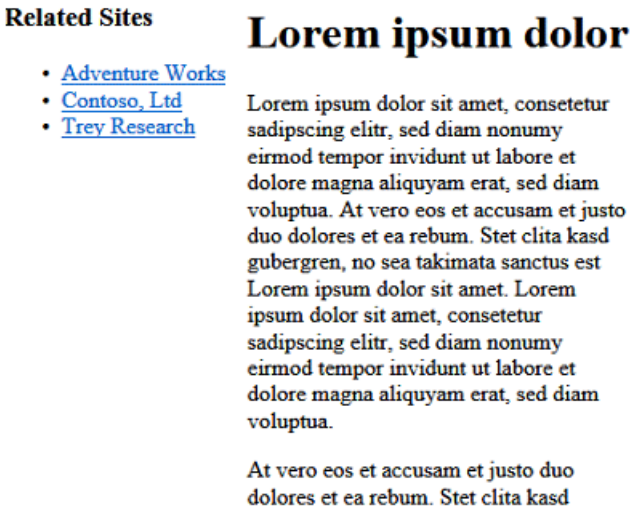
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title></title>
        <style type="text/css">
            #navigation {
                position: absolute;
                top: 0;
                left: 0;
                width: 10em;
            }
            #content {
            margin-left: 13em;
                margin-right: 10em;
            }
        </style>
    </head>
    <body>
        <div id="navigation">
            <h3>Related Sites</h3>
            <ul>
                <li><a href="http://www.adventure-works.com/">Adventure Works</a></li>
                <li><a href="http://www.contoso.com/">Contoso, Ltd</a></li>
                <li><a href="http://www.treyresearch.net/">Trey Research</a></li>
            </ul>
        </div>
        <div id="content">
            <h1>Lorem ipsum dolor</h1>
            <p>Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
            eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.
            At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren,
            no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit
            amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut
            labore et dolore magna aliquyam erat, sed diam voluptua. </p>
            <p>At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd
            gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum
            dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt
            ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam
            et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata
            sanctus est Lorem ipsum dolor sit amet.</p>
        </div>
    </body>
</html>
```

2. Create a file named *Page1.Mobile.cshtml* and replace the existing content with the following markup. Notice that the mobile version of the page omits the navigation section for better rendering on a smaller screen.

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title></title>
        <style type="text/css">
            #content {
            margin-left: 2em;
                margin-right: 5em;
            }
        </style>
    </head>
    <body>
        <div id="content">
            <h1>Lorem ipsum dolor</h1>
            <p>Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
            eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.
            At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren,
            no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit
            amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut
            labore et dolore magna aliquyam erat, sed diam voluptua. </p>
            <p>At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd
            gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum
            dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt
            ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam
            et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata
            sanctus est Lorem ipsum dolor sit amet.</p>
        </div>
    </body>
</html>
```

3. Run a desktop browser and browse to *Page1.cshtml*.



4. Run a mobile browser (or a mobile device emulator) and browse to *Page1.cshtml*. (Notice that you do not include *.mobile.* as part of the URL.) Even though the request is to *Page1.cshtml*, ASP.NET renders *Page1.Mobile.cshtml*.

Lorem ipsum dolor

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

NOTE

To test mobile pages, you can use a mobile device simulator that runs on a desktop computer. This tool lets you test web pages as they would look on mobile devices (that is, typically with a much smaller display area). One example of a simulator is the User Agent Switcher add-on for Mozilla Firefox, which lets you emulate various mobile browsers from a desktop version of Firefox.

Additional Resources

Windows Phone Emulator

Working with Data

2/20/2020 • 2 minutes to read • Edit Online

How to work with databases and files in Web Pages and how to display and update data.

- Working with Data
- Displaying Data in a Chart
- Working with Files

Introduction to Working with a Database in ASP.NET Web Pages (Razor) Sites

2/20/2020 • 25 minutes to read • Edit Online

by Tom FitzMacken

This article describes how to use Microsoft WebMatrix tools to create a database in an ASP.NET Web Pages (Razor) website, and how to create pages that let you display, add, edit, and delete data.

What you'll learn:

- How to create a database.
- How to connect to a database.
- How to display data in a web page.
- How to insert, update, and delete database records.

These are the features introduced in the article:

- Working with a Microsoft SQL Server Compact Edition database.
- Working with SQL queries.
- The `Database` class.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 2
- WebMatrix 2

This tutorial also works with WebMatrix 3. You can use ASP.NET Web Pages 3 and Visual Studio 2013 (or Visual Studio Express 2013 for Web); however, the user interface will be different.

Introduction to Databases

Imagine a typical address book. For each entry in the address book (that is, for each person) you have several pieces of information such as first name, last name, address, email address, and phone number.

A typical way to picture data like this is as a table with rows and columns. In database terms, each row is often referred to as a record. Each column (sometimes referred to as fields) contains a value for each type of data: first name, last name, and so on.

ID	FIRSTNAME	LASTNAME	ADDRESS	EMAIL	PHONE
1	Jim	Abrus	210 100th St SE Orcas WA 98031	jim@contoso.com	555 0100
2	Terry	Adams	1234 Main St. Seattle WA 99011	terry@cohowinery.com	555 0101

For most database tables, the table has to have a column that contains a unique identifier, like a customer number, account number, etc. This is known as the table's *primary key*, and you use it to identify each row in the table. In the example, the ID column is the primary key for the address book.

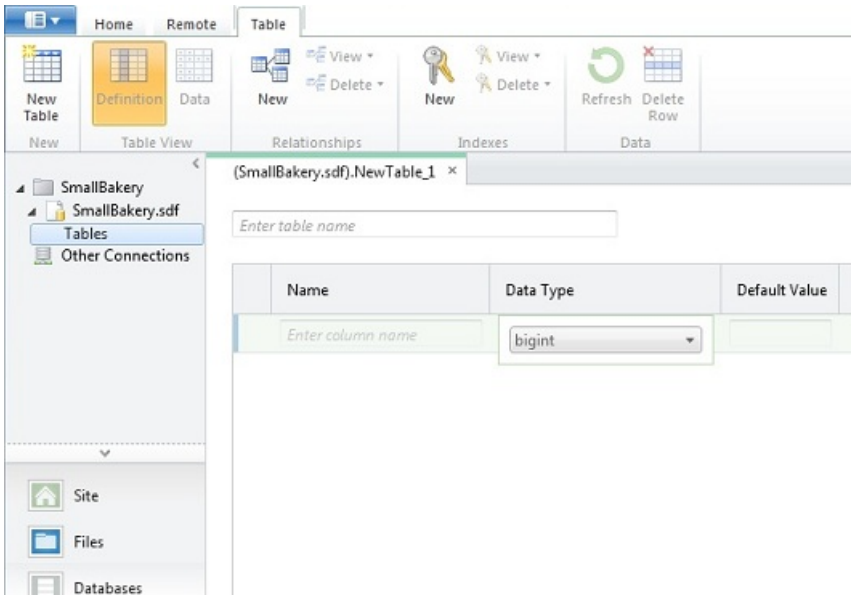
With this basic understanding of databases, you're ready to learn how to create a simple database and perform operations such as adding, modifying, and deleting data.

> **TIP**
>
> **Relational Databases**
>
> You can store data in lots of ways, including text files and spreadsheets. For most business uses, though, data is stored in a relational database.
>
> This article doesn't go very deeply into databases. However, you might find it useful to understand a little about them. In a relational database, information is logically divided into separate tables. For example, a database for a school might contain separate tables for students and for class offerings. The database software (such as SQL Server) supports powerful commands that let you dynamically establish relationships between the tables. For example, you can use the relational database to establish a logical relationship between students and classes in order to create a schedule. Storing data in separate tables reduces the complexity of the table structure and reduces the need to keep redundant data in tables.

Creating a Database

This procedure shows you how to create a database named SmallBakery by using the SQL Server Compact Database design tool that's included in WebMatrix. Although you can create a database using code, it's more typical to create the database and database tables using a design tool like WebMatrix.

1. Start WebMatrix, and on the Quick Start page, click **Site From Template**.
2. Select **Empty Site**, and in the **Site Name** box enter "SmallBakery" and then click **OK**. The site is created and displayed in WebMatrix.
3. In the left pane, click the **Databases** workspace.
4. In the ribbon, click **New Database**. An empty database is created with the same name as your site.
5. In the left pane, expand the **SmallBakery.sdf** node and then click **Tables**.
6. In the ribbon, click **New Table**. WebMatrix opens the table designer.

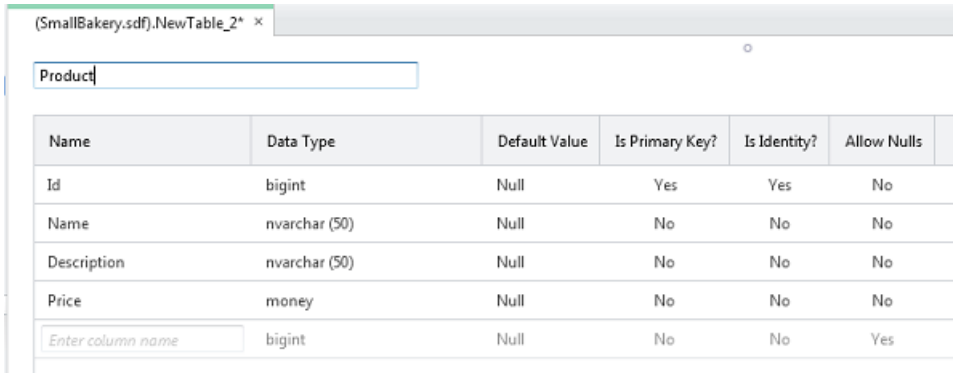


7. Click in the **Name** column and enter "Id".
8. In the **Data Type** column, select **int**.
9. Set the **Is Primary Key?** and **Is Identify?** options to **Yes**.

As the name suggests, **Is Primary Key** tells the database that this will be the table's primary key. **Is Identity** tells the database to automatically create an ID number for every new record and to assign it the next sequential number (starting at 1).

10. Click in the next row. The editor starts a new column definition.
11. For the Name value, enter "Name".
12. For **Data Type**, choose "nvarchar" and set the length to 50. The *var* part of `nvarchar` tells the database that the data for this column will be a string whose size might vary from record to record. (The *n* prefix represents *national*, indicating that the field can hold character data that represents any alphabet or writing system — that is, that the field holds Unicode data.)
13. Set the **Allow Nulls** option to **No**. This will enforce that the *Name* column is not left blank.
14. Using this same process, create a column named *Description*. Set **Data Type** to "nvarchar" and 50 for the length, and set **Allow Nulls** to false.
15. Create a column named *Price*. Set **Data Type to "money"** and set **Allow Nulls** to false.
16. In the box at the top, name the table "Product".

    When you're done, the definition will look like this:



17. Press Ctrl+S to save the table.

Adding Data to the Database

Now you can add some sample data to your database that you'll work with later in the article.

1. In the left pane, expand the **SmallBakery.sdf** node and then click **Tables**.
2. Right-click the Product table and then click **Data**.
3. In the edit pane, enter the following records:

NAME	DESCRIPTION	PRICE
Bread	Baked fresh every day.	2.99
Strawberry Shortcake	Made with organic strawberries from our garden.	9.99
Apple Pie	Second only to your mom's pie.	12.99
Pecan Pie	If you like pecans, this is for you.	10.99

NAME	DESCRIPTION	PRICE
Lemon Pie	Made with the best lemons in the world.	11.99
Cupcakes	Your kids and the kid in you will love these.	7.99

Remember that you don't have to enter anything for the *Id* column. When you created the *Id* column, you set its **Is Identity** property to true, which causes it to automatically be filled in.

When you're finished entering the data, the table designer will look like this:

Table - (SmallBakery.sdf).Product

Id	Name	Description	Price
1	Bread	Baked fresh every day	2.99
2	Strawberry Shortcake	Made with organic strawberries from our garden	9.99
3	Apple Pie	Second only to your mom's pie	12.99
4	Pecan Pie	If you like pecans, this is for you.	10.99
5	Lemon Pie	Made with the best lemons in the world.	11.99
6	Cupcakes	Your kids and the kid in you will love these.	7.99

4. Close the tab that contains the database data.

Displaying Data from a Database

Once you've got a database with data in it, you can display the data in an ASP.NET web page. To select the table rows to display, you use a SQL statement, which is a command that you pass to the database.

1. In the left pane, click the **Files** workspace.
2. In the root of the website, create a new CSHTML page named *ListProducts.cshtml*.
3. Replace the existing markup with the following:

```
@{
    var db = Database.Open("SmallBakery");
    var selectQueryString = "SELECT * FROM Product ORDER BY Name";
 }
<!DOCTYPE html>
<html>
 <head>
   <title>Small Bakery Products</title>
   <style>
       table, th, td {
         border: solid 1px #bbbbbb;
         border-collapse: collapse;
         padding: 2px;
       }
    </style>
 </head>
 <body>
   <h1>Small Bakery Products</h1>
   <table>
       <thead>
           <tr>
               <th>Id</th>
               <th>Product</th>
               <th>Description</th>
       <th>Price</th>
           </tr>
       </thead>
       <tbody>
           @foreach(var row in db.Query(selectQueryString)){
            <tr>
               <td>@row.Id</td>
                   <td>@row.Name</td>
                   <td>@row.Description</td>
                   <td>@row.Price</td>
            </tr>
           }
       </tbody>
   </table>
 </body>
</html>
```

In the first code block, you open the *SmallBakery.sdf* file (database) that you created earlier. The `Database.Open` method assumes that the *.sdf* file is in your website's *App_Data* folder. (Notice that you don't need to specify the *.sdf* extension — in fact, if you do, the `Open` method won't work.)

> **NOTE**
>
> The *App_Data* folder is a special folder in ASP.NET that's used to store data files. For more information, see Connecting to a Database later in this article.

You then make a request to query the database using the following SQL `Select` statement:

```
SELECT * FROM Product ORDER BY Name
```

In the statement, `Product` identifies the table to query. The `*` character specifies that the query should return all the columns from the table. (You could also list columns individually, separated by commas, if you wanted to see only some of the columns.) The `Order By` clause indicates how the data should be sorted — in this case, by the *Name* column. This means that the data is sorted alphabetically based on the value of the *Name* column for each row.

In the body of the page, the markup creates an HTML table that will be used to display the data. Inside the `<tbody>` element, you use a `foreach` loop to individually get each data row that's returned by the query. For each data row, you create an HTML table row (`<tr>` element). Then you create HTML table cells (`<td>` elements) for each column. Each time you go through the loop, the next available row from the database is in the `row` variable (you set this up in the `foreach` statement). To get an individual column from the row, you can use `row.Name` or `row.Description` or whatever the name is of the column you want.

4. Run the page in a browser. (Make sure the page is selected in the **Files** workspace before you run it.) The page displays a list like the following:

Id	Product	Description	Price
3	Apple Pie	Second only to your mom's pie.	12.99
1	Bread	Baked fresh every day	2.99
6	Cupcakes	Your kids and the kid in you will love these.	7.99
5	Lemon Pie	Made with the best lemons in the world.	11.99
4	Pecan Pie	If you like pecans, this is for you.	10.99
2	Strawberry Shortcake	Made with organic strawberries from our garden.	9.99

TIP

Structured Query Language (SQL)

SQL is a language that's used in most relational databases for managing data in a database. It includes commands that let you retrieve data and update it, and that let you create, modify, and manage database tables. SQL is different than a programming language (like the one you're using in WebMatrix) because with SQL, the idea is that you tell the database what you want, and it's the database's job to figure out how to get the data or perform the task. Here are examples of some SQL commands and what they do:

```
SELECT Id, Name, Price FROM Product WHERE Price > 10.00 ORDER BY Name
```

This fetches the *Id*, *Name*, and *Price* columns from records in the *Product* table if the value of *Price* is more than 10, and returns the results in alphabetical order based on the values of the *Name* column. This command will return a result set that contains the records that meet the criteria, or an empty set if no records match.

```
INSERT INTO Product (Name, Description, Price) VALUES ("Croissant", "A flaky delight", 1.99)
```

This inserts a new record into the *Product* table, setting the *Name* column to "Croissant", the *Description* column to "A flaky delight", and the price to 1.99.

```
DELETE FROM Product WHERE ExpirationDate < "01/01/2008"
```

This command deletes records in the *Product* table whose expiration date column is earlier than January 1, 2008. (This assumes that the *Product* table has such a column, of course.) The date is entered here in MM/DD/YYYY format, but it should be entered in the format that's used for your locale.

The `Insert Into` and `Delete` commands don't return result sets. Instead, they return a number that tells you how many records were affected by the command.

For some of these operations (like inserting and deleting records), the process that's requesting the operation has to have appropriate permissions in the database. This is why for production databases you often have to supply a username and password when you connect to the database.

There are dozens of SQL commands, but they all follow a pattern like this. You can use SQL commands to create database tables, count the number of records in a table, calculate prices, and perform many more operations.

Inserting Data in a Database

This section shows how to create a page that lets users add a new product to the *Product* database table. After a new product record is inserted, the page displays the updated table using the *ListProducts.cshtml* page that you

created in the previous section.

The page includes validation to make sure that the data that the user enters is valid for the database. For example, code in the page makes sure that a value has been entered for all required columns.

1. In the website, create a new CSHTML file named *InsertProducts.cshtml*.

2. Replace the existing markup with the following:

```
@{
    Validation.RequireField("Name", "Product name is required.");
    Validation.RequireField("Description", "Product description is required.");
    Validation.RequireField("Price", "Product price is required.");

    var db = Database.Open("SmallBakery");
    var Name = Request.Form["Name"];
    var Description = Request.Form["Description"];
    var Price = Request.Form["Price"];

    if (IsPost && Validation.IsValid()) {
        // Define the insert query. The values to assign to the
        // columns in the Product table are defined as parameters
        // with the VALUES keyword.
        if(ModelState.IsValid) {
            var insertQuery = "INSERT INTO Product (Name, Description, Price) " +
                "VALUES (@0, @1, @2)";
            db.Execute(insertQuery, Name, Description, Price);
            // Display the page that lists products.
            Response.Redirect("~/ListProducts");
        }
    }
}

<!DOCTYPE html>
<html>
<head>
 <title>Add Products</title>
 <style type="text/css">
    label {float:left; width: 8em; text-align: right;
           margin-right: 0.5em;}
    fieldset {padding: 1em; border: 1px solid; width: 50em;}
    legend {padding: 2px 4px; border: 1px solid; font-weight:bold;}
    .validation-summary-errors {font-weight:bold; color:red;
           font-size: 11pt;}
 </style>
</head>
<body>
 <h1>Add New Product</h1>

 @Html.ValidationSummary("Errors with your submission:")

 <form method="post" action="">
   <fieldset>
     <legend>Add Product</legend>
     <div>
       <label>Name:</label>
       <input name="Name" type="text" size="50" value="@Name" />
     </div>
     <div>
       <label>Description:</label>
       <input name="Description" type="text" size="50"
           value="@Description" />
     </div>
     <div>
       <label>Price:</label>
       <input name="Price" type="text" size="50" value="@Price" />
     </div>
     <div>
       <label> </label>
       <input type="submit" value="Insert" class="submit" />
     </div>
   </fieldset>
 </form>
</body>
</html>
```

The body of the page contains an HTML form with three text boxes that let users enter a name, description, and price. When users click the **Insert** button, the code at the top of the page opens a connection to the *SmallBakery.sdf* database. You then get the values that the user has submitted by using the `Request` object and assign those values to local variables.

To validate that the user entered a value for each required column, you register each `<input>` element that you want to validate:

```
Validation.RequireField("Name", "Product name is required.");
Validation.RequireField("Description", "Product description is required.");
Validation.RequireField("Price", "Product price is required.");
```

The `Validation` helper checks that there is a value in each of the fields that you've registered. You can test whether all the fields passed validation by checking `Validation.IsValid()`, which you typically do before you process the information you get from the user:

```
if (IsPost && Validation.IsValid()) {
    // Process information here
}
```

(The `&&` operator means AND — this test is *If this is a form submission AND all the fields have passed validation*.)

If all the columns validated (none were empty), you go ahead and create a SQL statement to insert the data and then execute it as shown next:

```
var insertQuery =
    "INSERT INTO Product (Name, Description, Price) VALUES (@0, @1, @2)";
```

For the values to insert, you include parameter placeholders (`@0`, `@1`, `@2`).

> **NOTE**
> As a security precaution, always pass values to a SQL statement using parameters, as you see in the preceding example. This gives you a chance to validate the user's data, plus it helps protect against attempts to send malicious commands to your database (sometimes referred to as SQL injection attacks).

To execute the query, you use this statement, passing to it the variables that contain the values to substitute for the placeholders:

```
db.Execute(insertQuery, Name, Description, Price);
```

After the `Insert Into` statement has executed, you send the user to the page that lists the products using this line:

```
Response.Redirect("~/ListProducts");
```

If validation didn't succeed, you skip the insert. Instead, you have a helper in the page that can display the accumulated error messages (if any):

```
@Html.ValidationSummary("Errors with your submission:")
```

Notice that the style block in the markup includes a CSS class definition named `.validation-summary-errors`. This is the name of the CSS class that's used by default for the `<div>` element that contains any validation errors. In this case, the CSS class specifies that validation summary errors are displayed in red and in bold, but you can define the `.validation-summary-errors` class to display any formatting you like.

Testing the Insert Page

1. View the page in a browser. The page displays a form that's similar to the one that's shown in the following illustration.

2. Enter values for all the columns, but make sure that you leave the *Price* column blank.

3. Click **Insert**. The page displays an error message, as shown in the following illustration. (No new record is created.)

4. Fill the form out completely, and then click **Insert**. This time, the *ListProducts.cshtml* page is displayed and shows the new record.

Updating Data in a Database

After data has been entered into a table, you might need to update it. This procedure shows you how to create two pages that are similar to the ones you created for data insertion earlier. The first page displays products and lets users select one to change. The second page lets the users actually make the edits and save them.

> **NOTE**
>
> **Important** In a production website, you typically restrict who's allowed to make changes to the data. For information about how to set up membership and about ways to authorize users to perform tasks on the site, see Adding Security and Membership to an ASP.NET Web Pages Site.

1. In the website, create a new CSHTML file named *EditProducts.cshtml*.

2. Replace the existing markup in the file with the following:

```
@{
    var db = Database.Open("SmallBakery");
    var selectQueryString = "SELECT * FROM Product ORDER BY Name";

}
<!DOCTYPE html>
<html>
<head>
    <title>Edit Products</title>
    <style type="text/css">
        table, th, td {
          border: solid 1px #bbbbbb;
          border-collapse: collapse;
          padding: 2px;
        }
    </style>
</head>
<body>
    <h1>Edit Small Bakery Products</h1>
    <table>
      <thead>
        <tr>
          <th> </th>
          <th>Name</th>
          <th>Description</th>
          <th>Price</th>
        </tr>
      </thead>
      <tbody>
        @foreach (var row in db.Query(selectQueryString)) {
          <tr>
            <td><a href="@Href("~/UpdateProducts", row.Id)">Edit</a></td>
            <td>@row.Name</td>
            <td>@row.Description</td>
            <td>@row.Price</td>
          </tr>
        }
      </tbody>
    </table>
</body>
</html>
```

The only difference between this page and the *ListProducts.cshtml* page from earlier is that the HTML table in this page includes an extra column that displays an **Edit** link. When you click this link, it takes you to the *UpdateProducts.cshtml* page (which you'll create next) where you can edit the selected record.

Look at the code that creates the **Edit** link:

```
<a href="@Href("~/UpdateProducts", row.Id)">Edit</a></td>
```

This creates an HTML `<a>` element whose `href` attribute is set dynamically. The `href` attribute specifies the page to display when the user clicks the link. It also passes the `Id` value of the current row to the link. When the page runs, the page source might contain links like these:

```
<a href="UpdateProducts/1">Edit</a></td>
<a href="UpdateProducts/2">Edit</a></td>
<a href="UpdateProducts/3">Edit</a></td>
```

Notice that the `href` attribute is set to `UpdateProducts/n`, where *n* is a product number. When a user clicks one of these links, the resulting URL will look something like this:

```
http://localhost:18816/UpdateProducts/6
```

In other words, the product number to be edited will be passed in the URL.

3. View the page in a browser. The page displays the data in a format like this:

	Name	Description	Price
Edit	Apple Pie	Second only to your mom's pie.	12.99
Edit	Bread	Baked fresh every day.	2.99
Edit	Cherry Pie	Made with organic cherries from our garden.	8.99
Edit	Cupcakes	Your kids and the kid in you will love these.	7.99
Edit	Lemon Pie	Made with the best lemmons in the world.	11.99
Edit	Pecan Pie	If you like pecans this is fro you.	10.99
Edit	Strawberry Shortcake	Made with organic strawberries from our garden.	9.99

Next, you'll create the page that lets users actually update the data. The update page includes validation to validate the data that the user enters. For example, code in the page makes sure that a value has been entered for all required columns.

4. In the website, create a new CSHTML file named *UpdateProducts.cshtml*.

5. Replace the existing markup in the file with the following.

```
@{
    Validation.RequireField("Name", "Product name is required.");
    Validation.RequireField("Description", "Product description is required.");
    Validation.RequireField("Price", "Product price is required.");

    var Name = "";
    var Description = "";
    var Price = Decimal.Zero;

    var ProductId  = UrlData[0];
    if (ProductId.IsEmpty()) {
         Response.Redirect("~/EditProducts");
    }

    var db = Database.Open("SmallBakery");

    if (IsPost && Validation.IsValid()) {
        var updateQueryString =
            "UPDATE Product SET Name=@0, Description=@1, Price=@2 WHERE Id=@3" ;
        Name = Request["Name"];
        Description = Request["Description"];
        Price = Request["Price"].AsDecimal();
        db.Execute(updateQueryString, Name, Description, Price, ProductId);
        Response.Redirect(@Href("~/EditProducts"));
    }
    else {
        var selectQueryString = "SELECT * FROM Product WHERE Id=@0";

        var row = db.QuerySingle(selectQueryString, ProductId);
        Name = row.Name;
        Description = row.Description;
        Price = row.Price;
    }

}

<!DOCTYPE html>
<html>
<head>
  <title>Add Products</title>
  <style type="text/css">
     label { float: left; width: 8em; text-align: right;
             margin-right: 0.5em;}
```

```
            margin-right: 0.5em;}
      fieldset { padding: 1em; border: 1px solid; width: 35em;}
      legend { padding: 2px 4px;  border: 1px solid; font-weight: bold;}
      .validation-summary-errors {font-weight:bold; color:red; font-size:11pt;}
    </style>
  </head>
  <body>
    <h1>Update Product</h1>
     @Html.ValidationSummary("Errors with your submission:")
     <form method="post" action="">
       <fieldset>
         <legend>Update Product</legend>
         <div>
           <label>Name:</label>
           <input name="Name" type="text" size="50" value="@Name" />
         </div>
         <div>
           <label>Description:</label>
           <input name="Description" type="text" size="50"
              value="@Description" />
         </div>
         <div>
            <label>Price:</label>
            <input name="Price" type="text" size="50" value="@Price" />
         </div>
         <div>
            <label> </label>
            <input type="submit" value="Update" class="submit" />
         </div>
      </fieldset>
    </form>
  </body>
  </html>
```

The body of the page contains an HTML form where a product is displayed and where users can edit it. To get the product to display, you use this SQL statement:

```
SELECT * FROM Product WHERE Id=@0
```

This will select the product whose ID matches the value that's passed in the `@0` parameter. (Because *Id* is the primary key and therefore must be unique, only one product record can ever be selected this way.) To get the ID value to pass to this `Select` statement, you can read the value that's passed to the page as part of the URL, using the following syntax:

```
var ProductId  = UrlData[0];
```

To actually fetch the product record, you use the `QuerySingle` method, which will return just one record:

```
var row = db.QuerySingle(selectQueryString, ProductId);
```

The single row is returned into the `row` variable. You can get data out of each column and assign it to local variables like this:

```
var Name = row.Name;
var Description = row.Description;
var Price = row.Price;
```

In the markup for the form, these values are displayed automatically in individual text boxes by using embedded code like the following:

```
<input name="Name" type="text" size="50" value="@Name" />
```

That part of the code displays the product record to be updated. Once the record has been displayed, the user can edit individual columns.

When the user submits the form by clicking the **Update** button, the code in the `if(IsPost)` block runs. This gets the user's values from the `Request` object, stores the values in variables, and validates that each column has been filled in. If validation passes, the code creates the following SQL Update statement:

```
UPDATE Product SET Name=@0, Description=@1, Price=@2, WHERE ID=@3
```

In a SQL `Update` statement, you specify each column to update and the value to set it to. In this code, the values are specified using the parameter placeholders `@0`, `@1`, `@2`, and so on. (As noted earlier, for security, you should always pass values to a SQL statement by using parameters.)

When you call the `db.Execute` method, you pass the variables that contain the values in the order that corresponds to the parameters in the SQL statement:

```
db.Execute(updateQueryString, Name, Description, Price, ProductId);
```

After the `Update` statement has been executed, you call the following method in order to redirect the user back to the edit page:

```
Response.Redirect(@Href("~/EditProducts"));
```

The effect is that the user sees an updated listing of the data in the database and can edit another product.

6. Save the page.

7. Run the *EditProducts.cshtml* page (not the update page) and then click **Edit** to select a product to edit. The *UpdateProducts.cshtml* page is displayed, showing the record you selected.

8. Make a change and click **Update**. The products list is shown again with your updated data.

Deleting Data in a Database

This section shows how to let users delete a product from the *Product* database table. The example consists of two pages. In the first page, users select a record to delete. The record to be deleted is then displayed in a second page that lets them confirm that they want to delete the record.

> **NOTE**
>
> **Important** In a production website, you typically restrict who's allowed to make changes to the data. For information about how to set up membership and about ways to authorize user to perform tasks on the site, see Adding Security and Membership to an ASP.NET Web Pages Site.

1. In the website, create a new CSHTML file named *ListProductsForDelete.cshtml*.

2. Replace the existing markup with the following:

```
@{
  var db = Database.Open("SmallBakery");
  var selectQueryString = "SELECT * FROM Product ORDER BY Name";
}
<!DOCTYPE html>
<html>
<head>
    <title>Delete a Product</title>
    <style>
        table, th, td {
          border: solid 1px #bbbbbb;
          border-collapse: collapse;
          padding: 2px;
        }
     </style>
</head>
<body>
  <h1>Delete a Product</h1>
  <form method="post" action="" name="form">
    <table border="1">
      <thead>
        <tr>
          <th> </th>
          <th>Name</th>
          <th>Description</th>
          <th>Price</th>
        </tr>
      </thead>
      <tbody>
        @foreach (var row in db.Query(selectQueryString)) {
          <tr>
            <td><a href="@Href("~/DeleteProduct", row.Id)">Delete</a></td>
            <td>@row.Name</td>
            <td>@row.Description</td>
            <td>@row.Price</td>
          </tr>
        }
      </tbody>
    </table>
  </form>
</body>
</html>
```

This page is similar to the *EditProducts.cshtml* page from earlier. However, instead of displaying an **Edit** link for each product, it displays a **Delete** link. The **Delete** link is created using the following embedded code in the markup:

```
<a href="@Href("~/DeleteProduct", row.Id)">Delete</a>
```

This creates a URL that looks like this when users click the link:

`http://<server>/DeleteProduct/4`

The URL calls a page named *DeleteProduct.cshtml* (which you'll create next) and passes it the ID of the product to delete (here, 4).

3. Save the file, but leave it open.

4. Create another CHTML file named *DeleteProduct.cshtml*. Replace the existing content with the following:

```
@{
  var db = Database.Open("SmallBakery");
  var ProductId = UrlData[0];
  if (ProductId.IsEmpty()) {
    Response.Redirect("~/ListProductsForDelete");
  }
  var prod = db.QuerySingle("SELECT * FROM PRODUCT WHERE ID = @0", ProductId);
  if( IsPost && !ProductId.IsEmpty()) {
    var deleteQueryString = "DELETE FROM Product WHERE Id=@0";
    db.Execute(deleteQueryString, ProductId);
    Response.Redirect("~/ListProductsForDelete");
  }
}

<!DOCTYPE html>
<html>
<head>
    <title>Delete Product</title>
</head>
<body>
  <h1>Delete Product - Confirmation</h1>
  <form method="post" action="" name="form">
    <p>Are you sure you want to delete the following product?</p>

    <p>Name: @prod.Name <br />
       Description: @prod.Description <br />
       Price: @prod.Price</p>
    <p><input type="submit" value="Delete" /></p>
  </form>
</body>
</html>
```

This page is called by *ListProductsForDelete.cshtml* and lets users confirm that they want to delete a product. To list the product to be deleted, you get the ID of the product to delete from the URL using the following code:

```
var ProductId = UrlData[0];
```

The page then asks the user to click a button to actually delete the record. This is an important security measure: when you perform sensitive operations in your website like updating or deleting data, these operations should always be done using a POST operation, not a GET operation. If your site is set up so that a delete operation can be performed using a GET operation, anyone can pass a URL like `http://<server>/DeleteProduct/4` and delete anything they want from your database. By adding the confirmation and coding the page so that the deletion can be performed only by using a POST, you add a measure of security to your site.

The actual delete operation is performed using the following code, which first confirms that this is a post operation and that the ID isn't empty:

```
if( IsPost && !ProductId.IsEmpty()) {
    var deleteQueryString = "DELETE FROM Product WHERE Id=@0";
    db.Execute(deleteQueryString, ProductId);
    Response.Redirect("~/ListProductsForDelete");
}
```

The code runs a SQL statement that deletes the specified record and then redirects the user back to the listing page.

5. Run *ListProductsForDelete.cshtml* in a browser.

	Name	Description	Price
Delete	Apple Pie	Second only to your mom's pie.	12.99
Delete	Bread	Baked fresh every day.	2.99
Delete	Cherry Pie	Made with organic cherries from our garden.	8.99
Delete	Cupcakes	Your kids and the kid in you will love these.	7.99
Delete	Lemon Pie	Made with the best lemmons in the world.	11.99
Delete	Pecan Pie	If you like pecans this is fro you.	10.99
Delete	Strawberry Shortcake	Made with organic strawberries from our garden.	9.99

6. Click the **Delete** link for one of the products. The *DeleteProduct.cshtml* page is displayed to confirm that you want to delete that record.

7. Click the **Delete** button. The product record is deleted and the page is refreshed with an updated product listing.

TIP

Connecting to a Database

You can connect to a database in two ways. The first is to use the `Database.Open` method and to specify the name of the database file (less the *.sdf* extension):

```
var db = Database.Open("SmallBakery");
```

The `Open` method assumes that the *.sdf* file is in the website's *App_Data* folder. This folder is designed specifically for holding data. For example, it has appropriate permissions to allow the website to read and write data, and as a security measure, WebMatrix does not allow access to files from this folder.

The second way is to use a connection string. A connection string contains information about how to connect to a database. This can include a file path, or it can include the name of a SQL Server database on a local or remote server, along with a user name and password to connect to that server. (If you keep data in a centrally managed version of SQL Server, such as on a hosting provider's site, you always use a connection string to specify the database connection information.)

In WebMatrix, connection strings are usually stored in an XML file named *Web.config*. As the name implies, you can use a *Web.config* file in the root of your website to store the site's configuration information, including any connection strings that your site might require. An example of a connection string in a *Web.config* file might look like the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
   <add
     name="SQLServerConnectionString"
     connectionString= "server=myServer;database=myDatabase;uid=username;pwd=password"
     providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

In the example, the connection string points to a database in an instance of SQL Server that's running on a server somewhere (as opposed to a local *.sdf* file). You would need to substitute the appropriate names for `myServer` and `myDatabase`, and specify SQL Server login values for `username` and `password`. (The username and password values are not necessarily the same as your Windows credentials or as the values that your hosting provider has given you for logging in to their servers. Check with the administrator for the exact values you need.)

The `Database.Open` method is flexible, because it lets you pass either the name of a database *.sdf* file or the name of a connection string that's stored in the *Web.config* file. The following example shows how to connect to the database using the connection string illustrated in the previous example:

```
@{
    var db = Database.Open("SQLServerConnectionString");
}
```

As noted, the `Database.Open` method lets you pass either a database name or a connection string, and it'll figure out which to use. This is very useful when you deploy (publish) your website. You can use an *.sdf* file in the *App_Data* folder when you're developing and testing your site. Then when you move your site to a production server, you can use a connection string in the *Web.config* file that has the same name as your *.sdf* file but that points to the hosting provider's database — all without having to change your code.

Finally, if you want to work directly with a connection string, you can call the `Database.OpenConnectionString` method and pass it the actual connection string instead of just the name of one in the *Web.config* file. This might be useful in situations where for some reason you don't have access to the connection string (or values in it, such as the *.sdf* file name) until the page is running. However, for most scenarios, you can use `Database.Open` as described in this article.

Additional Resources

- SQL Server Compact

- Connecting to a SQL Server or MySQL Database in WebMatrix
- Validating User Input in ASP.NET Web Pages Sites

Displaying Data in a Chart with ASP.NET Web Pages (Razor)

4/10/2020 • 20 minutes to read • Edit Online

by Microsoft

This article explains how to use a chart to display data in an ASP.NET Web Pages (Razor) website by using the `Chart` helper.

What you'll learn:

- How to display data in a chart.
- How to style charts using built-in themes.
- How to save charts and how to cache them for better performance.

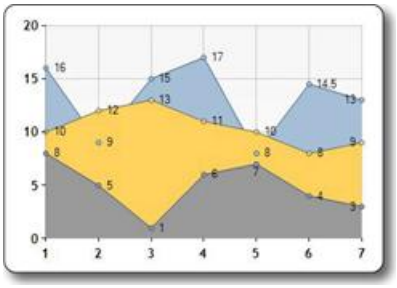
These are the ASP.NET programming features introduced in the article:

- The `Chart` helper.

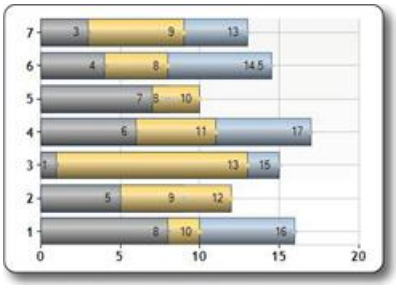
> **NOTE**
>
> The information in this article applies to ASP.NET Web Pages 1.0 and Web Pages 2.

The Chart Helper

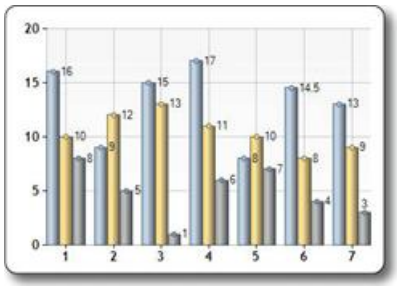
When you want to display your data in graphical form, you can use `Chart` helper. The `Chart` helper can render an image that displays data in a variety of chart types. It supports many options for formatting and labeling. The `Chart` helper can render more than 30 types of charts, including all the types of charts that you might be familiar with from Microsoft Excel or other tools — area charts, bar charts, column charts, line charts, and pie charts, along with more specialized charts like stock charts.



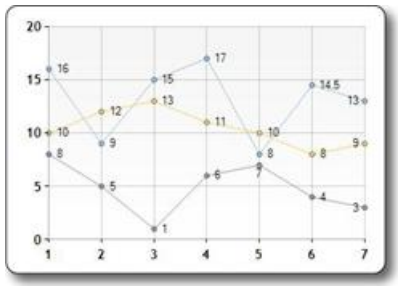
AREA CHART



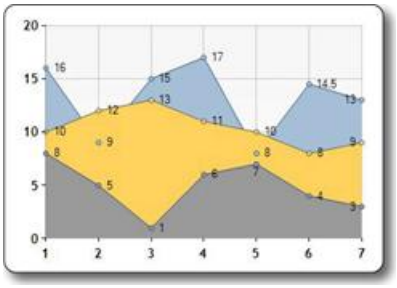
BAR CHART



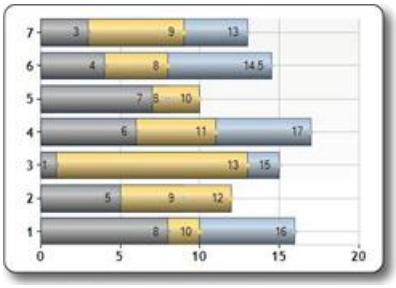
Column chart



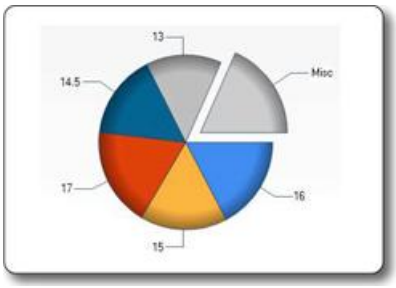
Line chart



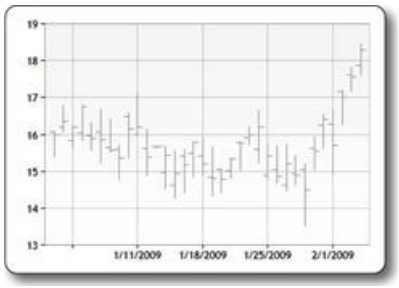
AREA CHART



BAR CHART



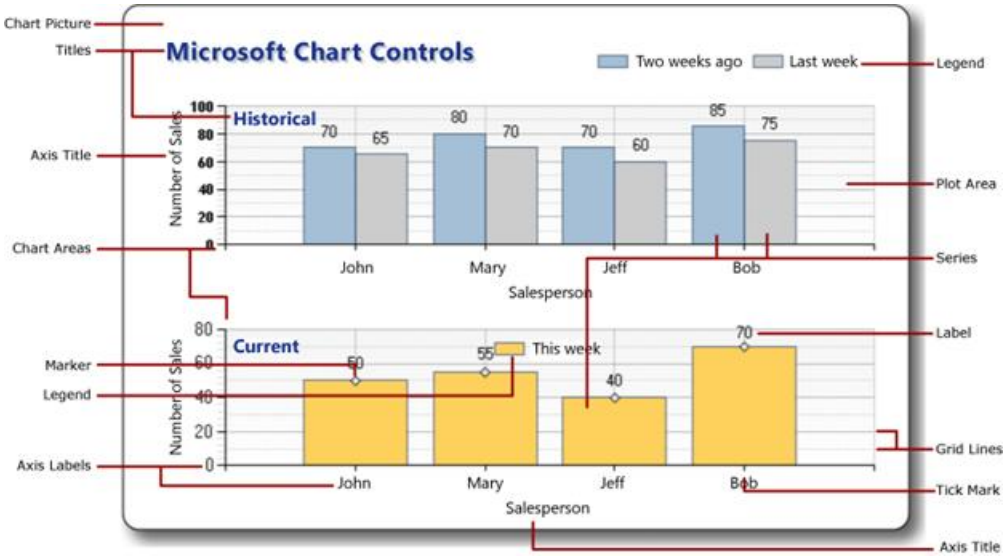
Pie chart



Stock chart

Chart Elements

Charts show data and additional elements like legends, axes, series, and so on. The following picture shows many of the chart elements that you can customize when you use the `Chart` helper. This article shows you how to set some (not all) of these elements.



Creating a Chart from Data

The data you display in a chart can be from an array, from the results returned from a database, or from data that's in an XML file.

Using an Array

As explained in Introduction to ASP.NET Web Pages Programming Using the Razor Syntax, an array lets you store a collection of similar items in a single variable. You can use arrays to contain the data that you want to include in your chart.

This procedure shows how you can create a chart from data in arrays, using the default chart type. It also shows how to display the chart within the page.

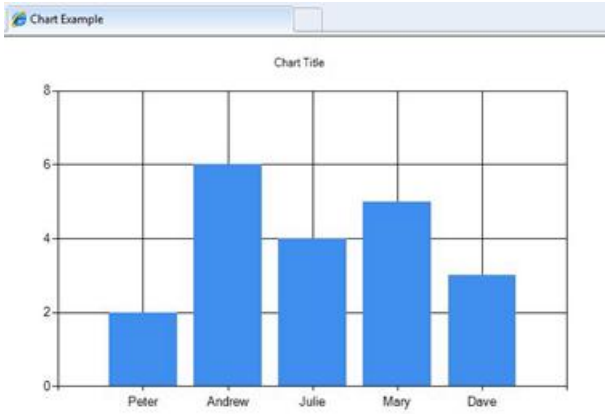
1. Create a new file named *ChartArrayBasic.cshtml*.
2. Replace the existing content with the following:

```
@{
    var myChart = new Chart(width: 600, height: 400)
        .AddTitle("Chart Title")
        .AddSeries(
            name: "Employee",
            xValue: new[] {  "Peter", "Andrew", "Julie", "Mary", "Dave" },
            yValues: new[] { "2", "6", "4", "5", "3" })
        .Write();
}
```

The code first creates a new chart and sets its width and height. You specify the chart title by using the `AddTitle` method. To add data, you use the `AddSeries` method. In this example, you use the `name`, `xValue`, and `yValues` parameters of the `AddSeries` method. The `name` parameter is displayed in the chart legend. The `xValue` parameter contains an array of data that's displayed along the horizontal axis of the chart. The `yValues` parameter contains an array of data that's used to plot the vertical points of the chart.

The `Write` method actually renders the chart. In this case, because you didn't specify a chart type, the `Chart` helper renders its default chart, which is a column chart.

3. Run the page in the browser. The browser displays the chart.



Using a Database Query for Chart Data

If the information you want to chart is in a database, you can run a database query and then use data from the results to create the chart. This procedure shows you how to read and display the data from the database created in the article Introduction to Working with a Database in ASP.NET Web Pages Sites.

1. Add an *App_Data* folder to the root of the website if the folder does not already exist.

2. In the *App_Data* folder, add the database file named *SmallBakery.sdf* that's described in Introduction to Working with a Database in ASP.NET Web Pages Sites.

3. Create a new file named *ChartDataQuery.cshtml*.

4. Replace the existing content with the following:

```
@{
    var db = Database.Open("SmallBakery");
    var data = db.Query("SELECT Name, Price FROM Product");
    var myChart = new Chart(width: 600, height: 400)
        .AddTitle("Product Sales")
        .DataBindTable(dataSource: data, xField: "Name")
        .Write();
}
```

The code first opens the SmallBakery database and assigns it to a variable named `db`. This variable

represents a `Database` object that can be used to read from and write to the database. Next, the code runs a SQL query to get the name and price of each product. The code creates a new chart and passes the database query to it by calling the chart's `DataBindTable` method. This method takes two parameters: the `dataSource` parameter is for the data from the query, and the `xField` parameter lets you set which data column is used for the chart's x-axis.

As an alternative to using the `DataBindTable` method, you can use the `AddSeries` method of the `Chart` helper. The `AddSeries` method lets you set the `xValue` and `yValues` parameters. For example, instead of using the `DataBindTable` method like this:

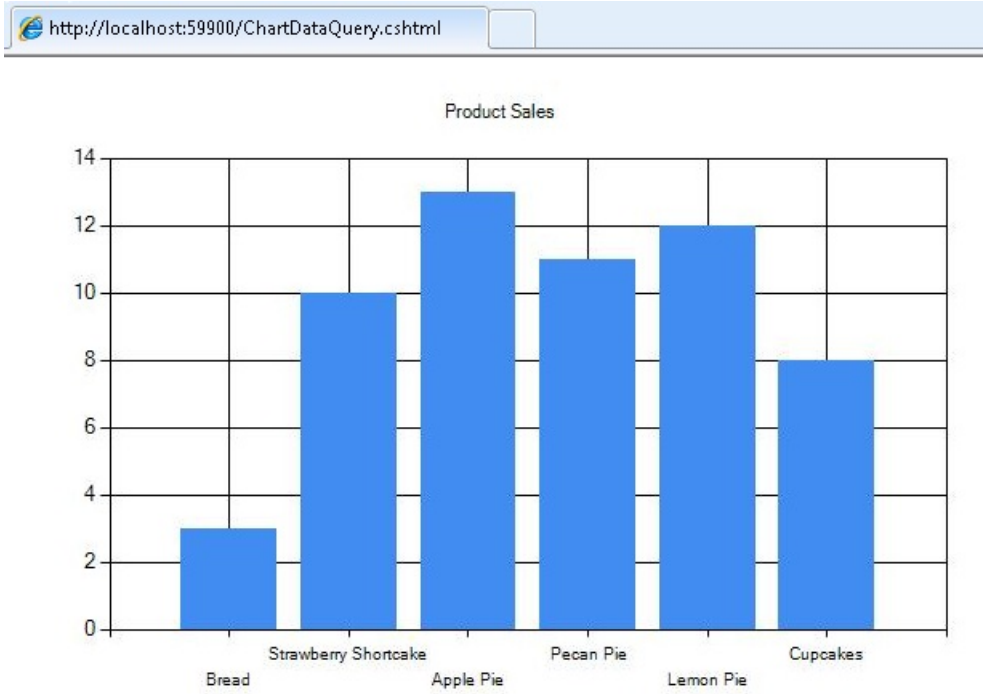
```
.DataBindTable(data, "Name")
```

You can use the `AddSeries` method like this:

```
.AddSeries("Default",
    xValue: data, xField: "Name",
    yValues: data, yFields: "Price")
```

Both render the same results. The `AddSeries` method is more flexible because you can specify the chart type and data more explicitly, but the `DataBindTable` method is easier to use if you don't need the extra flexibility.

5. Run the page in a browser.



Using XML Data

The third option for charting is to use an XML file as the data for the chart. This requires that the XML file also have a schema file (*.xsd* file) that describes the XML structure. This procedure shows you how to read data from an XML file.

1. In the *App_Data* folder, create a new XML file named *data.xml*.

2. Replace the existing XML with the following, which is some XML data about employees in a fictional company.

```
<?xml version="1.0" standalone="yes" ?>
<NewDataSet xmlns="http://tempuri.org/data.xsd">
    <Employee>
        <Name>Erin</Name>
        <Sales>10440</Sales>
    </Employee>
    <Employee>
        <Name>Kim</Name>
        <Sales>17772</Sales>
    </Employee>
    <Employee>
        <Name>Dean</Name>
        <Sales>23880</Sales>
    </Employee>
    <Employee>
        <Name>David</Name>
        <Sales>7663</Sales>
    </Employee>
    <Employee>
        <Name>Sanjay</Name>
        <Sales>21773</Sales>
    </Employee>
    <Employee>
        <Name>Michelle</Name>
        <Sales>32294</Sales>
    </Employee>
</NewDataSet>
```

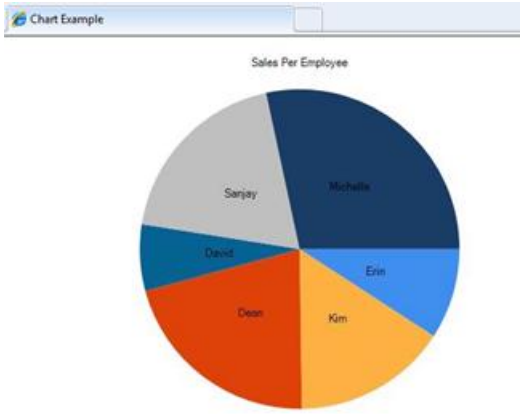
3. In the *App_Data* folder, create a new XML file named *data.xsd*. (Note that the extension this time is *.xsd*.)

4. Replace the existing XML with the following:

```
<?xml version="1.0" ?>
```

This example also shows you how to specify a particular chart type. When the data is added in the `AddSeries` method, the `chartType` parameter is also set to display a pie chart.

7. Run the page in a browser.



TIP

"Using" Statements and Fully Qualified Names

The .NET Framework that ASP.NET Web Pages with Razor syntax is based on consists of many thousands of components (classes). To make it manageable to work with all these classes, they're organized into *namespaces*, which are somewhat like libraries. For example, the `System.Web` namespace contains classes that support browser/server communication, the `System.Xml` namespace contains classes that are used to create and read XML files, and the `System.Data` namespace contains classes that let you work with data.

In order to access any given class in the .NET Framework, code needs to know not just the class name, but also the namespace that the class is in. For example, in order to use the `Chart` helper, code needs to find the `System.Web.Helpers.Chart` class, which combines the namespace (`System.Web.Helpers`) with the class name (`Chart`). This is known as the class's *fully-qualified* name — its complete, unambiguous location within the vastness of the .NET Framework. In code, this would look like the following:

```
var myChart = new System.Web.Helpers.Chart(width: 600, height: 400) // etc.
```

However, it's cumbersome (and error prone) to have to use these long, fully-qualified names every time you want to refer to a class or helper. Therefore, to make it easier to use class names, you can *import* the namespaces you're interested in, which is usually is just a handful from among the many namespaces in the .NET Framework. If you've imported a namespace, you can use just a class name (`Chart`) instead of the fully qualified name (`System.Web.Helpers.Chart`). When your code runs and encounters a class name, it can look in just the namespaces you've imported to find that class.

When you use ASP.NET Web Pages with Razor syntax to create web pages, you typically use the same set of classes each time, including the `WebPage` class, the various helpers, and so on. To save you the work of importing the relevant namespaces every time you create a website, ASP.NET is configured so it automatically imports a set of core namespaces for every website. That's why you haven't had to deal with namespaces or importing up to now; all the classes you've worked with are in namespaces that are already imported for you.

However, sometimes you need to work with a class that isn't in a namespace that's automatically imported for you. In that case, you can either use that class's fully-qualified name, or you can manually import the namespace that contains the class. To import a namespace, you use the `using` statement (`import` in Visual Basic), as you saw in an example earlier the article.

For example, the `DataSet` class is in the `System.Data` namespace. The `System.Data` namespace is not automatically available to ASP.NET Razor pages. Therefore, to work with the `DataSet` class using its fully qualified name, you can use code like this:

```
var dataSet = new System.Data.DataSet();
```

If you have to use the `DataSet` class repeatedly you can import a namespace like this and then use just the class name in code:

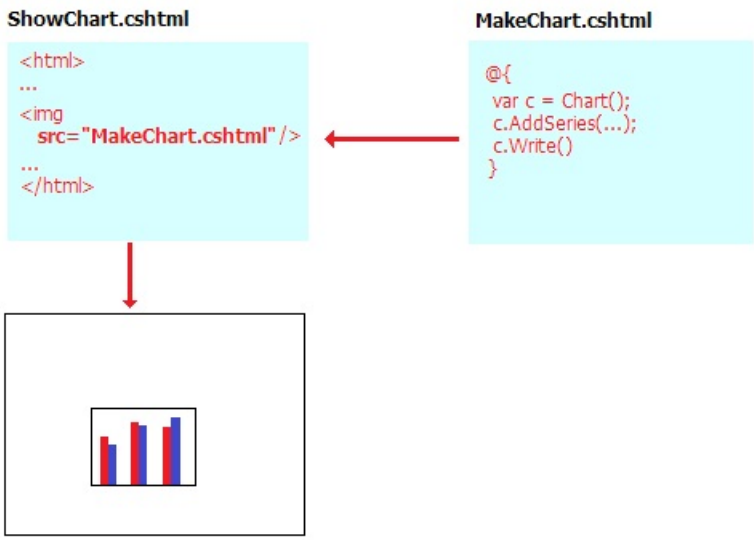
```
@using System.Data;
@{
    var dataSet = new DataSet();
    // etc.
}
```

You can add `using` statements for any other .NET Framework namespaces that you want to reference. However, as noted, you won't need to do this often, because most of the classes that you'll work with are in namespaces that are imported automatically by ASP.NET for use in *.cshtml* and *.vbhtml* pages.

Displaying Charts Inside a Web Page

In the examples you've seen so far, you create a chart and then the chart is rendered directly to the browser as a graphic. In many cases, though, you want to display a chart as part of a page, not just by itself in the browser. To do that requires a two-step process. The first step is to create a page that generates the chart, as you've already seen.

The second step is to display the resulting image in another page. To display the image, you use an HTML `<img>` element, in the same way you would to display any image. However, instead of referencing a *.jpg* or *.png* file, the `<img>` element references the *.cshtml* file that contains the `Chart` helper that creates the chart. When the display page runs, the `<img>` element gets the output of the `Chart` helper and renders the chart.



1. Create a file named *ShowChart.cshtml*.

2. Replace the existing content with the following:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Chart Example</title>
  </head>
  <body>
    <h1>Chart Example</h1>
    <p>The following chart is generated by the <em>ChartArrayBasic.cshtml</em> file, but is shown
       in this page.</p>
    <p><img src="ChartArrayBasic.cshtml" /> </p>
  </body>
</html>
```

   The code uses the `<img>` element to display the chart that you created earlier in the *ChartArrayBasic.cshtml* file.

3. Run the web page in a browser. The *ShowChart.cshtml* file displays the chart image based on the code contained in the *ChartArrayBasic.cshtml* file.

Styling a Chart

The `Chart` helper supports a large number of options that let you customize the appearance of the chart. You can set colors, fonts, borders, and so on. An easy way to customize the appearance of a chart is to use a *theme*. Themes are collections of information that specify how to render a chart using fonts, colors, labels, palettes, borders, and effects. (Note that the style of a chart does not indicate the type of chart.)

The following table lists built-in themes.

THEME	DESCRIPTION
`Vanilla`	Displays red columns on a white background.

THEME	DESCRIPTION
`Blue`	Displays blue columns on a blue gradient background.
`Green`	Displays blue columns on a green gradient background.
`Yellow`	Displays orange columns on a yellow gradient background.
`Vanilla3D`	Displays 3-D red columns on a white background.

You can specify the theme to use when you create a new chart.

1. Create a new file named *ChartStyleGreen.cshtml*.

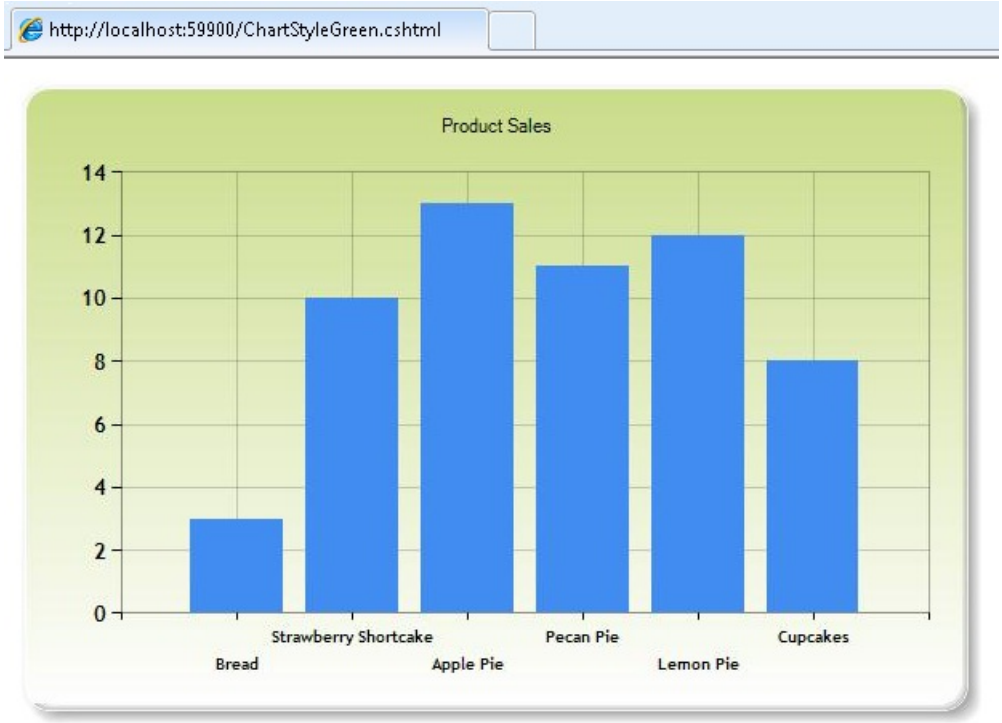
2. Replace the existing content in the page with the following:

```
@{
    var db = Database.Open("SmallBakery");
    var data = db.Query("SELECT Name, Price FROM Product");
    var myChart = new Chart(width: 600,
                        height: 400,
                        theme: ChartTheme.Green)
        .AddTitle("Product Sales")
        .DataBindTable(data, "Name")
        .Write();
}
```

   This code is the same as the earlier example that uses the database for data, but adds the `theme` parameter when it creates the `Chart` object. The following shows the changed code:

```
var myChart = new Chart(width: 600,
                    height: 400,
                    theme: ChartTheme.Green)
```

3. Run the page in a browser. You see the same data as before, but the chart looks more polished:



Saving a Chart

When you use the `Chart` helper as you've seen so far in this article, the helper re-creates the chart from scratch each time it's invoked. If necessary, the code for the chart also re-queries the database or re-reads the XML file to get the data. In some cases, doing this can be a complex operation, such as if the database that you're querying is large, or if the XML file contains a lot of data. Even if the chart doesn't involve a lot of data, the process of dynamically creating an image takes up server resources, and if many people request the page or pages that display the chart, there can be an impact on the performance of your website.

To help you reduce the potential performance impact of creating a chart, you can create a chart the first time you need it and then save it. When the chart is needed again, rather than regenerating it, you can just fetch the saved version and render that.

You can save a chart in these ways:

- Cache the chart in computer memory (on the server).
- Save the chart as an image file.
- Save the chart as an XML file. This option lets you modify the chart before you save it.

Caching a Chart

After you've created a chart, you can cache it. Caching a chart means that it doesn't have to be re-created if it needs to be displayed again. When you save a chart in the cache, you give it a key that must be unique to that chart.

Charts saved to the cache might be removed if the server runs low on memory. In addition, the cache is cleared if your application restarts for any reason. Therefore, the standard way to work with a cached chart is to always check first whether it's available in the cache, and if not, then to create or re-create it.

1. At the root of your website, create a file named *ShowCachedChart.cshtml*.

2. Replace the existing content with the following:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Chart Example</title>
    </head>
<body>
    <h1>Chart Example</h1>
    <img src="ChartSaveToCache.cshtml?key=myChartKey" />
    <p><a href="ClearCache.cshtml">Clear cache</a></p>
</body>
</html>
```

   The `<img>` tag includes a `src` attribute that points to the *ChartSaveToCache.cshtml* file and passes a key to the page as a query string. The key contains the value "myChartKey". The *ChartSaveToCache.cshtml* file contains the `Chart` helper that creates the chart. You'll create this page in a moment.

   At the end of the page there's a link to a page named *ClearCache.cshtml*. That's a page that you'll also create shortly. You need the *ClearCache.cshtml* only to test caching for this example — it's not a link or page that you'd normally include when working with cached charts.

3. At the root of your website, create a new file named *ChartSaveToCache.cshtml*.

4. Replace the existing content with the following:

```
@{
    var chartKey = Request["key"];
    if (chartKey != null) {
        var cachedChart = Chart.GetFromCache(key: chartKey);
        if (cachedChart == null) {
            cachedChart = new Chart(600, 400);
            cachedChart.AddTitle("Cached Chart -- Cached at " + DateTime.Now);
            cachedChart.AddSeries(
               name: "Employee",
               axisLabel: "Name",
               xValue: new[] { "Peter", "Andrew", "Julie", "Mary", "Dave" },
               yValues: new[] { "2", "6", "4", "5", "3" });
            cachedChart.SaveToCache(key: chartKey,
               minutesToCache: 2,
               slidingExpiration: false);
        }
        Chart.WriteFromCache(chartKey);
    }
}
```

The code first checks whether anything was passed as the key value in the query string. If so, the code tries to read a chart out of the cache by calling the `GetFromCache` method and passing it the key. If it turns out that there's nothing in the cache under that key (which would happen the first time that the chart is requested), the code creates the chart as usual. When the chart is finished, the code saves it to the cache by calling `SaveToCache`. That method requires a key (so the chart can be requested later), and the amount of time that the chart should be saved in the cache. (The exact time you'd cache a chart would depend on how often you thought the data it represents might change.) The `SaveToCache` method also requires a `slidingExpiration` parameter — if this is set to true, the timeout counter is reset each time the chart is accessed. In this case, it in effect means that the chart's cache entry expires 2 minutes after the last time someone accessed the chart. (The alternative to sliding expiration is absolute expiration, meaning that the cache entry would expire exactly 2 minutes after it was put into the cache, no matter how often it had been accessed.)

Finally, the code uses the `WriteFromCache` method to fetch and render the chart from the cache. Note that this method is outside the `if` block that checks the cache, because it will get the chart from the cache whether the chart was there to begin with or had to be generated and saved in the cache.

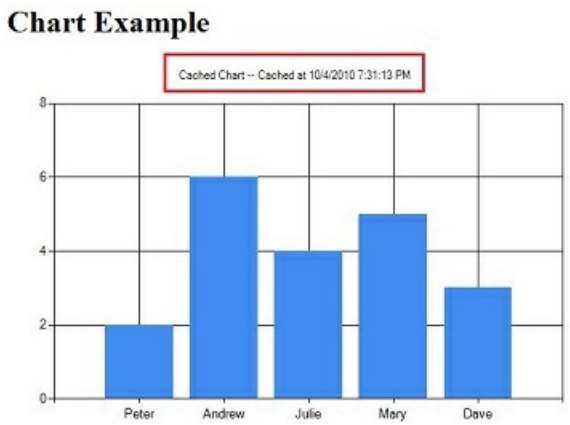
Notice that in the example, the `AddTitle` method includes a timestamp. (It adds the current date and time — `DateTime.Now` — to the title.)

5. Create a new page named *ClearCache.cshtml* and replace its content with the following:

```
@{
    WebCache.Remove("myChartKey");
}
<!DOCTYPE html>
<html lang="en">
  <body>
    <p>Cache has been cleared.</p>
    <p>Return to <a href="ShowCachedChart.cshtml">ShowCachedChart.cshtml</a></p>
  </body>
</html>
```

This page uses the `WebCache` helper to remove the chart that's cached in *ChartSaveToCache.cshtml*. As noted earlier, you don't normally have to have a page like this. You're creating it here only to make it easier to test caching.

6. Run the *ShowCachedChart.cshtml* web page in a browser. The page displays the chart image based on the code contained in the *ChartSaveToCache.cshtml* file. Take note of what the timestamp says in the chart title.



7. Close the browser.

8. Run the *ShowCachedChart.cshtml* again. Notice that the timestamp is the same as before, which indicates that the chart was not regenerated, but was instead read from the cache.

9. In *ShowCachedChart.cshtml*, click the **Clear cache** link. This takes you to *ClearCache.cshtml*, which reports that the cache has been cleared.

10. Click the **Return to ShowCachedChart.cshtml** link, or re-run *ShowCachedChart.cshtml* from WebMatrix. Notice that this time the timestamp has changed, because the cache has been cleared. Therefore, the code had to regenerate the chart and put it back into the cache.

Saving a Chart as an Image File

You can also save a chart as an image file (for example, as a *.jpg* file) on the server. You can then use the image file the way you would any image. The advantage is the file is stored rather than saved to a temporary cache. You can save a new chart image at different times (for example, every hour) and then keep a permanent record of the changes that occur over time. Note that you must make sure that your web application has permission to save a file to the folder on the server where you want to put the image file.

1. At the root of your website, create a folder named *_ChartFiles* if it does not already exist.

2. At the root of your website, create a new file named *ChartSave.cshtml*.

3. Replace the existing content with the following:

```
@{
    var filePathName = "_ChartFiles/chart01.jpg";
    if (!File.Exists(Server.MapPath(filePathName))) {
        var chartImage = new Chart(600, 400);
        chartImage.AddTitle("Chart Title");
        chartImage.AddSeries(
                name: "Employee",
                axisLabel: "Name",
                xValue: new[] {  "Peter", "Andrew", "Julie", "Mary", "Dave" },
                yValues: new[] { "2", "6", "4", "5", "3" });
        chartImage.Save(path: filePathName);
    }
}
<!DOCTYPE html>
<html>
    <head>
        <title>Chart Example</title>
    </head>
    <body>
        <img src="@filePathName" />
    </body>
</html>
```

The code first checks to see whether the *.jpg* file exists by calling the `File.Exists` method. If the file does not exist, the code creates a new `Chart` from an array. This time, the code calls the `Save` method and passes the `path` parameter to specify the file path and file name of where to save the chart. In the body of the page, an `<img>` element uses the path to point to the *.jpg* file to display.

4. Run the *ChartSave.cshtml* file.

5. Return to WebMatrix. Notice that an image file named *chart01.jpg* has been saved in the *_ChartFiles* folder.

Saving a Chart as an XML File

Finally, you can save a chart as an XML file on the server. An advantage of using this method over caching the chart or saving the chart to a file is that you could modify the XML before displaying the chart if you wanted to. Your application has to have read/write permissions for the folder on the server where you want to put the image file.

1. At the root of your website, create a new file named *ChartSaveXml.cshtml*.

2. Replace the existing content with the following:

```
@{
    Chart chartXml;
    var filePathName = "_ChartFiles/XmlChart.xml";
    if (File.Exists(Server.MapPath(filePathName))) {
        chartXml = new Chart(width: 600,
                             height: 400,
                             themePath: filePathName);
    }
    else {
        chartXml = new Chart(width: 600,
                             height: 400);
        chartXml.AddTitle("Chart Title -- Saved at " + DateTime.Now);
        chartXml.AddSeries(
            name: "Employee",
            axisLabel: "Name",
            xValue: new[] { "Peter", "Andrew", "Julie", "Mary", "Dave" },
            yValues: new[] { "2", "6", "4", "5", "3" });
        chartXml.SaveXml(path: filePathName);
    }
    chartXml.Write();
}
```

This code is similar to the code that you saw earlier for storing a chart in the cache, except that it uses an XML file. The code first checks to see whether the XML file exists by calling the `File.Exists` method. If the file does exist, the code creates a new `Chart` object and passes the file name as the `themePath` parameter. This creates the chart based on whatever's in the XML file. If the XML file doesn't already exist, the code creates a chart like normal and then calls `SaveXml` to save it. The chart is rendered using the `Write` method, as you've seen before.

As with the page that showed caching, this code includes a timestamp in the chart title.

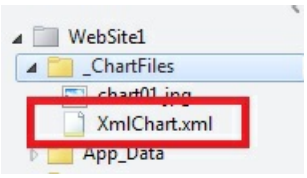
3. Create a new page named *ChartDisplayXMLChart.cshtml* and add the following markup to it:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Display chart from XML</title>
  </head>
  <body>
    <img src="ChartSaveXML.cshtml" />
  </body>
</html>
```

4. Run the *ChartDisplayXMLChart.cshtml* page. The chart is displayed. Take note of the timestamp in the chart's title.

5. Close the browser.

6. In WebMatrix, right-click the *_ChartFiles* folder, click **Refresh**, and then open the folder. The *XMLChart.xml* file in this folder was created by the `Chart` helper.



7. Run the *ChartDisplayXMLChart.cshtml* page again. The chart shows the same timestamp as the first time you ran the page. That's because the chart is being generated from the XML you saved earlier.

8. In WebMatrix, open the *_ChartFiles* folder and delete the *XMLChart.xml* file.

9. Run the *ChartDisplayXMLChart.cshtml* page once more. This time, the timestamp is updated, because the `Chart` helper had to recreate the XML file. If you want, check the *_ChartFiles* folder and notice that the XML file is back.

Additional Resources

- Introduction to Working with a Database in ASP.NET Web Pages Sites
- Using Caching in ASP.NET Web Pages Sites to Improve Performance
- Chart Class (ASP.NET Web Pages API reference on MSDN)

Working with Files in an ASP.NET Web Pages (Razor) Site

2/20/2020 • 16 minutes to read • Edit Online

by Tom FitzMacken

This article explains how to read, write, append, delete, and upload files in an ASP.NET Web Pages (Razor) site.

> **NOTE**
> If you want to upload images and manipulate them (for example, flip or resize them), see Working with Images in an ASP.NET Web Pages Site.

What you'll learn:

- How to create a text file and write data to it.
- How to append data to an existing file.
- How to read a file and display from it.
- How to delete files from a website.
- How to let users upload one file or multiple files.

These are the ASP.NET programming features introduced in the article:

- The `File` object, which provides a way to manage files.
- The `FileUpload` helper.
- The `Path` object, which provides methods that let you manipulate path and file names.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 2
- WebMatrix 2

This tutorial also works with WebMatrix 3.

Creating a Text File and Writing Data to It

In addition to using a database in your website, you might work with files. For example, you might use text files as a simple way to store data for the site. (A text file that's used to store data is sometimes called a *flat file*.) Text files can be in different formats, like *.txt*, *.xml*, or *.csv* (comma-delimited values).

If you want to store data in a text file, you can use the `File.WriteAllText` method to specify the file to create and the data to write to it. In this procedure, you'll create a page that contains a simple form with three `input` elements (first name, last name, and email address) and a **Submit** button. When the user submits the form, you'll store the user's input in a text file.

1. Create a new folder named *App_Data*, if it doesn't exist already.

2. At the root of your website, create a new file named *UserData.cshtml*.

3. Replace the existing content with the following:

```
@{
    var result = "";
    if (IsPost)
    {
        var firstName = Request["FirstName"];
        var lastName = Request["LastName"];
        var email = Request["Email"];

        var userData = firstName + "," + lastName +
            "," + email + Environment.NewLine;

        var dataFile = Server.MapPath("~/App_Data/data.txt");
        File.WriteAllText(@dataFile, userData);
        result = "Information saved.";
    }
}
<!DOCTYPE html>
<html>
<head>
    <title>Write Data to a File</title>
</head>
<body>
    <form id="form1" method="post">
    <div>
        <table>
            <tr>
                <td>First Name:</td>
                <td><input id="FirstName" name="FirstName" type="text" /></td>

            </tr>
            <tr>
                <td>Last Name:</td>
                <td><input id="LastName" name="LastName" type="text" /></td>
            </tr>
            <tr>
                <td>Email:</td>
                <td><input id="Email" name="Email" type="text" /></td>
            </tr>
            <tr>
                <td></td>
                <td><input type="submit" value="Submit"/></td>
            </tr>
        </table>
    </div>
    <div>
    @if(result != ""){
        <p>Result: @result</p>
    }
    </div>
    </form>
</body>
</html>
```

The HTML markup creates the form with the three text boxes. In the code, you use the `IsPost` property to determine whether the page has been submitted before you start processing.

The first task is to get the user input and assign it to variables. The code then concatenates the values of the separate variables into one comma-delimited string, which is then stored in a different variable. Notice that the comma separator is a string contained in quotation marks (","), because you're literally embedding a comma into the big string that you're creating. At the end of the data that you concatenate together, you add `Environment.NewLine`. This adds a line break (a newline character). What you're creating with all this concatenation is a string that looks like this:

```
David,Jones,davidj@contoso.com
```

(With an invisible line break at the end.)

You then create a variable (`dataFile`) that contains the location and name of the file to store the data in. Setting the location requires some special handling. In websites, it's a bad practice to refer in code to absolute paths like *C:\Folder\File.txt* for files on the web server. If a website is moved, an absolute path will be wrong. Moreover, for a hosted site (as opposed to on your own computer) you typically don't even know what the correct path is when you're writing the code.

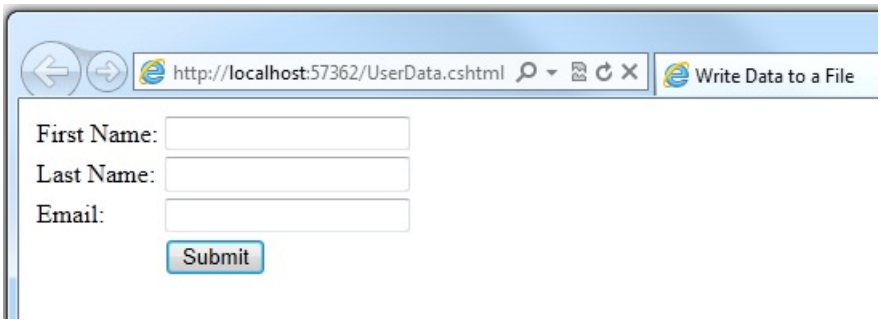
But sometimes (like now, for writing a file) you do need a complete path. The solution is to use the `MapPath` method of the `Server` object. This returns the complete path to your website. To get the path for the website root, you user the `~` operator (to represen the site's virtual root) to `MapPath`. (You can also pass a subfolder name to it, like *~/App_Data/*, to get the path for that subfolder.) You can then concatenate additional information onto whatever the method returns in order to create a complete path. In this example, you add a file name. (You can read more about how to work with file and folder paths in Introduction to ASP.NET Web Pages Programming Using the Razor Syntax.)

The file is saved in the *App_Data* folder. This folder is a special folder in ASP.NET that's used to store data files, as described in Introduction to Working with a Database in ASP.NET Web Pages Sites.

The `WriteAllText` method of the `File` object writes the data to the file. This method takes two parameters: the name (with path) of the file to write to, and the actual data to write. Notice that the name of the first parameter has an `@` character as a prefix. This tells ASP.NET that you're providing a verbatim string literal, and that characters like "/" should not be interpreted in special ways. (For more information, see Introduction to ASP.NET Web Programming Using the Razor Syntax.)

> **NOTE**
> In order for your code to save files in the *App_Data* folder, the application needs read-write permissions for that folder. On your development computer this is not typically an issue. However, when you publish your site to a hosting provider's web server, you might need to explicitly set those permissions. If you run this code on a hosting provider's server and get errors, check with the hosting provider to find out how to set those permissions.

- Run the page in a browser.



- Enter values into the fields and then click **Submit**.
- Close the browser.
- Return to the project and refresh the view.
- Open the *data.txt* file. The data you submitted in the form is in the file.

```
data.txt    UserData.cshtml
1 David,Jones,davidj@contoso.com
2
```

- Close the *data.txt* file.

Appending Data to an Existing File

In the previous example, you used `WriteAllText` to create a text file that's got just one piece of data in it. If you call the method again and pass it the same file name, the existing file is completely overwritten. However, after you've created a file you often want to add new data to the end of the file. You can do that using the `AppendAllText` method of the `File` object.

1. In the website, make a copy of the *UserData.cshtml* file and name the copy *UserDataMultiple.cshtml*.

2. Replace the code block before the opening `<!DOCTYPE html>` tag with the following code block:

```
@{
    var result = "";
    if (IsPost)
    {
        var firstName = Request["FirstName"];
        var lastName = Request["LastName"];
        var email = Request["Email"];

        var userData = firstName + "," + lastName +
            "," + email + Environment.NewLine;

        var dataFile = Server.MapPath("~/App_Data/data.txt");
        File.AppendAllText (@dataFile, userData);
        result = "Information saved.";
    }
}
```

   This code has one change in it from the previous example. Instead of using `WriteAllText`, it uses `the AppendAllText` method. The methods are similar, except that `AppendAllText` adds the data to the end of the file. As with `WriteAllText`, `AppendAllText` creates the file if it doesn't already exist.

3. Run the page in a browser.

4. Enter values for the fields and then click **Submit**.

5. Add more data and submit the form again.

6. Return to your project, right-click the project folder, and then click **Refresh**.

7. Open the *data.txt* file. It now contains the new data that you just entered.

```
data.txt    UserDataMultiple.cshtml
1 David,Jones,davidj@contoso.com
2 Jim,Hance,jimh@contoso.com
3
```

Reading and Displaying Data from a File

Even if you don't need to write data to a text file, you'll probably sometimes need to read data from one. To do this, you can again use the `File` object. You can use the `File` object to read each line individually (separated by line breaks) or to read individual item no matter how they're separated.

This procedure shows you how to read and display the data that you created in the previous example.

1. At the root of your website, create a new file named *DisplayData.cshtml*.
2. Replace the existing content with the following:

```
@{
    var result = "";
    Array userData = null;
    char[] delimiterChar = {','};

    var dataFile = Server.MapPath("~/App_Data/data.txt");

    if (File.Exists(dataFile)) {
        userData = File.ReadAllLines(dataFile);
        if (userData == null) {
            // Empty file.
            result = "The file is empty.";
        }
    }
    else {
        // File does not exist.
        result = "The file does not exist.";
    }
}
<!DOCTYPE html>

<html>
<head>
    <title>Reading Data from a File</title>
</head>
<body>
    <div>
        <h1>Reading Data from a File</h1>
        @result
        @if (result == "") {
            <ol>
            @foreach (string dataLine in userData) {
            <li>
                User
                <ul>
                @foreach (string dataItem in dataLine.Split(delimiterChar)) {
                    <li>@dataItem</li >
                }
                </ul>
            </li>
            }
            </ol>
        }
    </div>
</body>
</html>
```

The code starts by reading the file that you created in the previous example into a variable named `userData`, using this method call:

```
File.ReadAllLines(dataFile)
```

The code to do this is inside an `if` statement. When you want to read a file, it's a good idea to use the `File.Exists` method to determine first whether the file is available. The code also checks whether the file is empty.

The body of the page contains two `foreach` loops, one nested inside the other. The outer `foreach` loop gets one line at a time from the data file. In this case, the lines are defined by line breaks in the file — that is, each data item is on its own line. The outer loop creates a new item (`<li>` element) inside an ordered list (

`<ol>` element).

The inner loop splits each data line into items (fields) using a comma as a delimiter. (Based on the previous example, this means that each line contains three fields — the first name, last name, and email address, each separated by a comma.) The inner loop also creates a `<ul>` list and displays one list item for each field in the data line.

The code illustrates how to use two data types, an array and the `char` data type. The array is required because the `File.ReadAllLines` method returns data as an array. The `char` data type is required because the `Split` method returns an `array` in which each element is of the type `char`. (For information about arrays, see Introduction to ASP.NET Web Programming Using the Razor Syntax.)

3. Run the page in a browser. The data you entered for the previous examples is displayed.

TIP

Displaying Data from a Microsoft Excel Comma-Delimited File

You can use Microsoft Excel to save the data contained in a spreadsheet as a comma-delimited file (*.csv* file). When you do, the file is saved in plain text, not in Excel format. Each row in the spreadsheet is separated by a line break in the text file, and each data item is separated by a comma. You can use the code shown in the previous example to read an Excel comma-delimited file just by changing the name of the data file in your code.

Deleting Files

To delete files from your website, you can use the `File.Delete` method. This procedure shows how to let users delete an image (*.jpg* file) from an *images* folder if they know the name of the file.

NOTE

Important In a production website, you typically restrict who's allowed to make changes to the data. For information about how to set up membership and about ways to authorize users to perform tasks on the site, see Adding Security and Membership to an ASP.NET Web Pages Site.

1. In the website, create a subfolder named *images*.
2. Copy one or more *.jpg* files into the *images* folder.
3. In the root of the website, create a new file named *FileDelete.cshtml*.
4. Replace the existing content with the following:

```
@{
    bool deleteSuccess = false;
    var photoName = "";
    if (IsPost) {
        photoName = Request["photoFileName"] + ".jpg";
        var fullPath = Server.MapPath("~/images/" + photoName);

        if (File.Exists(fullPath))
        {
                File.Delete(fullPath);
                deleteSuccess = true;
        }
    }
}
<!DOCTYPE html>
<html>
  <head>
    <title>Delete a Photo</title>
  </head>
  <body>
    <h1>Delete a Photo from the Site</h1>
    <form name="deletePhoto" action="" method="post">
      <p>File name of image to delete (without .jpg extension):
      <input name="photoFileName" type="text" value="" />
      </p>
      <p><input type="submit" value="Submit" /></p>
    </form>

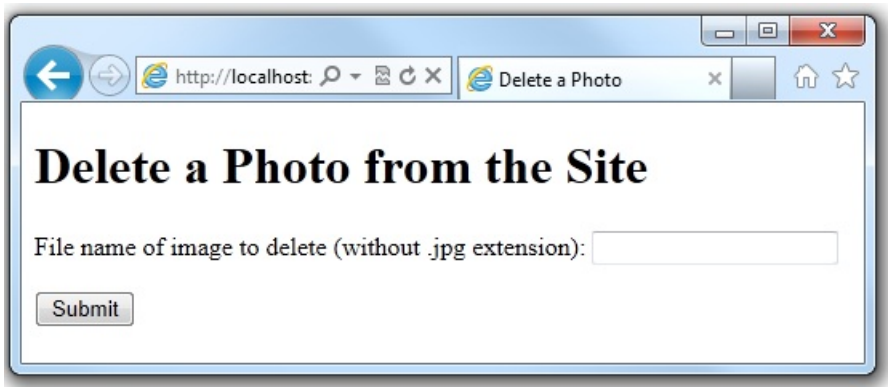
    @if(deleteSuccess) {
        <p>
        @photoName deleted!
        </p>
        }
  </body>
</html>
```

This page contains a form where users can enter the name of an image file. They don't enter the *.jpg* file-name extension; by restricting the file name like this, you help prevents users from deleting arbitrary files on your site.

The code reads the file name that the user has entered and then constructs a complete path. To create the path, the code uses the current website path (as returned by the `Server.MapPath` method), the *images* folder name, the name that the user has provided, and ".jpg" as a literal string.

To delete the file, the code calls the `File.Delete` method, passing it the full path that you just constructed. At the end of the markup, code displays a confirmation message that the file was deleted.

5. Run the page in a browser.



6. Enter the name of the file to delete and then click **Submit**. If the file was deleted, the name of the file is displayed at the bottom of the page.

Letting Users Upload a File

The `FileUpload` helper lets users upload files to your website. The procedure below shows you how to let users upload a single file.

1. Add the ASP.NET Web Helpers Library to your website as described in Installing Helpers in an ASP.NET Web Pages Site, if you didn't add it previously.

2. In the *App_Data* folder, create a new a folder and name it *UploadedFiles*.

3. In the root, create a new file named *FileUpload.cshtml*.

4. Replace the existing content in the page with the following:

```
@using Microsoft.Web.Helpers;
@{
    var fileName = "";
    if (IsPost) {
        var fileSavePath = "";
        var uploadedFile = Request.Files[0];
        fileName = Path.GetFileName(uploadedFile.FileName);
        fileSavePath = Server.MapPath("~/App_Data/UploadedFiles/" +
          fileName);
        uploadedFile.SaveAs(fileSavePath);
    }
}
<!DOCTYPE html>
<html>
    <head>
    <title>FileUpload - Single-File Example</title>
    </head>
    <body>
    <h1>FileUpload - Single-File Example</h1>
    @FileUpload.GetHtml(
        initialNumberOfFiles:1,
        allowMoreFilesToBeAdded:false,
        includeFormTag:true,
        uploadText:"Upload")
    @if (IsPost) {
        <span>File uploaded!</span><br/>
    }
    </body>
</html>
```

The body portion of the page uses the `FileUpload` helper to create the upload box and buttons that you're probably familiar with:

The properties that you set for the `FileUpload` helper specify that you want a single box for the file to upload and that you want the submit button to read **Upload**. (You'll add more boxes later in the article.)

When the user clicks **Upload**, the code at the top of the page gets the file and saves it. The `Request` object that you normally use to get values from form fields also has a `Files` array that contains the file (or files) that have been uploaded. You can get individual files out of specific positions in the array — for example, to get the first uploaded file, you get `Request.Files[0]`, to get the second file, you get `Request.Files[1]`, and so on. (Remember that in programming, counting usually starts at zero.)

When you fetch an uploaded file, you put it in a variable (here, `uploadedFile`) so that you can manipulate it. To determine the name of the uploaded file, you just get its `FileName` property. However, when the user uploads a file, `FileName` contains the user's original name, which includes the entire path. It might look like this:

C:\Users\Public\Sample.txt

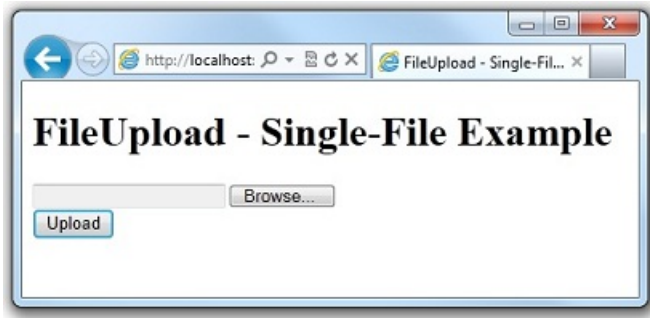
You don't want all that path information, though, because that's the path on the user's computer, not for your server. You just want the actual file name (*Sample.txt*). You can strip out just the file from a path by using the `Path.GetFileName` method, like this:

```
Path.GetFileName(uploadedFile.FileName)
```

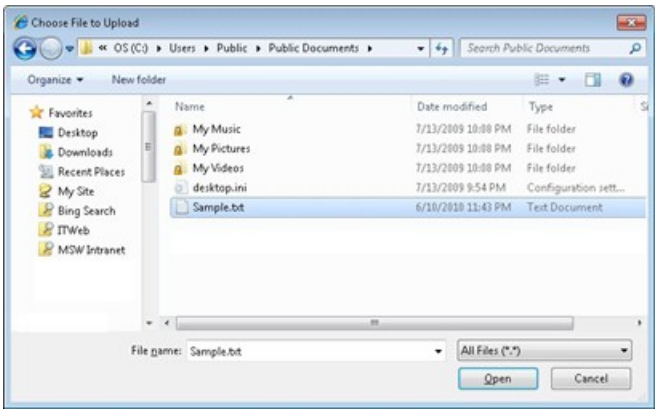
The `Path` object is a utility that has a number of methods like this that you can use to strip paths, combine paths, and so on.

Once you've gotten the name of the uploaded file, you can build a new path for where you want to store the uploaded file in your website. In this case, you combine `Server.MapPath`, the folder names (*App_Data/UploadedFiles*), and the newly stripped file name to create a new path. You can then call the uploaded file's `SaveAs` method to actually save the file.

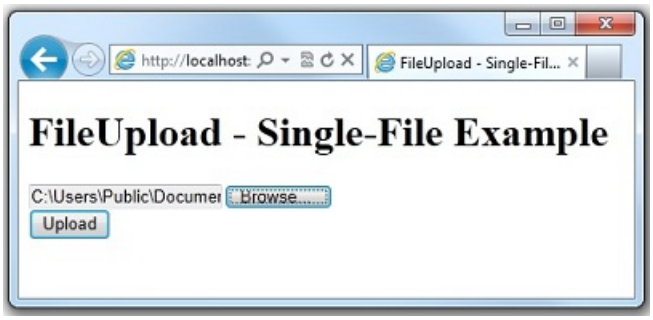
5. Run the page in a browser.



6. Click **Browse** and then select a file to upload.



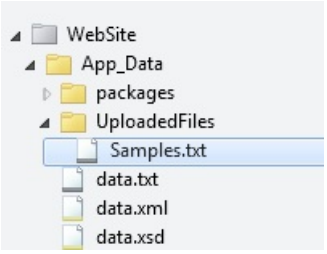
The text box next to the **Browse** button will contain the path and file location.



7. Click **Upload**.

8. In the website, right-click the project folder and then click **Refresh**.

9. Open the *UploadedFiles* folder. The file that you uploaded is in the folder.



Letting Users Upload Multiple Files

In the previous example, you let users upload one file. But you can use the `FileUpload` helper to upload more than one file at a time. This is handy for scenarios like uploading photos, where uploading one file at a time is tedious. (You can read about uploading photos in Working with Images in an ASP.NET Web Pages Site.) This example shows how to let users upload two at a time, although you can use the same technique to upload more than that.

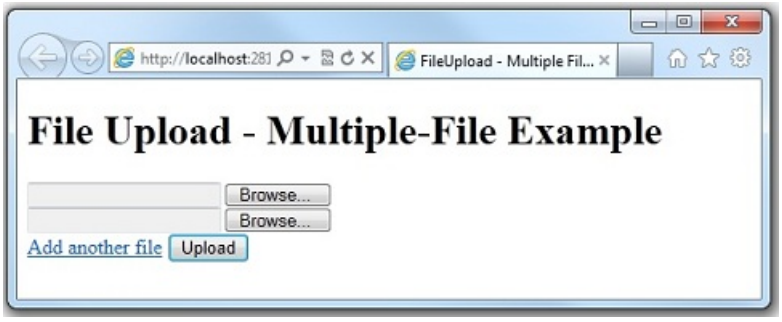
1. Add the ASP.NET Web Helpers Library to your website as described in Installing Helpers in an ASP.NET Web Pages Site, if you haven't already.

2. Create a new page named *FileUploadMultiple.cshtml*.

3. Replace the existing content in the page with the following:

```
@using Microsoft.Web.Helpers;
@{
  var message = "";
  if (IsPost) {
      var fileName = "";
      var fileSavePath = "";
      int numFiles = Request.Files.Count;
      int uploadedCount = 0;
      for(int i =0; i < numFiles; i++) {
          var uploadedFile = Request.Files[i];
          if (uploadedFile.ContentLength > 0) {
              fileName = Path.GetFileName(uploadedFile.FileName);
              fileSavePath = Server.MapPath("~/App_Data/UploadedFiles/" +
                fileName);
              uploadedFile.SaveAs(fileSavePath);
              uploadedCount++;
          }
       }
       message = "File upload complete. Total files uploaded: " +
         uploadedCount.ToString();
   }
}
<!DOCTYPE html>
<html>
    <head><title>FileUpload - Multiple File Example</title></head>
<body>
    <form id="myForm" method="post"
       enctype="multipart/form-data"
       action="">
    <div>
    <h1>File Upload - Multiple-File Example</h1>
    @if (!IsPost) {
        @FileUpload.GetHtml(
            initialNumberOfFiles:2,
            allowMoreFilesToBeAdded:true,
            includeFormTag:true,
            addText:"Add another file",
            uploadText:"Upload")
        }
    <span>@message</span>
    </div>
    </form>
</body>
</html>
```

In this example, the `FileUpload` helper in the body of the page is configured to let users upload two files by default. Because `allowMoreFilesToBeAdded` is set to `true`, the helper renders a link that lets user add more upload boxes:



To process the files that the user uploads, the code uses the same basic technique that you used in the previous example — get a file from `Request.Files` and then save it. (Including the various things you need to do to get the right file name and path.) The innovation this time is that the user might be uploading multiple files and you don't know many. To find out, you can get `Request.Files.Count`.

With this number in hand, you can loop through `Request.Files`, fetch each file in turn, and save it. When you want to loop a known number of times through a collection, you can use a `for` loop, like this:

```
for(int i =0; i < numFiles; i++) {
    var uploadedFile = Request.Files[i];
    if (uploadedFile.ContentLength > 0) {
        fileName = Path.GetFileName(uploadedFile.FileName);

    // etc.
}
```

The variable `i` is just a temporary counter that will go from zero to whatever upper limit you set. In this case, the upper limit is the number of files. But because the counter starts at zero, as is typical for counting scenarios in ASP.NET, the upper limit is actually one less than the file count. (If three files are uploaded, the count is zero to 2.)

The `uploadedCount` variable totals all the files that are successfully uploaded and saved. This code accounts for the possibility that an expected file may not be able to be uploaded.

4. Run the page in a browser. The browser displays the page and its two upload boxes.
5. Select two files to upload.
6. Click **Add another file**. The page displays a new upload box.

7. Click **Upload**.
8. In the website, right-click the project folder and then click **Refresh**.
9. Open the *UploadedFiles* folder to see the successfully uploaded files.

Additional Resources

Working with Images in an ASP.NET Web Pages Site

Exporting to a CSV File

Testing and Debugging

2/20/2020 • 2 minutes to read • Edit Online

How to test and debug ASP.NET Web Pages applications.

- Intro to Debugging
- ASP.NET Web Pages (Razor) Troubleshooting Guide

Introduction to Debugging ASP.NET Web Pages (Razor) Sites

2/20/2020 • 9 minutes to read • Edit Online

by Tom FitzMacken

This article explains various ways to debug pages in an ASP.NET Web Pages (Razor) website. Debugging is the process of finding and fixing errors in your code pages.

What you'll learn:

- How to display information that helps analyze and debug pages.
- How to use debugging tools in Visual Studio.

These are the ASP.NET features introduced in the article:

- The `ServerInfo` helper.
- `ObjectInfo` helper.

Software versions

- ASP.NET Web Pages (Razor) 3
- Visual Studio 2013

This tutorial also works with ASP.NET Web Pages 2. You can use WebMatrix 3 but the integrated debugger is not supported.

An important aspect of troubleshooting errors and problems in your code is to avoid them in the first place. You can do that by putting sections of your code that are likely to cause errors into `try/catch` blocks. For more information, see the section on handling errors in Introduction to ASP.NET Web Programming Using the Razor Syntax.

The `ServerInfo` helper is a diagnostic tool that gives you an overview of information about the web server environment that hosts your page. It also shows you HTTP request information that's sent when a browser requests the page. The `ServerInfo` helper displays the current user identity, the type of browser that made the request, and so on. This kind of information can help you troubleshoot common issues.

1. Create a new web page named *ServerInfo.cshtml*.
2. At the end of the page, just before the closing `</body>` tag, add `@ServerInfo.GetHtml()`:

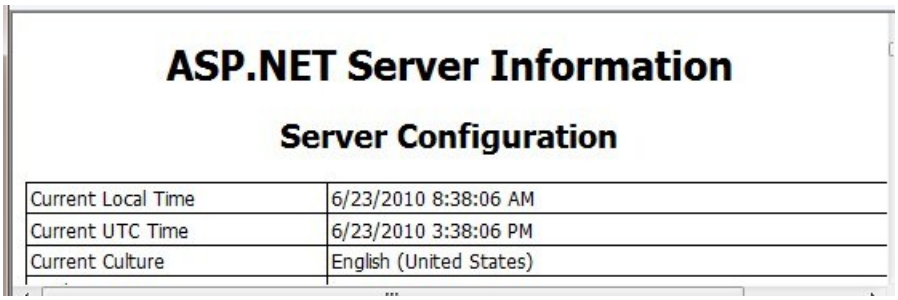
```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
   @ServerInfo.GetHtml()
    </body>
</html>
```

You can add the `ServerInfo` code anywhere in the page. But adding it at the end will keep its output

separate from your other page content, which makes it easier to read.

> **NOTE**
>
> **Important** You should remove any diagnostic code from your web pages before you move web pages to a production server. This applies to the `ServerInfo` helper as well as the other diagnostic techniques in this article that involve adding code to a page. You don't want your website visitors to see information about your server name, user names, paths on your server, and similar details, because this type of information might be useful to people with malicious intent.

3. Save the page and run it in a browser.



The `ServerInfo` helper displays four tables of information in the page:

- Server Configuration. This section provides information about the hosting web server, including computer name, the version of ASP.NET you're running, the domain name, and server time.
- ASP.NET Server Variables. This section provides details about the many HTTP protocol details (called HTTP variables) and values that are part of each web page request.
- HTTP Runtime Information. This section provides details about that the version of the Microsoft .NET Framework that your web page is running under, the path, details about the cache, and so on. (As you learned in Introduction to ASP.NET Web Programming Using the Razor Syntax, ASP.NET Web Pages using the Razor syntax are built on Microsoft's ASP.NET web server technology, which is itself built on an extensive software development library called the .NET Framework.)
- Environment Variables. This section provides a list of all the local environment variables and their values on the web server.

  A full description of all the server and request information is beyond the scope of this article, but you can see that the `ServerInfo` helper returns a lot of diagnostic information. For more information about the values that `ServerInfo` returns, see Recognized Environment Variables on the Microsoft TechNet website and IIS Server Variables on the MSDN website.

Embedding Output Expressions to Display Page Values

Another way to see what's happening in your code is to embed output expressions in the page. As you know, you can directly output the value of a variable by adding something like `@myVariable` or `@(subTotal * 12)` to the page. For debugging, you can place these output expressions at strategic points in your code. This enables you to see the value of key variables or the result of calculations when your page runs. When you're done debugging, you can remove the expressions or comment them out. This procedure illustrates a typical way to use embedded expressions to help debug a page.

1. Create a new WebMatrix page that's named *OutputExpression.cshtml*.
2. Replace the page content with the following:

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>

    @{
        var weekday = DateTime.Now.DayOfWeek;
        // As a test, add 1 day to the current weekday.
        if(weekday.ToString() != "Saturday") {
            // If weekday is not Saturday, simply add one day.
            weekday = weekday + 1;
        }
        else {
            // If weekday is Saturday, reset the day to 0, or Sunday.
            weekday = 0;
        }
        // Convert weekday to a string value for the switch statement.
        var weekdayText = weekday.ToString();

        var greeting = "";

        switch(weekdayText)
        {
            case "Monday":
                greeting = "Ok, it's a marvelous Monday.";
                break;
            case "Tuesday":
                greeting = "It's a tremendous Tuesday.";
                break;
            case "Wednesday":
                greeting = "Wild Wednesday is here!";
                break;
            case "Thursday":
                greeting = "All right, it's thrifty Thursday.";
                break;
            case "Friday":
                greeting = "It's finally Friday!";
                break;
            case "Saturday":
                greeting = "Another slow Saturday is here.";
                break;
            case "Sunday":
                greeting = "The best day of all: serene Sunday.";
                break;
            default:
                break;
        }
    }

    <h2>@greeting</h2>

    </body>
</html>
```

The example uses a `switch` statement to check the value of the `weekday` variable and then display a different output message depending on which day of the week it is. In the example, the `if` block within the first code block arbitrarily changes the day of the week by adding one day to the current weekday value. This is an error introduced for illustration purposes.

3. Save the page and run it in a browser.

   The page displays the message for the wrong day of the week. Whatever day of the week it actually is, you'll see the message for one day later. Although in this case you know why the message is off (because the code

deliberately sets the incorrect day value), in reality it's often hard to know where things are going wrong in the code. To debug, you need to find out what's happening to the value of key objects and variables such as `weekday`.

4. Add output expressions by inserting `@weekday` as shown in the two places indicated by comments in the code. These output expressions will display the values of the variable at that point in the code execution.

```
var weekday = DateTime.Now.DayOfWeek;
// DEBUG: Display the initial value of weekday.
@weekday

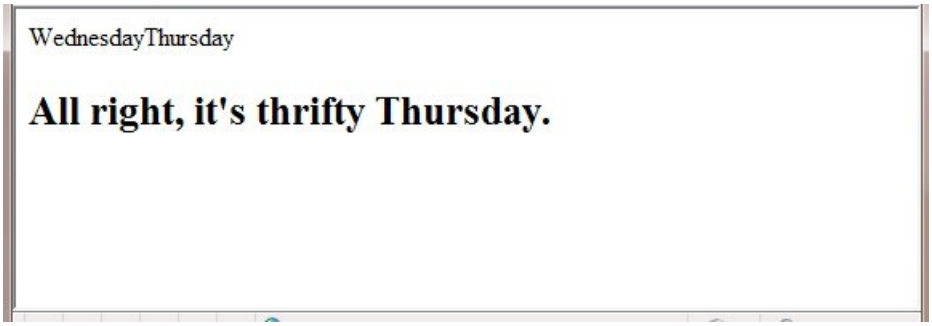
// As a test, add 1 day to the current weekday.
if(weekday.ToString() != "Saturday") {
    // If weekday is not Saturday, simply add one day.
    weekday = weekday + 1;
}
else {
    // If weekday is Saturday, reset the day to 0, or Sunday.
    weekday = 0;
}

// DEBUG: Display the updated test value of weekday.
@weekday

// Convert weekday to a string value for the switch statement.
var weekdayText = weekday.ToString();
```

5. Save and run the page in a browser.

   The page displays the real day of the week first, then the updated day of the week that results from adding one day, and then the resulting message from the `switch` statement. The output from the two variable expressions (`@weekday`) has no spaces between the days because you didn't add any HTML `<p>` tags to the output; the expressions are just for testing.



   Now you can see where the error is. When you first display the `weekday` variable in the code, it shows the correct day. When you display it the second time, after the `if` block in the code, the day is off by one. So you know that something has happened between the first and second appearance of the weekday variable. If this were a real bug, this kind of approach would help you narrow down the location of the code that's causing the problem.

6. Fix the code in the page by removing the two output expressions you added, and removing the code that changes the day of the week. The remaining, complete block of code looks like the following example:

```
@{
    var weekday = DateTime.Now.DayOfWeek;
    var weekdayText = weekday.ToString();

    var greeting = "";

    switch(weekdayText)
    {
        case "Monday":
            greeting = "Ok, it's a marvelous Monday.";
            break;
        case "Tuesday":
            greeting = "It's a tremendous Tuesday.";
            break;
        case "Wednesday":
            greeting = "Wild Wednesday is here!";
            break;
        case "Thursday":
            greeting = "All right, it's thrifty Thursday.";
            break;
        case "Friday":
            greeting = "It's finally Friday!";
            break;
        case "Saturday":
            greeting = "Another slow Saturday is here.";
            break;
        case "Sunday":
            greeting = "The best day of all: serene Sunday.";
            break;
        default:
            break;
    }
}
```

7. Run the page in a browser. This time you see the correct message displayed for the actual day of the week.

Using the ObjectInfo Helper to Display Object Values

The `ObjectInfo` helper displays the type and the value of each object you pass to it. You can use it to view the value of variables and objects in your code (like you did with output expressions in the previous example), plus you can see data type information about the object.

1. Open the file named *OutputExpression.cshtml* that you created earlier.

2. Replace all code in the page with the following block of code:

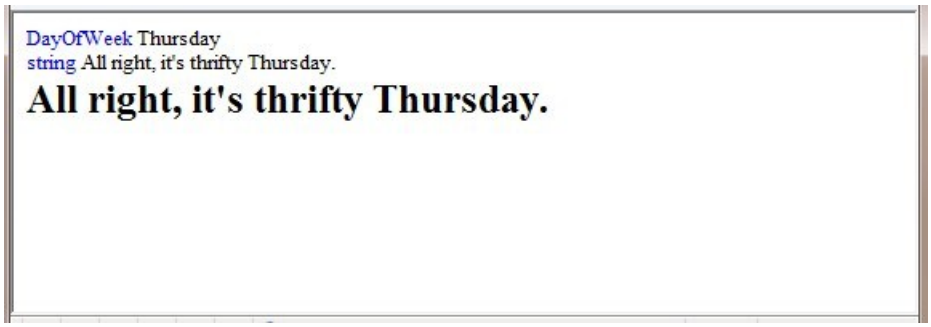
```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
    @{
      var weekday = DateTime.Now.DayOfWeek;
      @ObjectInfo.Print(weekday)
      var weekdayText = weekday.ToString();

      var greeting = "";

      switch(weekdayText)
      {
          case "Monday":
              greeting = "Ok, it's a marvelous Monday.";
              break;
          case "Tuesday":
              greeting = "It's a tremendous Tuesday.";
              break;
          case "Wednesday":
              greeting = "Wild Wednesday is here!";
              break;
          case "Thursday":
              greeting = "All right, it's thrifty Thursday.";
              break;
          case "Friday":
              greeting = "It's finally Friday!";
              break;
           case "Saturday":
              greeting = "Another slow Saturday is here.";
              break;
           case "Sunday":
              greeting = "The best day of all: serene Sunday.";
              break;
          default:
              break;
      }
    }
    @ObjectInfo.Print(greeting)
    <h2>@greeting</h2>

    </body>
</html>
```

3. Save and run the page in a browser.



In this example, the `ObjectInfo` helper displays two items:

- The type. For the first variable, the type is `DayOfWeek`. For the second variable, the type is `String`.
- The value. In this case, because you already display the value of the greeting variable in the page, the value is displayed again when you pass the variable to `ObjectInfo`.

For more complex objects, the `ObjectInfo` helper can display more information — basically, it can display the types and values of all of an object's properties.

Using Debugging Tools in Visual Studio

For a more comprehensive debugging experience, use Visual Studio. With Visual Studio, you can set a breakpoint in your code at the line that you want to inspect.

```
@{
    Layout = "~/_SiteLayout.cshtml";
    Page.Title = "Home Page";
    var myTime = DateTime.Now.TimeOfDay;
}
```

When you test the web site, the executing code halts at the breakpoint.

```
@{
    Layout = "~/_SiteLayout.cshtml";
    Page.Title = "Home Page";
    var myTime = DateTime.Now.TimeOfDay;
}
```

You can examine the current values of the variables, and step through the code line-by-line.

```
@{
    Layout = "~/_SiteLayout.cshtml";
    Page.Title = "Home Page";
    var myTime = DateTime.Now.TimeOfDay;
}
    myTime {14:54:10.4713083}
```

For information about using the integrated debugger in Visual Studio to debug ASP.NET Razor pages, see Programming ASP.NET Web Pages (Razor) Using Visual Studio.

Additional Resources

- Programming ASP.NET Web Pages (Razor) Using Visual Studio
- IIS Server Variables (MSDN)
- Recognized Environment Variables (TechNet)

ASP.NET Web Pages (Razor) Troubleshooting Guide

2/20/2020 • 7 minutes to read • Edit Online

by Tom FitzMacken

This article describes issues that you might have when working with ASP.NET Web Pages (Razor) and some suggested solutions.

Software versions

- ASP.NET Web Pages (Razor) 3

This tutorial also works with ASP.NET Web Pages 2 and ASP.NET Web Pages 1.0.

This topic contains the following sections:

- Issues with Running Pages
- Issues with Razor Code
- Issues with Security and Membership
- Issues with Sending Email
- Additional Resources

For general questions, see ASP.NET Web Pages (Razor) FAQ.

Issues with Running Pages

A variety of issues can prevent *.cshtml* and *.vbhtml* pages from running properly. This section lists common error messages and likely causes.

HTTP Error 403 - Forbidden: Access is denied

You do not have permission to view this directory or page using the credentials that you supplied.

This error can occur if the server is not running the correct version of the .NET Framework. Make sure that the computer that's running the server (locally or remotely) has at least the .NET Framework 4 installed. Also make sure that the application itself is configured to run the right version.

If you see this problem locally while working in WebMatrix, click the **Site** workspace, and then in the treeview click **Settings**. In the **Select .NET Framework Version** list, select **.NET 4 (Integrated)**. If this version is already set, try running WebMatrix as an administrator.

Make sure that the root of your website has at least one *.cshtml* file in it.

If you see this error when the web server is on a remote server, contact the server administrator. Make sure that the server has the .NET Framework 4 or later installed. Also make sure that the application is running in an application pool that's configured to use that version of the.NET Framework.

If you have control over the server, make sure it's running the correct version of the .NET Framework. You might also try repairing the installation by running the `aspnet_regiis -iru` command. (For example, if you install IIS after you install the .NET Framework, IIS will not be correctly configured to run ASP.NET pages.) For more information, see ASP.NET IIS Registration Tool (Aspnet_regiis.exe).

HTTP Error 403.14 - Forbidden

The Web server is configured to not list the contents of this directory.

This error can occur if you request a resource that's protected (like the *Web.config* file) or that's in a folder that's protected (like *App_Data* or *App_Code*).

HTTP Error 404.17 - Not Found

The requested content appears to be script and will not be served by the static file handler.

This error can occur if the server is not configured correctly to use the .NET Framework 4 or later, and therefore does not recognize the code in `@{ }` blocks. See the description earlier for *HTTP Error 403 - Forbidden: Access is denied*.

HTTP Error 404.7 - Not Found

The request filtering module is configured to deny the file extension

This error can occur if *.cshtml* or *.vbhtml* extensions have been explicitly blocked on the server. A symptom of this problem is that URLs work when they do not include the extension, but URLs that include *.cshtml* or *.vbhtml* do not work. A possible solution is to re-enable the extensions in the site's *Web.config* file. The following example shows how to enable the *.cshtml* extension.

```
<system.webServer>
  <security>
   <requestFiltering>
     <fileExtensions>
       <remove fileExtension=".cshtml" />
       <add fileExtension=".cshtml" allowed="true" />
     </fileExtensions>
    </requestFiltering>
  </security>
</system.webServer>
```

HTTP Error 404.8 - Not Found

The request filtering module is configured to deny a path in the URL that contains a hiddenSegment section.

This error can occur if you request a resource that's protected (like the *Web.config* file) or that's in a folder that's protected (like *App_Data* or *App_Code*).

This type of page is not served (Server Error in '/' Application)

See the description earlier for HTTP Error 404.17.

Issues with Razor code

The name '*class*' does not exist in the current context

Often, a reason you see this error is that `class` references a helper, but the helper is not installed. For example, if you try to use a helper, but if you haven't installed the package from NuGet, you'll see this error. Use the Gallery in WebMatrix to find and install the helper.

If the helper is installed, but the page still doesn't recognize it, try adding add a `using` statement to the code. In the `using` statement, reference the namespace that includes the helper. For example, the basic helpers that are in the ASP.NET Web Helpers package are in the `System.Web.Helpers` namespace. At the top of the page where you want to use the helper, add this line:

```
@using Microsoft.Web.Helpers;
```

Issues with Security and Membership

If you are using the built-in security (membership) system in ASP.NET Web Pages (Razor), you might encounter the

following issues.

To call this method, the "Membership.Provider" property must be an instance of "ExtendedMembershipProvider"

This error can indicate that no `AspNetSqlMembershipProvider` class is configured. (A symptom is that the site works fine locally but throws this error when you publish it to a hosting provider's server.) One fix for this problem is to explicitly enable simple membership by adding the following to the site's *Web.config* file:

```
<configuration>

  <!-- other setting -->

  <appSettings>
    <add key="enableSimpleMembership" value="true" />
  </appSettings>

  <!-- other settings -->

</configuration>
```

Issues with Sending Email

Problems with sending email can be challenging to debug. An initial problem can be that you can't connect to the SMTP server. If the connection is successful, ASP.NET hands the message off to the SMTP server. However, there can be problems with the message itself that prevents the SMTP server from sending it.

If your application does not successfully send email, try the following:

- The SMTP server name is often something like `smtp.provider.com` or `smtp.provider.net`. However, if you publish your site to a hosting provider, the SMTP server name at that point might be `localhost`. This situation occurs because after you've published and your site is running on the provider's server, the SMTP server might be local from the perspective of your application. This change in server names might mean that you have to change the SMTP server name as part of your publishing process.
- The port number is usually 25. However, some providers require you to use port 587 or some other port. Check with the owner of the SMTP server what port number they expect you to use.
- Make sure that you use the right credentials. If you've published your site to a hosting provider, use the credentials that the provider has specifically indicated are for email. These credentials might be different from the credentials you use to publish.
- Sometimes you don't need credentials at all. If you're sending email by using your personal ISP, your email provider might already know your credentials. After you publish, you might need to use different credentials than when you test on your local computer.
- If your email provider uses encryption, set `WebMail.EnableSsl` to `true`.

If there is an error sending email, you might see a standard ASP.NET error message, which looks like this:

Server Error in '/' Application.

The specified string is not in the form required for an e-mail address.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more code.

Exception Details: System.FormatException: The specified string is not in the form required for an e-mail address.

Source Error:

```
Line 16:
Line 17:        // Send email
Line 18:        WebMail.Send(to: customerEmail,
Line 19:            subject: "Help request from - " + customerName,
Line 20:            body: customerRequest
```

You can also debug problems with sending email by using a `try-catch` block, as in the following example. When you use a `try-catch` block, ASP.NET does not display its standard error messages. Instead, you can capture the error in the `catch` portion of the block.

```
var errorMessage = "";
try {
    // Initialize WebMail helper
    WebMail.SmtpServer = "your-SMTP-server-name";
    WebMail.SmtpPort = 25;   // Or the port you've been told to use
    WebMail.EnableSsl = false;
    WebMail.UserName = "your-login-name";
    WebMail.Password = "your-password";
    WebMail.From = "your-from-address";

    WebMail.Send(to: test-To-address,
        subject: "Test email message",
        body: "This is a debug email message"
    );
}
catch (Exception ex ) {
errorMessage = ex.Message;
}

// Other code or markup here ...

<!-- In markup, add the following -->
@if(!errorMessage.IsEmpty()){
    <p>@errorMessage</p>
}
```

Substitute the appropriate values for `your-SMTP-server-name`, and so on. Some of the error messages you might see this way include the following:

- *Failure sending mail.*

  -or-

  A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond

  This error usually means that the application could not connect to the SMTP server. Check the server name and port number.

- *Mailbox unavailable. The server response was: 5.1.0 <someuser@invaliddomain> sender rejected : invalid sender domain*

  This message can indicate that the `From` address is not correct or is missing.

- *The specified string is not in the form required for an email address.*

This error might indicate that the value of the `To` or `From` properties are not recognized as email addresses. (ASP.NET cannot check that the email address is valid, only that it's in the correct format, like *name@domain.com*.)

> **NOTE**
>
> Remove the markup that displays the error (`@errorMessage`) before you publish the page to a live site. It's not a good idea to let users see error messages that you get from a server.

Additional Resources

ASP.NET Web Pages (Razor) FAQ

WebMatrix and ASP.NET Web Pages forum on the ASP.NET website

Security

2/20/2020 • 2 minutes to read • Edit Online

How to add login capabilities to an ASP.NET Web Pages application, and how to help protect the application from bots.

- Adding Security and Membership
- Enabling Login from External Sites in an ASP.NET Web Pages Site
- Using a CAPTCHA to Prevent Automated Programs (Bots) from Using Your ASP.NET Web Site

Adding Security and Membership to an ASP.NET Web Pages (Razor) Site

2/20/2020 • 15 minutes to read • Edit Online

by Tom FitzMacken

This article explains how to secure an ASP.NET Web Pages (Razor) website so that some of the pages are available only to people who log in. (You'll also see how to create pages that anyone can access.)

What you'll learn:

- How to create a website that has a registration page and a login page so that for some pages you can limit access to only members.
- How to create public and member-only pages.
- How to define roles, which are groups that have different security permissions on your site, and how to assign users to a role.
- How to use CAPTCHA to prevent automated programs (bots) from creating member accounts.

These are the ASP.NET features introduced in the article:

- The WebMatrix **Starter Site** template.
- The `WebSecurity` helper and `Roles` class.
- The `ReCaptcha` helper.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 2
- WebMatrix 3
- ASP.NET Web Helpers Library

You can set up your website so that users can log into it — that is, so that the site supports *membership*. This can be useful for many reasons. For example, your site might have pages that should be available only to members. In some cases, you might require users to log in in order to send you feedback or leave a comment.

Even if your website supports membership, users aren't necessarily required to log in before they use some of the pages on the site. Users who aren't logged in are known as *anonymous users*.

A user can register on your website and can then log in to the site. The website requires a user name (an email address) and a password to confirm that users are who they claim to be. This process of logging in and confirming a user's identity is known as *authentication*.

You can set up security and membership in different ways:

- If you're using WebMatrix, an easy way is to create as new site based on the **Starter Site** template. This template is already configured for security and membership and already has a registration page, a login page, and so on.

  The site created by the template also has an option to let users log in using an external site like Facebook, Google, or Twitter.

- If you want to add security to an existing site, or if you don't want to use the **Starter Site** template, you can

create your own registration page, login page, and so on.

This article focuses on the first option — how to add security by using the **Starter Site** template. It also provides some basic information about how to implement your own security and then provides links to more information about how to do that. There is also information about how to enable external logins, which is described in more detail in a separate article.

Creating Website Security Using the Starter Site Template

In WebMatrix, you can use the **Starter Site** template to create a website that contains the following:

- A database that's used to store user names and passwords for your members.
- A registration page where anonymous (new) users can register.
- A login and logout page.
- A password recovery and reset page.

The following procedure describes how to create the site and configure it.

1. Start WebMatrix and in the **Quick Start** page, select **Site From Template**.

2. Select the **Starter Site** template and then click **OK**. WebMatrix creates a new site.

3. In the left pane, click the **Files** workspace selector.

4. In the root folder of your website, open the *_AppStart.cshtml* file, which is a special file that's used to contain global settings. It contains some statements that are commented out using the `//` characters:

   ```
   //WebMail.SmtpServer = "mailserver.example.com";
   //WebMail.EnableSsl = true;
   //WebMail.UserName = "username@example.com";
   //WebMail.Password = "your-password";
   //WebMail.From = "your-name-here@example.com";
   ```

   These statements configure the `WebMail` helper, which can be used to send email. The membership system can use email to send confirmation messages when users register or when they want to change their passwords. (For example, after users register, they get an email that includes a link that they can click in order to finish the registration process.)

   Sending email requires access to an SMTP server, as described in Adding Email to an ASP.NET Web Pages Site. You'll store the email settings in this central *_AppStart.cshtml* file so that you don't have to code them repeatedly into each page that can send email. (You don't need to configure SMTP settings to set up a registration database; you only need SMTP settings if you want to validate users from their email alias and let users reset a forgotten password.)

5. Uncomment the statements by removing `//` from in front of each one.

   If you do not want to set up email confirmation, you can skip this step and the next step. If the SMTP values are not set, the new account is immediately available without a confirmation email.

6. Modify the following email-related settings in the code:

   - Set `WebMail.SmtpServer` to the name of the SMTP server that you have access to.

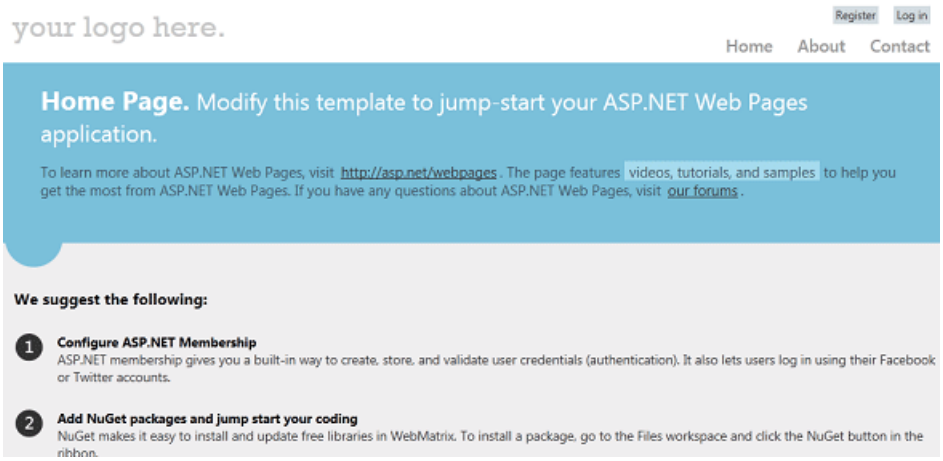
   - Leave `WebMail.EnableSsl` set to `true`. This setting secures the credentials that are sent to the SMTP server by encrypting them.

   - Set `WebMail.UserName` to the user name for your SMTP server account.

- Set `WebMail.Password` to the password for your SMTP server account.
- Set `WebMail.From` to your own email address. This is the email address that the message is sent from.

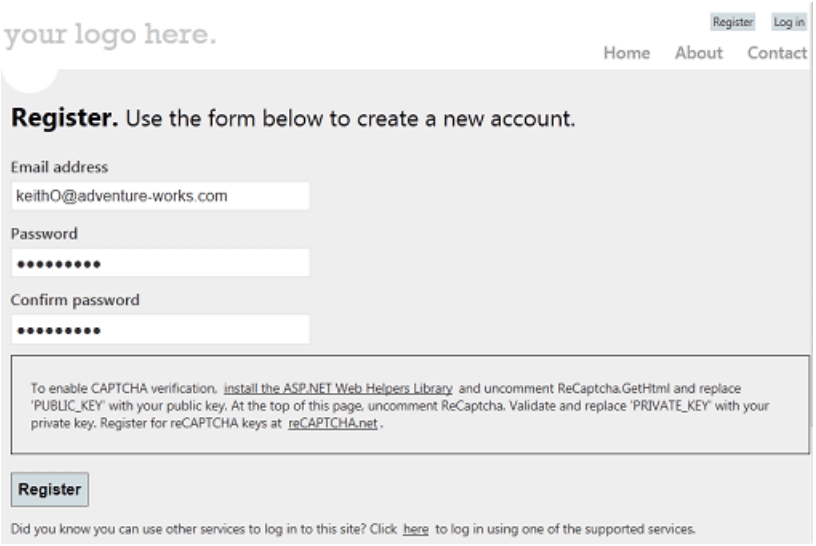
> **NOTE**
>
> **Tip** For additional information about the values for these properties, see Configuring Email Settings in Customizing Site-Wide Behavior for ASP.NET Web Pages.

7. Save and close *_AppStart.cshtml*.
8. Run the *Default.cshtml* page in a browser.

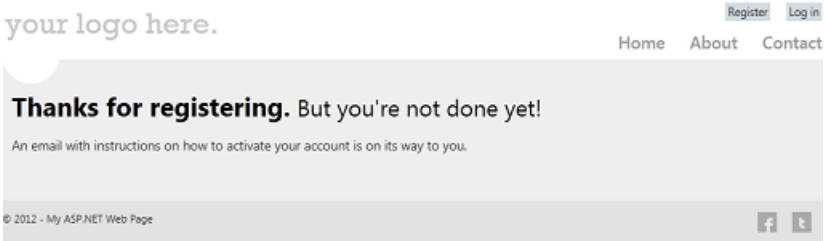


> **NOTE**
>
> If you see an error that tells you that a property must be an instance of `ExtendedMembershipProvider`, the site might not be configured to use the ASP.NET Web Pages membership system (SimpleMembership). This can sometimes occur if a hosting provider's server is configured differently than your local server. To fix this, add the following element to the site's *Web.config* file:
>
> ```
> <appSettings>
>  <add key="enableSimpleMembership" value="true" />
> </appSettings>
> ```
>
> Add this element as a child of the `<configuration>` element and as a peer of the `<system.web>` element.

9. In the upper-right corner of the page, click the **Register** link. The *Register.cshtml* page is displayed.
10. Enter a user name and password and then click **Register**.

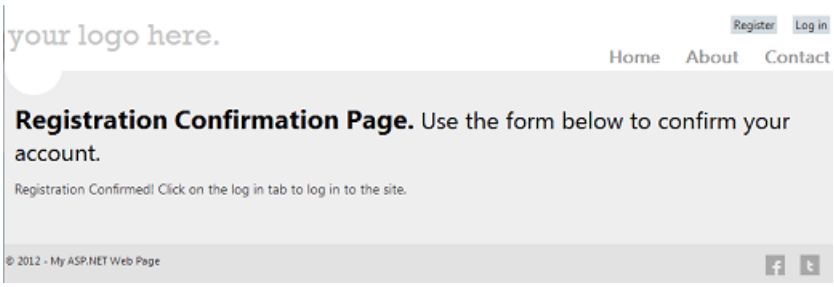


When you created the website from the **Starter Site** template, a database named *StarterSite.sdf* was created in the site's *App_Data* folder. During registration, your user information is added to the database. If you set the SMTP values, a message is sent to the email address you used so you can finish registering.



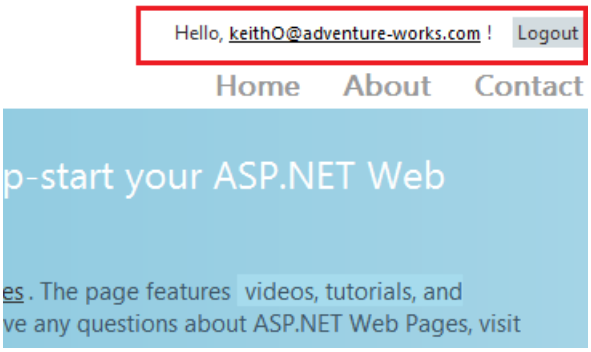
11. Go to your email program and find the message, which will have your confirmation code and a hyperlink to the site.

12. Click the hyperlink to activate your account. The confirmation hyperlink opens a registration confirmation page.



13. Click the **Login** link, and then log in using the account that you registered.

    After you log in, the **Login** and **Register** links are replaced by a **Logout** link. Your login name is displayed as a link. (The link lets you go to a page where you can change your password.)



> **NOTE**
> By default, ASP.NET web pages send credentials to the server in clear text (as human-readable text). A production site should use secure HTTP (https://, also known as the *secure sockets layer* or SSL) to encrypt sensitive information that's exchanged with the server. You can required email messages to be sent using SSL by setting `WebMail.EnableSsl=true` as in the previous example. For more information about SSL, see Securing Web Communications: Certificates, SSL, and https://.

Additional Membership Functionality in the Site

Your site contains other functionality that lets users manage their accounts. Users can do the following:

- Change their passwords. After logging in, they can click the user name (which is a link). This takes them to a page where they can create a new password (*Account/ChangePassword.cshtml*).
- Recover a forgotten password. On the login page, there's a link (**Did you forget your password?**) that takes users to a page (*Account/ForgotPassword.cshtml*) where they can enter an email address. The site sends them an email message that has a link that they can click in order to set a new password (*Account/PasswordReset.cshtml*).

You can also let users can also log in using an external site, as explained later.

Creating a Members-Only Page

For the time being, anyone can browse to any page in your website. But you might want to have pages that are available only to people who have logged in (that is, to members). ASP.NET lets you create pages that can be accessed only by logged-in members. Typically, if anonymous users try to access a members-only page, you redirect them to the login page.

In this procedure, you'll create a folder that will contain pages that are available only to logged-in users.

1. At the root of the site, create a new folder. (In the ribbon, click the arrow underneath **New** and then choose **New Folder**.)

2. Name the new folder *Members*.

3. Inside the *Members* folder, create a new page and named it *MembersInformation.cshtml*.

4. Replace the existing content with the following code and markup:

```
@{
    if (!WebSecurity.IsAuthenticated) {
        Response.Redirect("~/Account/Login?returnUrl="
            + Request.Url.LocalPath);
    }
    Layout = "~/_SiteLayout.cshtml";
    Page.Title = "Members Information";
}
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Members Information</title>
  </head>
  <body>
    <p>You can only see this information if you've logged into the site.</p>
  </body>
</html>
```

This code tests the `IsAuthenticated` property of the `WebSecurity` object, which returns `true` if the user has logged in. If the user is not logged in, the code calls `Response.Redirect` to send the user to the *Login.cshtml* page in the *Account* folder.

The URL of the redirection includes a `returnUrl` query string value that uses `Request.Url.LocalPath` to set the path of the current page. If you set the `returnUrl` value in the query string like this (and if the return URL is a local path), the login page will return users to this page after they log in.

The code also sets *_SiteLayout.cshtml* page as its layout page. (For more about layout pages, see Creating a Consistent Layout in ASP.NET Web Pages Sites.)

5. Run the site. If you're still logged in, click the **Logout** button at the top of the page.

6. In the browser, request the page */Members/MembersInformation*. For example, the URL might look like this:

   ```
   http://localhost:38366/Members/MembersInformation
   ```

   (The port number (38366) will probably be different in your URL.)

   You're redirected to the *Login.cshtml* page, because you aren't logged in.

7. Log in using the account you created earlier. You're redirected back to the *MembersInformation* page. Because you're logged in, this time you see the page contents.

To secure access to multiple pages, you can do this:

- Add the security check to each page.
- Create a *_PageStart.cshtml* page in the folder where you keep protected pages, and add the security check there. The *_PageStart.cshtml* page acts as a kind of global page for all the pages in the folder. This technique is explained in more detail in Customizing Site-Wide Behavior for ASP.NET Web Pages.

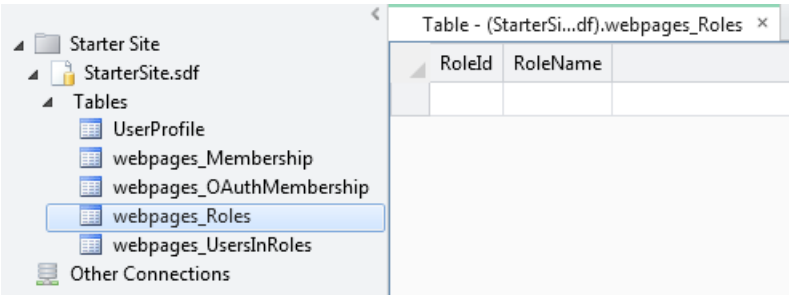
Creating Security for Groups of Users (Roles)

If your site has a lot of members, it's not efficient to check permission for each user individually before you let them see a page. What you can do instead is to create groups, or *roles*, that individual members belong to. You can then check permissions based on role. In this section, you'll create an "admin" role and then create a page that's accessible to users who are in (who belong to) that role.

The ASP.NET membership system is set up to support roles. However, unlike membership registration and login, the **Starter Site** template does not contain pages that help you manage roles. (Managing roles is an administrative task rather than a user task.) However, you can add groups directly in the membership database in WebMatrix.

1. In WebMatrix, click the **Databases** workspace selector.

2. In the left pane, open the *StarterSite.sdf* node, open the **Tables** node, and then double-click the *webpages_Roles* table.



3. Add a role named "admin". The *RoleId* field is filled in automatically. (It's the primary key and has been set

to be an identify field, as explained in Introduction to Working with a Database in ASP.NET Web Pages Sites.)

4. Take note of what the value is for the *RoleId* field. (If this is the first role you're defining, it will be 1.)

Table - (StarterSi...df).webpages_Roles

RoleId	RoleName
1	admin

5. Close the *webpages_Roles* table.

6. Open the *UserProfile* table.

7. Make a note of the *UserId* value of one or more of the users in the table and then close the table.

8. Open the *webpages_UserInRoles* table and enter a *UserID* and a *RoleID* value into the table. For example, to put user 2 into the "admin" role, you'd enter these values:

Table - (StarterSi...pages_UsersInRoles

UserId	RoleId
2	1

9. Close the *webpages_UsersInRoles* table.

   Now that you have roles defined, you can configure a page that's accessible to users who are in that role.

10. In the website root folder, create a new page named *AdminError.cshtml* and replace the existing content with the following code. This will be the page that users are redirected to if they aren't allowed access to a page.

```
@{
    Layout = "~/_SiteLayout.cshtml";
    PageData["Title"] = "Admin-only Error";
}
<p>You must log in as an admin to access that page.</p>
```

11. In the website root folder, create a new page named *AdminOnly.cshtml* and replace the existing code with the following code:

```
@{
    Layout = "~/_SiteLayout.cshtml";
    PageData["Title"] = "Administrators only";
}

@if ( Roles.IsUserInRole("admin")) {
    <span> Welcome <b>@WebSecurity.CurrentUserName</b>! </span>
}
else {
     Response.Redirect("~/AdminError");
}
```

    The `Roles.IsUserInRole` method returns `true` if the current user is a member of the specified role (in this case, the "admin" role).

12. Run *Default.cshtml* in a browser, but don't log in. (If you're already logged in, log out.)

13. In the browser's address bar, add *AdminOnly* in the URL. (In other words, request the *AdminOnly.cshtml* file.) You're redirected to the *AdminError.cshtml* page, because you aren't currently logged in as a user in the "admin" role.

14. Return to *Default.cshtml* and log in as the user you added to the "admin" role.
15. Browse to *AdminOnly.cshtml* page. This time you see the page.

Preventing Automated Programs from Joining Your Website

The login page will not stop automated programs (sometimes referred to as *web robots* or *bots*) from registering with your website. This procedure describes how to enable a ReCaptcha test for the registration page.

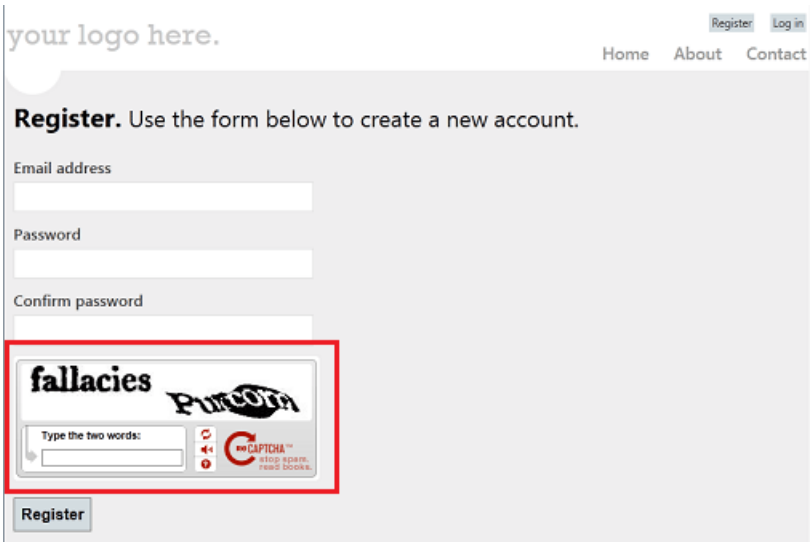
1. Register your website at ReCaptcha.Net (http://recaptcha.net). When you've completed registration, you'll get a public key and a private key.
2. Add the ASP.NET Web Helpers Library to your website as described in Installing Helpers in an ASP.NET Web Pages Site, if you haven't already.
3. In the *Account* folder, open the file named *Register.cshtml*.
4. In the code at the top of the page, find the following lines and uncomment them by removing the `//` comment characters:

```
if (!ReCaptcha.Validate("PRIVATE_KEY")) {
    ModelState.AddError("recaptcha", "Captcha response was not correct");
}
```

5. Replace `PRIVATE_KEY` with your own ReCaptcha private key.
6. In the markup of the page, remove the `@*` and `*@` commenting characters from around the following lines in the page markup:

```
@ReCaptcha.GetHtml("PUBLIC_KEY", theme: "white")
@Html.ValidationMessage("recaptcha")
```

7. Replace `PUBLIC_KEY` with your key.
8. If you haven't removed it already, remove the `<div>` element that contains text that starts with "To enable CAPTCHA verification ...". (Remove the entire `<div>` element and its contents.)
9. Run *Default.cshtml* in a browser. If you're logged into the site, click the **Logout** link.
10. Click the **Register** link and test the registration using the CAPTCHA test.



For more information about the `ReCaptcha` helper, see Using a CATPCHA to Prevent Automated Programs (Bots) from Using Your ASP.NET Web Site.

Letting Users Log In Using an External Site

The **Starter Site** template includes code and markup that lets users log in using Facebook, Windows Live, Twitter, Google, or Yahoo. By default, this functionality is not enabled. The general procedure for using letting users log in using these external providers is this:

- Decide which of the external sites you want to support.
- If required, go to that site and set up a login app. (For example, you have to do this in order to allow Facebook logins.)
- In your site, configure the provider. In most cases, you just have to uncomment some code in the *_AppStart.cshtml* file.
- Add markup to the registration page that lets people link to the external site for logging in. You can usually copy the markup you need and change the text slightly.

You can find step-by-step instructions in the topic Enabling Login from External Sites in an ASP.NET Web Pages Site.

After a user logs in from another site, the user returns to your site and *associates* that login with your site. In effect, this creates a membership entry in your site for the user's external login. This lets you use the normal facilities of membership (such as roles) with the external login.

Adding Security to an Existing Website

The procedure earlier in this article relies on using the **Starter Site** template as the basis for website security. If it isn't practical for you to start from the **Starter Site** template or to copy the relevant pages from a site based on that template, you can implement the same type of security in your own site by coding it yourself. You create the same types of pages — registration, login, and so on — and then use helpers and classes to set up membership.

The basic process is described in the blog post THE most basic way to implement ASP.NET Razor security. Most of the work is done using the following methods and properties of the `WebSecurity` helper:

- WebSecurty.UserExists, WebSecurity.CreateUserAndAccount. These methods let you determine whether someone is already registered and to register them.
- WebSecurty.IsAuthenticated. This property lets you determine whether the current user is logged in. This is useful to redirect users to a login page if they have not already logged in.
- WebSecurity.Login, WebSecurity.Logout. These methods log a user in or out.

- WebSecurity.CurrentUserName. This property is useful for displaying the current user's logged-in name (if the user is logged in).
- WebSecurity.ConfirmAccount. This method is useful if you set up email confirmation for registration. (Details are described in the blog post Using the confirmation feature for ASP.NET Web Pages security.)

To manage roles, you can use the Roles and Membership classes, as described in the blog entry.

Additional Resources

- Customizing Site-Wide Behavior
- Securing Web Communications: Certificates, SSL, and https://
- THE most basic way to implement ASP.NET Razor security and Using the confirmation feature for ASP.NET Web Pages security. These are blog posts that describe how to implement ASP.NET membership features without using the **Starter Site** template.
- Enabling Login from External Sites in an ASP.NET Web Pages Site
- WebSecurity Class API reference (MSDN)
- SimpleRoleProvider Class API reference (MSDN)
- SimpleMembershipProvider Class API reference (MSDN)

Logging In Using External Sites in an ASP.NET Web Pages (Razor) Site

2/20/2020 • 7 minutes to read • Edit Online

by Tom FitzMacken

This article explains how to log in to your ASP.NET Web Pages (Razor) site using Facebook, Google, Twitter, Yahoo, and other sites — that is, how to support OAuth and OpenID in your site.

What you'll learn:

- How to enable login from other sites when you use the WebMatrix Starter Site template.

This is the ASP.NET feature introduced in the article:

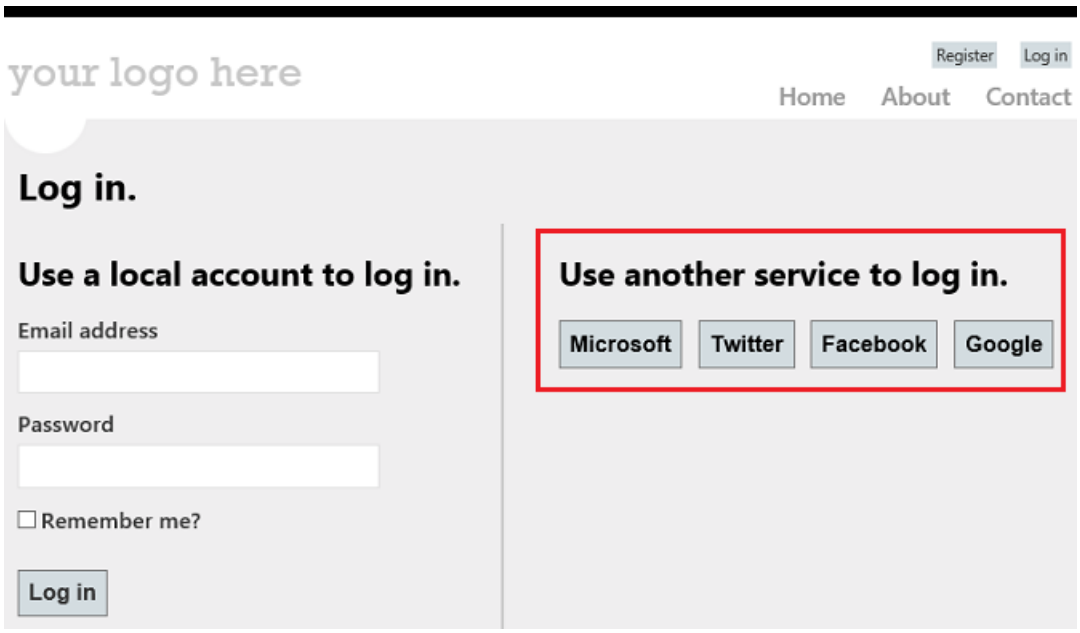
- The `OAuthWebSecurity` helper.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 2
- WebMatrix 3

ASP.NET Web Pages includes support for OAuth and OpenID providers. Using these providers, you can let users log into your site using their existing credentials from Facebook, Twitter, Microsoft, and Google. For example, to log in using a Facebook account, users can just choose a Facebook icon, which redirects them to the Facebook login page where they enter their user information. They can then associate the Facebook login with their account on your site. A related enhancement to the Web Pages membership features is that users can associate multiple logins (including logins from social networking sites) with a single account on your website.

This image shows the Login page from the **Starter Site** template, where a user can choose a Facebook, Twitter, Google or Microsoft icon to enable logging in with an external account:



You can enable OAuth and OpenID membership by uncommenting a few lines of code in the **Starter Site** template. The methods and properties you use to work with the OAuth and OpenID providers are in the

`WebMatrix.Security.OAuthWebSecurity` class. The **Starter Site** template includes a full membership infrastructure, complete with a login page, a membership database, and all the code you need to let users log into your site using either local credentials or those from another site.

This section provides an example of how to let users log in from external sites to a site that's based on the **Starter Site** template. After creating a starter site, you do this (details follow):

- For the sites that use an OAuth provider (Facebook, Twitter, and Microsoft), you create an application on the external site. This gives you application keys that you'll need in order to invoke the login feature for those sites.
- For sites that use an OpenID provider (Google), you do not have to create an application. For all of these sites, you must have an account in order to log in and to create developer applications.

> **NOTE**
> Microsoft applications only accept a live URL for a working website, so you cannot use a local website URL for testing logins.

- Edit a few files in your website in order to specify the appropriate authentication provider and to submit a login to the site you want to use.

This article provides separate instructions for the following tasks:

- Enabling Google logins
- Enabling Facebook logins
- Enabling Twitter logins

Enabling Google Logins

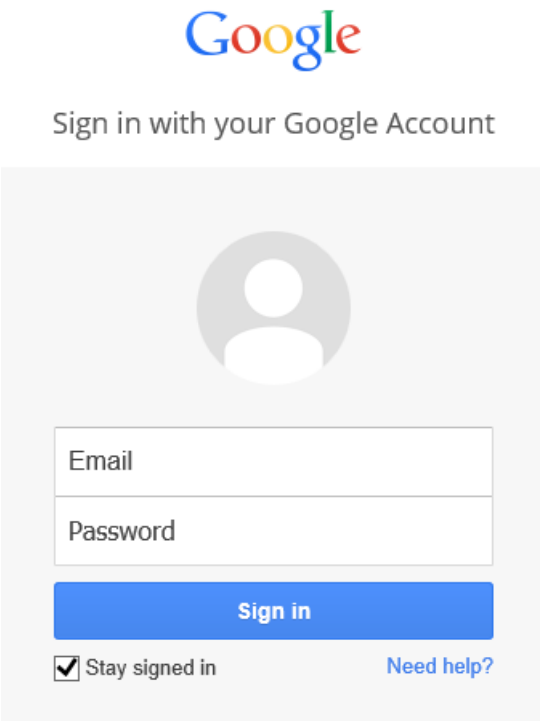
1. Create or open an ASP.NET Web Pages site that's based on the WebMatrix Starter Site template.
2. Open the *_AppStart.cshtml* page and uncomment the following line of code.

```
OAuthWebSecurity.RegisterGoogleClient();
```

Testing Google login

1. Run the *default.cshtml* page of your site and choose the **Log in** button.
2. On the *Login* page, in the **Use another service to log in** section, choose either the **Google** or **Yahoo** submit button. This example uses the Google login.

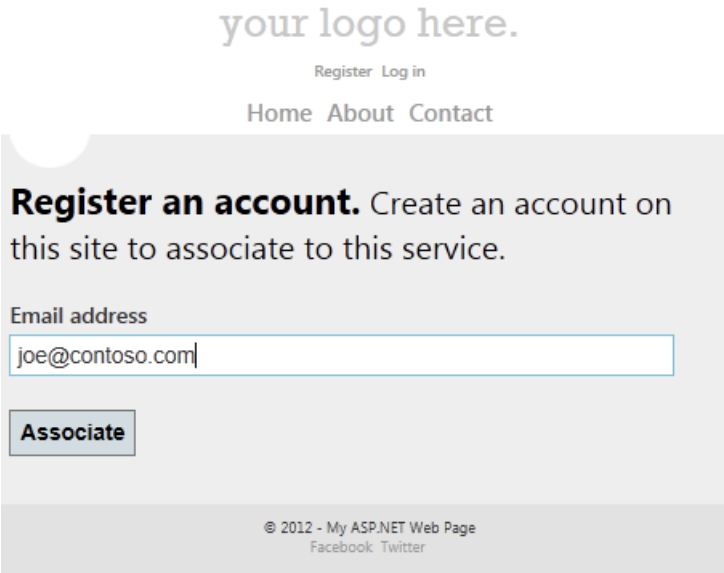
   The web page redirects the request to the Google login page.



3. Enter credentials for an existing Google account.

4. If Google asks you whether you want to allow *Localhost* to use information from the account, click **Allow**.

   The code uses the Google token to authenticate the user, and then returns to this page on your website. This page lets users associate their Google login with an existing account on your website, or they can register a new account on your site to associate the external login with.



5. Choose the **Associate** button. The browser returns to your application's home page.

Enabling Facebook Logins

1. Go to the Facebook developers site (log in if you're not already logged in).

2. Choose the **Create New App** button, and then follow the prompts to name and create the new application.

3. In the section **Select how your app will integrate with Facebook**, choose the **Website** section.

4. Fill in the **Site URL** field with the URL of your site (for example, `http://www.example.com`). The **Domain** field is optional; you can use this to provide authentication for an entire domain (such as *example.com*).

> **NOTE**
> If you are running a site on your local computer with a URL like `http://localhost:12345` (where the number is a local port number), you can add this value to the **Site URL** field for testing your site. However, any time the port number of your local site changes, you will need to update the **Site URL** field of your application.

5. Choose the **Save Changes** button.
6. Choose the **Apps** tab again, and then view the start page for your application.
7. Copy the **App ID** and **App Secret** values for your application and paste them into a temporary text file. You will pass these values to the Facebook provider in your website code.
8. Exit the Facebook developer site.

Now you make changes to two pages in your website so that users will able to log into the site using their Facebook accounts.

1. Create or open an ASP.NET Web Pages site that's based on the WebMatrix Starter Site template.
2. Open the *_AppStart.cshtml* page and uncomment the code for the Facebook OAuth provider. The uncommented code block looks like the following:

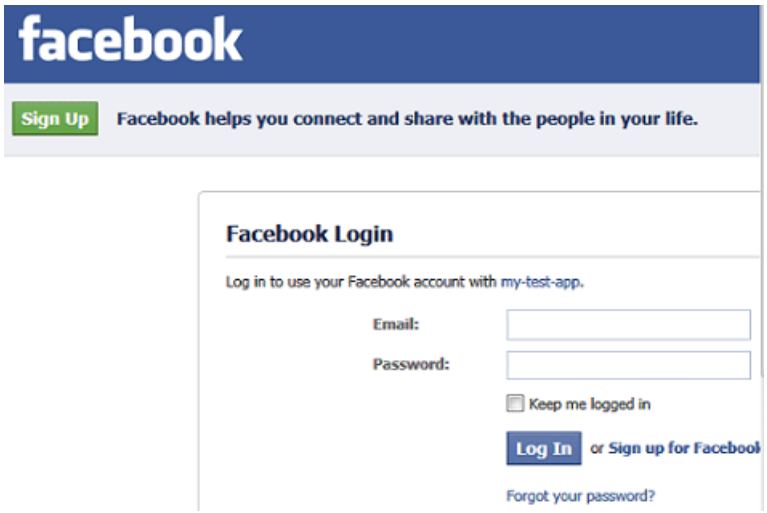
```
OAuthWebSecurity.RegisterFacebookClient(
        appId: "",
        appSecret: "");
```

3. Copy the **App ID** value from the Facebook application as the value of the `appId` parameter (inside the quotation marks).
4. Copy **App Secret** value from the Facebook application as the `appSecret` parameter value.
5. Save and close the file.

Testing Facebook login

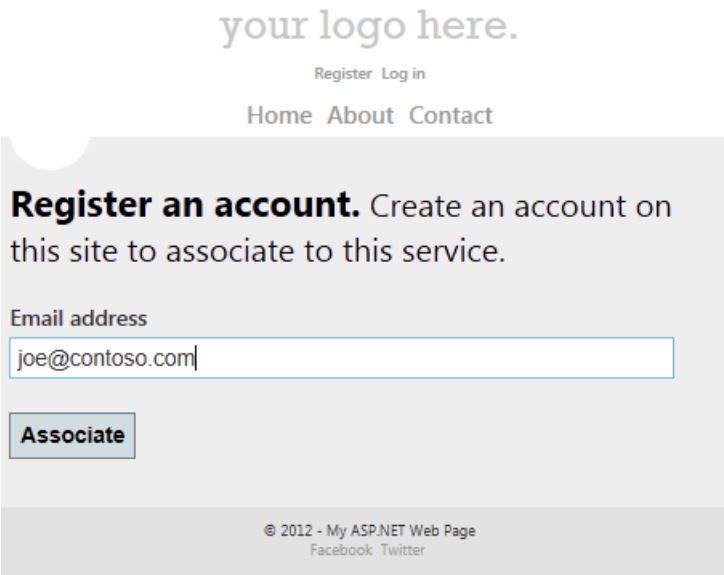
1. Run the site's *default.cshtml* page and choose the **Login** button.
2. On the *Login* page, in the **Use another service to log in** section, choose the **Facebook** icon.

   The web page redirects the request to the Facebook login page.



3. Log into a Facebook account.

The code uses the Facebook token to authenticate you and then returns to a page where you can associate your Facebook login with your site's login. Your user name or email address is filled into the **Email** field on the form.



4. Choose the **Associate** button.

   The browser returns to the home page and you are logged in.

Enabling Twitter Logins

1. Browse to the Twitter developers site.
2. Choose the **Create an App** link and then log into the site.
3. On the **Create an Application** form, fill in the **Name** and **Description** fields.
4. In the **WebSite** field, enter the URL of your site (for example, `http://www.example.com`).

   > **NOTE**
   > If you're testing your site locally (using a URL like `http://localhost:12345`), Twitter might not accept the URL. However, you might be able to use the local loopback IP address (for example `http://127.0.0.1:12345`). This simplifies the process of testing your application locally. However, every time the port number of your local site changes, you'll need to update the **WebSite** field of your application.

5. In the **Callback URL** field, enter a URL for the page in your website that you want users to return to after logging into Twitter. For example, to send users to the home page of the Starter Site (which will recognize their logged-in status), enter the same URL that you entered in the **WebSite** field.
6. Accept the terms and choose the **Create your Twitter application** button.
7. On the **My Applications** landing page, choose the application you created.
8. On the **Details** tab, scroll to the bottom and choose the **Create My Access Token** button.
9. On the **Details** tab, copy the **Consumer Key** and **Consumer Secret** values for your application and paste them into a temporary text file. You'll pass these values to the Twitter provider in your website code.
10. Exit the Twitter site.

Now you make changes to two pages in your website so that users will be able to log into the site using their

Twitter accounts.

1. Create or open an ASP.NET Web Pages site that's based on the WebMatrix Starter Site template.
2. Open the *_AppStart.cshtml* page and uncomment the code for the Twitter OAuth provider. The uncommented code block looks like this:

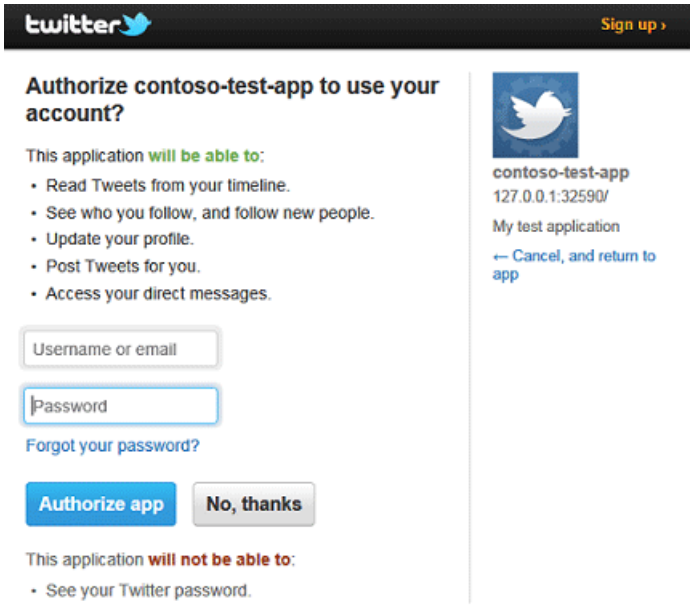
```
OAuthWebSecurity.RegisterTwitterClient(
        consumerKey: "",
        consumerSecret: "");
```

3. Copy the **Consumer Key** value from the Twitter application as the value of the `consumerKey` parameter (inside the quotation marks).
4. Copy the **Consumer Secret** value from the Twitter application as the value of the `consumerSecret` parameter.
5. Save and close the file.

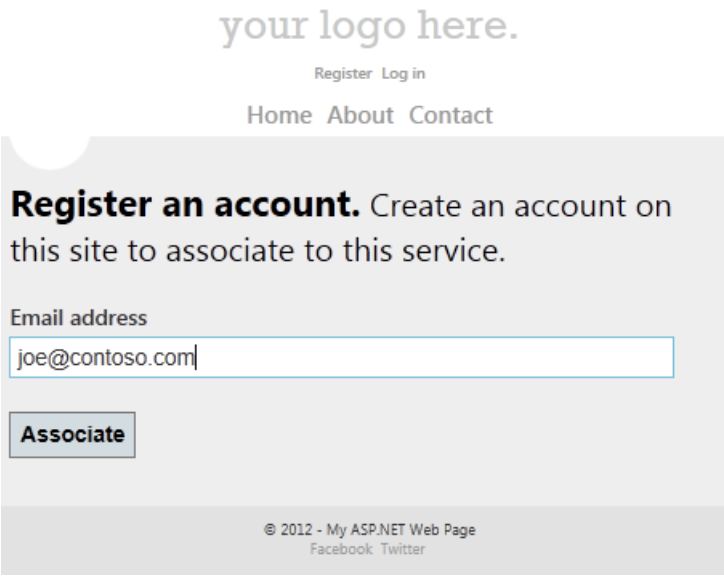
Testing Twitter login

1. Run the *default.cshtml* page of your site and choose the **Login** button.
2. On the *Login* page, in the **Use another service to log in** section, choose the **Twitter** icon.

   The web page redirects the request to a Twitter login page for the application you created.



3. Log into a Twitter account.
4. The code uses the Twitter token to authenticate the user and then returns you to a page where you can associate your login with your website account. Your name or email address is filled into the **Email** field on the form.



5. Choose the **Associate** button.

   The browser returns to the home page and you are logged in.

Additional Resources

- Customizing Site-Wide Behavior
- Adding Security and Membership to an ASP.NET Web Pages Site

Using a CAPTCHA to Prevent Bots from Using Your ASP.NET Web Razor) Site

4/10/2020 • 3 minutes to read • Edit Online

by Microsoft

This article explains how to use ReCaptcha (a security measure) to prevent automated programs (bots) from performing tasks in an ASP.NET Web Pages (Razor) website.

What you'll learn:

- How to add a CAPTCHA test to your site.

These are the ASP.NET features introduced in the article:

- The `ReCaptcha` helper.

> **NOTE**
>
> The information in this article applies to ASP.NET Web Pages 1.0 and Web Pages 2.

About CAPTCHAs

Any time you let people register in your site, or even just enter a name and URL (like for a blog comment), you might get a flood of fake names. These are often left by automated programs (bots) that try to leave URLs in every website they can find. (A common motivation is to post the URLs of products for sale.)

You can help make sure that a user is real person and not a computer program by using a *CAPTCHA* to validate users when they register or otherwise enter their name and site. CAPTCHA stands for Completely Automated Public Turing test to tell Computers and Humans Apart. A CAPTCHA is a *challenge-response* test in which the user is asked to do something that is easy for a person to do but hard for an automated program to do. The most common type of CAPTCHA is one where you see some distorted letters and are asked to type them. (The distortion is supposed to make it hard for bots to decipher the letters.)

Adding a ReCaptcha Test

In ASP.NET pages, you can use the `ReCaptcha` helper to render a CAPTCHA test that is based on the ReCaptcha service (http://recaptcha.net). The `ReCaptcha` helper displays an image of two distorted words that users have to enter correctly before the page is validated. The user response is validated by the ReCaptcha.Net service.

1. Register your website at ReCaptcha.Net (http://recaptcha.net). When you've completed registration, you'll get a public key and a private key.

2. Add the ASP.NET Web Helpers Library to your website as described in Installing Helpers in an ASP.NET Web Pages Site, if you haven't already.

3. If you don't already have a *_AppStart.cshtml* file, in the root folder of a website create a file named *_AppStart.cshtml*.

4. Add the following `Recaptcha` helper settings in the *_AppStart.cshtml* file:

```
@using Microsoft.Web.Helpers;
@{
  // Add the PublicKey and PrivateKey strings with your public
  // and private keys. Obtain your PublicKey and PrivateKey
  // at the ReCaptcha.Net (http://recaptcha.net) website.
  ReCaptcha.PublicKey = "your-public-key";
  ReCaptcha.PrivateKey = "your-private-key";
}
```

5. Set the `PublicKey` and `PrivateKey` properties using your own public and private keys.

6. Save the *_AppStart.cshtml* file and close it.

7. In the root folder of a website, create new page named *Recaptcha.cshtml*.

8. Replace the existing content with the following:

```
@using Microsoft.Web.Helpers;
@{
  var showRecaptcha = true;
  if (IsPost) {
    if (ReCaptcha.Validate()) {
        @:Your response passed!
        showRecaptcha = false;
    }
    else{
      @:Your response didn't pass!
    }
  }
}
<!DOCTYPE html>
<html>
    <head>
        <title>Testing Global Recaptcha Keys</title>
    </head>
    <body>
    <form action="" method="post">
    @if(showRecaptcha == true){
        if(ReCaptcha.PrivateKey != ""){
            <p>@ReCaptcha.GetHtml()</p>
            <input type="submit" value="Submit" />
        }
        else {
            <p>You can get your public and private keys at
            the ReCaptcha.Net website (http://recaptcha.net).
            Then add the keys to the _AppStart.cshtml file.</p>
        }
    }
    </form>
    </body>
</html>
```

9. Run the *Recaptcha.cshtml* page in a browser. If the `PrivateKey` value is valid, the page displays the ReCaptcha control and a button. If you had not set the keys globally in *_AppStart.html*, the page would display an error.

10. Enter the words for the test. If you pass the ReCaptcha test, you see a message to that effect. Otherwise you see an error message and the ReCaptcha control is redisplayed.

NOTE

If your computer is on a domain that uses proxy server, you might need to configure the `defaultproxy` element of the *Web.config* file. The following example shows a *Web.config* file with the `defaultproxy` element configured to enable the ReCaptcha service to work.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.net>
      <defaultProxy>
         <proxy
            usesystemdefault = "false"
            proxyaddress="http://myProxy.MyDomain.com"
            bypassonlocal="true"
            autoDetect="False"
         />
      </defaultProxy>
   </system.net>
</configuration>
```

Additional Resources

- Customizing Site-Wide Behavior for ASP.NET Web Pages Sites
- ReCaptcha site

Performance

2/20/2020 • 2 minutes to read • Edit Online

How to cache content for faster performance and how to measure site traffic.

- Caching to Improve the Performance of Your Website
- Analyzing Traffic
- Bundling and Minifying Assets in an ASP.NET Web Pages (Razor) Site

Caching Data in an ASP.NET Web Pages (Razor) Site for Better Performance

2/20/2020 • 5 minutes to read • Edit Online

by Tom FitzMacken

This article explains how to use a helper to cache information for faster performance in an ASP.NET Web Pages (Razor) website. You can speed up your website by having it store — that is, cache — the results of data that ordinarily would take considerable time to retrieve or process and that does not change often.

What you'll learn:

- How to use caching to improve the responsiveness of your website.

These are the ASP.NET features introduced in the article:

- The `WebCache` helper.

Software versions used in the tutorial

- ASP.NET Web Pages (Razor) 3

This tutorial also works with ASP.NET Web Pages 2.

Every time someone requests a page from your site, the web server has to do some work in order to fulfill the request. For some of your pages, the server might have to perform tasks that take a (comparatively) long time, such as retrieving data from a database. Even if these tasks don't take long in absolute terms, if your site experiences a lot of traffic, a whole series of individual requests that cause the web server to perform the complicated or slow task can add up to a lot of work. This can ultimately affect the performance of the site.

One way to improve the performance of your website in circumstances like this is to cache data. If your site gets repeated requests for the same information, and the information does not need to be modified for each person, and it's not time sensitive, instead of re-fetching or recalculating it, you can fetch the data once and then store the results. The next time a request comes in for that information, you just get it out of the cache.

In general, you cache information that doesn't change frequently. When you put information in the cache, it's stored in memory on the web server. You can specify how long it should be cached, from seconds to days. When the caching period expires, the information is automatically removed from the cache.

> **NOTE**
>
> Entries in the cache might be removed for reasons other than that they've expired. For example, the web server might temporarily run low on memory, and one way it can reclaim memory is by throwing entries out of the cache. As you'll see, even if you've put information into the cache, you have to check to be sure it's still there when you need it.

Imagine your website has a page that displays the current temperature and weather forecast. To get this type of information, you might send a request to an external service. Since this information doesn't change much (within a two-hour time period, for example) and since external calls require time and bandwidth, it's a good candidate for caching.

Adding Caching to a Page

ASP.NET includes a `WebCache` helper that makes it easy to add caching to your site and add data to the cache. In this procedure, you'll create a page that caches the current time. This isn't a real-world example, since the current time is something that does change often, and that moreover isn't complex to calculate. However, it's a good way to illustrate caching in action.

1. Add a new page named *WebCache.cshtml* to the website.

2. Add the following code and markup to the page:

```
@{
    var cacheItemKey = "CachedTime";
    var cacheHit = true;
    var time = WebCache.Get(cacheItemKey);

    if (time == null) {
        cacheHit = false;
    }

    if (cacheHit == false) {
        time = @DateTime.Now;
        WebCache.Set(cacheItemKey, time, 1, false);
    }
}
<!DOCTYPE html>
<html>
<head>
    <title>WebCache Helper Sample</title>
</head>
<body>
    <div>
        @if (cacheHit) {
            @:Found the time data in the cache.
        } else {
            @:Did not find the time data in the cache.
        }
    </div>
    <div>
        This page was cached at @time.
    </div>
</body>
</html>
```

When you cache data, you put it into the cache using a name this is unique across the website. In this case, you'll use a cache entry named `CachedTime`. This is the `cacheItemKey` shown in the code example.

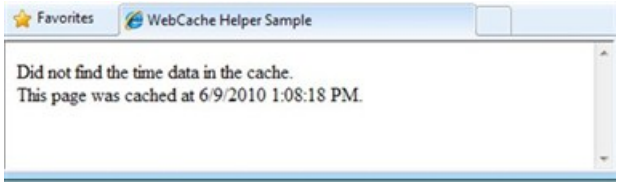
The code first reads the `CachedTime` cache entry. If a value is returned (that is, if the cache entry isn't null), the code just sets the value of the time variable to the cache data.

However, if the cache entry doesn't exist (that is, it's null), the code sets the time value, adds it to the cache, and sets an expiration value to one minute. After one minute, the cache entry is discarded. (The default expiration value for an item in the cache is 20 minutes.) The command `WebCache.Set(cacheItemKey, time, 1, false)` shows how to add the current time value to the cache and set its expiration to 1 minute. Setting the *slidingExpiration* parameter to `false` means the expiration time is not renewed each time it is requested. It will expire exactly 1 minute after it was originally added to the cache. If you set this value to `true` the 1 minute expiration time is reset each time the value is requested from the cache.

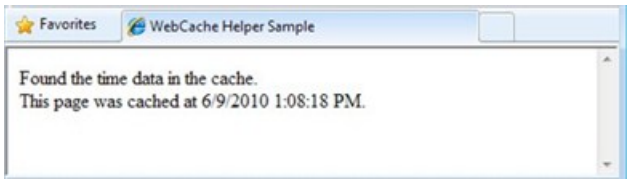
This code illustrates the pattern you should always use when you cache data. Before you get something out of the cache, always check first whether the `WebCache.Get` method has returned null. Remember that the

cache entry might have expired or might have been removed for some other reason, so any given entry is never guaranteed to be in the cache.

3. Run *WebCache.cshtml* in a browser. (Make sure the page is selected in the **Files** workspace before you run it.) The first time you request the page, the time data isn't in the cache, and the code has to add the time value to the cache.



4. Refresh *WebCache.cshtml* in the browser. This time, the time data is in the cache. Notice that the time hasn't changed since the last time you viewed the page.



5. Wait one minute for the cache to be emptied, and then refresh the page. The page again indicates that the time data wasn't found in the cache, and the updated time is added to the cache.

Additional Resources

- Displaying Data in a Chart
- WebCache API reference (MSDN)

Tracking Visitor Information (Analytics) for an ASP.NET Web Pages (Razor) Site

2/20/2020 • 4 minutes to read • Edit Online

by Tom FitzMacken

This article describes how to use a helper to add website analytics to pages in an ASP.NET Web Pages (Razor) website.

What you'll learn:

- How to send information about your website traffic to an analytics provider.

These are the ASP.NET programming features introduced in the article:

- The `Analytics` helper.

Software versions used in the tutorial

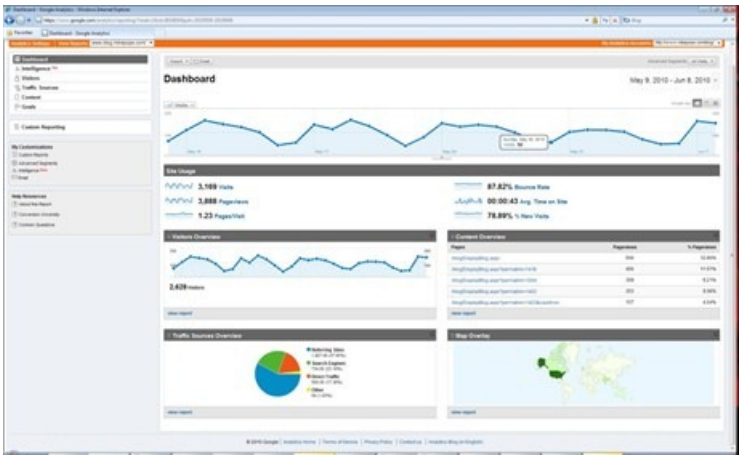
- ASP.NET Web Pages (Razor) 2
- ASP.NET Web Helpers Library (NuGet package)

Analytics is a general term for technology that measures traffic on your website so you can understand how people use the site. Many analytics services are available, including services from Google, Yahoo, StatCounter, and others.

The way analytics works is that you sign up for an account with the analytics provider, where you register the site that you want to track. The provider sends you a snippet of JavaScript code that includes an ID or tracking code for your account. You add the JavaScript snippet to the web pages on the site that you want to track. (You typically add the analytics snippet to a footer or layout page or other HTML markup that appears on every page in your site.) When users request a page that contains one of these JavaScript snippets, the snippet sends information about the current page to the analytics provider, who records various details about the page.

When you want to have a look at your site statistics, you log into the analytics provider's website. You can then view all sorts of reports about your site, like:

- The number of page views for individual pages. This tells you (roughly) how many people are visiting the site, and which pages on your site are the most popular.
- How long people spend on specific pages. This can tell you things like whether your home page is keeping people's interest.
- What sites people were on before they visited your site. This helps you understand whether your traffic is coming from links, from searches, and so on.
- When people visit your site and how long they stay.
- What countries your visitors are from.
- What browsers and operating systems your visitors are using.



Using a Helper to Add Analytics to a Page

ASP.NET Web Pages includes several analytics helpers (`Analytics.GetGoogleHtml` , `Analytics.GetYahooHtml` , and `Analytics.GetStatCounterHtml`) that make it easy to manage the JavaScript snippets used for analytics. Instead of figuring out how and where to put the JavaScript code, all you have to do is add the helper to a page. The only information you need to provide is your account name, ID, or tracking code. (For StatCounter, you also have to provide a few additional values.)

In this procedure, you'll create a layout page that uses the `GetGoogleHtml` helper. If you already have an account with one of the other analytics providers, you can use that account instead and make slight adjustments as needed.

> **NOTE**
> When you create an analytics account, you register the URL of the site that you want to be tracking. If you're testing everything on your local computer, you won't be tracking actual traffic (the only traffic is you), so you won't be able to record and view site statistics. But this procedure shows how you add an analytics helper to a page. When you publish your site, the live site will send information to your analytics provider.

1. Add the ASP.NET Web Helpers Library to your website as described in Installing Helpers in an ASP.NET Web Pages Site, if you haven't already added it.

2. Create an account with Google Analytics and record the account name.

3. Create a layout page named *Analytics.cshtml* and add the following markup:

```
@using Microsoft.Web.Helpers;
<!DOCTYPE html>
<html>
  <head>
    <title>Analytics Test</title>
  </head>
  <body>
    <h1>Analytics Test Page</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
    sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. </p>
    <div id="footer">
      &copy; 2012 MySite
    </div>
    @Analytics.GetGoogleHtml("myaccount")
  </body>
</html>
```

> **NOTE**
> You must place the call to the `Analytics` helper in the body of your web page (before the `</body>` tag). Otherwise, the browser will not run the script.

If you're using a different analytics provider, use one of the following helpers instead:

- (Yahoo) `@Analytics.GetYahooHtml("myaccount")`
- (StatCounter) `@Analytics.GetStatCounterHtml("project", "security")`

4. Replace `myaccount` with the name of the account, ID, or tracking code that you created in step 1.

5. Run the page in the browser. (Make sure the page is selected in the **Files** workspace before you run it.)

6. In the browser, view the page source. You'll be able to see the rendered analytics code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Analytics Test</title>
  </head>
  <body>
    <h1>Analytics Test Page</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
    sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
    <div id="footer">
      &copy; 2012 MySite
    </div>
    <script type="text/javascript">
      var gaJsHost = (("https:" == document.location.protocol) ? "https://ssl." : "http://www.");
      document.write(unescape("%3Cscript src='" + gaJsHost + "google-analytics.com/ga.js'
type='text/javascript'%3E%3C/script%3E"));
    </script>
    <script type="text/javascript">
      try{
      var pageTracker = _gat._getTracker("myaccount");
      pageTracker._trackPageview();
      } catch(err) {}
    </script>
  </body>
</html>
```

7. Log onto the Google Analytics site and examine the statistics for your site. If you're running the page on a live site, you see an entry that logs the visit to your page.

Additional Resources

- Google Analytics site
- Yahoo! Web Analytics site
- StatCounter site

Bundling and Minifying Assets in an ASP.NET Web Pages (Razor) Site

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Bundling and minification are ways to make your site faster. Bundling lets you combine multiple JavaScript (*.js*) files or multiple cascading style sheet (*.css*) files so that they can be downloaded as a unit, rather than one at a time. Minification squeezes out whitespace and performs other types of compression to make the downloaded files as small a possible.

> **NOTE**
>
> The RC release of ASP.NET Web Pages 2 does not support bundling and minification because the package that contains the required elements is not yet available in Microsoft WebMatrix. We apologize for this inconvenience. The package is expected to be available in the final release of ASP.NET Web Pages 2 and WebMatrix 2.

API Reference

2/20/2020 • 2 minutes to read • Edit Online

API Reference documentation for ASP.NET Web Pages.

- ASP.NET API Quick Reference

ASP.NET Web Pages (Razor) API Quick Reference

2/20/2020 • 13 minutes to read • Edit Online

by Tom FitzMacken

> This page contains a list with brief examples of the most commonly used objects, properties, and methods for programming ASP.NET Web Pages with Razor syntax.
>
> Descriptions marked with "(v2)" were introduced in ASP.NET Web Pages version 2.
>
> For API reference documentation, see the ASP.NET Web Pages Reference Documentation on MSDN.
>
> ## Software versions
>
> - ASP.NET Web Pages (Razor) 3
>
> This tutorial also works with ASP.NET Web Pages 2 and ASP.NET Web Pages 1.0 (except for features marked v2).

This page contains reference information for the following:

- Classes
- Data
- Helpers
- Validation

Classes

`AppState[key], AppState[index],App`

Contains data that can be shared by any pages in the application. You can use the dynamic `App` property to access the same data, as in the following example:

```
AppState["FavoriteColor"] = "red";
AppState[1] = "apples";
App.MyGreeting = "Good morning";
// Displays the value assigned to AppData[1] in the page.
@App[1]
// Displays the value assigned to App.MyGreeting.
@App.MyGreeting
```

`AsBool(), AsBool(true|false)`

Converts a string value to a Boolean value (true/false). Returns false or the specified value if the string does not represent true/false.

```
bool b = stringValue.AsBool();
```

`AsDateTime(), AsDateTime(value)`

Converts a string value to date/time. Returns `DateTime.MinValue` or the specified value if the string does not represent a date/time.

```
DateTime dt = stringValue.AsDateTime();
```

`AsDecimal(), AsDecimal(value)`

Converts a string value to a decimal value. Returns 0.0 or the specified value if the string does not represent a decimal value.

```
decimal d = stringValue.AsDecimal();
```

`AsFloat(), AsFloat(value)`

Converts a string value to a float. Returns 0.0 or the specified value if the string does not represent a decimal value.

```
float d = stringValue.AsFloat();
```

`AsInt(), AsInt(value)`

Converts a string value to an integer. Returns 0 or the specified value if the string does not represent an integer.

```
int i = stringValue.AsInt();
```

`Href(path [, param1 [, param2]])`

Creates a browser-compatible URL from a local file path, with optional additional path parts.

```
<a href="@Href("~/Folder/File")">Link to My File</a>
<a href="@Href("~/Product", "Tea")">Link to Product</a>
```

`Html.Raw(value)`

Renders *value* as HTML markup instead of rendering it as HTML-encoded output.

```
@* Inserts markup into the page. *@
@Html.Raw("<div>Hello <em>world</em>!</div>")
```

`IsBool(), IsDateTime(), IsDecimal(), IsFloat(), IsInt()`

Returns true if the value can be converted from a string to the specified type.

```
var isint = stringValue.IsInt();
```

`IsEmpty()`

Returns true if the object or variable has no value.

```
if (Request["companyname"].IsEmpty()) {
    @:Company name is required.<br />
}
```

`IsPost`

Returns true if the request is a POST. (Initial requests are usually a GET.)

```
if (IsPost) { Response.Redirect("Posted"); }
```

`Layout`

Specifies the path of a layout page to apply to this page.

```
Layout = "_MyLayout.cshtml";
```

`PageData[key], PageData[index],Page`

Contains data shared between the page, layout pages, and partial pages in the current request. You can use the dynamic `Page` property to access the same data, as in the following example:

```
PageData["FavoriteColor"] = "red";
PageData[1] = "apples";
Page.MyGreeting = "Good morning";
// Displays the value assigned to PageData[1] in the page.
@Page[1]
// Displays the value assigned to Page.MyGreeting.
@Page.MyGreeting
```

`RenderBody()`

(Layout pages) Renders the content of a content page that is not in any named sections.

```
@RenderBody()
```

`RenderPage(path, values)`

`RenderPage(path[,param1 [, param2]])`

Renders a content page using the specified path and optional extra data. You can get the values of the extra parameters from `PageData` by position (example 1) or key (example 2).

```
RenderPage("_MySubPage.cshtml", "red", 123, "apples")
RenderPage("_MySubPage.cshtml", new { color = "red", number = 123, food = "apples" })
```

`RenderSection(sectionName [, required = true|false])`

(Layout pages) Renders a content section that has a name. Set *required* to false to make a section optional.

```
@RenderSection("header")
```

`Request.Cookies[key]`

Gets or sets the value of an HTTP cookie.

```
var cookieValue = Request.Cookies["myCookie"].Value;
```

`Request.Files[key]`

Gets the files that were uploaded in the current request.

```
Request.Files["postedFile"].SaveAs(@"MyPostedFile");
```

`Request.Form[key]`

Gets data that was posted in a form (as strings). `Request[key]` checks both the `Request.Form` and the `Request.QueryString` collections.

```
var formValue = Request.Form["myTextBox"];
// This call produces the same result.
var formValue = Request["myTextBox"];
```

`Request.QueryString[key]`

Gets data that was specified in the URL query string. `Request[key]` checks both the `Request.Form` and the `Request.QueryString` collections.

```
var queryValue = Request.QueryString["myTextBox"];
// This call produces the same result.
var queryValue = Request["myTextBox"];
```

`Request.Unvalidated(key)`

`Request.Unvalidated().QueryString|Form|Cookies|Headers[key]`

Selectively disables request validation for a form element, query-string value, cookie, or header value. Request validation is enabled by default and prevents users from posting markup or other potentially dangerous content.

```
// Call the method directly to disable validation on the specified item from
// one of the Request collections.
Request.Unvalidated("userText");

// You can optionally specify which collection the value is from.
var prodID = Request.Unvalidated().QueryString["productID"];
var richtextValue = Request.Unvalidated().Form["richTextBox1"];
var cookie = Request.Unvalidated().Cookies["mostRecentVisit"];
```

`Response.AddHeader(name, value)`

Adds an HTTP server header to the response.

```
// Adds a header that requests client browsers to use basic authentication.
Response.AddHeader("WWW-Authenticate", "BASIC");
```

`Response.OutputCache(seconds [, sliding] [, varyByParams])`

Caches the page output for a specified time. Optionally set *sliding* to reset the timeout on each page access and *varyByParams* to cache different versions of the page for each different query string in the page request.

```
Response.OutputCache(60);
Response.OutputCache(3600, true);
Response.OutputCache(10, varyByParams : new[] {"category","sortOrder"});
```

`Response.Redirect(path)`

Redirects the browser request to a new location.

```
Response.Redirect("~/Folder/File");
```

`Response.SetStatus(httpStatusCode)`

Sets the HTTP status code sent to the browser.

```
Response.SetStatus(HttpStatusCode.Unauthorized);
Response.SetStatus(401);
```

`Response.WriteBinary(data [, mimetype])`

Writes the contents of *data* to the response with an optional MIME type.

```
Response.WriteBinary(image, "image/jpeg");
```

`Response.WriteFile(file)`

Writes the contents of a file to the response.

```
Response.WriteFile("file.ext");
```

`@section(sectionName) {content }`

(Layout pages) Defines a content section that has a name.

```
@section header { <div>Header text</div> }
```

`Server.HtmlDecode(htmlText)`

Decodes a string that is HTML encoded.

```
var htmlDecoded = Server.HtmlDecode("&lt;html&gt;");
```

`Server.HtmlEncode(text)`

Encodes a string for rendering in HTML markup.

```
var htmlEncoded = Server.HtmlEncode("<html>");
```

`Server.MapPath(virtualPath)`

Returns the server physical path for the specified virtual path.

```
var dataFile = Server.MapPath("~/App_Data/data.txt");
```

`Server.UrlDecode(urlText)`

Decodes text from a URL.

```
var urlDecoded = Server.UrlDecode("url%20data");
```

`Server.UrlEncode(text)`

Encodes text to put in a URL.

```
var urlEncoded = Server.UrlEncode("url data");
```

`Session[key]`

Gets or sets a value that exists until the user closes the browser.

```
Session["FavoriteColor"] = "red";
```

`ToString()`

Displays a string representation of the object's value.

```
<p>It is now @DateTime.Now.ToString()</p>
```

`UrlData[index]`

Gets additional data from the URL (for example, */MyPage/ExtraData*).

```
var pathInfo = UrlData[0];
```

`WebSecurity.ChangePassword(userName,currentPassword,newPassword)`

Changes the password for the specified user.

```
var success = WebSecurity.ChangePassword("my-username",
    "current-password", "new-password");
```

`WebSecurity.ConfirmAccount(accountConfirmationToken)`

Confirms an account using the account confirmation token.

```
var confirmationToken = Request.QueryString["ConfirmationToken"];
if(WebSecurity.ConfirmAccount(confirmationToken)) {
      //...
}
```

`WebSecurity.CreateAccount(userName, password`

`[, requireConfirmationToken = true|false])`

Creates a new user account with the specified user name and password. To require a confirmation token, pass true for *requireConfirmationToken.*

```
WebSecurity.CreateAccount("my-username", "secretpassword");
```

`WebSecurity.CurrentUserId`

Gets the integer identifier for the currently logged-in user.

```
var userId = WebSecurity.CurrentUserId;
```

`WebSecurity.CurrentUserName`

Gets the name for the currently logged-in user.

```
var welcome = "Hello " + WebSecurity.CurrentUserName;
```

`WebSecurity.GeneratePasswordResetToken(username`

`[, tokenExpirationInMinutesFromNow])`

Generates a password-reset token that can be sent in email to a user so that the user can reset the password.

```
var resetToken = WebSecurity.GeneratePasswordResetToken("my-username");
var message = "Visit http://example.com/reset-password/" + resetToken +
    " to reset your password";
WebMail.Send(..., message);
```

`WebSecurity.GetUserId(userName)`

Returns the user ID from the user name.

```
var userId = WebSecurity.GetUserId(userName);
```

`WebSecurity.IsAuthenticated`

Returns true if the current user is logged in.

```
if(WebSecurity.IsAuthenticated) {...}
```

`WebSecurity.IsConfirmed(userName)`

Returns true if the user has been confirmed (for example, through a confirmation email).

```
if(WebSecurity.IsConfirmed("joe@contoso.com")) { ... }
```

`WebSecurity.IsCurrentUser(userName)`

Returns true if the current user's name matches the specified user name.

```
if(WebSecurity.IsCurrentUser("joe@contoso.com")) { ... }
```

`WebSecurity.Login(userName,password[, persistCookie])`

Logs the user in by setting an authentication token in the cookie.

```
if(WebSecurity.Login("username", "password")) { ... }
```

`WebSecurity.Logout()`

Logs the user out by removing the authentication token cookie.

```
WebSecurity.Logout();
```

`WebSecurity.RequireAuthenticatedUser()`

If the user is not authenticated, sets the HTTP status to 401 (Unauthorized).

```
WebSecurity.RequireAuthenticatedUser();
```

`WebSecurity.RequireRoles(roles)`

If the current user is not a member of one of the specified roles, sets the HTTP status to 401 (Unauthorized).

```
WebSecurity.RequireRoles("Admin", "Power Users");
```

`WebSecurity.RequireUser(userId)`

`WebSecurity.RequireUser(userName)`

If the current user is not the user specified by *username*, sets the HTTP status to 401 (Unauthorized).

```
WebSecurity.RequireUser("joe@contoso.com");
```

`WebSecurity.ResetPassword(passwordResetToken,newPassword)`

If the password reset token is valid, changes the user's password to the new password.

```
WebSecurity.ResetPassword( "A0F36BFD9313", "new-password")
```

Data

`Database.Execute(SQLstatement [,parameters]`

Executes *SQLstatement* (with optional parameters) such as INSERT, DELETE, or UPDATE and returns a count of affected records.

```
db.Execute("INSERT INTO Data (Name) VALUES ('Smith')");

db.Execute("INSERT INTO Data (Name) VALUES (@0)", "Smith");
```

`Database.GetLastInsertId()`

Returns the identity column from the most recently inserted row.

```
db.Execute("INSERT INTO Data (Name) VALUES ('Smith')");
var id = db.GetLastInsertId();
```

`Database.Open(filename)`

`Database.Open(connectionStringName)`

Opens either the specified database file or the database specified using a named connection string from the *Web.config* file.

```
// Note that no filename extension is specified.
var db = Database.Open("SmallBakery"); // Opens SmallBakery.sdf in App_Data
// Opens a database by using a named connection string.
var db = Database.Open("SmallBakeryConnectionString");
```

`Database.OpenConnectionString(connectionString)`

Opens a database using the connection string. (This contrasts with `Database.Open`, which uses a connection string name.)

```
var db = Database.OpenConnectionString("Data Source=|DataDirectory|\SmallBakery.sdf");
```

`Database.Query(SQLstatement[,parameters])`

Queries the database using *SQLstatement* (optionally passing parameters) and returns the results as a collection.

```
foreach (var result in db.Query("SELECT * FROM PRODUCT")) {<p>@result.Name</p>}

foreach (var result = db.Query("SELECT * FROM PRODUCT WHERE Price > @0", 20))
   { <p>@result.Name</p> }
```

`Database.QuerySingle(SQLstatement [, parameters])`

Executes *SQLstatement* (with optional parameters) and returns a single record.

```
var product = db.QuerySingle("SELECT * FROM Product WHERE Id = 1");

var product = db.QuerySingle("SELECT * FROM Product WHERE Id = @0", 1);
```

`Database.QueryValue(SQLstatement [, parameters])`

Executes *SQLstatement* (with optional parameters) and returns a single value.

```
var count = db.QueryValue("SELECT COUNT(*) FROM Product");

var count = db.QueryValue("SELECT COUNT(*) FROM Product WHERE Price > @0", 20);
```

Helpers

`Analytics.GetGoogleHtml(webPropertyId)`

Renders the Google Analytics JavaScript code for the specified ID.

```
@Analytics.GetGoogleHtml("MyWebPropertyId")
```

`Analytics.GetStatCounterHtml(project,security)`

Renders the StatCounter Analytics JavaScript code for the specified project.

```
@Analytics.GetStatCounterHtml(89, "security")
```

`Analytics.GetYahooHtml(account)`

Renders the Yahoo Analytics JavaScript code for the specified account.

```
@Analytics.GetYahooHtml("myaccount")
```

`Bing.SearchBox([boxWidth])`

Passes a search to Bing. To specify the site to search and a title for the search box, you can set the `Bing.SiteUrl` and `Bing.SiteTitle` properties. Normally you set these properties in the *_AppStart* page.

```
@Bing.SearchBox() @* Searches the web.*@
```

```
@{
   Bing.SiteUrl = "www.asp.net";  @* Limits search to the www.asp.net site. *@
}
@Bing.SearchBox()
```

`Chart(width,height [, template] [, templatePath])`

Initializes a chart.

```
@{
   var myChart = new Chart(width: 600, height: 400);
}
```

`Chart.AddLegend([title] [, name])`

Adds a legend to a chart.

```
@{
var myChart = new Chart(width: 600, height: 400)
    .AddLegend("Basic Chart")
    .AddSeries(
        name: "Employee",
        xValue: new[] {  "Peter", "Andrew", "Julie", "Mary", "Dave" },
        yValues: new[] { "2", "6", "4", "5", "3" })
    .Write();
}
```

`Chart.AddSeries([name] [, chartType] [, chartArea]`

`[, axisLabel] [, legend] [, markerStep] [, xValue]`
`[, xField] [, yValues] [, yFields] [, options])`

Adds a series of values to the chart.

```
@{
var myChart = new Chart(width: 600, height: 400)
    .AddSeries(
        name: "Employee",
        xValue: new[] {  "Peter", "Andrew", "Julie", "Mary", "Dave" },
        yValues: new[] { "2", "6", "4", "5", "3" })
    .Write();
}
```

`Crypto.Hash(string [, algorithm])`

`Crypto.Hash(bytes [, algorithm])`

Returns a hash for the specified data. The default algorithm is `sha256`.

```
@Crypto.Hash("data")
```

`Facebook.LikeButton(href [, buttonLayout] [, showFaces] [, width] [, height]`

`[, action] [, font] [, colorScheme] [, refLabel])`

Lets Facebook users make a connection to pages.

```
@Facebook.LikeButton("www.asp.net")
```

`FileUpload.GetHtml([initialNumberOfFiles] [, allowMoreFilesToBeAdded]`

`[, includeFormTag] [, addText] [, uploadText])`

Renders UI for uploading files.

```
@FileUpload.GetHtml(initialNumberOfFiles:1, allowMoreFilesToBeAdded:false,
  includeFormTag:true, uploadText:"Upload")
```

`GamerCard.GetHtml(gamerTag)`

Renders the specified Xbox gamer tag.

```
@GamerCard.GetHtml("joe")
```

`Gravatar.GetHtml(email [, imageSize] [, defaultImage] [, rating]`

`[, imageExtension] [, attributes])`

Renders the Gravatar image for the specified email address.

```
@Gravatar.GetHtml("joe@contoso.com")
```

`Json.Encode(object)`

Converts a data object to a string in the JavaScript Object Notation (JSON) format.

```
var myJsonString = Json.Encode(dataObject);
```

`Json.Decode(string)`

Converts a JSON-encoded input string to a data object that you can iterate over or insert into a database.

```
var myJsonObj = Json.Decode(jsonString);
```

`LinkShare.GetHtml(pageTitle[, pageLinkBack] [, twitterUserName]`

`[, additionalTweetText] [, linkSites])`

Renders social networking links using the specified title and optional URL.

```
@LinkShare.GetHtml("ASP.NET Web Pages Samples")
@LinkShare.GetHtml("ASP.NET Web Pages Samples", "http://www.asp.net")
```

`ModelStateDictionary.AddError(key, errorMessage)`

Associates an error message with a form field. Use the `ModelState` helper to access this member.

```
ModelState.AddError("email", "Enter an email address");
```

`ModelStateDictionary.AddFormError(errorMessage)`

Associates an error message with a form. Use the `ModelState` helper to access this member.

```
ModelState.AddFormError("Password and confirmation password do not match.");
```

`ModelStateDictionary.IsValid`

Returns true if there are no validation errors. Use the `ModelState` helper to access this member.

```
if (ModelState.IsValid) { // Save the form to the database }
```

`ObjectInfo.Print(value [, depth] [, enumerationLength])`

Renders the properties and values of an object and any child objects.

```
@ObjectInfo.Print(person)
```

`Recaptcha.GetHtml([, publicKey] [, theme] [, language] [, tabIndex])`

Renders the reCAPTCHA verification test.

```
@ReCaptcha.GetHtml()
```

`ReCaptcha.PublicKey`

`ReCaptcha.PrivateKey`

Sets public and private keys for the reCAPTCHA service. Normally you set these properties in the *_AppStart* page.

```
ReCaptcha.PublicKey = "your-public-recaptcha-key";
ReCaptcha.PrivateKey = "your-private-recaptcha-key";
```

`ReCaptcha.Validate([, privateKey])`

Returns the result of the reCAPTCHA test.

```
if (ReCaptcha.Validate()) {
   // Test passed.
}
```

`ServerInfo.GetHtml()`

Renders status information about ASP.NET Web Pages.

```
@ServerInfo.GetHtml()
```

`Twitter.Profile(twitterUserName)`

Renders a Twitter stream for the specified user.

```
@Twitter.Profile("billgates")
```

`Twitter.Search(searchQuery)`

Renders a Twitter stream for the specified search text.

```
@Twitter.Search("asp.net")
```

`Video.Flash(filename [, width, height])`

Renders a Flash video player for the specified file with optional width and height.

```
@Video.Flash("test.swf", "100", "100")
```

`Video.MediaPlayer(filename [, width, height])`

Renders a Windows Media player for the specified file with optional width and height.

```
@Video.MediaPlayer("test.wmv", "100", "100")
```

`Video.Silverlight(filename, width, height)`

Renders a Silverlight player for the specified *.xap* file with required width and height.

```
@Video.Silverlight("test.xap", "100", "100")
```

`WebCache.Get(key)`

Returns the object specified by *key*, or null if the object is not found.

```
var username = WebCache.Get("username")
```

`WebCache.Remove(key)`

Removes the object specified by *key* from the cache.

```
WebCache.Remove("username")
```

`WebCache.Set(key, value [, minutesToCache] [, slidingExpiration])`

Puts *value* into the cache under the name specified by *key*.

```
WebCache.Set("username", "joe@contoso.com ")
```

`WebGrid(data)`

Creates a new `WebGrid` object using data from a query.

```
var db = Database.Open("SmallBakery");
var grid = new WebGrid(db.Query("SELECT * FROM Product"));
```

`WebGrid.GetHtml()`

Renders markup to display data in an HTML table.

```
@grid.GetHtml()// The 'grid' variable is set when WebGrid is created.
```

`WebGrid.Pager()`

Renders a pager for the `WebGrid` object.

```
@grid.Pager() // The 'grid' variable is set when WebGrid is created.
```

`WebImage(path)`

Loads an image from the specified path.

```
var image = new WebImage("test.png");
```

`WebImage.AddImagesWatermark(image)`

Adds the specified image as a watermark.

```
WebImage photo = new WebImage("test.png");
WebImage watermarkImage = new WebImage("logo.png");
photo.AddImageWatermark(watermarkImage);
```

`WebImage.AddTextWatermark(text)`

Adds the specified text to the image.

```
image.AddTextWatermark("Copyright")
```

`WebImage.FlipHorizontal()`

`WebImage.FlipVertical()`

Flips the image horizontally or vertically.

```
image.FlipHorizontal();
image.FlipVertical();
```

`WebImage.GetImageFromRequest()`

Loads an image when an image is posted to a page during a file upload.

```
var image = WebImage.GetImageFromRequest();
```

`WebImage.Resize(width,height)`

Resizes an the image.

```
image.Resize(100, 100);
```

`WebImage.RotateLeft()`

`WebImage.RotateRight()`

Rotates the image to the left or the right.

```
image.RotateLeft();
image.RotateRight();
```

`WebImage.Save(path [, imageFormat])`

Saves the image to the specified path.

```
image.Save("test.png");
```

`WebMail.Password`

Sets the password for the SMTP server. Normally you set this property in the *_AppStart* page.

```
WebMail.Password = "password";
```

`WebMail.Send(to, subject, body [, from] [, cc] [, filesToAttach] [, isBodyHtml]`

`[, additionalHeaders])`

Sends an email message.

```
WebMail.Send("touser@contoso.com", "subject", "body of message",
    "fromuser@contoso.com");
```

`WebMail.SmtpServer`

Sets the SMTP server name. Normally you set this property in the *_AppStart* page.

```
WebMail.SmtpServer = "smtp.mailserver.com";
```

`WebMail.UserName`

Sets the user name for the SMTP server. Normally you should set this property in the *_AppStart* page.

```
WebMail.UserName = "Joe";
```

Validation

`Html.ValidationMessage(field)`

(v2) Renders a validation error message for the specified field.

```
<input type="text"
    name="dateOfBirth"
    value="" />
@Html.ValidationMessage("dateOfBirth")
```

`Html.ValidationSummary([message])`

(v2) Displays a list of all validation errors.

```
@Html.ValidationSummary()

@Html.ValidationSummary("The following validation errors occurred:")
```

`Validation.Add(field, validationType)`

(v2) Registers a user input element for the specified type of validation.

```
Validation.Add("dateOfBirth", Validator.DateTime("Date of birth was not valid"));
Validation.Add("quantity", Validator.Integer("Enter a number"));
Validation.Add("confirmPassword",
    Validator.EqualsTo("password", "Passwords must match."));
```

`Validation.ClassFor(field)`

(v2) Dynamically renders CSS class attributes for client-side validation so that you can format validation error messages. (Requires that you reference the appropriate client-script libraries and that you define CSS classes.)

```
<head>
  <script
    src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.2.js">
  </script>
  <script
    src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8.1/jquery.validate.js">
  </script>
  <script
    src="~/Scripts/jquery.validate.unobtrusive.js">
  </script>

  <style>
    input-validation-error{ /* style rules  */ }
    field-validation-error{ /* style rules  */ }
    validation-summary-errors{ /* style rules  */ }
    field-validation-valid{ /* style rules  */ }
    input-validation-valid{ /* style rules  */ }
    validation-summary-valid{ /* style rules  */ }
  </style>
</head>

...

<input
    type="text"
    name="firstName"
    @Validation.For("firstName") @Validation.ClassFor("firstName")  />
```

`Validation.For(field)`

(v2) Enables client-side validation for the user input field. (Requires that you reference the appropriate client-script libraries.)

```
<head>
  <script
    src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.2.js">
  </script>
  <script
    src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8.1/jquery.validate.js">
  </script>
  <script
    src="~/Scripts/jquery.validate.unobtrusive.js">
  </script>
</head>

...

<input
    type="text"
    name="firstName"
    @Validation.For("firstName") />
```

`Validation.IsValid()`

(v2) Returns true if all user input elements that are registred for validation contain valid values.

```
if(IsPost){
    if (Validation.IsValid()) {
        // Process input
    }
}
```

`Validation.RequireField(field[, errorMessage])`

(v2) Specifies that users must provide a value for the user input element.

```
Validation.RequireField("dateOfBirth", "Date of birth is required");
```

`Validation.RequireFields(field1[, field12, field3, ...])`

(v2) Specifies that users must provide values for each of the user input elements. This method does not let you specify a custom error message.

```
Validation.RequireFields("firstName", "lastName", "dateOfBirth");
```

`Validator.DateTime ([error message])`

`Validator.Decimal([error message])`
`Validator.EqualsTo(otherField,[error message])`
`Validator.Float([error message])`
`Validator.Integer([error message])`
`Validator.Range(min,max [, error message])`
`Validator.RegEx(pattern[, error message])`
`Validator.Required([error message])`
`Validator.StringLength(length)`
`Validator.Url([error message])`

(v2) Specifies a validation test when you use the `Validation.Add` method.

```
Validation.Add("dateOfBirth", Validator.DateTime("Date of birth was not valid"));
Validation.Add("quantity", Validator.Integer("Enter a number"));
Validation.Add("confirmPassword", Validator.EqualsTo("password",
    "Passwords must match."));
```

Web Pages Releases

2/20/2020 • 2 minutes to read • Edit Online

What's new in each ASP.NET Web Pages release.

- What's New in ASP.NET Web Pages 3.2
- What's New in ASP.NET Web Pages 3.1
- Top Features in ASP.NET Web Pages 2
- ASP.NET Web Pages 2 Developer Preview ReadMe
- Running Different Versions of ASP.NET Web Pages Side by Side

What's New in ASP.NET Web Pages 3.2

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

This topic describes what's new for ASP.NET Web Pages 3.2, Web Pages 3.2.2 and Web Pages 3.2.3 beta1

ASP.NET Web Pages 3.2

This release fixes a bug and introduces one new feature.

Download

The runtime features are released as NuGet packages on the NuGet gallery. All the runtime packages follow the Semantic Versioning specification. The ASP.NET Web Pages 3.2 package has the following version: "3.2.0". You can install or update these packages through NuGet. The release also includes corresponding localized packages on NuGet.

You can install or update to the released NuGet packages by using the NuGet Package Manager Console:

```
Install-Package Microsoft.AspNet.WebPages -Version 3.2.0
```

New Feature and Bug Fix

We fixed one bug and made one minor feature enhancement in this release. You can find the query for the same here.

ASP.NET Web Pages 3.2.2

This release rolls-up the change in the ASP.NET Web Pages 3.2.1 Beta release which provides a significant performance improvement in rendering large razor pages. SeeCodeplex Issue 585. This release aligns with the MVC 5.2.2 packages which will now depend on this version.

We worked with the MSN team on rendering large pages. When pages render over 80 Kilobytes of data, we end up with objects on the large object heap. When multiple layers of layouts are used this effect can be multiplied.

The result on the server is extra CPU usage, longer retention of memory, and even long pauses during Gen 2 cleanup in the garbage collector.

Below is a table demonstrating the results of analyzing a perfview for a run. The CPU is held constant at about 68%, while large pages are being rendered. The table shows that the number of Generation 2 collections has been almost completely eliminated, and the result is higher request rate and a considerable reduction in pauses due to garbage collection.

AREA	BEFORE (3.2)	AFTER (3.2.1)	DELTA %
Total request (count)	26,986	32,591	20.80%
Trace duration (seconds)	196.20	198.60	1.20%

AREA	BEFORE (3.2)	AFTER (3.2.1)	DELTA %
Request/second	137.53	164.10	19.30%
CPU Load	68.80%	68.50%	-0.40%
GC CPU Samples	124,323	17,543	-85.90%
Total allocations (count)	55,357,146	57,222,949	3.40%
Total GC Pause (samples)	15,091	8,515	-43.60%
Gen0 GC (count)	403	1,216	201.70%
Gen1 GC (count)	290	367	26.60%
Gen2 GC (count)	229	2	-99.10%
CPU / request (samples/req)	19.73	16.47	-16.50%

COLOR CODING:	CORE IMPROVEMENT	POSITIVE IMPACT ON PERFORMANCE

ASP.NET Web Pages 3.2.3 beta1

This release contains only bug fixes. You can use this query to see the list of issues fixed in this release.

What's New in ASP.NET Web Pages 3.1

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

This topic describes what's new for ASP.NET Web Pages 3.1.

This release fixes bugs and does not introduce any new features.

Download

The runtime features are released as NuGet packages on the NuGet gallery. All the runtime packages follow the Semantic Versioning specification. The latest ASP.NET Web Pages 3.1 RTM package has the following version: "3.1.2". You can install or update these packages through NuGet. The release also includes corresponding localized packages on NuGet.

You can install or update to the released NuGet packages by using the NuGet Package Manager Console:

```
Install-Package Microsoft.AspNet.WebPages -Version 3.1.2
```

The Top Features in ASP.NET Web Pages 2

4/10/2020 • 32 minutes to read • Edit Online

by Microsoft

This article provides an overview of the top new features in the ASP.NET Web Pages 2 RC, a lightweight web programming framework that is included with Microsoft WebMatrix 2 RC.

What's included:

- Installing WebMatrix
- New and enhanced features
  - Changes for the RC release
  - Changes for the Beta release
  - Using the New and Updated Site Templates
  - Validating User Input
  - Enabling logins from Facebook and other sites using OAuth and OpenID
  - Adding Maps using the Maps Helper
  - Running Web Pages Applications Side by Side
  - Rendering Pages for Mobile Devices
- Additional Resources

> **NOTE**
>
> This topic assumes that you are using WebMatrix to work with your ASP.NET Web Pages 2 code. However, as with Web Pages 1, you can also create Web Pages 2 websites using Visual Studio, which gives you enhanced IntelliSense capabilities and debugging. To work with Web Pages in Visual Studio, you must first install Visual Studio 2010 SP1, Visual Web Developer Express 2010 SP1, or Visual Studio 11 Beta. Then install the ASP.NET MVC 4 Beta, which includes templates and tools for creating ASP.NET MVC 4 and Web Pages 2 applications in Visual Studio.

Last update: 18 June 2012

Installing WebMatrix

To install Web Pages, you can use the Microsoft Web Platform Installer, which is a free application that makes it easy to install and configure web-related technologies. You will install the WebMatrix 2 Beta, which includes Web Pages 2 Beta.

1. Browse to the installation page for the latest version of the Web Platform Installer:

   https://go.microsoft.com/fwlink/?LinkId=226883

   > **NOTE**
   >
   > If you already have WebMatrix 1 installed, this installation updates it to WebMatrix 2 Beta. You can run websites that were created using version 1 or 2 on the same computer. For more information, see the section on Running Web Pages Applications Side by Side.

2. Choose **Install Now**.

   If you use Internet Explorer, go to the next step. If you use a different browser like Mozilla Firefox or Google Chrome, you are prompted to save the *Webmatrix.exe* file to your computer. Save the file and then click it to launch the installer.

3. Run the installer and choose the **Install** button. This installs WebMatrix and Web Pages.

New and Enhanced Features

Changes for the RC Version (June 2012)

The RC version release in June 2012 has a few changes from the Beta version refresh that was released in March 2012. These changes are:

- A `Validation.AddFormError` method was added to the `Validation` helper. This is useful if you perform validation manually (for example, you validate a value that is passed in the query string) and you want to add an error message that can be displayed by the `Html.ValidationSummary` method. For more information, see the section Validating Data That Doesn't Come Directly From Users in Validating User Input in ASP.NET Web Pages (Razor) Sites.
- The functionality for bundling and minification has been removed from the core ASP.NET Web Pages 2 assemblies. As a consequence, the `Assets` helper listed later in this document is not available. Instead, you must install the ASP.NET Optimization NuGet package. For more information, see Bundling and Minifying Assets in an ASP.NET Web Pages (Razor) Site.
- Additional assemblies to support ASP.NET Web Pages 2 have been added. The only noticeable effect of this change is that you might see more assemblies in a site's *bin* folder after you create a site or deploy a site to a hosting provider.

Changes for the Beta Version (February 2012)

The Beta version released in February 2012 has only a few changes from the Beta version that was released in December 2011. These changes are:

- Razor now supports conditional attributes. In an HTML element, if you set an attribute to a value that resolves in server code to `false` or `null`, ASP.NET does not render the attribute at all. For example, imagine you have the following markup for a check box:

  ```
  <input type="checkbox"
  name="check1"
  value="check1"
  checked="@checked1" />
  ```

  If the value of `checked1` resolves to `false` or to `null`, the `checked` attribute is not rendered. This is a breaking change.

- The `Validation.GetHtml` method has been renamed to `Validation.For`. This is a breaking change; `Validation.GetHtml` will not work in the Beta release.

- You can now include the `~` operator in markup to reference the site root without using the `Href` function. (That is, the Razor parser can now find and resolve the `~` operator without requiring an explicit method call to `Href`.) The `Href` method still works, so this is not a breaking change.

  For example, if you previously had markup like this:

  ```
  <a href="@Href("~/Default.cshtml")">Home</a>
  ```

  You can now use markup like this:

```
<a href="~/Default.cshtml">Home</a>
```

- The `Scripts` helper for assets (resource) management has been replaced with the `Assets` helper, which has slightly different methods, such as the following:
  - For `Scripts.Add`, use `Assets.AddScript`
  - For `Scripts.GetScriptTags`, use `Assets.GetScripts`

    This is a breaking change; the `Scripts` class is not available in the Beta release. The code examples in this document that use asset management have been updated with this change.

Using the New and Updated Site Templates

The **Starter Site** template has been updated so that it runs on Web Pages 2 by default. It also includes the following new capabilities:

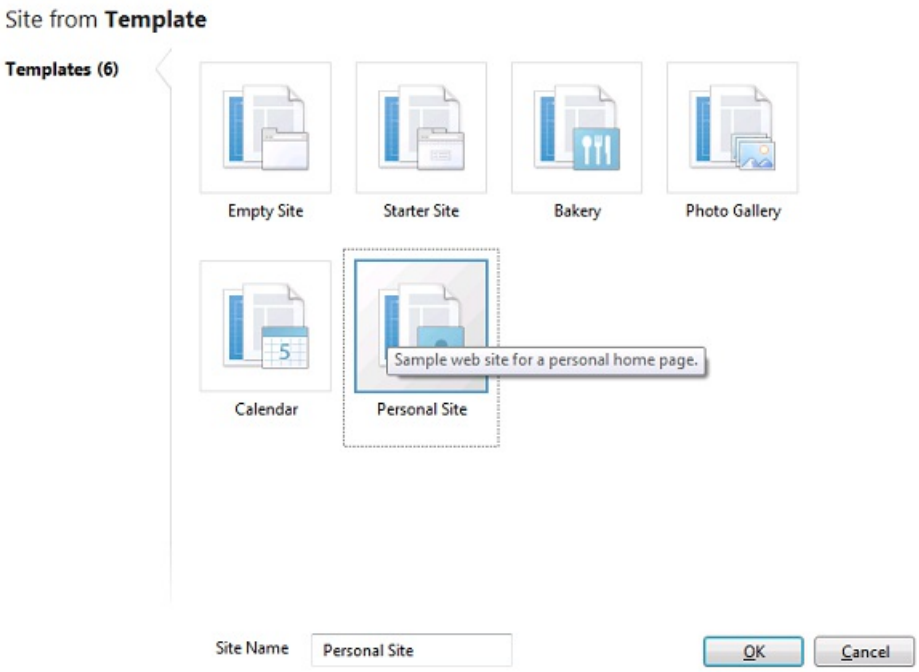
- Mobile-friendly page rendering. Through the use of CSS styles and the `@media` selector, the **Starter Site** provides improved rendering of pages on smaller screens, including mobile device screens.
- Improved membership and authentication options. You can let users log into your site using their accounts from other social networking sites, such as Twitter, Facebook, and Windows Live. For more information, see the Enabling Logins from Facebook and Other Sites using OAuth and OpenID section.
- HTML5 elements.

The new **Personal Site** template lets you create a website that contains a personal blog, a photos page, and a Twitter page. You can customize a site based on the **Personal Site** template by doing the following:

- Change the look of the site by editing the layout file (*_SiteLayout.cshtml*) and the styles file (*Site.css*).
- Install NuGet packages that add functionality to your site. For information about how to install packages, including the ASP.NET Web Helpers Library, see the tutorial about installing helpers.

To access the **Personal Site** template, choose **Templates** on the WebMatrix **Quick Start** screen.

In the **Templates** dialog box, choose the **Personal Site** template.



The landing page of the **Personal Site** template lets you follow links to set up your blog, Twitter page, and photos page.

Validating User Input

In Web Pages 1, to validate user input on submitted forms, you use the `System.Web.WebPages.Html.ModelState` class. (This is illustrated in several of the code samples in the Web Pages 1 tutorial titled Working with Data.) You can still use this approach in Web Pages 2. However, Web Pages 2 also offers improved tools for validating user input:

- New validation classes, including `System.Web.WebPages.ValidationHelper` and `System.Web.WebPages.Validator`, that let you do powerful validation tasks with a few lines of code.
- Optionally, client-side validation, which provides immediate feedback to the user instead of requiring a round trip to the server to check for validation errors. (For security reasons, validation is performed on the server even if the checks have been performed in the client beforehand.)

To use the new validation features, do the following:

In the page's code, register an element to be validated by using methods of the `Validation` helper: `Validation.RequireField`, `Validation.RequireFields` (to register multiple elements to be required), or `Validation.Add`. The `Add` method lets you specify other types of validation checks, like data-type checking, comparing entries in different fields, string-length checks, and patterns (using regular expressions). Here are some examples:

```
Validation.RequireField("text1");
Validation.RequireField("text1", "The text1 field is required");
Validation.RequireFields("text1", "text2", "text3");

Validation.Add("text1", Validation.StringLength(5));
Validation.Add("textDate", Validation.DateTime("Enter a date"));
Validation.Add("textCount", Validation.Integer("Enter a number"));
Validation.Add("textCount",
Validation.Range(1, 10, "Enter a value between 1 and 10"));
```

To display a field-specific error, call `Html.ValidationMessage` in the markup for each element being validated:

```
<input type="text" name="course"
value="@Request["course"]" />
@Html.ValidationMessage("course")
```

To display a summary (`<ul>` list) of all the errors in the page, `Html.ValidationSummary` in the markup:

```
@Html.ValidationSummary()
```

These steps are enough to implement server-side validation. If you want to add client-side validation, do the following in addition.

Add the following script file references inside the `<head>` section of a web page. The first two script references point to remote files on a content delivery network (CDN) server. The third reference points to a local script file. Production apps should implement a fallback when the CDN is unavailable. Test the fallback.

```
<script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.2.js"></script>
<script src="http://ajax.aspnetcdn.com/ajax/jquery.validate/1.8.1/jquery.validate.min.js"></script>
<script src="~/Scripts/jquery.validate.unobtrusive.min.js"></script>
```

The easiest way to get a local copy of the *jquery.validate.unobtrusive.min.js* library is to create a new Web Pages site based on one of the site templates (such as Starter Site). The site created by the template includes *jquery.validate.unobtrusive.js* file in its Scripts folder, from which you can copy it to your site.

If your website uses a *_SiteLayout* page to control the page layout, you can include these script references in that page so that validation is available to all content pages. If you want to perform validation only on particular pages, you can use the assets manager to register the scripts on only those pages. To do this, call `Assets.AddScript(path)` in the page that you want to validate and reference each of the script files. Then add a call to `Assets.GetScripts` in the *_SiteLayout* page in order to render the registered `<script>` tags. For more information, see the section Registering Scripts with the Assets Manager.

In the markup for an individual element, call the `Validation.For` method. This method emits attributes that jQuery can hook in order to provide client-side validation. For example:

```
<input type="text" name="course"
value="@Request["course"]"
@Validation.For("course")
/>
@Html.ValidationMessage("course")
```

The following example shows a page that validates user input on a form. To run and test this validation code, do this:

1. Create a new web site using one of the WebMatrix 2 site templates that includes a *Scripts* folder, such as the **Starter Site** template.
2. In the new site, create a new *.cshtml* page, and replace the contents of the page with the following code.
3. Run the page in a browser. Enter valid and invalid values to see the effects on validation. For example, leave a required field blank or enter a letter in the **Credits** field.

```
@{
// Specify what fields users must fill in.
Validation.RequireFields("course", "professorname", "credits");
// Add validation criteria.  Here, require that input to Credits is an integer.
Validation.Add("credits", Validator.Integer());

if (IsPost)  {
// Wrap the postback code with a validation check.
if (Validation.IsValid()) {
    var course = Request["course"];
    var professorName = Request["professorname"];
    var credits = Request["credits"];
    // Display valid input values.
    <text>
        You entered: <br />
        Class: @course <br />
        Professor: @professorName <br />
        Credits: @credits <br />
    </text>
}
}
}
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<title>Testing Validation in ASP.NET Web Pages version 2</title>
<script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.2.js"></script>
<script  src="http://ajax.aspnetcdn.com/ajax/jquery.validate/1.8.1/jquery.validate.min.js"></script>
<script src="@Href("~/Scripts/jquery.validate.unobtrusive.min.js")"></script>
</head>
<body>
<form method="post" action="">
<!-- Display a summary message about any validation issues. -->
@Html.ValidationSummary()
<fieldset>
  <legend>Add Class</legend>
<div>
  <label for="Course">Class:</label>
  <!-- Validation.For("course") adds validation attributes to the input element. -->
  <input type="text" name="Course" value="@Request["course"]"  @Validation.For("course") />

  <!-- Display a field-specific message about validation issues. -->
  @Html.ValidationMessage("course")
</div>

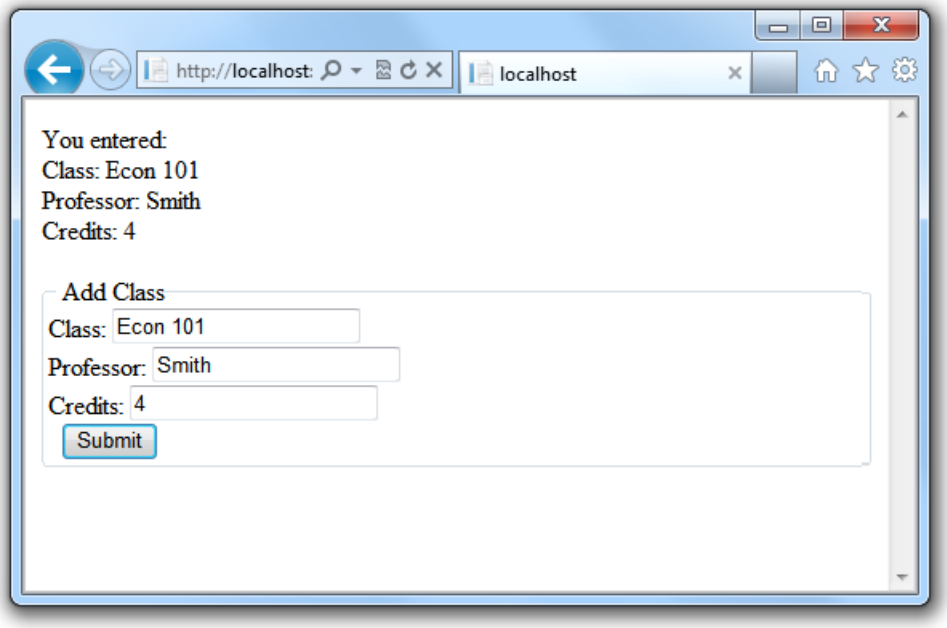
<div>
  <label for="ProfessorName">Professor:</label>
  <input type="text" name="ProfessorName" value="@Request["professorname"]"
```

```
      @Validation.For("professorname") />
  @Html.ValidationMessage("professorname")
</div>

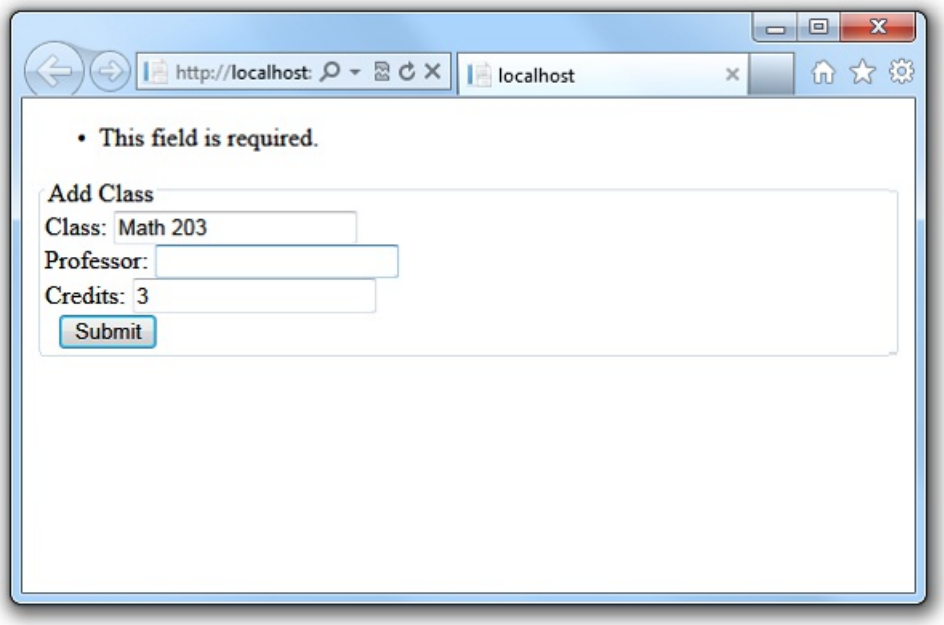
<div>
  <label for="Credits">Credits:</label>
  <input type="text" name="Credits" value="@Request["credits"]" @Validation.For("credits") />
  @Html.ValidationMessage("credits")
</div>

<div>
  <label> </label>
  <input type="submit" value="Submit" class="submit" />
</div>
  </fieldset>
</form>
</body>
</html>
```

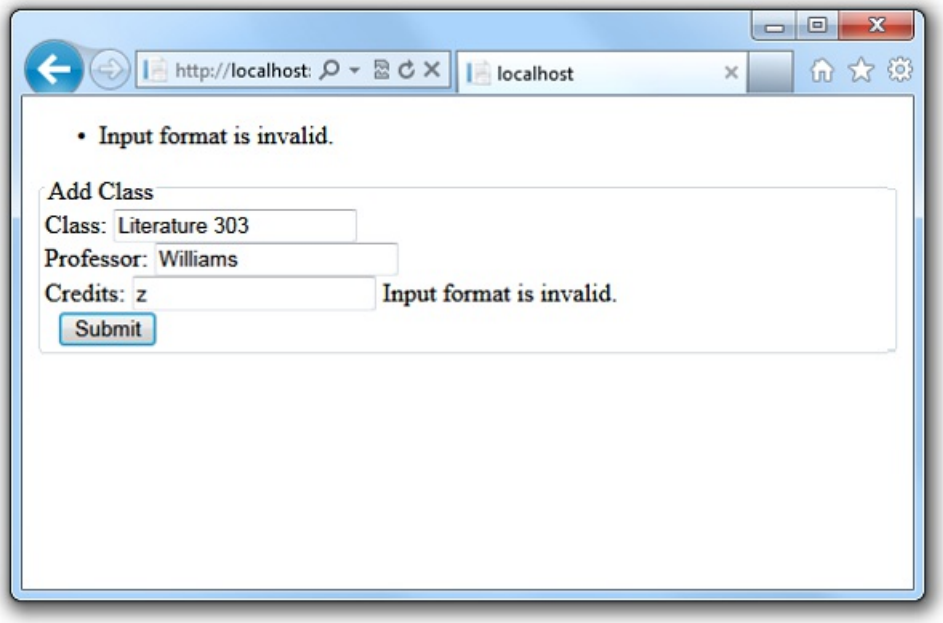
Here is the page when a user submits valid input:



Here is the page when a user submits it with a required field left empty:



Here is the page when a user submits it with something other than an integer in the **Credits** field:



For more information, see the following blog posts:

- Updated validation in Web Pages v2 Basics of adding validation using the `Validation` helper (server-side only)
- Updated validation in Web Pages v2, Part 2 Adding client-side validation.
- Updated validation in Web Pages v2, Part 3 Formatting validation errors.

Registering Scripts Using the Assets Manager

The assets manager is a new feature that you can use in server code to register and render client scripts. This feature is helpful when you are working with code from multiple files (such as layout pages, content pages, helpers, etc.) that are combined into a single page at run time. The assets manager coordinates the source files to make sure that script files are referenced correctly and efficiently on the rendered page, regardless of which code files they are called from or how many times they are called. The assets manager also renders `<script>` tags in the right place so that the page can load quickly (without downloading scripts while rendering) and to avoid errors that can occur if scripts are called before rendering is complete.

For example, suppose that you create a custom helper that calls a JavaScript file, and you call this helper at three different places in your content page code. If you don't use the assets manager to register the script calls in the helper, three different `<script>` tags that all point to the same script file will appear in your rendered page. Plus, depending on where the `<script>` tags are inserted in the rendered page, errors may occur if the script tries to access certain page elements before the page fully loads. If you use the assets manager to register the script, you avoid these problems.

You can register a script with the assets manager by doing this:

- In the code that needs to reference the script, call the `Assets.AddScript` method.
- In a *_SiteLayout* page, call the `Assets.GetScripts` method to render the `<script>` tags.

> **NOTE**
> Put calls to `Assets.GetScripts` as the very last item in the `<body>` element of the *_SiteLayout* page. This helps the page load faster and can help avoid script errors.

The following example shows how the assets manager works. The code contains the following items:

- A custom helper named `MakeNote`. This helper renders a string inside a box by wrapping a `div` element

around it that's styled with a border and by adding "Note:" to it. The helper also calls a JavaScript file that adds run-time behavior to the note. Rather than reference the script with a `<script>` tag, the helper registers the script by calling `Assets.AddScript` .

- A JavaScript file. This is the file that's called by the helper, and it temporarily increases the font size of note items during a `mouseover` event.
- A content page, which references the *_SiteLayout* page, renders some content in the body, and then calls the `MakeNote` helper.
- A *_SiteLayout* page. This page provides a common header and a page layout structure. It also includes a call to `Assets.GetScripts`, which is how the assets manager renders script calls in a page.

To run the sample:

1. Create an empty Web Pages 2 website. You can use the WebMatrix **Empty Site** template for this.
2. Create a folder named *Scripts* in the site.
3. In the *Scripts* folder, create a file named *Test.js*, copy the *Test.js* content into it from the example, and save the file..
4. Create a folder named *App_Code* in the site.
5. In the *App_Code* folder, create a file named *Helpers.cshtml*, copy the example code into it, and save it in a folder named *App_Code* in the root folder.
6. In the site's root folder, create a file named *_SiteLayout.cshtml,* copy the example into it, and save the file.
7. In the site root, create a file named *ContentPage.cshtml*, add the example code, and save it.
8. Run *ContentPage* in a browser. The string you passed to the `MakeNote` helper is rendered as a boxed note.
9. Pass the mouse pointer over the note. The script temporarily increases the font size of the note.
10. View the source of the rendered page. Because of where you placed the call to `Assets.GetScripts`, the rendered `<script>` tag that calls *Test.js* is the very last item in the body of the page.

Test.js

```
function UpdateNoteStyle(id) {
var theNote = document.getElementById(id);
theNote.style.fontSize = "150%";
}
function ReturnNoteStyle(id) {
var theNote = document.getElementById(id);
theNote.style.fontSize = "inherit";
}
```

Helpers.cshtml

```
@helper MakeNote(string content, string noteid) {
Assets.AddScript("~/Scripts/Test.js");

<div id="@noteid" style="border: 1px solid black; width: 90%; padding: 5px; margin-left: 15px;"
 onmouseover="UpdateNoteStyle('@noteid')" onmouseout="ReturnNoteStyle('@noteid')">
  <p>
  <strong>Note</strong>   @content
   </p>
</div>
}
```

_SiteLayout.cshtml

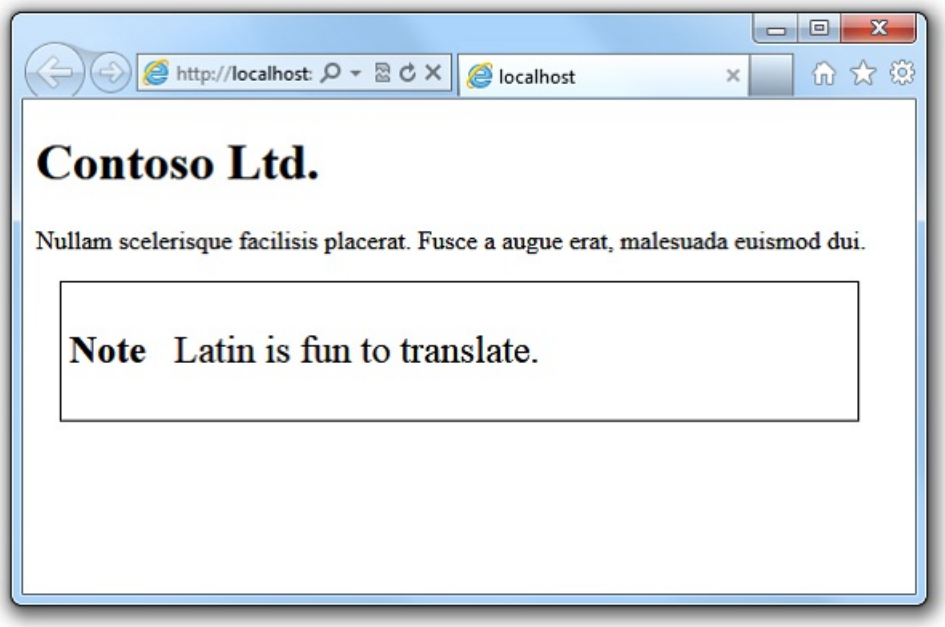
```
<!DOCTYPE html>

<html lang="en">
<head>
<meta charset="utf-8" />
<title></title>
</head>
<body>
<header>
    <div class="content-wrapper">
        <h1>Contoso Ltd.</h1>
    </div>
</header>
<div id="body">
    <section class="content-wrapper main-content clear-fix">
        @RenderBody()
    </section>
</div>
@Assets.GetScripts()
</body>
</html>
```

ContentPage.cshtml

```
@{
Layout = "~/_SiteLayout.cshtml";
}
<p>
Nullam scelerisque facilisis placerat. Fusce a augue
erat, malesuada euismod dui.
</p>
@Helpers.MakeNote("Latin is fun to translate.", "note1")
```

The following screenshot shows *ContentPage.cshtml* in a browser when you hold the mouse pointer over the note:

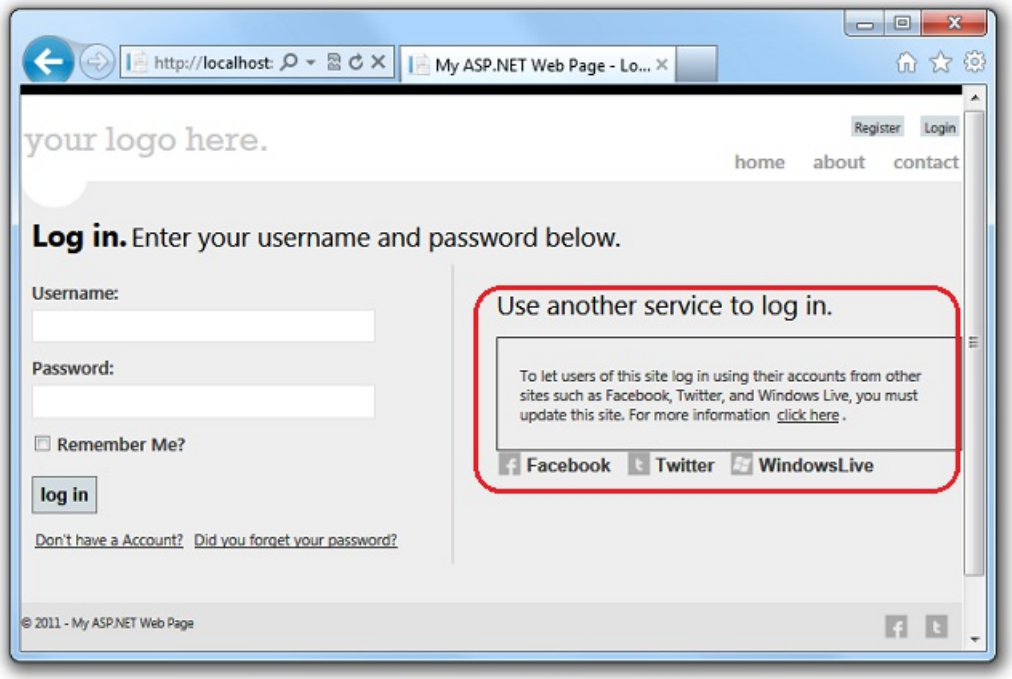


Enabling Logins from Facebook and Other Sites Using OAuth and OpenID

Web Pages 2 provides enhanced options for membership and authentication. The main enhancement is that there are new OAuth and OpenID providers. Using these providers, you can let users log into your site using their existing credentials from Facebook, Twitter, Windows Live, Google, and Yahoo. For example, to log in using a Facebook account, users can just choose a Facebook icon, which redirects them to the Facebook login page where they enter their user information. They can then associate the Facebook login with their account on your site. A

related enhancement to the Web Pages membership features is that users can associate multiple logins (including logins from social networking sites) with a single account on your website.

This image shows the Login page from the **Starter Site** template, where a user can choose a Facebook, Twitter, or Windows Live icon to enable logging in with an external account:



You can enable OAuth and OpenID membership using just a few lines of code. The methods and properties you use to work with the OAuth and OpenID providers are in the `WebMatrix.Security.OAuthWebSecurity` class.

However, instead of using code to enable logins from other sites, a recommended way to get started with the new providers is to use the new **Starter Site** template that is included with WebMatrix 2 Beta. The **Starter Site** template includes a full membership infrastructure, complete with a login page, a membership database, and all the code you need to let users log into your site using either local credentials or those from another site.

How to Enable Logins using the OAuth and OpenID Providers

This section provides an example of how to let users log in from external sites (Facebook, Twitter, Windows Live, Google, or Yahoo) to a site that's based on the **Starter Site** template. After creating a starter site, you do this (details follow):

- For the sites that use an OAuth provider (Facebook, Twitter, and Windows Live), create an application on the external site. This gives you application keys that you'll need in order to invoke the login feature for those sites. For sites that use an OpenID provider (Google, Yahoo), you do not have to create an application. For all of these sites, you must have an account in order to log in and to create developer applications.

  > **NOTE**
  > Windows Live applications only accept a live URL for a working website, so you cannot use a local website URL for testing logins.

- Edit a few files in your website in order to specify the appropriate authentication provider and to submit a login to the site you want to use.

To enable Google and Yahoo logins:

1. In your website, edit the *_AppStart.cshtml* page and add the following two lines of code in the Razor code block after the call to the `WebSecurity.InitializeDatabaseConnection` method. This code enables both the Google and Yahoo OpenID providers.

```
OAuthWebSecurity.RegisterOpenIDClient(BuiltInOpenIDClient.Google);
OAuthWebSecurity.RegisterOpenIDClient(BuiltInOpenIDClient.Yahoo);
```

2. In the *~/Account/Login.cshtml* page, remove the comments from the following `<fieldset>` block of markup near the end of the page. To uncomment the code, remove the `@*` characters that precede and follow the `<fieldset>` block. The resulting code block looks like this:

```
<fieldset>
<legend>Log in using another service</legend>
<input type="submit" name="provider" id="facebook" value="Facebook" title="Log in using your Facebook
account." />
<input type="submit" name="provider" id="twitter" value="Twitter" title="Log in using your Twitter
account." />
<input type="submit" name="provider" id="windowsLive" value="WindowsLive" title="Log in using your
Windows Live account." />
</fieldset>
```

3. Add an `<input>` element for the Google or Yahoo provider to the `<fieldset>` group in the *~/Account/Login.cshtml* page. The updated `<fieldset>` group with `<input>` elements for both Google and Yahoo looks like the following example:

```
<fieldset>
<legend>Log in using another service</legend>
<input type="submit" name="provider" id="facebook" value="Facebook" title="Log in using your Facebook
account." />
<input type="submit" name="provider" id="twitter" value="Twitter" title="Log in using your Twitter
account." />
<input type="submit" name="provider" id="windowsLive" value="WindowsLive" title="Log in using your
Windows Live account." />
<input type="submit" name="provider" id="yahoo" value="Yahoo" title="Log in using your Yahoo account."
/>
<input type="submit" name="provider" id="google" value="Google" title="Log in using your Google
account." />
</fieldset>
```

4. In the *~/Account/AssociateServiceAccount.cshtml* page, add `<input>` elements for Google or Yahoo to the `<fieldset>` group near the end of the file. You can copy the same `<input>` elements that you just added to the `<fieldset>` section in the *~/Account/Login.cshtml* page.

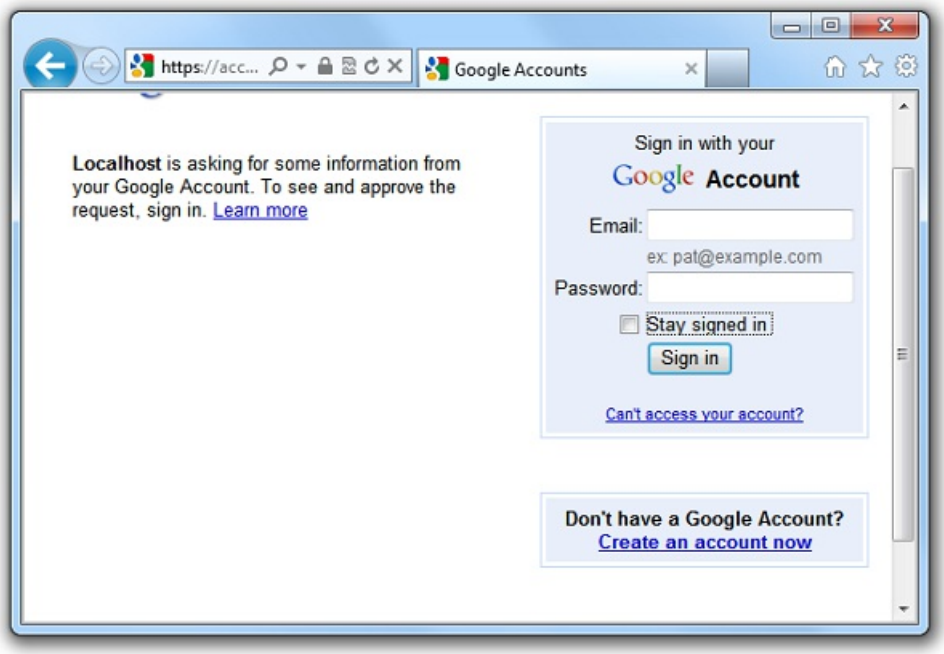
   The *~/Account/AssociateServiceAccount.cshtml* page in the Starter Site template can be used if you want to create a page on which users can associate multiple logins from other sites with a single account on your website.

Now you can test Google and Yahoo logins.

1. Run the *default.cshtml* page of your site and choose the **Log in** button.

2. On the *Login* page, in the **Use another service to log in** section, choose either the **Google** or **Yahoo** submit button. This example uses the Google login.

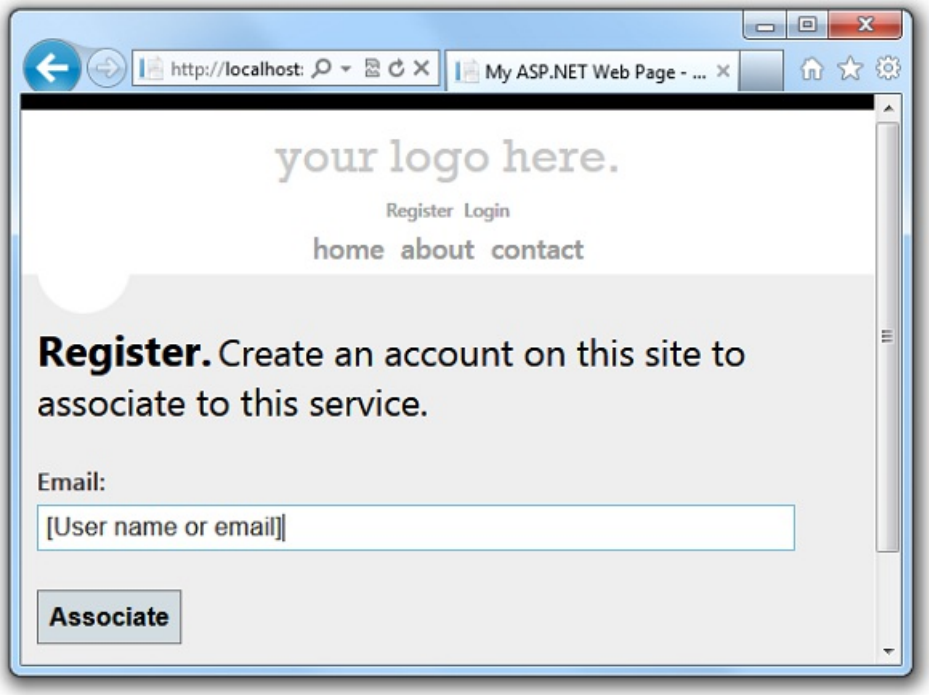
   The web page redirects the request to the Google login page.



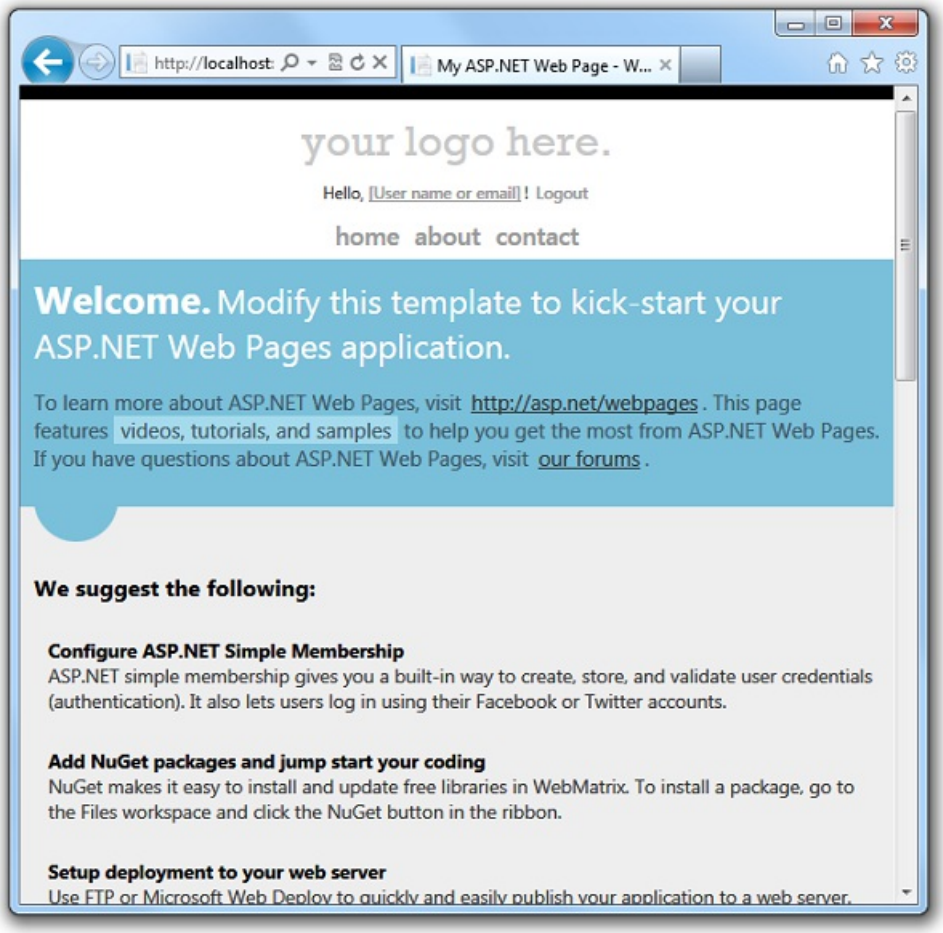
3. Enter credentials for an existing Google account.

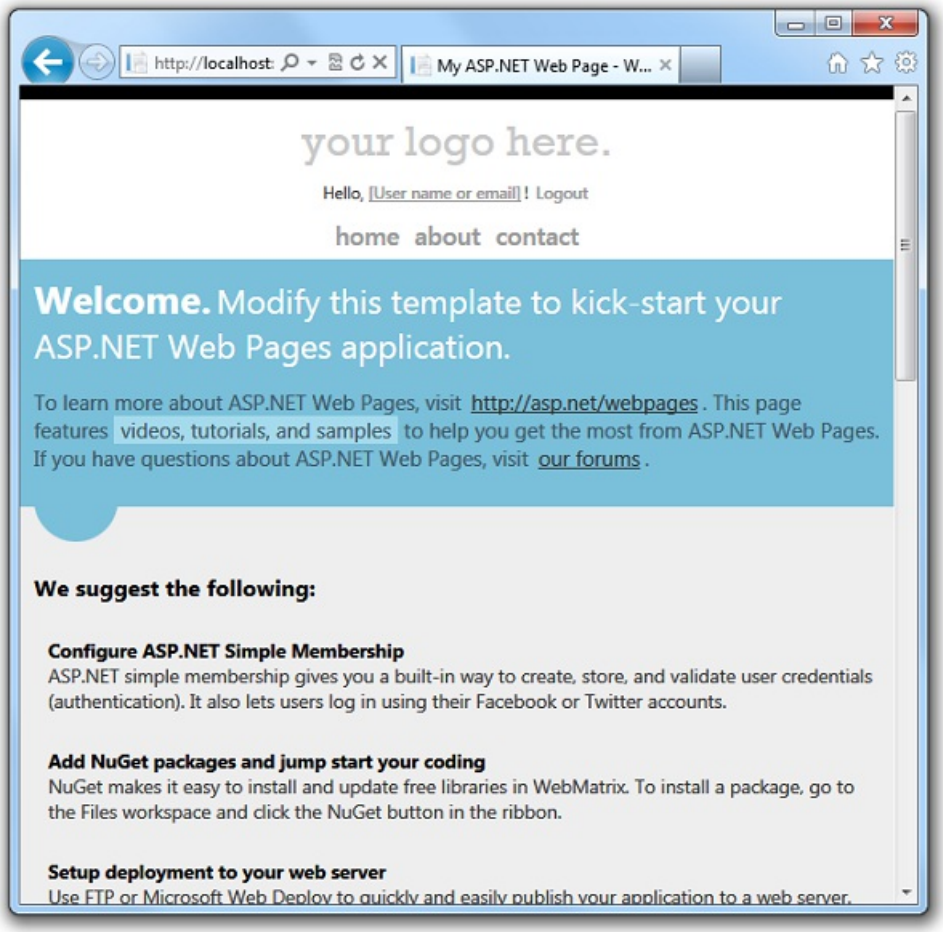
4. If Google asks you whether you want to allow Localhost to use information from the account, click **Allow**.

   The code uses the Google token to authenticate the user, and then returns to this page on your website. This page lets users associate their Google login with an existing account on your website, or they can register a new account on your site to associate the external login with.



5. Choose the **Associate** button. The browser returns to your application's home page.





To enable Facebook logins:

1. Go to the Facebook developers site (log in if you're not already logged in).

2. Choose the **Create New App** button, and then follow the prompts to name and create the new application.

3. In the section **Select how your app will integrate with Facebook**, choose the **Website** section.

4. Fill in the **Site URL** field with the URL of your site (for example, `http://www.example.com`). The **Domain** field is optional; you can use this to provide authentication for an entire domain (such as *example.com*).

> **NOTE**
> If you are running a site on your local computer with a URL like `http://localhost:12345` (where the number is a local port number), you can add this value to the **Site URL** field for testing your site. However, any time the port number of your local site changes, you will need to update the **Site URL** field of your application.

5. Choose the **Save Changes** button.

6. Choose the **Apps** tab again, and then view the start page for your application.

7. Copy the **App ID** and **App Secret** values for your application and paste them into a temporary text file. You will pass these values to the Facebook provider in your website code.

8. Exit the Facebook developer site.

Now you make changes to two pages in your website so that users will able to log into the site using their Facebook accounts.

1. In your website, edit the *_AppStart.cshtml* page and uncomment the code for the Facebook OAuth provider. The uncommented code block looks like the following:

```
OAuthWebSecurity.RegisterOAuthClient(
BuiltInOAuthClient.Facebook,
consumerKey: "",   // for Facebook, consumerKey is called AppID in the SDK
consumerSecret: "");
```

2. Copy the **App ID** value from the Facebook application as the value of the `consumerKey` parameter (inside the quotation marks).

3. Copy **App Secret** value from the Facebook application as the `consumerSecret` parameter value.

4. Save and close the file.

5. Edit the *~/Account/Login.cshtml* page and remove the comments from the `<fieldset>` block near the end of the page. To uncomment the code, remove the `@*` characters that precede and follow the `<fieldset>` block. The code block with comments removed looks like the following:

```
<fieldset>
<legend>Log in using another service</legend>
<input type="submit" name="provider" id="facebook" value="Facebook" title="Log in using your Facebook
account." />
<input type="submit" name="provider" id="twitter" value="Twitter" title="Log in using your Twitter
account." />
<input type="submit" name="provider" id="windowsLive" value="WindowsLive" title="Log in using your
Windows Live account." />
</fieldset>
```

6. Save and close the file.

Now you can test the Facebook login.

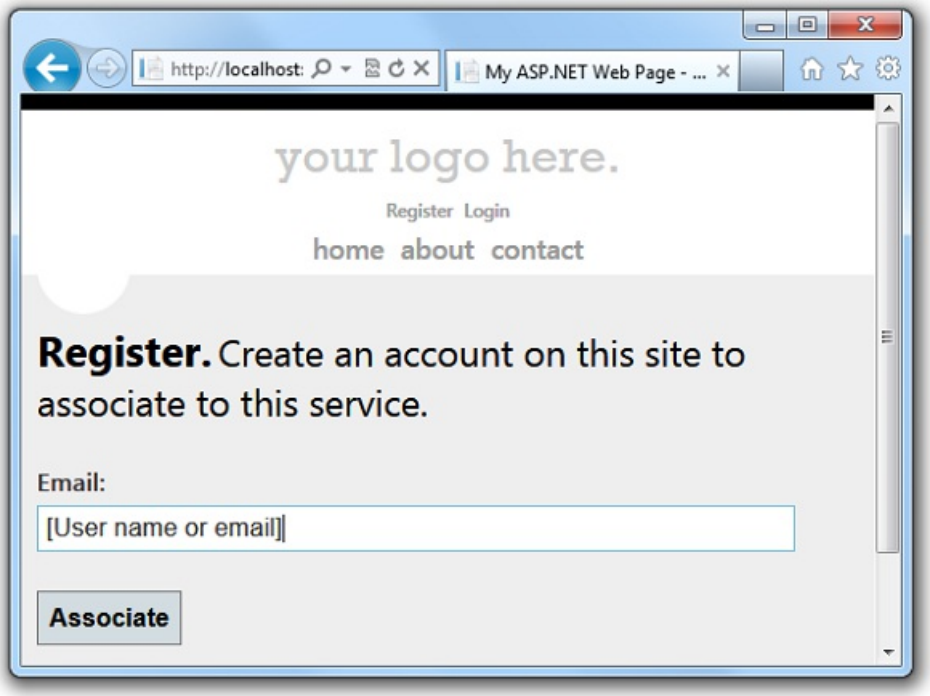
1. Run the site's *default.cshtml* page and choose the **Login** button.

2. On the *Login* page, in the **Use another service to log in** section, choose the **Facebook** icon.

   The web page redirects the request to the Facebook login page.

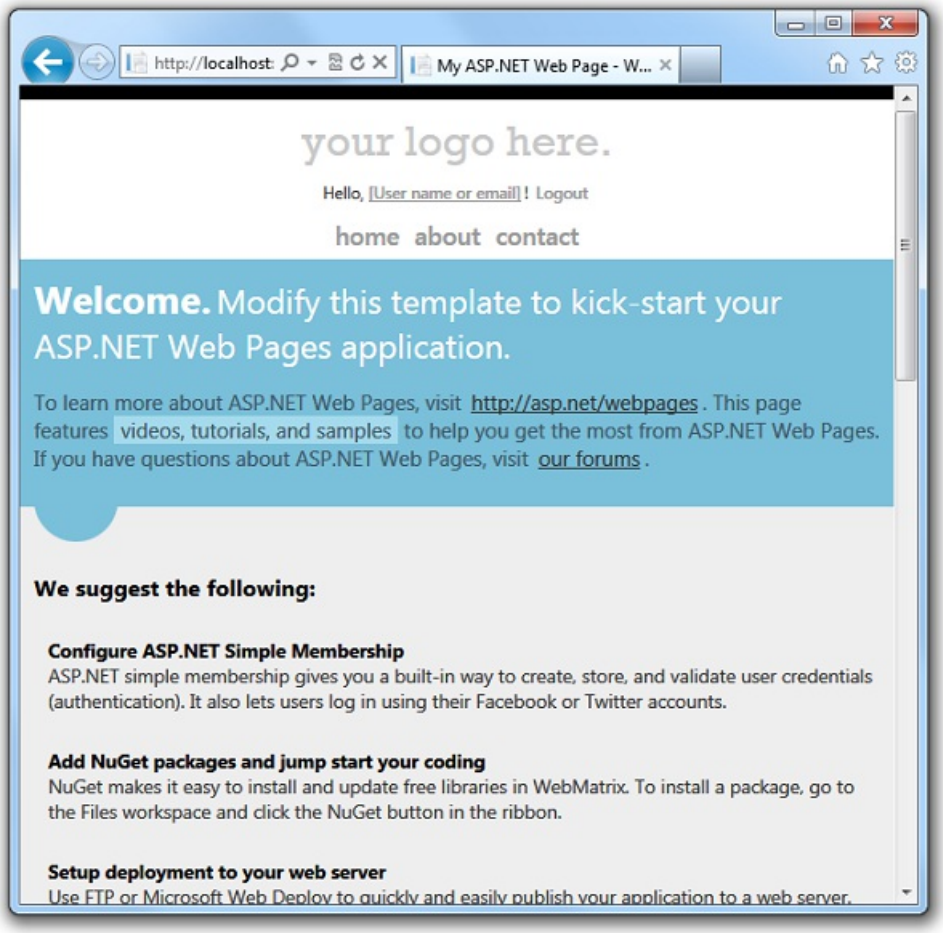
3. Log into a Facebook account.

   The code uses the Facebook token to authenticate you and then returns to a page where you can associate your Facebook login with your site's login. Your user name or email address is filled into the **Email** field on the form.



4. Choose the **Associate** button.

   The browser returns to the home page and you are logged in.



To enable Twitter logins:

1. Browse to the Twitter developers site.
2. Choose the **Create an App** link and then log into the site.
3. On the **Create an Application** form, fill in the **Name** and **Description** fields.
4. In the **WebSite** field, enter the URL of your site (for example, `http://www.example.com`).

> **NOTE**
> If you're testing your site locally (using a URL like `http://localhost:12345`), Twitter might not accept the URL. However, you might be able to use the local loopback IP address (for example `http://127.0.0.1:12345`). This simplifies the process of testing your application locally. However, every time the port number of your local site changes, you'll need to update the **WebSite** field of your application.

5. In the **Callback URL** field, enter a URL for the page in your website that you want users to return to after logging into Twitter. For example, to send users to the home page of the Starter Site (which will recognize their logged-in status), enter the same URL that you entered in the **WebSite** field.
6. Accept the terms and choose the **Create your Twitter application** button.
7. On the **My Applications** landing page, choose the application you created.
8. On the **Details** tab, scroll to the bottom and choose the **Create My Access Token** button.
9. On the **Details** tab, copy the **Consumer Key** and **Consumer Secret** values for your application and paste them into a temporary text file. You'll pass these values to the Twitter provider in your website code.
10. Exit the Twitter site.

Now you make changes to two pages in your website so that users will be able to log into the site using their Twitter accounts.

1. In your website, edit the *_AppStart.cshtml* page and uncomment the code for the Twitter OAuth provider. The uncommented code block looks like this:

```
OAuthWebSecurity.RegisterOAuthClient(
BuiltInOAuthClient.Twitter,
consumerKey: "",
consumerSecret: "");
```

2. Copy the **Consumer Key** value from the Twitter application as the value of the `consumerKey` parameter (inside the quotation marks).

3. Copy the **Consumer Secret** value from the Twitter application as the value of the `consumerSecret` parameter.

4. Save and close the file.

5. Edit the *~/Account/Login.cshtml* page and remove the comments from the `<fieldset>` block near the end of the page. To uncomment the code, remove the `@*` characters that precede and follow the `<fieldset>` block. The code block with comments removed looks like the following:

```
<fieldset>
<legend>Log in using another service</legend>
<input type="submit" name="provider" id="facebook" value="Facebook" title="Log in using your Facebook
account." />
<input type="submit" name="provider" id="twitter" value="Twitter" title="Log in using your Twitter
account." />
<input type="submit" name="provider" id="windowsLive" value="WindowsLive" title="Log in using your
Windows Live account." />
</fieldset>
```

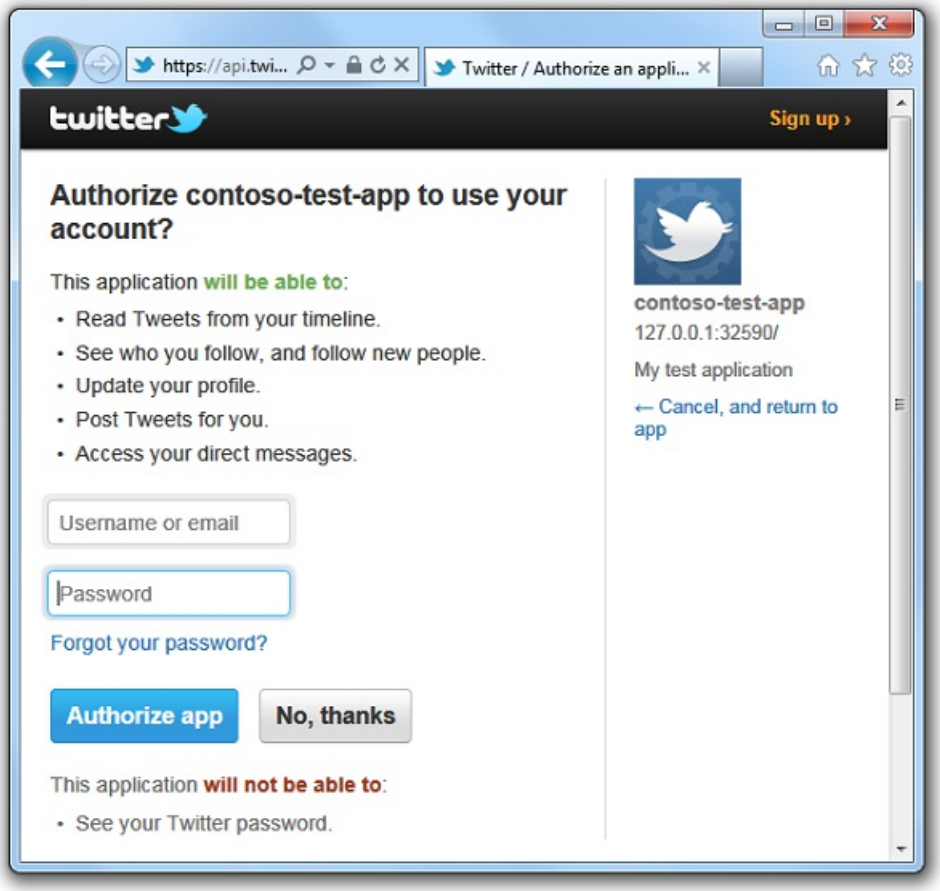
6. Save and close the file.

Now you can test the Twitter login.

1. Run the *default.cshtml* page of your site and choose the **Login** button.

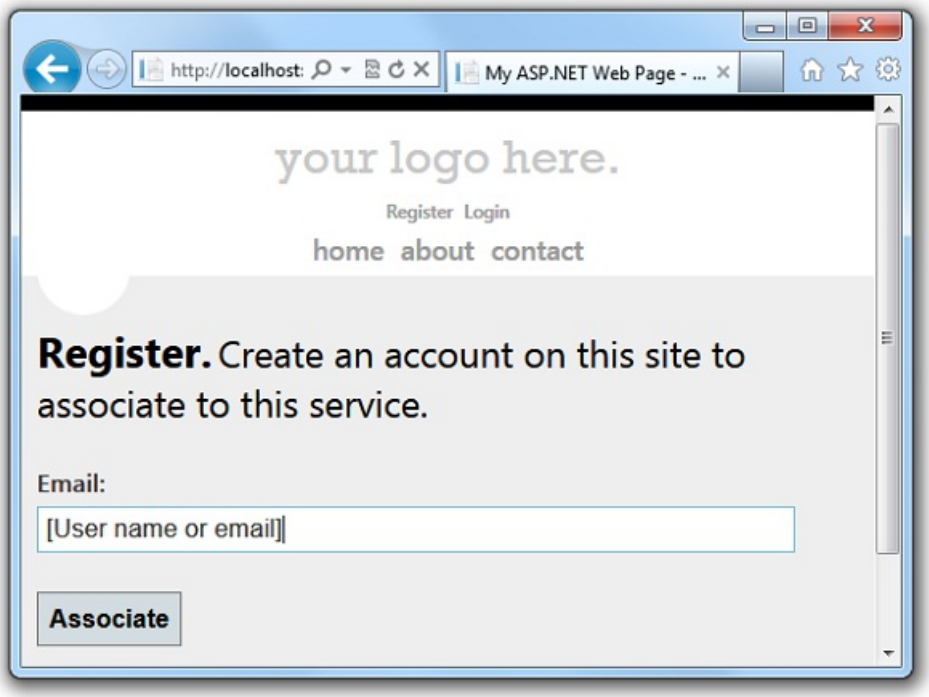
2. On the *Login* page, in the **Use another service to log in** section, choose the **Twitter** icon.

   The web page redirects the request to a Twitter login page for the application you created.



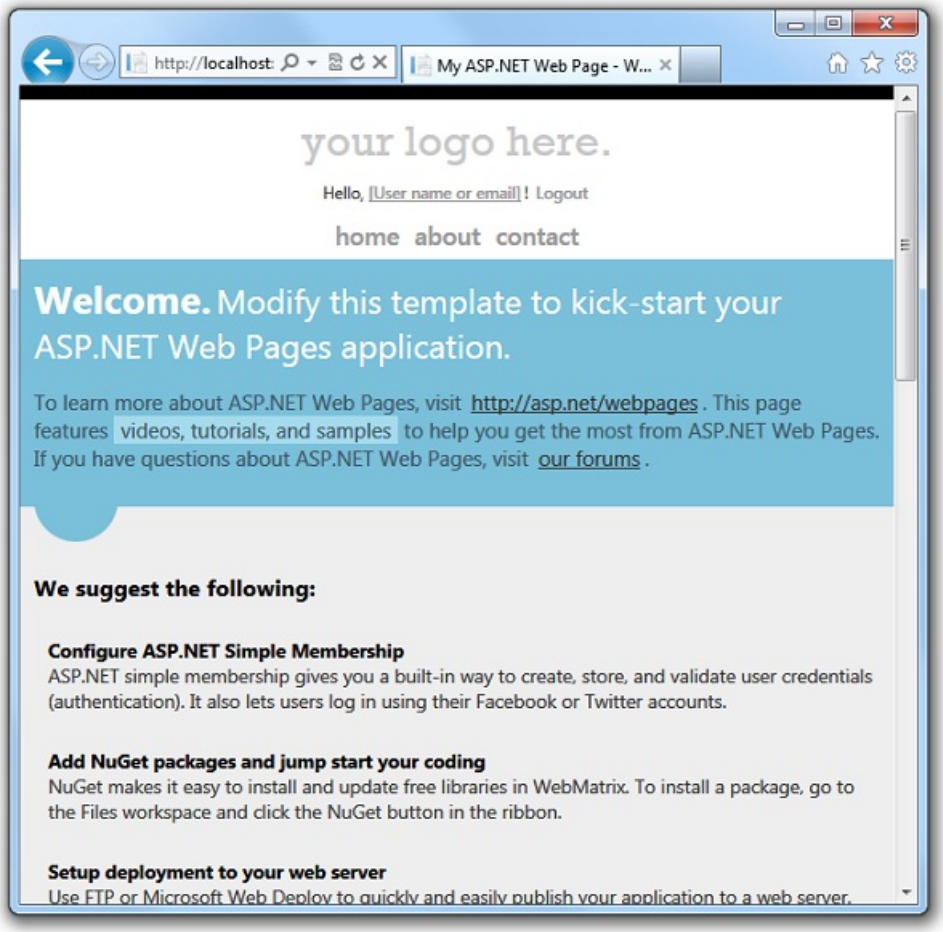
3. Log into a Twitter account.

4. The code uses the Twitter token to authenticate the user and then returns you to a page where you can associate your login with your website account. Your name or email address is filled into the **Email** field on the form.



5. Choose the **Associate** button.

   The browser returns to the home page and you are logged in.



Adding Maps Using the Maps Helper

Web Pages 2 includes additions to the ASP.NET Web Helpers Library, which is a package of add-ins for a Web Pages site. One of these is a mapping component provided by the `Microsoft.Web.Helpers.Maps` class. You can use the `Maps` class to generate maps based either on an address or on a set of longitude and latitude coordinates. The `Maps` class lets you call directly into popular map engines including Bing, Google, MapQuest, and Yahoo.

To use the new `Maps` class in your website, you must first install the version 2 of the Web Helpers Library. To do this, go to the instructions for installing the currently released version of the ASP.NET Web Helpers Library and install version 2.

The steps for adding mapping to a page are the same regardless of which of the map engines you call. You just add a JavaScript file reference to your mapping page and then add a call that renders the `<script>` tags on your page. Then on your mapping page, call the map engine you want to use.

The following example shows how to create a page that renders a map based on an address, and another page that renders a map based on longitude and latitude coordinates. The address mapping example uses Google Maps, and the coordinate mapping example uses Bing Maps. Note the following elements in the code:

- The call to `Assets.AddScript` at the top of the two mapping pages. This method adds a reference to the *jquery-1.6.2.min.js* file that is included with the **Starter Site** template and that's required by the `Maps` class.
- The call to the `Assets.GetScripts` method in the layout file. This method renders the `<script>` tag on the two mapping pages.
- The call to the `@Maps.GetGoogleHtml` and the `@Maps.GetBingHtml` methods in the mapping pages. To map an address, you must pass an address string. To map coordinates, you must pass longitude and latitude coordinates. For the Bing Maps engine, you must also pass a key (which you get for free by signing up at the Bing Maps Developers site). The methods for the other map engines work in a similar way (`@Maps.GetYahooHtml` , `@Maps.GetMapQuestHtml`).

To create mapping pages:

1. Create a website based on the **Starter Site** template.
2. Create a file named *MapAddress.cshtml* in the root of the site. This page will generate a map based on an address that you pass to it.
3. Copy the following code into the file, overwriting the existing content.

```
@{
Layout = "~/_MapLayout.cshtml";
Assets.AddScript("~/Scripts/jquery-1.6.2.min.js");
}

<div id="navigation">
<h3>Map an Address:</h3>
<form method="post" action="" id="coordinates" style="width: 320px">
<fieldset>
  <div>
    <label for="address">Address:</label>
    <input style="width: 300px" type="text"  name="address"  value="@Request["address"]"/>
    <input type="submit" value="Map It!" />
   </div>
</fieldset>
</form>
</div>
<div id="content">
@if(IsPost) {
var theAddress = Request["address"];
@Maps.GetGoogleHtml(theAddress,
    width: "800",
    height: "800")
}
</div>
```

4. Create a file named *_MapLayout.cshtml* in the root of the site. This page will be the layout page for the two mapping pages.
5. Copy the following code into the file, overwriting the existing content.

```
<!DOCTYPE html>

<html lang="en">
<head>
<meta charset="utf-8" />

<title></title>
<style type="text/css">
    #navigation {
        position: absolute;
        top: 1em;
        left: 3em;
        width: 18em;
    }
    #content {
        margin-top: 10em;
        margin-left: 3em;
    }
</style>
</head>
<body>
  @RenderBody()

  @*
Call the Assets helper to render tags for the Maps helper. For
releases after Web Pages 2 Beta, call Assets.GetScripts().
  *@
  @Assets.GetScripts()
</body>
</html>
```

6. Create a file named *MapCoordinates.cshtml* in the root of the site. This page will generate a map based on a set of coordinates that you pass to it.

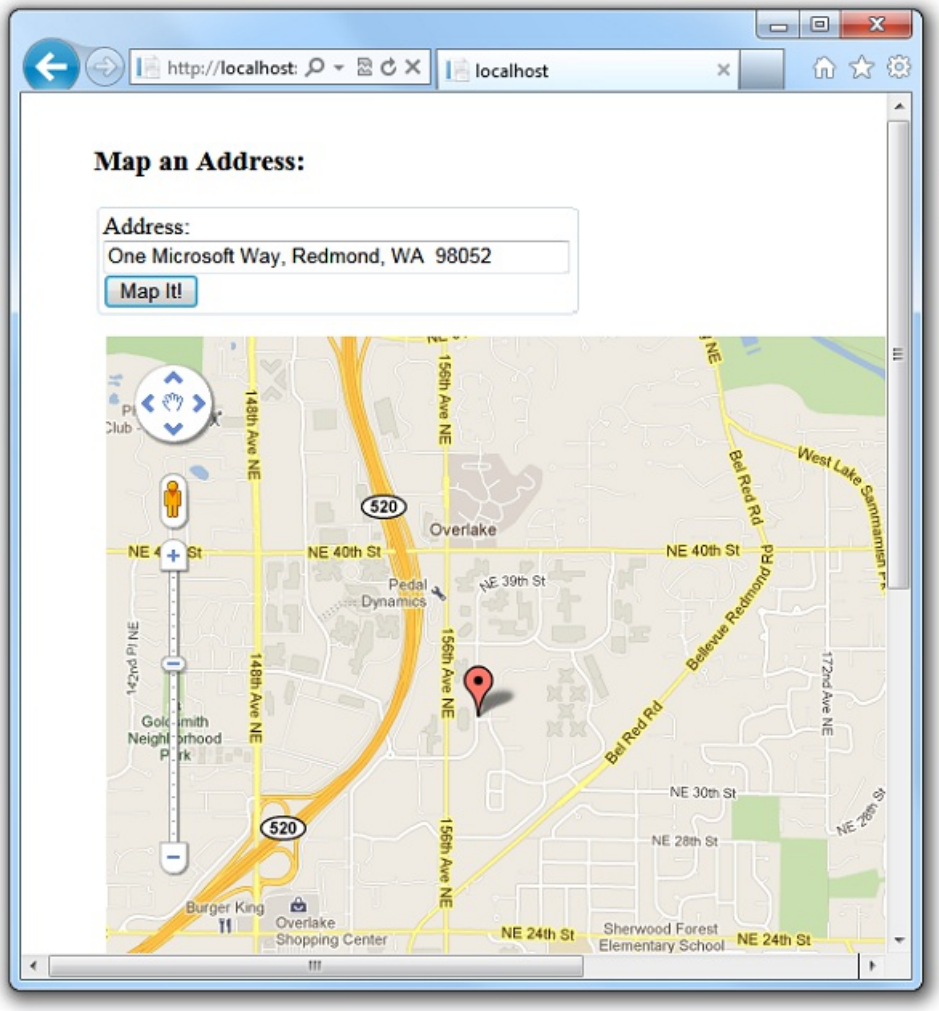
7. Copy the following code into the file, overwriting the existing content.

```
@{
Layout = "~/_MapLayout.cshtml";
Assets.AddScript("~/Scripts/jquery-1.6.2.min.js");
}

<div id="navigation">
<h3>Map a set of coordinates:</h3>
<form method="post" action="" id="coordinates">
<fieldset>
<div>
    <label for="latitude">Latitude:   </label>
    <input type="text"  name="latitude" value="@Request["latitude"]"/>
</div>
<div>
    <label for="longitude">Longitude:</label>
    <input type="text" name="longitude" value="@Request["longitude"]"/>
</div>
<div>
    <input type="submit" value="Map It!" />
</div>
</fieldset>
</form>
</div>
<div id="content">
@if(IsPost) {
var theLatitude = Request["latitude"];
var theLongitude = Request["longitude"];
@Maps.GetBingHtml("Ag6C5Ci9VUGz9AIhQyJ7YNwGraanqwy5-V3LK1qGDpdEVPV-MUPBryG18ELoC6g6",
    "",
    theLatitude,
    theLongitude,
    width: "800",
    height: "800")
}
</div>
```

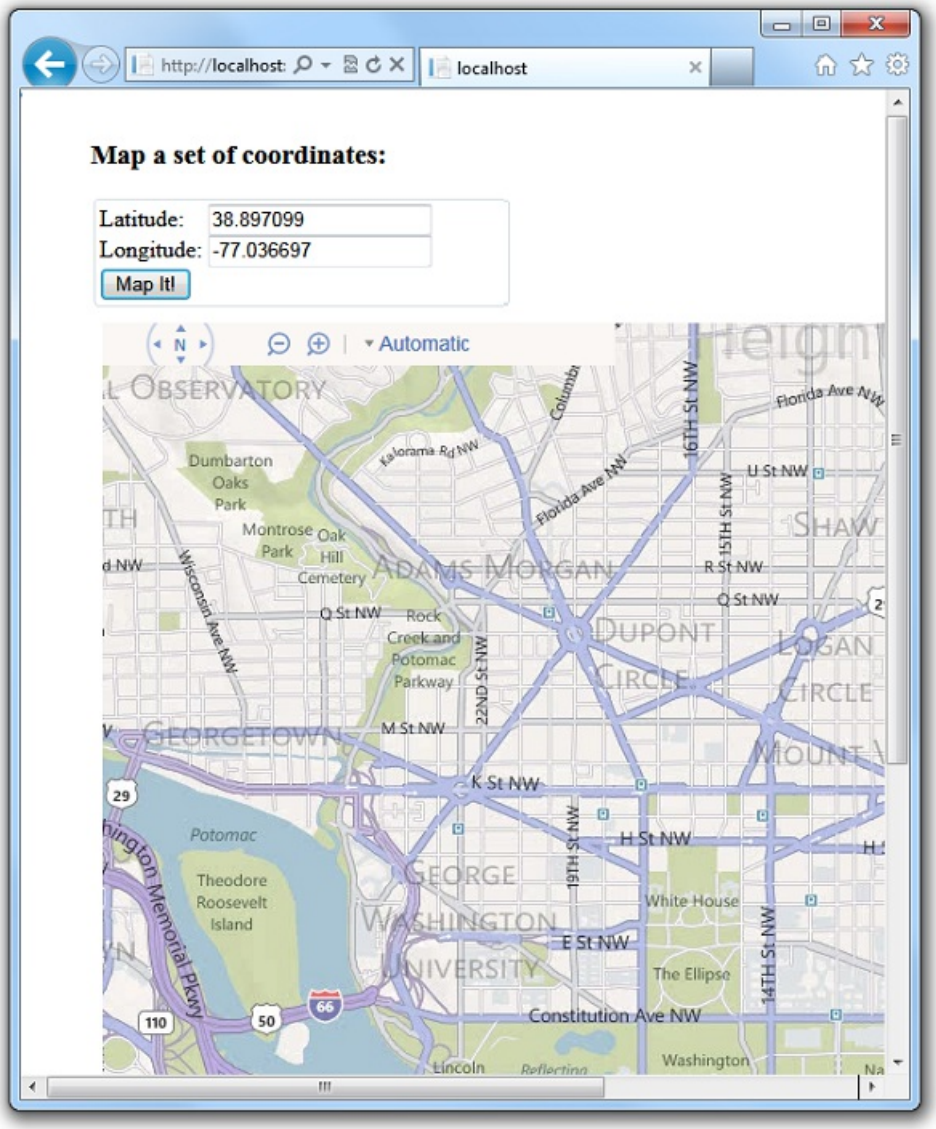
To test your mapping pages:

1. Run the page *MapAddress.cshtml* file.
2. Enter a full address string including a street address, state or province, and postal code, and then choose the **Map It** button. The page renders a map from Google Maps:



3. Find a set of latitude and longitude coordinates for a specific location.

4. Run the page *MapCoordinates.cshtml*. Enter the coordinates and then choose the **Map It** button. The page renders a map from Bing Maps:



Running Web Pages Applications Side by Side

Web Pages 2 adds the ability to run applications side by side. This lets you continue to run your Web Pages 1 applications, build new Web Pages 2 applications, and run all of them on the same computer.

Here are some things to remember when you install the Web Pages 2 Beta with WebMatrix:

- By default, existing Web Pages applications will run as version 2 applications on your computer. (The assemblies for version 2 are installed in the GAC and will be used automatically.)
- If you want to run a site using Web Pages version 1 (instead of the default, as in the previous point), you can configure the site to do that. If your site doesn't already have a *web.config* file in the root of the site, create a new one and copy the following XML into it, overwriting the existing content. If the site already contains a *web.config* file, add an `<appSettings>` element like the following one to the `<configuration>` section.

  ```
  <?xml version="1.0"?>
  <configuration>
  <appSettings>
  <add key="webPages:Version" value="1.0"/>
  </appSettings>
  </configuration>
  ```

  `- If you do not specify a version in the *web.config* file, a site is deployed as a version 2 site. (The version 2 assemblies are copied to the *bin* folder in the deployed site.)
- New applications that you create using the site templates in Web Matrix version 2 Beta include the Web

Pages version 2 assemblies in the site's *bin* folder.

In general, you can always control which version of Web Pages to use with your site by using NuGet to install the appropriate assemblies into the site's *bin* folder. To find packages, visit NuGet.org.

Rendering Pages for Mobile Devices

Web Pages 2 lets you create custom displays for rendering content on mobile or other devices.

The `System.Web.WebPages` namespace contains the following classes that let you work with display modes: `DefaultDisplayMode`, `DisplayInfo`, and `DisplayModes`. You can use these classes directly and write code that renders the right output for specific devices.

Alternatively, you can create device-specific pages by using a file-naming pattern like this: *FileName.Mobile.cshtml*. For example, you can create two versions of a page, one named *MyFile.cshtml* and one named *MyFile.Mobile.cshtml*. At run time, when a mobile device requests *MyFile.cshtml*, Web Pages renders the content from *MyFile.Mobile.cshtml*. Otherwise, *MyFile.cshtml* is rendered.

The following example shows how to enable mobile rendering by adding a content page for mobile devices. *Page1.cshtml* contains content plus a navigation sidebar. *Page1.Mobile.cshtml* contains the same content, but omits the sidebar.

To build and run the code sample:

1. In a Web Pages site, create a file named *Page1.cshtml* and copy the *Page1.cshtml* content into it from the example.
2. Create a file named *Page1.Mobile.cshtml* and copy the *Page1.Mobile.cshtml* content into it from the example. Notice that the mobile version of the page omits the navigation section for better rendering on a smaller screen.
3. Run a desktop browser and browse to *Page1.cshtml*.
4. Run a mobile browser (or a mobile device emulator) and browse to *Page1.cshtml*. Notice that this time Web Pages renders the mobile version of the page.

> **NOTE**
>
> To test mobile pages, you can use a mobile device simulator that runs on a desktop computer. This tool lets you test web pages as they would look on mobile devices (that is, typically with a much smaller display area). One example of a simulator is the User Agent Switcher add-on for Mozilla Firefox, which lets you emulate various mobile browsers from a desktop version of Firefox.

Page1.cshtml

```
<!DOCTYPE html>

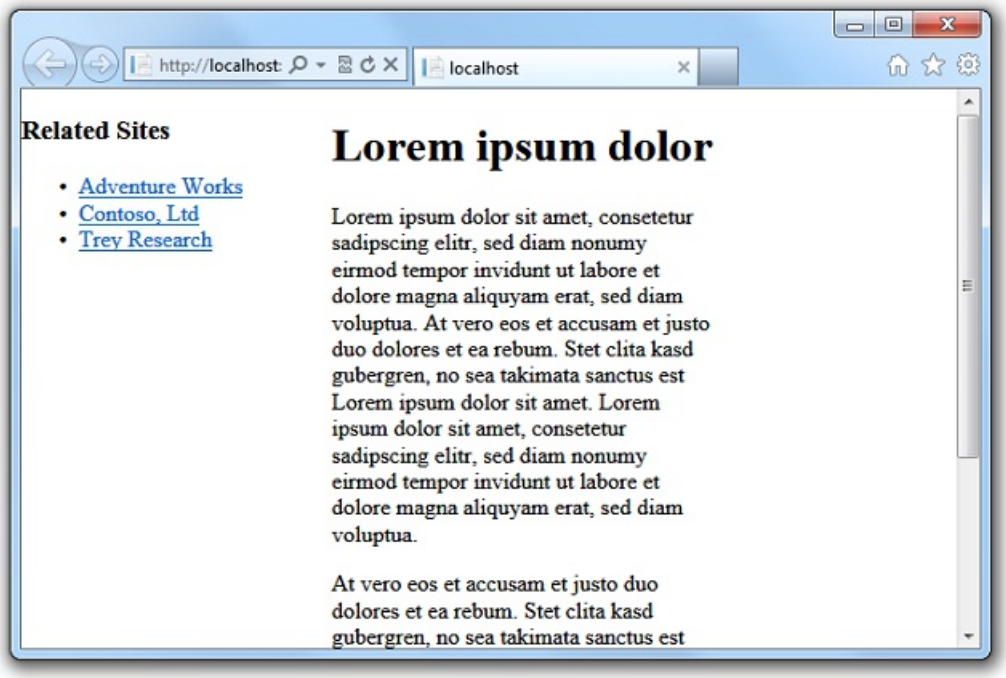
<html lang="en">
<head>
<meta charset="utf-8" />
<title></title>
<style type="text/css">
    #navigation {
        position: absolute;
        top: 0;
        left: 0;
        width: 10em;
    }
    #content {
    margin-left: 13em;
        margin-right: 10em;
    }
</style>
</head>
<body>
<div id="navigation">
    <h3>Related Sites</h3>
    <ul>
        <li><a href="http://www.adventure-works.com/">Adventure Works</a></li>
        <li><a href="http://www.contoso.com/">Contoso, Ltd</a></li>
        <li><a href="http://www.treyresearch.net/">Trey Research</a></li>
    </ul>
</div>
<div id="content">
    <h1>Lorem ipsum dolor</h1>
    <p>Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
    eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.
    At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren,
    no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit
    amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut
    labore et dolore magna aliquyam erat, sed diam voluptua. </p>
    <p>At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd
    gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum
    dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt
    ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam
    et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata
    sanctus est Lorem ipsum dolor sit amet.</p>
</div>
</body>
</html>
```

Page1.Mobile.cshtml

```
<!DOCTYPE html>

<html lang="en">
<head>
<meta charset="utf-8" />
<title></title>
<style type="text/css">
    #content {
    margin-left: 2em;
        margin-right: 5em;
    }
</style>
</head>
<body>
<div id="content">
    <h1>Lorem ipsum dolor</h1>
    <p>Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy
    eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.
    At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren,
    no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit
    amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut
    labore et dolore magna aliquyam erat, sed diam voluptua. </p>
    <p>At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd
    gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum
    dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt
    ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam
    et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata
    sanctus est Lorem ipsum dolor sit amet.</p>
</div>
</body>
</html>
```

Page1.cshtml rendered in a desktop browser:



Page1.Mobile.cshtml displayed in an Apple iPhone simulator view in the Firefox browser. Even though the request is to *Page1.cshtml*, the application renders *Page1.Mobile.cshtml*.

Additional Resources

ASP.NET Web Pages 1 Resources

> **NOTE**
>
> Most Web Pages 1 programming and API resources still apply to Web Pages 2.

- Introduction to ASP.NET Web Pages Programming

WebMatrix Resources

- WebMatrix 2 What's New
- Microsoft WebMatrix Site
- Starting Web Development with Microsoft WebMatrix(includes a full-length sample Web Pages application)

ASP.NET Web Pages 2 Developer Preview ReadMe

4/10/2020 • 3 minutes to read • Edit Online

by Microsoft

ASP.NET Web Pages 2 Developer Preview ReadMe

14 September 2011

Contents

Installation Notes

To install Web Pages 2 Developer Preview, you have these options:

- Install WebMatrix 2 Beta using the Web Platform Installer. WebMatrix is a set of free web development tools that includes ASP.NET Web Pages. For more information, see the installation section in The Top Features in ASP.NET Web Pages 2 Developer Preview.
- Install Web Pages 2 Developer Preview directly using the download link. Use this approach if you want to create Web Pages applications using a text editor such as Notepad. In order to run Web Pages 2 applications, you must have IIS Express 7.5. (This is included automatically with WebMatrix.) For tips on how to test a Web Pages page using IIS Express, see the sidebar "Creating and Testing ASP.NET Pages Using Your Own Text Editor" in Getting Started with WebMatrix and ASP.NET Web Pages.

ASP.NET Web Pages 2 Developer Preview can be installed and can run side-by-side with ASP.NET Web Pages 1. For details, see the section "Running Web Pages Applications Side-by-Side" in The Top Features in Web Pages 2 Developer Preview.

Documentation

Tutorials and other information about ASP.NET Web Pages are available on the Web Pages page of the ASP.NET website (https://www.asp.net/web-pages/). For information about new features and enhancements in Web Pages 2, see The Top Features in Web Pages 2 Developer Preview.

Support

This is a preview release and is not officially supported. If you have questions about working with this release, post them to the ASP.NET Web Pages forum (https://forums.asp.net/1224.aspx/1?WebMatrix), where members of the ASP.NET community are frequently able to provide informal support.

Software Requirements

ASP.NET Web Pages 2 requires the .NET Framework 4. It also works with the .NET Framework 4.5 Developer Preview release.

Fixes, Known Issues, and Breaking Changes

- **Is* methods (for example, IsDateTime) now return correct values for all cultures.** Some methods like *IsDateTime* previously returned *false* when they should have returned *true* because they were previously performing culture-specific checks. These methods have been fixed to now take culture into account. This is a breaking change; if your application relies on the old behavior, it will break.
- **The behavior of the Href method has changed.** Previously, calling Href("~/SomeFile") would return a URL relative to the currently executing file. Now Href("~/SomeFile") always returns an absolute path from the root of the application. For most cases, this behavior won't make a difference in the return value. This change was made to fix certain Ajax scenarios. For example, consider the following example code:

```
/Folder/File
@Href("~/Images/Logo.jpg")
```

This code previously would resolve to Images/Logo.jpg, which would be incorrect for an Ajax request to that page. It will now resolve to the root of the (/MySite/Images/Logo.jpg).

- **The HttpContext.RedirectLocal method has changed**. This method now accepts only URLs that are relative to the current application. Fully qualified URLs are rejected.
- **The ModelState.IsValid method now requires you to call Validate first**. If you are converting your application to use the new input validation methods and are calling the *ModelState.IsValid* method, you must now call *Validation.Validate* beforehand. For example, you must now follow this pattern:

```
Validation.RequireField("MyField");
if (IsPost) {
    Validation.Validate();
    if (ModelState.IsValid) {
        // do something
    }
}
```

However, we recommend that if you use the new input validation methods, don't use *ModelState.IsValid*. Instead, structure your code like this:

```
Validation.RequireField("MyField");
if (IsPost) {
    if (Validation.IsValid()) {
        // do something
    }
}
```

- **On Internet Explorer 7 and Internet Explorer 8, client-side validation does not work**. Client-side validation does not work due to incompatibilities with jQuery 1.6.2, which is included with the default project template. (Server-side validation works.).

Disclaimer

© 2011 Microsoft Corporation. All rights reserved. This document is provided "as-is." Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

Running Different Versions of ASP.NET Web Pages (Razor) Side by Side

2/20/2020 • 2 minutes to read • Edit Online

by Tom FitzMacken

This article explains how to run ASP.NET Web Pages (Razor) websites on the same computer or server when the websites are configured to use different versions of ASP.NET Web Pages.

What you'll learn:

- What the default behavior is in ASP.NET when you have sites built with ASP.NET Web Pages.
- How to configure a new site to run with an older version of ASP.NET Web Pages.

This is the ASP.NET feature introduced in the article:

- The `webPages:Version` configuration setting.

Software versions

- ASP.NET Web Pages (Razor) 3

This tutorial also works with ASP.NET Web Pages 2 and ASP.NET Web Pages 1.0.

ASP.NET Web Pages supports the ability to run websites side by side. This lets you continue to run your older ASP.NET Web Pages applications, build new ASP.NET Web Pages applications, and run all of them on the same computer.

Here are some things to remember when you install the Web Pages with WebMatrix:

- By default, existing Web Pages applications will run as the latest version on your computer. (The assemblies are installed in the global assembly cache (GAC) and are used automatically.)

- If you want to run a site using a different version of ASP.NET Web Pages, you can configure the site to do that. If your site doesn't already have a *web.config* file in the root of the site, create a new one and copy the following XML into it, overwriting the existing content. If the site already contains a *web.config* file, add an `<appSettings>` element like the following one to the `<configuration>` section.

  ```
  <?xml version="1.0"?>
  <configuration>
    <appSettings>
      <add key="webPages:Version" value="1.0"/>
    </appSettings>
  </configuration>
  ```

  `- If you do not specify a version in the *web.config* file, a site is deployed as the latest version. (The assemblies are copied to the *bin* folder in the deployed site.)

- New applications that you create using the site templates in Web Matrix include the Web Pages version assemblies in the site's *bin* folder.

In general, you can always control which version of Web Pages to use with your site by using NuGet to install the

appropriate assemblies into the site's *bin* folder. To find packages, visit NuGet.org.

Additional Resources

The Top Features in ASP.NET Web Pages 2

Videos

4/10/2020 • 2 minutes to read • Edit Online

> **NOTE**
>
> WebMatrix is no longer recommended as an integrated development environment for ASP.NET Web Pages. Use Visual Studio or Visual Studio Code.

- Introduction
  - 5 Minute Intro to ASP.NET Web Pages
  - Install and Use the Microsoft WebMatrix Tool
  - Create a Website Using WebMatrix
  - Change the Visual Style of a Web Page
  - Use the Layout Features in WebMatrix
  - Create a Data-Driven Dynamic Web Page
  - Create a Web Interface in WebMatrix
  - Create an Edit Data Page in WebMatrix
  - Create a Delete Data Page in WebMatrix
  - Publish a Website to the Internet Using WebMatrix
- ASP.NET Razor Pages
  - Getting Started with WebMatrix and ASP.NET Web Pages
  - Intro to ASP.NET Web Programming Razor Syntax
  - Creating a Consistent Look (Part 1)
  - Creating a Consistent Look (Part 2)
  - Working with Forms (Part 1)
  - Working with Forms (Part 2)
  - Working with Data (Part 1)
  - Working with Data (Part 2)
  - Displaying Data in a Grid
  - Displaying Data in a Chart (Part 1)
  - Displaying Data in a Chart (Part 2)
  - Working with Files
  - Working with Images
  - Working with Video
  - Adding Email to Your Web Site
  - Adding Search to Your Web Site
  - Adding Social Networking to Your Website

Introduction to Web Pages Videos

4/10/2020 • 2 minutes to read • Edit Online

Which videos you can view to learn the basics about Web Pages and WebMatrix.

> **NOTE**
>
> WebMatrix is no longer recommended as an integrated development environment for ASP.NET Web Pages. Use Visual Studio or Visual Studio Code.

- 5 Minute Intro to ASP.NET Web Pages
- Install and Use the Microsoft WebMatrix Tool
- Create a Website Using WebMatrix
- Change the Visual Style of a Web Page
- Use the Layout Features in WebMatrix
- Create a Data-Driven Dynamic Web Page
- Create a Web Interface in WebMatrix
- Create an Edit Data Page in WebMatrix
- Create a Delete Data Page in WebMatrix
- Publish a Website to the Internet Using WebMatrix

5 Minute Introduction to ASP.NET Web Pages

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Watch video (4 minutes)

Install and Use the Microsoft WebMatrix Tool

4/10/2020 • 2 minutes to read • Edit Online

by Advaiya

> **NOTE**
>
> WebMatrix is no longer recommended as an integrated development environment for ASP.NET Web Pages. Use Visual Studio or Visual Studio Code.

Part 1 introduces you to WebMatrix and how you can install and use it.

Watch video (2 minutes)

Create a Website Using WebMatrix

2/20/2020 • 2 minutes to read • Edit Online

by Advaiya

Part 2 teaches you how to create your first Web Page using WebMatrix.

Watch video (2 minutes)

Change the Visual Style of a Web Page

2/20/2020 • 2 minutes to read • Edit Online

by Advaiya

Part 3 gets you started with CSS and style sheets so you can make your page more beautiful.

Watch video (4 minutes)

Use the Layout Features in WebMatrix

2/20/2020 • 2 minutes to read • Edit Online

by Advaiya

Part 4 teaches you layout and how you can use layout in WebMatrix for common code across your pages.

Watch video (5 minutes)

Create a Data-Driven Dynamic Web Page

2/20/2020 • 2 minutes to read • Edit Online

by Advaiya

Part 5 is all about data, where you'll learn how to use a database in your web page, and make it more dynamic.

Watch video (4 minutes)

Create a Web Interface in WebMatrix

2/20/2020 • 2 minutes to read • Edit Online

by Advaiya

Part 6 shows you how to build a page that allows your users to add data to your database, so your site can render it.

Watch video (3 minutes)

Create an Edit Data Page in WebMatrix

2/20/2020 • 2 minutes to read • Edit Online

by Advaiya

Part 7 builds on this to show you how you can have a page to edit your database, and see the changes updated automatically.

Watch video (3 minutes)

Create a Delete Data Page in WebMatrix

2/20/2020 • 2 minutes to read • Edit Online

by Advaiya

Part 8 finishes off the application by showing you how to create a delete page for your data.

Watch video (3 minutes)

Publish a Website to the Internet Using WebMatrix

2/20/2020 • 2 minutes to read • Edit Online

by Advaiya

Part 9 shows you how to publish your site to the internet using WebMatrix.

Watch video (4 minutes)

ASP.NET Web Pages

2/20/2020 • 2 minutes to read • Edit Online

Which videos you can view to drill down into Web Pages and WebMatrix.

- Getting Started with WebMatrix and ASP.NET Web Pages
- Intro to ASP.NET Web Programming Razor Syntax
- Creating a Consistent Look (Part 1)
- Creating a Consistent Look (Part 2)
- Working with Forms (Part 1)
- Working with Forms (Part 2)
- Working with Data (Part 1)
- Working with Data (Part 2)
- Displaying Data in a Grid
- Displaying Data in a Chart (Part 1)
- Displaying Data in a Chart (Part 2)
- Working with Files
- Working with Images
- Working with Video
- Adding Email to Your Web Site
- Adding Search to Your Web Site
- Adding Social Networking to Your Website

Getting Started with WebMatrix and ASP.NET Web Pages

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Getting Started with WebMatrix and ASP.NET Web Pages tutorial

Watch video (27 minutes)

NEXT

Introduction to ASP.NET Web Programming Using the Razor Syntax

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Introduction to ASP.NET Web Programming Using the Razor Syntax tutorial

Watch video (9 minutes)

PREVIOUS NEXT

Creating a Consistent Look (Part 1)

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Creating a Consistent Look tutorial (part 1)

Watch video (23 minutes)

PREVIOUS NEXT

Creating a Consistent Look (Part 2)

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Creating a Consistent Look tutorial (Part 2)

Watch video (5 minutes)

PREVIOUS NEXT

Working with Forms (Part 1)

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Working with Forms tutorial (Part 1)

Watch video (14 minutes)

PREVIOUS NEXT

Working with Forms (Part 2)

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Working with Forms tutorial (Part 2)

Watch video (16 minutes)

PREVIOUS NEXT

Working with Data (Part 1)

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Working with Data tutorial (Part 1)

Watch video (8 minutes)

PREVIOUS NEXT

Working with Data (Part 2)

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Working with Data tutorial (Part 2)

Watch video (6 minutes)

PREVIOUS NEXT

Displaying Data in a Grid

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Displaying Data in a Grid tutorial

Watch video (17 minutes)

PREVIOUS NEXT

Displaying Data in a Chart (Part 1)

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Displaying Data in a Chart tutorial (Part 1)

Watch video (12 minutes)

PREVIOUS NEXT

Displaying Data in a Chart (Part 2)

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Displaying Data in a Chart (Part 2) tutorial

Watch video (9 minutes)

PREVIOUS NEXT

Working with Files

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Working with Files tutorial

Watch video (23 minutes)

PREVIOUS NEXT

Working with Images

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Working with Images tutorial

Watch video (26 minutes)

PREVIOUS NEXT

Working with Video

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Working with Video tutorial

Watch video (9 minutes)

PREVIOUS NEXT

Adding Email to Your Web Site

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Adding Email to Your Web Site (Part 1)

Watch video (10 minutes)

PREVIOUS NEXT

Adding Search to Your Web Site

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Adding Search to Your Web Site tutorial

Watch video (6 minutes)

PREVIOUS NEXT

Adding Social Networking to Your Website

2/20/2020 • 2 minutes to read • Edit Online

by Joe Stagner

Video accompaniment to the Adding Social Networking to Your Website tutorial

Watch video (8 minutes)

PREVIOUS

WebMatrix and Web Pages Video Training from Pluralsight

4/10/2020 • 2 minutes to read • Edit Online

> **NOTE**
>
> WebMatrix is no longer recommended as an integrated development environment for ASP.NET Web Pages. Use Visual Studio or Visual Studio Code.

Videos

- Introducing WebMatrix
- Sites and Files Workspaces
- Working with Databases
- Working with Standard Web Documents
- ASP.NET Web Pages
- Reporting and Deployment
- Source Control
- Mobile Websites

"This course recorded by Matt Milner for Pluralsight is a great way to get all the details you need to create dynamic data-driven applications powered by ASP.NET Web Pages but also the powerful new Razor syntax. Big thanks to Pluralsight for this free source! The video player, content and flow is really well done. I encourage you to check it out now!"

Scott Hanselman, Principal Program Manager, Web Platform and Azure Tools, Microsoft

Unlock access to 3,000+ dev, IT and creative courses that you can watch anytime, anywhere.

Starting at $ 35/mo.Start Free Trial or Subscribe Now

WebMatrix Readme

2/20/2020 • 2 minutes to read • Edit Online

- WebMatrix Readme
- Web Matrix and ASP.NET Web Pages (Razor) Beta 3 Release Readme

WebMatrix Readme

2/20/2020 • 21 minutes to read • Edit Online

13 January 2011

Contents

> **NOTE**
>
> This readme applies to the 1.0 release of WebMatrix.

- Overview
- Installation
- How to Publish Applications
- Changes and Issues
  - WebMatrix 1.0 Installation
  - ASP.NET Web Pages
  - WebMatrix
  - IIS Express
  - SQL Server Compact
  - Installing Applications
  - Publishing Applications
- For More Information

Overview

Microsoft WebMatrix 1.0 is a free web development stack that installs in minutes. It integrates a web server with database and programming frameworks to create a single, integrated experience. You can use WebMatrix to streamline the way you code, test, and publish your own ASP.NET or PHP website, or you can use WebMatrix to start a new website using popular open-source apps like DotNetNuke, Umbraco, WordPress, or Joomla. WebMatrix uses the same powerful web server, database engine, and frameworks environment that will run your website on the internet, which makes the transition from development to production smooth and seamless.

Installation

To install WebMatrix 1.0, you must first install the Microsoft Web Platform Installer 3.0. After you've installed the Web Platform Installer, you can use it to install WebMatrix.

If you have problems during installation, refer to Troubleshooting Problems with Microsoft Web Platform Installer.

How to Publish Applications

See Step-by-Step Instructions for Publishing Applications

Changes and Issues

WebMatrix 1.0 Installation Issues

Issue: WebMatrix 1.0 is available only on platforms that support Microsoft .NET Framework 4

The .NET Framework version 4 is required for WebMatrix. In certain cases, the WebMatrix 1.0 installer will let you try to install on a platform that is not part of the supported configuration set. In particular, Windows Vista without the SP1 update will let you begin the installation of WebMatrix, but the .NET Framework 4 component will fail and block your installation.

Workaround

Install on a supported platform, which includes:

- Windows 7
- Windows Server 2008
- Windows Server 2008 R2
- Windows Vista SP1 or later
- Windows XP SP3
- Windows Server 2003 SP2

Issue: Cannot install WebMatrix 1.0 if Microsoft Visual Studio 2008 is installed without Microsoft Visual Studio 2008 SP1

Workaround

Install Microsoft Visual Studio 2008 SP1 from the Microsoft Download Center.

Issue: Some assemblies for SQL Server Compact 4.0 are not installed in the GAC

The managed assemblies for SQL Server Compact 4.0 are not placed in the global assembly cache (GAC) when you install SQL Server Compact 4.0 on a 64-bit computer and the computer has only the .NET Framework 3.5 SP1 Client Profile installed. The managed assemblies that are not installed in the GAC are:

- *System.Data.SqlServerCe.dll* (ADO.NET provider)
- *System.Data.SqlServerCe.Entity.dll* (ADO.NET Entity Framework)

Workaround

Uninstall SQL Server Compact 4.0. Download and install the full version of .NET Framework 3.5 SP1 from the following location:

Microsoft .NET Framework 3.5 Service pack 1 (Full Package)

Then reinstall SQL Server Compact 4.0.

Issue: Cannot uninstall SQL Server Compact using the command line

Uninstallation of SQL Server Compact using command-line options does not work in this release.

Workaround

Use *Programs and Features* in the Windows Control Panel to uninstall Microsoft SQL Server Compact 4.0.

ASP.NET Web Pages

This section of the document describes new features, changes, and known issues with the 1.0 release of ASP.NET Web Pages with Razor syntax.

- New features
- Changes
- Issues

New Features

New: Configuration setting added to disable the package manager

A new `asp:AdminManagerEnabled` key is available for the `<appSettings>` element in the *web.config* file, which lets you completely disable the package manager. The default value for this element is true, meaning that if it is not included in the *web.config* file, the package manager is enabled. To disable the package manager, add the following element to the *web.config* file in the root of the website:

```
<configuration>
  <appSettings>
    <add key="asp:AdminManagerEnabled" value="false"/>
  </appSettings>
</configuration>
```

Changes

Change: "webPages:AdminFolderVirtualPath" key renamed to "asp:AdminFolderVirtualPath"

The `webPages:AdminFolderVirtualPath` key that can be added to the *web.config* file to specify the location of the package manager has been renamed to use the `asp:` namespace instead of the `webPages` namespace. If you have used this element, you must rename it in the configuration file.

Known Issues

Issue: Passwords for membership users no longer recognized

The algorithm for creating and storing membership (login) passwords has been changed to be more secure. As a result, the passwords stored for members (users) created in Beta versions of ASP.NET Razor will not be recognized.

Workaround If the site has not yet been put into production, remove the user records from the membership database. If database is live, programmatically regenerate existing passwords in the membership database.

Issue: Unexpected behavior when using a custom user table for membership

To initialize the membership provider for an ASP.NET Razor website, you call the `WebSecurity.InitializeDatabaseConnection` method. (In WebMatrix, the Starter Site template includes a call to this method in the *_AppStart.cshtml* file.) If the `autoCreateTables` parameter of this method is set to true (by default, it is set to true in the Starter Site template), and if an unrecognized table name is passed to the method (the second parameter), the method does not throw an error. Instead, it automatically creates the table.

This can be a problem if you intend to use a custom user table for membership but pass the wrong table name to the `WebSecurity.InitializeDatabaseConnection` method. Because the method does not by default raise an error if the table you specify does not exist, and because it instead creates a new table, the application can appear to be working. However, application code that relies on your custom user table (and on fields in it) can eventually fail with unexpected errors.

Workaround

Make sure that the name passed in the `InitializeDatabaseConnection` method matches the user profile table in the membership database, or make sure that the `autoCreateTables` parameter is set to false.

Issue: Error message "The Admin Module requires access to ~/App_Data"

Under some circumstances, trying to create users or otherwise work with the ASP.NET membership system can cause the page to display the error *The Admin Module requires access to ~/App_Data*. This occurs if the account that IIS or IIS Express is running under does not have permissions to create and write to the *App_Data* folder under the website root.

Workaround Manually create an *App_Data* folder for the website. Then make sure that the Windows account that the application runs under (typically NETWORK SERVICE) has read/write permissions for root folders of the application and for subfolders such as App_Data. More detailed information is available in the KnowledgeBase article Problems with SQL Server Express user instancing and ASP.net Web Application Projects.

Issue: "Failed to generate a user instance of SQL Server" error

If a WebMatrix Web application uses SQL Server Express and is running IIS 7.5 on Windows 7 or Windows Server 2008 R2, you might see an error that indicates that SQL Server cannot retrieve the user's local application path at run time.

Workaround Make sure that the Windows account that the application runs under (typically NETWORK SERVICE) has read/write permissions for root folders of the application and for subfolders such as *App_Data*. More detailed information is available in the KnowledgeBase article Problems with SQL Server Express user instancing and ASP.net Web Application Projects.

Issue: Files that contains package-manager resources or package-manager passwords are servable under IIS 6.0 and earlier

If you deploy an ASP.NET Web Pages (Razor) application that was built using the RC2 release, and if the application contains a *password.txt* or *packagesources.txt* file under */App_Data/admin*, IIS 6.0 will serve the file if requested, potentially exposing the passwords for your package manager instance.

Workaround Rename the *password.txt* or *packagesources.txt* file to *password.config* or *packagesources.config*. By default, IIS 6.0 does not serve files that have the *.config* extension. (In IIS 7, no files in the *App_Data* folder are served, so you do not need to rename the files.)

Issue: Uninstalling packages installed using the Beta 3 release does not completely remove package components

If you installed a package using the package manager in the Beta 3 release and then try to uninstall it using the current release, the package is not completely uninstalled. Using the package manager's **Uninstall** button removes some components, but leaves the package's library code and does not update the *package.config* file.

Workaround
Perform these steps:

1. Delete the *App_Data\packages* folder. This removes all packages.
2. Delete the *packages.config* file in the root of the website.

Issue: In Visual Studio, invoking the web-based package manager takes the application offline

If you are working in Visual Studio (not WebMatrix) and use the *_admin* functionality to start the package manager, Visual Studio takes the application offline and posts the *app_offline.htm* into the website root, which disrupts your ability to use the package manager.

NOTE

Although you would most typically see this behavior when using the web-based package manager interface, the same behavior occurs if you add, remove, or modify any files in the *App_Data* folder.

Workaround

To work with packages in Visual Studio, use the NuGet extension instead of the web-based package manager. For information, see the NuGet documentation. If you are working with other files in the *App_Data* folder, consider keeping the files elsewhere to avoid this issue. If that's not practical, delete the *app_offline.htm* file manually or wait until the site comes back online automatically (by default, after 30 seconds).

Issue: Visual Studio IntelliSense and project templates available only in ASP.NET MVC version 3

Installing ASP.NET Web Pages does not also install tools for Visual Studio such as IntelliSense and project templates for ASP.NET Web Pages applications.

Workaround To use IntelliSense and project templates for ASP.NET Web Pages applications in Visual Studio, install ASP.NET MVC 3 RC either through the Web Platform Installer or the stand-alone installer.

Issue: Reading feeds or other external data via a proxy server

If the server running the site is behind a proxy server, you might need to configure proxy information in the *web.config* file in order to be able to read information that comes from outside your site. For example, if you use the `ReCaptcha` helper, the helper communicates with the reCAPTCHA service, but might be blocked by your proxy server. Similarly, feeds that are used in ASP.NET Web Pages, such as the feed used by the package manager, might require proxy configuration.

If you experience problems in working with an external service or working with the package feed, put the following elements into your application's root *web.config* file:

```
<system.net>
  <defaultProxy>
    <proxy autoDetect="False"
       proxyaddress="[proxy URL]"/>
  </defaultProxy>
</system.net>
```

For more information about configuring a proxy server, see <proxy> Element (Network Settings) on the MSDN Web site.

Issue: Uninstalling the .NET Framework version 4 disables ASP.NET Web Pages with Razor Syntax

If you uninstall the .NET Framework version 4 and then reinstall it, ASP.NET Web Pages with Razor syntax is disabled. Pages with the *.cshtml* extension do not run correctly. ASP.NET Web Pages registers an assembly in the machine root *web.config* file, and removing the .NET Framework removes that file. Reinstalling the .NET Framework installs a new version of the configuration file, but does not add the reference for the ASP.NET Web Pages assembly.

Workaround After reinstalling the .NET Framework, reinstall ASP.NET Web Pages with Razor syntax. This adds the following element to the *web.config* file in the machine root, which is typically in the following location:

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config (32-bit)
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config (64-bit)
```

```
<compilation>
  <assemblies>
    <add assembly="System.Web.WebPages.Configuration, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />

    <-- Additional assemblies here. -->

  </assemblies>
</compilation>
```

Issue: Extensionless URLs do not find .cshtml/.vbhtml files on IIS 7 or IIS 7.5

On IIS 7 or IIS 7.5, requests with a URL like the following are not able to find pages that have the *.cshtml* or *.vbhtml* extension:

`http://www.example.com/ExampleSite/ExampleFile`

The issue arises because URL rewriting is not enabled by default for IIS 7 or IIS 7.5. The likeliest scenario is that you do not see the problem when testing locally using IIS Express, but you experience it when you deploy your website to a hosting website.

Workaround

- If you have control over the server computer, on the server computer install the update that is described in A update is available that enables certain IIS 7.0 or IIS 7.5 handlers to handle requests whose URLs do not end with a period.
- If you do not have control over the server computer (for example, you are deploying to a hosting website), add the following to the website's *web.config* file:

```
<system.webServer>
  <modules runAllManagedModulesForAllRequests="true"/>
</system.webServer>
```

Issue: Deploying an application to a computer that does not have SQL Server Compact installed

Applications that include SQL Server Compact databases can run on a computer where SQL Server Compact is not installed. Microsoft WebMatrix 1.0 automatically copies these binaries for you and performs the appropriate *web.config* file transforms.

Workaround If you need to copy these files and make the *web.config* file changes manually, do the following:

1. Copy the database engine assemblies to the *Bin* folder (and subfolders) of the application on the target computer:
   - Copy *C:\Program Files\Microsoft SQL Server Edition\v4.0\Desktop\System.Data.SqlServerCe.dll* **to** *\Bin*
   - Copy *C:\Program Files\Microsoft SQL Server Compact Edition\v4.0\Private\x86* **to** *\Bin\x86*
   - Copy *C:\Program Files\Microsoft SQL Server Compact Edition\v4.0\Private\amd64** **to** *\Bin\amd64*
2. In the root folder of the website, create or open a *web.config* file. (In WebMatrix 1.0, this file type is available if you click **All** in the **Choose a File Type** dialog box.)
3. Add the following element as a child of the `<configuration>` element (not inside the `<system.web>` element):

```
<system.data>
  <DbProviderFactories>
    <remove invariant="System.Data.SqlServerCe.4.0"></remove>
    <add name="Microsoft SQL Server Compact Data Provider"
      invariant="System.Data.SqlServerCe.4.0"
      Description=".NET Framework Data Provider for Microsoft SQL Server Compact"
      type="System.Data.SqlServerCe.SqlCeProviderFactory, System.Data.SqlServerCe, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=89845dcd8080cc91"/>
  </DbProviderFactories>
</system.data>
```

Issue: "Database" and "WebGrid" helpers do not work in Medium Trust in Visual Basic

If you are using Visual Basic (creating *.vbhtml* files), the `Database` and `WebGrid` helpers will not work if the application is set to use Medium Trust.

Workaround

If you use Visual Studio 2010, you can resolve this problem by installing the Service Pack 1 release. Until the final version of the SP1 release is available, you can download the Beta version of SP1 from the Microsoft Visual Studio 2010 Service Pack 1 Beta page on the Microsoft Download Center.

If this is not practical, or if you do not use Visual Studio 2010, you can temporarily set the application to use Full Trust.

Issue: "ApplicationPart" resources are externally accessible

If an assembly contains objects that derives from the `ApplicationPart` class, that assembly's resources are exposed by the `ResourceRouteHandler` class. For example, consider the following URL:

`~/r.ashx/System.Web.WebPages.Administration/Resources/AdminResources.resources`

This request downloads all of the resource strings in the *System.Web.WebPages.Administration.dll* assembly. All of the embedded resources (even those that are not intended to be served as static content) are downloaded. If the embedded resources contain sensitive information, this can represent a security risk.

Workaround

If you create an **ApplicationPart** object, make sure that the embedded resources associated with that **ApplicationPart** object's assembly do not contain sensitive information.

WebMatrix

> **NOTE**
>
> For information about installation issues for WebMatrix, see WebMatrix Installation Issues earlier in this document.

This section of the document describes known issues for the WebMatrix development environment.

Issue: Changes in the username or password of a database connection string in a web.config file are not reflected in the Databases workspace

Workaround

1. In the *web.config* file, change the database name in the connection string (for example, add "1" to it).
2. Save the *web.config* file.
3. Click **Databases** and refresh.
4. Change the database name in the connection string in the *web.config* file back to the original database name.

5. Save the *web.config* file.
6. Click **Databases** and refresh.

Issue: Folders created by WebMatrix cannot be deleted

If WebMatrix is running using elevated permissions (that is, you started WebMatrix using the **Run as Administrator** option in Windows), folders that are created by WebMatrix cannot be deleted using Windows Explorer.

Workaround

Run Windows Explorer using elevated permissions. Follow these steps:

1. In Windows, click **Start**.
2. Enter "Windows Explorer" and right-click the entry for **Windows Explorer**.
3. Click **Run as Administrator**. You can then delete the folders.

Issue: WebMatrix 1.0 is unable to perform certain tasks that require elevation

WebMatrix 1.0 is unable to perform certain tasks that require elevation, such as installing additional components in the following situations:

- On Windows Vista or Windows 7, you are logged in with an account that does not have administrative privileges and User Account Control (UAC) is disabled.
- You are using Microsoft Windows XP or Microsoft Windows Server 2003.

Workaround

Most tasks in WebMatrix 1.0 do not require administrative permission. For those that do, you can perform the operation as an administrator, or follow these steps:

- On Windows Vista or Windows 7, enable UAC.
- On Windows XP, add the user to the Administrators security group.

Issue: "Site from Web Gallery" is disabled

The **Site from Web Gallery** option is disabled if the Web Platform Installer 3.0 is not installed.

Workaround

Install the Microsoft Web Platform Installer 3.0.

Issue: Google Chrome is not available as a Run option

Google Chrome is not displayed in the list of browsers under **Run** on the **Home** tab.

Workaround

Some versions of Google Chrome do not register themselves correctly with the Default Programs feature in Windows. As a workaround, start Google Chrome, click the *Customize and control Google Chrome* menu, click *Options*, and then click *Make Google Chrome my default browser*.

Issue: The "Foreign Key" dialog box doesn't allow entering a primary key

The **Foreign Key** dialog box does not allow you to enter the primary key name from the primary key table.

Workaround

This is intentional. You do not need to enter the name of the primary key from the primary key table.

Issue: IntelliSense is not available in WebMatrix for Razor syntax, C#, or Visual Basic

IntelliSense is supported in WebMatrix for HTML and CSS. However, it is not available for other languages.

Workaround
None.

Issue: IntelliSense for HTML and CSS suggests elements that are not contextually appropriate

IntelliSense for markup in WebMatrix supports HTML using the XHTML 1.0 Transitional schema and CSS using the CSS 2.1 schema. Because IntelliSense is based on these specific schemas, certain tags, attributes, or properties might be suggested that are not appropriate for the current page or style definition. For HTML, it can also lead to unexpected suggestions in content that might be interpreted as malformed XHTML (for example, when tags are not closed). This issue might be more noticeable if the insertion point is inside an incomplete tag; in that case, IntelliSense might suggest new opening tags or offer other incorrect suggestions.

Workaround
For HTML, make sure that you are working within a well-formed, complete XHTML page. For CSS, there is no workaround.

Issue: IntelliSense is not invoked while you type

At times, IntelliSense might not be invoked as HTML or CSS is being entered in the editor. In particular, this might happen when the insertion point is directly next to another element or at the end of a file.

Workaround
Make sure that there is whitespace around the insertion point and that the insertion point is not at the end of a file. You can also invoke IntelliSense manually by pressing Ctrl+Space.

Issue: No UI is available for disabling IntelliSense

WebMatrix 1.0 provides no UI or gesture for disabling IntelliSense.

Workaround
Start WebMatrix using the following command, which includes a switch that disables IntelliSense:

```
WebMatrix.exe #ExecuteCommand# EditorIntelliSense off
```

IIS Express

IIS Express has its own readme file, which is available at the following URL:

https://go.microsoft.com/fwlink/?LinkID=207675&clcid=0x409

SQL Server Compact

SQL Server Compact has its own readme file, which is available at the following URL:

https://go.microsoft.com/fwlink/?LinkID=208545

For information about issues that involve installing SQL Server Compact as part of WebMatrix, see WebMatrix Installation Issues earlier in this document.

Installing Applications

Issue: Installing an application can take a long time if the user's My Documents folder is redirected to a network share

Workaround
None. The application might take a while to install, but will install correctly.

Publishing Applications

Issue: "Required permissions cannot be acquired" error when publishing a SQL Compact Database

WebMatrix does not fully support deploying supporting binaries for SQL Server Compact to a server that is running .NET Framework version 3.5 with a medium trust configuration.

Workaround

The preferred workaround is to install the .NET Framework 4 on the server. Alternatively, do the following:

1. Add the following elements to the `SecurityClasses` section in *Web_MediumTrust.config* file:

```
<SecurityClass Name="ReflectionPermission"
  Description="System.Security.Permissions.ReflectionPermission, mscorlib,
    Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>

<SecurityClass Name="RegistryPermission"
  Description="System.Security.Permissions.RegistryPermission, mscorlib,
  Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
```

2. Create a new permission set in the *Web_MediumTrust.config* file with the following required permissions:

```
<PermissionSet class="NamedPermission" version="1" Name="SqlCe_Trust">
  <IPermission class="SecurityPermission" version="1" Flags="UnmanagedCode,SkipVerification"/>
  <IPermission class="EnvironmentPermission" version="1" Read="PROCESSOR_ARCHITECTURE"/>
  <IPermission class="RegistryPermission" version="1" Read="HKEY_LOCAL_MACHINE\" />
</PermissionSet>
```

3. Apply the permission set to SQL Server Compact by putting the following elements in the *Web_MediumTrust.config* file:

```
<CodeGroup class="UnionCodeGroup" version="1" PermissionSetName="ASP.Net">
  <IMembershipCondition class="UrlMembershipCondition" version="1" Url="$AppDirUrl$/*"/>
  <CodeGroup class="UnionCodeGroup"
      version="1"
      PermissionSetName="SqlCe_Trust"
      Name="SqlCe_Strong_Name"
      Description="This code group grants code signed with the SQLCE strong name SqlCe_Trust. ">
    <IMembershipCondition class="StrongNameMembershipCondition"
      version="1"

PublicKeyBlob="002400000480000094000000060200000024000052534131000400000100010027273 6ad6e5f9586bac2d
531eabc3acc666c2f8ec879fa94f8f7b0327d2ff2ed523448f83c3d5c5dd2dfc7bc99c5286b2c125117bf5cbe242b9d41750
732b2bdffe649c6efb8e5526d526fdd130095ecdb7bf210809c6cdad8824faa9ac0310ac3cba2aa0523567b2dfa7fe250b30
facbd62d4ec99b94ac47c7d3b28f1f6e4c8" Name="System.Data.SqlServerCe" />
  </CodeGroup>
</CodeGroup>
```

Issue: Gallery and PhpBB web applications display a "Service is unavailable" error after publishing

Under some circumstances, publishing an application causes a "service is unavailable" error.

Workaround

In WebMatrix, add a backslash () to the end of the server name in the **Publish Settings** window and then publish the application again.

Issue: Moodle website layout and links are broken after publishing

After you publish a Moodle application, the application does not work correctly.

Workaround

In WebMatrix, add a slash (/) to the end of the **Site Name** field in the **Publish Settings** window and then

publish the application again.

Issue: Publishing nopCommerce fails with a database error

Publishing nopCommerce fails and reports a database error like "Insert into the nop_log table failed."

Workaround

1. In WebMatrix, click **Run** to launch nopCommerce locally.
2. Log into the administration page.
3. Click the **System** menu.
4. Click the **Log** option.
5. Click the **Clear Log** button.
6. Publish nopCommerce again.

Issue: Silverstripe CMS displays a "HTTP 500 PHP FCGI Error" when you download a published site

Workaround
After you click **Download published site**, skip `silverstripe-cache/manifest_main` in **Publish Preview**. This file is used for caching purposes and is specific to each computer.

Issue: Subtext displays "Server Error in '/' Application" when you download a published site

Workaround
Open the site's *web.config* file and replace the user ID and password in the database connection string with the SQL Server administrator credentials (the "sa" credentials).

Alternatively, follow these steps in order to give the user account you are logged in with `db_owner` permissions:

1. Install SQL Server Management Studio using the Web Platform Installer.
2. Connect to the local SQL Server Express instance (by default, `.\SQLEXPRESS`).
3. Click **Databases** > *[localSubtextDatabase]* > **Security** > **Users** > *[localSubtextUser]* (default is `subtextuser`], right-click, and click **Properties**.
4. Select **db_owner** in the role membership section.

Issue: Site might not work after publishing if the "Destination URL" field is not prefixed with http:// or https://

In the **Publishing Settings** dialog box, if the destination URL does not begin with `http://` or `https://`, the site might not work after deployment.

Workaround
Make sure that before you publish a site, the destination URL in the **Publish Settings** dialog box starts with `http://` or `https://`.

Issue: Publishing a MySQL database fails with the error "Failed to publish the database. This can happen if the remote database cannot run the script."

The error can occur for a number of reasons. One reason you can see this error is if the database script contains a single quotation character (') and the destination MySQL database's default character set is not to UTF-8.

Workaround
Set the default character set for the remote MySQL database to UTF-8.

Issue: Some links are not visible in DotNetNuke after publishing or downloading the site

If you publish or download a DotNetNuke site, you might need to clear the cache to get the new links to appear on the site.

Workaround

1. Log in as "Host".
2. Go to the host menu and select **Host Settings**.
3. Scroll down and under **Advanced Settings**, expand **Performance Settings**.
4. Click the **Clear Cache** link for pages.
5. Go to the bottom of the page and restart the application.

Issue: Some links in AtomSite are broken after you download a published site

Workaround

In the *service.config* file, *users.config* file, and all *.xml* files, replace the URL string (for example, `http://myhost.com/atomsite`) with the local one (for example, `http://localhost:1239`).

Issue: MySQL-based applications like WordPress fail to publish and report a database error

By default, WebMatrix installs MySQL with the UTF-8 character set. If you install MySQL on your own, and the character set is not UTF-8 (for example, it is Latin1), the publish process for databases might fail.

Workaround

1. Change the character set for MySQL to UTF-8. (For details, see Server Character Set and Collation on the MySQL website.)
2. Reinstall the application.
3. Republish the application.

Issue: "Download published site" fails for applications that have browser-based setup

Some applications (for example, Kentico CMS) require you to launch them in the browser in order to perform post-installation setup such as creating a database. If you publish an application like this without completing the browser-based setup, attempting to download the same site from a remote server will fail.

Workaround

Finish browser-based setup before publishing the site.

Issue: "Download published site" fails with a database error for DotNetNuke and Kooboo CMS

If you try to download an application from a server and you have administrator credentials in the database connection string in the **Publish Settings** dialog, you might see the following error in the publish log:

```
1:28:29 PM: An error occurred during execution of the database script. The approximate location of the
error was between lines '3260' and '3262' of the script. The verbose log may have more information about
the error. The command started with:

1:28:29 PM: CREATE LOGIN [MACHINE\Administrator] FROM WINDOW

1:28:29 PM: https://go.microsoft.com/fwlink/?LinkId=178587.
```

Workaround

If practical, republish the site (or have it published) using non-administrator credentials for the database.

For More Information

For more information about WebMatrix 1.0, see the following websites:

- IIS.net
- ASP.NET
- Microsoft.com/web

© 2011 Microsoft Corporation. All Rights Reserved. Terms of Use.

Web Matrix and ASP.NET Web Pages (Razor) Beta 3 Release Readme

2/20/2020 • 23 minutes to read • Edit Online

> Web Matrix and ASP.NET Web Pages (Razor) Beta 3 Release Readme

9 November 2010

Contents

- Overview
- Installation
- New Features, Changes, and Known Issues in the Beta 3 release
  - WebMatrix Installation Issues
  - ASP.NET Web Pages
  - SQL Server Compact
  - Installing Applications
  - Publishing Applications
  - Other Issues
- For More Information

Overview

> Microsoft WebMatrix Beta is a free web development stack that installs in minutes. It integrates a web server with database and programming frameworks to create a single, integrated experience. You can use WebMatrix Beta to streamline the way you code, test, and publish your own ASP.NET or PHP website, or you can use WebMatrix Beta to start a new website using popular open-source apps like DotNetNuke, Umbraco, WordPress, or Joomla. WebMatrix Beta uses the same powerful web server, database engine, and frameworks environment that will run your website on the internet, which makes the transition from development to production smooth and seamless.

Installation

> To install WebMatrix Beta 3, you use Microsoft Web Platform Installer 3.0. After you've installed the Web Platform Installer, you can use it to install WebMatrix Beta 3.
>
> If you have problems during installation, refer to Troubleshooting Problems with Microsoft Web Platform Installer.

Instructions for Publishing Applications

> See Step-by-Step Instructions for Publishing Applications

New Features, Changes, andKnown Issues

WebMatrix Beta 3 Installation

Issue: WebMatrix Beta 3 is only available on platforms that support Microsoft .NET Framework 4

The .NET Framework version 4 is required for WebMatrix Beta. In certain cases, the WebMatrix Beta installer will let you try to install on a platform that is not part of the supported configuration set. In particular, Windows Vista without the SP1 update will let you begin the installation of WebMatrix Beta, but the .NET Framework 4 component will fail and block your installation.

Workaround

Install on a supported platform, which includes:

- Windows 7
- Windows Server 2008
- Windows Server 2008 R2
- Windows Vista SP1 or later
- Windows XP SP3
- Windows Server 2003 SP2

Issue: Cannot install WebMatrix Beta 3 if Microsoft Visual Studio 2008 is installed without Microsoft Visual Studio 2008 SP1

Workaround

Install Microsoft Visual Studio 2008 SP1 from the Microsoft Download Center.

Issue: Some assemblies for SQL Server Compact 4.0 are not installed in the GAC

The managed assemblies for SQL Server Compact 4.0 are not placed in the global assembly cache (GAC) when you install SQL Server Compact 4.0 on a 64-bit computer and the computer has only the .NET Framework 3.5 SP1 Client Profile installed. The managed assemblies that are not installed in the GAC are:

- *System.Data.SqlServerCe.dll* (ADO.NET provider)
- *System.Data.SqlServerCe.Entity.dll* (ADO.NET Entity Framework)

Workaround

Uninstall SQL Server Compact 4.0. Download and install the full version of .NET Framework 3.5 SP1 from the following location:

Microsoft .NET Framework 3.5 Service pack 1 (Full Package)

Then reinstall SQL Server Compact 4.0.

Issue: Cannot uninstall SQL Server Compact using the command line

Uninstallation of SQL Server Compact using command-line options does not work in this release.

Workaround

Use *Programs and Features* in the Windows Control Panel to uninstall Microsoft SQL Server Compact 4.0.

ASP.NET Web Pages

This section of the document describes new features, changes, and known issues with the Beta 3 release of ASP.NET Web Pages with Razor syntax.

- New features
- Changes
- Issues

New Features in Beta 3 for ASP.NET Web Pages with Razor Syntax

New: "Html.Raw" method renders unencoded markup

The new `Html.Raw` method lets you render HTML markup as markup instead of rendering encoded output. (By default, ASP.NET Razor encodes strings before rendering them.) The syntax is:

`Html.Raw(value)`

The following example shows how to use `Html.Raw`:

```
@* Inserts literal markup into the page as specified in the value string. *@
@Html.Raw("<div>Hello <em>world</em>!</div>")
```

Changes in Beta 3 for ASP.NET Web Pages with Razor Syntax

Change: "HrefAttribute" method removed

The `HrefAttribute` method of the `WebPage` class has been removed. This helper was used to encode unsafe characters in URLs. It is no longer required because ASP.NET Razor automatically encodes strings. (Use the new `Html.Raw` method to render unencoded strings.)

Change: Syntax for declarative "@helper" helpers changed

In the Beta 3 release, ASP.NET changes how it parses helpers that are created using the `@helper` syntax. In essence, the `@helper` syntax is now parsed as a code block instead of as a block of markup that can include code. Therefore, code inside the helper does not need to be enclosed in `@{ }` blocks. Conversely, markup inside the helper has to be explicitly included in HTML elements or in ASP.NET Razor `<text></text>` tags.

For example, the following `@helper` syntax works in the Beta 3 release:

```
@helper ThumbnailLink(string imagePath, int width, int height) {
    @{
        string newFileName = "";
        WebImage thumbnail = new WebImage(imagePath);
        if(thumbnail != null){
           thumbnail.Resize(width, height, true, true);
           newFileName = @"~\thumb_" + Path.GetFileName(thumbnail.FileName);
           thumbnail.Save(newFileName);
        }
    }
    <a href="@Href(imagePath)"><img src="@Href(newFileName)" /></a>
}
```

In the Beta 3 release, this helper must be changed to look like the following example:

```
@helper ThumbnailLink(string imagePath, int width, int height) {
    string newFileName = "";
    WebImage thumbnail = new WebImage(imagePath);
    if(thumbnail != null){
       thumbnail.Resize(width, height, true, true);
       newFileName = @"~\thumb_" + Path.GetFileName(thumbnail.FileName);
       thumbnail.Save(newFileName);
    }
    <a href="@Href(imagePath)"><img src="@Href(newFileName)" /></a>
}
```

Notice that the `@{ }` characters around the initial code in the helper is no longer used. This is because the contents of the helpers are treated as a code block by default. The helper renders markup, which starts with the opening `<a>` tag. If the helper must render plain text or tags that do not include a closing tag (for example,

`<meta>` tags), the content to be rendered must be in `<text></text>` tags.

Change: "WebPageContext.HttpContext" removed

The `WebPageContext.HttpContext` property has been removed. Use `HttpContext.Current` instead. (The `WebPageContext.HttpContext` property simply wrapped this.)

Change: "Facebook" helper moved to new package

The `Facebook` helper has been moved to the *Facebook.Helper* library, which includes the `Facebook` helper and additional functionality. You must install this library as a separate package, as described in "Installing Helpers with Package Manager" in the tutorial Getting Started with ASP.NET Pages.

Change: Membership, Role, and Security types moves to new assembly

The following types were moved to the `WebMatrix.WebData` assembly:

- `ExtendedMembershipProvider`
- `SimpleMembershipProvider`
- `SimpleRoleProvider`
- `WebSecurity`

Change: "TagBuilder" class moved to System.Web.WebPages.dll assembly

The `TagBuilde` r class has been moved to the System.Web.WebPages.dll assembly. Previously, this was in an assembly that was part of ASP.NET MVC. This change means that you do not have to install ASP.NET MVC in order to use the `TagBuilder` class.

However, the class is still in the `System.Web.Mvc` namespace. In order to use the `TagBuilder` class (for example, in a custom ASP.NET Razor helper), you must reference the namespace (for example, by adding `@using System.Web.Mvc` to your code).

Change: Request validation syntax changed; "Validation" class removed

In the Beta 3 release, to disable validation for an individual field or set of fields, you can call the `Validation.Exclude` method, passing in the name or names of the fields to exclude from validation. A new syntax is available in the Beta 3 release for bypassing validation. The `Validation` method used in Beta 3 has been removed.

> **NOTE**
>
> If you do not disable request validation, if users try to upload HTML markup (for example, by using a rich text editor on a page), the website will report an error like *A potentially dangerous Request.Form value was detected from the client* and the user input is not accepted. If you disable request validation, you must manually check user input to make sure that it does not contain potentially dangerous markup or script using something like the Microsoft Anti-Cross Site Scripting Library V4.0.

To disable automatic request validation, call the `Request.Unvalidated` method, passing it the name of the field or other post object that you want to bypass request validation for. You can use this method to bypass validation for any items in the `Form`, `QueryString`, `Cookies`, and `ServerVariables` collections. The following examples show how to use the `Unvalidated` method:

```
Request["userInput"]; // Validated
Request.Unvalidated("userInput"); // Validation bypassed
Request.Unvalidated().Form["userInput"]; // Validation bypassed

Request.QueryString["userPreference"]; // Validated
Request.Unvalidated().QueryString["userPreference"]; // Validation bypassed
```

Known Issues for ASP.NET Web Pages with Razor Syntax

Issue: Unexpected behavior when using a custom user table for membership

To initialize the membership provider for an ASP.NET Razor website, you call the `WebSecurity.InitializeDatabaseConnection` method. (In WebMatrix, the Starter Site template includes a call to this method in the *_AppStart.cshtml* file.) If the `autoCreateTables` parameter of this method is set to true (by default, it is set to true in the Starter Site template), and if an unrecognized table name is passed to the method (the second parameter), the method does not throw an error. Instead, it automatically creates the table.

This can be a problem if you intend to use a custom user table for membership but pass the wrong table name to the `WebSecurity.InitializeDatabaseConnection` method. Because the method does not by default raise an error if the table you specify does not exist, and because it instead creates a new table, the application can appear to be working. However, application code that relies on your custom user table (and on fields in it) can eventually fail with unexpected errors.

Workaround

Make sure that the name passed in the `InitializeDatabaseConnection` method matches the user profile table in the membership database, or make sure that the `autoCreateTables` parameter is set to false.

Issue: "Failed to generate a user instance of SQL Server" error

If a WebMatrix Web application uses SQL Server Express and is running IIS 7.5 on Windows 7 or Windows Server 2008 R2, you might see an error that indicates that SQL Server cannot retrieve the user's local application path at run time.

Workaround Make sure that the Windows account that the application runs under (typically NETWORK SERVICE) has read/write permissions for root folders of the application and for subfolders such as *App_Data*. More detailed information is available in the KnowledgeBase article Problems with SQL Server Express user instancing and ASP.net Web Application Projects.

Issue: In Visual Studio, namespaces for custom assemblies (DLLs) are not imported automatically

If you use custom assemblies in a project in Visual Studio, the namespaces declared in those assemblies are not automatically imported at design time. As a result, references to custom types might not be recognized at design time and are marked as not recognized in Visual Studio (using a "squiggle"). This problem occurs only at design time in Visual Studio; the application itself runs properly.

Workaround

Include a `using` statement (`imports` in Visual Basic) that references the entities that are not recognized at design time.

Issue: Visual Studio IntelliSense and project templates available only in ASP.NET MVC version 3

Installing ASP.NET Web Pages does not also install tools for Visual Studio such as IntelliSense and project templates for ASP.NET Web Pages applications.

Workaround To use IntelliSense and project templates for ASP.NET Web Pages applications in Visual Studio, install ASP.NET MVC 3 RC either through the Web Platform Installer or the stand-alone installer.

Issue: "<helper> class cannot be found" error

After you upgrade to Beta 3, you might see an error that a helper class (for example, the `Facebook` class) cannot not be found. Starting in Beta 2 and continuing in Beta 3, helpers have been moved to packages that you must explicitly install. Existing sites are not upgraded to include these packages; this includes sites in the *\My Documents\IISExpress* or *\My Documents\My Web Sites* folders. In particular, you will see this error if you use the default site in *My Sites* (WebSite1), which includes a reference to the `Twitter` helper.

Workaround
Comment out calls to any helpers in the site, run the *_Admin* page, and install the package or packages that include the helpers that you want to use. After you've installed the package, you can uncomment the lines that reference helpers.

Issue: Deploying Beta 3 ASP.NET Razor assemblies to the Bin folder might not work on hosting sites

If you deploy an ASP.NET Web Pages website to a hosting site, and if you deploy the ASP.NET Razor Beta 3 assemblies to the site's *Bin* folder, you might experience errors, including the following:

```
Could not load type 'Microsoft.Web.Infrastructure.DynamicModuleHelper.DynamicModuleUtility' from assembly
'Microsoft.Web.Infrastructure, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35'.
```

This can happen if the hosting provider has installed the ASP.NET Web Pages Beta 1 assemblies into the server's global application cache (GAC). Assemblies in the GAC get precedence over assemblies installed locally in the *Bin* folder.

Workaround Contact your hosting provider to confirm that the errors you are seeing are due to a conflict between the provider's versions of the assemblies and yours. If so, request that the hosting provider update the assemblies in the server's GAC.

Issue: Reading feeds or other external data via a proxy server

If the server running the site is behind a proxy server, you might need to configure proxy information in the *Web.config* file in order to be able to read information that comes from outside your site. For example, if you use the `ReCaptcha` helper, the helper communicates with the reCAPTCHA service, but might be blocked by your proxy server. Similarly, feeds that are used in ASP.NET Web Pages, such as the feed used by the package manager, might require proxy configuration.

If you experience problems in working with an external service or working with the package feed, put the following elements into your application's root *Web.config* file:

```
<system.net>
  <defaultProxy>
    <proxy autoDetect="False"
       proxyaddress="[proxy URL]"/>
  </defaultProxy>
</system.net>
```

For more information about configuring a proxy server, see <proxy> Element (Network Settings) on the MSDN Web site.

Issue: "Microsoft.Web.Infrastructure.dll cannot be loaded" error

If you previously installed the Beta 1 version of ASP.NET Web Pages with Razor syntax and then install the Beta 3 version, all appropriate assemblies are installed in the GAC except *Microsoft.Web.Infrastructure.dll*. As a consequence, when you run ASP.NET Razor pages, you see an error that indicates that *Microsoft.Web.Infrastructure.dll* could not be loaded.

This issue does not occur if you loaded the Beta 3 release on a clean computer.

Workaround

In Control Panel, uninstall ASP.NET Web Pages. Then reinstall the Beta 3 release.

Issue: Uninstalling the .NET Framework version 4 disables ASP.NET Web Pages with Razor Syntax

If you uninstall the .NET Framework version 4 and then reinstall it, ASP.NET Web Pages with Razor syntax is disabled. Pages with the *.cshtml* extension do not run correctly. ASP.NET Web Pages registers an assembly in the machine root *Web.config* file, and removing the .NET Framework removes that file. Reinstalling the .NET Framework installs a new version of the configuration file, but does not add the reference for the ASP.NET Web Pages assembly.

Workaround After reinstalling the .NET Framework, reinstall ASP.NET Web Pages with Razor syntax. This adds the following element to the *Web.config* file in the machine root, which is typically in the following location:

`C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config (32-bit)`

`C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config (64-bit)`

```
<compilation>
  <assemblies>
    <add assembly="Microsoft.WebPages.Configuration, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />

    <-- Additional assemblies here. -->

  </assemblies>
</compilation>
```

Issue: Applications previously deployed with ASP.NET assemblies in the Bin folder experience errors

During deployment, copies of the ASP.NET Web Pages assemblies (for example, *Microsoft.WebPages.dll*) to the *Bin* folder of the website on the server. (This might have happened automatically during deployment or because the developer explicitly copied the assemblies.) However, when the Beta 3 release is installed, errors occurs, such as errors that certain types cannot be found. This occurs because a number of ASP.NET Web Pages types were moved into different namespaces for the Beta 3 release.

Workaround

Clear the *Bin* folder of the deployed application, copy the new assemblies to the folder (or redeploy the application), and then restart the application.

Issue: Extensionless URLs do not find .cshtml/.vbhtml files on IIS 7 or IIS 7.5

On IIS 7 or IIS 7.5, requests with a URL like the following are not able to find pages that have the *.cshtml* or *.vbhtml* extension:

`http://www.example.com/ExampleSite/ExampleFile`

The issue arises because URL rewriting is not enabled by default for IIS 7 or IIS 7.5. The likeliest scenario is that you do not see the problem when testing locally using IIS Express, but you experience it when you deploy your website to a hosting website.

Workaround

- If you have control over the server computer, on the server computer install the update that is described in A update is available that enables certain IIS 7.0 or IIS 7.5 handlers to handle requests whose URLs do not end with a period.

- If you do not have control over the server computer (for example, you are deploying to a hosting website), add the following to the website's *Web.config* file:

```
<system.webServer>
  <modules runAllManagedModulesForAllRequests="true"/>
</system.webServer>
```

Issue: Using Web Application Project or ASP.NET MVC and ASP.NET Web pages in the same application

If you were using ASP.NET Web Pages in a Web Application project or ASP.NET MVC application, you might see an error that *WebPageHttpApplication* cannot be found.

Workaround

If you get this error, change the base class from which the application derives. In the *Global.asax* file, change the following line:

```
public class MvcApplication :  WebPageHttpApplication { ... }
```

To this:

```
public class MvcApplication :  HttpApplication { ... }
```

This in effect reverses a change that was introduced for the Beta 1 release of ASP.NET Web Pages with Razor syntax.

Issue: Deploying an application to a computer that does not have SQL Server Compact installed

Applications that include SQL Server Compact databases can run on a computer where SQL Server Compact is not installed. Microsoft WebMatrix Beta 3 automatically copies these binaries for you and performs the appropriate *Web.config* file transforms.

Workaround If you need to copy these files and make the *Web.config* file changes manually, do the following :

1. Copy the database engine assemblies to the *Bin* folder (and subfolders) of the application on the target computer:
   - Copy *C:\Program Files\Microsoft SQL Server Compact Edition\v4.0\Desktop\System.Data.SqlServerCe.dll* to *\Bin*
   - Copy *C:\Program Files\Microsoft SQL Server Compact Edition\v4.0\Private\x86** to *\Bin\x86*
   - Copy *C:\Program Files\Microsoft SQL Server Compact Edition\v4.0\Private\amd64** to *\Bin\amd64*
2. In the root folder of the website, create or open a *Web.config* file. (In WebMatrix Beta 3, this file type is available if you click **All** in the **Choose a File Type** dialog box.)
3. Add the following element as a child of the **<configuration>** element (not inside the **<system.web>** element):

```
<system.data>
  <DbProviderFactories>
    <remove invariant="System.Data.SqlServerCe.4.0"></remove>
    <add name="Microsoft SQL Server Compact Data Provider"
      invariant="System.Data.SqlServerCe.4.0"
      Description=".NET Framework Data Provider for Microsoft SQL Server Compact"
      type="System.Data.SqlServerCe.SqlCeProviderFactory, System.Data.SqlServerCe, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=89845dcd8080cc91"/>
  </DbProviderFactories>
</system.data>
```

Issue: Database and WebGrid helpers do not work in Medium Trust in Visual Basic

If you are using Visual Basic (creating *.vbhtml* files), the `Database` and `WebGrid` helpers will not work if the application is set to use Medium Trust.

Workaround

Temporarily set the application to use Full Trust.

SQL Server Compact

Issue: "Encrypt" property is not recognized

SQL Server Compact 4.0 does not recognize the `Encrypt` property of the `SqlCeConnection` class. You should not use this property to encrypt database files. The `Encrypt` property was deprecated in SQL Server Compact 3.5 release and was retained only for backward compatibility.

Workaround

Use the `Encryption Mode` property of the `SqlCeConnection` class to encrypt SQL Server Compact 4.0 database files. The following example shows how to create an encrypted SQL Server Compact 4.0 database using the `Encryption Mode` property:

```
SqlCeEngine engine = new SqlCeEngine("Data Source=Northwind.sdf;encryption mode=platform default;Password=
<enterStrongPasswordHere>;");
engine.CreateDatabase();
```

```
Dim engine As SqlCeEngine = New SqlCeEngine("Data Source=Northwind.sdf;encryption mode=platform
default;Password=<enterStrongPasswordHere>;")
engine.CreateDatabase()
```

To change the encryption mode of an existing SQL Server Compact 4.0 database, do the following:

```
SqlCeEngine engine = new SqlCeEngine("Data Source=Northwind.sdf;Password=<enterStrongPasswordHere>;");
engine.Compact("Data Source=Northwind.sdf;encryption mode=ppc2003 compatibility;Password=
<enterStrongPasswordHere>;");
```

```
Dim engine As SqlCeEngine = New SqlCeEngine("Data Source=Northwind.sdf;Password=
<enterStrongPasswordHere>;")
engine.Compact("Data Source=Northwind.sdf;encryption mode=ppc2003 compatibility;Password=
<enterStrongPasswordHere>;")
```

To encrypt an unencrypted SQL Server Compact 4.0 database, do the following:

```
SqlCeEngine engine = new SqlCeEngine("Data Source=Northwind.sdf");
engine.Compact("Data Source=Northwind.sdf;encryption mode=platform default;Password=
<enterStrongPasswordHere>;");
```

```
Dim engine As SqlCeEngine = New SqlCeEngine("Data Source=Northwind.sdf;")
engine.Compact("Data Source=Northwind.sdf;encryption mode=platform default;Password=
<enterStrongPasswordHere>;")
```

Issue: Microsoft Visual C++ 2008 runtime libraries are required

The native DLLs of SQL Server Compact 4.0 need the Microsoft Visual C++ 2008 Runtime Libraries (x86, IA64, and x64), Service Pack 1.

Workaround

Install the .NET Framework 3.5 SP1. This also installs the Visual C++ 2008 Runtime Libraries SP1. You can download the libraries from the following location:

Microsoft Visual C++ 2008 Service Pack 1 Redistributable Package ATL Security Update

> **NOTE**
>
> Note that installing the .NET Framework 2.0, 3.0, or 4 does *not* install the Visual C++ 2008 Runtime Libraries SP1.

Issue: If SQL Server Compact is installed prior to installing .NET Framework on the computer, its provider invariant name is not registered in the .NET Framework machine.config file

SQL Server Compact can be installed on a machine that does not have .NET Framework installed because SQL Server Compact does require the .NET framework. If neither .NET Framework version 3.5 nor 4 is installed before you install SQL Server Compact, the SQL Server Compact Setup does not register its provider invariant name in the *machine.config* file. Any application that relies on the SQL Server Compact entry in the *machine.config* file will fail. The invariant name registration entry in *machine.config* looks like the following example:

```
<system.data>
  <DbProviderFactories>
    <remove invariant="System.Data.SqlServerCe.4.0"></remove>
    <add
name="Microsoft SQL Server Compact Data Provider"
invariant="System.Data.SqlServerCe.4.0"
        Description=".NET Framework Data Provider for Microsoft SQL Server Compact"
        type="System.Data.SqlServerCe.SqlCeProviderFactory, System.Data.SqlServerCe,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91"/>
    </DbProviderFactories>
</system.data>
```

Workaround

Uninstall SQL Server Compact 4.0 CTP1. Download and install the full versions of the .NET Framework from the following location:

Microsoft .NET Framework 3.5 Service pack 1 (Full Package)
Microsoft .NET Framework 4.0 Release (Full Package)

Then reinstall SQL Server Compact 4.0 CTP1.

Installing Applications

Issue: Installing an application can take a long time if the user's My Documents folder is redirected to a network share

Workaround
None. The application might take a while to install, but will install correctly.

Publishing Applications

Issue: Site might not work after publishing if the "Destination URL" field is not prefixed with http:// or https://

In the **Publishing Settings** dialog box, if the destination URL does not begin with `http://` or `https://`, the site might not work after deployment.

Workaround
Make sure that before you publish a site, the destination URL in the **Publish Settings** dialog box starts with `http://` or `https://`.

Issue: Publishing a MySQL database fails with the error "Failed to publish the database. This can happen if the remote database cannot run the script."

The error can occur for a number of reasons. One reason you can see this error is if the database script contains a single quotation character (') and the destination MySQL database's default character set is not to UTF-8.

Workaround
Set the default character set for the remote MySQL database to UTF-8.

Other Issues

Issue: Search/Filter does not work in Reports for Group By: Issue Type

When you run a report for a site, if you enter text in the *Filter by URL* box and click *Search*, nothing happens. This is because this control is not functional while the *Group By* state of the report is set to *Issue Type*, which is the default.

Workaround In the *Group By* tab of ribbon, click *URL* to group the entries by their source URL. The text box and button to filter the entries are functional while in this state.

Issue: WCF applications fail to run with IIS Express

Browsing to a WCF application results in an error like the following one:

Could not load file or assembly 'Microsoft.Web.Administration, Version=7.0.0.0, Culture=neutral,PublicKeyToken=31bf3856ad364e35' or one of its dependencies. The system cannot find the file specified.

This occurs because IIS Express Beta release doesn't support WCF by default.

Workaround Use any one of the following workarounds (workaround #2 requires Microsoft Windows Vista or higher):

1. Copy the *Microsoft.Web.dll* and *Microsoft.Web.Administration.dll* assemblies from the WebMatrix installation location to the *bin* directory of the WCF application. By default, WebMatrix is installed in the *Microsoft WebMatrix* subfolder under the system's *Program Files* folder.

2. On Microsoft Windows Vista or higher, create a symlink to the assemblies in the *bin* directory using the following commands. (This approach has the advantage that it does not create a copy of the assemblies.)

```
mklink Microsoft.Web.Administration.dll "c:\Program Files (x86)\Microsoft
WebMatrix\Microsoft.Web.Administration.dll"
mklink Microsoft.Web.dll "c:\Program Files (x86)\Microsoft WebMatrix\Microsoft.Web.dll"
```

3. Install the two assemblies in the GAC. From an elevated prompt, run the following commands:

```
gacutil /i  "c:\Program Files (x86)\Microsoft WebMatrix\Microsoft.Web.Administration.dll"
gacutil /i "c:\Program Files (x86)\Microsoft WebMatrix\Microsoft.Web.dll"
```

Issue: WebMatrix Beta 3 is unable to perform certain tasks that require elevation

WebMatrix Beta 3 is unable to perform certain tasks that require elevation, such as installing additional components in the following situations:

- On Windows Vista or Windows 7, you are logged in with an account that does not have administrative privileges and User Account Control (UAC) is disabled.
- You are using Microsoft Windows XP or Microsoft Windows Server 2003.

Workaround

Most tasks in WebMatrix Beta 3 do not require administrative permission. For those that do, you can perform the operation as an administrator, or follow these steps:

- On Windows Vista or Windows 7, enable UAC.
- On Windows XP, add the user to the Administrators security group.

Issue: "Site from Web Gallery" is disabled

The **Site from Web Gallery** option is disabled if the Web Platform Installer 3.0 is not installed.

Workaround

Install the Microsoft Web Platform Installer 3.0.

Issue: On Windows Server 2003, IIS Express does not start for a non-administrative user

On Windows Server 2003, when you launch a page or start IIS Express, IIS Express does not start. For Web pages, an error is displayed that indicates that the application has been started by a non-administrative user.

Workaround

Start WebMatrix Beta 3 as an administrative user. For more details, see the following KnowledgeBase article:

An application that is started by a non-administrative user cannot listen to the HTTP traffic of the computer on which the application is running in Windows Vista, Windows Server 2003, or Windows XP.

Issue: Google Chrome is not available as a Run option

Google Chrome is not displayed in the list of browsers under **Run** on the **Home** tab.

Workaround

Some versions of Google Chrome do not register themselves correctly with the Default Programs feature in Windows. As a workaround, start Google Chrome, click the *Customize and control Google Chrome* menu, click *Options*, and then click *Make Google Chrome my default browser*.

Issue: The "Foreign Key" dialog box doesn't allow entering a primary key

The **Foreign Key** dialog box does not allow you to enter the primary key name from the primary key table.

Workaround

This is intentional. You do not need to enter the name of the primary key from the primary key table.

Issue: The "Relationships" button is disabled

The **Relationships** button under the **Table** tab in the **Databases** workspace is disabled for SQL Server Compact databases.

Workaround

None. SQL Server Compact does not support relationships between tables.

Issue: Parameterized SQL queries throw exceptions

In SQL Server Compact 4.0, if you do not specify a data type such as `SqlDbType` or `DbType` for parameters in parameterized queries, an exception is thrown when the query runs.

Workaround

Explicitly set the data type for parameters such as `SqlDbType` or `DbType`. This is critical in the case of BLOB data types (`image` and `ntext`). Use code like the following:

```
SqlCeEngine engine = new SqlCeEngine(connString);
engine.CreateDatabase();
engine.Dispose();
SqlCeConnection conn = new SqlCeConnection(connString);
conn.Open();
SqlCeCommand cmd = conn.CreateCommand();
cmd.CommandText = "CREATE TABLE BlobTable(name nvarchar(128), blob ntext);";
cmd.ExecuteNonQuery();
cmd.CommandText = "INSERT INTO BlobTable(name, blob) VALUES (@name, @blob);";
SqlCeParameter paramName = cmd.Parameters.Add("name", SqlDbType.NVarChar, 128);
SqlCeParameter paramBlob = cmd.Parameters.Add("blob", SqlDbType.NText);
paramName.Value = "Name1";
paramBlob.Value = "Name1".PadLeft(4001);
cmd.ExecuteNonQuery();
```

```
Dim engine As SqlCeEngine = New SqlCeEngine(connString)
engine.CreateDatabase()
engine.Dispose()
Dim conn As SqlCeConnection = New SqlCeConnection(connString)
conn.Open()
Dim cmd As SqlCeCommand = conn.CreateCommand()
cmd.CommandText = "CREATE TABLE BlobTable(name nvarchar(128), blob ntext);"
cmd.ExecuteNonQuery()
cmd.CommandText = "INSERT INTO BlobTable(name, blob) VALUES (@name, @blob);"
Dim paramName As SqlCeParameter
Dim paramBlob As SqlCeParameterparamName = cmd.Parameters.Add("name", SqlDbType.NVarChar, 128)
paramName.Value = "Name1"
paramBlob = cmd.Parameters.Add("blob", SqlDbType.NText)
paramBlob.Value = "Name1".PadLeft(4001)
cmd.ExecuteNonQuery()
```

For More Information

For more information about WebMatrix Beta 3, see the following websites:

- IIS.net
- ASP.NET
- Microsoft.com/web

© 2010 Microsoft Corporation. All Rights Reserved. Terms of Use.

WebMatrix Content Guide

4/10/2020 • 4 minutes to read • Edit Online

> **NOTE**
>
> WebMatrix is no longer recommended as an integrated development environment for ASP.NET Web Pages. Use Visual Studio or Visual Studio Code.

WebMatrix is a set of tools that provides an easy way to build Web sites for Microsoft Windows. It includes IIS Express (a development web server), ASP.NET (a web programming framework), and SQL Server Compact (an embedded database). To see a quick tour of WebMatrix, see WebMatrix Overview. To review the top WebMatrix features, see WebMatrix Features.

SCENARIO	CONTENT
Get started with WebMatrix	Introduction to WebMatrix (video) Getting Started with WebMatrix and ASP.NET Web Pages Start from Scratch (video) Learn how to use ASP.NET Razor code to write a custom website. Your First Website Using WebMatrix Start with Web Apps (video) Use a free open-source application to create your own site.
Install components (including WebMatrix)	Using the Microsoft Web Platform Installer Web Platform Installer 2.0 Walkthrough Web Platform Installer Resources
Work with the WebMatrix UI	WebMatrix Site Workspace Help WebMatrix Files Workspace Help WebMatrix Databases Workspace Help WebMatrix Reports Workspace Help
Work with code	Introduction to ASP.NET Web Programming Using the Razor Syntax Simple to code (video) Programming ASP.NET Web Pages with Visual Studio ASP.NET Web Pages Visual Basic ASP.NET Quick API Reference
Create a consistent website look	Creating a Consistent Look - Creating Reusable Blocks of Content - Creating a Consistent Look Using Layout Pages - Designing Layout Pages That Have Multiple Content Sections - Making Content Sections Optional - Passing Data to Layout Pages - Creating and Using a Basic Helper
Work with HTML forms in web pages	Working with Forms - Creating a Simple HTML Form - Reading User Input from a Form - Validating User Input - Restoring Form Values After Postbacks

SCENARIO	CONTENT
Display and manage data	Database made simple (video) Connecting to a SQL Server or MySQL Database in WebMatrix Working with Data (programmatically) - Introduction to Databases - Creating a database - Adding data to the database - Displaying data from a database - Inserting data in a database - Updating data in a database - Deleting data in a database - Connecting to a Database - Understanding SQL - Migrating a Database to SQL Server Displaying Data in a Grid - Displaying Data in a Grid - Specifying and Formatting Columns to Display - Styling the Grid as a Whole - Paging Through Data
Display charts in a web page	Displaying Data in a Chart - The Chart Helper - Creating a Chart from Data - Displaying Charts Inside a Web Page - Styling a Chart - Saving a Chart
Read and write files in a website	Working with Files - Creating a Text File and Writing Data to It - Appending Data to an Existing File - Reading and Displaying Data from a File - Deleting Files - Letting Users Upload a File - Letting Users Upload Multiple Files
Display and manipulate images in a web page	Working with Images - Adding an Image to a Web Page Dynamically - Uploading an Image - Resizing an Image - Rotating and Flipping an Image - Adding a Watermark to an Image - Using an Image As a Watermark
Display video (and play audio) in a web page	Working with Video - Choosing a Video Player - Playing Flash (.swf) Videos - Playing MediaPlayer (.wmv) Videos - Playing Silverlight Videos
Send email messages from a website	Adding Email to Your Web Site - Sending Email Messages from Your Web site - Sending a File Using Email
Add search to a website	Adding Search to Your Web Site - Searching from Your Website
Add social network feeds and bookmarking to a website	Adding Social Networking to Your Web Site - Linking Your Website on Social Networking Sites - Adding a Twitter Feed - Rendering a Gravatar Image - Displaying an Xbox Gamer Card - Displaying a Facebook "Like" Button
Analyze site traffic and optimize for SEO	Analyzing Traffic - Tracking Visitor Information (Analytics) Use WebMatrix to optimize your site for search engines Optimize for search (video)
Improve performance on a website	Adding Caching for Faster Websites - Caching to Improve Website Responsiveness
Add security (membership) to a website	Adding Security and Membership - Introduction to Website Membership - Creating a Website That Has Registration and Login Pages - Creating a Members-Only Page - Creating Security for Groups of Users (Roles) - Creating a Password-Change Page - Letting Users Generate a New Password - Preventing Automated Programs from Joining Your Website
Test secure communication (SSL)	Securing Web Communications: Certificates, SSL, and https://

SCENARIO	CONTENT
Debug web pages using tools and coding techniques	Introduction to Debugging - Using the ServerInfo Helper to Display Server Information - Embedding Output Expressions to Display Page Values - Using the ObjectInfo Helper to Display Object Values - Using Debugging Tools
Add site-wide functionality in code	Customizing Site-Wide Behavior - Adding Website Startup Code - Running Code Before and After Files in a Folder - Creating More Readable and Searchable URLs
Publish a website	How to publish a web application using WebMatrix Download, Edit, and Republish Your Website with WebMatrix Microsoft WebMatrix: Publish Compatibility Microsoft WebMatrix: Certificate errors during publishing Site publishing (video)
Use IIS Express to test websites	IIS Express Overview Use the Windows System Tray to Manage Websites and Applications Running IIS Express from the Command Line Securing Web Communications: Certificates, SSL, and https:// IIS Express FAQ

Additional Resources

Read

- Microsoft Web Camps training kit for WebMatrix
- Submit a bug or suggestion
- WebMatrix Readme
- App Gallery FAQs
- IIS Express Documentation
- ASP.NET Razor Syntax code samples and tutorial download page
- Ask a question or discuss in the forums
- SQL Server Compact information

View (videos)

- Introduction to WebMatrix
- Start from Scratch
- Start with Web Apps
- Small, but complete package
- Grows with you
- Open Source
- Database made simple
- Elegant interface, simple experience
- Simple to code
- More than "Hello, World"
- Desktop or server, it's all the same
- Tightly knit, fully integrated
- Optimize for search
- Site publishing
- Customize and deploy your CMS

Configuration builders for ASP.NET

9/16/2020 • 13 minutes to read • Edit Online

By Stephen Molloy and Rick Anderson

Configuration builders provide a modern and agile mechanism for ASP.NET apps to get configuration values from external sources.

Configuration builders:

- Are available in .NET Framework 4.7.1 and later.
- Provide a flexible mechanism for reading configuration values.
- Address some of the basic needs of apps as they move into a container and cloud focused environment.
- Can be used to improve protection of configuration data by drawing from sources previously unavailable (for example, Azure Key Vault and environment variables) in the .NET configuration system.

Key/value configuration builders

A common scenario that can be handled by configuration builders is to provide a basic key/value replacement mechanism for configuration sections that follow a key/value pattern. The .NET Framework concept of ConfigurationBuilders is not limited to specific configuration sections or patterns. However, many of the configuration builders in `Microsoft.Configuration.ConfigurationBuilders` (github, NuGet) work within the key/value pattern.

Key/value configuration builders settings

The following settings apply to all key/value configuration builders in `Microsoft.Configuration.ConfigurationBuilders`.

Mode

The configuration builders use an external source of key/value information to populate selected key/value elements of the configuration system. Specifically, the `<appSettings/>` and `<connectionStrings/>` sections receive special treatment from the configuration builders. The builders work in three modes:

- `Strict` - The default mode. In this mode, the configuration builder only operates on well-known key/value-centric configuration sections. `Strict` mode enumerates each key in the section. If a matching key is found in the external source:
  - The configuration builders replace the value in the resulting configuration section with the value from the external source.
- `Greedy` - This mode is closely related to `Strict` mode. Rather than being limited to keys that already exist in the original configuration:
  - The configuration builders adds all key/value pairs from the external source into the resulting configuration section.
- `Expand` - Operates on the raw XML before it's parsed into a configuration section object. It can be thought of as an expansion of tokens in a string. Any part of the raw XML string that matches the pattern `${token}` is a candidate for token expansion. If no corresponding value is found in the external source, then the token is not changed. Builders in this mode are not limited to the `<appSettings/>` and `<connectionStrings/>` sections.

The following markup from *web.config* enables the EnvironmentConfigBuilder in `Strict` mode:

```
<configuration>

  <configSections>
    <section name="configBuilders"
             type="System.Configuration.ConfigurationBuildersSection,
             System.Configuration, Version=4.0.0.0, Culture=neutral,
             PublicKeyToken=b03f5f7f11d50a3a"
             restartOnExternalChanges="false" requirePermission="false" />
  </configSections>

  <configBuilders>
    <builders>
      <add name="MyEnvironment"
           type="Microsoft.Configuration.ConfigurationBuilders.EnvironmentConfigBuilder,
           Microsoft.Configuration.ConfigurationBuilders.Environment,
           Version=1.0.0.0, Culture=neutral" />
    </builders>
  </configBuilders>

  <appSettings configBuilders="MyEnvironment">
    <add key="ServiceID" value="ServiceID value from web.config" />
    <add key="ServiceKey" value="ServiceKey value from web.config" />
  </appSettings>

  <connectionStrings configBuilders="MyEnvironment">
    <add name="default" connectionString="Data Source=web.config/mydb.db" />
  </connectionStrings>
```

The following code reads the `<appSettings/>` and `<connectionStrings/>` shown in the preceding *web.config* file:

```
using System;
using System.Configuration;
using System.Web.UI;

namespace MyConfigBuilders
{
    public partial class About : Page
    {
        public string ServiceID { get; set; }
        public string ServiceKey { get; set; }
        public string ConString { get; set; }

        protected void Page_Load(object sender, EventArgs e)
        {
            ServiceID = ConfigurationManager.AppSettings["ServiceID"];
            ServiceKey = ConfigurationManager.AppSettings["ServiceKey"];
            ConString = ConfigurationManager.ConnectionStrings["default"]
                                                ?.ConnectionString;
        }
    }
}
```

The preceding code will set the property values to:

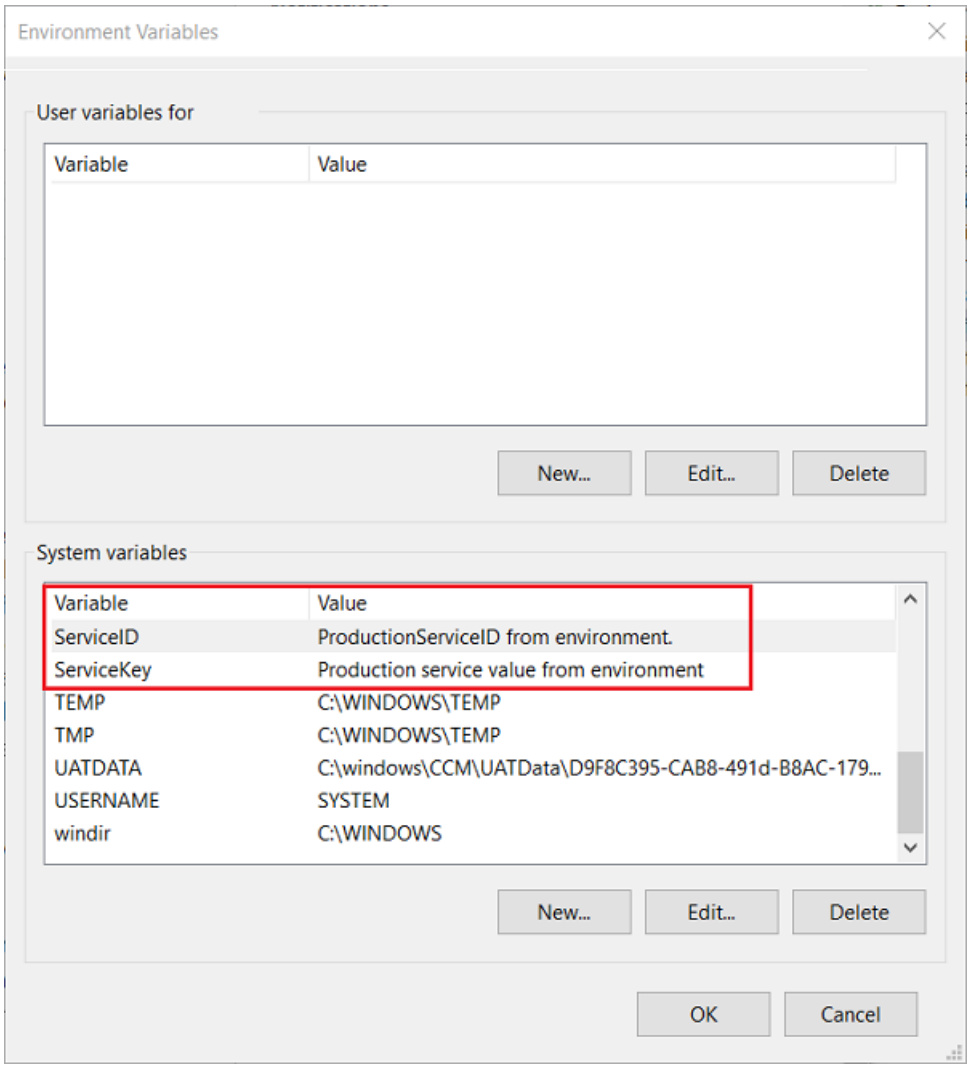
- The values in the *web.config* file if the keys are not set in environment variables.
- The values of the environment variable, if set.

For example, `ServiceID` will contain:

- "ServiceID value from web.config", if the environment variable `ServiceID` is not set.

- The value of the `ServiceID` environment variable, if set.

The following image shows the `<appSettings/>` keys/values from the preceding *web.config* file set in the environment editor:



Note: You might need to exit and restart Visual Studio to see changes in environment variables.

Prefix handling

Key prefixes can simplify setting keys because:

- The .NET Framework configuration is complex and nested.
- External key/value sources are commonly basic and flat by nature. For example, environment variables are not nested.

Use any of the following approaches to inject both `<appSettings/>` and `<connectionStrings/>` into the configuration via environment variables:

- With the `EnvironmentConfigBuilder` in the default `Strict` mode and the appropriate key names in the configuration file. The preceding code and markup takes this approach. Using this approach you can **not** have identically named keys in both `<appSettings/>` and `<connectionStrings/>`.
- Use two `EnvironmentConfigBuilder`s in `Greedy` mode with distinct prefixes and `stripPrefix`. With this approach, the app can read `<appSettings/>` and `<connectionStrings/>` without needing to update the configuration file. The next section, stripPrefix, shows how to do this.
- Use two `EnvironmentConfigBuilder`s in `Greedy` mode with distinct prefixes. With this approach you can't have duplicate key names as key names must differ by prefix. For example:

```
<configuration>

  <configSections>
    <section name="configBuilders"
             type="System.Configuration.ConfigurationBuildersSection,
             System.Configuration, Version=4.0.0.0, Culture=neutral,
             PublicKeyToken=b03f5f7f11d50a3a"
             restartOnExternalChanges="false" requirePermission="false" />
  </configSections>

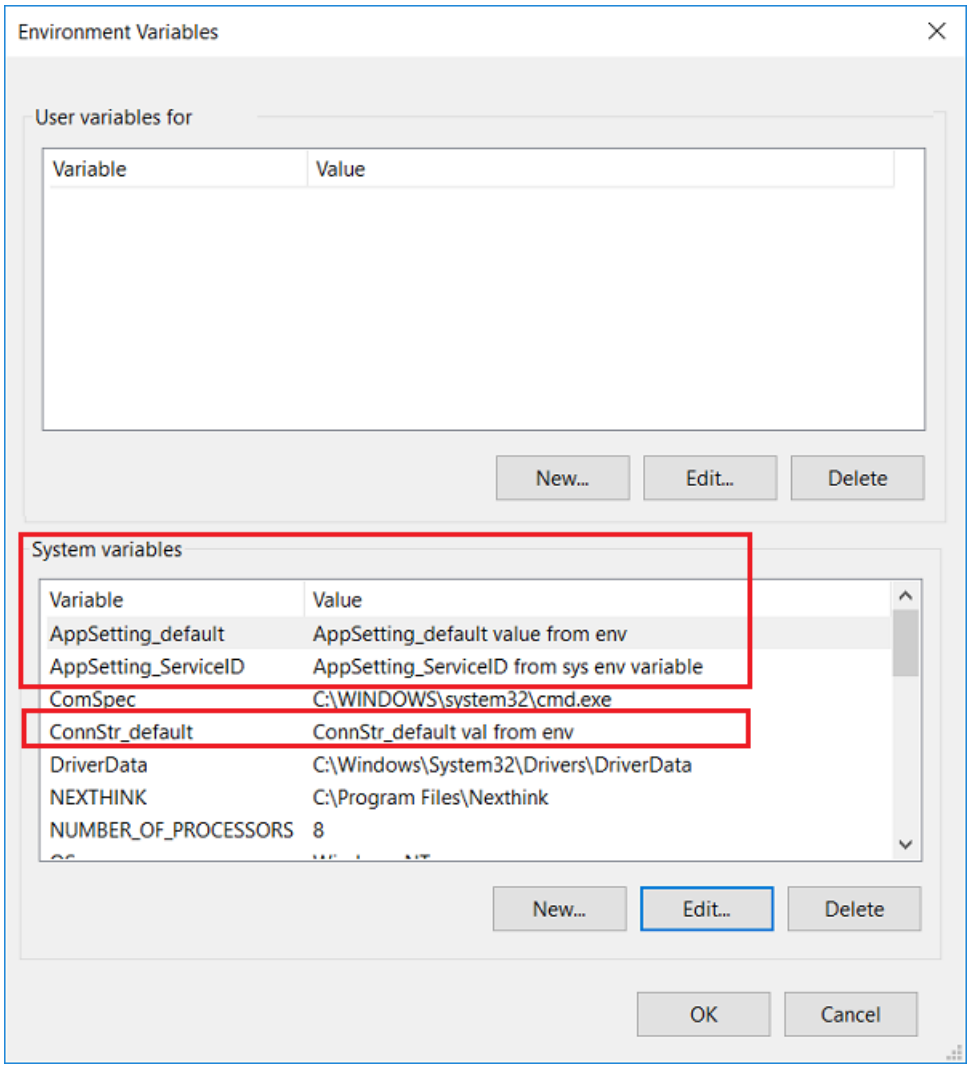
  <configBuilders>
    <builders>
      <add name="AS_Environment" mode="Greedy" prefix="AppSetting_"
           type="Microsoft.Configuration.ConfigurationBuilders.EnvironmentConfigBuilder,
           Microsoft.Configuration.ConfigurationBuilders.Environment" />
      <add name="CS_Environment" mode="Greedy" prefix="ConnStr_"
           type="Microsoft.Configuration.ConfigurationBuilders.EnvironmentConfigBuilder,
           Microsoft.Configuration.ConfigurationBuilders.Environment" />
    </builders>
  </configBuilders>

  <appSettings configBuilders="AS_Environment">
    <add key="AppSetting_ServiceID" value="ServiceID value from web.config" />
    <add key="AppSetting_default" value="AppSetting_default value from web.config" />
  </appSettings>

  <connectionStrings configBuilders="CS_Environment">
    <add name="ConnStr_default" connectionString="Data Source=web.config/mydb.db" />
  </connectionStrings>
```

With the preceding markup, the same flat key/value source can be used to populate configuration for two different sections.

The following image shows the `<appSettings/>` and `<connectionStrings/>` keys/values from the preceding *web.config* file set in the environment editor:



The following code reads the `<appSettings/>` and `<connectionStrings/>` keys/values contained in the preceding *web.config* file:

```
public partial class Contact : Page
{
    public string ServiceID { get; set; }
    public string AppSetting_default { get; set; }
    public string ConString { get; set; }

    protected void Page_Load(object sender, EventArgs e)
    {
        ServiceID = ConfigurationManager.AppSettings["AppSetting_ServiceID"];
        AppSetting_default = ConfigurationManager.AppSettings["AppSetting_default"];
        ConString = ConfigurationManager.ConnectionStrings["ConnStr_default"]
                                    ?.ConnectionString;
    }
}
```

The preceding code will set the property values to:

- The values in the *web.config* file if the keys are not set in environment variables.
- The values of the environment variable, if set.

For example, using the previous *web.config* file, the keys/values in the previous environment editor image, and the previous code, the following values are set:

KEY	VALUE
AppSetting_ServiceID	AppSetting_ServiceID from env variables
AppSetting_default	AppSetting_default value from env
ConnStr_default	ConnStr_default val from env

stripPrefix

`stripPrefix`: boolean, defaults to `false`.

The preceding XML markup separates app settings from connection strings but requires all the keys in the *web.config* file to use the specified prefix. For example, the prefix `AppSetting` must be added to the `ServiceID` key ("AppSetting_ServiceID"). With `stripPrefix`, the prefix is not used in the *web.config* file. The prefix is required in the configuration builder source (for example, in the environment.) We anticipate most developers will use `stripPrefix`.

Applications typically strip off the prefix. The following *web.config* strips the prefix:

```
<configuration>

  <configSections>
    <section name="configBuilders"
             type="System.Configuration.ConfigurationBuildersSection,
             System.Configuration, Version=4.0.0.0, Culture=neutral,
             PublicKeyToken=b03f5f7f11d50a3a"
             restartOnExternalChanges="false" requirePermission="false" />
  </configSections>

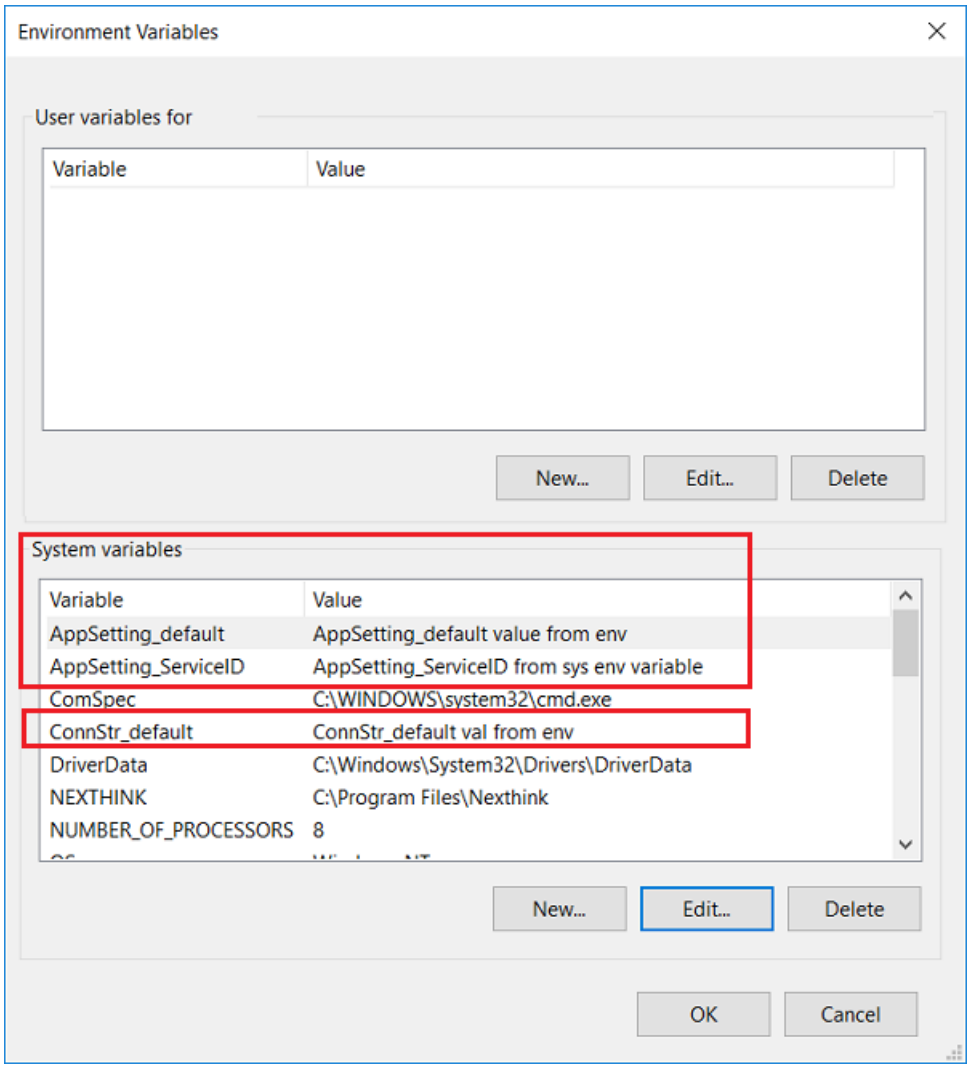
  <configBuilders>
    <builders>
      <add name="AS_Environment" mode="Greedy" prefix="AppSetting_"
           stripPrefix="true"
           type="Microsoft.Configuration.ConfigurationBuilders.EnvironmentConfigBuilder,
           Microsoft.Configuration.ConfigurationBuilders.Environment,
           Version=1.0.0.0, Culture=neutral" />
      <add name="CS_Environment" mode="Greedy" prefix="ConnStr_"
           stripPrefix="true"
            type="Microsoft.Configuration.ConfigurationBuilders.EnvironmentConfigBuilder,
           Microsoft.Configuration.ConfigurationBuilders.Environment,
           Version=1.0.0.0, Culture=neutral" />
    </builders>
  </configBuilders>

  <appSettings configBuilders="AS_Environment">
    <add key="ServiceID" value="ServiceID value from web.config" />
    <add key="default" value="AppSetting_default value from web.config" />
  </appSettings>

  <connectionStrings configBuilders="CS_Environment">
    <add name="default" connectionString="Data Source=web.config/mydb.db" />
  </connectionStrings>
```

In the preceding *web.config* file, the `default` key is in both the `<appSettings/>` and `<connectionStrings/>`.

The following image shows the `<appSettings/>` and `<connectionStrings/>` keys/values from the preceding *web.config* file set in the environment editor:



The following code reads the `<appSettings/>` and `<connectionStrings/>` keys/values contained in the preceding *web.config* file:

```
public partial class About2 : Page
{
    public string ServiceID { get; set; }
    public string AppSetting_default { get; set; }
    public string ConString { get; set; }

    protected void Page_Load(object sender, EventArgs e)
    {
        ServiceID = ConfigurationManager.AppSettings["ServiceID"];
        AppSetting_default = ConfigurationManager.AppSettings["default"];
        ConString = ConfigurationManager.ConnectionStrings["default"]
                                        ?.ConnectionString;
    }
}
```

The preceding code will set the property values to:

- The values in the *web.config* file if the keys are not set in environment variables.
- The values of the environment variable, if set.

For example, using the previous *web.config* file, the keys/values in the previous environment editor image, and the previous code, the following values are set:

KEY	VALUE
ServiceID	AppSetting_ServiceID from env variables
default	AppSetting_default value from env
default	ConnStr_default val from env

tokenPattern

`tokenPattern`: String, defaults to `@"\$\{(\w+)\}"`

The `Expand` behavior of the builders searches the raw XML for tokens that look like `${token}`. Searching is done with the default regular expression `@"\$\{(\w+)\}"`. The set of characters that matches `\w` is more strict than XML and many configuration sources allow. Use `tokenPattern` when more characters than `@"\$\{(\w+)\}"` are required in the token name.

`tokenPattern`: String:

- Allows developers to change the regex that is used for token matching.
- No validation is done to make sure it is a well-formed, non-dangerous regex.
- It must contain a capture group. The entire regex must match the entire token. The first capture must be the token name to look up in the configuration source.

Configuration builders in Microsoft.Configuration.ConfigurationBuilders

EnvironmentConfigBuilder

```
<add name="Environment"
    [mode|prefix|stripPrefix|tokenPattern]
    type="Microsoft.Configuration.ConfigurationBuilders.EnvironmentConfigBuilder,
    Microsoft.Configuration.ConfigurationBuilders.Environment" />
```

The EnvironmentConfigBuilder:

- Is the simplest of the configuration builders.
- Reads values from the environment.
- Does not have any additional configuration options.
- The `name` attribute value is arbitrary.

Note: In a Windows container environment, variables set at run time are only injected into the EntryPoint process environment. Apps that run as a service or a non-EntryPoint process do not pick up these variables unless they are otherwise injected through a mechanism in the container. For IIS/ASP.NET-based containers, the current version of ServiceMonitor.exe handles this in the *DefaultAppPool* only. Other Windows-based container variants may need to develop their own injection mechanism for non-EntryPoint processes.

UserSecretsConfigBuilder

> **WARNING**
>
> Never store passwords, sensitive connection strings, or other sensitive data in source code. Production secrets should not be used for development or test.

```
<add name="UserSecrets"
    [mode|prefix|stripPrefix|tokenPattern]
    (userSecretsId="{secret string, typically a GUID}" | userSecretsFile="~\secrets.file")
    [optional="true"]
    type="Microsoft.Configuration.ConfigurationBuilders.UserSecretsConfigBuilder,
    Microsoft.Configuration.ConfigurationBuilders.UserSecrets" />
```

This configuration builder provides a feature similar to ASP.NET Core Secret Manager.

The UserSecretsConfigBuilder can be used in .NET Framework projects, but a secrets file must be specified. Alternatively, you can define the `UserSecretsId` property in the project file and create the raw secrets file in the correct location for reading. To keep external dependencies out of your project, the secret file is XML formatted. The XML formatting is an implementation detail, and the format should not be relied upon. If you need to share a *secrets.json* file with .NET Core projects, consider using the SimpleJsonConfigBuilder. The `SimpleJsonConfigBuilder` format for .NET Core should also be considered an implementation detail subject to change.

Configuration attributes for `UserSecretsConfigBuilder`:

- `userSecretsId` - This is the preferred method for identifying an XML secrets file. It works similar to .NET Core, which uses a `UserSecretsId` project property to store this identifier. The string must be unique, it doesn't need to be a GUID. With this attribute, the `UserSecretsConfigBuilder` look in a well-known local location (`%APPDATA%\Microsoft\UserSecrets\<UserSecrets Id>\secrets.xml`) for a secrets file belonging to this identifier.
- `userSecretsFile` - An optional attribute specifying the file containing the secrets. The `~` character can be used at the start to reference the application root. Either this attribute or the `userSecretsId` attribute is required. If both are specified, `userSecretsFile` takes precedence.
- `optional`: boolean, default value `true` - Prevents an exception if the secrets file cannot be found.
- The `name` attribute value is arbitrary.

The secrets file has the following format:

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
  <secrets ver="1.0">
    <secret name="secret key name" value="secret value" />
  </secrets>
</root>
```

AzureKeyVaultConfigBuilder

```
<add name="AzureKeyVault"
    [mode|prefix|stripPrefix|tokenPattern]
    (vaultName="MyVaultName" |
     uri="https:/MyVaultName.vault.azure.net")
    [connectionString="connection string"]
    [version="secrets version"]
    [preloadSecretNames="true"]
    type="Microsoft.Configuration.ConfigurationBuilders.AzureKeyVaultConfigBuilder,
    Microsoft.Configuration.ConfigurationBuilders.Azure" />
```

The AzureKeyVaultConfigBuilder reads values stored in the Azure Key Vault.

`vaultName` is required (either the name of the vault or a URI to the vault). The other attributes allow control about which vault to connect to, but are only necessary if the application is not running in an environment that works with `Microsoft.Azure.Services.AppAuthentication`. The Azure Services Authentication library is used to automatically pick up connection information from the execution environment if possible. You can override automatically pick up of connection information by providing a connection string.

- `vaultName` - Required if `uri` in not provided. Specifies the name of the vault in your Azure subscription from which to read key/value pairs.
- `connectionString` - A connection string usable by AzureServiceTokenProvider
- `uri` - Connects to other Key Vault providers with the specified `uri` value. If not specified, Azure (`vaultName`) is the vault provider.
- `version` - Azure Key Vault provides a versioning feature for secrets. If `version` is specified, the builder only retrieves secrets matching this version.
- `preloadSecretNames` - By default, this builder querys **all** key names in the key vault when it is initialized. To prevent reading all key values, set this attribute to `false`. Setting this to `false` reads secrets one at a time. Reading secrets one at a time can useful if the vault allows "Get" access but not "List" access. **Note:** When using `Greedy` mode, `preloadSecretNames` must be `true` (the default.)

KeyPerFileConfigBuilder

```
<add name="KeyPerFile"
    [mode|prefix|stripPrefix|tokenPattern]
    (directoryPath="PathToSourceDirectory")
    [ignorePrefix="ignore."]
    [keyDelimiter=":"]
    [optional="false"]
    type="Microsoft.Configuration.ConfigurationBuilders.KeyPerFileConfigBuilder,
    Microsoft.Configuration.ConfigurationBuilders.KeyPerFile" />
```

KeyPerFileConfigBuilder is a basic configuration builder that uses a directory's files as a source of values. A file's name is the key, and the contents are the value. This configuration builder can be useful when running in an orchestrated container environment. Systems like Docker Swarm and Kubernetes provide `secrets` to their orchestrated windows containers in this key-per-file manner.

Attribute details:

- `directoryPath` - Required. Specifies a path to look in for values. Docker for Windows secrets are stored in the *C:\ProgramData\Docker\secrets* directory by default.
- `ignorePrefix` - Files that start with this prefix are excluded. Defaults to "ignore.".
- `keyDelimiter` - Default value is `null`. If specified, the configuration builder traverses multiple levels of the directory, building up key names with this delimiter. If this value is `null`, the configuration builder only looks at the top level of the directory.
- `optional` - Default value is `false`. Specifies whether the configuration builder should cause errors if the source directory doesn't exist.

SimpleJsonConfigBuilder

> **WARNING**
> Never store passwords, sensitive connection strings, or other sensitive data in source code. Production secrets should not be used for development or test.

```
<add name="SimpleJson"
    [mode|prefix|stripPrefix|tokenPattern]
    jsonFile="~\config.json"
    [optional="true"]
    [jsonMode="(Flat|Sectional)"]
    type="Microsoft.Configuration.ConfigurationBuilders.SimpleJsonConfigBuilder,
    Microsoft.Configuration.ConfigurationBuilders.Json" />
```

.NET Core projects frequently use JSON files for configuration. The SimpleJsonConfigBuilder builder allows .NET

Core JSON files to be used in the .NET Framework. This configuration builder is provides a basic mapping from a flat key/value source into specific key/value areas of .NET Framework configuration. This configuration builder does **not** provide for hierarchical configurations. The JSON backing file is similar to a dictionary, not a complex hierarchical object. A multi-level hierarchical file can be used. This provider `flatten`s the depth by appending the property name at each level using `:` as a delimiter.

Attribute details:

- `jsonFile` - Required. Specifies the JSON file to read from. The `~` character can be used at the start to reference the app root.
- `optional` - Boolean, default value is `true`. Prevents throwing exceptions if the JSON file cannot be found.
- `jsonMode` - `[Flat|Sectional]`. `Flat` is the default. When `jsonMode` is `Flat`, the JSON file is a single flat key/value source. The `EnvironmentConfigBuilder` and `AzureKeyVaultConfigBuilder` are also single flat key/value sources. When the `SimpleJsonConfigBuilder` is configured in `Sectional` mode:
  - The JSON file is conceptually divided just at the top level into multiple dictionaries.
  - Each of the dictionaries is only applied to the configuration section that matches the top-level property name attached to them. For example:

```
{
    "appSettings" : {
        "setting1" : "value1",
        "setting2" : "value2",
        "complex" : {
            "setting1" : "complex:value1",
            "setting2" : "complex:value2",
        }
    }
}
```

Configuration builders order

See ConfigurationBuilders Order of Execution in the aspnet/MicrosoftConfigurationBuilders GitHub repository.

Implementing a custom key/value configuration builder

If the configuration builders don't meet your needs, you can write a custom one. The `KeyValueConfigBuilder` base class handles substitution modes and most prefix concerns. An implementing project need only:

- Inherit from the base class and implement a basic source of key/value pairs through the `GetValue` and `GetAllValues`:
- Add the Microsoft.Configuration.ConfigurationBuilders.Base to the project.

```
using Microsoft.Configuration.ConfigurationBuilders;
using System.Collections.Generic;

public class MyCustomConfigBuilder : KeyValueConfigBuilder
{
    public override string GetValue(string key)
    {
        // Key lookup should be case-insensitive, because most key/value collections in
        // .NET Framework config sections are case-insensitive.
        return "Value for given key, or null.";
    }

    public override ICollection<KeyValuePair<string, string>> GetAllValues(string prefix)
    {
        // Populate the return collection.
        return new Dictionary<string, string>() { { "one", "1" }, { "two", "2" } };
    }
}
```

The `KeyValueConfigBuilder` base class provides much of the work and consistent behavior across key/value configuration builders.

Additional resources

- Configuration Builders GitHub repository
- Service-to-service authentication to Azure Key Vault using .NET

SignalR

2/20/2020 • 2 minutes to read • Edit Online

- Guidance
  - Getting Started
    - Introduction to SignalR
    - Supported Platforms
    - Tutorial: Getting Started with SignalR 2
    - Tutorial: Getting Started with SignalR 2 and MVC 5
    - Tutorial: High-Frequency Realtime with SignalR 2
    - Tutorial: Server Broadcast with SignalR 2
    - Hands On Lab: Real-Time Web Applications with SignalR
  - Guide to the API
    - Hubs API Guide - Server (C#)
    - Hubs API Guide - JavaScript Client
    - Hubs API Guide - .NET Client (C#)
    - Understanding and Handling Connection Lifetime Events in SignalR
    - Working with Groups in SignalR
    - Mapping SignalR Users to Connections
  - Testing and Debugging
    - SignalR Troubleshooting
    - Unit Testing SignalR Applications
    - Enabling SignalR Tracing
  - Security
    - Introduction to SignalR Security
    - Authentication and Authorization for SignalR Hubs
    - Authentication and Authorization for SignalR Persistent Connections
  - Performance
    - SignalR Performance
    - Introduction to Scaleout in SignalR
    - SignalR Scaleout with Azure Service Bus
    - SignalR Scaleout with Redis
    - SignalR Scaleout with SQL Server
    - SignalR Connection Density Testing with Crank
    - Using SignalR Performance Counters in an Azure Web Role
  - Deployment and Hosting
    - Tutorial: SignalR Self-Host
    - Using SignalR with Web Apps in Azure App Service
  - Advanced
    - Dependency Injection in SignalR

- Releases
  - Upgrading SignalR 1.x Projects to version 2
- Older Versions
  - Tutorial: Getting Started with SignalR 1.x
  - Tutorial: Getting Started with SignalR 1.x and MVC 4
  - High-Frequency Realtime with SignalR 1.x
  - Tutorial: Server Broadcast with ASP.NET SignalR 1.x
  - Hubs API Guide - Server
  - SignalR 1.x Hubs API Guide - JavaScript Client
  - Hubs API Guide - .NET Client
  - Understanding and Handling Connection Lifetime Events in SignalR
  - Working with Groups in SignalR 1.x
  - Mapping SignalR Users to Connections in SignalR 1.x
  - Introduction to SignalR Security (SignalR 1.x)
  - Authentication and Authorization for SignalR Hubs (SignalR 1.x)
  - Authentication and Authorization for SignalR Persistent Connections (SignalR 1.x)
  - SignalR Performance (SignalR 1.x)
  - Introduction to Scaleout in SignalR 1.x
  - SignalR Scaleout with Azure Service Bus (SignalR 1.x)
  - SignalR Scaleout with Redis (SignalR 1.x)
  - SignalR Scaleout with SQL Server (SignalR 1.x)
  - SignalR Troubleshooting (SignalR 1.x)
  - Dependency Injection in SignalR 1.x
- Videos
  - ASP.NET SignalR Videos
    - SignalR and Web Sockets

SignalR Guidance

2/20/2020 • 2 minutes to read • Edit Online

- Getting Started
  - Introduction to SignalR
  - Supported Platforms
  - Tutorial: Getting Started with SignalR 2
  - Tutorial: Getting Started with SignalR 2 and MVC 5
  - Tutorial: High-Frequency Realtime with SignalR 2
  - Tutorial: Server Broadcast with SignalR 2
  - Hands On Lab: Real-Time Web Applications with SignalR
- Guide to the API
  - Hubs API Guide - Server (C#)
  - Hubs API Guide - JavaScript Client
  - Hubs API Guide - .NET Client (C#)
  - Understanding and Handling Connection Lifetime Events in SignalR
  - Working with Groups in SignalR
  - Mapping SignalR Users to Connections
- Testing and Debugging
  - SignalR Troubleshooting
  - Unit Testing SignalR Applications
  - Enabling SignalR Tracing
- Security
  - Introduction to SignalR Security
  - Authentication and Authorization for SignalR Hubs
  - Authentication and Authorization for SignalR Persistent Connections
- Performance
  - SignalR Performance
  - Introduction to Scaleout in SignalR
  - SignalR Scaleout with Azure Service Bus
  - SignalR Scaleout with Redis
  - SignalR Scaleout with SQL Server
  - SignalR Connection Density Testing with Crank
  - Using SignalR Performance Counters in an Azure Web Role
- Deployment and Hosting
  - Tutorial: SignalR Self-Host
  - Using SignalR with Web Apps in Azure App Service
- Advanced
  - Dependency Injection in SignalR
- Releases

  - Upgrading SignalR 1.x Projects to version 2
- Older Versions
  - Tutorial: Getting Started with SignalR 1.x
  - Tutorial: Getting Started with SignalR 1.x and MVC 4
  - High-Frequency Realtime with SignalR 1.x
  - Tutorial: Server Broadcast with ASP.NET SignalR 1.x
  - Hubs API Guide - Server
  - SignalR 1.x Hubs API Guide - JavaScript Client
  - Hubs API Guide - .NET Client
  - Understanding and Handling Connection Lifetime Events in SignalR
  - Working with Groups in SignalR 1.x
  - Mapping SignalR Users to Connections in SignalR 1.x
  - Introduction to SignalR Security (SignalR 1.x)
  - Authentication and Authorization for SignalR Hubs (SignalR 1.x)
  - Authentication and Authorization for SignalR Persistent Connections (SignalR 1.x)
  - SignalR Performance (SignalR 1.x)
  - Introduction to Scaleout in SignalR 1.x
  - SignalR Scaleout with Azure Service Bus (SignalR 1.x)
  - SignalR Scaleout with Redis (SignalR 1.x)
  - SignalR Scaleout with SQL Server (SignalR 1.x)
  - SignalR Troubleshooting (SignalR 1.x)
  - Dependency Injection in SignalR 1.x

SignalR Getting Started

2/20/2020 • 2 minutes to read • Edit Online

What SignalR is, platforms it runs on, and how to get started.

- Introduction to SignalR
- Supported Platforms
- Tutorial: Getting Started with SignalR 2
- Tutorial: Getting Started with SignalR 2 and MVC 5
- Tutorial: High-Frequency Realtime with SignalR 2
- Tutorial: Server Broadcast with SignalR 2
- Hands On Lab: Real-Time Web Applications with SignalR

Introduction to SignalR

9/16/2020 • 8 minutes to read • Edit Online

by Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This article describes what SignalR is, and some of the solutions it was designed to create.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

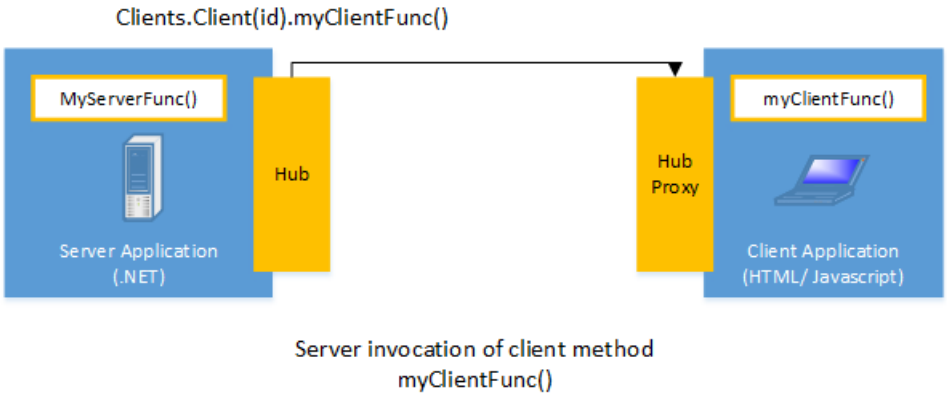
What is SignalR?

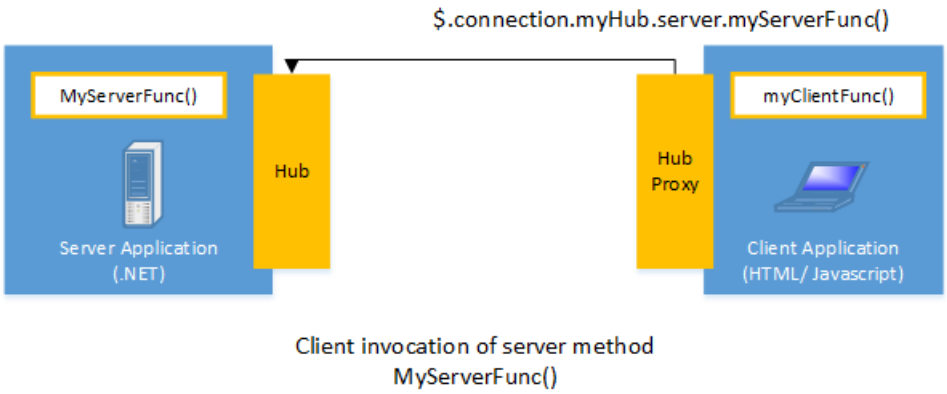
ASP.NET SignalR is a library for ASP.NET developers that simplifies the process of adding real-time web functionality to applications. Real-time web functionality is the ability to have server code push content to connected clients instantly as it becomes available, rather than having the server wait for a client to request new data.

SignalR can be used to add any sort of "real-time" web functionality to your ASP.NET application. While chat is often used as an example, you can do a whole lot more. Any time a user refreshes a web page to see new data, or the page implements long polling to retrieve new data, it is a candidate for using SignalR. Examples include dashboards and monitoring applications, collaborative applications (such as simultaneous editing of documents), job progress updates, and real-time forms.

SignalR also enables completely new types of web applications that require high frequency updates from the server, for example, real-time gaming.

SignalR provides a simple API for creating server-to-client remote procedure calls (RPC) that call JavaScript functions in client browsers (and other client platforms) from server-side .NET code. SignalR also includes API for connection management (for instance, connect and disconnect events), and grouping connections.





SignalR handles connection management automatically, and lets you broadcast messages to all connected clients simultaneously, like a chat room. You can also send messages to specific clients. The connection between the client and server is persistent, unlike a classic HTTP connection, which is re-established for each communication.

SignalR supports "server push" functionality, in which server code can call out to client code in the browser using Remote Procedure Calls (RPC), rather than the request-response model common on the web today.

SignalR applications can scale out to thousands of clients using built-in, and third-party scale-out providers.

Built-in providers include:

- Service Bus
- SQL Server
- Redis

Third-party providers include:

- NCache.

SignalR is open-source, accessible through GitHub.

SignalR and WebSocket

SignalR uses the new WebSocket transport where available and falls back to older transports where necessary. While you could certainly write your app using WebSocket directly, using SignalR means that a lot of the extra functionality you would need to implement is already done for you. Most importantly, this means that you can code your app to take advantage of WebSocket without having to worry about creating a separate code path for older clients. SignalR also shields you from having to worry about updates to WebSocket, since SignalR is updated to support changes in the underlying transport, providing your application a consistent interface across versions of WebSocket.

Transports and fallbacks

SignalR is an abstraction over some of the transports that are required to do real-time work between client and server. SignalR first attempts to establish a WebSocket connection if possible. WebSocket is the optimal transport for SignalR because it has:

- The most efficient use of server memory.
- The lowest latency.
- The most underlying features, such as full duplex communication between client and server.
- The most stringent requirements, WebSocket requires the server:
  - Run on Windows Server 2012 or Windows 8.
  - .NET Framework 4.5.

If these requirements are not met, SignalR attempts to use other transports to make its connections.

HTML 5 transports

These transports depend on support for HTML 5. If the client browser does not support the HTML 5 standard, older transports will be used.

- **WebSocket** (if both the server and browser indicate they can support Websocket). WebSocket is the only transport that establishes a true persistent, two-way connection between client and server. However, WebSocket also has the most stringent requirements; it is fully supported only in the latest versions of Microsoft Internet Explorer, Google Chrome, and Mozilla Firefox, and only has a partial implementation in other browsers such as Opera and Safari.
- **Server Sent Events**, also known as EventSource (if the browser supports Server Sent Events, which is basically all browsers except Internet Explorer.)

Comet transports

The following transports are based on the Comet web application model, in which a browser or other client maintains a long-held HTTP request, which the server can use to push data to the client without the client specifically requesting it.

- **Forever Frame** (for Internet Explorer only). Forever Frame creates a hidden IFrame which makes a request to an endpoint on the server that does not complete. The server then continually sends script to the client which is immediately executed, providing a one-way realtime connection from server to client. The connection from client to server uses a separate connection from the server to client connection, and like a standard HTTP request, a new connection is created for each piece of data that needs to be sent.
- **Ajax long polling**. Long polling does not create a persistent connection, but instead polls the server with a request that stays open until the server responds, at which point the connection closes, and a new connection is requested immediately. This may introduce some latency while the connection resets.

For more information on what transports are supported under which configurations, see Supported Platforms.

Transport selection process

The following list shows the steps that SignalR uses to decide which transport to use.

1. If the browser is Internet Explorer 8 or earlier, Long Polling is used.

2. If JSONP is configured (that is, the `jsonp` parameter is set to `true` when the connection is started), Long Polling is used.

3. If a cross-domain connection is being made (that is, if the SignalR endpoint is not in the same domain as the hosting page), then WebSocket will be used if the following criteria are met:

   - The client supports CORS (Cross-Origin Resource Sharing). For details on which clients support CORS, see CORS at caniuse.com.

- The client supports WebSocket
- The server supports WebSocket

  If any of these criteria are not met, Long Polling will be used. For more information on cross-domain connections, see How to establish a cross-domain connection.

4. If JSONP is not configured and the connection is not cross-domain, WebSocket will be used if both the client and server support it.
5. If either the client or server do not support WebSocket, Server Sent Events is used if it is available.
6. If Server Sent Events is not available, Forever Frame is attempted.
7. If Forever Frame fails, Long Polling is used.

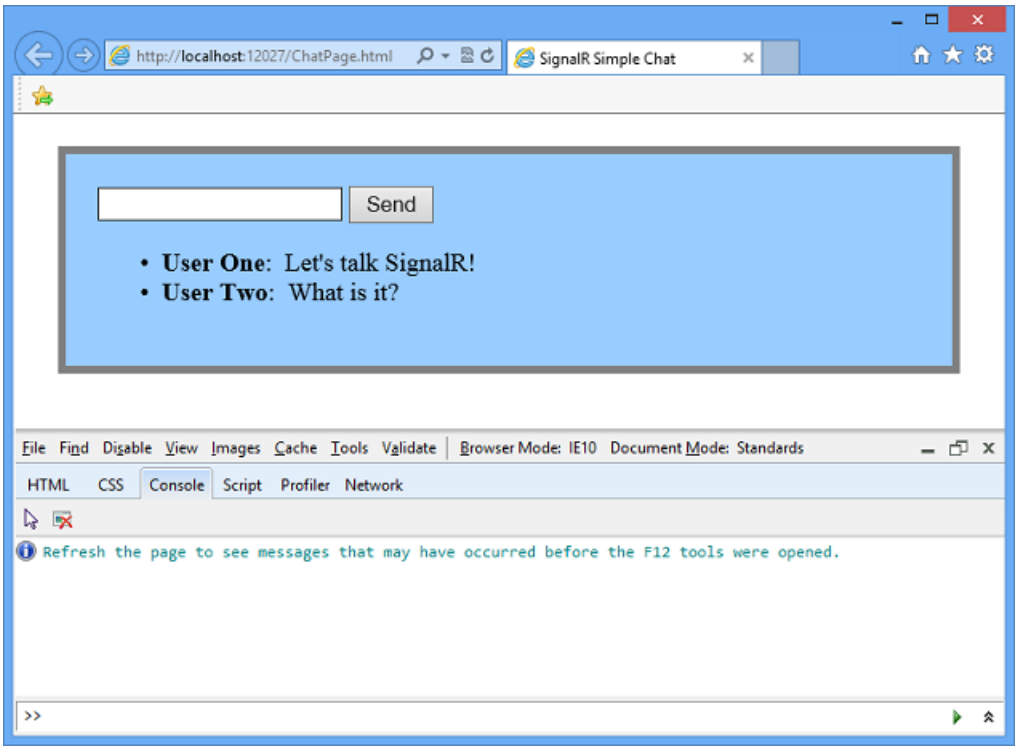
Monitoring transports

You can determine what transport your application is using by enabling logging on your hub, and opening the console window in your browser.

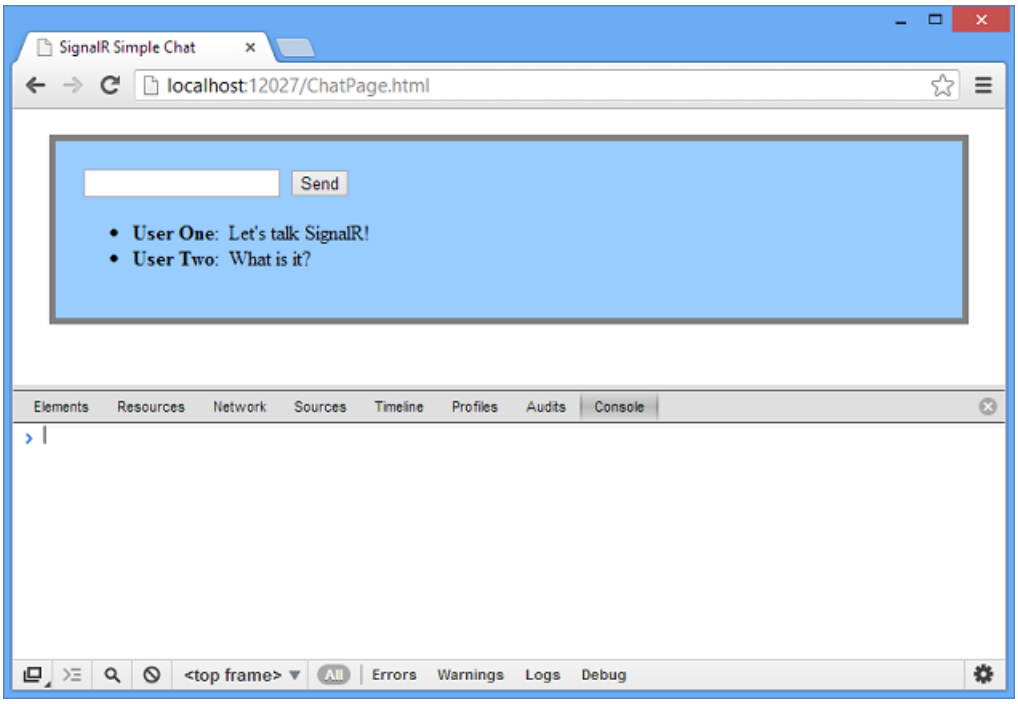
To enable logging for your hub's events in a browser, add the following command to your client application:

```
$.connection.hub.logging = true;
```

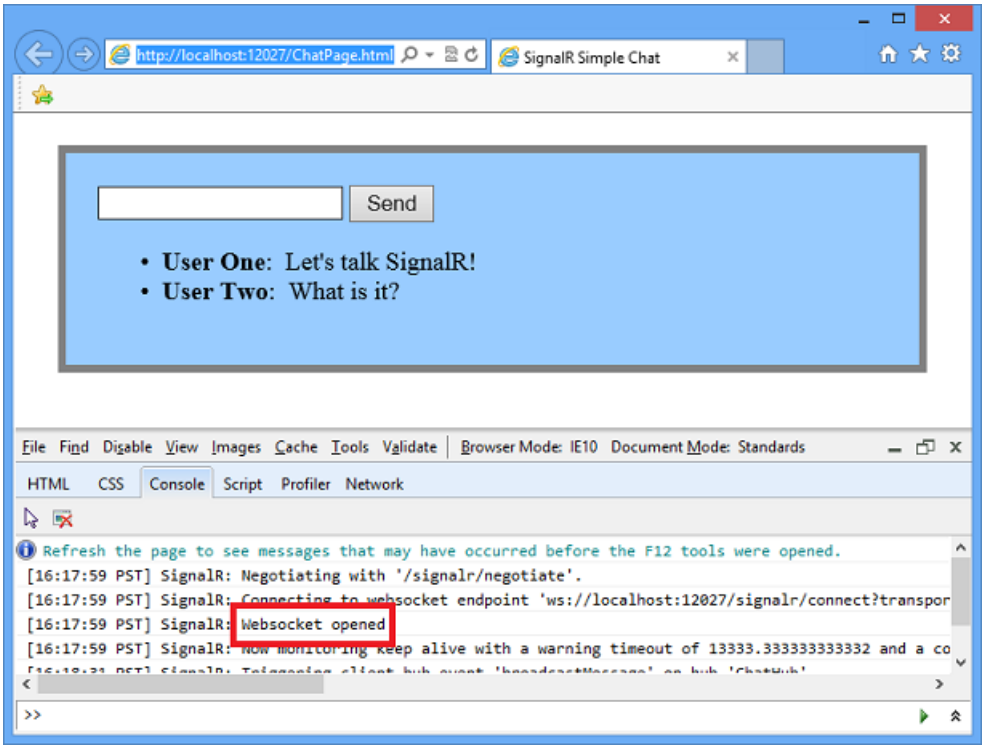
- In Internet Explorer, open the developer tools by pressing F12, and click the Console tab.



- In Chrome, open the console by pressing Ctrl+Shift+J.



With the console open and logging enabled, you'll be able to see which transport is being used by SignalR.



Specifying a transport

Negotiating a transport takes a certain amount of time and client/server resources. If the client capabilities are known, then a transport can be specified when the client connection is started. The following code snippet demonstrates starting a connection using the Ajax Long Polling transport, as would be used if it was known that the client did not support any other protocol:

```
connection.start({ transport: 'longPolling' });
```

You can specify a fallback order if you want a client to try specific transports in order. The following code snippet demonstrates trying WebSocket, and failing that, going directly to Long Polling.

```
connection.start({ transport: ['webSockets','longPolling'] });
```

The string constants for specifying transports are defined as follows:

- `webSockets`
- `foreverFrame`
- `serverSentEvents`
- `longPolling`

Connections and Hubs

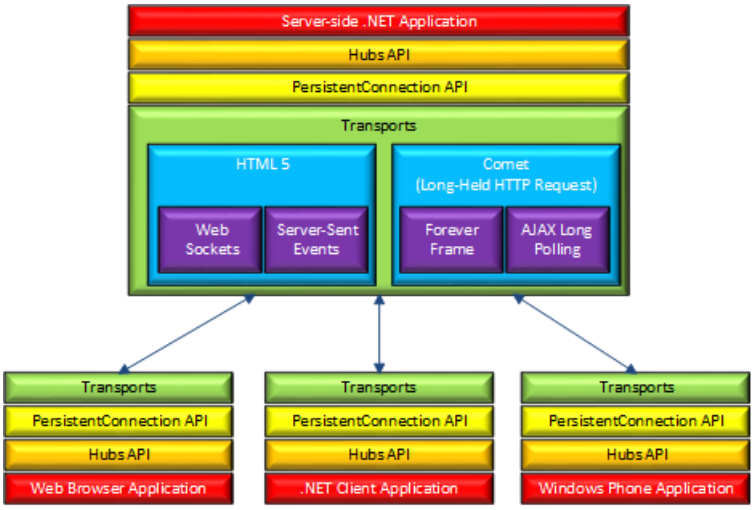
The SignalR API contains two models for communicating between clients and servers: Persistent Connections and Hubs.

A Connection represents a simple endpoint for sending single-recipient, grouped, or broadcast messages. The Persistent Connection API (represented in .NET code by the PersistentConnection class) gives the developer direct access to the low-level communication protocol that SignalR exposes. Using the Connections communication model will be familiar to developers who have used connection-based APIs such as Windows Communication Foundation.

A Hub is a more high-level pipeline built upon the Connection API that allows your client and server to call methods on each other directly. SignalR handles the dispatching across machine boundaries as if by magic, allowing clients to call methods on the server as easily as local methods, and vice versa. Using the Hubs communication model will be familiar to developers who have used remote invocation APIs such as .NET Remoting. Using a Hub also allows you to pass strongly typed parameters to methods, enabling model binding.

Architecture diagram

The following diagram shows the relationship between Hubs, Persistent Connections, and the underlying technologies used for transports.



How Hubs work

When server-side code calls a method on the client, a packet is sent across the active transport that contains the name and parameters of the method to be called (when an object is sent as a method parameter, it is serialized using JSON). The client then matches the method name to methods defined in client-side code. If there is a match, the client method will be executed using the deserialized parameter data.

The method call can be monitored using tools like Fiddler. The following image shows a method call sent from a SignalR server to a web browser client in the Logs pane of Fiddler. The method call is being sent from a hub called `MoveShapeHub`, and the method being invoked is called `updateShape`.

```
10:38:03:1298 Session846.WebSocket'WebSocket #846' - Pushing 104 bytes from server WebSocket
81 66 7B 22 43 22 3A 22 42 2C 31 35 7C 43 2C 30     f{"C":"B,15|C,0
7C 44 2C 30 7C 45 2C 30 22 2C 22 4D 22 3A 5B 7B  |D,0|E,0","M":[{
22 48 22 3A 22 4D 6F 76 65 53 68 61 70 65 48 75  "H":"MoveShapeHu
62 22 2C 22 4D 22 3A 22 75 70 64 61 74 65 53 68  b","M":"updateSh
61 70 65 22 2C 22 41 22 3A 5B 7B 22 6C 65 66 74  ape","A":[{"left
22 3A 35 30 31 2E 30 2C 22 74 6F 70 22 3A 33 30  ":501.0,"top":30
32 2E 30 7D 5D 7D 5D 7D                          2.0}]}]}
```

In this example, the hub name is identified with the `H` parameter; the method name is identified with the `M` parameter, and the data being sent to the method is identified with the `A` parameter. The application that generated this message is created in the High-Frequency Realtime tutorial.

Choosing a communication model

Most applications should use the Hubs API. The Connections API could be used in the following circumstances:

- The format of the actual message sent needs to be specified.
- The developer prefers to work with a messaging and dispatching model rather than a remote invocation model.
- An existing application that uses a messaging model is being ported to use SignalR.

Supported Platforms

2/20/2020 • 4 minutes to read • Edit Online

by Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This article describes what clients and servers are supported by SignalR.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

SignalR is supported under a variety of server and client configurations. In addition, each transport option has a set of requirements of its own; if the system requirements for a transport are not available, SignalR will gracefully failover to other transports. For more information on the transports that SignalR supports, see Transports and Fallbacks.

Server system requirements

The SignalR server component can be hosted on a variety of server configurations. This section describes the supported versions of operating systems, .NET framework, Internet Information Server, and other components.

Supported server operating systems

The SignalR server component can be hosted in the following server or client operating systems. Note that for SignalR to use WebSockets, Windows Server 2012, Windows Server 2016, or Windows 8 is required (WebSocket can be used on Windows Azure Web Sites, as long as the site's .NET framework version is set to 4.5, and Web Sockets is enabled in the site's Configuration page).

- Windows Server 2016
- Windows Server 2012
- Windows Server 2008 r2
- Windows 10
- Windows 8
- Windows 7
- Windows Azure

Supported server .NET Framework version

SignalR 2 is only supported on .NET Framework 4.5. See the Recommended Updates section for updates that enhance reliability, compatibility, stability, and performance.

Supported server IIS versions

When SignalR is hosted in IIS, the following versions are supported. Note that if a client operating system is used, such as for development (Windows 8 or Windows 7), full versions of IIS or Cassini should not be used, since there will be a limit of 10 simultaneous connections imposed, which will be reached very quickly since connections are

transient, frequently re-established, and are not disposed immediately upon no longer being used. IIS Express should be used on client operating systems.

Also note that for SignalR to use WebSocket, IIS 8 or IIS 8 Express must be used, the server must be using Windows 8, Windows Server 2012, or later, and WebSocket must be enabled in IIS. For information on how to enable WebSocket in IIS, see IIS 8.0 WebSocket Protocol Support.

- IIS 8 or IIS 8 Express.
- IIS 7 and 7.5. Support for extensionless URLs is required.
- IIS must be running in integrated mode; classic mode is not supported. Message delays of up to 30 seconds may be experienced if IIS is run in classic mode using the Server-Sent Events transport.
- The hosting application must be running in full trust mode.

Client system requirements

SignalR can be used in a variety of client platforms. This section describes the system requirements for using SignalR in web browsers, Windows desktop applications, Silverlight applications, and mobile devices.

Web browsers

SignalR can be used in a variety of web browsers, but typically, only the latest two versions are supported.

Applications that use SignalR in browsers must use jQuery version 1.6.4 or major later versions (such as 1.7.2, 1.8.2, or 1.9.1).

SignalR can be used in the following browsers:

- Microsoft Internet Explorer versions 8, 9, 10, and 11. Modern, Desktop, and Mobile versions are supported.
- Mozilla Firefox: current version - 1, both Windows and Mac versions.
- Google Chrome: current version - 1, both Windows and Mac versions.
- Safari: current version - 1, both Mac and iOS versions.
- Opera: current version - 1, Windows only.
- Android browser

In addition to requiring certain browsers, the various transports that SignalR uses have requirements of their own. The following transports are supported under the following configurations:

Web Browser Transport Requirements

TRANSPORT	INTERNET EXPLORER	CHROME (WINDOWS OR IOS)	FIREFOX	SAFARI (OSX OR IOS)	ANDROID
WebSockets	10+	current - 1	current - 1	current - 1	N/A
Server-Sent Events	N/A	current - 1	current - 1	current - 1	N/A
ForeverFrame	8+	N/A	N/A	N/A	4.1
Long Polling	8+	current - 1	current - 1	current - 1	4.1

*: 6+ required for full functionality.

Unsupported Browsers

While SignalR *may* run without major issues in older browser versions, we do not actively test SignalR in them and generally do not fix bugs that may appear in them.

Windows Desktop and Silverlight Applications

In addition to running in a web browser, SignalR can be hosted in standalone Windows client or Silverlight applications. Windows Desktop and Silverlight SignalR applications have the following system requirements.

- Applications using .NET 4 are supported on Windows XP SP3 or later.
- Applications using .NET Framework 4.5 are supported on Windows Vista or later.

In addition to operating system and .NET framework requirements, the transports available to SignalR have requirements of their own. The following transports are supported under the following configurations:

Windows Desktop and Silverlight Transport Requirements

TRANSPORT	.NET APPLICATION	SILVERLIGHT
Web Sockets	Windows 8+ and .NET 4.5+	N/A
Forever Frame	N/A	N/A
Server-Sent Events	.NET 4+	5+
Long Polling	.NET 4+	5+

Windows Store and Windows Phone Applications

SignalR can be used in Windows Store applications and Windows Phone 8 applications. The following transports are supported under the following configurations:

Windows Store and Windows Phone Transport Requirements

TRANSPORT	WINDOWS STORE/ .NET	WINDOWS STORE/ JAVASCRIPT	WINDOWS PHONE/ IE	WINDOWS PHONE/ .NET
WebSockets	N/A	Win8+	8+	N/A
Forever Frame	N/A	Win8+	7.5+	N/A
Server-Sent Events	Win8+	N/A	N/A	8+
Long Polling	Win8+	Win8+	7.5+	8+

Recommended Updates

The following updates are recommended for SignalR servers:

- An update for .NET Framework 4.5 is available here.
- Microsoft will periodically release QFEs for ASP.NET. These should be applied as available.

Tutorial: Real-time chat with SignalR 2

2/20/2020 • 6 minutes to read • Edit Online

This tutorial shows you how to use SignalR to create a real-time chat application. You add SignalR to an empty ASP.NET web application and create an HTML page to send and display messages.

In this tutorial, you:

- Set up the project
- Run the sample
- Examine the code

> **WARNING**
>
> This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

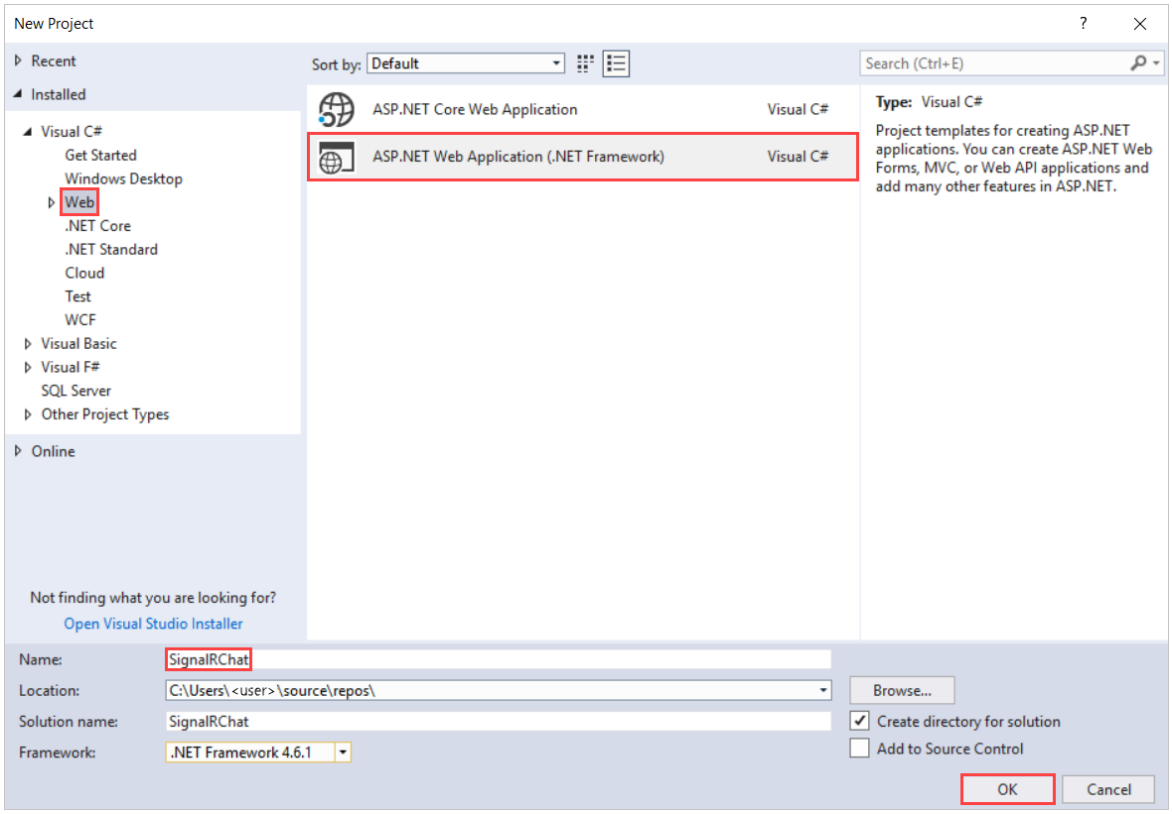
Prerequisites

- Visual Studio 2017 with the **ASP.NET and web development** workload.

Set up the Project

This section shows how to use Visual Studio 2017 and SignalR 2 to create an empty ASP.NET web application, add SignalR, and create the chat application.

1. In Visual Studio, create an ASP.NET Web Application.



2. In the **New ASP.NET Project - SignalRChat** window, leave **Empty** selected and select **OK**.

3. In **Solution Explorer**, right-click the project and select **Add** > **New Item**.
4. In **Add New Item - SignalRChat**, select **Installed** > **Visual C#** > **Web** > **SignalR** and then select **SignalR Hub Class (v2)**.
5. Name the class *ChatHub* and add it to the project.

   This step creates the *ChatHub.cs* class file and adds a set of script files and assembly references that support SignalR to the project.
6. Replace the code in the new *ChatHub.cs* class file with this code:

```
using System;
using System.Web;
using Microsoft.AspNet.SignalR;
namespace SignalRChat
{
    public class ChatHub : Hub
    {
        public void Send(string name, string message)
        {
            // Call the broadcastMessage method to update clients.
            Clients.All.broadcastMessage(name, message);
        }
    }
}
```

7. In **Solution Explorer**, right-click the project and select **Add** > **New Item**.
8. In **Add New Item - SignalRChat** select **Installed** > **Visual C#** > **Web** and then select **OWIN Startup Class**.
9. Name the class *Startup* and add it to the project.
10. Replace the default code in *Startup* class with this code:

```
using Microsoft.Owin;
using Owin;
[assembly: OwinStartup(typeof(SignalRChat.Startup))]
namespace SignalRChat
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Any connection or hub wire up and configuration should go here
            app.MapSignalR();
        }
    }
}
```

11. In **Solution Explorer**, right-click the project and select **Add** > **HTML Page**.
12. Name the new page *index* and select **OK**.
13. In **Solution Explorer**, right-click the HTML page you created and select **Set as Start Page**.
14. Replace the default code in the HTML page with this code:

```
<!DOCTYPE html>
<html>
<head>
    <title>SignalR Simple Chat</title>
    <style type="text/css">
        .container {
            background-color: #99CCFF;
            border: thick solid #808080;
            padding: 20px;
            margin: 20px;
        }
    </style>
</head>
<body>
    <div class="container">
        <input type="text" id="message" />
        <input type="button" id="sendmessage" value="Send" />
        <input type="hidden" id="displayname" />
        <ul id="discussion">
        </ul>
    </div>
    <!--Script references. -->
    <!--Reference the jQuery library. -->
    <script src="Scripts/jquery-3.1.1.min.js" ></script>
    <!--Reference the SignalR library. -->
    <script src="Scripts/jquery.signalR-2.2.1.min.js"></script>
    <!--Reference the autogenerated SignalR hub script. -->
    <script src="signalr/hubs"></script>
    <!--Add script to update the page and send messages.-->
    <script type="text/javascript">
        $(function () {
            // Declare a proxy to reference the hub.
            var chat = $.connection.chatHub;
            // Create a function that the hub can call to broadcast messages.
            chat.client.broadcastMessage = function (name, message) {
                // Html encode display name and message.
                var encodedName = $('<div />').text(name).html();
                var encodedMsg = $('<div />').text(message).html();
                // Add the message to the page.
                $('#discussion').append('<li><strong>' + encodedName
                    + '</strong>:  ' + encodedMsg + '</li>');
            };
            // Get the user name and store it to prepend to messages.
            $('#displayname').val(prompt('Enter your name:', ''));
            // Set initial focus to message input box.
            $('#message').focus();
            // Start the connection.
            $.connection.hub.start().done(function () {
                $('#sendmessage').click(function () {
                    // Call the Send method on the hub.
                    chat.server.send($('#displayname').val(), $('#message').val());
                    // Clear text box and reset focus for next comment.
                    $('#message').val('').focus();
                });
            });
        });
    </script>
</body>
</html>
```

15. In **Solution Explorer**, expand **Scripts**.

    Script libraries for jQuery and SignalR are visible in the project.

> **IMPORTANT**
>
> The package manager may have installed a later version of the SignalR scripts.

16. Check that the script references in the code block correspond to the versions of the script files in the project.

    Script references from the original code block:

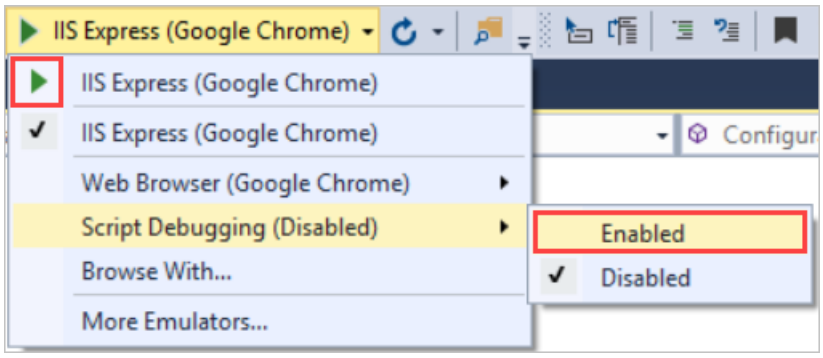
    ```
    <!--Script references. -->
    <!--Reference the jQuery library. -->
    <script src="Scripts/jquery-3.1.1.min.js" ></script>
    <!--Reference the SignalR library. -->
    <script src="Scripts/jquery.signalR-2.2.1.min.js"></script>
    ```

17. If they don't match, update the *.html* file.

18. From the menu bar, select **File** > **Save All**.

Run the Sample

1. In the toolbar, turn on **Script Debugging** and then select the play button to run the sample in Debug mode.



2. When the browser opens, enter a name for your chat identity.

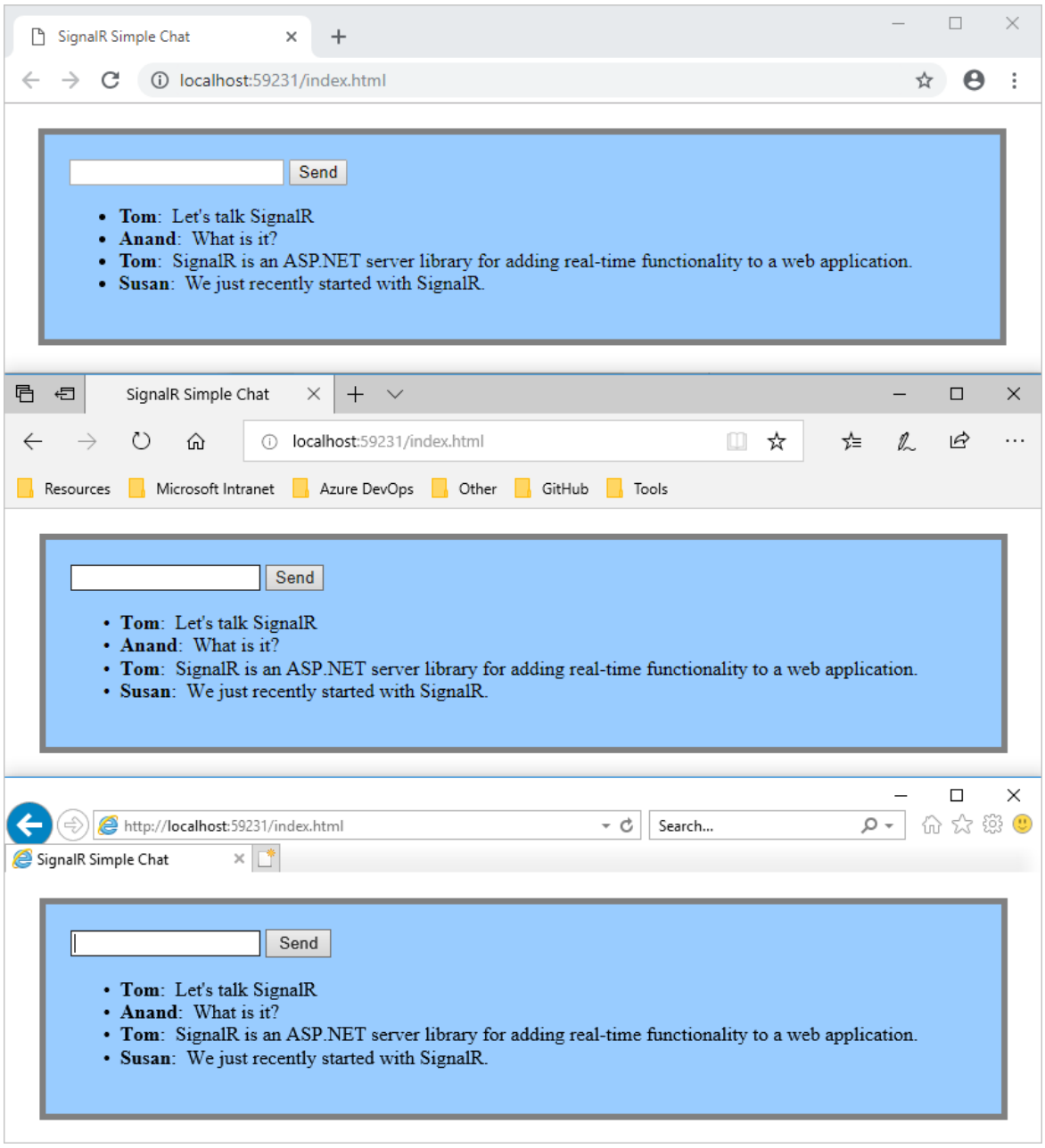
3. Copy the URL from the browser, open two other browsers, and paste the URLs into the address bars.

4. In each browser, enter a unique name.

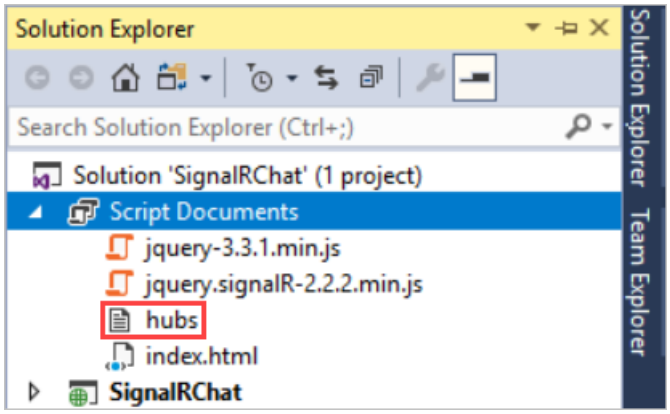
5. Now, add a comment and select **Send**. Repeat that in the other browsers. The comments appear in real-time.

> **NOTE**
>
> This simple chat application does not maintain the discussion context on the server. The hub broadcasts comments to all current users. Users who join the chat later will see messages added from the time they join.

See how the chat application runs in three different browsers. When Tom, Anand, and Susan send messages, all browsers update in real time:



6. In **Solution Explorer**, inspect the **Script Documents** node for the running application. There's a script file named *hubs* that the SignalR library generates at runtime. This file manages the communication between jQuery script and server-side code.



Examine the Code

The SignalRChat application demonstrates two basic SignalR development tasks. It shows you how to create a hub. The server uses that hub as the main coordination object. The hub uses the SignalR jQuery library to send

and receive messages.

SignalR Hubs in the ChatHub.cs

In the code sample above, the `ChatHub` class derives from the `Microsoft.AspNet.SignalR.Hub` class. Deriving from the `Hub` class is a useful way to build a SignalR application. You can create public methods on your hub class and then use those methods by calling them from scripts in a web page.

In the chat code, clients call the `ChatHub.Send` method to send a new message. The hub then sends the message to all clients by calling `Clients.All.broadcastMessage`.

The `Send` method demonstrates several hub concepts:

- Declare public methods on a hub so that clients can call them.
- Use the `Microsoft.AspNet.SignalR.Hub.Clients` dynamic property to communicate with all clients connected to this hub.
- Call a function on the client (like the `broadcastMessage` function) to update clients.

```
public class ChatHub : Hub
{
    public void Send(string name, string message)
    {
        // Call the broadcastMessage method to update clients.
        Clients.All.broadcastMessage(name, message);
    }
}
```

SignalR and jQuery in the index.html

The *index.html* page in the code sample shows how to use the SignalR jQuery library to communicate with a SignalR hub. The code carries out many important tasks. It declares a proxy to reference the hub, declares a function that the server can call to push content to clients, and it starts a connection to send messages to the hub.

```
var chat = $.connection.chatHub;
```

> **NOTE**
>
> In JavaScript the reference to the server class and its members has to be camelCase. The code sample references the C# *ChatHub* class in JavaScript as `chatHub`.

In this code block, you create a callback function in the script.

```
chat.client.broadcastMessage = function (name, message) {
        // Html encode display name and message.
        var encodedName = $('<div />').text(name).html();
        var encodedMsg = $('<div />').text(message).html();
        // Add the message to the page.
        $('#discussion').append('<li><strong>' + encodedName
            + '</strong>:  ' + encodedMsg + '</li>');
    };
```

The hub class on the server calls this function to push content updates to each client. The two lines that HTML-encode the content before displaying it are optional and show a good way to prevent script injection.

This code opens a connection with the hub.

```
$.connection.hub.start().done(function () {
        $('#sendmessage').click(function () {
            // Call the Send method on the hub.
            chat.server.send($('#displayname').val(), $('#message').val());
            // Clear text box and reset focus for next comment.
            $('#message').val('').focus();
        });
    });
```

> **NOTE**
>
> This approach ensures that the code establishes a connection before the event handler executes.

The code starts the connection and then passes it a function to handle the click event on the **Send** button in the HTML page.

Get the code

Download Completed Project

Additional resources

For more about SignalR, see the following resources:

- SignalR Project
- SignalR Github and Samples
- SignalR Wiki

Next steps

In this tutorial you:

- Set up the project
- Ran the sample
- Examined the code

Advance to the next article to learn how to use SignalR and MVC 5.

SignalR 2 and MVC 5

Tutorial: Real-time chat with SignalR 2 and MVC 5

2/20/2020 • 6 minutes to read • Edit Online

This tutorial shows how to use ASP.NET SignalR 2 to create a real-time chat application. You add SignalR to an MVC 5 application and create a chat view to send and display messages.

In this tutorial, you:

- Set up the project
- Run the sample
- Examine the code

> **WARNING**
>
> This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

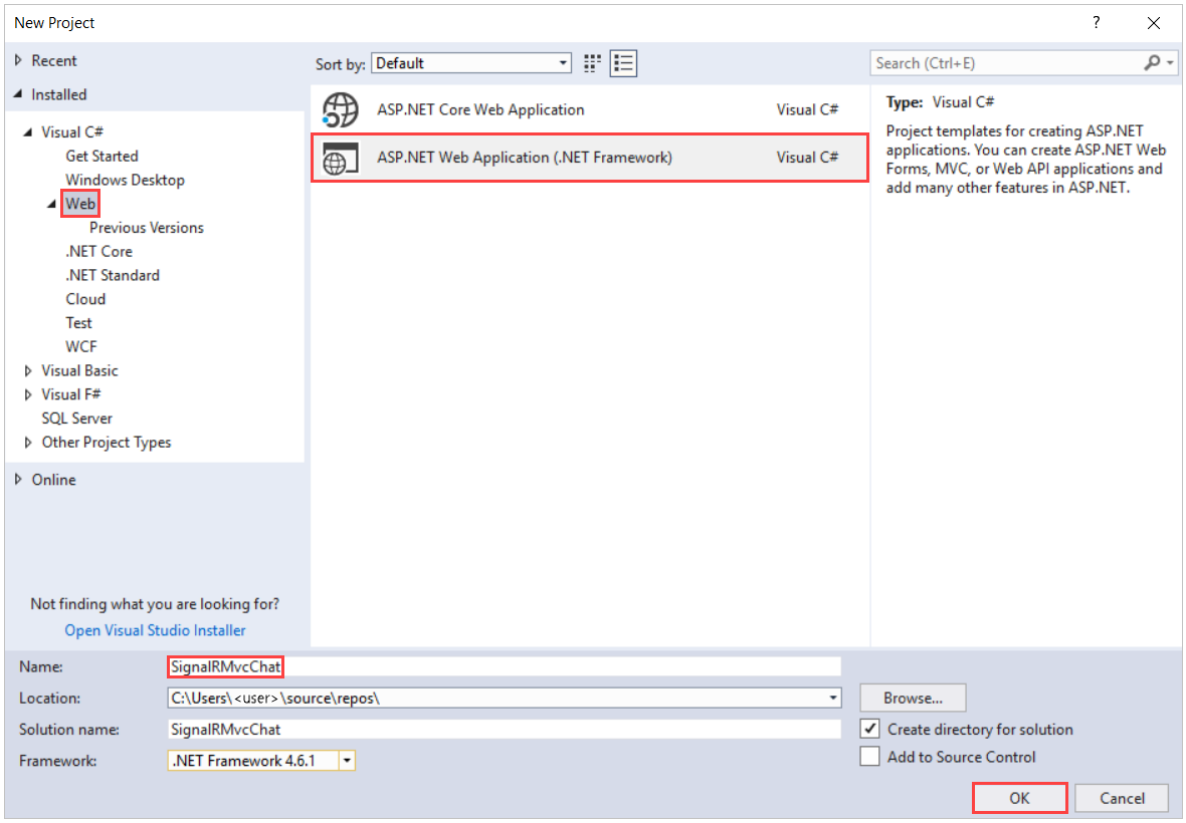
Prerequisites

- Visual Studio 2017 with the **ASP.NET and web development** workload.

Set up the Project

This section shows how to use Visual Studio 2017 and SignalR 2 to create an empty ASP.NET MVC 5 application, add the SignalR library, and create the chat application.

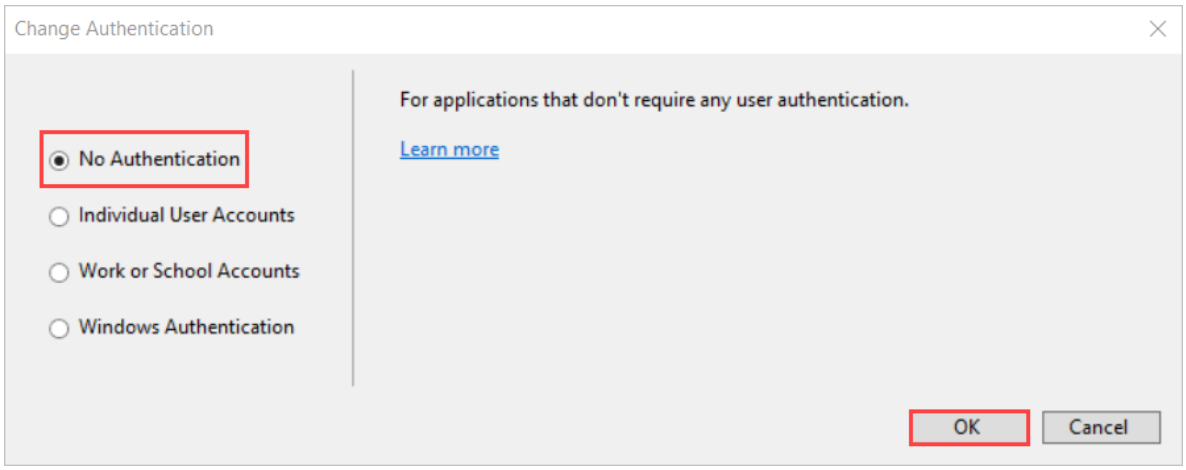
1. In Visual Studio, create a C# ASP.NET application that targets .NET Framework 4.5, name it SignalRChat, and click OK.



2. In **New ASP.NET Web Application - SignalRMvcChat**, select **MVC** and then select **Change**

Authentication.

3. In **Change Authentication**, select **No Authentication** and click **OK**.



4. In **New ASP.NET Web Application - SignalRMvcChat**, select **OK**.
5. In **Solution Explorer**, right-click the project and select **Add** > **New Item**.
6. In **Add New Item - SignalRChat**, select **Installed** > **Visual C#** > **Web** > **SignalR** and then select **SignalR Hub Class (v2)**.
7. Name the class *ChatHub* and add it to the project.

   This step creates the *ChatHub.cs* class file and adds a set of script files and assembly references that support SignalR to the project.

8. Replace the code in the new *ChatHub.cs* class file with this code:

```
using System;
using System.Web;
using Microsoft.AspNet.SignalR;
namespace SignalRChat
{
    public class ChatHub : Hub
    {
        public void Send(string name, string message)
        {
            // Call the addNewMessageToPage method to update clients.
            Clients.All.addNewMessageToPage(name, message);
        }
    }
}
```

9. In **Solution Explorer**, right-click the project and select **Add** > **Class**.
10. Name the new class *Startup* and add it to the project.
11. Replace the code in the *Startup.cs* class file with this code:

```
using Owin;
using Microsoft.Owin;
[assembly: OwinStartup(typeof(SignalRChat.Startup))]
namespace SignalRChat
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Any connection or hub wire up and configuration should go here
            app.MapSignalR();
        }
    }
}
```

12. In **Solution Explorer**, select **Controllers** > **HomeController.cs**.

13. Add this method to the *HomeController.cs*.

```
public ActionResult Chat()
{
    return View();
}
```

    This method returns the **Chat** view that you create in a later step.

14. In **Solution Explorer**, right-click **Views** > **Home**, and select **Add** > **View**.

15. In **Add View**, name the new view **Chat** and select **Add**.

16. Replace the contents of **Chat.cshtml** with this code:

```
@{
    ViewBag.Title = "Chat";
}
<h2>Chat</h2>
<div class="container">
    <input type="text" id="message" />
    <input type="button" id="sendmessage" value="Send" />
    <input type="hidden" id="displayname" />
    <ul id="discussion">
    </ul>
</div>
@section scripts {
    <!--Script references. -->
    <!--The jQuery library is required and is referenced by default in _Layout.cshtml. -->
    <!--Reference the SignalR library. -->
    <script src="~/Scripts/jquery.signalR-2.1.0.min.js"></script>
    <!--Reference the autogenerated SignalR hub script. -->
    <script src="~/signalr/hubs"></script>
    <!--SignalR script to update the chat page and send messages.-->
    <script>
        $(function () {
            // Reference the auto-generated proxy for the hub.
            var chat = $.connection.chatHub;
            // Create a function that the hub can call back to display messages.
            chat.client.addNewMessageToPage = function (name, message) {
                // Add the message to the page.
                $('#discussion').append('<li><strong>' + htmlEncode(name)
                    + '</strong>: ' + htmlEncode(message) + '</li>');
            };
            // Get the user name and store it to prepend to messages.
            $('#displayname').val(prompt('Enter your name:', ''));
            // Set initial focus to message input box.
            $('#message').focus();
            // Start the connection.
            $.connection.hub.start().done(function () {
                $('#sendmessage').click(function () {
                    // Call the Send method on the hub.
                    chat.server.send($('#displayname').val(), $('#message').val());
                    // Clear text box and reset focus for next comment.
                    $('#message').val('').focus();
                });
            });
        });
        // This optional function html-encodes messages for display in the page.
        function htmlEncode(value) {
            var encodedValue = $('<div />').text(value).html();
            return encodedValue;
        }
    </script>
}
```

17. In **Solution Explorer**, expand **Scripts**.

    Script libraries for jQuery and SignalR are visible in the project.

> **IMPORTANT**
>
> The package manager may have installed a later version of the SignalR scripts.

18. Check that the script references in the code block correspond to the versions of the script files in the project.

    Script references from the original code block:

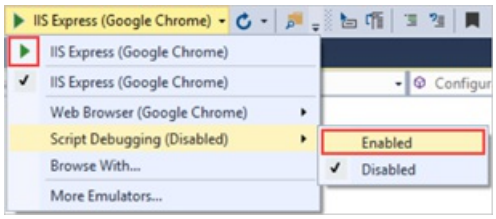
```
<!--Script references. -->
<!--The jQuery library is required and is referenced by default in _Layout.cshtml. -->
<!--Reference the SignalR library. -->
<script src="~/Scripts/jquery.signalR-2.1.0.min.js"></script>
```

19. If they don't match, update the *.cshtml* file.

20. From the menu bar, select **File** > **Save All**.

Run the Sample

1. In the toolbar, turn on **Script Debugging** and then select the play button to run the sample in Debug mode.



2. When the browser opens, enter a name for your chat identity.

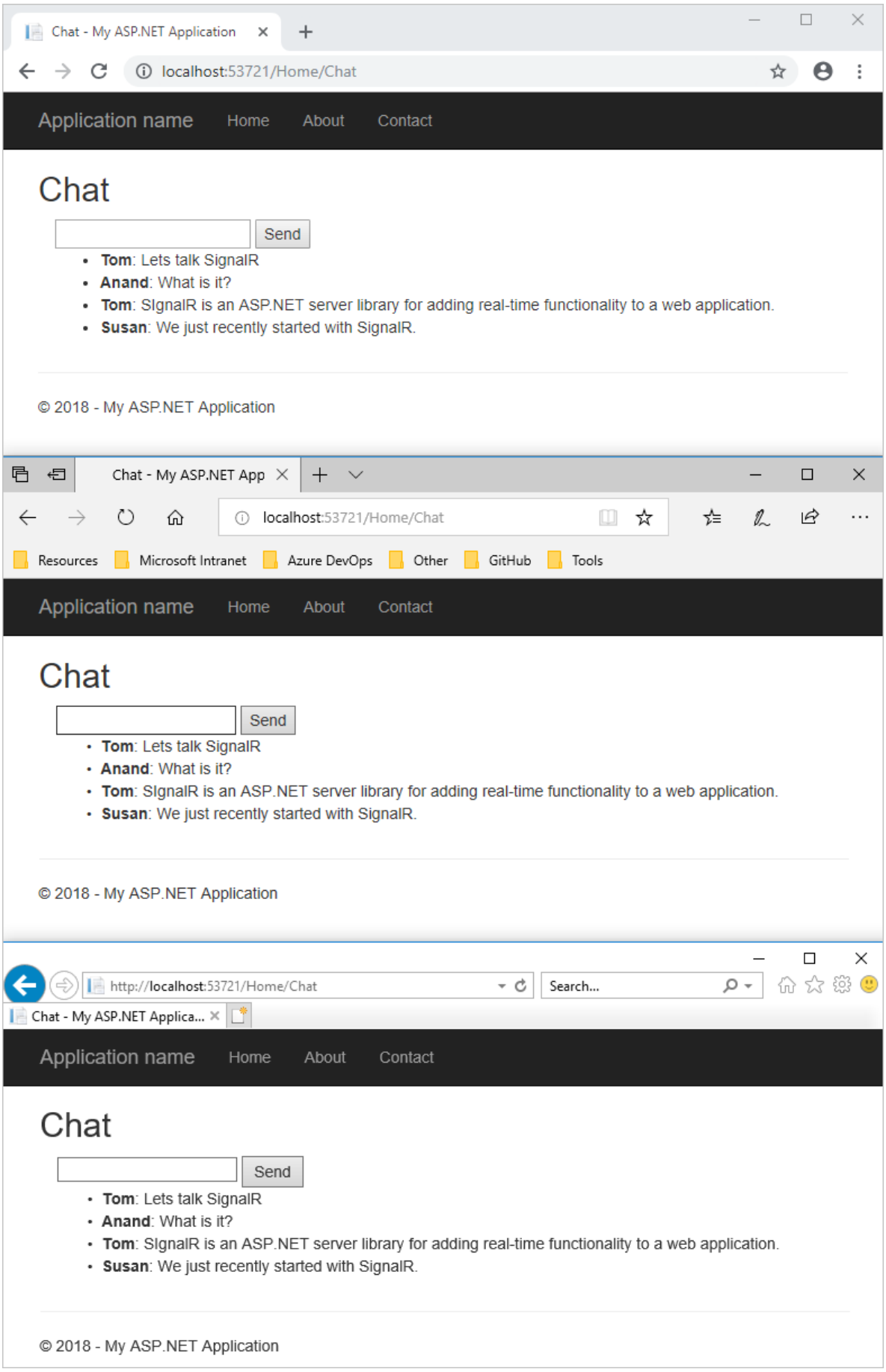
3. Copy the URL from the browser, open two other browsers, and paste the URLs into the address bars.

4. In each browser, enter a unique name.

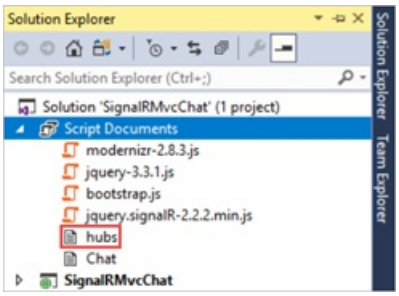
5. Now, add a comment and select **Send**. Repeat that in the other browsers. The comments appear in real time.

> **NOTE**
>
> This simple chat application does not maintain the discussion context on the server. The hub broadcasts comments to all current users. Users who join the chat later will see messages added from the time they join.

See how the chat application runs in three different browsers. When Tom, Anand, and Susan send messages, all browsers update in real time:



6. In **Solution Explorer**, inspect the **Script Documents** node for the running application. There's a script file named *hubs* that the SignalR library generates at runtime. This file manages the communication between jQuery script and server-side code.



Examine the Code

The SignalR chat application demonstrates two basic SignalR development tasks. It shows you how to create a hub. The server uses that hub as the main coordination object. The hub uses the SignalR jQuery library to send and receive messages.

SignalR Hubs in the ChatHub.cs

In the code sample, the `ChatHub` class derives from the `Microsoft.AspNet.SignalR.Hub` class. Deriving from the `Hub` class is a useful way to build a SignalR application. You can create public methods on your hub class and then access those methods by calling them from scripts in a web page.

In the chat code, clients call the `ChatHub.Send` method to send a new message. The hub in turn sends the message to all clients by calling `Clients.All.addNewMessageToPage`.

The `Send` method demonstrates several hub concepts:

- Declare public methods on a hub so that clients can call them.
- Use the `Microsoft.AspNet.SignalR.Hub.Clients` dynamic property to communicate with all clients connected to this hub.
- Call a function on the client (like the `addNewMessageToPage` function) to update clients.

```
public class ChatHub : Hub
{
    public void Send(string name, string message)
    {
        Clients.All.addNewMessageToPage(name, message);
    }
}
```

SignalR and jQuery Chat.cshtml

The *Chat.cshtml* view file in the code sample shows how to use the SignalR jQuery library to communicate with a SignalR hub. The code carries out many important tasks. It creates a reference to the autogenerated proxy for the hub, declares a function that the server can call to push content to clients, and it starts a connection to send messages to the hub.

```
var chat = $.connection.chatHub;
```

> **NOTE**
>
> In JavaScript, the reference to the server class and its members is in camelCase. The code sample references the C# `ChatHub` class in JavaScript as `chatHub`.

In this code block, you create a callback function in the script.

```
chat.client.addNewMessageToPage = function (name, message) {
    // Add the message to the page.
    $('#discussion').append('<li><strong>' + htmlEncode(name)
        + '</strong>: ' + htmlEncode(message) + '</li>');
};
```

The hub class on the server calls this function to push content updates to each client. The optional call to the `htmlEncode` function shows a way to HTML encode the message content before displaying it in the page. It's a way to prevent script injection.

This code opens a connection with the hub.

```
$.connection.hub.start().done(function () {
    $('#sendmessage').click(function () {
        // Call the Send method on the hub.
        chat.server.send($('#displayname').val(), $('#message').val());
        // Clear text box and reset focus for next comment.
        $('#message').val('').focus();
    });
});
```

> **NOTE**
>
> This approach ensures that you establish a connection before the event handler executes.

The code starts the connection and then passes it a function to handle the click event on the **Send** button in the Chat page.

Get the code

Download Completed Project

Additional resources

For more about SignalR, see the following resources:

- SignalR Project
- SignalR GitHub and Samples
- SignalR Wiki

Next steps

In this tutorial, you:

- Set up the project
- Ran the sample
- Examined the code

Advance to the next article to learn how to create a web application that uses ASP.NET SignalR 2 to provide high-frequency messaging functionality.

Web app with high-frequency messaging

Tutorial: Create high-frequency real-time app with SignalR 2

2/20/2020 • 12 minutes to read • Edit Online

This tutorial shows how to create a web application that uses ASP.NET SignalR 2 to provide high-frequency messaging functionality. In this case, "high-frequency messaging" means the server sends updates at a fixed rate. You send up to 10 messages a second.

The application you create displays a shape that users can drag. The server updates the position of the shape in all connected browsers to match the position of the dragged shape using timed updates.

Concepts introduced in this tutorial have applications in real-time gaming and other simulation applications.

In this tutorial, you:

- Set up the project
- Create the base application
- Map to the hub when app starts
- Add the client
- Run the app
- Add the client loop
- Add the server loop
- Add smooth animation

> **WARNING**
>
> This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

Prerequisites

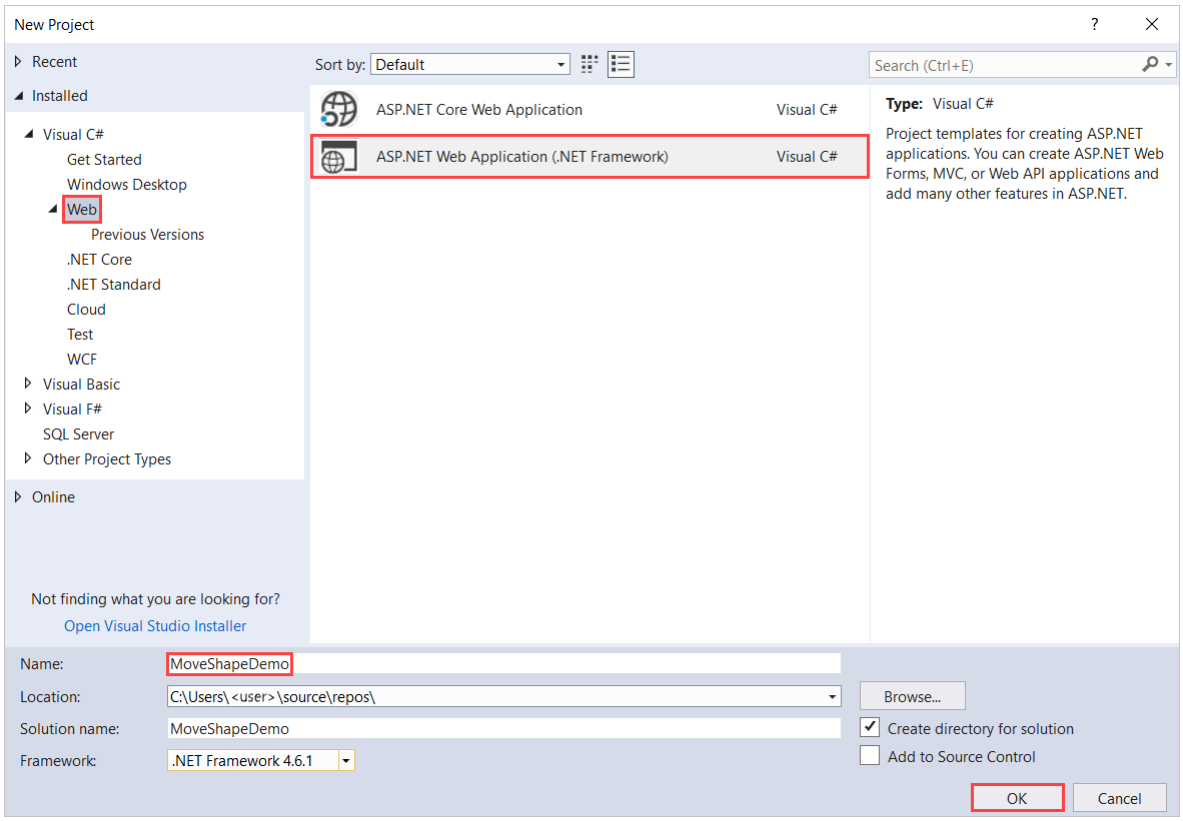
- Visual Studio 2017 with the **ASP.NET and web development** workload.

Set up the project

In this section, you create the project in Visual Studio 2017.

This section shows how to use Visual Studio 2017 to create an empty ASP.NET Web Application and add the SignalR and jQuery.UI libraries.

1. In Visual Studio, create an ASP.NET Web Application.



2. In the **New ASP.NET Web Application - MoveShapeDemo** window, leave **Empty** selected and select **OK**.

3. In **Solution Explorer**, right-click the project and select **Add** > **New Item**.

4. In **Add New Item - MoveShapeDemo**, select **Installed** > **Visual C#** > **Web** > **SignalR** and then select **SignalR Hub Class (v2)**.

5. Name the class *MoveShapeHub* and add it to the project.

   This step creates the *MoveShapeHub.cs* class file. Simultaneously, it adds a set of script files and assembly references that support SignalR to the project.

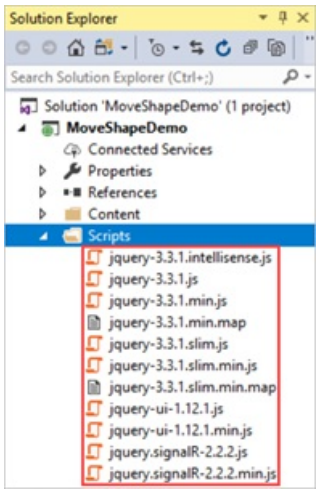
6. Select **Tools** > **NuGet Package Manager** > **Package Manager Console**.

7. In **Package Manager Console**, run this command:

   ```
   Install-Package jQuery.UI.Combined
   ```

   The command installs the jQuery UI library. You use it to animate the shape.

8. In **Solution Explorer**, expand the Scripts node.



Script libraries for jQuery, jQueryUI, and SignalR are visible in the project.

Create the base application

In this section, you create a browser application. The app sends the location of the shape to the server during each mouse move event. The server broadcasts this information to all other connected clients in real time. You learn more about this application in later sections.

1. Open the *MoveShapeHub.cs* file.

2. Replace the code in the *MoveShapeHub.cs* file with this code:

```
using Microsoft.AspNet.SignalR;
using Newtonsoft.Json;

namespace MoveShapeDemo
{
    public class MoveShapeHub : Hub
    {
        public void UpdateModel(ShapeModel clientModel)
        {
            clientModel.LastUpdatedBy = Context.ConnectionId;
            // Update the shape model within our broadcaster
            Clients.AllExcept(clientModel.LastUpdatedBy).updateShape(clientModel);
        }
    }
    public class ShapeModel
    {
        // We declare Left and Top as lowercase with
        // JsonProperty to sync the client and server models
        [JsonProperty("left")]
        public double Left { get; set; }
        [JsonProperty("top")]
        public double Top { get; set; }
        // We don't want the client to get the "LastUpdatedBy" property
        [JsonIgnore]
        public string LastUpdatedBy { get; set; }
    }
}
```

3. Save the file.

The `MoveShapeHub` class is an implementation of a SignalR hub. As in the Getting Started with SignalR tutorial, the hub has a method that the clients call directly. In this case, the client sends an object with the new X and Y coordinates of the shape to the server. Those coordinates get broadcasted to all other connected clients. SignalR automatically serializes this object using JSON.

The app sends the `ShapeModel` object to the client. It has members to store the position of the shape. The version of the object on the server also has a member to track which client's data is being stored. This object prevents the server from sending a client's data back to itself. This member uses the `JsonIgnore` attribute to keep the application from serializing the data and sending it back to the client.

Map to the hub when app starts

Next, you set up mapping to the hub when the application starts. In SignalR 2, adding an OWIN startup class creates the mapping.

1. In **Solution Explorer**, right-click the project and select **Add** > **New Item**.
2. In **Add New Item - MoveShapeDemo** select **Installed** > **Visual C#** > **Web** and then select **OWIN Startup Class**.
3. Name the class *Startup* and select **OK**.
4. Replace the default code in the *Startup.cs* file with this code:

```
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(MoveShapeDemo.Startup))]
namespace MoveShapeDemo
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Any connection or hub wire up and configuration should go here
            app.MapSignalR();
        }
    }
}
```

The OWIN startup class calls `MapSignalR` when the app executes the `Configuration` method. The app adds the class to OWIN's startup process using the `OwinStartup` assembly attribute.

Add the client

Add the HTML page for the client.

1. In **Solution Explorer**, right-click the project and select **Add** > **HTML Page**.
2. Name the page **Default** and select **OK**.
3. In **Solution Explorer**, right-click *Default.html* and select **Set as Start Page**.
4. Replace the default code in the *Default.html* file with this code:

```
<!DOCTYPE html>
<html>
<head>
    <title>SignalR MoveShape Demo</title>
    <style>
        #shape {
            width: 100px;
            height: 100px;
            background-color: #FF0000;
        }
    </style>
</head>
<body>
<script src="Scripts/jquery-1.10.2.min.js"></script>
<script src="Scripts/jquery-ui-1.10.4.min.js"></script>
<script src="Scripts/jquery.signalR-2.1.0.js"></script>
<script src="/signalr/hubs"></script>
<script>
 $(function () {
            var moveShapeHub = $.connection.moveShapeHub,
            $shape = $("#shape"),
            shapeModel = {
                left: 0,
                top: 0
            };
            moveShapeHub.client.updateShape = function (model) {
                shapeModel = model;
                $shape.css({ left: model.left, top: model.top });
            };
            $.connection.hub.start().done(function () {
                $shape.draggable({
                    drag: function () {
                        shapeModel = $shape.offset();
                        moveShapeHub.server.updateModel(shapeModel);
                    }
                });
            });
        });
</script

    <div id="shape" />
</body>
</html>
```

5. In **Solution Explorer**, expand **Scripts**.

   Script libraries for jQuery and SignalR are visible in the project.

   > **IMPORTANT**
   >
   > The package manager installs a later version of the SignalR scripts.

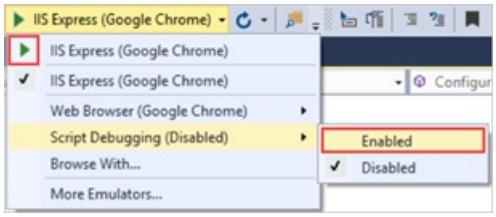
6. Update the script references in the code block to correspond to the versions of the script files in the project.

This HTML and JavaScript code creates a red `div` called `shape`. It enables the shape's dragging behavior using the jQuery library and uses the `drag` event to send the shape's position to the server.

Run the app

You can run the app to se`e it work. When you drag the shape around a browser window, the shape moves in the other browsers too.

1. In the toolbar, turn on **Script Debugging** and then select the play button to run the application in Debug mode.



   A browser window opens with the red shape in the upper-right corner.

2. Copy the page's URL.

3. Open another browser and paste the URL into the address bar.

4. Drag the shape in one of the browser windows. The shape in the other browser window follows.

While the application functions using this method, it's not a recommended programming model. There's no upper limit to the number of messages getting sent. As a result, the clients and server get overwhelmed with messages and performance degrades. Also, the app displays a disjointed animation on the client. This jerky animation happens because the shape moves instantly by each method. It's better if the shape moves smoothly to each new location. Next, you learn how to fix those issues.

Add the client loop

Sending the location of the shape on every mouse move event creates an unnecessary amount of network traffic. The app needs to throttle the messages from the client.

Use the javascript `setInterval` function to set up a loop that sends new position information to the server at a fixed rate. This loop is a basic representation of a "game loop." It's a repeatedly called function that drives all the functionality of a game.

1. Replace the client code in the *Default.html* file with this code:

```
<!DOCTYPE html>
<html>
<head>
<title>SignalR MoveShape Demo</title>
<style>
    #shape {
        width: 100px;
        height: 100px;
        background-color: #FF0000;
    }
</style>
</head>
<body>
<script src="Scripts/jquery-1.10.2.min.js"></script>
<script src="Scripts/jquery-ui-1.10.4.min.js"></script>
<script src="Scripts/jquery.signalR-2.1.0.js"></script>
<script src="/signalr/hubs"></script>
<script>
    $(function () {
        var moveShapeHub = $.connection.moveShapeHub,
            $shape = $("#shape"),
            // Send a maximum of 10 messages per second
            // (mouse movements trigger a lot of messages)
            messageFrequency = 10,
            // Determine how often to send messages in
            // time to abide by the messageFrequency
            updateRate = 1000 / messageFrequency,
            shapeModel = {
                left: 0,
                top: 0
            },
            moved = false;
        moveShapeHub.client.updateShape = function (model) {
            shapeModel = model;
            $shape.css({ left: model.left, top: model.top });
        };
        $.connection.hub.start().done(function () {
            $shape.draggable({
                drag: function () {
                    shapeModel = $shape.offset();
                    moved = true;
                }
            });
            // Start the client side server update interval
            setInterval(updateServerModel, updateRate);
        });
        function updateServerModel() {
            // Only update server if we have a new movement
            if (moved) {
                moveShapeHub.server.updateModel(shapeModel);
                moved = false;
            }
        }
    });
</script>

<div id="shape" />
</body>
</html>
```

IMPORTANT

You have to replace the script references again. They must match the versions of the scripts in the project.

This new code adds the `updateServerModel` function. It gets called on a fixed frequency. The function sends the position data to the server whenever the `moved` flag indicates that there's new position data to send.

2. Select the play button to start the application
3. Copy the page's URL.
4. Open another browser and paste the URL into the address bar.
5. Drag the shape in one of the browser windows. The shape in the other browser window follows.

Since the app throttles the number of messages that get sent to the server, the animation won't appear as smooth did at first.

Add the server loop

In the current application, messages sent from the server to the client go out as often as they're received. This network traffic presents a similar problem as we see on the client.

The app can send messages more often than they're needed. The connection can become flooded as a result. This section describes how to update the server to add a timer that throttles the rate of the outgoing messages.

1. Replace the contents of `MoveShapeHub.cs` with this code:

```
using System;
using System.Threading;
using Microsoft.AspNet.SignalR;
using Newtonsoft.Json;

namespace MoveShapeDemo
{
    public class Broadcaster
    {
        private readonly static Lazy<Broadcaster> _instance =
            new Lazy<Broadcaster>(() => new Broadcaster());
        // We're going to broadcast to all clients a maximum of 25 times per second
        private readonly TimeSpan BroadcastInterval =
            TimeSpan.FromMilliseconds(40);
        private readonly IHubContext _hubContext;
        private Timer _broadcastLoop;
        private ShapeModel _model;
        private bool _modelUpdated;
        public Broadcaster()
        {
            // Save our hub context so we can easily use it
            // to send to its connected clients
            _hubContext = GlobalHost.ConnectionManager.GetHubContext<MoveShapeHub>();
            _model = new ShapeModel();
            _modelUpdated = false;
            // Start the broadcast loop
            _broadcastLoop = new Timer(
                BroadcastShape,
                null,
                BroadcastInterval,
                BroadcastInterval);
        }
        public void BroadcastShape(object state)
        {
            // No need to send anything if our model hasn't changed
            if (_modelUpdated)
            {
                // This is how we can access the Clients property
                // in a static hub method or outside of the hub entirely
                _hubContext.Clients.AllExcept(_model.LastUpdatedBy).updateShape(_model);
                _modelUpdated = false;
```

```
            }
        }
        public void UpdateShape(ShapeModel clientModel)
        {
            _model = clientModel;
            _modelUpdated = true;
        }
        public static Broadcaster Instance
        {
            get
            {
                return _instance.Value;
            }
        }
    }

    public class MoveShapeHub : Hub
    {
        // Is set via the constructor on each creation
        private Broadcaster _broadcaster;
        public MoveShapeHub()
            : this(Broadcaster.Instance)
        {
        }
        public MoveShapeHub(Broadcaster broadcaster)
        {
            _broadcaster = broadcaster;
        }
        public void UpdateModel(ShapeModel clientModel)
        {
            clientModel.LastUpdatedBy = Context.ConnectionId;
            // Update the shape model within our broadcaster
            _broadcaster.UpdateShape(clientModel);
        }
    }
    public class ShapeModel
    {
        // We declare Left and Top as lowercase with
        // JsonProperty to sync the client and server models
        [JsonProperty("left")]
        public double Left { get; set; }
        [JsonProperty("top")]
        public double Top { get; set; }
        // We don't want the client to get the "LastUpdatedBy" property
        [JsonIgnore]
        public string LastUpdatedBy { get; set; }
    }

}
```

2. Select the play button to start the application.
3. Copy the page's URL.
4. Open another browser and paste the URL into the address bar.
5. Drag the shape in one of the browser windows.

This code expands the client to add the `Broadcaster` class. The new class throttles the outgoing messages using the `Timer` class from the .NET framework.

It's good to learn that the hub itself is transitory. It's created every time it's needed. So the app creates the `Broadcaster` as a singleton. It uses lazy initialization to defer the `Broadcaster`'s creation until it's needed. That guarantees that the app creates the first hub instance completely before starting the timer.

The call to the clients' `UpdateShape` function is then moved out of the hub's `UpdateModel` method. It's no longer

called immediately whenever the app receives incoming messages. Instead, the app sends the messages to the clients at a rate of 25 calls per second. The process is managed by the `_broadcastLoop` timer from within the `Broadcaster` class.

Lastly, instead of calling the client method from the hub directly, the `Broadcaster` class needs to get a reference to the currently operating `_hubContext` hub. It gets the reference with the `GlobalHost`.

Add smooth animation

The application is almost finished, but we could make one more improvement. The app moves the shape on the client in response to server messages. Instead of setting the position of the shape to the new location given by the server, use the JQuery UI library's `animate` function. It can move the shape smoothly between its current and new position.

1. Update the client's `updateShape` method in the *Default.html* file to look like the highlighted code:

```
<!DOCTYPE html>
<html>
<head>
    <title>SignalR MoveShape Demo</title>
    <style>
        #shape {
            width: 100px;
            height: 100px;
            background-color: #FF0000;
        }
    </style>
</head>
<body>
<script src="Scripts/jquery-1.10.2.min.js"></script>
<script src="Scripts/jquery-ui-1.10.4.min.js"></script>
<script src="Scripts/jquery.signalR-2.1.0.js"></script>
<script src="/signalr/hubs"></script>
<script>
        $(function () {
            var moveShapeHub = $.connection.moveShapeHub,
                $shape = $("#shape"),
                // Send a maximum of 10 messages per second
                // (mouse movements trigger a lot of messages)
                messageFrequency = 10,
                // Determine how often to send messages in
                // time to abide by the messageFrequency
                updateRate = 1000 / messageFrequency,
                shapeModel = {
                    left: 0,
                    top: 0
                },
                moved = false;
            moveShapeHub.client.updateShape = function (model) {
                 shapeModel = model;
                 // Gradually move the shape towards the new location (interpolate)
                 // The updateRate is used as the duration because by the time
                 // we get to the next location we want to be at the "last" location
                 // We also clear the animation queue so that we start a new
                 // animation and don't lag behind.
                 $shape.animate(shapeModel, { duration: updateRate, queue: false });
            };
            $.connection.hub.start().done(function () {
                $shape.draggable({
                    drag: function () {
                        shapeModel = $shape.offset();
                        moved = true;
                    }
                });
                // Start the client side server update interval
                setInterval(updateServerModel, updateRate);
            });
            function updateServerModel() {
                // Only update server if we have a new movement
                if (moved) {
                    moveShapeHub.server.updateModel(shapeModel);
                    moved = false;
                }
            }
        });
</script>

    <div id="shape" />
</body>
</html>
```

2. Select the play button to start the application.

3. Copy the page's URL.
4. Open another browser and paste the URL into the address bar.
5. Drag the shape in one of the browser windows.

The movement of the shape in the other window appears less jerky. The app interpolates its movement over time rather than being set once per incoming message.

This code moves the shape from the old location to the new one. The server gives the position of the shape over the course of the animation interval. In this case, that's 100 milliseconds. The app clears any previous animation running on the shape before the new animation starts.

Get the code

Download Completed Project

Additional resources

The communication paradigm you just learned about is useful for developing online games and other simulations, like the ShootR game created with SignalR.

For more about SignalR, see the following resources:

- SignalR Project
- SignalR GitHub and Samples
- SignalR Wiki

Next steps

In this tutorial, you:

- Set up the project
- Created the base application
- Mapped to the hub when app starts
- Added the client
- Ran the app
- Added the client loop
- Added the server loop
- Added smooth animation

Advance to the next article to learn how to create a web application that uses ASP.NET SignalR 2 to provide server broadcast functionality.

SignalR 2 and server broadcasting

Tutorial: Server broadcast with SignalR 2

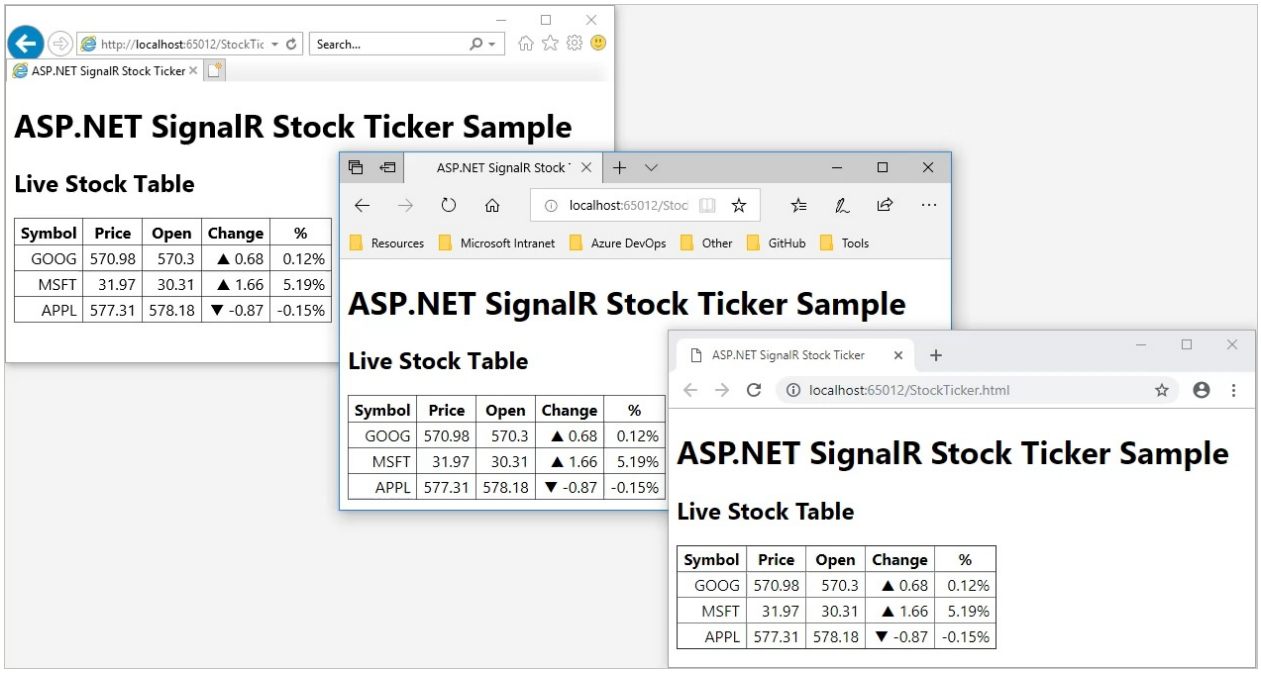
2/20/2020 • 26 minutes to read • Edit Online

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This tutorial shows how to create a web application that uses ASP.NET SignalR 2 to provide server broadcast functionality. Server broadcast means that the server starts the communications sent to clients.

The application that you'll create in this tutorial simulates a stock ticker, a typical scenario for server broadcast functionality. Periodically, the server randomly updates stock prices and broadcast the updates to all connected clients. In the browser, the numbers and symbols in the **Change** and **%** columns dynamically change in response to notifications from the server. If you open additional browsers to the same URL, they all show the same data and the same changes to the data simultaneously.



In this tutorial, you:

- Create the project
- Set up the server code
- Examine the server code
- Set up the client code
- Examine the client code
- Test the application
- Enable logging

> **IMPORTANT**
>
> If you don't want to work through the steps of building the application, you can install the SignalR.Sample package in a new Empty ASP.NET Web Application project. If you install the NuGet package without performing the steps in this tutorial, you must follow the instructions in the *readme.txt* file. To run the package you need to add an OWIN startup class which calls the `ConfigureSignalR` method in the installed package. You will receive an error if you do not add the OWIN startup class. See the Install the StockTicker sample section of this article.

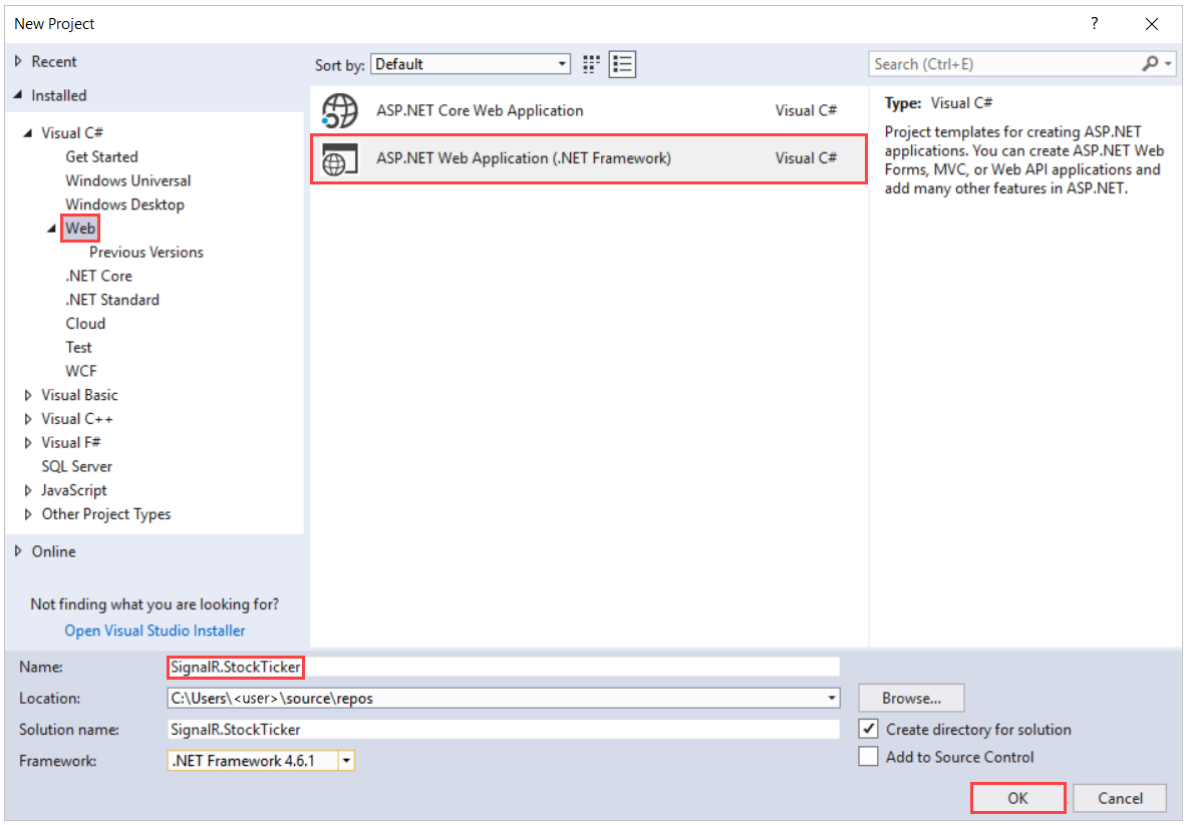
Prerequisites

- Visual Studio 2017 with the **ASP.NET and web development** workload.

Create the project

This section shows how to use Visual Studio 2017 to create an empty ASP.NET Web Application.

1. In Visual Studio, create an ASP.NET Web Application.



2. In the **New ASP.NET Web Application - SignalR.StockTicker** window, leave **Empty** selected and select **OK**.

Set up the server code

In this section, you set up the code that runs on the server.

Create the Stock class

You begin by creating the *Stock* model class that you'll use to store and transmit information about a stock.

1. In **Solution Explorer**, right-click the project and select **Add** > **Class**.

2. Name the class *Stock* and add it to the project.

3. Replace the code in the *Stock.cs* file with this code:

```
using System;

namespace SignalR.StockTicker
{
    public class Stock
    {
        private decimal _price;

        public string Symbol { get; set; }

        public decimal Price
        {
            get
            {
                return _price;
            }
            set
            {
                if (_price == value)
                {
                    return;
                }

                _price = value;

                if (DayOpen == 0)
                {
                    DayOpen = _price;
                }
            }
        }

        public decimal DayOpen { get; private set; }

        public decimal Change
        {
            get
            {
                return Price - DayOpen;
            }
        }

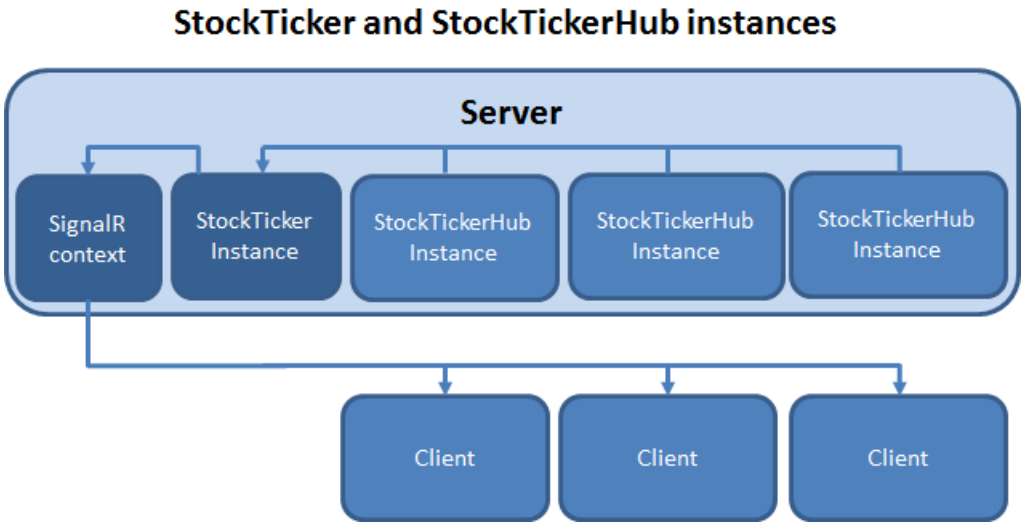
        public double PercentChange
        {
            get
            {
                return (double)Math.Round(Change / Price, 4);
            }
        }
    }
}
```

The two properties that you'll set when you create stocks are `Symbol` (for example, MSFT for Microsoft) and `Price`. The other properties depend on how and when you set `Price`. The first time you set `Price`, the value gets propagated to `DayOpen`. After that, when you set `Price`, the app calculates the `Change` and `PercentChange` property values based on the difference between `Price` and `DayOpen`.

Create the StockTickerHub and StockTicker classes

You'll use the SignalR Hub API to handle server-to-client interaction. A `StockTickerHub` class that derives from the SignalR `Hub` class will handle receiving connections and method calls from clients. You also need to maintain stock data and run a `Timer` object. The `Timer` object will periodically trigger price updates independent of client connections. You can't put these functions in a `Hub` class, because Hubs are transient. The app creates a `Hub` class instance for each task on the hub, like connections and calls from the client to the server. So the mechanism

that keeps stock data, updates prices, and broadcasts the price updates has to run in a separate class. You'll name the class `StockTicker`.



You only want one instance of the `StockTicker` class to run on the server, so you'll need to set up a reference from each `StockTickerHub` instance to the singleton `StockTicker` instance. The `StockTicker` class has to broadcast to clients because it has the stock data and triggers updates, but `StockTicker` isn't a `Hub` class. The `StockTicker` class has to get a reference to the SignalR Hub connection context object. It can then use the SignalR connection context object to broadcast to clients.

Create StockTickerHub.cs

1. In **Solution Explorer**, right-click the project and select **Add** > **New Item**.
2. In **Add New Item - SignalR.StockTicker**, select **Installed** > **Visual C#** > **Web** > **SignalR** and then select **SignalR Hub Class (v2)**.
3. Name the class *StockTickerHub* and add it to the project.

   This step creates the *StockTickerHub.cs* class file. Simultaneously, it adds a set of script files and assembly references that supports SignalR to the project.
4. Replace the code in the *StockTickerHub.cs* file with this code:

```
using System.Collections.Generic;
using Microsoft.AspNet.SignalR;
using Microsoft.AspNet.SignalR.Hubs;

namespace SignalR.StockTicker
{
    [HubName("stockTickerMini")]
    public class StockTickerHub : Hub
    {
        private readonly StockTicker _stockTicker;

        public StockTickerHub() : this(StockTicker.Instance) { }

        public StockTickerHub(StockTicker stockTicker)
        {
            _stockTicker = stockTicker;
        }

        public IEnumerable<Stock> GetAllStocks()
        {
            return _stockTicker.GetAllStocks();
        }
    }
}
```

5. Save the file.

The app uses the Hub class to define methods the clients can call on the server. You're defining one method: `GetAllStocks()`. When a client initially connects to the server, it will call this method to get a list of all of the stocks with their current prices. The method can run synchronously and return `IEnumerable<Stock>` because it's returning data from memory.

If the method had to get the data by doing something that would involve waiting, like a database lookup or a web service call, you would specify `Task<IEnumerable<Stock>>` as the return value to enable asynchronous processing. For more information, see ASP.NET SignalR Hubs API Guide - Server - When to execute asynchronously.

The `HubName` attribute specifies how the app will reference the Hub in JavaScript code on the client. The default name on the client if you don't use this attribute, is a camelCase version of the class name, which in this case would be `stockTickerHub`.

As you'll see later when you create the `StockTicker` class, the app creates a singleton instance of that class in its static `Instance` property. That singleton instance of `StockTicker` is in memory no matter how many clients connect or disconnect. That instance is what the `GetAllStocks()` method uses to return current stock information.

Create StockTicker.cs

1. In **Solution Explorer**, right-click the project and select **Add** > **Class**.

2. Name the class *StockTicker* and add it to the project.

3. Replace the code in the *StockTicker.cs* file with this code:

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Threading;
using Microsoft.AspNet.SignalR;
using Microsoft.AspNet.SignalR.Hubs;

namespace SignalR.StockTicker
{
    public class StockTicker
    {
```

```
        // Singleton instance
        private readonly static Lazy<StockTicker> _instance = new Lazy<StockTicker>(() => new
StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>().Clients));

        private readonly ConcurrentDictionary<string, Stock> _stocks = new
ConcurrentDictionary<string, Stock>();

        private readonly object _updateStockPricesLock = new object();

        //stock can go up or down by a percentage of this factor on each change
        private readonly double _rangePercent = .002;

        private readonly TimeSpan _updateInterval = TimeSpan.FromMilliseconds(250);
        private readonly Random _updateOrNotRandom = new Random();

        private readonly Timer _timer;
        private volatile bool _updatingStockPrices = false;

        private StockTicker(IHubConnectionContext<dynamic> clients)
        {
            Clients = clients;

            _stocks.Clear();
            var stocks = new List<Stock>
            {
                new Stock { Symbol = "MSFT", Price = 30.31m },
                new Stock { Symbol = "APPL", Price = 578.18m },
                new Stock { Symbol = "GOOG", Price = 570.30m }
            };
            stocks.ForEach(stock => _stocks.TryAdd(stock.Symbol, stock));

            _timer = new Timer(UpdateStockPrices, null, _updateInterval, _updateInterval);

        }

        public static StockTicker Instance
        {
            get
            {
                return _instance.Value;
            }
        }

        private IHubConnectionContext<dynamic> Clients
        {
            get;
            set;
        }

        public IEnumerable<Stock> GetAllStocks()
        {
            return _stocks.Values;
        }

        private void UpdateStockPrices(object state)
        {
            lock (_updateStockPricesLock)
            {
                if (!_updatingStockPrices)
                {
                    _updatingStockPrices = true;

                    foreach (var stock in _stocks.Values)
                    {
                        if (TryUpdateStockPrice(stock))
                        {
                            BroadcastStockPrice(stock);
                        }
                    }
```

```
                _updatingStockPrices = false;
            }
        }
    }

    private bool TryUpdateStockPrice(Stock stock)
    {
        // Randomly choose whether to update this stock or not
        var r = _updateOrNotRandom.NextDouble();
        if (r > .1)
        {
            return false;
        }

        // Update the stock price by a random factor of the range percent
        var random = new Random((int)Math.Floor(stock.Price));
        var percentChange = random.NextDouble() * _rangePercent;
        var pos = random.NextDouble() > .51;
        var change = Math.Round(stock.Price * (decimal)percentChange, 2);
        change = pos ? change : -change;

        stock.Price += change;
        return true;
    }

    private void BroadcastStockPrice(Stock stock)
    {
        Clients.All.updateStockPrice(stock);
    }

  }
}
```

Since all threads will be running the same instance of StockTicker code, the StockTicker class has to be thread-safe.

Examine the server code

If you examine the server code, it will help you understand how the app works.

Storing the singleton instance in a static field

The code initializes the static `_instance` field that backs the `Instance` property with an instance of the class. Because the constructor is private, it's the only instance of the class that the app can create. The app uses Lazy initialization for the `_instance` field. It's not for performance reasons. It's to make sure the instance creation is thread-safe.

```
private readonly static Lazy<StockTicker> _instance = new Lazy<StockTicker>(() => new
StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>().Clients));

public static StockTicker Instance
{
    get
    {
        return _instance.Value;
    }
}
```

Each time a client connects to the server, a new instance of the StockTickerHub class running in a separate thread gets the StockTicker singleton instance from the `StockTicker.Instance` static property, as you saw earlier in the `StockTickerHub` class.

Storing stock data in a ConcurrentDictionary

The constructor initializes the `_stocks` collection with some sample stock data, and `GetAllStocks` returns the stocks. As you saw earlier, this collection of stocks is returned by `StockTickerHub.GetAllStocks`, which is a server method in the `Hub` class that clients can call.

```
private readonly ConcurrentDictionary<string, Stock> _stocks = new ConcurrentDictionary<string, Stock>();
```

```
private StockTicker(IHubConnectionContext<dynamic> clients)
{
    Clients = clients;

    _stocks.Clear();
    var stocks = new List<Stock>
    {
        new Stock { Symbol = "MSFT", Price = 30.31m },
        new Stock { Symbol = "APPL", Price = 578.18m },
        new Stock { Symbol = "GOOG", Price = 570.30m }
    };
    stocks.ForEach(stock => _stocks.TryAdd(stock.Symbol, stock));

    _timer = new Timer(UpdateStockPrices, null, _updateInterval, _updateInterval);
}

public IEnumerable<Stock> GetAllStocks()
{
    return _stocks.Values;
}
```

The stocks collection is defined as a ConcurrentDictionary type for thread safety. As an alternative, you could use a Dictionary object and explicitly lock the dictionary when you make changes to it.

For this sample application, it's OK to store application data in memory and to lose the data when the app disposes of the `StockTicker` instance. In a real application, you would work with a back-end data store like a database.

Periodically updating stock prices

The constructor starts up a `Timer` object that periodically calls methods that update stock prices on a random basis.

```
_timer = new Timer(UpdateStockPrices, null, _updateInterval, _updateInterval);

private void UpdateStockPrices(object state)
{
    lock (_updateStockPricesLock)
    {
        if (!_updatingStockPrices)
        {
            _updatingStockPrices = true;

            foreach (var stock in _stocks.Values)
            {
                if (TryUpdateStockPrice(stock))
                {
                    BroadcastStockPrice(stock);
                }
            }

            _updatingStockPrices = false;
        }
    }
}

private bool TryUpdateStockPrice(Stock stock)
{
    // Randomly choose whether to update this stock or not
    var r = _updateOrNotRandom.NextDouble();
    if (r > .1)
    {
        return false;
    }

    // Update the stock price by a random factor of the range percent
    var random = new Random((int)Math.Floor(stock.Price));
    var percentChange = random.NextDouble() * _rangePercent;
    var pos = random.NextDouble() > .51;
    var change = Math.Round(stock.Price * (decimal)percentChange, 2);
    change = pos ? change : -change;

    stock.Price += change;
    return true;
}
```

`Timer` calls `UpdateStockPrices`, which passes in null in the state parameter. Before updating prices, the app takes a lock on the `_updateStockPricesLock` object. The code checks if another thread is already updating prices, and then it calls `TryUpdateStockPrice` on each stock in the list. The `TryUpdateStockPrice` method decides whether to change the stock price, and how much to change it. If the stock price changes, the app calls `BroadcastStockPrice` to broadcast the stock price change to all connected clients.

The `_updatingStockPrices` flag designated volatile to make sure it is thread-safe.

```
private volatile bool _updatingStockPrices = false;
```

In a real application, the `TryUpdateStockPrice` method would call a web service to look up the price. In this code, the app uses a random number generator to make changes randomly.

Getting the SignalR context so that the StockTicker class can broadcast to clients

Because the price changes originate here in the `StockTicker` object, it's the object that needs to call an `updateStockPrice` method on all connected clients. In a `Hub` class, you have an API for calling client methods, but `StockTicker` doesn't derive from the `Hub` class and doesn't have a reference to any `Hub` object. To broadcast to connected clients, the `StockTicker` class has to get the SignalR context instance for the `StockTickerHub` class and

use that to call methods on clients.

The code gets a reference to the SignalR context when it creates the singleton class instance, passes that reference to the constructor, and the constructor puts it in the `Clients` property.

There are two reasons why you want to get the context only once: getting the context is an expensive task, and getting it once makes sure the app preserves the intended order of messages sent to the clients.

```
private readonly static Lazy<StockTicker> _instance =
    new Lazy<StockTicker>(() => new StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>
().Clients));

private StockTicker(IHubConnectionContext<dynamic> clients)
{
    Clients = clients;

    // Remainder of constructor ...
}

private IHubConnectionContext<dynamic> Clients
{
    get;
    set;
}

private void BroadcastStockPrice(Stock stock)
{
    Clients.All.updateStockPrice(stock);
}
```

Getting the `Clients` property of the context and putting it in the `StockTickerClient` property lets you write code to call client methods that looks the same as it would in a `Hub` class. For instance, to broadcast to all clients you can write `Clients.All.updateStockPrice(stock)`.

The `updateStockPrice` method that you're calling in `BroadcastStockPrice` doesn't exist yet. You'll add it later when you write code that runs on the client. You can refer to `updateStockPrice` here because `Clients.All` is dynamic, which means the app will evaluate the expression at runtime. When this method call executes, SignalR will send the method name and the parameter value to the client, and if the client has a method named `updateStockPrice`, the app will call that method and pass the parameter value to it.

`Clients.All` means send to all clients. SignalR gives you other options to specify which clients or groups of clients to send to. For more information, see HubConnectionContext.

Register the SignalR route

The server needs to know which URL to intercept and direct to SignalR. To do that, add an OWIN startup class:

1. In **Solution Explorer**, right-click the project and select **Add** > **New Item**.
2. In **Add New Item - SignalR.StockTicker** select **Installed** > **Visual C#** > **Web** and then select **OWIN Startup Class**.
3. Name the class *Startup* and select **OK**.
4. Replace the default code in the *Startup.cs* file with this code:

```
using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(SignalR.StockTicker.Startup))]

namespace SignalR.StockTicker
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Any connection or hub wire up and configuration should go here
            app.MapSignalR();
        }

    }
}
```

You have now finished setting up the server code. In the next section, you'll set up the client.

Set up the client code

In this section, you set up the code that runs on the client.

Create the HTML page and JavaScript file

The HTML page will display the data and the JavaScript file will organize the data.

Create StockTicker.html

First, you'll add the HTML client.

1. In **Solution Explorer**, right-click the project and select **Add** > **HTML Page**.
2. Name the file *StockTicker* and select **OK**.
3. Replace the default code in the *StockTicker.html* file with this code:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>ASP.NET SignalR Stock Ticker</title>
    <style>
        body {
            font-family: 'Segoe UI', Arial, Helvetica, sans-serif;
            font-size: 16px;
        }
        #stockTable table {
            border-collapse: collapse;
        }
            #stockTable table th, #stockTable table td {
                padding: 2px 6px;
            }
            #stockTable table td {
                text-align: right;
            }
        #stockTable .loading td {
            text-align: left;
        }
    </style>
</head>
<body>
    <h1>ASP.NET SignalR Stock Ticker Sample</h1>

    <h2>Live Stock Table</h2>
    <div id="stockTable">
        <table border="1">
            <thead>
                <tr><th>Symbol</th><th>Price</th><th>Open</th><th>Change</th><th>%</th></tr>
            </thead>
            <tbody>
                <tr class="loading"><td colspan="5">loading...</td></tr>
            </tbody>
        </table>
    </div>

    <!--Script references. -->
    <!--Reference the jQuery library. -->
    <script src="/Scripts/jquery-1.10.2.min.js" ></script>
    <!--Reference the SignalR library. -->
    <script src="/Scripts/jquery.signalR-2.1.0.js"></script>
    <!--Reference the autogenerated SignalR hub script. -->
    <script src="/signalr/hubs"></script>
    <!--Reference the StockTicker script. -->
    <script src="StockTicker.js"></script>
</body>
</html>
```

The HTML creates a table with five columns, a header row, and a data row with a single cell that spans all five columns. The data row shows "loading..." momentarily when the app starts. JavaScript code will remove that row and add in its place rows with stock data retrieved from the server.

The script tags specify:

- The jQuery script file.
- The SignalR core script file.
- The SignalR proxies script file.
- A StockTicker script file that you'll create later.

The app dynamically generates the SignalR proxies script file. It specifies the "/signalr/hubs" URL and

defines proxy methods for the methods on the Hub class, in this case, for `StockTickerHub.GetAllStocks`. If you prefer, you can generate this JavaScript file manually by using SignalR Utilities. Don't forget to disable dynamic file creation in the `MapHubs` method call.

4. In **Solution Explorer**, expand **Scripts**.

   Script libraries for jQuery and SignalR are visible in the project.

   > **IMPORTANT**
   >
   > The package manager will install a later version of the SignalR scripts.

5. Update the script references in the code block to correspond to the versions of the script files in the project.

6. In **Solution Explorer**, right-click *StockTicker.html*, and then select **Set as Start Page**.

Create StockTicker.js

Now create the JavaScript file.

1. In **Solution Explorer**, right-click the project and select **Add** > **JavaScript File**.

2. Name the file *StockTicker* and select **OK**.

3. Add this code to the *StockTicker.js* file:

```
// A simple templating method for replacing placeholders enclosed in curly braces.
if (!String.prototype.supplant) {
    String.prototype.supplant = function (o) {
        return this.replace(/{([^{}]*)}/g,
            function (a, b) {
                var r = o[b];
                return typeof r === 'string' || typeof r === 'number' ? r : a;
            }
        );
    };
}

$(function () {

    var ticker = $.connection.stockTickerMini, // the generated client-side hub proxy
        up = '▲',
        down = '▼',
        $stockTable = $('#stockTable'),
        $stockTableBody = $stockTable.find('tbody'),
        rowTemplate = '<tr data-symbol="{Symbol}"><td>{Symbol}</td><td>{Price}</td><td>{DayOpen}</td>
<td>{Direction} {Change}</td><td>{PercentChange}</td></tr>';

    function formatStock(stock) {
        return $.extend(stock, {
            Price: stock.Price.toFixed(2),
            PercentChange: (stock.PercentChange * 100).toFixed(2) + '%',
            Direction: stock.Change === 0 ? '' : stock.Change >= 0 ? up : down
        });
    }

    function init() {
        ticker.server.getAllStocks().done(function (stocks) {
            $stockTableBody.empty();
            $.each(stocks, function () {
                var stock = formatStock(this);
                $stockTableBody.append(rowTemplate.supplant(stock));
            });
        });
    }

    // Add a client-side hub method that the server will call
    ticker.client.updateStockPrice = function (stock) {
        var displayStock = formatStock(stock),
            $row = $(rowTemplate.supplant(displayStock));

        $stockTableBody.find('tr[data-symbol=' + stock.Symbol + ']')
            .replaceWith($row);
        }

    // Start the connection
    $.connection.hub.start().done(init);

});
```

Examine the client code

If you examine the client code, it will help you learn how the client code interacts with the server code to make the app work.

Starting the connection

`$.connection` refers to the SignalR proxies. The code gets a reference to the proxy for the `StockTickerHub` class and puts it in the `ticker` variable. The proxy name is the name that was set by the `HubName` attribute:

```
var ticker = $.connection.stockTickerMini
```

```
[HubName("stockTickerMini")]
public class StockTickerHub : Hub
```

After you define all the variables and functions, the last line of code in the file initializes the SignalR connection by calling the SignalR `start` function. The `start` function executes asynchronously and returns a jQuery Deferred object. You can call the done function to specify the function to call when the app finishes the asynchronous action.

```
$.connection.hub.start().done(init);
```

Getting all the stocks

The `init` function calls the `getAllStocks` function on the server and uses the information that the server returns to update the stock table. Notice that, by default, you have to use camelCasing on the client even though the method name is pascal-cased on the server. The camelCasing rule only applies to methods, not objects. For example, you refer to `stock.Symbol` and `stock.Price`, not `stock.symbol` or `stock.price`.

```
function init() {
    ticker.server.getAllStocks().done(function (stocks) {
        $stockTableBody.empty();
        $.each(stocks, function () {
            var stock = formatStock(this);
            $stockTableBody.append(rowTemplate.supplant(stock));
        });
    });
}
```

```
public IEnumerable<Stock> GetAllStocks()
{
    return _stockTicker.GetAllStocks();
}
```

In the `init` method, the app creates HTML for a table row for each stock object received from the server by calling `formatStock` to format properties of the `stock` object, and then by calling `supplant` to replace placeholders in the `rowTemplate` variable with the `stock` object property values. The resulting HTML is then appended to the stock table.

> **NOTE**
> You call `init` by passing it in as a `callback` function that executes after the asynchronous `start` function finishes. If you called `init` as a separate JavaScript statement after calling `start`, the function would fail because it would run immediately without waiting for the start function to finish establishing the connection. In that case, the `init` function would try to call the `getAllStocks` function before the app establishes a server connection.

Getting updated stock prices

When the server changes a stock's price, it calls the `updateStockPrice` on connected clients. The app adds the function to the client property of the `stockTicker` proxy to make it available to calls from the server.

```
ticker.client.updateStockPrice = function (stock) {
    var displayStock = formatStock(stock),
        $row = $(rowTemplate.supplant(displayStock));

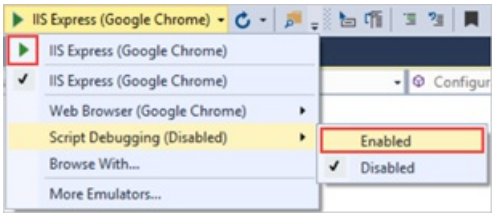
    $stockTableBody.find('tr[data-symbol=' + stock.Symbol + ']')
        .replaceWith($row);
    }
```

The `updateStockPrice` function formats a stock object received from the server into a table row the same way as in the `init` function. Instead of appending the row to the table, it finds the stock's current row in the table and replaces that row with the new one.

Test the application

You can test the app to make sure it's working. You'll see all browser windows display the live stock table with stock prices fluctuating.

1. In the toolbar, turn on **Script Debugging** and then select the play button to run the app in Debug mode.



   A browser window will open displaying the **Live Stock Table**. The stock table initially shows the "loading..." line, then, after a short time, the app shows the initial stock data, and then the stock prices start to change.

2. Copy the URL from the browser, open two other browsers, and paste the URLs into the address bars.

   The initial stock display is the same as the first browser and changes happen simultaneously.

3. Close all browsers, open a new browser, and go to the same URL.

   The StockTicker singleton object continued to run in the server. The **Live Stock Table** shows that the stocks have continued to change. You don't see the initial table with zero change figures.

4. Close the browser.

Enable logging

SignalR has a built-in logging function that you can enable on the client to aid in troubleshooting. In this section, you enable logging and see examples that show how logs tell you which of the following transport methods SignalR is using:

- WebSockets, supported by IIS 8 and current browsers.
- Server-sent events, supported by browsers other than Internet Explorer.
- Forever frame, supported by Internet Explorer.
- Ajax long polling, supported by all browsers.

For any given connection, SignalR chooses the best transport method that both the server and the client support.

1. Open *StockTicker.js*.
2. Add this highlighted line of code to enable logging immediately before the code that initializes the

connection at the end of the file:

```
// Start the connection
$.connection.hub.logging = true;
$.connection.hub.start().done(init);
```

3. Press **F5** to run the project.

4. Open your browser's developer tools window, and select the Console to see the logs. You might have to refresh the page to see the logs of SignalR negotiating the transport method for a new connection.

   - If you're running Internet Explorer 10 on Windows 8 (IIS 8), the transport method is **WebSockets**.
   - If you're running Internet Explorer 10 on Windows 7 (IIS 7.5), the transport method is **iframe**.
   - If you're running Firefox 19 on Windows 8 (IIS 8), the transport method is **WebSockets**.

     > **TIP**
     > In Firefox, install the Firebug add-in to get a Console window.

   - If you're running Firefox 19 on Windows 7 (IIS 7.5), the transport method is **server-sent** events.

Install the StockTicker sample

The Microsoft.AspNet.SignalR.Sample installs the StockTicker application. The NuGet package includes more features than the simplified version that you created from scratch. In this section of the tutorial, you install the NuGet package and review the new features and the code that implements them.

> **IMPORTANT**
> If you install the package without performing the earlier steps of this tutorial, you must add an OWIN startup class to your project. This readme.txt file for the NuGet package explains this step.

Install the SignalR.Sample NuGet package

1. In **Solution Explorer**, right-click the project and select **Manage NuGet Packages**.

2. In **NuGet Package manager: SignalR.StockTicker**, select **Browse**.

3. From **Package source**, select **nuget.org**.

4. Enter *SignalR.Sample* in the search box and select **Microsoft.AspNet.SignalR.Sample** > **Install**.

5. In **Solution Explorer**, expand the *SignalR.Sample* folder.

   Installing the SignalR.Sample package created the folder and its contents.

6. In the *SignalR.Sample* folder, right-click *StockTicker.html*, and then select **Set As Start Page**.

   > **NOTE**
   > Installing The SignalR.Sample NuGet package might change the version of jQuery that you have in your *Scripts* folder. The new *StockTicker.html* file that the package installs in the *SignalR.Sample* folder will be in sync with the jQuery version that the package installs, but if you want to run your original *StockTicker.html* file again, you might have to update the jQuery reference in the script tag first.

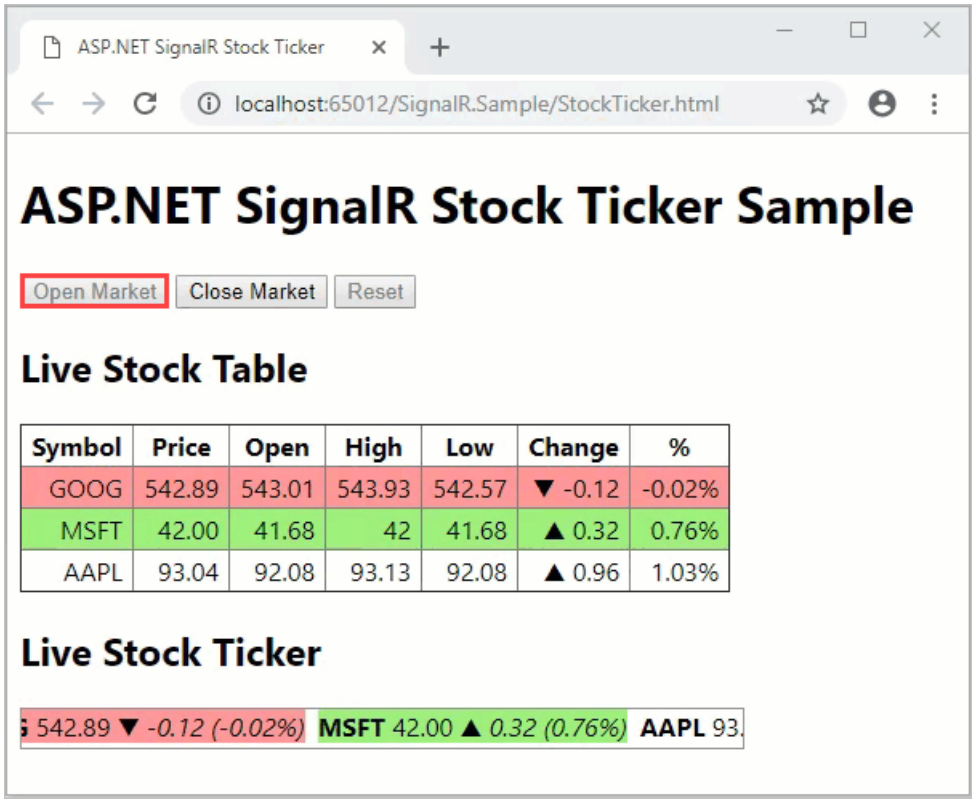
Run the application

The table that you saw in the first app had useful features. The full stock ticker application shows new features: a horizontally scrolling window that shows the stock data and stocks that change color as they rise and fall.

1. Press **F5** to run the app.

   When you run the app for the first time, the "market" is "closed" and you see a static table and a ticker window that isn't scrolling.

2. Select **Open Market**.



   - The **Live Stock Ticker** box starts to scroll horizontally, and the server starts to periodically broadcast stock price changes on a random basis.
   - Each time a stock price changes, the app updates both the **Live Stock Table** and the **Live Stock Ticker**.
   - When a stock's price change is positive, the app shows the stock with a green background.
   - When the change is negative, the app shows the stock with a red background.

3. Select **Close Market**.
   - The table updates stop.
   - The ticker stops scrolling.

4. Select **Reset**.
   - All stock data is reset.
   - The app restores the initial state before price changes started.

5. Copy the URL from the browser, open two other browsers, and paste the URLs into the address bars.

6. You see the same data dynamically updated at the same time in each browser.

7. When you select any of the controls, all browsers respond the same way at the same time.

Live Stock Ticker display

The **Live Stock Ticker** display is an unordered list in a `<div>` element formatted into a single line by CSS styles. The app initializes and updates the ticker the same way as the table: by replacing placeholders in an `<li>` template string and dynamically adding the `<li>` elements to the `<ul>` element. The app includes scrolling by using the jQuery `animate` function to vary the margin-left of the unordered list within the `<div>`.

SignalR.Sample StockTicker.html

The stock ticker HTML code:

```
<h2>Live Stock Ticker</h2>
<div id="stockTicker">
    <div class="inner">
        <ul>
            <li class="loading">loading...</li>
        </ul>
    </div>
</div>
```

SignalR.Sample StockTicker.css

The stock ticker CSS code:

```
#stockTicker {
    overflow: hidden;
    width: 450px;
    height: 24px;
    border: 1px solid #999;
    }

    #stockTicker .inner {
        width: 9999px;
    }

    #stockTicker ul {
        display: inline-block;
        list-style-type: none;
        margin: 0;
        padding: 0;
    }

    #stockTicker li {
        display: inline-block;
        margin-right: 8px;
    }

    /*<li data-symbol="{Symbol}"><span class="symbol">{Symbol}</span><span class="price">{Price}</span><span
class="change">{PercentChange}</span></li>*/
    #stockTicker .symbol {
        font-weight: bold;
    }

    #stockTicker .change {
        font-style: italic;
    }
```

SignalR.Sample SignalR.StockTicker.js

The jQuery code that makes it scroll:

```
function scrollTicker() {
    var w = $stockTickerUl.width();
    $stockTickerUl.css({ marginLeft: w });
    $stockTickerUl.animate({ marginLeft: -w }, 15000, 'linear', scrollTicker);
}
```

Additional methods on the server that the client can call

To add flexibility to the app, there are additional methods the app can call.

SignalR.Sample StockTickerHub.cs

The `StockTickerHub` class defines four additional methods that the client can call:

```
public string GetMarketState()
{
    return _stockTicker.MarketState.ToString();
}

public void OpenMarket()
{
    _stockTicker.OpenMarket();
}

public void CloseMarket()
{
    _stockTicker.CloseMarket();
}

public void Reset()
{
    _stockTicker.Reset();
}
```

The app calls `OpenMarket`, `CloseMarket`, and `Reset` in response to the buttons at the top of the page. They demonstrate the pattern of one client triggering a change in state immediately propagated to all clients. Each of these methods calls a method in the `StockTicker` class that causes the market state change and then broadcasts the new state.

SignalR.Sample StockTicker.cs

In the `StockTicker` class, the app maintains the state of the market with a `MarketState` property that returns a `MarketState` enum value:

```
public MarketState MarketState
{
    get { return _marketState; }
    private set { _marketState = value; }
}

public enum MarketState
{
    Closed,
    Open
}
```

Each of the methods that change the market state do so inside a lock block because the `StockTicker` class has to be thread-safe:

```
public void OpenMarket()
{
    lock (_marketStateLock)
    {
        if (MarketState != MarketState.Open)
        {
            _timer = new Timer(UpdateStockPrices, null, _updateInterval, _updateInterval);
            MarketState = MarketState.Open;
            BroadcastMarketStateChange(MarketState.Open);
        }
    }
}

public void CloseMarket()
{
    lock (_marketStateLock)
    {
        if (MarketState == MarketState.Open)
        {
            if (_timer != null)
            {
                _timer.Dispose();
            }
            MarketState = MarketState.Closed;
            BroadcastMarketStateChange(MarketState.Closed);
        }
    }
}

public void Reset()
{
    lock (_marketStateLock)
    {
        if (MarketState != MarketState.Closed)
        {
            throw new InvalidOperationException("Market must be closed before it can be reset.");
        }
        LoadDefaultStocks();
        BroadcastMarketReset();
    }
}
```

To make sure this code is thread-safe, the `_marketState` field that backs the `MarketState` property designated `volatile`:

```
private volatile MarketState _marketState;
```

The `BroadcastMarketStateChange` and `BroadcastMarketReset` methods are similar to the BroadcastStockPrice method that you already saw, except they call different methods defined at the client:

```
private void BroadcastMarketStateChange(MarketState marketState)
{
    switch (marketState)
    {
        case MarketState.Open:
            Clients.All.marketOpened();
            break;
        case MarketState.Closed:
            Clients.All.marketClosed();
            break;
        default:
            break;
    }
}

private void BroadcastMarketReset()
{
    Clients.All.marketReset();
}
```

Additional functions on the client that the server can call

The `updateStockPrice` function now handles both the table and the ticker display, and it uses `jQuery.Color` to flash red and green colors.

New functions in *SignalR.StockTicker.js* enable and disable the buttons based on market state. They also stop or start the **Live Stock Ticker** horizontal scrolling. Since many functions are being added to `ticker.client`, the app uses the jQuery extend function to add them.

```
$.extend(ticker.client, {
    updateStockPrice: function (stock) {
        var displayStock = formatStock(stock),
            $row = $(rowTemplate.supplant(displayStock)),
            $li = $(liTemplate.supplant(displayStock)),
            bg = stock.LastChange === 0
                ? '255,216,0' // yellow
                : stock.LastChange > 0
                    ? '154,240,117' // green
                    : '255,148,148'; // red

        $stockTableBody.find('tr[data-symbol=' + stock.Symbol + ']')
            .replaceWith($row);
        $stockTickerUl.find('li[data-symbol=' + stock.Symbol + ']')
            .replaceWith($li);

        $row.flash(bg, 1000);
        $li.flash(bg, 1000);
    },

    marketOpened: function () {
        $("#open").prop("disabled", true);
        $("#close").prop("disabled", false);
        $("#reset").prop("disabled", true);
        scrollTicker();
    },

    marketClosed: function () {
        $("#open").prop("disabled", false);
        $("#close").prop("disabled", true);
        $("#reset").prop("disabled", false);
        stopTicker();
    },

    marketReset: function () {
        return init();
    }
});
```

Additional client setup after establishing the connection

After the client establishes the connection, it has some additional work to do:

- Find out if the market is open or closed to call the appropriate `marketOpened` or `marketClosed` function.
- Attach the server method calls to the buttons.

```
$.connection.hub.start()
    .pipe(init)
    .pipe(function () {
        return ticker.server.getMarketState();
    })
    .done(function (state) {
        if (state === 'Open') {
            ticker.client.marketOpened();
        } else {
            ticker.client.marketClosed();
        }

        // Wire up the buttons
        $("#open").click(function () {
            ticker.server.openMarket();
        });

        $("#close").click(function () {
            ticker.server.closeMarket();
        });

        $("#reset").click(function () {
            ticker.server.reset();
        });
    });
```

The server methods aren't wired up to the buttons until after the app establishes the connection. It's so the code can't call the server methods before they're available.

Additional resources

In this tutorial you've learned how to program a SignalR application that broadcasts messages from the server to all connected clients. Now you can broadcast messages on a periodic basis and in response to notifications from any client. You can use the concept of multi-threaded singleton instance to maintain server state in multi-player online game scenarios. For an example, see the ShootR game based on SignalR.

For tutorials that show peer-to-peer communication scenarios, see Getting Started with SignalR and Real-Time Updating with SignalR.

For more about SignalR, see the following resources:

- ASP.NET SignalR
- SignalR Project
- SignalR GitHub and Samples
- SignalR Wiki

Next steps

In this tutorial, you:

- Created the project
- Set up the server code
- Examined the server code
- Set up the client code
- Examined the client code
- Tested the application
- Enabled logging

Advance to the next article to learn how to create a real-time web application that uses ASP.NET SignalR 2.

Create real-time web app with SignalR

Hands On Lab: Real-Time Web Applications with SignalR

2/20/2020 • 17 minutes to read • Edit Online

by Web Camps Team

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

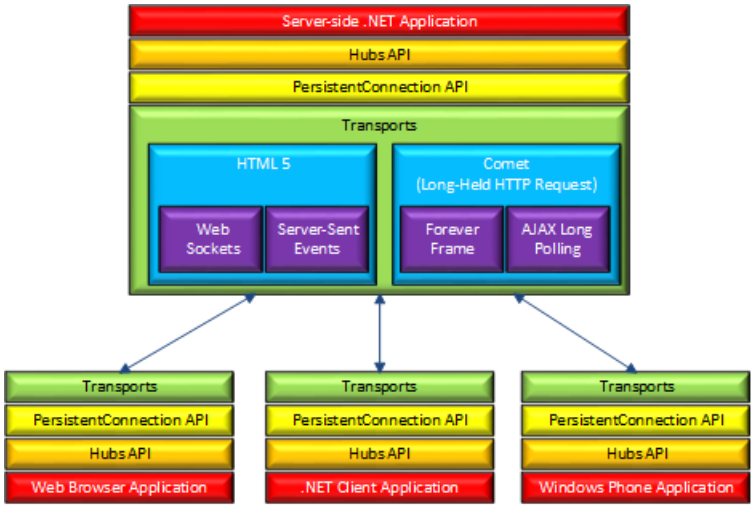
Download Web Camps Training Kit, October 2015 Release

Real-time Web applications feature the ability to push server-side content to the connected clients as it happens, in real-time. For ASP.NET developers, **ASP.NET SignalR** is a library to add real-time web functionality to their applications. It takes advantage of several transports, automatically selecting the best available transport given the client and server's best available transport. It takes advantage of **WebSocket**, an HTML5 API that enables bi-directional communication between the browser and server.

SignalR also provides a simple, high-level API for doing server to client RPC (call JavaScript functions in your clients' browsers from server-side .NET code) in your ASP.NET application, as well as adding useful hooks for connection management, such as connect/disconnect events, grouping connections, and authorization.

SignalR is an abstraction over some of the transports that are required to do real-time work between client and server. A **SignalR** connection starts as HTTP, and is then promoted to a **WebSocket** connection if available. **WebSocket** is the ideal transport for **SignalR**, since it makes the most efficient use of server memory, has the lowest latency, and has the most underlying features (such as full duplex communication between client and server), but it also has the most stringent requirements: **WebSocket** requires the server to be using **Windows Server 2012** or **Windows 8**, along with **.NET Framework 4.5**. If these requirements are not met, **SignalR** will attempt to use other transports to make its connections (like *Ajax long polling*).

The **SignalR** API contains two models for communicating between clients and servers: **Persistent Connections** and **Hubs**. A **Connection** represents a simple endpoint for sending single-recipient, grouped, or broadcast messages. A **Hub** is a more high-level pipeline built upon the Connection API that allows your client and server to call methods on each other directly.



All sample code and snippets are included in the Web Camps Training Kit, October 2015 Release, available at

> https://github.com/Microsoft-Web/WebCampTrainingKit/releases/tag/v2015.10.13b. Please note that the Installer link on that page no longer works; use one of the links under the Assets section instead.

Overview

Objectives

In this hands-on lab, you will learn how to:

- Send notifications from server to client using SignalR.
- Scale Out your SignalR application using **SQL Server**.

Prerequisites

The following is required to complete this hands-on lab:

- Visual Studio Express 2013 for Web or greater

Setup

In order to run the exercises in this hands-on lab, you will need to set up your environment first.

1. Open a Windows Explorer window and browse to the lab's **Source** folder.
2. Right-click **Setup.cmd** and select **Run as administrator** to launch the setup process that will configure your environment and install the Visual Studio code snippets for this lab.
3. If the User Account Control dialog box is shown, confirm the action to proceed.

> **NOTE**
> Make sure you have checked all the dependencies for this lab before running the setup.

Using the Code Snippets

Throughout the lab document, you will be instructed to insert code blocks. For your convenience, most of this code is provided as Visual Studio Code Snippets, which you can access from within Visual Studio 2013 to avoid having to add it manually.

> **NOTE**
> Each exercise is accompanied by a starting solution located in the **Begin** folder of the exercise that allows you to follow each exercise independently of the others. Please be aware that the code snippets that are added during an exercise are missing from these starting solutions and may not work until you have completed the exercise. Inside the source code for an exercise, you will also find an **End** folder containing a Visual Studio solution with the code that results from completing the steps in the corresponding exercise. You can use these solutions as guidance if you need additional help as you work through this hands-on lab.

Exercises

This hands-on lab includes the following exercises:

1. Working with Real-Time Data Using SignalR
2. Scaling Out Using SQL Server

Estimated time to complete this lab: **60 minutes**

> **NOTE**
> When you first start Visual Studio, you must select one of the predefined settings collections. Each predefined collection is designed to match a particular development style and determines window layouts, editor behavior, IntelliSense code snippets, and dialog box options. The procedures in this lab describe the actions necessary to accomplish a given task in Visual Studio when using the **General Development Settings** collection. If you choose a different settings collection for your development environment, there may be differences in the steps that you should take into account.

Exercise 1: Working with Real-Time Data Using SignalR

While chat is often used as an example, you can do a whole lot more with real-time Web functionality. Any time a user refreshes a web page to see new data or the page implements Ajax long polling to retrieve new data, you can use SignalR.

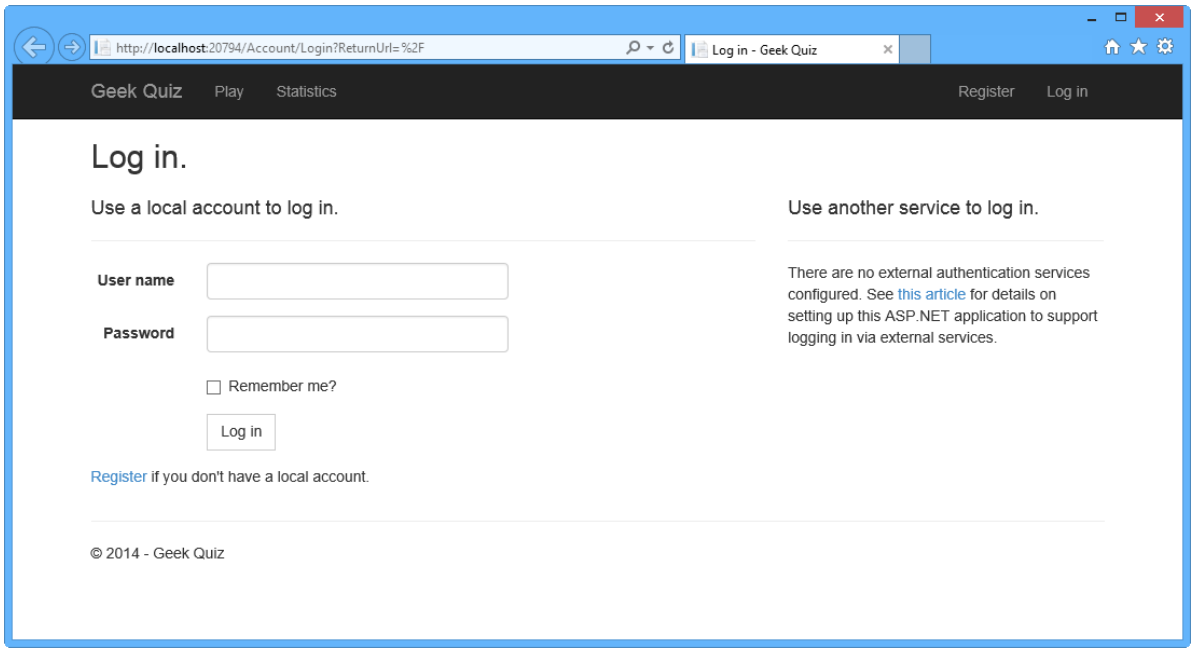
SignalR supports **server push** or **broadcasting** functionality; it handles connection management automatically. In classic HTTP connections for client-server communication, connection is re-established for each request, but SignalR provides persistent connection between the client and server. In SignalR the server code calls out to a client code in the browser using Remote Procedure Calls (RPC), rather than the request-response model we know today.

In this exercise, you will configure the **Geek Quiz** application to use SignalR to display the Statistics dashboard with the updated metrics without the need to refresh the entire page.

Task 1 – Exploring the Geek Quiz Statistics Page

In this task, you will go through the application and verify how the statistics page is shown and how you can improve the way the information is updated.

1. Open **Visual Studio Express 2013 for Web** and open the **GeekQuiz.sln** solution located in the **Source\Ex1-WorkingWithRealTimeData\Begin** folder.
2. Press **F5** to run the solution. The **Log in** page should appear in the browser.

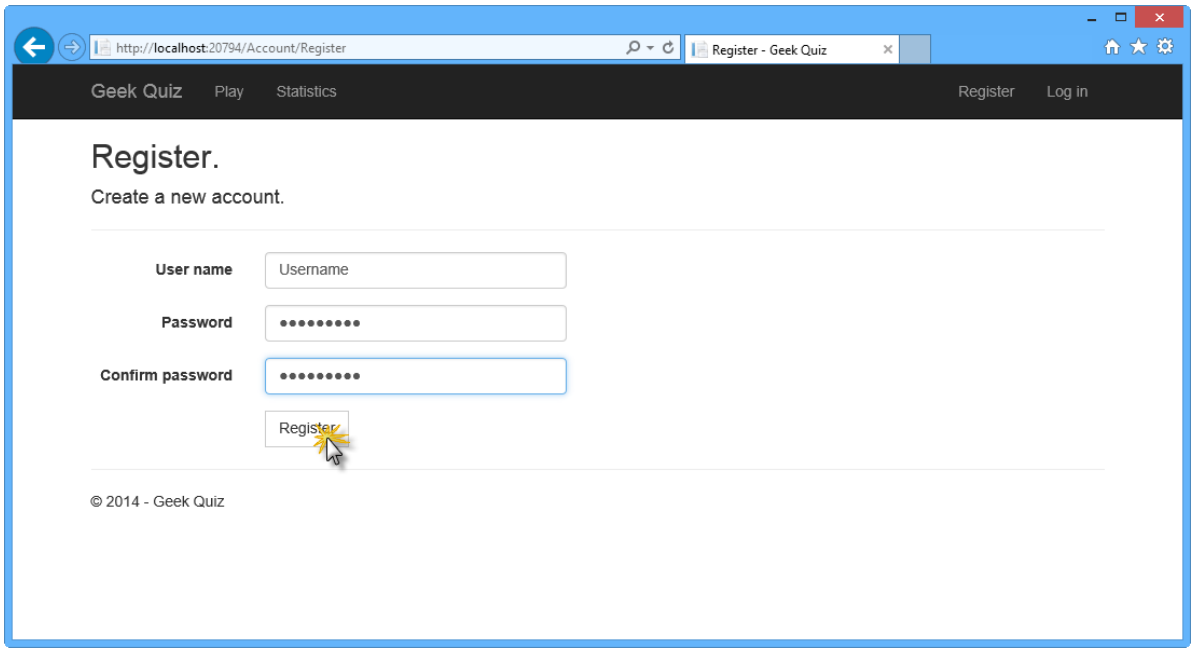


Running the solution

3. Click **Register** in the upper-right corner of the page to create a new user in the application.

Register link

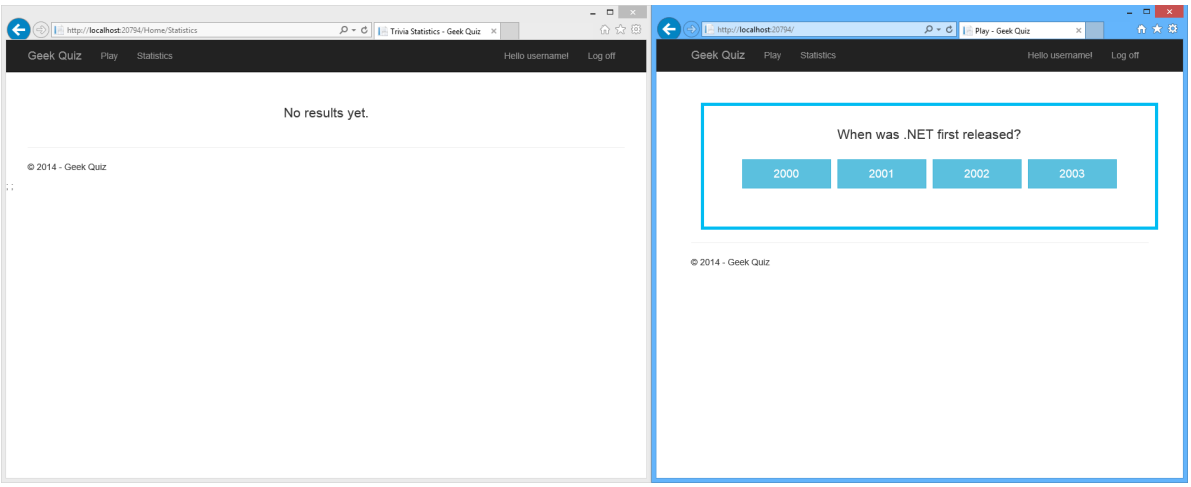
4. In the **Register** page, enter a **User name** and **Password**, and then click **Register**.



Registering a user

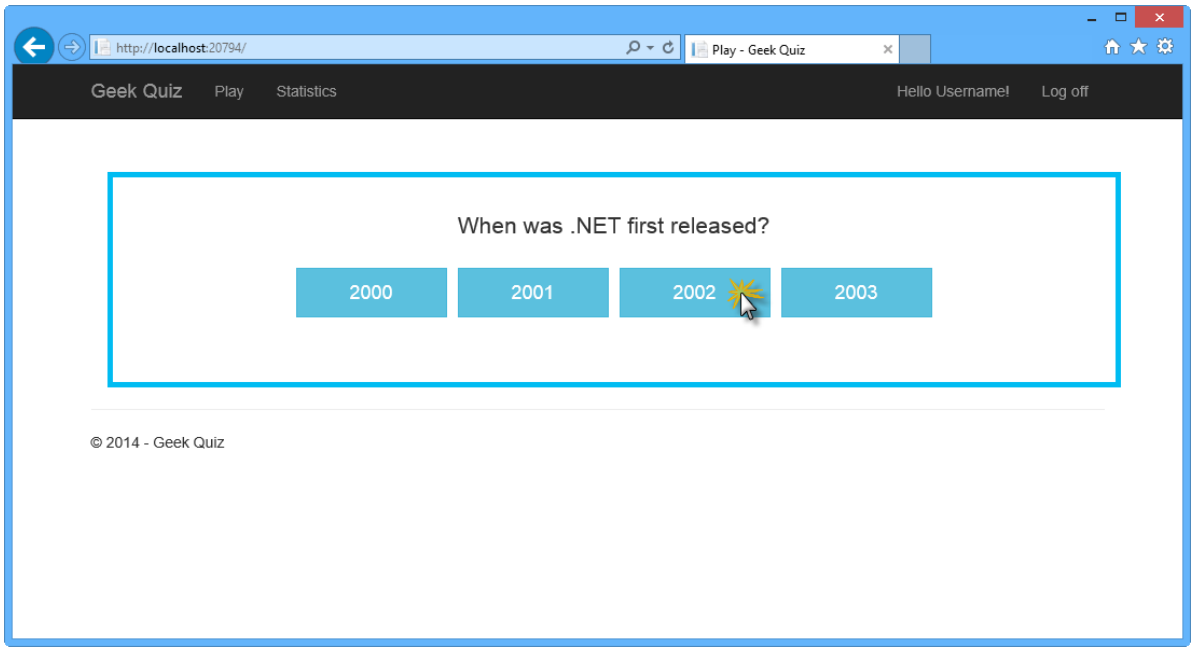
5. The application registers the new account and the user is authenticated and redirected back to the home page showing the first quiz question.

6. Open the **Statistics** page in a new window and put the **Home** page and **Statistics** page side-by-side.



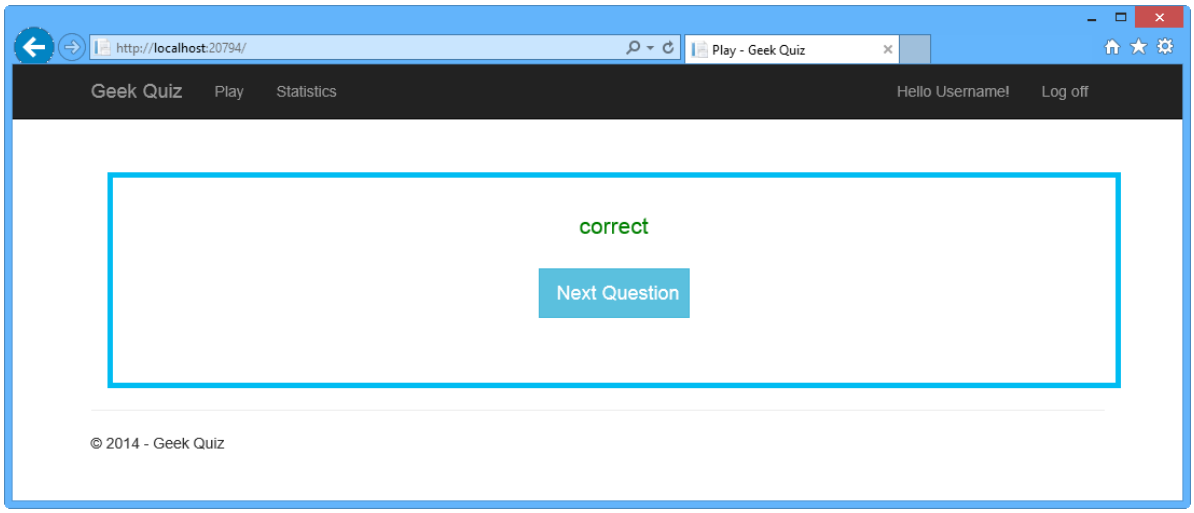
Side-by-side windows

7. In the **Home** page, answer the question by clicking one of the options.



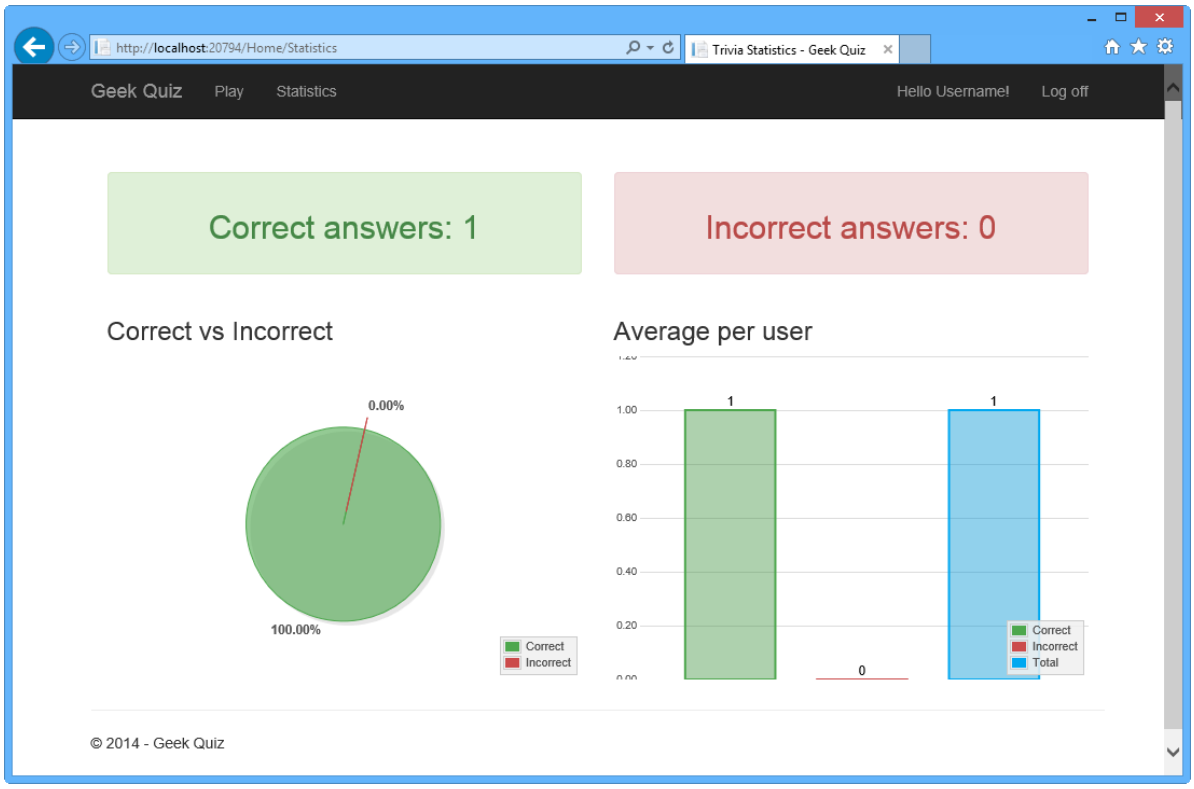
Answering a question

8. After clicking one of the buttons, the answer should appear.



Question answered correctly

9. Notice that the information provided in the Statistics page is outdated. Refresh the page in order to see the updated results.



Statistics page

10. Go back to Visual Studio and stop debugging.

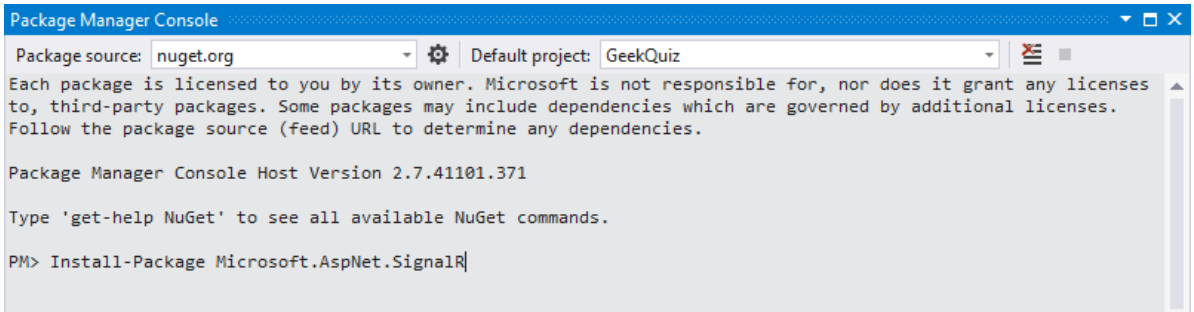
Task 2 – Adding SignalR to Geek Quiz to Show Online Charts

In this task, you will add SignalR to the solution and send updates to the clients automatically when a new answer is sent to the server.

1. From the **Tools** menu in Visual Studio, select **NuGet Package Manager**, and then click **Package Manager Console**.

2. In the **Package Manager Console** window, execute the following command:

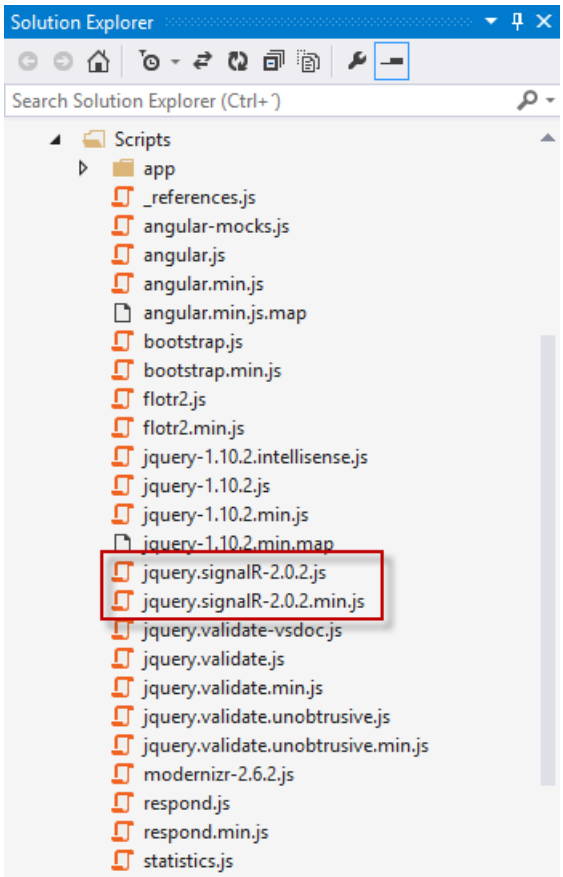
```
Install-Package Microsoft.AspNet.SignalR
```



SignalR package installation

> **NOTE**
>
> When installing **SignalR** NuGet packages version 2.0.2 from a brand new MVC 5 application, you will need to manually update **OWIN** packages to version 2.0.1 (or higher) before installing SignalR. To do this, you can execute the following script in the **Package Manager Console**:
>
> ```
> get-package | where-object { $_.Id -like "Microsoft.Owin*"} | Update-Package
> ```
>
> In a future release of SignalR, OWIN dependencies will be automatically updated.

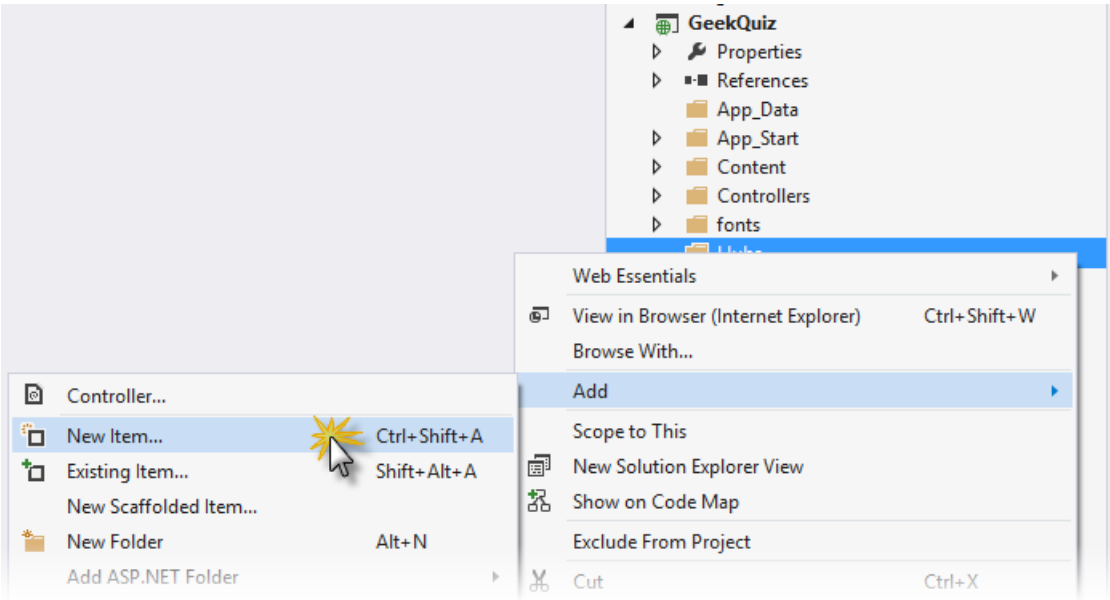
3. In **Solution Explorer**, expand the **Scripts** folder and notice that the SignalR *js* files were added to the solution.



SignalR JavaScript references

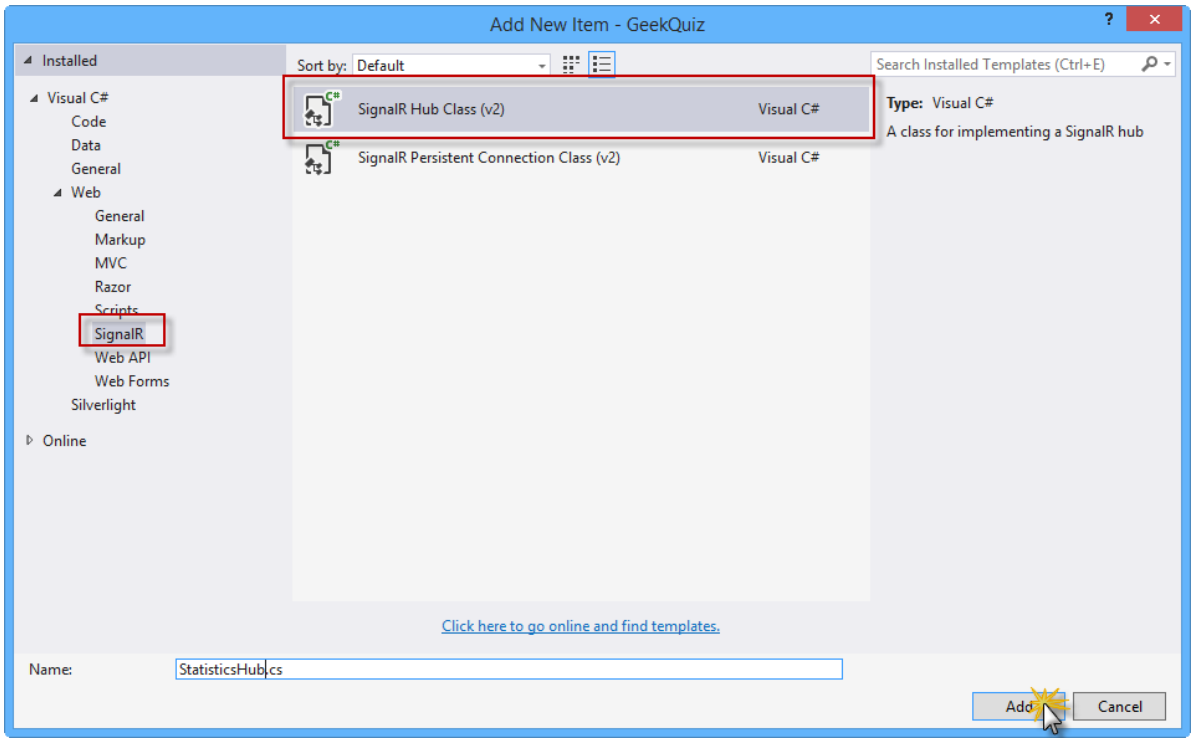
4. In **Solution Explorer**, right-click the **GeekQuiz** project, select **Add | New Folder**, and name it **Hubs**.

5. Right-click the **Hubs** folder and select **Add | New Item**.



Add new item

6. In the **Add New Item** dialog box, select the **Visual C# | Web | SignalR** node in the left pane, select **SignalR Hub Class (v2)** from the center pane, name the file **StatisticsHub.cs** and click **Add**.



Add new item dialog box

7. Replace the code in the **StatisticsHub** class with the following code.

(Code Snippet - *RealTimeSignalR - Ex1 - StatisticsHubClass*)

```
namespace GeekQuiz.Hubs
{
    using Microsoft.AspNet.SignalR;

    public class StatisticsHub : Hub
    {
    }
}
```

8. Open **Startup.cs** and add the following line at the end of the **Configuration** method.

   (Code Snippet - *RealTimeSignalR - Ex1 - MapSignalR*)

```
public void Configuration(IAppBuilder app)
{
    this.ConfigureAuth(app);
    app.MapSignalR();
}
```

9. Open the **StatisticsService.cs** page inside the **Services** folder and add the following using directives.

   (Code Snippet - *RealTimeSignalR - Ex1 - UsingDirectives*)

```
using Microsoft.AspNet.SignalR;
using GeekQuiz.Hubs;
```

10. To notify connected clients of updates, you first retrieve a **Context** object for the current connection. The **Hub** object contains methods to send messages to a single client or broadcast to all connected clients. Add the following method to the **StatisticsService** class to broadcast the statistics data.

    (Code Snippet - *RealTimeSignalR - Ex1 - NotifyUpdatesMethod*)

```
public async Task NotifyUpdates()
{
    var hubContext = GlobalHost.ConnectionManager.GetHubContext<StatisticsHub>();
    if (hubContext != null)
    {
         var stats = await this.GenerateStatistics();
         hubContext.Clients.All.updateStatistics(stats);
    }
}
```

> **NOTE**
>
> In the code above, you are using an arbitrary method name to call a function on the client (i.e.: *updateStatistics*). The method name that you specify is interpreted as a dynamic object, which means there is no IntelliSense or compile-time validation for it. The expression is evaluated at run time. When the method call executes, SignalR sends the method name and the parameter values to the client. If the client has a method that matches the name, that method is called and the parameter values are passed to it. If no matching method is found on the client, no error is raised. For more information, refer to ASP.NET SignalR Hubs API Guide.

11. Open the **TriviaController.cs** page inside the **Controllers** folder and add the following using directives.

```
using GeekQuiz.Services;
```

12. Add the following highlighted code to the **Post** action method.

    (Code Snippet - *RealTimeSignalR - Ex1 - NotifyUpdatesCall*)

```
public async Task<IHttpActionResult> Post(TriviaAnswer answer)
{
    if (!ModelState.IsValid)
    {
         return this.BadRequest(this.ModelState);
    }

    answer.UserId = User.Identity.Name;

    var isCorrect = await this.StoreAsync(answer);

    var statisticsService = new StatisticsService(this.db);
    await statisticsService.NotifyUpdates();

    return this.Ok<bool>(isCorrect);
}
```

13. Open the **Statistics.cshtml** page inside the **Views | Home** folder. Locate the **Scripts** section and add the following script references at the beginning of the section.

    (Code Snippet - *RealTimeSignalR - Ex1 - SignalRScriptReferences*)

```
@section Scripts {
    @Scripts.Render("~/Scripts/jquery.signalR-2.0.2.min.js");
    @Scripts.Render("~/signalr/hubs");
    ...
}
```

> **NOTE**
> When you add SignalR and other script libraries to your Visual Studio project, the Package Manager might install a version of the SignalR script file that is more recent than the version shown in this topic. Make sure that the script reference in your code matches the version of the script library installed in your project.

14. Add the following highlighted code to connect the client to the SignalR hub and update the statistics data when a new message is received from the hub.

    (Code Snippet - *RealTimeSignalR - Ex1 - SignalRClientCode*)

```
@section Scripts {
    ...
    <script>
        ...

        var connection = $.hubConnection();
        var hub = connection.createHubProxy("StatisticsHub");
        hub.on("updateStatistics", function (statistics) {
                statisticsData = statistics;
                $("#correctAnswersCounter").text(statistics.CorrectAnswers);
                $("#incorrectAnswersCounter").text(statistics.IncorrectAnswers);

                showCharts(statisticsData);
          });

        connection.start();
    </script>
}
```

In this code, you are creating a Hub Proxy and registering an event handler to listen for messages sent by

the server. In this case, you listen for messages sent through the *updateStatistics* method.

Task 3 – Running the Solution

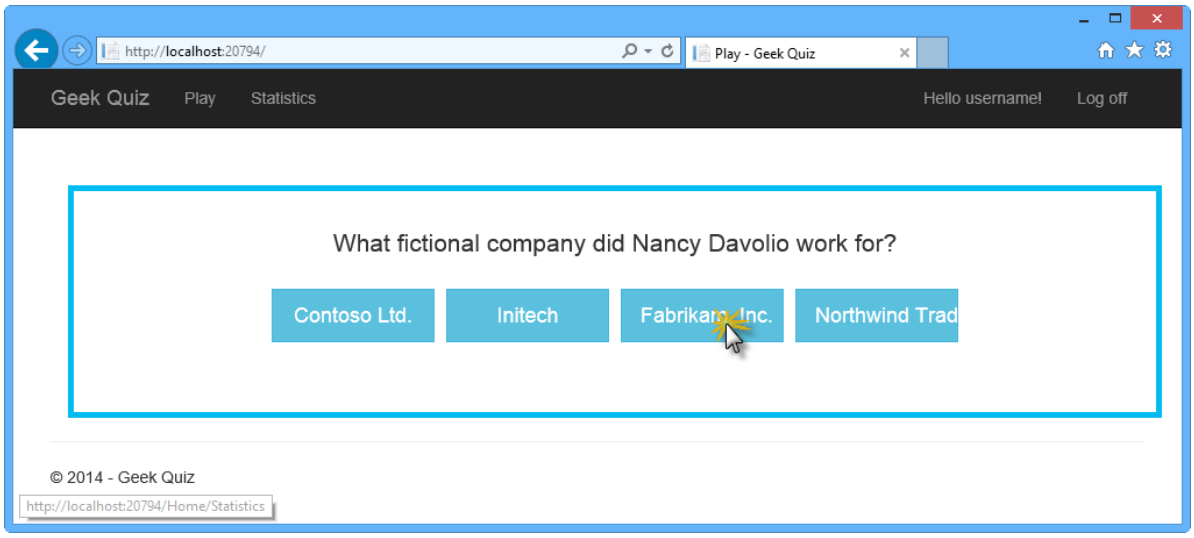
In this task, you will run the solution to verify that the statistics view is updated automatically using SignalR after answering a new question.

1. Press **F5** to run the solution.

> **NOTE**
>
> If not already logged in to the application, log in with the user you created in Task 1.

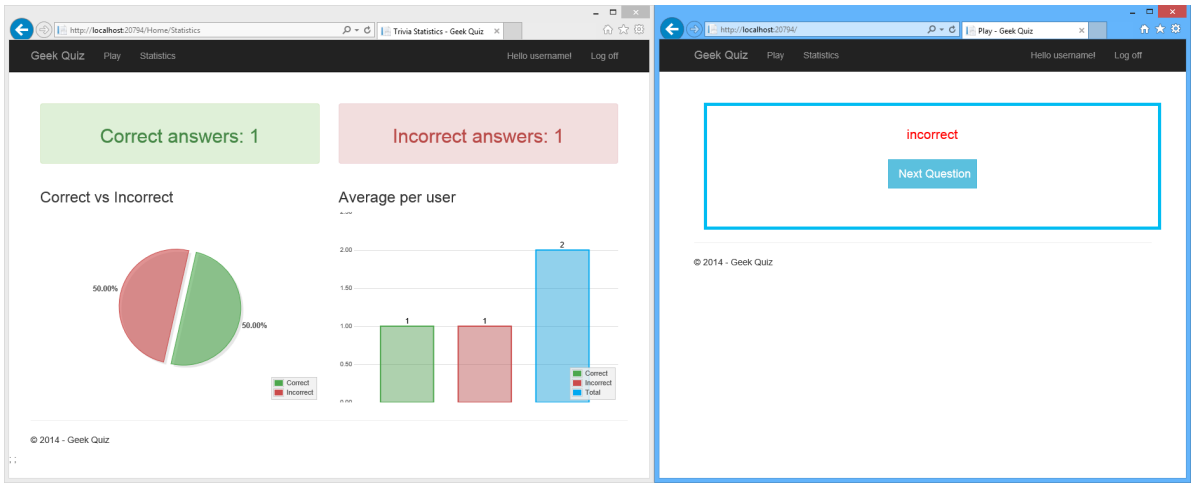
2. Open the **Statistics** page in a new window and put the **Home** page and **Statistics** page side-by-side as you did in Task 1.

3. In the **Home** page, answer the question by clicking one of the options.



Answering another question

4. After clicking one of the buttons, the answer should appear. Notice that the Statistics information on the page is updated automatically after answering the question with the updated information without the need to refresh the entire page.



Statistics page refreshed after answer

Exercise 2: Scaling Out Using SQL Server

When scaling a web application, you can generally choose between *scaling up* and *scaling out* options. *Scale up* means using a larger server, with more resources (CPU, RAM, etc.) while *scale out* means adding more servers to

handle the load. The problem with the latter is that the clients can get routed to different servers. A client that is connected to one server will not receive messages sent from another server.

You can solve these issues by using a component called *backplane*, to forward messages between servers. With a backplane enabled, each application instance sends messages to the backplane, and the backplane forwards them to the other application instances.

There are currently three types of backplanes for SignalR:

- **Windows Azure Service Bus**. Service Bus is a messaging infrastructure that allows components to send loosely coupled messages.
- **SQL Server**. The SQL Server backplane writes messages to SQL tables. The backplane uses Service Broker for efficient messaging. However, it also works if Service Broker is not enabled.
- **Redis**. Redis is an in-memory key-value store. Redis supports a publish/subscribe ("pub/sub") pattern for sending messages.

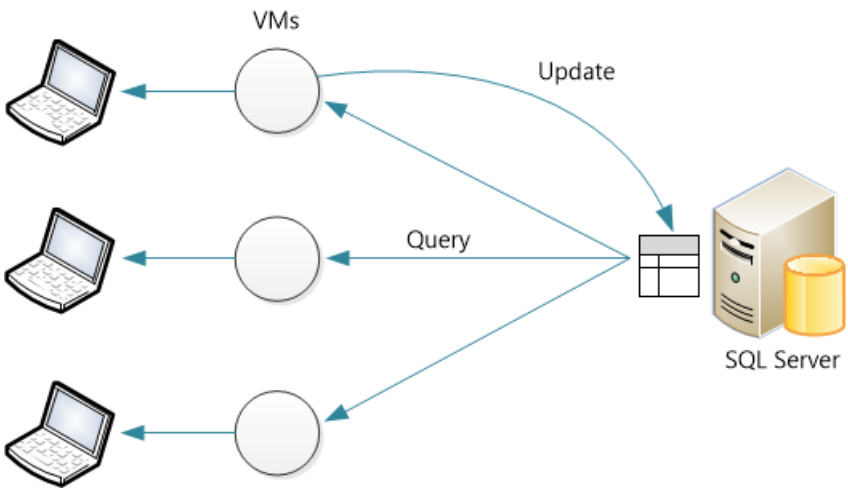
Every message is sent through a message bus. A message bus implements the IMessageBus interface, which provides a publish/subscribe abstraction. The backplanes work by replacing the default **IMessageBus** with a bus designed for that backplane.

Each server instance connects to the backplane through the bus. When a message is sent, it goes to the backplane, and the backplane sends it to every server. When a server receives a message from the backplane, it stores the message in its local cache. The server then delivers messages to clients from its local cache.

For more information about how the SignalR backplane works, read this article.

> **NOTE**
>
> There are some scenarios where a backplane can become a bottleneck. Here are some typical SignalR scenarios:
>
> - Server broadcast (e.g., stock ticker): Backplanes work well for this scenario, because the server controls the rate at which messages are sent.
> - Client-to-client (e.g., chat): In this scenario, the backplane might be a bottleneck if the number of messages scales with the number of clients; that is, if the rate of messages grows proportionally as more clients join.
> - High-frequency realtime (e.g., real-time games): A backplane is not recommended for this scenario.

In this exercise, you will use **SQL Server** to distribute messages across the **Geek Quiz** application. You will run these tasks on a single test machine to learn how to set up the configuration, but in order to get the full effect, you will need to deploy the SignalR application to two or more servers. You must also install SQL Server on one of the servers, or on a separate dedicated server.

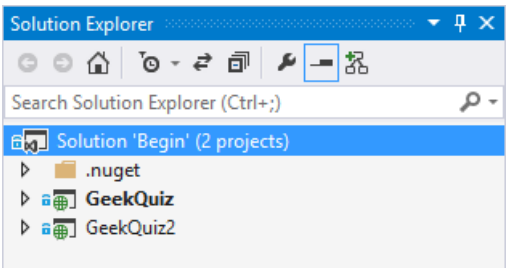


Task 1 - Understanding the Scenario

In this task, you will run 2 instances of **Geek Quiz** simulating multiple IIS instances on your local machine. In this

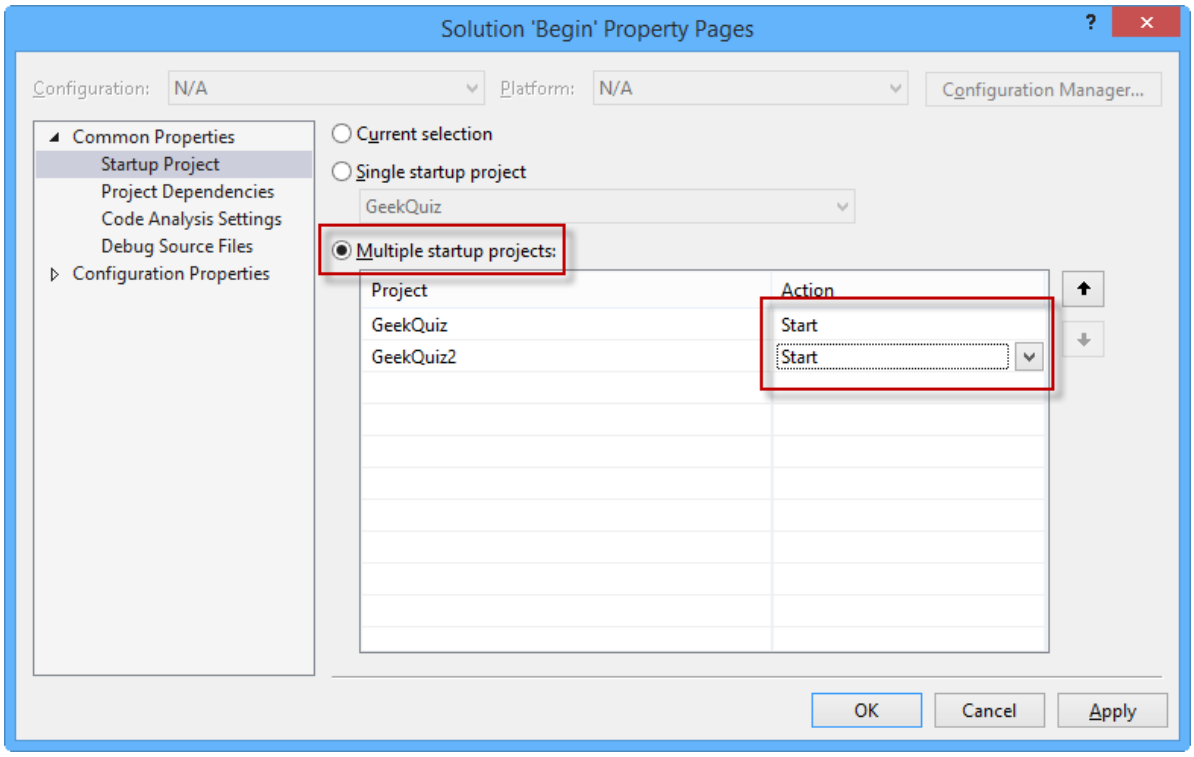
scenario, when answering trivia questions on one application, update won't be notified on the statistics page of the second instance. This simulation resembles an environment where your application is deployed on multiple instances and using a load balancer to communicate with them.

1. Open the **Begin.sln** solution located in the **Source/Ex2-ScalingOutWithSQLServer/Begin** folder. Once loaded, you will notice on the **Server Explorer** that the solution has two projects with identical structures but different names. This will simulate running two instances of the same application on your local machine.



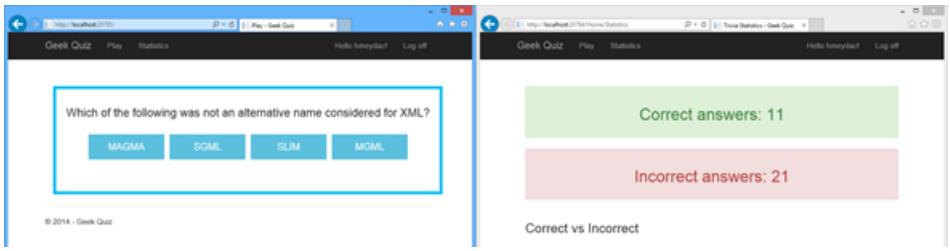
Begin Solution Simulating 2 Instances of Geek Quiz

2. Open the properties page of the solution by right-clicking the solution node and selecting **Properties**. Under **Startup Project**, select **Multiple startup projects** and change the **Action** value for both projects to *Start*.

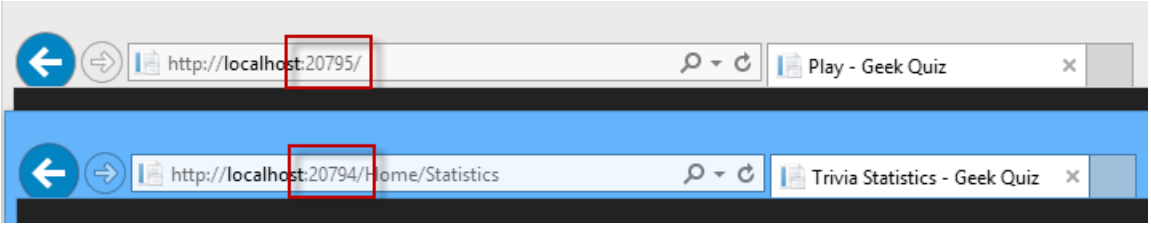


Starting Multiple Projects

3. Press **F5** to run the solution. The application will launch two instances of **Geek Quiz** in different ports, simulating multiple instances of the same application. Pin one of the browsers on left and the other on the right of your screen. Log in with your credentials or register a new user. Once logged in, keep the Trivia page on the left and go to the **Statistics** page in the browser on the right.



Geek Quiz Side by Side



Geek Quiz in Different Ports

4. Start answering questions in the left browser and you will notice that the **Statistics** page in the right browser is not being updated. This is because **SignalR** uses a local cache to distribute messages across their clients and this scenario is simulating multiple instances, therefore the cache is not shared between them. You can verify that **SignalR** is working by testing the same steps but using a single app. In the following tasks you will configure a backplane to replicate the messages across instances.

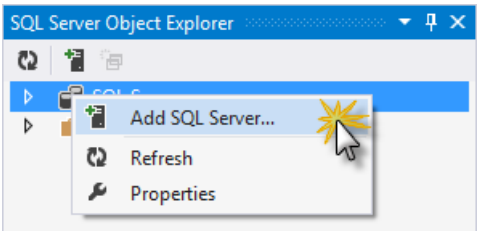
5. Go back to Visual Studio and stop debugging.

Task 2 – Creating the SQL Server Backplane

In this task, you will create a database that will serve as a backplane for the **Geek Quiz** application. You will use **SQL Server Object Explorer** to browse your server and initialize the database. Additionally, you will enable the **Service Broker**.

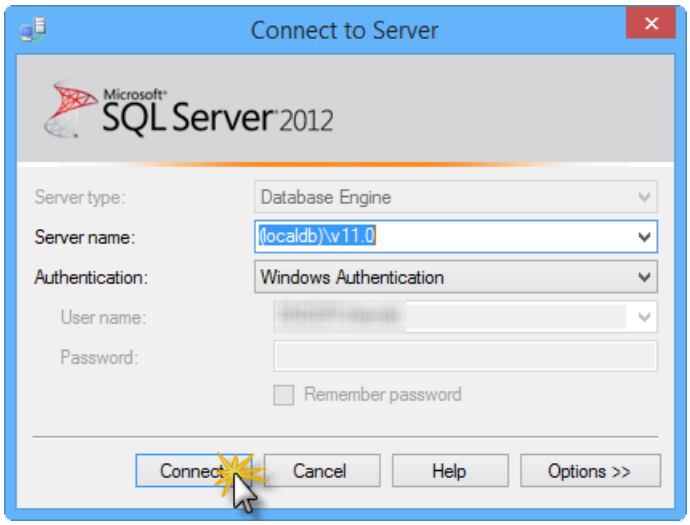
1. In **Visual Studio**, open menu **View** and select **SQL Server Object Explorer**.

2. Connect to your LocalDB instance by right-clicking the **SQL Server** node and selecting **Add SQL Server...** option.



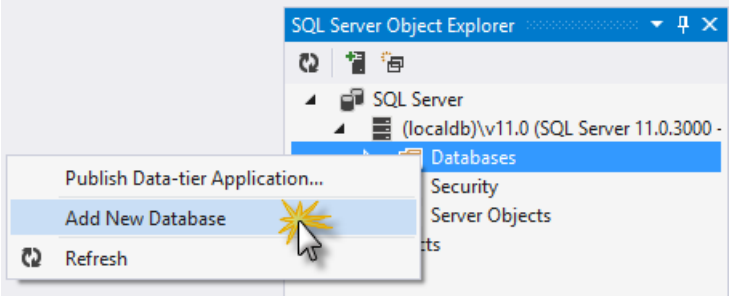
Adding a SQL Server instance to SQL Server Object Explorer

3. Set the **server name** to *(localdb)\v11.0* and leave **Windows Authentication** as your authentication mode. Click **Connect** to continue.



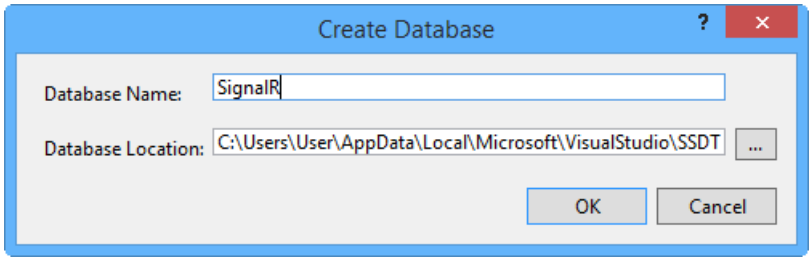
Connecting to LocalDB

4. Now that you are connected to your LocalDB instance, you will need to create a database that will represent the SQL Server backplane for SignalR. To do this, right-click the **Databases** node and select **Add New Database**.



Adding a new database

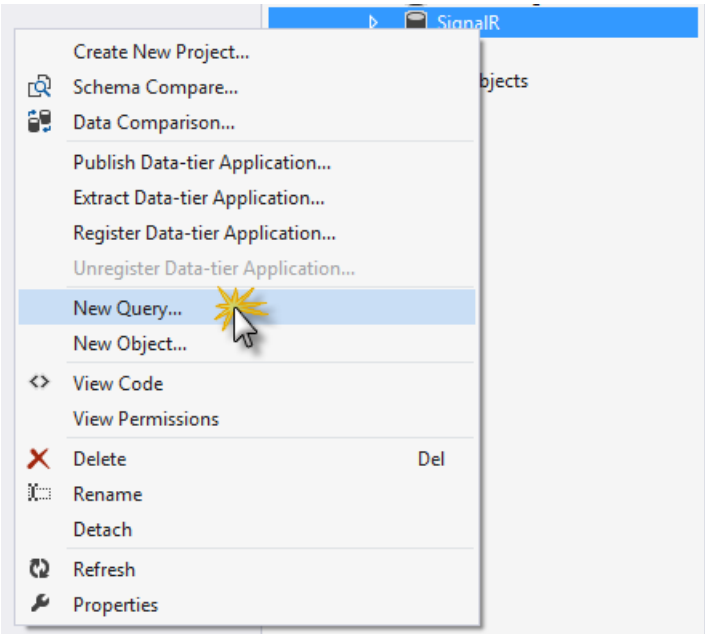
5. Set the database name to *SignalR* and click **OK** to create it.



Creating the SignalR database

> **NOTE**
>
> You can choose any name for the database.

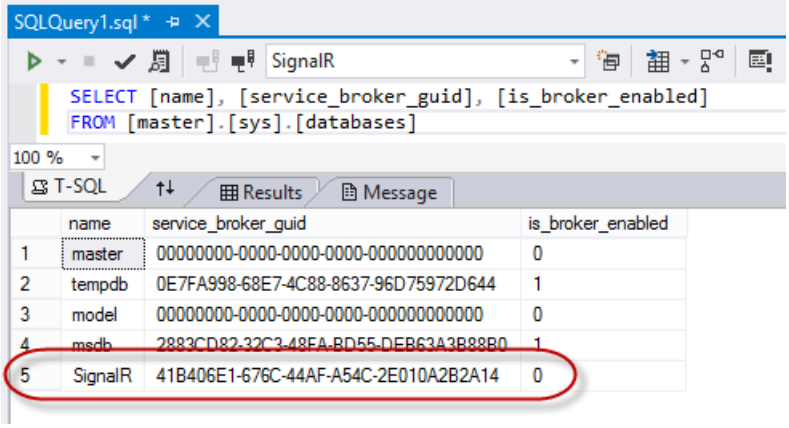
6. To receive updates more efficiently from the backplane, it is recommended to enable Service Broker for the database. Service Broker provides native support for messaging and queuing in SQL Server. The backplane also works without Service Broker. Open a new query by right-clicking the database and select **New Query**.



Opening a New Query

7. To check whether Service Broker is enabled, query the **is_broker_enabled** column in the **sys.databases** catalog view. Execute the following script in the recently opened query window.

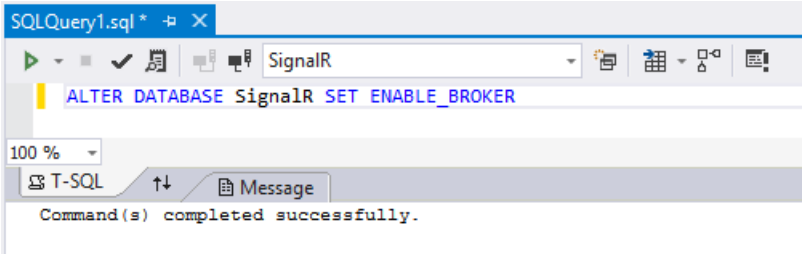
```
SELECT [name], [service_broker_guid], [is_broker_enabled] FROM [master].[sys].[databases]
```



Querying the Service Broker Status

8. If the value of the **is_broker_enabled** column in your database is "0", use the following command to enable it. Replace **<YOUR-DATABASE>** with the name you set when creating the database (e.g.: SignalR).

```
ALTER DATABASE <YOUR-DATABASE> SET ENABLE_BROKER
```



Enabling Service Broker

> **NOTE**
> If this query appears to deadlock, make sure there are no applications connected to the DB.

Task 3 – Configuring the SignalR Application

In this task, you will configure **Geek Quiz** to connect to the SQL Server backplane. You will first add the **SignalR.SqlServer** NuGet package and set the connection string to your backplane database.

1. Open the **Package Manager Console** from **Tools** > **NuGet Package Manager**. Make sure that **GeekQuiz** project is selected in the **Default project** drop-down list. Type the following command to install the **Microsoft.AspNet.SignalR.SqlServer** NuGet package.

```
Install-Package Microsoft.AspNet.SignalR.SqlServer
```

2. Repeat the previous step but this time for project **GeekQuiz2**.

3. To configure the SQL Server backplane, open the **Startup.cs** file of the **GeekQuiz** project and add the following code to the **Configure** method. Replace **<YOUR-DATABASE>** with your database name you used when creating the SQL Server backplane. Repeat this step for the **GeekQuiz2** project.

   (Code Snippet - *RealTimeSignalR - Ex2 - StartupConfiguration*)

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        var sqlConnectionString = @"Server=(localdb)\v11.0;Database=<YOUR-DATABASE>;Integrated
Security=True;";
        GlobalHost.DependencyResolver.UseSqlServer(sqlConnectionString);
        this.ConfigureAuth(app);
        app.MapSignalR();
    }
}
```

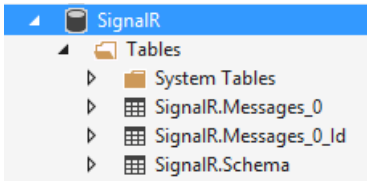
4. Now that both projects are configured to use the SQL Server backplane, press **F5** to run them simultaneously.

5. Again, **Visual Studio** will launch two instances of **Geek Quiz** in different ports. Pin one of the browsers on the left and the other on the right of your screen and log in with your credentials. Keep the Trivia page on the left and go to **Statistics** pagein the right browser.

6. Start answering questions in the left browser. This time, the **Statistics** page is updated thanks to the backplane. Switch between applications (**Statistics** is now on the left, and **Trivia** is on the right) and repeat the test to validate that it is working for both instances. The backplane serves as a *shared cache* of messages for each connected server, and each server will store the messages in their own local cache to distribute to connected clients.

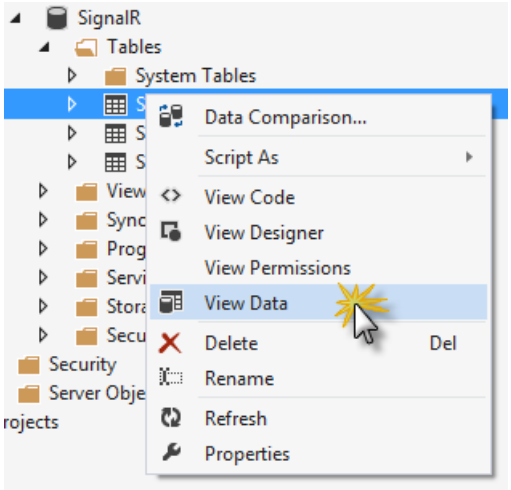
7. Go back to Visual Studio and stop debugging.

8. The SQL Server backplane component automatically generates the necessary tables on the specified database. In the **SQL Server Object Explorer** panel, open the database you created for the backplane (e.g.: SignalR) and expand its tables. You should see the following tables:



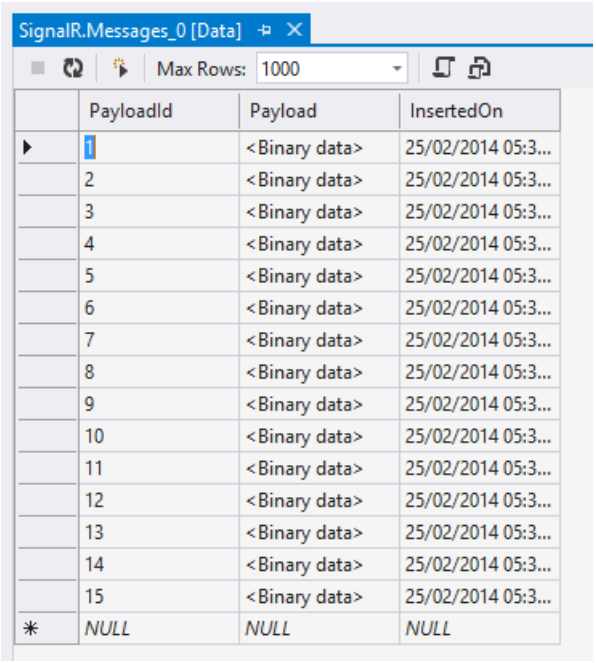
Backplane Generated Tables

9. Right-click the **SignalR.Messages_0** table and select **View Data**.



View SignalR Backplane Messages Table

10. You can see the different messages sent to the **Hub** when answering the trivia questions. The backplane distributes these messages to any connected instance.



PayloadId	Payload	InsertedOn
1	<Binary data>	25/02/2014 05:3...
2	<Binary data>	25/02/2014 05:3...
3	<Binary data>	25/02/2014 05:3...
4	<Binary data>	25/02/2014 05:3...
5	<Binary data>	25/02/2014 05:3...
6	<Binary data>	25/02/2014 05:3...
7	<Binary data>	25/02/2014 05:3...
8	<Binary data>	25/02/2014 05:3...
9	<Binary data>	25/02/2014 05:3...
10	<Binary data>	25/02/2014 05:3...
11	<Binary data>	25/02/2014 05:3...
12	<Binary data>	25/02/2014 05:3...
13	<Binary data>	25/02/2014 05:3...
14	<Binary data>	25/02/2014 05:3...
15	<Binary data>	25/02/2014 05:3...
NULL	NULL	NULL

Backplane Messages Table

Summary

In this hands-on lab, you have learned how to add **SignalR** to your application and send notifications from the server to your connected clients using **Hubs**. Additionally, you learned how to scale out your application by using a *backplane* component when your application is deployed in multiple IIS instances.

SignalR Guide to the API

2/20/2020 • 2 minutes to read • Edit Online

Documentation on how to use the SignalR Hubs API. For authorization API, see the Security tab.

- Hubs API Guide - Server (C#)
- Hubs API Guide - JavaScript Client
- Hubs API Guide - .NET Client (C#)
- Understanding and Handling Connection Lifetime Events in SignalR
- Working with Groups in SignalR
- Mapping SignalR Users to Connections

ASP.NET SignalR Hubs API Guide - Server (C#)

2/20/2020 • 32 minutes to read • Edit Online

by Patrick Fletcher, Tom Dykstra

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This document provides an introduction to programming the server side of the ASP.NET SignalR Hubs API for SignalR version 2, with code samples demonstrating common options.

The SignalR Hubs API enables you to make remote procedure calls (RPCs) from a server to connected clients and from clients to the server. In server code, you define methods that can be called by clients, and you call methods that run on the client. In client code, you define methods that can be called from the server, and you call methods that run on the server. SignalR takes care of all of the client-to-server plumbing for you.

SignalR also offers a lower-level API called Persistent Connections. For an introduction to SignalR, Hubs, and Persistent Connections, see Introduction to SignalR 2.

Software versions used in this topic

- Visual Studio 2013
- .NET 4.5
- SignalR version 2

Topic versions

For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Overview

This document contains the following sections:

- How to register SignalR middleware
  - The /signalr URL
  - Configuring SignalR options
- How to create and use Hub classes
  - Hub object lifetime
  - Camel-casing of Hub names in JavaScript clients
  - Multiple Hubs
  - Strongly-Typed Hubs

- How to define methods in the Hub class that clients can call
  - Camel-casing of method names in JavaScript clients
  - When to execute asynchronously
  - Defining overloads
  - Reporting progress from hub method invocations
- How to call client methods from the Hub class
  - Selecting which clients will receive the RPC
  - No compile-time validation for method names
  - Case-insensitive method name matching
  - Asynchronous execution
- How to manage group membership from the Hub class
  - Asynchronous execution of Add and Remove methods
  - Group membership persistence
  - Single-user groups
- How to handle connection lifetime events in the Hub class
  - When OnConnected, OnDisconnected, and OnReconnected are called
  - Caller state not populated
- How to get information about the client from the Context property
- How to pass state between clients and the Hub class
- How to handle errors in the Hub class
- How to call client methods and manage groups from outside the Hub class
  - Calling client methods
  - Managing group membership
- How to enable tracing
- How to customize the Hubs pipeline

For documentation on how to program clients, see the following resources:

- SignalR Hubs API Guide - JavaScript Client
- SignalR Hubs API Guide - .NET Client

The server components for SignalR 2 are only available in .NET 4.5. Servers running .NET 4.0 must use SignalR v1.x.

How to register SignalR middleware

To define the route that clients will use to connect to your Hub, call the `MapSignalR` method when the application starts. `MapSignalR` is an extension method for the `OwinExtensions` class. The following example shows how to define the SignalR Hubs route using an OWIN startup class.

```
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(MyApplication.Startup))]
namespace MyApplication
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Any connection or hub wire up and configuration should go here
            app.MapSignalR();
        }

    }
}
```

If you are adding SignalR functionality to an ASP.NET MVC application, make sure that the SignalR route is added before the other routes. For more information, see Tutorial: Getting Started with SignalR 2 and MVC 5.

The /signalr URL

By default, the route URL which clients will use to connect to your Hub is "/signalr". (Don't confuse this URL with the "/signalr/hubs" URL, which is for the automatically generated JavaScript file. For more information about the generated proxy, see SignalR Hubs API Guide - JavaScript Client - The generated proxy and what it does for you.)

There might be extraordinary circumstances that make this base URL not usable for SignalR; for example, you have a folder in your project named *signalr* and you don't want to change the name. In that case, you can change the base URL, as shown in the following examples (replace "/signalr" in the sample code with your desired URL).

Server code that specifies the URL

```
app.MapSignalR("/signalr", new HubConfiguration());
```

JavaScript client code that specifies the URL (with the generated proxy)

```
$.connection.hub.url = "/signalr"
$.connection.hub.start().done(init);
```

JavaScript client code that specifies the URL (without the generated proxy)

```
var connection = $.hubConnection("/signalr", { useDefaultPath: false });
```

.NET client code that specifies the URL

```
var hubConnection = new HubConnection("http://contoso.com/signalr", useDefaultUrl: false);
```

Configuring SignalR Options

Overloads of the `MapSignalR` method enable you to specify a custom URL, a custom dependency resolver, and the following options:

- Enable cross-domain calls using CORS or JSONP from browser clients.

  Typically if the browser loads a page from `http://contoso.com`, the SignalR connection is in the same domain, at `http://contoso.com/signalr`. If the page from `http://contoso.com` makes a connection to `http://fabrikam.com/signalr`, that is a cross-domain connection. For security reasons, cross-domain

connections are disabled by default. For more information, see ASP.NET SignalR Hubs API Guide - JavaScript Client - How to establish a cross-domain connection.

- Enable detailed error messages.

  When errors occur, the default behavior of SignalR is to send to clients a notification message without details about what happened. Sending detailed error information to clients is not recommended in production, because malicious users might be able to use the information in attacks against your application. For troubleshooting, you can use this option to temporarily enable more informative error reporting.

- Disable automatically generated JavaScript proxy files.

  By default, a JavaScript file with proxies for your Hub classes is generated in response to the URL "/signalr/hubs". If you don't want to use the JavaScript proxies, or if you want to generate this file manually and refer to a physical file in your clients, you can use this option to disable proxy generation. For more information, see SignalR Hubs API Guide - JavaScript Client - How to create a physical file for the SignalR generated proxy.

The following example shows how to specify the SignalR connection URL and these options in a call to the `MapSignalR` method. To specify a custom URL, replace "/signalr" in the example with the URL that you want to use.

```
var hubConfiguration = new HubConfiguration();
hubConfiguration.EnableDetailedErrors = true;
hubConfiguration.EnableJavaScriptProxies = false;
app.MapSignalR("/signalr", hubConfiguration);
```

How to create and use Hub classes

To create a Hub, create a class that derives from Microsoft.Aspnet.Signalr.Hub. The following example shows a simple Hub class for a chat application.

```
public class ContosoChatHub : Hub
{
    public async Task NewContosoChatMessage(string name, string message)
    {
        await Clients.All.addNewMessageToPage(name, message);
    }
}
```

In this example, a connected client can call the `NewContosoChatMessage` method, and when it does, the data received is broadcasted to all connected clients.

Hub object lifetime

You don't instantiate the Hub class or call its methods from your own code on the server; all that is done for you by the SignalR Hubs pipeline. SignalR creates a new instance of your Hub class each time it needs to handle a Hub operation such as when a client connects, disconnects, or makes a method call to the server.

Because instances of the Hub class are transient, you can't use them to maintain state from one method call to the next. Each time the server receives a method call from a client, a new instance of your Hub class processes the message. To maintain state through multiple connections and method calls, use some other method such as a database, or a static variable on the Hub class, or a different class that does not derive from `Hub`. If you persist data in memory, using a method such as a static variable on the Hub class, the data will be lost when the app domain recycles.

If you want to send messages to clients from your own code that runs outside the Hub class, you can't do it by

instantiating a Hub class instance, but you can do it by getting a reference to the SignalR context object for your Hub class. For more information, see How to call client methods and manage groups from outside the Hub class later in this topic.

Camel-casing of Hub names in JavaScript clients

By default, JavaScript clients refer to Hubs by using a camel-cased version of the class name. SignalR automatically makes this change so that JavaScript code can conform to JavaScript conventions. The previous example would be referred to as `contosoChatHub` in JavaScript code.

Server

```
public class ContosoChatHub : Hub
```

JavaScript client using generated proxy

```
var contosoChatHubProxy = $.connection.contosoChatHub;
```

If you want to specify a different name for clients to use, add the `HubName` attribute. When you use a `HubName` attribute, there is no name change to camel case on JavaScript clients.

Server

```
[HubName("PascalCaseContosoChatHub")]
public class ContosoChatHub : Hub
```

JavaScript client using generated proxy

```
var contosoChatHubProxy = $.connection.PascalCaseContosoChatHub;
```

Multiple Hubs

You can define multiple Hub classes in an application. When you do that, the connection is shared but groups are separate:

- All clients will use the same URL to establish a SignalR connection with your service ("/signalr" or your custom URL if you specified one), and that connection is used for all Hubs defined by the service.

  There is no performance difference for multiple Hubs compared to defining all Hub functionality in a single class.

- All Hubs get the same HTTP request information.

  Since all Hubs share the same connection, the only HTTP request information that the server gets is what comes in the original HTTP request that establishes the SignalR connection. If you use the connection request to pass information from the client to the server by specifying a query string, you can't provide different query strings to different Hubs. All Hubs will receive the same information.

- The generated JavaScript proxies file will contain proxies for all Hubs in one file.

  For information about JavaScript proxies, see SignalR Hubs API Guide - JavaScript Client - The generated proxy and what it does for you.

- Groups are defined within Hubs.

  In SignalR you can define named groups to broadcast to subsets of connected clients. Groups are maintained separately for each Hub. For example, a group named "Administrators" would include one set

of clients for your `ContosoChatHub` class, and the same group name would refer to a different set of clients for your `StockTickerHub` class.

Strongly-Typed Hubs

To define an interface for your hub methods that your client can reference (and enable Intellisense on your hub methods), derive your hub from `Hub<T>` (introduced in SignalR 2.1) rather than `Hub`:

```
public class StrongHub : Hub<IClient>
{
    public async Task Send(string message)
    {
        await Clients.All.NewMessage(message);
    }
}

public interface IClient
{
    Task NewMessage(string message);
}
```

How to define methods in the Hub class that clients can call

To expose a method on the Hub that you want to be callable from the client, declare a public method, as shown in the following examples.

```
public class ContosoChatHub : Hub
{
    public async Task NewContosoChatMessage(string name, string message)
    {
        await Clients.All.addNewMessageToPage(name, message);
    }
}
```

```
public class StockTickerHub : Hub
{
    public IEnumerable<Stock> GetAllStocks()
    {
        return _stockTicker.GetAllStocks();
    }
}
```

You can specify a return type and parameters, including complex types and arrays, as you would in any C# method. Any data that you receive in parameters or return to the caller is communicated between the client and the server by using JSON, and SignalR handles the binding of complex objects and arrays of objects automatically.

Camel-casing of method names in JavaScript clients

By default, JavaScript clients refer to Hub methods by using a camel-cased version of the method name. SignalR automatically makes this change so that JavaScript code can conform to JavaScript conventions.

Server

```
public void NewContosoChatMessage(string userName, string message)
```

JavaScript client using generated proxy

```
contosoChatHubProxy.server.newContosoChatMessage(userName, message);
```

If you want to specify a different name for clients to use, add the `HubMethodName` attribute.

Server

```
[HubMethodName("PascalCaseNewContosoChatMessage")]
public void NewContosoChatMessage(string userName, string message)
```

JavaScript client using generated proxy

```
contosoChatHubProxy.server.PascalCaseNewContosoChatMessage(userName, message);
```

When to execute asynchronously

If the method will be long-running or has to do work that would involve waiting, such as a database lookup or a web service call, make the Hub method asynchronous by returning a Task (in place of `void` return) or Task<T> object (in place of `T` return type). When you return a `Task` object from the method, SignalR waits for the `Task` to complete, and then it sends the unwrapped result back to the client, so there is no difference in how you code the method call in the client.

Making a Hub method asynchronous avoids blocking the connection when it uses the WebSocket transport. When a Hub method executes synchronously and the transport is WebSocket, subsequent invocations of methods on the Hub from the same client are blocked until the Hub method completes.

The following example shows the same method coded to run synchronously or asynchronously, followed by JavaScript client code that works for calling either version.

Synchronous

```
public IEnumerable<Stock> GetAllStocks()
{
    // Returns data from memory.
    return _stockTicker.GetAllStocks();
}
```

Asynchronous

```
public async Task<IEnumerable<Stock>> GetAllStocks()
{
    // Returns data from a web service.
    var uri = Util.getServiceUri("Stocks");
    using (HttpClient httpClient = new HttpClient())
    {
        var response = await httpClient.GetAsync(uri);
        return (await response.Content.ReadAsAsync<IEnumerable<Stock>>());
    }
}
```

JavaScript client using generated proxy

```
stockTickerHubProxy.server.getAllStocks().done(function (stocks) {
    $.each(stocks, function () {
        alert(this.Symbol + ' ' + this.Price);
    });
});
```

For more information about how to use asynchronous methods in ASP.NET 4.5, see Using Asynchronous Methods in ASP.NET MVC 4.

Defining Overloads

If you want to define overloads for a method, the number of parameters in each overload must be different. If you differentiate an overload just by specifying different parameter types, your Hub class will compile but the SignalR service will throw an exception at run time when clients try to call one of the overloads.

Reporting progress from hub method invocations

SignalR 2.1 adds support for the progress reporting pattern introduced in .NET 4.5. To implement progress reporting, define an `IProgress<T>` parameter for your hub method that your client can access:

```
public class ProgressHub : Hub
{
    public async Task<string> DoLongRunningThing(IProgress<int> progress)
    {
        for (int i = 0; i <= 100; i+=5)
        {
            await Task.Delay(200);
            progress.Report(i);
        }
        return "Job complete!";
    }
}
```

When writing a long-running server method, it is important to use an asynchronous programming pattern like Async/ Await rather than blocking the hub thread.

How to call client methods from the Hub class

To call client methods from the server, use the `Clients` property in a method in your Hub class. The following example shows server code that calls `addNewMessageToPage` on all connected clients, and client code that defines the method in a JavaScript client.

Server

```
public class ContosoChatHub : Hub
{
    public async Task NewContosoChatMessage(string name, string message)
    {
        await Clients.All.addNewMessageToPage(name, message);
    }
}
```

Invoking a client method is an asynchronous operation and returns a `Task`. Use `await`:

- To ensure the message is sent without error.
- To enable catching and handling errors in a try-catch block.

JavaScript client using generated proxy

```
contosoChatHubProxy.client.addNewMessageToPage = function (name, message) {
    // Add the message to the page.
    $('#discussion').append('<li><strong>' + htmlEncode(name)
        + '</strong>: ' + htmlEncode(message) + '<li>');
};
```

You can't get a return value from a client method; syntax such as `int x = Clients.All.add(1,1)` does not work.

You can specify complex types and arrays for the parameters. The following example passes a complex type to the client in a method parameter.

Server code that calls a client method using a complex object

```
public async Task SendMessage(string name, string message)
{
    await Clients.All.addContosoChatMessageToPage(new ContosoChatMessage() { UserName = name, Message =
message });
}
```

Server code that defines the complex object

```
public class ContosoChatMessage
{
    public string UserName { get; set; }
    public string Message { get; set; }
}
```

JavaScript client using generated proxy

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addMessageToPage = function (message) {
    console.log(message.UserName + ' ' + message.Message);
});
```

Selecting which clients will receive the RPC

The Clients property returns a HubConnectionContext object that provides several options for specifying which clients will receive the RPC:

- All connected clients.

  ```
  Clients.All.addContosoChatMessageToPage(name, message);
  ```

- Only the calling client.

  ```
  Clients.Caller.addContosoChatMessageToPage(name, message);
  ```

- All clients except the calling client.

  ```
  Clients.Others.addContosoChatMessageToPage(name, message);
  ```

- A specific client identified by connection ID.

```
Clients.Client(Context.ConnectionId).addContosoChatMessageToPage(name, message);
```

This example calls `addContosoChatMessageToPage` on the calling client and has the same effect as using `Clients.Caller`.

- All connected clients except the specified clients, identified by connection ID.

```
Clients.AllExcept(connectionId1, connectionId2).addContosoChatMessageToPage(name, message);
```

- All connected clients in a specified group.

```
Clients.Group(groupName).addContosoChatMessageToPage(name, message);
```

- All connected clients in a specified group except the specified clients, identified by connection ID.

```
Clients.Group(groupName, connectionId1, connectionId2).addContosoChatMessageToPage(name, message);
```

- All connected clients in a specified group except the calling client.

```
Clients.OthersInGroup(groupName).addContosoChatMessageToPage(name, message);
```

- A specific user, identified by userId.

```
Clients.User(userid).addContosoChatMessageToPage(name, message);
```

By default, this is `IPrincipal.Identity.Name`, but this can be changed by registering an implementation of IUserIdProvider with the global host.

- All clients and groups in a list of connection IDs.

```
Clients.Clients(ConnectionIds).broadcastMessage(name, message);
```

- A list of groups.

```
Clients.Groups(GroupIds).broadcastMessage(name, message);
```

- A user by name.

```
Clients.Client(username).broadcastMessage(name, message);
```

- A list of user names (introduced in SignalR 2.1).

```
Clients.Users(new string[] { "myUser", "myUser2" }).broadcastMessage(name, message);
```

No compile-time validation for method names

The method name that you specify is interpreted as a dynamic object, which means there is no IntelliSense or compile-time validation for it. The expression is evaluated at run time. When the method call executes, SignalR sends the method name and the parameter values to the client, and if the client has a method that matches the

name, that method is called and the parameter values are passed to it. If no matching method is found on the client, no error is raised. For information about the format of the data that SignalR transmits to the client behind the scenes when you call a client method, see Introduction to SignalR.

Case-insensitive method name matching

Method name matching is case-insensitive. For example, `Clients.All.addContosoChatMessageToPage` on the server will execute `AddContosoChatMessageToPage`, `addcontosochatmessagetopage`, or `addContosoChatMessageToPage` on the client.

Asynchronous execution

The method that you call executes asynchronously. Any code that comes after a method call to a client will execute immediately without waiting for SignalR to finish transmitting data to clients unless you specify that the subsequent lines of code should wait for method completion. The following code example shows how to execute two client methods sequentially.

Using Await (.NET 4.5)

```
public async Task NewContosoChatMessage(string name, string message)
{
    await Clients.Others.addContosoChatMessageToPage(data);
    await Clients.Caller.notifyMessageSent();
}
```

If you use `await` to wait until a client method finishes before the next line of code executes, that does not mean that clients will actually receive the message before the next line of code executes. "Completion" of a client method call only means that SignalR has done everything necessary to send the message. If you need verification that clients received the message, you have to program that mechanism yourself. For example, you could code a `MessageReceived` method on the Hub, and in the `addContosoChatMessageToPage` method on the client you could call `MessageReceived` after you do whatever work you need to do on the client. In `MessageReceived` in the Hub you can do whatever work depends on actual client reception and processing of the original method call.

How to use a string variable as the method name

If you want to invoke a client method by using a string variable as the method name, cast `Clients.All` (or `Clients.Others`, `Clients.Caller`, etc.) to `IClientProxy` and then call Invoke(methodName, args...).

```
public async Task NewContosoChatMessage(string name, string message)
{
    string methodToCall = "addContosoChatMessageToPage";
    IClientProxy proxy = Clients.All;
    await proxy.Invoke(methodToCall, name, message);
}
```

How to manage group membership from the Hub class

Groups in SignalR provide a method for broadcasting messages to specified subsets of connected clients. A group can have any number of clients, and a client can be a member of any number of groups.

To manage group membership, use the Add and Remove methods provided by the `Groups` property of the Hub class. The following example shows the `Groups.Add` and `Groups.Remove` methods used in Hub methods that are called by client code, followed by JavaScript client code that calls them.

Server

```
public class ContosoChatHub : Hub
{
    public Task JoinGroup(string groupName)
    {
        return Groups.Add(Context.ConnectionId, groupName);
    }

    public Task LeaveGroup(string groupName)
    {
        return Groups.Remove(Context.ConnectionId, groupName);
    }
}
```

JavaScript client using generated proxy

```
contosoChatHubProxy.server.joinGroup(groupName);
```

```
contosoChatHubProxy.server.leaveGroup(groupName);
```

You don't have to explicitly create groups. In effect a group is automatically created the first time you specify its name in a call to `Groups.Add`, and it is deleted when you remove the last connection from membership in it.

There is no API for getting a group membership list or a list of groups. SignalR sends messages to clients and groups based on a pub/sub model, and the server does not maintain lists of groups or group memberships. This helps maximize scalability, because whenever you add a node to a web farm, any state that SignalR maintains has to be propagated to the new node.

Asynchronous execution of Add and Remove methods

The `Groups.Add` and `Groups.Remove` methods execute asynchronously. If you want to add a client to a group and immediately send a message to the client by using the group, you have to make sure that the `Groups.Add` method finishes first. The following code example shows how to do that.

Adding a client to a group and then messaging that client

```
public async Task JoinGroup(string groupName)
{
    await Groups.Add(Context.ConnectionId, groupName);
    await Clients.Group(groupname).addContosoChatMessageToPage(Context.ConnectionId + " added to group");
}
```

Group membership persistence

SignalR tracks connections, not users, so if you want a user to be in the same group every time the user establishes a connection, you have to call `Groups.Add` every time the user establishes a new connection.

After a temporary loss of connectivity, sometimes SignalR can restore the connection automatically. In that case, SignalR is restoring the same connection, not establishing a new connection, and so the client's group membership is automatically restored. This is possible even when the temporary break is the result of a server reboot or failure, because connection state for each client, including group memberships, is round-tripped to the client. If a server goes down and is replaced by a new server before the connection times out, a client can reconnect automatically to the new server and re-enroll in groups it is a member of.

When a connection can't be restored automatically after a loss of connectivity, or when the connection times out, or when the client disconnects (for example, when a browser navigates to a new page), group memberships are lost. The next time the user connects will be a new connection. To maintain group memberships when the same user establishes a new connection, your application has to track the associations between users and groups, and

restore group memberships each time a user establishes a new connection.

For more information about connections and reconnections, see How to handle connection lifetime events in the Hub class later in this topic.

Single-user groups

Applications that use SignalR typically have to keep track of the associations between users and connections in order to know which user has sent a message and which user(s) should be receiving a message. Groups are used in one of the two commonly used patterns for doing that.

- Single-user groups.

  You can specify the user name as the group name, and add the current connection ID to the group every time the user connects or reconnects. To send messages to the user you send to the group. A disadvantage of this method is that the group doesn't provide you with a way to find out if the user is online or offline.

- Track associations between user names and connection IDs.

  You can store an association between each user name and one or more connection IDs in a dictionary or database, and update the stored data each time the user connects or disconnects. To send messages to the user you specify the connection IDs. A disadvantage of this method is that it takes more memory.

How to handle connection lifetime events in the Hub class

Typical reasons for handling connection lifetime events are to keep track of whether a user is connected or not, and to keep track of the association between user names and connection IDs. To run your own code when clients connect or disconnect, override the `OnConnected`, `OnDisconnected`, and `OnReconnected` virtual methods of the Hub class, as shown in the following example.

```
public class ContosoChatHub : Hub
{
    public override Task OnConnected()
    {
        // Add your own code here.
        // For example: in a chat application, record the association between
        // the current connection ID and user name, and mark the user as online.
        // After the code in this method completes, the client is informed that
        // the connection is established; for example, in a JavaScript client,
        // the start().done callback is executed.
        return base.OnConnected();
    }

    public override Task OnDisconnected(bool stopCalled)
    {
        // Add your own code here.
        // For example: in a chat application, mark the user as offline,
        // delete the association between the current connection id and user name.
        return base.OnDisconnected(stopCalled);
    }

    public override Task OnReconnected()
    {
        // Add your own code here.
        // For example: in a chat application, you might have marked the
        // user as offline after a period of inactivity; in that case
        // mark the user as online again.
        return base.OnReconnected();
    }
}
```

When OnConnected, OnDisconnected, and OnReconnected are called

Each time a browser navigates to a new page, a new connection has to be established, which means SignalR will execute the `OnDisconnected` method followed by the `OnConnected` method. SignalR always creates a new connection ID when a new connection is established.

The `OnReconnected` method is called when there has been a temporary break in connectivity that SignalR can automatically recover from, such as when a cable is temporarily disconnected and reconnected before the connection times out. The `OnDisconnected` method is called when the client is disconnected and SignalR can't automatically reconnect, such as when a browser navigates to a new page. Therefore, a possible sequence of events for a given client is `OnConnected`, `OnReconnected`, `OnDisconnected`; or `OnConnected`, `OnDisconnected`. You won't see the sequence `OnConnected`, `OnDisconnected`, `OnReconnected` for a given connection.

The `OnDisconnected` method doesn't get called in some scenarios, such as when a server goes down or the App Domain gets recycled. When another server comes on line or the App Domain completes its recycle, some clients may be able to reconnect and fire the `OnReconnected` event.

For more information, see Understanding and Handling Connection Lifetime Events in SignalR.

Caller state not populated

The connection lifetime event handler methods are called from the server, which means that any state that you put in the `state` object on the client will not be populated in the `Caller` property on the server. For information about the `state` object and the `Caller` property, see How to pass state between clients and the Hub class later in this topic.

How to get information about the client from the Context property

To get information about the client, use the `Context` property of the Hub class. The `Context` property returns a HubCallerContext object which provides access to the following information:

- The connection ID of the calling client.

  ```
  string connectionID = Context.ConnectionId;
  ```

  The connection ID is a GUID that is assigned by SignalR (you can't specify the value in your own code). There is one connection ID for each connection, and the same connection ID is used by all Hubs if you have multiple Hubs in your application.

- HTTP header data.

  ```
  System.Collections.Specialized.NameValueCollection headers = Context.Request.Headers;
  ```

  You can also get HTTP headers from `Context.Headers`. The reason for multiple references to the same thing is that `Context.Headers` was created first, the `Context.Request` property was added later, and `Context.Headers` was retained for backward compatibility.

- Query string data.

  ```
  System.Collections.Specialized.NameValueCollection queryString = Context.Request.QueryString;
  string parameterValue = queryString["parametername"]
  ```

  You can also get query string data from `Context.QueryString`.

  The query string that you get in this property is the one that was used with the HTTP request that established the SignalR connection. You can add query string parameters in the client by configuring the connection, which is a convenient way to pass data about the client from the client to the server. The

following example shows one way to add a query string in a JavaScript client when you use the generated proxy.

```
$.connection.hub.qs = { "version" : "1.0" };
```

For more information about setting query string parameters, see the API guides for the JavaScript and .NET clients.

You can find the transport method used for the connection in the query string data, along with some other values used internally by SignalR:

```
string transportMethod = queryString["transport"];
```

The value of `transportMethod` will be "webSockets", "serverSentEvents", "foreverFrame" or "longPolling". Note that if you check this value in the `OnConnected` event handler method, in some scenarios you might initially get a transport value that is not the final negotiated transport method for the connection. In that case the method will throw an exception and will be called again later when the final transport method is established.

- Cookies.

```
System.Collections.Generic.IDictionary<string, Cookie> cookies = Context.Request.Cookies;
```

You can also get cookies from `Context.RequestCookies`.

- User information.

```
System.Security.Principal.IPrincipal user = Context.User;
```

- The HttpContext object for the request :

```
System.Web.HttpContextBase httpContext = Context.Request.GetHttpContext();
```

Use this method instead of getting `HttpContext.Current` to get the `HttpContext` object for the SignalR connection.

How to pass state between clients and the Hub class

The client proxy provides a `state` object in which you can store data that you want to be transmitted to the server with each method call. On the server you can access this data in the `Clients.Caller` property in Hub methods that are called by clients. The `Clients.Caller` property is not populated for the connection lifetime event handler methods `OnConnected`, `OnDisconnected`, and `OnReconnected`.

Creating or updating data in the `state` object and the `Clients.Caller` property works in both directions. You can update values in the server and they are passed back to the client.

The following example shows JavaScript client code that stores state for transmission to the server with every method call.

```
contosoChatHubProxy.state.userName = "Fadi Fakhouri";
contosoChatHubProxy.state.computerName = "fadivm1";
```

The following example shows the equivalent code in a .NET client.

```
contosoChatHubProxy["userName"] = "Fadi Fakhouri";
chatHubProxy["computerName"] = "fadivm1";
```

In your Hub class, you can access this data in the `Clients.Caller` property. The following example shows code that retrieves the state referred to in the previous example.

```
public async Task NewContosoChatMessage(string data)
{
    string userName = Clients.Caller.userName;
    string computerName = Clients.Caller.computerName;
    await Clients.Others.addContosoChatMessageToPage(message, userName, computerName);
}
```

> **NOTE**
> This mechanism for persisting state is not intended for large amounts of data, since everything you put in the `state` or `Clients.Caller` property is round-tripped with every method invocation. It's useful for smaller items such as user names or counters.

In VB.NET or in a strongly-typed hub, the caller state object can't be accessed through `Clients.Caller`; instead, use `Clients.CallerState` (introduced in SignalR 2.1):

Using CallerState in C#

```
public async Task NewContosoChatMessage(string data)
{
    string userName = Clients.CallerState.userName;
    string computerName = Clients.CallerState.computerName;
    await Clients.Others.addContosoChatMessageToPage(data, userName, computerName);
}
```

Using CallerState in Visual Basic

```
Public Async Function NewContosoChatMessage(message As String) As Task
    Dim userName As String = Clients.CallerState.userName
    Dim computerName As String = Clients.CallerState.computerName
    Await Clients.Others.addContosoChatMessageToPage(message, userName, computerName)
End Sub
```

How to handle errors in the Hub class

To handle errors that occur in your Hub class methods, first ensure you "observe" any exceptions from async operations (such as invoking client methods) by using `await`. Then use one or more of the following methods:

- Wrap your method code in try-catch blocks and log the exception object. For debugging purposes you can send the exception to the client, but for security reasons sending detailed information to clients in production is not recommended.
- Create a Hubs pipeline module that handles the OnIncomingError method. The following example shows a pipeline module that logs errors, followed by code in Startup.cs that injects the module into the Hubs pipeline.

```
public class ErrorHandlingPipelineModule : HubPipelineModule
{
    protected override void OnIncomingError(ExceptionContext exceptionContext,
IHubIncomingInvokerContext invokerContext)
    {
        Debug.WriteLine("=> Exception " + exceptionContext.Error.Message);
        if (exceptionContext.Error.InnerException != null)
        {
            Debug.WriteLine("=> Inner Exception " + exceptionContext.Error.InnerException.Message);
        }
        base.OnIncomingError(exceptionContext, invokerContext);

    }
}
```

```
public void Configuration(IAppBuilder app)
{
    // Any connection or hub wire up and configuration should go here
    GlobalHost.HubPipeline.AddModule(new ErrorHandlingPipelineModule());
    app.MapSignalR();
}
```

- Use the `HubException` class (introduced in SignalR 2). This error can be thrown from any hub invocation. The `HubError` constructor takes a string message, and an object for storing extra error data. SignalR will auto-serialize the exception and send it to the client, where it will be used to reject or fail the hub method invocation.

  The following code samples demonstrate how to throw a `HubException` during a Hub invocation, and how to handle the exception on JavaScript and .NET clients.

  #### Server code demonstrating the HubException class

```
public class MyHub : Hub
{
    public async Task Send(string message)
    {
        if(message.Contains("<script>"))
        {
            throw new HubException("This message will flow to the client", new { user =
Context.User.Identity.Name, message = message });
        }

        await Clients.All.send(message);
    }
}
```

  #### JavaScript client code demonstrating response to throwing a HubException in a hub

```
myHub.server.send("<script>")
            .fail(function (e) {
                if (e.source === 'HubException') {
                    console.log(e.message + ' : ' + e.data.user);
                }
            });
```

  #### .NET client code demonstrating response to throwing a HubException in a hub

```
try
{
    await myHub.Invoke("Send", "<script>");
}
catch(HubException ex)
{
    Console.WriteLine(ex.Message);
}
```

For more information about Hub pipeline modules, see How to customize the Hubs pipeline later in this topic.

How to enable tracing

To enable server-side tracing, add a system.diagnostics element to your Web.config file, as shown in this example:

```
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit https://go.microsoft.com/fwlink/?
LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework, Version=5.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false"
/>
  </configSections>
  <connectionStrings>
    <add name="SignalRSamples" connectionString="Data Source=(local);Initial
Catalog=SignalRSamples;Integrated Security=SSPI;Asynchronous Processing=True;" />
    <add name="SignalRSamplesWithMARS" connectionString="Data Source=(local);Initial
Catalog=SignalRSamples;Integrated Security=SSPI;Asynchronous Processing=True;MultipleActiveResultSets=true;"
/>
  </connectionStrings>
  <system.web>
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>
  <system.webServer>
    <modules runAllManagedModulesForAllRequests="true" />
  </system.webServer>
  <system.diagnostics>
    <sources>
      <source name="SignalR.SqlMessageBus">
        <listeners>
          <add name="SignalR-Bus" />
        </listeners>
      </source>
     <source name="SignalR.ServiceBusMessageBus">
        <listeners>
            <add name="SignalR-Bus" />
        </listeners>
     </source>
     <source name="SignalR.ScaleoutMessageBus">
        <listeners>
            <add name="SignalR-Bus" />
        </listeners>
      </source>
      <source name="SignalR.Transports.WebSocketTransport">
        <listeners>
          <add name="SignalR-Transports" />
        </listeners>
      </source>
      <source name="SignalR.Transports.ServerSentEventsTransport">
          <listeners>
              <add name="SignalR-Transports" />
          </listeners>
      </source>
      <source name="SignalR.Transports.ForeverFrameTransport">
```

```
          <listeners>
              <add name="SignalR-Transports" />
          </listeners>
      </source>
      <source name="SignalR.Transports.LongPollingTransport">
        <listeners>
            <add name="SignalR-Transports" />
        </listeners>
      </source>
      <source name="SignalR.Transports.TransportHeartBeat">
        <listeners>
            <add name="SignalR-Transports" />
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="SignalRSwitch" value="Verbose" />
    </switches>
    <sharedListeners>
      <add name="SignalR-Transports"
           type="System.Diagnostics.TextWriterTraceListener"
           initializeData="transports.log.txt" />
        <add name="SignalR-Bus"
           type="System.Diagnostics.TextWriterTraceListener"
           initializeData="bus.log.txt" />
    </sharedListeners>
    <trace autoflush="true" />
  </system.diagnostics>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
EntityFramework">
      <parameters>
        <parameter value="v11.0" />
      </parameters>
    </defaultConnectionFactory>
  </entityFramework>
</configuration>
```

When you run the application in Visual Studio, you can view the logs in the **Output** window.

How to call client methods and manage groups from outside the Hub class

To call client methods from a different class than your Hub class, get a reference to the SignalR context object for the Hub and use that to call methods on the client or manage groups.

The following sample `StockTicker` class gets the context object, stores it in an instance of the class, stores the class instance in a static property, and uses the context from the singleton class instance to call the `updateStockPrice` method on clients that are connected to a Hub named `StockTickerHub`.

```
// For the complete example, go to
// http://www.asp.net/signalr/overview/getting-started/tutorial-server-broadcast-with-aspnet-signalr
// This sample only shows code related to getting and using the SignalR context.
public class StockTicker
{
    // Singleton instance
    private readonly static Lazy<StockTicker> _instance = new Lazy<StockTicker>(
        () => new StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>()));

    private IHubContext _context;

    private StockTicker(IHubContext context)
    {
        _context = context;
    }

    // This method is invoked by a Timer object.
    private void UpdateStockPrices(object state)
    {
        foreach (var stock in _stocks.Values)
        {
            if (TryUpdateStockPrice(stock))
            {
                _context.Clients.All.updateStockPrice(stock);
            }
        }
    }
```

If you need to use the context multiple-times in a long-lived object, get the reference once and save it rather than getting it again each time. Getting the context once ensures that SignalR sends messages to clients in the same sequence in which your Hub methods make client method invocations. For a tutorial that shows how to use the SignalR context for a Hub, see Server Broadcast with ASP.NET SignalR.

Calling client methods

You can specify which clients will receive the RPC, but you have fewer options than when you call from a Hub class. The reason for this is that the context is not associated with a particular call from a client, so any methods that require knowledge of the current connection ID, such as `Clients.Others`, or `Clients.Caller`, or `Clients.OthersInGroup`, are not available. The following options are available:

- All connected clients.

  ```
  context.Clients.All.addContosoChatMessageToPage(name, message);
  ```

- A specific client identified by connection ID.

  ```
  context.Clients.Client(connectionID).addContosoChatMessageToPage(name, message);
  ```

- All connected clients except the specified clients, identified by connection ID.

  ```
  context.Clients.AllExcept(connectionId1, connectionId2).addContosoChatMessageToPage(name, message);
  ```

- All connected clients in a specified group.

  ```
  context.Clients.Group(groupName).addContosoChatMessageToPage(name, message);
  ```

- All connected clients in a specified group except specified clients, identified by connection ID.

```
Clients.Group(groupName, connectionId1, connectionId2).addContosoChatMessageToPage(name, message);
```

If you are calling into your non-Hub class from methods in your Hub class, you can pass in the current connection ID and use that with `Clients.Client`, `Clients.AllExcept`, or `Clients.Group` to simulate `Clients.Caller`, `Clients.Others`, or `Clients.OthersInGroup`. In the following example, the `MoveShapeHub` class passes the connection ID to the `Broadcaster` class so that the `Broadcaster` class can simulate `Clients.Others`.

```
// For the complete example, see
// http://www.asp.net/signalr/overview/signalr-20/getting-started-with-signalr-20/tutorial-server-broadcast-
with-signalr-20
// This sample only shows code that passes connection ID to the non-Hub class,
// in order to simulate Clients.Others.
public class MoveShapeHub : Hub
{
    // Code not shown puts a singleton instance of Broadcaster in this variable.
    private Broadcaster _broadcaster;

    public void UpdateModel(ShapeModel clientModel)
    {
        clientModel.LastUpdatedBy = Context.ConnectionId;
        // Update the shape model within our broadcaster
        _broadcaster.UpdateShape(clientModel);
    }
}

public class Broadcaster
{
    public Broadcaster()
    {
        _hubContext = GlobalHost.ConnectionManager.GetHubContext<MoveShapeHub>();
    }

    public void UpdateShape(ShapeModel clientModel)
    {
        _model = clientModel;
        _modelUpdated = true;
    }

    // Called by a Timer object.
    public void BroadcastShape(object state)
    {
        if (_modelUpdated)
        {
            _hubContext.Clients.AllExcept(_model.LastUpdatedBy).updateShape(_model);
            _modelUpdated = false;
        }
    }
}
```

Managing group membership

For managing groups you have the same options as you do in a Hub class.

- Add a client to a group

```
context.Groups.Add(connectionID, groupName);
```

- Remove a client from a group

```
context.Groups.Remove(connectionID, groupName);
```

How to customize the Hubs pipeline

SignalR enables you to inject your own code into the Hub pipeline. The following example shows a custom Hub pipeline module that logs each incoming method call received from the client and outgoing method call invoked on the client:

```
public class LoggingPipelineModule : HubPipelineModule
{
    protected override bool OnBeforeIncoming(IHubIncomingInvokerContext context)
    {
        Debug.WriteLine("=> Invoking " + context.MethodDescriptor.Name + " on hub " +
context.MethodDescriptor.Hub.Name);
        return base.OnBeforeIncoming(context);
    }
    protected override bool OnBeforeOutgoing(IHubOutgoingInvokerContext context)
    {
        Debug.WriteLine("<= Invoking " + context.Invocation.Method + " on client hub " +
context.Invocation.Hub);
        return base.OnBeforeOutgoing(context);
    }
}
```

The following code in the *Startup.cs* file registers the module to run in the Hub pipeline:

```
public void Configuration(IAppBuilder app)
{
    GlobalHost.HubPipeline.AddModule(new LoggingPipelineModule());
    app.MapSignalR();
}
```

There are many different methods that you can override. For a complete list, see HubPipelineModule Methods.

ASP.NET SignalR Hubs API Guide - JavaScript Client

2/20/2020 • 22 minutes to read • Edit Online

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This document provides an introduction to using the Hubs API for SignalR version 2 in JavaScript clients, such as browsers and Windows Store (WinJS) applications.

The SignalR Hubs API enables you to make remote procedure calls (RPCs) from a server to connected clients and from clients to the server. In server code, you define methods that can be called by clients, and you call methods that run on the client. In client code, you define methods that can be called from the server, and you call methods that run on the server. SignalR takes care of all of the client-to-server plumbing for you.

SignalR also offers a lower-level API called Persistent Connections. For an introduction to SignalR, Hubs, and Persistent Connections, see Introduction to SignalR.

Software versions used in this topic

- Visual Studio 2017
- .NET 4.5
- SignalR version 2

Previous versions of this topic

For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Overview

This document contains the following sections:

- The generated proxy and what it does for you
  - When to use the generated proxy
- Client Setup
  - How to reference the dynamically generated proxy
  - How to create a physical file for the SignalR generated proxy
- How to establish a connection
  - $.connection.hub is the same object that $.hubConnection() creates
  - Asynchronous execution of the start method
- How to establish a cross-domain connection

- How to configure the connection
  - How to specify query string parameters
  - How to specify the transport method
- How to get a proxy for a Hub class
- How to define methods on the client that the server can call
- How to call server methods from the client
- How to handle connection lifetime events
- How to handle errors
- How to enable client-side logging

For documentation on how to program the server or .NET clients, see the following resources:

- SignalR Hubs API Guide - Server
- SignalR Hubs API Guide - .NET Client

The SignalR 2 server component is only available on .NET 4.5 (though there is a .NET client for SignalR 2 on .NET 4.0).

The generated proxy and what it does for you

You can program a JavaScript client to communicate with a SignalR service with or without a proxy that SignalR generates for you. What the proxy does for you is simplify the syntax of the code you use to connect, write methods that the server calls, and call methods on the server.

When you write code to call server methods, the generated proxy enables you to use syntax that looks as though you were executing a local function: you can write `serverMethod(arg1, arg2)` instead of `invoke('serverMethod', arg1, arg2)`. The generated proxy syntax also enables an immediate and intelligible client-side error if you mistype a server method name. And if you manually create the file that defines the proxies, you can also get IntelliSense support for writing code that calls server methods.

For example, suppose you have the following Hub class on the server:

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addContosoChatMessageToPage(name, message);
    }
}
```

The following code examples show what JavaScript code looks like for invoking the `NewContosoChatMessage` method on the server and receiving invocations of the `addContosoChatMessageToPage` method from the server.

With the generated proxy

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addContosoChatMessageToPage = function (name, message) {
    console.log(name + ' ' + message);
};
$.connection.hub.start().done(function () {
    // Wire up Send button to call NewContosoChatMessage on the server.
    $('#newContosoChatMessage').click(function () {
         contosoChatHubProxy.server.newContosoChatMessage($('#displayname').val(), $('#message').val());
         $('#message').val('').focus();
     });
});
```

Without the generated proxy

```
var connection = $.hubConnection();
var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
contosoChatHubProxy.on('addContosoChatMessageToPage', function(name, message) {
    console.log(name + ' ' + message);
});
connection.start().done(function() {
    // Wire up Send button to call NewContosoChatMessage on the server.
    $('#newContosoChatMessage').click(function () {
        contosoChatHubProxy.invoke('newContosoChatMessage', $('#displayname').val(), $('#message').val());
        $('#message').val('').focus();
                });
    });
```

When to use the generated proxy

If you want to register multiple event handlers for a client method that the server calls, you can't use the generated proxy. Otherwise, you can choose to use the generated proxy or not based on your coding preference. If you choose not to use it, you don't have to reference the "signalr/hubs" URL in a `script` element in your client code.

Client setup

A JavaScript client requires references to jQuery and the SignalR core JavaScript file. The jQuery version must be 1.6.4 or major later versions, such as 1.7.2, 1.8.2, or 1.9.1. If you decide to use the generated proxy, you also need a reference to the SignalR generated proxy JavaScript file. The following example shows what the references might look like in an HTML page that uses the generated proxy.

```
<script src="Scripts/jquery-1.10.2.min.js"></script>
<script src="Scripts/jquery.signalR-2.1.0.min.js"></script>
<script src="signalr/hubs"></script>
```

These references must be included in this order: jQuery first, SignalR core after that, and SignalR proxies last.

How to reference the dynamically generated proxy

In the preceding example, the reference to the SignalR generated proxy is to dynamically generated JavaScript code, not to a physical file. SignalR creates the JavaScript code for the proxy on the fly and serves it to the client in response to the "/signalr/hubs" URL. If you specified a different base URL for SignalR connections on the server in your `MapSignalR` method, the URL for the dynamically generated proxy file is your custom URL with "/hubs" appended to it.

> **NOTE**
> For Windows 8 (Windows Store) JavaScript clients, use the physical proxy file instead of the dynamically generated one. For more information, see How to create a physical file for the SignalR generated proxy later in this topic.

In an ASP.NET MVC 4 or 5 Razor view, use the tilde to refer to the application root in your proxy file reference:

```
<script src="~/signalr/hubs"></script>
```

For more information about using SignalR in MVC 5, see Getting Started with SignalR and MVC 5.

In an ASP.NET MVC 3 Razor view, use `Url.Content` for your proxy file reference:

```
<script src="@Url.Content("~/signalr/hubs")"></script>
```

In an ASP.NET Web Forms application, use `ResolveClientUrl` for your proxies file reference or register it via the ScriptManager using an app root relative path (starting with a tilde):

```
<script src='<%: ResolveClientUrl("~/signalr/hubs") %>'></script>
```

As a general rule, use the same method for specifying the "/signalr/hubs" URL that you use for CSS or JavaScript files. If you specify a URL without using a tilde, in some scenarios your application will work correctly when you test in Visual Studio using IIS Express but will fail with a 404 error when you deploy to full IIS. For more information, see **Resolving References to Root-Level Resources** in Web Servers in Visual Studio for ASP.NET Web Projects on the MSDN site.

When you run a web project in Visual Studio 2017 in debug mode, and if you use Internet Explorer as your browser, you can see the proxy file in **Solution Explorer** under **Scripts**.

To see the contents of the file, double-click **hubs**. If you are not using Visual Studio 2012 or 2013 and Internet Explorer, or if you are not in debug mode, you can also get the contents of the file by browsing to the "/signalR/hubs" URL. For example, if your site is running at `http://localhost:56699`, go to `http://localhost:56699/SignalR/hubs` in your browser.

How to create a physical file for the SignalR generated proxy

As an alternative to the dynamically generated proxy, you can create a physical file that has the proxy code and reference that file. You might want to do that for control over caching or bundling behavior, or to get IntelliSense when you are coding calls to server methods.

To create a proxy file, perform the following steps:

1. Install the Microsoft.AspNet.SignalR.Utils NuGet package.
2. Open a command prompt and browse to the *tools* folder that contains the SignalR.exe file. The tools folder is at the following location:

   `[your solution folder]\packages\Microsoft.AspNet.SignalR.Utils.2.1.0\tools`

3. Enter the following command:

   `signalr ghp /path:[path to the .dll that contains your Hub class]`

   The path to your *.dll* is typically the *bin* folder in your project folder.

   This command creates a file named *server.js* in the same folder as *signalr.exe*.

4. Put the *server.js* file in an appropriate folder in your project, rename it as appropriate for your application, and add a reference to it in place of the "signalr/hubs" reference.

How to establish a connection

Before you can establish a connection, you have to create a connection object, create a proxy, and register event handlers for methods that can be called from the server. When the proxy and event handlers are set up, establish the connection by calling the `start` method.

If you are using the generated proxy, you don't have to create the connection object in your own code because the generated proxy code does it for you.

Establish a connection (with the generated proxy)

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addContosoChatMessageToPage = function (name, message) {
    console.log(userName + ' ' + message);
};
$.connection.hub.start()
    .done(function(){ console.log('Now connected, connection ID=' + $.connection.hub.id); })
    .fail(function(){ console.log('Could not Connect!'); });
```

Establish a connection (without the generated proxy)

```
var connection = $.hubConnection();
var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
contosoChatHubProxy.on('addContosoChatMessageToPage', function(userName, message) {
    console.log(userName + ' ' + message);
});
connection.start()
    .done(function(){ console.log('Now connected, connection ID=' + connection.id); })
    .fail(function(){ console.log('Could not connect'); });
```

The sample code uses the default "/signalr" URL to connect to your SignalR service. For information about how to specify a different base URL, see ASP.NET SignalR Hubs API Guide - Server - The /signalr URL.

By default, the hub location is the current server; if you are connecting to a different server, specify the URL before calling the `start` method, as shown in the following example:

```
$.connection.hub.url = '<yourbackendurl>;
```

> **NOTE**
>
> Normally you register event handlers before calling the `start` method to establish the connection. If you want to register some event handlers after establishing the connection, you can do that, but you must register at least one of your event handler(s) before calling the `start` method. One reason for this is that there can be many Hubs in an application, but you wouldn't want to trigger the `OnConnected` event on every Hub if you are only going to use to one of them. When the connection is established, the presence of a client method on a Hub's proxy is what tells SignalR to trigger the `OnConnected` event. If you don't register any event handlers before calling the `start` method, you will be able to invoke methods on the Hub, but the Hub's `OnConnected` method won't be called and no client methods will be invoked from the server.

$.connection.hub is the same object that $.hubConnection() creates

As you can see from the examples, when you use the generated proxy, `$.connection.hub` refers to the connection object. This is the same object that you get by calling `$.hubConnection()` when you aren't using the generated

proxy. The generated proxy code creates the connection for you by executing the following statement:

```
// This line is at the end of the generated proxy code.
signalR.hub = $.hubConnection("/signalr", { useDefaultPath: false });
```

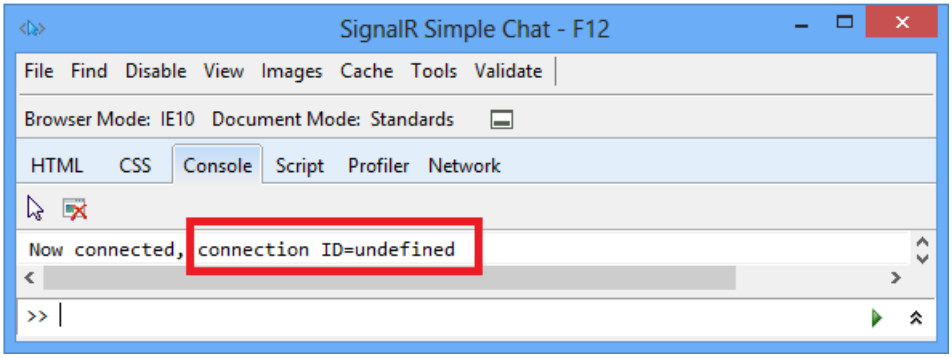
When you're using the generated proxy, you can do anything with `$.connection.hub` that you can do with a connection object when you're not using the generated proxy.

Asynchronous execution of the start method

The `start` method executes asynchronously. It returns a jQuery Deferred object, which means that you can add callback functions by calling methods such as `pipe`, `done`, and `fail`. If you have code that you want to execute after the connection is established, such as a call to a server method, put that code in a callback function or call it from a callback function. The `.done` callback method is executed after the connection has been established, and after any code that you have in your `OnConnected` event handler method on the server finishes executing.

If you put the "Now connected" statement from the preceding example as the next line of code after the `start` method call (not in a `.done` callback), the `console.log` line will execute before the connection is established, as shown in the following example:

```
$.connection.hub.start();
console.log("Now connected, connection ID=" + $.connection.hub.id);
```



How to establish a cross-domain connection

Typically if the browser loads a page from `http://contoso.com`, the SignalR connection is in the same domain, at `http://contoso.com/signalr`. If the page from `http://contoso.com` makes a connection to `http://fabrikam.com/signalr`, that is a cross-domain connection. For security reasons, cross-domain connections are disabled by default.

In SignalR 1.x, cross domain requests were controlled by a single EnableCrossDomain flag. This flag controlled both JSONP and CORS requests. For greater flexibility, all CORS support has been removed from the server component of SignalR (JavaScript clients still use CORS normally if it is detected that the browser supports it), and new OWIN middleware has been made available to support these scenarios.

If JSONP is required on the client (to support cross-domain requests in older browsers), it will need to be enabled explicitly by setting `EnableJSONP` on the `HubConfiguration` object to `true`, as shown below. JSONP is disabled by default, as it is less secure than CORS.

Adding Microsoft.Owin.Cors to your project: To install this library, run the following command in the Package Manager Console:

```
Install-Package Microsoft.Owin.Cors
```

This command will add the 2.1.0 version of the package to your project.

Calling UseCors

The following code snippet demonstrates how to implement cross-domain connections in SignalR 2.

Implementing cross-domain requests in SignalR 2

The following code demonstrates how to enable CORS or JSONP in a SignalR 2 project. This code sample uses `Map` and `RunSignalR` instead of `MapSignalR`, so that the CORS middleware runs only for the SignalR requests that require CORS support (rather than for all traffic at the path specified in `MapSignalR`.) Map can also be used for any other middleware that needs to run for a specific URL prefix, rather than for the entire application.

```
using Microsoft.AspNet.SignalR;
using Microsoft.Owin.Cors;
using Owin;
namespace MyWebApplication
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Branch the pipeline here for requests that start with "/signalr"
            app.Map("/signalr", map =>
            {
                // Setup the CORS middleware to run before SignalR.
                // By default this will allow all origins. You can
                // configure the set of origins and/or http verbs by
                // providing a cors options with a different policy.
                map.UseCors(CorsOptions.AllowAll);
                var hubConfiguration = new HubConfiguration
                {
                    // You can enable JSONP by uncommenting line below.
                    // JSONP requests are insecure but some older browsers (and some
                    // versions of IE) require JSONP to work cross domain
                    // EnableJSONP = true
                };
                // Run the SignalR pipeline. We're not using MapSignalR
                // since this branch already runs under the "/signalr"
                // path.
                map.RunSignalR(hubConfiguration);
            });
        }
    }
}
```

NOTE

- Don't set `jQuery.support.cors` to true in your code.

  ```
  // Don't put the following line in your code.
  jQuery.support.cors = true;
  $.connection.hub.start();
  ```

  SignalR handles the use of CORS. Setting `jQuery.support.cors` to true disables JSONP because it causes SignalR to assume the browser supports CORS.

- When you're connecting to a localhost URL, Internet Explorer 10 won't consider it a cross-domain connection, so the application will work locally with IE 10 even if you haven't enabled cross-domain connections on the server.
- For information about using cross-domain connections with Internet Explorer 9, see this StackOverflow thread.
- For information about using cross-domain connections with Chrome, see this StackOverflow thread.
- The sample code uses the default "/signalr" URL to connect to your SignalR service. For information about how to specify a different base URL, see ASP.NET SignalR Hubs API Guide - Server - The /signalr URL.

How to configure the connection

Before you establish a connection, you can specify query string parameters or specify the transport method.

How to specify query string parameters

If you want to send data to the server when the client connects, you can add query string parameters to the connection object. The following examples show how to set a query string parameter in client code.

Set a query string value before calling the start method (with the generated proxy)

```
$.connection.hub.qs = { 'version' : '1.0' };
```

Set a query string value before calling the start method (without the generated proxy)

```
var connection = $.hubConnection();
connection.qs = { 'version' : '1.0' };
```

The following example shows how to read a query string parameter in server code.

```
public class ContosoChatHub : Hub
{
    public override Task OnConnected()
    {
        var version = Context.QueryString['version'];
        if (version != '1.0')
        {
            Clients.Caller.notifyWrongVersion();
        }
        return base.OnConnected();
    }
}
```

How to specify the transport method

As part of the process of connecting, a SignalR client normally negotiates with the server to determine the best transport that is supported by both server and client. If you already know which transport you want to use, you can bypass this negotiation process by specifying the transport method when you call the `start` method.

Client code that specifies the transport method (with the generated proxy)

```
$.connection.hub.start( { transport: 'longPolling' });
```

Client code that specifies the transport method (without the generated proxy)

```
var connection = $.hubConnection();
connection.start({ transport: 'longPolling' });
```

As an alternative, you can specify multiple transport methods in the order in which you want SignalR to try them:

Client code that specifies a custom transport fallback scheme (with the generated proxy)

```
$.connection.hub.start( { transport: ['webSockets', 'longPolling'] });
```

Client code that specifies a custom transport fallback scheme (without the generated proxy)

```
var connection = $.hubConnection();
connection.start({ transport: ['webSockets', 'longPolling'] });
```

You can use the following values for specifying the transport method:

- "webSockets"
- "foreverFrame"
- "serverSentEvents"
- "longPolling"

The following examples show how to find out which transport method is being used by a connection.

Client code that displays the transport method used by a connection (with the generated proxy)

```
$.connection.hub.start().done(function () {
    console.log("Connected, transport = " + $.connection.hub.transport.name);
});
```

Client code that displays the transport method used by a connection (without the generated proxy)

```
var connection = $.hubConnection();
connection.hub.start().done(function () {
    console.log("Connected, transport = " + connection.transport.name);
});
```

For information about how to check the transport method in server code, see ASP.NET SignalR Hubs API Guide - Server - How to get information about the client from the Context property. For more information about transports and fallbacks, see Introduction to SignalR - Transports and Fallbacks.

How to get a proxy for a Hub class

Each connection object that you create encapsulates information about a connection to a SignalR service that contains one or more Hub classes. To communicate with a Hub class, you use a proxy object which you create yourself (if you're not using the generated proxy) or which is generated for you.

On the client the proxy name is a camel-cased version of the Hub class name. SignalR automatically makes this change so that JavaScript code can conform to JavaScript conventions.

Hub class on server

```
public class ContosoChatHub : Hub
```

Get a reference to the generated client proxy for the Hub

```
var myHubProxy = $.connection.contosoChatHub
```

Create client proxy for the Hub class (without generated proxy)

```
var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
```

If you decorate your Hub class with a `HubName` attribute, use the exact name without changing case.

Hub class on server with HubName attribute

```
[HubName("ContosoChatHub")]
public class ChatHub : Hub
```

Get a reference to the generated client proxy for the Hub

```
var contosoChatHubProxy = $.connection.ContosoChatHub
```

Create client proxy for the Hub class (without generated proxy)

```
var contosoChatHubProxy = connection.createHubProxy('ContosoChatHub');
```

How to define methods on the client that the server can call

To define a method that the server can call from a Hub, add an event handler to the Hub proxy by using the `client` property of the generated proxy, or call the `on` method if you aren't using the generated proxy. The parameters can be complex objects.

Add the event handler before you call the `start` method to establish the connection. (If you want to add event handlers after calling the `start` method, see the note in How to establish a connection earlier in this document, and use the syntax shown for defining a method without using the generated proxy.)

Method name matching is case-insensitive. For example, `Clients.All.addContosoChatMessageToPage` on the server will execute `AddContosoChatMessageToPage`, `addContosoChatMessageToPage`, or `addcontosochatmessagetopage` on the client.

Define method on client (with the generated proxy)

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addContosoChatMessageToPage = function (userName, message) {
    console.log(userName + ' ' + message);
};
$.connection.hub.start()
    .done(function(){ console.log('Now connected, connection ID=' + $.connection.hub.id); })
    .fail(function(){ console.log('Could not Connect!'); });
```

Alternate way to define method on client (with the generated proxy)

```
$.extend(contosoChatHubProxy.client, {
    addContosoChatMessageToPage: function(userName, message) {
    console.log(userName + ' ' + message);
    };
});
```

Define method on client (without the generated proxy, or when adding after calling the start method)

```
var connection = $.hubConnection();
var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
contosoChatHubProxy.on('addContosoChatMessageToPage', function(userName, message) {
    console.log(userName + ' ' + message);
});
connection.start()
    .done(function(){ console.log('Now connected, connection ID=' + connection.id); })
    .fail(function(){ console.log('Could not connect'); });
```

Server code that calls the client method

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addContosoChatMessageToPage(name, message);
    }
}
```

The following examples include a complex object as a method parameter.

Define method on client that takes a complex object (with the generated proxy)

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addMessageToPage = function (message) {
    console.log(message.UserName + ' ' + message.Message);
});
```

Define method on client that takes a complex object (without the generated proxy)

```
var connection = $.hubConnection();
var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
chatHubProxy.on('addMessageToPage', function (message) {
    console.log(message.UserName + ' ' + message.Message);
});
```

Server code that defines the complex object

```
public class ContosoChatMessage
{
    public string UserName { get; set; }
    public string Message { get; set; }
}
```

Server code that calls the client method using a complex object

```
public void SendMessage(string name, string message)
{
    Clients.All.addContosoChatMessageToPage(new ContosoChatMessage() { UserName = name, Message = message });
}
```

How to call server methods from the client

To call a server method from the client, use the `server` property of the generated proxy or the `invoke` method on the Hub proxy if you aren't using the generated proxy. The return value or parameters can be complex objects.

Pass in a camel-case version of the method name on the Hub. SignalR automatically makes this change so that JavaScript code can conform to JavaScript conventions.

The following examples show how to call a server method that doesn't have a return value and how to call a server method that does have a return value.

Server method with no HubMethodName attribute

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(ChatMessage message)
    {
        Clients.All.addContosoChatMessageToPage(message);
    }
}
```

Server code that defines the complex object passed in a parameter

```
public class ChatMessage
{
    public string UserName { get; set; }
    public string Message { get; set; }
}
```

Client code that invokes the server method (with the generated proxy)

```
contosoChatHubProxy.server.newContosoChatMessage({ UserName: userName, Message: message}).done(function () {
        console.log ('Invocation of NewContosoChatMessage succeeded');
    }).fail(function (error) {
        console.log('Invocation of NewContosoChatMessage failed. Error: ' + error);
    });
```

Client code that invokes the server method (without the generated proxy)

```
contosoChatHubProxy.invoke('newContosoChatMessage', { UserName: userName, Message: message}).done(function ()
{
        console.log ('Invocation of NewContosoChatMessage succeeded');
    }).fail(function (error) {
        console.log('Invocation of NewContosoChatMessage failed. Error: ' + error);
    });
```

If you decorated the Hub method with a `HubMethodName` attribute, use that name without changing case.

Server method with a HubMethodName attribute

```
public class ContosoChatHub : Hub
{
    [HubMethodName("NewContosoChatMessage")]
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addContosoChatMessageToPage(name, message);
    }
}
```

Client code that invokes the server method (with the generated proxy)

```
contosoChatHubProxy.server.NewContosoChatMessage(userName, message).done(function () {
        console.log ('Invocation of NewContosoChatMessage succeeded');
    }).fail(function (error) {
        console.log('Invocation of NewContosoChatMessage failed. Error: ' + error);
    });
```

Client code that invokes the server method (without the generated proxy)

```
contosoChatHubProxy.invoke('NewContosoChatMessage', userName, message).done(function () {
        console.log ('Invocation of NewContosoChatMessage succeeded');
    }).fail(function (error) {
        console.log('Invocation of NewContosoChatMessage failed. Error: ' + error);
    });
```

The preceding examples show how to call a server method that has no return value. The following examples show how to call a server method that has a return value.

Server code for a method that has a return value

```
public class StockTickerHub : Hub
{
    public IEnumerable<Stock> GetAllStocks()
    {
        return _stockTicker.GetAllStocks();
    }
}
```

The Stock class used for the return value

```
public class Stock
{
    public string Symbol { get; set; }
    public decimal Price { get; set; }
}
```

Client code that invokes the server method (with the generated proxy)

```
function init() {
    return stockTickerProxy.server.getAllStocks().done(function (stocks) {
        $.each(stocks, function () {
            var stock = this;
            console.log("Symbol=" + stock.Symbol + " Price=" + stock.Price);
        });
    }).fail(function (error) {
        console.log('Error: ' + error);
    });
}
```

Client code that invokes the server method (without the generated proxy)

```
function init() {
    return stockTickerProxy.invoke('getAllStocks').done(function (stocks) {
        $.each(stocks, function () {
            var stock = this;
            console.log("Symbol=" + stock.Symbol + " Price=" + stock.Price);
        });
    }).fail(function (error) {
        console.log('Error: ' + error);
    });
}
```

How to handle connection lifetime events

SignalR provides the following connection lifetime events that you can handle:

- `starting`: Raised before any data is sent over the connection.

- `received`: Raised when any data is received on the connection. Provides the received data.
- `connectionSlow`: Raised when the client detects a slow or frequently dropping connection.
- `reconnecting`: Raised when the underlying transport begins reconnecting.
- `reconnected`: Raised when the underlying transport has reconnected.
- `stateChanged`: Raised when the connection state changes. Provides the old state and the new state (Connecting, Connected, Reconnecting, or Disconnected).
- `disconnected`: Raised when the connection has disconnected.

For example, if you want to display warning messages when there are connection problems that might cause noticeable delays, handle the `connectionSlow` event.

Handle the connectionSlow event (with the generated proxy)

```
$.connection.hub.connectionSlow(function () {
    console.log('We are currently experiencing difficulties with the connection.')
});
```

Handle the connectionSlow event (without the generated proxy)

```
var connection = $.hubConnection();
connection.connectionSlow(function () {
    console.log('We are currently experiencing difficulties with the connection.')
});
```

For more information, see Understanding and Handling Connection Lifetime Events in SignalR.

How to handle errors

The SignalR JavaScript client provides an `error` event that you can add a handler for. You can also use the fail method to add a handler for errors that result from a server method invocation.

If you don't explicitly enable detailed error messages on the server, the exception object that SignalR returns after an error contains minimal information about the error. For example, if a call to `newContosoChatMessage` fails, the error message in the error object contains "`There was an error invoking Hub method 'contosoChatHub.newContosoChatMessage'.`" Sending detailed error messages to clients in production is not recommended for security reasons, but if you want to enable detailed error messages for troubleshooting purposes, use the following code on the server.

```
var hubConfiguration = new HubConfiguration();
hubConfiguration.EnableDetailedErrors = true;
app.MapSignalR(hubConfiguration);
```

The following example shows how to add a handler for the error event.

Add an error handler (with the generated proxy)

```
$.connection.hub.error(function (error) {
    console.log('SignalR error: ' + error)
});
```

Add an error handler (without the generated proxy)

```
var connection = $.hubConnection();
connection.error(function (error) {
    console.log('SignalR error: ' + error)
});
```

The following example shows how to handle an error from a method invocation.

Handle an error from a method invocation (with the generated proxy)

```
contosoChatHubProxy.newContosoChatMessage(userName, message)
    .fail(function(error) {
        console.log( 'newContosoChatMessage error: ' + error)
    });
```

Handle an error from a method invocation (without the generated proxy)

```
contosoChatHubProxy.invoke('newContosoChatMessage', userName, message)
    .fail(function(error) {
        console.log( 'newContosoChatMessage error: ' + error)
    });
```

If a method invocation fails, the `error` event is also raised, so your code in the `error` method handler and in the `.fail` method callback would execute.

How to enable client-side logging

To enable client-side logging on a connection, set the `logging` property on the connection object before you call the `start` method to establish the connection.

Enable logging (with the generated proxy)

```
$.connection.hub.logging = true;
$.connection.hub.start();
```

Enable logging (without the generated proxy)

```
var connection = $.hubConnection();
connection.logging = true;
connection.start();
```

To see the logs, open your browser's developer tools and go to the Console tab. For a tutorial that shows step-by-step instructions and screen shots that show how to do this, see Server Broadcast with ASP.NET Signalr - Enable Logging.

ASP.NET SignalR Hubs API Guide - .NET Client (C#)

2/20/2020 • 15 minutes to read • Edit Online

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This document provides an introduction to using the Hubs API for SignalR version 2 in .NET clients, such as Windows Store (WinRT), WPF, Silverlight, and console applications.

The SignalR Hubs API enables you to make remote procedure calls (RPCs) from a server to connected clients and from clients to the server. In server code, you define methods that can be called by clients, and you call methods that run on the client. In client code, you define methods that can be called from the server, and you call methods that run on the server. SignalR takes care of all of the client-to-server plumbing for you.

SignalR also offers a lower-level API called Persistent Connections. For an introduction to SignalR, Hubs, and Persistent Connections, or for a tutorial that shows how to build a complete SignalR application, see SignalR - Getting Started.

Software versions used in this topic

- Visual Studio 2017
- .NET 4.5
- SignalR version 2

Previous versions of this topic

For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Overview

This document contains the following sections:

- Client Setup
- How to establish a connection
  - Cross-domain connections from Silverlight clients
- How to configure the connection
  - How to set the maximum number of concurrent connections in WPF clients
  - How to specify query string parameters
  - How to specify the transport method
  - How to specify HTTP headers

- How to specify client certificates
- How to create the Hub proxy
- How to define methods on the client that the server can call
  - Methods without parameters
  - Methods with parameters, specifying parameter types
  - Methods with parameters, specifying dynamic objects for the parameters
  - How to remove a handler
- How to call server methods from the client
- How to handle connection lifetime events
- How to handle errors
- How to enable client-side logging
- WPF, Silverlight, and console application code samples for client methods that the server can call

For a sample .NET client projects, see the following resources:

- gustavo-armenta / SignalR-Samples on GitHub.com (WinRT, Silverlight, console app examples).
- DamianEdwards / SignalR-MoveShapeDemo / MoveShape.Desktop on GitHub.com (WPF example).
- SignalR / Microsoft.AspNet.SignalR.Client.Samples on GitHub.com (Console app example).

For documentation on how to program the server or JavaScript clients, see the following resources:

- SignalR Hubs API Guide - Server
- SignalR Hubs API Guide - JavaScript Client

Links to API Reference topics are to the .NET 4.5 version of the API. If you're using .NET 4, see the .NET 4 version of the API topics.

Client setup

Install the Microsoft.AspNet.SignalR.Client NuGet package (not the Microsoft.AspNet.SignalR package). This package supports WinRT, Silverlight, WPF, console application, and Windows Phone clients, for both .NET 4 and .NET 4.5.

If the version of SignalR that you have on the client is different from the version that you have on the server, SignalR is often able to adapt to the difference. For example, a server running SignalR version 2 will support clients that have 1.1.x installed as well as clients that have version 2 installed. If the difference between the version on the server and the version on the client is too great, or if the client is newer than the server, SignalR throws an `InvalidOperationException` exception when the client tries to establish a connection. The error message is "

```
You are using a version of the client that isn't compatible with the server. Client version X.X, server version X.X
```

".

How to establish a connection

Before you can establish a connection, you have to create a `HubConnection` object and create a proxy. To establish the connection, call the `Start` method on the `HubConnection` object.

```
using (var hubConnection = new HubConnection("http://www.contoso.com/"))
{
    IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
    stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new
price {1}", stock.Symbol, stock.Price));
    await hubConnection.Start();
}
```

> **NOTE**
>
> For JavaScript clients you have to register at least one event handler before calling the `Start` method to establish the connection. This is not necessary for .NET clients. For JavaScript clients, the generated proxy code automatically creates proxies for all Hubs that exist on the server, and registering a handler is how you indicate which Hubs your client intends to use. But for a .NET client you create Hub proxies manually, so SignalR assumes that you will be using any Hub that you create a proxy for.

The sample code uses the default "/signalr" URL to connect to your SignalR service. For information about how to specify a different base URL, see ASP.NET SignalR Hubs API Guide - Server - The /signalr URL.

The `Start` method executes asynchronously. To make sure that subsequent lines of code don't execute until after the connection is established, use `await` in an ASP.NET 4.5 asynchronous method or `.Wait()` in a synchronous method. Don't use `.Wait()` in a WinRT client.

```
await connection.Start();
```

```
connection.Start().Wait();
```

Cross-domain connections from Silverlight clients

For information about how to enable cross-domain connections from Silverlight clients, see Making a Service Available Across Domain Boundaries.

How to configure the connection

Before you establish a connection, you can specify any of the following options:

- Concurrent connections limit.
- Query string parameters.
- The transport method.
- HTTP headers.
- Client certificates.

How to set the maximum number of concurrent connections in WPF clients

In WPF clients, you might have to increase the maximum number of concurrent connections from its default value of 2. The recommended value is 10.

```
using (var hubConnection = new HubConnection("http://www.contoso.com/"))
{
    IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
    stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new
price {1}", stock.Symbol, stock.Price));
    ServicePointManager.DefaultConnectionLimit = 10;
    await hubConnection.Start();
}
```

For more information, see ServicePointManager.DefaultConnectionLimit.

How to specify query string parameters

If you want to send data to the server when the client connects, you can add query string parameters to the connection object. The following example shows how to set a query string parameter in client code.

```
var querystringData = new Dictionary<string, string>();
querystringData.Add("contosochatversion", "1.0");
var connection = new HubConnection("http://contoso.com/", querystringData);
```

The following example shows how to read a query string parameter in server code.

```
public class StockTickerHub : Hub
{
    public override Task OnConnected()
    {
        var version = Context.QueryString["contosochatversion"];
        if (version != "1.0")
        {
            Clients.Caller.notifyWrongVersion();
        }
        return base.OnConnected();
    }
}
```

How to specify the transport method

As part of the process of connecting, a SignalR client normally negotiates with the server to determine the best transport that is supported by both server and client. If you already know which transport you want to use, you can bypass this negotiation process. To specify the transport method, pass in a transport object to the Start method. The following example shows how to specify the transport method in client code.

```
using (var hubConnection = new HubConnection("http://www.contoso.com/"))
{
    IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
    stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new
price {1}", stock.Symbol, stock.Price));
    await hubConnection.Start(new LongPollingTransport());
}
```

The Microsoft.AspNet.SignalR.Client.Transports namespace includes the following classes that you can use to specify the transport.

- LongPollingTransport
- ServerSentEventsTransport
- WebSocketTransport (Available only when both server and client use .NET 4.5.)
- AutoTransport (Automatically chooses the best transport that is supported by both the client and the server. This is the default transport. Passing this in to the `Start` method has the same effect as not passing in

anything.)

The ForeverFrame transport is not included in this list because it is used only by browsers.

For information about how to check the transport method in server code, see ASP.NET SignalR Hubs API Guide - Server - How to get information about the client from the Context property. For more information about transports and fallbacks, see Introduction to SignalR - Transports and Fallbacks.

How to specify HTTP headers

To set HTTP headers, use the `Headers` property on the connection object. The following example shows how to add an HTTP header.

```
hubConnection = new hubConnection("http://www.contoso.com/");
connection.Headers.Add("headername", "headervalue");
IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new price
{1}", stock.Symbol, stock.Price));
await connection.Start();
```

How to specify client certificates

To add client certificates, use the `AddClientCertificate` method on the connection object.

```
hubConnection = new hubConnection("http://www.contoso.com/");
hubConnection.AddClientCertificate(X509Certificate.CreateFromCertFile("MyCert.cer"));
IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new price
{1}", stock.Symbol, stock.Price));
await connection.Start();
```

How to create the Hub proxy

In order to define methods on the client that a Hub can call from the server, and to invoke methods on a Hub at the server, create a proxy for the Hub by calling `CreateHubProxy` on the connection object. The string you pass in to `CreateHubProxy` is the name of your Hub class, or the name specified by the `HubName` attribute if one was used on the server. Name matching is case-insensitive.

Hub class on server

```
public class StockTickerHub : Hub
```

Create client proxy for the Hub class

```
using (var hubConnection = new HubConnection("http://www.contoso.com/"))
{
    IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
    stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new
price {1}", stock.Symbol, stock.Price));
    await hubConnection.Start();
}
```

If you decorate your Hub class with a `HubName` attribute, use that name.

Hub class on server

```
[HubName("stockTicker")]
public class StockTickerHub : Hub
```

Create client proxy for the Hub class

```
using (var hubConnection = new HubConnection("http://www.contoso.com/"))
{
    IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("stockTicker");
    stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
        Console.WriteLine("Stock update for {0} new price {1}", stock.Symbol, stock.Price));
    await hubConnection.Start();
}
```

If you call `HubConnection.CreateHubProxy` multiple times with the same `hubName`, you get the same cached `IHubProxy` object.

How to define methods on the client that the server can call

To define a method that the server can call, use the proxy's `On` method to register an event handler.

Method name matching is case-insensitive. For example, `Clients.All.UpdateStockPrice` on the server will execute `updateStockPrice`, `updatestockprice`, or `UpdateStockPrice` on the client.

Different client platforms have different requirements for how you write method code to update the UI. The examples shown are for WinRT (Windows Store .NET) clients. WPF, Silverlight, and console application examples are provided in a separate section later in this topic.

Methods without parameters

If the method you're handling does not have parameters, use the non-generic overload of the `On` method:

Server code calling client method without parameters

```
public class StockTickerHub : Hub
{
    public void NotifyAllClients()
    {
         Clients.All.Notify();
    }
}
```

WinRT Client code for method called from server without parameters (see WPF and Silverlight examples later in this topic)

```
using (var hubConnection = new HubConnection("http://www.contoso.com/"))
{
    IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
    stockTickerHub.On("notify", () =>
        // Context is a reference to SynchronizationContext.Current
        Context.Post(delegate
        {
            textBox.Text += "Notified!\n";
        }, null)
    );
    await hubConnection.Start();
}
```

Methods with parameters, specifying the parameter types

If the method you're handling has parameters, specify the types of the parameters as the generic types of the `On` method. There are generic overloads of the `On` method to enable you to specify up to 8 parameters (4 on Windows Phone 7). In the following example, one parameter is sent to the `UpdateStockPrice` method.

Server code calling client method with a parameter

```
public void BroadcastStockPrice(Stock stock)
{
    context.Clients.Others.UpdateStockPrice(stock);
}
```

The Stock class used for the parameter

```
public class Stock
{
    public string Symbol { get; set; }
    public decimal Price { get; set; }
}
```

WinRT Client code for a method called from server with a parameter (see WPF and Silverlight examples later in this topic)

```
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
    // Context is a reference to SynchronizationContext.Current
    Context.Post(delegate
    {
        textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
    }, null)
);
```

Methods with parameters, specifying dynamic objects for the parameters

As an alternative to specifying parameters as generic types of the `On` method, you can specify parameters as dynamic objects:

Server code calling client method with a parameter

```
public void BroadcastStockPrice(Stock stock)
{
    context.Clients.Others.UpdateStockPrice(stock);
}
```

The Stock class used for the parameter

```
public class Stock
{
    public string Symbol { get; set; }
    public decimal Price { get; set; }
}
```

WinRT Client code for a method called from server with a parameter, using a dynamic object for the parameter (see WPF and Silverlight examples later in this topic)

```
stockTickerHubProxy.On("UpdateStockPrice", stock =>
    // Context is a reference to SynchronizationContext.Current
    Context.Post(delegate
    {
        textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
    }, null)
);
```

How to remove a handler

To remove a handler, call its `Dispose` method.

Client code for a method called from server

```
var updateStockPriceHandler = stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
    Context.Post(delegate
    {
        textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
    }, null)
);
```

Client code to remove the handler

```
updateStockPriceHandler.Dispose();
```

How to call server methods from the client

To call a method on the server, use the `Invoke` method on the Hub proxy.

If the server method has no return value, use the non-generic overload of the `Invoke` method.

Server code for a method that has no return value

```
public class StockTickerHub : Hub
{
    public void JoinGroup(string groupName)
    {
        Groups.Add(Context.ConnectionId, groupName);
    }
}
```

Client code calling a method that has no return value

```
stockTickerHubProxy.Invoke("JoinGroup", "SignalRChatRoom");
```

If the server method has a return value, specify the return type as the generic type of the `Invoke` method.

Server code for a method that has a return value and takes a complex type parameter

```
public IEnumerable<Stock> AddStock(Stock stock)
{
    _stockTicker.AddStock(stock);
    return _stockTicker.GetAllStocks();
}
```

The Stock class used for the parameter and return value

```
public class Stock
{
    public string Symbol { get; set; }
    public decimal Price { get; set; }
}
```

Client code calling a method that has a return value and takes a complex type parameter, in an ASP.NET 4.5 async method

```
var stocks = await stockTickerHub.Invoke<IEnumerable<Stock>>("AddStock", new Stock() { Symbol = "MSFT" });
foreach (Stock stock in stocks)
{
    textBox.Text += string.Format("Symbol: {0} price: {1}\n", stock.Symbol, stock.Price);
}
```

Client code calling a method that has a return value and takes a complex type parameter, in a synchronous method

```
var stocks = stockTickerHub.Invoke<IEnumerable<Stock>>("AddStock", new Stock() { Symbol = "MSFT" }).Result;
foreach (Stock stock in stocks)
{
    textBox.Text += string.Format("Symbol: {0} price: {1}\n", stock.Symbol, stock.Price);
}
```

The `Invoke` method executes asynchronously and returns a `Task` object. If you don't specify `await` or `.Wait()`, the next line of code will execute before the method that you invoke has finished executing.

How to handle connection lifetime events

SignalR provides the following connection lifetime events that you can handle:

- `Received`: Raised when any data is received on the connection. Provides the received data.
- `ConnectionSlow`: Raised when the client detects a slow or frequently dropping connection.
- `Reconnecting`: Raised when the underlying transport begins reconnecting.
- `Reconnected`: Raised when the underlying transport has reconnected.
- `StateChanged`: Raised when the connection state changes. Provides the old state and the new state. For information about connection state values see ConnectionState Enumeration.
- `Closed`: Raised when the connection has disconnected.

For example, if you want to display warning messages for errors that are not fatal but cause intermittent connection problems, such as slowness or frequent dropping of the connection, handle the `ConnectionSlow` event.

```
hubConnection.ConnectionSlow += () => Console.WriteLine("Connection problems.");
```

For more information, see Understanding and Handling Connection Lifetime Events in SignalR.

How to handle errors

If you don't explicitly enable detailed error messages on the server, the exception object that SignalR returns after an error contains minimal information about the error. For example, if a call to `newContosoChatMessage` fails, the error message in the error object contains "
`There was an error invoking Hub method 'contosoChatHub.newContosoChatMessage'.`" Sending detailed error

messages to clients in production is not recommended for security reasons, but if you want to enable detailed error messages for troubleshooting purposes, use the following code on the server.

```
var hubConfiguration = new HubConfiguration();
hubConfiguration.EnableDetailedErrors = true;
App.MapSignalR(hubConfiguration);
```

To handle errors that SignalR raises, you can add a handler for the `Error` event on the connection object.

```
hubConnection.Error += ex => Console.WriteLine("SignalR error: {0}", ex.Message);
```

To handle errors from method invocations, wrap the code in a try-catch block.

```
try
{
    IEnumerable<Stock> stocks = await stockTickerHub.Invoke<IEnumerable<Stock>>("GetAllStocks");
    foreach (Stock stock in stocks)
    {
        Console.WriteLine("Symbol: {0} price: {1}", stock.Symbol, stock.Price);
    }
}
catch (Exception ex)
{
    Console.WriteLine("Error invoking GetAllStocks: {0}", ex.Message);
}
```

How to enable client-side logging

To enable client-side logging, set the `TraceLevel` and `TraceWriter` properties on the connection object.

```
using (var hubConnection = new HubConnection("http://www.contoso.com/"))
{
    hubConnection.TraceLevel = TraceLevels.All;
    hubConnection.TraceWriter = Console.Out;
    IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
    stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new
price {1}", stock.Symbol, stock.Price));
    await hubConnection.Start();
}
```

WPF, Silverlight, and console application code samples for client methods that the server can call

The code samples shown earlier for defining client methods that the server can call apply to WinRT clients. The following samples show the equivalent code for WPF, Silverlight, and console application clients.

Methods without parameters

WPF client code for method called from server without parameters

```
stockTickerHub.On<Stock>("notify", () =>
    Dispatcher.InvokeAsync(() =>
        {
            SignalRTextBlock.Text += string.Format("Notified!");
        })
);
```

Silverlight client code for method called from server without parameters

```
stockTickerHub.On<Stock>("notify", () =>
    // Context is a reference to SynchronizationContext.Current
    Context.Post(delegate
    {
        textBox.Text += "Notified!";
    }, null)
);
```

Console application client code for method called from server without parameters

```
stockTickerHubProxyProxy.On("Notify", () => Console.WriteLine("Notified!"));
```

Methods with parameters, specifying the parameter types

WPF client code for a method called from server with a parameter

```
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
    Dispatcher.InvokeAsync(() =>
        {
            textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
        })
);
```

Silverlight client code for a method called from server with a parameter

```
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
    // Context is a reference to SynchronizationContext.Current
    Context.Post(delegate
    {
        textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
    }, null)
);
```

Console application client code for a method called from server with a parameter

```
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
    Console.WriteLine("Symbol {0} Price {1}", stock.Symbol, stock.Price));
```

Methods with parameters, specifying dynamic objects for the parameters

WPF client code for a method called from server with a parameter, using a dynamic object for the parameter

```
stockTickerHubProxy.On("UpdateStockPrice", stock =>
    Dispatcher.InvokeAsync(() =>
        {
            textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
        })
);
```

Silverlight client code for a method called from server with a parameter, using a dynamic object for the parameter

```
stockTickerHubProxy.On("UpdateStockPrice", stock =>
    // Context is a reference to SynchronizationContext.Current
    Context.Post(delegate
    {
        textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
    }, null)
);
```

Console application client code for a method called from server with a parameter, using a dynamic object for the parameter

```
stockTickerHubProxy.On("UpdateStockPrice", stock =>
    Console.WriteLine("Symbol {0} Price {1}", stock.Symbol, stock.Price));
```

Understanding and Handling Connection Lifetime Events in SignalR

2/20/2020 • 17 minutes to read • Edit Online

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This article provides an overview of the SignalR connection, reconnection, and disconnection events that you can handle, and timeout and keepalive settings that you can configure.

The article assumes you already have some knowledge of SignalR and connection lifetime events. For an introduction to SignalR, see Introduction to SignalR. For lists of connection lifetime events, see the following resources:

- How to handle connection lifetime events in the Hub class
- How to handle connection lifetime events in JavaScript clients
- How to handle connection lifetime events in .NET clients

Software versions used in this topic

- Visual Studio 2017
- .NET 4.5
- SignalR version 2

Previous versions of this topic

For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Overview

This article contains the following sections:

- Connection lifetime terminology and scenarios
  - SignalR connections, transport connections, and physical connections
  - Transport disconnection scenarios
  - Client disconnection scenarios
  - Server disconnection scenarios
- Timeout and keepalive settings
  - ConnectionTimeout

- DisconnectTimeout
- KeepAlive
- How to change timeout and keepalive settings

- How to notify the user about disconnections
- How to continuously reconnect
- How to disconnect a client in server code
- Detecting the reason for a disconnection

Links to API Reference topics are to the .NET 4.5 version of the API. If you're using .NET 4, see the .NET 4 version of the API topics.

Connection lifetime terminology and scenarios

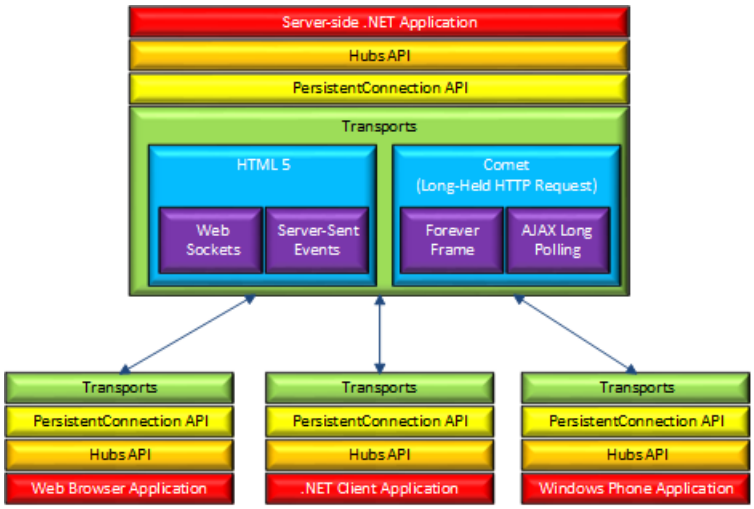
The `OnReconnected` event handler in a SignalR Hub can execute directly after `OnConnected` but not after `OnDisconnected` for a given client. The reason you can have a reconnection without a disconnection is that there are several ways in which the word "connection" is used in SignalR.

SignalR connections, transport connections, and physical connections

This article will differentiate between *SignalR connections*, *transport connections*, and *physical connections*:

- **SignalR connection** refers to a logical relationship between a client and a server URL, maintained by the SignalR API and uniquely identified by a connection ID. The data about this relationship is maintained by SignalR and is used to establish a transport connection. The relationship ends and SignalR disposes of the data when the client calls the `Stop` method or a timeout limit is reached while SignalR is attempting to re-establish a lost transport connection.
- **Transport connection** refers to a logical relationship between a client and a server, maintained by one of the four transport APIs: WebSockets, server-sent events, forever frame, or long polling. SignalR uses the transport API to create a transport connection, and the transport API depends on the existence of a physical network connection to create the transport connection. The transport connection ends when SignalR terminates it or when the transport API detects that the physical connection is broken.
- **Physical connection** refers to the physical network links -- wires, wireless signals, routers, etc. -- that facilitate communication between a client computer and a server computer. The physical connection must be present in order to establish a transport connection, and a transport connection must be established in order to establish a SignalR connection. However, breaking the physical connection doesn't always immediately end the transport connection or the SignalR connection, as will be explained later in this topic.

In the following diagram, the SignalR connection is represented by the Hubs API and PersistentConnection API SignalR layer, the transport connection is represented by the Transports layer, and the physical connection is represented by the lines between the server and the clients.



When you call the `Start` method in a SignalR client, you are providing SignalR client code with all the information it needs in order to establish a physical connection to a server. SignalR client code uses this information to make an HTTP request and establish a physical connection that uses one of the four transport methods. If the transport connection fails or the server fails, the SignalR connection doesn't go away immediately because the client still has the information it needs to automatically re-establish a new transport connection to the same SignalR URL. In this scenario, no intervention from the user application is involved, and when the SignalR client code establishes a new transport connection, it does not start a new SignalR connection. The continuity of the SignalR connection is reflected in the fact that the connection ID, which is created when you call the `Start` method, does not change.

The `OnReconnected` event handler on the Hub executes when a transport connection is automatically re-established after having been lost. The `OnDisconnected` event handler executes at the end of a SignalR connection. A SignalR connection can end in any of the following ways:

- If the client calls the `Stop` method, a stop message is sent to the server, and both client and server end the SignalR connection immediately.
- After connectivity between client and server is lost, the client tries to reconnect and the server waits for the client to reconnect. If the attempts to reconnect are unsuccessful and the disconnect timeout period ends, both client and server end the SignalR connection. The client stops trying to reconnect, and the server disposes of its representation of the SignalR connection.
- If the client stops running without having a chance to call the `Stop` method, the server waits for the client to reconnect, and then ends the SignalR connection after the disconnect timeout period.
- If the server stops running, the client tries to reconnect (re-create the transport connection), and then ends the SignalR connection after the disconnect timeout period.

When there are no connection problems, and the user application ends the SignalR connection by calling the `Stop` method, the SignalR connection and the transport connection begin and end at about the same time. The following sections describe in more detail the other scenarios.

Transport disconnection scenarios

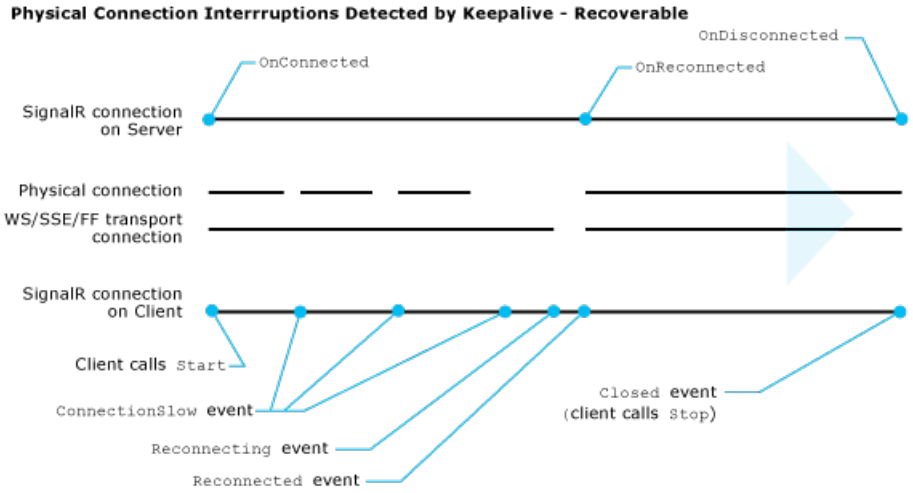
Physical connections might be slow or there might be interruptions in connectivity. Depending on factors such as the length of the interruption, the transport connection might be dropped. SignalR then tries to re-establish the transport connection. Sometimes the transport connection API detects the interruption and drops the transport connection, and SignalR finds out immediately that the connection is lost. In other scenarios, neither the transport connection API nor SignalR becomes aware immediately that connectivity has been lost. For all transports except long polling, the SignalR client uses a function called *keepalive* to check for loss of connectivity that the transport API is unable to detect. For information about long polling connections, see Timeout and keepalive settings later in this topic.

When a connection is inactive, periodically the server sends a keepalive packet to the client. As of the date this

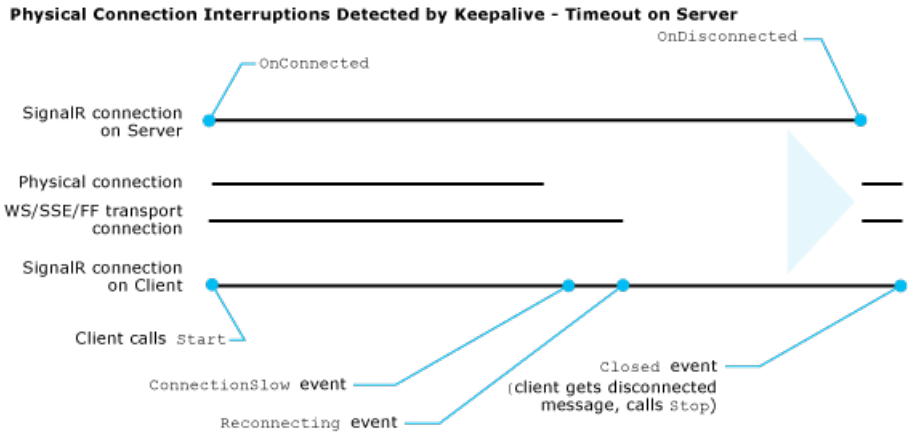
article is being written, the default frequency is every 10 seconds. By listening for these packets, clients can tell if there is a connection problem. If a keepalive packet is not received when expected, after a short time the client assumes that there are connection problems such as slowness or interruptions. If the keepalive is still not received after a longer time, the client assumes that the connection has been dropped, and it begins trying to reconnect.

The following diagram illustrates the client and server events that are raised in a typical scenario when there are problems with the physical connection that aren't immediately recognized by the transport API. The diagram applies to the following circumstances:

- The transport is WebSockets, forever frame, or server-sent events.
- There are varying periods of interruption in the physical network connection.
- The transport API does not become aware of the interruptions, so SignalR relies on the keepalive functionality to detect them.



If the client goes into reconnecting mode but can't establish a transport connection within the disconnect timeout limit, the server terminates the SignalR connection. When that happens, the server executes the Hub's `OnDisconnected` method and queues up a disconnect message to send to the client in case the client manages to connect later. If the client then does reconnect, it receives the disconnect command and calls the `Stop` method. In this scenario, `OnReconnected` is not executed when the client reconnects, and `OnDisconnected` is not executed when the client calls `Stop`. The following diagram illustrates this scenario.



The SignalR connection lifetime events that may be raised on the client are the following:

- `ConnectionSlow` client event.

  Raised when a preset proportion of the keepalive timeout period has passed since the last message or keepalive ping was received. The default keepalive timeout warning period is 2/3 of the keepalive timeout. The keepalive timeout is 20 seconds, so the warning occurs at about 13 seconds.

By default, the server sends keepalive pings every 10 seconds, and the client checks for keepalive pings about every 2 seconds (one third of the difference between the keepalive timeout value and the keepalive timeout warning value).

If the transport API becomes aware of a disconnection, SignalR might be informed of the disconnection before the keepalive timeout warning period passes. In that case, the `ConnectionSlow` event would not be raised, and SignalR would go directly to the `Reconnecting` event.

- `Reconnecting` client event.

  Raised when (a) the transport API detects that the connection is lost, or (b) the keepalive timeout period has passed since the last message or keepalive ping was received. The SignalR client code begins trying to reconnect. You can handle this event if you want your application to take some action when a transport connection is lost. The default keepalive timeout period is currently 20 seconds.

  If your client code tries to call a Hub method while SignalR is in reconnecting mode, SignalR will try to send the command. Most of the time, such attempts will fail, but in some circumstances they might succeed. For the server-sent events, forever frame, and long polling transports, SignalR uses two communication channels, one that the client uses to send messages and one that it uses to receive messages. The channel used for receiving is the permanently open one, and that's the one that is closed when the physical connection is interrupted. The channel used for sending remains available, so if physical connectivity is restored, a method call from client to server might be successful before the receive channel is re-established. The return value would not be received until SignalR re-opens the channel used for receiving.

- `Reconnected` client event.

  Raised when the transport connection is reestablished. The `OnReconnected` event handler in the Hub executes.

- `Closed` client event (`disconnected` event in JavaScript).

  Raised when the disconnect timeout period expires while the SignalR client code is trying to reconnect after losing the transport connection. The default disconnect timeout is 30 seconds. (This event is also raised when the connection ends because the `Stop` method is called.)

Transport connection interruptions that are not detected by the transport API and don't delay the reception of keepalive pings from the server for longer than the keepalive timeout warning period might not cause any connection lifetime events to be raised.

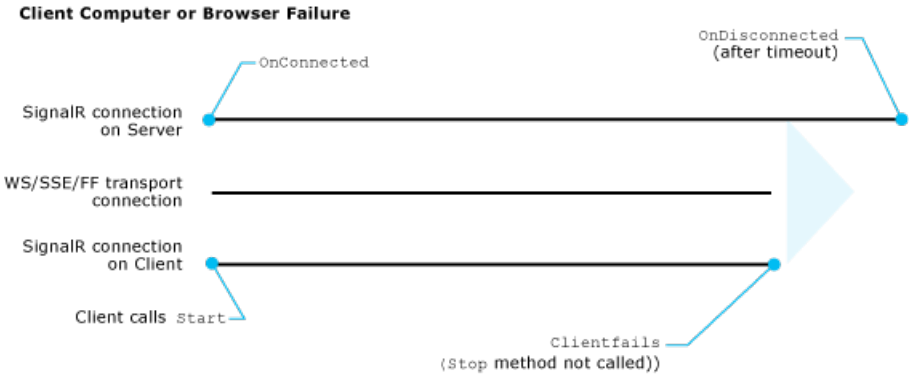
Some network environments deliberately close idle connections, and another function of the keepalive packets is to help prevent this by letting these networks know that a SignalR connection is in use. In extreme cases the default frequency of keepalive pings might not be enough to prevent closed connections. In that case you can configure keepalive pings to be sent more often. For more information, see Timeout and keepalive settings later in this topic.

> **NOTE**
>
> **Important**: The sequence of events described here is not guaranteed. SignalR makes every attempt to raise connection lifetime events in a predictable manner according to this scheme, but there are many variations of network events and many ways in which underlying communications frameworks such as transport APIs handle them. For example, the `Reconnected` event might not be raised when the client reconnects, or the `OnConnected` handler on the server might run when the attempt to establish a connection is unsuccessful. This topic describes only the effects that would normally be produced by certain typical circumstances.

Client disconnection scenarios

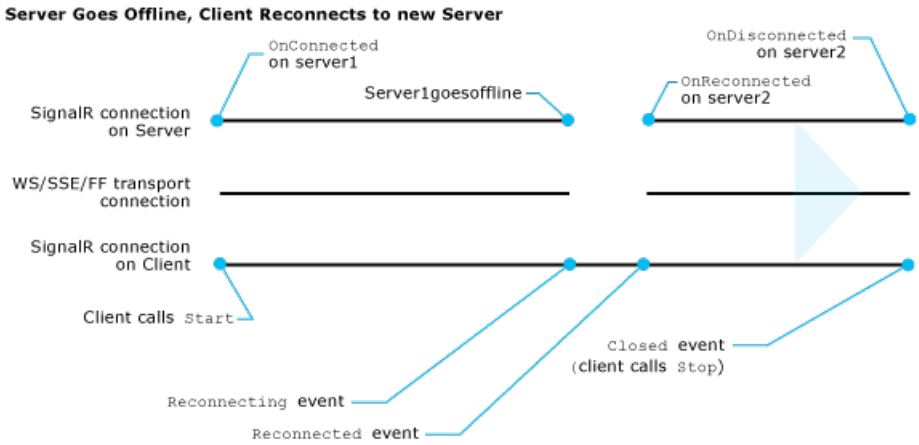
In a browser client, the SignalR client code that maintains a SignalR connection runs in the JavaScript context of a web page. That's why the SignalR connection has to end when you navigate from one page to another, and that's why you have multiple connections with multiple connection IDs if you connect from multiple browser windows or tabs. When the user closes a browser window or tab, or navigates to a new page or refreshes the page, the SignalR connection immediately ends because SignalR client code handles that browser event for you and calls the `Stop` method. In these scenarios, or in any client platform when your application calls the `Stop` method, the `OnDisconnected` event handler executes immediately on the server and the client raises the `Closed` event (the event is named `disconnected` in JavaScript).

If a client application or the computer that it's running on crashes or goes to sleep (for example, when the user closes the laptop), the server is not informed about what happened. As far as the server knows, the loss of the client might be due to connectivity interruption and the client might be trying to reconnect. Therefore, in these scenarios the server waits to give the client a chance to reconnect, and `OnDisconnected` does not execute until the disconnect timeout period expires (about 30 seconds by default). The following diagram illustrates this scenario.



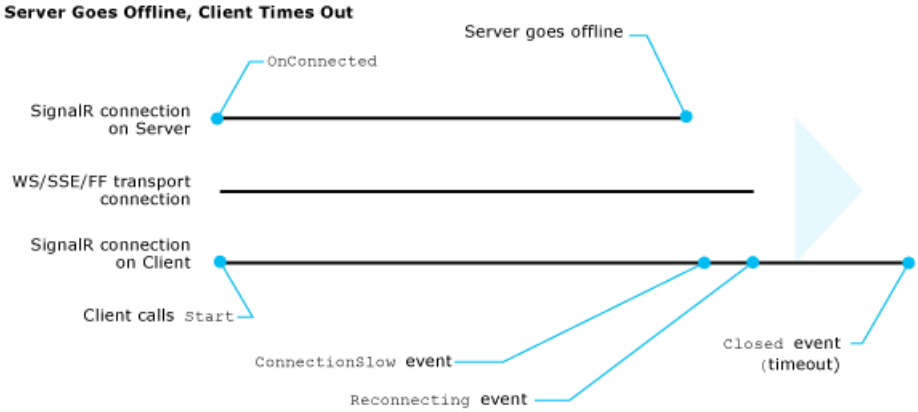
Server disconnection scenarios

When a server goes offline -- it reboots, fails, the app domain recycles, etc. -- the result might be similar to a lost connection, or the transport API and SignalR might know immediately that the server is gone, and SignalR might begin trying to reconnect without raising the `ConnectionSlow` event. If the client goes into reconnecting mode, and if the server recovers or restarts or a new server is brought online before the disconnect timeout period expires, the client will reconnect to the restored or new server. In that case, the SignalR connection continues on the client and the `Reconnected` event is raised. On the first server, `OnDisconnected` is never executed, and on the new server, `OnReconnected` is executed although `OnConnected` was never executed for that client on that server before. (The effect is the same if the client reconnects to the same server after a reboot or app domain recycle, because when the server restarts it has no memory of prior connection activity.) The following diagram assumes that the transport API becomes aware of the lost connection immediately, so the `ConnectionSlow` event is not raised.



If a server does not become available within the disconnect timeout period, the SignalR connection ends. In this scenario, the `Closed` event (`disconnected` in JavaScript clients) is raised on the client but `OnDisconnected` is never called on the server. The following diagram assumes that the transport API does not become aware of the

lost connection, so it is detected by SignalR keepalive functionality and the `ConnectionSlow` event is raised.



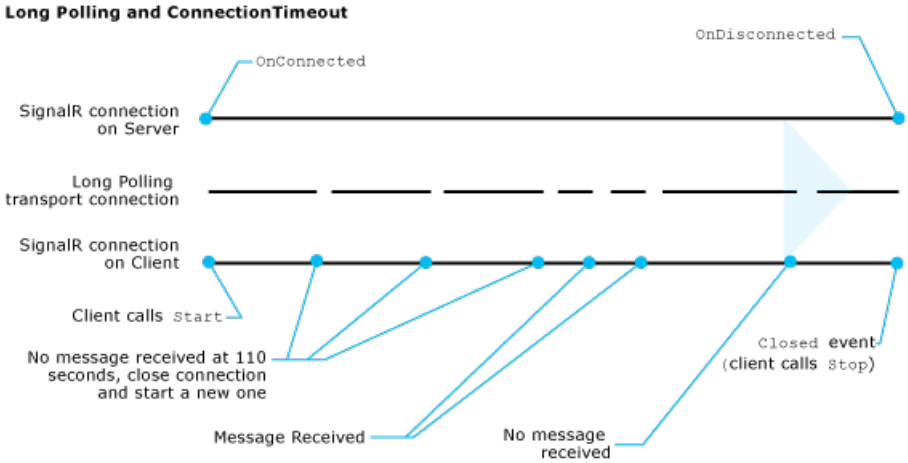
Timeout and keepalive settings

The default `ConnectionTimeout`, `DisconnectTimeout`, and `KeepAlive` values are appropriate for most scenarios but can be changed if your environment has special needs. For example, if your network environment closes connections that are idle for 5 seconds, you might have to decrease the keepalive value.

ConnectionTimeout

This setting represents the amount of time to leave a transport connection open and waiting for a response before closing it and opening a new connection. The default value is 110 seconds.

This setting applies only when keepalive functionality is disabled, which normally applies only to the long polling transport. The following diagram illustrates the effect of this setting on a long polling transport connection.



DisconnectTimeout

This setting represents the amount of time to wait after a transport connection is lost before raising the `Disconnected` event. The default value is 30 seconds. When you set `DisconnectTimeout`, `KeepAlive` is automatically set to 1/3 of the `DisconnectTimeout` value.

KeepAlive

This setting represents the amount of time to wait before sending a keepalive packet over an idle connection. The default value is 10 seconds. This value must not be more than 1/3 of the `DisconnectTimeout` value.

If you want to set both `DisconnectTimeout` and `KeepAlive`, set `KeepAlive` after `DisconnectTimeout`. Otherwise your `KeepAlive` setting will be overwritten when `DisconnectTimeout` automatically sets `KeepAlive` to 1/3 of the timeout value.

If you want to disable keepalive functionality, set `KeepAlive` to null. Keepalive functionality is automatically disabled for the long polling transport.

How to change timeout and keepalive settings

To change the default values for these settings, set them in `Application_Start` in your *Global.asax* file, as shown in the following example. The values shown in the sample code are the same as the default values.

```
protected void Application_Start(object sender, EventArgs e)
{
    // Make long polling connections wait a maximum of 110 seconds for a
    // response. When that time expires, trigger a timeout command and
    // make the client reconnect.
    GlobalHost.Configuration.ConnectionTimeout = TimeSpan.FromSeconds(110);

    // Wait a maximum of 30 seconds after a transport connection is lost
    // before raising the Disconnected event to terminate the SignalR connection.
    GlobalHost.Configuration.DisconnectTimeout = TimeSpan.FromSeconds(30);

    // For transports other than long polling, send a keepalive packet every
    // 10 seconds.
    // This value must be no more than 1/3 of the DisconnectTimeout value.
    GlobalHost.Configuration.KeepAlive = TimeSpan.FromSeconds(10);

    RouteTable.Routes.MapHubs();
}
```

How to notify the user about disconnections

In some applications you might want to display a message to the user when there are connectivity problems. You have several options for how and when to do this. The following code samples are for a JavaScript client using the generated proxy.

- Handle the `connectionSlow` event to display a message as soon as SignalR is aware of connection problems, before it goes into reconnecting mode.

  ```
  $.connection.hub.connectionSlow(function() {
      notifyUserOfConnectionProblem(); // Your function to notify user.
  });
  ```

- Handle the `reconnecting` event to display a message when SignalR is aware of a disconnection and is going into reconnecting mode.

  ```
  $.connection.hub.reconnecting(function() {
      notifyUserOfTryingToReconnect(); // Your function to notify user.
  });
  ```

- Handle the `disconnected` event to display a message when an attempt to reconnect has timed out. In this scenario, the only way to re-establish a connection with the server again is to restart the SignalR connection by calling the `Start` method, which will create a new connection ID. The following code sample uses a flag to make sure that you issue the notification only after a reconnecting timeout, not after a normal end to the SignalR connection caused by calling the `Stop` method.

```
var tryingToReconnect = false;

$.connection.hub.reconnecting(function() {
    tryingToReconnect = true;
});

$.connection.hub.reconnected(function() {
    tryingToReconnect = false;
});

$.connection.hub.disconnected(function() {
    if(tryingToReconnect) {
        notifyUserOfDisconnect(); // Your function to notify user.
    }
});
```

How to continuously reconnect

In some applications you might want to automatically re-establish a connection after it has been lost and the attempt to reconnect has timed out. To do that, you can call the `Start` method from your `Closed` event handler (`disconnected` event handler on JavaScript clients). You might want to wait a period of time before calling `Start` in order to avoid doing this too frequently when the server or the physical connection are unavailable. The following code sample is for a JavaScript client using the generated proxy.

```
$.connection.hub.disconnected(function() {
   setTimeout(function() {
       $.connection.hub.start();
   }, 5000); // Restart connection after 5 seconds.
});
```

A potential problem to be aware of in mobile clients is that continuous reconnection attempts when the server or physical connection isn't available could cause unnecessary battery drain.

How to disconnect a client in server code

SignalR version 2 does not have a built-in server API for disconnecting clients. There are plans for adding this functionality in the future. In the current SignalR release, the simplest way to disconnect a client from the server is to implement a disconnect method on the client and call that method from the server. The following code sample shows a disconnect method for a JavaScript client using the generated proxy.

```
var myHubProxy = $.connection.myHub
myHubProxy.client.stopClient = function() {
    $.connection.hub.stop();
};
```

WARNING

Security - Neither this method for disconnecting clients nor the proposed built-in API will address the scenario of hacked clients that are running malicious code, since the clients could reconnect or the hacked code might remove the `stopClient` method or change what it does. The appropriate place to implement stateful denial-of-service (DOS) protection is not in the framework or the server layer, but rather in front-end infrastructure.

Detecting the reason for a disconnection

SignalR 2.1 adds an overload to the server `OnDisconnect` event that indicates if the client deliberately disconnected rather than timing out. The `StopCalled` parameter is true if the client explicitly closed the connection. In JavaScript, if a server error led the client to disconnect, the error information will be passed to the client as `$.connection.hub.lastError`.

C# server code: `stopCalled` parameter

```
public override System.Threading.Tasks.Task OnDisconnected(bool stopCalled)
{
    if (stopCalled)
    {
        Console.WriteLine(String.Format("Client {0} explicitly closed the connection.",
Context.ConnectionId));
    }
    else
    {
        Console.WriteLine(String.Format("Client {0} timed out .", Context.ConnectionId));
    }

    return base.OnDisconnected(stopCalled);
}
```

JavaScript client code: accessing `lastError` in the `disconnect` event.

```
$.connection.hub.disconnected(function () {
    if ($.connection.hub.lastError)
        { alert("Disconnected. Reason: " +  $.connection.hub.lastError.message); }
});
```

Working with Groups in SignalR

2/20/2020 • 8 minutes to read • Edit Online

by Patrick Fletcher, Tom FitzMacken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This topic describes how to add users to groups and persist group membership information.

Software versions used in this topic

- Visual Studio 2013
- .NET 4.5
- SignalR version 2

Previous versions of this topic

For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Overview

Groups in SignalR provide a method for broadcasting messages to specified subsets of connected clients. A group can have any number of clients, and a client can be a member of any number of groups. You don't have to explicitly create groups. In effect, a group is automatically created the first time you specify its name in a call to Groups.Add, and it is deleted when you remove the last connection from membership in it. For an introduction to using groups, see How to manage group membership from the Hub class in the Hubs API - Server Guide.

There is no API for getting a group membership list or a list of groups. SignalR sends messages to clients and groups based on a pub/sub model, and the server does not maintain lists of groups or group memberships. This helps maximize scalability, because whenever you add a node to a web farm, any state that SignalR maintains has to be propagated to the new node.

When you add a user to a group using the `Groups.Add` method, the user receives messages directed to that group for the duration of the current connection, but the user's membership in that group is not persisted beyond the current connection. If you want to permanently retain information about groups and group membership, you must store that data in a repository such as a database or Azure table storage. Then, each time a user connects to your application, you retrieve from the repository which groups the user belongs to, and manually add that user to those groups.

When reconnecting after a temporary disruption, the user automatically re-joins the previously-assigned groups. Automatically rejoining a group only applies when reconnecting, not when establishing a new connection. A digitally-signed token is passed from the client that contains the list of previously-assigned groups. If you want to

verify whether the user belongs to the requested groups, you can override the default behavior.

This topic includes the following sections:

- Adding and removing users
- Calling members of a group
- Storing group membership in a database
- Storing group membership in Azure table storage
- Verifying group membership when reconnecting

Adding and removing users

To add or remove users from a group, you call the Add or Remove methods, and pass in the user's connection id and group's name as parameters. You do not need to manually remove a user from a group when the connection ends.

The following example shows the `Groups.Add` and `Groups.Remove` methods used in Hub methods.

```
public class ContosoChatHub : Hub
{
    public Task JoinRoom(string roomName)
    {
        return Groups.Add(Context.ConnectionId, roomName);
    }

    public Task LeaveRoom(string roomName)
    {
        return Groups.Remove(Context.ConnectionId, roomName);
    }
}
```

The `Groups.Add` and `Groups.Remove` methods execute asynchronously.

If you want to add a client to a group and immediately send a message to the client by using the group, you have to make sure that the Groups.Add method finishes first. The following code examples show how to do that.

```
public async Task JoinRoom(string roomName)
{
    await Groups.Add(Context.ConnectionId, roomName);
    Clients.Group(roomName).addChatMessage(Context.User.Identity.Name + " joined.");
}
```

In general, you should not include `await` when calling the `Groups.Remove` method because the connection id that you are trying to remove might no longer be available. In that case, `TaskCanceledException` is thrown after the request times out. If your application must ensure that the user has been removed from the group before sending a message to the group, you can add `await` before `Groups.Remove`, and then catch the `TaskCanceledException` exception that might be thrown.

Calling members of a group

You can send messages to all of the members of a group or only specified members of the group, as shown in the following examples.

- **All** connected clients in a specified group.

  ```
  Clients.Group(groupName).addChatMessage(name, message);
  ```

- All connected clients in a specified group **except the specified clients**, identified by connection ID.

```
Clients.Group(groupName, connectionId1, connectionId2).addChatMessage(name, message);
```

- All connected clients in a specified group **except the calling client**.

```
Clients.OthersInGroup(groupName).addChatMessage(name, message);
```

Storing group membership in a database

The following examples show how to retain group and user information in a database. You can use any data access technology; however, the example below shows how to define models using Entity Framework. These entity models correspond to database tables and fields. Your data structure could vary considerably depending on the requirements of your application. This example includes a class named `ConversationRoom` which would be unique to an application that enables users to join conversations about different subjects, such as sports or gardening. This example also includes a class for the connections. The connection class is not absolutely required for tracking group membership but is frequently part of robust solution to tracking users.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity;

namespace GroupsExample
{
    public class UserContext : DbContext
    {
        public DbSet<User> Users { get; set; }
        public DbSet<Connection> Connections { get; set; }
        public DbSet<ConversationRoom> Rooms { get; set; }
    }

    public class User
    {
        [Key]
        public string UserName { get; set; }
        public ICollection<Connection> Connections { get; set; }
        public virtual ICollection<ConversationRoom> Rooms { get; set; }
    }

    public class Connection
    {
        public string ConnectionID { get; set; }
        public string UserAgent { get; set; }
        public bool Connected { get; set; }
    }

    public class ConversationRoom
    {
        [Key]
        public string RoomName { get; set; }
        public virtual ICollection<User> Users { get; set; }
    }
}
```

Then, in the hub, you can retrieve the group and user information from the database and manually add the user to the appropriate groups. The example does not include code for tracking the user connections. In this example, the `await` keyword is not applied before `Groups.Add` because a message is not immediately sent to members of the group. If you want to send a message to all members of the group immediately after adding the new member,

you would want to apply the `await` keyword to make sure the asynchronous operation has completed.

```
using Microsoft.AspNet.SignalR;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Threading.Tasks;

namespace GroupsExample
{
    [Authorize]
    public class ChatHub : Hub
    {
        public override Task OnConnected()
        {
            using (var db = new UserContext())
            {
                // Retrieve user.
                var user = db.Users
                    .Include(u => u.Rooms)
                    .SingleOrDefault(u => u.UserName == Context.User.Identity.Name);

                // If user does not exist in database, must add.
                if (user == null)
                {
                    user = new User()
                    {
                        UserName = Context.User.Identity.Name
                    };
                    db.Users.Add(user);
                    db.SaveChanges();
                }
                else
                {
                    // Add to each assigned group.
                    foreach (var item in user.Rooms)
                    {
                        Groups.Add(Context.ConnectionId, item.RoomName);
                    }
                }
            }
            return base.OnConnected();
        }

        public void AddToRoom(string roomName)
        {
            using (var db = new UserContext())
            {
                // Retrieve room.
                var room = db.Rooms.Find(roomName);

                if (room != null)
                {
                    var user = new User() { UserName = Context.User.Identity.Name};
                    db.Users.Attach(user);

                    room.Users.Add(user);
                    db.SaveChanges();
                    Groups.Add(Context.ConnectionId, roomName);
                }
            }
        }

        public void RemoveFromRoom(string roomName)
        {
            using (var db = new UserContext())
            {
```

```
                // Retrieve room.
                var room = db.Rooms.Find(roomName);
                if (room != null)
                {
                    var user = new User() { UserName = Context.User.Identity.Name };
                    db.Users.Attach(user);

                    room.Users.Remove(user);
                    db.SaveChanges();

                    Groups.Remove(Context.ConnectionId, roomName);
                }
            }
        }
    }
}
```

Storing group membership in Azure table storage

Using Azure table storage to store group and user information is similar to using a database. The following example shows a table entity that stores the user name and group name.

```
using Microsoft.WindowsAzure.Storage.Table;
using System;

namespace GroupsExample
{
    public class UserGroupEntity : TableEntity
    {
        public UserGroupEntity() { }

        public UserGroupEntity(string userName, string groupName)
        {
            this.PartitionKey = userName;
            this.RowKey = groupName;
        }
    }
}
```

In the hub, you retrieve the assigned groups when the user connects.

```
using Microsoft.AspNet.SignalR;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.WindowsAzure.Storage.Table;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure;

namespace GroupsExample
{
    [Authorize]
    public class ChatHub : Hub
    {
        public override Task OnConnected()
        {
            string userName = Context.User.Identity.Name;

            var table = GetRoomTable();
            table.CreateIfNotExists();
            var query = new TableQuery<UserGroupEntity>()
                .Where(TableQuery.GenerateFilterCondition(
                "PartitionKey", QueryComparisons.Equal, userName));
```

```
            foreach (var entity in table.ExecuteQuery(query))
            {
                Groups.Add(Context.ConnectionId, entity.RowKey);
            }

            return base.OnConnected();
        }

        public Task AddToRoom(string roomName)
        {
            string userName = Context.User.Identity.Name;

            var table = GetRoomTable();

            var insertOperation = TableOperation.InsertOrReplace(
                new UserGroupEntity(userName, roomName));
            table.Execute(insertOperation);

            return Groups.Add(Context.ConnectionId, roomName);
        }

        public Task RemoveFromRoom(string roomName)
        {
            string userName = Context.User.Identity.Name;

            var table = GetRoomTable();

            var retrieveOperation = TableOperation.Retrieve<UserGroupEntity>(
                userName, roomName);
            var retrievedResult = table.Execute(retrieveOperation);

            var deleteEntity = (UserGroupEntity)retrievedResult.Result;

            if (deleteEntity != null)
            {
                var deleteOperation = TableOperation.Delete(deleteEntity);
                table.Execute(deleteOperation);
            }

            return Groups.Remove(Context.ConnectionId, roomName);
        }

       private CloudTable GetRoomTable()
        {
            var storageAccount =
                CloudStorageAccount.Parse(
                CloudConfigurationManager.GetSetting("StorageConnectionString"));
            var tableClient = storageAccount.CreateCloudTableClient();
            return tableClient.GetTableReference("room");
        }
    }
}
```

Verifying group membership when reconnecting

By default, SignalR automatically re-assigns a user to the appropriate groups when reconnecting from a temporary disruption, such as when a connection is dropped and re-established before the connection times out. The user's group information is passed in a token when reconnecting, and that token is verified on the server. For information about the verification process for rejoining users to groups, see Rejoining groups when reconnecting.

In general, you should use the default behavior of automatically rejoining groups on reconnect. SignalR groups are not intended as a security mechanism for restricting access to sensitive data. However, if your application must double-check a user's group membership when reconnecting, you can override the default behavior. Changing the default behavior can add a burden to your database because a user's group membership must be

retrieved for each reconnection rather than just when the user connects.

If you must verify group membership on reconnect, create a new hub pipeline module that returns a list of assigned groups, as shown below.

```
using Microsoft.AspNet.SignalR;
using Microsoft.AspNet.SignalR.Hubs;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;

namespace GroupsExample
{
    public class RejoingGroupPipelineModule : HubPipelineModule
    {
        public override Func<HubDescriptor, IRequest, IList<string>, IList<string>>
            BuildRejoiningGroups(Func<HubDescriptor, IRequest, IList<string>, IList<string>>
            rejoiningGroups)
        {
            rejoiningGroups = (hb, r, l) =>
            {
                List<string> assignedRooms = new List<string>();
                using (var db = new UserContext())
                {
                    var user = db.Users.Include(u => u.Rooms)
                        .Single(u => u.UserName == r.User.Identity.Name);
                    foreach (var item in user.Rooms)
                    {
                        assignedRooms.Add(item.RoomName);
                    }
                }
                return assignedRooms;
            };

            return rejoiningGroups;
        }
    }
}
```

Then, add that module to the hub pipeline, as highlighted below.

```
public partial class Startup {
    public void Configuration(IAppBuilder app) {
        app.MapSignalR();
        GlobalHost.HubPipeline.AddModule(new RejoingGroupPipelineModule());
    }
}
```

Mapping SignalR Users to Connections

2/20/2020 • 8 minutes to read • Edit Online

by Tom FitzMacken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This topic shows how to retain information about users and their connections.

Patrick Fletcher helped write this topic.

Software versions used in this topic

- Visual Studio 2013
- .NET 4.5
- SignalR version 2

Previous versions of this topic

For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Introduction

Each client connecting to a hub passes a unique connection id. You can retrieve this value in the `Context.ConnectionId` property of the hub context. If your application needs to map a user to the connection id and persist that mapping, you can use one of the following:

- The User ID Provider (SignalR 2)
- In-memory storage, such as a dictionary
- SignalR group for each user
- Permanent, external storage, such as a database table or Azure table storage

Each of these implementations is shown in this topic. You use the `OnConnected`, `OnDisconnected`, and `OnReconnected` methods of the `Hub` class to track the user connection status.

The best approach for your application depends on:

- The number of web servers hosting your application.
- Whether you need to get a list of the currently connected users.
- Whether you need to persist group and user information when the application or server restarts.
- Whether the latency of calling an external server is an issue.

The following table shows which approach works for these considerations.

	MORE THAN ONE SERVER	GET LIST OF CURRENTLY CONNECTED USERS	PERSIST INFORMATION AFTER RESTARTS	OPTIMAL PERFORMANCE
UserID Provider	✓			✓
In-memory		✓		✓
Single-user groups	✓			✓
Permanent, external	✓	✓	✓	

IUserID provider

This feature allows users to specify what the userId is based on an IRequest via a new interface IUserIdProvider.

The IUserIdProvider

```
public interface IUserIdProvider
{
    string GetUserId(IRequest request);
}
```

By default, there will be an implementation that uses the user's `IPrincipal.Identity.Name` as the user name. To change this, register your implementation of `IUserIdProvider` with the global host when your application starts:

```
GlobalHost.DependencyResolver.Register(typeof(IUserIdProvider), () => new MyIdProvider());
```

From within a hub, you'll be able to send messages to these users via the following API:

Sending a message to a specific user

```
public class MyHub : Hub
{
    public void Send(string userId, string message)
    {
        Clients.User(userId).send(message);
    }
}
```

In-memory storage

The following examples show how to retain connection and user information in a dictionary that is stored in memory. The dictionary uses a `HashSet` to store the connection id. At any time a user could have more than one connection to the SignalR application. For example, a user who is connected through multiple devices or more than one browser tab would have more than one connection id.

If the application shuts down, all of the information is lost, but it will be re-populated as the users re-establish their connections. In-memory storage does not work if your environment includes more than one web server because each server would have a separate collection of connections.

The first example shows a class that manages the mapping of users to connections. The key for the HashSet will be the user's name.

```
using System.Collections.Generic;
using System.Linq;

namespace BasicChat
{
    public class ConnectionMapping<T>
    {
        private readonly Dictionary<T, HashSet<string>> _connections =
            new Dictionary<T, HashSet<string>>();

        public int Count
        {
            get
            {
                return _connections.Count;
            }
        }

        public void Add(T key, string connectionId)
        {
            lock (_connections)
            {
                HashSet<string> connections;
                if (!_connections.TryGetValue(key, out connections))
                {
                    connections = new HashSet<string>();
                    _connections.Add(key, connections);
                }

                lock (connections)
                {
                    connections.Add(connectionId);
                }
            }
        }

        public IEnumerable<string> GetConnections(T key)
        {
            HashSet<string> connections;
            if (_connections.TryGetValue(key, out connections))
            {
                return connections;
            }

            return Enumerable.Empty<string>();
        }

        public void Remove(T key, string connectionId)
        {
            lock (_connections)
            {
                HashSet<string> connections;
                if (!_connections.TryGetValue(key, out connections))
                {
                    return;
                }

                lock (connections)
                {
                    connections.Remove(connectionId);

                    if (connections.Count == 0)
                    {
                        _connections.Remove(key);
                    }
                }
            }
        }
```

```
    }
}
```

The next example shows how to use the connection mapping class from a hub. The instance of the class is stored in a variable name `_connections`.

```
using System.Threading.Tasks;
using Microsoft.AspNet.SignalR;

namespace BasicChat
{
    [Authorize]
    public class ChatHub : Hub
    {
        private readonly static ConnectionMapping<string> _connections =
            new ConnectionMapping<string>();

        public void SendChatMessage(string who, string message)
        {
            string name = Context.User.Identity.Name;

            foreach (var connectionId in _connections.GetConnections(who))
            {
                Clients.Client(connectionId).addChatMessage(name + ": " + message);
            }
        }

        public override Task OnConnected()
        {
            string name = Context.User.Identity.Name;

            _connections.Add(name, Context.ConnectionId);

            return base.OnConnected();
        }

        public override Task OnDisconnected(bool stopCalled)
        {
            string name = Context.User.Identity.Name;

            _connections.Remove(name, Context.ConnectionId);

            return base.OnDisconnected(stopCalled);
        }

        public override Task OnReconnected()
        {
            string name = Context.User.Identity.Name;

            if (!_connections.GetConnections(name).Contains(Context.ConnectionId))
            {
                _connections.Add(name, Context.ConnectionId);
            }

            return base.OnReconnected();
        }
    }
}
```

Single-user groups

You can create a group for each user, and then send a message to that group when you want to reach only that user. The name of each group is the name of the user. If a user has more than one connection, each connection id is

added to the user's group.

You should not manually remove the user from the group when the user disconnects. This action is automatically performed by the SignalR framework.

The following example shows how to implement single-user groups.

```
using Microsoft.AspNet.SignalR;
using System;
using System.Threading.Tasks;

namespace BasicChat
{
    [Authorize]
    public class ChatHub : Hub
    {
        public void SendChatMessage(string who, string message)
        {
            string name = Context.User.Identity.Name;

            Clients.Group(who).addChatMessage(name + ": " + message);
        }

        public override Task OnConnected()
        {
            string name = Context.User.Identity.Name;

            Groups.Add(Context.ConnectionId, name);

            return base.OnConnected();
        }
    }
}
```

Permanent, external storage

This topic shows how to use either a database or Azure table storage for storing connection information. This approach works when you have multiple web servers because each web server can interact with the same data repository. If your web servers stop working or the application restarts, the `OnDisconnected` method is not called. Therefore, it is possible that your data repository will have records for connection ids that are no longer valid. To clean up these orphaned records, you may wish to invalidate any connection that was created outside of a timeframe that is relevant to your application. The examples in this section include a value for tracking when the connection was created, but do not show how to clean up old records because you may want to do that as background process.

Database

The following examples show how to retain connection and user information in a database. You can use any data access technology; however, the example below shows how to define models using Entity Framework. These entity models correspond to database tables and fields. Your data structure could vary considerably depending on the requirements of your application.

The first example shows how to define a user entity that can be associated with many connection entities.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity;

namespace MapUsersSample
{
    public class UserContext : DbContext
    {
        public DbSet<User> Users { get; set; }
        public DbSet<Connection> Connections { get; set; }
    }

    public class User
    {
        [Key]
        public string UserName { get; set; }
        public ICollection<Connection> Connections { get; set; }
    }

    public class Connection
    {
        public string ConnectionID { get; set; }
        public string UserAgent { get; set; }
        public bool Connected { get; set; }
    }
}
```

Then, from the hub, you can track the state of each connection with the code shown below.

```
using System;
using System.Data.Entity;
using System.Linq;
using System.Threading.Tasks;
using System.Collections.Concurrent;
using Microsoft.AspNet.SignalR;

namespace MapUsersSample
{
    [Authorize]
    public class ChatHub : Hub
    {
        public void SendChatMessage(string who, string message)
        {
            var name = Context.User.Identity.Name;
            using (var db = new UserContext())
            {
                var user = db.Users.Find(who);
                if (user == null)
                {
                    Clients.Caller.showErrorMessage("Could not find that user.");
                }
                else
                {
                    db.Entry(user)
                        .Collection(u => u.Connections)
                        .Query()
                        .Where(c => c.Connected == true)
                        .Load();

                    if (user.Connections == null)
                    {
                        Clients.Caller.showErrorMessage("The user is no longer connected.");
                    }
                    else
                    {
```

```
                        foreach (var connection in user.Connections)
                        {
                            Clients.Client(connection.ConnectionID)
                                .addChatMessage(name + ": " + message);
                        }
                    }
                }
            }
        }

        public override Task OnConnected()
        {
            var name = Context.User.Identity.Name;
            using (var db = new UserContext())
            {
                var user = db.Users
                    .Include(u => u.Connections)
                    .SingleOrDefault(u => u.UserName == name);

                if (user == null)
                {
                    user = new User
                    {
                        UserName = name,
                        Connections = new List<Connection>()
                    };
                    db.Users.Add(user);
                }

                user.Connections.Add(new Connection
                {
                    ConnectionID = Context.ConnectionId,
                    UserAgent = Context.Request.Headers["User-Agent"],
                    Connected = true
                });
                db.SaveChanges();
            }
            return base.OnConnected();
        }

        public override Task OnDisconnected(bool stopCalled)
        {
            using (var db = new UserContext())
            {
                var connection = db.Connections.Find(Context.ConnectionId);
                connection.Connected = false;
                db.SaveChanges();
            }
            return base.OnDisconnected(stopCalled);
        }
    }
}
```

Azure table storage

The following Azure table storage example is similar to the database example. It does not include all of the information that you would need to get started with Azure Table Storage Service. For information, see How to use Table storage from .NET.

The following example shows a table entity for storing connection information. It partitions the data by user name, and identifies each entity by the connection id, so a user can have multiple connections at any time.

```
using Microsoft.WindowsAzure.Storage.Table;
using System;

namespace MapUsersSample
{
    public class ConnectionEntity : TableEntity
    {
        public ConnectionEntity() { }

        public ConnectionEntity(string userName, string connectionID)
        {
            this.PartitionKey = userName;
            this.RowKey = connectionID;
        }
    }
}
```

In the hub, you track the status of each user's connection.

```
using Microsoft.AspNet.SignalR;
using Microsoft.WindowsAzure;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Table;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace MapUsersSample
{
    public class ChatHub : Hub
    {
        public void SendChatMessage(string who, string message)
        {
            var name = Context.User.Identity.Name;

            var table = GetConnectionTable();

            var query = new TableQuery<ConnectionEntity>()
                .Where(TableQuery.GenerateFilterCondition(
                "PartitionKey",
                QueryComparisons.Equal,
                who));

            var queryResult = table.ExecuteQuery(query).ToList();
            if (queryResult.Count == 0)
            {
                Clients.Caller.showErrorMessage("The user is no longer connected.");
            }
            else
            {
                foreach (var entity in queryResult)
                {
                    Clients.Client(entity.RowKey).addChatMessage(name + ": " + message);
                }
            }
        }

        public override Task OnConnected()
        {
            var name = Context.User.Identity.Name;
            var table = GetConnectionTable();
            table.CreateIfNotExists();

            var entity = new ConnectionEntity(
                name.ToLower(),
                Context.ConnectionId);
```

```
            var insertOperation = TableOperation.InsertOrReplace(entity);
            table.Execute(insertOperation);

            return base.OnConnected();
        }

        public override Task OnDisconnected(bool stopCalled)
        {
            var name = Context.User.Identity.Name;
            var table = GetConnectionTable();

            var deleteOperation = TableOperation.Delete(
                new ConnectionEntity(name, Context.ConnectionId) { ETag = "*" });
            table.Execute(deleteOperation);

            return base.OnDisconnected(stopCalled);
        }

        private CloudTable GetConnectionTable()
        {
            var storageAccount =
                CloudStorageAccount.Parse(
                CloudConfigurationManager.GetSetting("StorageConnectionString"));
            var tableClient = storageAccount.CreateCloudTableClient();
            return tableClient.GetTableReference("connection");
        }
    }
}
```

SignalR Testing and Debugging

2/20/2020 • 2 minutes to read • Edit Online

How to test and debug SignalR applications.

- SignalR Troubleshooting
- Unit Testing SignalR Applications
- Enabling SignalR Tracing

SignalR Troubleshooting

2/20/2020 • 17 minutes to read • Edit Online

by Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This document describes common troubleshooting issues with SignalR.

Software versions used in this topic

- Visual Studio 2013
- .NET 4.5
- SignalR version 2

Previous versions of this topic

For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

This document contains the following sections.

- Calling methods between the client and server silently fails
- Configuring IIS websockets to ping/pong to detect a dead client
- Other connection issues
- Compilation and server-side errors
- Visual Studio issues
- Internet Information Services issues
- Microsoft Azure issues

Calling methods between the client and server silently fails

This section describes possible causes for a method call between client and server to fail without a meaningful error message. In a SignalR application, the server has no information about the methods that the client implements; when the server invokes a client method, the method name and parameter data are sent to the client, and the method is executed only if it exists in the format that the server specified. If no matching method is found on the client, nothing happens, and no error message is raised on the server.

To further investigate client methods not getting called, you can turn on logging before the calling the start method on the hub to see what calls are coming from the server. To enable logging in a JavaScript application, see How to enable client-side logging (JavaScript client version). To enable logging in a .NET client application, see How to enable client-side logging (.NET Client version).

Misspelled method, incorrect method signature, or incorrect hub name

If the name or signature of a called method does not exactly match an appropriate method on the client, the call will fail. Verify that the method name called by the server matches the name of the method on the client. Also, SignalR creates the hub proxy using camel-cased methods, as is appropriate in JavaScript, so a method called `SendMessage` on the server would be called `sendMessage` in the client proxy. If you use the `HubName` attribute in your server-side code, verify that the name used matches the name used to create the hub on the client. If you do not use the `HubName` attribute, verify that the name of the hub in a JavaScript client is camel-cased, such as chatHub instead of ChatHub.

Duplicate method name on client

Verify that you do not have a duplicate method on the client that differs only by case. If your client application has a method called `sendMessage`, verify that there isn't also a method called `SendMessage` as well.

Missing JSON parser on the client

SignalR requires a JSON parser to be present to serialize calls between the server and the client. If your client doesn't have a built-in JSON parser (such as Internet Explorer 7), you'll need to include one in your application. You can download the JSON parser here.

Mixing Hub and PersistentConnection syntax

SignalR uses two communication models: Hubs and PersistentConnections. The syntax for calling these two communication models is different in the client code. If you have added a hub in your server code, verify that all of your client code uses the proper hub syntax.

JavaScript client code that creates a PersistentConnection in a JavaScript client

```
var myConnection = $.connection('/echo');
```

JavaScript client code that creates a Hub Proxy in a Javascript client

```
var myHub = $.connection.MyHub;
```

C# server code that maps a route to a PersistentConnection

```
RouteTable.Routes.MapConnection<MyConnection>("my", "/echo");
```

C# server code that maps a route to a Hub, or to multiple hubs if you have multiple applications

```
App.MapSignalR();
```

Connection started before subscriptions are added

If the Hub's connection is started before methods that can be called from the server are added to the proxy, messages will not be received. The following JavaScript code will not start the hub properly:

Incorrect JavaScript client code that will not allow Hubs messages to be received

```
var chat = $.connection.chatHub;
$.connection.hub.start().done(function () {
    chat.client.broadcastMessage = function (name, message) {...};
});
```

Instead, add the method subscriptions before calling Start:

JavaScript client code that correctly adds subscriptions to a hub

```
var chat = $.connection.chatHub;
chat.client.broadcastMessage = function (name, message) {...};
    $.connection.hub.start().done(function () {
        ...
    });
```

Missing method name on the hub proxy

Verify that the method defined on the server is subscribed to on the client. Even though the server defines the method, it must still be added to the client proxy. Methods can be added to the client proxy in the following ways (Note that the method is added to the `client` member of the hub, not the hub directly):

JavaScript client code that adds methods to a hub proxy

```
// Method added to proxy in JavaScript:
myHubProxy.server.method1 = function (param1, param2) {...};
//Multiple methods added to proxy in JavaScript using jQuery:
$.extend(myHubProxy.server, {
    method1: function (param1, param2) {...},
    method2: function (param3, param4) {...}
});
```

Hub or hub methods not declared as Public

To be visible on the client, the hub implementation and methods must be declared as `public`.

Accessing hub from a different application

SignalR Hubs can only be accessed through applications that implement SignalR clients. SignalR cannot interoperate with other communication libraries (like SOAP or WCF web services.) If there is no SignalR client available for your target platform, you cannot access the server's endpoint directly.

Manually serializing data

SignalR will automatically use JSON to serialize your method parameters- there's no need to do it yourself.

Remote Hub method not executed on client in OnDisconnected function

This behavior is by design. When `OnDisconnected` is called, the hub has already entered the `Disconnected` state, which does not allow further hub methods to be called.

C# server code that correctly executes code in the OnDisconnected event

```
public class MyHub : Hub
{
    public override Task OnDisconnected()
    {
        // Do what you want here
        return base.OnDisconnected();
    }
}
```

OnDisconnect not firing at consistent times

This behavior is by design. When a user attempts to navigate away from a page with an active SignalR connection, the SignalR client will then make a best-effort attempt to notify the server that the client connection will be stopped. If the SignalR client's best-effort attempt fails to reach the server, the server will dispose of the connection after a configurable `DisconnectTimeout` later, at which time the `OnDisconnected` event will fire. If the SignalR client's best-effort attempt is successful, the `OnDisconnected` event will fire immediately.

For information on setting the `DisconnectTimeout` setting, see Handling connection lifetime events: DisconnectTimeout.

Connection limit reached

When using the full version of IIS on a client operating system like Windows 7, a 10-connection limit is imposed. When using a client OS, use IIS Express instead to avoid this limit.

Cross-domain connection not set up properly

If a cross-domain connection (a connection for which the SignalR URL is not in the same domain as the hosting page) is not set up correctly, the connection may fail without an error message. For information on how to enable cross-domain communication, see How to establish a cross-domain connection.

Connection using NTLM (Active Directory) not working in .NET client

A connection in a .NET client application that uses Domain security may fail if the connection is not configured properly. To use SignalR in a domain environment, set the requisite connection property as follows:

C# client code that implements connection credentials

```
connection.Credentials = CredentialCache.DefaultCredentials;
```

Configuring IIS websockets to ping/pong to detect a dead client

SignalR servers don't know if the client is dead or not and they rely on notification from the underlying websocket for connection failures, that is, the `OnClose` callback. One solution to this problem is to configure IIS websockets to do the ping/pong for you. This ensures that your connection will close if it breaks unexpectedly. For more information see this stackoverflow post.

Other connection issues

This section describes the causes and solutions for specific symptoms or error messages that occur during a connection.

"Start must be called before data can be sent" error

This error is commonly seen if code references SignalR objects before the connection is started. The wireup for handlers and the like that will call methods defined on the server must be added after the connection completes. Note that the call to `Start` is asynchronous, so code after the call may be executed before it completes. The best way to add handlers after a connection starts completely is to put them into a callback function that is passed as a parameter to the start method:

JavaScript client code that correctly adds event handlers that reference SignalR objects

```
$.connection.hub.start().done(function () {
    // Wire up Send button to call NewContosoChatMessage on the server.
    $('#newContosoChatMessage').click(function () {
        contosoChatHubProxy.server.newContosoChatMessage(
            $('#displayname').val(), $('#message').val());
            $('#message').val('').focus();
    });
```

This error will also be seen if a connection stops while SignalR objects are still being referenced.

"301 Moved Permanently" or "302 Moved Temporarily" error

This error may be seen if the project contains a folder called SignalR, which will interfere with the automatically-created proxy. To avoid this error, do not use a folder called `SignalR` in your application, or turn automatic proxy

generation off. See The Generated Proxy and what it does for you for more details.

"403 Forbidden" error in .NET or Silverlight client

This error may occur in cross-domain environments where cross-domain communication is not properly enabled. For information on how to enable cross-domain communication, see How to establish a cross-domain connection. To establish a cross-domain connection in a Silverlight client, see Cross-domain connections from Silverlight clients.

"404 Not Found" error

There are several causes for this issue. Verify all of the following:

- **Hub proxy address reference not formatted correctly:** This error is commonly seen if the reference to the generated hub proxy address is not formatted correctly. Verify that the reference to the hub address is made properly. See How to reference the dynamically generated proxy for details.
- **Adding routes to application before adding the hub route:** If your application uses other routes, verify that the first route added is the call to `MapSignalR`.
- **Using IIS 7 or 7.5 without the update for extensionless URLs:** Using IIS 7 or 7.5 requires an update for extensionless URLs so that the server can provide access to the hub definitions at `/signalr/hubs`. The update can be found here.
- **IIS cache out of date or corrupt:** To verify that the cache contents are not out of date, enter the following command in a PowerShell window to clear the cache:

```
net stop w3svc
Remove-Item -Path "C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Temporary ASP.NET Files\root\*" -
Force -Recurse
net start w3svc
```

"500 Internal Server Error"

This is a very generic error that could have a wide variety of causes. The details of the error should appear in the server's event log, or can be found through debugging the server. More detailed error information may be obtained by turning on detailed errors on the server. For more information, see How to handle errors in the Hub class.

This error is also commonly seen if a firewall or proxy is not configured properly, causing the request headers to be rewritten. The solution is to make sure that port 80 is enabled on the firewall or proxy.

"Unexpected response code: 500"

This error may occur if the version of .NET framework used in the application does not match the version specified in Web.Config. The solution is to verify that .NET 4.5 is used in both the application settings and the Web.Config file.

"TypeError: <hubType> is undefined" error

This error will result if the call to `MapSignalR` is not made properly. See How to register SignalR Middleware and configure SignalR options for more information.

JsonSerializationException was unhandled by user code

Verify that the parameters you send to your methods do not include non-serializable types (like file handles or database connections). If you need to use members on a server-side object that you don't want to be sent to the client (either for security or for reasons of serialization), use the `JSONIgnore` attribute.

"Protocol error: Unknown transport" error

This error may occur if the client does not support the transports that SignalR uses. See Transports and Fallbacks

for information on which browsers can be used with SignalR.

"JavaScript Hub proxy generation has been disabled."

This error will occur if `DisableJavaScriptProxies` is set while also including a reference to the dynamically generated proxy at `signalr/hubs`. For more information on creating the proxy manually, see The generated proxy and what it does for you.

"The connection ID is in the incorrect format" or "The user identity cannot change during an active SignalR connection" error

This error may be seen if authentication is being used, and the client is logged out before the connection is stopped. The solution is to stop the SignalR connection before logging the client out.

"Uncaught Error: SignalR: jQuery not found. Please ensure jQuery is referenced before the SignalR.js file" error

The SignalR JavaScript client requires jQuery to run. Verify that your reference to jQuery is correct, that the path used is valid, and that the reference to jQuery is before the reference to SignalR.

"Uncaught TypeError: Cannot read property '<property>' of undefined" error

This error results from not having jQuery or the hubs proxy referenced properly. Verify that your reference to jQuery and the hubs proxy is correct, that the path used is valid, and that the reference to jQuery is before the reference to the hubs proxy. The default reference to the hubs proxy should look like the following:

HTML client-side code that correctly references the Hubs proxy

```
<script src="/signalr/hubs"></script>
```

"RuntimeBinderException was unhandled by user code" error

This error may occur when the incorrect overload of `Hub.On` is used. If the method has a return value, the return type must be specified as a generic type parameter:

Method defined on the client (without generated proxy)

```
MyHub.On<ReturnType>("MethodName", LocalMethod);
```

Connection ID is inconsistent or connection breaks between page loads

This behavior is by design. Since the hub object is hosted in the page object, the hub is destroyed when the page refreshes. A multi-page application needs to maintain the association between users and connection IDs so that they will be consistent between page loads. The connection IDs can be stored on the server in either a `ConcurrentDictionary` object or a database.

"Value cannot be null" error

Server-side methods with optional parameters are not currently supported; if the optional parameter is omitted, the method will fail. For more information, see Optional Parameters.

"Firefox can't establish a connection to the server at <address>" error in Firebug

This error message can be seen in Firebug if negotiation of the WebSocket transport fails and another transport is used instead. This behavior is by design.

"The remote certificate is invalid according to the validation procedure" error in .NET client application

If your server requires custom client certificates, then you can add an x509certificate to the connection before the request is made. Add the certificate to the connection using `Connection.AddClientCertificate`.

Connection drops after authentication times out

This behavior is by design. Authentication credentials cannot be modified while a connection is active; to refresh

credentials, the connection must be stopped and restarted.

OnConnected gets called twice when using jQuery Mobile

jQuery Mobile's `initializePage` function forces the scripts in each page to be re-executed, thus creating a second connection. Solutions for this issue include:

- Include the reference to jQuery Mobile before your JavaScript file.
- Disable the `initializePage` function by setting `$.mobile.autoInitializePage = false`.
- Wait for the page to finish initializing before starting the connection.

Messages are delayed in Silverlight applications using Server Sent Events

Messages are delayed when using server sent events on Silverlight. To force long polling to be used instead, use the following when starting the connection:

```
connection.Start(new LongPollingTransport());
```

"Permission Denied" using Forever Frame protocol

This is a known issue, described here. This symptom may be seen using the latest JQuery library; the workaround is to downgrade your application to JQuery 1.8.2.

"InvalidOperationException: Not a valid web socket request.

This error may occur if the WebSocket protocol is used, but the network proxy is modifying the request headers. The solution is to configure the proxy to allow WebSocket on port 80.

"Exception: <method name> method could not be resolved" when client calls method on server

This error can result from using data types that cannot be discovered in a JSON payload, such as Array. The workaround is to use a data type that is discoverable by JSON, such as IList. For more information, see .NET Client unable to call hub methods with array parameters.

Compilation and server-side errors

The following section contains possible solutions to compiler and server-side runtime errors.

Reference to Hub instance is null

Since a hub instance is created for each connection, you can't create an instance of a hub in your code yourself. To call methods on a hub from outside the hub itself, see How to call client methods and manage groups from outside the Hub class for how to obtain a reference to the hub context.

HTTPContext.Current.Session is null

This behavior is by design. SignalR does not support the ASP.NET session state, since enabling the session state would break duplex messaging.

No suitable method to override

You may see this error if you are using code from older documentation or blogs. Verify that you are not referencing names of methods that have been changed or deprecated (like `OnConnectedAsync`).

HostContextExtensions.WebSocketServerUrl is null

This behavior is by design. This member is deprecated and should not be used.

"A route named 'signalr.hubs' is already in the route collection" error

This error will be seen if `MapSignalR` is called twice by your application. Some example applications call `MapSignalR` directly in the Startup class; others make the call in a wrapper class. Ensure that your application does not do both.

WebSocket is not used

If you have verified that your server and clients meet the requirements for WebSocket (listed in the Supported Platforms document), you will need to enable WebSocket on your server. Instructions for doing this can be found here.

$.connection is undefined

This error indicates that either the scripts on a page are not being loaded properly, or the hub proxy is not reachable or is being accessed incorrectly. Verify that the script references on your page correspond to the scripts loaded in your project, and that /signalr/hubs can be accessed in a browser when the server is running.

One or more types required to compile a dynamic expression cannot be found

This error indicates that the `Microsoft.CSharp` library is missing. Add it in the **Assemblies->Framework** tab.

Caller state cannot be accessed from Clients.Caller in Visual Basic or in a strongly typed hub; "Conversion from type 'Task(Of Object)' to type 'String' is not valid" error

To access caller state in Visual Basic or in a strongly typed hub, use the `Clients.CallerState` property (introduced in SignalR 2.1) instead of `Clients.Caller`.

Visual Studio issues

This section describes issues encountered in Visual Studio.

Script Documents node does not appear in Solution Explorer

Some of our tutorials direct you to the "Script Documents" node in Solution Explorer while debugging. This node is produced by the JavaScript debugger, and will only appear while debugging browser clients in Internet Explorer; the node will not appear if Chrome or Firefox are used. The JavaScript debugger will also not run if another client debugger is running, such as the Silverlight debugger.

SignalR does not work on Visual Studio 2008 or earlier

This behavior is by design. SignalR requires .NET Framework 4 or later; this requires that SignalR applications be developed in Visual Studio 2010 or later. The server component of SignalR requires .NET Framework 4.5.

IIS issues

This section contains issues with Internet Information Services.

SignalR works on Visual Studio development server, but not in IIS

SignalR is supported on IIS 7.0 and 7.5, but support for extensionless URLs must be added. To add support for extensionless URLs, see https://support.microsoft.com/kb/980368

SignalR requires ASP.NET to be installed on the server (ASP.NET is not installed on IIS by default). To install ASP.NET, see ASP.NET Downloads.

Microsoft Azure issues

This section contains issues with Microsoft Azure.

FileLoadException when hosting SignalR in an Azure Worker Role

Hosting SignalR in an Azure Worker Role might result in the exception "Could not load file or assembly 'Microsoft.Owin, Version=2.0.0.0". This is a known issue with NuGet; Binding redirects are not added automatically in Azure Worker Role projects. To fix this, you can add the binding redirects manually. Add the following lines to the `app.config` file for your Worker Role project.

```
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <dependentAssembly>
      <assemblyIdentity name="Microsoft.Owin" publicKeyToken="31bf3856ad364e35" culture="neutral" />
      <bindingRedirect oldVersion="0.0.0.0-2.0.2.0" newVersion="2.0.2.0" />
    </dependentAssembly>
    <dependentAssembly>
      <assemblyIdentity name="Microsoft.Owin.Security" publicKeyToken="31bf3856ad364e35" culture="neutral" />
      <bindingRedirect oldVersion="0.0.0.0-2.0.2.0" newVersion="2.0.2.0" />
    </dependentAssembly>
  </assemblyBinding>
</runtime>
```

Messages are not received through the Azure backplane after altering topic names

The topics used by the Azure backplane are maintained internally; they are not intended to be user-configurable.

Unit Testing SignalR Applications

2/20/2020 • 4 minutes to read • Edit Online

by Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This article describes using the Unit Testing features of SignalR 2.

Software versions used in this topic

- Visual Studio 2013
- .NET 4.5
- SignalR version 2

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Unit testing SignalR applications

You can use the unit test features in SignalR 2 to create unit tests for your SignalR application. SignalR 2 includes the IHubCallerConnectionContext interface, which can be used to create a mock object to simulate your hub methods for testing.

In this section, you'll add unit tests for the application created in the Getting Started tutorial using XUnit.net and Moq.

XUnit.net will be used to control the test; Moq will be used to create a mock object for testing. Other mocking frameworks can be used if desired; NSubstitute is also a good choice. This tutorial demonstrates how to set up the mock object in two ways: First, using a `dynamic` object (introduced in .NET Framework 4), and second, using an interface.

Contents

This tutorial contains the following sections.

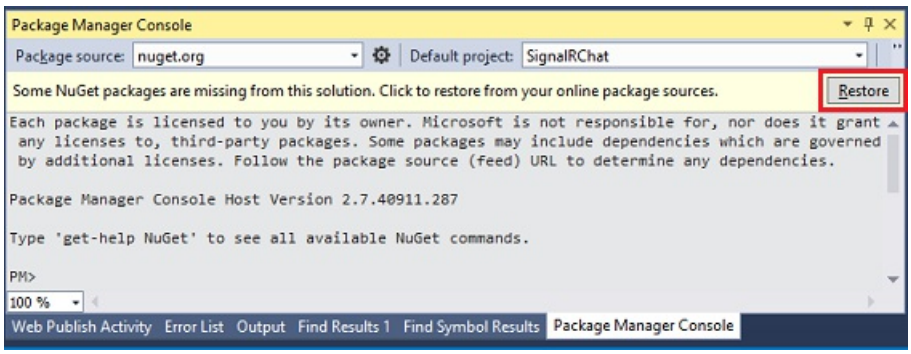
- Unit testing with Dynamic
- Unit testing by type

Unit testing with Dynamic

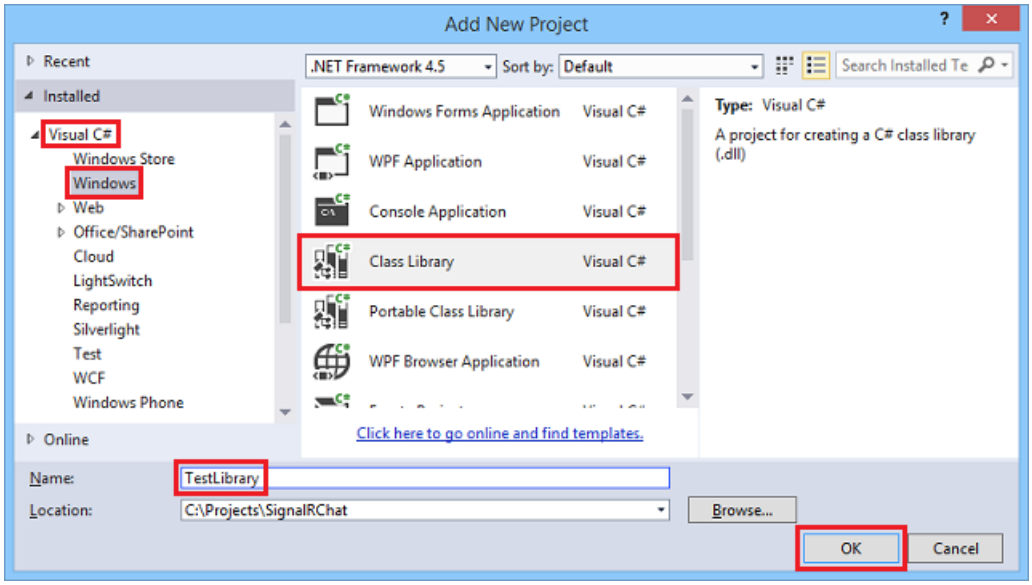
In this section, you'll add a unit test for the application created in the Getting Started tutorial using a dynamic object.

1. Install the XUnit Runner extension for Visual Studio 2013.
2. Either complete the Getting Started tutorial, or download the completed application from MSDN Code Gallery.

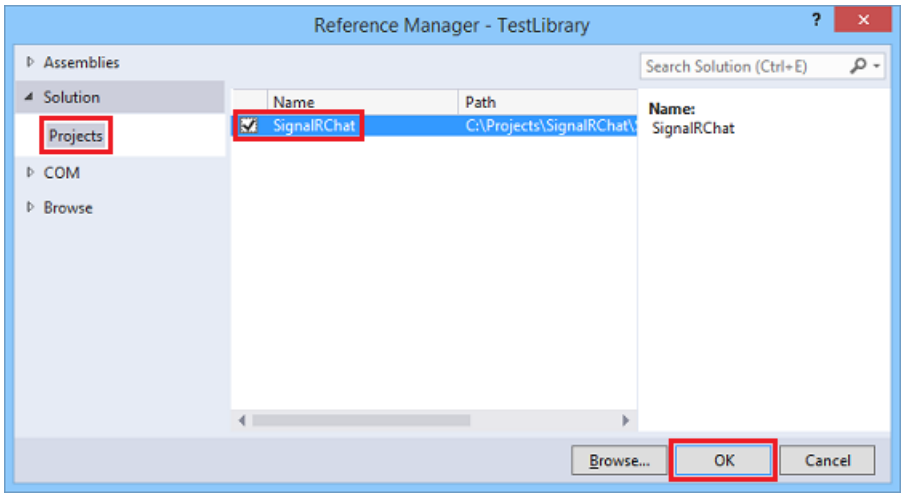
3. If you are using the download version of the Getting Started application, open **Package Manager Console** and click **Restore** to add the SignalR package to the project.



4. Add a project to the solution for the unit test. Right-click your solution in **Solution Explorer** and select **Add**, **New Project...**. Under the **C#** node, select the **Windows** node. Select **Class Library**. Name the new project **TestLibrary** and click **OK**.



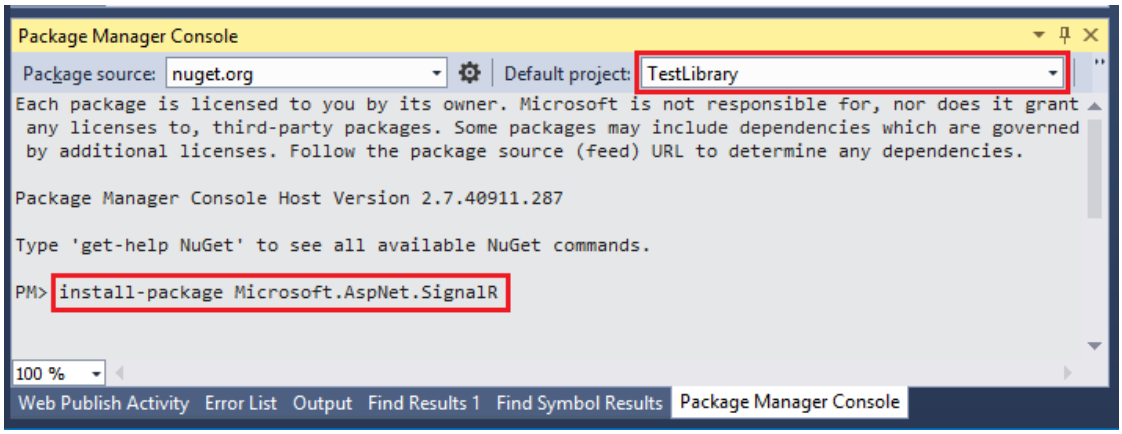
5. Add a reference in the test library project to the SignalRChat project. Right-click the **TestLibrary** project and select **Add**, **Reference...**. Select the **Projects** node under the **Solution** node, and check **SignalRChat**. Click **OK**.



6. Add the SignalR, Moq, and XUnit packages to the **TestLibrary** project. In the **Package Manager Console**, set the **Default Project** dropdown to **TestLibrary**. Run the following commands in the console window:

   - `Install-Package Microsoft.AspNet.SignalR`

- `Install-Package Moq`
- `Install-Package XUnit`



7. Create the test file. Right-click the **TestLibrary** project and click **Add...**, **Class**. Name the new class **Tests.cs**.

8. Replace the contents of Tests.cs with the following code.

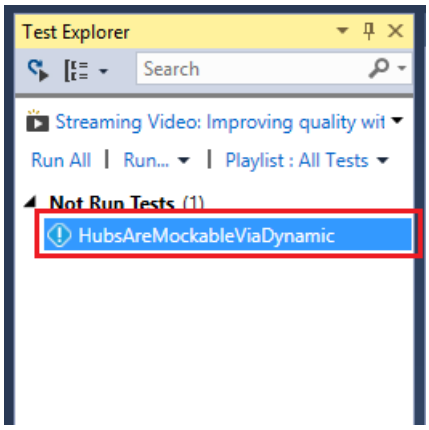
```
using System;
using Xunit;
using SignalRChat;
using Microsoft.AspNet.SignalR.Hubs;
using Moq;
using System.Dynamic;

namespace TestLibrary
{
    public class Tests
    {
        [Fact]
        public void HubsAreMockableViaDynamic()
        {
            bool sendCalled = false;
            var hub = new ChatHub();
            var mockClients = new Mock<IHubCallerConnectionContext<dynamic>>();
            hub.Clients = mockClients.Object;
            dynamic all = new ExpandoObject();
            all.broadcastMessage = new Action<string, string>((name, message) => {
                sendCalled = true;
            });
            mockClients.Setup(m => m.All).Returns((ExpandoObject)all);
            hub.Send("TestUser", "TestMessage");
            Assert.True(sendCalled);
        }
    }
}
```

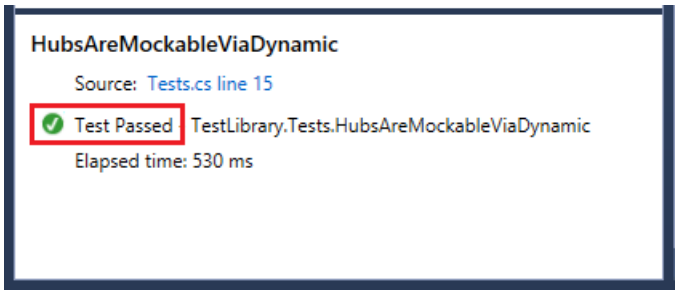
In the code above, a test client is created using the `Mock` object from the Moq library, of type IHubCallerConnectionContext (in SignalR 2.1, assign `dynamic` for the type parameter.) The `IHubCallerConnectionContext` interface is the proxy object with which you invoke methods on the client. The `broadcastMessage` function is then defined for the mock client so that it can be called by the `ChatHub` class. The test engine then calls the `Send` method of the `ChatHub` class, which in turn calls the mocked `broadcastMessage` function.

9. Build the solution by pressing **F6**.

10. Run the unit test. In Visual Studio, select **Test**, **Windows**, **Test Explorer**. In the Test Explorer window, right-click **HubsAreMockableViaDynamic** and select **Run Selected Tests**.



11. Verify that the test passed by checking the lower pane in the Test Explorer window. The window will show that the test passed.



Unit testing by type

In this section, you'll add a test for the application created in the Getting Started tutorial using an interface that contains the method to be tested.

1. Complete steps 1-7 in the Unit testing with Dynamic tutorial above.

2. Replace the contents of Tests.cs with the following code.

```
using Xunit;
using SignalRChat;
using Microsoft.AspNet.SignalR.Hubs;
using Moq;

namespace TestLibrary
{
    public class Tests
    {

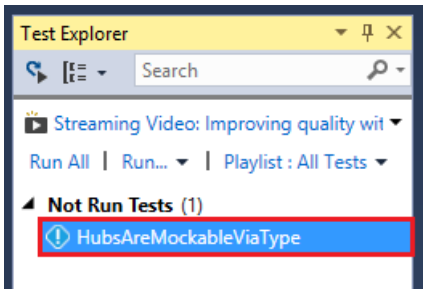
       public interface IClientContract
       {
           void broadcastMessage(string name, string message);
       }
       [Fact]
       public void HubsAreMockableViaType()
       {
           var hub = new ChatHub();
           var mockClients = new Mock<IHubCallerConnectionContext<dynamic>>();
           var all = new Mock<IClientContract>();
           hub.Clients = mockClients.Object;
           all.Setup(m => m.broadcastMessage(It.IsAny<string>(),
                It.IsAny<string>())).Verifiable();
           mockClients.Setup(m => m.All).Returns(all.Object);
           hub.Send("TestUser", "TestMessage");
           all.VerifyAll();
       }
    }
}
```

In the code above, an interface is created defining the signature of the `broadcastMessage` method for which the test engine will create a mock client. A mock client is then created using the `Mock` object, of type IHubCallerConnectionContext (in SignalR 2.1, assign `dynamic` for the type parameter.) The `IHubCallerConnectionContext` interface is the proxy object with which you invoke methods on the client.

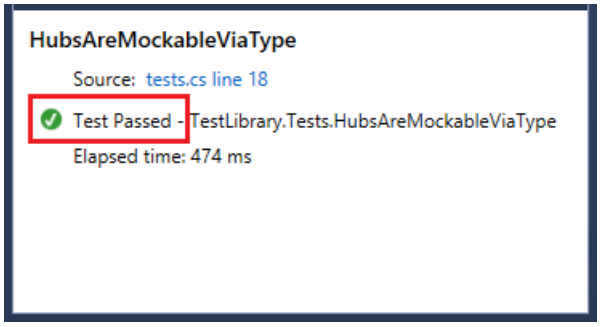
The test then creates an instance of `ChatHub`, and then creates a mock version of the `broadcastMessage` method, which in turn is invoked by calling the `Send` method on the hub.

3. Build the solution by pressing **F6**.

4. Run the unit test. In Visual Studio, select **Test**, **Windows**, **Test Explorer**. In the Test Explorer window, right-click **HubsAreMockableViaDynamic** and select **Run Selected Tests**.



5. Verify that the test passed by checking the lower pane in the Test Explorer window. The window will show that the test passed.



Enabling SignalR Tracing

2/20/2020 • 7 minutes to read • Edit Online

by Tom FitzMacken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This document describes how to enable and configure tracing for SignalR servers and clients. Tracing enables you to view diagnostic information about events in your SignalR application.

This topic was originally written by Patrick Fletcher.

Software versions used in the tutorial

- Visual Studio 2013
- .NET Framework 4.5
- SignalR version 2

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

When tracing is enabled, a SignalR application creates log entries for events. You can log events from both the client and the server. Tracing on the server logs connection, scaleout provider, and message bus events. Tracing on the client logs connection events. In SignalR 2.1 and later, tracing on the client logs the full content of hub invocation messages.

Contents

- Enabling tracing on the server
  - Logging server events to text files
  - Logging server events to the event log
- Enabling tracing in the .NET client (Windows Desktop apps)
  - Logging Desktop client events to the console
  - Logging Desktop client events to a text file
- Enabling tracing in Windows Phone 8 clients
  - Logging Windows Phone client events to the UI
  - Logging Windows Phone client events to the debug console
- Enabling tracing in the JavaScript client

Enabling tracing on the server

You enable tracing on the server within the application's configuration file (either App.config or Web.config depending on the type of project.) You specify which categories of events you want to log. In the configuration file, you also specify whether to log the events to a text file, the Windows event log, or a custom log using an implementation of TraceListener.

The server event categories include the following sorts of messages:

SOURCE	MESSAGES
SignalR.SqlMessageBus	SQL Message Bus scaleout provider setup, database operation, error, and timeout events
SignalR.ServiceBusMessageBus	Service bus scaleout provider topic creation and subscription, error, and messaging events
SignalR.RedisMessageBus	Redis scaleout provider connection, disconnection, and error events
SignalR.ScaleoutMessageBus	Scaleout messaging events
SignalR.Transports.WebSocketTransport	WebSocket transport connection, disconnection, messaging, and error events
SignalR.Transports.ServerSentEventsTransport	ServerSentEvents transport connection, disconnection, messaging, and error events
SignalR.Transports.ForeverFrameTransport	ForeverFrame transport connection, disconnection, messaging, and error events
SignalR.Transports.LongPollingTransport	LongPolling transport connection, disconnection, messaging, and error events
SignalR.Transports.TransportHeartBeat	Transport connection, disconnection, and keepalive events
SignalR.ReflectedHubDescriptorProvider	Hub discovery events

Logging server events to text files

The following code shows how to enable tracing for each category of event. This sample configures the application to log events to text files.

XML server code for enabling tracing

```
<system.diagnostics>
    <sources>
      <source name="SignalR.SqlMessageBus">
        <listeners>
          <add name="SignalR-Bus" />
        </listeners>
      </source>
      <source name="SignalR.ServiceBusMessageBus">
        <listeners>
          <add name="SignalR-Bus" />
        </listeners>
      </source>
      <source name="SignalR.RedisMessageBus">
        <listeners>
          <add name="SignalR-Bus" />
        </listeners>
      </source>
```

```
      </source>
      <source name="SignalR.ScaleoutMessageBus">
        <listeners>
          <add name="SignalR-Bus" />
        </listeners>
      </source>
      <source name="SignalR.Transports.WebSocketTransport">
        <listeners>
          <add name="SignalR-Transports" />
        </listeners>
      </source>
      <source name="SignalR.Transports.ServerSentEventsTransport">
        <listeners>
          <add name="SignalR-Transports" />
        </listeners>
      </source>
      <source name="SignalR.Transports.ForeverFrameTransport">
        <listeners>
          <add name="SignalR-Transports" />
        </listeners>
      </source>
      <source name="SignalR.Transports.LongPollingTransport">
        <listeners>
          <add name="SignalR-Transports" />
        </listeners>
      </source>
      <source name="SignalR.Transports.TransportHeartBeat">
        <listeners>
          <add name="SignalR-Transports" />
        </listeners>
      </source>
      <source name="SignalR.ReflectedHubDescriptorProvider">
        <listeners>
          <add name="SignalR-Init" />
        </listeners>
      </source>
    </sources>
    <!-- Sets the trace verbosity level -->
    <switches>
      <add name="SignalRSwitch" value="Verbose" />
    </switches>
    <!-- Specifies the trace writer for output -->
    <sharedListeners>
      <!-- Listener for transport events -->
      <add name="SignalR-Transports" type="System.Diagnostics.TextWriterTraceListener"
initializeData="transports.log.txt" />
      <!-- Listener for scaleout provider events -->
      <add name="SignalR-Bus" type="System.Diagnostics.TextWriterTraceListener" initializeData="bus.log.txt"
/>
      <!-- Listener for hub discovery events -->
      <add name="SignalR-Init" type="System.Diagnostics.TextWriterTraceListener" initializeData="init.log.txt"
/>
    </sharedListeners>
    <trace autoflush="true" />
  </system.diagnostics>
```

In the code above, the `SignalRSwitch` entry specifies the TraceLevel used for events sent to the specified log. In this case, it is set to `Verbose` which means all debugging and tracing messages are logged.

The following output shows entries from the `transports.log.txt` file for an application using the above configuration file. It shows a new connection, a removed connection, and transport heartbeat events.

```
SignalR.Transports.TransportHeartBeat Information: 0 : Connection 9aa62c9b-09b3-416c-b367-06520e24f780 is New.
SignalR.Transports.TransportHeartBeat Verbose: 0 : KeepAlive(9aa62c9b-09b3-416c-b367-06520e24f780)
SignalR.Transports.TransportHeartBeat Verbose: 0 : KeepAlive(9aa62c9b-09b3-416c-b367-06520e24f780)
SignalR.Transports.TransportHeartBeat Verbose: 0 : KeepAlive(9aa62c9b-09b3-416c-b367-06520e24f780)
SignalR.Transports.WebSocketTransport Information: 0 : CloseSocket(9aa62c9b-09b3-416c-b367-06520e24f780)
SignalR.Transports.WebSocketTransport Information: 0 : Abort(9aa62c9b-09b3-416c-b367-06520e24f780)
SignalR.Transports.TransportHeartBeat Information: 0 : Removing connection 9aa62c9b-09b3-416c-b367-
06520e24f780
SignalR.Transports.WebSocketTransport Information: 0 : End(9aa62c9b-09b3-416c-b367-06520e24f780)
SignalR.Transports.WebSocketTransport Verbose: 0 : DrainWrites(9aa62c9b-09b3-416c-b367-06520e24f780)
SignalR.Transports.WebSocketTransport Information: 0 : CompleteRequest (9aa62c9b-09b3-416c-b367-06520e24f780)
```

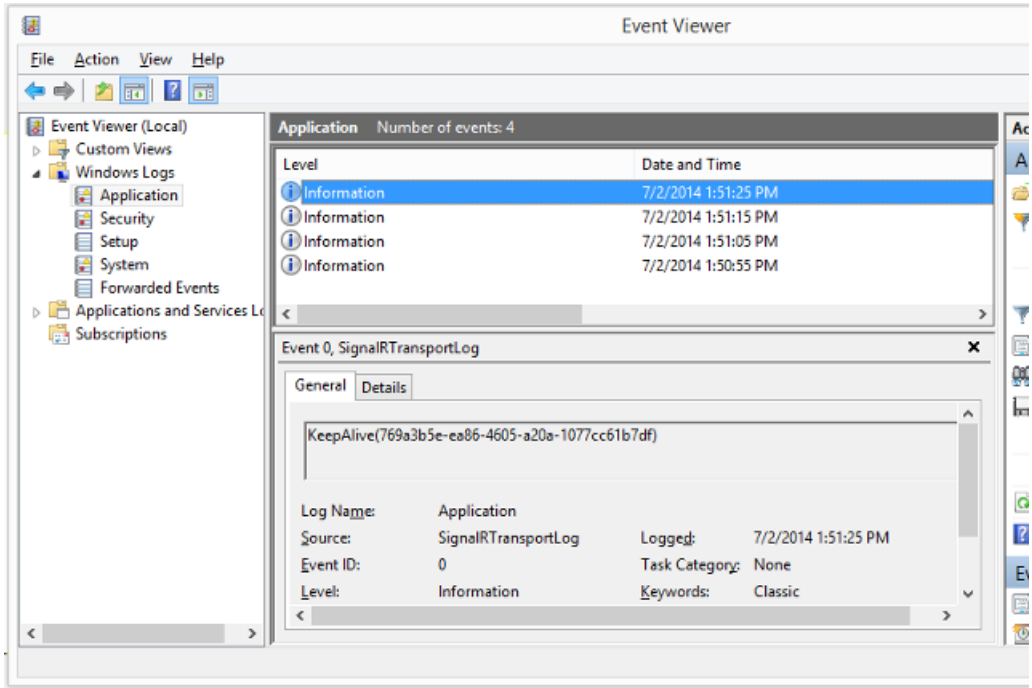
Logging server events to the event log

To log events to the event log rather than a text file, change the values for the entries in the `sharedListeners` node. The following code shows how to log server events to the event log:

XML server code for logging events to the event log

```
<sharedListeners>
  <!-- Listener for transport events -->
  <add name="SignalR-Transports" type="System.Diagnostics.EventLogTraceListener"
initializeData="SignalRTransportLog" />
  <!-- Listener for scaleout provider events -->
  <add name="SignalR-Bus" type="System.Diagnostics.EventLogTraceListener" initializeData="SignalRScaleoutLog"
/>
  <!-- Listener for hub discovery events -->
  <add name="SignalR-Init" type="System.Diagnostics.EventLogTraceListener" initializeData="SignalRInitLog" />
</sharedListeners>
```

The events are logged in the Application log, and are available through the Event Viewer, as shown below:



> **NOTE**
>
> When using the event log, set the **TraceLevel** to **Error** to keep the number of messages manageable.

Enabling tracing in the .NET client (Windows Desktop apps)

The .NET client can log events to the console, a text file, or to a custom log using an implementation of TextWriter.

To enable logging in the .NET client, set the connection's `TraceLevel` property to a TraceLevels value, and the `TraceWriter` property to a valid TextWriter instance.

Logging Desktop client events to the console

The following C# code shows how to log events in the .NET client to the console:

```
var hubConnection = new HubConnection("http://www.contoso.com/");
hubConnection.TraceLevel = TraceLevels.All;
hubConnection.TraceWriter = Console.Out;
IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new price
{1}", stock.Symbol, stock.Price));
await hubConnection.Start();
```

Logging Desktop client events to a text file

The following C# code shows how to log events in the .NET client to a text file:

```
var hubConnection = new HubConnection("http://www.contoso.com/");
var writer = new StreamWriter("ClientLog.txt");
writer.AutoFlush = true;
hubConnection.TraceLevel = TraceLevels.All;
hubConnection.TraceWriter = writer;
IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new price
{1}", stock.Symbol, stock.Price));
await hubConnection.Start();
```

The following output shows entries from the `ClientLog.txt` file for an application using the above configuration file. It shows the client connecting to the server, and the hub invoking a client method called `addMessage`:

```
19:41:39.9103763 - null - ChangeState(Disconnected, Connecting)
19:41:40.3750726 - dd61fd48-d796-4518-b36b-ec1dcb970d72 - WS Connecting to:
ws://localhost:8080/signalr/signalr/connect?
transport=webSockets&clientProtocol=1.4&connectionToken=AQAAANCMnd8BFdERjHoAwE%2FCl%2BsBAAAAh8Lp
KH5%2FDkCQeR4ALAwR%2BAAAAAACAAAAAAADZgAAwAAAABAAAADHpCa7wm%2FbOhjluf%2Fm9GA9AAAAAASAAACgAAAAEA
AAAEqRfJihLExRI6tZy7lWRwYoAAAApotSsJXW0OiwEgiUUi0pzhK6oKbz%2BkMeVbezuEDQLnJecM9otFe9PRQAAAAuHK
BlOnPmXt%2FhXV%2Felr1QvC156Q%3D%3D&connectionData=[{"Name":"MyHub"}]
19:41:40.4442923 - dd61fd48-d796-4518-b36b-ec1dcb970d72 - WS: OnMessage({"C":"d-5196BF5C-
A,0|B,0|C,1|D,0","S":1,"M":[]})
19:41:40.4874324 - dd61fd48-d796-4518-b36b-ec1dcb970d72 - ChangeState(Connecting, Connected)
19:41:47.4511770 - dd61fd48-d796-4518-b36b-ec1dcb970d72 - WS: OnMessage({"C":"d-5196BF5C-A,1|B,0|C,1|D,0","M":
[{"H":"MyHub","M":"addMessage","A":["User One","Hello!"]}]})
19:41:47.4576968 - dd61fd48-d796-4518-b36b-ec1dcb970d72 - WS: OnMessage({"I":"0"})
19:41:50.3959119 - dd61fd48-d796-4518-b36b-ec1dcb970d72 - WS: OnMessage({})
19:41:50.8928084 - dd61fd48-d796-4518-b36b-ec1dcb970d72 - WS: OnMessage({"C":"d-5196BF5C-A,2|B,0|C,1|D,0","M":
[{"H":"MyHub","M":"addMessage","A":["User Two","Hello!"]}]})
```

Enabling tracing in Windows Phone 8 clients

SignalR applications for Windows Phone apps use the same .NET client as desktop apps, but Console.Out and writing to a file with StreamWriter are not available. Instead, you need to create a custom implementation of TextWriter for tracing.

Logging Windows Phone client events to the UI

The SignalR codebase includes a Windows Phone sample that writes trace output to a TextBlock using a custom TextWriter implementation called `TextBlockWriter`. This class can be found in the

samples/Microsoft.AspNet.SignalR.Client.WP8.Samples project. When creating an instance of `TextBlockWriter`, pass in the current SynchronizationContext, and a StackPanel where it will create a TextBlock to use for trace output:

```
Connection = new HubConnection(ServerURI);
var writer = new TextBlockWriter(SynchronizationContext.Current, StackPanelConsole);
Connection.TraceWriter = writer;
Connection.TraceLevel = TraceLevels.All;
```

The trace output will then be written to a new TextBlock created in the StackPanel you passed in:

Logging Windows Phone client events to the debug console

To send output to the debug console rather than the UI, create an implementation of TextWriter that writes to the debug window, and assign it to your connection's TraceWriter property:

```
Connection = new HubConnection(ServerURI);
var writer = new DebugTextWriter();
Connection.TraceWriter = writer;
Connection.TraceLevel = TraceLevels.All;

...

private class DebugTextWriter : TextWriter
{
    private StringBuilder buffer;

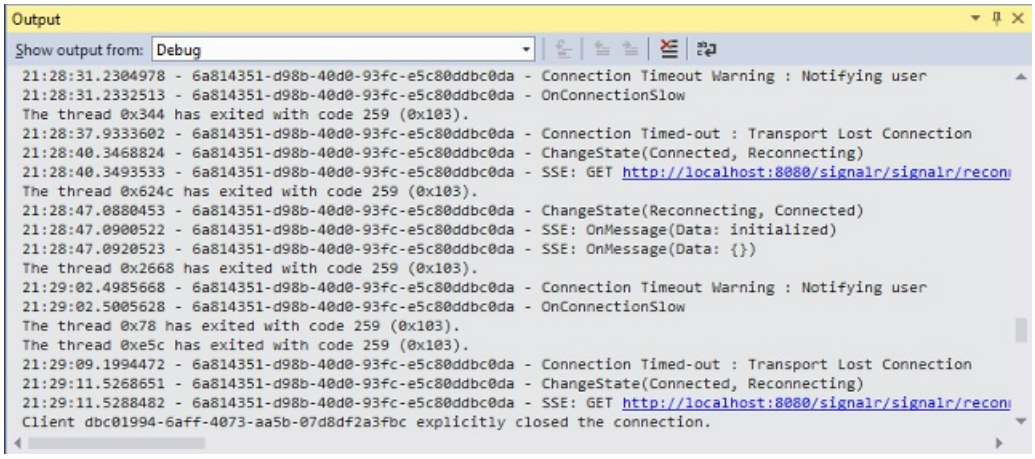
    public DebugTextWriter()
    {
        buffer = new StringBuilder();
    }

    public override void Write(char value)
    {
        switch (value)
        {
            case '\n':
                return;
            case '\r':
                Debug.WriteLine(buffer.ToString());
                buffer.Clear();
                return;
            default:
                buffer.Append(value);
                break;
        }
    }

    public override void Write(string value)
    {
        Debug.WriteLine(value);

    }
    #region implemented abstract members of TextWriter
    public override Encoding Encoding
    {
        get { throw new NotImplementedException(); }
    }
    #endregion
}
```

Trace information will then be written to the debug window in Visual Studio:



Enabling tracing in the JavaScript client

To enable client-side logging on a connection, set the `logging` property on the connection object before you call the `start` method to establish the connection.

Client JavaScript code for enabling tracing to the browser console (with the generated proxy)

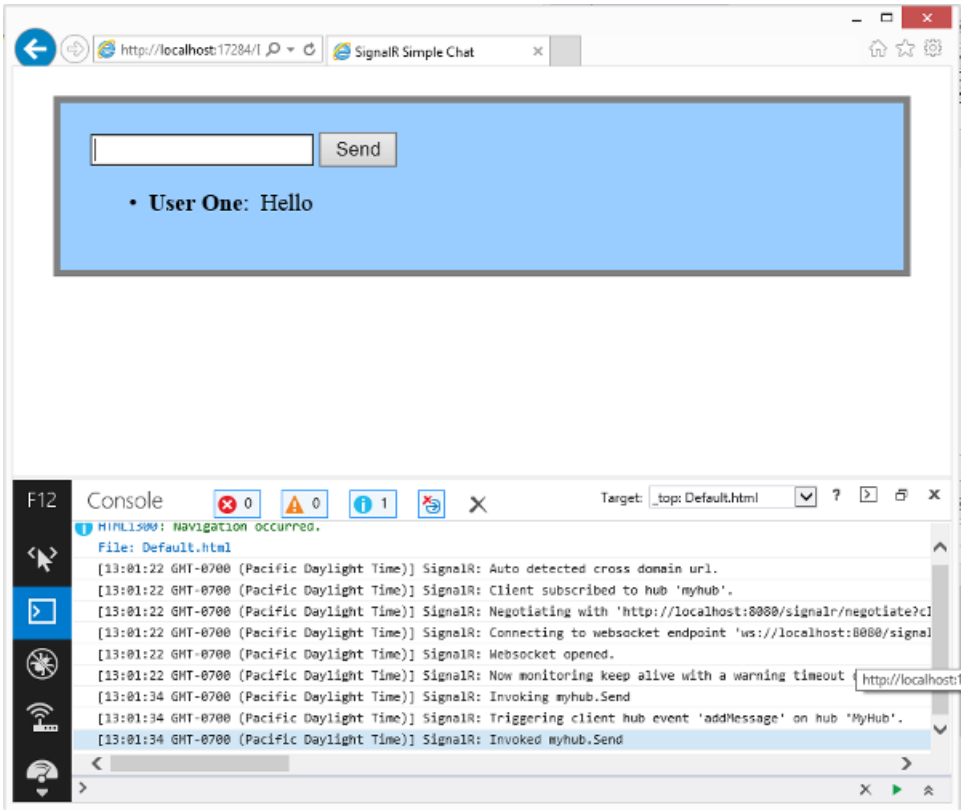
```
$.connection.hub.logging = true;
$.connection.hub.start();
```

Client JavaScript code for enabling tracing to the browser console (without the generated proxy)

```
var connection = $.hubConnection();
connection.logging = true;
connection.start();
```

When tracing is enabled, the JavaScript client logs events to the browser console. To access the browser console, see Monitoring Transports.

The following screenshot shows a SignalR JavaScript client with tracing enabled. It shows connection and hub invocation events in the browser console:



SignalR Security

2/20/2020 • 2 minutes to read • Edit Online

Understanding and handling security issues in SignalR applications.

- Introduction to SignalR Security
- Authentication and Authorization for SignalR Hubs
- Authentication and Authorization for SignalR Persistent Connections

Introduction to SignalR Security

2/20/2020 • 11 minutes to read • Edit Online

by Patrick Fletcher, Tom FitzMacken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This article describes the security issues you must consider when developing a SignalR application.

Software versions used in this topic

- Visual Studio 2013
- .NET 4.5
- SignalR version 2

Previous versions of this topic

For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Overview

This document contains the following sections:

- SignalR Security Concepts
  - Authentication and authorization
  - Connection token
  - Rejoining groups when reconnecting
- How SignalR prevents Cross-Site Request Forgery
- SignalR Security Recommendations
  - Secure Socket Layers (SSL) protocol
  - Do not use groups as a security mechanism
  - Safely handling input from clients
  - Reconciling a change in user status with an active connection
  - Automatically generated JavaScript proxy files
  - Exceptions

SignalR Security Concepts

Authentication and authorization

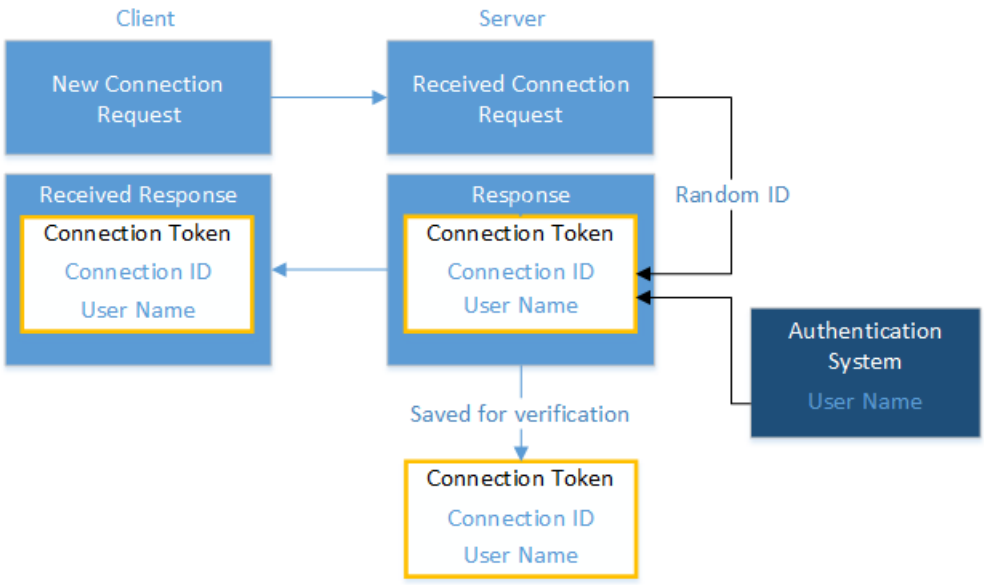
SignalR does not provide any features for authenticating users. Instead, you integrate the SignalR features into the existing authentication structure for an application. You authenticate users as you would normally in your application, and work with the results of the authentication in your SignalR code. For example, you might authenticate your users with ASP.NET forms authentication, and then in your hub, enforce which users or roles are authorized to call a method. In your hub, you can also pass authentication information, such as user name or whether a user belongs to a role, to the client.

SignalR provides the Authorize attribute to specify which users have access to a hub or method. You apply the Authorize attribute to either a hub or particular methods in a hub. Without the Authorize attribute, all public methods on the hub are available to a client that is connected to the hub. For more information about hubs, see Authentication and Authorization for SignalR Hubs.

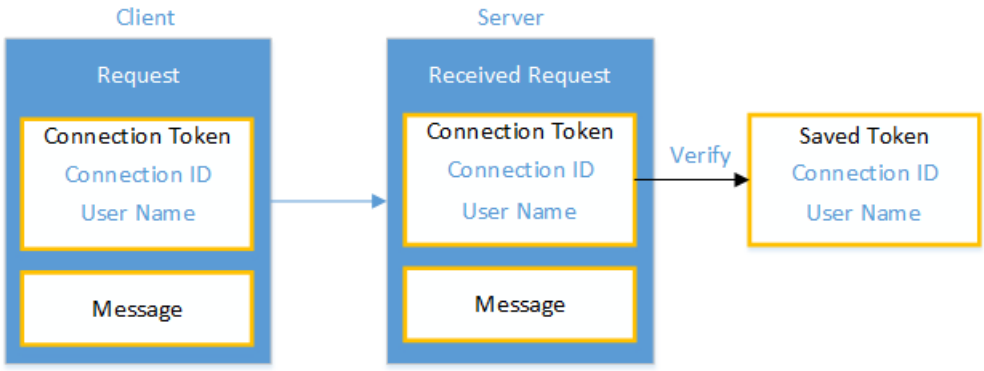
You apply the `Authorize` attribute to hubs, but not persistent connections. To enforce authorization rules when using a `PersistentConnection` you must override the `AuthorizeRequest` method. For more information about persistent connections, see Authentication and Authorization for SignalR Persistent Connections.

Connection token

SignalR mitigates the risk of executing malicious commands by validating the identity of the sender. For each request, the client and server pass a connection token which contains the connection id and username for authenticated users. The connection id uniquely identifies each connected client. The server randomly generates the connection id when a new connection is created, and persists that id for the duration of the connection. The authentication mechanism for the web application provides the username. SignalR uses encryption and a digital signature to protect the connection token.



For each request, the server validates the contents of the token to ensure that the request is coming from the specified user. The username must correspond to the connection id. By validating both the connection id and the username, SignalR prevents a malicious user from easily impersonating another user. If the server cannot validate the connection token, the request fails.



Because the connection id is part of the verification process, you should not reveal one user's connection id to other users or store the value on the client, such as in a cookie.

Connection tokens vs. other token types

Connection tokens are occasionally flagged by security tools because they appear to be session tokens or authentication tokens, which poses a risk if exposed.

SignalR's connection token isn't an authentication token. It is used to confirm that the user making this request is the same one that created the connection. The connection token is necessary because ASP.NET SignalR allows connections to move between servers. The token associates the connection with a particular user but doesn't assert the identity of the user making the request. For a SignalR request to be properly authenticated, it must have some other token that asserts the identity of the user, such as a cookie or bearer token. However, the connection token itself makes no claim that the request was made by that user, only that the connection ID contained within the token is associated with that user.

Since the connection token provides no authentication claim of its own, it isn't considered a "session" or "authentication" token. Taking a given user's connection token and replaying it in a request authenticated as a different user (or an unauthenticated request) will fail, because the user identity of the request and the identity stored in the token won't match.

Rejoining groups when reconnecting

By default, the SignalR application will automatically re-assign a user to the appropriate groups when reconnecting from a temporary disruption, such as when a connection is dropped and re-established before the connection times out. When reconnecting, the client passes a group token that includes the connection id and the assigned groups. The group token is digitally signed and encrypted. The client retains the same connection id after a reconnection; therefore, the connection id passed from the reconnected client must match the previous connection id used by the client. This verification prevents a malicious user from passing requests to join unauthorized groups when reconnecting.

However, it's important to note, that the group token does not expire. If a user belonged to a group in the past, but was banned from that group, that user may be able to mimic a group token that includes the prohibited group. If you need to securely manage which users belong to which groups, you need to store that data on the server, such as in a database. Then, add logic to your application that verifies on the server whether a user belongs to a group. For an example of verifying group membership, see Working with groups.

Automatically rejoining groups only applies when a connection is reconnected after a temporary disruption. If a user disconnects by navigating away from the application or the application restarts, your application must handle how to add that user to the correct groups. For more information, see Working with groups.

How SignalR prevents Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) is an attack where a malicious site sends a request to a vulnerable site where the user is currently logged in. SignalR prevents CSRF by making it extremely unlikely for a malicious site to create a valid request for your SignalR application.

Description of CSRF attack

Here is an example of a CSRF attack:

1. A user logs into www.example.com, using forms authentication.
2. The server authenticates the user. The response from the server includes an authentication cookie.
3. Without logging out, the user visits a malicious web site. This malicious site contains the following HTML form:

   ```
   <h1>You Are a Winner!</h1>
   <form action="http://example.com/api/account" method="post">
       <input type="hidden" name="Transaction" value="withdraw" />
       <input type="hidden" name="Amount" value="1000000" />
       <input type="submit" value="Click Me"/>
   </form>
   ```

   Notice that the form action posts to the vulnerable site, not to the malicious site. This is the "cross-site" part of CSRF.
4. The user clicks the submit button. The browser includes the authentication cookie with the request.
5. The request runs on the example.com server with the user's authentication context, and can do anything that an authenticated user is allowed to do.

Although this example requires the user to click the form button, the malicious page could just as easily run a script that sends an AJAX request to your SignalR application. Moreover, using SSL does not prevent a CSRF attack, because the malicious site can send an "https://" request.

Typically, CSRF attacks are possible against web sites that use cookies for authentication, because browsers send all relevant cookies to the destination web site. However, CSRF attacks are not limited to exploiting cookies. For example, Basic and Digest authentication are also vulnerable. After a user logs in with Basic or Digest authentication, the browser automatically sends the credentials until the session ends.

CSRF mitigations taken by SignalR

SignalR takes the following steps to prevent a malicious site from creating valid requests to your application. SignalR takes these steps by default, you do not need to take any action in your code.

- **Disable cross domain requests** SignalR disables cross domain requests to prevent users from calling a SignalR endpoint from an external domain. SignalR considers any request from an external domain to be invalid and blocks the request. We recommend that you keep this default behavior; otherwise, a malicious site could trick users into sending commands to your site. If you need to use cross domain requests, see How to establish a cross-domain connection .
- **Pass connection token in query string, not cookie** SignalR passes the connection token as a query string value, instead of as a cookie. Storing the connection token in a cookie is unsafe because the browser can inadvertently forward the connection token when malicious code is encountered. Also, passing the connection token in the query string prevents the connection token from persisting beyond the current connection. Therefore, a malicious user cannot make a request under another user's authentication credentials.
- **Verify connection token** As described in the Connection token section, the server knows which connection id is associated with each authenticated user. The server does not process any request from a connection id that does not match the user name. It is unlikely a malicious user could guess a valid request because the malicious user would have to know the user name and the current randomly-generated connection id. That connection id becomes invalid as soon as the connection is ended. Anonymous users should not have access to any sensitive information.

SignalR Security Recommendations

Secure Socket Layers (SSL) protocol

The SSL protocol uses encryption to secure the transport of data between a client and server. If your SignalR application transmits sensitive information between the client and server, use SSL for the transport. For more information about setting up SSL, see How to set up SSL on IIS 7.

Do not use groups as a security mechanism

Groups are a convenient way of collecting related users, but they are not a secure mechanism for limiting access to sensitive information. This is especially true when users can automatically rejoin groups during a reconnect. Instead, consider adding privileged users to a role and limiting access to a hub method to only members of that role. For an example of restricting access based on a role, see Authentication and Authorization for SignalR Hubs. For an example of checking user access to groups when reconnecting, see Working with groups.

Safely handling input from clients

To ensure that a malicious user does not send script to other users, you must encode all input from clients that is intended for broadcast to other clients. You should encode messages on the receiving clients rather than the server, because your SignalR application may have many different types of clients. Therefore, HTML-encoding works for a web client, but not for other types of clients. For example, a web client method to display a chat message would safely handle the user name and message by calling the `html()` function.

```
chat.client.addMessageToPage = function (name, message) {
    // Html encode display name and message.
    var encodedName = $('<div />').text(name).html();
    var encodedMsg = $('<div />').text(message).html();
    // Add the message to the page.
    $('#discussion').append('<li><strong>' + encodedName
        + '</strong>:  ' + encodedMsg + '</li>');
};
```

Reconciling a change in user status with an active connection

If a user's authentication status changes while an active connection exists, the user will receive an error that states, "The user identity cannot change during an active SignalR connection." In that case, your application should re-connect to the server to make sure the connection id and username are coordinated. For example, if your application allows the user to log out while an active connection exists, the username for the connection will no longer match the name that is passed in for the next request. You will want to stop the connection before the user logs out, and then restart it.

However, it is important to note that most applications will not need to manually stop and start the connection. If your application redirects users to a separate page after logging out, such as the default behavior in a Web Forms application or MVC application, or refreshes the current page after logging out, the active connection is automatically disconnected and does not require any additional action.

The following example shows how to stop and start a connection when the user status has changed.

```
<script type="text/javascript">
    $(function () {
        var chat = $.connection.sampleHub;
        $.connection.hub.start().done(function () {
            $('#logoutbutton').click(function () {
                chat.connection.stop();
                $.ajax({
                    url: "Services/SampleWebService.svc/LogOut",
                    type: "POST"
                }).done(function () {
                    chat.connection.start();
                });
            });
        });
    });
</script>
```

Or, the user's authentication status may change if your site uses sliding expiration with Forms Authentication, and there is no activity to keep the authentication cookie valid. In that case, the user will be logged out and the user name will no longer match the user name in the connection token. You can fix this problem by adding some script that periodically requests a resource on the web server to keep the authentication cookie valid. The following example shows how to request a resource every 30 minutes.

```
$(function () {
    setInterval(function() {
        $.ajax({
            url: "Ping.aspx",
            cache: false
        });
    }, 1800000);
});
```

Automatically generated JavaScript proxy files

If you do not want to include all of the hubs and methods in the JavaScript proxy file for each user, you can disable the automatic generation of the file. You might choose this option if you have multiple hubs and methods, but do not want every user to be aware of all of the methods. You disable automatic generation by setting `EnableJavaScriptProxies` to `false`.

```
var hubConfiguration = new HubConfiguration();
hubConfiguration.EnableJavaScriptProxies = false;
app.MapSignalR(hubConfiguration);
```

For more information about the JavaScript proxy files, see The generated proxy and what it does for you.

Exceptions

You should avoid passing exception objects to clients because the objects may expose sensitive information to the clients. Instead, call a method on the client that displays the relevant error message.

```
public Task SampleMethod()
{
    try
    {
        // code that can throw an exception
    }
    catch(Exception e)
    {
        // add code to log exception and take remedial steps

        return Clients.Caller.DisplayError("Sorry, the request could not be processed.");
    }
}
```

Authentication and Authorization for SignalR Hubs

2/20/2020 • 7 minutes to read • Edit Online

by Patrick Fletcher, Tom FitzMacken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This topic describes how to restrict which users or roles can access hub methods.

Software versions used in this topic

- Visual Studio 2013
- .NET 4.5
- SignalR version 2

Previous versions of this topic

For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Overview

This topic contains the following sections:

- Authorize attribute
- Require authentication for all hubs
- Customized authorization
- Pass authentication information to clients
- Authentication options for .NET clients
  - Cookie with Forms Authentication
  - Windows Authentication
  - Connection header
  - Certificate

Authorize attribute

SignalR provides the Authorize attribute to specify which users or roles have access to a hub or method. This attribute is located in the `Microsoft.AspNet.SignalR` namespace. You apply the `Authorize` attribute to either a hub or particular methods in a hub. When you apply the `Authorize` attribute to a hub class, the specified

authorization requirement is applied to all of the methods in the hub. This topic provides examples of the different types of authorization requirements that you can apply. Without the `Authorize` attribute, a connected client can access any public method on the hub.

If you have defined a role named "Admin" in your web application, you could specify that only users in that role can access a hub with the following code.

```
[Authorize(Roles = "Admin")]
public class AdminAuthHub : Hub
{
}
```

Or, you can specify that a hub contains one method that is available to all users, and a second method that is only available to authenticated users, as shown below.

```
public class SampleHub : Hub
{
    public void UnrestrictedSend(string message){ . . . }

    [Authorize]
    public void AuthenticatedSend(string message){ . . . }
}
```

The following examples address different authorization scenarios:

- `[Authorize]` – only authenticated users
- `[Authorize(Roles = "Admin,Manager")]` – only authenticated users in the specified roles
- `[Authorize(Users = "user1,user2")]` – only authenticated users with the specified user names
- `[Authorize(RequireOutgoing=false)]` – only authenticated users can invoke the hub, but calls from the server back to clients are not limited by authorization, such as, when only certain users can send a message but all others can receive the message. The RequireOutgoing property can only be applied to the entire hub, not on individuals methods within the hub. When RequireOutgoing is not set to false, only users that meet the authorization requirement are called from the server.

Require authentication for all hubs

You can require authentication for all hubs and hub methods in your application by calling the RequireAuthentication method when the application starts. You might use this method when you have multiple hubs and want to enforce an authentication requirement for all of them. With this method, you cannot specify requirements for role, user, or outgoing authorization. You can only specify that access to the hub methods is restricted to authenticated users. However, you can still apply the Authorize attribute to hubs or methods to specify additional requirements. Any requirement you specify in an attribute is added to the basic requirement of authentication.

The following example shows a Startup file which restricts all hub methods to authenticated users.

```
public partial class Startup {
    public void Configuration(IAppBuilder app) {
        app.MapSignalR();
        GlobalHost.HubPipeline.RequireAuthentication();
    }
}
```

If you call the `RequireAuthentication()` method after a SignalR request has been processed, SignalR will throw a `InvalidOperationException` exception. SignalR throws this exception because you cannot add a module to the

HubPipeline after the pipeline has been invoked. The previous example shows calling the `RequireAuthentication` method in the `Configuration` method which is executed one time prior to handling the first request.

Customized authorization

If you need to customize how authorization is determined, you can create a class that derives from `AuthorizeAttribute` and override the UserAuthorized method. For each request, SignalR invokes this method to determine whether the user is authorized to complete the request. In the overridden method, you provide the necessary logic for your authorization scenario. The following example shows how to enforce authorization through claims-based identity.

```
[AttributeUsage(AttributeTargets.Class, Inherited = false, AllowMultiple = false)]
public class AuthorizeClaimsAttribute : AuthorizeAttribute
{
    protected override bool UserAuthorized(System.Security.Principal.IPrincipal user)
    {
        if (user == null)
        {
            throw new ArgumentNullException("user");
        }

        var principal = user as ClaimsPrincipal;

        if (principal != null)
        {
            Claim authenticated = principal.FindFirst(ClaimTypes.Authentication);
            if (authenticated != null && authenticated.Value == "true")
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        else
        {
            return false;
        }
    }
}
```

Pass authentication information to clients

You may need to use authentication information in the code that runs on the client. You pass the required information when calling the methods on the client. For example, a chat application method could pass as a parameter the user name of the person posting a message, as shown below.

```
public Task SendChatMessage(string message)
{
    string name;
    var user = Context.User;

    if (user.Identity.IsAuthenticated)
    {
        name = user.Identity.Name;
    }
    else
    {
        name = "anonymous";
    }
    return Clients.All.addMessageToPage(name, message);
}
```

Or, you can create an object to represent the authentication information and pass that object as a parameter, as shown below.

```
public class SampleHub : Hub
{
    public override Task OnConnected()
    {
        return Clients.All.joined(GetAuthInfo());
    }

    protected object GetAuthInfo()
    {
        var user = Context.User;
        return new
        {
            IsAuthenticated = user.Identity.IsAuthenticated,
            IsAdmin = user.IsInRole("Admin"),
            UserName = user.Identity.Name
        };
    }
}
```

You should never pass one client's connection id to other clients, as a malicious user could use it to mimic a request from that client.

Authentication options for .NET clients

When you have a .NET client, such as a console app, which interacts with a hub that is limited to authenticated users, you can pass the authentication credentials in a cookie, the connection header, or a certificate. The examples in this section show how to use those different methods for authenticating a user. They are not fully-functional SignalR apps. For more information about .NET clients with SignalR, see Hubs API Guide - .NET Client.

Cookie

When your .NET client interacts with a hub that uses ASP.NET Forms Authentication, you will need to manually set the authentication cookie on the connection. You add the cookie to the `CookieContainer` property on the HubConnection object. The following example shows a console app that retrieves an authentication cookie from a web page and adds that cookie to the connection.

```
class Program
{
    static void Main(string[] args)
    {
        var connection = new HubConnection("http://www.contoso.com/");
        Cookie returnedCookie;

        Console.Write("Enter user name: ");
        string username = Console.ReadLine();

        Console.Write("Enter password: ");
        string password = Console.ReadLine();

        var authResult = AuthenticateUser(username, password, out returnedCookie);

        if (authResult)
        {
            connection.CookieContainer = new CookieContainer();
            connection.CookieContainer.Add(returnedCookie);
            Console.WriteLine("Welcome " + username);
        }
        else
        {
            Console.WriteLine("Login failed");
        }
    }

    private static bool AuthenticateUser(string user, string password, out Cookie authCookie)
    {
        var request = WebRequest.Create("https://www.contoso.com/RemoteLogin") as HttpWebRequest;
        request.Method = "POST";
        request.ContentType = "application/x-www-form-urlencoded";
        request.CookieContainer = new CookieContainer();

        var authCredentials = "UserName=" + user + "&Password=" + password;
        byte[] bytes = System.Text.Encoding.UTF8.GetBytes(authCredentials);
        request.ContentLength = bytes.Length;
        using (var requestStream = request.GetRequestStream())
        {
            requestStream.Write(bytes, 0, bytes.Length);
        }

        using (var response = request.GetResponse() as HttpWebResponse)
        {
            authCookie = response.Cookies[FormsAuthentication.FormsCookieName];
        }

        if (authCookie != null)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

The console app posts the credentials to **www.contoso.com/RemoteLogin** which could refer to an empty page that contains the following code-behind file.

```
using System;
using System.Web.Security;

namespace SignalRWithConsoleChat
{
    public partial class RemoteLogin : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            string username = Request["UserName"];
            string password = Request["Password"];
            bool result = Membership.ValidateUser(username, password);
            if (result)
            {
                FormsAuthentication.SetAuthCookie(username, false);
            }
        }
    }
}
```

Windows authentication

When using Windows authentication, you can pass the current user's credentials by using the DefaultCredentials property. You set the credentials for the connection to the value of the DefaultCredentials.

```
class Program
{
    static void Main(string[] args)
    {
        var connection = new HubConnection("http://www.contoso.com/");
        connection.Credentials = CredentialCache.DefaultCredentials;
        connection.Start().Wait();
    }
}
```

Connection header

If your application is not using cookies, you can pass user information in the connection header. For example, you can pass a token in the connection header.

```
class Program
{
    static void Main(string[] args)
    {
        var connection = new HubConnection("http://www.contoso.com/");
        connection.Headers.Add("myauthtoken", /* token data */);
        connection.Start().Wait();
    }
}
```

Then, in the hub, you would verify the user's token.

Certificate

You can pass a client certificate to verify the user. You add the certificate when creating the connection. The following example shows only how to add a client certificate to the connection; it does not show the full console app. It uses the X509Certificate class which provides several different ways to create the certificate.

```
class Program
{
    static void Main(string[] args)
    {
        var connection = new HubConnection("http://www.contoso.com/");
        connection.AddClientCertificate(X509Certificate.CreateFromCertFile("MyCert.cer"));
        connection.Start().Wait();
    }
}
```

Authentication and Authorization for SignalR Persistent Connections

2/20/2020 • 2 minutes to read • Edit Online

by Patrick Fletcher, Tom FitzMacken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This topic describes how to enforce authorization on a persistent connection. For general information about integrating security into a SignalR application, see Introduction to Security.

Software versions used in this topic

- Visual Studio 2013
- .NET 4.5
- SignalR version 2

Previous versions of this topic

For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Enforce authorization

To enforce authorization rules when using a PersistentConnection you must override the `AuthorizeRequest` method. You cannot use the `Authorize` attribute with persistent connections. The `AuthorizeRequest` method is called by the SignalR Framework before every request to verify that the user is authorized to perform the requested action. The `AuthorizeRequest` method is not called from the client; instead, you authenticate the user through your application's standard authentication mechanism.

The example below shows how to limit requests to authenticated users.

```
public class AuthenticatedConnection : PersistentConnection
{
    protected override bool AuthorizeRequest(IRequest request)
    {
        return request.User.Identity.IsAuthenticated;
    }
}
```

You can add any customized authorization logic in the AuthorizeRequest method; such as, checking whether a user belongs to a particular role.

SignalR Performance

2/20/2020 • 2 minutes to read • Edit Online

How to maximize the performance of SignalR applications.

- SignalR Performance
- Introduction to Scaleout in SignalR
- SignalR Scaleout with Azure Service Bus
- SignalR Scaleout with Redis
- SignalR Scaleout with SQL Server
- SignalR Connection Density Testing with Crank
- Using SignalR Performance Counters in an Azure Web Role

SignalR Performance

2/20/2020 • 10 minutes to read • Edit Online

by Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This topic describes how to design for, measure, and improve performance in a SignalR application.

Software versions used in this topic

- Visual Studio 2013
- .NET 4.5
- SignalR version 2

Previous versions of this topic

For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

For a recent presentation on SignalR performance and scaling, see Scaling the Real-time Web with ASP.NET SignalR.

This topic contains the following sections:

- Design considerations
- Tuning your SignalR server for performance
- Troubleshooting performance issues
- Using SignalR performance counters
- Using other performance counters
- Other resources

Design considerations

This section describes patterns that can be implemented during the design of a SignalR application, to ensure that performance is not being hindered by generating unnecessary network traffic.

Throttling message frequency

Even in an application that sends out messages at a high frequency (such as a realtime gaming application), most applications don't need to send more than a few messages a second. To reduce the amount of traffic that each client generates, a message loop can be implemented that queues and sends out messages no more frequently than a fixed rate (that is, up to a certain number of messages will be sent every second, if there are messages in

that time interval to be sent). For a sample application that throttles messages to a certain rate (from both client and server), see High-Frequency Realtime with SignalR.

Reducing message size

You can reduce the size of a SignalR message by reducing the size of your serialized objects. In server code, if you're sending an object that contains properties that don't need to be transmitted, prevent those properties from being serialized by using the `JsonIgnore` attribute. The names of properties are also stored in the message; the names of properties can be shortened using the `JsonProperty` attribute. The following code sample demonstrates how to exclude a property from being sent to the client, and how to shorten property names:

.NET server code that demonstrates the JsonIgnore attribute to exclude data from being sent to the client, and the JsonProperty attribute to reduce message size

```
using Newtonsoft.Json;
using System;
public class ShapeModel
{
    [JsonProperty("l")]
    public double Left { get; set; }
    [JsonProperty("t")]
    public double Top { get; set; }
    // We don't want the client to get the "LastUpdatedBy" property
    [JsonIgnore]
    public string LastUpdatedBy { get; set; }
}
```

In order to retain readability/ maintainability in the client code, the abbreviated property names can be remapped to human-friendly names after the message is received. The following code sample demonstrates one possible way of remapping shortened names to longer ones, by defining a message contract (mapping), and using the `reMap` function to apply the contract to the optimized message class:

Client-side JavaScript code that remaps shortened property names to human-readable names

```
function reMap(smallObject, contract) {
    var largeObject = {};
    for (var smallProperty in contract) {
        largeObject[contract[smallProperty]] = smallObject[smallProperty];
    }
    return largeObject;
}
var shapeModelContract = {
    l: "left",
    t: "top"
};
myHub.client.setShape = function (shapeModelSmall) {
    var shapeModel = reMap(shapeModelSmall, shapeModelContract);
    // shapeModelSmall has "l" and "t" properties  but after remapping
    // shapeModel now has "left" and "top" properties
};
```

Names can be shortened in messages from the client to the server as well, using the same method.

Reducing the memory footprint (that is, the amount of memory used for the message) of the message object can also improve performance. For example, if the full range of an `int` is not needed, a `short` or `byte` can be used instead.

Since messages are stored in the message bus in server memory, reducing the size of messages can also address server memory issues.

Tuning your SignalR server for performance

The following configuration settings can be used to tune your server for better performance in a SignalR application. For general information on how to improve performance in an ASP.NET application, see Improving ASP.NET Performance.

SignalR configuration settings

- **DefaultMessageBufferSize**: By default, SignalR retains 1000 messages in memory per hub per connection. If large messages are being used, this may create memory issues which can be alleviated by reducing this value. This setting can be set in the `Application_Start` event handler in an ASP.NET application, or in the `Configuration` method of an OWIN startup class in a self-hosted application. The following sample demonstrates how to reduce this value in order to reduce the memory footprint of your application in order to reduce the amount of server memory used:

  .NET server code in Startup.cs for decreasing default message buffer size

  ```
  public class Startup
  {
      public void Configuration(IAppBuilder app)
      {
          // Any connection or hub wire up and configuration should go here
          GlobalHost.Configuration.DefaultMessageBufferSize = 500;
          app.MapSignalR();
      }
  }
  ```

IIS configuration settings

- **Max concurrent requests per application**: Increasing the number of concurrent IIS requests will increase server resources available for serving requests. The default value is 5000; to increase this setting, execute the following commands in an elevated command prompt:

  ```
  cd %windir%\System32\inetsrv\
  appcmd.exe set config /section:system.webserver/serverRuntime
          /appConcurrentRequestLimit:10000
  ```

- **ApplicationPool QueueLength**: This is the maximum number of requests that Http.sys queues for the application pool. When the queue is full, new requests receive a 503 "Service Unavailable" response. The default value is 1000.

  Shortening the queue length for the worker process in the application pool hosting your application will conserve memory resources. For more information, see Managing, Tuning, and Configuring Application Pools.

ASP.NET configuration settings

This section includes configuration settings that can be set in the `aspnet.config` file. This file is found in one of two locations, depending on platform:

- `%windir%\Microsoft.NET\Framework\v4.0.30319\aspnet.config`
- `%windir%\Microsoft.NET\Framework64\v4.0.30319\aspnet.config`

ASP.NET settings that may improve SignalR performance include the following:

- **Maximum concurrent requests per CPU**: Increasing this setting may alleviate performance bottlenecks. To increase this setting, add the following configuration setting to the `aspnet.config` file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    <system.web>
        <applicationPool maxConcurrentRequestsPerCPU="20000" />
    </system.web>
</configuration>
```

- **Request Queue Limit**: When the total number of connections exceeds the `maxConcurrentRequestsPerCPU` setting, ASP.NET will start throttling requests using a queue. To increase the size of the queue, you can increase the `requestQueueLimit` setting. To do this, add the following configuration setting to the `processModel` node in `config/machine.config` (rather than `aspnet.config`):

```
<processModel autoConfig="false" requestQueueLimit="250000" />
```

Troubleshooting performance issues

This section describes ways to find performance bottlenecks in your application.

Verifying that WebSocket is being used

While SignalR can use a variety of transports for communication between client and server, WebSocket offers a significant performance advantage, and should be used if the client and server support it. To determine if your client and server meet the requirements for WebSocket, see Transports and Fallbacks. To determine what transport is being used in your application, you can use the browser developer tools, and examine the logs to see what transport is being used for the connection. For information on using the browser development tools in Internet Explorer and Chrome, see Transports and Fallbacks.

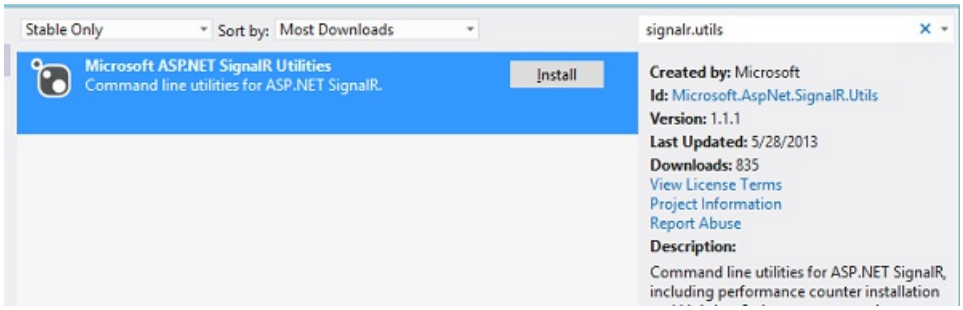
Using SignalR performance counters

This section describes how to enable and use SignalR performance counters, found in the `Microsoft.AspNet.SignalR.Utils` package.

Installing signalr.exe

Performance counters can be added to the server using a utility called SignalR.exe. To install this utility, follow these steps:

1. In Visual Studio, select **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution**
2. Search for **signalr.utils**, and select Install.



3. Accept the license agreement to install the package.
4. SignalR.exe will be installed to `<project folder>/packages/Microsoft.AspNet.SignalR.Utils.<version>/tools`.

Installing performance counters with SignalR.exe

To install SignalR performance counters, run SignalR.exe in an elevated command prompt with the following parameter:

```
SignalR.exe ipc
```

To remove SignalR performance counters, run SignalR.exe in an elevated command prompt with the following parameter:

```
SignalR.exe upc
```

SignalR Performance counters

The utilities package installs the following performance counters. The "Total" counters measure the number of events since the last application pool or server restart.

Connection metrics

The following metrics measure the connection lifetime events that occur. For more information, see Understanding and Handling Connection Lifetime Events.

- **Connections Connected**
- **Connections Reconnected**
- **Connections Disconnected**
- **Connections Current**

Message metrics

The following metrics measure the message traffic generated by SignalR.

- **Connection Messages Received Total**
- **Connection Messages Sent Total**
- **Connection Messages Received/Sec**
- **Connection Messages Sent/Sec**

Message bus metrics

The following metrics measure traffic through the internal SignalR message bus, the queue in which all incoming and outgoing SignalR messages are placed. A message is **Published** when it is sent or broadcast. A **Subscriber** in this context is a subscription on the message bus; this should equal the number of clients plus the server itself. An **Allocated Worker** is a component that sends data to active connections; a **Busy Worker** is one that is actively sending a message.

- **Message Bus Messages Received Total**
- **Message Bus Messages Received/Sec**
- **Message Bus Messages Published Total**
- **Message Bus Messages Published/Sec**
- **Message Bus Subscribers Current**
- **Message Bus Subscribers Total**
- **Message Bus Subscribers/Sec**
- **Message Bus Allocated Workers**
- **Message Bus Busy Workers**
- **Message Bus Topics Current**

Error metrics

The following metrics measure errors generated by SignalR message traffic. **Hub Resolution** errors occur when a hub or hub method cannot be resolved. **Hub Invocation** errors are exceptions thrown while invoking a hub

method. **Transport** errors are connection errors thrown during an HTTP request or response.

- **Errors: All Total**
- **Errors: All/Sec**
- **Errors: Hub Resolution Total**
- **Errors: Hub Resolution/Sec**
- **Errors: Hub Invocation Total**
- **Errors: Hub Invocation/Sec**
- **Errors: Transport Total**
- **Errors: Transport/Sec**

Scaleout metrics

The following metrics measure traffic and errors generated by the scaleout provider. A **Stream** in this context is a scale unit used by the scaleout provider; this is a table if SQL Server is used, a Topic if Service Bus is used, and a Subscription if Redis is used. Each stream ensures ordered read and write operations; a single stream is a potential scale bottleneck, so the number of streams can be increased to help reduce that bottleneck. If multiple streams are used, SignalR will automatically distribute (shard) messages across these streams in a way that ensures messages sent from any given connection are in order.

The MaxQueueLength setting controls the length of the scaleout send queue maintained by SignalR. Setting it to a value greater than 0 will place all messages in a send queue to be sent one at a time to the configured messaging backplane. If the size of the queue goes above the configured length, subsequent calls to send will fail immediately with an InvalidOperationException until the number of messages in the queue is less than the setting again. Queueing is disabled by default because the implemented backplanes generally have their own queuing or flow-control in place. In the case of SQL Server, connection pooling effectively limits the number of sends going on at any one time.

By default, only one stream is used for SQL Server and Redis, five streams are used for Service Bus, and queueing is disabled, but these settings can be changed through configuration on SQL Server and Service Bus:

.NET Server Code for configuring table count and queue length for SQL Server backplane

```
var connectionString = "(your connection string)";
var config = new SqlScaleoutConfiguration(connectionString) {
TableCount = 3,
MaxQueueLength = 50 };
GlobalHost.DependencyResolver.UseSqlServer(config);
```

.NET Server Code for configuring topic count and queue length for Service Bus backplane

```
string connectionString = "(your connection string)";
var config = new ServiceBusScaleoutConfiguration(connectionString, "YourAppName") {
TopicCount = 3,
MaxQueueLength = 50 };
GlobalHost.DependencyResolver.UseServiceBus(config);
```

A **Buffering** stream is one that has entered a faulted state; when the stream is in the faulted state, all messages sent to the backplane will fail immediately until the stream is no longer faulting. The **Send Queue Length** is the number of messages that have been posted but not yet sent.

- **Scaleout Message Bus Messages Received/Sec**
- **Scaleout Streams Total**
- **Scaleout Streams Open**
- **Scaleout Streams Buffering**

- **Scaleout Errors Total**
- **Scaleout Errors/Sec**
- **Scaleout Send Queue Length**

For more information on what these counters are measuring, see SignalR Scaleout with Azure Service Bus.

Using other performance counters

The following performance counters may also be useful in monitoring your application's performance.

Memory

- .NET CLR Memory\# bytes in all Heaps (for w3wp)

ASP.NET

- ASP.NET\Requests Current
- ASP.NET\Queued
- ASP.NET\Rejected

CPU

- Processor Information\Processor Time

TCP/IP

- TCPv6/Connections Established
- TCPv4/Connections Established

Web Service

- Web Service\Current Connections
- Web Service\Maximum Connections

Threading

- .NET CLR Locks And Threads\# of current logical Threads
- .NET CLR Locks And Threads\# of current physical Threads

Other Resources

For more information on ASP.NET performance monitoring and tuning, see the following topics:

- ASP.NET Performance Overview
- ASP.NET Thread Usage on IIS 7.5, IIS 7.0, and IIS 6.0
- <applicationPool> Element (Web Settings)

Introduction to Scaleout in SignalR

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson, Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

Software versions used in this topic

- Visual Studio 2013
- .NET 4.5
- SignalR version 2

Previous versions of this topic

For information about earlier versions of SignalR, see SignalR Older Versions.

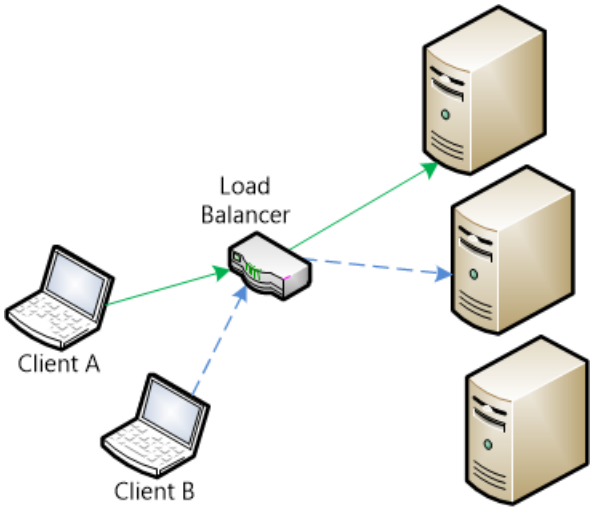
Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

In general, there are two ways to scale a web application: *scale up* and *scale out*.

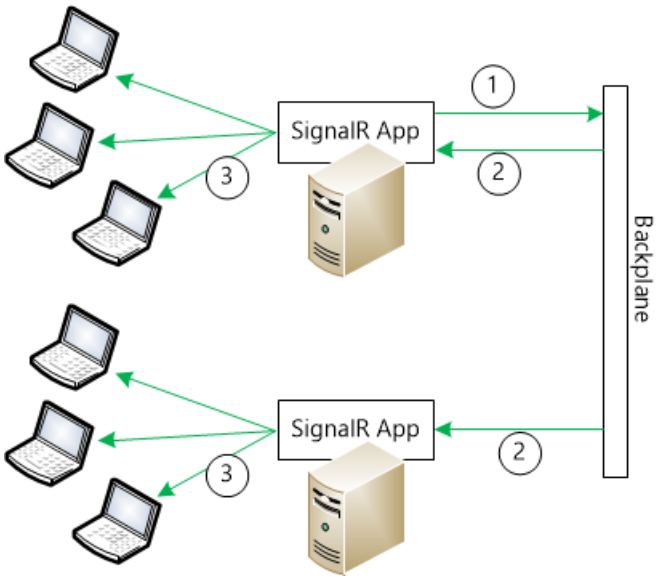
- Scale up means using a larger server (or a larger VM) with more RAM, CPUs, etc.
- Scale out means adding more servers to handle the load.

The problem with scaling up is that you quickly hit a limit on the size of the machine. Beyond that, you need to scale out. However, when you scale out, clients can get routed to different servers. A client that is connected to one server will not receive messages sent from another server.



One solution is to forward messages between servers, using a component called a *backplane*. With a backplane enabled, each application instance sends messages to the backplane, and the backplane forwards them to the

other application instances. (In electronics, a backplane is a group of parallel connectors. By analogy, a SignalR backplane connects multiple servers.)



SignalR currently provides three backplanes:

- **Azure Service Bus**. Service Bus is a messaging infrastructure that allows components to send messages in a loosely coupled way.
- **Redis**. Redis is an in-memory key-value store. Redis supports a publish/subscribe ("pub/sub") pattern for sending messages.
- **SQL Server**. The SQL Server backplane writes messages to SQL tables. The backplane uses Service Broker for efficient messaging. However, it also works if Service Broker is not enabled.

If you deploy your application on Azure, consider using the Redis backplane using Azure Redis Cache. If you are deploying to your own server farm, consider the SQL Server or Redis backplanes.

The following topics contain step-by-step tutorials for each backplane:

- SignalR Scaleout with Azure Service Bus
- SignalR Scaleout with Redis
- SignalR Scaleout with SQL Server

Implementation

In SignalR, every message is sent through a message bus. A message bus implements the IMessageBus interface, which provides a publish/subscribe abstraction. The backplanes work by replacing the default **IMessageBus** with a bus designed for that backplane. For example, the message bus for Redis is RedisMessageBus, and it uses the Redis pub/sub mechanism to send and receive messages.

Each server instance connects to the backplane through the bus. When a message is sent, it goes to the backplane, and the backplane sends it to every server. When a server gets a message from the backplane, it puts the message in its local cache. The server then delivers messages to clients from its local cache.

For each client connection, the client's progress in reading the message stream is tracked using a cursor. (A cursor represents a position in the message stream.) If a client disconnects and then reconnects, it asks the bus for any messages that arrived after the client's cursor value. The same thing happens when a connection uses long polling. After a long poll request completes, the client opens a new connection and asks for messages that arrived after the cursor.

The cursor mechanism works even if a client is routed to a different server on reconnect. The backplane is aware of all the servers, and it doesn't matter which server a client connects to.

Limitations

Using a backplane, the maximum message throughput is lower than it is when clients talk directly to a single server node. That's because the backplane forwards every message to every node, so the backplane can become a bottleneck. Whether this limitation is a problem depends on the application. For example, here are some typical SignalR scenarios:

- Server broadcast (e.g., stock ticker): Backplanes work well for this scenario, because the server controls the rate at which messages are sent.
- Client-to-client (e.g., chat): In this scenario, the backplane might be a bottleneck if the number of messages scales with the number of clients; that is, if the rate of messages grows proportionally as more clients join.
- High-frequency realtime (e.g., real-time games): A backplane is not recommended for this scenario.

Enabling Tracing For SignalR Scaleout

To enable tracing for the backplanes, add the following sections to the web.config file, under the root **configuration** element:

```
<configuration>
  <system.diagnostics>
    <sources>
      <source name="SignalR.SqlMessageBus">
        <listeners>
          <add name="SignalR-Bus" />
        </listeners>
      </source>
      <source name="SignalR.ServiceBusMessageBus">
        <listeners>
          <add name="SignalR-Bus" />
        </listeners>
      </source>
      <source name="SignalR.ScaleoutMessageBus">
        <listeners>
          <add name="SignalR-Bus" />
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="SignalRSwitch" value="Verbose" />
      <!-- Off, Critical, Error, Warning, Information, Verbose -->
    </switches>
    <sharedListeners>
      <add name="SignalR-Bus"
          type="System.Diagnostics.TextWriterTraceListener"
          initializeData="bus.log.txt" />
    </sharedListeners>
    <trace autoflush="true" />
  </system.diagnostics>
  . . .
</configuration>
```

SignalR Scaleout with Azure Service Bus

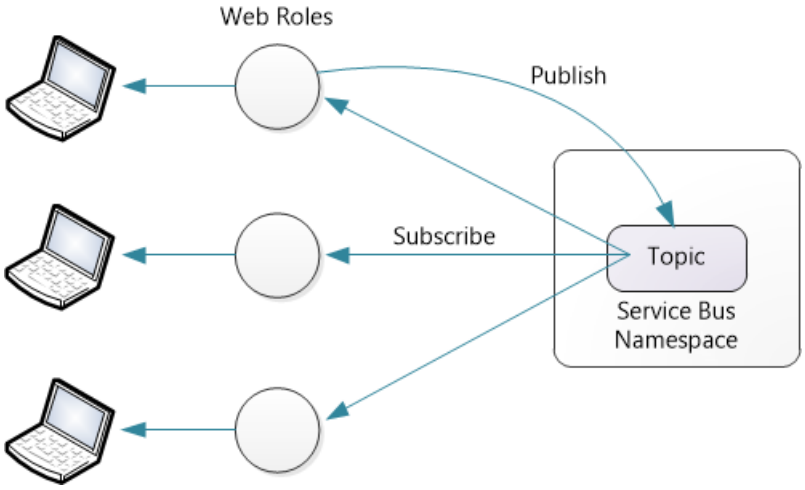
2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson, Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

In this tutorial, you will deploy a SignalR application to a Windows Azure Web Role, using the Service Bus backplane to distribute messages to each role instance. (You can also use the Service Bus backplane with web apps in Azure App Service.)



Prerequisites:

- A Windows Azure account.
- The Windows Azure SDK.
- Visual Studio 2012 or 2013.

The service bus backplane is also compatible with Service Bus for Windows Server, version 1.1. However, it is not compatible with version 1.0 of Service Bus for Windows Server.

Pricing

The Service Bus backplane uses topics to send messages. For the latest pricing information, see Service Bus. At the time of this writing, you can send 1,000,000 messages per month for less than $1. The backplane sends a service bus message for each invocation of a SignalR hub method. There are also some control messages for connections, disconnections, joining or leaving groups, and so forth. In most applications, the majority of the message traffic will be hub method invocations.

Overview

Before we get to the detailed tutorial, here is a quick overview of what you will do.

1. Use the Windows Azure portal to create a new Service Bus namespace.

2. Add these NuGet packages to your application:

- Microsoft.AspNet.SignalR
- Microsoft.AspNet.SignalR.ServiceBus3 or Microsoft.AspNet.SignalR.ServiceBus

3. Create a SignalR application.

4. Add the following code to Startup.cs to configure the backplane:

```
public void Configuration(IAppBuilder app)
{
    string connectionString = "Service Bus connection string";
    GlobalHost.DependencyResolver.UseServiceBus(connectionString, "YourAppName");

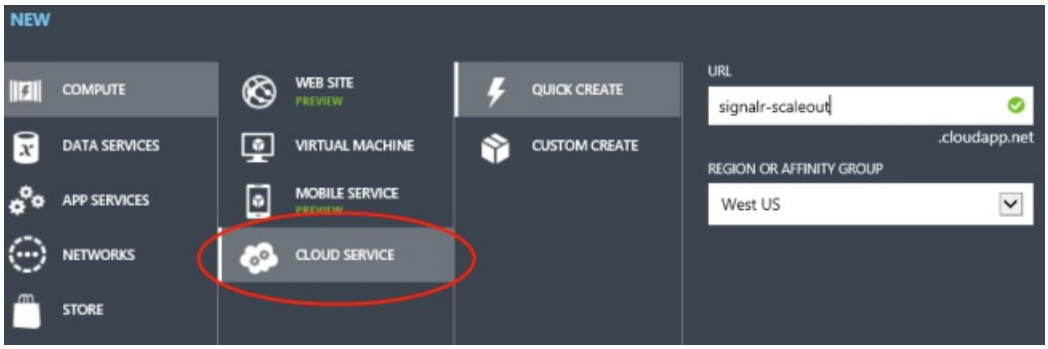
    app.MapSignalR();
    // ...
}
```

This code configures the backplane with the default values for TopicCount and MaxQueueLength. For information on changing these values, see SignalR Performance: Scaleout Metrics.

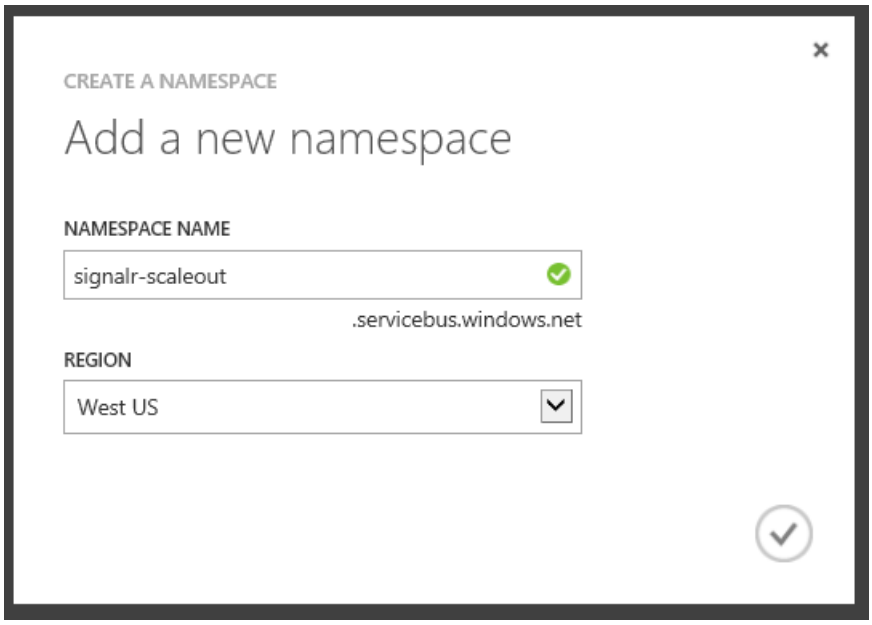
For each application, pick a different value for "YourAppName". Do not use the same value across multiple applications.

Create the Azure Services

Create a Cloud Service, as described in How to Create and Deploy a Cloud Service. Follow the steps in the section "How to: Create a cloud service using Quick Create". For this tutorial, you do not need to upload a certificate.



Create a new Service Bus namespace, as described in How to Use Service Bus Topics/Subscriptions. Follow the steps in the section "Create a Service Namespace".

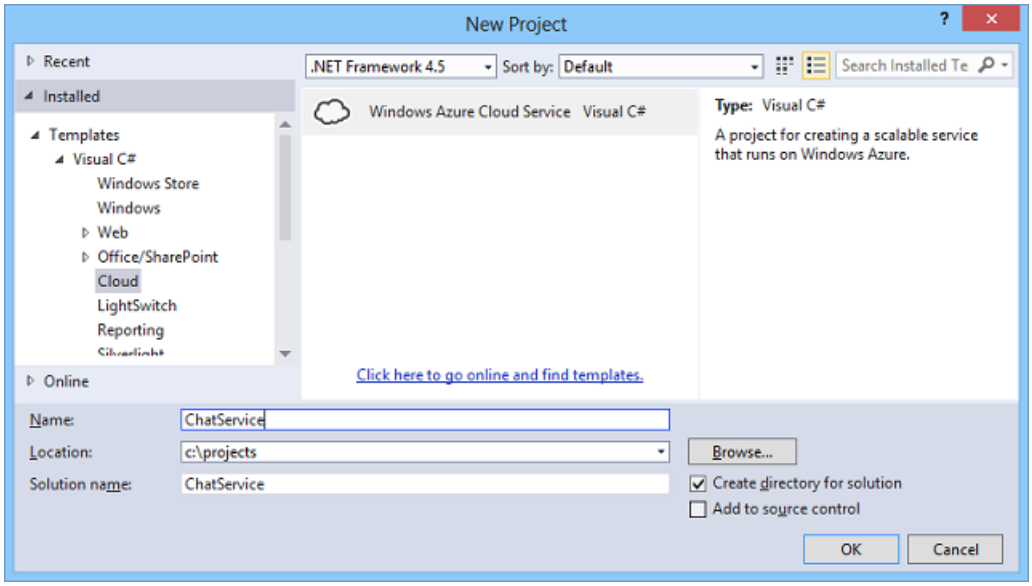


> **NOTE**
>
> Make sure to select the same region for the cloud service and the Service Bus namespace.

Create the Visual Studio Project

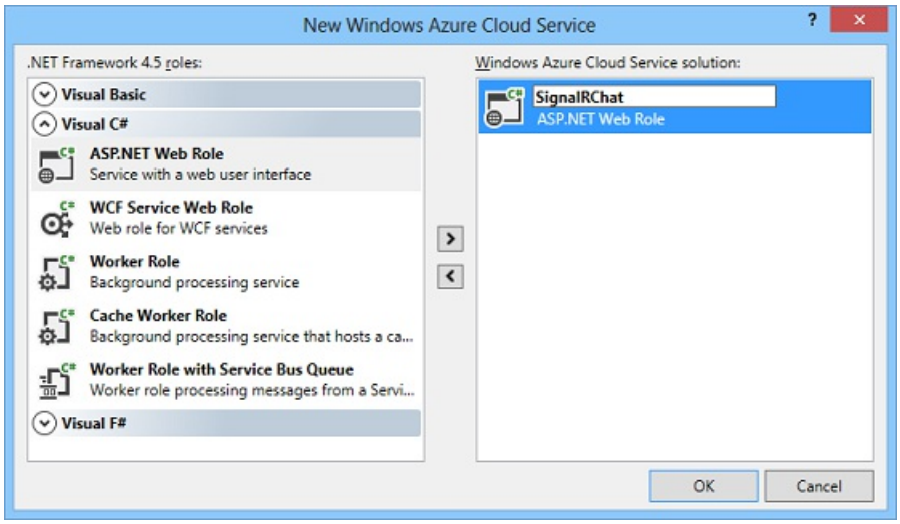
Start Visual Studio. From the **File** menu, click **New Project**.

In the **New Project** dialog box, expand **Visual C#**. Under **Installed Templates**, select **Cloud** and then select **Windows Azure Cloud Service**. Keep the default .NET Framework 4.5. Name the application ChatService and click **OK**.

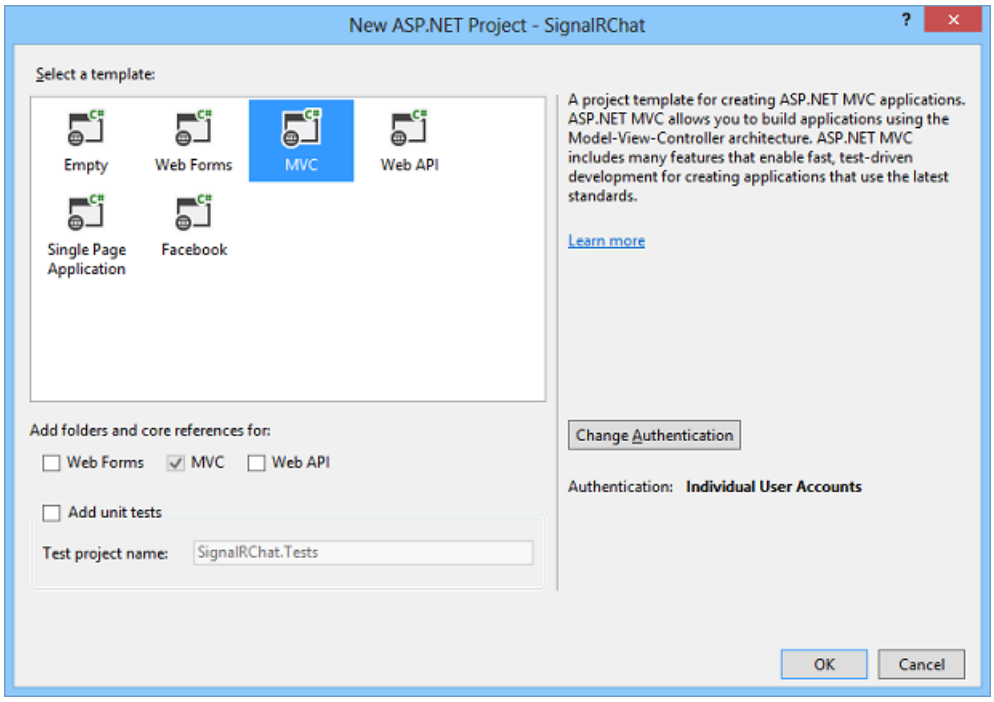


In the **New Windows Azure Cloud Service** dialog, select ASP.NET Web Role. Click the right-arrow button (>) to add the role to your solution.

Hover the mouse over the new role, so the pencil icon visible. Click this icon to rename the role. Name the role "SignalRChat" and click **OK**.



In the **New ASP.NET Project** dialog, select **MVC**, and click OK.



The project wizard creates two projects:

- ChatService: This project is the Windows Azure application. It defines the Azure roles and other configuration options.
- SignalRChat: This project is your ASP.NET MVC 5 project.

Create the SignalR Chat Application

To create the chat application, follow the steps in the tutorial Getting Started with SignalR and MVC 5.

Use NuGet to install the required libraries. From the **Tools** menu, select **NuGet Package Manager**, then select **Package Manager Console**. In the **Package Manager Console** window, enter the following commands:

```
Install-Package -ProjectName SignalRChat Microsoft.AspNet.SignalR
Install-Package -ProjectName SignalRChat Microsoft.AspNet.SignalR.ServiceBus
```

Use the `-ProjectName` option to install the packages to the ASP.NET MVC project, rather than the Windows Azure project.

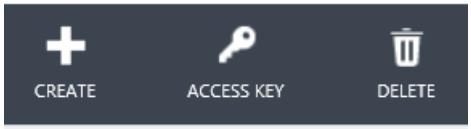
Configure the Backplane

In your application's Startup.cs file, add the following code:

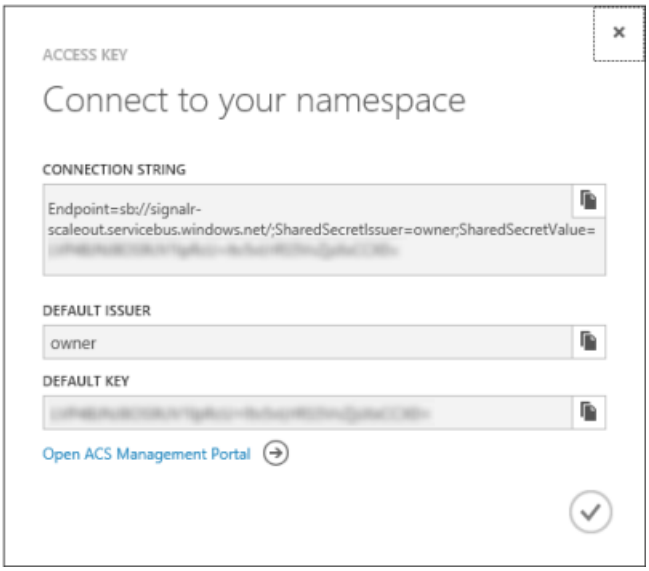
```
public void Configuration(IAppBuilder app)
{
    // Any connection or hub wire up and configuration should go here
    string connectionString = "";
    GlobalHost.DependencyResolver.UseServiceBus(connectionString, "Chat");

    app.MapSignalR();
}
```

Now you need to get your service bus connection string. In the Azure portal, select the service bus namespace that you created and click the Access Key icon.



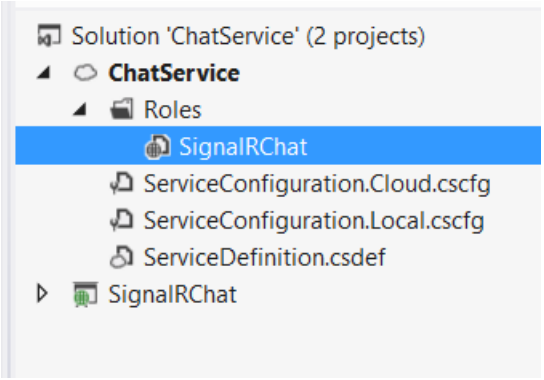
Copy the connection string to the clipboard, then paste it into the *connectionString* variable.



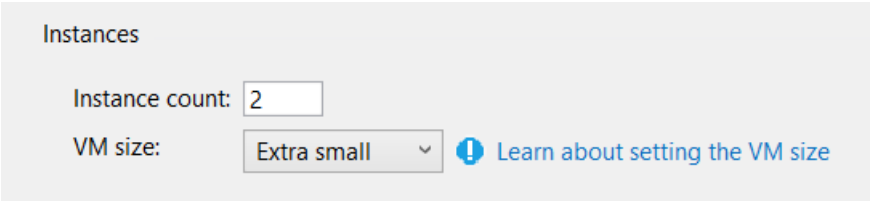

```
string connectionString = "Endpoint=sb://xxxx.servicebus.windows.net/;
SharedSecretIssuer=owner;SharedSecretValue=XXXXXXXX";
```

Deploy to Azure

In Solution Explorer, expand the **Roles** folder inside the ChatService project.

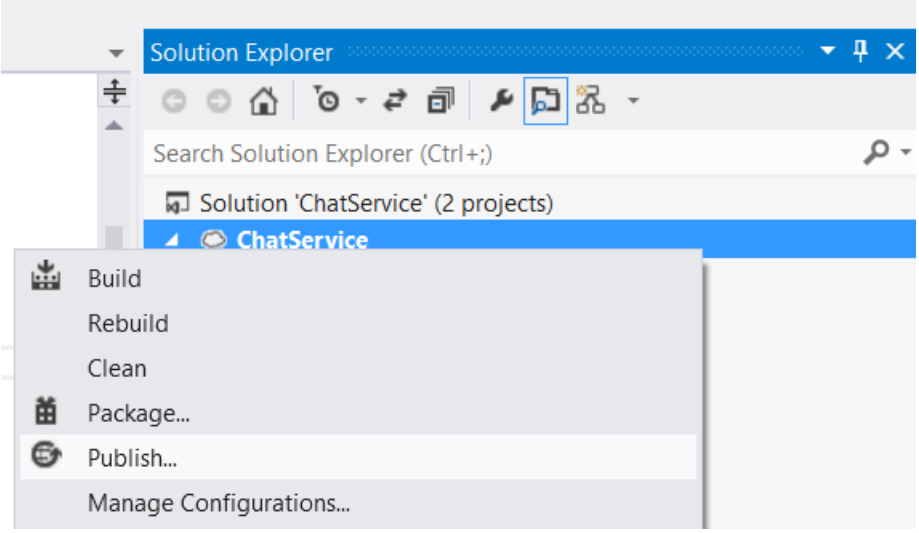


Right-click the SignalRChat role and select **Properties**. Select the **Configuration** tab. Under **Instances** select 2. You can also set the VM size to **Extra Small**.

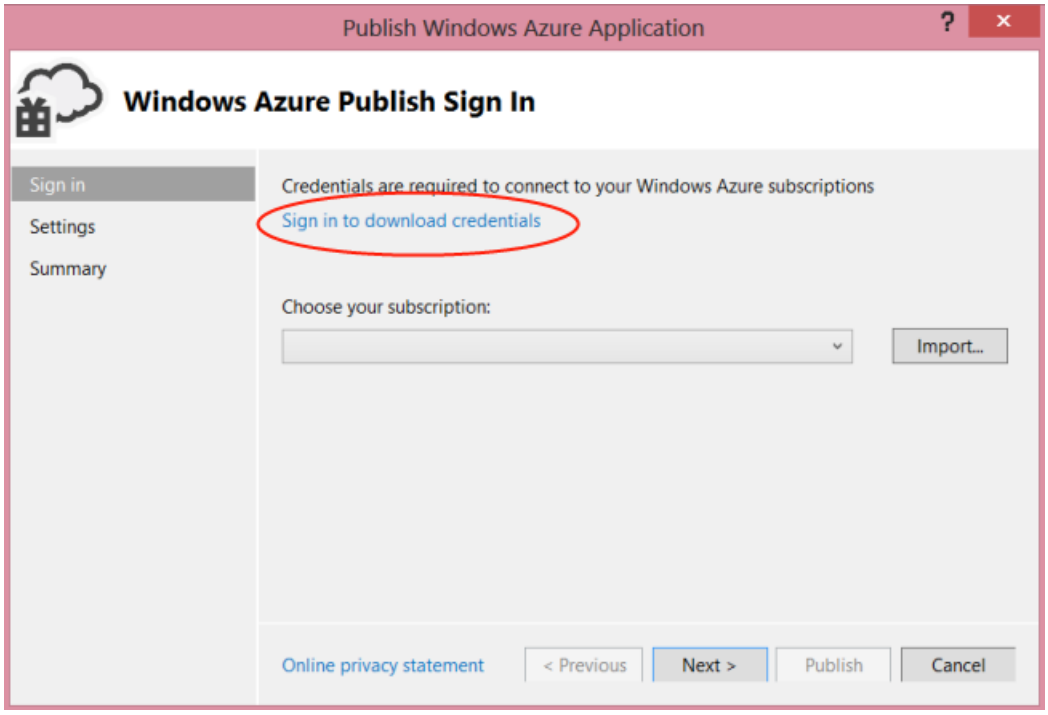


Save the changes.

In Solution Explorer, right-click the ChatService project. Select **Publish**.

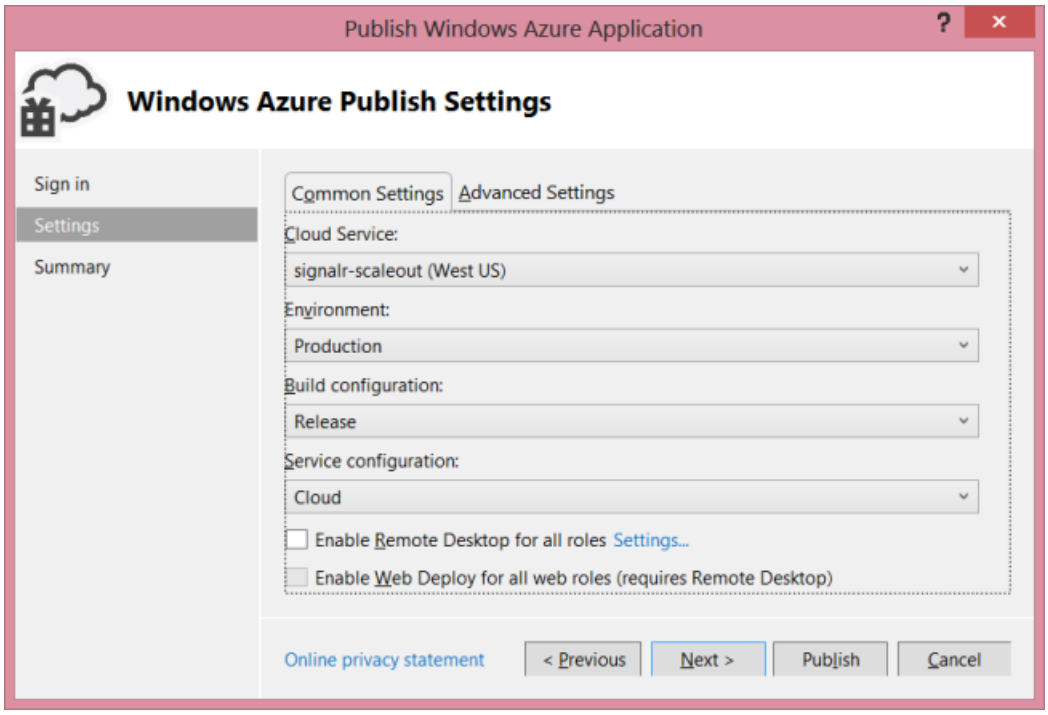


If this is your first time publishing to Windows Azure, you must download your credentials. In the **Publish** wizard, click "Sign in to download credentials". This will prompt you to sign into the Windows Azure portal and download a publish settings file.



Click **Import** and select the publish settings file that you downloaded.

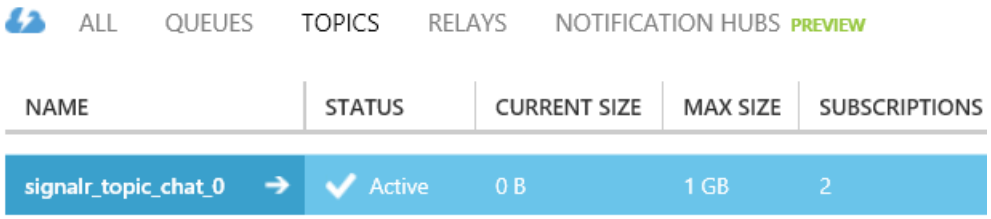
Click **Next**. In the **Publish Settings** dialog, under **Cloud Service**, select the cloud service that you created earlier.



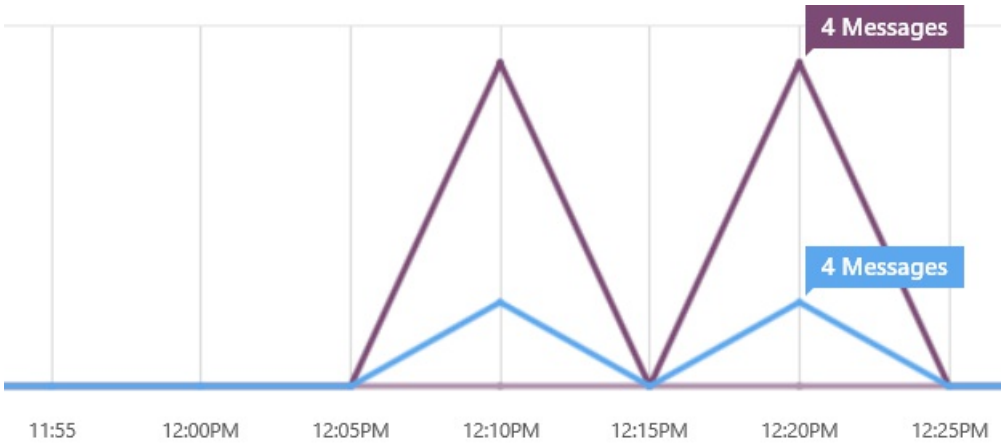
Click **Publish**. It can take a few minutes to deploy the application and start the VMs.

Now when you run the chat application, the role instances communicate through Azure Service Bus, using a Service Bus topic. A topic is a message queue that allows multiple subscribers.

The backplane automatically creates the topic and the subscriptions. To see the subscriptions and message activity, open the Azure portal, select the Service Bus namespace, and click on "Topics".



It make take a few minutes for the message activity to show up in the dashboard.



SignalR manages the topic lifetime. As long as your application is deployed, don't try to manually delete topics or change settings on the topic.

Troubleshooting

System.InvalidOperationException "The only supported IsolationLevel is 'IsolationLevel.Serializable'."

This error can occur if the transaction level for an operation is set to something other than `Serializable`. Verify that no operations are being performed with other transaction levels.

SignalR Scaleout with Redis

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

Software versions used in this topic

- Visual Studio 2013
- .NET 4.5
- SignalR version 2.4

Previous versions of this topic

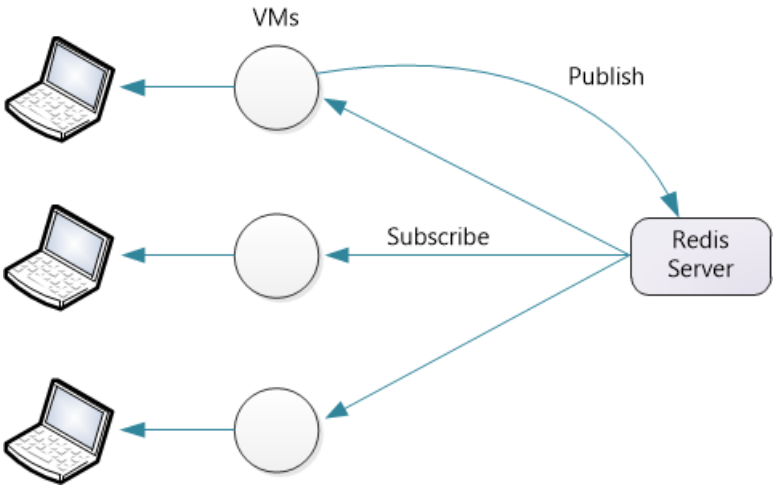
For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

In this tutorial, you will use Redis to distribute messages across a SignalR application that is deployed on two separate IIS instances.

Redis is an in-memory key-value store. It also supports a messaging system with a publish/subscribe model. The SignalR Redis backplane uses the pub/sub feature to forward messages to other servers.



For this tutorial, you will use three servers:

- Two servers running Windows, which you will use to deploy a SignalR application.
- One server running Linux, which you will use to run Redis. For the screenshots in this tutorial, I used Ubuntu 12.04 TLS.

If you don't have three physical servers to use, you can create VMs on Hyper-V. Another option is to create VMs on Azure.

Although this tutorial uses the official Redis implementation, there is also a Windows port of Redis from MSOpenTech. Setup and configuration are different, but otherwise the steps are the same.

> **NOTE**
>
> SignalR scaleout with Redis does not support Redis clusters.

Overview

Before we get to the detailed tutorial, here is a quick overview of what you will do.

1. Install Redis and start the Redis server.
2. Add these NuGet packages to your application:
   - Microsoft.AspNet.SignalR
   - Microsoft.AspNet.SignalR.StackExchangeRedis
3. Create a SignalR application.
4. Add the following code to Startup.cs to configure the backplane:

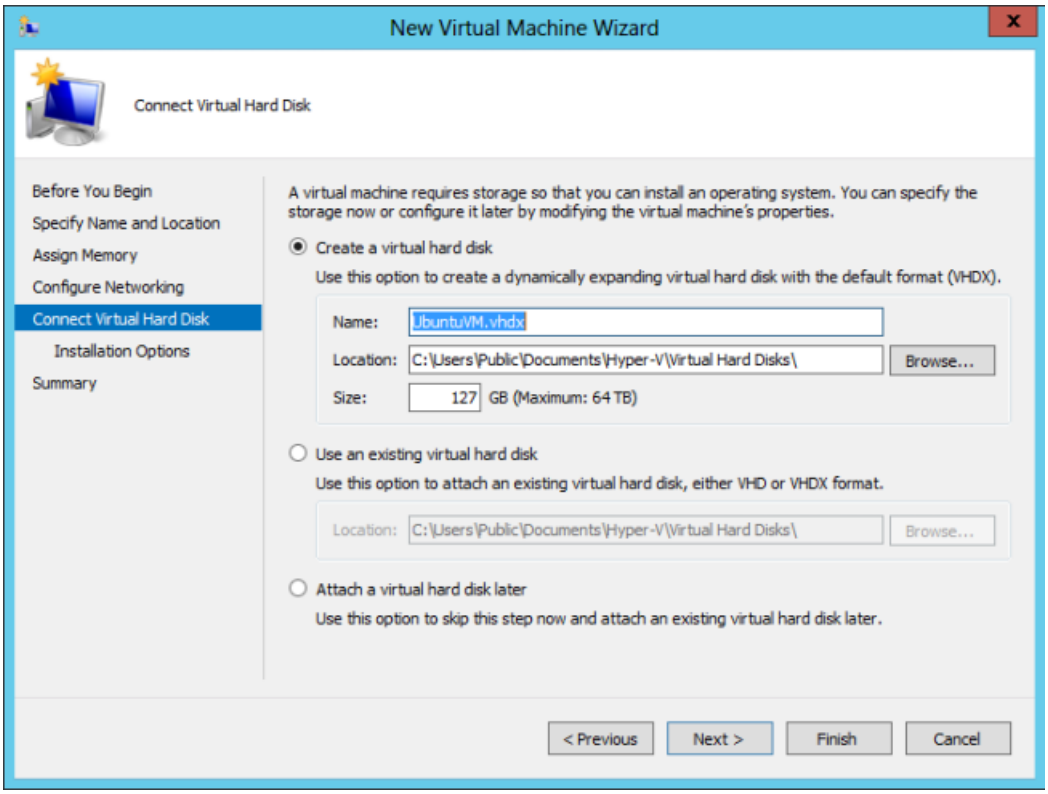
```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        // Any connection or hub wire up and configuration should go here
        GlobalHost.DependencyResolver.UseStackExchangeRedis("server", port, "password", "AppName");
        app.MapSignalR();
    }
}
```

Ubuntu on Hyper-V

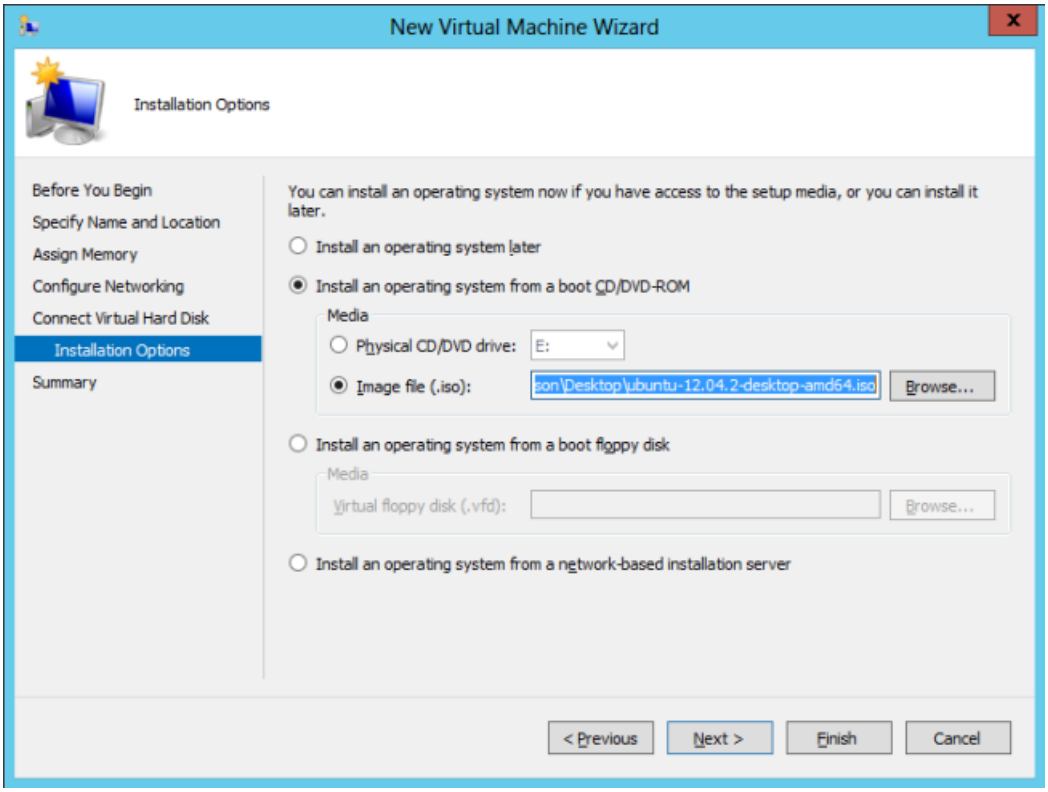
Using Windows Hyper-V, you can easily create an Ubuntu VM on Windows Server.

Download the Ubuntu ISO from http://www.ubuntu.com.

In Hyper-V, add a new VM. In the **Connect Virtual Hard Disk** step, select **Create a virtual hard disk**.



In the **Installation Options** step, select **Image file (.iso)**, click **Browse**, and browse to the Ubuntu installation ISO.



Install Redis

Follow the steps at http://redis.io/download to download and build Redis.

```
wget http://redis.googlecode.com/files/redis-2.6.12.tar.gz
tar xzf redis-2.6.12.tar.gz
cd redis-2.6.12
make
```

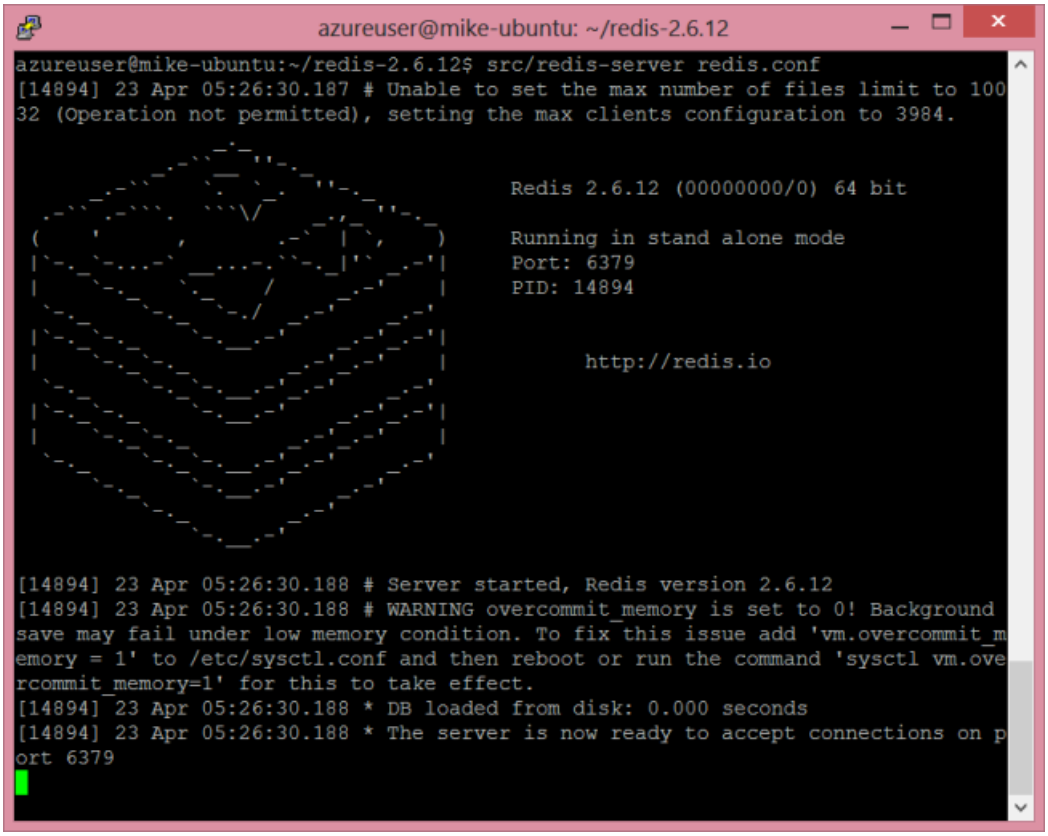
This builds the Redis binaries in the `src` directory.

By default, Redis does not require a password. To set a password, edit the `redis.conf` file, which is located in the root directory of the source code. (Make a backup copy of the file before you edit it!) Add the following directive to `redis.conf`:

```
requirepass YourStrongPassword1234
```

Now start the Redis server:

```
src/redis-server redis.conf
```



Open port 6379, which is the default port that Redis listens on. (You can change the port number in the configuration file.)

Create the SignalR Application

Create a SignalR application by following either of these tutorials:

- Getting Started with SignalR 2.0
- Getting Started with SignalR 2.0 and MVC 5

Next, we'll modify the chat application to support scaleout with Redis. First, add the `Microsoft.AspNet.SignalR.StackExchangeRedis` NuGet package to your project. In Visual Studio, from the **Tools** menu, select **NuGet Package Manager**, then select **Package Manager Console**. In the Package Manager Console window, enter the following command:

```
Install-Package Microsoft.AspNet.SignalR.StackExchangeRedis
```

Next, open the Startup.cs file. Add the following code to the **Configuration** method:

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        // Any connection or hub wire up and configuration should go here
        GlobalHost.DependencyResolver.UseStackExchangeRedis("server", port, "password", "AppName");
        app.MapSignalR();
    }
}
```

- "server" is the name of the server that is running Redis.
- *port* is the port number
- "password" is the password that you defined in the redis.conf file.
- "AppName" is any string. SignalR creates a Redis pub/sub channel with this name.

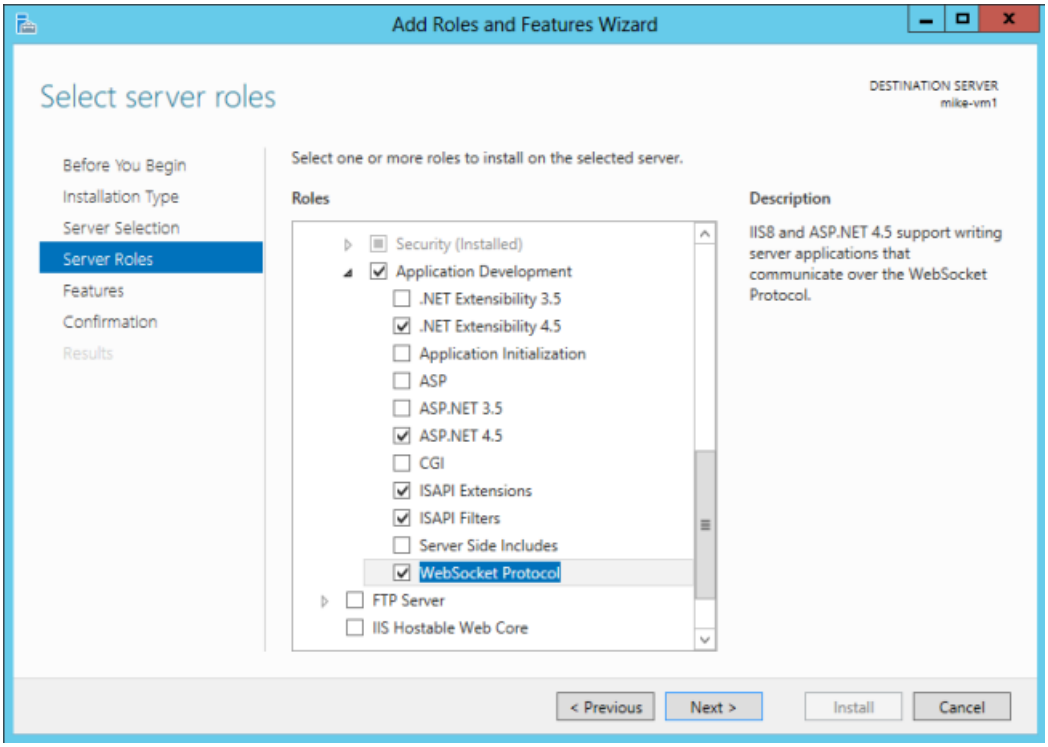
For example:

```
GlobalHost.DependencyResolver.UseStackExchangeRedis("redis-server.cloudapp.net", 6379,
    "MyStrongPassword1234", "ChatApp");
```

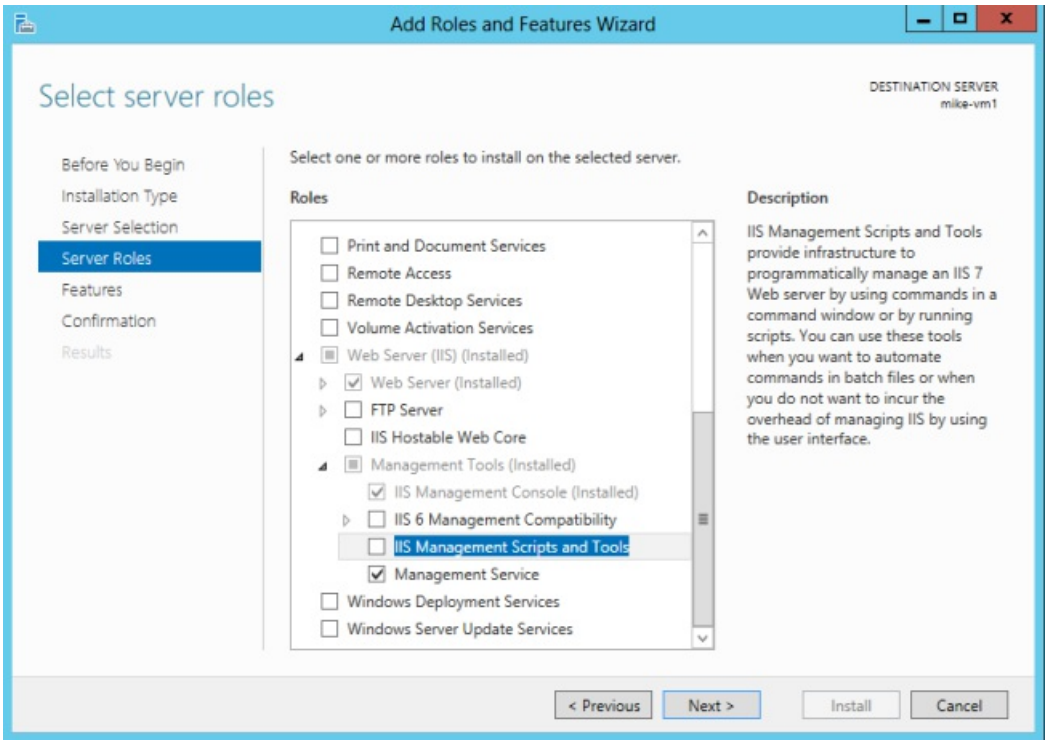
Deploy and Run the Application

Prepare your Windows Server instances to deploy the SignalR application.

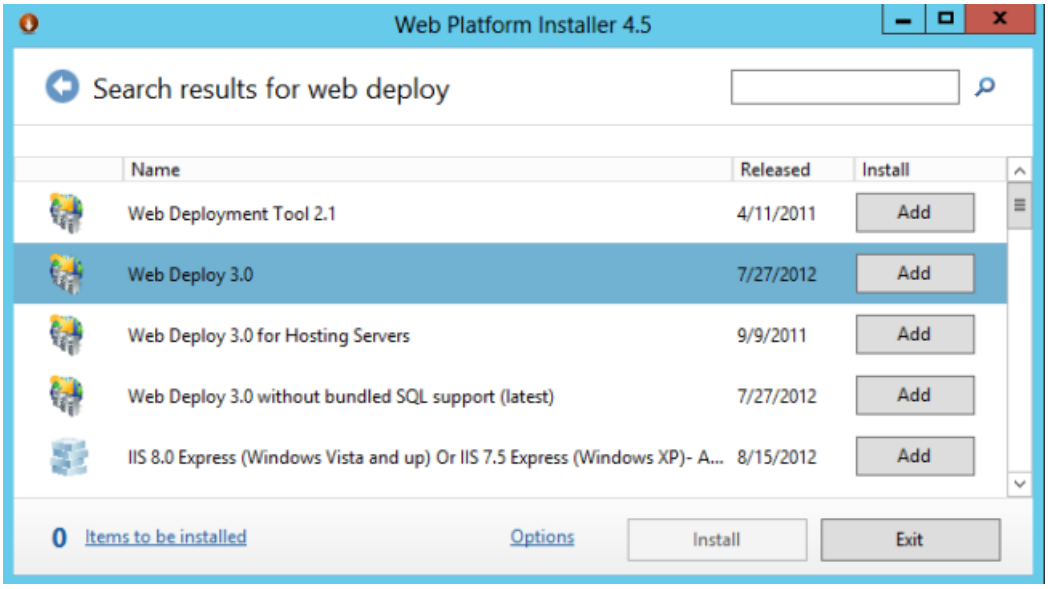
Add the IIS role. Include "Application Development" features, including the WebSocket Protocol.



Also include the Management Service (listed under "Management Tools").



Install Web Deploy 3.0. When you run IIS Manager, it will prompt you to install Microsoft Web Platform, or you can download the installer. In the Platform Installer, search for Web Deploy and install Web Deploy 3.0



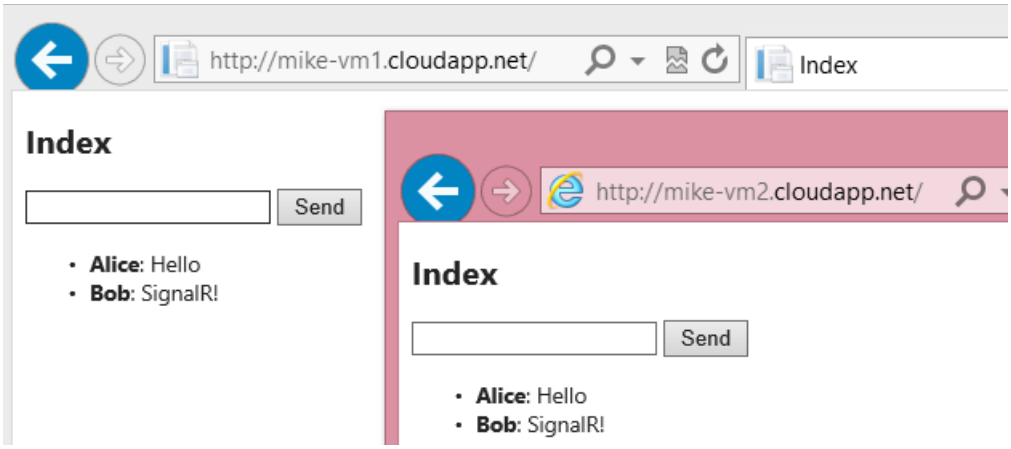
Check that the Web Management Service is running. If not, start the service. (If you don't see Web Management Service in the list of Windows services, make sure that you installed the Management Service when you added the IIS role.)

By default, the Web Management Service listens on TCP port 8172. In Windows Firewall, create a new inbound rule to allow TCP traffic on port 8172. For more information, see Configuring Firewall Rules. (If you are hosting the VMs on Azure, you can do this directly in the Azure portal. See How to Set Up Endpoints to a Virtual Machine.)

Now you are ready to deploy the Visual Studio project from your development machine to the server. In Solution Explorer, right-click the solution and click **Publish**.

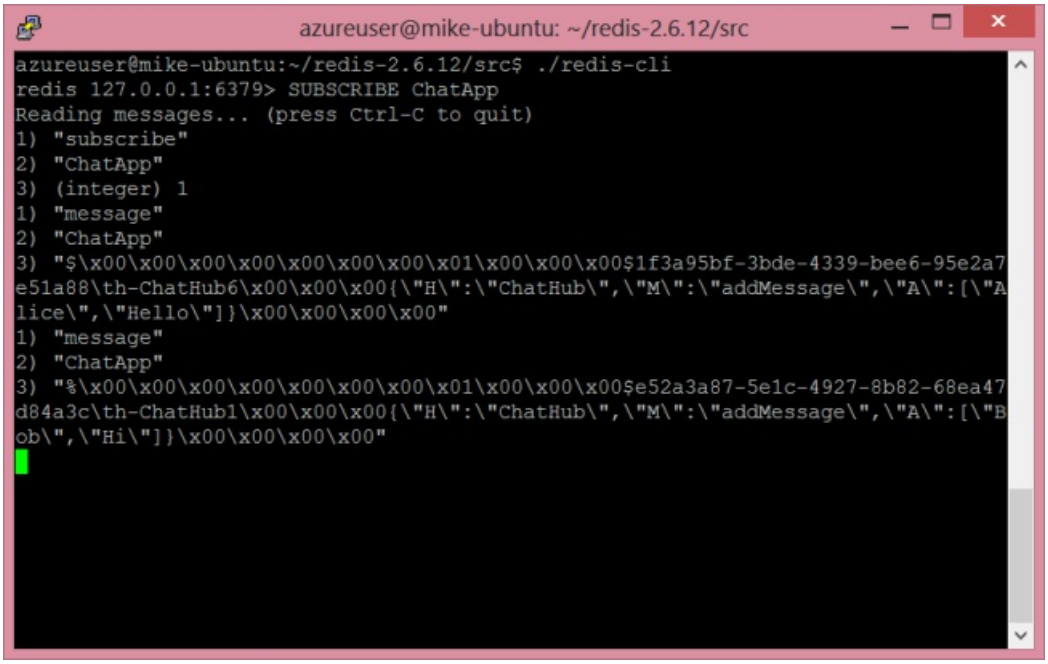
For more detailed documentation about web deployment, see Web Deployment Content Map for Visual Studio and ASP.NET.

If you deploy the application to two servers, you can open each instance in a separate browser window and see that they each receive SignalR messages from the other. (Of course, in a production environment, the two servers would sit behind a load balancer.)



If you're curious to see the messages that are sent to Redis, you can use the **redis-cli** client, which installs with Redis.

```
redis-cli -a password
SUBSCRIBE ChatApp
```



SignalR Scaleout with SQL Server

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson, Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

Software versions used in this topic

- Visual Studio 2013
- .NET 4.5
- SignalR version 2

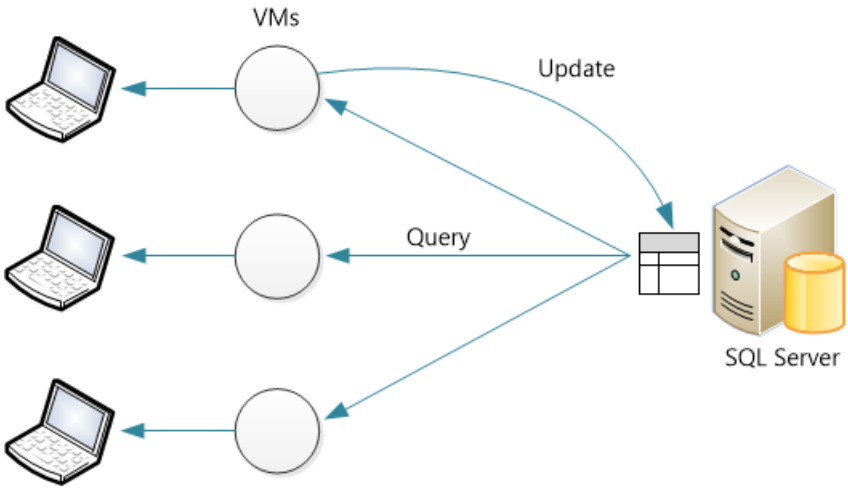
Previous versions of this topic

For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

In this tutorial, you will use SQL Server to distribute messages across a SignalR application that is deployed in two separate IIS instances. You can also run this tutorial on a single test machine, but to get the full effect, you need to deploy the SignalR application to two or more servers. You must also install SQL Server on one of the servers, or on a separate dedicated server. Another option is to run the tutorial using VMs on Azure.



Prerequisites

Microsoft SQL Server 2005 or later. The backplane supports both desktop and server editions of SQL Server. It does not support SQL Server Compact Edition or Azure SQL Database. (If your application is hosted on Azure, consider the Service Bus backplane instead.)

Overview

Before we get to the detailed tutorial, here is a quick overview of what you will do.

1. Create a new empty database. The backplane will create the necessary tables in this database.
2. Add these NuGet packages to your application:
   - Microsoft.AspNet.SignalR
   - Microsoft.AspNet.SignalR.SqlServer
3. Create a SignalR application.
4. Add the following code to Startup.cs to configure the backplane:

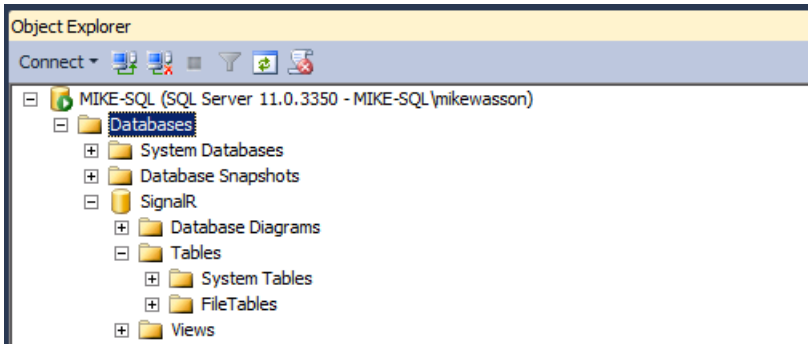
```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        // Any connection or hub wire up and configuration should go here
        string sqlConnectionString = "Connecton string to your SQL DB";
        GlobalHost.DependencyResolver.UseSqlServer(sqlConnectionString);
        app.MapSignalR();
    }
}
```

   This code configures the backplane with the default values for TableCount and MaxQueueLength. For information on changing these values, see SignalR Performance: Scaleout Metrics.

Configure the Database

Decide whether the application will use Windows authentication or SQL Server authentication to access the database. Regardless, make sure the database user has permissions to log in, create schemas, and create tables.

Create a new database for the backplane to use. You can give the database any name. You don't need to create any tables in the database; the backplane will create the necessary tables.



Enable Service Broker

It is recommended to enable Service Broker for the backplane database. Service Broker provides native support for messaging and queuing in SQL Server, which lets the backplane receive updates more efficiently. (However, the backplane also works without Service Broker.)

To check whether Service Broker is enabled, query the **is_broker_enabled** column in the **sys.databases** catalog view.

```
SELECT [name], [service_broker_guid], [is_broker_enabled]
FROM [master].[sys].[databases]
```

SQLQuery1.sql - MIK...QL\mikewasson (58))*

```
/****** Script for SelectTopNRows command from SSMS  ***
SELECT TOP 1000 [name]
      ,[service_broker_guid]
      ,[is_broker_enabled]
  FROM [master].[sys].[databases]
```

100 %

Results | Messages

	name	service_broker_guid	is_broker_enabled
1	master	00000000-0000-0000-0000-000000000000	0
2	tempdb	A86F7D5A-CD9E-439E-921F-281645FBCB92	1
3	model	00000000-0000-0000-0000-000000000000	0
4	msdb	F45AA830-122E-4297-8765-BC60F488E1C1	1
5	SignalR	DCBA980E-9228-4343-890F-41FA45CCB614	0

To enable Service Broker, use the following SQL query:

```
ALTER DATABASE YOUR_DATABASE SET ENABLE_BROKER
```

> **NOTE**
>
> If this query appears to deadlock, make sure there are no applications connected to the DB.

If you have enabled tracing, the traces will also show whether Service Broker is enabled.

Create a SignalR Application

Create a SignalR application by following either of these tutorials:

- Getting Started with SignalR 2.0
- Getting Started with SignalR 2.0 and MVC 5

Next, we'll modify the chat application to support scaleout with SQL Server. First, add the SignalR.SqlServer NuGet package to your project. In Visual Studio, from the **Tools** menu, select **NuGet Package Manager**, then select **Package Manager Console**. In the Package Manager Console window, enter the following command:

```
Install-Package Microsoft.AspNet.SignalR.SqlServer
```

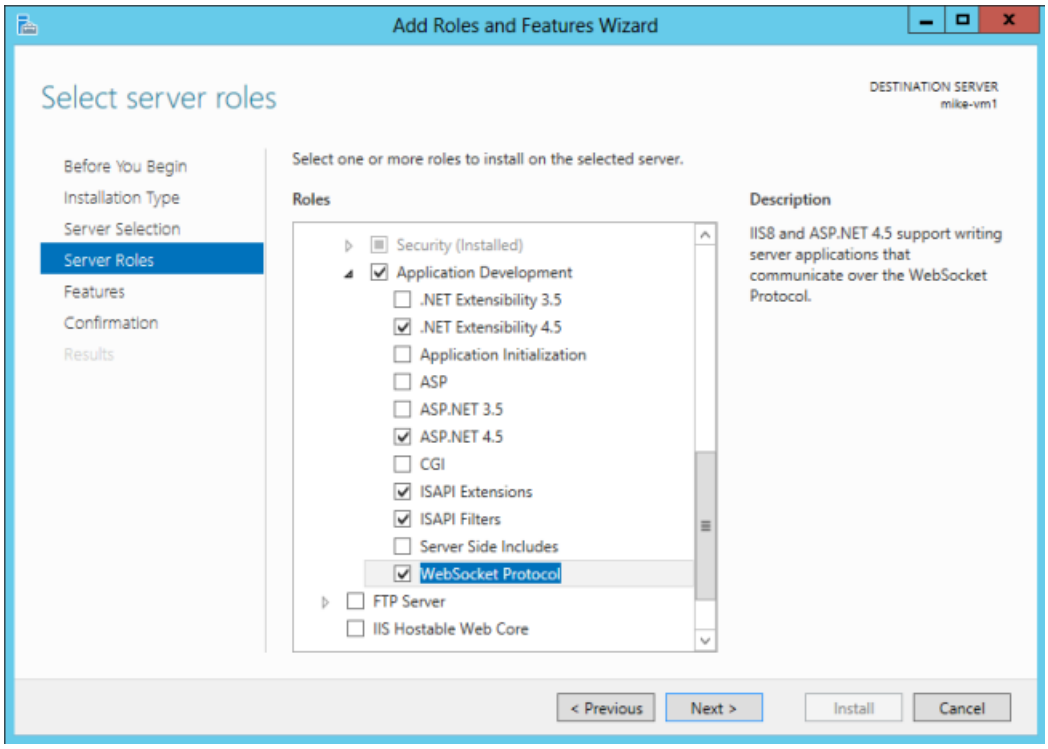
Next, open the Startup.cs file. Add the following code to the **Configure** method:

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        // Any connection or hub wire up and configuration should go here
        string sqlConnectionString = "Connecton string to your SQL DB";
        GlobalHost.DependencyResolver.UseSqlServer(sqlConnectionString);
        app.MapSignalR();
    }
}
```

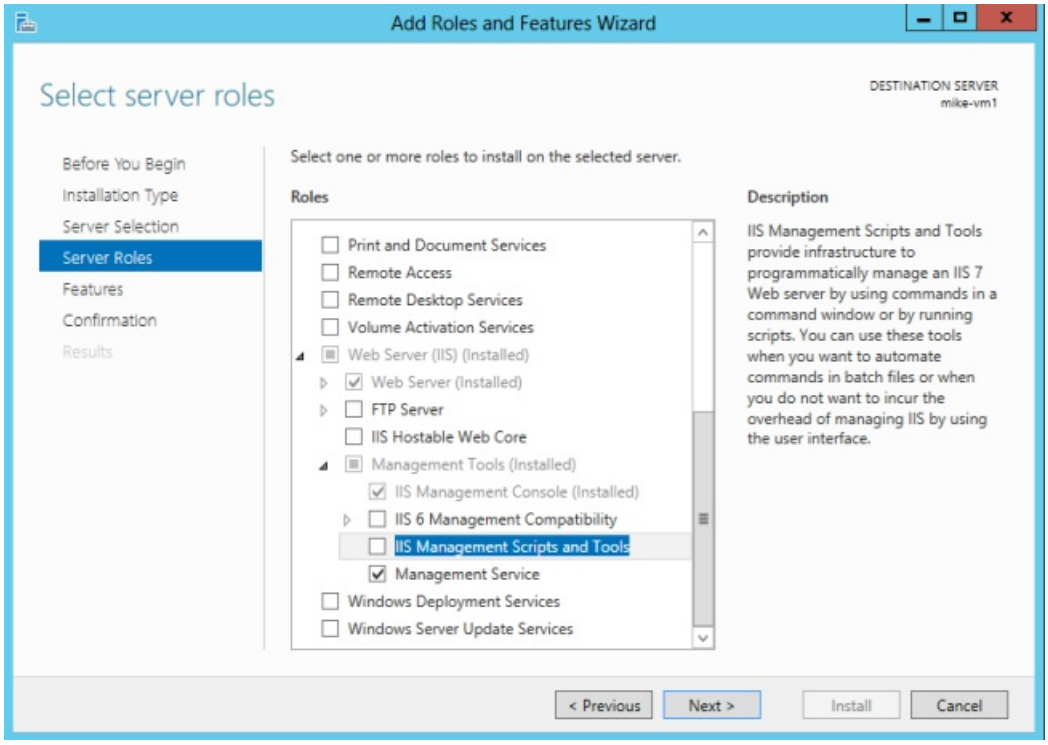
Deploy and Run the Application

Prepare your Windows Server instances to deploy the SignalR application.

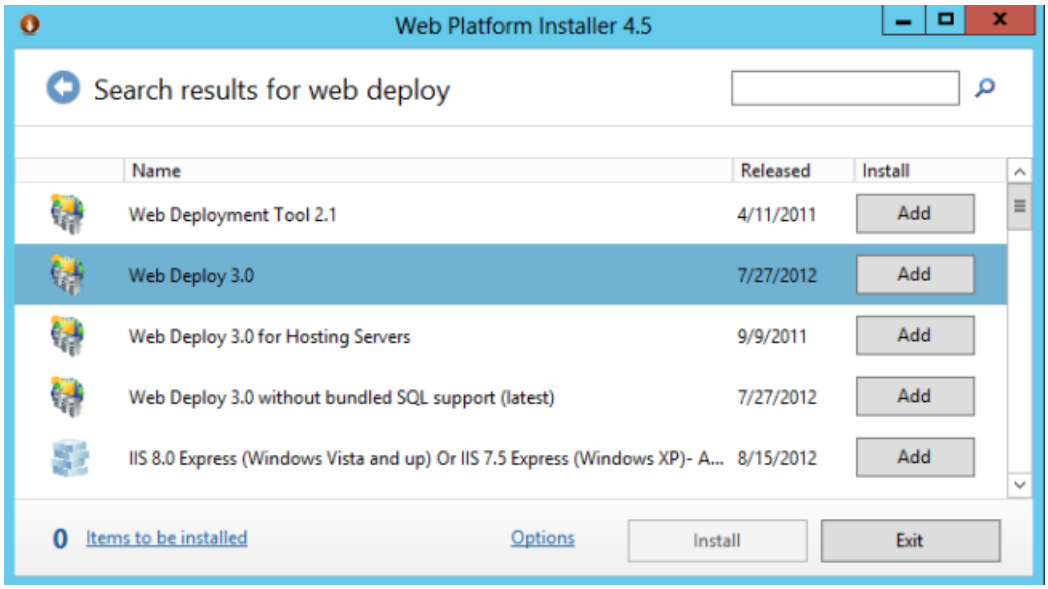
Add the IIS role. Include "Application Development" features, including the WebSocket Protocol.



Also include the Management Service (listed under "Management Tools").



Install Web Deploy 3.0. When you run IIS Manager, it will prompt you to install Microsoft Web Platform, or you can download the installer. In the Platform Installer, search for Web Deploy and install Web Deploy 3.0



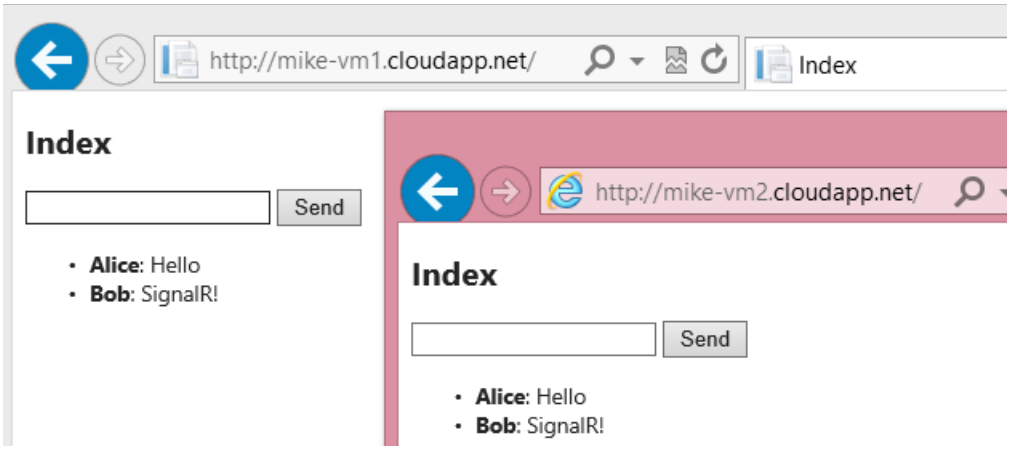
Check that the Web Management Service is running. If not, start the service. (If you don't see Web Management Service in the list of Windows services, make sure that you installed the Management Service when you added the IIS role.)

Finally, open port 8172 for TCP. This is the port that the Web Deploy tool uses.

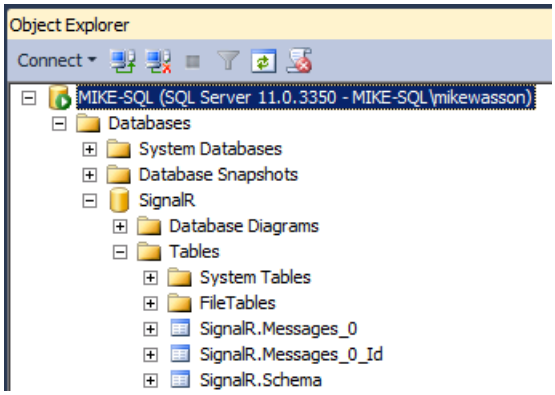
Now you are ready to deploy the Visual Studio project from your development machine to the server. In Solution Explorer, right-click the solution and click **Publish**.

For more detailed documentation about web deployment, see Web Deployment Content Map for Visual Studio and ASP.NET.

If you deploy the application to two servers, you can open each instance in a separate browser window and see that they each receive SignalR messages from the other. (Of course, in a production environment, the two servers would sit behind a load balancer.)



After you run the application, you can see that SignalR has automatically created tables in the database:



SignalR manages the tables. As long as your application is deployed, don't delete rows, modify the table, and so forth.

SignalR Connection Density Testing with Crank

2/20/2020 • 3 minutes to read • Edit Online

by Tom FitzMacken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

> This article describes how to use the Crank tool to test an application with multiple simulated clients.

Once your application is running in its hosting environment (either an Azure web role, IIS, or self-hosted using Owin), you can test application's response to a high level of connection density using the Crank tool. The hosting environment can be an Internet Information Services (IIS) server, an Owin host, or an Azure web role. (Note: Performance counters are not available on Azure App Service Web Apps, so you will not be able to get performance data from a connection density test.)

Connection Density refers to the number of simultaneous TCP connections that can be established on a server. Each TCP connection incurs its own overhead, and opening a large number of idle connections will eventually create a memory bottleneck.

The SignalR codebase includes a load-testing tool called **Crank**. The latest version of Crank can be found in the Dev branch on GitHub. You can download a Zip archive of the Dev branch of the SignalR codebase here.

Crank may be used to fully saturate the server's memory in order to calculate the total number of idle connections possible on the server hardware. Alternatively, you may also use Crank to load test the server under a certain amount of memory pressure, by ramping up connections until a specific count or a specific memory threshold is reached.

When testing, it is important to use remote client(s) to avoid any competition for resources (i.e., TCP connections and memory). Monitor the client(s) to ensure that they are not hitting any bottlenecks that may prevent the server from reaching its full capacity (memory or CPU). You may need to increase the number of clients in order to fully load the server.

Running a Connection Density Test

This section describes the steps needed to run a connection density test on a SignalR application.

1. Download and build the Dev branch of the SignalR codebase. In a command prompt, navigate to <project directory>\src\Microsoft.AspNet.SignalR.Crank\bin\debug.
2. Deploy your application to its intended hosting environment. Make a note of the endpoint that your application uses; for example, in the application created in the Getting Started tutorial, the endpoint is `http://<yourhost>:8080/signalr`.
3. Install SignalR performance counters on the server. If your application is running on Azure, see Using SignalR Performance Counters in an Azure Web Role.

Once you've downloaded and built the codebase, and installed performance counters on your host, the Crank command-line tool can be found in the `src\Microsoft.AspNet.SignalR.Crank\bin\Debug` folder.

Available options for the Crank tool include:

- **/?**: Shows the help screen. The available options are also displayed if the **Url** parameter is omitted.

- **/Url**: The URL for SignalR connections. This parameter is required. For a SignalR application using the default mapping, the path will end in "/signalr".
- **/Transport**: The name of the transport used. The default is `auto`, which will select the best available protocol. Options include `WebSockets`, `ServerSentEvents`, and `LongPolling` (`ForeverFrame` is not an option for Crank, since the .NET client rather than Internet Explorer is used). For more information on how SignalR selects transports, see Transports and Fallbacks.
- **/BatchSize**: The number of clients added in each batch. The default is 50.
- **/ConnectInterval**: The interval in milliseconds between adding connections. The default is 500.
- **/Connections**: The number of connections used to load-test the application. The default is 100,000.
- **/ConnectTimeout**: The timeout in seconds before aborting the test. The default is 300.
- **MinServerMBytes**: The minimum server megabytes to reach. The default is 500.
- **SendBytes**: The size of the payload sent to the server in bytes. The default is 0.
- **SendInterval**: The delay in milliseconds between messages to the server. The default is 500.
- **SendTimeout**: The timeout in milliseconds for messages to the server. The default is 300.
- **ControllerUrl**: The Url where one client will host a controller hub. The default is null (no controller hub). The controller hub is started when the Crank session starts; no further contact between the controller hub and Crank is made.
- **NumClients**: The number of simulated clients to connect to the application. The default is one.
- **Logfile**: The filename for the logfile for the test run. The default is `crank.csv`.
- **SampleInterval**: The time in milliseconds between performance counter samples. The default is 1000.
- **SignalRInstance**: The instance name for the performance counters on the server. The default is to use the client connection state.

Example

The following command will test a site called `pfsignalr` on Azure that hosts an application on port 8080 with a hub named "ControllerHub", using 100 connections.

```
crank /Connections:100 /Url:http://pfsignalr.cloudapp.net:8080/signalr
```

Using SignalR performance counters in an Azure Web Role

9/16/2020 • 9 minutes to read • Edit Online

By Luke Latham

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

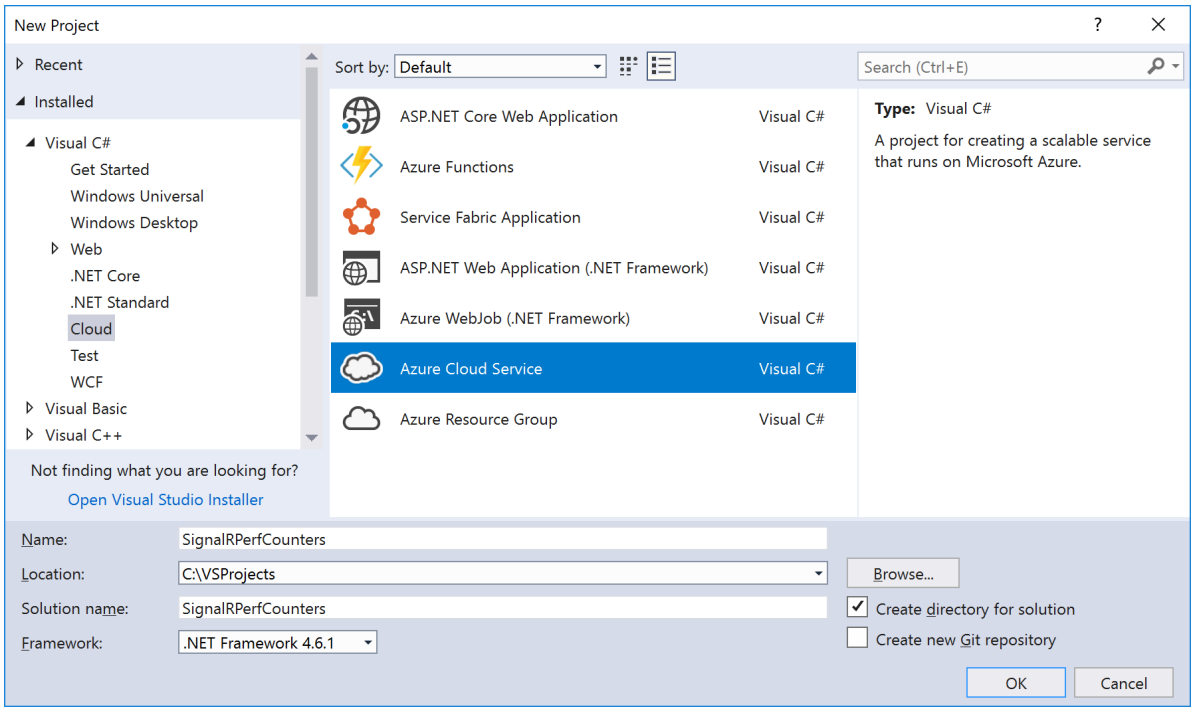
SignalR performance counters are used to monitor your app's performance in an Azure Web Role. The counters are captured by Microsoft Azure Diagnostics. You install SignalR performance counters on Azure with *signalr.exe*, the same tool used for standalone or on-premises apps. Since Azure roles are transient, you configure an app to install and register SignalR performance counters upon startup.

Prerequisites

- Visual Studio 2015 or 2017
- Microsoft Azure SDK for Visual Studio **Note: Restart your machine after installing the SDK.**
- Microsoft Azure subscription: To sign up for a free Azure trial account, see Azure Free Trial.

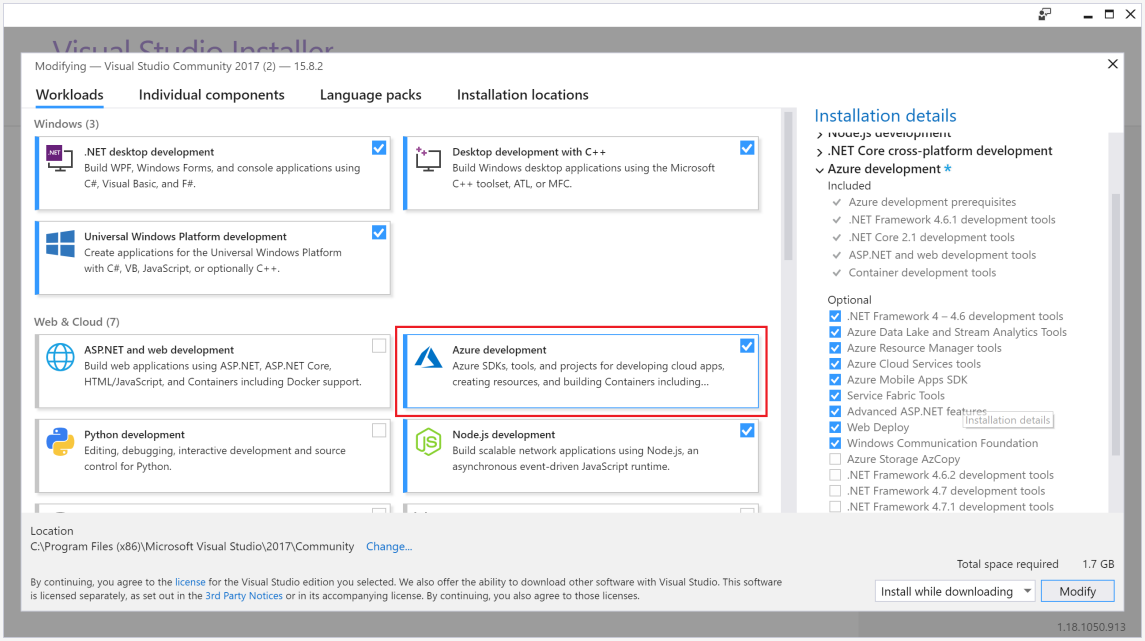
Creating an Azure Web Role application that exposes SignalR performance counters

1. Open Visual Studio.
2. In Visual Studio, select **File** > **New** > **Project**.
3. In the **New Project** dialog box, select the **Visual C#** > **Cloud** category on the left, and then select the **Azure Cloud Service** template. Name the app **SignalRPerfCounters** and select **OK**.

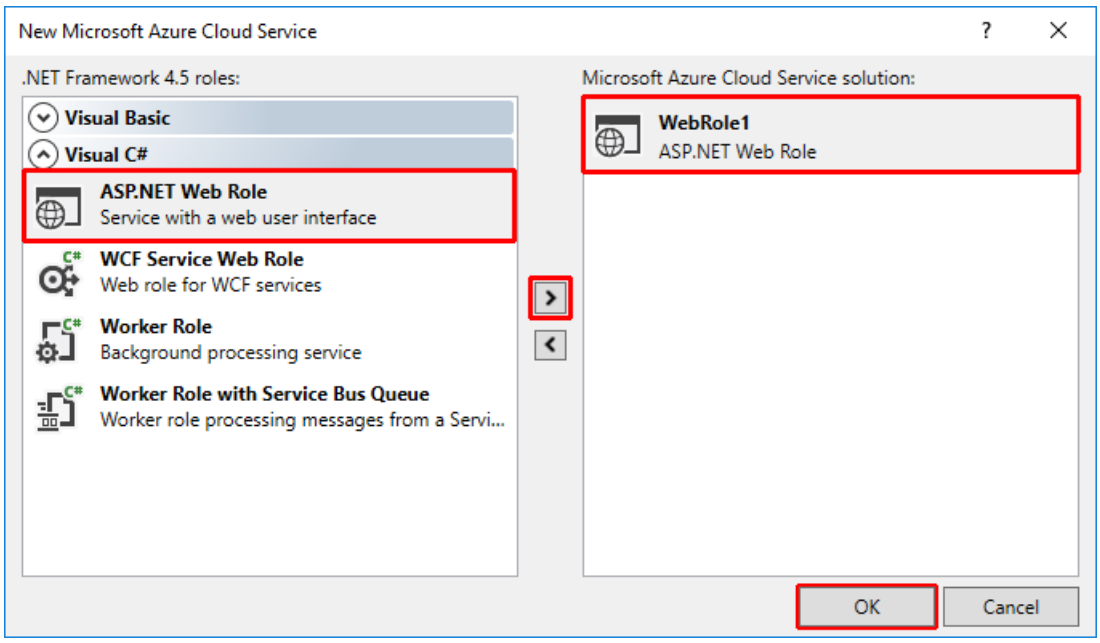


NOTE

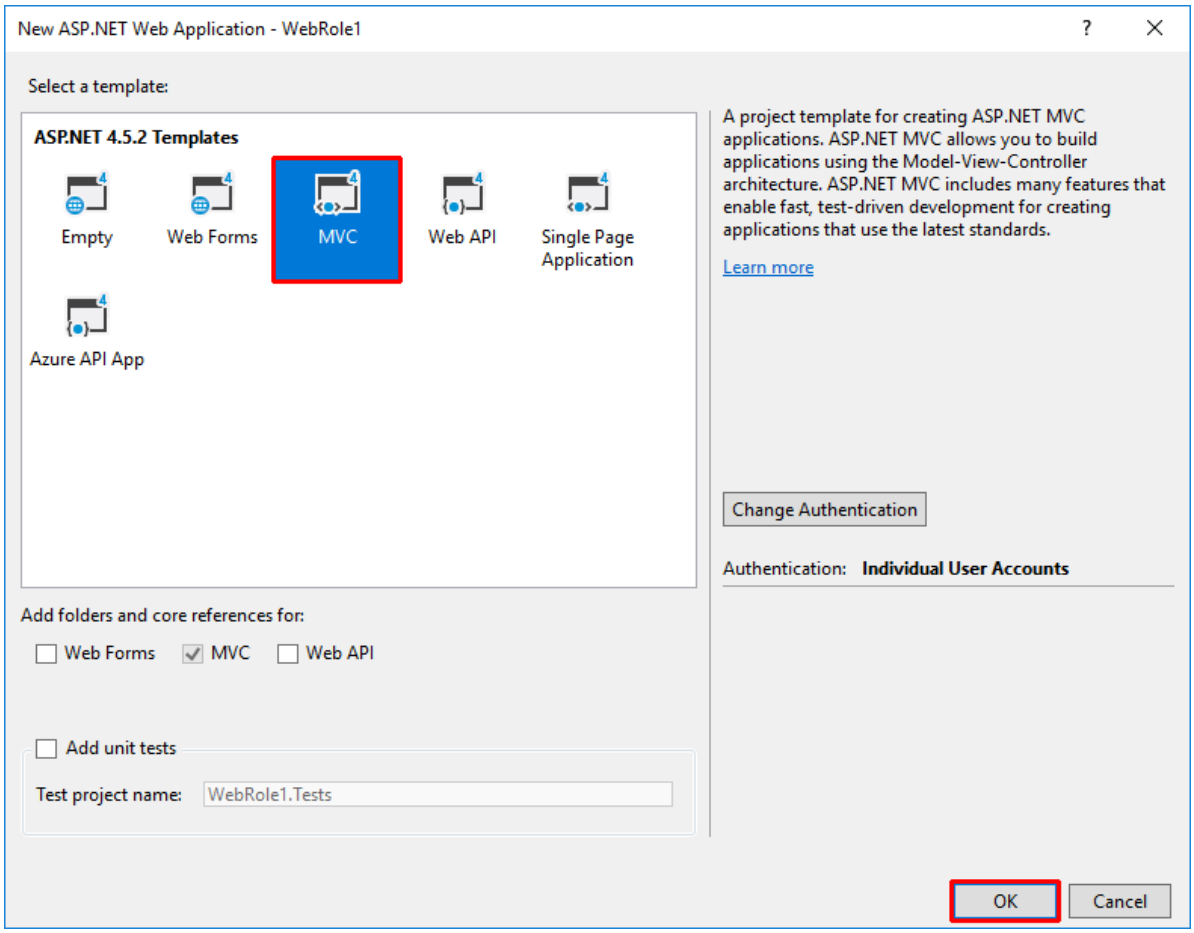
If you don't see the **Cloud** template category or the **Azure Cloud Service** template, you need to install the **Azure development** workload for Visual Studio 2017. Choose the **Open Visual Studio Installer** link on the bottom-left side of the **New Project** dialog to open Visual Studio Installer. Select the **Azure development** workload, and then choose **Modify** to start installing the workload.



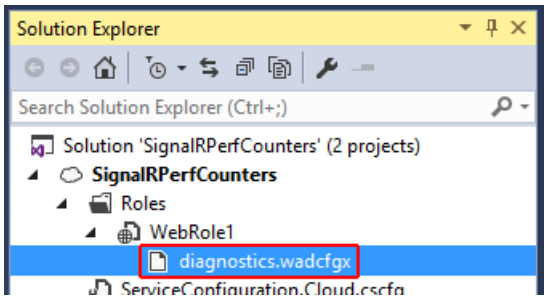
4. In the **New Microsoft Azure Cloud Service** dialog, select **ASP.NET Web Role** and select the > button to add the role to the project. Select **OK**.



5. In the **New ASP.NET Web Application - WebRole1** dialog, select the **MVC** template, and then select **OK**.



6. In **Solution Explorer**, open the *diagnostics.wadcfgx* file under **WebRole1**.



7. Replace the contents of the file with the following configuration and save the file:

```
<?xml version="1.0" encoding="utf-8"?>
<DiagnosticsConfiguration
xmlns="http://schemas.microsoft.com/ServiceHosting/2010/10/DiagnosticsConfiguration">
  <PublicConfig>
    <WadCfg>
      <DiagnosticMonitorConfiguration overallQuotaInMB="4096">
        <DiagnosticInfrastructureLogs scheduledTransferLogLevelFilter="Error" />
        <Logs scheduledTransferPeriod="PT1M" scheduledTransferLogLevelFilter="Error" />
        <Directories scheduledTransferPeriod="PT1M">
          <IISLogs containerName ="wad-iis-logfiles" />
          <FailedRequestLogs containerName ="wad-failedrequestlogs" />
        </Directories>
        <WindowsEventLog scheduledTransferPeriod="PT1M">
          <DataSource name="Application!*[System[(Level=1 or Level=2 or Level=3)]]" />
          <DataSource name="Windows Azure!*[System[(Level=1 or Level=2 or Level=3 or Level=4)]]" />
        </WindowsEventLog>
        <CrashDumps containerName="wad-crashdumps" dumpType="Mini">
          <CrashDumpConfiguration processName="WaIISHost.exe" />
          <CrashDumpConfiguration processName="WaWorkerHost.exe" />
          <CrashDumpConfiguration processName="w3wp.exe" />
        </CrashDumps>
        <PerformanceCounters scheduledTransferPeriod="PT1M">
```

```
          <PerformanceCounterConfiguration counterSpecifier="\Memory\Available MBytes"
sampleRate="PT3M" />
          <PerformanceCounterConfiguration counterSpecifier="\Web Service(_Total)\ISAPI Extension
Requests/sec" sampleRate="PT3M" />
          <PerformanceCounterConfiguration counterSpecifier="\Web Service(_Total)\Bytes Total/Sec"
sampleRate="PT3M" />
          <PerformanceCounterConfiguration counterSpecifier="\ASP.NET
Applications(__Total__)\Requests/Sec" sampleRate="PT3M" />
          <PerformanceCounterConfiguration counterSpecifier="\ASP.NET Applications(__Total__)\Errors
Total/Sec" sampleRate="PT3M" />
          <PerformanceCounterConfiguration counterSpecifier="\ASP.NET\Requests Queued"
sampleRate="PT3M" />
          <PerformanceCounterConfiguration counterSpecifier="\ASP.NET\Requests Rejected"
sampleRate="PT3M" />
          <PerformanceCounterConfiguration counterSpecifier="\Processor(_Total)\% Processor Time"
sampleRate="PT3M" />
          <PerformanceCounterConfiguration counterSpecifier="\.NET CLR Memory(w3wp)\% Time in GC"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\.NET CLR Exceptions(w3wp)\# of Exceps
Thrown / sec" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\.NET CLR LocksAndThreads(w3wp)\# of
current logical Threads" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\.NET CLR LocksAndThreads(w3wp)\# of
current physical Threads" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\.NET CLR LocksAndThreads(w3wp)\Current
Queue Length" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\.NET CLR LocksAndThreads(w3wp)\Contention
Rate / sec" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\.NET CLR Memory(w3wp)\# Bytes in all
Heaps" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\.NET CLR Memory(w3wp)\# GC Handles"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\.NET CLR Memory(w3wp)\# of Pinned
Objects" sampleRate="PT10S" />

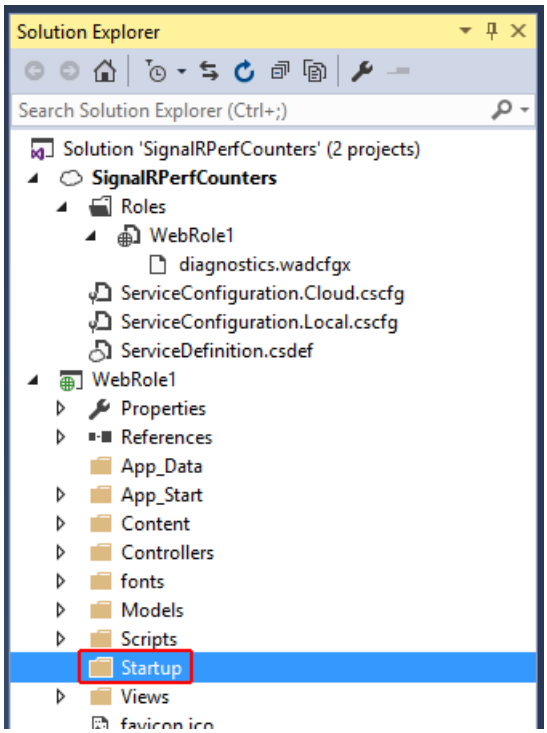
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Connections Connected"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Connections Reconnected"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Connections Disconnected"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Connections Current"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Connection Messages Received
Total" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Connection Messages Sent
Total" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Connection Messages
Received/Sec" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Connection Messages Sent/Sec"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Message Bus Messages Received
Total" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Message Bus Messages
Received/Sec" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Scaleout Message Bus Messages
Received/Sec" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Message Bus Messages Published
Total" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Message Bus Messages
Published/Sec" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Message Bus Subscribers
Current" sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Message Bus Subscribers Total"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Message Bus Subscribers/Sec"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Message Bus Allocated Workers"
sampleRate="PT10S" />
```

```
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Message Bus Busy Workers"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Message Bus Topics Current"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Errors: All Total"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Errors: All/Sec"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Errors: Hub Resolution Total"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Errors: Hub Resolution/Sec"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Errors: Hub Invocation Total"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Errors: Hub Invocation/Sec"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Errors: Tranport Total"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Errors: Transport/Sec"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Scaleout Streams Total"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Scaleout Streams Open"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Scaleout Streams Buffering"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Scaleout Errors Total"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Scaleout Errors/Sec"
sampleRate="PT10S" />
          <PerformanceCounterConfiguration counterSpecifier="\SignalR(*)\Scaleout Send Queue Length"
sampleRate="PT10S" />
        </PerformanceCounters>
      </DiagnosticMonitorConfiguration>
    </WadCfg>
    <StorageAccount></StorageAccount>
  </PublicConfig>
  <PrivateConfig>
    <StorageAccount name="" key="" endpoint="" />
  </PrivateConfig>
  <IsEnabled>true</IsEnabled>
</DiagnosticsConfiguration>
```

8. Open the **Package Manager Console** from **Tools** > **NuGet Package Manager**. Enter the following commands to install the latest version of SignalR and the SignalR utilities package:

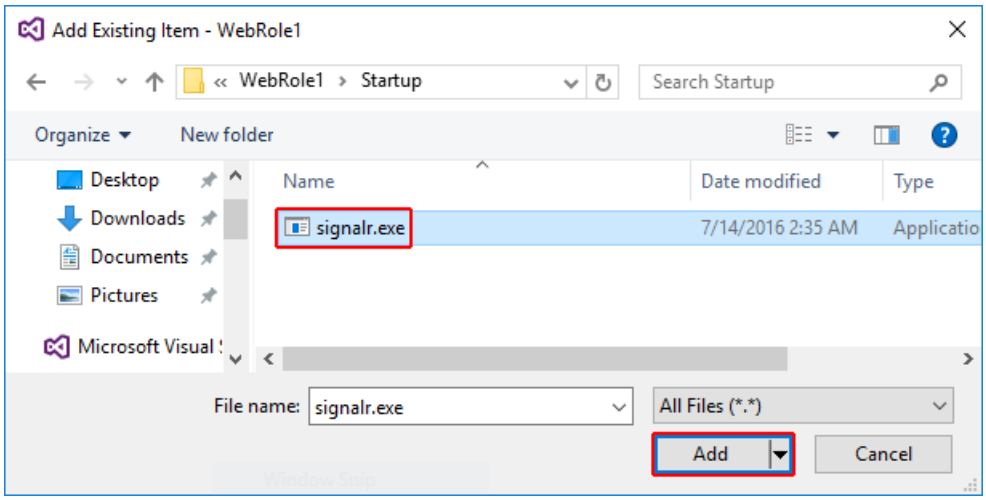
```
install-package microsoft.aspnet.signalr
install-package microsoft.aspnet.signalr.utils
```

9. Configure the app to install the SignalR performance counters into the role instance when it starts up or recycles. In **Solution Explorer**, right-click on the **WebRole1** project and select **Add** > **New Folder**. Name the new folder *Startup*.

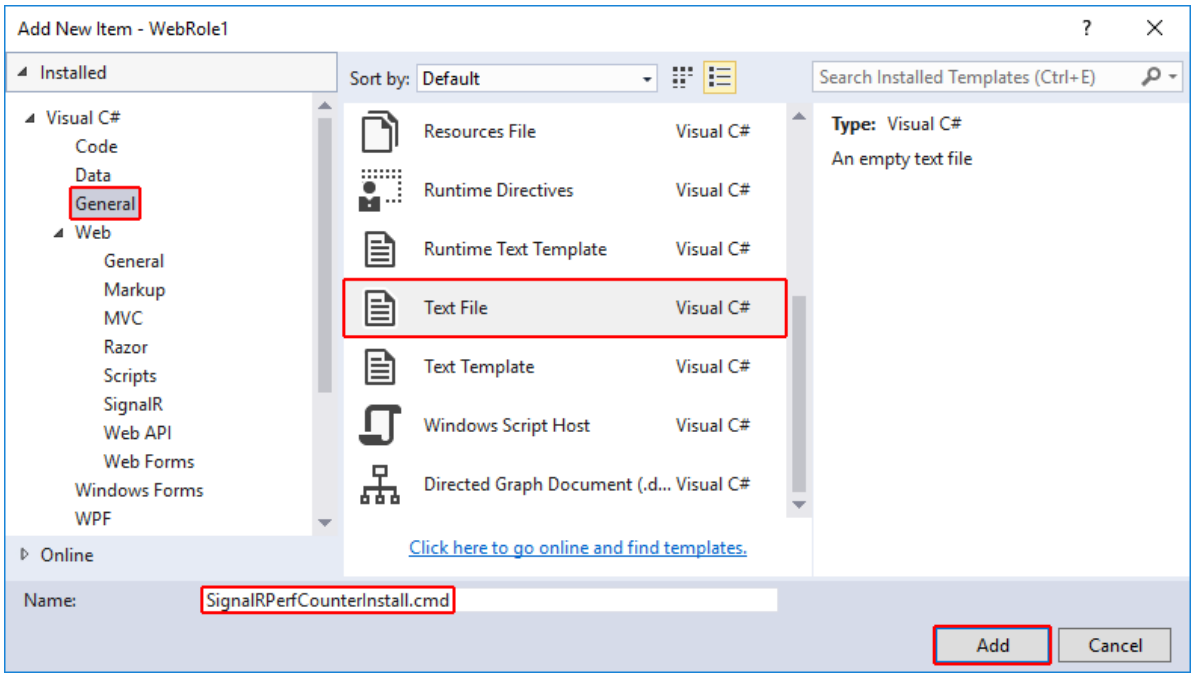


10. Copy the *signalr.exe* file (added with the **Microsoft.AspNet.SignalR.Utils** package) from <project folder>/SignalRPerfCounters/packages/Microsoft.AspNet.SignalR.Utils.<version>/tools to the *Startup* folder you created in the previous step.

11. In **Solution Explorer**, right-click the *Startup* folder and select **Add** > **Existing Item**. In the dialog that appears, select *signalr.exe* and select **Add**.



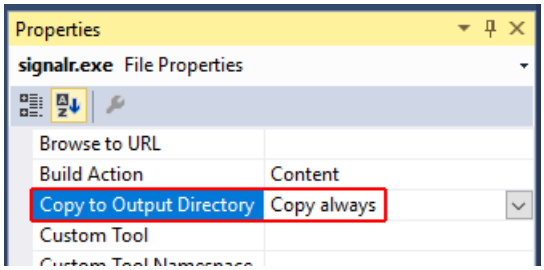
12. Right-click on the *Startup* folder you created. Select **Add** > **New Item**. Select the **General** node, select **Text File**, and name the new item *SignalRPerfCounterInstall.cmd*. This command file will install the SignalR performance counters into the web role.



13. When Visual Studio creates the *SignalRPerfCounterInstall.cmd* file, it will automatically open in the main window. Replace the contents of the file with the following script, then save and close the file. This script executes *signalr.exe*, which adds the SignalR performance counters to the role instance.

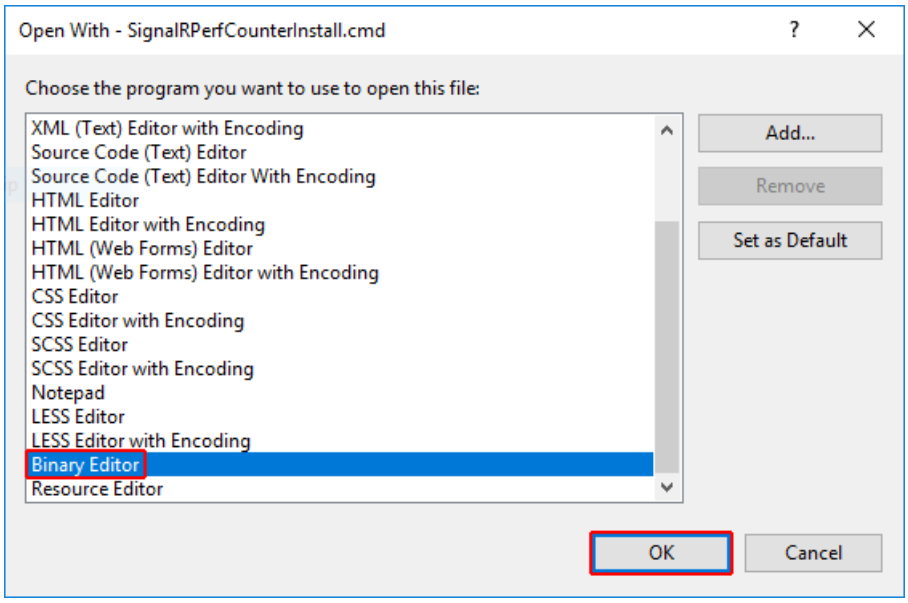
```
SET SignalR_LogDir=%~dp0Log\
MKDIR "%SignalR_LogDir%"
cd %~dp0
signalr.exe ipc >> "%SignalR_LogDir%SignalR_Log.txt" 2>&1
net localgroup "Performance Monitor Users" "Network Service" /ADD >> "%SignalR_LogDir%NetworkAdd.txt"
2>&1
```

14. Select the *signalr.exe* file in **Solution Explorer**. In the file's **Properties**, set **Copy to Output Directory** to **Copy Always**.

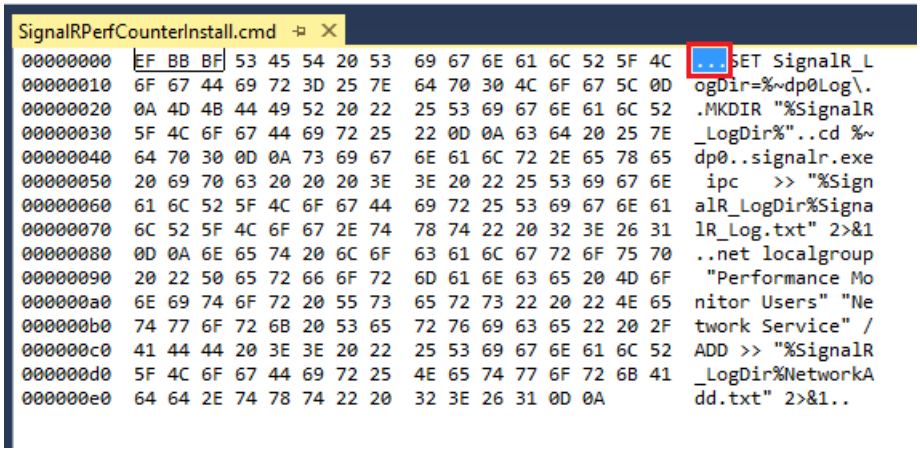


15. Repeat the previous step for the *SignalRPerfCounterInstall.cmd* file.

16. Right-click on the *SignalRPerfCounterInstall.cmd* file and select **Open With**. In the dialog that appears, select **Binary Editor** and select **OK**.



17. In the binary editor, select any leading bytes in the file and delete them. Save and close the file.



18. Open *ServiceDefinition.csdef* and add a startup task that executes the *SignalrPerfCounterInstall.cmd* file when the service starts up:

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceDefinition name="SignalRPerfCounters"
xmlns="http://schemas.microsoft.com/ServiceHosting/2008/10/ServiceDefinition" schemaVersion="2015-
04.2.6">
  <WebRole name="WebRole1" vmsize="Small">
    <Startup>
      <Task commandLine="Startup\SignalRPerfCounterInstall.cmd" executionContext="elevated"
taskType="background" />
    </Startup>
    <Sites>
      <Site name="Web">
        <Bindings>
          <Binding name="Endpoint1" endpointName="Endpoint1" />
        </Bindings>
      </Site>
    </Sites>
    <Endpoints>
      <InputEndpoint name="Endpoint1" protocol="http" port="80" />
    </Endpoints>
  </WebRole>
</ServiceDefinition>
```

19. Open `Views/Shared/_Layout.cshtml` and remove the jQuery bundle script from the end of the file.

```
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
    </footer>
</div>

@Scripts.Render("~/bundles/jquery")
@Scripts.Render("~/bundles/bootstrap")
@RenderSection("scripts", required: false)
</body>
</html>
```

20. Add a JavaScript client that continuously calls the `increment` method on the server. Open `Views/Home/Index.cshtml` and replace the contents with the following code:

```
@{
    ViewBag.Title = "Home Page";
}

<script src="~/Scripts/jquery-1.10.2.min.js"></script>
<script src="~/Scripts/jquery.signalR-2.2.1.min.js"></script>
<script src="~/signalr/hubs" type="text/javascript"></script>

<div id="body">
    <section class="featured">
        <div class="content-wrapper">
            <p>
                Hello World!
            </p>
            <div style="font-size:large;">
                My Counter: <span id="counter"></span>
            </div>
        </div>
    </section>
    <section class="content-wrapper main-content clear-fix"></section>
</div>

<script type="text/javascript">
  $(document).ready(function () {
    var hub = $.connection.myHub;

    hub.client.sendResult = function (x) {
      console.log('sendResult(' + x + ')');
      $("#counter").text(x);
      window.setTimeout(function () {
        hub.server.increment(x);
      }, 1000);
    };

    $.connection.hub.connected = function () {};
    $.connection.hub.disconnected = function () {};

    $.connection.hub.stateChanged(function (change) {
      console.log('new State' + change.newState);
      if (change.newState === $.signalR.connectionState.disconnected) {
        $.connection.hub.start();
      }
      if (change.newState === $.signalR.connectionState.reconnecting) {
        console.log('Re-connecting');
      } else if (change.newState === $.signalR.connectionState.connected) {
        console.log('The server is online');
      }
    });

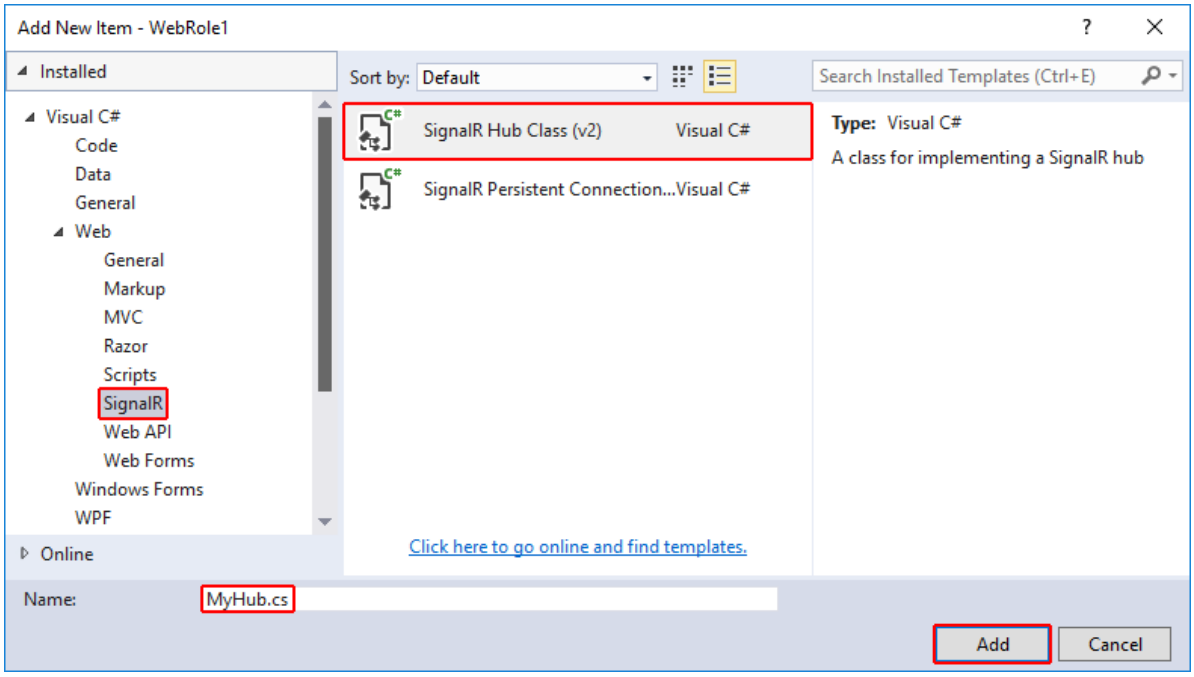
    $.connection.hub.error(function (error) {
      console.log('error ' + error);
    });

    $.connection.hub.logging = true;

    $.connection.hub.reconnected(function () {
      console.log('Reconnected');
      hub.server.increment(0);
    });

    $.connection.hub.start().done(function () {
      console.log('hub started');
      hub.server.increment(0);
    });
  });
</script>
```

21. Create a new folder in the **WebRole1** project named *Hubs*. Right-click the *Hubs* folder in **Solution Explorer** and select **Add** > **New Item**. In the **Add New Item** dialog box, select the **Web** > **SignalR** category, and then select the **SignalR Hub Class (v2)** item template. Name the new hub *MyHub.cs* and select **Add**.



22. *MyHub.cs* will automatically open in the main window. Replace the contents with the following code, then save and close the file:

```
using System.Threading.Tasks;
using Microsoft.AspNet.SignalR;

namespace WebRole1.Hubs
{
    public class MyHub : Hub
    {
        public async Task Increment(int x)
        {
            await this.Clients.Caller.sendResult(x + 1);
        }
    }
}
```

23. *Crank.exe* is a connection density testing tool provided with the SignalR codebase. Since Crank requires a persistent connection, you add one to your site for use when testing. Add a new folder to the **WebRole1** project called *PersistentConnections*. Right-click this folder and select **Add** > **Class**. Name the new class file *MyPersistentConnections.cs* and select **Add**.

24. Visual Studio will open the *MyPersistentConnections.cs* file in the main window. Replace the contents with the following code, then save and close the file:

```
using System.Threading.Tasks;
using Microsoft.AspNet.SignalR;
using Microsoft.AspNet.SignalR.Infrastructure;

namespace WebRole1.PersistentConnections
{
    public class MyPersistentConnection : PersistentConnection
    {
        protected override Task OnReceived(IRequest request, string connectionId, string data)
        {
            //Return data to calling user
            return Connection.Send(connectionId, data);
        }
    }
}
```

25. Using the `Startup` class, the SignalR objects start when OWIN starts up. Open or create *Startup.cs* and replace the contents with the following code:

```
using Microsoft.Owin;
using Owin;
using WebRole1.PersistentConnections;

// Marks this class for automatic OWIN startup
[assembly: OwinStartup(typeof(WebRole1.Startup))]
namespace WebRole1
{
    public partial class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            ConfigureAuth(app);
            // Only needed if "No Authentication" was not selected for the project
            app.MapSignalR();
            app.MapSignalR<MyPersistentConnection>("/echo");
        }
    }
}
```

In the code above, the `OwinStartup` attribute marks this class to start OWIN. The `Configuration` method starts SignalR.

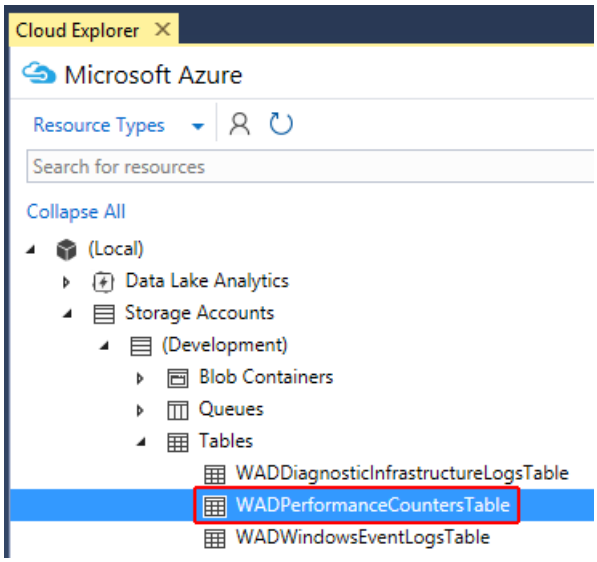
26. Test your application in the Microsoft Azure Emulator by pressing **F5**.

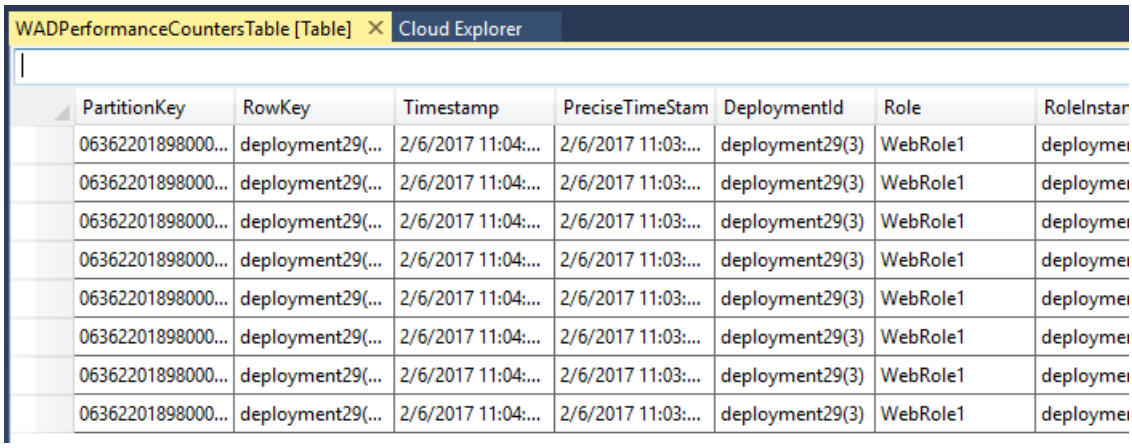
> **NOTE**
>
> If you encounter a **FileLoadException** at **MapSignalR**, change the binding redirects in *web.config* to the following:

```
<dependentAssembly>
  <assemblyIdentity name="Microsoft.Owin" publicKeyToken="31bf3856ad364e35" culture="neutral" />
  <bindingRedirect oldVersion="0.0.0.0-2.0.2.0" newVersion="2.0.0.0" />
</dependentAssembly>
<dependentAssembly>
  <assemblyIdentity name="Microsoft.Owin.Security" publicKeyToken="31bf3856ad364e35" culture="neutral"
/>
  <bindingRedirect oldVersion="0.0.0.0-2.0.2.0" newVersion="2.0.0.0" />
</dependentAssembly>
```

27. Wait about one minute. Open the Cloud Explorer tool window in Visual Studio (**View** > **Cloud Explorer**) and expand the path `(Local)/Storage Accounts/(Development)/Tables`. Double-click

WADPerformanceCountersTable. You should see SignalR counters in the table data. If you don't see the table, you may need to re-enter your Azure Storage credentials. You may need to select the **Refresh** button to see the table in **Cloud Explorer** or select the **Refresh** button in the open table window to see data in the table.





28. To test your application in the cloud, update the **ServiceConfiguration.Cloud.cscfg** file and set the `Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString` to a valid Azure Storage account connection string.

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceConfiguration serviceName="SignalRPerfCounters"
xmlns="http://schemas.microsoft.com/ServiceHosting/2008/10/ServiceConfiguration" osFamily="4"
osVersion="*" schemaVersion="2015-04.2.6">
  <Role name="WebRole1">
    <Instances count="1" />
    <ConfigurationSettings>
      <Setting name="Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString"
value="DefaultEndpointsProtocol=https;AccountName=&lt;account-name&gt;;AccountKey=&lt;account-key&gt;"
/>
    </ConfigurationSettings>
  </Role>
</ServiceConfiguration>
```

29. Deploy the application to your Azure subscription. For details on how to deploy an application to Azure, see How to Create and Deploy a Cloud Service.

30. Wait a few minutes. In **Cloud Explorer**, locate the storage account you configured above and find the `WADPerformanceCountersTable` table in it. You should see SignalR counters in the table data. If you don't see the table, you may need to re-enter your Azure Storage credentials. You may need to select the **Refresh**

button to see the table in **Cloud Explorer** or select the **Refresh** button in the open table window to see data in the table.

Special thanks to Martin Richard for the original content used in this tutorial.

SignalR Deployment and Hosting

2/20/2020 • 2 minutes to read • Edit Online

How to deploy and host SignalR applications.

- Tutorial: SignalR Self-Host
- Using SignalR with Web Apps in Azure App Service

Tutorial: SignalR Self-Host

9/16/2020 • 7 minutes to read • Edit Online

by Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This tutorial shows how to create a self-hosted SignalR 2 server, and how to connect to it with a JavaScript client.

Software versions used in the tutorial

- Visual Studio 2013
- .NET 4.5
- SignalR version 2

Using Visual Studio 2012 with this tutorial

To use Visual Studio 2012 with this tutorial, do the following:

- Update your Package Manager to the latest version.
- Install the Web Platform Installer.
- In the Web Platform Installer, search for and install **ASP.NET and Web Tools 2013.1 for Visual Studio 2012**. This will install Visual Studio templates for SignalR classes such as **Hub**.
- Some templates (such as **OWIN Startup Class**) will not be available; for these, use a Class file instead.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Overview

A SignalR server is usually hosted in an ASP.NET application in IIS, but it can also be self-hosted (such as in a console application or Windows service) using the self-host library. This library, like all of SignalR 2, is built on OWIN (Open Web Interface for .NET). OWIN defines an abstraction between .NET web servers and web applications. OWIN decouples the web application from the server, which makes OWIN ideal for self-hosting a web application in your own process, outside of IIS.

Reasons for not hosting in IIS include:

- Environments where IIS is not available or desirable, such as an existing server farm without IIS.
- The performance overhead of IIS needs to be avoided.
- SignalR functionality is to be added to an existing application that runs in a Windows Service, Azure worker role, or other process.

If a solution is being developed as self-host for performance reasons, it's recommended to also test the

application hosted in IIS to determine the performance benefit.

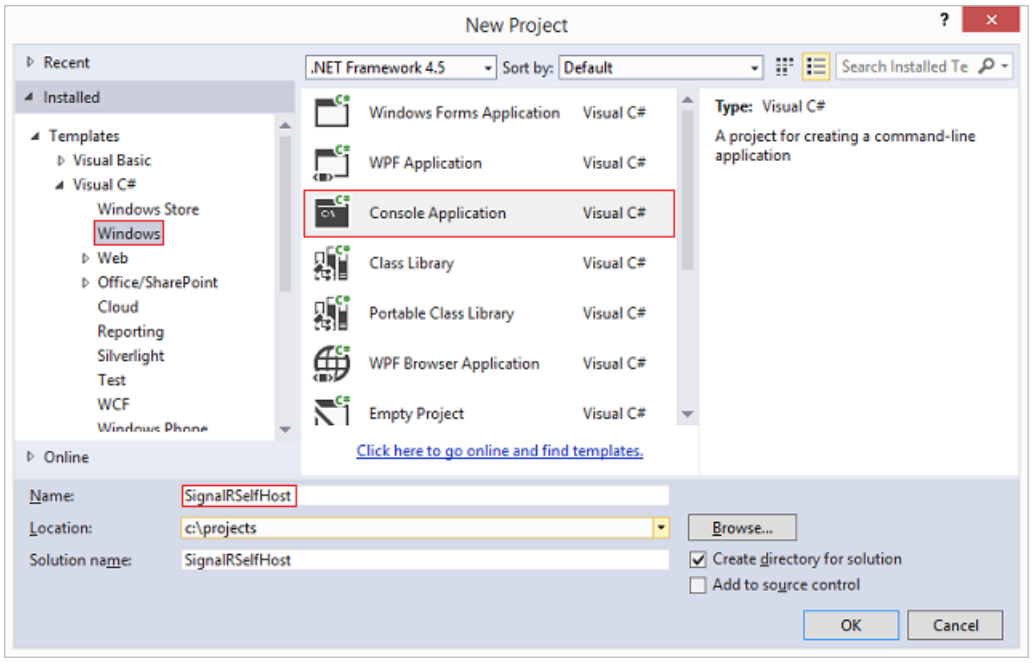
This tutorial contains the following sections:

- Creating the server
- Accessing the server with a JavaScript client

Creating the server

In this tutorial, you'll create a server that's hosted in a console application, but the server can be hosted in any sort of process, such as a Windows service or Azure worker role. For sample code for hosting a SignalR server in a Windows Service, see Self-Hosting SignalR in a Windows Service.

1. Open Visual Studio 2013 with administrator privileges. Select **File**, **New Project**. Select **Windows** under the **Visual C#** node in the **Templates** pane, and select the **Console Application** template. Name the new project "SignalRSelfHost" and click **OK**.



2. Open the NuGet package manager console by selecting **Tools** > **NuGet Package Manager** > **Package Manager Console**.

3. In the package manager console, enter the following command:

```
Install-Package Microsoft.AspNet.SignalR.SelfHost
```

   This command adds the SignalR 2 Self-Host libraries to the project.

4. In the package manager console, enter the following command:

```
Install-Package Microsoft.Owin.Cors
```

   This command adds the Microsoft.Owin.Cors library to the project. This library will be used for cross-domain support, which is required for applications that host SignalR and a web page client in different domains. Since you'll be hosting the SignalR server and the web client on different ports, this means that cross-domain must be enabled for communication between these components.

5. Replace the contents of Program.cs with the following code.

```
using System;
using Microsoft.AspNet.SignalR;
using Microsoft.Owin.Hosting;
using Owin;
using Microsoft.Owin.Cors;

namespace SignalRSelfHost
{
    class Program
    {
        static void Main(string[] args)
        {
            // This will *ONLY* bind to localhost, if you want to bind to all addresses
            // use http://*:8080 to bind to all addresses.
            // See http://msdn.microsoft.com/library/system.net.httplistener.aspx
            // for more information.
            string url = "http://localhost:8080";
            using (WebApp.Start(url))
            {
                Console.WriteLine("Server running on {0}", url);
                Console.ReadLine();
            }
        }
    }
    class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.UseCors(CorsOptions.AllowAll);
            app.MapSignalR();
        }
    }
    public class MyHub : Hub
    {
        public void Send(string name, string message)
        {
            Clients.All.addMessage(name, message);
        }
    }
}
```

The above code includes three classes:

- **Program**, including the **Main** method defining the primary path of execution. In this method, a web application of type **Startup** is started at the specified URL (`http://localhost:8080`). If security is required on the endpoint, SSL can be implemented. See How to: Configure a Port with an SSL Certificate for more information.
- **Startup**, the class containing the configuration for the SignalR server (the only configuration this tutorial uses is the call to `UseCors`), and the call to `MapSignalR` , which creates routes for any Hub objects in the project.
- **MyHub**, the SignalR Hub class that the application will provide to clients. This class has a single method, **Send**, that clients will call to broadcast a message to all other connected clients.

6. Compile and run the application. The address that the server is running should show in a console window.

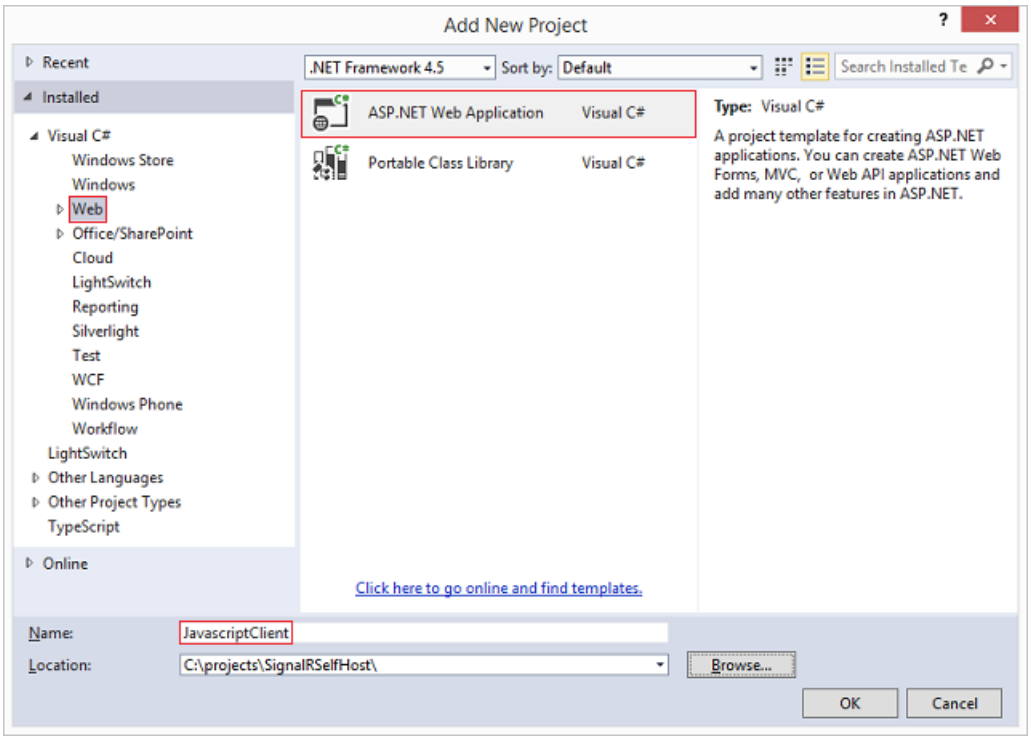
7. If execution fails with the exception `System.Reflection.TargetInvocationException was unhandled`, you will need to restart Visual Studio with administrator privileges.

8. Stop the application before proceeding to the next section.

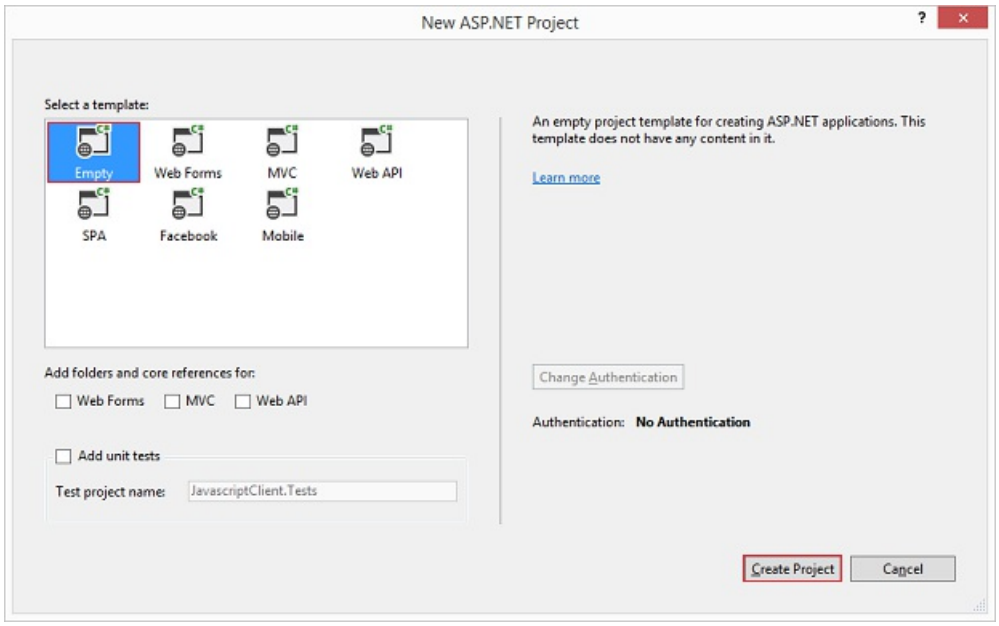
Accessing the server with a JavaScript client

In this section, you'll use the same JavaScript client from the Getting Started tutorial. We'll only make one modification to the client, which is to explicitly define the hub URL. With a self-hosted application, the server may not necessarily be at the same address as the connection URL (due to reverse proxies and load balancers), so the URL needs to be defined explicitly.

1. In **Solution Explorer**, right-click on the solution and select **Add**, **New Project**. Select the **Web** node, and select the **ASP.NET Web Application** template. Name the project "JavascriptClient" and click **OK**.



2. Select the **Empty** template, and leave the remaining options unselected. Select **Create Project**.

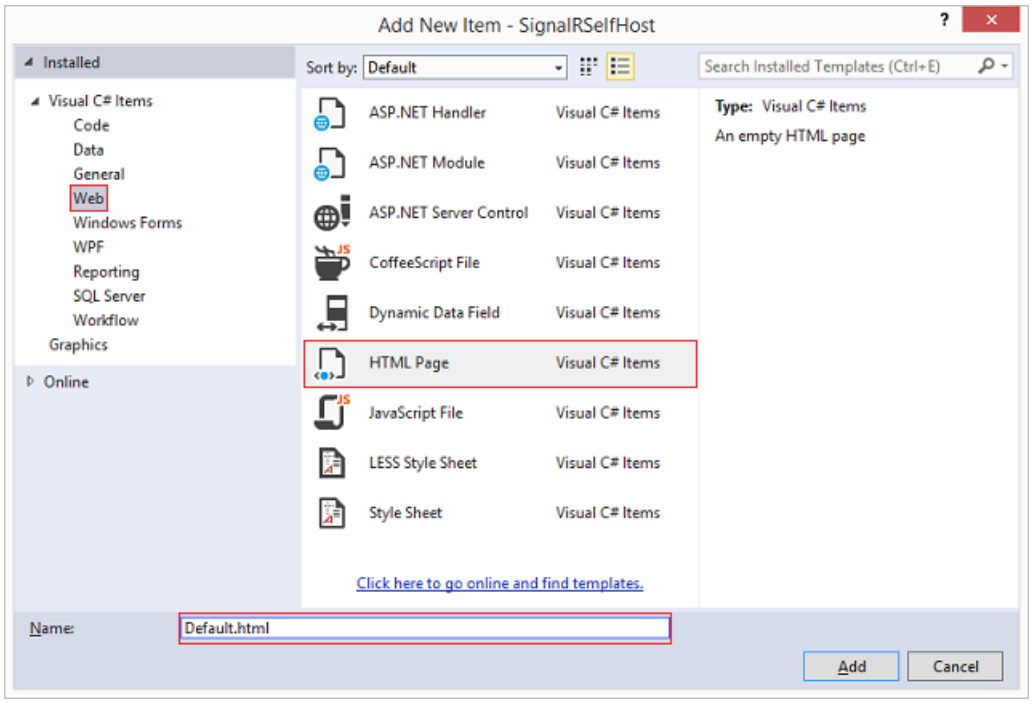


3. In the package manager console, select the "JavascriptClient" project in the **Default project** drop-down, and execute the following command:

```
Install-Package Microsoft.AspNet.SignalR.JS
```

This command installs the SignalR and JQuery libraries that you'll need in the client.

4. Right-click on your project and select **Add**, **New Item**. Select the **Web** node, and select HTML Page. Name the page **Default.html**.



5. Replace the contents of the new HTML page with the following code. Verify that the script references here match the scripts in the Scripts folder of the project.

```
<!DOCTYPE html>
<html>
<head>
    <title>SignalR Simple Chat</title>
    <style type="text/css">
        .container {
            background-color: #99CCFF;
            border: thick solid #808080;
            padding: 20px;
            margin: 20px;
        }
    </style>
</head>
<body>
    <div class="container">
        <input type="text" id="message" />
        <input type="button" id="sendmessage" value="Send" />
        <input type="hidden" id="displayname" />
        <ul id="discussion"></ul>
    </div>
    <!--Script references. -->
    <!--Reference the jQuery library. -->
    <script src="Scripts/jquery-1.6.4.min.js"></script>
    <!--Reference the SignalR library. -->
    <script src="Scripts/jquery.signalR-2.1.0.min.js"></script>
    <!--Reference the autogenerated SignalR hub script. -->
    <script src="http://localhost:8080/signalr/hubs"></script>
    <!--Add script to update the page and send messages.-->
    <script type="text/javascript">
        $(function () {
        //Set the hubs URL for the connection
            $.connection.hub.url = "http://localhost:8080/signalr";

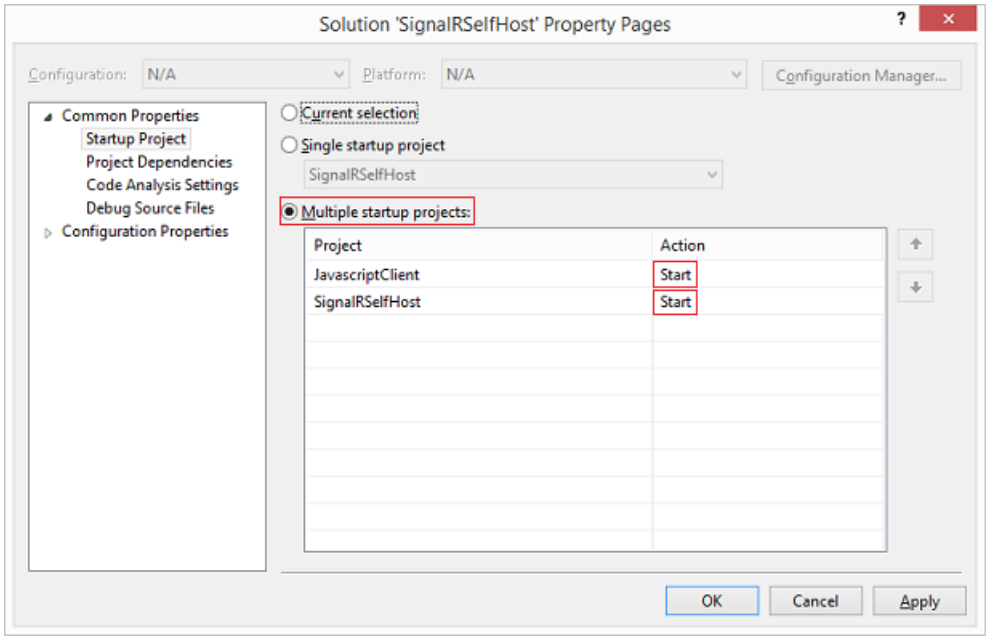
            // Declare a proxy to reference the hub.
            var chat = $.connection.myHub;

            // Create a function that the hub can call to broadcast messages.
            chat.client.addMessage = function (name, message) {
                // Html encode display name and message.
                var encodedName = $('<div />').text(name).html();
                var encodedMsg = $('<div />').text(message).html();
                // Add the message to the page.
                $('#discussion').append('<li><strong>' + encodedName
                    + '</strong>:  ' + encodedMsg + '</li>');
            };
            // Get the user name and store it to prepend to messages.
            $('#displayname').val(prompt('Enter your name:', ''));
            // Set initial focus to message input box.
            $('#message').focus();
            // Start the connection.
            $.connection.hub.start().done(function () {
                $('#sendmessage').click(function () {
                    // Call the Send method on the hub.
                    chat.server.send($('#displayname').val(), $('#message').val());
                    // Clear text box and reset focus for next comment.
                    $('#message').val('').focus();
                });
            });
        });
    </script>
</body>
</html>
```

The following code (highlighted in the code sample above) is the addition that you've made to the client used in the Getting Stared tutorial (in addition to upgrading the code to SignalR version 2 beta). This line of code explicitly sets the base connection URL for SignalR on the server.

```
//Set the hubs URL for the connection
$.connection.hub.url = "http://localhost:8080/signalr";
```

6. Right-click on the solution, and select **Set Startup Projects...**. Select the **Multiple startup projects** radio button, and set both projects' **Action** to **Start**.



7. Right-click on "Default.html" and select **Set As Start Page**.

8. Run the application. The server and page will launch. You may need to reload the web page (or select **Continue** in the debugger) if the page loads before the server is started.

9. In the browser, provide a username when prompted. Copy the page's URL into another browser tab or window and provide a different username. You will be able to send messages from one browser pane to the other, as in the Getting Started tutorial.

Using SignalR with Web Apps in Azure App Service

2/20/2020 • 3 minutes to read • Edit Online

by Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This document describes how to configure a SignalR application that runs on Microsoft Azure.

Software versions used in the tutorial

- Visual Studio 2013 or Visual Studio 2012
- .NET 4.5
- SignalR version 2
- Azure SDK 2.3 for Visual Studio 2013 or 2012

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum, StackOverflow.com, or the Microsoft Azure forums.

Table of Contents

- Introduction
- Deploying a SignalR Web App to Azure App Service
- Enabling WebSockets on Azure App Service
- Using the Azure Redis Cache Backplane
- Next Steps

Introduction

ASP.NET SignalR can be used to bring a new level of interactivity between servers and web or .NET clients. When hosted in Azure, SignalR applications can take advantage of the highly available, scalable, and performant environment that running in the cloud provides.

Deploying a SignalR Web App to Azure App Service

SignalR doesn't add any particular complications to deploying an application to Azure versus deploying to an on-premises server. An application that uses SignalR can be hosted in Azure without any changes in configuration or other settings (though for WebSockets support, see Enabling WebSockets on Azure App Service below.) For this tutorial, you'll deploy the application created in the Getting Started Tutorial to Azure.

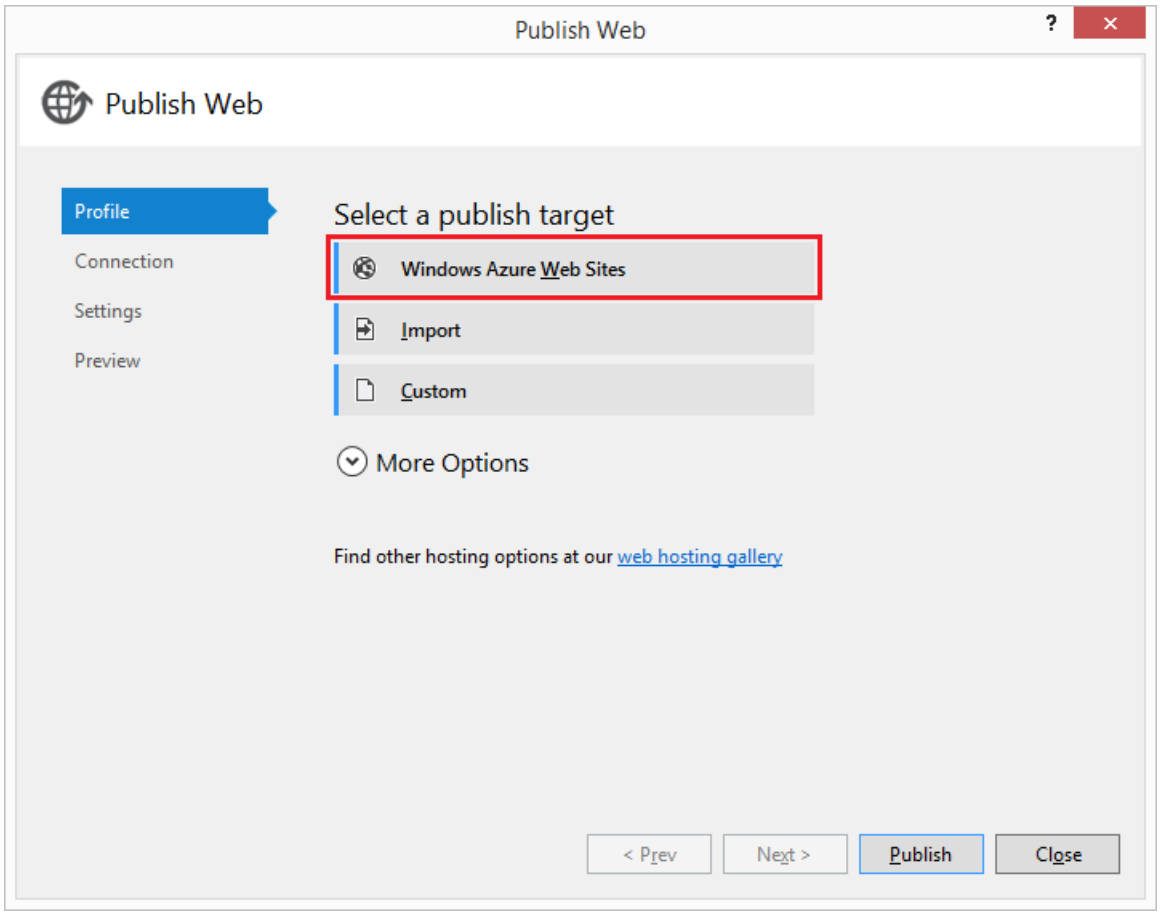
Prerequisites

- Visual Studio 2013. If you don't have Visual Studio, Visual Studio 2013 Express for Web is included in the install of the Azure SDK.

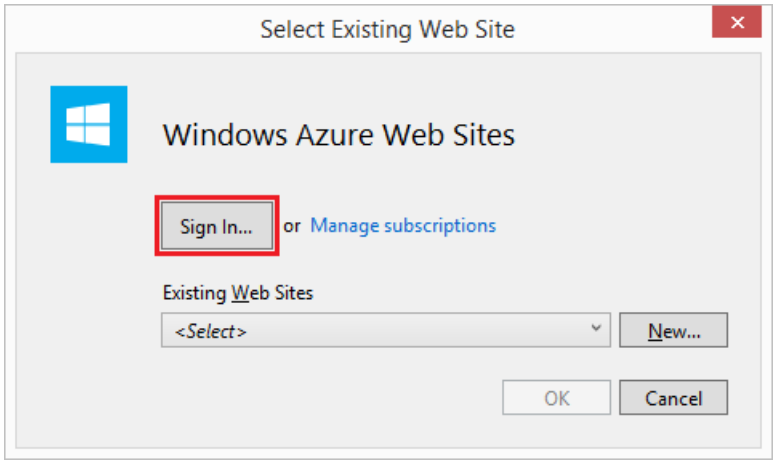
- Azure SDK 2.3 for Visual Studio 2013 or Azure SDK 2.3 for Visual Studio 2012.
- To complete this tutorial, you will need an Azure subscription. You can activate your MSDN subscriber benefits, or sign up for a trial subscription.

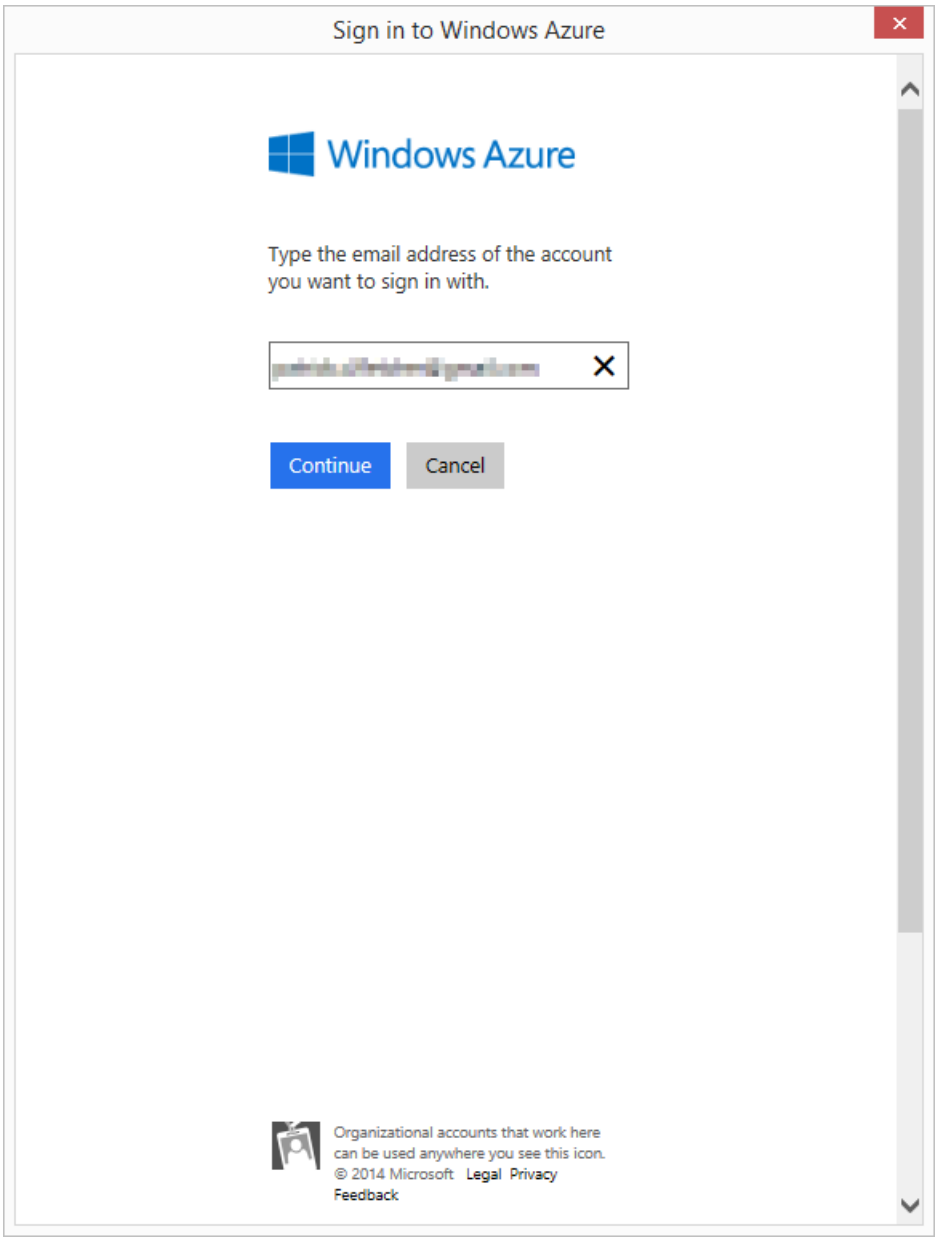
Deploying a SignalR web app to Azure

1. Complete the Getting Started Tutorial, or download the finished project from Code Gallery.
2. In Visual Studio, select **Build**, **Publish SignalR Chat**.
3. In the "Publish Web" dialog, select "Windows Azure Web Sites".

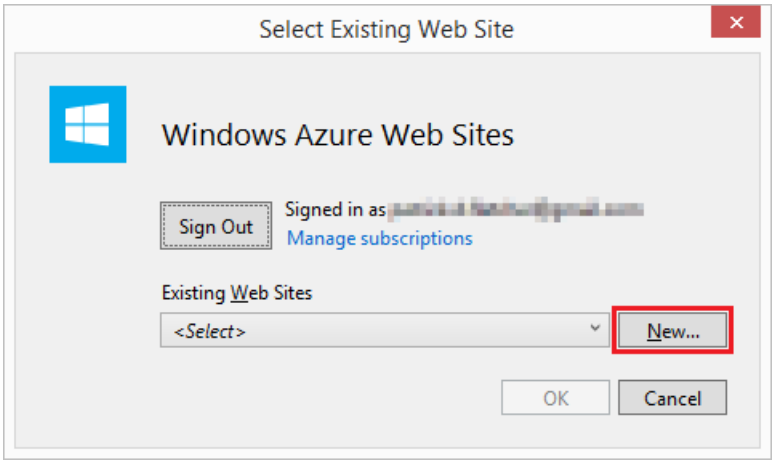


4. If you aren't signed in to your Microsoft account, click **Sign In...** in the "Select Existing Web Site" dialog, and sign in.

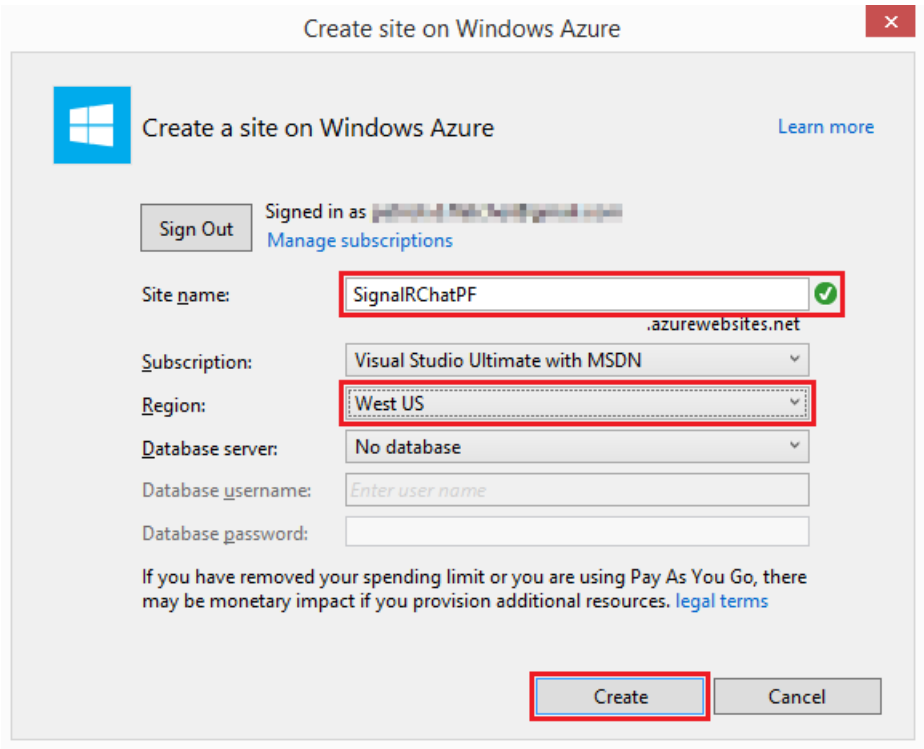




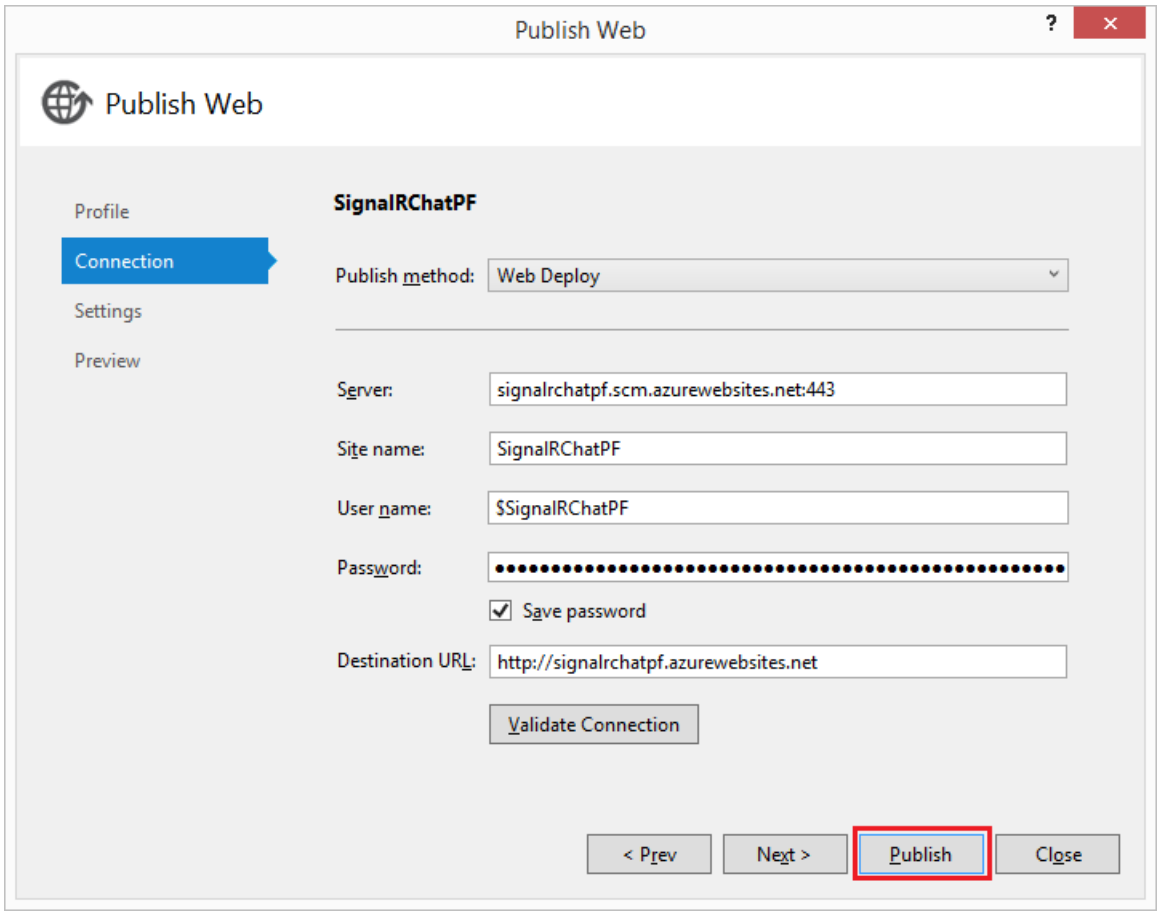
5. In the "Select Existing Web Site" dialog, click **New**.



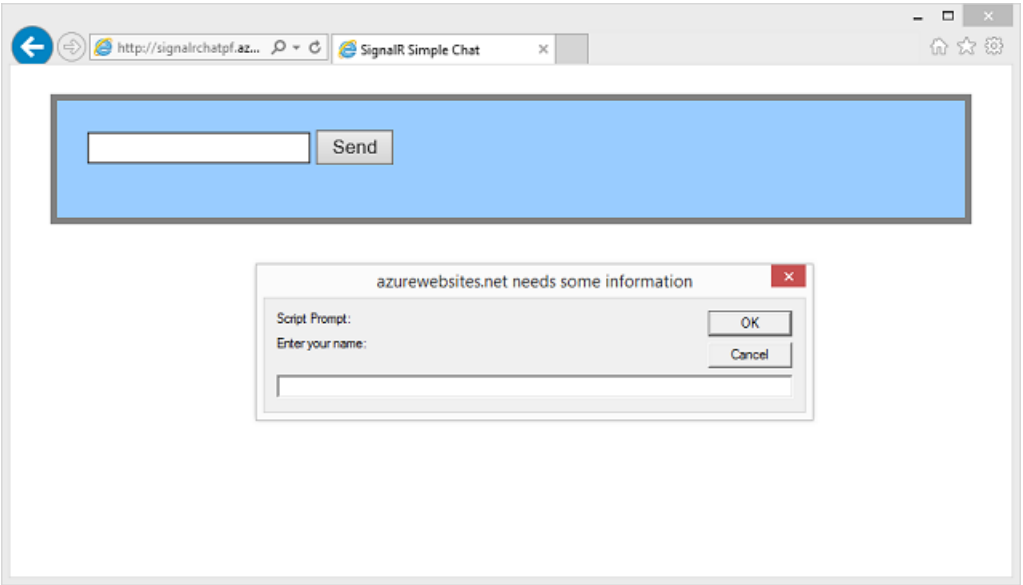
6. In the "Create site on Windows Azure" dialog, enter a unique app name. Select the region closest to you in the Region dropdown. Click **Create**.



7. In the "Publish Web" dialog, click **Publish**.



8. When the app has completed publishing, the SignalR Chat application hosted in Azure App Service Web Apps will open in a browser.

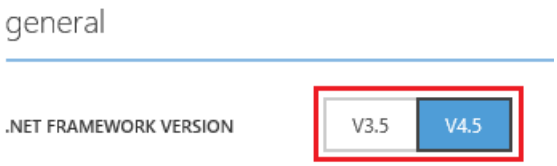


Enabling WebSockets on Azure App Service Web Apps

WebSockets needs to be explicitly enabled in your web app to be used in a SignalR application; otherwise, other protocols will be used (See Transports and Fallbacks for details).

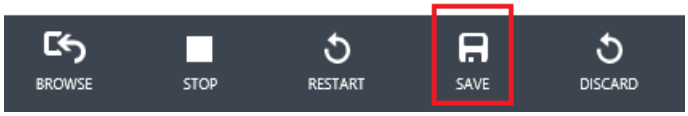
In order to use WebSockets on Azure App Service Web Apps, enable it in the configuration section of the web app. To do this, open your web app in the Azure Management Portal, and select Configure.

At the top of the configuration page, ensure that .NET 4.5 is used for your web app.



On the configuration page, in the **WebSockets** setting, select **On**.

At the bottom of the Configuration page, select **Save** to save your changes.



Using the Azure Redis Cache Backplane

If you use multiple instances for your web app, and the users of those instances need to interact with one another (so that, for instance, chat messages created in one instance can reach the users connected to other instances), the Azure Redis Cache backplane must be implemented in your application.

Next Steps

For more information on Web Apps in Azure App Service, see Web Apps overview.

SignalR Advanced Topics

2/20/2020 • 2 minutes to read • Edit Online

SignalR documentation - advanced topics.

- Dependency Injection in SignalR

Dependency Injection in SignalR

2/20/2020 • 8 minutes to read • Edit Online

by Mike Wasson, Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

Software versions used in this topic

- Visual Studio 2013
- .NET 4.5
- SignalR version 2

Previous versions of this topic

For information about earlier versions of SignalR, see SignalR Older Versions.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Dependency injection is a way to remove hard-coded dependencies between objects, making it easier to replace an object's dependencies, either for testing (using mock objects) or to change run-time behavior. This tutorial shows how to perform dependency injection on SignalR hubs. It also shows how to use IoC containers with SignalR. An IoC container is a general framework for dependency injection.

What is Dependency Injection?

Skip this section if you are already familiar with dependency injection.

Dependency injection (DI) is a pattern where objects are not responsible for creating their own dependencies. Here is a simple example to motivate DI. Suppose you have an object that needs to log messages. You might define a logging interface:

```
interface ILogger
{
    void LogMessage(string message);
}
```

In your object, you can create an `ILogger` to log messages:

```
// Without dependency injection.
class SomeComponent
{
    ILogger _logger = new FileLogger(@"C:\logs\log.txt");

    public void DoSomething()
    {
        _logger.LogMessage("DoSomething");
    }
}
```

This works, but it's not the best design. If you want to replace `FileLogger` with another `ILogger` implementation, you will have to modify `SomeComponent`. Supposing that a lot of other objects use `FileLogger`, you will need to change all of them. Or if you decide to make `FileLogger` a singleton, you'll also need to make changes throughout the application.

A better approach is to "inject" an `ILogger` into the object—for example, by using a constructor argument:

```
// With dependency injection.
class SomeComponent
{
    ILogger _logger;

    // Inject ILogger into the object.
    public SomeComponent(ILogger logger)
    {
        if (logger == null)
        {
            throw new NullReferenceException("logger");
        }
        _logger = logger;
    }

    public void DoSomething()
    {
        _logger.LogMessage("DoSomething");
    }
}
```

Now the object is not responsible for selecting which `ILogger` to use. You can switch `ILogger` implementations without changing the objects that depend on it.

```
var logger = new TraceLogger(@"C:\logs\log.etl");
var someComponent = new SomeComponent(logger);
```

This pattern is called constructor injection. Another pattern is setter injection, where you set the dependency through a setter method or property.

Simple Dependency Injection in SignalR

Consider the Chat application from the tutorial Getting Started with SignalR. Here is the hub class from that application:

```
public class ChatHub : Hub
{
    public void Send(string name, string message)
    {
        Clients.All.addMessage(name, message);
    }
}
```

Suppose that you want to store chat messages on the server before sending them. You might define an interface that abstracts this functionality, and use DI to inject the interface into the `ChatHub` class.

```
public interface IChatRepository
{
    void Add(string name, string message);
    // Other methods not shown.
}

public class ChatHub : Hub
{
    private IChatRepository _repository;

    public ChatHub(IChatRepository repository)
    {
        _repository = repository;
    }

    public void Send(string name, string message)
    {
        _repository.Add(name, message);
        Clients.All.addMessage(name, message);
    }
```

The only problem is that a SignalR application does not directly create hubs; SignalR creates them for you. By default, SignalR expects a hub class to have a parameterless constructor. However, you can easily register a function to create hub instances, and use this function to perform DI. Register the function by calling **GlobalHost.DependencyResolver.Register**.

```
public void Configuration(IAppBuilder app)
{
    GlobalHost.DependencyResolver.Register(
        typeof(ChatHub),
        () => new ChatHub(new ChatMessageRepository()));

    App.MapSignalR();

    // ...
}
```

Now SignalR will invoke this anonymous function whenever it needs to create a `ChatHub` instance.

IoC Containers

The previous code is fine for simple cases. But you still had to write this:

```
... new ChatHub(new ChatMessageRepository()) ...
```

In a complex application with many dependencies, you might need to write a lot of this "wiring" code. This code can be hard to maintain, especially if dependencies are nested. It is also hard to unit test.

One solution is to use an IoC container. An IoC container is a software component that is responsible for managing dependencies.You register types with the container, and then use the container to create objects. The container automatically figures out the dependency relations. Many IoC containers also allow you to control things like object lifetime and scope.

> **NOTE**
> "IoC" stands for "inversion of control", which is a general pattern where a framework calls into application code. An IoC container constructs your objects for you, which "inverts" the usual flow of control.

Using IoC Containers in SignalR

The Chat application is probably too simple to benefit from an IoC container. Instead, let's look at the StockTicker sample.

The StockTicker sample defines two main classes:

- `StockTickerHub`: The hub class, which manages client connections.
- `StockTicker`: A singleton that holds stock prices and periodically updates them.

`StockTickerHub` holds a reference to the `StockTicker` singleton, while `StockTicker` holds a reference to the **IHubConnectionContext** for the `StockTickerHub`. It uses this interface to communicate with `StockTickerHub` instances. (For more information, see Server Broadcast with ASP.NET SignalR.)

We can use an IoC container to untangle these dependencies a bit. First, let's simplify the `StockTickerHub` and `StockTicker` classes. In the following code, I've commented out the parts that we don't need.

Remove the parameterless constructor from `StockTickerHub`. Instead, we will always use DI to create the hub.

```
[HubName("stockTicker")]
public class StockTickerHub : Hub
{
    private readonly StockTicker _stockTicker;

    //public StockTickerHub() : this(StockTicker.Instance) { }

    public StockTickerHub(StockTicker stockTicker)
    {
        if (stockTicker == null)
        {
            throw new ArgumentNullException("stockTicker");
        }
        _stockTicker = stockTicker;
    }

    // ...
```

For StockTicker, remove the singleton instance. Later, we'll use the IoC container to control the StockTicker lifetime. Also, make the constructor public.

```
public class StockTicker
{
    //private readonly static Lazy<StockTicker> _instance = new Lazy<StockTicker>(
    //    () => new StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>().Clients));

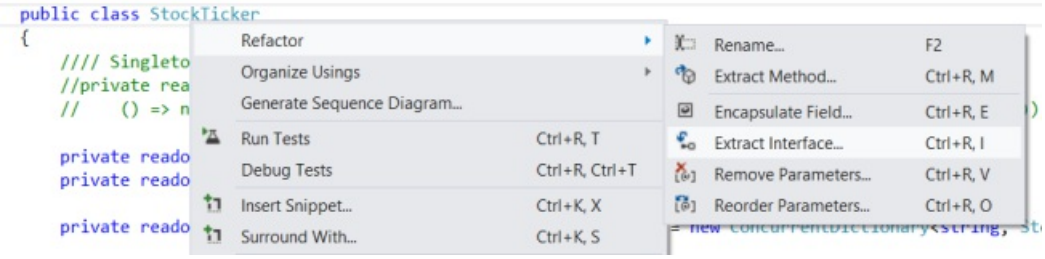
    // Important! Make this constructor public.
    public StockTicker(IHubConnectionContext<dynamic> clients)
    {
        if (clients == null)
        {
            throw new ArgumentNullException("clients");
        }

        Clients = clients;
        LoadDefaultStocks();
    }

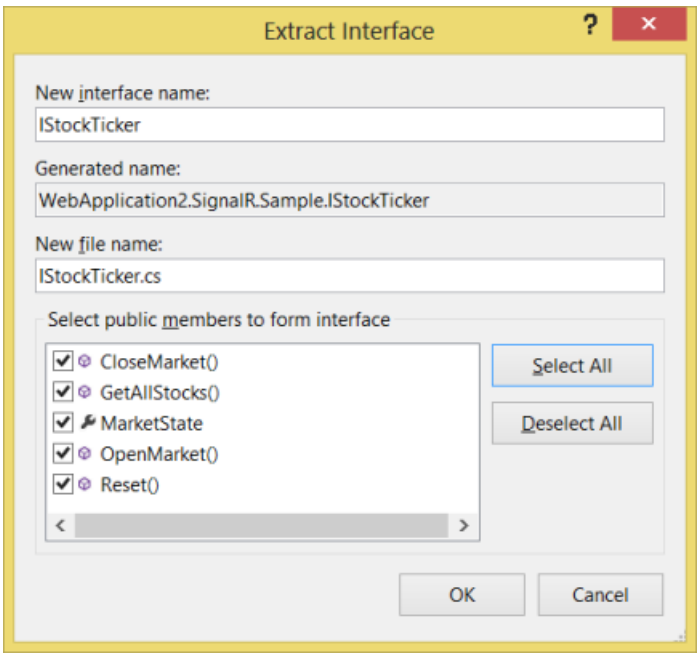
    //public static StockTicker Instance
    //{
    //    get
    //    {
    //        return _instance.Value;
    //    }
    //}
```

Next, we can refactor the code by creating an interface for `StockTicker`. We'll use this interface to decouple the `StockTickerHub` from the `StockTicker` class.

Visual Studio makes this kind of refactoring easy. Open the file StockTicker.cs, right-click on the `StockTicker` class declaration, and select **Refactor** ... **Extract Interface**.



In the **Extract Interface** dialog, click **Select All**. Leave the other defaults. Click **OK**.



Visual Studio creates a new interface named `IStockTicker`, and also changes `StockTicker` to derive from `IStockTicker`.

Open the file IStockTicker.cs and change the interface to **public**.

```
public interface IStockTicker
{
    void CloseMarket();
    IEnumerable<Stock> GetAllStocks();
    MarketState MarketState { get; }
    void OpenMarket();
    void Reset();
}
```

In the `StockTickerHub` class, change the two instances of `StockTicker` to `IStockTicker`:

```
[HubName("stockTicker")]
public class StockTickerHub : Hub
{
    private readonly IStockTicker _stockTicker;

    public StockTickerHub(IStockTicker stockTicker)
    {
        if (stockTicker == null)
        {
            throw new ArgumentNullException("stockTicker");
        }
        _stockTicker = stockTicker;
    }
```

Creating an `IStockTicker` interface isn't strictly necessary, but I wanted to show how DI can help to reduce coupling between components in your application.

Add the Ninject Library

There are many open-source IoC containers for .NET. For this tutorial, I'll use Ninject. (Other popular libraries include Castle Windsor, Spring.Net, Autofac, Unity, and StructureMap.)

Use NuGet Package Manager to install the Ninject library. In Visual Studio, from the **Tools** menu select **NuGet Package Manager** > **Package Manager Console**. In the Package Manager Console window, enter the following command:

```
Install-Package Ninject -Version 3.0.1.10
```

Replace the SignalR Dependency Resolver

To use Ninject within SignalR, create a class that derives from **DefaultDependencyResolver**.

```
internal class NinjectSignalRDependencyResolver : DefaultDependencyResolver
{
    private readonly IKernel _kernel;
    public NinjectSignalRDependencyResolver(IKernel kernel)
    {
        _kernel = kernel;
    }

    public override object GetService(Type serviceType)
    {
        return _kernel.TryGet(serviceType) ?? base.GetService(serviceType);
    }

    public override IEnumerable<object> GetServices(Type serviceType)
    {
        return _kernel.GetAll(serviceType).Concat(base.GetServices(serviceType));
    }
}
```

This class overrides the **GetService** and **GetServices** methods of **DefaultDependencyResolver**. SignalR calls these methods to create various objects at runtime, including hub instances, as well as various services used internally by SignalR.

- The **GetService** method creates a single instance of a type. Override this method to call the Ninject kernel's **TryGet** method. If that method returns null, fall back to the default resolver.
- The **GetServices** method creates a collection of objects of a specified type. Override this method to concatenate the results from Ninject with the results from the default resolver.

Configure Ninject Bindings

Now we'll use Ninject to declare type bindings.

Open your application's Startup.cs class (that you either created manually as per the package instructions in `readme.txt`, or that was created by adding authentication to your project). In the `Startup.Configuration` method, create the Ninject container, which Ninject calls the *kernel*.

```
var kernel = new StandardKernel();
```

Create an instance of our custom dependency resolver:

```
var resolver = new NinjectSignalRDependencyResolver(kernel);
```

Create a binding for `IStockTicker` as follows:

```
kernel.Bind<IStockTicker>()
    .To<Microsoft.AspNet.SignalR.StockTicker.StockTicker>()  // Bind to StockTicker.
    .InSingletonScope();  // Make it a singleton object.
```

This code is saying two things. First, whenever the application needs an `IStockTicker`, the kernel should create an instance of `StockTicker`. Second, the `StockTicker` class should be a created as a singleton object. Ninject will create one instance of the object, and return the same instance for each request.

Create a binding for **IHubConnectionContext** as follows:

```
kernel.Bind(typeof(IHubConnectionContext<dynamic>)).ToMethod(context =>
                    resolver.Resolve<IConnectionManager>().GetHubContext<StockTickerHub>().Clients
                     ).WhenInjectedInto<IStockTicker>();
```

This code creates an anonymous function that returns an **IHubConnection**. The **WhenInjectedInto** method tells Ninject to use this function only when creating `IStockTicker` instances. The reason is that SignalR creates **IHubConnectionContext** instances internally, and we don't want to override how SignalR creates them. This function only applies to our `StockTicker` class.

Pass the dependency resolver into the **MapSignalR** method by adding a hub configuration:

```
var config = new HubConfiguration();
config.Resolver = resolver;
Microsoft.AspNet.SignalR.StockTicker.Startup.ConfigureSignalR(app, config);
```

Update the Startup.ConfigureSignalR method in the sample's Startup class with the new parameter:

```
public static void ConfigureSignalR(IAppBuilder app, HubConfiguration config)
{
    app.MapSignalR(config);
}
```

Now SignalR will use the resolver specified in **MapSignalR**, instead of the default resolver.

Here is the complete code listing for `Startup.Configuration`.

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?
LinkID=316888

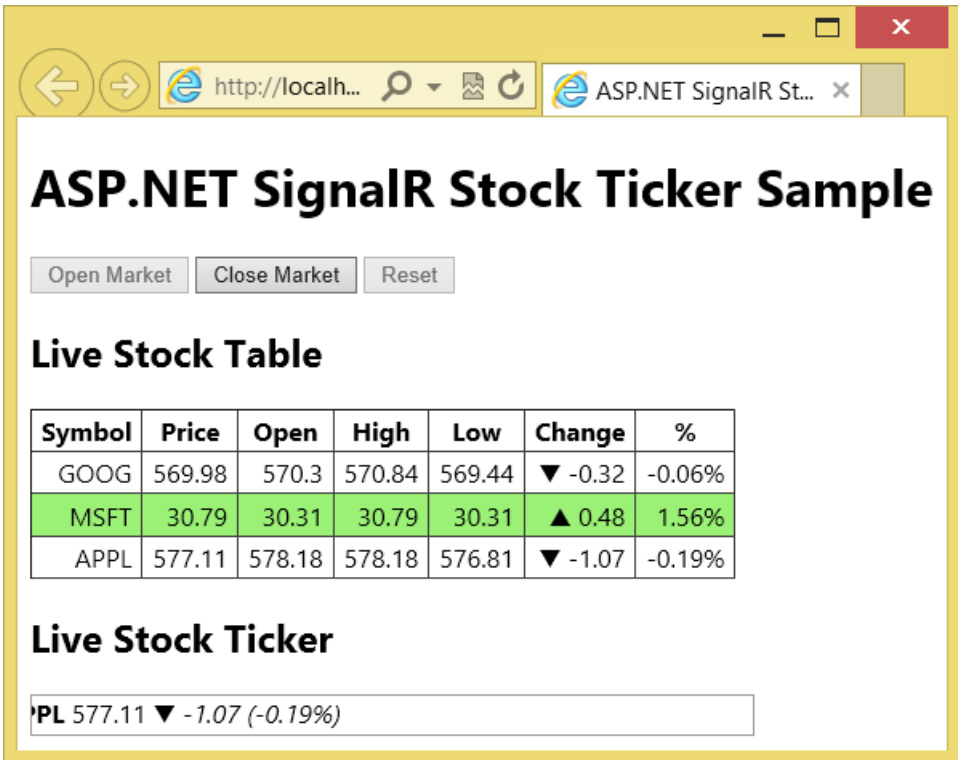
        var kernel = new StandardKernel();
        var resolver = new NinjectSignalRDependencyResolver(kernel);

        kernel.Bind<IStockTicker>()
            .To<Microsoft.AspNet.SignalR.StockTicker.StockTicker>()  // Bind to StockTicker.
            .InSingletonScope();  // Make it a singleton object.

        kernel.Bind(typeof(IHubConnectionContext<dynamic>)).ToMethod(context =>
                resolver.Resolve<IConnectionManager>().GetHubContext<StockTickerHub>().Clients
                    ).WhenInjectedInto<IStockTicker>();

        var config = new HubConfiguration();
        config.Resolver = resolver;
        Microsoft.AspNet.SignalR.StockTicker.Startup.ConfigureSignalR(app, config);
    }
}
```

To run the StockTicker application in Visual Studio, press F5. In the browser window, navigate to `http://localhost:*port*/SignalR.Sample/StockTicker.html`.



The application has exactly the same functionality as before. (For a description, see Server Broadcast with ASP.NET SignalR.) We haven't changed the behavior; just made the code easier to test, maintain, and evolve.

SignalR Releases

2/20/2020 • 2 minutes to read • Edit Online

Information about SignalR releases and how to upgrade from earlier to later releases.

- Upgrading SignalR 1.x Projects to version 2

Upgrading SignalR 1.x Projects to version 2

2/20/2020 • 3 minutes to read • Edit Online

by Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This topic describes how to upgrade an existing SignalR 1.x project to SignalR 2.x, and how to troubleshoot issues that may arise during the upgrade process.

Software versions used in the tutorial

- Visual Studio 2013
- .NET 4.5
- SignalR versions 1 and 2

Using Visual Studio 2012 with this tutorial

To use Visual Studio 2012 with this tutorial, do the following:

- Update your Package Manager to the latest version.
- Install the Web Platform Installer.
- In the Web Platform Installer, search for and install **ASP.NET and Web Tools 2013.1 for Visual Studio 2012**. This will install Visual Studio templates for SignalR classes such as **Hub**.
- Some templates (such as **OWIN Startup Class**) will not be available; for these, use a Class file instead.

Questions and comments

Please leave feedback on how you liked this tutorial and what we could improve in the comments at the bottom of the page. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

SignalR 2 offers a consistent development experience across server platforms using OWIN. This article describes the few steps that are needed to update a SignalR 1.x application to version 2.

While it is encouraged to upgrade applications to SignalR 2, SignalR 1.x will still be supported.

This tutorial describes how to upgrade a web-hosted application to SignalR 2. Self-hosted applications (those that host a server in a console application, Windows service, or other process) are now supported under SignalR 2. For information on how to get started creating a self-hosted application with SignalR 2, see Tutorial: SignalR Self-Host.

Contents

The following sections describe tasks involved with upgrading SignalR projects, and how to troubleshoot issues that may arise.

- Example: Upgrading the Getting Started tutorial to SignalR 2
- Troubleshooting errors encountered during upgrading

Example: Upgrading the Getting Started tutorial application to SignalR 2

In this section, you'll update the application created in the SignalR 1.x version of the Getting Started Tutorial to use SignalR 2.

1. Once you've finished the Getting Started tutorial, right-click on the project, and select **Properties**. Verify that the **Target framework** is set to **.NET Framework 4.5.**

2. Open the Package Manager Console. Remove SignalR 1.x from the project using the following command:

```
Uninstall-Package Microsoft.AspNet.SignalR -RemoveDependencies
```

3. Install SignalR 2 using the following command:

```
Install-Package Microsoft.AspNet.SignalR
```

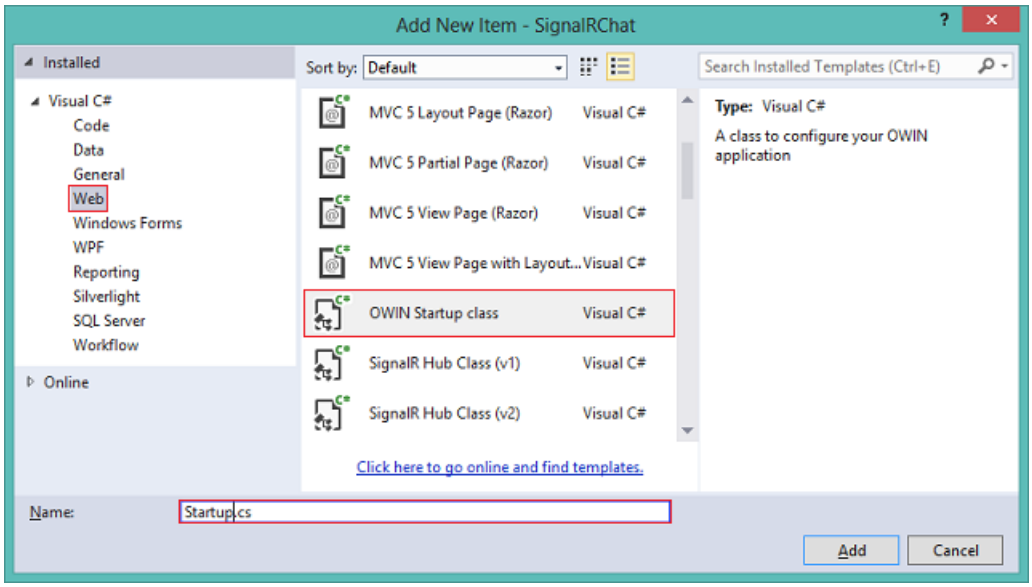
4. In the HTML page, update the script reference for SignalR to match the version of the script now included in the project.

```
<!--Reference the SignalR library. -->
<script src="Scripts/jquery.signalR-2.1.0.min.js"></script>
```

5. In the global application class, remove the call to MapHubs.

```
protected void Application_Start(object sender, EventArgs e)
{
    RouteTable.Routes.MapHubs();
}
```

6. Right-click the solution, and select **Add**, **New Item...**. In the dialog, select **Owin Startup Class**. Name the new class **Startup.cs**.



7. Replace the contents of Startup.cs with the following code:

```
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(SignalRChat.Startup))]
namespace SignalRChat
{

    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.MapSignalR();
        }
    }
}
```

The assembly attribute adds the class to Owin's startup process, which executes the `Configuration` method when Owin starts up. This in turn calls the `MapSignalR` method, which creates routes for all SignalR hubs in the application.

8. Run the project, and copy the URL of the main page into another browser or browser pane, as before. Each page will ask for a username, and messages sent from each page should be visible in both browser panes.

Troubleshooting errors encountered during upgrading

This section describes issues that may arise during upgrading. For a more comprehensive list of errors and issues that may occur with a SignalR application, see SignalR Troubleshooting.

'The call is ambiguous between the following methods or properties'

This error will occur if a reference to `Microsoft.AspNet.SignalR.Owin` is not removed. This package is deprecated; the reference must be removed and the 1.x version of the SelfHost package must be uninstalled.

Hub methods fail silently

Verify that the script references in your client are up to date, and that the `OwinStartup` attribute for your Startup class has the correct class and assembly names for your project. Also, try opening up the hubs address (/signalr/hubs) in your browser; any error that appears will offer more information about what's going wrong.

SignalR Older Versions

2/20/2020 • 2 minutes to read • Edit Online

Documentation that pertains to earlier versions of SignalR.

- Tutorial: Getting Started with SignalR 1.x
- Tutorial: Getting Started with SignalR 1.x and MVC 4
- High-Frequency Realtime with SignalR 1.x
- Tutorial: Server Broadcast with ASP.NET SignalR 1.x
- Hubs API Guide - Server
- SignalR 1.x Hubs API Guide - JavaScript Client
- Hubs API Guide - .NET Client
- Understanding and Handling Connection Lifetime Events in SignalR
- Working with Groups in SignalR 1.x
- Mapping SignalR Users to Connections in SignalR 1.x
- Introduction to SignalR Security (SignalR 1.x)
- Authentication and Authorization for SignalR Hubs (SignalR 1.x)
- Authentication and Authorization for SignalR Persistent Connections (SignalR 1.x)
- SignalR Performance (SignalR 1.x)
- Introduction to Scaleout in SignalR 1.x
- SignalR Scaleout with Azure Service Bus (SignalR 1.x)
- SignalR Scaleout with Redis (SignalR 1.x)
- SignalR Scaleout with SQL Server (SignalR 1.x)
- SignalR Troubleshooting (SignalR 1.x)
- Dependency Injection in SignalR 1.x

Tutorial: Getting Started with SignalR 1.x

2/20/2020 • 8 minutes to read • Edit Online

by Patrick Fletcher, Tim Teebken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This tutorial shows how to use SignalR to create a real-time chat application. You will add SignalR to an empty ASP.NET web application and create an HTML page to send and display messages.

Overview

This tutorial introduces SignalR development by showing how to build a simple browser-based chat application. You will add the SignalR library to an empty ASP.NET web application, create a hub class for sending messages to clients, and create an HTML page that lets users send and receive chat messages. For a similar tutorial that shows how to create a chat application in MVC 4 using an MVC view, see Getting Started with SignalR and MVC 4.

NOTE

This tutorial uses the release (1.x) version of SignalR. For details on changes between SignalR 1.x and 2.0, see Upgrading SignalR 1.x Projects.

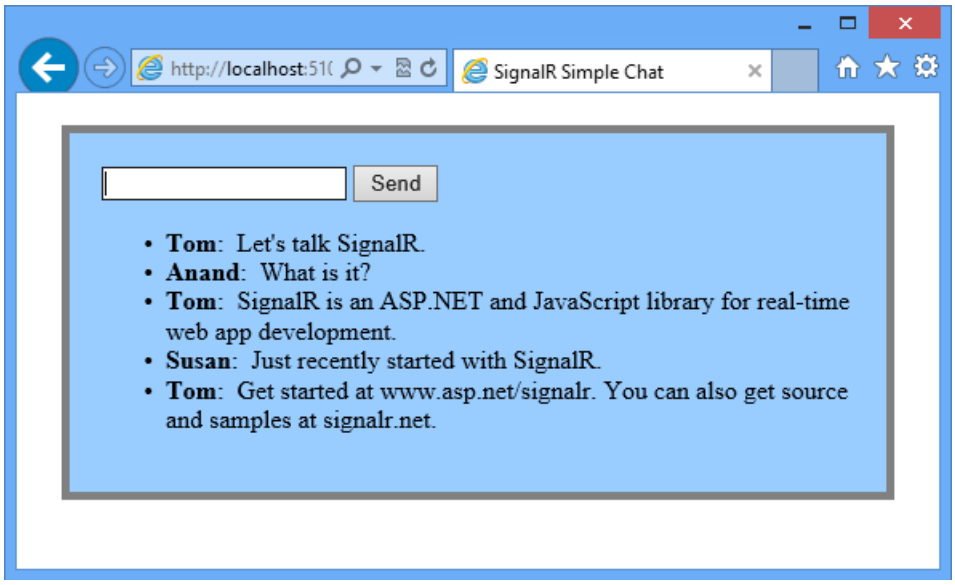
SignalR is an open-source .NET library for building web applications that require live user interaction or real-time data updates. Examples include social applications, multiuser games, business collaboration, and news, weather, or financial update applications. These are often called real-time applications.

SignalR simplifies the process of building real-time applications. It includes an ASP.NET server library and a JavaScript client library to make it easier to manage client-server connections and push content updates to clients. You can add the SignalR library to an existing ASP.NET application to gain real-time functionality.

The tutorial demonstrates the following SignalR development tasks:

- Adding the SignalR library to an ASP.NET web application.
- Creating a hub class to push content to clients.
- Using the SignalR jQuery library in a web page to send messages and display updates from the hub.

The following screen shot shows the chat application running in a browser. Each new user can post comments and see comments added after the user joins the chat.



Sections:

- Set up the Project
- Run the Sample
- Examine the Code
- Next Steps

Set up the Project

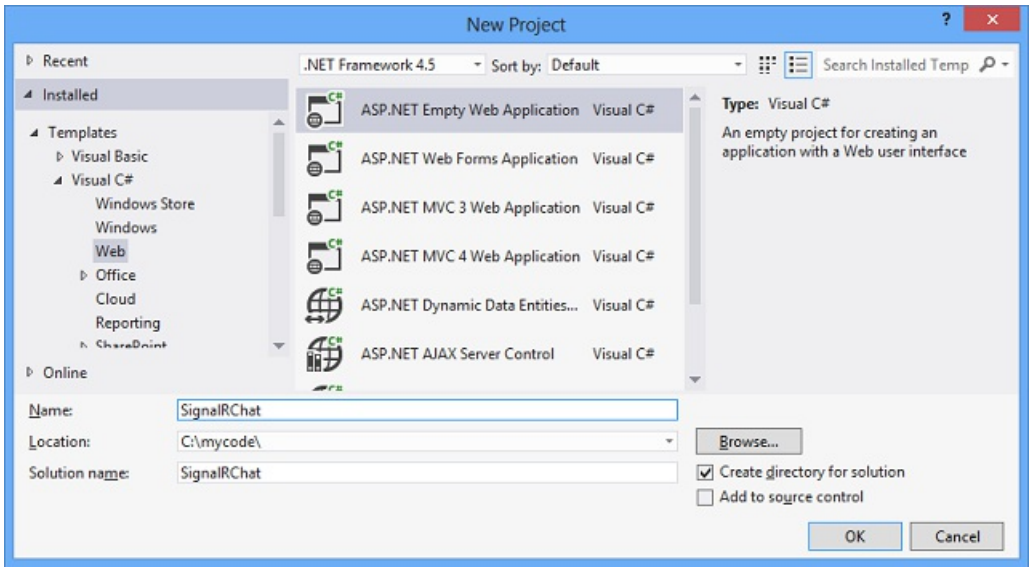
This section shows how to create an empty ASP.NET web application, add SignalR, and create the chat application.

Prerequisites:

- Visual Studio 2010 SP1 or 2012. If you do not have Visual Studio, see ASP.NET Downloads to get the free Visual Studio 2012 Express Development Tool.
- Microsoft ASP.NET and Web Tools 2012.2. For Visual Studio 2012, this installer adds new ASP.NET features including SignalR templates to Visual Studio. For Visual Studio 2010 SP1, an installer is not available but you can complete the tutorial by installing the SignalR NuGet package as described in the setup steps.

The following steps use Visual Studio 2012 to create an ASP.NET Empty Web Application and add the SignalR library:

1. In Visual Studio create an ASP.NET Empty Web Application.



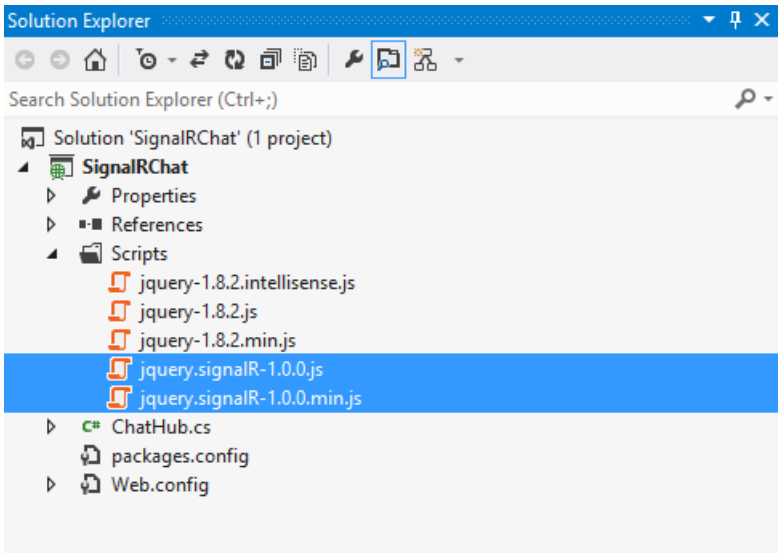
2. Open the **Package Manager Console** by selecting **Tools | NuGet Package Manager | Package Manager Console**. Enter the following command into the console window:

```
Install-Package Microsoft.AspNet.SignalR -Version 1.1.3
```

   This command installs the latest version of SignalR 1.x.

3. In **Solution Explorer**, right-click the project, select **Add | Class**. Name the new class **ChatHub**.

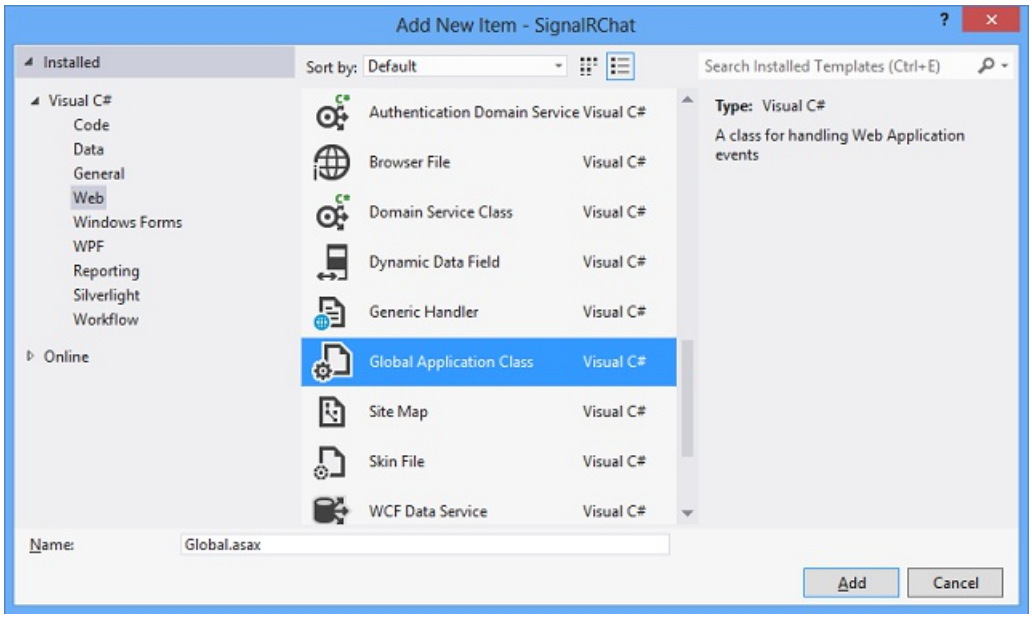
4. In **Solution Explorer** expand the Scripts node. Script libraries for jQuery and SignalR are visible in the project.



5. Replace the code in the **ChatHub** class with the following code.

```
using System;
using System.Web;
using Microsoft.AspNet.SignalR;
namespace SignalRChat
{
    public class ChatHub : Hub
    {
        public void Send(string name, string message)
        {
            // Call the broadcastMessage method to update clients.
            Clients.All.broadcastMessage(name, message);
        }
    }
}
```

6. In **Solution Explorer**, right-click the project, then click **Add | New Item**. In the **Add New Item** dialog, select **Global Application Class** and click **Add**.



7. Add the following `using` statements after the provided `using` statements in the Global.asax.cs class.

```
using System.Web.Routing;
using Microsoft.AspNet.SignalR;
```

8. Add the following line of code in the `Application_Start` method of the Global class to register the default route for SignalR hubs.

```
// Register the default hubs route: ~/signalr/hubs
RouteTable.Routes.MapHubs();
```

9. In **Solution Explorer**, right-click the project, then click **Add | New Item**. In the **Add New Item** dialog, select Html Page and click **Add**.

10. In **Solution Explorer**, right-click the HTML page you just created and click **Set as Start Page**.

11. Replace the default code in the HTML page with the following code.

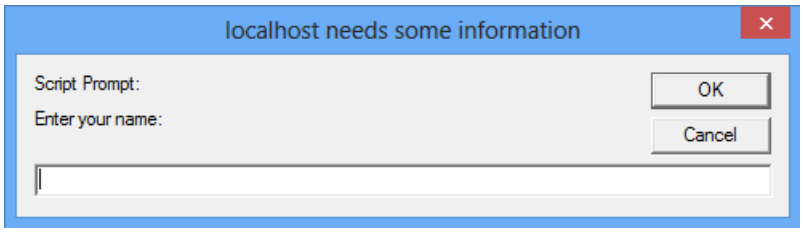
```
<!DOCTYPE html>
<html>
<head>
    <title>SignalR Simple Chat</title>
    <style type="text/css">
        .container {
            background-color: #99CCFF;
            border: thick solid #808080;
            padding: 20px;
            margin: 20px;
        }
    </style>
</head>
<body>
    <div class="container">
        <input type="text" id="message" />
        <input type="button" id="sendmessage" value="Send" />
        <input type="hidden" id="displayname" />
        <ul id="discussion">
        </ul>
    </div>
    <!--Script references. -->
    <!--Reference the jQuery library. -->
    <script src="/Scripts/jquery-1.6.4.min.js" ></script>
    <!--Reference the SignalR library. -->
    <script src="/Scripts/jquery.signalR-1.1.4.js"></script>
    <!--Reference the autogenerated SignalR hub script. -->
    <script src="/signalr/hubs"></script>
    <!--Add script to update the page and send messages.-->
    <script type="text/javascript">
        $(function () {
            // Declare a proxy to reference the hub.
            var chat = $.connection.chatHub;
            // Create a function that the hub can call to broadcast messages.
            chat.client.broadcastMessage = function (name, message) {
                // Html encode display name and message.
                var encodedName = $('<div />').text(name).html();
                var encodedMsg = $('<div />').text(message).html();
                // Add the message to the page.
                $('#discussion').append('<li><strong>' + encodedName
                    + '</strong>:  ' + encodedMsg + '</li>');
            };
            // Get the user name and store it to prepend to messages.
            $('#displayname').val(prompt('Enter your name:', ''));
            // Set initial focus to message input box.
            $('#message').focus();
            // Start the connection.
            $.connection.hub.start().done(function () {
                $('#sendmessage').click(function () {
                    // Call the Send method on the hub.
                    chat.server.send($('#displayname').val(), $('#message').val());
                    // Clear text box and reset focus for next comment.
                    $('#message').val('').focus();
                });
            });
        });
    </script>
</body>
</html>
```

12. **Save All** for the project.

Run the Sample

1. Press F5 to run the project in debug mode. The HTML page loads in a browser instance and prompts for a

user name.



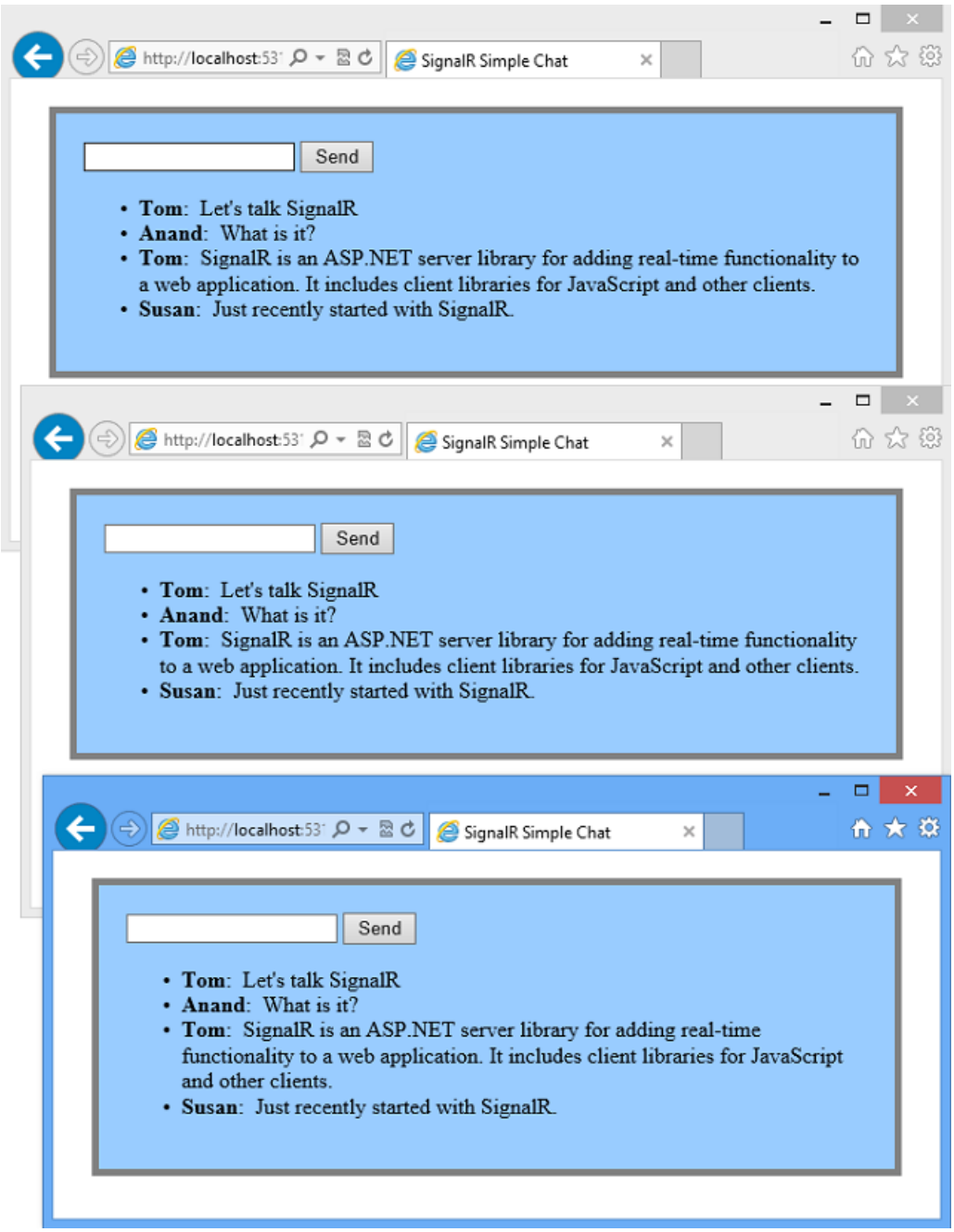
2. Enter a user name.

3. Copy the URL from the address line of the browser and use it to open two more browser instances. In each browser instance, enter a unique user name.

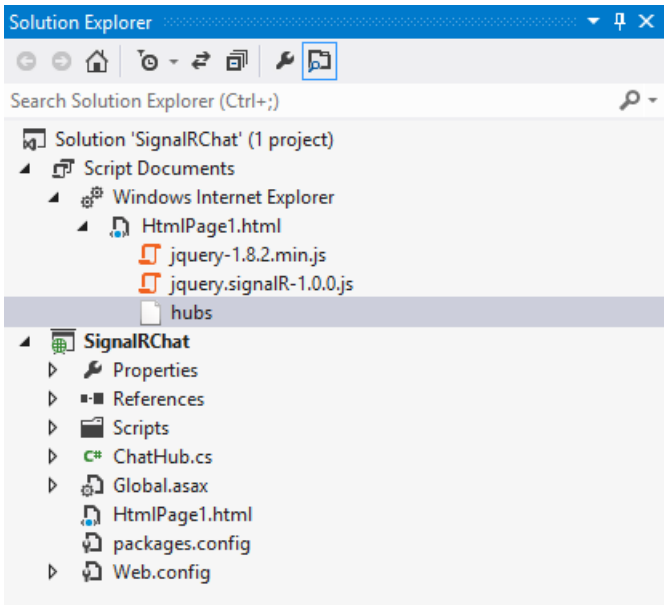
4. In each browser instance, add a comment and click **Send**. The comments should display in all browser instances.

> **NOTE**
>
> This simple chat application does not maintain the discussion context on the server. The hub broadcasts comments to all current users. Users who join the chat later will see messages added from the time they join.

The following screen shot shows the chat application running in three browser instances, all of which are updated when one instance sends a message:



5. In **Solution Explorer**, inspect the **Script Documents** node for the running application. There is a script file named **hubs** that the SignalR library dynamically generates at runtime. This file manages the communication between jQuery script and server-side code.



Examine the Code

The SignalR chat application demonstrates two basic SignalR development tasks: creating a hub as the main coordination object on the server, and using the SignalR jQuery library to send and receive messages.

SignalR Hubs

In the code sample the **ChatHub** class derives from the **Microsoft.AspNet.SignalR.Hub** class. Deriving from the **Hub** class is a useful way to build a SignalR application. You can create public methods on your hub class and then access those methods by calling them from jQuery scripts in a web page.

In the chat code, clients call the **ChatHub.Send** method to send a new message. The hub in turn sends the message to all clients by calling **Clients.All.broadcastMessage**.

The **Send** method demonstrates several hub concepts :

- Declare public methods on a hub so that clients can call them.
- Use the **Microsoft.AspNet.SignalR.Hub.Clients** dynamic property to access all clients connected to this hub.
- Call a jQuery function on the client (such as the `broadcastMessage` function) to update clients.

```
public class ChatHub : Hub
{
    public void Send(string name, string message)
    {
        // Call the broadcastMessage method to update clients.
        Clients.All.broadcastMessage(name, message);
    }
}
```

SignalR and jQuery

The HTML page in the code sample shows how to use the SignalR jQuery library to communicate with a SignalR hub. The essential tasks in the code are declaring a proxy to reference the hub, declaring a function that the server can call to push content to clients, and starting a connection to send messages to the hub.

The following code declares a proxy for a hub.

```
var chat = $.connection.chatHub;
```

> **NOTE**
> In jQuery the reference to the server class and its members is in camel case. The code sample references the C# **ChatHub** class in jQuery as **chatHub**.

The following code is how you create a callback function in the script. The hub class on the server calls this function to push content updates to each client. The two lines that HTML encode the content before displaying it are optional and show a simple way to prevent script injection.

```
chat.client.broadcastMessage = function (name, message) {
        // Html encode display name and message.
        var encodedName = $('<div />').text(name).html();
        var encodedMsg = $('<div />').text(message).html();
        // Add the message to the page.
        $('#discussion').append('<li><strong>' + encodedName
            + '</strong>:  ' + encodedMsg + '</li>');
    };
```

The following code shows how to open a connection with the hub. The code starts the connection and then passes it a function to handle the click event on the **Send** button in the HTML page.

> **NOTE**
> This approach insures that the connection is established before the event handler executes.

```
$.connection.hub.start().done(function () {
        $('#sendmessage').click(function () {
            // Call the Send method on the hub.
            chat.server.send($('#displayname').val(), $('#message').val());
            // Clear text box and reset focus for next comment.
            $('#message').val('').focus();
        });
    });
```

Next Steps

You learned that SignalR is a framework for building real-time web applications. You also learned several SignalR development tasks: how to add SignalR to an ASP.NET application, how to create a hub class, and how to send and receive messages from the hub.

You can make the sample application in this tutorial or other SignalR applications available over the Internet by deploying them to a hosting provider. Microsoft offers free web hosting for up to 10 web sites in a free Windows Azure trial account. For a walkthrough on how to deploy the sample SignalR application, see Publish the SignalR Getting Started Sample as a Windows Azure Web Site. For detailed information about how to deploy a Visual Studio web project to a Windows Azure Web Site, see Deploying an ASP.NET Application to a Windows Azure Web Site. (Note: The WebSocket transport is not currently supported for Windows Azure Web Sites. When WebSocket transport is not available, SignalR uses the other available transports as described in the Transports section of the Introduction to SignalR topic.)

To learn more advanced SignalR developments concepts, visit the following sites for SignalR source code and resources:

- SignalR Project
- SignalR Github and Samples

- SignalR Wiki

Tutorial: Getting Started with SignalR 1.x and MVC 4

2/20/2020 • 8 minutes to read • Edit Online

by Patrick Fletcher, Tim Teebken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This tutorial shows how to use ASP.NET SignalR to create a real-time chat application. You will add SignalR to an MVC 4 application and create a chat view to send and display messages.

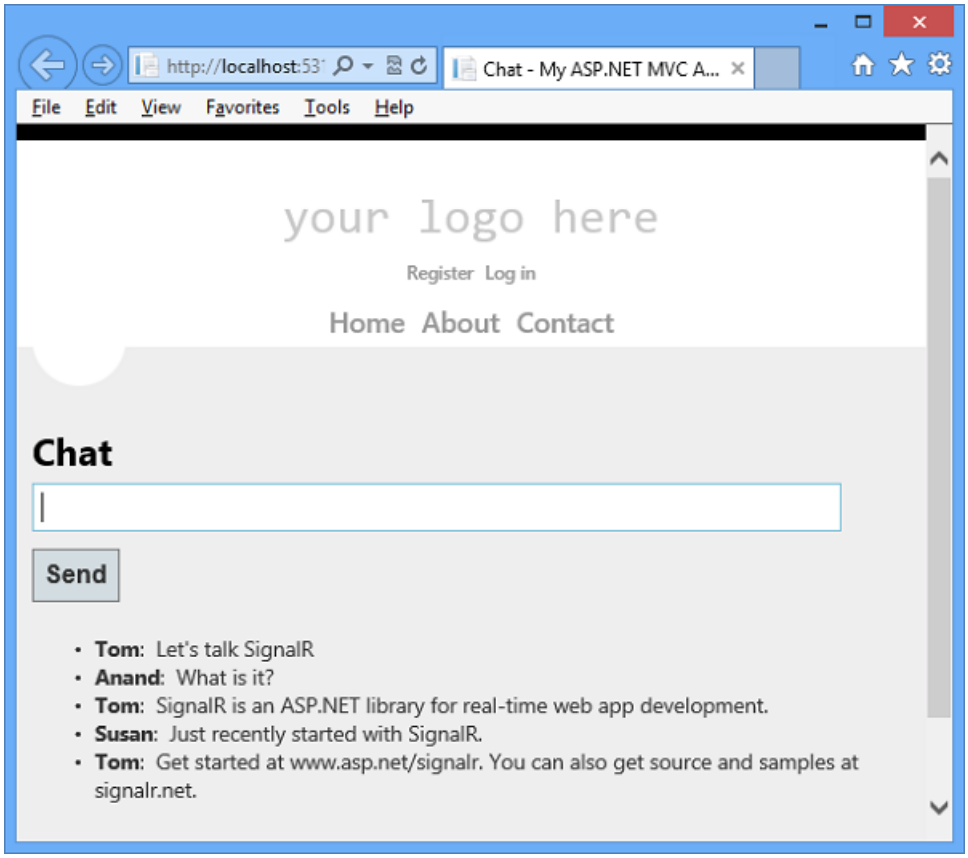
Overview

This tutorial introduces you to real-time web application development with ASP.NET SignalR and ASP.NET MVC 4. The tutorial uses the same chat application code as the SignalR Getting Started tutorial, but shows how to add it to an MVC 4 application based on the Internet template.

In this topic you will learn the following SignalR development tasks:

- Adding the SignalR library to an MVC 4 application.
- Creating a hub class to push content to clients.
- Using the SignalR jQuery library in a web page to send messages and display updates from the hub.

The following screen shot shows the completed chat application running in a browser.



Sections:

- Set up the Project
- Run the Sample
- Examine the Code
- Next Steps

Set up the Project

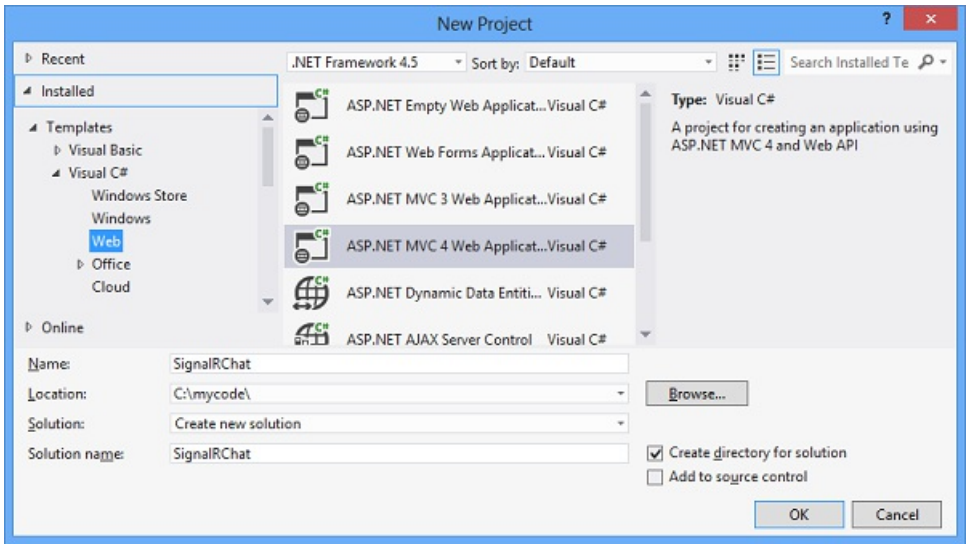
Prerequisites:

- Visual Studio 2010 SP1, Visual Studio 2012, or Visual Studio 2012 Express. If you do not have Visual Studio, see ASP.NET Downloads to get the free Visual Studio 2012 Express Development Tool.
- For Visual Studio 2010, install ASP.NET MVC 4.

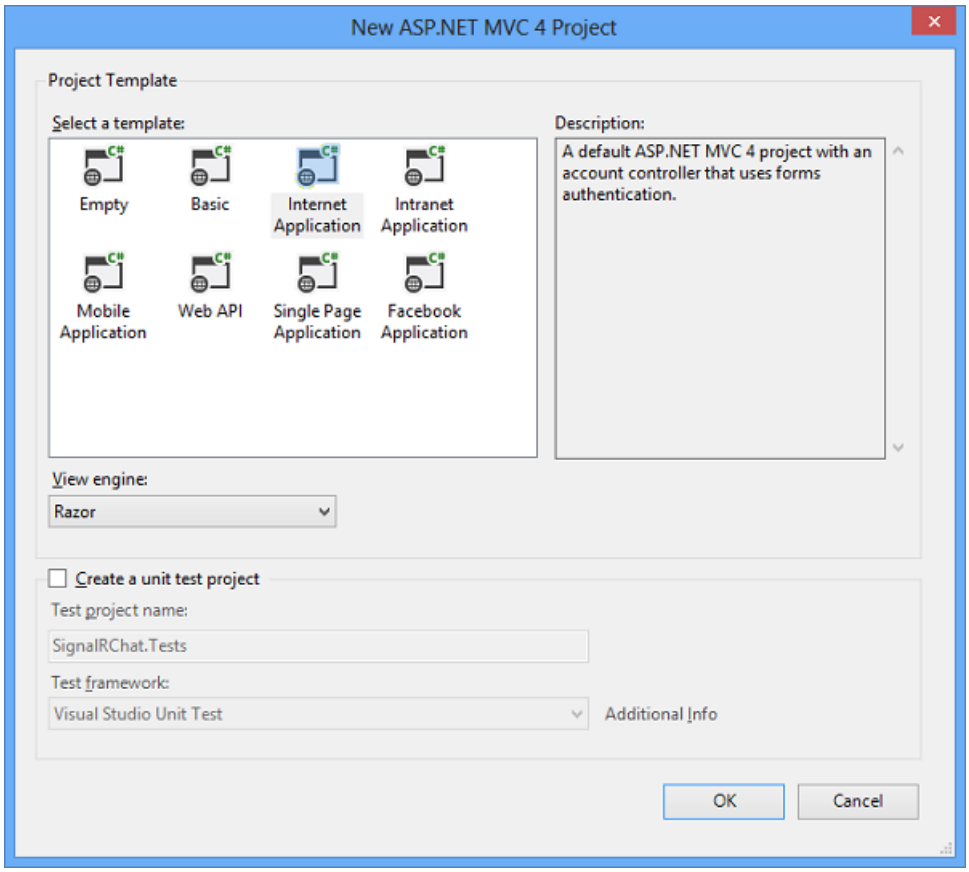
This section shows how to create an ASP.NET MVC 4 application, add the SignalR library, and create the chat application.

1. a. In Visual Studio create an ASP.NET MVC 4 application, name it SignalRChat, and click OK.

> **NOTE**
>
> In VS 2010, select **.NET Framework 4** in the Framework version dropdown control. SignalR code runs on .NET Framework versions 4 and 4.5.

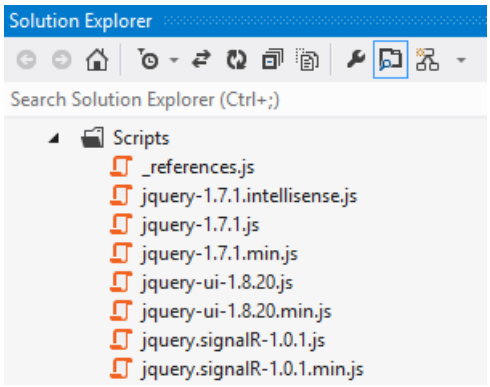


2. Select the Internet Application template, clear the option to **Create a unit test project**, and click OK.



3. Open the **Tools > NuGet Package Manager > Package Manager Console** and run the following command. This step adds to the project a set of script files and assembly references that enable SignalR functionality.

`install-package Microsoft.AspNet.SignalR -Version 1.1.3` 4. In **Solution Explorer** expand the Scripts folder. Note that script libraries for SignalR have been added to the project.



5. In **Solution Explorer**, right-click the project, select **Add | New Folder**, and add a new folder named **Hubs**. 6. Right-click the **Hubs** folder, click **Add | Class**, and create a new C# class named **ChatHub.cs**. You will use this class as a SignalR server hub that sends messages to all clients.

> **NOTE**
>
> If you use Visual Studio 2012 and have installed the ASP.NET and Web Tools 2012.2 update, you can use the new SignalR item template to create the hub class. To do that, right-click the **Hubs** folder, click **Add | New Item**, select **SignalR Hub Class (v1)**, and name the class **ChatHub.cs**.

1. Replace the code in the **ChatHub** class with the following code.

```
using System;
using System.Web;
using Microsoft.AspNet.SignalR;

namespace SignalRChat
{
    public class ChatHub : Hub
    {
        public void Send(string name, string message)
        {
            // Call the addNewMessageToPage method to update clients.
            Clients.All.addNewMessageToPage(name, message);
        }
    }
}
```

2. Open the **Global.asax** file for the project, and add a call to the method `RouteTable.Routes.MapHubs();` as the first line of code in the `Application_Start` method. This code registers the default route for SignalR hubs and must be called before you register any other routes. The completed `Application_Start` method looks like the following example.

```
protected void Application_Start()
{
    RouteTable.Routes.MapHubs();
    AreaRegistration.RegisterAllAreas();

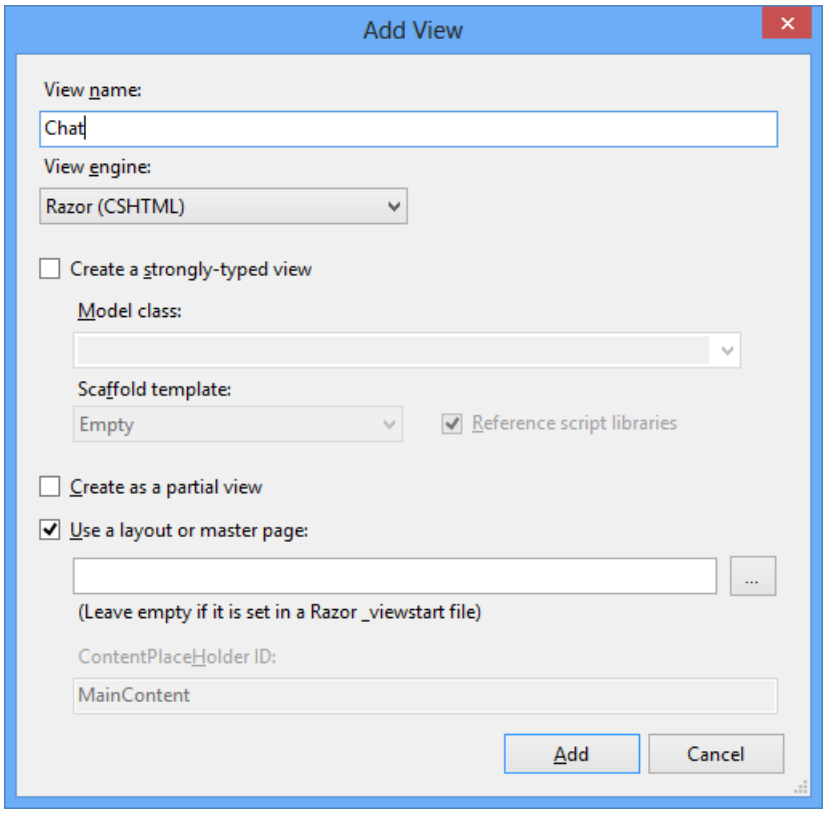
    WebApiConfig.Register(GlobalConfiguration.Configuration);
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
    AuthConfig.RegisterAuth();
}
```

3. Edit the `HomeController` class found in **Controllers/HomeController.cs** and add the following method to the class. This method returns the **Chat** view that you will create in a later step.

```
public ActionResult Chat()
{
    return View();
}
```

4. Right-click within the `Chat` method you just created, and click **Add View** to create a new view file.

5. In the **Add View** dialog, make sure the check box is selected to **Use a layout or master page** (clear the other check boxes), and then click **Add**.



6. Edit the new view file named **Chat.cshtml**. After the <h2> tag, paste the following section and `@section scripts` code block into the page. This script enables the page to send chat messages and display messages from the server. The complete code for the chat view appears in the following code block.

> **IMPORTANT**
>
> When you add SignalR and other script libraries to your Visual Studio project, the Package Manager might install versions of the scripts that are more recent than the versions shown in this topic. Make sure that the script references in your code match the versions of the script libraries installed in your project.

```
@{
    ViewBag.Title = "Chat";
}

<h2>Chat</h2>

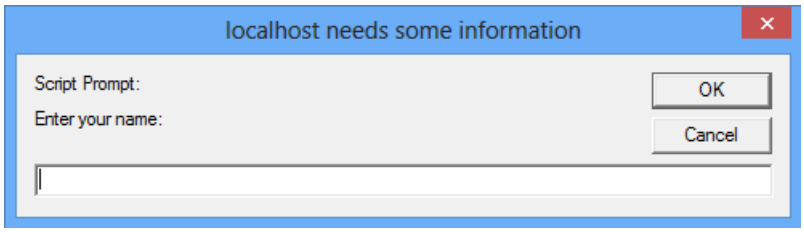
<div class="container">
    <input type="text" id="message" />
    <input type="button" id="sendmessage" value="Send" />
    <input type="hidden" id="displayname" />
    <ul id="discussion">
    </ul>
</div>

@section scripts {
    <!--Script references. -->
    <!--The jQuery library is required and is referenced by default in _Layout.cshtml. -->
    <!--Reference the SignalR library. -->
    <script src="~/Scripts/jquery.signalR-1.0.1.js"></script>
    <!--Reference the autogenerated SignalR hub script. -->
    <script src="~/signalr/hubs"></script>
    <!--SignalR script to update the chat page and send messages.-->
    <script>
        $(function () {
            // Reference the auto-generated proxy for the hub.
            var chat = $.connection.chatHub;
            // Create a function that the hub can call back to display messages.
            chat.client.addNewMessageToPage = function (name, message) {
                // Add the message to the page.
                $('#discussion').append('<li><strong>' + htmlEncode(name)
                    + '</strong>: ' + htmlEncode(message) + '</li>');
            };
            // Get the user name and store it to prepend to messages.
            $('#displayname').val(prompt('Enter your name:', ''));
            // Set initial focus to message input box.
            $('#message').focus();
            // Start the connection.
            $.connection.hub.start().done(function () {
                $('#sendmessage').click(function () {
                    // Call the Send method on the hub.
                    chat.server.send($('#displayname').val(), $('#message').val());
                    // Clear text box and reset focus for next comment.
                    $('#message').val('').focus();
                });
            });
        });
        // This optional function html-encodes messages for display in the page.
        function htmlEncode(value) {
            var encodedValue = $('<div />').text(value).html();
            return encodedValue;
        }
    </script>
}
```

7. **Save All** for the project.

Run the Sample

1. Press F5 to run the project in debug mode.
2. In the browser address line, append **/home/chat** to the URL of the default page for the project. The Chat page loads in a browser instance and prompts for a user name.



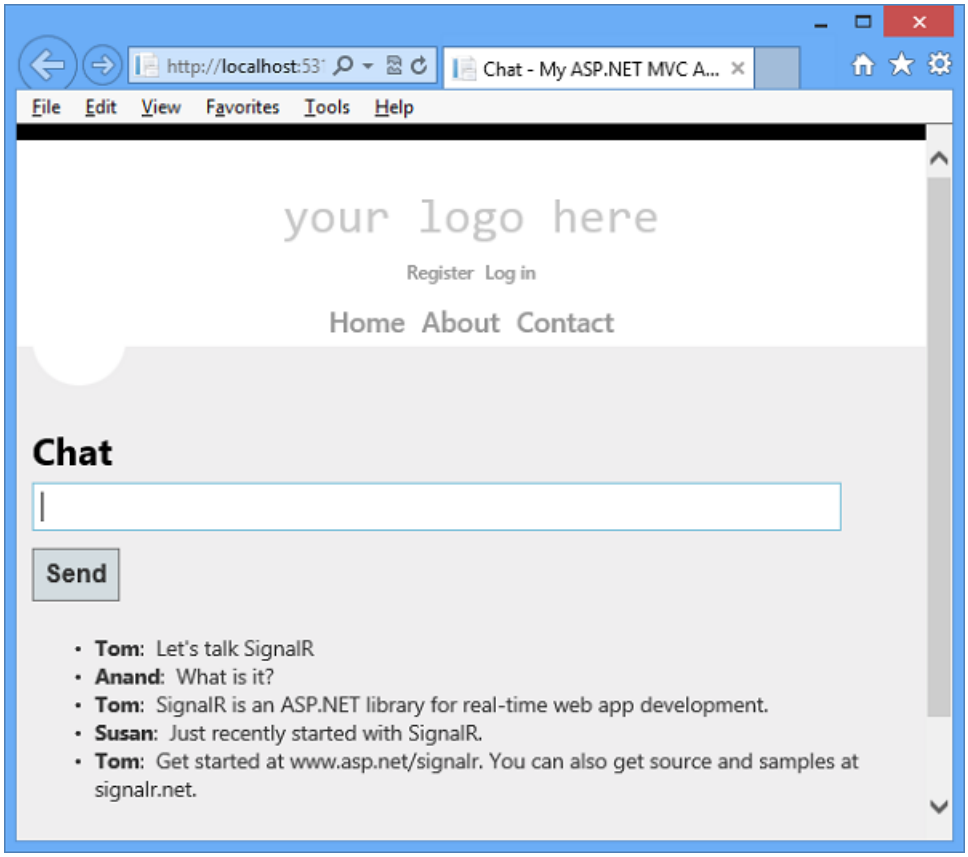
3. Enter a user name.

4. Copy the URL from the address line of the browser and use it to open two more browser instances. In each browser instance, enter a unique user name.

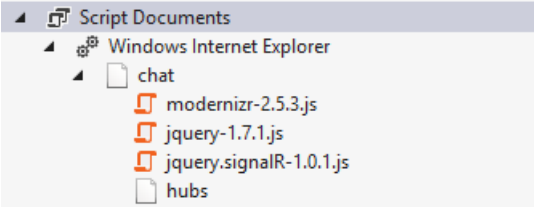
5. In each browser instance, add a comment and click **Send**. The comments should display in all browser instances.

> **NOTE**
>
> This simple chat application does not maintain the discussion context on the server. The hub broadcasts comments to all current users. Users who join the chat later will see messages added from the time they join.

6. The following screen shot shows the chat application running in a browser.



7. In **Solution Explorer**, inspect the **Script Documents** node for the running application. This node is visible in debug mode if you are using Internet Explorer as your browser. There is a script file named **hubs** that the SignalR library dynamically generates at runtime. This file manages the communication between jQuery script and server-side code. If you use a browser other than Internet Explorer, you can also access the dynamic **hubs** file by browsing to it directly, for example http://mywebsite/signalr/hubs.



Examine the Code

The SignalR chat application demonstrates two basic SignalR development tasks: creating a hub as the main coordination object on the server, and using the SignalR jQuery library to send and receive messages.

SignalR Hubs

In the code sample the **ChatHub** class derives from the **Microsoft.AspNet.SignalR.Hub** class. Deriving from the **Hub** class is a useful way to build a SignalR application. You can create public methods on your hub class and then access those methods by calling them from jQuery scripts in a web page.

In the chat code, clients call the **ChatHub.Send** method to send a new message. The hub in turn sends the message to all clients by calling **Clients.All.addNewMessageToPage**.

The **Send** method demonstrates several hub concepts :

- Declare public methods on a hub so that clients can call them.
- Use the **Microsoft.AspNet.SignalR.Hub.Clients** property to access all clients connected to this hub.
- Call a jQuery function on the client (such as the `addNewMessageToPage` function) to update clients.

```
public class ChatHub : Hub
{
    public void Send(string name, string message)
    {
        Clients.All.addNewMessageToPage(name, message);
    }
}
```

SignalR and jQuery

The **Chat.cshtml** view file in the code sample shows how to use the SignalR jQuery library to communicate with a SignalR hub. The essential tasks in the code are creating a reference to the auto-generated proxy for the hub, declaring a function that the server can call to push content to clients, and starting a connection to send messages to the hub.

The following code declares a proxy for a hub.

```
var chat = $.connection.chatHub;
```

> **NOTE**
> In jQuery the reference to the server class and its members is in camel case. The code sample references the C# **ChatHub** class in jQuery as **chatHub**. If you want to reference the `ChatHub` class in jQuery with conventional Pascal casing as you would in C#, edit the ChatHub.cs class file. Add a `using` statement to reference the `Microsoft.AspNet.SignalR.Hubs` namespace. Then add the `HubName` attribute to the `ChatHub` class, for example `[HubName("ChatHub")]`. Finally, update your jQuery reference to the `ChatHub` class.

The following code shows how to create a callback function in the script. The hub class on the server calls this

function to push content updates to each client. The optional call to the `htmlEncode` function shows a way to HTML encode the message content before displaying it in the page, as a way to prevent script injection.

```
chat.client.addNewMessageToPage = function (name, message) {
    // Add the message to the page.
    $('#discussion').append('<li><strong>' + htmlEncode(name)
        + '</strong>: ' + htmlEncode(message) + '</li>');
};
```

The following code shows how to open a connection with the hub. The code starts the connection and then passes it a function to handle the click event on the **Send** button in the Chat page.

> **NOTE**
>
> This approach ensures that the connection is established before the event handler executes.

```
$.connection.hub.start().done(function () {
    $('#sendmessage').click(function () {
        // Call the Send method on the hub.
        chat.server.send($('#displayname').val(), $('#message').val());
        // Clear text box and reset focus for next comment.
        $('#message').val('').focus();
    });
});
```

Next Steps

You learned that SignalR is a framework for building real-time web applications. You also learned several SignalR development tasks: how to add SignalR to an ASP.NET application, how to create a hub class, and how to send and receive messages from the hub.

To learn more advanced SignalR developments concepts, visit the following sites for SignalR source code and resources :

- SignalR Project
- SignalR Github and Samples
- SignalR Wiki

High-Frequency Realtime with SignalR 1.x

2/20/2020 • 13 minutes to read • Edit Online

by Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This tutorial shows how to create a web application that uses ASP.NET SignalR to provide high-frequency messaging functionality. High-frequency messaging in this case means updates that are sent at a fixed rate; in the case of this application, up to 10 messages a second.

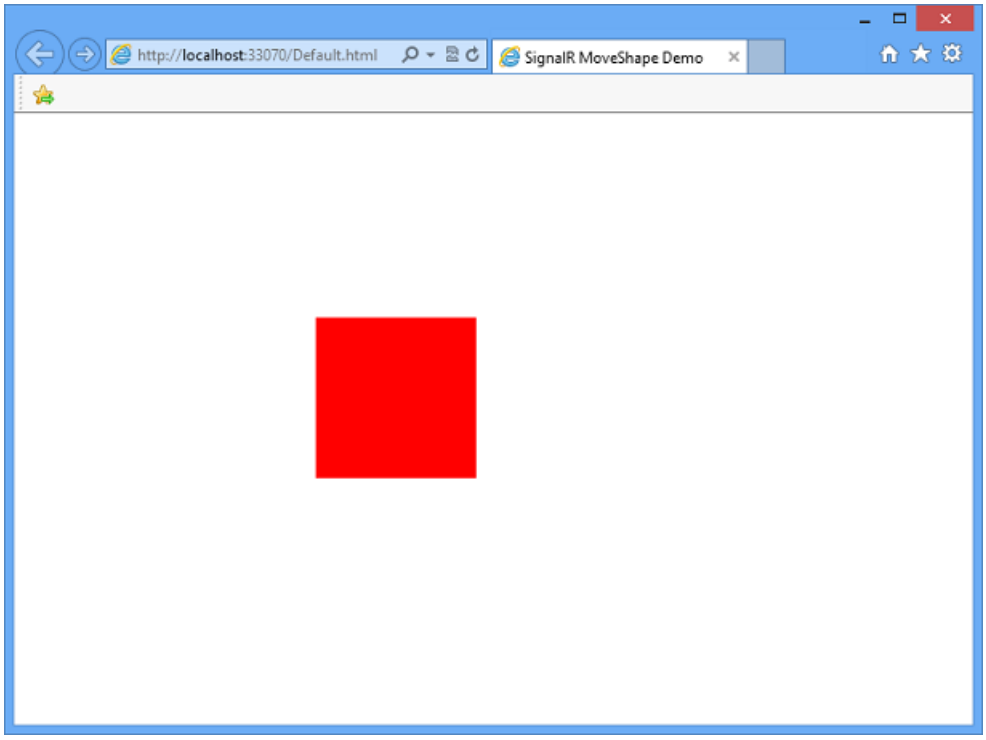
The application you'll create in this tutorial displays a shape that users can drag. The position of the shape in all other connected browsers will then be updated to match the position of the dragged shape using timed updates.

Concepts introduced in this tutorial have applications in real-time gaming and other simulation applications.

Comments on the tutorial are welcome. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Overview

This tutorial demonstrates how to create an application that shares the state of an object with other browsers in real time. The application we'll create is called MoveShape. The MoveShape page will display an HTML Div element that the user can drag; when the user drags the Div, its new position will be sent to the server, which will then tell all other connected clients to update the shape's position to match.



The application created in this tutorial is based on a demo by Damian Edwards. A video containing this demo can be seen here.

The tutorial will start by demonstrating how to send SignalR messages from each event that fires as the shape is dragged. Each connected client will then update the position of the local version of the shape each time a message is received.

While the application will function using this method, this is not a recommended programming model, since there would be no upper limit to the number of messages getting sent, so the clients and server could get overwhelmed with messages and performance would degrade. The displayed animation on the client would also be disjointed, as the shape would be moved instantly by each method, rather than moving smoothly to each new location. Later sections of the tutorial will demonstrate how to create a timer function that restricts the maximum rate at which messages are sent by either the client or server, and how to move the shape smoothly between locations. The final version of the application created in this tutorial can be downloaded from Code Gallery.

This tutorial contains the following sections:

- Prerequisites
- Create the project
- Add the ASP.NET SignalR and JQuery.UI NuGet packages
- Create the base application
- Add the client loop
- Add the server loop
- Add smooth animation on the client
- Further Steps

Prerequisites

This tutorial requires Visual Studio 2012 or Visual Studio 2010. If Visual Studio 2010 is used, the project will use .NET Framework 4 rather than .NET Framework 4.5.

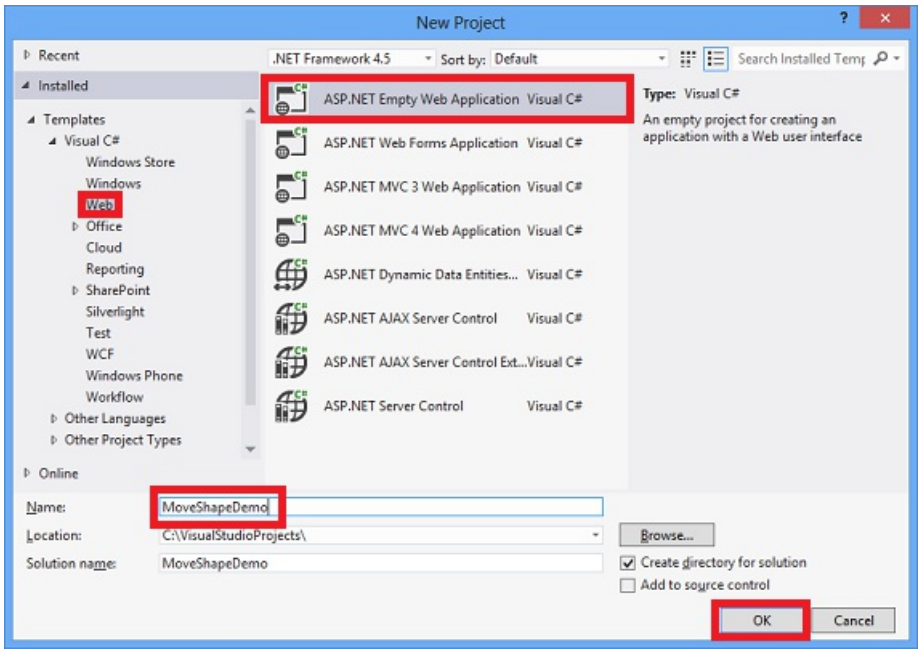
If you are using Visual Studio 2012, it's recommended that you install the ASP.NET and Web Tools 2012.2 update. This update contains new features such as enhancements to publishing, new functionality, and new templates.

If you have Visual Studio 2010, make sure that NuGet is installed.

Create the project

In this section, we'll create the project in Visual Studio.

1. From the **File** menu click **New Project**.
2. In the **New Project** dialog box, expand **C#** under **Templates** and select **Web**.
3. Select the **ASP.NET Empty Web Application** template, name the project *MoveShapeDemo*, and click **OK**.



Add the SignalR and JQuery.UI NuGet Packages

You can add SignalR functionality to a project by installing a NuGet package. This tutorial will also use the JQuery.UI package for allowing the shape to be dragged and animated.

1. Click **Tools | NuGet Package Manager | Package Manager Console**.

2. Enter the following command in the package manager.

   ```
   Install-Package Microsoft.AspNet.SignalR -Version 1.1.3
   ```

   The SignalR package installs a number of other NuGet packages as dependencies. When the installation is finished you have all of the server and client components required to use SignalR in an ASP.NET application.

3. Enter the following command into the package manager console to install the JQuery and JQuery.UI packages.

   ```
   Install-Package jQuery.ui.combined
   ```

Create the base application

In this section, we'll create a browser application that sends the location of the shape to the server during each mouse move event. The server then broadcasts this information to all other connected clients as it is received. We'll expand on this application in later sections.

1. In **Solution Explorer**, right-click on the project and select **Add**, **Class...**. Name the class **MoveShapeHub** and click **Add**.

2. Replace the code in the new **MoveShapeHub** class with the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNet.SignalR;
using Microsoft.AspNet.SignalR.Hubs;
using Newtonsoft.Json;

namespace MoveShapeDemo
{
    public class MoveShapeHub : Hub
    {
        public void UpdateModel(ShapeModel clientModel)
        {
            clientModel.LastUpdatedBy = Context.ConnectionId;
            // Update the shape model within our broadcaster
            Clients.AllExcept(clientModel.LastUpdatedBy).updateShape(clientModel);
        }
    }
    public class ShapeModel
    {
        // We declare Left and Top as lowercase with
        // JsonProperty to sync the client and server models
        [JsonProperty("left")]
        public double Left { get; set; }

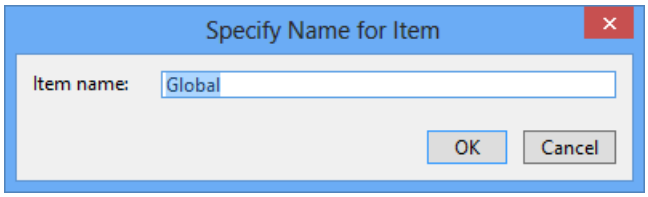
        [JsonProperty("top")]
        public double Top { get; set; }

        // We don't want the client to get the "LastUpdatedBy" property
        [JsonIgnore]
        public string LastUpdatedBy { get; set; }
    }
}
```

The `MoveShapeHub` class above is an implementation of a SignalR hub. As in the Getting Started with SignalR tutorial, the hub has a method that the clients will call directly. In this case, the client will send an object containing the new X and Y coordinates of the shape to the server, which then gets broadcasted to all other connected clients. SignalR will automatically serialize this object using JSON.

The object that will be sent to the client (`ShapeModel`) contains members to store the position of the shape. The version of the object on the server also contains a member to track which client's data is being stored, so that a given client won't be sent their own data. This member uses the `JsonIgnore` attribute to keep it from being serialized and sent to the client.

3. Next, we'll set up the hub when the application starts. In **Solution Explorer**, right-click the project, then click **Add | Global Application Class**. Accept the default name of *Global* and click **OK**.



4. Add the following `using` statement after the provided **using** statements in the Global.asax.cs class.

```
using System.Web.Routing;
```

5. Add the following line of code in the `Application_Start` method of the Global class to register the default route for SignalR.

```
RouteTable.Routes.MapHubs();
```

Your global.asax file should look like the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.SessionState;

using System.Web.Routing;

namespace MoveShapeDemo
{
    public class Global : System.Web.HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            RouteTable.Routes.MapHubs();
        }
    }
}
```

6. Next, we'll add the client. In **Solution Explorer**, right-click the project, then click **Add | New Item**. In the **Add New Item** dialog, select **Html Page**. Give the page an appropriate name (like **Default.html**) and click **Add**.

7. In **Solution Explorer**, right-click the page you just created and click **Set as Start Page**.

8. Replace the default code in the HTML page with the following code snippet.

> **NOTE**
>
> Verify that the script references below match the packages added to your project in the Scripts folder. In Visual Studio 2010, the version of JQuery and SignalR added to the project may not match the version numbers below.

```
<!DOCTYPE html>
<html>
<head>
    <title>SignalR MoveShape Demo</title>
    <style>
        #shape {
            width: 100px;
            height: 100px;
            background-color: #FF0000;
        }
    </style>
</head>
<body>
<script src="Scripts/jquery-1.6.4.js"></script>
<script src="Scripts/jquery-ui-1.10.2.js"></script>
<script src="Scripts/jquery.signalR-1.0.1.js"></script>
<script src="/signalr/hubs"></script>
<script>
 $(function () {
            var moveShapeHub = $.connection.moveShapeHub,
            $shape = $("#shape"),

            shapeModel = {
                left: 0,
                top: 0
            };

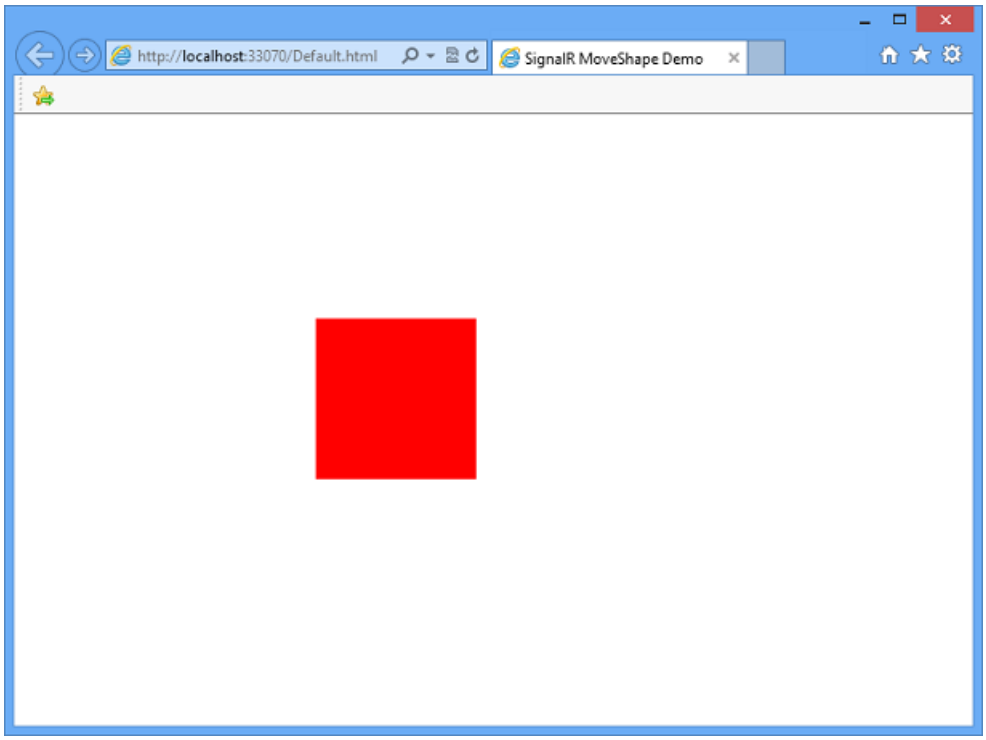
            moveShapeHub.client.updateShape = function (model) {
                shapeModel = model;
                $shape.css({ left: model.left, top: model.top });
            };

            $.connection.hub.start().done(function () {
                $shape.draggable({
                    drag: function () {
                        shapeModel = $shape.offset();
                        moveShapeHub.server.updateModel(shapeModel);
                    }
                });
            });
        });
</script>

    <div id="shape" />
</body>
</html>
```

The above HTML and JavaScript code creates a red Div called Shape, enables the shape's dragging behavior using the jQuery library, and uses the shape's `drag` event to send the shape's position to the server.

9. Start the application by pressing F5. Copy the page's URL, and paste it into a second browser window. Drag the shape in one of the browser windows; the shape in the other browser window should move.



Add the client loop

Since sending the location of the shape on every mouse move event will create an unnecessary amount of network traffic, the messages from the client need to be throttled. We'll use the javascript `setInterval` function to set up a loop that sends new position information to the server at a fixed rate. This loop is a very basic representation of a "game loop", a repeatedly called function that drives all of the functionality of a game or other simulation.

1. Update the client code in the HTML page to match the following code snippet.

```
<!DOCTYPE html>
<html>
<head>
    <title>SignalR MoveShape Demo</title>
    <style>
        #shape {
            width: 100px;
            height: 100px;
            background-color: #FF0000;
        }
    </style>

</head>
<body>
<script src="Scripts/jquery-1.6.4.js"></script>
<script src="Scripts/jquery-ui-1.10.2.js"></script>
<script src="Scripts/jquery.signalR-1.0.1.js"></script>
<script src="/signalr/hubs"></script>
<script>
        $(function () {
            var moveShapeHub = $.connection.moveShapeHub,
                $shape = $("#shape"),
                // Send a maximum of 10 messages per second
                // (mouse movements trigger a lot of messages)
                messageFrequency = 10,
                // Determine how often to send messages in
                // time to abide by the messageFrequency
                updateRate = 1000 / messageFrequency,
                shapeModel = {
                    left: 0,
                    top: 0
                },
                moved = false;

            moveShapeHub.client.updateShape = function (model) {
                shapeModel = model;
                $shape.css({ left: model.left, top: model.top });
            };

            $.connection.hub.start().done(function () {
                $shape.draggable({
                    drag: function () {
                        shapeModel = $shape.offset();
                        moved = true;
                    }
                });

                // Start the client side server update interval
                setInterval(updateServerModel, updateRate);
            });

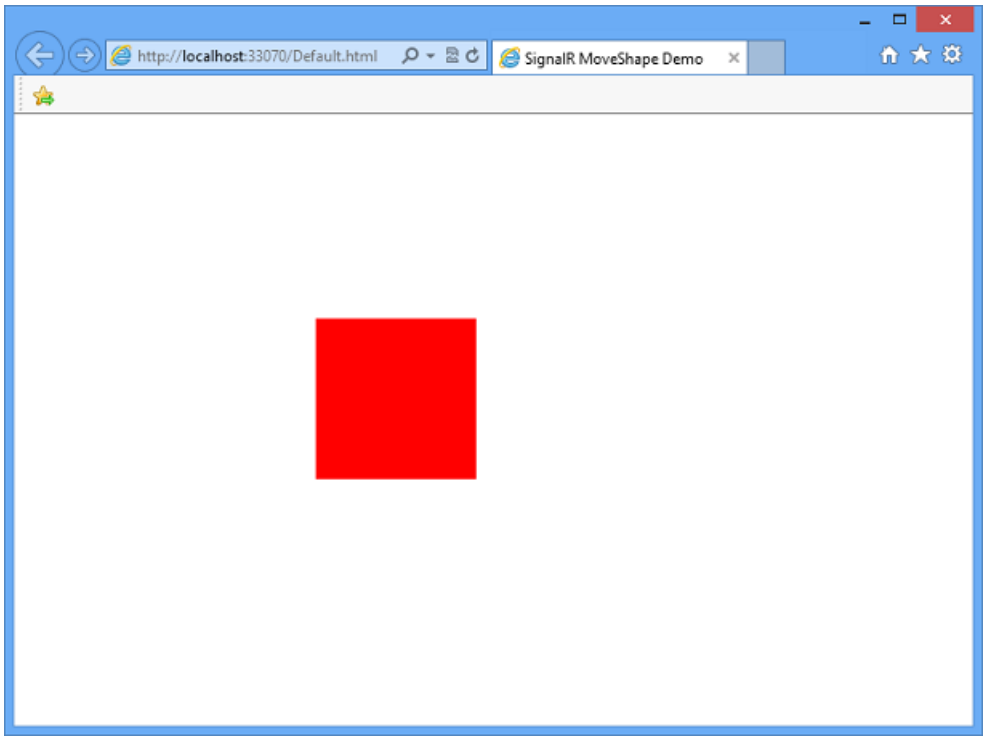
            function updateServerModel() {
                // Only update server if we have a new movement
                if (moved) {
                    moveShapeHub.server.updateModel(shapeModel);
                    moved = false;
                }
            }
        });
</script>

    <div id="shape" />
</body>
</html>
```

The above update adds the `updateServerModel` function, which gets called on a fixed frequency. This

function sends the position data to the server whenever the `moved` flag indicates that there is new position data to send.

2. Start the application by pressing F5. Copy the page's URL, and paste it into a second browser window. Drag the shape in one of the browser windows; the shape in the other browser window should move. Since the number of messages that get sent to the server will be throttled, the animation will not appear as smooth as in the previous section.



Add the server loop

In the current application, messages sent from the server to the client go out as often as they are received. This presents a similar problem as was seen on the client; messages can be sent more often than they are needed, and the connection could become flooded as a result. This section describes how to update the server to implement a timer that throttles the rate of the outgoing messages.

1. Replace the contents of `MoveShapeHub.cs` with the following code snippet.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

using System.Threading;
using Microsoft.AspNet.SignalR;

using Newtonsoft.Json;

namespace MoveShapeDemo
{
    public class Broadcaster
    {
        private readonly static Lazy<Broadcaster> _instance =
            new Lazy<Broadcaster>(() => new Broadcaster());
        // We're going to broadcast to all clients a maximum of 25 times per second
        private readonly TimeSpan BroadcastInterval =
            TimeSpan.FromMilliseconds(40);
        private readonly IHubContext _hubContext;
        private Timer _broadcastLoop;
        private ShapeModel _model;
```

```
        private bool _modelUpdated;

        public Broadcaster()
        {
            // Save our hub context so we can easily use it
            // to send to its connected clients
            _hubContext = GlobalHost.ConnectionManager.GetHubContext<MoveShapeHub>();

            _model = new ShapeModel();
            _modelUpdated = false;

            // Start the broadcast loop
            _broadcastLoop = new Timer(
                BroadcastShape,
                null,
                BroadcastInterval,
                BroadcastInterval);
        }

        public void BroadcastShape(object state)
        {
            // No need to send anything if our model hasn't changed
            if (_modelUpdated)
            {
                // This is how we can access the Clients property
                // in a static hub method or outside of the hub entirely
                _hubContext.Clients.AllExcept(_model.LastUpdatedBy).updateShape(_model);
                _modelUpdated = false;
            }
        }

        public void UpdateShape(ShapeModel clientModel)
        {
            _model = clientModel;
            _modelUpdated = true;
        }

        public static Broadcaster Instance
        {
            get
            {
                return _instance.Value;
            }
        }
    }

    public class MoveShapeHub : Hub
    {
        // Is set via the constructor on each creation
        private Broadcaster _broadcaster;

        public MoveShapeHub()
            : this(Broadcaster.Instance)
        {
        }

        public MoveShapeHub(Broadcaster broadcaster)
        {
            _broadcaster = broadcaster;
        }

        public void UpdateModel(ShapeModel clientModel)
        {
            clientModel.LastUpdatedBy = Context.ConnectionId;
            // Update the shape model within our broadcaster
            _broadcaster.UpdateShape(clientModel);
        }
    }
    public class ShapeModel
```

```
    {
        // We declare Left and Top as lowercase with
        // JsonProperty to sync the client and server models
        [JsonProperty("left")]
        public double Left { get; set; }

        [JsonProperty("top")]
        public double Top { get; set; }

        // We don't want the client to get the "LastUpdatedBy" property
        [JsonIgnore]
        public string LastUpdatedBy { get; set; }
    }

}
```

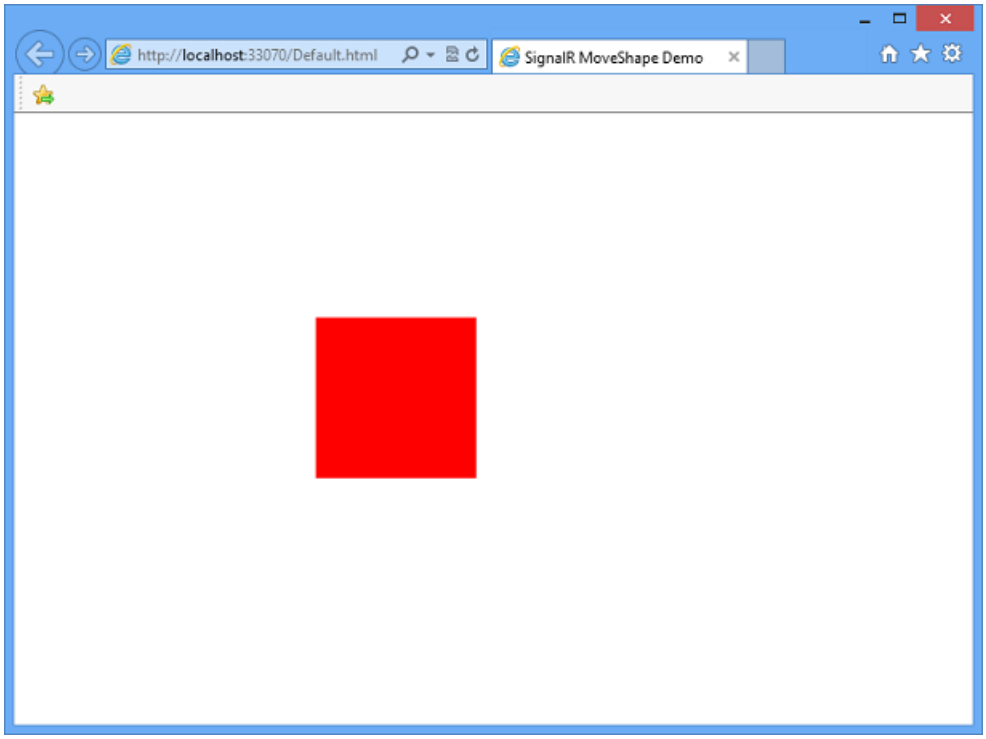
The above code expands the client to add the `Broadcaster` class, which throttles the outgoing messages using the `Timer` class from the .NET framework.

Since the hub itself is transitory (it is created every time it is needed), the `Broadcaster` will be created as a singleton. Lazy initialization (introduced in .NET 4) is used to defer its creation until it is needed, ensuring that the first hub instance is completely created before the timer is started.

The call to the clients' `UpdateShape` function is then moved out of the hub's `UpdateModel` method, so that it is no longer called immediately whenever incoming messages are received. Instead, the messages to the clients will be sent at a rate of 25 calls per second, managed by the `_broadcastLoop` timer from within the `Broadcaster` class.

Lastly, instead of calling the client method from the hub directly, the `Broadcaster` class needs to obtain a reference to the currently operating hub (`_hubContext`) using the `GlobalHost`.

2. Start the application by pressing F5. Copy the page's URL, and paste it into a second browser window. Drag the shape in one of the browser windows; the shape in the other browser window should move. There will not be a visible difference in the browser from the previous section, but the number of messages that get sent to the client will be throttled.



Add smooth animation on the client

The application is almost complete, but we could make one more improvement, in the motion of the shape on the client as it is moved in response to server messages. Rather than setting the position of the shape to the new location given by the server, we'll use the JQuery UI library's `animate` function to move the shape smoothly between its current and new position.

1. Update the client's `updateShape` method to look like the highlighted code below:

```
<!DOCTYPE html>
<html>
<head>
    <title>SignalR MoveShape Demo</title>
    <style>
        #shape {
            width: 100px;
            height: 100px;
            background-color: #FF0000;
        }
    </style>

</head>
<body>
<script src="Scripts/jquery-1.6.4.js"></script>
<script src="Scripts/jquery-ui-1.10.2.js"></script>
<script src="Scripts/jquery.signalR-1.0.1.js"></script>
<script src="/signalr/hubs"></script>
<script>
        $(function () {
            var moveShapeHub = $.connection.moveShapeHub,
                $shape = $("#shape"),
                // Send a maximum of 10 messages per second
                // (mouse movements trigger a lot of messages)
                messageFrequency = 10,
                // Determine how often to send messages in
                // time to abide by the messageFrequency
                updateRate = 1000 / messageFrequency,
                shapeModel = {
                    left: 0,
                    top: 0
                },
                moved = false;

             moveShapeHub.client.updateShape = function (model) {
                 shapeModel = model;
                 // Gradually move the shape towards the new location (interpolate)
                 // The updateRate is used as the duration because by the time
                 // we get to the next location we want to be at the "last" location
                 // We also clear the animation queue so that we start a new
                 // animation and don't lag behind.
                 $shape.animate(shapeModel, { duration: updateRate, queue: false });
            };

            $.connection.hub.start().done(function () {
                $shape.draggable({
                    drag: function () {
                        shapeModel = $shape.offset();
                        moved = true;
                    }
                });

                // Start the client side server update interval
                setInterval(updateServerModel, updateRate);
            });

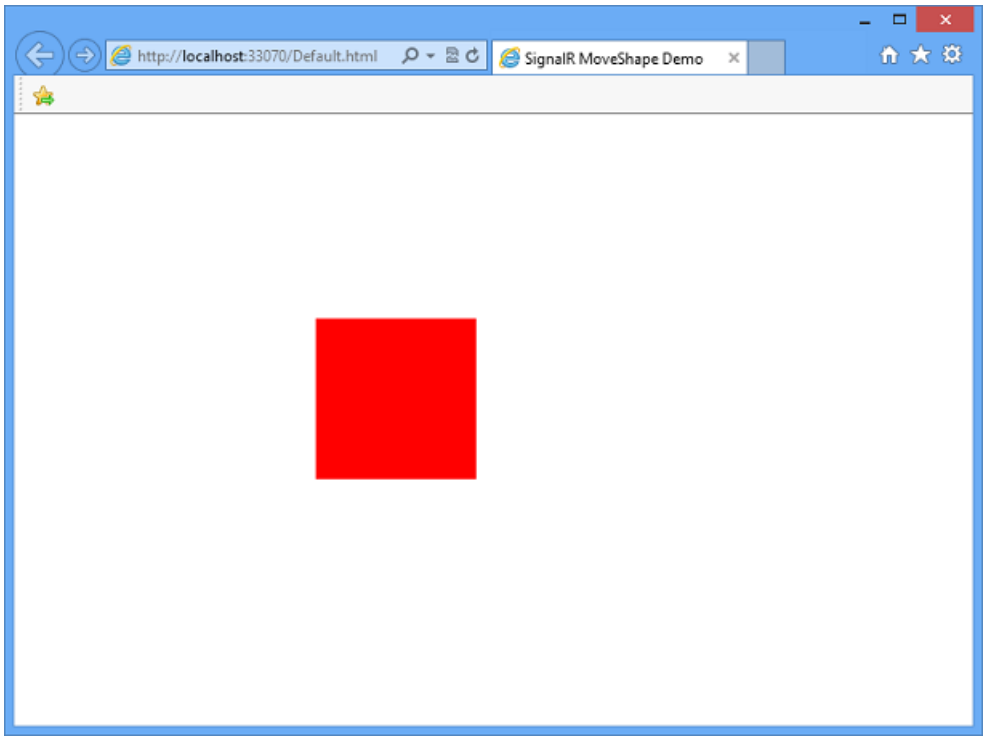
            function updateServerModel() {
                // Only update server if we have a new movement
                if (moved) {
                    moveShapeHub.server.updateModel(shapeModel);
```

```
                    moved = false;
                }
            }
        });
</script>

    <div id="shape" />
</body>
</html>
```

The above code moves the shape from the old location to the new one given by the server over the course of the animation interval (in this case, 100 milliseconds). Any previous animation running on the shape is cleared before the new animation starts.

2. Start the application by pressing F5. Copy the page's URL, and paste it into a second browser window. Drag the shape in one of the browser windows; the shape in the other browser window should move. The movement of the shape in the other window should appear less jerky as its movement is interpolated over time rather than being set once per incoming message.



Further Steps

In this tutorial, you've learned how to program a SignalR application that sends high-frequency messages between clients and servers. This communication paradigm is useful for developing online games and other simulations, such as the ShootR game created with SignalR.

The complete application created in this tutorial can be downloaded from Code Gallery.

To learn more about SignalR development concepts, visit the following sites for SignalR source code and resources:

- SignalR Project
- SignalR Github and Samples
- SignalR Wiki

Tutorial: Server Broadcast with ASP.NET SignalR 1.x

2/20/2020 • 27 minutes to read • Edit Online

by Patrick Fletcher, Tom Dykstra

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This tutorial shows how to create a web application that uses ASP.NET SignalR to provide server broadcast functionality. Server broadcast means that communications sent to clients are initiated by the server. This scenario requires a different programming approach than peer-to-peer scenarios such as chat applications, in which communications sent to clients are initiated by one or more of the clients.

The application that you'll create in this tutorial simulates a stock ticker, a typical scenario for server broadcast functionality.

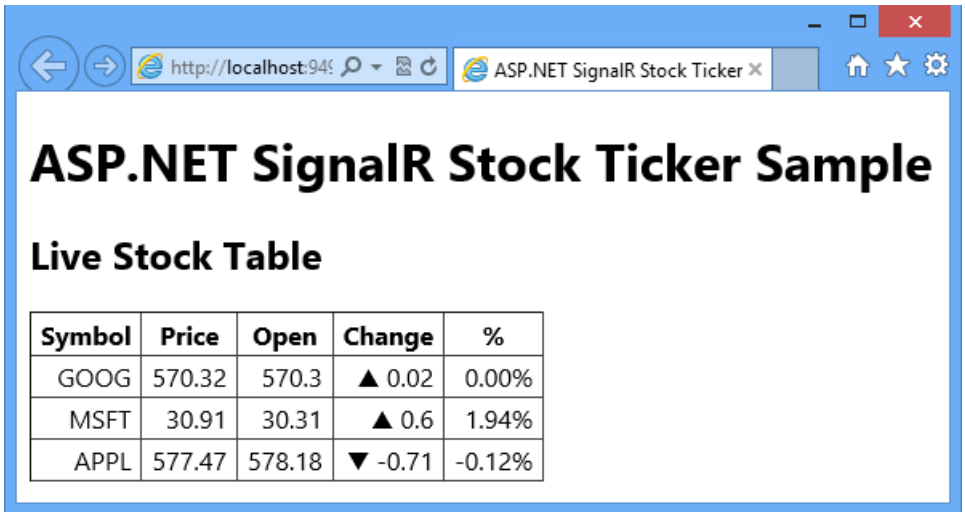
Comments on the tutorial are welcome. If you have questions that are not directly related to the tutorial, you can post them to the ASP.NET SignalR forum or StackOverflow.com.

Overview

The Microsoft.AspNet.SignalR.Sample NuGet package installs a sample simulated stock ticker application in a Visual Studio project. In the first part of this tutorial, you'll create a simplified version of that application from scratch. In the remainder of the tutorial, you'll install the NuGet package and review the additional features and code that it creates.

The stock ticker application is a representative of a kind of real-time application in which you want to periodically "push," or broadcast, notifications from the server to all connected clients.

The application that you'll build in the first part of this tutorial displays a grid with stock data.



Periodically the server randomly updates stock prices and pushes the updates to all connected clients. In the browser the numbers and symbols in the **Change** and **%** columns dynamically change in response to notifications from the server. If you open additional browsers to the same URL, they all show the same data and the same changes to the data simultaneously.

This tutorial contains the following sections:

- Prerequisites
- Create the project
- Add the SignalR NuGet packages
- Set up the server code
- Set up the client code
- Test the application
- Enable logging
- Install and review the full StockTicker sample
- Next steps

> **NOTE**
>
> If you don't want to work through the steps of building the application, you can install the SignalR.Sample package in a new **Empty ASP.NET Web Application** project, and read through these steps to get explanations of the code. The first part of the tutorial covers a subset of the SignalR.Sample code, and the second part explains key features of the additional functionality in the SignalR.Sample package.

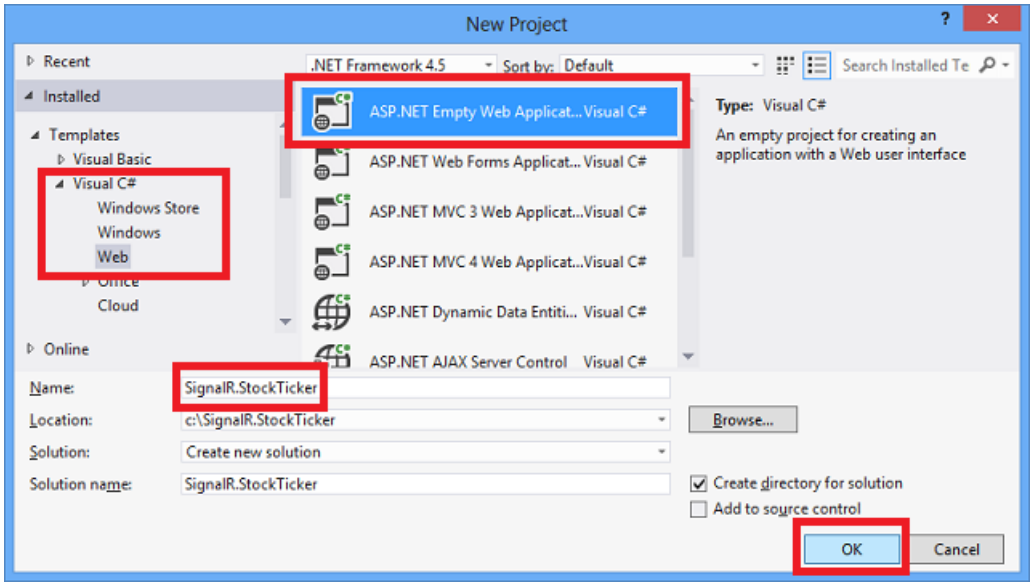
Prerequisites

Before you start, make sure that you have Visual Studio 2012 or 2010 SP1 installed on your computer. If you don't have Visual Studio, see ASP.NET Downloads to get the free Visual Studio 2012 Express for Web.

If you have Visual Studio 2010, make sure that NuGet is installed.

Create the project

1. From the **File** menu click **New Project**.
2. In the **New Project** dialog box, expand **C#** under **Templates** and select **Web**.
3. Select the **ASP.NET Empty Web Application** template, name the project *SignalR.StockTicker*, and click **OK**.



Add the SignalR NuGet Packages

Add the SignalR and JQuery NuGet Packages

You can add SignalR functionality to a project by installing a NuGet package.

1. Click **Tools | NuGet Package Manager | Package Manager Console**.
2. Enter the following command in the package manager.

   ```
   Install-Package Microsoft.AspNet.SignalR -Version 1.1.3
   ```

   The SignalR package installs a number of other NuGet packages as dependencies. When the installation is finished you have all of the server and client components required to use SignalR in an ASP.NET application.

Set up the server code

In this section you set up the code that runs on the server.

Create the Stock class

You begin by creating the Stock model class that you'll use to store and transmit information about a stock.

1. Create a new class file in the project folder, name it *Stock.cs*, and then replace the template code with the following code:

```
using System;

namespace SignalR.StockTicker
{
    public class Stock
    {
        private decimal _price;

        public string Symbol { get; set; }

        public decimal Price
        {
            get
            {
                return _price;
            }
            set
            {
                if (_price == value)
                {
                    return;
                }

                _price = value;

                if (DayOpen == 0)
                {
                    DayOpen = _price;
                }
            }
        }

        public decimal DayOpen { get; private set; }

        public decimal Change
        {
            get
            {
                return Price - DayOpen;
            }
        }

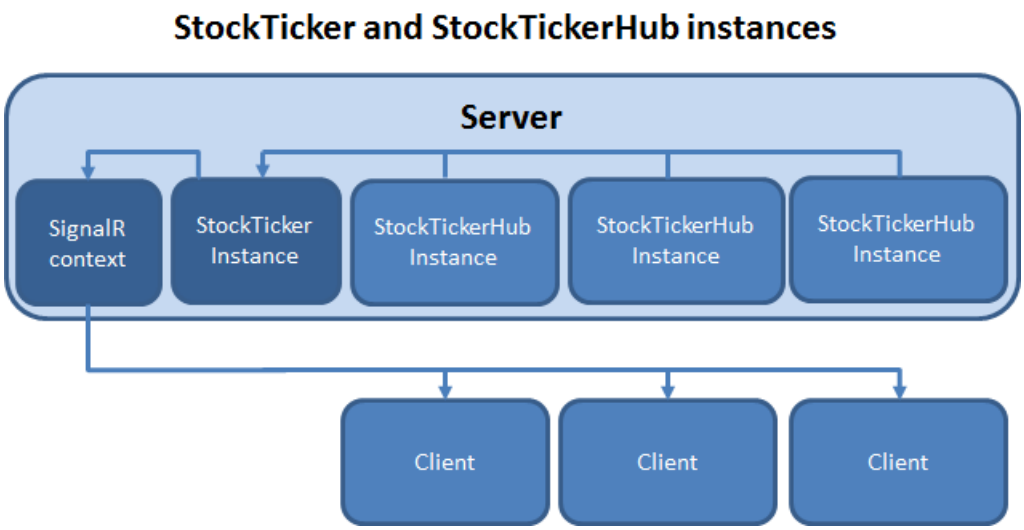
        public double PercentChange
        {
            get
            {
                return (double)Math.Round(Change / Price, 4);
            }
        }
    }
}
```

The two properties that you'll set when you create stocks are the Symbol (for example, MSFT for Microsoft) and the Price. The other properties depend on how and when you set Price. The first time you set Price, the value gets propagated to DayOpen. Subsequent times when you set Price, the Change and PercentChange property values are calculated based on the difference between Price and DayOpen.

Create the StockTicker and StockTickerHub classes

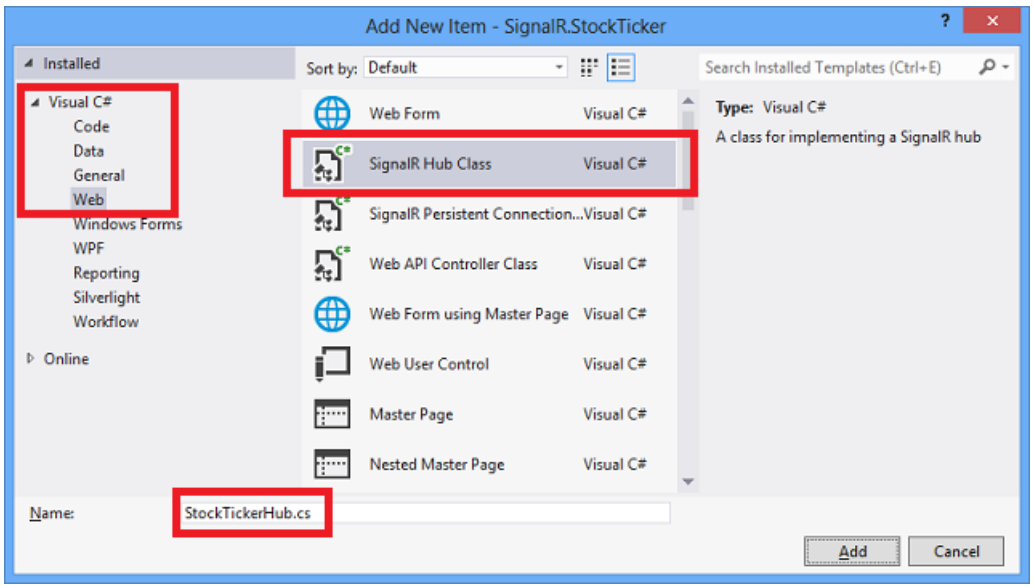
You'll use the SignalR Hub API to handle server-to-client interaction. A StockTickerHub class that derives from the SignalR Hub class will handle receiving connections and method calls from clients. You also need to maintain stock data and run a Timer object to periodically trigger price updates, independently of client connections. You can't put these functions in a Hub class, because Hub instances are transient. A Hub class instance is created for each operation on the hub, such as connections and calls from the client to the server. So the mechanism that keeps

stock data, updates prices, and broadcasts the price updates has to run in a separate class, which you'll name StockTicker.



You only want one instance of the StockTicker class to run on the server, so you'll need to set up a reference from each StockTickerHub instance to the singleton StockTicker instance. The StockTicker class has to be able to broadcast to clients because it has the stock data and triggers updates, but StockTicker is not a Hub class. Therefore, the StockTicker class has to get a reference to the SignalR Hub connection context object. It can then use the SignalR connection context object to broadcast to clients.

1. In **Solution Explorer**, right-click the project and click **Add New Item**.
2. If you have Visual Studio 2012 with the ASP.NET and Web Tools 2012.2 Update, click **Web** under **Visual C#** and select the **SignalR Hub Class** item template. Otherwise, select the **Class** template.
3. Name the new class *StockTickerHub.cs*, and then click **Add**.



4. Replace the template code with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Microsoft.AspNet.SignalR;
using Microsoft.AspNet.SignalR.Hubs;

namespace SignalR.StockTicker
{
    [HubName("stockTickerMini")]
    public class StockTickerHub : Hub
    {
        private readonly StockTicker _stockTicker;

        public StockTickerHub() : this(StockTicker.Instance) { }

        public StockTickerHub(StockTicker stockTicker)
        {
            _stockTicker = stockTicker;
        }

        public IEnumerable<Stock> GetAllStocks()
        {
            return _stockTicker.GetAllStocks();
        }
    }
}
```

The Hub class is used to define methods the clients can call on the server. You are defining one method: `GetAllStocks()`. When a client initially connects to the server, it will call this method to get a list of all of the stocks with their current prices. The method can execute synchronously and return `IEnumerable<Stock>` because it is returning data from memory. If the method had to get the data by doing something that would involve waiting, such as a database lookup or a web service call, you would specify `Task<IEnumerable<Stock>>` as the return value to enable asynchronous processing. For more information, see ASP.NET SignalR Hubs API Guide - Server - When to execute asynchronously.

The HubName attribute specifies how the Hub will be referenced in JavaScript code on the client. The default name on the client if you don't use this attribute is a camel-cased version of the class name, which in this case would be stockTickerHub.

As you'll see later when you create the StockTicker class, a singleton instance of that class is created in its static Instance property. That singleton instance of StockTicker remains in memory no matter how many clients connect or disconnect, and that instance is what the GetAllStocks method uses to return current stock information.

5. Create a new class file in the project folder, name it *StockTicker.cs*, and then replace the template code with the following code:

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Threading;
using Microsoft.AspNet.SignalR;
using Microsoft.AspNet.SignalR.Hubs;

namespace SignalR.StockTicker
{
    public class StockTicker
    {
        // Singleton instance
        private readonly static Lazy<StockTicker> _instance = new Lazy<StockTicker>(() => new
StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>().Clients));
```

```
        private readonly ConcurrentDictionary<string, Stock> _stocks = new ConcurrentDictionary<string,
Stock>();

        private readonly object _updateStockPricesLock = new object();

        //stock can go up or down by a percentage of this factor on each change
        private readonly double _rangePercent = .002;

        private readonly TimeSpan _updateInterval = TimeSpan.FromMilliseconds(250);
        private readonly Random _updateOrNotRandom = new Random();

        private readonly Timer _timer;
        private volatile bool _updatingStockPrices = false;

        private StockTicker(IHubConnectionContext clients)
        {
            Clients = clients;

            _stocks.Clear();
            var stocks = new List<Stock>
            {
                new Stock { Symbol = "MSFT", Price = 30.31m },
                new Stock { Symbol = "APPL", Price = 578.18m },
                new Stock { Symbol = "GOOG", Price = 570.30m }
            };
            stocks.ForEach(stock => _stocks.TryAdd(stock.Symbol, stock));

            _timer = new Timer(UpdateStockPrices, null, _updateInterval, _updateInterval);

        }

        public static StockTicker Instance
        {
            get
            {
                return _instance.Value;
            }
        }

        private IHubConnectionContext Clients
        {
            get;
            set;
        }

        public IEnumerable<Stock> GetAllStocks()
        {
            return _stocks.Values;
        }

        private void UpdateStockPrices(object state)
        {
            lock (_updateStockPricesLock)
            {
                if (!_updatingStockPrices)
                {
                    _updatingStockPrices = true;

                    foreach (var stock in _stocks.Values)
                    {
                        if (TryUpdateStockPrice(stock))
                        {
                            BroadcastStockPrice(stock);
                        }
                    }

                    _updatingStockPrices = false;
                }
```

```
            }
        }

        private bool TryUpdateStockPrice(Stock stock)
        {
            // Randomly choose whether to udpate this stock or not
            var r = _updateOrNotRandom.NextDouble();
            if (r > .1)
            {
                return false;
            }

            // Update the stock price by a random factor of the range percent
            var random = new Random((int)Math.Floor(stock.Price));
            var percentChange = random.NextDouble() * _rangePercent;
            var pos = random.NextDouble() > .51;
            var change = Math.Round(stock.Price * (decimal)percentChange, 2);
            change = pos ? change : -change;

            stock.Price += change;
            return true;
        }

        private void BroadcastStockPrice(Stock stock)
        {
            Clients.All.updateStockPrice(stock);
        }

    }
}
```

Since multiple threads will be running the same instance of StockTicker code, the StockTicker class has to be threadsafe.

Storing the singleton instance in a static field

The code initializes the static _instance field that backs the Instance property with an instance of the class, and this is the only instance of the class that can be created, because the constructor is marked as private. Lazy initialization is used for the _instance field, not for performance reasons but to ensure that the instance creation is threadsafe.

```
private readonly static Lazy<StockTicker> _instance = new Lazy<StockTicker>(() => new
StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>().Clients));

public static StockTicker Instance
{
    get
    {
        return _instance.Value;
    }
}
```

Each time a client connects to the server, a new instance of the StockTickerHub class running in a separate thread gets the StockTicker singleton instance from the StockTicker.Instance static property, as you saw earlier in the StockTickerHub class.

Storing stock data in a ConcurrentDictionary

The constructor initializes the _stocks collection with some sample stock data, and GetAllStocks returns the stocks. As you saw earlier, this collection of stocks is in turn returned by StockTickerHub.GetAllStocks which is a server method in the Hub class that clients can call.

```
private readonly ConcurrentDictionary<string, Stock> _stocks = new ConcurrentDictionary<string, Stock>
();
```

```
private StockTicker(IHubConnectionContext clients)
{
    Clients = clients;

    _stocks.Clear();
    var stocks = new List<Stock>
    {
        new Stock { Symbol = "MSFT", Price = 30.31m },
        new Stock { Symbol = "APPL", Price = 578.18m },
        new Stock { Symbol = "GOOG", Price = 570.30m }
    };
    stocks.ForEach(stock => _stocks.TryAdd(stock.Symbol, stock));

    _timer = new Timer(UpdateStockPrices, null, _updateInterval, _updateInterval);
}

public IEnumerable<Stock> GetAllStocks()
{
    return _stocks.Values;
}
```

The stocks collection is defined as a ConcurrentDictionary type for thread safety. As an alternative, you could use a Dictionary object and explicitly lock the dictionary when you make changes to it.

For this sample application, it's OK to store application data in memory and to lose the data when the StockTicker instance is disposed. In a real application you would work with a back-end data store such as a database.

Periodically updating stock prices

The constructor starts up a Timer object that periodically calls methods that update stock prices on a random basis.

```
_timer = new Timer(UpdateStockPrices, null, _updateInterval, _updateInterval);

private void UpdateStockPrices(object state)
{
    lock (_updateStockPricesLock)
    {
        if (!_updatingStockPrices)
        {
            _updatingStockPrices = true;

            foreach (var stock in _stocks.Values)
            {
                if (TryUpdateStockPrice(stock))
                {
                    BroadcastStockPrice(stock);
                }
            }

            _updatingStockPrices = false;
        }
    }
}

private bool TryUpdateStockPrice(Stock stock)
{
    // Randomly choose whether to update this stock or not
    var r = _updateOrNotRandom.NextDouble();
    if (r > .1)
    {
        return false;
    }

    // Update the stock price by a random factor of the range percent
    var random = new Random((int)Math.Floor(stock.Price));
    var percentChange = random.NextDouble() * _rangePercent;
    var pos = random.NextDouble() > .51;
    var change = Math.Round(stock.Price * (decimal)percentChange, 2);
    change = pos ? change : -change;

    stock.Price += change;
    return true;
}
```

UpdateStockPrices is called by the Timer, which passes in null in the state parameter. Before updating prices, a lock is taken on the _updateStockPricesLock object. The code checks if another thread is already updating prices, and then it calls TryUpdateStockPrice on each stock in the list. The TryUpdateStockPrice method decides whether to change the stock price, and how much to change it. If the stock price is changed, BroadcastStockPrice is called to broadcast the stock price change to all connected clients.

The _updatingStockPrices flag is marked as volatile to ensure that access to it is threadsafe.

```
private volatile bool _updatingStockPrices = false;
```

In a real application, the TryUpdateStockPrice method would call a web service to look up the price; in this code it uses a random number generator to make changes randomly.

Getting the SignalR context so that the StockTicker class can broadcast to clients

Because the price changes originate here in the StockTicker object, this is the object that needs to call an updateStockPrice method on all connected clients. In a Hub class you have an API for calling client methods, but StockTicker does not derive from the Hub class and does not have a reference to any Hub object. Therefore, in order to broadcast to connected clients, the StockTicker class has to get the SignalR context

instance for the StockTickerHub class and use that to call methods on clients.

The code gets a reference to the SignalR context when it creates the singleton class instance, passes that reference to the constructor, and the constructor puts it in the Clients property.

There are two reasons why you want to get the context just once: getting the context is an expensive operation, and getting it once ensures that the intended order of messages sent to clients is preserved.

```
private readonly static Lazy<StockTicker> _instance =
    new Lazy<StockTicker>(() => new
StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>().Clients));

private StockTicker(IHubConnectionContext clients)
{
    Clients = clients;

    // Remainder of constructor ...
}

private IHubConnectionContext Clients
{
    get;
    set;
}

private void BroadcastStockPrice(Stock stock)
{
    Clients.All.updateStockPrice(stock);
}
```

Getting the Clients property of the context and putting it in the StockTickerClient property lets you write code to call client methods that looks the same as it would in a Hub class. For instance, to broadcast to all clients you can write Clients.All.updateStockPrice(stock).

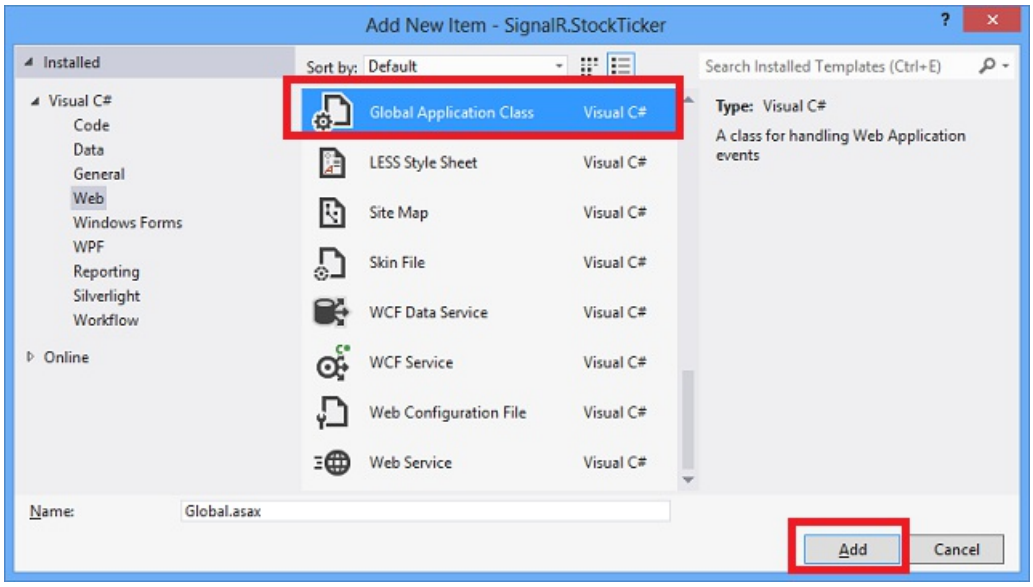
The updateStockPrice method that you are calling in BroadcastStockPrice doesn't exist yet; you'll add it later when you write code that runs on the client. You can refer to updateStockPrice here because Clients.All is dynamic, which means the expression will be evaluated at runtime. When this method call executes, SignalR will send the method name and the parameter value to the client, and if the client has a method named updateStockPrice, that method will be called and the parameter value will be passed to it.

Clients.All means send to all clients. SignalR gives you other options to specify which clients or groups of clients to send to. For more information, see HubConnectionContext.

Register the SignalR route

The server needs to know which URL to intercept and direct to SignalR. To do that you'll add some code to the *Global.asax* file.

1. In **Solution Explorer**, right-click the project, and then click **Add New Item**.
2. Select the **Global Application Class** item template, and then click **Add**.



3. Add the SignalR route registration code to the Application_Start method:

```
protected void Application_Start(object sender, EventArgs e)
{
    RouteTable.Routes.MapHubs();
}
```

By default, the base URL for all SignalR traffic is "/signalr", and "/signalr/hubs" is used to retrieve a dynamically generated JavaScript file that defines proxies for all the Hubs you have in your application. The MapHubs method includes overloads that let you specify a different base URL and certain SignalR options in an instance of the HubConfiguration class.

4. Add a using statement at the top of the file:

```
using System.Web.Routing;
```

5. Save and close the *Global.asax* file, and build the project.

You have now completed setting up the server code. In the next section you'll set up the client.

Set up the client code

1. Create a new HTML file in the project folder, and name it *StockTicker.html*.

2. Replace the template code with the following code:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>ASP.NET SignalR Stock Ticker</title>
    <style>
        body {
            font-family: 'Segoe UI', Arial, Helvetica, sans-serif;
            font-size: 16px;
        }
        #stockTable table {
            border-collapse: collapse;
        }
            #stockTable table th, #stockTable table td {
                padding: 2px 6px;
            }
            #stockTable table td {
                text-align: right;
            }
        #stockTable .loading td {
            text-align: left;
        }
    </style>
</head>
<body>
    <h1>ASP.NET SignalR Stock Ticker Sample</h1>

    <h2>Live Stock Table</h2>
    <div id="stockTable">
        <table border="1">
            <thead>
                <tr><th>Symbol</th><th>Price</th><th>Open</th><th>Change</th><th>%</th></tr>
            </thead>
            <tbody>
                <tr class="loading"><td colspan="5">loading...</td></tr>
            </tbody>
        </table>
    </div>

    <!--Script references. -->
    <!--Reference the jQuery library. -->
    <script src="/Scripts/jquery-1.8.2.min.js" ></script>
    <!--Reference the SignalR library. -->
    <script src="/Scripts/jquery.signalR-1.0.1.js"></script>
    <!--Reference the autogenerated SignalR hub script. -->
    <script src="/signalr/hubs"></script>
    <!--Reference the StockTicker script. -->
    <script src="StockTicker.js"></script>
</body>
</html>
```

The HTML creates a table with 5 columns, a header row, and a data row with a single cell that spans all 5 columns. The data row displays "loading..." and will only be shown momentarily when the application starts. JavaScript code will remove that row and add in its place rows with stock data retrieved from the server.

The script tags specify the jQuery script file, the SignalR core script file, the SignalR proxies script file, and a StockTicker script file that you'll create later. The SignalR proxies script file, which specifies the "/signalr/hubs" URL, is dynamically generated and defines proxy methods for the methods on the Hub class, in this case for StockTickerHub.GetAllStocks. If you prefer, you can generate this JavaScript file manually by using SignalR Utilities and disable dynamic file creation in the MapHubs method call.

3. **IMPORTANT**

   Make sure that the JavaScript file references in *StockTicker.html* are correct. That is, make sure that the jQuery version in your script tag (1.8.2 in the example) is the same as the jQuery version in your project's *Scripts* folder, and make sure that the SignalR version in your script tag is the same as the SignalR version in your project's *Scripts* folder. Change the file names in the script tags if necessary.

4. In **Solution Explorer**, right-click *StockTicker.html*, and then click **Set as Start Page**.

5. Create a new JavaScript file in the project folder and name it *StockTicker.js*..

6. Replace the template code with the following code:

```
// A simple templating method for replacing placeholders enclosed in curly braces.
if (!String.prototype.supplant) {
    String.prototype.supplant = function (o) {
        return this.replace(/{([^{}]*)}/g,
            function (a, b) {
                var r = o[b];
                return typeof r === 'string' || typeof r === 'number' ? r : a;
            }
        );
    };
}

$(function () {

    var ticker = $.connection.stockTickerMini, // the generated client-side hub proxy
        up = '▲',
        down = '▼',
        $stockTable = $('#stockTable'),
        $stockTableBody = $stockTable.find('tbody'),
        rowTemplate = '<tr data-symbol="{Symbol}"><td>{Symbol}</td><td>{Price}</td><td>{DayOpen}</td>
<td>{Direction} {Change}</td><td>{PercentChange}</td></tr>';

    function formatStock(stock) {
        return $.extend(stock, {
            Price: stock.Price.toFixed(2),
            PercentChange: (stock.PercentChange * 100).toFixed(2) + '%',
            Direction: stock.Change === 0 ? '' : stock.Change >= 0 ? up : down
        });
    }

    function init() {
        ticker.server.getAllStocks().done(function (stocks) {
            $stockTableBody.empty();
            $.each(stocks, function () {
                var stock = formatStock(this);
                $stockTableBody.append(rowTemplate.supplant(stock));
            });
        });
    }

    // Add a client-side hub method that the server will call
    ticker.client.updateStockPrice = function (stock) {
        var displayStock = formatStock(stock),
            $row = $(rowTemplate.supplant(displayStock));

        $stockTableBody.find('tr[data-symbol=' + stock.Symbol + ']')
            .replaceWith($row);
        }

    // Start the connection
    $.connection.hub.start().done(init);

});
```

$.connection refers to the SignalR proxies. The code gets a reference to the proxy for the StockTickerHub class and puts it in the ticker variable. The proxy name is the name that was set by the [HubName] attribute:

```
var ticker = $.connection.stockTickerMini
```

```
[HubName("stockTickerMini")]
public class StockTickerHub : Hub
```

After all the variables and functions are defined, the last line of code in the file initializes the SignalR

connection by calling the SignalR start function. The start function executes asynchronously and returns a jQuery Deferred object, which means you can call the done function to specify the function to call when the asynchronous operation is completed..

```
$.connection.hub.start().done(init);
```

The init function calls the getAllStocks function on the server and uses the information that the server returns to update the stock table. Notice that by default, you have to use camel casing on the client although the method name is pascal-cased on the server. The camel-casing rule only applies to methods, not objects. For example, you refer to stock.Symbol and stock.Price, not stock.symbol or stock.price.

```
function init() {
    ticker.server.getAllStocks().done(function (stocks) {
        $stockTableBody.empty();
        $.each(stocks, function () {
            var stock = formatStock(this);
            $stockTableBody.append(rowTemplate.supplant(stock));
        });
    });
}
```

```
public IEnumerable<Stock> GetAllStocks()
{
    return _stockTicker.GetAllStocks();
}
```

If you wanted to use pascal casing on the client, or if you wanted to use a completely different method name, you could decorate the Hub method with the HubMethodName attribute the same way you decorated the Hub class itself with the HubName attribute.

In the init method, HTML for a table row is created for each stock object received from the server by calling formatStock to format properties of the stock object, and then by calling supplant (which is defined at the top of *StockTicker.js*) to replace placeholders in the rowTemplate variable with the stock object property values. The resulting HTML is then appended to the stock table.

You call init by passing it in as a callback function that executes after the asynchronous start function completes. If you called init as a separate JavaScript statement after calling start, the function would fail because it would execute immediately without waiting for the start function to finish establishing the connection. In that case, the init function would try to call the getAllStocks function before the server connection is established.

When the server changes a stock's price, it calls the updateStockPrice on connected clients. The function is added to the client property of the stockTicker proxy in order to make it available to calls from the server.

```
ticker.client.updateStockPrice = function (stock) {
    var displayStock = formatStock(stock),
        $row = $(rowTemplate.supplant(displayStock));

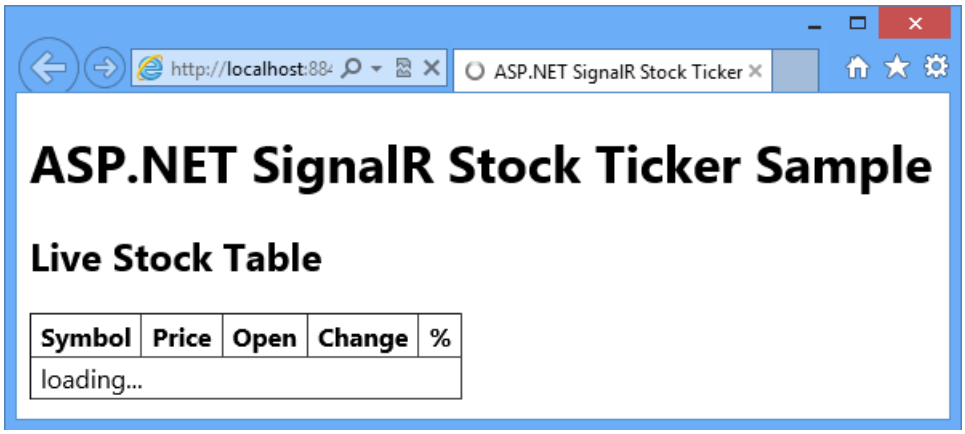
    $stockTableBody.find('tr[data-symbol=' + stock.Symbol + ']')
        .replaceWith($row);
    }
```

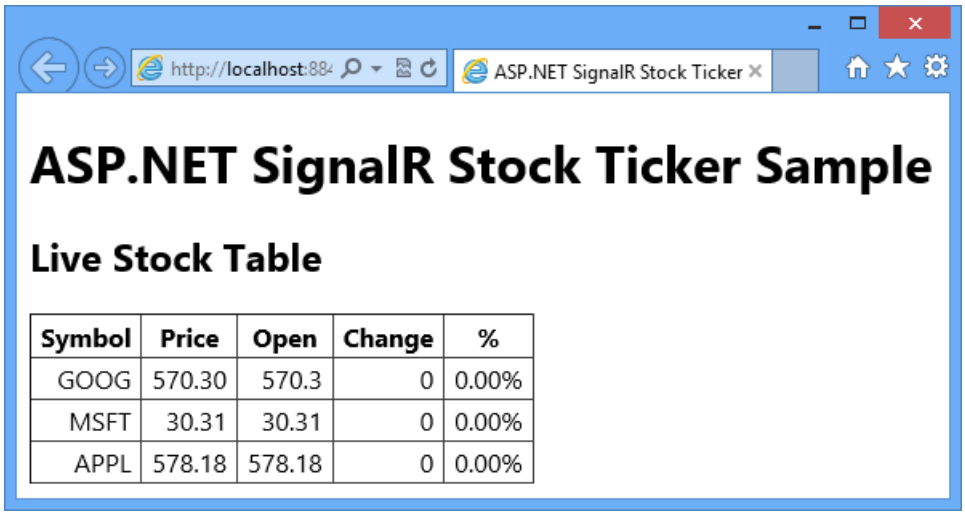
The updateStockPrice function formats a stock object received from the server into a table row the same way as in the init function. However, instead of appending the row to the table, it finds the stock's current row in the table and replaces that row with the new one.

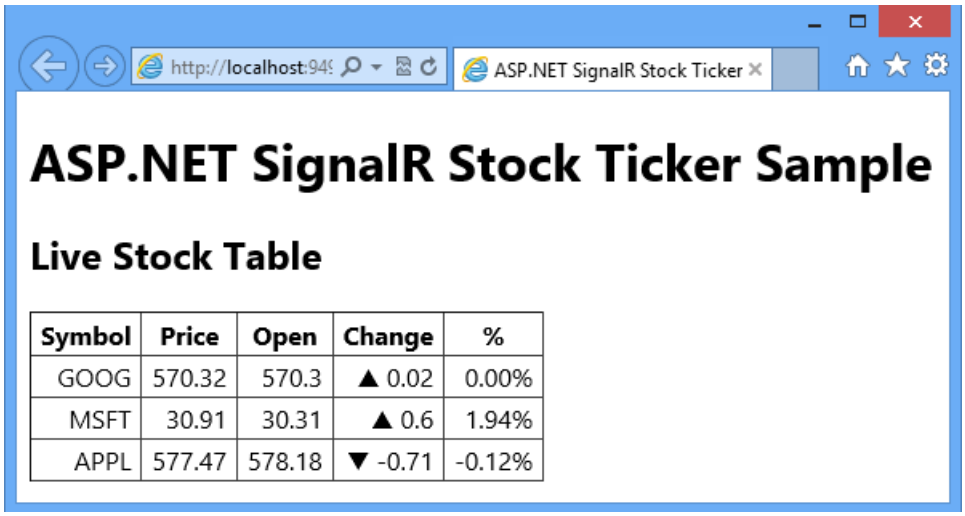
Test the application

1. Press F5 to run the application in debug mode.

   The stock table initially displays the "loading..." line, then after a short delay the initial stock data is displayed, and then the stock prices start to change.







2. Copy the URL from the browser address bar and paste it into one or more new browser window(s).

   The initial stock display is the same as the first browser and changes happen simultaneously.

3. Close all browsers and open a new browser, then go to the same URL.

   The StockTicker singleton object has continued to run in the server, so the stock table display shows that the stocks have continued to change. (You don't see the initial table with zero change figures.)

4. Close the browser.

Enable logging

SignalR has a built-in logging function that you can enable on the client to aid in troubleshooting. In this section you enable logging and see examples that show how logs tell you which of the following transport methods SignalR is using:

- WebSockets, supported by IIS 8 and current browsers.
- Server-sent events, supported by browsers other than Internet Explorer.
- Forever frame, supported by Internet Explorer.
- Ajax long polling, supported by all browsers.

For any given connection, SignalR chooses the best transport method that both the server and the client support.

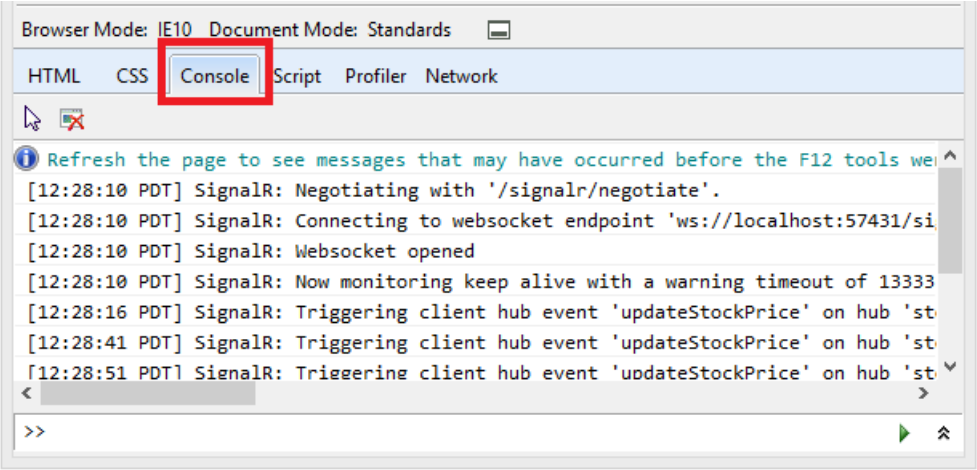
1. Open *StockTicker.js* and add a line of code to enable logging immediately before the code that initializes the connection at the end of the file:

```
// Start the connection
$.connection.hub.logging = true;
$.connection.hub.start().done(init);
```

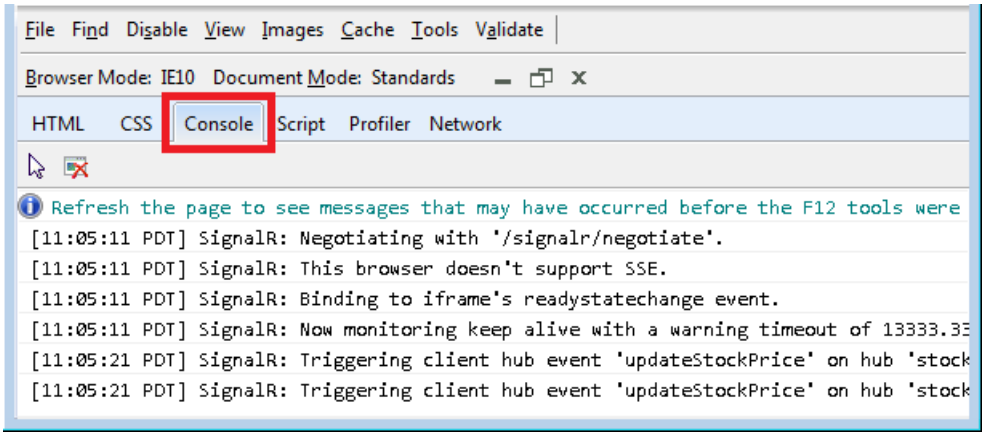
2. Press F5 to run the project.

3. Open your browser's developer tools window, and select the Console to see the logs. You might have to refresh the page to see the logs of Signalr negotiating the transport method for a new connection.

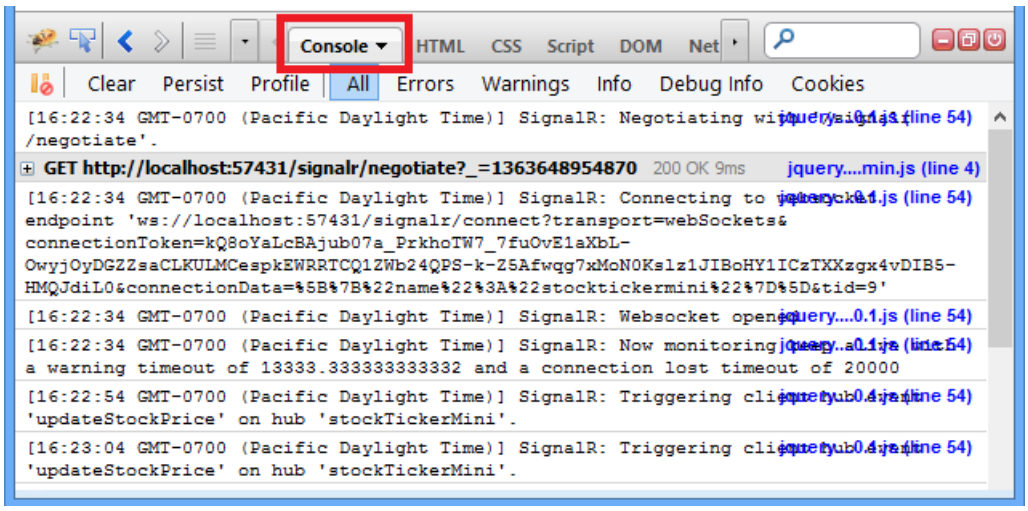
   If you are running Internet Explorer 10 on Windows 8 (IIS 8), the transport method is WebSockets.



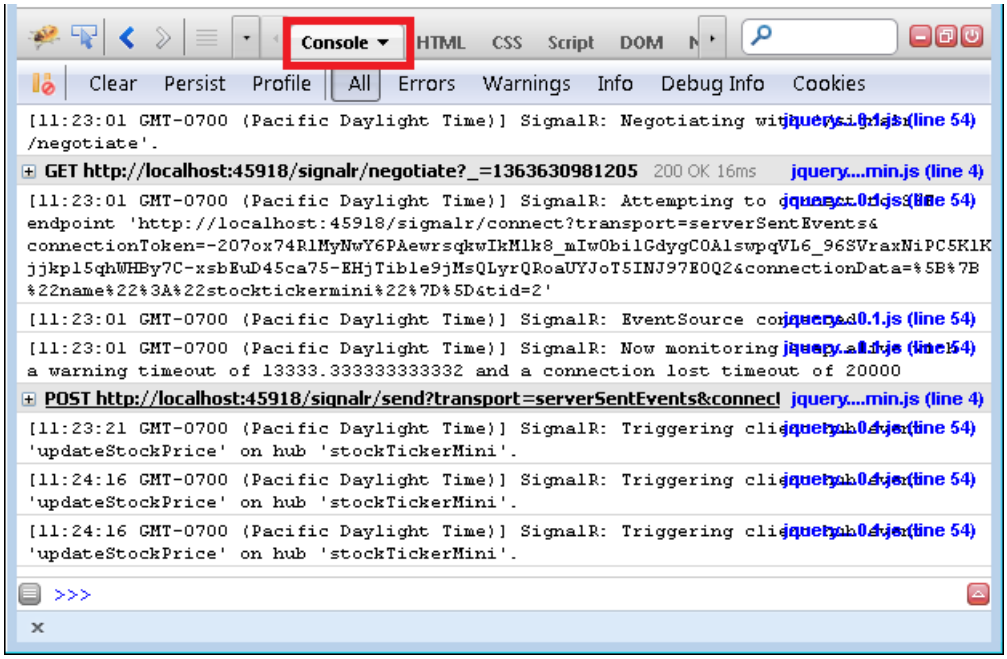
   If you are running Internet Explorer 10 on Windows 7 (IIS 7.5), the transport method is iframe.



In Firefox, install the Firebug add-in to get a Console window. If you are running Firefox 19 on Windows 8 (IIS 8), the transport method is WebSockets.



If you are running Firefox 19 on Windows 7 (IIS 7.5), the transport method is server-sent events.

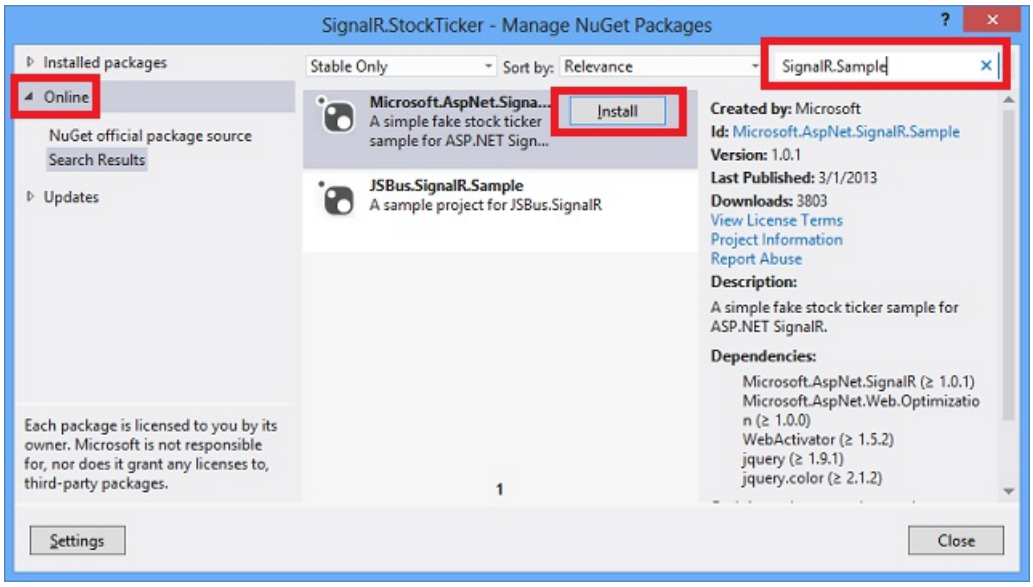


Install and review the full StockTicker sample

The StockTicker application that is installed by the Microsoft.AspNet.SignalR.Sample NuGet package includes more features than the simplified version that you just created from scratch. In this section of the tutorial, you install the NuGet package and review the new features and the code that implements them.

Install the SignalR.Sample NuGet package

1. In **Solution Explorer**, right-click the project and click **Manage NuGet Packages**.
2. In the **Manage NuGet Packages** dialog box, click **Online**, enter *SignalR.Sample* in the **Search Online** box, and then click **Install** in the **SignalR.Sample** package.



3. In the *Global.asax* file, comment out the RouteTable.Routes.MapHubs(); line that you added earlier in the Application_Start method.

   The code in *Global.asax* is no longer needed because the SignalR.Sample package registers the SignalR route in the *App_Start/RegisterHubs.cs* file:

```
[assembly: WebActivator.PreApplicationStartMethod(typeof(SignalR.StockTicker.RegisterHubs), "Start")]

namespace SignalR.StockTicker
{
    public static class RegisterHubs
    {
        public static void Start()
        {
            // Register the default hubs route: ~/signalr/hubs
            RouteTable.Routes.MapHubs();
        }
    }
}
```

   The WebActivator class that is referenced by the assembly attribute is included in the WebActivatorEx NuGet package, which is installed as a dependency of the SignalR.Sample package.

4. In **Solution Explorer**, expand the *SignalR.Sample* folder which was created by installing the SignalR.Sample package.

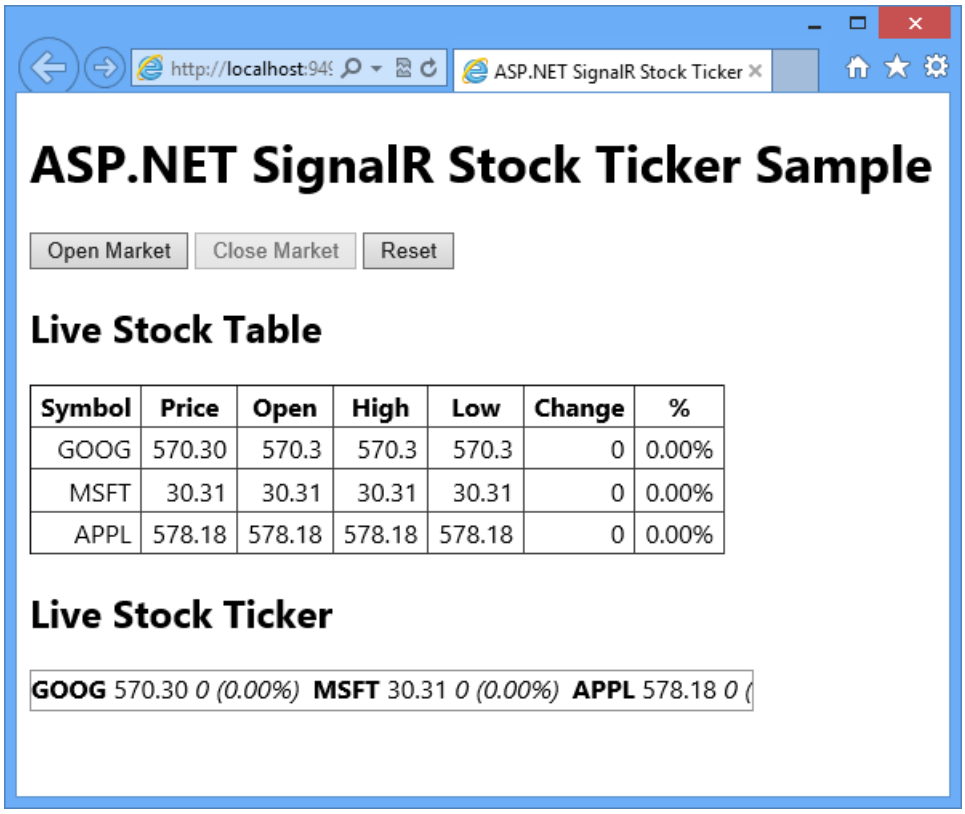
5. In the *SignalR.Sample* folder, right-click *StockTicker.html*, and then click **Set As Start Page**.

> **NOTE**
>
> Installing The SignalR.Sample NuGet package might change the version of jQuery that you have in your *Scripts* folder. The new *StockTicker.html* file that the package installs in the *SignalR.Sample* folder will be in sync with the jQuery version that the package installs, but if you want to run your original *StockTicker.html* file again, you might have to update the jQuery reference in the script tag first.

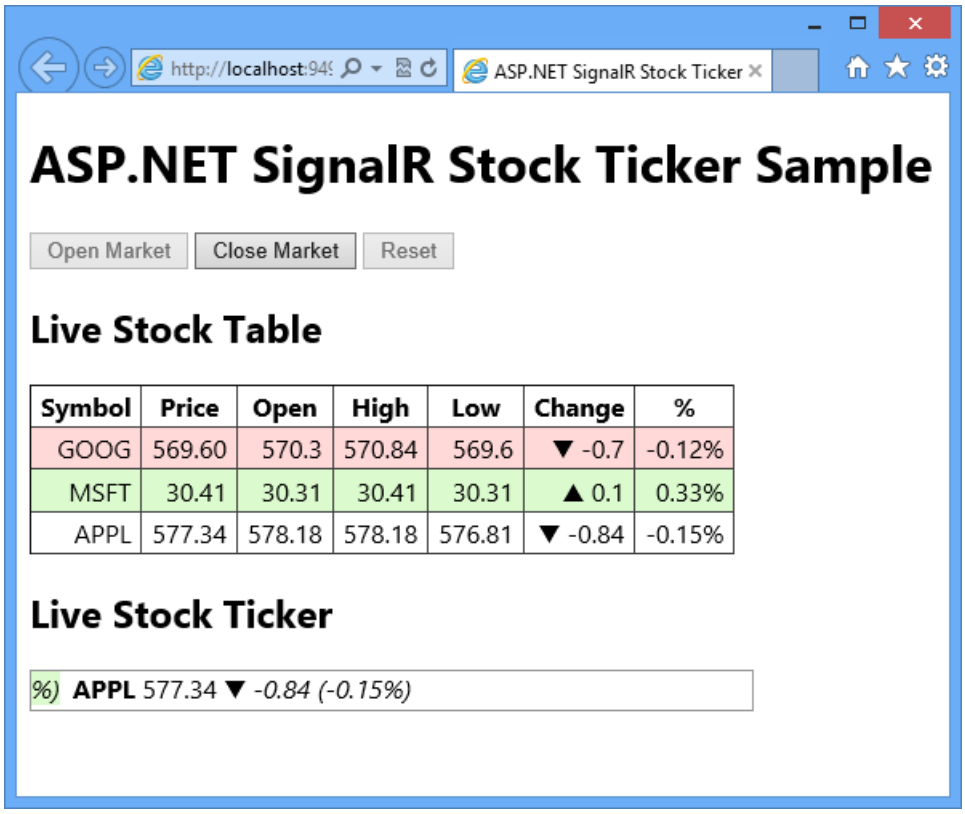
Run the application

1. Press F5 to run the application.

   In addition to the grid that you saw earlier, the full stock ticker application shows a horizontally scrolling window that displays the same stock data. When you run the application for the first time, the "market" is "closed" and you see a static grid and a ticker window that isn't scrolling.



When you click **Open Market**, the **Live Stock Ticker** box starts to scroll horizontally, and the server starts to periodically broadcast stock price changes on a random basis. Each time a stock price changes, both the **Live Stock Table** grid and the **Live Stock Ticker** box are updated. When a stock's price change is positive, the stock is shown with a green background, and when the change is negative, the stock is shown with a red background.



The **Close Market** button stops the changes and stops the ticker scrolling, and the **Reset** button resets all stock data to the initial state before price changes started. If you open more browser windows and go to the same URL, you see the same data dynamically updated at the same time in each browser. When you click one of the buttons, all browsers respond the same way at the same time.

Live Stock Ticker display

The **Live Stock Ticker** display is an unordered list in a div element that is formatted into a single line by CSS styles. The ticker is initialized and updated the same way as the table: by replacing placeholders in a <li> template string and dynamically adding the <li> elements to the <ul> element. The scrolling is performed by using the jQuery animate function to vary the margin-left of the unordered list within the div.

The stock ticker HTML:

```
<h2>Live Stock Ticker</h2>
<div id="stockTicker">
    <div class="inner">
        <ul>
            <li class="loading">loading...</li>
        </ul>
    </div>
</div>
```

The stock ticker CSS:

```
#stockTicker {
    overflow: hidden;
    width: 450px;
    height: 24px;
    border: 1px solid #999;
    }

    #stockTicker .inner {
        width: 9999px;
    }

    #stockTicker ul {
        display: inline-block;
        list-style-type: none;
        margin: 0;
        padding: 0;
    }

    #stockTicker li {
        display: inline-block;
        margin-right: 8px;
    }

    /*<li data-symbol="{Symbol}"><span class="symbol">{Symbol}</span><span class="price">{Price}</span><span
class="change">{PercentChange}</span></li>*/
    #stockTicker .symbol {
        font-weight: bold;
    }

    #stockTicker .change {
        font-style: italic;
    }
```

The jQuery code that makes it scroll:

```
function scrollTicker() {
    var w = $stockTickerUl.width();
    $stockTickerUl.css({ marginLeft: w });
    $stockTickerUl.animate({ marginLeft: -w }, 15000, 'linear', scrollTicker);
}
```

Additional methods on the server that the client can call

The StockTickerHub class defines four additional methods that the client can call:

```
public string GetMarketState()
{
    return _stockTicker.MarketState.ToString();
}

public void OpenMarket()
{
    _stockTicker.OpenMarket();
}

public void CloseMarket()
{
    _stockTicker.CloseMarket();
}

public void Reset()
{
    _stockTicker.Reset();
}
```

OpenMarket, CloseMarket, and Reset are called in response to the buttons at the top of the page. They demonstrate the pattern of one client triggering a change in state that is immediately propagated to all clients. Each of these methods calls a method in the StockTicker class that effects the market state change and then broadcasts the new state.

In the StockTicker class, the state of the market is maintained by a MarketState property that returns a MarketState enum value:

```
public MarketState MarketState
{
    get { return _marketState; }
    private set { _marketState = value; }
}

public enum MarketState
{
    Closed,
    Open
}
```

Each of the methods that change the market state do so inside a lock block because the StockTicker class has to be threadsafe:

```
public void OpenMarket()
{
    lock (_marketStateLock)
    {
        if (MarketState != MarketState.Open)
        {
            _timer = new Timer(UpdateStockPrices, null, _updateInterval, _updateInterval);
            MarketState = MarketState.Open;
            BroadcastMarketStateChange(MarketState.Open);
        }
    }
}

public void CloseMarket()
{
    lock (_marketStateLock)
    {
        if (MarketState == MarketState.Open)
        {
            if (_timer != null)
            {
                _timer.Dispose();
            }
            MarketState = MarketState.Closed;
            BroadcastMarketStateChange(MarketState.Closed);
        }
    }
}

public void Reset()
{
    lock (_marketStateLock)
    {
        if (MarketState != MarketState.Closed)
        {
            throw new InvalidOperationException("Market must be closed before it can be reset.");
        }
        LoadDefaultStocks();
        BroadcastMarketReset();
    }
}
```

To ensure that this code is threadsafe, the _marketState field that backs the MarketState property is marked as volatile,

```
private volatile MarketState _marketState;
```

The BroadcastMarketStateChange and BroadcastMarketReset methods are similar to the BroadcastStockPrice method that you already saw, except they call different methods defined at the client:

```
private void BroadcastMarketStateChange(MarketState marketState)
{
    switch (marketState)
    {
        case MarketState.Open:
            Clients.All.marketOpened();
            break;
        case MarketState.Closed:
            Clients.All.marketClosed();
            break;
        default:
            break;
    }
}

private void BroadcastMarketReset()
{
    Clients.All.marketReset();
}
```

Additional functions on the client that the server can call

The updateStockPrice function now handles both the grid and the ticker display, and it uses jQuery.Color to flash red and green colors.

New functions in *SignalR.StockTicker.js* enable and disable the buttons based on market state, and they stop or start the ticker window horizontal scrolling. Since multiple functions are being added to ticker.client, the jQuery extend function is used to add them.

```
$.extend(ticker.client, {
    updateStockPrice: function (stock) {
        var displayStock = formatStock(stock),
            $row = $(rowTemplate.supplant(displayStock)),
            $li = $(liTemplate.supplant(displayStock)),
            bg = stock.LastChange === 0
                ? '255,216,0' // yellow
                : stock.LastChange > 0
                    ? '154,240,117' // green
                    : '255,148,148'; // red

        $stockTableBody.find('tr[data-symbol=' + stock.Symbol + ']')
            .replaceWith($row);
        $stockTickerUl.find('li[data-symbol=' + stock.Symbol + ']')
            .replaceWith($li);

        $row.flash(bg, 1000);
        $li.flash(bg, 1000);
    },

    marketOpened: function () {
        $("#open").prop("disabled", true);
        $("#close").prop("disabled", false);
        $("#reset").prop("disabled", true);
        scrollTicker();
    },

    marketClosed: function () {
        $("#open").prop("disabled", false);
        $("#close").prop("disabled", true);
        $("#reset").prop("disabled", false);
        stopTicker();
    },

    marketReset: function () {
        return init();
    }
});
```

Additional client setup after establishing the connection

After the client establishes the connection, it has some additional work to do: find out if the market is open or closed in order to call the appropriate marketOpened or marketClosed function, and attach the server method calls to the buttons.

```
$.connection.hub.start()
    .pipe(init)
    .pipe(function () {
        return ticker.server.getMarketState();
    })
    .done(function (state) {
        if (state === 'Open') {
            ticker.client.marketOpened();
        } else {
            ticker.client.marketClosed();
        }

        // Wire up the buttons
        $("#open").click(function () {
            ticker.server.openMarket();
        });

        $("#close").click(function () {
            ticker.server.closeMarket();
        });

        $("#reset").click(function () {
            ticker.server.reset();
        });
    });
```

The server methods are not wired up to the buttons until after the connection is established, so that the code can't try to call the server methods before they are available.

Next steps

In this tutorial you've learned how to program a SignalR application that broadcasts messages from the server to all connected clients, both on a periodic basis and in response to notifications from any client. The pattern of using a multi-threaded singleton instance to maintain server state can also be also used in multi-player online game scenarios. For an example, see the ShootR game that is based on SignalR.

For tutorials that show peer-to-peer communication scenarios, see Getting Started with SignalR and Real-Time Updating with SignalR.

To learn more advanced SignalR development concepts, visit the following sites for SignalR source code and resources:

- ASP.NET SignalR
- SignalR Project
- SignalR Github and Samples
- SignalR Wiki

ASP.NET SignalR Hubs API Guide - Server (SignalR 1.x)

2/20/2020 • 29 minutes to read • Edit Online

by Patrick Fletcher, Tom Dykstra

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This document provides an introduction to programming the server side of the ASP.NET SignalR Hubs API for SignalR version 1.1, with code samples demonstrating common options.

The SignalR Hubs API enables you to make remote procedure calls (RPCs) from a server to connected clients and from clients to the server. In server code, you define methods that can be called by clients, and you call methods that run on the client. In client code, you define methods that can be called from the server, and you call methods that run on the server. SignalR takes care of all of the client-to-server plumbing for you.

SignalR also offers a lower-level API called Persistent Connections. For an introduction to SignalR, Hubs, and Persistent Connections, or for a tutorial that shows how to build a complete SignalR application, see SignalR - Getting Started.

Overview

This document contains the following sections:

- How to register the SignalR route and configure SignalR options
  - The /signalr URL
  - Configuring SignalR options
- How to create and use Hub classes
  - Hub object lifetime
  - Camel-casing of Hub names in JavaScript clients
  - Multiple Hubs
- How to define methods in the Hub class that clients can call
  - Camel-casing of method names in JavaScript clients
  - When to execute asynchronously
  - Defining overloads
- How to call client methods from the Hub class
  - Selecting which clients will receive the RPC
  - No compile-time validation for method names
  - Case-insensitive method name matching
  - Asynchronous execution
- How to manage group membership from the Hub class
  - Asynchronous execution of Add and Remove methods

- Group membership persistence
- Single-user groups
- How to handle connection lifetime events in the Hub class
  - When OnConnected, OnDisconnected, and OnReconnected are called
  - Caller state not populated
- How to get information about the client from the Context property
- How to pass state between clients and the Hub class
- How to handle errors in the Hub class
- How to call client methods and manage groups from outside the Hub class
  - Calling client methods
  - Managing group membership
- How to enable tracing
- How to customize the Hubs pipeline

For documentation on how to program clients, see the following resources:

- SignalR Hubs API Guide - JavaScript Client
- SignalR Hubs API Guide - .NET Client

Links to API Reference topics are to the .NET 4.5 version of the API. If you're using .NET 4, see the .NET 4 version of the API topics.

How to register the SignalR route and configure SignalR options

To define the route that clients will use to connect to your Hub, call the MapHubs method when the application starts. `MapHubs` is an extension method for the `System.Web.Routing.RouteCollection` class. The following example shows how to define the SignalR Hubs route in the *Global.asax* file.

```
using System.Web.Routing;
```

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        RouteTable.Routes.MapHubs();
    }
}
```

If you are adding SignalR functionality to an ASP.NET MVC application, make sure that the SignalR route is added before the other routes. For more information, see Tutorial: Getting Started with SignalR and MVC 4.

The /signalr URL

By default, the route URL which clients will use to connect to your Hub is "/signalr". (Don't confuse this URL with the "/signalr/hubs" URL, which is for the automatically generated JavaScript file. For more information about the generated proxy, see SignalR Hubs API Guide - JavaScript Client - The generated proxy and what it does for you.)

There might be extraordinary circumstances that make this base URL not usable for SignalR; for example, you have a folder in your project named *signalr* and you don't want to change the name. In that case, you can change the base URL, as shown in the following examples (replace "/signalr" in the sample code with your desired URL).

Server code that specifies the URL

```
RouteTable.Routes.MapHubs("/signalr", new HubConfiguration());
```

JavaScript client code that specifies the URL (with the generated proxy)

```
$.connection.hub.url = "/signalr"
$.connection.hub.start().done(init);
```

JavaScript client code that specifies the URL (without the generated proxy)

```
var connection = $.hubConnection("/signalr", { useDefaultPath: false });
```

.NET client code that specifies the URL

```
var hubConnection = new HubConnection("http://contoso.com/signalr", useDefaultUrl: false);
```

Configuring SignalR Options

Overloads of the `MapHubs` method enable you to specify a custom URL, a custom dependency resolver, and the following options:

- Enable cross-domain calls from browser clients.

  Typically if the browser loads a page from `http://contoso.com`, the SignalR connection is in the same domain, at `http://contoso.com/signalr`. If the page from `http://contoso.com` makes a connection to `http://fabrikam.com/signalr`, that is a cross-domain connection. For security reasons, cross-domain connections are disabled by default. For more information, see ASP.NET SignalR Hubs API Guide - JavaScript Client - How to establish a cross-domain connection.

- Enable detailed error messages.

  When errors occur, the default behavior of SignalR is to send to clients a notification message without details about what happened. Sending detailed error information to clients is not recommended in production, because malicious users might be able to use the information in attacks against your application. For troubleshooting, you can use this option to temporarily enable more informative error reporting.

- Disable automatically generated JavaScript proxy files.

  By default, a JavaScript file with proxies for your Hub classes is generated in response to the URL "/signalr/hubs". If you don't want to use the JavaScript proxies, or if you want to generate this file manually and refer to a physical file in your clients, you can use this option to disable proxy generation. For more information, see SignalR Hubs API Guide - JavaScript Client - How to create a physical file for the SignalR generated proxy.

The following example shows how to specify the SignalR connection URL and these options in a call to the `MapHubs` method. To specify a custom URL, replace "/signalr" in the example with the URL that you want to use.

```
var hubConfiguration = new HubConfiguration();
hubConfiguration.EnableCrossDomain = true;
hubConfiguration.EnableDetailedErrors = true;
hubConfiguration.EnableJavaScriptProxies = false;
RouteTable.Routes.MapHubs("/signalr", hubConfiguration);
```

How to create and use Hub classes

To create a Hub, create a class that derives from Microsoft.Aspnet.Signalr.Hub. The following example shows a simple Hub class for a chat application.

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addNewMessageToPage(name, message);
    }
}
```

In this example, a connected client can call the `NewContosoChatMessage` method, and when it does, the data received is broadcasted to all connected clients.

Hub object lifetime

You don't instantiate the Hub class or call its methods from your own code on the server; all that is done for you by the SignalR Hubs pipeline. SignalR creates a new instance of your Hub class each time it needs to handle a Hub operation such as when a client connects, disconnects, or makes a method call to the server.

Because instances of the Hub class are transient, you can't use them to maintain state from one method call to the next. Each time the server receives a method call from a client, a new instance of your Hub class processes the message. To maintain state through multiple connections and method calls, use some other method such as a database, or a static variable on the Hub class, or a different class that does not derive from `Hub`. If you persist data in memory, using a method such as a static variable on the Hub class, the data will be lost when the app domain recycles.

If you want to send messages to clients from your own code that runs outside the Hub class, you can't do it by instantiating a Hub class instance, but you can do it by getting a reference to the SignalR context object for your Hub class. For more information, see How to call client methods and manage groups from outside the Hub class later in this topic.

Camel-casing of Hub names in JavaScript clients

By default, JavaScript clients refer to Hubs by using a camel-cased version of the class name. SignalR automatically makes this change so that JavaScript code can conform to JavaScript conventions. The previous example would be referred to as `contosoChatHub` in JavaScript code.

Server

```
public class ContosoChatHub : Hub
```

JavaScript client using generated proxy

```
var contosoChatHubProxy = $.connection.contosoChatHub;
```

If you want to specify a different name for clients to use, add the `HubName` attribute. When you use a `HubName` attribute, there is no name change to camel case on JavaScript clients.

Server

```
[HubName("PascalCaseContosoChatHub")]
public class ContosoChatHub : Hub
```

JavaScript client using generated proxy

```
var contosoChatHubProxy = $.connection.PascalCaseContosoChatHub;
```

Multiple Hubs

You can define multiple Hub classes in an application. When you do that, the connection is shared but groups are separate:

- All clients will use the same URL to establish a SignalR connection with your service ("/signalr" or your custom URL if you specified one), and that connection is used for all Hubs defined by the service.

  There is no performance difference for multiple Hubs compared to defining all Hub functionality in a single class.

- All Hubs get the same HTTP request information.

  Since all Hubs share the same connection, the only HTTP request information that the server gets is what comes in the original HTTP request that establishes the SignalR connection. If you use the connection request to pass information from the client to the server by specifying a query string, you can't provide different query strings to different Hubs. All Hubs will receive the same information.

- The generated JavaScript proxies file will contain proxies for all Hubs in one file.

  For information about JavaScript proxies, see SignalR Hubs API Guide - JavaScript Client - The generated proxy and what it does for you.

- Groups are defined within Hubs.

  In SignalR you can define named groups to broadcast to subsets of connected clients. Groups are maintained separately for each Hub. For example, a group named "Administrators" would include one set of clients for your `ContosoChatHub` class, and the same group name would refer to a different set of clients for your `StockTickerHub` class.

How to define methods in the Hub class that clients can call

To expose a method on the Hub that you want to be callable from the client, declare a public method, as shown in the following examples.

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addNewMessageToPage(name, message);
    }
}
```

```
public class StockTickerHub : Hub
{
    public IEnumerable<Stock> GetAllStocks()
    {
        return _stockTicker.GetAllStocks();
    }
}
```

You can specify a return type and parameters, including complex types and arrays, as you would in any C# method. Any data that you receive in parameters or return to the caller is communicated between the client and the server by using JSON, and SignalR handles the binding of complex objects and arrays of objects automatically.

Camel-casing of method names in JavaScript clients

By default, JavaScript clients refer to Hub methods by using a camel-cased version of the method name. SignalR automatically makes this change so that JavaScript code can conform to JavaScript conventions.

Server

```
public void NewContosoChatMessage(string userName, string message)
```

JavaScript client using generated proxy

```
contosoChatHubProxy.server.newContosoChatMessage($(userName, message);
```

If you want to specify a different name for clients to use, add the `HubMethodName` attribute.

Server

```
[HubMethodName("PascalCaseNewContosoChatMessage")]
public void NewContosoChatMessage(string userName, string message)
```

JavaScript client using generated proxy

```
contosoChatHubProxy.server.PascalCaseNewContosoChatMessage(userName, message);
```

When to execute asynchronously

If the method will be long-running or has to do work that would involve waiting, such as a database lookup or a web service call, make the Hub method asynchronous by returning a Task (in place of `void` return) or Task<T> object (in place of `T` return type). When you return a `Task` object from the method, SignalR waits for the `Task` to complete, and then it sends the unwrapped result back to the client, so there is no difference in how you code the method call in the client.

Making a Hub method asynchronous avoids blocking the connection when it uses the WebSocket transport. When a Hub method executes synchronously and the transport is WebSocket, subsequent invocations of methods on the Hub from the same client are blocked until the Hub method completes.

The following example shows the same method coded to run synchronously or asynchronously, followed by JavaScript client code that works for calling either version.

Synchronous

```
public IEnumerable<Stock> GetAllStocks()
{
    // Returns data from memory.
    return _stockTicker.GetAllStocks();
}
```

Asynchronous - ASP.NET 4.5

```
public async Task<IEnumerable<Stock>> GetAllStocks()
{
    // Returns data from a web service.
    var uri = Util.getServiceUri("Stocks");
    using (HttpClient httpClient = new HttpClient())
    {
        var response = await httpClient.GetAsync(uri);
        return (await response.Content.ReadAsAsync<IEnumerable<Stock>>());
    }
}
```

JavaScript client using generated proxy

```
stockTickerHubProxy.server.getAllStocks().done(function (stocks) {
    $.each(stocks, function () {
        alert(this.Symbol + ' ' + this.Price);
    });
});
```

For more information about how to use asynchronous methods in ASP.NET 4.5, see Using Asynchronous Methods in ASP.NET MVC 4.

Defining Overloads

If you want to define overloads for a method, the number of parameters in each overload must be different. If you differentiate an overload just by specifying different parameter types, your Hub class will compile but the SignalR service will throw an exception at run time when clients try to call one of the overloads.

How to call client methods from the Hub class

To call client methods from the server, use the `Clients` property in a method in your Hub class. The following example shows server code that calls `addNewMessageToPage` on all connected clients, and client code that defines the method in a JavaScript client.

Server

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addNewMessageToPage(name, message);
    }
}
```

JavaScript client using generated proxy

```
contosoChatHubProxy.client.addNewMessageToPage = function (name, message) {
    // Add the message to the page.
    $('#discussion').append('<li><strong>' + htmlEncode(name)
        + '</strong>: ' + htmlEncode(message) + '<li>');
};
```

You can't get a return value from a client method; syntax such as `int x = Clients.All.add(1,1)` does not work.

You can specify complex types and arrays for the parameters. The following example passes a complex type to the client in a method parameter.

Server code that calls a client method using a complex object

```
public void SendMessage(string name, string message)
{
    Clients.All.addContosoChatMessageToPage(new ContosoChatMessage() { UserName = name, Message = message });
}
```

Server code that defines the complex object

```
public class ContosoChatMessage
{
    public string UserName { get; set; }
    public string Message { get; set; }
}
```

JavaScript client using generated proxy

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addMessageToPage = function (message) {
    console.log(message.UserName + ' ' + message.Message);
});
```

Selecting which clients will receive the RPC

The Clients property returns a HubConnectionContext object that provides several options for specifying which clients will receive the RPC:

- All connected clients.

  ```
  Clients.All.addContosoChatMessageToPage(name, message);
  ```

- Only the calling client.

  ```
  Clients.Caller.addContosoChatMessageToPage(name, message);
  ```

- All clients except the calling client.

  ```
  Clients.Others.addContosoChatMessageToPage(name, message);
  ```

- A specific client identified by connection ID.

  ```
  Clients.Client(Context.ConnectionId).addContosoChatMessageToPage(name, message);
  ```

  This example calls `addContosoChatMessageToPage` on the calling client and has the same effect as using `Clients.Caller`.

- All connected clients except the specified clients, identified by connection ID.

  ```
  Clients.AllExcept(connectionId1, connectionId2).addContosoChatMessageToPage(name, message);
  ```

- All connected clients in a specified group.

  ```
  Clients.Group(groupName).addContosoChatMessageToPage(name, message);
  ```

- All connected clients in a specified group except the specified clients, identified by connection ID.

```
Clients.Group(groupName, connectionId1, connectionId2).addContosoChatMessageToPage(name, message);
```

- All connected clients in a specified group except the calling client.

```
Clients.OthersInGroup(groupName).addContosoChatMessageToPage(name, message);
```

No compile-time validation for method names

The method name that you specify is interpreted as a dynamic object, which means there is no IntelliSense or compile-time validation for it. The expression is evaluated at run time. When the method call executes, SignalR sends the method name and the parameter values to the client, and if the client has a method that matches the name, that method is called and the parameter values are passed to it. If no matching method is found on the client, no error is raised. For information about the format of the data that SignalR transmits to the client behind the scenes when you call a client method, see Introduction to SignalR.

Case-insensitive method name matching

Method name matching is case-insensitive. For example, `Clients.All.addContosoChatMessageToPage` on the server will execute `AddContosoChatMessageToPage`, `addcontosochatmessagetopage`, or `addContosoChatMessageToPage` on the client.

Asynchronous execution

The method that you call executes asynchronously. Any code that comes after a method call to a client will execute immediately without waiting for SignalR to finish transmitting data to clients unless you specify that the subsequent lines of code should wait for method completion. The following code examples show how to execute two client methods sequentially, one using code that works in .NET 4.5, and one using code that works in .NET 4.

.NET 4.5 example

```
async public Task NewContosoChatMessage(string name, string message)
{
    await Clients.Others.addContosoChatMessageToPage(data);
    Clients.Caller.notifyMessageSent();
}
```

.NET 4 example

```
public void NewContosoChatMessage(string name, string message)
{
    (Clients.Others.addContosoChatMessageToPage(data) as Task).ContinueWith(antecedent =>
        Clients.Caller.notifyMessageSent());
}
```

If you use `await` or `ContinueWith` to wait until a client method finishes before the next line of code executes, that does not mean that clients will actually receive the message before the next line of code executes. "Completion" of a client method call only means that SignalR has done everything necessary to send the message. If you need verification that clients received the message, you have to program that mechanism yourself. For example, you could code a `MessageReceived` method on the Hub, and in the `addContosoChatMessageToPage` method on the client you could call `MessageReceived` after you do whatever work you need to do on the client. In `MessageReceived` in the Hub you can do whatever work depends on actual client reception and processing of the original method call.

How to use a string variable as the method name

If you want to invoke a client method by using a string variable as the method name, cast `Clients.All` (or

`Clients.Others`, `Clients.Caller`, etc.) to `IClientProxy` and then call Invoke(methodName, args...).

```
public void NewContosoChatMessage(string name, string message)
{
    string methodToCall = "addContosoChatMessageToPage";
    IClientProxy proxy = Clients.All;
    proxy.Invoke(methodToCall, name, message);
}
```

How to manage group membership from the Hub class

Groups in SignalR provide a method for broadcasting messages to specified subsets of connected clients. A group can have any number of clients, and a client can be a member of any number of groups.

To manage group membership, use the Add and Remove methods provided by the `Groups` property of the Hub class. The following example shows the `Groups.Add` and `Groups.Remove` methods used in Hub methods that are called by client code, followed by JavaScript client code that calls them.

Server

```
public class ContosoChatHub : Hub
{
    public Task JoinGroup(string groupName)
    {
        return Groups.Add(Context.ConnectionId, groupName);
    }

    public Task LeaveGroup(string groupName)
    {
        return Groups.Remove(Context.ConnectionId, groupName);
    }
}
```

JavaScript client using generated proxy

```
contosoChatHubProxy.server.joinGroup(groupName);
```

```
contosoChatHubProxy.server.leaveGroup(groupName);
```

You don't have to explicitly create groups. In effect a group is automatically created the first time you specify its name in a call to `Groups.Add`, and it is deleted when you remove the last connection from membership in it.

There is no API for getting a group membership list or a list of groups. SignalR sends messages to clients and groups based on a pub/sub model, and the server does not maintain lists of groups or group memberships. This helps maximize scalability, because whenever you add a node to a web farm, any state that SignalR maintains has to be propagated to the new node.

Asynchronous execution of Add and Remove methods

The `Groups.Add` and `Groups.Remove` methods execute asynchronously. If you want to add a client to a group and immediately send a message to the client by using the group, you have to make sure that the `Groups.Add` method finishes first. The following code examples show how to do that, one by using code that works in .NET 4.5, and one by using code that works in .NET 4

.NET 4.5 example

```
async public Task JoinGroup(string groupName)
{
    await Groups.Add(Context.ConnectionId, groupName);
    Clients.Group(groupname).addContosoChatMessageToPage(Context.ConnectionId + " added to group");
}
```

.NET 4 example

```
public void JoinGroup(string groupName)
{
    (Groups.Add(Context.ConnectionId, groupName) as Task).ContinueWith(antecedent =>
        Clients.Group(groupName).addContosoChatMessageToPage(Context.ConnectionId + " added to group"));
}
```

Group membership persistence

SignalR tracks connections, not users, so if you want a user to be in the same group every time the user establishes a connection, you have to call `Groups.Add` every time the user establishes a new connection.

After a temporary loss of connectivity, sometimes SignalR can restore the connection automatically. In that case, SignalR is restoring the same connection, not establishing a new connection, and so the client's group membership is automatically restored. This is possible even when the temporary break is the result of a server reboot or failure, because connection state for each client, including group memberships, is round-tripped to the client. If a server goes down and is replaced by a new server before the connection times out, a client can reconnect automatically to the new server and re-enroll in groups it is a member of.

When a connection can't be restored automatically after a loss of connectivity, or when the connection times out, or when the client disconnects (for example, when a browser navigates to a new page), group memberships are lost. The next time the user connects will be a new connection. To maintain group memberships when the same user establishes a new connection, your application has to track the associations between users and groups, and restore group memberships each time a user establishes a new connection.

For more information about connections and reconnections, see How to handle connection lifetime events in the Hub class later in this topic.

Single-user groups

Applications that use SignalR typically have to keep track of the associations between users and connections in order to know which user has sent a message and which user(s) should be receiving a message. Groups are used in one of the two commonly used patterns for doing that.

- Single-user groups.

  You can specify the user name as the group name, and add the current connection ID to the group every time the user connects or reconnects. To send messages to the user you send to the group. A disadvantage of this method is that the group doesn't provide you with a way to find out if the user is online or offline.

- Track associations between user names and connection IDs.

  You can store an association between each user name and one or more connection IDs in a dictionary or database, and update the stored data each time the user connects or disconnects. To send messages to the user you specify the connection IDs. A disadvantage of this method is that it takes more memory.

How to handle connection lifetime events in the Hub class

Typical reasons for handling connection lifetime events are to keep track of whether a user is connected or not, and to keep track of the association between user names and connection IDs. To run your own code when clients connect or disconnect, override the `OnConnected`, `OnDisconnected`, and `OnReconnected` virtual methods of the Hub

class, as shown in the following example.

```
public class ContosoChatHub : Hub
{
    public override Task OnConnected()
    {
        // Add your own code here.
        // For example: in a chat application, record the association between
        // the current connection ID and user name, and mark the user as online.
        // After the code in this method completes, the client is informed that
        // the connection is established; for example, in a JavaScript client,
        // the start().done callback is executed.
        return base.OnConnected();
    }

    public override Task OnDisconnected()
    {
        // Add your own code here.
        // For example: in a chat application, mark the user as offline,
        // delete the association between the current connection id and user name.
        return base.OnDisconnected();
    }

    public override Task OnReconnected()
    {
        // Add your own code here.
        // For example: in a chat application, you might have marked the
        // user as offline after a period of inactivity; in that case
        // mark the user as online again.
        return base.OnReconnected();
    }
}
```

When OnConnected, OnDisconnected, and OnReconnected are called

Each time a browser navigates to a new page, a new connection has to be established, which means SignalR will execute the `OnDisconnected` method followed by the `OnConnected` method. SignalR always creates a new connection ID when a new connection is established.

The `OnReconnected` method is called when there has been a temporary break in connectivity that SignalR can automatically recover from, such as when a cable is temporarily disconnected and reconnected before the connection times out. The `OnDisconnected` method is called when the client is disconnected and SignalR can't automatically reconnect, such as when a browser navigates to a new page. Therefore, a possible sequence of events for a given client is `OnConnected`, `OnReconnected`, `OnDisconnected`; or `OnConnected`, `OnDisconnected`. You won't see the sequence `OnConnected`, `OnDisconnected`, `OnReconnected` for a given connection.

The `OnDisconnected` method doesn't get called in some scenarios, such as when a server goes down or the App Domain gets recycled. When another server comes on line or the App Domain completes its recycle, some clients may be able to reconnect and fire the `OnReconnected` event.

For more information, see Understanding and Handling Connection Lifetime Events in SignalR.

Caller state not populated

The connection lifetime event handler methods are called from the server, which means that any state that you put in the `state` object on the client will not be populated in the `Caller` property on the server. For information about the `state` object and the `Caller` property, see How to pass state between clients and the Hub class later in this topic.

How to get information about the client from the Context property

To get information about the client, use the `Context` property of the Hub class. The `Context` property returns a

HubCallerContext object which provides access to the following information:

- The connection ID of the calling client.

  ```
  string connectionID = Context.ConnectionId;
  ```

  The connection ID is a GUID that is assigned by SignalR (you can't specify the value in your own code). There is one connection ID for each connection, and the same connection ID is used by all Hubs if you have multiple Hubs in your application.

- HTTP header data.

  ```
  System.Collections.Specialized.NameValueCollection headers = Context.Request.Headers;
  ```

  You can also get HTTP headers from `Context.Headers`. The reason for multiple references to the same thing is that `Context.Headers` was created first, the `Context.Request` property was added later, and `Context.Headers` was retained for backward compatibility.

- Query string data.

  ```
  System.Collections.Specialized.NameValueCollection queryString = Context.Request.QueryString;
  string parameterValue = queryString["parametername"]
  ```

  You can also get query string data from `Context.QueryString`.

  The query string that you get in this property is the one that was used with the HTTP request that established the SignalR connection. You can add query string parameters in the client by configuring the connection, which is a convenient way to pass data about the client from the client to the server. The following example shows one way to add a query string in a JavaScript client when you use the generated proxy.

  ```
  $.connection.hub.qs = { "version" : "1.0" };
  ```

  For more information about setting query string parameters, see the API guides for the JavaScript and .NET clients.

  You can find the transport method used for the connection in the query string data, along with some other values used internally by SignalR:

  ```
  string transportMethod = queryString["transport"];
  ```

  The value of `transportMethod` will be "webSockets", "serverSentEvents", "foreverFrame" or "longPolling". Note that if you check this value in the `OnConnected` event handler method, in some scenarios you might initially get a transport value that is not the final negotiated transport method for the connection. In that case the method will throw an exception and will be called again later when the final transport method is established.

- Cookies.

  ```
  System.Collections.Generic.IDictionary<string, Cookie> cookies = Context.Request.Cookies;
  ```

  You can also get cookies from `Context.RequestCookies`.

- User information.

```
System.Security.Principal.IPrincipal user = Context.User;
```

- The HttpContext object for the request :

```
System.Web.HttpContextBase httpContext = Context.Request.GetHttpContext();
```

  Use this method instead of getting `HttpContext.Current` to get the `HttpContext` object for the SignalR connection.

How to pass state between clients and the Hub class

The client proxy provides a `state` object in which you can store data that you want to be transmitted to the server with each method call. On the server you can access this data in the `Clients.Caller` property in Hub methods that are called by clients. The `Clients.Caller` property is not populated for the connection lifetime event handler methods `OnConnected`, `OnDisconnected`, and `OnReconnected`.

Creating or updating data in the `state` object and the `Clients.Caller` property works in both directions. You can update values in the server and they are passed back to the client.

The following example shows JavaScript client code that stores state for transmission to the server with every method call.

```
contosoChatHubProxy.state.userName = "Fadi Fakhouri";
contosoChatHubProxy.state.computerName = "fadivm1";
```

The following example shows the equivalent code in a .NET client.

```
contosoChatHubProxy["userName"] = "Fadi Fakhouri";
chatHubProxy["computerName"] = "fadivm1";
```

In your Hub class, you can access this data in the `Clients.Caller` property. The following example shows code that retrieves the state referred to in the previous example.

```
public void NewContosoChatMessage(string data)
{
    string userName = Clients.Caller.userName;
    string computerName = Clients.Caller.computerName;
    Clients.Others.addContosoChatMessageToPage(message, userName, computerName);
}
```

> **NOTE**
> This mechanism for persisting state is not intended for large amounts of data, since everything you put in the `state` or `Clients.Caller` property is round-tripped with every method invocation. It's useful for smaller items such as user names or counters.

How to handle errors in the Hub class

To handle errors that occur in your Hub class methods, use one or both of the following methods:

- Wrap your method code in try-catch blocks and log the exception object. For debugging purposes you can send the exception to the client, but for security reasons sending detailed information to clients in production is not recommended.
- Create a Hubs pipeline module that handles the OnIncomingError method. The following example shows a pipeline module that logs errors, followed by code in Global.asax that injects the module into the Hubs pipeline.

```
public class ErrorHandlingPipelineModule : HubPipelineModule
{
    protected override void OnIncomingError(Exception ex, IHubIncomingInvokerContext context)
    {
        Debug.WriteLine("=> Exception " + ex.Message);
        if (ex.InnerException != null)
        {
            Debug.WriteLine("=> Inner Exception " + ex.InnerException.Message);
        }
        base.OnIncomingError(ex, context);
    }
}
```

```
protected void Application_Start(object sender, EventArgs e)
{
    GlobalHost.HubPipeline.AddModule(new ErrorHandlingPipelineModule());
    RouteTable.Routes.MapHubs();
}
```

For more information about Hub pipeline modules, see How to customize the Hubs pipeline later in this topic.

How to enable tracing

To enable server-side tracing, add a system.diagnostics element to your Web.config file, as shown in this example:

```
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit https://go.microsoft.com/fwlink/?
LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework, Version=5.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false"
/>
  </configSections>
  <connectionStrings>
    <add name="SignalRSamples" connectionString="Data Source=(local);Initial Catalog=SignalRSamples;Integrated
Security=SSPI;Asynchronous Processing=True;" />
    <add name="SignalRSamplesWithMARS" connectionString="Data Source=(local);Initial
Catalog=SignalRSamples;Integrated Security=SSPI;Asynchronous Processing=True;MultipleActiveResultSets=true;"
/>
  </connectionStrings>
  <system.web>
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>
  <system.webServer>
    <modules runAllManagedModulesForAllRequests="true" />
  </system.webServer>
  <system.diagnostics>
    <sources>
      <source name="SignalR.SqlMessageBus">
        <listeners>
          <add name="SignalR-Bus" />
        </listeners>
      </source>
```

```
      <source name="SignalR.ServiceBusMessageBus">
        <listeners>
            <add name="SignalR-Bus" />
        </listeners>
      </source>
      <source name="SignalR.ScaleoutMessageBus">
        <listeners>
            <add name="SignalR-Bus" />
        </listeners>
      </source>
      <source name="SignalR.Transports.WebSocketTransport">
        <listeners>
          <add name="SignalR-Transports" />
        </listeners>
      </source>
      <source name="SignalR.Transports.ServerSentEventsTransport">
          <listeners>
              <add name="SignalR-Transports" />
          </listeners>
      </source>
      <source name="SignalR.Transports.ForeverFrameTransport">
          <listeners>
              <add name="SignalR-Transports" />
          </listeners>
      </source>
      <source name="SignalR.Transports.LongPollingTransport">
        <listeners>
            <add name="SignalR-Transports" />
        </listeners>
      </source>
      <source name="SignalR.Transports.TransportHeartBeat">
        <listeners>
            <add name="SignalR-Transports" />
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="SignalRSwitch" value="Verbose" />
    </switches>
    <sharedListeners>
      <add name="SignalR-Transports"
           type="System.Diagnostics.TextWriterTraceListener"
           initializeData="transports.log.txt" />
        <add name="SignalR-Bus"
           type="System.Diagnostics.TextWriterTraceListener"
           initializeData="bus.log.txt" />
    </sharedListeners>
    <trace autoflush="true" />
  </system.diagnostics>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
EntityFramework">
      <parameters>
        <parameter value="v11.0" />
      </parameters>
    </defaultConnectionFactory>
  </entityFramework>
</configuration>
```

When you run the application in Visual Studio, you can view the logs in the **Output** window.

How to call client methods and manage groups from outside the Hub class

To call client methods from a different class than your Hub class, get a reference to the SignalR context object for

the Hub and use that to call methods on the client or manage groups.

The following sample `StockTicker` class gets the context object, stores it in an instance of the class, stores the class instance in a static property, and uses the context from the singleton class instance to call the `updateStockPrice` method on clients that are connected to a Hub named `StockTickerHub`.

```
// For the complete example, go to
// http://www.asp.net/signalr/overview/signalr-1x/getting-started/tutorial-server-broadcast-with-aspnet-
signalr
// This sample only shows code related to getting and using the SignalR context.
public class StockTicker
{
    // Singleton instance
    private readonly static Lazy<StockTicker> _instance = new Lazy<StockTicker>(
        () => new StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>()));

    private IHubContext _context;

    private StockTicker(IHubContext context)
    {
        _context = context;
    }

    // This method is invoked by a Timer object.
    private void UpdateStockPrices(object state)
    {
        foreach (var stock in _stocks.Values)
        {
            if (TryUpdateStockPrice(stock))
            {
                _context.Clients.All.updateStockPrice(stock);
            }
        }
    }
```

If you need to use the context multiple-times in a long-lived object, get the reference once and save it rather than getting it again each time. Getting the context once ensures that SignalR sends messages to clients in the same sequence in which your Hub methods make client method invocations. For a tutorial that shows how to use the SignalR context for a Hub, see Server Broadcast with ASP.NET SignalR.

Calling client methods

You can specify which clients will receive the RPC, but you have fewer options than when you call from a Hub class. The reason for this is that the context is not associated with a particular call from a client, so any methods that require knowledge of the current connection ID, such as `Clients.Others`, or `Clients.Caller`, or `Clients.OthersInGroup`, are not available. The following options are available:

- All connected clients.

  ```
  context.Clients.All.addContosoChatMessageToPage(name, message);
  ```

- A specific client identified by connection ID.

  ```
  context.Clients.Client(connectionID).addContosoChatMessageToPage(name, message);
  ```

- All connected clients except the specified clients, identified by connection ID.

  ```
  context.Clients.AllExcept(connectionId1, connectionId2).addContosoChatMessageToPage(name, message);
  ```

- All connected clients in a specified group.

```
context.Clients.Group(groupName).addContosoChatMessageToPage(name, message);
```

- All connected clients in a specified group except specified clients, identified by connection ID.

```
Clients.Group(groupName, connectionId1, connectionId2).addContosoChatMessageToPage(name, message);
```

If you are calling into your non-Hub class from methods in your Hub class, you can pass in the current connection ID and use that with `Clients.Client`, `Clients.AllExcept`, or `Clients.Group` to simulate `Clients.Caller`, `Clients.Others`, or `Clients.OthersInGroup`. In the following example, the `MoveShapeHub` class passes the connection ID to the `Broadcaster` class so that the `Broadcaster` class can simulate `Clients.Others`.

```
// For the complete example, see
// http://www.asp.net/signalr/overview/getting-started/tutorial-high-frequency-realtime-with-signalr
// This sample only shows code that passes connection ID to the non-Hub class,
// in order to simulate Clients.Others.
public class MoveShapeHub : Hub
{
    // Code not shown puts a singleton instance of Broadcaster in this variable.
    private Broadcaster _broadcaster;

    public void UpdateModel(ShapeModel clientModel)
    {
        clientModel.LastUpdatedBy = Context.ConnectionId;
        // Update the shape model within our broadcaster
        _broadcaster.UpdateShape(clientModel);
    }
}

public class Broadcaster
{
    public Broadcaster()
    {
        _hubContext = GlobalHost.ConnectionManager.GetHubContext<MoveShapeHub>();
    }

    public void UpdateShape(ShapeModel clientModel)
    {
        _model = clientModel;
        _modelUpdated = true;
    }

    // Called by a Timer object.
    public void BroadcastShape(object state)
    {
        if (_modelUpdated)
        {
            _hubContext.Clients.AllExcept(_model.LastUpdatedBy).updateShape(_model);
            _modelUpdated = false;
        }
    }
}
```

Managing group membership

For managing groups you have the same options as you do in a Hub class.

- Add a client to a group

```
context.Groups.Add(connectionID, groupName);
```

- Remove a client from a group

```
context.Groups.Remove(connectionID, groupName);
```

How to customize the Hubs pipeline

SignalR enables you to inject your own code into the Hub pipeline. The following example shows a custom Hub pipeline module that logs each incoming method call received from the client and outgoing method call invoked on the client:

```
public class LoggingPipelineModule : HubPipelineModule
{
    protected override bool OnBeforeIncoming(IHubIncomingInvokerContext context)
    {
        Debug.WriteLine("=> Invoking " + context.MethodDescriptor.Name + " on hub " +
context.MethodDescriptor.Hub.Name);
        return base.OnBeforeIncoming(context);
    }
    protected override bool OnBeforeOutgoing(IHubOutgoingInvokerContext context)
    {
        Debug.WriteLine("<= Invoking " + context.Invocation.Method + " on client hub " +
context.Invocation.Hub);
        return base.OnBeforeOutgoing(context);
    }
}
```

The following code in the *Global.asax* file registers the module to run in the Hub pipeline:

```
protected void Application_Start(object sender, EventArgs e)
{
    GlobalHost.HubPipeline.AddModule(new LoggingPipelineModule());
    RouteTable.Routes.MapHubs();
}
```

There are many different methods that you can override. For a complete list, see HubPipelineModule Methods.

SignalR 1.x Hubs API Guide - JavaScript Client

2/20/2020 • 21 minutes to read • Edit Online

by Patrick Fletcher, Tom Dykstra

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This document provides an introduction to using the Hubs API for SignalR version 1.1 in JavaScript clients, such as browsers and Windows Store (WinJS) applications.

The SignalR Hubs API enables you to make remote procedure calls (RPCs) from a server to connected clients and from clients to the server. In server code, you define methods that can be called by clients, and you call methods that run on the client. In client code, you define methods that can be called from the server, and you call methods that run on the server. SignalR takes care of all of the client-to-server plumbing for you.

SignalR also offers a lower-level API called Persistent Connections. For an introduction to SignalR, Hubs, and Persistent Connections, or for a tutorial that shows how to build a complete SignalR application, see SignalR - Getting Started.

Overview

This document contains the following sections:

- The generated proxy and what it does for you
  - When to use the generated proxy
- Client Setup
  - How to reference the dynamically generated proxy
  - How to create a physical file for the SignalR generated proxy
- How to establish a connection
  - $.connection.hub is the same object that $.hubConnection() creates
  - Asynchronous execution of the start method
- How to establish a cross-domain connection
- How to configure the connection
  - How to specify query string parameters
  - How to specify the transport method
- How to get a proxy for a Hub class
- How to define methods on the client that the server can call
- How to call server methods from the client
- How to handle connection lifetime events
- How to handle errors
- How to enable client-side logging

For documentation on how to program the server or .NET clients, see the following resources:

- SignalR Hubs API Guide - Server
- SignalR Hubs API Guide - .NET Client

Links to API Reference topics are to the .NET 4.5 version of the API. If you're using .NET 4, see the .NET 4 version of the API topics.

The generated proxy and what it does for you

You can program a JavaScript client to communicate with a SignalR service with or without a proxy that SignalR generates for you. What the proxy does for you is simplify the syntax of the code you use to connect, write methods that the server calls, and call methods on the server.

When you write code to call server methods, the generated proxy enables you to use syntax that looks as though you were executing a local function: you can write `serverMethod(arg1, arg2)` instead of `invoke('serverMethod', arg1, arg2)`. The generated proxy syntax also enables an immediate and intelligible client-side error if you mistype a server method name. And if you manually create the file that defines the proxies, you can also get IntelliSense support for writing code that calls server methods.

For example, suppose you have the following Hub class on the server:

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addContosoChatMessageToPage(name, message);
    }
}
```

The following code examples show what JavaScript code looks like for invoking the `NewContosoChatMessage` method on the server and receiving invocations of the `addContosoChatMessageToPage` method from the server.

With the generated proxy

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addContosoChatMessageToPage = function (name, message) {
    console.log(name + ' ' + message);
};
$.connection.hub.start().done(function () {
    // Wire up Send button to call NewContosoChatMessage on the server.
    $('#newContosoChatMessage').click(function () {
         contosoChatHubProxy.server.newContosoChatMessage($('#displayname').val(), $('#message').val());
         $('#message').val('').focus();
     });
});
```

Without the generated proxy

```
var connection = $.hubConnection();
var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
contosoChatHubProxy.on('addContosoChatMessageToPage', function(name, message) {
    console.log(name + ' ' + message);
});
connection.start().done(function() {
    // Wire up Send button to call NewContosoChatMessage on the server.
    $('#newContosoChatMessage').click(function () {
        contosoChatHubProxy.invoke('newContosoChatMessage', $('#displayname').val(), $('#message').val());
        $('#message').val('').focus();
                });
    });
```

When to use the generated proxy

If you want to register multiple event handlers for a client method that the server calls, you can't use the generated proxy. Otherwise, you can choose to use the generated proxy or not based on your coding preference. If you choose not to use it, you don't have to reference the "signalr/hubs" URL in a `script` element in your client code.

Client setup

A JavaScript client requires references to jQuery and the SignalR core JavaScript file. The jQuery version must be 1.6.4 or major later versions, such as 1.7.2, 1.8.2, or 1.9.1. If you decide to use the generated proxy, you also need a reference to the SignalR generated proxy JavaScript file. The following example shows what the references might look like in an HTML page that uses the generated proxy.

```
<script src="Scripts/jquery-1.6.4.min.js"></script>
<script src="Scripts/jquery.signalR-1.0.1.min.js"></script>
<script src="signalr/hubs"></script>
```

These references must be included in this order: jQuery first, SignalR core after that, and SignalR proxies last.

How to reference the dynamically generated proxy

In the preceding example, the reference to the SignalR generated proxy is to dynamically generated JavaScript code, not to a physical file. SignalR creates the JavaScript code for the proxy on the fly and serves it to the client in response to the "/signalr/hubs" URL. If you specified a different base URL for SignalR connections on the server in your `MapHubs` method, the URL for the dynamically generated proxy file is your custom URL with "/hubs" appended to it.

> **NOTE**
> For Windows 8 (Windows Store) JavaScript clients, use the physical proxy file instead of the dynamically generated one. For more information, see How to create a physical file for the SignalR generated proxy later in this topic.

In an ASP.NET MVC 4 Razor view, use the tilde to refer to the application root in your proxy file reference:

```
<script src="~/signalr/hubs"></script>
```

For more information about using SignalR in MVC 4, see Getting Started with SignalR and MVC 4.

In an ASP.NET MVC 3 Razor view, use `Url.Content` for your proxy file reference:

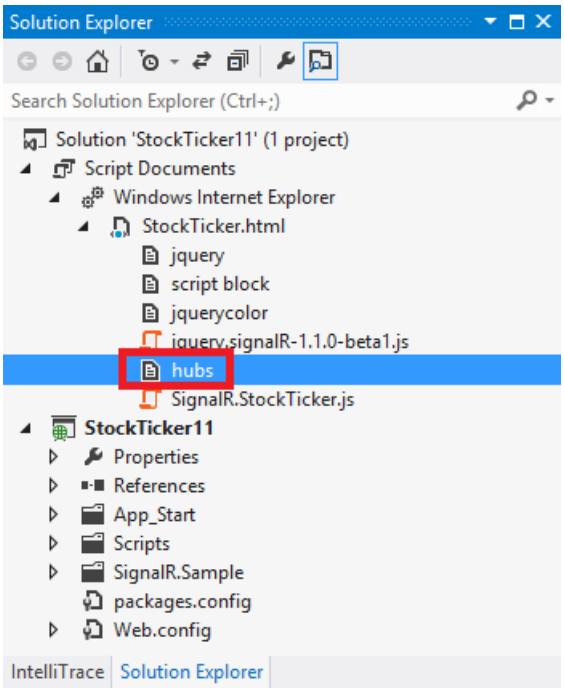
```
<script src="@Url.Content("~/signalr/hubs")"></script>
```

In an ASP.NET Web Forms application, use `ResolveClientUrl` for your proxies file reference or register it via the ScriptManager using an app root relative path (starting with a tilde):

```
<script src='<%: ResolveClientUrl("~/signalr/hubs") %>'></script>
```

As a general rule, use the same method for specifying the "/signalr/hubs" URL that you use for CSS or JavaScript files. If you specify a URL without using a tilde, in some scenarios your application will work correctly when you test in Visual Studio using IIS Express but will fail with a 404 error when you deploy to full IIS. For more information, see **Resolving References to Root-Level Resources** in Web Servers in Visual Studio for ASP.NET Web Projects on the MSDN site.

When you run a web project in Visual Studio 2012 in debug mode, and if you use Internet Explorer as your browser, you can see the proxy file in **Solution Explorer** under **Script Documents**, as shown in the following illustration.



To see the contents of the file, double-click **hubs**. If you are not using Visual Studio 2012 and Internet Explorer, or if you are not in debug mode, you can also get the contents of the file by browsing to the "/signalR/hubs" URL. For example, if your site is running at `http://localhost:56699`, go to `http://localhost:56699/SignalR/hubs` in your browser.

How to create a physical file for the SignalR generated proxy

As an alternative to the dynamically generated proxy, you can create a physical file that has the proxy code and reference that file. You might want to do that for control over caching or bundling behavior, or to get IntelliSense when you are coding calls to server methods.

To create a proxy file, perform the following steps:

1. Install the Microsoft.AspNet.SignalR.Utils NuGet package.

2. Open a command prompt and browse to the *tools* folder that contains the SignalR.exe file. The tools folder is at the following location:

   ```
   [your solution folder]\packages\Microsoft.AspNet.SignalR.Utils.1.0.1\tools
   ```

3. Enter the following command:

   ```
   signalr ghp /path:[path to the .dll that contains your Hub class]
   ```

The path to your *.dll* is typically the *bin* folder in your project folder.

This command creates a file named *server.js* in the same folder as *signalr.exe*.

4. Put the *server.js* file in an appropriate folder in your project, rename it as appropriate for your application, and add a reference to it in place of the "signalr/hubs" reference.

How to establish a connection

Before you can establish a connection, you have to create a connection object, create a proxy, and register event handlers for methods that can be called from the server. When the proxy and event handlers are set up, establish the connection by calling the `start` method.

If you are using the generated proxy, you don't have to create the connection object in your own code because the generated proxy code does it for you.

Establish a connection (with the generated proxy)

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addContosoChatMessageToPage = function (name, message) {
    console.log(userName + ' ' + message);
};
$.connection.hub.start()
    .done(function(){ console.log('Now connected, connection ID=' + $.connection.hub.id); })
    .fail(function(){ console.log('Could not Connect!'); });
});
```

Establish a connection (without the generated proxy)

```
var connection = $.hubConnection();
var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
contosoChatHubProxy.on('addContosoChatMessageToPage', function(userName, message) {
    console.log(userName + ' ' + message);
});
connection.start()
    .done(function(){ console.log('Now connected, connection ID=' + connection.id); })
    .fail(function(){ console.log('Could not connect'); });
```

The sample code uses the default "/signalr" URL to connect to your SignalR service. For information about how to specify a different base URL, see ASP.NET SignalR Hubs API Guide - Server - The /signalr URL.

> **NOTE**
> Normally you register event handlers before calling the `start` method to establish the connection. If you want to register some event handlers after establishing the connection, you can do that, but you must register at least one of your event handler(s) before calling the `start` method. One reason for this is that there can be many Hubs in an application, but you wouldn't want to trigger the `OnConnected` event on every Hub if you are only going to use to one of them. When the connection is established, the presence of a client method on a Hub's proxy is what tells SignalR to trigger the `OnConnected` event. If you don't register any event handlers before calling the `start` method, you will be able to invoke methods on the Hub, but the Hub's `OnConnected` method won't be called and no client methods will be invoked from the server.

$.connection.hub is the same object that $.hubConnection() creates

As you can see from the examples, when you use the generated proxy, `$.connection.hub` refers to the connection object. This is the same object that you get by calling `$.hubConnection()` when you aren't using the generated proxy. The generated proxy code creates the connection for you by executing the following statement:

```
// This line is at the end of the generated proxy code.
signalR.hub = $.hubConnection("/signalr", { useDefaultPath: false });
```

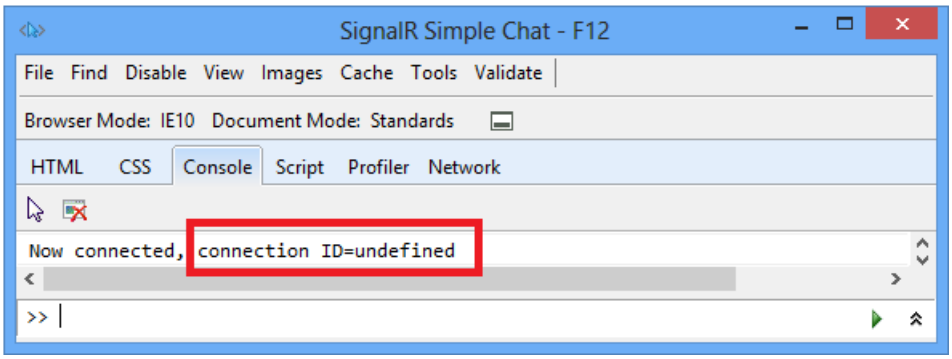
When you're using the generated proxy, you can do anything with `$.connection.hub` that you can do with a connection object when you're not using the generated proxy.

Asynchronous execution of the start method

The `start` method executes asynchronously. It returns a jQuery Deferred object, which means that you can add callback functions by calling methods such as `pipe`, `done`, and `fail`. If you have code that you want to execute after the connection is established, such as a call to a server method, put that code in a callback function or call it from a callback function. The `.done` callback method is executed after the connection has been established, and after any code that you have in your `OnConnected` event handler method on the server finishes executing.

If you put the "Now connected" statement from the preceding example as the next line of code after the `start` method call (not in a `.done` callback), the `console.log` line will execute before the connection is established, as shown in the following example:

```
$.connection.hub.start();
console.log("Now connected, connection ID=" + $.connection.hub.id);
```



How to establish a cross-domain connection

Typically if the browser loads a page from `http://contoso.com`, the SignalR connection is in the same domain, at `http://contoso.com/signalr`. If the page from `http://contoso.com` makes a connection to `http://fabrikam.com/signalr`, that is a cross-domain connection. For security reasons, cross-domain connections are disabled by default. To establish a cross-domain connection, make sure that cross-domain connections are enabled on the server, and specify the connection URL when you create the connection object. SignalR will use the appropriate technology for cross-domain connections, such as JSONP or CORS.

On the server, enable cross-domain connections by selecting that option when you call the `MapHubs` method.

```
var hubConfiguration = new HubConfiguration();
hubConfiguration.EnableCrossDomain = true;
RouteTable.Routes.MapHubs(hubConfiguration);
```

On the client, specify the URL when you create the connection object (without the generated proxy) or before you call the start method (with the generated proxy).

Client code that specifies a cross-domain connection (with the generated proxy)

```
$.connection.hub.url = 'http://fabrikam.com/signalr'
$.connection.hub.start().done(init);
```

Client code that specifies a cross-domain connection (without the generated proxy)

```
var connection = $.hubConnection('http://fabrikam.com/');
```

When you use the `$.hubConnection` constructor, you don't have to include `signalr` in the URL because it is added automatically (unless you specify `useDefaultUrl` as `false`).

You can create multiple connections to different endpoints.

```
var connection1 = $.hubConnection('http://contoso.com/');
var connection2 = $.hubConnection('http://fabrikam.com/');
```

NOTE

- Don't set `jQuery.support.cors` to true in your code.

  ```
  // Don't put the following line in your code.
  jQuery.support.cors = true;
  $.connection.hub.start();
  ```

  SignalR handles the use of JSONP or CORS. Setting `jQuery.support.cors` to true disables JSONP because it causes SignalR to assume the browser supports CORS.

- When you're connecting to a localhost URL, Internet Explorer 10 won't consider it a cross-domain connection, so the application will work locally with IE 10 even if you haven't enabled cross-domain connections on the server.
- For information about using cross-domain connections with Internet Explorer 9, see this StackOverflow thread.
- For information about using cross-domain connections with Chrome, see this StackOverflow thread.
- The sample code uses the default "/signalr" URL to connect to your SignalR service. For information about how to specify a different base URL, see ASP.NET SignalR Hubs API Guide - Server - The /signalr URL.

How to configure the connection

Before you establish a connection, you can specify query string parameters or specify the transport method.

How to specify query string parameters

If you want to send data to the server when the client connects, you can add query string parameters to the connection object. The following examples show how to set a query string parameter in client code.

Set a query string value before calling the start method (with the generated proxy)

```
$.connection.hub.qs = { 'version' : '1.0' };
```

Set a query string value before calling the start method (without the generated proxy)

```
var connection = $.hubConnection();
connection.qs = { 'version' : '1.0' };
```

The following example shows how to read a query string parameter in server code.

```
public class ContosoChatHub : Hub
{
    public override Task OnConnected()
    {
        var version = Context.QueryString['version'];
        if (version != '1.0')
        {
            Clients.Caller.notifyWrongVersion();
        }
        return base.OnConnected();
    }
}
```

How to specify the transport method

As part of the process of connecting, a SignalR client normally negotiates with the server to determine the best transport that is supported by both server and client. If you already know which transport you want to use, you can bypass this negotiation process by specifying the transport method when you call the `start` method.

Client code that specifies the transport method (with the generated proxy)

```
$.connection.hub.start( { transport: 'longPolling' });
```

Client code that specifies the transport method (without the generated proxy)

```
var connection = $.hubConnection();
connection.start({ transport: 'longPolling' });
```

As an alternative, you can specify multiple transport methods in the order in which you want SignalR to try them:

Client code that specifies a custom transport fallback scheme (with the generated proxy)

```
$.connection.hub.start( { transport: ['webSockets', 'longPolling'] });
```

Client code that specifies a custom transport fallback scheme (without the generated proxy)

```
var connection = $.hubConnection();
connection.start({ transport: ['webSockets', 'longPolling'] });
```

You can use the following values for specifying the transport method:

- "webSockets"
- "foreverFrame"
- "serverSentEvents"
- "longPolling"

The following examples show how to find out which transport method is being used by a connection.

Client code that displays the transport method used by a connection (with the generated proxy)

```
$.connection.hub.start().done(function () {
    console.log("Connected, transport = " + $.connection.hub.transport.name);
});
```

Client code that displays the transport method used by a connection (without the generated proxy)

```
var connection = $.hubConnection();
connection.hub.start().done(function () {
    console.log("Connected, transport = " + connection.transport.name);
});
```

For information about how to check the transport method in server code, see ASP.NET SignalR Hubs API Guide - Server - How to get information about the client from the Context property. For more information about transports and fallbacks, see Introduction to SignalR - Transports and Fallbacks.

How to get a proxy for a Hub class

Each connection object that you create encapsulates information about a connection to a SignalR service that contains one or more Hub classes. To communicate with a Hub class, you use a proxy object which you create yourself (if you're not using the generated proxy) or which is generated for you.

On the client the proxy name is a camel-cased version of the Hub class name. SignalR automatically makes this change so that JavaScript code can conform to JavaScript conventions.

Hub class on server

```
public class ContosoChatHub : Hub
```

Get a reference to the generated client proxy for the Hub

```
var myHubProxy = $.connection.contosoChatHub
```

Create client proxy for the Hub class (without generated proxy)

```
var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
```

If you decorate your Hub class with a `HubName` attribute, use the exact name without changing case.

Hub class on server with HubName attribute

```
[HubName("ContosoChatHub")]
public class ChatHub : Hub
```

Get a reference to the generated client proxy for the Hub

```
var contosoChatHubProxy = $.connection.ContosoChatHub
```

Create client proxy for the Hub class (without generated proxy)

```
var contosoChatHubProxy = connection.createHubProxy('ContosoChatHub');
```

How to define methods on the client that the server can call

To define a method that the server can call from a Hub, add an event handler to the Hub proxy by using the `client` property of the generated proxy, or call the `on` method if you aren't using the generated proxy. The parameters can be complex objects.

Add the event handler before you call the `start` method to establish the connection. (If you want to add event handlers after calling the `start` method, see the note in How to establish a connection earlier in this document, and use the syntax shown for defining a method without using the generated proxy.)

Method name matching is case-insensitive. For example, `Clients.All.addContosoChatMessageToPage` on the server will execute `AddContosoChatMessageToPage`, `addContosoChatMessageToPage`, or `addcontosochatmessagetopage` on the client.

Define method on client (with the generated proxy)

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addContosoChatMessageToPage = function (userName, message) {
    console.log(userName + ' ' + message);
};
$.connection.hub.start()
    .done(function(){ console.log('Now connected, connection ID=' + $.connection.hub.id); })
    .fail(function(){ console.log('Could not Connect!'); });
});
```

Alternate way to define method on client (with the generated proxy)

```
$.extend(contosoChatHubProxy.client, {
    addContosoChatMessageToPage: function(userName, message) {
    console.log(userName + ' ' + message);
    };
});
```

Define method on client (without the generated proxy, or when adding after calling the start method)

```
var connection = $.hubConnection();
var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
contosoChatHubProxy.on('addContosoChatMessageToPage', function(userName, message) {
    console.log(userName + ' ' + message);
});
connection.start()
    .done(function(){ console.log('Now connected, connection ID=' + connection.id); })
    .fail(function(){ console.log('Could not connect'); });
```

Server code that calls the client method

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addContosoChatMessageToPage(name, message);
    }
}
```

The following examples include a complex object as a method parameter.

Define method on client that takes a complex object (with the generated proxy)

```
var contosoChatHubProxy = $.connection.contosoChatHub;
contosoChatHubProxy.client.addMessageToPage = function (message) {
    console.log(message.UserName + ' ' + message.Message);
});
```

Define method on client that takes a complex object (without the generated proxy)

```
var connection = $.hubConnection();
var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
chatHubProxy.on('addMessageToPage', function (message) {
    console.log(message.UserName + ' ' + message.Message);
});
```

Server code that defines the complex object

```
public class ContosoChatMessage
{
    public string UserName { get; set; }
    public string Message { get; set; }
}
```

Server code that calls the client method using a complex object

```
public void SendMessage(string name, string message)
{
    Clients.All.addContosoChatMessageToPage(new ContosoChatMessage() { UserName = name, Message = message });
}
```

How to call server methods from the client

To call a server method from the client, use the `server` property of the generated proxy or the `invoke` method on the Hub proxy if you aren't using the generated proxy. The return value or parameters can be complex objects.

Pass in a camel-case version of the method name on the Hub. SignalR automatically makes this change so that JavaScript code can conform to JavaScript conventions.

The following examples show how to call a server method that doesn't have a return value and how to call a server method that does have a return value.

Server method with no HubMethodName attribute

```
public class ContosoChatHub : Hub
{
    public void NewContosoChatMessage(ChatMessage message)
    {
        Clients.All.addContosoChatMessageToPage(message);
    }
}
```

Server code that defines the complex object passed in a parameter

```
public class ChatMessage
{
    public string UserName { get; set; }
    public string Message { get; set; }
}
```

Client code that invokes the server method (with the generated proxy)

```
contosoChatHubProxy.server.newContosoChatMessage({ UserName: userName, Message: message}).done(function () {
        console.log ('Invocation of NewContosoChatMessage succeeded');
    }).fail(function (error) {
        console.log('Invocation of NewContosoChatMessage failed. Error: ' + error);
    });
```

Client code that invokes the server method (without the generated proxy)

```
contosoChatHubProxy.invoke('newContosoChatMessage', { UserName: userName, Message: message}).done(function ()
{
        console.log ('Invocation of NewContosoChatMessage succeeded');
    }).fail(function (error) {
        console.log('Invocation of NewContosoChatMessage failed. Error: ' + error);
    });
```

If you decorated the Hub method with a `HubMethodName` attribute, use that name without changing case.

Server method with a HubMethodName attribute

```
public class ContosoChatHub : Hub
{
    [HubMethodName("NewContosoChatMessage")]
    public void NewContosoChatMessage(string name, string message)
    {
        Clients.All.addContosoChatMessageToPage(name, message);
    }
}
```

Client code that invokes the server method (with the generated proxy)

```
contosoChatHubProxy.server.NewContosoChatMessage(userName, message).done(function () {
        console.log ('Invocation of NewContosoChatMessage succeeded');
    }).fail(function (error) {
        console.log('Invocation of NewContosoChatMessage failed. Error: ' + error);
    });
```

Client code that invokes the server method (without the generated proxy)

```
contosoChatHubProxy.invoke('NewContosoChatMessage', userName, message).done(function () {
        console.log ('Invocation of NewContosoChatMessage succeeded');
    }).fail(function (error) {
        console.log('Invocation of NewContosoChatMessage failed. Error: ' + error);
    });
```

The preceding examples show how to call a server method that has no return value. The following examples show how to call a server method that has a return value.

Server code for a method that has a return value

```
public class StockTickerHub : Hub
{
    public IEnumerable<Stock> GetAllStocks()
    {
        return _stockTicker.GetAllStocks();
    }
}
```

The Stock class used for the return value

```
public class Stock
{
    public string Symbol { get; set; }
    public decimal Price { get; set; }
}
```

Client code that invokes the server method (with the generated proxy)

```
function init() {
    return stockTickerProxy.server.getAllStocks().done(function (stocks) {
        $.each(stocks, function () {
            var stock = this;
            console.log("Symbol=" + stock.Symbol + " Price=" + stock.Price);
        });
    }).fail(function (error) {
        console.log('Error: ' + error);
    });
}
```

Client code that invokes the server method (without the generated proxy)

```
function init() {
    return stockTickerProxy.invoke('getAllStocks').done(function (stocks) {
        $.each(stocks, function () {
            var stock = this;
            console.log("Symbol=" + stock.Symbol + " Price=" + stock.Price);
        });
    }).fail(function (error) {
        console.log('Error: ' + error);
    });
}
```

How to handle connection lifetime events

SignalR provides the following connection lifetime events that you can handle:

- `starting`: Raised before any data is sent over the connection.
- `received`: Raised when any data is received on the connection. Provides the received data.
- `connectionSlow`: Raised when the client detects a slow or frequently dropping connection.
- `reconnecting`: Raised when the underlying transport begins reconnecting.
- `reconnected`: Raised when the underlying transport has reconnected.
- `stateChanged`: Raised when the connection state changes. Provides the old state and the new state (Connecting, Connected, Reconnecting, or Disconnected).
- `disconnected`: Raised when the connection has disconnected.

For example, if you want to display warning messages when there are connection problems that might cause noticeable delays, handle the `connectionSlow` event.

Handle the connectionSlow event (with the generated proxy)

```
$.connection.hub.connectionSlow(function () {
    console.log('We are currently experiencing difficulties with the connection.')
});
```

Handle the connectionSlow event (without the generated proxy)

```
var connection = $.hubConnection();
connection.connectionSlow(function () {
    console.log('We are currently experiencing difficulties with the connection.')
});
```

For more information, see Understanding and Handling Connection Lifetime Events in SignalR.

How to handle errors

The SignalR JavaScript client provides an `error` event that you can add a handler for. You can also use the fail method to add a handler for errors that result from a server method invocation.

If you don't explicitly enable detailed error messages on the server, the exception object that SignalR returns after an error contains minimal information about the error. For example, if a call to `newContosoChatMessage` fails, the error message in the error object contains "`There was an error invoking Hub method 'contosoChatHub.newContosoChatMessage'.`" Sending detailed error messages to clients in production is not recommended for security reasons, but if you want to enable detailed error messages for troubleshooting purposes, use the following code on the server.

```
var hubConfiguration = new HubConfiguration();
hubConfiguration.EnableDetailedErrors = true;
RouteTable.Routes.MapHubs(hubConfiguration);
```

The following example shows how to add a handler for the error event.

Add an error handler (with the generated proxy)

```
$.connection.hub.error(function (error) {
    console.log('SignalR error: ' + error)
});
```

Add an error handler (without the generated proxy)

```
var connection = $.hubConnection();
connection.error(function (error) {
    console.log('SignalR error: ' + error)
});
```

The following example shows how to handle an error from a method invocation.

Handle an error from a method invocation (with the generated proxy)

```
contosoChatHubProxy.newContosoChatMessage(userName, message)
    .fail(function(error) {
        console.log( 'newContosoChatMessage error: ' + error)
    });
```

Handle an error from a method invocation (without the generated proxy)

```
contosoChatHubProxy.invoke('newContosoChatMessage', userName, message)
    .fail(function(error) {
        console.log( 'newContosoChatMessage error: ' + error)
    });
```

If a method invocation fails, the `error` event is also raised, so your code in the `error` method handler and in the `.fail` method callback would execute.

How to enable client-side logging

To enable client-side logging on a connection, set the `logging` property on the connection object before you call the `start` method to establish the connection.

Enable logging (with the generated proxy)

```
$.connection.hub.logging = true;
$.connection.hub.start();
```

Enable logging (without the generated proxy)

```
var connection = $.hubConnection();
connection.logging = true;
connection.start();
```

To see the logs, open your browser's developer tools and go to the Console tab. For a tutorial that shows step-by-step instructions and screen shots that show how to do this, see Server Broadcast with ASP.NET Signalr - Enable Logging.

ASP.NET SignalR Hubs API Guide - .NET Client (SignalR 1.x)

2/20/2020 • 14 minutes to read • Edit Online

by Patrick Fletcher, Tom Dykstra

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This document provides an introduction to using the Hubs API for SignalR version 2 in .NET clients, such as Windows Store (WinRT), WPF, Silverlight, and console applications.

The SignalR Hubs API enables you to make remote procedure calls (RPCs) from a server to connected clients and from clients to the server. In server code, you define methods that can be called by clients, and you call methods that run on the client. In client code, you define methods that can be called from the server, and you call methods that run on the server. SignalR takes care of all of the client-to-server plumbing for you.

SignalR also offers a lower-level API called Persistent Connections. For an introduction to SignalR, Hubs, and Persistent Connections, or for a tutorial that shows how to build a complete SignalR application, see SignalR - Getting Started.

Overview

This document contains the following sections:

- Client Setup
- How to establish a connection
  - Cross-domain connections from Silverlight clients
- How to configure the connection
  - How to set the maximum number of concurrent connections in WPF clients
  - How to specify query string parameters
  - How to specify the transport method
  - How to specify HTTP headers
  - How to specify client certificates
- How to create the Hub proxy
- How to define methods on the client that the server can call
  - Methods without parameters
  - Methods with parameters, specifying parameter types
  - Methods with parameters, specifying dynamic objects for the parameters
  - How to remove a handler
- How to call server methods from the client
- How to handle connection lifetime events

- How to handle errors
- How to enable client-side logging
- WPF, Silverlight, and console application code samples for client methods that the server can call

For a sample .NET client projects, see the following resources:

- gustavo-armenta / SignalR-Samples on GitHub.com (WinRT, Silverlight, console app examples).
- DamianEdwards / SignalR-MoveShapeDemo / MoveShape.Desktop on GitHub.com (WPF example).
- SignalR / Microsoft.AspNet.SignalR.Client.Samples on GitHub.com (Console app example).

For documentation on how to program the server or JavaScript clients, see the following resources:

- SignalR Hubs API Guide - Server
- SignalR Hubs API Guide - JavaScript Client

Links to API Reference topics are to the .NET 4.5 version of the API. If you're using .NET 4, see the .NET 4 version of the API topics.

Client setup

Install the Microsoft.AspNet.SignalR.Client NuGet package (not the Microsoft.AspNet.SignalR package). This package supports WinRT, Silverlight, WPF, console application, and Windows Phone clients, for both .NET 4 and .NET 4.5.

If the version of SignalR that you have on the client is different from the version that you have on the server, SignalR is often able to adapt to the difference. For example, when SignalR version 2.0 is released and you install that on the server, the server will support clients that have 1.1.x installed as well as clients that have 2.0 installed. If the difference between the version on the server and the version on the client is too great, SignalR throws an `InvalidOperationException` exception when the client tries to establish a connection. The error message is "

```
You are using a version of the client that isn't compatible with the server. Client version X.X, server version X.X
```

".

How to establish a connection

Before you can establish a connection, you have to create a `HubConnection` object and create a proxy. To establish the connection, call the `Start` method on the `HubConnection` object.

```
var hubConnection = new HubConnection("http://www.contoso.com/");
IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new price {1}", stock.Symbol, stock.Price));
await hubConnection.Start();
```

NOTE

For JavaScript clients you have to register at least one event handler before calling the `Start` method to establish the connection. This is not necessary for .NET clients. For JavaScript clients, the generated proxy code automatically creates proxies for all Hubs that exist on the server, and registering a handler is how you indicate which Hubs your client intends to use. But for a .NET client you create Hub proxies manually, so SignalR assumes that you will be using any Hub that you create a proxy for.

The sample code uses the default "/signalr" URL to connect to your SignalR service. For information about how to specify a different base URL, see ASP.NET SignalR Hubs API Guide - Server - The /signalr URL.

The `Start` method executes asynchronously. To make sure that subsequent lines of code don't execute until after the connection is established, use `await` in an ASP.NET 4.5 asynchronous method or `.Wait()` in a synchronous method. Don't use `.Wait()` in a WinRT client.

```
await connection.Start();
```

```
connection.Start().Wait();
```

The `HubConnection` class is thread-safe.

Cross-domain connections from Silverlight clients

For information about how to enable cross-domain connections from Silverlight clients, see Making a Service Available Across Domain Boundaries.

How to configure the connection

Before you establish a connection, you can specify any of the following options:

- Concurrent connections limit.
- Query string parameters.
- The transport method.
- HTTP headers.
- Client certificates.

How to set the maximum number of concurrent connections in WPF clients

In WPF clients, you might have to increase the maximum number of concurrent connections from its default value of 2. The recommended value is 10.

```
var hubConnection = new HubConnection("http://www.contoso.com/");
IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new price
{1}", stock.Symbol, stock.Price));
ServicePointManager.DefaultConnectionLimit = 10;
await hubConnection.Start();
```

For more information, see ServicePointManager.DefaultConnectionLimit.

How to specify query string parameters

If you want to send data to the server when the client connects, you can add query string parameters to the connection object. The following example shows how to set a query string parameter in client code.

```
var querystringData = new Dictionary<string, string>();
querystringData.Add("contosochatversion", "1.0");
var connection = new HubConnection("http://contoso.com/", querystringData);
```

The following example shows how to read a query string parameter in server code.

```
public class StockTickerHub : Hub
{
    public override Task OnConnected()
    {
        var version = Context.QueryString["contosochatversion"];
        if (version != "1.0")
        {
            Clients.Caller.notifyWrongVersion();
        }
        return base.OnConnected();
    }
}
```

How to specify the transport method

As part of the process of connecting, a SignalR client normally negotiates with the server to determine the best transport that is supported by both server and client. If you already know which transport you want to use, you can bypass this negotiation process. To specify the transport method, pass in a transport object to the Start method. The following example shows how to specify the transport method in client code.

```
var hubConnection = new HubConnection("http://www.contoso.com/");
IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new price
{1}", stock.Symbol, stock.Price));
await hubConnection.Start(new LongPollingTransport());
```

The Microsoft.AspNet.SignalR.Client.Transports namespace includes the following classes that you can use to specify the transport.

- LongPollingTransport
- ServerSentEventsTransport
- WebSocketTransport (Available only when both server and client use .NET 4.5.)
- AutoTransport (Automatically chooses the best transport that is supported by both the client and the server. This is the default transport. Passing this in to the `Start` method has the same effect as not passing in anything.)

The ForeverFrame transport is not included in this list because it is used only by browsers.

For information about how to check the transport method in server code, see ASP.NET SignalR Hubs API Guide - Server - How to get information about the client from the Context property. For more information about transports and fallbacks, see Introduction to SignalR - Transports and Fallbacks.

How to specify HTTP headers

To set HTTP headers, use the `Headers` property on the connection object. The following example shows how to add an HTTP header.

```
hubConnection = new hubConnection("http://www.contoso.com/");
connection.Headers.Add("headername", "headervalue");
IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new price
{1}", stock.Symbol, stock.Price));
await connection.Start();
```

How to specify client certificates

To add client certificates, use the `AddClientCertificate` method on the connection object.

```
hubConnection = new hubConnection("http://www.contoso.com/");
hubConnection.AddClientCertificate(X509Certificate.CreateFromCertFile("MyCert.cer"));
IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new price
{1}", stock.Symbol, stock.Price));
await connection.Start();
```

How to create the Hub proxy

In order to define methods on the client that a Hub can call from the server, and to invoke methods on a Hub at the server, create a proxy for the Hub by calling `CreateHubProxy` on the connection object. The string you pass in to `CreateHubProxy` is the name of your Hub class, or the name specified by the `HubName` attribute if one was used on the server. Name matching is case-insensitive.

Hub class on server

```
public class StockTickerHub : Hub
```

Create client proxy for the Hub class

```
var hubConnection = new HubConnection("http://www.contoso.com/");
IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new price
{1}", stock.Symbol, stock.Price));
await hubConnection.Start();
```

If you decorate your Hub class with a `HubName` attribute, use that name.

Hub class on server

```
[HubName("stockTicker")]
public class StockTickerHub : Hub
```

Create client proxy for the Hub class

```
var hubConnection = new HubConnection("http://www.contoso.com/");
IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("stockTicker");
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
    Console.WriteLine("Stock update for {0} new price {1}", stock.Symbol, stock.Price));
await hubConnection.Start();
```

The proxy object is thread-safe. In fact, if you call `HubConnection.CreateHubProxy` multiple times with the same `hubName`, you get the same cached `IHubProxy` object.

How to define methods on the client that the server can call

To define a method that the server can call, use the proxy's `On` method to register an event handler.

Method name matching is case-insensitive. For example, `Clients.All.UpdateStockPrice` on the server will execute `updateStockPrice`, `updatestockprice`, or `UpdateStockPrice` on the client.

Different client platforms have different requirements for how you write method code to update the UI. The examples shown are for WinRT (Windows Store .NET) clients. WPF, Silverlight, and console application examples are provided in a separate section later in this topic.

Methods without parameters

If the method you're handling does not have parameters, use the non-generic overload of the `On` method:

Server code calling client method without parameters

```
public class StockTickerHub : Hub
{
    public void NotifyAllClients()
    {
         Clients.All.Notify();
    }
}
```

WinRT Client code for method called from server without parameters (see WPF and Silverlight examples later in this topic)

```
var hubConnection = new HubConnection("http://www.contoso.com/");
IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
stockTickerHub.On("notify", () =>
    // Context is a reference to SynchronizationContext.Current
    Context.Post(delegate
    {
        textBox.Text += "Notified!\n";
    }, null)
);
await hubConnection.Start();
```

Methods with parameters, specifying the parameter types

If the method you're handling has parameters, specify the types of the parameters as the generic types of the `On` method. There are generic overloads of the `On` method to enable you to specify up to 8 parameters (4 on Windows Phone 7). In the following example, one parameter is sent to the `UpdateStockPrice` method.

Server code calling client method with a parameter

```
public void BroadcastStockPrice(Stock stock)
{
    context.Clients.Others.UpdateStockPrice(stock);
}
```

The Stock class used for the parameter

```
public class Stock
{
    public string Symbol { get; set; }
    public decimal Price { get; set; }
}
```

WinRT Client code for a method called from server with a parameter (see WPF and Silverlight examples later in this topic)

```
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
    // Context is a reference to SynchronizationContext.Current
    Context.Post(delegate
    {
        textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
    }, null)
);
```

Methods with parameters, specifying dynamic objects for the parameters

As an alternative to specifying parameters as generic types of the `On` method, you can specify parameters as dynamic objects:

Server code calling client method with a parameter

```
public void BroadcastStockPrice(Stock stock)
{
    context.Clients.Others.UpdateStockPrice(stock);
}
```

The Stock class used for the parameter

```
public class Stock
{
    public string Symbol { get; set; }
    public decimal Price { get; set; }
}
```

WinRT Client code for a method called from server with a parameter, using a dynamic object for the parameter (see WPF and Silverlight examples later in this topic)

```
stockTickerHubProxy.On("UpdateStockPrice", stock =>
    // Context is a reference to SynchronizationContext.Current
    Context.Post(delegate
    {
        textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
    }, null)
);
```

How to remove a handler

To remove a handler, call its `Dispose` method.

Client code for a method called from server

```
var updateStockPriceHandler = stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
    Context.Post(delegate
    {
        textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
    }, null)
);
```

Client code to remove the handler

```
updateStockPriceHandler.Dispose();
```

How to call server methods from the client

To call a method on the server, use the `Invoke` method on the Hub proxy.

If the server method has no return value, use the non-generic overload of the `Invoke` method.

Server code for a method that has no return value

```
public class StockTickerHub : Hub
{
    public void JoinGroup(string groupName)
    {
        Groups.Add(Context.ConnectionId, groupName);
    }
}
```

Client code calling a method that has no return value

```
stockTickerHubProxy.Invoke("JoinGroup", hubConnection.ConnectionID, "SignalRChatRoom");
```

If the server method has a return value, specify the return type as the generic type of the `Invoke` method.

Server code for a method that has a return value and takes a complex type parameter

```
public IEnumerable<Stock> AddStock(Stock stock)
{
    _stockTicker.AddStock(stock);
    return _stockTicker.GetAllStocks();
}
```

The Stock class used for the parameter and return value

```
public class Stock
{
    public string Symbol { get; set; }
    public decimal Price { get; set; }
}
```

Client code calling a method that has a return value and takes a complex type parameter, in an ASP.NET 4.5 async method

```
var stocks = await stockTickerHub.Invoke<IEnumerable<Stock>>("AddStock", new Stock() { Symbol = "MSFT" });
foreach (Stock stock in stocks)
{
    textBox.Text += string.Format("Symbol: {0} price: {1}\n", stock.Symbol, stock.Price);
}
```

Client code calling a method that has a return value and takes a complex type parameter, in a synchronous method

```
var stocks = stockTickerHub.Invoke<IEnumerable<Stock>>("AddStock", new Stock() { Symbol = "MSFT" }).Result;
foreach (Stock stock in stocks)
{
    textBox.Text += string.Format("Symbol: {0} price: {1}\n", stock.Symbol, stock.Price);
}
```

The `Invoke` method executes asynchronously and returns a `Task` object. If you don't specify `await` or `.Wait()`, the next line of code will execute before the method that you invoke has finished executing.

How to handle connection lifetime events

SignalR provides the following connection lifetime events that you can handle:

- `Received`: Raised when any data is received on the connection. Provides the received data.
- `ConnectionSlow`: Raised when the client detects a slow or frequently dropping connection.
- `Reconnecting`: Raised when the underlying transport begins reconnecting.
- `Reconnected`: Raised when the underlying transport has reconnected.
- `StateChanged`: Raised when the connection state changes. Provides the old state and the new state. For information about connection state values see ConnectionState Enumeration.
- `Closed`: Raised when the connection has disconnected.

For example, if you want to display warning messages for errors that are not fatal but cause intermittent connection problems, such as slowness or frequent dropping of the connection, handle the `ConnectionSlow` event.

```
hubConnection.ConnectionSlow += () => Console.WriteLine("Connection problems.");
```

For more information, see Understanding and Handling Connection Lifetime Events in SignalR.

How to handle errors

If you don't explicitly enable detailed error messages on the server, the exception object that SignalR returns after an error contains minimal information about the error. For example, if a call to `newContosoChatMessage` fails, the error message in the error object contains "`There was an error invoking Hub method 'contosoChatHub.newContosoChatMessage'.`" Sending detailed error messages to clients in production is not recommended for security reasons, but if you want to enable detailed error messages for troubleshooting purposes, use the following code on the server.

```
var hubConfiguration = new HubConfiguration();
hubConfiguration.EnableDetailedErrors = true;
RouteTable.Routes.MapHubs(hubConfiguration);
```

To handle errors that SignalR raises, you can add a handler for the `Error` event on the connection object.

```
hubConnection.Error += ex => Console.WriteLine("SignalR error: {0}", ex.Message);
```

To handle errors from method invocations, wrap the code in a try-catch block.

```
try
{
    IEnumerable<Stock> stocks = await stockTickerHub.Invoke<IEnumerable<Stock>>("GetAllStocks");
    foreach (Stock stock in stocks)
    {
        Console.WriteLine("Symbol: {0} price: {1}", stock.Symbol, stock.Price);
    }
}
catch (Exception ex)
{
    Console.WriteLine("Error invoking GetAllStocks: {0}", ex.Message);
}
```

How to enable client-side logging

To enable client-side logging, set the `TraceLevel` and `TraceWriter` properties on the connection object.

```
var hubConnection = new HubConnection("http://www.contoso.com/");
hubConnection.TraceLevel = TraceLevels.All;
hubConnection.TraceWriter = Console.Out;
IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock => Console.WriteLine("Stock update for {0} new price
{1}", stock.Symbol, stock.Price));
await hubConnection.Start();
```

WPF, Silverlight, and console application code samples for client methods that the server can call

The code samples shown earlier for defining client methods that the server can call apply to WinRT clients. The following samples show the equivalent code for WPF, Silverlight, and console application clients.

Methods without parameters

WPF client code for method called from server without parameters

```
stockTickerHub.On<Stock>("notify", () =>
    Dispatcher.InvokeAsync(() =>
        {
            SignalRTextBlock.Text += string.Format("Notified!");
        })
);
```

Silverlight client code for method called from server without parameters

```
stockTickerHub.On<Stock>("notify", () =>
    // Context is a reference to SynchronizationContext.Current
    Context.Post(delegate
    {
        textBox.Text += "Notified!";
    }, null)
);
```

Console application client code for method called from server without parameters

```
stockTickerHubProxyProxy.On("Notify", () => Console.WriteLine("Notified!"));
```

Methods with parameters, specifying the parameter types

WPF client code for a method called from server with a parameter

```
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
    Dispatcher.InvokeAsync(() =>
        {
            textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
        })
);
```

Silverlight client code for a method called from server with a parameter

```
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
    // Context is a reference to SynchronizationContext.Current
    Context.Post(delegate
    {
        textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
    }, null)
);
```

Console application client code for a method called from server with a parameter

```
stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
    Console.WriteLine("Symbol {0} Price {1}", stock.Symbol, stock.Price));
```

Methods with parameters, specifying dynamic objects for the parameters

WPF client code for a method called from server with a parameter, using a dynamic object for the parameter

```
stockTickerHubProxy.On("UpdateStockPrice", stock =>
    Dispatcher.InvokeAsync(() =>
        {
            textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
        })
);
```

Silverlight client code for a method called from server with a parameter, using a dynamic object for the parameter

```
stockTickerHubProxy.On("UpdateStockPrice", stock =>
    // Context is a reference to SynchronizationContext.Current
    Context.Post(delegate
    {
        textBox.Text += string.Format("Stock update for {0} new price {1}\n", stock.Symbol, stock.Price);
    }, null)
);
```

Console application client code for a method called from server with a parameter, using a dynamic object for the parameter

```
stockTickerHubProxy.On("UpdateStockPrice", stock =>
    Console.WriteLine("Symbol {0} Price {1}", stock.Symbol, stock.Price));
```

Understanding and Handling Connection Lifetime Events in SignalR 1.x

2/20/2020 • 16 minutes to read • Edit Online

by Patrick Fletcher, Tom Dykstra

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This article provides an overview of the SignalR connection, reconnection, and disconnection events that you can handle, and timeout and keepalive settings that you can configure.

The article assumes you already have some knowledge of SignalR and connection lifetime events. For an introduction to SignalR, see SignalR - Overview - Getting Started. For lists of connection lifetime events, see the following resources:

- How to handle connection lifetime events in the Hub class
- How to handle connection lifetime events in JavaScript clients
- How to handle connection lifetime events in .NET clients

Overview

This article contains the following sections:

- Connection lifetime terminology and scenarios
  - SignalR connections, transport connections, and physical connections
  - Transport disconnection scenarios
  - Client disconnection scenarios
  - Server disconnection scenarios
- Timeout and keepalive settings
  - ConnectionTimeout
  - DisconnectTimeout
  - KeepAlive
  - How to change timeout and keepalive settings
- How to notify the user about disconnections
- How to continuously reconnect
- How to disconnect a client in server code

Links to API Reference topics are to the .NET 4.5 version of the API. If you're using .NET 4, see the .NET 4 version of the API topics.

Connection lifetime terminology and scenarios

The `OnReconnected` event handler in a SignalR Hub can execute directly after `OnConnected` but not after

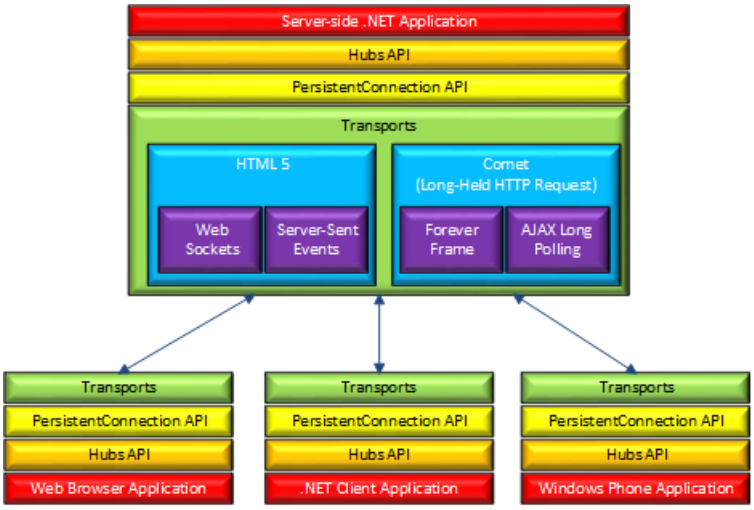
`OnDisconnected` for a given client. The reason you can have a reconnection without a disconnection is that there are several ways in which the word "connection" is used in SignalR.

SignalR connections, transport connections, and physical connections

This article will differentiate between *SignalR connections*, *transport connections*, and *physical connections*:

- **SignalR connection** refers to a logical relationship between a client and a server URL, maintained by the SignalR API and uniquely identified by a connection ID. The data about this relationship is maintained by SignalR and is used to establish a transport connection. The relationship ends and SignalR disposes of the data when the client calls the `Stop` method or a timeout limit is reached while SignalR is attempting to re-establish a lost transport connection.
- **Transport connection** refers to a logical relationship between a client and a server, maintained by one of the four transport APIs: WebSockets, server-sent events, forever frame, or long polling. SignalR uses the transport API to create a transport connection, and the transport API depends on the existence of a physical network connection to create the transport connection. The transport connection ends when SignalR terminates it or when the transport API detects that the physical connection is broken.
- **Physical connection** refers to the physical network links -- wires, wireless signals, routers, etc. -- that facilitate communication between a client computer and a server computer. The physical connection must be present in order to establish a transport connection, and a transport connection must be established in order to establish a SignalR connection. However, breaking the physical connection doesn't always immediately end the transport connection or the SignalR connection, as will be explained later in this topic.

In the following diagram, the SignalR connection is represented by the Hubs API and PersistentConnection API SignalR layer, the transport connection is represented by the Transports layer, and the physical connection is represented by the lines between the server and the clients.



When you call the `Start` method in a SignalR client, you are providing SignalR client code with all the information it needs in order to establish a physical connection to a server. SignalR client code uses this information to make an HTTP request and establish a physical connection that uses one of the four transport methods. If the transport connection fails or the server fails, the SignalR connection doesn't go away immediately because the client still has the information it needs to automatically re-establish a new transport connection to the same SignalR URL. In this scenario, no intervention from the user application is involved, and when the SignalR client code establishes a new transport connection, it does not start a new SignalR connection. The continuity of the SignalR connection is reflected in the fact that the connection ID, which is created when you call the `Start` method, does not change.

The `OnReconnected` event handler on the Hub executes when a transport connection is automatically re-established after having been lost. The `OnDisconnected` event handler executes at the end of a SignalR connection. A SignalR connection can end in any of the following ways:

- If the client calls the `Stop` method, a stop message is sent to the server, and both client and server end the

SignalR connection immediately.

- After connectivity between client and server is lost, the client tries to reconnect and the server waits for the client to reconnect. If the attempts to reconnect are unsuccessful and the disconnect timeout period ends, both client and server end the SignalR connection. The client stops trying to reconnect, and the server disposes of its representation of the SignalR connection.
- If the client stops running without having a chance to call the `Stop` method, the server waits for the client to reconnect, and then ends the SignalR connection after the disconnect timeout period.
- If the server stops running, the client tries to reconnect (re-create the transport connection), and then ends the SignalR connection after the disconnect timeout period.

When there are no connection problems, and the user application ends the SignalR connection by calling the `Stop` method, the SignalR connection and the transport connection begin and end at about the same time. The following sections describe in more detail the other scenarios.

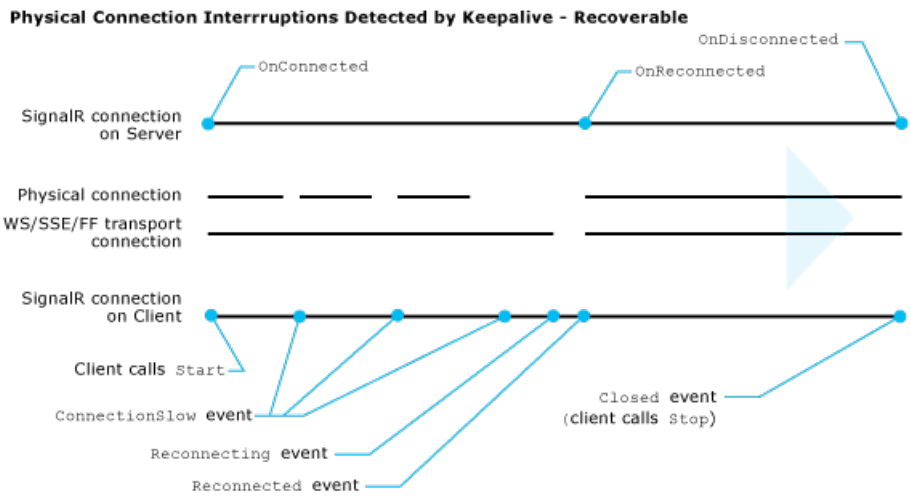
Transport disconnection scenarios

Physical connections might be slow or there might be interruptions in connectivity. Depending on factors such as the length of the interruption, the transport connection might be dropped. SignalR then tries to re-establish the transport connection. Sometimes the transport connection API detects the interruption and drops the transport connection, and SignalR finds out immediately that the connection is lost. In other scenarios, neither the transport connection API nor SignalR becomes aware immediately that connectivity has been lost. For all transports except long polling, the SignalR client uses a function called *keepalive* to check for loss of connectivity that the transport API is unable to detect. For information about long polling connections, see Timeout and keepalive settings later in this topic.

When a connection is inactive, periodically the server sends a keepalive packet to the client. As of the date this article is being written, the default frequency is every 10 seconds. By listening for these packets, clients can tell if there is a connection problem. If a keepalive packet is not received when expected, after a short time the client assumes that there are connection problems such as slowness or interruptions. If the keepalive is still not received after a longer time, the client assumes that the connection has been dropped, and it begins trying to reconnect.

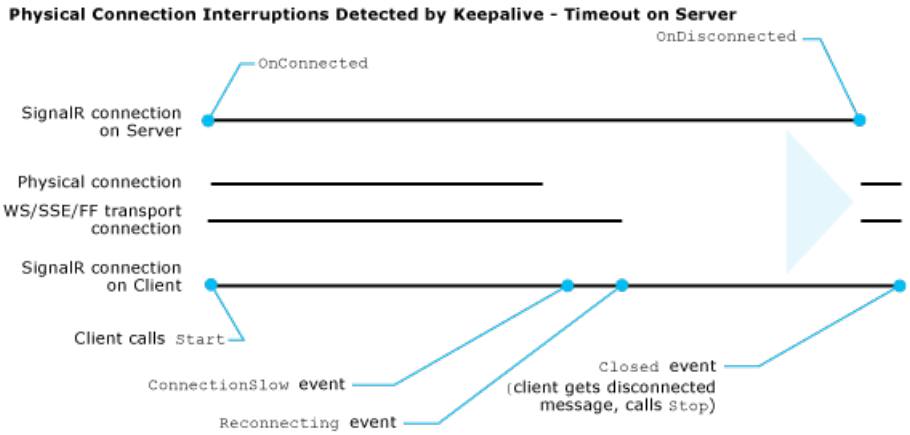
The following diagram illustrates the client and server events that are raised in a typical scenario when there are problems with the physical connection that aren't immediately recognized by the transport API. The diagram applies to the following circumstances:

- The transport is WebSockets, forever frame, or server-sent events.
- There are varying periods of interruption in the physical network connection.
- The transport API does not become aware of the interruptions, so SignalR relies on the keepalive functionality to detect them.



If the client goes into reconnecting mode but can't establish a transport connection within the disconnect timeout limit, the server terminates the SignalR connection. When that happens, the server executes the Hub's

`OnDisconnected` method and queues up a disconnect message to send to the client in case the client manages to connect later. If the client then does reconnect, it receives the disconnect command and calls the `Stop` method. In this scenario, `OnReconnected` is not executed when the client reconnects, and `OnDisconnected` is not executed when the client calls `Stop`. The following diagram illustrates this scenario.



The SignalR connection lifetime events that may be raised on the client are the following:

- `ConnectionSlow` client event.

  Raised when a preset proportion of the keepalive timeout period has passed since the last message or keepalive ping was received. The default keepalive timeout warning period is 2/3 of the keepalive timeout. The keepalive timeout is 20 seconds, so the warning occurs at about 13 seconds.

  By default, the server sends keepalive pings every 10 seconds, and the client checks for keepalive pings about every 2 seconds (one third of the difference between the keepalive timeout value and the keepalive timeout warning value).

  If the transport API becomes aware of a disconnection, SignalR might be informed of the disconnection before the keepalive timeout warning period passes. In that case, the `ConnectionSlow` event would not be raised, and SignalR would go directly to the `Reconnecting` event.

- `Reconnecting` client event.

  Raised when (a) the transport API detects that the connection is lost, or (b) the keepalive timeout period has passed since the last message or keepalive ping was received. The SignalR client code begins trying to reconnect. You can handle this event if you want your application to take some action when a transport connection is lost. The default keepalive timeout period is currently 20 seconds.

  If your client code tries to call a Hub method while SignalR is in reconnecting mode, SignalR will try to send the command. Most of the time, such attempts will fail, but in some circumstances they might succeed. For the server-sent events, forever frame, and long polling transports, SignalR uses two communication channels, one that the client uses to send messages and one that it uses to receive messages. The channel used for receiving is the permanently open one, and that's the one that is closed when the physical connection is interrupted. The channel used for sending remains available, so if physical connectivity is restored, a method call from client to server might be successful before the receive channel is re-established. The return value would not be received until SignalR re-opens the channel used for receiving.

- `Reconnected` client event.

  Raised when the transport connection is reestablished. The `OnReconnected` event handler in the Hub executes.

- `Closed` client event (`disconnected` event in JavaScript).

  Raised when the disconnect timeout period expires while the SignalR client code is trying to reconnect after

losing the transport connection. The default disconnect timeout is 30 seconds. (This event is also raised when the connection ends because the `Stop` method is called.)

Transport connection interruptions that are not detected by the transport API and don't delay the reception of keepalive pings from the server for longer than the keepalive timeout warning period might not cause any connection lifetime events to be raised.

Some network environments deliberately close idle connections, and another function of the keepalive packets is to help prevent this by letting these networks know that a SignalR connection is in use. In extreme cases the default frequency of keepalive pings might not be enough to prevent closed connections. In that case you can configure keepalive pings to be sent more often. For more information, see Timeout and keepalive settings later in this topic.

NOTE

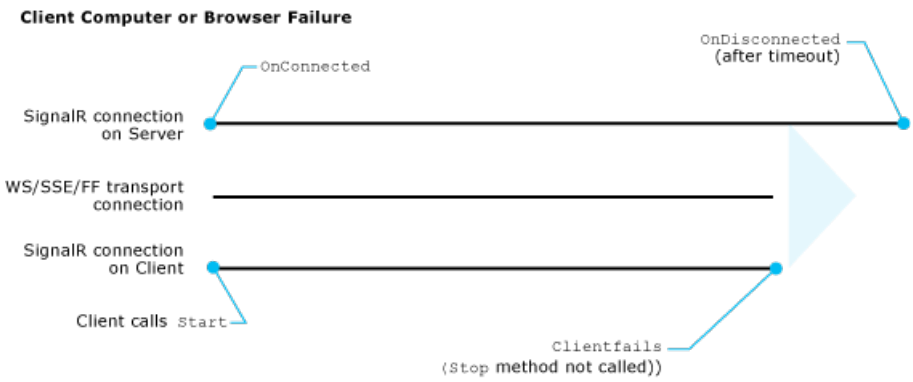
IMPORTANT

The sequence of events described here is not guaranteed. SignalR makes every attempt to raise connection lifetime events in a predictable manner according to this scheme, but there are many variations of network events and many ways in which underlying communications frameworks such as transport APIs handle them. For example, the `Reconnected` event might not be raised when the client reconnects, or the `OnConnected` handler on the server might run when the attempt to establish a connection is unsuccessful. This topic describes only the effects that would normally be produced by certain typical circumstances.

Client disconnection scenarios

In a browser client, the SignalR client code that maintains a SignalR connection runs in the JavaScript context of a web page. That's why the SignalR connection has to end when you navigate from one page to another, and that's why you have multiple connections with multiple connection IDs if you connect from multiple browser windows or tabs. When the user closes a browser window or tab, or navigates to a new page or refreshes the page, the SignalR connection immediately ends because SignalR client code handles that browser event for you and calls the `Stop` method. In these scenarios, or in any client platform when your application calls the `Stop` method, the `OnDisconnected` event handler executes immediately on the server and the client raises the `Closed` event (the event is named `disconnected` in JavaScript).

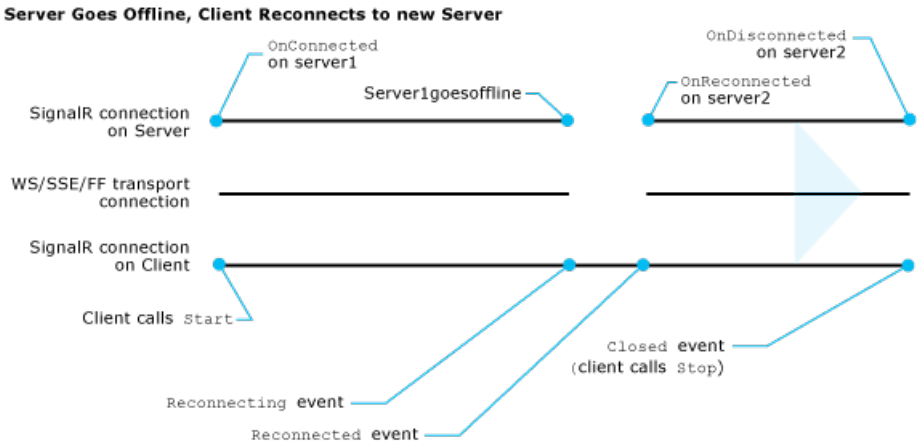
If a client application or the computer that it's running on crashes or goes to sleep (for example, when the user closes the laptop), the server is not informed about what happened. As far as the server knows, the loss of the client might be due to connectivity interruption and the client might be trying to reconnect. Therefore, in these scenarios the server waits to give the client a chance to reconnect, and `OnDisconnected` does not execute until the disconnect timeout period expires (about 30 seconds by default). The following diagram illustrates this scenario.



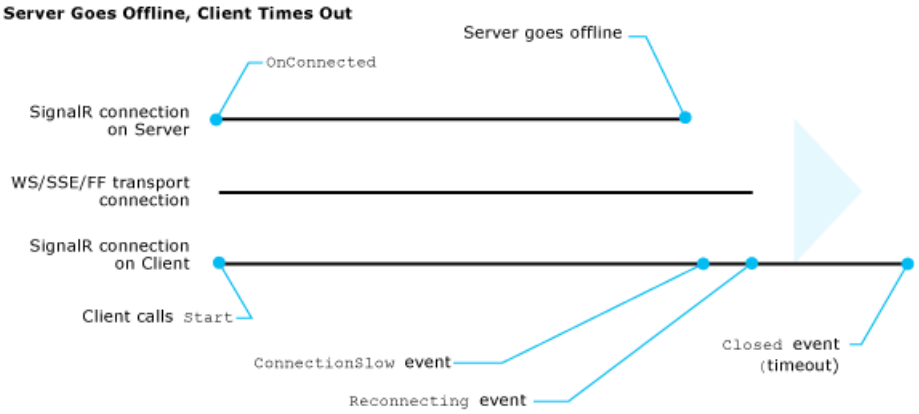
Server disconnection scenarios

When a server goes offline -- it reboots, fails, the app domain recycles, etc. -- the result might be similar to a lost

connection, or the transport API and SignalR might know immediately that the server is gone, and SignalR might begin trying to reconnect without raising the `ConnectionSlow` event. If the client goes into reconnecting mode, and if the server recovers or restarts or a new server is brought online before the disconnect timeout period expires, the client will reconnect to the restored or new server. In that case, the SignalR connection continues on the client and the `Reconnected` event is raised. On the first server, `OnDisconnected` is never executed, and on the new server, `OnReconnected` is executed although `OnConnected` was never executed for that client on that server before. (The effect is the same if the client reconnects to the same server after a reboot or app domain recycle, because when the server restarts it has no memory of prior connection activity.) The following diagram assumes that the transport API becomes aware of the lost connection immediately, so the `ConnectionSlow` event is not raised.



If a server does not become available within the disconnect timeout period, the SignalR connection ends. In this scenario, the `Closed` event (`disconnected` in JavaScript clients) is raised on the client but `OnDisconnected` is never called on the server. The following diagram assumes that the transport API does not become aware of the lost connection, so it is detected by SignalR keepalive functionality and the `ConnectionSlow` event is raised.



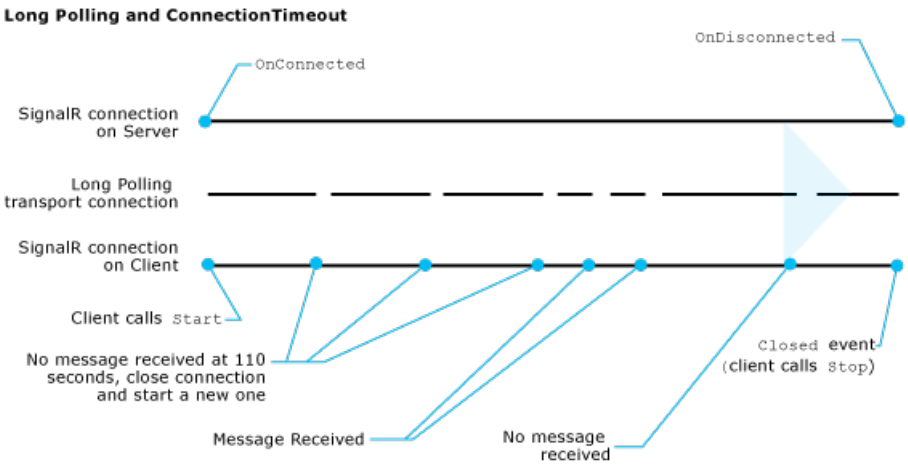
Timeout and keepalive settings

The default `ConnectionTimeout`, `DisconnectTimeout`, and `KeepAlive` values are appropriate for most scenarios but can be changed if your environment has special needs. For example, if your network environment closes connections that are idle for 5 seconds, you might have to decrease the keepalive value.

ConnectionTimeout

This setting represents the amount of time to leave a transport connection open and waiting for a response before closing it and opening a new connection. The default value is 110 seconds.

This setting applies only when keepalive functionality is disabled, which normally applies only to the long polling transport. The following diagram illustrates the effect of this setting on a long polling transport connection.



DisconnectTimeout

This setting represents the amount of time to wait after a transport connection is lost before raising the `Disconnected` event. The default value is 30 seconds. When you set `DisconnectTimeout`, `KeepAlive` is automatically set to 1/3 of the `DisconnectTimeout` value.

KeepAlive

This setting represents the amount of time to wait before sending a keepalive packet over an idle connection. The default value is 10 seconds. This value must not be more than 1/3 of the `DisconnectTimeout` value.

If you want to set both `DisconnectTimeout` and `KeepAlive`, set `KeepAlive` after `DisconnectTimeout`. Otherwise your `KeepAlive` setting will be overwritten when `DisconnectTimeout` automatically sets `KeepAlive` to 1/3 of the timeout value.

If you want to disable keepalive functionality, set `KeepAlive` to null. Keepalive functionality is automatically disabled for the long polling transport.

How to change timeout and keepalive settings

To change the default values for these settings, set them in `Application_Start` in your *Global.asax* file, as shown in the following example. The values shown in the sample code are the same as the default values.

```
protected void Application_Start(object sender, EventArgs e)
{
    // Make long polling connections wait a maximum of 110 seconds for a
    // response. When that time expires, trigger a timeout command and
    // make the client reconnect.
    GlobalHost.Configuration.ConnectionTimeout = TimeSpan.FromSeconds(110);

    // Wait a maximum of 30 seconds after a transport connection is lost
    // before raising the Disconnected event to terminate the SignalR connection.
    GlobalHost.Configuration.DisconnectTimeout = TimeSpan.FromSeconds(30);

    // For transports other than long polling, send a keepalive packet every
    // 10 seconds.
    // This value must be no more than 1/3 of the DisconnectTimeout value.
    GlobalHost.Configuration.KeepAlive = TimeSpan.FromSeconds(10);
}
```

How to notify the user about disconnections

In some applications you might want to display a message to the user when there are connectivity problems. You have several options for how and when to do this. The following code samples are for a JavaScript client using the generated proxy.

- Handle the `connectionSlow` event to display a message as soon as SignalR is aware of connection problems,

before it goes into reconnecting mode.

```
$.connection.hub.connectionSlow(function() {
    notifyUserOfConnectionProblem(); // Your function to notify user.
});
```

- Handle the `reconnecting` event to display a message when SignalR is aware of a disconnection and is going into reconnecting mode.

```
$.connection.hub.reconnecting(function() {
    notifyUserOfTryingToReconnect(); // Your function to notify user.
});
```

- Handle the `disconnected` event to display a message when an attempt to reconnect has timed out. In this scenario, the only way to re-establish a connection with the server again is to restart the SignalR connection by calling the `Start` method, which will create a new connection ID. The following code sample uses a flag to make sure that you issue the notification only after a reconnecting timeout, not after a normal end to the SignalR connection caused by calling the `Stop` method.

```
var tryingToReconnect = false;

$.connection.hub.reconnecting(function() {
    tryingToReconnect = true;
});

$.connection.hub.reconnected(function() {
    tryingToReconnect = false;
});

$.connection.hub.disconnected(function() {
    if(tryingToReconnect) {
        notifyUserOfDisconnect(); // Your function to notify user.
    }
});
```

How to continuously reconnect

In some applications you might want to automatically re-establish a connection after it has been lost and the attempt to reconnect has timed out. To do that, you can call the `Start` method from your `Closed` event handler (`disconnected` event handler on JavaScript clients). You might want to wait a period of time before calling `Start` in order to avoid doing this too frequently when the server or the physical connection are unavailable. The following code sample is for a JavaScript client using the generated proxy.

```
$.connection.hub.disconnected(function() {
   setTimeout(function() {
       $.connection.hub.start();
   }, 5000); // Restart connection after 5 seconds.
});
```

A potential problem to be aware of in mobile clients is that continuous reconnection attempts when the server or physical connection isn't available could cause unnecessary battery drain.

How to disconnect a client in server code

SignalR version 1.1.1 does not have a built-in server API for disconnecting clients. There are plans for adding this

functionality in the future. In the current SignalR release, the simplest way to disconnect a client from the server is to implement a disconnect method on the client and call that method from the server. The following code sample shows a disconnect method for a JavaScript client using the generated proxy.

```
var myHubProxy = $.connection.myHub
myHubProxy.client.stopClient = function() {
    $.connection.hub.stop();
};
```

WARNING

Security - Neither this method for disconnecting clients nor the proposed built-in API will address the scenario of hacked clients that are running malicious code, since the clients could reconnect or the hacked code might remove the `stopClient` method or change what it does. The appropriate place to implement stateful denial-of-service (DOS) protection is not in the framework or the server layer, but rather in front-end infrastructure.

Working with Groups in SignalR 1.x

2/20/2020 • 8 minutes to read • Edit Online

by Patrick Fletcher, Tom FitzMacken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This topic describes how to add users to groups and persist group membership information.

Overview

Groups in SignalR provide a method for broadcasting messages to specified subsets of connected clients. A group can have any number of clients, and a client can be a member of any number of groups. You don't have to explicitly create groups. In effect, a group is automatically created the first time you specify its name in a call to Groups.Add, and it is deleted when you remove the last connection from membership in it. For an introduction to using groups, see How to manage group membership from the Hub class in the Hubs API - Server Guide.

There is no API for getting a group membership list or a list of groups. SignalR sends messages to clients and groups based on a pub/sub model, and the server does not maintain lists of groups or group memberships. This helps maximize scalability, because whenever you add a node to a web farm, any state that SignalR maintains has to be propagated to the new node.

When you add a user to a group using the `Groups.Add` method, the user receives messages directed to that group for the duration of the current connection, but the user's membership in that group is not persisted beyond the current connection. If you want to permanently retain information about groups and group membership, you must store that data in a repository such as a database or Azure table storage. Then, each time a user connects to your application, you retrieve from the repository which groups the user belongs to, and manually add that user to those groups.

When reconnecting after a temporary disruption, the user automatically re-joins the previously-assigned groups. Automatically rejoining a group only applies when reconnecting, not when establishing a new connection. A digitally-signed token is passed from the client that contains the list of previously-assigned groups. If you want to verify whether the user belongs to the requested groups, you can override the default behavior.

This topic includes the following sections:

- Adding and removing users
- Calling members of a group
- Storing group membership in a database
- Storing group membership in Azure table storage
- Verifying group membership when reconnecting

Adding and removing users

To add or remove users from a group, you call the Add or Remove methods, and pass in the user's connection id and group's name as parameters. You do not need to manually remove a user from a group when the connection ends.

The following example shows the `Groups.Add` and `Groups.Remove` methods used in Hub methods.

```
public class ContosoChatHub : Hub
{
    public Task JoinRoom(string roomName)
    {
        return Groups.Add(Context.ConnectionId, roomName);
    }

    public Task LeaveRoom(string roomName)
    {
        return Groups.Remove(Context.ConnectionId, roomName);
    }
}
```

The `Groups.Add` and `Groups.Remove` methods execute asynchronously.

If you want to add a client to a group and immediately send a message to the client by using the group, you have to make sure that the Groups.Add method finishes first. The following code examples show how to do that, one by using code that works in .NET 4.5, and one by using code that works in .NET 4.

Asynchronous .NET 4.5 Example

```
public async Task JoinRoom(string roomName)
{
    await Groups.Add(Context.ConnectionId, roomName);
    Clients.Group(roomName).addChatMessage(Context.User.Identity.Name + " joined.");
}
```

Asynchronous .NET 4 Example

```
public void JoinRoom(string roomName)
{
    (Groups.Add(Context.ConnectionId, roomName) as Task).ContinueWith(antecedent =>
      Clients.Group(roomName).addChatMessage(Context.User.Identity.Name + " joined."));
}
```

In general, you should not include `await` when calling the `Groups.Remove` method because the connection id that you are trying to remove might no longer be available. In that case, `TaskCanceledException` is thrown after the request times out. If your application must ensure that the user has been removed from the group before sending a message to the group, you can add `await` before Groups.Remove, and then catch the `TaskCanceledException` exception that might be thrown.

Calling members of a group

You can send messages to all of the members of a group or only specified members of the group, as shown in the following examples.

- **All** connected clients in a specified group.

  ```
  Clients.Group(groupName).addChatMessage(name, message);
  ```

- All connected clients in a specified group **except the specified clients**, identified by connection ID.

  ```
  Clients.Group(groupName, connectionId1, connectionId2).addChatMessage(name, message);
  ```

- All connected clients in a specified group **except the calling client**.

```
Clients.OthersInGroup(groupName).addChatMessage(name, message);
```

Storing group membership in a database

The following examples show how to retain group and user information in a database. You can use any data access technology; however, the example below shows how to define models using Entity Framework. These entity models correspond to database tables and fields. Your data structure could vary considerably depending on the requirements of your application. This example includes a class named `ConversationRoom` which would be unique to an application that enables users to join conversations about different subjects, such as sports or gardening. This example also includes a class for the connections. The connection class is not absolutely required for tracking group membership but is frequently part of robust solution to tracking users.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity;

namespace GroupsExample
{
    public class UserContext : DbContext
    {
        public DbSet<User> Users { get; set; }
        public DbSet<Connection> Connections { get; set; }
        public DbSet<ConversationRoom> Rooms { get; set; }
    }

    public class User
    {
        [Key]
        public string UserName { get; set; }
        public ICollection<Connection> Connections { get; set; }
        public virtual ICollection<ConversationRoom> Rooms { get; set; }
    }

    public class Connection
    {
        public string ConnectionID { get; set; }
        public string UserAgent { get; set; }
        public bool Connected { get; set; }
    }

    public class ConversationRoom
    {
        [Key]
        public string RoomName { get; set; }
        public virtual ICollection<User> Users { get; set; }
    }
}
```

Then, in the hub, you can retrieve the group and user information from the database and manually add the user to the appropriate groups. The example does not include code for tracking the user connections. In this example, the `await` keyword is not applied before `Groups.Add` because a message is not immediately sent to members of the group. If you want to send a message to all members of the group immediately after adding the new member, you would want to apply the `await` keyword to make sure the asynchronous operation has completed.

```
using Microsoft.AspNet.SignalR;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
```

```
using System.Threading.Tasks;

namespace GroupsExample
{
    [Authorize]
    public class ChatHub : Hub
    {
        public override Task OnConnected()
        {
            using (var db = new UserContext())
            {
                // Retrieve user.
                var user = db.Users
                    .Include(u => u.Rooms)
                    .SingleOrDefault(u => u.UserName == Context.User.Identity.Name);

                // If user does not exist in database, must add.
                if (user == null)
                {
                    user = new User()
                    {
                        UserName = Context.User.Identity.Name
                    };
                    db.Users.Add(user);
                    db.SaveChanges();
                }
                else
                {
                    // Add to each assigned group.
                    foreach (var item in user.Rooms)
                    {
                        Groups.Add(Context.ConnectionId, item.RoomName);
                    }
                }
            }
            return base.OnConnected();
        }

        public void AddToRoom(string roomName)
        {
            using (var db = new UserContext())
            {
                // Retrieve room.
                var room = db.Rooms.Find(roomName);

                if (room != null)
                {
                    var user = new User() { UserName = Context.User.Identity.Name};
                    db.Users.Attach(user);

                    room.Users.Add(user);
                    db.SaveChanges();
                    Groups.Add(Context.ConnectionId, roomName);
                }
            }
        }

        public void RemoveFromRoom(string roomName)
        {
            using (var db = new UserContext())
            {
                // Retrieve room.
                var room = db.Rooms.Find(roomName);
                if (room != null)
                {
                    var user = new User() { UserName = Context.User.Identity.Name };
                    db.Users.Attach(user);

                    room.Users.Remove(user);
```

```
                    db.SaveChanges();

                    Groups.Remove(Context.ConnectionId, roomName);
                }
            }
        }
    }
}
```

Storing group membership in Azure table storage

Using Azure table storage to store group and user information is similar to using a database. The following example shows a table entity that stores the user name and group name.

```
using Microsoft.WindowsAzure.Storage.Table;
using System;

namespace GroupsExample
{
    public class UserGroupEntity : TableEntity
    {
        public UserGroupEntity() { }

        public UserGroupEntity(string userName, string groupName)
        {
            this.PartitionKey = userName;
            this.RowKey = groupName;
        }
    }
}
```

In the hub, you retrieve the assigned groups when the user connects.

```
using Microsoft.AspNet.SignalR;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.WindowsAzure.Storage.Table;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure;

namespace GroupsExample
{
    [Authorize]
    public class ChatHub : Hub
    {
        public override Task OnConnected()
        {
            string userName = Context.User.Identity.Name;

            var table = GetRoomTable();
            table.CreateIfNotExists();
            var query = new TableQuery<UserGroupEntity>()
                .Where(TableQuery.GenerateFilterCondition(
                "PartitionKey", QueryComparisons.Equal, userName));

            foreach (var entity in table.ExecuteQuery(query))
            {
                Groups.Add(Context.ConnectionId, entity.RowKey);
            }

            return base.OnConnected();
        }
```

```
        public Task AddToRoom(string roomName)
        {
            string userName = Context.User.Identity.Name;

            var table = GetRoomTable();

            var insertOperation = TableOperation.InsertOrReplace(
                new UserGroupEntity(userName, roomName));
            table.Execute(insertOperation);

            return Groups.Add(Context.ConnectionId, roomName);
        }

        public Task RemoveFromRoom(string roomName)
        {
            string userName = Context.User.Identity.Name;

            var table = GetRoomTable();

            var retrieveOperation = TableOperation.Retrieve<UserGroupEntity>(
                userName, roomName);
            var retrievedResult = table.Execute(retrieveOperation);

            var deleteEntity = (UserGroupEntity)retrievedResult.Result;

            if (deleteEntity != null)
            {
                var deleteOperation = TableOperation.Delete(deleteEntity);
                table.Execute(deleteOperation);
            }

            return Groups.Remove(Context.ConnectionId, roomName);
        }

       private CloudTable GetRoomTable()
        {
            var storageAccount =
                CloudStorageAccount.Parse(
                CloudConfigurationManager.GetSetting("StorageConnectionString"));
            var tableClient = storageAccount.CreateCloudTableClient();
            return tableClient.GetTableReference("room");
        }
    }
}
```

Verifying group membership when reconnecting

By default, SignalR automatically re-assigns a user to the appropriate groups when reconnecting from a temporary disruption, such as when a connection is dropped and re-established before the connection times out. The user's group information is passed in a token when reconnecting, and that token is verified on the server. For information about the verification process for rejoining users to groups, see Rejoining groups when reconnecting.

In general, you should use the default behavior of automatically rejoining groups on reconnect. SignalR groups are not intended as a security mechanism for restricting access to sensitive data. However, if your application must double-check a user's group membership when reconnecting, you can override the default behavior. Changing the default behavior can add a burden to your database because a user's group membership must be retrieved for each reconnection rather than just when the user connects.

If you must verify group membership on reconnect, create a new hub pipeline module that returns a list of assigned groups, as shown below.

```
using Microsoft.AspNet.SignalR;
using Microsoft.AspNet.SignalR.Hubs;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;

namespace GroupsExample
{
    public class RejoingGroupPipelineModule : HubPipelineModule
    {
        public override Func<HubDescriptor, IRequest, IList<string>, IList<string>>
            BuildRejoiningGroups(Func<HubDescriptor, IRequest, IList<string>, IList<string>>
            rejoiningGroups)
        {
            rejoiningGroups = (hb, r, l) =>
            {
                List<string> assignedRooms = new List<string>();
                using (var db = new UserContext())
                {
                    var user = db.Users.Include(u => u.Rooms)
                        .Single(u => u.UserName == r.User.Identity.Name);
                    foreach (var item in user.Rooms)
                    {
                        assignedRooms.Add(item.RoomName);
                    }
                }
                return assignedRooms;
            };

            return rejoiningGroups;
        }
    }
}
```

Then, add that module to the hub pipeline, as highlighted below.

```
public class Global : HttpApplication
{
    void Application_Start(object sender, EventArgs e)
    {
        // Code that runs on application startup
        BundleConfig.RegisterBundles(BundleTable.Bundles);
        AuthConfig.RegisterOpenAuth();
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        RouteTable.Routes.MapHubs();
        GlobalHost.HubPipeline.AddModule(new RejoingGroupPipelineModule());
    }
}
```

Mapping SignalR Users to Connections in SignalR 1.x

2/20/2020 • 7 minutes to read • Edit Online

by Patrick Fletcher, Tom FitzMacken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

> This topic shows how to retain information about users and their connections.

Introduction

Each client connecting to a hub passes a unique connection id. You can retrieve this value in the `Context.ConnectionId` property of the hub context. If your application needs to map a user to the connection id and persist that mapping, you can use one of the following:

- In-memory storage, such as a dictionary
- SignalR group for each user
- Permanent, external storage, such as a database table or Azure table storage

Each of these implementations is shown in this topic. You use the `OnConnected`, `OnDisconnected`, and `OnReconnected` methods of the `Hub` class to track the user connection status.

The best approach for your application depends on:

- The number of web servers hosting your application.
- Whether you need to get a list of the currently connected users.
- Whether you need to persist group and user information when the application or server restarts.
- Whether the latency of calling an external server is an issue.

The following table shows which approach works for these considerations.

	MORE THAN ONE SERVER	GET LIST OF CURRENTLY CONNECTED USERS	PERSIST INFORMATION AFTER RESTARTS	OPTIMAL PERFORMANCE
In-memory		✓		✓
Single-user groups	✓			✓
Permanent, external	✓	✓	✓	

In-memory storage

The following examples show how to retain connection and user information in a dictionary that is stored in memory. The dictionary uses a `HashSet` to store the connection id. At any time a user could have more than one connection to the SignalR application. For example, a user who is connected through multiple devices or more than one browser tab would have more than one connection id.

If the application shuts down, all of the information is lost, but it will be re-populated as the users re-establish their connections. In-memory storage does not work if your environment includes more than one web server because each server would have a separate collection of connections.

The first example shows a class that manages the mapping of users to connections. The key for the HashSet will be the user's name.

```
using System.Collections.Generic;
using System.Linq;

namespace BasicChat
{
    public class ConnectionMapping<T>
    {
        private readonly Dictionary<T, HashSet<string>> _connections =
            new Dictionary<T, HashSet<string>>();

        public int Count
        {
            get
            {
                return _connections.Count;
            }
        }

        public void Add(T key, string connectionId)
        {
            lock (_connections)
            {
                HashSet<string> connections;
                if (!_connections.TryGetValue(key, out connections))
                {
                    connections = new HashSet<string>();
                    _connections.Add(key, connections);
                }

                lock (connections)
                {
                    connections.Add(connectionId);
                }
            }
        }

        public IEnumerable<string> GetConnections(T key)
        {
            HashSet<string> connections;
            if (_connections.TryGetValue(key, out connections))
            {
                return connections;
            }

            return Enumerable.Empty<string>();
        }

        public void Remove(T key, string connectionId)
        {
            lock (_connections)
            {
                HashSet<string> connections;
                if (!_connections.TryGetValue(key, out connections))
                {
                    return;
                }

                lock (connections)
                {
                    connections.Remove(connectionId);
```

```
                    if (connections.Count == 0)
                    {
                        _connections.Remove(key);
                    }
                }
            }
        }
    }
}
```

The next example shows how to use the connection mapping class from a hub. The instance of the class is stored in a variable name `_connections`.

```
using System.Threading.Tasks;
using Microsoft.AspNet.SignalR;

namespace BasicChat
{
    [Authorize]
    public class ChatHub : Hub
    {
        private readonly static ConnectionMapping<string> _connections =
            new ConnectionMapping<string>();

        public void SendChatMessage(string who, string message)
        {
            string name = Context.User.Identity.Name;

            foreach (var connectionId in _connections.GetConnections(who))
            {
                Clients.Client(connectionId).addChatMessage(name + ": " + message);
            }
        }

        public override Task OnConnected()
        {
            string name = Context.User.Identity.Name;

            _connections.Add(name, Context.ConnectionId);

            return base.OnConnected();
        }

        public override Task OnDisconnected()
        {
            string name = Context.User.Identity.Name;

            _connections.Remove(name, Context.ConnectionId);

            return base.OnDisconnected();
        }

        public override Task OnReconnected()
        {
            string name = Context.User.Identity.Name;

            if (!_connections.GetConnections(name).Contains(Context.ConnectionId))
            {
                _connections.Add(name, Context.ConnectionId);
            }

            return base.OnReconnected();
        }
    }
}
```

Single-user groups

You can create a group for each user, and then send a message to that group when you want to reach only that user. The name of each group is the name of the user. If a user has more than one connection, each connection id is added to the user's group.

You should not manually remove the user from the group when the user disconnects. This action is automatically performed by the SignalR framework.

The following example shows how to implement single-user groups.

```
using Microsoft.AspNet.SignalR;
using System;
using System.Threading.Tasks;

namespace BasicChat
{
    [Authorize]
    public class ChatHub : Hub
    {
        public void SendChatMessage(string who, string message)
        {
            string name = Context.User.Identity.Name;

            Clients.Group(who).addChatMessage(name + ": " + message);
        }

        public override Task OnConnected()
        {
            string name = Context.User.Identity.Name;

            Groups.Add(Context.ConnectionId, name);

            return base.OnConnected();
        }
    }
}
```

Permanent, external storage

This topic shows how to use either a database or Azure table storage for storing connection information. This approach works when you have multiple web servers because each web server can interact with the same data repository. If your web servers stop working or the application restarts, the `OnDisconnected` method is not called. Therefore, it is possible that your data repository will have records for connection ids that are no longer valid. To clean up these orphaned records, you may wish to invalidate any connection that was created outside of a timeframe that is relevant to your application. The examples in this section include a value for tracking when the connection was created, but do not show how to clean up old records because you may want to do that as background process.

Database

The following examples show how to retain connection and user information in a database. You can use any data access technology; however, the example below shows how to define models using Entity Framework. These entity models correspond to database tables and fields. Your data structure could vary considerably depending on the requirements of your application.

The first example shows how to define a user entity that can be associated with many connection entities.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity;

namespace MapUsersSample
{
    public class UserContext : DbContext
    {
        public DbSet<User> Users { get; set; }
        public DbSet<Connection> Connections { get; set; }
    }

    public class User
    {
        [Key]
        public string UserName { get; set; }
        public ICollection<Connection> Connections { get; set; }
    }

    public class Connection
    {
        public string ConnectionID { get; set; }
        public string UserAgent { get; set; }
        public bool Connected { get; set; }
    }
}
```

Then, from the hub, you can track the state of each connection with the code shown below.

```
using System;
using System.Data.Entity;
using System.Linq;
using System.Threading.Tasks;
using System.Collections.Concurrent;
using Microsoft.AspNet.SignalR;

namespace MapUsersSample
{
    [Authorize]
    public class ChatHub : Hub
    {
        public void SendChatMessage(string who, string message)
        {
            var name = Context.User.Identity.Name;
            using (var db = new UserContext())
            {
                var user = db.Users.Find(who);
                if (user == null)
                {
                    Clients.Caller.showErrorMessage("Could not find that user.");
                }
                else
                {
                    db.Entry(user)
                        .Collection(u => u.Connections)
                        .Query()
                        .Where(c => c.Connected == true)
                        .Load();

                    if (user.Connections == null)
                    {
                        Clients.Caller.showErrorMessage("The user is no longer connected.");
                    }
                    else
                    {
```

```
                        foreach (var connection in user.Connections)
                        {
                            Clients.Client(connection.ConnectionID)
                                .addChatMessage(name + ": " + message);
                        }
                    }
                }
            }
        }

        public override Task OnConnected()
        {
            var name = Context.User.Identity.Name;
            using (var db = new UserContext())
            {
                var user = db.Users
                    .Include(u => u.Connections)
                    .SingleOrDefault(u => u.UserName == name);

                if (user == null)
                {
                    user = new User
                    {
                        UserName = name,
                        Connections = new List<Connection>()
                    };
                    db.Users.Add(user);
                }

                user.Connections.Add(new Connection
                {
                    ConnectionID = Context.ConnectionId,
                    UserAgent = Context.Request.Headers["User-Agent"],
                    Connected = true
                });
                db.SaveChanges();
            }
            return base.OnConnected();
        }

        public override Task OnDisconnected()
        {
            using (var db = new UserContext())
            {
                var connection = db.Connections.Find(Context.ConnectionId);
                connection.Connected = false;
                db.SaveChanges();
            }
            return base.OnDisconnected();
        }
    }
}
```

Azure table storage

The following Azure table storage example is similar to the database example. It does not include all of the information that you would need to get started with Azure Table Storage Service. For information, see How to use Table storage from .NET.

The following example shows a table entity for storing connection information. It partitions the data by user name, and identifies each entity by the connection id, so a user can have multiple connections at any time.

```
using Microsoft.WindowsAzure.Storage.Table;
using System;

namespace MapUsersSample
{
    public class ConnectionEntity : TableEntity
    {
        public ConnectionEntity() { }

        public ConnectionEntity(string userName, string connectionID)
        {
            this.PartitionKey = userName;
            this.RowKey = connectionID;
        }
    }
}
```

In the hub, you track the status of each user's connection.

```
using Microsoft.AspNet.SignalR;
using Microsoft.WindowsAzure;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Table;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace MapUsersSample
{
    public class ChatHub : Hub
    {
        public void SendChatMessage(string who, string message)
        {
            var name = Context.User.Identity.Name;

            var table = GetConnectionTable();

            var query = new TableQuery<ConnectionEntity>()
                .Where(TableQuery.GenerateFilterCondition(
                "PartitionKey",
                QueryComparisons.Equal,
                who));

            var queryResult = table.ExecuteQuery(query).ToList();
            if (queryResult.Count == 0)
            {
                Clients.Caller.showErrorMessage("The user is no longer connected.");
            }
            else
            {
                foreach (var entity in queryResult)
                {
                    Clients.Client(entity.RowKey).addChatMessage(name + ": " + message);
                }
            }
        }

        public override Task OnConnected()
        {
            var name = Context.User.Identity.Name;
            var table = GetConnectionTable();
            table.CreateIfNotExists();

            var entity = new ConnectionEntity(
                name.ToLower(),
                Context.ConnectionId);
```

```
            var insertOperation = TableOperation.InsertOrReplace(entity);
            table.Execute(insertOperation);

            return base.OnConnected();
        }

        public override Task OnDisconnected()
        {
            var name = Context.User.Identity.Name;
            var table = GetConnectionTable();

            var deleteOperation = TableOperation.Delete(
                new ConnectionEntity(name, Context.ConnectionId) { ETag = "*" });
            table.Execute(deleteOperation);

            return base.OnDisconnected();
        }

        private CloudTable GetConnectionTable()
        {
            var storageAccount =
                CloudStorageAccount.Parse(
                CloudConfigurationManager.GetSetting("StorageConnectionString"));
            var tableClient = storageAccount.CreateCloudTableClient();
            return tableClient.GetTableReference("connection");
        }
    }
}
```

Introduction to SignalR Security (SignalR 1.x)

2/20/2020 • 10 minutes to read • Edit Online

by Patrick Fletcher, Tom FitzMacken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

> This article describes the security issues you must consider when developing a SignalR application.

Overview

This document contains the following sections:

- SignalR Security Concepts
  - Authentication and authorization
  - Connection token
  - Rejoining groups when reconnecting
- How SignalR prevents Cross-Site Request Forgery
- SignalR Security Recommendations
  - Secure Socket Layers (SSL) protocol
  - Do not use groups as a security mechanism
  - Safely handling input from clients
  - Reconciling a change in user status with an active connection
  - Automatically generated JavaScript proxy files
  - Exceptions

SignalR Security Concepts

Authentication and authorization

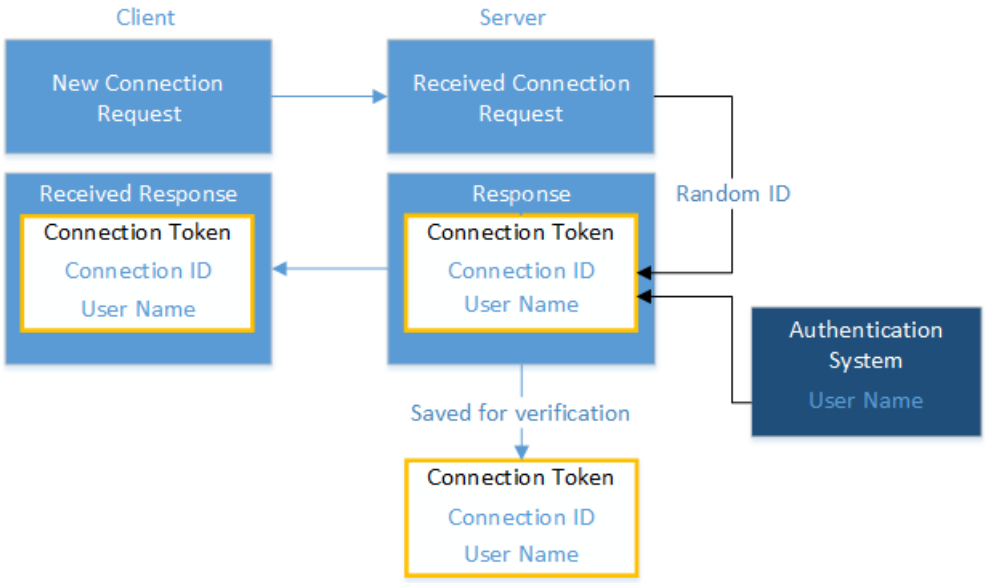
SignalR is designed to be integrated into the existing authentication structure for an application. It does not provide any features for authenticating users. Instead, you authenticate users as you would normally in your application, and then work with the results of the authentication in your SignalR code. For example, you might authenticate your users with ASP.NET forms authentication, and then in your hub, enforce which users or roles are authorized to call a method. In your hub, you can also pass authentication information, such as user name or whether a user belongs to a role, to the client.

SignalR provides the Authorize attribute to specify which users have access to a hub or method. You apply the Authorize attribute to either a hub or particular methods in a hub. Without the Authorize attribute, all public methods on the hub are available to a client that is connected to the hub. For more information about hubs, see Authentication and Authorization for SignalR Hubs.

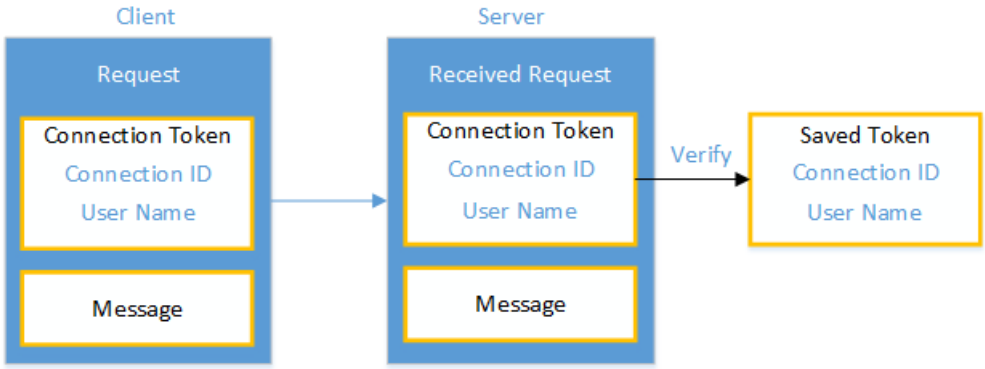
The `Authorize` attribute is used only with hubs. To enforce authorization rules when using a `PersistentConnection` you must override the `AuthorizeRequest` method. For more information about persistent connections, see Authentication and Authorization for SignalR Persistent Connections.

Connection token

SignalR mitigates the risk of executing malicious commands by validating the identity of the sender. A connection token, containing the connection id and username for authenticated users, is passed between the client and server for each request. The connection id is a unique identifier that is randomly generated by the server when a new connection is created, and is persisted for the duration of the connection. The username is provided by the authentication mechanism for the web application. The connection token is protected with encryption and a digital signature.



For each request, the server validates the contents of the token to ensure that the request is coming from the specified user. The username must correspond to the connection id. By validating both the connection id and the username, SignalR prevents a malicious user from easily impersonating another user. If the server cannot validate the connection token, the request fails.



Because the connection id is part of the verification process, you should not reveal one user's connection id to other users or store the value on the client, such as in a cookie.

Rejoining groups when reconnecting

By default, the SignalR application will automatically re-assign a user to the appropriate groups when reconnecting from a temporary disruption, such as when a connection is dropped and re-established before the connection times out. When reconnecting, the client passes a group token that includes the connection id and the assigned groups. The group token is digitally signed and encrypted. The client retains the same connection id after a reconnection; therefore, the connection id passed from the reconnected client must match the previous connection id used by the client. This verification prevents a malicious user from passing requests to join unauthorized groups when reconnecting.

However, it is important to note, that the group token does not expire. If a user belonged to a group in the past, but was banned from that group, that user may be able to mimic a group token that includes the prohibited group. If

you need to securely manage which users belong to which groups, you need to store that data on the server, such as in a database. Then, add logic to your application that verifies on the server whether a user belongs to a group. For an example of verifying group membership, see Working with groups.

Automatically rejoining groups only applies when a connection is reconnected after a temporary disruption. If a user disconnects by navigating away from the application or the application restarts, your application must handle how to add that user to the correct groups. For more information, see Working with groups.

How SignalR prevents Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) is an attack where a malicious site sends a request to a vulnerable site where the user is currently logged in. SignalR prevents CSRF by making it extremely unlikely for a malicious site to create a valid request for your SignalR application.

Description of CSRF attack

Here is an example of a CSRF attack:

1. A user logs into `www.example.com`, using forms authentication.
2. The server authenticates the user. The response from the server includes an authentication cookie.
3. Without logging out, the user visits a malicious web site. This malicious site contains the following HTML form:

   ```
   <h1>You Are a Winner!</h1>
   <form action="http://example.com/api/account" method="post">
       <input type="hidden" name="Transaction" value="withdraw" />
       <input type="hidden" name="Amount" value="1000000" />
       <input type="submit" value="Click Me"/>
   </form>
   ```

   Notice that the form action posts to the vulnerable site, not to the malicious site. This is the "cross-site" part of CSRF.
4. The user clicks the submit button. The browser includes the authentication cookie with the request.
5. The request runs on the example.com server with the user's authentication context, and can do anything that an authenticated user is allowed to do.

Although this example requires the user to click the form button, the malicious page could just as easily run a script that sends an AJAX request to your SignalR application. Moreover, using SSL does not prevent a CSRF attack, because the malicious site can send an "https://" request.

Typically, CSRF attacks are possible against web sites that use cookies for authentication, because browsers send all relevant cookies to the destination web site. However, CSRF attacks are not limited to exploiting cookies. For example, Basic and Digest authentication are also vulnerable. After a user logs in with Basic or Digest authentication, the browser automatically sends the credentials until the session ends.

CSRF mitigations taken by SignalR

SignalR takes the following steps to prevent a malicious site from creating valid requests to your SignalR application. These steps are taken by default and do not require any action in your code.

- **Disable cross domain requests**
  By default, cross domain requests are disabled in a SignalR application to prevent users from calling a SignalR endpoint from an external domain. Any request that comes from an external domain is automatically considered invalid and is blocked. It is recommended that you keep this default behavior; otherwise, a malicious site could trick users into sending commands to your site. If you need to use cross domain requests,

see How to establish a cross-domain connection .

- **Pass connection token in query string, not cookie**
  SignalR passes the connection token as a query string value, instead of as a cookie. By not storing the connection token as a cookie, the connection token is not inadvertently forwarded by the browser when malicious code is encountered. Also, the connection token is not persisted beyond the current connection. Therefore, a malicious user cannot make a request under another user's authentication credentials.
- **Verify connection token**
  As described in the Connection token section, the server knows which connection id is associated with each authenticated user. The server does not process any request from a connection id that does not match the user name. It is unlikely a malicious user could guess a valid request because the malicious user would have to know the user name and the current randomly-generated connection id. That connection id becomes invalid as soon as the connection is ended. Anonymous users should not have access to any sensitive information.

SignalR Security Recommendations

Secure Socket Layers (SSL) protocol

The SSL protocol uses encryption to secure the transport of data between a client and server. If your SignalR application transmits sensitive information between the client and server, use SSL for the transport. For more information about setting up SSL, see How to set up SSL on IIS 7.

Do not use groups as a security mechanism

Groups are a convenient way of collecting related users, but they are not a secure mechanism for limiting access to sensitive information. This is especially true when users can automatically rejoin groups during a reconnect. Instead, consider adding privileged users to a role and limiting access to a hub method to only members of that role. For an example of restricting access based on a role, see Authentication and Authorization for SignalR Hubs. For an example of checking user access to groups when reconnecting, see Working with groups.

Safely handling input from clients

All input from clients that is intended for broadcast to other clients must be encoded to ensure that a malicious user does not send script to other users. It is best to encode messages on the receiving clients rather than the server, because your SignalR application may have many different types of clients. Therefore, HTML-encoding works for a web client, but not for other types of clients. For example, a web client method to display a chat message would safely handle the user name and message by calling the `html()` function.

```
chat.client.addMessageToPage = function (name, message) {
    // Html encode display name and message.
    var encodedName = $('<div />').text(name).html();
    var encodedMsg = $('<div />').text(message).html();
    // Add the message to the page.
    $('#discussion').append('<li><strong>' + encodedName
        + '</strong>:  ' + encodedMsg + '</li>');
};
```

Reconciling a change in user status with an active connection

If a user's authentication status changes while an active connection exists, the user will receive an error that states, "The user identity cannot change during an active SignalR connection." In that case, your application should re-connect to the server to make sure the connection id and username are coordinated. For example, if your application allows the user to log out while an active connection exists, the username for the connection will no longer match the name that is passed in for the next request. You will want to stop the connection before the user logs out, and then restart it.

However, it is important to note that most applications will not need to manually stop and start the connection. If your application redirects users to a separate page after logging out, such as the default behavior in a Web Forms application or MVC application, or refreshes the current page after logging out, the active connection is

automatically disconnected and does not require any additional action.

The following example shows how to stop and start a connection when the user status has changed.

```
<script type="text/javascript">
    $(function () {
        var chat = $.connection.sampleHub;
        $.connection.hub.start().done(function () {
            $('#logoutbutton').click(function () {
                chat.connection.stop();
                $.ajax({
                    url: "Services/SampleWebService.svc/LogOut",
                    type: "POST"
                }).done(function () {
                    chat.connection.start();
                });
            });
        });
    });
</script>
```

Or, the user's authentication status may change if your site uses sliding expiration with Forms Authentication, and there is no activity to keep the authentication cookie valid. In that case, the user will be logged out and the user name will no longer match the user name in the connection token. You can fix this problem by adding some script that periodically requests a resource on the web server to keep the authentication cookie valid. The following example shows how to request a resource every 30 minutes.

```
$(function () {
    setInterval(function() {
        $.ajax({
            url: "Ping.aspx",
            cache: false
        });
    }, 1800000);
});
```

Automatically generated JavaScript proxy files

If you do not want to include all of the hubs and methods in the JavaScript proxy file for each user, you can disable the automatic generation of the file. You might choose this option if you have multiple hubs and methods, but do not want every user to be aware of all of the methods. You disable automatic generation by setting `EnableJavaScriptProxies` to `false`.

```
var hubConfiguration = new HubConfiguration();
hubConfiguration.EnableJavaScriptProxies = false;
RouteTable.Routes.MapHubs("/signalr", hubConfiguration);
```

For more information about the JavaScript proxy files, see The generated proxy and what it does for you.

Exceptions

You should avoid passing exception objects to clients because the objects may expose sensitive information to the clients. Instead, call a method on the client that displays the relevant error message.

```
public Task SampleMethod()
{
    try
    {
        // code that can throw an exception
    }
    catch(Exception e)
    {
        // add code to log exception and take remedial steps

        return Clients.Caller.DisplayError("Sorry, the request could not be processed.");
    }
}
```

Authentication and Authorization for SignalR Hubs (SignalR 1.x)

2/20/2020 • 7 minutes to read • Edit Online

by Patrick Fletcher, Tom FitzMacken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This topic describes how to restrict which users or roles can access hub methods.

Overview

This topic contains the following sections:

- Authorize attribute
- Require authentication for all hubs
- Customized authorization
- Pass authentication information to clients
- Authentication options for .NET clients
  - Cookie with Forms Authentication
  - Windows Authentication
  - Connection header
  - Certificate

Authorize attribute

SignalR provides the Authorize attribute to specify which users or roles have access to a hub or method. This attribute is located in the `Microsoft.AspNet.SignalR` namespace. You apply the `Authorize` attribute to either a hub or particular methods in a hub. When you apply the `Authorize` attribute to a hub class, the specified authorization requirement is applied to all of the methods in the hub. The different types of authorization requirements that you can apply are shown below. Without the `Authorize` attribute, all public methods on the hub are available to a client that is connected to the hub.

If you have defined a role named "Admin" in your web application, you could specify that only users in that role can access a hub with the following code.

```
[Authorize(Roles = "Admin")]
public class AdminAuthHub : Hub
{
}
```

Or, you can specify that a hub contains one method that is available to all users, and a second method that is only available to authenticated users, as shown below.

```
public class SampleHub : Hub
{
    public void UnrestrictedSend(string message){ . . . }

    [Authorize]
    public void AuthenticatedSend(string message){ . . . }
}
```

The following examples address different authorization scenarios:

- `[Authorize]` – only authenticated users
- `[Authorize(Roles = "Admin,Manager")]` – only authenticated users in the specified roles
- `[Authorize(Users = "user1,user2")]` – only authenticated users with the specified user names
- `[Authorize(RequireOutgoing=false)]` – only authenticated users can invoke the hub, but calls from the server back to clients are not limited by authorization, such as, when only certain users can send a message but all others can receive the message. The RequireOutgoing property can only be applied to the entire hub, not on individuals methods within the hub. When RequireOutgoing is not set to false, only users that meet the authorization requirement are called from the server.

Require authentication for all hubs

You can require authentication for all hubs and hub methods in your application by calling the RequireAuthentication method when the application starts. You might use this method when you have multiple hubs and want to enforce an authentication requirement for all of them. With this method, you cannot specify role, user, or outgoing authorization. You can only specify that access to the hub methods is restricted to authenticated users. However, you can still apply the Authorize attribute to hubs or methods to specify additional requirements. Any requirement you specify in attributes is applied in addition to the basic requirement of authentication.

The following example shows a Global.asax file which restricts all hub methods to authenticated users.

```
public class Global : HttpApplication
{
    void Application_Start(object sender, EventArgs e)
    {
        RouteTable.Routes.MapHubs();
        GlobalHost.HubPipeline.RequireAuthentication();
    }
}
```

If you call the `RequireAuthentication()` method after a SignalR request has been processed, SignalR will throw a `InvalidOperationException` exception. This exception is thrown because you cannot add a module to the HubPipeline after the pipeline has been invoked. The previous example shows calling the `RequireAuthentication` method in the `Application_Start` method which is executed one time prior to handling the first request.

Customized authorization

If you need to customize how authorization is determined, you can create a class that derives from `AuthorizeAttribute` and override the UserAuthorized method. This method is called for each request to determine whether the user is authorized to complete the request. In the overridden method, you provide the necessary logic for your authorization scenario. The following example shows how to enforce authorization through claims-based identity.

```
[AttributeUsage(AttributeTargets.Class, Inherited = false, AllowMultiple = false)]
public class AuthorizeClaimsAttribute : AuthorizeAttribute
{
    protected override bool UserAuthorized(System.Security.Principal.IPrincipal user)
    {
        if (user == null)
        {
            throw new ArgumentNullException("user");
        }

        var principal = (ClaimsPrincipal)user;

        if (principal != null)
        {
            Claim authenticated = principal.FindFirst(ClaimTypes.Authentication);
            return authenticated.Value == "true" ? true : false;
        }
        else
        {
            return false;
        }
    }
}
```

Pass authentication information to clients

You may need to use authentication information in the code that runs on the client. You pass the required information when calling the methods on the client. For example, a chat application method could pass as a parameter the user name of the person posting a message, as shown below.

```
public Task SendChatMessage(string message)
{
    string name;
    var user = Context.User;

    if (user.Identity.IsAuthenticated)
    {
        name = user.Identity.Name;
    }
    else
    {
        name = "anonymous";
    }
    return Clients.All.addMessageToPage(name, message);
}
```

Or, you can create an object to represent the authentication information and pass that object as a parameter, as shown below.

```
public class SampleHub : Hub
{
    public override Task OnConnected()
    {
        return Clients.All.joined(GetAuthInfo());
    }

    protected object GetAuthInfo()
    {
        var user = Context.User;
        return new
        {
            IsAuthenticated = user.Identity.IsAuthenticated,
            IsAdmin = user.IsInRole("Admin"),
            UserName = user.Identity.Name
        };
    }
}
```

You should never pass one client's connection id to other clients, as a malicious user could use it to mimic a request from that client.

Authentication options for .NET clients

When you have a .NET client, such as a console app, which interacts with a hub that is limited to authenticated users, you can pass the authentication credentials in a cookie, the connection header, or a certificate. The examples in this section show how to use those different methods for authenticating a user. They are not fully-functional SignalR apps. For more information about .NET clients with SignalR, see Hubs API Guide - .NET Client.

Cookie

When your .NET client interacts with a hub that uses ASP.NET Forms Authentication, you will need to manually set the authentication cookie on the connection. You add the cookie to the `CookieContainer` property on the HubConnection object. The following example shows a console app that retrieves an authentication cookie from a web page and adds that cookie to the connection. The URL `https://www.contoso.com/RemoteLogin` in the example points to a web page that you would need to create. The page would retrieve the posted user name and password, and attempt to log in the user with the credentials.

```
class Program
{
    static void Main(string[] args)
    {
        var connection = new HubConnection("http://www.contoso.com/");
        Cookie returnedCookie;

        Console.Write("Enter user name: ");
        string username = Console.ReadLine();

        Console.Write("Enter password: ");
        string password = Console.ReadLine();

        var authResult = AuthenticateUser(username, password, out returnedCookie);

        if (authResult)
        {
            connection.CookieContainer = new CookieContainer();
            connection.CookieContainer.Add(returnedCookie);
            Console.WriteLine("Welcome " + username);
        }
        else
        {
            Console.WriteLine("Login failed");
        }
    }

    private static bool AuthenticateUser(string user, string password, out Cookie authCookie)
    {
        var request = WebRequest.Create("https://www.contoso.com/RemoteLogin") as HttpWebRequest;
        request.Method = "POST";
        request.ContentType = "application/x-www-form-urlencoded";
        request.CookieContainer = new CookieContainer();

        var authCredentials = "UserName=" + user + "&Password=" + password;
        byte[] bytes = System.Text.Encoding.UTF8.GetBytes(authCredentials);
        request.ContentLength = bytes.Length;
        using (var requestStream = request.GetRequestStream())
        {
            requestStream.Write(bytes, 0, bytes.Length);
        }

        using (var response = request.GetResponse() as HttpWebResponse)
        {
            authCookie = response.Cookies[FormsAuthentication.FormsCookieName];
        }

        if (authCookie != null)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

The console app posts the credentials to www.contoso.com/RemoteLogin which could refer to an empty page that contains the following code-behind file.

```
using System;
using System.Web.Security;

namespace SignalRWithConsoleChat
{
    public partial class RemoteLogin : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            string username = Request["UserName"];
            string password = Request["Password"];
            bool result = Membership.ValidateUser(username, password);
            if (result)
            {
                FormsAuthentication.SetAuthCookie(username, false);
            }
        }
    }
}
```

Windows authentication

When using Windows authentication, you can pass the current user's credentials by using the DefaultCredentials property. You set the credentials for the connection to the value of the DefaultCredentials.

```
class Program
{
    static void Main(string[] args)
    {
        var connection = new HubConnection("http://www.contoso.com/");
        connection.Credentials = CredentialCache.DefaultCredentials;
        connection.Start().Wait();
    }
}
```

Connection header

If your application is not using cookies, you can pass user information in the connection header. For example, you can pass a token in the connection header.

```
class Program
{
    static void Main(string[] args)
    {
        var connection = new HubConnection("http://www.contoso.com/");
        connection.Headers.Add("myauthtoken", /* token data */);
        connection.Start().Wait();
    }
}
```

Then, in the hub, you would verify the user's token.

Certificate

You can pass a client certificate to verify the user. You add the certificate when creating the connection. The following example shows only how to add a client certificate to the connection; it does not show the full console app. It uses the X509Certificate class which provides several different ways to create the certificate.

```
class Program
{
    static void Main(string[] args)
    {
        var connection = new HubConnection("http://www.contoso.com/");
        connection.AddClientCertificate(X509Certificate.CreateFromCertFile("MyCert.cer"));
        connection.Start().Wait();
    }
}
```

Authentication and Authorization for SignalR Persistent Connections (SignalR 1.x)

2/20/2020 • 2 minutes to read • Edit Online

by Patrick Fletcher, Tom FitzMacken

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This topic describes how to enforce authorization on a persistent connection. For general information about integrating security into a SignalR application, see Introduction to Security.

Enforce authorization

To enforce authorization rules when using a PersistentConnection you must override the `AuthorizeRequest` method. You cannot use the `Authorize` attribute with persistent connections. The `AuthorizeRequest` method is called by the SignalR Framework before every request to verify that the user is authorized to perform the requested action. The `AuthorizeRequest` method is not called from the client; instead, you authenticate the user through your application's standard authentication mechanism.

The example below shows how to limit requests to authenticated users.

```
public class AuthenticatedConnection : PersistentConnection
{
    protected override bool AuthorizeRequest(IRequest request)
    {
        return request.User.Identity.IsAuthenticated;
    }
}
```

You can add any customized authorization logic in the AuthorizeRequest method; such as, checking whether a user belongs to a particular role.

SignalR Performance (SignalR 1.x)

2/20/2020 • 8 minutes to read • Edit Online

by Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

> This topic describes how to design for, measure, and improve performance in a SignalR application.

For a recent presentation on SignalR performance and scaling, see Scaling the Real-time Web with ASP.NET SignalR.

This topic contains the following sections:

- Design considerations
- Tuning your SignalR server for performance
- Troubleshooting performance issues
- Using SignalR performance counters
- Using other performance counters
- Other resources

Design considerations

This section describes patterns that can be implemented during the design of a SignalR application, to ensure that performance is not being hindered by generating unnecessary network traffic.

Throttling message frequency

Even in an application that sends out messages at a high frequency (such as a realtime gaming application), most applications don't need to send more than a few messages a second. To reduce the amount of traffic that each client generates, a message loop can be implemented that queues and sends out messages no more frequently than a fixed rate (that is, up to a certain number of messages will be sent every second, if there are messages in that time interval to be sent). For a sample application that throttles messages to a certain rate (from both client and server), see High-Frequency Realtime with SignalR.

Reducing message size

You can reduce the size of a SignalR message by reducing the size of your serialized objects. In server code, if you're sending an object that contains properties that don't need to be transmitted, prevent those properties from being serialized by using the `JsonIgnore` attribute. The names of properties are also stored in the message; the names of properties can be shortened using the `JsonProperty` attribute. The following code sample demonstrates how to exclude a property from being sent to the client, and how to shorten property names:

.NET server code that demonstrates the JsonIgnore attribute to exclude data from being sent to the client, and the JsonProperty attribute to reduce message size

```
using Newtonsoft.Json;
using System;
public class ShapeModel
{
[JsonProperty("l")]
    public double Left { get; set; }
[JsonProperty("t")]
    public double Top { get; set; }
    // We don't want the client to get the "LastUpdatedBy" property
[JsonIgnore]
    public string LastUpdatedBy { get; set; }
}
```

In order to retain readability/ maintainability in the client code, the abbreviated property names can be remapped to human-friendly names after the message is received. The following code sample demonstrates one possible way of remapping shortened names to longer ones, by defining a message contract (mapping), and using the `reMap` function to apply the contract to the optimized message class:

Client-side JavaScript code that remaps shortened property names to human-readable names

```
function reMap(smallObject, contract) {
    var largeObject = {};
    for (var smallProperty in contract) {
        largeObject[contract[smallProperty]] = smallObject[smallProperty];
    }
    return largeObject;
}
var shapeModelContract = {
    l: "left",
    t: "top"
};
myHub.client.setShape = function (shapeModelSmall) {
    var shapeModel = reMap(shapeModelSmall, shapeModelContract);
    // shapeModelSmall has "l" and "t" properties  but after remapping
    // shapeModel now has "left" and "top" properties
};
```

Names can be shortened in messages from the client to the server as well, using the same method.

Reducing the memory footprint (that is, the amount of memory used for the message) of the message object can also improve performance. For example, if the full range of an `int` is not needed, a `short` or `byte` can be used instead.

Since messages are stored in the message bus in server memory, reducing the size of messages can also address server memory issues.

Tuning your SignalR server for performance

The following configuration settings can be used to tune your server for better performance in a SignalR application. For general information on how to improve performance in an ASP.NET application, see Improving ASP.NET Performance.

SignalR configuration settings

- **DefaultMessageBufferSize**: By default, SignalR retains 1000 messages in memory per hub per connection. If large messages are being used, this may create memory issues which can be alleviated by reducing this value. This setting can be set in the `Application_Start` event handler in an ASP.NET application, or in the `Configuration` method of an OWIN startup class in a self-hosted application. The following sample demonstrates how to reduce this value in order to reduce the memory footprint of your application in order to reduce the amount of server memory used:

.NET server code in Global.asax for decreasing default message buffer size

```
protected void Application_Start(object sender, EventArgs e)
{
    GlobalHost.Configuration.DefaultMessageBufferSize = 500;
}
```

IIS configuration settings

- **Max concurrent requests per application**: Increasing the number of concurrent IIS requests will increase server resources available for serving requests. The default value is 5000; to increase this setting, execute the following commands in an elevated command prompt:

```
cd %windir%\System32\inetsrv\
appcmd.exe set config /section:system.webserver/serverRuntime
        /appConcurrentRequestLimit:10000
```

ASP.NET configuration settings

This section includes configuration settings that can be set in the `aspnet.config` file. This file is found in one of two locations, depending on platform:

- `%windir%\Microsoft.NET\Framework\v4.0.30319\aspnet.config`
- `%windir%\Microsoft.NET\Framework64\v4.0.30319\aspnet.config`

ASP.NET settings that may improve SignalR performance include the following:

- **Maximum concurrent requests per CPU**: Increasing this setting may alleviate performance bottlenecks. To increase this setting, add the following configuration setting to the `aspnet.config` file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    <system.web>
        <applicationPool maxConcurrentRequestsPerCPU="20000" />
    </system.web>
</configuration>
```

- **Request Queue Limit**: When the total number of connections exceeds the `maxConcurrentRequestsPerCPU` setting, ASP.NET will start throttling requests using a queue. To increase the size of the queue, you can increase the `requestQueueLimit` setting. To do this, add the following configuration setting to the `processModel` node in `config/machine.config` (rather than `aspnet.config`):

```
<processModel autoConfig="false" requestQueueLimit="250000" />
```

Troubleshooting performance issues

This section describes ways to find performance bottlenecks in your application.

Verifying that WebSocket is being used

While SignalR can use a variety of transports for communication between client and server, WebSocket offers a significant performance advantage, and should be used if the client and server support it. To determine if your client and server meet the requirements for WebSocket, see Transports and Fallbacks. To determine what transport is being used in your application, you can use the browser developer tools, and examine the logs to see what transport is being used for the connection. For information on using the browser development tools in Internet

Explorer and Chrome, see Transports and Fallbacks.

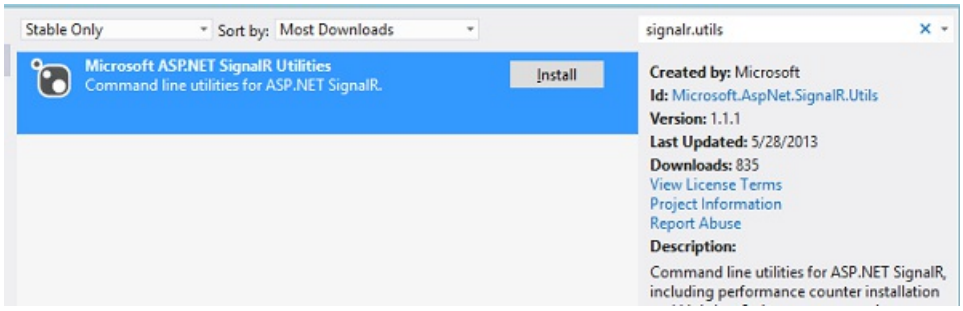
Using SignalR performance counters

This section describes how to enable and use SignalR performance counters, found in the `Microsoft.AspNet.SignalR.Utils` package.

Installing signalr.exe

Performance counters can be added to the server using a utility called SignalR.exe. To install this utility, follow these steps:

1. In Visual Studio, select **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution**
2. Search for **signalr.utils**, and select Install.



3. Accept the license agreement to install the package.
4. SignalR.exe will be installed to `<project folder>/packages/Microsoft.AspNet.SignalR.Utils.<version>/tools`.

Installing performance counters with SignalR.exe

To install SignalR performance counters, run SignalR.exe in an elevated command prompt with the following parameter:

```
SignalR.exe ipc
```

To remove SignalR performance counters, run SignalR.exe in an elevated command prompt with the following parameter:

```
SignalR.exe upc
```

SignalR Performance counters

The utilities package installs the following performance counters. The "Total" counters measure the number of events since the last application pool or server restart.

Connection metrics

The following metrics measure the connection lifetime events that occur. For more information, see Understanding and Handling Connection Lifetime Events.

- Connections Connected
- Connections Reconnected
- Connections Disconnected
- Connections Current

Message metrics

The following metrics measure the message traffic generated by SignalR.

- **Connection Messages Received Total**
- **Connection Messages Sent Total**
- **Connection Messages Received/Sec**
- **Connection Messages Sent/Sec**

Message bus metrics

The following metrics measure traffic through the internal SignalR message bus, the queue in which all incoming and outgoing SignalR messages are placed. A message is **Published** when it is sent or broadcast. A **Subscriber** in this context is a subscription on the message bus; this should equal the number of clients plus the server itself. An **Allocated Worker** is a component that sends data to active connections; a **Busy Worker** is one that is actively sending a message.

- **Message Bus Messages Received Total**
- **Message Bus Messages Received/Sec**
- **Message Bus Messages Published Total**
- **Message Bus Messages Published/Sec**
- **Message Bus Subscribers Current**
- **Message Bus Subscribers Total**
- **Message Bus Subscribers/Sec**
- **Message Bus Allocated Workers**
- **Message Bus Busy Workers**
- **Message Bus Topics Current**

Error metrics

The following metrics measure errors generated by SignalR message traffic. **Hub Resolution** errors occur when a hub or hub method cannot be resolved. **Hub Invocation** errors are exceptions thrown while invoking a hub method. **Transport** errors are connection errors thrown during an HTTP request or response.

- **Errors: All Total**
- **Errors: All/Sec**
- **Errors: Hub Resolution Total**
- **Errors: Hub Resolution/Sec**
- **Errors: Hub Invocation Total**
- **Errors: Hub Invocation/Sec**
- **Errors: Transport Total**
- **Errors: Transport/Sec**

Scaleout metrics

The following metrics measure traffic and errors generated by the scaleout provider. A **Stream** in this context is a scale unit used by the scaleout provider; this is a table if SQL Server is used, a Topic if Service Bus is used, and a Subscription if Redis is used. By default, only one stream is used, but this can be increased through configuration on SQL Server and Service Bus. A **Buffering** stream is one that has entered a faulted state; when the stream is in the faulted state, all messages sent to the backplane will fail immediately until the stream is no longer faulting. The **Send Queue Length** is the number of messages that have been posted but not yet sent.

- **Scaleout Message Bus Messages Received/Sec**
- **Scaleout Streams Total**
- **Scaleout Streams Open**
- **Scaleout Streams Buffering**
- **Scaleout Errors Total**

- **Scaleout Errors/Sec**
- **Scaleout Send Queue Length**

For more information on what these counters are measuring, see SignalR Scaleout with Azure Service Bus.

Using other performance counters

The following performance counters may also be useful in monitoring your application's performance.

Memory

- .NET CLR Memory# bytes in all Heaps (for w3wp)

ASP.NET

- ASP.NET\Requests Current
- ASP.NET\Queued
- ASP.NET\Rejected

CPU

- Processor Information\Processor Time

TCP/IP

- TCPv6/Connections Established
- TCPv4/Connections Established

Web Service

- Web Service\Current Connections
- Web Service\Maximum Connections

Threading

- .NET CLR LocksAndThreads# of current logical Threads
- .NET CLR LocksAnd Threads# of current physical Threads

Other Resources

For more information on ASP.NET performance monitoring and tuning, see the following topics:

- ASP.NET Performance Overview
- ASP.NET Thread Usage on IIS 7.5, IIS 7.0, and IIS 6.0
- <applicationPool> Element (Web Settings)

Introduction to Scaleout in SignalR 1.x

2/20/2020 • 3 minutes to read • Edit Online

by Mike Wasson, Patrick Fletcher

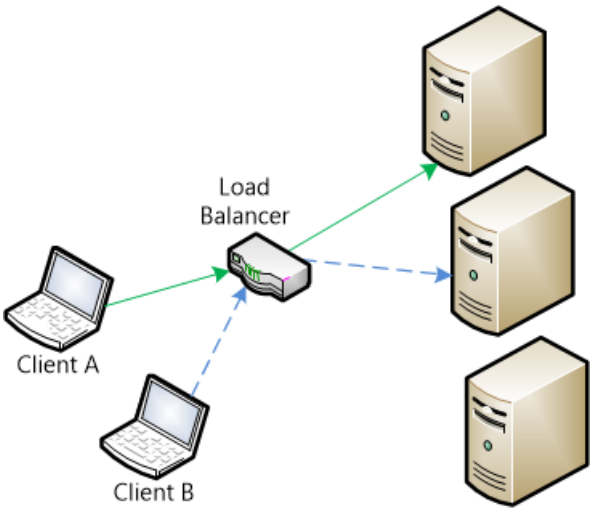
WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

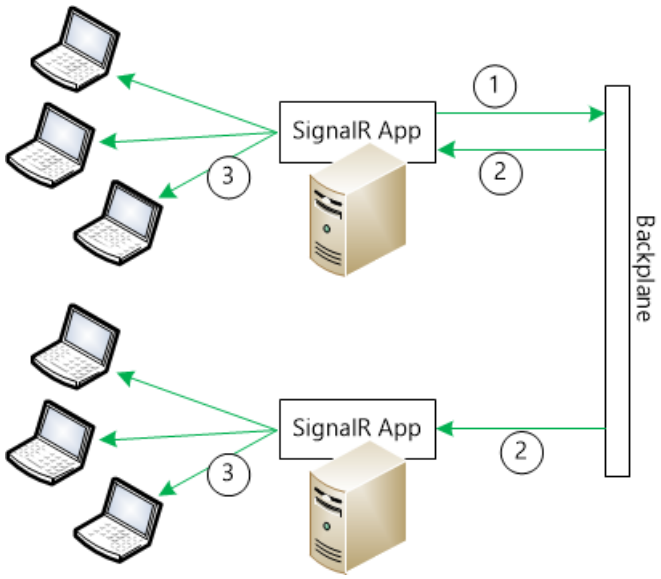
In general, there are two ways to scale a web application: *scale up* and *scale out*.

- Scale up means using a larger server (or a larger VM) with more RAM, CPUs, etc.
- Scale out means adding more servers to handle the load.

The problem with scaling up is that you quickly hit a limit on the size of the machine. Beyond that, you need to scale out. However, when you scale out, clients can get routed to different servers. A client that is connected to one server will not receive messages sent from another server.



One solution is to forward messages between servers, using a component called a *backplane*. With a backplane enabled, each application instance sends messages to the backplane, and the backplane forwards them to the other application instances. (In electronics, a backplane is a group of parallel connectors. By analogy, a SignalR backplane connects multiple servers.)



SignalR currently provides three backplanes:

- **Azure Service Bus**. Service Bus is a messaging infrastructure that allows components to send messages in a loosely coupled way.
- **Redis**. Redis is an in-memory key-value store. Redis supports a publish/subscribe ("pub/sub") pattern for sending messages.
- **SQL Server**. The SQL Server backplane writes messages to SQL tables. The backplane uses Service Broker for efficient messaging. However, it also works if Service Broker is not enabled.

If you deploy your application on Azure, consider using the Azure Service Bus backplane. If you are deploying to your own server farm, consider the SQL Server or Redis backplanes.

The following topics contain step-by-step tutorials for each backplane:

- SignalR Scaleout with Azure Service Bus
- SignalR Scaleout with Redis
- SignalR Scaleout with SQL Server

Implementation

In SignalR, every message is sent through a message bus. A message bus implements the IMessageBus interface, which provides a publish/subscribe abstraction. The backplanes work by replacing the default **IMessageBus** with a bus designed for that backplane. For example, the message bus for Redis is RedisMessageBus, and it uses the Redis pub/sub mechanism to send and receive messages.

Each server instance connects to the backplane through the bus. When a message is sent, it goes to the backplane, and the backplane sends it to every server. When a server gets a message from the backplane, it puts the message in its local cache. The server then delivers messages to clients from its local cache.

For each client connection, the client's progress in reading the message stream is tracked using a cursor. (A cursor represents a position in the message stream.) If a client disconnects and then reconnects, it asks the bus for any messages that arrived after the client's cursor value. The same thing happens when a connection uses long polling. After a long poll request completes, the client opens a new connection and asks for messages that arrived after the cursor.

The cursor mechanism works even if a client is routed to a different server on reconnect. The backplane is aware of all the servers, and it doesn't matter which server a client connects to.

Limitations

Using a backplane, the maximum message throughput is lower than it is when clients talk directly to a single server node. That's because the backplane forwards every message to every node, so the backplane can become a bottleneck. Whether this limitation is a problem depends on the application. For example, here are some typical SignalR scenarios:

- Server broadcast (e.g., stock ticker): Backplanes work well for this scenario, because the server controls the rate at which messages are sent.
- Client-to-client (e.g., chat): In this scenario, the backplane might be a bottleneck if the number of messages scales with the number of clients; that is, if the rate of messages grows proportionally as more clients join.
- High-frequency realtime (e.g., real-time games): A backplane is not recommended for this scenario.

Enabling Tracing For SignalR Scaleout

To enable tracing for the backplanes, add the following sections to the web.config file, under the root **configuration** element:

```
<configuration>
  <system.diagnostics>
    <sources>
      <source name="SignalR.SqlMessageBus">
        <listeners>
          <add name="SignalR-Bus" />
        </listeners>
      </source>
      <source name="SignalR.ServiceBusMessageBus">
        <listeners>
          <add name="SignalR-Bus" />
        </listeners>
      </source>
      <source name="SignalR.ScaleoutMessageBus">
        <listeners>
          <add name="SignalR-Bus" />
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="SignalRSwitch" value="Verbose" />
      <!-- Off, Critical, Error, Warning, Information, Verbose -->
    </switches>
    <sharedListeners>
      <add name="SignalR-Bus"
          type="System.Diagnostics.TextWriterTraceListener"
          initializeData="bus.log.txt" />
    </sharedListeners>
    <trace autoflush="true" />
  </system.diagnostics>
  . . .
</configuration>
```

SignalR Scaleout with Azure Service Bus (SignalR 1.x)

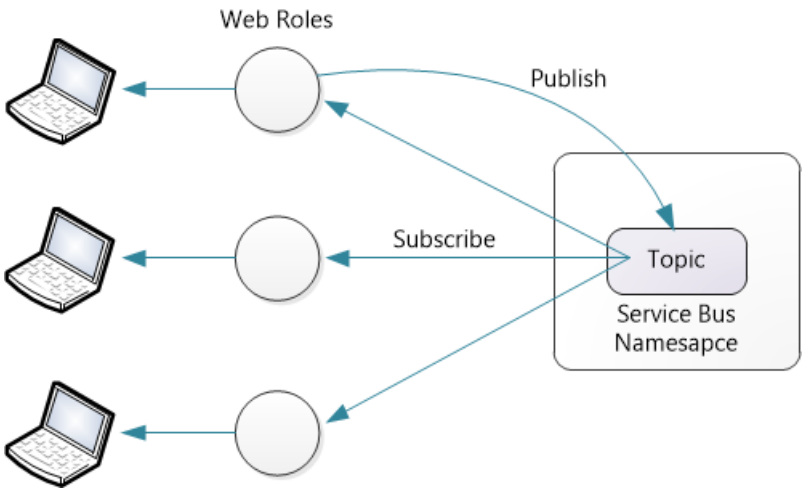
2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson, Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

In this tutorial, you will deploy a SignalR application to a Windows Azure Web Role, using the Service Bus backplane to distribute messages to each role instance.



Prerequisites:

- A Windows Azure account.
- The Windows Azure SDK.
- Visual Studio 2012.

The service bus backplane is also compatible with Service Bus for Windows Server, version 1.1. However, it is not compatible with version 1.0 of Service Bus for Windows Server.

Pricing

The Service Bus backplane uses topics to send messages. For the latest pricing information, see Service Bus. At the time of this writing, you can send 1,000,000 messages per month for less than $1. The backplane sends a service bus message for each invocation of a SignalR hub method. There are also some control messages for connections, disconnections, joining or leaving groups, and so forth. In most applications, the majority of the message traffic will be hub method invocations.

Overview

Before we get to the detailed tutorial, here is a quick overview of what you will do.

1. Use the Windows Azure portal to create a new Service Bus namespace.
2. Add these NuGet packages to your application:
   - Microsoft.AspNet.SignalR

- Microsoft.AspNet.SignalR.ServiceBus

3. Create a SignalR application.

4. Add the following code to Global.asax to configure the backplane:

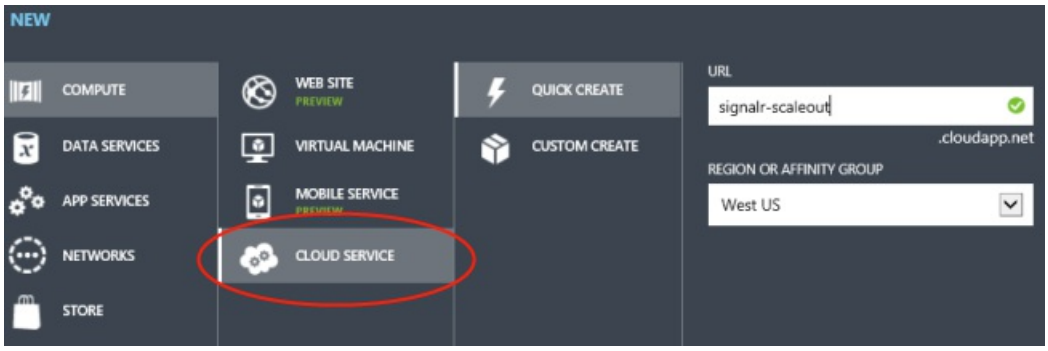
```
protected void Application_Start()
{
    string connectionString = "Service Bus connection string";
    GlobalHost.DependencyResolver.UseServiceBus(connectionString, "YourAppName");

    RouteTable.Routes.MapHubs();
    // ...
}
```

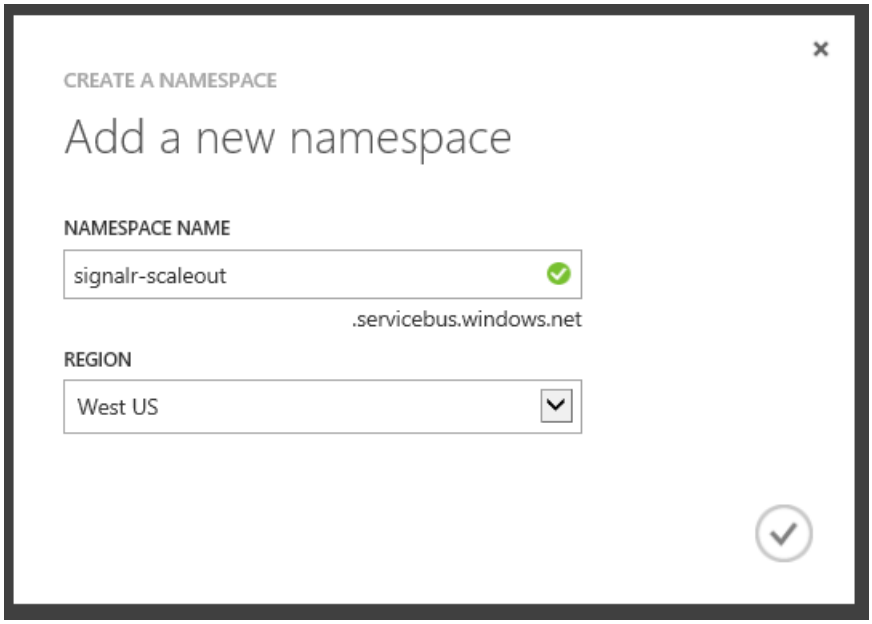
For each application, pick a different value for "YourAppName". Do not use the same value across multiple applications.

Create the Azure Services

Create a Cloud Service, as described in How to Create and Deploy a Cloud Service. Follow the steps in the section "How to: Create a cloud service using Quick Create". For this tutorial, you do not need to upload a certificate.



Create a new Service Bus namespace, as described in How to Use Service Bus Topics/Subscriptions. Follow the steps in the section "Create a Service Namespace".

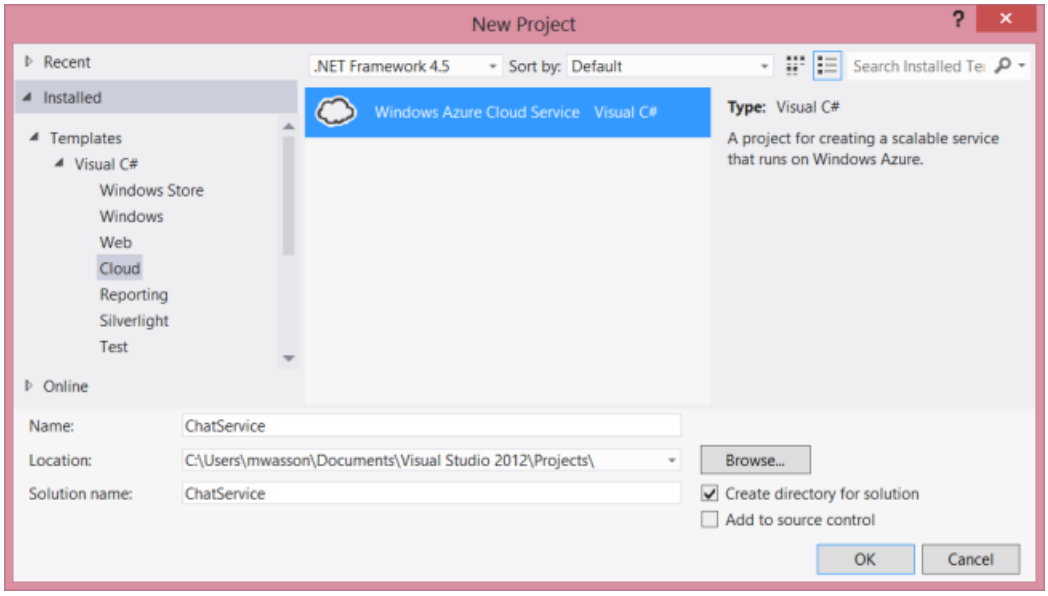


> **NOTE**
>
> Make sure to select the same region for the cloud service and the Service Bus namespace.

Create the Visual Studio Project

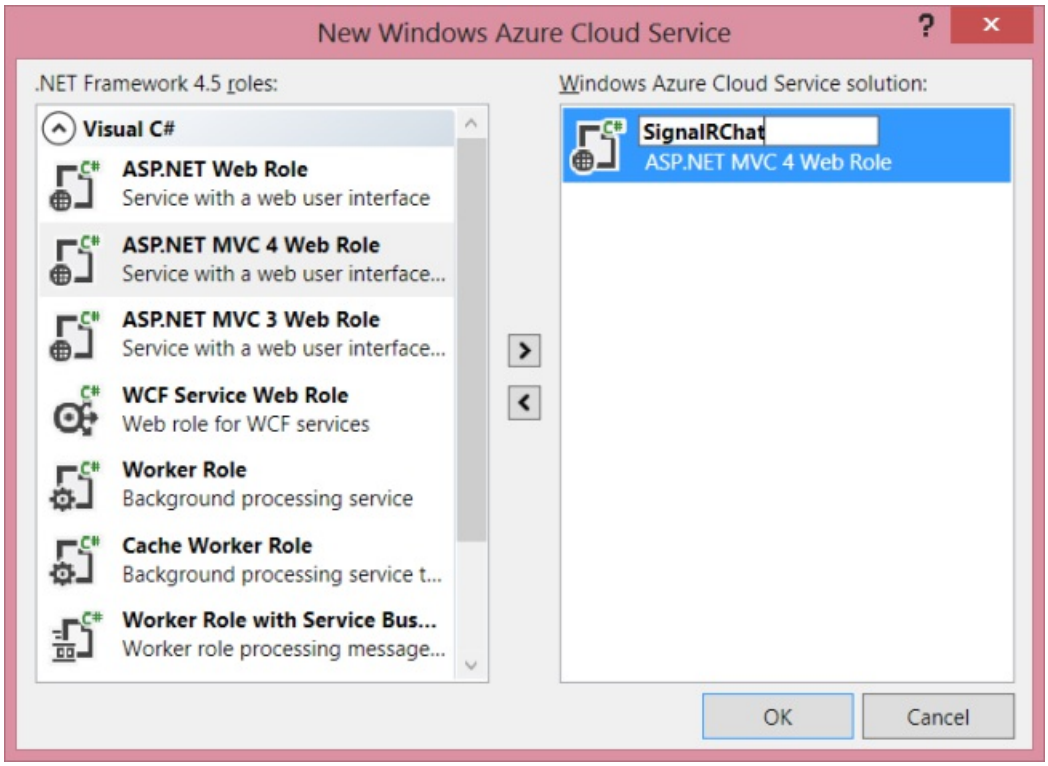
Start Visual Studio. From the **File** menu, click **New Project**.

In the **New Project** dialog box, expand **Visual C#**. Under **Installed Templates**, select **Cloud** and then select **Windows Azure Cloud Service**. Keep the default .NET Framework 4.5. Name the application ChatService and click **OK**.



In the **New Windows Azure Cloud Service** dialog, select ASP.NET MVC 4 Web Role. Click the right-arrow button (>) to add the role to your solution.

Hover the mouse over the new role, so the pencil icon visible. Click this icon to rename the role. Name the role "SignalRChat" and click **OK**.



In the **New ASP.NET MVC 4 Project** wizard, select **Internet Application**. Click **OK**. The project wizard creates two projects:

- ChatService: This project is the Windows Azure application. It defines the Azure roles and other configuration options.

- SignalRChat: This project is your ASP.NET MVC 4 project.

Create the SignalR Chat Application

To create the chat application, follow the steps in the tutorial Getting Started with SignalR and MVC 4.

Use NuGet to install the required libraries. From the **Tools** menu, select **NuGet Package Manager**, then select **Package Manager Console**. In the **Package Manager Console** window, enter the following commands:

```
Install-Package -ProjectName SignalRChat Microsoft.AspNet.SignalR
Install-Package -ProjectName SignalRChat Microsoft.AspNet.SignalR.ServiceBus
```

Use the `-ProjectName` option to install the packages to the ASP.NET MVC project, rather than the Windows Azure project.

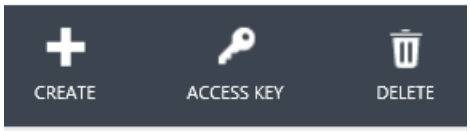
Configure the Backplane

In your application's Global.asax file, add the following code:

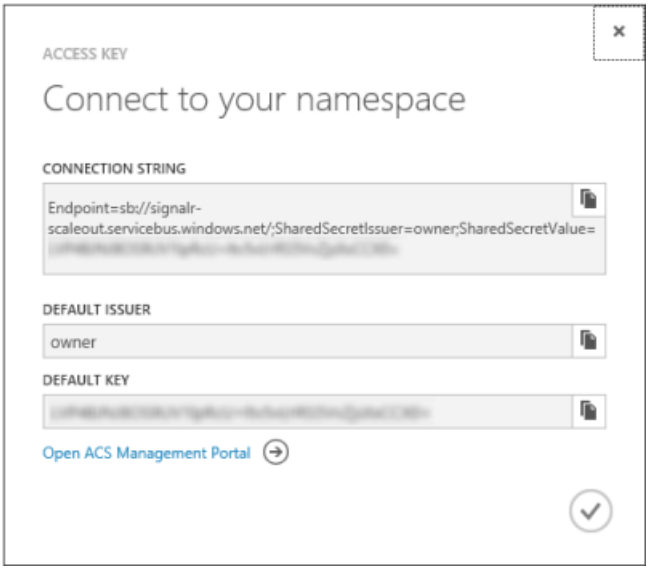
```
protected void Application_Start()
{
    string connectionString = "";
    GlobalHost.DependencyResolver.UseServiceBus(connectionString, "Chat");

    RouteTable.Routes.MapHubs();
}
```

Now you need to get your service bus connection string. In the Azure portal, select the service bus namespace that you created and click the Access Key icon.



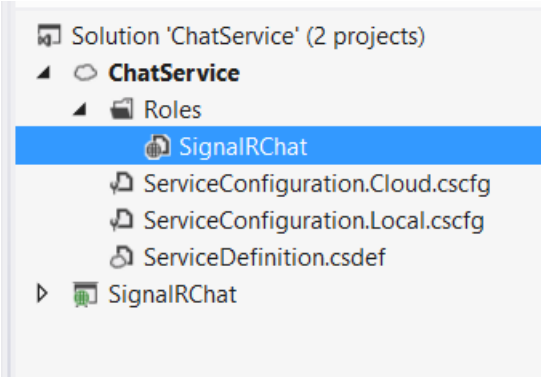
Copy the connection string to the clipboard, then paste it into the *connectionString* variable.



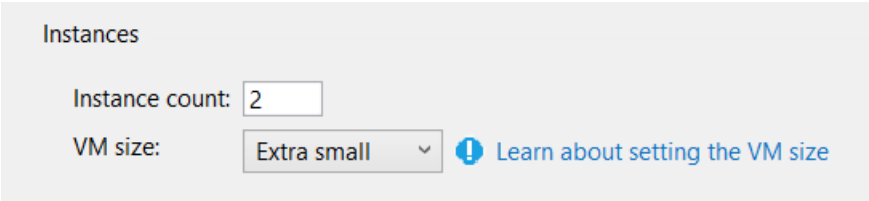

```
string connectionString = "Endpoint=sb://xxxx.servicebus.windows.net/;
SharedSecretIssuer=owner;SharedSecretValue=XXXXXXXX";
```

Deploy to Azure

In Solution Explorer, expand the **Roles** folder inside the ChatService project.

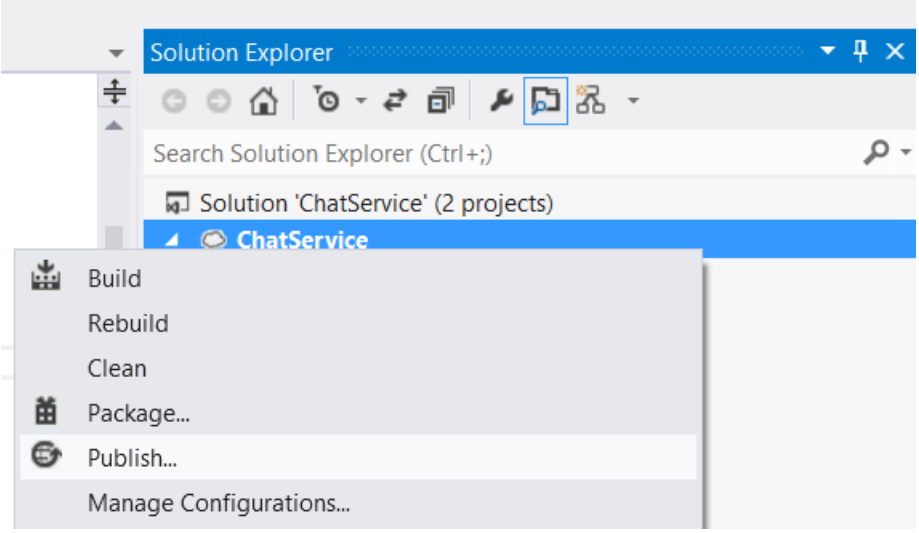


Right-click the SignalRChat role and select **Properties**. Select the **Configuration** tab. Under **Instances** select 2. You can also set the VM size to **Extra Small**.

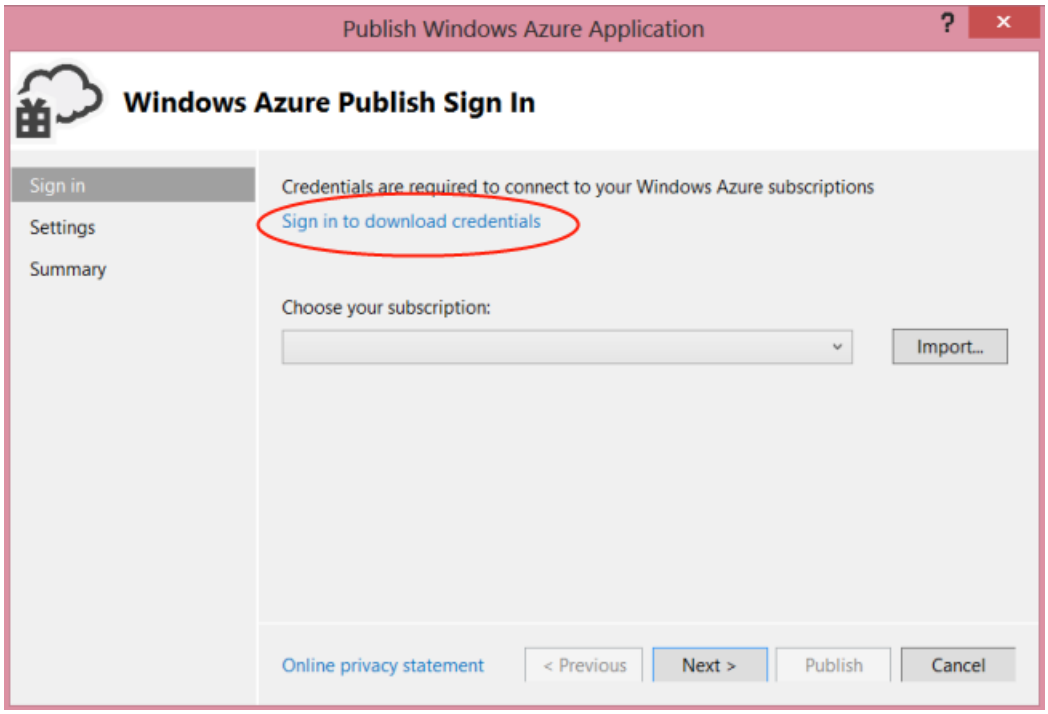


Save the changes.

In Solution Explorer, right-click the ChatService project. Select **Publish**.

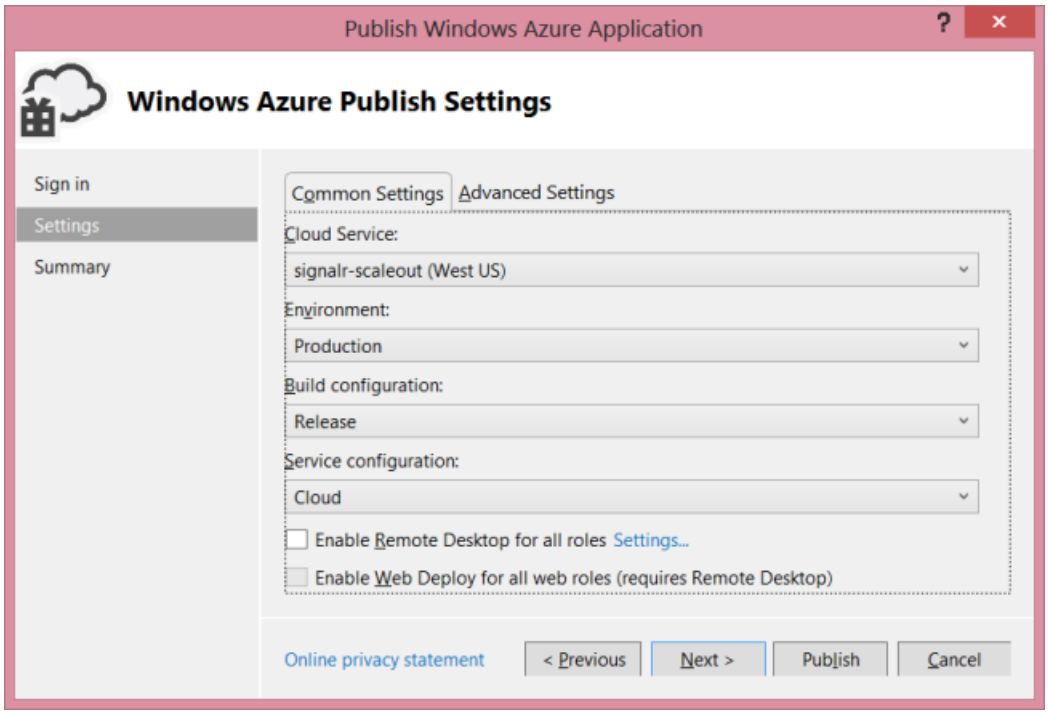


If this is your first time publishing to Windows Azure, you must download your credentials. In the **Publish** wizard, click "Sign in to download credentials". This will prompt you to sign into the Windows Azure portal and download a publish settings file.



Click **Import** and select the publish settings file that you downloaded.

Click **Next**. In the **Publish Settings** dialog, under **Cloud Service**, select the cloud service that you created earlier.



Click **Publish**. It can take a few minutes to deploy the application and start the VMs.

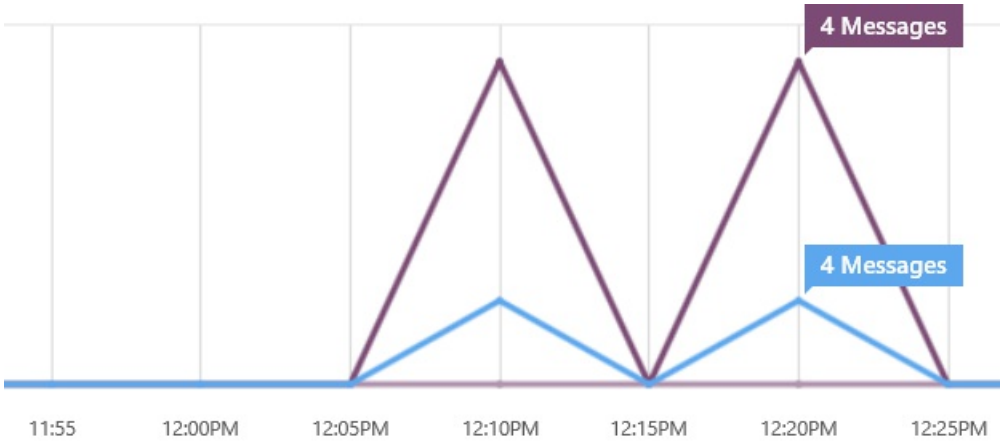
Now when you run the chat application, the role instances communicate through Azure Service Bus, using a Service Bus topic. A topic is a message queue that allows multiple subscribers.

The backplane automatically creates the topic and the subscriptions. To see the subscriptions and message activity, open the Azure portal, select the Service Bus namespace, and click on "Topics".

ALL QUEUES TOPICS RELAYS NOTIFICATION HUBS PREVIEW

NAME	STATUS	CURRENT SIZE	MAX SIZE	SUBSCRIPTIONS
signalr_topic_chat_0	Active	0 B	1 GB	2

It make take a few minutes for the message activity to show up in the dashboard.



SignalR manages the topic lifetime. As long as your application is deployed, don't try to manually delete topics or change settings on the topic.

SignalR Scaleout with Redis (SignalR 1.x)

2/20/2020 • 4 minutes to read • Edit Online

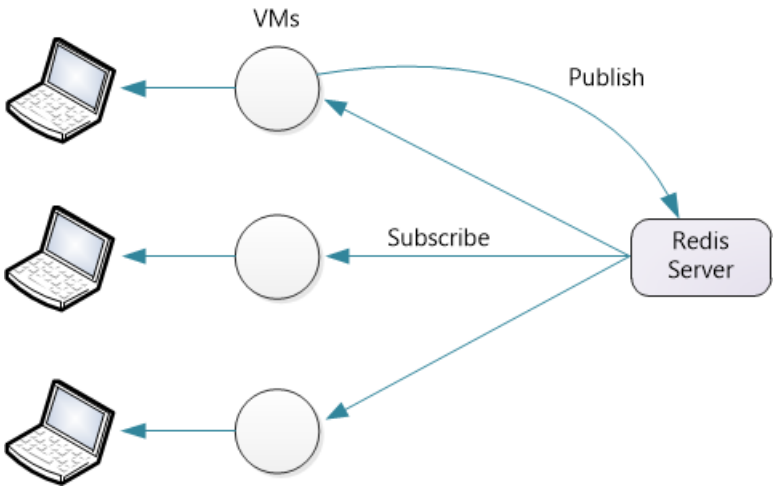
by Mike Wasson, Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

In this tutorial, you will use Redis to distribute messages across a SignalR application that is deployed on two separate IIS instances.

Redis is an in-memory key-value store. It also supports a messaging system with a publish/subscribe model. The SignalR Redis backplane uses the pub/sub feature to forward messages to other servers.



For this tutorial, you will use three servers:

- Two servers running Windows, which you will use to deploy a SignalR application.
- One server running Linux, which you will use to run Redis. For the screenshots in this tutorial, I used Ubuntu 12.04 TLS.

If you don't have three physical servers to use, you can create VMs on Hyper-V. Another option is to create VMs on Azure.

Although this tutorial uses the official Redis implementation, there is also a Windows port of Redis from MSOpenTech. Setup and configuration are different, but otherwise the steps are the same.

NOTE

SignalR scaleout with Redis does not support Redis clusters.

Overview

Before we get to the detailed tutorial, here is a quick overview of what you will do.

1. Install Redis and start the Redis server.
2. Add these NuGet packages to your application:

- Microsoft.AspNet.SignalR
- Microsoft.AspNet.SignalR.Redis

3. Create a SignalR application.

4. Add the following code to Global.asax to configure the backplane:

```
protected void Application_Start()
{
    GlobalHost.DependencyResolver.UseRedis("server", port, "password", "AppName");

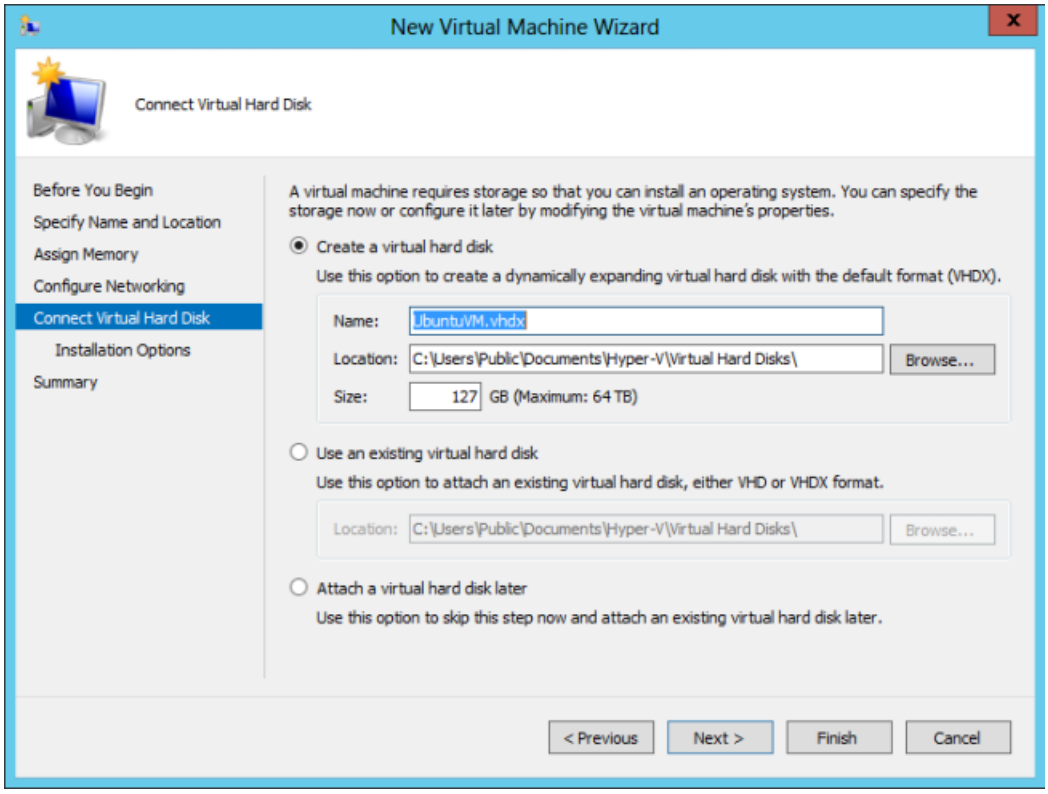
    RouteTable.Routes.MapHubs();
    // ...
}
```

Ubuntu on Hyper-V

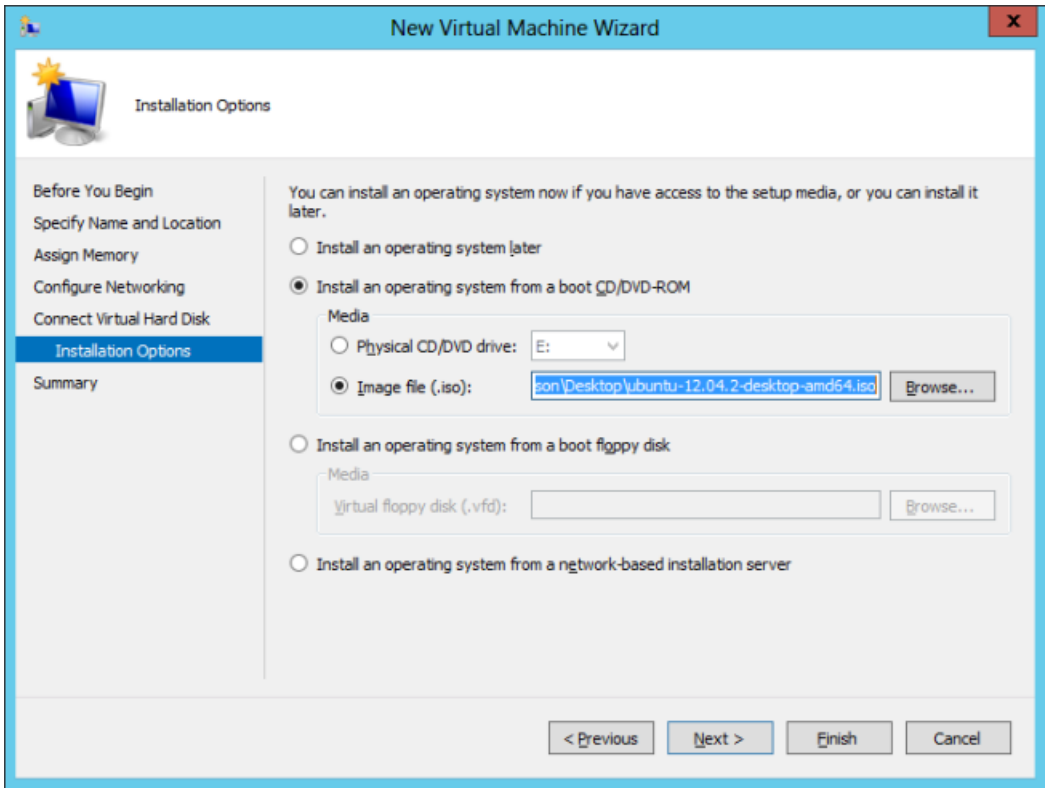
Using Windows Hyper-V, you can easily create an Ubuntu VM on Windows Server.

Download the Ubuntu ISO from http://www.ubuntu.com.

In Hyper-V, add a new VM. In the **Connect Virtual Hard Disk** step, select **Create a virtual hard disk**.



In the **Installation Options** step, select **Image file (.iso)**, click **Browse**, and browse to the Ubuntu installation ISO.



Install Redis

Follow the steps at http://redis.io/download to download and build Redis.

```
wget http://redis.googlecode.com/files/redis-2.6.12.tar.gz
tar xzf redis-2.6.12.tar.gz
cd redis-2.6.12
make
```

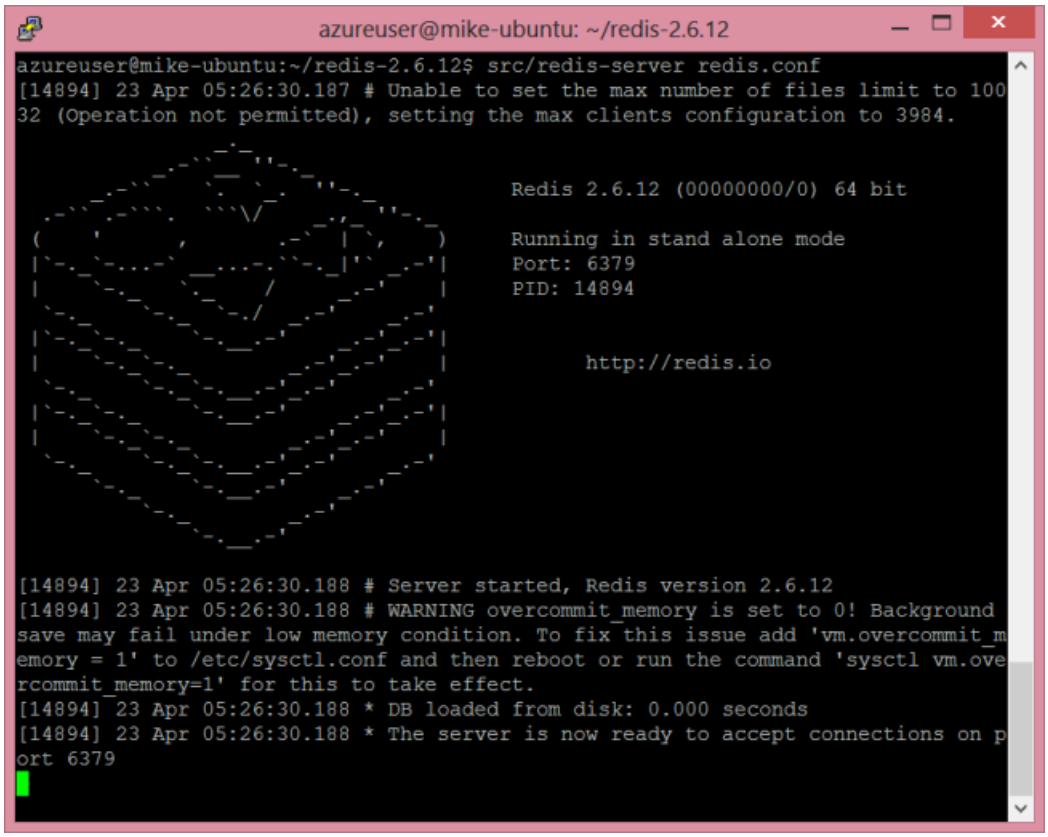
This builds the Redis binaries in the `src` directory.

By default, Redis does not require a password. To set a password, edit the `redis.conf` file, which is located in the root directory of the source code. (Make a backup copy of the file before you edit it!) Add the following directive to `redis.conf`:

```
requirepass YourStrongPassword1234
```

Now start the Redis server:

```
src/redis-server redis.conf
```



Open port 6379, which is the default port that Redis listens on. (You can change the port number in the configuration file.)

Create the SignalR Application

Create a SignalR application by following either of these tutorials:

- Getting Started with SignalR
- Getting Started with SignalR and MVC 4

Next, we'll modify the chat application to support scaleout with Redis. First, add the SignalR.Redis NuGet package to your project. In Visual Studio, from the **Tools** menu, select **NuGet Package Manager**, then select **Package Manager Console**. In the Package Manager Console window, enter the following command:

```
Install-Package Microsoft.AspNet.SignalR.Redis
```

Next, open the Global.asax file. Add the following code to the **Application_Start** method:

```
protected void Application_Start()
{
    GlobalHost.DependencyResolver.UseRedis("server", port, "password", "AppName");

    RouteTable.Routes.MapHubs();
}
```

- "server" is the name of the server that is running Redis.
- *port* is the port number
- "password" is the password that you defined in the redis.conf file.
- "AppName" is any string. SignalR creates a Redis pub/sub channel with this name.

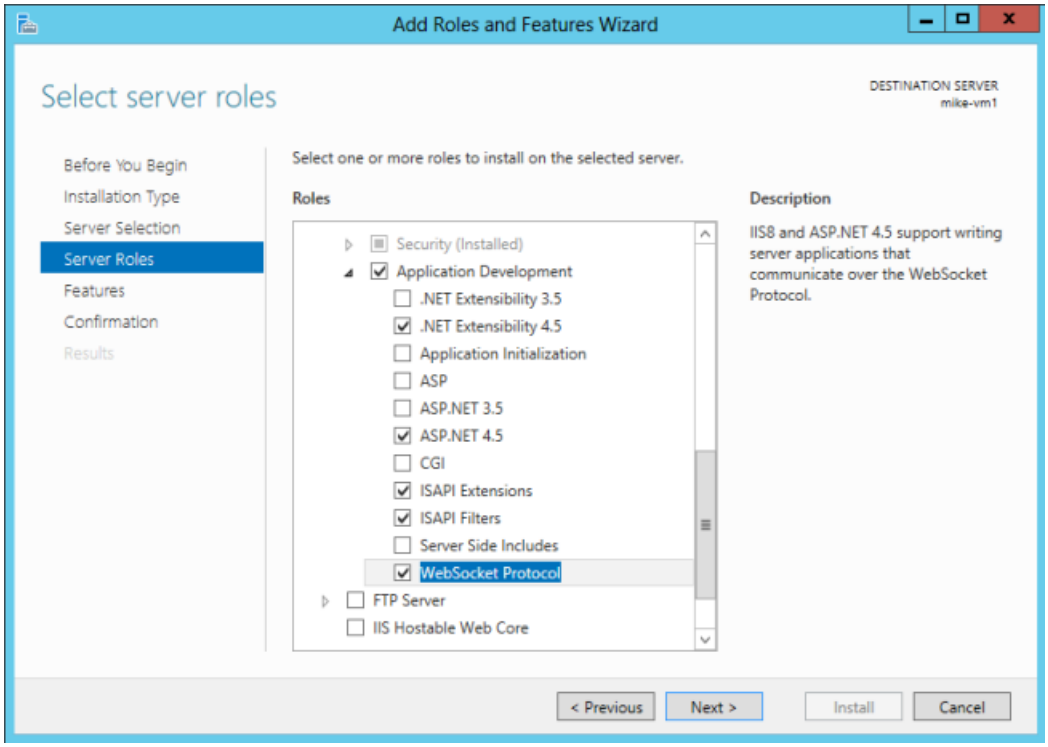
For example:

```
GlobalHost.DependencyResolver.UseRedis("redis-server.cloudapp.net", 6379,
    "MyStrongPassword1234", "ChatApp");
```

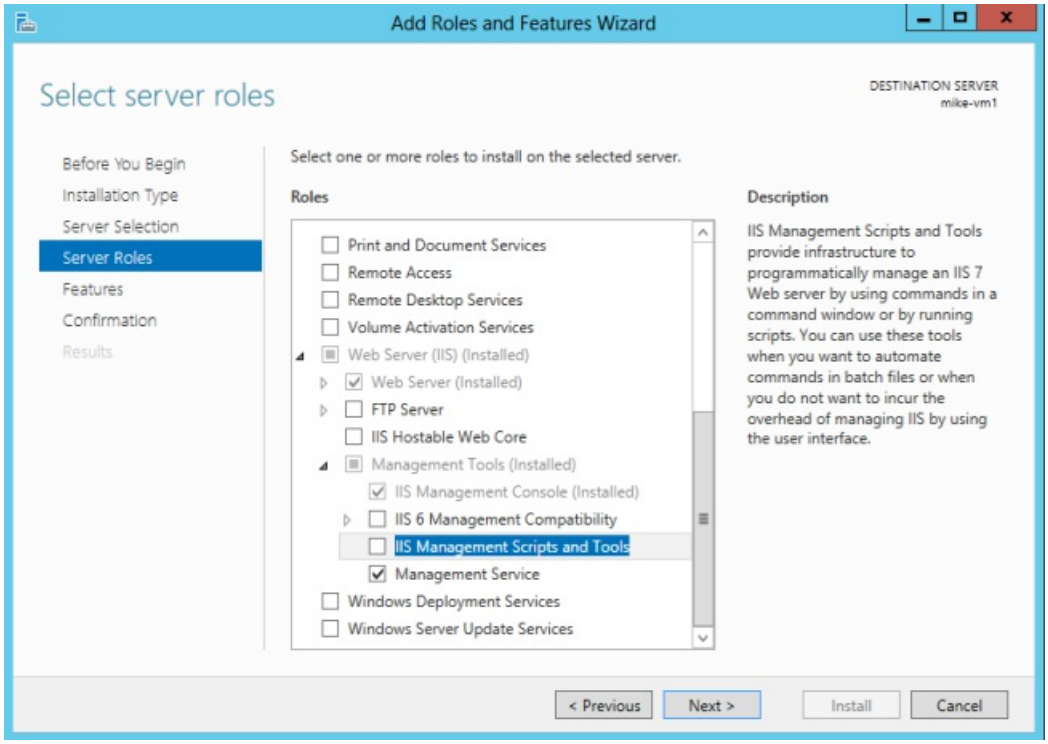
Deploy and Run the Application

Prepare your Windows Server instances to deploy the SignalR application.

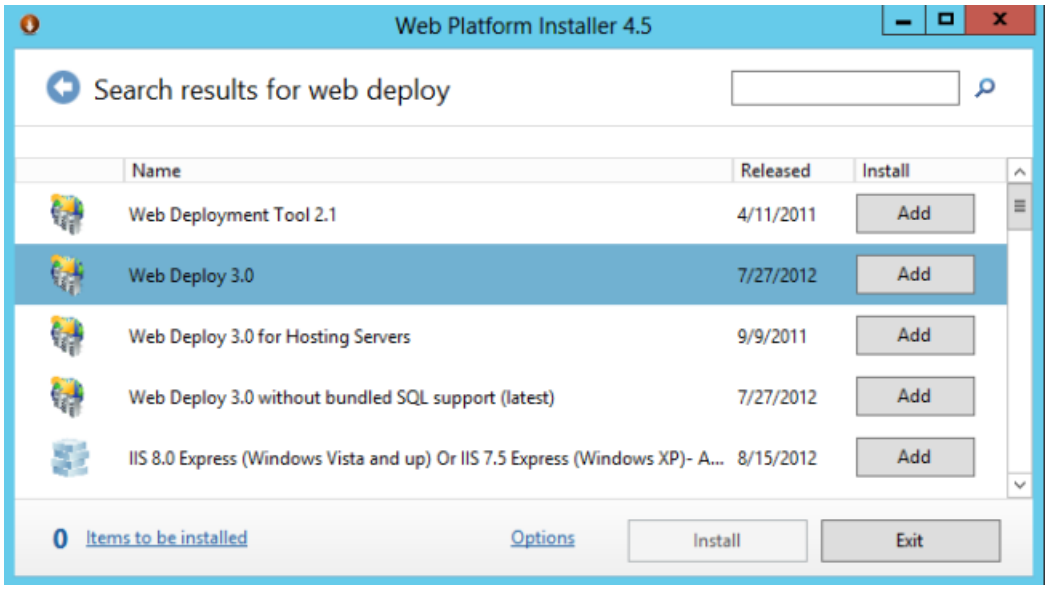
Add the IIS role. Include "Application Development" features, including the WebSocket Protocol.



Also include the Management Service (listed under "Management Tools").



Install Web Deploy 3.0. When you run IIS Manager, it will prompt you to install Microsoft Web Platform, or you can download the installer. In the Platform Installer, search for Web Deploy and install Web Deploy 3.0



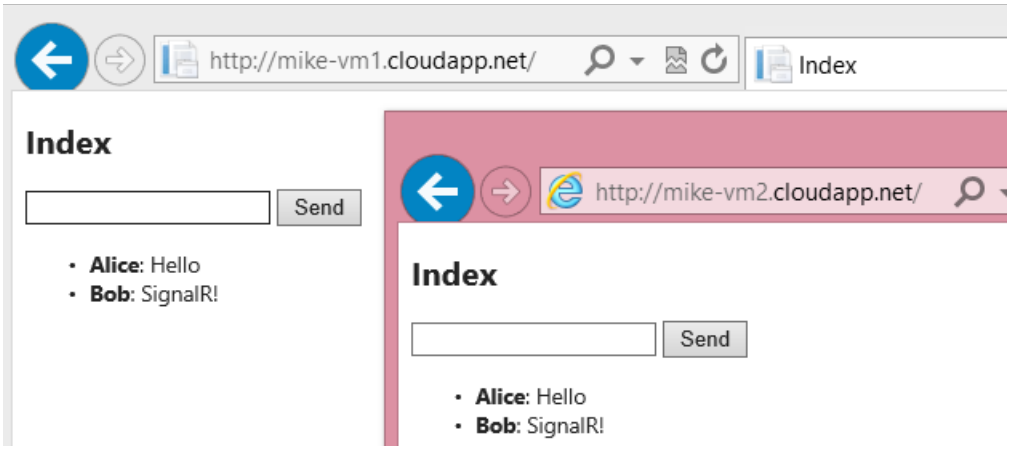
Check that the Web Management Service is running. If not, start the service. (If you don't see Web Management Service in the list of Windows services, make sure that you installed the Management Service when you added the IIS role.)

By default, the Web Management Service listens on TCP port 8172. In Windows Firewall, create a new inbound rule to allow TCP traffic on port 8172. For more information, see Configuring Firewall Rules. (If you are hosting the VMs on Azure, you can do this directly in the Azure portal. See How to Set Up Endpoints to a Virtual Machine.)

Now you are ready to deploy the Visual Studio project from your development machine to the server. In Solution Explorer, right-click the solution and click **Publish**.

For more detailed documentation about web deployment, see Web Deployment Content Map for Visual Studio and ASP.NET.

If you deploy the application to two servers, you can open each instance in a separate browser window and see that they each receive SignalR messages from the other. (Of course, in a production environment, the two servers would sit behind a load balancer.)



If you're curious to see the messages that are sent to Redis, you can use the **redis-cli** client, which installs with Redis.

```
redis-cli -a password
SUBSCRIBE ChatApp
```

azureuser@mike-ubuntu: ~/redis-2.6.12/src

```
azureuser@mike-ubuntu:~/redis-2.6.12/src$ ./redis-cli
redis 127.0.0.1:6379> SUBSCRIBE ChatApp
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "ChatApp"
3) (integer) 1
1) "message"
2) "ChatApp"
3) "$\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00$1f3a95bf-3bde-4339-bee6-95e2a7
e51a88\th-ChatHub6\x00\x00\x00{\"H\":\"ChatHub\",\"M\":\"addMessage\",\"A\":[\"A
lice\",\"Hello\"]}\x00\x00\x00\x00"
1) "message"
2) "ChatApp"
3) "%\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00$e52a3a87-5e1c-4927-8b82-68ea47
d84a3c\th-ChatHub1\x00\x00\x00{\"H\":\"ChatHub\",\"M\":\"addMessage\",\"A\":[\"B
ob\",\"Hi\"]}\x00\x00\x00\x00"
```

SignalR Scaleout with SQL Server (SignalR 1.x)

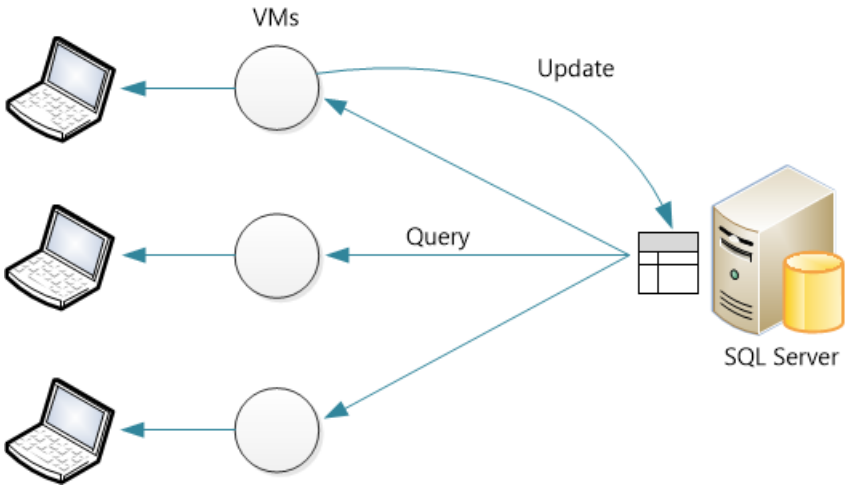
2/20/2020 • 3 minutes to read • Edit Online

by Mike Wasson, Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

In this tutorial, you will use SQL Server to distribute messages across a SignalR application that is deployed in two separate IIS instances. You can also run this tutorial on a single test machine, but to get the full effect, you need to deploy the SignalR application to two or more servers. You must also install SQL Server on one of the servers, or on a separate dedicated server. Another option is to run the tutorial using VMs on Azure.



Prerequisites

Microsoft SQL Server 2005 or later. The backplane supports both desktop and server editions of SQL Server. It does not support SQL Server Compact Edition or Azure SQL Database. (If your application is hosted on Azure, consider the Service Bus backplane instead.)

Overview

Before we get to the detailed tutorial, here is a quick overview of what you will do.

1. Create a new empty database. The backplane will create the necessary tables in this database.
2. Add these NuGet packages to your application:
   - Microsoft.AspNet.SignalR
   - Microsoft.AspNet.SignalR.SqlServer
3. Create a SignalR application.
4. Add the following code to Global.asax to configure the backplane:

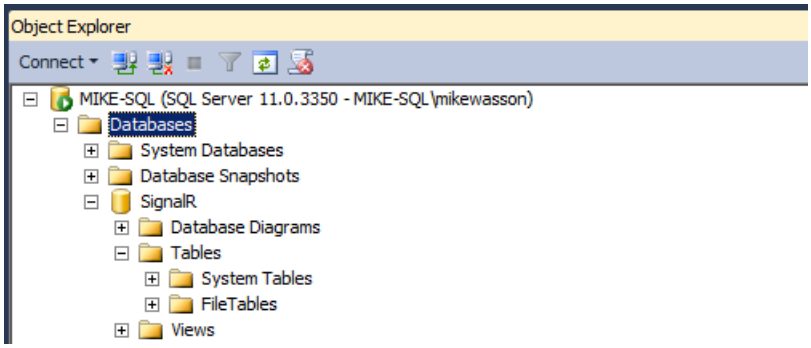
```
protected void Application_Start()
{
    string sqlConnectionString = "Connecton string to your SQL DB";
    GlobalHost.DependencyResolver.UseSqlServer(sqlConnectionString);

    RouteTable.Routes.MapHubs();
    // ...
}
```

Configure the Database

Decide whether the application will use Windows authentication or SQL Server authentication to access the database. Regardless, make sure the database user has permissions to log in, create schemas, and create tables.

Create a new database for the backplane to use. You can give the database any name. You don't need to create any tables in the database; the backplane will create the necessary tables.

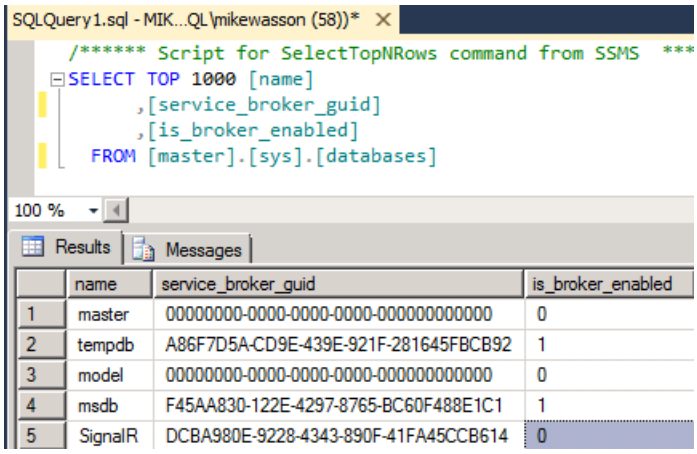


Enable Service Broker

It is recommended to enable Service Broker for the backplane database. Service Broker provides native support for messaging and queuing in SQL Server, which lets the backplane receive updates more efficiently. (However, the backplane also works without Service Broker.)

To check whether Service Broker is enabled, query the **is_broker_enabled** column in the **sys.databases** catalog view.

```
SELECT [name], [service_broker_guid], [is_broker_enabled]
FROM [master].[sys].[databases]
```



To enable Service Broker, use the following SQL query:

```
ALTER DATABASE YOUR_DATABASE SET ENABLE_BROKER
```

> **NOTE**
> If this query appears to deadlock, make sure there are no applications connected to the DB.

If you have enabled tracing, the traces will also show whether Service Broker is enabled.

Create a SignalR Application

Create a SignalR application by following either of these tutorials:

- Getting Started with SignalR
- Getting Started with SignalR and MVC 4

Next, we'll modify the chat application to support scaleout with SQL Server. First, add the SignalR.SqlServer NuGet package to your project. In Visual Studio, from the **Tools** menu, select **NuGet Package Manager**, then select **Package Manager Console**. In the Package Manager Console window, enter the following command:

```
Install-Package Microsoft.AspNet.SignalR.SqlServer
```

Next, open the Global.asax file. Add the following code to the **Application_Start** method:

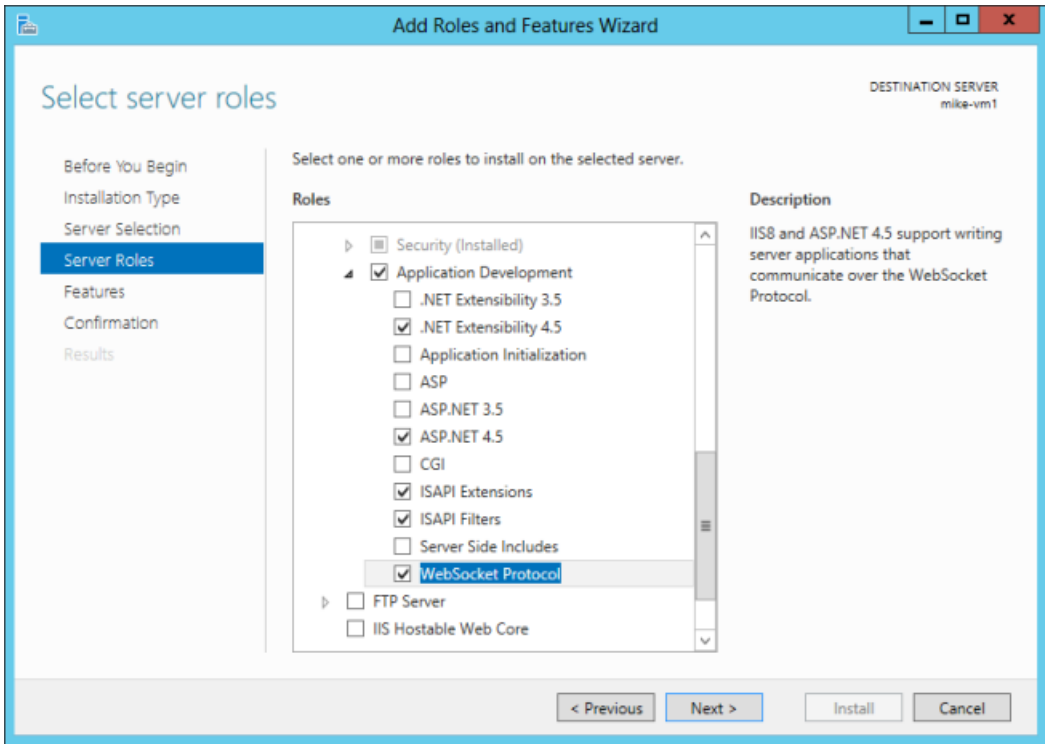
```
protected void Application_Start()
{
    string sqlConnectionString = "<add your SQL connection string here>";
    GlobalHost.DependencyResolver.UseSqlServer(sqlConnectionString);

    RouteTable.Routes.MapHubs();
}
```

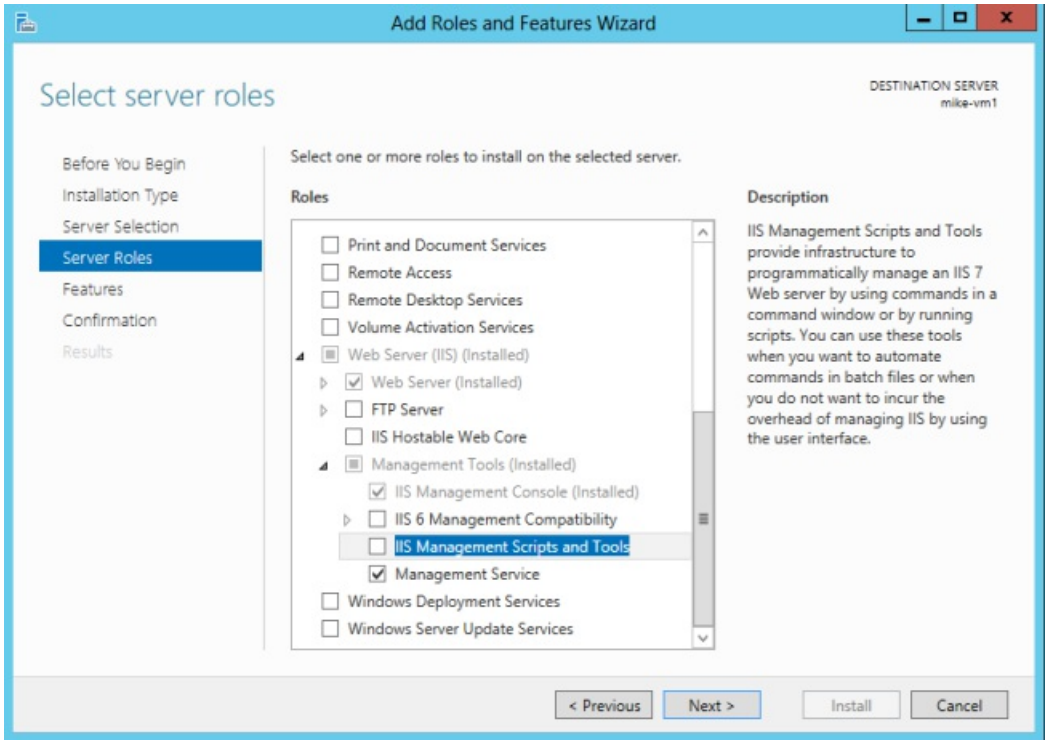
Deploy and Run the Application

Prepare your Windows Server instances to deploy the SignalR application.

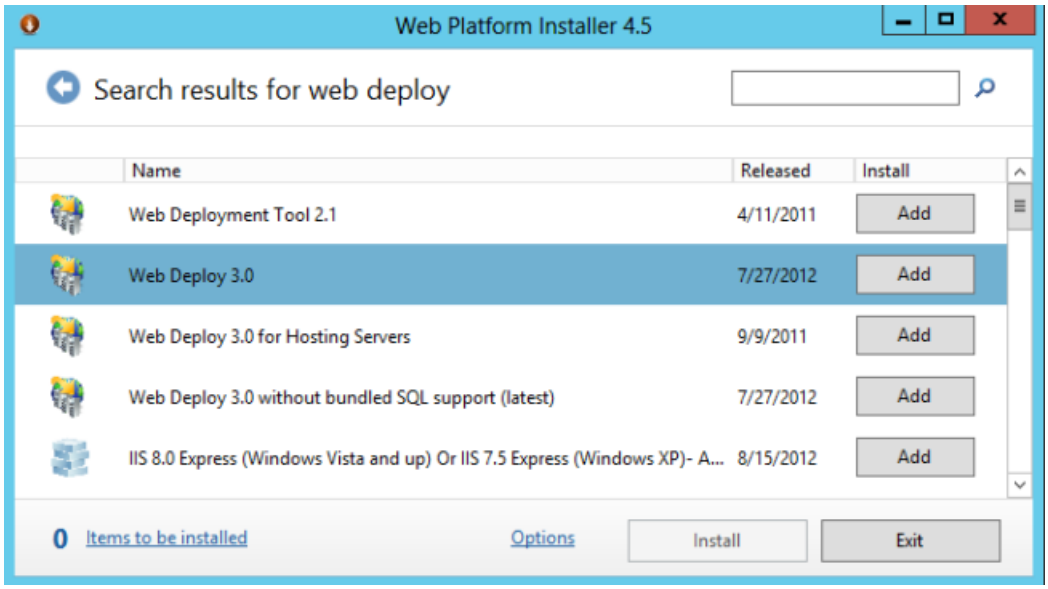
Add the IIS role. Include "Application Development" features, including the WebSocket Protocol.



Also include the Management Service (listed under "Management Tools").



Install Web Deploy 3.0. When you run IIS Manager, it will prompt you to install Microsoft Web Platform, or you can download the installer. In the Platform Installer, search for Web Deploy and install Web Deploy 3.0



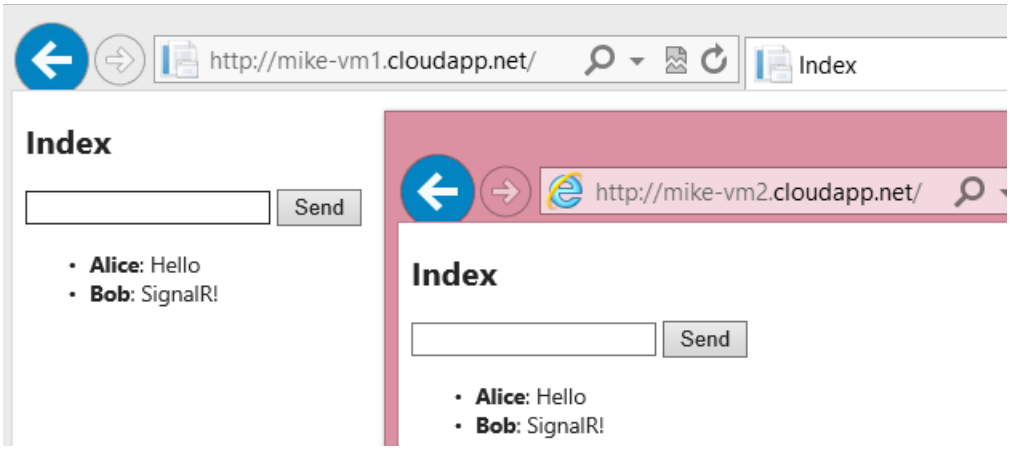
Check that the Web Management Service is running. If not, start the service. (If you don't see Web Management Service in the list of Windows services, make sure that you installed the Management Service when you added the IIS role.)

Finally, open port 8172 for TCP. This is the port that the Web Deploy tool uses.

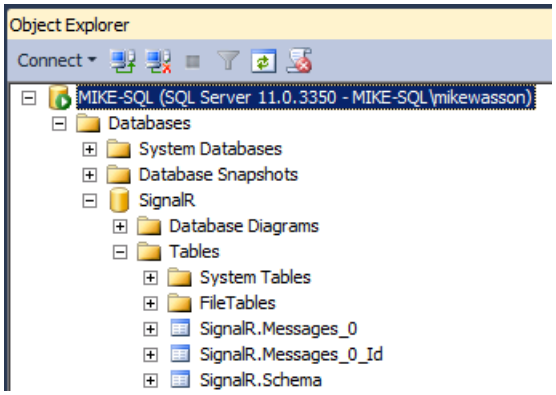
Now you are ready to deploy the Visual Studio project from your development machine to the server. In Solution Explorer, right-click the solution and click **Publish**.

For more detailed documentation about web deployment, see Web Deployment Content Map for Visual Studio and ASP.NET.

If you deploy the application to two servers, you can open each instance in a separate browser window and see that they each receive SignalR messages from the other. (Of course, in a production environment, the two servers would sit behind a load balancer.)



After you run the application, you can see that SignalR has automatically created tables in the database:



SignalR manages the tables. As long as your application is deployed, don't delete rows, modify the table, and so forth.

SignalR Troubleshooting (SignalR 1.x)

4/17/2020 • 13 minutes to read • Edit Online

by Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

This document describes common troubleshooting issues with SignalR.

This document contains the following sections.

- Calling methods between the client and server silently fails
- Other connection issues
- Compilation and server-side errors
- Visual Studio issues
- Internet Information Services issues
- Azure issues

Calling methods between the client and server silently fails

This section describes possible causes for a method call between client and server to fail without a meaningful error message. In a SignalR application, the server has no information about the methods that the client implements; when the server invokes a client method, the method name and parameter data are sent to the client, and the method is executed only if it exists in the format that the server specified. If no matching method is found on the client, nothing happens, and no error message is raised on the server.

To further investigate client methods not getting called, you can turn on logging before calling the start method on the hub to see what calls are coming from the server. To enable logging in a JavaScript application, see How to enable client-side logging (JavaScript client version). To enable logging in a .NET client application, see How to enable client-side logging (.NET Client version).

Misspelled method, incorrect method signature, or incorrect hub name

If the name or signature of a called method does not exactly match an appropriate method on the client, the call will fail. Verify that the method name called by the server matches the name of the method on the client. Also, SignalR creates the hub proxy using camel-cased methods, as is appropriate in JavaScript, so a method called `SendMessage` on the server would be called `sendMessage` in the client proxy. If you use the `HubName` attribute in your server-side code, verify that the name used matches the name used to create the hub on the client. If you do not use the `HubName` attribute, verify that the name of the hub in a JavaScript client is camel-cased, such as chatHub instead of ChatHub.

Duplicate method name on client

Verify that you do not have a duplicate method on the client that differs only by case. If your client application has a method called `sendMessage`, verify that there isn't also a method called `SendMessage` as well.

Missing JSON parser on the client

SignalR requires a JSON parser to be present to serialize calls between the server and the client. If your client doesn't have a built-in JSON parser (such as Internet Explorer 7), you'll need to include one in your application. You

can download the JSON parser here.

Mixing Hub and PersistentConnection syntax

SignalR uses two communication models: Hubs and PersistentConnections. The syntax for calling these two communication models is different in the client code. If you have added a hub in your server code, verify that all of your client code uses the proper hub syntax.

JavaScript client code that creates a PersistentConnection in a JavaScript client

```
var myConnection = $.connection('/echo');
```

JavaScript client code that creates a Hub Proxy in a Javascript client

```
var myHub = $.connection.MyHub;
```

C# server code that maps a route to a PersistentConnection

```
RouteTable.Routes.MapConnection<MyConnection>("my", "/echo");
```

C# server code that maps a route to a Hub, or to multiple hubs if you have multiple applications

```
RouteTable.Routes.MapHubs();
```

Connection started before subscriptions are added

If the Hub's connection is started before methods that can be called from the server are added to the proxy, messages will not be received. The following JavaScript code will not start the hub properly:

Incorrect JavaScript client code that will not allow Hubs messages to be received

```
var chat = $.connection.chatHub;
$.connection.hub.start().done(function () {
    chat.client.broadcastMessage = function (name, message) {...};
});
```

Instead, add the method subscriptions before calling Start:

JavaScript client code that correctly adds subscriptions to a hub

```
var chat = $.connection.chatHub;
chat.client.broadcastMessage = function (name, message) {...};
    $.connection.hub.start().done(function () {
        ...
    });
```

Missing method name on the hub proxy

Verify that the method defined on the server is subscribed to on the client. Even though the server defines the method, it must still be added to the client proxy. Methods can be added to the client proxy in the following ways (Note that the method is added to the `client` member of the hub, not the hub directly):

JavaScript client code that adds methods to a hub proxy

```
// Method added to proxy in JavaScript:
myHubProxy.server.method1 = function (param1, param2) {...};
//Multiple methods added to proxy in JavaScript using jQuery:
$.extend(myHubProxy.server, {
    method1: function (param1, param2) {...},
    method2: function (param3, param4) {...}
});
```

Hub or hub methods not declared as Public

To be visible on the client, the hub implementation and methods must be declared as `public`.

Accessing hub from a different application

SignalR Hubs can only be accessed through applications that implement SignalR clients. SignalR cannot interoperate with other communication libraries (like SOAP or WCF web services.) If there is no SignalR client available for your target platform, you cannot access the server's endpoint directly.

Manually serializing data

SignalR will automatically use JSON to serialize your method parameters- there's no need to do it yourself.

Remote Hub method not executed on client in OnDisconnected function

This behavior is by design. When `OnDisconnected` is called, the hub has already entered the `Disconnected` state, which does not allow further hub methods to be called.

C# server code that correctly executes code in the OnDisconnected event

```
public class MyHub : Hub
{
    public override Task OnDisconnected()
    {
        // Do what you want here
        return base.OnDisconnected();
    }
}
```

Connection limit reached

When using the full version of IIS on a client operating system like Windows 7, a 10-connection limit is imposed. When using a client OS, use IIS Express instead to avoid this limit.

Cross-domain connection not set up properly

If a cross-domain connection (a connection for which the SignalR URL is not in the same domain as the hosting page) is not set up correctly, the connection may fail without an error message. For information on how to enable cross-domain communication, see How to establish a cross-domain connection.

Connection using NTLM (Active Directory) not working in .NET client

A connection in a .NET client application that uses Domain security may fail if the connection is not configured properly. To use SignalR in a domain environment, set the requisite connection property as follows:

C# client code that implements connection credentials

```
connection.Credentials = CredentialCache.DefaultCredentials;
```

Other connection issues

This section describes the causes and solutions for specific symptoms or error messages that occur during a connection.

"Start must be called before data can be sent" error

This error is commonly seen if code references SignalR objects before the connection is started. The wireup for handlers and the like that will call methods defined on the server must be added after the connection completes. Note that the call to `Start` is asynchronous, so code after the call may be executed before it completes. The best way to add handlers after a connection starts completely is to put them into a callback function that is passed as a parameter to the start method:

JavaScript client code that correctly adds event handlers that reference SignalR objects

```
$.connection.hub.start().done(function () {
    // Wire up Send button to call NewContosoChatMessage on the server.
    $('#newContosoChatMessage').click(function () {
        contosoChatHubProxy.server.newContosoChatMessage(
            $('#displayname').val(), $('#message').val());
            $('#message').val('').focus();
    });
```

This error will also be seen if a connection stops while SignalR objects are still being referenced.

"301 Moved Permanently" or "302 Moved Temporarily" error

This error may be seen if the project contains a folder called SignalR, which will interfere with the automatically-created proxy. To avoid this error, do not use a folder called `SignalR` in your application, or turn automatic proxy generation off. See The Generated Proxy and what it does for you for more details.

"403 Forbidden" error in .NET or Silverlight client

This error may occur in cross-domain environments where cross-domain communication is not properly enabled. For information on how to enable cross-domain communication, see How to establish a cross-domain connection. To establish a cross-domain connection in a Silverlight client, see Cross-domain connections from Silverlight clients.

"404 Not Found" error

There are several causes for this issue. Verify all of the following:

- **Hub proxy address reference not formatted correctly:** This error is commonly seen if the reference to the generated hub proxy address is not formatted correctly. Verify that the reference to the hub address is made properly. See How to reference the dynamically generated proxy for details.
- **Adding routes to application before adding the hub route:** If your application uses other routes, verify that the first route added is the call to `MapHubs`.

"500 Internal Server Error"

This is a very generic error that could have a wide variety of causes. The details of the error should appear in the server's event log, or can be found through debugging the server. More detailed error information may be obtained by turning on detailed errors on the server. For more information, see How to handle errors in the Hub class.

"TypeError: <hubType> is undefined" error

This error will result if the call to `MapHubs` is not made properly. See How to register the SignalR route and configure SignalR options for more information.

JsonSerializationException was unhandled by user code

Verify that the parameters you send to your methods do not include non-serializable types (like file handles or database connections). If you need to use members on a server-side object that you don't want to be sent to the client (either for security or for reasons of serialization), use the `JSONIgnore` attribute.

"Protocol error: Unknown transport" error

This error may occur if the client does not support the transports that SignalR uses. See Transports and Fallbacks for information on which browsers can be used with SignalR.

"JavaScript Hub proxy generation has been disabled."

This error will occur if `DisableJavaScriptProxies` is set while also including a reference to the dynamically generated proxy at `signalr/hubs`. For more information on creating the proxy manually, see The generated proxy and what it does for you.

"The connection ID is in the incorrect format" or "The user identity cannot change during an active SignalR connection" error

This error may be seen if authentication is being used, and the client is logged out before the connection is stopped. The solution is to stop the SignalR connection before logging the client out.

"Uncaught Error: SignalR: jQuery not found. Please ensure jQuery is referenced before the SignalR.js file" error

The SignalR JavaScript client requires jQuery to run. Verify that your reference to jQuery is correct, that the path used is valid, and that the reference to jQuery is before the reference to SignalR.

"Uncaught TypeError: Cannot read property '<property>' of undefined" error

This error results from not having jQuery or the hubs proxy referenced properly. Verify that your reference to jQuery and the hubs proxy is correct, that the path used is valid, and that the reference to jQuery is before the reference to the hubs proxy. The default reference to the hubs proxy should look like the following:

HTML client-side code that correctly references the Hubs proxy

```
<script src="/signalr/hubs"></script>
```

"RuntimeBinderException was unhandled by user code" error

This error may occur when the incorrect overload of `Hub.On` is used. If the method has a return value, the return type must be specified as a generic type parameter:

Method defined on the client (without generated proxy)

```
MyHub.On<ReturnType>("MethodName", LocalMethod);
```

Connection ID is inconsistent or connection breaks between page loads

This behavior is by design. Since the hub object is hosted in the page object, the hub is destroyed when the page refreshes. A multi-page application needs to maintain the association between users and connection IDs so that they will be consistent between page loads. The connection IDs can be stored on the server in either a `ConcurrentDictionary` object or a database.

"Value cannot be null" error

Server-side methods with optional parameters are not currently supported; if the optional parameter is omitted, the method will fail. For more information, see Optional Parameters.

"Firefox can't establish a connection to the server at <address>" error in Firebug

This error message can be seen in Firebug if negotiation of the WebSocket transport fails and another transport is used instead. This behavior is by design.

"The remote certificate is invalid according to the validation procedure" error in .NET client application

If your server requires custom client certificates, then you can add an x509certificate to the connection before the request is made. Add the certificate to the connection using `Connection.AddClientCertificate`.

Connection drops after authentication times out

This behavior is by design. Authentication credentials cannot be modified while a connection is active; to refresh credentials, the connection must be stopped and restarted.

OnConnected gets called twice when using jQuery Mobile

jQuery Mobile's `initializePage` function forces the scripts in each page to be re-executed, thus creating a second connection. Solutions for this issue include:

- Include the reference to jQuery Mobile before your JavaScript file.
- Disable the `initializePage` function by setting `$.mobile.autoInitializePage = false`.
- Wait for the page to finish initializing before starting the connection.

Messages are delayed in Silverlight applications using Server Sent Events

Messages are delayed when using server sent events on Silverlight. To force long polling to be used instead, use the following when starting the connection:

```
connection.Start(new LongPollingTransport());
```

"Permission Denied" using Forever Frame protocol

This is a known issue, described here. This symptom may be seen using the latest JQuery library; the workaround is to downgrade your application to JQuery 1.8.2.

Compilation and server-side errors

The following section contains possible solutions to compiler and server-side runtime errors.

Reference to Hub instance is null

Since a hub instance is created for each connection, you can't create an instance of a hub in your code yourself. To call methods on a hub from outside the hub itself, see How to call client methods and manage groups from outside the Hub class for how to obtain a reference to the hub context.

HTTPContext.Current.Session is null

This behavior is by design. SignalR does not support the ASP.NET session state, since enabling the session state would break duplex messaging.

No suitable method to override

You may see this error if you are using code from older documentation or blogs. Verify that you are not referencing names of methods that have been changed or deprecated (like `OnConnectedAsync`).

HostContextExtensions.WebSocketServerUrl is null

This behavior is by design. This member is deprecated and should not be used.

"A route named 'signalr.hubs' is already in the route collection" error

This error will be seen if `MapHubs` is called twice by your application. Some example applications call `MapHubs` directly in the global application file; others make the call in a wrapper class. Ensure that your application does not do both.

Visual Studio issues

This section describes issues encountered in Visual Studio.

Script Documents node does not appear in Solution Explorer

Some of our tutorials direct you to the "Script Documents" node in Solution Explorer while debugging. This node is produced by the JavaScript debugger, and will only appear while debugging browser clients in Internet Explorer; the node will not appear if Chrome or Firefox are used. The JavaScript debugger will also not run if another client

debugger is running, such as the Silverlight debugger.

SignalR does not work on Visual Studio 2008 or earlier

This behavior is by design. SignalR requires .NET Framework 4 or later; this requires that SignalR applications be developed in Visual Studio 2010 or later.

IIS issues

This section contains issues with Internet Information Services.

Web site crashes after MapHubs call

This issue has been fixed in the latest version of SignalR. Verify that you are using the latest released version of SignalR by updating your installation using NuGet.

Azure issues

This section contains issues with Microsoft Azure.

Messages are not received through the Azure backplane after altering topic names

The topics used by the Azure backplane are not intended to be user-configurable.

Dependency Injection in SignalR 1.x

2/20/2020 • 8 minutes to read • Edit Online

by Mike Wasson, Patrick Fletcher

WARNING

This documentation isn't for the latest version of SignalR. Take a look at ASP.NET Core SignalR.

Dependency injection is a way to remove hard-coded dependencies between objects, making it easier to replace an object's dependencies, either for testing (using mock objects) or to change run-time behavior. This tutorial shows how to perform dependency injection on SignalR hubs. It also shows how to use IoC containers with SignalR. An IoC container is a general framework for dependency injection.

What is Dependency Injection?

Skip this section if you are already familiar with dependency injection.

Dependency injection (DI) is a pattern where objects are not responsible for creating their own dependencies. Here is a simple example to motivate DI. Suppose you have an object that needs to log messages. You might define a logging interface:

```
interface ILogger
{
    void LogMessage(string message);
}
```

In your object, you can create an `ILogger` to log messages:

```
// Without dependency injection.
class SomeComponent
{
    ILogger _logger = new FileLogger(@"C:\logs\log.txt");

    public void DoSomething()
    {
        _logger.LogMessage("DoSomething");
    }
}
```

This works, but it's not the best design. If you want to replace `FileLogger` with another `ILogger` implementation, you will have to modify `SomeComponent`. Supposing that a lot of other objects use `FileLogger`, you will need to change all of them. Or if you decide to make `FileLogger` a singleton, you'll also need to make changes throughout the application.

A better approach is to "inject" an `ILogger` into the object—for example, by using a constructor argument:

```
// With dependency injection.
class SomeComponent
{
    ILogger _logger;

    // Inject ILogger into the object.
    public SomeComponent(ILogger logger)
    {
        if (logger == null)
        {
            throw new NullReferenceException("logger");
        }
        _logger = logger;
    }

    public void DoSomething()
    {
        _logger.LogMessage("DoSomething");
    }
}
```

Now the object is not responsible for selecting which `ILogger` to use. You can switch `ILogger` implementations without changing the objects that depend on it.

```
var logger = new TraceLogger(@"C:\logs\log.etl");
var someComponent = new SomeComponent(logger);
```

This pattern is called constructor injection. Another pattern is setter injection, where you set the dependency through a setter method or property.

Simple Dependency Injection in SignalR

Consider the Chat application from the tutorial Getting Started with SignalR. Here is the hub class from that application:

```
public class ChatHub : Hub
{
    public void Send(string name, string message)
    {
        Clients.All.addMessage(name, message);
    }
}
```

Suppose that you want to store chat messages on the server before sending them. You might define an interface that abstracts this functionality, and use DI to inject the interface into the `ChatHub` class.

```
public interface IChatRepository
{
    void Add(string name, string message);
    // Other methods not shown.
}

public class ChatHub : Hub
{
    private IChatRepository _repository;

    public ChatHub(IChatRepository repository)
    {
        _repository = repository;
    }

    public void Send(string name, string message)
    {
        _repository.Add(name, message);
        Clients.All.addMessage(name, message);
    }
```

The only problem is that a SignalR application does not directly create hubs; SignalR creates them for you. By default, SignalR expects a hub class to have a parameterless constructor. However, you can easily register a function to create hub instances, and use this function to perform DI. Register the function by calling **GlobalHost.DependencyResolver.Register**.

```
protected void Application_Start()
{
    GlobalHost.DependencyResolver.Register(
        typeof(ChatHub),
        () => new ChatHub(new ChatMessageRepository()));

    RouteTable.Routes.MapHubs();

    // ...
}
```

Now SignalR will invoke this anonymous function whenever it needs to create a `ChatHub` instance.

IoC Containers

The previous code is fine for simple cases. But you still had to write this:

```
... new ChatHub(new ChatMessageRepository()) ...
```

In a complex application with many dependencies, you might need to write a lot of this "wiring" code. This code can be hard to maintain, especially if dependencies are nested. It is also hard to unit test.

One solution is to use an IoC container. An IoC container is a software component that is responsible for managing dependencies.You register types with the container, and then use the container to create objects. The container automatically figures out the dependency relations. Many IoC containers also allow you to control things like object lifetime and scope.

> **NOTE**
> "IoC" stands for "inversion of control", which is a general pattern where a framework calls into application code. An IoC container constructs your objects for you, which "inverts" the usual flow of control.

Using IoC Containers in SignalR

The Chat application is probably too simple to benefit from an IoC container. Instead, let's look at the StockTicker sample.

The StockTicker sample defines two main classes:

- `StockTickerHub`: The hub class, which manages client connections.
- `StockTicker`: A singleton that holds stock prices and periodically updates them.

`StockTickerHub` holds a reference to the `StockTicker` singleton, while `StockTicker` holds a reference to the **IHubConnectionContext** for the `StockTickerHub`. It uses this interface to communicate with `StockTickerHub` instances. (For more information, see Server Broadcast with ASP.NET SignalR.)

We can use an IoC container to untangle these dependencies a bit. First, let's simplify the `StockTickerHub` and `StockTicker` classes. In the following code, I've commented out the parts that we don't need.

Remove the parameterless constructor from `StockTicker`. Instead, we will always use DI to create the hub.

```
[HubName("stockTicker")]
public class StockTickerHub : Hub
{
    private readonly StockTicker _stockTicker;

    //public StockTickerHub() : this(StockTicker.Instance) { }

    public StockTickerHub(StockTicker stockTicker)
    {
        if (stockTicker == null)
        {
            throw new ArgumentNullException("stockTicker");
        }
        _stockTicker = stockTicker;
    }

    // ...
```

For StockTicker, remove the singleton instance. Later, we'll use the IoC container to control the StockTicker lifetime. Also, make the constructor public.

```
public class StockTicker
{
    //private readonly static Lazy<StockTicker> _instance = new Lazy<StockTicker>(
    //    () => new StockTicker(GlobalHost.ConnectionManager.GetHubContext<StockTickerHub>().Clients));

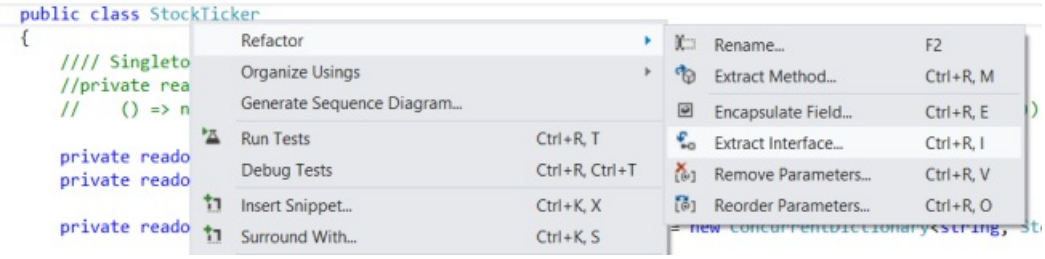
    // Important! Make this constructor public.
    public StockTicker(IHubConnectionContext clients)
    {
        if (clients == null)
        {
            throw new ArgumentNullException("clients");
        }

        Clients = clients;
        LoadDefaultStocks();
    }

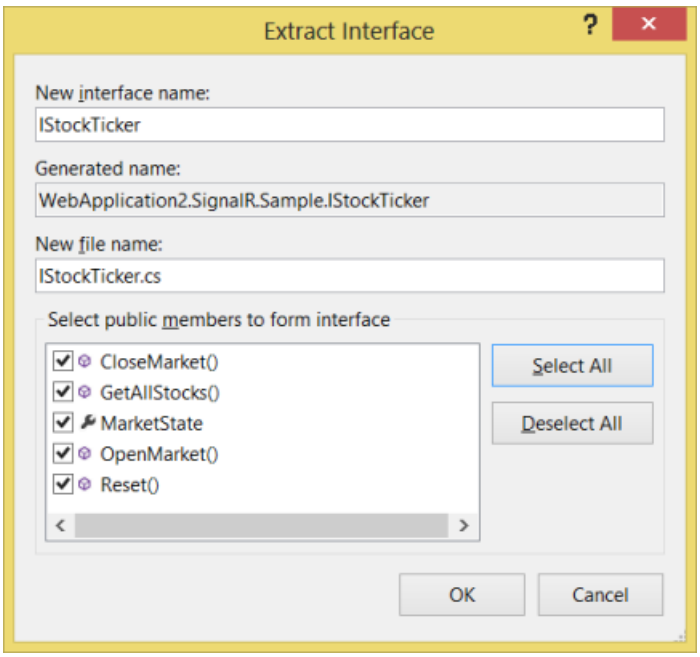
    //public static StockTicker Instance
    //{
    //    get
    //    {
    //        return _instance.Value;
    //    }
    //}
```

Next, we can refactor the code by creating an interface for `StockTicker`. We'll use this interface to decouple the `StockTickerHub` from the `StockTicker` class.

Visual Studio makes this kind of refactoring easy. Open the file StockTicker.cs, right-click on the `StockTicker` class declaration, and select **Refactor** ... **Extract Interface**.



In the **Extract Interface** dialog, click **Select All**. Leave the other defaults. Click **OK**.



Visual Studio creates a new interface named `IStockTicker`, and also changes `StockTicker` to derive from `IStockTicker`.

Open the file IStockTicker.cs and change the interface to **public**.

```
public interface IStockTicker
{
    void CloseMarket();
    IEnumerable<Stock> GetAllStocks();
    MarketState MarketState { get; }
    void OpenMarket();
    void Reset();
}
```

In the `StockTickerHub` class, change the two instances of `StockTicker` to `IStockTicker`:

```
[HubName("stockTicker")]
public class StockTickerHub : Hub
{
    private readonly IStockTicker _stockTicker;

    public StockTickerHub(IStockTicker stockTicker)
    {
        if (stockTicker == null)
        {
            throw new ArgumentNullException("stockTicker");
        }
        _stockTicker = stockTicker;
    }
```

Creating an `IStockTicker` interface isn't strictly necessary, but I wanted to show how DI can help to reduce coupling between components in your application.

Add the Ninject Library

There are many open-source IoC containers for .NET. For this tutorial, I'll use Ninject. (Other popular libraries include Castle Windsor, Spring.Net, Autofac, Unity, and StructureMap.)

Use NuGet Package Manager to install the Ninject library. In Visual Studio, from the **Tools** menu select **NuGet Package Manager** > **Package Manager Console**. In the Package Manager Console window, enter the following command:

```
Install-Package Ninject -Version 3.0.1.10
```

Replace the SignalR Dependency Resolver

To use Ninject within SignalR, create a class that derives from **DefaultDependencyResolver**.

```
internal class NinjectSignalRDependencyResolver : DefaultDependencyResolver
{
    private readonly IKernel _kernel;
    public NinjectSignalRDependencyResolver(IKernel kernel)
    {
        _kernel = kernel;
    }

    public override object GetService(Type serviceType)
    {
        return _kernel.TryGet(serviceType) ?? base.GetService(serviceType);
    }

    public override IEnumerable<object> GetServices(Type serviceType)
    {
        return _kernel.GetAll(serviceType).Concat(base.GetServices(serviceType));
    }
}
```

This class overrides the **GetService** and **GetServices** methods of **DefaultDependencyResolver**. SignalR calls these methods to create various objects at runtime, including hub instances, as well as various services used internally by SignalR.

- The **GetService** method creates a single instance of a type. Override this method to call the Ninject kernel's **TryGet** method. If that method returns null, fall back to the default resolver.
- The **GetServices** method creates a collection of objects of a specified type. Override this method to concatenate the results from Ninject with the results from the default resolver.

Configure Ninject Bindings

Now we'll use Ninject to declare type bindings.

Open the file RegisterHubs.cs. In the `RegisterHubs.Start` method, create the Ninject container, which Ninject calls the *kernel*.

```
var kernel = new StandardKernel();
```

Create an instance of our custom dependency resolver:

```
var resolver = new NinjectSignalRDependencyResolver(kernel);
```

Create a binding for `IStockTicker` as follows:

```
kernel.Bind<IStockTicker>()
    .To<Microsoft.AspNet.SignalR.StockTicker.StockTicker>()  // Bind to StockTicker.
    .InSingletonScope();  // Make it a singleton object.
```

This code is saying two things. First, whenever the application needs an `IStockTicker`, the kernel should create an instance of `StockTicker`. Second, the `StockTicker` class should be a created as a singleton object. Ninject will create one instance of the object, and return the same instance for each request.

Create a binding for **IHubConnectionContext** as follows:

```
kernel.Bind<IHubConnectionContext>().ToMethod(context =>
    resolver.Resolve<IConnectionManager>().GetHubContext<StockTickerHub>().Clients
).WhenInjectedInto<IStockTicker>();
```

This code creates an anonymous function that returns an **IHubConnection**. The **WhenInjectedInto** method tells Ninject to use this function only when creating `IStockTicker` instances. The reason is that SignalR creates **IHubConnectionContext** instances internally, and we don't want to override how SignalR creates them. This function only applies to our `StockTicker` class.

Pass the dependency resolver into the **MapHubs** method:

```
RouteTable.Routes.MapHubs(config);
```

Now SignalR will use the resolver specified in **MapHubs**, instead of the default resolver.

Here is the complete code listing for `RegisterHubs.Start`.

```
public static class RegisterHubs
{
    public static void Start()
    {
        var kernel = new StandardKernel();
        var resolver = new NinjectSignalRDependencyResolver(kernel);

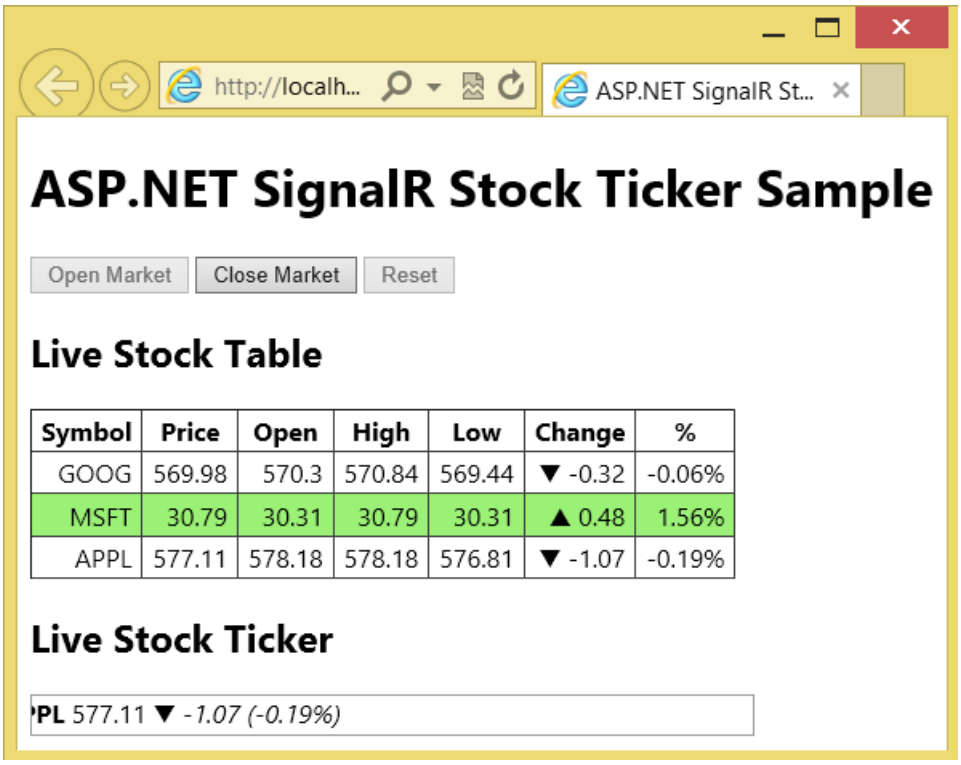
        kernel.Bind<IStockTicker>()
            .To<Microsoft.AspNet.SignalR.StockTicker.StockTicker>()
            .InSingletonScope();

        kernel.Bind<IHubConnectionContext>().ToMethod(context =>
                resolver.Resolve<IConnectionManager>().
                    GetHubContext<StockTickerHub>().Clients
            ).WhenInjectedInto<IStockTicker>();

        var config = new HubConfiguration()
        {
            Resolver = resolver
        };

        // Register the default hubs route: ~/signalr/hubs
        RouteTable.Routes.MapHubs(config);
    }
}
```

To run the StockTicker application in Visual Studio, press F5. In the browser window, navigate to `http://localhost:*port*/SignalR.Sample/StockTicker.html`.



The application has exactly the same functionality as before. (For a description, see Server Broadcast with ASP.NET SignalR.) We haven't changed the behavior; just made the code easier to test, maintain, and evolve.

ASP.NET SignalR Videos

2/20/2020 • 2 minutes to read • Edit Online

- ASP.NET SignalR Videos
  - SignalR and Web Sockets

ASP.NET SignalR Videos

2/20/2020 • 2 minutes to read • Edit Online

> These videos give you background and application examples for using SignalR to add real-time functionality to web applications.

- SignalR and Web Sockets

SignalR and Web Sockets

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Watch video (6 minutes)

Mobile Apps & Sites with ASP.NET

9/16/2020 • 2 minutes to read • Edit Online

Develop Mobile Apps

Cloud-Connected Mobile Apps

Part 1 - Create a Web Service with Azure Web Apps and WebJobs

In the first part of a two-part article, the authors explore some of the issues involved in creating a cloud back end that aggregates and processes data and serves it to mobile clients.

Part 2 - Build a Xamarin App with Authentication and Offline Support

In the second article in their series, the authors discuss how they use Xamarin to target multiple mobile client platforms, and show how to implement authentication and maintain a synchronized client-side cache of the data.

Calling Web API from a Windows Phone 8 Application

This end-to-end tutorial shows how to create an ASP.NET Web API application that provides data to a Windows Phone 8 application.

Get Started with Azure Mobile Services

Get started using Azure Mobile Services with ASP.NET Web API. This tutorial shows how to create a Mobile Services app from the Azure Management Portal.

Authentication in Azure Mobile Services

This tutorial shows how to authenticate users in Azure Mobile Services from your app.

Using Push Notifications in Azure Mobile Services

Use Azure Mobile Services to send push notifications to a mobile device.

Develop Mobile Sites

Deploy an Mobile-Friendly ASP.NET MVC 5 Web Application on Windows Azure Web Sites

This tutorial will teach you the basics of how to deploy an MVC 5 application to a Windows Azure web site and create mobile-friendly views using responsive Bootstrap design.

Add mobile pages to your ASP.NET Web Forms / MVC application

This How To describes various ways to serve pages optimized for mobile devices from your ASP.NET Web Forms / MVC application, and suggests architectural and design issues to consider when targeting a broad range of devices.

Detect the latest mobile devices using 51degrees.mobi Foundation

51degrees.mobi Foundation is a third-party open source project that enhances ASP.NET's Request.Browser API so you can easily detect recently-released mobile devices including Windows Phone 7, Android, and Apple iPad.

Mobile Web Sites with ASP.NET MVC and the Mobile Browser Definition File

In this talk presented at the Microsoft MIX conference, Scott Hanselman discusses how you can create mobile device friendly websites using ASP.NET MVC.

Develop iPhone applications with ASP.NET

ComponentOne Studio for iPhone is a suite of ASP.NET controls that mimic the look and feel of the iPhone and iPod

touch user interfaces when rendered in Mobile Safari.

ASP.NET Mobile Controls Tested Devices

2/20/2020 • 11 minutes to read • Edit Online

The following devices and browsers have been tested with the Microsoft Mobile Internet Toolkit v1.0 and ASP.NET mobile controls.

- Microsoft Mobile Internet Toolkit [MMIT 1.0]: *(January 15, 2002)*
- Device Update 1 [DU 1.0]: *(May 14, 2002)*
- Device Update 2 [DU 2.0]: *(October 28, 2002)*
- Device Update 3 [DU 3.0]: *(August 5, 2003)*
- Device Update 4 [DU 4.0]: *(December 18, 2003)*

MANUFACTURER	MODEL	BROWSER MANUFACTURER	BROWSER MODEL	RELEASED
@Migo	PD-600C	Microsoft	Pocket IE 2002 (4.1)	DU 1.0
Alcatel	One Touch 311	Openwave	UP.Browser 4.1.23a	DU 1.0
Alcatel	One Touch 501	Openwave	UP.Browser 4.1.19e	DU 1.0
Alcatel	One Touch 511	Openwave	UP.Browser 4.1.23a	DU 1.0
Audiovox	Maestro HandHeld Pocket PC	Microsoft	Pocket IE 2002 (4.1)	DU 1.0
Casio	BE-500	Original Equipment Manufacturer's	Version 1.00.62(J) (HTML)	DU 1.0
Compaq	Ipaq H3870	Microsoft	Pocket IE 2002 (4.1)	DU 1.0
Ericsson	T20e/ec	Ericsson	T20/R2A	DU 1.0
Ericsson	T29s/sc	Ericsson	T20/R2A	DU 1.0
Ericsson	T39m/mc	Ericsson	T39/R202	DU 1.0
Ericsson	T65	Ericsson	T65/R101	DU 1.0
Fujitsu	F211i	ACCESS	Compact NetFront 2.0	DU 1.0
Fujitsu	F671i	ACCESS	Compact NetFront 2.0	DU 1.0
Handspring	Treo 180	Blazer	2.0	DU 1.0
Handspring	Visor Platinum	Blazer	2.0	DU 1.0
HP	Jornada 567	Microsoft	Pocket IE 2002 (4.1)	DU 1.0

MANUFACTURER	MODEL	BROWSER MANUFACTURER	BROWSER MODEL	RELEASED
Mitsubishi	D211i	ACCESS	Compact NetFront 2.0	DU 1.0
Mitsubishi	D503is	ACCESS	Compact NetFront 2.0	DU 1.0
Motorola	i80s	Openwave	UP.Browser 4.1.23	DU 1.0
Motorola	i90c	Openwave	UP.Browser 4.1.23	DU 1.0
Motorola	Talkabout 191	Openwave	UP.Browser 4.1	DU 1.0
Motorola	Timeport 270c	Openwave	UP.Browser 4.1.21b	DU 1.0
Motorola	Timeport T260	Openwave	UP.Browser 4.1.8	DU 1.0
Motorola	V200	Openwave	UP.Browser 4.1	DU 1.0
Motorola	V50	Openwave	UP.Browser 4.1.16s	DU 1.0
NEC	FOMA N2001	ACCESS	Compact NetFront 2.0	DU 1.0
NEC	N211i	ACCESS	Compact NetFront 2.0	DU 1.0
NEC	N503i	ACCESS	Compact NetFront 2.0	DU 1.0
NEC	N503iS	ACCESS	Compact NetFront 2.0	DU 1.0
Nokia	3285	Openwave	UP.Browser 3.1.11A.1	DU 1.0
Nokia	6310	Nokia	1.0 (4.10)	DU 1.0
Nokia	8310	Nokia	1.0 (3.07)	DU 1.0
Nokia	9110i	Nokia	1.1 (HTML)	DU 1.0
Nokia	9210	Symbian	Crystal 6.0 (HTML)	DU 1.0
Nokia	Mobile Internet Toolkit 3.0 with 3330/3395 Skin	Nokia	Mobile Browser 3.05	DU 1.0
Nokia	Mobile Internet Toolkit 3.0 with Default Skin	Nokia	Mobile Browser 3.0	DU 1.0
Openwave	5.0 emulator	Openwave	UP.Browser 5.0	DU 1.0
Palm	m505	Blazer	2.0	DU 1.0
Palm	VIIx	AvantGo	4.0	DU 1.0

MANUFACTURER	MODEL	BROWSER MANUFACTURER	BROWSER MODEL	RELEASED
Palm	Vx	GoAmerica	Go.Web 6.0	DU 1.0
Panasonic	GD95	Original Equipment Manufacturer's	Version	DU 1.0
Panasonic	P503i	Original Equipment Manufacturer's	Version	DU 1.0
Panasonic	P503is	Original Equipment Manufacturer's	Version	DU 1.0
Panasonic	P821i	ACCESS	Compact NetFront 2.0	DU 1.0
Samsung	SGH-A300	Openwave	UP.Browser 4.1.19k	DU 1.0
Samsung	SGH-A400	Openwave	UP.Browser 4.1.19k	DU 1.0
Samsung	SPH-i300	Openwave	UP.Browser 4.1.22c1	DU 1.0
Samsung	SPH-N200	Openwave	UP.Browser 4.1.22b	DU 1.0
Sanyo	C1001SA	Openwave	UP.Browser 3.2	DU 1.0
Sanyo	SCP-4700	Openwave	UP.Browser 4.1.24f	DU 1.0
Sanyo	SCP-6000	Openwave	UP.Browser 4.1.23a	DU 1.0
Sharp	J-SH07	Original Equipment Manufacturer's	Version 3.0	DU 1.0
Sharp	SH821i	ACCESS	Compact NetFront 2.0	DU 1.0
Sharp	Zaurus MI-E21	Original Equipment Manufacturer's	Version 6.2	DU 1.0
Siemens	M35	Openwave	UP.Browser 4.1.16m	DU 1.0
Siemens	ME45	Openwave	UP.Browser 5.0.1.1.102 (GUI)	DU 1.0
Siemens	S40	Openwave	UP.Browser 4.1.16	DU 1.0
Sony	C1002S	Openwave	UP.Browser 3.2	DU 1.0
Sony	PEG T600C	ilinx	Xiino 1.0.6a4SJ	DU 1.0
Sony	SO210i	ACCESS	Compact NetFront 2.0	DU 1.0
Sony	SO503iS	ACCESS	Compact NetFront 2.0	DU 1.0
Sprint LG	TP5200	Openwave	UP.Browser 4.1.22b	DU 1.0

MANUFACTURER	MODEL	BROWSER MANUFACTURER	BROWSER MODEL	RELEASED
Toshiba	Genio e550x	Microsoft	Pocket IE 2002 (4.1) ja	DU 1.0
Toshiba	J-T06	Original Equipment Manufacturer's	Version 3.0	DU 1.0
Casio	A3012CA	Openwave	UP.Browser 6.0	DU 2.0
Casio	Cassiopeia E-2000	Microsoft	Pocket Internet Explorer 2002 (4.1)	DU 2.0
Compaq	iPAQ H3650	Jataayu	Jbrowser 3.0	DU 2.0
Compaq	iPAQ H3870	EZOS	EzWap 2.5 (beta 2)	DU 2.0
DoCoMo	Sigmarion II	Microsoft	Internet Explorer 4.0	DU 2.0
Ericsson	T39	Ericsson	T39/R201	DU 2.0
Ericsson	T68	Ericsson	T68/R101	DU 2.0
Ericsson	T68i	SonyEricsson	T68/R201 A	DU 2.0
Ericsson	WAP Toolkit 3.2	Ericsson	WAP IDE 3.2.1	DU 2.0
Fujitsu	F504i	ACCESS	Compact NetFront 2.0	DU 2.0
Handspring	Treo 270	Handspring	Blazer 2.1	DU 2.0
Hitachi	NPD-10JWL	Microsoft	Internet Explorer 5.5 for Windows CE .NET	DU 2.0
HP	Jornada 568	Microsoft	Pocket Internet Explorer 2002 (4.1)	DU 2.0
Jataayu	WAP2 Toolkit with Default skin	Jataayu	J-Browser (beta)	DU 2.0
Legend	TianJi X100	Microsoft	Pocket Internet Explorer 2002 (4.1)	DU 2.0
LG	LP9000	Openwave	UP.Browser 4.1.21c	DU 2.0
Microsoft	Smartphone Emulator for Windows CE	Microsoft	Pocket Internet Explorer 2002 (4.1)	DU 2.0
Motorola	Accompli 008/6288	Openwave	UP.Browser 4.1.23i	DU 2.0
Motorola	Talkabout 192	Openwave	UP.Browser 4.1.23c	DU 2.0
NEC	FOMA N2002	ACCESS	Compact NetFront 2.0	DU 2.0

MANUFACTURER	MODEL	BROWSER MANUFACTURER	BROWSER MODEL	RELEASED
NEC	N504i	ACCESS	Compact NetFront 2.0	DU 2.0
Nokia	6510	Nokia	1.0 (4.0)	DU 2.0
Nokia	Mobile Internet Toolkit 3.1 with Default skin	Nokia	Mobile Browser 3.0.1	DU 2.0
OKWap	i108	Openwave	UP.Browser 4.1.23c	DU 2.0
Openwave	Universal Edition 1.1 with default skin	Openwave	UP.Browser 6.0	DU 2.0
Panasonic	P211i	Original Equipment Manufacturer's	Version 2.0	DU 2.0
Panasonic	P504i	Original Equipment Manufacturer's	Version 2.0	DU 2.0
Psion	Series 7	Symbian	EPOC Web 2.0	DU 2.0
RIM	Blackberry 957	GoAmerica	Go.Web 6.2.023. (HTML)	DU 2.0
Samsung	Nexio S150	Microsoft	Internet Explorer 4.0 (for WinCE 3.0)	DU 2.0
Sanyo	A3011SA	Openwave	UP.Browser 6.0	DU 2.0
Sharp	J-SH08	Original Equipment Manufacturer's	Version 3.0	DU 2.0
Sharp	J-SH51	Original Equipment Manufacturer's	Version 4.0	DU 2.0
Sharp	SH251i	ACCESS	Compact Netfront 2.0	DU 2.0
Sharp	Zaurus MI-E25DC	Original Equipment Manufacturer's	Version 6.2	DU 2.0
Sharp	Zaurus SL 5500	Opera	Ver 5.0	DU 2.0
Siemens	S45	Openwave	UP.Browser 5.0.1.1.102 (GUI).	DU 2.0
Sony	CLIE PEG-NR70V	Ilinx	Xiino 2.0SJ	DU 2.0
Sony	CLIE PEG-T650C	Ilinx	Xiino 2.0SJ	DU 2.0
T-Mobil	Pocket PC Phone	Microsoft	Internet Explorer 2002 (4.1)	DU 2.0

MANUFACTURER	MODEL	BROWSER MANUFACTURER	BROWSER MODEL	RELEASED
Toshiba	C5001T	Openwave	UP.Browser 6.0	DU 2.0
Toshiba	Genio e550G	Microsoft	Internet Explorer 2002 (4.1)	DU 2.0
Toshiba	J-T51	Original Equipment Manufacturer's	Version 4.0	DU 2.0
Casio	Casio A5302CA	Openwave	UP.Browser 6.0	DU 3.0
Cyberbank	POZ	Microsoft	Pocket IE 2002 (4.1)	DU 3.0
Docomo	Musea	Microsoft	Pocket IE 2002 (4.1)	DU 3.0
DOPOD	686	Microsoft	Pocket IE 2002 (4.1)	DU 3.0
Fujitsu	F212i	Access	Compact NetFront 2.0	DU 3.0
Hitachi	SH-P300	Openwave	UP.Browser 6.1.0.135	DU 3.0
JTEL	Cellvic XG	NATE Browser	v2.2	DU 3.0
LEGEND	G808	AU-Systems	V4.2	DU 3.0
LG	LX5350	Openwave	UP.Browser 6.1.B.292	DU 3.0
Motorola	E360	Openwave	UP.Browser 5.0.1.7	DU 3.0
Motorola	V-730	AU-Systems	V2.5	DU 3.0
Motorola	V60G	Openwave	UP.Browser4.1.23c	DU 3.0
Motorola	V70	Openwave	UP.Browser 4.1.25i	DU 3.0
NEC	N251i	Access	Compact NetFront 2.0	DU 3.0
NEC	N504iS	Access	Compact NetFront 2.0	DU 3.0
Nokia	6590	Nokia	1.0 (30.34)	DU 3.0
Nokia	7650	Nokia	1.2.1	DU 3.0
OpenWave	6.1 Emulator	OpenWave	UP.Browser 6.1	DU 3.0
Orange	SmartPhone 2002	Microsoft	Pocket IE 2002 (4.1)	DU 3.0
Palm	Tungsten T	Palm	WebPro v 1.0	DU 3.0
Panasonic	P504iS	Panasonic Proprietary	2.0	DU 3.0

MANUFACTURER	MODEL	BROWSER MANUFACTURER	BROWSER MODEL	RELEASED
Philips	Fisio 820	Openwave	UP.Browser 5.0.1.11	DU 3.0
Sanyo	J-SA51	Original Equipment Manufacturer's	Version 4.1	DU 3.0
Sharp	J-SH52	Original Equipment Manufacturer's	Version 4.1	DU 3.0
Sharp	SL-B500	Access	NetFront 3.0	DU 3.0
Siemens	S46	Openwave	UP.Browser 5.0.2.1.103	DU 3.0
SonyEricsson	SonyEricsson A1101S	Openwave	UP.Browser 6.0	DU 3.0
SonyEricsson	SonyEricsson P800	SonyEricsson	P800/R101	DU 3.0
SonyEricsson	SonyEricsson T300	SonyEricsson	T300/R101	DU 3.0
SonyEricsson	T68i	SonyEricsson	T68/R301A	DU 3.0
T-Mobile	SideKick	AvantGo	3.2	DU 3.0
Toshiba	Genio e550GX	Microsoft	Pocket IE 2002 (4.1)	DU 3.0
Toshiba	J-T08	Original Equipment Manufacturer's	Version 3.0	DU 3.0
BenQ	S830c	Openwave	UP.Browser 6.1.0.5	DU 4.0
Compaq	Ipaq3650	Microsoft	PIE 2003 (MSIE 4.01)	DU 4.0
Mitsubishi	D505i	ACCESS	Compact NetFront 2.0	DU 4.0
NEC	J-N51	Original Equipment Manufacturer's	Version 4.2	DU 4.0
NEC	N251iS	ACCESS	Compact NetFront 2.0	DU 4.0
Nokia	3590	Nokia	1.0(7.03)	DU 4.0
Nokia	3595	Nokia	1.0 (7.01)	DU 4.0
Nokia	3650	Nokia	1.2.12X	DU 4.0
Nokia	5100	Nokia	1.2.1	DU 4.0
Nokia	6200	Nokia	1.0 (3.05)	DU 4.0
Nokia	6800	Nokia	2.0	DU 4.0

MANUFACTURER	MODEL	BROWSER MANUFACTURER	BROWSER MODEL	RELEASED
NTT DoCoMo	i-mode HTML Simulator	NTT DoCoMo	i-mode HTML Simulator Version 6.0	DU 4.0
NTT DoCoMo	Sigmarion III	Microsoft	IE 5.5 (for CE.NET)	DU 4.0
Openwave	6.2	Openwave	6.2	DU 4.0
Panasonic	GD87	OEM	Panasonic-GAD87/A39	DU 4.0
Panasonic	GU87	Panasonic	GAD87/A38	DU 4.0
Pantech&Curitel	PS-E200	Infraware	1.0	DU 4.0
Samsung	SCH-E150	Infraware	1.0	DU 4.0
Samsung	SPH-i330	Blazer	V2.1.1s2 Build 31	DU 4.0
Sharp	GX10	Openwave	UP.Browser v6.1	DU 4.0
Sharp	J-SH53	Original Equipment Manufacturer's	Version 4.2	DU 4.0
Sharp	SH251iS	Original Equipment Manufacturer	NA	DU 4.0
Sharp	SL 5600	Opera	6.0	DU 4.0
Siemens	S55	Openwave	UP.Browser 6.1.0.5.121	DU 4.0
Sony	CLIE PEG-TG50	ACCESS	NetFront 3.0	DU 4.0
Sony Ericsson	Sony Ericsson T610	Sony Ericsson	T610/R101	DU 4.0
SonyEricsson	SO505i	Original Equipment Manufacturer	NA	DU 4.0
SonyEricsson	SonyEricsson T310	SonyEricsson	T310/R201	DU 4.0
TCL	L668	Openwave	UP.Browser 5.0.3.2	DU 4.0
Toshiba	A5304T	Openwave	UP.Browser 6.0	DU 4.0
Toshiba	Genio e550C	Microsoft	Pocket IE 2002 (4.1) ja	DU 4.0
Alcatel	One Touch 701	Openwave	UP.Browser 4.1.19	MIT 1.0
Audiovox	CDM-135	Openwave	UP.Browser 4.1.2	MIT 1.0
Audiovox	CDM-9000	Openwave	UP.Browser 3.1.11	MIT 1.0

MANUFACTURER	MODEL	BROWSER MANUFACTURER	BROWSER MODEL	RELEASED
Audiovox	CDM-9100	Openwave	UP.Browser 4.1.21c	MIT 1.0
Benefon	Q	Microsoft	Mobile Explorer 2.01	MIT 1.0
Casio	Cassiopeia E-125	Microsoft	Pocket IE 2000 (4.01)	MIT 1.0
Compaq	iPaq H3630	Microsoft	Pocket IE 2000 (4.01)	MIT 1.0
Compaq	iPaq H3650	GoAmerica	Go.Web 5.0.1	MIT 1.0
Compaq	iPaq H3650	Microsoft	Pocket IE 2000 (4.01)	MIT 1.0
Compaq	iPaq H3650	Omnisky	2.0	MIT 1.0
Compaq	iPaq H3670	Microsoft	Pocket IE 2002 (4.1)	MIT 1.0
Ericsson	R280LX	Openwave	UP.Browser 3.03	MIT 1.0
Ericsson	R320	Ericsson	R320/R1A	MIT 1.0
Ericsson	R380	Ericsson	R380 2.0	MIT 1.0
Ericsson	R380 Emulator	Ericsson	R380 2.0	MIT 1.0
Ericsson	R520	Ericsson	R520/R201	MIT 1.0
Ericsson	T20s	Ericsson	T20/R2A	MIT 1.0
Fujitsu	F503i	ACCESS	Compact NetFront 2.0	MIT 1.0
Handspring	Visor Platinum	Blazer	1.0	MIT 1.0
Handspring	Visor Platinum	Qualcomm	Euora Internet Suite 2.1	MIT 1.0
Handspring	Visor Prism	Omnisky	2.1.0.15	MIT 1.0
Hitachi	C407H	Openwave	UP.Browser 3.1	MIT 1.0
HP	Jornada 720	Microsoft	Pocket IE 2000 (4.01)	MIT 1.0
IBM	WorkPad C505	ilinx	Xiino 1.01J	MIT 1.0
Jungle	DoCoMo 502 Em.	i-page master	1.0 (502i)	MIT 1.0
Kyocera	QCP 2035A	Openwave	UP.Browser 3.1.17	MIT 1.0
Kyocera	QCP 3035	Openwave	UP.Browser 3.1.17	MIT 1.0
Kyocera	QCP 6035	Qualcomm	Eudora 2.0	MIT 1.0

MANUFACTURER	MODEL	BROWSER MANUFACTURER	BROWSER MODEL	RELEASED
LG	V111	Openwave	UP.Browser 3.1	MIT 1.0
Microsoft	Emulator	Microsoft	Mobile Explorer 2.01	MIT 1.0
Microsoft	Emulator	Microsoft	Mobile Explorer 3.0 (HTML)	MIT 1.0
Microsoft	Emulator	Microsoft	Pocket IE 2000 (4.01)	MIT 1.0
Mitsubishi	D502i	ACCESS	Compact NetFront 2.0	MIT 1.0
Mitsubishi	D503i	ACCESS	Compact NetFront 2.0	MIT 1.0
Mitsubishi	T250	Openwave	UP.Browser 3.1.11A.1	MIT 1.0
Motorola	i1000 plus	Openwave	UP.Browser 4.1.17	MIT 1.0
Motorola	i2000 plus	Openwave	UP.Browser 4.1.17	MIT 1.0
Motorola	i50sx	Openwave	UP.Browser 4.1.21b	MIT 1.0
Motorola	i85s	Openwave	UP.Browser 4.1.21b	MIT 1.0
Motorola	StarTAC 7868W	Openwave	UP.Browser 3.1.02	MIT 1.0
Motorola	T2288	Openwave	UP.Browser 4.1.8	MIT 1.0
Motorola	TimePort P7382i	Openwave	UP.Browser 4.1.19i	MIT 1.0
Motorola	TimePort P7389	Openwave	UP.Browser 4.1.16g	MIT 1.0
Motorola	TimePort P8767	Openwave	UP.Browser 3.1.10	MIT 1.0
Motorola	V100	Openwave	UP.Browser 4.1.22	MIT 1.0
Motorola	V120c	Openwave	UP.Browser 4.1.21b	MIT 1.0
Motorola	V2288	Openwave	UP.Browser 4.1.16f	MIT 1.0
Motorola	V60c	Openwave	UP.Browser 4.1.21b	MIT 1.0
NEC	N210i	ACCESS	Compact NetFront 2.0	MIT 1.0
NEC	N502i	ACCESS	Compact NetFront 2.0	MIT 1.0
NetFarm	JPhone Em.	J-Sky Viewer	1.2	MIT 1.0
Nokia	3330	Nokia	1.0 (3.12)	MIT 1.0
Nokia	6210	Nokia	1.0 (4.27)	MIT 1.0

MANUFACTURER	MODEL	BROWSER MANUFACTURER	BROWSER MODEL	RELEASED
Nokia	7110	Nokia	1.0 (5.01)	MIT 1.0
Nokia	9110i	Nokia	1.1	MIT 1.0
Nokia	9210	Symbian	Crystal 6.0	MIT 1.0
Openwave	Emulator for KDDI phones	Openwave	UP.Browser 3.3.1	MIT 1.0
Openwave	UP 3.2 Emulator with Mitsubishi T250 Skin	Openwave	UP.Browser 3.2	MIT 1.0
Openwave	UP 4.1 Emulator with Default skin	Openwave	UP.Browser 4.1	MIT 1.0
Palm	m505	Palm	MyPalm 1.1	MIT 1.0
Palm	VIIx	Palm	MyPalm 1.0	MIT 1.0
Palm	Vx	AU-Systems	2.12181.1	MIT 1.0
Palm	Vx	GoAmerica	Go.Web 3.0.6	MIT 1.0
Palm	Vx	Omnisky	2.0.0.4	MIT 1.0
Panasonic	P210i	Original Equipment Manufacturer's	Version	MIT 1.0
Panasonic	P502i Phone	Original Equipment Manufacturer's	Version	MIT 1.0
Personal	Computer	Microsoft	Internet Explorer 5.5	MIT 1.0
Personal	Computer	Microsoft	Internet Explorer 6.0	MIT 1.0
RIM	Blackberry 857	GoAmerica	Go.Web 1.8.004 (HTML)	MIT 1.0
RIM	Blackberry 857	GoAmerica	Go.Web 6.0.010 (HTML)	MIT 1.0
RIM	Blackberry 950	GoAmerica	Go.Web 1.000.6 (HTML)	MIT 1.0
RIM	Blackberry 950	GoAmerica	Go.Web 6.010.0 (HTML)	MIT 1.0
RIM	Blackberry 957	GoAmerica	Go.Web 1.6.000 (HTML)	MIT 1.0

MANUFACTURER	MODEL	BROWSER MANUFACTURER	BROWSER MODEL	RELEASED
RIM	Blackberry 957	GoAmerica	Go.Web 6.0.10.0 (HTML)	MIT 1.0
Samsung	SCH-6100	Openwave	UP.Browser 3.1	MIT 1.0
Samsung	SCH-850	Openwave	UP.Browser 3.1	MIT 1.0
Samsung	SCH-8500	Openwave	UP.Browser 3.1.140.TC14	MIT 1.0
Samsung	UpRoar M100	Openwave	UP.Browser 3.1	MIT 1.0
Sanyo	C401SA	Openwave	UP.Browser 3.2	MIT 1.0
Sanyo	SCP-4500	Openwave	UP.Browser 3.1.04	MIT 1.0
Sanyo	SCP-5000	Openwave	UP.Browser 3.2	MIT 1.0
Sharp	J-SH04	Original Equipment Manufacturer's	Version 3.0	MIT 1.0
Sharp	Zaurus MI-E1	Original Equipment Manufacturer's	Version 6.1	MIT 1.0
Siemens	C35i	Openwave	UP.Browser 4.1.8c	MIT 1.0
Siemens	S35i	Openwave	UP.Browser 4.1.16	MIT 1.0
Siemens	SL45	Openwave	UP.Browser 4.1	MIT 1.0
Sony	CLIE PEG-N700C	ilinx	Palmscape 4.0SJ	MIT 1.0
Sony	CMD-J5	Microsoft	Mobile Explorer 2.0 r54ue	MIT 1.0
Sony	CMD-z5	Microsoft	Mobile Explorer 2.0 r54ue	MIT 1.0
Sony	SO503i	ACCESS	Compact NetFront 2.0	MIT 1.0
Sprint Denso	Touchpoint 120	Openwave	UP.Browser 3.1	MIT 1.0
Sprint Denso	Touchpoint 2200	Openwave	UP.Browser 3.1	MIT 1.0
Sprint LG	TP3000	Openwave	UP.Browser 3.1	MIT 1.0
Toshiba	J-T05	Original Equipment Manufacturer's	Version 3.0	MIT 1.0

Device Count: 265

ASP.NET Single Page Application

2/20/2020 • 2 minutes to read • Edit Online

- Overview
  - Introduction
    - Single Page Application: KnockoutJS template
    - Know a library other than Knockout?
  - Templates
    - Backbone Template
    - Breeze/Angular template
    - Breeze/Knockout template
    - EmberJS template
    - Hot Towel template

Single Page Application Overview

2/20/2020 • 2 minutes to read • Edit Online

- Introduction
  - Single Page Application: KnockoutJS template
  - Know a library other than Knockout?
- Templates
  - Backbone Template
  - Breeze/Angular template
  - Breeze/Knockout template
  - EmberJS template
  - Hot Towel template

Introduction to ASP.NET Single Page Application

2/20/2020 • 2 minutes to read • Edit Online

Content for using the new SPA template and Knockoutjs.

- Single Page Application: KnockoutJS template
- Know a library other than Knockout?

Single Page Application: KnockoutJS template

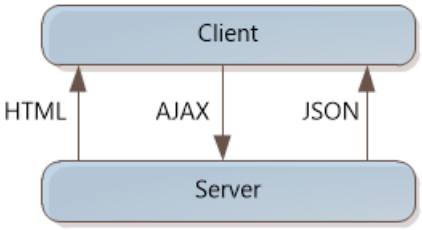
2/20/2020 • 12 minutes to read • Edit Online

by Mike Wasson

> The Knockout MVC Template is part of ASP.NET and Web Tools 2012.2
>
> Download ASP.NET and Web Tools 2012.2

The ASP.NET and Web Tools 2012.2 update includes a Single-Page Application (SPA) template for ASP.NET MVC 4. This template is designed to get you started quickly building interactive client-side web apps.

"Single-page application" (SPA) is the general term for a web application that loads a single HTML page and then updates the page dynamically, instead of loading new pages. After the initial page load, the SPA talks with the server through AJAX requests.



AJAX is nothing new, but today there are JavaScript frameworks that make it easier to build and maintain a large sophisticated SPA application. Also, HTML 5 and CSS3 are making it easier to create rich UIs.

To get you started, the SPA template creates a sample "To-do list" application. In this tutorial, we'll take a guided tour of the template. First we'll look at the To-do list application itself, then examine the technology pieces that make it work.

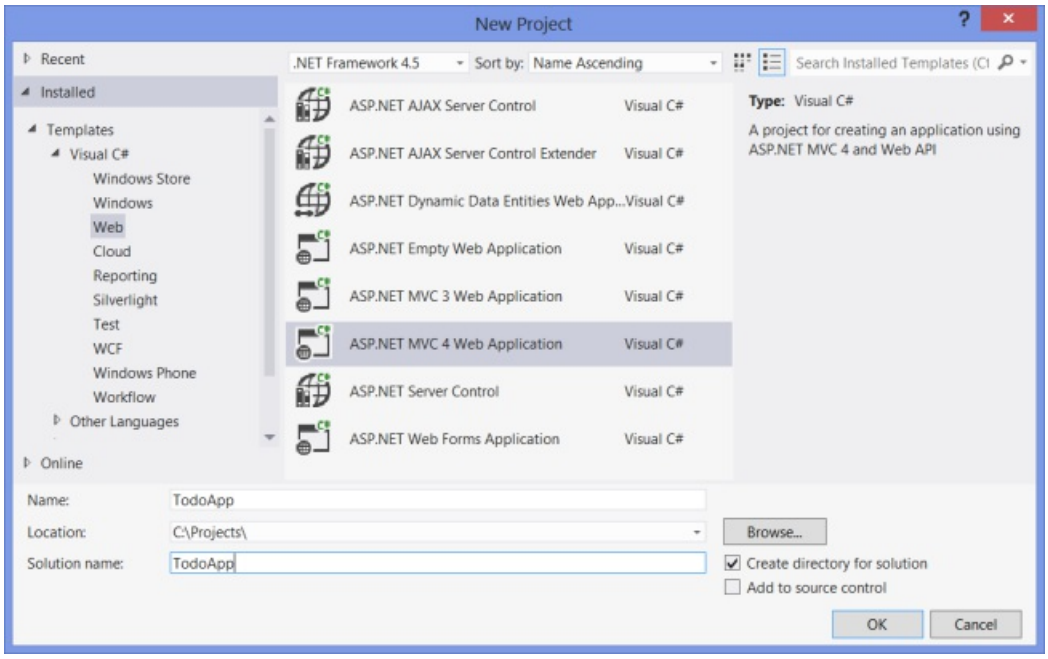
Create a New SPA Template Project

Requirements:

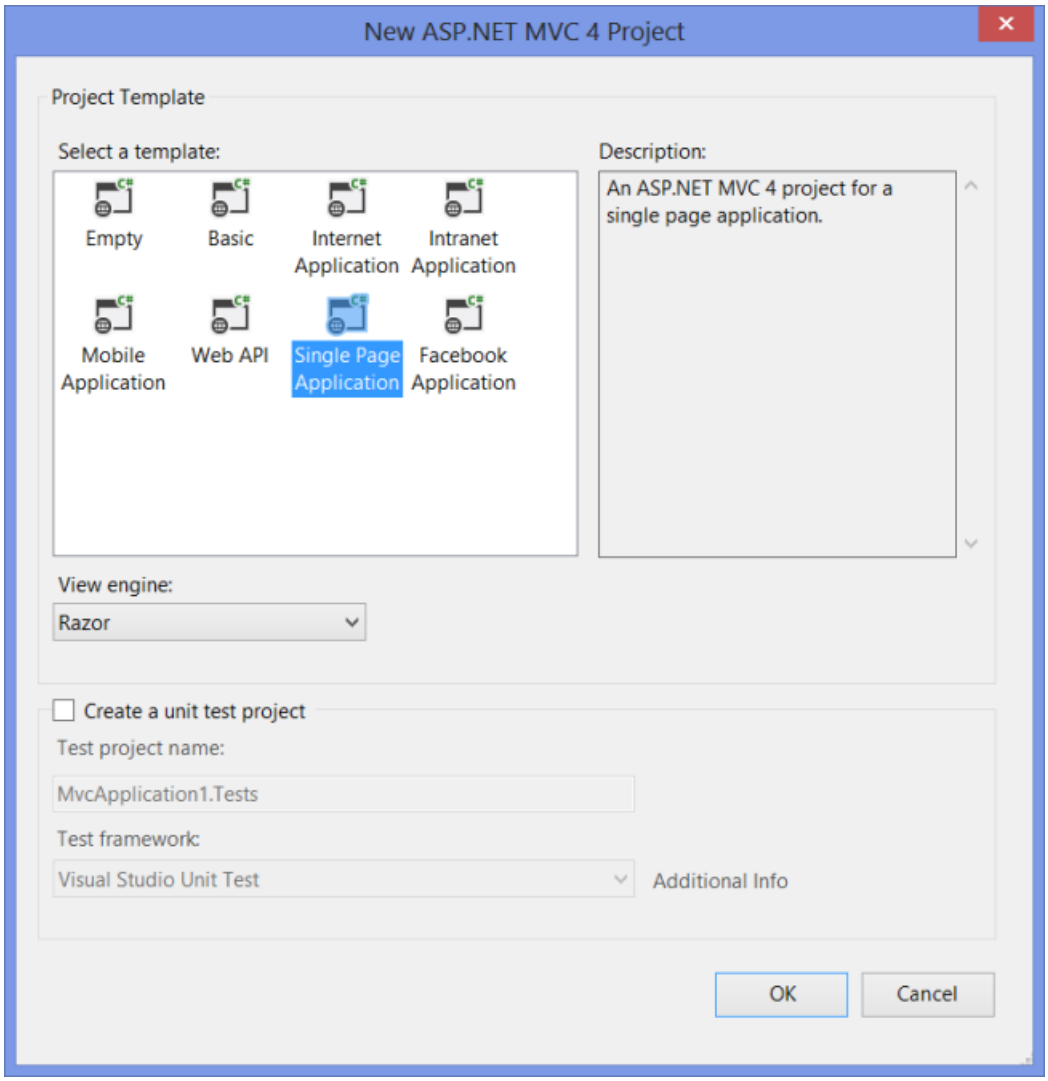
- Visual Studio 2012 or Visual Studio Express 2012 for Web
- ASP.NET Web Tools 2012.2 update. You can install the update here.

Start Visual Studio and select **New Project** from the Start page. Or, from the **File** menu, select **New** and then **Project**.

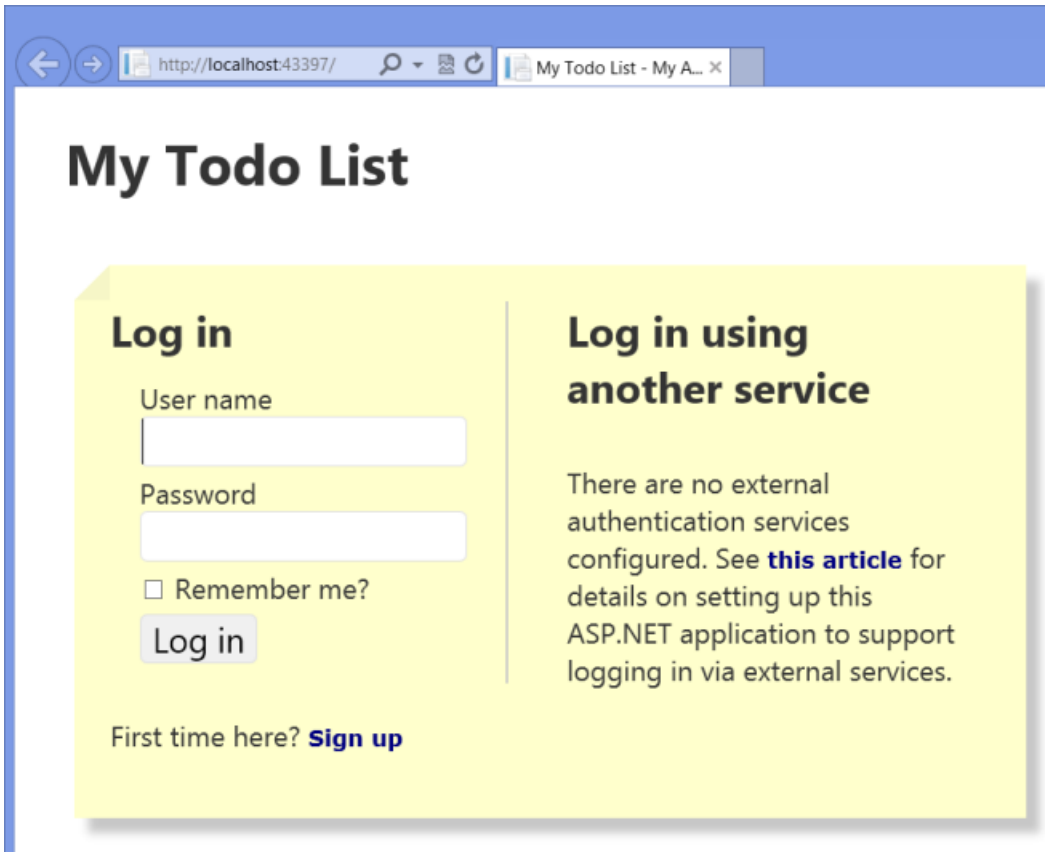
In the **Templates** pane, select **Installed Templates** and expand the **Visual C#** node. Under **Visual C#**, select **Web**. In the list of project templates, select **ASP.NET MVC 4 Web Application**. Name the project and click **OK**.



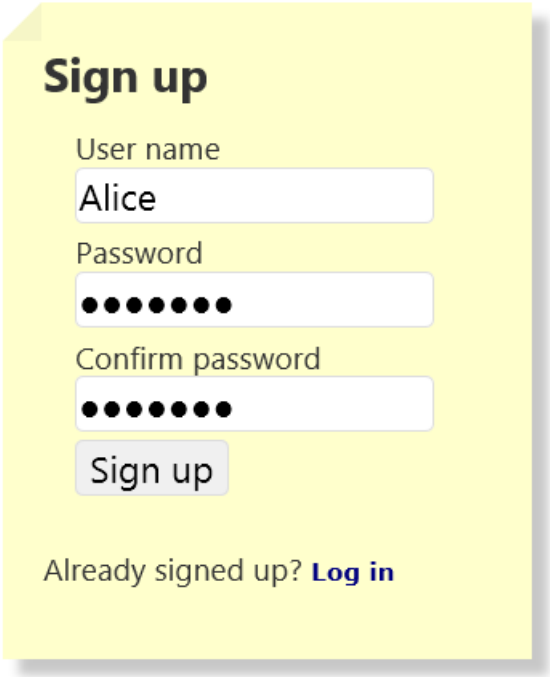
In the **New Project** wizard, select **Single Page Application**.



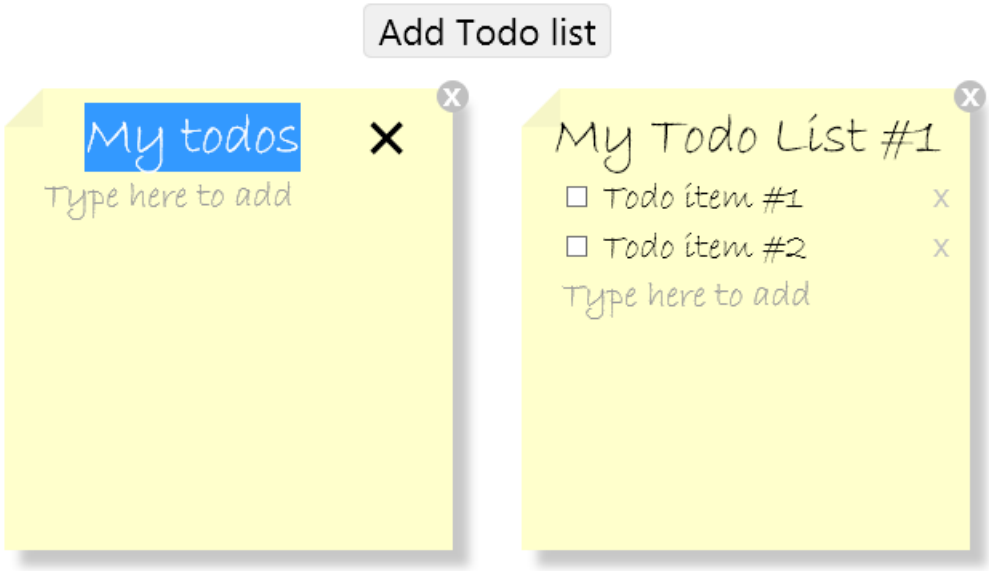
Press F5 to build and run the application. When the application first runs, it displays a login screen.



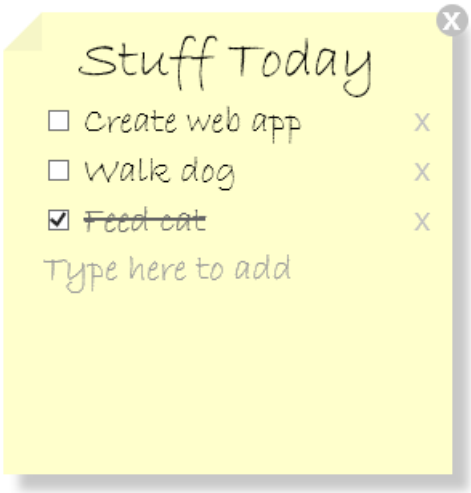
Click the "Sign up" link and create a new user.



After you log in, the application creates a default Todo list with two items. You can click "Add Todo list" to add a new list.

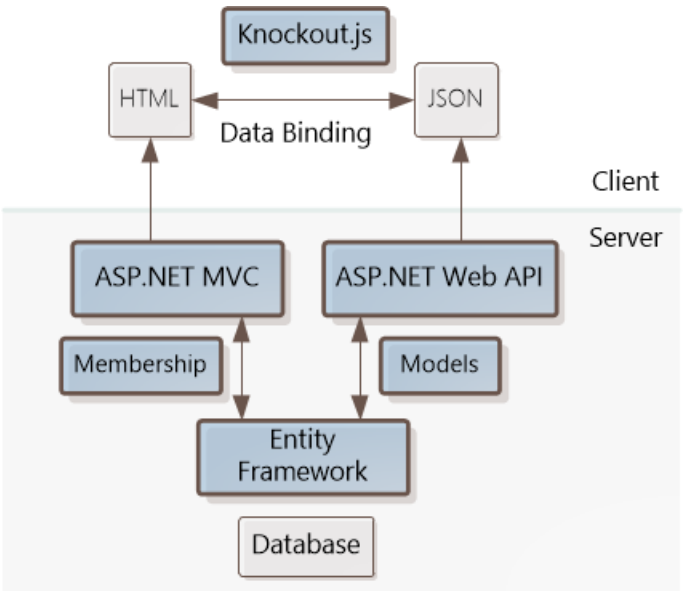


Rename the list, add items to the list, and check them off. You can also delete items or delete an entire list. The changes are automatically persisted to a database on the server (actually LocalDB at this point, because you are running the application locally).



Architecture of the SPA Template

This diagram shows the main building blocks for the application.



On the server side, ASP.NET MVC serves the HTML and also handles forms-based authentication.

ASP.NET Web API handles all requests that relate to the ToDoLists and ToDoItems, including getting, creating, updating and deleting. The client exchanges data with Web API in JSON format.

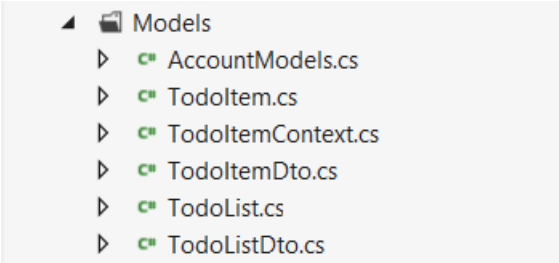
Entity Framework (EF) is the O/RM layer. It mediates between the object-oriented world of ASP.NET and the underlying database. The database uses LocalDB but you can change this in the Web.config file. Typically you would use LocalDB for local development and then deploy to a SQL database on the server, using EF code-first migration.

On the client side, the Knockout.js library handles page updates from AJAX requests. Knockout uses data binding to synchronize the page with the latest data. That way, you don't have to write any of the code that walks through the JSON data and updates the DOM. Instead, you put declarative attributes in the HTML that tell Knockout how to present the data.

A big advantage of this architecture is that it separates the presentation layer from the application logic. You can create the Web API portion without knowing anything about how your web page will look. On the client side, you create a "view model" to represent that data, and the view model uses Knockout to bind to the HTML. That lets you easily change the HTML without changing the view model. (We'll look at Knockout a bit later.)

Models

In the Visual Studio project, the Models folder contains the models that are used on the server side. (There are also models on the client side; we'll get to those.)



TodoItem, TodoList

These are the database models for Entity Framework Code First. Notice that these models have properties that point to each other. `ToDoList` contains a collection of ToDoItems, and each `ToDoItem` has a reference back to its parent ToDoList. These properties are called navigation properties, and they represent the one-to-many relation a to-do list and its to-do items.

The `ToDoItem` class also uses the **[ForeignKey]** attribute to specify that `ToDoListId` is a foreign key into the `ToDoList` table. This tells EF to add a foreign-key constraint to the database.

```
[ForeignKey("TodoList")]
public int TodoListId { get; set; }
public virtual TodoList TodoList { get; set; }
```

TodoItemDto, TodoListDto

These classes define the data that will be sent to the client. "DTO" stands for "data transfer object." The DTO defines how the entities will be serialized into JSON. In general, there are several reasons to use DTOs:

- To control which properties are serialized. The DTO can contain a subset of the properties from the domain model. You might do this for security reasons (to hide sensitive data) or simply to reduce the amount of data that you send.
- To change the shape of the data – e.g. to flatten a more complex data structure.

- To keep any business logic out of the DTO (separation of concerns).
- If your domain models cannot be serialized for some reason. For example, circular references can cause problems when you serialize an object There are ways to handle this problem in Web API (see Handling Circular Object References); but using a DTO simply avoids the problem altogether.

In the SPA template, the DTOs contains the same data as the domain models. However, they are still useful because they avoid circular references from the navigation properties, and they demonstrate the general DTO pattern.

AccountModels.cs

This file contains models for site membership. The `UserProfile` class defines the schema for user profiles in the membership DB. (In this case, the only information is the user ID and the user name.) The other model classes in this file are used to create the user registration and login forms.

Entity Framework

The SPA template uses EF Code First. In Code First development, you define the models first in code, and then EF uses the model to create the database. You can also use EF with an existing database (Database First).

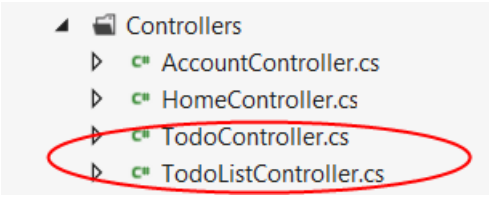
The `TodoItemContext` class in the Models folder derives from **DbContext**. This class provides the "glue" between the models and EF. The `TodoItemContext` holds a `ToDoItem` collection and a `TodoList` collection. To query the database, you simply write a LINQ query against these collections. For example, here is how you can select all of the to-do lists for user "Alice":

```
TodoItemContext db = new TodoItemContext();
IEnumerable<TodoList> lists =
    from td in db.TodoLists where td.UserId == "Alice" select td;
```

You can also add new items to the collection, update items, or delete items from the collection, and persist the changes to the database.

ASP.NET Web API Controllers

In ASP.NET Web API, controllers are objects that handle HTTP requests. As mentioned, the SPA template uses Web API to enable CRUD operations on `ToDoList` and `ToDoItem` instances. The controllers are located in the Controllers folder of the solution.



- `TodoController`: Handles HTTP requests for to-do items
- `TodoListController`: Handles HTTP requests for to-do lists.

These names are significant, because Web API matches the URI path to the controller name. (To learn how Web API routes HTTP requests to controllers, see Routing in ASP.NET Web API.)

Let's look at the `ToDoListController` class. It contains a single data member:

```
private TodoItemContext db = new TodoItemContext();
```

The `TodoItemContext` is used to communicate with EF, as described earlier. The methods on the controller

implement the CRUD operations. Web API maps HTTP requests from the client to controller methods, as follows:

HTTP REQUEST	CONTROLLER METHOD	DESCRIPTION
GET /api/todo	`GetTodoLists`	Gets a collection of to-do lists.
GET /api/todo/*id*	`GetTodoList`	Gets a to-do list by ID
PUT /api/todo/*id*	`PutTodoList`	Updates a to-do list.
POST /api/todo	`PostTodoList`	Creates a new to-do list.
DELETE /api/todo/*id*	`DeleteTodoList`	Deletes a TODO list.

Notice that the URIs for some operations contain placeholders for the ID value. For example, to delete a to-list with an ID of 42, the URI is `/api/todo/42`.

To learn more about using Web API for CRUD operations, see Creating a Web API that Supports CRUD Operations. The code for this controller is fairly straightforward. Here are some interesting points:

- The `GetTodoLists` method uses a LINQ query to filter the results by the ID of the logged-in user. That way, a user only sees the data that belongs to him or her. Also, notice that a Select statement is used to convert the `ToDoList` instances into `TodoListDto` instances.
- The PUT and POST methods check the model state before modifying the database. If **ModelState.IsValid** is false, these methods return HTTP 400, Bad Request. Read more about model validation in Web API at Model Validation.
- The controller class is also decorated with the **[Authorize]** attribute. This attribute checks whether the HTTP request is authenticated. If the request is not authenticated, the client receives HTTP 401, Unauthorized. Read more about authentication at Authentication and Authorization in ASP.NET Web API.

The `TodoController` class is very similar to `TodoListController`. The biggest difference is that it does not define any GET methods, because the client will get the to-do items along with each to-do list.

MVC Controllers and Views

The MVC controllers are also located in the Controllers folder of the solution. `HomeController` renders the main HTML for the application. The view for the Home controller is defined in Views/Home/Index.cshtml. The Home view renders different content depending on whether the user is logged in:

```
@if (@User.Identity.IsAuthenticated)
{
    // ....
}
```

When users are logged in, they see the main UI. Otherwise, they see the login panel. Note that this conditional rendering happens on the server side. Never try to hide sensitive content on the client side—anything that you send in an HTTP response is visible to someone who is watching the raw HTTP messages.

Client-Side JavaScript and Knockout.js

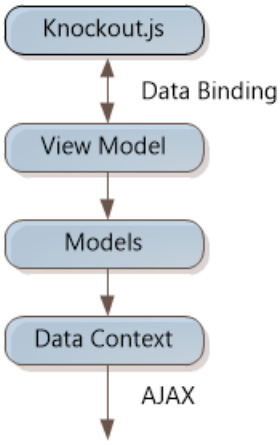
Now let's turn from the server side of the application to the client. The SPA template uses a combination of jQuery and Knockout.js to create a smooth, interactive UI. Knockout.js is a JavaScript library that makes it easy to bind HTML to data. Knockout.js uses a pattern called "Model-View-ViewModel."

- The model is the domain data (ToDo lists and ToDo items).
- The view is the HTML document.
- The view-model is a JavaScript object that holds the model data. The view-model is a code abstraction of the UI. It has no knowledge of the HTML representation. Instead, it represents abstract features of the view, such as "a list of ToDo items".

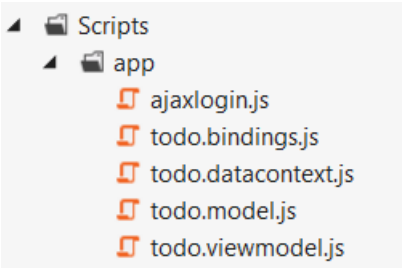
The view is data-bound to the view-model. Updates to the view-model are automatically reflected in the view. Bindings work the other direction as well. Events in the DOM (such as clicks) are data-bound to functions on the view model, which trigger AJAX calls.

The SPA template organizes the client-side JavaScript into three layers:

- todo.datacontext.js: Sends AJAX requests.
- todo.model.js: Defines the models.
- todo.viewmodel.js: Defines the view model.



These script files are located in the Scripts/app folder of the solution.



todo.datacontext handles all AJAX calls to the Web API controllers. (The AJAX calls for logging in are defined elsewhere, in ajaxlogin.js.)

todo.model.js defines the client-side (browser) models for the to-do lists. There are two model classes: todoItem and todoList.

Many of the properties in the model classes are of type "ko.observable". Observables are how Knockout does its magic. From the Knockout documentation: An observable is a "JavaScript object that can notify subscribers about changes." When the value of an observable changes, Knockout updates any HTML elements that are bound to those observables. For example, todoItem has observables for the title and isDone properties:

```
self.title = ko.observable(data.title);
self.isDone = ko.observable(data.isDone);
```

You can also subscribe to an observable in code. For example, the todoItem class subscribes to changes in the "isDone" and "title" properties:

```
saveChanges = function () {
    return datacontext.saveChangedTodoItem(self);
};

// Auto-save when these properties change
self.isDone.subscribe(saveChanges);
self.title.subscribe(saveChanges);
```

View Model

The view model is defined in todo.viewmodel.js. The view model is the central point where the application binds the HTML page elements to the domain data. In the SPA template, the view model contains an observable array of todoLists. The following code in the view model tells Knockout to apply the bindings:

```
ko.applyBindings(window.todoApp.todoListViewModel);
```

HTML and Data Binding

The main HTML for the page is defined in Views/Home/Index.cshtml. Because we are using data-binding, the HTML is only a template for what actually gets rendered. Knockout uses *declarative* bindings. You bind page elements to data by adding a "data-bind" attribute to the element. Here is a very simple example, taken from the Knockout documentation:

```
<p>There are <span data-bind="text: myItems().count"></span> items<p>
```

In this example, Knockout updates the contents of the **** element with the value of `myItems.count()`. Whenever this value changes, Knockout updates the document.

Knockout provides a number of different binding types. Here are some of the bindings used in the SPA template:

- **foreach**: Lets you iterate through a loop and apply the same markup to each item in the list. This is used to render the to-do lists and to-do items. Within the **foreach**, the bindings are applied to the elements of the list.
- **visible**: Used to toggle visibility. Hide markup when a collection is empty, or make the error message visible.
- **value**: Used to populate form values.
- **click**: Binds a click event to a function on the view model.

Anti-CSRF Protection

Cross-Site Request Forgery (CSRF) is an attack where a malicious site sends a request to a vulnerable site where the user is currently logged in. To help prevent CSRF attacks, ASP.NET MVC uses *anti-forgery tokens*, also called request verification tokens. The idea is that the server puts a randomly generated token into a web page. When the client submits data to the server, it must include this value in the request message.

Anti-forgery tokens work because the malicious page cannot read the user's tokens, due to same-origin policies. (Same-origin policies prevent documents hosted on two different sites from accessing each other's content.)

ASP.NET MVC provides built-in support for anti-forgery tokens, through the AntiForgery class and the [ValidateAntiForgeryToken] attribute. Currently, this functionality is not built into Web API. However, the SPA template includes a custom implementation for Web API. This code is defined in the `ValidateHttpAntiForgeryTokenAttribute` class, which is located in the Filters folder of the solution. To learn more about anti-CSRF in Web API, see Preventing Cross-Site Request Forgery (CSRF) Attacks.

Conclusion

The SPA template is designed to get you started quickly writing modern, interactive web applications. It uses the Knockout.js library to separate presentation (HTML markup) from the data and application logic. But Knockout is not the only JavaScript library you can use to create a SPA. If you want to explore some other options, take a look at the community-created SPA templates.

Know a library other than Knockout?

2/20/2020 • 2 minutes to read • Edit Online

by Mads Kristensen

The Single Page Application (SPA) template is a great way to get started writing single-page applications. The template uses KnockoutJS to bind application data to DOM elements.

But Knockout is not the only JavaScript library for creating rich client applications. Other libraries solve similar challenges in different ways. You might prefer a one library over another, so we've made several community-created templates available for download. Each of these templates uses a different mix of client JavaScript libraries.

To install a community-created template, visit one of the template pages listed below and click the Download button. The templates are provided as VSIX files.

BackboneJS

Backbone.js SPA template. This template provides an initial skeleton for developing a Backbone.js application in ASP.NET MVC. Out of the box it provides basic user login functionality, including user sign-up, sign-in, password reset, and user confirmation with basic email templates.

BreezeJS

BreezeJS is an open source library for managing rich data in a JavaScript client. Breeze handles querying, caching, change tracking, validation, and more. Two templates feature Breeze:

- The Breeze/Knockout template extends the Knockout SPA template, showing how easily you can build a single-page application with Breeze for data management and KnockoutJS for data binding.
- The Breeze/Angular template also extends the Knockout SPA template with Breeze, but using the AngularJS library for data binding, dependency injection, and screen management.

In addition, the Hot Towel SPA template uses BreezeJS.

EmberJS

EmberJS SPA template. This template uses Ember, a powerful MVC JavaScript library that solves a wide array of challenges for building rich client applications.

The Ember SPA template is a re-implementation of the Knockout SPA template, using EmberJS and Handlebars templating.

Hot Towel

Hot Towel SPA template. This template brings in several JavaScript libraries, including Breeze, Knockout, RequireJS and Twitter Bootstrap.

Compared with the other templates listed here, the Hot Towel template provides a more complete application from which you can build your own. There are more concepts to be aware of, but once you understand them, this template might just be what you are looking for. If you want to build a SPA but can't decide where to start, use Hot Towel and in seconds you'll have a SPA and all the tools you need to build on it.

Feature table

Here are the features provided by each SPA template:

	ASP.NET SPA	BACKBONE	BREEZE/ANGULAR	BREEZE/KO	EMBER	HOT TOWEL
ToDo sample	✓		✓	✓	✓	
Bare template		✓				✓
Navigation and history		✓	✓		✓	✓
Libraries						
Angular			✓			
Backbone		✓				
Breeze			✓	✓		✓
Durandal						✓
Ember					✓	
Knockout	✓			✓		✓

Templates

2/20/2020 • 2 minutes to read • Edit Online

Introduction to using Single Page Application templates

- Backbone Template
- Breeze/Angular template
- Breeze/Knockout template
- EmberJS template
- Hot Towel template

Backbone Template

2/20/2020 • 7 minutes to read • Edit Online

by Mads Kristensen

> The Backbone SPA Template was written by Kazi Manzur Rashid
>
> Download the Backbone.js SPA Template

The Backbone.js SPA template is designed to get you started quickly building interactive client-side web apps using Backbone.js.

The template provides an initial skeleton for developing a Backbone.js application in ASP.NET MVC. Out of the box it provides basic user login functionality, including user sign-up, sign-in, password reset, and user confirmation with basic email templates.

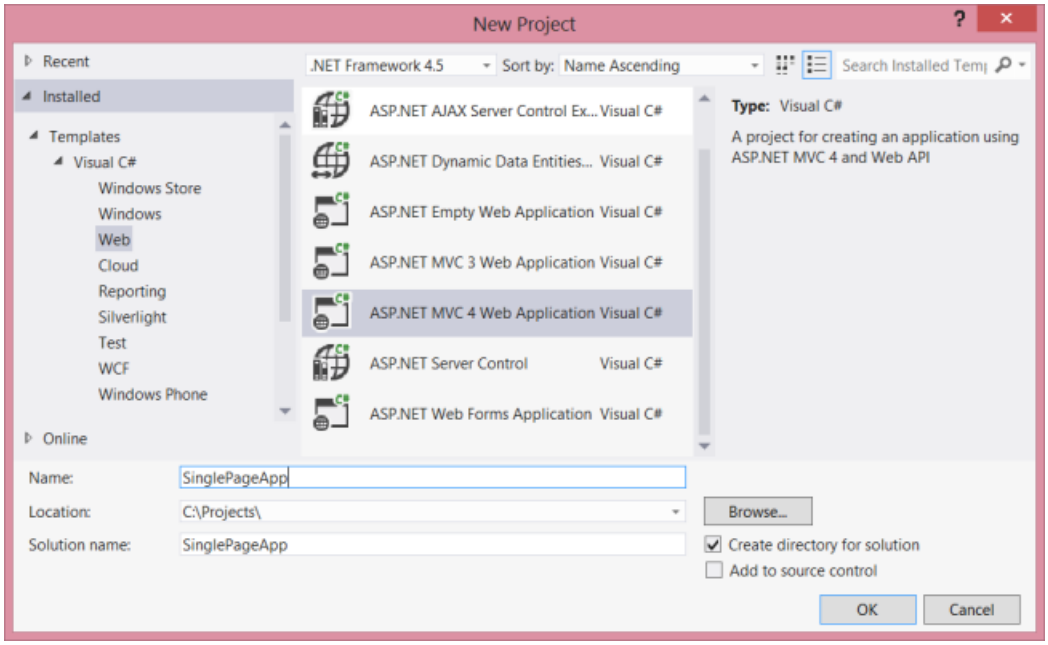
Requirements:

- ASP.NET and Web Tools 2012.2 update

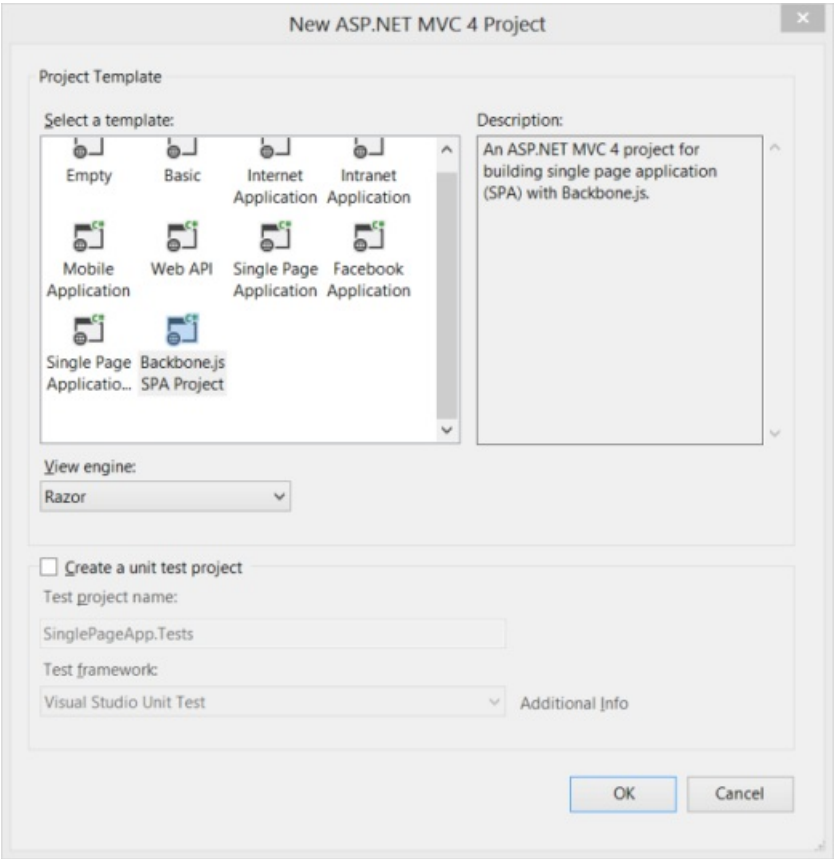
Create a Backbone Template Project

Download and install the template by clicking the Download button above. The template is packaged as a Visual Studio Extension (VSIX) file. You might need to restart Visual Studio.

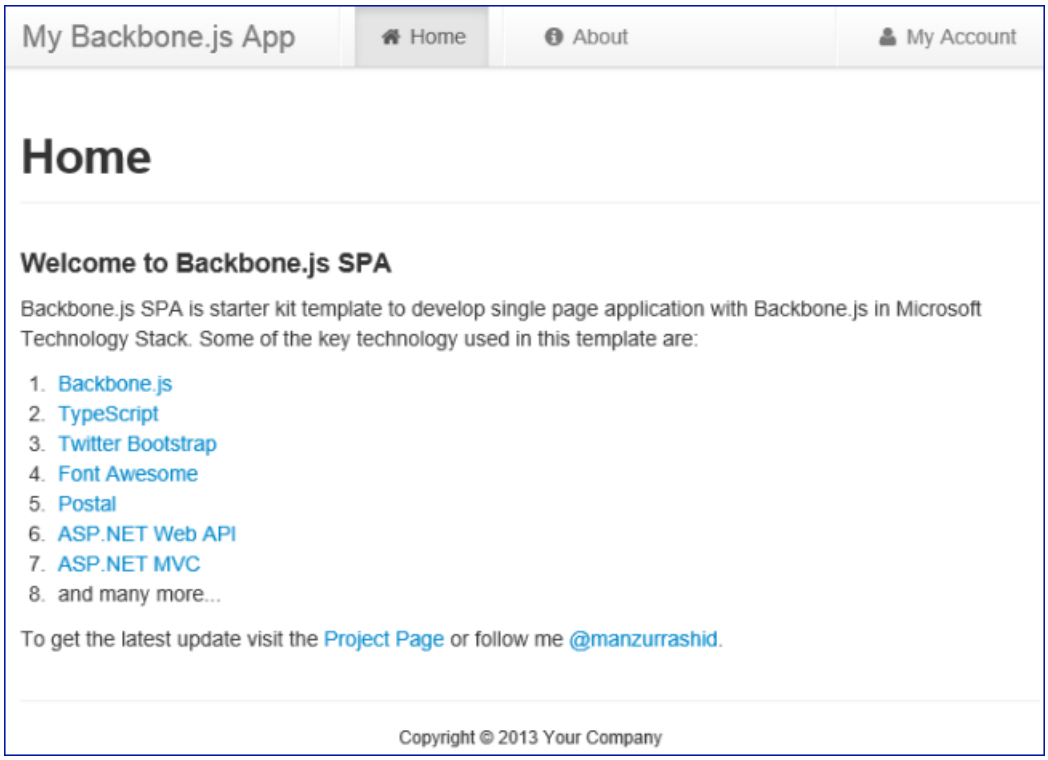
In the **Templates** pane, select **Installed Templates** and expand the **Visual C#** node. Under **Visual C#**, select **Web**. In the list of project templates, select **ASP.NET MVC 4 Web Application**. Name the project and click **OK**.



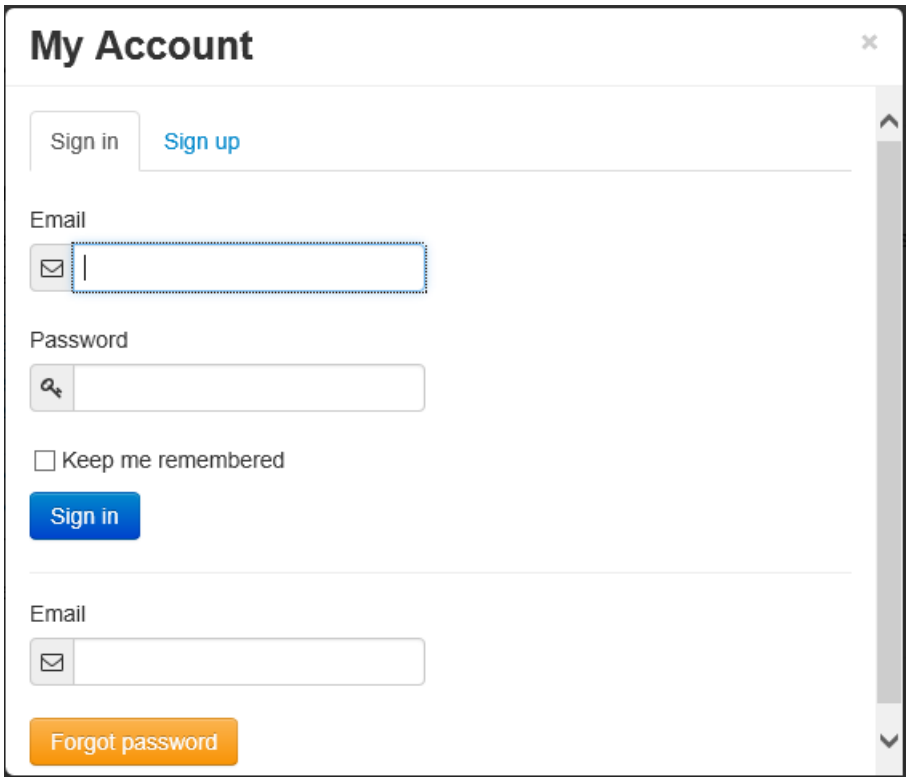
In the **New Project** wizard, select Backbone.js SPA Project.



Press Ctrl-F5 to build and run the application without debugging, or press F5 to run with debugging.



Clicking "My Account" brings up the login page:



Walkthrough: Client Code

Let's starts with the client side. The client application scripts are located in the ~/Scripts/application folder. The application is written in TypeScript (.ts files) which are compiled into JavaScript (.js files).

Application

`Application` is defined in application.ts. This object initializes the application and acts as the root namespace. It maintains configuration and state information that is shared across the application, such as whether the user is signed in.

The `application.start` method creates the modal views and attaches event handlers for application-level events, such as user sign-in. Next, it creates the default router and checks whether any client-side URL is specified. If not, it redirects to the default url (#!/).

Events

Events are always important when developing loosely coupled components. Applications often perform multiple operations in response to a user action. Backbone provides built-in events with components such as Model, Collection, and View. Instead of creating inter-dependencies among these components, the template uses a "pub/sub" model: The `events` object, defined in events.ts, acts as an event hub for publishing and subscribing to application events. The `events` object is a singleton. The following code shows how to subscribe to an event and then trigger the event:

```
events.on('myEvent', (e: MyEventArg) => {
    // Do your work
});

// Later in the code
events.trigger('myEvent', { arg: 'myValue' });
```

Router

In Backbone.js, a router provides methods for routing client-side pages and connecting them to actions and events. The template defines a single router, in router.ts. The router creates the activable views and maintains the state

when switching views. (Activable views are described in the next section.) Initially, the project has two dummy views, Home and About. It also has a NotFound view, which is displayed if the route is not known.

Views

The views are defined in ~/Scripts/application/views. There are two kinds of views, activable views and modal dialog views. Activable views are invoked by the router. When an activable view is shown, all other activable views become inactive. To create an activable view, extend the view with the `Activable` object:

```
export class MyView extends Backbone.View {
    // Other implementation details
}

// Extending with Activable
_.extend(MyView.prototype, Activable);
```

Extending with `Activable` adds two new methods to the view, `activate` and `deactivate`. The router calls these methods to activate and deactive the view.

Modal views are implemented as Twitter Bootstrap modal dialogs. The `Membership` and `Profile` views are modal views. Model views can be invoked by any application events. For example, in the `Navigation` view, clicking the "My Account" link shows either the `Membership` view or the `Profile` view, depending on whether the user is logged in. The `Navigation` attaches click event handlers to any child elements that have the `data-command` attribute. Here is the HTML markup:

```
<li>
  <a href="#" data-command="myAccount">
    <i class="icon-user"></i> My Account
  </a>
</li>
```

Here is the code in navigation.ts to hook up the events:

```
export class Navigation extends Backbone.View {
    // Other implementation details
    handleCommand(e: JQueryEventObject) {
        var command = $(e.currentTarget).attr('data-command');
        if (command) {
            events.trigger(command);
        }
    }
}
Navigation.prototype.events = () => {
    return {
        'click [data-command]': 'handleCommand'
    };
};
```

Models

The models are defined in ~/Scripts/application/models. The models all have three basic things: default attributes, validation rules, and a server-side end point. Here is a typical example:

```
export class Session extends Backbone.Model {
    urlRoot() {
        return serverUrlPrefix + '/sessions'
    }

    defaults(): ISessionAttributes {
        return {
          email: null,
          password: null,
          rememberMe: false
        }
    }

    validate(attributes: ISessionAttributes): IValidationResult {
        var errors = {};

        if (!attributes.email) {
            Validation.addError(errors, 'email', 'Email is required.');
        }

        if (!attributes.password) {
            Validation.addError(errors, 'password', 'Password is required.');
        }

        if (!_.isEmpty(errors)) {
            return { errors: errors };
        }
    }
}
```

Plug-ins

The ~/Scripts/application/lib folder contains a few handy jQuery plug-ins. The form.ts file defines a plug-in for working with form data. Often you need to serialize or deserialize form data and show any model validation errors. The form.ts plug-in has methods such as `serializeFields`, `deserializeFields`, and `showFieldErrors`. The following example shows how to serialize a form to a model.

```
// Here $el is the form element
// Hide existing errors if there is any
this.$el.hideSummaryError().hideFieldErrors();

// Subscribe invalid event which
// is fired when validation fails
model.on('invalid', () =>
    this.$el.showFieldErrors{(
        errors: model.validationError.errors;
    )}
);

model.save(this.$el.serializeFields(), {
    success: () => { }, // lets do something good
    error: (m, jqxhr: JQueryXHR) => {
        if (jqxhr.status === 400) { // bad request
            // Handle server side field errors
            var response = <any>$.parseJSON(jqxhr.responseText);
            if (response && _.has(response, 'ModelState')) {
                return this.$el.showFieldErrors({
                    errors: response.ModelState
                });
            }
        }

        // All other server errors
        this.$el.showSummaryError({
            message: 'An unexpected error has occurred while performing ' +
                'operation.'
        });
    }
});
```

The flashbar.ts plug-in gives various kinds of feedback messages to the user. The methods are `$.showSuccessbar`, `$.showErrorbar` and `$.showInfobar`. Behind the scenes, it uses Twitter Bootstrap alerts to show nicely animated messages.

The confirm.ts plug-in replaces the browser's confirm dialog, although the API is somewhat different:

```
$.confirm({
    prompt: 'Are you sure you want to do it?',
    ok: => { //Do something useful },
    cancel: => { // Do something else }
)};
```

Walkthrough: Server Code

Now let's look at the server side.

Controllers

In a single page application, the server plays only a small role in the user interface. Typically, the server renders the initial page and then sends and receives JSON data.

The template has two MVC controllers: `HomeController` renders the initial page, and `SupportsController` is used to confirm new user accounts and reset passwords. All other controllers in the template are ASP.NET Web API controllers, which send and receive JSON data. By default, the controllers use the new `WebSecurity` class to perform user-related tasks. However, they also have optional constructors that let you pass in delegates for these tasks. This makes testing easier, and lets you replace `WebSecurity` with something else, by using an IoC Container. Here is an example:

```
public class SessionsController : ApiController
{
    private readonly Func<string, string, bool, bool> signIn;
    private readonly Action signOut;

    public SessionsController() : this(WebSecurity.Login, WebSecurity.Logout)
    {
    }

    public SessionsController(
        Func<string, string, bool, bool> signIn,
        Action signOut)
    {
      this.signIn = signIn;
      this.signOut = signOut;
    }

    // Rest of the code
}
```

Views

The views are designed to be modular: Each section of a page has its own dedicated view. In a single page application, it is common to include views that do not have any corresponding controller. You can include a view by calling `@Html.Partial('myView')`, but this gets tedious. To make this easier, the template defines a helper method, `IncludeClientViews`, that renders all of the views in a specified folder:

```
@Html.IncludeClientViews('yourViewFolder')
```

If the folder name is not specified, the default folder name is "ClientViews". If your client view also uses partial views, name the partial view with an underscore character (for example, `_SignUp`). The `IncludeClientViews` method excludes any views whose name starts with an underscore. To include a partial view in the client view, call `Html.ClientView('SignUp')` instead of `Html.Partial('_SignUp')`.

Sending Email

To send email, the template uses Postal. However, Postal is abstracted from the rest of the code with the `IMailer` interface, so you can easily replace it with another implementation. The email templates are located in the Views/Emails folder. The sender's email address is specified in the web.config file, in the `sender.email` key of the **appSettings** section. Also, when `debug="true"` in web.config, the application does not require user email confirmation, to speed up development.

GitHub

You can also find the Backbone.js SPA template on GitHub.

Breeze/Angular template

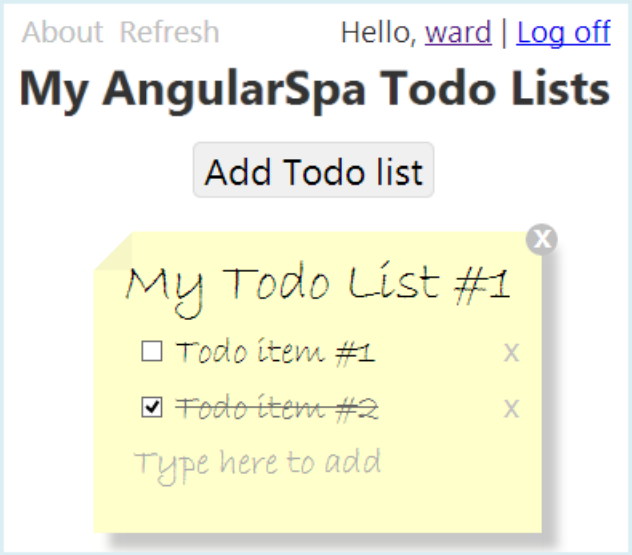
2/20/2020 • 5 minutes to read • Edit Online

by Mads Kristensen

> The Breeze/Angular MVC Template was written by Ward Bell
>
> Download the Breeze/Angular MVC Template

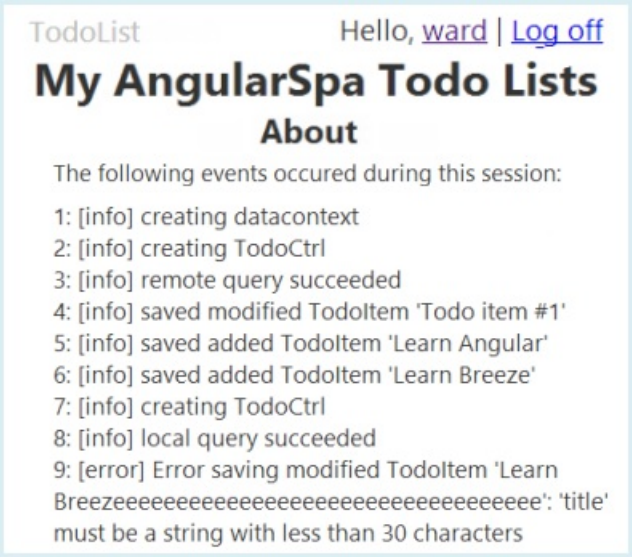
AngularJS is an open source library from Google for building Single Page Applications (SPAs). It offers data binding, dependency injection, and screen management. Combine it with BreezeJS, another open source library for data modeling and data management, and you have the essential ingredients for a great HTML/JavaScript client app.

The Breeze/Angular SPA template is a variation on the KnockoutJS SPA template included in the ASP.NET and Web Tools 2012.2 Update. If you've got Visual Studio, you'll have an example SPA up and running in less than 60 seconds.



Outwardly, the application looks the very similar to the KnockoutJS SPA template. But it's quite different under the hood. The KnockoutJS template uses Knockout for data binding and raw AJAX for data access. The Breeze/Angular template uses Angular for data binding and Breeze for data access. These libraries enable additional capabilities, including page navigation and history.

Here is the application's About page:



This page displays a running log of events during the current user session, including:

- Paging. Note the Todo controller creation at #2 and #7.
- Remote queries (#3) and local cache queries (#7).
- Saving new (#5, #6) and modified (#4) entities.
- Changes validated on the client (#9), so the user can correct mistakes before committing changes to the database.

There's more to explore in this template, including:

- Dynamic loading of HTML view templates.
- Custom data binding through Angular "directives."
- Modularity and dependency injection.
- Query filters, sorts, paging, projections, and inclusion of related entities.
- Sharing data across multiple screens.
- Saving multiple changes as a single transaction.
- Validation rules propagated automatically from the server to the JavaScript client.

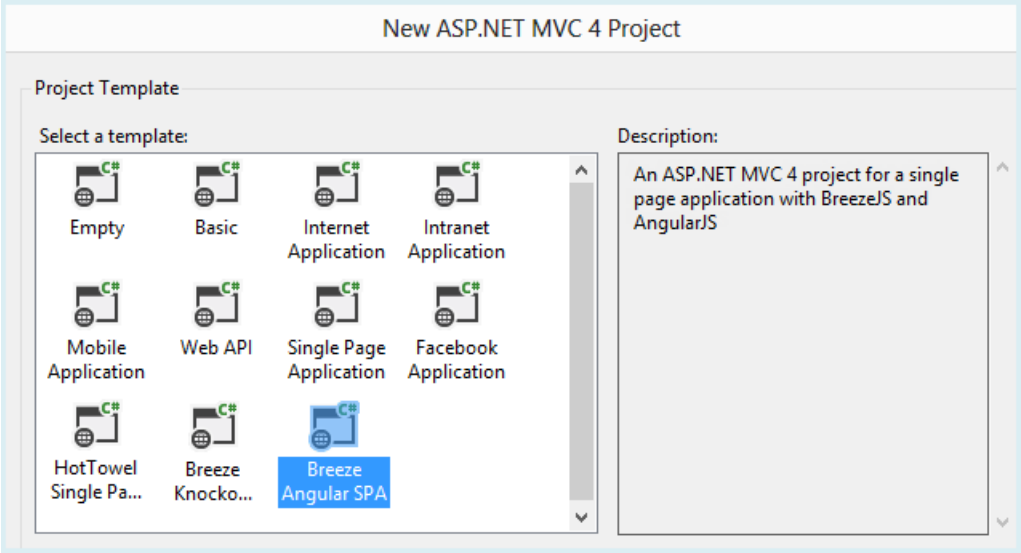
Let's get started.

Create a Breeze/Angular Template Project

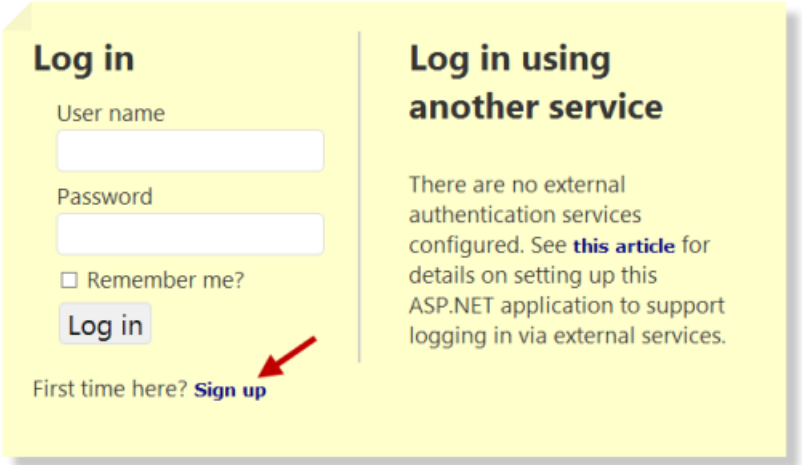
Download and install the template by clicking the Download button above. The template is packaged as a Visual Studio Extension (VSIX) file. You might need to restart Visual Studio.

In the **Templates** pane, select **Installed Templates** and expand the **Visual C#** node. Under **Visual C#**, select **Web**. In the list of project templates, select **ASP.NET MVC 4 Web Application**. Name the project and click **OK**.

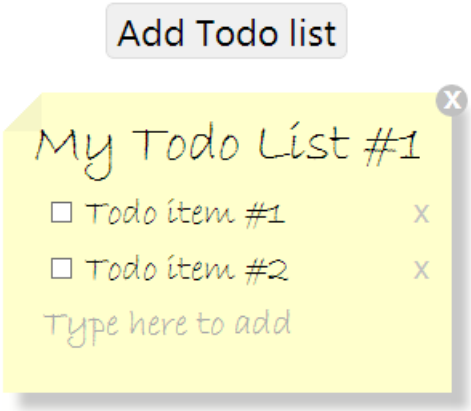
In the **New Project** wizard, select **Breeze Angular SPA**.



Press Ctrl-F5 to build and run the application without debugging, or press F5 to run with debugging.



When the application first runs, it displays a login screen. Click the "Sign up" link and a new page glides into view, where you can enter a username and password. (The login and registration pages are built using ASP.NET MVC.) When you submit the registration form, the server generates a TodoList with two items for your account. Then it presents them to you on a yellow note.



Now you are in the land of SPA. Everything you see and experience while manipulating Todos is rendered and managed on the client with the help of Knockout and Breeze. Explore the app as a user ... but with a developer's eye. Use the developer tools in your browser to capture the network traffic. (In Internet Explorer: Press F12, select the **Network** tab, and click **Start capturing**.) Now try the following:

- Add a new Todo item.
- Click the label and edit the Todo item title
- Check a checkbox to mark the item done. Notice that the textbox is disabled, so the title is no longer editable.

- Click the 'x' to the right of the label. The item disappears and is deleted from the database.
- Pick another item and clear its title. You'll get a validation error that the title is required. After a brief pause, the previous title is restored.
- Type in a ridiculously long title. You'll get a different validation error that the title is too long.
- Click the "Add Todo List" button. A new list appears to the left of the previous list.
- Play with the TodoList title, triggering its required and length validations.
- Click in the title textbox to clear the error message.
- Click the "x" in the circle in the upper right corner to delete the TodoList and its todos.
- Click the "About" link in the upper right to see a log of these activities.

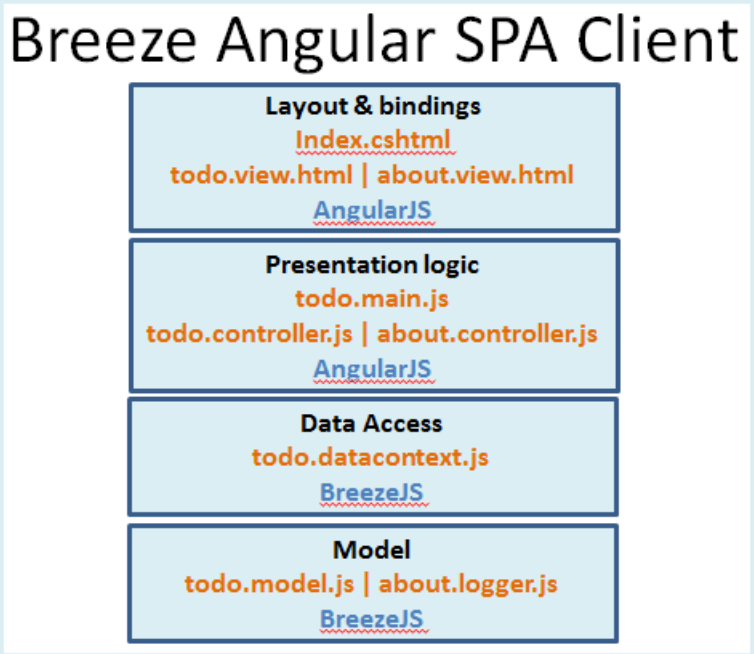
The validation logic is performed client-side by Breeze. Validation attributes on the server model classes are propagated to the client and executed automatically before the client contacts the server.

Review the network traffic. Notice that there were no calls to the server when Breeze detected an error. Each valid change resulted in a POST request to "/api/Todo/SaveChanges". Breeze bundles the changes and sends them together as a single request to the Web API controller's `SaveChanges` method. That's different from KnockoutJS SPA template, which makes PUT, POST, and DELETE requests for each item individually.

Also, notice there is no network traffic when you switch between the TodoList and About pages. That's because the query has been constrained to the local Breeze cache.

Peek inside

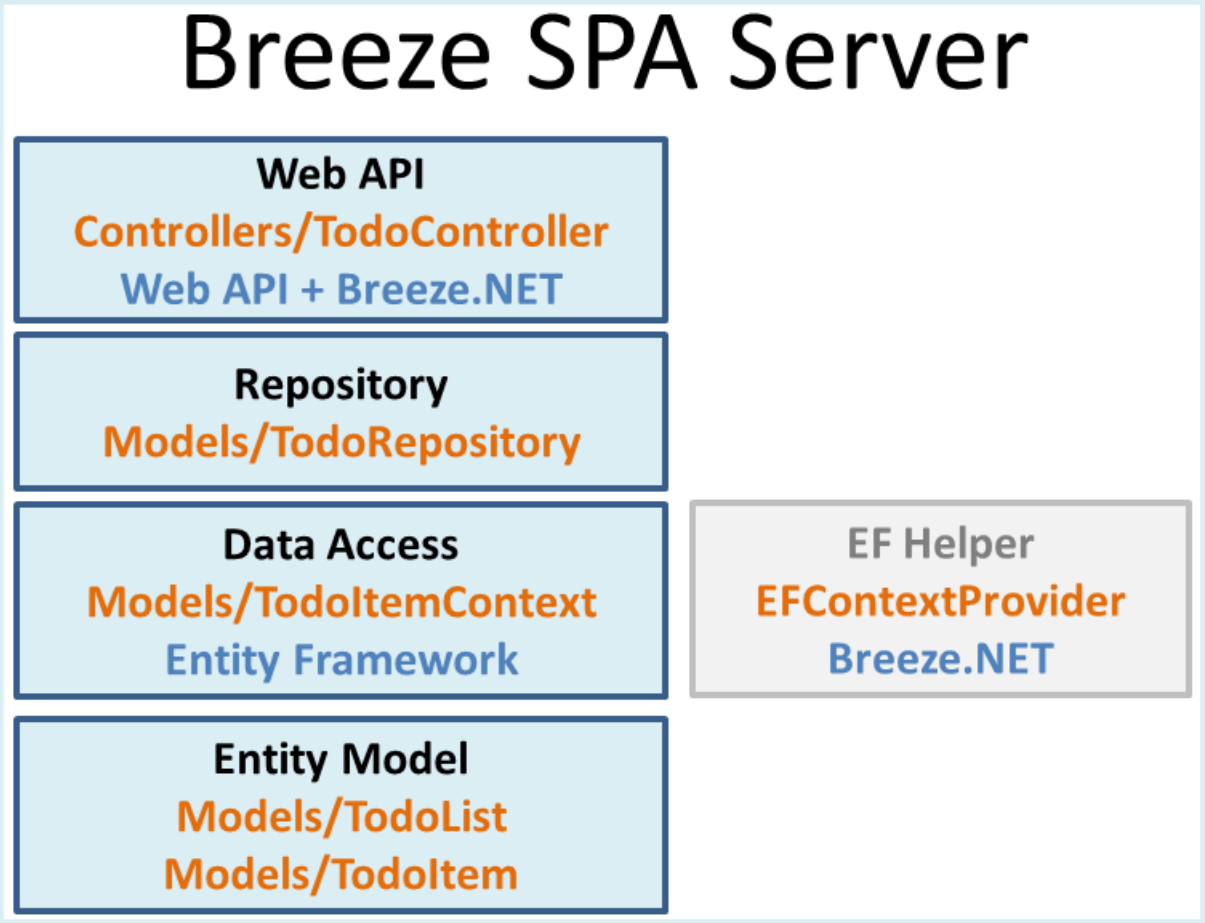
This application has a client side and a server side. The client-side stack consists of a little HTML and a combination of application JavaScript modules (in the "app" folder) plus third-party JavaScript libraries (in the "Scripts" folder).



The UI architecture separates the HTML widgets of the views from the supporting presentation code in the controllers. The Angular data-binding system coordinates views and controllers so that each can do its job without intimate knowledge of the other.

The controller asks the data context to acquire and save the model entities. The data context delegates most of the work to Breeze, which constructs self-tracking model objects from JSON query results.

The server-side stack consists of some developer code and three principle .NET libraries: Web API, Entity Framework, and Breeze.NET:



The basic architecture is the same as the KnockoutJS SPA template. However, the implementation is much simpler: The DTOs were deleted, and most of the Entity Framework details have been delegated to Breeze.NET.

Next Steps

We suggest that you explore the code, guided by the extensive discussion of both the client and the server stacks on the Breeze website.

You might try playing with Breeze client-side query; add some filters and sorts. You might add more model properties and more entities to get a better feel for end-to-end SPA development. When you are confident of the design, you can tear out the Todo features and replace them with your own.

Happy coding!

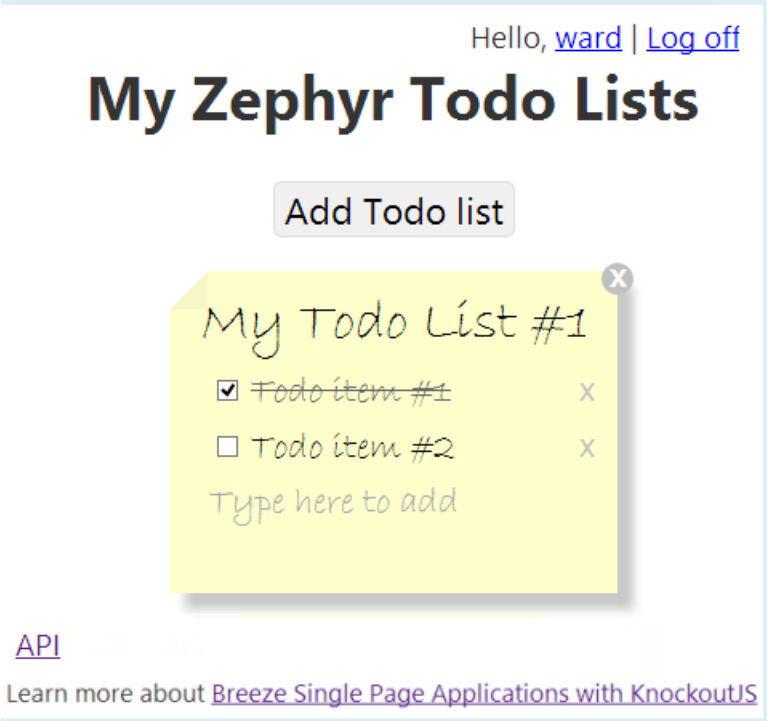
Breeze/Knockout template

2/20/2020 • 6 minutes to read • Edit Online

by Mads Kristensen

> The Breeze/Knockout MVC Template was written by Ward Bell
>
> Download the Breeze/Knockout MVC Template

You've heard of "single page application" (SPA) and wondered what it is. While you could read about it, you'd rather experience it for yourself. But who has time to download a sample? If you've got Visual Studio, you'll have an example SPA up and running in less than 60 seconds with the ASP.NET MVC 4 "Breeze/Knockout Single Page Application" template.



What is the Breeze/Knockout SPA Template?

Most project templates generate an application skeleton. You put flesh on those bones by adding your code and eventually deliver a working application. The Breeze/Knockout SPA template is different. It generates a sample application for you to study. It demonstrates a SPA application design and many of the techniques for building a SPA.

The Breeze/Knockout template is a variation on the KnockoutJS SPA template included in the ASP.NET and Web Tools 2012.2 Update. The Breeze SPA template generates an application with the same user experience, but it has a different implementation, using Breeze for data management.

The KnockoutJS SPA template makes service requests with raw jQuery AJAX, which is adequate for a simple application. But more sophisticated apps have more demanding data management requirements. For example, most applications:

- Query and re-query the server during an extended user session.
- Add query filters, sorting, and paging.

- Share the same data across multiple screens.
- Accumulate changes to many objects, then save them as a single transaction.
- Validate changes on the client, so the user can correct mistakes before committing changes to the database.

The BreezeJS library handles these chores for you, freeing you to develop the application logic and user experience that matter most.

Breeze is an open source library for building rich data applications in JavaScript and HTML, the kinds of apps that have historically been delivered as stand-alone desktop applications.

The Breeze/Knockout template helps you take that first crucial step toward a more robust data management infrastructure. It produces a sample Todo application that is outwardly identical to the KnockoutJS SPA template. On the inside, it replaces the AJAX data layer with Breeze, so you can compare the two approaches side-by-side. Of course, it barely touches the potential of a Breeze application. But you'll see how Breeze works and how little is required to make that transition.

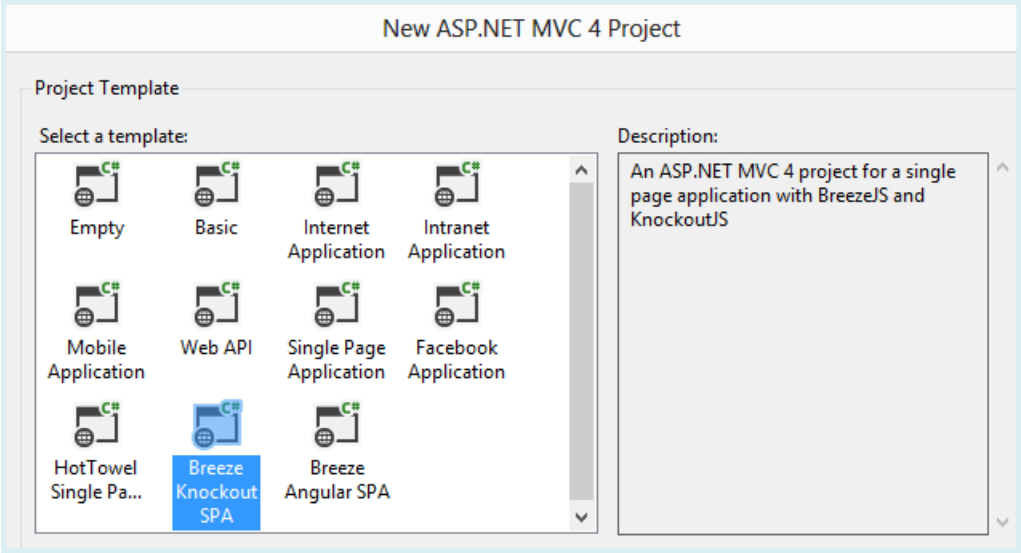
Let's get started.

Create a Breeze/Knockout Template Project

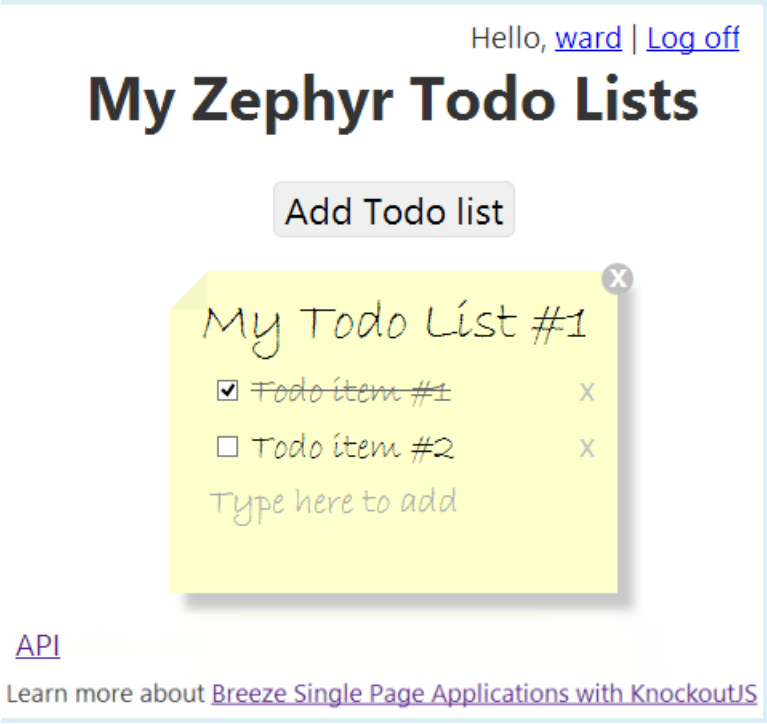
Download and install the template by clicking the Download button above. The template is packaged as a Visual Studio Extension (VSIX) file. You might need to restart Visual Studio.

In the **Templates** pane, select **Installed Templates** and expand the **Visual C#** node. Under **Visual C#**, select **Web**. In the list of project templates, select **ASP.NET MVC 4 Web Application**. Name the project and click **OK**.

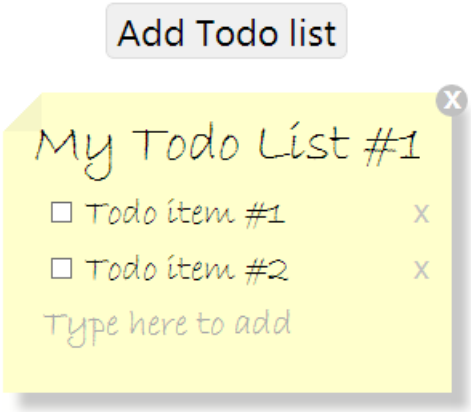
In the **New Project** wizard, select **Breeze Knockout SPA**.



Press Ctrl-F5 to build and run the application without debugging, or press F5 to run with debugging.



When the application first runs, it displays a login screen. Click the "Sign up" link and a new page glides into view, where you can enter a username and password. (The login and registration pages are built using ASP.NET MVC.) When you submit the registration form, the server generates a TodoList with two items for your account. Then it presents them to you on a yellow note.



Now you are in the land of SPA. Everything you see and experience while manipulating Todos is rendered and managed on the client with the help of Knockout and Breeze. Explore the app as a user ... but with a developer's eye. Use the developer tools in your browser to capture the network traffic. (In Internet Explorer: Press F12, select the **Network** tab, and click **Start capturing**.) Now try the following:

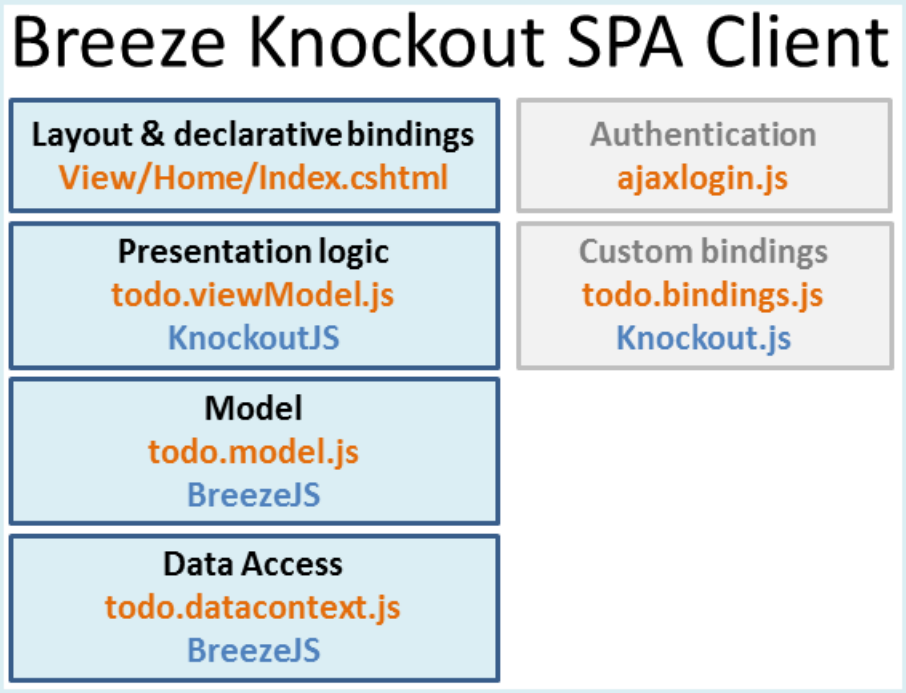
- Add a new Todo item.
- Click the label and edit the Todo item title
- Check a checkbox to mark the item done. Notice that the textbox is disabled, so the title is no longer editable.
- Click the 'x' to the right of the label. The item disappears and is deleted from the database.
- Pick another item and clear its title. You'll get a validation error that the title is required. After a brief pause, the previous title is restored.
- Type in a ridiculously long title. You'll get a different validation error that the title is too long.
- Click the "Add Todo List" button. A new list appears to the left of the previous list.
- Play with the TodoList title, triggering its required and length validations.
- Click in the title textbox to clear the error message.
- Click the "x" in the circle in the upper right corner to delete the TodoList and its todos.

The validation logic is performed client-side by Breeze. Validation attributes on the server model classes are propagated to the client and executed automatically before the client contacts the server.

Review the network traffic. Notice that there were no calls to the server when Breeze detected an error. Each valid change resulted in a POST request to "/api/Todo/SaveChanges". Breeze bundles the changes and sends them together as a single request to the Web API controller's `SaveChanges` method. That's different from KnockoutJS SPA template, which makes PUT, POST, and DELETE requests for each item individually.

Peek inside

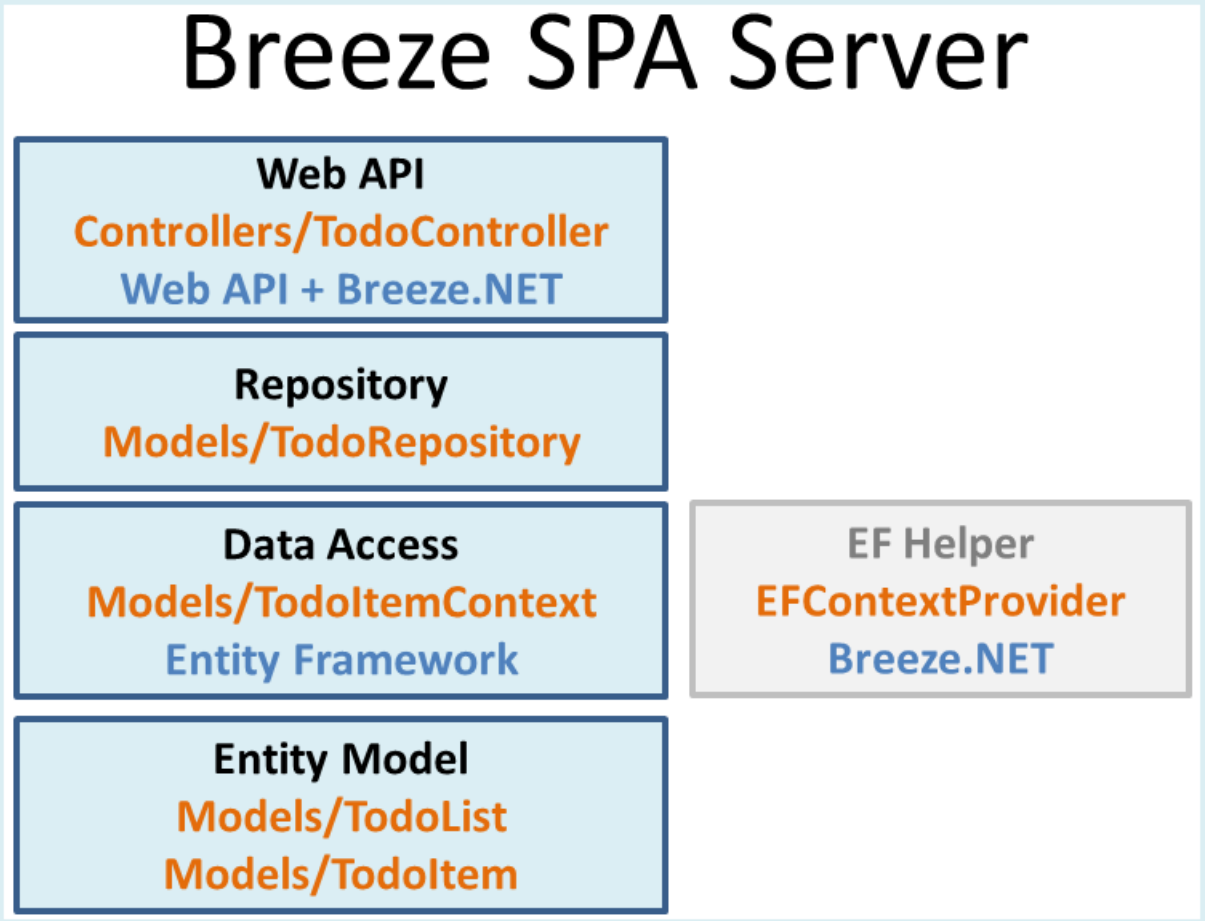
This application has a client side and a server side. The client-side stack consists of a little HTML and a combination of application JavaScript modules (in the "app" folder) plus third-party JavaScript libraries (in the "Scripts" folder).



If you've investigated the KnockoutJS SPA template, this should look very familiar. Focus on the blue boxes. The UI architecture is Model-View-ViewModel (MVVM), in which the HTML widgets of the view are cleanly separated from the supporting presentation code in the view-model. A data binding system (Knockout in this case) coordinates the view and view-model so that each can do its job without intimate knowledge of the other.

The model encapsulates the Todo data. Entities in the model are constructed by Breeze with Knockout observable properties, so they can be bound directly to widgets in the view. The view-model asks the data context to acquire and save the model entities. The data context delegates most of the work to Breeze.

The server-side stack consists of some developer code and three principle .NET libraries: Web API, Entity Framework, and Breeze.NET:



The basic architecture is the same as the KnockoutJS SPA template. However, the implementation is much simpler: The DTOs were deleted, and most of the Entity Framework details have been delegated to Breeze.NET.

Next Steps

We suggest that you explore the code, guided by the extensive discussion of both the client and the server stacks on the Breeze website.

You might try playing with Breeze client-side query; add some filters and sorts. You might add more model properties and more entities to get a better feel for end-to-end SPA development. When you are confident of the design, you can tear out the Todo features and replace them with your own.

Soon you'll be ready for the next big step: Adding client-side screens and navigating among them. You'll leave this SPA template behind and turn to a more complete SPA stack, such as John Papa's Hot Towel, which adds Durandal to the Breeze and Knockout mix.

EmberJS template

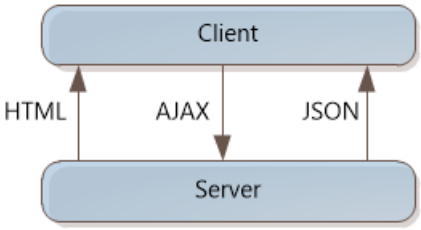
2/20/2020 • 6 minutes to read • Edit Online

by Xinyang Qiu

> The EmberJS MVC Template is written by Nathan Totten, Thiago Santos, and Xinyang Qiu.
>
> Download the EmberJS MVC Template

The EmberJS SPA template is designed to get you started quickly building interactive client-side web apps using EmberJS.

"Single-page application" (SPA) is the general term for a web application that loads a single HTML page and then updates the page dynamically, instead of loading new pages. After the initial page load, the SPA talks with the server through AJAX requests.



AJAX is nothing new, but today there are JavaScript frameworks that make it easier to build and maintain a large sophisticated SPA application. Also, HTML 5 and CSS3 are making it easier to create rich UIs.

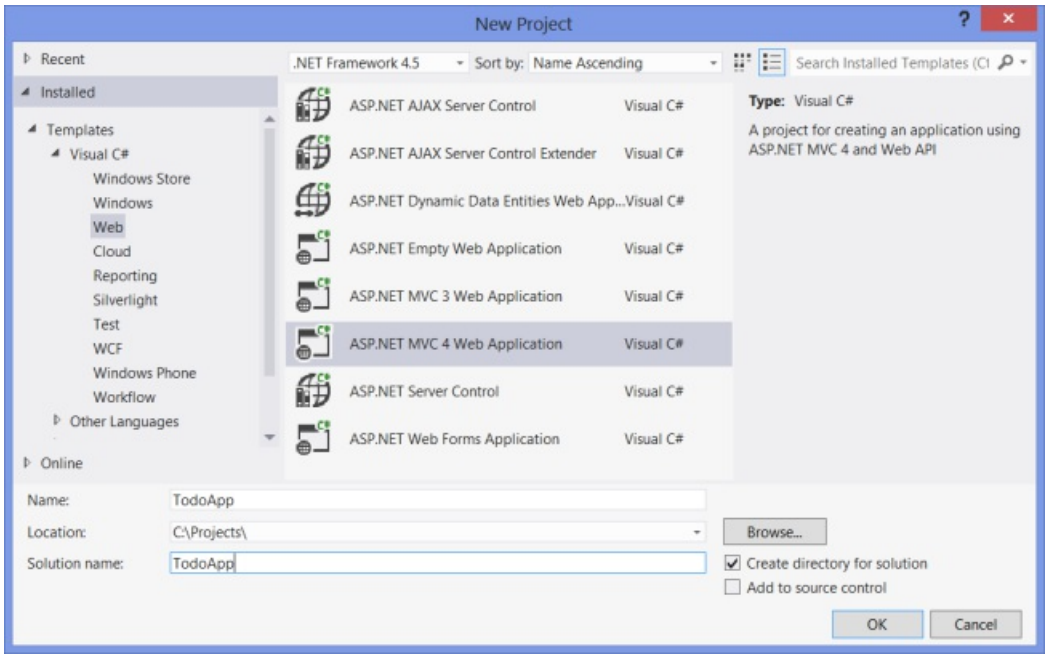
The EmberJS SPA Template uses the Ember JavaScript library to handle page updates from AJAX requests. Ember.js uses data binding to synchronize the page with the latest data. That way, you don't have to write any of the code that walks through the JSON data and updates the DOM. Instead, you put declarative attributes in the HTML that tell Ember.js how to present the data.

On the server side, the EmberJS template is almost identical to the KnockoutJS SPA template. It uses ASP.NET MVC to serve HTML documents, and ASP.NET Web API to handle AJAX requests from the client. For more information about those aspects of the template, refer to the KnockoutJS template documentation. This topic focuses on the differences between the Knockout template and the EmberJS template.

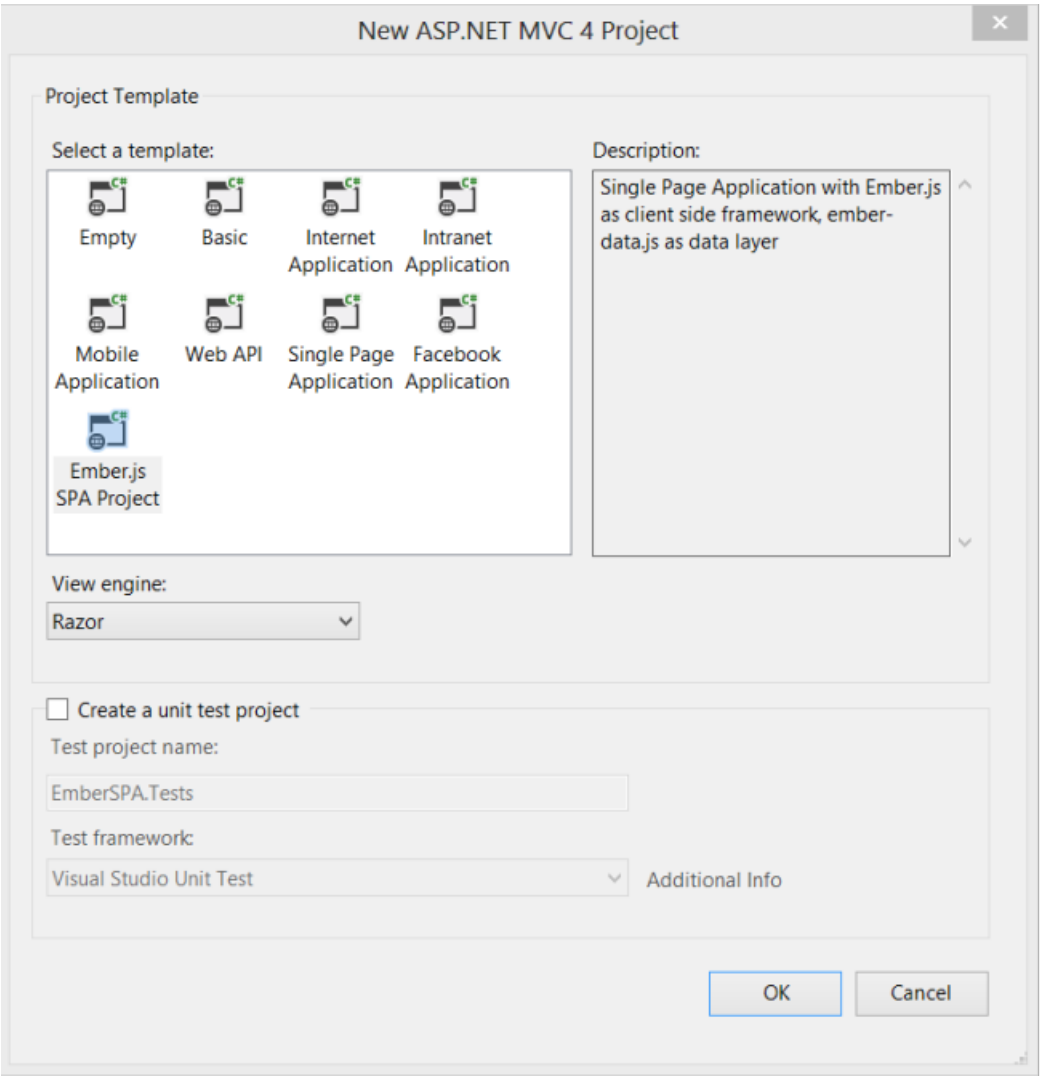
Create an EmberJS SPA Template Project

Download and install the template by clicking the Download button above. You might need to restart Visual Studio.

In the **Templates** pane, select **Installed Templates** and expand the **Visual C#** node. Under **Visual C#**, select **Web**. In the list of project templates, select **ASP.NET MVC 4 Web Application**. Name the project and click **OK**.



In the **New Project** wizard, select **Ember.js SPA Project**.



EmberJS SPA Template Overview

The EmberJS template uses a combination of jQuery, Ember.js, Handlebars.js to create a smooth, interactive UI.

Ember.js is a JavaScript library that uses a client-side MVC pattern.

- A *template*, written in the Handlebars templating language, describes the application user interface. In release mode, the Handlebars compiler is used to bundle and compile the handlebars template.
- A *model* stores the application data that it gets from the server (ToDo lists and ToDo items).
- A *controller* stores application state. Controllers often present model data to the corresponding templates.
- A *view* translates primitive events from the application and passes these to the controller.
- A *router* manages application state, keeping URLs and templates in sync.

In addition, the Ember Data library can be used to synchronize JSON objects (obtained from the server through a RESTful API) and the client models.

The EmberJS SPA template organizes the scripts into eight layers:

- webapi_adapter.js, webapi_serializer.js: Extends the Ember Data library to work with ASP.NET Web API.
- Scripts/helpers.js: Defines new Ember Handlebars helpers.
- Scripts/app.js: Creates the app and configures the adapter and serializer.
- Scripts/app/models/*.js: Defines the models.
- Scripts/app/views/*.js: Defines the views.
- Scripts/app/controllers/*.js: Defines the controllers.
- Scripts/app/routes, Scripts/app/router.js: Defines the routes.
- Templates/*.hbs: Defines the handlebars templates.

Let's look at some of these scripts in more detail.

Models

The models are defined in the Scripts/app/models folder. There are two model files: todoItem.js and todoList.js.

todo.model.js defines the client-side (browser) models for the to-do lists. There are two model classes: todoItem and todoList. In Ember, models are subclasses of DS.Model. A model can have properties with attributes:

```
todoItemId: attr('number'),
title: attr('string')
```

Models can define relationships to other models:

```
todoList: DS.belongsTo('App.TodoList'),
```

Models can have computed properties that bind to other properties:

```
hasError: function () {
    var currentError = this.get("error");
    return !(currentError === '' || currentError === null);
}.property('error'),
```

Models can have observer functions, which are invoked when an observed property changes:

```
saveCheckbox: function () {
    if(this.get("isDirty")){
        if (this.get("todoItemId")) {
            App.store.commit();
        }
    }
}.observes('isDone'),
```

Views

The views are defined in the Scripts/app/views folder. A view translates events from the application UI. An event handler can call back to controller functions, or simply call the data context directly.

For example, the following code is from views/TodoItemEditView.js. It defines the event handling for an input text field.

```
App.TodoItemEditView = Em.TextField.extend({
    lastValue: '',
    focusIn: function (evt) {
        this.lastValue = this.get('parentView').templateData.view.content.get("title");
    },
    focusOut: function (evt) {
        this.changeContent();
    },

    insertNewline: function (evt) {
        $(evt.target).blur();
    },

    changeContent: function () {
        var todoItem = this.get('parentView').templateData.view.content;
        var newValue = todoItem.get("title");
        if (this.lastValue != newValue) {
            App.store.commit();
            this.lastValue = newValue;
        }
    }
});
```

Controller

The controllers are defined in the Scripts/app/controllers folder. To represent a single model, extend `Ember.ObjectController`:

```
App.TodoItemController = Ember.ObjectController.extend({
});
```

A controller can also represent a collection of models by extending `Ember.ArrayController`. For example, the TodoListController represents an array of `todoList` objects. The controller sorts by todoList ID, in descending order:

```
App.TodoListController = Ember.ArrayController.extend({
    error: "",
    sortProperties: ['todoListId'],
    sortAscending: true,

    // ...
```

The controller defines a function named `addTodoList`, which creates a new todoList and adds it to the array. To see how this function gets called, open the template file named todoListTemplate.html, in the Templates folder. The following template code binds a button to the `addTodoList` function:

```
<input type="button" {{action "addTodoList"}} class="isActive" value="Add Todo list"></input>
```

The controller also contains an `error` property, which holds an error message. Here is the template code to

display the error message (also in todoListTemplate.html):

```
<p class="error">{{error}}</p>
```

Routes

Router.js defines the routes and the default template to display, sets up application state, and matches URLs to routes:

```
App.Router.map(function () {
    this.route("index", { path: "/" });
    this.route("about");
    this.route("todoList", { path: "/todo" });
});
```

TodoListRoute.js loads data for the TodoListRoute by overriding the setupController function:

```
App.TodoListRoute = Ember.Route.extend({
    setupController: function (controller, model) {
        controller.set('content', App.TodoList.find());
    }
});
```

Ember uses naming conventions to match URLs, route names, controllers, and templates. For more information, see http://emberjs.com/guides/routing/defining-your-routes/ at the EmberJS documentation.

Templates

The Templates folder contains four templates:

- application.hbs: The default template that is rendered when the application is started.
- about.hbs: The template for the "/about" route.
- index.hbs: The template for the root "/" route.
- todoList.hbs: The template for the "/todo" route.
- _navbar.hbs: The template defines the navigation menu.

The application template acts like a master page. It contains a header, a footer, and an "{{outlet}}" to insert other templates in depending on the route. For more information about application templates in Ember, see http://guides.emberjs.com/v1.10.0/templates/the-application-template//.

The "/todoList" template contains two loop expressions. The outside loop is `{{#each controller}}`, and the inside loop is `{{#each todos}}`. The following code shows a built-in `Ember.Checkbox` view, a customized `App.TodoItemEditView`, and a link with a `deleteTodo` action.

```
{{view Ember.Checkbox checkedBinding="isDone"}}

{{view App.TodoItemEditView valueBinding="title" class="required" disabledBinding="isDone"}}

<a href="#" {{action "deleteTodo" on="click" target="view"}}>X</a>
```

The `HtmlHelperExtensions` class, defined in Controllers/HtmlHelperExtensions.cs, defines a helper function to cache and insert template files when **debug** is set to **true** in the Web.config file. This function is called from the ASP.NET MVC view file defined in Views/Home/App.cshtml:

```
@if (HttpContext.Current.IsDebuggingEnabled)
{
    @Html.RenderEmber()
}
else
{
    @Scripts.Render("~/bundles/templates")
}
```

Called with no arguments, the function renders all of the template files in the Templates folder. You can also specify a subfolder or a specific template file.

When **debug** is **false** in Web.config, the application includes the bundle item "~/bundles/templates". This bundle item is added in BundleConfig.cs, using the Handlebars compiler library:

```
if (!HttpContext.Current.IsDebuggingEnabled)
{
    bundles.Add(new Bundle("~/bundles/templates",
        new EmberHandlebarsBundleTransform()).Include(
            "~/scripts/app/templates/*.hbs"
        ));
}
```

Hot Towel template

2/20/2020 • 6 minutes to read • Edit Online

by Mads Kristensen

> The Hot Towel MVC Template is written by John Papa
>
> Choose which version to download:
>
> Hot Towel MVC Template for Visual Studio 2012
>
> Hot Towel MVC Template for Visual Studio 2013
>
> Hot Towel: Because you don't want to go to the SPA without one!

Want to build a SPA but can't decide where to start? Use Hot Towel and in seconds you'll have a SPA and all the tools you need to build on it!

Hot Towel creates a great starting point for building a Single Page Application (SPA) with ASP.NET. Out of the box you it provides a modular structure for your code, view navigation, data binding, rich data management and simple but elegant styling. Hot Towel provides everything you need to build a SPA, so you can focus on your app, not the plumbing.

> Learn more about building a SPA from John Papa's videos, tutorials and Pluralsight courses.

Application Structure

Hot Towel SPA provides an App folder which contains the JavaScript and HTML files that define your application.

Inside the App folder:

- durandal
- services
- viewmodels
- views

The App folder contains a collection of modules. These modules encapsulate functionality and declare dependencies on other modules. The views folder contains the HTML for your application and the viewmodels folder contains the presentation logic for the views (a common MVVM pattern). The services folder is ideal for housing any common services that your application may need such as HTTP data retrieval or local storage interaction. It is common for multiple viewmodels to re-use code from the service modules.

ASP.NET MVC Server Side Application Structure

Hot Towel builds on the familiar and powerful ASP.NET MVC structure.

- App_Start
- Content
- Controllers
- Models
- Scripts
- Views

Featured Libraries

- ASP.NET MVC
- ASP.NET Web API
- ASP.NET Web Optimization - bundling and minification
- Breeze.js - rich data management
- Durandal.js - navigation and View composition
- Knockout.js - data bindings
- Require.js - Modularity with AMD and optimization
- Toastr.js - pop-up messages
- Twitter Bootstrap - robust CSS styling

Installing via the Visual Studio 2012 Hot Towel SPA Template

Hot Towel can be installed as a Visual Studio 2012 template. Just click `File` | `New Project` and choose `ASP.NET MVC 4 Web Application`. Then select the 'Hot Towel Single Page Application" template and run!

Installing via the NuGet Package

Hot Towel is also a NuGet package that augments an existing empty ASP.NET MVC project. Just install using Nuget and then run!

```
Install-Package HotTowel
```

How Do I Build On Hot Towel?

Simply start adding code!

1. Add your own server-side code, preferably Entity Framework and WebAPI (which really shine with Breeze.js)
2. Add views to the `App/views` folder
3. Add viewmodels to the `App/viewmodels` folder
4. Add HTML and Knockout data bindings to your new views
5. Update the navigation routes in `shell.js`

Walkthrough of the HTML/JavaScript

Views/HotTowel/index.cshtml

index.cshtml is the starting route and view for the MVC application. It contains all the standard meta tags, css links, and JavaScript references you would expect. The body contains a single `<div>` which is where all of the content (your views) will be placed when they are requested. The `@Scripts.Render` uses Require.js to run the entrance point for the application's code, which is contained in the `main.js` file. A splash screen is provided to demonstrate how to create a splash screen while your app loads.

```
<body>
    <div id="applicationHost">
        @Html.Partial("_splash")
    </div>

    @Scripts.Render("~/scripts/vendor")
        <script type="text/javascript" src="~/App/durandal/amd/require.js"
            data-main="@Url.Content("~/App/main")"></script>
</body>
```

App/main.js

The `main.js` file contains the code that will run as soon as your app is loaded. This is where you want to define your navigation routes, set your start up views, and perform any setup/bootstrapping such as priming your application's data.

The `main.js` file defines several of durandal's modules to help the application kick start. The define statement helps resolve the modules dependencies so they are available for the function. First the debugging messages are enabled, which send messages about what events the application is performing to the console window. The app.start code tells durandal framework to start the application. The conventions are set so that durandal knows all views and viewmodels are contained in the same named folders, respectively. Finally, the `app.setRoot` kicks loads the `shell` using a predefined `entrance` animation.

```
define(['durandal/app',
        'durandal/viewLocator',
        'durandal/system',
        'durandal/plugins/router'],
    function (app, viewLocator, system, router) {

    // Enable debug message to show in the console
    system.debug(true);

    app.start().then(function () {
        router.useConvention();
        viewLocator.useConvention();
        //Show the app by setting the root view model for our application.
        app.setRoot('viewmodels/shell', 'entrance');
    });
});
```

Views

Views are found in the `App/views` folder.

shell.html

The `shell.html` contains the master layout for your HTML. All of your other views will be composed somewhere in side of your `shell` view. Hot Towel provides a `shell` with three such regions: a header, a content area, and a footer. Each of these regions is loaded with contents form other views when requested.

The `compose` bindings for the header and footer are hard coded in Hot Towel to point to the `nav` and `footer` views, respectively. The compose binding for the section `#content` is bound to the `router` module's active item. In other words, when you click a navigation link it's corresponding view is loaded in this area.

```
<div>
    <header>
        <!--ko compose: {view: 'nav'} --><!--/ko-->
    </header>
     <section id="content" class="main container-fluid">
        <!--ko compose: {model: router.activeItem,
            afterCompose: router.afterCompose,
            transition: 'entrance'} -->
        <!--/ko-->
    </section>
    <footer>
        <!--ko compose: {view: 'footer'} --><!--/ko-->
    </footer>
</div>
```

nav.html

The `nav.html` contains the navigation links for the SPA. This is where the menu structure can be placed, for example. Often this is data bound (using Knockout) to the `router` module to display the navigation you defined in the `shell.js`. Knockout looks for the data-bind attributes and binds those to the `shell` viewmodel to display the navigation routes and to show a progressbar (using Twitter Bootstrap) if the `router` module is in the middle of navigating from one view to another (see `router.isNavigating`).

```
<nav class="navbar navbar-fixed-top">
    <div class="navbar-inner">
        <a class="brand" href="/">
            <span class="title">Hot Towel SPA</span>
        </a>
        <div class="btn-group" data-bind="foreach: router.visibleRoutes">
            <a data-bind="css: { active: isActive }, attr: { href: hash }, text: name"
                class="btn btn-info" href="#"></a>
        </div>
        <div class="loader pull-right" data-bind="css: { active: router.isNavigating }">
            <div class="progress progress-striped active page-progress-bar">
                <div class="bar" style="width: 100px;"></div>
            </div>
        </div>
    </div>
</nav>
```

home.html and details.html

These views contain HTML for custom views. When the `home` link in the `nav` view's menu is clicked, the `home` view will be placed in the content area of the `shell` view. These views can be augmented or replaced with your own custom views.

footer.html

The `footer.html` contains HTML that appears in the footer, at the bottom of the `shell` view.

ViewModels

ViewModels are found in the `App/viewmodels` folder.

shell.js

The `shell` viewmodel contains properties and functions that are bound to the `shell` view. Often this is where the menu navigation bindings are found (see the `router.mapNav` logic).

```
define(['durandal/system', 'durandal/plugins/router', 'services/logger'],
    function (system, router, logger) {
        var shell = {
            activate: activate,
            router: router
        };

        return shell;

        function activate() {
            return boot();
        }

        function boot() {
            router.mapNav('home');
            router.mapNav('details');
            log('Hot Towel SPA Loaded!', null, true);
            return router.activate('home');
        }

        function log(msg, data, showToast) {
            logger.log(msg, data, system.getModuleId(shell), showToast);
        }
    });
```

home.js and details.js

These viewmodels contain the properties and functions that are bound to the `home` view. it also contains the presentation logic for the view, and is the glue between the data and the view.

```
define(['services/logger'], function (logger) {
    var vm = {
        activate: activate,
        title: 'Home View'
    };

    return vm;

    function activate() {
        logger.log('Home View Activated', null, 'home', true);
        return true;
    }
});
```

Services

Services are found in the App/services folder. Ideally your future services such as a dataservice module, that is responsible for getting and posting remote data, could be placed.

logger.js

Hot Towel provides a `logger` module in the services folder. The `logger` module is ideal for logging messages to the console and to the user in pop up toasts.

ASP.NET WebHooks overview

4/6/2020 • 3 minutes to read • Edit Online

WebHooks is a lightweight HTTP pattern providing a simple pub/sub model for wiring together Web APIs and SaaS services. When an event happens in a service, a notification is sent in the form of an HTTP POST request to registered subscribers. The POST request contains information about the event which makes it possible for the receiver to act accordingly.

Because of their simplicity, WebHooks are already exposed by a large number of services including Dropbox, GitHub, Bitbucket, MailChimp, PayPal, Slack, Stripe, Trello, and many more. For example, a WebHook can indicate that a file has changed in Dropbox, or a code change has been committed in GitHub, or a payment has been initiated in PayPal, or a card has been created in Trello. The possibilities are endless!

Microsoft ASP.NET WebHooks makes it easier to both send and receive WebHooks as part of your ASP.NET application:

- On the receiving side, it provides a common model for receiving and processing WebHooks from any number of WebHook providers. It comes out of the box with support for Dropbox, GitHub, Bitbucket, MailChimp, PayPal, Pusher, Salesforce, Slack, Stripe, Trello,WordPress and Zendesk but it is easy to add support for more.
- On the sending side it provides support for managing and storing subscriptions as well as for sending event notifications to the right set of subscribers. This allows you to define your own set of events that subscribers can subscribe to and notify them when things happens.

The two parts can be used together or apart depending on your scenario. If you only need to receive WebHooks from other services then you can use just the receiver part; if you only want to expose WebHooks for others to consume, then you can do just that.

The code targets ASP.NET Web API 2 and ASP.NET MVC 5 and is available as OSS on GitHub.

WebHooks Overview

WebHooks is a pattern which means that it varies how it is used from service to service but the basic idea is the same. You can think of WebHooks as a simple pub/sub model where a user can subscribe to events happening elsewhere. The event notifications are propagated as HTTP POST requests containing information about the event itself.

Typically the HTTP POST request contains a JSON object or HTML form data determined by the WebHook sender including information about the event causing the WebHook to trigger. For example, a WebHook POST request body from GitHub looks like this as a result of a new issue being opened in a particular repository:

```
{
  "action": "opened",
  "issue": {
      "url": "https://api.github.com/repos/octocat/Hello-World/issues/1347",
      "number": 1347,
      ...
  },
  "repository": {
      "id": 1296269,
      "full_name": "octocat/Hello-World",
      "owner": {
          "login": "octocat",
          "id": 1
          ...
      },
      ...
  },
  "sender": {
      "login": "octocat",
      "id": 1,
      ...
  }
}
```

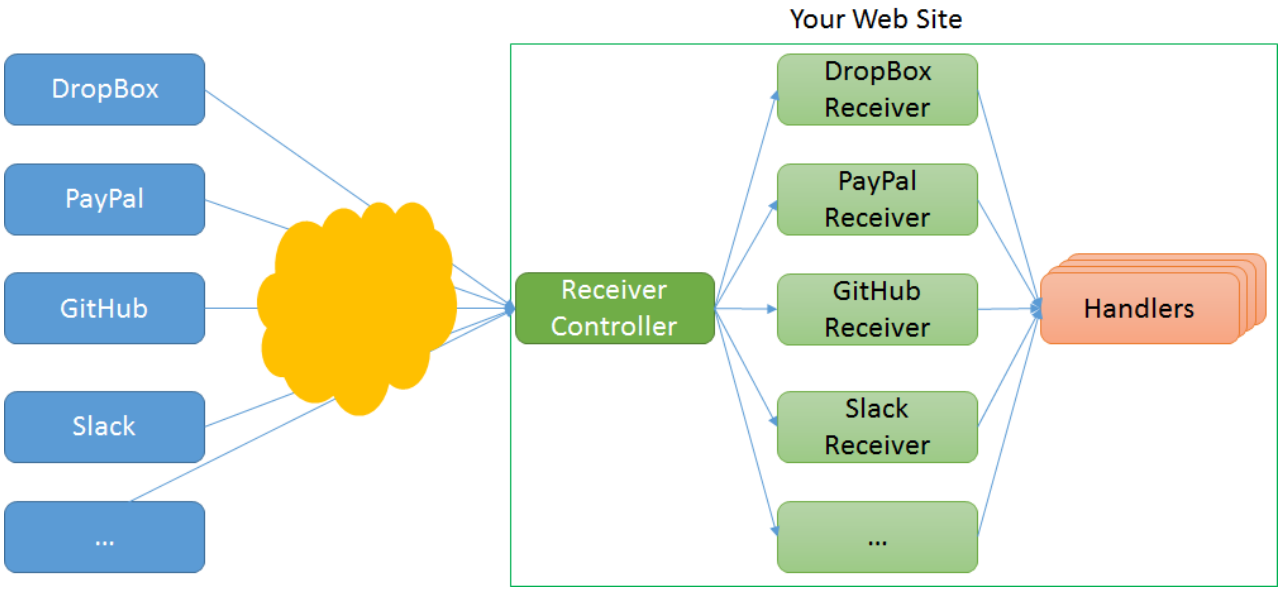
To ensure that the WebHook is indeed from the intended sender, the POST request is secured in some way and then verified by the receiver. For example, GitHub WebHooks includes an *X-Hub-Signature* HTTP header with a hash of the request body which is checked by the receiver implementation so you don't have to worry about it.

The WebHook flow generally goes something like this:

- The WebHook sender exposes events that a client can subscribe to. The events describe observable changes to the system, for example that a new data item has been inserted, that a process has completed, or something else.
- The WebHook receiver subscribes by registering a WebHook consisting of four things:
  1. A URI for where the event notification should be posted in the form of an HTTP POST request;
  2. A set of filters describing the particular events for which the WebHook should be fired;
  3. A secret key which is used to sign the HTTP POST request;
  4. Additional data which is to be included in the HTTP POST request. This can for example be additional HTTP header fields or properties included in the HTTP POST request body.
- Once an event happens, the matching WebHook registrations are found and HTTP POST requests are submitted. Typically, the generation of the HTTP POST requests are retried several times if for some reason the recipient is not responding or the HTTP POST request results in an error response.

WebHooks Processing Pipeline

The Microsoft ASP.NET WebHooks processing pipeline for incoming WebHooks looks like this:



The two key concepts here are *Receivers* and *Handlers*:

- *Receivers* are responsible for handling the particular flavor of WebHook from a given sender and for enforcing security checks to ensure that the WebHook request indeed is from the intended sender.
- *Handlers* are typically where user code runs processing the particular WebHook.

In the following nodes these concepts are described in more details.

ASP.NET WebHooks source code and NuGet packages

4/6/2020 • 2 minutes to read • Edit Online

Microsoft ASP.NET WebHooks is part of the Microsoft ASP.NET family of modules and is hosted as an Open Source Project on GitHub. This means that we accept contributions, but please look at the Contribution Guidelines before submitting a pull request.

This online documentation which you are reading now is also hosted as Open Source on GitHub and also accepts contributions.

NuGet packages

The NuGet packages are divided into three parts:

- Common: A common package that is shared between senders and receivers.
- Sender: A set of packages supporting sending your own WebHooks to others. The functionality for sending WebHooks is described in more detail in Sending WebHooks.
- Receivers: A set of packages supporting receiving WebHooks from others. The functionality for receiving WebHooks is described in more detail in Receiving WebHooks.

ASP.NET WebHooks - receiving and processing

4/6/2020 • 2 minutes to read • Edit Online

- WebHook Receivers
- Processing WebHooks
- Receiver Dependencies

ASP.NET WebHooks receivers

4/6/2020 • 2 minutes to read • Edit Online

Receiving WebHooks depends on who the sender is. Sometimes there are additional steps registering a WebHook verifying that the subscriber is really listening. Some WebHooks provide a push-to-pull model where the HTTP POST request only contains a reference to the event information which is then to be retrieved independently. Often the security model varies quite a bit.

The purpose of Microsoft ASP.NET WebHooks is to make it both simpler and more consistent to wire up your API without spending a lot of time figuring out how to handle any particular variant of WebHooks.

A WebHook receiver is responsible for accepting and verifying WebHooks from a particular sender. A WebHook receiver can support any number of WebHooks, each with their own configuration. For example, the GitHub WebHook receiver can accept WebHooks from any number of GitHub repositories.

WebHook Receiver URIs

By installing Microsoft ASP.NET WebHooks you get a general WebHook controller which accepts WebHook requests from an open-ended number of services. When a request arrives, it picks the appropriate receiver that you have installed for handling a particular WebHook sender.

The URI of this controller is the WebHook URI that you register with the service and is of the form:

```
https://<host>/api/webhooks/incoming/<receiver>/{id}
```

For security reasons, many WebHook receivers require that the URI is an *https* URI and in some cases it must also contain an additional query parameter which is used to enforce that only the intended party can send WebHooks to the URI above.

The `<receiver>` component is the name of the receiver, for example `github` or `slack`.

The *{id}* is an optional identifier which can be used to identify a particular WebHook receiver configuration. This can be used to register N WebHooks with a particular receiver. For example, the following three URIs can be used to register for three independent WebHooks:

```
https://<host>/api/webhooks/incoming/github
https://<host>/api/webhooks/incoming/github/12345
https://<host>/api/webhooks/incoming/github/54321
```

Installing a WebHook Receiver

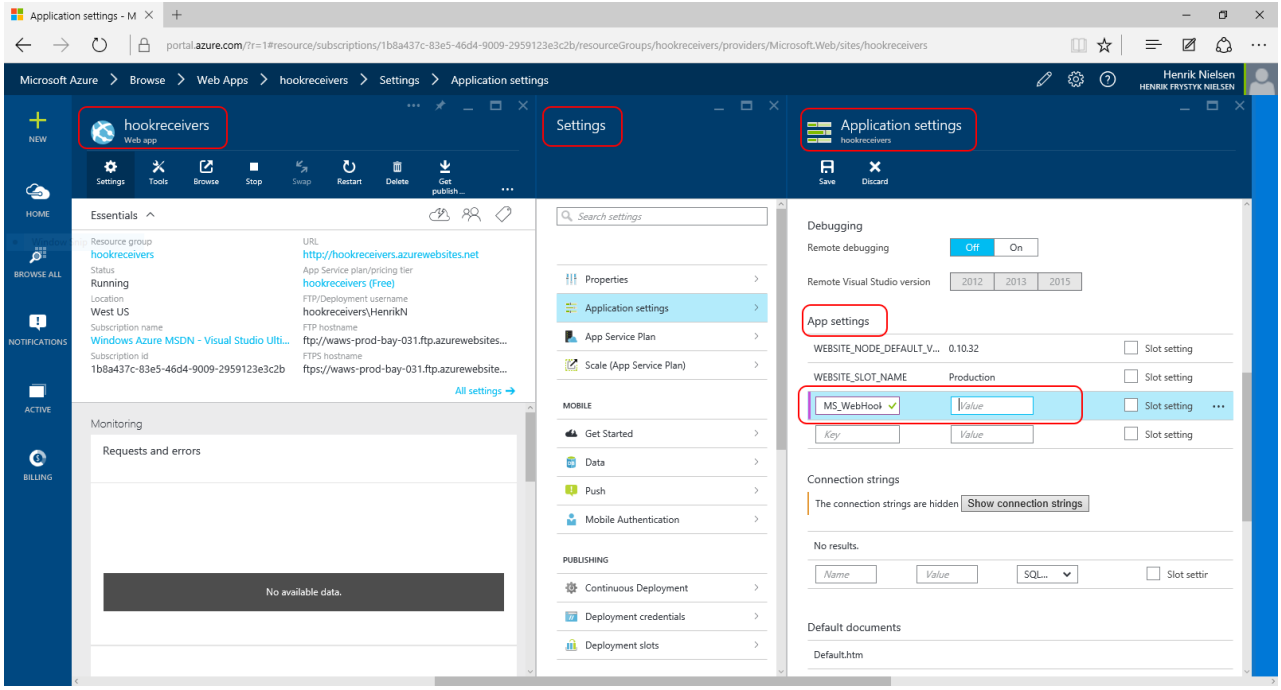
To receive WebHooks using Microsoft ASP.NET WebHooks, you first install the Nuget package for the WebHook provider or providers you want to receive WebHooks from. The Nuget packages are named Microsoft.AspNet.WebHooks.Receivers.* where the last part indicates the service supported. For example

Microsoft.AspNet.WebHooks.Receivers.GitHub provides support for receiving WebHooks from GitHub and Microsoft.AspNet.WebHooks.Receivers.Custom provides support for receiving WebHooks generated by ASP.NET WebHooks.

Out of the box you can find support for Dropbox, GitHub, MailChimp, PayPal, Pusher, Salesforce, Slack, Stripe, Trello, and WordPress but it is possible to support any number of other providers.

Configuring a WebHook Receiver

WebHook Receivers are configured through the IWebHookReceiverConfig inteface and particular implementations of that interface can be registered using any dependency injection model. The default implementation uses Application Settings which can either be set in the Web.config file, or, if using Azure Web Apps, can be set through the Azure Portal.



The format for Application Setting keys is as follows:

```
MS_WebHookReceiverSecret_<receiver>
```

The value is a comma-separated list of values matching the *{id}* values for which WebHooks have been registered, for example:

```
MS_WebHookReceiverSecret_GitHub = <secret1>, 12345=<secret2>, 54321=<secret3>
```

Initializing a WebHook Receiver

WebHook Receivers are initialized by registering them, typically in the *WebApiConfig* static class, for example:

```
namespace WebHookReceivers
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Web API configuration and services

            // Web API routes
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );

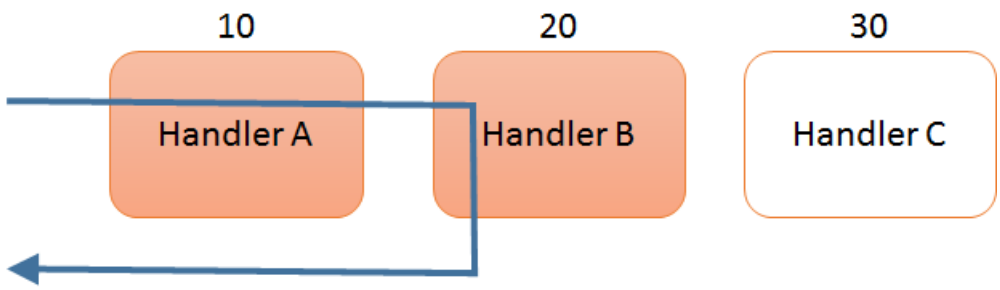
            // Load receivers
            config.InitializeReceiveGitHubWebHooks();
        }
    }
}
```

ASP.NET WebHooks handlers

4/6/2020 • 2 minutes to read • Edit Online

Once WebHooks requests has been validated by a WebHook receiver, it is ready to be processed by user code. This is where *handlers* come in. Handlers derive from the IWebHookHandler interface but typically uses the WebHookHandler class instead of deriving directly from the interface.

A WebHook request can be processed by one or more handlers. Handlers are called in order based on their respective *Order* property going from lowest to highest where Order is a simple integer (suggested to be between 1 and 100):



A handler can optionally set the *Response* property on the WebHookHandlerContext which will lead the processing to stop and the response to be sent back as the HTTP response to the WebHook. In the case above, Handler C won't get called because it has a higher order than B and B sets the response.

Setting the response is typically only relevant for WebHooks where the response can carry information back to the originating API. This is for example the case with Slack WebHooks where the response is posted back to the channel where the WebHook came from. Handlers can set the Receiver property if they only want to receive WebHooks from that particular receiver. If they don't set the receiver they are called for all of them.

One other common use of a response is to use a *410 Gone* response to indicate that the WebHook no longer is active and no further requests should be submitted.

By default a handler will be called by all WebHook receivers. However, if the *Receiver* property is set to the name of a handler then that handler will only receive WebHook requests from that receiver.

Processing a WebHook

When a handler is called, it gets a WebHookHandlerContext containing information about the WebHook request. The data, typically the HTTP request body, is available from the *Data* property.

The type of the data is typically JSON or HTML form data, but it is possible to cast to a more specific type if desired. For example, the custom WebHooks generated by ASP.NET WebHooks can be cast to the type CustomNotifications as follows:

```
public class MyWebHookHandler : WebHookHandler
{
    public MyWebHookHandler()
    {
        this.Receiver = "custom";
    }

    public override Task ExecuteAsync(string generator, WebHookHandlerContext context)
    {
        CustomNotifications notifications = context.GetDataOrDefault<CustomNotifications>();
        foreach (var notification in notifications.Notifications)
        {
            ...
        }
        return Task.FromResult(true);
    }
}
```

Queued Processing

Most WebHook senders will resend a WebHook if a response is not generated within a handful of seconds. This means that your handler must complete the processing within that time frame in order not for it to be called again.

If the processing takes longer, or is better handled separately then the WebHookQueueHandler can be used to submit the WebHook request to a queue, for example Azure Storage Queue.

An outline of a WebHookQueueHandler implementation is provided here:

```
public class QueueHandler : WebHookQueueHandler
{
    public override Task EnqueueAsync(WebHookQueueContext context)
    {

        // Enqueue WebHookQueueContext to your queuing system of choice

        return Task.FromResult(true);
    }
}
```

ASP.NET WebHooks receiver dependencies

4/6/2020 • 2 minutes to read • Edit Online

Microsoft ASP.NET WebHooks is designed with dependency injection in mind. Most dependencies in the system can be replaced with alternative implementations using a dependency injection engine.

See DependencyScopeExtensions for a list of receiver dependencies. If no dependency has been registered, a default implementation is used. See ReceiverServices for a list of default implementations.

ASP.NET WebHook senders

4/10/2020 • 2 minutes to read • Edit Online

See the blog Sending WebHooks with ASP.NET WebHooks Preview for more details.

ASP.NET WebHooks Diagnostics

4/6/2020 • 2 minutes to read • Edit Online

- Logging
- Debugging

ASP.NET WebHooks logging

4/6/2020 • 2 minutes to read • Edit Online

Microsoft ASP.NET WebHooks uses logging as a way of reporting issues and problems. By default logs are written using System.Diagnostics.Trace where they can be manged using Trace Listeners like any other log stream.

When deploying your Web Application as an Azure Web App, the logs are automatically picked up and can be managed together with any other System.Diagnostics.Trace logging. For details, please see Enable diagnostics logging for web apps in Azure App Service

In addition, logs can be obtained straight from inside Visual Studio as described in Troubleshoot a web app in Azure App Service using Visual Studio.

Redirecting Logs

Instead of writing logs to System.Diagnostics.Trace, it is possible to provide an alternate logging implementation that can log directly to a log manager such as Log4Net and NLog. Simply provide an implementation of ILogger and register it with a dependency injection engine of your choice and it will get picked up by Microsoft ASP.NET WebHooks. Please see Dependency Injection in ASP.NET Web API 2 for details.

ASP.NET WebHooks debugging

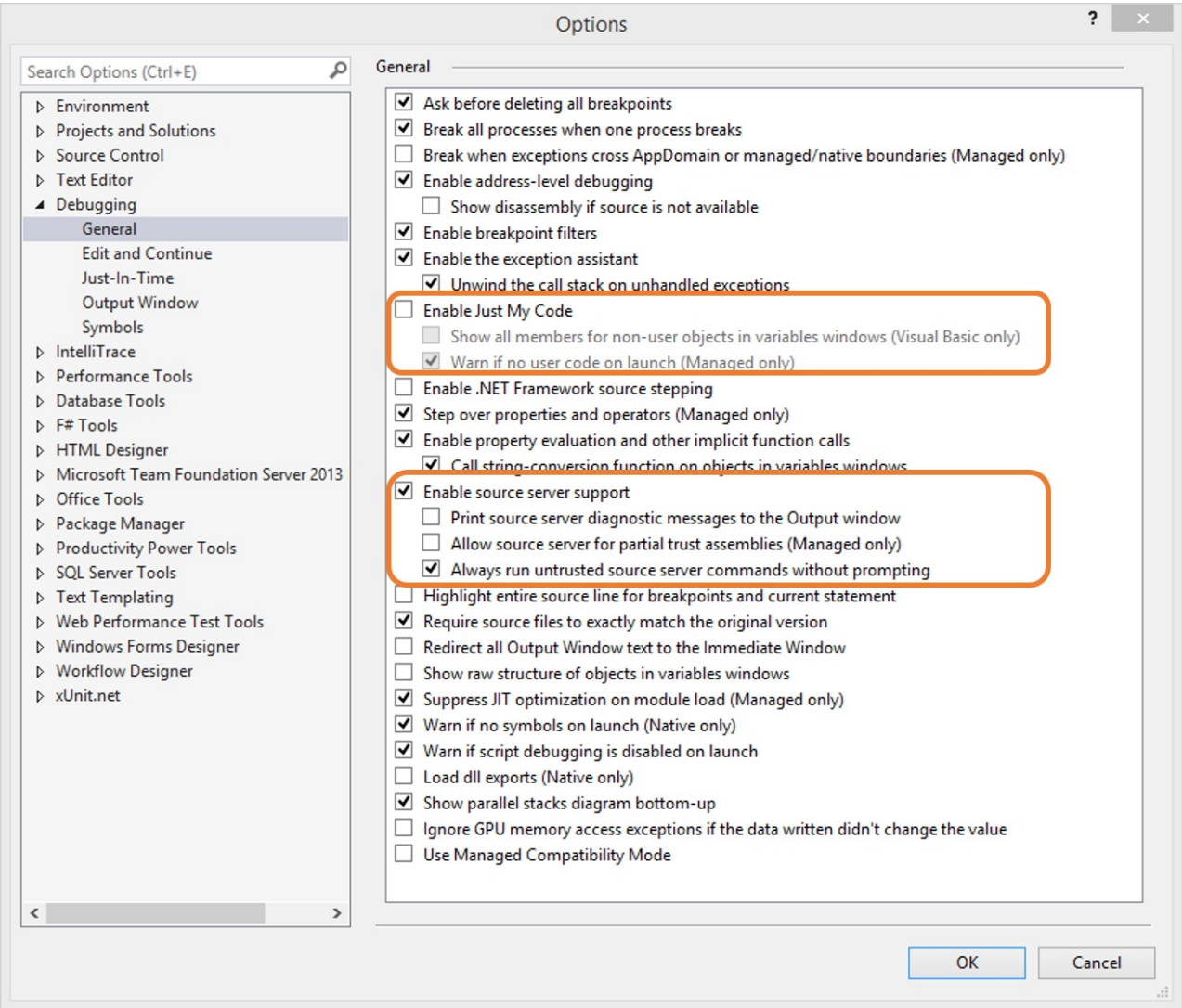
4/6/2020 • 2 minutes to read • Edit Online

Debugging in Azure

To debug your Web Application while running in Azure, please see the tutorial Troubleshoot a web app in Azure App Service using Visual Studio.

Debugging with Source and Symbols

In addition to debugging your own code, it is possible to debug directly into Microsoft ASP.NET WebHooks, and in fact all of .NET. This works regardless of whether you debug locally or remotely. First, configure Visual Studio to find the source and symbols by going to **Debug** and then **Options and Settings**. Set the options like this:



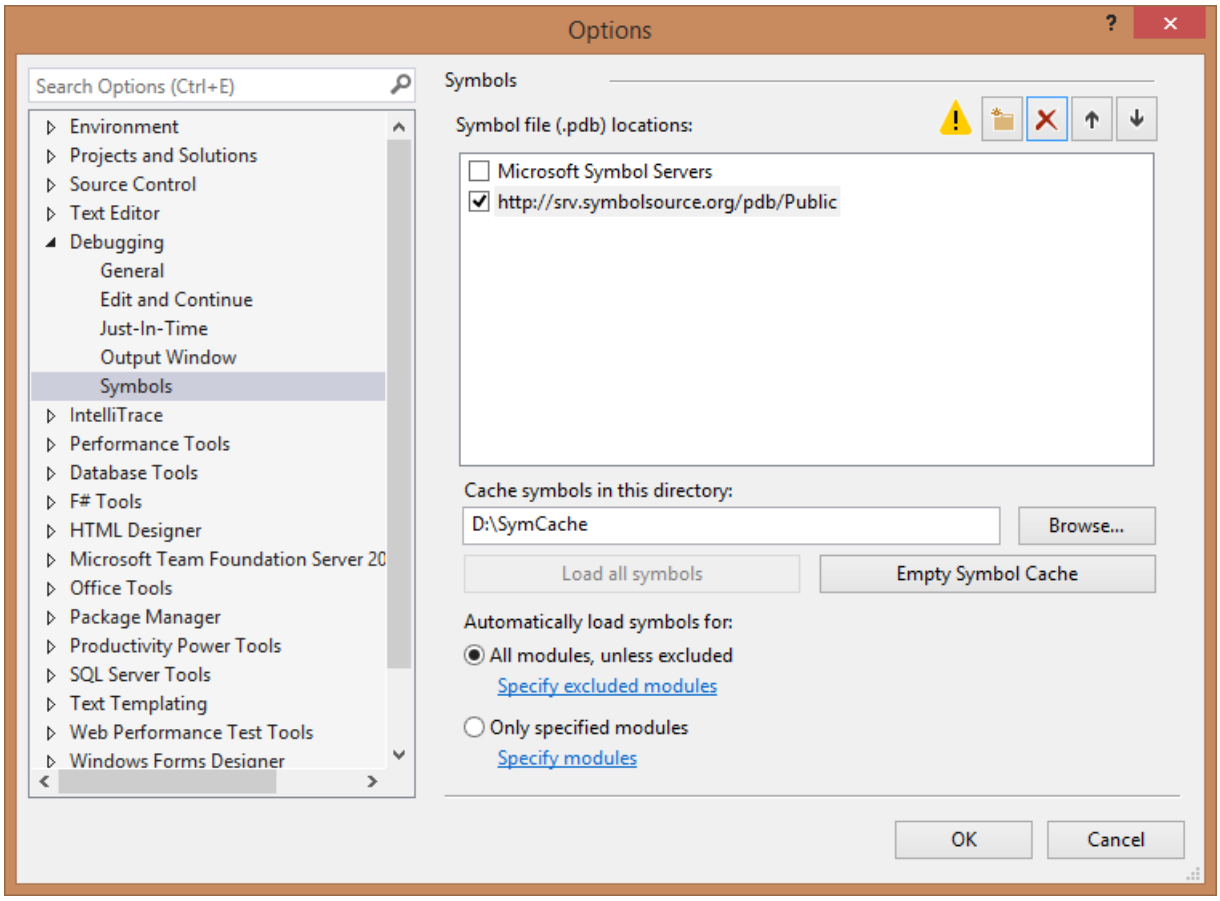
Then add a link to symbolsource.org for downloading the source and symbols. Go to the **Symbols** tab of the menu above and add the following as a symbol location:

```
http://srv.symbolsource.org/pdb/Public
```

In addition, make sure that the cache directory has a short name; otherwise the file names can get too long which will cause the symbols to not load. A sample path is:

```
C:\SymCache
```

The settings should look similar to this:



ASP.NET and Visual Studio 2017

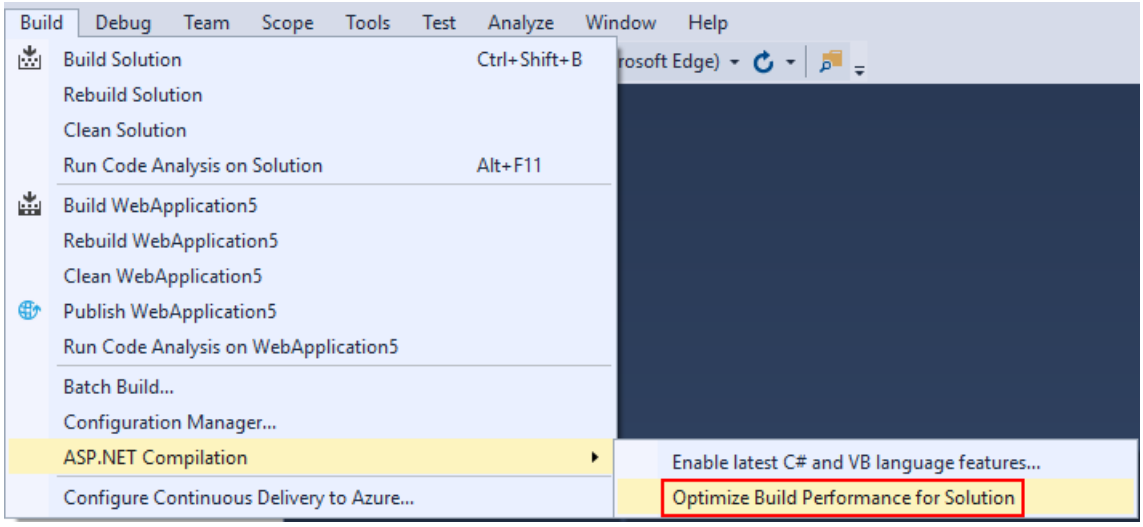
2/20/2020 • 2 minutes to read • Edit Online

Optimize build performance for solution

Optimize build performance for solution

2/20/2020 • 2 minutes to read • Edit Online

Visual Studio 2017 15.8 or later include a menu item: **Build** > **ASP.NET Compilation** > **Optimize Build Performance for Solution**.



ASP.NET compiles its views at runtime, which means that an ASP.NET project carries with it a copy of the compiler. However on a developer machine when the copy of the compiler doesn't match Visual Studio's copy, build performance is impacted on the order of 1-3 seconds per incremental build. This feature updates your project's copy of the compiler to match Visual Studio's, which usually speeds up incremental builds.

This is applicable to ASP.NET Framework 4.7.1 or later projects only, it does not apply to ASP.NET Core.

ASP.NET and Visual Studio 2013

2/20/2020 • 2 minutes to read • Edit Online

Developing ASP.NET web apps with Visual Studio 2013

- Creating ASP.NET Web Projects in Visual Studio 2013
- Visual Studio 2013 Web Editor Features - Browser Link
- Visual Studio 2013 Web Editor Features - CSS
- Visual Studio 2013 Web Editor Features - JavaScript
- Visual Studio 2013 Web Editor Features - Publishing
- Visual Studio 2013 Web Editor Features - Page Inspector
- Visual Studio 2013 Web Editor Features - HTML Editor
- Visual Studio 2013 Web Editor Features - HTML5
- ASP.NET Scaffolding in Visual Studio 2013
- Browser Link
- ASP.NET and Web Tools for Visual Studio 2013 Release Notes
- ASP.NET and Web Tools 2013.2 for Visual Studio 2013 Release Notes
- Hands On Lab: Visual Studio 2013 Web Tools
- Hands On Lab: One ASP.NET: Integrating ASP.NET Web Forms, MVC and Web API

Creating ASP.NET Web Projects in Visual Studio 2013

2/20/2020 • 22 minutes to read • Edit Online

by Tom Dykstra

This topic explains the options for creating ASP.NET web projects in Visual Studio 2013 with Update 3

Here are some of the new features for web development compared to earlier versions of Visual Studio:

- A simple UI for creating projects that offer support for multiple ASP.NET frameworks (Web Forms, MVC, and Web API).
- ASP.NET Identity, a new ASP.NET membership system that works the same in all ASP.NET frameworks and works with web hosting software other than IIS.
- The use of Bootstrap to provide responsive design and theming capabilities.
- New features for Web Forms that used to be offered only for MVC, such as automatic test project creation and an Intranet site template.

For information about how to create web projects for Azure Cloud Services or Azure Mobile Services, see Get Started with Azure Cloud Services and ASP.NET and Creating a Leaderboard App with Azure Mobile Services .NET Backend.

Prerequisites

This article applies to Visual Studio 2013 with Update 3 installed.

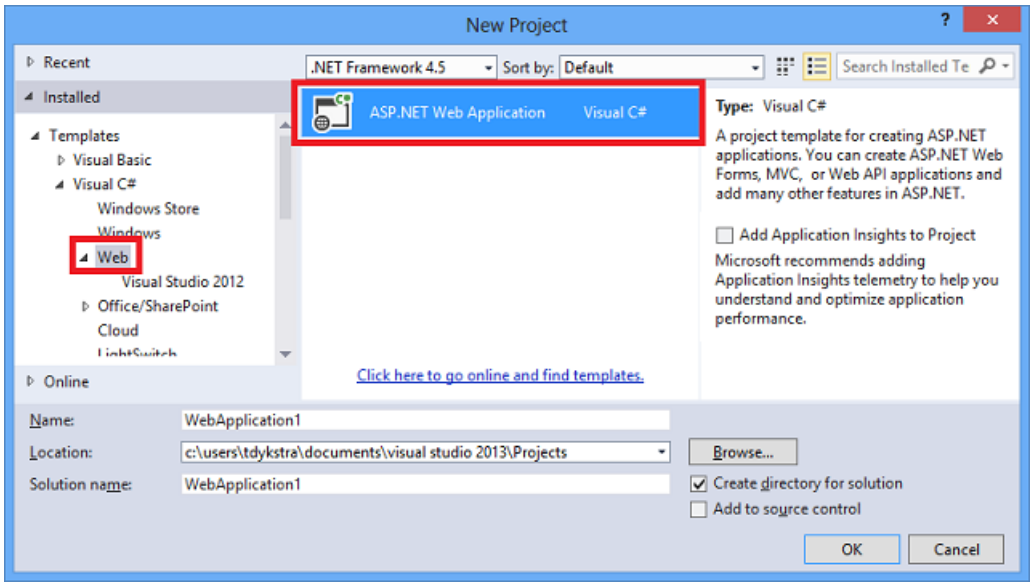
Web application projects versus web site projects

ASP.NET gives you a choice between two kinds of web projects: *web application projects* and *web site projects*. We recommend web application projects for new development, and this article applies only to web application projects. For more information, see Web Application Projects versus Web Site Projects in Visual Studio on the MSDN site.

Overview of web application project creation

The following steps show how to create a web project:

1. Click **New Project** in the **Start** page or in the **File** menu.
2. In the **New Project** dialog, click **Web** in the left pane and **ASP.NET Web Application** in the middle pane.

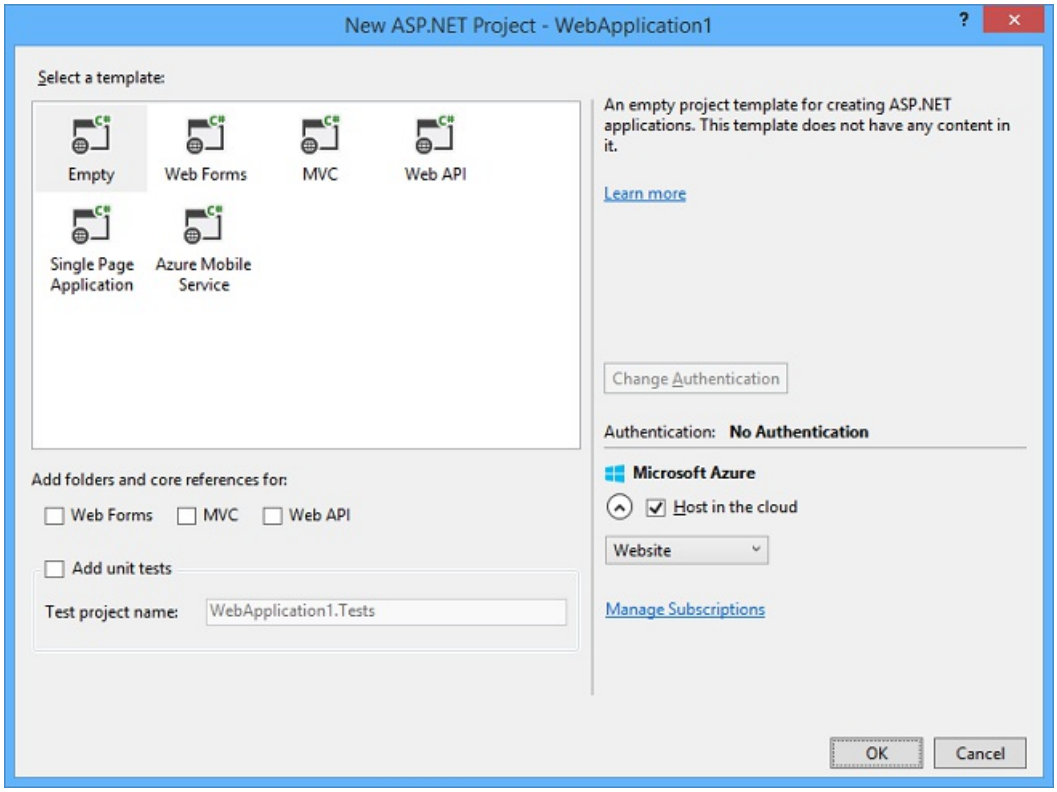


You can choose **Cloud** in the left pane to create an Azure Cloud Service, Azure Mobile Service, or Azure WebJob. This topic doesn't cover those templates.

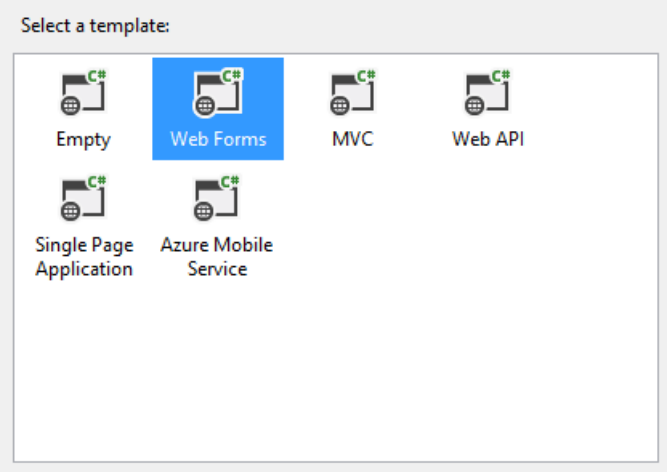
3. In the right pane, click the **Add Application Insights to Project** check box if you want health and usage monitoring for your application. For more information, see Monitor performance in web applications.

4. Specify project **Name**, **Location**, and other options, and then click **OK**.

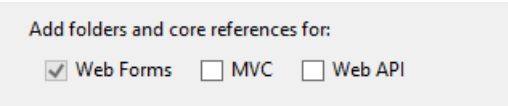
   The **New ASP.NET Project** dialog appears.



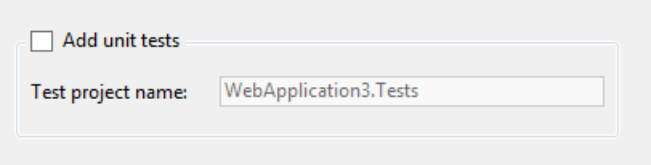
5. Click a template.



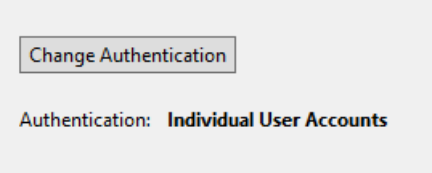
6. If you want to add support for additional frameworks not included in the template, click the appropriate check box. (In the example shown, you could add MVC and/or Web API to a Web Forms project.)

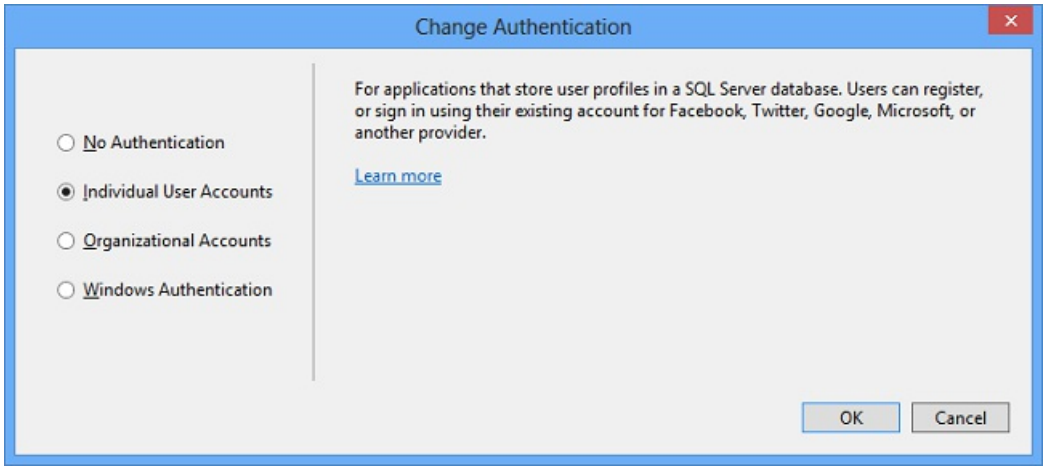


7. If you want to add a unit test project, click **Add unit tests**.



8. If you want a different authentication method than what the template provides by default, click **Change Authentication**.





Create a web app or virtual machine in Azure

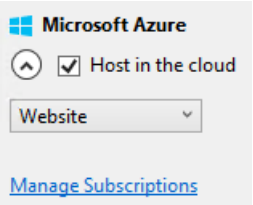
Visual Studio includes features that make it easy to work with Azure services for hosting web applications. For example, you can do all of the following right from the Visual Studio IDE:

- Create and manage web apps or virtual machines that make your application available over the Internet.
- View logs created by the application as it runs in the cloud.
- Run in debug mode remotely while the application runs in the cloud.

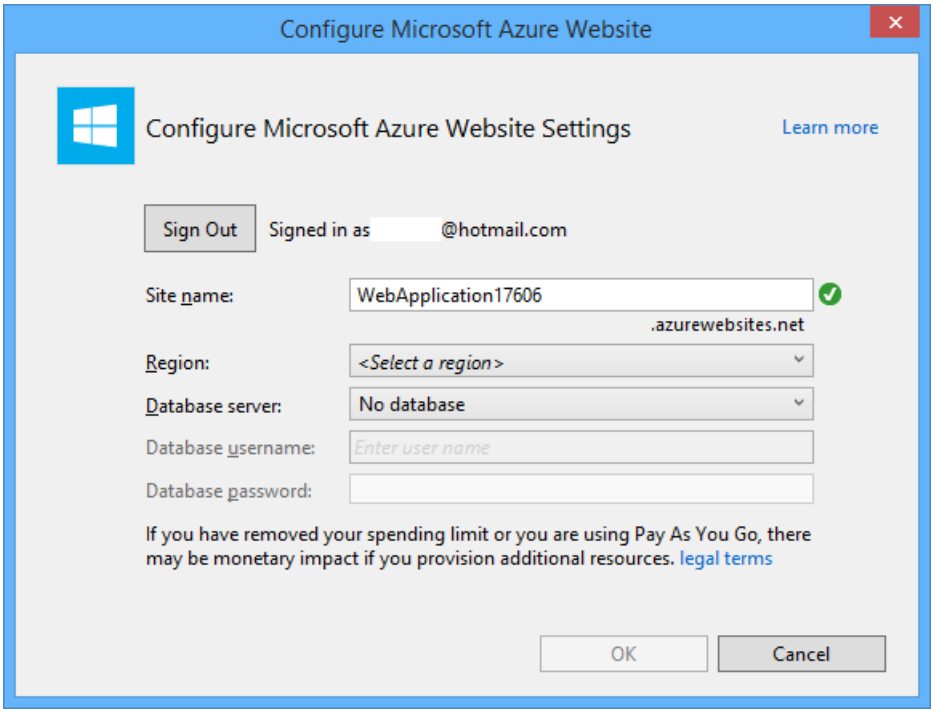
- View and manage other Azure services such as SQL databases.

You can create an Azure account that includes basic services such as web apps for free, and if you are an MSDN subscriber you can activate benefits that give you monthly credits toward additional Azure services.

By default the **New ASP.NET Project** dialog box enables you to create a web app or virtual machine for a new web project. If you don't want to create a new web app or virtual machine, clear the **Host in the cloud** check box.



The check box caption might be **Host in the cloud** or **Create remote resources**, and in either case the effect is the same. If you leave the check box selected, Visual Studio creates a web app in Azure App Service by default. You can use the drop-down box to change that to a **Virtual Machine** if you prefer. If you're not already signed in to Azure, you're prompted for Azure credentials. After you sign in, a dialog box enables you to configure the resources that Visual Studio will create for your project. The following illustration shows the dialog for a web app; different options appear if you choose to create a virtual machine.



For more information about how to use this process for creating Azure resources, see Get Started with Azure and ASP.NET and Creating a virtual machine for a web site with Visual Studio.

The remainder of this article provides more information about the available templates and their options. The article also introduces Bootstrap, the layout and theming framework used in the templates.

Visual Studio 2013 Web Project Templates

Visual Studio uses templates to create web projects. A project template can create files and folders in the new project, install NuGet packages, and provide sample code for a rudimentary working application. Templates implement the latest web standards and are intended to demonstrate best practices for how to use ASP.NET technologies as well as give you a jump start on creating your own application.

Visual Studio 2013 provides the following choices for web project templates for projects that target .NET 4.5 or later versions of the .NET framework:

- Empty template
- Web Forms template
- MVC template
- Web API template
- Single Page Application template
- Azure Mobile Service template
- Visual Studio 2012 Templates

You can also install a Visual Studio extension that provides a Facebook template.

For information about how to create projects that target .NET 4, see Visual Studio 2012 Templates later in this topic.

For information about how to create ASP.NET applications for mobile clients, see Mobile Support in ASP.NET.

Empty Template

The Empty template provides the bare minimum folders and files for an ASP.NET web app, such as a project file (*.csproj* or *.vbproj*) and a *Web.config* file. You can add support for Web Forms, MVC, and/or Web API by using the check boxes under the **Add folders and core references for:** label.

For the Empty template no authentication options are available. Authentication functionality is implemented in sample applications, and the Empty template does not create a sample application.

Web Forms Template

The Web Forms framework provides the following features that let you rapidly build web sites that are rich in UI and data access features:

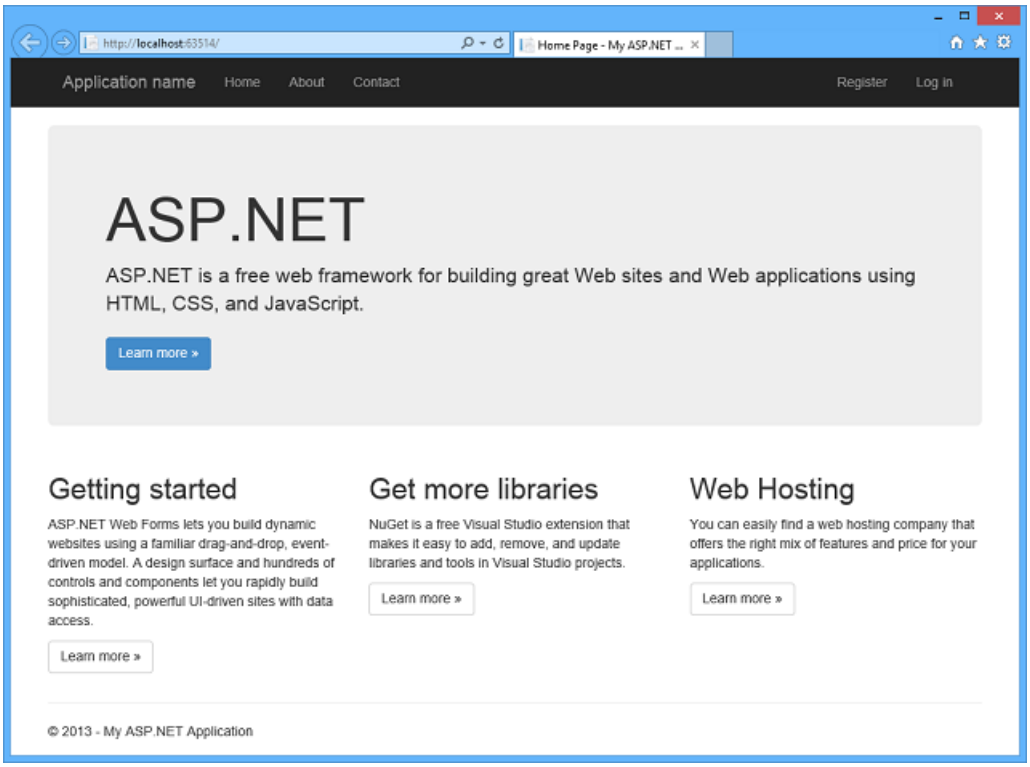
- A WYSIWYG designer in Visual Studio.
- Server controls that render HTML and that you can customize by setting properties and styles.
- A rich assortment of controls for data access and data display.
- An event model that exposes events which you can program like you would program a client application such as WPF.
- Automatic preservation of state (data) between HTTP requests.

In general, creating a Web Forms application requires less programming effort than creating the same application by using the ASP.NET MVC framework. However, Web Forms is not just for rapid application development. There are many complex commercial applications and frameworks built on top of Web Forms.

Because a Web Forms page and the controls on the page automatically generate much of the markup that's sent to the browser, you don't have the kind of fine-grained control over the HTML that ASP.NET MVC offers. The declarative model for configuring pages and controls minimizes the amount of code you have to write but hides some of the behavior of HTML and HTTP. For example, it's not always possible to specify exactly what markup might be generated by a control.

The Web Forms framework doesn't lend itself as readily as ASP.NET MVC to patterns-based development practices such as test-driven development, separation of concerns, inversion of control, and dependency injection. If you want to write code factored that way, you can; it's just not as automatic as it is in the ASP.NET MVC framework. The ASP.NET Web Forms MVP project shows an approach that facilitates separation of concerns and testability while maintaining the rapid development that Web Forms was built to deliver. Microsoft SharePoint is built on Web Forms MVP.

The Web Forms template creates a sample Web Forms application that uses Bootstrap to provide responsive design and theming features. The following illustration shows the home page.



For more information about Web Forms, see ASP.NET Web Forms. For information about what the Web Forms template does for you, see Building a basic Web Forms application using Visual Studio 2013.

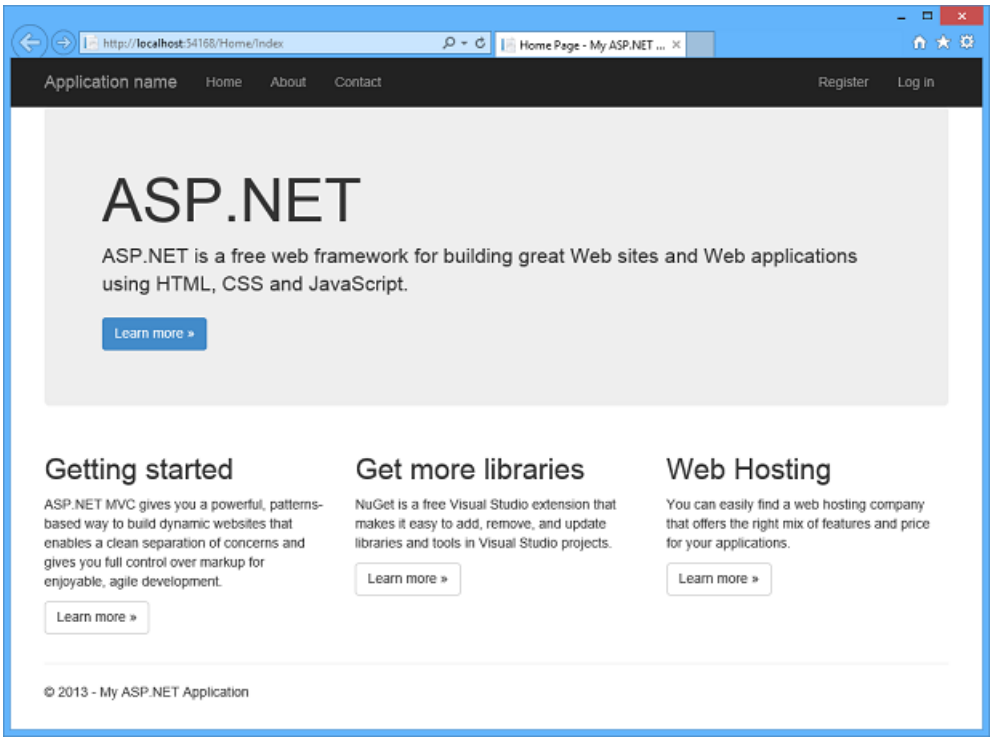
MVC Template

ASP.NET MVC was designed to facilitate patterns-based development practices such as test-driven development, separation of concerns, inversion of control, and dependency injection. The framework encourages separating the business logic layer of a web application from its presentation layer. By dividing the application into models (M), views (V), and controllers (C), ASP.NET MVC can make it easier to manage complexity in larger applications.

With ASP.NET MVC, you work more directly with HTML and HTTP than in Web Forms. For example, Web Forms can automatically preserve state between HTTP requests, but you have to code that explicitly in MVC. The advantage of the MVC model is that it enables you to take complete control over exactly what your application is doing and how it behaves in the web environment. The disadvantage is that you have to write more code.

MVC was designed to be extensible, providing power developers the ability to customize the framework for their application needs. In addition, the ASP.NET MVC source code is available under an OSI license.

The MVC template creates a sample MVC 5 application that uses Bootstrap to provide responsive design and theming features. The following illustration shows the home page.



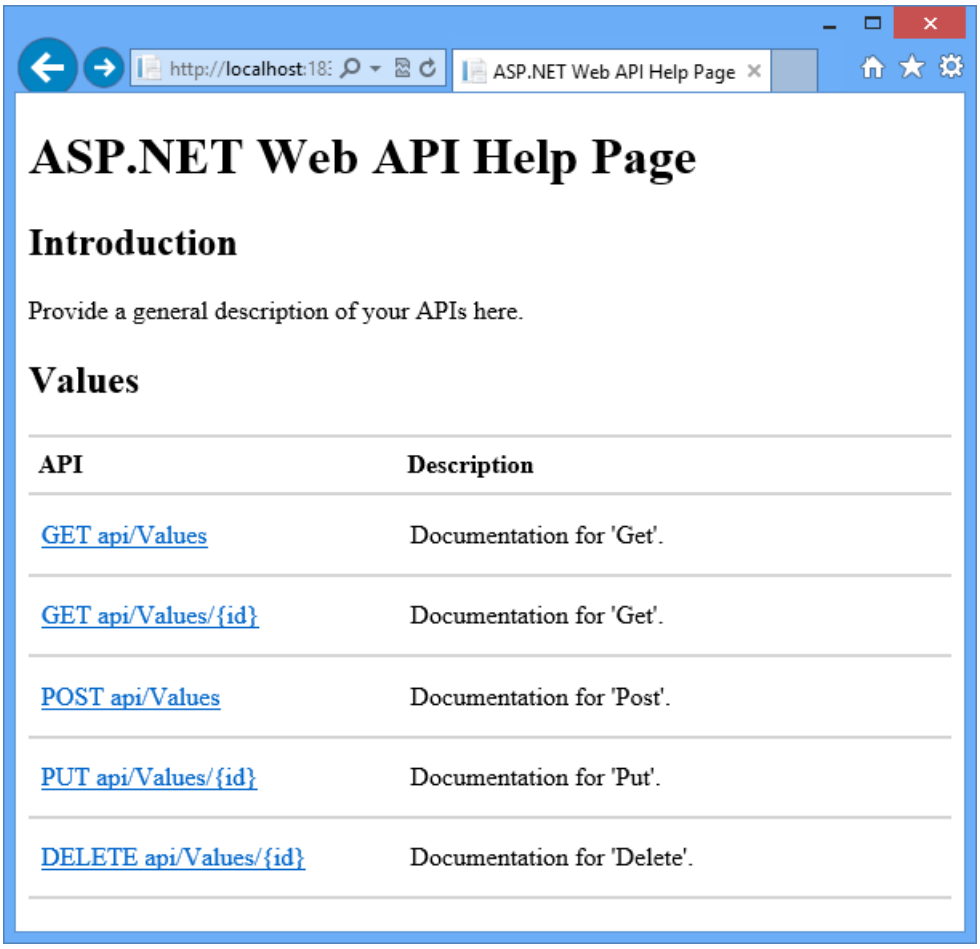
For more information about MVC, see ASP.NET MVC. For information about how to select the MVC 4 template, see Visual Studio 2012 templates later in this article.

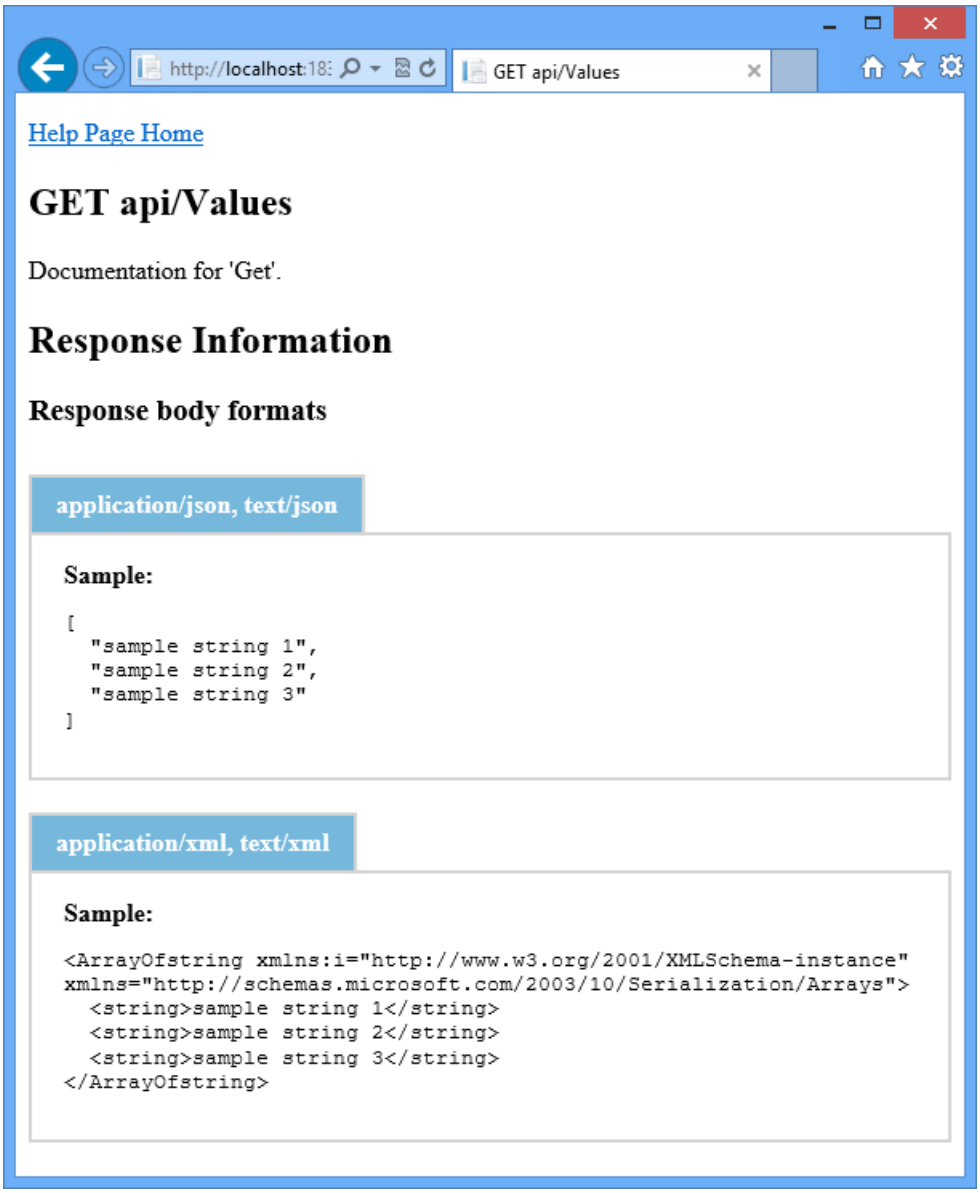
Web API Template

The Web API template creates a sample web service based on Web API, including API help pages based on MVC.

ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Web API is an ideal platform for building RESTful services on the .NET Framework.

The Web API template creates a sample web service. The following illustrations show sample help pages.


http://localhost:18:
ASP.NET Web API Help Page
ASP.NET Web API Help Page
Introduction
Provide a general description of your APIs here.
Values
API
Description
GET api/Values
Documentation for 'Get'.
GET api/Values/{id}
Documentation for 'Get'.
POST api/Values
Documentation for 'Post'.
PUT api/Values/{id}
Documentation for 'Put'.
DELETE api/Values/{id}
Documentation for 'Delete'.



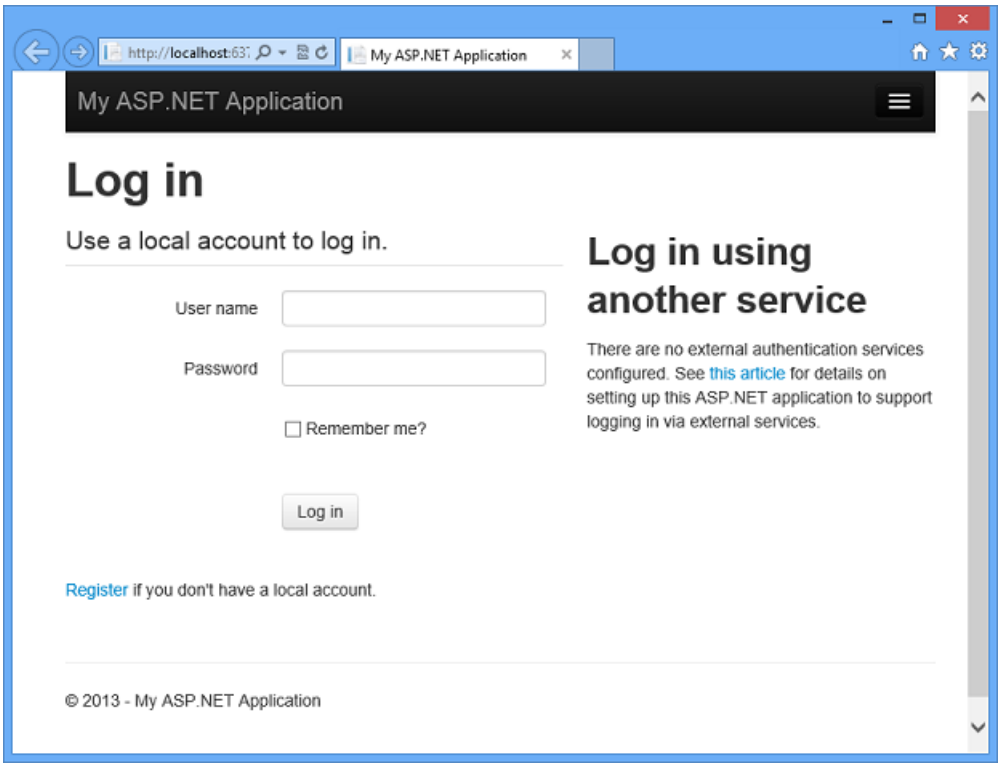
For more information about Web API, see ASP.NET Web API.

Single Page Application Template

The Single Page Application (SPA) template creates a sample application that uses JavaScript, HTML 5, and KnockoutJS on the client, and ASP.NET Web API on the server.

The only authentication option for the SPA template is Individual User Accounts.

The following illustration shows the initial state of the sample application that the SPA template builds.



For information about how to create an application by using the SPA template, see Web API - External Authentication Services.

For more information about ASP.NET Single Page Applications, and about additional SPA templates that use JavaScript frameworks other than KnockoutJS, see the following resources:

- ASP.NET Single Page Application.
- Understanding Security Features in the SPA Template for VS2013 RC
- Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET

Facebook Template

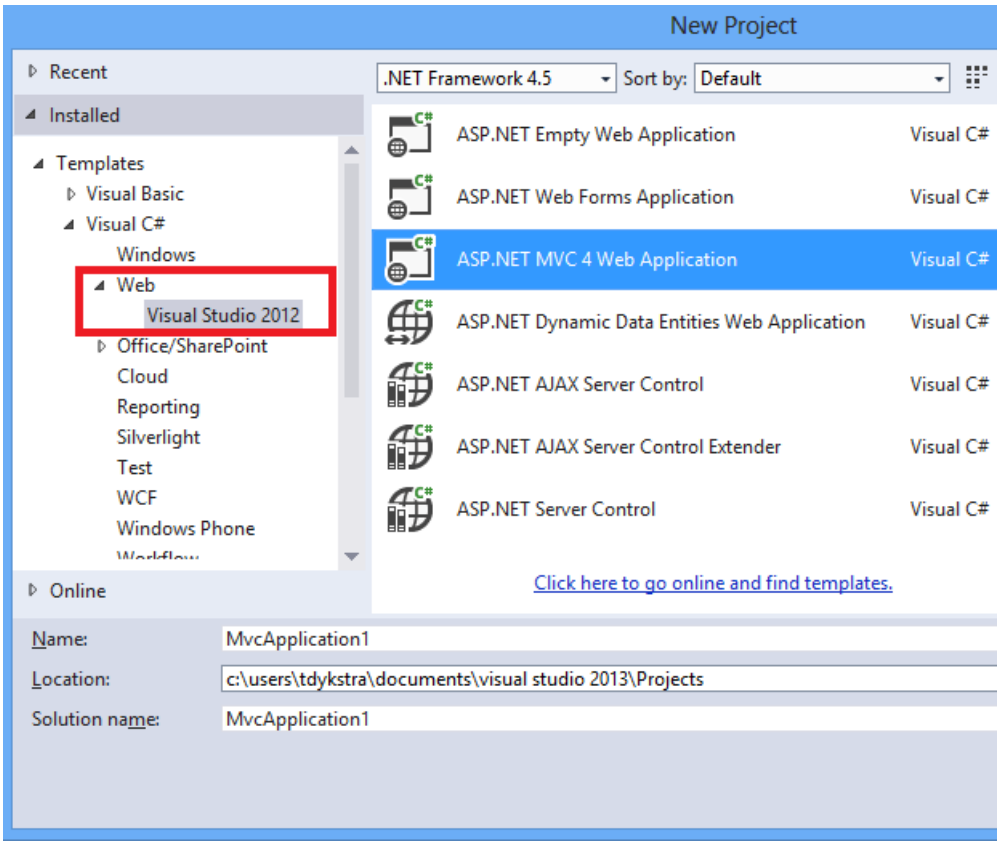
You can install a Visual Studio extension that provides a Facebook template. This template creates a sample application that is designed to run inside the Facebook web site. It is based on ASP.NET MVC and uses Web API for real-time update functionality.

No authentication options are available for the Facebook template because Facebook applications run within the Facebook site and rely on Facebook's authentication.

For more information about ASP.NET Facebook applications, see Updating the MVC Facebook API.

Visual Studio 2012 Templates

The Visual Studio 2013 web project creation dialog does not provide access to some templates that were available in Visual Studio 2012. If you want to use one of these templates, you can click the Visual Studio 2012 node in the left pane of the Visual Studio New Project dialog box.

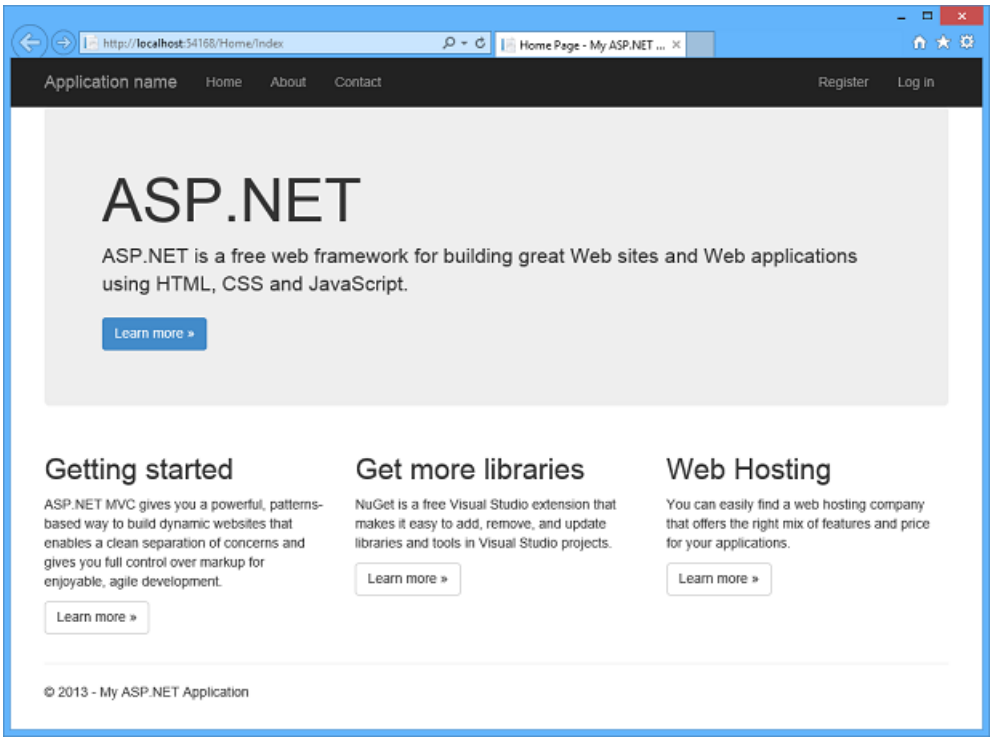


The **Visual Studio 2012** node lets you choose the following web templates that you don't have access to in the default list of templates for Visual Studio 2013:

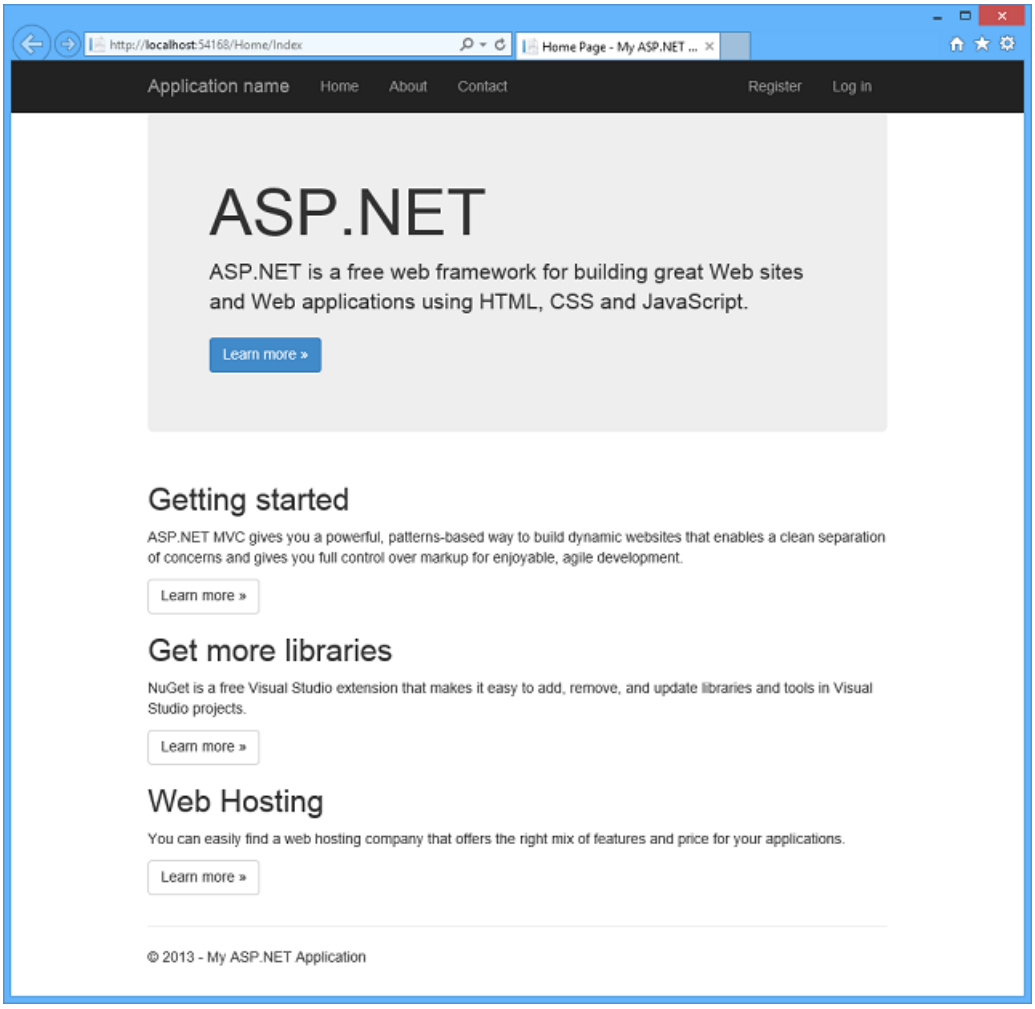
- ASP.NET MVC 4 Web Application
- ASP.NET Dynamic Data Entities Web Application
- ASP.NET AJAX Server Control
- ASP.NET AJAX Server Control Extender
- ASP.NET Server Control

Bootstrap in the Visual Studio 2013 web project templates

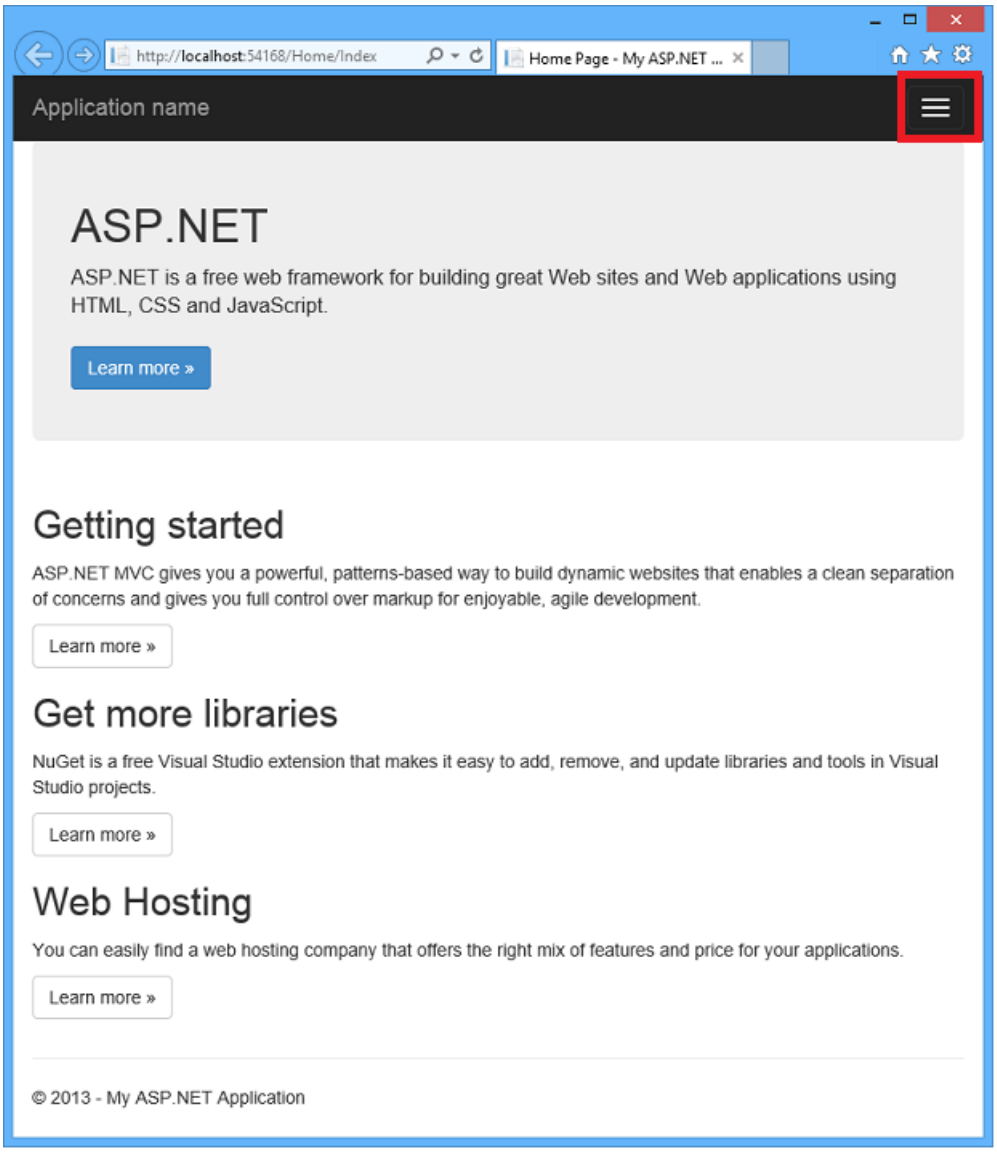
The Visual Studio 2013 project templates use Bootstrap, a layout and theming framework created by Twitter. Bootstrap uses CSS3 to provide responsive design, which means layouts can dynamically adapt to different browser window sizes. For example, in a wide browser window the home page created by the Web Forms template looks like the following illustration:

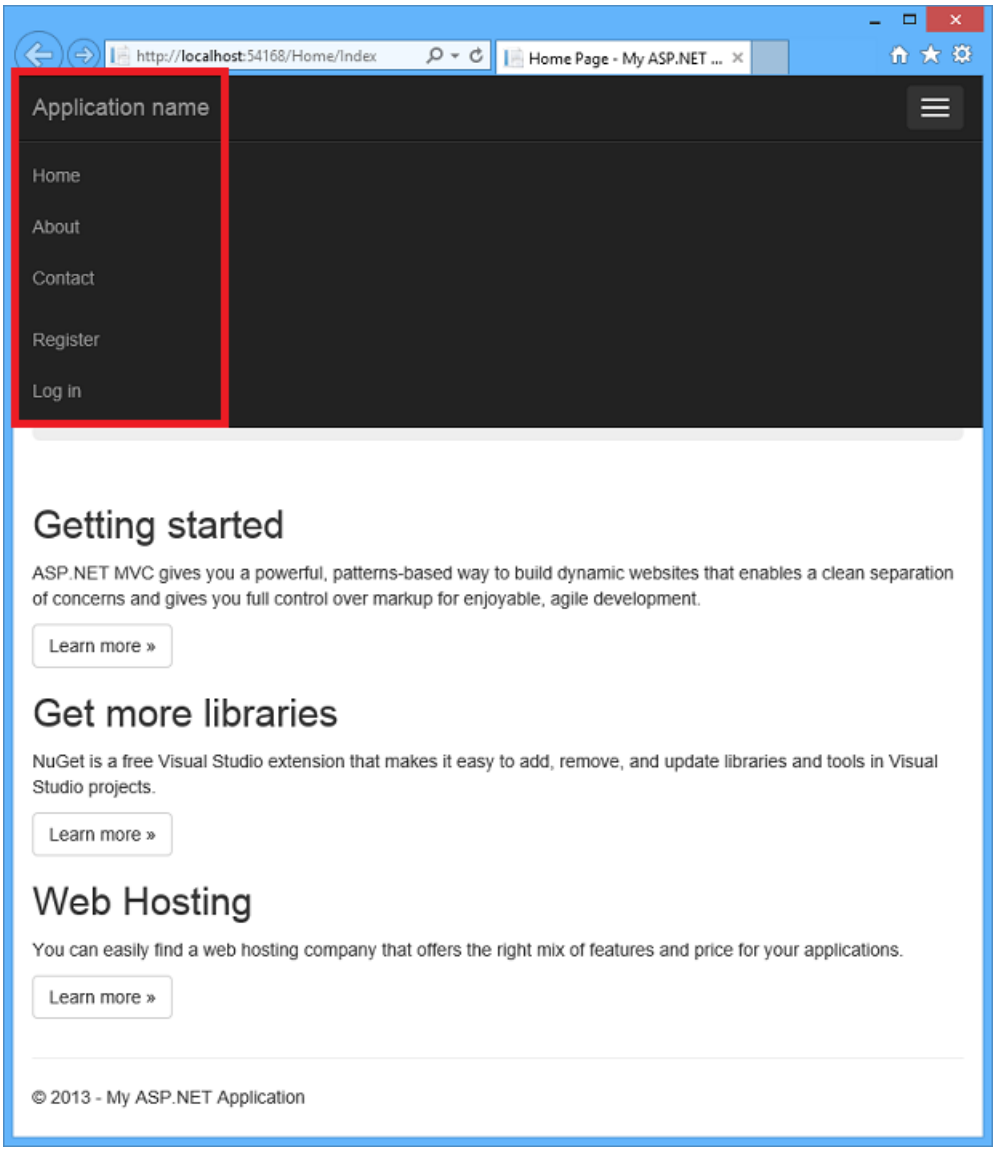


Make the window narrower, and the horizontally arranged columns move into vertical arrangement:



Narrow the window a little more, and the horizontal top menu turns into an icon that you can click to expand into a vertically oriented menu:

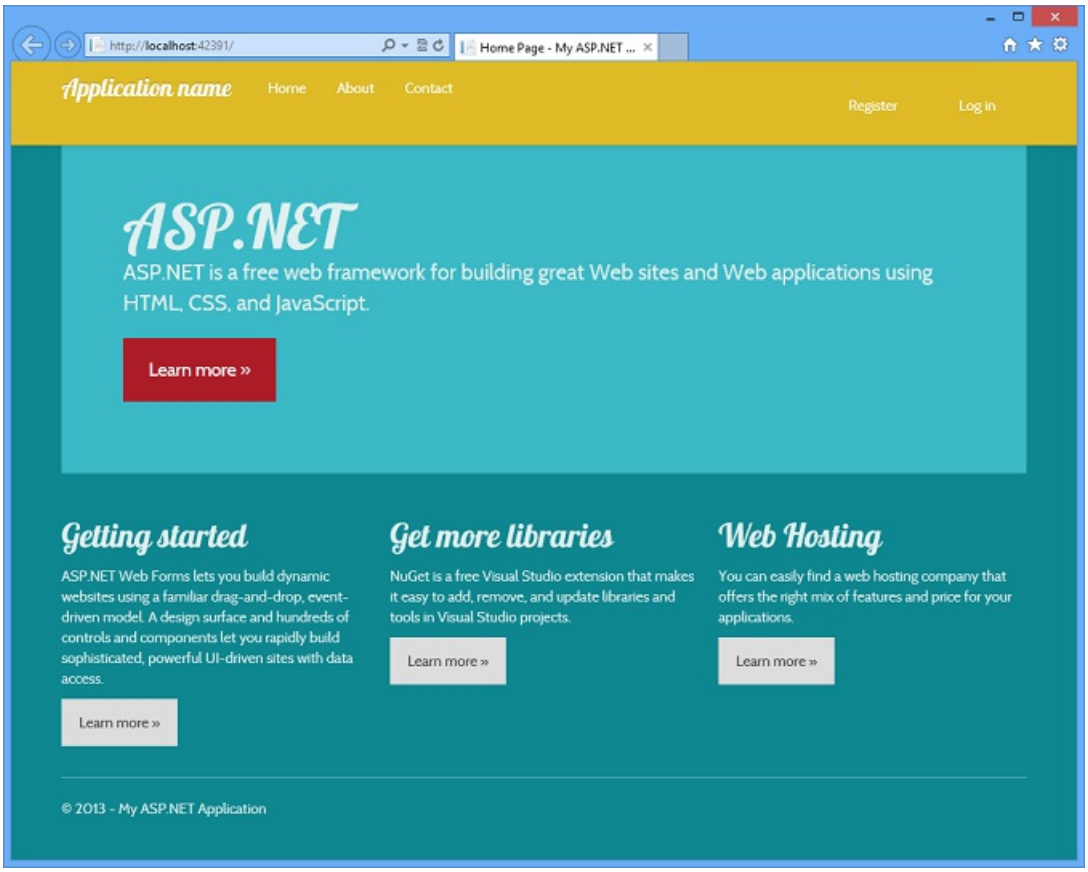

http://localhost:54168/Home/Index
Home Page - My ASP.NET ...
Application name
ASP.NET
ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.
Learn more »
Getting started
ASP.NET MVC gives you a powerful, patterns-based way to build dynamic websites that enables a clean separation of concerns and gives you full control over markup for enjoyable, agile development.
Learn more »
Get more libraries
NuGet is a free Visual Studio extension that makes it easy to add, remove, and update libraries and tools in Visual Studio projects.
Learn more »
Web Hosting
You can easily find a web hosting company that offers the right mix of features and price for your applications.
Learn more »
© 2013 - My ASP.NET Application



You can also use Bootstrap's theming feature to easily effect a change in the application's look and feel. For example, you can do the following steps to change the theme.

1. In your browser, go to http://Bootswatch.com, choose a theme, and then click **Download**. (This downloads *bootstrap.min.css* by default; if you want to examine the CSS code, get *bootstrap.css* instead of the minified version.)
2. Copy the contents of the downloaded CSS file.
3. In Visual Studio, create a new **Style Sheet** file named *bootstrap-theme.css* in the *Content* folder and paste the downloaded CSS code into it.
4. Open *App_Start/Bundle.config* and change *bootstrap.css* to *bootstrap-theme.css*.

Run the project again, and the application has a new look. The following illustration shows the effect of the Amelia theme:

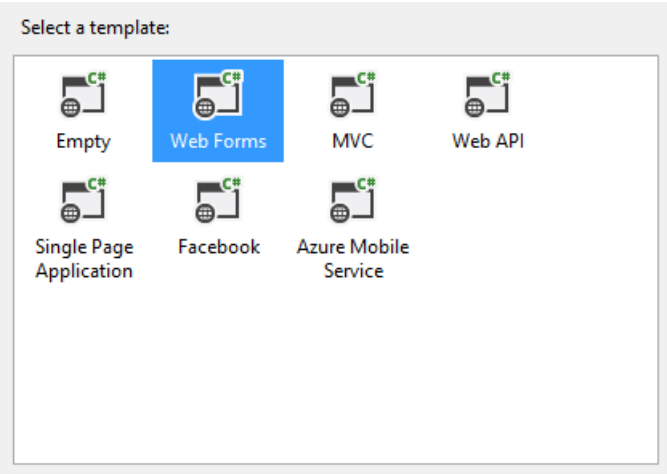


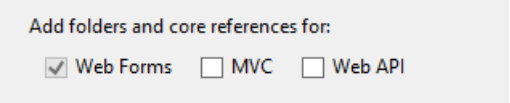
Many Bootstrap themes are available, both free and premium versions. Bootstrap also offers a wide variety of UI components, such as drop-downs, button groups, and icons. For more information about Bootstrap, see the Bootstrap site.

If you use the Web Forms designer in Visual Studio, note that the designer doesn't support CSS3, so it doesn't accurately show all the effects of Bootstrap themes or responsive layout changes. However, the Web Forms pages will display correctly when viewed with a browser.

Adding Support for Additional Frameworks

When you select a template, the check box for the framework(s) used by the template is automatically selected. For example, if you select the **Web Forms** template, the **Web Forms** check box is selected and you can't clear it.





You can select the check box for a framework that isn't included in the template in order to add support for that framework when the project is created. For example, to enable the use of Web Forms *.aspx* pages when you've

selected the MVC template, select the **Web Forms** check box. Or to enable MVC when you're using the Web Forms template, click the **MVC** check box. Adding a framework enables design-time as well as run-time support. For example, if you add MVC support to a Web Forms project, you will be able to scaffold controllers and views.

If you combine Web Forms and MVC in a project and enable Friendly URLs in Web Forms, there might be unexpected routing problems where one URL has multiple possible targets. The routes that are defined first will take precedence. For example, if you have a `Home` controller and a *Home.aspx* page, the `http://contoso.com/home` URL will go to *Home.aspx* if you call the `EnableFriendlyUrls` method before you call the `MapRoute` method in *RouteConfig.cs*, or the same URL will go to the default view for your `Home` controller if you call `MapRoute` before `EnableFriendlyUrls`.

Adding a framework does not add any sample application functionality. For example, if you add Web Forms support when you've selected the MVC template, no *Default.aspx* home page file is created. Only the folders, files, and references required to support the framework are added. For that reason, adding frameworks doesn't change authentication options, which are implemented by code in sample applications created by the templates. For example, if you select the Empty template and add Web Forms or MVC support, the **Configure Authentication** button will still be disabled.

The following sections describe briefly the effect of each check box.

Add Web Forms Support

Creates empty *App_Data* and *Models* folders and a *Global.asax* file. These are already created by all templates other than the Empty template, so selecting the Web Forms check box makes no difference for other templates.

The Web Forms template enables Friendly URLs by default, but when you add Web Forms support to other templates by selecting the Web Forms check box Friendly URLs are not automatically enabled.

Add MVC Support

Installs MVC, Razor, and WebPages NuGet packages, creates empty *App_Data*, *Controllers*, *Models*, and *Views* folders, creates *App_Start* folder with *RouteConfig.cs* file, and creates *Global.asax* file.

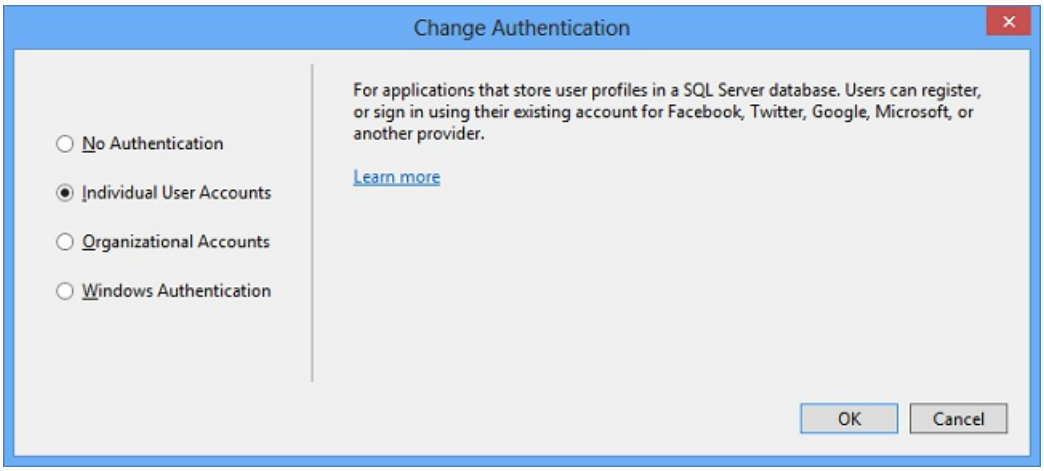
Add Web API Support

Installs WebApi and Newtonsoft.Json NuGet packages, creates empty *App_Data*, *Controllers*, and *Models* folders, creates *App_Start* folder with *WebApiConfig.cs* file, and creates *Global.asax* file.

Authentication Methods

Visual Studio 2013 offers several authentication options for the Web Forms, MVC, and Web API templates:

- No Authentication
- Individual User Accounts (ASP.NET Identity, formerly known as ASP.NET membership)
- Organizational Accounts (Windows Server Active Directory or Azure Active Directory)
- Windows Authentication (Intranet)



No Authentication

If you select **No Authentication**, the sample application will contain no web pages for logging in, no UI indicating who is logged in, no entity classes for a membership database, and no connection string for a membership database.

Individual User Accounts

If you select **Individual User Accounts**, the sample application will be configured to use ASP.NET Identity (formerly known as ASP.NET membership) for user authentication. ASP.NET Identity enables a user to register an account, by creating a username and password on the site or by signing in with social providers such as Facebook, Google, Microsoft Account, or Twitter. The default data store for user profiles in ASP.NET Identity is a SQL Server LocalDB database, which you can deploy to SQL Server or Azure SQL Database for the production site.

In Visual Studio 2013 these features are the same as in Visual Studio 2012, but the underlying code for the ASP.NET membership system has been rewritten. Advantages of the new code base include the following:

- The new membership system is based on OWIN rather than the ASP.NET Forms Authentication module. This means that you can use the same authentication mechanism whether you're using Web Forms or MVC in IIS, or you're self-hosting Web API or SignalR.
- The new membership database is managed by Entity Framework Code First, and all of the tables are represented by entity classes that you can modify. This means that you can easily customize the database schema and profile-related web UI to fit your own needs, and you can easily deploy your updates using Code First Migrations.

The new membership system is implemented automatically in the new templates, and it can be implemented manually in any project that targets .NET 4.5 or later.

ASP.NET Identity is a good choice if you are creating an Internet web site which is mainly for external customers. If your organization uses Active Directory or Office 365 and you want to create a project that enables single-sign-on for employees and business partners, the **Organizational Accounts** option might be a better choice.

For more information about the Individual User Accounts option, see the following resources:

- www.asp.net/identity. Documentation about ASP.NET Identity on the ASP.NET web site.
- Create an ASP.NET MVC 5 App with Facebook and Google OAuth2 and OpenID Sign-on. Also shows how to customize user profile data.
- Web API - External Authentication Services
- Adding External Logins to your ASP.NET application in Visual Studio 2013

Organizational Accounts

If you select **Organizational Accounts**, the sample application will be configured to use Windows Identity Foundation (WIF) for authentication based on user accounts in Azure Active Directory (Azure AD, which includes Office 365) or Windows Server Active Directory. For more information, see Organizational account authentication

options later in this topic.

Windows Authentication

If you select **Windows Authentication**, the sample application will be configured to use the Windows Authentication IIS module for authentication. The application will display the domain and user ID of the Active directory or local machine account that is logged into Windows but won't include user registration or log-in UI. This option is intended for Intranet web sites.

Alternatively, you can create an Intranet site that uses AD authentication by choosing the On-Premises option under Organizational Accounts. The On-Premises option uses Windows Identity Foundation (WIF) instead of the Windows Authentication module. Some additional steps are required in order to set up the On-Premises option, but WIF enables features that aren't available with the Windows Authentication module. For example, with WIF you can configure application access in Active Directory and query directory data.

Organizational account authentication options

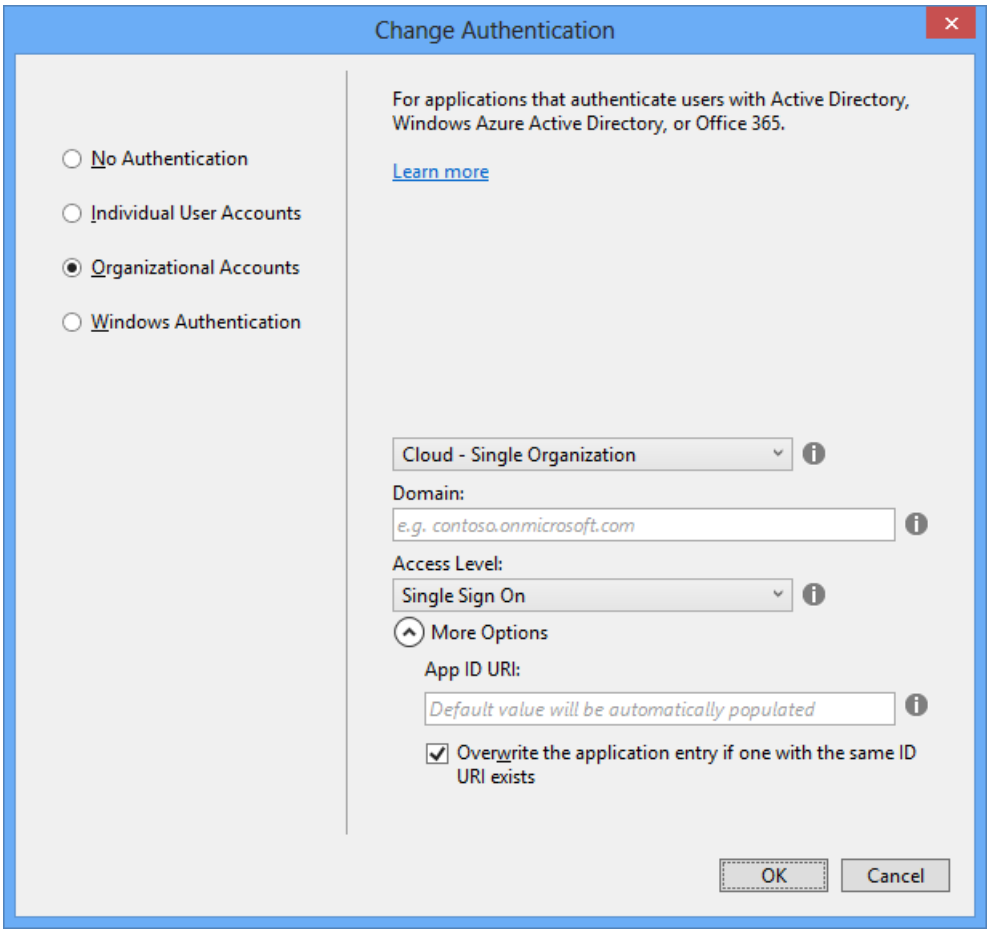
The **Configure Authentication** dialog gives you several options for Azure Active Directory (Azure AD, which includes Office 365) or Windows Server Active Directory (AD) account authentication:

- Cloud - Single Organization (Azure AD, or AD using directory integration with Azure AD)
- Cloud - Multi Organization (Azure AD, or AD using directory integration with Azure AD)
- On-Premises (AD)

If you want to try one of the Azure AD options but don't have an account yet, click here to sign up for a Azure AD account.

> **NOTE**
>
> If you choose one of the Azure AD options, your project requires a database and you have to sign in to a global administrator account for your Azure AD tenant. Enter the name and password for an organizational account (for example, admin@contoso.onmicrosoft.com) that has administrative permissions for your Azure AD tenant.
>
> **Don't enter credentials for a Microsoft account (for example, contoso@hotmail.com) in the sign-in dialog box.**

Cloud - Single Organization Authentication



Choose this option if you want to enable authentication for user accounts that are defined in one Azure AD tenant. For example, the site is contoso.com and it will be made available to employees of the Contoso Company who are in the contoso.onmicrosoft.com tenant. You won't be able to configure Azure AD to allow users from other tenants to access the application.

Domain

Enter the Azure AD domain that you want to set up the application in, for example: `contoso.onmicrosoft.com`. If you have a custom domain, such as `contoso.com` instead of `contoso.onmicrosoft.com`, you can enter that here.

Access Level

If the application needs to query or update directory information by using the Graph API, choose **Single Sign-On, Read Directory Data** or **Single Sign-On, Read and Write Directory Data**. Otherwise, choose **Single Sign-On**. For more information, see Application Access Levels and Using the Graph API to Query Azure AD.

Application ID URI

By default, the template creates an application ID URI for you by appending the project name to the Azure AD domain. For example, if the project name is `Example` and the domain is `contoso.onmicrosoft.com`, the application ID URI becomes `https://contoso.onmicrosoft.com/Example`. If you want to manually specify the application ID URI, expand the **More Options** section and enter the application ID URI in the text box. The application ID URI must begin with `https://`.

By default, if an application that is already provisioned in Azure AD has the same application ID URI as the one that Visual Studio is using for the project, the project will be connected to the existing application instead of provisioning a new one. If you want a new application to be provisioned in that case, clear the **Overwrite the application entry if one with the same ID already exists** check box.

If the **Overwrite** check box is cleared, and Visual Studio finds an existing application with the same application ID URI, it creates a new URI by appending a number to the URI it was going to use. For example, suppose the project name is `Example`, you leave the text box blank, you clear the **Overwrite** check box, and the Azure AD tenant already has an application with the URI `https://contoso.onmicrosoft.com/Example`. In that case, a new application

will be provisioned with an application ID URI like `https://contoso.onmicrosoft.com/Example_20130619330903`.

Provisioning the application in Azure AD

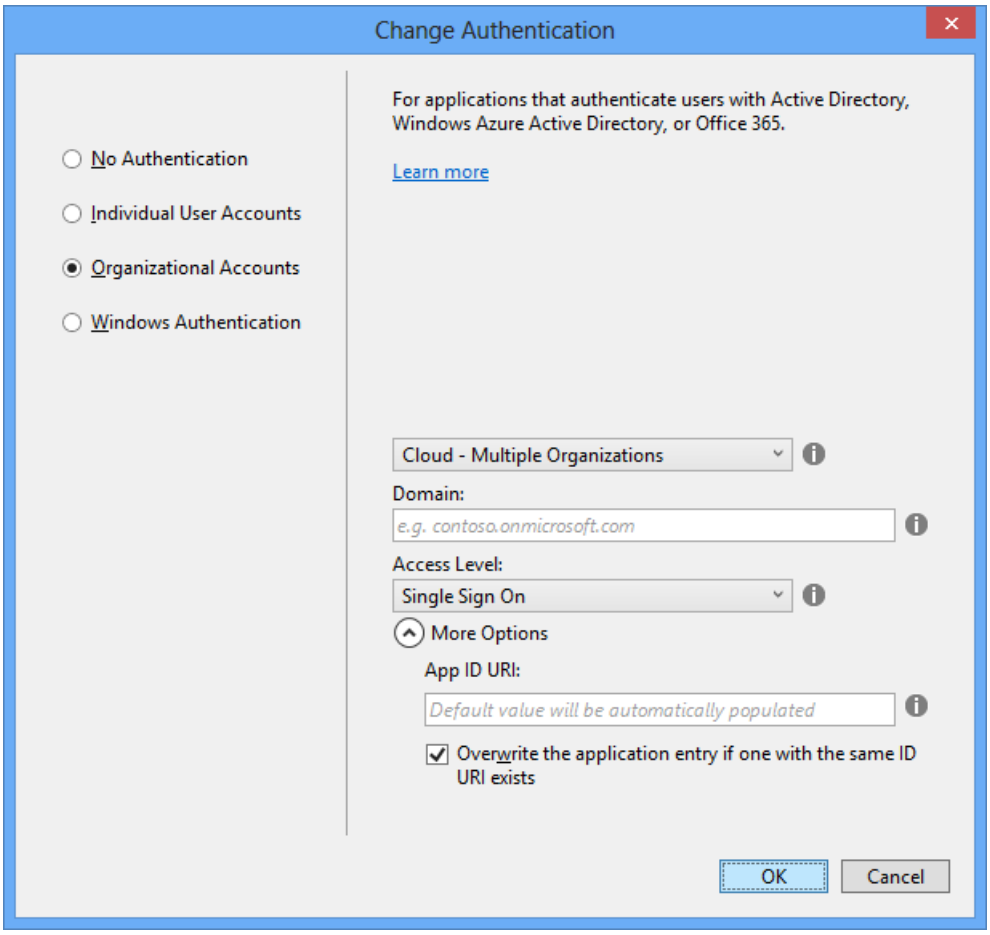
In order to provision the application in Azure AD or connect the project to an existing application, Visual Studio needs the credentials of a global administrator for the domain. When you click **OK** in the **Configure Authentication** dialog box, you are prompted for the user name and password of a global administrator for the domain you specified. Later, when you click **Create Project** in the **New ASP.NET Project** dialog, Visual Studio provisions the application in Azure AD. Note that as part of this process Visual Studio embeds client secret values in the Web.config file that expire one year after creation.

For information about how to create applications that use **Cloud - Single Organization** authentication, see the following resources:

- Azure Authentication
- Adding Sign-On to Your Web Application Using Azure AD
- Developing ASP.NET Apps with Azure Active Directory
- Secure ASP.NET Web API with Azure AD and Microsoft OWIN Components

The tutorials have not yet been updated for Visual Studio 2013; some of what the tutorials direct you to do manually is automated in Visual Studio 2013.

Cloud - Multi Organization Authentication



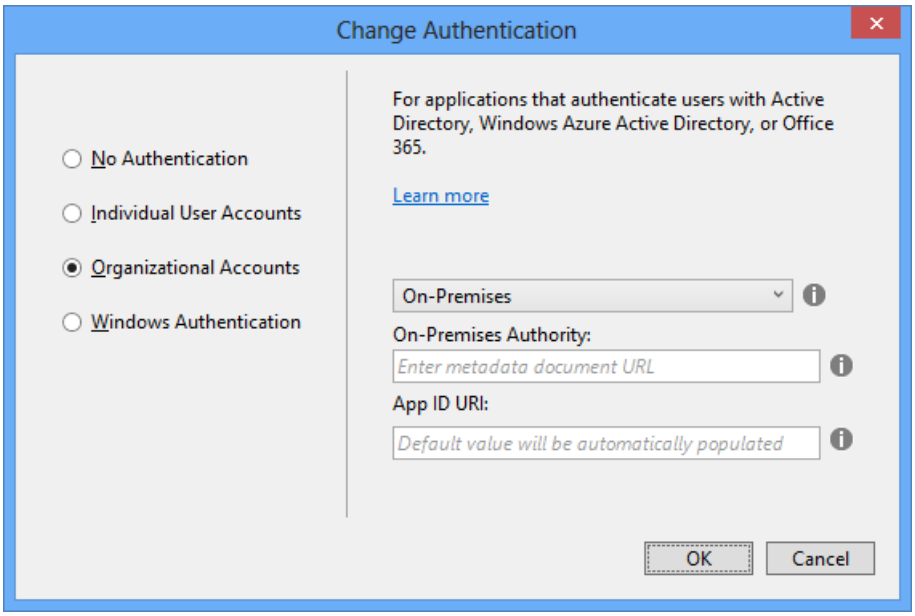
Choose this option if you want to enable authentication for user accounts that are defined in multiple Azure AD tenants. For example, the site is contoso.com and it will be made available to employees of the Contoso Company who are in the contoso.onmicrosoft.com tenant, and employees of the Fabrikam Company who are in the fabrikam.onmicrosoft.com tenant.

The settings that you enter and the application provisioning step are similar to single organization authentication.

For information about how to create applications that use **Cloud - Multi Organization** authentication, see the following resources:

- Easy Web App Integration with Azure Active Directory, ASP.NET & Visual Studio on the Active Directory Team blog.
- Developing Multi-Tenant Web Applications with Azure AD tutorial. The tutorial hasn't yet been updated for Visual Studio 2013; some of what the tutorial directs you to do manually is automated in Visual Studio 2013.
- You Have to Sign Up With Your Own Multiple Organizations ASP.NET App Before You Can Sign In. Blog by Vittorio Bertocci that explains how to resolve a common problem people encounter when creating a project that uses multi-organization authentication.

On-Premises Organizational Authentication



Choose this option if you want to enable authentication for user accounts that are defined in Windows Server Active Directory (AD), and you don't want to use Azure AD. You can use this option to create an Intranet site or an Internet site. For an Internet site, use Active Directory Federation Services (ADFS) to provide access to AD. For more information, see Use the On-Premises Organizational Authentication Option (ADFS) With ASP.NET in Visual Studio 2013.

For an Intranet site, as an alternative you can choose Windows Authentication instead of this option. For the Windows Authentication option you don't have to provide a metadata document URL. However, Windows Authentication does not give you the ability to control application access in Active Directory or to query directory data.

On-Premises Authority

Enter a URL that points to the metadata document. The metadata document contains the coordinates of the authority. The application will use those coordinates to drive the web sign-on flow.

Application ID URI

Provide a unique URI that AD can use to identify this application, or leave blank to let Visual Studio create one.

Next steps

This document has provided some basic help for creating a new ASP.NET web project in Visual Studio 2013. For more information about using for Visual Studio for web development, see https://www.asp.net/visual-studio/.

Visual Studio 2013 Web Editor Features - Browser Link

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Scott Hanselman demonstrates Browser Link in Visual Studio 2013, which allows for two-way communication between Visual Studio and multiple connected browsers.

Watch video (3 minutes)

Visual Studio 2013 Web Editor Features - CSS

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Scott Hanselman demonstrates CSS editing features in Visual Studio 2013 and the Visual Studio Web Essentials extension.

Watch video (3 minutes)

Visual Studio 2013 Web Editor Features - JavaScript

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Scott Hanselman explains the JavaScript editor features in Visual Studio 2013.

Watch video (3 minutes)

Visual Studio 2013 Web Editor Features - Publishing

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Scott Hanselman overviews the web publishing features in Visual Studio 2013.

Watch video (3 minutes)

Visual Studio 2013 Web Editor Features - Page Inspector

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Scott Hanselman shows how the Page Inspector feature in Visual Studio 2013 allows you to work interactively with the rendered page markup, fully integrated with the source code that rendered it.

Watch video (5 minutes)

Visual Studio 2013 Web Editor Features - HTML Editor

2/20/2020 • 2 minutes to read • Edit Online

by Mads Kristensen

Mads Kristensen shows off HTML editor features in Visual Studio 2013, including full IntelliSense for HTML elements, all CSS classes and ID's in your project, HTML entities, rich snippets, drag and drop for media, support for modern libraries like AngularJS, KnockoutJS and Handlebars, and a variety of meta tags.

Watch video (3 minutes)

Visual Studio 2013 Web Editor Features - HTML5

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Scott Hanselman overviews HTML5 features in the Visual Studio web developer tools.

Watch video (4 minutes)

ASP.NET Scaffolding in Visual Studio 2013

2/20/2020 • 3 minutes to read • Edit Online

by Tom FitzMacken

> ASP.NET Scaffolding is a new feature that is included in Visual Studio 2013.

Overview

ASP.NET Scaffolding is a code generation framework for ASP.NET Web applications. Visual Studio 2013 includes pre-installed code generators for MVC and Web API projects. You add scaffolding to your project when you want to quickly add code that interacts with data models. Using scaffolding can reduce the amount of time to develop standard data operations in your project.

By default, Visual Studio 2013 does not support generating code for a Web Forms project, but you can use scaffolding with Web Forms by either adding MVC dependencies to the project or installing an extension. Both approaches are shown below.

Visual Studio 2013 Update 2 (currently RC) provides the ability to extend ASP.NET Scaffolding to meet the requirements of your scenario. With this functionality, you can create a customized scaffolding template and add it to Add New Scaffold dialog. Within the customized template, you specify the code that is generated when adding a scaffolded item. For more information, see Creating a Custom Scaffolder for Visual Studio.

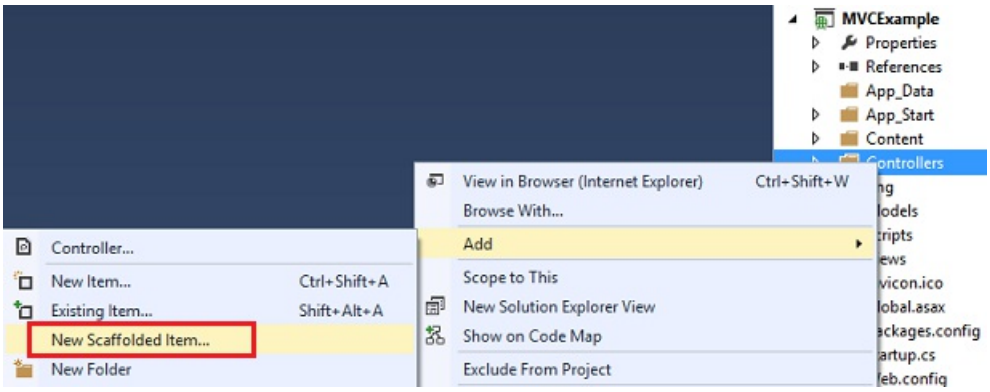
Prerequisites

To use ASP.NET Scaffolding, you must have:

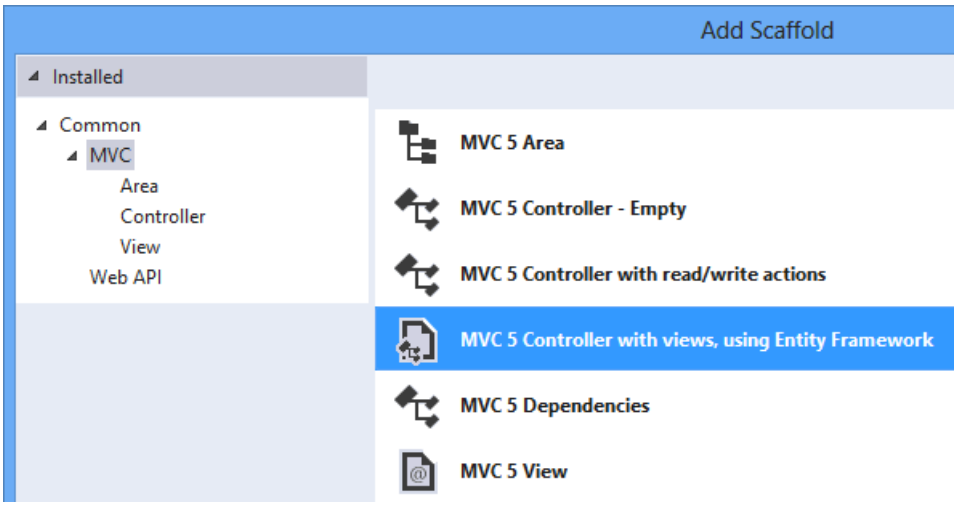
- Microsoft Visual Studio 2013
- Web Developer Tools (part of default Visual Studio 2013 installation)
- ASP.NET Web Frameworks and Tools 2013 (part of default Visual Studio 2013 installation)

Add a scaffolded item to MVC or Web API

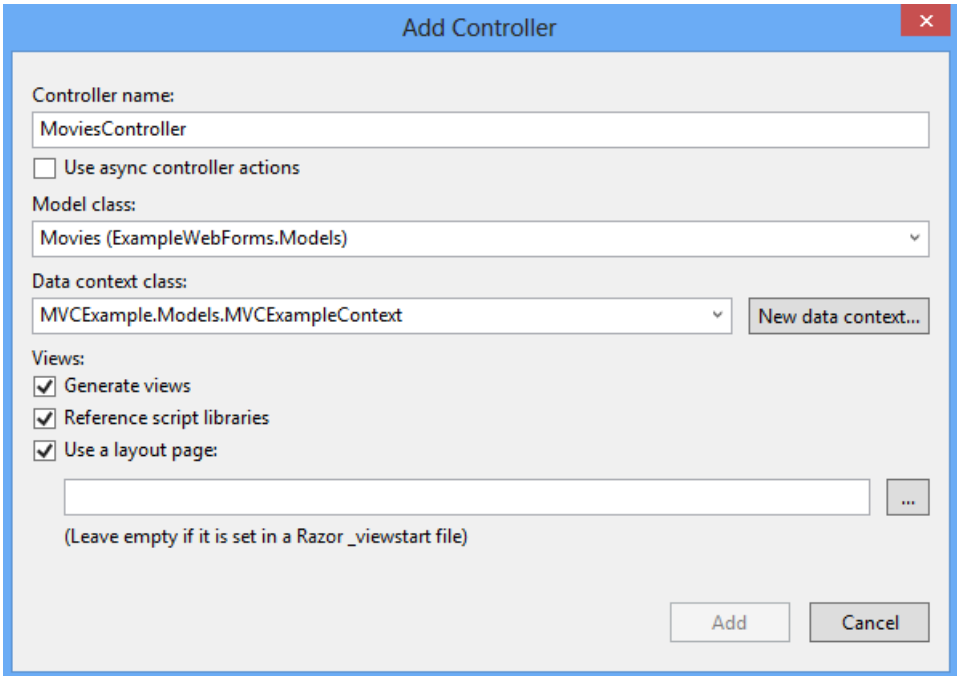
To add a scaffold, right-click either the project or a folder within the project, and select **Add – New Scaffolded Item**, as shown in the following image.



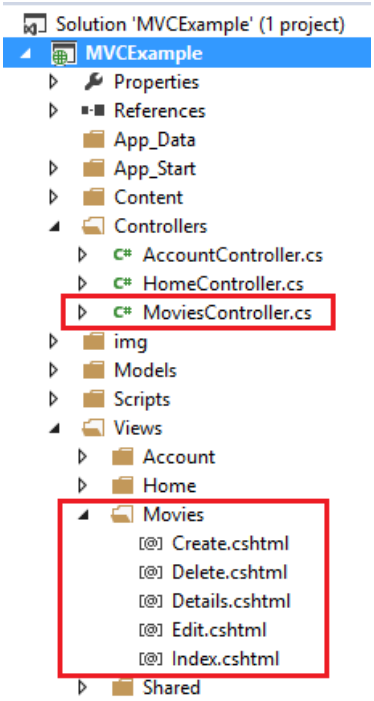
From the **Add Scaffold** window, select the type of scaffold to add.



The **Add Controller** window gives you the opportunity to select options for generating the controller, including whether you want to use the new async features from Entity Framework 6.



The relevant classes and pages are created for your scenario. For example, the following image shows the MVC controller and views that were created through scaffolding for a model class named Movies.

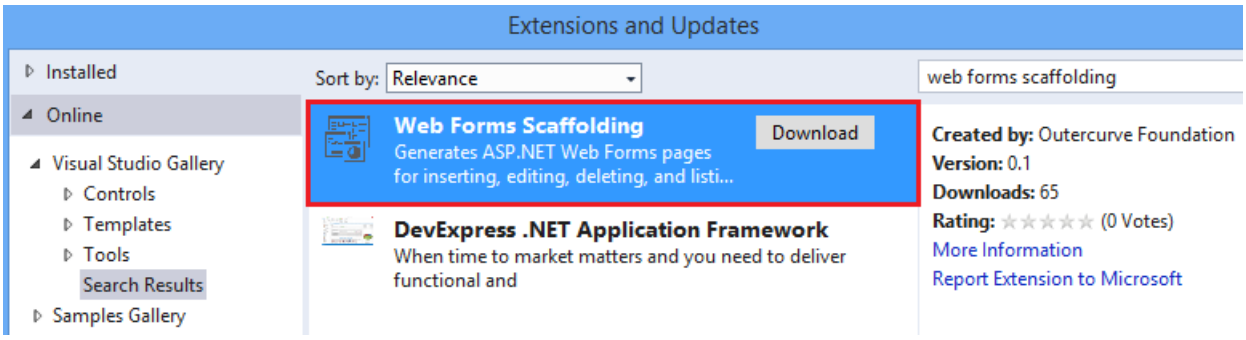


Add a scaffolded item to Web Forms

To add scaffolding that generates Web Forms code, you must either install an extension to Visual Studio or add MVC dependencies. Both approaches are shown below, but you only need to do one of these approaches.

Web Forms Scaffolding Extension

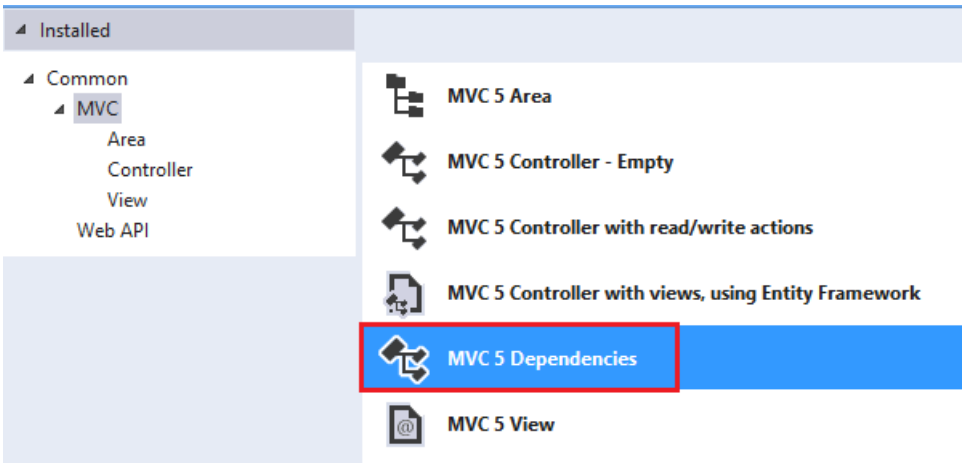
You can install a Visual Studio extension that enable you to use scaffolding with a Web Forms project. In Visual Studio, select **Tools** and then **Extensions and Updates**. From this dialog search the Visual Studio Gallery for **Web Forms Scaffolding**.



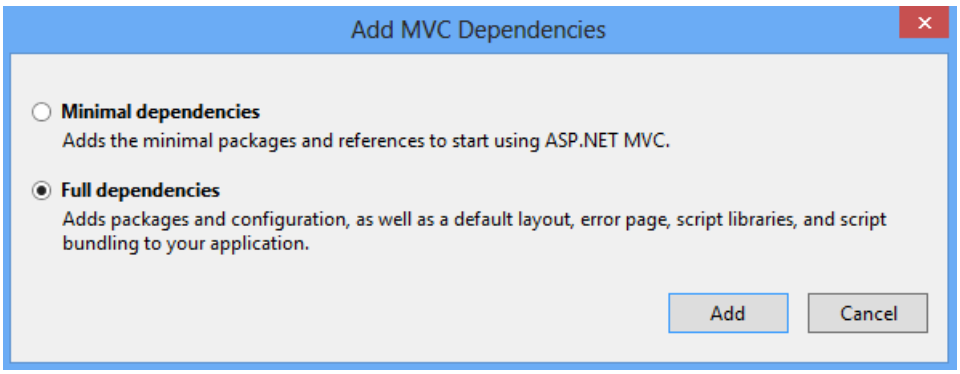
For more information, see Web Forms Scaffolding.

MVC Dependencies

To add MVC dependencies, select **Add - New Scaffolded Item**. In the Add Scaffold window, select **MVC Dependencies**, as shown below.



There are two options for scaffolding MVC; Minimal and Full. If you select Minimal, only the NuGet packages and references for ASP.NET MVC are added to your project. If you select the Full option, the Minimal dependencies are added, as well as the required content files for an MVC project. To easily use scaffolding, select Full dependencies.



After adding the dependencies, you will see a **readme.txt** file. Carefully follow the instructions in this file to ensure that your project works correctly.

When you have completed the steps in the readme.txt file, you can add a new scaffolded item as shown in the previous section about MVC and Web API. The automatically-generated views and controller will function correctly within your project.

Tutorials

To create a customized scaffolder, see Creating a Custom Scaffolder for Visual Studio.

To customize the generated files, see How to customize the generated files from the New Scaffolded Item dialog.

For an example of using scaffolding with **Database First development**, see EF Database First with ASP.NET MVC.

For an example of using scaffolding in an **MVC** project, see Getting Started with ASP.NET MVC 5.

For an example of using scaffolding in a **Web API** project, see Create a REST API with Attribute Routing in Web API 2.

Using Browser Link in Visual Studio 2013

2/20/2020 • 3 minutes to read • Edit Online

by Mike Wasson

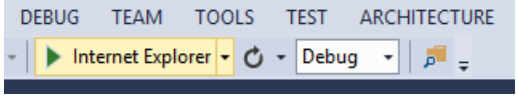
Browser Link is a new feature in Visual Studio 2013 that creates a communication channel between the development environment and one or more web browsers. You can use Browser Link to refresh your web application in several browsers at once, which is useful for cross-browser testing.

- Browser Refresh
- Viewing the Browser Link Dashboard
- Enabling Browser Link for Static HTML Files
- Disabling Browser Link
- How Does It Work?

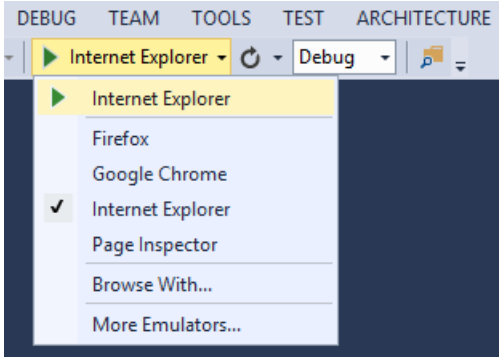
Browser Refresh

With Browser Refresh, you can refresh multiple browsers that are connected to Visual Studio through Browser Link.

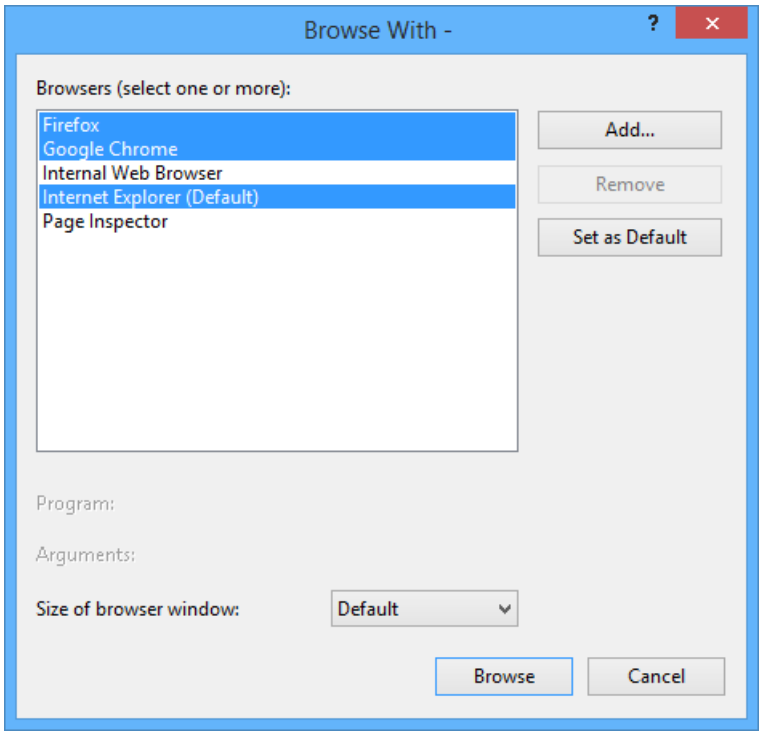
To use Browser Refresh, first create an ASP.NET application, using any of the project templates. Debug the application by pressing F5 or clicking the arrow icon in the toolbar:



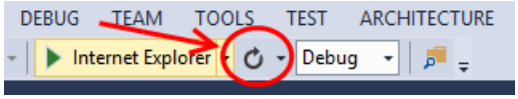
You can also use the dropdown to select a specific browser for debugging.



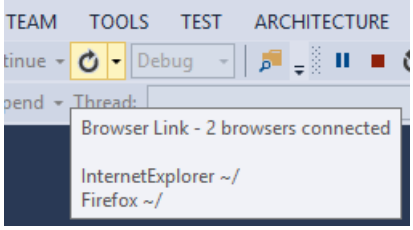
To debug with multiple browsers, select **Browse With**. In the **Browse With** dialog, hold down the CTRL key to select more than one browser. Click **Browse** to debug with the selected browsers. Browser Link also works if you launch a browser from outside Visual Studio and navigate to the application URL.



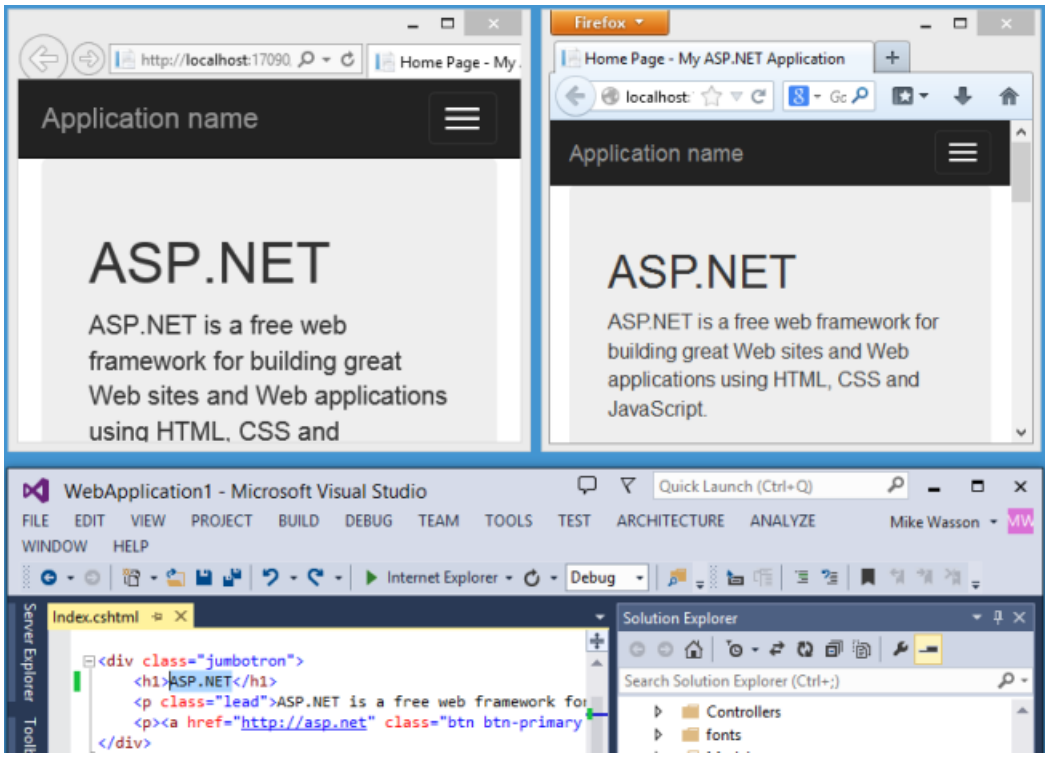
The Browser Link controls are located in the dropdown with the circular arrow icon. The arrow icon is the **Refresh** button.



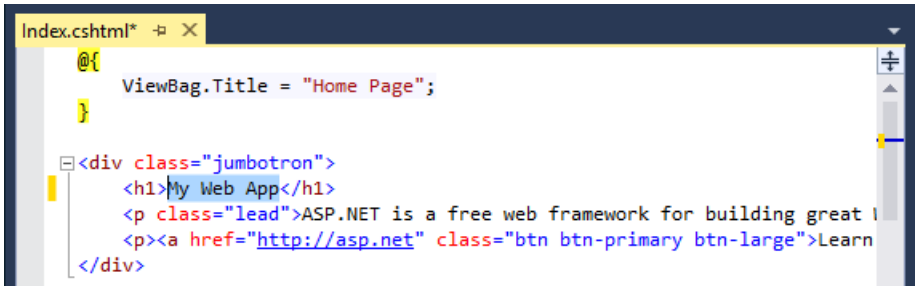
To see which browsers are connected, hover the mouse over the **Refresh** button while debugging. The connected browsers are shown in a ToolTip window.



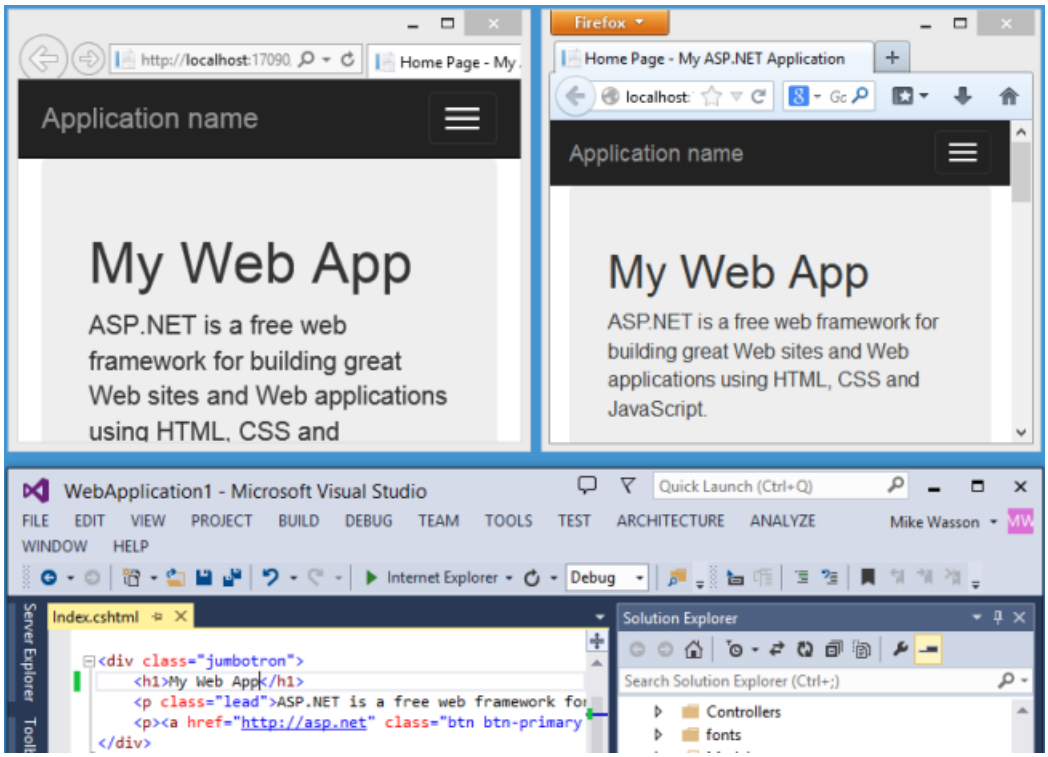
To refresh the connected browsers, click the **Refresh** button or press CTRL+ALT+ENTER. For example, the following screenshot shows an ASP.NET project, which I created using the MVC 5 project template. You can see the application running in two browsers at the top. At the bottom, the project is open in Visual Studio.



In Visual Studio, I changed the <h1> heading for the home page:



When I clicked the **Refresh** button, the change appeared in both browser windows:

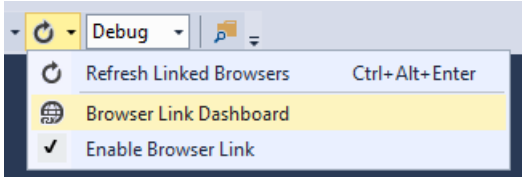


Notes

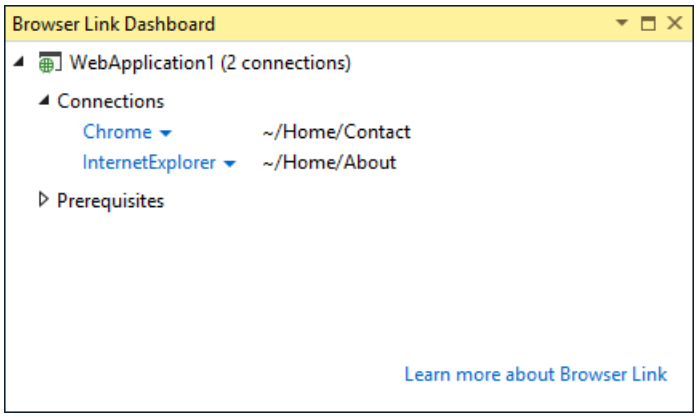
- To enable Browser Link, set `debug=true` in the <compilation> element in the project's Web.config file.
- The application must be running on localhost.
- The application must target .NET 4.0 or later.

Viewing the Browser Link Dashboard

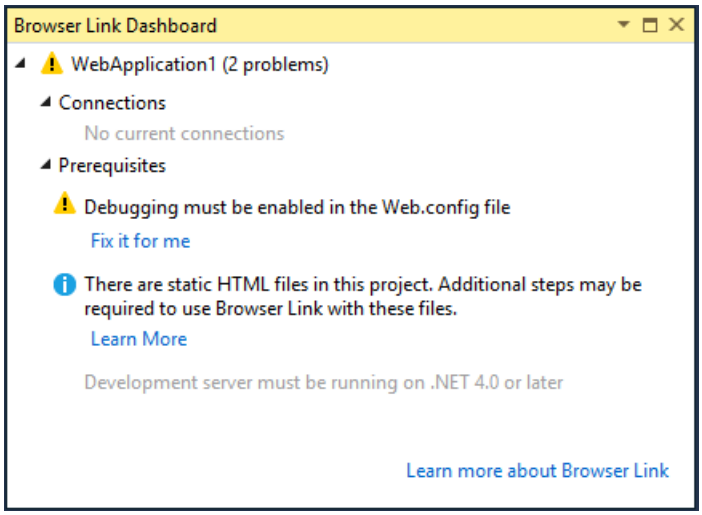
The Browser Link dashboard shows information about the Browser Link connections. To view the dashboard, select the Browser Link dropdown menu (the small arrow next to the **Refresh** button). Then click **Browser Link Dashboard**.



The dashboard lists the connected Browsers and the URL to which each browser has navigated.



The **Prerequisites** section shows any steps needed to enable Browser Link for that project. For example, the following screenshot shows a project where "debug" is set to false in the Web.config file.



Enabling Browser Link for Static HTML Files

To enable Browser Link for static HTML files, add the following to your Web.config file.

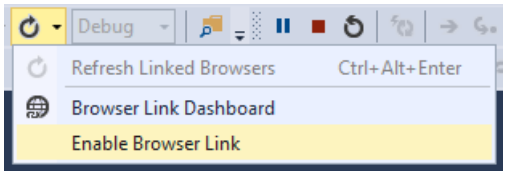
```
<configuration>
  <system.webServer>
    <handlers>
      <add name="Browser Link for HTML" path="*.html" verb="*"
           type="System.Web.StaticFileHandler, System.Web, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a"
           resourceType="File" preCondition="integratedMode" />
    </handlers>
  </system.webServer>
</configuration>
```

For performance reasons, remove this setting when you publish your project.

Disabling Browser Link

Browser Link is enabled by default. There are several ways to disable it:

- In the Browser Link dropdown menu, uncheck **Enable Browser Link**.



- In the Web.config file, add a key named "vs:EnableBrowserLink" with the value "false" in the appSettings section.

```
<appSettings>
  <add key="vs:EnableBrowserLink" value="false"/>
</appSettings>
```

- In the Web.config file, set debug to false.

```
<system.web>
  <compilation debug="false" targetFramework="4.5" />
</system.web>
```

How Does It Work?

Browser Link uses SignalR to create a communication channel between Visual Studio and the browser. When Browser Link is enabled, Visual Studio acts as a SignalR server that multiple clients (browsers) can connect to. Browser Link also registers an HTTP module with ASP.NET. This module injects special <script> references into every page request from the server. You can see the script references by selecting "View source" in the browser.

```
83 <!-- Visual Studio Browser Link -->
84 <script type="application/json" id="__browserLink_initializationData">
85     {"appName":"InternetExplorer","requestId":"c762efb8f5fc495d8bf82460d3867b9d"}
86 </script>
87 <script type="text/javascript" src="http://localhost:57058/cb8085b94cca43e58d81a3e9580baf51/browserLin
88 <!-- End Browser Link -->
89
90 </body>
91 </html>
92
```

Your source files are not modified. The HTTP module injects the script references dynamically.

Because the browser-side code is all JavaScript, it works on all browsers that SignalR supports, without requiring

any browser plug-in.

ASP.NET and Web Tools for Visual Studio 2013 Release Notes

4/10/2020 • 33 minutes to read • Edit Online

by Microsoft

This document describes the release of ASP.NET and Web Tools for Visual Studio 2013.

Contents

- Installation Notes
- Documentation
- Software Requirements

New Features in ASP.NET and Web Tools for Visual Studio 2013

- One ASP.NET
- New Web Project Experience
- ASP.NET Scaffolding
- Browser Link
- Visual Studio Web Editor Enhancements
- Azure App Service Web Apps Support in Visual Studio
- Web Publish Enhancements
- NuGet 2.7
- ASP.NET Web Forms
- ASP.NET MVC 5
- ASP.NET Web API 2
- ASP.NET SignalR
- ASP.NET Identity
- Microsoft OWIN Components
- Entity Framework 6
- ASP.NET Razor 3
- ASP.NET App Suspend
- Known Issues and Breaking Changes

Installation Notes

ASP.NET and Web Tools for Visual Studio 2013 are bundled in the main installer and can be downloaded here.

Documentation

Tutorials and other information about ASP.NET and Web Tools for Visual Studio 2013 are available from the ASP.NET web site.

Software Requirements

ASP.NET and Web Tools requires Visual Studio 2013.

New Features in ASP.NET and Web Tools for Visual Studio 2013

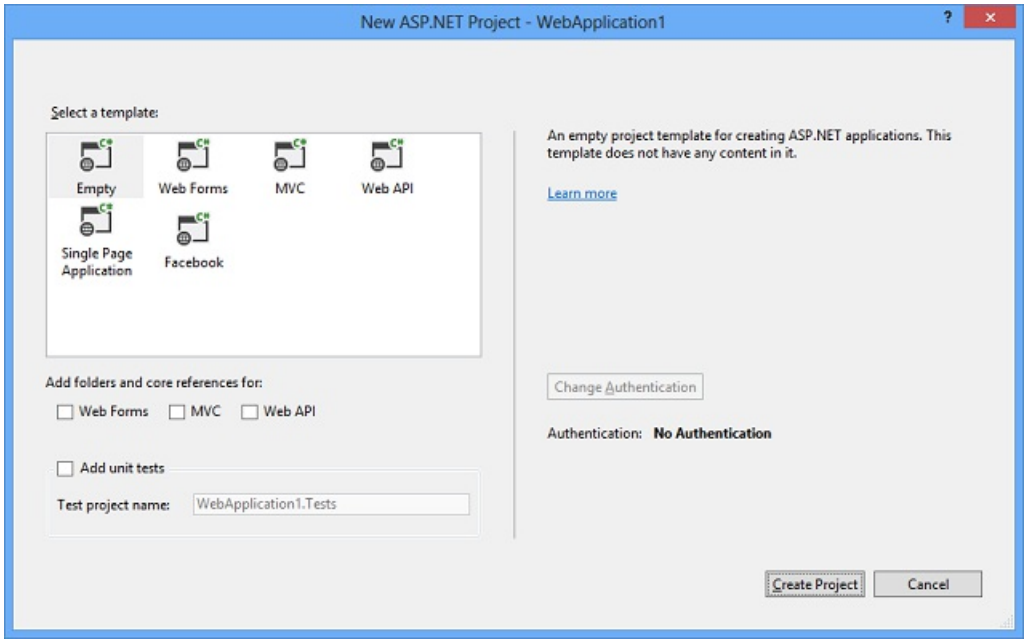
The following sections describe the features that have been introduced in the release.

One ASP.NET

With the release of Visual Studio 2013, we have taken a step towards unifying the experience of using ASP.NET technologies, so that you can easily mix and match the ones you want. For example, you can start a project using MVC and easily add Web Forms pages to the project later, or scaffold Web APIs in a Web Forms project. One ASP.NET is all about making it easier for you as a developer to do the things you love in ASP.NET. No matter what technology you choose, you can have confidence that you are building on the trusted underlying framework of One ASP.NET.

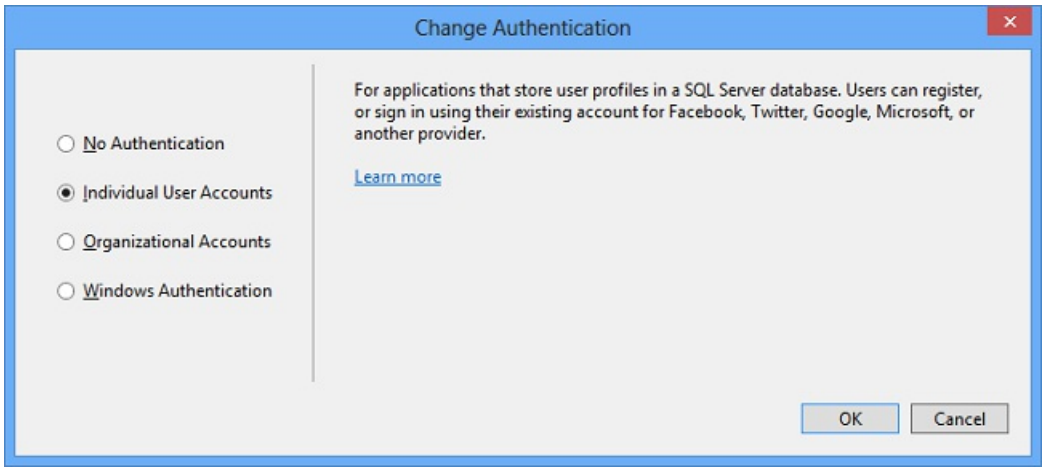
New Web Project Experience

We have enhanced the experience of creating new web projects in Visual Studio 2013. In the **New ASP.NET Web Project** dialog you can select the project type you want, configure any combination of technologies (Web Forms, MVC, Web API), configure authentication options, and add a unit test project.



The new dialog enables you to change the default authentication options for many of the templates. For example, when you create an ASP.NET Web Forms project you can select any of the following options:

- No Authentication
- Individual User Accounts (ASP.NET membership or social provider log in)
- Organizational Accounts (Active Directory in an internet application)
- Windows Authentication (Active Directory in an intranet application)



For more information about the new process for creating web projects, see Creating ASP.NET Web Projects in Visual Studio 2013. For more information about the new authentication options, see ASP.NET Identity later in this document.

ASP.NET Scaffolding

ASP.NET Scaffolding is a code generation framework for ASP.NET Web applications. It makes it easy to add boilerplate code to your project that interacts with a data model.

In previous versions of Visual Studio, scaffolding was limited to ASP.NET MVC projects. With Visual Studio 2013, you can now use scaffolding for any ASP.NET project, including Web Forms. Visual Studio 2013 does not currently support generating pages for a Web Forms project, but you can still use scaffolding with Web Forms by adding MVC dependencies to the project. Support for generating pages for Web Forms will be added in a future update.

When using scaffolding, we ensure that all required dependencies are installed in the project. For example, if you start with an ASP.NET Web Forms project and then use scaffolding to add a Web API Controller, the required NuGet packages and references are added to your project automatically.

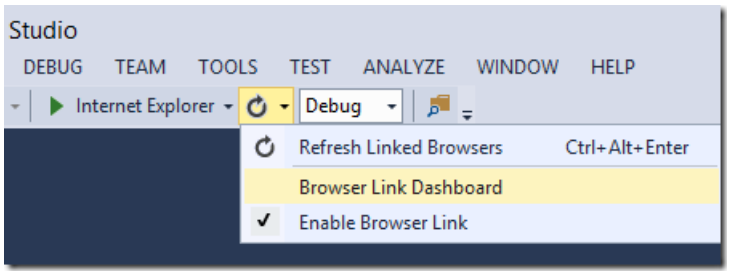
To add MVC scaffolding to a Web Forms project, add a **New Scaffolded Item** and select **MVC 5 Dependencies** in the dialog window. There are two options for scaffolding MVC; Minimal and Full. If you select Minimal, only the NuGet packages and references for ASP.NET MVC are added to your project. If you select the Full option, the Minimal dependencies are added, as well as the required content files for an MVC project.

Support for scaffolding async controllers uses the new async features from Entity Framework 6.

For more information and tutorials, see ASP.NET Scaffolding Overview.

Browser Link – SignalR channel between browser and Visual Studio

The new Browser Link feature lets you connect multiple browsers to Visual Studio and refresh them all by clicking a button in the toolbar. You can connect multiple browsers to your development site, including mobile emulators, and click refresh to refresh all the browsers all at the same time. Browser Link also exposes an API to enable developers to write Browser Link extensions.



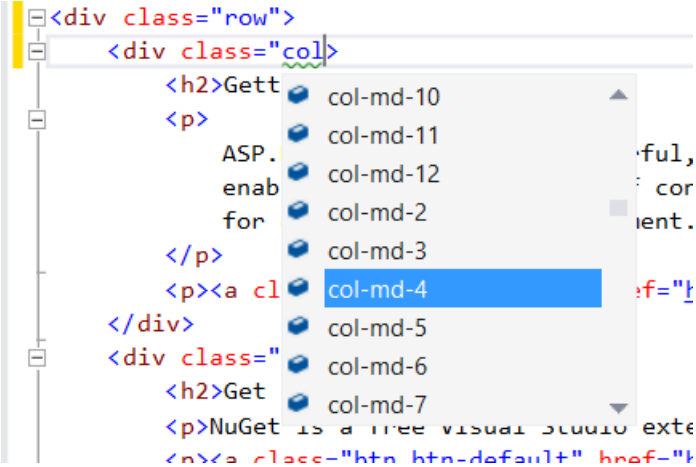
By enabling developers to take advantage of the Browser Link API, it becomes possible to create very advanced

scenarios that crosses boundaries between Visual Studio and any browser that's connected. Web Essentials takes advantage of the API to create an integrated experience between Visual Studio and the browser's developer tools, remote controlling mobile emulators and a lot more.

Visual Studio Web Editor Enhancements

Visual Studio 2013 includes a new HTML editor for Razor files and HTML files in web applications. The new HTML editor provides a single unified schema based on HTML5. It has automatic brace completion, jQuery UI and AngularJS attribute IntelliSense, attribute IntelliSense Grouping, ID and class name Intellisense, and other improvements including better performance, formatting and SmartTags.

The following screenshot demonstrates using Bootstrap attribute IntelliSense in the HTML editor.



Visual Studio 2013 also comes with both CoffeeScript and LESS editors built in. The LESS editor comes with all the cool features from the CSS editor and has specific Intellisense for variables and mixins across all the LESS documents in the @import chain.

Azure App Service Web Apps Support in Visual Studio

In Visual Studio 2013 with the Azure SDK for .NET 2.2, you can use **Server Explorer** to interact directly with your remote web apps. You can sign in to your Azure account, create new web apps, configure apps, view real-time logs, and more. Coming soon after SDK 2.2 is released, you'll be able to run in debug mode remotely in Azure. Most of the new features for Azure App Service Web Apps also work in Visual Studio 2012 when you install the current release of the Azure SDK for .NET.

For more information, see the following resources:

- Create an ASP.NET web app in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio

Web Publish Enhancements

Visual Studio 2013 includes new and enhanced Web Publish features. Here are a few of them:

- Easily automate Web.config file encryption. (This link and the following two point to documentation on MSDN that might not be available until late in the day on 10/17.)
- Easily automate taking an application offline during deployment.
- Configure Web Deploy to use file checksum instead of last-changed date for determining which files should be copied to the server.
- Quickly publish individual selected files (including Web.config) when you're using the FTP or file system publish methods as well as with Web Deploy.

For more information about ASP.NET web deployment, see the ASP.NET site.

NuGet 2.7

NuGet 2.7 includes a rich set of new features which are described in detail at NuGet 2.7 Release Notes.

This version of NuGet also removes the need to provide explicit consent for NuGet's package restore feature to download packages. Consent (and the associated checkbox in NuGet's preferences dialog) is now granted by installing NuGet. Now package restore simply works by default.

ASP.NET Web Forms

One ASP.NET

The Web Forms project templates integrate seamlessly with the new One ASP.NET experience. You can add MVC and Web API support to your Web Forms project, and you can configure authentication using the One ASP.NET project creation wizard. For more information, see Creating ASP.NET Web Projects in Visual Studio 2013.

ASP.NET Identity

The Web Forms project templates support the new ASP.NET Identity framework. In addition, the templates now support creation of a Web Forms intranet project. For more information, see Authentication Methods in **Creating ASP.NET Web Projects in Visual Studio 2013**.

Bootstrap

The Web Forms templates use Bootstrap to provide a sleek and responsive look and feel that you can easily customize. For more information, see Bootstrap in the Visual Studio 2013 web project templates.

ASP.NET MVC 5

One ASP.NET

The Web MVC project templates integrate seamlessly with the new One ASP.NET experience. You can customize your MVC project and configure authentication using the One ASP.NET project creation wizard. An introductory tutorial to ASP.NET MVC 5 can be found at Getting Started with ASP.NET MVC 5.

For information on upgrading MVC 4 projects to MVC 5, see How to Upgrade an ASP.NET MVC 4 and Web API Project to ASP.NET MVC 5 and Web API 2.

ASP.NET Identity

The MVC project templates have been updated to use ASP.NET Identity for authentication and identity management. A tutorial featuring Facebook and Google authentication and the new membership API can be found at Create an ASP.NET MVC 5 App with Facebook and Google OAuth2 and OpenID Sign-on and Create an ASP.NET MVC app with auth and SQL DB and deploy to Azure App Service.

Bootstrap

The MVC project template has been updated to use Bootstrap to provide a sleek and responsive look and feel that you can easily customize. For more information, see Bootstrap in the Visual Studio 2013 web project templates.

Authentication filters

Authentication filters are a new kind of filter in ASP.NET MVC that run prior to authorization filters in the ASP.NET MVC pipeline and allow you to specify authentication logic per-action, per-controller, or globally for all controllers. Authentication filters process credentials in the request and provide a corresponding principal. Authentication filters can also add authentication challenges in response to unauthorized requests.

Filter overrides

You can now override which filters apply to a given action method or controller by specifying an override filter.

Override filters specify a set of filter types that should not be run for a given scope (action or controller). This allows you to configure filters that apply globally but then exclude certain global filters from applying to specific actions or controllers.

Attribute routing

ASP.NET MVC now supports attribute routing, thanks to a contribution by Tim McCall, the author of http://attributerouting.net. With attribute routing you can specify your routes by annotating your actions and controllers.

ASP.NET Web API 2

Attribute routing

ASP.NET Web API now supports attribute routing, thanks to a contribution by Tim McCall, the author of http://attributerouting.net. With attribute routing you can specify your Web API routes by annotating your actions and controllers like this:

```
[RoutePrefix("orders")]
public class OrdersController : ApiController
{
    [Route("{id}")]
    public Order Get(int id) { }
    [Route("{id}/approve")]
    public Order Approve(int id) { }
}
```

Attribute routing gives you more control over the URIs in your web API. For example, you can easily define a resource hierarchy using a single API controller:

```
public class MoviesController : ApiController
{
    [Route("movies")]
    public IEnumerable<Movie> Get() { }
    [Route("actors/{actorId}/movies")]
    public IEnumerable<Movie> GetByActor(int actorId) { }
    [Route("directors/{directorId}/movies")]
    public IEnumerable<Movie> GetByDirector(int directorId) { }
}
```

Attribute routing also provides a convenient syntax for specifying optional parameters, default values, and route constraints:

```
// Optional parameter
[Route("people/{name?}")]
// Default value
[Route("people/{name=Dan}")]
// Constraint: Alphabetic characters only.
[Route("people/{name:alpha}")]
```

For more information about attribute routing, see Attribute Routing in Web API 2.

OAuth 2.0

The Web API and Single Page Application project templates now support authorization using OAuth 2.0. OAuth 2.0 is a framework for authorizing client access to protected resources. It works for a variety of clients including browsers and mobile devices.

Support for OAuth 2.0 is based on new security middleware provided by the Microsoft OWIN Components for

bearer authentication and implementing the authorization server role. Alternatively, clients can be authorized using an organizational authorization server, such as Azure Active Directory or ADFS in Windows Server 2012 R2.

OData Improvements

Support for $select, $expand, $batch, and $value

ASP.NET Web API OData now has full support for $select, $expand, and $value. You can also use $batch for request batching and processing of change sets.

The $select and $expand options let you change the shape of the data that is returned from an OData endpoint. For more information, see Introducing $select and $expand support in Web API OData.

Improved extensibility

The OData formatters are now extensible. You can add Atom entry metadata, support named stream and media link entries, add instance annotations, and customize how links are generated.

Type-less support

You can now build OData services without needing to define CLR types for your entity types. Instead, your OData controllers can take or return instances of **IEdmObject**, which are the OData formatters serialize/deserialize.

Reuse an existing model

If you already have an existing entity data model (EDM), you can now reuse it directly, instead of having to build a new one. For example, if you are using Entity Framework, you can use the EDM that EF builds for you.

Request Batching

Request batching combines multiple operations into a single HTTP POST request, to reduce network traffic and provide a smoother, less chatty user interface. ASP.NET Web API now supports several strategies for request batching:

- Use the $batch endpoint of an OData service.
- Package multiple requests into a single MIME multipart request.
- Use a custom batching format.

To enable request batching, simply add a route with a batching handler to your Web API configuration:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.MapHttpBatchRoute(
            routeName: "WebApiBatch",
            routeTemplate: "api/batch",
            batchHandler: new DefaultHttpBatchHandler(GlobalConfiguration.DefaultServer));
    }
}
```

You can also control whether requests or executed sequentially or in any order.

Portable ASP.NET Web API Client

You can now use the ASP.NET Web API Client to create portable class libraries that work across your Windows Store and Windows Phone 8 applications. You can also create portable formatters that can be shared across client and server.

Improved Testability

Web API 2 makes it much easier to unit test your API controllers. Just instantiate your API controller with your request message and configuration, and then call the action method you wish to test. It is also easy to mock the

UrlHelper class, for action methods that perform link generation.

IHttpActionResult

You can now implement IHttpActionResult to encapsulate the result of your Web API action methods. An IHttpActionResult returned from a Web API action method is executed by the ASP.NET Web API runtime to produce the resultant response message. An IHttpActionResult can be returned from any Web API action to simplify unit testing of your Web API implementation. For convenience a number of IHttpActionResult implementations are provided out of the box including results for returning specific status codes, formatted content or content-negotiated responses.

HttpRequestContext

The new **HttpRequestContext** tracks any state that is tied to the request but is not immediately available from the request. For example, you can use the **HttpRequestContext** to get route data, the principal associated with the request, the client certificate, the **UrlHelper** and the virtual path root. You can easily create an **HttpRequestContext** for unit testing purposes.

Because the principal for the request is flowed with the request instead of relying on **Thread.CurrentPrincipal**, the principal is now available throughout the lifetime of the request while it is in the Web API pipeline.

CORS

Thanks to another great contribution from Brock Allen, ASP.NET now fully supports Cross Origin Request Sharing (CORS).

Browser security prevents a web page from making AJAX requests to another domain. CORS is a W3C standard that allows a server to relax the same-origin policy. Using CORS, a server can explicitly allow some cross-origin requests while rejecting others.

Web API 2 now supports CORS, including automatic handling of preflight requests. For more information, see Enabling Cross-Origin Requests in ASP.NET Web API.

Authentication Filters

Authentication filters are a new kind of filter in ASP.NET Web API that run prior to authorization filters in the ASP.NET Web API pipeline and allow you to specify authentication logic per-action, per-controller, or globally for all controllers. Authentication filters process credentials in the request and provide a corresponding principal. Authentication filters can also add authentication challenges in response to unauthorized requests.

Filter Overrides

You can now override which filters apply to a given action method or controller, by specifying an override filter. Override filters specify a set of filter types that should not run for a given scope (action or controller). This allows you to add global filters, but then exclude some from specific actions or controllers.

OWIN Integration

ASP.NET Web API now fully supports OWIN and can be run on any OWIN capable host. Also included is a **HostAuthenticationFilter** that provides integration with the OWIN authentication system.

With OWIN integration, you can self-host Web API in your own process alongside other OWIN middleware, such as SignalR. For more information, see Use OWIN to Self-Host ASP.NET Web API.

ASP.NET SignalR 2.0

The following sections describe features of SignalR 2.0.

- Built on OWIN
- MapHubs and MapConnection are now MapSignalR
- Cross-Domain Support

- iOS and Android support via MonoTouch and MonoDroid
- Portable .NET Client
- New Self-Host Package
- Backward-compatible server support
- Removed server support for .NET 4.0
- Sending a message to a list of clients and groups
- Sending a message to a specific user
- Better error handling support
- Easier unit testing of hubs
- JavaScript error handling

For an example of how to upgrade an existing 1.x project to SignalR 2.0, see Upgrading a SignalR 1.x Project.

Built on OWIN

SignalR 2.0 is built completely on OWIN (the Open Web Interface for .NET). This change makes the setup process for SignalR much more consistent between web-hosted and self-hosted SignalR applications, but has also required a number of API changes.

MapHubs and MapConnection are now MapSignalR

For compatibility with OWIN standards, these methods have been renamed to `MapSignalR`. `MapSignalR` called without parameters will map all hubs (as `MapHubs` does in version 1.x); to map individual **PersistentConnection** objects, specify the connection type as the type parameter, and the URL extension for the connection as the first argument.

The `MapSignalR` method is called in an Owin startup class. Visual Studio 2013 contains a new template for an Owin startup class; to use this template, do the following:

1. Right-click on the project
2. Select **Add**, **New Item...**
3. Select **Owin Startup class**. Name the new class **Startup.cs**.

In a **Web application,** the Owin startup class containing the `MapSignalR` method is then added to Owin's startup process using an entry in the application settings node of the Web.Config file, as shown below.

In a **Self-hosted application**, the Startup class is passed as the type parameter of the `WebApp.Start` method.

Mapping hubs and connections in SignalR 1.x (from the global application file in a web application):

```
protected void Application_Start(object sender, EventArgs e)
{
    // Map all hubs to "/signalr"
    RouteTable.Routes.MapHubs();
    // Map the Echo PersistentConnection to "/echo"
    RouteTable.Routes.MapConnection<myconnection>("echo", "/echo");
}
```

Mapping hubs and connections in SignalR 2.0 (from an Owin Startup class file):

```
using Microsoft.AspNet.SignalR;
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(MyWebApplication.Startup))]

namespace MyWebApplication
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Map all hubs to "/signalr"
            app.MapSignalR();
            // Map the Echo PersistentConnection to "/echo"
            app.MapSignalR<echoconnection>("/echo");
        }
    }
}
```

In a **Self-hosted application**, the Startup class is passed as the type parameter for the `WebApp.Start` method, as shown below.

```
string url = "http://localhost:8080";
using (WebApp.Start<startup>(url))
{
    Console.WriteLine("Server running on {0}", url);
    Console.ReadLine();
}
```

Cross-Domain Support

In SignalR 1.x, cross domain requests were controlled by a single EnableCrossDomain flag. This flag controlled both JSONP and CORS requests. For greater flexibility, all CORS support has been removed from the server component of SignalR (JavaScript clients still use CORS normally if it is detected that the browser supports it), and new OWIN middleware has been made available to support these scenarios.

In SignalR 2.0, If JSONP is required on the client (to support cross-domain requests in older browsers), it will need to be enabled explicitly by setting `EnableJSONP` on the `HubConfiguration` object to `true`, as shown below. JSONP is disabled by default, as it is less secure than CORS.

To add the new CORS middleware in SignalR 2.0, add the `Microsoft.Owin.Cors` library to your project, and call `UseCors` before your SignalR middleware, as shown in the section below.

Adding Microsoft.Owin.Cors to your project: To install this library, run the following command in the Package Manager Console:

```
Install-Package Microsoft.Owin.Cors
```

This command will add the 2.0.0 version of the package to your project.

Calling UseCors

The following code snippets demonstrate how to implement cross-domain connections in SignalR 1.x and 2.0.

Implementing cross-domain requests in SignalR 1.x (from the global application file)

```
protected void Application_Start(object sender, EventArgs e)
{
    var hubConfiguration = new HubConfiguration();
    hubConfiguration.EnableCrossDomain = true;
    RouteTable.Routes.MapHubs(hubConfiguration);
}
```

Implementing cross-domain requests in SignalR 2.0 (from a C# code file)

The following code demonstrates how to enable CORS or JSONP in a SignalR 2.0 project. This code sample uses `Map` and `RunSignalR` instead of `MapSignalR`, so that the CORS middleware runs only for the SignalR requests that require CORS support (rather than for all traffic at the path specified in `MapSignalR`.) `Map` can also be used for any other middleware that needs to run for a specific URL prefix, rather than for the entire application.

```
using Microsoft.AspNet.SignalR;
using Microsoft.Owin.Cors;
using Owin;

[assembly: OwinStartup(typeof(MyWebApplication.Startup))]

namespace MyWebApplication
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // Branch the pipeline here for requests that start with "/signalr"
            app.Map("/signalr", map =>
            {
                // Setup the CORS middleware to run before SignalR.
                // By default this will allow all origins. You can
                // configure the set of origins and/or http verbs by
                // providing a cors options with a different policy.
                map.UseCors(CorsOptions.AllowAll);
                var hubConfiguration = new HubConfiguration
                {
                    // You can enable JSONP by uncommenting line below.
                    // JSONP requests are insecure but some older browsers (and some
                    // versions of IE) require JSONP to work cross domain
                    // EnableJSONP = true
                };
                // Run the SignalR pipeline. We're not using MapSignalR
                // since this branch already runs under the "/signalr"
                // path.
                map.RunSignalR(hubConfiguration);
            });
        }
    }
}
```

iOS and Android support via MonoTouch and MonoDroid

Support has been added for iOS and Android clients using MonoTouch and MonoDroid components from the Xamarin library. For more information on how to use them, see Using Xamarin Components. These components will be available in the Xamarin Store when the SignalR RTW release is available.

Portable .NET client

To better facilitate cross-platform development, the Silverlight, WinRT and Windows Phone clients have been replaced with a single portable .NET client that supports the following platforms:

- NET 4.5

- Silverlight 5
- WinRT (.NET for Windows Store Apps)
- Windows Phone 8

New Self-Host Package

There is now a NuGet package to make it easier to get started with SignalR Self-Host (SignalR applications that are hosted in a process or other application, rather than being hosted in a web server). To upgrade a self-host project built with SignalR 1.x, remove the Microsoft.AspNet.SignalR.Owin package, and add the Microsoft.AspNet.SignalR.SelfHost package. For more information on getting started with the self-host package, see Tutorial: SignalR Self-Host.

Backward-compatible server support

In previous versions of SignalR, the versions of the SignalR package used in the client and the server needed to be identical. In order to support thick-client applications that would be difficult to update, SignalR 2.0 now supports using a newer server version with an older client. **Note: SignalR 2.0 does not support servers built with older versions with newer clients.**

Removed server support for .NET 4.0

SignalR 2.0 has dropped support for server interoperability with .NET 4.0. .NET 4.5 must be used with SignalR 2.0 servers. There is still a .NET 4.0 client for SignalR 2.0.

Sending a message to a list of clients and groups

In SignalR 2.0, it's possible to send a message using a list of client and group IDs. The following code snippets demonstrate how to do this.

Sending a message to a list of clients and groups using PersistentConnection

```
using Microsoft.AspNet.SignalR;
using System.Collections.Generic;
public class ChatConnection : PersistentConnection
{
    static List<string> ConnectionIds = new List<string>();
    static List<string> groups = new List<string>{"chatGroup", "chatGroup2"};
    protected override System.Threading.Tasks.Task OnReceived(IRequest request, string connectionId, string
data)
    {
        Connection.Send(ConnectionIds, data);
        Groups.Send(groups, data);
        return base.OnReceived(request, connectionId, data);
    }
    protected override System.Threading.Tasks.Task OnConnected(IRequest request, string connectionId)
    {
        ConnectionIds.Add(connectionId);
        Groups.Add(connectionId, "chatGroup");
        return base.OnConnected(request, connectionId);
    }
    protected override System.Threading.Tasks.Task OnDisconnected(IRequest request, string connectionId)
    {
        ConnectionIds.Remove(connectionId);
        return base.OnDisconnected(request, connectionId);
    }
}
```

Sending a message to a list of clients and groups using Hubs

```
using Microsoft.AspNet.SignalR;
using System.Collections.Generic;
public class ChatHub : Hub
{
    static List<string> ConnectionIds = new List<string>();
    static List<string> groups = new List<string> { "chatGroup", "chatGroup2" };
    public void Send(string name, string message)
    {
        // Call the broadcastMessage method to update clients.
        Clients.Clients(ConnectionIds).broadcastMessage(name, message);
        Clients.Groups(groups).broadcastMessage(name, message);
    }
    public override System.Threading.Tasks.Task OnConnected()
    {
        ConnectionIds.Add(Context.ConnectionId);
        Groups.Add(Context.ConnectionId, "chatGroup");
        return base.OnConnected();
    }
    public override System.Threading.Tasks.Task OnDisconnected()
    {
        ConnectionIds.Remove(Context.ConnectionId);
        return base.OnDisconnected();
    }
}
```

Sending a message to a specific user

This feature allows users to specify what the userId is based on an IRequest via a new interface IUserIdProvider:

The IUserIdProvider interface

```
public interface IUserIdProvider
{
    string GetUserId(IRequest request);
}
```

By default there will be an implementation that uses the user's IPrincipal.Identity.Name as the user name.

In hubs, you'll be able to send messages to these users via a new API:

Using the Clients.User API

```
public class MyHub : Hub
{
    public void Send(string userId, string message)
    {
        Clients.User(userId).send(message);
    }
}
```

Better Error Handling Support

Users can now throw **HubException** from any hub invocation. The constructor of the **HubException** can take a string message and an object extra error data. SignalR will auto-serialize the exception and send it to the client where it will be used to reject/fail the hub method invocation.

The **show detailed hub exceptions** setting has no bearing on **HubException** being sent back to the client or not; it is always sent.

Server-side code demonstrating sending a HubException to the client

```
public class MyHub : Hub
{
    public void Send(string message)
    {
        if(message.Contains("<script>"))
        {
            throw new HubException("This message will flow to the client", new { user =
Context.User.Identity.Name, message = message });
        }

        Clients.All.send(message);
    }
}
```

JavaScript client code demonstrating responding to a HubException sent from the server

```
myHub.server.send("<script>")
            .fail(function (e) {
                if (e.source === 'HubException') {
                    console.log(e.message + ' : ' + e.data.user);
                }
            });
```

.NET client code demonstrating responding to a HubException sent from the server

```
try
{
    await myHub.Invoke("Send", "<script>");
}
catch(HubException ex)
{
    Conosle.WriteLine(ex.Message);
}
```

Easier unit testing of hubs

SignalR 2.0 includes an interface called `IHubCallerConnectionContext` on Hubs that makes it easier to create mock client side invocations. The following code snippets demonstrate using this interface with popular test harnesses xUnit.net and moq.

Unit testing SignalR with xUnit.net

```
[Fact]
public void HubsAreMockableViaDynamic()
{
    bool sendCalled = false;
    var hub = new MyHub();
    var mockClients = new Mock<IHubCallerConnectionContext>();
    hub.Clients = mockClients.Object;
    dynamic all = new ExpandoObject();
    all.send = new Action<string>(message =>
    {
        sendCalled = true;
    });
    mockClients.Setup(m => m.All).Returns((ExpandoObject)all);
    hub.Send("foo");
    Assert.True(sendCalled);
}
```

Unit testing SignalR with moq

```
[Fact]
public interface IClientContract
{
    void send(string message);
}
public void HubsAreMockableViaType()
{
    var hub = new MyHub();
    var mockClients = new Mock<IHubCallerConnectionContext>();
    var all = new Mock<IClientContract>();
    hub.Clients = mockClients.Object;
    all.Setup(m => m.send(It.IsAny<string>())).Verifiable();
    mockClients.Setup(m => m.All).Returns(all.Object);
    hub.Send("foo");
    all.VerifyAll();
```

JavaScript error handling

In SignalR 2.0, all JavaScript error handling callbacks return JavaScript error objects instead of raw strings. This allows SignalR to flow richer information to your error handlers. You can get the inner exception from the `source` property of the error.

JavaScript client code that handles the Start.Fail exception

```
connection.start().fail(function(e) {
    console.log('The error is: ' + e.message);
});
```

ASP.NET Identity

New ASP.NET Membership System

ASP.NET Identity is the new membership system for ASP.NET applications. ASP.NET Identity makes it easy to integrate user-specific profile data with application data. ASP.NET Identity also allows you to choose the persistence model for user profiles in your application. You can store the data in a SQL Server database or another data store, including NoSQL data stores such as Azure Storage Tables. For more information, see Individual User Accounts in **Creating ASP.NET Web Projects in Visual Studio 2013**.

Claims-based authentication

ASP.NET now supports claims-based authentication, where the user's identity is represented as a set of claims from a trusted issuer. Users can be authenticated using a username and password maintained in an application database, or using social identity providers (for example: Microsoft Accounts, Facebook, Google, Twitter), or using organizational accounts through Azure Active Directory or Active Directory Federation Services (ADFS).

Integration with Azure Active Directory and Windows Server Active Directory

You can now create ASP.NET projects that use Azure Active Directory or Windows Server Active Directory (AD) for authentication. For more information, see Organizational Accounts in **Creating ASP.NET Web Projects in Visual Studio 2013**.

OWIN Integration

ASP.NET authentication is now based on OWIN middleware that can be used on any OWIN-based host. For more information about OWIN, see the following Microsoft OWIN Components section.

Microsoft OWIN Components

Open Web Interface for .NET (OWIN) defines an abstraction between .NET web servers and web applications. OWIN decouples the web application from the server, making web applications host-agnostic. For example, you

can host an OWIN-based web application in IIS or self-host it in a custom process.

Changes introduced in the Microsoft OWIN components (also known as the Katana project) include new server and host components, new helper libraries and middleware, and new authentication middleware.

For more information about OWIN and Katana, see What's new in OWIN and Katana.

Note: OWIN applications cannot run in IIS classic mode; they must be run in integrated mode.

Note: OWIN applications must be run in full trust.

New Servers and Hosts

With this release, new components were added to enable self-host scenarios. These components include the following NuGet packages:

- **Microsoft.Owin.Host.HttpListener**. Provides an OWIN server that uses **HttpListener** to listen for HTTP requests and direct them into the OWIN pipeline.
- **Microsoft.Owin.Hosting** Provides a library for developers who wish to self-host an OWIN pipeline in a custom process, such as a console application or Windows service.
- **OwinHost**. Provides a stand-alone executable that wraps `Microsoft.Owin.Hosting` and lets you self-host an OWIN pipeline without having to write a custom host application.

In addition, the `Microsoft.Owin.Host.SystemWeb` package now enables middleware to provide hints to the **SystemWeb** server, indicating that the middleware should be called during a specific ASP.NET pipeline stage. This feature is particularly useful for authentication middleware, which should run early in the ASP.NET pipeline.

Helper Libraries and Middleware

Although you can write OWIN components using only the function and type definitions from the OWIN specification, the new `Microsoft.Owin` package provides a more user-friendly set of abstractions. This package combines several earlier packages (e.g., `Owin.Extensions`, `Owin.Types`) into a single, well-structured object model that can then be easily used by other OWIN components. In fact, the majority of Microsoft OWIN components now use this package.

> **NOTE**
> OWIN applications cannot run in IIS classic mode; they must be run in integrated mode.

> **NOTE**
> OWIN applications must be run in full trust.

This release also includes the Microsoft.Owin.Diagnostics package, which includes middleware to validate a running OWIN application, plus error-page middleware to help investigate failures.

Authentication Components

The following authentication components are available.

- **Microsoft.Owin.Security.ActiveDirectory**. Enables authentication using on-premise or cloud-based directory services.
- **Microsoft.Owin.Security.Cookies** Enables authentication using cookies. This package was previously named `Microsoft.Owin.Security.Forms`.
- **Microsoft.Owin.Security.Facebook** Enables authentication using Facebook's OAuth-based service.
- **Microsoft.Owin.Security.Google** Enables authentication using Google's OpenID-based service.
- **Microsoft.Owin.Security.Jwt** Enables authentication using JWT tokens.

- **Microsoft.Owin.Security.MicrosoftAccount** Enables authentication using Microsoft accounts.
- **Microsoft.Owin.Security.OAuth**. Provides an OAuth authorization server as well as middleware for authenticating bearer tokens.
- **Microsoft.Owin.Security.Twitter** Enables authentication using Twitter's OAuth-based service.

This release also includes the `Microsoft.Owin.Cors` package, which contains middleware for processing cross-origin HTTP requests.

> **NOTE**
> Support for JWT signing has been removed in the final version of Visual Studio 2013.

Entity Framework 6

For a list of new features and other changes in Entity Framework 6, see Entity Framework Version History.

ASP.NET Razor 3

ASP.NET Razor 3 includes the following new features:

- Support for Tab editing. Previously, the **Format Document** command, auto indenting, and auto formatting in Visual Studio did not work correctly when using the **Keep Tabs** option. This change corrects Visual Studio formatting for Razor code for tab formatting.
- Support for URL Rewrite rules when generating links.
- Removal of security transparent attribute.

  > **NOTE**
  > This is a breaking change, and makes Razor 3 incompatible with MVC4 and earlier, while Razor 2 is incompatible with MVC5 or assemblies compiled against MVC5.

Razor 3 issues fixed in Visual Studio 2013 from pre-release versions can be found here.

ASP.NET App Suspend

ASP.NET App Suspend is a game-changing feature in the .NET Framework 4.5.1 that radically changes the user experience and economic model for hosting large numbers of ASP.NET sites on a single machine. For more information, see ASP.NET App Suspend – responsive shared .NET web hosting.

Known Issues and Breaking Changes

This section describes known issues and breaking changes in the ASP.NET and Web Tools for Visual Studio 2013.

NuGet

- New package restore doesn't work on Mono when using SLN file – will be fixed in an upcoming nuget.exe download and NuGet.CommandLine package update.
- New package restore doesn't work with Wix projects – will be fixed in an upcoming nuget.exe download and NuGet.CommandLine package update.
- Automatic Package restore doesn't work for projects under a solution folder – will be fixed in NuGet 2.8.

ASP.NET Web API

1. `ODataQueryOptions<T>.ApplyTo(IQueryable)` doesn't return `IQueryable<T>` always, as we added support for `$select` and `$expand`.

Our earlier samples for `ODataQueryOptions<T>` always casted the return value from `ApplyTo` to `IQueryable<T>`. This worked earlier because the query options that we supported earlier (`$filter`, `$orderby`, `$skip`, `$top`) do not change the shape of the query. Now that we support `$select` and `$expand` the return value from `ApplyTo` will not be `IQueryable<T>` always.

```
// Sample ODataQueryOptions<T> usage from earlier
public IQueryable<Customer> Get(ODataQueryOptions<Customer> query)
{
    IQueryable<customer> result="query.ApplyTo(_customers)" as iqueryable<customer>; return result;
}
```

If you are using the sample code from earlier, it will continue working if the client does not send `$select` and `$expand`. However, if you wish to support `$select` and `$expand` you have to change that code to this.

```
public IHttpActionResult Get(ODataQueryOptions<Customer> query)
{
    IQueryable result = query.ApplyTo(_customers);
    return Ok(result, result.GetType());
}

private IHttpActionResult Ok(object content, Type type)
{
    Type resultType = typeof(OkNegotiatedContentResult<>).MakeGenericType(type);
    return Activator.CreateInstance(resultType, content, this) as IHttpActionResult;
}
```

2. **Request.Url or RequestContext.Url is null during a batch request**

   In a batching scenario, **UrlHelper** is null when accessed from **Request.Url** or **RequestContext.Url**.

   This issue is currently tracked here: BatchRequestContext.Url is null for batching request.

   The workaround for this issue is to create a new instance of **UrlHelper**, as in the following example:

   Creating a new instance of UrlHelper

```
if (RequestContext.Url == null)
{
    RequestContext.Url = new UrlHelper(Request);
}
```

ASP.NET MVC

1. When using MVC5 and OrgAuth, if you have views which do AntiForgerToken validation, you might come across the following error when you post data to the view:

   Error:

   Server Error in '/' Application.

   A claim of type 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier' or 'https://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider' was not present on the provided ClaimsIdentity. To enable anti-forgery token support with claims-based authentication, please verify that the configured claims provider is providing both of these claims on the ClaimsIdentity instances it generates. If the configured claims provider instead uses a different claim type as a unique identifier, it can be configured by setting the static property AntiForgeryConfig.UniqueClaimTypeIdentifier.

   Workaround:

Add the following line in Global.asax to fix it:

```
AntiForgeryConfig.UniqueClaimTypeIdentifier = ClaimTypes.Name;
```

This will be fixed for the next release.

2. After upgrading an MVC4 app to MVC5, build the solution and launch it. You should see the following error:

   [A]System.Web.WebPages.Razor.Configuration.HostSection cannot be cast to [B]System.Web.WebPages.Razor.Configuration.HostSection. Type A originates from 'System.Web.WebPages.Razor, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35' in the context 'Default' at location 'C:\windows\Microsoft.Net\assembly\GAC_MSIL\System.Web.WebPages.Razor\v4.0_2.0.0.0__31bf3856ad364e35\System.Web.WebPages.Razor.dll'. Type B originates from 'System.Web.WebPages.Razor, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35' in the context 'Default' at location 'C:\Windows\Microsoft.NET\Framework\v4.0.30319\Temporary ASP.NET Files\root\6d05bbd0\e8b5908e\assembly\dl3\c9cbca63\f8910382_6273ce01\System.Web.WebPages.Razor.dll'.

   To fix the above error, open *all* the Web.config files (including the ones in the Views folder) in your project and do the following:

   a. Update all occurrences of version "4.0.0.0" of "System.Web.Mvc" to "5.0.0.0".

   b. Update all occurrences of version "2.0.0.0" of "System.Web.Helpers", "System.Web.WebPages" and "System.Web.WebPages.Razor" to "3.0.0.0"

      For example, after you make the above changes, the assembly bindings should look like this:

```
<dependentAssembly>
  <assemblyIdentity name="System.Web.Helpers" publicKeyToken="31bf3856ad364e35" />
  <bindingRedirect oldVersion="1.0.0.0-2.0.0.0" newVersion="3.0.0.0" />
  </dependentAssembly>
  <dependentAssembly>
  <assemblyIdentity name="System.Web.WebPages" publicKeyToken="31bf3856ad364e35" />
  <bindingRedirect oldVersion="1.0.0.0-2.0.0.0" newVersion="3.0.0.0" />
  </dependentAssembly>
  <dependentAssembly>
  <assemblyIdentity name="System.Web.WebPages.Razor" publicKeyToken="31bf3856ad364e35" />
  <bindingRedirect oldVersion="1.0.0.0-3.0.0.0" newVersion="3.0.0.0" />
  </dependentAssembly>
  <dependentAssembly>
  <assemblyIdentity name="System.Web.Mvc" publicKeyToken="31bf3856ad364e35" />
  <bindingRedirect oldVersion="1.0.0.0-4.0.0.0" newVersion="5.0.0.0" />
</dependentAssembly>
```

      For information on upgrading MVC 4 projects to MVC 5, see How to Upgrade an ASP.NET MVC 4 and Web API Project to ASP.NET MVC 5 and Web API 2.

3. When using client-side validation with jQuery Unobtrusive Validation, the validation message is sometimes incorrect for an HTML input element with type='number'. The validation error for a required value ("The Age field is required") is shown when an invalid number is entered instead of the correct message that a valid number is required.

   This issue is commonly found with scaffolded code for a model with an integer property on the Create and Edit views.

   To work around this issue, change the editor helper from:

```
@Html.EditorFor(person => person.Age)
```

To:

```
@Html.TextBoxFor(person => person.Age)
```

4. ASP.NET MVC 5 no longer supports partial trust. Projects linking to the MVC or WebAPI binaries should remove the SecurityTransparent attribute and the AllowPartiallyTrustedCallers attribute. Removing these attributes will eliminate compiler errors such as the following.

```
Attempt by security transparent method ‘MyComponent' to access security critical type
'System.Web.Mvc.MvcHtmlString' failed. Assembly 'PagedList.Mvc, Version=4.3.0.0, Culture=neutral,
PublicKeyToken=abbb863e9397c5e1' is marked with the AllowPartiallyTrustedCallersAttribute, and uses the
level 2 security transparency model. Level 2 transparency causes all methods in
AllowPartiallyTrustedCallers assemblies to become security transparent by default, which may be the
cause of this exception.
```

> Note, as a side effect of this you cannot use 4.0 and 5.0 assemblies in the same application. You need to update all of them to 5.0.

SPA Template with Facebook authorization may cause instability in IE while the web site is hosted in intranet zone

The SPA template provides external log in with Facebook. When the project created with the template is running locally, signing in may cause IE to crash.

Solution:

1. Host the web site in internet zone; or
2. Test the scenario in a browser other than IE.

Web Forms Scaffolding

Web Forms Scaffolding has been removed from VS2013 and will be available in a future update to Visual Studio. However, you can still use scaffolding within a Web Forms project by adding MVC dependencies and generating scaffolding for MVC. Your project will contain a combination of Web Forms and MVC.

To add MVC to your Web Forms project, add a new scaffolded item and select **MVC 5 Dependencies**. Select either Minimal or Full depending on whether you need all of the content files, such as scripts. Then, add a scaffolded item for MVC, which will create views and a controller in your project.

MVC and Web API Scaffolding - HTTP 404, Not Found error

If an error is encountered when adding a scaffolded item to a project, it is possible your project will be left in an inconsistent state. Some of the changes made be scaffolding will be rolled back but other changes, such as the installed NuGet packages, will not be rolled back. If the routing configuration changes are rolled back, users will receive an HTTP 404 error when navigating to scaffolded items.

Workaround:

- To fix this error for MVC, add a new scaffolded item and select MVC 5 Dependencies (either Minimal or Full). This process will add all of the required changes to your project.
- To fix this error for Web API:
  1. Add the WebApiConfig class to your project.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.MapHttpAttributeRoutes();
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

```
Public Module WebApiConfig
    Public Sub Register(ByVal config As HttpConfiguration)
        config.MapHttpAttributeRoutes()
        config.Routes.MapHttpRoute(
          name:="DefaultApi",
          routeTemplate:="api/{controller}/{id}",
          defaults:=New With {.id = RouteParameter.Optional}
        )
    End Sub
End Module
```

2. Configure WebApiConfig.Register in the Application_Start method in Global.asax as follows:

```
public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        GlobalConfiguration.Configure(WebApiConfig.Register);
    }
}
```

```
Public Class WebApiApplication
     Inherits System.Web.HttpApplication

     Sub Application_Start()
       GlobalConfiguration.Configure(AddressOf WebApiConfig.Register)
     End Sub
End Class
```

ASP.NET and Web Tools 2013.2 for Visual Studio 2013 Release Notes

4/10/2020 • 14 minutes to read • Edit Online

by Microsoft

Installation Notes

ASP.NET and Web Tools for Visual Studio 2013.2 are bundled in the main installer and can be downloaded as part of Visual Studio 2013 Update 2.

Documentation

Tutorials and other information about ASP.NET and Web Tools for Visual Studio 2013.2 are available from the ASP.NET web site.

Software Requirements

ASP.NET and Web Tools for Visual Studio 2013.2 requires Visual Studio 2013.

New Features in ASP.NET and Web Tools for Visual Studio 2013.2

The following sections describe the features that have been introduced in the release.

- One ASP.NET Project Templates
- Support SSL when launching Web Applications on IIS Express
- Visual Studio Web Editor Enhancements
- Browser Link
- Support for Azure App Service Web Apps in Visual Studio
- Create remote Azure resources when creating a new Web project
- Web Publish Enhancements
- ASP.NET Scaffolding
- NuGet 2.8.1
- ASP.NET Web Forms
- ASP.NET MVC 5.1.2
- ASP.NET Web API 2.1.2
- ASP.NET Web Pages 3.1.2
- Entity Framework 6.1
- ASP.NET Identity 2.0.0
- Microsoft OWIN components
- ASP.NET SignalR 2.0.2

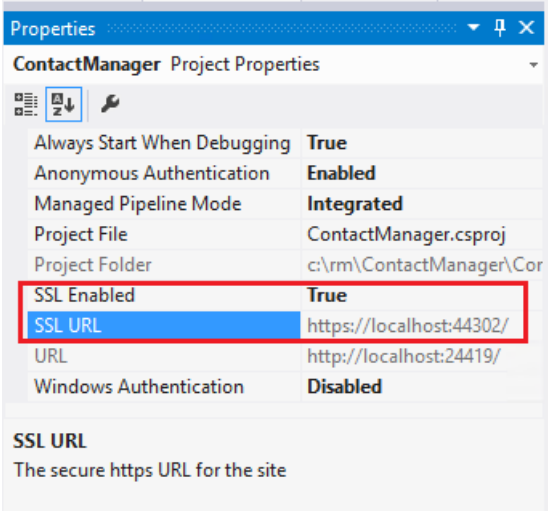
One ASP.NET Project Templates

- Updates to ASP.NET Project templates to support Account confirmation and Password Reset.
- Update ASP.NET Web API template to support authentication using On Premises Organizational Accounts.
- The ASP.NET SPA template now contains authentication that is based on MVC and server side views. The template has a WebAPI controller which can only be accessed by authenticated users.

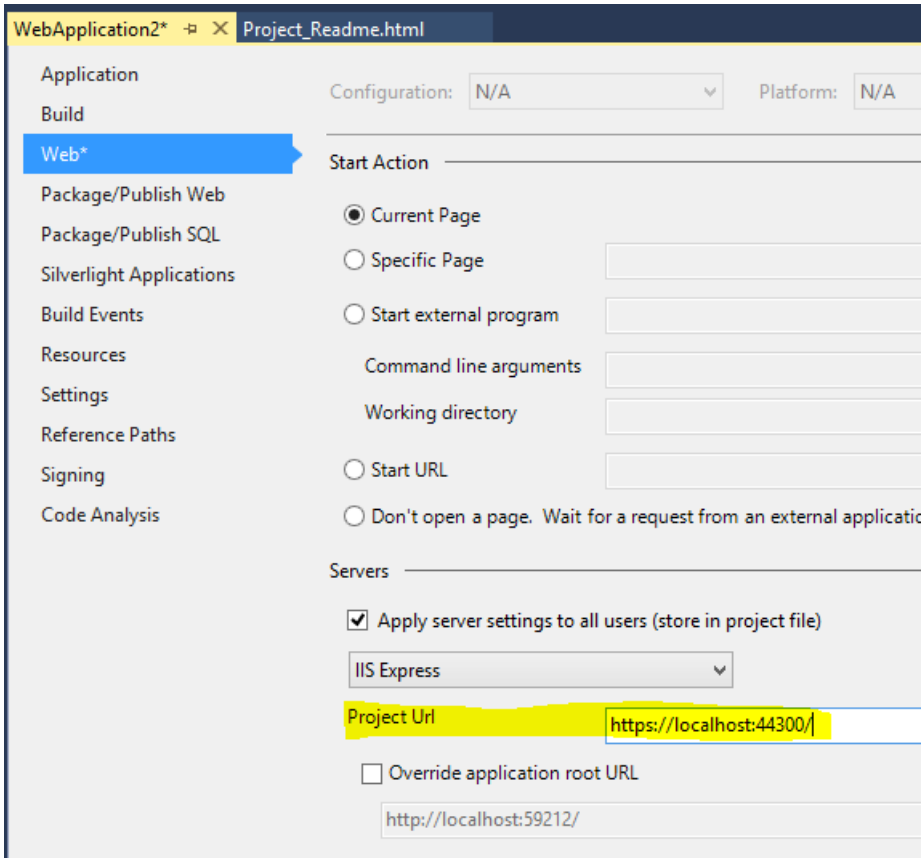
Support SSL when launching Web Applications on IIS Express

To eliminate the security warning when browsing and debugging HTTPS on localhost, we added a dialog to allow Internet Explorer and Chrome to trust the self-signed IIS express SSL certificate.

For example, a web project property can be set to use SSL. Click F4 to bring up the properties dialog. Change **SSL Enabled** to true. Copy the SSL URL.

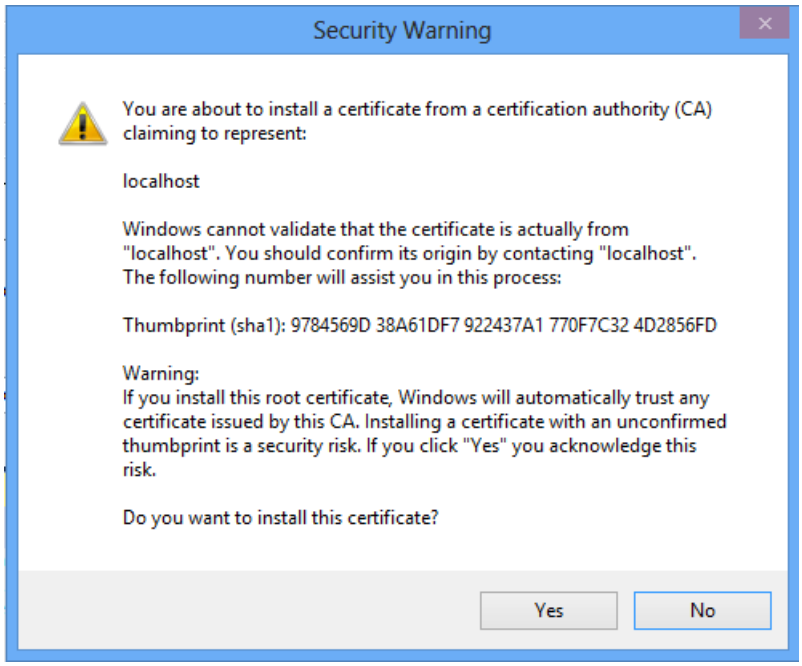


Set the web project property page web tab to use the HTTPS based URL (The SSL URL will be `https://localhost:44300/` unless you've previously created SSL Web Sites.)

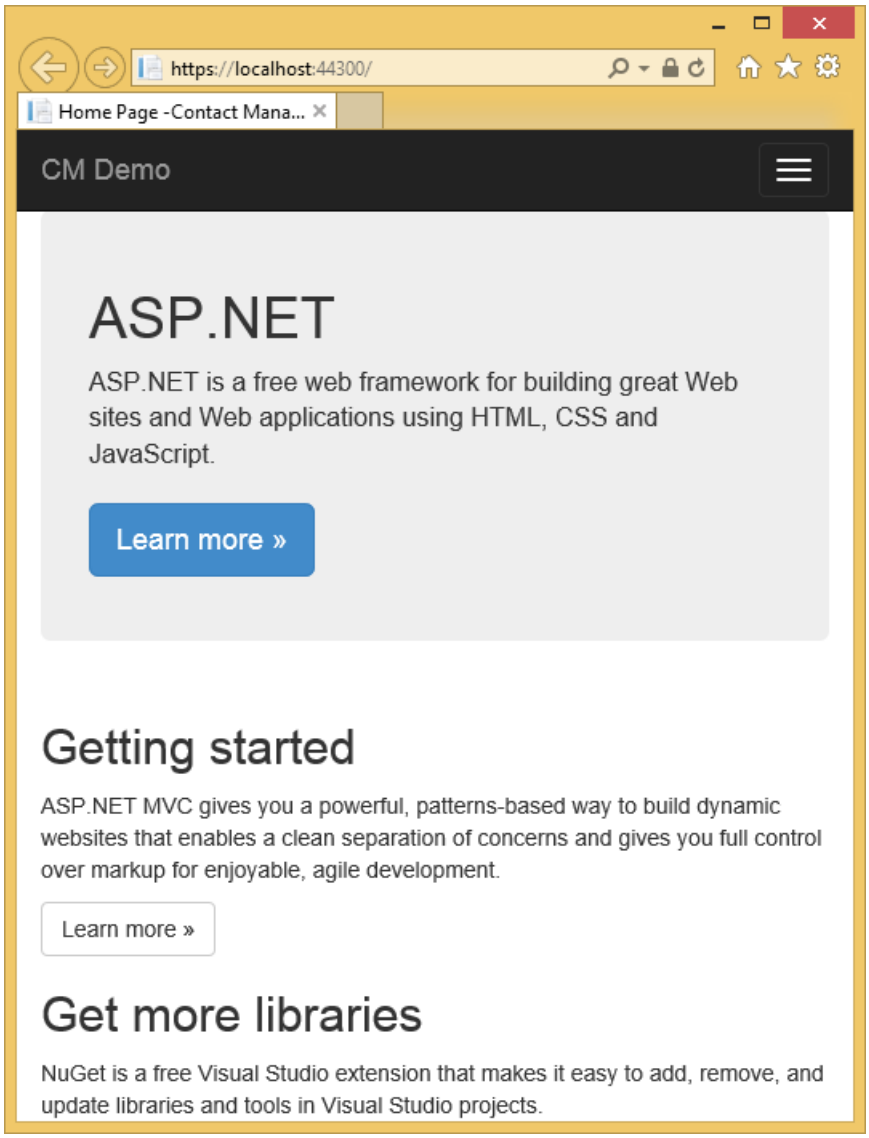


Press CTRL+F5 to run the application. Follow the instructions to trust the self-signed certificate that IIS Express has generated.

Read the **Security Warning** dialog and then click **Yes** if you want to install the certificate representing localhost.



The site will be shown in IE or Chrome without the certificate warning in the browser.



Firefox uses its own certificate store, so it will display a warning.

Visual Studio Web Editor Enhancements

- **New JSON project item and editor**: We have added a JSON project item and editor to Visual Studio. Current JSON editor features include colorization, syntax validation, brace completion, outlining, tools option setting and more.

```
{"web-app": {
  "servlet": [
    {
      "servlet-name": "cofaxCDS",
      "servlet-class": "org.cofax.cds.CDSServlet",
      "init-param": ...},
    {
      "servlet-name": "cofaxEmail",
      "servlet-class": "org.cofax.cds.EmailServlet",
      "init-param": {
      "mailHost": "mail1",
      "mailHostOverride": "mail2"}},
    {
      "servlet-name": "cofaxAdmin",
      "servlet-class": "org.cofax.cds.AdminServlet"},

    {
      "servlet-name": "fileServlet",
      "servlet-class": "org.cofax.cds.FileServlet"},
    {
      "servlet-name": "cofaxTools",
      "servlet-class": "org.cofax.cms.CofaxToolsServlet",
      "init-param": {
        "templatePath": "toolstemplates/",
        "log": 1,
        "logLocation": "/usr/local/tomcat/logs/CofaxTools.log",
        "logMaxSize": "",
        "dataLog": 1,
        "dataLogLocation": "/usr/local/tomcat/logs/dataLog.log",
        "dataLogMaxSize": "",
        "removePageCache": "/content/admin/remove?cache=pages&id=",
        "removeTemplateCache": "/content/admin/remove?cache=templates&id=",
        "fileTransferFolder": "/usr/local/tomcat/webapps/content/fileTransferFolder",
        "lookInContext": 1,
        "adminGroupID": 4,
        "betaServer": true}}],
  "servlet-mapping": {
    "cofaxCDS": "/",
    "cofaxEmail": "/cofaxutil/aemail/*",
    "cofaxAdmin": "/admin/*",
    "fileServlet": "/static/*",
    "cofaxTools": "/tools/*"},

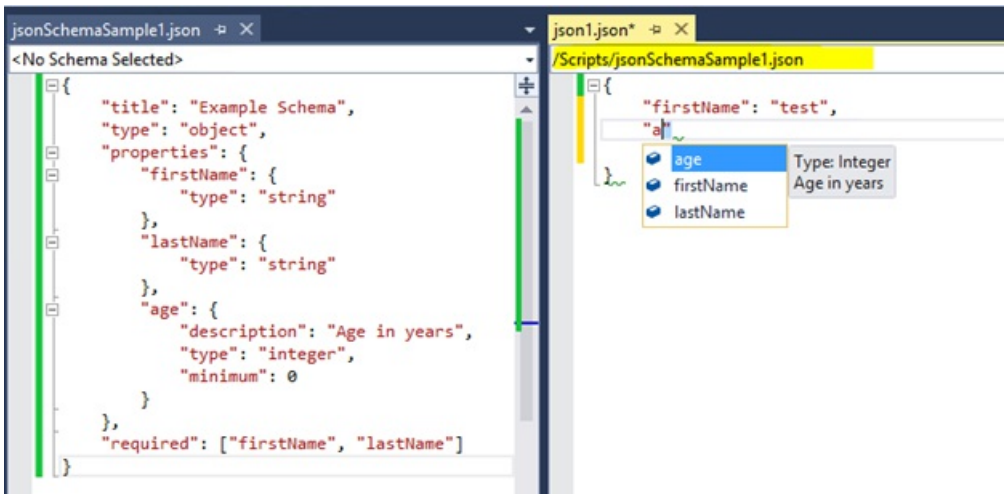
  "taglib": {
    "taglib-uri": "cofax.tld",
    "taglib-location": "/WEB-INF/tlds/cofax.tld"}}}
```

IntelliSense now supports JSON Schema v3 and v4. There is a schema combo box to choose existing schemas, edit the local schema path, or simply drag drop a project JSON file to it to get the relative path.

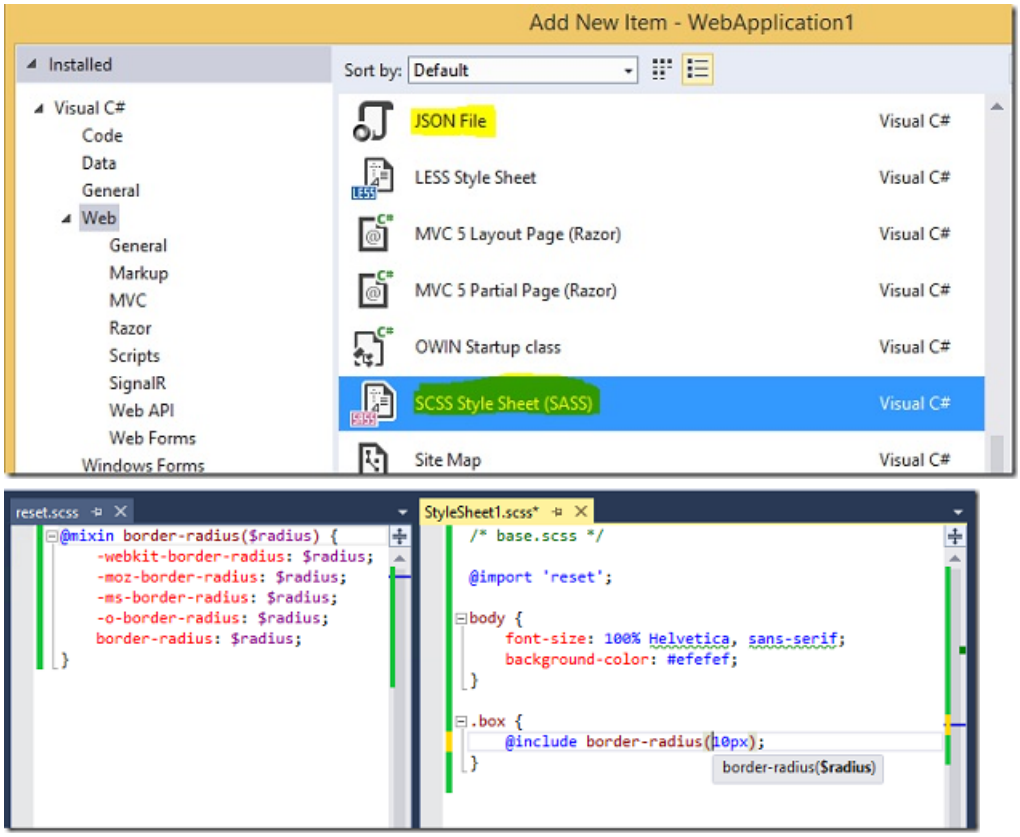
```
http://json-schema.org/address
{
    "country-name": "USA",
    ""
}
```

country-name
extended-address
locality
postal-code
post-office-box
region
street-address

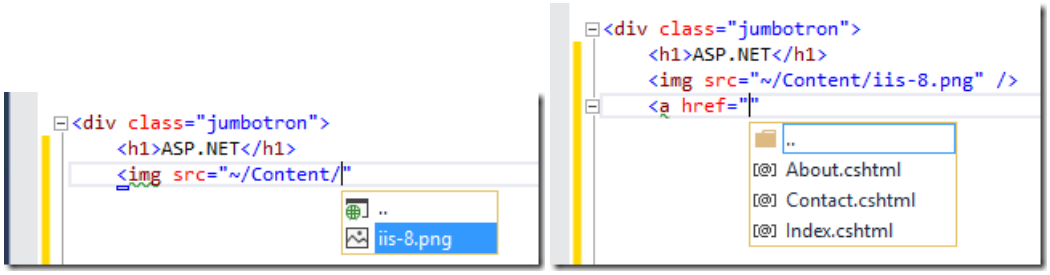
Type: String

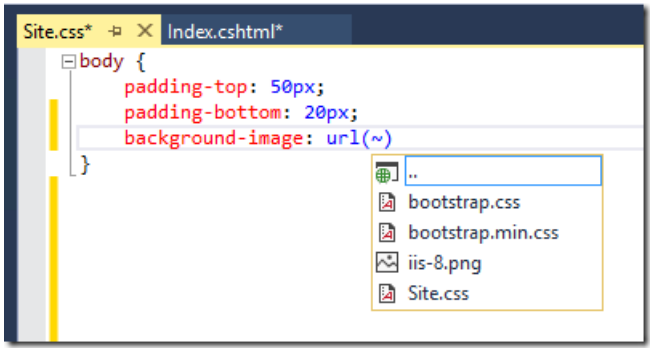


- **New Sass (SCSS) editor**: We added LESS in VS2013 RTM, and we now have a Sass project item and editor. Sass editor features are comparable to the LESS editor, and include colorization, variable and Mixins IntelliSense, comment/uncomment, quick info, formatting, syntax validation, outlining, goto definition, color picker, tools option setting etc.

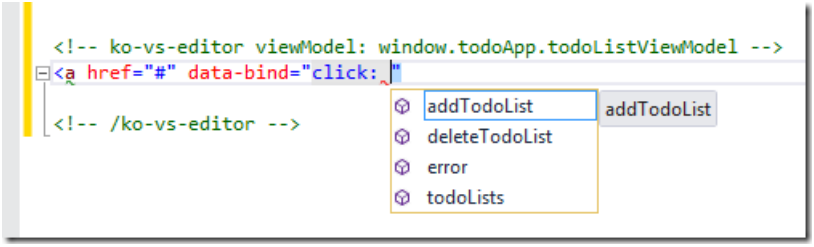


- **New URL Picker in HTML, Razor, CSS, LESS and Sass documents:** VS 2013 shipped with no URL picker outside of Web Forms pages. The new URL picker for HTML, Razor, CSS, LESS and Sass editors is a dialog-free, fluent typing picker that understands '..' and filters file lists appropriately for img tags and links.





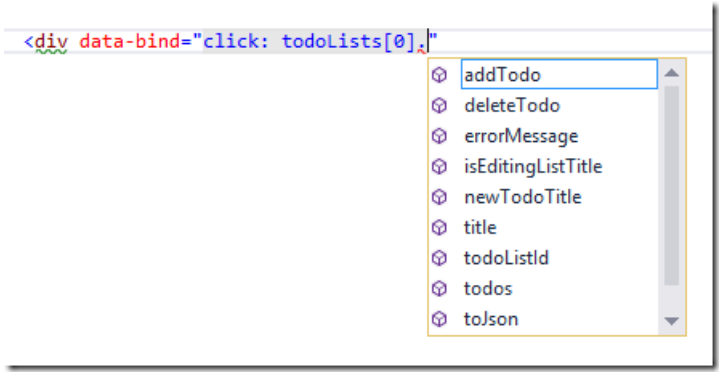
- **Updates to LESS editor by adding more features**
- **Knockout Intellisense Upgrade**: We added a non-standard KnockOut syntax for VS intelliSense, "ko-vs-editor viewModel:" syntax. It can be used to bind to multiple view models on a page using comments in the form:



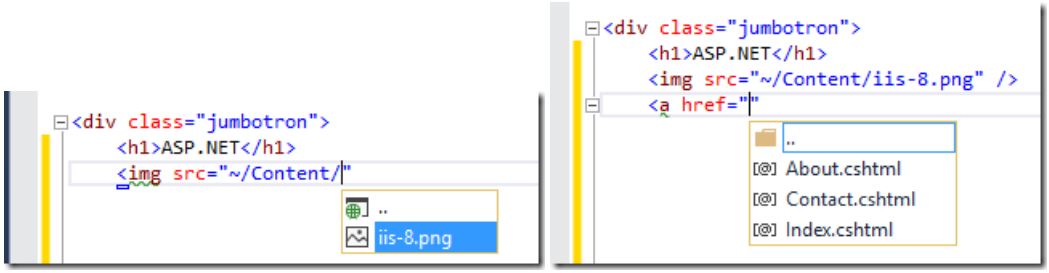
We also added support for nested ViewModel IntelliSense, so you may drill into deeply nested objects on the ViewModel.

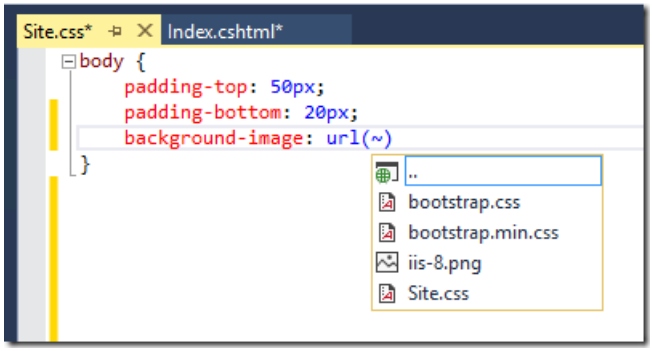
```
<div data-bind="text: foo.bar.baz.etc" />
```

The IntelliSense displayed is the full IntelliSense of the JavaScript Object.



- **New URL Picker in HTML, Razor, CSS, LESS and Sass documents**: VS 2013 shipped with no URL picker outside of Web Forms pages. The new URL picker for HTML, Razor, CSS, LESS and Sass editors is a dialog-free, fluent typing picker that understands '..' and filters file lists appropriately for img tags and links.





Browser Link

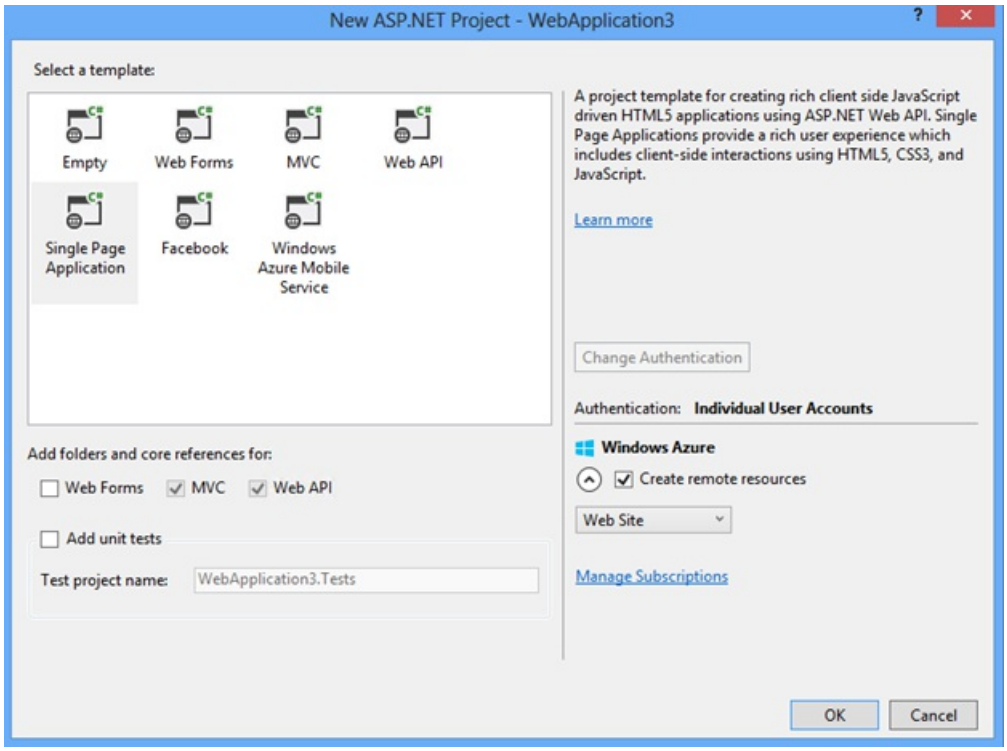
- Browser Link now supports HTTPS connections and will list that in Dashboard with other connections as long as the certificate is trusted by browser.
- Static HTML source mapping
- SPA support for mapping data
- Auto-update mapping data

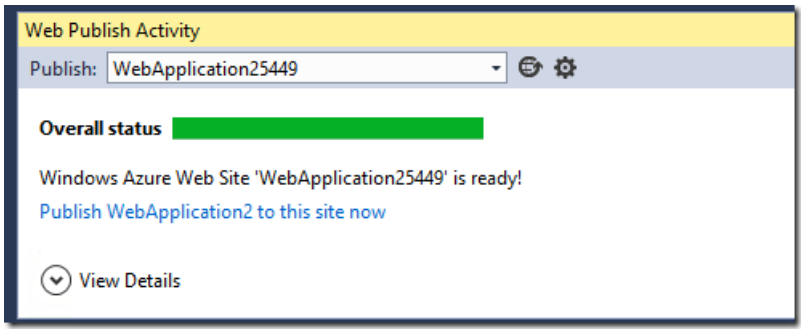
Support for Azure App Service Web Apps in Visual Studio

- **Support Azure sign in.**
- **Remote debugging and Remote View for web apps**: We now support remote debugging for web apps in Azure App Service and remote view of web app content files in the server explorer.

Create remote Azure resources when creating a new Web project

We added an Azure "Create Remote Resources" checkbox on the new web application dialog. By choosing it, you will be able to integrate the experience of creating a new web application, setting up the Azure publishing site for testing, and creating publishing profile in a few simple steps.





Web Publish Enhancements

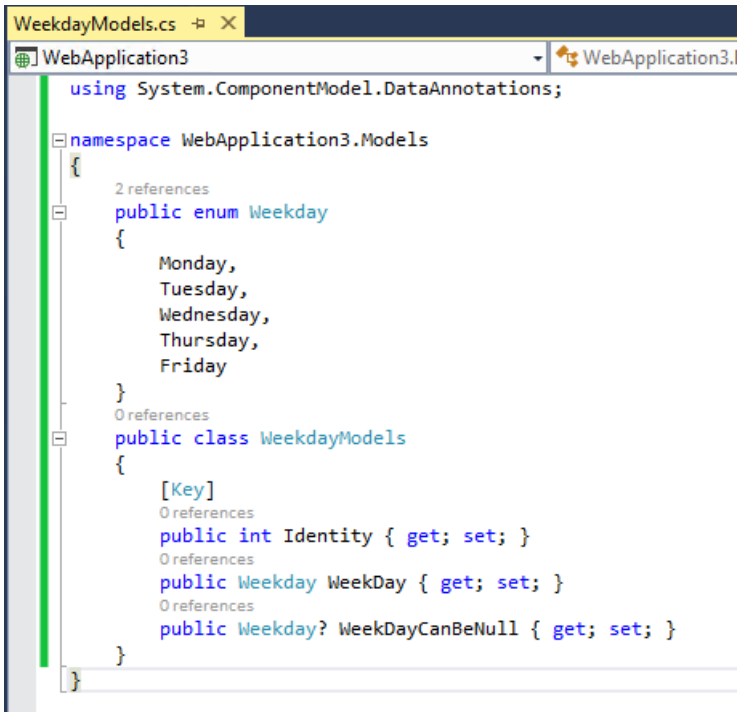
- Improve User experience for publishing.

ASP.NET Scaffolding

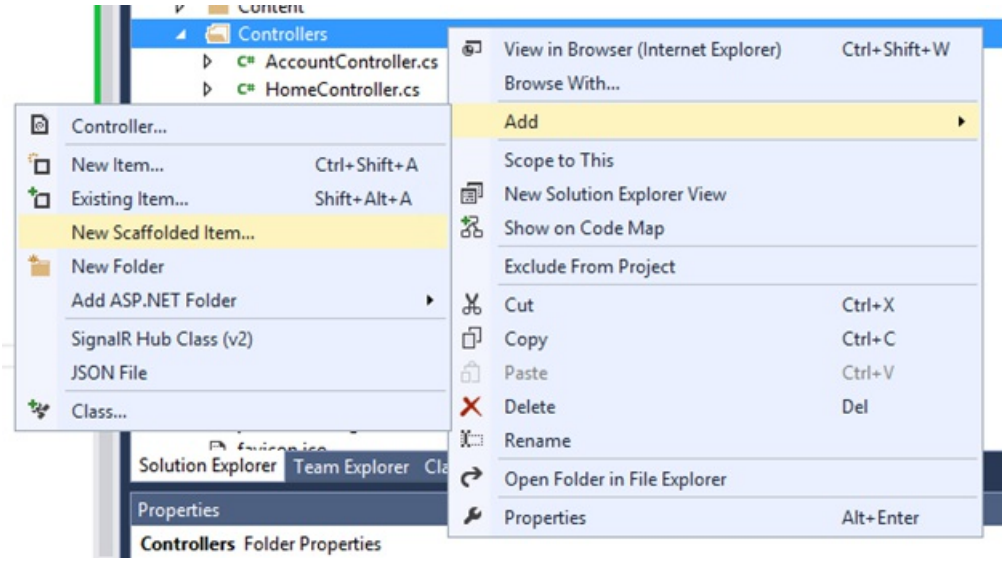
- **Enum support:** If your model is using Enums, then the MVC Scaffolder will generate dropdown for Enum. This uses the Enum helpers in MVC.
- **Bootstrap support**: Updated the EditorFor templates in MVC Scaffolding so they use the Bootstrap classes.
- **Package support**: MVC and Web API Scaffolders will add 5.1 packages for MVC and Web API

The following screenshots demonstrate scaffolding models.

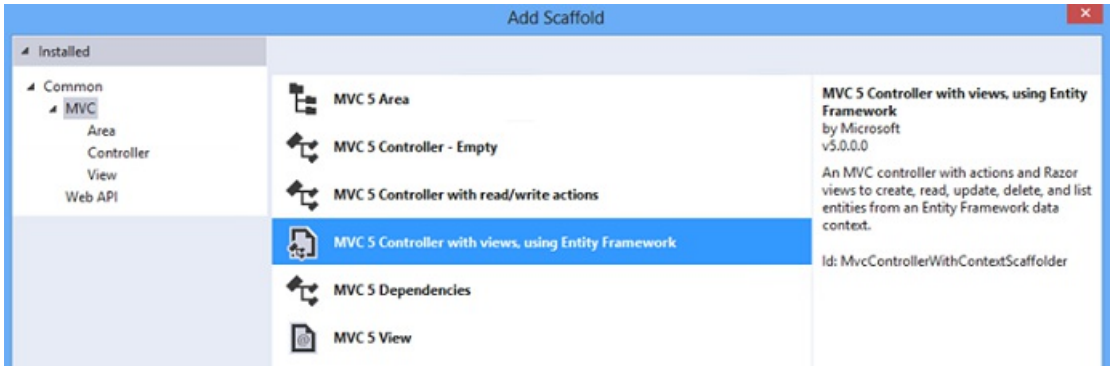
- Model code:



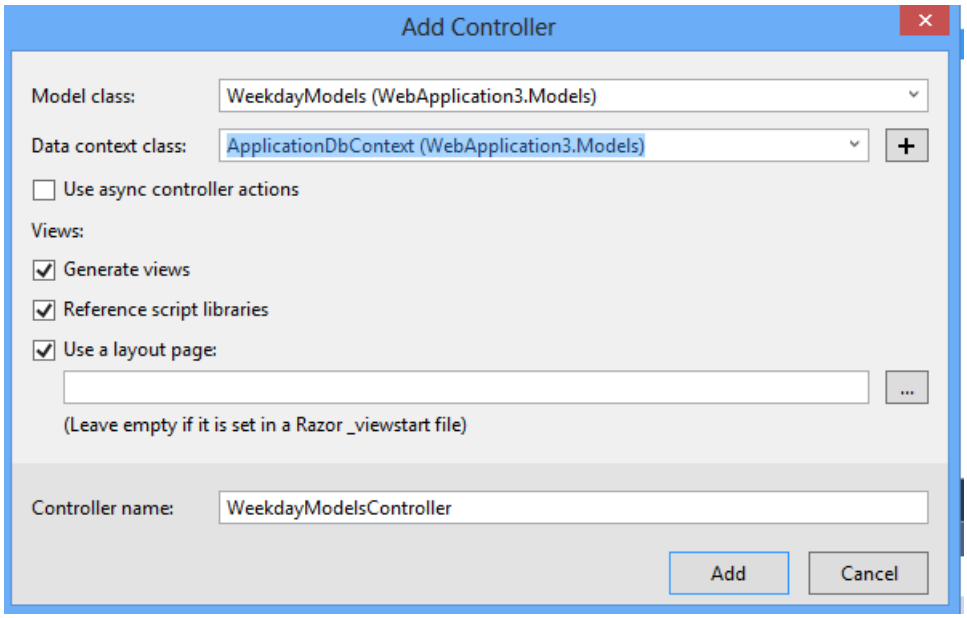
- Compile the model code, right-click, and select **Add**, **New Scaffolded Item**.



- Choose **MVC5 Controller with views, using Entity Framework**:



- **Add Controller** using the model:



- Check the generated code, for example Views/WeekdayModels/Edit.cshtml contains `@Html.EnumDropDownListFor`:

```
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()

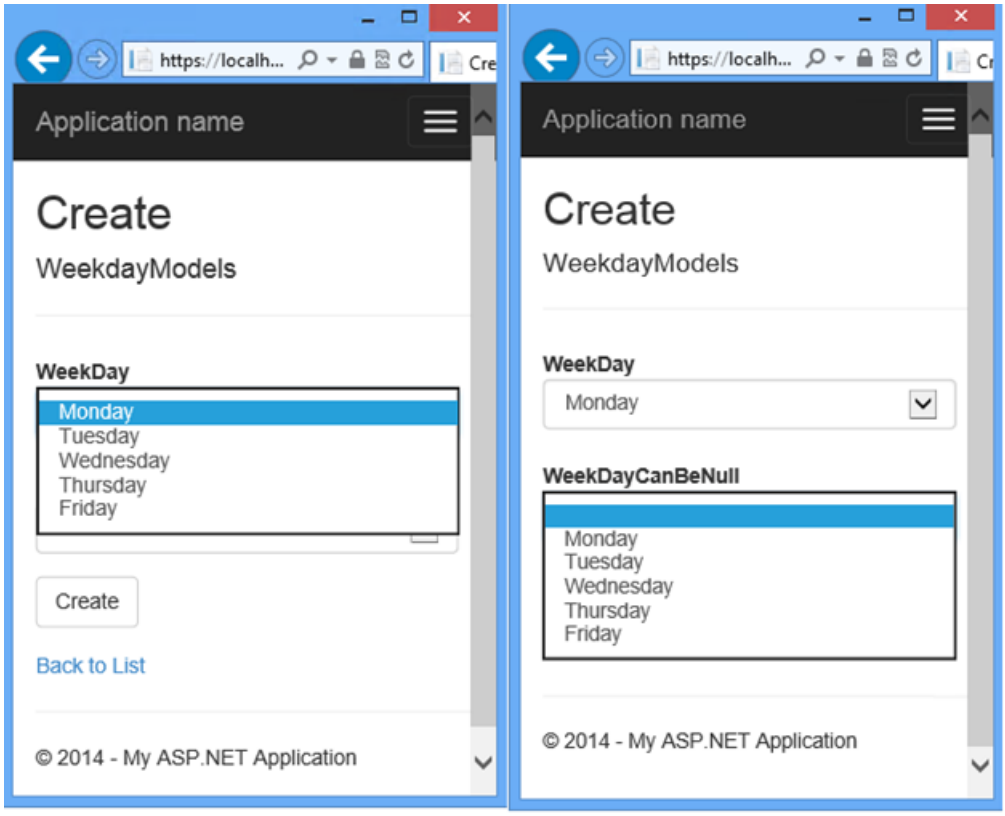
    <div class="form-horizontal">
        <h4>WeekdayModels</h4>
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        @Html.HiddenFor(model => model.Identity)

        <div class="form-group">
            @Html.LabelFor(model => model.WeekDay, htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EnumDropDownListFor(model => model.WeekDay, htmlAttributes: new { @class = "form-control" })
                @Html.ValidationMessageFor(model => model.WeekDay, "", new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.WeekDayCanBeNull, htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EnumDropDownListFor(model => model.WeekDayCanBeNull, htmlAttributes: new { @class = "form-control" })
                @Html.ValidationMessageFor(model => model.WeekDayCanBeNull, "", new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
}
```

- Run the page to see the enum combobox generated, notice that if a value can be null, an empty string can be chosen for the combo box. For example, the **Create** page shows the following:



NuGet 2.8.1

NuGet 2.8.1 RTM will be released in April 2014. Here are the salient points from the release notes, but please check the full release notes for more information on these changes.

- **Target Windows Phone 8.1 Applications**: NuGet 2.8.1 now supports targeting Windows Phone 8.1 Applications using the target framework monikers 'WindowsPhoneApp', 'WPA', 'WindowsPhoneApp81', and 'WPA81'.

- **Patch Resolution for Dependencies**: When resolving package dependencies, NuGet has historically implemented a strategy of selecting the lowest major and minor package version which satisfies the dependencies on the package. Unlike the major and minor version, however, the patch version was always resolved to the highest version. Though the behavior was well-intentioned, it created a lack of determinism for installing packages with dependencies.

- **DependencyVersion Switch**: Though NuGet 2.8 changes the *default* behavior for resolving dependencies, it also adds more precise control over dependency resolution process via the -DependencyVersion switch in the package manager console. The switch enables resolving dependencies to the lowest possible version (default behavior), the highest possible version, or the highest minor or patch version. This switch only works for install-package in the powershell command.

- **DependencyVersion Attribute**: In addition to the -DependencyVersion switch detailed above, NuGet has also allowed for the ability to set a new attribute in the nuget.config file defining what the default value is, if the -DependencyVersion switch is not specified in an invocation of install-package. This value will also be respected by the NuGet Package Manager Dialog for any install package operations. To set this value, add the attribute below to your nuget.config file:

  ```
  <config> <add key="dependencyversion" value="Highest" /> </config>
  ```

- **Preview NuGet Operations With -WhatIf**: Some NuGet packages can have deep dependency graphs, and as such, it can be helpful during an install, uninstall, or update operation to first see what will happen. NuGet 2.8 adds the standard PowerShell -what if switch to the install-package, uninstall-package, and update-package commands to enable visualizing the entire closure of packages to which the command will be applied.

- **Downgrade Package**: It is not uncommon to install a prerelease version of a package in order to investigate new features and then decide to roll back to the last stable version. Prior to NuGet 2.8, this was a multi-step process of uninstalling the prerelease package and its dependencies, and then installing the earlier version. With NuGet 2.8, however, the update-package will now roll back the entire package closure (e.g. the package's dependency tree) to the previous version.

- **Development Dependencies**: Many different types of capabilities can be delivered as NuGet packages - including tools that are used for optimizing the development process. These components, while they can be instrumental in developing a new package, should not be considered a dependency of the new package when it is later published. NuGet 2.8 enables a package to identify itself in the .nuspec file as a developmentDependency. When installed, this metadata will also be added to the packages.config file of the project into which the package was installed. When that packages.config file is later analyzed for NuGet dependencies during nuget.exe pack, it will exclude those dependencies marked as development dependencies.

- **Individual packages.config Files for Different Platforms**: When developing applications for multiple target platforms, it is common to have different project files for each of the respective build environments. It is also common to consume different NuGet packages in different project files, as packages have varying levels of support for different platforms. NuGet 2.8 provides improved support for this scenario by creating different packages.config files for different platform-specific project files.

- **Fallback to Local Cache**: Though NuGet packages are typically consumed from a remote gallery such as the NuGet gallery using a network connection, there are many scenarios where the client is not connected. Without a network connection, the NuGet client was not able to successfully install packages - even when those packages were already on the client's machine in the local NuGet cache. NuGet 2.8 adds automatic cache fallback to the package manager console.

  The cache fallback feature does not require any specific command arguments. Additionally, cache fallback currently works only in the package manager console - the behavior does not currently work in the package manager dialog.

- **Bug Fixes**: One of the major bug fixes made was performance improvement in the update-package -reinstall command.

  In addition to these features and the aforementioned performance fix, this release of NuGet also includes many other bug fixes. There were 181 total issues addressed in the release. For a full list of the work items

fixed in NuGet 2.8, please view the NuGet Issue Tracker for this release.

ASP.NET Web Forms

- The Web Forms templates now show how to do Account Confirmation and Password Reset for ASP.NET Identity.
- The Entity Data Source control and the Dynamic Data Provider for Entity Framework 6. For more details please see the following MSDN blog: Dynamic Data provider and EntityDataSource control for Entity Framework 6.

ASP.NET MVC 5.1.2

- Attribute Routing Improvements
- Bootstrap support for editor templates
- Enum support in views
- Unobtrusive support for MinLength/ MaxLength attributes
- Supporting the 'this' context in Unobtrusive Ajax
- Various bug fixes

ASP.NET Web API 2.1.2

- Global error handling
- Attribute routing enhancements
- Help page improvements
- IgnoreRoute support
- BSON media-type formatter
- Better support for async filters
- Query parsing for the client formatting library
- Various bug fixes

ASP.NET Web Pages 3.1.2

- Various bug fixes

Entity Framework 6.1

Entity Framework has been updated to version 6.1 for both runtime and tooling. Entity Framework (EF) 6.1 is a minor update to Entity Framework 6 and includes a number of bug fixes and new features. For detailed information on EF6.1, including links to documentation for the new features, see Entity Framework Version History. The new features in this release include:

- **Tooling consolidation** provides a consistent way to create a new EF model. This feature extends the ADO.NET Entity Data Model wizard to support creating Code First models, including reverse engineering from an existing database. These features were previously available in Beta quality in the EF Power Tools.
- **Handling of transaction commit failures** provides the new System.Data.Entity.Infrastructure.CommitFailureHandler which makes use of the newly introduced ability to intercept transaction operations. The **CommitFailureHandler** allows automatic recovery from connection failures whilst committing a transaction.
- **IndexAttribute** allows indexes to be specified by placing an attribute on a property (or properties) in your Code First model. Code First will then create a corresponding index in the database.
- **The public mapping API** provides access to the information EF has on how properties and types are mapped to columns and tables in the database. In past releases this API was internal.
- **Ability to configure interceptors via the App/Web.config file**(allowing interceptors to be added without recompiling the application).
- **DatabaseLogger** is a new interceptor that makes it easy to log all database operations to a file. In combination with the previous feature, this allows you to easily switch on logging of database operations for a deployed application, without the need to recompile.
- **Migrations model change detection** has been improved so that scaffolded migrations are more accurate;

performance of the change detection process has also been greatly enhanced.

- **Performance improvements** including reduced database operations during initialization, optimizations for null equality comparison in LINQ queries, faster view generation (model creation) in more scenarios, and more efficient materialization of tracked entities with multiple associations.

ASP.NET Identity 2.0.0

- **Two-factor authentication**: ASP.NET Identity now supports two-factor authentication. Two-factor authentication provides an extra layer of security to your user accounts in the case where your password gets compromised. There is also protection for brute force attacks against the two factor codes.
- **Account Lockout:** Provides a way to lock out the user if the user enters their password or two-factor codes incorrectly. The number of invalid attempts and the timespan for the users are locked out can be configured. A developer can optionally turn off Account Lockout for certain user accounts should they need to.
- **Account Confirmation:** The ASP.NET Identity system now supports Account Confirmation. This is a fairly common scenario in most websites today where, when you register for a new account on the website, you are required to confirm your email before you can do anything in the website. Email Confirmation is useful because it prevents bogus accounts from being created. This is extremely useful if you are using email as a method of communicating with the users of your website such as Forum sites, banking, ecommerce, or social web sites.
- **Password Reset:** Password Reset is a feature where the user can reset their passwords if they have forgotten their password.
- **Security Stamp (Sign out everywhere):** Supports a way to regenerate the Security Token for the user in cases when the User changes their password or any other security related information such as removing an associated login (such as Facebook, Google, Microsoft Account and so on). This is needed to ensure that any tokens generated with the old password are invalidated. In the sample project, if you change the user's password then a new token is generated for the user and any previous tokens are invalidated. This feature provides an extra layer of security to your application since when you change your password, you will be logged out from everywhere (all other browsers) where you have logged into this application.
- **Make the type of Primary Key be extensible for Users and Roles**: In ASP.NET Identity 1.0, the type of primary key for table Users and Roles was strings. This means when the ASP.NET Identity system was persisted in SQL Server by using Entity Framework, we were using nvarchar. There were many discussions around this default implementation on Stack Overflow and based on the incoming feedback. We have provided an extensibility hook where you can specify what should be the primary key of your Users and Roles table. This extensibility hook is particularly useful if you are migrating your application and the application was storing UserIds are GUIDs or ints.
- **Support IQueryable on Users and Roles**: Added support for IQueryable on UsersStore and RolesStore, you can easily get the list of Users and Roles.
- **Support Delete operation through the UserManager**
- **Indexing on UserName**: In ASP.NET Identity Entity Framework implementation, we have added a unique index on the Username by using the new IndexAttribute in EF 6.1.0. This makes sure that Usernames are always unique and there was no race condition in which you could end up with duplicate usernames.
- **Enhanced Password Validator:** The password validator that was shipped in ASP.NET Identity 1.0 was a fairly basic password validator that was only validating the minimum length. There is a new password validator that gives you more control over the complexity of the password. Please note that even if you turn on all the settings in this password, we do encourage you to enable two-factor authentication for the user accounts.
- **IdentityFactory Middleware/ CreatePerOwinContext**:

- **User Manager**: You can use Factory implementation to obtain an instance of UserManager from the OWIN context. This pattern is similar to what we use for getting AuthenticationManager from OWIN context for SignIn and SignOut. This is a recommended way of obtaining an instance of UserManager per request for the application.
- **DbContextFactory**: ASP.NET Identity uses Entity Framework for persisting the Identity system in SQL Server. To do this the Identity System has a reference to the ApplicationDbContext. The DbContextFactory Middleware returns an instance of the ApplicationDbContext per request that you can use in your application.

- **ASP.NET Identity Samples NuGet package**: The Samples NuGet package can make it easier to install and run samples for ASP.NET Identity and follow the best practices. This is a sample ASP.NET MVC application. Please modify the code to suit your application before you deploy this in production. The sample should be installed in an empty ASP.NET application. For more information about the package, go to the following blog post: Announcing RTM of ASP.NET Identity 2.0.0

Microsoft OWIN components

There were lots of bugs that were fixed in this release. Please see the release notes for the 2.1.0 release for more detailed information.

ASP.NET SignalR 2.0.2

There were lots of bugs that were fixed in this release. Please see the release notes for the 2.0.2 release for more detailed information.

Hands On Lab: Visual Studio 2013 Web Tools

2/20/2020 • 17 minutes to read • Edit Online

by Web Camps Team

Download Web Camps Training Kit

> Visual Studio is an excellent development environment for .NET-based Windows and web projects. It includes a powerful text editor that can easily be used to edit standalone files without a project.
>
> Visual Studio maintains a full-featured parse tree as you edit each file. This allows Visual Studio to provide unparalleled auto-completion and document-based actions while making the development experience much faster and more pleasant. These features are especially powerful in HTML and CSS documents.
>
> All of this power is also available for extensions, making it simple to extend the editors with powerful new features to suit your needs. Web Essentials is a collection of (mostly) web-related enhancements to Visual Studio. It includes lots of new IntelliSense completions (especially for CSS), new Browser Link features, automatic JSHint for JavaScript files, new warnings for HTML and CSS, and many other features that are essential to modern web development.
>
> All sample code and snippets are included in the Web Camps Training Kit, available at https://aka.ms/webcamps-training-kit.

Overview

Objectives

In this hands-on lab, you will learn how to:

- Use new HTML editor features included in Web Essentials such as rich HTML5 code snippets and Zen coding
- Use new CSS editor features included in Web Essentials such as the Color picker and Browser matrix tooltip
- Use new JavaScript editor features included in Web Essentials such as Extract to File and IntelliSense for all HTML elements
- Exchange data between your browser and Visual Studio using Browser Link

Prerequisites

The following is required to complete this hands-on lab:

- Microsoft Visual Studio Professional 2013 or greater
- Web Essentials 2013
- Google Chrome

Setup

In order to run the exercises in this hands-on lab, you will need to set up your environment first.

1. Open a Windows Explorer window and browse to the lab's **Source** folder.
2. Right-click **Setup.cmd** and select **Run as administrator** to launch the setup process that will configure your environment and install the Visual Studio code snippets for this lab.
3. If the User Account Control dialog box is shown, confirm the action to proceed.

> **NOTE**
> Make sure you have checked all the dependencies for this lab before running the setup.

Using the Code Snippets

Throughout the lab document, you will be instructed to insert code blocks. For your convenience, most of this code is provided as Visual Studio Code Snippets, which you can access from within Visual Studio 2013 to avoid having to add it manually.

> **NOTE**
> Each exercise is accompanied by a starting solution located in the **Begin** folder of the exercise that allows you to follow each exercise independently of the others. Please be aware that the code snippets that are added during an exercise are missing from these starting solutions and may not work until you have completed the exercise. Inside the source code for an exercise, you will also find an **End** folder containing a Visual Studio solution with the code that results from completing the steps in the corresponding exercise. You can use these solutions as guidance if you need additional help as you work through this hands-on lab.

Exercises

This hands-on lab includes the following exercises:

1. Working with Browser Link and Web Essentials
2. Taking Advantage of Code Snippets and IntelliSense

> **NOTE**
> When you first start Visual Studio, you must select one of the predefined settings collections. Each predefined collection is designed to match a particular development style and determines window layouts, editor behavior, IntelliSense code snippets, and dialog box options. The procedures in this lab describe the actions necessary to accomplish a given task in Visual Studio when using the **General Development Settings** collection. If you choose a different settings collection for your development environment, there may be differences in the steps that you should take into account.

Exercise 1: Working with Browser Link and Web Essentials

Web Essentials is a Visual Studio extension that adds a variety of useful features for modern web development, mostly focused on making the web development experience much faster and more pleasant. You can install Web Essentials from the Extension Gallery in Visual Studio.

Browser Link is a new feature included in Visual Studio 2013 that provides a channel between the Visual Studio IDE and any open browser to exchange data between your web application and Visual Studio. Web Essentials extends Browser Link with tools to manipulate the DOM object model and the CSS styles of your web pages directly from the browser.

In this exercise, you will explore some of the features supported by **Web Essentials** and **Browser Link** to enhance a simple quiz page.

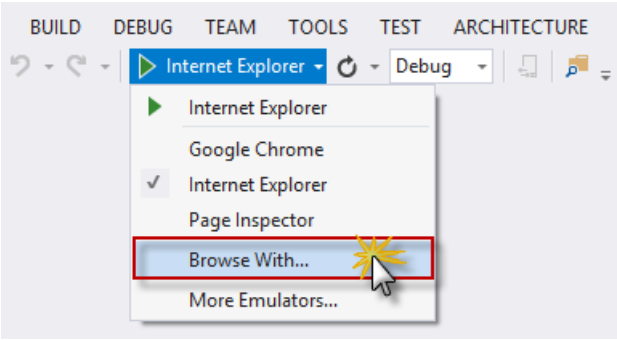
Task 1 - Running the Project in Multiple Browsers

In this task, you will configure your web application to run in multiple browsers at once, which is useful for cross-browser testing.

1. Open **Microsoft Visual Studio**.
2. In the **File** menu, select **Open | Project/Solution...** and browse to **Ex1-**

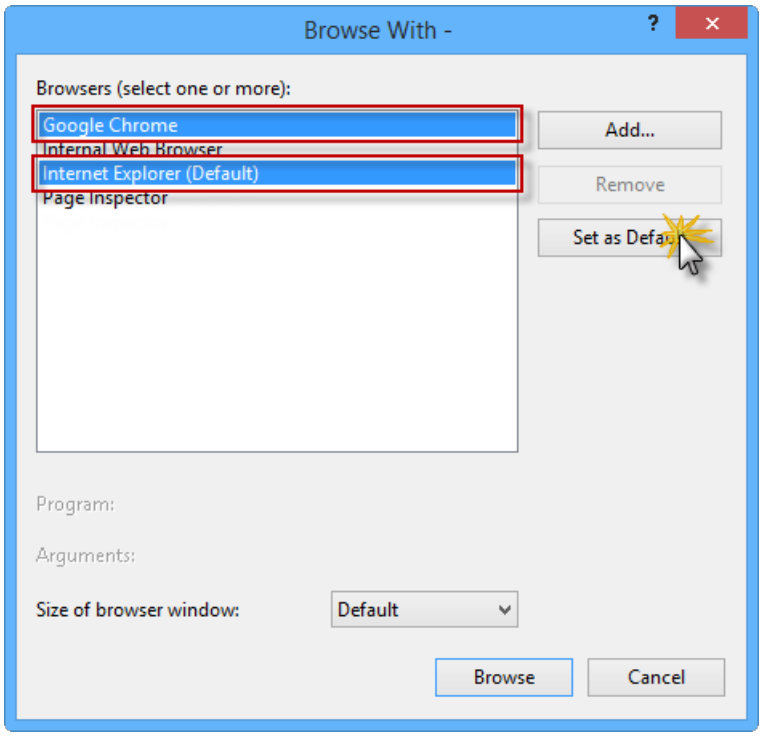
WorkingwithBrowserLinkandWebEssentials\Begin in the **Source** folder of the lab (C:\WebCampsTK\HOL\VSWebTooling\Source). Select **Begin.sln** and click **Open**.

3. In the Visual Studio toolbar, expand the browser menu and select **Browse With...**.



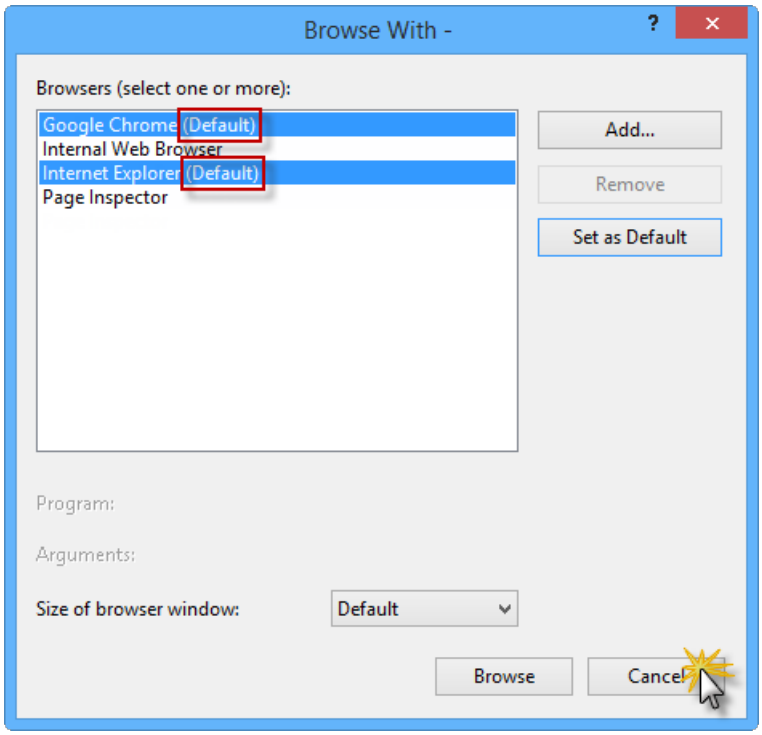
Browse With menu option

4. In the **Browse With** dialog box, select both **Google Chrome** and **Internet Explorer** by holding down the **CTRL** key and click **Set as Default**.



Selecting multiple default browsers

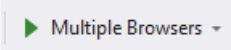
5. Both Google Chrome and Internet Explorer should now appear as the default browsers. Click **Cancel** to close the dialog box.



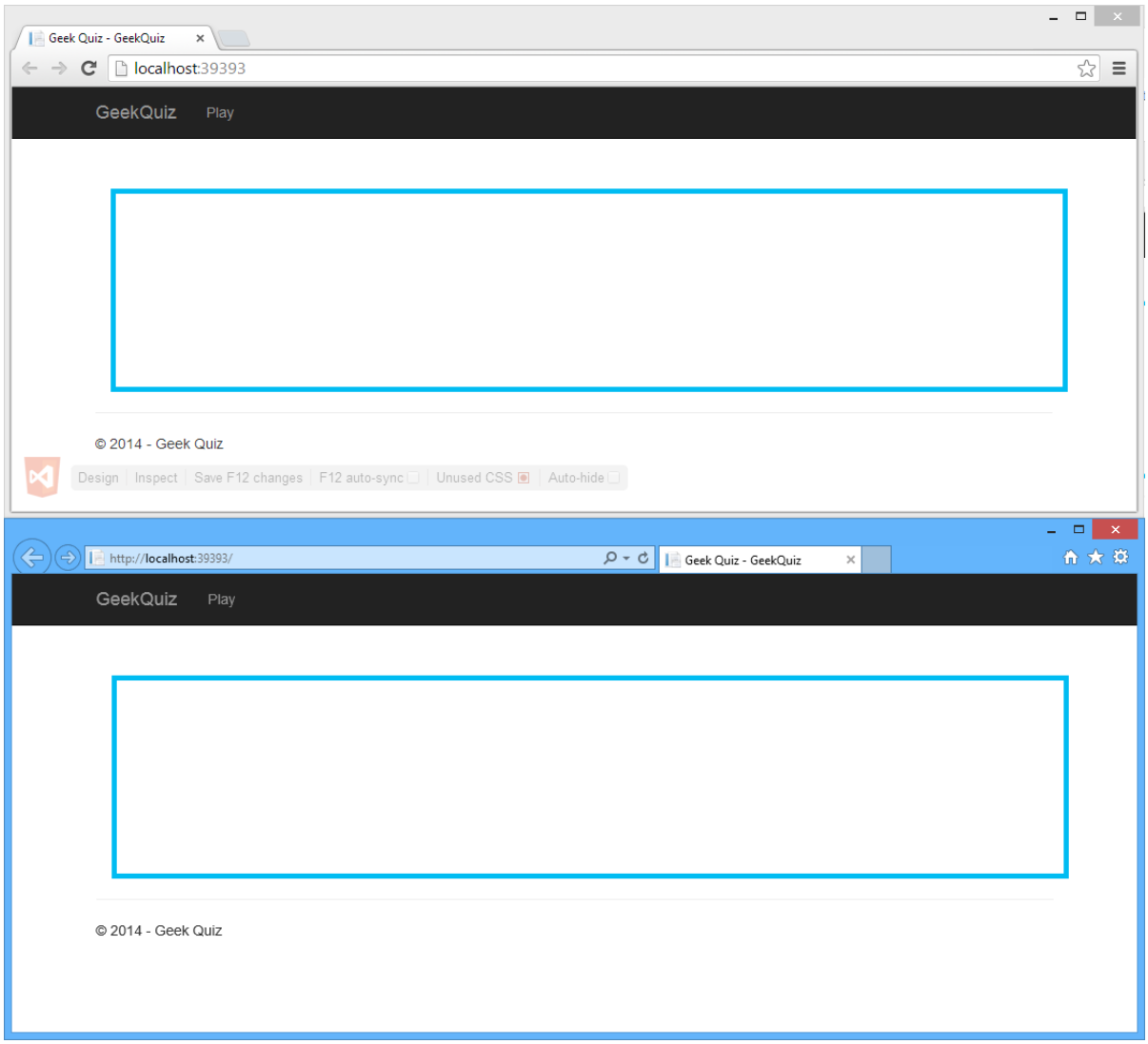
Google Chrome and Internet Explorer as default browsers

NOTE

After configuring the default browsers, the **Multiple Browsers** option is selected in the browser menu.



6. Press **CTRL** + **F5** to run the application without debugging.
7. When both browser windows open, place one of them above the other in order to see the updates on both browsers simultaneously. The browsers should display a web page with a light-blue rectangle.



Placing one browser above the other

8. Do not close the browsers. You will use them in the next task.

Task 2 - Using Zen Coding to Create HTML Elements

Zen Coding is an editor plugin for high-speed HTML, XML, XSL (or any other structured code format) coding and editing. The core of this plugin is a powerful abbreviation engine which allows you to expand expressions -similar to CSS selectors- into HTML code. Zen Coding is a fast way to write HTML using a CSS style selector syntax.

In this exercise, you will use the Zen Coding feature provided by Web Essentials to generate the HTML buttons that represent the options of the question.

1. Switch back to Visual Studio.
2. Open the **Index.cshtml** file located in the **Views | Home** folder.
3. Replace the **<!-- TODO: add options here-->** comment with the following code, and press **TAB**.

```
button.btn.btn-info.btn-lg.option{Answer $}*4
```

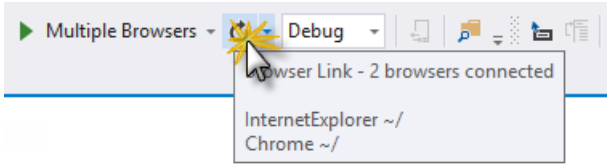
4. The code should be expanded to HTML.

```
<section id="content">
    <div class="container">
        <div class="row">
            <div class="flip-container text-center col-md-12">
                <div class="front">
                    <div class="row text-center">
                        <button class="btn btn-info btn-lg option">Answer 1</button><button class="btn btn-info btn-lg option">Answer 2<
                    </div>
                </div>
            </div>
        </div>
    </div>
</section>
```

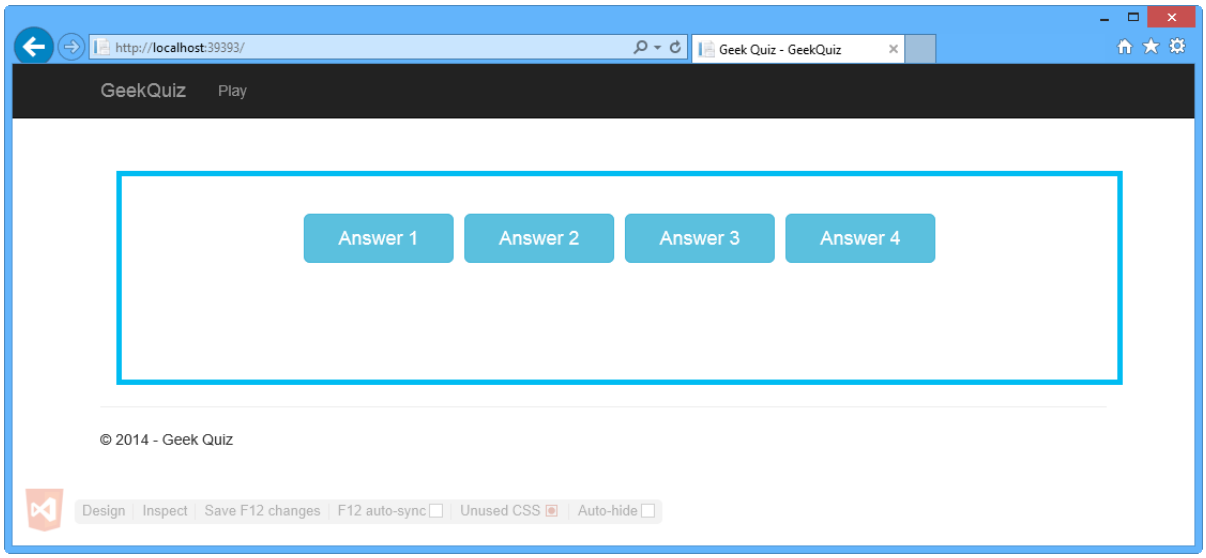
Expanded HTML

> **NOTE**
>
> To learn more about Zen Coding syntax, see the following article.

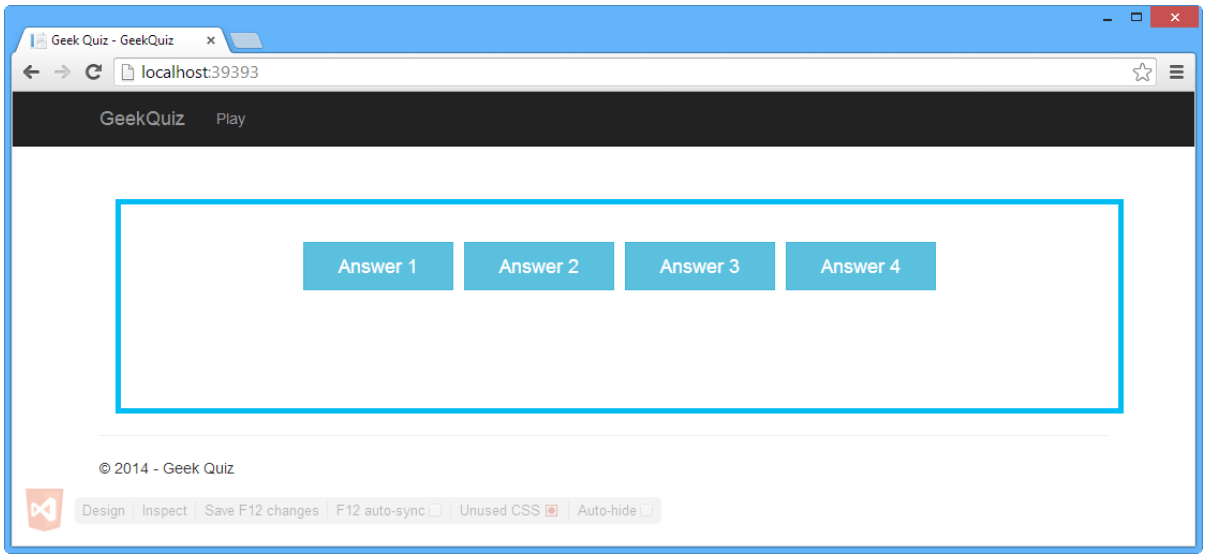
5. Click the **Refresh linked browsers** button to update both browsers.



Refresh linked browsers



Internet Explorer - Page updated with four buttons



Google Chrome - Page updated with four buttons

6. Switch back to Visual Studio.

7. You have added the buttons to the page, but you still need to add a template question. To do so, you will use a new feature in Web Essentials called **Lorem Ipsum generator**. Locate the **div** element with the **class** attribute **front**.

8. Add the following code as the first child element of the **div**, and press **TAB**.

```
p.lead>lorem5
```

9. The code should be expanded to HTML.

```
<section id="content">
    <div class="container">
        <div class="row">
            <div class="flip-container text-center col-md-12">
                <div class="front">
                    <p class="lead">
                        Lorem ipsum dolor sit amet.
                    </p>
                    <div class="row text-center">
                        <button class="btn btn-info btn-lg option">Answer 1</button><button class="btn
                    </div>
                </div>
            </div>
        </div>
    </div>
</section>
```

Lorem Ipsum autogenerated

> **NOTE**
>
> As part of Zen Coding, you can now generate Lorem Ipsum code directly in the HTML editor. Simply type **lorem** and hit **TAB** and a 30 word Lorem Ipsum text will be inserted. E.g. *lorem10* inserts 10 Lorem Ipsum words.

10. You will add a logo at the top of the question by using another new feature in Web Essentials called **Lorem Pixel generator**. Add the following code as the first child element of the **div** element with **container** as **class** value, and press **TAB**.

```
div.row.header>pix-436x185-abstract
```

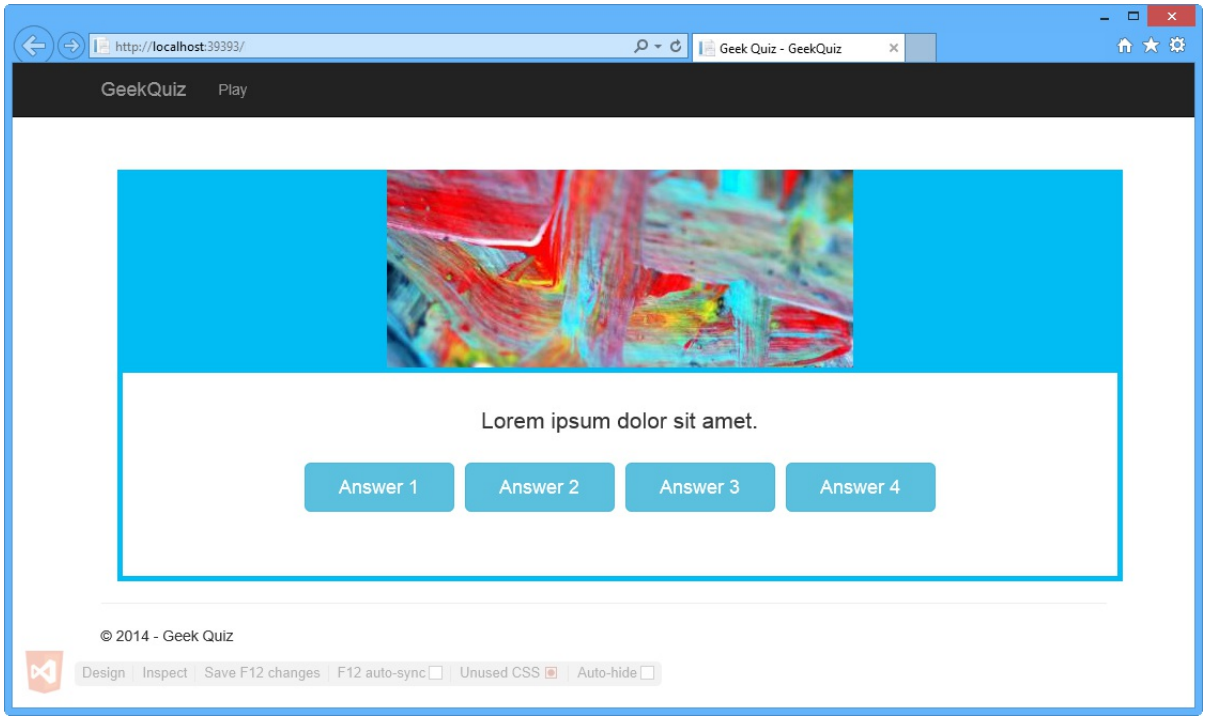
11. The code should expand to HTML.

```
<section id="content">
    <div class="container">
        <div class="row header"><img src="http://lorempixel.com/436/185/abstract/3/" alt="" /></div>
        <div class="row">
            <div class="flip-container text-center col-md-12">
                <div class="front">
                    <p class="lead">
                        Lorem ipsum dolor sit amet.
                    </p>
                    <div class="row text-center">
                        <button class="btn btn-info btn-lg option">Answer 1</button><button class="btn btn-info btn-lg option">Answer 2</button><bu
                    </div>
                </div>
            </div>
        </div>
    </div>
</section>
```

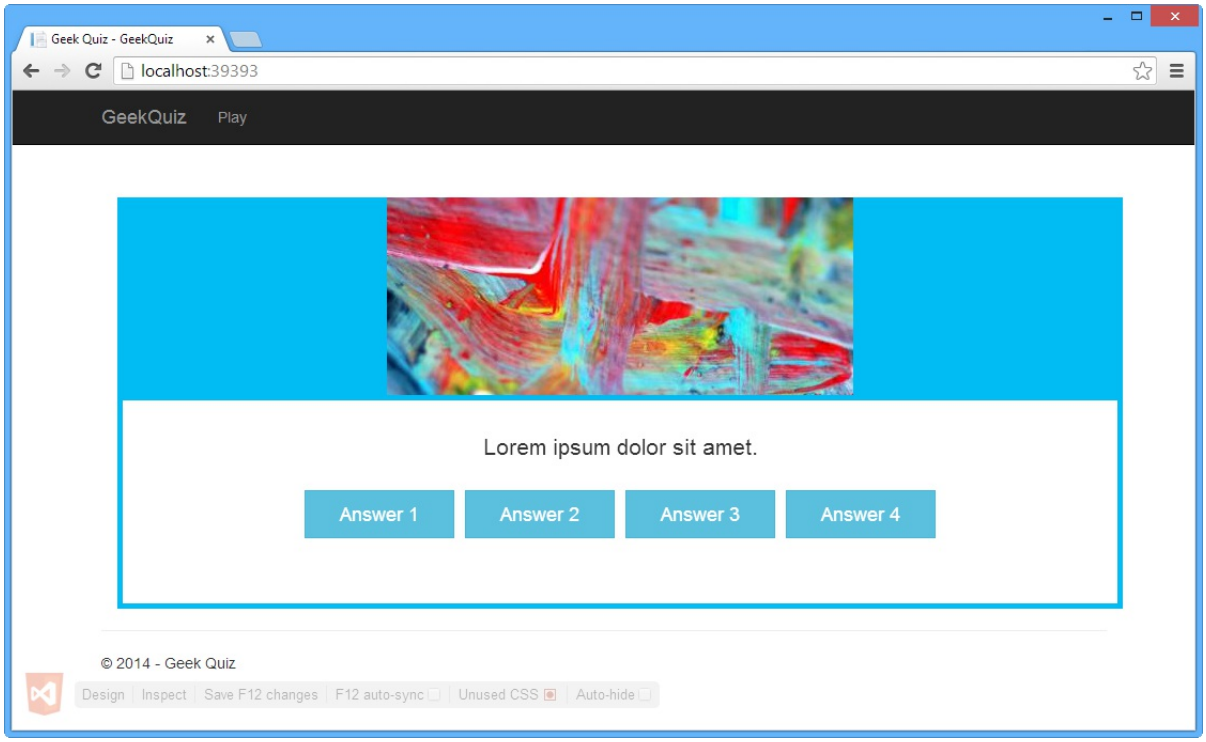
Lorem Pixel autogenerated

> **NOTE**
>
> As part of Zen Coding, you can also generate Lorem Pixel code directly in the HTML editor. Simply type **pix-200x200-animals** and hit **TAB** and an **img** tag with a 200x200 image of an animal will be inserted. For more information, refer to Lorem Pixel.

12. Click the **Refresh linked browsers** button to update both browsers.



Internet Explorer - Autogenerated image and text



Google Chrome - Autogenerated image and text

> **NOTE**
>
> Because the image is selected randomly when adding the code snippet, the image shown in the browsers may differ.

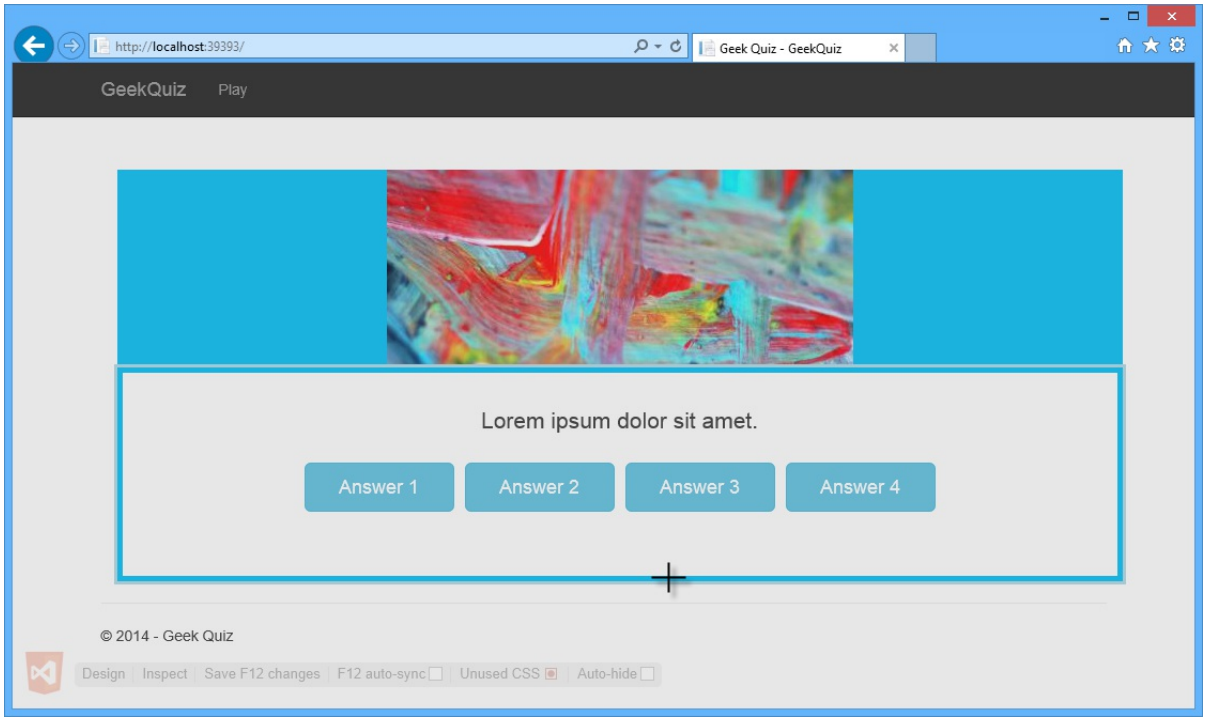
13. Do not close the browsers. You will use them in the next task.

Task 3 - Updating a Style Property

In this task, you will use the Browser Link's **Inspect Mode** feature to detect the exact location where the specific DOM element is generated and then update the color property of that element using a color picker provided by Web Essentials.

1. In the Internet Explorer browser, press **CTRL** + **ALT** + **I** to enable Inspect Mode.

2. Move the pointer over the light blue border and click.



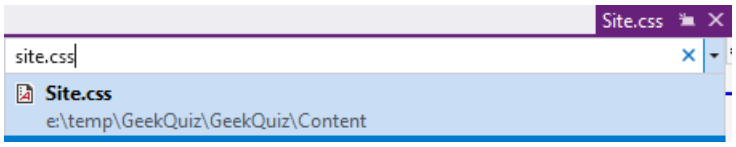
Moving the pointer over the light blue border

3. Switch back to Visual Studio. Notice how the HTML element that you selected in the browser is also selected in the Visual Studio HTML editor.

```
<section id="content">
    <div class="container">
        <div class="row header"><img src="http://lorempixel.com/436/185/abstract/1/" alt="" /></div>
        <div class="row">
            <div class="flip-container text-center col-md-12">
                <div class="front">
                    <p class="lead">
                        Lorem ipsum dolor sit amet.
                    </p>
                    <div class="row text-center">
                        <button class="btn btn-info btn-lg option">Answer 1</button><button class="btn btn-info btn-lg option">Answer 2</b
                    </div>
                </div>
            </div>
        </div>
    </div>
</section>
```

HTML element selected in the Visual Studio HTML editor

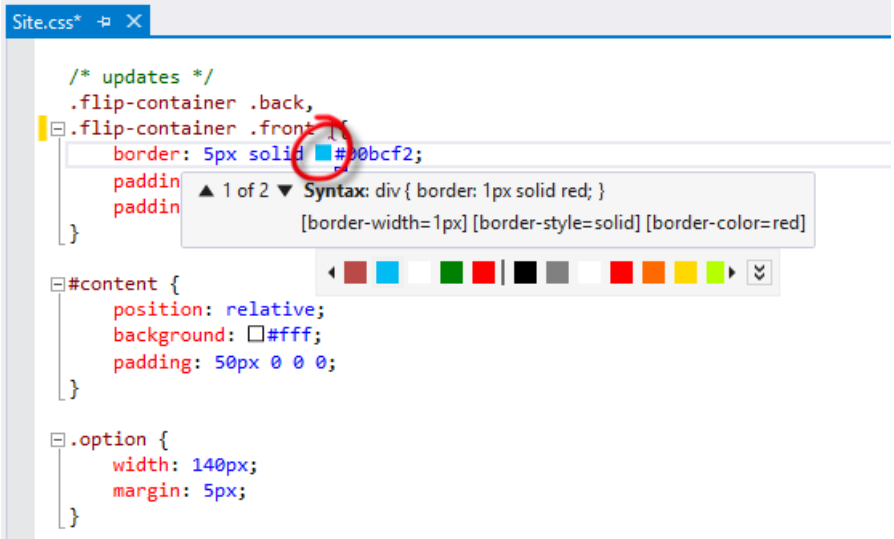
4. You will now update the **front** CSS class in order to change the styling of the selected element. To do so, press **CTRL** + **,** to open the **Navigate To** search box. Type **site.css** and press **ENTER** to open the file.



Opening file Site.css

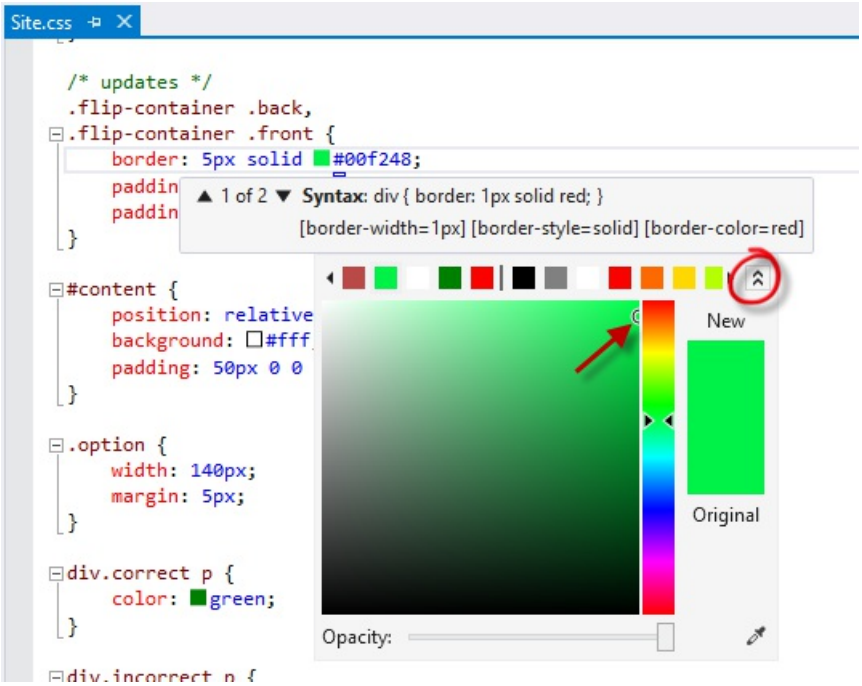
5. Press **CTRL** + **F** and type **.flip-container .front** to find the CSS selector.

6. Click the light blue square in the border property of the class to open the Color Picker.



Opening the Color Picker

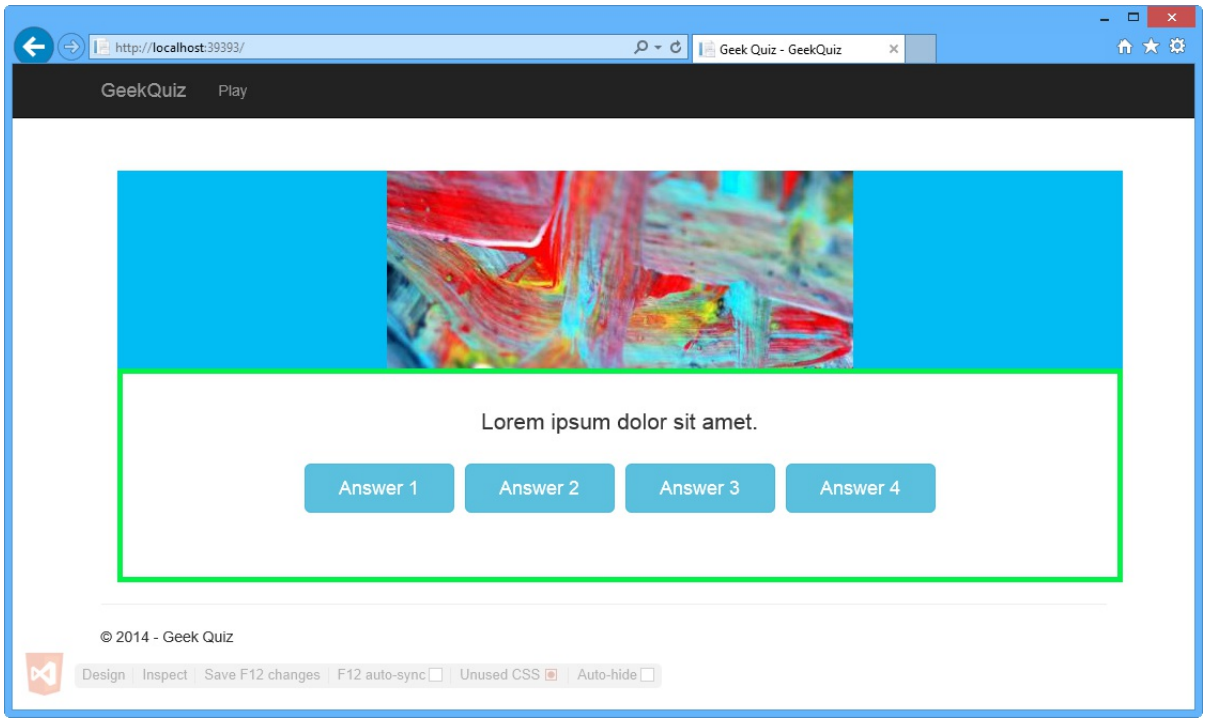
7. Expand the Color Picker by clicking the chevron button and select a new color.



Expanding the Color Picker

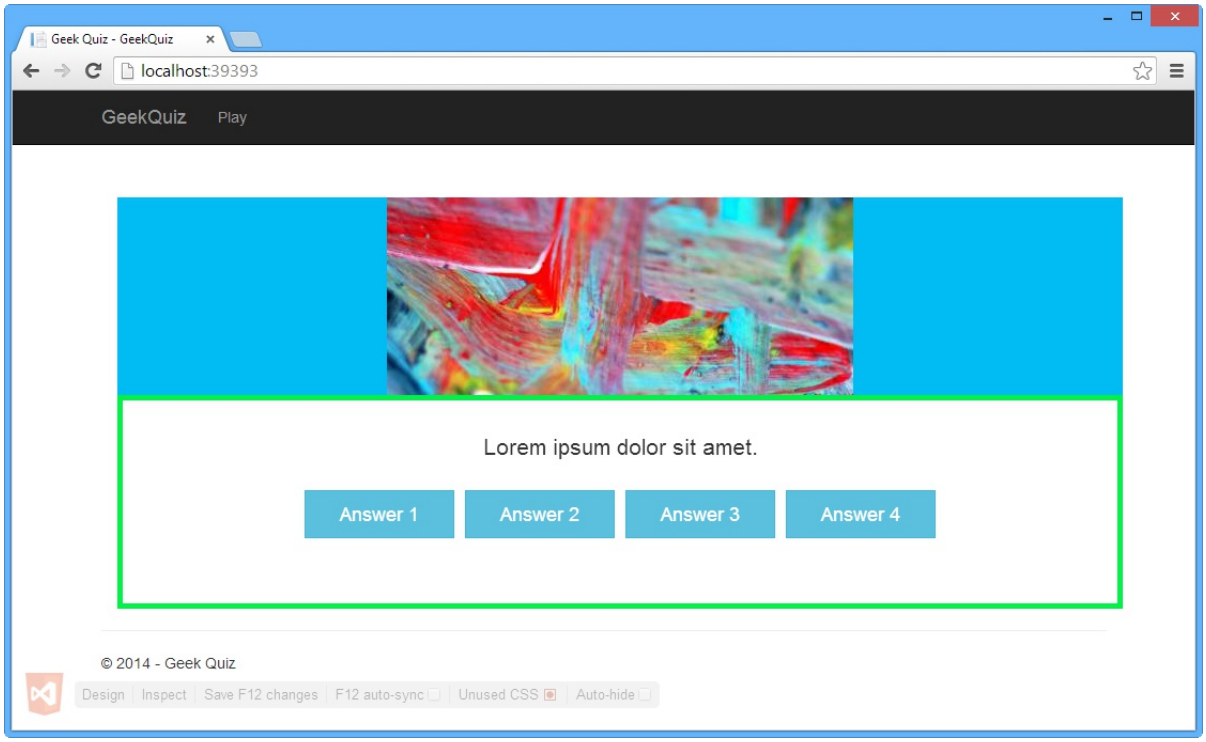
8. Press **CTRL** + **ALT** + **ENTER** to refresh linked browsers.

9. Switch to Internet Explorer and notice how the color of the border has changed.



Internet Explorer - Border color updated

10. Switch to Google Chrome and notice how the color of the border has changed.



Google Chrome - Border color updated

11. Switch back to Visual Studio.

12. Go to the end of the **Site.css** file and press **CTRL** + **F** to locate the **.btn** selector.

13. Notice that the **-webkit-border-radius** property is underlined in green.

```
.btn {
    -webkit-border-radius: 0px;
    word-wrap: break-word;
}

.header {
    background-color: #00bcf2;
    margin-left: 0;
    margin-right: -30px;
    text-align: center;
}
```

-webkit-border-radius property of the btn selector

14. Place the caret in the **-webkit-border-radius** property. A blue line should appear under the first letter of the first word of the property. This is the **smart tag**.

15. Press **CTRL** + **.** to open the suggestions menu and click **Add missing standard property (border-radius)**.

```
.btn {
    -webkit-border-radius: 0px;
    wrap: break-word;
}
    Add missing standard property (border-radius)
.header {
    background-color: #00bcf2;
    margin-left: 0;
    margin-right: -30px;
    text-align: center;
}
```

Add missing standard property suggestion

16. The **border-radius** property is automatically added to the CSS rule.

```
.btn {
    -webkit-border-radius: 0px;
    border-radius: 0px;
    word-wrap: break-word;
}

.header {
    background-color: #00bcf2;
    margin-left: 0;
    margin-right: -30px;
    text-align: center;
}
```

Missing standard property added

17. Move the pointer over the **border-radius** property to display the **Browser matrix tooltip**. The **Browser matrix tooltip** shows the availability of the property in each browser.

```
.btn {
    -webkit-border-radius: 0px;
    border-radius: 0px;
}
    Example: td { border-radius: 3px 4px; }
    4  9  10.5  5
.hea
    margin-left: 0;
    margin-right: -30px;
    text-align: center;
}
```

Browser matrix tooltip

18. Notice that the value of the **border-radius** property is still underlined. Move the pointer over the value to

see the warning message.

```
.btn {
    -webkit-border-radius: 0px;
    border-radius: 0px;
    word-wrap: brea
}

.header {
    background-color: #00bcf2;
    margin-left: 0;
    margin-right: -30px;
    text-align: center;
}
```

Best Practice: Don't specify the unit type (px) when the value is zero. (Disable this check in Tools -> Options)

Border-radius property value warning

19. Remove the unit of the **border-radius** property value as suggested by the tooltip.

20. As **border-radius** is the standard property for defining how rounded border corners are, you can remove the **-webkit-border-radius** property and value from the CSS rule.

21. Place the caret in the **word-wrap** property and notice that the smart tag also appears below.

22. Open the menu and click **Add missing vendor specifics**.

```
.btn {
    border-radius: 0;
    word-wrap: break-word;
}

.hea
    background-color: #00bcf2;
    margin-left: 0;
    margin-right: -30px;
    text-align: center;
}
```

Add missing vendor specifics

Add missing vendor specifics suggestion

23. The **-ms-word-wrap** property is automatically added to the CSS rule.

```
.btn {
    border-radius: 0;
    -ms-word-wrap: break-word;
    word-wrap: break-word;
}

.header {
    background-color: #00bcf2;
    margin-left: 0;
    margin-right: -30px;
    text-align: center;
}
```

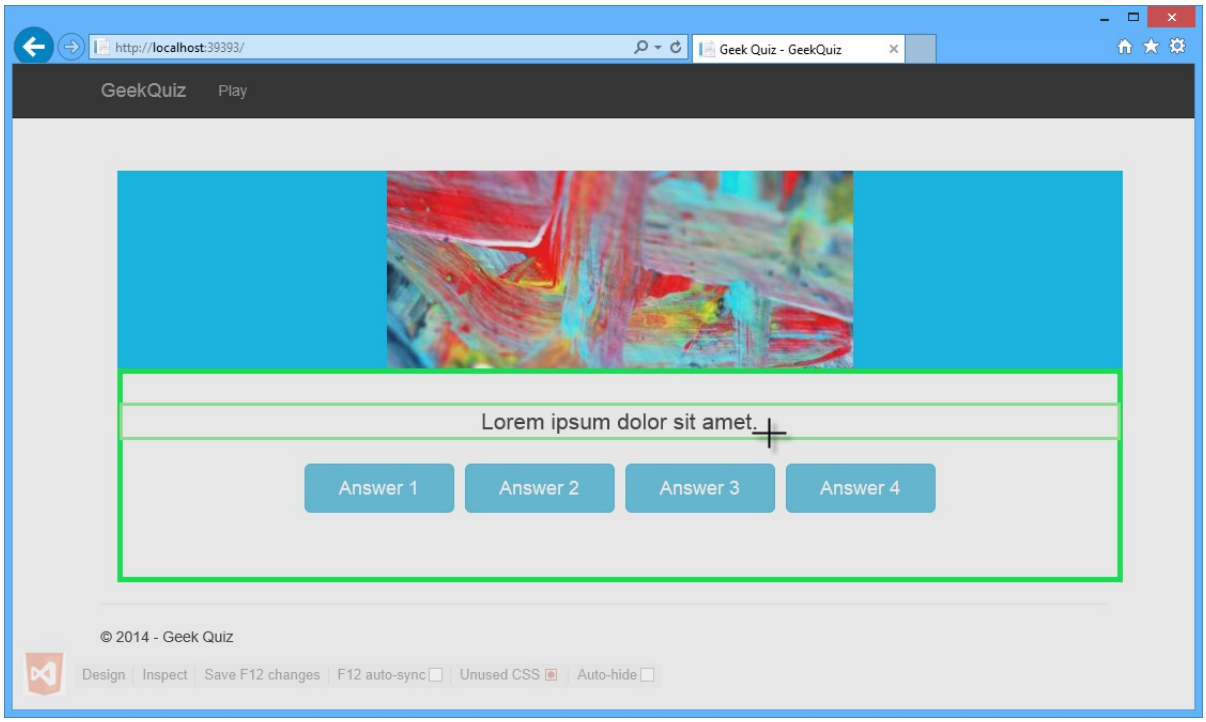
Vendor specific property added

Task 4 - Updating the HTML Code from the Browser

In this task, you will use the Browser Link's **Design Mode** feature to edit the DOM object from the browser and transfer the changes to the HTML source file in Visual Studio.

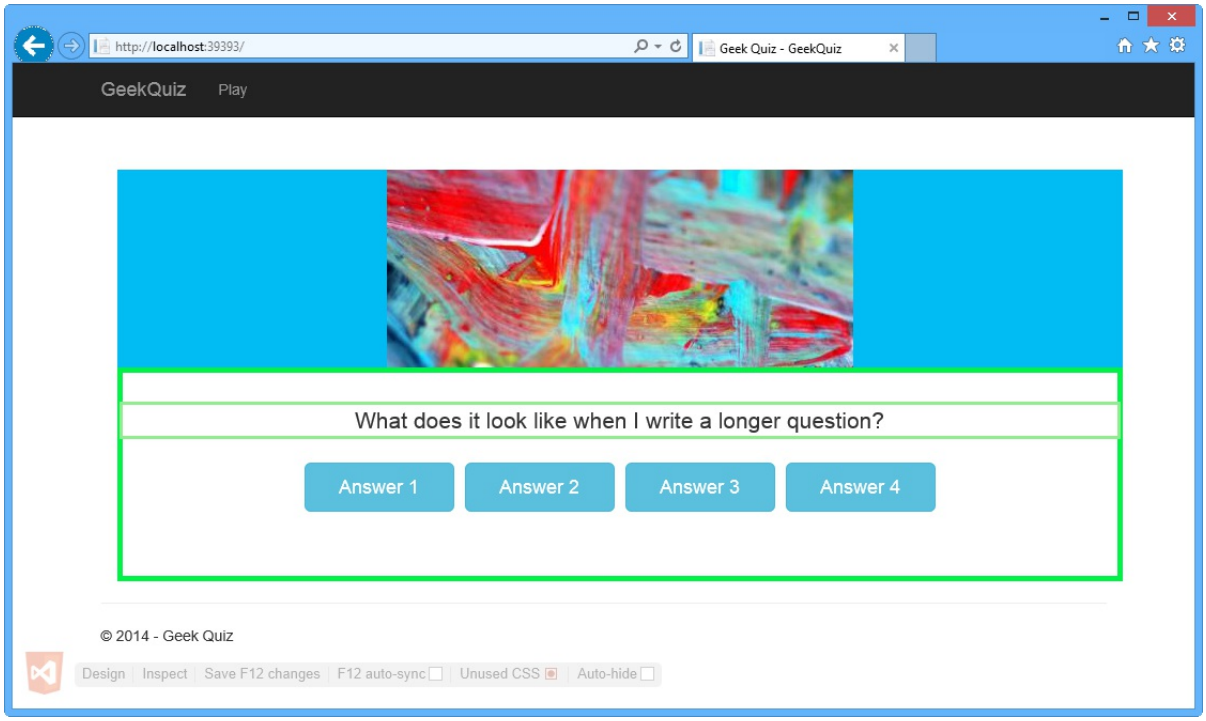
1. In Google Chrome, press **CTRL** + **ALT** + **D** to enable Design Mode.

2. Move the pointer over the **Lorem Ipsum dolor sit amet** label and click.



Editing the question

3. A cursor should appear. Replace the original text with *What does it look like when I write a longer question?*, and then press **ESC** to exit Design Mode.



Question edited

4. Switch back to Visual Studio and open **Index.cshtml**, if not already opened. Notice that the inner text of the **<p>** element has been updated.

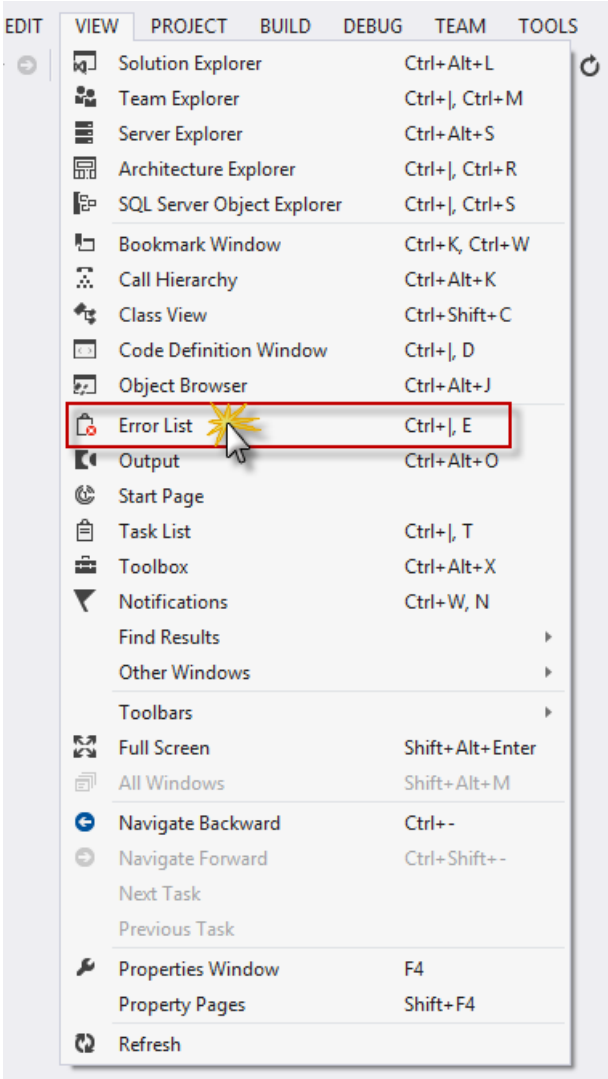
```
<section id="content">
    <div class="container">
        <div class="row header"><img src="http://lorempixel.com/436/185/abstract/1/" alt="" /></div>
        <div class="row">
            <div class="flip-container text-center col-md-12">
                <div class="front">
                    <p class="lead">What does it look like when I write a longer question?</p>
                    <div class="row text-center">
                        <button class="btn btn-info btn-lg option">Answer 1</button><button class="btn btn-infc
                    </div>
                </div>
            </div>
        </div>
    </div>
</section>
```

Updated question in the HTML page

Task 5 - Reviewing SEO Related Warnings

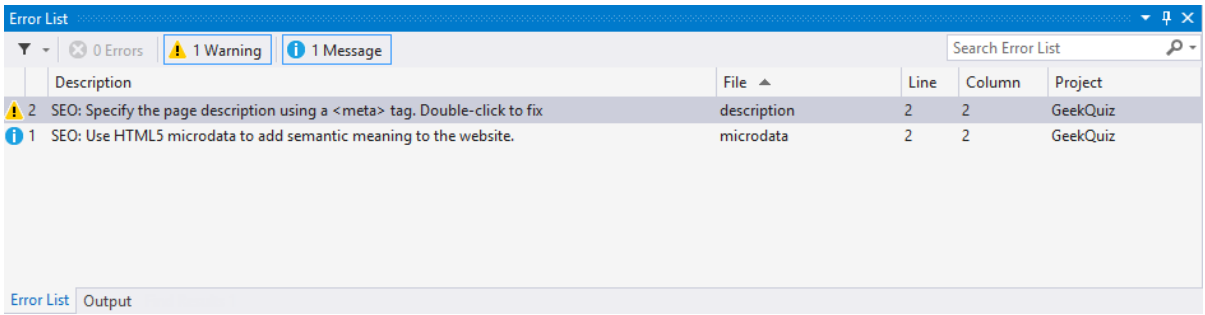
Search Engine Optimization (SEO) is the process of making a website rank higher on a search engine's list of results. The higher the site ranks and the more consistently it is listed, the more visitors the site will get from that search engine. Web Essentials incorporates an analytical tool that examines HTML, reports the issues found and provides assistance to fix them.

1. Go to the **View** menu and click **Error List** to open the **Error List** window.



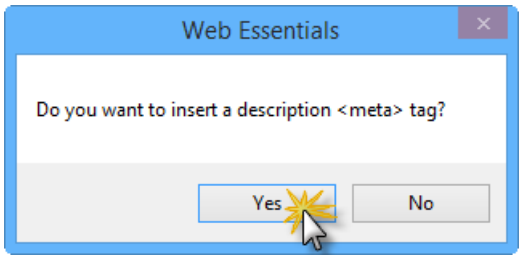
Error List in View menu

2. Notice that there is an SEO warning notifying that a **<meta>** tag for the page description is missing. Double-click the SEO warning entry to fix it.



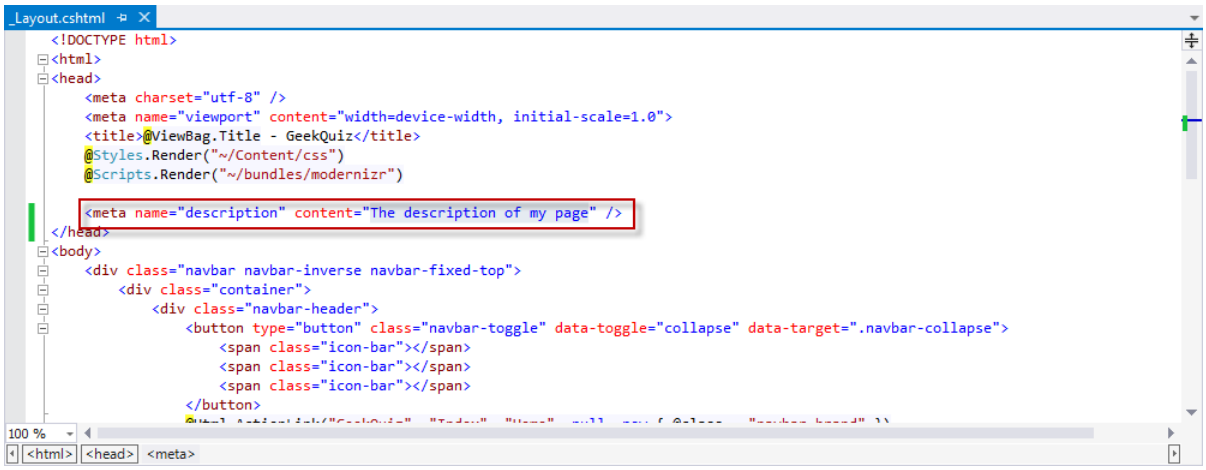
Error List window

3. In the **Web Essentials** dialog box, click **Yes** to insert a description <meta> tag.



Web Essentials dialog box

4. The editor for **_Layout.cshtml** opens and the **<meta>** tag is automatically added to the **head** section of the HTML file.



Meta tag automatically added to _Layout page

5. Change the value of the **content** attribute to *GeekQuiz* and save the file.

Exercise 2: Taking Advantage of Code Snippets and IntelliSense

With Web Essentials, the HTML editor has been extended with extra functionality. In this exercise, you will see some new features that are helpful when developing web applications.

Task 1 - Using IntelliSense in HTML Documents

The first new feature you will see in this task is called **Dynamic IntelliSense**. Dynamic IntelliSense reads other tags and attributes to infer the possible ids you will use.

In this task, you will create a new HTML form element which contains a label and an input field. Then you will add a **for** attribute to the label to bind it to the input, and you will see IntelliSense suggestions based on the ids of the inputs in scope.

1. Open **Visual Studio Express 2013 for Web** and the **Begin.sln** solution located in the **Source/Ex2-TakingAdvantageofCodeSnippetsandIntelliSense/Begin** folder. Alternatively, you can continue with the solution that you obtained in the previous exercise.

2. In **Solution Explorer**, open the **Index.cshtml** file located in the **Views | Home** folder.

3. Add the following form inside the **<section>** element.

   (Code Snippet - *VisualStudio2013WebTooling - Ex2 - Form*)

```
<form>
     <input type="text" id="name" />
</form>
```

4. The input tag should be preceded by a label with some description of the field. Add the following label before the input tag.

```
<form>
    <label id="name">Name</label>
    <input type="text" id="name" />
</form>
```

5. The **for** attribute of a **<label>** specifies which form element a label is bound to. The attribute's value should be equal to the id of the related element. Add the **for** attribute to the **<label>** element. As shown in the following figure, the "name" value pops up in the IntelliSense box, based on the id of the elements within the same scope (the enclosing **<form>**).

   Showing the id in IntelliSense

6. Delete the recently added **<form>** element and its content.

Task 2 - Using HTML Code Snippets

HTML5 introduced more than 25 new semantic tags. Visual Studio already had IntelliSense support for these tags, but Visual Studio 2013 makes it faster and easier to write markup by adding new code snippets. Though these tags are not complicated, they come with a few small subtleties, such as adding the correct codec fallbacks for the *audio* tag. In this task, you will see the HTML code snippets for the audio tag.

1. In the **Index.cshtml** file, type **<aud** inside the **<section>** element as shown in the following figure.

```
<section id="content">
    <div class="container">
        <div class="row header"><img src="http://lorempixel.com/436/185/abstract/1/" alt="" /></div>
        <div class="row">
            <div class="flip-container text-center col-md-12">
                <div class="front">
                    <p class="lead">What does it look like when I write a longer question?</p>
                    <div class="row text-center">
                        <button class="btn btn-info btn-lg option">Answer 1</button><button class="btn btn-info btn-lg option">A
                    </div>
                </div>
            </div>
        </div>
    </div>
    <aud
</sec audio
```

Inserting an audio element

2. Press **TAB** twice and notice how the following code is added on the page and the cursor is placed on the **src** attribute of the first source.

```
<audio controls="controls">
     <source src="file.mp3" type="audio/mp3" />
     <source src="file.ogg" type="audio/ogg" />
</audio>
```

> **NOTE**
>
> By pressing the **TAB** key twice, the code snippet is inserted. The audio snippet shows the standard usage of the *audio* tag, with two source files for improved support.

3. Delete the second line and update the source of the first line with the following link to the WebCampsTV Katana show: http://media.ch9.ms/ch9/11d8/604b8163-fad3-4f12-9607-b404201211d8/KatanaProject.mp3. The resulting code is shown below.

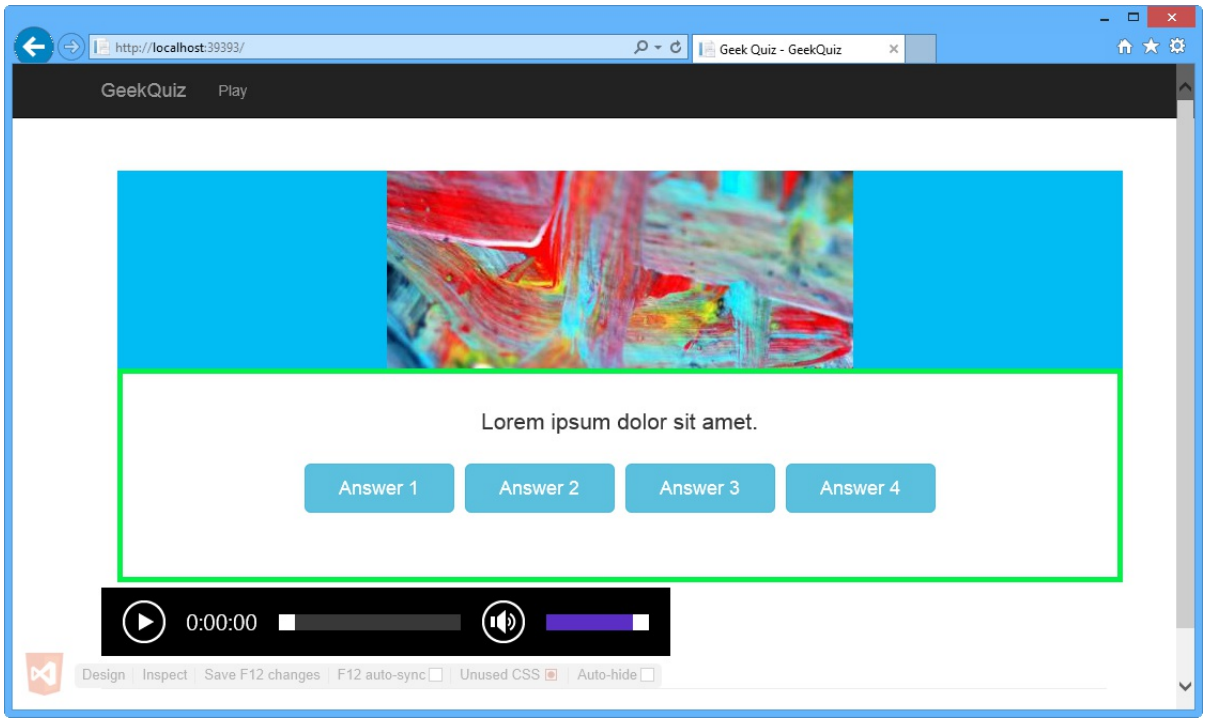
```
<audio controls="controls">
     <source src="http://media.ch9.ms/ch9/11d8/604b8163-fad3-4f12-9607-b404201211d8/KatanaProject.mp3"
type="audio/mp3" />
</audio>
```

> **NOTE**
>
> The source file *KatanaProject.mp3* is used as an example. You can use another source if you prefer.

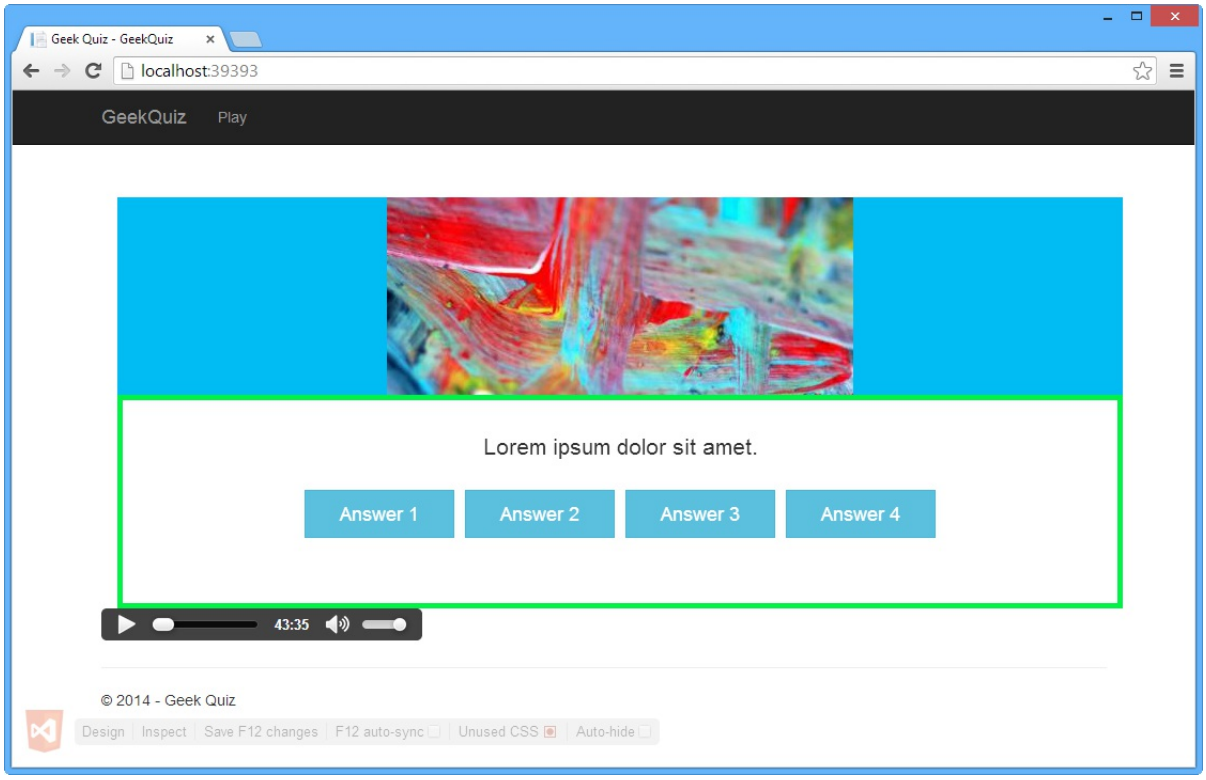
4. Press **CTRL** + **S** to save the file.

5. Press **CTRL** + **F5** to start the application.

6. Notice that an audio player was added to the application.



Audio player in Internet Explorer



Audio player in Google Chrome

7. Do not close the browsers. You will use them in the next task.

Task 3 - Using IntelliSense in JavaScript Documents

With Web Essentials 2013, style sheets and HTML pages produce a list of IDs and class names. In this task, you will learn how those lists improve JavaScript IntelliSense support in Web Essentials 2013.

1. In the **Index.cshtml** file, add the following code to define a **script** tag for JavaScript code.

```
...
</section>
@section scripts{
    <script type="text/javascript">

    </script>
}
```

2. Add the following code inside the **script** tag to define the ready callback function.

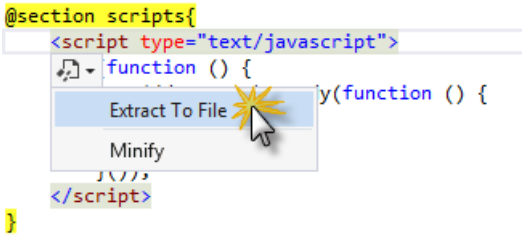
   (Code Snippet - *VisualStudio2013WebTooling - Ex2 - ReadyFunction*)

```
(function () {
    $(document).ready(function () {

    });
}());
```

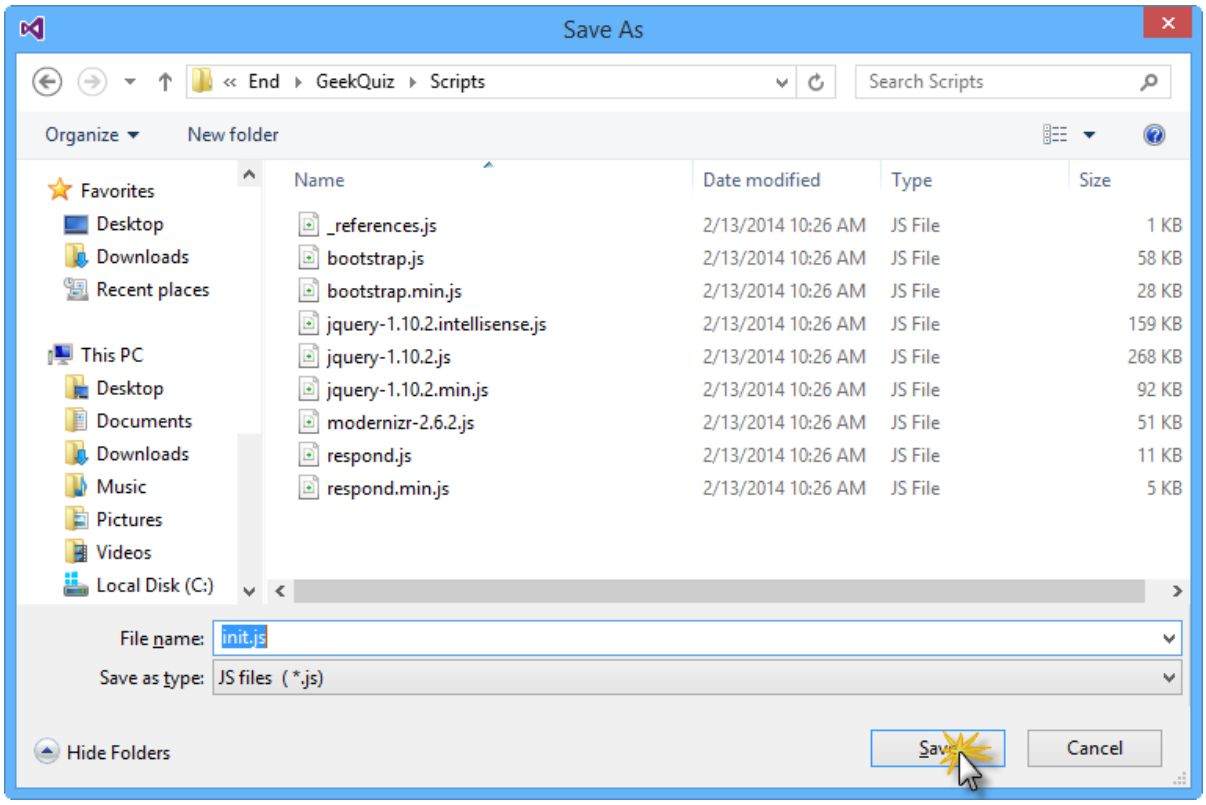
3. Place the caret in the **script** tag and press **CTRL** + . to open the suggestion menu.

4. Click **Extract To File**.



JavaScript extract to file suggestion

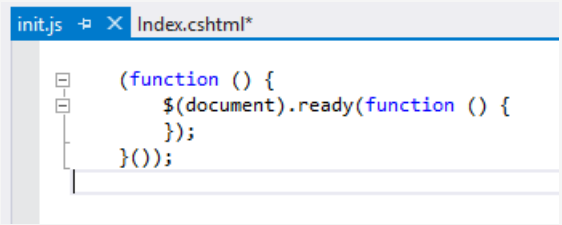
5. In the **Save As** window, select the **Scripts** folder, name the file **init.js** and click **Save**.



Save As window

NOTE

The **init.js** file is created and the content of the script is moved to the file.



Init.js file created with the content included

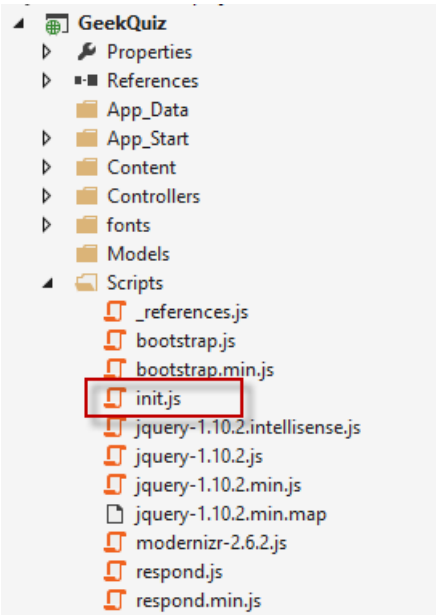
6. Open the **Index.cshtml** file and check that the script tag was replaced with a reference to the **init.js** file.

```
    </audio>
</section>

@section scripts{
    <script src="/Scripts/init.js"></script>
}
```

Init.js html reference

7. Go to the **Solution Explorer** and notice that the **init.js** file was included automatically in the solution.



Init.js file included in solution

8. Switch back to the **init.js** file to update the **ready** function callback.

9. Inside the function callback definition that is passed to *ready*, add the following code to get all the elements by a specific class attribute.

```
(function () {
    $(document).ready(function () {
        document.getElementsByClassName("")
    });
}());
```

10. Press **CTRL** + **Space** between the quotes inside the **getElementsByClassName** function call.

```
1 (function () {
2     $(document).ready(function () {
3         document.getElementsByClassName("")
4     });
5 }());
```

NodeList getElementsByClassName(**String classNames**)

"active"
"affix"
"alert"
"alert-danger"
"alert-dismissable"
"alert-info"
"alert-link"
"alert-success"
"alert-warning"

Showing IntelliSense for the getElementsByClassName function

> **NOTE**
>
> Notice that IntelliSense shows the classes defined in the project style sheets.

11. Replace the line that you have created with the following code.

```
(function () {
    $(document).ready(function () {
        var audioElements = document.getElementsByTagName("au");
    });
}());
```

12. Position the cursor after **au** inside the quotes in the **getElementsByTagName** function and press **CTRL** + **Space**.

```
(function () {
    $(document).ready(function () {
        var audioElements = document.getElementsByTagName("au");
    });
}());
```

"animateTransform"
"area"
"article"
"aside"
"audio"
"b"
"bdi"
"bdo"
"blockquote"

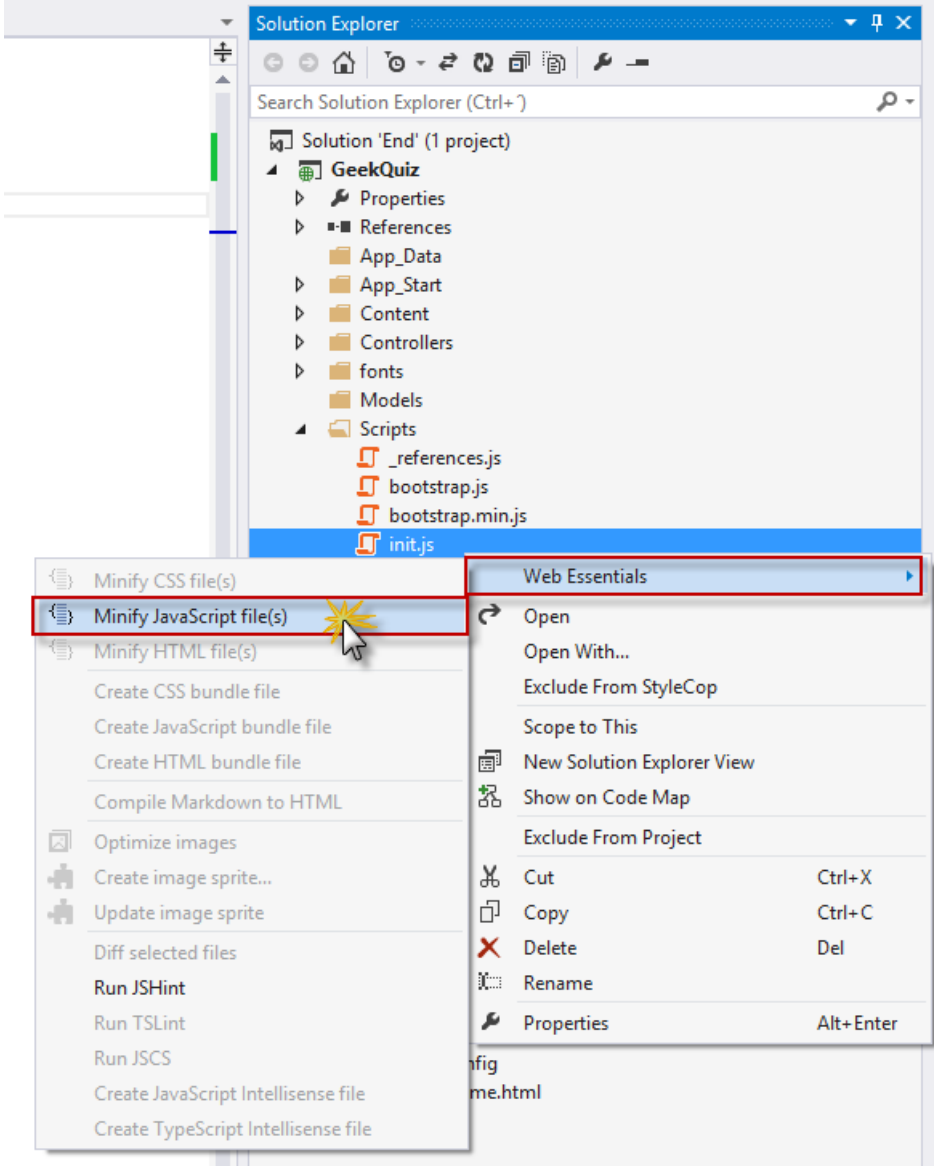
Showing IntelliSense for the getElementsByTagName method

13. Select **"audio"** from the list and press **ENTER**. The result is shown in the following figure.

```
1 (function () {
2     $(document).ready(function () {
3         var audioElements = document.getElementsByTagName("audio");
4     });
5 }());
```

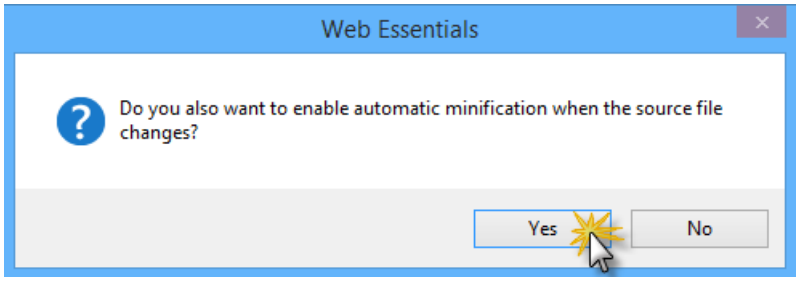
Retrieving Audio Elements

14. In **Solution Explorer**, right-click the **init.js** file in the **Scripts** folder and select **Minify JavaScript file(s)** from the **Web Essentials** menu.



Minify JavaScript file(s)

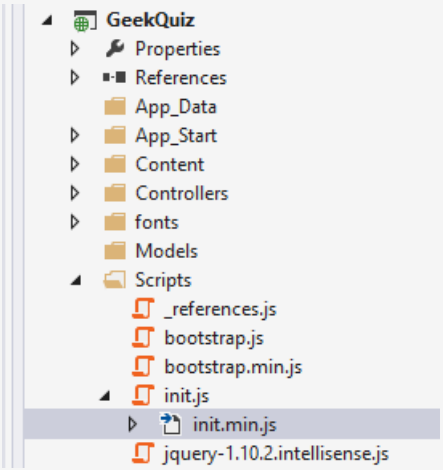
15. When prompted to enable automatic minification when the source file changes click **Yes**.



Enabling automatic minification warning

NOTE

The **init.min.js** is created and is added as a dependency of the **init.js** file.



Init.min.js file created

16. Open the **init.min.js** file and notice that the file is minified.

```
init.min.js   init.js   Index.cshtml
(function(){$(document).ready(function(){var n=document.getElementsByTagName("audio")})})();
/*
//# sourceMappingURL=init.min.js.map
*/
```

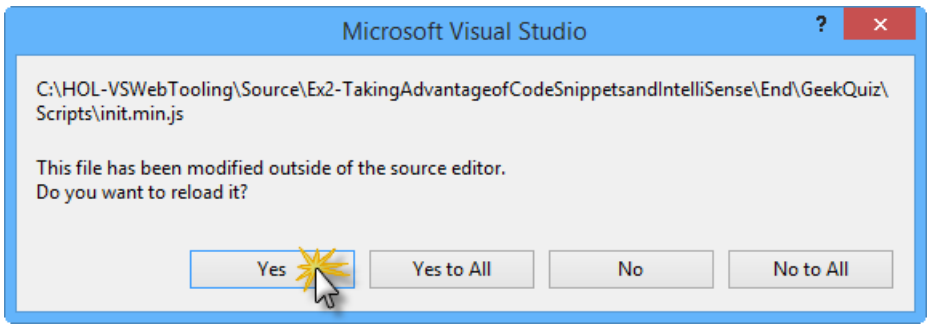
Init.min.js file content

17. In the **init.js** file, add the following code below the **getElementsByTagName** function call to play all the audio elements.

(Code Snippet - *VisualStudio2013WebTooling - Ex2 - PlayAudioElements*)

```
var len = audioElements.length;
for (var i = 0; i < len; i++) {
    audioElements[i].play();
}
```

18. Click **CTRL** + **S** to save the file. Since the minified file is already opened, you will see a dialog box saying that the file was modified outside of the source editor. Click **Yes**.



Microsoft Visual Studio warning

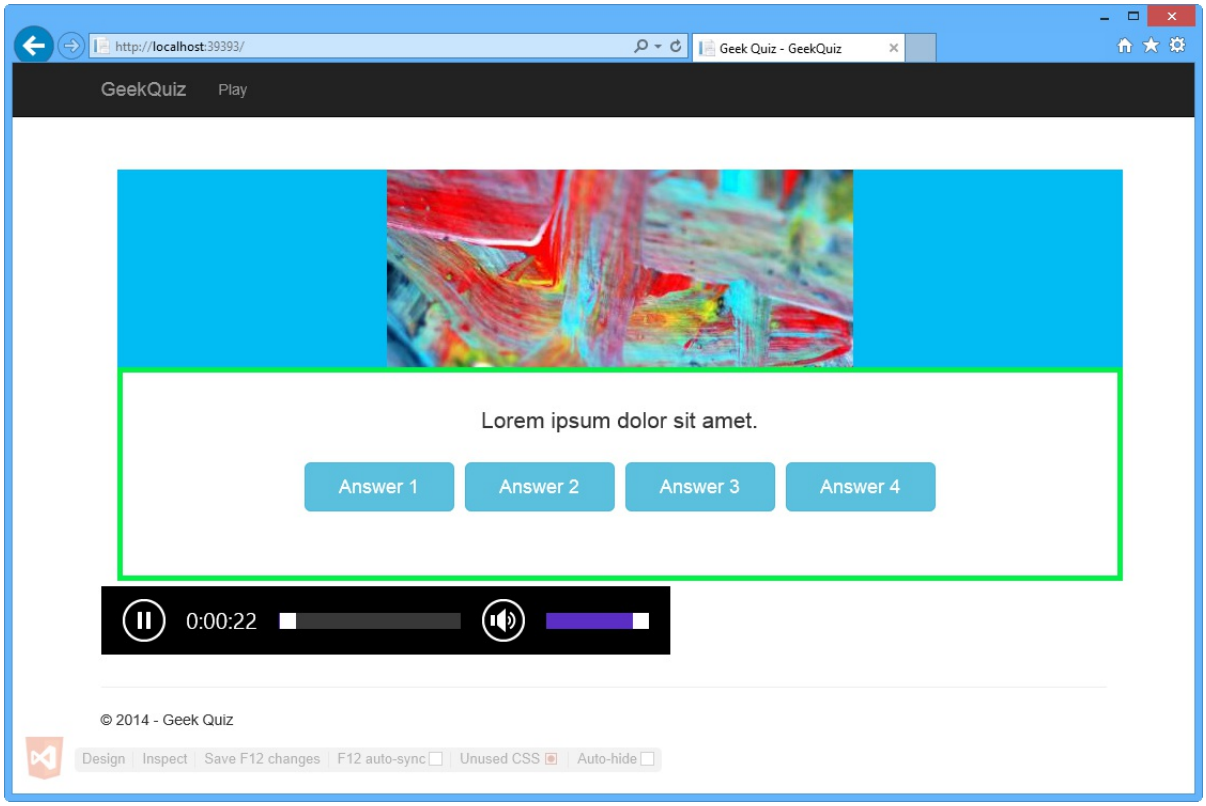
19. Switch back to the **init.min.js** file to verify that the file was updated with the new code.

```
init.min.js   init.js   Index.cshtml
(function(){$(document).ready(function(){for(var t=document.getElementsByTagName("audio"),i=t.length,n=0;n<i;n++)t[n].play()})})();
/*
//# sourceMappingURL=init.min.js.map
*/
```

Init.min.js file updated

20. Click the **Browser Link Refresh** button.

21. Once both browsers are refreshed the audio players you saw in the previous task will start playing automatically.



Audio player included in view

Summary

By completing this hands-on lab you have learned how to:

- Use new HTML editor features included in Web Essentials such as rich HTML5 code snippets and Zen coding
- Use new CSS editor features included in Web Essentials such as the Color picker and Browser matrix tooltip
- Use new JavaScript editor features included in Web Essentials such as Extract to File and IntelliSense for all HTML elements
- Exchange data between your browser and Visual Studio using Browser Link

Hands On Lab: One ASP.NET: Integrating ASP.NET Web Forms, MVC and Web API

2/20/2020 • 18 minutes to read • Edit Online

by Web Camps Team

Download Web Camps Training Kit

> ASP.NET is a framework for building Web sites, apps and services using specialized technologies such as MVC, Web API and others. With the expansion ASP.NET has seen since its creation and the expressed need to have these technologies integrated, there have been recent efforts in working towards **One ASP.NET**.
>
> Visual Studio 2013 introduces a new unified project system which lets you build an application and use all the ASP.NET technologies in one project. This feature eliminates the need to pick one technology at the start of a project and stick with it, and instead encourages the use of multiple ASP.NET frameworks within one project.
>
> All sample code and snippets are included in the Web Camps Training Kit, available at https://aka.ms/webcamps-training-kit.

Overview

Objectives

In this hands-on lab, you will learn how to:

- Create a Web site based on the **One ASP.NET** project type
- Use different **ASP.NET** frameworks like **MVC** and **Web API** in the same project
- Identify the main components of an **ASP.NET** application
- Take advantage of the **ASP.NET Scaffolding** framework to automatically create Controllers and Views to perform CRUD operations based on your model classes
- Expose the same set of information in machine- and human-readable formats using the right tool for each job

Prerequisites

The following is required to complete this hands-on lab:

- Visual Studio Express 2013 for Web or greater
- Visual Studio 2013 Update 1

Setup

In order to run the exercises in this hands-on lab, you will need to set up your environment first.

1. Open Windows Explorer and browse to the lab's **Source** folder.
2. Right-click on **Setup.cmd** and select **Run as administrator** to launch the setup process that will configure your environment and install the Visual Studio code snippets for this lab.
3. If the User Account Control dialog box is shown, confirm the action to proceed.

> **NOTE**
>
> Make sure you have checked all the dependencies for this lab before running the setup.

Using the Code Snippets

Throughout the lab document, you will be instructed to insert code blocks. For your convenience, most of this code is provided as Visual Studio Code Snippets, which you can access from within Visual Studio 2013 to avoid having to add it manually.

> **NOTE**
>
> Each exercise is accompanied by a starting solution located in the **Begin** folder of the exercise that allows you to follow each exercise independently of the others. Please be aware that the code snippets that are added during an exercise are missing from these starting solutions and may not work until you have completed the exercise. Inside the source code for an exercise, you will also find an **End** folder containing a Visual Studio solution with the code that results from completing the steps in the corresponding exercise. You can use these solutions as guidance if you need additional help as you work through this hands-on lab.

Exercises

This hands-on lab includes the following exercises:

1. Creating a New Web Forms Project
2. Creating an MVC Controller Using Scaffolding
3. Creating a Web API Controller Using Scaffolding

Estimated time to complete this lab: **60 minutes**

> **NOTE**
>
> When you first start Visual Studio, you must select one of the predefined settings collections. Each predefined collection is designed to match a particular development style and determines window layouts, editor behavior, IntelliSense code snippets, and dialog box options. The procedures in this lab describe the actions necessary to accomplish a given task in Visual Studio when using the **General Development Settings** collection. If you choose a different settings collection for your development environment, there may be differences in the steps that you should take into account.

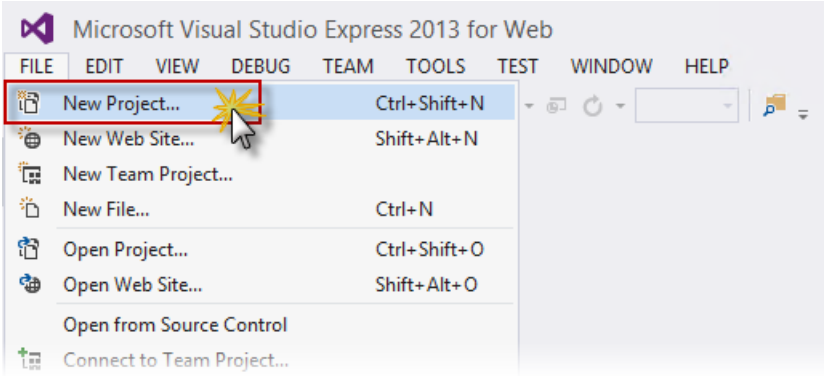
Exercise 1: Creating a New Web Forms Project

In this exercise you will create a new Web Forms site in Visual Studio 2013 using the **One ASP.NET** unified project experience, which will allow you to easily integrate Web Forms, MVC and Web API components in the same application. You will then explore the generated solution and identify its parts, and finally you will see the Web site in action.

Task 1 – Creating a New Site Using the One ASP.NET Experience

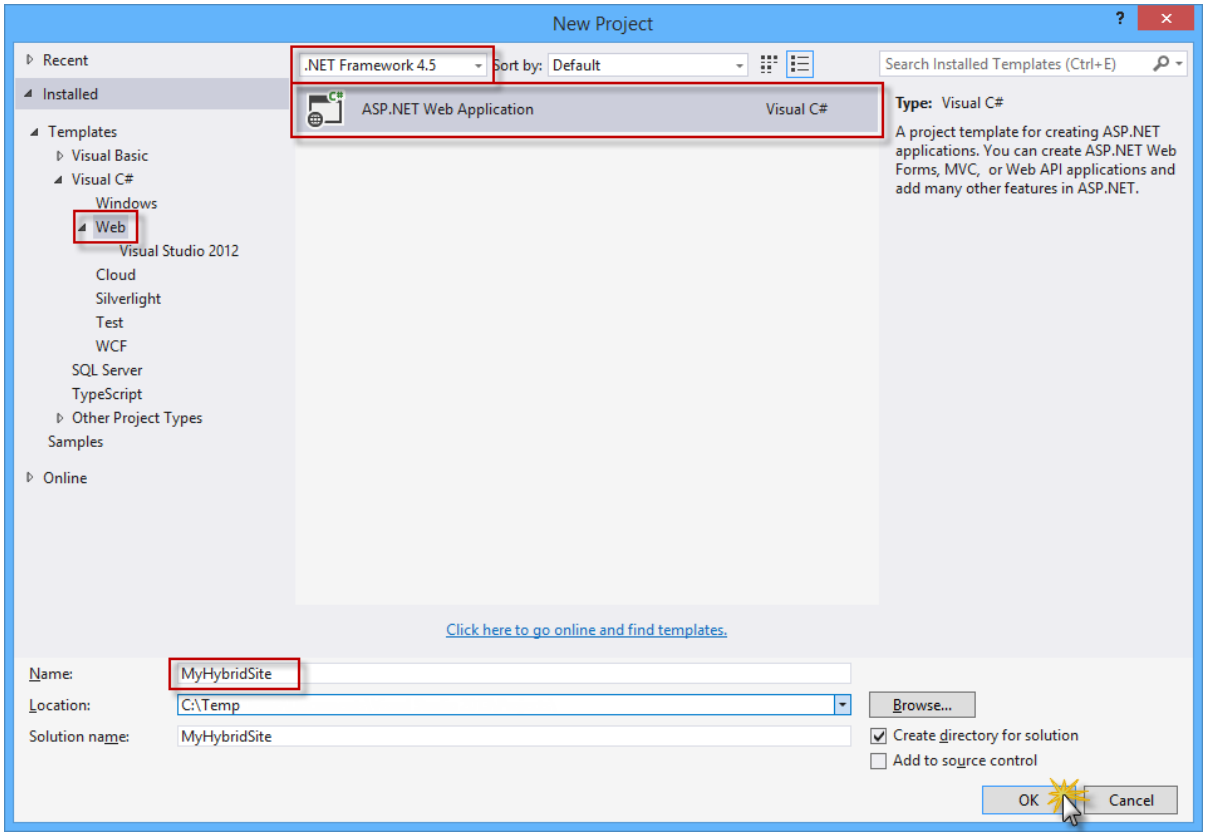
In this task you will start creating a new Web site in Visual Studio based on the **One ASP.NET** project type. **One ASP.NET** unifies all ASP.NET technologies and gives you the option to mix and match them as desired. You will then recognize the different components of Web Forms, MVC and Web API that live side by side within your application.

1. Open **Visual Studio Express 2013 for Web** and select **File | New Project...** to start a new solution.



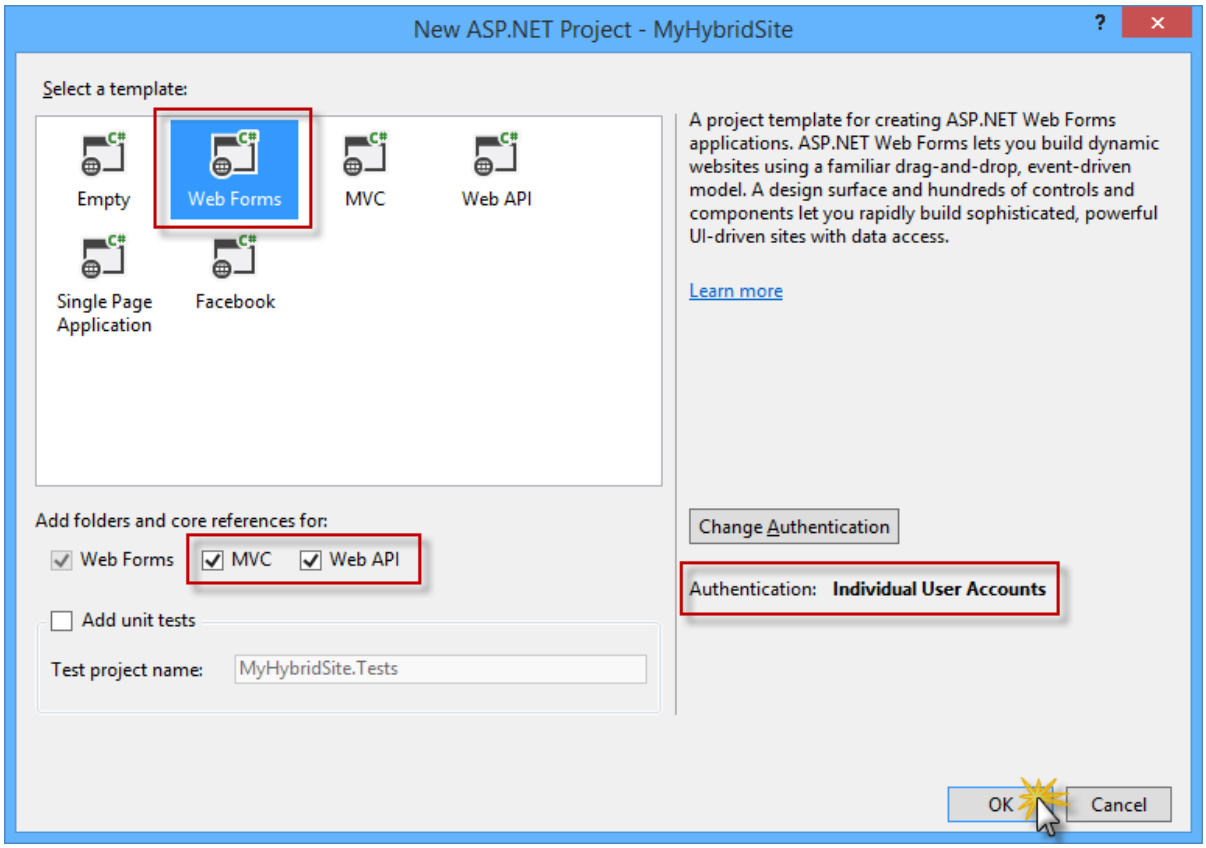
Creating a New Project

2. In the **New Project** dialog box, select **ASP.NET Web Application** under the **Visual C# | Web** tab, and make sure **.NET Framework 4.5** is selected. Name the project *MyHybridSite*, choose a **Location** and click **OK**.



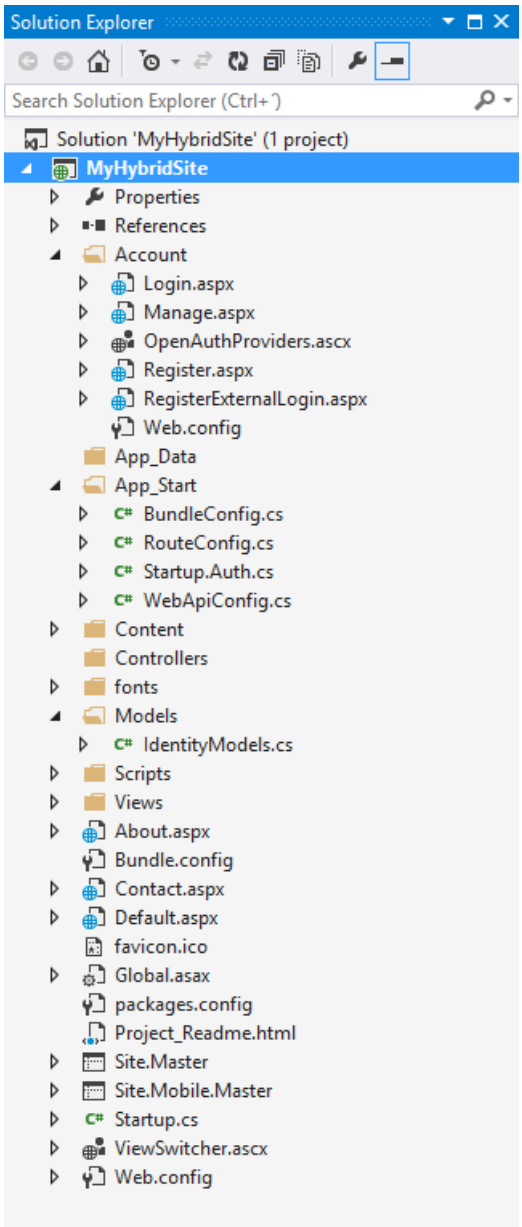
Creating a new ASP.NET Web Application project

3. In the **New ASP.NET Project** dialog box, select the **Web Forms** template and select the **MVC** and **Web API** options. Also, make sure that the **Authentication** option is set to **Individual User Accounts**. Click **OK** to continue.



Creating a new project with the Web Forms template, including Web API and MVC components

4. You can now explore the structure of the generated solution.



Exploring the generated solution

a. **Account:** This folder contains the Web Form pages to register, log in to and manage the application's user accounts. This folder is added when the **Individual User Accounts** authentication option is selected during the configuration of the Web Forms project template.
b. **Models:** This folder will contain the classes that represent your application data.
c. **Controllers** and **Views**: These folders are required for the **ASP.NET MVC** and **ASP.NET Web API** components. You will explore the MVC and Web API technologies in the next exercises.
d. The **Default.aspx**, **Contact.aspx** and **About.aspx** files are pre-defined Web Form pages that you can use as starting points to build the pages specific to your application. The programming logic of those files resides in a separate file referred to as the "code-behind" file, which has an ".aspx.vb" or ".aspx.cs" extension (depending on the language used). The code-behind logic runs on the server and dynamically produces the HTML output for your page.
e. The **Site.Master** and **Site.Mobile.Master** pages define the look and feel and the standard behavior of all the pages in the application.

5. Double-click the **Default.aspx** file to explore the content of the page.

```
Default.aspx
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.Master"
    AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="MyHybridSite._Default" %>

<asp:Content ID="BodyContent" ContentPlaceHolderID="MainContent" runat="server">

    <div class="jumbotron">
        <h1>ASP.NET</h1>
        <p class="lead">ASP.NET is a free web framework for building great Web sites and Web applications
        <p><a href="http://www.asp.net" class="btn btn-primary btn-large">Learn more &raquo;</a></p>
    </div>

    <div class="row">
        <div class="col-md-4">
            <h2>Getting started</h2>
            <p>
                ASP.NET Web Forms lets you build dynamic websites using a familiar drag-and-drop, event-d
            A design surface and hundreds of controls and components let you rapidly build sophisticated,
            </p>
            <p>
                <a class="btn btn-default" href="http://go.microsoft.com/fwlink/?LinkId=301948">Learn mor
            </p>
        </div>
        <div class="col-md-4">
            <h2>Get more libraries</h2>
```

Exploring the Default.aspx page

> **NOTE**
>
> The **Page** directive at the top of the file defines the attributes of the Web Forms page. For example, the **MasterPageFile** attribute specifies the path to the master page -in this case, the *Site.Master* page- and the **Inherits** attribute defines the code-behind class for the page to inherit. This class is located in the file determined by the **CodeBehind** attribute.
>
> The **asp:Content** control holds the actual content of the page (text, markup and controls) and is mapped to a **asp:ContentPlaceHolder** control on the master page. In this case, the page content will be rendered inside the *MainContent* control defined in the *Site.Master* page.

6. Expand the **App_Start** folder and notice the **WebApiConfig.cs** file. Visual Studio included that file in the generated solution because you included Web API when configuring your project with the One ASP.NET template.

7. Open the **WebApiConfig.cs** file. In the *WebApiConfig* class you will find the configuration associated with Web API, which maps HTTP routes to **Web API controllers**.

```
public static void Register(HttpConfiguration config)
{
    // Web API configuration and services

    // Web API routes
    config.MapHttpAttributeRoutes();

    config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
}
```

8. Open the **RouteConfig.cs** file. Inside the *RegisterRoutes* method you will find the configuration associated with MVC, which maps HTTP routes to **MVC controllers**.

```
public static void RegisterRoutes(RouteCollection routes)
{
    var settings = new FriendlyUrlSettings();
    settings.AutoRedirectMode = RedirectMode.Permanent;
    routes.EnableFriendlyUrls(settings);

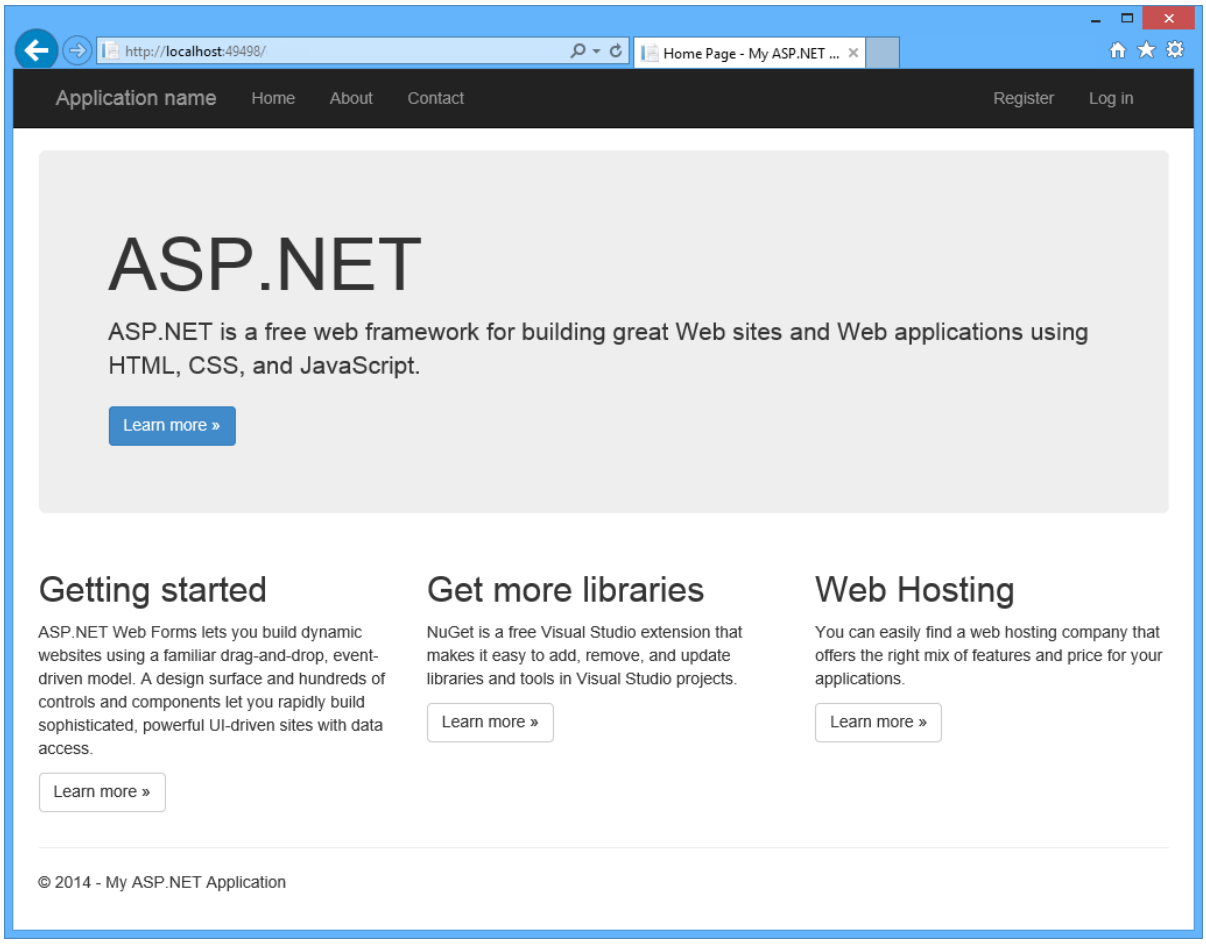
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { action = "Index", id = UrlParameter.Optional }
        );
}
```

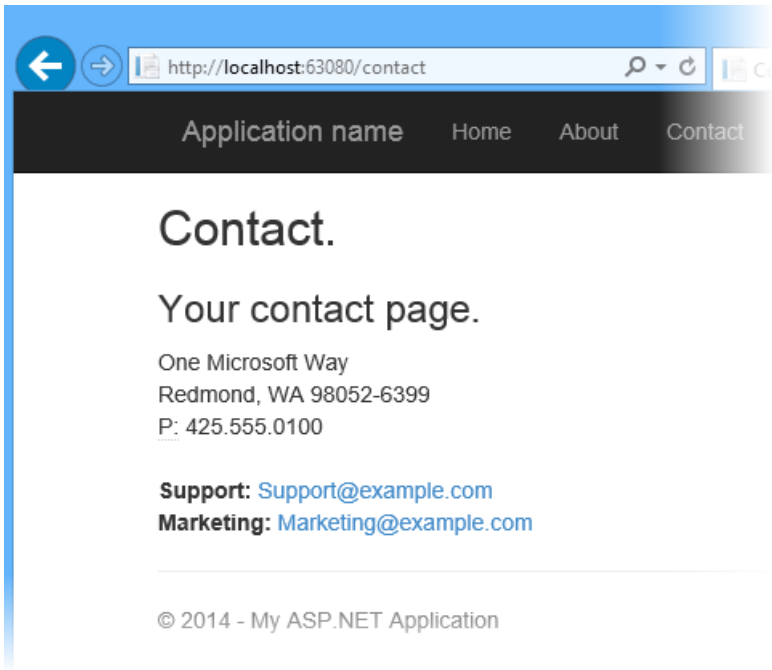
Task 2 – Running the Solution

In this task you will run the generated solution, explore the app and some of its features, like URL rewriting and built-in authentication.

1. To run the solution, press **F5** or click the **Start** button located on the toolbar. The application home page should open in the browser.



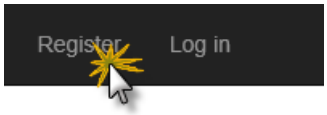
2. Verify that the Web Forms pages are being invoked. To do this, append **/contact.aspx** to the URL in the address bar and press **Enter**.



Friendly URLs

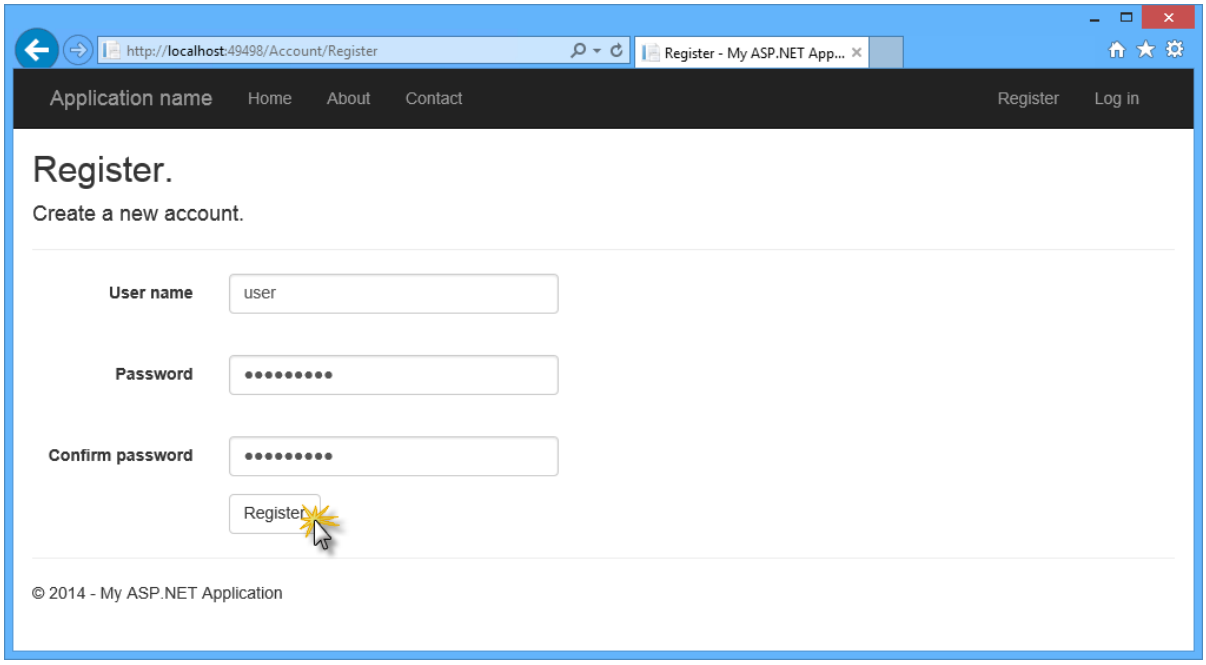
> **NOTE**
>
> As you can see, the URL changes to **/contact**. Starting from **ASP.NET 4**, URL routing capabilities were added to Web Forms, so you can write URLs like *http://www.mysite.com/products/software* instead of *http://www.mysite.com/products.aspx?category=software*. For more information refer to URL Routing.

3. You will now explore the authentication flow integrated into the application. To do this, click **Register** in the upper-right corner of the page.



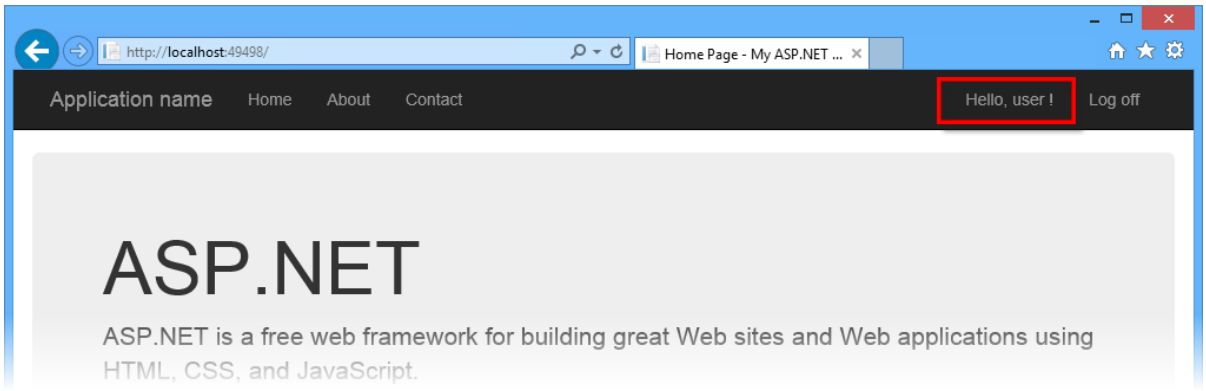
Registering a new user

4. In the **Register** page, enter a **User name** and **Password**, and then click **Register**.



Register page

5. The application registers the new account, and the user is authenticated.



User authenticated

6. Go back to Visual Studio and press **SHIFT + F5** to stop debugging.

Exercise 2: Creating an MVC Controller Using Scaffolding

In this exercise you will take advantage of the ASP.NET Scaffolding framework provided by Visual Studio to create an ASP.NET MVC 5 controller with actions and Razor views to perform CRUD operations, without writing a single line of code. The scaffolding process will use Entity Framework Code First to generate the data context and the database schema in the SQL database.

About Entity Framework Code First

Entity Framework (EF) is an object-relational mapper (ORM) that enables you to create data access applications by programming with a conceptual application model instead of programming directly using a relational storage schema.

The Entity Framework Code First modeling workflow allows you to use your own domain classes to represent the model that EF relies on when performing querying, change-tracking and updating functions. Using the Code First development workflow, you do not need to begin your application by creating a database or specifying a schema. Instead, you can write standard .NET classes that define the most appropriate domain model objects for your application, and Entity Framework will create the database for you.

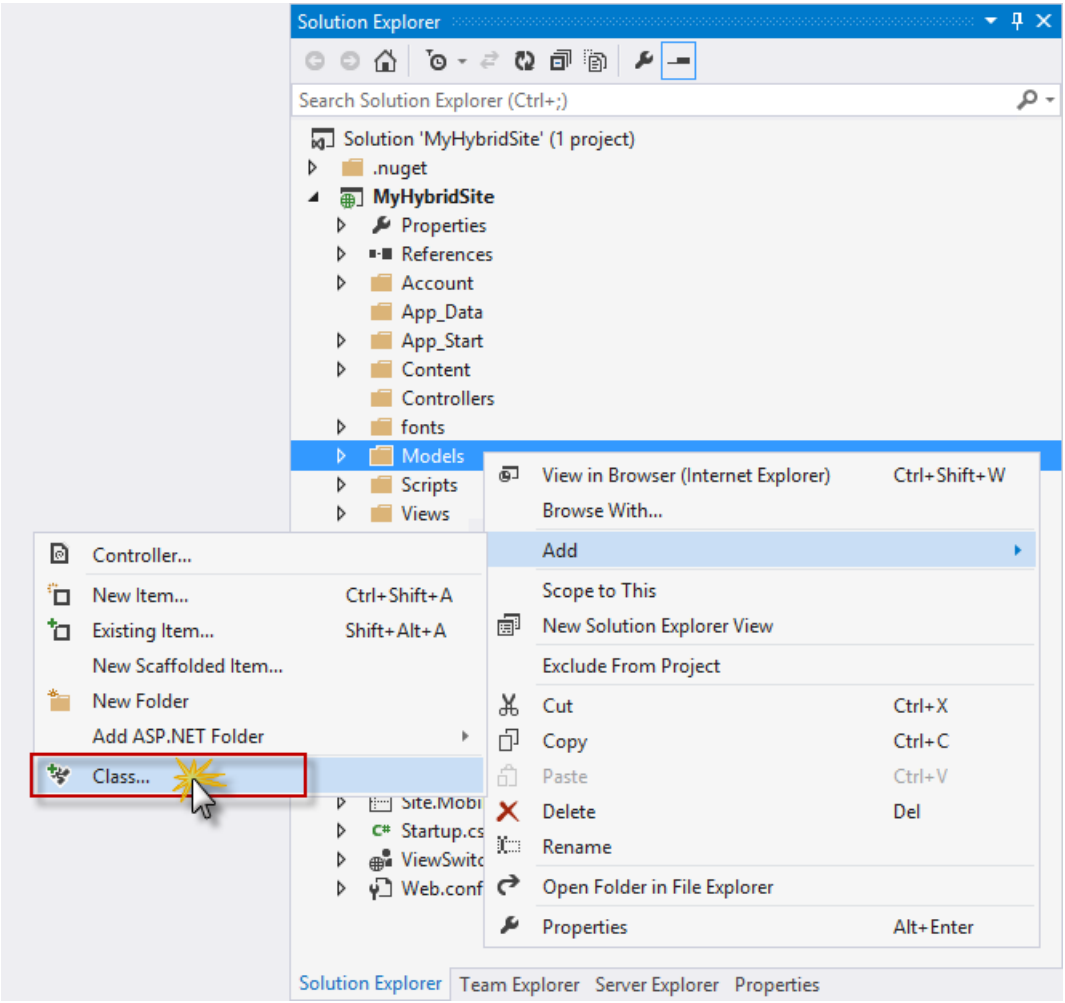
> **NOTE**
>
> You can learn more about Entity Framework here.

Task 1 – Creating a New Model

You will now define a **Person** class, which will be the model used by the scaffolding process to create the MVC controller and the views. You will start by creating a **Person** model class, and the CRUD operations in the controller will be automatically created using scaffolding features.

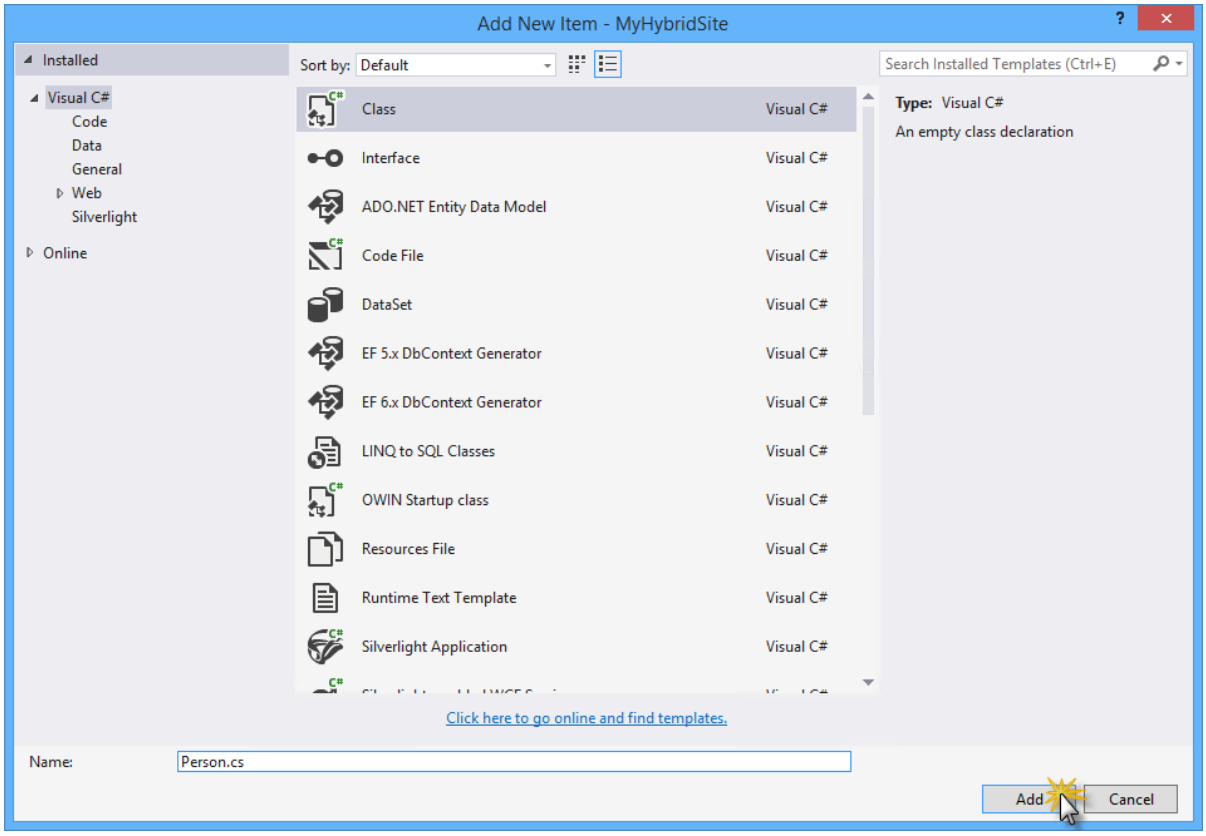
1. Open **Visual Studio Express 2013 for Web** and the **MyHybridSite.sln** solution located in the **Source/Ex2-MvcScaffolding/Begin** folder. Alternatively, you can continue with the solution that you obtained in the previous exercise.

2. In **Solution Explorer**, right-click the **Models** folder of the **MyHybridSite** project and select **Add | Class...**.



Adding the Person model class

3. In the **Add New Item** dialog box, name the file *Person.cs* and click **Add**.



Creating the Person model class

4. Replace the content of the **Person.cs** file with the following code. Press **CTRL + S** to save the changes.

   (Code Snippet - *BringingTogetherOneAspNet - Ex2 - PersonClass*)

```
namespace MyHybridSite.Models
{
    public class Person
    {
        public int Id { get; set; }

        public string Name { get; set; }

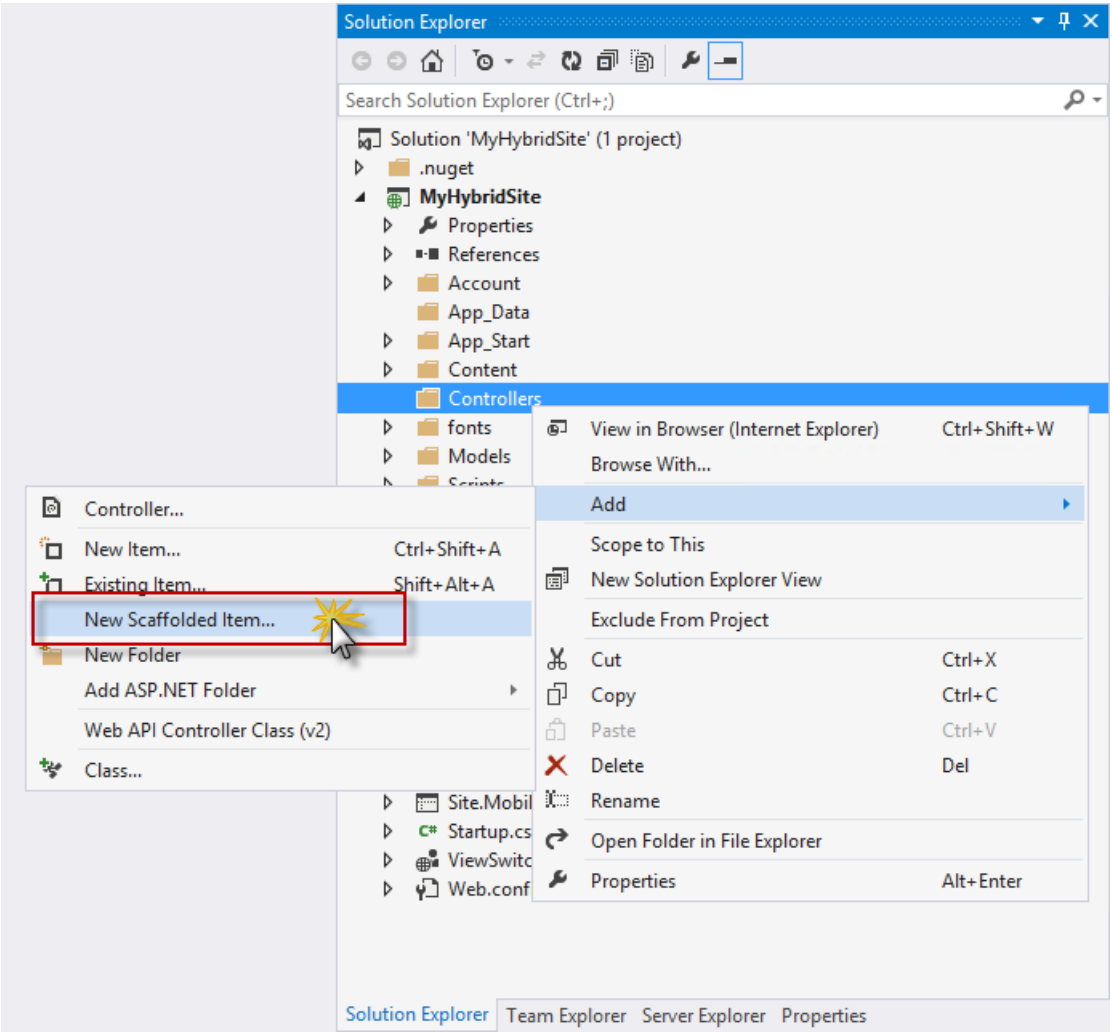
        public int Age { get; set; }
    }
}
```

5. In **Solution Explorer**, right-click the **MyHybridSite** project and select **Build**, or press **CTRL + SHIFT + B** to build the project.

Task 2 – Creating an MVC Controller

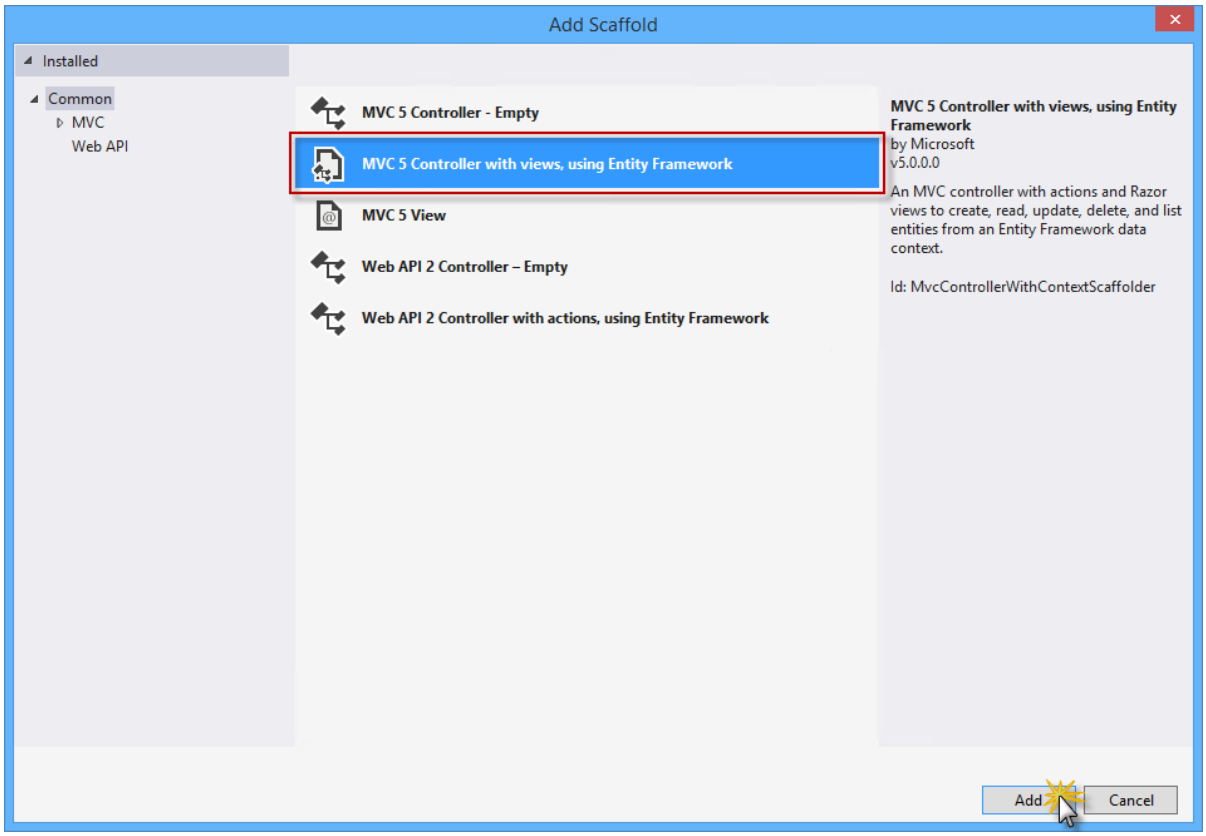
Now that the **Person** model is created, you will use ASP.NET MVC scaffolding with Entity Framework to create the CRUD controller actions and views for **Person**.

1. In **Solution Explorer**, right-click the **Controllers** folder of the **MyHybridSite** project and select **Add | New Scaffolded Item...**.



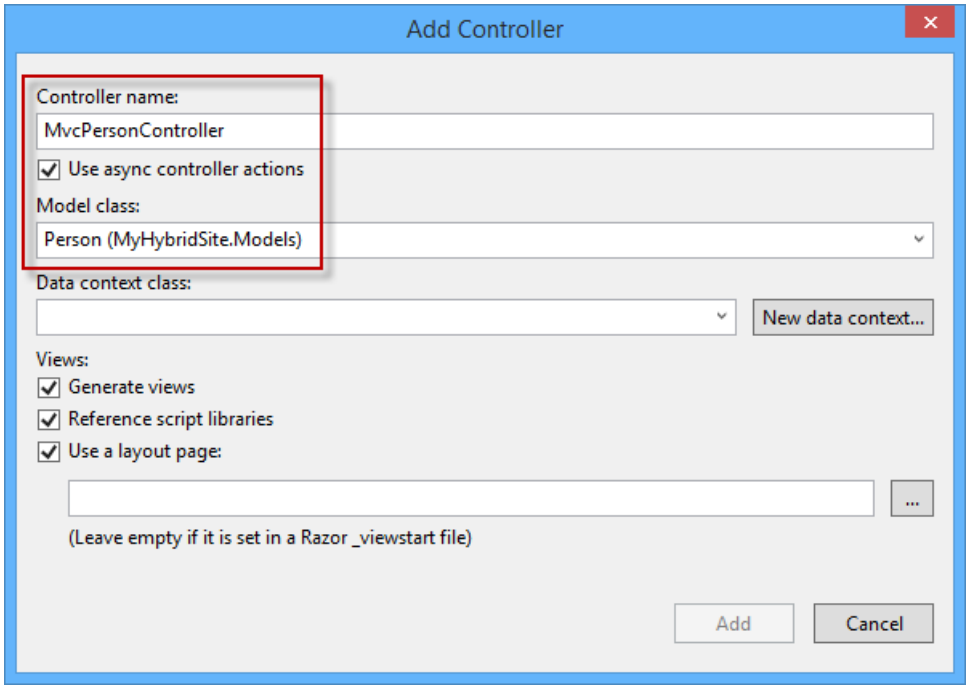
Creating a new Scaffolded Controller

2. In the **Add Scaffold** dialog box, select **MVC 5 Controller with views, using Entity Framework** and then click **Add**.



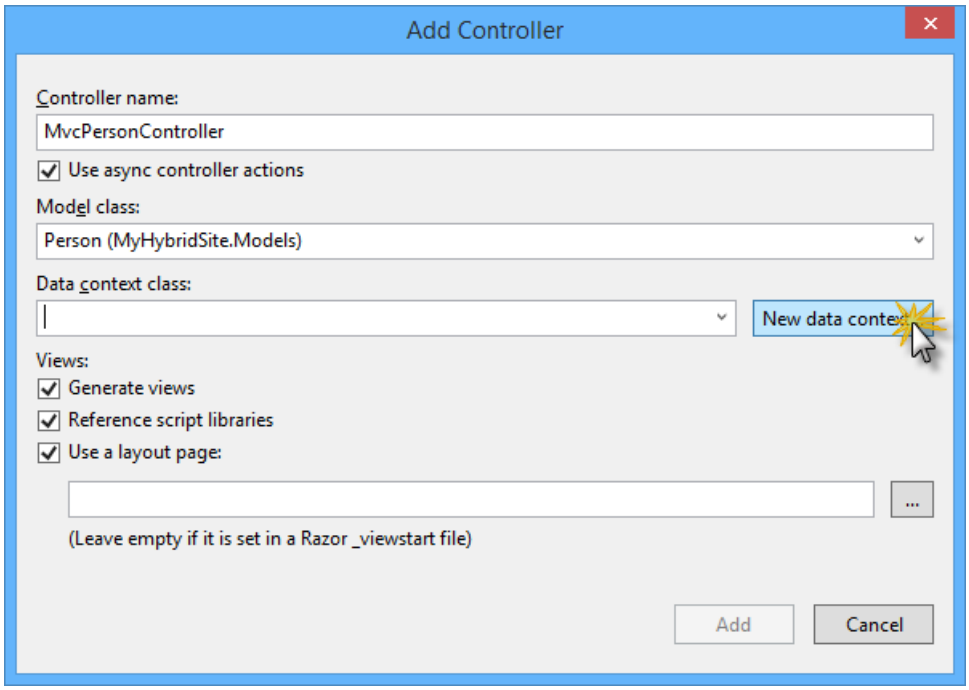
Selecting MVC 5 Controller with views and Entity Framework

3. Set *MvcPersonController* as the **Controller name**, select the **Use async controller actions** option and select **Person (MyHybridSite.Models)** as the **Model class**.



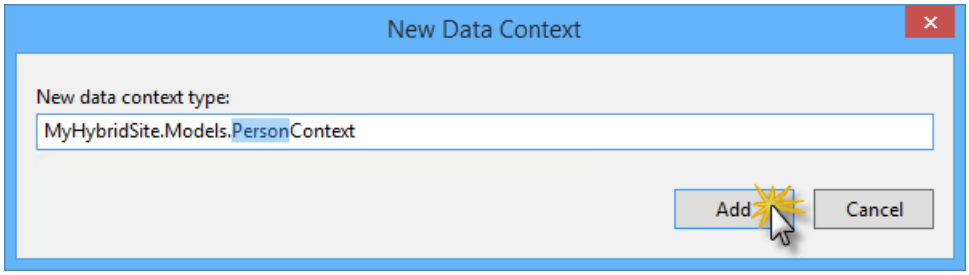
Adding an MVC controller with scaffolding

4. Under **Data context class**, click **New data context...**.



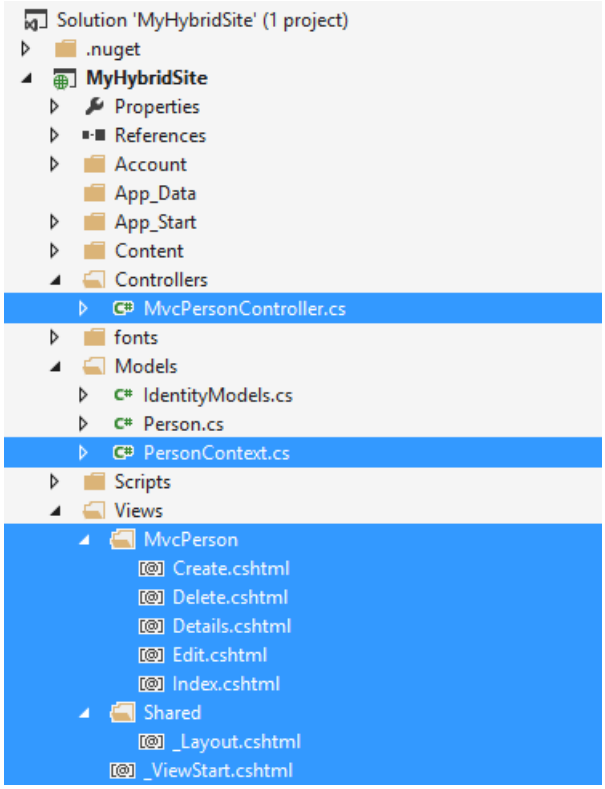
Creating a new data context

5. In the **New Data Context** dialog box, name the new data context *PersonContext* and click **Add**.



Creating the new PersonContext type

6. Click **Add** to create the new controller for **Person** with scaffolding. Visual Studio will then generate the controller actions, the Person data context and the Razor views.



After creating the MVC controller with scaffolding

7. Open the **MvcPersonController.cs** file in the **Controllers** folder. Notice that the CRUD action methods have been generated automatically.

```
...

// POST: /MvcPerson/Create
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see https://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Create([Bind(Include="Id,Name,Age")] Person person)
{
     if (ModelState.IsValid)
     {
          db.People.Add(person);
          await db.SaveChangesAsync();
          return RedirectToAction("Index");
     }

     return View(person);
}

// GET: /MvcPerson/Edit/5
public async Task<ActionResult> Edit(int? id)
{
     if (id == null)
     {
          return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
     }
     Person person = await db.People.FindAsync(id);
     if (person == null)
     {
          return HttpNotFound();
     }
     return View(person);
}

...
```

> **NOTE**
>
> By selecting the **Use async controller actions** check box from the scaffolding options in the previous steps, Visual Studio generates asynchronous action methods for all actions that involve access to the Person data context. It is recommended that you use asynchronous action methods for long-running, non-CPU bound requests to avoid blocking the Web server from performing work while the request is being processed.

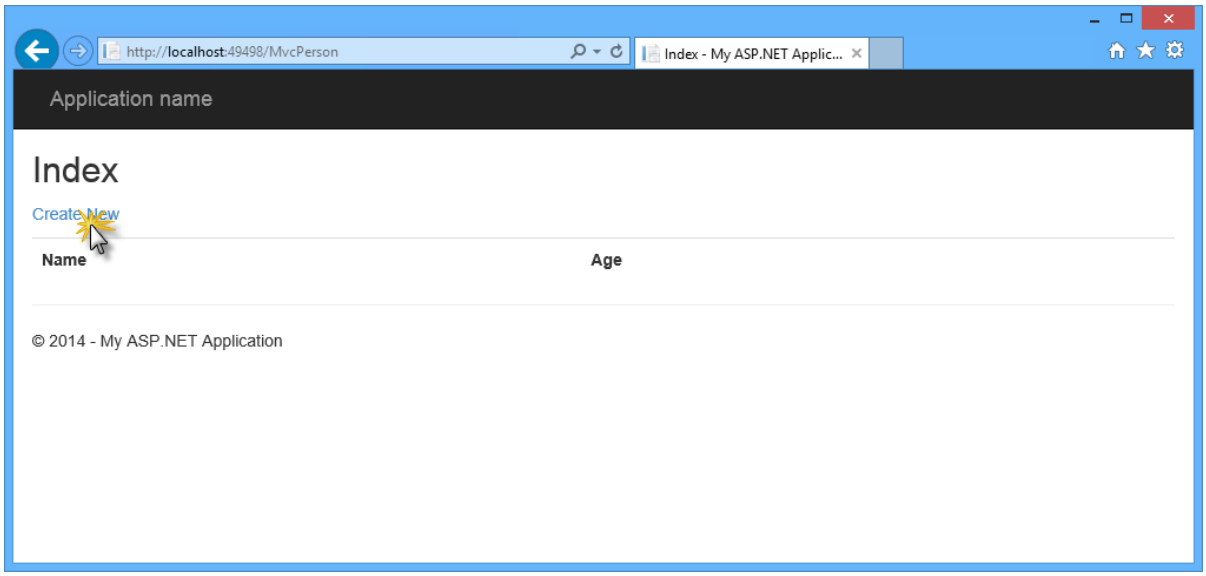
Task 3 – Running the Solution

In this task, you will run the solution again to verify that the views for **Person** are working as expected. You will add a new person to verify that it is successfully saved to the database.

1. Press **F5** to run the solution.

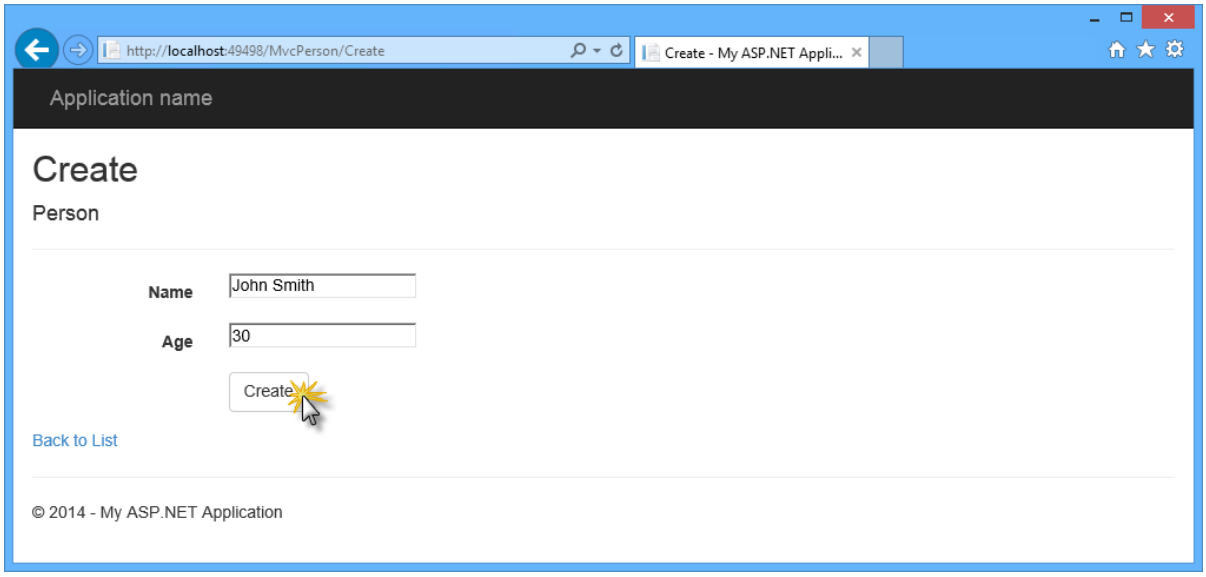
2. Navigate to **/MvcPerson**. The scaffolded view that shows the list of people should appear.

3. Click **Create New** to add a new person.



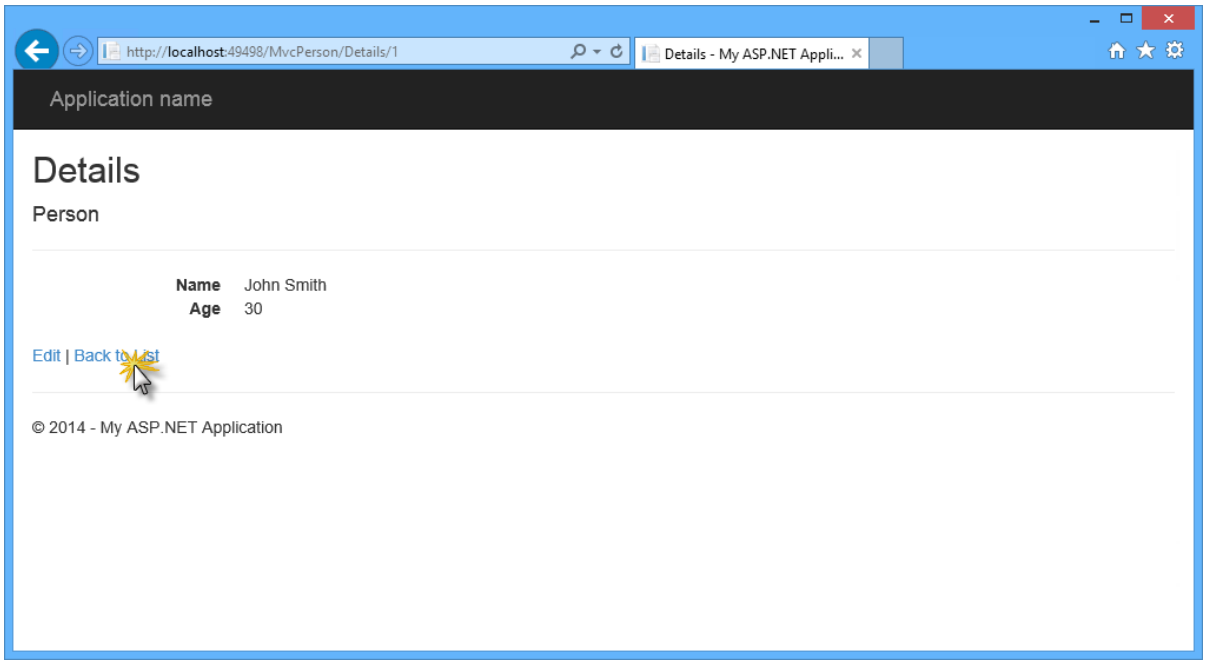
Navigating to the scaffolded MVC views

4. In the **Create** view, provide a **Name** and an **Age** for the person, and click **Create**.



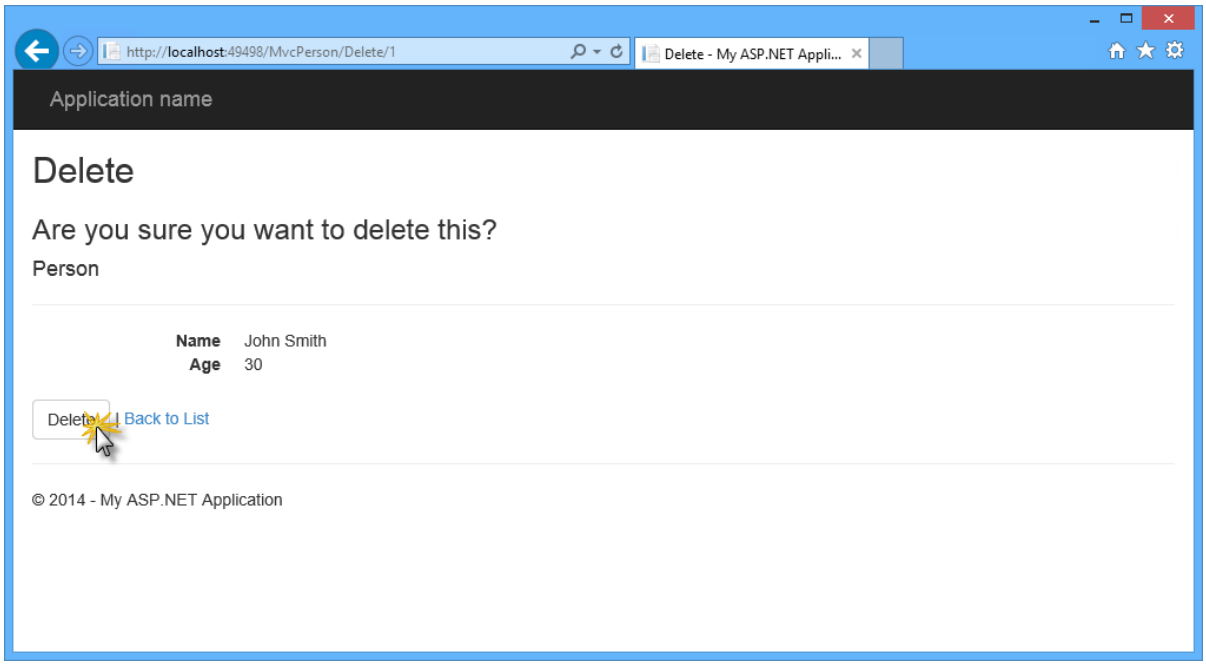
Adding a new person

5. The new person is added to the list. In the element list, click **Details** to display the person's details view. Then, in the **Details** view, click **Back to List** to go back to the list view.



Person's details view

6. Click the **Delete** link to delete the person. In the **Delete** view, click **Delete** to confirm the operation.



Deleting a person

7. Go back to Visual Studio and press **SHIFT + F5** to stop debugging.

Exercise 3: Creating a Web API Controller Using Scaffolding

The Web API framework is part of the ASP.NET Stack and designed to make implementing HTTP services easier, generally sending and receiving JSON- or XML-formatted data through a RESTful API.

In this exercise, you will use ASP.NET Scaffolding again to generate a Web API controller. You will use the same **Person** and **PersonContext** classes from the previous exercise to provide the same person data in JSON format. You will see how you can expose the same resources in different ways within the same ASP.NET application.

Task 1 – Creating a Web API Controller

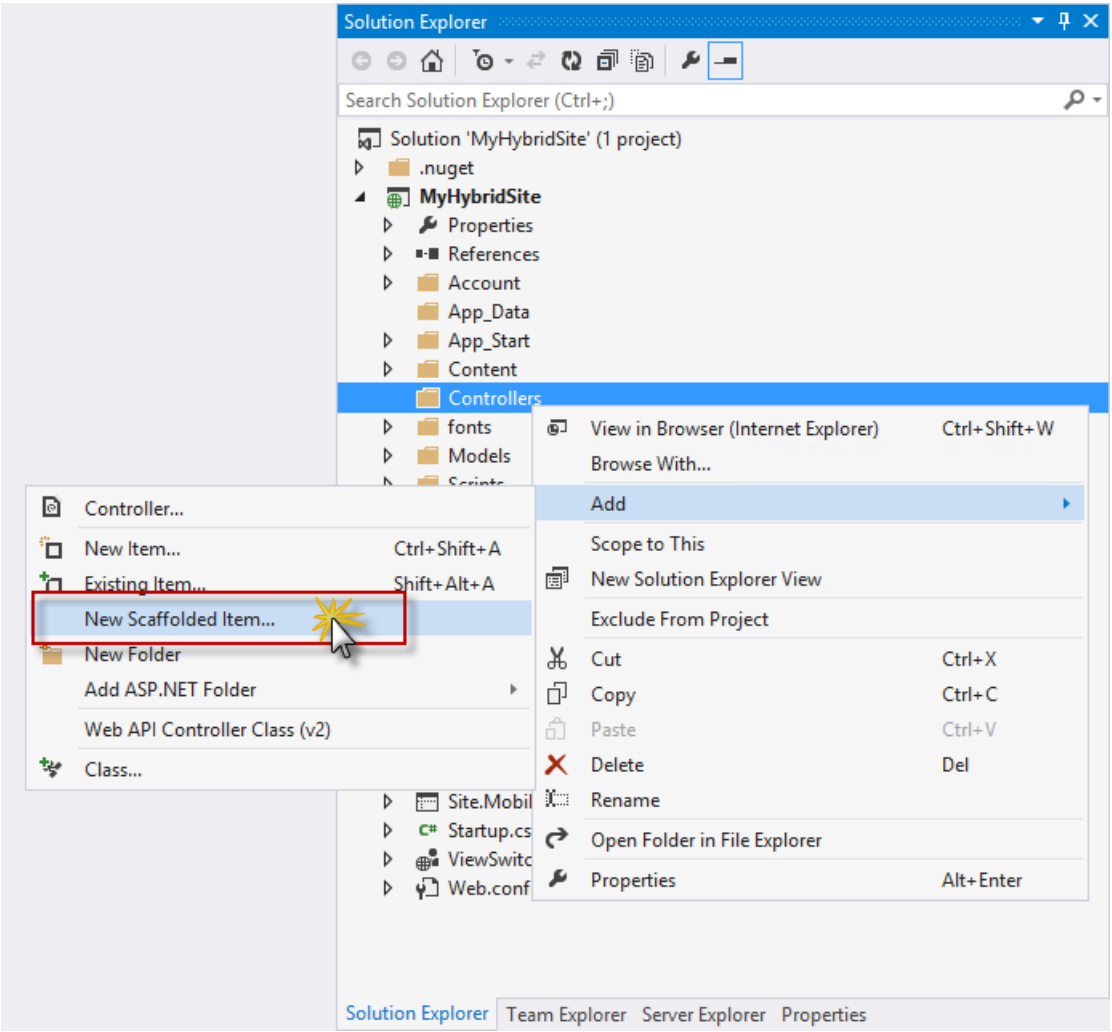
In this task you will create a new **Web API Controller** that will expose the person data in a machine-consumable format like JSON.

1. If not already opened, open **Visual Studio Express 2013 for Web** and open the **MyHybridSite.sln**

solution located in the **Source/Ex3-WebAPI/Begin** folder. Alternatively, you can continue with the solution that you obtained in the previous exercise.

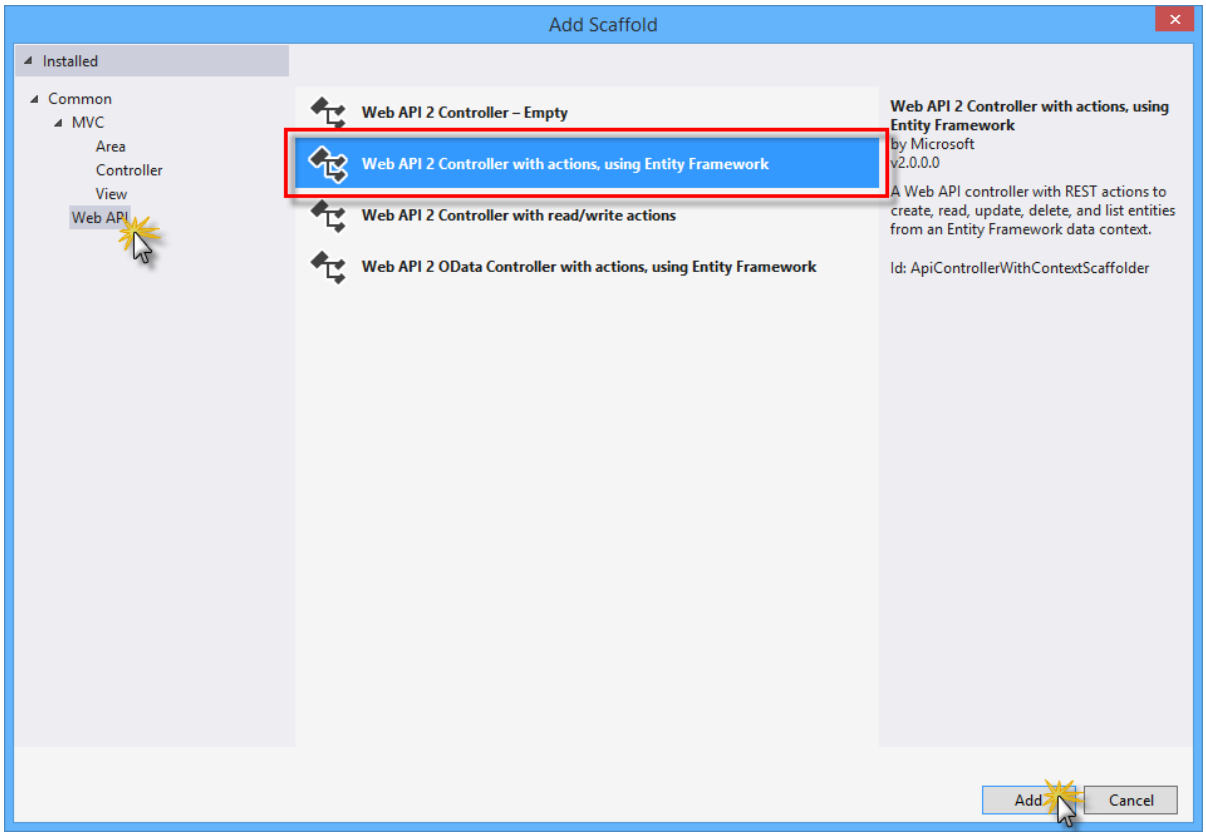
> **NOTE**
>
> If you start with the Begin solution from Exercise 3, press **CTRL + SHIFT + B** to build the solution.

2. In **Solution Explorer**, right-click the **Controllers** folder of the **MyHybridSite** project and select **Add | New Scaffolded Item...**.



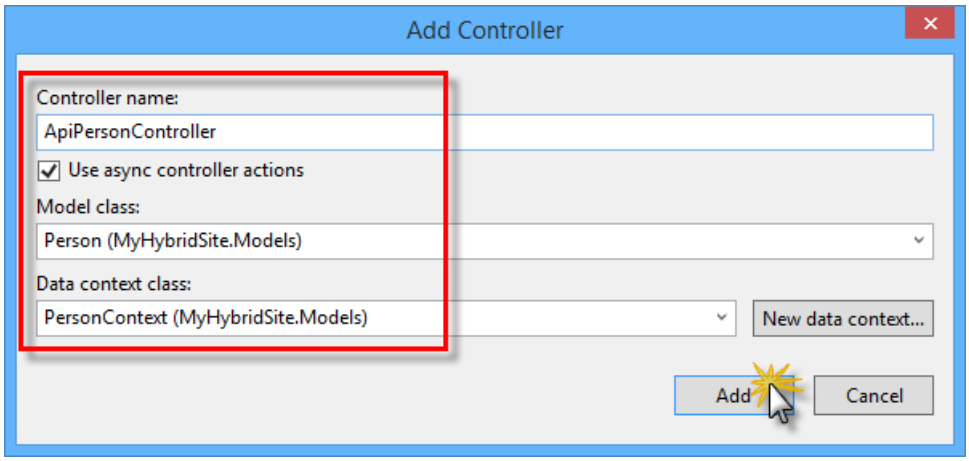
Creating a new scaffolded Controller

3. In the **Add Scaffold** dialog box, select **Web API** in the left pane, then **Web API 2 Controller with actions, using Entity Framework** in the middle pane and then click **Add**.



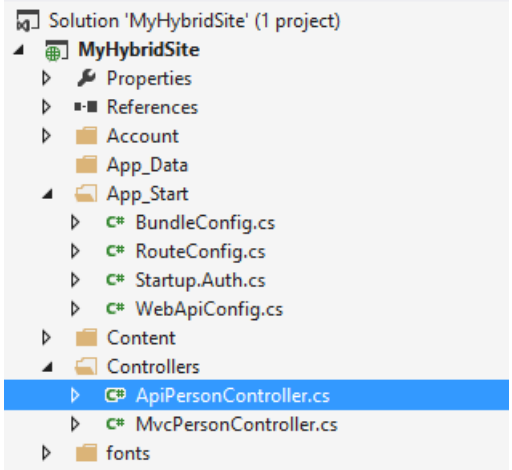
Selecting Web API 2 Controller with actions and Entity Framework

4. Set *ApiPersonController* as the **Controller name**, select the **Use async controller actions** option and select **Person (MyHybridSite.Models)** and **PersonContext (MyHybridSite.Models)** as the **Model** and **Data context** classes respectively. Then click **Add**.



Adding a Web API controller with scaffolding

5. Visual Studio will then generate the **ApiPersonController** class with the four CRUD actions to work with your data.



After creating the Web API controller with scaffolding

6. Open the **ApiPersonController.cs** file and inspect the *GetPeople* action method. This method queries the db field of **PersonContext** type in order to get the people data.

```
// GET api/ApiPerson
public IQueryable<Person> GetPeople()
{
    return db.People;
}
```

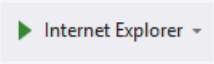
7. Now notice the comment above the method definition. It provides the URI that exposes this action which you will use in the next task.

```
// GET api/ApiPerson
public IQueryable<Person> GetPeople()
{
    return db.People;
}
```

> **NOTE**
>
> By default, Web API is configured to catch the queries to the */api* path to avoid collisions with MVC controllers. If you need to change this setting, refer to Routing in ASP.NET Web API.

Task 2 – Running the Solution

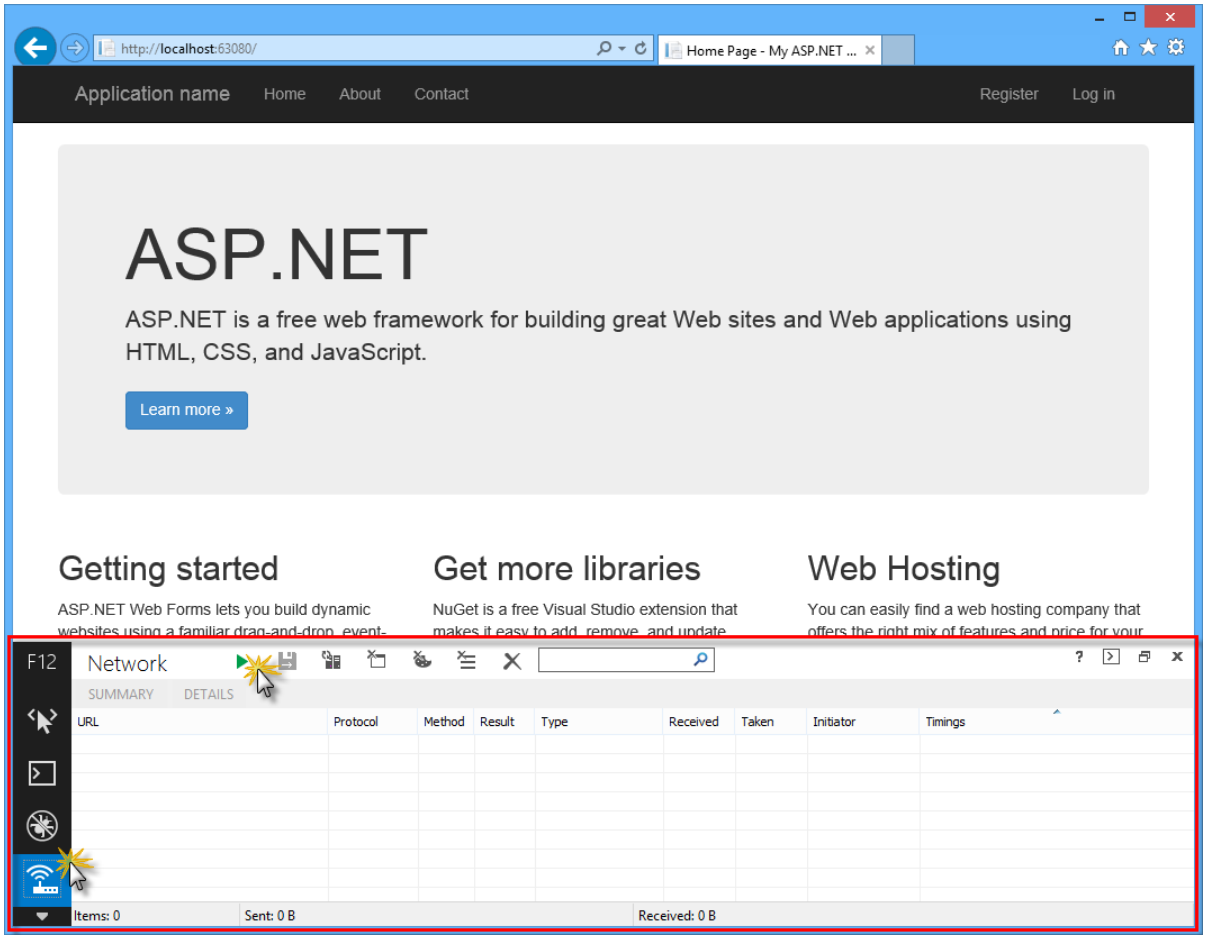
In this task you will use the Internet Explorer **F12 developer tools** to inspect the full response from the Web API controller. You will see how you can capture network traffic to get more insight into your application data.

> **NOTE**
>
> Make sure that **Internet Explorer** is selected in the **Start** button located on the Visual Studio toolbar.
>
> 
>
>
> The **F12 developer tools** have a wide set of functionality that is not covered in this hands-on-lab. If you want to learn more about it, refer to Using the F12 developer tools.

1. Press **F5** to run the solution.

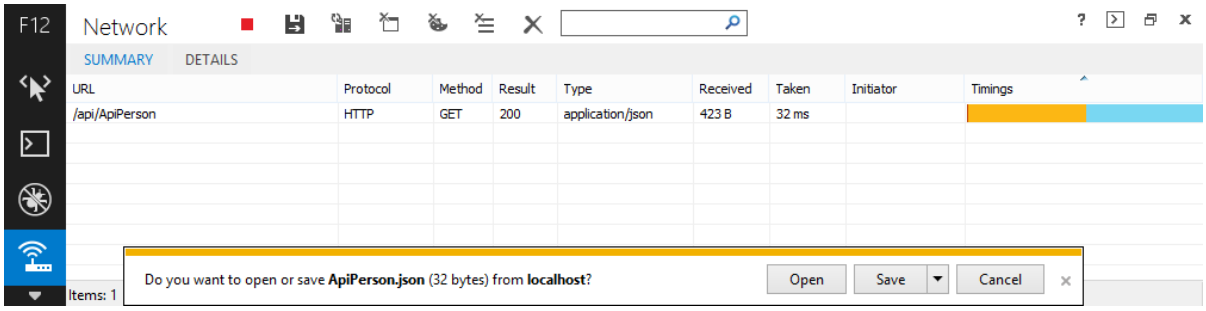
> **NOTE**
>
> In order to follow this task correctly, your application needs to have data. If your database is empty, you can go back to Task 3 in Exercise 2 and follow the steps on how to create a new person using the MVC views.

2. In the browser, press **F12** to open the **Developer Tools** panel. Press **CTRL + 4** or click the **Network** icon, and then click the green arrow button to begin capturing network traffic.



Initiating Web API network capture

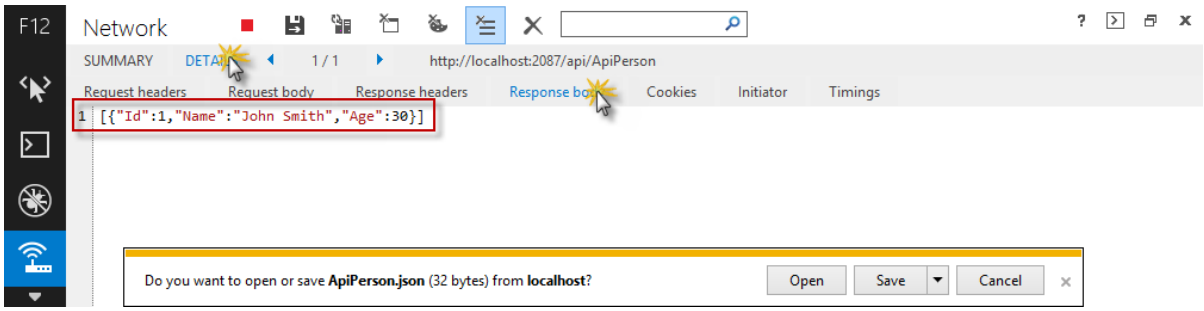
3. Append **api/ApiPerson** to the URL in the browser's address bar. You will now inspect the details of the response from the **ApiPersonController**.



Retrieving person data through Web API

> **NOTE**
>
> Once the download finishes, you will be prompted to make an action with the downloaded file. Leave the dialog box open in order to be able to watch the response content through the Developers Tool window.

4. Now you will inspect the body of the response. To do this, click the **Details** tab and then click **Response body**. You can check that the downloaded data is a list of objects with the properties **Id**, **Name** and **Age** that correspond to the **Person** class.



Viewing Web API Response Body

Task 3 – Adding Web API Help Pages

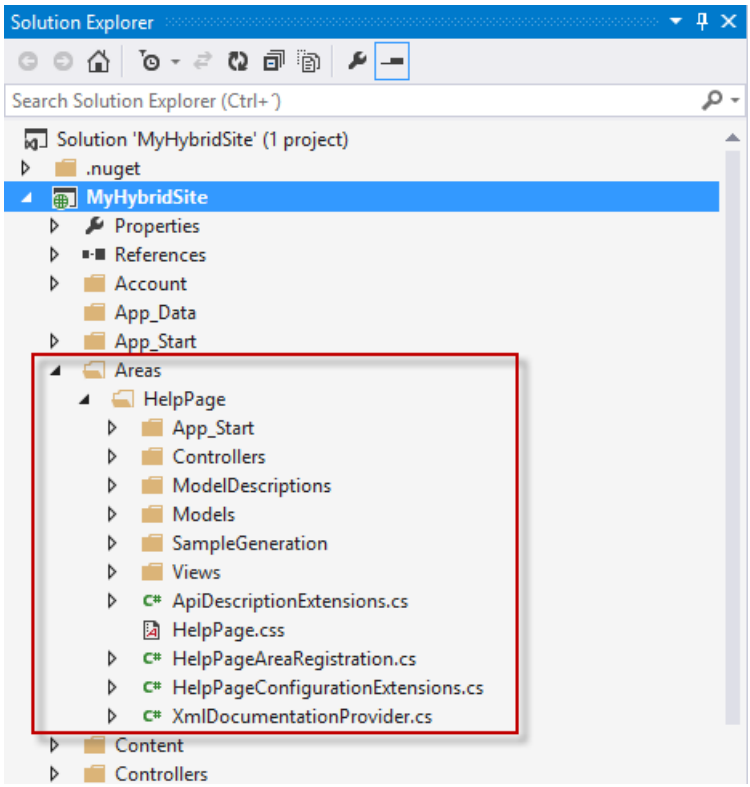
When you create a Web API, it is useful to create a help page so that other developers will know how to call your API. You could create and update the documentation pages manually, but it is better to auto-generate them to avoid having to do maintenance work. In this task you will use a Nuget package to automatically generate Web API help pages to the solution.

1. From the **Tools** menu in Visual Studio, select **NuGet Package Manager**, and then click **Package Manager Console**.

2. In the **Package Manager Console** window, execute the following command:

```
Install-Package Microsoft.AspNet.WebApi.HelpPage
```

> **NOTE**
>
> The **Microsoft.AspNet.WebApi.HelpPage** package installs the necessary assemblies and adds MVC Views for the help pages under the **Areas/HelpPage** folder.

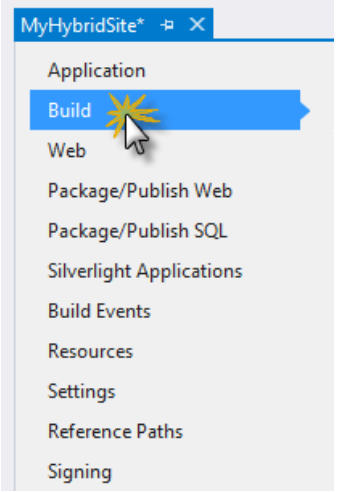


HelpPage Area

3. By default, the help pages have placeholder strings for documentation. You can use XML documentation comments to create the documentation. To enable this feature, open the **HelpPageConfig.cs** file located in the **Areas/HelpPage/App_Start** folder and uncomment the following line:

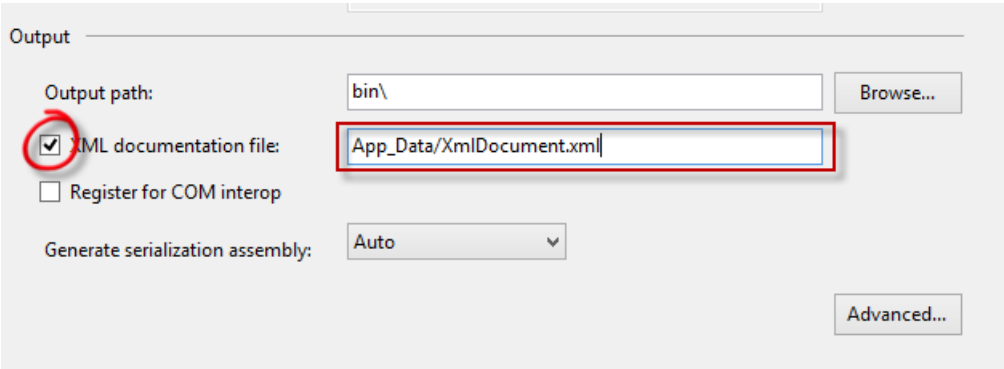
```
config.SetDocumentationProvider(new
XmlDocumentationProvider(HttpContext.Current.Server.MapPath("~/App_Data/XmlDocument.xml")));
```

4. In **Solution Explorer**, right-click the project **MyHybridSite**, select **Properties** and click the **Build** tab.



Build tab

5. Under **Output**, select **XML documentation file**. In the edit box, type **App_Data/XmlDocument.xml**.



Output section in Build tab

6. Press **CTRL + S** to save the changes.

7. Open the **ApiPersonController.cs** file from the **Controllers** folder.

8. Enter a new line between the *GetPeople* method signature and the *// GET api/ApiPerson* comment, and then type three forward slashes.

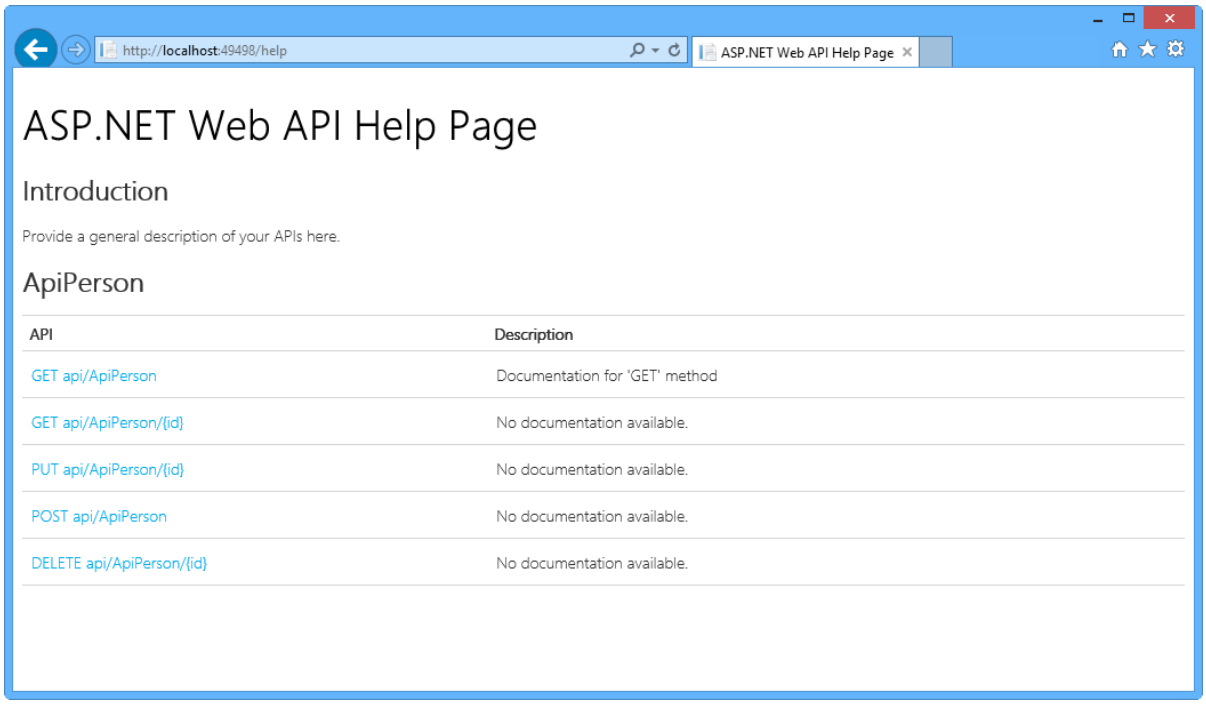
> **NOTE**
>
> Visual Studio automatically inserts the XML elements which define the method documentation.

9. Add a summary text and the return value for the *GetPeople* method. It should look like the following.

```
// GET api/ApiPerson
/// <summary>
/// Documentation for 'GET' method
/// </summary>
/// <returns>Returns a list of people in the requested format</returns>
public IQueryable<Person> GetPeople()
{
    return db.People;
}
```

10. Press **F5** to run the solution.

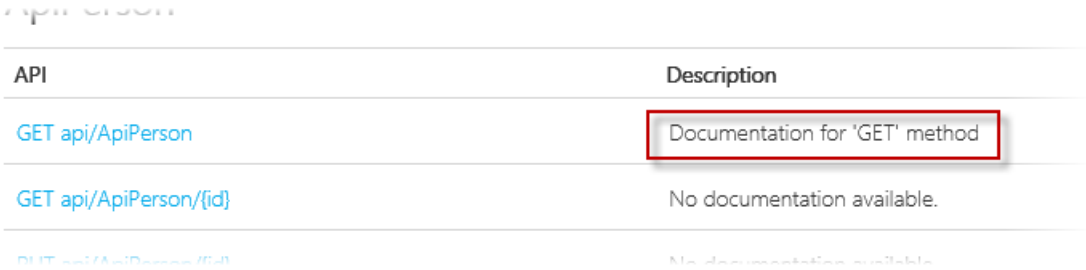
11. Append **/help** to the URL in the address bar to browse to the help page.



ASP.NET Web API Help Page

> **NOTE**
>
> The main content of the page is a table of APIs, grouped by controller. The table entries are generated dynamically, using the **IApiExplorer** interface. If you add or update an API controller, the table will be automatically updated the next time you build the application.
>
> The **API** column lists the HTTP method and relative URI. The **Description** column contains information that has been extracted from the method's documentation.

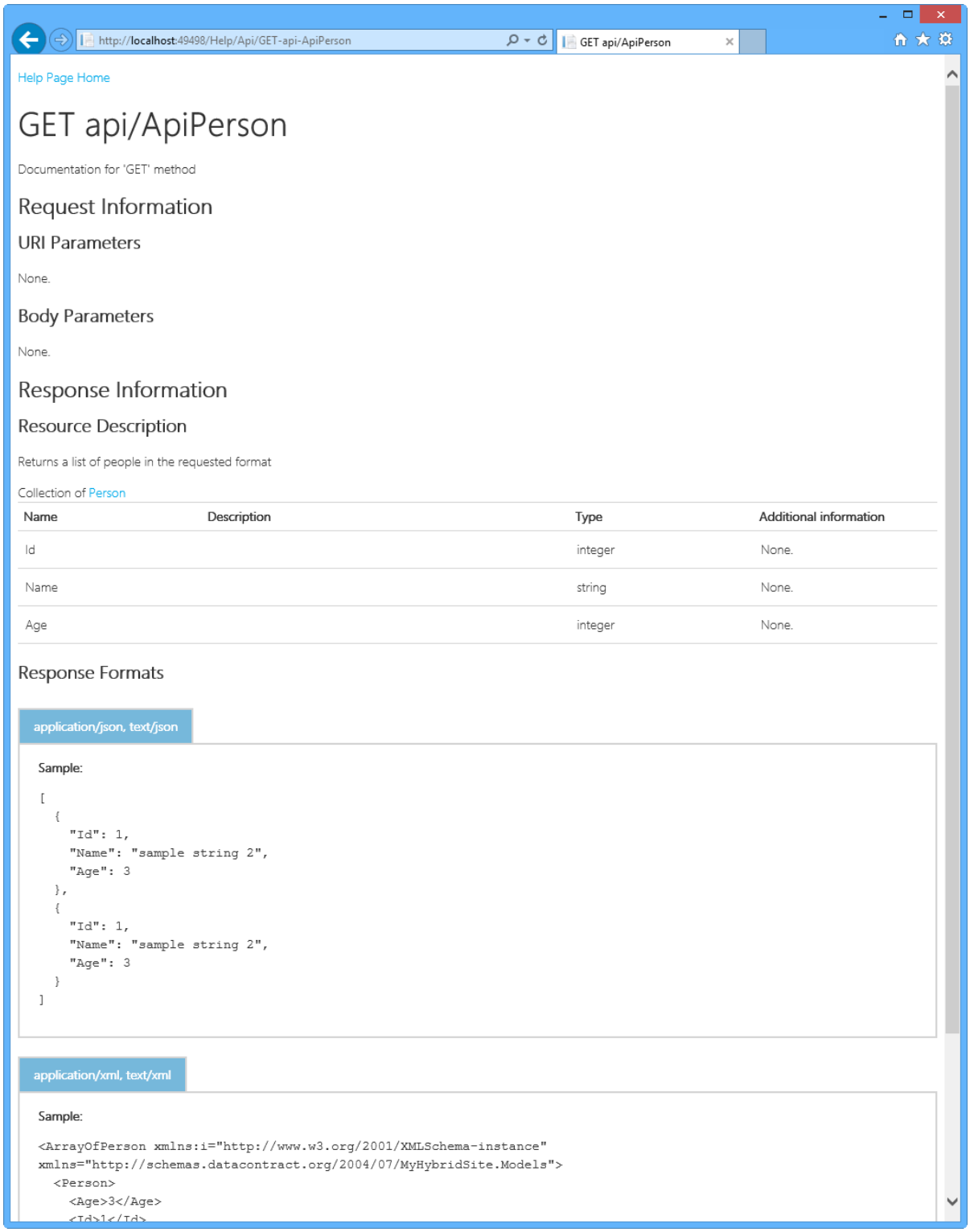
12. Note that the description you added above the method definition is displayed in the description column.



API method description

13. Click one of the API methods to navigate to a page with more detailed information, including sample

response bodies.



Detailed information page

Summary

By completing this hands-on lab you have learned how to:

- Create a new Web application using the One ASP.NET Experience in Visual Studio 2013
- Integrate multiple ASP.NET technologies into one single project
- Generate MVC controllers and views from your model classes using ASP.NET Scaffolding
- Generate Web API controllers, which use features such as Async Programming and data access through Entity Framework

- Automatically generate Web API Help Pages for your controllers

ASP.NET and Visual Studio 2012

2/20/2020 • 2 minutes to read • Edit Online

Developing ASP.NET web apps with Visual Studio 2012

- Visual Studio 2012 HTML Editing Features
- Visual Studio 2012 CSS Editor
- Visual Studio 2012 JavaScript Editor
- Visual Studio 2012 Page Inspector
- Visual Studio 2012 Web Publishing Improvements
- Release Notes for ASP.NET and Web Tools 2013.1 for Visual Studio 2012
- HTML Editor Smart Tasks and Event Handler Generation
- CSS Editor Hierarchical Indentation
- CSS Editor Snippets
- CSS Editor Color Picker
- Page Inspector - Introduction
- Page Inspector - Decomposing your Web Application
- Windows Azure Authentication
- ASP.NET and Web Tools 2012.2 Release Notes

Visual Studio 2012 HTML Editing Features

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Watch video (4 minutes)

Visual Studio 2012 CSS Editor

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Watch video (4 minutes)

Visual Studio 2012 JavaScript Editor

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Watch video (5 minutes)

Visual Studio 2012 Page Inspector

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Watch video (3 minutes)

Visual Studio 2012 Web Publishing Improvements

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Watch video (5 minutes)

Release Notes for ASP.NET and Web Tools 2013.1 for Visual Studio 2012

4/10/2020 • 6 minutes to read • Edit Online

by Microsoft

This document describes the release of ASP.NET and Web Tools 2013.1 for Visual Studio 2012.

Contents

- Installation Notes
- Software Requirements
- New Features in ASP.NET and Web Tools 2013.1 for Visual Studio 2012
  - Bootstrap
  - Templates
    - ASP.NET MVC 5 template
    - ASP.NET Web API 2 template
    - Item Templates
  - Entity Framework 6
  - ASP.NET Scaffolding
  - Razor Editor
  - NuGet 2.7
- Known Issues and Breaking Changes
  - ASP.NET Scaffolding
    - MVC and Web API Scaffolding - HTTP 404, Not Found error
    - Visual Studio Express 2012 for Web stops working after adding a scaffolded item
  - ASP.NET Razor 3
    - Viewing cshtml file with Browse With or F5 causes a server error
    - Url Rewrite and Tilde(~)
  - Templates

Installation Notes

Install ASP.NET and Web Tools 2013.1 for Visual Studio 2012.

Software Requirements

You must have either Visual Studio 2012 or Visual Studio Express 2012 for Web.

New Features in ASP.NET and Web Tools 2013.1 for Visual Studio 2012

Bootstrap

When you scaffold MVC 5 controllers and views, the markup for the views uses Bootstrap.

Templates

ASP.NET MVC 5 template

We added a new MVC 5 template. It references the latest MVC 5 NuGet packages, and you can use scaffolding to add controllers and views.

ASP.NET Web API 2 template

We added a new Web API 2 template. It references the latest Web API 2 NuGet packages, and you can use scaffolding to add controllers and views.

Item Templates

We added new item templates for MVC 5 views, Web Pages (Razor 3), and Web API 2 controllers. They install the related NuGet packages to the project while adding new items.

Entity Framework 6

When you scaffold an MVC or Web API controller using Entity Framework, we use Framework 6. For more information about Entity Framework, see the Entity Framework Version History.

You can also download and install the Entity Framework 6 Tools for Visual Studio 2012. See the Get Entity Framework.

ASP.NET Scaffolding

ASP.NET Scaffolding is a code generation framework for ASP.NET Web applications. It makes it easy to add boilerplate code to your project that interacts with a data model.

In previous versions of Visual Studio, scaffolding was limited to ASP.NET MVC projects. With this update, you can now use scaffolding for any ASP.NET project, including Web Forms. This update does not support generating pages for a Web Forms project, but you can still use scaffolding with Web Forms by adding MVC dependencies to the project. Support for generating pages for Web Forms will be added in a future update.

When using scaffolding, we ensure that all required dependencies are installed in the project. For example, if you start with an ASP.NET Web Forms project and then use scaffolding to add a Web API Controller, the required NuGet packages and references are added to your project automatically.

To add MVC scaffolding to a Web Forms project, add a **New Scaffolded Item** and select **MVC 5 Dependencies** in the dialog window. There are two options for scaffolding MVC; Minimal and Full. If you select Minimal, only the NuGet packages and references for ASP.NET MVC are added to your project. If you select the Full option, the Minimal dependencies are added, as well as the required content files for an MVC project.

Support for scaffolding async controllers uses the new async features from Entity Framework 6.

For more information and tutorials, see ASP.NET Scaffolding Overview. These tutorials show scaffolding with Visual Studio 2013, but they are also applicable to ASP.NET and Web Tools 2013.1 for Visual Studio 2012.

Razor Editor

With this update, Visual Studio 2012 now supports Razor 3 tooling/editing.

NuGet 2.7

NuGet 2.7 includes a rich set of new features which are described in detail at NuGet 2.7 Release Notes.

This version of NuGet removes the need for users to explicitly allow NuGet to restore missing packages. When installing NuGet 2.7, users implicitly consent to automatically restoring missing packages. Users can explicitly opt out of package restoration through the NuGet settings in Visual Studio. This change simplifies how package restoration works.

Known Issues and Breaking Changes

ASP.NET Scaffolding

MVC and Web API Scaffolding - HTTP 404, Not Found error

If you encounter an error when adding a scaffolded item to a project, it is possible your project will be left in an inconsistent state. Some of the changes made be scaffolding will be rolled back but other changes, such as the installed NuGet packages, will not be rolled back. If the routing configuration changes are rolled back, users will receive an HTTP 404 error when navigating to scaffolded items.

To fix this error for MVC, add a new scaffolded item and select MVC 5 Dependencies (either Minimal or Full). This process will add all of the required changes to your project.

To fix this error for Web API:

1. Add the following WebApiConfig class to your project.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.MapHttpAttributeRoutes();
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

```
Public Module WebApiConfig
    Public Sub Register(ByVal config As HttpConfiguration)
        config.MapHttpAttributeRoutes()
        config.Routes.MapHttpRoute(
          name:="DefaultApi",
          routeTemplate:="api/{controller}/{id}",
          defaults:=New With {.id = RouteParameter.Optional}
        )
    End Sub
End Module
```

2. Configure WebApiConfig.Register in the Application_Start method in Global.asax as follows:

```
public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        GlobalConfiguration.Configure(WebApiConfig.Register);
    }
}
```

```
Public Class WebApiApplication
     Inherits System.Web.HttpApplication

     Sub Application_Start()
       GlobalConfiguration.Configure(AddressOf WebApiConfig.Register)
     End Sub
End Class
```

Visual Studio Express 2012 for Web stops working after adding a scaffolded item

If Visual Studio Express 2012 for Web stops working after adding scaffolded item with Entity Framework (such as Web API 2 Controller with actions, using Entity Framework), it is possible that Visual Studio Express failed to load the native image of an assembly dependent on System.Web.Extensions.

To correct this problem, configure Visual Studio Express to work with the MSIL image of System.Web.Extensions:

1. Open Command Prompt in the Administrator mode.
2. Go to %ProgramFiles%\Microsoft Visual Studio 11.0\Common7\IDE or %ProgramFiles(x86)%\Microsoft Visual Studio 11.0\Common7\IDE (for 64 bit Windows).
3. Open VWDExpress.exe.config in a text editor.
4. Add the following line under the <configuration>/<runtime> element:

```
<disableNativeImageLoad>
    <assemblyIdentity name="System.Web.Extensions" />
</disableNativeImageLoad>
```

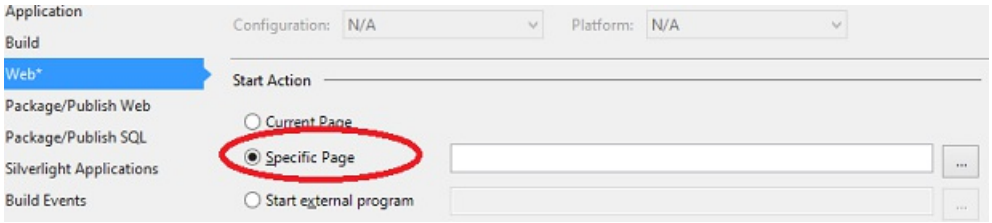
5. Restart Visual Studio Express 2012 for Web.

ASP.NET Razor 3

Viewing cshtml file with Browse With or F5 causes a server error

When you create an MVC 5 project in Visual Studio 2012 (or open in Visual Studio 2012 an MVC 5 project that was created in Visual Studio 2013) and attempt to view a cshtml file by using Browse With or F5, you will receive an error that states - **Server Error in '/' Application**. The server attempts to navigate to `http://localhost:XXXX/Views/../XXXX.cshtml`

To resolve this issue, change the **Start Action** setting in your project to **Specific Page**. You do not need to provide a value for the page.



After making this change, selecting F5 navigates to the root of your application (`http://localhost:XXXX`). This behavior is not the same as the behavior for MVC 5 projects in Visual Studio 2013, where the **Current Page** setting launches the open page.

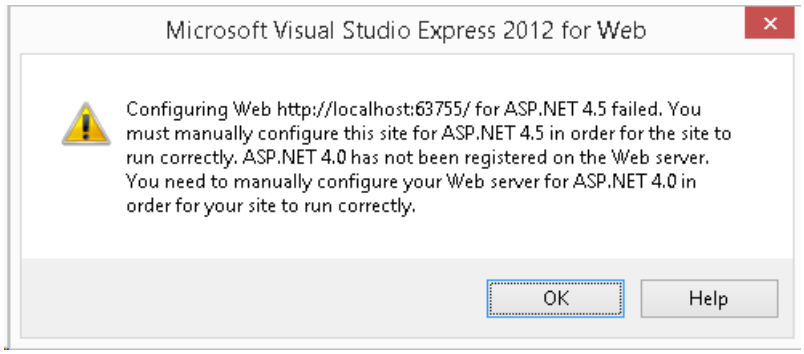
Url Rewrite and Tilde(~)

After upgrading to ASP.NET Razor 3 or ASP.NET MVC 5, the tilde(~) notation may no longer work correctly if you are using URL rewrites. The URL rewrite affects the tilde(~) notation in HTML elements such as <A/>, <SCRIPT/>, <LINK/>, and as a result the tilde no longer maps to the root directory.

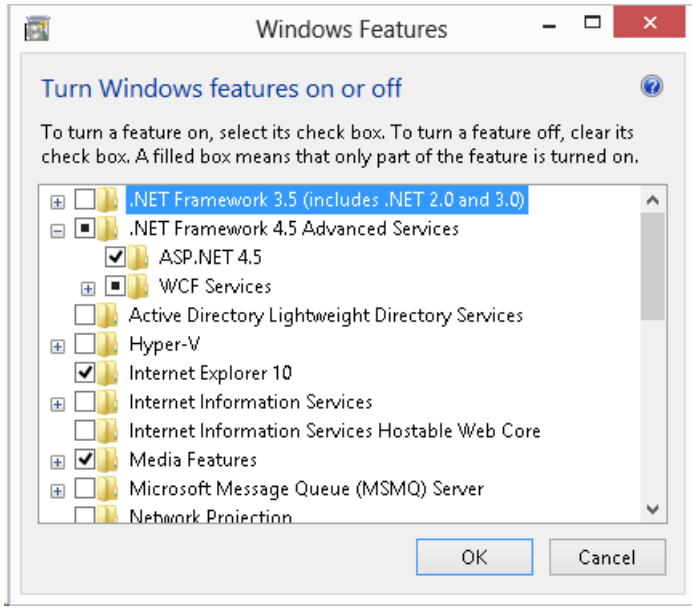
For example, if you rewrite requests for **asp.net/content** to **asp.net**, the href attribute in <A href="~/content/"/> resolves to **/content/content/** instead of **/**. To suppress this change, you can set the **IIS_WasUrlRewritten** context to false in each Web Page or in **Application_BeginRequest** in Global.asax.

Templates

When you create ASP.NET MVC projects with Visual Studio 2012 on Windows 8.1 or Windows Server 2012 R2, Visual Studio displays an error message that states "Configuring Web [url] for ASP.NET 4.5 failed."



You see this error because Visual Studio 2012 does not enable the ASP.NET 4.5 feature when it is installed on those releases of Windows. To enable ASP.NET 4.5, perform the steps described in Turn Windows features on or off.



Alternatively, you can enable ASP.NET 4.5 through the command line.

1. Open Command Prompt in the Administrator mode.
2. Run the following command to enable ASP.NET 4.5.

```
dism /Online /Enable-Feature /FeatureName:NetFx4Extended-ASPNET45 /Quiet /NoRestart
```

Visual Studio vNext Videos: HTML Editor Smart Tasks and Event Handler Generation

2/20/2020 • 2 minutes to read • Edit Online

by Mads Kristensen

In this video, Mads Kristensen demonstrates improvements in the Visual Studio vNext HTML editor which simplifies performing common tasks.

Watch video (1 minutes)

Visual Studio vNext Videos: CSS Editor Hierarchical Indentation

2/20/2020 • 2 minutes to read • Edit Online

by Mads Kristensen

In this video, Mads Kristensen demonstrates improvements in the Visual Studio vNext CSS editor which automatically indents style rules based on their selectors.

Watch video (1 minutes)

Visual Studio vNext Videos: CSS Snippets

2/20/2020 • 2 minutes to read • Edit Online

by Mads Kristensen

In this video, Mads Kristensen demonstrates an improvement in the Visual Studio vNext CSS editor which adds some useful snippets for CSS3 transforms, media queries, font embedding, and more.

Watch video (1 minutes)

Visual Studio vNext Videos: CSS Editor Color Picker

2/20/2020 • 2 minutes to read • Edit Online

by Mads Kristensen

In this video, Mads Kristensen demonstrates the Color Picker in the Visual Studio vNext CSS editor which gives quick access to all colors in the current spreadsheet and provides an advanced color selector with an eyedropper tool.

Watch video (1 minutes)

Visual Studio vNext Videos: Page Inspector - Introduction

2/20/2020 • 2 minutes to read • Edit Online

by Jorge Gabuardi Gonzalez

In this video, Jorge Gabuardi Gonzalez introduces the Page Inspector in Visual Studio vNext.

Watch video (1 minutes)

Visual Studio vNext Videos: Page Inspector - Decomposing your Web Application

2/20/2020 • 2 minutes to read • Edit Online

by Jorge Gabuardi Gonzalez

In this video, Jorge Gabuardi Gonzalez shows how to use the Page Inspector in Visual Studio vNext to troubleshoot HTML and styles in your ASP.NET applications.

Watch video (1 minutes)

Windows Azure Authentication

9/16/2020 • 6 minutes to read • Edit Online

by Rick Anderson

Microsoft ASP.NET tools for Windows Azure Active Directory makes it simple to enable authentication for web applications hosted on Windows Azure Web Sites. You can use Windows Azure Authentication to authenticate Office 365 users from your organization, corporate accounts synced from your on-premise Active Directory or users created in your own custom Windows Azure Active Directory domain. Enabling Windows Azure Authentication configures your application to authenticate users using a single Windows Azure Active Directory tenant.

The ASP.NET Windows Azure Authentication tool is not supported for web roles in a cloud service but we plan to do so in a future release. Windows Identity Foundation (WIF) is supported in Windows Azure web roles.

For details on how to setup synchronization between your on-premise Active Directory and your Windows Azure Active Directory tenant please see Use AD FS 2.0 to implement and manage single sign-on.

Windows Azure Active Directory is currently available as a free preview service.

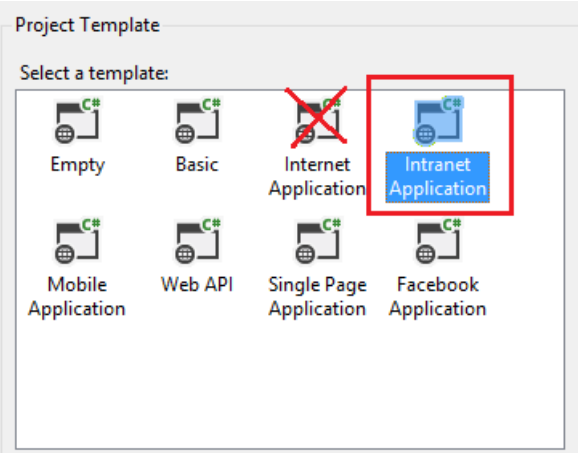
Requirements:

- Visual Studio 2012 or Visual Studio Express 2012
- Web Tools Extensions for Visual Studio 2012 or Web Tools Extensions for Visual Studio Express 2012
- Microsoft ASP.NET Tools for Windows Azure Active Directory – Visual Studio 2012 or Microsoft ASP.NET Tools for Windows Azure Active Directory – Visual Studio Express 2012 for Web

Create an ASP.NET Web Application with Visual Studio 2012

You can create any Web Application with Visual Studio 2012, this tutorial uses the ASP.NET MVC intranet template.

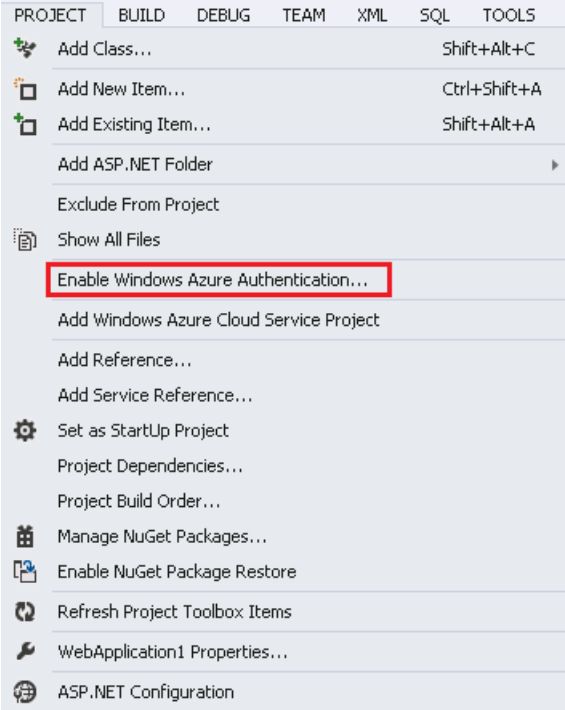
1. Create a new ASP.NET MVC 4 Intranet Application and accept all the defaults. (It must be an In **tra** net and not In **ter** net project).



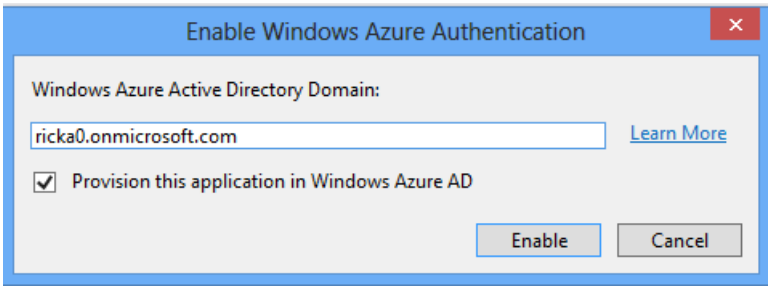
Enable Window Azure Authentication (When you are a Global Administrator of the Tenet)

If you do not have an existing Windows Azure Active Directory tenant (For example, through an existing Office 365 account) you can create a new tenant by signing up for a new Windows Azure Active Directory account.

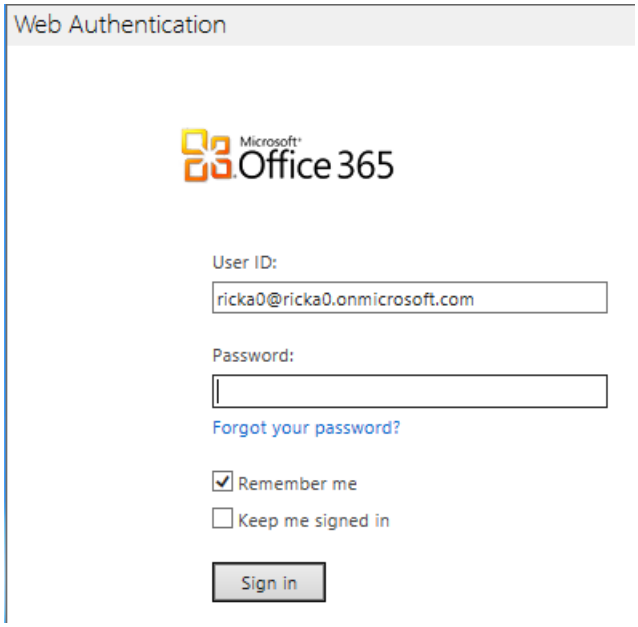
1. From the Project menu select **Enable Windows Azure Authentication**:

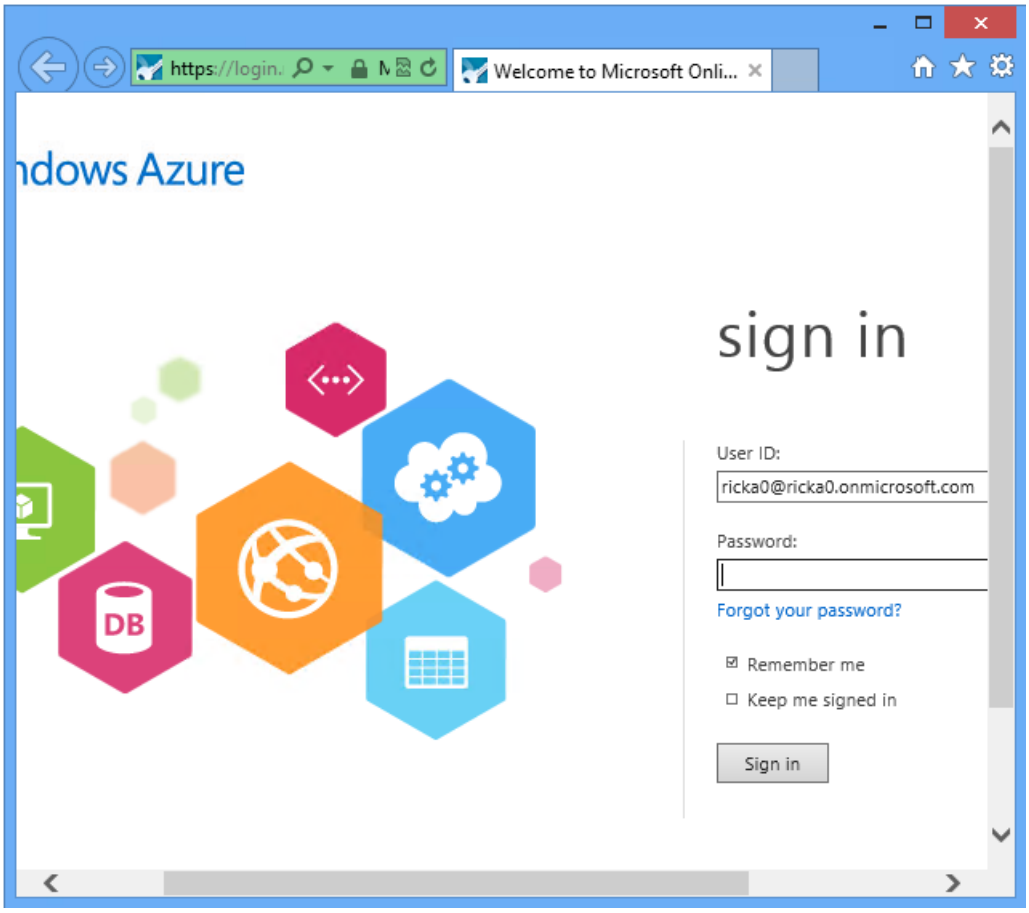


2. Enter the domain for your Windows Azure Active Directory tenant (for example, contoso.onmicrosoft.com) and click **Enable**:



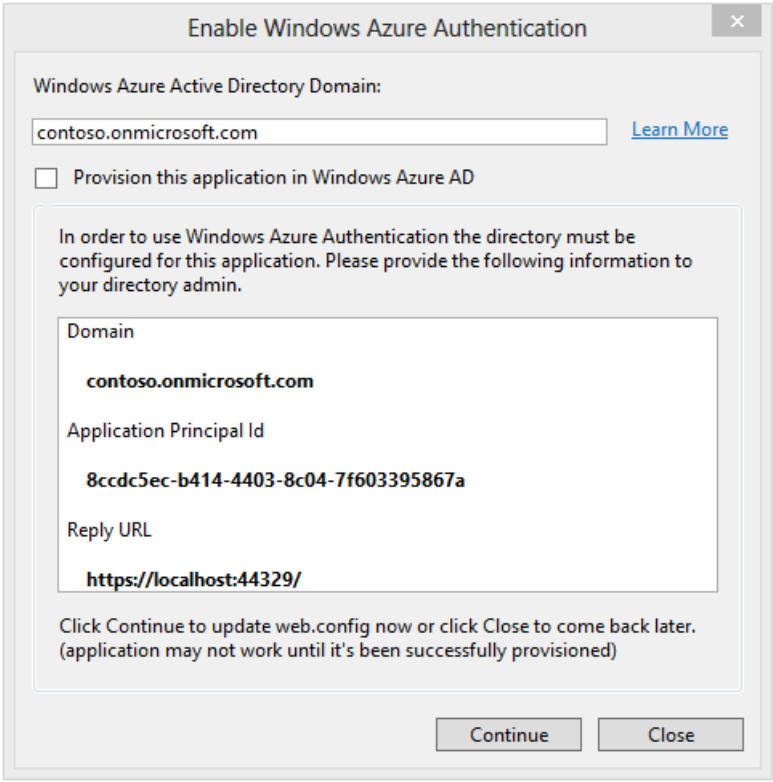
3. In the Web Authentication dialog sign in as an administrator for your Windows Azure Active Directory tenant:





Enable Window Azure by a non-administrator of the Tenet

If you do not have Global Administrator privilege for your Windows Azure Active Directory tenant, you can uncheck the checkbox for provisioning the application.

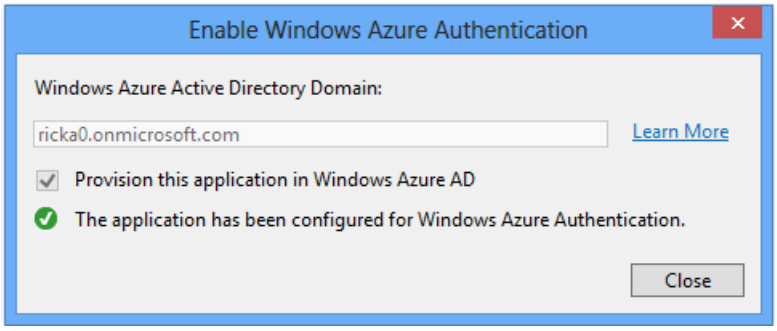


The dialog will display the **Domain**, **Application Principal Id** and **Reply URL** which are required for provisioning the application with an Azure Active Directory tenet. You need to give this information to someone who has sufficient privilege to provision the application. SeeHow to implement single sign-on with Windows

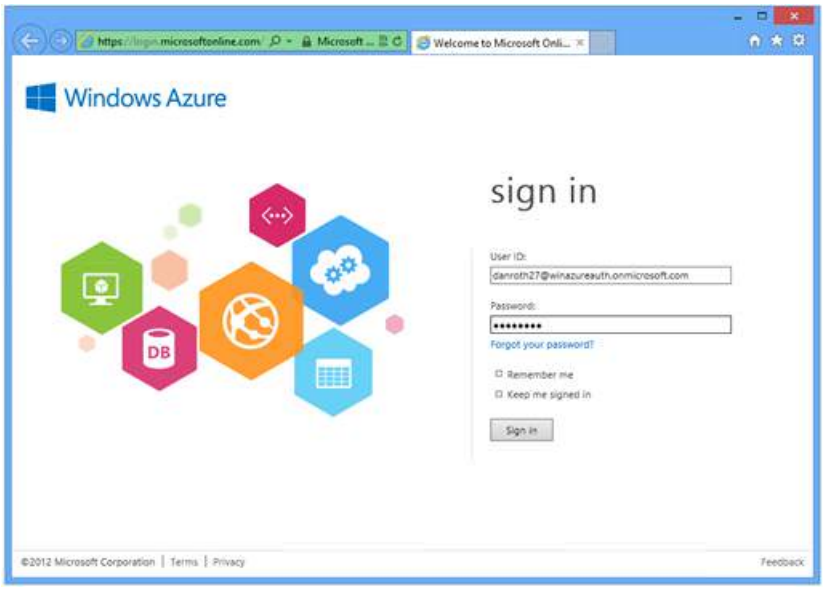
Azure Active Directory - ASP.NET Application for details on how to use cmdlet to create the service principal manually. Once the application has been successfully provisioned, you can click on **Continue to update web.config with the selected settings**. If you want to continue developing the application while waiting for provisioning to happen, you can click **Close to remember the settings in project file**. The next time you invoke Enable Windows Azure Authentication and uncheck the provisioning checkbox, you will see the same settings and you can click **Continue**, then click, **Apply these settings in web.config**.

1. Wait while your application is configured for Windows Azure Authentication and provisioned with Windows Azure Active Directory.

2. Once Windows Azure Authentication has been enabled for your application, click **Close:**

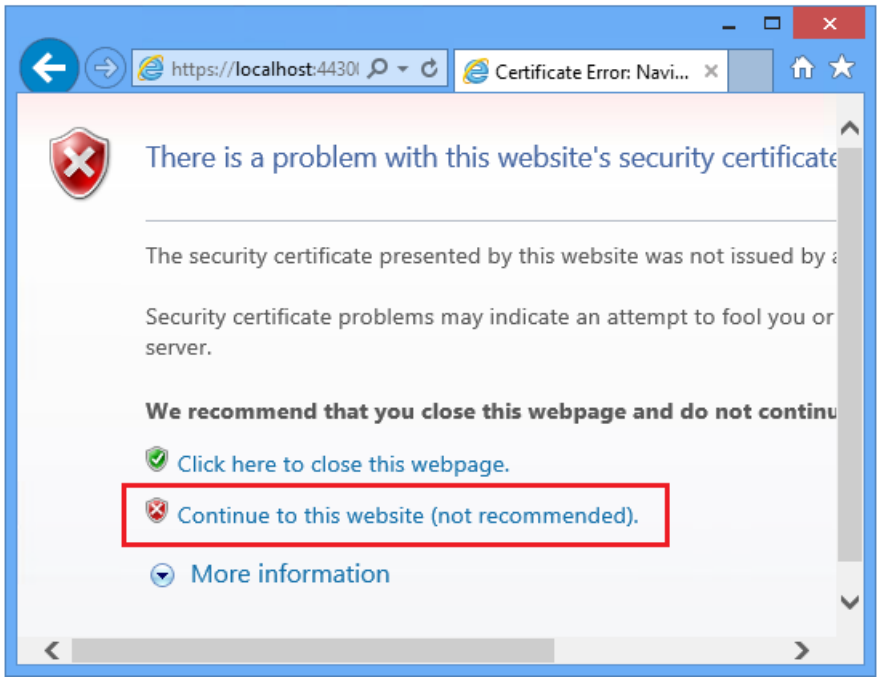


3. Hit F5 to run your application. You should automatically get redirected to login page. Use the directory tenet user credentials to login to the application..

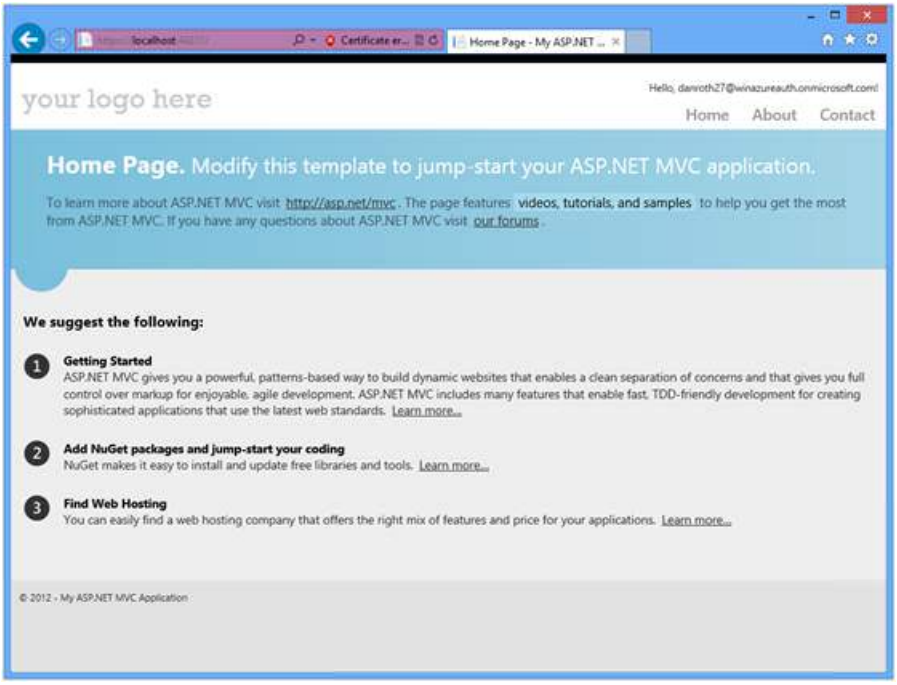


4. Because your application is currently using a self-signed test certificate you will receive a warning from the browser that the certificate was not issued by a trusted certificate authority.

   This warning can be safely ignored during local development by clicking **Continue to this website:**



5. You have now successfully logged in to your application using Windows Azure Authentication!



Enabling Windows Azure authentication makes the following changes to your application:

- An Anti-Cross-Site Request Forgery (CSRF) class (*App_Start\AntiXsrfConfig.cs*) is added to your project.
- The NuGet packages `System.IdentityModel.Tokens.ValidatingIssuerNameRegistry` is added to your project.
- Windows Identity Foundation settings in your application will be configured to accept security tokens from your Windows Azure Active Directory tenant. Click on the image below to see an expanded view of the changes made to the *Web.config* file.

```
9     <section name="entityFramework" type="System.Data.Entity.
10    <section name="system.identityModel" type="System.Identit
11    <section name="system.identityModel.services" type="Syste
12  </configSections>
18    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
19    <add key="ida:FederationMetadataLocation" value="https://
20    <add key="ida:ProviderSelection" value="productionSTS" />
21    <add key="enableSimpleMembership" value="false" />
22  </appSettings>
23  <location path="FederationMetadata">
24    <system.web>
25      <authorization>
26        <allow users="*" />
27      </authorization>
28    </system.web>
29  </location>
30  <system.web>
31    <authentication mode="None" />
32    <compilation debug="true" targetFramework="4.5" />
33    <httpRuntime targetFramework="4.5" requestValidationMode=
34    <!--Commented out by Windows Azure Authentication tool-->
35    <!--<authentication mode="Windows" />-->
61    </handlers>
62    <modules>
63      <add name="WSFederationAuthenticationModule" type="Sy...
64      <add name="SessionAuthenticationModule" type="System.
65    </modules>
66  </system.webServer>
67  <runtime>
92    </defaultConnectionFactory>
93  </entityFramework>
94  <system.identityModel>
95    <identityConfiguration>
96      <audienceUris>
97        <add value="spn:b63b9031-7d95-4bd0-b4ea-edadd1
```

- A service principal for your application in your Windows Azure Active Directory tenant will be provisioned.
- HTTPS is enabled.

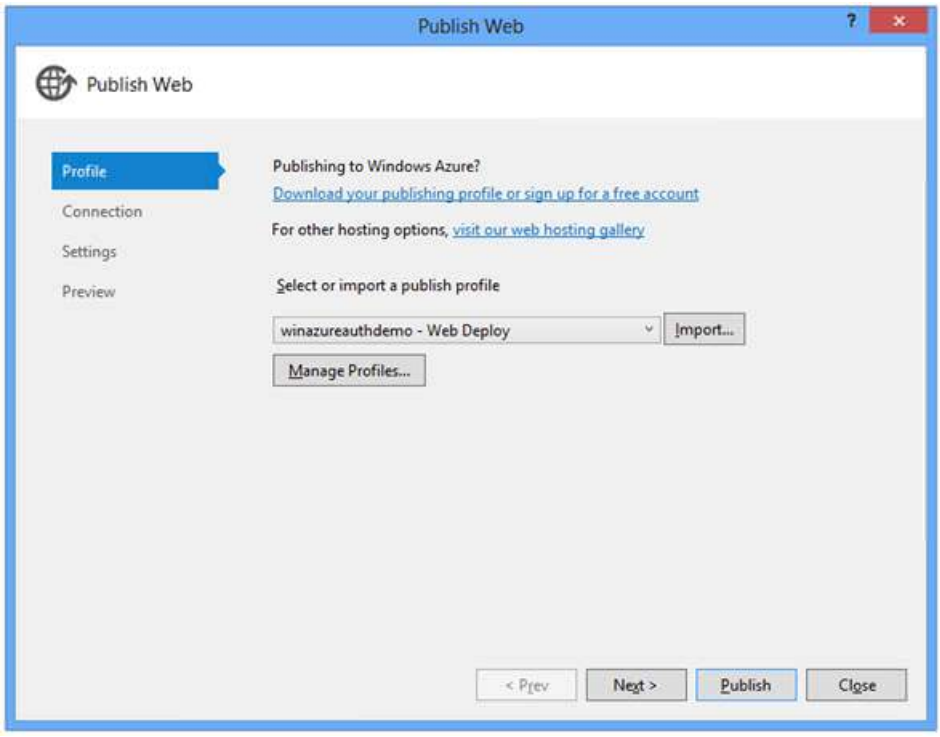
Deploy the application to Windows Azure

For complete instructions, see Deploying an ASP.NET Web Application to a Windows Azure Web Site.

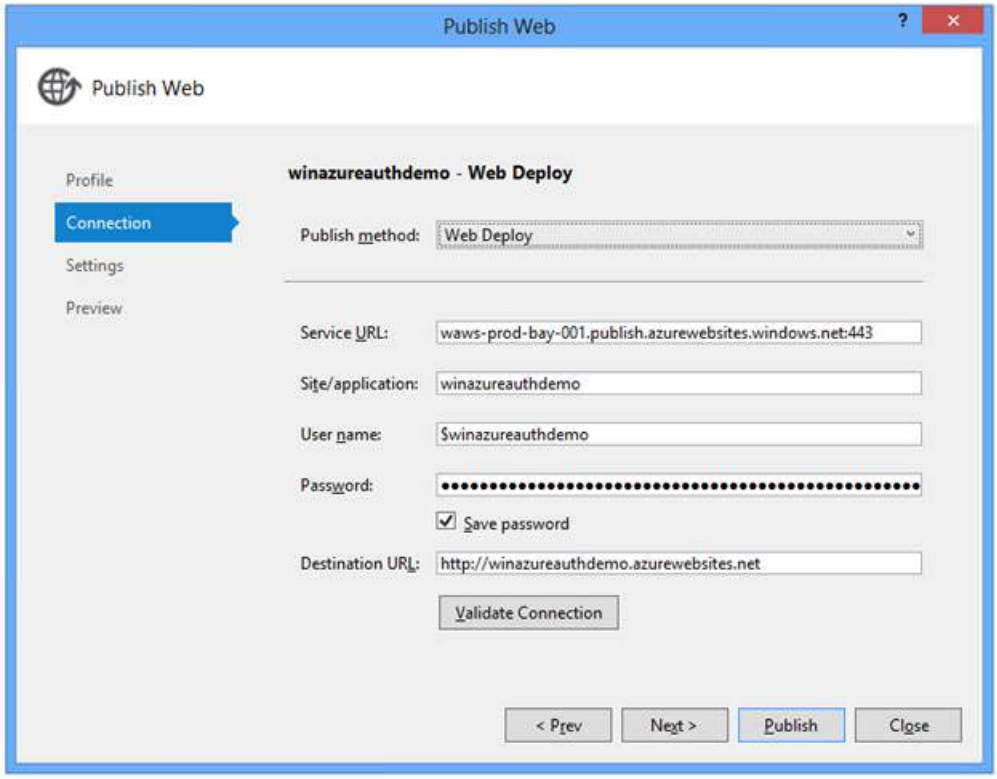
To publish an application using Windows Azure Authentication to an Azure Web Site:

1. Right click on your application and select **Publish:**

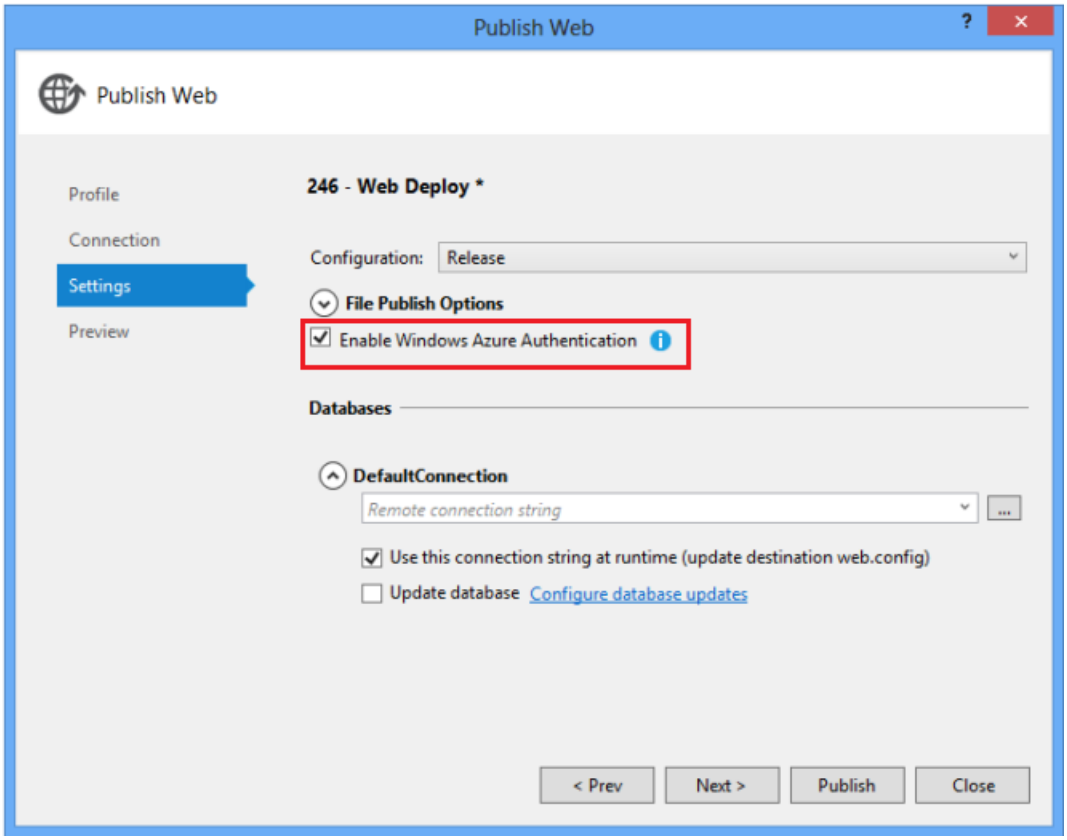
2. From the Publish Web dialog download and import a publishing profile for your Azure Web Site.



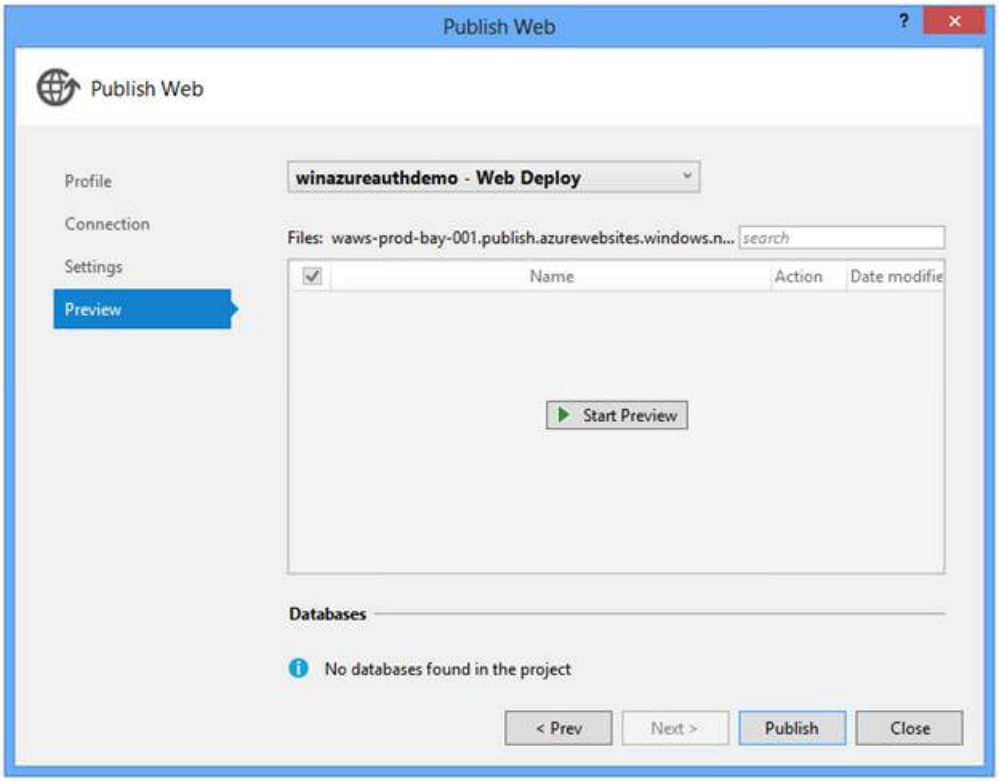
3. The **Connection** tab shows the **Destination URL** (the public facing URL for your application). Click **Validate Connection** to test your connection:



4. If you have published to this Azure Web Site previously consider checking the **Remove additional files at destination** setting to ensure your application publishes cleanly. Notice the **Enable Windows Azure Authentication** check box is selected.

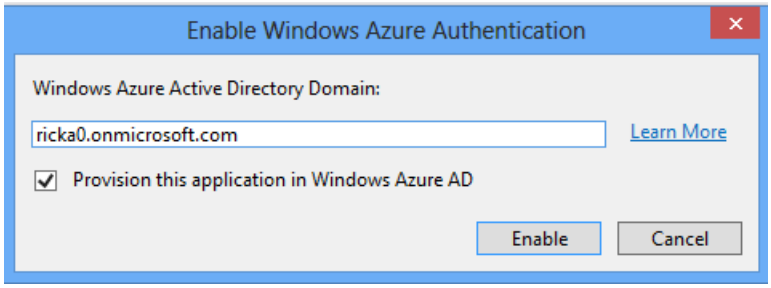


5. Optional: On the **Preview** tab click **Start Preview** to see the files deployed.

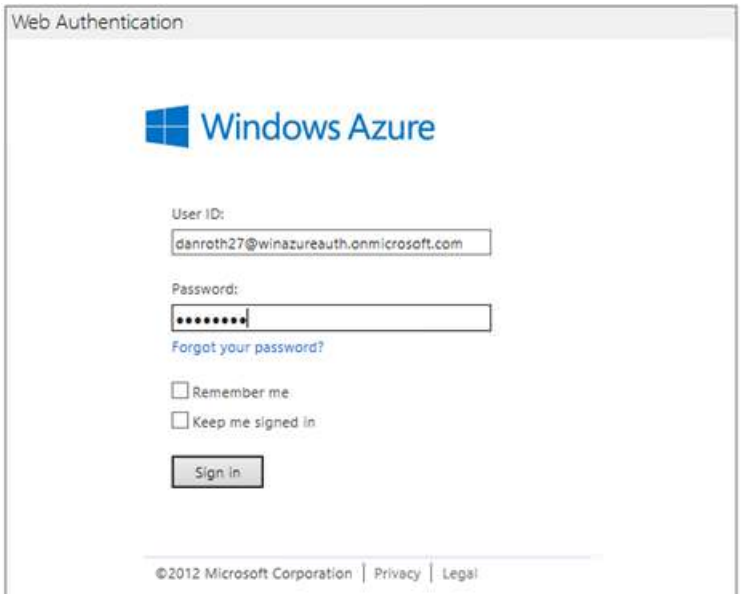


6. Click **Publish**.

   You will be prompted to enable Windows Azure Authentication for the target host. Click **Enable** to continue:



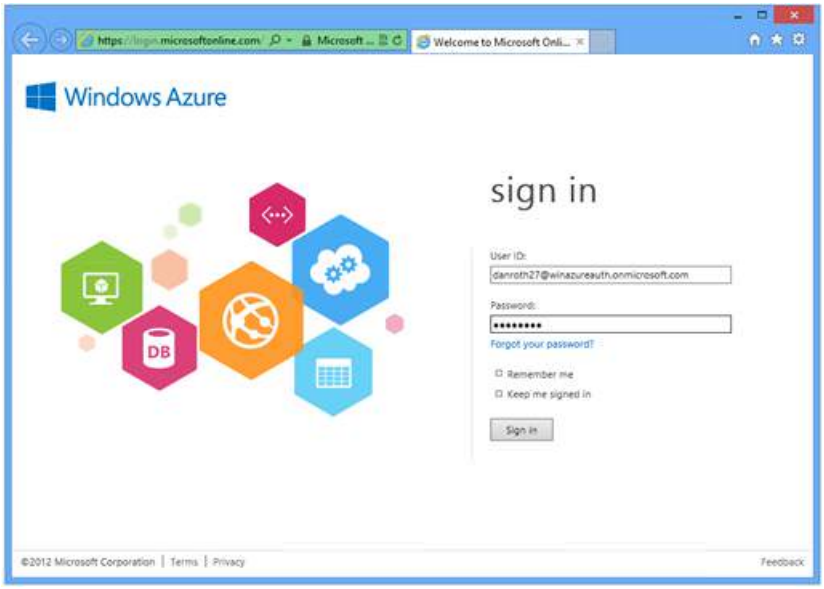
7. Enter your administrator credentials for your Windows Azure Active Directory tenant:



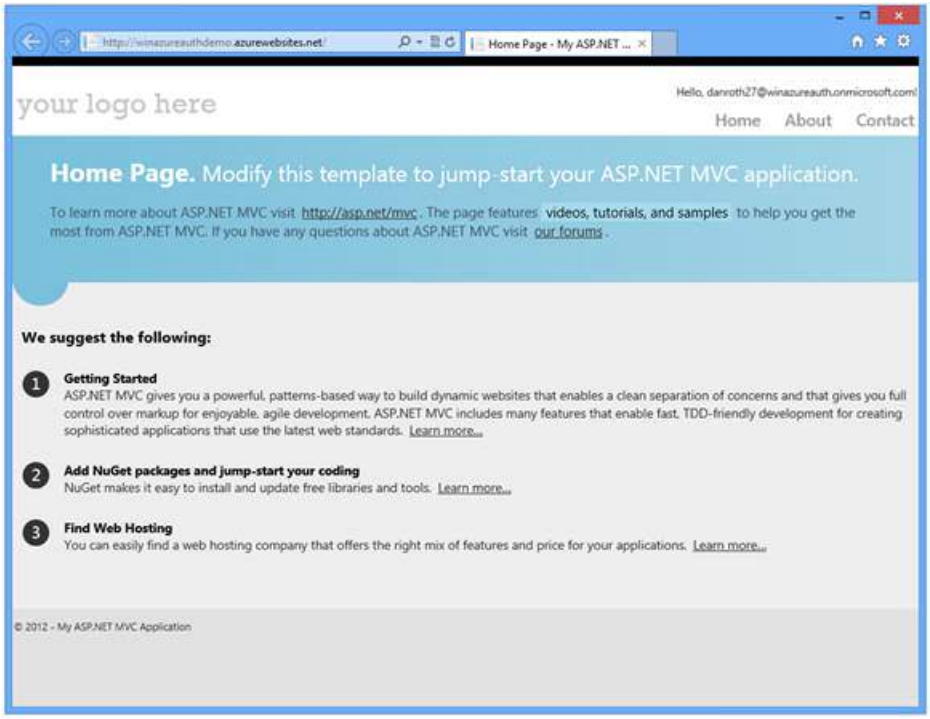
8. Once your application has been successfully published, a browser will open to the published web site.

> **NOTE**
>
> It may take up to five minutes (typically must less) for your application to be fully provisioned with Windows Azure Active Directory after enabling Windows Azure Authentication for the target host. When you first run your application if you receive error ACS50001: Relying party with name '[realm]' was not found, then wait a few minutes and try running the application again.

9. When prompted, log in as a user in your directory:



10. You have now successfully logged into your Azure hosted application using Windows Azure Authentication.



Known Issues

Role-based authorization fails when using Windows Azure Authentication

Windows Azure Authentication does not currently provide the necessary role claim so that role-based authorization can be performed. The role of the authenticated user must be manually retrieved from Windows Azure Active Directory.

Browsing to an application with Windows Azure Authentication results in the error "ACS20016 The domain of the logged in user (live.com) does not match any allowed domain of this STS"

If you are already logged in to a Microsoft Account (for example hotmail.com, live.com, outlook.com) and you attempt to access an application that has enabled Windows Azure Authentication you may get a 400 error response because the domain of your Microsoft Account is not recognized by Windows Azure Active Directory. To log into the application, log out from your Microsoft Account first.

Logging into an application with Windows Azure Authentication enabled and a X509CertificateValidationMode other than None results in certificate validation errors for the accounts.accesscontrol.windows.net certificate

Certificate validation is not required and should be left disabled. The thumbprint of the issuer certificate is validated by the WSFederationAuthenticationModule.

When attempting to enable Windows Azure Authentication the Web Authentication dialog shows the error "ACS20016: The domain of the logged in user (contoso.onmicrosoft.com) does not match any allowed domain of this STS."

You may see this error when you have previously successfully logged in using a different Windows Azure Active Directory account from within the same Visual Studio process. Log out from the specified account or restart Visual Studio. If you previously logged in and selected the option to "Keep me signed in" then you may need to clear your browser cookies.

ACS20012: The request is not a valid WS-Federation protocol message

This can happen if you are already logged in with some other Microsoft ID to one of the Azure services. Use Private browser window like InPrivate in IE or Incognito in Chrome or clear all the cookies.

Additional Resources

- Microsoft ASP.NET Tools for Windows Azure Active Directory – Visual Studio 2012 – Vittorio Bertocci
- Windows Azure Features: Identity
- TechNet: Windows Azure Active Directory

- Windows Azure Active Directory: Develop Apps for your organization
- Windows Azure Active Directory: Develop Apps for multiple organizations
- How to implement single sign-on with Windows Azure Active Directory
- Single Sign-On with Windows Azure Active Directory: a Deep Dive – Vittorio Bertocci
- Use AD FS 2.0 to implement and manage single sign-on

ASP.NET and Web Tools 2012.2 Release Notes

2/20/2020 • 11 minutes to read • Edit Online

This document describes the release of ASP.NET and Web Tools 2012.2. It is an update to Visual Studio Web Tooling and ASP.NET.

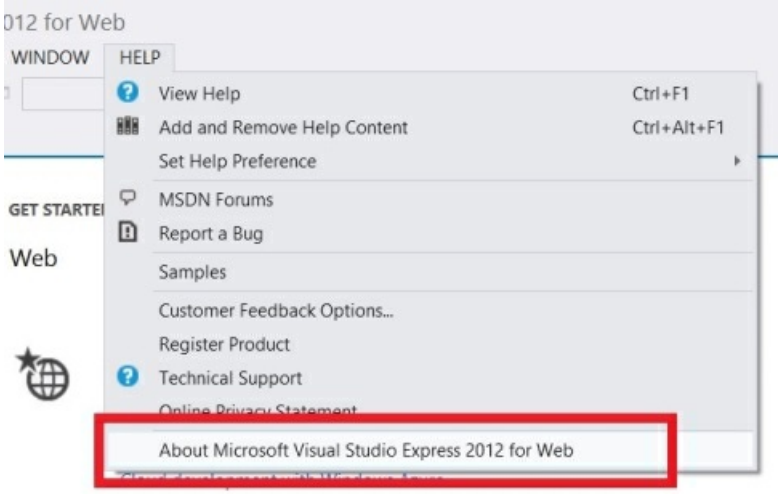
- Installation Notes
- Documentation
- Support
- Software Requirements
- New Features in ASP.NET and Web Tools 2012.2
  - Tooling
  - Web Publishing
  - ASP.NET MVC Templates
  - ASP.NET Web API
  - ASP.NET SignalR
  - ASP.NET Friendly URLs
- Known Issues and Breaking Changes

Installation Notes

ASP.NET and Web Tools 2012.2 for Visual Studio 2012 can be installed using Web Platform installer. This is an update to Visual Studio 2012 or Visual Studio Express 2012 for Web, which is required. If you do not have Visual Studio installed, Visual Studio Express 2012 for Web will be installed.

You can also install ASP.NET and Web Tools 2012.2 manually. You must have Visual Studio 2012 or Visual Studio Express 2012 for Web installed. Then use the following instructions:

1. Download ASP.NET and Web Frameworks 2012.2 installer from Download Center.
2. When prompted click Run. You can also save the file to run it later.
3. Verify the version of Visual Studio you will update. You can do this by launching the Visual Studio you wish to update. Then click the Help menu item.



4. If you see the menu item "About Microsoft Visual Studio 2012 for Web" then download Web Developer Tools 2012.2 - Visual Studio Express 2012 for Web. Otherwise download Web Developer Tools 2012.2 - Visual Studio 2012.
5. When prompted click Run. You can also save the file to run it later.

> **NOTE**
> ASP.NET and Web Tools 2012.2 release does not include SQL Server Data Tools. SQL Server and Windows Azure SQL Databases provides a richer set of database tooling including offline project-backed development, schema comparison and enhanced database deployment capabilities. For more information or to install SQL Server Data Tools visit https://go.microsoft.com/fwlink/?LinkID=237127.

Documentation

Tutorials and other information about ASP.NET and Web Tools 2012.2 are available from ASP.NET web site (https://www.asp.net).

Support

ASP.NET and Web Tools 2012.2 is officially released and supported. You can use your normal support channel. You can also post questions to the ASP.NET forums (https://forums.asp.net/), where members of the ASP.NET community are frequently able to provide informal support.

Software Requirements

The ASP.NET and Web Tools 2012.2 requires Visual Studio 2012 or Visual Studio Express 2012 for Web.

New Features in ASP.NET and Web Tools 2012.2

This section describes features that have been introduced in the ASP.NET and Web Tools 2012.2 release.

Tooling

- Page Inspector
  - Support JavaScript selection mapping allowing Page Inspector to map items that were dynamically added to the page back to the corresponding JavaScript code.
  - The ability to see CSS updates in real-time.
  - For more information, read CSS Auto-Sync and JavaScript Selection Mapping in Page Inspector.
- Editor

  - Support syntax highlighting for CoffeeScript, Mustache, Handlebars, and JsRender.
  - The HTML editor provides Intellisense for Knockout bindings.
  - LESS editing and compiler support to enable building dynamic CSS using LESS.
  - Paste JSON as a .NET class. Using this Special Paste command to paste JSON into a C# or VB.NET code file, and Visual Studio will automatically generate .NET classes inferred from the JSON.
- Mobile Emulator support adds extensibility hooks so that third-party emulators can be installed as a VSIX. The installed emulators will show up in the F5 dropdown, so that developers can preview their websites on a variety of mobile devices. Read more about this feature in Scott Hanselman's blog entry on the new BrowserStack integration with Visual Studio.

Web Publishing

- Web site projects now have the same publishing experience as Web Application projects including publishing to Windows Azure Web Sites.
- Selective publish – for one or more files you can perform the following actions (after publishing to a Web Deploy endpoint):
  - Publish selected files.
  - See the difference between a local file and a remote file.
  - Update the local file with the remote file or update the remote file with the local file.

ASP.NET MVC Templates

- The new Facebook Application template makes writing Facebook Canvas applications easy. In a few simple steps, you can create a Facebook application that gets data from a logged in user and integrates with their friends. The template includes a new library to take care of all the plumbing involved in building a Facebook app, including authentication, permissions, accessing Facebook data and more. For more information on using the Facebook Application template see https://go.microsoft.com/fwlink/?LinkID=269921.
- A new Single Page Application MVC template allows developers to build interactive client-side web apps using HTML 5, CSS 3, and the popular Knockout and jQuery JavaScript libraries, on top of ASP.NET Web API. The template includes a "todo" list application that demonstrates common practices for building a JavaScript HTML5 application that uses a RESTful server API. You can read more at https://www.asp.net/single-page-application.
- You can now create a VSIX that adds new templates to the ASP.NET MVC New Project dialog. Learn how here: https://go.microsoft.com/fwlink/?LinkId=275019
- FixedDisplayModes package – MVC project templates have been updated to include the new 'FixedDisplayModes' NuGet package, which contains a workaround for a bug in MVC 4. For more information on the fix contained in the package, refer to this blog post (https://blogs.msdn.com/b/rickandy/archive/2012/09/17/asp-net-mvc-4-mobile-caching-bug-fixed.aspx) from the MVC team.

ASP.NET Web API

ASP.NET Web API has been enhanced with several new features:

- ASP.NET Web API OData
- ASP.NET Web API Tracing
- ASP.NET Web API Help Page

ASP.NET Web API OData

ASP.NET Web API OData gives you the flexibility you need to build OData endpoints with rich business logic over any data source. With ASP.NET Web API OData you control the amount of OData semantics that you want to expose. ASP.NET Web API OData is included with the ASP.NET MVC 4 project templates and is also available from NuGet (http://www.nuget.org/packages/microsoft.aspnet.webapi.odata).

ASP.NET Web API OData currently supports the following features:

- Enable OData query semantics by applying the [Queryable] attribute.
- Easily validate OData queries and restrict the set of supported query options, operators and functions.
- Parameter bind to ODataQueryOptions directly to get an abstract syntax tree representation of the query that can then be validated and applied to an IQueryable or IEnumerable.
- Enable service-driven paging and next page link generation by specifying result limits on [Queryable] attribute.
- Request an inlined count of the total number of matching resources using $inlinecount.
- Control null propagation.
- Any/All operators in $filter.
- Infer an entity data model by convention or explicitly customize a model in a manner similar to Entity Framework Code-First.
- Expose entity sets by deriving from EntitySetController.
- Simple, customizable conventions for exposing navigation properties, manipulating links and implementing OData actions.
- Simplified routing using the MapODataRoute extension method.
- Support for versioning by exposing multiple EDM models.
- Expose service document and $metadata so you can generate clients (.NET, Windows Phone, Windows Store, etc.) for your Web API.
- Support for the OData Atom, JSON, and JSON verbose formats.
- Create, update, partially update (PATCH) and delete entities.
- Query and manipulate relationships between entities.
- Create relationship links that wire up to your routes.
- Complex types.
- Entity Type Inheritance.
- Collection properties.
- Enums.
- OData actions.
- Built upon the same foundation as WCF Data Services, namely ODataLib (http://www.nuget.org/packages/microsoft.data.odata).

For more information on ASP.NET Web API OData see https://go.microsoft.com/fwlink/?LinkId=271141.

ASP.NET Web API Tracing

ASP.NET Web API Tracing integrates tracing data from your web APIs with .NET Tracing. It is now enabled by default in the Web API project template. Tracing data for your web APIs is sent to the Output window and is made available through IntelliTrace. ASP.NET Web API Tracing enables you to trace information about your Web API when hosted on Windows Azure through integration with Windows Azure Diagnostics. You can also install and enable ASP.NET Web API Tracing in any application using the ASP.NET Web API Tracing NuGet package (http://www.nuget.org/packages/microsoft.aspnet.webapi.tracing).

For more information on configuring and using ASP.NET Web API Tracing see https://go.microsoft.com/fwlink/?LinkID=269874.

ASP.NET Web API Help Page

The ASP.NET Web API Help Page is now included by default in the Web API project template. The ASP.NET Web API Help Page automatically generates documentation for web APIs including the HTTP endpoints, the supported HTTP methods, parameters and example request and response message payloads. Documentation is automatically pulled from comments in your code. You can also add the ASP.NET Web API Help Page to any application using the ASP.NET Web API Help Page NuGet package (http://www.nuget.org/packages/microsoft.aspnet.webapi.helppage).

For more information on setting up and customizing the ASP.NET Web API Help Page see https://go.microsoft.com/fwlink/?LinkId=271140.

ASP.NET SignalR

ASP.NET SignalR makes it simple to add real-time web capabilities to your ASP.NET application, using WebSockets if available and automatically falling back to other techniques when it isn't.

For more information on using ASP.NET SignalR see https://go.microsoft.com/fwlink/?LinkId=271271.

ASP.NET Friendly URLs

ASP.NET FriendlyURLs makes it very easy for web forms developers to generate cleaner looking URLs(without the .aspx extension). It requires little to no configuration and can be used with existing ASP.NET v4.0 applications. The FriendlyURLs feature also makes it easier for developers to add mobile support to their applications, by supporting switching between desktop and mobile views.

For more information on installing and using ASP.NET Friendly URLs see http://www.hanselman.com/blog/IntroducingASPNETFriendlyUrlsCleanerURLsEasierRoutingAndMobileViewsForASPNETWebForms.aspx.

Known Issues and Breaking Changes

This section describes known issues and breaking changes that are in the ASP.NET and Web Tools 2012.2 release.

Installation Issues

Out of order installs of Visual Studio 2012

Installing an additional SKU of Visual Studio 2012 after installing the ASP.NET and Web Tools 2012.2 will require a repair operation. Consider the following sequence:

1. Install Visual Studio 2012 Express for Web
2. Install ASP.NET and Web Tools 2012.2
3. Install Visual Studio 2012 Professional, Premium or Ultimate

Step 2 would only result in installing updates for Express for Web. To ensure that the additional SKU installed during step 3 contains the update you will need to repair the ASP.NET and Web Tools 2012.2 to install the updates for the last SKU installed. This also applies if the SKUs in Step 1 and 3 are reversed.

Installing Microsoft ASP.NET and Web Tools 2012.2 when Visual Studio is open

If VS is open during installation of Microsoft ASP.NET and Web Tools 2012.2, Visual Studio might end up in a bad state. It is recommended that users close all instances of Visual Studio before proceeding with install.

Canceling ASP.NET and Web Tools 2012.2 setup in the middle of installation

Canceling ASP.NET and Web Tools 2012.2 setup in the middle of installation will leave Visual Studio in a bad state. To address this problem follow these steps:

- Go to Add Remove Programs
- Uninstall Microsoft ASP.NET and Web Tools 2012.2, if present.
- Reinstall Microsoft ASP.NET and Web Tools 2012.2

After uninstalling ASP.NET and Web Tools 2012.2 the ASP.NET MVC 4 templates and Razor v2 Web Site templates are missing

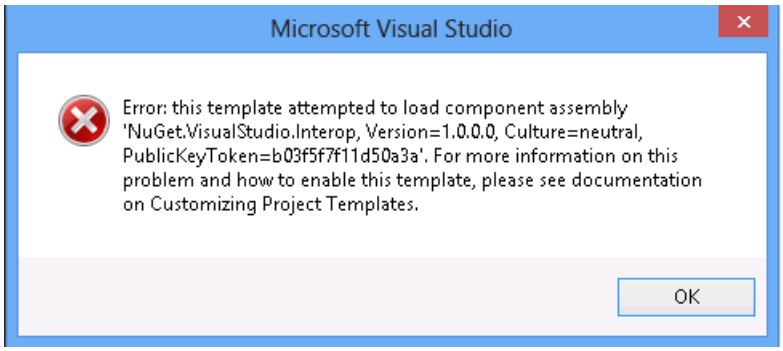
Uninstalling ASP.NET and Web Tools 2012.2 will also uninstall all of ASP.NET MVC 4 and Razor v2 Web Site templates from Visual Studio 2012.

The workaround is to repair your Visual Studio 2012 installation to reinstall ASP.NET MVC 4 and Razor v2 Web Site templates.

Tooling Issues

NuGet error reported during project creation

After installing ASP.NET and Web Tools 2012.2 you may see the following error when creating an MVC 4 project



The ASP.NET and Web Tools 2012.2 ships NuGet 2.1 and will upgrade the extension in Visual Studio 2012. In some cases, the VSIX installer will fail to correctly update the VSIX. The following steps will allow you to address this problem:

1. Start Visual Studio 2012 as an Administrator
2. Go to Tools->Extensions and Updates and uninstall NuGet.
3. Close Visual Studio
4. Navigate to the ASP.NET and Web Tools 2012.2 installation folder:
   a. For Visual Studio 2012: **Program Files\Microsoft ASP.NET\ASP.NET Web Stack\Visual Studio 2012**
   b. For Visual Studio 2012 Express for Web: **Program Files\Microsoft ASP.NET\ASP.NET Web Stack\Visual Studio Express 2012 for Web**
5. Double click on the NuGet.Tools.vsix to reinstall NuGet

Web API Issues

Parsing issues in $filter and DateTime literals

The OData URI parser fails to parse partial datetime literals properly. For example, $filter=start eq datetime'2012-12-31T12:00' fails to parse properly. A workaround is to use the full literal, $filter=start eq datetime'2012-12-31T12:00:00'.

OData doesn't support case-insensitive property names.

OData doesn't support case-insensitive property names in OData queries and odata path. See workitems:

- http://aspnetwebstack.codeplex.com/workitem/366
- http://aspnetwebstack.codeplex.com/workitem/704

If users have different casing on javascript client side and server side, they probably will encounter this issue. This issue is by design in odata protocol. However, many users reports this issue. To work around it, users have to correct their cases in URL.

Default OData routing conventions doesn't support POST/PUT on navigation property.

Default OData routing conventions doesn't support POST/PUT on navigation property. See workitem http://aspnetwebstack.codeplex.com/workitem/366. We are missing this commonly used convention in default conventions.

To work around it, users need to extend new routing convention to support it.

Facebook Template Issues

Facebook Application template only works using .NET 4.5

You must select .NET 4.5 in the framework dropdown list in the New Project dialog to see the Facebook Application template in ASP.NET MVC 4.

Real-time Update Controller

The Facebook Application template allows user easily create a Web API Controller to handle real-time updates from

Facebook. If your development machine is behind NAT, your Controller may not work without further network configuration. See here for details: http://facebook.stackoverflow.com/questions/5259467/can-a-computer-behind-a-nat-router-receive-realtime-updates-from-facebook

Query string values conflict with Facebook OAuth parameters

The following fields conflict with Facebook OAuth dialog's call back URL. Do not add your own query string values with the following names: code, error, error_description, error_reason.

Using Page Inspector with Facebook Template

You can't use the Page Inspector feature in Visual Studio 2012 while debugging your Facebook Application. The Page Inspector does not currently support iframes.

Single Page Application Template Issues

With JQuery 1.9/Knockout 2.2.1 update, when running default MVC SPA project, new todo item edit enter focus event is not handled properly.

With JQuery 1.9/Knockout 2.2.1 update, when running default MVC SPA project, new todo item edit enter no longer focus back to the new todo item edit box after entering the new todo item to the todo list.

To workaround reference http://knockoutjs.com/documentation/hasfocus-binding.html, and make similar fix to the following sample code:

File todo.model.js
function todolist(data), add following:
self.isSelected = ko.observable(false);

function todoList.prototype.addTodo, add the following blacked text:
self.isSelected(true);
self.newTodoTitle("");

File index.cshtml, add the following blacked text:
<form data-bind="submit: addTodo">
<input class="addTodo" type="text" data-bind="value: newTodoTitle, placeholder: 'Type here to add', blurOnEnter: true, **hasfocus: isSelected**, event: { blur: addTodo }" />
</form>

Entity Framework

2/20/2020 • 2 minutes to read • Edit Online

Entity Framework (EF) is an object-relational mapper that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write.

- Get it: Add this to your project and start taking advantage of this powerful O/RM.
- Get Started: Visit the guide for videos and tutorials that will help you make the most of your data.
- Get Help: Find out how to get help from the experts and contribute to the data community on Stack Overflow.
- Contribute: Find out how you can help make Entity Framework, an open development model, even better by visiting GitHub site.

Use Entity Framework with ASP.NET

- Web Forms Tutorials
- MVC Tutorials
- Sample Application

ASP.NET Identity

2/20/2020 • 2 minutes to read • Edit Online

- Overview
  - Getting Started with ASP.NET Identity
    - Introduction to ASP.NET Identity
    - ASP.NET Identity Recommended Resources
    - Adding ASP.NET Identity to an Empty or Existing Web Forms Project
    - Developing ASP.NET Apps with Azure Active Directory
    - ASP.NET Identity: Using MySQL Storage with an EntityFramework MySQL Provider (C#)
  - Features & API
    - Best practices for deploying passwords and other sensitive data to ASP.NET and Azure App Service
    - Account Confirmation and Password Recovery with ASP.NET Identity (C#)
    - Two-factor authentication using SMS and email with ASP.NET Identity
  - Extensibility
    - Overview of Custom Storage Providers for ASP.NET Identity
    - Implementing a Custom MySQL ASP.NET Identity Storage Provider
    - Change Primary Key for Users in ASP.NET Identity
  - Migrations
    - Migrating an Existing Website from SQL Membership to ASP.NET Identity
    - Migrating Universal Provider Data for Membership and User Profiles to ASP.NET Identity (C#)

ASP.NET Identity Overview

2/20/2020 • 2 minutes to read • Edit Online

- Getting Started with ASP.NET Identity
  - Introduction to ASP.NET Identity
  - ASP.NET Identity Recommended Resources
  - Adding ASP.NET Identity to an Empty or Existing Web Forms Project
  - Developing ASP.NET Apps with Azure Active Directory
  - ASP.NET Identity: Using MySQL Storage with an EntityFramework MySQL Provider (C#)
- Features & API
  - Best practices for deploying passwords and other sensitive data to ASP.NET and Azure App Service
  - Account Confirmation and Password Recovery with ASP.NET Identity (C#)
  - Two-factor authentication using SMS and email with ASP.NET Identity
- Extensibility
  - Overview of Custom Storage Providers for ASP.NET Identity
  - Implementing a Custom MySQL ASP.NET Identity Storage Provider
  - Change Primary Key for Users in ASP.NET Identity
- Migrations
  - Migrating an Existing Website from SQL Membership to ASP.NET Identity
  - Migrating Universal Provider Data for Membership and User Profiles to ASP.NET Identity (C#)

Getting Started with ASP.NET Identity

2/20/2020 • 2 minutes to read • Edit Online

The ASP.NET Identity system is designed to replace the previous ASP.NET Membership and Simple Membership systems. It includes profile support, OAuth integration, works with OWIN, and is included with the ASP.NET templates shipped with Visual Studio 2013.

- Introduction to ASP.NET Identity
- ASP.NET Identity Recommended Resources
- Adding ASP.NET Identity to an Empty or Existing Web Forms Project
- Developing ASP.NET Apps with Azure Active Directory
- ASP.NET Identity: Using MySQL Storage with an EntityFramework MySQL Provider (C#)

Introduction to ASP.NET Identity

2/20/2020 • 10 minutes to read • Edit Online

The ASP.NET membership system was introduced with ASP.NET 2.0 back in 2005, and since then there have been many changes in the ways web applications typically handle authentication and authorization. ASP.NET Identity is a fresh look at what the membership system should be when you are building modern applications for the web, phone, or tablet.

Background: Membership in ASP.NET

ASP.NET Membership

ASP.NET Membership was designed to solve site membership requirements that were common in 2005, which involved Forms Authentication, and a SQL Server database for user names, passwords, and profile data. Today there is a much broader array of data storage options for web applications, and most developers want to enable their sites to use social identity providers for authentication and authorization functionality. The limitations of ASP.NET Membership's design make this transition difficult:

- The database schema was designed for SQL Server and you can't change it. You can add profile information, but the additional data is packed into a different table, which makes it difficult to access by any means except through the Profile Provider API.
- The provider system enables you to change the backing data store, but the system is designed around assumptions appropriate for a relational database. You can write a provider to store membership information in a non-relational storage mechanism, such as Azure Storage Tables, but then you have to work around the relational design by writing much code and a lot of `System.NotImplementedException` exceptions for methods that don't apply to NoSQL databases.
- Since the log-in/log-out functionality is based on Forms Authentication, the membership system can't use OWIN. OWIN includes middleware components for authentication, including support for log-ins using external identity providers (like Microsoft Accounts, Facebook, Google, Twitter), and log-ins using organizational accounts from on-premises Active Directory or Azure Active Directory. OWIN also includes support for OAuth 2.0, JWT and CORS.

ASP.NET Simple Membership

ASP.NET simple membership was developed as a membership system for ASP.NET Web Pages. It was released with WebMatrix and Visual Studio 2010 SP1. The goal of Simple Membership was to make it easy to add membership functionality to a Web Pages application.

Simple Membership did make it easier to customize user profile information, but it still shares the other problems with ASP.NET Membership, and it has some limitations:

- It was hard to persist membership system data in a non-relational store.
- You can't use it with OWIN.
- It doesn't work well with existing ASP.NET Membership providers, and it's not extensible.

ASP.NET Universal Providers

ASP.NET Universal Providers were developed to make it possible to persist membership information in Microsoft Azure SQL Database, and they also work with SQL Server Compact. The Universal Providers were built on Entity Framework Code First, which means that the Universal Providers can be used to persist data in any store supported by EF. With the Universal Providers, the database schema was cleaned up quite a lot as well.

The Universal Providers are built on the ASP.NET Membership infrastructure, so they still carry the same

limitations as the SqlMembership Provider. That is, they were designed for relational databases and it's hard to customize profile and user information. These providers also still use Forms Authentication for sign-in and sign-out functionality.

ASP.NET Identity

As the membership story in ASP.NET has evolved over the years, the ASP.NET team has learned a lot from feedback from customers.

The assumption that users will sign in by entering a user name and password that they have registered in your own application is no longer valid. The web has become more social. Users are interacting with each other in real time through social channels such as Facebook, Twitter, and other social web sites. Developers want users to be able to sign in with their social identities so that they can have a rich experience on their web sites. A modern membership system must enable redirection-based log-ins to authentication providers such as Facebook, Twitter, and others.

As web development evolved, so did the patterns of web development. Unit testing of application code became a core concern for application developers. In 2008 ASP.NET added a new framework based on the Model-View-Controller (MVC) pattern, in part to help developers build unit testable ASP.NET applications. Developers who wanted to unit test their application logic also wanted to be able to do that with the membership system.

Considering these changes in web application development, ASP.NET Identity was developed with the following goals:

- **One ASP.NET Identity system**
  - ASP.NET Identity can be used with all of the ASP.NET frameworks, such as ASP.NET MVC, Web Forms, Web Pages, Web API, and SignalR.
  - ASP.NET Identity can be used when you are building web, phone, store, or hybrid applications.
- **Ease of plugging in profile data about the user**
  - You have control over the schema of user and profile information. For example, you can easily enable the system to store birth dates entered by users when they register an account in your application.
- **Persistence control**
  - By default, the ASP.NET Identity system stores all the user information in a database. ASP.NET Identity uses Entity Framework Code First to implement all of its persistence mechanism.
  - Since you control the database schema, common tasks such as changing table names or changing the data type of primary keys is simple to do.
  - It's easy to plug in different storage mechanisms such as SharePoint, Azure Storage Table Service, NoSQL databases, etc., without having to throw `System.NotImplementedExceptions` exceptions.
- **Unit testability**
  - ASP.NET Identity makes the web application more unit testable. You can write unit tests for the parts of your application that use ASP.NET Identity.
- **Role provider**
  - There is a role provider which lets you restrict access to parts of your application by roles. You can easily create roles such as "Admin" and add users to roles.
- **Claims Based**
  - ASP.NET Identity supports claims-based authentication, where the user's identity is represented as a set of claims. Claims allow developers to be a lot more expressive in describing a user's identity than roles allow. Whereas role membership is just a boolean (member or non-member), a claim can include rich information about the user's identity and membership.

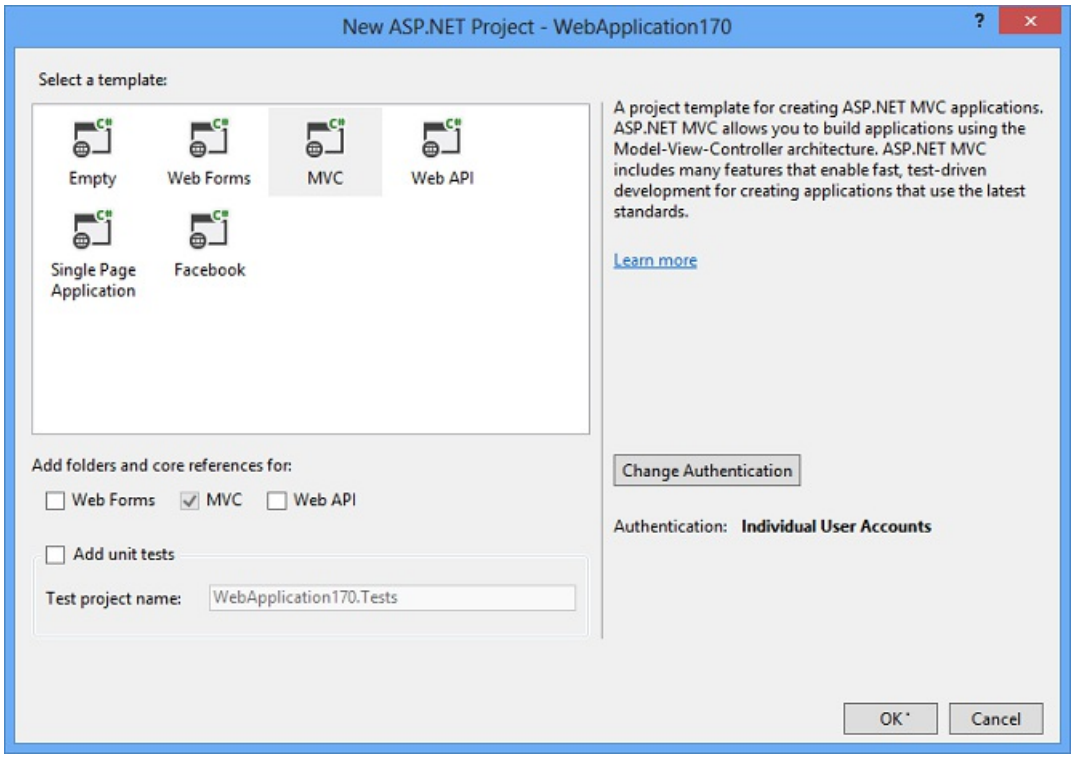
- **Social Login Providers**
  - You can easily add social log-ins such as Microsoft Account, Facebook, Twitter, Google, and others to your application, and store the user-specific data in your application.
- **OWIN Integration**
  - ASP.NET authentication is now based on OWIN middleware that can be used on any OWIN-based host. ASP.NET Identity does not have any dependency on System.Web. It is a fully compliant OWIN framework and can be used in any OWIN hosted application.
  - ASP.NET Identity uses OWIN Authentication for log-in/log-out of users in the web site. This means that instead of using FormsAuthentication to generate the cookie, the application uses OWIN CookieAuthentication to do that.
- **NuGet package**
  - ASP.NET Identity is redistributed as a NuGet package which is installed in the ASP.NET MVC, Web Forms and Web API templates that ship with Visual Studio 2017. You can download this NuGet package from the NuGet gallery.
  - Releasing ASP.NET Identity as a NuGet package makes it easier for the ASP.NET team to iterate on new features and bug fixes, and deliver these to developers in an agile manner.

Get started with ASP.NET Identity

ASP.NET Identity is used in the Visual Studio 2017 project templates for ASP.NET MVC, Web Forms, Web API and SPA. In this walkthrough, we'll illustrate how the project templates use ASP.NET Identity to add functionality to register, sign in and sign out a user.

ASP.NET Identity is implemented using the following procedure. The purpose of this article is to give you a high level overview of ASP.NET Identity; you can follow it step by step or just read the details. For more detailed instructions on creating apps using ASP.NET Identity, including using the new API to add users, roles and profile information, see the Next Steps section at the end of this article.

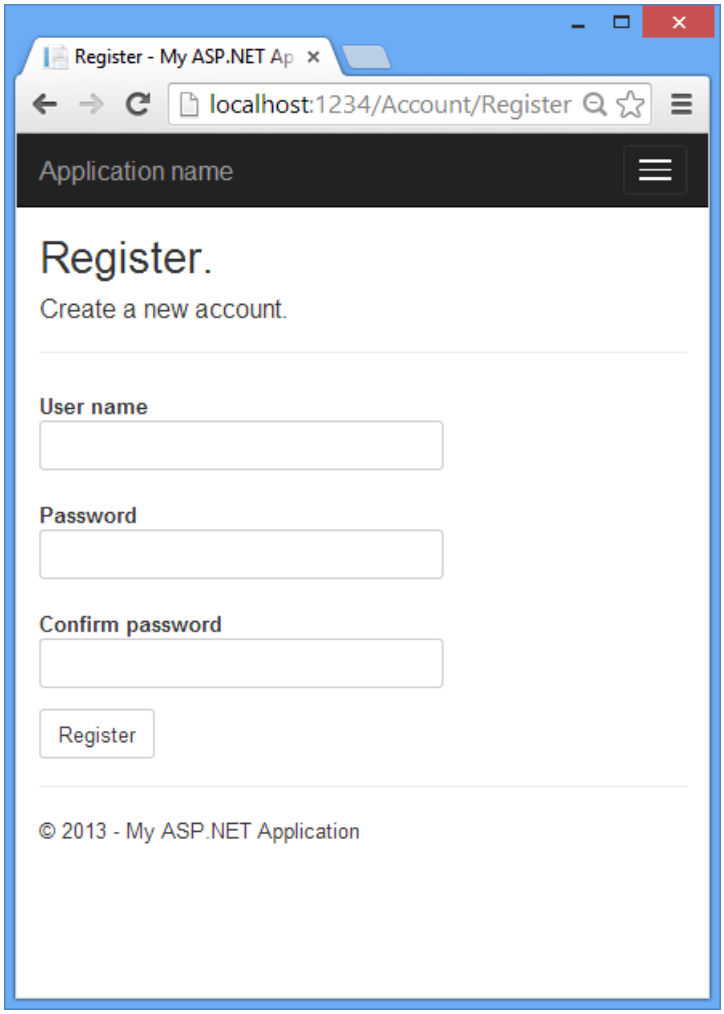
1. Create an ASP.NET MVC application with Individual Accounts. You can use ASP.NET Identity in ASP.NET MVC, Web Forms, Web API, SignalR etc. In this article we will start with an ASP.NET MVC application.



2. The created project contains the following three packages for ASP.NET Identity.

- `Microsoft.AspNet.Identity.EntityFramework`
  This package has the Entity Framework implementation of ASP.NET Identity which will persist the ASP.NET Identity data and schema to SQL Server.
- `Microsoft.AspNet.Identity.Core`
  This package has the core interfaces for ASP.NET Identity. This package can be used to write an implementation for ASP.NET Identity that targets different persistence stores such as Azure Table Storage, NoSQL databases etc.
- `Microsoft.AspNet.Identity.OWIN`
  This package contains functionality that is used to plug in OWIN authentication with ASP.NET Identity in ASP.NET applications. This is used when you add sign in functionality to your application and call into OWIN Cookie Authentication middleware to generate a cookie.

3. Creating a user.
   Launch the application and then click on the **Register** link to create a user. The following image shows the Register page that collects the user name and password.



When the user selects the **Register** button, the `Register` action of the Account controller creates the user by calling the ASP.NET Identity API, as highlighted below:

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser() { UserName = model.UserName };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            await SignInAsync(user, isPersistent: false);
            return RedirectToAction("Index", "Home");
        }
        else
        {
            AddErrors(result);
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

4. Sign in.
   If the user was successfully created, she is signed in by the `SignInAsync` method.

```
 [HttpPost]
 [AllowAnonymous]
 [ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
             await SignInManager.SignInAsync(user, isPersistent:false, rememberBrowser:false);

             // For more information on how to enable account confirmation and password reset please
visit https://go.microsoft.com/fwlink/?LinkID=320771
             // Send an email with this link
             // string code = await UserManager.GenerateEmailConfirmationTokenAsync(user.Id);
             // var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code =
code }, protocol: Request.Url.Scheme);
             // await UserManager.SendEmailAsync(user.Id, "Confirm your account", "Please confirm your
account by clicking <a href=\"" + callbackUrl + "\">here</a>");

             return RedirectToAction("Index", "Home");
         }
         AddErrors(result);
     }

    // If we got this far, something failed, redisplay form
     return View(model);
 }
```

The `SignInManager.SignInAsync` method generates a ClaimsIdentity. Since ASP.NET Identity and OWIN Cookie Authentication are claims-based system, the framework requires the app to generate a ClaimsIdentity for the user. ClaimsIdentity has information about all the claims for the user, such as what roles the user belongs to.

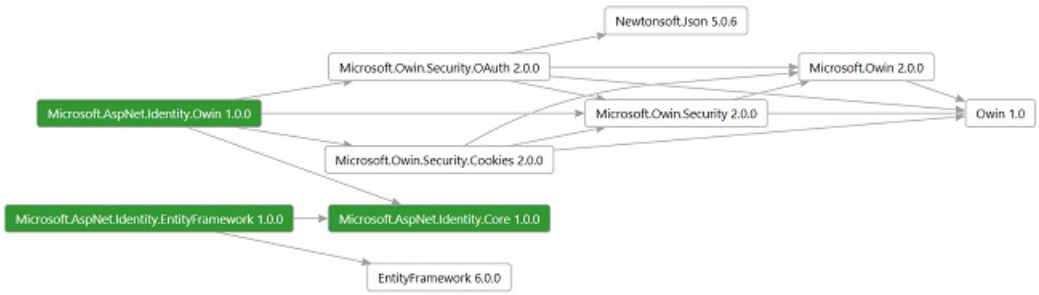
5. Log off.
   Select the **Log off** link to call the LogOff action in the account controller.

```
// POST: /Account/LogOff
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult LogOff()
{
    AuthenticationManager.SignOut();
    return RedirectToAction("Index", "Home");
}
```

   The highlighted code above shows the OWIN `AuthenticationManager.SignOut` method. This is analogous to FormsAuthentication.SignOut method used by the FormsAuthentication module in Web Forms.

Components of ASP.NET Identity

The diagram below shows the components of the ASP.NET Identity system (select on this or on the diagram to enlarge it). The packages in green make up the ASP.NET Identity system. All the other packages are dependencies which are needed to use the ASP.NET Identity system in ASP.NET applications.



The following is a brief description of the NuGet packages not mentioned previously:

- Microsoft.Owin.Security.Cookies
  Middleware that enables an application to use cookie based authentication, similar to ASP.NET's Forms Authentication.
- EntityFramework
  Entity Framework is Microsoft's recommended data access technology for relational databases.

Migrating from Membership to ASP.NET Identity

We hope to soon provide guidance on migrating your existing apps that use ASP.NET Membership or Simple Membership to the new ASP.NET Identity system.

Next Steps

- Create an ASP.NET MVC 5 App with Facebook and Google OAuth2 and OpenID Sign-on
  The tutorial uses the ASP.NET Identity API to add profile information to the user database, and how to authenticate with Google and Facebook.
- Create an ASP.NET MVC app with auth and SQL DB and deploy to Azure App Service
  This tutorial shows how to use the Identity API to add users and roles.
- https://github.com/rustd/AspnetIdentitySample
  Sample application that shows how to add basic roles and user support and how to do roles and user

management.

ASP.NET Identity Recommended Resources

2/20/2020 • 7 minutes to read • Edit Online

by Rick Anderson

> This topic provides links to documentation resources about how to use ASP.NET Identity.
>
> If you know a great blog post, stackoverflow thread, or any other link that would be useful, [send us an email](aspnetue@microsoft.com?subject=Identity recommended resources) with the link or just leave a message at the bottom of this page.

- Getting Started with ASP.NET Identity
- New featured Must Read articles
- Intermediate ASP.NET Identity
- Videos
- Where to ask questions, request features, report a bug and nightly builds
- Blog posts on Identity
- Custom Storage Providers for ASP.NET Identity
- Additional Identity Resources
- Q & A (question/answer)

Getting Started with ASP.NET Identity

- MVC 5 App with Facebook, Twitter, LinkedIn and Google OAuth2 Sign-on This tutorial shows you how to write an ASP.NET MVC 5 app with Facebook and Google OAuth 2 authorization. It also shows how to add additional data to the Identity database.
- Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL Database to a Azure. This tutorial adds Azure deployment, how to secure your app with roles, how to use the membership API to add users and roles, and additional security features.
- Introduction to ASP.NET Identity
- Create a secure ASP.NET MVC 5 web app with log in, email confirmation and password reset
- ASP.NET MVC 5 app with SMS and email Two-Factor Authentication

New featured Must Read articles

- Walkthrough: ASP.NET MVC Identity with Microsoft Account Authentication by Benjamin Day
- ASP.NET Identity 2.0 Extending Identity Models and Using Integer Keys Instead of Strings
- AngularJS Token Authentication using ASP.NET Web API 2, Owin, and Identity
- Thinktecture.IdentityManager as a replacement for the WSAT
- ASP.NET Identity 2.0: Customizing Users and Roles

Intermediate ASP.NET Identity

- Account Confirmation and Password Recovery with ASP.NET Identity
- Two-factor authentication using SMS and email with ASP.NET Identity
- Migrating an Existing Website from SQL Membership to ASP.NET Identity
- Adding ASP.NET Identity to an Empty or Existing Web Forms Project

- MSDN Magazine External Authentication with ASP.NET Identity by Dino Esposito
- MSDN MagazineA First Look at ASP.NET Identity by Dino Esposito
- ASP.NET Identity – User Lockout

Where to ask questions, request features, report a bug and nightly builds

- For StackOverflow, use the tag aspnet-identity
- For the ASP.NET forums, post to the Security forum and add **ASP.NET Identity** to the title.
- ASP.NET Identity on GitHub Get nightly builds, request features, open bugs.

Blog posts on Identity

- What is a SecurityStamp in ASP.NET Identity?
- By John Atten
  - ASP.NET Identity 2.0 Extending Identity Models and Using Integer Keys Instead of Strings
  - ASP.NET Identity 2.0: Customizing Users and Roles
  - ASP.NET MVC and Identity 2.0: Understanding the Basics
  - Setting Up Account Validation and Two-Factor Authorization
  - Configuring Db Connection and Code-First Migration for Identity Accounts in ASP.NET MVC 5 and Visual Studio 2013
- By Taiseer Joudeh
  - Token Based Authentication using ASP.NET Web API 2, Owin middleware, and ASP.NET Identity
  - AngularJS Token Authentication using ASP.NET Web API 2, Owin, and Identity
  - Enable OAuth Refresh Tokens in AngularJS App using ASP .NET Web API 2, and Owin – Part 3.
- By Anders Abel
  - Understanding the Owin External Authentication Pipeline
  - ASP.NET Identity and Owin Overview

  By K. Scott Allen on Ode to Code
  - ASP.NET Core Identity This blog examines the core abstractions, including IUser, IUserStore and the I*Store interfaces.
  - ASP.NET Identity with the Entity Framework Individual User Accounts in MVC 5, Web API and SPA apps, connection strings and managing contexts
  - Customization Options With ASP.NET Identity
  - Implementing ASP.NET Identity
- Benjamin DayWalkthrough: ASP.NET MVC Identity with Microsoft Account Authentication
- Brock Allen
  - A primer on external login providers (social logins) with OWIN/Katana authentication middleware
  - Introducing IdentityReboot: a set of extensions to ASP.NET Identity that implement the major missing features I've complained about.
- Pranav Rastogi
  - Get more information from Social providers
- @beabigrockstar (Jerrie Pelser)
  - 2 Factor authentication

  - Using Google Authenticator with ASP.NET Identity
  - ASP.NET MVC 5 Authentication Guides
- Get more information from Social providers used in the VS 2013 project templates
- Building a simple ToDo application with ASP.NET Identity and associating Users with ToDoes
- Google OpenId integration issues with ASP.NET Identity If you get the error: HTTP Error 404.15 – Not Found The request filtering module is configured to deny a request where the query string is too long
- Thinktecture.IdentityManager as a replacement for the WSAT
- AngularJS Token Authentication using ASP.NET Web API 2, Owin, and Identity
- Simple Asp.net Identity Core Without Entity Framework
- Working with Roles in ASP.NET Identity for MVC by Sheo Narayan
- Moving to ASP.NET Identity from ASP.NET Membership by Alistair Matthews

Videos

- Channel 9 Securing ASP.NET Applications and Services: Security Facelift for Modern Applications by Ido Flatow
- Channel 9 ASP.NET Identity Intro by Pranav Rastogi
- Channel 9 ASP.NET Authentication using ASP.NET Identity by Cory Fowler
- Channel 9 Building Modern Web Apps: ASP.NET Identity by Jeff Koch
- Channel 9 Securing your website with ASP.NET Identity by Alex Thissen
- Use ASP.NET Identity on an existing DB-Model by Alexander Schmidt
- ASP.NET One Identity by Ivaylo Kenov of Telerik
- Czech ASP.NET Identity In this lecture we will show how to deploy basic authentication, how to add support for external identity providers such as Twitter or Facebook, and how to use one-time passwords (OTP). [ASP.NET Identity je nástupce Membership a Role providerů v ASP.NET, tedy knihovna pro zajištění autentizace uživatelů. V této přednášce si ukážeme, jak nasad]

Custom Storage Providers for ASP.NET Identity

If you want to write your own provider, read Overview of Custom Storage Providers for ASP.NET Identity and Implementing ASP.NET Identity and then examine the source of one of the OSS projects listed below.

- Tutorial: Overview of Custom Storage Providers for ASP.NET Identity by Tom FitzMacken
- Blog: Implementing ASP.NET Identity
- Tutorial:Setting up the basic Identity accounts, and pointing them at an external DB. By @xivSolutions.
- Tutorial: Implementing a Custom MySQL ASP.NET Identity Storage Provider
- Azure Table Storage by James Randall.
- Azure Table Storage: AspNet.Identity.TableStorage by @stuartleeks.
- CouchDB / Cloudant by Daniel Wertheim.
- Elastic Search: Elastic Identity by Bombsquad AB.
- MongoDB by Jonathan Sheely Jonathan Sheely.
- NHibernate.AspNet.Identity by Antônio Milesi Bastos.
- RavenDB by @tourismgeek.
- RavenDB.AspNet.Identity by ILMServices.
- Redis: Redis.AspNet.Identity
- T4 Templates to generate EF code for a "database first" user store: AspNet.Identity.EntityFramework

Additional ASP.NET Identity Resources

- Introducing the Yahoo and LinkedIn OAuth security providers for OWIN by Jerrie Pelser for Yahoo and LinkedIn instructions.

Q&A (question/answer)

- Q: Locked out users who have enabled "remember me" (so they don't have to go through 2FA on that computer/browser) are not locked out. Why and how do I prevent that? Answer here.
- **Q**: How can I store custom claims, such as the user's real name, in the ASP.NET Identity cookie to avoid unnecessary database queries on every request. Answer here.
- **Q: Updating AspNetUser Password Hash**: I have 2 projects. One of them is using ASP.NET authentication, the other uses Windows authentication, which is the administration side. I want the Admin project to be able to manage the users of the other. I can modify everything except the password. Answer here.
- **Q**: How can I reset password as a admin for other users? Answer here.
- **Q**: Can I change the displayed name of the UserName field in ASP.NET MVC IdentityUser? Answer here.
- **Q**: How can I gran users permissions to add other users to certain roles? Answer here.
- **Q**: Storing profile information in the AspNetUsers table vs. the AspNetUserClaims table. Answer here.
- **Q**: Remember me when using an external authentication provider. Answer here.
- **Q**: Why does every request require a ApplicationDBContext, isn't that too much overhead?. Answer, No, the overhead is low.
- Q: How do I get a list of logged in users? Answer here.
- Q: How can I detect when a user logs in with Microsoft.AspNet.Identity? Answer here.
- Q: How do I get localized error messages for Identity? Answer here.
- Q: How do I configure the CookieMiddleware to get fresh claims every 30 minutes? Answer here.
- Q: How do modify the claims for the user after they have signed in? Answer here.
- Q: How do I invalidate security tokens? Answer here.
- Q: How do is store claims in the cookie middleware? Answer here.
- Q: I'd like to have a PIN or security check on each action method in my MVC app, but I'd like to store the users success so they don't have to enter the PIN on every request to that action method. Answer here.
- Q: I'd like to save the returned email address from a social provider to the DB, how do I do that? Answer here:
- Q: How can I detect when a user logs in both with/with-out a "remember me" cookie? Answer here.
- Q: Can I modify claims in ASP.NET Identity with OWIN after calling SignIn? Answer: Calling SignIn is exactly what you are supposed to do when you want to modify the claims for the user. It basically causes the ClaimsIdentity to be serialized into the cookie, which is why you see the new claims show up on subsequent requests.

Adding ASP.NET Identity to an Empty or Existing Web Forms Project

2/20/2020 • 10 minutes to read • Edit Online

This tutorial shows you how to add ASP.NET Identity (the new membership system for ASP.NET) to an ASP.NET application.

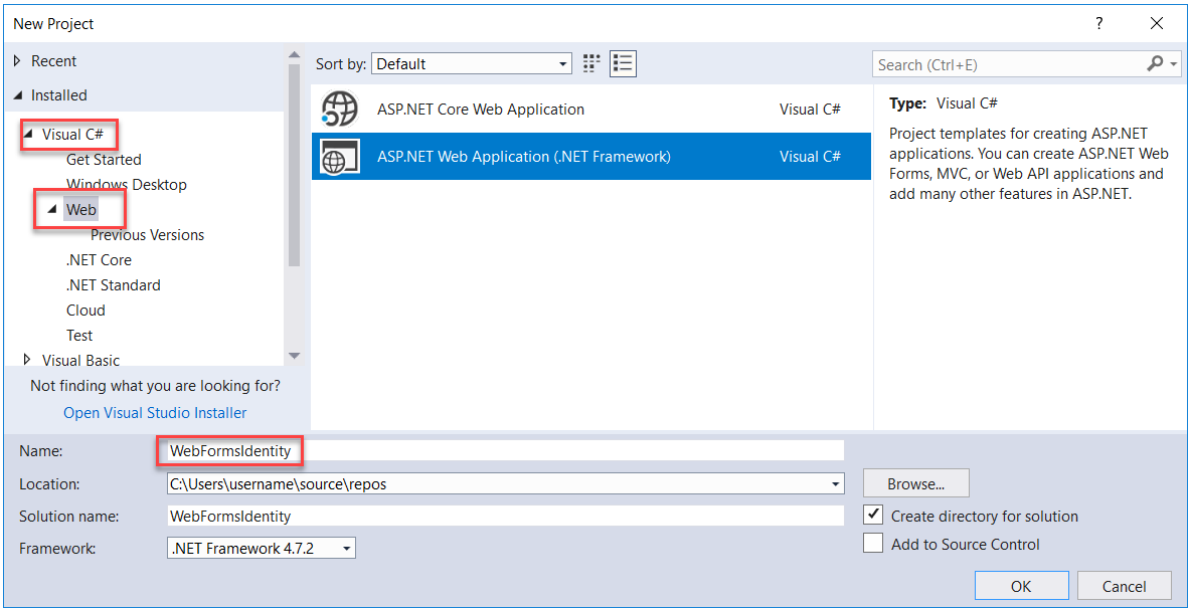
When you create a new Web Forms or MVC project in Visual Studio 2017 RTM with Individual Accounts, Visual Studio will install all the required packages and add all necessary classes for you. This tutorial will illustrate the steps to add ASP.NET Identity support to your existing Web Forms project or a new empty project. I will outline all the NuGet packages you need to install, and classes you need to add. I will go over sample Web Forms for registering new users and logging in while highlighting all main entry point APIs for user management and authentication. This sample will use the ASP.NET Identity default implementation for SQL data storage which is built on Entity Framework. This tutorial, we will use LocalDB for the SQL database.

Get started with ASP.NET Identity

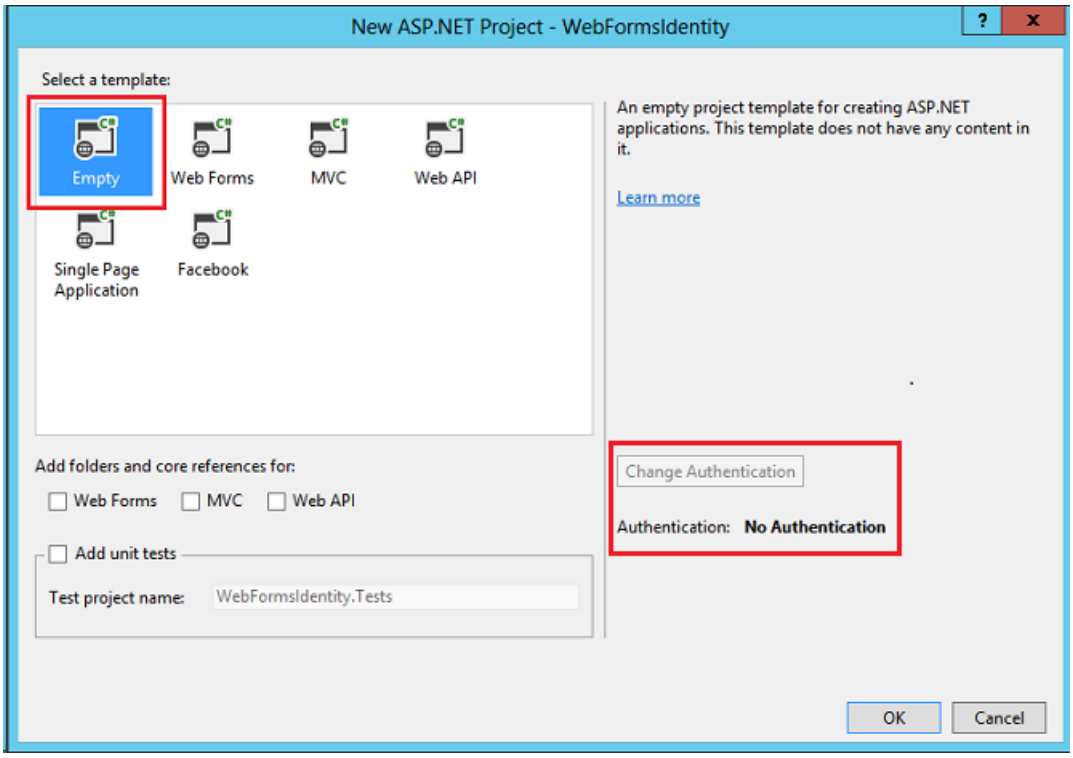
1. Start by installing and running Visual Studio 2017.

2. Select **New Project** from the Start page, or you can use the menu and select **File**, and then **New Project**.

3. In the left pane, expand **Visual C#**, then select **Web**, then **ASP.NET Web Application (.Net Framework)**. Name your project "WebFormsIdentity" and select **OK**.



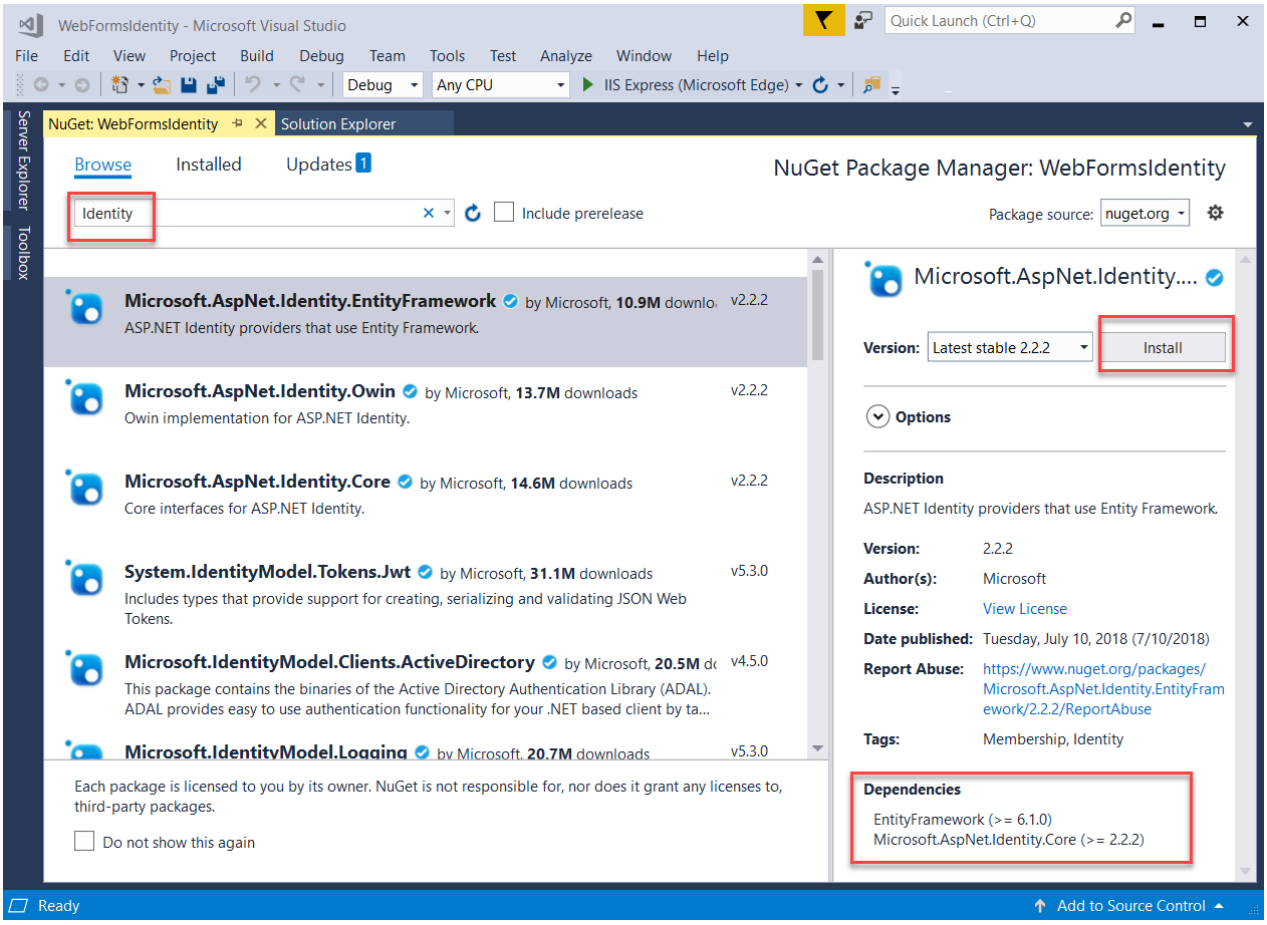
4. In the **New ASP.NET Project** dialog, select the **Empty** template.



Notice the **Change Authentication** button is disabled and no authentication support is provided in this template. The Web Forms, MVC and Web API templates allow you to select the authentication approach.

Add Identity packages to your app

In Solution Explorer, right-click your project and select **Manage NuGet Packages**. Search for and install the **Microsoft.AspNet.Identity.EntityFramework** package.

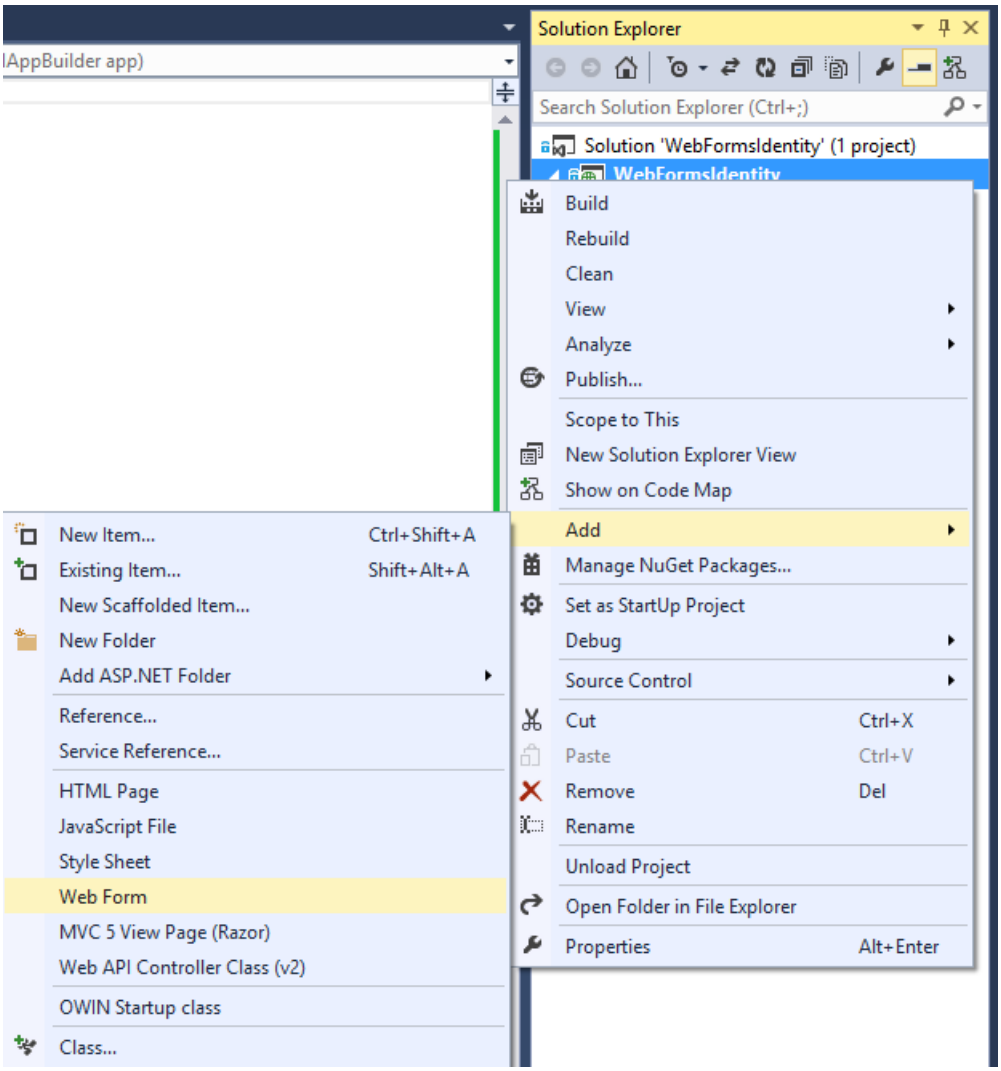


Note that this package will install the dependency packages: **EntityFramework** and **Microsoft ASP.NET**

Identity Core.

Add a web form to register users

1. In **Solution Explorer**, right-click your project and select **Add**, and then **Web Form**.



2. In the **Specify Name for Item** dialog box, name the new web form **Register**, and then select **OK**

3. Replace the markup in the generated *Register.aspx* file with the code below. The code changes are highlighted.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Register.aspx.cs"
Inherits="WebFormsIdentity.Register" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body style="font-family: Arial, Helvetica, sans-serif; font-size: small">
    <form id="form1" runat="server">
    <div>
        <h4 style="font-size: medium">Register a new user</h4>
        <hr />
        <p>
            <asp:Literal runat="server" ID="StatusMessage" />
        </p>
        <div style="margin-bottom:10px">
            <asp:Label runat="server" AssociatedControlID="UserName">User name</asp:Label>
            <div>
                <asp:TextBox runat="server" ID="UserName" />
            </div>
        </div>
        <div style="margin-bottom:10px">
            <asp:Label runat="server" AssociatedControlID="Password">Password</asp:Label>
            <div>
                <asp:TextBox runat="server" ID="Password" TextMode="Password" />
            </div>
        </div>
        <div style="margin-bottom:10px">
            <asp:Label runat="server" AssociatedControlID="ConfirmPassword">Confirm
password</asp:Label>
            <div>
                <asp:TextBox runat="server" ID="ConfirmPassword" TextMode="Password" />
            </div>
        </div>
        <div>
            <div>
                <asp:Button runat="server" OnClick="CreateUser_Click" Text="Register" />
            </div>
        </div>
    </div>
    </form>
</body>
</html>
```

NOTE

This is just a simplified version of the *Register.aspx* file that is created when you create a new ASP.NET Web Forms project. The markup above adds form fields and a button to register a new user.

4. Open the *Register.aspx.cs* file and replace the contents of the file with the following code:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using System;
using System.Linq;

namespace WebFormsIdentity
{
   public partial class Register : System.Web.UI.Page
   {
      protected void CreateUser_Click(object sender, EventArgs e)
      {
         // Default UserStore constructor uses the default connection string named: DefaultConnection
         var userStore = new UserStore<IdentityUser>();
         var manager = new UserManager<IdentityUser>(userStore);

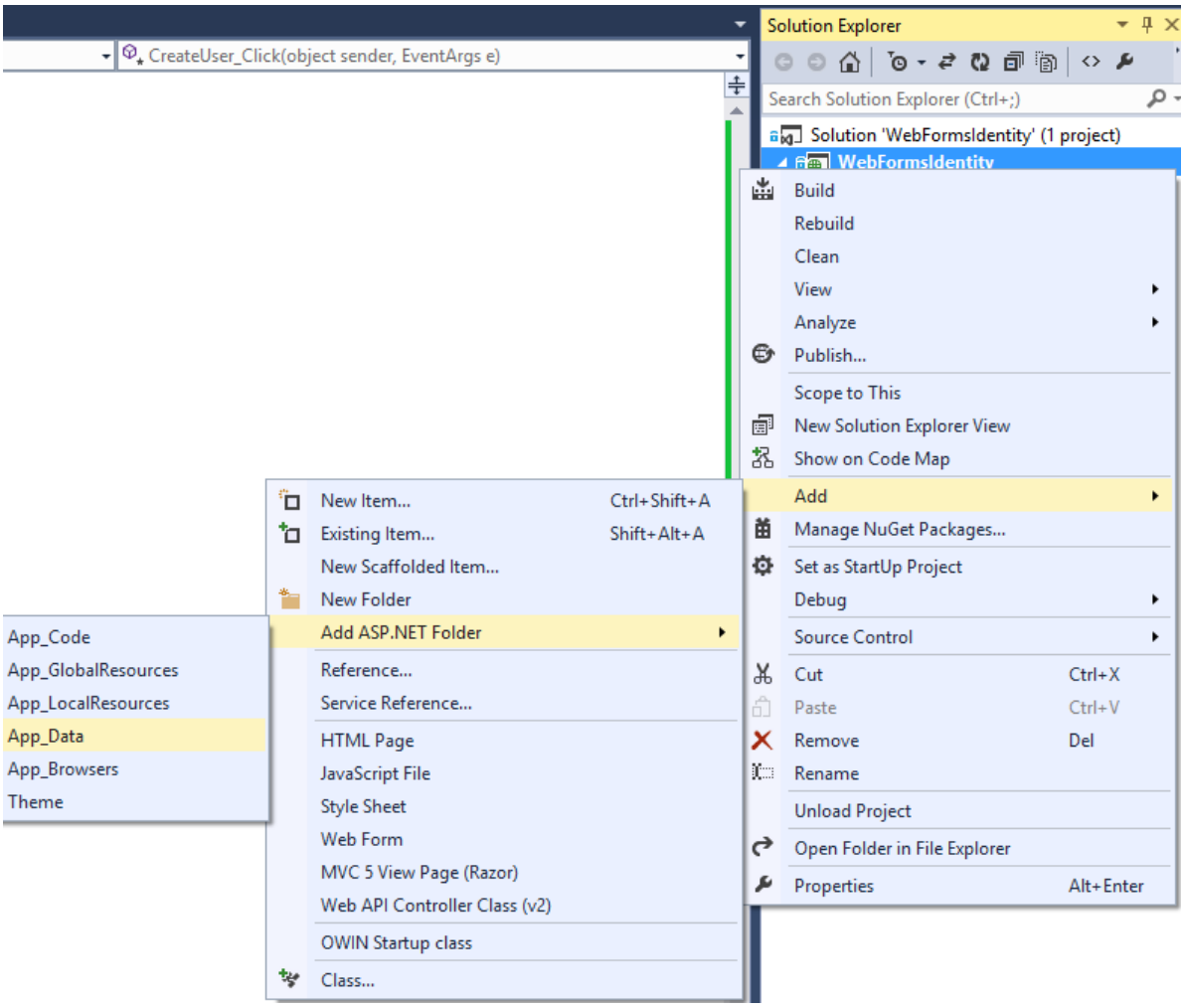
         var user = new IdentityUser() { UserName = UserName.Text };
         IdentityResult result = manager.Create(user, Password.Text);

         if (result.Succeeded)
         {
            StatusMessage.Text = string.Format("User {0} was created successfully!", user.UserName);
         }
         else
         {
            StatusMessage.Text = result.Errors.FirstOrDefault();
         }
      }
   }
}
```

NOTE

1. The code above is a simplified version of the *Register.aspx.cs* file that is created when you create a new ASP.NET Web Forms project.
2. The *IdentityUser* class is the default EntityFramework implementation of the *IUser* interface. *IUser* interface is the minimal interface for a user on ASP.NET Identity Core.
3. The *UserStore* class is the default EntityFramework implementation of a user store. This class implements the ASP.NET Identity Core's minimal interfaces: *IUserStore*, *IUserLoginStore*, *IUserClaimStore* and *IUserRoleStore*.
4. The *UserManager* class exposes user related APIs which will automatically save changes to the *UserStore*.
5. The *IdentityResult* class represents the result of an identity operation.

5. In **Solution Explorer**, right-click your project and select **Add**, **Add ASP.NET Folder** and then **App_Data**.



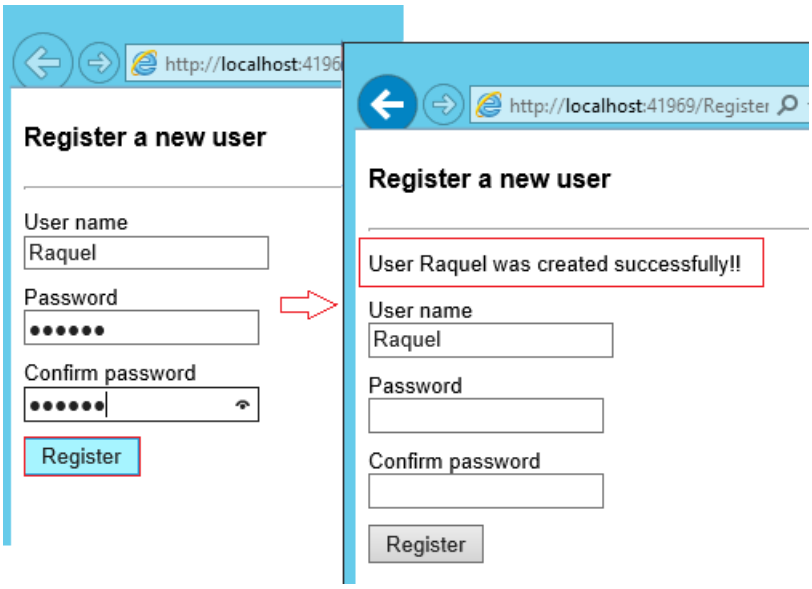
6. Open the *Web.config* file and add a connection string entry for the database we will use to store user information. The database will be created at runtime by EntityFramework for the Identity entities. The connection string is similar to one created for you when you create a new Web Forms project. The highlighted code shows the markup you should add:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  For more information on how to configure your ASP.NET application, please visit
  https://go.microsoft.com/fwlink/?LinkId=169433
  -->
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit
https://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=6.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false" />
  </configSections>
   <connectionStrings>
      <add name="DefaultConnection" connectionString="Data Source=
(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\WebFormsIdentity.mdf;Initial
Catalog=WebFormsIdentity;Integrated Security=True"
            providerName="System.Data.SqlClient" />
   </connectionStrings>
  <system.web>
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory,
EntityFramework">
      <parameters>
        <parameter value="v11.0" />
      </parameters>
    </defaultConnectionFactory>
    <providers>
      <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
</configuration>
```

NOTE

For Visual Studio 2015 or higher, replace `(localdb)\v11.0` with `(localdb)\MSSQLLocalDB` in your connection string.

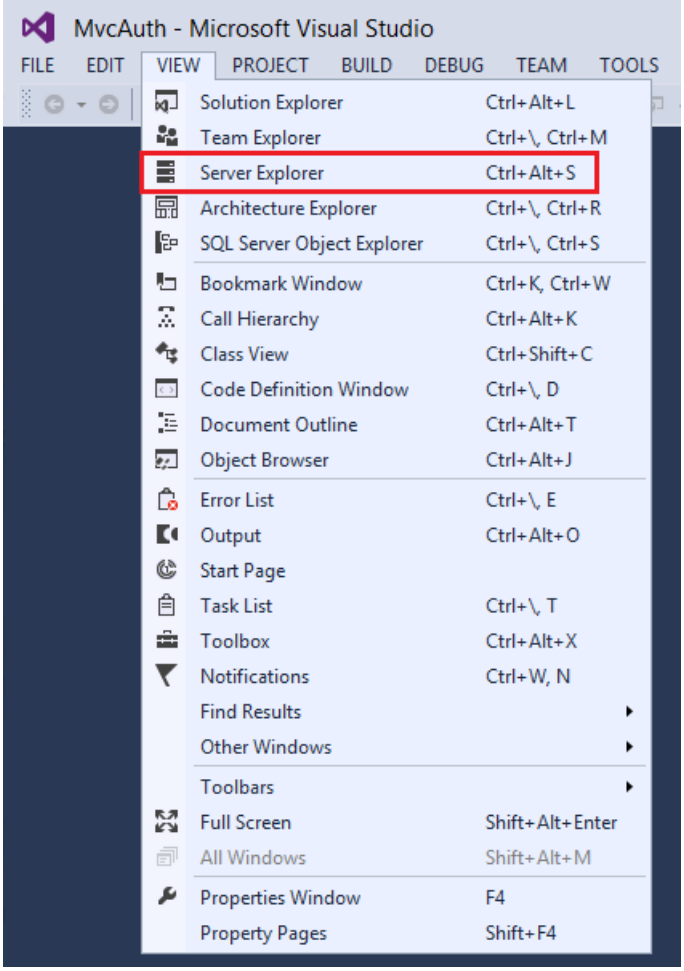
7. Right click file *Register.aspx* in your project and select **Set as Start Page**. Press Ctrl + F5 to build and run the web application. Enter a new user name and password and then select **Register**.



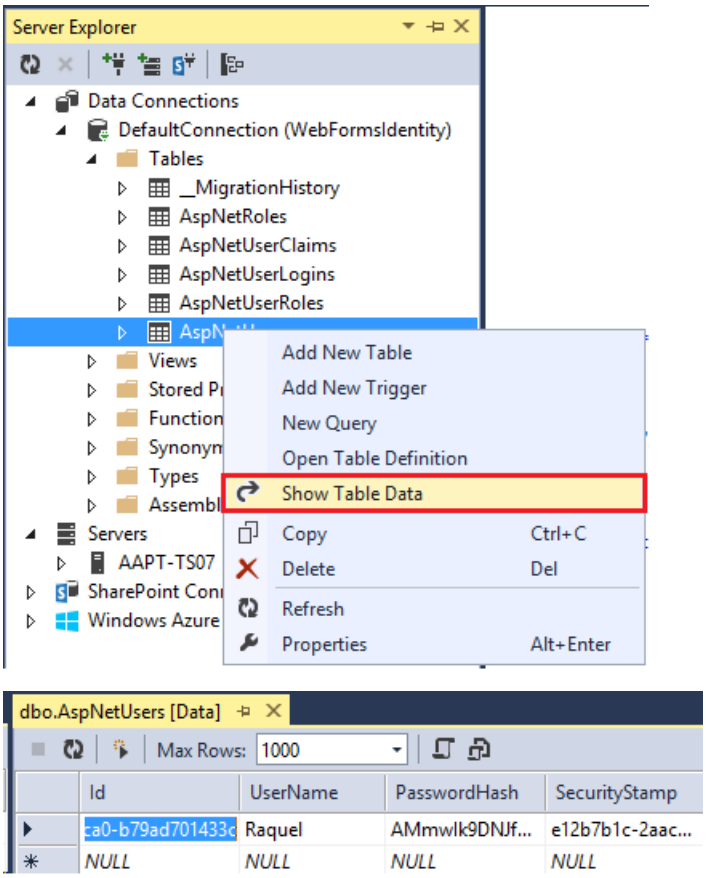
> **NOTE**
>
> ASP.NET Identity has support for validation and in this sample you can verify the default behavior on User and Password validators that come from the Identity Core package. The default validator for User (`UserValidator`) has a property `AllowOnlyAlphanumericUserNames` that has default value set to `true`. The default validator for Password (`MinimumLengthValidator`) ensures that password has at least 6 characters. These validators are properties on `UserManager` that can be overridden if you want to have custom validation,

Verify the LocalDb Identity database and tables generated by Entity Framework

1. In the **View** menu, select **Server Explorer**.



2. Expand **DefaultConnection (WebFormsIdentity)**, expand **Tables**, right-click **AspNetUsers** and then select **Show Table Data**.

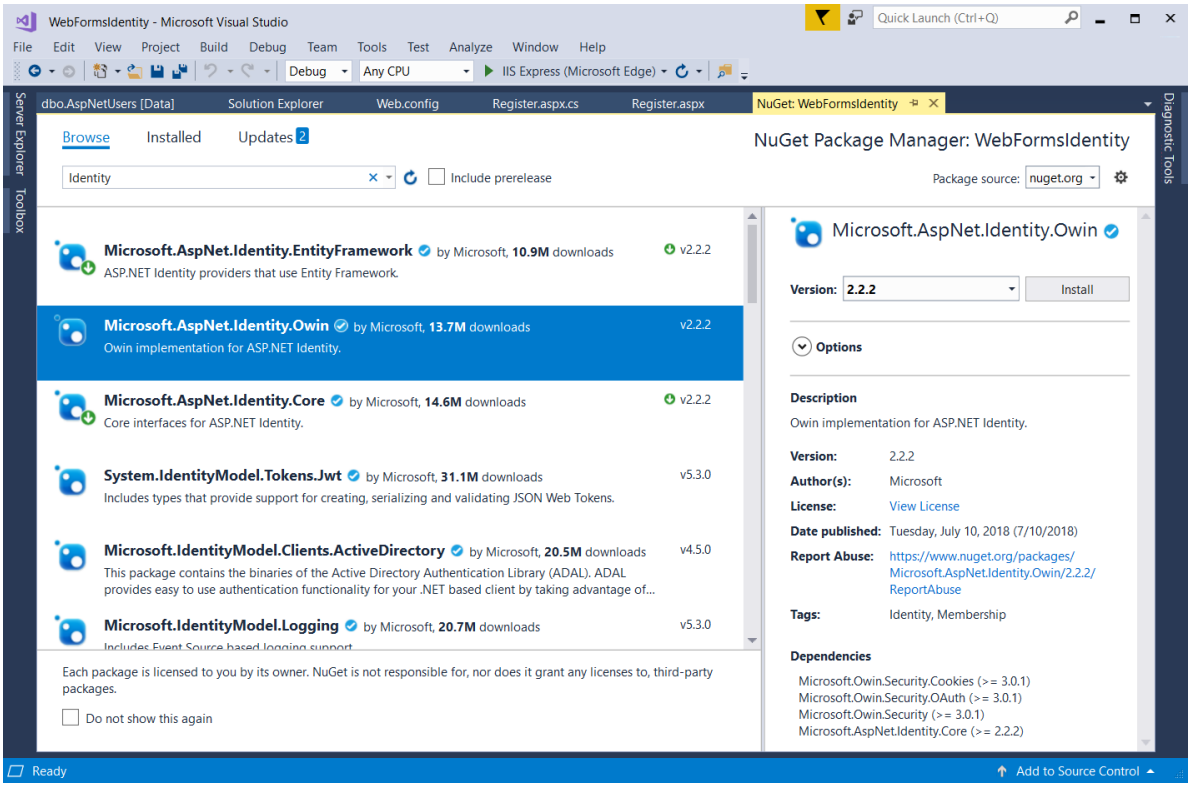


Configure the application for OWIN authentication

At this point we have only added support for creating users. Now, we are going to demonstrate how we can add authentication to login a user. ASP.NET Identity uses Microsoft OWIN Authentication middleware for forms authentication. The OWIN Cookie Authentication is a cookie and claims based authentication mechanism that can be used by any framework hosted on OWIN or IIS. With this model, the same authentication packages can be used across multiple frameworks including ASP.NET MVC and Web Forms. For more information on project Katana and how to run middleware in a host agnostic see Getting Started with the Katana Project.

Install authentication packages to your application

1. In Solution Explorer, right-click your project and select **Manage NuGet Packages**. Search for and install the ***Microsoft.AspNet.Identity.Owin*** package.

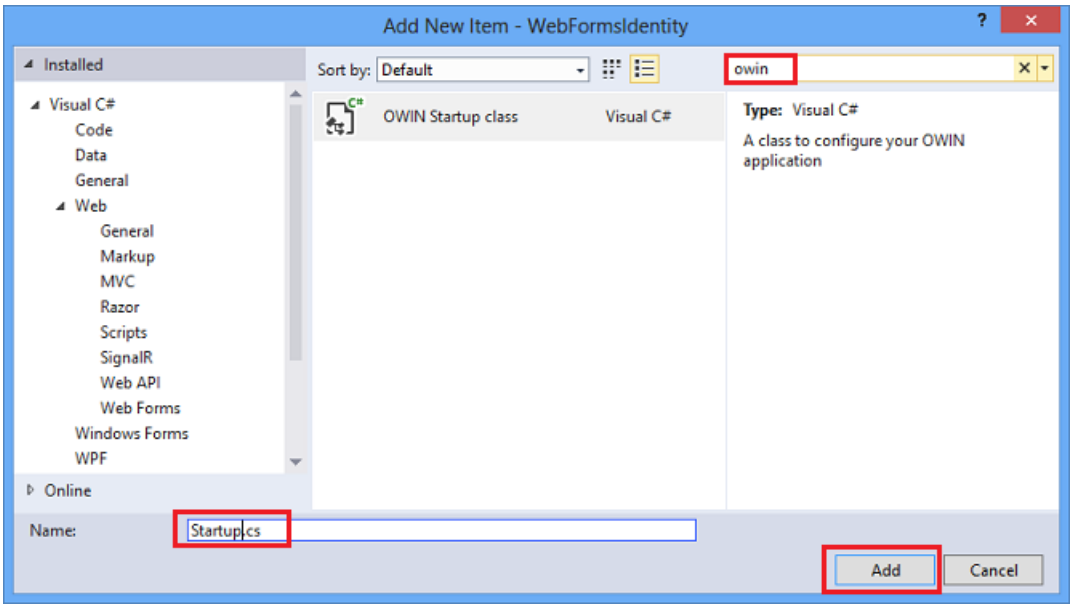


2. Search for and install the ***Microsoft.Owin.Host.SystemWeb*** package.

> **NOTE**
>
> The **Microsoft.Aspnet.Identity.Owin** package contains a set of OWIN extension classes to manage and configure OWIN authentication middleware to be consumed by ASP.NET Identity Core packages. The **Microsoft.Owin.Host.SystemWeb** package contains an OWIN server that enables OWIN-based applications to run on IIS using the ASP.NET request pipeline. For more information see OWIN Middleware in the IIS integrated pipeline.

Add OWIN startup and authentication configuration classes

1. In **Solution Explorer**, right-click your project, select **Add**, and then **Add New Item**. In the search text box dialog, type "*owin*". Name the class "*Startup*" and select **Add**.



2. In the Startup.cs file, add the highlighted code shown below to configure OWIN cookie authentication.

```
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Owin;

[assembly: OwinStartup(typeof(WebFormsIdentity.Startup))]

namespace WebFormsIdentity
{
   public class Startup
   {
      public void Configuration(IAppBuilder app)
      {
         // For more information on how to configure your application, visit
https://go.microsoft.com/fwlink/?LinkID=316888
         app.UseCookieAuthentication(new CookieAuthenticationOptions
         {
            AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
            LoginPath = new PathString("/Login")
         });
      }
   }
}
```

NOTE

This class contains the `OwinStartup` attribute for specifying the OWIN startup class. Every OWIN application has a startup class where you specify components for the application pipeline. See OWIN Startup Class Detection for more info on this model.

Add web forms for registering and signing in users

1. Open the *Register.aspx.cs* file and add the following code which signs in the user when registration succeeds.

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.Owin.Security;
using System;
using System.Linq;
using System.Web;

namespace WebFormsIdentity
{
   public partial class Register : System.Web.UI.Page
   {
      protected void CreateUser_Click(object sender, EventArgs e)
      {
         // Default UserStore constructor uses the default connection string named: DefaultConnection
         var userStore = new UserStore<IdentityUser>();
         var manager = new UserManager<IdentityUser>(userStore);
         var user = new IdentityUser() { UserName = UserName.Text };

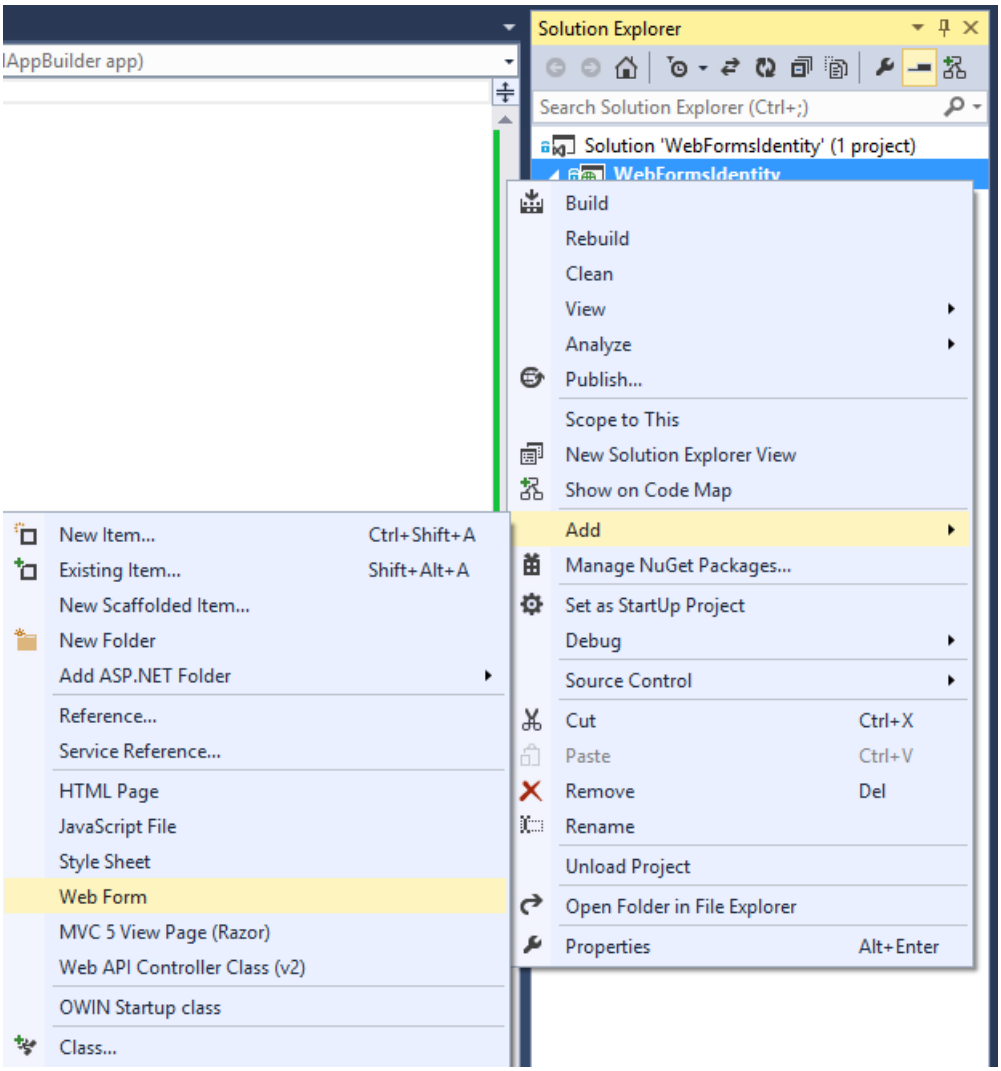
         IdentityResult result = manager.Create(user, Password.Text);

         if (result.Succeeded)
         {
            var authenticationManager = HttpContext.Current.GetOwinContext().Authentication;
            var userIdentity = manager.CreateIdentity(user,
DefaultAuthenticationTypes.ApplicationCookie);
            authenticationManager.SignIn(new AuthenticationProperties() { }, userIdentity);
            Response.Redirect("~/Login.aspx");
         }
         else
         {
            StatusMessage.Text = result.Errors.FirstOrDefault();
         }
      }
   }
}
```

NOTE

- Since ASP.NET Identity and OWIN Cookie Authentication are claims based system, the framework requires the app developer to generate a ClaimsIdentity for the user. ClaimsIdentity has information about all the claims for the user such as what Roles the user belongs to. You can also add more claims for the user at this stage.
- You can sign in the user by using the AuthenticationManager from OWIN and calling `SignIn` and passing in the ClaimsIdentity as shown above. This code will sign in the user and generate a cookie as well. This call is analogous to FormAuthentication.SetAuthCookie used by the FormsAuthentication module.

2. In **Solution Explorer**, right-click your project, select **Add**, and then **Web Form**. Name the web form **Login**.



3. Replace the contents of the *Login.aspx* file with the following code:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Login.aspx.cs"
Inherits="WebFormsIdentity.Login" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
   <title></title>
</head>
<body style="font-family: Arial, Helvetica, sans-serif; font-size: small">
   <form id="form1" runat="server">
      <div>
         <h4 style="font-size: medium">Log In</h4>
         <hr />
         <asp:PlaceHolder runat="server" ID="LoginStatus" Visible="false">
            <p>
               <asp:Literal runat="server" ID="StatusText" />
            </p>
         </asp:PlaceHolder>
         <asp:PlaceHolder runat="server" ID="LoginForm" Visible="false">
            <div style="margin-bottom: 10px">
               <asp:Label runat="server" AssociatedControlID="UserName">User name</asp:Label>
               <div>
                  <asp:TextBox runat="server" ID="UserName" />
               </div>
            </div>
            <div style="margin-bottom: 10px">
               <asp:Label runat="server" AssociatedControlID="Password">Password</asp:Label>
               <div>
                  <asp:TextBox runat="server" ID="Password" TextMode="Password" />
               </div>
            </div>
            <div style="margin-bottom: 10px">
               <div>
                  <asp:Button runat="server" OnClick="SignIn" Text="Log in" />
               </div>
            </div>
         </asp:PlaceHolder>
         <asp:PlaceHolder runat="server" ID="LogoutButton" Visible="false">
            <div>
               <div>
                  <asp:Button runat="server" OnClick="SignOut" Text="Log out" />
               </div>
            </div>
         </asp:PlaceHolder>
      </div>
   </form>
</body>
</html>
```

4. Replace the contents of the *Login.aspx.cs* file with the following:

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.Owin.Security;
using System;
using System.Web;
using System.Web.UI.WebControls;

namespace WebFormsIdentity
{
   public partial class Login : System.Web.UI.Page
   {
      protected void Page_Load(object sender, EventArgs e)
      {
         if (!IsPostBack)
         {
            if (User.Identity.IsAuthenticated)
            {
               StatusText.Text = string.Format("Hello {0}!!", User.Identity.GetUserName());
               LoginStatus.Visible = true;
               LogoutButton.Visible = true;
            }
            else
            {
               LoginForm.Visible = true;
            }
         }
      }

      protected void SignIn(object sender, EventArgs e)
      {
         var userStore = new UserStore<IdentityUser>();
         var userManager = new UserManager<IdentityUser>(userStore);
         var user = userManager.Find(UserName.Text, Password.Text);

         if (user != null)
         {
            var authenticationManager = HttpContext.Current.GetOwinContext().Authentication;
            var userIdentity = userManager.CreateIdentity(user,
DefaultAuthenticationTypes.ApplicationCookie);

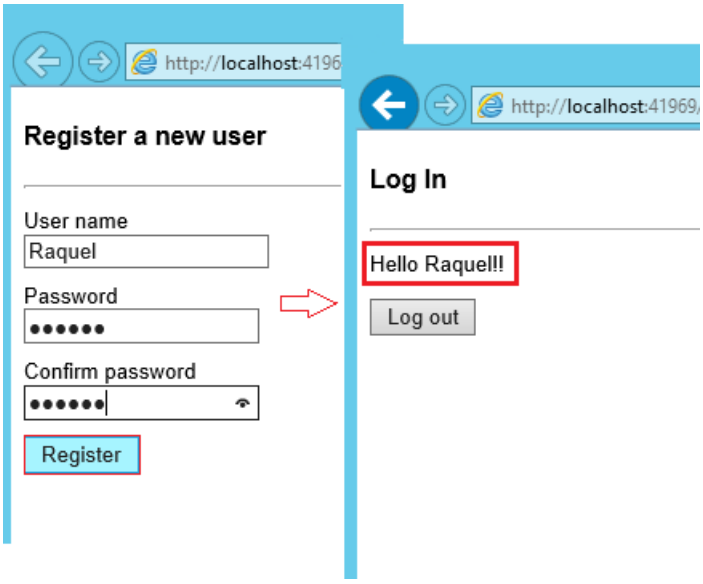
            authenticationManager.SignIn(new AuthenticationProperties() { IsPersistent = false },
userIdentity);
            Response.Redirect("~/Login.aspx");
         }
         else
         {
            StatusText.Text = "Invalid username or password.";
            LoginStatus.Visible = true;
         }
      }

      protected void SignOut(object sender, EventArgs e)
      {
         var authenticationManager = HttpContext.Current.GetOwinContext().Authentication;
         authenticationManager.SignOut();
         Response.Redirect("~/Login.aspx");
      }
   }
}
```

NOTE

- The `Page_Load` now checks for the status of current user and takes action based on its `Context.User.Identity.IsAuthenticated` status. **Display Logged in User Name** : The Microsoft ASP.NET Identity Framework has added extension methods on System.Security.Principal.IIdentity that allows you to get the `UserName` and `UserId` for the logged in User. These extension methods are defined in the `Microsoft.AspNet.Identity.Core` assembly. These extension methods are the replacement for HttpContext.User.Identity.Name .
- SignIn method: `This` method replaces the previous `CreateUser_Click` method in this sample and now signs in the user after successfully creating the user.
  The Microsoft OWIN Framework has added extension methods on `System.Web.HttpContext` that allows you to get a reference to an `IOwinContext`. These extension methods are defined in `Microsoft.Owin.Host.SystemWeb` assembly. The `OwinContext` class exposes an `IAuthenticationManager` property that represents the Authentication middleware functionality available on the current request. You can sign in the user by using the `AuthenticationManager` from OWIN and calling `SignIn` and passing in the `ClaimsIdentity` as shown above. Because ASP.NET Identity and OWIN Cookie Authentication are claims-based system, the framework requires the app to generate a `ClaimsIdentity` for the user. The `ClaimsIdentity` has information about all the claims for the user, such as what roles the user belongs to. You can also add more claims for the user at this stage This code will sign in the user and generate a cookie as well. This call is analogous to FormAuthentication.SetAuthCookie used by the FormsAuthentication module.
- `SignOut` method: Gets a reference to the `AuthenticationManager` from OWIN and calls `SignOut`. This is analogous to FormsAuthentication.SignOut method used by the FormsAuthentication module.

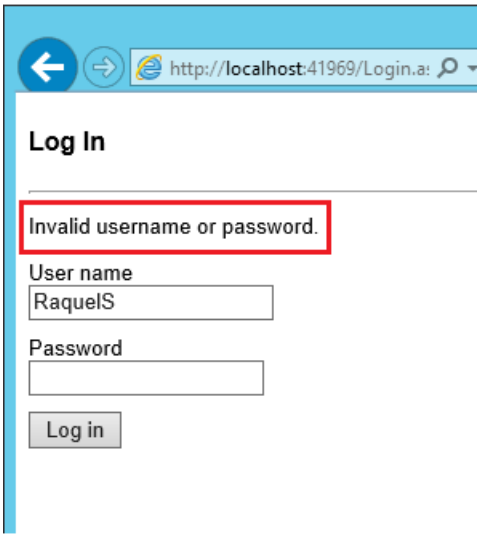
5. Press **Ctrl + F5** to build and run the web application. Enter a new user name and password and then select **Register**.



Note: At this point, the new user is created and logged in.

6. Select the **Log out** button. You are redirected to the Log in form.

7. Enter an invalid user name or password and select the **Log in** button. The `UserManager.Find` method will return null and the error message: " *Invalid user name or password* " will be displayed.


http://localhost:41969/Login.a
Log In
Invalid username or password.
User name
RaquelS
Password
Log in

Developing ASP.NET Apps with Azure Active Directory

2/20/2020 • 9 minutes to read • Edit Online

by Rick Anderson

Microsoft ASP.NET tools for Azure Active Directory simplifies enabling authentication for web apps hosted on Azure. You can use Azure Authentication to authenticate Office 365 users from your organization, corporate accounts synced from your on-premise Active Directory or users created in your own custom Azure Active Directory domain. Enabling Windows Azure Authentication configures your application to authenticate users using a single Azure Active Directory tenant.

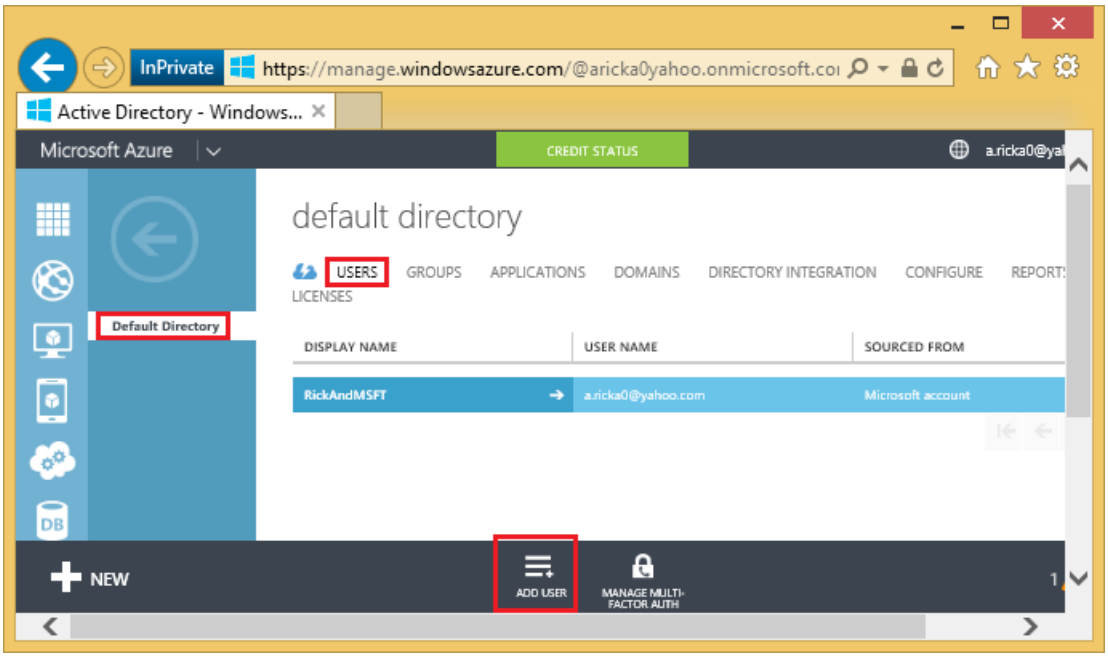
This tutorial will show you how to create an ASP.NET application that is configured for sign-on with Azure Active Directory (Azure AD). You will also learn how to call the Graph API to get information about the currently signed-in user and how to deploy the application to Azure.

Prerequisites

1. Visual Studio Express 2013 for Web or Visual Studio 2013.
2. Visual Studio 2013 Update 4 - Update 3 or higher is required.
3. An Azure account. Click here for a free trial if you do not already have an account.

Add a Global Administrator to your Active Directory

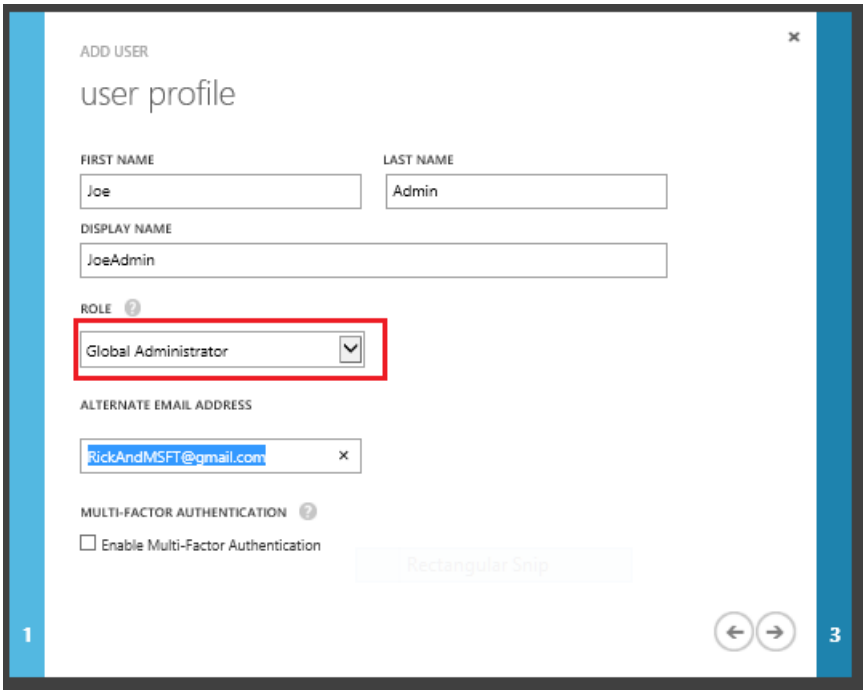
1. Sign in to the Azure Management Portal.
2. All Azure accounts contain a **Default Directory** -- click on it, then click the **Users** tab at the top of the page (see image below).
3. Click Add User.



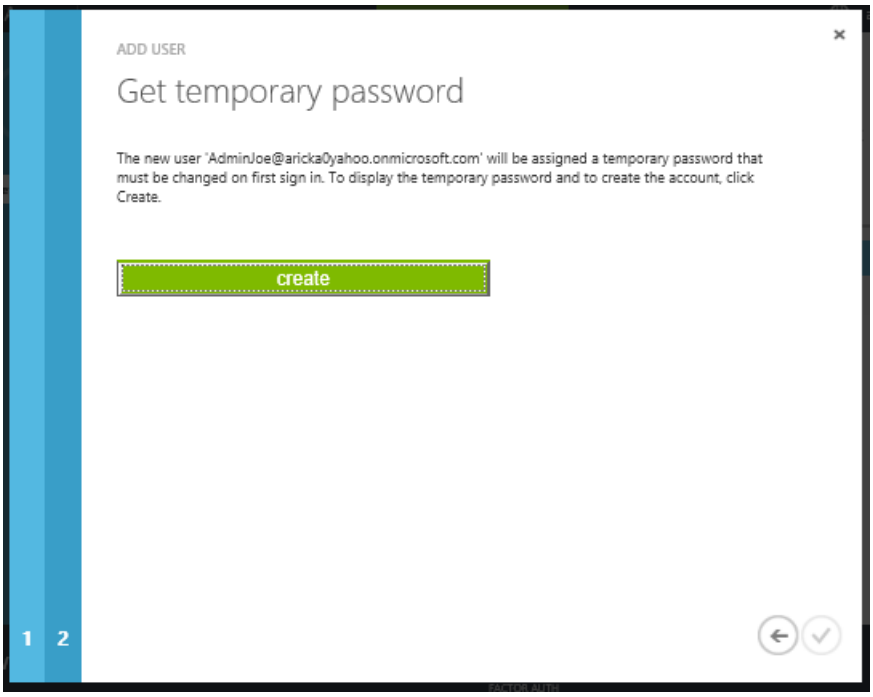
4. Create a new user with the **Global Administrator** role. Click **Users** from the top menu, and then click the **Add User** button on the command bar.

5. In the **Add User** dialog, enter a name for the new user and then click the right arrow.

6. Enter the user name and set the role to **Global Administrator**. Global administrators require an alternate email address for password recovery purposes. After you're finished, click the right arrow.



7. On the next page of the dialog, click **Create**. A temporary password will be created for the new user and displayed in the dialog.



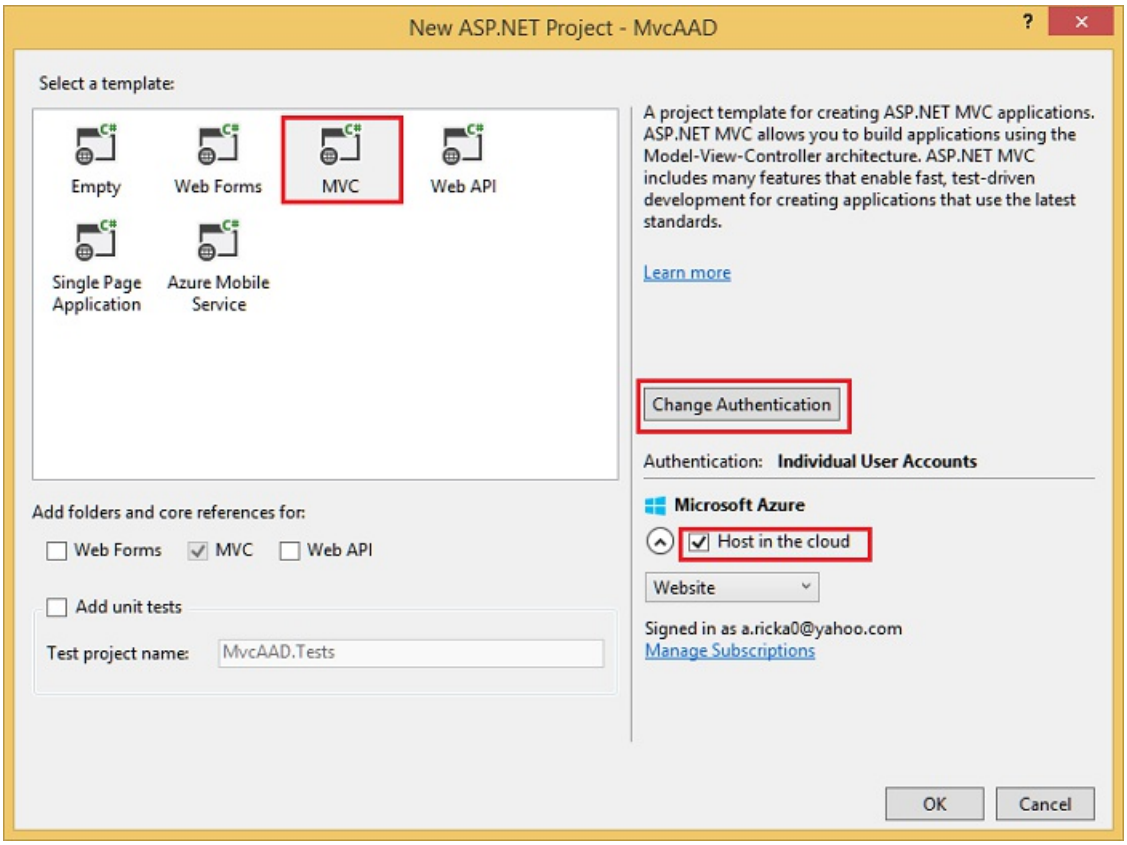
Save the password, you will be required to change the password after the first log in. The following image shows the new admin account. You must use the Azure Active Directory to log into your app, not the Microsoft account also shown on this page.

Create an ASP.NET Application

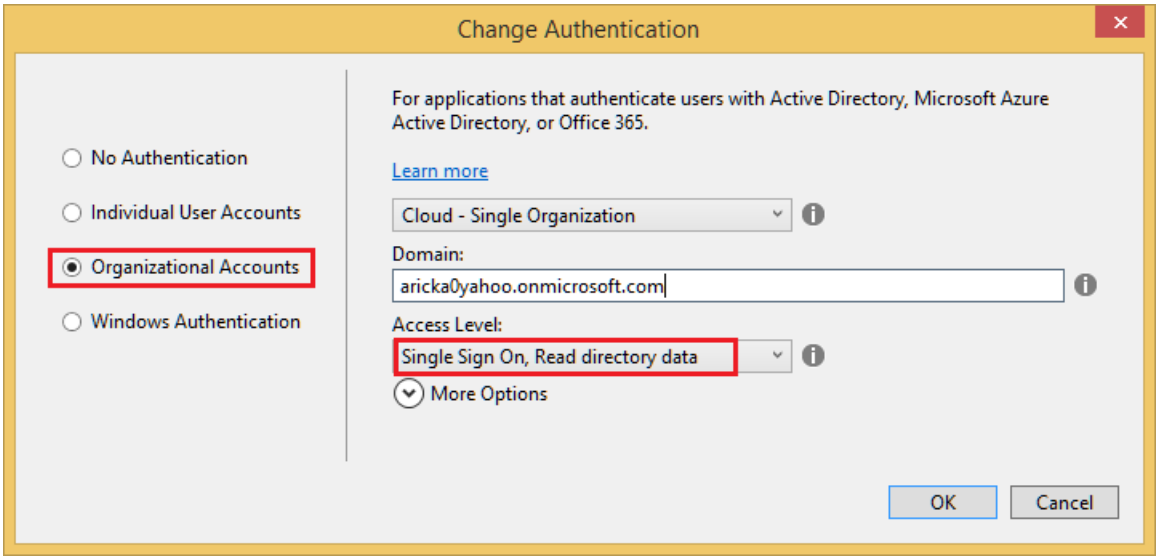
The following steps use Visual Studio Express 2013 for Web, and requires Visual Studio 2013 Update 3.

1. In Visual Studio, click **File** and then **New Project**. On the **New Project** dialog, select the Visual C# Web project from the left menu and click **OK**. You may also want to uncheck the **Add Application Insights to Project** if you don't want the functionality for your application.

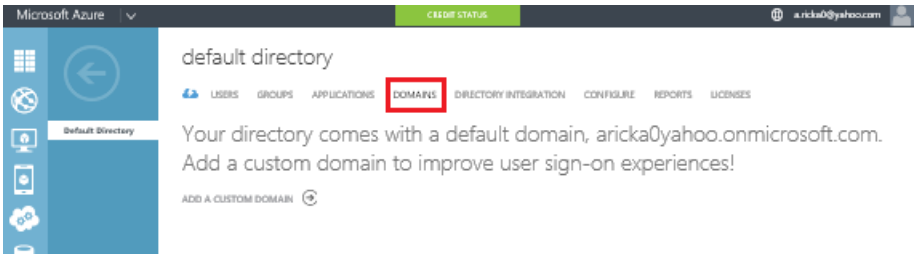
2. In the **New ASP.NET Project** dialog, select **MVC**, and then click **Change Authentication**.



3. On the **Change Authentication** dialog, select **Organizational Accounts**. These options can be used to automatically register your application with Azure AD as well as automatically configure your application to integrate with Azure AD. You don't have to use the **Change Authentication** dialog to register and configure your application, but it makes it much easier. If you are using Visual Studio 2012 for example, you can still manually register the application in the Azure Management Portal and update its configuration to integrate with Azure AD. In the drop-down menus, select **Cloud - Single Organization** and **Single Sign On, Read directory data**. Enter the domain for your Azure AD directory, for example (in the images below) *aricka0yahoo.onmicrosoft.com*, and then click **OK**. You can get the domain name from the Domains tab for the Default Directory on the azure portal (see the next image down).

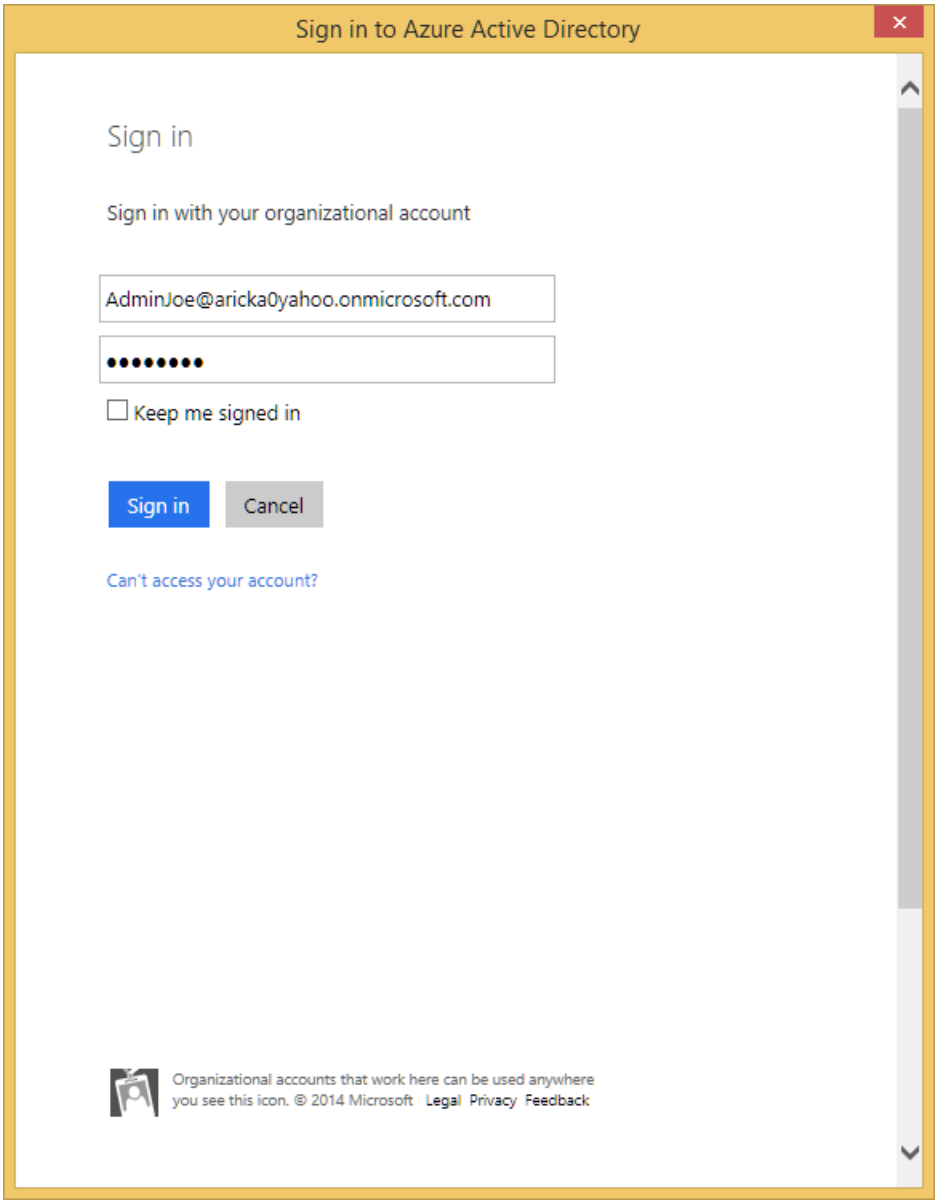


The following image shows the domain name from the Azure portal.



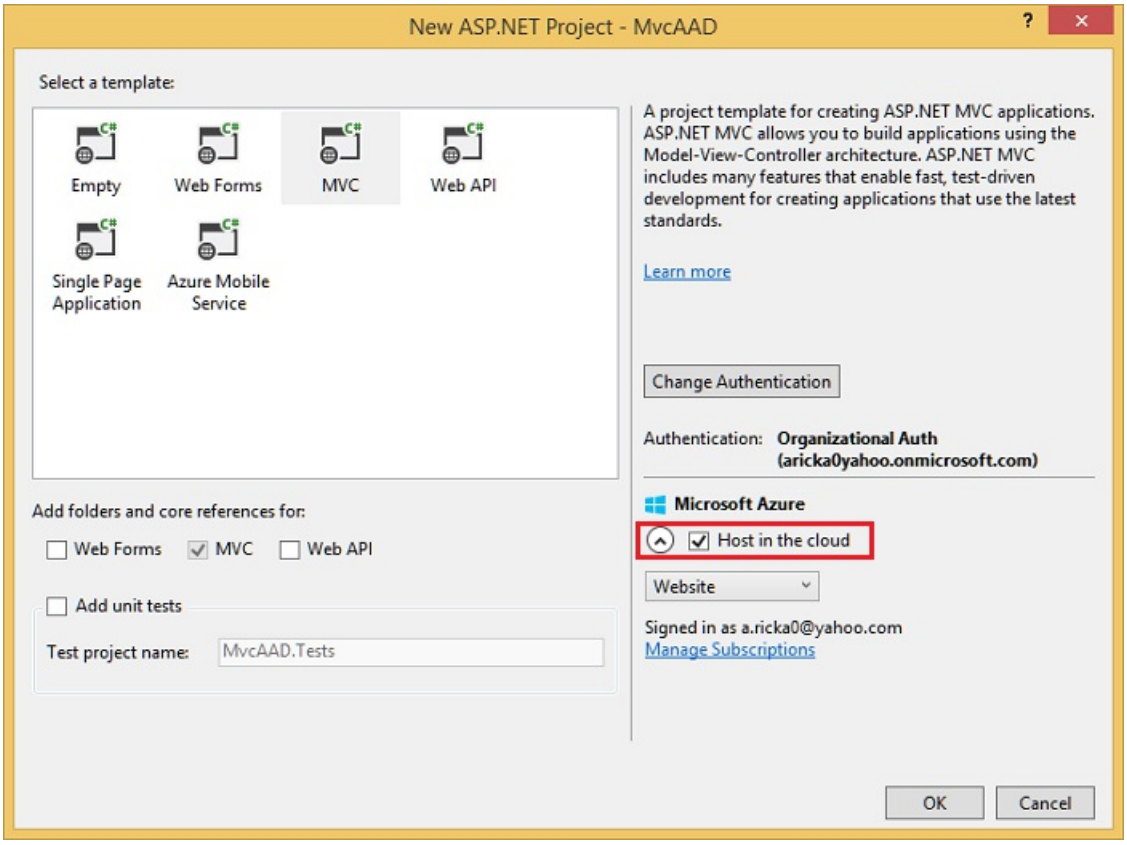
> **NOTE**
>
> You can optionally configure the Application ID URI that will be registered in Azure AD by clicking **More Options**. The App ID URI is the unique identifier for an application, which is registered in Azure AD and used by the application to identify itself when communicating with Azure AD. For more information about the App ID URI and other properties of registered applications, see this topic. By clicking the checkbox below the App ID URI field, you can also choose to overwrite an existing registration in Azure AD that uses the same App ID URI.

4. After clicking **OK**, a sign-in dialog will appear, and you'll need to sign in using a Global Administrator account (not the Microsoft account associated with your subscription). If you created a new Administrator account earlier, you'll be required to change the password and then sign in again using the new password.



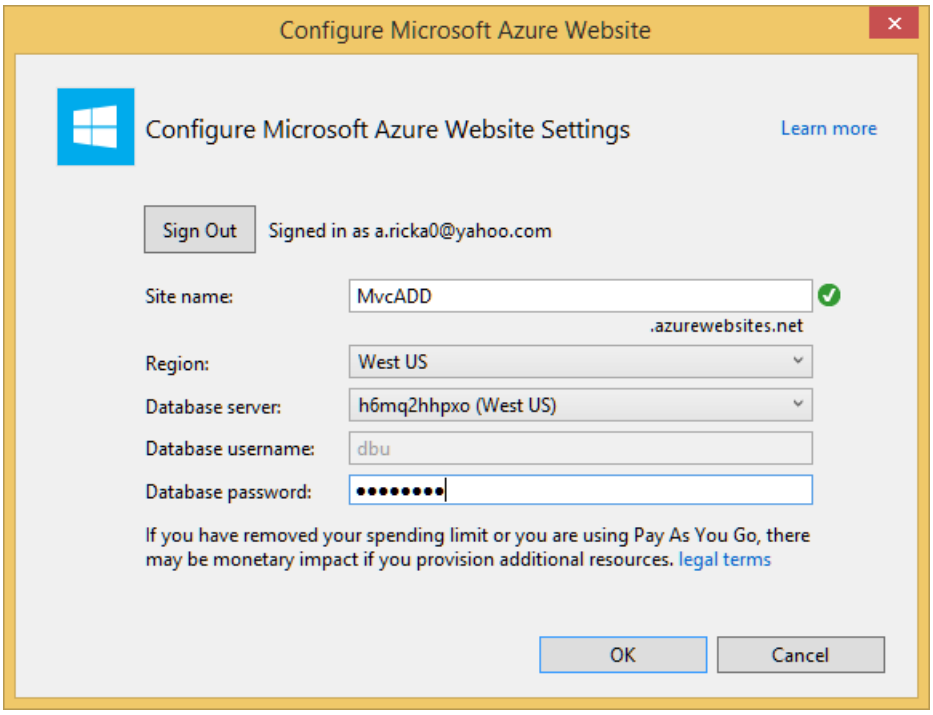
5. After you've successfully authenticated, the **New ASP.NET Project** dialog will show your authentication choice (**Organizational**) and the directory where the new application will be registered

(*aricka0yahoo.onmicrosoft.com* in the image below). Below this information, select the checkbox labeled **Host in the cloud**. If this checkbox is selected, the project will be provisioned as an Azure web app and will be enabled for easy publishing later. Click **OK**.

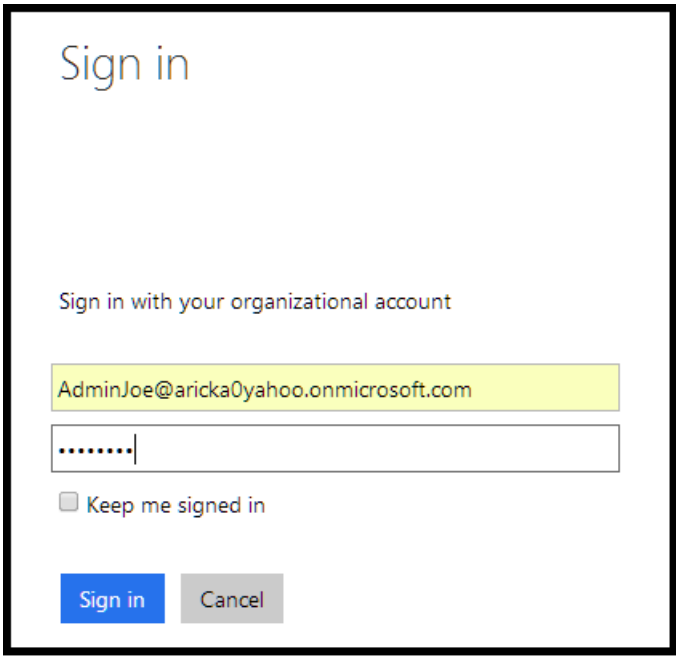


6. The **Configure Azure Website** dialog will appear, using an auto-generated site name and region. Also note the account you're currently signed into in the dialog. You want to make sure that this account is the one that your Azure subscription is attached to, typically a Microsoft account.

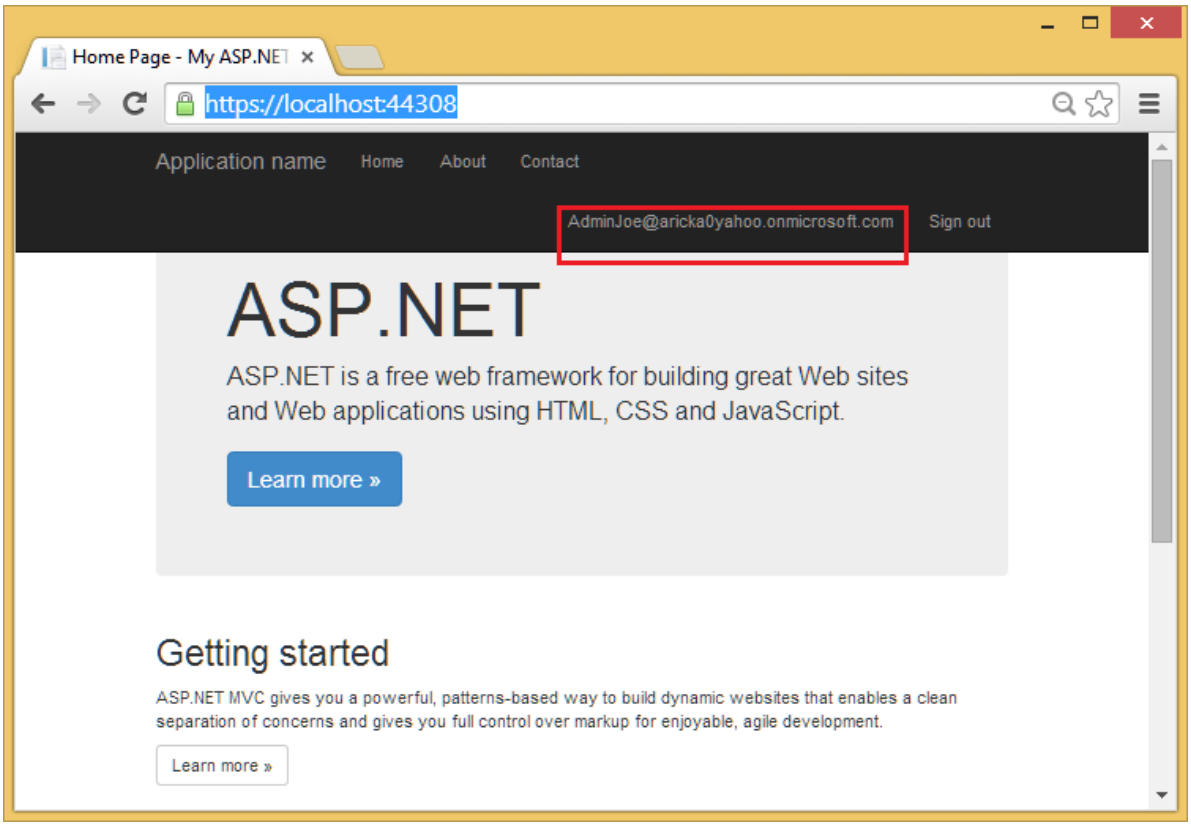
> **NOTE**
>
> This project requires a database. You need to select one of your existing databases, or create a new one. A database is required because the project already uses a local database file to store a small amount of authentication configuration data. When you deploy the application to an Azure Website, this database isn't packaged with the deployment, so you need to choose one that's accessible in the cloud. Click **OK**.



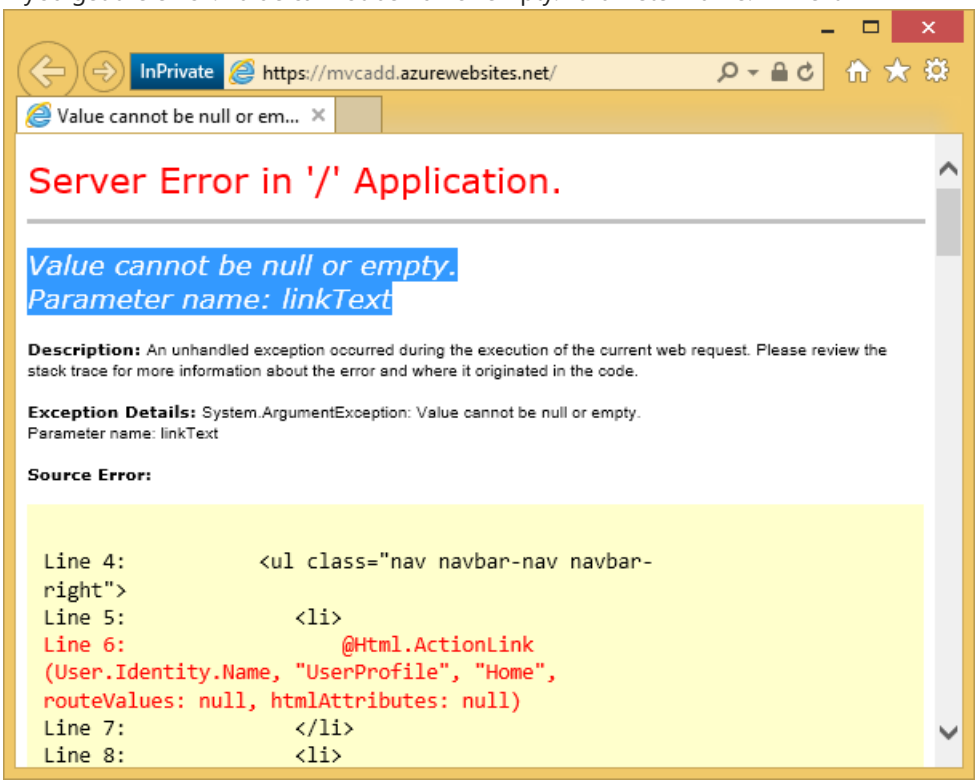
7. The project will be created, and your authentication options and web app options will be automatically configured with the project. Once this process has completed, run the project locally by pressing **^F5**. You will be required to sign in using your organizational account. Provide the username and password for the account you created earlier and click **Sign in**.



8. After successful sign in, the ASP.NET site will show that you've authenticated by displaying the username in the top right corner of the page.



If you get the error: Value cannot be null or empty. Parameter name: linkText



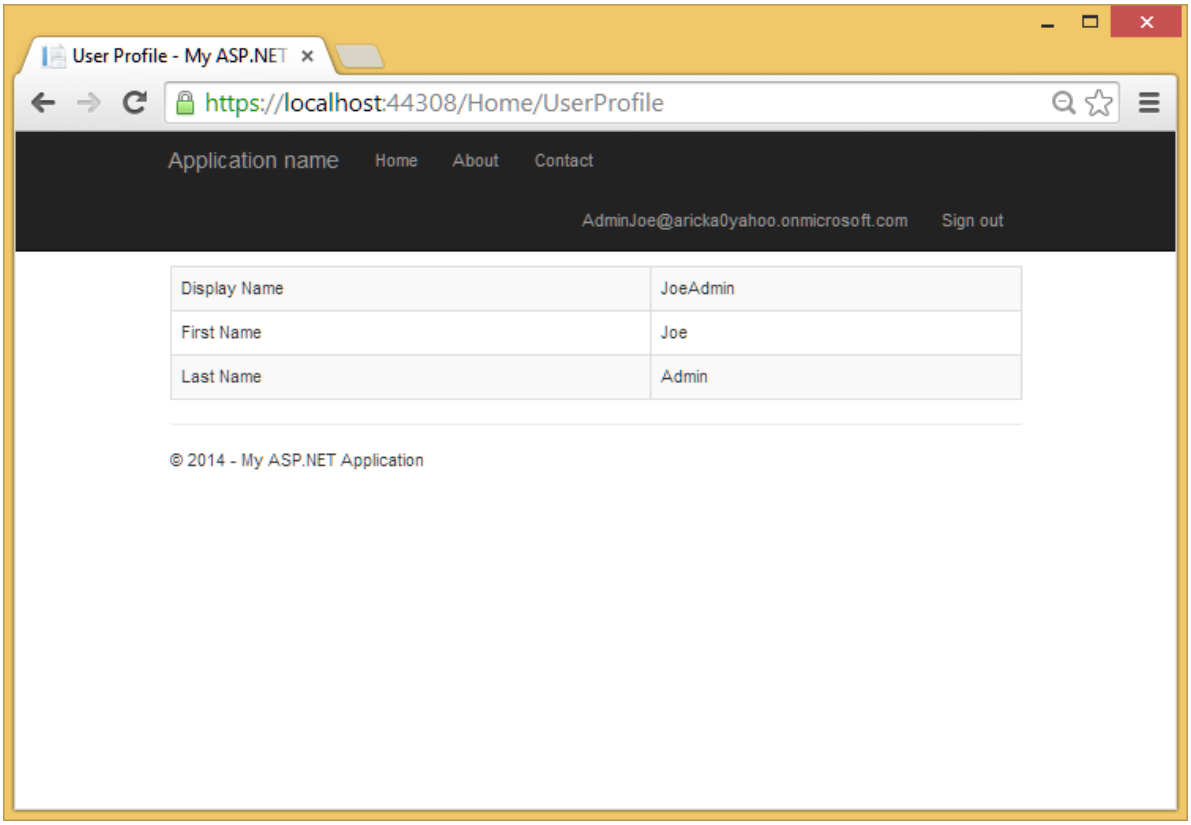
see the debug section at the end of the tutorial.

Basics of the Graph API

The Graph API is the programmatic interface used to perform CRUD and other operations on objects in your Azure AD directory. If you select an Organizational Account option for authentication when creating a new project in Visual Studio 2013, your application will already be configured to call the Graph API. This section briefly shows you how the Graph API works.

1. In your running application, click on the name of the signed-in user at the top right of the page. This will

take you to the User Profile page, which is an action on the Home Controller. You'll notice that the table contains user information about the administrator account you created earlier. This information is stored in your directory, and the Graph API is called to retrieve this information when the page loads.



2. Go back to Visual Studio and expand the **Controllers** folder and then open the **HomeController.cs** file. You'll see a **UserProfile()** action that contains code to retrieve a token and then call the Graph API. This code is duplicated below:

```
[Authorize]
public async Task UserProfile()
{
    string tenantId = ClaimsPrincipal.Current.FindFirst(TenantIdClaimType).Value;

    // Get a token for calling the Azure Active Directory Graph
    AuthenticationContext authContext = new
AuthenticationContext(String.Format(CultureInfo.InvariantCulture, LoginUrl, tenantId));
    ClientCredential credential = new ClientCredential(AppPrincipalId, AppKey);
    AuthenticationResult assertionCredential = authContext.AcquireToken(GraphUrl, credential);
    string authHeader = assertionCredential.CreateAuthorizationHeader();
    string requestUrl = String.Format(
        CultureInfo.InvariantCulture,
        GraphUserUrl,
        HttpUtility.UrlEncode(tenantId),
        HttpUtility.UrlEncode(User.Identity.Name));

    HttpClient client = new HttpClient();
    HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get, requestUrl);
    request.Headers.TryAddWithoutValidation("Authorization", authHeader);
    HttpResponseMessage response = await client.SendAsync(request);
    string responseString = await response.Content.ReadAsStringAsync();
    UserProfile profile = JsonConvert.DeserializeObject<UserProfile>(responseString);

    return View(profile);
}
```

To call the Graph API, you first need to retrieve a token. When the token is retrieved, its string value must be appended in the Authorization header for all subsequent requests to the Graph API. Most of the code above

handles the details of authenticating to Azure AD to get a token, using the token to make a call to the Graph API, and then transforming the response so that it can be presented in the View.

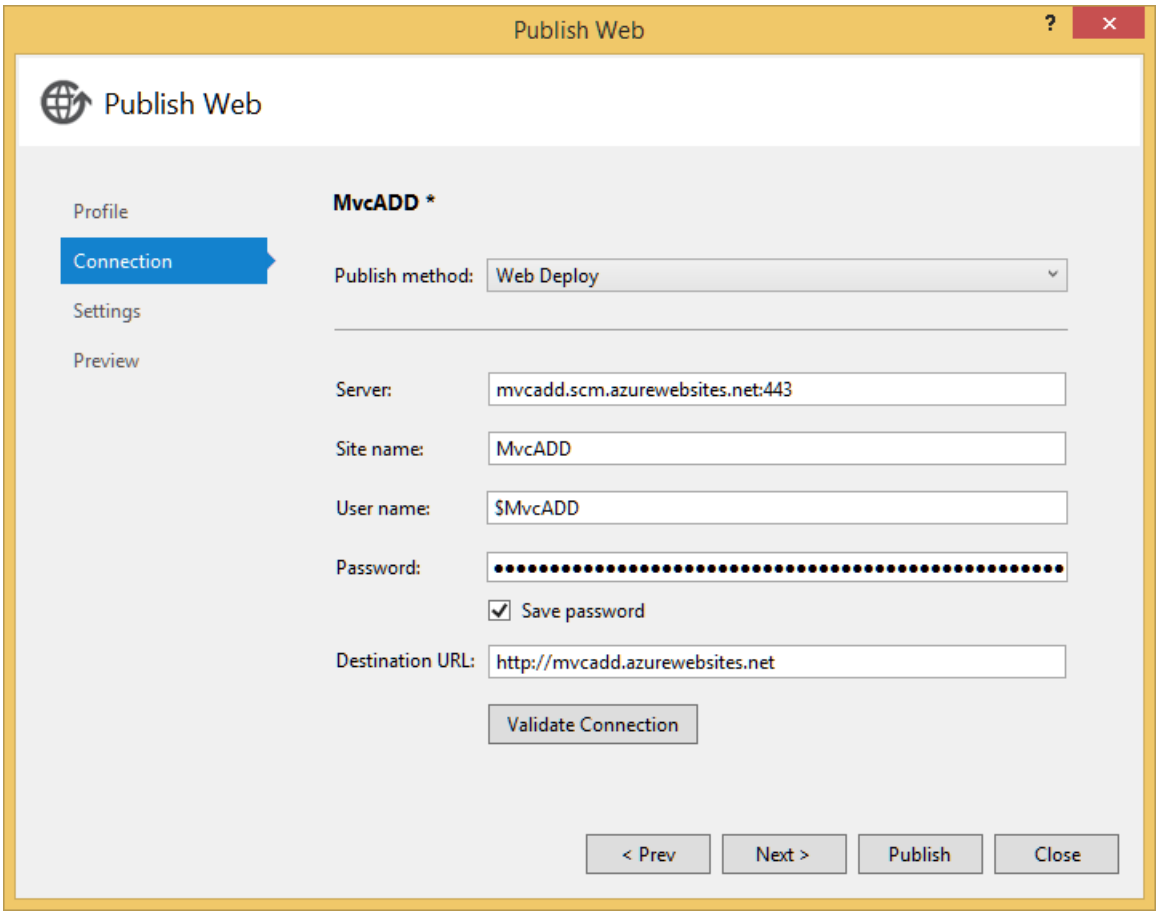
The most relevant portion for discussion is the following highlighted line: `UserProfile profile = JsonConvert.DeserializeObject<UserProfile>(responseString);`. This line represents the name of the user, which has been deserialized from the JSON response and is presented in the View.

You can call the Graph API using HttpClient and handle the raw data yourself, but an easier way is to use the Graph Client Library which is available via NuGet. The Client Library handles the raw HTTP requests and the transformation of the returned data for you, and makes it much easier to work with the Graph API in a .NET environment. See the related Graph API code samples on GitHub.

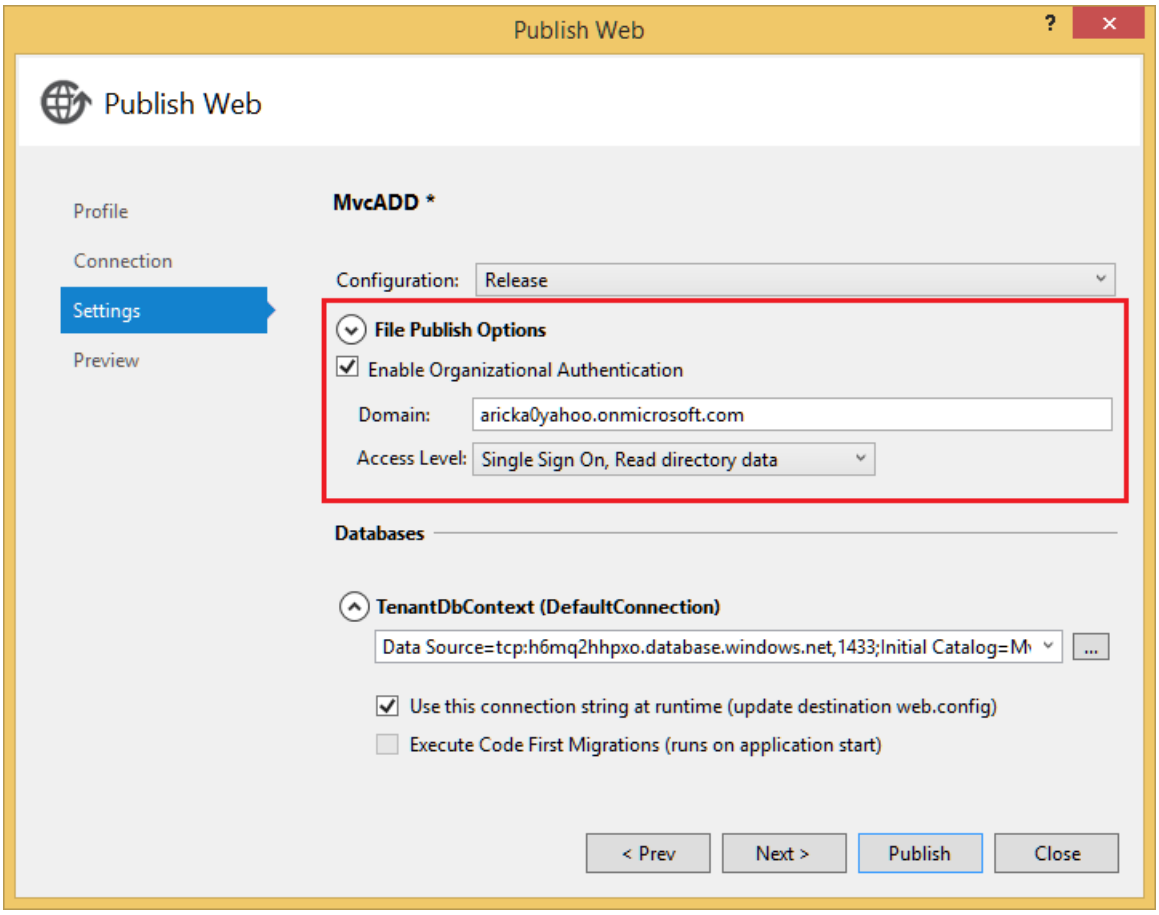
Deploy the Application to Azure

The following steps will show you how to deploy the application to Azure. In the earlier steps, you connected your new project with a web app on Azure, so it's ready to be published in just a few steps.

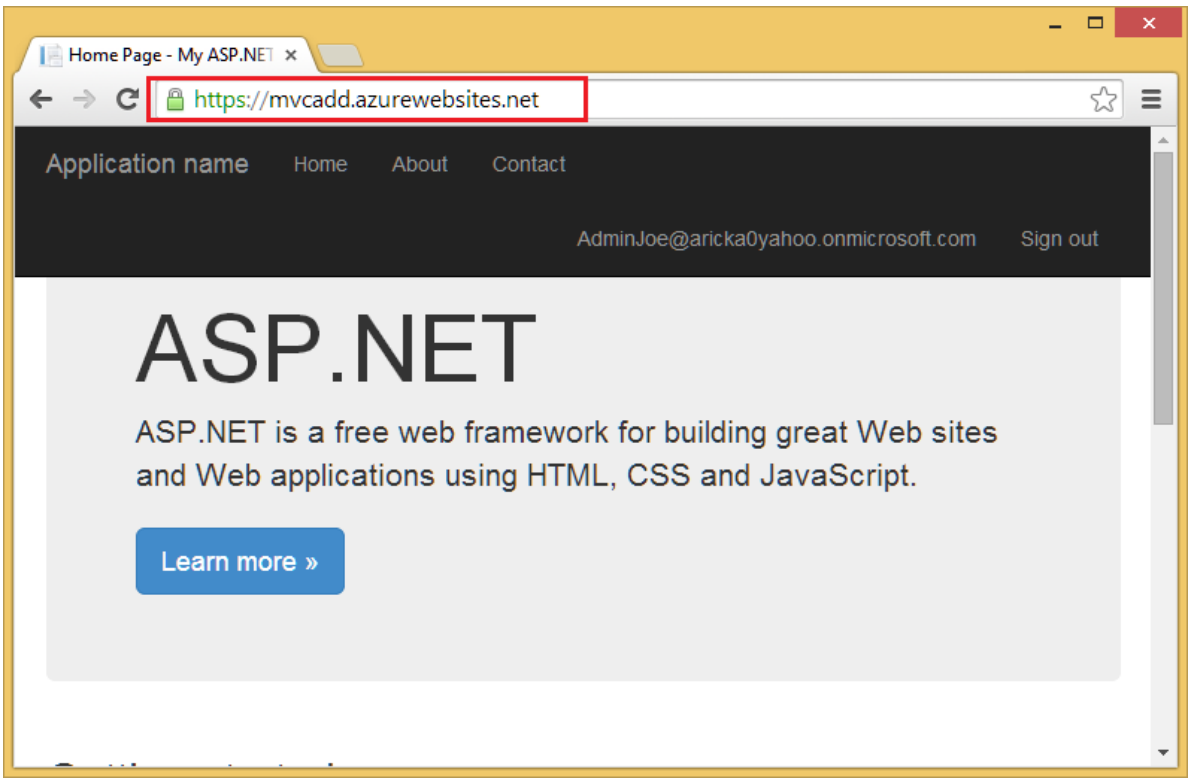
1. In Visual Studio, right-click on the project and select **Publish**. The **Publish Web** dialog will appear with each setting already configured. Click on the **Next** button to go to the **Settings** page. You may be prompted to authenticate; make sure you authenticate using your Azure subscription account (typically a Microsoft account) and not the organizational account you created earlier.



2. Check the **Enable Organizational Authentication** option. In the **Domain** field, enter the domain for your directory. From the **Access Level** drop-down, select **Single Sign On, Read directory data**. You'll notice that the previous database you used is already populated in the **Databases** section. Click **Publish**.

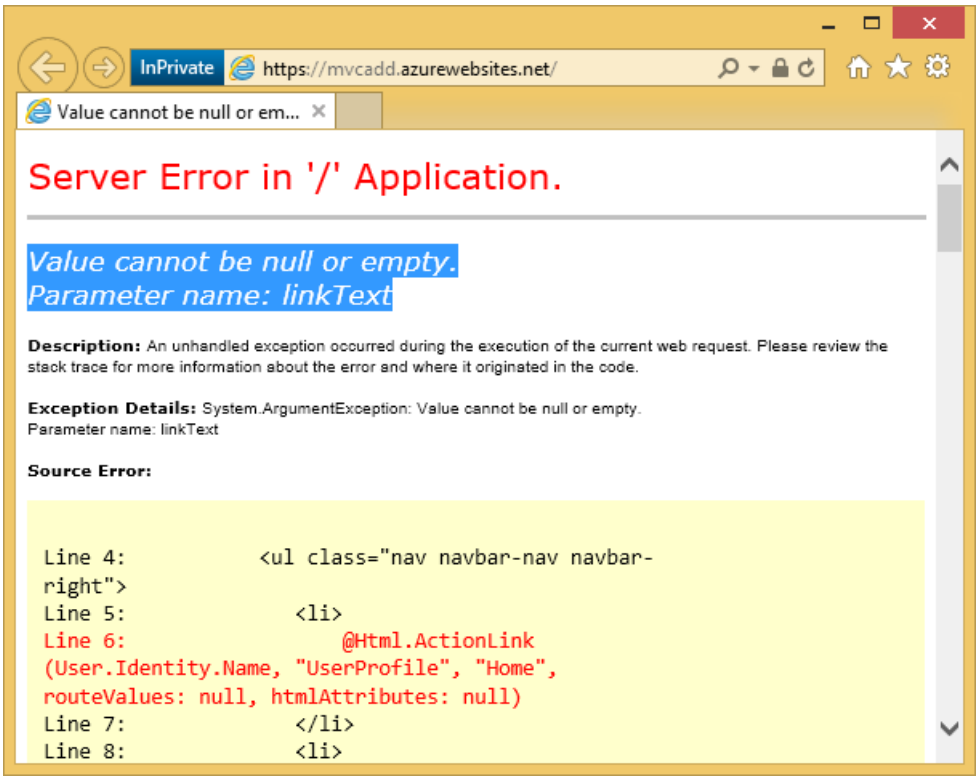


3. Visual Studio will begin deploying your website, and then a new browser window will appear. You may be prompted to authenticate to your directory once again. Once you've authenticated, you'll be redirected to your newly published website on Azure.



Debugging the app

If you get the following error: Value cannot be null or empty. Parameter name: linkText



Replace the code in the *Views\Shared_LoginPartial.cshtml* file with the following:

```
@{
   var user = "Null User";
   if (!String.IsNullOrEmpty(User.Identity.Name))
   {
      user = User.Identity.Name;
   }

}

@if (Request.IsAuthenticated)
{
    <text>
         <ul class="nav navbar-nav navbar-right">
            <li>
                @*@Html.ActionLink(User.Identity.Name, "UserProfile", "Home", routeValues: null,
htmlAttributes: null)*@
               @Html.ActionLink(user, "UserProfile", "Home", routeValues: null, htmlAttributes: null)
            </li>
            <li>
                @Html.ActionLink("Sign out", "SignOut", "Account")
            </li>
        </ul>
    </text>
}
else
{
     <ul class="nav navbar-nav navbar-right">
        <li>@Html.ActionLink("Sign in", "Index", "Home", routeValues: null, htmlAttributes: new { id =
"loginLink" })</li>
    </ul>
}
```

After running the app, if the logged in user shows "Null User", sign out, and sign back in with the Active Directory account you created earlier.

An excellent tutorial to follow is Rick Rainey's Deep Dive: Azure Websites and Organizational Authentication using Azure AD.

More Information

- Deep Dive: Azure Websites and Organizational Authentication using Azure AD
- Azure AD Graph API Overview
- Authentication Scenarios in Azure AD
- Azure AD Code Samples on GitHub

ASP.NET Identity: Using MySQL Storage with an EntityFramework MySQL Provider (C#)

2/20/2020 • 7 minutes to read • Edit Online

by Maurycy Markowski, Raquel Soares De Almeida, Robert McMurray

> This tutorial shows you how to replace the default data storage mechanism for **ASP.NET Identity** with EntityFramework (SQL client provider) with a MySQL provider.

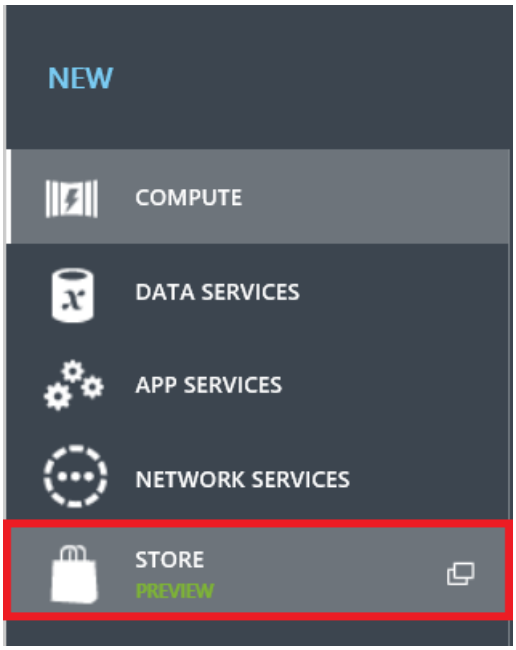
The following topics will be covered in this tutorial:

- Creating a MySQL database on Azure
- Creating an MVC application using Visual Studio 2013 MVC template
- Configuring EntityFramework to work with a MySQL database provider
- Running the application to verify the results

At the end of this tutorial, you will have an MVC application with the ASP.NET Identity store that is using a MySQL database that is hosted in Azure.

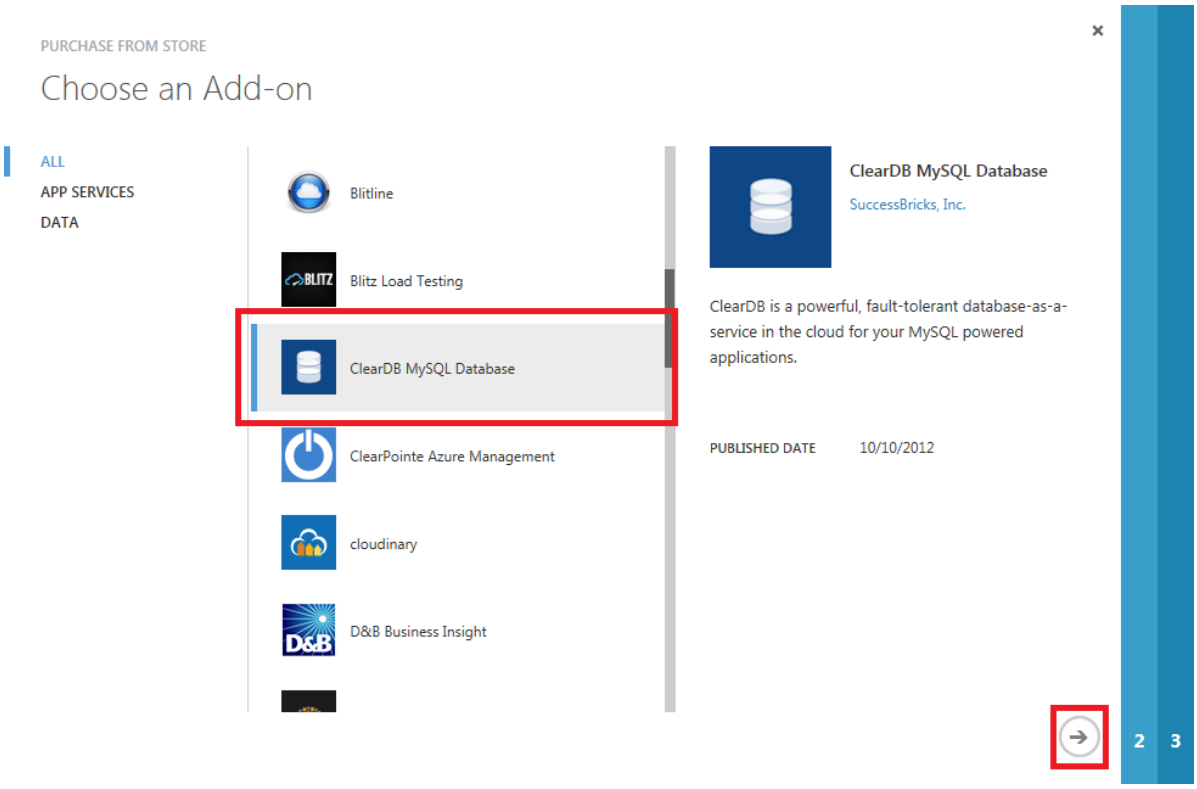
Creating a MySQL database instance on Azure

1. Log in to the Azure Portal.
2. Click **NEW** at the bottom of the page, and then select **STORE**:



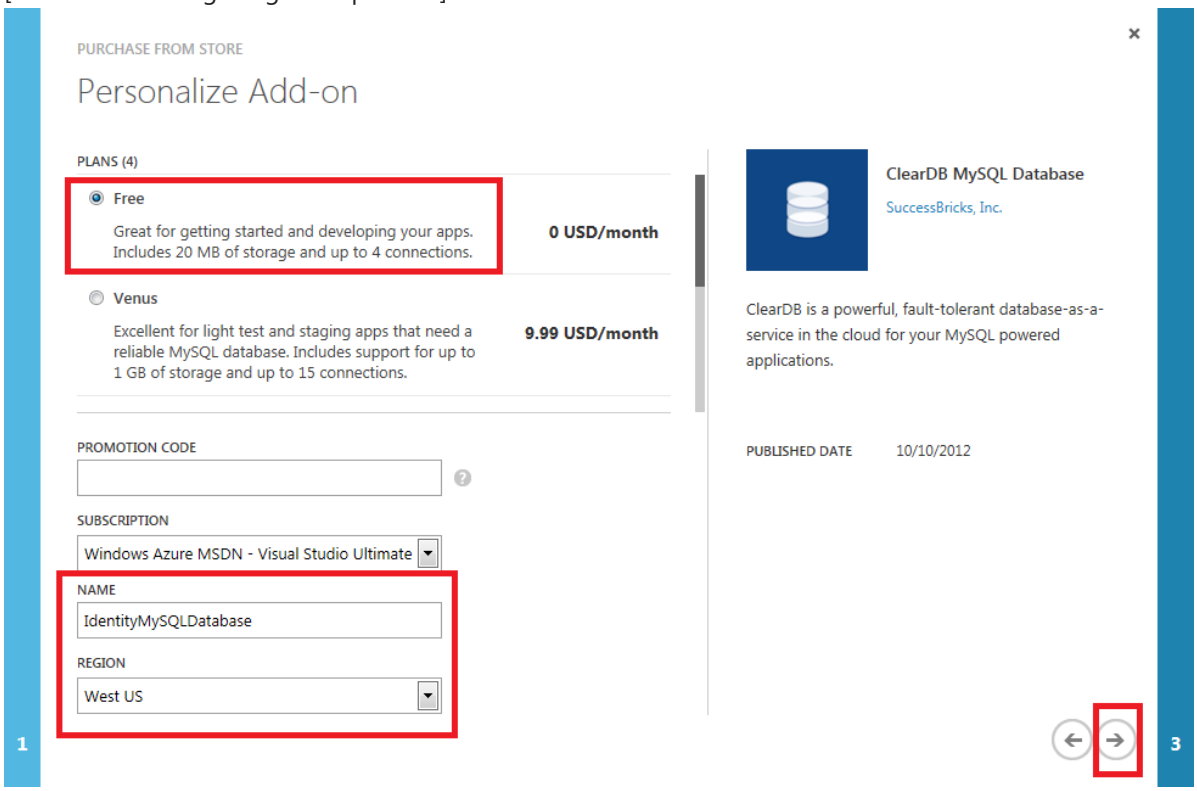
3. In the **Choose and Add-on** wizard, select **ClearDB MySQL Database**, and then click the **Next** arrow at the bottom of the frame:

   [Click the following image to expand it.]



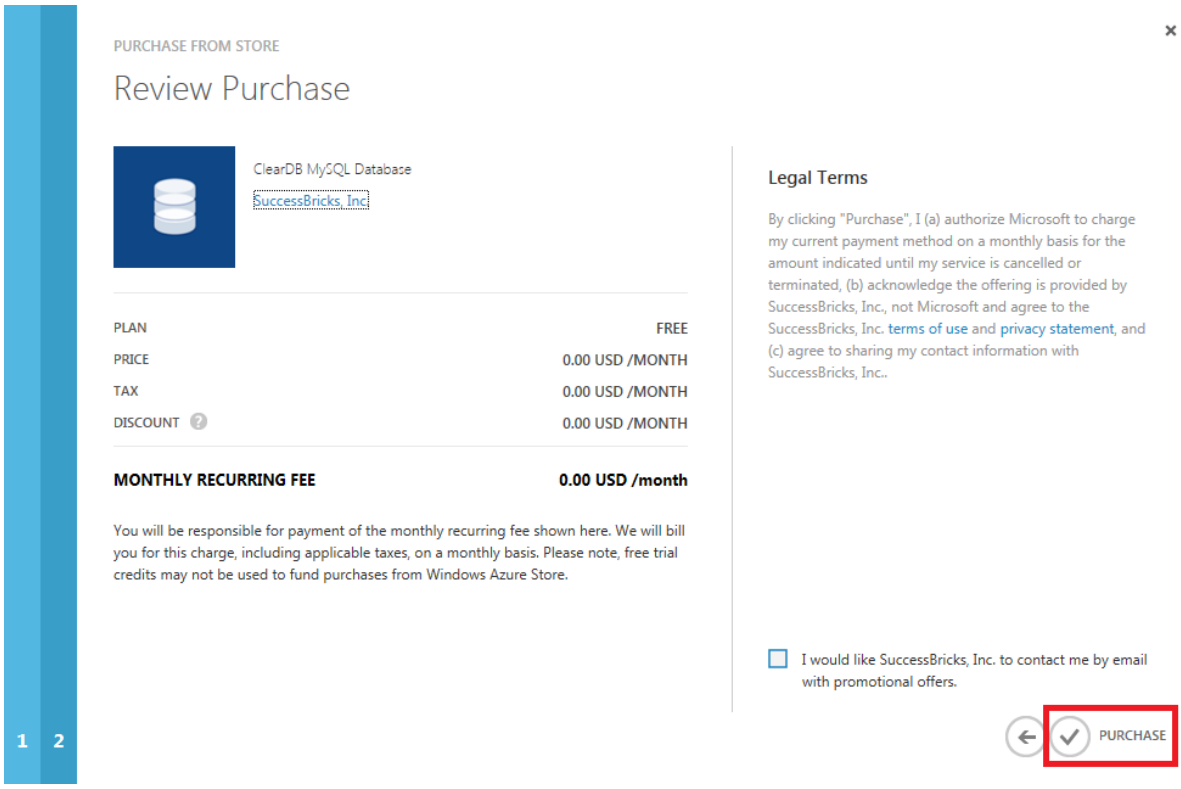
4. Keep the default **Free** plan, change the **NAME** to **IdentityMySQLDatabase**, select the region that is nearest to you, and then click the **Next** arrow at the bottom of the frame:

   [Click the following image to expand it.]



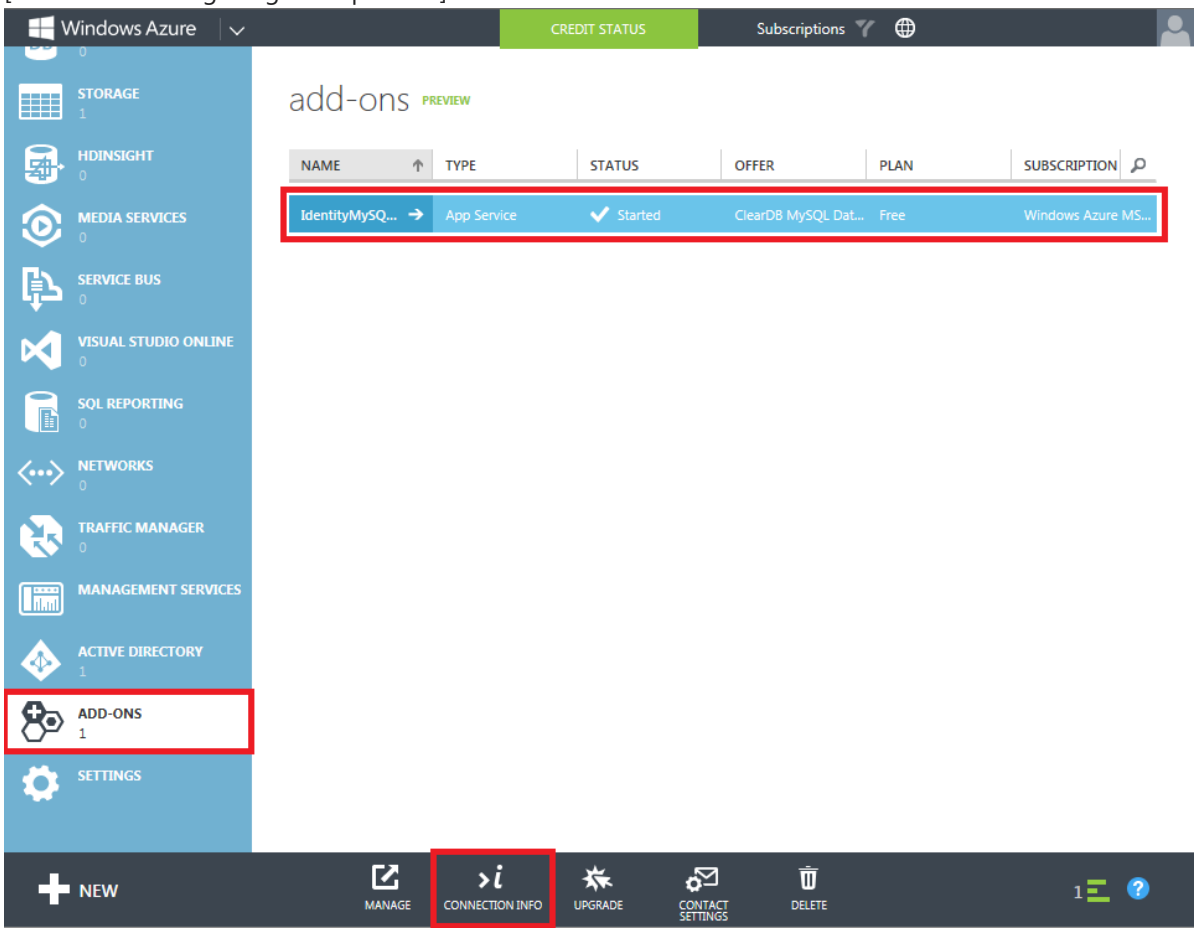
5. Click the **PURCHASE** checkmark to complete the database creation.

   [Click the following image to expand it.]



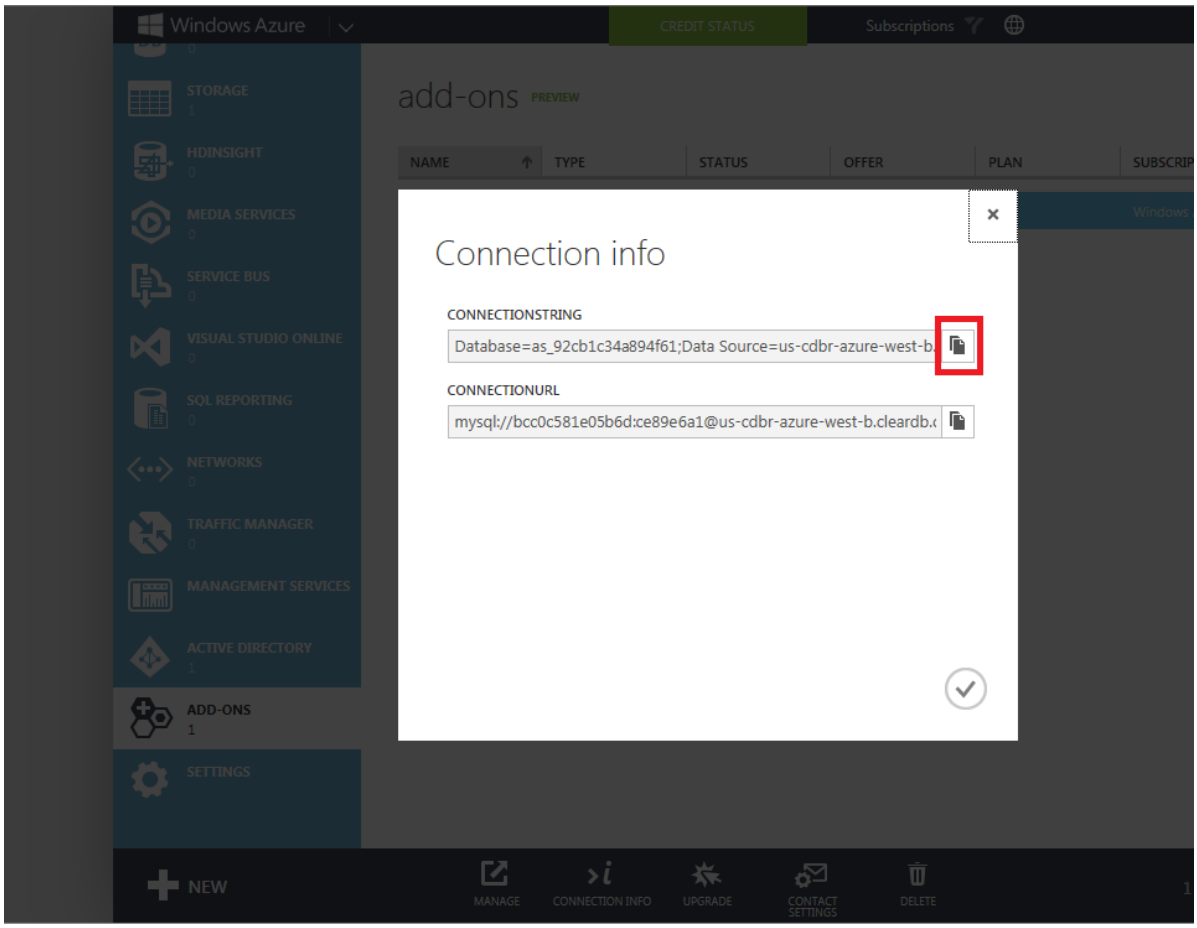
6. After your database has been created, you can manage it from the **ADD-ONS** tab in the management portal. To retrieve the connection information for your database, click **CONNECTION INFO** at the bottom of the page:

   [Click the following image to expand it.]



7. Copy the connection string by clicking on the copy button by the **CONNECTIONSTRING** field and save it; you will use this information later in this tutorial for your MVC application:

   [Click the following image to expand it.]

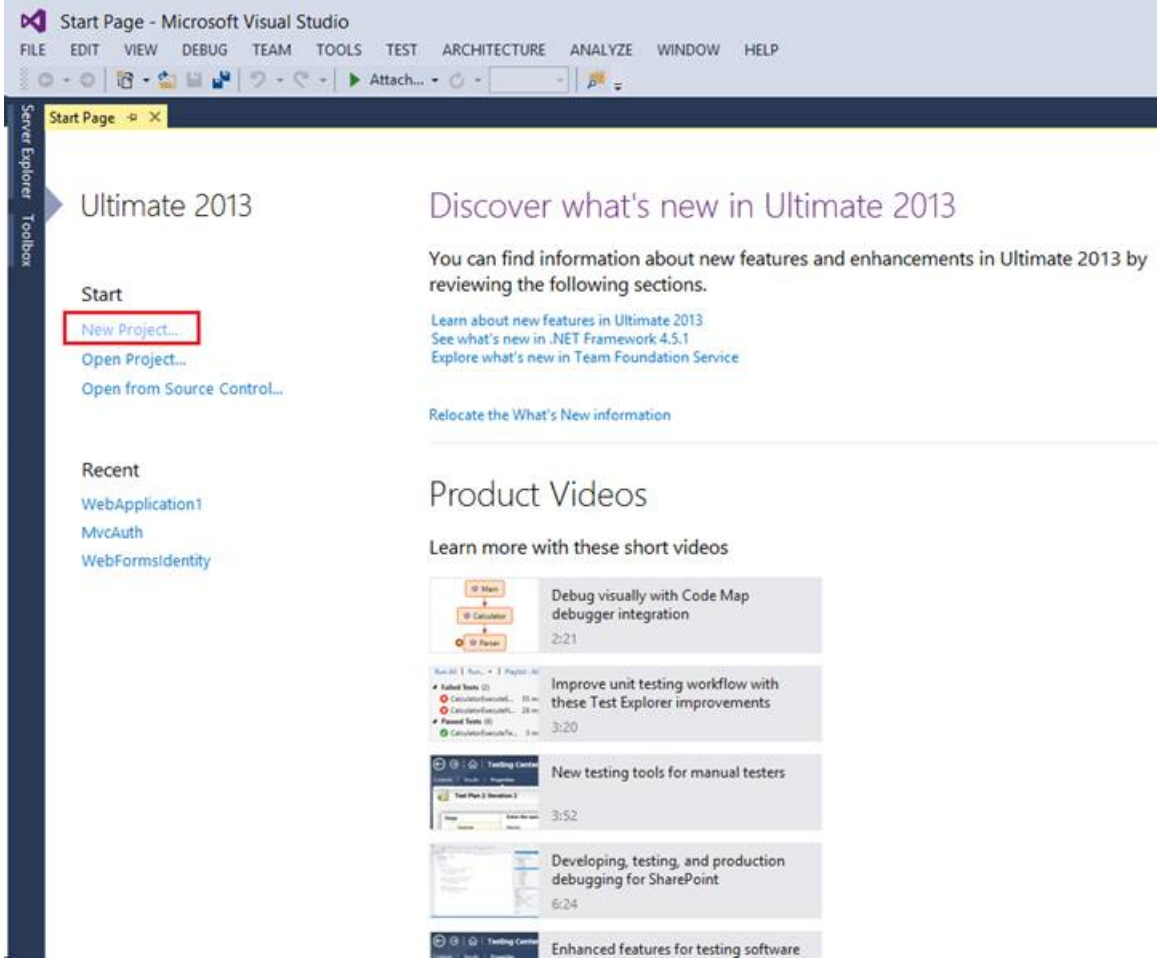


Creating an MVC application project

To complete the steps in this section of the tutorial, you will first need to install Visual Studio Express 2013 for Web or Visual Studio 2013. Once Visual Studio has been installed, use the following steps to create a new MVC application project:

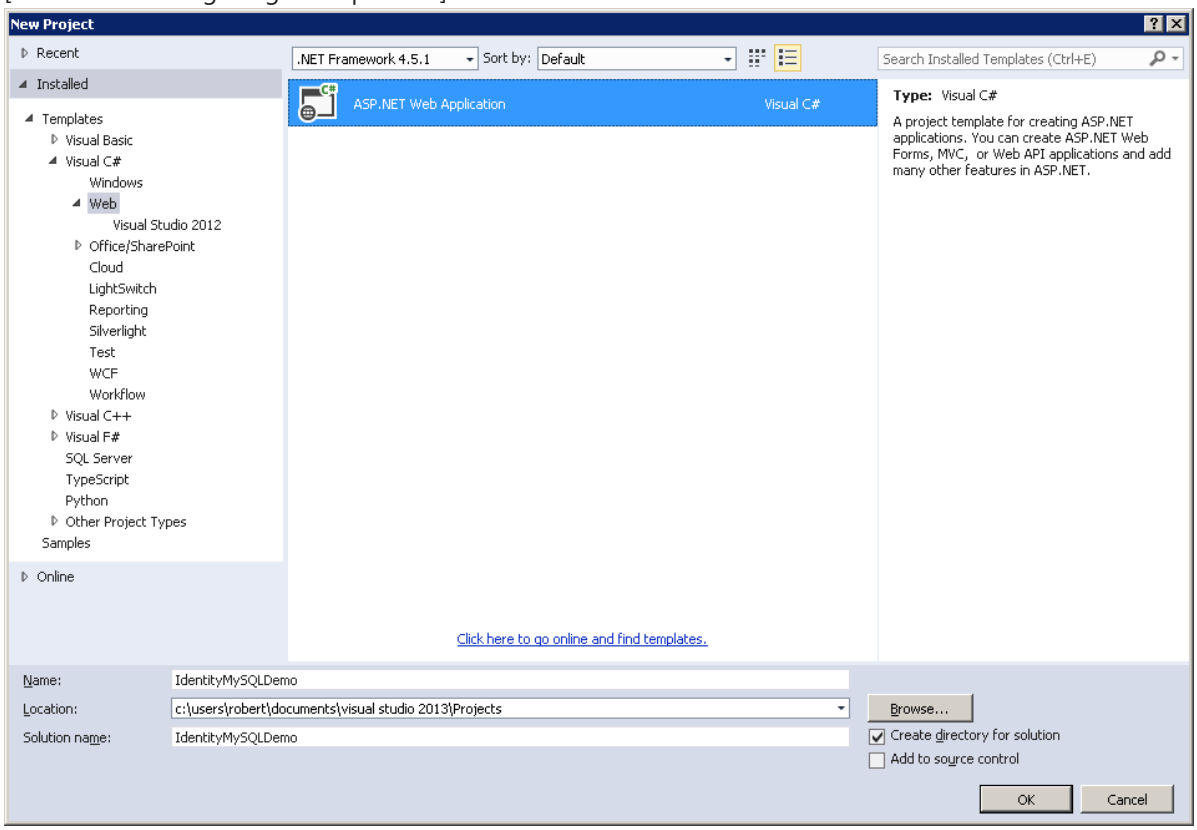
1. Open Visual Studio 2103.
2. Click **New Project** from the **Start** page, or you can click the **File** menu and then **New Project**:

   [Click the following image to expand it.]



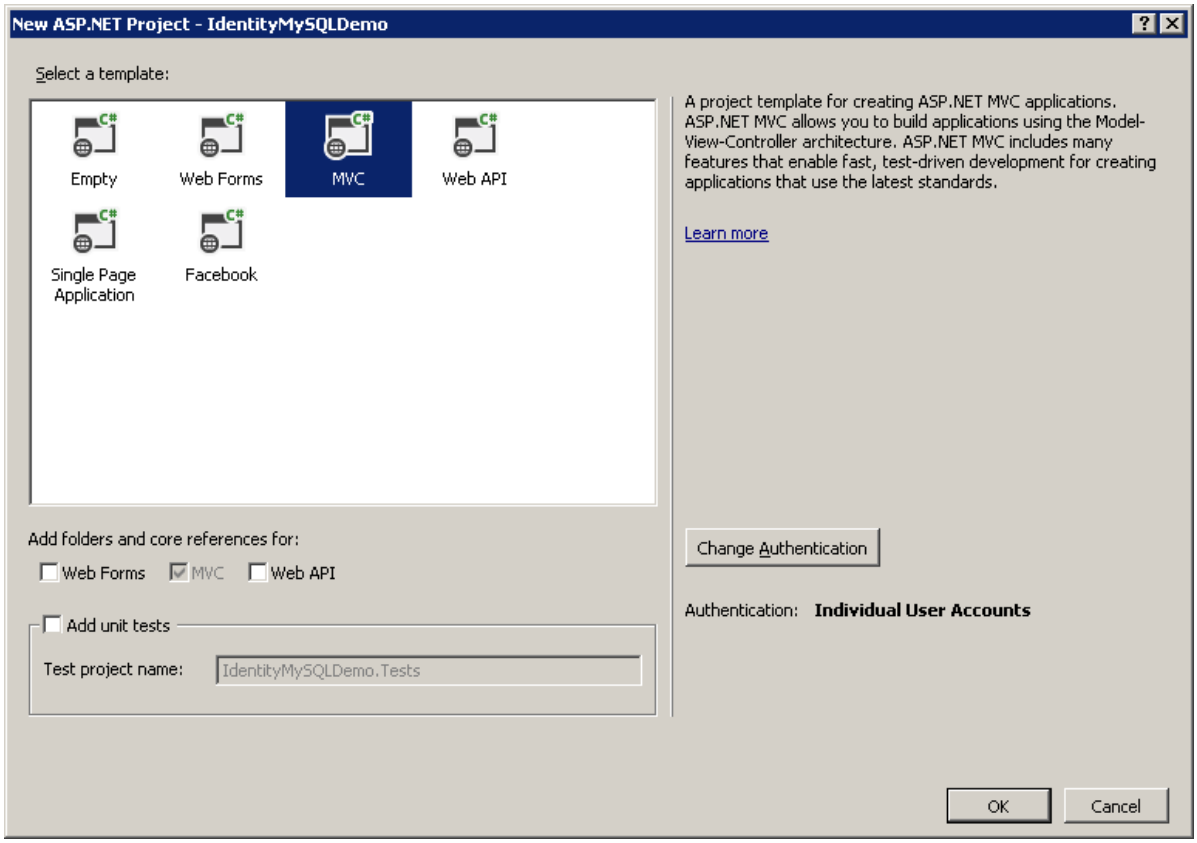
3. When the **New Project** dialog box is displayed, expand **Visual C#** in the list of templates, then click **Web**, and select **ASP.NET Web Application**. Name your project **IdentityMySQLDemo** and then click **OK**:

   [Click the following image to expand it.]



4. In the **New ASP.NET Project** dialog, select the **MVC** templatewith the default options; this will configure **Individual User Accounts** as the authentication method. Click **OK**:

[Click the following image to expand it.]



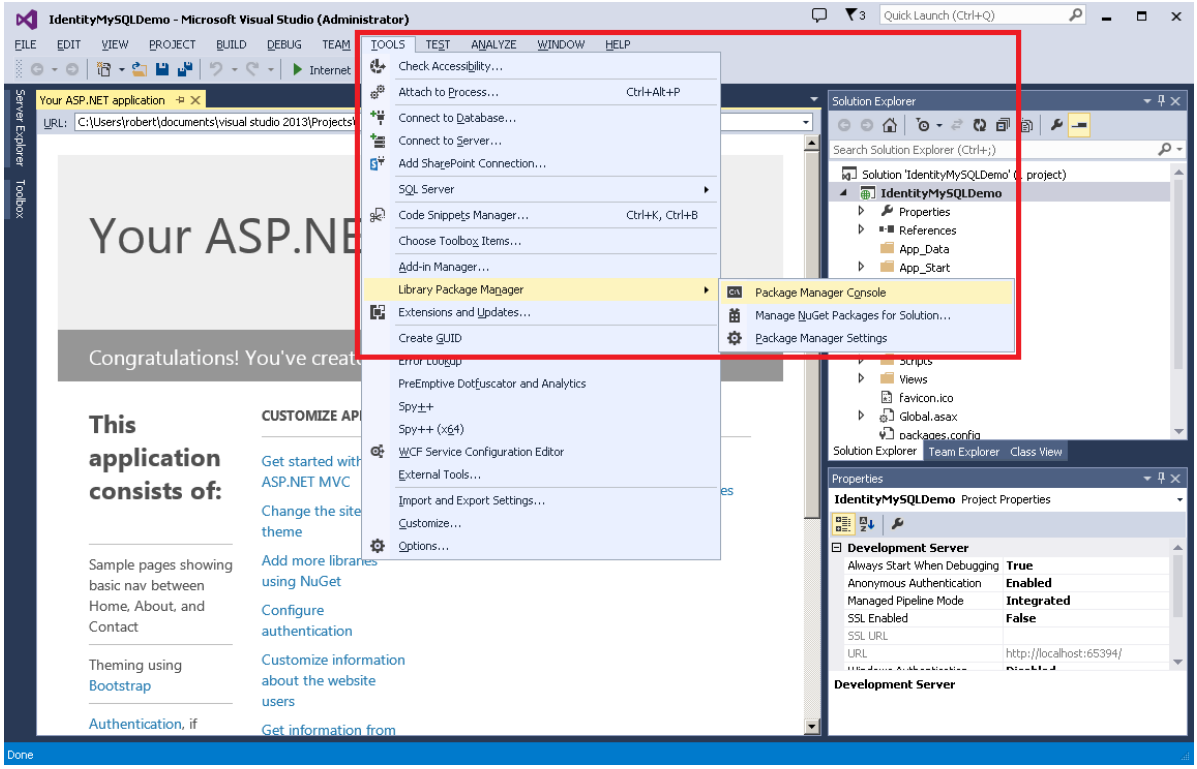
Configure EntityFramework to work with a MySQL database

Update the Entity Framework assembly for your project

The MVC application that was created from the Visual Studio 2013 template contains a reference to the EntityFramework 6.0.0 package, but there have been updates to that assembly since its release which contain significant performance improvements. In order to use these latest updates in your application, use the following steps.

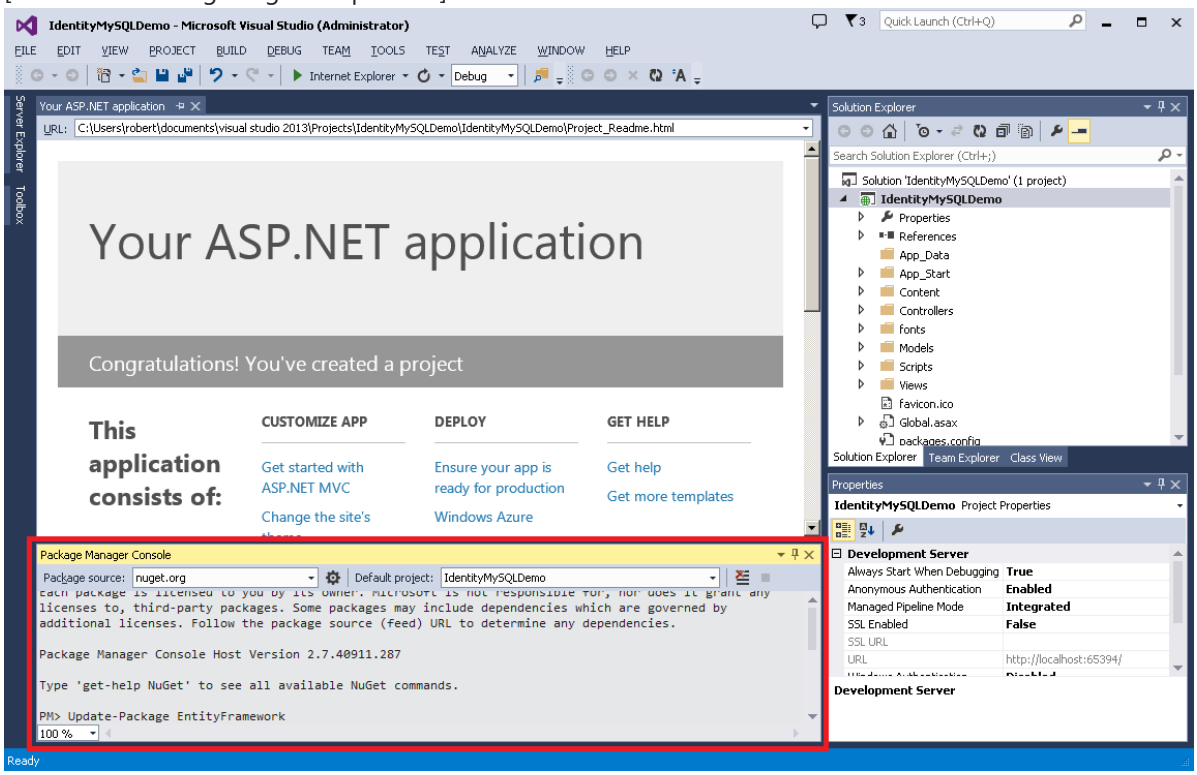
1. Open your MVC project in Visual Studio.
2. Click **Tools**, then click **NuGet Package Manager**, and then click **Package Manager Console**:

   [Click the following image to expand it.]



3. The **Package Manager Console** will appear in the bottom section of Visual Studio. Type "**Update-Package EntityFramework**" and press Enter:

   [Click the following image to expand it.]

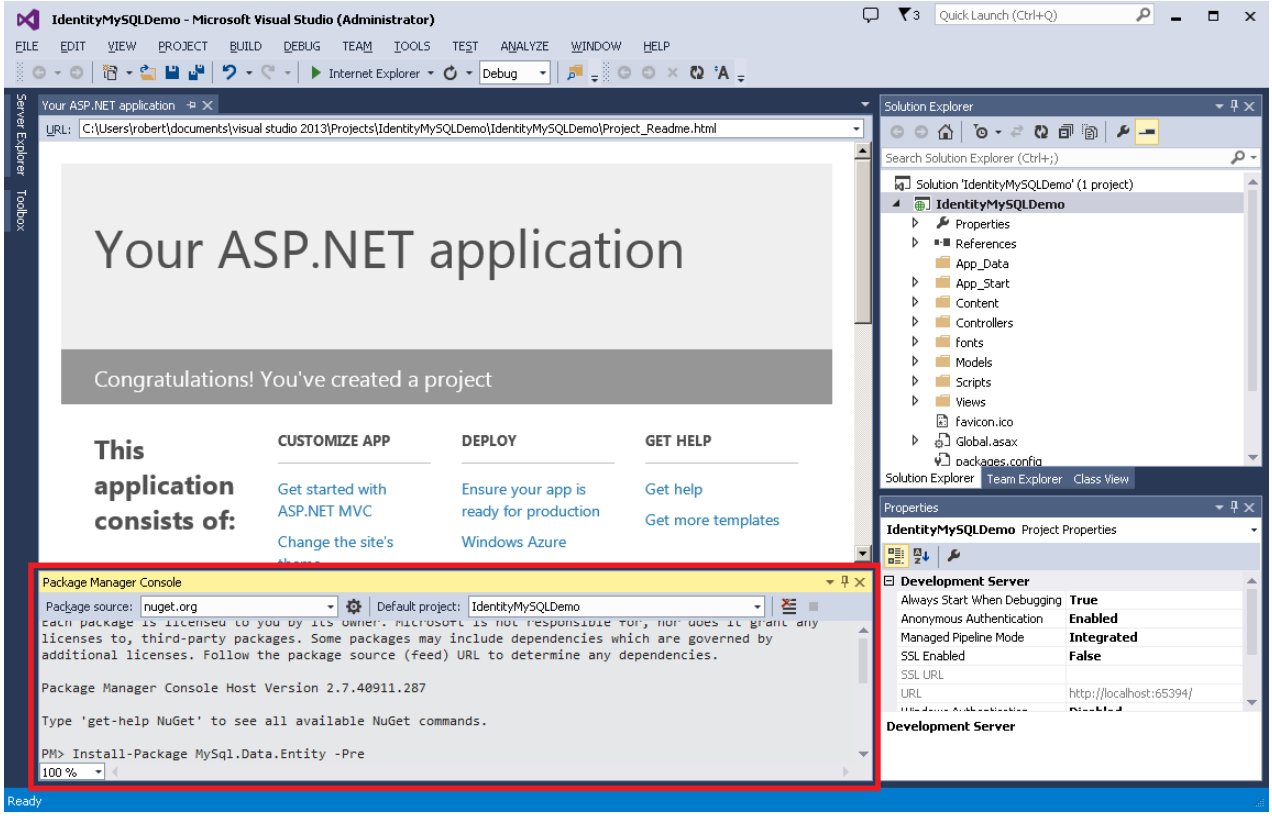


Install the MySQL provider for EntityFramework

In order for EntityFramework to connect to MySQL database, you need to install a MySQL provider. To do so, open the **Package Manager Console** and type "**Install-Package MySql.Data.Entity -Pre**", and then press Enter.

> **NOTE**
>
> This is a pre-release version of the assembly, and as such it may contain bugs. You should not use a pre-release version of the provider in production.

[Click the following image to expand it.]



Making project configuration changes to the Web.config file for your application

In this section you will configure the Entity Framework to use the MySQL provider that you just installed, register the MySQL provider factory, and add your connection string from Azure.

> **NOTE**
>
> The following examples contain a specific assembly version for MySql.Data.dll. If the assembly version changes, you will need to modify the appropriate configuration settings with the correct version.

1. Open the Web.config file for your project in Visual Studio 2013.

2. Locate the following configuration settings, which define the default database provider and factory for the Entity Framework:

```
<entityFramework>
  <defaultConnectionFactory
      type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory, EntityFramework">
    <parameters>
      <parameter value="v11.0" />
    </parameters>
  </defaultConnectionFactory>
  <providers>
    <provider
      invariantName="System.Data.SqlClient"
      type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
  </providers>
</entityFramework>
```

3. Replace those configuration settings with the following, which will configure the Entity Framework to use the MySQL provider:

```
<entityFramework>
  <providers>
    <provider invariantName="MySql.Data.MySqlClient"
      type="MySql.Data.MySqlClient.MySqlProviderServices, MySql.Data.Entity"/>
  </providers>
</entityFramework>
<system.data>
  <DbProviderFactories>
    <remove invariant="MySql.Data.MySqlClient"></remove>
    <add name="MySQL Data Provider"
      invariant="MySql.Data.MySqlClient"
      description=".Net Framework Data Provider for MySQL"
      type="MySql.Data.MySqlClient.MySqlClientFactory, MySql.Data, Version=6.7.2.0"/>
  </DbProviderFactories>
</system.data>
```

4. Locate the <connectionStrings> section and replace it with the following code, which will define the connection string for your MySQL database that is hosted on Azure (note that providerName value has also been changed from the original):

```
<connectionStrings>
  <add name="DefaultConnection"
    providerName="MySql.Data.MySqlClient"
    connectionString="[Insert your ConnectionString from Azure here]"/>
</connectionStrings>
```

Adding custom MigrationHistory context

Entity Framework Code First uses a **MigrationHistory** table to keep track of model changes and to ensure the consistency between the database schema and conceptual schema. However, this table does not work for MySQL by default because the primary key is too large. To remedy this situation, you will need to shrink the key size for that table. To do so, use the following steps:

1. The schema information for this table is captured in a **HistoryContext**, which can be modified as any other **DbContext**. To do so, add a new class file named **MySqlHistoryContext.cs** to the project, and replace its contents with the following code:

```
using System.Data.Common;
using System.Data.Entity;
using System.Data.Entity.Migrations.History;

namespace IdentityMySQLDemo
{
  public class MySqlHistoryContext : HistoryContext
  {
    public MySqlHistoryContext(
      DbConnection existingConnection,
      string defaultSchema)
    : base(existingConnection, defaultSchema)
    {
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
      base.OnModelCreating(modelBuilder);
      modelBuilder.Entity<HistoryRow>().Property(h => h.MigrationId).HasMaxLength(100).IsRequired();
      modelBuilder.Entity<HistoryRow>().Property(h => h.ContextKey).HasMaxLength(200).IsRequired();
    }
  }
}
```

2. Next you will need to configure Entity Framework to use the modified **HistoryContext**, rather than default one. This can be done by leveraging code-based configuration features. To do so, add new class file named **MySqlConfiguration.cs** to your project and replace its contents with:

```
using System.Data.Entity;

namespace IdentityMySQLDemo
{
  public class MySqlConfiguration : DbConfiguration
  {
    public MySqlConfiguration()
    {
      SetHistoryContext(
      "MySql.Data.MySqlClient", (conn, schema) => new MySqlHistoryContext(conn, schema));
    }
  }
}
```

Creating a custom EntityFramework initializer for ApplicationDbContext

The MySQL provider that is featured in this tutorial does not currently support Entity Framework migrations, so you will need to use model initializers in order to connect to the database. Because this tutorial is using a MySQL instance on Azure, you will need to create a custom Entity Framework initializer.

> **NOTE**
> This step is not required if you are connecting to a SQL Server instance on Azure or if you are using a database that is hosted on premises.

To create a custom Entity Framework initializer for MySQL, use the following steps:

1. Add a new class file named **MySqlInitializer.cs** to the project, and replace it's contents with the following code:

```
using IdentityMySQLDemo.Models;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;

namespace IdentityMySQLDemo
{
  public class MySqlInitializer : IDatabaseInitializer<ApplicationDbContext>
  {
    public void InitializeDatabase(ApplicationDbContext context)
    {
      if (!context.Database.Exists())
      {
        // if database did not exist before - create it
        context.Database.Create();
      }
      else
      {
        // query to check if MigrationHistory table is present in the database
        var migrationHistoryTableExists =
((IObjectContextAdapter)context).ObjectContext.ExecuteStoreQuery<int>(
          "SELECT COUNT(*) FROM information_schema.tables WHERE table_schema = 'IdentityMySQLDatabase'
AND table_name = '__MigrationHistory'");

        // if MigrationHistory table is not there (which is the case first time we run) - create it
        if (migrationHistoryTableExists.FirstOrDefault() == 0)
        {
          context.Database.Delete();
          context.Database.Create();
        }
      }
    }
  }
}
```

2. Open the **IdentityModels.cs** file for your project, which is located in the **Models** directory, and replace it's contents with the following:

```
using Microsoft.AspNet.Identity.EntityFramework;
using System.Data.Entity;

namespace IdentityMySQLDemo.Models
{
  // You can add profile data for the user by adding more properties to your ApplicationUser
  // class, please visit https://go.microsoft.com/fwlink/?LinkID=317594 to learn more.
  public class ApplicationUser : IdentityUser
  {
  }

  public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
  {
    static ApplicationDbContext()
    {
      Database.SetInitializer(new MySqlInitializer());
    }

    public ApplicationDbContext()
      : base("DefaultConnection")
    {
    }
  }
}
```

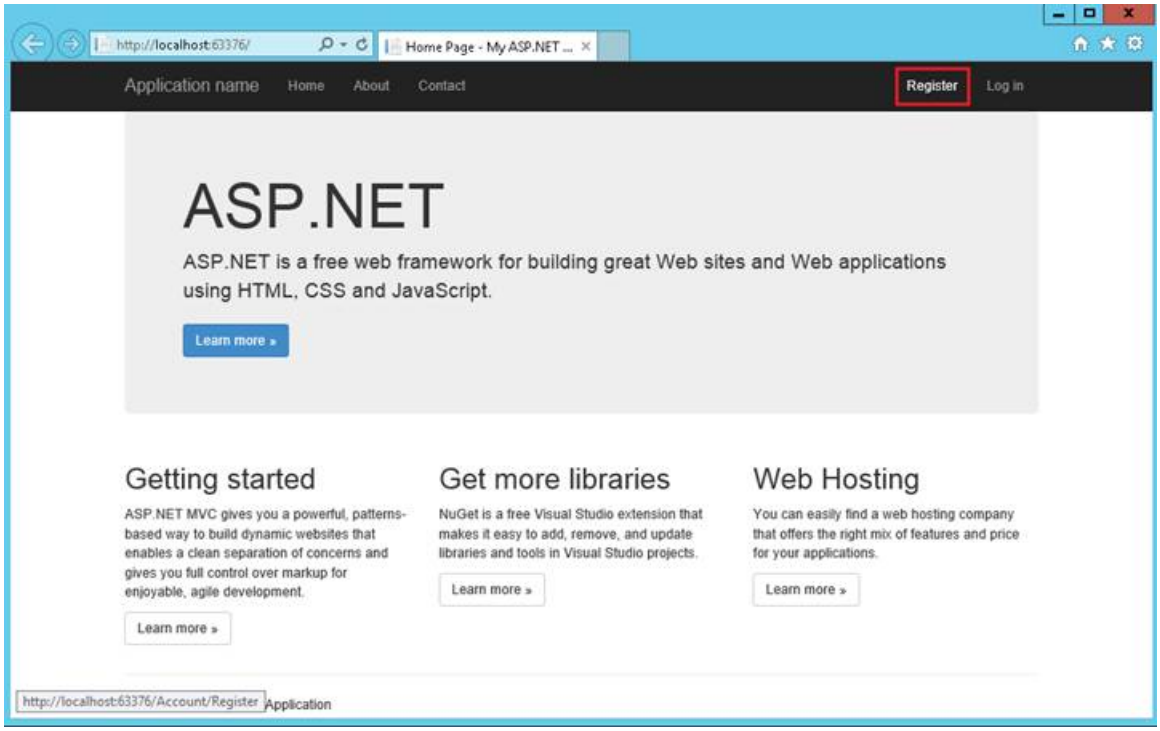
Running the application and verifying the database

Once you have completed the steps in the preceding sections, you should test your database. To do so, use the following steps:

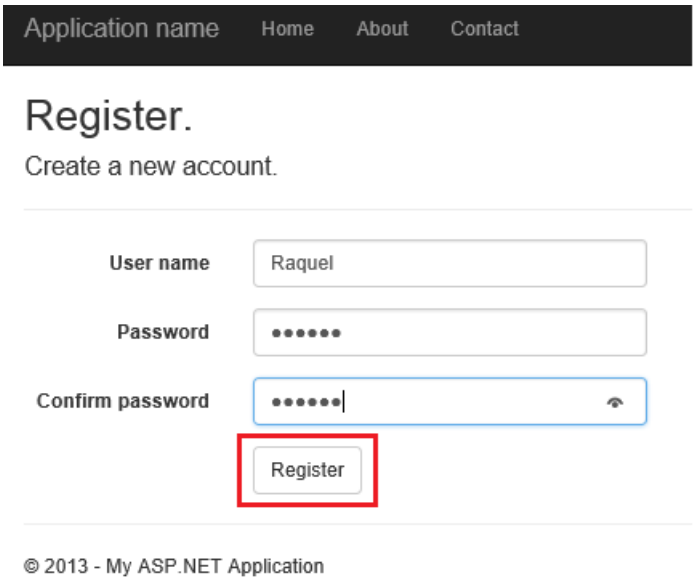
1. Press **Ctrl + F5** to build and run the web application.

2. Click the **Register** tab on the top of the page:

   [Click the following image to expand it.]



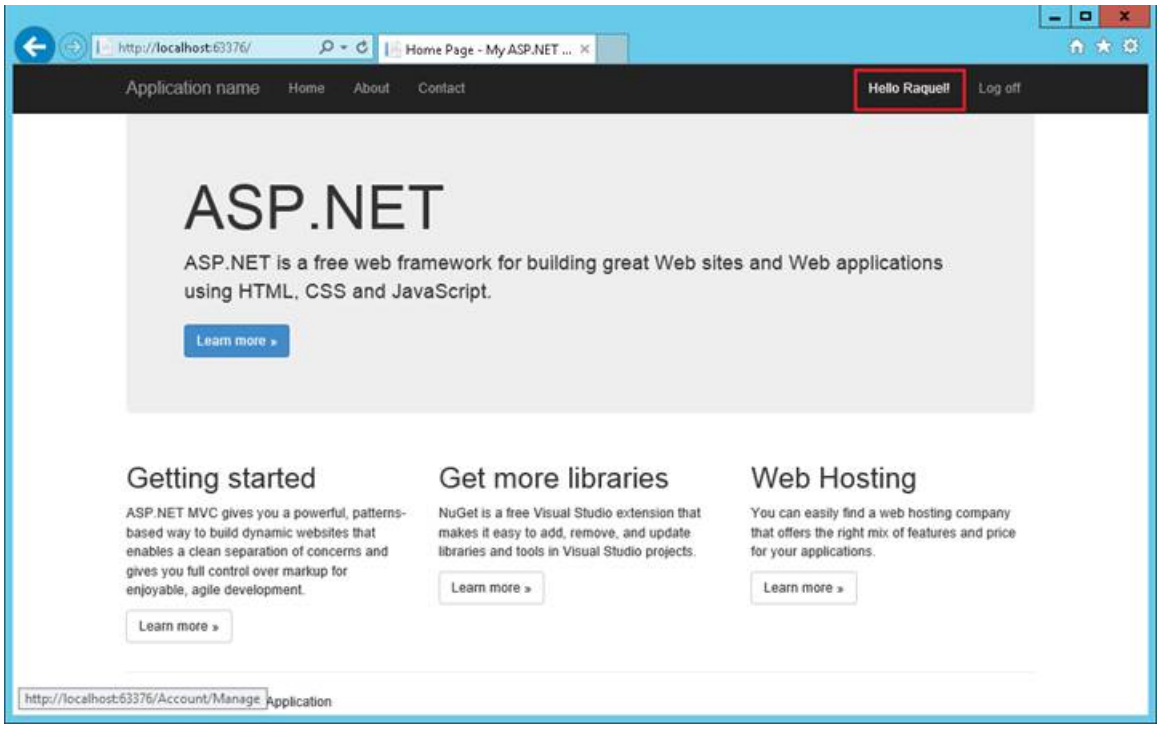
3. Enter a new user name and password, and then click **Register**:



   [Click the following image to expand it.]

4. At this point the ASP.NET Identity tables are created on the MySQL Database, and the user is registered and logged into the application:

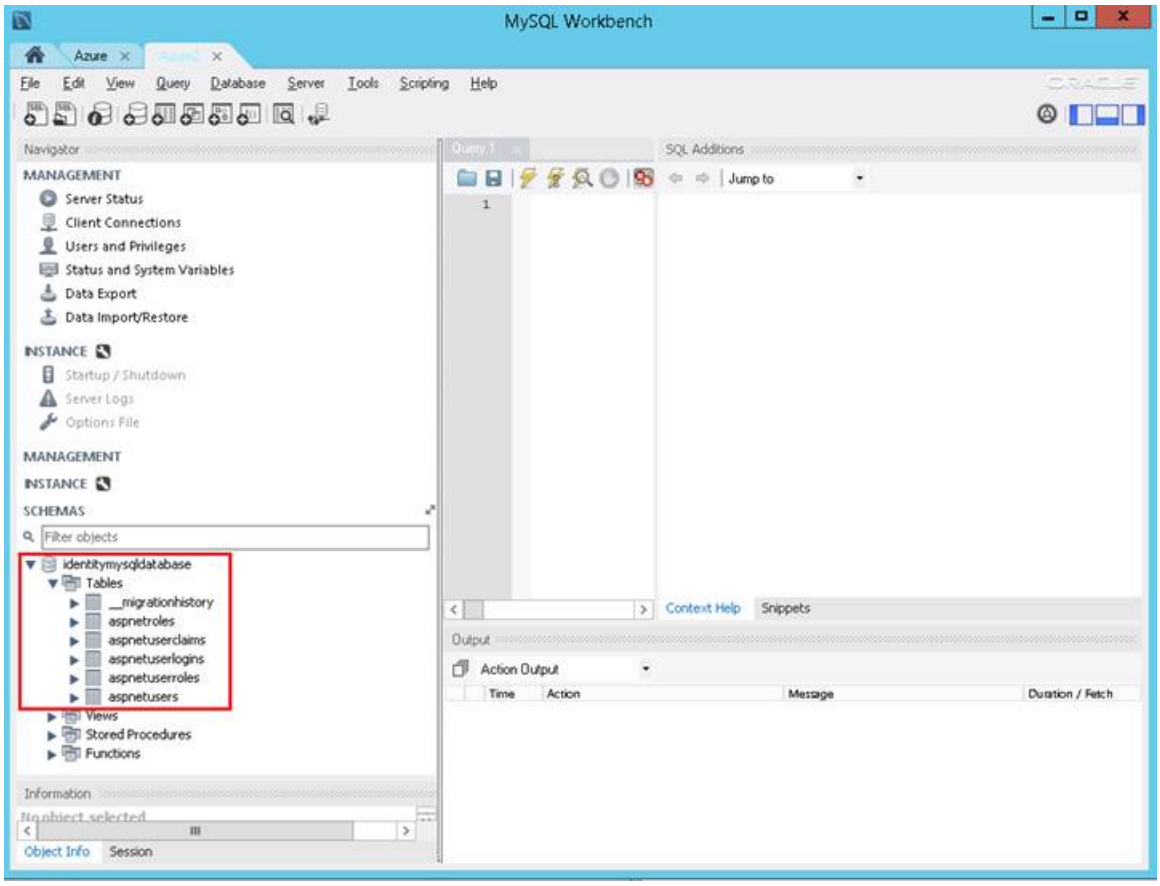
   [Click the following image to expand it.]



Installing MySQL Workbench tool to verify the data

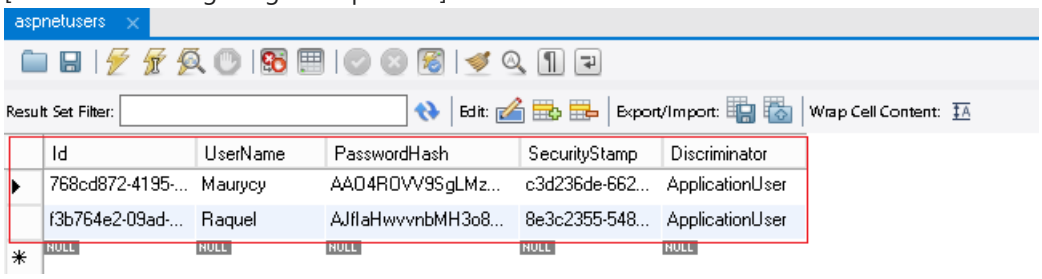
1. Install the **MySQL Workbench** tool from the MySQL downloads page
2. In the installation wizard: **Feature Selection** tab, select **MySQL Workbench** under **applications** section.
3. Launch the app and add a new connection using the connection string data from the Azure MySQL database you created at the begging of this tutorial.
4. After establishing the connection, inspect the **ASP.NET Identity** tables created on the **IdentityMySQLDatabase.**
5. You will see that all ASP.NET Identity required tables are created as shown in the image below:

   [Click the following image to expand it.]



6. Inspect the **aspnetusers** table for instance to check for the entries as you register new users.

[Click the following image to expand it.]



ASP.NET 4.x Identity Features & API

2/20/2020 • 2 minutes to read • Edit Online

Features & API

- Best practices for deploying passwords and other sensitive data to ASP.NET and Azure App Service
- Account Confirmation and Password Recovery with ASP.NET Identity (C#)
- Two-factor authentication using SMS and email with ASP.NET Identity

Best practices for deploying passwords and other sensitive data to ASP.NET and Azure App Service

2/20/2020 • 7 minutes to read • Edit Online

by Rick Anderson

> This tutorial shows how your code can securely store and access secure information. The most important point is you should never store passwords or other sensitive data in source code, and you shouldn't use production secrets in development and test mode.
>
> The sample code is a simple WebJob console app and a ASP.NET MVC app that needs access to a database connection string password, Twilio, Google and SendGrid secure keys.
>
> On premise settings and PHP is also mentioned.

- Working with passwords in the development environment
- Working with connection strings in the development environment
- WebJobs console apps
- Deploying secrets to Azure
- Notes for On-Premise and PHP
- Additional Resources

Working with passwords in the development environment

Tutorials frequently show sensitive data in source code, hopefully with a caveat that you should never store sensitive data in source code. For example, my ASP.NET MVC 5 app with SMS and email 2FA tutorial shows the following in the *web.config* file:

```
</connectionStrings>
   <appSettings>
      <add key="webpages:Version" value="3.0.0.0" />
      <!-- Markup removed for clarity. -->

      <!-- SendGrid-->
      <add key="mailAccount" value="account" />
      <add key="mailPassword" value="my password" />
      <!-- Twilio-->
      <add key="TwilioSid" value="My SID" />
      <add key="TwilioToken" value="My Token" />
      <add key="TwilioFromPhone" value="+12065551234" />

      <add key="GoogClientID" value="1234.apps.googleusercontent.com" />
      <add key="GoogClientSecret" value="My GCS" />
   </appSettings>
 <system.web>
```

The *web.config* file is source code, so these secrets should never be stored in that file. Fortunately, the `<appSettings>` element has a `file` attribute that allows you to specify an external file that contains sensitive app config settings. You can move all your secrets to an external file as long as the external file is not checked into your source tree. For example, in the following markup, the file *AppSettingsSecrets.config* contains all the app secrets:

```
</connectionStrings>
   <appSettings file="..\..\AppSettingsSecrets.config">
      <add key="webpages:Version" value="3.0.0.0" />
      <add key="webpages:Enabled" value="false" />
      <add key="ClientValidationEnabled" value="true" />
      <add key="UnobtrusiveJavaScriptEnabled" value="true" />
   </appSettings>
  <system.web>
```

The markup in the external file (*AppSettingsSecrets.config* in this sample), is the same markup found in the *web.config* file:

```
<appSettings>
   <!-- SendGrid-->
   <add key="mailAccount" value="My mail account." />
   <add key="mailPassword" value="My mail password." />
   <!-- Twilio-->
   <add key="TwilioSid" value="My Twilio SID." />
   <add key="TwilioToken" value="My Twilio Token." />
   <add key="TwilioFromPhone" value="+12065551234" />

   <add key="GoogClientID" value="1.apps.googleusercontent.com" />
   <add key="GoogClientSecret" value="My Google client secret." />
</appSettings>
```

The ASP.NET runtime merges the contents of the external file with the markup in <appSettings> element. The runtime ignores the file attribute if the specified file cannot be found.

WARNING

Security - Do not add your *secrets .config* file to your project or check it into source control. By default, Visual Studio sets the `Build Action` to `Content`, which means the file is deployed. For more information see Why don't all of the files in my project folder get deployed? Although you can use any extension for the *secrets .config* file, it's best to keep it *.config*, as config files are not served by IIS. Notice also that the *AppSettingsSecrets.config* file is two directory levels up from the *web.config* file, so it's completely out of the solution directory. By moving the file out of the solution directory, "git add *" won't add it to your repository.

Working with connection strings in the development environment

Visual Studio creates new ASP.NET projects that use LocalDB. LocalDB was created specifically for the development environment. It doesn't require a password, therefore you don't need to do anything to prevent secrets from being checked into your source code. Some development teams use the full versions of SQL Server (or other DBMS) that require a password.

You can use the `configSource` attribute to replace the entire `<connectionStrings>` markup. Unlike the `<appSettings>` `file` attribute that merges the markup, the `configSource` attribute replaces the markup. The following markup shows the `configSource` attribute in the *web.config* file:

```
<connectionStrings configSource="ConnectionStrings.config">
</connectionStrings>
```

NOTE

If you use the `configSource` attribute as shown above to move your connection strings to an external file, and have Visual Studio create a new web site, it won't be able to detect you are using a database, and you won't get the option of configuring the database when you publish to Azure from Visual Studio. If you are using the `configSource` attribute, you can use PowerShell to create and deploy your web site and database, or you can create the web site and the database in the portal before you publish. The New-AzureWebsitewithDB.ps1 script will create a new web site and database.

WARNING

Security - Unlike the *AppSettingsSecrets.config* file, the external connection strings file must be in the same directory as the root *web.config* file, so you'll have to take precautions to ensure you don't check it into your source repository.

NOTE

Security Warning on secrets file: A best practice is to not use production secrets in test and development. Using production passwords in test or development leaks those secrets.

WebJobs console apps

The *app.config* file used by a console app doesn't support relative paths, but it does support absolute paths. You can use an absolute path to move your secrets out of your project directory. The following markup shows the secrets in the *C:\secrets\AppSettingsSecrets.config* file, and non sensitive data in the *app.config* file.

```
<configuration>
  <appSettings file="C:\secrets\AppSettingsSecrets.config">
    <add key="TwitterMaxThreads" value="24" />
    <add key="StackOverflowMaxThreads" value="24" />
    <add key="MaxDaysForPurge" value="30" />
  </appSettings>
</configuration>
```

Deploying secrets to Azure

When you deploy your web app to Azure, the *AppSettingsSecrets.config* file won't be deployed (that's what you want). You could go to the Azure Management Portal and set them manually, to do that:

1. Go to https://portal.azure.com, and sign in with your Azure credentials.
2. Click **Browse** > **Web Apps**, then click the name of your web app.
3. Click **All settings** > **Application settings**.

The **app settings** and **connection string** values override the same settings in the *web.config* file. In our example, we did not deploy these settings to Azure, but if these keys were in the *web.config* file, the settings shown on the portal would take precedence.

A best practice is to follow a DevOps workflow and use Azure PowerShell (or another framework such as Chef or Puppet) to automate setting these values in Azure. The following PowerShell script uses Export-CliXml to export the encrypted secrets to disk:

```
param(
  [Parameter(Mandatory=$true)]
  [String]$Name,
  [Parameter(Mandatory=$true)]
  [String]$Password)

$credPath = $PSScriptRoot + '\' + $Name + ".credential"
$PWord = ConvertTo-SecureString –String $Password –AsPlainText -Force
$Credential = New-Object –TypeName `
System.Management.Automation.PSCredential –ArgumentList $Name, $PWord
$Credential | Export-CliXml $credPath
```

In the script above, 'Name' is the name of the secret key, such as '"FB_AppSecret" or "TwitterSecret". You can view the ".credential" file created by the script in your browser. The snippet below tests each of the credential files and sets the secrets for the named web app:

```
Function GetPW_fromCredFile { Param( [String]$CredFile )
  $Credential = GetCredsFromFile $CredFile
  $PW = $Credential.GetNetworkCredential().Password
  # $user just for debugging.
  $user = $Credential.GetNetworkCredential().username
  Return $PW
}
$AppSettings = @{
  "FB_AppSecret"     = GetPW_fromCredFile "FB_AppSecret.credential";
  "GoogClientSecret" = GetPW_fromCredFile "GoogClientSecret.credential";
  "TwitterSecret"    = GetPW_fromCredFile "TwitterSecret.credential";
}
Set-AzureWebsite -Name $WebSiteName -AppSettings $AppSettings
```

WARNING

Security - Don't include passwords or other secrets in the PowerShell script, doing so defeats the purpose of using a PowerShell script to deploy sensitive data. The Get-Credential cmdlet provides a secure mechanism to obtain a password. Using a UI prompt can prevent leaking a password.

Deploying DB connection strings

DB connection strings are handled similarly to app settings. If you deploy your web app from Visual Studio, the connection string will be configured for you. You can verify this in the portal. The recommended way to set the connection string is with PowerShell. For an example of a PowerShell script the creates a website and database and sets the connection string in the website, download New-AzureWebsitewithDB.ps1 from the Azure Script library.

Notes for PHP

Since the key-value pairs for both **app settings** and **connection strings** are stored in environment variables on Azure App Service, developers that use any web app frameworks (such as PHP) can easily retrieve these values. See Stefan Schackow's Windows Azure Web Sites: How Application Strings and Connection Strings Work blog post which shows a PHP snippet to read app settings and connection strings.

Notes for on-premises servers

If you are deploying to on-premises web servers, you can help secure secrets by encrypting the configuration sections of configuration files. As an alternative, you can use the same approach recommended for Azure Websites: keep development settings in configuration files, and use environment variable values for production settings. In this case, however, you have to write application code for functionality that is automatic in Azure

Websites: retrieve settings from environment variables and use those values in place of configuration file settings, or use configuration file settings when environment variables are not found.

Additional Resources

For an example of a PowerShell script that creates a web app + database, sets the connection string + app settings, download New-AzureWebsitewithDB.ps1 from the Azure Script library.

See Stefan Schackow's Windows Azure Web Sites: How Application Strings and Connection Strings Work

Special thanks to Barry Dorrans (@blowdart) and Carlos Farre for reviewing.

Account confirmation and password recovery with ASP.NET Identity (C#)

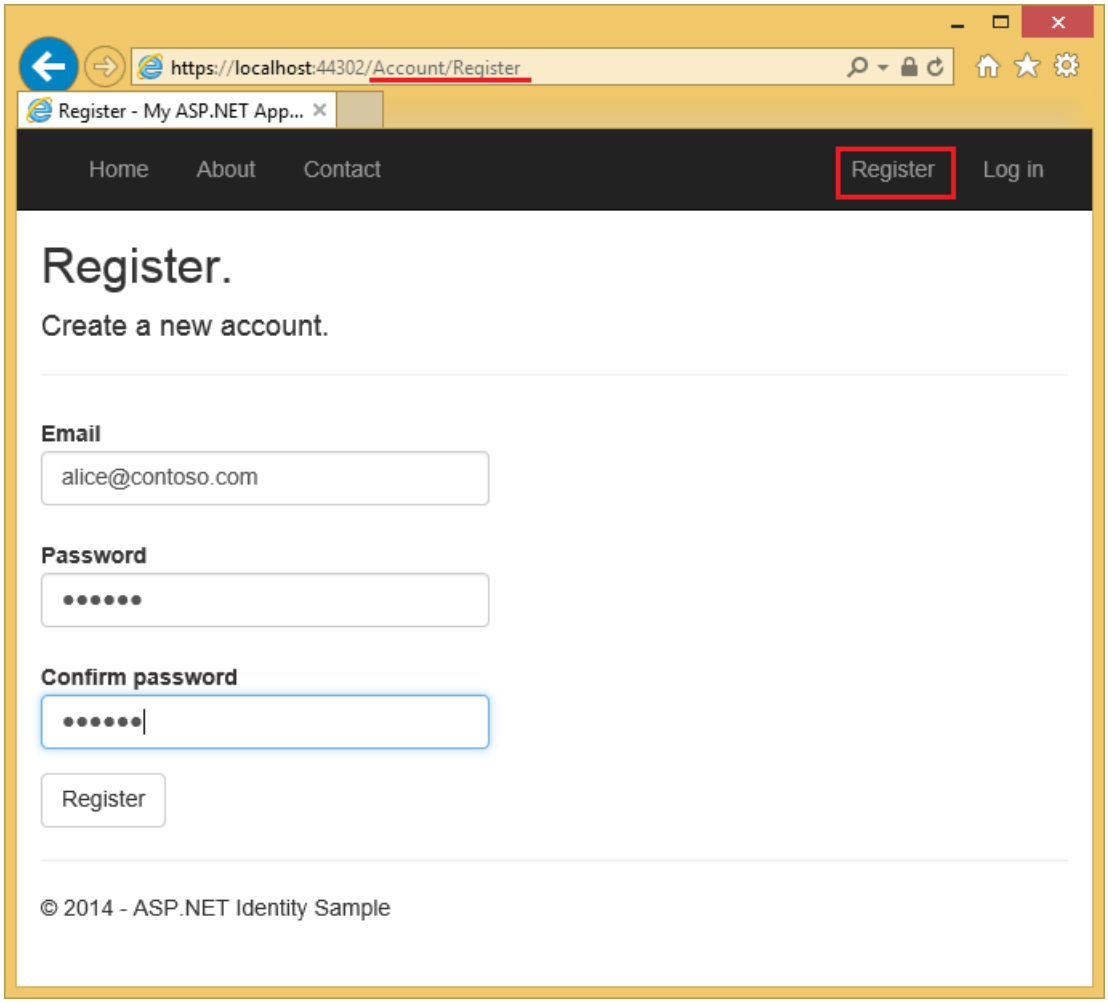
2/20/2020 • 15 minutes to read • Edit Online

Before doing this tutorial you should first complete Create a secure ASP.NET MVC 5 web app with log in, email confirmation and password reset. This tutorial contains more details and will show you how to set up email for local account confirmation and allow users to reset their forgotten password in ASP.NET Identity.

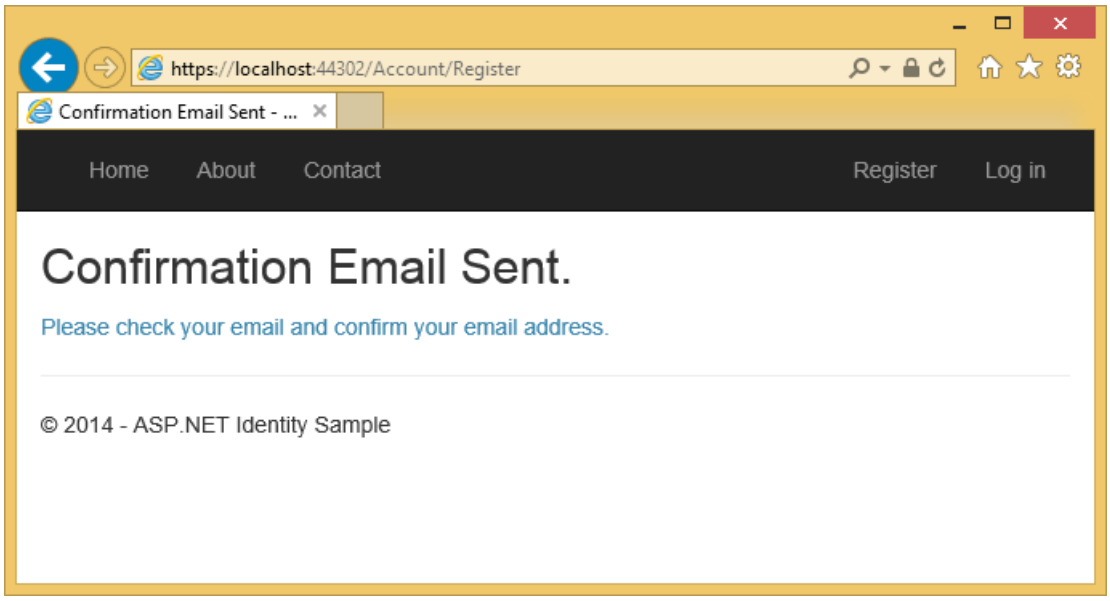
A local user account requires the user to create a password for the account, and that password is stored (securely) in the web app. ASP.NET Identity also supports social accounts, which don't require the user to create a password for the app. Social accounts use a third party (such as Google, Twitter, Facebook, or Microsoft) to authenticate users. This topic covers the following:

- Create an ASP.NET MVC app and explore ASP.NET Identity features.
- Build the Identity sample
- Set up email confirmation

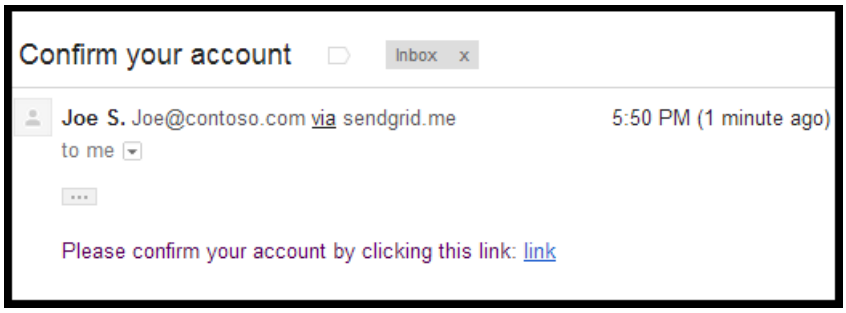
New users register their email alias, which creates a local account.



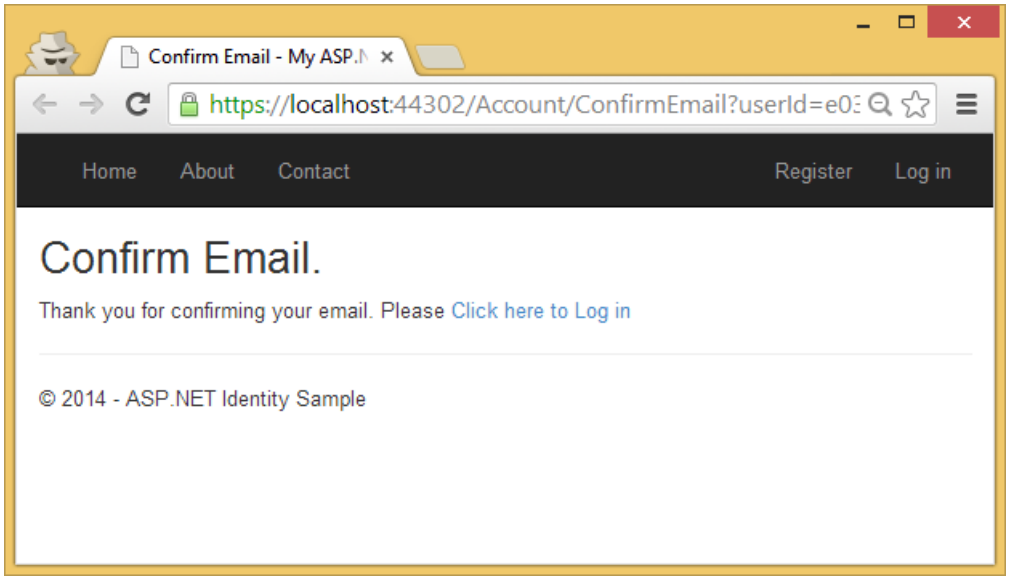
Selecting the Register button sends a confirmation email containing a validation token to their email address.



The user is sent an email with a confirmation token for their account.

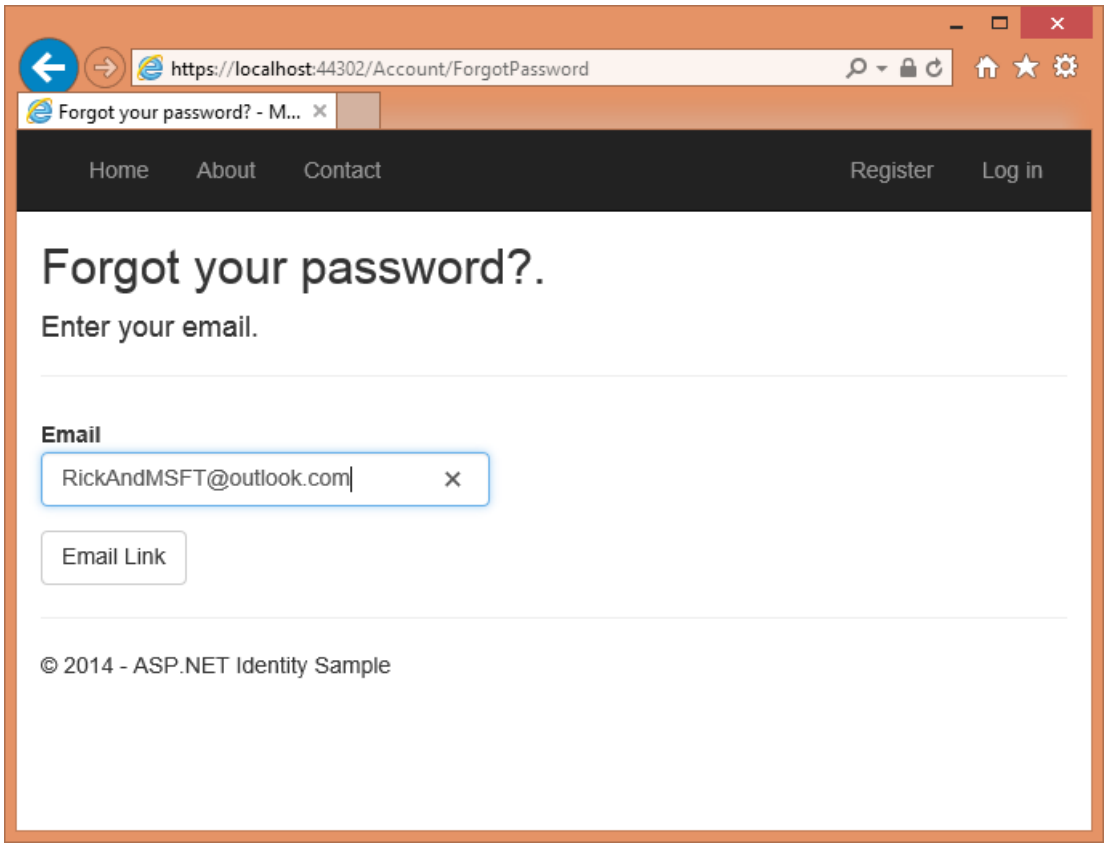


Selecting the link confirms the account.

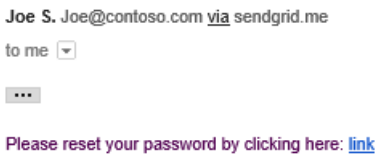


Password recovery/reset

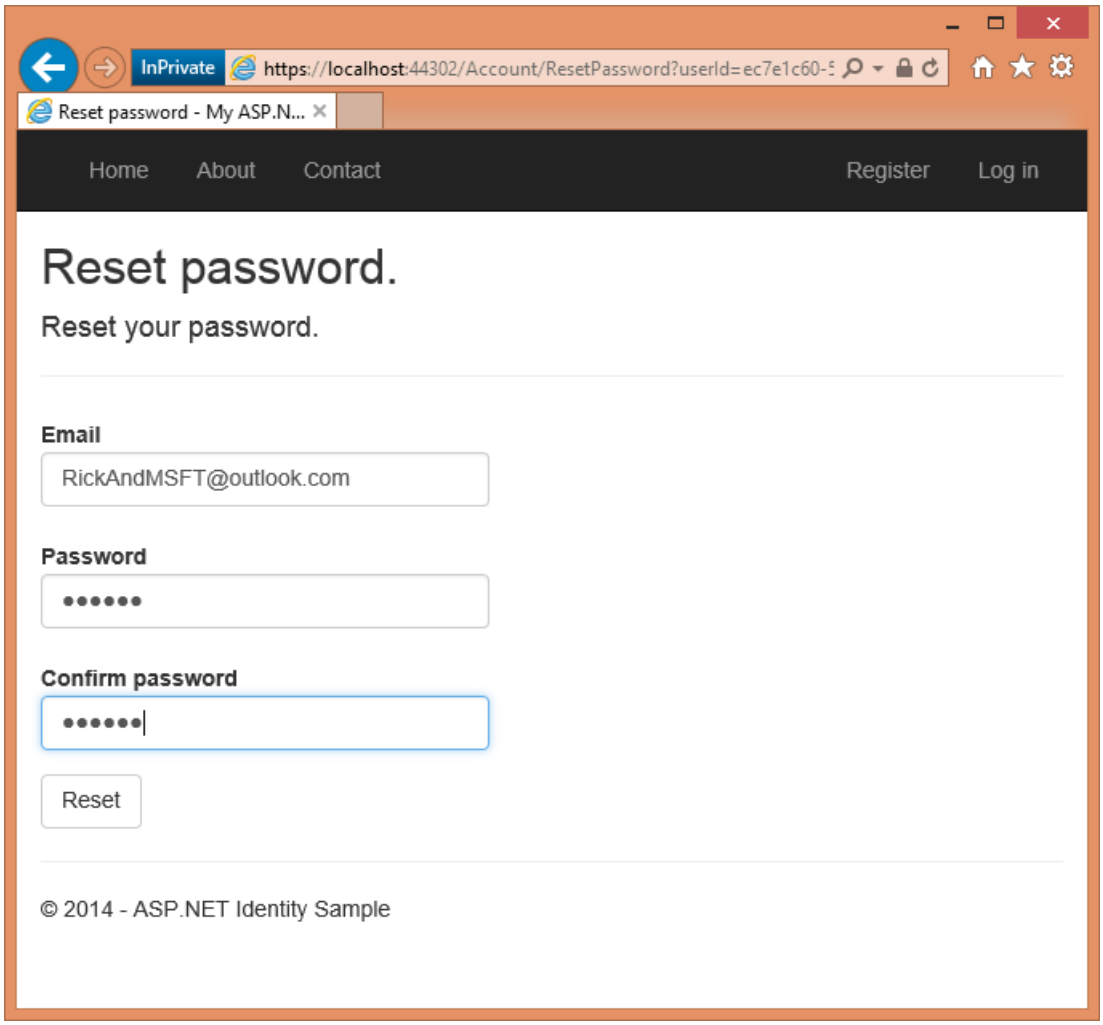
Local users who forget their password can have a security token sent to their email account, enabling them to reset their password.



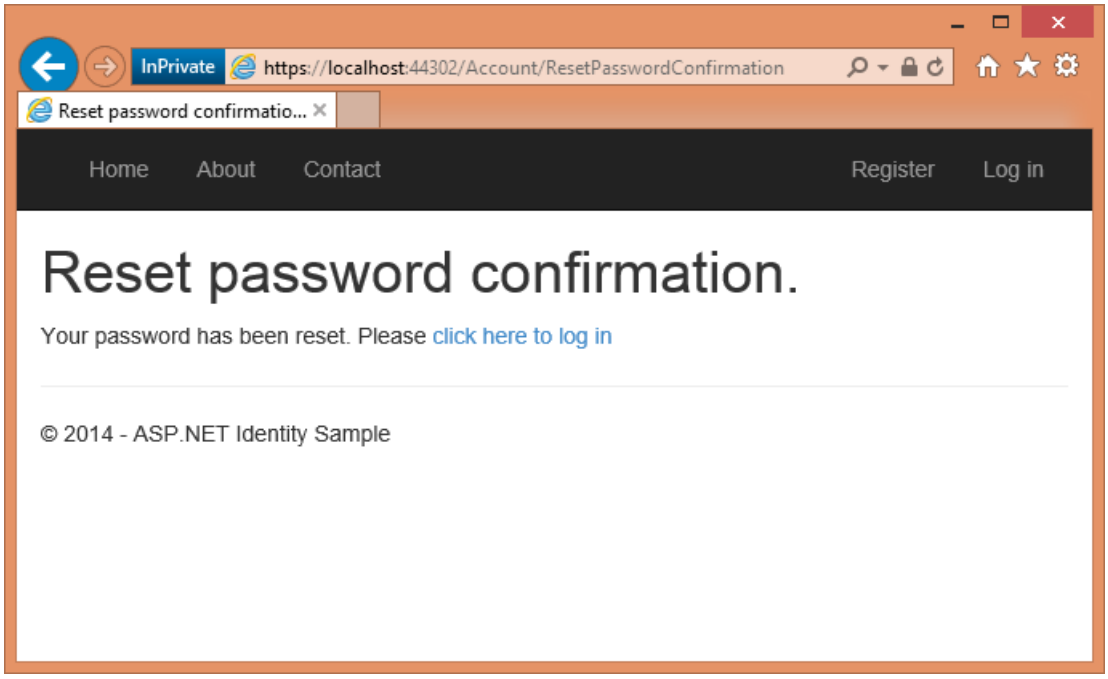
The user will soon get an email with a link allowing them to reset their password.



Selecting the link will take them to the Reset page.



Selecting the **Reset** button will confirm the password has been reset.



Create an ASP.NET web app

Start by installing and running Visual Studio 2017.

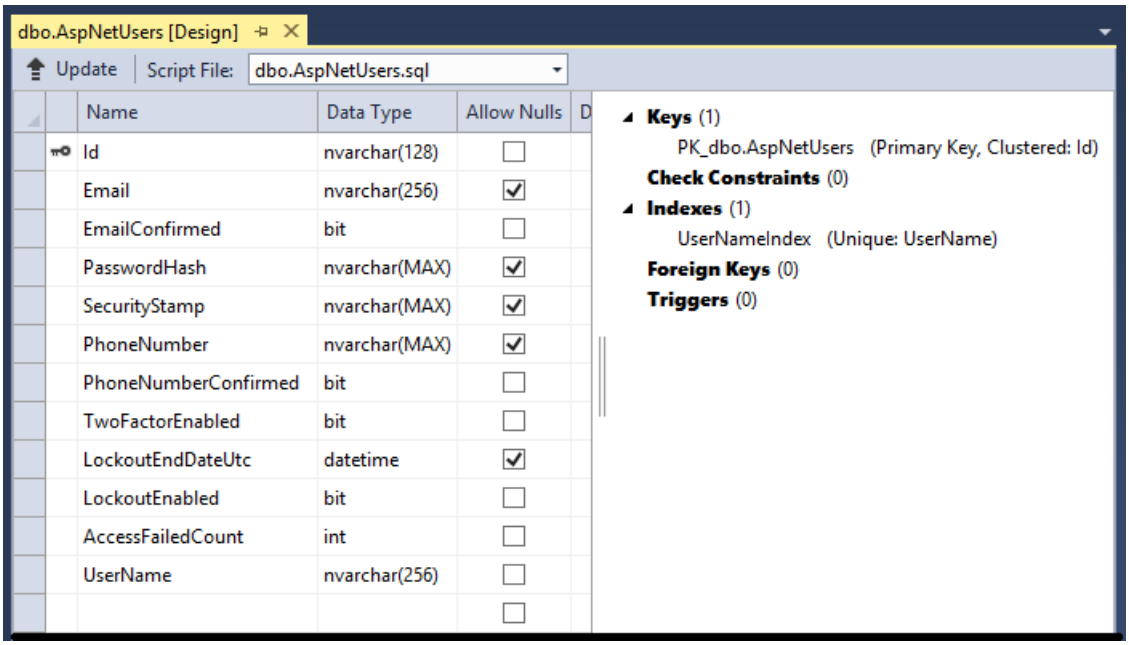
1. Create a new ASP.NET Web project and select the MVC template. Web Forms also support ASP.NET Identity, so you could follow similar steps in a web forms app.

2. Change the authentication to **Individual User Accounts**.

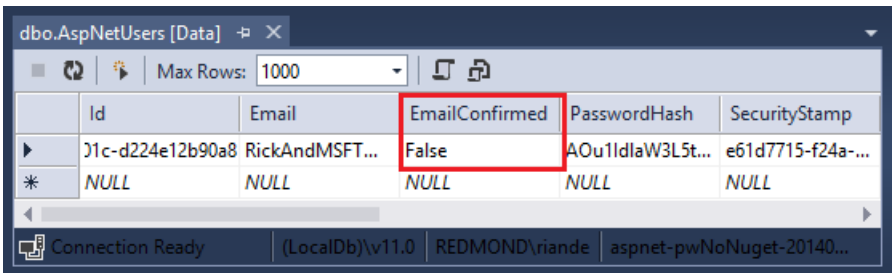
3. Run the app, select the **Register** link and register a user. At this point, the only validation on the email is with the [EmailAddress] attribute.

4. In Server Explorer, navigate to **Data Connections\DefaultConnection\Tables\AspNetUsers**, right-click and select **Open table definition**.

   The following image shows the `AspNetUsers` schema:



5. Right-click on the **AspNetUsers** table and select **Show Table Data**.



   At this point the email has not been confirmed.

The default data store for ASP.NET Identity is Entity Framework, but you can configure it to use other data stores and to add additional fields. See Additional Resources section at the end of this tutorial.

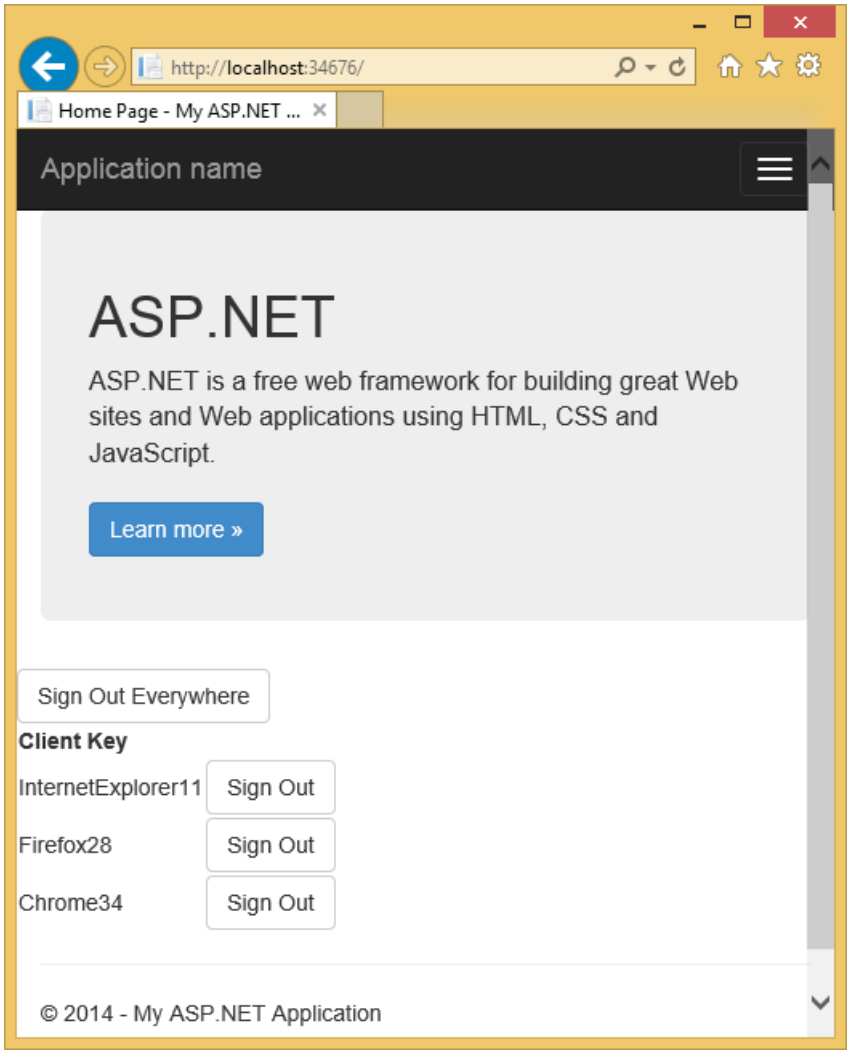
The OWIN startup class (*Startup.cs*) is called when the app starts and invokes the `ConfigureAuth` method in *App_Start\Startup.Auth.cs*, which configures the OWIN pipeline and initializes ASP.NET Identity. Examine the `ConfigureAuth` method. Each `CreatePerOwinContext` call registers a callback (saved in the `OwinContext`) that will be called once per request to create an instance of the specified type. You can set a break point in the constructor and `Create` method of each type (`ApplicationDbContext, ApplicationUserManager`) and verify they are called on each request. A instance of `ApplicationDbContext` and `ApplicationUserManager` is stored in the OWIN context, which can be accessed throughout the application. ASP.NET Identity hooks into the OWIN pipeline through cookie middleware. For more information, see Per request lifetime management for UserManager class in ASP.NET Identity.

When you change your security profile, a new security stamp is generated and stored in the `SecurityStamp` field of the *AspNetUsers* table. Note, the `SecurityStamp` field is different from the security cookie. The security cookie is not stored in the `AspNetUsers` table (or anywhere else in the Identity DB). The security cookie token is self-signed using DPAPI and is created with the `UserId, SecurityStamp` and expiration time information.

The cookie middleware checks the cookie on each request. The `SecurityStampValidator` method in the `Startup` class hits the DB and checks security stamp periodically, as specified with the `validateInterval`. This only happens every 30 minutes (in our sample) unless you change your security profile. The 30 minute interval was chosen to minimize trips to the database. See my two-factor authentication tutorial for more details.

Per the comments in the code, the `UseCookieAuthentication` method supports cookie authentication. The `SecurityStamp` field and associated code provides an extra layer of security to your app, when you change your password, you will be logged out of the browser you logged in with. The `SecurityStampValidator.OnValidateIdentity` method enables the app to validate the security token when the user logs in, which is used when you change a password or use the external login. This is needed to ensure that any tokens (cookies) generated with the old password are invalidated. In the sample project, if you change the users password then a new token is generated for the user, any previous tokens are invalidated and the `SecurityStamp` field is updated.

The Identity system allow you to configure your app so when the users security profile changes (for example, when the user changes their password or changes associated login (such as from Facebook, Google, Microsoft account, etc.), the user is logged out of all browser instances. For example, the image below shows the Single signout sample app, which allows the user to sign out of all browser instances (in this case, IE, Firefox and Chrome) by selecting one button. Alternatively, the sample allows you to only log out of a specific browser instance.



The Single signout sample app shows how ASP.NET Identity allows you to regenerate the security token. This is needed to ensure that any tokens (cookies) generated with the old password are invalidated. This feature provides an extra layer of security to your application; when you change your password, you will be logged out where you have logged into this application.

The *App_Start\IdentityConfig.cs* file contains the `ApplicationUserManager`, `EmailService` and `SmsService` classes. The `EmailService` and `SmsService` classes each implement the `IIdentityMessageService` interface, so you have common methods in each class to configure email and SMS. Although this tutorial only shows how to add email notification through SendGrid, you can send email using SMTP and other mechanisms.

The `Startup` class also contains boiler plate to add social logins (Facebook, Twitter, etc.), see my tutorial MVC 5 App with Facebook, Twitter, LinkedIn and Google OAuth2 Sign-on for more info.

Examine the `ApplicationUserManager` class, which contains the users identity information and configures the following features:

- Password strength requirements.
- User lock out (attempts and time).
- Two-factor authentication (2FA). I'll cover 2FA and SMS in another tutorial.
- Hooking up the email and SMS services. (I'll cover SMS in another tutorial).

The `ApplicationUserManager` class derives from the generic `UserManager<ApplicationUser>` class. `ApplicationUser` derives from IdentityUser. `IdentityUser` derives from the generic `IdentityUser` class:

```
//     Default EntityFramework IUser implementation
public class IdentityUser<TKey, TLogin, TRole, TClaim> : IUser<TKey>
   where TLogin : IdentityUserLogin<TKey>
   where TRole : IdentityUserRole<TKey>
   where TClaim : IdentityUserClaim<TKey>
{
   public IdentityUser()
   {
      Claims = new List<TClaim>();
      Roles = new List<TRole>();
      Logins = new List<TLogin>();
   }

   ///     User ID (Primary Key)
   public virtual TKey Id { get; set; }

   public virtual string Email { get; set; }
   public virtual bool EmailConfirmed { get; set; }

   public virtual string PasswordHash { get; set; }

   ///     A random value that should change whenever a users credentials have changed (password changed,
login removed)
   public virtual string SecurityStamp { get; set; }

   public virtual string PhoneNumber { get; set; }
   public virtual bool PhoneNumberConfirmed { get; set; }

   public virtual bool TwoFactorEnabled { get; set; }

   ///     DateTime in UTC when lockout ends, any time in the past is considered not locked out.
   public virtual DateTime? LockoutEndDateUtc { get; set; }

   public virtual bool LockoutEnabled { get; set; }

   ///     Used to record failures for the purposes of lockout
   public virtual int AccessFailedCount { get; set; }

   ///     Navigation property for user roles
   public virtual ICollection<TRole> Roles { get; private set; }

   ///     Navigation property for user claims
   public virtual ICollection<TClaim> Claims { get; private set; }

   ///     Navigation property for user logins
   public virtual ICollection<TLogin> Logins { get; private set; }

   public virtual string UserName { get; set; }
}
```

The properties above coincide with the properties in the `AspNetUsers` table, shown above.

Generic arguments on `IUser` enable you to derive a class using different types for the primary key. See the ChangePK sample which shows how to change the primary key from string to int or GUID.

ApplicationUser

`ApplicationUser` (`public class ApplicationUserManager : UserManager<ApplicationUser>`) is defined in *Models\IdentityModels.cs* as:

```
public class ApplicationUser : IdentityUser
{
    public async Task<ClaimsIdentity> GenerateUserIdentityAsync(
        UserManager<ApplicationUser> manager)
    {
        // Note the authenticationType must match the one defined in
       //   CookieAuthenticationOptions.AuthenticationType
        var userIdentity = await manager.CreateIdentityAsync(this,
    DefaultAuthenticationTypes.ApplicationCookie);
        // Add custom user claims here
        return userIdentity;
    }
}
```

The highlighted code above generates a ClaimsIdentity. ASP.NET Identity and OWIN Cookie Authentication are claims-based, therefore the framework requires the app to generate a `ClaimsIdentity` for the user. `ClaimsIdentity` has information about all the claims for the user, such as the user's name, age and what roles the user belongs to. You can also add more claims for the user at this stage.

The OWIN `AuthenticationManager.SignIn` method passes in the `ClaimsIdentity` and signs in the user:

```
private async Task SignInAsync(ApplicationUser user, bool isPersistent)
{
    AuthenticationManager.SignOut(DefaultAuthenticationTypes.ExternalCookie);
    AuthenticationManager.SignIn(new AuthenticationProperties(){
       IsPersistent = isPersistent },
       await user.GenerateUserIdentityAsync(UserManager));
}
```

MVC 5 App with Facebook, Twitter, LinkedIn and Google OAuth2 Sign-on shows how you can add additional properties to the `ApplicationUser` class.

Email confirmation

It's a good idea to confirm the email a new user register with to verify they are not impersonating someone else (that is, they haven't registered with someone else's email). Suppose you had a discussion forum, you would want to prevent `"bob@example.com"` from registering as `"joe@contoso.com"`. Without email confirmation, `"joe@contoso.com"` could get unwanted email from your app. Suppose Bob accidentally registered as `"bib@example.com"` and hadn't noticed it, he wouldn't be able to use password recover because the app doesn't have his correct email. Email confirmation provides only limited protection from bots and doesn't provide protection from determined spammers, they have many working email aliases they can use to register.In the sample below, the user won't be able to change their password until their account has been confirmed (by them selecting a confirmation link received on the email account they registered with.) You can apply this work flow to other scenarios, for example sending a link to confirm and reset the password on new accounts created by the administrator, sending the user an email when they have changed their profile and so on. You generally want to prevent new users from posting any data to your web site before they have been confirmed by email, a SMS text message or another mechanism.

Build a more complete sample

In this section, you'll use NuGet to download a more complete sample we will work with.

1. Create a new ***empty*** ASP.NET Web project.

2. In the Package Manager Console, enter the following commands:

```
Install-Package SendGrid
Install-Package -Prerelease Microsoft.AspNet.Identity.Samples
```

In this tutorial, we'll use SendGrid to send email. The `Identity.Samples` package installs the code we will be working with.

3. Set the project to use SSL.
4. Test local account creation by running the app, selecting the **Register** link, and posting the registration form.
5. Select the demo email link, which simulates email confirmation.
6. Remove the demo email link confirmation code from the sample (The `ViewBag.Link` code in the account controller. See the `DisplayEmail` and `ForgotPasswordConfirmation` action methods and razor views).

> **WARNING**
> If you change any of the security settings in this sample, productions apps will need to undergo a security audit that explicitly calls the changes made.

Examine the code in App_Start\IdentityConfig.cs

The sample shows how to create an account and add it to the *Admin* role. You should replace the email in the sample with the email you will be using for the admin account. The easiest way right now to create an administrator account is programmatically in the `Seed` method. We hope to have a tool in the future that will allow you to create and administer users and roles. The sample code does let you create and manage users and roles, but you must first have an administrators account to run the roles and user admin pages. In this sample, the admin account is created when the DB is seeded.

Change the password and change the name to an account where you can receive email notifications.

> **WARNING**
> Security - Never store sensitive data in your source code.

As mentioned previously, the `app.CreatePerOwinContext` call in the startup class adds callbacks to the `Create` method of the app DB content, user manager and role manger classes. The OWIN pipeline calls the `Create` method on these classes for each request and stores the context for each class. The account controller exposes the user manager from the HTTP context (which contains the OWIN context):

```
public ApplicationUserManager UserManager
{
    get
    {
        return _userManager ??
    HttpContext.GetOwinContext().GetUserManager<ApplicationUserManager>();
    }
    private set
    {
        _userManager = value;
    }
}
```

When a user registers a local account, the `HTTP Post Register` method is called:

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await UserManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            var code = await UserManager.GenerateEmailConfirmationTokenAsync(user.Id);
            var callbackUrl = Url.Action(
               "ConfirmEmail", "Account",
               new { userId = user.Id, code = code },
               protocol: Request.Url.Scheme);

            await UserManager.SendEmailAsync(user.Id,
               "Confirm your account",
               "Please confirm your account by clicking this link: <a href=\""
                                                + callbackUrl + "\">link</a>");
            // ViewBag.Link = callbackUrl;   // Used only for initial demo.
            return View("DisplayEmail");
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

The code above uses the model data to create a new user account using the email and password entered. If the email alias is in the data store, account creation fails and the form is displayed again. The `GenerateEmailConfirmationTokenAsync` method creates a secure confirmation token and stores it in the ASP.NET Identity data store. The Url.Action method creates a link containing the `UserId` and confirmation token. This link is then emailed to the user, the user can select on the link in their email app to confirm their account.

Set up email confirmation

Go to the Azure SendGrid sign up page and register for free account. Add code similar to the following to configure SendGrid:

```
public class EmailService : IIdentityMessageService
{
   public Task SendAsync(IdentityMessage message)
   {
      return configSendGridasync(message);
   }

   private Task configSendGridasync(IdentityMessage message)
   {
      var myMessage = new SendGridMessage();
      myMessage.AddTo(message.Destination);
      myMessage.From = new System.Net.Mail.MailAddress(
                          "Joe@contoso.com", "Joe S.");
      myMessage.Subject = message.Subject;
      myMessage.Text = message.Body;
      myMessage.Html = message.Body;

      var credentials = new NetworkCredential(
                 ConfigurationManager.AppSettings["mailAccount"],
                 ConfigurationManager.AppSettings["mailPassword"]
                 );

      // Create a Web transport for sending email.
      var transportWeb = new Web(credentials);

      // Send the email.
      if (transportWeb != null)
      {
         return transportWeb.DeliverAsync(myMessage);
      }
      else
      {
         return Task.FromResult(0);
      }
   }
}
```

NOTE

Email clients frequently accept only text messages (no HTML). You should provide the message in text and HTML. In the SendGrid sample above, this is done with the `myMessage.Text` and `myMessage.Html` code shown above.

The following code shows how to send email using the MailMessage class where `message.Body` returns only the link.

```
void sendMail(Message message)
{
#region formatter
   string text = string.Format("Please click on this link to {0}: {1}", message.Subject, message.Body);
   string html = "Please confirm your account by clicking this link: <a href=\"" + message.Body +
"\">link</a><br/>";

   html += HttpUtility.HtmlEncode(@"Or click on the copy the following link on the browser:" +
message.Body);
#endregion

   MailMessage msg = new MailMessage();
   msg.From = new MailAddress("joe@contoso.com");
   msg.To.Add(new MailAddress(message.Destination));
   msg.Subject = message.Subject;
   msg.AlternateViews.Add(AlternateView.CreateAlternateViewFromString(text, null,
MediaTypeNames.Text.Plain));
   msg.AlternateViews.Add(AlternateView.CreateAlternateViewFromString(html, null,
MediaTypeNames.Text.Html));

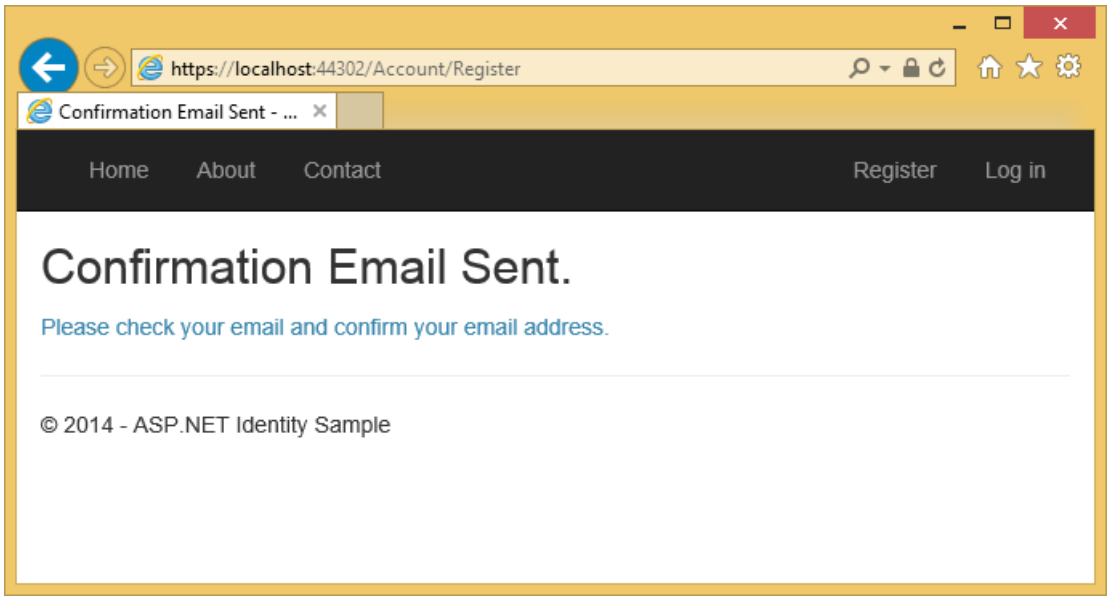
   SmtpClient smtpClient = new SmtpClient("smtp.gmail.com", Convert.ToInt32(587));
   System.Net.NetworkCredential credentials = new System.Net.NetworkCredential("joe@contoso.com",
"XXXXXX");
   smtpClient.Credentials = credentials;
   smtpClient.EnableSsl = true;
   smtpClient.Send(msg);
}
```

WARNING

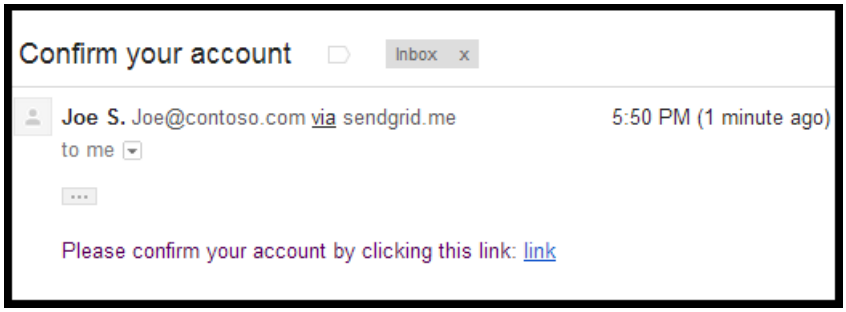
Security - Never store sensitive data in your source code. The account and credentials are stored in the appSetting. On Azure, you can securely store these values on the **Configure** tab in the Azure portal. See Best practices for deploying passwords and other sensitive data to ASP.NET and Azure.

Enter your SendGrid credentials, run the app, register with an email alias can select the confirm link in your email. To see how to do this with your Outlook.com email account, see John Atten's C# SMTP Configuration for Outlook.Com SMTP Host and hisASP.NET Identity 2.0: Setting Up Account Validation and Two-Factor Authorization posts.

Once a user selects the **Register** button a confirmation email containing a validation token is sent to their email address.



The user is sent an email with a confirmation token for their account.



Examine the code

The following code shows the `POST ForgotPassword` method.

```
public async Task<ActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = await UserManager.FindByNameAsync(model.Email);
        if (user == null || !(await UserManager.IsEmailConfirmedAsync(user.Id)))
        {
            // Don't reveal that the user does not exist or is not confirmed
            return View("ForgotPasswordConfirmation");
        }

        var code = await UserManager.GeneratePasswordResetTokenAsync(user.Id);
        var callbackUrl = Url.Action("ResetPassword", "Account",
    new { UserId = user.Id, code = code }, protocol: Request.Url.Scheme);
        await UserManager.SendEmailAsync(user.Id, "Reset Password",
    "Please reset your password by clicking here: <a href=\"" + callbackUrl + "\">link</a>");
        return View("ForgotPasswordConfirmation");
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

The method fails silently if the user email has not been confirmed. If an error was posted for an invalid email address, malicious users could use that information to find valid userId (email aliases) to attack.

The following code shows the `ConfirmEmail` method in the account controller that is called when the user selects the confirmation link in the email sent to them:

```
public async Task<ActionResult> ConfirmEmail(string userId, string code)
{
    if (userId == null || code == null)
    {
        return View("Error");
    }
    var result = await UserManager.ConfirmEmailAsync(userId, code);
    if (result.Succeeded)
    {
        return View("ConfirmEmail");
    }
    AddErrors(result);
    return View();
}
```

Once a forgotten password token has been used, it's invalidated. The following code change in the `Create`

method (in the *App_Start\IdentityConfig.cs* file) sets the tokens to expire in 3 hours.

```
if (dataProtectionProvider != null)
 {
    manager.UserTokenProvider =
       new DataProtectorTokenProvider<ApplicationUser>
          (dataProtectionProvider.Create("ASP.NET Identity"))
          {
             TokenLifespan = TimeSpan.FromHours(3)
          };
 }
```

With the code above, the forgotten password and the email confirmation tokens will expire in 3 hours. The default `TokenLifespan` is one day.

The following code shows the email confirmation method:

```
// GET: /Account/ConfirmEmail
[AllowAnonymous]
public async Task<ActionResult> ConfirmEmail(string userId, string code)
{
   if (userId == null || code == null)
   {
      return View("Error");
   }
   IdentityResult result;
   try
   {
      result = await UserManager.ConfirmEmailAsync(userId, code);
   }
   catch (InvalidOperationException ioe)
   {
      // ConfirmEmailAsync throws when the userId is not found.
      ViewBag.errorMessage = ioe.Message;
      return View("Error");
   }

   if (result.Succeeded)
   {
      return View();
   }

   // If we got this far, something failed.
   AddErrors(result);
   ViewBag.errorMessage = "ConfirmEmail failed";
   return View("Error");
}
```

To make your app more secure, ASP.NET Identity supports Two-Factor authentication (2FA). See ASP.NET Identity 2.0: Setting Up Account Validation and Two-Factor Authorization by John Atten. Although you can set account lockout on login password attempt failures, that approach makes your login susceptible to DOS lockouts. We recommend you use account lockout only with 2FA.

Additional resources

- Overview of Custom Storage Providers for ASP.NET Identity
- MVC 5 App with Facebook, Twitter, LinkedIn and Google OAuth2 Sign-on also shows how to add profile information to the users table.
- ASP.NET MVC and Identity 2.0: Understanding the Basics by John Atten.
- Introduction to ASP.NET Identity

- Announcing RTM of ASP.NET Identity 2.0.0 by Pranav Rastogi.

Two-factor authentication using SMS and email with ASP.NET Identity

2/20/2020 • 17 minutes to read • Edit Online

by Hao Kung, Pranav Rastogi, Rick Anderson, Suhas Joshi

> This tutorial will show you how to set up Two-factor authentication (2FA) using SMS and email.
>
> This article was written by Rick Anderson (@RickAndMSFT), Pranav Rastogi (@rustd), Hao Kung, and Suhas Joshi. The NuGet sample was written primarily by Hao Kung.

This topic covers the following:

- Building the Identity sample
- Set up SMS for Two-factor authentication
- Enable Two-factor authentication
- How to register a Two-factor authentication provider
- Combine social and local login accounts
- Account lockout from brute force attacks

Building the Identity sample

In this section, you'll use NuGet to download a sample we will work with. Start by installing and running Visual Studio Express 2013 for Web or Visual Studio 2013. Install Visual Studio 2013 Update 2 or higher.

> **NOTE**
>
> Warning: You must install Visual Studio 2013 Update 2 to complete this tutorial.

1. Create a new ***empty*** ASP.NET Web project.

2. In the Package Manager Console, enter the following the following commands:

   ```
   Install-Package SendGrid
   Install-Package -Prerelease Microsoft.AspNet.Identity.Samples
   ```

   In this tutorial, we'll use SendGrid to send email and Twilio or ASPSMS for sms texting. The `Identity.Samples` package installs the code we will be working with.

3. Set the project to use SSL.

4. *Optional*: Follow the instructions in my Email confirmation tutorial to hook up SendGrid and then run the app and register an email account.

5. *Optional:* Remove the demo email link confirmation code from the sample (The `ViewBag.Link` code in the account controller. See the `DisplayEmail` and `ForgotPasswordConfirmation` action methods and razor views).

6. *Optional:* Remove the `ViewBag.Status` code from the Manage and Account controllers and from the *Views\Account\VerifyCode.cshtml* and *Views\Manage\VerifyPhoneNumber.cshtml* razor views. Alternatively, you can keep the `ViewBag.Status` display to test how this app works locally without having

to hook up and send email and SMS messages.

> **NOTE**
>
> Warning: If you change any of the security settings in this sample, productions apps will need to undergo a security audit that explicitly calls out the changes made.

Set up SMS for Two-factor authentication

This tutorial provides instructions for using either Twilio or ASPSMS but you can use any other SMS provider.

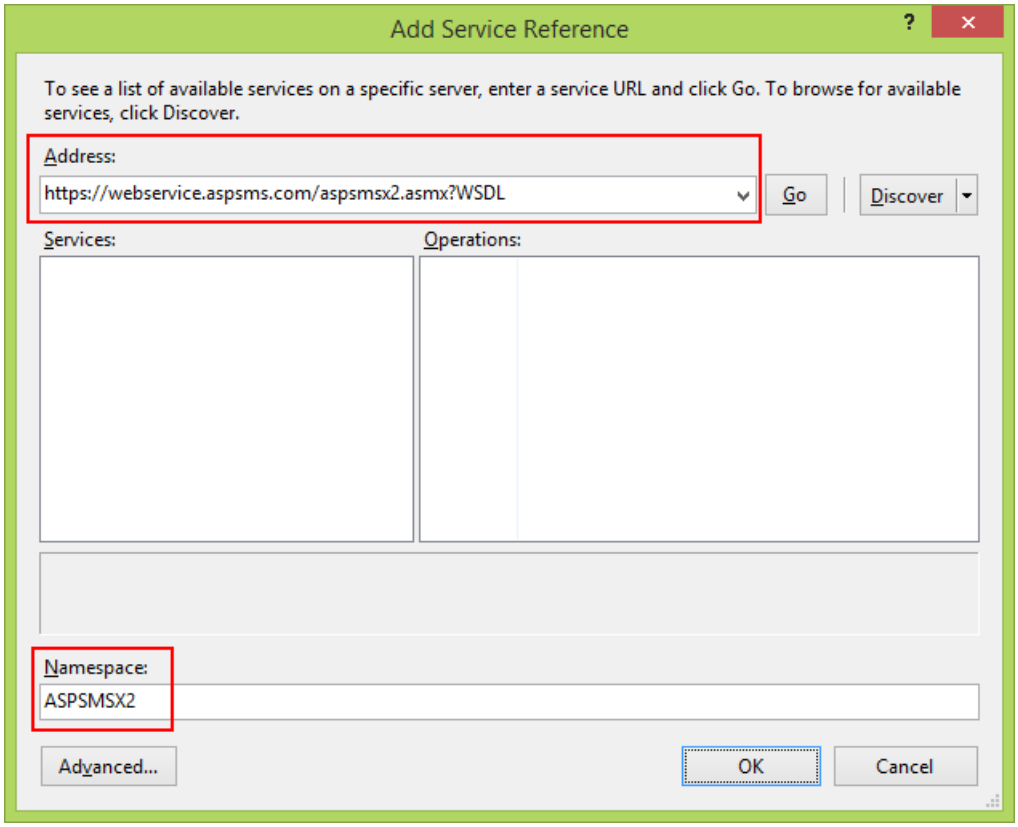
1. **Creating a User Account with an SMS provider**

   Create a Twilio or an ASPSMS account.

2. **Installing additional packages or adding service references**

   Twilio:
   In the Package Manager Console, enter the following command:
   `Install-Package Twilio`

   ASPSMS:
   The following service reference needs to be added:



   Address:
   `https://webservice.aspsms.com/aspsmsx2.asmx?WSDL`

   Namespace:
   `ASPSMSX2`

3. **Figuring out SMS Provider User credentials**

   Twilio:
   From the **Dashboard** tab of your Twilio account, copy the **Account SID** and **Auth token**.

ASPSMS:
From your account settings, navigate to **Userkey** and copy it together with your self-defined **Password**.

We will later store these values in the variables `SMSAccountIdentification` and `SMSAccountPassword` .

4. **Specifying SenderID / Originator**

   Twilio:
   From the **Numbers** tab, copy your Twilio phone number.

   ASPSMS:
   Within the **Unlock Originators** Menu, unlock one or more Originators or choose an alphanumeric Originator (Not supported by all networks).

   We will later store this value in the variable `SMSAccountFrom` .

5. **Transferring SMS provider credentials into app**

   Make the credentials and sender phone number available to the app:

```
public static class Keys
{
   public static string SMSAccountIdentification = "My Idenfitication";
   public static string SMSAccountPassword = "My Password";
   public static string SMSAccountFrom = "+15555551234";
}
```

> **WARNING**
> Security - Never store sensitive data in your source code. The account and credentials are added to the code above to keep the sample simple. See Jon Atten's ASP.NET MVC: Keep Private Settings Out of Source Control.

6. **Implementation of data transfer to SMS provider**

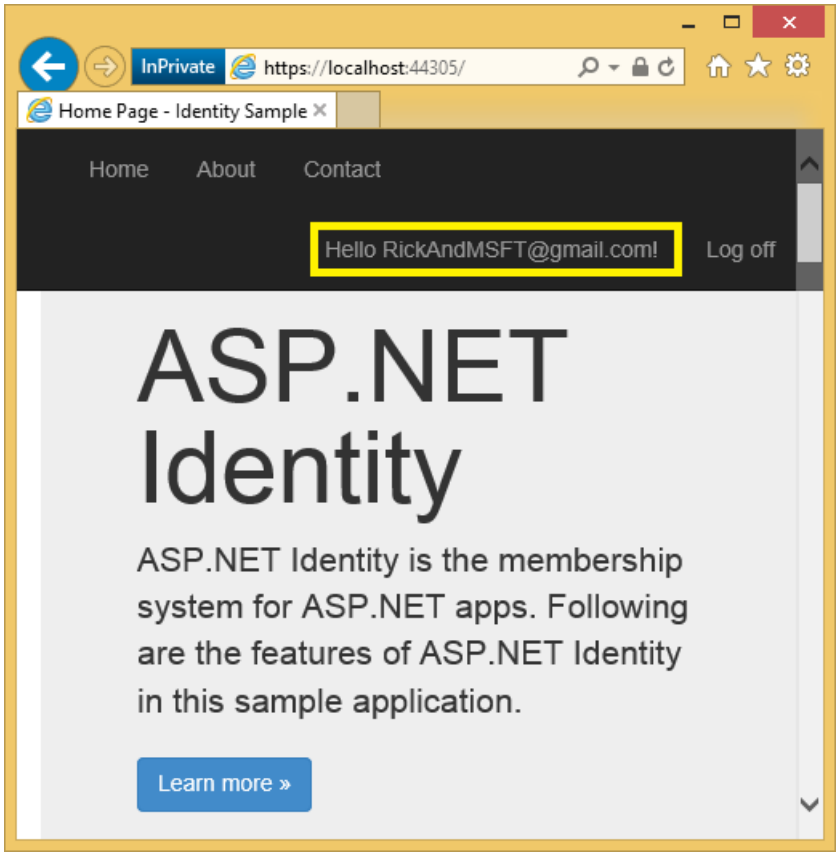
   Configure the `SmsService` class in the *App_Start\IdentityConfig.cs* file.

   Depending on the used SMS provider activate either the **Twilio** or the **ASPSMS** section:

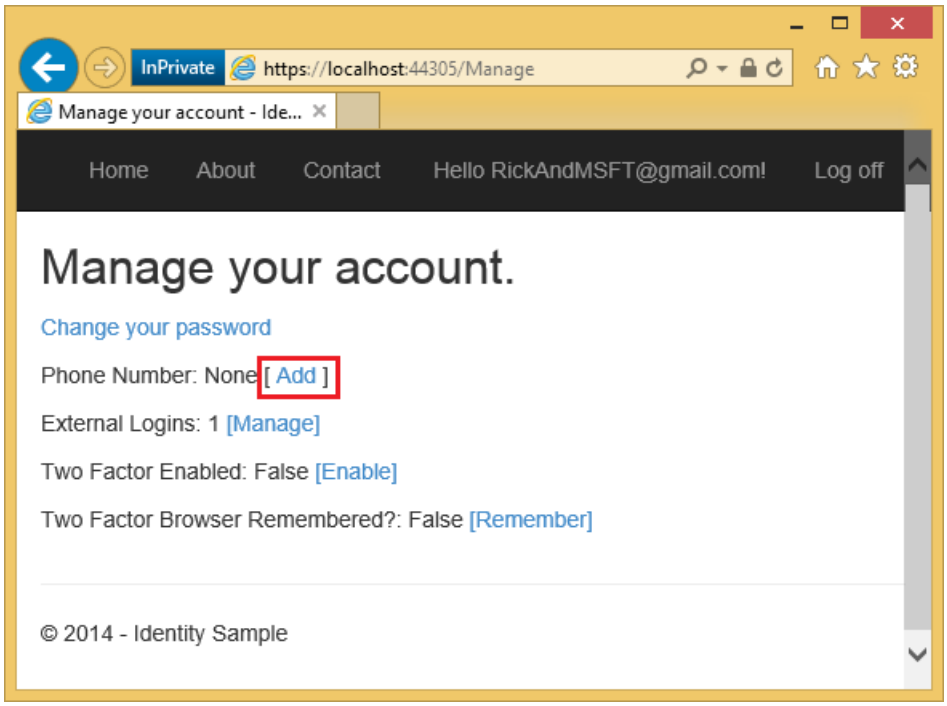
```
public class SmsService : IIdentityMessageService
{
    public Task SendAsync(IdentityMessage message)
    {
        // Twilio Begin
        // var Twilio = new TwilioRestClient(
        //   Keys.SMSAccountIdentification,
        //   Keys.SMSAccountPassword);
        // var result = Twilio.SendMessage(
        //   Keys.SMSAccountFrom,
        //   message.Destination, message.Body
        // );
        // Status is one of Queued, Sending, Sent, Failed or null if the number is not valid
        // Trace.TraceInformation(result.Status);
        // Twilio doesn't currently have an async API, so return success.
        // return Task.FromResult(0);
        // Twilio End

        // ASPSMS Begin
        // var soapSms = new WebApplication1.ASPSMSX2.ASPSMSX2SoapClient("ASPSMSX2Soap");
        // soapSms.SendSimpleTextSMS(
        //   Keys.SMSAccountIdentification,
        //   Keys.SMSAccountPassword,
        //   message.Destination,
        //   Keys.SMSAccountFrom,
        //   message.Body);
        // soapSms.Close();
        // return Task.FromResult(0);
        // ASPSMS End
    }
}
```

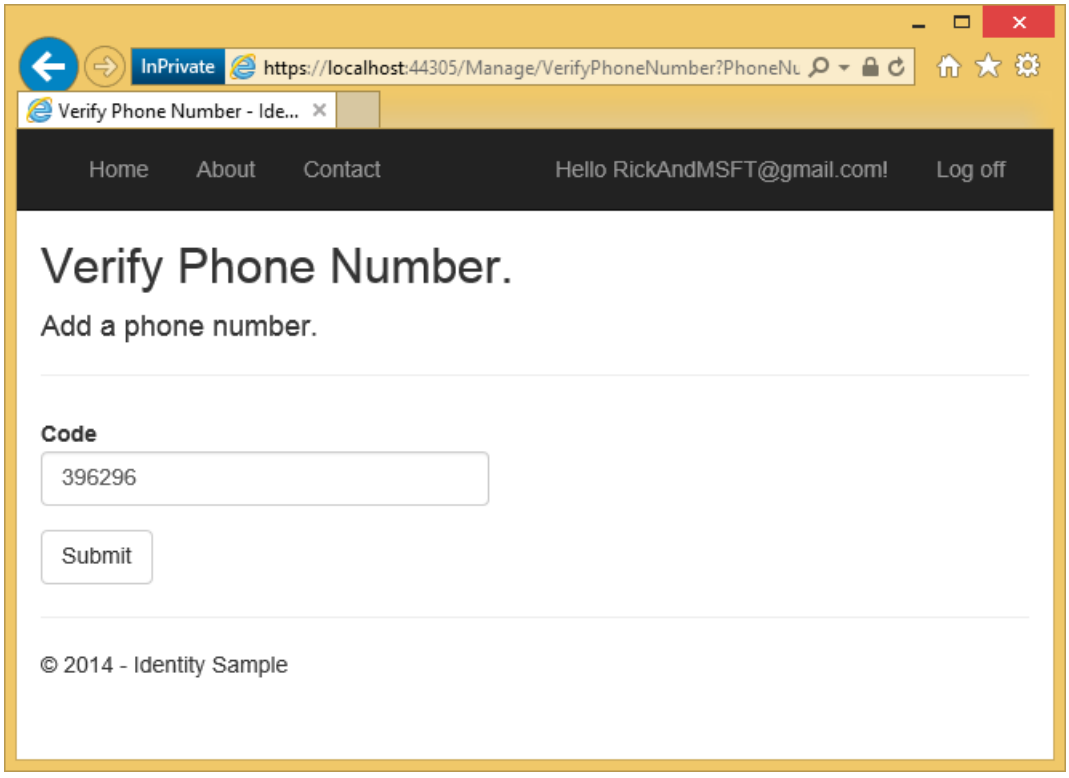
7. Run the app and log in with the account you previously registered.
8. Click on your User ID, which activates the `Index` action method in `Manage` controller.



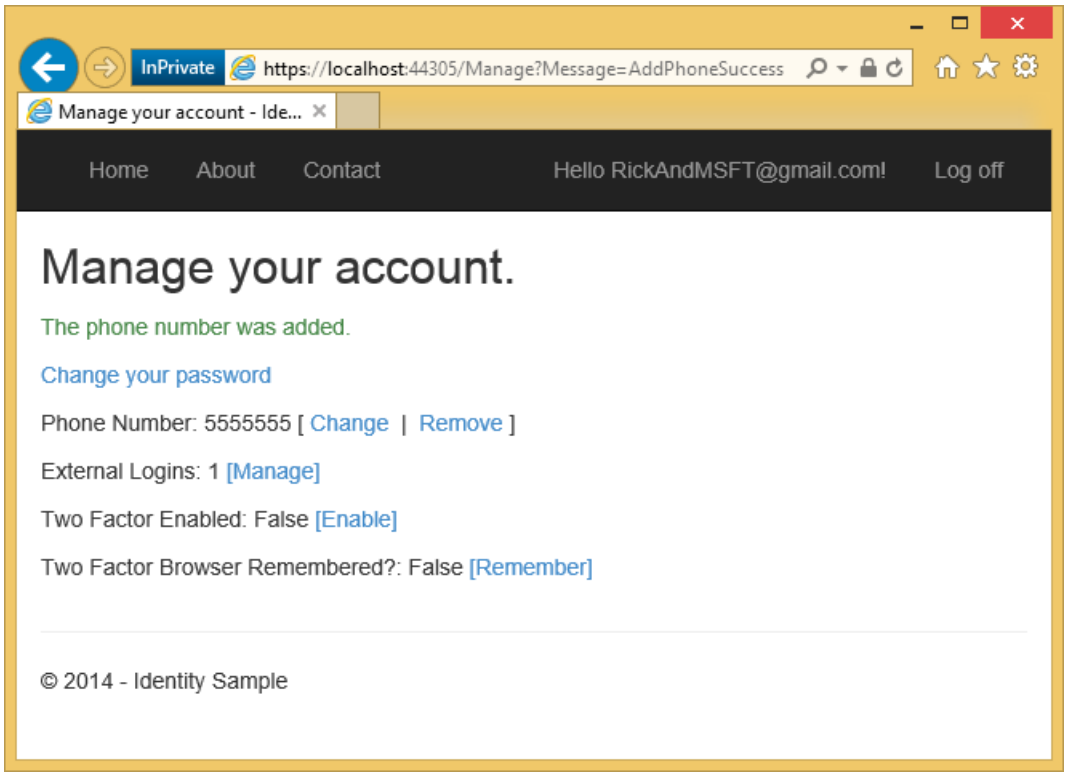
9. Click Add.



10. In a few seconds you will get a text message with the verification code. Enter it and press **Submit**.



11. The Manage view shows your phone number was added.



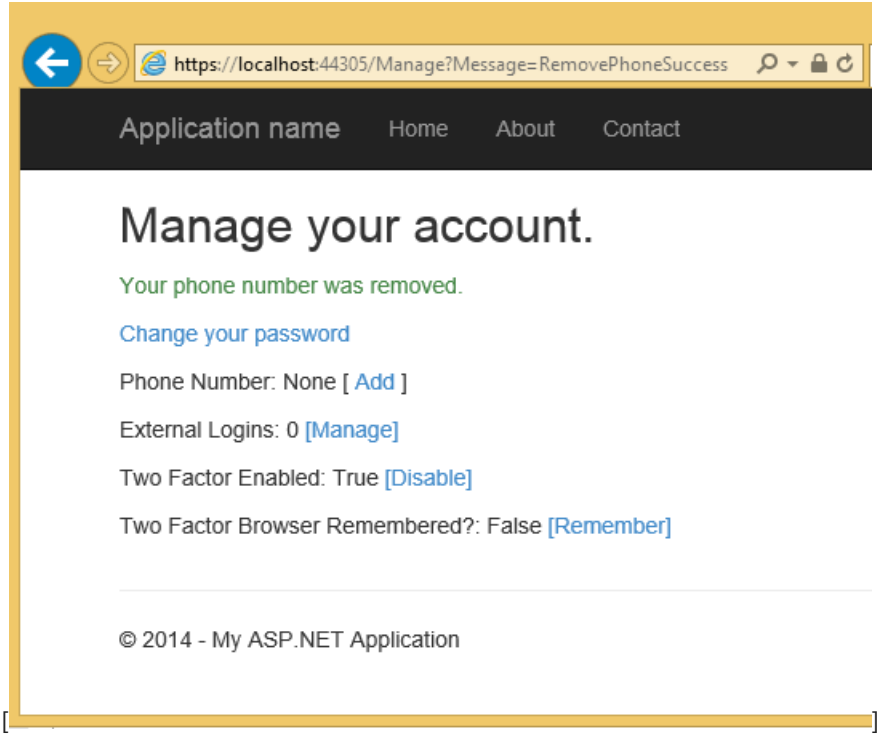
Examine the code

```
// GET: /Account/Index
public async Task<ActionResult> Index(ManageMessageId? message)
{
    ViewBag.StatusMessage =
        message == ManageMessageId.ChangePasswordSuccess ? "Your password has been changed."
        : message == ManageMessageId.SetPasswordSuccess ? "Your password has been set."
        : message == ManageMessageId.SetTwoFactorSuccess ? "Your two factor provider has been set."
        : message == ManageMessageId.Error ? "An error has occurred."
        : message == ManageMessageId.AddPhoneSuccess ? "The phone number was added."
        : message == ManageMessageId.RemovePhoneSuccess ? "Your phone number was removed."
        : "";

    var model = new IndexViewModel
    {
        HasPassword = HasPassword(),
        PhoneNumber = await UserManager.GetPhoneNumberAsync(User.Identity.GetUserId()),
        TwoFactor = await UserManager.GetTwoFactorEnabledAsync(User.Identity.GetUserId()),
        Logins = await UserManager.GetLoginsAsync(User.Identity.GetUserId()),
        BrowserRemembered = await
AuthenticationManager.TwoFactorBrowserRememberedAsync(User.Identity.GetUserId())
    };
    return View(model);
}
```

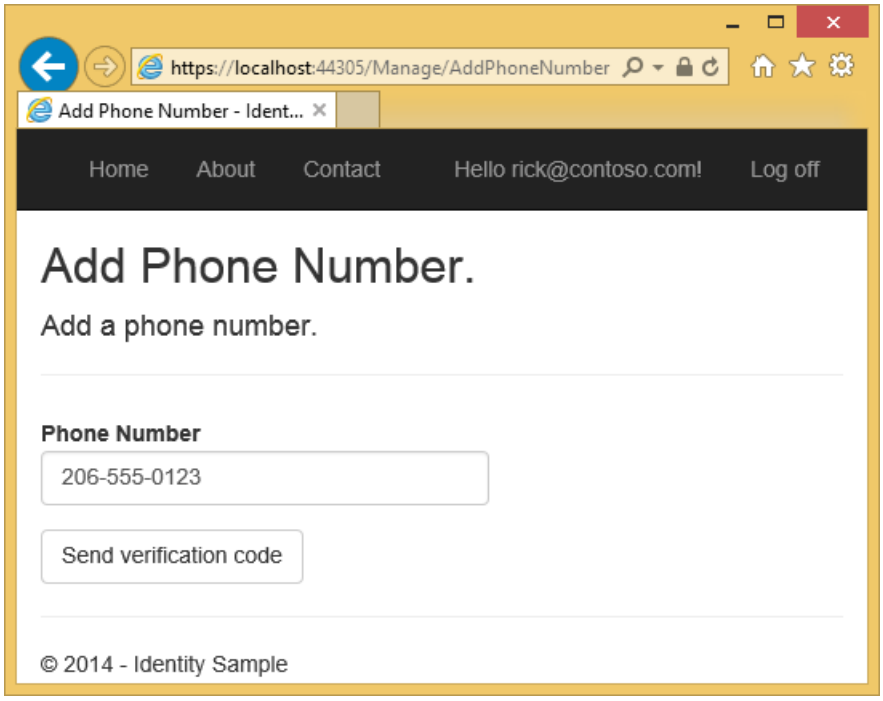
The `Index` action method in `Manage` controller sets the status message based on your previous action and provides links to change your local password or add a local account. The `Index` method also displays the state or your 2FA phone number, external logins, 2FA enabled, and remember 2FA method for this browser(explained later). Clicking on your user ID (email) in the title bar doesn't pass a message. Clicking on the **Phone Number : remove** link passes `Message=RemovePhoneSuccess` as a query string.

`https://localhost:44300/Manage?Message=RemovePhoneSuccess`



The `AddPhoneNumber` action method displays a dialog box to enter a phone number that can receive SMS messages.

```
// GET: /Account/AddPhoneNumber
public ActionResult AddPhoneNumber()
{
   return View();
}
```

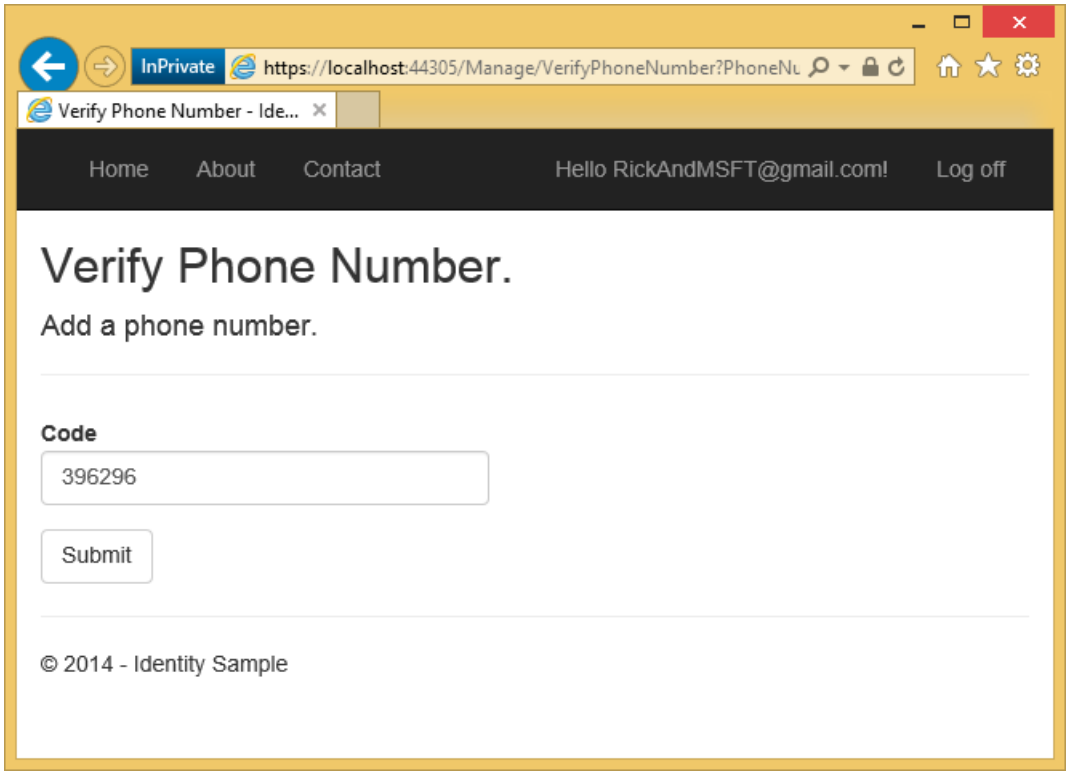


Clicking on the **Send verification code** button posts the phone number to the HTTP POST `AddPhoneNumber` action method.

```
// POST: /Account/AddPhoneNumber
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> AddPhoneNumber(AddPhoneNumberViewModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }

    // Generate the token
    var code = await UserManager.GenerateChangePhoneNumberTokenAsync(
                                User.Identity.GetUserId(), model.Number);
    if (UserManager.SmsService != null)
    {
        var message = new IdentityMessage
        {
            Destination = model.Number,
            Body = "Your security code is: " + code
        };
        // Send token
        await UserManager.SmsService.SendAsync(message);
    }
    return RedirectToAction("VerifyPhoneNumber", new { PhoneNumber = model.Number });
}
```

The `GenerateChangePhoneNumberTokenAsync` method generates the security token which will be set in the SMS message. If the SMS service has been configured, the token is sent as the string "Your security code is <token>". The `SmsService.SendAsync` method to is called asynchronously, then the app is redirected to the `VerifyPhoneNumber` action method (which displays the following dialog), where you can enter the verification code.



Once you enter the code and click submit, the code is posted to the HTTP POST `VerifyPhoneNumber` action method.

```
// POST: /Account/VerifyPhoneNumber
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> VerifyPhoneNumber(VerifyPhoneNumberViewModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
    var result = await UserManager.ChangePhoneNumberAsync(User.Identity.GetUserId(), model.PhoneNumber,
model.Code);
    if (result.Succeeded)
    {
        var user = await UserManager.FindByIdAsync(User.Identity.GetUserId());
        if (user != null)
        {
            await SignInAsync(user, isPersistent: false);
        }
        return RedirectToAction("Index", new { Message = ManageMessageId.AddPhoneSuccess });
    }
    // If we got this far, something failed, redisplay form
    ModelState.AddModelError("", "Failed to verify phone");
    return View(model);
}
```

The `ChangePhoneNumberAsync` method checks the posted security code. If the code is correct, the phone number is added to the `PhoneNumber` field of the `AspNetUsers` table. If that call is successful, the `SignInAsync` method is called:

```
private async Task SignInAsync(ApplicationUser user, bool isPersistent)
{
   // Clear the temporary cookies used for external and two factor sign ins
    AuthenticationManager.SignOut(DefaultAuthenticationTypes.ExternalCookie,
       DefaultAuthenticationTypes.TwoFactorCookie);
    AuthenticationManager.SignIn(new AuthenticationProperties
    {
       IsPersistent = isPersistent
    },
       await user.GenerateUserIdentityAsync(UserManager));
}
```

The `isPersistent` parameter sets whether the authentication session is persisted across multiple requests.

When you change your security profile, a new security stamp is generated and stored in the `SecurityStamp` field of the *AspNetUsers* table. Note, the `SecurityStamp` field is different from the security cookie. The security cookie is not stored in the `AspNetUsers` table (or anywhere else in the Identity DB). The security cookie token is self-signed using DPAPI and is created with the `UserId, SecurityStamp` and expiration time information.

The cookie middleware checks the cookie on each request. The `SecurityStampValidator` method in the `Startup` class hits the DB and checks security stamp periodically, as specified with the `validateInterval`. This only happens every 30 minutes (in our sample) unless you change your security profile. The 30 minute interval was chosen to minimize trips to the database.

The `SignInAsync` method needs to be called when any change is made to the security profile. When the security profile changes, the database is updates the `SecurityStamp` field, and without calling the `SignInAsync` method you would stay logged in *only* until the next time the OWIN pipeline hits the database (the `validateInterval`). You can test this by changing the `SignInAsync` method to return immediately, and setting the cookie `validateInterval` property from 30 minutes to 5 seconds:

```
private async Task SignInAsync(ApplicationUser user, bool isPersistent)
{
   return;

   // Clear any partial cookies from external or two factor partial sign ins
    AuthenticationManager.SignOut(DefaultAuthenticationTypes.ExternalCookie,
       DefaultAuthenticationTypes.TwoFactorCookie);
    AuthenticationManager.SignIn(new AuthenticationProperties
    {
       IsPersistent = isPersistent
    },
       await user.GenerateUserIdentityAsync(UserManager));
}
```

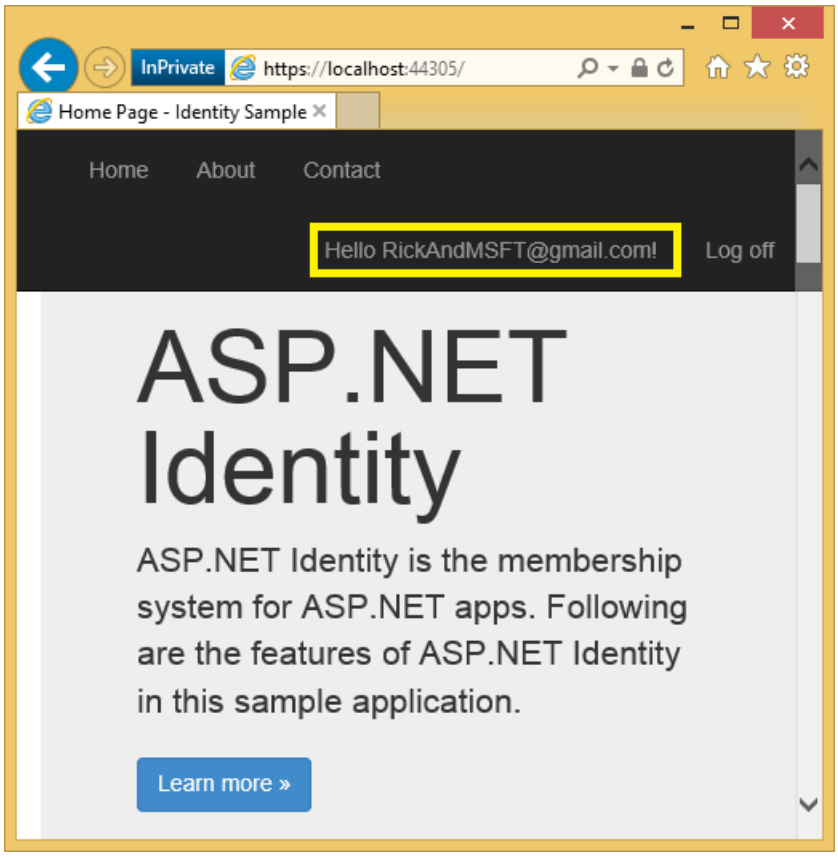
```
public void ConfigureAuth(IAppBuilder app) {
    // Configure the db context, user manager and role manager to use a single instance per request
    app.CreatePerOwinContext(ApplicationDbContext.Create);
    app.CreatePerOwinContext<ApplicationUserManager>(ApplicationUserManager.Create);
    app.CreatePerOwinContext<ApplicationRoleManager>(ApplicationRoleManager.Create);
    app.CreatePerOwinContext<ApplicationSignInManager>(ApplicationSignInManager.Create);

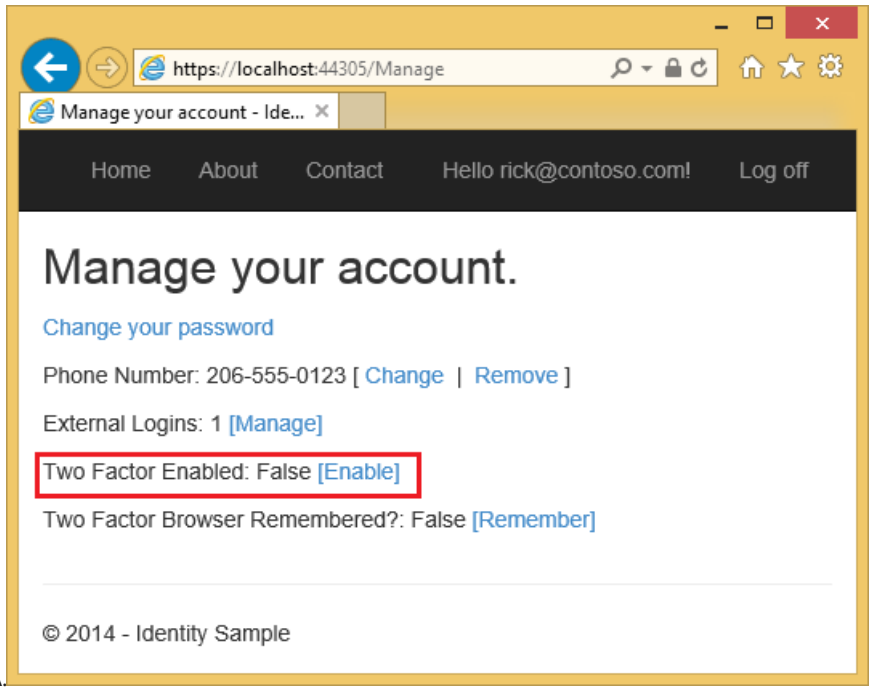
    // Enable the application to use a cookie to store information for the signed in user
    // and to use a cookie to temporarily store information about a user logging in with a
    // third party login provider
    // Configure the sign in cookie
    app.UseCookieAuthentication(new CookieAuthenticationOptions {
        AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
        LoginPath = new PathString("/Account/Login"),
        Provider = new CookieAuthenticationProvider {
            // Enables the application to validate the security stamp when the user logs in.
            // This is a security feature which is used when you change a password or add
            // an external login to your account.
            OnValidateIdentity = SecurityStampValidator.OnValidateIdentity<ApplicationUserManager,
ApplicationUser>(
                //validateInterval: TimeSpan.FromMinutes(30),
                validateInterval: TimeSpan.FromSeconds(5),
                regenerateIdentity: (manager, user) => user.GenerateUserIdentityAsync(manager))
        }
    });
    app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);
```

With the above code changes, you can change your security profile (for example, by changing the state of **Two Factor Enabled**) and you will be logged out in 5 seconds when the `SecurityStampValidator.OnValidateIdentity` method fails. Remove the return line in the `SignInAsync` method, make another security profile change and you will not be logged out. The `SignInAsync` method generates a new security cookie.

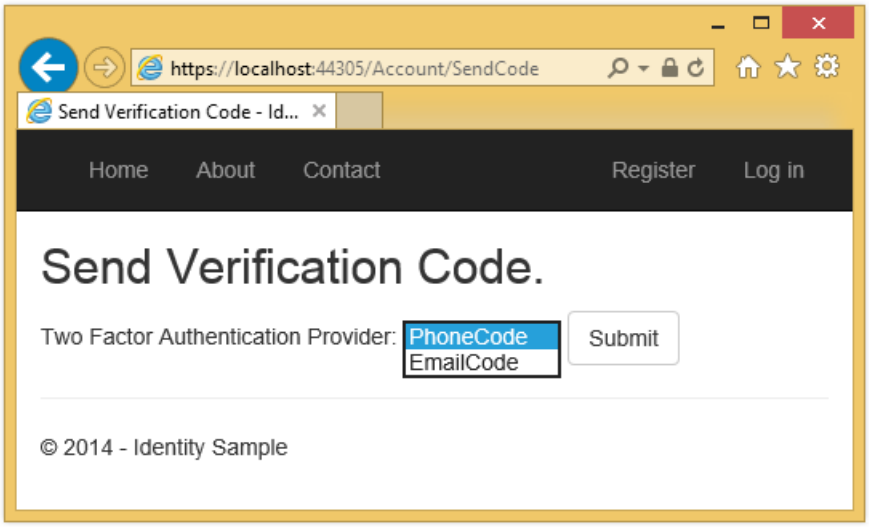
Enable two-factor authentication

In the sample app, you need to use the UI to enable two-factor authentication (2FA). To enable 2FA, click on your user ID (email alias) in the navigation bar.

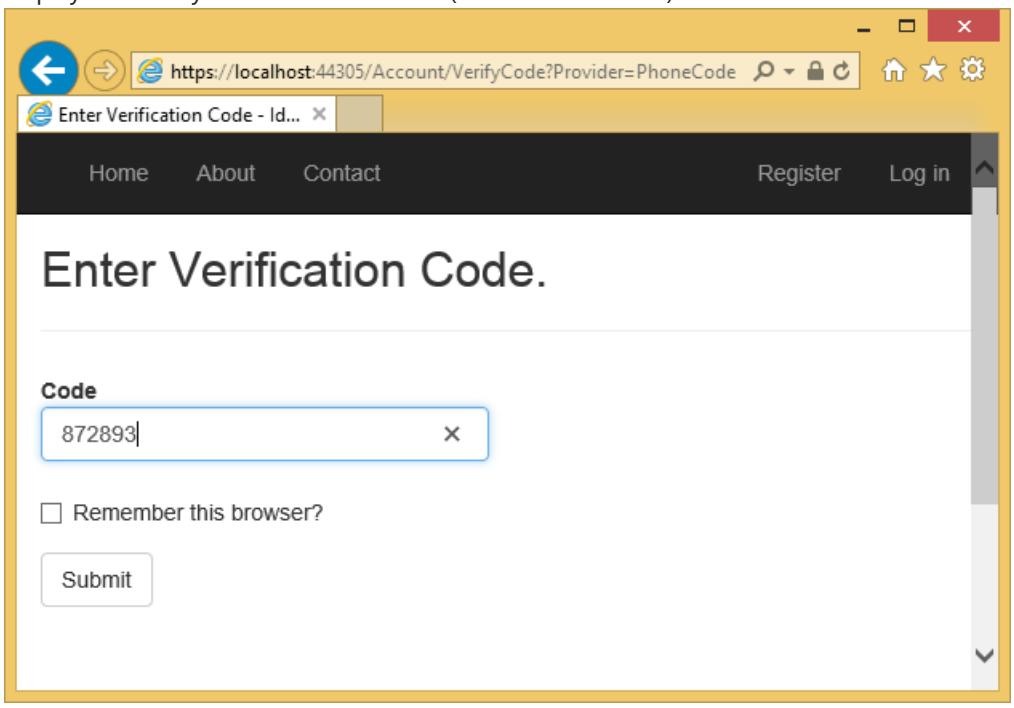




Click on enable 2FA. Log out, then log back in. If you've enabled email (see my previous tutorial), you can select the SMS or email for 2FA.



The Verify Code page is displayed where you can enter the code (from SMS or email).



Clicking on the **Remember this browser** check box will exempt you from needing to use 2FA to log on with that computer and browser. Enabling 2FA and clicking on the **Remember this browser** will provide you with strong 2FA protection from malicious users trying to access your account, as long as they don't have access to your computer. You can do this on any private machine you regularly use. By setting **Remember this browser**, you get the added security of 2FA from computers you don't regularly use, and you get the convenience on not having to go through 2FA on your own computers.

How to register a Two-factor authentication provider

When you create a new MVC project, the *IdentityConfig.cs* file contains the following code to register a Two-factor authentication provider:

```
public static ApplicationUserManager Create(
   IdentityFactoryOptions<ApplicationUserManager> options,
   IOwinContext context)
{
    var manager = new ApplicationUserManager(
       new UserStore<ApplicationUser>(context.Get<ApplicationDbContext>()));
    // Configure validation logic for usernames
    manager.UserValidator = new UserValidator<ApplicationUser>(manager)
    {
        AllowOnlyAlphanumericUserNames = false,
        RequireUniqueEmail = true
    };
    // Configure validation logic for passwords
    manager.PasswordValidator = new PasswordValidator
    {
        RequiredLength = 6,
        RequireNonLetterOrDigit = true,
        RequireDigit = true,
        RequireLowercase = true,
        RequireUppercase = true,
    };
    // Register two factor authentication providers. This application uses Phone and Emails as a
    // step of receiving a code for verifying the user
    // You can write your own provider and plug it in here.
    manager.RegisterTwoFactorProvider("PhoneCode", new PhoneNumberTokenProvider<ApplicationUser>
    {
        MessageFormat = "Your security code is: {0}"
    });
    manager.RegisterTwoFactorProvider("EmailCode", new EmailTokenProvider<ApplicationUser>
    {
        Subject = "Security Code",
        BodyFormat = "Your security code is: {0}"
    });
    manager.EmailService = new EmailService();
    manager.SmsService = new SmsService();

    var dataProtectionProvider = options.DataProtectionProvider;
    if (dataProtectionProvider != null)
    {
        manager.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser>
           (dataProtectionProvider.Create("ASP.NET Identity"));
    }
    return manager;
}
```

Add a phone number for 2FA

The `AddPhoneNumber` action method in the `Manage` controller generates a security token and sends it to the phone number you have provided.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> AddPhoneNumber(AddPhoneNumberViewModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
    // Generate the token and send it
    var code = await UserManager.GenerateChangePhoneNumberTokenAsync(
       User.Identity.GetUserId(), model.Number);
    if (UserManager.SmsService != null)
    {
        var message = new IdentityMessage
        {
            Destination = model.Number,
            Body = "Your security code is: " + code
        };
        await UserManager.SmsService.SendAsync(message);
    }
    return RedirectToAction("VerifyPhoneNumber", new { PhoneNumber = model.Number });
}
```

After sending the token, it redirects to the `VerifyPhoneNumber` action method, where you can enter the code to register SMS for 2FA. SMS 2FA is not used until you have verified the phone number.

Enabling 2FA

The `EnableTFA` action method enables 2FA:

```
// POST: /Manage/EnableTFA
[HttpPost]
public async Task<ActionResult> EnableTFA()
{
    await UserManager.SetTwoFactorEnabledAsync(User.Identity.GetUserId(), true);
    var user = await UserManager.FindByIdAsync(User.Identity.GetUserId());
    if (user != null)
    {
        await SignInAsync(user, isPersistent: false);
    }
    return RedirectToAction("Index", "Manage");
}
```

Note the `SignInAsync` must be called because enable 2FA is a change to the security profile. When 2FA is enabled, the user will need to use 2FA to log in, using the 2FA approaches they have registered (SMS and email in the sample).

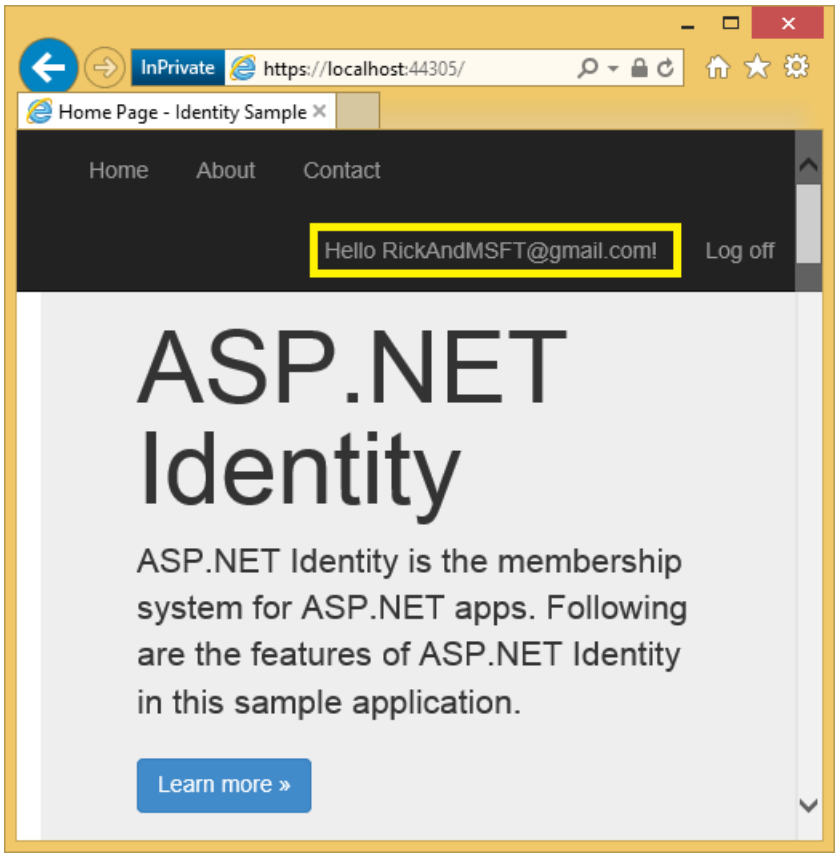
You can add more 2FA providers such as QR code generators or you can write you own (See Using Google Authenticator with ASP.NET Identity).

> **NOTE**
>
> The 2FA codes are generated using Time-based One-time Password Algorithm and codes are valid for six minutes. If you take more than six minutes to enter the code, you'll get an Invalid code error message.

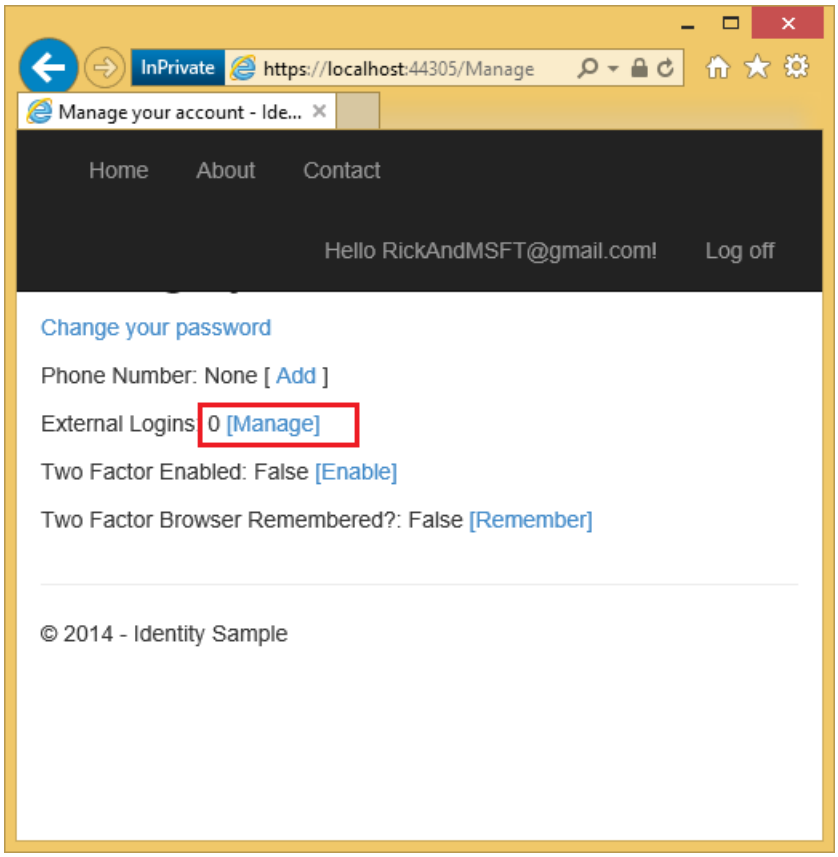
Combine social and local login accounts

You can combine local and social accounts by clicking on your email link. In the following sequence "RickAndMSFT@gmail.com" is first created as a local login, but you can create the account as a social log in

first, then add a local login.



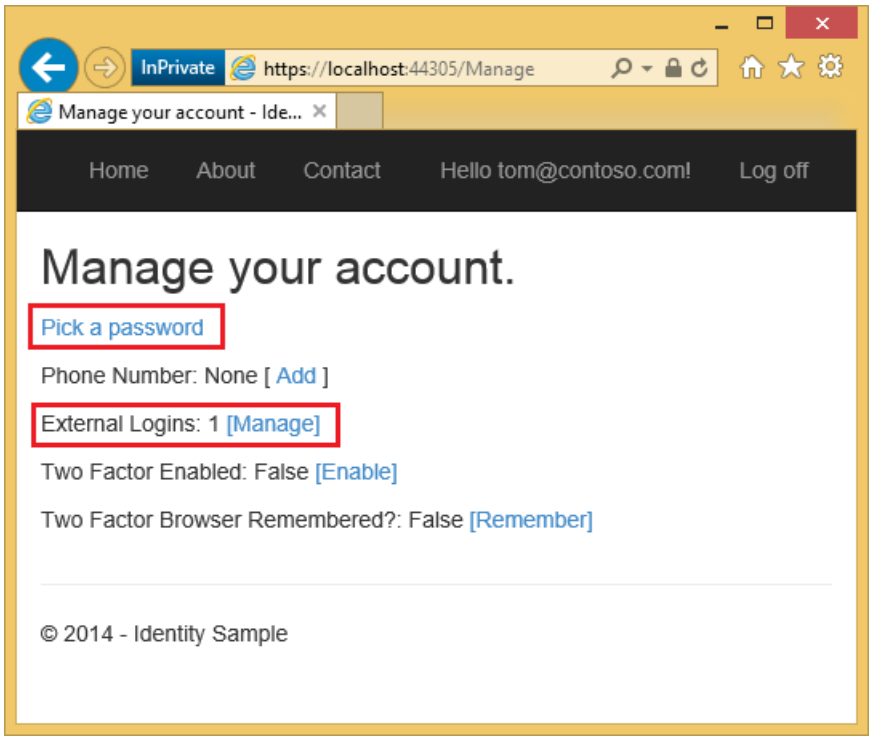
Click on the **Manage** link. Note the 0 external (social logins) associated with this account.



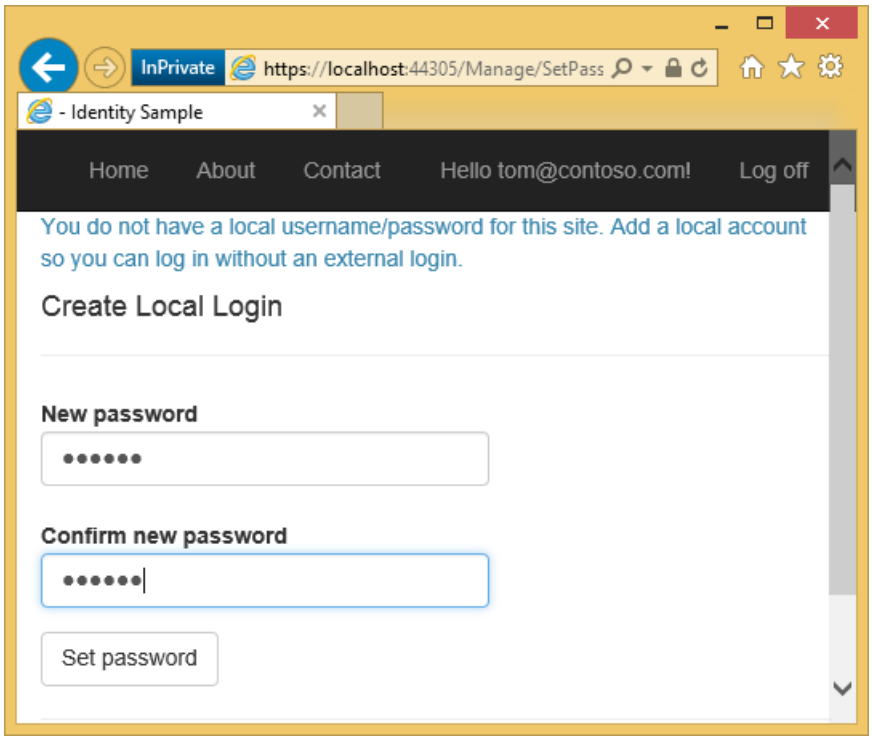
Click the link to another log in service and accept the app requests. The two accounts have been combined, you will be able to log on with either account. You might want your users to add local accounts in case their social log in authentication service is down, or more likely they have lost access to their social account.

In the following image, Tom is a social log in (which you can see from the **External Logins: 1** shown on the

page).



Clicking on **Pick a password** allows you to add a local log on associated with the same account.



Account lockout from brute force attacks

You can protect the accounts on your app from dictionary attacks by enabling user lockout. The following code in the `ApplicationUserManager Create` method configures lock out:

```
// Configure user lockout defaults
manager.UserLockoutEnabledByDefault = true;
manager.DefaultAccountLockoutTimeSpan = TimeSpan.FromMinutes(5);
manager.MaxFailedAccessAttemptsBeforeLockout = 5;
```

The code above enables lockout for two factor authentication only. While you can enable lock out for logins by

changing `shouldLockout` to true in the `Login` method of the account controller, we recommend you not enable lock out for logins because it makes the account susceptible to DOS login attacks. In the sample code, lockout is disabled for the admin account created in the `ApplicationDbInitializer Seed` method:

```
public static void InitializeIdentityForEF(ApplicationDbContext db)
{
    var userManager = HttpContext.Current.GetOwinContext().GetUserManager<ApplicationUserManager>();
    var roleManager = HttpContext.Current.GetOwinContext().Get<ApplicationRoleManager>();
    const string name = "admin@example.com";
    const string roleName = "Admin";

    //Create Role Admin if it does not exist
    var role = roleManager.FindByName(roleName);
    if (role == null)
    {
        role = new IdentityRole(roleName);
        var roleresult = roleManager.Create(role);
    }

    var user = userManager.FindByName(name);
    if (user == null)
    {
        user = new ApplicationUser { UserName = name, Email = name };
        var result = userManager.Create(user, GetSecurePassword());
        result = userManager.SetLockoutEnabled(user.Id, false);
    }

    // Add user admin to Role Admin if not already added
    var rolesForUser = userManager.GetRoles(user.Id);
    if (!rolesForUser.Contains(role.Name))
    {
        var result = userManager.AddToRole(user.Id, role.Name);
    }
}
```

Requiring a user to have a validated email account

The following code requires a user to have a validated email account before they can log in:

```
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }

   // Require the user to have a confirmed email before they can log on.
    var user = await UserManager.FindByNameAsync(model.Email);
    if (user != null)
    {
       if (!await UserManager.IsEmailConfirmedAsync(user.Id))
       {
          ViewBag.errorMessage = "You must have a confirmed email to log on.";
          return View("Error");
       }
    }
    // This doen't count login failures towards lockout only two factor authentication
    // To enable password failures to trigger lockout, change to shouldLockout: true
    var result = await SignInManager.PasswordSignInAsync(model.Email, model.Password,
       model.RememberMe, shouldLockout: false);
    switch (result)
    {
        case SignInStatus.Success:
            return RedirectToLocal(returnUrl);
        case SignInStatus.LockedOut:
            return View("Lockout");
        case SignInStatus.RequiresVerification:
            return RedirectToAction("SendCode", new { ReturnUrl = returnUrl });
        case SignInStatus.Failure:
        default:
            ModelState.AddModelError("", "Invalid login attempt.");
            return View(model);
    }
}
```

How SignInManager checks for 2FA requirement

Both the local log in and social log in check to see if 2FA is enabled. If 2FA is enabled, the `SignInManager` logon method returns `SignInStatus.RequiresVerification`, and the user will be redirected to the `SendCode` action method, where they will have to enter the code to complete the log in sequence. If the user has RememberMe is set on the users local cookie, the `SignInManager` will return `SignInStatus.Success` and they will not have to go through 2FA.

```
public async Task<ActionResult> Login(LoginViewModel model, string returnUrl)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }

   // Require the user to have a confirmed email before they can log on.
    var user = await UserManager.FindByNameAsync(model.Email);
    if (user != null)
    {
       if (!await UserManager.IsEmailConfirmedAsync(user.Id))
       {
          ViewBag.errorMessage = "You must have a confirmed email to log on.";
          return View("Error");
       }
    }
    // This doen't count login failures towards lockout only two factor authentication
    // To enable password failures to trigger lockout, change to shouldLockout: true
    var result = await SignInManager.PasswordSignInAsync(model.Email, model.Password,
       model.RememberMe, shouldLockout: false);
    switch (result)
    {
        case SignInStatus.Success:
            return RedirectToLocal(returnUrl);
        case SignInStatus.LockedOut:
            return View("Lockout");
        case SignInStatus.RequiresVerification:
            return RedirectToAction("SendCode", new { ReturnUrl = returnUrl });
        case SignInStatus.Failure:
        default:
            ModelState.AddModelError("", "Invalid login attempt.");
            return View(model);
    }
}
```

```
public async Task<ActionResult> ExternalLoginCallback(string returnUrl)
{
    var loginInfo = await AuthenticationManager.GetExternalLoginInfoAsync();
    if (loginInfo == null)
    {
        return RedirectToAction("Login");
    }

    // Sign in the user with this external login provider if the user already has a login
    var result = await SignInManager.ExternalSignInAsync(loginInfo, isPersistent: false);
    switch (result)
    {
        case SignInStatus.Success:
            return RedirectToLocal(returnUrl);
        case SignInStatus.LockedOut:
            return View("Lockout");
        case SignInStatus.RequiresVerification:
            return RedirectToAction("SendCode", new { ReturnUrl = returnUrl });
        case SignInStatus.Failure:
        default:
            // If the user does not have an account, then prompt the user to create an account
            ViewBag.ReturnUrl = returnUrl;
            ViewBag.LoginProvider = loginInfo.Login.LoginProvider;
            return View("ExternalLoginConfirmation", new ExternalLoginConfirmationViewModel { Email =
loginInfo.Email });
    }
}
```

The following code shows the `SendCode` action method. A SelectListItem is created with all the 2FA methods

enabled for the user. The SelectListItem is passed to the DropDownListFor helper, which allows the user to select the 2FA approach (typically email and SMS).

```
public async Task<ActionResult> SendCode(string returnUrl)
{
    var userId = await SignInManager.GetVerifiedUserIdAsync();
    if (userId == null)
    {
        return View("Error");
    }
    var userFactors = await UserManager.GetValidTwoFactorProvidersAsync(userId);
    var factorOptions = userFactors.Select(purpose => new SelectListItem { Text = purpose, Value = purpose
}).ToList();
    return View(new SendCodeViewModel { Providers = factorOptions, ReturnUrl = returnUrl });
}
```

Once the user posts the 2FA approach, the `HTTP POST SendCode` action method is called, the `SignInManager` sends the 2FA code, and the user is redirected to the `VerifyCode` action method where they can enter the code to complete the log in.

```
//
// POST: /Account/SendCode
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> SendCode(SendCodeViewModel model)
{
    if (!ModelState.IsValid)
    {
        return View();
    }

    // Generate the token and send it
    if (!await SignInManager.SendTwoFactorCodeAsync(model.SelectedProvider))
    {
        return View("Error");
    }
    return RedirectToAction("VerifyCode", new { Provider = model.SelectedProvider, ReturnUrl =
model.ReturnUrl });
}
```

2FA Lockout

Although you can set account lockout on login password attempt failures, that approach makes your login susceptible to DOS lockouts. We recommend you use account lockout only with 2FA. When the `ApplicationUserManager` is created, the sample code sets 2FA lockout and `MaxFailedAccessAttemptsBeforeLockout` to five. Once a user logs in (through local account or social account), each failed attempt at 2FA is stored, and if the maximum attempts is reached, the user is locked out for five minutes (you can set the lock out time with `DefaultAccountLockoutTimeSpan`).

Additional Resources

- ASP.NET Identity Recommended Resources Complete list of Identity blogs, videos, tutorials and great SO links.
- MVC 5 App with Facebook, Twitter, LinkedIn and Google OAuth2 Sign-on also shows how to add profile information to the users table.
- ASP.NET MVC and Identity 2.0: Understanding the Basics by John Atten.
- Account Confirmation and Password Recovery with ASP.NET Identity
- Introduction to ASP.NET Identity

- Announcing RTM of ASP.NET Identity 2.0.0 by Pranav Rastogi.
- ASP.NET Identity 2.0: Setting Up Account Validation and Two-Factor Authorization by John Atten.

ASP.NET Identity Extensibility

2/20/2020 • 2 minutes to read • Edit Online

Extensibility

- Overview of Custom Storage Providers for ASP.NET Identity
- Implementing a Custom MySQL ASP.NET Identity Storage Provider
- Change Primary Key for Users in ASP.NET Identity

Overview of Custom Storage Providers for ASP.NET Identity

2/20/2020 • 16 minutes to read • Edit Online

by Tom FitzMacken

ASP.NET Identity is an extensible system which enables you to create your own storage provider and plug it into your application without re-working the application. This topic describes how to create a customized storage provider for ASP.NET Identity. It covers the important concepts for creating your own storage provider, but it is not step-by-step walkthrough of implementing a custom storage provider.

For an example of implementing a custom storage provider, see Implementing a Custom MySQL ASP.NET Identity Storage Provider.

This topic was updated for ASP.NET Identity 2.0.

Software versions used in the tutorial

- Visual Studio 2013 with Update 2
- ASP.NET Identity 2

Introduction

By default, the ASP.NET Identity system stores user information in a SQL Server database, and uses Entity Framework Code First to create the database. For many applications, this approach works well. However, you may prefer to use a different type of persistence mechanism, such as Azure Table Storage, or you may already have database tables with a very different structure than the default implementation. In either case, you can write a customized provider for your storage mechanism and plug that provider into your application.

ASP.NET Identity is included by default in many of the Visual Studio 2013 templates. You can get updates to ASP.NET Identity through Microsoft AspNet Identity EntityFramework NuGet package.

This topic includes the following sections:

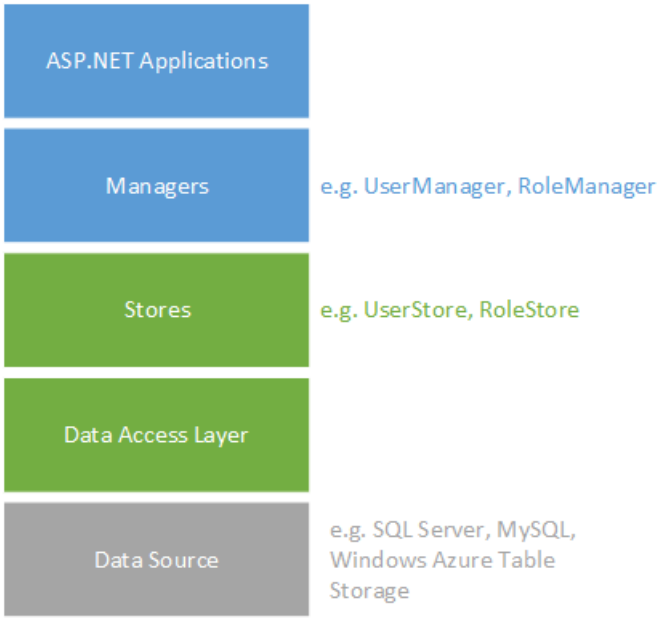
- Understand the architecture
- Understand the data that is stored
- Create the data access layer
- Customize the user class
- Customize the user store
- Customize the role class
- Customize the role store
- Reconfigure application to use new storage provider
- Other implementations of custom storage providers

Understand the architecture

ASP.NET Identity consists of classes called managers and stores. Managers are high-level classes which an application developer uses to perform operations, such as creating a user, in the ASP.NET Identity system. Stores are lower-level classes that specify how entities, such as users and roles, are persisted. Stores are closely coupled

with the persistence mechanism, but managers are decoupled from stores which means you can replace the persistence mechanism without disrupting the entire application.

The following diagram shows how your web application interacts with the managers, and stores interact with the data access layer.



To create a customized storage provider for ASP.NET Identity, you have to create the data source, the data access layer, and the store classes that interact with this data access layer. You can continue using the same manager APIs to perform data operations on the user but now that data is saved to a different storage system.

You do not need to customize the manager classes because when creating a new instance of UserManager or RoleManager you provide the type of the user class and pass an instance of the store class as an argument. This approach enables you to plug your customized classes into the existing structure. You will see how to instantiate UserManager and RoleManager with your customized store classes in the section Reconfigure application to use new storage provider.

Understand the data that is stored

To implement a custom storage provider, you must understand the types of data used with ASP.NET Identity, and decide which features are relevant to your application.

DATA	DESCRIPTION
Users	Registered users of your web site. Includes the user Id and user name. Might include a hashed password if users log in with credentials that are specific to your site (rather than using credentials from an external site like Facebook), and security stamp to indicate whether anything has changed in the user credentials. Might also include email address, phone number, whether two factor authentication is enabled, the current number of failed logins, and whether an account has been locked.
User Claims	A set of statements (or claims) about the user that represent the user's identity. Can enable greater expression of the user's identity than can be achieved through roles.
User Logins	Information about the external authentication provider (like Facebook) to use when logging in a user.

DATA	DESCRIPTION
Roles	Authorization groups for your site. Includes the role Id and role name (like "Admin" or "Employee").

Create the data access layer

This topic assumes you are familiar with the persistence mechanism that you are going to use and how to create entities for that mechanism. This topic does not provide details about how to create the repositories or data access classes; instead, it provides some suggestions about the design decisions you need to make when working with ASP.NET Identity.

You have a lot of freedom when designing the repositories for a customized store provider. You only need to create repositories for features that you intend to use in your application. For example, if you are not using roles in your application, you do not need to create storage for roles or user roles. Your technology and existing infrastructure may require a structure that is very different from the default implementation of ASP.NET Identity. In your data access layer, you provide the logic to work with the structure of your repositories.

For a MySQL implementation of data repositories for ASP.NET Identity 2.0, see MySQLIdentity.sql.

In the data access layer, you provide the logic to save the data from ASP.NET Identity to your data source. The data access layer for your customized storage provider might include the following classes to store user and role information.

CLASS	DESCRIPTION	EXAMPLE
Context	Encapsulates the information to connect to your persistence mechanism and execute queries. This class is central to your data access layer. The other data classes will require an instance of this class to perform their operations. You will also initialize your store classes with an instance of this class.	MySQLDatabase
User Storage	Stores and retrieves user information (such as user name and password hash).	UserTable (MySQL)
Role Storage	Stores and retrieves role information (such as the role name).	RoleTable (MySQL)
UserClaims Storage	Stores and retrieves user claim information (such as the claim type and value).	UserClaimsTable (MySQL)
UserLogins Storage	Stores and retrieves user login information (such as an external authentication provider).	UserLoginsTable (MySQL)
UserRole Storage	Stores and retrieves which roles a user is assigned to.	UserRoleTable (MySQL)

Again, you only need to implement the classes that you intend to use in your application.

In the data access classes, you provide code to perform data operations for your particular persistence

mechanism. For example, within the MySQL implementation, the UserTable class contains a method to insert a new record into the Users database table. The variable named `_database` is an instance of the MySQLDatabase class.

```
public int Insert(TUser user)
{
    string commandText = @"Insert into Users (UserName, Id, PasswordHash,
SecurityStamp,Email,EmailConfirmed,PhoneNumber,PhoneNumberConfirmed,
AccessFailedCount,LockoutEnabled,LockoutEndDateUtc,TwoFactorEnabled)
        values (@name, @id, @pwdHash,
@SecStamp,@email,@emailconfirmed,@phonenumber,@phonenumberconfirmed,@accesscount,@lockoutenabled,@lockoutendd
ate,@twofactorenabled)";
    Dictionary<string, object> parameters = new Dictionary<string, object>();
    parameters.Add("@name", user.UserName);
    parameters.Add("@id", user.Id);
    parameters.Add("@pwdHash", user.PasswordHash);
    parameters.Add("@SecStamp", user.SecurityStamp);
    parameters.Add("@email", user.Email);
    parameters.Add("@emailconfirmed", user.EmailConfirmed);
    parameters.Add("@phonenumber", user.PhoneNumber);
    parameters.Add("@phonenumberconfirmed", user.PhoneNumberConfirmed);
    parameters.Add("@accesscount", user.AccessFailedCount);
    parameters.Add("@lockoutenabled", user.LockoutEnabled);
    parameters.Add("@lockoutenddate", user.LockoutEndDateUtc);
    parameters.Add("@twofactorenabled", user.TwoFactorEnabled);

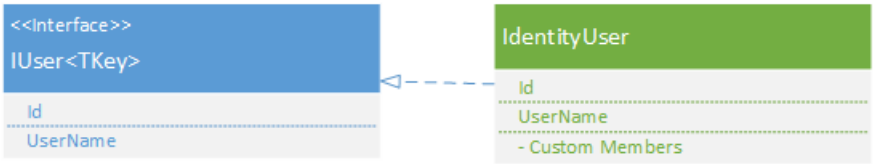
    return _database.Execute(commandText, parameters);
}
```

After creating your data access classes, you must create store classes that call the specific methods in the data access layer.

Customize the user class

When implementing your own storage provider, you must create a user class which is equivalent to the IdentityUser class in the Microsoft.ASP.NET.Identity.EntityFramework namespace:

The following diagram shows the IdentityUser class that you must create and the interface to implement in this class.



The IUser<TKey> interface defines the properties that the UserManager attempts to call when performing requested operations. The interface contains two properties - Id and UserName. The IUser<TKey> interface enables you to specify the type of the key for the user through the generic **TKey** parameter. The type of the Id property matches the value of the TKey parameter.

The Identity framework also provides the IUser interface (without the generic parameter) when you want to use a string value for the key.

The IdentityUser class implements IUser and contains any additional properties or constructors for users on your web site. The following example shows an IdentityUser class that uses an integer for the key. The Id field is set to **int** to match the value of the generic parameter.

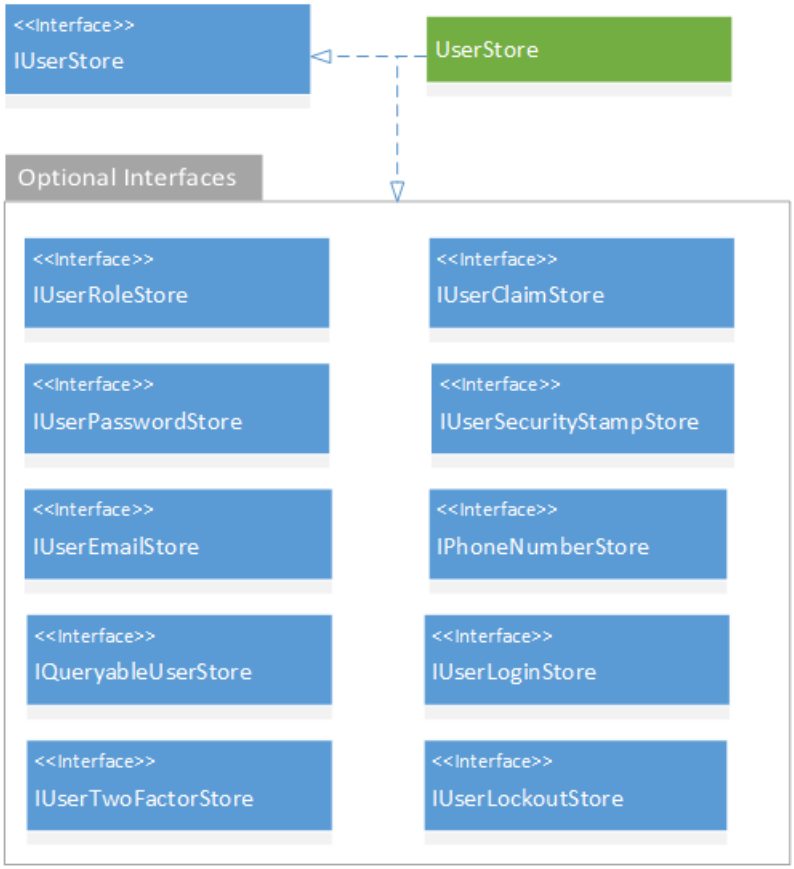
```
public class IdentityUser : IUser<int>
{
    public IdentityUser() { ... }
    public IdentityUser(string userName) { ... }
    public int Id { get; set; }
    public string UserName { get; set; }
    // can also define optional properties such as:
    //    PasswordHash
    //    SecurityStamp
    //    Claims
    //    Logins
    //    Roles
}
```

For a complete implementation, see IdentityUser (MySQL).

Customize the user store

You also create a UserStore class that provides the methods for all data operations on the user. This class is equivalent to the UserStore<TUser> class in the Microsoft.ASP.NET.Identity.EntityFramework namespace. In your UserStore class, you implement the IUserStore<TUser, TKey> and any of the optional interfaces. You select which optional interfaces to implement based on the functionality you wish to provide in your application.

The following image shows the UserStore class you must create and the relevant interfaces.



The default project template in Visual Studio contains code that assumes many of the optional interfaces have been implemented in the user store. If you are using the default template with a customized user store, you must either implement optional interfaces in your user store or alter the template code to no longer call methods in the interfaces you have not implemented.

The following example shows a simple user store class. The **TUser** generic parameter takes the type of your user class which usually is the IdentityUser class you defined. The **TKey** generic parameter takes the type of your user key.

```
public class UserStore : IUserStore<IdentityUser, int>
{
    public UserStore() { ... }
    public UserStore(ExampleStorage database) { ... }
    public Task CreateAsync(IdentityUser user) { ... }
    public Task DeleteAsync(IdentityUser user) { ... }
    public Task<IdentityUser> FindByIdAsync(int userId) { ... }
    public Task<IdentityUser> FindByNameAsync(string userName) { ... }
    public Task UpdateAsync(IdentityUser user) { ... }
    public void Dispose() { ... }
}
```

In this example, the constructor that takes a parameter named *database* of type ExampleDatabase is only an illustration of how to pass in your data access class. For example, in the MySQL implementation, this constructor takes a parameter of type MySQLDatabase.

Within your UserStore class, you use the data access classes that you created to perform operations. For example, in the MySQL implementation, the UserStore class has the CreateAsync method which uses an instance of UserTable to insert a new record. The **Insert** method on the **userTable** object is the same method that was shown in the previous section.

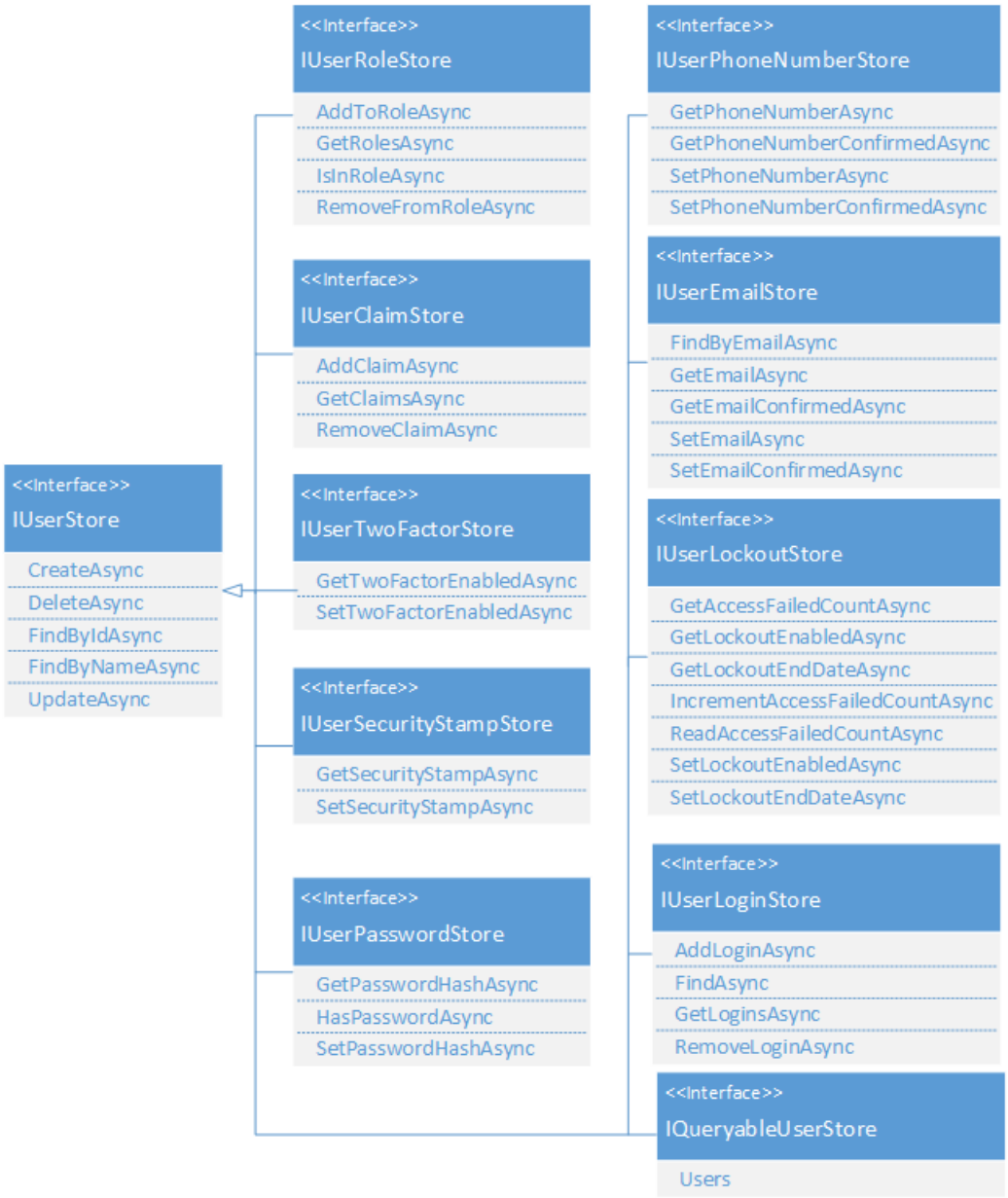
```
public Task CreateAsync(IdentityUser user)
{
    if (user == null) {
        throw new ArgumentNullException("user");
    }

    userTable.Insert(user);

    return Task.FromResult<object>(null);
}
```

Interfaces to implement when customizing user store

The next image shows more details about the functionality defined in each interface. All of the optional interfaces inherit from IUserStore.



- **IUserStore**
  The IUserStore<TUser, TKey> interface is the only interface you must implement in your user store. It defines methods for creating, updating, deleting, and retrieving users.

- **IUserClaimStore**
  The IUserClaimStore<TUser, TKey> interface defines the methods you must implement in your user store to enable user claims. It contains methods or adding, removing and retrieving user claims.

- **IUserLoginStore**
  The IUserLoginStore<TUser, TKey> defines the methods you must implement in your user store to enable external authentication providers. It contains methods for adding, removing and retrieving user logins, and a method for retrieving a user based on the login information.

- **IUserRoleStore**
  The IUserRoleStore<TKey, TUser> interface defines the methods you must implement in your user store to map a user to a role. It contains methods to add, remove, and retrieve a user's roles, and a method to check if a user is assigned to a role.

- **IUserPasswordStore**
  The IUserPasswordStore<TUser, TKey> interface defines the methods you must implement in your user store to persist hashed passwords. It contains methods for getting and setting the hashed password, and a method that indicates whether the user has set a password.

- **IUserSecurityStampStore**
  The IUserSecurityStampStore<TUser, TKey> interface defines the methods you must implement in your user store to use a security stamp for indicating whether the user's account information has changed. This stamp is updated when a user changes the password, or adds or removes logins. It contains methods for getting and setting the security stamp.

- **IUserTwoFactorStore**
  The IUserTwoFactorStore<TUser, TKey> interface defines the methods you must implement to implement two factor authentication. It contains methods for getting and setting whether two factor authentication is enabled for a user.

- **IUserPhoneNumberStore**
  The IUserPhoneNumberStore<TUser, TKey> interface defines the methods you must implement to store user phone numbers. It contains methods for getting and setting the phone number and whether the phone number is confirmed.

- **IUserEmailStore**
  The IUserEmailStore<TUser, TKey> interface defines the methods you must implement to store user email addresses. It contains methods for getting and setting the email address and whether the email is confirmed.

- **IUserLockoutStore**
  The IUserLockoutStore<TUser, TKey> interface defines the methods you must implement to store information about locking an account. It contains methods for getting the current number of failed access attempts, getting and setting whether the account can be locked, getting and setting the lock out end date, incrementing the number of failed attempts, and resetting the number of failed attempts.

- **IQueryableUserStore**
  The IQueryableUserStore<TUser, TKey> interface defines the members you must implement to provide a queryable user store. It contains a property that holds the queryable users.

  You implement the interfaces that are needed in your application; such as, the IUserClaimStore, IUserLoginStore, IUserRoleStore, IUserPasswordStore, and IUserSecurityStampStore interfaces as shown below.

```
public class UserStore : IUserStore<IdentityUser, int>,
                         IUserClaimStore<IdentityUser, int>,
                         IUserLoginStore<IdentityUser, int>,
                         IUserRoleStore<IdentityUser, int>,
                         IUserPasswordStore<IdentityUser, int>,
                         IUserSecurityStampStore<IdentityUser, int>
{
    // interface implementations not shown
}
```

For a complete implementation (including all of interfaces), see UserStore (MySQL).

IdentityUserClaim, IdentityUserLogin, and IdentityUserRole

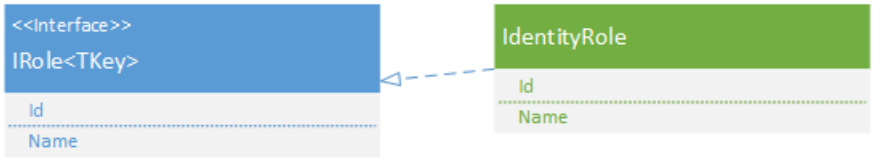
The Microsoft.AspNet.Identity.EntityFramework namespace contains implementations of the IdentityUserClaim, IdentityUserLogin, and IdentityUserRole classes. If you are using these features, you may want to create your own versions of these classes and define the properties for your application. However, sometimes it is more efficient to not load these entities into memory when performing basic operations (such as adding or removing a user's claim). Instead, the backend store classes can execute these operations directly on the data source. For example, the UserStore.GetClaimsAsync() method can call the userClaimTable.FindByUserId(user.Id) method to execute a query on that table directly and return a list of claims.

```
public Task<IList<Claim>> GetClaimsAsync(IdentityUser user)
{
    ClaimsIdentity identity = userClaimsTable.FindByUserId(user.Id);
    return Task.FromResult<IList<Claim>>(identity.Claims.ToList());
}
```

Customize the role class

When implementing your own storage provider, you must create a role class which is equivalent to the IdentityRole class in the Microsoft.ASP.NET.Identity.EntityFramework namespace:

The following diagram shows the IdentityRole class that you must create and the interface to implement in this class.



The IRole<TKey> interface defines the properties that the RoleManager attempts to call when performing requested operations. The interface contains two properties - Id and Name. The IRole<TKey> interface enables you to specify the type of the key for the role through the generic **TKey** parameter. The type of the Id property matches the value of the TKey parameter.

The Identity framework also provides the IRole interface (without the generic parameter) when you want to use a string value for the key.

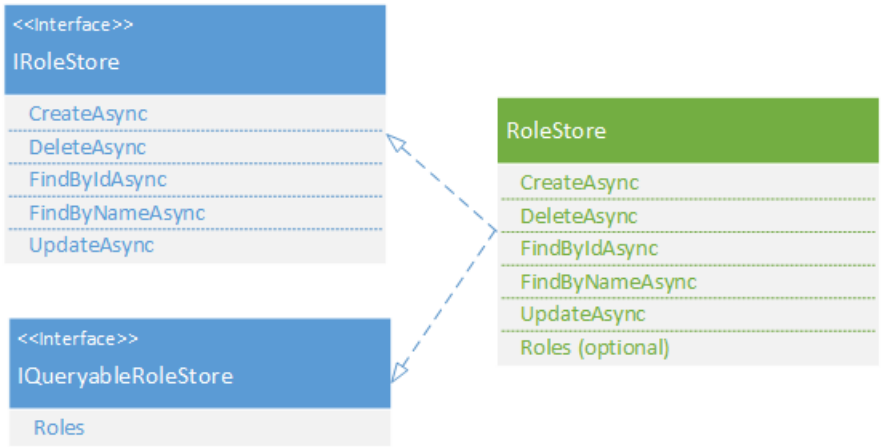
The following example shows an IdentityRole class that uses an integer for the key. The Id field is set to int to match the value of the generic parameter.

```
public class IdentityRole : IRole<int>
{
    public IdentityRole() { ... }
    public IdentityRole(string roleName) { ... }
    public int Id { get; set; }
    public string Name { get; set; }
}
```

For a complete implementation, see IdentityRole (MySQL).

Customize the role store

You also create a RoleStore class that provides the methods for all data operations on roles. This class is equivalent to the RoleStore<TRole> class in the Microsoft.ASP.NET.Identity.EntityFramework namespace. In your RoleStore class, you implement the IRoleStore<TRole, TKey> and optionally the IQueryableRoleStore<TRole, TKey> interface.



The following example shows a role store class. The TRole generic parameter takes the type of your role class which usually is the IdentityRole class you defined. The TKey generic parameter takes the type of your role key.

```
public class RoleStore : IRoleStore<IdentityRole, int>
{
    public RoleStore() { ... }
    public RoleStore(ExampleStorage database) { ... }
    public Task CreateAsync(IdentityRole role) { ... }
    public Task DeleteAsync(IdentityRole role) { ... }
    public Task<IdentityRole> FindByIdAsync(int roleId) { ... }
    public Task<IdentityRole> FindByNameAsync(string roleName) { ... }
    public Task UpdateAsync(IdentityRole role) { ... }
    public void Dispose() { ... }
}
```

- **IRoleStore<TRole>**
  The IRoleStore interface defines the methods to implement in your role store class. It contains methods for creating, updating, deleting and retrieving roles.

- **RoleStore<TRole>**
  To customize RoleStore, create a class that implements the IRoleStore interface. You only have to implement this class if want to use roles on your system. The constructor that takes a parameter named *database* of type ExampleDatabase is only an illustration of how to pass in your data access class. For example, in the MySQL implementation, this constructor takes a parameter of type MySQLDatabase.

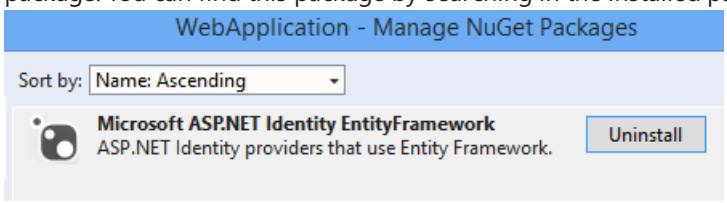
  For a complete implementation, see RoleStore (MySQL) .

Reconfigure application to use new storage provider

You have implemented your new storage provider. Now, you must configure your application to use this storage provider. If the default storage provider was included in your project, you must remove the default provider and replace it with your provider.

Replace default storage provider in MVC project

1. In the **Manage NuGet Packages** window, uninstall the **Microsoft ASP.NET Identity EntityFramework** package. You can find this package by searching in the Installed packages for Identity.EntityFramework.

   You will be asked if you also want to uninstall Entity Framework. If you do not need it in other parts of your application, you can uninstall it.

2. In the IdentityModels.cs file in the Models folder, delete or comment out the **ApplicationUser** and

ApplicationDbContext classes. In an MVC application, you can delete the entire IdentityModels.cs file. In a Web Forms application, delete the two classes but make sure you keep the helper class that is also located in the IdentityModels.cs file.

3. If your storage provider resides in a separate project, add a reference to it in your web application.

4. Replace all references to `using Microsoft.AspNet.Identity.EntityFramework;` with a using statement for the namespace of your storage provider.

5. In the **Startup.Auth.cs** class, change the **ConfigureAuth** method to use a single instance of the appropriate context.

```
public void ConfigureAuth(IAppBuilder app)
{
    app.CreatePerOwinContext(ExampleStorageContext.Create);
    app.CreatePerOwinContext(ApplicationUserManager.Create);
    ...
```

6. In the App_Start folder, open **IdentityConfig.cs**. In the ApplicationUserManager class, change the **Create** method to return a user manager that uses your customized user store.

```
public static ApplicationUserManager Create(IdentityFactoryOptions<ApplicationUserManager> options,
IOwinContext context)
{
    var manager = new ApplicationUserManager(new UserStore(context.Get<ExampleStorageContext>()));
    ...
}
```

7. Replace all references to **ApplicationUser** with **IdentityUser**.

8. The default project includes some members in user class which are not defined in the IUser interface; such as Email, PasswordHash, and GenerateUserIdentityAsync. If your user class does not have these members, you must either implement them or change the code that uses these members.

9. If you have created any instances of RoleManager, change that code to use your new RoleStore class.

```
var roleManager = new RoleManager<IdentityRole>(new RoleStore(context.Get<ExampleStorageContext>()));
```

10. The default project is designed for a user class that has a string value for the key. If your user class has a different type for the key (such as an integer), you must change the project to work with your type. See Change Primary Key for Users in ASP.NET Identity.

11. If needed, add the connection string to the Web.config file.

Other resources

- Blog: Implementing ASP.NET Identity
- Tutorial and GIT code: Simple.Data Asp.Net Identity Provider
- Tutorial:Setting up the basic Identity accounts, and pointing them at an external DB. By @xivSolutions.
- Tutorial: Implementing a Custom MySQL ASP.NET Identity Storage Provider
- CodeFluent Entities by SoftFluent
- Azure Table Storage by James Randall.
- Azure Table Storage: AspNet.Identity.TableStorage by @stuartleeks.
- CouchDB / Cloudant by Daniel Wertheim.
- Elastic Search: Elastic Identity by Bombsquad AB.

- MongoDB by Jonathan Sheely Jonathan Sheely.
- NHibernate.AspNet.Identity by Antônio Milesi Bastos.
- RavenDB by @tourismgeek.
- RavenDB.AspNet.Identity by ILMServices.
- Redis: Redis.AspNet.Identity
- T4 Templates to generate EF code for a "database first" user store: AspNet.Identity.EntityFramework

Implementing a Custom MySQL ASP.NET Identity Storage Provider

2/20/2020 • 6 minutes to read • Edit Online

by Raquel Soares De Almeida, Suhas Joshi, Tom FitzMacken

ASP.NET Identity is an extensible system which enables you to create your own storage provider and plug it into your application without re-working the application. This topic describes how to create a MySQL storage provider for ASP.NET Identity. For an overview of creating custom storage providers, see Overview of Custom Storage Providers for ASP.NET Identity.

To complete this tutorial, you must have Visual Studio 2013 with Update 2.

This tutorial will:

- Show how to create a MySQL database instance on Azure.
- Show how to use a MySQL client tool (MySQL Workbench) to create tables and manage your remote database on Azure.
- Show how to replace the default ASP.NET Identity storage implementation with our custom implementation on a MVC application project.

This tutorial was originally written by Raquel Soares De Almeida and Rick Anderson (@RickAndMSFT). The sample project was updated for Identity 2.0 by Suhas Joshi. The topic was updated for Identity 2.0 by Tom FitzMacken.

Download completed project

At the end of this tutorial, you will have an MVC application project with ASP.NET Identity working with a MySQL database hosted on Azure.

You can download the completed MySQL storage provider at AspNet.Identity.MySQL (GitHub).

The steps you will perform

In this tutorial you will:

1. Create a MySQL database on Azure
2. Create the ASP.NET Identity tables in MySQL
3. Create an MVC application and configure it to use the MySQL provider
4. Run the app

This topic does not cover the architecture of ASP.NET Identity and the decisions you must make when implementing a customer storage provider. For that information, see Overview of Custom Storage Providers for ASP.NET Identity.

Review MySQL storage provider classes

Before jumping into the steps to create the MySQL storage provider, let's look at the classes that make up the storage provider. You will need classes that manage the database operations and classes that are called from the application to manage users and roles.

Storage classes

- IdentityUser - contains properties for the user.
- UserStore - contains operations for adding, updating or retrieving users.
- IdentityRole - contains properties for roles.
- RoleStore - contains operations for adding, deleting, updating and retrieving roles.

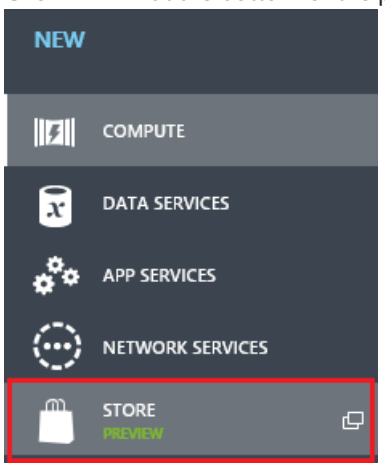
Data access layer classes

For this example, the data access layer classes contain SQL statements for working with the tables; however, in your code you might want to use object-relational mapping (ORM) such as Entity Framework or NHibernate. In particular, your application may experience poor performance without an ORM that includes lazy loading and object caching. For more information, see ASP.NET Identity 2.0 without Entity Framework?

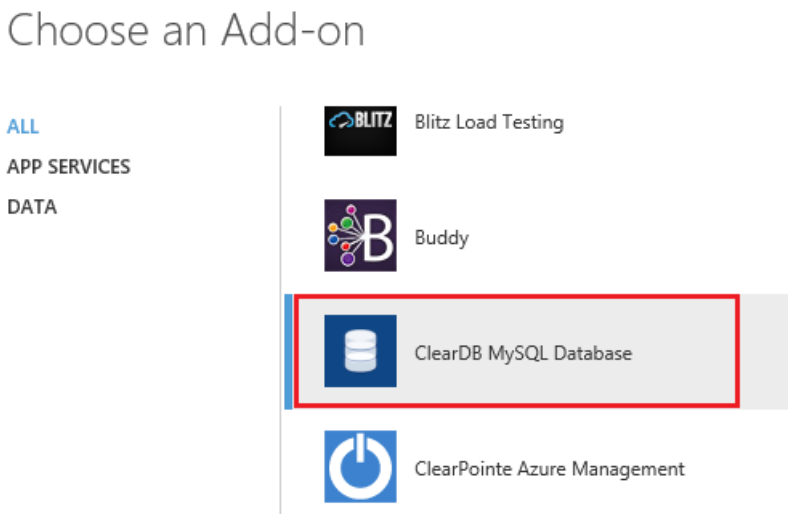
- MySQLDatabase - contains the MySQL database connection and methods for performing database operations. UserStore and RoleStore are both instantiated with an instance of this class.
- RoleTable - contains database operations for the table that stores roles.
- UserClaimsTable - contains database operations for the table that stores user claims.
- UserLoginsTable - contains database operations for the table that stores user login information.
- UserRoleTable - contains database operations for the table that stores which users are assigned to which roles.
- UserTable - contains database operations for the table that stores users.

Create a MySQL database instance on Azure

1. Log in to the Azure Portal.
2. Click **+NEW** at the bottom of the page, and then select **STORE**.

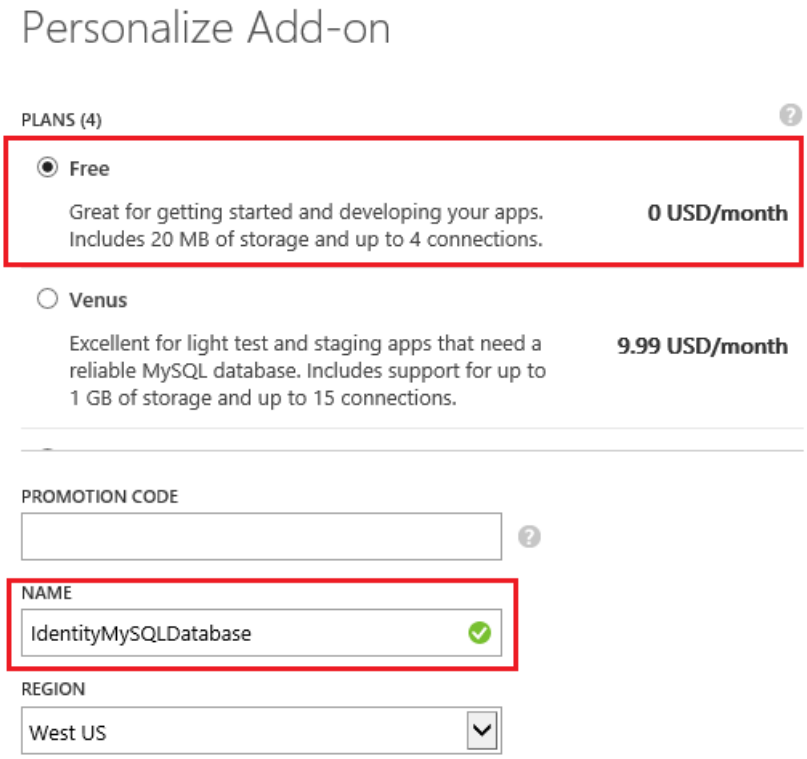


3. In the **Choose and Add-on** wizard, select **ClearDB MySQL Database** and click on the next arrow at the bottom right of the dialog.

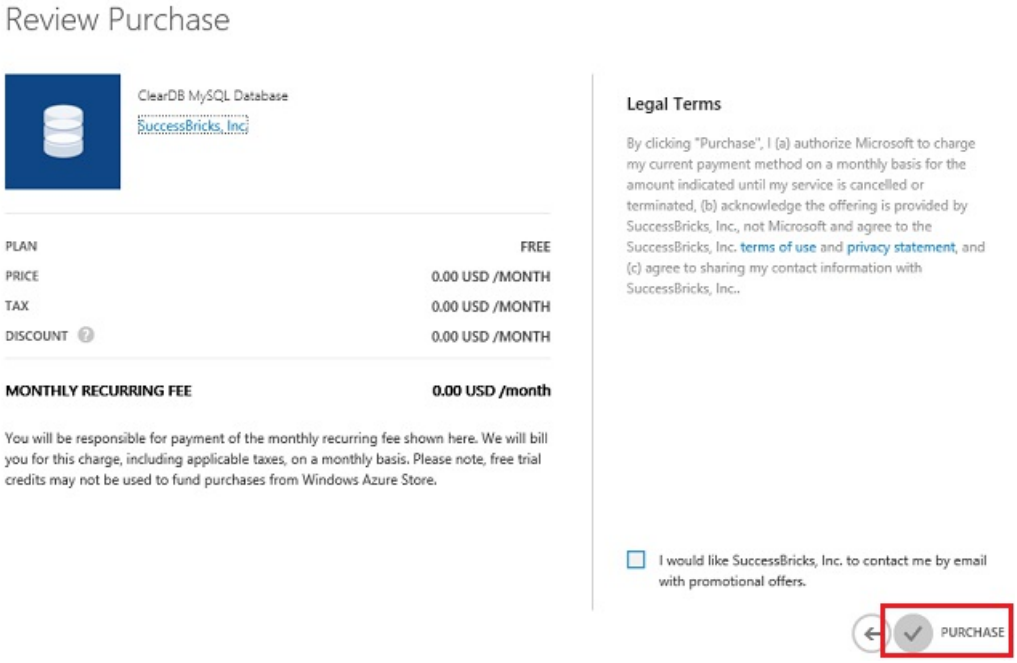


4. Keep the default **Free** plan and change the **Name** to **IdentityMySQLDatabase**. Select the region nearest you

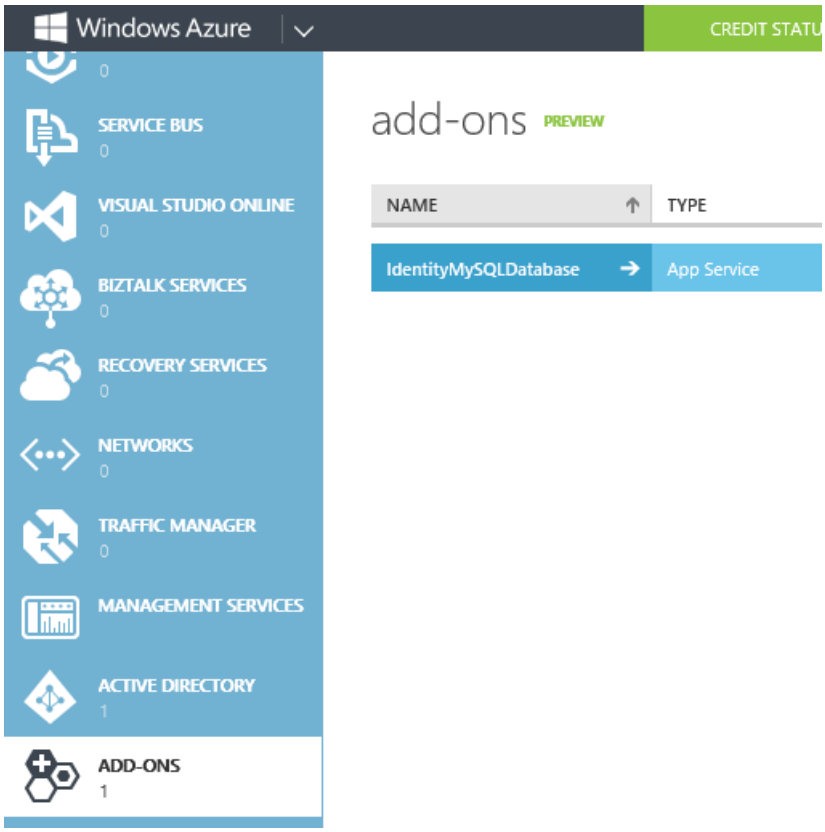
and then click the next arrow.



5. Click the checkmark to complete the database creation.

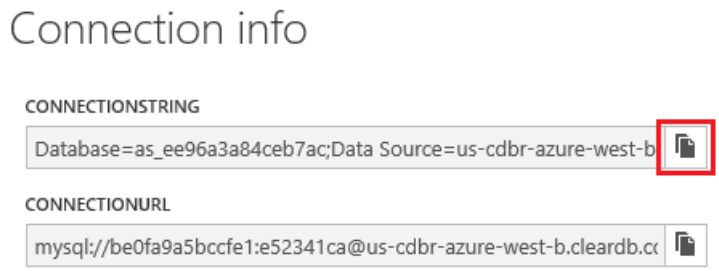


6. After your database has been created, you can manage it from the **ADD-ONS** tab in the management portal.



7. You can get the database connection information by clicking on **CONNECTION INFO** at the bottom of the page.

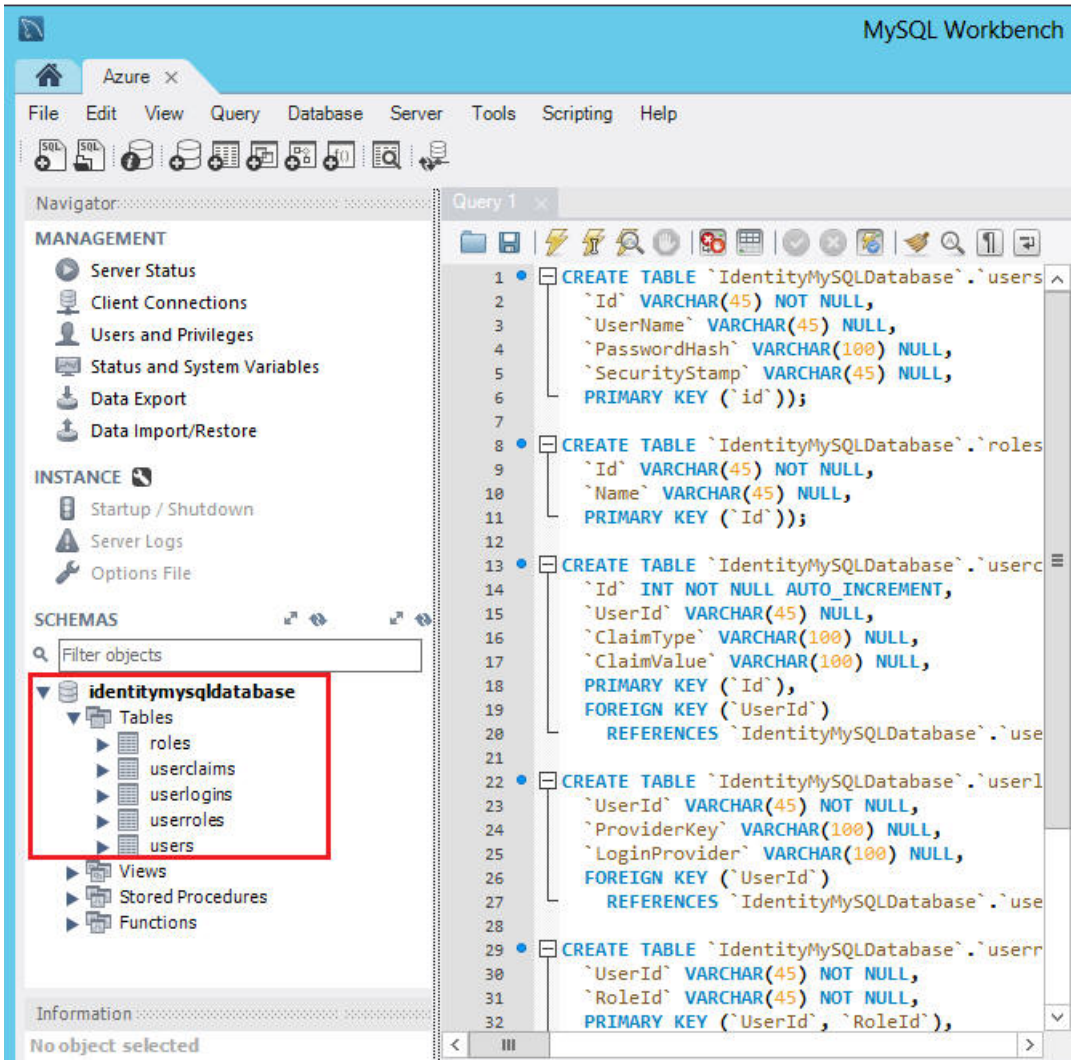
8. Copy the connection string by clicking on the copy button and save it so you can use later in your MVC application.



Create the ASP.NET Identity tables in a MySQL database

Install MySQL Workbench tool to connect and manage MySQL database

1. Install the **MySQL Workbench** tool from the MySQL downloads page
2. Launch the app and add click on the **MySQLConnections +** button to add a new connection. Use the connection string data you copied from the Azure MySQL database you created earlier in this tutorial.
3. After establishing the connection, open a new **Query** tab; paste the commands from MySQLIdentity.sql into the query and execute it in order to create the database tables.
4. You now have all the ASP.NET Identity necessary tables created on a MySQL database hosted on Azure as shown below.



Create an MVC application project from template and configure it to use MySQL provider

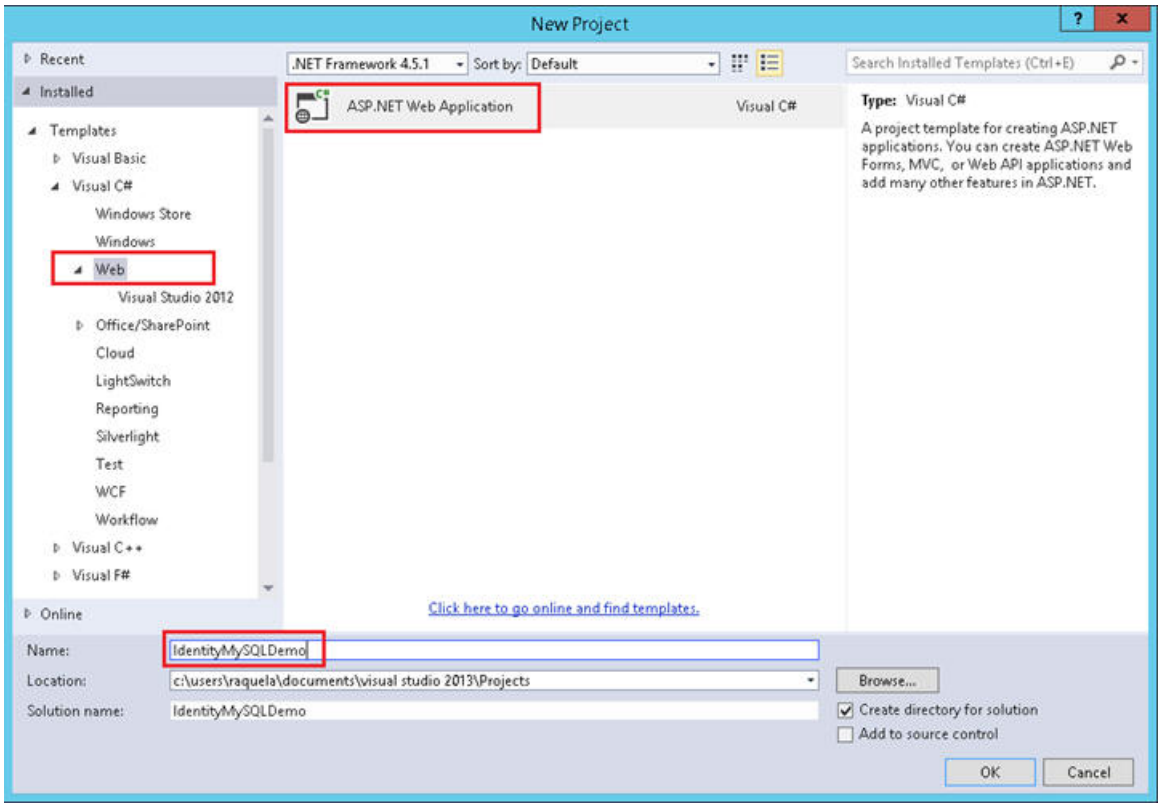
If needed, install either Visual Studio Express 2013 for Web or Visual Studio 2013 with Update 2.

Download the ASP.NET.Identity.MySQL project from GitHub

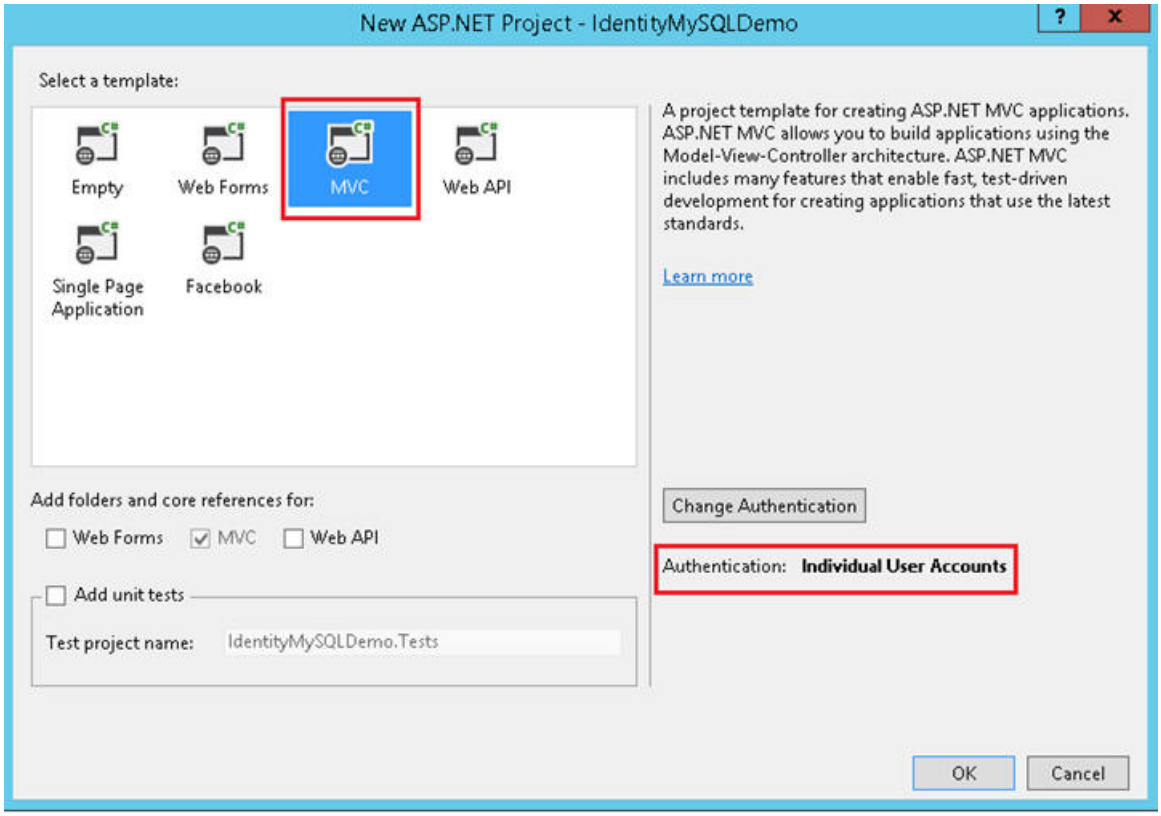
1. Browse to the repository URL at AspNet.Identity.MySQL (GitHub).
2. Download the source code.
3. Extract the .zip file into a local folder.
4. Open the AspNet.Identity.MySQL solution and build it.

Create a new MVC application project from template

1. Right click the **AspNet.Identity.MySQL** solution and **Add**, **New Project**
2. In the **Add New Project** Dialog select **Visual C#** on the left, then **Web** and then select **ASP.NET Web Application**. Name your project **IdentityMySQLDemo**; and then click OK.



3. In the **New ASP.NET Project** dialog, select the MVC template with the default options (that includes **Individual User Accounts** as authentication method) and click **OK**.



4. In Solution Explorer, right-click your IdentityMySQLDemo project and select **Manage NuGet Packages**. In the search text box dialog, type **Identity.EntityFramework**. Select this package in the list of results and click **Uninstall**. You will be prompted to uninstall the dependency package EntityFramework. Click on Yes as we will no longer this package on this application.

5. Right click the IdentityMySQLDemo project, select **Add**, **Reference, Solution, Projects;** select the AspNet.Identity.MySQL project and click **OK**.

6. In the IdentityMySQLDemo project, replace all references to

`using Microsoft.AspNet.Identity.EntityFramework;`

with

`using AspNet.Identity.MySQL;`

7. In IdentityModels.cs, set **ApplicationDbContext** to derive from **MySqlDatabase** and include a constructor that take a single parameter with the connection name.

```
public class ApplicationDbContext : MySQLDatabase
{
    public ApplicationDbContext(string connectionName)
        : base(connectionName)
    {
    }

    public static ApplicationDbContext Create()
    {
        return new ApplicationDbContext("DefaultConnection");
    }
}
```

8. Open the IdentityConfig.cs file. In the **ApplicationUserManager.Create** method, replace instantiating UserManager with the following code:

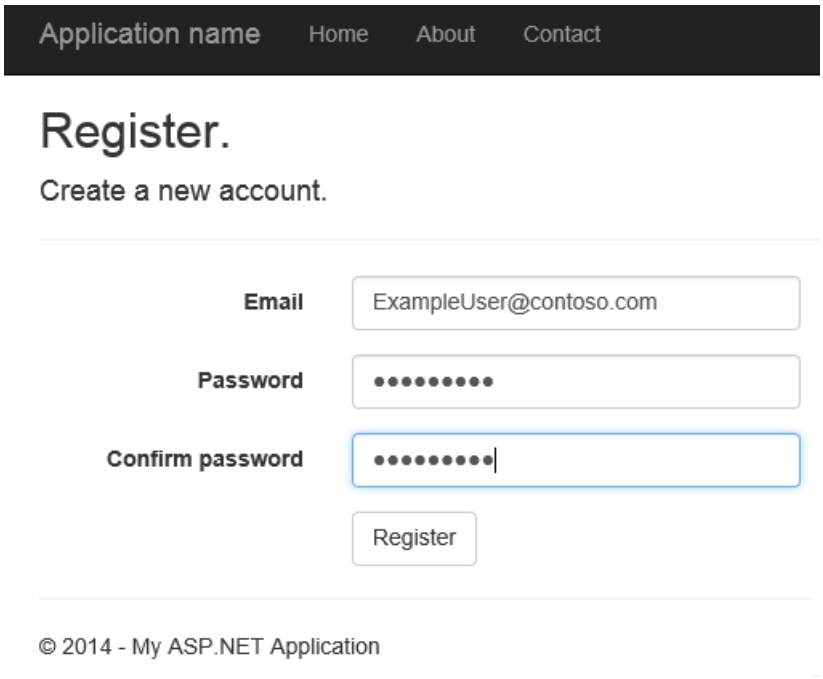
```
var manager = new ApplicationUserManager(
    new UserStore<ApplicationUser>(
    context.Get<ApplicationDbContext>() as MySQLDatabase));
```

9. Open the web.config file and replace the DefaultConnection string with this entry replacing the highlighted values with the connection string of the MySQL database you created on previous steps:

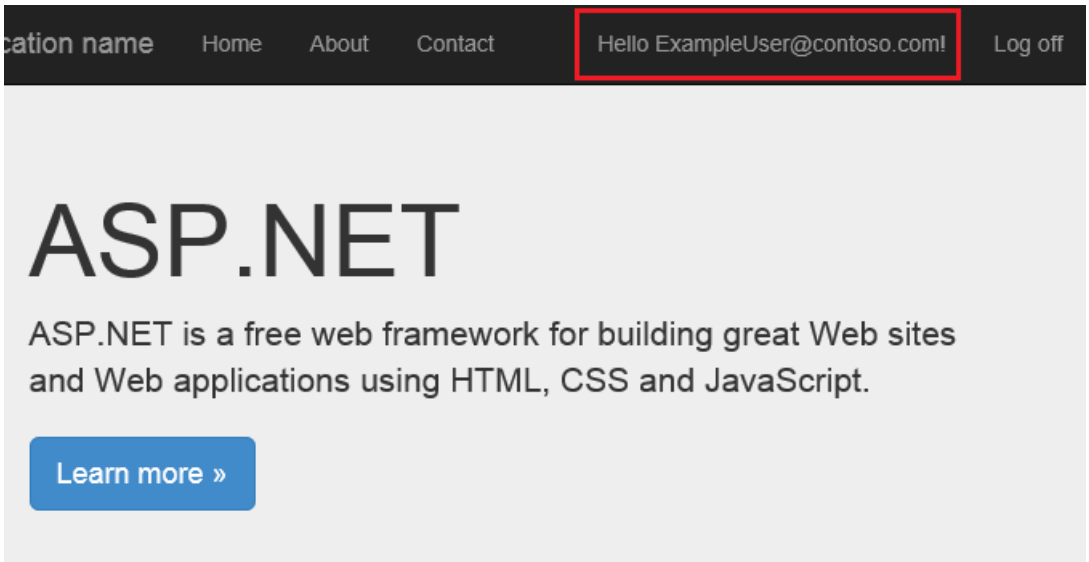
```
<add name="DefaultConnection" connectionString="Database=IdentityMySQLDatabase;
Data Source=<DataSource>;User Id=<UserID>;Password=<Password>"
providerName="MySql.Data.MySqlClient" />
```

Run the app and connect to the MySQL DB

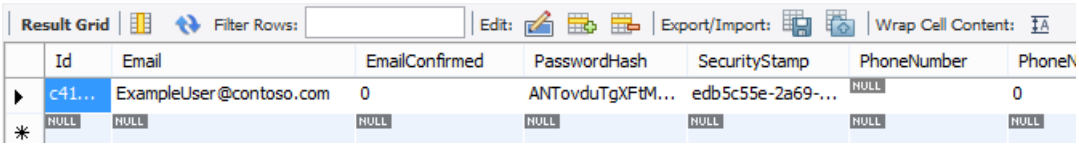
1. Right click the **IdentityMySQLDemo** project and select **Set as Startup Project**
2. Press **Ctrl + F5** to build and run the app.
3. Click on **Register** tab on the top of the page.
4. Enter a new user name and password and then click on **Register**.



5. The new user is now registered and logged in.



6. Go back to the MySQL Workbench tool and inspect the **IdentityMySQLDatabase** table's contents. Inspect the users table for the entries as you register new users.



Id	Email	EmailConfirmed	PasswordHash	SecurityStamp	PhoneNumber	PhoneN
c41...	ExampleUser@contoso.com	0	ANTovduTgXFtM...	edb5c55e-2a69-...	NULL	0
NULL	NULL	NULL	NULL	NULL	NULL	NULL

Next Steps

For more information on how to enable other authentication methods on this app, refer to Create an ASP.NET MVC 5 App with Facebook and Google OAuth2 and OpenID Sign-on.

To learn how to integrate your DB with OAuth and to set up roles to limit users access to your app, see Deploy a Secure ASP.NET MVC 5 app with Membership, OAuth, and SQL Database to Azure.

Change Primary Key for Users in ASP.NET Identity

2/20/2020 • 20 minutes to read • Edit Online

by Tom FitzMacken

In Visual Studio 2013, the default web application uses a string value for the key for user accounts. ASP.NET Identity enables you to change the type of the key to meet your data requirements. For example, you can change the type of the key from a string to an integer.

This topic shows how to start with the default web application and change the user account key to an integer. You can use the same modifications to implement any type of key in your project. It shows how to make these changes in the default web application, but you could apply similar modifications to a customized application. It shows the changes needed when working with MVC or Web Forms.

Software versions used in the tutorial

- Visual Studio 2013 with Update 2 (or later)
- ASP.NET Identity 2.1 or later

To perform the steps in this tutorial, you must have Visual Studio 2013 Update 2 (or later), and a web application created from the ASP.NET Web Application template. The template changed in Update 3. This topic shows how to change the template in Update 2 and Update 3.

This topic contains the following sections:

- Change the type of the key in the Identity user class
- Add customized Identity classes that use the key type
- Change the context class and user manager to use the key type
- Change start-up configuration to use the key type
- For MVC with Update 2, change the AccountController to pass the key type
- For MVC with Update 3, change the AccountController and ManageController to pass the key type
- For Web Forms with Update 2, change Account pages to pass the key type
- For Web Forms with Update 3, change Account pages to pass the key type
- Run application
- Other resources

Change the type of the key in the Identity user class

In your project created from the ASP.NET Web Application template, specify that the ApplicationUser class uses an integer for the key for user accounts. In IdentityModels.cs, change the ApplicationUser class to inherit from IdentityUser that has a type of **int** for the TKey generic parameter. You also pass the names of three customized class which you have not implemented yet.

```
public class ApplicationUser : IdentityUser<int, CustomUserLogin, CustomUserRole,
    CustomUserClaim>
{
    ...
```

You have changed the type of the key, but, by default, the rest of the application still assumes the key is a string.

You must explicitly indicate the type of the key in code that assumes a string.

In the **ApplicationUser** class, change the **GenerateUserIdentityAsync** method to include int, as shown in the highlighted code below. This change is not necessary for Web Forms projects with the Update 3 template.

```
public async Task<ClaimsIdentity> GenerateUserIdentityAsync(
    UserManager<ApplicationUser, int> manager)
{
    // Note the authenticationType must match the one defined in
    // CookieAuthenticationOptions.AuthenticationType
    var userIdentity = await manager.CreateIdentityAsync(
        this, DefaultAuthenticationTypes.ApplicationCookie);
    // Add custom user claims here
    return userIdentity;
}
```

Add customized Identity classes that use the key type

The other Identity classes, such as IdentityUserRole, IdentityUserClaim, IdentityUserLogin, IdentityRole, UserStore, RoleStore, are still set up to use a string key. Create new versions of these classes that specify an integer for the key. You do not need to provide much implementation code in these classes, you are primarily just setting int as the key.

Add the following classes to your IdentityModels.cs file.

```
public class CustomUserRole : IdentityUserRole<int> { }
public class CustomUserClaim : IdentityUserClaim<int> { }
public class CustomUserLogin : IdentityUserLogin<int> { }

public class CustomRole : IdentityRole<int, CustomUserRole>
{
    public CustomRole() { }
    public CustomRole(string name) { Name = name; }
}

public class CustomUserStore : UserStore<ApplicationUser, CustomRole, int,
    CustomUserLogin, CustomUserRole, CustomUserClaim>
{
    public CustomUserStore(ApplicationDbContext context)
        : base(context)
    {
    }
}

public class CustomRoleStore : RoleStore<CustomRole, int, CustomUserRole>
{
    public CustomRoleStore(ApplicationDbContext context)
        : base(context)
    {
    }
}
```

Change the context class and user manager to use the key type

In IdentityModels.cs, change the definition of the **ApplicationDbContext** class to use your new customized classes and an **int** for the key, as shown in the highlighted code.

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser, CustomRole,
    int, CustomUserLogin, CustomUserRole, CustomUserClaim>
{
    ...
```

The ThrowIfV1Schema parameter is no longer valid in the constructor. Change the constructor so it does not pass a ThrowIfV1Schema value.

```
public ApplicationDbContext()
    : base("DefaultConnection")
{
}
```

Open IdentityConfig.cs, and change the **ApplicationUserManger** class to use your new user store class for persisting data and an **int** for the key.

```
public class ApplicationUserManager : UserManager<ApplicationUser, int>
{
    public ApplicationUserManager(IUserStore<ApplicationUser, int> store)
        : base(store)
    {
    }

    public static ApplicationUserManager Create(
        IdentityFactoryOptions<ApplicationUserManager> options, IOwinContext context)
    {
        var manager = new ApplicationUserManager(
            new CustomUserStore(context.Get<ApplicationDbContext>()));
        // Configure validation logic for usernames
        manager.UserValidator = new UserValidator<ApplicationUser, int>(manager)
        {
            AllowOnlyAlphanumericUserNames = false,
            RequireUniqueEmail = true
        };
        // Configure validation logic for passwords
        manager.PasswordValidator = new PasswordValidator
        {
            RequiredLength = 6,
            RequireNonLetterOrDigit = true,
            RequireDigit = true,
            RequireLowercase = true,
            RequireUppercase = true,
        };
        // Register two factor authentication providers. This application uses Phone
        // and Emails as a step of receiving a code for verifying the user
        // You can write your own provider and plug in here.
        manager.RegisterTwoFactorProvider("PhoneCode",
            new PhoneNumberTokenProvider<ApplicationUser, int>
        {
            MessageFormat = "Your security code is: {0}"
        });
        manager.RegisterTwoFactorProvider("EmailCode",
            new EmailTokenProvider<ApplicationUser, int>
        {
            Subject = "Security Code",
            BodyFormat = "Your security code is: {0}"
        });
        manager.EmailService = new EmailService();
        manager.SmsService = new SmsService();
        var dataProtectionProvider = options.DataProtectionProvider;
        if (dataProtectionProvider != null)
        {
            manager.UserTokenProvider =
                new DataProtectorTokenProvider<ApplicationUser, int>(
                    dataProtectionProvider.Create("ASP.NET Identity"));
        }
        return manager;
    }
}
```

In the Update 3 template, you must change the ApplicationSignInManager class.

```
public class ApplicationSignInManager : SignInManager<ApplicationUser, int>
{ ... }
```

Change start-up configuration to use the key type

In Startup.Auth.cs, replace the OnValidateIdentity code, as highlighted below. Notice that the getUserIdCallback definition, parses the string value into an integer.

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
    {
        AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
        LoginPath = new PathString("/Account/Login"),
        Provider = new CookieAuthenticationProvider
        {
            OnValidateIdentity = SecurityStampValidator
                .OnValidateIdentity<ApplicationUserManager, ApplicationUser, int>(
                    validateInterval: TimeSpan.FromMinutes(30),
                    regenerateIdentityCallback: (manager, user) =>
                        user.GenerateUserIdentityAsync(manager),
                    getUserIdCallback:(id)=>(id.GetUserId<int>()))
        }
    });
```

If your project does not recognize the generic implementation of the **GetUserId** method, you may need to update the ASP.NET Identity NuGet package to version 2.1

You have made a lot of changes to the infrastructure classes used by ASP.NET Identity. If you try compiling the project, you will notice a lot of errors. Fortunately, the remaining errors are all similar. The Identity class expects an integer for the key, but the controller (or Web Form) is passing a string value. In each case, you need to convert from a string to and integer by calling **GetUserId<int>**. You can either work through the error list from compilation or follow the changes below.

The remaining changes depend on the type of project you are creating and which update you have installed in Visual Studio. You can go directly to the relevant section through the following links

- For MVC with Update 2, change the AccountController to pass the key type
- For MVC with Update 3, change the AccountController and ManageController to pass the key type
- For Web Forms with Update 2, change Account pages to pass the key type
- For Web Forms with Update 3, change Account pages to pass the key type

For MVC with Update 2, change the AccountController to pass the key type

Open the AccountController.cs file. You need to change the following methods.

ConfirmEmail method

```
public async Task<ActionResult> ConfirmEmail(int userId, string code)
{
    if (userId == default(int) || code == null)
    {
        return View("Error");
    }

    IdentityResult result = await UserManager.ConfirmEmailAsync(userId, code);
    if (result.Succeeded)
    {
        return View("ConfirmEmail");
    }
    else
    {
        AddErrors(result);
        return View();
    }
}
```

Disassociate method

```
public async Task<ActionResult> Disassociate(string loginProvider, string providerKey)
{
    ManageMessageId? message = null;
    IdentityResult result = await UserManager.RemoveLoginAsync(
        User.Identity.GetUserId<int>(),
        new UserLoginInfo(loginProvider, providerKey));
    if (result.Succeeded)
    {
        var user = await UserManager.FindByIdAsync(User.Identity.GetUserId<int>());
        await SignInAsync(user, isPersistent: false);
        message = ManageMessageId.RemoveLoginSuccess;
    }
    else
    {
        message = ManageMessageId.Error;
    }
    return RedirectToAction("Manage", new { Message = message });
}
```

Manage(ManageUserViewModel) method

```
public async Task<ActionResult> Manage(ManageUserViewModel model)
{
    bool hasPassword = HasPassword();
    ViewBag.HasLocalPassword = hasPassword;
    ViewBag.ReturnUrl = Url.Action("Manage");
    if (hasPassword)
    {
        if (ModelState.IsValid)
        {
            IdentityResult result = await UserManager.ChangePasswordAsync(
                User.Identity.GetUserId<int>(),
                model.OldPassword,
                model.NewPassword);
            if (result.Succeeded)
            {
                var user = await UserManager.FindByIdAsync(
                  User.Identity.GetUserId<int>());
                await SignInAsync(user, isPersistent: false);
                return RedirectToAction("Manage", new {
                    Message = ManageMessageId.ChangePasswordSuccess });
            }
            else
            {
                AddErrors(result);
            }
        }
    }
    else
    {
        // User does not have a password so remove any validation errors caused
        // by a missing OldPassword field
        ModelState state = ModelState["OldPassword"];
        if (state != null)
        {
            state.Errors.Clear();
        }

        if (ModelState.IsValid)
        {
            IdentityResult result = await UserManager.AddPasswordAsync(
                User.Identity.GetUserId<int>(), model.NewPassword);
            if (result.Succeeded)
            {
                return RedirectToAction("Manage", new {
                    Message = ManageMessageId.SetPasswordSuccess });
            }
            else
            {
                AddErrors(result);
            }
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

LinkLoginCallback method

```
public async Task<ActionResult> LinkLoginCallback()
{
    var loginInfo = await AuthenticationManager.GetExternalLoginInfoAsync(XsrfKey,
        User.Identity.GetUserId());
    if (loginInfo == null)
    {
        return RedirectToAction("Manage", new { Message = ManageMessageId.Error });
    }
    IdentityResult result = await UserManager.AddLoginAsync(
        User.Identity.GetUserId<int>(), loginInfo.Login);
    if (result.Succeeded)
    {
        return RedirectToAction("Manage");
    }
    return RedirectToAction("Manage", new { Message = ManageMessageId.Error });
}
```

RemoveAccountList method

```
public ActionResult RemoveAccountList()
{
    var linkedAccounts = UserManager.GetLogins(User.Identity.GetUserId<int>());
    ViewBag.ShowRemoveButton = HasPassword() || linkedAccounts.Count > 1;
    return (ActionResult)PartialView("_RemoveAccountPartial", linkedAccounts);
}
```

HasPassword method

```
private bool HasPassword()
{
    var user = UserManager.FindById(User.Identity.GetUserId<int>());
    if (user != null)
    {
        return user.PasswordHash != null;
    }
    return false;
}
```

You can now run the application and register a new user.

For MVC with Update 3, change the AccountController and ManageController to pass the key type

Open the AccountController.cs file. You need to change the following method.

ConfirmEmail method

```
public async Task<ActionResult> ConfirmEmail(int userId, string code)
{
    if (userId == default(int) || code == null)
    {
        return View("Error");
    }

    IdentityResult result = await UserManager.ConfirmEmailAsync(userId, code);
    return View(result.Succeeded ? "ConfirmEmail" : "Error");
}
```

SendCode method

```
public async Task<ActionResult> SendCode(string returnUrl, bool rememberMe)
{
    var userId = await SignInManager.GetVerifiedUserIdAsync();
    if (userId == default(int))
    {
        return View("Error");
    }
    var userFactors = await UserManager.GetValidTwoFactorProvidersAsync(userId);
    var factorOptions = userFactors.Select(purpose => new SelectListItem { Text = purpose, Value = purpose
}).ToList();
    return View(new SendCodeViewModel { Providers = factorOptions, ReturnUrl = returnUrl, RememberMe =
rememberMe });
}
```

Open the ManageController.cs file. You need to change the following methods.

Index method

```
public async Task<ActionResult> Index(ManageMessageId? message)
{
    ViewBag.StatusMessage =
        message == ManageMessageId.ChangePasswordSuccess ? "Your password has been changed."
        : message == ManageMessageId.SetPasswordSuccess ? "Your password has been set."
        : message == ManageMessageId.SetTwoFactorSuccess ? "Your two-factor authentication provider has been
set."
        : message == ManageMessageId.Error ? "An error has occurred."
        : message == ManageMessageId.AddPhoneSuccess ? "Your phone number was added."
        : message == ManageMessageId.RemovePhoneSuccess ? "Your phone number was removed."
        : "";

    var model = new IndexViewModel
    {
        HasPassword = HasPassword(),
        PhoneNumber = await UserManager.GetPhoneNumberAsync(User.Identity.GetUserId<int>()),
        TwoFactor = await UserManager.GetTwoFactorEnabledAsync(User.Identity.GetUserId<int>()),
        Logins = await UserManager.GetLoginsAsync(User.Identity.GetUserId<int>()),
        BrowserRemembered = await AuthenticationManager.TwoFactorBrowserRememberedAsync(
          User.Identity.GetUserId())
    };
    return View(model);
}
```

RemoveLogin methods

```
public ActionResult RemoveLogin()
{
    var linkedAccounts = UserManager.GetLogins((User.Identity.GetUserId<int>()));
    ViewBag.ShowRemoveButton = HasPassword() || linkedAccounts.Count > 1;
    return View(linkedAccounts);
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> RemoveLogin(string loginProvider, string providerKey)
{
    ManageMessageId? message;
    var result = await UserManager.RemoveLoginAsync(User.Identity.GetUserId<int>(),
        new UserLoginInfo(loginProvider, providerKey));
    if (result.Succeeded)
    {
        var user = await UserManager.FindByIdAsync(User.Identity.GetUserId<int>());
        if (user != null)
        {
            await SignInAsync(user, isPersistent: false);
        }
        message = ManageMessageId.RemoveLoginSuccess;
    }
    else
    {
        message = ManageMessageId.Error;
    }
    return RedirectToAction("ManageLogins", new { Message = message });
}
```

AddPhoneNumber method

```
public async Task<ActionResult> AddPhoneNumber(AddPhoneNumberViewModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
    // Generate the token and send it
    var code = await UserManager.GenerateChangePhoneNumberTokenAsync(
        User.Identity.GetUserId<int>(), model.Number);
    if (UserManager.SmsService != null)
    {
        var message = new IdentityMessage
        {
            Destination = model.Number,
            Body = "Your security code is: " + code
        };
        await UserManager.SmsService.SendAsync(message);
    }
    return RedirectToAction("VerifyPhoneNumber", new { PhoneNumber = model.Number });
}
```

EnableTwoFactorAuthentication method

```
public async Task<ActionResult> EnableTwoFactorAuthentication()
{
    await UserManager.SetTwoFactorEnabledAsync(User.Identity.GetUserId<int>(), true);
    var user = await UserManager.FindByIdAsync(User.Identity.GetUserId<int>());
    if (user != null)
    {
        await SignInAsync(user, isPersistent: false);
    }
    return RedirectToAction("Index", "Manage");
}
```

DisableTwoFactorAuthentication method

```
public async Task<ActionResult> DisableTwoFactorAuthentication()
{
    await UserManager.SetTwoFactorEnabledAsync(User.Identity.GetUserId<int>(), false);
    var user = await UserManager.FindByIdAsync(User.Identity.GetUserId<int>());
    if (user != null)
    {
        await SignInAsync(user, isPersistent: false);
    }
    return RedirectToAction("Index", "Manage");
}
```

VerifyPhoneNumber methods

```
public async Task<ActionResult> VerifyPhoneNumber(string phoneNumber)
{
    var code = await UserManager.GenerateChangePhoneNumberTokenAsync(
        User.Identity.GetUserId<int>(), phoneNumber);
    // Send an SMS through the SMS provider to verify the phone number
    return phoneNumber == null ? View("Error") : View(new VerifyPhoneNumberViewModel { PhoneNumber =
phoneNumber });
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> VerifyPhoneNumber(VerifyPhoneNumberViewModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
    var result = await UserManager.ChangePhoneNumberAsync(
        User.Identity.GetUserId<int>(), model.PhoneNumber, model.Code);
    if (result.Succeeded)
    {
        var user = await UserManager.FindByIdAsync(User.Identity.GetUserId<int>());
        if (user != null)
        {
            await SignInAsync(user, isPersistent: false);
        }
        return RedirectToAction("Index", new { Message = ManageMessageId.AddPhoneSuccess });
    }
    // If we got this far, something failed, redisplay form
    ModelState.AddModelError("", "Failed to verify phone");
    return View(model);
}
```

RemovePhoneNumber method

```
public async Task<ActionResult> RemovePhoneNumber()
{
    var result = await UserManager.SetPhoneNumberAsync(User.Identity.GetUserId<int>(), null);
    if (!result.Succeeded)
    {
        return RedirectToAction("Index", new { Message = ManageMessageId.Error });
    }
    var user = await UserManager.FindByIdAsync(User.Identity.GetUserId<int>());
    if (user != null)
    {
        await SignInAsync(user, isPersistent: false);
    }
    return RedirectToAction("Index", new { Message = ManageMessageId.RemovePhoneSuccess });
}
```

ChangePassword method

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> ChangePassword(ChangePasswordViewModel model)
{
    if (!ModelState.IsValid)
    {
        return View(model);
    }
    var result = await UserManager.ChangePasswordAsync(
        User.Identity.GetUserId<int>(), model.OldPassword, model.NewPassword);
    if (result.Succeeded)
    {
        var user = await UserManager.FindByIdAsync(User.Identity.GetUserId<int>());
        if (user != null)
        {
            await SignInAsync(user, isPersistent: false);
        }
        return RedirectToAction("Index", new { Message = ManageMessageId.ChangePasswordSuccess });
    }
    AddErrors(result);
    return View(model);
}
```

SetPassword method

```
public async Task<ActionResult> SetPassword(SetPasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        var result = await UserManager.AddPasswordAsync(User.Identity.GetUserId<int>(), model.NewPassword);
        if (result.Succeeded)
        {
            var user = await UserManager.FindByIdAsync(User.Identity.GetUserId<int>());
            if (user != null)
            {
                await SignInAsync(user, isPersistent: false);
            }
            return RedirectToAction("Index", new { Message = ManageMessageId.SetPasswordSuccess });
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

ManageLogins method

```
public async Task<ActionResult> ManageLogins(ManageMessageId? message)
{
    ViewBag.StatusMessage =
        message == ManageMessageId.RemoveLoginSuccess ? "The external login was removed."
        : message == ManageMessageId.Error ? "An error has occurred."
        : "";
    var user = await UserManager.FindByIdAsync(User.Identity.GetUserId<int>());
    if (user == null)
    {
        return View("Error");
    }
    var userLogins = await UserManager.GetLoginsAsync(User.Identity.GetUserId<int>());
    var otherLogins = AuthenticationManager.GetExternalAuthenticationTypes().Where(auth => userLogins.All(ul
=> auth.AuthenticationType != ul.LoginProvider)).ToList();
    ViewBag.ShowRemoveButton = user.PasswordHash != null || userLogins.Count > 1;
    return View(new ManageLoginsViewModel
    {
        CurrentLogins = userLogins,
        OtherLogins = otherLogins
    });
}
```

LinkLoginCallback method

```
public async Task<ActionResult> LinkLoginCallback()
{
    var loginInfo = await AuthenticationManager.GetExternalLoginInfoAsync(XsrfKey,
User.Identity.GetUserId());
    if (loginInfo == null)
    {
        return RedirectToAction("ManageLogins", new { Message = ManageMessageId.Error });
    }
    var result = await UserManager.AddLoginAsync(User.Identity.GetUserId<int>(),
        loginInfo.Login);
    return result.Succeeded ? RedirectToAction("ManageLogins") :
        RedirectToAction("ManageLogins", new { Message = ManageMessageId.Error });
}
```

HasPassword method

```
private bool HasPassword()
{
    var user = UserManager.FindById(User.Identity.GetUserId<int>());
    if (user != null)
    {
        return user.PasswordHash != null;
    }
    return false;
}
```

HasPhoneNumber method

```
private bool HasPhoneNumber()
{
    var user = UserManager.FindById(User.Identity.GetUserId<int>());
    if (user != null)
    {
        return user.PhoneNumber != null;
    }
    return false;
}
```

You can now run the application and register a new user.

For Web Forms with Update 2, change Account pages to pass the key type

For Web Forms with Update 2, you need to change the following pages.

Confirm.aspx.cx

```
protected void Page_Load(object sender, EventArgs e)
{
    string code = IdentityHelper.GetCodeFromRequest(Request);
    string userId = IdentityHelper.GetUserIdFromRequest(Request);
    if (code != null && userId != null)
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        var result = manager.ConfirmEmail(Int32.Parse(userId), code);
        if (result.Succeeded)
        {
            StatusMessage = "Thank you for confirming your account.";
            return;
        }
    }

    StatusMessage = "An error has occurred";
}
```

RegisterExternalLogin.aspx.cs

```
protected void Page_Load()
{
    // Process the result from an auth provider in the request
    ProviderName = IdentityHelper.GetProviderNameFromRequest(Request);
    if (String.IsNullOrEmpty(ProviderName))
    {
        RedirectOnFail();
        return;
    }
    if (!IsPostBack)
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        var loginInfo = Context.GetOwinContext().Authentication.GetExternalLoginInfo();
        if (loginInfo == null)
        {
            RedirectOnFail();
            return;
        }
        var user = manager.Find(loginInfo.Login);
        if (user != null)
        {
            IdentityHelper.SignIn(manager, user, isPersistent: false);
            IdentityHelper.RedirectToReturnUrl(Request.QueryString["ReturnUrl"], Response);
        }
        else if (User.Identity.IsAuthenticated)
        {
            // Apply Xsrf check when linking
            var verifiedloginInfo = Context.GetOwinContext().Authentication
                .GetExternalLoginInfo(IdentityHelper.XsrfKey, User.Identity.GetUserId());
            if (verifiedloginInfo == null)
            {
                RedirectOnFail();
                return;
            }

            var result = manager.AddLogin(User.Identity.GetUserId<int>(),
                verifiedloginInfo.Login);
            if (result.Succeeded)
            {
                IdentityHelper.RedirectToReturnUrl(Request.QueryString["ReturnUrl"],
                    Response);
            }
            else
            {
                AddErrors(result);
                return;
            }
        }
        else
        {
            email.Text = loginInfo.Email;
        }
    }
}
```

Manage.aspx.cs

```
private bool HasPassword(ApplicationUserManager manager)
{
    return manager.HasPassword(User.Identity.GetUserId<int>());
}

protected void Page_Load()
{
    if (!IsPostBack)
    {
```

```
        // Determine the sections to render
         var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        if (HasPassword(manager))
        {
            changePasswordHolder.Visible = true;
        }
        else
        {
            setPassword.Visible = true;
            changePasswordHolder.Visible = false;
        }
        CanRemoveExternalLogins = manager.GetLogins(
            User.Identity.GetUserId<int>()).Count() > 1;

        // Render success message
        var message = Request.QueryString["m"];
        if (message != null)
        {
            // Strip the query string from action
            Form.Action = ResolveUrl("~/Account/Manage");

            SuccessMessage =
                message == "ChangePwdSuccess" ? "Your password has been changed."
                : message == "SetPwdSuccess" ? "Your password has been set."
                : message == "RemoveLoginSuccess" ? "The account was removed."
                : String.Empty;
            successMessage.Visible = !String.IsNullOrEmpty(SuccessMessage);
        }
    }
}

protected void ChangePassword_Click(object sender, EventArgs e)
{
    if (IsValid)
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        IdentityResult result = manager.ChangePassword(
            User.Identity.GetUserId<int>(),
            CurrentPassword.Text,
            NewPassword.Text);
        if (result.Succeeded)
        {
            var user = manager.FindById(User.Identity.GetUserId<int>());
            IdentityHelper.SignIn(manager, user, isPersistent: false);
            Response.Redirect("~/Account/Manage?m=ChangePwdSuccess");
        }
        else
        {
            AddErrors(result);
        }
    }
}

protected void SetPassword_Click(object sender, EventArgs e)
{
    if (IsValid)
    {
        // Create the local login info and link the local account to the user
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        IdentityResult result = manager.AddPassword(User.Identity.GetUserId<int>(),
            password.Text);
        if (result.Succeeded)
        {
            Response.Redirect("~/Account/Manage?m=SetPwdSuccess");
        }
        else
        {
            AddErrors(result);
        }
```

```
    }
}

public IEnumerable<UserLoginInfo> GetLogins()
{
    var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
    var accounts = manager.GetLogins(User.Identity.GetUserId<int>());
    CanRemoveExternalLogins = accounts.Count() > 1 || HasPassword(manager);
    return accounts;
}

public void RemoveLogin(string loginProvider, string providerKey)
{
    var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
    var result = manager.RemoveLogin(User.Identity.GetUserId<int>(),
        new UserLoginInfo(loginProvider, providerKey));
    string msg = String.Empty;
    if (result.Succeeded)
    {
        var user = manager.FindById(User.Identity.GetUserId<int>());
        IdentityHelper.SignIn(manager, user, isPersistent: false);
        msg = "?m=RemoveLoginSuccess";
    }
    Response.Redirect("~/Account/Manage" + msg);
}
```

You can now run the application and register a new user.

For Web Forms with Update 3, change Account pages to pass the key type

For Web Forms with Update 3, you need to change the following pages.

Confirm.aspx.cx

```
protected void Page_Load(object sender, EventArgs e)
{
    string code = IdentityHelper.GetCodeFromRequest(Request);
    string userId = IdentityHelper.GetUserIdFromRequest(Request);
    if (code != null && userId != null)
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        var result = manager.ConfirmEmail(Int32.Parse(userId), code);
        if (result.Succeeded)
        {
            StatusMessage = "Thank you for confirming your account.";
            return;
        }
    }

    StatusMessage = "An error has occurred";
}
```

RegisterExternalLogin.aspx.cs

```
protected void Page_Load()
{
    // Process the result from an auth provider in the request
    ProviderName = IdentityHelper.GetProviderNameFromRequest(Request);
    if (String.IsNullOrEmpty(ProviderName))
    {
        RedirectOnFail();
        return;
    }
    if (!IsPostBack)
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        var loginInfo = Context.GetOwinContext().Authentication.GetExternalLoginInfo();
        if (loginInfo == null)
        {
            RedirectOnFail();
            return;
        }
        var user = manager.Find(loginInfo.Login);
        if (user != null)
        {
            IdentityHelper.SignIn(manager, user, isPersistent: false);
            IdentityHelper.RedirectToReturnUrl(Request.QueryString["ReturnUrl"], Response);
        }
        else if (User.Identity.IsAuthenticated)
        {
            // Apply Xsrf check when linking
            var verifiedloginInfo = Context.GetOwinContext().Authentication
                .GetExternalLoginInfo(IdentityHelper.XsrfKey, User.Identity.GetUserId());
            if (verifiedloginInfo == null)
            {
                RedirectOnFail();
                return;
            }

            var result = manager.AddLogin(User.Identity.GetUserId<int>(),
                verifiedloginInfo.Login);
            if (result.Succeeded)
            {
                IdentityHelper.RedirectToReturnUrl(Request.QueryString["ReturnUrl"],
                    Response);
            }
            else
            {
                AddErrors(result);
                return;
            }
        }
        else
        {
            email.Text = loginInfo.Email;
        }
    }
}
```

Manage.aspx.cs

```
public partial class Manage : System.Web.UI.Page
{
    protected string SuccessMessage
    {
        get;
        private set;
    }

    private bool HasPassword(ApplicationUserManager manager)
```

```
    {
        return manager.HasPassword(User.Identity.GetUserId<int>());
    }

    public bool HasPhoneNumber { get; private set; }

    public bool TwoFactorEnabled { get; private set; }

    public bool TwoFactorBrowserRemembered { get; private set; }

    public int LoginsCount { get; set; }

    protected void Page_Load()
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();

        HasPhoneNumber = String.IsNullOrEmpty(manager.GetPhoneNumber(
            User.Identity.GetUserId<int>()));

        // Enable this after setting up two-factor authentientication
        //PhoneNumber.Text = manager.GetPhoneNumber(User.Identity.GetUserId()) ?? String.Empty;

        TwoFactorEnabled = manager.GetTwoFactorEnabled(User.Identity.GetUserId<int>());

        LoginsCount = manager.GetLogins(User.Identity.GetUserId<int>()).Count;

        var authenticationManager = HttpContext.Current.GetOwinContext().Authentication;

        if (!IsPostBack)
        {
            // Determine the sections to render
            if (HasPassword(manager))
            {
                ChangePassword.Visible = true;
            }
            else
            {
                CreatePassword.Visible = true;
                ChangePassword.Visible = false;
            }

            // Render success message
            var message = Request.QueryString["m"];
            if (message != null)
            {
                // Strip the query string from action
                Form.Action = ResolveUrl("~/Account/Manage");

                SuccessMessage =
                    message == "ChangePwdSuccess" ? "Your password has been changed."
                    : message == "SetPwdSuccess" ? "Your password has been set."
                    : message == "RemoveLoginSuccess" ? "The account was removed."
                    : message == "AddPhoneNumberSuccess" ? "Phone number has been added"
                    : message == "RemovePhoneNumberSuccess" ? "Phone number was removed"
                    : String.Empty;
                successMessage.Visible = !String.IsNullOrEmpty(SuccessMessage);
            }
        }
    }

    private void AddErrors(IdentityResult result)
    {
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError("", error);
        }
    }

    // Remove phonenumber from user
```

```
    protected void RemovePhone_Click(object sender, EventArgs e)
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        var result = manager.SetPhoneNumber(User.Identity.GetUserId<int>(), null);
        if (!result.Succeeded)
        {
            return;
        }
        var user = manager.FindById(User.Identity.GetUserId<int>());
        if (user != null)
        {
            IdentityHelper.SignIn(manager, user, isPersistent: false);
            Response.Redirect("/Account/Manage?m=RemovePhoneNumberSuccess");
        }
    }

    // DisableTwoFactorAuthentication
    protected void TwoFactorDisable_Click(object sender, EventArgs e)
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        manager.SetTwoFactorEnabled(User.Identity.GetUserId<int>(), false);

        Response.Redirect("/Account/Manage");
    }

    //EnableTwoFactorAuthentication
    protected void TwoFactorEnable_Click(object sender, EventArgs e)
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        manager.SetTwoFactorEnabled(User.Identity.GetUserId<int>(), true);

        Response.Redirect("/Account/Manage");
    }
}
```

VerifyPhoneNumber.aspx.cs

```
public partial class VerifyPhoneNumber : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        var phonenumber = Request.QueryString["PhoneNumber"];
        var code = manager.GenerateChangePhoneNumberToken(
            User.Identity.GetUserId<int>(), phonenumber);
        PhoneNumber.Value = phonenumber;
    }

    protected void Code_Click(object sender, EventArgs e)
    {
        if (!ModelState.IsValid)
        {
            ModelState.AddModelError("", "Invalid code");
            return;
        }

        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();

        var result = manager.ChangePhoneNumber(
            User.Identity.GetUserId<int>(), PhoneNumber.Value, Code.Text);

        if (result.Succeeded)
        {
            var user = manager.FindById(User.Identity.GetUserId<int>());

            if (user != null)
            {
                IdentityHelper.SignIn(manager, user, false);
                Response.Redirect("/Account/Manage?m=AddPhoneNumberSuccess");
            }
        }

        // If we got this far, something failed, redisplay form
        ModelState.AddModelError("", "Failed to verify phone");
    }
}
```

AddPhoneNumber.aspx.cs

```
public partial class AddPhoneNumber : System.Web.UI.Page
{
    protected void PhoneNumber_Click(object sender, EventArgs e)
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        var code = manager.GenerateChangePhoneNumberToken(
            User.Identity.GetUserId<int>(), PhoneNumber.Text);
        if (manager.SmsService != null)
        {
            var message = new IdentityMessage
            {
                Destination = PhoneNumber.Text,
                Body = "Your security code is " + code
            };

            manager.SmsService.Send(message);
        }

        Response.Redirect("/Account/VerifyPhoneNumber?PhoneNumber=" +
HttpUtility.UrlEncode(PhoneNumber.Text));
    }
}
```

ManagePassword.aspx.cs

```
public partial class ManagePassword : System.Web.UI.Page
{
    protected string SuccessMessage
    {
        get;
        private set;
    }

    private bool HasPassword(ApplicationUserManager manager)
    {
        return manager.HasPassword(User.Identity.GetUserId<int>());
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();

        if (!IsPostBack)
        {
            // Determine the sections to render
            if (HasPassword(manager))
            {
                changePasswordHolder.Visible = true;
            }
            else
            {
                setPassword.Visible = true;
                changePasswordHolder.Visible = false;
            }

            // Render success message
            var message = Request.QueryString["m"];
            if (message != null)
            {
                // Strip the query string from action
                Form.Action = ResolveUrl("~/Account/Manage");
            }
        }
    }

    protected void ChangePassword_Click(object sender, EventArgs e)
    {
        if (IsValid)
        {
            var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
            IdentityResult result = manager.ChangePassword(
                User.Identity.GetUserId<int>(), CurrentPassword.Text, NewPassword.Text);
            if (result.Succeeded)
            {
                var user = manager.FindById(User.Identity.GetUserId<int>());
                IdentityHelper.SignIn(manager, user, isPersistent: false);
                Response.Redirect("~/Account/Manage?m=ChangePwdSuccess");
            }
            else
            {
                AddErrors(result);
            }
        }
    }

    protected void SetPassword_Click(object sender, EventArgs e)
    {
        if (IsValid)
        {
            // Create the local login info and link the local account to the user
            var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
```

```
            IdentityResult result = manager.AddPassword(
                User.Identity.GetUserId<int>(), password.Text);
            if (result.Succeeded)
            {
                Response.Redirect("~/Account/Manage?m=SetPwdSuccess");
            }
            else
            {
                AddErrors(result);
            }
        }
    }

    private void AddErrors(IdentityResult result)
    {
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError("", error);
        }
    }
}
```

ManageLogins.aspx.cs

```
public partial class ManageLogins : System.Web.UI.Page
{
    protected string SuccessMessage
    {
        get;
        private set;
    }
    protected bool CanRemoveExternalLogins
    {
        get;
        private set;
    }

    private bool HasPassword(ApplicationUserManager manager)
    {
        return manager.HasPassword(User.Identity.GetUserId<int>());
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        CanRemoveExternalLogins = manager.GetLogins(
            User.Identity.GetUserId<int>()).Count() > 1;

        SuccessMessage = String.Empty;
        successMessage.Visible = !String.IsNullOrEmpty(SuccessMessage);
    }

    public IEnumerable<UserLoginInfo> GetLogins()
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        var accounts = manager.GetLogins(User.Identity.GetUserId<int>());
        CanRemoveExternalLogins = accounts.Count() > 1 || HasPassword(manager);
        return accounts;
    }

    public void RemoveLogin(string loginProvider, string providerKey)
    {
        var manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        var result = manager.RemoveLogin(
            User.Identity.GetUserId<int>(), new UserLoginInfo(loginProvider, providerKey));
        string msg = String.Empty;
        if (result.Succeeded)
        {
            var user = manager.FindById(User.Identity.GetUserId<int>());
            IdentityHelper.SignIn(manager, user, isPersistent: false);
            msg = "?m=RemoveLoginSuccess";
        }
        Response.Redirect("~/Account/ManageLogins" + msg);
    }
}
```

TwoFactorAuthenticationSignIn.aspx.cs

```
public partial class TwoFactorAuthenticationSignIn : System.Web.UI.Page
{
    private ApplicationSignInManager signinManager;
    private ApplicationUserManager manager;

    public TwoFactorAuthenticationSignIn()
    {
        manager = Context.GetOwinContext().GetUserManager<ApplicationUserManager>();
        signinManager = Context.GetOwinContext().GetUserManager<ApplicationSignInManager>();
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        var userId = signinManager.GetVerifiedUserId<ApplicationUser, int>();
        if (userId == default(int))
        {
            Response.Redirect("/Account/Error", true);
        }
        var userFactors = manager.GetValidTwoFactorProviders(userId);
        Providers.DataSource = userFactors.Select(x => x).ToList();
        Providers.DataBind();
    }

    protected void CodeSubmit_Click(object sender, EventArgs e)
    {
        bool rememberMe = false;
        bool.TryParse(Request.QueryString["RememberMe"], out rememberMe);

        var result = signinManager.TwoFactorSignIn<ApplicationUser, int>(SelectedProvider.Value, Code.Text,
isPersistent: rememberMe, rememberBrowser: RememberBrowser.Checked);
        switch (result)
        {
            case SignInStatus.Success:
                IdentityHelper.RedirectToReturnUrl(Request.QueryString["ReturnUrl"], Response);
                break;
            case SignInStatus.LockedOut:
                Response.Redirect("/Account/Lockout");
                break;
            case SignInStatus.Failure:
            default:
                FailureText.Text = "Invalid code";
                ErrorMessage.Visible = true;
                break;
        }
    }

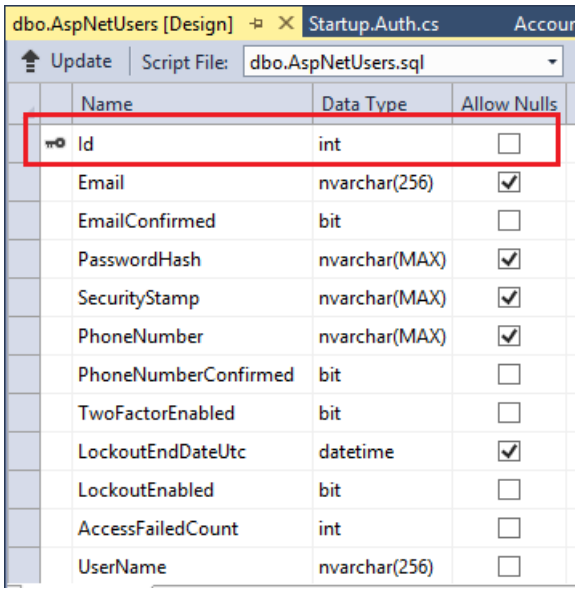
    protected void ProviderSubmit_Click(object sender, EventArgs e)
    {
        if (!signinManager.SendTwoFactorCode(Providers.SelectedValue))
        {
            Response.Redirect("/Account/Error");
        }

        var user = manager.FindById(signinManager.GetVerifiedUserId<ApplicationUser, int>());
        if (user != null)
        {
            var code = manager.GenerateTwoFactorToken(user.Id, Providers.SelectedValue);
        }

        SelectedProvider.Value = Providers.SelectedValue;
        sendcode.Visible = false;
        verifycode.Visible = true;
    }
}
```

Run application

You have finished all of the required changes to the default Web Application template. Run the application and register a new user. After registering the user, you will notice that the AspNetUsers table has an Id column that is an integer.



If you have previously created the ASP.NET Identity tables with a different primary key, you need to make some additional changes. If possible, just delete the existing database. The database will be re-created with the correct design when you run the web application and add a new user. If deletion is not possible, run code first migrations to change the tables. However, the new integer primary key will not be set up as a SQL IDENTITY property in the database. You must manually set the Id column as an IDENTITY.

Other resources

- Overview of Custom Storage Providers for ASP.NET Identity
- Migrating an Existing Website from SQL Membership to ASP.NET Identity
- Migrating Universal Provider Data for Membership and User Profiles to ASP.NET Identity
- Sample application with changed primary key

Migrating to ASP.NET Identity

2/20/2020 • 2 minutes to read • Edit Online

> Migration to ASP.NET Identity from other membership providers

- Migrating an Existing Website from SQL Membership to ASP.NET Identity
- Migrating Universal Provider Data for Membership and User Profiles to ASP.NET Identity (C#)

> Some commands aren't supported if the app uses SQLite as its Identity data store. Due to limitations in the database engine, `Alter` commands throw the following exception:
>
> "System.NotSupportedException: SQLite does not support this migration operation."
>
> As a work around, run Code First migrations on the database to change the tables.

Migrating an Existing Website from SQL Membership to ASP.NET Identity

2/20/2020 • 16 minutes to read • Edit Online

by Rick Anderson, Suhas Joshi

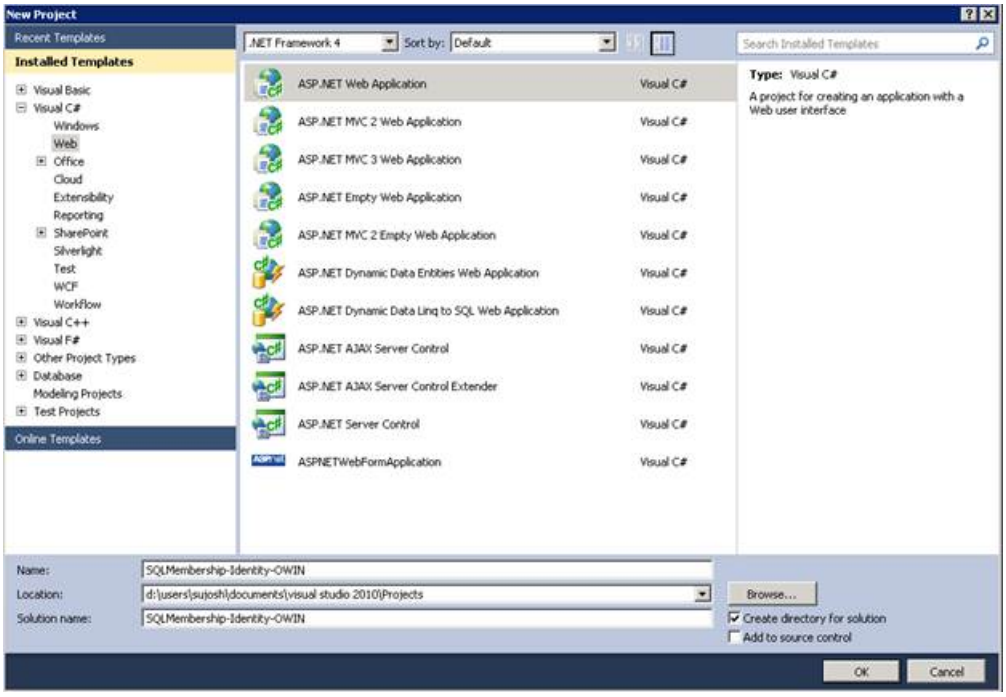
This tutorial illustrates the steps to migrate an existing web application with user and role data created using SQL Membership to the new ASP.NET Identity system. This approach involves changing the existing database schema to the one needed by the ASP.NET Identity and hook in the old/new classes to it. After you adopt this approach, once your database is migrated, future updates to Identity will be handled effortlessly.

For this tutorial, we will take a web application template (Web Forms) created using Visual Studio 2010 to create user and role data. We will then use SQL scripts to migrate the existing database to tables needed by the Identity system. Next we'll install the necessary NuGet packages and add new account management pages which use the Identity system for membership management. As a test of migration, users created using SQL membership should be able to log in and new users should be able to register. You can find the complete sample here. See also Migrating from ASP.NET Membership to ASP.NET Identity.

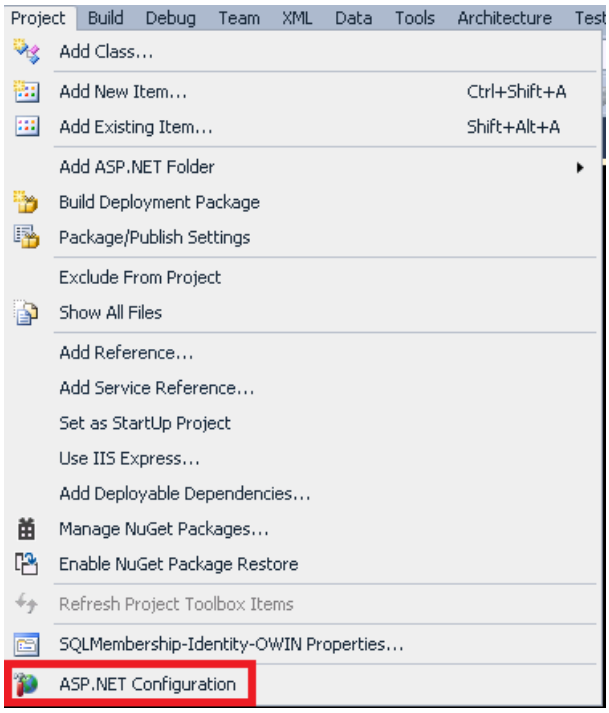
Getting started

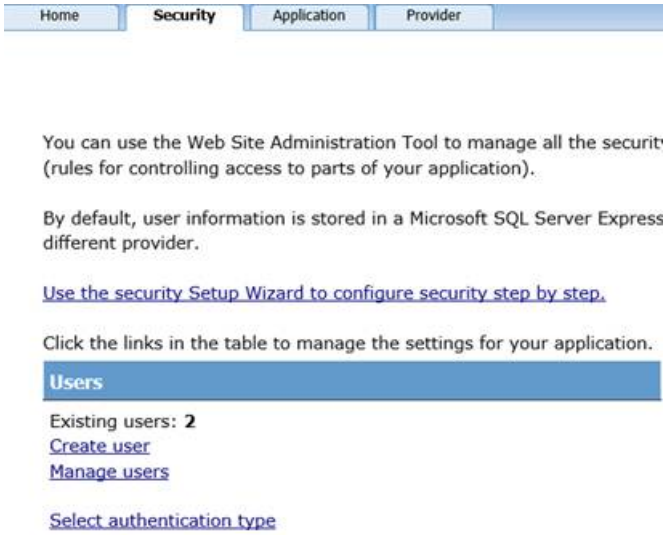
Creating an application with SQL Membership

1. We need to start with an existing application that uses SQL membership and has user and role data. For the purpose of this article, let's create a web application in Visual Studio 2010.

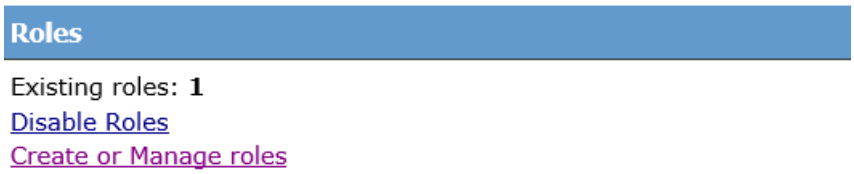


2. Using the ASP.NET Configuration tool, create 2 users: **oldAdminUser** and **oldUser**.

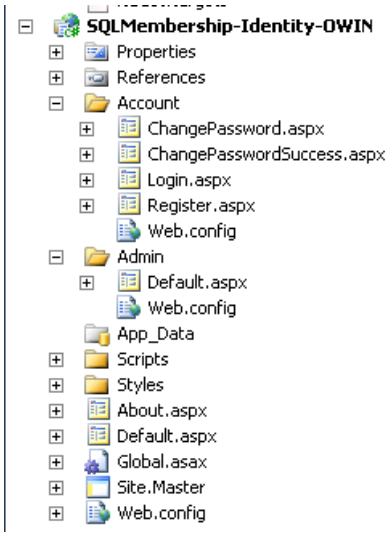




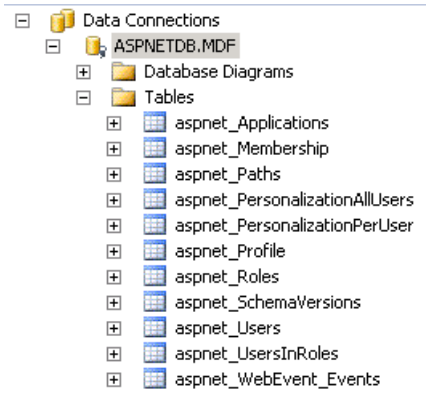
3. Create a role named Admin and add 'oldAdminUser' as a user in that role.



4. Create an Admin section of the site with a Default.aspx. Set the authorization tag in the web.config file to enable access only to users in Admin roles. More information can be found here https://www.asp.net/web-forms/tutorials/security/roles/role-based-authorization-cs



5. View the database in Server Explorer to understand the tables created by the SQL membership system. The user login data is stored in the aspnet_Users and aspnet_Membership tables, while role data is stored in the aspnet_Roles table. Information about which users are in which roles is stored in the aspnet_UsersInRoles table. For basic membership management it is sufficient to port the information in the above tables to the ASP.NET Identity system.



Migrating to Visual Studio 2013

1. Install Visual Studio Express 2013 for Web or Visual Studio 2013 along with the latest updates.

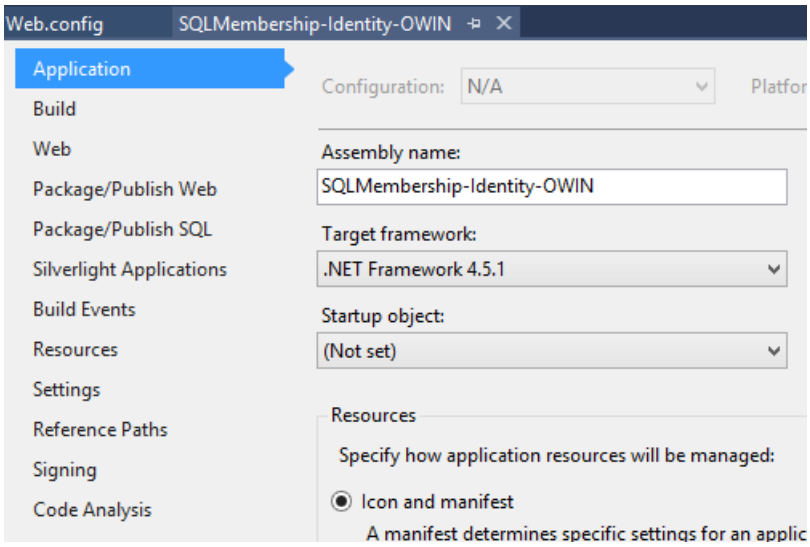
2. Open the above project in your installed version of Visual Studio. If SQL Server Express is not installed on the machine, a prompt is displayed when you open the project, since the connection string uses SQL Express. You can either choose to install SQL Express or as work around change the connection string to LocalDb. For this article we'll change it to LocalDb.

3. Open web.config and change the connection string from .SQLExpress to (LocalDb)v11.0. Remove 'User Instance=true' from the connection string.

```
<configuration>
  <connectionStrings>
    <add name="ApplicationServices"
         connectionString="data source=(LocalDb)\v11.0;Integrated Security=SSPI;AttachDBFilename=|DataDirectory|\aspnetdb.mdf
         providerName="System.Data.SqlClient" />
  </connectionStrings>
```

4. Open Server Explorer and verify that the table schema and data can be observed.

5. The ASP.NET Identity system works with version 4.5 or higher of the framework. Retarget the application to 4.5 or higher.

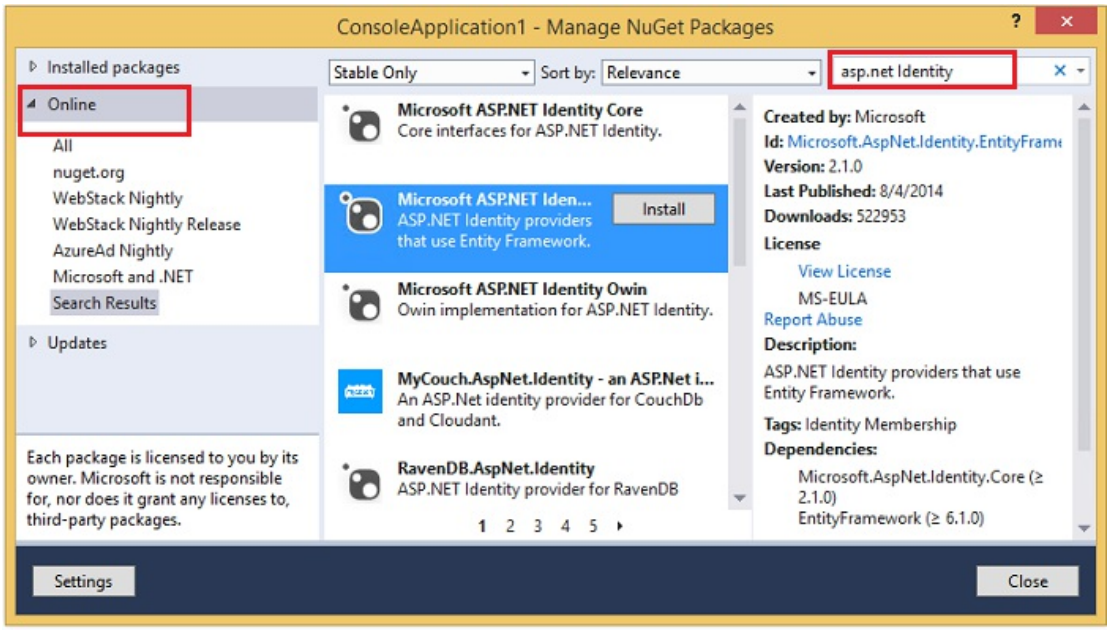


Build the project to verify that there are no errors.

Installing the Nuget packages

1. In Solution Explorer, right-click the project > **Manage NuGet Packages**. In the search box, enter "Asp.net Identity". Select the package in the list of results and click install. Accept the license agreement by clicking on "I Accept" button. Note that this package will install the dependency packages: EntityFramework and Microsoft ASP.NET Identity Core. Similarly install the following packages (skip the last 4 OWIN packages if you don't want to enable OAuth log-in):

   - Microsoft.AspNet.Identity.Owin
   - Microsoft.Owin.Host.SystemWeb
   - Microsoft.Owin.Security.Facebook
   - Microsoft.Owin.Security.Google
   - Microsoft.Owin.Security.MicrosoftAccount
   - Microsoft.Owin.Security.Twitter



Migrate database to the new Identity system

The next step is to migrate the existing database to a schema required by the ASP.NET Identity system. To achieve this we run a SQL script which has a set of commands to create new tables and migrate existing user information

to the new tables. The script file can be found here.

This script file is specific to this sample. If the schema for the tables created using SQL membership is customized or modified the scripts need to be changed accordingly.

How to generate the SQL script for schema migration

For ASP.NET Identity classes to work out of the box with the data of existing users, we need to migrate the database schema to the one needed by ASP.NET Identity. We can do this by adding new tables and copying the existing information to those tables. By default ASP.NET Identity uses EntityFramework to map the Identity model classes back to the database to store/retrieve information. These model classes implement the core Identity interfaces defining user and role objects. The tables and the columns in the database are based on these model classes. The EntityFramework model classes in Identity v2.1.0 and their properties are as defined below

IDENTITYUSER	TYPE	IDENTITYROLE	IDENTITYUSERROLE	IDENTITYUSERLOGIN	IDENTITYUSERCLAIM
Id	string	Id	RoleId	ProviderKey	Id
Username	string	Name	UserId	UserId	ClaimType
PasswordHash	string			LoginProvider	ClaimValue
SecurityStamp	string				User_Id
Email	string				
EmailConfirmed	bool				
PhoneNumber	string				
PhoneNumberConfirmed	bool				
LockoutEnabled	bool				
LockoutEndDate	DateTime				
AccessFailedCount	int				

We need to have tables for each of these models with columns corresponding to the properties. The mapping between classes and tables is defined in the `OnModelCreating` method of the `IdentityDBContext`. This is known as the fluent API method of configuration and more information can be found here. The configuration for the classes is as mentioned below

CLASS	TABLE	PRIMARY KEY	FOREIGN KEY
IdentityUser	AspnetUsers	Id	
IdentityRole	AspnetRoles	Id	
IdentityUserRole	AspnetUserRole	UserId + RoleId	User_Id->AspnetUsers RoleId->AspnetRoles

CLASS	TABLE	PRIMARY KEY	FOREIGN KEY
IdentityUserLogin	AspnetUserLogins	ProviderKey+UserId + LoginProvider	UserId->AspnetUsers
IdentityUserClaim	AspnetUserClaims	Id	User_Id->AspnetUsers

With this information we can create SQL statements to create new tables. We can either write each statement individually or generate the entire script using EntityFramework PowerShell commands which we can then edit as required. To do this, in VS open the **Package Manager Console** from the **View** or **Tools** menu

- Run command "Enable-Migrations" to enable EntityFramework migrations.
- Run command "Add-migration initial" which creates the initial setup code to create the database in C#/VB.
- The final step is to run "Update-Database –Script" command that generates the SQL script based on the model classes.

> Some commands aren't supported if the app uses SQLite as its Identity data store. Due to limitations in the database engine, `Alter` commands throw the following exception:
>
> "System.NotSupportedException: SQLite does not support this migration operation."
>
> As a work around, run Code First migrations on the database to change the tables.

This database generation script can be used as a start where we'll be making additional changes to add new columns and copy data. The advantage of this is that we generate the `_MigrationHistory` table which is used by EntityFramework to modify the database schema when the model classes change for future versions of Identity releases.

The SQL membership user information had other properties in addition to the ones in the Identity user model class namely email, password attempts, last login date, last lock-out date etc. This is useful information and we would like it to be carried over to the Identity system. This can be done by adding additional properties to the user model and mapping them back to the table columns in the database. We can do this by adding a class that subclasses the `IdentityUser` model. We can add the properties to this custom class and edit the SQL script to add the corresponding columns when creating the table. The code for this class is described further in the article. The SQL script for creating the `AspnetUsers` table after adding the new properties would be

```
CREATE TABLE [dbo].[AspNetUsers] (
    [Id]            NVARCHAR (128) NOT NULL,
    [UserName]      NVARCHAR (MAX) NULL,
    [PasswordHash]  NVARCHAR (MAX) NULL,
    [SecurityStamp] NVARCHAR (MAX) NULL,
    [EmailConfirmed]       BIT            NOT NULL,
    [PhoneNumber]          NVARCHAR (MAX) NULL,
    [PhoneNumberConfirmed] BIT            NOT NULL,
    [TwoFactorEnabled]     BIT            NOT NULL,
    [LockoutEndDateUtc]    DATETIME       NULL,
    [LockoutEnabled]       BIT            NOT NULL,
    [AccessFailedCount]    INT            NOT NULL,
    [ApplicationId]                          UNIQUEIDENTIFIER NOT NULL,
    [LegacyPasswordHash]  NVARCHAR (MAX) NULL,
    [LoweredUserName]  NVARCHAR (256)   NOT NULL,
    [MobileAlias]      NVARCHAR (16)    DEFAULT (NULL) NULL,
    [IsAnonymous]      BIT              DEFAULT ((0)) NOT NULL,
    [LastActivityDate] DATETIME2         NOT NULL,
    [MobilePIN]                              NVARCHAR (16)    NULL,
    [Email]                                  NVARCHAR (256)   NULL,
    [LoweredEmail]                           NVARCHAR (256)   NULL,
    [PasswordQuestion]                       NVARCHAR (256)   NULL,
    [PasswordAnswer]                         NVARCHAR (128)   NULL,
    [IsApproved]                             BIT              NOT NULL,
    [IsLockedOut]                            BIT              NOT NULL,
    [CreateDate]                             DATETIME2               NOT NULL,
    [LastLoginDate]                          DATETIME2         NOT NULL,
    [LastPasswordChangedDate]                DATETIME2         NOT NULL,
    [LastLockoutDate]                        DATETIME2         NOT NULL,
    [FailedPasswordAttemptCount]             INT              NOT NULL,
    [FailedPasswordAttemptWindowStart]       DATETIME2         NOT NULL,
    [FailedPasswordAnswerAttemptCount]       INT              NOT NULL,
    [FailedPasswordAnswerAttemptWindowStart] DATETIME2         NOT NULL,
    [Comment]                                NTEXT            NULL,
    CONSTRAINT [PK_dbo.AspNetUsers] PRIMARY KEY CLUSTERED ([Id] ASC),
    FOREIGN KEY ([ApplicationId]) REFERENCES [dbo].[aspnet_Applications] ([ApplicationId]),
);
```

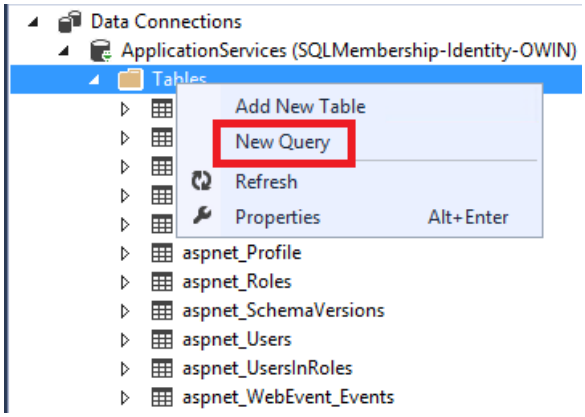
Next we need to copy the existing information from the SQL membership database to the newly added tables for Identity. This can be done through SQL by copying data directly from one table to another. To add data into the rows of table, we use the `INSERT INTO [Table]` construct. To copy from another table we can use the `INSERT INTO` statement along with the `SELECT` statement. To get all the user information we need to query the *aspnet_Users* and *aspnet_Membership* tables and copy the data to the *AspNetUsers* table. We use the `INSERT INTO` and `SELECT` along with `JOIN` and `LEFT OUTER JOIN` statements. For more information about querying and copying data between tables, refer to this link. Additionally the AspnetUserLogins and AspnetUserClaims tables are empty to begin with since there is no information in SQL membership that maps to this by default. The only information copied is for users and roles. For the project created in the previous steps, the SQL query to copy information to the users table would be

```
INSERT INTO AspNetUsers(Id,UserName,PasswordHash,SecurityStamp,EmailConfirmed,
PhoneNumber,PhoneNumberConfirmed,TwoFactorEnabled,LockoutEndDateUtc,LockoutEnabled,AccessFailedCount,
ApplicationId,LoweredUserName,MobileAlias,IsAnonymous,LastActivityDate,LegacyPasswordHash,
MobilePIN,Email,LoweredEmail,PasswordQuestion,PasswordAnswer,IsApproved,IsLockedOut,CreateDate,
LastLoginDate,LastPasswordChangedDate,LastLockoutDate,FailedPasswordAttemptCount,
FailedPasswordAnswerAttemptWindowStart,FailedPasswordAnswerAttemptCount,FailedPasswordAttemptWindowStart,Comm
ent)
SELECT aspnet_Users.UserId,aspnet_Users.UserName,
(aspnet_Membership.Password+'|'+CAST(aspnet_Membership.PasswordFormat as
varchar)+'|'+aspnet_Membership.PasswordSalt),NewID(),
'true',NULL,'false','true',aspnet_Membership.LastLockoutDate,'true','0',
aspnet_Users.ApplicationId,aspnet_Users.LoweredUserName,
aspnet_Users.MobileAlias,aspnet_Users.IsAnonymous,aspnet_Users.LastActivityDate,aspnet_Membership.Password,
aspnet_Membership.MobilePIN,aspnet_Membership.Email,aspnet_Membership.LoweredEmail,aspnet_Membership.Password
Question,aspnet_Membership.PasswordAnswer,
aspnet_Membership.IsApproved,aspnet_Membership.IsLockedOut,aspnet_Membership.CreateDate,aspnet_Membership.Las
tLoginDate,aspnet_Membership.LastPasswordChangedDate,
aspnet_Membership.LastLockoutDate,aspnet_Membership.FailedPasswordAttemptCount,
aspnet_Membership.FailedPasswordAnswerAttemptWindowStart,
aspnet_Membership.FailedPasswordAnswerAttemptCount,aspnet_Membership.FailedPasswordAttemptWindowStart,aspnet_
Membership.Comment
FROM aspnet_Users
LEFT OUTER JOIN aspnet_Membership ON aspnet_Membership.ApplicationId = aspnet_Users.ApplicationId
AND aspnet_Users.UserId = aspnet_Membership.UserId;
```

In the above SQL statement, information about each user from the *aspnet_Users* and *aspnet_Membership* tables is copied into the columns of the *AspnetUsers* table. The only modification done here is when we copy the password. Since the encryption algorithm for passwords in SQL membership used 'PasswordSalt' and 'PasswordFormat', we copy that too along with the hashed password so that it can be used to decrypt the password by Identity. This is explained further in the article when hooking up a custom password hasher.

This script file is specific to this sample. For applications which have additional tables, developers can follow a similar approach to add additional properties on the user model class and map them to columns in the AspnetUsers table. To run the script,

1. Open Server Explorer. Expand the 'ApplicationServices' connection to display the tables. Right click on the Tables node and select the 'New Query' option



2. In the query window, copy and paste the entire SQL script from the Migrations.sql file. Run the script file by hitting the 'Execute' arrow button.

```
SQLQuery1.sql *
B1615B31750BE7C9C207609833D8(
DROP TABLE AspNetUserRoles;

DROP TABLE AspNetUserClaims;

DROP TABLE AspNetUserLogins;

DROP TABLE AspNetRoles;

DROP TABLE AspNetUsers;

CREATE TABLE [dbo].[AspNetUsers] (
    [Id]              NVARCHAR (128) NOT NULL,
    [UserName]        NVARCHAR (MAX) NULL,
    [PasswordHash]    NVARCHAR (MAX) NULL,
    [SecurityStamp]   NVARCHAR (MAX) NULL,
    [Discriminator]   NVARCHAR (128) NOT NULL,
    [ApplicationId]                          UNIQUEIDENTIFIER NOT NULL,
    [PasswordFormat]                         INT              DEFAULT ((0)) NULL,
    [PasswordSalt]                           NVARCHAR (128)   NULL,
    [LegacyPasswordHash]  NVARCHAR (MAX) NULL,
    [LoweredUserName]  NVARCHAR (256)   NOT NULL,
    [MobileAlias]      NVARCHAR (16)    DEFAULT (NULL) NULL,
    [IsAnonymous]      BIT              DEFAULT ((0)) NOT NULL,
    [LastActivityDate] DATETIME         NOT NULL,
    [MobilePIN]                              NVARCHAR (16)    NULL,
    [Email]                                  NVARCHAR (256)   NULL,
    [LoweredEmail]                           NVARCHAR (256)   NULL,
    [PasswordQuestion]                       NVARCHAR (256)   NULL,
    [PasswordAnswer]                         NVARCHAR (128)   NULL,
    [IsApproved]                             BIT              NOT NULL,
    [IsLockedOut]                            BIT              NOT NULL,
    [CreateDate]                             DATETIME         NOT NULL,
    [LastLoginDate]                          DATETIME         NOT NULL,
    [LastPasswordChangedDate]                DATETIME         NOT NULL,
    [LastLockoutDate]                        DATETIME         NOT NULL,
    [FailedPasswordAttemptCount]             INT              NOT NULL,
    [FailedPasswordAttemptWindowStart]       DATETIME         NOT NULL,
    [FailedPasswordAnswerAttemptCount]       INT              NOT NULL,
100 %
```

Refresh the Server Explorer window. Five new tables are created in the database.

```
SQLQuery1.sql *   User.cs
C8D7444F4FA827C3FB15138DF9C8
IF OBJECT_ID('AspNetUserRoles', 'U') IS NOT NULL
BEGIN
  DROP TABLE AspNetUserRoles;
  END

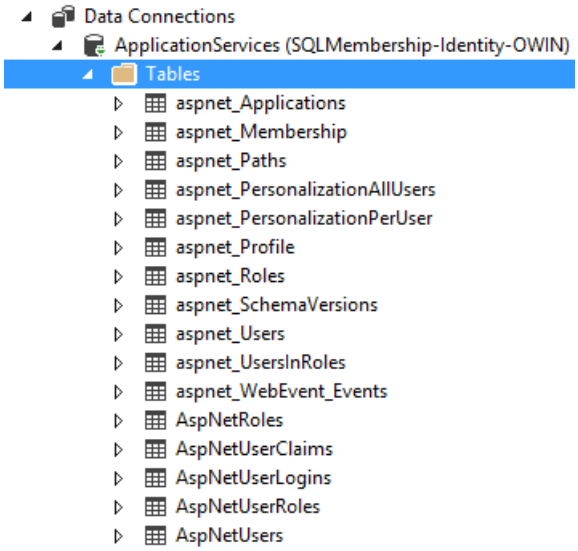
IF OBJECT_ID('AspNetUserClaims', 'U') IS NOT NULL
BEGIN
  DROP TABLE AspNetUserClaims;
  END

IF OBJECT_ID('AspNetUserLogins', 'U') IS NOT NULL
BEGIN
  DROP TABLE AspNetUserLogins;
  END

IF OBJECT_ID('AspNetRoles', 'U') IS NOT NULL
BEGIN
  DROP TABLE AspNetRoles;
  END

IF OBJECT_ID('AspNetUsers', 'U') IS NOT NULL
BEGIN
  DROP TABLE AspNetUsers;
  END

CREATE TABLE [dbo].[AspNetUsers] (
    [Id]              NVARCHAR (128) NOT NULL,
    [UserName]        NVARCHAR (MAX) NULL,
    [PasswordHash]    NVARCHAR (MAX) NULL,
    [SecurityStamp]   NVARCHAR (MAX) NULL,
    [EmailConfirmed]        BIT            NOT NULL,
    [PhoneNumber]           NVARCHAR (MAX) NULL,
    [PhoneNumberConfirmed]  BIT            NOT NULL,
    [TwoFactorEnabled]      BIT            NOT NULL,
    [LockoutEndDateUtc]     DATETIME       NULL,
    [LockoutEnabled]        BIT            NOT NULL,
```



Below is how the information in the SQL membership tables are mapped to the new Identity system.

aspnet_Roles --> AspNetRoles

asp_netUsers and asp_netMembership --> AspNetUsers

aspnet_UserInRoles --> AspNetUserRoles

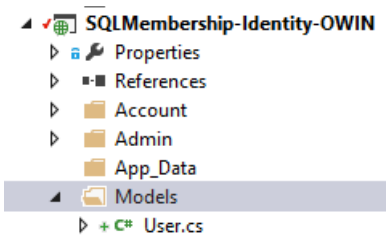
As explained in the above section, the AspNetUserClaims and AspNetUserLogins tables are empty. The 'Discriminator' field in the AspNetUser table should match the model class name which is defined as a next step. Also the PasswordHash column is in the form 'encrypted password |password salt|password format'. This enables you to use special SQL membership crypto logic so that you can reuse old passwords. That is explained in later in the article.

Creating models and membership pages

As mentioned earlier, the Identity feature uses Entity Framework to talk to the database for storing account information by default. To work with existing data in the table, we need to create model classes which map back to the tables and hook them up in the Identity system. As part of the Identity contract, the model classes should either implement the interfaces defined in the Identity.Core dll or can extend the existing implementation of these interfaces available in Microsoft.AspNet.Identity.EntityFramework.

In our sample, the AspNetRoles, AspNetUserClaims, AspNetLogins and AspNetUserRole tables have columns that are similar to the existing implementation of the Identity system. Hence we can reuse the existing classes to map to these tables. The AspNetUser table has some additional columns which are used to store additional information from the SQL membership tables. This can be mapped by creating a model class that extend the existing implementation of 'IdentityUser' and add the additional properties.

1. Create a Models folder in the project and add a class User. The name of the class should match the data added in the 'Discriminator' column of 'AspnetUsers' table.



The User class should extend the IdentityUser class found in the *Microsoft.AspNet.Identity.EntityFramework* dll. Declare the properties in class that map back to the AspNetUser columns. The properties ID, Username, PasswordHash and SecurityStamp are defined in the IdentityUser and so are omitted. Below is the code for the User class that has all the properties

```
public class User : IdentityUser
{
    public User()
    {
        CreateDate = DateTime.Now;
        IsApproved = false;
        LastLoginDate = DateTime.Now;
        LastActivityDate = DateTime.Now;
        LastPasswordChangedDate = DateTime.Now;
        LastLockoutDate = DateTime.Parse("1/1/1754");
        FailedPasswordAnswerAttemptWindowStart = DateTime.Parse("1/1/1754");
        FailedPasswordAttemptWindowStart = DateTime.Parse("1/1/1754");
    }

    public System.Guid ApplicationId { get; set; }
    public string MobileAlias { get; set; }
    public bool IsAnonymous { get; set; }
    public System.DateTime LastActivityDate { get; set; }
    public string MobilePIN { get; set; }
    public string LoweredEmail { get; set; }
    public string LoweredUserName { get; set; }
    public string PasswordQuestion { get; set; }
    public string PasswordAnswer { get; set; }
    public bool IsApproved { get; set; }
    public bool IsLockedOut { get; set; }
    public System.DateTime CreateDate { get; set; }
    public System.DateTime LastLoginDate { get; set; }
    public System.DateTime LastPasswordChangedDate { get; set; }
    public System.DateTime LastLockoutDate { get; set; }
    public int FailedPasswordAttemptCount { get; set; }
    public System.DateTime FailedPasswordAttemptWindowStart { get; set; }
    public int FailedPasswordAnswerAttemptCount { get; set; }
    public System.DateTime FailedPasswordAnswerAttemptWindowStart { get; set; }
    public string Comment { get; set; }
}
```

2. An Entity Framework DbContext class is required in order to persist data in models back to tables and retrieve data from tables to populate the models. *Microsoft.AspNet.Identity.EntityFramework* dll defines the IdentityDbContext class which interacts with the Identity tables to retrieve and store information. The IdentityDbContext<tuser> takes a 'TUser' class which can be any class that extends the IdentityUser class.

   Create a new class ApplicationDBContext that extends IdentityDbContext under the 'Models' folder, passing in the 'User' class created in step 1

```
public class ApplicationDbContext : IdentityDbContext<User>
{

}
```

3. User management in the new Identity system is done using the UserManager<tuser> class defined in the *Microsoft.AspNet.Identity.EntityFramework* dll. We need to create a custom class that extends UserManager, passing in the 'User' class created in step 1.

   In the Models folder create a new class UserManager that extends UserManager<user>

```
public class UserManager : UserManager<User>
{

}
```

4. The passwords of the users of the application are encrypted and stored in the database. The crypto

algorithm used in SQL membership is different from the one in the new Identity system. To reuse old passwords we need to selectively decrypt passwords when old users log in using the SQL memberships algorithm while using the crypto algorithm in Identity for the new users.

The UserManager class has a property 'PasswordHasher' which stores an instance of a class that implements the 'IPasswordHasher' interface. This is used to encrypt/decrypt passwords during user authentication transactions. In the UserManager class defined in step 3, create a new class SQLPasswordHasher and copy the below code.

```
public class SQLPasswordHasher : PasswordHasher
{
    public override string HashPassword(string password)
    {
        return base.HashPassword(password);
    }

    public override PasswordVerificationResult VerifyHashedPassword(string  hashedPassword, string
providedPassword)
    {
        string[] passwordProperties = hashedPassword.Split('|');
        if (passwordProperties.Length != 3)
        {
            return base.VerifyHashedPassword(hashedPassword, providedPassword);
        }
        else
        {
            string passwordHash = passwordProperties[0];
            int passwordformat = 1;
            string salt = passwordProperties[2];
            if (String.Equals(EncryptPassword(providedPassword, passwordformat, salt), passwordHash,
StringComparison.CurrentCultureIgnoreCase))
            {
                return PasswordVerificationResult.SuccessRehashNeeded;
            }
            else
            {
                return PasswordVerificationResult.Failed;
            }
        }
    }

    //This is copied from the existing SQL providers and is provided only for back-compat.
    private string EncryptPassword(string pass, int passwordFormat, string salt)
    {
        if (passwordFormat == 0) // MembershipPasswordFormat.Clear
            return pass;

        byte[] bIn = Encoding.Unicode.GetBytes(pass);
        byte[] bSalt = Convert.FromBase64String(salt);
        byte[] bRet = null;

        if (passwordFormat == 1)
        { // MembershipPasswordFormat.Hashed
            HashAlgorithm hm = HashAlgorithm.Create("SHA1");
            if (hm is KeyedHashAlgorithm)
            {
                KeyedHashAlgorithm kha = (KeyedHashAlgorithm)hm;
                if (kha.Key.Length == bSalt.Length)
                {
                    kha.Key = bSalt;
                }
                else if (kha.Key.Length < bSalt.Length)
                {
                    byte[] bKey = new byte[kha.Key.Length];
                    Buffer.BlockCopy(bSalt, 0, bKey, 0, bKey.Length);
                    kha.Key = bKey;
```

```
                }
                else
                {
                    byte[] bKey = new byte[kha.Key.Length];
                    for (int iter = 0; iter < bKey.Length; )
                    {
                        int len = Math.Min(bSalt.Length, bKey.Length - iter);
                        Buffer.BlockCopy(bSalt, 0, bKey, iter, len);
                        iter += len;
                    }
                    kha.Key = bKey;
                }
                bRet = kha.ComputeHash(bIn);
            }
            else
            {
                byte[] bAll = new byte[bSalt.Length + bIn.Length];
                Buffer.BlockCopy(bSalt, 0, bAll, 0, bSalt.Length);
                Buffer.BlockCopy(bIn, 0, bAll, bSalt.Length, bIn.Length);
                bRet = hm.ComputeHash(bAll);
            }
        }

        return Convert.ToBase64String(bRet);
    }
```

Resolve the compilation errors by importing the System.Text and System.Security.Cryptography namespaces.

The EncodePassword method encrypts the password according to the default SQL membership crypto implementation. This is taken from the System.Web dll. If the old app used a custom implementation then it should be reflected here. We need to define two other methods *HashPassword* and *VerifyHashedPassword* that use the *EncodePassword* method to hash a given password or verify a plain text password with the one existing in the database.

The SQL membership system used PasswordHash, PasswordSalt and PasswordFormat to hash the password entered by users when they register or change their password. During the migration all the three fields are stored in the PasswordHash column in the AspNetUser table separated by the '|' character. When a user logs in and the password has these fields, we use the SQL membership crypto to check the password; otherwise we use the Identity system's default crypto to verify the password. This way old users would not have to change their passwords once the app is migrated.

5. Declare the constructor for the UserManager class and pass this as the SQLPasswordHasher to the property in the constructor.

```
public UserManager()
            : base(new UserStore<User>(new ApplicationDbContext()))
{
            this.PasswordHasher = new SQLPasswordHasher();
}
```

Create new account management pages

The next step in the migration is to add account management pages that will let a user register and log in. The old account pages from SQL membership use controls that don't work with the new Identity system. To add the new user management pages follow the tutorial at this link https://www.asp.net/identity/overview/getting-started/adding-aspnet-identity-to-an-empty-or-existing-web-forms-project starting from the step 'Adding Web Forms for registering users to your application' since we have already created the project and added the NuGet packages.

We need to make some changes for the sample to work with the project we have here.

- The Register.aspx.cs and Login.aspx.cs code behind classes use the `UserManager` from the Identity packages to create a User. For this example use the UserManager added in the Models folder by following the steps mentioned earlier.
- Use the User class created instead of the IdentityUser in Register.aspx.cs and Login.aspx.cs code behind classes. This hooks in our custom user class into the Identity system.
- The part to create the database can be skipped.
- The developer needs to set the ApplicationId for the new user to match the current application ID. This can be done by querying the ApplicationId for this application before a user object is created in the Register.aspx.cs class and setting it before creating user.

  Example:

  Define a method in Register.aspx.cs page to query the aspnet_Applications table and get the application Id according to application name

  ```
  private Guid GetApplicationID()
  {
      using (SqlConnection connection = new
  SqlConnection(ConfigurationManager.ConnectionStrings["ApplicationServices"].ConnectionString))
      {
          string queryString = "SELECT ApplicationId from aspnet_Applications WHERE ApplicationName =
  '/'"; //Set application name as in database

          SqlCommand command = new SqlCommand(queryString, connection);
          command.Connection.Open();

          var reader = command.ExecuteReader();
          while (reader.Read())
          {
              return reader.GetGuid(0);
          }

          return Guid.NewGuid();
      }
  }
  ```

  Now get set this on the user object

  ```
  var currentApplicationId = GetApplicationID();

  User user = new User() { UserName = Username.Text,
  ApplicationId=currentApplicationId, …};
  ```

Use the old username and password to login an existing user. Use the Register page to create a new user. Also verify that the users are in roles as expected.

Porting to the Identity system helps the user add Open Authentication (OAuth) to the application. Please refer to the sample here which has OAuth enabled.

Next Steps

In this tutorial we showed how to port users from SQL membership to ASP.NET Identity, but we didn't port Profile data. In the next tutorial we'll look into porting Profile data from SQL membership to the new Identity system.

You can leave feedback at the bottom of this article.

Thanks to Tom Dykstra and Rick Anderson for reviewing the article.

Migrating Universal Provider Data for Membership and User Profiles to ASP.NET Identity (C#)

2/20/2020 • 9 minutes to read • Edit Online

by Pranav Rastogi, Rick Anderson, Robert McMurray, Suhas Joshi

This tutorial describes the steps that are necessary to migrate user and role data and user profile data created using Universal Providers of an existing application to the ASP.NET Identity model. The approach mentioned here to migrate user profile data can be used in an application with SQL membership as well.

With the release of Visual Studio 2013, the ASP.NET team introduced a new ASP.NET Identity system, and you can read more about that release here. Following up on the article to migrate web applications from SQL Membership to the new Identity system, this article illustrates the steps to migrate existing applications that follow the Providers model for user and role management to the new Identity model. The focus of this tutorial will be primarily on migrating the user profile data to seamlessly hook it into the new system. Migrating user and role information is similar for SQL membership. The approach followed to migrate profile data can be used in an application with SQL membership as well.

As an example, we will start with a web app created using Visual Studio 2012 which uses the Providers model. We'll then add code for profile management, register a user, add profile data for the users, migrate the database schema, and then change the application to use the Identity system for user and role management. As a test of migration, users created using Universal Providers should be able to log in and new users should be able to register.

> **NOTE**
>
> You can find the complete sample at https://github.com/suhasj/UniversalProviders-Identity-Migrations.

Profile data migration summary

Before starting with the migrations, let us look at the experience of storing profile data in the Providers model. Profile data for application users can be stored in multiple ways, the most common among them being using the inbuilt profile providers shipped along with the Universal Providers. The steps would include

1. Add a class that has properties used to store Profile data.
2. Add a class that extends 'ProfileBase' and implements methods to get the above profile data for the user.
3. Enable using default profile providers in the *web.config* file and define the class declared in step #2 to be used in accessing profile information.

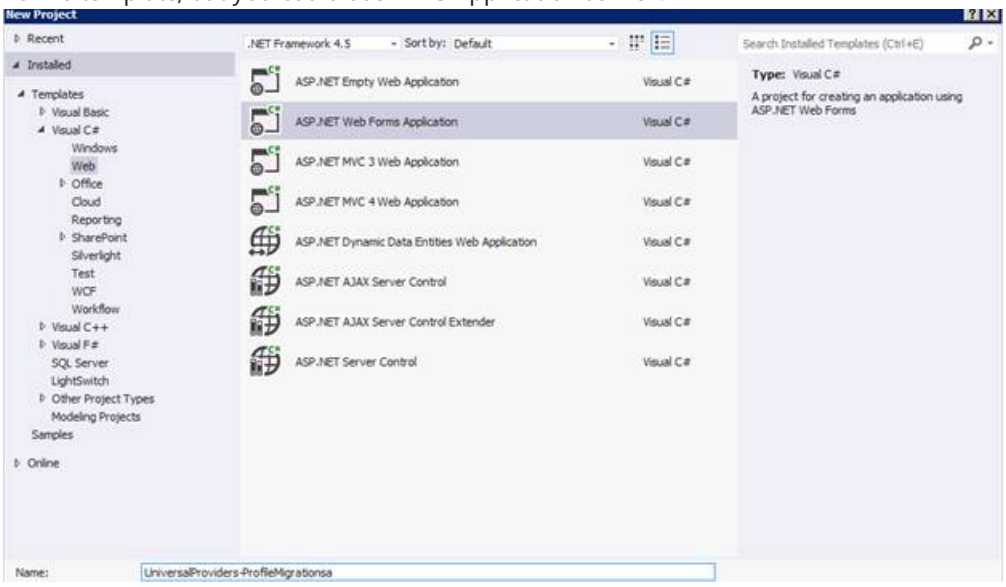
The profile information is stored as serialized xml and binary data in the 'Profiles' table in the database.

After migrating the application to use the new ASP.NET Identity system, the profile information is deserialized and stored as properties on the user class. Each property can then be mapped onto columns in the user table. The advantage here is that the properties can be worked on directly using the user class in addition to not having to serialize/deserialize data information each time when accessing it.

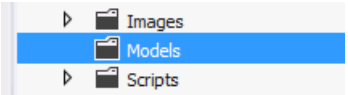
Getting Started

1. Create a new ASP.NET 4.5 Web Forms application in Visual Studio 2012. The current sample uses the Web

Forms template, but you could use MVC Application as well.



2. Create a new folder 'Models' to store profile information



3. As an example, let us store the date of birth, city, height and weight of the user in profile. The height and weight are stored as a custom class called 'PersonalStats'. To store and retrieve the profile, we need a class that extends 'ProfileBase'. Let's create a new class 'AppProfile' to get and store profile information.

```
public class ProfileInfo
{
    public ProfileInfo()
    {
        UserStats = new PersonalStats();
    }
    public DateTime? DateOfBirth { get; set; }
    public PersonalStats UserStats { get; set; }
    public string City { get; set; }
}

public class PersonalStats
{
    public int? Weight { get; set; }
    public int? Height { get; set; }
}

public class AppProfile : ProfileBase
{
    public ProfileInfo ProfileInfo
    {
        get { return (ProfileInfo)GetPropertyValue("ProfileInfo"); }
    }
    public static AppProfile GetProfile()
    {
        return (AppProfile)HttpContext.Current.Profile;
    }
    public static AppProfile GetProfile(string userName)
    {
        return (AppProfile)Create(userName);
    }
}
```

4. Enable profile in the *web.config* file. Enter the class name to be used to store/retrieve user information created in step #3.

```
<profile defaultProvider="DefaultProfileProvider" enabled="true"
    inherits="UniversalProviders_ProfileMigrations.Models.AppProfile">
  <providers>
    .....
  </providers>
</profile>
```

5. Add a web forms page in 'Account' folder to get the profile data from the user and store it. Right click on project and select 'Add new Item'. Add a new webforms page with master page 'AddProfileData.aspx'. Copy the following in the 'MainContent' section:

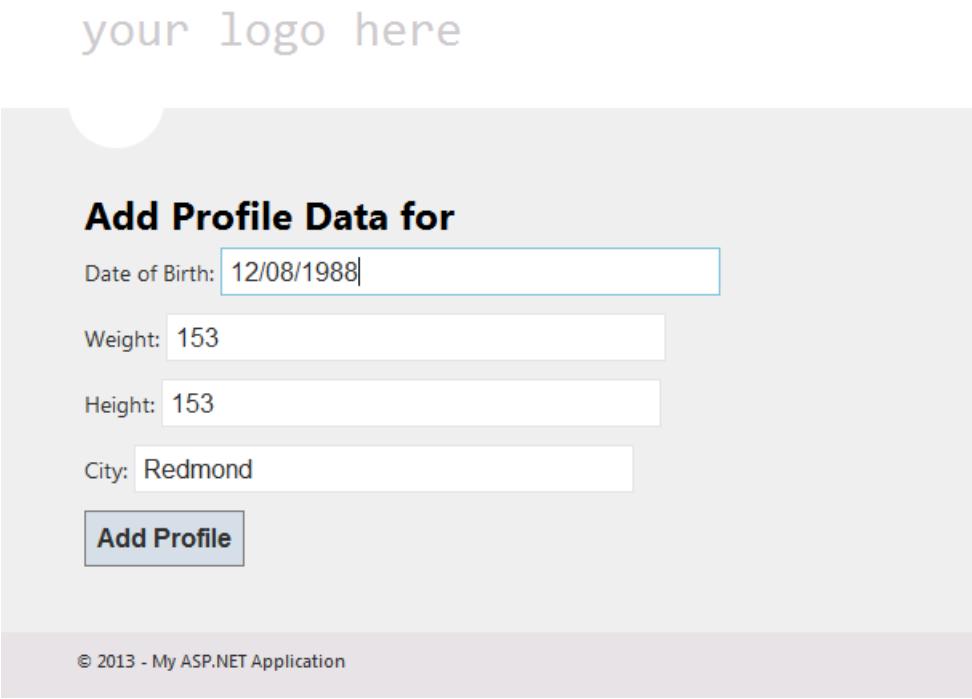
```
<h2> Add Profile Data for <%# User.Identity.Name %></h2>
<asp:Label Text="" ID="Result" runat="server" />
<div>
    Date of Birth:
    <asp:TextBox runat="server" ID="DateOfBirth"/>
</div>
<div>
    Weight:
    <asp:TextBox runat="server" ID="Weight"/>
</div>
<div>
    Height:
    <asp:TextBox runat="server" ID="Height"/>
</div>
<div>
    City:
    <asp:TextBox runat="server" ID="City"/>
</div>
<div>
    <asp:Button Text="Add Profile" ID="Add" OnClick="Add_Click" runat="server" />
</div>
```

Add the following code in the code behind:

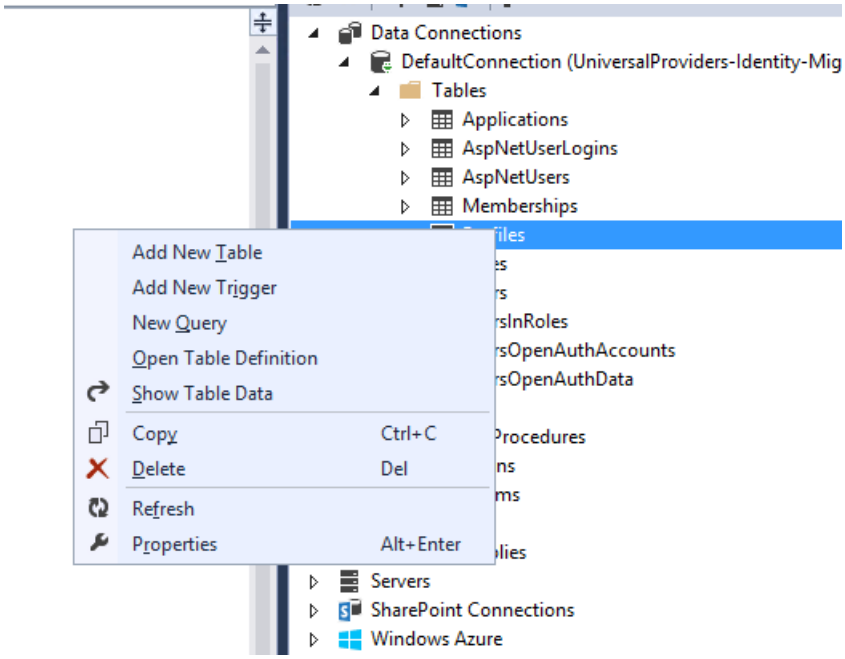
```
protected void Add_Click(object sender, EventArgs e)
{
    AppProfile profile = AppProfile.GetProfile(User.Identity.Name);
    profile.ProfileInfo.DateOfBirth = DateTime.Parse(DateOfBirth.Text);
    profile.ProfileInfo.UserStats.Weight = Int32.Parse(Weight.Text);
    profile.ProfileInfo.UserStats.Height = Int32.Parse(Height.Text);
    profile.ProfileInfo.City = City.Text;
    profile.Save();
}
```

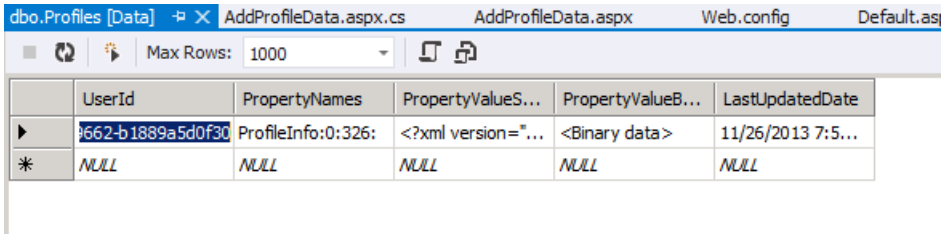
Add the namespace under which AppProfile class is defined to remove the compilation errors.

6. Run the app and create a new user with username '**olduser**'. Navigate to the 'AddProfileData' page and add profile information for the user.



You can verify that the data is stored as serialized xml in the 'Profiles' table using the Server Explorer window. In Visual Studio, from 'View' menu, choose 'Server Explorer'. There should be a data connection for the database defined in the *web.config* file. Clicking on the data connection shows different sub categories. Expand 'Tables' to show the different tables in your database, then right click on 'Profiles' and choose 'Show Table Data' to view the profile data stored in the Profiles table.

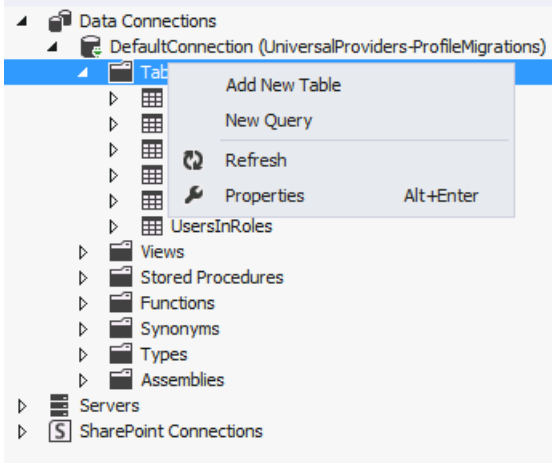




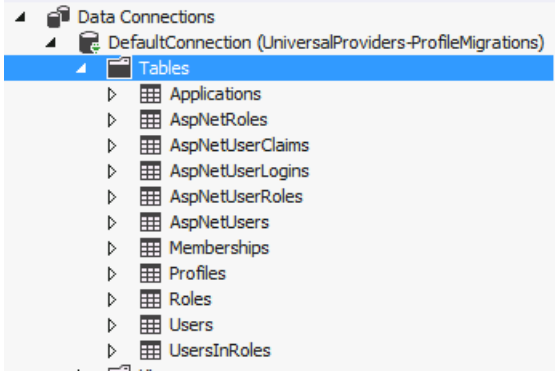
Migrating database schema

To make the existing database work with the Identity system, we need to update the schema in the Identity database to support the fields we added to the original database. This can be done using SQL scripts to create new

tables and copy the existing information. In the 'Server Explorer' window, expand the 'DefaultConnection' to display the tables. Right click Tables and select 'New Query'



Paste the SQL script from https://raw.github.com/suhasj/UniversalProviders-Identity-Migrations/master/Migration.txt and run it. If the 'DefaultConnection' is refreshed, we can see that the new tables are added. You can check the data inside the tables to see that the information has been migrated.



Migrating the application to use ASP.NET Identity

1. Install the Nuget packages needed for ASP.NET Identity:

   - Microsoft.AspNet.Identity.EntityFramework
   - Microsoft.AspNet.Identity.Owin
   - Microsoft.Owin.Host.SystemWeb
   - Microsoft.Owin.Security.Facebook
   - Microsoft.Owin.Security.Google
   - Microsoft.Owin.Security.MicrosoftAccount
   - Microsoft.Owin.Security.Twitter

   More information on managing Nuget packages can be found here

2. To work with existing data in the table, we need to create model classes which map back to the tables and hook them up in the Identity system. As part of the Identity contract, the model classes should either implement the interfaces defined in the Identity.Core dll or can extend the existing implementation of these interfaces available in Microsoft.AspNet.Identity.EntityFramework. We will be using the existing classes for role, user logins and user claims. We need to use a custom user for our sample. Right click on the project and create new folder 'IdentityModels'. Add a new 'User' class as shown below:

```
using Microsoft.AspNet.Identity.EntityFramework;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using UniversalProviders_ProfileMigrations.Models;

namespace UniversalProviders_Identity_Migrations
{
    public class User : IdentityUser
    {
        public User()
        {
            CreateDate = DateTime.UtcNow;
            IsApproved = false;
            LastLoginDate = DateTime.UtcNow;
            LastActivityDate = DateTime.UtcNow;
            LastPasswordChangedDate = DateTime.UtcNow;
            Profile = new ProfileInfo();
        }

        public System.Guid ApplicationId { get; set; }
        public bool IsAnonymous { get; set; }
        public System.DateTime? LastActivityDate { get; set; }
        public string Email { get; set; }
        public string PasswordQuestion { get; set; }
        public string PasswordAnswer { get; set; }
        public bool IsApproved { get; set; }
        public bool IsLockedOut { get; set; }
        public System.DateTime? CreateDate { get; set; }
        public System.DateTime? LastLoginDate { get; set; }
        public System.DateTime? LastPasswordChangedDate { get; set; }
        public System.DateTime? LastLockoutDate { get; set; }
        public int FailedPasswordAttemptCount { get; set; }
        public System.DateTime? FailedPasswordAttemptWindowStart { get; set; }
        public int FailedPasswordAnswerAttemptCount { get; set; }
        public System.DateTime? FailedPasswordAnswerAttemptWindowStart { get; set; }
        public string Comment { get; set; }
        public ProfileInfo Profile { get; set; }
    }
}
```

Notice that the 'ProfileInfo' is now a property on the user class. Hence we can use the user class to directly work with profile data.

Copy the files in the **IdentityModels** and **IdentityAccount** folders from the download source (https://github.com/suhasj/UniversalProviders-Identity-Migrations/tree/master/UniversalProviders-Identity-Migrations). These have the remaining model classes and the new pages needed for user and role management using the ASP.NET Identity APIs. The approach used is similar to the SQL Membership and the detailed explanation can be found here.

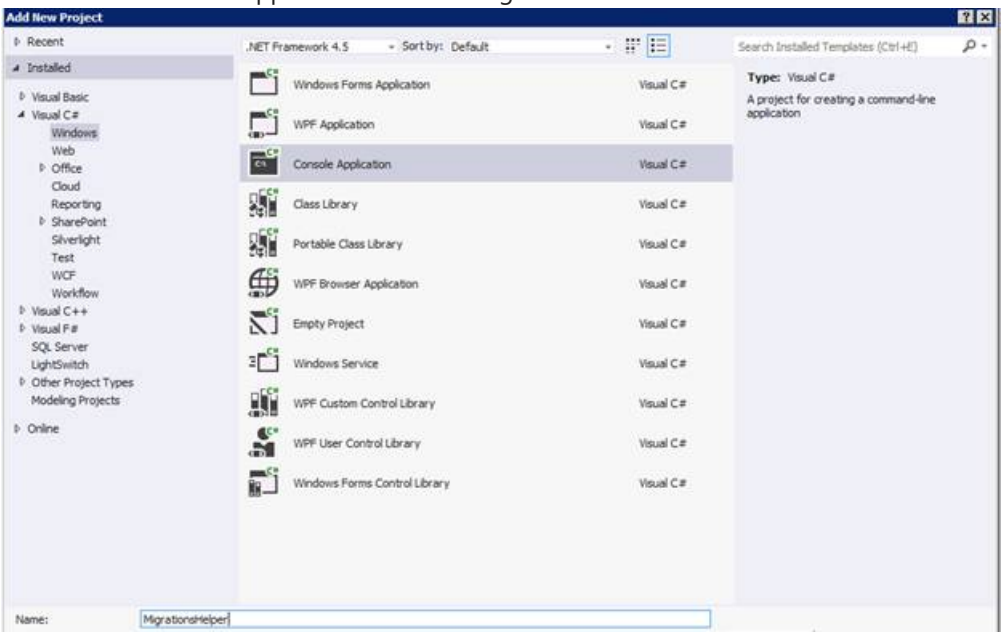
> Some commands aren't supported if the app uses SQLite as its Identity data store. Due to limitations in the database engine, `Alter` commands throw the following exception:
>
> "System.NotSupportedException: SQLite does not support this migration operation."
>
> As a work around, run Code First migrations on the database to change the tables.

Copying Profile data to the new tables

As mentioned earlier, we need to deserialize the xml data in the Profiles tables and store it in the columns of the AspNetUsers table. The new columns were created in the users table in the previous step so all that is left is to

populate those columns with the necessary data. To do this, we will use a console application which is run once to populate the newly created columns in the users table.

1. Create a new console application in the exiting solution.



2. Install the latest version of the Entity Framework package.

3. Add the web application created above as a reference to the console application. To do this right click on Project, then 'Add References', then Solution, then click on the project and click OK.

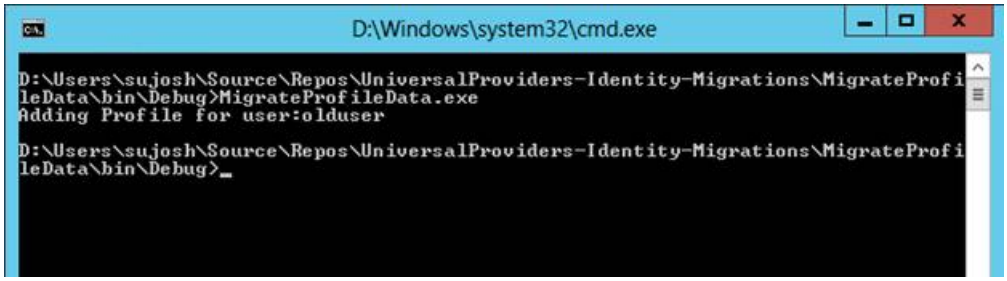
4. Copy the below code in the Program.cs class. This logic reads profile data for each user, serializes it as 'ProfileInfo' object and stores it back to the database.

```
public class Program
{
    var dbContext = new ApplicationDbContext();
    foreach (var profile in dbContext.Profiles)
    {
        var stringId = profile.UserId.ToString();
        var user = dbContext.Users.Where(x => x.Id == stringId).FirstOrDefault();
        Console.WriteLine("Adding Profile for user:" + user.UserName);
        var serializer = new XmlSerializer(typeof(ProfileInfo));
        var stringReader = new StringReader(profile.PropertyValueStrings);
        var profileData = serializer.Deserialize(stringReader) as ProfileInfo;
        if (profileData == null)
        {
            Console.WriteLine("Profile data deserialization error for user:" + user.UserName);
        }
        else
        {
            user.Profile = profileData;
        }
    }
    dbContext.SaveChanges();
}
```

   Some of the models used are defined in the 'IdentityModels' folder of the web application project, so you must include the corresponding namespaces.

5. The above code works on the database file in the App_Data folder of the web application project created in the previous steps. To reference that, update the connection string in the app.config file of the console application with the connection string in the web.config of the web application. Also provide the complete physical path in the 'AttachDbFilename' property.

6. Open a command prompt and navigate to the bin folder of the above console application. Run the executable and review the log output as shown in the following image.



7. Open the 'AspNetUsers' table in the Server Explorer and verify the data in the new columns that hold the properties. They should be updated with the corresponding property values.

Verify functionality

Use the newly added membership pages that are implemented using ASP.NET Identity to login a user from the old database. The user should be able to log in using the same credentials. Try the other functionalities like adding OAuth, creating a new user, changing a password, adding roles, add users to roles, etc.

The Profile data for the old user and the new users should be retrieved and stored in the users table. The old table should no longer be referenced.

Conclusion

The article described the process of migrating web applications that used the provider model for membership to ASP.NET Identity. The article additionally outlined migrating profile data for users to be hooked into the Identity system. Please leave comments below for questions and issues encountered when you migrate your app.

Thanks to Rick Anderson and Robert McMurray for reviewing the article.

ASP.NET Ajax : Enhanced Interactivity and Responsiveness

2/20/2020 • 2 minutes to read • Edit Online

- AJAX Control Toolkit
- jQuery
- Microsoft Ajax CDN

Microsoft Ajax Content Delivery Network

2/20/2020 • 2 minutes to read • Edit Online

- Microsoft Ajax Content Delivery Network
- jQuery UI
  - jQuery UI 1.10.4 on the Microsoft Ajax CDN
  - jQuery UI 1.10.3 on the Microsoft Ajax CDN
  - jQuery UI 1.10.2 on the Microsoft Ajax CDN
  - jQuery UI 1.10.1 on the Microsoft Ajax CDN
  - jQuery UI 1.10.0 on the Microsoft Ajax CDN
  - jQuery UI 1.9.2 on the Microsoft Ajax CDN
  - jQuery UI 1.9.1 on the Microsoft Ajax CDN
  - jQuery UI 1.9.0 on the Microsoft Ajax CDN
  - jQuery UI 1.8.24 on the Microsoft Ajax CDN
  - jQuery UI 1.8.23 on the Microsoft Ajax CDN
  - jQuery UI 1.8.22 on the Microsoft Ajax CDN
  - jQuery UI 1.8.21 on the Microsoft Ajax CDN
  - jQuery UI 1.8.20 on the Microsoft Ajax CDN
  - jQuery UI 1.8.19 on the Microsoft Ajax CDN
  - jQuery UI 1.8.18 on the Microsoft Ajax CDN
  - jQuery UI 1.8.17 on the Microsoft Ajax CDN
  - jQuery UI 1.8.16 on the Microsoft Ajax CDN
  - jQuery UI 1.8.15 on the Microsoft Ajax CDN
  - jQuery UI 1.8.14 on the Microsoft Ajax CDN
  - jQuery UI 1.8.13 on the Microsoft Ajax CDN
  - jQuery UI 1.8.12 on the Microsoft Ajax CDN
  - jQuery UI 1.8.11 on the Microsoft Ajax CDN
  - jQuery UI 1.8.10 on the Microsoft Ajax CDN
  - jQuery UI 1.8.9 on the Microsoft Ajax CDN
  - jQuery UI 1.8.8 on the Microsoft Ajax CDN
  - jQuery UI 1.8.7 on the Microsoft Ajax CDN
  - jQuery UI 1.8.6 on the Microsoft Ajax CDN
  - jQuery UI 1.8.5
  - jQuery UI 1.11.2 on the Microsoft Ajax CDN
  - jQuery UI 1.11.1 on the Microsoft Ajax CDN
  - jQuery UI 1.11.0 on the Microsoft Ajax CDN
  - jQuery UI 1.11.3 on the Microsoft Ajax CDN
  - jQuery UI 1.11.4 on the Microsoft Ajax CDN
  - jQuery UI 1.12.0 on the Microsoft Ajax CDN
  - jQuery UI 1.12.1 on the Microsoft Ajax CDN
- jQuery Validate
  - jQuery Validation 1.17.0

- jQuery Validation 1.16.0
- jQuery Validation 1.15.1
- jQuery Validation 1.15.0
- jQuery Validation 1.14.0
- jQuery Validation 1.13.1
- jQuery Validation 1.13.0
- jQuery Validation 1.12.0
- jQuery Validation 1.11.1
- jQuery Validation 1.11.0
- jQuery Validation 1.10.0
- jquery.validate version 1.9
- jquery.validate version 1.8.1
- jquery.validate version 1.8
- jquery.validate version 1.7
- jQuery Validate 1.6
- jQuery Validate 1.5.5

- jQuery Mobile
  - jQuery Mobile 1.4.2 on the Microsoft Ajax CDN
  - jQuery Mobile 1.4.1 on the Microsoft Ajax CDN
  - jQuery Mobile 1.4.0 on the Microsoft Ajax CDN
  - jQuery Mobile 1.3.2 on the Microsoft Ajax CDN
  - jQuery Mobile 1.3.1 on the Microsoft Ajax CDN
  - jQuery Mobile 1.3.0 on the Microsoft Ajax CDN
  - jQuery Mobile 1.2.0 on the Microsoft Ajax CDN
  - jQuery Mobile 1.1.2 on the Microsoft Ajax CDN
  - jQuery Mobile 1.1.1 on the Microsoft Ajax CDN
  - jQuery Mobile 1.1.0 on the Microsoft Ajax CDN
  - jQuery Mobile 1.1.0 RC2 on the Microsoft Ajax CDN
  - jQuery Mobile 1.0.1 on the Microsoft Ajax CDN
  - jQuery Mobile 1.0 on the Microsoft Ajax CDN
  - jQuery Mobile 1.0 RC2 on the Microsoft Ajax CDN
  - jQuery Mobile 1.0 RC1 on the Microsoft Ajax CDN
  - jQuery Mobile 1.0 Beta 3 on the Microsoft Ajax CDN
  - jQuery Mobile 1.4.5 on the Microsoft Ajax CDN
- jQuery Templates
  - jQuery Templates Beta 1
- jQuery Cycle
  - jQuery Cycle 2.99
  - jQuery Cycle 2.94
  - jQuery Cycle 2.88
- jQuery DataTables
  - jQuery DataTables 1.9.4
  - jQuery DataTables 1.9.3
  - jQuery DataTables 1.9.2
  - jQuery DataTables 1.9.1

  - jQuery DataTables 1.9.0
  - jQuery DataTables 1.8.2
  - jQuery DataTables 1.10.4
  - jQuery DataTables 1.10.5
- ASP.NET Web Forms and Ajax 4
- ASP.NET Ajax 3.5
- ASP.NET Web Forms and Ajax 4.5.2
- ASP.NET Web Forms and Ajax 4.5.1

Microsoft Ajax Content Delivery Network

9/16/2020 • 14 minutes to read • Edit Online

WARNING

Production applications should not take a hard dependency on CDN assets. Applications should test for the CDN asset referenced, and use a fallback asset when the CDN is not available.

The Microsoft Ajax CDN has no SLA above and beyond using an Azure CDN.

Use this GitHub issue to report problems with the Microsoft Ajax CDN.

Table of Contents

ajax.microsoft.com renamed to ajax.aspnetcdn.com
Visual Studio .vsdoc Support
Using ASP.NET Ajax from the CDN
Using jQuery from the CDN
Using jQuery UI from the CDN
Third-Party Files on the CDN

jQuery Releases on the CDN
jQuery Migrate Releases on the CDN
jQuery UI Releases on the CDN
jQuery Validation Releases on the CDN
jQuery Mobile Releases on the CDN
jQuery Templates Releases on the CDN
jQuery Cycle Releases on the CDN
jQuery DataTables Releases on the CDN
Modernizr Releases on the CDN
JSHint Releases on the CDN
Knockout Releases on the CDN
Globalize Releases on the CDN
Respond Releases on the CDN
Bootstrap Releases on the CDN
Bootstrap TouchCarousel Releases on the CDN
Hammer.js Releases on the CDN
ASP.NET Web Forms and Ajax Releases on the CDN
ASP.NET MVC Releases on the CDN
ASP.NET SignalR Releases on the CDN

The Microsoft Ajax Content Delivery Network (CDN) hosts popular third party JavaScript libraries such as jQuery and enables you to easily add them to your Web applications. For example, you can start using jQuery which is hosted on this CDN simply by adding a <script> tag to your page that points to ajax.aspnetcdn.com.

By taking advantage of the CDN, you can significantly improve the performance of your Ajax applications. The contents of the CDN are cached on servers located around the world. In addition, the CDN enables browsers to reuse cached third party JavaScript files for web sites that are located in different domains.

The CDN supports SSL (HTTPS) in case you need to serve a web page using the Secure Sockets Layer.

The CDN hosts the following third party script libraries which have been uploaded, and are licensed to you, by the owners of those libraries:

- jQuery (www.jquery.com)
- jQuery UI (www.jqueryui.com)
- jQuery Mobile (www.jquerymobile.com)
- jQuery Validation (https://jqueryvalidation.org/)
- jQuery Cycle (www.malsup.com/jquery/cycle/)
- jQuery DataTables (http://datatables.net/)

The Microsoft Ajax CDN also includes the following libraries which have been uploaded by Microsoft:

- ASP.NET Ajax
- ASP.NET MVC JavaScript Files
- ASP.NET SignalR JavaScript Files

Microsoft does not claim ownership of any third-party libraries hosted on this CDN. The copyright owners of the libraries are licensing these libraries to you. Any rights that you may have to download and use such libraries are granted solely by the respective copyright owners. Because these are not Microsoft libraries, Microsoft provides no warranties or intellectual property rights licenses (including no implied patent rights) for the third party libraries hosted on this CDN.

If you wish to submit your JavaScript library and your library is one of the top JavaScript libraries (as listed on http://trends.builtwith.com) or extensions/plugins to these libraries that are (a) popular; or (b) helpful for use on ASP.NET then please contact AjaxCDNSubmission@Microsoft.com.

ajax.microsoft.com renamed to ajax.aspnetcdn.com

The CDN used to use the microsoft.com domain name and has been changed to use the aspnetcdn.com domain name. This change was made to increase performance because when a browser referenced the microsoft.com domain it would send any cookies from that domain across the wire with each request. By renaming to a domain name other than microsoft.com performance can be increased by as much to 25%. Note ajax.microsoft.com will continue to function but ajax.aspnetcdn.com is recommended.

- Old Format: https://ajax.microsoft.com/ajax/jQuery/jquery-1.8.0.js
- New Format: https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.0.js

Visual Studio .vsdoc Support

To use the .vsdoc files properly with Visual Studio 2008 you need to make sure that you have VS 2008 SP1 installed and the hotfix for vsdoc files installed. You can get these from here:

- Download Visual Studio 2008 SP1
- Download .vsdoc hotfix for Visual Studio 2008 SP1

Visual Studio 2010 supports .vsdoc files without any additional patches.

Using ASP.NET Ajax from the CDN

When using ASP.NET 4, you can redirect all requests for ASP.NET framework scripts to the CDN. Retrieving scripts from the CDN instead of your local web server can substantially improve the performance of public ASP.NET websites.

Use the ScriptManager EnableCDN property to redirect all ASP.NET framework script requests to the Microsoft Ajax CDN:

```
<asp:ScriptManager
    ID="ScriptManager1"
    EnableCdn="true"
    Runat="Server" />
```

Using jQuery from the CDN

You can use jQuery scripts hosted on CDN in your Web application by adding the following script element to a page:

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-1.9.0.js"></script>
```

The CDN also includes the minified version of the jQuery script, which you can get using the following element:

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-1.9.0.min.js"></script>
```

To allow your page to fallback to loading jQuery from a local path on your own website if the CDN happens to be unavailable, add the following element immediately after the element referencing the CDN:

```
<script>
  // Fallback to loading jQuery from a local path if the CDN is unavailable
  (window.jQuery || document.write('<script src="/scripts/jquery-1.9.0.min.js"><\/script>'));
</script>
```

The following sample page uses the CDN version of the jQuery library (with fallback to a local copy) to display the contents of a div element when a button is clicked.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>jQuery from Microsoft AJAX CDN</title>
</head>
<body>
    <button id="btn">Show Message</button>

    <div id="message" style="display:none">
        <h1>Hello from jQuery!</h1>
    </div>

    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-1.9.0.min.js"></script>
    <script>
        // Fallback to loading jQuery from a local path if the CDN is unavailable
        (window.jQuery || document.write('<script src="/scripts/jquery-1.9.0.min.js"><\/script>'));
    </script>
    <script>
        function domReady() {
            $('#btn').click( showMessage );
        }

        function showMessage() {
            $('#message').fadeIn('slow');
        }

        $( domReady );
    </script>
</body>
</html>
```

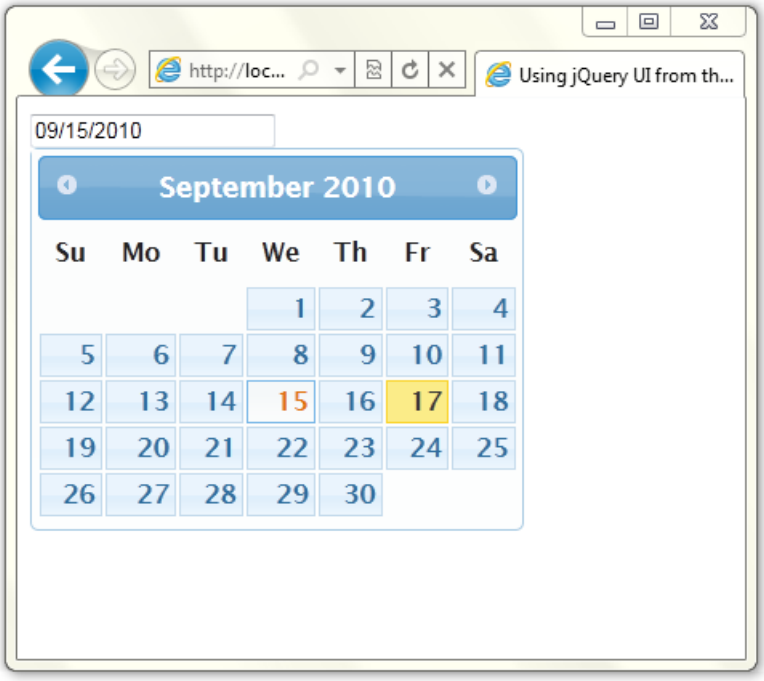
You can learn more about jQuery and download a local copy of jQuery by visiting the jQuery Web site.

Using jQuery UI from the CDN

The CDN also hosts the jQuery UI library. The jQuery UI library includes a rich set of widgets and effects that you can use in your ASP.NET applications. For example, the following page illustrates how you can use the jQuery UI Datepicker in the context of an ASP.NET Web Forms application to display a pop-up calendar:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="WebForm1.aspx.cs"
Inherits="TestjQueryUICDN.WebForm1" %>
<!DOCTYPE html >
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Using jQuery UI from the CDN</title>
    <link rel="Stylesheet" href="https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/redmond/jquery-
ui.css" />
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:TextBox ID="txtStartDate" ClientIDMode="Static" runat="server" />
    </div>
    </form>
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-1.8.0.js"></script>
    <script src="https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/jquery-ui.js"></script>
    <script>
        $("#txtStartDate").datepicker();
    </script>
</body>
</html>
```

When you move focus to the TextBox using your keyboard, a calendar is displayed:



Notice that you must include three files from the CDN in the code above:

- The jQuery library — The jQuery UI library depends on the jQuery library. You must add the jQuery library to your page before you add the jQuery UI library.
- The jQuery UI library — The jQuery UI library contains all of the jQuery UI effects and widgets such as the

Datepicker widget used in the page above.

- A jQuery UI theme — The jQuery UI supports different themes. The page above includes a link to a CSS file to import the Redmond theme.

All of the standard jQuery UI themes are hosted on the CDN. Visit this page to view thumbnails for each theme.

To learn more about the jQuery UI library, visit the official jQuery UI website.

Third-Party Files on the CDN

The CDN hosts some of the most popular third party JavaScript libraries. Microsoft does not claim ownership of any third-party libraries hosted on this CDN. The copyright owners of the libraries are licensing these libraries to you. Any rights that you may have to download and use such libraries are granted solely by the respective copyright owners. Because these are not Microsoft libraries, Microsoft provides no warranties or intellectual property rights licenses (including no implied patent rights) for the third party libraries hosted on this CDN.

jQuery Releases on the CDN

The following releases of jQuery are hosted on the CDN:

jQuery version 3.5.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.5.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.5.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.5.1.min.map
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.5.1.slim.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.5.1.slim.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.5.1.slim.min.map

jQuery version 3.5.0

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.5.0.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.5.0.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.5.0.min.map
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.5.0.slim.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.5.0.slim.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.5.0.slim.min.map

jQuery version 3.4.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.1.min.map
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.1.slim.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.1.slim.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.1.slim.min.map

jQuery version 3.4.0

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.0.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.0.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.0.min.map
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.0.slim.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.0.slim.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.0.slim.min.map

jQuery version 3.3.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.3.1.js

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.3.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.3.1.min.map
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.3.1.slim.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.3.1.slim.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.3.1.slim.min.map

jQuery version 3.2.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.min.map
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.slim.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.slim.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.1.slim.min.map

jQuery version 3.2.0

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.0.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.0.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.0.min.map
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.0.slim.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.0.slim.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.2.0.slim.min.map

jQuery version 3.1.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.1.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.1.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.1.1.min.map
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.1.1.slim.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.1.1.slim.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.1.1.slim.min.map

jQuery version 3.1.0

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.1.0.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.1.0.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.1.0.min.map
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.1.0.slim.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.1.0.slim.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.1.0.slim.min.map

jQuery version 3.0.0

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.0.0.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.0.0.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.0.0.min.map
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.0.0.slim.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.0.0.slim.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.0.0.slim.min.map

jQuery version 2.2.4

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.4.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.4.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.4.min.map

jQuery version 2.2.3

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.3.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.3.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.3.min.map

jQuery version 2.2.2

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.2.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.2.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.2.min.map

jQuery version 2.2.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.1.min.map

jQuery version 2.2.0

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.0.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.0.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.2.0.min.map

jQuery version 2.1.4

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.4.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.4.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.4.min.map

jQuery version 2.1.3

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.3.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.3.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.3.min.map

jQuery version 2.1.2

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.2.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.2.min.js

jQuery version 2.1.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.1.min.map

jQuery version 2.1.0

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.0.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.0.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.0-vsdoc.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.1.0.min.map

jQuery version 2.0.3

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.3.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.3.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.3-vsdoc.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.3.min.map

jQuery version 2.0.2

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.2.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.2.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.2-vsdoc.js

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.2.min.map

jQuery version 2.0.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.1-vsdoc.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.1.min.map

jQuery version 2.0.0

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.0.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.0.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.0-vsdoc.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.0.min.map

jQuery version 1.12.4

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.4.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.4.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.4.min.map

jQuery version 1.12.3

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.3.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.3.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.3.min.map

jQuery version 1.12.2

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.2.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.2.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.2.min.map

jQuery version 1.12.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.1.min.map

jQuery version 1.12.0

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.0.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.0.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.12.0.min.map

jQuery version 1.11.3

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.3.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.3.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.3.min.map

jQuery version 1.11.2

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.2.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.2.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.2.min.map

jQuery version 1.11.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.1.min.map

jQuery version 1.11.0

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.0.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.0.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.0-vsdoc.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.0.min.map

jQuery version 1.10.2

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.10.2.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.10.2.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.10.2-vsdoc.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.10.2.min.map

jQuery version 1.10.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.10.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.10.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.10.1-vsdoc.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.10.1.min.map

jQuery version 1.10.0

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.10.0.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.10.0.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.10.0-vsdoc.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.10.0.min.map

jQuery version 1.9.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.9.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.9.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.9.1-vsdoc.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.9.1.min.map

jQuery version 1.9.0

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.9.0.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.9.0.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.9.0-vsdoc.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.9.0.min.map

jQuery version 1.8.3

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.3.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.3.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.3-vsdoc.js

jQuery version 1.8.2

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.2.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.2.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.2-vsdoc.js

jQuery version 1.8.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.1-vsdoc.js

jQuery version 1.8.0

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.0.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.0.min.js

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.8.0-vsdoc.js

jQuery version 1.7.2

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.7.2.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.7.2.min.js

jQuery version 1.7.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.7.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.7.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.7.1-vsdoc.js

jQuery version 1.7

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.7.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.7.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.7-vsdoc.js

jQuery version 1.6.4

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.4.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.4.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.4-vsdoc.js

jQuery version 1.6.3

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.3.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.3.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.3-vsdoc.js

jQuery version 1.6.2

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.2.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.2.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.2-vsdoc.js

jQuery version 1.6.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.1-vsdoc.js

jQuery version 1.6

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6-vsdoc.js

jQuery version 1.5.2

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.5.2.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.5.2.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.5.2-vsdoc.js

jQuery version 1.5.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.5.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.5.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.5.1-vsdoc.js

jQuery version 1.5

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.5.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.5.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.5-vsdoc.js

jQuery version 1.4.4

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.4.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.4.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.4-vsdoc.js

jQuery version 1.4.3

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.3.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.3.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.3-vsdoc.js

jQuery version 1.4.2

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.2.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.2.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.2-vsdoc.js

jQuery version 1.4.1

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.1.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.1.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.1-vsdoc.js

jQuery version 1.4

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.4.min.js

jQuery version 1.3.2

- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.3.2.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.3.2.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.3.2-vsdoc.js
- https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.3.2.min-vsdoc.js

jQuery Migrate Releases on the CDN

The following releases of jQuery Migrate are hosted on the CDN:

jQuery Migrate version 3.0.0

- https://ajax.aspnetcdn.com/ajax/jquery.migrate/jquery-migrate-3.0.0.js
- https://ajax.aspnetcdn.com/ajax/jquery.migrate/jquery-migrate-3.0.0.min.js

jQuery Migrate version 1.2.1

- https://ajax.aspnetcdn.com/ajax/jquery.migrate/jquery-migrate-1.2.1.js
- https://ajax.aspnetcdn.com/ajax/jquery.migrate/jquery-migrate-1.2.1.min.js

jQuery Migrate version 1.2.0

- https://ajax.aspnetcdn.com/ajax/jquery.migrate/jquery-migrate-1.2.0.js
- https://ajax.aspnetcdn.com/ajax/jquery.migrate/jquery-migrate-1.2.0.min.js

jQuery Migrate version 1.1.1

- https://ajax.aspnetcdn.com/ajax/jquery.migrate/jquery-migrate-1.1.1.js
- https://ajax.aspnetcdn.com/ajax/jquery.migrate/jquery-migrate-1.1.1.min.js

jQuery Migrate version 1.1.0

- https://ajax.aspnetcdn.com/ajax/jquery.migrate/jquery-migrate-1.1.0.js
- https://ajax.aspnetcdn.com/ajax/jquery.migrate/jquery-migrate-1.1.0.min.js

jQuery Migrate version 1.0.0

- https://ajax.aspnetcdn.com/ajax/jquery.migrate/jquery-migrate-1.0.0.js

- https://ajax.aspnetcdn.com/ajax/jquery.migrate/jquery-migrate-1.0.0.min.js

jQuery UI Releases on the CDN

The following releases of the jQuery UI library are hosted on this CDN. Click each link to see the actual list of files.

- jQuery UI 1.12.1
- jQuery UI 1.12.0
- jQuery UI 1.11.4
- jQuery UI 1.11.3
- jQuery UI 1.11.2
- jQuery UI 1.11.1
- jQuery UI 1.11.0
- jQuery UI 1.10.4
- jQuery UI 1.10.3
- jQuery UI 1.10.2
- jQuery UI 1.10.1
- jQuery UI 1.10.0
- jQuery UI 1.9.2
- jQuery UI 1.9.1
- jQuery UI 1.9.0
- jQuery UI 1.8.24
- jQuery UI 1.8.23
- jQuery UI 1.8.22
- jQuery UI 1.8.21
- jQuery UI 1.8.20
- jQuery UI 1.8.19
- jQuery UI 1.8.18
- jQuery UI 1.8.17
- jQuery UI 1.8.16
- jQuery UI 1.8.15
- jQuery UI 1.8.14
- jQuery UI 1.8.13
- jQuery UI 1.8.12
- jQuery UI 1.8.11
- jQuery UI 1.8.10
- jQuery UI 1.8.9
- jQuery UI 1.8.8
- jQuery UI 1.8.7
- jQuery UI 1.8.6
- jQuery UI 1.8.5

jQuery Validation Releases on the CDN

The following releases of the jQuery Validation plugin are hosted on this CDN. Click each link to see the actual list of files.

- jQuery Validate 1.19.2
- jQuery Validate 1.19.1
- jQuery Validate 1.19.0
- jQuery Validate 1.17.0

- jQuery Validate 1.16.0
- jQuery Validate 1.15.1
- jQuery Validate 1.15.0
- jQuery Validate 1.14.0
- jQuery Validate 1.13.1
- jQuery Validate 1.13.0
- jQuery Validate 1.12.0
- jQuery Validate 1.11.1
- jQuery Validate 1.11.0
- jQuery Validate 1.10.0
- jQuery Validate 1.9
- jQuery Validate 1.8.1
- jQuery Validate 1.8
- jQuery Validate 1.7
- jQuery Validate 1.6
- jQuery Validate 1.5.5

jQuery Mobile Releases on the CDN

The following releases of the jQuery Mobile library are hosted on this CDN. Click each link to see the actual list of files.

- jQuery Mobile 1.4.5
- jQuery Mobile 1.4.2
- jQuery Mobile 1.4.1
- jQuery Mobile 1.4.0
- jQuery Mobile 1.3.2
- jQuery Mobile 1.3.1
- jQuery Mobile 1.3.0
- jQuery Mobile 1.2.0
- jQuery Mobile 1.1.2
- jQuery Mobile 1.1.1
- jQuery Mobile 1.1.0
- jQuery Mobile 1.1.0 RC 2
- jQuery Mobile 1.0.1
- jQuery Mobile 1.0
- jQuery Mobile 1.0 RC 2
- jQuery Mobile 1.0 RC 1
- jQuery Mobile 1.0 beta 3

jQuery Templates Releases on the CDN

The following releases of the jQuery Templates plugin are hosted on this CDN. Click each link to see the actual list of files.

- jQuery Templates Beta 1

jQuery Cycle Releases on the CDN

The following releases of the jQuery Cycle plugin are hosted on this CDN. Click each link to see the actual list of files.

- jQuery Cycle 2.99

- jQuery Cycle 2.94
- jQuery Cycle 2.88

jQuery DataTables Releases on the CDN

The following releases of the jQuery DataTables plugin are hosted on this CDN. Click each link to see the actual list of files.

- jQuery DataTables 1.10.5
- jQuery DataTables 1.10.4
- jQuery DataTables 1.9.4
- jQuery DataTables 1.9.3
- jQuery DataTables 1.9.2
- jQuery DataTables 1.9.1
- jQuery DataTables 1.9.0
- jQuery DataTables 1.8.2

Modernizr Releases on the CDN

The following releases of Modernizr are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/modernizr/modernizr-3.5.0.js
- https://ajax.aspnetcdn.com/ajax/modernizr/modernizr-2.8.3.js
- https://ajax.aspnetcdn.com/ajax/modernizr/modernizr-2.7.2.js
- https://ajax.aspnetcdn.com/ajax/modernizr/modernizr-2.7.1.js
- https://ajax.aspnetcdn.com/ajax/modernizr/modernizr-2.6.2.js
- https://ajax.aspnetcdn.com/ajax/modernizr/modernizr-1.7-development-only.js
- https://ajax.aspnetcdn.com/ajax/modernizr/modernizr-2.0.6-development-only.js

JSHint Releases on the CDN

The following releases of JSHint are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jshint/r07/jshint.js

Knockout Releases on the CDN

The following releases of Knockout are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/knockout/knockout-2.2.1.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-2.2.1.debug.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-2.2.0.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-2.2.0.debug.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-2.1.0.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-2.1.0.debug.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.0.0.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.0.0.debug.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.1.0.debug.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.2.0.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.2.0.debug.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.debug.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.4.0.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.4.0.debug.js

- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.4.1.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.4.1.debug.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.4.2.js
- https://ajax.aspnetcdn.com/ajax/knockout/knockout-3.4.2.debug.js

Globalize Releases on the CDN

The following releases of Globalize are hosted on the CDN:

Globalize version 1.0.0

- https://ajax.aspnetcdn.com/ajax/globalize/1.0.0/globalize.js
- https://ajax.aspnetcdn.com/ajax/globalize/1.0.0/node-main.js
- https://ajax.aspnetcdn.com/ajax/globalize/1.0.0/globalize/currency.js
- https://ajax.aspnetcdn.com/ajax/globalize/1.0.0/globalize/date.js
- https://ajax.aspnetcdn.com/ajax/globalize/1.0.0/globalize/message.js
- https://ajax.aspnetcdn.com/ajax/globalize/1.0.0/globalize/number.js
- https://ajax.aspnetcdn.com/ajax/globalize/1.0.0/globalize/plural.js
- https://ajax.aspnetcdn.com/ajax/globalize/1.0.0/globalize/relative-time.js

Globalize version 0.1.1

- https://ajax.aspnetcdn.com/ajax/globalize/0.1.1/globalize.min.js
- https://ajax.aspnetcdn.com/ajax/globalize/0.1.1/globalize.js
- https://ajax.aspnetcdn.com/ajax/globalize/0.1.1/cultures/globalize.cultures.js
  - all cultures
- https://ajax.aspnetcdn.com/ajax/globalize/0.1.1/cultures/globalize.culture.{culture-code}.js
  - Replace "{culture-code}" with the desired culture code, e.g. globalize.culture.en-GB.js== Microsoft Files on the CDN ==These libraries were uploaded by Microsoft.

Respond Releases on the CDN

The following releases of Respond are hosted on the CDN:

Respond version 1.4.2

- https://ajax.aspnetcdn.com/ajax/respond/1.4.2/respond.js
- https://ajax.aspnetcdn.com/ajax/respond/1.4.2/respond.min.js
- https://ajax.aspnetcdn.com/ajax/respond/1.4.2/respond.matchmedia.addListener.js
- https://ajax.aspnetcdn.com/ajax/respond/1.4.2/respond.matchmedia.addListener.min.js

Respond version 1.4.1

- https://ajax.aspnetcdn.com/ajax/respond/1.4.1/respond.js
- https://ajax.aspnetcdn.com/ajax/respond/1.4.1/respond.min.js
- https://ajax.aspnetcdn.com/ajax/respond/1.4.1/respond.matchmedia.addListener.js
- https://ajax.aspnetcdn.com/ajax/respond/1.4.1/respond.matchmedia.addListener.min.js

Respond version 1.4.0

- https://ajax.aspnetcdn.com/ajax/respond/1.4.0/respond.js
- https://ajax.aspnetcdn.com/ajax/respond/1.4.0/respond.min.js
- https://ajax.aspnetcdn.com/ajax/respond/1.4.0/respond.matchmedia.addListener.js
- https://ajax.aspnetcdn.com/ajax/respond/1.4.0/respond.matchmedia.addListener.min.js

Respond version 1.3.0

- https://ajax.aspnetcdn.com/ajax/respond/1.3.0/respond.js

Respond version 1.2.0

- https://ajax.aspnetcdn.com/ajax/respond/1.2.0/respond.js

Bootstrap Releases on the CDN

The following releases of getbootstrap.com bootstrap are hosted on the CDN:

Bootstrap version 4.5.2

- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.2/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.2/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.2/bootstrap.bundle.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.2/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.2/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.2/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.2/css/bootstrap-grid.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.2/css/bootstrap-grid.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.2/css/bootstrap-grid.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.2/css/bootstrap-reboot.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.2/css/bootstrap-reboot.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.2/css/bootstrap-reboot.css.map

Bootstrap version 4.5.0

- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.0/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.0/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.0/bootstrap.bundle.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.0/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.0/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.0/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.0/css/bootstrap-grid.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.0/css/bootstrap-grid.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.0/css/bootstrap-grid.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.0/css/bootstrap-reboot.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.0/css/bootstrap-reboot.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.5.0/css/bootstrap-reboot.css.map

Bootstrap version 4.4.1

- https://ajax.aspnetcdn.com/ajax/bootstrap/4.4.1/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.4.1/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.4.1/bootstrap.bundle.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.4.1/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.4.1/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.4.1/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.4.1/css/bootstrap-grid.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.4.1/css/bootstrap-grid.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.4.1/css/bootstrap-grid.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.4.1/css/bootstrap-reboot.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.4.1/css/bootstrap-reboot.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.4.1/css/bootstrap-reboot.css.map

Bootstrap version 4.3.1

- https://ajax.aspnetcdn.com/ajax/bootstrap/4.3.1/bootstrap.js

- https://ajax.aspnetcdn.com/ajax/bootstrap/4.3.1/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.3.1/bootstrap.bundle.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.3.1/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.3.1/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.3.1/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.3.1/css/bootstrap-grid.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.3.1/css/bootstrap-grid.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.3.1/css/bootstrap-grid.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.3.1/css/bootstrap-reboot.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.3.1/css/bootstrap-reboot.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.3.1/css/bootstrap-reboot.css.map

Bootstrap version 4.2.1

- https://ajax.aspnetcdn.com/ajax/bootstrap/4.2.1/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.2.1/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.2.1/bootstrap.bundle.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.2.1/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.2.1/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.2.1/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.2.1/css/bootstrap-grid.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.2.1/css/bootstrap-grid.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.2.1/css/bootstrap-grid.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.2.1/css/bootstrap-reboot.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.2.1/css/bootstrap-reboot.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.2.1/css/bootstrap-reboot.css.map

Bootstrap version 4.1.1

- https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/bootstrap.bundle.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/css/bootstrap-grid.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/css/bootstrap-grid.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/css/bootstrap-grid.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/css/bootstrap-reboot.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/css/bootstrap-reboot.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/css/bootstrap-reboot.css.map

Bootstrap version 4.0.0

- https://ajax.aspnetcdn.com/ajax/bootstrap/4.0.0/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.0.0/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.0.0/bootstrap.bundle.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.0.0/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.0.0/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.0.0/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.0.0/css/bootstrap-grid.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.0.0/css/bootstrap-grid.min.css

- https://ajax.aspnetcdn.com/ajax/bootstrap/4.0.0/css/bootstrap-grid.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.0.0/css/bootstrap-reboot.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.0.0/css/bootstrap-reboot.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/4.0.0/css/bootstrap-reboot.css.map

Bootstrap version 3.4.1

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.1/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.1/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.1/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.1/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.1/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.1/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.1/css/bootstrap-theme.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.1/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.1/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.1/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.1/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.1/fonts/glyphicons-halflings-regular.woff
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.1/fonts/glyphicons-halflings-regular.woff2

Bootstrap version 3.4.0

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.0/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.0/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.0/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.0/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.0/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.0/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.0/css/bootstrap-theme.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.0/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.0/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.0/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.0/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.0/fonts/glyphicons-halflings-regular.woff
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.4.0/fonts/glyphicons-halflings-regular.woff2

Bootstrap version 3.3.7

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap-theme.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/fonts/glyphicons-halflings-regular.woff
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/fonts/glyphicons-halflings-regular.woff2

Bootstrap version 3.3.6

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap-theme.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/fonts/glyphicons-halflings-regular.woff
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/fonts/glyphicons-halflings-regular.woff2

Bootstrap version 3.3.5

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/css/bootstrap-theme.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/fonts/glyphicons-halflings-regular.woff
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/fonts/glyphicons-halflings-regular.woff2

Bootstrap version 3.3.4

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.4/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.4/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.4/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.4/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.4/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.4/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.4/css/bootstrap-theme.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.4/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.4/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.4/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.4/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.4/fonts/glyphicons-halflings-regular.woff
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.4/fonts/glyphicons-halflings-regular.woff2

Bootstrap version 3.3.2

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/css/bootstrap.css

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/css/bootstrap-theme.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/fonts/glyphicons-halflings-regular.woff
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.2/fonts/glyphicons-halflings-regular.woff2

Bootstrap version 3.3.1

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.1/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.1/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.1/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.1/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.1/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.1/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.1/css/bootstrap-theme.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.1/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.1/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.1/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.1/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.1/fonts/glyphicons-halflings-regular.woff

Bootstrap version 3.3.0

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.0/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.0/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.0/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.0/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.0/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.0/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.0/css/bootstrap-theme.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.0/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.0/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.0/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.0/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.0/fonts/glyphicons-halflings-regular.woff

Bootstrap version 3.2.0

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/css/bootstrap-theme.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/fonts/glyphicons-halflings-regular.eot

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.2.0/fonts/glyphicons-halflings-regular.woff

Bootstrap version 3.1.1

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.1/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.1/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.1/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.1/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.1/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.1/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.1/css/bootstrap-theme.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.1/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.1/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.1/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.1/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.1/fonts/glyphicons-halflings-regular.woff

Bootstrap version 3.1.0

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.0/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.0/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.0/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.0/css/bootstrap.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.0/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.0/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.0/css/bootstrap-theme.css.map
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.0/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.0/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.0/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.0/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.1.0/fonts/glyphicons-halflings-regular.woff

Bootstrap version 3.0.3

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.3/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.3/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.3/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.3/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.3/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.3/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.3/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.3/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.3/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.3/fonts/glyphicons-halflings-regular.woff

Bootstrap version 3.0.2

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.2/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.2/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.2/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.2/css/bootstrap.min.css

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.2/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.2/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.2/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.2/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.2/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.2/fonts/glyphicons-halflings-regular.woff

Bootstrap version 3.0.1

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.1/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.1/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.1/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.1/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.1/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.1/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.1/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.1/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.1/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.1/fonts/glyphicons-halflings-regular.woff

Bootstrap version 3.0.0

- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/css/bootstrap-theme.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/css/bootstrap-theme.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/fonts/glyphicons-halflings-regular.eot
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/fonts/glyphicons-halflings-regular.svg
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/fonts/glyphicons-halflings-regular.ttf
- https://ajax.aspnetcdn.com/ajax/bootstrap/3.0.0/fonts/glyphicons-halflings-regular.woff

Bootstrap version 2.3.2

- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.2/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.2/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.2/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.2/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.2/css/bootstrap-responsive.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.2/css/bootstrap-responsive.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.2/img/glyphicons-halflings.png
- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.2/img/glyphicons-halflings-white.png

Bootstrap version 2.3.1

- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.1/bootstrap.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.1/bootstrap.min.js
- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.1/css/bootstrap.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.1/css/bootstrap.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.1/css/bootstrap-responsive.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.1/css/bootstrap-responsive.min.css
- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.1/img/glyphicons-halflings.png

- https://ajax.aspnetcdn.com/ajax/bootstrap/2.3.1/img/glyphicons-halflings-white.png

Bootstrap TouchCarousel Releases on the CDN

The following releases of https://github.com/ixisio/bootstrap-touch-carousel Bootstrap TouchCarousel releases are hosted on the CDN:

Bootstrap TouchCarousel version 0.8.0

- https://ajax.aspnetcdn.com/ajax/bootstrap-touch-carousel/0.8.0/css/bootstrap-touch-carousel.css
- https://ajax.aspnetcdn.com/ajax/bootstrap-touch-carousel/0.8.0/js/bootstrap-touch-carousel.js

Hammer.js Releases on the CDN

The following releases of http://hammerjs.github.io/ Hammer.js releases are hosted on the CDN:

Hammer.js version 2.0.4

- https://ajax.aspnetcdn.com/ajax/hammer.js/2.0.4/hammer.js
- https://ajax.aspnetcdn.com/ajax/hammer.js/2.0.4/hammer.min.js
- https://ajax.aspnetcdn.com/ajax/hammer.js/2.0.4/hammer.min.map

ASP.NET Web Forms and Ajax Releases on the CDN

The following releases of the ASP.NET Ajax Library are hosted on the CDN. Click each link to see the actual list of files.

- ASP.NET Web Forms and Ajax version 4.5.2
- ASP.NET Web Forms and Ajax version 4
- ASP.NET Ajax version 3.5

ASP.NET MVC Releases on the CDN

The following ASP.NET MVC JavaScript files are hosted on this CDN:

ASP.NET MVC 5.2.3

- https://ajax.aspnetcdn.com/ajax/mvc/5.2.3/jquery.validate.unobtrusive.js
- https://ajax.aspnetcdn.com/ajax/mvc/5.2.3/jquery.validate.unobtrusive.min.js

ASP.NET MVC 5.1

- https://ajax.aspnetcdn.com/ajax/mvc/5.1/jquery.validate.unobtrusive.js
- https://ajax.aspnetcdn.com/ajax/mvc/5.1/jquery.validate.unobtrusive.min.js

ASP.NET MVC 5.0

- https://ajax.aspnetcdn.com/ajax/mvc/5.0/jquery.validate.unobtrusive.js
- https://ajax.aspnetcdn.com/ajax/mvc/5.0/jquery.validate.unobtrusive.min.js

ASP.NET MVC 4.0

- https://ajax.aspnetcdn.com/ajax/mvc/4.0/jquery.validate.unobtrusive.js
- https://ajax.aspnetcdn.com/ajax/mvc/4.0/jquery.validate.unobtrusive.min.js

ASP.NET MVC 3.0

- https://ajax.aspnetcdn.com/ajax/mvc/3.0/jquery.unobtrusive-ajax.js
- https://ajax.aspnetcdn.com/ajax/mvc/3.0/jquery.unobtrusive-ajax.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.unobtrusive-ajax/3.2.5/jquery.unobtrusive-ajax.js
- https://ajax.aspnetcdn.com/ajax/jquery.unobtrusive-ajax/3.2.5/jquery.unobtrusive-ajax.min.js
- https://ajax.aspnetcdn.com/ajax/mvc/3.0/jquery.validate.unobtrusive.js
- https://ajax.aspnetcdn.com/ajax/mvc/3.0/jquery.validate.unobtrusive.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validation.unobtrusive/3.2.10/jquery.validate.unobtrusive.js
- https://ajax.aspnetcdn.com/ajax/jquery.validation.unobtrusive/3.2.10/jquery.validate.unobtrusive.min.js
- https://ajax.aspnetcdn.com/ajax/mvc/3.0/MicrosoftMvcAjax.js

- https://ajax.aspnetcdn.com/ajax/mvc/3.0/MicrosoftMvcAjax.debug.js

ASP.NET MVC 2.0

- https://ajax.aspnetcdn.com/ajax/mvc/2.0/MicrosoftMvcAjax.js
- https://ajax.aspnetcdn.com/ajax/mvc/2.0/MicrosoftMvcAjax.debug.js

ASP.NET MVC 1.0

- https://ajax.aspnetcdn.com/ajax/mvc/1.0/MicrosoftMvcAjax.js
- https://ajax.aspnetcdn.com/ajax/mvc/1.0/MicrosoftMvcAjax.debug.js

ASP.NET SignalR Releases on the CDN

The following ASP.NET SignalR JavaScript files are hosted on this CDN:

ASP.NET SignalR 2.2.2

- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.2.2.min.js
- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.2.2.js

ASP.NET SignalR 2.2.1

- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.2.1.min.js
- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.2.1.js

ASP.NET SignalR 2.2.0

- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.2.0.min.js
- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.2.0.js

ASP.NET SignalR 2.1.0

- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.1.0.min.js
- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.1.0.js

ASP.NET SignalR 2.0.3

- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.0.3.min.js
- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.0.3.js

ASP.NET SignalR 2.0.2

- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.0.2.min.js
- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.0.2.js

ASP.NET SignalR 2.0.1

- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.0.1.min.js
- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.0.1.js

ASP.NET SignalR 2.0.0

- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.0.0.min.js
- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-2.0.0.js

ASP.NET SignalR 1.1.3

- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-1.1.3.min.js
- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-1.1.3.js

ASP.NET SignalR 1.1.2

- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-1.1.2.min.js
- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-1.1.2.js

ASP.NET SignalR 1.1.1

- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-1.1.1.min.js
- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-1.1.1.js

ASP.NET SignalR 1.1.0

- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-1.1.0.min.js
- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-1.1.0.js

ASP.NET SignalR 1.0.1

- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-1.0.1.min.js
- https://ajax.aspnetcdn.com/ajax/signalr/jquery.signalr-1.0.1.js

For information about the terms of use for the CDN, see Microsoft Ajax CDN Terms of Use.

jQuery UI

2/20/2020 • 2 minutes to read • Edit Online

- jQuery UI 1.10.4 on the Microsoft Ajax CDN
- jQuery UI 1.10.3 on the Microsoft Ajax CDN
- jQuery UI 1.10.2 on the Microsoft Ajax CDN
- jQuery UI 1.10.1 on the Microsoft Ajax CDN
- jQuery UI 1.10.0 on the Microsoft Ajax CDN
- jQuery UI 1.9.2 on the Microsoft Ajax CDN
- jQuery UI 1.9.1 on the Microsoft Ajax CDN
- jQuery UI 1.9.0 on the Microsoft Ajax CDN
- jQuery UI 1.8.24 on the Microsoft Ajax CDN
- jQuery UI 1.8.23 on the Microsoft Ajax CDN
- jQuery UI 1.8.22 on the Microsoft Ajax CDN
- jQuery UI 1.8.21 on the Microsoft Ajax CDN
- jQuery UI 1.8.20 on the Microsoft Ajax CDN
- jQuery UI 1.8.19 on the Microsoft Ajax CDN
- jQuery UI 1.8.18 on the Microsoft Ajax CDN
- jQuery UI 1.8.17 on the Microsoft Ajax CDN
- jQuery UI 1.8.16 on the Microsoft Ajax CDN
- jQuery UI 1.8.15 on the Microsoft Ajax CDN
- jQuery UI 1.8.14 on the Microsoft Ajax CDN
- jQuery UI 1.8.13 on the Microsoft Ajax CDN
- jQuery UI 1.8.12 on the Microsoft Ajax CDN
- jQuery UI 1.8.11 on the Microsoft Ajax CDN
- jQuery UI 1.8.10 on the Microsoft Ajax CDN
- jQuery UI 1.8.9 on the Microsoft Ajax CDN
- jQuery UI 1.8.8 on the Microsoft Ajax CDN
- jQuery UI 1.8.7 on the Microsoft Ajax CDN
- jQuery UI 1.8.6 on the Microsoft Ajax CDN
- jQuery UI 1.8.5
- jQuery UI 1.11.2 on the Microsoft Ajax CDN
- jQuery UI 1.11.1 on the Microsoft Ajax CDN
- jQuery UI 1.11.0 on the Microsoft Ajax CDN
- jQuery UI 1.11.3 on the Microsoft Ajax CDN
- jQuery UI 1.11.4 on the Microsoft Ajax CDN
- jQuery UI 1.12.0 on the Microsoft Ajax CDN
- jQuery UI 1.12.1 on the Microsoft Ajax CDN

jQuery UI 1.10.4 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.10.4 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/jquery-ui.min.js

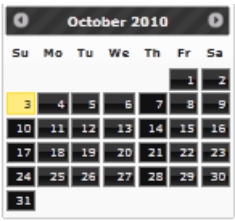
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/black-tie/jquery-ui.css

Blitzer

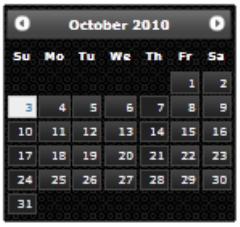
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/cupertino/jquery-ui.css

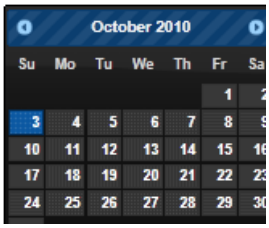
---

Dark-Hive



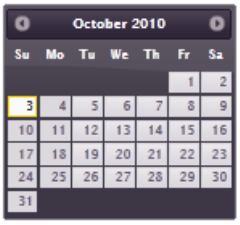
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/dot-luv/jquery-ui.css

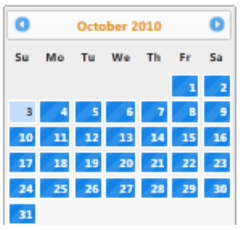
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/eggplant/jquery-ui.css

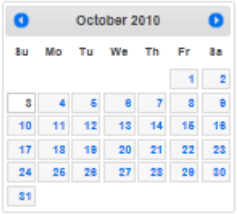
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/excite-bike/jquery-ui.css

Flick



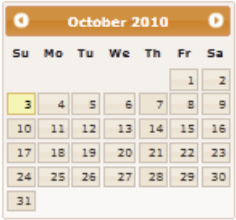
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/humanity/jquery-ui.css

Le-Frog

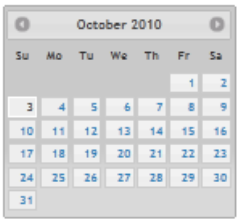
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/mint-choc/jquery-ui.css

---

Overcast

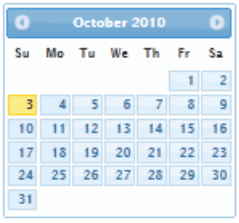


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/pepper-grinder/jquery-ui.css

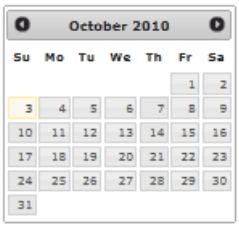
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/start/jquery-ui.css

---

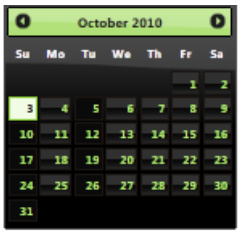
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/swanky-purse/jquery-ui.css

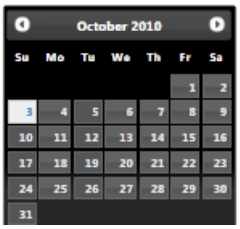
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/trontastic/jquery-ui.css

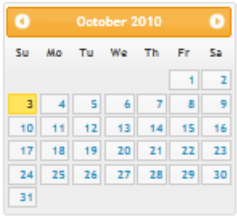
---

UI-Darkness



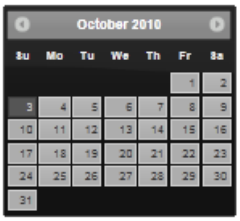
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.4/themes/vader/jquery-ui.css

jQuery UI 1.10.3 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.10.3 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/jquery-ui.min.js

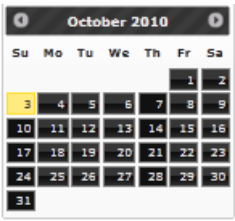
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/black-tie/jquery-ui.css

Blitzer

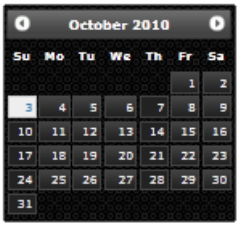
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/cupertino/jquery-ui.css

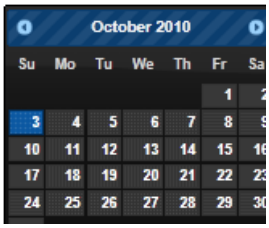
---

Dark-Hive



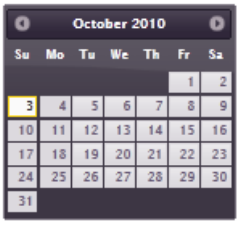
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/dot-luv/jquery-ui.css

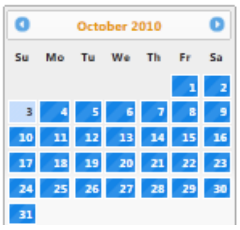
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/eggplant/jquery-ui.css

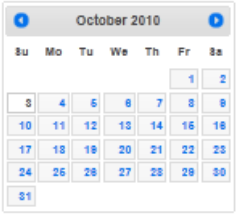
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/excite-bike/jquery-ui.css

Flick



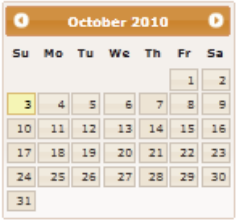
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/humanity/jquery-ui.css

Le-Frog

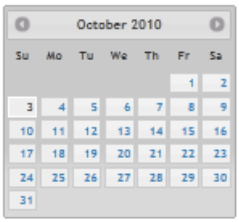
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/mint-choc/jquery-ui.css

---

Overcast

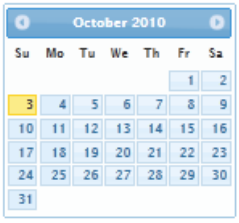


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/pepper-grinder/jquery-ui.css

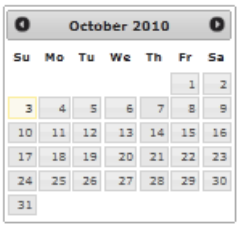
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/start/jquery-ui.css

---

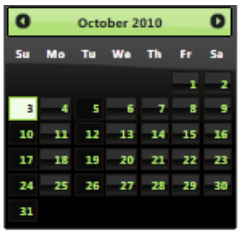
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/swanky-purse/jquery-ui.css

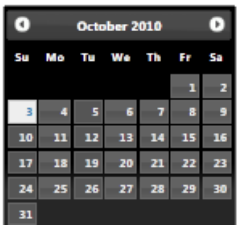
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/trontastic/jquery-ui.css

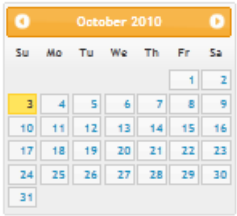
---

UI-Darkness



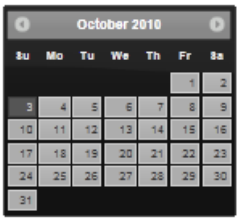
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.3/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.10.2 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.10.2 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/jquery-ui.min.js

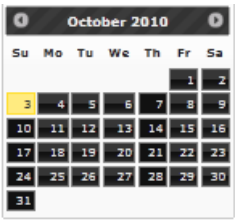
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/black-tie/jquery-ui.css

Blitzer

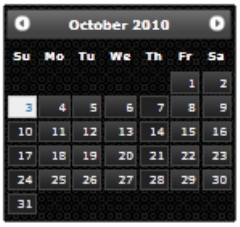
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/cupertino/jquery-ui.css

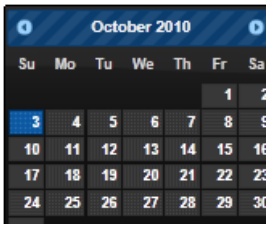
---

Dark-Hive



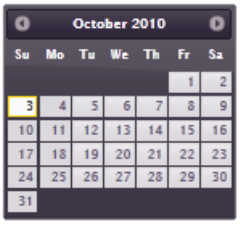
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/dot-luv/jquery-ui.css

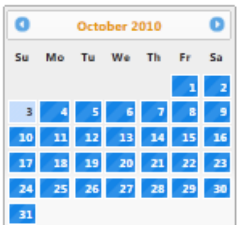
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/eggplant/jquery-ui.css

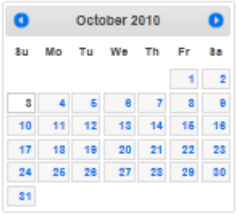
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/excite-bike/jquery-ui.css

Flick



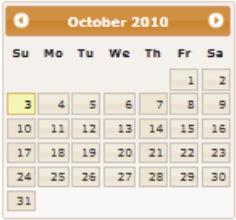
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/humanity/jquery-ui.css

Le-Frog

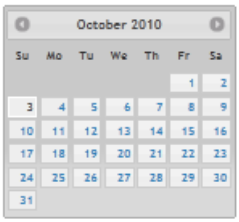
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/mint-choc/jquery-ui.css

---

Overcast

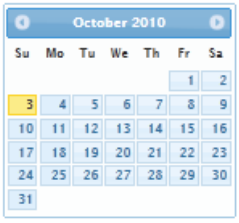


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/pepper-grinder/jquery-ui.css

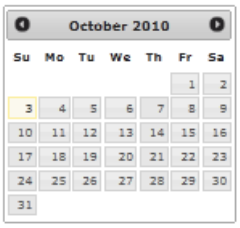
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/redmond/jquery-ui.css

---

Smoothness

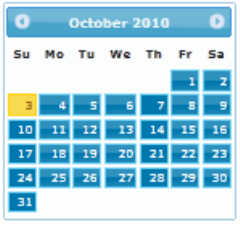


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/south-street/jquery-ui.css

Start



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/start/jquery-ui.css

---

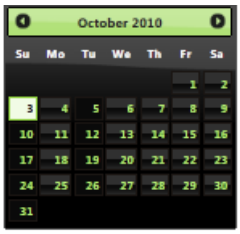
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/swanky-purse/jquery-ui.css

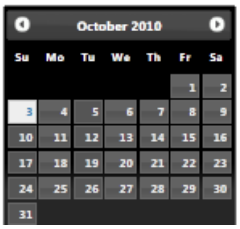
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/trontastic/jquery-ui.css

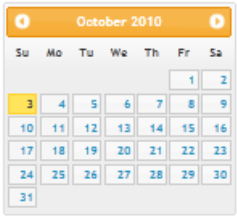
---

UI-Darkness



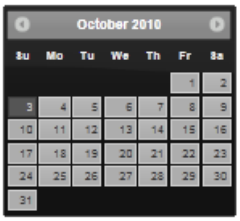
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.2/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.10.1 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.10.1 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/jquery-ui.min.js

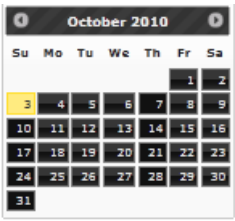
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/black-tie/jquery-ui.css

Blitzer

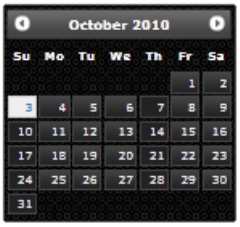
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/cupertino/jquery-ui.css

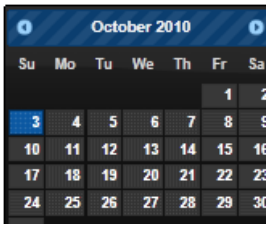
---

Dark-Hive



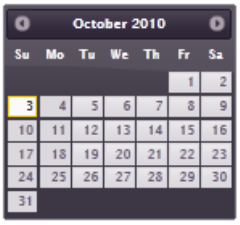
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/dot-luv/jquery-ui.css

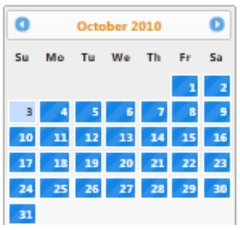
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/eggplant/jquery-ui.css

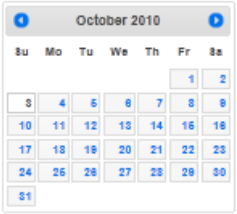
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/excite-bike/jquery-ui.css

Flick



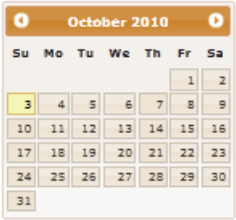
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/humanity/jquery-ui.css

Le-Frog

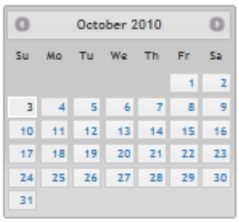
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/mint-choc/jquery-ui.css

---

Overcast

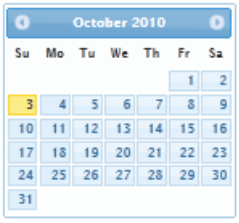


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/pepper-grinder/jquery-ui.css

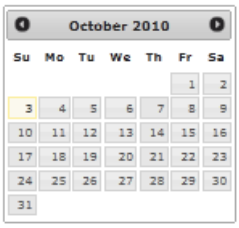
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/start/jquery-ui.css

---

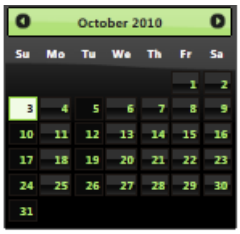
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/swanky-purse/jquery-ui.css

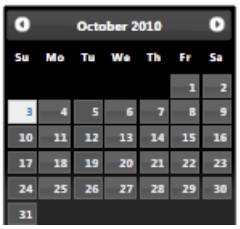
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/trontastic/jquery-ui.css

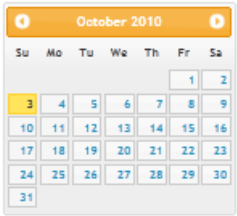
---

UI-Darkness



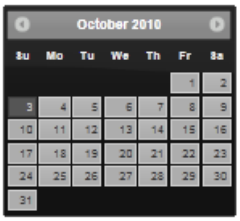
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.1/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.10.0 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.10.0 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/jquery-ui.min.js

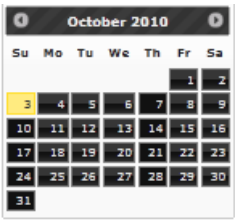
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/black-tie/jquery-ui.css

Blitzer

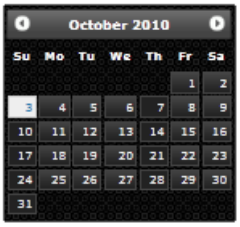
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/cupertino/jquery-ui.css

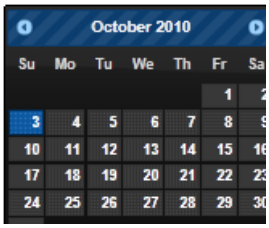
---

Dark-Hive



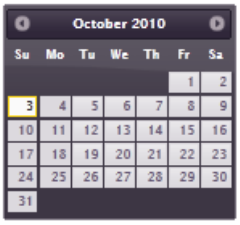
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/dot-luv/jquery-ui.css

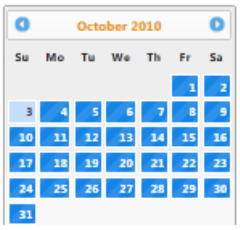
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/eggplant/jquery-ui.css

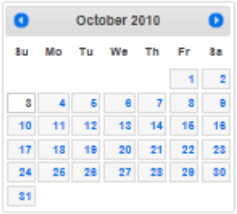
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/excite-bike/jquery-ui.css

Flick



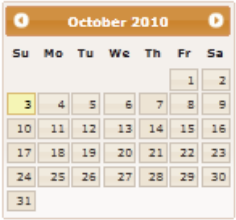
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/humanity/jquery-ui.css

Le-Frog

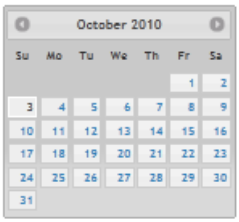
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/mint-choc/jquery-ui.css

---

Overcast

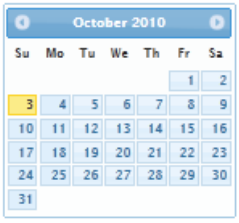


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/pepper-grinder/jquery-ui.css

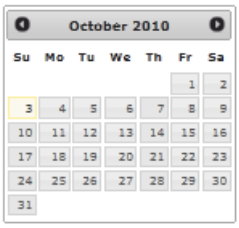
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/start/jquery-ui.css

---

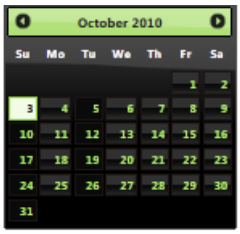
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/swanky-purse/jquery-ui.css

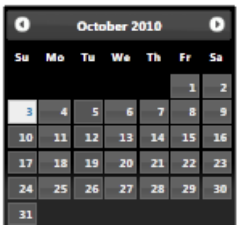
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/trontastic/jquery-ui.css

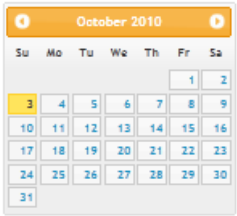
---

UI-Darkness



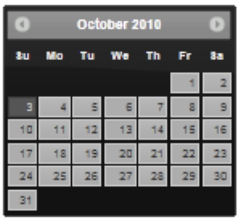
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.10.0/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.9.2 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.9.2 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/jquery-ui.min.js

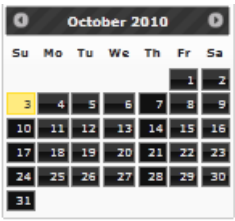
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/black-tie/jquery-ui.css

Blitzer

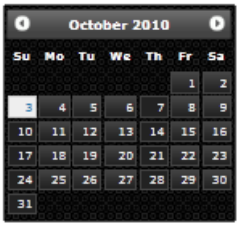
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/cupertino/jquery-ui.css

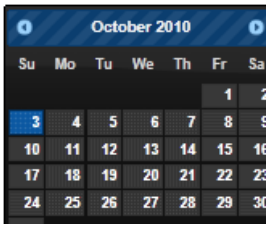
---

Dark-Hive



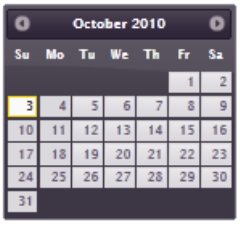
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/dot-luv/jquery-ui.css

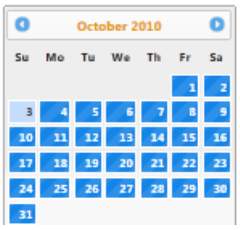
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/eggplant/jquery-ui.css

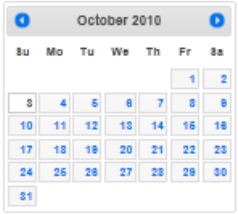
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/excite-bike/jquery-ui.css

Flick



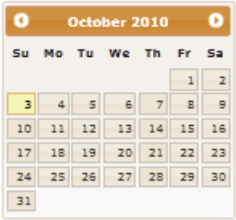
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/humanity/jquery-ui.css

Le-Frog

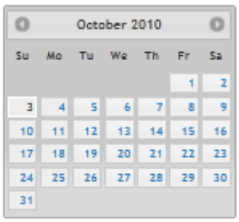
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/mint-choc/jquery-ui.css

---

Overcast

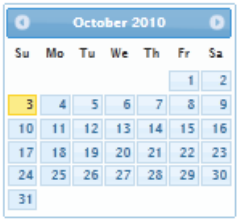


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/pepper-grinder/jquery-ui.css

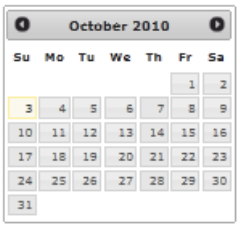
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/redmond/jquery-ui.css

---

Smoothness

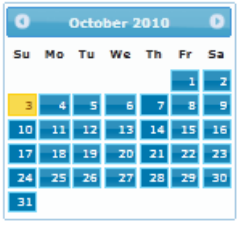


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/south-street/jquery-ui.css

Start



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/start/jquery-ui.css

---

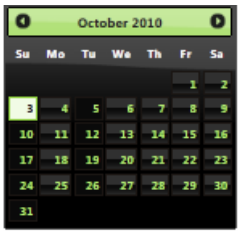
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/swanky-purse/jquery-ui.css

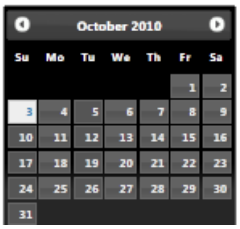
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/trontastic/jquery-ui.css

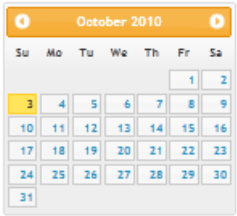
---

UI-Darkness



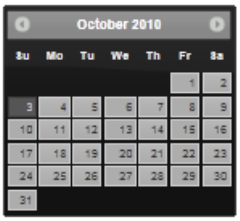
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.2/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.9.1 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.9.1 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/jquery-ui.min.js

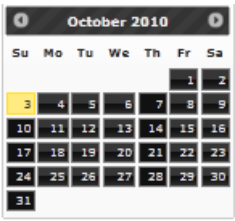
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/black-tie/jquery-ui.css

Blitzer

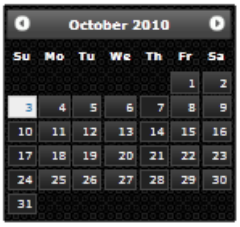
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/cupertino/jquery-ui.css

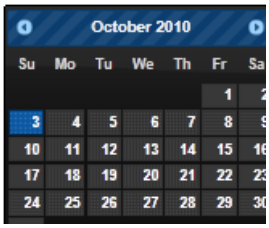
---

Dark-Hive



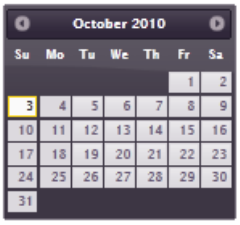
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/dot-luv/jquery-ui.css

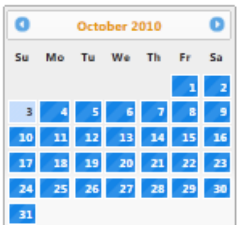
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/eggplant/jquery-ui.css

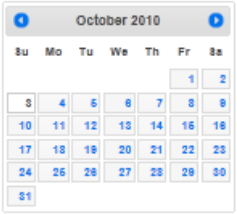
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/excite-bike/jquery-ui.css

Flick



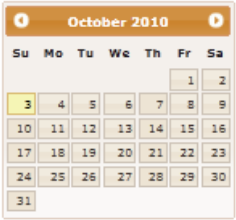
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/humanity/jquery-ui.css

Le-Frog

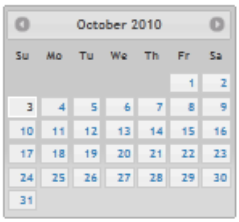
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/mint-choc/jquery-ui.css

---

Overcast

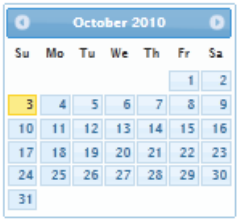


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/pepper-grinder/jquery-ui.css

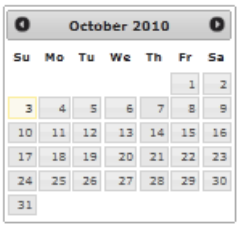
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/start/jquery-ui.css

---

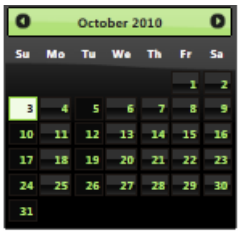
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/swanky-purse/jquery-ui.css

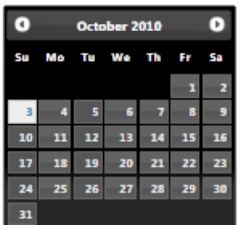
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/trontastic/jquery-ui.css

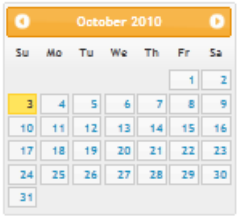
---

UI-Darkness



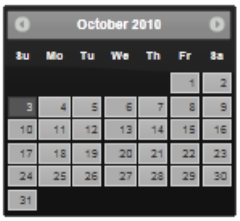
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.1/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.9.0 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.9.0 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/jquery-ui.min.js

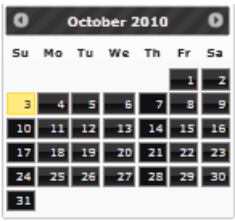
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/black-tie/jquery-ui.css

Blitzer

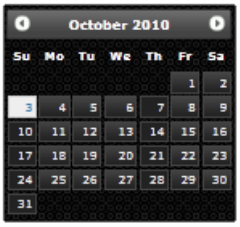
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/cupertino/jquery-ui.css

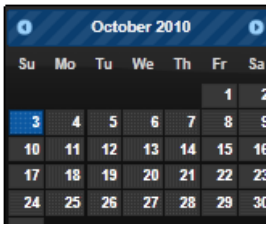
---

Dark-Hive



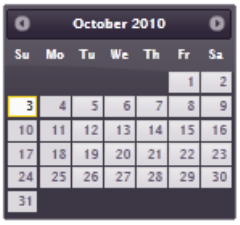
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/dot-luv/jquery-ui.css

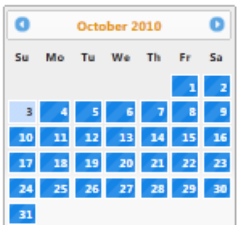
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/eggplant/jquery-ui.css

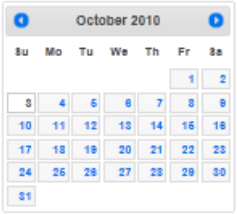
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/excite-bike/jquery-ui.css

Flick



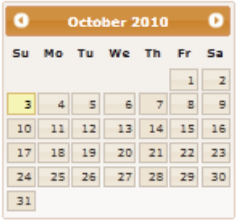
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/humanity/jquery-ui.css

Le-Frog

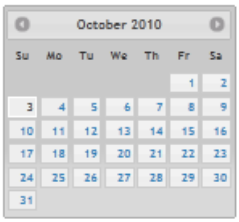
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/mint-choc/jquery-ui.css

---

Overcast

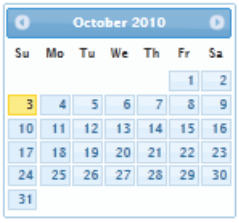


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/pepper-grinder/jquery-ui.css

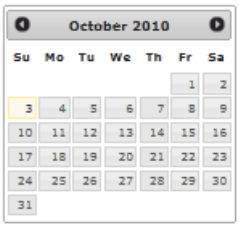
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/start/jquery-ui.css

---

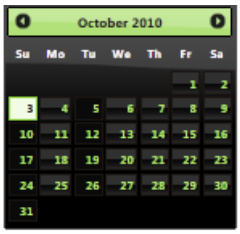
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/swanky-purse/jquery-ui.css

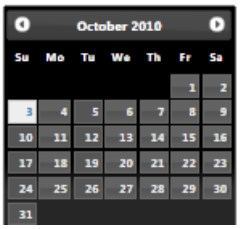
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/trontastic/jquery-ui.css

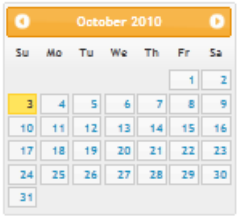
---

UI-Darkness



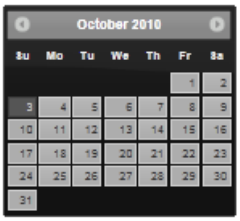
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.9.0/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.24 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.24 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/jquery-ui.min.js

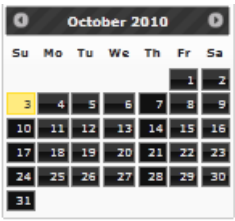
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/black-tie/jquery-ui.css

Blitzer

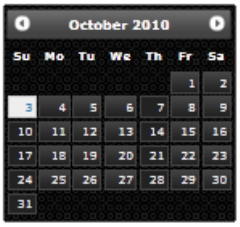
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/cupertino/jquery-ui.css

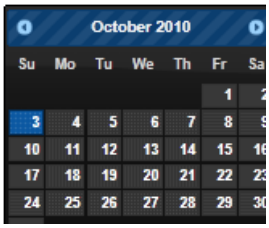
---

Dark-Hive



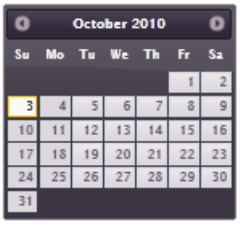
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/dot-luv/jquery-ui.css

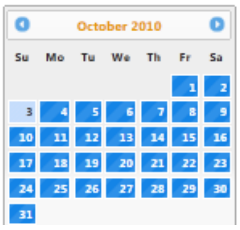
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/eggplant/jquery-ui.css

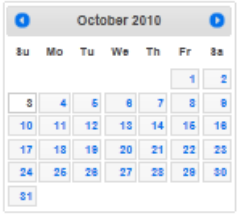
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/excite-bike/jquery-ui.css

Flick



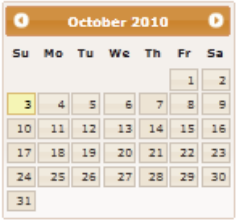
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/humanity/jquery-ui.css

Le-Frog

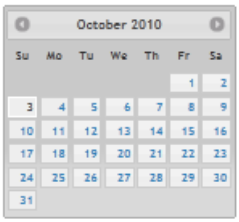
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/mint-choc/jquery-ui.css

---

Overcast

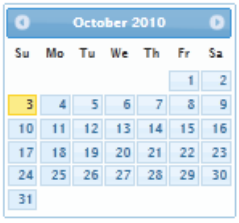


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/pepper-grinder/jquery-ui.css

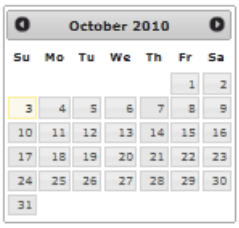
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/start/jquery-ui.css

---

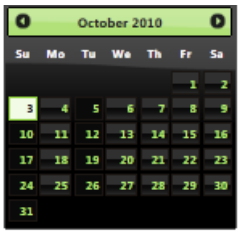
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/swanky-purse/jquery-ui.css

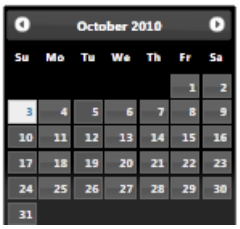
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/trontastic/jquery-ui.css

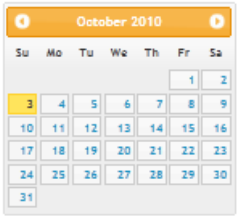
---

UI-Darkness



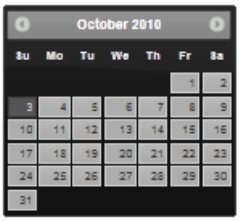
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.24/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.23 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.23 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/jquery-ui.min.js

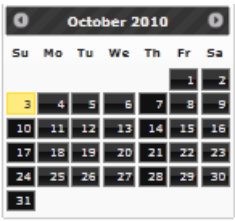
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/black-tie/jquery-ui.css

Blitzer

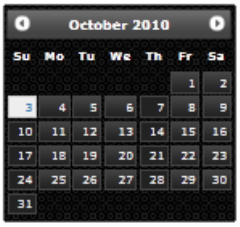
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/cupertino/jquery-ui.css

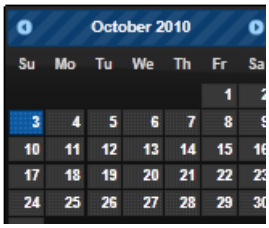
---

Dark-Hive



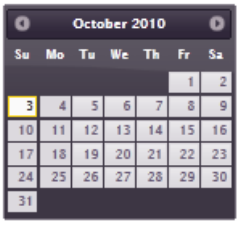
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/dot-luv/jquery-ui.css

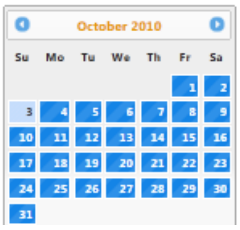
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/eggplant/jquery-ui.css

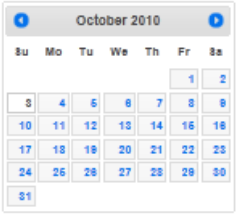
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/excite-bike/jquery-ui.css

Flick



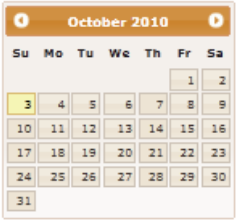
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/humanity/jquery-ui.css

Le-Frog

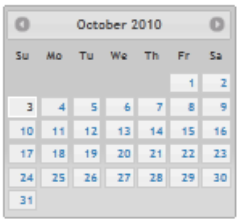
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/mint-choc/jquery-ui.css

---

Overcast

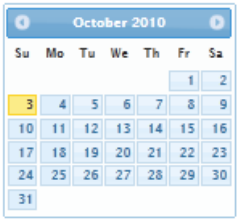


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/pepper-grinder/jquery-ui.css

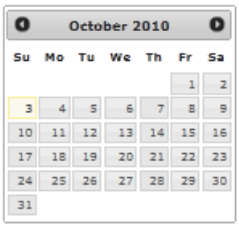
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/start/jquery-ui.css

---

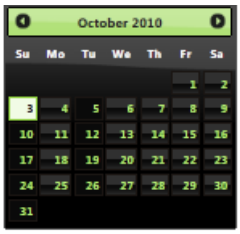
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/swanky-purse/jquery-ui.css

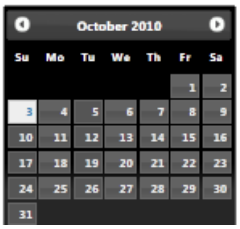
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/trontastic/jquery-ui.css

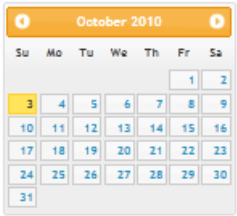
---

UI-Darkness



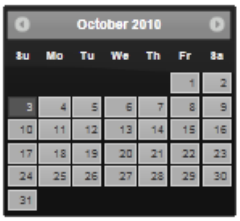
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.23/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.22 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.22 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/jquery-ui.min.js

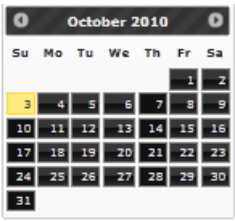
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/black-tie/jquery-ui.css

Blitzer

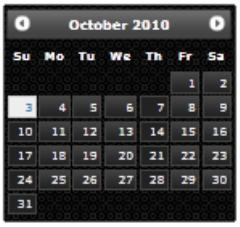
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/cupertino/jquery-ui.css

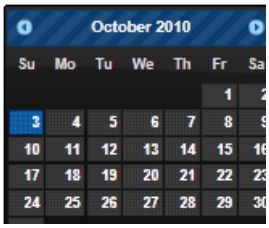
---

Dark-Hive



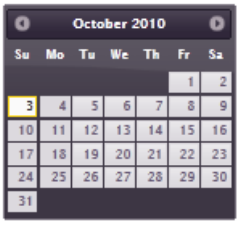
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/dot-luv/jquery-ui.css

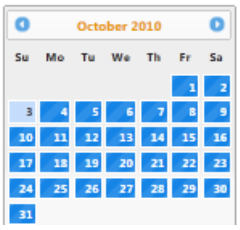
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/eggplant/jquery-ui.css

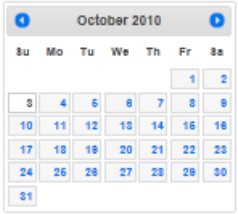
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/excite-bike/jquery-ui.css

Flick



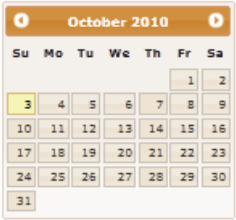
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/humanity/jquery-ui.css

Le-Frog

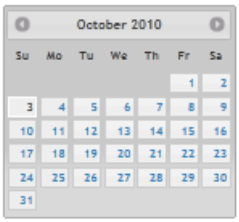
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/mint-choc/jquery-ui.css

---

Overcast

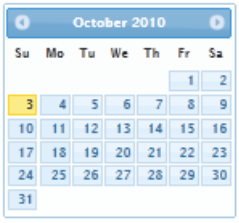


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/pepper-grinder/jquery-ui.css

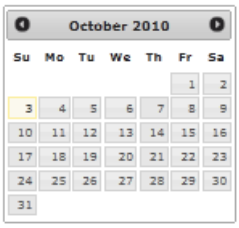
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/start/jquery-ui.css

---

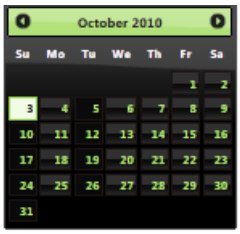
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/swanky-purse/jquery-ui.css

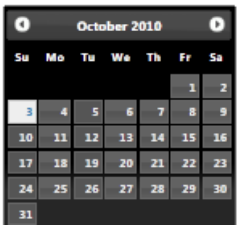
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/trontastic/jquery-ui.css

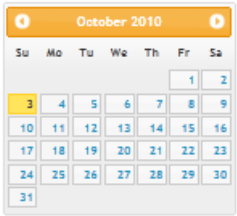
---

UI-Darkness



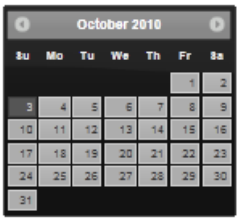
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.22/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.21 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.21 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/jquery-ui.min.js

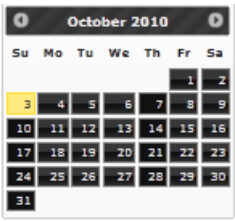
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/black-tie/jquery-ui.css

Blitzer

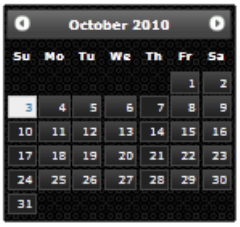
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/cupertino/jquery-ui.css

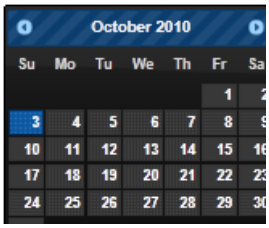
---

Dark-Hive



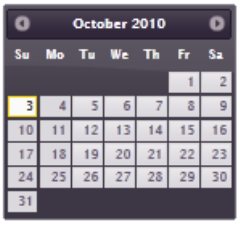
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/dot-luv/jquery-ui.css

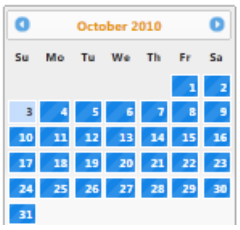
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/eggplant/jquery-ui.css

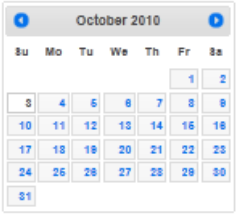
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/excite-bike/jquery-ui.css

Flick



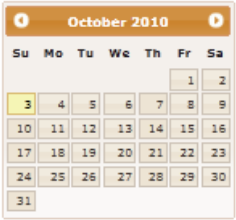
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/humanity/jquery-ui.css

Le-Frog

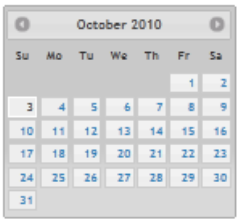
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/mint-choc/jquery-ui.css

---

Overcast

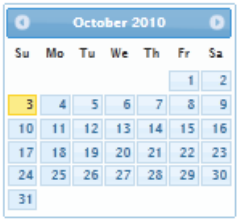


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/pepper-grinder/jquery-ui.css

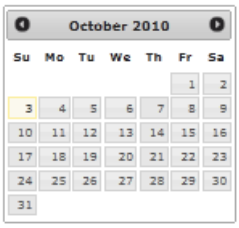
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/start/jquery-ui.css

---

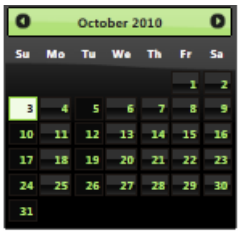
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/swanky-purse/jquery-ui.css

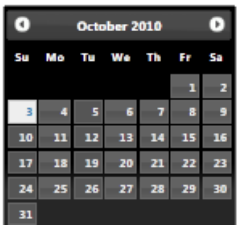
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/trontastic/jquery-ui.css

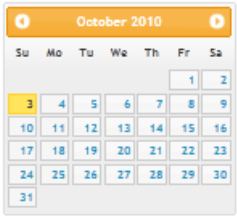
---

UI-Darkness



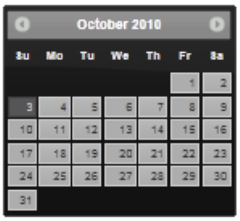
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.21/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.20 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.20 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/jquery-ui.min.js

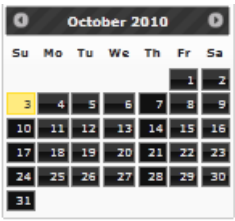
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/black-tie/jquery-ui.css

Blitzer

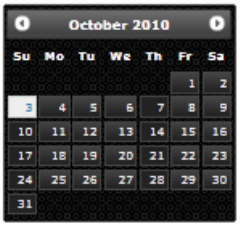
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/cupertino/jquery-ui.css

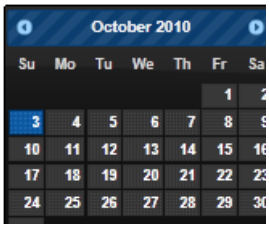
---

Dark-Hive



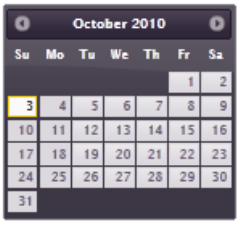
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/dot-luv/jquery-ui.css

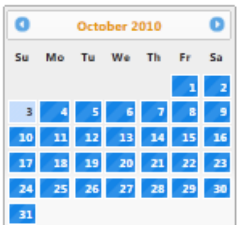
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/eggplant/jquery-ui.css

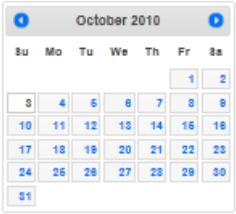
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/excite-bike/jquery-ui.css

Flick



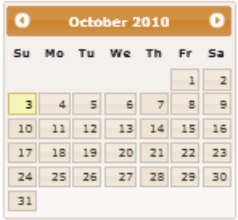
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/humanity/jquery-ui.css

Le-Frog

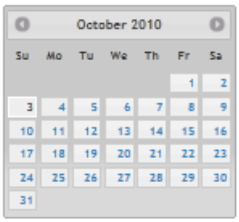
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/mint-choc/jquery-ui.css

---

Overcast

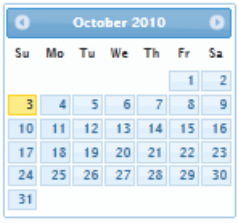


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/pepper-grinder/jquery-ui.css

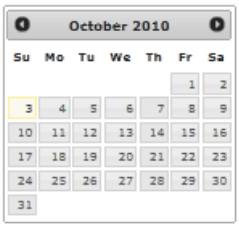
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/start/jquery-ui.css

---

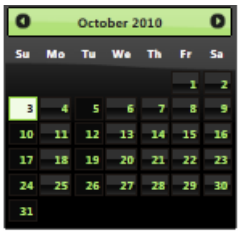
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/swanky-purse/jquery-ui.css

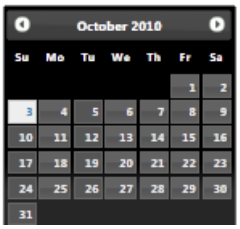
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/trontastic/jquery-ui.css

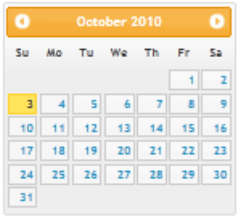
---

UI-Darkness



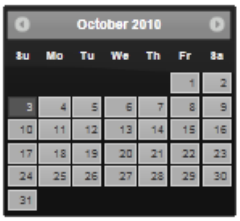
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.20/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.19 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.19 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/jquery-ui.min.js

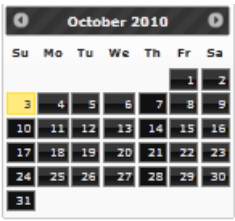
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/black-tie/jquery-ui.css

Blitzer

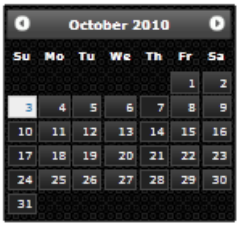
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/cupertino/jquery-ui.css

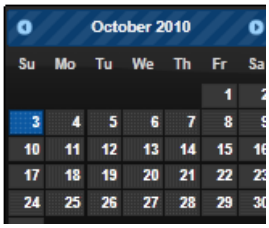
---

Dark-Hive



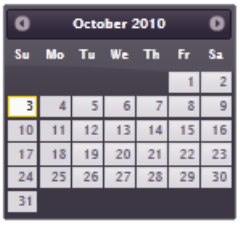
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/dot-luv/jquery-ui.css

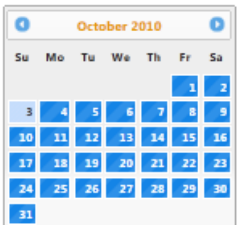
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/eggplant/jquery-ui.css

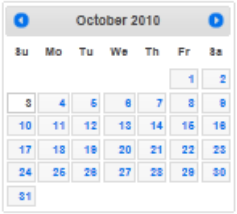
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/excite-bike/jquery-ui.css

Flick



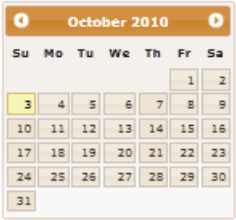
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/humanity/jquery-ui.css

Le-Frog

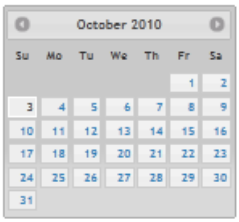
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/mint-choc/jquery-ui.css

---

Overcast

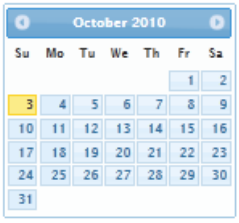


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/pepper-grinder/jquery-ui.css

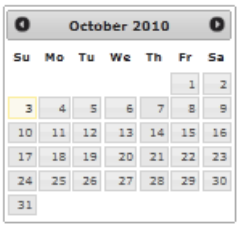
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/start/jquery-ui.css

---

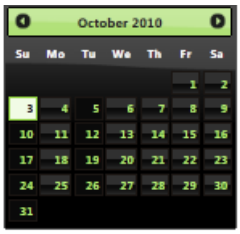
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/swanky-purse/jquery-ui.css

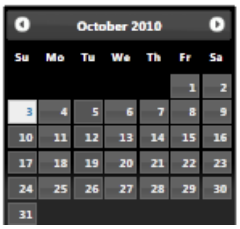
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/trontastic/jquery-ui.css

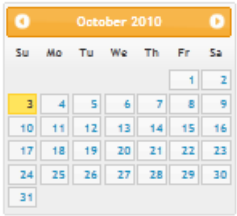
---

UI-Darkness



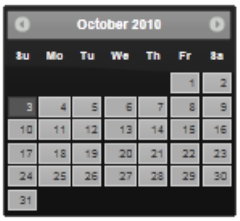
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.19/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.18 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.18 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/jquery-ui.min.js

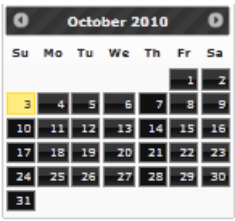
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/black-tie/jquery-ui.css

Blitzer

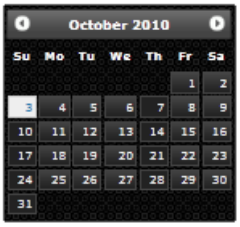
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/cupertino/jquery-ui.css

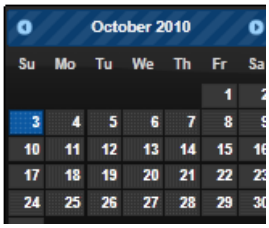
---

Dark-Hive



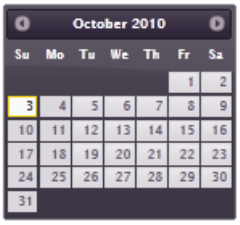
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/dot-luv/jquery-ui.css

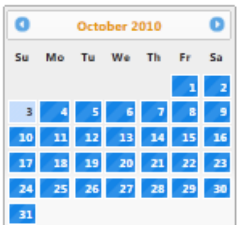
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/eggplant/jquery-ui.css

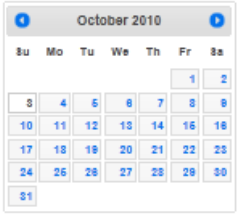
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/excite-bike/jquery-ui.css

Flick



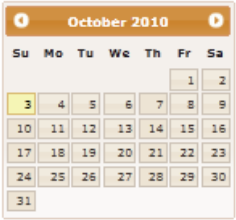
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/humanity/jquery-ui.css

Le-Frog

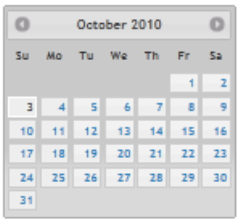
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/mint-choc/jquery-ui.css

---

Overcast

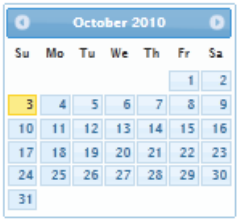


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/pepper-grinder/jquery-ui.css

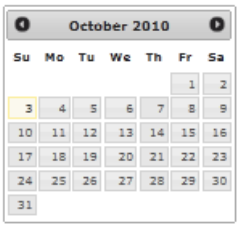
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/start/jquery-ui.css

---

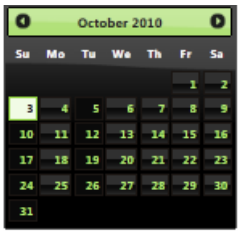
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/swanky-purse/jquery-ui.css

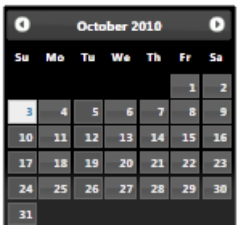
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/trontastic/jquery-ui.css

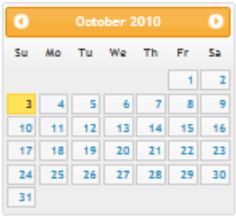
---

UI-Darkness



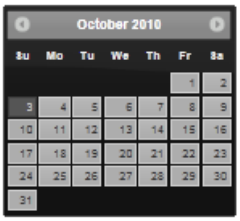
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.18/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.17 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.17 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/jquery-ui.min.js

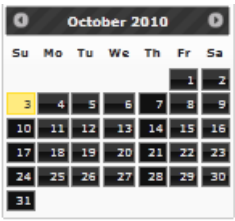
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/black-tie/jquery-ui.css

Blitzer

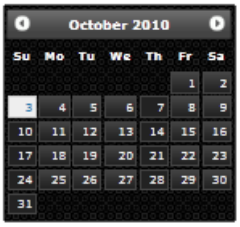
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/cupertino/jquery-ui.css

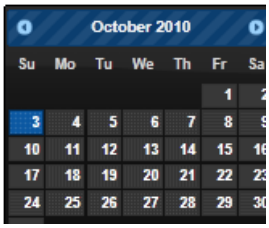
---

Dark-Hive



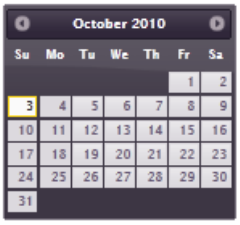
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/dot-luv/jquery-ui.css

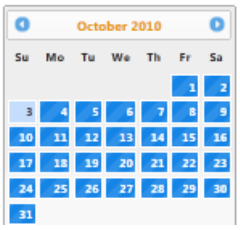
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/eggplant/jquery-ui.css

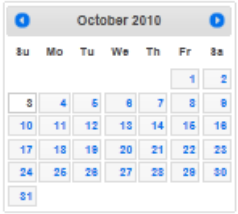
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/excite-bike/jquery-ui.css

Flick



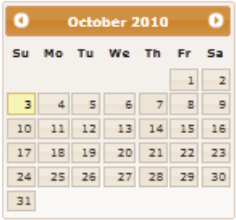
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/humanity/jquery-ui.css

Le-Frog

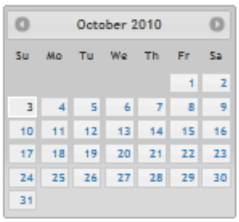
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/mint-choc/jquery-ui.css

---

Overcast

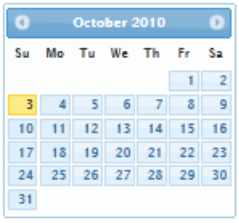


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/pepper-grinder/jquery-ui.css

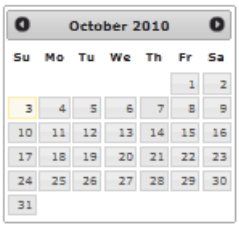
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/start/jquery-ui.css

---

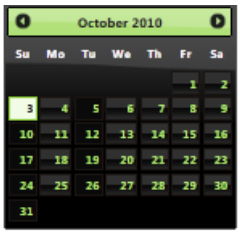
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/swanky-purse/jquery-ui.css

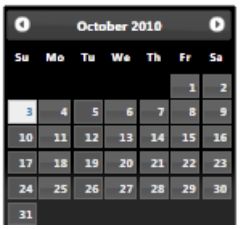
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/trontastic/jquery-ui.css

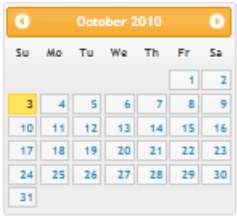
---

UI-Darkness



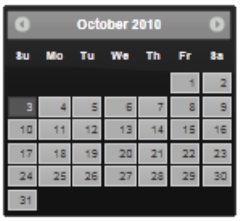
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.17/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.16 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.16 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/jquery-ui.min.js

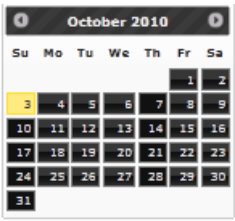
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/black-tie/jquery-ui.css

Blitzer

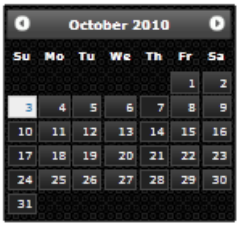
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/cupertino/jquery-ui.css

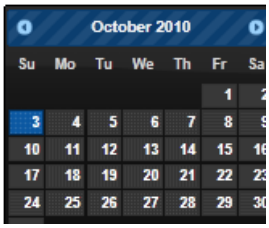
---

Dark-Hive



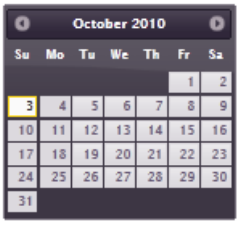
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/dot-luv/jquery-ui.css

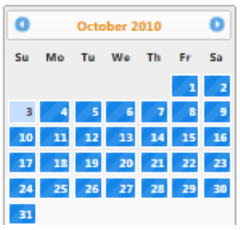
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/eggplant/jquery-ui.css

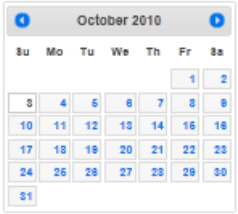
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/excite-bike/jquery-ui.css

Flick



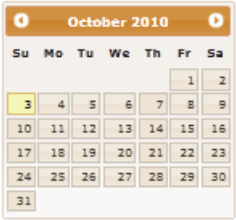
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/humanity/jquery-ui.css

Le-Frog

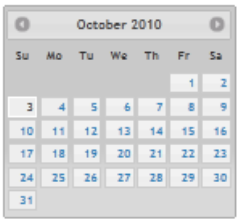
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/mint-choc/jquery-ui.css

---

Overcast

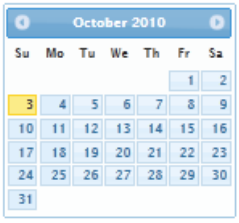


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/pepper-grinder/jquery-ui.css

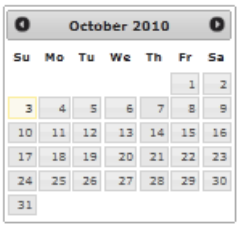
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/start/jquery-ui.css

---

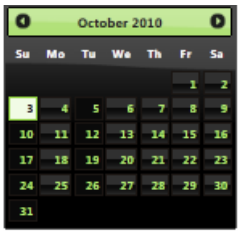
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/swanky-purse/jquery-ui.css

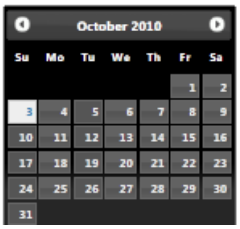
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/trontastic/jquery-ui.css

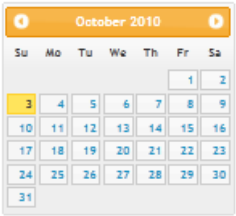
---

UI-Darkness



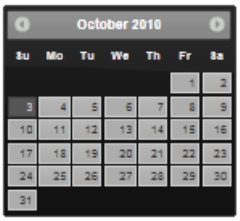
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.16/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.15 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.15 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/jquery-ui.min.js

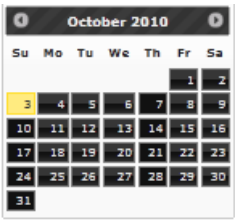
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/black-tie/jquery-ui.css

Blitzer

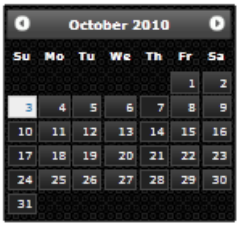
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/cupertino/jquery-ui.css

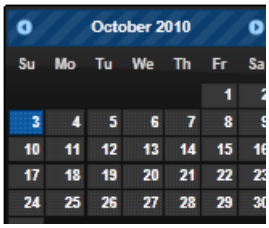
---

Dark-Hive



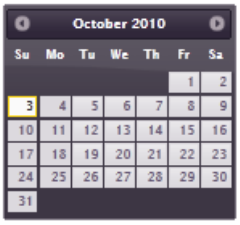
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/dot-luv/jquery-ui.css

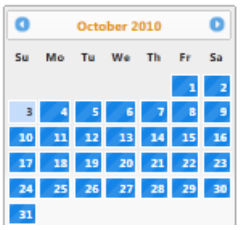
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/eggplant/jquery-ui.css

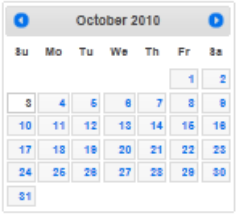
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/excite-bike/jquery-ui.css

Flick



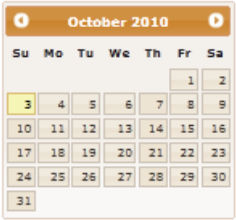
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/humanity/jquery-ui.css

Le-Frog

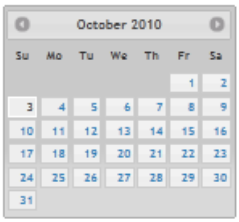
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/mint-choc/jquery-ui.css

---

Overcast

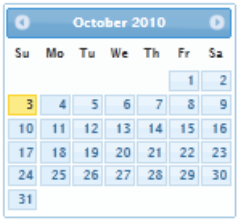


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/pepper-grinder/jquery-ui.css

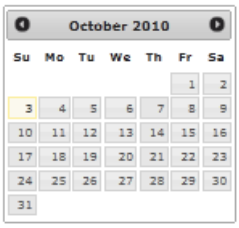
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/start/jquery-ui.css

---

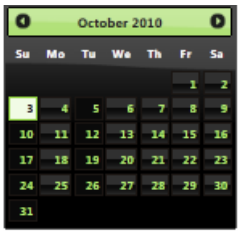
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/swanky-purse/jquery-ui.css

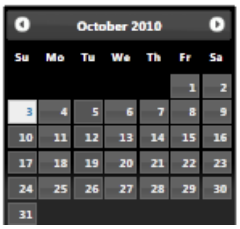
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/trontastic/jquery-ui.css

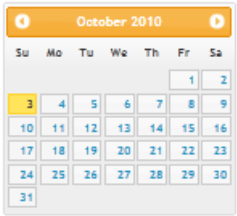
---

UI-Darkness



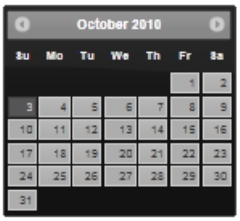
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.15/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.14 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.14 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/jquery-ui.min.js

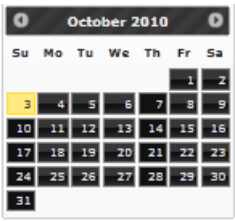
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/black-tie/jquery-ui.css

Blitzer

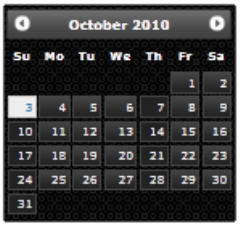
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/cupertino/jquery-ui.css

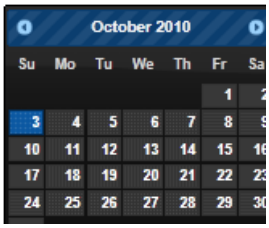
---

Dark-Hive



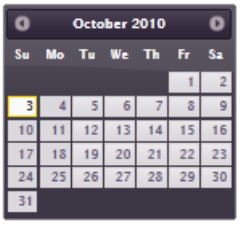
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/dot-luv/jquery-ui.css

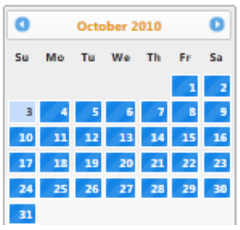
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/eggplant/jquery-ui.css

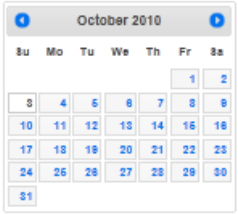
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/excite-bike/jquery-ui.css

Flick



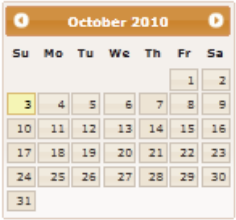
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/humanity/jquery-ui.css

Le-Frog

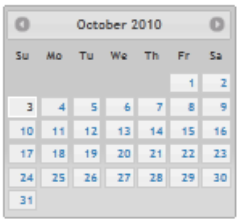
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/mint-choc/jquery-ui.css

---

Overcast

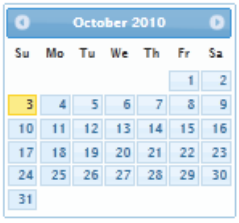


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/pepper-grinder/jquery-ui.css

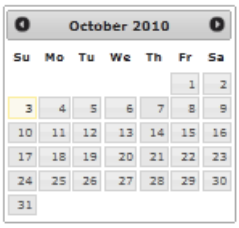
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/start/jquery-ui.css

---

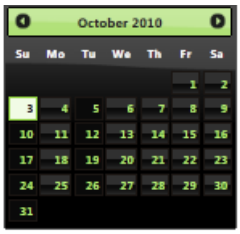
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/swanky-purse/jquery-ui.css

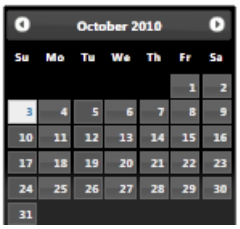
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/trontastic/jquery-ui.css

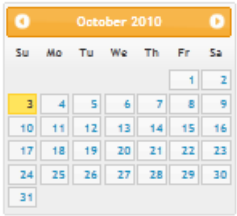
---

UI-Darkness



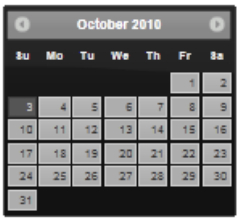
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.14/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.13 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.13 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/jquery-ui.min.js

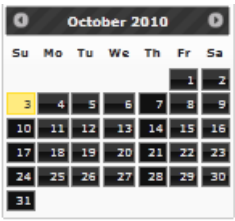
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/black-tie/jquery-ui.css

Blitzer

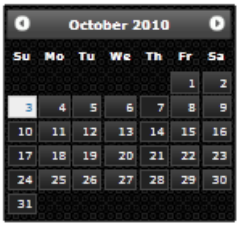
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/cupertino/jquery-ui.css

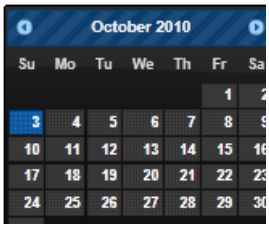
---

Dark-Hive



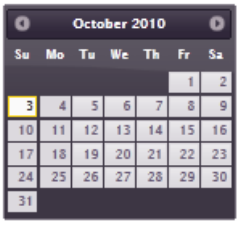
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/dot-luv/jquery-ui.css

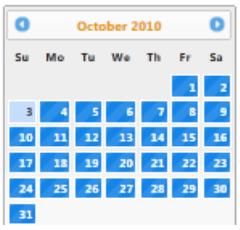
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/eggplant/jquery-ui.css

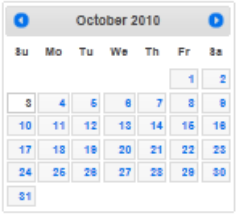
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/excite-bike/jquery-ui.css

Flick



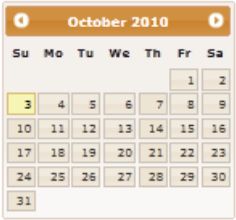
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/humanity/jquery-ui.css

Le-Frog

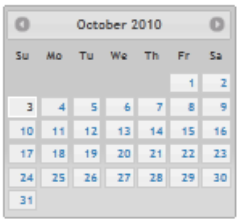
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/mint-choc/jquery-ui.css

---

Overcast

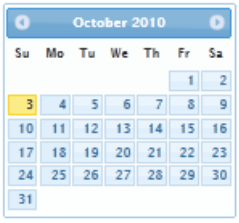


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/pepper-grinder/jquery-ui.css

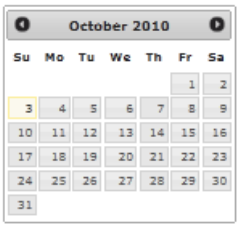
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/start/jquery-ui.css

---

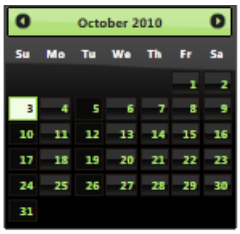
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/swanky-purse/jquery-ui.css

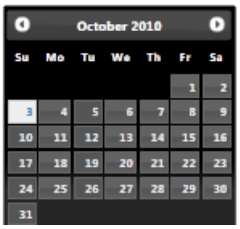
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/trontastic/jquery-ui.css

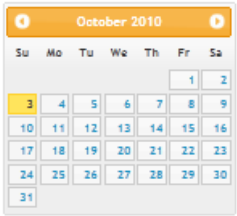
---

UI-Darkness



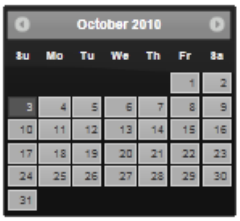
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.13/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.12 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.12 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/jquery-ui.min.js

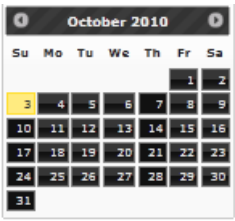
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/black-tie/jquery-ui.css

Blitzer

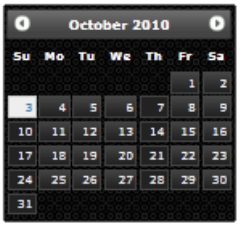
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/cupertino/jquery-ui.css

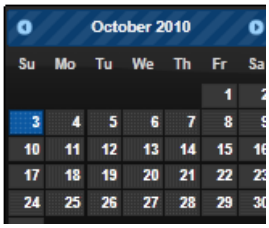
---

Dark-Hive



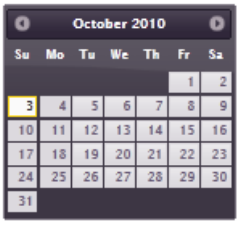
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/dot-luv/jquery-ui.css

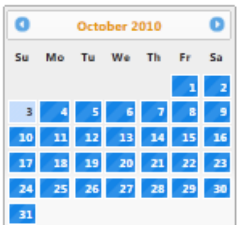
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/eggplant/jquery-ui.css

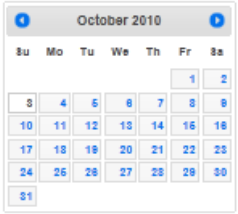
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/excite-bike/jquery-ui.css

Flick



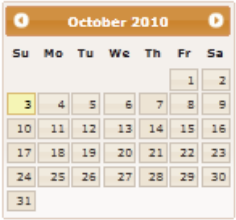
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/humanity/jquery-ui.css

Le-Frog

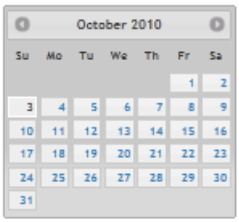
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/mint-choc/jquery-ui.css

---

Overcast

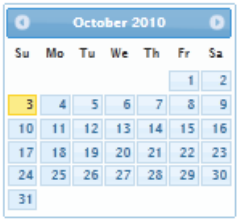


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/pepper-grinder/jquery-ui.css

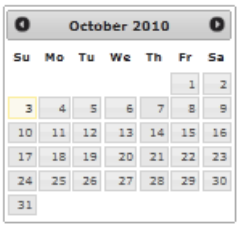
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/start/jquery-ui.css

---

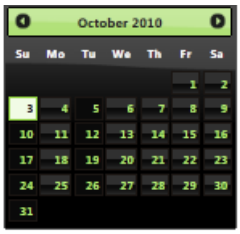
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/swanky-purse/jquery-ui.css

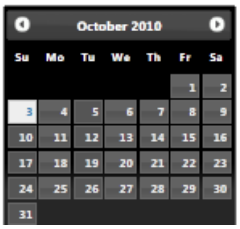
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/trontastic/jquery-ui.css

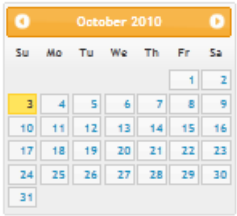
---

UI-Darkness



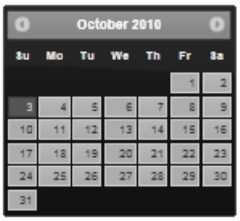
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.12/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.11 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.11 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/jquery-ui.min.js

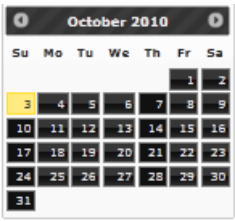
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/black-tie/jquery-ui.css

Blitzer

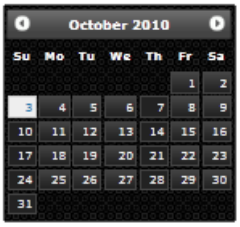
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/cupertino/jquery-ui.css

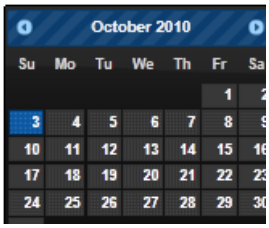
---

Dark-Hive



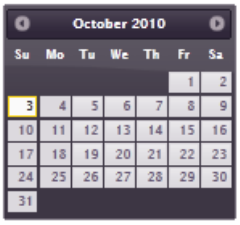
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/dot-luv/jquery-ui.css

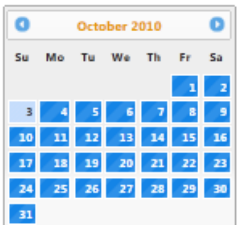
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/eggplant/jquery-ui.css

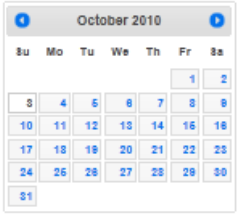
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/excite-bike/jquery-ui.css

Flick



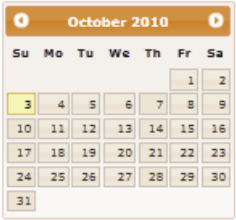
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/humanity/jquery-ui.css

Le-Frog

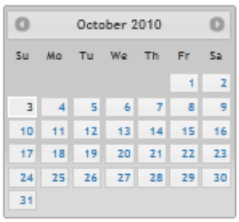
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/mint-choc/jquery-ui.css

---

Overcast

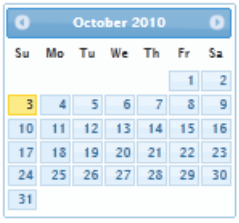


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/pepper-grinder/jquery-ui.css

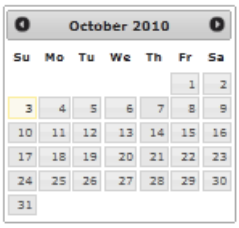
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/start/jquery-ui.css

---

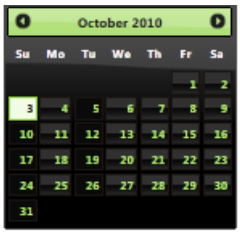
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/swanky-purse/jquery-ui.css

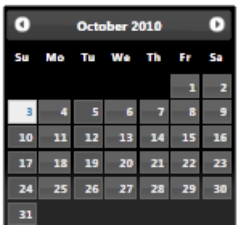
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/trontastic/jquery-ui.css

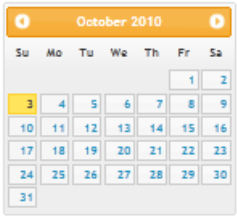
---

UI-Darkness



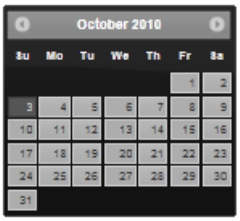
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.11/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.10 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.10 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/jquery-ui.min.js

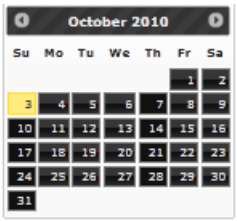
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/black-tie/jquery-ui.css

Blitzer

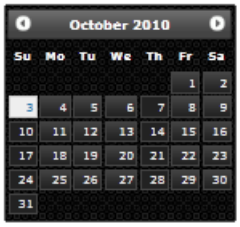
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/cupertino/jquery-ui.css

---

Dark-Hive

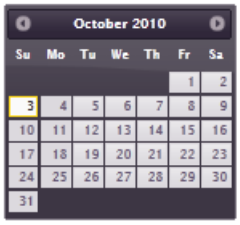


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/dark-hive/jquery-ui.css

Dot-Luv

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/dot-luv/jquery-ui.css

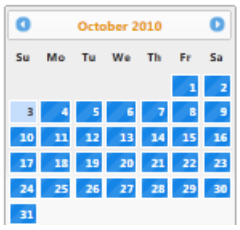
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/eggplant/jquery-ui.css

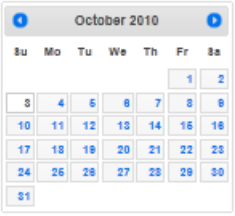
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/excite-bike/jquery-ui.css

Flick



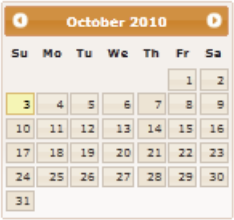
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/humanity/jquery-ui.css

Le-Frog

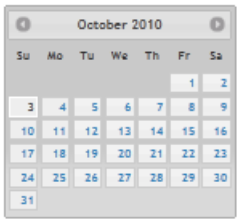
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/mint-choc/jquery-ui.css

---

Overcast

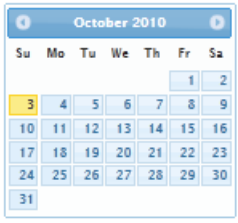


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/pepper-grinder/jquery-ui.css

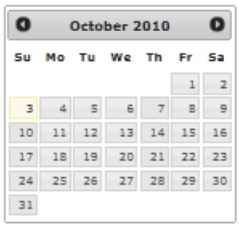
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/redmond/jquery-ui.css

---

Smoothness

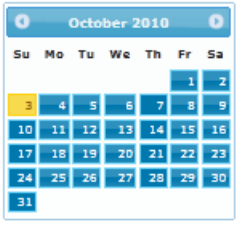


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/south-street/jquery-ui.css

Start



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/start/jquery-ui.css

---

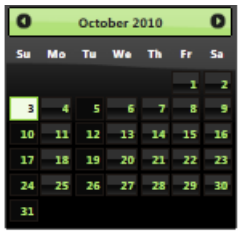
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/swanky-purse/jquery-ui.css

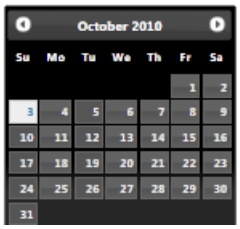
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/trontastic/jquery-ui.css

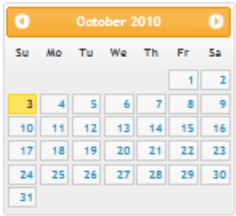
---

UI-Darkness



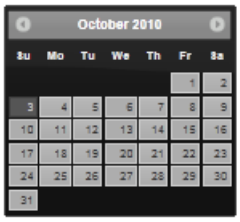
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.10/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.9 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.9 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/jquery-ui.min.js

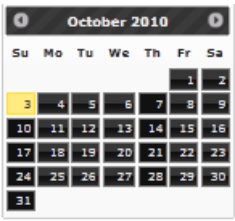
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/black-tie/jquery-ui.css

Blitzer

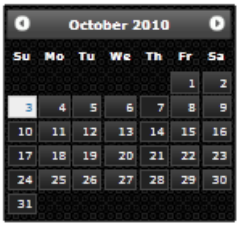
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/cupertino/jquery-ui.css

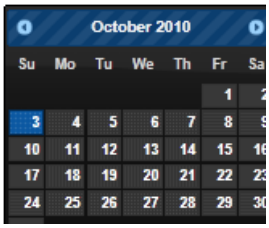
---

Dark-Hive



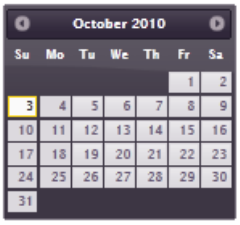
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/dot-luv/jquery-ui.css

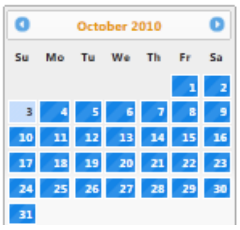
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/eggplant/jquery-ui.css

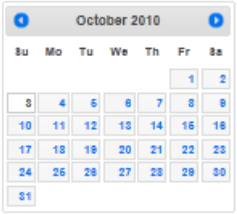
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/excite-bike/jquery-ui.css

Flick



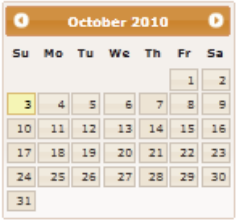
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/humanity/jquery-ui.css

Le-Frog

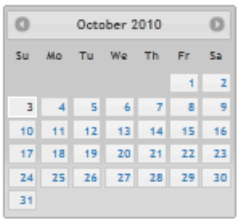
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/mint-choc/jquery-ui.css

---

Overcast

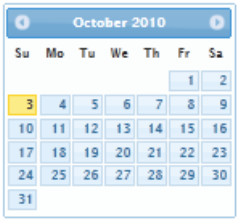


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/pepper-grinder/jquery-ui.css

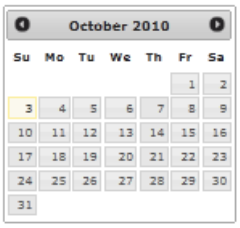
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/redmond/jquery-ui.css

---

Smoothness

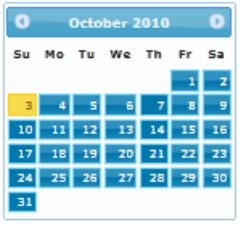


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/south-street/jquery-ui.css

Start



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/start/jquery-ui.css

---

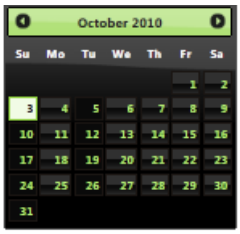
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/swanky-purse/jquery-ui.css

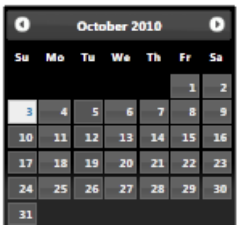
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/trontastic/jquery-ui.css

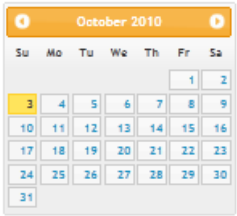
---

UI-Darkness



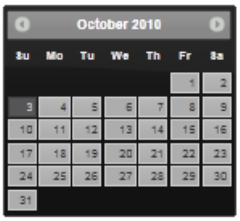
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.9/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.8 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.8 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/jquery-ui.min.js

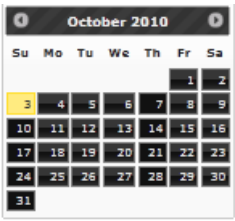
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/black-tie/jquery-ui.css

Blitzer

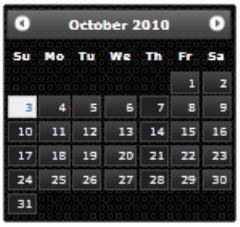
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/cupertino/jquery-ui.css

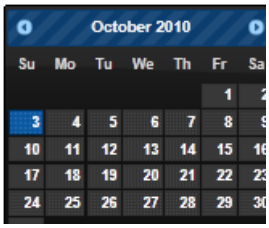
---

Dark-Hive



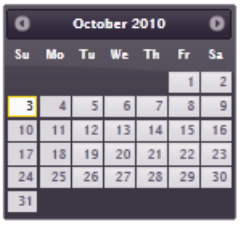
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/dot-luv/jquery-ui.css

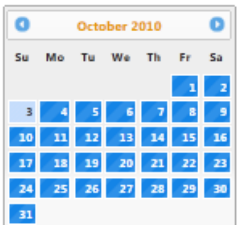
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/eggplant/jquery-ui.css

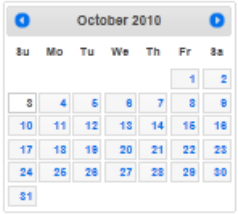
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/excite-bike/jquery-ui.css

Flick



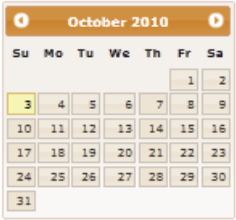
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/humanity/jquery-ui.css

Le-Frog

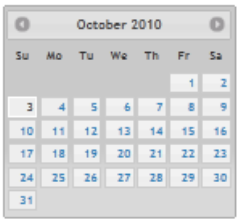
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/mint-choc/jquery-ui.css

---

Overcast

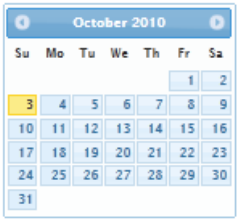


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/pepper-grinder/jquery-ui.css

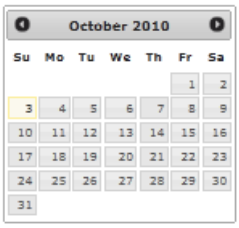
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/start/jquery-ui.css

---

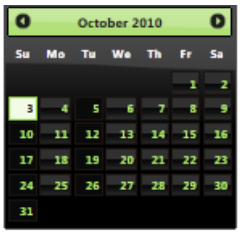
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/swanky-purse/jquery-ui.css

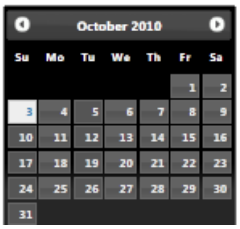
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/trontastic/jquery-ui.css

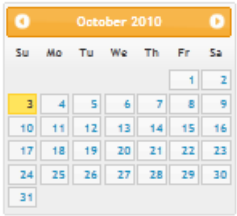
---

UI-Darkness



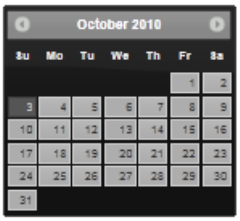
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.8/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.7 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.7 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/jquery-ui.min.js

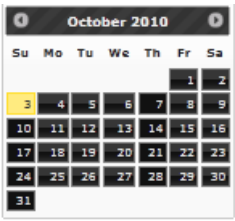
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/black-tie/jquery-ui.css

Blitzer

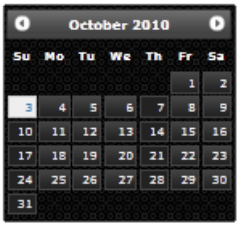
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/cupertino/jquery-ui.css

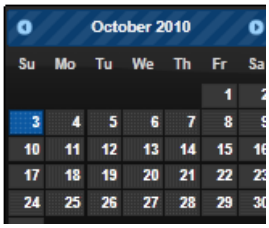
---

Dark-Hive



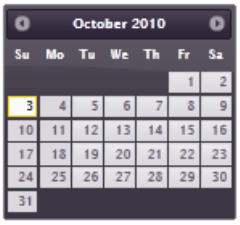
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/dot-luv/jquery-ui.css

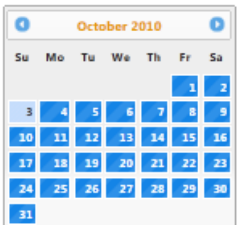
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/eggplant/jquery-ui.css

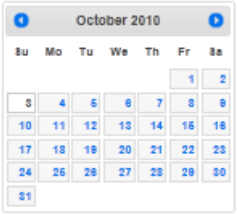
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/excite-bike/jquery-ui.css

Flick



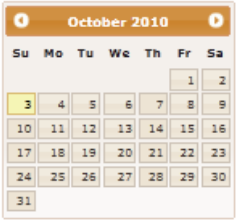
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/humanity/jquery-ui.css

Le-Frog

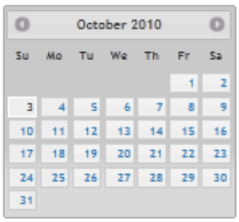
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/mint-choc/jquery-ui.css

---

Overcast

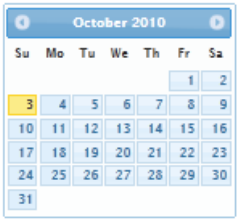


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/pepper-grinder/jquery-ui.css

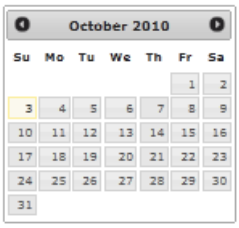
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/start/jquery-ui.css

---

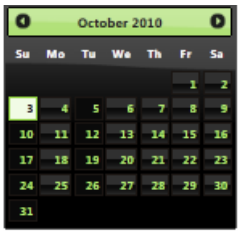
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/swanky-purse/jquery-ui.css

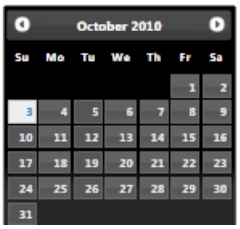
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/trontastic/jquery-ui.css

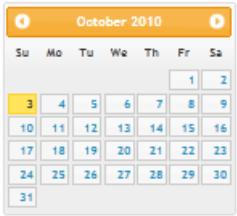
---

UI-Darkness



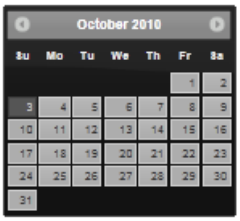
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.7/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.6 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.6 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/jquery-ui.min.js

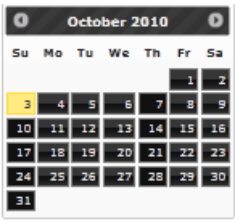
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/black-tie/jquery-ui.css

Blitzer

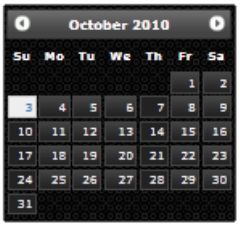
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/cupertino/jquery-ui.css

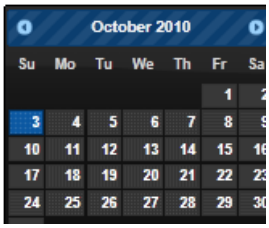
---

Dark-Hive



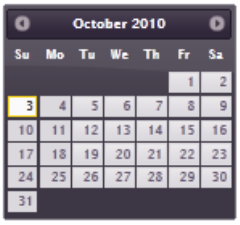
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/dot-luv/jquery-ui.css

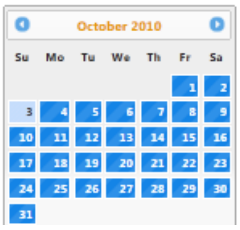
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/eggplant/jquery-ui.css

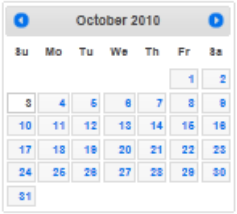
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/excite-bike/jquery-ui.css

Flick



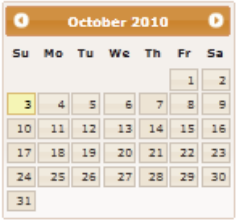
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/humanity/jquery-ui.css

Le-Frog

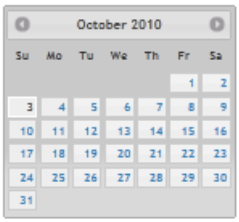
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/mint-choc/jquery-ui.css

---

Overcast

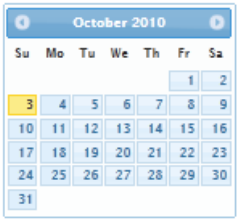


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/pepper-grinder/jquery-ui.css

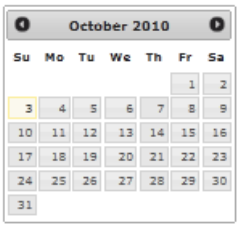
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/start/jquery-ui.css

---

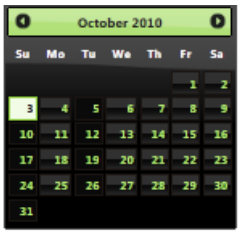
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/swanky-purse/jquery-ui.css

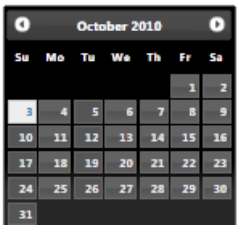
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/trontastic/jquery-ui.css

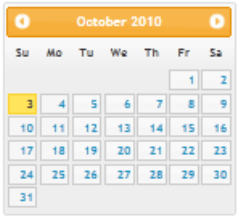
---

UI-Darkness



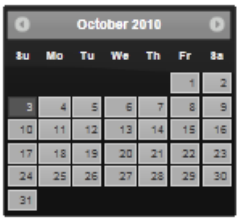
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.6/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.8.5

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.8.5 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/jquery-ui.min.js

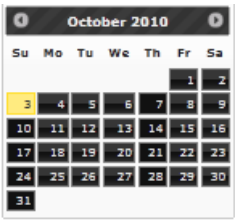
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/black-tie/jquery-ui.css

Blitzer

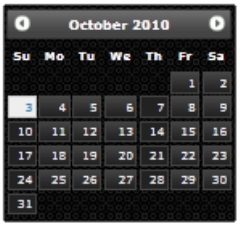
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/cupertino/jquery-ui.css

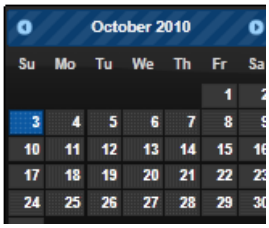
---

Dark-Hive



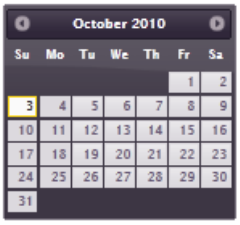
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/dot-luv/jquery-ui.css

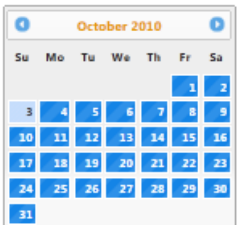
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/eggplant/jquery-ui.css

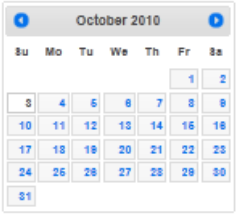
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/excite-bike/jquery-ui.css

Flick



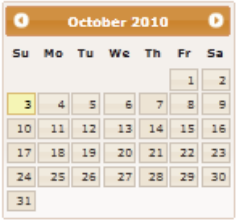
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/humanity/jquery-ui.css

Le-Frog

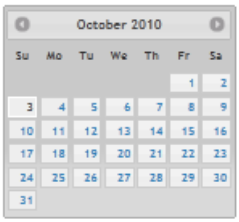
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/mint-choc/jquery-ui.css

---

Overcast

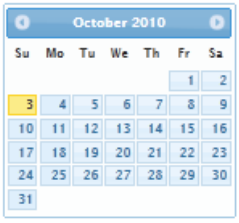


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/pepper-grinder/jquery-ui.css

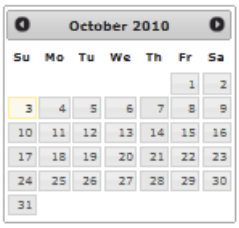
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/start/jquery-ui.css

---

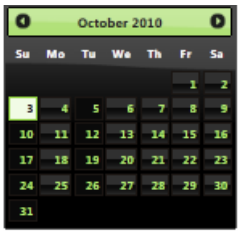
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/swanky-purse/jquery-ui.css

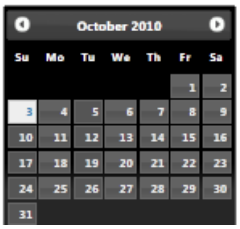
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/trontastic/jquery-ui.css

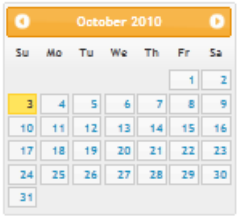
---

UI-Darkness



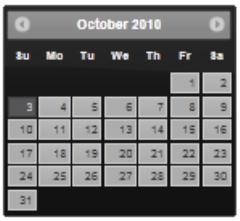
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.8.5/themes/vader/jquery-ui.css

Internationalization/Localization

The i18n folder contains localization information for the jQuery UI library.

jQuery UI 1.11.2 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.11.2 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/jquery-ui.min.js

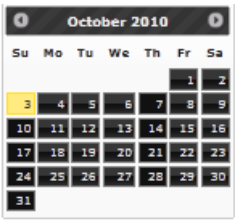
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/black-tie/jquery-ui.css

Blitzer

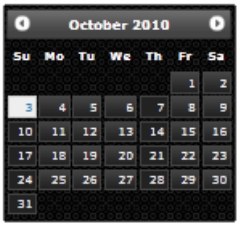
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/cupertino/jquery-ui.css

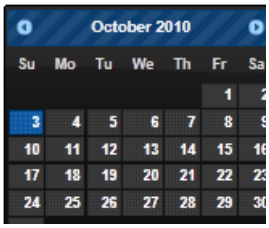
---

Dark-Hive



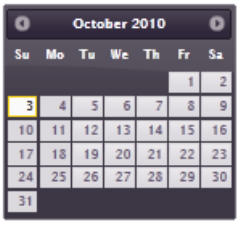
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/dot-luv/jquery-ui.css

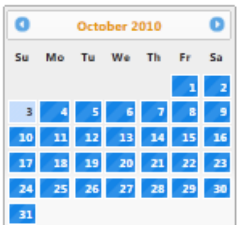
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/eggplant/jquery-ui.css

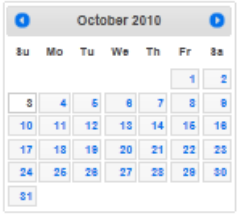
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/excite-bike/jquery-ui.css

Flick



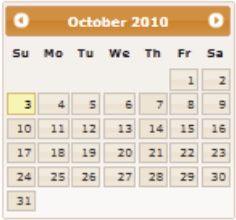
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/humanity/jquery-ui.css

Le-Frog

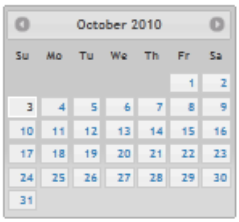
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/mint-choc/jquery-ui.css

---

Overcast

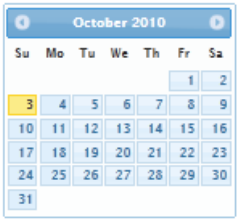


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/pepper-grinder/jquery-ui.css

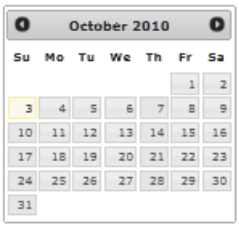
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/start/jquery-ui.css

---

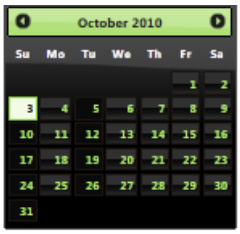
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/swanky-purse/jquery-ui.css

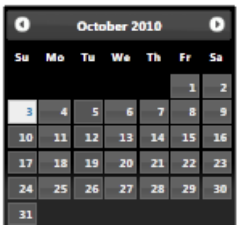
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/trontastic/jquery-ui.css

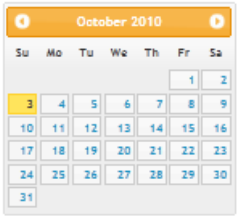
---

UI-Darkness



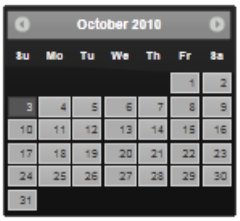
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.2/themes/vader/jquery-ui.css

jQuery UI 1.11.1 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.11.1 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/jquery-ui.min.js

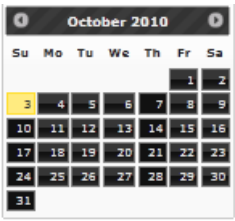
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/black-tie/jquery-ui.css

Blitzer

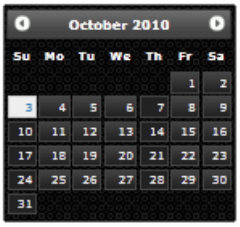
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/cupertino/jquery-ui.css

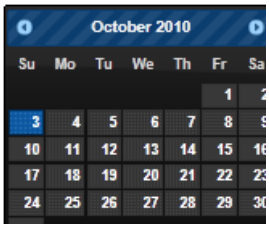
---

Dark-Hive



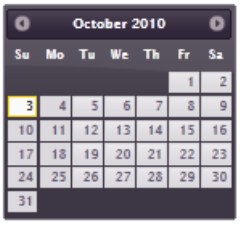
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/dot-luv/jquery-ui.css

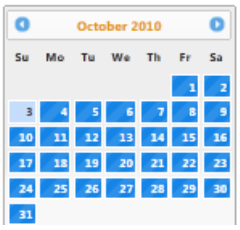
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/eggplant/jquery-ui.css

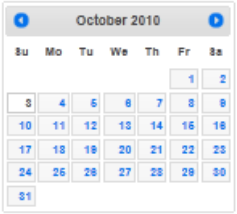
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/excite-bike/jquery-ui.css

Flick



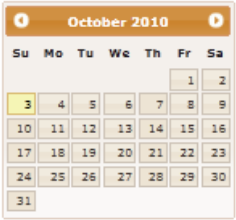
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/humanity/jquery-ui.css

Le-Frog

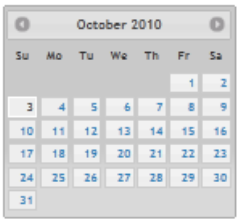
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/mint-choc/jquery-ui.css

---

Overcast

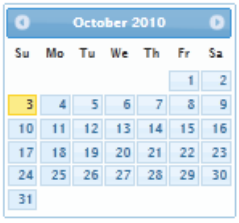


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/pepper-grinder/jquery-ui.css

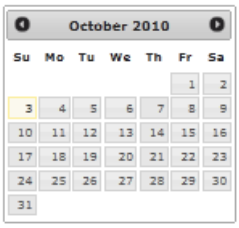
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/start/jquery-ui.css

---

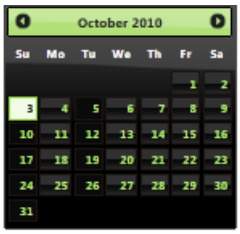
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/swanky-purse/jquery-ui.css

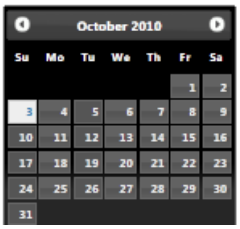
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/trontastic/jquery-ui.css

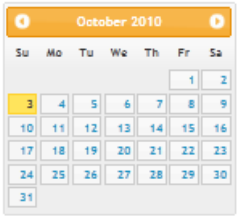
---

UI-Darkness



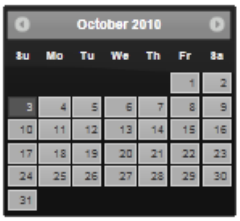
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.1/themes/vader/jquery-ui.css

jQuery UI 1.11.0 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.11.0 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/jquery-ui.min.js

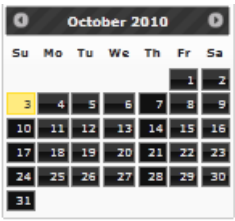
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/black-tie/jquery-ui.css

Blitzer

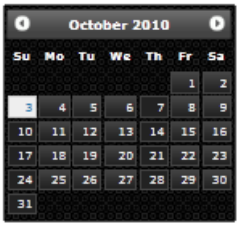
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/cupertino/jquery-ui.css

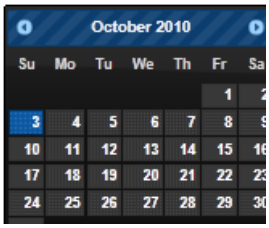
---

Dark-Hive



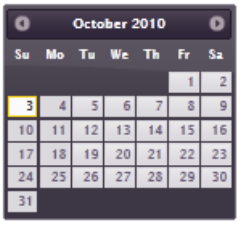
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/dot-luv/jquery-ui.css

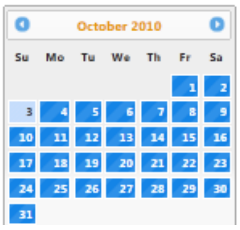
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/eggplant/jquery-ui.css

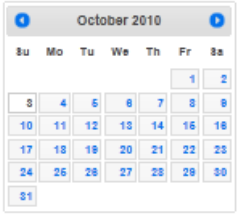
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/excite-bike/jquery-ui.css

Flick



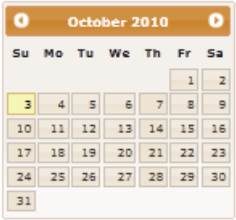
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/humanity/jquery-ui.css

Le-Frog

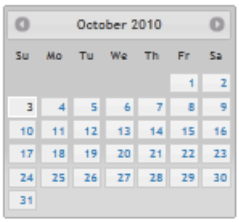
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/mint-choc/jquery-ui.css

---

Overcast

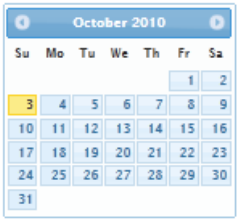


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/pepper-grinder/jquery-ui.css

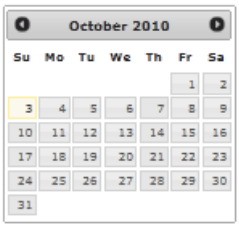
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/start/jquery-ui.css

---

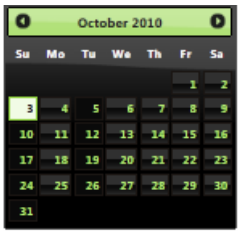
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/swanky-purse/jquery-ui.css

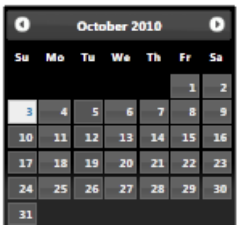
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/trontastic/jquery-ui.css

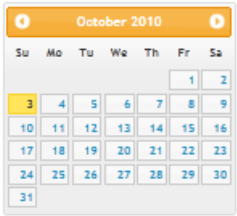
---

UI-Darkness



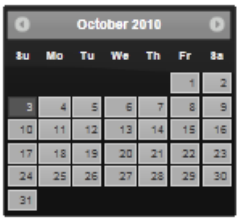
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.0/themes/vader/jquery-ui.css

jQuery UI 1.11.3 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.11.3 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/jquery-ui.min.js

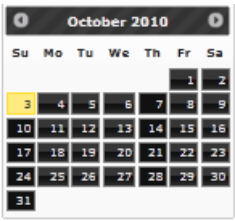
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/black-tie/jquery-ui.css

Blitzer

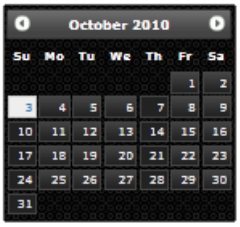
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/cupertino/jquery-ui.css

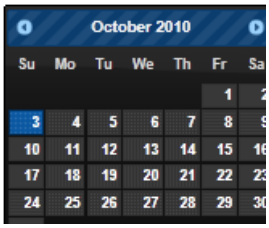
---

Dark-Hive



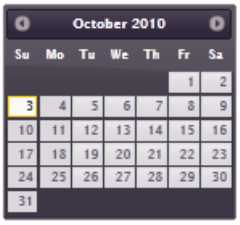
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/dot-luv/jquery-ui.css

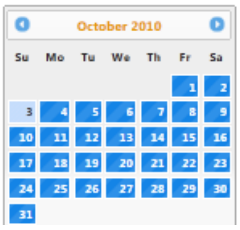
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/eggplant/jquery-ui.css

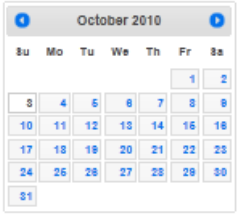
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/excite-bike/jquery-ui.css

Flick



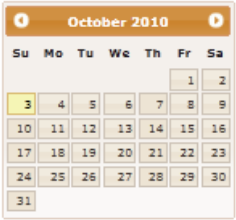
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/humanity/jquery-ui.css

Le-Frog

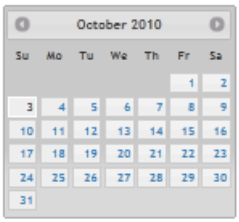
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/mint-choc/jquery-ui.css

---

Overcast

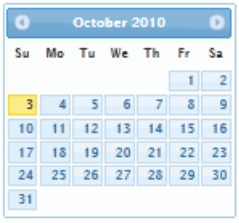


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/pepper-grinder/jquery-ui.css

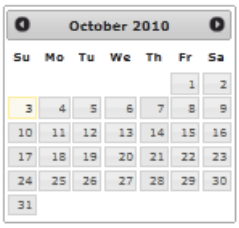
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/start/jquery-ui.css

---

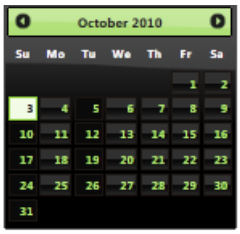
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/swanky-purse/jquery-ui.css

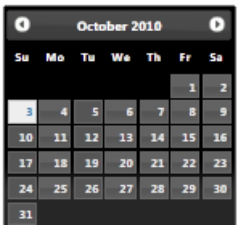
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/trontastic/jquery-ui.css

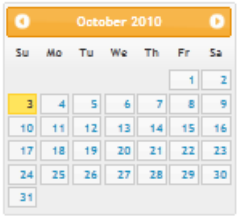
---

UI-Darkness



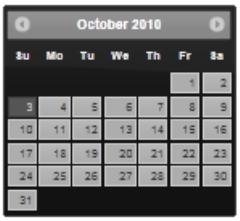
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.3/themes/vader/jquery-ui.css

jQuery UI 1.11.4 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.11.4 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/jquery-ui.min.js

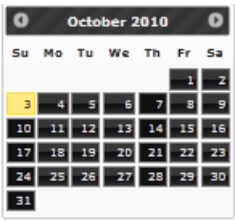
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/black-tie/jquery-ui.css

Blitzer

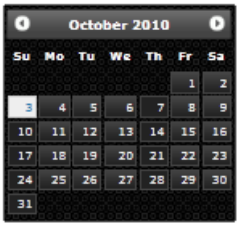
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/cupertino/jquery-ui.css

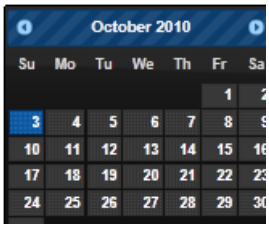
---

Dark-Hive



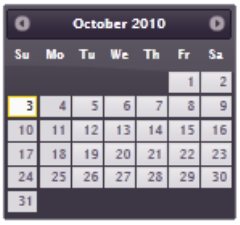
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/dot-luv/jquery-ui.css

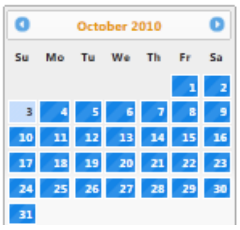
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/eggplant/jquery-ui.css

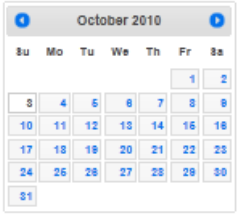
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/excite-bike/jquery-ui.css

Flick



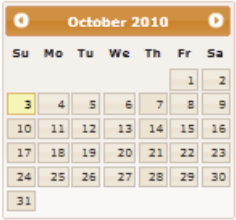
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/humanity/jquery-ui.css

Le-Frog

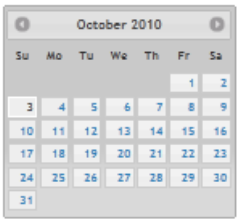
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/mint-choc/jquery-ui.css

---

Overcast

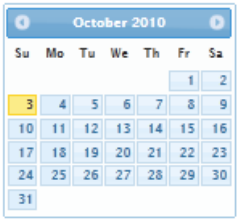


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/pepper-grinder/jquery-ui.css

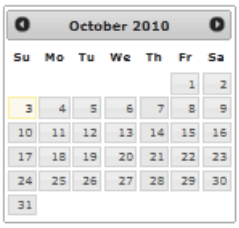
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/start/jquery-ui.css

---

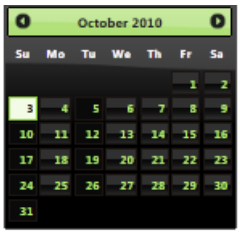
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/swanky-purse/jquery-ui.css

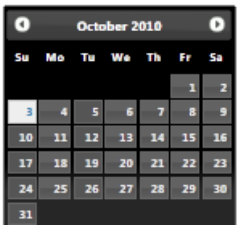
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/trontastic/jquery-ui.css

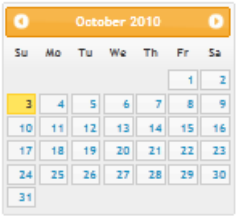
---

UI-Darkness



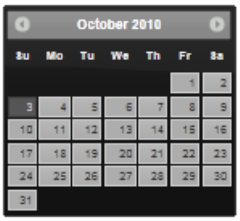
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.11.4/themes/vader/jquery-ui.css

jQuery UI 1.12.0 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.12.0 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/jquery-ui.min.js

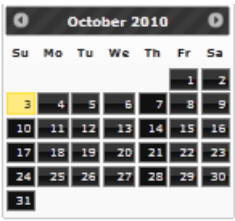
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/black-tie/jquery-ui.css

Blitzer

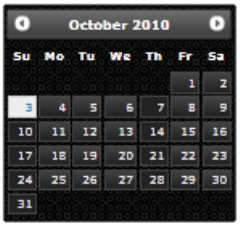
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/cupertino/jquery-ui.css

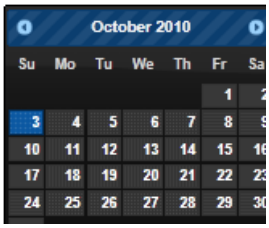
---

Dark-Hive



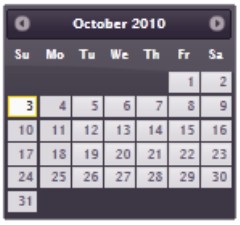
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/dot-luv/jquery-ui.css

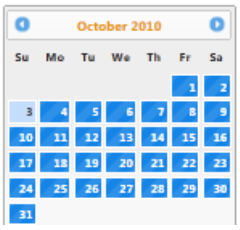
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/eggplant/jquery-ui.css

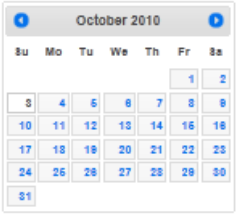
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/excite-bike/jquery-ui.css

Flick



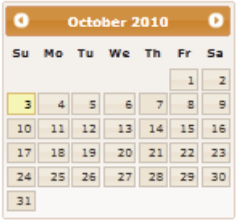
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/humanity/jquery-ui.css

Le-Frog

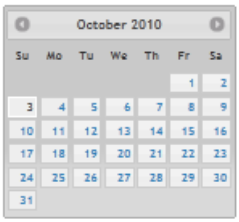
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/mint-choc/jquery-ui.css

---

Overcast

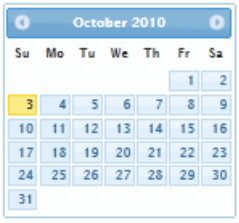


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/pepper-grinder/jquery-ui.css

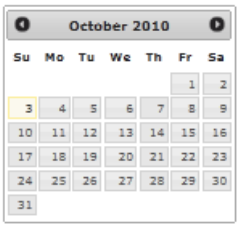
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/start/jquery-ui.css

---

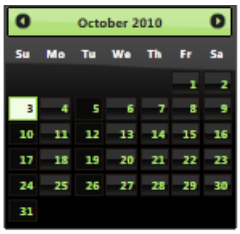
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/swanky-purse/jquery-ui.css

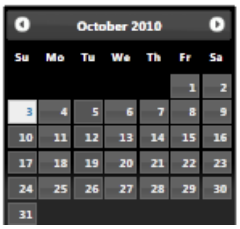
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/trontastic/jquery-ui.css

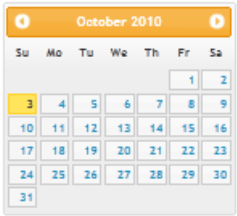
---

UI-Darkness



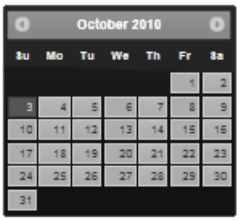
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.0/themes/vader/jquery-ui.css

jQuery UI 1.12.1 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery UI 1.12.1 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/jquery-ui.js
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/jquery-ui.min.js

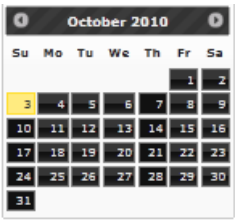
Themes

All of the standard jQuery UI themes are hosted on this CDN.

Click any of the theme images to view that theme on jQuery UI ThemeRoller.

Click any of the input fields to select the full URL to the theme's CDN hosted CSS file.

Black Tie



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/black-tie/jquery-ui.css

Blitzer

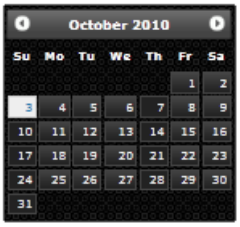
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/blitzer/jquery-ui.css

Cupertino

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/cupertino/jquery-ui.css

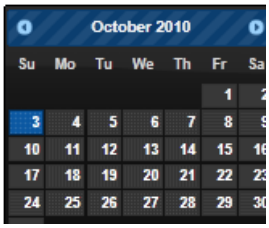
---

Dark-Hive



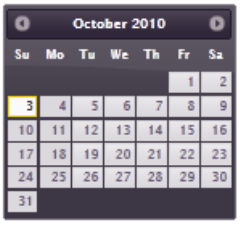
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/dark-hive/jquery-ui.css

Dot-Luv



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/dot-luv/jquery-ui.css

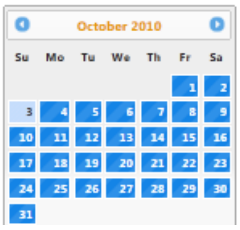
Eggplant



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/eggplant/jquery-ui.css

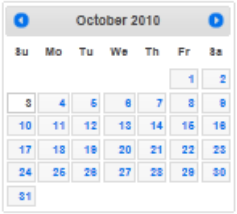
---

Excite-Bike



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/excite-bike/jquery-ui.css

Flick



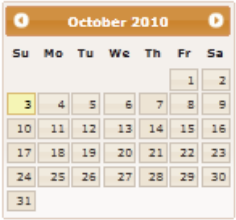
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/flick/jquery-ui.css

Hot-Sneaks

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/hot-sneaks/jquery-ui.css

---

Humanity



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/humanity/jquery-ui.css

Le-Frog

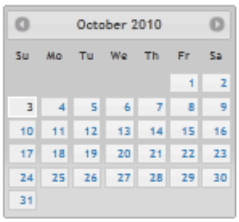
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/le-frog/jquery-ui.css

Mint-Choc

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/mint-choc/jquery-ui.css

---

Overcast

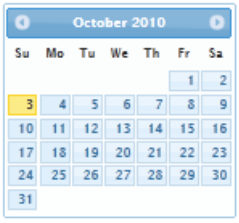


- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/overcast/jquery-ui.css

Pepper-Grinder

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/pepper-grinder/jquery-ui.css

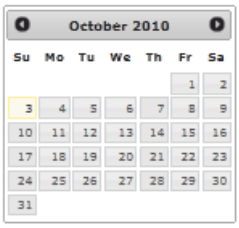
Redmond



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/redmond/jquery-ui.css

---

Smoothness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/smoothness/jquery-ui.css

South-Street

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/south-street/jquery-ui.css

Start

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/start/jquery-ui.css

---

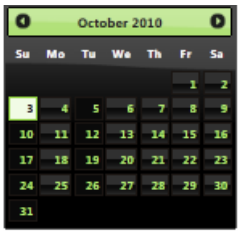
Sunny

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/sunny/jquery-ui.css

Swanky-Purse

- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/swanky-purse/jquery-ui.css

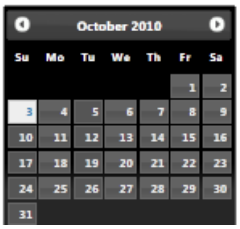
Trontastic



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/trontastic/jquery-ui.css

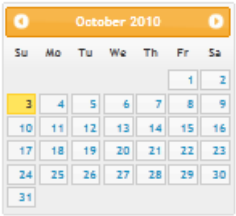
---

UI-Darkness



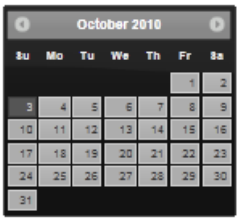
- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/ui-darkness/jquery-ui.css

UI-Lightness



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/ui-lightness/jquery-ui.css

Vader



- https://ajax.aspnetcdn.com/ajax/jquery.ui/1.12.1/themes/vader/jquery-ui.css

jQuery Validate

2/20/2020 • 2 minutes to read • Edit Online

- jQuery Validation 1.17.0
- jQuery Validation 1.16.0
- jQuery Validation 1.15.1
- jQuery Validation 1.15.0
- jQuery Validation 1.14.0
- jQuery Validation 1.13.1
- jQuery Validation 1.13.0
- jQuery Validation 1.12.0
- jQuery Validation 1.11.1
- jQuery Validation 1.11.0
- jQuery Validation 1.10.0
- jquery.validate version 1.9
- jquery.validate version 1.8.1
- jquery.validate version 1.8
- jquery.validate version 1.7
- jQuery Validate 1.6
- jQuery Validate 1.5.5

jQuery Validation 1.13.0

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.13.0/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.13.0/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.13.0/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.13.0/additional-methods.min.js
- Localization Files at https://ajax.aspnetcdn.com/ajax/jquery.validate/1.13.0/localization/messages_##.js where ## is the loc code.

jQuery Validation 1.12.0

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.12.0/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.12.0/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.12.0/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.12.0/additional-methods.min.js
- Localization Files at https://ajax.aspnetcdn.com/ajax/jquery.validate/1.12.0/localization/messages_##.js where ## is the loc code.

jQuery Validation 1.11.1

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.11.1/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.11.1/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.11.1/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.11.1/additional-methods.min.js
- Localization Files at https://ajax.aspnetcdn.com/ajax/jquery.validate/1.11.1/localization/messages_##.js where ## is the loc code.

jQuery Validation 1.11.0

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.11.0/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.11.0/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.11.0/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.11.0/additional-methods.min.js
- Localization Files at https://ajax.aspnetcdn.com/ajax/jquery.validate/1.11.0/localization/messages_##.js where ## is the loc code.

jQuery Validation 1.10.0

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.10.0/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.10.0/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.10.0/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.10.0/additional-methods.min.js
- Localization Files at https://ajax.aspnetcdn.com/ajax/jquery.validate/1.10.0/localization/messages_##.js where ## is the loc code.

jquery.validate version 1.9

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.9/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.9/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.9/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.9/additional-methods.min.js
- Localization Files at https://ajax.aspnetcdn.com/ajax/jquery.validate/1.9/localization/messages_##.js where ## is the loc code.

jquery.validate version 1.8.1

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8.1/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8.1/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8.1/jquery.validate-vsdoc.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8.1/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8.1/additional-methods.min.js
- Localization Files at https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8.1/localization/messages_##.js where ## is the loc code.

jquery.validate version 1.8

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8/jquery.validate-vsdoc.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8/additional-methods.min.js
- Localization Files at https://ajax.aspnetcdn.com/ajax/jquery.validate/1.8/localization/messages_##.js where ## is the loc code.

jquery.validate version 1.7

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.7/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.7/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.7/jquery.validate.pack.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.7/additional-methods.js
- Localization Files

jQuery Validate 1.6

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/jQuery.Validate.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/jQuery.Validate.min.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/jQuery.Validate.pack.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_ar.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_bg.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_cn.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_cs.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_da.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_de.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_es.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_fa.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_fi.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_fr.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_hu.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_it.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_kk.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_nl.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_no.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_pl.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_ptbr.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_ptpt.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_ro.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_ru.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_se.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_sk.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_tr.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_tw.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/messages_ua.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/methods_de.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/methods_nl.js
- https://ajax.aspnetcdn.com/ajax/jQuery.Validate/1.6/localization/methods_pt.js

jQuery Validate 1.5.5

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.5.5/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.5.5/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.5.5/jquery.validate-vsdoc.js

jQuery Validation 1.13.1

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.13.1/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.13.1/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.13.1/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.13.1/additional-methods.min.js
- Localization Files at https://ajax.aspnetcdn.com/ajax/jquery.validate/1.13.1/localization/messages_##.js where ## is the loc code.

jQuery Validation 1.14.0

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/additional-methods.min.js
- Localization Files at https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/localization/messages_##.js where ## is the loc code.

jQuery Validation 1.15.0

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.15.0/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.15.0/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.15.0/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.15.0/additional-methods.min.js
- Localization Files at https://ajax.aspnetcdn.com/ajax/jquery.validate/1.15.0/localization/messages_##.js where ## is the loc code.

jQuery Validation 1.15.1

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.15.1/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.15.1/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.15.1/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.15.1/additional-methods.min.js
- Localization Files at https://ajax.aspnetcdn.com/ajax/jquery.validate/1.15.1/localization/messages_##.js where ## is the loc code.

jQuery Validation 1.16.0

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.16.0/jquery.validate.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.16.0/jquery.validate.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.16.0/additional-methods.js
- https://ajax.aspnetcdn.com/ajax/jquery.validate/1.16.0/additional-methods.min.js
- Localization Files at https://ajax.aspnetcdn.com/ajax/jquery.validate/1.16.0/localization/messages_##.js where ## is the loc code.

jQuery Mobile

2/20/2020 • 2 minutes to read • Edit Online

- jQuery Mobile 1.4.2 on the Microsoft Ajax CDN
- jQuery Mobile 1.4.1 on the Microsoft Ajax CDN
- jQuery Mobile 1.4.0 on the Microsoft Ajax CDN
- jQuery Mobile 1.3.2 on the Microsoft Ajax CDN
- jQuery Mobile 1.3.1 on the Microsoft Ajax CDN
- jQuery Mobile 1.3.0 on the Microsoft Ajax CDN
- jQuery Mobile 1.2.0 on the Microsoft Ajax CDN
- jQuery Mobile 1.1.2 on the Microsoft Ajax CDN
- jQuery Mobile 1.1.1 on the Microsoft Ajax CDN
- jQuery Mobile 1.1.0 on the Microsoft Ajax CDN
- jQuery Mobile 1.1.0 RC2 on the Microsoft Ajax CDN
- jQuery Mobile 1.0.1 on the Microsoft Ajax CDN
- jQuery Mobile 1.0 on the Microsoft Ajax CDN
- jQuery Mobile 1.0 RC2 on the Microsoft Ajax CDN
- jQuery Mobile 1.0 RC1 on the Microsoft Ajax CDN
- jQuery Mobile 1.0 Beta 3 on the Microsoft Ajax CDN
- jQuery Mobile 1.4.5 on the Microsoft Ajax CDN

jQuery Mobile 1.4.2 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.4.2 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.2/jquery.mobile.structure-1.4.2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.2/jquery.mobile.structure-1.4.2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.2/jquery.mobile.theme-1.4.2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.2/jquery.mobile.theme-1.4.2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.2/jquery.mobile-1.4.2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.2/jquery.mobile-1.4.2.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.2/jquery.mobile-1.4.2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.2/jquery.mobile-1.4.2.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.2/images/ajax-loader.gif
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.2/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.2/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.2/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.2/images/icons-36-white.png

jQuery Mobile 1.4.1 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.4.1 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.1/jquery.mobile.structure-1.4.1.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.1/jquery.mobile.structure-1.4.1.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.1/jquery.mobile.theme-1.4.1.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.1/jquery.mobile.theme-1.4.1.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.1/jquery.mobile-1.4.1.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.1/jquery.mobile-1.4.1.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.1/jquery.mobile-1.4.1.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.1/jquery.mobile-1.4.1.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.1/images/ajax-loader.gif
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.1/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.1/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.1/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.1/images/icons-36-white.png

jQuery Mobile 1.4.0 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.4.0 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.0/jquery.mobile.structure-1.4.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.0/jquery.mobile.structure-1.4.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.0/jquery.mobile.theme-1.4.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.0/jquery.mobile.theme-1.4.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.0/jquery.mobile-1.4.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.0/jquery.mobile-1.4.0.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.0/jquery.mobile-1.4.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.0/jquery.mobile-1.4.0.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.0/images/ajax-loader.gif
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.0/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.0/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.0/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.0/images/icons-36-white.png

jQuery Mobile 1.3.2 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.3.2 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.2/jquery.mobile.structure-1.3.2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.2/jquery.mobile.structure-1.3.2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.2/jquery.mobile.theme-1.3.2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.2/jquery.mobile.theme-1.3.2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.2/jquery.mobile-1.3.2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.2/jquery.mobile-1.3.2.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.2/jquery.mobile-1.3.2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.2/jquery.mobile-1.3.2.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.2/images/ajax-loader.gif
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.2/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.2/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.2/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.2/images/icons-36-white.png

jQuery Mobile 1.3.1 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.3.1 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/jquery.mobile.structure-1.3.1.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/jquery.mobile.structure-1.3.1.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/jquery.mobile.theme-1.3.1.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/jquery.mobile.theme-1.3.1.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/jquery.mobile-1.3.1.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/jquery.mobile-1.3.1.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/jquery.mobile-1.3.1.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/jquery.mobile-1.3.1.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/images/ajax-loader.gif
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/images/ajax-loader.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.1/images/icons-36-white.png

jQuery Mobile 1.3.0 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.3.0 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/jquery.mobile.structure-1.3.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/jquery.mobile.structure-1.3.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/jquery.mobile.theme-1.3.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/jquery.mobile.theme-1.3.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/jquery.mobile-1.3.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/jquery.mobile-1.3.0.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/jquery.mobile-1.3.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/jquery.mobile-1.3.0.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/images/ajax-loader.gif
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/images/ajax-loader.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.3.0/images/icons-36-white.png

jQuery Mobile 1.2.0 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.2.0 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/jquery.mobile.structure-1.2.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/jquery.mobile.structure-1.2.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/jquery.mobile.theme-1.2.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/jquery.mobile.theme-1.2.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/jquery.mobile-1.2.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/jquery.mobile-1.2.0.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/jquery.mobile-1.2.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/jquery.mobile-1.2.0.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/images/ajax-loader.gif
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/images/ajax-loader.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.2.0/images/icons-36-white.png

jQuery Mobile 1.1.2 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.1.2 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/jquery.mobile.structure-1.1.2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/jquery.mobile.structure-1.1.2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/jquery.mobile.theme-1.1.2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/jquery.mobile.theme-1.1.2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/jquery.mobile-1.1.2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/jquery.mobile-1.1.2.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/jquery.mobile-1.1.2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/jquery.mobile-1.1.2.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/images/ajax-loader.gif
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/images/ajax-loader.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.2/images/icons-36-white.png

jQuery Mobile 1.1.1 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.1.1 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/jquery.mobile.structure-1.1.1.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/jquery.mobile.structure-1.1.1.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/jquery.mobile.theme-1.1.1.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/jquery.mobile.theme-1.1.1.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/jquery.mobile-1.1.1.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/jquery.mobile-1.1.1.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/jquery.mobile-1.1.1.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/jquery.mobile-1.1.1.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/images/ajax-loader.gif
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/images/ajax-loader.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.1/images/icons-36-white.png

jQuery Mobile 1.1.0 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.1.0 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/jquery.mobile.structure-1.1.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/jquery.mobile.structure-1.1.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/jquery.mobile.theme-1.1.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/jquery.mobile.theme-1.1.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/jquery.mobile-1.1.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/jquery.mobile-1.1.0.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/jquery.mobile-1.1.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/jquery.mobile-1.1.0.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/images/ajax-loader.gif
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/images/ajax-loader.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0/imagesicons-36-white.png

jQuery Mobile 1.1.0 RC2 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.1.0 RC 2 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/jquery.mobile.structure-1.1.0-rc.2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/jquery.mobile.structure-1.1.0-rc.2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/jquery.mobile.theme-1.1.0-rc.2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/jquery.mobile.theme-1.1.0-rc.2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/jquery.mobile-1.1.0-rc.2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/jquery.mobile-1.1.0-rc.2.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/jquery.mobile-1.1.0-rc.2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/jquery.mobile-1.1.0-rc.2.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/images/ajax-loader.gif
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/images/ajax-loader.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.1.0rc2/imagesicons-36-white.png

jQuery Mobile 1.0.1 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.0.1 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0.1/jquery.mobile-1.0.1.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0.1/jquery.mobile-1.0.1.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0.1/jquery.mobile-1.0.1.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0.1/jquery.mobile-1.0.1.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0.1/jquery.mobile.structure-1.0.1.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0.1/jquery.mobile.structure-1.0.1.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0.1/images/ajax-loader.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0.1/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0.1/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0.1/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0.1/images/icons-36-white.png

jQuery Mobile 1.0 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.0 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0/jquery.mobile-1.0.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0/jquery.mobile-1.0.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0/jquery.mobile-1.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0/jquery.mobile-1.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0/jquery.mobile.structure-1.0.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0/jquery.mobile.structure-1.0.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0/images/ajax-loader.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0/images/icons-36-white.png

jQuery Mobile 1.0 RC2 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.0 RC 2 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc2/jquery.mobile-1.0rc2.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc2/jquery.mobile-1.0rc2.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc2/jquery.mobile-1.0rc2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc2/jquery.mobile-1.0rc2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc2/jquery.mobile.structure-1.0rc2.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc2/jquery.mobile.structure-1.0rc2.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc2/images/ajax-loader.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc2/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc2/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc2/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc2/images/icons-36-white.png

jQuery Mobile 1.0 RC1 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.0 RC 1 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc1/jquery.mobile-1.0rc1.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc1/jquery.mobile-1.0rc1.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc1/jquery.mobile-1.0rc1.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc1/jquery.mobile-1.0rc1.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc1/images/ajax-loader.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc1/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc1/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc1/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0rc1/images/icons-36-white.png

jQuery Mobile 1.0 Beta 3 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.0 Beta 3 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0b3/jquery.mobile-1.0b3.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0b3/jquery.mobile-1.0b3.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0b3/jquery.mobile-1.0b3.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0b3/jquery.mobile-1.0b3.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0b3/images/ajax-loader.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0b3/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0b3/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0b3/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0b3/images/icons-36-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.0b3/images/icons-search-black.png

jQuery Mobile 1.4.5 on the Microsoft Ajax CDN

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Mobile 1.4.5 library includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.5/jquery.mobile.structure-1.4.5.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.5/jquery.mobile.structure-1.4.5.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.5/jquery.mobile.theme-1.4.5.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.5/jquery.mobile.theme-1.4.5.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.5/jquery.mobile-1.4.5.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.5/jquery.mobile-1.4.5.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.5/jquery.mobile-1.4.5.min.css
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.5/jquery.mobile-1.4.5.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.5/images/ajax-loader.gif
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.5/images/icons-18-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.5/images/icons-18-white.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.5/images/icons-36-black.png
- https://ajax.aspnetcdn.com/ajax/jquery.mobile/1.4.5/images/icons-36-white.png

jQuery Templates

2/20/2020 • 2 minutes to read • Edit Online

- jQuery Templates Beta 1

jQuery Templates Beta 1

2/20/2020 • 2 minutes to read • Edit Online

The jQuery Templates plugin includes the following files:

- https://ajax.aspnetcdn.com/ajax/jquery.templates/beta1/jquery.tmpl.js
- https://ajax.aspnetcdn.com/ajax/jquery.templates/beta1/jquery.tmpl.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.templates/beta1/jquery.tmplPlus.js
- https://ajax.aspnetcdn.com/ajax/jquery.templates/beta1/jquery.tmplPlus.min.js

jQuery Cycle

2/20/2020 • 2 minutes to read • Edit Online

- jQuery Cycle 2.99
- jQuery Cycle 2.94
- jQuery Cycle 2.88

jQuery Cycle 2.99

2/20/2020 • 2 minutes to read • Edit Online

The following jQuery Cycle files are hosted on this CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.99/jquery.cycle.all.js — The full version of the jQuery Cycle plugin. Includes more than two dozen special effects and all of the options.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.99/jquery.cycle.all.min.js — The minified version of the file above. Use this version for production applications to improve your website performance.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.99/jquery.cycle.js — This version of the jQuery Cycle plugin is smaller and contains only the "fade" transition effect.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.99/jquery.cycle.min.js — The minified version of the file above. Use this version for production applications to improve your website performance.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.99/jquery.cycle.lite.js — An even smaller version of the jQuery Cycle plugin that contains only the "fade" transition without all of the options.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.99/jquery.cycle.lite.min.js — The minified version of the file above. Use this version for production applications to improve your website performance.

jQuery Cycle 2.94

2/20/2020 • 2 minutes to read • Edit Online

The following jQuery Cycle files are hosted on this CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.94/jquery.cycle.all.js — The full version of the jQuery Cycle plugin. Includes all the more than two dozen special effects and all of the options.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.94/jquery.cycle.all.min.js — The minified version of the file above. Use this version for production applications to improve your website performance.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.94/jquery.cycle.js — This version of the jQuery Cycle plugin is smaller and contains only the "fade" transition effect.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.94/jquery.cycle.min.js — The minified version of the file above. Use this version for production applications to improve your website performance.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.94/jquery.cycle.lite.js — An even smaller version of the jQuery Cycle plugin that contains only the "fade" transition without all of the options.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.94/jquery.cycle.lite.min.js — The minified version of the file above. Use this version for production applications to improve your website performance.

jQuery Cycle 2.88

2/20/2020 • 2 minutes to read • Edit Online

The following jQuery Cycle files are hosted on this CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.88/jquery.cycle.all.js — The full version of the jQuery Cycle plugin. Includes all the more than two dozen special effects and all of the options.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.88/jquery.cycle.all.min.js — The minified version of the file above. Use this version for production applications to improve your website performance.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.88/jquery.cycle.js — This version of the jQuery Cycle plugin is smaller and contains only the "fade" transition effect.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.88/jquery.cycle.min.js — The minified version of the file above. Use this version for production applications to improve your website performance.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.88/jquery.cycle.lite.js — An even smaller version of the jQuery Cycle plugin that contains only the "fade" transition without all of the options.
- https://ajax.aspnetcdn.com/ajax/jquery.cycle/2.88/jquery.cycle.lite.min.js — The minified version of the file above. Use this version for production applications to improve your website performance.

jQuery DataTables

2/20/2020 • 2 minutes to read • Edit Online

- jQuery DataTables 1.9.4
- jQuery DataTables 1.9.3
- jQuery DataTables 1.9.2
- jQuery DataTables 1.9.1
- jQuery DataTables 1.9.0
- jQuery DataTables 1.8.2
- jQuery DataTables 1.10.4
- jQuery DataTables 1.10.5

jQuery DataTables 1.9.4

2/20/2020 • 2 minutes to read • Edit Online

The following files for the jQuery DataTables plugins are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/jquery.dataTables.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/jquery.dataTables.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/css/jquery.dataTables.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/css/jquery.dataTables_themeroller.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/images/back_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/images/back_enabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/images/back_enabled_hover.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/images/forward_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/images/forward_enabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/images/forward_enabled_hover.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/images/sort_asc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/images/sort_asc_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/images/sort_both.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/images/sort_desc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.4/images/sort_desc_disabled.png

jQuery DataTables 1.9.3

2/20/2020 • 2 minutes to read • Edit Online

The following files for the jQuery DataTables plugins are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/jquery.dataTables.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/jquery.dataTables.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/css/jquery.dataTables.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/css/jquery.dataTables_themeroller.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/images/back_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/images/back_enabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/images/back_enabled_hover.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/images/forward_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/images/forward_enabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/images/forward_enabled_hover.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/images/sort_asc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/images/sort_asc_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/images/sort_both.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/images/sort_desc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.3/images/sort_desc_disabled.png

jQuery DataTables 1.9.2

2/20/2020 • 2 minutes to read • Edit Online

The following files for the jQuery DataTables plugins are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/jquery.dataTables.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/jquery.dataTables.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/css/jquery.dataTables.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/css/jquery.dataTables_themeroller.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/images/back_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/images/back_enabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/images/back_enabled_hover.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/images/forward_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/images/forward_enabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/images/forward_enabled_hover.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/images/sort_asc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/images/sort_asc_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/images/sort_both.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/images/sort_desc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.2/images/sort_desc_disabled.png

jQuery DataTables 1.9.1

2/20/2020 • 2 minutes to read • Edit Online

The following files for the jQuery DataTables plugins are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/jquery.dataTables.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/jquery.dataTables.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/css/jquery.dataTables.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/css/jquery.dataTables_themeroller.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/images/back_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/images/back_enabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/images/back_enabled_hover.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/images/forward_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/images/forward_enabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/images/forward_enabled_hover.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/images/sort_asc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/images/sort_asc_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/images/sort_both.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/images/sort_desc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.1/images/sort_desc_disabled.png

jQuery DataTables 1.9.0

2/20/2020 • 2 minutes to read • Edit Online

The following files for the jQuery DataTables plugins are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.0/jquery.dataTables.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.0/jquery.dataTables.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.0/css/jquery.dataTables.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.0/css/jquery.dataTables_themeroller.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.0/images/back_disabled.jpg
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.0/images/back_enabled.jpg
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.0/images/forward_disabled.jpg
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.0/images/forward_enabled.jpg
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.0/images/sort_asc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.0/images/sort_asc_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.0/images/sort_both.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.0/images/sort_desc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.9.0/images/sort_desc_disabled.png

jQuery DataTables 1.8.2

2/20/2020 • 2 minutes to read • Edit Online

The following files for the jQuery DataTables plugins are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.8.2/jquery.dataTables.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.8.2/jquery.dataTables.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.8.2/css/jquery.dataTables.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.8.2/css/jquery.dataTables_themeroller.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.8.2/images/back_disabled.jpg
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.8.2/images/back_enabled.jpg
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.8.2/images/forward_disabled.jpg
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.8.2/images/forward_enabled.jpg
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.8.2/images/sort_asc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.8.2/images/sort_asc_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.8.2/images/sort_both.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.8.2/images/sort_desc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.8.2/images/sort_desc_disabled.png

jQuery DataTables 1.10.4

2/20/2020 • 2 minutes to read • Edit Online

The following files for the jQuery DataTables plugins are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/jquery.dataTables.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/jquery.dataTables.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/css/jquery.dataTables.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/css/jquery.dataTables_themeroller.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/images/back_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/images/back_enabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/images/back_enabled_hover.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/images/forward_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/images/forward_enabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/images/forward_enabled_hover.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/images/sort_asc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/images/sort_asc_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/images/sort_both.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/images/sort_desc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.4/images/sort_desc_disabled.png

jQuery DataTables 1.10.5

2/20/2020 • 2 minutes to read • Edit Online

The following files for the jQuery DataTables plugins are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/jquery.dataTables.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/jquery.dataTables.min.js
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/css/jquery.dataTables.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/css/jquery.dataTables_themeroller.css
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/images/back_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/images/back_enabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/images/back_enabled_hover.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/images/forward_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/images/forward_enabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/images/forward_enabled_hover.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/images/sort_asc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/images/sort_asc_disabled.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/images/sort_both.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/images/sort_desc.png
- https://ajax.aspnetcdn.com/ajax/jquery.dataTables/1.10.5/images/sort_desc_disabled.png

ASP.NET Web Forms and Ajax 4

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/4.0/1/Date.HijriCalendar.debug.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/Date.HijriCalendar.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/Date.UmAlQuraCalendar.debug.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/Date.UmAlQuraCalendar.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/DetailsView.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/Focus.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/GridView.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/Menu.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MenuStandards.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjax.debug.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjax.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxApplicationServices.debug.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxApplicationServices.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxComponentModel.debug.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxComponentModel.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxCore.debug.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxCore.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxGlobalization.debug.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxGlobalization.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxHistory.debug.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxHistory.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxNetwork.debug.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxNetwork.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxSerialization.debug.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxSerialization.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxTimer.debug.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxTimer.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxWebForms.debug.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxWebForms.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxWebServices.debug.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjaxWebServices.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/SmartNav.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/TreeView.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/WebForms.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/WebParts.js
- https://ajax.aspnetcdn.com/ajax/4.0/1/WebUIValidation.js

Note: The globalization scripts, such as fr-FR.js, can be found in the following folder:

- https://ajax.aspnetcdn.com/ajax/4.0/1/globalization/

ASP.NET Ajax 3.5

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/3.5/MicrosoftAjax.debug.js
- https://ajax.aspnetcdn.com/ajax/3.5/MicrosoftAjax.js

ASP.NET Web Forms and Ajax 4.5.2

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/4.5.2/1/Date.HijriCalendar.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/Date.HijriCalendar.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/Date.UmAlQuraCalendar.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/Date.UmAlQuraCalendar.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/DetailsView.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/Focus.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/GridView.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/Menu.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MenuStandards.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjax.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjax.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxApplicationServices.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxApplicationServices.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxComponentModel.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxComponentModel.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxCore.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxCore.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxGlobalization.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxGlobalization.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxHistory.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxHistory.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxNetwork.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxNetwork.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxSerialization.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxSerialization.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxTimer.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxTimer.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxWebForms.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxWebForms.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxWebServices.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/MicrosoftAjaxWebServices.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/SmartNav.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/TreeView.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/WebForms.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/WebParts.js
- https://ajax.aspnetcdn.com/ajax/4.5.2/1/WebUIValidation.js

Note: The globalization scripts, such as fr-FR.js, can be found in the following folder:

- https://ajax.aspnetcdn.com/ajax/4.5.2/1/globalization/

ASP.NET Web Forms and Ajax 4.5.1

2/20/2020 • 2 minutes to read • Edit Online

The following files are hosted on the CDN:

- https://ajax.aspnetcdn.com/ajax/4.5.1/1/Date.HijriCalendar.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/Date.HijriCalendar.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/Date.UmAlQuraCalendar.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/Date.UmAlQuraCalendar.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/DetailsView.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/Focus.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/GridView.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/Menu.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MenuStandards.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjax.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjax.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxApplicationServices.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxApplicationServices.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxComponentModel.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxComponentModel.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxCore.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxCore.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxGlobalization.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxGlobalization.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxHistory.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxHistory.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxNetwork.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxNetwork.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxSerialization.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxSerialization.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxTimer.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxTimer.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxWebForms.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxWebForms.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxWebServices.debug.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/MicrosoftAjaxWebServices.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/SmartNav.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/TreeView.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/WebForms.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/WebParts.js
- https://ajax.aspnetcdn.com/ajax/4.5.1/1/WebUIValidation.js

Note: The globalization scripts, such as fr-FR.js, can be found in the following folder:

- https://ajax.aspnetcdn.com/ajax/4.5.1/1/globalization/

Web Development Best Practices

2/20/2020 • 2 minutes to read • Edit Online

- What not to do in ASP.NET, and what to do instead
- Async and Await
- Building Real World Cloud Apps with Azure
- Hands on Lab: Maintainable Azure Websites: Managing Change and Scale

What not to do in ASP.NET, and what to do instead

2/20/2020 • 11 minutes to read • Edit Online

This topic describes several common mistakes people make within ASP.NET web projects. It provides recommendations for what you should do to avoid these common mistakes. It is based on a presentation by **Damian Edwards** at Norwegian Developers Conference.

Disclaimer

This topic is not intended as a complete guide to ensure your application is secure and efficient. You still need to follow best practices for security and performance that are not outlined in this topic. It only suggests how to avoid common mistakes related to .NET classes and processes.

Overview

This topic contains the following sections:

- Standards Compliance
  - Control Adapters
  - Style Properties on Controls
  - Page and Control Callbacks
  - Browser Capability Detection
- Security
  - Request Validation
  - Cookieless Forms Authentication and Session
  - EnableViewStateMac
  - Medium Trust
  - <appSettings>
  - UrlPathEncode
- Reliability and Performance
  - PreSendRequestHeaders and PreSendRequestContent
  - Asynchronous Page Events with Web Forms
  - Fire-and-Forget Work
  - Request Entity Body
  - Response.Redirect and Response.End
  - EnableViewState and ViewStateMode
  - SqlMembershipProvider
  - Long Running Requests (>110 seconds)

Standards Compliance

Control adapters

Recommendation: Stop using control adapters for adaptive rendering, and instead use CSS media queries and standards-compliant HTML.

Controls Adapters were introduced in .NET 2.0 to render presentation code that was customized for different

devices and environments. Now, this adaptive rendering can be accomplished with CSS and HTML. You should stop using Control Adapters and convert any existing adapters to CSS and HTML.

For more information, see Media Queries and How To: Add Mobile Pages to Your ASP.NET Web Forms / MVC Application.

Style properties on controls

Recommendation: Stop setting style values in the control markup, and instead set formatting values in CSS stylesheets.

Web server controls contain dozens of properties which can be used to set in-line style properties. For example, the ForeColor property sets the color of the text for a control. You can accomplish this same effect more efficiently through CSS stylesheets. Stylesheets enable you to centralize style values and avoid setting these values throughout your application.

The following example shows a CSS class the sets text to red.

```
.CautionRow {
    color: red;
}
```

The next example shows how to dynamically apply the CSS class.

```
protected void CustomersGridView_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.Cells[2].Text == "Unconfirmed")
    {
        e.Row.CssClass = "CautionRow";
    }
}
```

Page and control callbacks

Recommendation: Stop using page and control callbacks, and instead use any of the following: AJAX, UpdatePanel, MVC action methods, Web API, or SignalR.

In earlier versions of ASP.NET, Page and Control callback methods enabled you to update part of the web page without refreshing an entire page. You can now accomplish partial-page updates through AJAX, UpdatePanel, MVC, Web API or SignalR. You should stop using callback methods because they can cause issues with friendly URLs and routing. By default, controls do not enable callback methods, but if you enabled this feature in a control, you should disable it.

Browser capability detection

Recommendation: Stop using static browser capability detection, and instead use dynamic feature detection.

In earlier versions of ASP.NET, the supported features for each browser were stored in an XML file. Detecting feature support through a static lookup is not the best approach. Now, you can dynamically detect a browser's supported features by using a feature detection framework, such as Modernizr. Feature detection determines support by attempting to use a method or property and then checking to see if the browser produced the desired result. By default, Modernizr is included in the Web application templates.

Security

Request validation

Recommendation: Validate user input, and encode output from users.

Request validation is a feature of ASP.NET that inspects each request and stops the request if a perceived threat is

found. Do not depend on request validation for securing your application against cross-site scripting attacks. Instead, validate all input from users and encode the output. In some limited cases, you can use regular expressions to validate the input, but in more complicated cases you should validate user input by using .NET classes that determine if the value matches allowed values.

The following example shows how to use a static method in the Uri class to determine whether the Uri provided by a user is valid.

```
var isValidUri = Uri.IsWellFormedUriString(passedUri, UriKind.Absolute);
```

However, to sufficiently verify the Uri, you should also check to make sure it specifies `http` or `https`. The following example uses instance methods to verify that the Uri is valid.

```
var uriToVerify = new Uri(passedUri);
var isValidUri = uriToVerify.IsWellFormedOriginalString();
var isValidScheme = uriToVerify.Scheme == "http" || uriToVerify.Scheme == "https";
```

Before rendering user input as HTML or including user input in a SQL query, encode the values to ensure malicious code is not included.

You can HTML encode the value in markup with the <%: %> syntax, as shown below.

```
<span><%: userInput %></span>
```

Or, in Razor syntax, you can HTML encode with @, as shown below.

```
<span>@userInput</span>
```

The next example shows how to HTML encode a value in code-behind.

```
var encodedInput = Server.HtmlEncode(userInput);
```

To safely encode a value for SQL commands, use command parameters such as the SqlParameter.

Cookieless forms authentication and session

Recommendation: Require cookies.

Passing authentication information in the query string is not secure. Therefore, require cookies when your application includes authentication. If your cookie stores sensitive information, consider requiring SSL for the cookie.

The following example shows how to specify in the Web.config file that Forms Authentication requires a cookie that is transmitted over SSL.

```
<authentication mode="Forms">
  <forms loginUrl="member_login.aspx"
    cookieless="UseCookies"
    requireSSL="true"
    path="/MyApplication" />
</authentication>
```

EnableViewStateMac

Recommendation: Never set to false.

By default, EnableViewStateMac is set to true. Even if your application is not using view state, do not set EnableViewStateMac to false. Setting this value to false will make your application vulnerable to cross-site scripting.

Starting with ASP.NET 4.5.2, the runtime enforces **EnableViewStateMac=true**. Even if you set it to false, the runtime ignores this value and proceeds with the value set to true. For more information, see ASP.NET 4.5.2 and EnableViewStateMac.

The following example shows how to set EnableViewStateMac to true. You do not need to actually set this value to true because it is true by default. However, if you have set it to false on any page in your application, you must immediately correct this value.

```
<%@ Page language="C#" EnableViewStateMac="true" %>
```

Medium trust

Recommendation: Do not depend on Medium Trust (or any other trust level) as a security boundary.

Partial trust does not adequately protect your application and should not be used. Instead, use Full Trust, and isolate untrusted applications in separate application pools. Also, run each application pool under a unique identity. For more information, see ASP.NET Partial Trust does not guarantee application isolation.

<appSettings>

Recommendation: Do not disable security settings in <appSettings> element.

The appSettings element contains many values which are required for security updates. You should not change or disable these values. If you must disable these values when deploying an update, immediately re-enable after completing deployment.

For details, see ASP.NET appSettings Element.

UrlPathEncode

Recommendation: Use UrlEncode instead.

The UrlPathEncode method was added to the .NET Framework to resolve a very specific browser compatibility problem. It does not adequately encode a URL, and does not protect your application from cross-site scripting. You should never use it in your application. Instead, use UrlEncode.

The following example shows how to pass an encoded URL as a query string parameter for a hyperlink control.

```
string destinationURL = "http://www.contoso.com/default.aspx?user=test";
NextPage.NavigateUrl = "~/Finish?url=" + Server.UrlEncode(destinationURL);
```

Reliability and performance

PreSendRequestHeaders and PreSendRequestContent

Recommendation: Do not use these events with managed modules. Instead, write a native IIS module to perform the required task. See Creating Native-Code HTTP Modules.

You can use the PreSendRequestHeaders and PreSendRequestContent events with native IIS modules.

> **WARNING**
> Do not use `PreSendRequestHeaders` and `PreSendRequestContent` with managed modules that implement `IHttpModule`. Setting these properties can cause issues with asynchronous requests. The combination of Application Requested Routing (ARR) and websockets might lead to access violation exceptions that can cause w3wp to crash. For example, iiscore!W3_CONTEXT_BASE::GetIsLastNotification+68 in iiscore.dll has caused an access violation exception (0xC0000005).

Asynchronous page events with web forms

Recommendation: In Web Forms, avoid writing async void methods for Page lifecycle events, and instead use Page.RegisterAsyncTask for asynchronous code.

When you mark a page event with **async** and **void**, you cannot determine when the asynchronous code has finished. Instead, use Page.RegisterAsyncTask to run the asynchronous code in a way that enables you to track its completion.

The following example shows a button click handler that contains asynchronous code. This example includes reading a string value asynchronously, which is provided only as a simplified example of an asynchronous task and not as a recommended practice.

```
protected void StartAsync_Click(object sender, EventArgs e)
{
    Page.RegisterAsyncTask(new PageAsyncTask(async() =>
    {
        string stringToRead = "Long text value";

        using (StringReader reader = new StringReader(stringToRead))
        {
            string readText = await reader.ReadToEndAsync();
            Result.Text = readText;
        }
    }));
}
```

If you are using asynchronous Tasks, set the Http runtime target framework to 4.5 (or later) in the Web.config file. Setting the target framework to 4.5 turns on the new synchronization context that was added in .NET 4.5. This value is set by default in new projects in Visual Studio, but is not be set if you are working with an existing project.

```
<system.web>
    <httpRuntime TargetFramework="4.5" />
</system.web>
```

Fire-and-forget work

Recommendation: When handling a request within ASP.NET, avoid launching fire-and-forget work (such calling the ThreadPool.QueueUserWorkItem method or creating a timer that repeatedly calls a delegate).

If your application has fire-and-forget work that runs within ASP.NET, your application can get out of sync. At any time, the app domain can be destroyed which means your ongoing process may no longer match the current state of the application.

You should move this type of work outside of ASP.NET. You can use a Web Jobs, Windows Service or a Worker role in Azure to perform ongoing work, and run that code from another process.

If you must perform this work within ASP.NET, you can add the Nuget package called WebBackgrounder to run the code.

Request entity body

Recommendation: Avoid reading Request.Form or Request.InputStream before the handler's execute event.

The earliest you should read from Request.Form or Request.InputStream is during the handler's execute event. In MVC, the Controller is the handler and the execute event is when the action method runs. In Web Forms, the Page is the handler and the execute event is when the Page.Init event fires. If you read the request entity body earlier than the execute event, you interfere with the processing of the request.

If you need to read the request entity body before the execute event, use either Request.GetBufferlessInputStream or Request.GetBufferedInputStream. When you use GetBufferlessInputStream, you get the raw stream from the request, and assume responsibility for processing the entire request. After calling GetBufferlessInputStream, Request.Form and Request.InputStream are not available because they have not been populated by ASP.NET. When you use GetBufferedInputStream, you get a copy of the stream from the request. Request.Form and Request.InputStream are still available later in the request because ASP.NET populates the other copy.

Response.Redirect and Response.End

Recommendation: Be aware of differences in how thread is handled after calling Response.Redirect(String).

The Response.Redirect(String) method calls the Response.End method. In a synchronous process, calling Request.Redirect causes the current thread to immediately abort. However, in an asynchronous process, calling Response.Redirect does not abort the current thread, so code execution continues for the request. In an asynchronous process, you must return the Task from the method to stop the code execution.

In an MVC project, you should not call Response.Redirect. Instead, return a RedirectResult.

EnableViewState and ViewStateMode

Recommendation: Use ViewStateMode, instead of EnableViewState, to provide granular control over which controls use view state.

When you set EnableViewState to false in the Page directive, view state is disabled for all controls within the page and cannot be enabled. If you want to enable view state for only certain controls in your page, set ViewStateMode to Disabled for the Page.

```
<%@ Page ViewStateMode="Disabled" . . . %>
```

Then, set ViewStateMode to Enabled on only the controls that actually need view state.

```
<asp:GridView ViewStateMode="Enabled" runat="server">
```

By enabling view state for only the controls that need it, you can shrink the size of the view state for your web pages.

SqlMembershipProvider

Recommendation: Use Universal Providers.

In the current project templates, SqlMembershipProvider has been replaced by ASP.NET Universal Providers, which is available as a NuGet package. If you are using SqlMembershipProvider in a project that was built with an earlier version of the templates, you should switch to Universal Providers. The Universal Providers work with all databases that are supported by Entity Framework.

For more information, see Introducing ASP.NET Universal Providers.

Long-running requests (>110 seconds)

Recommendation: Use WebSockets or SignalR for connected clients, and use asynchronous I/O operations.

Long-running requests can cause unpredictable results and poor performance in your web application. The default timeout setting for a request is 110 seconds. If you are using session state with a long-running request, ASP.NET

will release the lock on the Session object after 110 seconds. However, your application might be in the middle of an operation on the Session object when the lock is released, and the operation might not complete successfully. If a second request from the user is blocked while the first request is running, the second request might access the Session object in an inconsistent state.

If your application includes blocking (or synchronous) I/O operations, the application will be unresponsive.

To improve performance, use the asynchronous I/O operations in the .NET Framework. Also, use WebSockets or SignalR for connecting clients to the server. These features are designed to efficiently handle long-running requests.

Async and Await

2/20/2020 • 2 minutes to read • Edit Online

by Scott Hanselman

Watch video (5 minutes)

Building Real World Cloud Apps with Azure

2/20/2020 • 2 minutes to read • Edit Online

Building Real World Cloud Apps with Azure

- Introduction
- Automate Everything
- Source Control
- Continuous Integration and Delivery
- Web Development Best Practices
- Single Sign-On
- Data Storage Options
- Data Partitioning Strategies
- Unstructured Blob Storage
- Design to Survive Failures
- Monitoring and Telemetry
- Transient Fault Handling
- Distributed Caching
- Queue-Centric Work Pattern
- More Patterns and Guidance
- Appendix: The Fix It Sample Application

Building Real-World Cloud Apps with Azure

2/20/2020 • 7 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> This e-book walks you through a patterns-based approach to building real-world cloud solutions. The patterns apply to the development process as well as to architecture and coding practices.
>
> The content is based on a presentation developed by Scott Guthrie and delivered by him at the Norwegian Developers Conference (NDC) in June of 2013 (part 1, part 2), and at Microsoft Tech Ed Australia in September, 2013 (part 1, part 2). Many others updated and augmented the content while transitioning it from video to written form.

Intended Audience

Developers who are curious about developing for the cloud, considering a move to the cloud, or are new to cloud development will find here a concise overview of the most important concepts and practices they need to know. The concepts are illustrated with concrete examples, and each chapter links to other resources for more in-depth information. The examples and the links to additional resources are for Microsoft frameworks and services, but the principles illustrated apply to other web development frameworks and cloud environments as well.

Developers who are already developing for the cloud may find ideas here that will help make them more successful. Each chapter in the series can be read independently, so you can pick and choose topics that you're interested in.

Anyone who watched Scott Guthrie's *Building Real World Cloud Apps with Azure* presentation and wants more details and updated information will find that here.

Cloud development patterns

This e-book explains thirteen recommended patterns for cloud development. "Pattern" is used here in a broad sense to mean a recommended way to do things: how best to go about developing, designing, and coding cloud apps. These are key patterns which will help you "fall into the pit of success" if you follow them.

- Automate everything.
  - Use scripts to maximize efficiency and minimize errors in repetitive processes.
  - Demo: Azure management scripts.
- Source control.
  - Set up branching structure in source control to facilitate DevOps workflow.
  - Demo: add scripts to source control.
  - Demo: keep sensitive data out of source control.
  - Demo: use Git in Visual Studio.
- Continuous integration and delivery.
  - Automate build and deployment with each source control check-in.
- Web development best practices.

- Keep web tier stateless.
- Demo: scaling and auto-scaling in Web Apps in Azure App Service.
- Avoid session state.
- Use a CDN with a fallback when the CDN is unavailable.
- Use asynchronous programming model.
- Demo: async in ASP.NET MVC and Entity Framework.

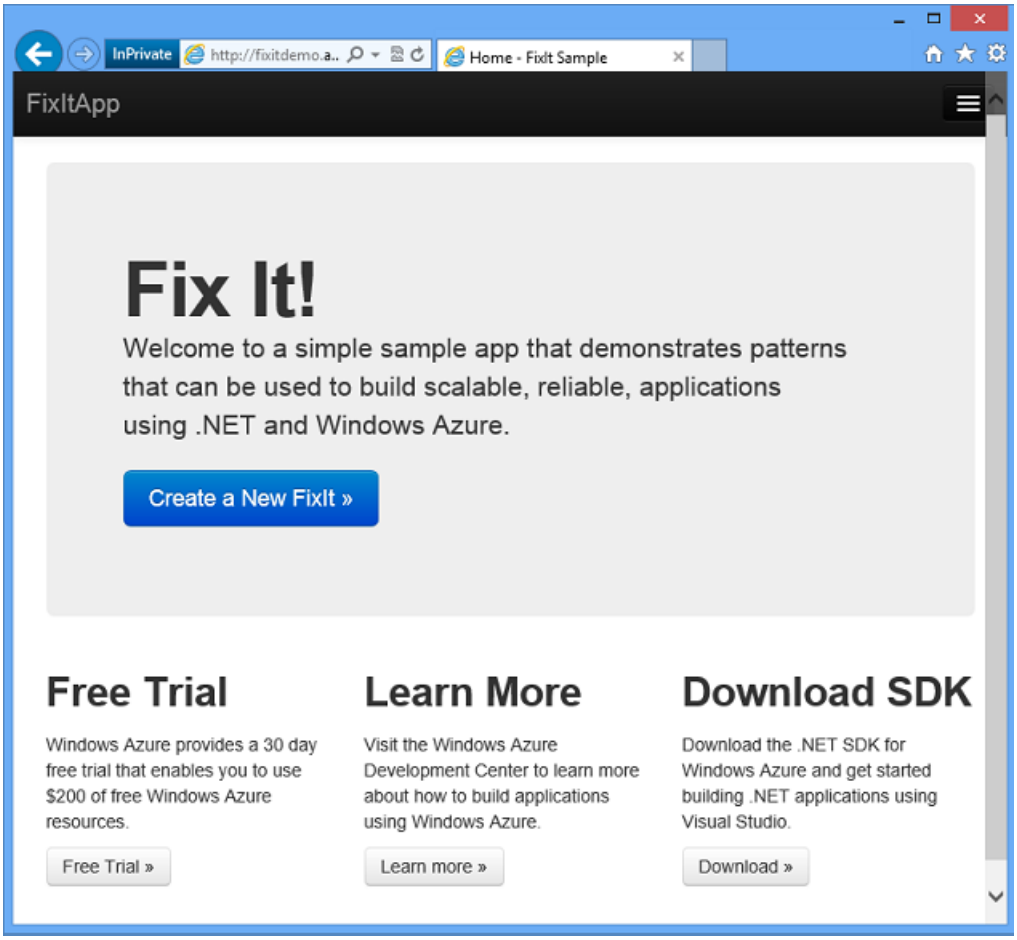
- Single sign-on.
  - Introduction to Azure Active Directory.
  - Demo: create an ASP.NET app that uses Azure Active Directory.
- Data storage options.
  - Types of data stores.
  - How to choose the right data store.
  - Demo: Azure SQL Database.
- Data partitioning strategies.
  - Partition data vertically, horizontally, or both to facilitate scaling a relational database.
- Unstructured blob storage.
  - Store files in the cloud by using the blob service.
  - Demo: using blob storage in the Fix It app.
- Design to survive failures.
  - Types of failures.
  - Failure Scope.
  - Understanding SLAs.
- Monitoring and telemetry.
  - Why you should both buy a telemetry app and write your own code to instrument your app.
  - Demo: New Relic for Azure
  - Demo: logging code in the Fix It app.
  - Demo: dependency injection in the Fix It app.
  - Demo: built-in logging support in Azure.
- Transient fault handling.
  - Use smart retry/back-off logic to mitigate the effect of transient failures.
  - Demo: retry/back-off in Entity Framework 6.
- Distributed caching.
  - Improve scalability and reduce database transaction costs by using distributed caching.
- Queue-centric work pattern.
  - Enable high availability and improve scalability by loosely coupling web and worker tiers.
  - Demo: Azure storage queues in the Fix It app.
- More cloud app patterns and guidance.
- Appendix: The Fix It Sample Application
  - Known Issues
  - Best Practices
  - How to download, build, run, and deploy.

These patterns apply to all cloud environments, but we'll illustrate them by using examples based on Microsoft technologies and services, such as Visual Studio, Team Foundation Service, ASP.NET, and Azure.

This remainder of this chapter introduces the Fix It sample application and the Web Apps in Azure App Service cloud environment that the Fix It app runs in.

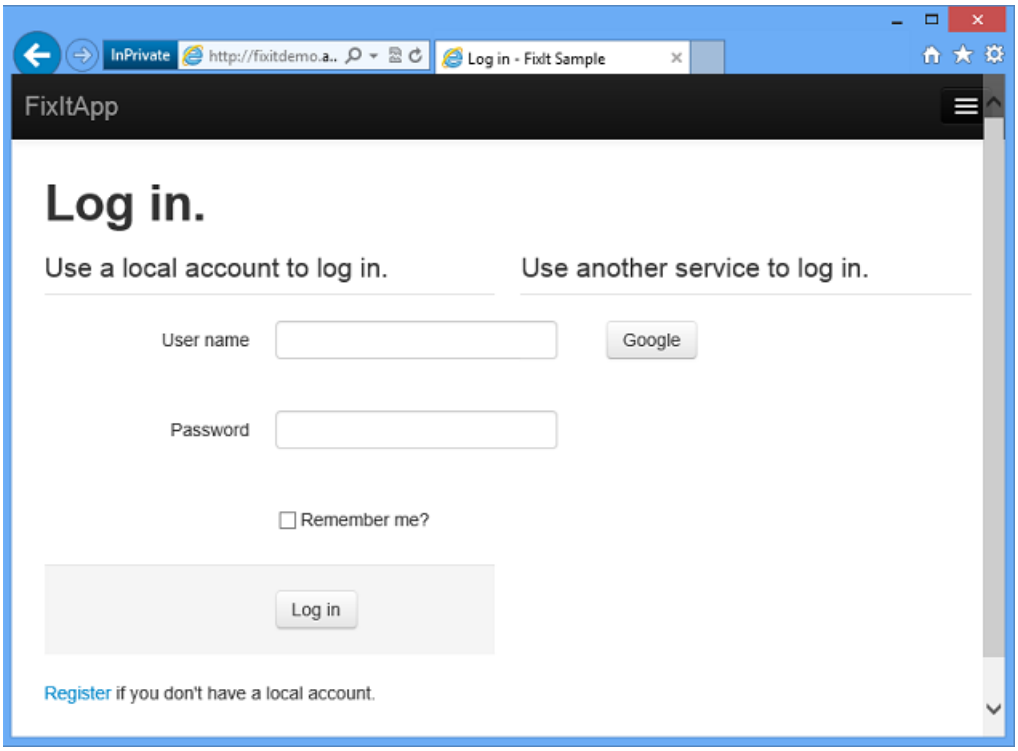
The Fix it sample application

Most of the screen shots and code examples shown in this e-book are based on the Fix It app originally developed by Scott Guthrie to demonstrate recommended cloud app development patterns and practices.

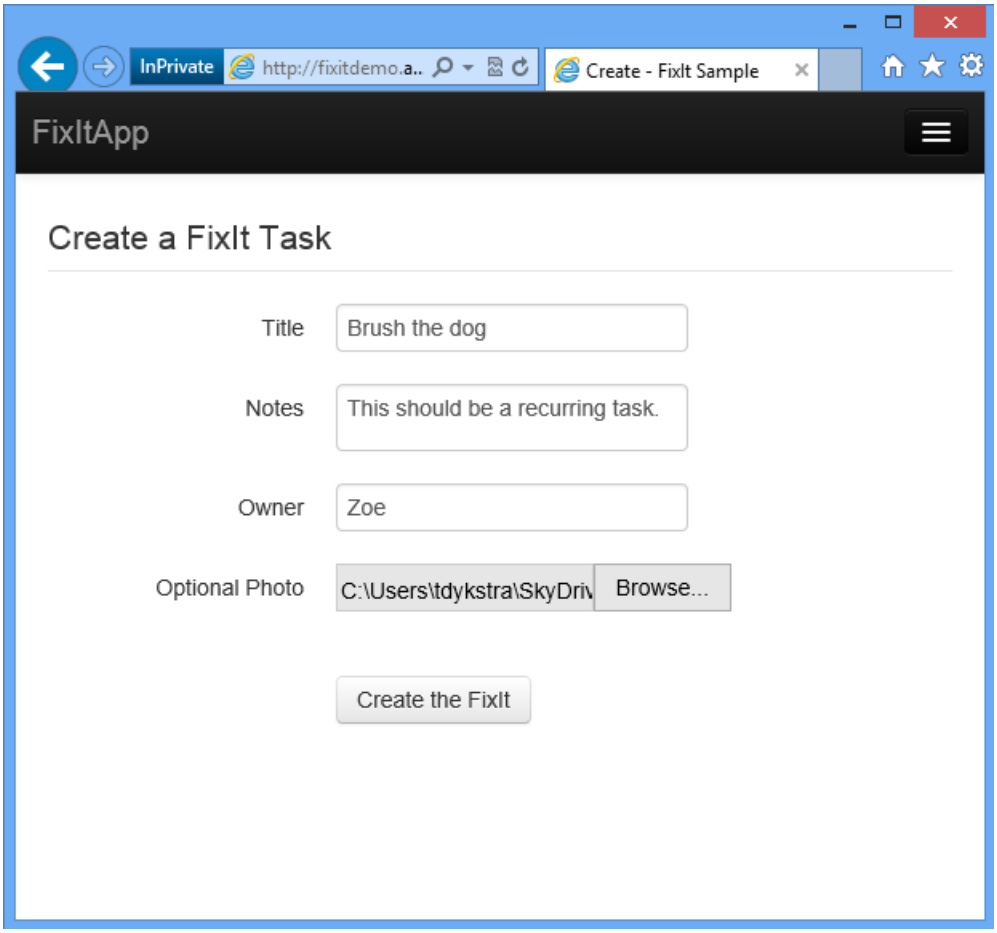


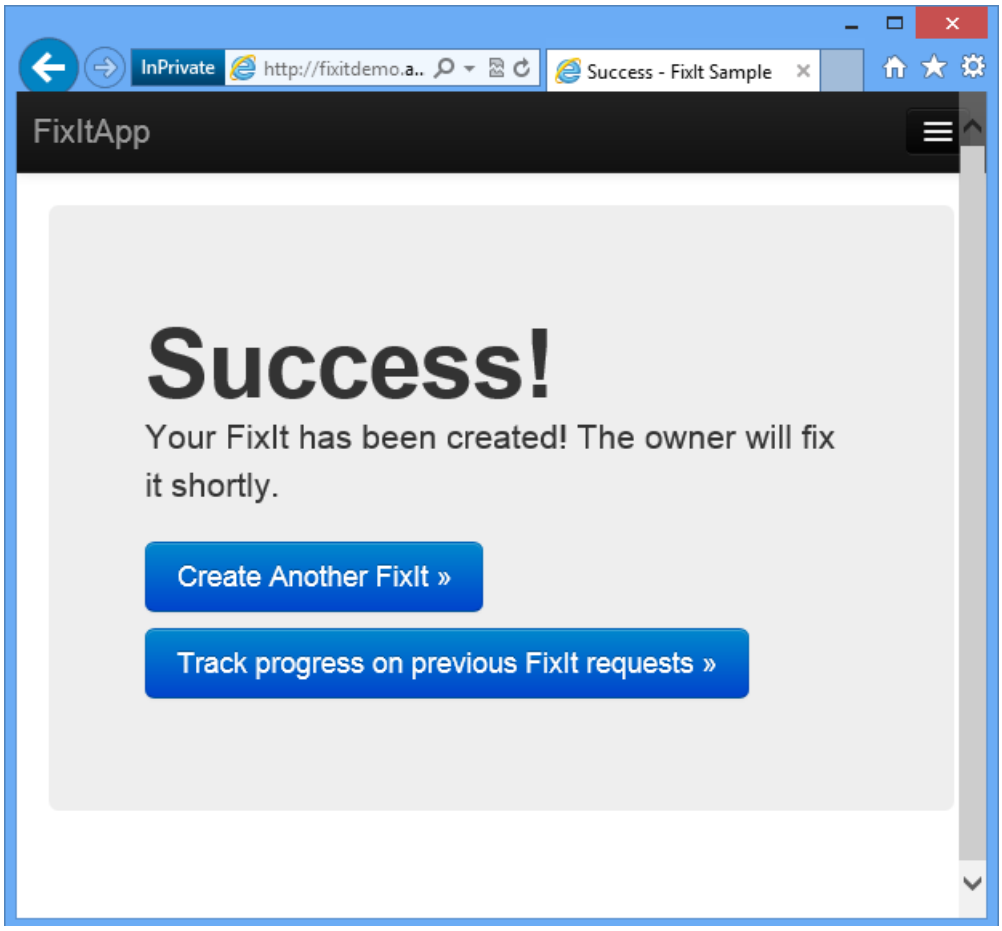
The sample app is a simple work item ticketing system. When you need something fixed, you create a ticket and assign it to someone, and others can log in and see the tickets assigned to them and mark tickets as completed when the work is done.

It's a standard Visual Studio web project. It is built on ASP.NET MVC and uses a SQL Server database. It can run locally in IIS Express and can be deployed to a Azure Web Site to run in the cloud. You can log in using forms authentication and a local database or by using a social provider such as Google. (Later we'll also show how to log in with an Active Directory organizational account.)



Once you're logged in you can create a ticket, assign it to someone, and upload a picture of what you want to get fixed.





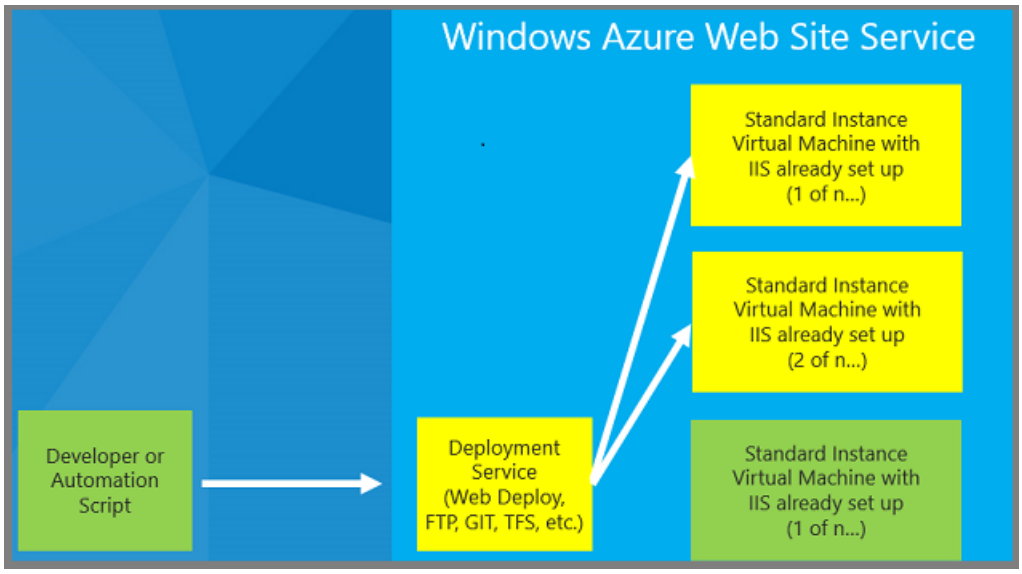
You can track the progress of work items you created, see tickets assigned to you, view ticket details, and mark items as completed.

This is a very simple app from a feature perspective, but you'll see how to build it so that it can scale to millions of users and will be resilient to things like database failures and connection terminations. You'll also see how to create an automated and agile development workflow, which enables you to start simple and make the app better and better by iterating the development cycle efficiently and quickly.

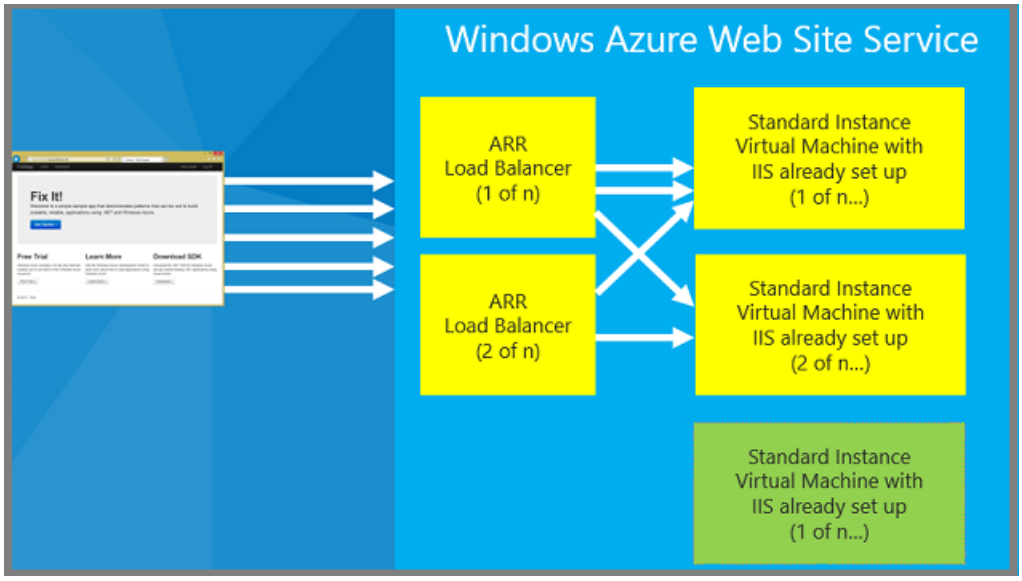
Web Apps in Azure App Service

The cloud environment used for the Fix It application is a service of Azure that we call Web Sites. This service is a way that you can host your own web app in Azure without having to create VMs and keep them updated, install and configure IIS, etc. We host your site on our VMs and automatically provide backup and recovery and other services for you. The Web Sites service works with ASP.NET, Node.js, PHP, and Python. It enables you to deploy very quickly using Visual Studio, Web Deploy, FTP, Git, or TFS. It's usually just a few seconds between the time you start a deployment and the time your update is available over the Internet. It's all free to get started, and you can scale up as your traffic grows.

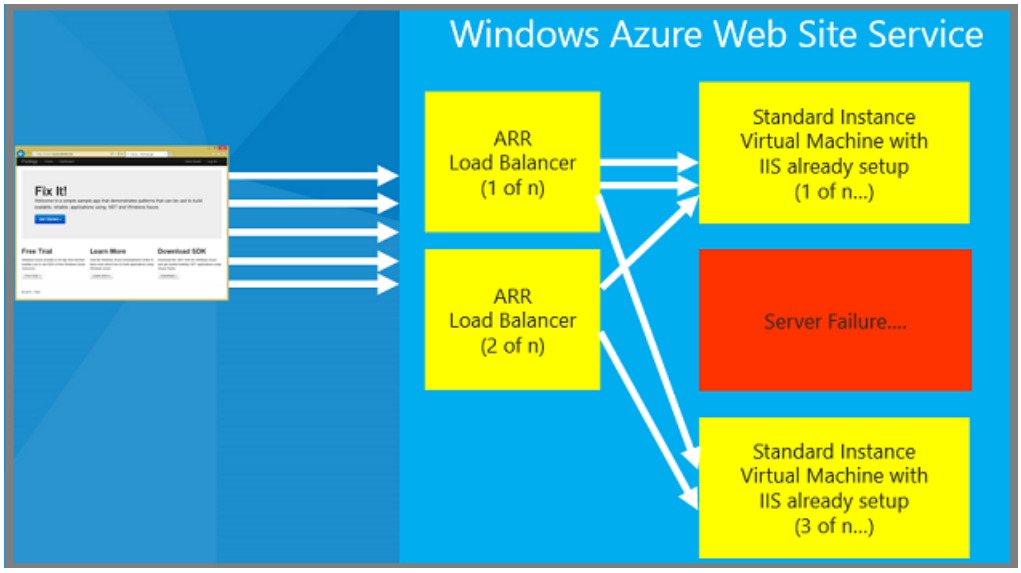
Behind the scenes, Web Apps in Azure App Service provides a lot of architectural components and features that you'd have to build yourself if you were going to host a web site using IIS on your own VMs. One component is a deployment end point that automatically configures IIS and installs your application on as many VMs as you want to run your site on.



When a user hits the web site, they don't hit the IIS VMs directly, they go through Application Request Routing (ARR) load balancers. You can use these with your own servers, but the advantage here is that they're set up for you automatically. They use a smart heuristic that takes into account factors such as session affinity, queue depth in IIS, and CPU usage on each machine to direct traffic to the VMs that host your web site.



If a machine goes down, Azure automatically pulls it from the rotation, spins up a new VM instance, and starts directing traffic to the new instance -- all with no down time for your application.



All of this takes place automatically. All you need to do is create a web site and deploy your application to it, using Windows PowerShell, Visual Studio, or the Azure management portal.

For a quick and easy step-by-step tutorial that shows how to create a web application in Visual Studio and deploy it to a Azure Web Site, see Get started with Azure and ASP.NET.

Summary

This introduction has provided a list of topics the book will cover, screenshots of the sample application, and a brief overview of the Web Apps in Azure App Service cloud environment. One of the great advantages of developing apps in and for the cloud is that it's easy to automate repetitive development tasks such as creating a test environment and deploying your code to it. How to do that is the subject of the next chapter.

Resources

For more information about the topics covered in this chapter, see the following resources.

Documentation:

- Web Apps in Azure App Service. Portal page for Azure documentation about Web Apps.
- Web Apps, Cloud Services, and VMs: When to use which? WAWS as shown in this chapter is just one of three ways you can run web apps in Azure. This article explains the differences between the three ways and gives guidance on how to choose which one is right for your scenario. Like Web Sites, Cloud Services is a PaaS feature of Azure. VMs are an IaaS feature. For an explanation of PaaS versus IaaS, see the Data Options chapter.

Videos:

- Scott Guthrie starts at Step 0 - What is the Azure Cloud OS?
- Web Sites Architecture - with Stefan Schackow.
- Azure Web Sites Internals with Nir Mashkowski.

NEXT

Automate Everything (Building Real-World Cloud Apps with Azure)

2/20/2020 • 14 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For an introduction to the e-book, see the first chapter.

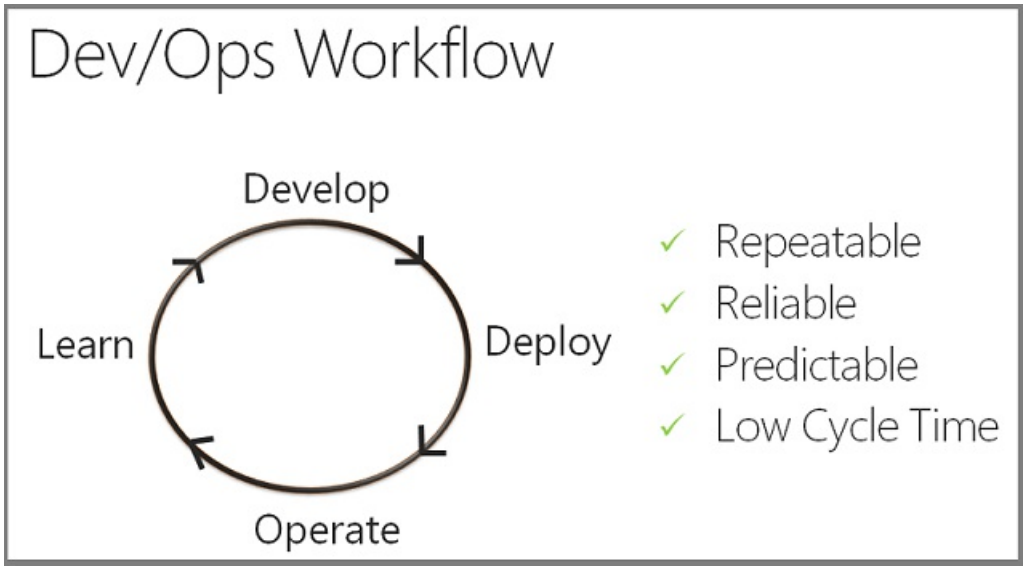
The first three patterns we'll look at actually apply to any software development project, but especially to cloud projects. This pattern is about automating development tasks. It's an important topic because manual processes are slow and error-prone; automating as many of them as possible helps set up a fast, reliable, and agile workflow. It's uniquely important for cloud development because you can easily automate many tasks that are difficult or impossible to automate in an on-premises environment. For example, you can set up whole test environments including new web server and back-end VMs, databases, blob storage (file storage), queues, etc.

DevOps Workflow

Increasingly you hear the term "DevOps." The term developed out of a recognition that you have to integrate development and operations tasks in order to develop software efficiently. The kind of workflow you want to enable is one in which you can develop an app, deploy it, learn from production usage of it, change it in response to what you've learned, and repeat the cycle quickly and reliably.

Some successful cloud development teams deploy multiple times a day to a live environment. The Azure team used to deploy a major update every 2-3 months, but now it releases minor updates every 2-3 days and major releases every 2-3 weeks. Getting into that cadence really helps you be responsive to customer feedback.

In order to do that, you have to enable a development and deployment cycle that is repeatable, reliable, predictable, and has low cycle time.



In other words, the period of time between when you have an idea for a feature and when the customers are using it and providing feedback must be as short as possible. The first three patterns – automate everything,

source control, and continuous integration and delivery -- are all about best practices that we recommend in order to enable that kind of process.

Azure management scripts

In the introduction to this e-book, you saw the web-based console, the Azure Management Portal. The management portal enables you to monitor and manage all of the resources that you have deployed on Azure. It's an easy way to create and delete services such as web apps and VMs, configure those services, monitor service operation, and so forth. It's a great tool, but using it is a manual process. If you're going to develop a production application of any size, and especially in a team environment, we recommend that you go through the portal UI in order to learn and explore Azure, and then automate the processes that you'll be doing repetitively.

Nearly everything that you can do manually in the management portal or from Visual Studio can also be done by calling the REST management API. You can write scripts using Windows PowerShell, or you can use an open source framework such as Chef or Puppet. You can also use the Bash command-line tool in a Mac or Linux environment. Azure has scripting APIs for all those different environments, and it has a .NET management API in case you want to write code instead of script.

For the Fix It app we've created some Windows PowerShell scripts that automate the processes of creating a test environment and deploying the project to that environment, and we'll review some of the contents of those scripts.

Environment creation script

The first script we'll look at is named *New-AzureWebsiteEnv.ps1*. It creates an Azure environment that you can deploy the Fix It app to for testing. The main tasks that this script performs are the following:

- Create a web app.
- Create a storage account. (Required for blobs and queues, as you'll see in later chapters.)
- Create a SQL Database server and two databases: an application database, and a membership database.
- Store settings in Azure that the app will use to access the storage account and databases.
- Create settings files that will be used to automate deployment.

Run the script

> **NOTE**
> This part of the chapter shows examples of scripts and the commands that you enter in order to run them. This a demo and doesn't provide everything you need to know in order to run the scripts. For step-by-step how-to-do-it instructions, see Appendix: The Fix It Sample Application.

To run a PowerShell script that manages Azure services you have to install the Azure PowerShell console and configure it to work with your Azure subscription. Once you're set up, you can run the Fix It environment creation script with a command like this one:

```
.\New-AzureWebsiteEnv.ps1 -Name <websitename> -SqlDatabasePassword <password>
```

The `Name` parameter specifies the name to be used when creating the database and storage accounts, and the `SqlDatabasePassword` parameter specifies the password for the admin account that will be created for SQL Database. There are other parameters you can use that we'll look at later.

```
Windows Azure PowerShell
PS C:\MyFixIt\Automation> .\New-AzureWebsiteEnv.ps1 -Name fixitdemo2 -SqlDatabasePassword P
VERBOSE: Checking for required files.
VERBOSE: Verifying that Windows Azure credentials in the Windows PowerShell session have no
VERBOSE: [Start] creating Windows Azure website environment: fixitdemo2
VERBOSE: Creating a Windows Azure website: fixitdemo2
VERBOSE: Creating a Windows Azure storage account: fixitdemo2storage
VERBOSE: [Start] creating fixitdemo2storage storage account West US location
VERBOSE: 3:02:44 PM - Begin Operation: New-AzureStorageAccount
VERBOSE: 3:03:16 PM - Completed Operation: New-AzureStorageAccount
VERBOSE: [Finish] creating fixitdemo2storage storage account in West US location
VERBOSE: 3:03:16 PM - Begin Operation: Get-AzureStorageKey
VERBOSE: 3:03:17 PM - Completed Operation: Get-AzureStorageKey
VERBOSE: Creating a Windows Azure database server and databases
VERBOSE: [Start] creating SQL Azure database server in West US location with username dbuse
VERBOSE: [Finish] creating SQL Azure database server vvqx9566n0 in location West US with us
Passw0rd1
VERBOSE: [start] Creating a new firewall rule fixitdemo2rule for the website in database se
VERBOSE: Calling 'Set Firewall Rule' Azure SQL REST API
VERBOSE: POST
https://management.database.windows.net:8443/aeb4ae60-b7cb-4f3d-966d-fa43b6607f30/servers/v
tdemo2rule?op=AutoDetectClientIP with 0-byte payload
VERBOSE: received 95-byte response of content type application/xml; charset=utf-8
VERBOSE: Editing firewall rule to add IP address range: 131.107.192.0 - 131.107.192.255
VERBOSE: [finish] Created firewall rule fixitdemo2rule for IP Address
VERBOSE: [Finish] creating fixitdemo2rule firewall rule in database server vvqx9566n0 for I
131.107.192.255
VERBOSE: [Start] creating firewall rule AllowAllAzureIP in database server vvqx9566n0 for I
0.0.0.0
VERBOSE: Adding firewall rule "AllowAllAzureIP" for Windows Azure Sql Database server "vvqx
VERBOSE: [Finish] creating AllowAllAzureIP firewall rule in database server vvqx9566n0 for
0.0.0.0
WARNING: The client model does not match the server model. While the current set of Cmdlets
 recommended that you update to the latest version of Windows Azure PowerShell to ensure fu
VERBOSE: [Start] creating database  appdb in database server vvqx9566n0
VERBOSE: Creating a new Windows Azure Sql Database.
VERBOSE: [Finish] creating database appdb in database server vvqx9566n0
VERBOSE: [Start] creating database memberdb in database server vvqx9566n0
VERBOSE: Creating a new Windows Azure Sql Database.
VERBOSE: [Finish] creating database memberdb in database server vvqx9566n0
VERBOSE: Creating database connection string for appdb in database server vvqx9566n0
VERBOSE: Creating database connection string for memberdb in database server vvqx9566n0
VERBOSE: Creating hash table to return...
VERBOSE: [Start] Adding settings to website: fixitdemo2
VERBOSE: Adding connection strings and storage account name/key to the new fixitdemo2 websi
VERBOSE: [Finish] Adding settings to website: fixitdemo2
VERBOSE: [Finish] creating Windows Azure environment: fixitdemo2
VERBOSE: [Begin] writing environment info to website-environment.xml
VERBOSE: C:\MyFixIt\Automation\website-environment.xml
VERBOSE: [Finish] writing environment info to website-environment.xml
VERBOSE: [Begin] generating fixitdemo2.pubxml file
VERBOSE: GET
https://management.core.windows.net:8443/aeb4ae60-b7cb-4f3d-966d-fa43b6607f30/services/WebS
ixitdemo2/publishxml with 0-byte payload
VERBOSE: received 2821-byte response of content type application/xml; charset=utf-8
VERBOSE: C:\MyFixIt\Automation\fixitdemo2.pubxml
VERBOSE: [Finish] generating fixitdemo2.pubxml file
VERBOSE: Script is complete.
Total time used (seconds): 65.4549211
PS C:\MyFixIt\Automation>
```

After the script finishes you can see in the management portal what was created. You'll find two databases:

sql databases

DATABASES SERVERS

NAME	STATUS	LOCATION	SUBSCRIPTION	SERVER	EDITION	MAX SIZE
appdb	Online	West US	Windows Azure...	m9qcm7sjzx	Web	1 GB
memberdb	Online	West US	Windows Azure...	m9qcm7sjzx	Web	1 GB

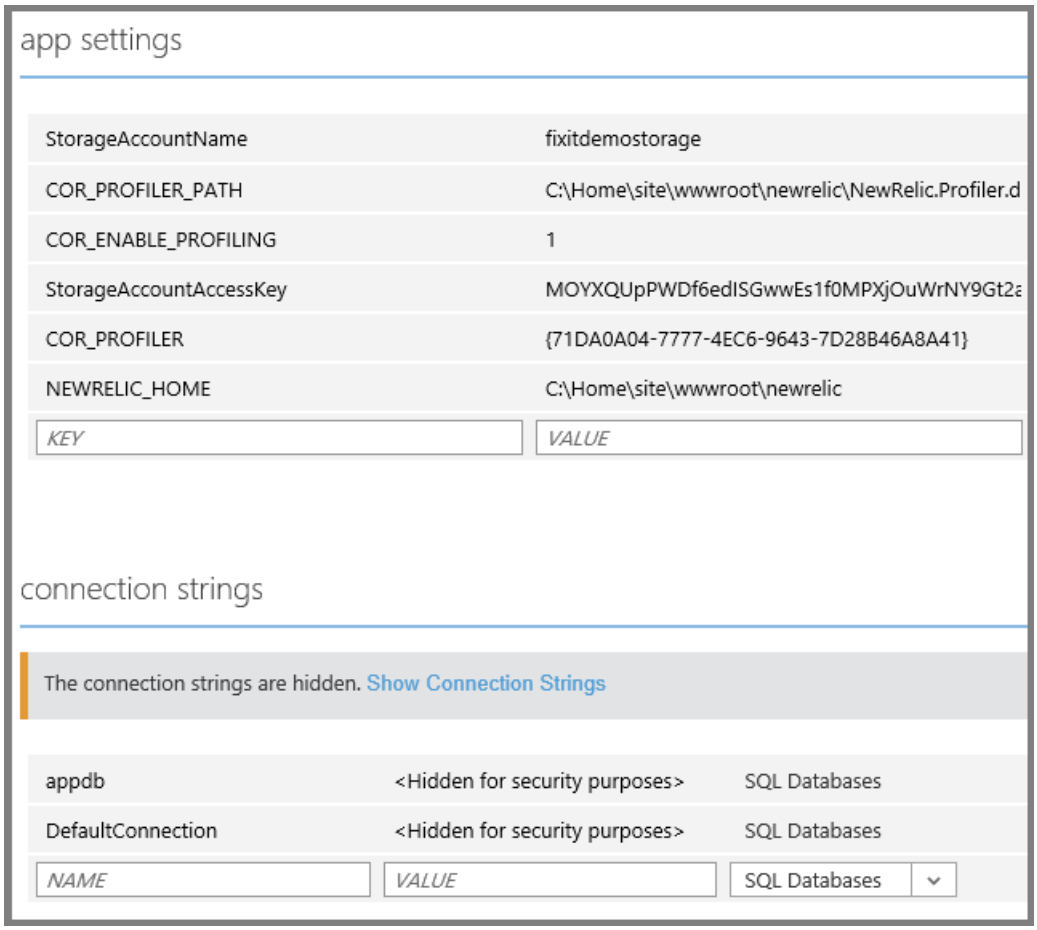
A storage account:

storage

NAME	STATUS	LOCATION	SUBSCRIPTION
fixitdemostorage	Online	West US	Windows Azure MSDN

And a web app:

On the **Configure** tab for the web app, you can see that it has the storage account settings and SQL database connection strings set up for the Fix It app.



The *Automation* folder now also contains a *<websitename>.pubxml* file. This file stores settings that MSBuild will use to deploy the application to the Azure environment that was just created. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>MSDeploy</WebPublishMethod>
    <SiteUrlToLaunchAfterPublish>http://fixitdemo.azurewebsites.net</SiteUrlToLaunchAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <MSDeployServiceURL>waws-prod-bay-003.publish.azurewebsites.windows.net:443</MSDeployServiceURL>
    <DeployIisAppPath>fixitdemo</DeployIisAppPath>
    <RemoteSitePhysicalPath />
    <SkipExtraFilesOnServer>True</SkipExtraFilesOnServer>
    <MSDeployPublishMethod>WMSVC</MSDeployPublishMethod>
    <EnableMSDeployBackup>True</EnableMSDeployBackup>
    <UserName>$fixitdemo</UserName>
    <PublishDatabaseSettings></PublishDatabaseSettings>
  </PropertyGroup>
</Project>
```

As you can see, the script has created a complete test environment, and the whole process is done in about 90

seconds.

If someone else on your team wants to create a test environment, they can just run the script. Not only is it fast, but also they can be confident that they are using an environment identical to the one you're using. You couldn't be quite as confident of that if everyone was setting things up manually by using the management portal UI.

A look at the scripts

There are actually three scripts that do this work. You call one from the command line and it automatically uses the other two to do some of the tasks:

- *New-AzureWebSiteEnv.ps1* is the main script.
  - *New-AzureStorage.ps1* creates the storage account.
  - *New-AzureSql.ps1* creates the databases.

Parameters in the main script

The main script, *New-AzureWebSiteEnv.ps1*, defines several parameters:

```
[CmdletBinding(PositionalBinding=$True)]
Param(
    [Parameter(Mandatory = $true)]
    [ValidatePattern("^[a-z0-9]*$")]
    [String]$Name,
    [String]$Location = "West US",
    [String]$SqlDatabaseUserName = "dbuser",
    [String]$SqlDatabasePassword,
    [String]$StartIPAddress,
    [String]$EndIPAddress
    )
```

Two parameters are required:

- The name of the web app that the script creates. (This is also used for the URL: `<name>.azurewebsites.net`.)
- The password for the new administrative user of the database server that the script creates.

Optional parameters enable you to specify the data center location (defaults to "West US"), database server administrator name (defaults to "dbuser"), and a firewall rule for the database server.

Create the web app

The first thing the script does is create the web app by calling the `New-AzureWebsite` cmdlet, passing in to it the web app name and location parameter values:

```
# Create a new website
$website = New-AzureWebsite -Name $Name -Location $Location -Verbose
```

Create the storage account

Then the main script runs the *New-AzureStorage.ps1* script, specifying "*<websitename>*storage" for the storage account name, and the same data center location as the web app.

```
$storageAccountName = $Name + "storage"

$storage = $scriptPath\New-AzureStorage.ps1" -Name $storageAccountName -Location $Location
```

New-AzureStorage.ps1 calls the `New-AzureStorageAccount` cmdlet to create the storage account, and it returns the account name and access key values. The application will need these values in order to access the blobs and queues in the storage account.

```
# Create a new storage account
New-AzureStorageAccount -StorageAccountName $Name -Location $Location -Verbose

# Get the access key of the storage account
$key = Get-AzureStorageKey -StorageAccountName $Name

# Generate the connection string of the storage account
$connectionString =
"BlobEndpoint=http://$Name.blob.core.windows.net/;QueueEndpoint=http://$Name.queue.core.windows.net/;TableEn
dpoint=http://$Name.table.core.windows.net/;AccountName=$Name;AccountKey=$primaryKey"

#Return a hashtable of storage account values
Return @{AccountName = $Name; AccessKey = $key.Primary; ConnectionString = $connectionString}
```

You might not always want to create a new storage account; you could enhance the script by adding a parameter that optionally directs it to use an existing storage account.

Create the databases

The main script then runs the database creation script, *New-AzureSql.ps1*, after setting up default database and firewall rule names:

```
$sqlAppDatabaseName = "appdb"
$sqlMemberDatabaseName = "memberdb"
$sqlDatabaseServerFirewallRuleName = $Name + "rule"
```

```
# Create a SQL Azure database server, app and member databases
$sql = $scriptPath\New-AzureSql.ps1 `
    -AppDatabaseName $sqlAppDatabaseName `
    -MemberDatabaseName $sqlMemberDatabaseName `
    -UserName $SqlDatabaseUserName `
    -Password $SqlDatabasePassword `
    -FirewallRuleName $sqlDatabaseServerFirewallRuleName `
    -StartIPAddress $StartIPAddress `
    -EndIPAddress $EndIPAddress `
    -Location $Location
```

The database creation script retrieves the dev machine's IP address and sets a firewall rule so the dev machine can connect to and manage the server. The database creation script then goes through several steps to set up the databases:

- Creates the server by using the `New-AzureSqlDatabaseServer` cmdlet.

  ```
  $databaseServer = New-AzureSqlDatabaseServer -AdministratorLogin $UserName -
  AdministratorLoginPassword $Password -Location $Location
  ```

- Creates firewall rules to enable the dev machine to manage the server and to enable the web app to connect to it.

  ```
  # Create a SQL Azure database server firewall rule for the IP address of the machine in which this
  script will run
  # This will also whitelist all the Azure IP so that the website can access the database server
  New-AzureSqlDatabaseServerFirewallRule -ServerName $databaseServerName -RuleName $FirewallRuleName -
  StartIpAddress $StartIPAddress
  -EndIpAddress $EndIPAddress -Verbose
  New-AzureSqlDatabaseServerFirewallRule -ServerName $databaseServer.ServerName -AllowAllAzureServices
  -RuleName "AllowAllAzureIP" -Verbose
  ```

- Creates a database context that includes the server name and credentials, by using the `New-AzureSqlDatabaseServerContext` cmdlet.

  ```
  # Create a database context which includes the server name and credential
  # These are all local operations. No API call to Azure
  $credential = New-PSCredentialFromPlainText -UserName $UserName -Password $Password
  $context = New-AzureSqlDatabaseServerContext -ServerName $databaseServer.ServerName -Credential
  $credential
  ```

  `New-PSCredentialFromPlainText` is a function in the script that calls the `ConvertTo-SecureString` cmdlet to encrypt the password and returns a `PSCredential` object, the same type that the `Get-Credential` cmdlet returns.

- Creates the application database and the membership database by using the `New-AzureSqlDatabase` cmdlet.

  ```
  # Use the database context to create app database
  New-AzureSqlDatabase -DatabaseName $AppDatabaseName -Context $context -Verbose

  # Use the database context to create member database
  New-AzureSqlDatabase -DatabaseName $MemberDatabaseName -Context $context -Verbose
  ```

- Calls a locally defined function to create a connection string for each database. The application will use these connection strings to access the databases.

  ```
  $appDatabaseConnectionString = Get-SQLAzureDatabaseConnectionString -DatabaseServerName
  $databaseServerName -DatabaseName $AppDatabaseName -UserName $UserName -Password $Password
  $memberDatabaseConnectionString = Get-SQLAzureDatabaseConnectionString -DatabaseServerName
  $databaseServerName -DatabaseName $MemberDatabaseName -UserName $UserName -Password $Password
  ```

  Get-SQLAzureDatabaseConnectionString is a function defined in the script that creates the connection string from the parameter values supplied to it.

  ```
  Function Get-SQLAzureDatabaseConnectionString
  {
      Param(
          [String]$DatabaseServerName,
          [String]$DatabaseName,
          [String]$UserName,
          [String]$Password
      )

      Return "Server=tcp:$DatabaseServerName.database.windows.net,1433;Database=$DatabaseName;User
  ID=$UserName@$DatabaseServerName;Password=$Password;Trusted_Connection=False;Encrypt=True;Connection
  Timeout=30;"
  }
  ```

- Returns a hash table with the database server name and the connection strings.

  ```
  Return @{ `
      Server = $databaseServer.ServerName; UserName = $UserName; Password = $Password; `
      AppDatabase = @{Name = $AppDatabaseName; ConnectionString = $appDatabaseConnectionString}; `
      MemberDatabase = @{Name = $MemberDatabaseName; ConnectionString =
  $memberDatabaseConnectionString} `
  }
  ```

The Fix It app uses separate membership and application databases. It's also possible to put both membership

and application data in a single database.

Store app settings and connection strings

Azure has a feature that enables you to store settings and connection strings that automatically override what is returned to the application when it tries to read the `appSettings` or `connectionStrings` collections in the Web.config file. This is an alternative to applying Web.config transformations when you deploy. For more information, see Store sensitive data in Azure later in this e-book.

The environment creation script stores in Azure all of the `appSettings` and `connectionStrings` values that the application needs to access the storage account and databases when it runs in Azure.

```
# Configure app settings for storage account and New Relic
$appSettings = @{ `
    "StorageAccountName" = $storageAccountName; `
    "StorageAccountAccessKey" = $storage.AccessKey; `
    "COR_ENABLE_PROFILING" = "1"; `
    "COR_PROFILER" = "{71DA0A04-7777-4EC6-9643-7D28B46A8A41}"; `
    "COR_PROFILER_PATH" = "C:\Home\site\wwwroot\newrelic\NewRelic.Profiler.dll"; `
    "NEWRELIC_HOME" = "C:\Home\site\wwwroot\newrelic" `
}
```

```
# Configure connection strings for appdb and ASP.NET member db
$connectionStrings = ( `
    @{Name = $sqlAppDatabaseName; Type = "SQLAzure"; ConnectionString = $sql.AppDatabase.ConnectionString},
`
    @{Name = "DefaultConnection"; Type = "SQLAzure"; ConnectionString =
$sql.MemberDatabase.ConnectionString}
)
```

```
# Add the connection string and storage account name/key to the website
Set-AzureWebsite -Name $Name -AppSettings $appSettings -ConnectionStrings $connectionStrings
```

New Relic is a telemetry framework that we demonstrate in the Monitoring and Telemetry chapter. The environment creation script also restarts the web app to make sure that it picks up the New Relic settings.

```
# Restart the website to let New Relic hook kick in
Restart-AzureWebsite -Name $websiteName
```

Preparing for deployment

At the end of the process, the environment creation script calls two functions to create files that will be used by the deployment script.

One of these functions creates a publish profile *(<websitename>.pubxml* file). The code calls the Azure REST API to get the publish settings, and it saves the information in a *.publishsettings* file. Then it uses the information from that file along with a template file (*pubxml.template*) to create the *.pubxml* file that contains the publish profile. This two-step process simulates what you do in Visual Studio: download a *.publishsettings* file and import that to create a publish profile.

The other function uses another template file (website-environment.template) to create a *website-environment.xml* file that contains settings the deployment script will use along with the *.pubxml* file.

Troubleshooting and error handling

Scripts are like programs: they can fail, and when they do you want to know as much as you can about the failure and what caused it. For this reason, the environment creation script changes the value of the `VerbosePreference` variable from `SilentlyContinue` to `Continue` so that all verbose messages are displayed. It also changes the

value of the `ErrorActionPreference` variable from `Continue` to `Stop`, so that the script stops even when it encounters non-terminating errors:

```
# Set the output level to verbose and make the script stop on error
$VerbosePreference = "Continue"
$ErrorActionPreference = "Stop"
```

Before it does any work, the script stores the start time so that it can calculate the elapsed time when it's done:

```
# Mark the start time of the script execution
$startTime = Get-Date
```

After it completes its work, the script displays the elapsed time:

```
# Mark the finish time of the script execution
$finishTime = Get-Date
# Output the time consumed in seconds
Write-Output ("Total time used (seconds): {0}" -f ($finishTime - $startTime).TotalSeconds)
```

And for every key operation the script writes verbose messages, for example:

```
Write-Verbose "[Start] creating $websiteName website in $Location location"
$website = New-AzureWebsite -Name $websiteName -Location $Location -Verbose
Write-Verbose "[Finish] creating $websiteName website in $Location location"
```

Deployment script

What the *New-AzureWebsiteEnv.ps1* script does for environment creation, the *Publish-AzureWebsite.ps1* script does for application deployment.

The deployment script gets the name of the web app from the *website-environment.xml* file created by the environment creation script.

```
[Xml]$envXml = Get-Content "$scriptPath\website-environment.xml"
$websiteName = $envXml.environment.name
```

It gets the deployment user password from the *.publishsettings* file:

```
[Xml]$xml = Get-Content $scriptPath\$websiteName.publishsettings
$password = $xml.publishData.publishProfile.userPWD[0]
$publishXmlFile = Join-Path $scriptPath -ChildPath ($websiteName + ".pubxml")
```

It executes the MSBuild command that builds and deploys the project:

```
& "$env:windir\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe" $ProjectFile `
    /p:VisualStudioVersion=12.0 `
    /p:DeployOnBuild=true `
    /p:PublishProfile=$publishXmlFile `
    /p:Password=$password
```

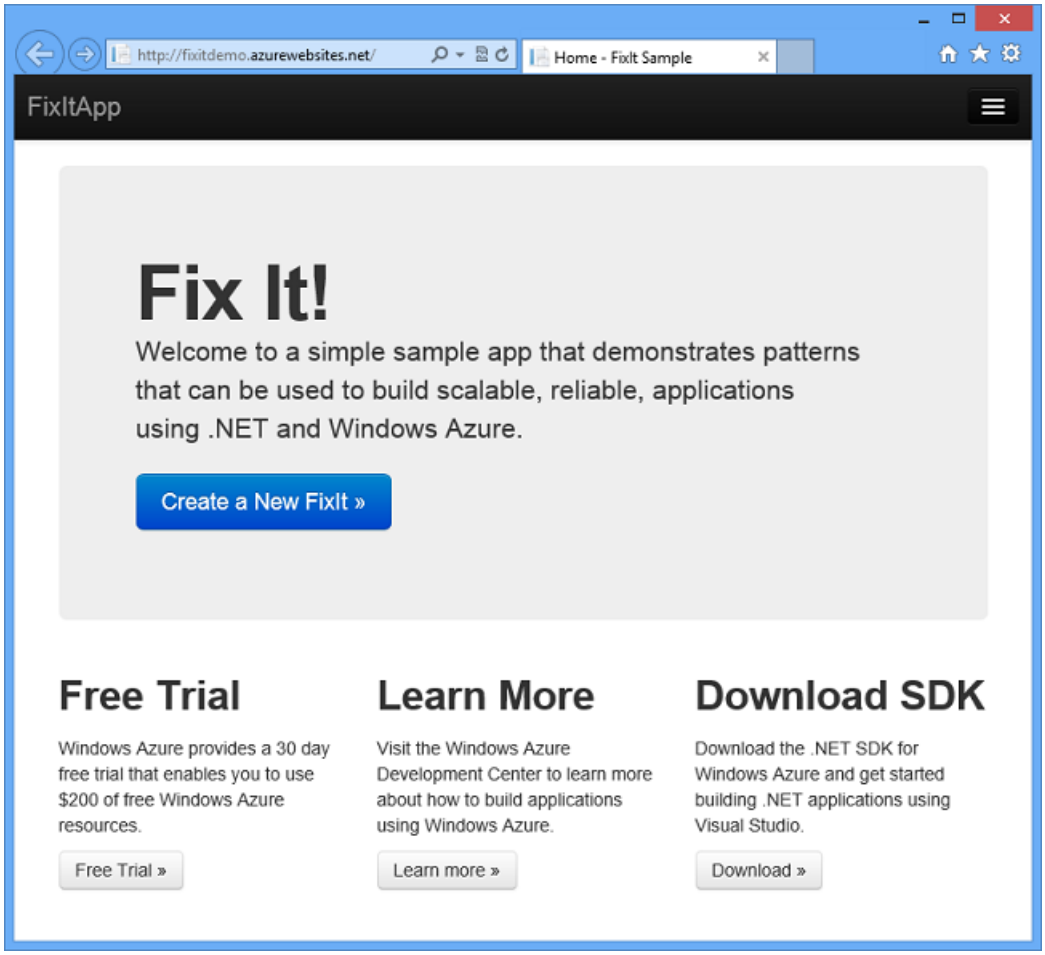
And if you've specified the `Launch` parameter on the command line, it calls the `Show-AzureWebsite` cmdlet to open your default browser to the website URL.

```
If ($Launch)
{
    Show-AzureWebsite -Name $websiteName
}
```

You can run the deployment script with a command like this one:

```
.\Publish-AzureWebsite.ps1 ..\MyFixIt\MyFixIt.csproj -Launch
```

And when it's done, the browser opens with the site running in the cloud at the `<websitename>.azurewebsites.net` URL.



Summary

With these scripts you can be confident that the same steps will always be executed in the same order using the same options. This helps ensure that each developer on the team doesn't miss something or mess something up or deploy something custom on his own machine that won't actually work the same way in another team member's environment or in production.

In a similar way, you can automate most Azure management functions that you can do in the management portal, by using the REST API, Windows PowerShell scripts, a .NET language API, or a Bash utility that you can run on Linux or Mac.

In the next chapter we'll look at source code and explain why it's important to include your scripts in your source code repository.

Resources

- Install and Configure Windows PowerShell for Azure. Explains how to install the Azure PowerShell cmdlets

and how to install the certificate that you need on your computer in order to manage your Azure account. This is a great place to get started because it also has links to resources for learning PowerShell itself.

- Azure Script Center. WindowsAzure.com portal to resources for developing scripts that manage Azure services, with links to getting started tutorials, cmdlet reference documentation and source code, and sample scripts
- Weekend Scripter: Getting Started with Azure and PowerShell. In a blog dedicated to Windows PowerShell, this post provides a great introduction to using PowerShell for Azure management functions.
- Install and Configure the Azure Cross-Platform Command-Line Interface. Getting-started tutorial for an Azure scripting framework that works on Mac and Linux as well as Windows systems.
- Command-line tools section of the Download Azure SDKs and Tools topic. Portal page for documentation and downloads related to command-line tools for Azure.
- Automating everything with the Azure Management Libraries and .NET. Scott Hanselman introduces the .NET management API for Azure.
- Using Windows PowerShell Scripts to Publish to Dev and Test Environments. MSDN documentation that explains how to use publish scripts that Visual Studio automatically generates for web projects.
- PowerShell Tools for Visual Studio 2013. Visual Studio extension that adds language support for Windows PowerShell in Visual Studio.

PREVIOUS NEXT

Source Control (Building Real-World Cloud Apps with Azure)

2/20/2020 • 13 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the e-book, see the first chapter.

Source control is essential for all cloud development projects, not just team environments. You wouldn't think of editing source code or even a Word document without an undo function and automatic backups, and source control gives you those functions at a project level where they can save even more time when something goes wrong. With cloud source control services, you no longer have to worry about complicated set-up, and you can use Azure Repos source control free for up to 5 users.

The first part of this chapter explains three key best practices to keep in mind:

- Treat automation scripts as source code and version them together with your application code.
- Never check in secrets (sensitive data such as credentials) into a source code repository.
- Set up source branches to enable the DevOps workflow.

The remainder of the chapter gives some sample implementations of these patterns in Visual Studio, Azure, and Azure Repos:

- Add scripts to source control in Visual Studio
- Store sensitive data in Azure
- Use Git in Visual Studio and Azure Repos

Treat automation scripts as source code

When you're working on a cloud project you're changing things frequently and you want to be able to react quickly to issues reported by your customers. Responding quickly involves using automation scripts, as explained in the Automate Everything chapter. All of the scripts that you use to create your environment, deploy to it, scale it, etc., need to be in sync with your application source code.

To keep scripts in sync with code, store them in your source control system. Then if you ever need to roll back changes or make a quick fix to production code which is different from development code, you don't have to waste time trying to track down which settings have changed or which team members have copies of the version you need. You're assured that the scripts you need are in sync with the code base that you need them for, and you're assured that all team members are working with the same scripts. Then whether you need to automate testing and deployment of a hot fix to production or new feature development, you'll have the right script for the code that needs to be updated.

Don't check in secrets

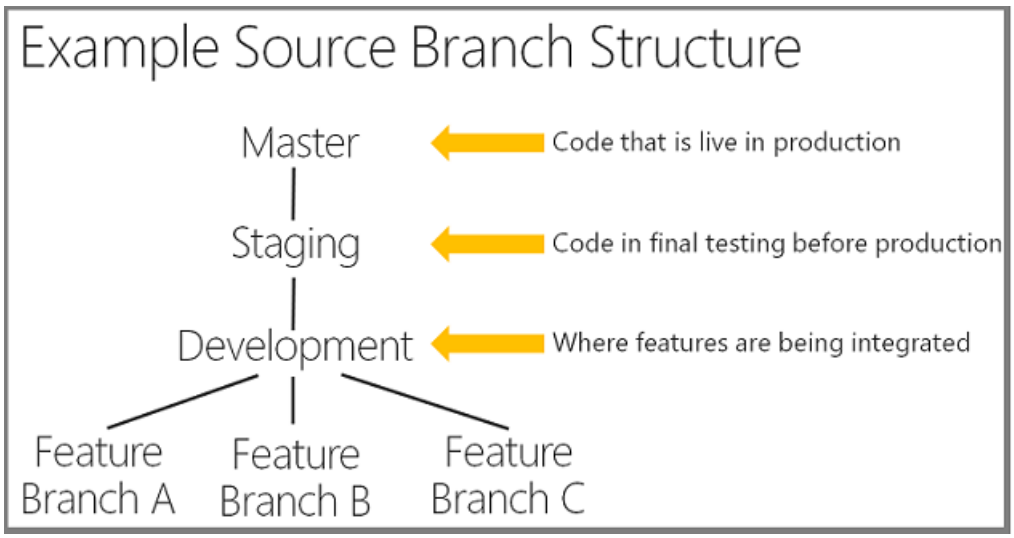
A source code repository is typically accessible to too many people for it to be an appropriately secure place for sensitive data such as passwords. If scripts rely on secrets such as passwords, parameterize those settings so that

they don't get saved in source code, and store your secrets somewhere else.

For example, Azure lets you download files that contain publish settings in order to automate the creation of publish profiles. These files include user names and passwords that are authorized to manage your Azure services. If you use this method to create publish profiles, and if you check in these files to source control, anyone with access to your repository can see those user names and passwords. You can safely store the password in the publish profile itself because it's encrypted and it's in a *.pubxml.user* file that by default is not included in source control.

Structure source branches to facilitate DevOps workflow

How you implement branches in your repository affects your ability to both develop new features and fix issues in production. Here is a pattern that a lot of medium sized teams use:



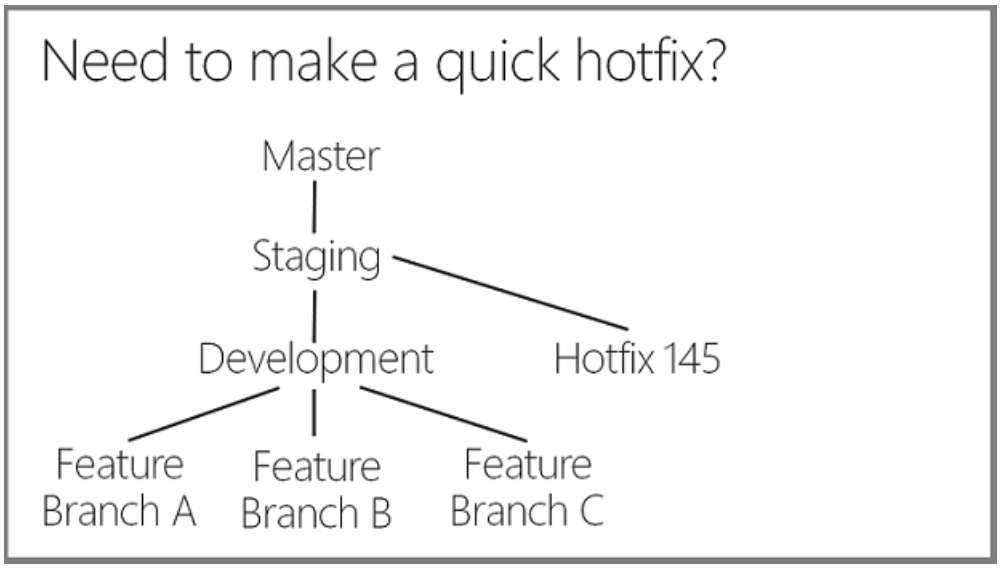
The master branch always matches code that is in production. Branches underneath master correspond to different stages in the development life cycle. The development branch is where you implement new features. For a small team you might just have master and development, but we often recommend that people have a staging branch between development and master. You can use staging for final integration testing before an update is moved to production.

For big teams there may be separate branches for each new feature; for a smaller team you might have everyone checking in to the development branch.

If you have a branch for each feature, when Feature A is ready you merge its source code changes up into the development branch and down into the other feature branches. This source code merging process can be time-consuming, and to avoid that work while still keeping features separate, some teams implement an alternative called *feature toggles* (also known as *feature flags*). This means all of the code for all of the features is in the same branch, but you enable or disable each feature by using switches in the code. For example, suppose Feature A is a new field for Fix It app tasks, and Feature B adds caching functionality. The code for both features can be in the development branch, but the app will only display the new field when a variable is set to true, and it will only use caching when a different variable is set to true. If Feature A isn't ready to be promoted but the Feature B is ready, you can promote all of the code to Production with the Feature A switch off and the Feature B switch on. You can then finish Feature A and promote it later, all with no source code merging.

Whether or not you use branches or toggles for features, a branching structure like this enables you to flow your code from development into production in an agile and repeatable way.

This structure also enables you to react quickly to customer feedback. If you need to make a quick fix to production, you can also do that efficiently in an agile way. You can create a branch off of master or staging, and when it's ready merge it up into master and down into development and feature branches.



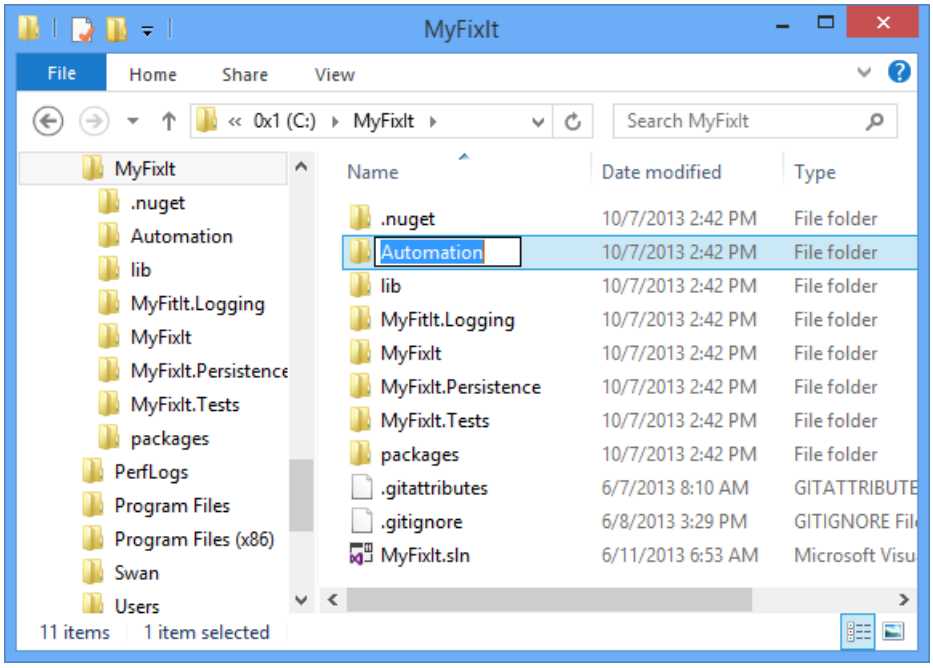
Without a branching structure like this with its separation of production and development branches, a production problem could put you in the position of having to promote new feature code along with your production fix. The new feature code might not be fully tested and ready for production and you might have to do a lot of work backing out changes that aren't ready. Or you might have to delay your fix in order to test changes and get them ready to deploy.

Next you'll see examples of how to implement these three patterns in Visual Studio, Azure, and Azure Repos. These are examples rather than detailed step-by-step how-to-do-it instructions; for detailed instructions that provide all of the context necessary, see the Resources section at the end of the chapter.

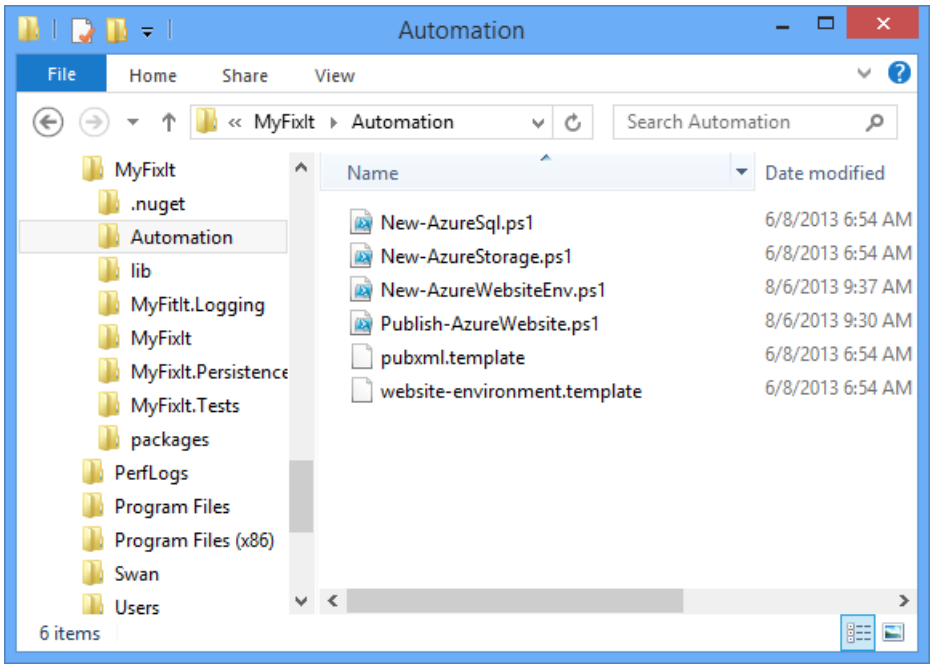
Add scripts to source control in Visual Studio

You can add scripts to source control in Visual Studio by including them in a Visual Studio solution folder (assuming your project is in source control). Here's one way to do it.

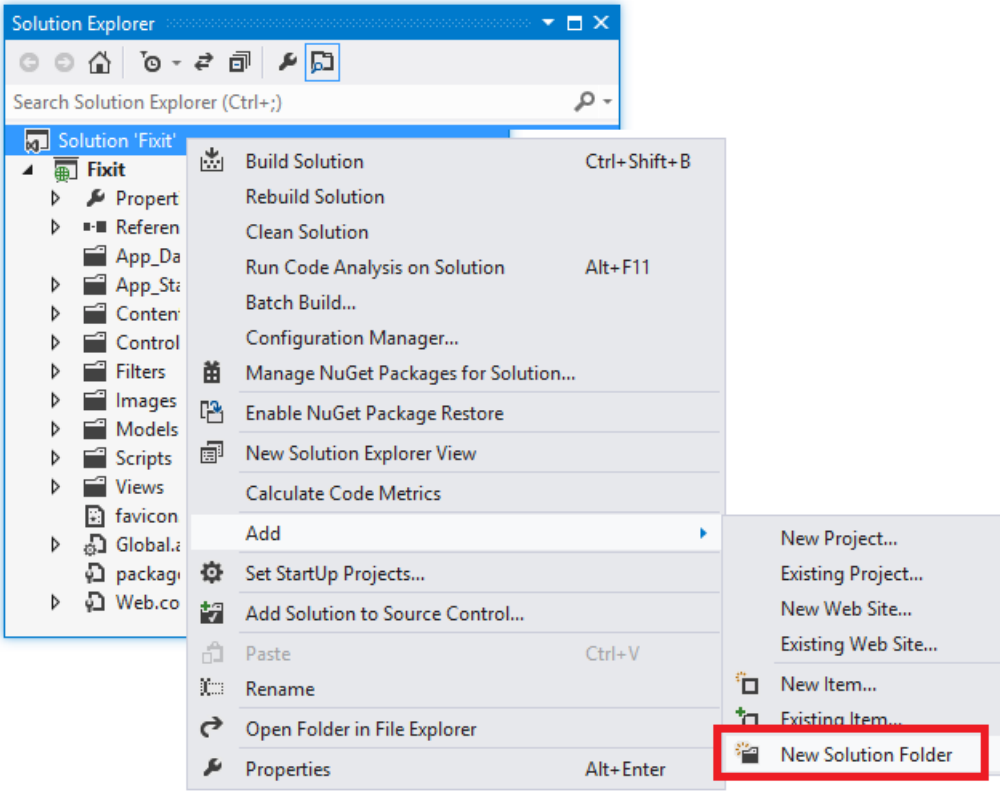
Create a folder for the scripts in your solution folder (the same folder that has your *.sln* file).



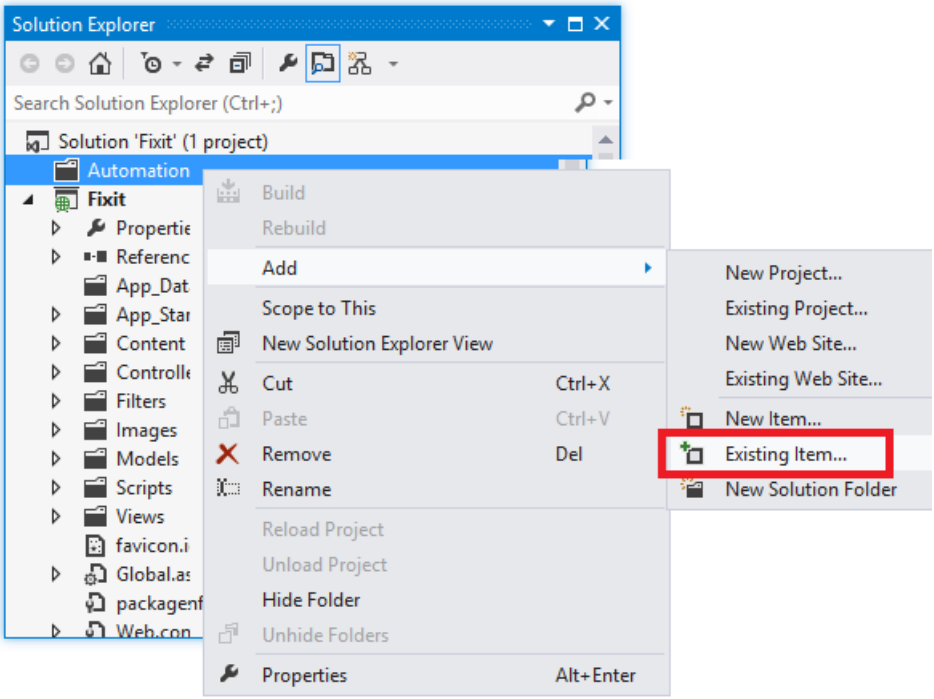
Copy the script files into the folder.

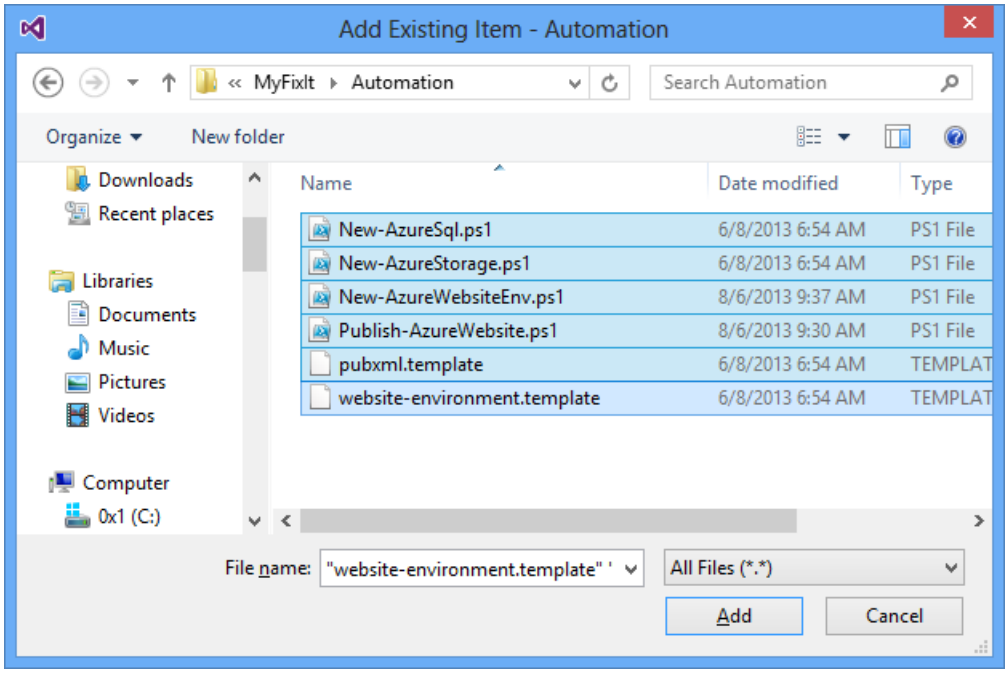


In Visual Studio, add a solution folder to the project.



And add the script files to the solution folder.





The script files are now included in your project and source control is tracking their version changes along with corresponding source code changes.

Store sensitive data in Azure

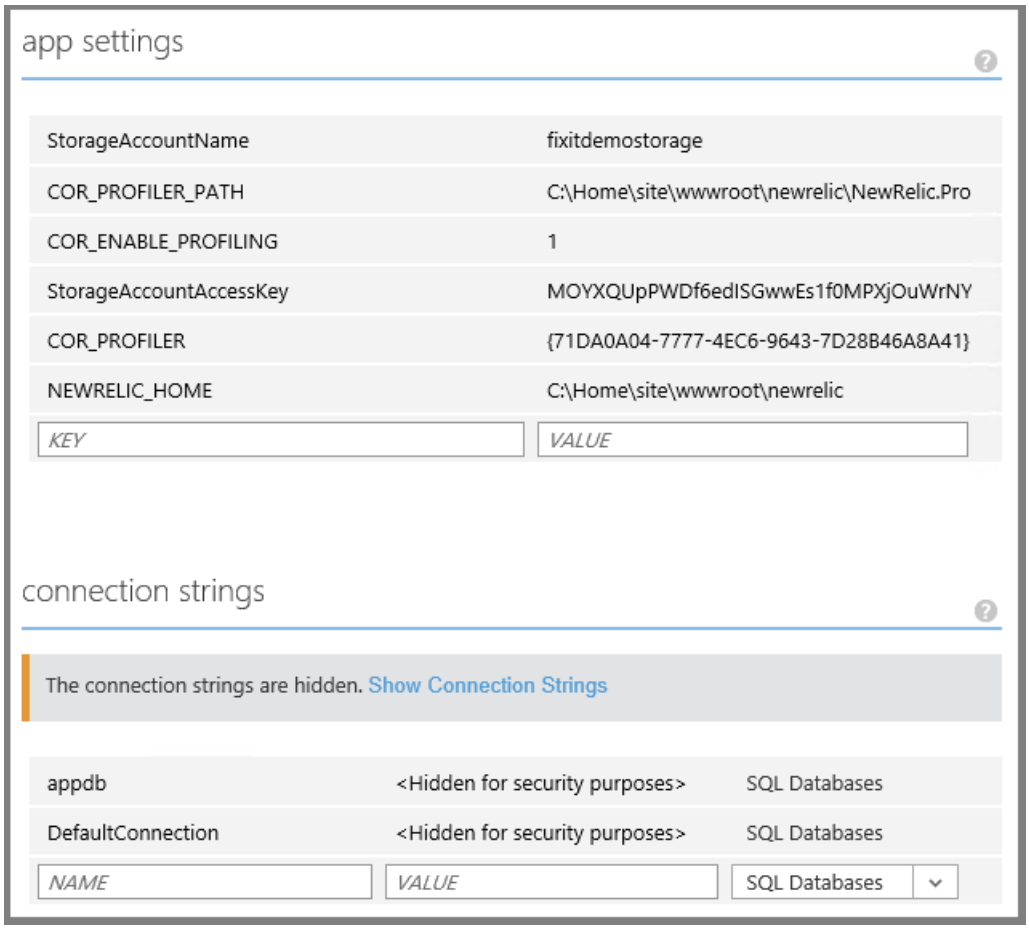
If you run your application in an Azure Web Site, one way to avoid storing credentials in source control is to store them in Azure instead.

For example, the Fix It application stores in its Web.config file two connection strings that will have passwords in production and a key that gives access to your Azure storage account.

```
<connectionStrings>
  <add name="DefaultConnection" connectionString="Data Source=
(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\MyFixItMembership.mdf;Initial
Catalog=MyFixItMembership;Integrated Security=True" providerName="System.Data.SqlClient" />
  <add name="appdb" connectionString="Data Source=
(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\MyFixItTasks.mdf;Initial Catalog=aspnet-
MyFixItTasks;Integrated Security=True" providerName="System.Data.SqlClient" />
</connectionStrings>
<appSettings>
  <add key="webpages:Version" value="3.0.0.0" />
  <add key="webpages:Enabled" value="false" />
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  <add key="StorageAccountName" value="fixitdemostorage" />
  <add key="StorageAccountAccessKey" value="[accesskeyvalue]" />
</appSettings>
```

If you put actual production values for these settings in your *Web.config* file, or if you put them in the *Web.Release.config* file to configure a Web.config transform to insert them during deployment, they'll be stored in the source repository. If you enter the database connection strings into the production publish profile, the password will be in your *.pubxml* file. (You could exclude the *.pubxml* file from source control, but then you lose the benefit of sharing all the other deployment settings.)

Azure gives you an alternative for the **appSettings** and connection strings sections of the *Web.config* file. Here is the relevant part of the **Configuration** tab for a web site in the Azure management portal:



When you deploy a project to this web site and the application runs, whatever values you have stored in Azure override whatever values are in the Web.config file.

You can set these values in Azure by using either the management portal or scripts. The environment creation automation script you saw in the Automate Everything chapter creates an Azure SQL Database, gets the storage and SQL Database connection strings, and stores these secrets in the settings for your web site.

```
# Configure app settings for storage account and New Relic
$appSettings = @{ `
    "StorageAccountName" = $storageAccountName; `
    "StorageAccountAccessKey" = $storage.AccessKey; `
    "COR_ENABLE_PROFILING" = "1"; `
    "COR_PROFILER" = "{71DA0A04-7777-4EC6-9643-7D28B46A8A41}"; `
    "COR_PROFILER_PATH" = "C:\Home\site\wwwroot\newrelic\NewRelic.Profiler.dll"; `
    "NEWRELIC_HOME" = "C:\Home\site\wwwroot\newrelic" `
}
```

```
# Configure connection strings for appdb and ASP.NET member db
$connectionStrings = ( `
    @{Name = $sqlAppDatabaseName; Type = "SQLAzure"; ConnectionString = $sql.AppDatabase.ConnectionString},
`
    @{Name = "DefaultConnection"; Type = "SQLAzure"; ConnectionString =
$sql.MemberDatabase.ConnectionString}
)
```

Notice that the scripts are parameterized so that actual values don't get persisted to the source repository.

When you run locally in your development environment, the app reads your local Web.config file and your connection string points to a LocalDB SQL Server database in the *App_Data* folder of your web project. When you run the app in Azure and the app tries to read these values from the Web.config file, what it gets back and uses are the values stored for the Web Site, not what's actually in Web.config file.

Use Git in Visual Studio and Azure DevOps

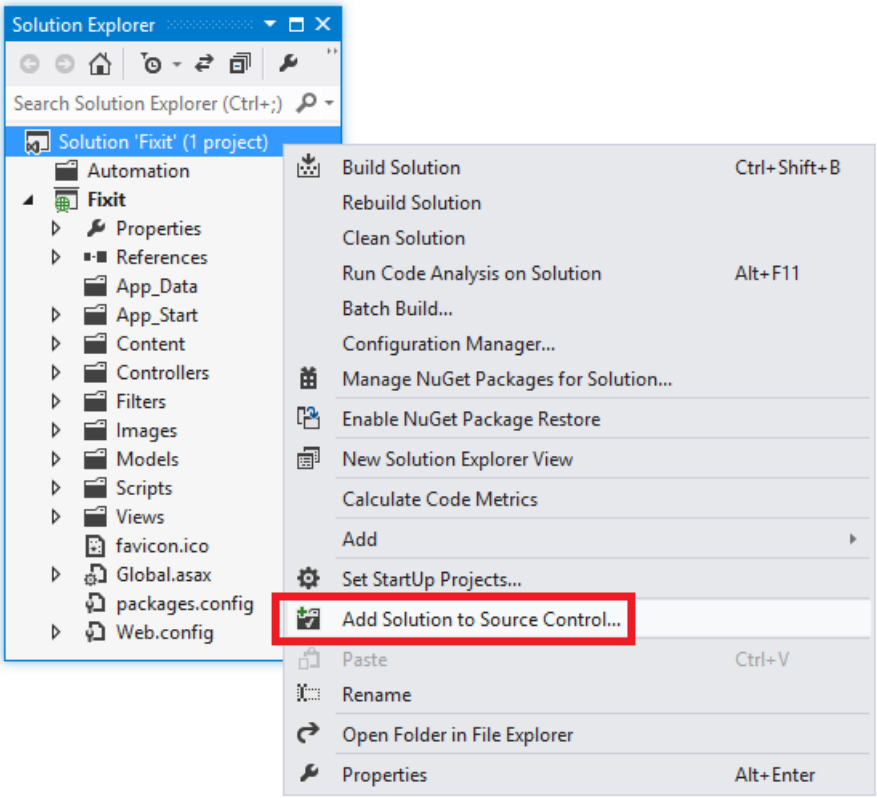
You can use any source control environment to implement the DevOps branching structure presented earlier. For distributed teams a distributed version control system (DVCS) might work best; for other teams a centralized system might work better.

Git is a popular distributed version control system. When you use Git for source control, you have a complete copy of the repository with all of its history on your local computer. Many people prefer that because it's easier to continue working when you're not connected to the network -- you can continue to do commits and rollbacks, create and switch branches, and so forth. Even when you're connected to the network, it's easier and quicker to create branches and switch branches when everything is local. You can also do local commits and rollbacks without having an impact on other developers. And you can batch commits before sending them to the server.

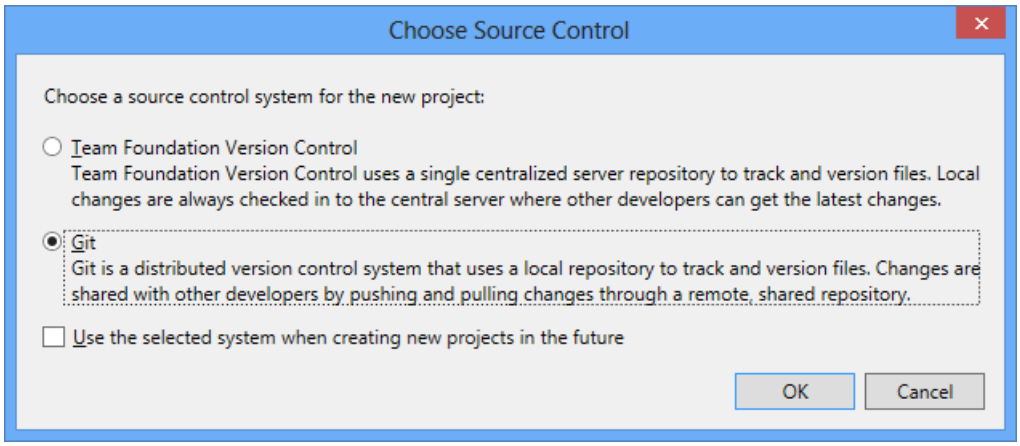
Azure Repos offers both Git and Team Foundation Version Control (TFVC; centralized source control). Get started with Azure DevOps here.

Visual Studio 2017 includes built-in, first-class Git support. Here's a quick demo of how that works.

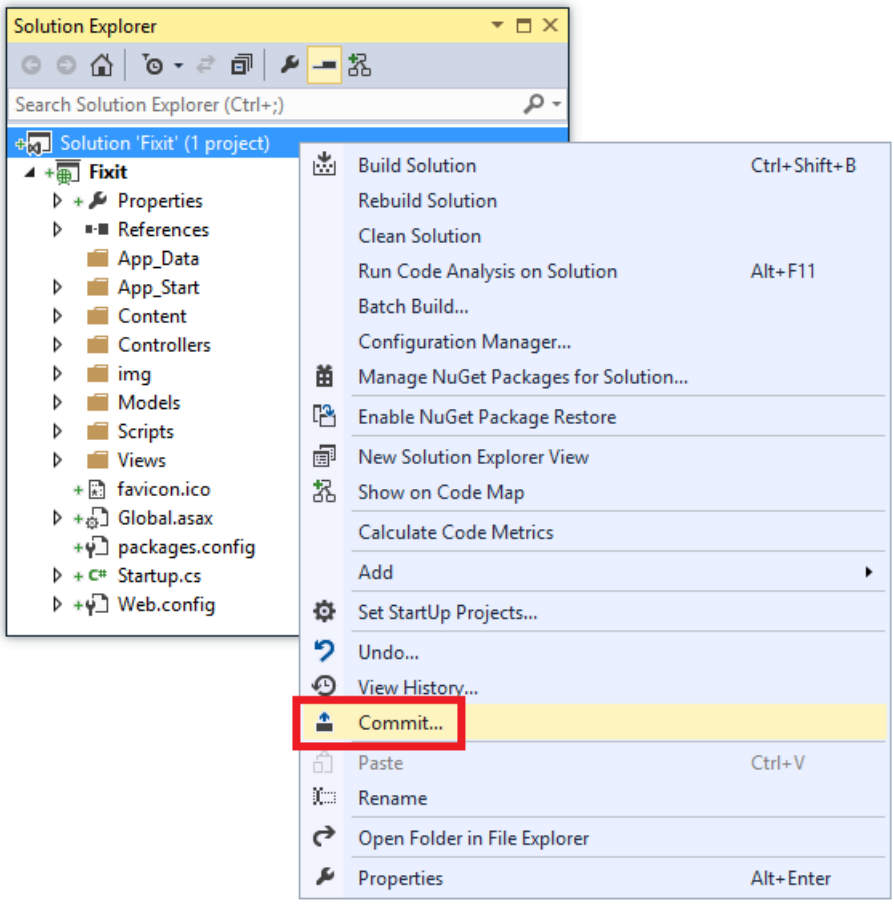
With a project open in Visual Studio, right-click the solution in **Solution Explorer**, and then choose **Add Solution to Source Control**.



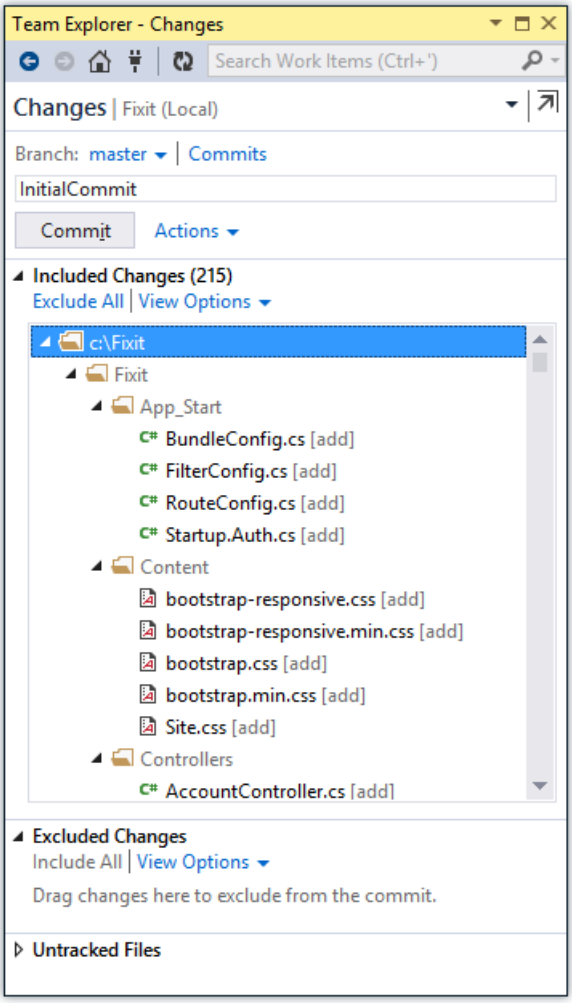
Visual Studio asks if you want to use TFVC (centralized version control) or Git.



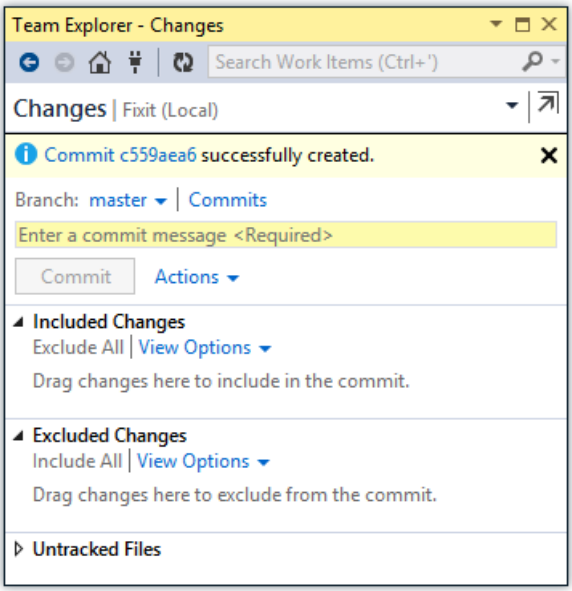
When you select Git and click **OK**, Visual Studio creates a new local Git repository in your solution folder. The new repository has no files yet; you have to add them to the repository by doing a Git commit. Right-click the solution in **Solution Explorer**, and then click **Commit**.



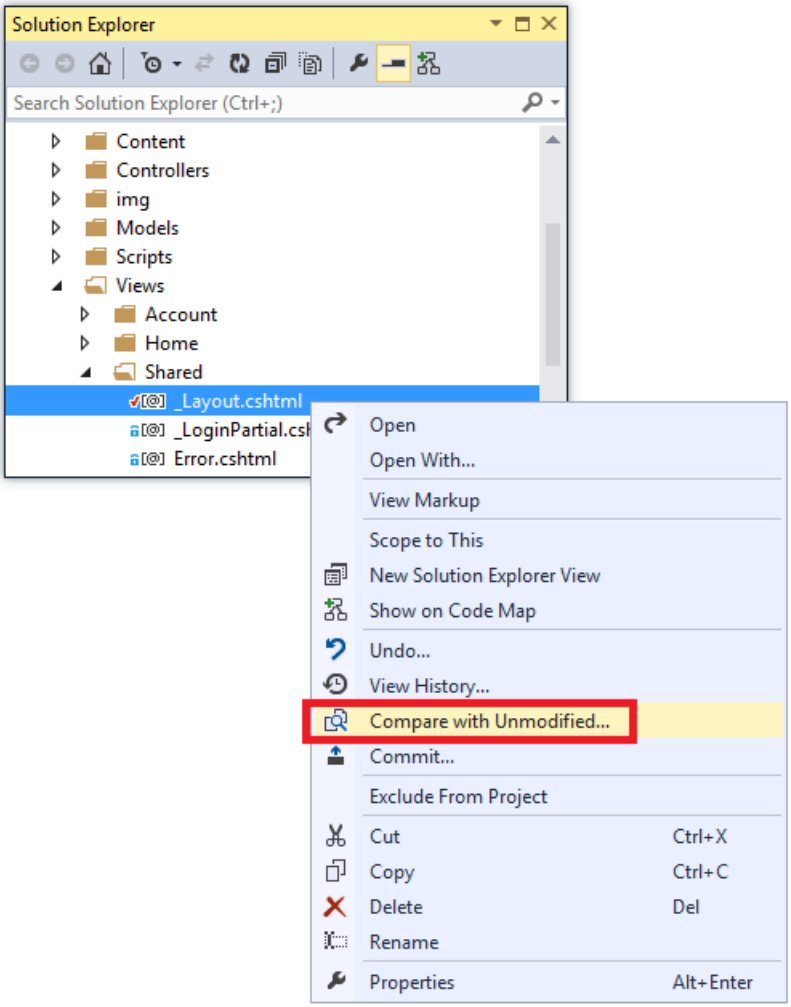
Visual Studio automatically stages all of the project files for the commit and lists them in **Team Explorer** in the **Included Changes** pane. (If there were some you didn't want to include in the commit, you could select them, right-click, and click **Exclude**.)

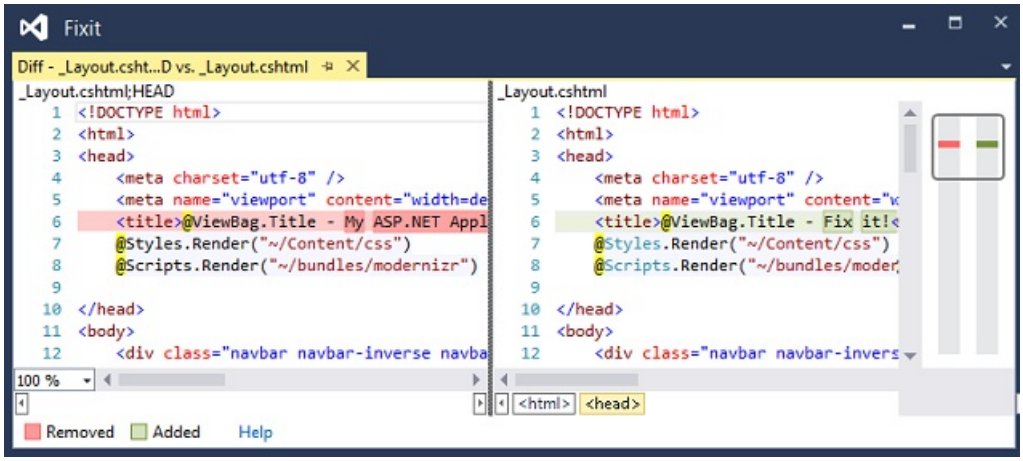


Enter a commit comment and click **Commit**, and Visual Studio executes the commit and displays the commit ID.



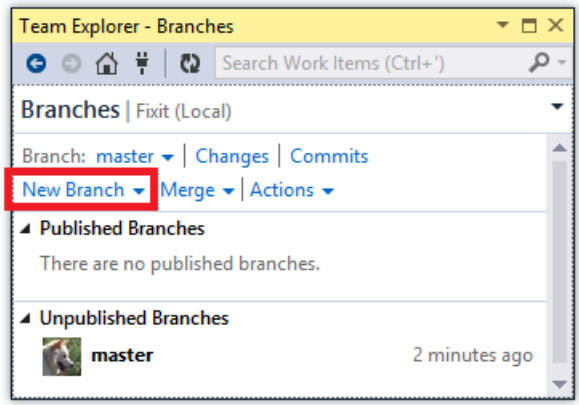
Now if you change some code so that it's different from what's in the repository, you can easily view the differences. Right-click a file that you've changed, select **Compare with Unmodified**, and you get a comparison display that shows your uncommitted change.



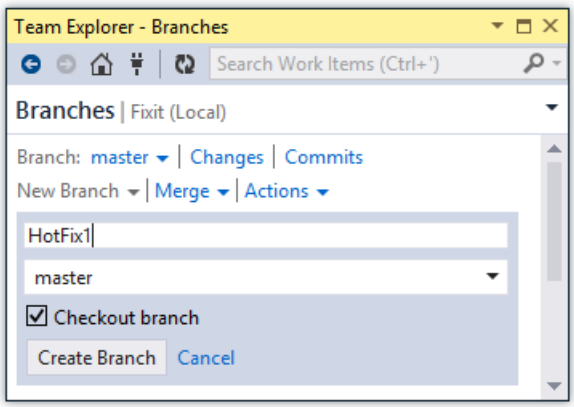


You can easily see what changes you're making and check them in.

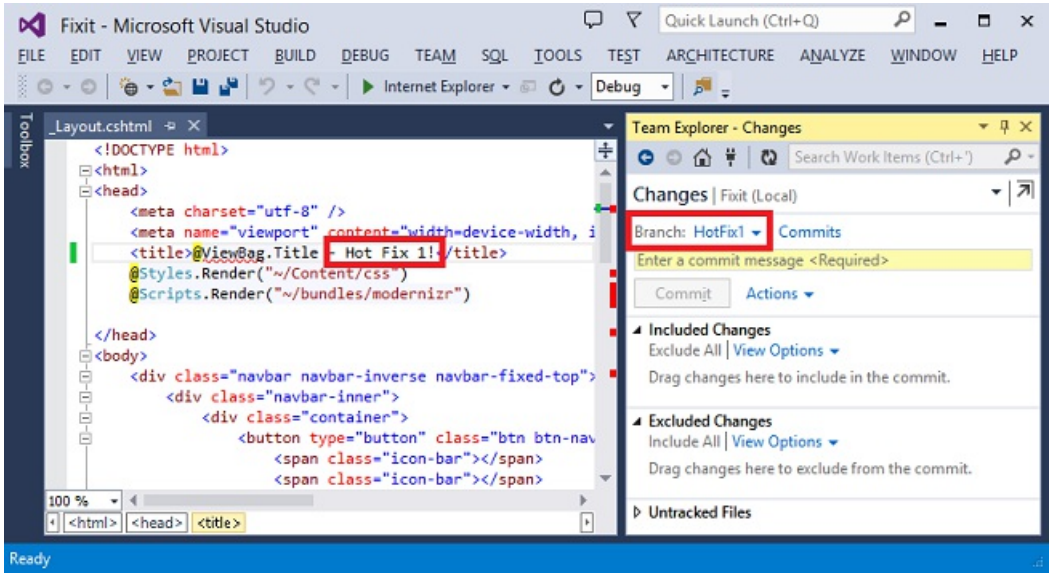
Suppose you need to make a branch – you can do that in Visual Studio too. In **Team Explorer**, click **New Branch**.



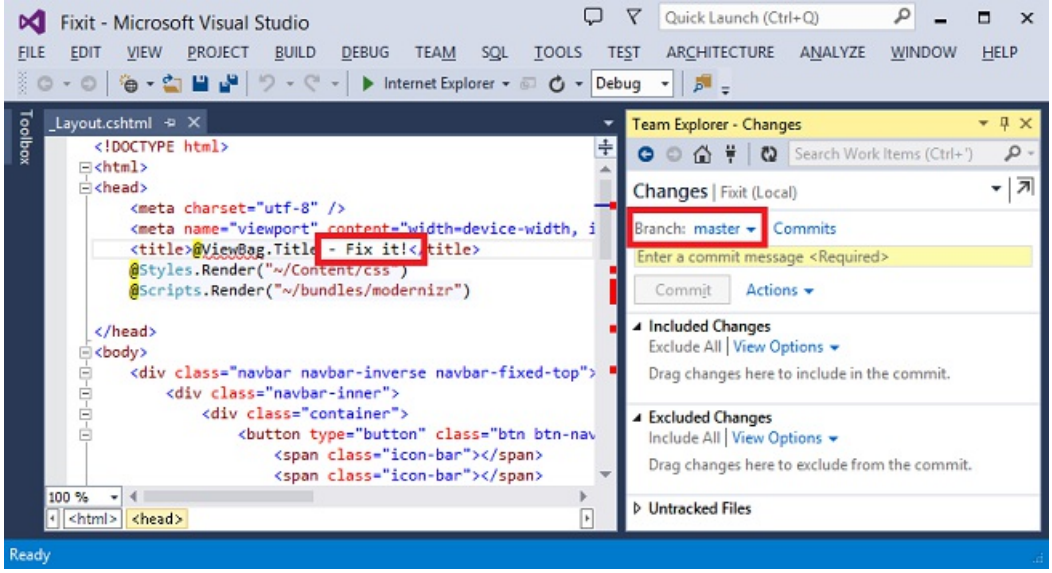
Enter a branch name, click **Create Branch**, and if you selected **Checkout branch**, Visual Studio automatically checks out the new branch.



You can now make changes to files and check them in to this branch. And you can easily switch between branches and Visual Studio automatically syncs the files to whichever branch you have checked out. In this example the web page title in *_Layout.cshtml* has been changed to "Hot Fix 1" in HotFix1 branch.



If you switch back to the master branch, the contents of the *_Layout.cshtml* file automatically revert to what they are in the master branch.



This a simple example of how you can quickly create a branch and flip back and forth between branches. This feature enables a highly agile workflow using the branch structure and automation scripts presented in the Automate Everything chapter. For example, you can be working in the Development branch, create a hot fix branch off of master, switch to the new branch, make your changes there and commit them, and then switch back to the Development branch and continue what you were doing.

What you've seen here is how you work with a local Git repository in Visual Studio. In a team environment you typically also push changes up to a common repository. The Visual Studio tools also enable you to point to a remote Git repository. You can use GitHub.com for that purpose, or you can use Git and Azure Repos integrated with all the other Azure DevOps capabilities such as work item and bug tracking.

This isn't the only way you can implement an agile branching strategy, of course. You can enable the same agile workflow using a centralized source control repository.

Summary

Measure the success of your source control system based on how quickly you can make a change and get it live in a safe and predictable way. If you find yourself scared to make a change because you have to do a day or two of manual testing on it, you might ask yourself what you have to do process-wise or test-wise so that you can make that change in minutes or at worst no longer than an hour. One strategy for doing that is to implement continuous integration and continuous delivery, which we'll cover in the next chapter.

Resources

For more information about branching strategies, see the following resources:

- Building a Release Pipeline with Team Foundation Server 2012. Microsoft Patterns and Practices documentation. See chapter 6 for a discussion of branching strategies. Advocates feature toggles over feature branches, and if branches for features are used, advocates keeping them short-lived (hours or days at most).
- Version Control Guide. Guide to branching strategies by the ALM Rangers. See Branching Strategies.pdf on the Downloads tab.
- Software Development with Feature Toggles. MSDN Magazine article.
- Feature Toggle. Introduction to feature toggles / feature flags on Martin Fowler's blog.
- Feature Toggles vs Feature Branches. Another blog post about feature toggles, by Dylan Smith.

For more information about how to handle sensitive information that should not be kept in source control repositories, see the following resources:

- Best practices for deploying passwords and other sensitive data to ASP.NET and Azure App Service.

- Azure Web Sites: How Application Strings and Connection Strings Work. Explains the Azure feature that overrides `appSettings` and `connectionStrings` data in the *Web.config* file.
- Custom configuration and application settings in Azure Web Sites - with Stefan Schackow.

For information about other methods for keeping sensitive information out of source control, see ASP.NET MVC: Keep Private Settings Out of Source Control.

PREVIOUS NEXT

Continuous Integration and Continuous Delivery (Building Real-World Cloud Apps with Azure)

2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the e-book, see the first chapter.

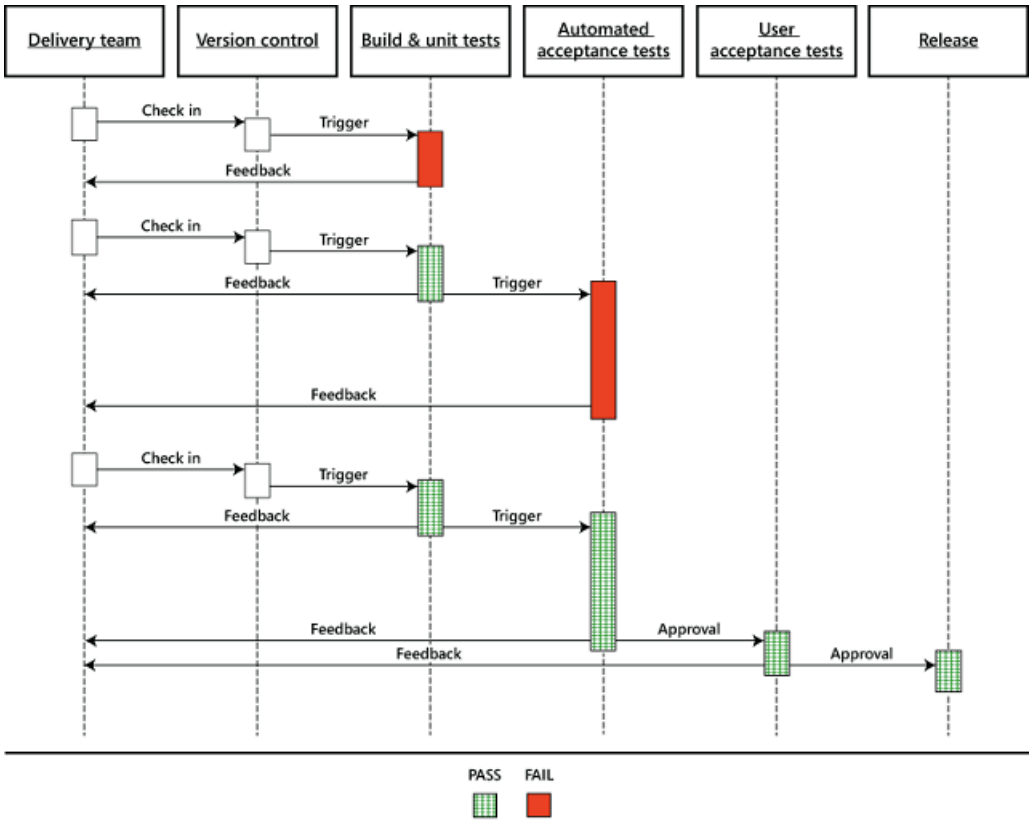
The first two recommended development process patterns were Automate Everything and Source Control, and the third process pattern combines them. Continuous integration (CI) means that whenever a developer checks in code to the source repository, a build is automatically triggered. Continuous delivery (CD) takes this one step further: after a build and automated unit tests are successful, you automatically deploy the application to an environment where you can do more in-depth testing.

The cloud enables you to minimize the cost of maintaining a test environment because you only pay for the environment resources as long as you're using them. Your CD process can set up the test environment when you need it, and you can take down the environment when you're done testing.

Continuous Integration and Continuous Delivery workflow

Generally we recommend that you do continuous delivery to your development and staging environments. Most teams, even at Microsoft, require a manual review and approval process for production deployment. For a production deployment you might want to make sure it happens when key people on the development team are available for support, or during low-traffic periods. But there's nothing to prevent you from completely automating your development and test environments so that all a developer has to do is check in a change and an environment is set up for acceptance testing.

The following diagram from a Microsoft Patterns and Practices e-book about continuous delivery illustrates a typical workflow. Click the image to see it full size in its original context.



How the cloud enables cost-effective CI and CD

Automating these processes in Azure is easy. Because you're running everything in the cloud, you don't have to buy or manage servers for your builds or your test environments. And you don't have to wait for a server to be available to do your testing on. With every build that you do, you could spin up a test environment in Azure using your automation script, run acceptance tests or more in-depth tests against it, and then when you're done just tear it down. And if you only run that server for 2 hours or 8 hours or a day, the amount of money that you have to pay for it is minimal, because you're only paying for the time that a machine is actually running. For example, the environment required for the Fix it application basically costs about 1 cent per hour if you go one tier up from the free level. Over the course of a month, if you only ran the environment an hour at a time, your testing environment would probably cost less than a latte that you buy at Starbucks.

Azure DevOps Services

Azure DevOps Services provides a number of features to assist you with application development from planning to deployment.

- It supports both Git (distributed) and TFVC (centralized) source control.
- It offers an elastic build service, which means it dynamically creates build servers when they're needed and takes them down when they're done. You can automatically kick off a build when someone checks in source code changes, and you don't have to have allocate and pay for your own build servers that lie idle most of the time. The build service is free as long as you don't exceed a certain number of builds. If you expect to do a high volume of builds, you can pay a little extra for reserved build servers.
- It supports continuous delivery to Azure.
- It supports automated load testing. Load testing is critical to a cloud app but is often neglected until it's too late. Load testing simulates heavy use of an app by thousands of users, enabling you to find bottlenecks and improve throughput —before you release the app to production.
- It supports team room collaboration, which facilitates real-time communication and collaboration for small agile teams.
- It supports agile project management.

For more information on the continuous integration and delivery features of Azure DevOps Services, see the Azure DevOps documentation.

If you're looking for a turn-key project management, team collaboration, and source control solution, check out Azure DevOps Services. Sign up at Azure DevOps Services.

Summary

The first three cloud development patterns have been about how to implement a repeatable, reliable, predictable development process with low cycle time. In the next chapter we start to look at architectural and coding patterns.

Resources

For more information, see Deploy a web app in Azure App Service.

See also the following resources:

- Building a Release Pipeline with Team Foundation Server 2012. E-book, hands-on labs, and sample code by Microsoft Patterns and Practices, provides an in-depth introduction to continuous delivery. Covers use of Visual Studio Lab Management and Visual Studio Release Management.
- ALM Rangers DevOps Tooling and Guidance. The ALM Rangers introduced the DevOps Workbench sample companion solution and practical guidance in collaboration with the Patterns & Practices book *Building a Release Pipeline with TFS 2012*, as a great way to start learning the concepts of DevOps & Release Management for TFS 2012 and to kick the tires. The guidance shows how to build once and deploy to multiple environments.
- Testing for Continuous Delivery with Visual Studio 2012. E-book by Microsoft Patterns and Practices, explains how to integrate automated testing with continuous delivery.
- WindowsAzureDeploymentTracker. Source code for a tool designed to capture a build from TFS (based on a label), build it, package it, allow someone in the DevOps role to configure specific aspects of it, and push it into Azure. The tool tracks the deployment process in order to enable operations to "roll back" to a previously deployed version. The tool has no external dependencies and can function stand-alone using TFS APIs and the Azure SDK.
- Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Book by Jez Humble.
- Release It! Design and Deploy Production-Ready Software. Book by Michael T. Nygard.

PREVIOUS NEXT

Web Development Best Practices (Building Real-World Cloud Apps with Azure)

2/20/2020 • 10 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the e-book, see the first chapter.

The first three patterns were about setting up an agile development process; the rest are about architecture and code. This one is a collection of web development best practices:

- Stateless web servers behind a smart load balancer.
- Avoid session state (or if you can't avoid it, use distributed cache rather than a database).
- Use a CDN to edge-cache static file assets (images, scripts).
- Use .NET 4.5's async support to avoid blocking calls.

These practices are valid for all web development, not just for cloud apps, but they're especially important for cloud apps. They work together to help you make optimal use of the highly flexible scaling offered by the cloud environment. If you don't follow these practices, you'll run into limitations when you try to scale your application.

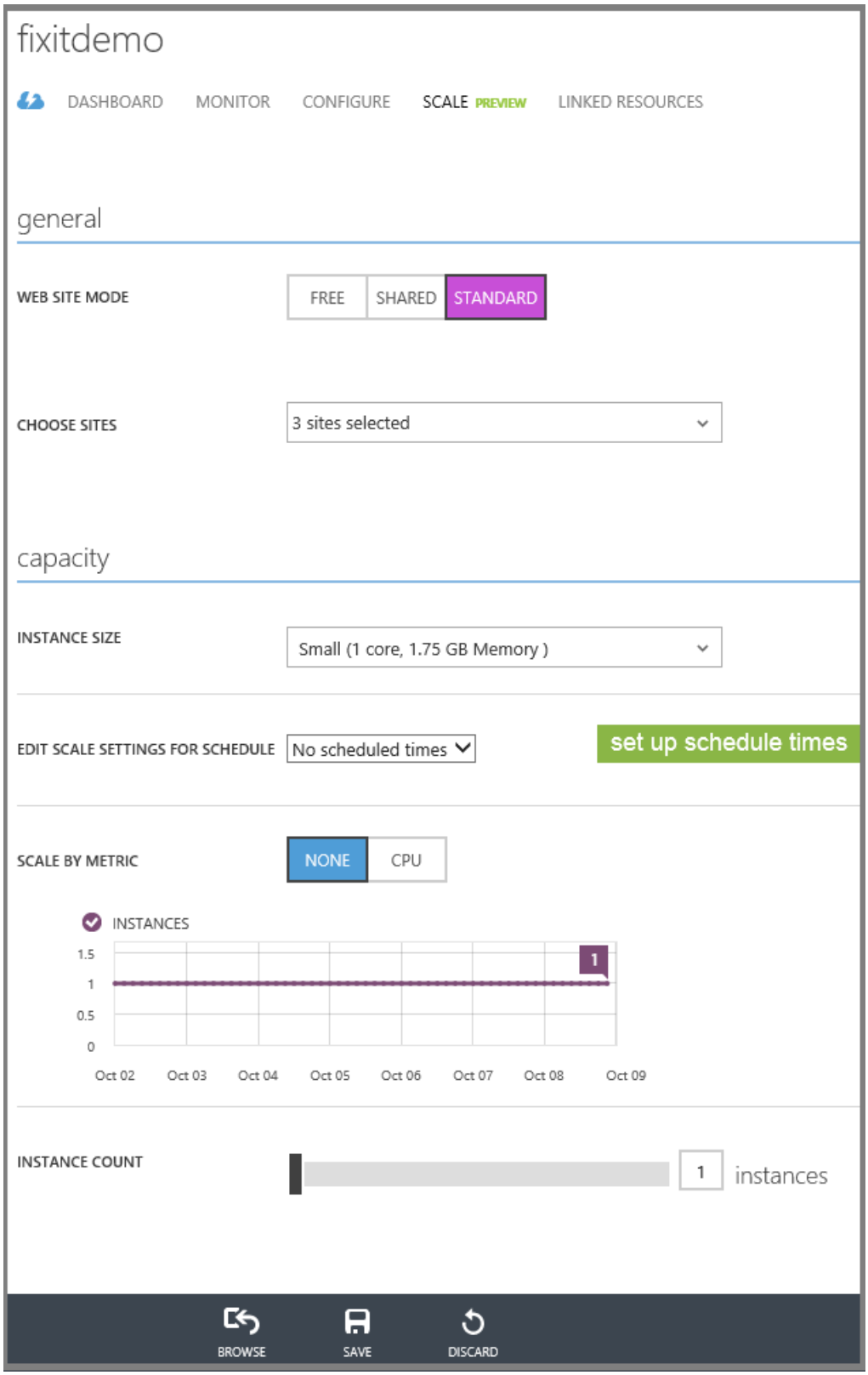
Stateless web tier behind a smart load balancer

Stateless web tier means you don't store any application data in the web server memory or file system. Keeping your web tier stateless enables you to both provide a better customer experience and save money:

- If the web tier is stateless and it sits behind a load balancer, you can quickly respond to changes in application traffic by dynamically adding or removing servers. In the cloud environment where you only pay for server resources for as long as you actually use them, that ability to respond to changes in demand can translate into huge savings.
- A stateless web tier is architecturally much simpler to scale out the application. That too enables you to respond to scaling needs more quickly, and spend less money on development and testing in the process.
- Cloud servers, like on-premises servers, need to be patched and rebooted occasionally; and if the web tier is stateless, re-routing traffic when a server goes down temporarily won't cause errors or unexpected behavior.

Most real-world applications do need to store state for a web session; the main point here is not to store it on the web server. You can store state in other ways, such as on the client in cookies or out of process server-side in ASP.NET session state using a cache provider. You can store files in Windows Azure Blob storage instead of the local file system.

As an example of how easy it is to scale an application in Windows Azure Web Sites if your web tier is stateless, see the **Scale** tab for a Windows Azure Web Site in the management portal:



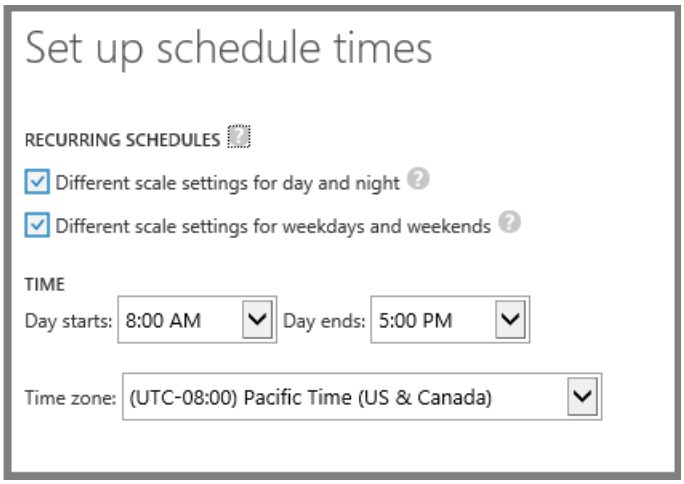
If you want to add web servers, you can just drag the instance count slider to the right. Set it to 5 and click **Save**, and within seconds you have 5 web servers in Windows Azure handling your web site's traffic.

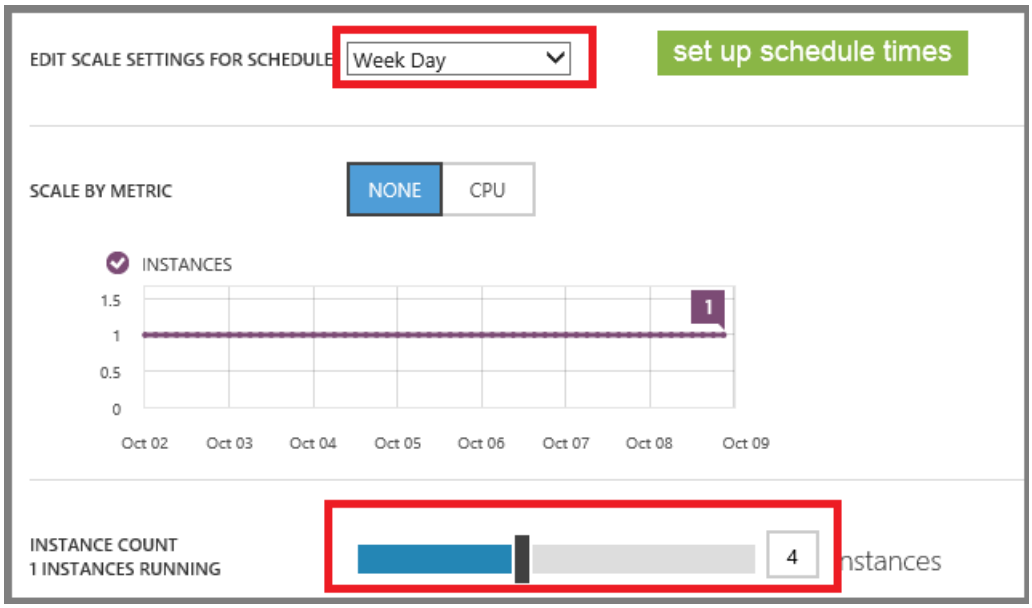
You can just as easily set the instance count down to 3 or back down to 1. When you scale back, you start saving money immediately because Windows Azure charges by the minute, not by the hour.

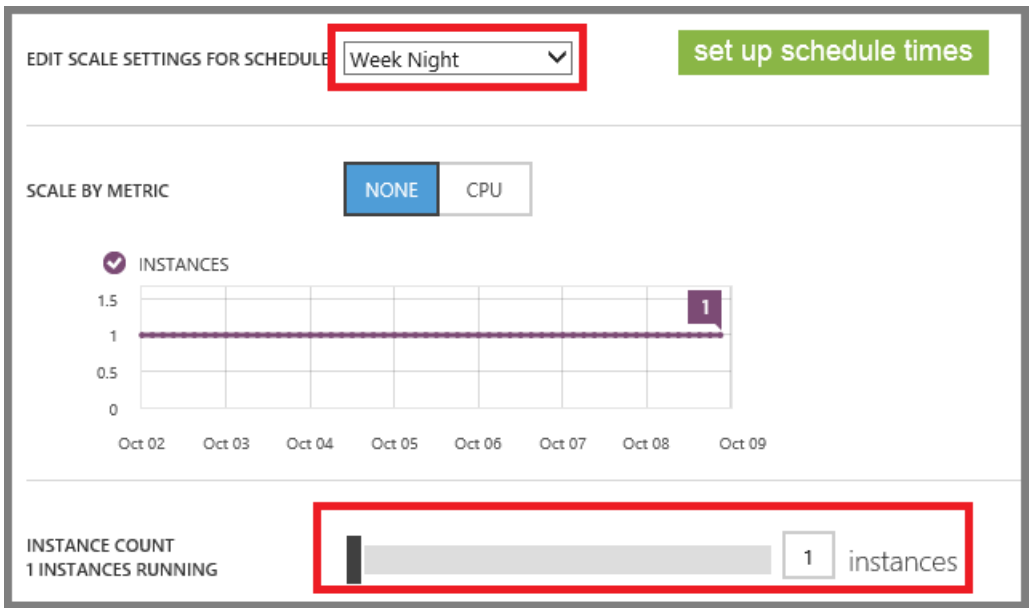
You can also tell Windows Azure to automatically increase or decrease the number of web servers based on CPU usage. In the following example, when CPU usage goes below 60%, the number of web servers will decrease to a

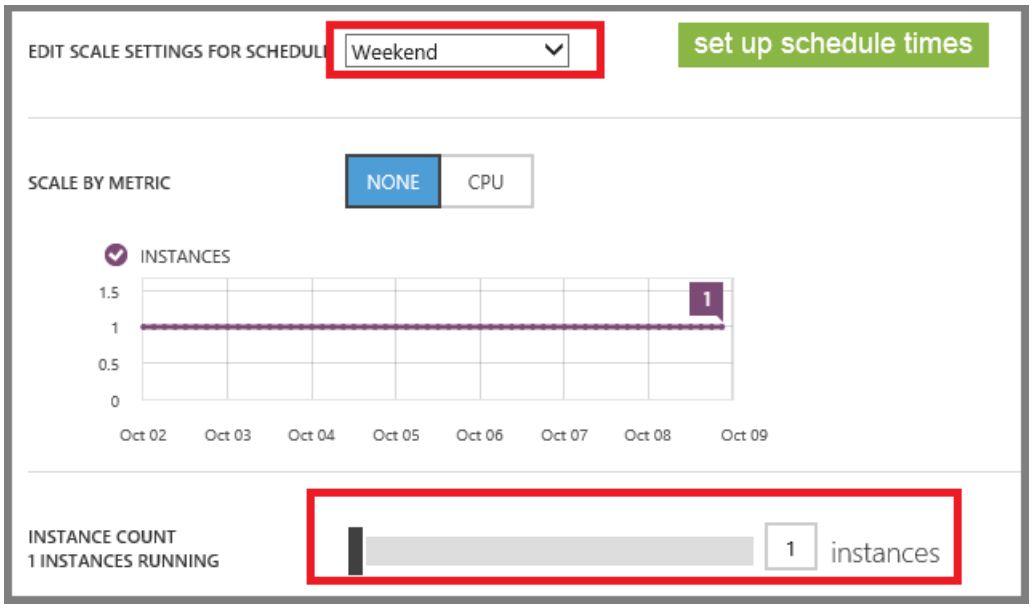
minimum of 2, and if CPU usage goes above 80%, the number of web servers will be increased up to a maximum of 4.

Or what if you know that your site will only be busy during working hours? You can tell Windows Azure to run multiple servers during the daytime and decrease to a single server evenings, nights, and weekends. The following series of screen shots shows how to set up the web site to run one server in off hours and 4 servers during work hours from 8 AM to 5 PM.









And of course all of this can be done in scripts as well as in the portal.

The ability of your application to scale out is almost unlimited in Windows Azure, so long as you avoid impediments to dynamically adding or removing server VMs, by keeping the web tier stateless.

Avoid session state

It's often not practical in a real-world cloud app to avoid storing some form of state for a user session, but some approaches impact performance and scalability more than others. If you have to store state, the best solution is to keep the amount of state small and store it in cookies. If that isn't feasible, the next best solution is to use ASP.NET session state with a provider for distributed, in-memory cache. The worst solution from a performance and scalability standpoint is to use a database backed session state provider.

Use a CDN to cache static file assets

CDN is an acronym for Content Delivery Network. You provide static file assets such as images and script files to a CDN provider, and the provider caches these files in data centers all over the world so that wherever people access your application, they get relatively quick response and low latency for the cached assets. This speeds up the overall load time of the site and reduces the load on your web servers. CDNs are especially important if you are reaching an audience that is widely distributed geographically.

Windows Azure has a CDN, and you can use other CDNs in an application that runs in Windows Azure or any web hosting environment.

Use .NET 4.5's async support to avoid blocking calls

.NET 4.5 enhanced the C# and VB programming languages in order to make it much simpler to handle tasks asynchronously. The benefit of asynchronous programming is not just for parallel processing situations such as when you want to kick off multiple web service calls simultaneously. It also enables your web server to perform more efficiently and reliable under high load conditions. A web server only has a limited number of threads available, and under high load conditions when all of the threads are in use, incoming requests have to wait until threads are freed up. If your application code doesn't handle tasks like database queries and web service calls asynchronously, many threads are unnecessarily tied up while the server is waiting for an I/O response. This limits the amount of traffic the server can handle under high load conditions. With asynchronous programming, threads that are waiting for a web service or database to return data are freed up to service new requests until the data the is received. In a busy web server, hundreds or thousands of requests can then be processed promptly which would otherwise be waiting for threads to be freed up.

As you saw earlier, it's as easy to decrease the number of web servers handling your web site as it is to increase them. So if a server can achieve greater throughput, you don't need as many of them and you can decrease your costs because you need fewer servers for a given traffic volume than you otherwise would.

Support for the .NET 4.5 asynchronous programming model is included in ASP.NET 4.5 for Web Forms, MVC, and Web API; in Entity Framework 6, and in the Windows Azure Storage API.

Async support in ASP.NET 4.5

In ASP.NET 4.5, support for asynchronous programming has been added not just to the language but also to the MVC, Web Forms, and Web API frameworks. For example, an ASP.NET MVC controller action method receives data from a web request and passes the data to a view which then creates the HTML to be sent to the browser. Frequently the action method needs to get data from a database or web service in order to display it in a web page or to save data entered in a web page. In those scenarios it's easy to make the action method asynchronous: instead of returning an *ActionResult* object, you return *Task<ActionResult>* and mark the method with the *async* keyword. Inside the method, when a line of code kicks off an operation that involves wait time, you mark it with the await keyword.

Here is a simple action method that calls a repository method for a database query:

```
public ActionResult Index()
{
    string currentUser = User.Identity.Name;
    var result = fixItRepository.FindOpenTasksByOwner(currentUser);

    return View(result);
}
```

And here is the same method that handles the database call asynchronously:

```
public async Task<ActionResult> Index()
{
    string currentUser = User.Identity.Name;
    var result = await fixItRepository.FindOpenTasksByOwnerAsync(currentUser);

    return View(result);
}
```

Under the covers the compiler generates the appropriate asynchronous code. When the application makes the call to `FindTaskByIdAsync`, ASP.NET makes the `FindTask` request and then unwinds the worker thread and makes it available to process another request. When the `FindTask` request is done, a thread is restarted to continue processing the code that comes after that call. During the interim between when the `FindTask` request is initiated and when the data is returned, you have a thread available to do useful work which otherwise would be tied up waiting for the response.

There is some overhead for asynchronous code, but under low-load conditions, that overhead is negligible, while under high-load conditions you're able to process requests that otherwise would be held up waiting for available threads.

It has been possible to do this kind of asynchronous programming since ASP.NET 1.1, but it was difficult to write, error-prone, and difficult to debug. Now that we've simplified the coding for it in ASP.NET 4.5, there's no reason not to do it anymore.

Async support in Entity Framework 6

As part of async support in 4.5 we shipped async support for web service calls, sockets, and file system I/O, but the most common pattern for web applications is to hit a database, and our data libraries didn't support async. Now Entity Framework 6 adds async support for database access.

In Entity Framework 6 all methods that cause a query or command to be sent to the database have async versions. The example here shows the async version of the *Find* method.

```
public async Task<FixItTask> FindTaskByIdAsync(int id)
{
    FixItTask fixItTask = null;
    Stopwatch timespan = Stopwatch.StartNew();

    try
    {
        fixItTask = await db.FixItTasks.FindAsync(id);

        timespan.Stop();
        log.TraceApi("SQL Database", "FixItTaskRepository.FindTaskByIdAsync", timespan.Elapsed, "id={0}",
id);
    }
    catch(Exception e)
    {
        log.Error(e, "Error in FixItTaskRepository.FindTaskByIdAsynx(id={0})", id);
    }

    return fixItTask;
}
```

And this async support works not just for inserts, deletes, updates, and simple finds, it also works with LINQ queries:

```
public async Task<List<FixItTask>> FindOpenTasksByOwnerAsync(string userName)
{
    Stopwatch timespan = Stopwatch.StartNew();

    try
    {
        var result = await db.FixItTasks
            .Where(t => t.Owner == userName)
            .Where(t=>t.IsDone == false)
            .OrderByDescending(t => t.FixItTaskId).ToListAsync();

        timespan.Stop();
        log.TraceApi("SQL Database", "FixItTaskRepository.FindTasksByOwnerAsync", timespan.Elapsed,
"username={0}", userName);

        return result;
    }
    catch (Exception e)
    {
        log.Error(e, "Error in FixItTaskRepository.FindTasksByOwnerAsync(userName={0})", userName);
        return null;
    }
}
```

There's an `Async` version of the `ToList` method because in this code that's the method that causes a query to be sent to the database. The `Where` and `OrderByDescending` methods only configure the query, while the `ToListAsync` method executes the query and stores the response in the `result` variable.

Summary

You can implement the web development best practices outlined here in any web programming framework and any cloud environment, but we have tools in ASP.NET and Windows Azure to make it easy. If you follow these patterns, you can easily scale out your web tier, and you'll minimize your expenses because each server will be able to handle more traffic.

The next chapter looks at how the cloud enables single sign-on scenarios.

Resources

For more information see the following resources.

Stateless web servers:

- Microsoft Patterns and Practices - Autoscaling guidance.
- Disabling ARR's Instance Affinity in Windows Azure Web Sites. Blog post by Erez Benari, explains session affinity in Windows Azure Web Sites.

CDN:

- FailSafe: Building Scalable, Resilient Cloud Services. Nine-part video series by Ulrich Homann, Marc Mercuri, and Mark Simms. See the CDN discussion in episode 3 starting at 1:34:00.
- Microsoft Patterns and Practices Static Content Hosting pattern
- CDN Reviews. Overview of many CDNs.

Asynchronous programming:

- Using Asynchronous Methods in ASP.NET MVC 4. Tutorial by Rick Anderson.
- Asynchronous Programming with Async and Await (C# and Visual Basic). MSDN white paper that explains rationale for asynchronous programming, how it works in ASP.NET 4.5, and how to write code to implement it.
- Entity Framework Async Query and Save
- How to Build ASP.NET Web Applications Using Async. Video presentation by Rowan Miller. Includes a graphic demonstration of how asynchronous programming can facilitate dramatic increases in web server throughput under high load conditions.
- FailSafe: Building Scalable, Resilient Cloud Services. Nine-part video series by Ulrich Homann, Marc Mercuri, and Mark Simms. For discussions about the impact of asynchronous programming on scalability, see episode 4 and episode 8.
- The Magic of using Asynchronous Methods in ASP.NET 4.5 plus an important gotcha. Blog post by Scott Hanselman, primarily about using async in ASP.NET Web Forms applications.

For additional web development best practices, see the following resources:

- The Fix It Sample Application - Best Practices. The appendix to this e-book lists a number of best practices that were implemented in the Fix It application.
- Web Developer Checklist

PREVIOUS NEXT

Single Sign-On (Building Real-World Cloud Apps with Azure)

2/20/2020 • 8 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the e-book, see the first chapter.

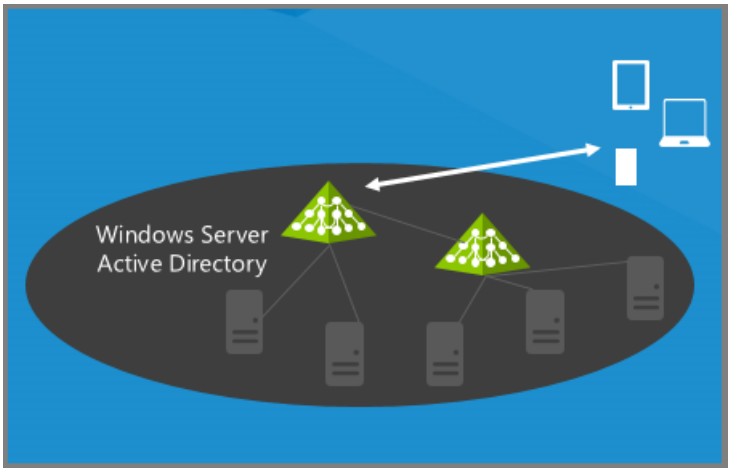
There are many security issues to think about when you're developing a cloud app, but for this series we'll focus on just one: single sign-on. A question people often ask is this: "I'm primarily building apps for the employees of my company; how do I host these apps in the cloud and still enable them to use the same security model that my employees know and use in the on-premises environment when they're running apps that are hosted inside the firewall?" One of the ways we enable this scenario is called Azure Active Directory (Azure AD). Azure AD enables you to make enterprise line-of-business (LOB) apps available over the Internet, and it enables you to make these apps available to business partners as well.

Introduction to Azure AD

Azure AD provides Active Directory in the cloud. Key features include the following:

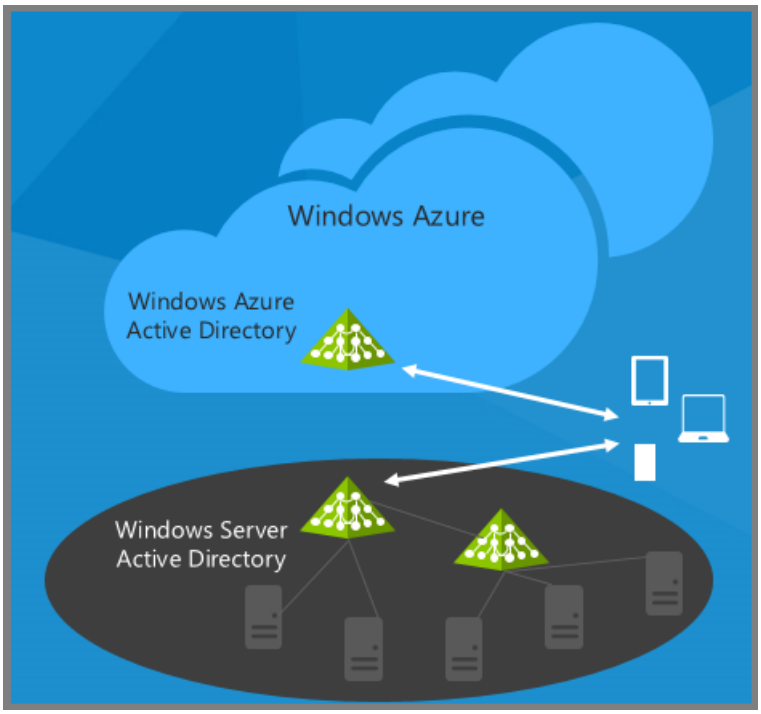
- It integrates with on-premises Active Directory.
- It enables single sign-on with your apps.
- It supports open standards such as SAML, WS-Fed, and OAuth 2.0.
- It supports Enterprise Graph REST API.

Suppose you have an on-premises Windows Server Active Directory environment that you use to enable employees to sign on to Intranet apps:

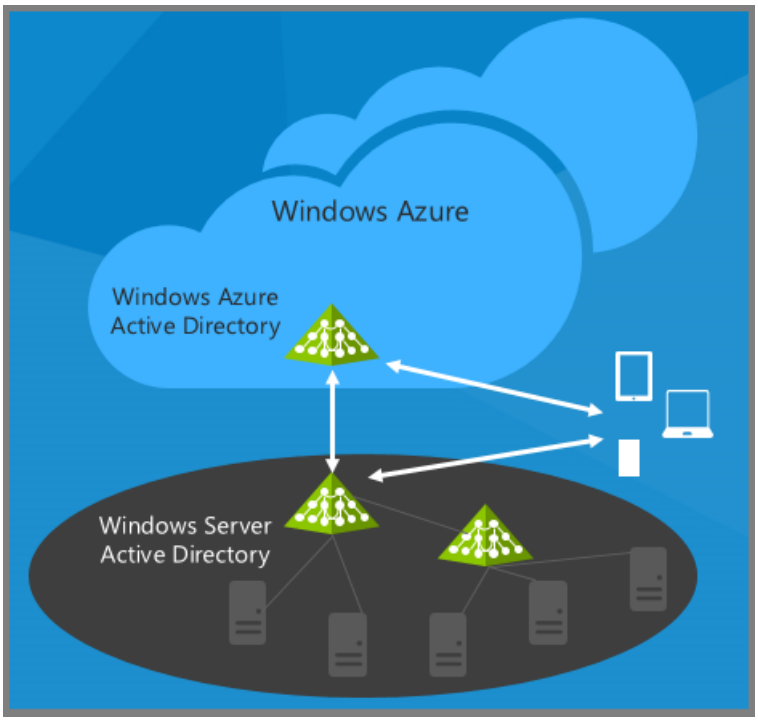


What Azure AD enables you to do is create a directory in the cloud. It's a free feature and easy to set up.

It can be entirely independent from your on-premises Active Directory; you can put anyone you want in it and authenticate them in Internet apps.

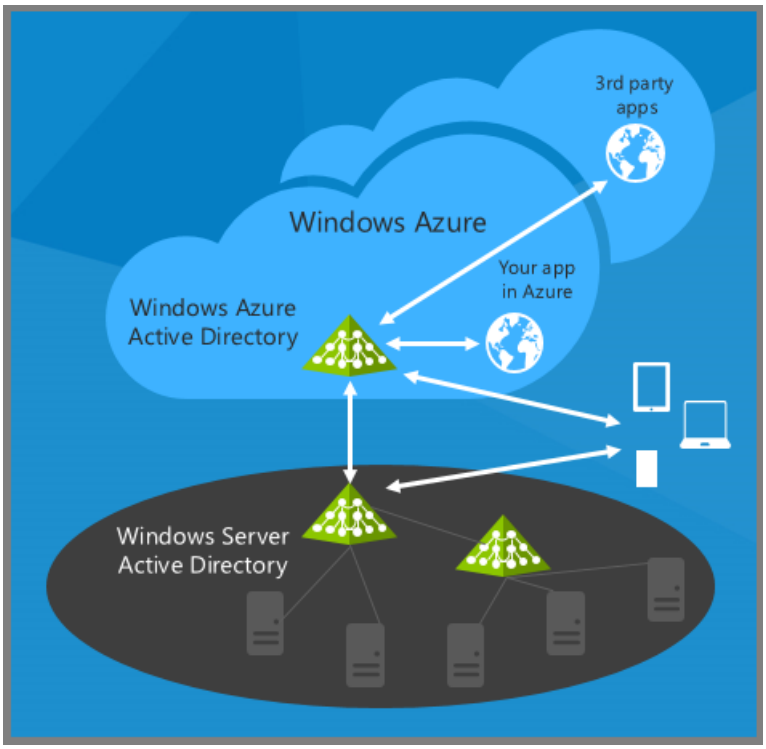


Or you can integrate it with your on-premises AD.



Now all the employees who can authenticate on-premises can also authenticate over the Internet – without you having to open up a firewall or deploy any new servers in your data center. You can continue to leverage all the existing Active Directory environment that you know and use today to give your internal apps single-sign on capability.

Once you've made this connection between AD and Azure AD, you can also enable your web apps and your mobile devices to authenticate your employees in the cloud, and you can enable third-party apps, such as Office 365, SalesForce.com, or Google apps, to accept your employees' credentials. If you're using Office 365, you're already set up with Azure AD because Office 365 uses Azure AD for authentication and authorization.



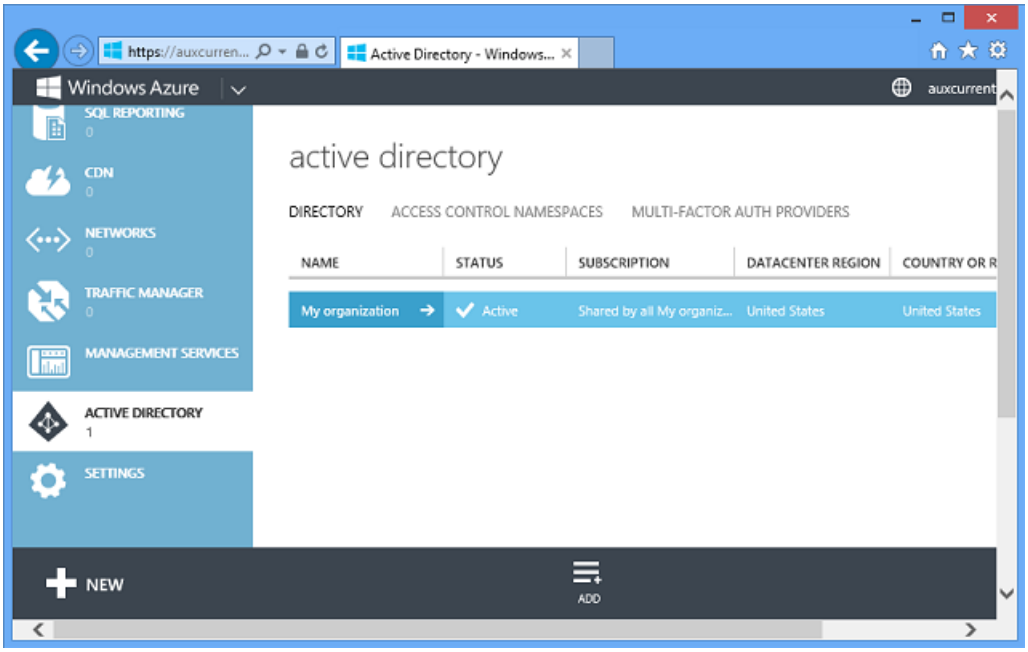
The beauty of this approach is that any time your organization adds or deletes a user, or a user changes a password, you use the same process that you use today in your on-premises environment. All of your on-premises AD changes are automatically propagated to the cloud environment.

If your company is using or moving to Office 365, the good news is that you'll have Azure AD set up automatically because Office 365 uses Azure AD for authentication. So you can easily use in your own apps the same authentication that Office 365 uses.

Set up an Azure AD tenant

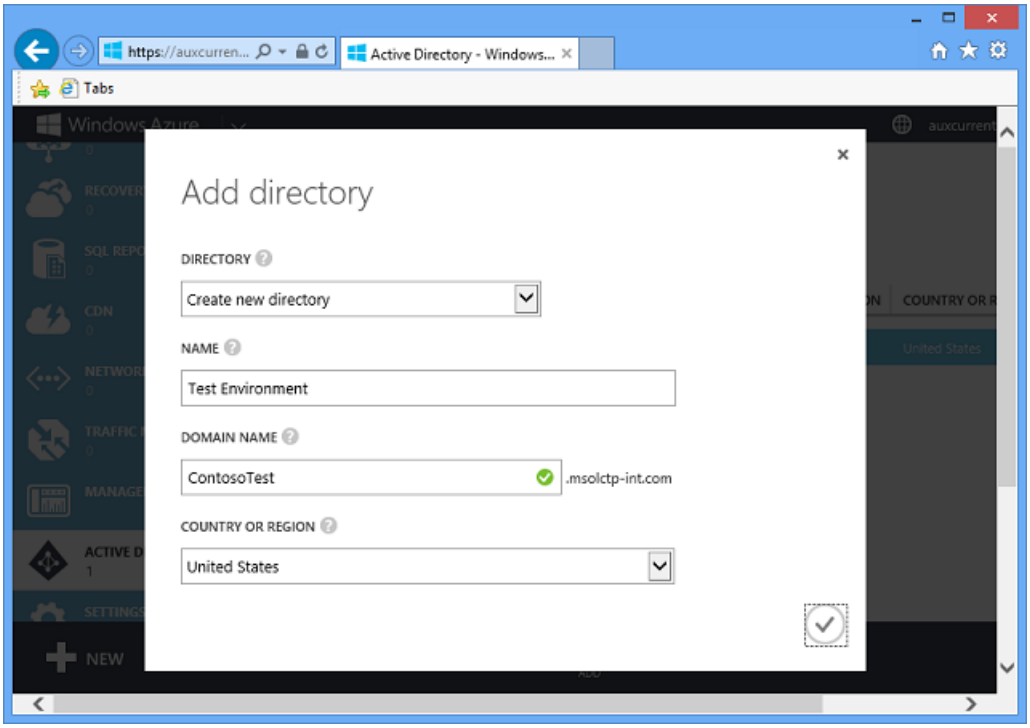
an Azure AD directory is called an Azure AD tenant, and setting up a tenant is pretty easy. We'll show you how it's done in the Azure Management Portal in order to illustrate the concepts, but of course like the other portal functions you can also do it by using a script or management API.

In the management portal click the Active Directory tab.

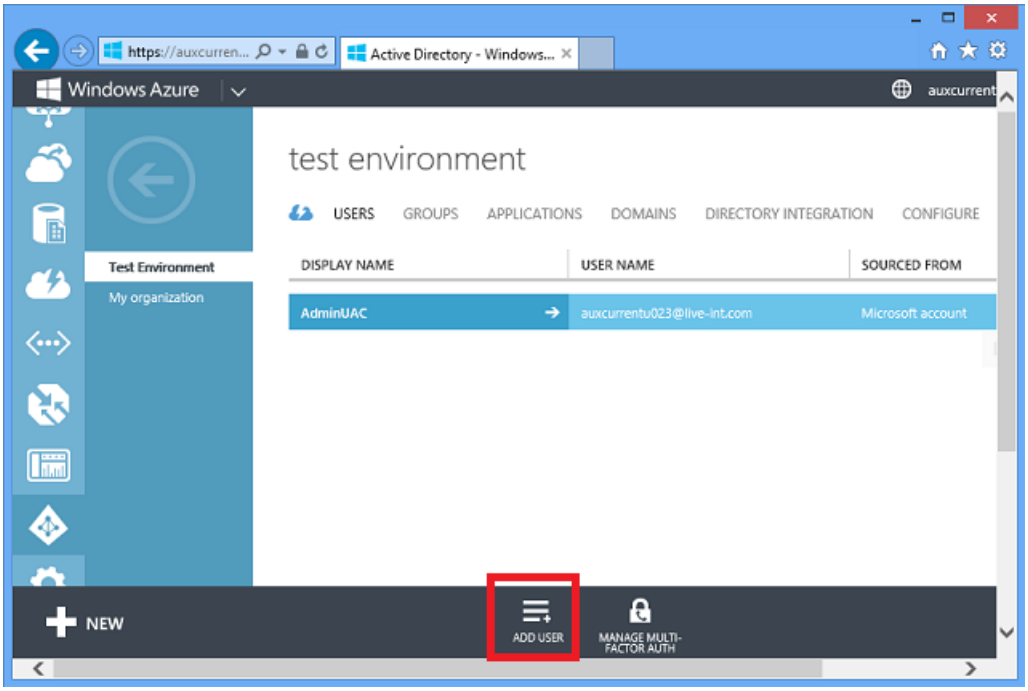


You automatically have one Azure AD tenant for your Azure account, and you can click the **Add** button at the

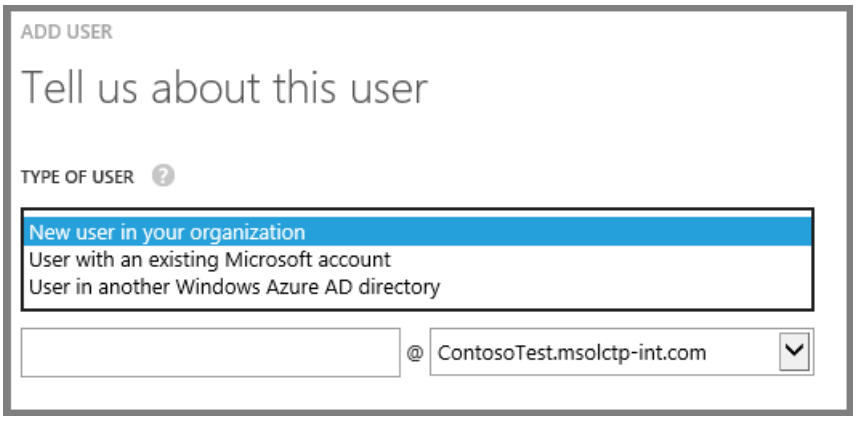
bottom of the page to create additional directories. You might want one for a test environment and one for production, for example. Think carefully about what you name a new directory. If you use your name for the directory and then you use your name again for one of the users, that can be confusing.

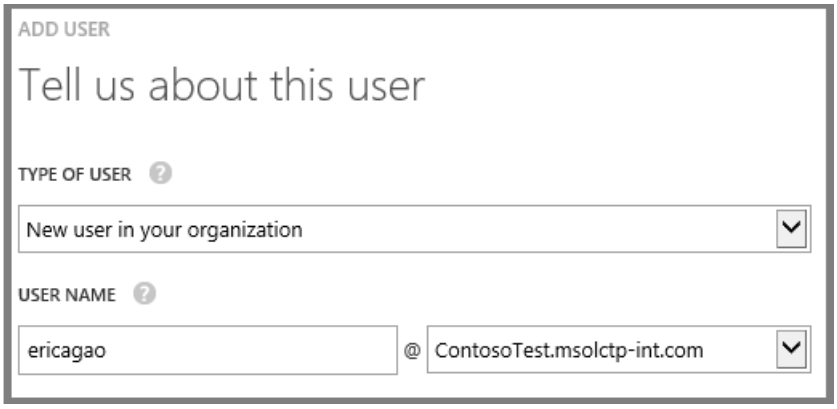


The portal has full support for creating, deleting, and managing users within this environment. For example, to add a user go to the **Users** tab and click the **Add User** button.

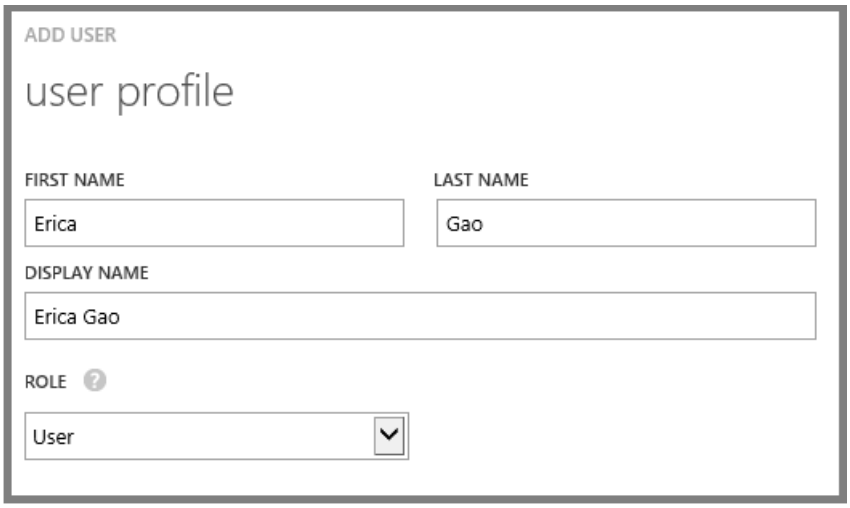


You can create a new user who exists only in this directory, or you can register a Microsoft Account as a user in this directory, or register or a user from another Azure AD directory as a user in this directory. (In a real directory, the default domain would be ContosoTest.onmicrosoft.com. You can also use a domain of your own choosing, like contoso.com.)

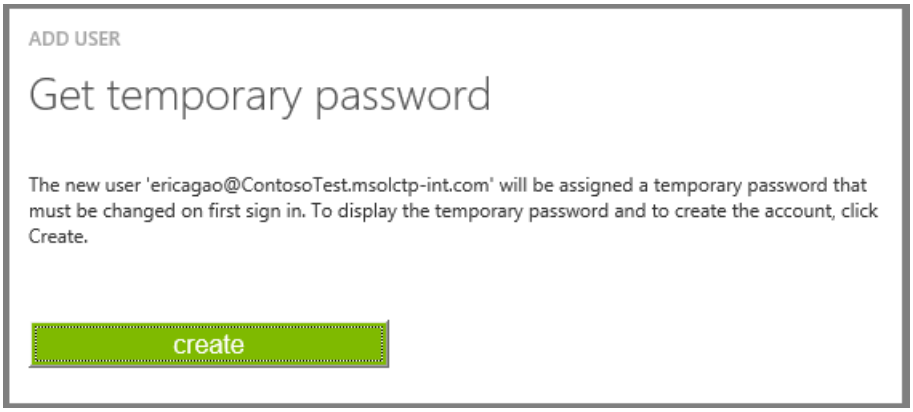




You can assign the user to a role.

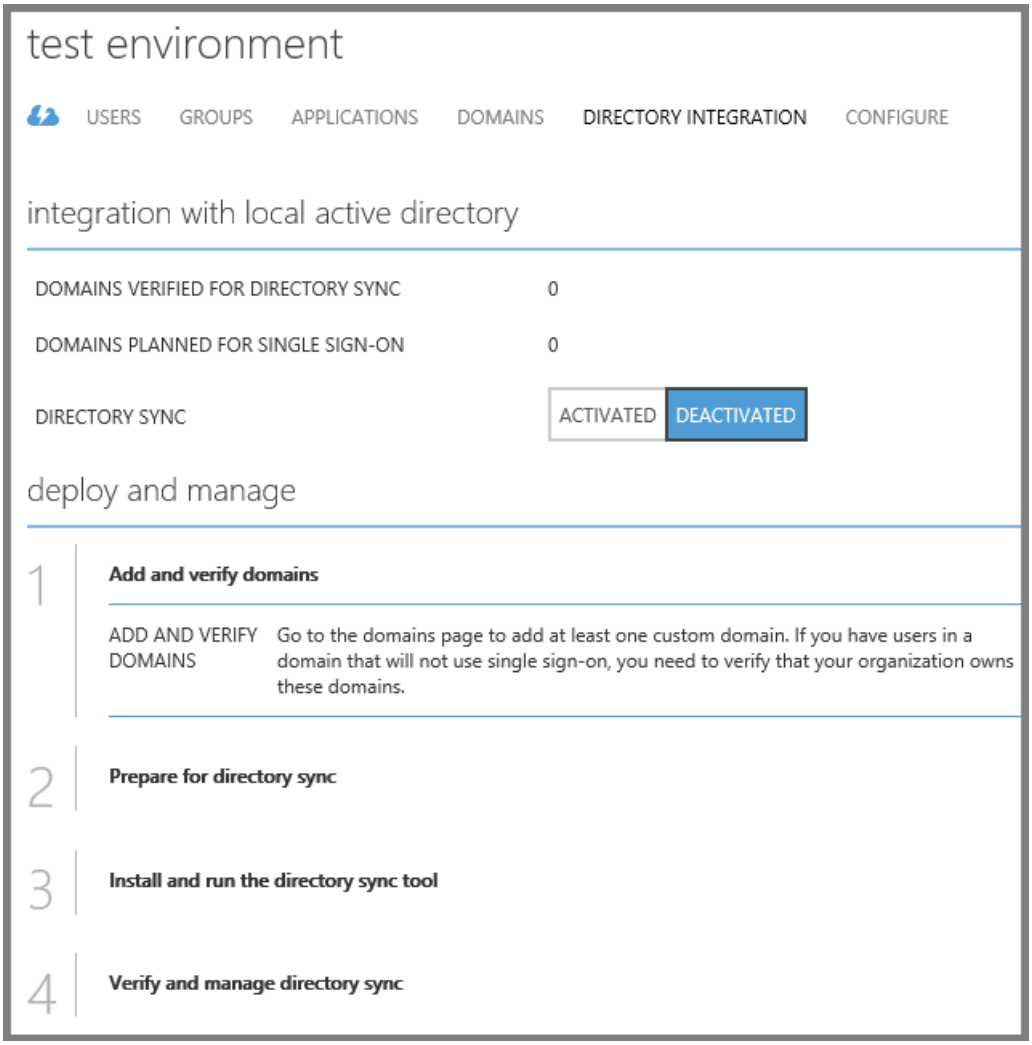


And the account is created with a temporary password.



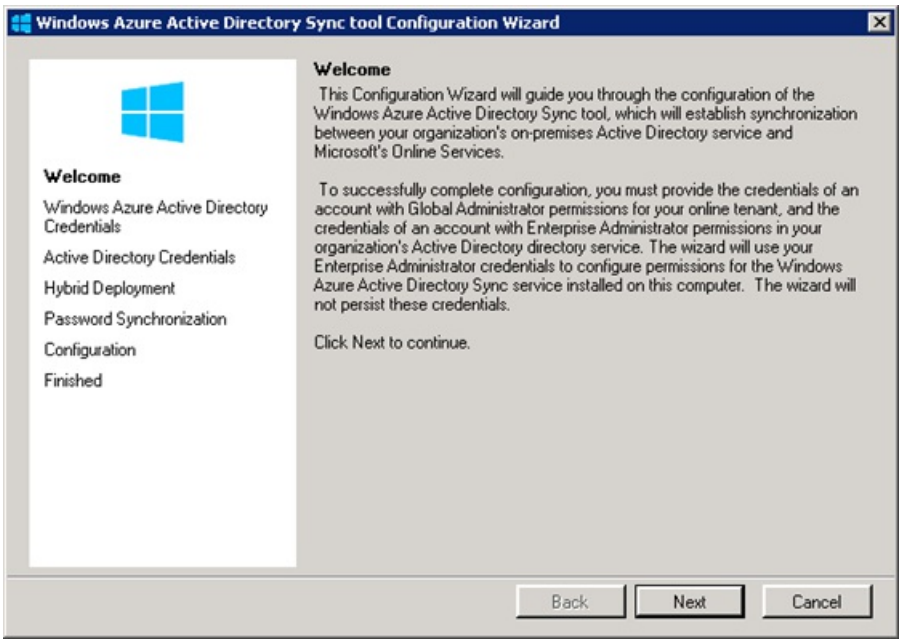
The users you create this way can immediately log in to your web apps using this cloud directory.

What's great for enterprise single sign-on, though, is the **Directory Integration** tab:

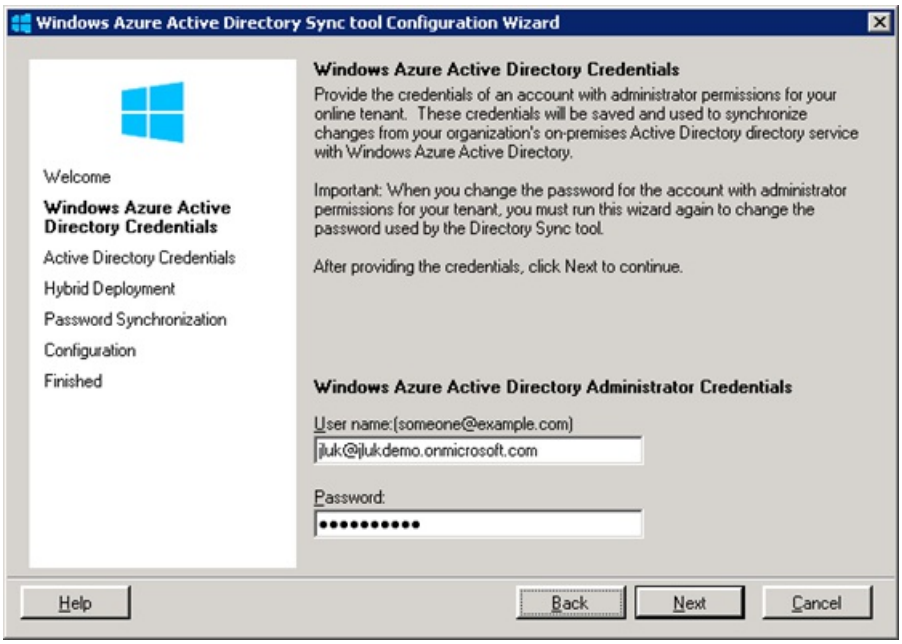


If you enable directory integration, and download a tool, you can sync this cloud directory with your existing on-premises Active Directory that you're already using inside your organization. Then all of the users stored in your directory will show up in this cloud directory. Your cloud apps can now authenticate all of your employees using their existing Active Directory credentials. And all this is free – both the sync tool and Azure AD itself.

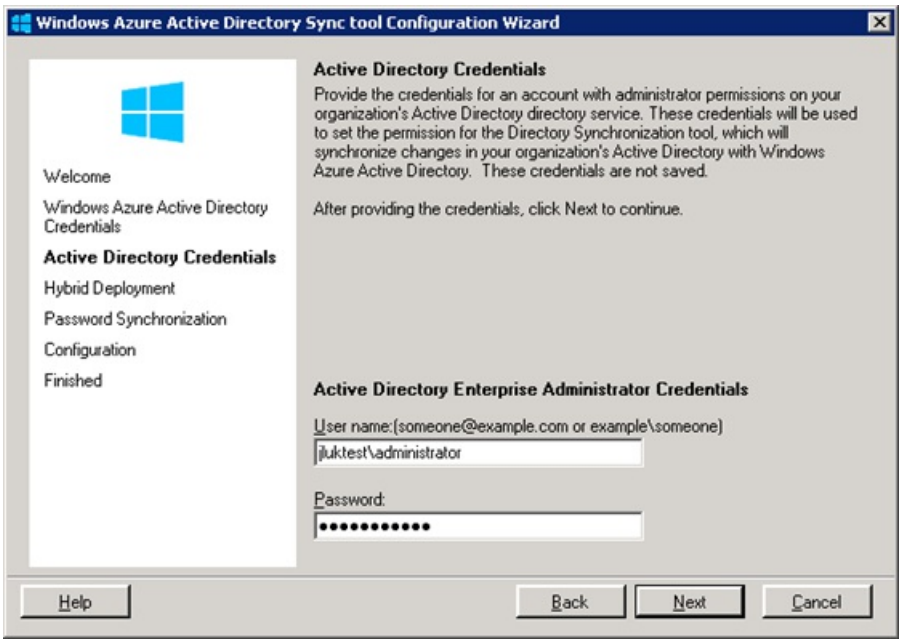
The tool is a wizard that is easy to use, as you can see from these screen shots. These are not complete instructions, just an example showing you the basic process. For more detailed how-to-do-it information, see the links in the Resources section at the end of the chapter.



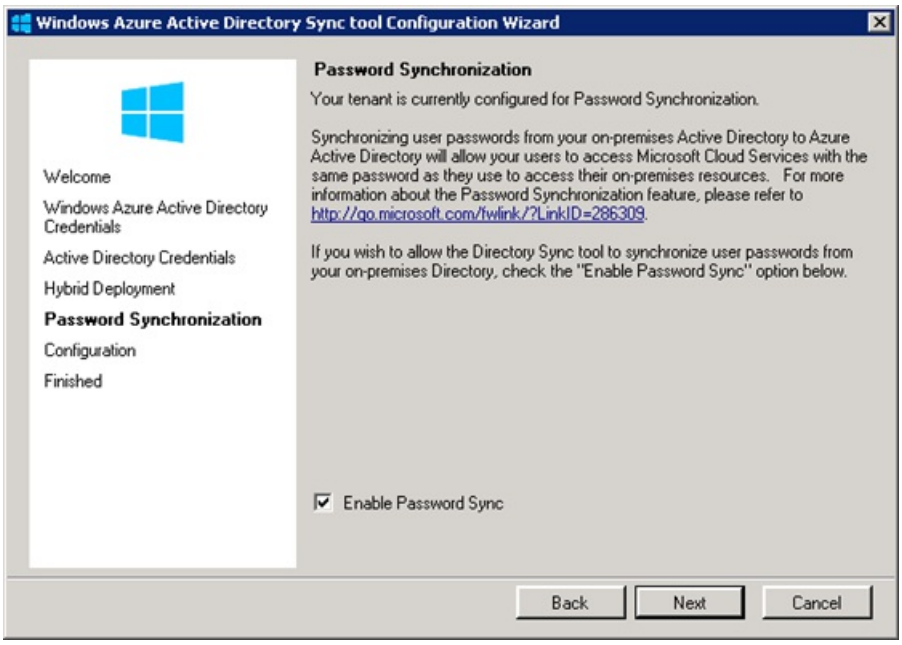
Click **Next**, and then enter your Azure Active Directory credentials.



Click **Next**, and then enter your on-premises AD credentials.

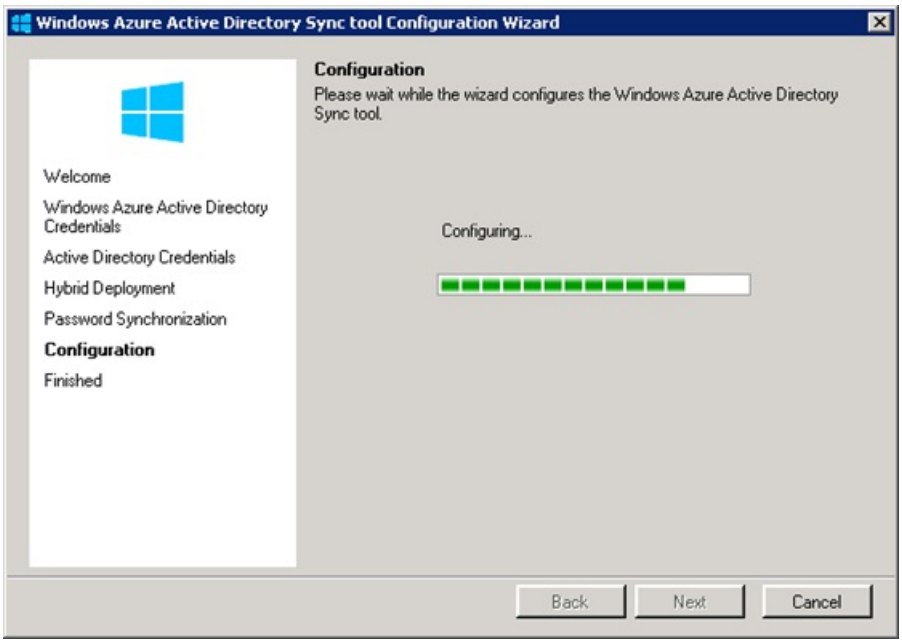


Click **Next**, and then indicate if you want to store a hash of your AD passwords in the cloud.

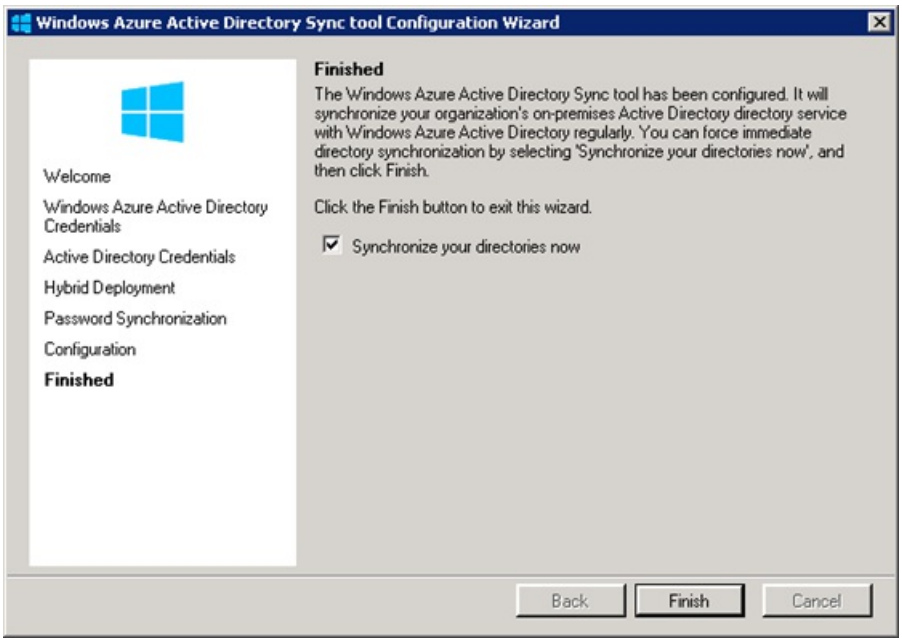


The password hash that you can store in the cloud is a one-way hash; actual passwords are never stored in Azure AD. If you decide against storing hashes in the cloud, you'll have to use Active Directory Federation Services (ADFS). There are also other factors to consider when choosing whether or not to use ADFS. The ADFS option requires a few additional configuration steps.

If you choose to store hashes in the cloud, you're done, and the tool starts synchronizing directories when you click **Next**.

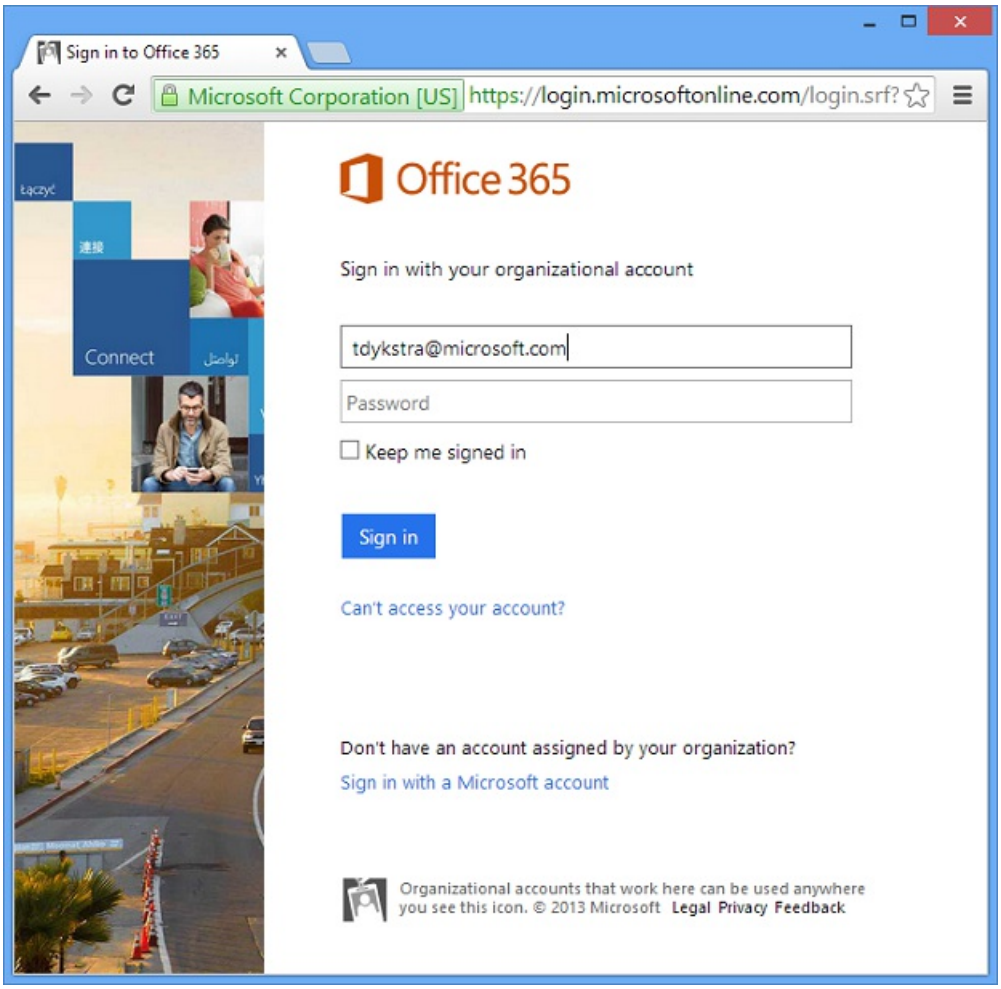


And in a few minutes you're done.

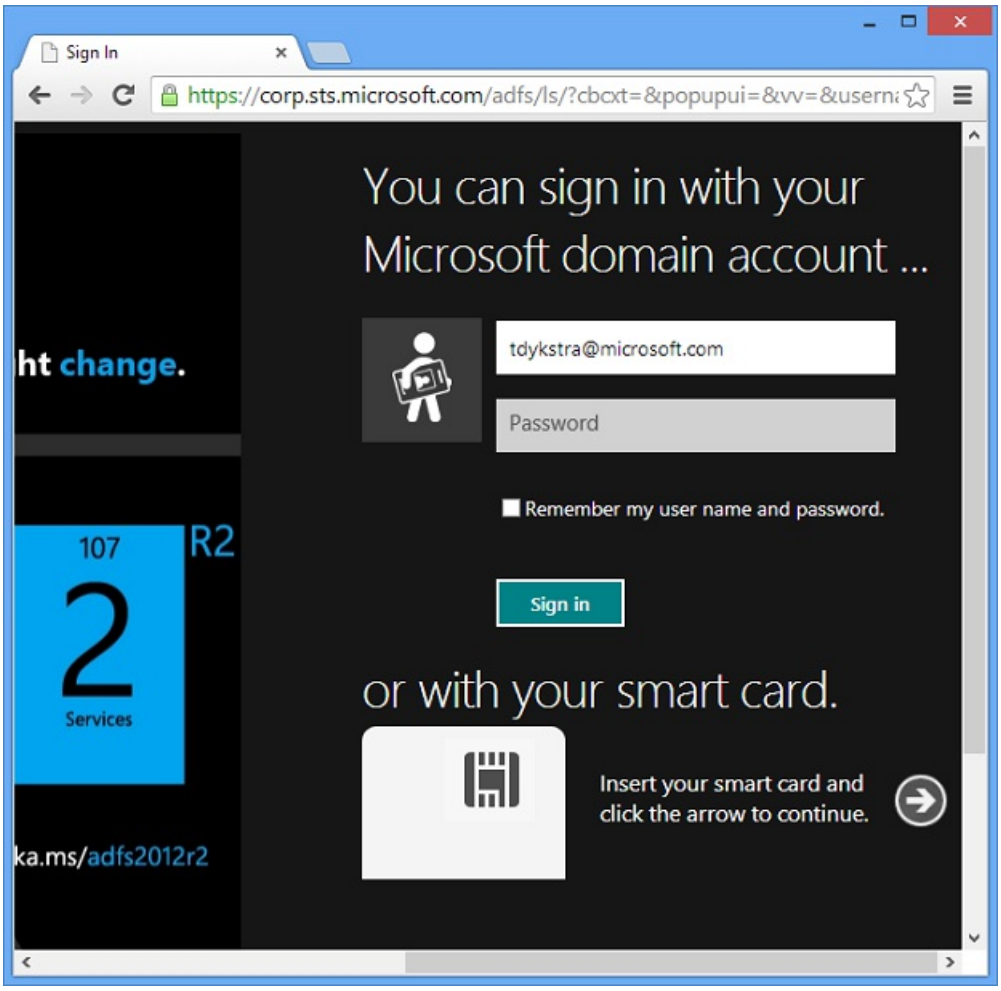


You only have to run this on one domain controller in the organization, on Windows 2003 or higher. And no need to reboot. When you're done, all of your users are in the cloud and you can do single sign-on from any web or mobile application, using SAML, OAuth, or WS-Fed.

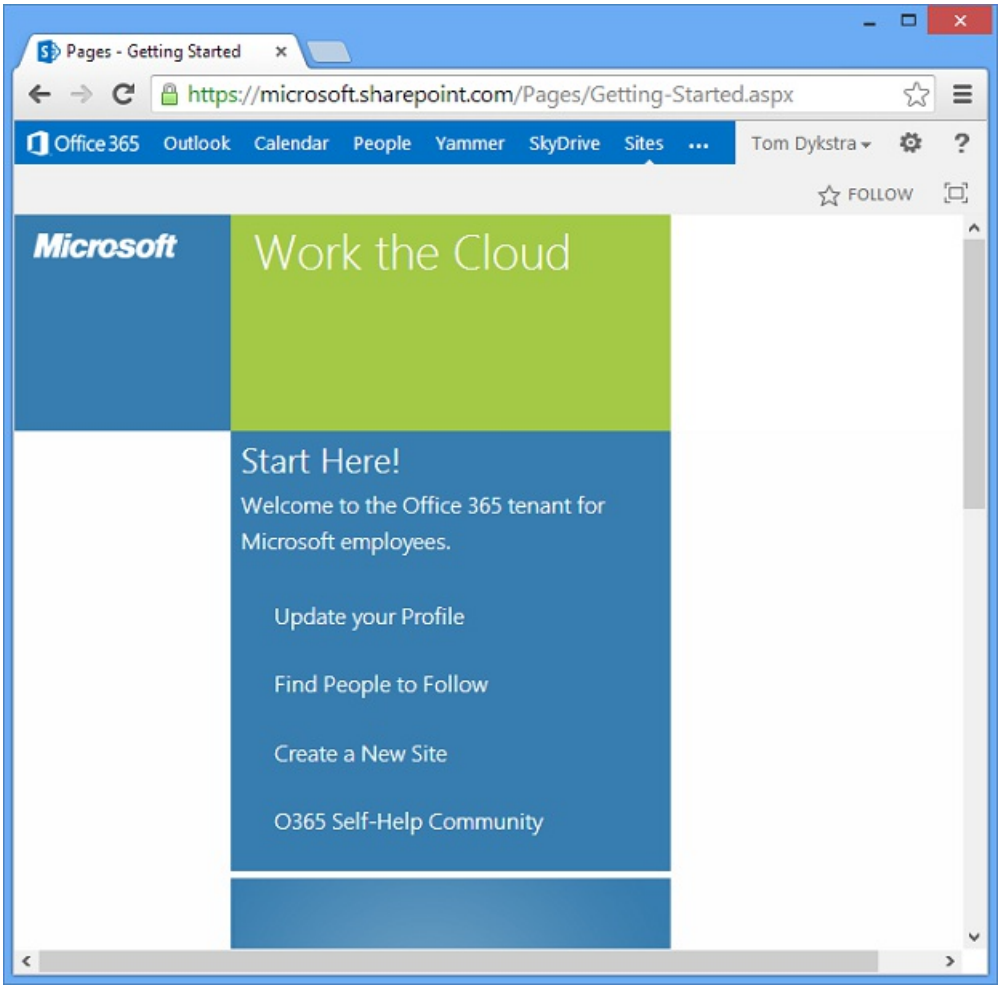
Sometimes we get asked about how secure this is – does Microsoft use it for their own sensitive business data? And the answer is yes we do. For example, if you go to the internal Microsoft SharePoint site at https://microsoft.sharepoint.com/, you get prompted to log in.



Microsoft has enabled ADFS, so when you enter a Microsoft ID, you get redirected to an ADFS log-in page.



And once you enter credentials stored in an internal Microsoft AD account, you have access to this internal application.

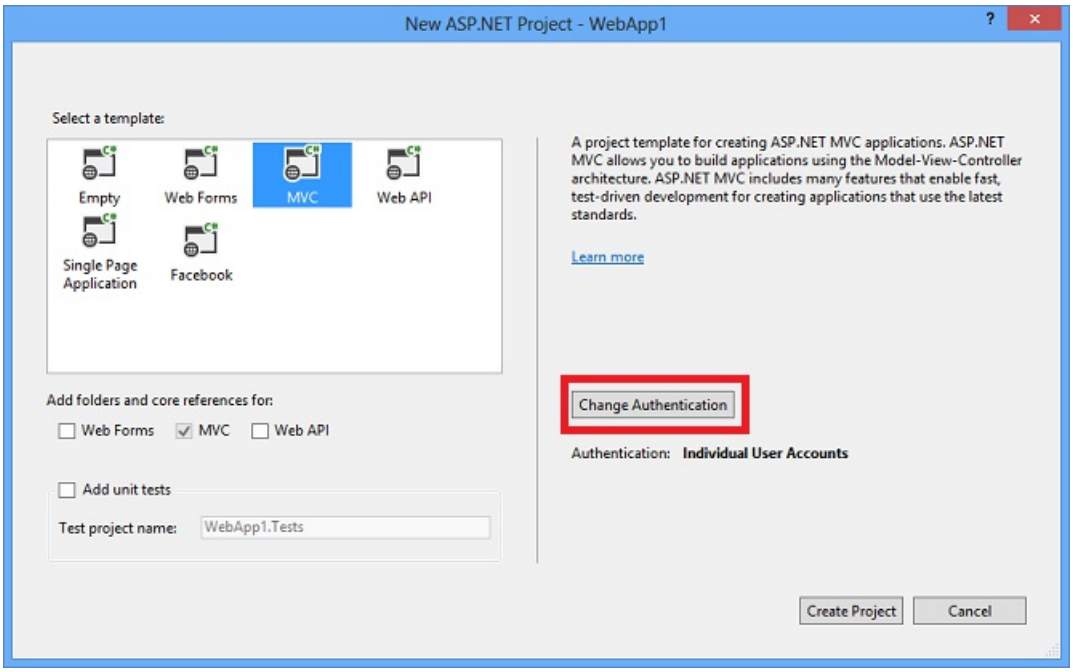


We're using an AD sign-in server mainly because we already had ADFS set up before Azure AD became available, but the log-in process is going through an Azure AD directory in the cloud. We put our important documents, source control, performance management files, sales reports, and more, in the cloud and are using this exact same solution to secure them.

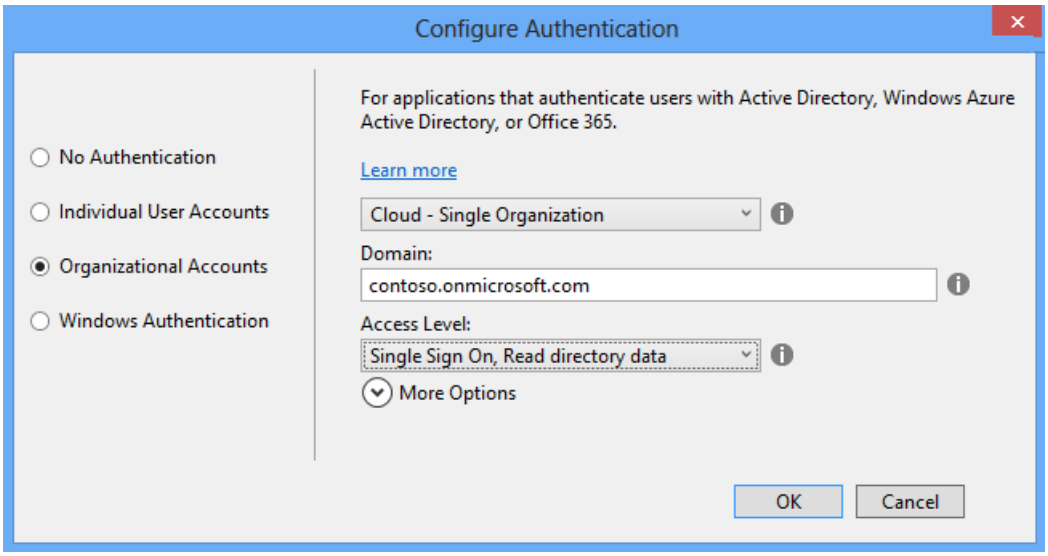
Create an ASP.NET app that uses Azure AD for single sign-on

Visual Studio makes it really easy to create an app that uses Azure AD for single sign-on, as you can see from a few screen shots.

When you create a new ASP.NET application, either MVC or Web Forms, the default authentication method is ASP.NET Identity. To change that to Azure AD, you click a **Change Authentication** button.



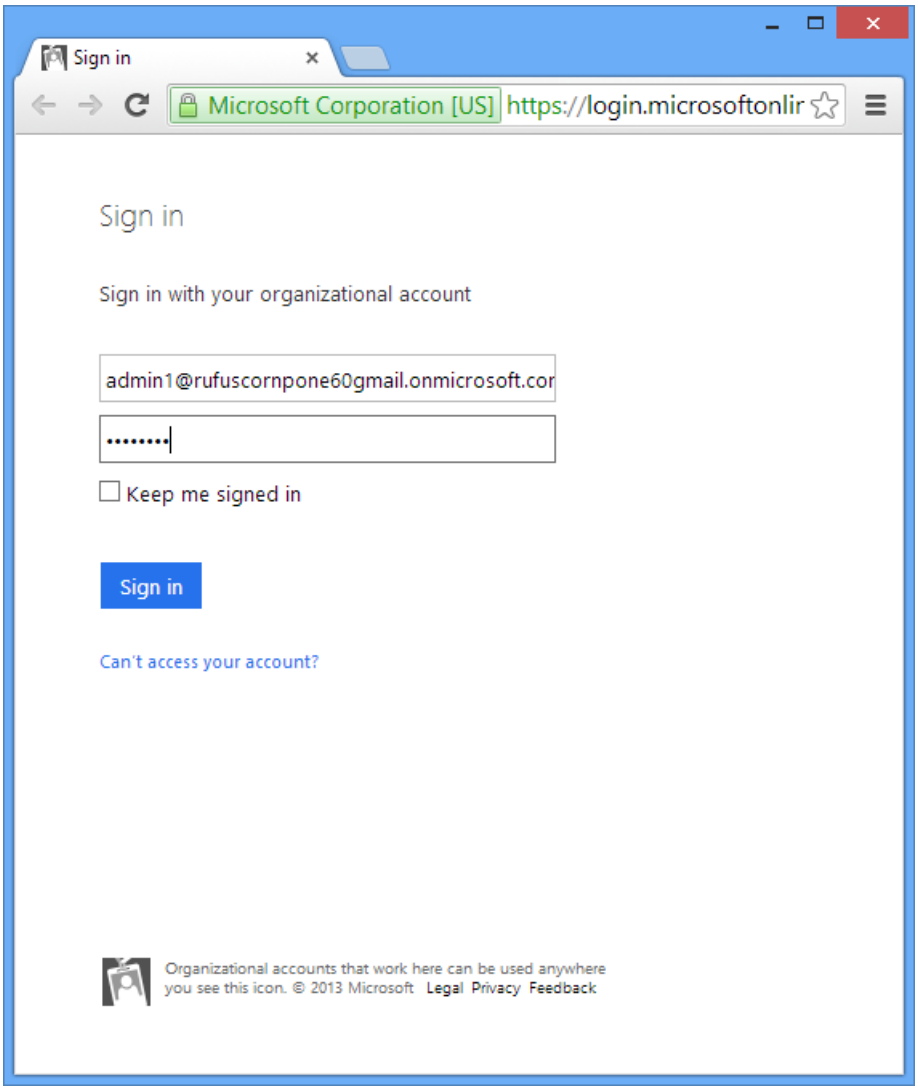
Select Organizational Accounts, enter your domain name, and then select Single Sign On.

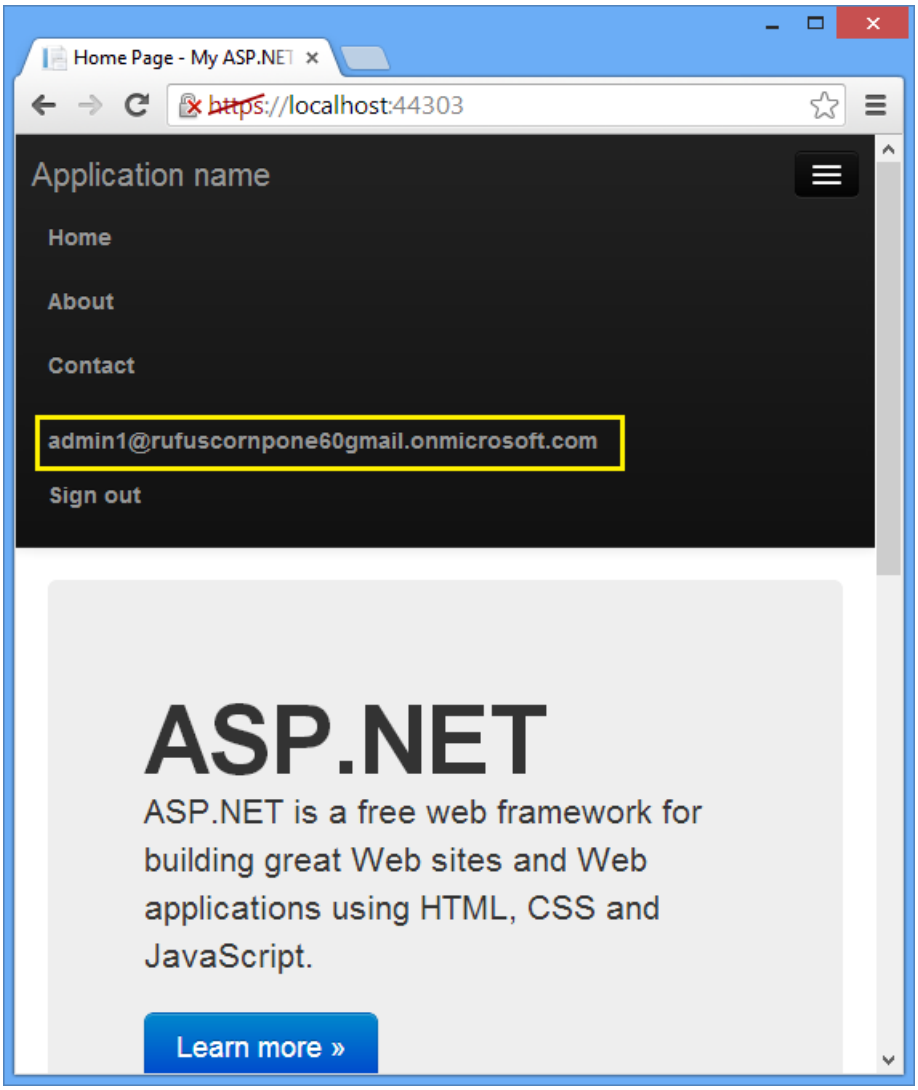


You can also give the app read or read/write permission for directory data. If you do that, it can use the Azure Graph REST API to look up users' phone number, find out if they're in the office, when they last logged on, etc.

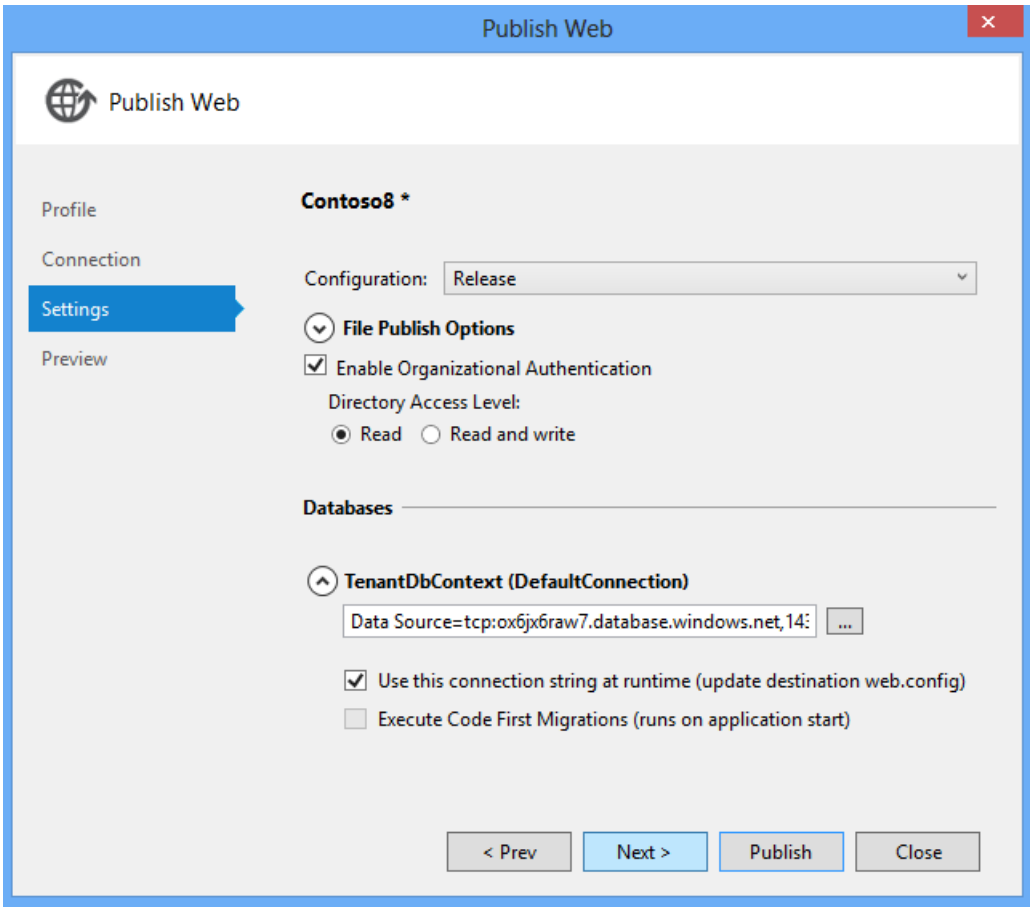
That's all you have to do - Visual Studio asks for the credentials for an administrator for your Azure AD tenant, and then it configures both your project and your Azure AD tenant for the new application.

When you run the project, you'll see a sign-in page, and you can sign in with credentials of a user in your Azure AD directory.


Sign in
Microsoft Corporation [US]
https://login.microsoftonlir
Sign in
Sign in with your organizational account
admin1@rufuscornpone60gmail.onmicrosoft.cor
Keep me signed in
Sign in
Can't access your account?
Organizational accounts that work here can be used anywhere
you see this icon. © 2013 Microsoft
Legal
Privacy
Feedback



When you deploy the app to Azure, all you have to do is select an **Enable Organizational Authentication** check box, and once again Visual Studio takes care of all the configuration for you.



These screen shots come from a complete step-by-step tutorial that shows how to build an app that uses Azure AD authentication: Developing ASP.NET Apps with Azure Active Directory.

Summary

In this chapter you saw that Azure Active Directory, Visual Studio, and ASP.NET, make it easy to set up single sign-on in Internet applications for your organization's users. Your users can sign on in Internet apps using the same credentials they use to sign on using Active Directory in your internal network.

The next chapter looks at the data storage options available for a cloud app.

Resources

For more information, see the following resources:

- Azure Active Directory Documentation. Portal page for Azure AD documentation on the windowsazure.com site. For step by step tutorials, see the **Develop** section.
- Azure Multi-Factor Authentication. Portal page for documentation about multi-factor authentication in Azure.
- Organizational account authentication options. Explanation of the Azure AD authentication options in the Visual Studio 2013 new-project dialog.
- Microsoft Patterns and Practices - Federated Identity Pattern.
- HowTo: Install the Azure Active Directory Sync Tool.
- Active Directory Federation Services 2.0 Content Map. Links to documentation about ADFS 2.0.
- Role-Based and ACL-Based Authorization in a Windows Azure AD Application. Sample application.
- Azure Active Directory Graph API blog.
- Access Control in BYOD and Directory Integration in a Hybrid Identity Infrastructure. Tech Ed 2014 session video by Gayana Bagdasaryan.

Data Storage Options (Building Real-World Cloud Apps with Azure)

2/20/2020 • 26 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the e-book, see the first chapter.

Most people are used to relational databases, and they tend to overlook other data storage options when they're designing a cloud app. The result can be suboptimal performance, high expenses, or worse, because NoSQL (non-relational) databases can handle some tasks more efficiently than relational databases. When customers ask us for help resolving a critical data storage problem, it's often because they have a relational database where one of the NoSQL options would have worked better. In those situations the customer would have been better off if they had implemented the NoSQL solution before deploying the app to production.

On the other hand, it would also be a mistake to assume that a NoSQL database can do everything well or well enough. There is no single best data management choice for all data storage tasks; different data management solutions are optimized for different tasks. Most real-world cloud apps have a variety of data storage requirements and are often served best by a combination of multiple data storage solutions.

The purpose of this chapter is to give you a broader sense of the data storage options available to a cloud app, and some basic guidance on how to choose the ones that fit your scenario. It's best to be aware of the options available to you and think about their strengths and weaknesses before you develop an application. Changing data storage options in a production app can be extremely difficult, like having to change a jet engine while the plane is in flight.

Data storage options on Azure

The cloud makes it relatively easy to use a variety of relational and NoSQL data stores. Here are some of the data storage platforms that you can use in Azure.

Relational	Key/Value	Column Family	Document	Graph
• Windows Azure SQL Database • SQL Server • Oracle • MySQL • SQL Compact • SQLite • Postgres	• Windows Azure Blob Storage • Windows Azure Table Storage • Windows Azure Cache • Redis • Memcached • Riak	• Cassandra • HBase	• MongoDB • RavenDB • CouchDB	• Neo4J

The table shows four types of NoSQL databases:

- Key/value databases store a single serialized object for each key value. They're good for storing large volumes of data where you want to get one item for a given key value and you don't have to query based on other properties of the item.

  Azure Blob storage is a key/value database that functions like file storage in the cloud, with key values that correspond to folder and file names. You retrieve a file by its folder and file name, not by searching for values in the file contents.

  Azure Table storage is also a key/value database. Each value is called an *entity* (similar to a row, identified by a partition key and row key) and contains multiple *properties* (similar to columns, but not all entities in a table have to share the same columns). Querying on columns other than the key is extremely inefficient and should be avoided. For example, you can store user profile data, with one partition storing information about a single user. You could store data such as user name, password hash, birth date, and so forth, in separate properties of one entity or in separate entities in the same partition. But you wouldn't want to query for all users with a given range of birth dates, and you can't execute a join query between your profile table and another table. Table storage is more scalable and less expensive than a relational database, but it doesn't enable complex queries or joins.

- Documentdatabases are key/value databases in which the values are *documents*. "Document" here isn't used in the sense of a Word or Excel document but means a collection of named fields and values, any of which could be a child document. For example, in an order history table an order document might have order number, order date, and customer fields; and the customer field might have name and address fields. The database encodes field data in a format such as XML, YAML, JSON, or BSON; or it can use plain text. One feature that sets document databases apart from key/value databases is the ability to query on non-key fields and define secondary indexes to make querying more efficient. This ability makes a document database more suitable for applications that need to retrieve data based on criteria more complex than the value of the document key. For example, in a sales order history document database you could query on various fields such as product ID, customer ID, customer name, and so forth. MongoDB is a popular document database.

- Column-family databases are key/value data stores that enable you to structure data storage into collections of related columns called column families. For example, a census database might have one group of columns for a person's name (first, middle, last), one group for the person's address, and one group for the person's profile information (DOB, gender, etc.). The database can then store each column family in a separate partition while keeping all of the data for one person related to the same key. You can then read all profile information without having to read through all of the name and address information

as well. Cassandra is a popular column-family database.

- Graph databases store information as a collection of objects and relationships. The purpose of a graph database is to enable an application to efficiently perform queries that traverse the network of objects and the relationships between them. For example, the objects might be employees in a human resources database, and you might want to facilitate queries such as "find all employees who directly or indirectly work for Scott." Neo4j is a popular graph database.

Compared to relational databases, the NoSQL options offer far greater scalability and cost-effectiveness for storage and analysis of unstructured data. The tradeoff is that they don't provide the rich queryability and robust data integrity capabilities of relational databases. NoSQL would work well for IIS log data, which involves high volume with no need for join queries. NoSQL would not work so well for banking transactions, which requires absolute data integrity and involves many relationships to other account-related data.

There is also a newer category of database platform called NewSQL that combines the scalability of a NoSQL database with the queryability and transactional integrity of a relational database. NewSQL databases are designed for distributed storage and query processing, which is often hard to implement in "OldSQL" databases. NuoDB is an example of a NewSQL database that can be used on Azure.

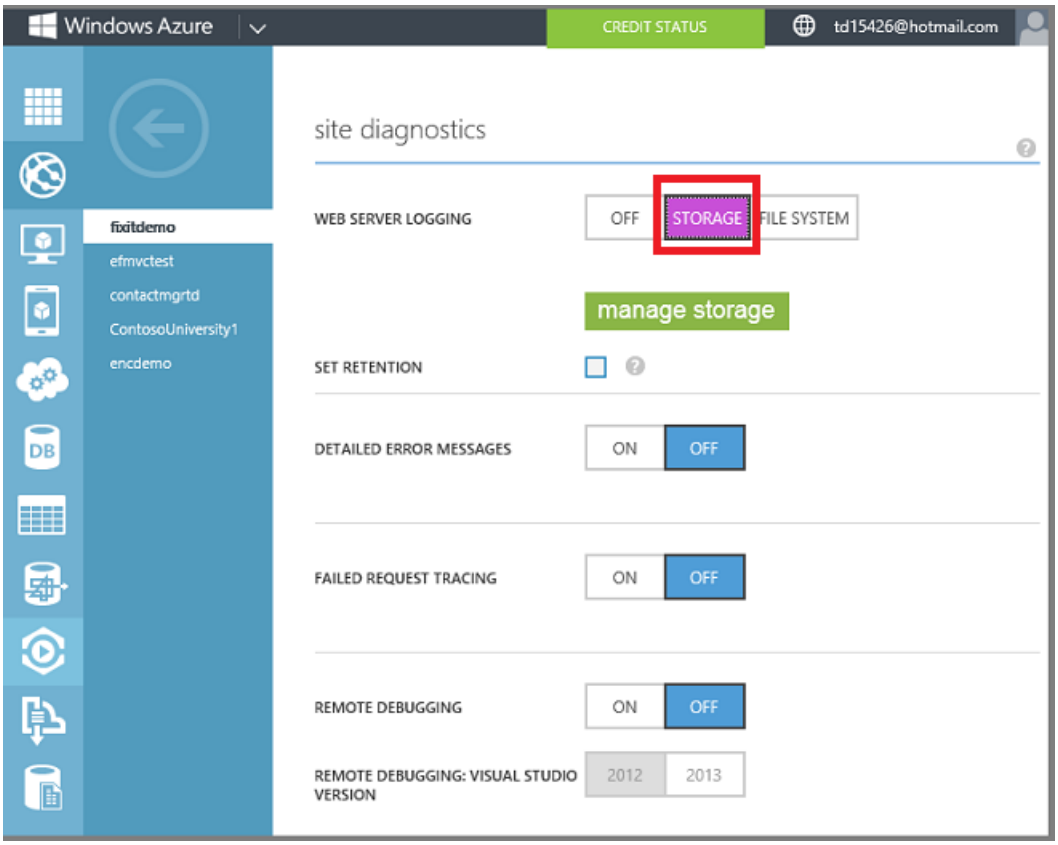
Hadoop and MapReduce

The high volumes of data that you can store in NoSQL databases may be difficult to analyze efficiently in a timely manner. To do that you can use a framework like Hadoop which implements MapReduce functionality. Essentially what a MapReduce process does is the following:

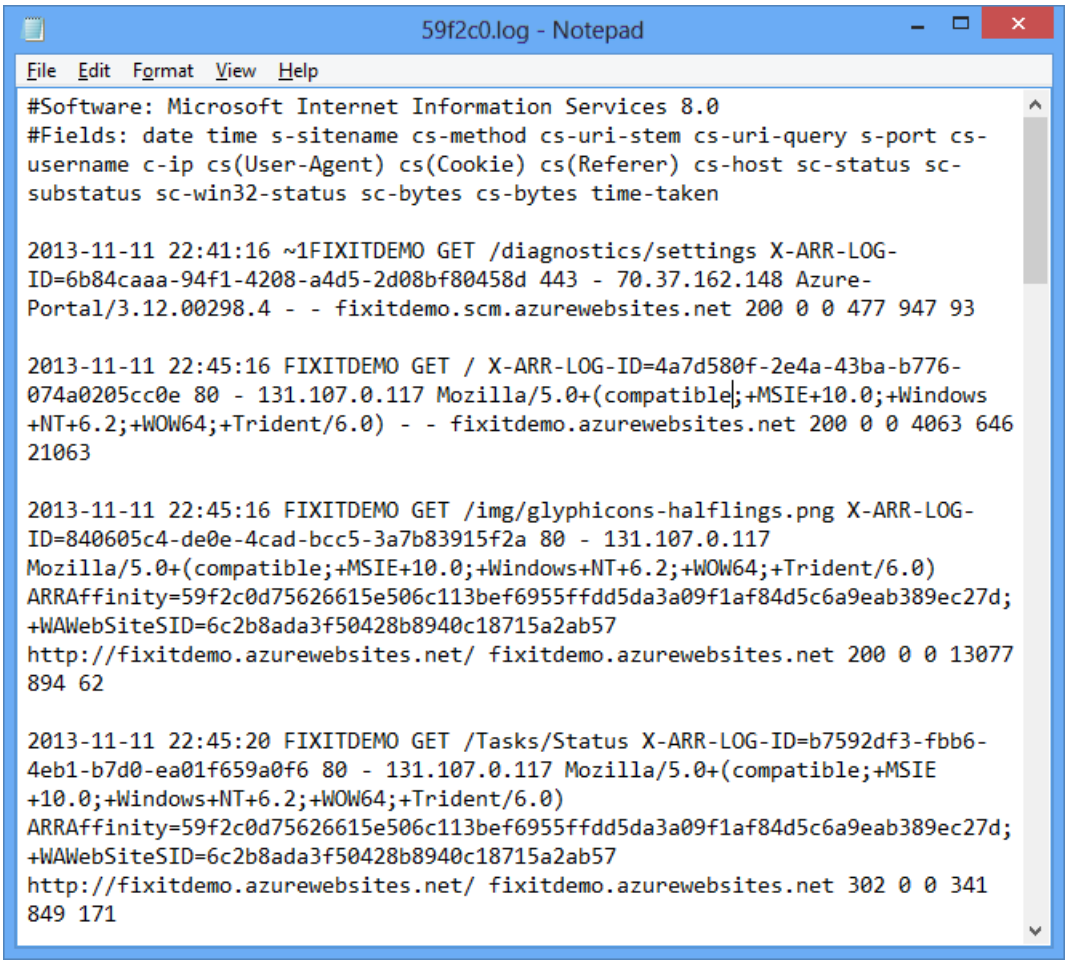
- Limit the size of the data that needs to be processed by selecting out of the data store only the data you actually need to analyze. For example, you want to know the makeup of your user base by birth year, so you select only birth years out of your user profile data store.
- Break down the data into parts and send them to different computers for processing. Computer A calculates the number of people with 1950-1959 dates, computer B does 1960-1969, etc. This group of computers is called a *Hadoop cluster*.
- Put the results of each part back together after the processing on the parts is done. You now have a relatively short list of how many people for each birth year and the task of calculating percentages in this overall list is manageable.

On Azure, HDInsight enables you to process, analyze, and gain new insights from big data using the power of Hadoop. For example, you could use it to analyze web server logs:

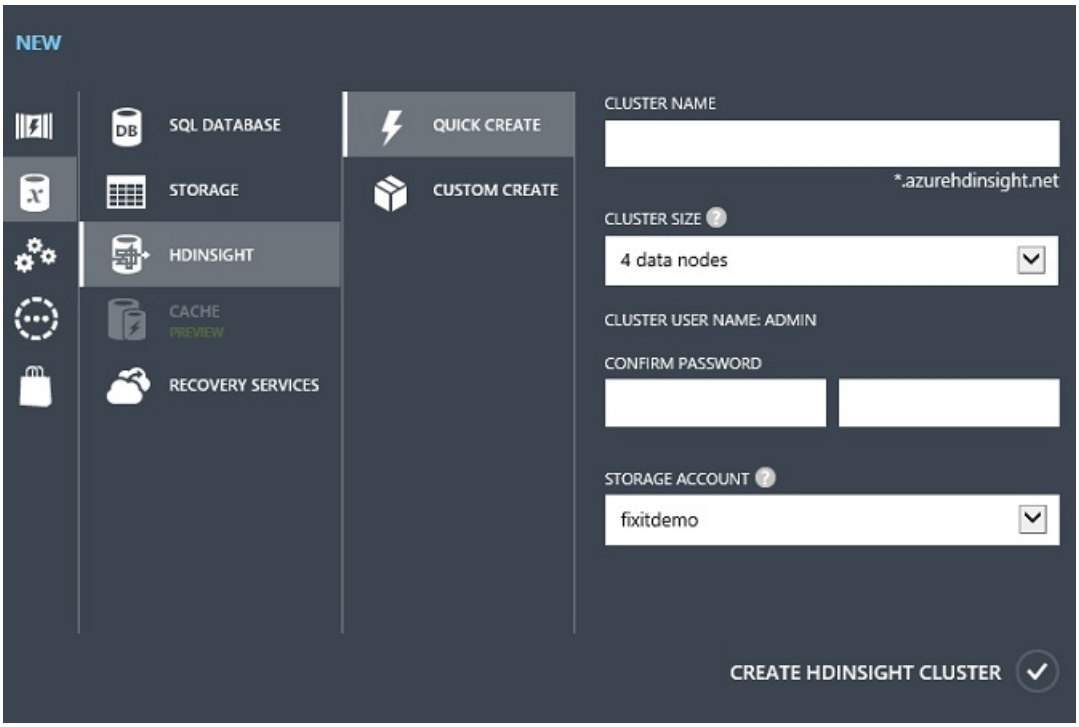
- Enable web server logging to your storage account. This sets up Azure to write logs to the Blob Service for every HTTP request to your application. The Blob Service is basically cloud file storage, and it integrates nicely with HDInsight.



- As the app gets traffic, web server IIS logs are written to Blob storage.



- In the portal, click **New** - **Data Services** - **HDInsight** - **Quick Create**, and specify an HDInsight cluster name, cluster size (number of HDInsight cluster data nodes), and a user name and password for the HDInsight cluster.



You can now set up MapReduce jobs to analyze your logs and get answers to questions such as:

- What times of day does my app get the most or least traffic?
- What countries is my traffic coming from?
- What is the average neighborhood income of the areas my traffic comes from. (There's a public dataset that gives you neighborhood income by IP address, and you can match that against IP address in the web server logs.)
- How does neighborhood income correlate to specific pages or products in the site?

You could then use the answers to questions like these to target ads based on the likelihood a customer would be interested in or likely to buy a particular product.

As explained in the Automate Everything chapter, most functions that you can do in the portal can be automated, and that includes setting up and executing HDInsight analysis jobs. A typical HDInsight script might contain the following steps:

- Provision an HDInsight cluster and link it to your storage account for Blob storage input.
- Upload the MapReduce job executables (.jar or .exe files) to the HDInsight cluster.
- Submit a MapReduce that stores the output data to Blob storage.
- Wait for the job to complete.
- Delete the HDInsight cluster.
- Access the output from Blob storage.

By running a script that does all this, you minimize the amount of time that the HDInsight cluster is provisioned, which minimizes your costs.

Platform as a Service (PaaS) versus Infrastructure as a Service (IaaS)

The data storage options listed earlier include both Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) solutions. PaaS means that we manage the hardware and software infrastructure and you just use the service. SQL Database is a PaaS feature of Azure. You ask for databases, and behind the scenes Azure sets up and configures the VMs and sets up the databases on them. You don't have direct access to the VMs and don't have to manage them.IaaS means that you set up, configure, and manage VMs that run in our data center infrastructure, and you put whatever you want on them. We provide a gallery of pre-configured VM images for common VM

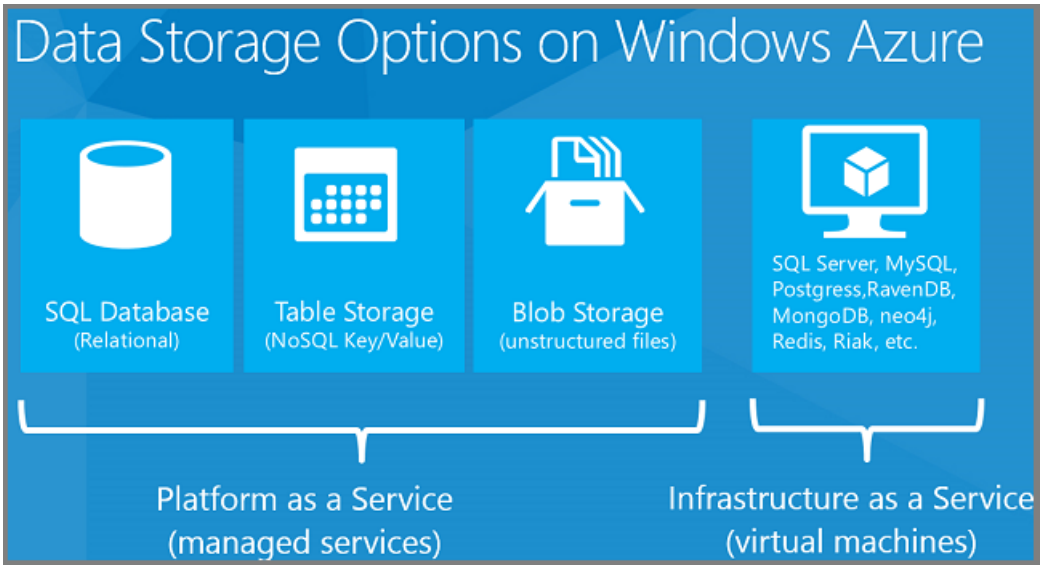
configurations. For example, you can install pre-configured VM images for Windows Server 2008, Windows Server 2012, BizTalk Server, Oracle WebLogic Server, Oracle Database, etc.

PaaS data solutions that Azure offers include:

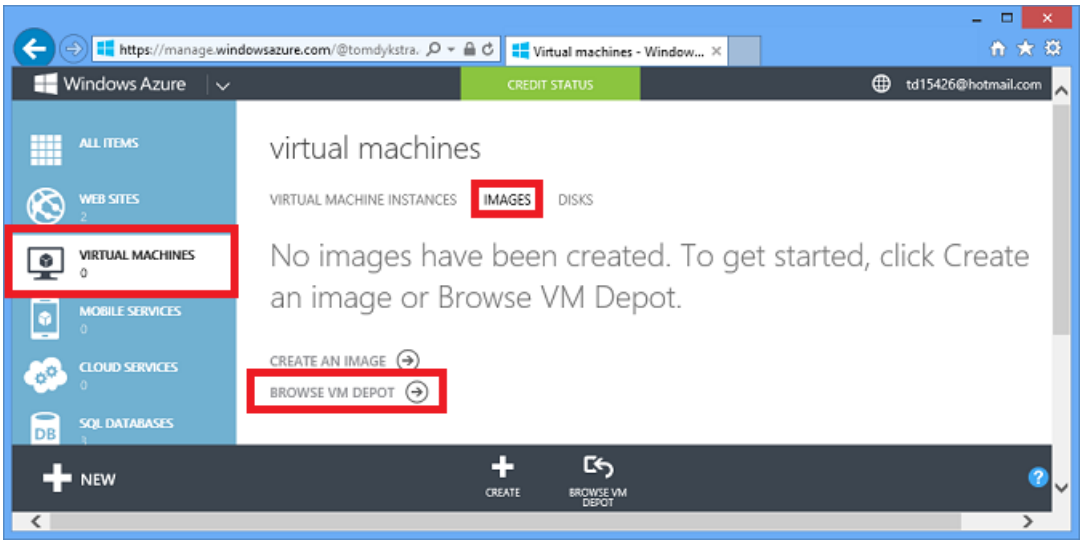
- Azure SQL Database (formerly known as SQL Azure). A cloud relational database based on SQL Server.
- Azure Table storage. A key/value NoSQL database.
- Azure Blob storage. File storage in the cloud.

For IaaS, you can run anything you can load onto a VM, for example:

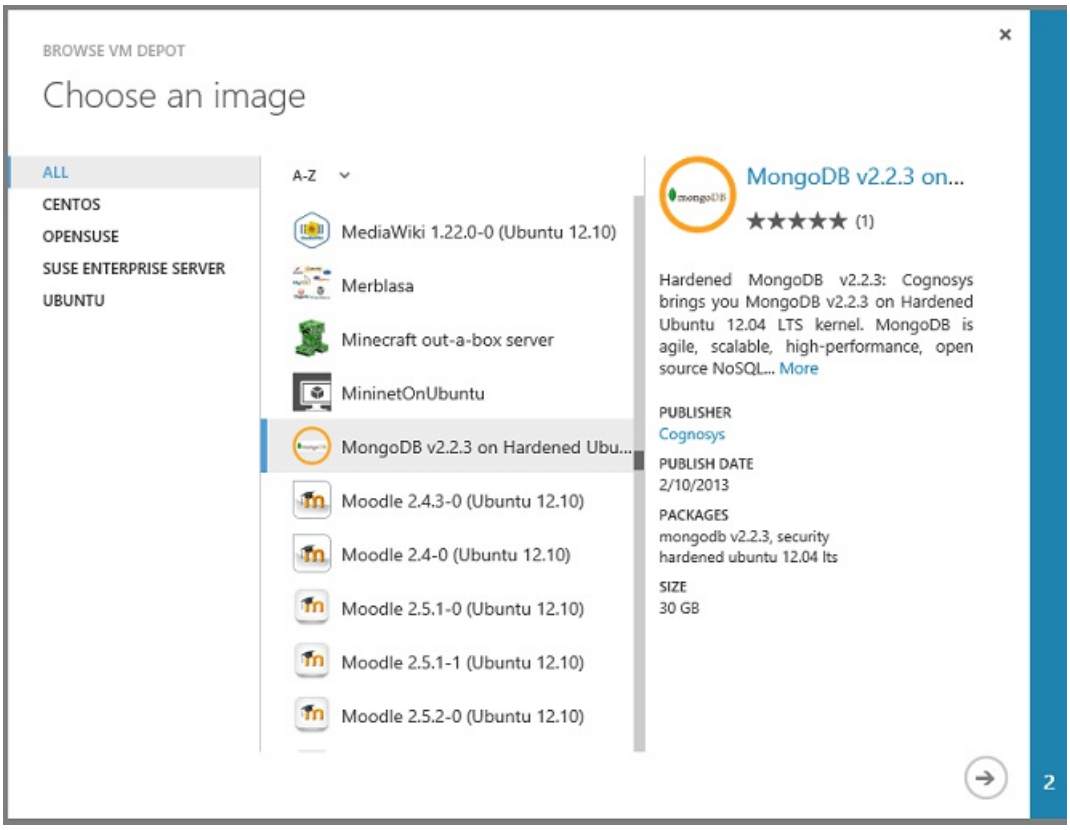
- Relational databases such as SQL Server, Oracle, MySQL, SQL Compact, SQLite, or Postgres.
- Key/value data stores such as Memcached, Redis, Cassandra, and Riak.
- Column data stores such as HBase.
- Document databases such as MongoDB, RavenDB, and CouchDB.
- Graph databases such as Neo4j.



The IaaS option gives you almost unlimited data storage options, and many of them are especially easy to use because you can create VMs using preconfigured images. For example, in the management portal go to **Virtual Machines**, click the **Images** tab, and click **Browse VM Depot**.



You then see a list of hundreds of preconfigured VM images, and you can create a VM from an image that has a database management system preinstalled, such as MongoDB, Neo4J, Redis, Cassandra, or CouchDB:



Azure makes IaaS data storage options as easy to use as possible, but the PaaS offerings have many advantages that make them more cost-effective and practical for many scenarios:

- You don't have to create VMs, you just use the portal or a script to set up a data store. If you want a 200 terabyte data store, you can just click a button or run a command, and in seconds it's ready for you to use.
- You don't have to manage or patch the VMs used by the service; Microsoft does that for you automatically.- You don't have to worry about setting up infrastructure for scaling or high availability; Microsoft handles all that for you.
- You don't have to buy licenses; license fees are included in the service fees.
- You only pay for what you use.

PaaS data storage options in Azure include offerings by third-party providers. For example, you can choose the MongoLab Add-On from the Azure Store to provision a MongoDB database as a service.

Choosing a data storage option

No one approach is right for all scenarios. If anyone says that this technology is the answer, the first thing to ask is "What is the question?", because different solutions are optimized for different things. There are definite advantages to the relational model; that's why it's been around for so long. But there are also down-sides to SQL that can be addressed with a NoSQL solution.

Often what we see work best is a compositional approach, where you use SQL and NoSQL in a single solution. Even when people say they're embracing NoSQL, if you drill into what they're doing you often find that they're using several different NoSQL frameworks: they're using CouchDB, and Redis, and Riak for different things. Even Facebook, which uses NoSQL extensively, uses different NoSQL frameworks for different parts of the service. The flexibility to mix and match data storage approaches is one of the things that's nice about the cloud, because it's easy to use multiple data solutions and integrate them in a single app.

Here are some questions to think about when you're choosing an approach:

DATA SEMANTIC	- WHAT IS THE CORE DATA STORAGE AND DATA ACCESS SEMANTIC (ARE YOU STORING RELATIONAL OR UNSTRUCTURED DATA)? UNSTRUCTURED DATA SUCH AS MEDIA FILES FITS BEST IN BLOB STORAGE; A COLLECTION OF RELATED DATA SUCH AS PRODUCTS, INVENTORIES, SUPPLIERS, CUSTOMER ORDERS, ETC., FITS BEST IN A RELATIONAL DATABASE.
Query support	- How easy is it to query the data? - What types of questions can be efficiently asked? Key/value data stores are very good at getting a single row given a key value but not so good for complex queries. For a user profile data store where you are always getting the data for one particular user, a key/value data store could work well; for a product catalog where you want to get different groupings based on various product attributes a relational database might work better. NoSQL databases can store large volumes of data efficiently, but you have to structure the database around how the app queries the data, and this makes ad hoc queries harder to do. With a relational database, you can build almost any kind of query.
Functional projection	- Can questions, aggregations, etc., be executed server-side? If I run SELECT COUNT(*) from a table in SQL, it will very efficiently do all the work on the server and return the number I'm looking for. If I want the same calculation from a NoSQL data store that doesn't support aggregation, this is an inefficient "unbounded query" and will probably time out. Even if the query succeeds I have to retrieve all of the data from the server to the client and count the rows on the client. - What languages or types of expressions can be used? With a relational database I can use SQL. With some NoSQL databases such as Azure Table storage, I'll be using OData, and all I can do is filter on primary key and get projections (select a subset of the available fields).
Ease of scalability	- How often and how much will the data need to scale? - Does the platform natively implement scale-out? - How easy is it to add/remove capacity (size and throughput)? Relational databases and tables aren't automatically partitioned to make them scalable, so they are difficult to scale beyond certain limitations. NoSQL data stores like Azure Table storage inherently partition everything, and there is almost no limit to adding partitions. You can readily scale Table Storage up to 200 terabytes, but the maximum database size for Azure SQL Database is 500 gigabytes. You can scale relational data by partitioning it into multiple databases, but setting up an application to support that model involves a lot of programming work.
Instrumentation and Manageability	- How easy is the platform to instrument, monitor, and manage? You will need to keep informed about the health and performance of your data store, so you need to know up front what metrics a platform gives you for free, and what you have to develop yourself.
Operations	- How easy is the platform to deploy and run on Azure? PaaS? IaaS? Linux? Table Storage and SQL Database are easy to set up on Azure. Platforms that aren't built-in Azure PaaS solutions require more effort.

DATA SEMANTIC	- WHAT IS THE CORE DATA STORAGE AND DATA ACCESS SEMANTIC (ARE YOU STORING RELATIONAL OR UNSTRUCTURED DATA)? UNSTRUCTURED DATA SUCH AS MEDIA FILES FITS BEST IN BLOB STORAGE; A COLLECTION OF RELATED DATA SUCH AS PRODUCTS, INVENTORIES, SUPPLIERS, CUSTOMER ORDERS, ETC., FITS BEST IN A RELATIONAL DATABASE.
API Support	- Is an API available that makes it easy to work with the platform? For the Azure Table Service there's an SDK with a .NET API that supports the .NET 4.5 asynchronous programming model. If you're writing a .NET app, it'll be much easier to write and test code for the Azure Table Service compared to another key/value column data store platform that has no API or a less comprehensive one.
Transactional integrity and data consistency	- Is it critical that the platform support transactions in order to guarantee data consistency? For keeping track of bulk emails sent, performance and low data storage cost might be more important than automatic support for transactions or referential integrity in the data platform, making the Azure Table Service a good choice. For tracking bank account balances or purchase orders a relational database platform that provides strong transactional guarantees would be a better choice.
Business continuity	- How easy are backup, restore, and disaster recovery? Sooner or later production data will get corrupted and you'll need an undo function. Relational databases often have more fine-grained restore capabilities, such as the ability to restore to a point in time. Understanding what restore features are available in each platform you're considering is an important factor to consider.
Cost	- If more than one platform can support your data workload, how do they compare in cost? For example, if you use ASP.NET Identity, you can store user profile data in Azure Table Service or Azure SQL Database. If you don't need the rich querying facilities of SQL Database, you might choose Azure Tables in part because it costs much less for a given amount of storage.

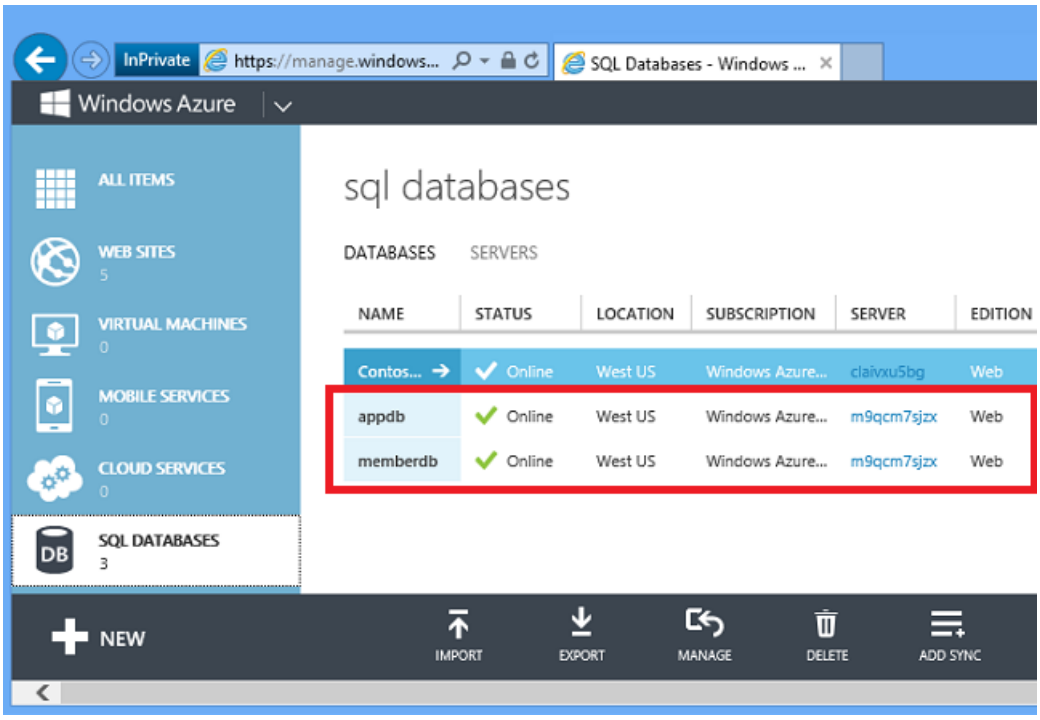
What we generally recommend is know the answer to the questions in each of these categories before you choose your data storage solutions.

In addition, your workload might have specific requirements that some platforms can support better than others. For example:

- Does your application require audit capabilities?
- What are your data longevity requirements -- do you require automated archival or purging capabilities?
- Do you have specialized security needs? For example, the data includes PII (personally identifiable information) but you have to be able to make sure that PII is excluded from query results.
- If you have some data that can't be stored in the cloud for regulatory or technological reasons, you might need a cloud data storage platform that facilitates integrating with your on-premises storage.

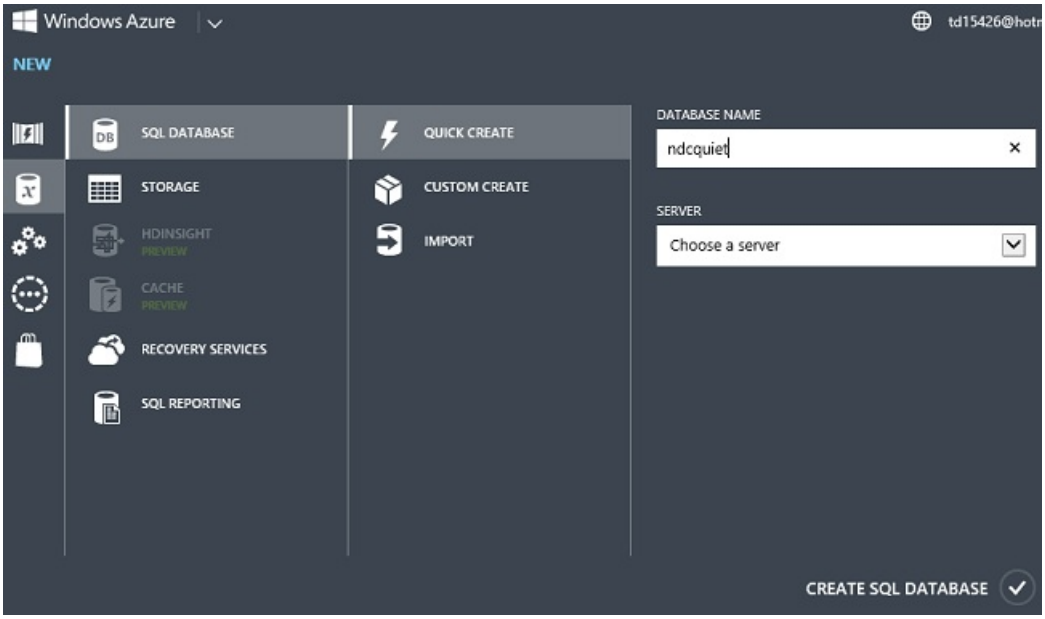
Demo – using SQL Database in Azure

The Fix It app uses a relational database to store tasks. The environment creation Windows PowerShell script shown in the Automate Everything chapter creates two SQL Database instances. You can see these in the portal by clicking the **SQL Databases** tab.

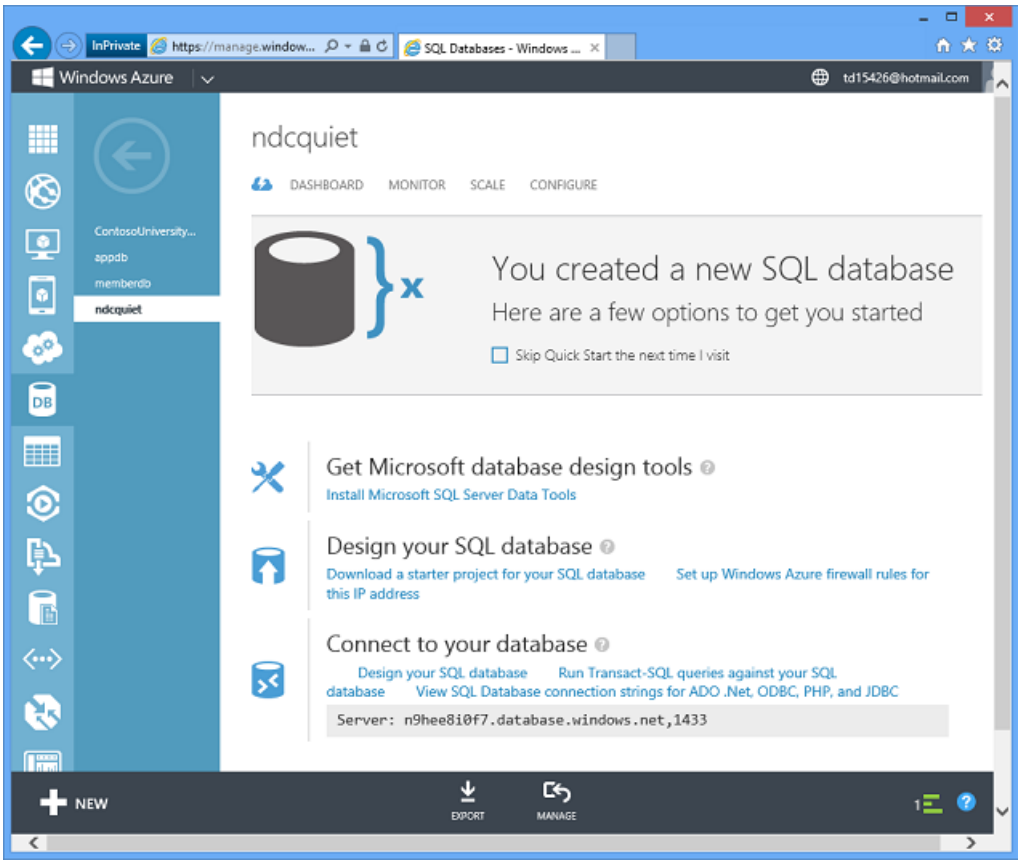


It's also easy to create databases by using the portal.

Click **New -- Data Services -- SQL Database -- Quick Create**, enter a database name, choose a server you already have in your account or create a new one, and click **Create SQL Database**.



Wait several seconds, and you have a database in Azure ready for you to use.

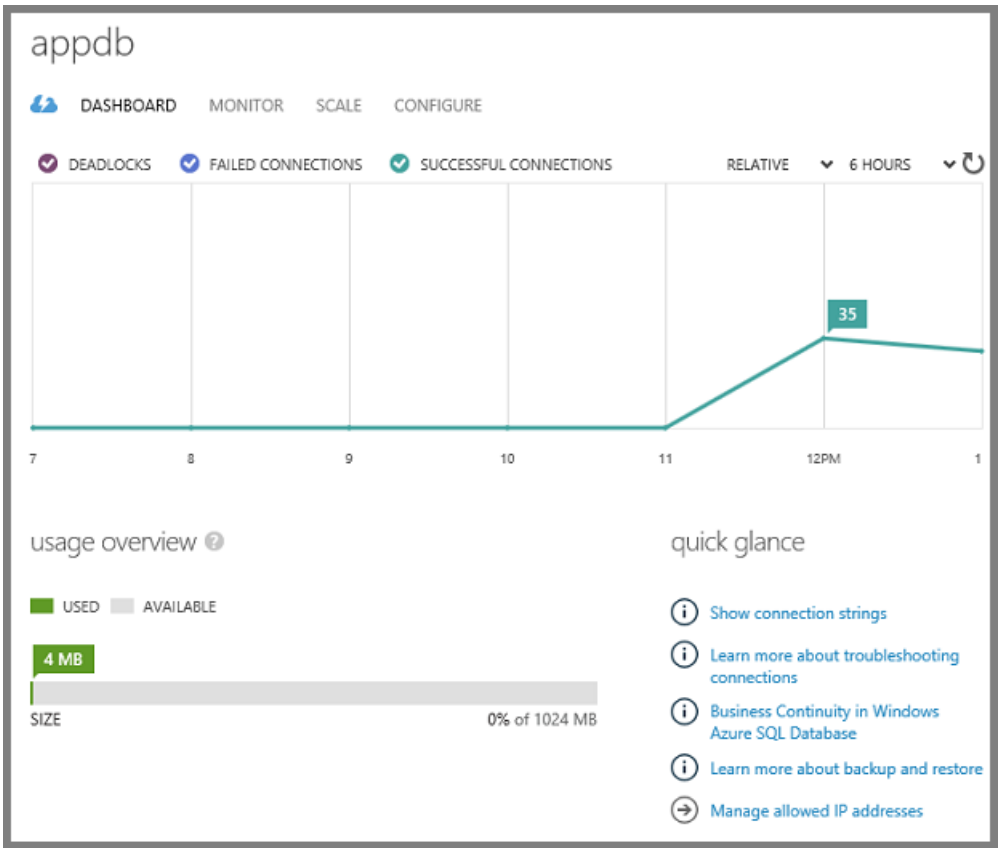


So Azure does in a few seconds what it may take you a day or a week or longer to accomplish in the on-premises environment. And since you can just as easily create databases automatically in a script or by using a management API, you can dynamically scale out by spreading your data across multiple databases, so long as your application has been programmed for that.

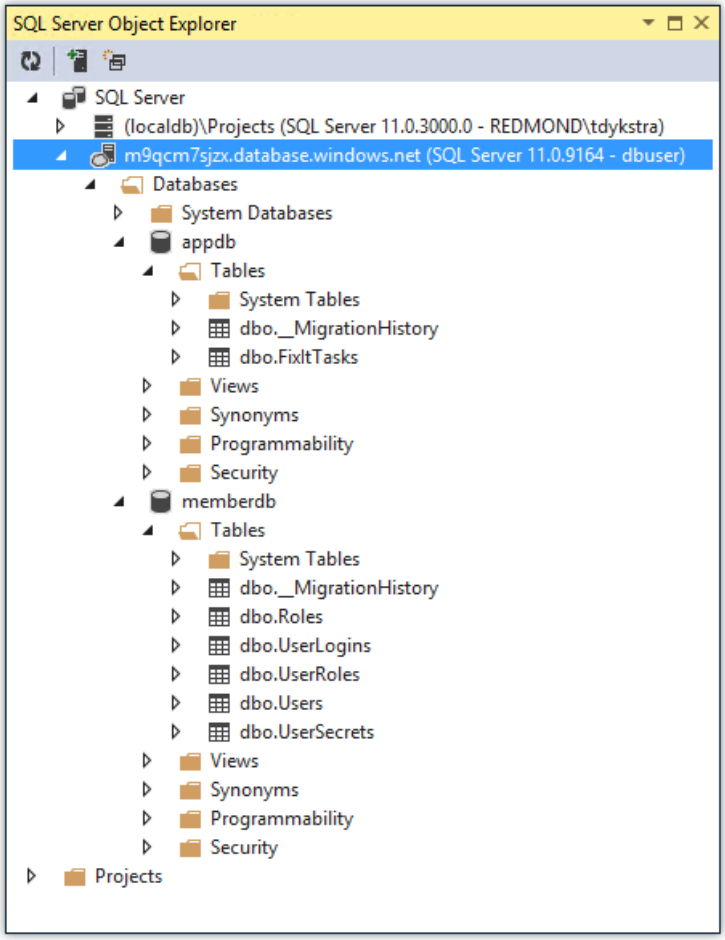
This is an example of our Platform-as-a-Service model. You don't have to manage the servers, we do it. You don't have to worry about backups, we do it. It's running in high availability – the data in the database is replicated across three servers automatically. If a machine dies, we automatically fail over and you lose no data. The server is patched regularly, you don't need to worry about that.

Click a button and you get the exact connection string you need and can immediately start using the new database.

The Dashboard shows you connection history and amount of storage used.

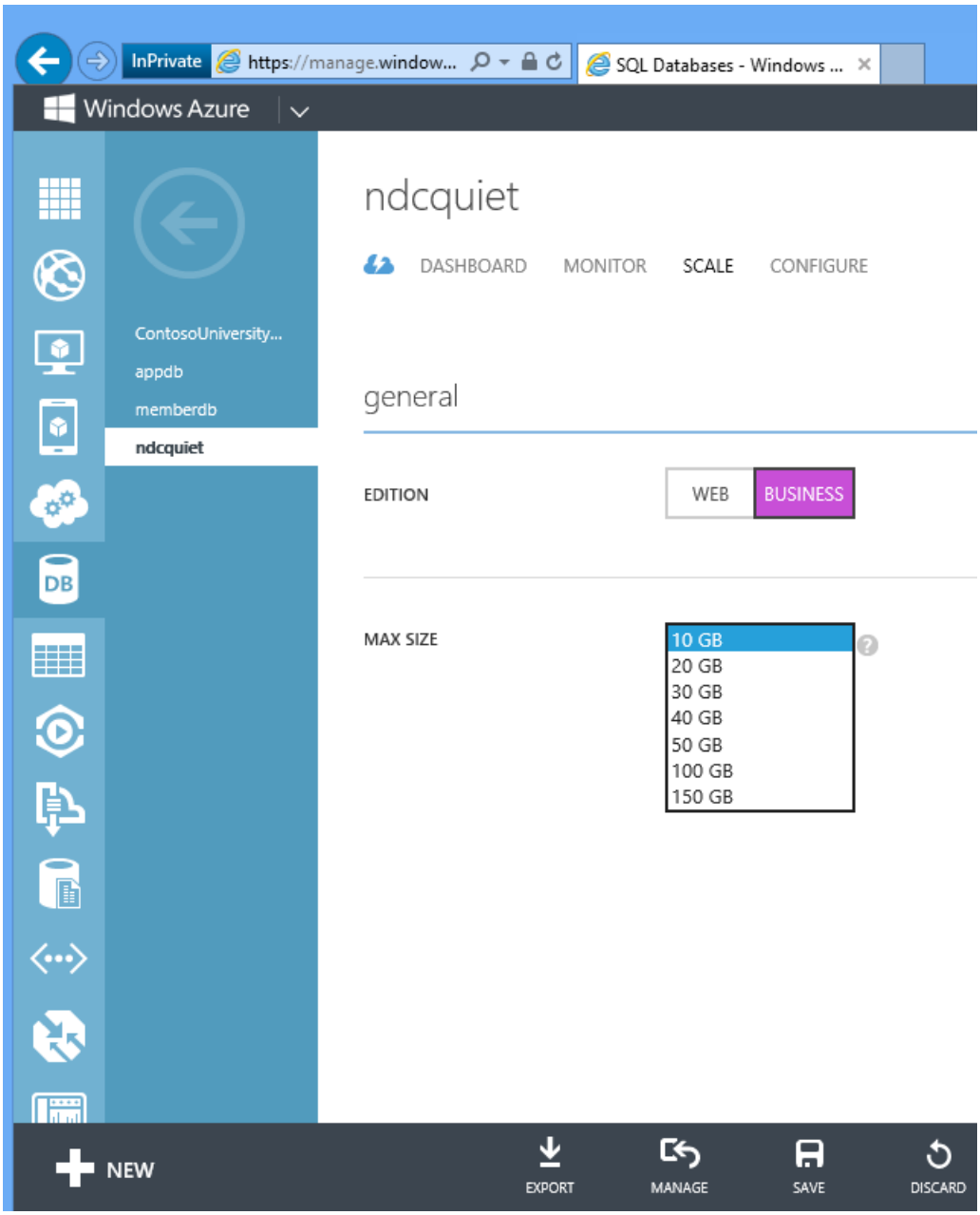


You can manage databases in the portal or by using SQL Server tools you're already familiar with, including SQL Server Management Studio (SSMS) and the Visual Studio tools SQL Server Object Explorer (SSOX) and Server Explorer.



Another nice thing is the pricing model. You can start development with a free 20 MB database, and a production database starts at about $5 per month. You pay only for the amount of data you actually store in the database, not the maximum capacity. You don't have to buy a license.

SQL Database is easy to scale. For the Fix It app, the database we create in our automation script is capped at 1 gig. If you want to scale it up to 150 gig, you can just go into the portal and change that setting, or execute a REST API command, and in seconds you have a 150 gig database that you can deploy data into.



That's the power of the cloud to stand up infrastructure quickly and easily and start using it immediately.

The Fix It app uses two SQL databases, one for membership (authentication and authorization) and one for data, and this is all you have to do to provision it and scale it. You saw earlier how to provision the databases through Windows PowerShell scripts, and now you've also seen how easy it is to do in the portal.

Entity Framework versus direct database access using ADO.NET

The Fix It app accesses these databases by using the Entity Framework, Microsoft's recommended ORM (object-relational mapper) for .NET applications. An ORM is a great tool that facilitates developer productivity, but productivity comes at the expense of degraded performance in some scenarios. In a real-world cloud app you won't be making a choice between using EF or using ADO.NET directly -- you'll use both. Most of the time when you're writing code that works with the database, getting maximum performance is not critical and you can take advantage of the simplified coding and testing that you get with the Entity Framework. In situations where the EF overhead would cause unacceptable performance, you can write and execute your own queries using ADO.NET, ideally by calling stored procedures.

Whatever method you use to access the database, you want to minimize "chattiness" as much as possible. In other words, if you can get all the data you need in one larger query result set rather than dozens or hundreds of smaller ones, that's usually preferable. For example, if you need to list students and the courses they're enrolled in,

it's usually better to get all of the data in one join query rather than getting the students in one query and executing separate queries for each student's courses.

SQL databases and the Entity Framework in the Fix It app

In the Fix It app the `FixItContext` class, which derives from the Entity Framework `DbContext` class, identifies the database and specifies the tables in the database. The context specifies an entity set (table) for tasks, and the code passes in to the context the connection string name. That name refers to a connection string that is defined in the Web.config file.

```
public class MyFixItContext : DbContext
{
    public MyFixItContext()
        : base("name=appdb")
    {
    }

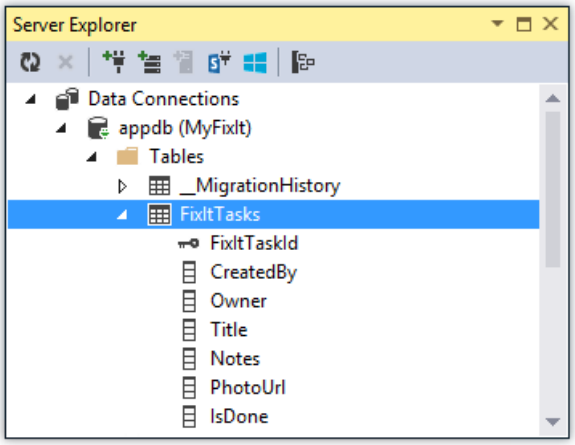
    public DbSet<MyFixIt.Persistence.FixItTask> FixItTasks { get; set; }
}
```

The connection string in the *Web.config* file is named appdb (here pointing to the local development database):

```
<connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=aspnet-
MyFixIt-20130604091232_4;Integrated Security=True" providerName="System.Data.SqlClient" />
    <add name="appdb" connectionString="Data Source=(localdb)\v11.0; Initial Catalog=MyFixItContext-
20130604091609_11;Integrated Security=True; MultipleActiveResultSets=True"
providerName="System.Data.SqlClient" />
  </connectionStrings>
```

The Entity Framework creates a *FixItTasks* table based on the properties included in the `FixItTask` entity class. This is a simple POCO (Plain Old CLR Object) class, which means it doesn't inherit from or have any dependencies on the Entity Framework. But Entity Framework knows how to create a table based on it and execute CRUD (create-read-update-delete) operations with it.

```
public class FixItTask
{
    public int FixItTaskId  { get; set; }
    public string CreatedBy { get; set; }
    [Required]
    public string Owner     { get; set; }
    [Required]
    public string Title     { get; set; }
    public string Notes     { get; set; }
    public string PhotoUrl  { get; set; }
    public bool IsDone      { get; set; }
}
```



The Fix It app includes a repository interface that it uses for CRUD operations working with the data store.

```
public interface IFixItTaskRepository
{
    Task<List<FixItTask>> FindOpenTasksByOwnerAsync(string userName);
    Task<List<FixItTask>> FindTasksByCreatorAsync(string userName);

    Task<MyFixIt.Persistence.FixItTask> FindTaskByIdAsync(int id);

    Task CreateAsync(FixItTask taskToAdd);
    Task UpdateAsync(FixItTask taskToSave);
    Task DeleteAsync(int id);
}
```

Notice that the repository methods are all async, so all data access can be done in a completely asynchronous way.

The repository implementation calls Entity Framework async methods to work with the data, including LINQ queries as well as for insert, update, and delete operations. Here's an example of the code for looking up a Fix It task.

```
public async Task<FixItTask> FindTaskByIdAsync(int id)
{
    FixItTask fixItTask = null;
    Stopwatch timespan = Stopwatch.StartNew();

    try
    {
        fixItTask = await db.FixItTasks.FindAsync(id);

        timespan.Stop();
        log.TraceApi("SQL Database", "FixItTaskRepository.FindTaskByIdAsync", timespan.Elapsed, "id={0}",
id);
    }
    catch(Exception e)
    {
        log.Error(e, "Error in FixItTaskRepository.FindTaskByIdAsynx(id={0})", id);
    }

    return fixItTask;
}
```

You'll notice there's also some timing and error logging code here, we'll look at that later in the Monitoring and Telemetry chapter.

Choosing SQL Database (PaaS) versus SQL Server in a VM (IaaS) in

Azure

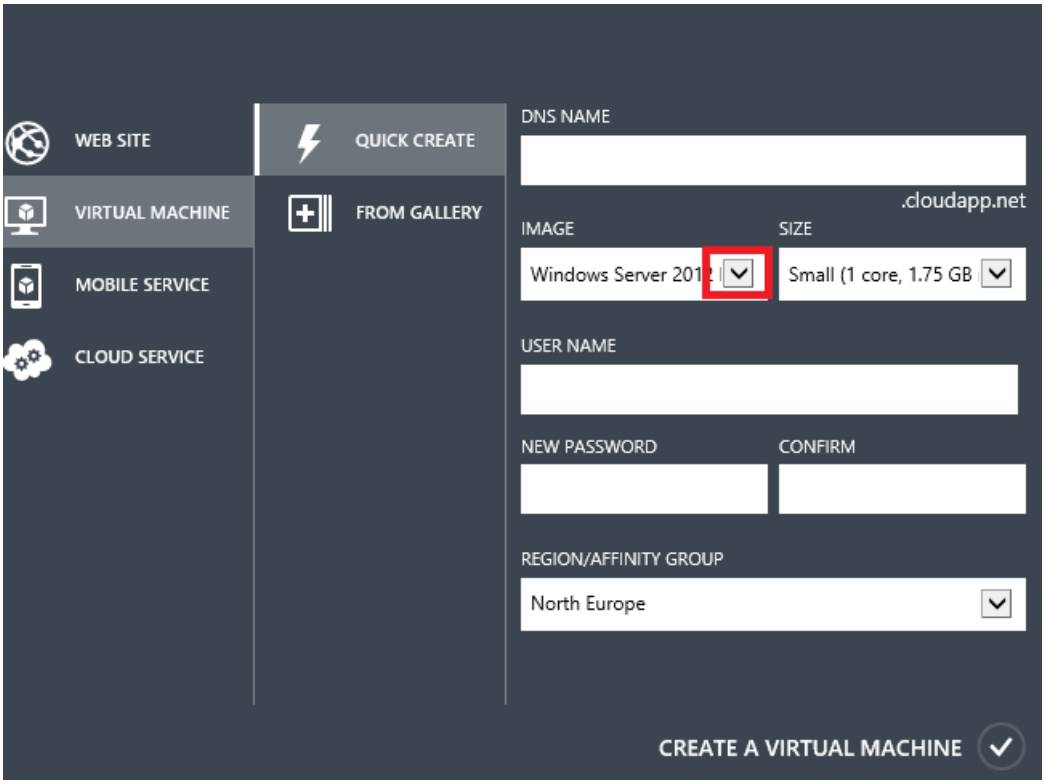
A nice thing about SQL Server and Azure SQL Database is that the core programming model for both of them is identical. You can use most of the same skills in both environments. You can even use a SQL Server database in development and a SQL Database instance in the cloud, which is how the Fix It app is set up.

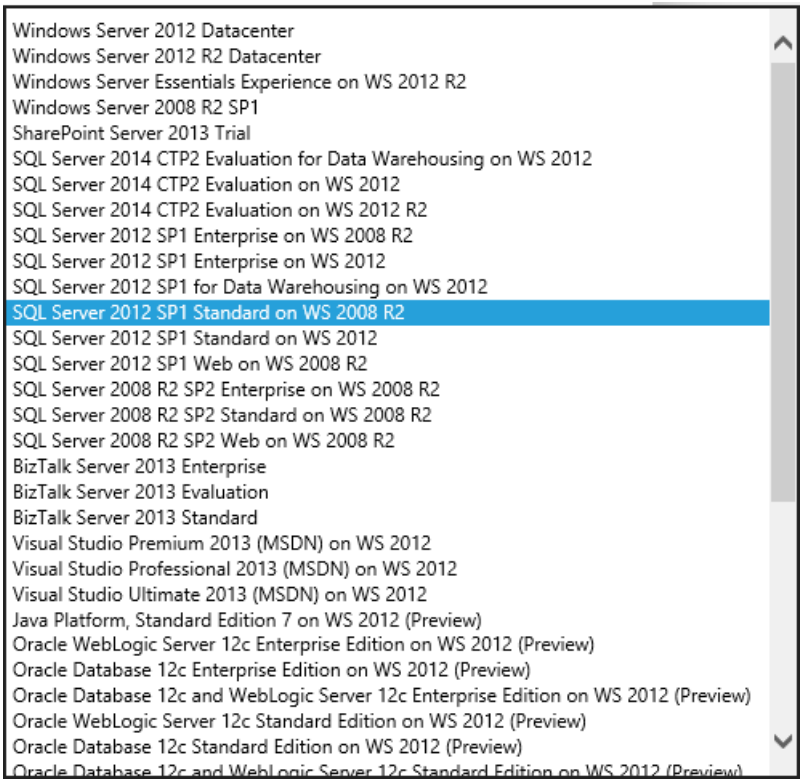
As an alternative, you can run the same SQL Server in the cloud that you run on-premises by installing it on IaaS VMs. For some legacy applications, running SQL Server in a VM might be a better solution. Because a SQL Server database runs on a dedicated VM, it has more resources available to it than a SQL Database database that runs on a shared server. That means a SQL Server database can be larger and still perform well. In general, the smaller the database size and table size, the better the use case works for SQL Database (PaaS).

Here are some guidelines on how to choose between the two models.

AZURE SQL DATABASE (PAAS)	SQL SERVER IN A VIRTUAL MACHINE (IAAS)
Pros - You don't have to create or manage VMs, update or patch OS or SQL; Azure does that for you. - Built-in High Availability, with a database-level SLA. - Low total cost of ownership (TCO) because you pay only for what you use (no license required). - Good for handling large numbers of smaller databases (<=500 GB each). - Easy to dynamically create new databases to enable scale-out.	***Pros*** - Feature-compatible with on-premises SQL Server. - Can implement SQL Server High Availability via AlwaysOn in 2+ VMs, with VM-level SLA. - You have complete control over how SQL is managed. - Can re-use SQL licenses you already own, or pay by the hour for one. - Good for handling fewer but larger (1 TB+) databases.
Cons - Some feature gaps compared to on-premises SQL Server (lack of CLR integration, TDE, compression support, SQL Server Reporting Services, etc.) - Database size limit of 500GB.	***Cons*** - Updates/patches (OS and SQL) are your responsibility - Creation and management of DBs are your responsibility - Disk IOPS (input/output operations per second) limited to about 8000 (via 16 data drives).

If you want to use SQL Server in a VM, you can use your own SQL Server license, or you can pay for one by the hour. For example, in the portal or via the REST API you can create a new VM using a SQL Server image.





When you create a VM with a SQL Server image, we pro-rate the SQL Server license cost by the hour based on your usage of the VM. If you have a project that's only going to run for a couple of months, it's cheaper to pay by the hour. If you think your project is going to last for years, it's cheaper to buy the license the way you normally do.

Summary

Cloud computing makes it practical to mix and match data storage approaches to best fit the needs of your application. If you're building a new application, think carefully about the questions listed here in order to pick approaches that will continue to work well when your application grows. The next chapter will explain some partitioning strategies that you can use to combine multiple data storage approaches.

Resources

For more information, see the following resources.

Choosing a database platform:

- Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence. E-book by Microsoft Patterns and Practices that goes in depth into the different kinds of data stores available for cloud applications.
- Microsoft Patterns and Practices - Azure Guidance. See Data Consistency Primer, Data Replication and Synchronization Guidance, Index Table pattern, Materialized View pattern.
- BASE: An Acid Alternative. Article about tradeoffs between data consistency and scalability.
- Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement. Book by Eric Redmond and Jim R. Wilson. Highly recommended for introducing yourself to the range of data storage platforms available today.

Choosing between SQL Server and SQL Database:

- Premium Preview for SQL Database Guidance. An introduction to SQL Database Premium, and guidance on when to choose it over the SQL Database Web and Business editions.
- Guidelines and Limitations (Azure SQL Database). Portal page that links to documentation about limitations of SQL Database, including one that focuses on SQL Server features that SQL Database doesn't support.

- SQL Server in Azure Virtual Machines. Portal page that links to documentation about running SQL Server in Azure.
- Scott Guthrie explains SQL Databases in Azure. 6-minute video introduction to SQL Database by Scott Guthrie.
- Application Patterns and Development Strategies for SQL Server in Azure Virtual Machines.

Using Entity Framework and SQL Database in an ASP.NET Web app

- Getting Started with EF 6 using MVC 5. Nine-part tutorial series that walks you through building an MVC app that uses EF and deploys the database to Azure and SQL Database.
- ASP.NET Web Deployment using Visual Studio. Twelve-part tutorial series that goes into more depth about how to deploy a database by using EF Code First.
- Deploy a Secure ASP.NET MVC 5 app with Membership, OAuth, and SQL Database to an Azure Web Site. Step-by-step tutorial that walks you through creating a web app that uses authentication, stores application tables in the membership database, modifies the database schema, and deploys the app to Azure.
- ASP.NET Data Access Content Map. Links to resources for working with EF and SQL Database.

Using MongoDB on Azure:

- MongoLab - MongoDB on Azure. Portal page for documentation about running MongoDB on Azure.
- Create an Azure web site that connects to MongoDB running on a virtual machine in Azure. Step-by-step tutorial that shows how to use a MongoDB database in an ASP.NET web application.

HDInsight (Hadoop on Azure):

- HDInsight. Portal to HDInsight documentation on the Azure website.
- Hadoop and HDInsight: Big Data in Azure. MSDN Magazine article by Bruno Terkaly and Ricardo Villalobos, introducing Hadoop on Azure.
- Microsoft Patterns and Practices - Azure Guidance. See MapReduce pattern.

PREVIOUS NEXT

Data Partitioning Strategies (Building Real-World Cloud Apps with Azure)

2/20/2020 • 7 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the series, see the first chapter.

Earlier we saw how easy it is to scale the web tier of a cloud application, by adding and removing web servers. But if they're all hitting the same data store, your application's bottleneck moves from the front-end to the back-end, and the data tier is the hardest to scale. In this chapter we look at how you can make your data tier scalable by partitioning data into multiple relational databases, or by combining relational database storage with other data storage options.

Setting up a partitioning scheme is best done up front for the same reason mentioned earlier: it's very difficult to change your data storage strategy after an app is in production. If you think hard up front about different approaches, you can avoid having a "Twitter moment" when your app crashes or goes down for a long time while you reorganize your app's data and data access code.

The three Vs of data storage

In order to determine whether you need a partitioning strategy and what it should be, consider three questions about your data:

- Volume – How much data will you ultimately store? A couple gigabytes? A couple hundred gigabytes? Terabytes? Petabytes?
- Velocity – What is the rate at which your data will grow? Is it an internal app that isn't generating a lot of data? An external app that customers will be uploading images and videos into?
- Variety – What type of data will you store? Relational, images, key-value pairs, social graphs?

If you think you're going to have a lot of volume, velocity, or variety, you have to carefully consider what kind of partitioning scheme will best enable your app to scale efficiently and effectively as it grows, and to ensure you don't run into any bottlenecks.

There are basically three approaches to partitioning:

- Vertical partitioning
- Horizontal partitioning
- Hybrid partitioning

Vertical partitioning

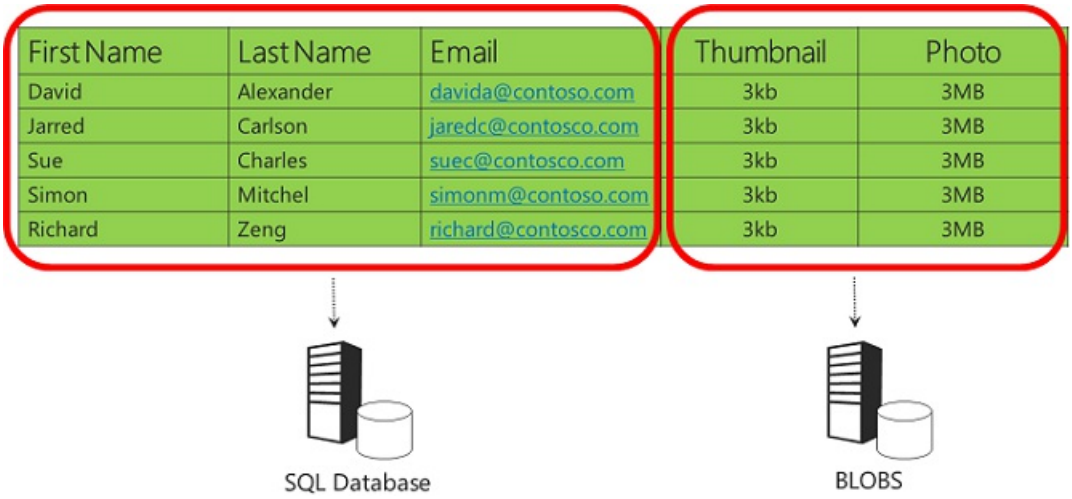
Vertical portioning is like splitting up a table by columns: one set of columns goes into one data store, and another set of columns goes into a different data store.

For example, suppose my app stores data about people, including images:

First Name	Last Name	Email	Thumbnail	Photo
David	Alexander	davida@contoso.com	3kb	3MB
Jarred	Carlson	jaredc@contosco.com	3kb	3MB
Sue	Charles	suec@contosco.com	3kb	3MB
Simon	Mitchel	simonm@contoso.com	3kb	3MB
Richard	Zeng	richard@contosco.com	3kb	3MB

When you represent this data as a table and look at the different varieties of data, you can see that the three columns on the left have string data that can be efficiently stored by a relational database, while the two columns on the right are essentially byte arrays that come from image files. It's possible to store image file data in a relational database, and a lot of people do that because they don't want to save the data to the file system. They might not have a file system capable of storing the required volumes of data or they might not want to manage a separate back-up and restore system. This approach works well for on-premises databases and for small amounts of data in cloud databases. In the on-premises environment, it might be easier to just let the DBA take care of everything.

But in a cloud database, storage is relatively expensive, and a high volume of images could make the size of the database grow beyond the limits at which it can operate efficiently. You can address these problems by partitioning the data vertically, which means you choose the most appropriate data store for each column in your table of data. What might work best for this example is to put the string data in a relational database and the images in Blob storage.



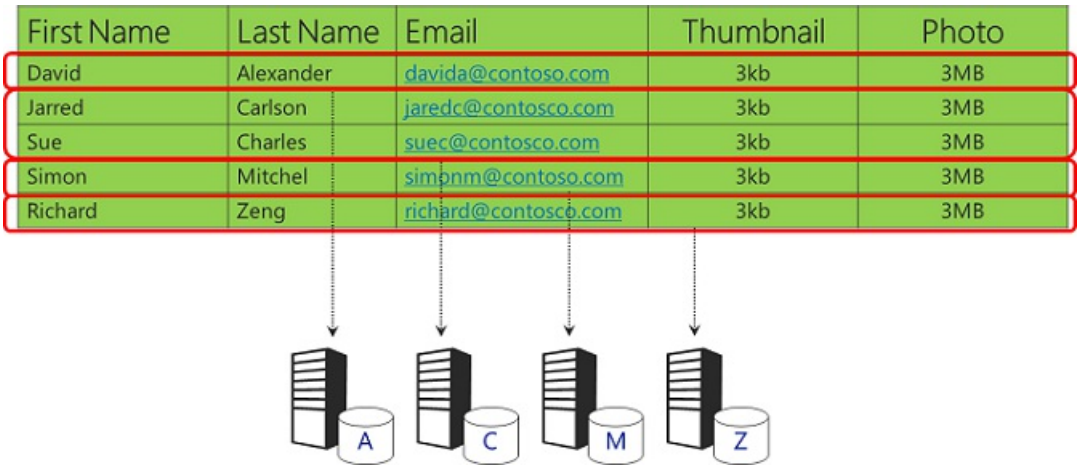
Storing images in Blob storage instead of a database is more practical in the cloud than in an on-premises environment because you don't have to worry about setting up file servers or managing back-up and restore of data stored outside of the relational database: all that is handled for you automatically by the Blob storage service.

This is the partitioning approach we implemented in the Fix It app, and we'll look at the code for that in the Blob storage chapter. Without this partitioning scheme, and assuming an average image size of 3 megabytes, the Fix It app would only be able to store about 40,000 tasks before hitting the maximum database size of 150 gigabytes. After removing the images, the database can store 10 times as many tasks; you can go much longer before you have to think about implementing a horizontal partitioning scheme. And as the app scales, your expenses grow more slowly because the bulk of your storage needs are going into very inexpensive Blob storage.

Horizontal partitioning (sharding)

Horizontal portioning is like splitting up a table by rows: one set of rows goes into one data store, and another set of rows goes into a different data store.

Given the same set of data, another option would be to store different ranges of customer names in different databases.

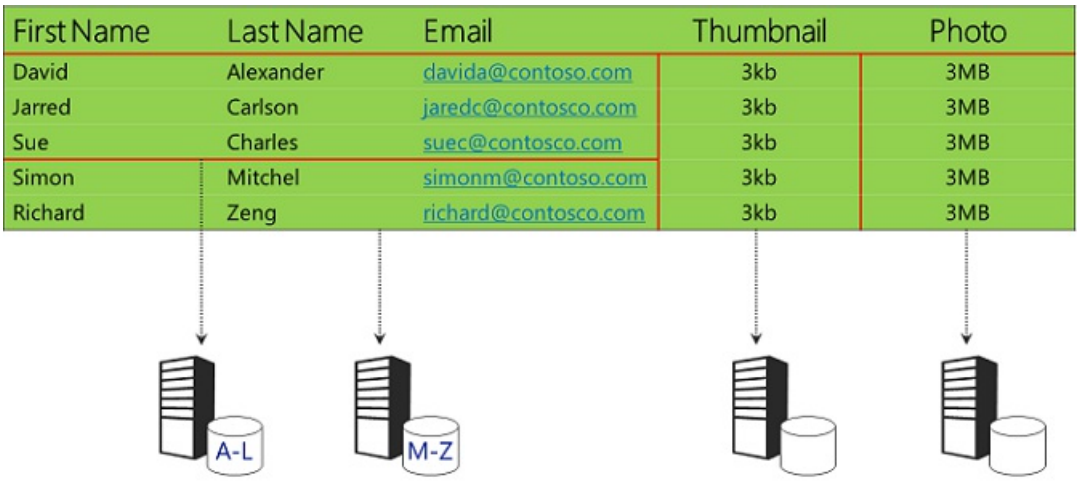


You want to be very careful about your sharding scheme to make sure that data is evenly distributed in order to avoid hot spots. This simple example using the first letter of the last name doesn't meet that requirement, because a lot of people have last names that start with certain common letters. You'd hit table size limitations earlier than you might expect because some databases would get very large while most would remain small.

A down side of horizontal partitioning is that it might be hard to do queries across all of the data. In this example, a query would have to draw from up to 26 different databases to get all of the data stored by the app.

Hybrid partitioning

You can combine vertical and horizontal partitioning. For example, in the example data you could store the images in Blob storage and horizontally partition the string data.



Partitioning a production application

Conceptually it's easy to see how a partitioning scheme would work, but any partitioning scheme does increase code complexity and introduces many new complications that you have to deal with. If you're moving images to blob storage, what do you do when the storage service is down? How do you handle blob security? What happens if the database and blob storage get out of sync? If you're sharding, how will you handle querying across all of the databases?

The complications are manageable so long as you're planning for them before you go to production. Many people who didn't do that wish they had later. On average our Customer Advisory Team (CAT) team gets panicked phone calls about once a month from customers whose apps are taking off in a really big way, and they didn't do this planning. And they say something like: "Help! I put everything in a single data store, and in 45 days I'm going to run out of space on it!" And if you have a lot of business logic built into how you access your data store and you have customers who are using your app, there's no good time to go down for a day while you migrate. We end up going through herculean efforts to help the customer partition their data on the fly with no down time. It's very

exciting and very scary, and not something you want to be involved in if you can avoid it! Thinking about this up front and integrating it into your app will make your life a lot easier if the app grows later.

Summary

An effective partitioning scheme can enable your cloud app to scale to petabytes of data in the cloud without bottlenecks. And you don't have to pay up front for massive machines or extensive infrastructure as you might if you were running the app in an on-premises data center. In the cloud you can incrementally add capacity as you need it, and you're only paying for as much as you're using when you use it.

In the next chapter we'll see how the Fix It app implements vertical partitioning by storing images in Blob storage.

Resources

For more information about partitioning strategies, see the following resources.

Documentation:

- Best Practices for the Design of Large-Scale Services on Windows Azure Cloud Services. White paper by Mark Simms and Michael Thomassy.
- Microsoft Patterns and Practices - Cloud Design Patterns. See Data Partitioning guidance, Sharding pattern.

Videos:

- FailSafe: Building Scalable, Resilient Cloud Services. Nine-part series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents high-level concepts and architectural principles in a very accessible and interesting way, with stories drawn from Microsoft Customer Advisory Team (CAT) experience with actual customers. See the partitioning discussion in episode 7.
- Building Big: Lessons learned from Windows Azure customers - Part I. Mark Simms discusses partitioning schemes, sharding strategies, how to implement sharding, and SQL Database Federations, starting at 19:49. Similar to the Failsafe series but goes into more how-to details.

Sample code:

- Cloud Service Fundamentals in Windows Azure. Sample application that includes a sharded database. For a description of the sharding scheme implemented, see DAL – Sharding of RDBMS on the Windows Azure blog.

PREVIOUS NEXT

Unstructured Blob Storage (Building Real-World Cloud Apps with Azure)

2/20/2020 • 10 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the e-book, see the first chapter.

In the previous chapter we looked at partitioning schemes and explained how the Fix It app stores images in the Azure Storage Blob service, and other task data in Azure SQL Database. In this chapter we go deeper into the Blob service and show how it's implemented in Fix It project code.

What is Blob storage?

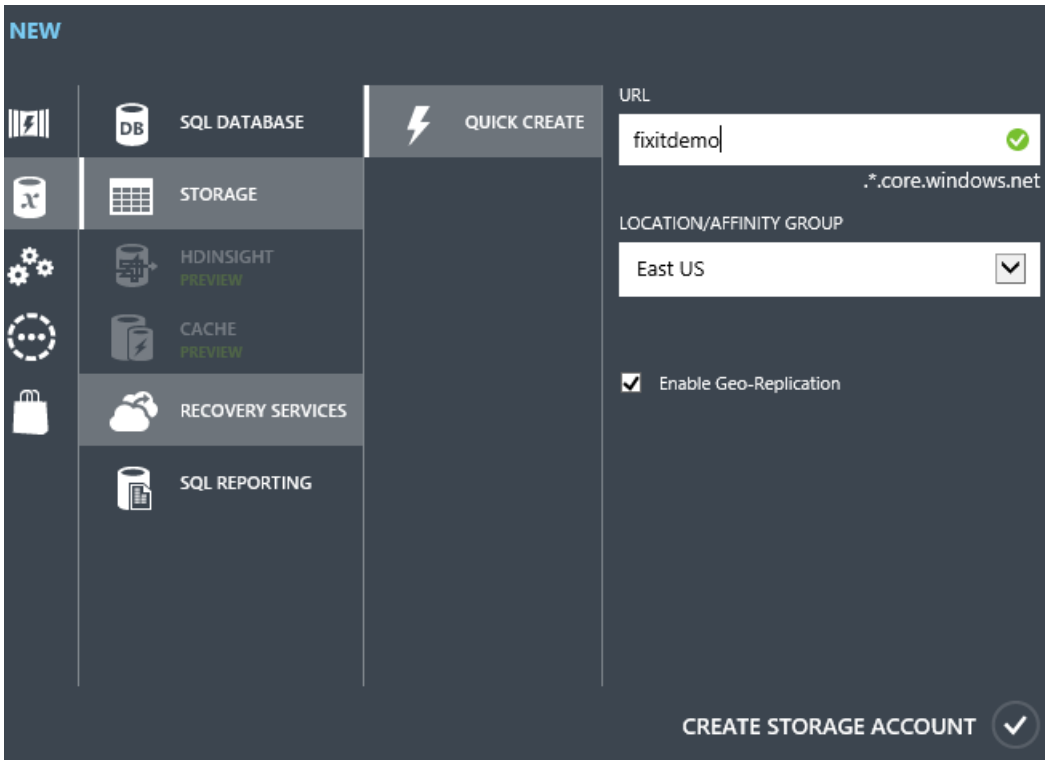
The Azure Storage Blob service provides a way to store files in the cloud. The Blob service has a number of advantages over storing files in a local network file system:

- It's highly scalable. A single Storage account can store hundreds of terabytes, and you can have multiple Storage accounts. Some of the biggest Azure customers store hundreds of petabytes. Microsoft SkyDrive uses blob storage.
- It's durable. Every file you store in the Blob service is automatically backed up.
- It provides high availability. The SLA for Storage promises 99.9% or 99.99% uptime, depending on which geo-redundancy option you choose.
- It's a platform-as-a-service (PaaS) feature of Azure, which means you just store and retrieve files, paying only for the actual amount of storage you use, and Azure automatically takes care of setting up and managing all of the VMs and disk drives required for the service.
- You can access the Blob service by using a REST API or by using a programming language API. SDKs are available for .NET, Java, Ruby, and others.
- When you store a file in the Blob service, you can easily make it publicly available over the Internet.
- You can secure files in the Blob service so they can accessed only by authorized users, or you can provide temporary access tokens that makes them available to someone only for a limited period of time.

Anytime you're building an app for Azure and you want to store a lot of data that in an on-premises environment would go in files -- such as images, videos, PDFs, spreadsheets, etc. -- consider the Blob service.

Creating a Storage account

To get started with the Blob service you create a Storage account in Azure. In the portal, click **New** -- **Data Services** -- **Storage** -- **Quick Create**, and then enter a URL and a data center location. The data center location should be the same as your web app.



You pick the primary region where you want to store the content, and if you choose the geo-replication option, Azure creates replicas of all your data in a different data center in another region of the country. For example, if you choose the Western US data center, when you store a file it goes to the Western US data center, but in the background Azure also copies it to one of the other US data centers. If a disaster happens in one region of the country, your data is still safe.

Azure won't replicate data across geo-political boundaries: if your primary location is in the U.S., your files are only replicated to another region within the U.S.; if your primary location is Australia, your files are only replicated to another data center in Australia.

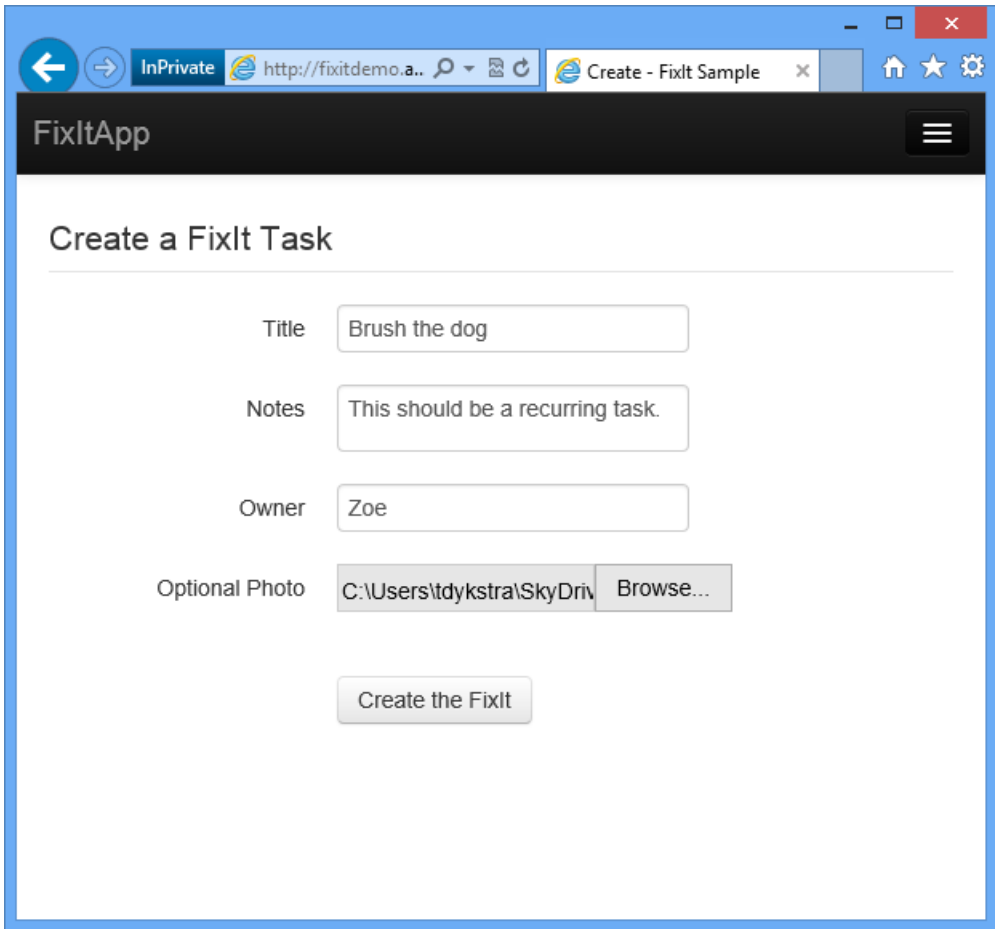
Of course, you can also create a Storage account by executing commands from a script, as we saw earlier. Here's a Windows PowerShell command to create a Storage account:

```
# Create a new storage account
New-AzureStorageAccount -StorageAccountName $Name -Location $Location -Verbose
```

Once you have a Storage account, you can immediately start storing files in the Blob service.

Using Blob storage in the Fix It app

The Fix It app enables you to upload photos.



When you click **Create the FixIt**, the application uploads the specified image file and stores it in the Blob service.

Set up the Blob container

In order to store a file in the Blob service you need a *container* to store it in. A Blob service container corresponds to a file system folder. The environment creation scripts that we reviewed in the Automate Everything chapter create the Storage account, but they don't create a container. So the purpose of the `CreateAndConfigure` method of the `PhotoService` class is to create a container if it doesn't already exist. This method is called from the `Application_Start` method in *Global.asax*.

```
public async void CreateAndConfigureAsync()
{
    try
    {
        CloudStorageAccount storageAccount = StorageUtils.StorageAccount;

        // Create a blob client and retrieve reference to images container
        CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
        CloudBlobContainer container = blobClient.GetContainerReference("images");

        // Create the "images" container if it doesn't already exist.
        if (await container.CreateIfNotExistsAsync())
        {
            // Enable public access on the newly created "images" container
            await container.SetPermissionsAsync(
                new BlobContainerPermissions
                {
                    PublicAccess =
                        BlobContainerPublicAccessType.Blob
                });

            log.Information("Successfully created Blob Storage Images Container and made it public");
        }
    }
    catch (Exception ex)
    {
        log.Error(ex, "Failure to Create or Configure images container in Blob Storage Service");
    }
}
```

The storage account name and access key are stored in the `appSettings` collection of the *Web.config* file, and code in the `StorageUtils.StorageAccount` method uses those values to build a connection string and establish a connection:

```
string account = CloudConfigurationManager.GetSetting("StorageAccountName");
string key = CloudConfigurationManager.GetSetting("StorageAccountAccessKey");
string connectionString = String.Format("DefaultEndpointsProtocol=https;AccountName={0};AccountKey={1}",
account, key);
return CloudStorageAccount.Parse(connectionString);
```

The `CreateAndConfigureAsync` method then creates an object that represents the Blob service, and an object that represents a container (folder) named "images" in the Blob service:

```
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
CloudBlobContainer container = blobClient.GetContainerReference("images");
```

If a container named "images" doesn't exist yet -- which will be true the first time you run the app against a new storage account -- the code creates the container and sets permissions to make it public. (By default, new blob containers are private and are accessible only to users who have permission to access your storage account.)

```
if (await container.CreateIfNotExistsAsync())
{
    // Enable public access on the newly created "images" container
    await container.SetPermissionsAsync(
        new BlobContainerPermissions
        {
            PublicAccess =
                BlobContainerPublicAccessType.Blob
        });

    log.Information("Successfully created Blob Storage Images Container and made it public");
}
```

Store the uploaded photo in Blob storage

To upload and save the image file, the app uses an `IPhotoService` interface and an implementation of the interface in the `PhotoService` class. The *PhotoService.cs* file contains all of the code in the Fix It app that communicates with the Blob service.

The following MVC controller method is called when the user clicks **Create the FixIt**. In this code, `photoService` refers to an instance of the `PhotoService` class, and `fixittask` refers to an instance of the `FixItTask` entity class that stores data for a new task.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Create([Bind(Include =
"FixItTaskId,CreatedBy,Owner,Title,Notes,PhotoUrl,IsDone")]FixItTask fixittask, HttpPostedFileBase photo)
{
    if (ModelState.IsValid)
    {
        fixittask.CreatedBy = User.Identity.Name;
        fixittask.PhotoUrl = await photoService.UploadPhotoAsync(photo);
        await fixItRepository.CreateAsync(fixittask);
        return RedirectToAction("Success");
    }

    return View(fixittask);
}
```

The `UploadPhotoAsync` method in the `PhotoService` class stores the uploaded file in the Blob service and returns a URL that points to the new blob.

```
public async Task<string> UploadPhotoAsync(HttpPostedFileBase photoToUpload)
{
    if (photoToUpload == null || photoToUpload.ContentLength == 0)
    {
        return null;
    }

    string fullPath = null;
    Stopwatch timespan = Stopwatch.StartNew();

    try
    {
        CloudStorageAccount storageAccount = StorageUtils.StorageAccount;

        // Create the blob client and reference the container
        CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
        CloudBlobContainer container = blobClient.GetContainerReference("images");

        // Create a unique name for the images we are about to upload
        string imageName = String.Format("task-photo-{0}{1}",
            Guid.NewGuid().ToString(),
            Path.GetExtension(photoToUpload.FileName));

        // Upload image to Blob Storage
        CloudBlockBlob blockBlob = container.GetBlockBlobReference(imageName);
        blockBlob.Properties.ContentType = photoToUpload.ContentType;
        await blockBlob.UploadFromStreamAsync(photoToUpload.InputStream);

        // Convert to be HTTP based URI (default storage path is HTTPS)
        var uriBuilder = new UriBuilder(blockBlob.Uri);
        uriBuilder.Scheme = "http";
        fullPath = uriBuilder.ToString();

        timespan.Stop();
        log.TraceApi("Blob Service", "PhotoService.UploadPhoto", timespan.Elapsed, "imagepath={0}",
fullPath);
    }
    catch (Exception ex)
    {
        log.Error(ex, "Error upload photo blob to storage");
    }

    return fullPath;
}
```

As in the `CreateAndConfigure` method, the code connects to the storage account and creates an object that represents the "images" blob container, except here it assumes the container already exists.

Then it creates a unique identifier for the image about to be uploaded, by concatenating a new GUID value with the file extension:

```
string imageName = String.Format("task-photo-{0}{1}",
    Guid.NewGuid().ToString(),
    Path.GetExtension(photoToUpload.FileName));
```

The code then uses the blob container object and the new unique identifier to create a blob object, sets an attribute on that object indicating what kind of file it is, and then uses the blob object to store the file in blob storage.

```
CloudBlockBlob blockBlob = container.GetBlockBlobReference(imageName);
blockBlob.Properties.ContentType = photoToUpload.ContentType;
blockBlob.UploadFromStream(photoToUpload.InputStream);
```

Finally, it gets a URL that references the blob. This URL will be stored in the database and can be used in Fix It web pages to display the uploaded image.

```
fullPath = String.Format("http://{0}{1}", blockBlob.Uri.DnsSafeHost, blockBlob.Uri.AbsolutePath);
```

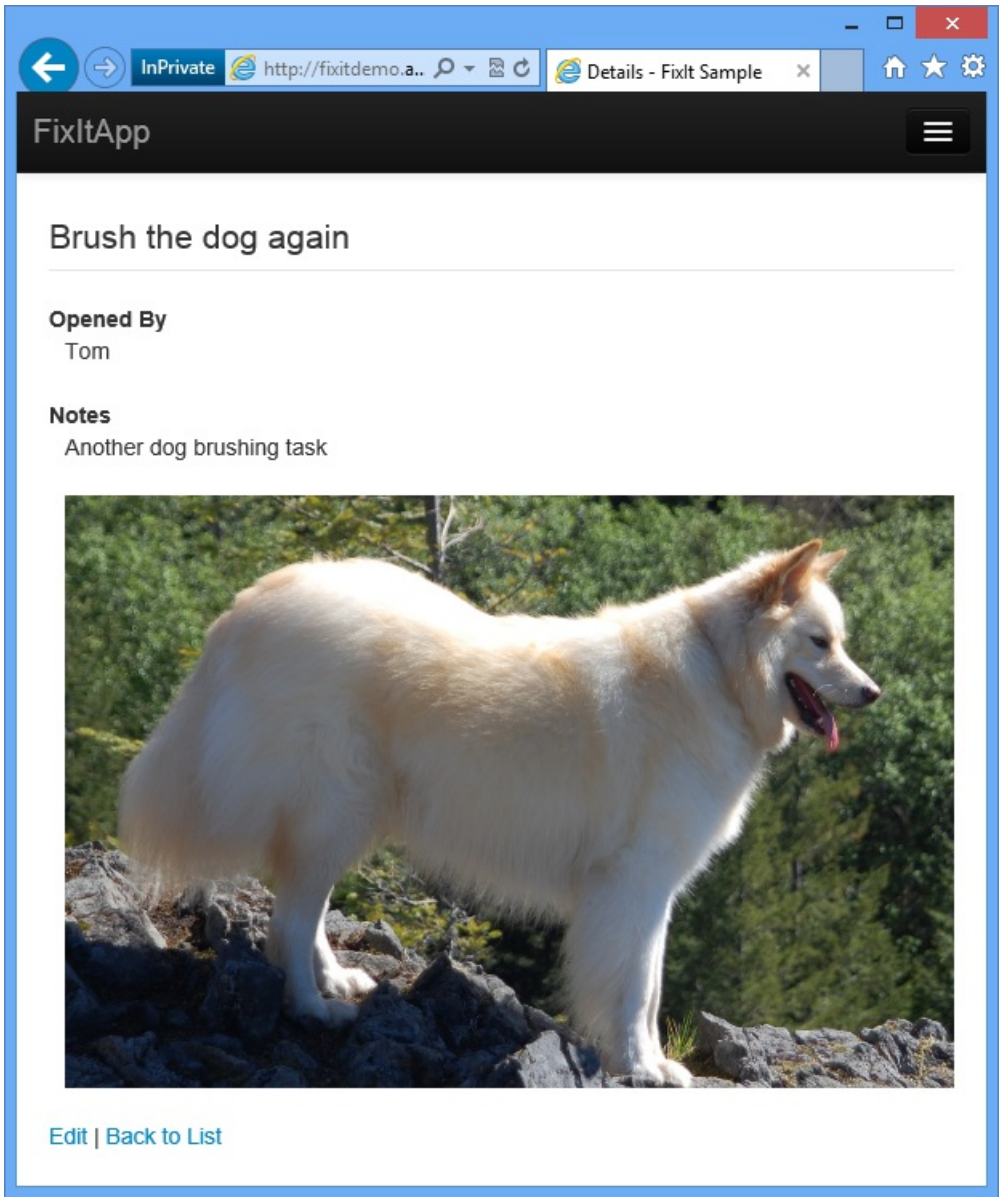
This URL is stored in the database as one of the columns of the FixItTask table.

```
public class FixItTask
{
    public int FixItTaskId  { get; set; }
    public string CreatedBy { get; set; }
    [Required]
    public string Owner     { get; set; }
    [Required]
    public string Title     { get; set; }
    public string Notes     { get; set; }
    public string PhotoUrl  { get; set; }
    public bool IsDone      { get; set; }
}
```

With only the URL in the database, and images in Blob storage, the Fix It app keeps the database small, scalable, and inexpensive, while the images are stored where storage is cheap and capable of handling terabytes or petabytes. One storage account can store hundreds of terabytes of Fix It photos, and you only pay for what you use. So you can start off small paying 9 cents for the first gigabyte, and add more images for pennies per additional gigabyte.

Display the uploaded file

The Fix It application displays the uploaded image file when it displays details for a task.



To display the image, all the MVC view has to do is include the `PhotoUrl` value in the HTML sent to the browser. The web server and the database are not using cycles to display the image, they are only serving up a few bytes to the image URL. In the following Razor code, `Model` refers to an instance of the `FixItTask` entity class.

```
<fieldset>
<legend>@Html.DisplayFor(model => model.Title)</legend>
<dl>
    <dt>Opened By</dt>
    <dd>@Html.DisplayFor(model => model.CreatedBy)</dd>
                <br />
    <dt>@Html.DisplayNameFor(model => model.Notes)</dt>
    <dd>@Html.DisplayFor(model => model.Notes)</dd>
                <br />
                @if(Model.PhotoUrl != null) {
        <dd><img src="@Model.PhotoUrl" title="@Model.Title" /></dd>
                }
</dl>
</fieldset>
```

If you look at the HTML of the page that displays, you see the URL pointing directly to the image in blob storage, something like this:

```
<fieldset>
<legend>Brush the dog again</legend>
<dl>
    <dt>Opened By</dt>
    <dd>Tom</dd>
                <br />
    <dt>Notes</dt>
    <dd>Another dog brushing task</dd>
                <br />
    <dd>
<img src="http://storageaccountname.blob.core.windows.net/images/task-photo-312dd635-ba87-4542-8b15-
767032c55f4e.jpg"
           title="Brush the dog again" />
    </dd>
</dl>
</fieldset>
```

Summary

You've seen how the Fix It app stores images in the Blob service and only image URLs in the SQL database. Using the Blob service keeps the SQL database much smaller than it otherwise would be, makes it possible to scale up to an almost unlimited number of tasks, and can be done without writing a lot of code.

You can have hundreds of terabytes in a storage account, and the storage cost is much less expensive than SQL Database storage, starting at about 3 cents per gigabyte per month plus a small transaction charge. And keep in mind that you're not paying for the maximum capacity but only for the amount you actually store, so your app is ready to scale but you're not paying for all that extra capacity.

In the next chapter we'll talk about the importance of making a cloud app capable of gracefully handling failures.

Resources

For more information see the following resources:

- An Introduction to Azure BLOB Storage. Blog by Mike Wood.
- How to use the Azure Blob Storage Service in .NET. Official documentation on the MicrosoftAzure.com site. A brief introduction to blob storage followed by code examples showing how to connect to blob storage, create containers, upload and download blobs, etc.
- FailSafe: Building Scalable, Resilient Cloud Services. Nine-part video series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents high-level concepts and architectural principles in a very accessible and interesting way, with stories drawn from Microsoft Customer Advisory Team (CAT) experience with actual customers. For a discussion of Azure Storage service and blobs, see episode 5 starting at 35:13.
- Microsoft Patterns and Practices - Azure Guidance. See Valet Key pattern.

PREVIOUS NEXT

Design to Survive Failures (Building Real-World Cloud Apps with Azure)

2/20/2020 • 11 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the e-book, see the first chapter.

One of the things you have to think about when you build any type of application, but especially one that will run in the cloud where lots of people will be using it, is how to design the app so that it can gracefully handle failures and continue to deliver value as much as possible. Given enough time, things are going to go wrong in any environment or any software system. How your app handles those situations determines how upset your customers will get and how much time you have to spend analyzing and fixing problems.

Types of failures

There are two basic categories of failures that you'll want to handle differently:

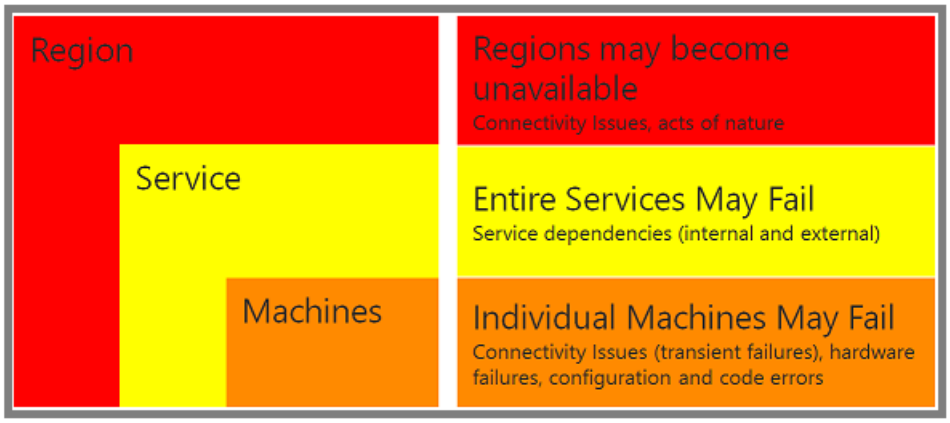
- Transient, self-healing failures such as intermittent network connectivity issues.
- Enduring failures that require intervention.

For transient failures, you can implement a retry policy to ensure that most of the time the app recovers quickly and automatically. Your customers might notice slightly longer response time, but otherwise they won't be affected. We'll show some ways to handle these errors in the Transient Fault Handling chapter.

For enduring failures, you can implement monitoring and logging functionality that notifies you promptly when issues arise and that facilitates root cause analysis. We'll show some ways to help you stay on top of these kinds of errors in the Monitoring and Telemetry chapter.

Failure scope

You also have to think about failure scope – whether a single machine is affected, a whole service such as SQL Database or Storage, or an entire region.



Machine failures

In Azure, a failed server is automatically replaced by a new one, and a well-designed cloud app recovers from this kind of failure automatically and quickly. Earlier we stressed the scalability benefits of a stateless web tier, and ease of recovery from a failed server is another benefit of statelessness. Ease of recovery is also one of the benefits of platform-as-a-service (PaaS) features such as SQL Database and Azure App Service Web Apps. Hardware failures are rare, but when they occur these services handle them automatically; you don't even have to write code to handle machine failures when you're using one of these services.

Service failures

Cloud apps typically use multiple services. For example, the Fix It app uses the SQL Database service, the Storage service, and the web app is deployed to Azure App Service. What will your app do if one of the services you depend on fails? For some service failures a friendly "sorry, try again later" message might be the best you can do. But in many scenarios you can do better. For example, when your back-end data store is down, you can accept user input, display "your request has been received," and store the input someplace else temporarily; then when the service you need is operational again, you can retrieve the input and process it.

The Queue-centric Work Pattern chapter shows one way to handle this scenario. The Fix It app stores tasks in SQL Database, but it doesn't have to quit working when SQL Database is down. In that chapter we'll see how to store user input for a task in a queue, and use a worker process to read the queue and update the task. If SQL is down, the ability to create Fix It tasks is unaffected; the worker process can wait and process new tasks when SQL Database is available.

Region failures

Entire regions may fail. A natural disaster might destroy a data center, it might get flattened by a meteor, the trunk line into the datacenter could be cut by a farmer burying a cow with a backhoe, etc. If your app is hosted in the stricken datacenter what do you do? It's possible to set up your app in Azure to run in multiple regions simultaneously so that if there's a disaster in one, you continue running in another region. Such failures are extremely rare occurrences, and most apps don't jump through the hoops necessary to ensure uninterrupted service through failures of this sort. See the Resources section at the end of the chapter for information about how to keep your app available even through a region failure.

A goal of Azure is to make handling all of these kinds of failures a lot easier, and you'll see some examples of how we're doing that in the following chapters.

SLAs

People often hear about service-level agreements (SLAs) in the cloud environment. Basically these are promises that companies make about how reliable their service is. A 99.9% SLA means you should expect the service to be working correctly 99.9% of the time. That's a fairly typical value for an SLA and it sounds like a very high number, but you might not realize how much down time .1% actually amounts to. Here's a table that shows how much downtime various SLA percentages amount to over a year, a month, and a week.

Availability %	Downtime per year	Downtime per month*	Downtime per week
90% ("one nine")	36.5 days	72 hours	16.8 hours
99% ("two nines")	3.65 days	7.20 hours	1.68 hours
99.9% ("three nines")	8.76 hours	43.2 minutes	10.1 minutes
99.99% ("four nines")	52.56 minutes	4.32 minutes	1.01 minutes
99.999% ("five nines")	5.26 minutes	25.9 seconds	6.05 seconds
99.9999% ("six nines")	31.5 seconds	2.59 seconds	0.605 seconds

So a 99.9% SLA means your service could be down 8.76 hours a year or 43.2 minutes a month. That's more down time than most people realize. So as a developer you want to be aware that a certain amount of down time is possible and handle it in a graceful way. At some point someone is going to be using your app, and a service is going to be down, and you want to minimize the negative impact of that on the customer.

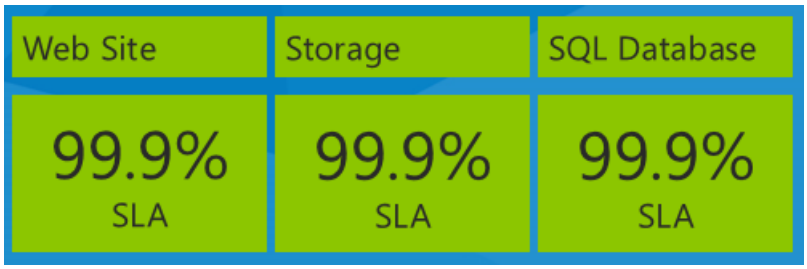
One thing you should know about an SLA is what time-frame it refers to: does the clock get reset every week,

every month, or every year? In Azure we reset the clock every month, which is better for you than a yearly SLA, since a yearly SLA could hide bad months by offsetting them with a series of good months.

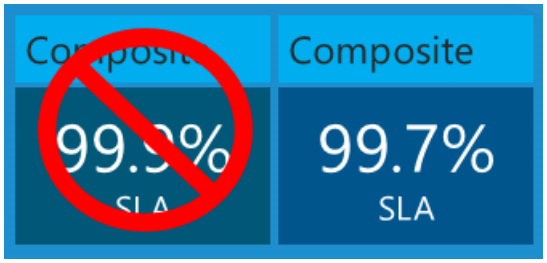
Of course we always aspire to do better than the SLA; usually you'll be down much less than that. The promise is that if we're ever down for longer than the maximum down time you can ask for money back. The amount of money you get back probably wouldn't fully compensate you for the business impact of the excess down time, but that aspect of the SLA acts as an enforcement policy and lets you know that we do take it very seriously.

Composite SLAs

An important thing to think about when you're looking at SLAs is the impact of using multiple services in an app, with each service having a separate SLA. For example, the Fix It app uses Azure App Service Web Apps, Azure Storage, and SQL Database. Here are their SLA numbers as of the date this e-book is being written in December, 2013:



What is the maximum down time you would expect for the app based on these service SLAs? You might think that your down time would be equal to the worst SLA percentage, or 99.9% in this case. That would be true if all three services always failed at the same time, but that isn't necessarily what actually happens. Each service may fail independently at different times, so you have to calculate the composite SLA by multiplying the individual SLA numbers.



So your app could be down not just 43.2 minutes a month but 3 times that amount, 108 minutes a month – and still be within the Azure SLA limits.

This issue is not unique to Azure. We actually provide the best cloud SLAs of any cloud service available, and you'll have similar issues to deal with if you use any vendor's cloud services. What this highlights is the importance of thinking about how you can design your app to handle the inevitable service failures gracefully, because they might happen often enough to impact your customers or users.

Cloud SLAs compared to enterprise down-time experience

People sometimes say, "In my enterprise app I never have these problems." If you ask how much down time a month they actually have, they usually say, "Well, it happens occasionally." And if you ask how often, they admit that "Sometimes we do need to back up or install a new server or update software." Of course, that counts as down time. Most enterprise apps unless they are especially mission-critical are actually down for more than the amount of time allowed by our service SLAs. But when it's your server and your infrastructure and you're responsible for it and in control of it, you tend to feel less angst about down times. In a cloud environment you're dependent on someone else and you don't know what's going on, so you might tend to get more worried about it.

When an enterprise achieves a greater up-time percentage than you get from a cloud SLA, they do it by spending a lot more money on hardware. A cloud service could do that but would have to charge much more for its services. Instead, you take advantage of a cost-effective service and design your software so that the inevitable

failures cause minimum disruption to your customers. Your job as a cloud app designer is not so much to avoid failure as to avoid catastrophe, and you do that by focusing on software, not on hardware. Whereas enterprise apps strive to maximize mean time between failures, cloud apps strive to minimize mean time to recover.

Not all cloud services have SLAs

Be aware also that not every cloud service even has an SLA. If your app is dependent on a service with no up-time guarantee, you could be down far longer than you might imagine. For example, if you enable log-in to your site using a social provider such as Facebook or Twitter, check with the service provider to find out if there is an SLA, and you might find out there isn't one. But if the authentication service goes down or is unable to support the volume of requests you throw at it, your customers are locked out of your app. You could be down for days or longer. The creators of one new app expected hundreds of millions of downloads and took a dependency on Facebook authentication – but didn't talk to Facebook before going live and discovered too late that there was no SLA for that service.

Not all downtime counts toward SLAs

Some cloud services may deliberately deny service if your app over-uses them. This is called *throttling*. If a service has an SLA, it should state the conditions under which you might be throttled, and your app design should avoid those conditions and react appropriately to the throttling if it happens. For example, if requests to a service start to fail when you exceed a certain number per second, you want to make sure automatic retries don't happen so fast that they cause the throttling to continue. We'll have more to say about throttling in the Transient Fault Handling chapter.

Summary

This chapter has tried to help you realize why a real world cloud app has to be designed to survive failures gracefully. Starting with the next chapter, the remaining patterns in this series go into more detail about some strategies you can use to do that:

- Have good monitoring and telemetry, so that you find out quickly about failures that require intervention, and you have sufficient information to resolve them.
- Handle transient faults by implementing intelligent retry logic, so that your app recovers automatically when it can and falls back to circuit breaker logic when it can't.
- Use distributed caching in order to minimize throughput, latency, and connection problems with database access.
- Implement loose coupling via the queue-centric work pattern, so that your app front end can continue to work when the back end is down.

Resources

For more information, see later chapters in this e-book and the following resources.

Documentation:

- Failsafe: Guidance for Resilient Cloud Architectures. White paper by Marc Mercuri, Ulrich Homann, and Andrew Townhill. Web page version of the FailSafe video series.
- Best Practices for the Design of Large-Scale Services on Azure Cloud Services. White paper by Mark Simms and Michael Thomassy.
- Azure Business Continuity Technical Guidance. White paper by Patrick Wickline and Jason Roth.
- Disaster Recovery and High Availability for Azure Applications. White paper by Michael McKeown, Hanu Kommalapati, and Jason Roth.
- Microsoft Patterns and Practices - Azure Guidance. See Multi Data Center Deployment guidance, Circuit breaker pattern.
- Azure Support - Service Level Agreements.

- Business Continuity in Azure SQL Database. Documentation about SQL Database high availability and disaster recovery features.
- High Availability and Disaster Recovery for SQL Server in Azure Virtual Machines.

Videos:

- FailSafe: Building Scalable, Resilient Cloud Services. Nine-part series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents high-level concepts and architectural principles in a very accessible and interesting way, with stories drawn from Microsoft Customer Advisory Team (CAT) experience with actual customers. Episodes 1 and 8 go in depth into the reasons for designing cloud apps to survive failures. See also the follow-up discussion of throttling in episode 2 starting at 49:57, the discussion of failure points and failure modes in episode 2 starting at 56:05, and the discussion of circuit breakers in episode 3 starting at 40:55.
- Building Big: Lessons learned from Azure customers - Part II. Mark Simms talks about designing for failure and instrumenting everything. Similar to the Failsafe series but goes into more how-to details.

PREVIOUS NEXT

Monitoring and Telemetry (Building Real-World Cloud Apps with Azure)

2/20/2020 • 18 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the e-book, see the first chapter.

A lot of people rely on customers to let them know when their application is down. That's not really a best practice anywhere, and especially not in the cloud. There's no guarantee of quick notification, and when you do get notified, you often get minimal or misleading data about what happened. With good telemetry and logging systems you can be aware of what's going on with your app, and when something does go wrong you find out right away and have helpful troubleshooting information to work with.

Buy or rent a telemetry solution

> **NOTE**
>
> This article was written before Application Insights was released. Application Insights is the preferred approach for telemetry solutions on Azure. See Set up Application Insights for your ASP.NET website for more information.

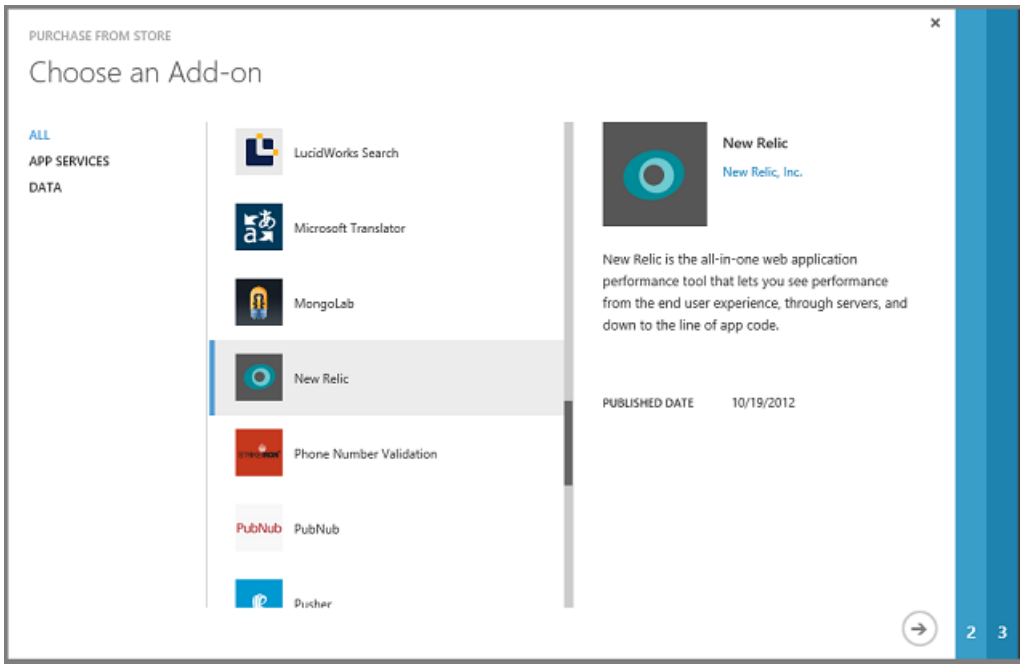
One of the things that's great about the cloud environment is that it's really easy to buy or rent your way to victory. Telemetry is an example. Without a lot of effort you can get a really good telemetry system up and running, very cost-effectively. There are a bunch of great partners that integrate with Azure, and some of them have free tiers – so you can get basic telemetry for nothing. Here are just a few of the ones currently available on Azure:

- New Relic
- AppDynamics
- Dynatrace

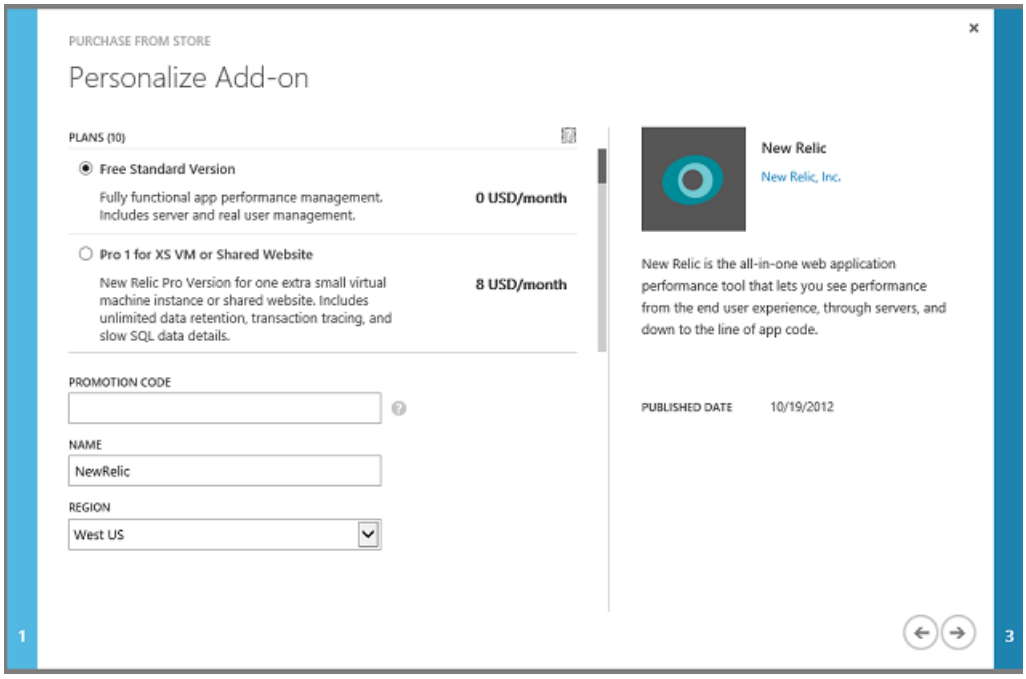
Microsoft System Center also includes monitoring features.

We'll quickly walk through setting up New Relic to show how easy it can be to use a telemetry system.

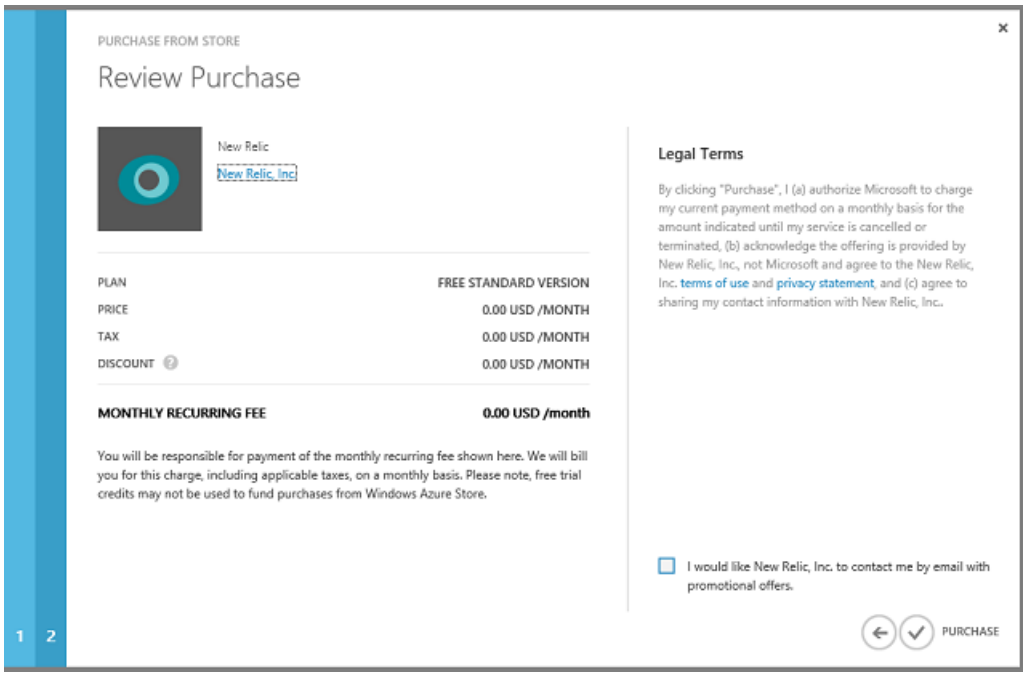
In the Azure management portal, sign up for the service. Click **New**, and then click **Store**. The **Choose an Add-on** dialog box appears. Scroll down and click **New Relic**.

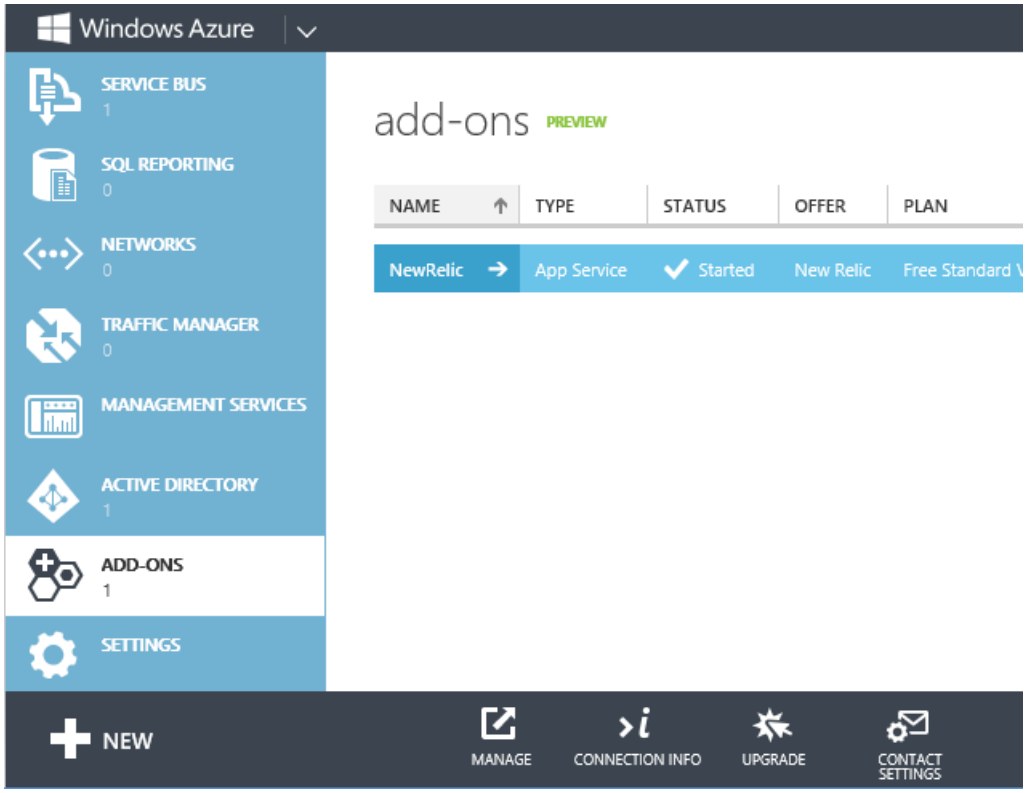


Click the right arrow and choose the service tier you want. For this demo we'll use the free tier.



Click the right arrow, confirm the "purchase," and New Relic now shows up as an Add-on in the portal.

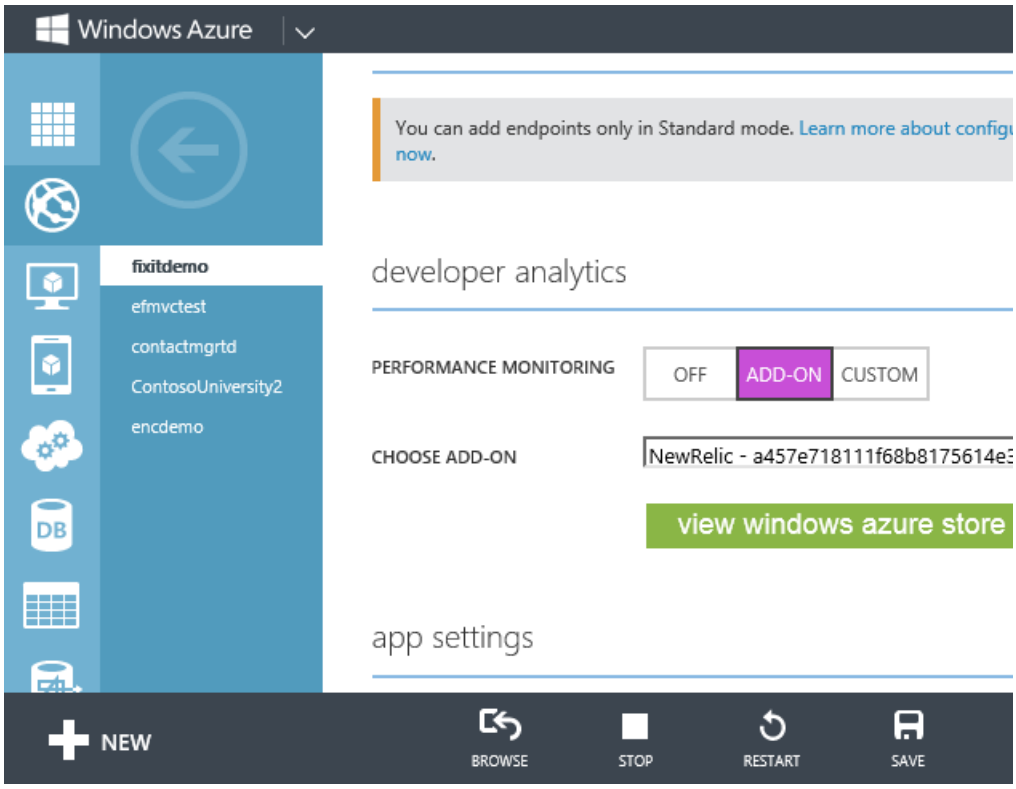




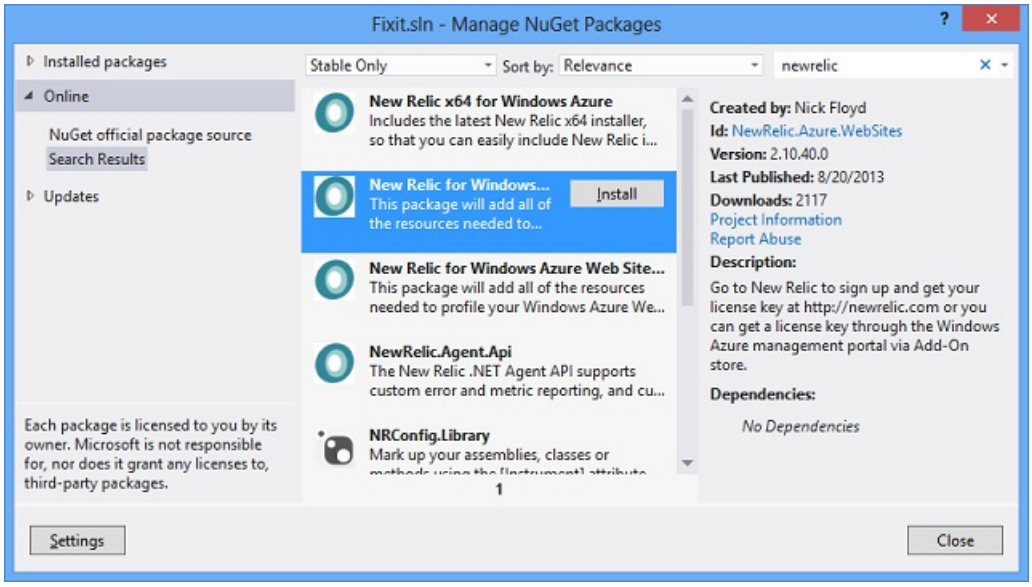
Click **Connection Info**, and copy the license key.

Go to the **Configure** tab for your web app in the portal, set **Performance Monitoring** to **Add-On**, and set the

Choose Add-On drop-down list to **New Relic**. Then click **Save**.

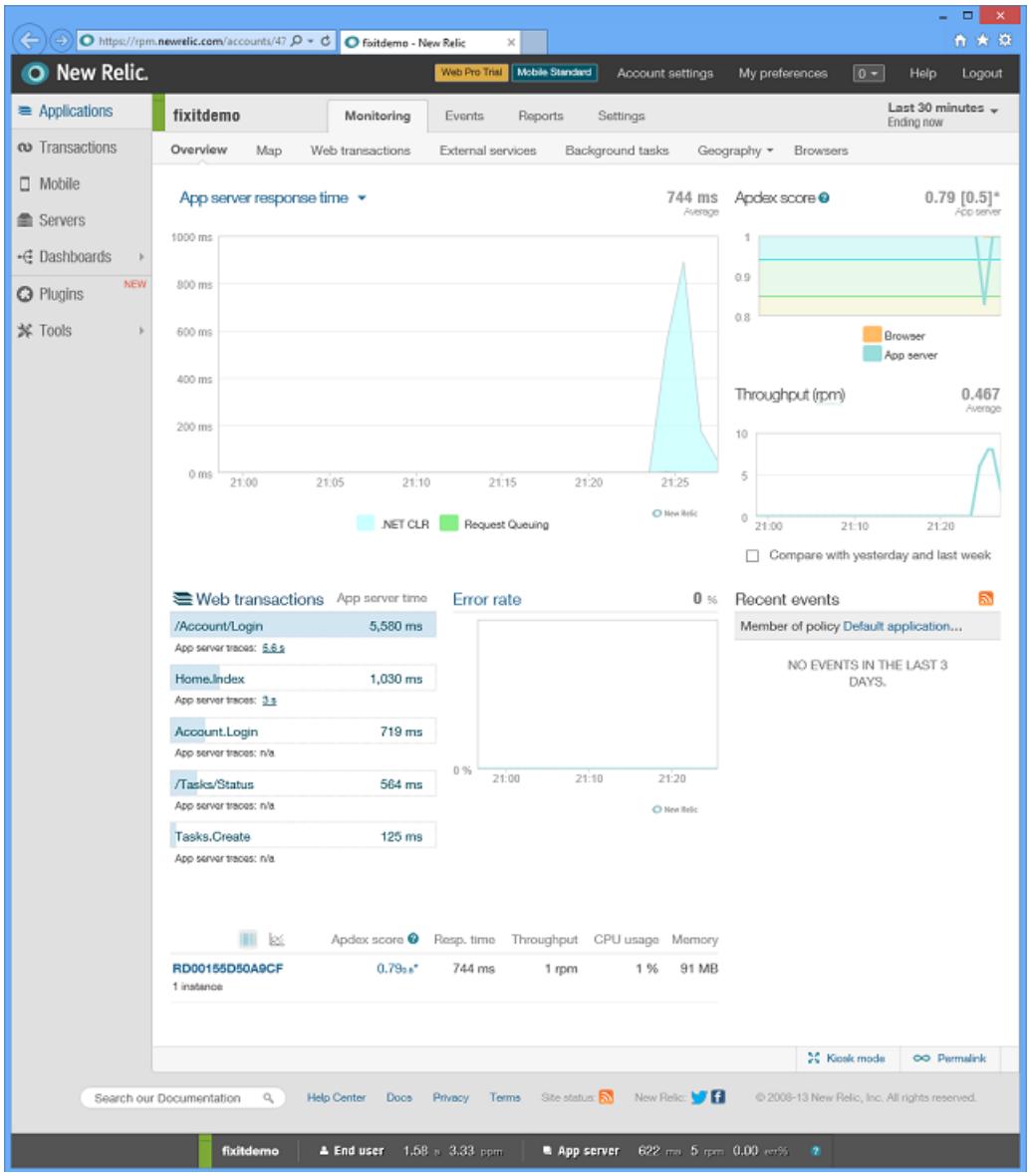


In Visual Studio, install the New Relic NuGet package in your app.



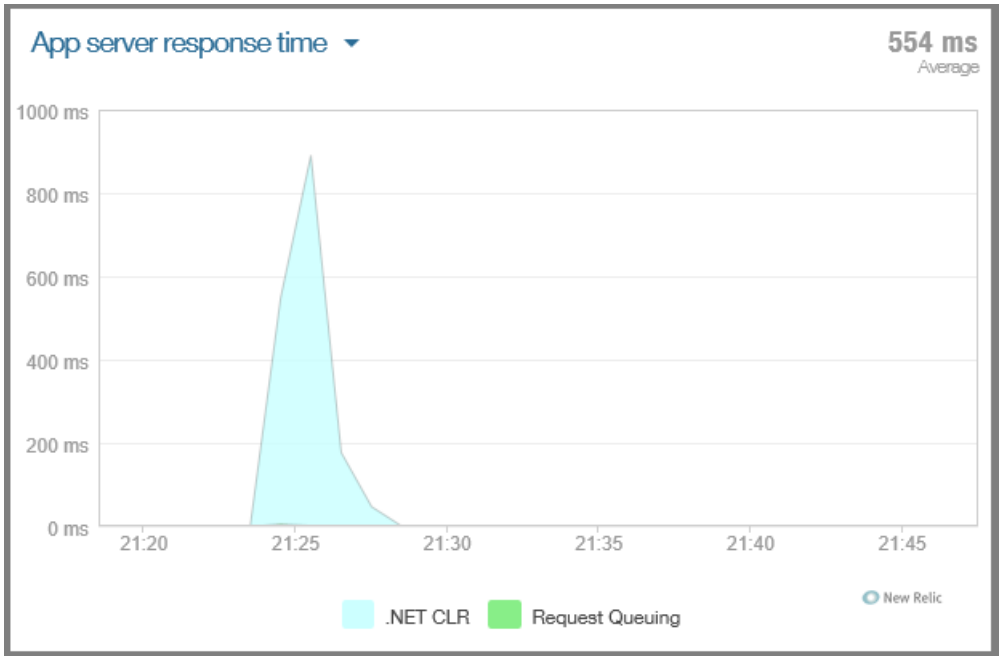
Deploy the app to Azure and start using it. Create a few Fix It tasks to provide some activity for New Relic to monitor.

Then go back to the **New Relic** page in the **Add-ons** tab of the portal and click **Manage**. The portal sends you to the New Relic management portal, using single sign-on for authentication so you don't have to enter your credentials again. The Overview page presents a variety of performance statistics. (Click the image to see the overview page full size.)



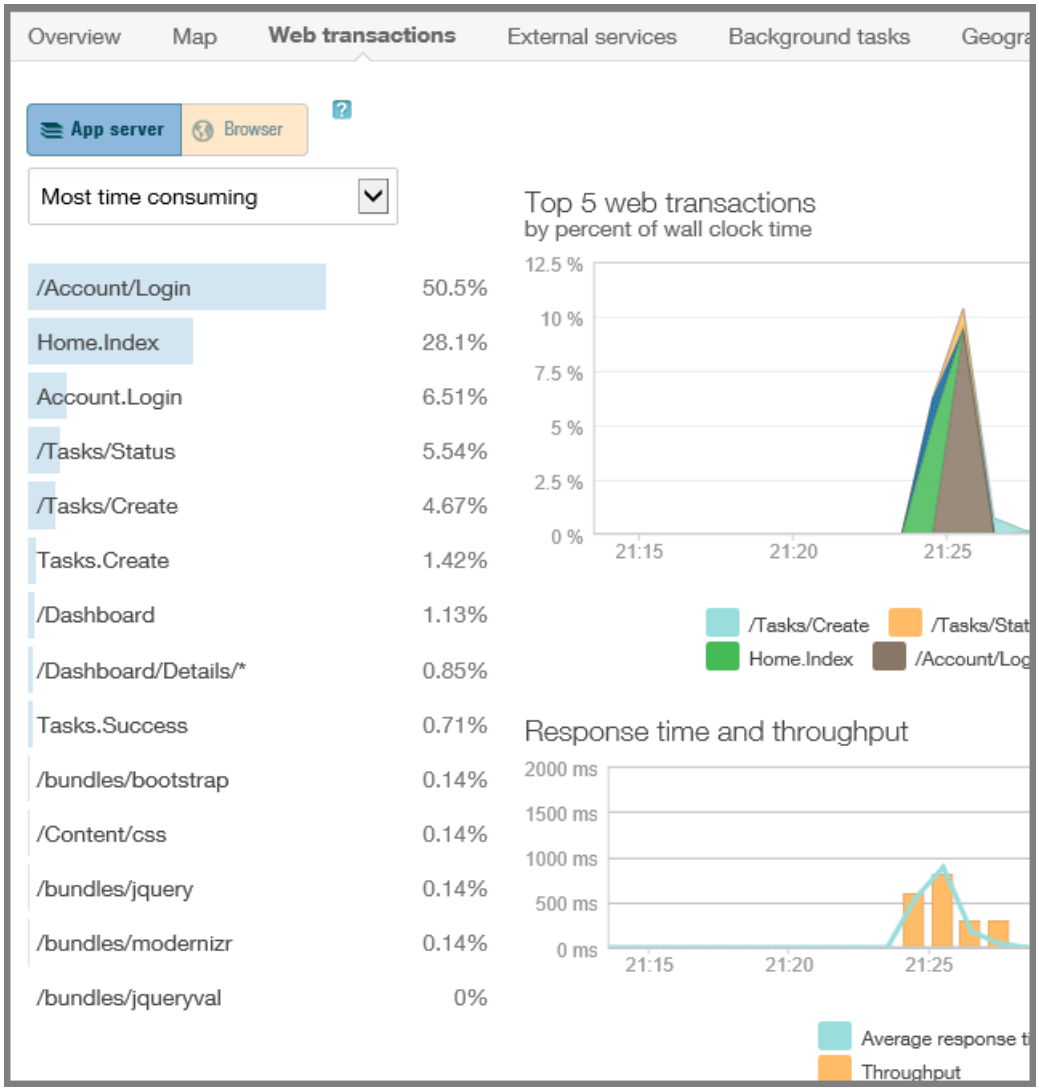
Here are just a few of the statistics you can see:

- Average response time at different times of day.



- Throughput rates (in requests per minute) at different times of day.

- Server CPU time spent handling different HTTP requests.



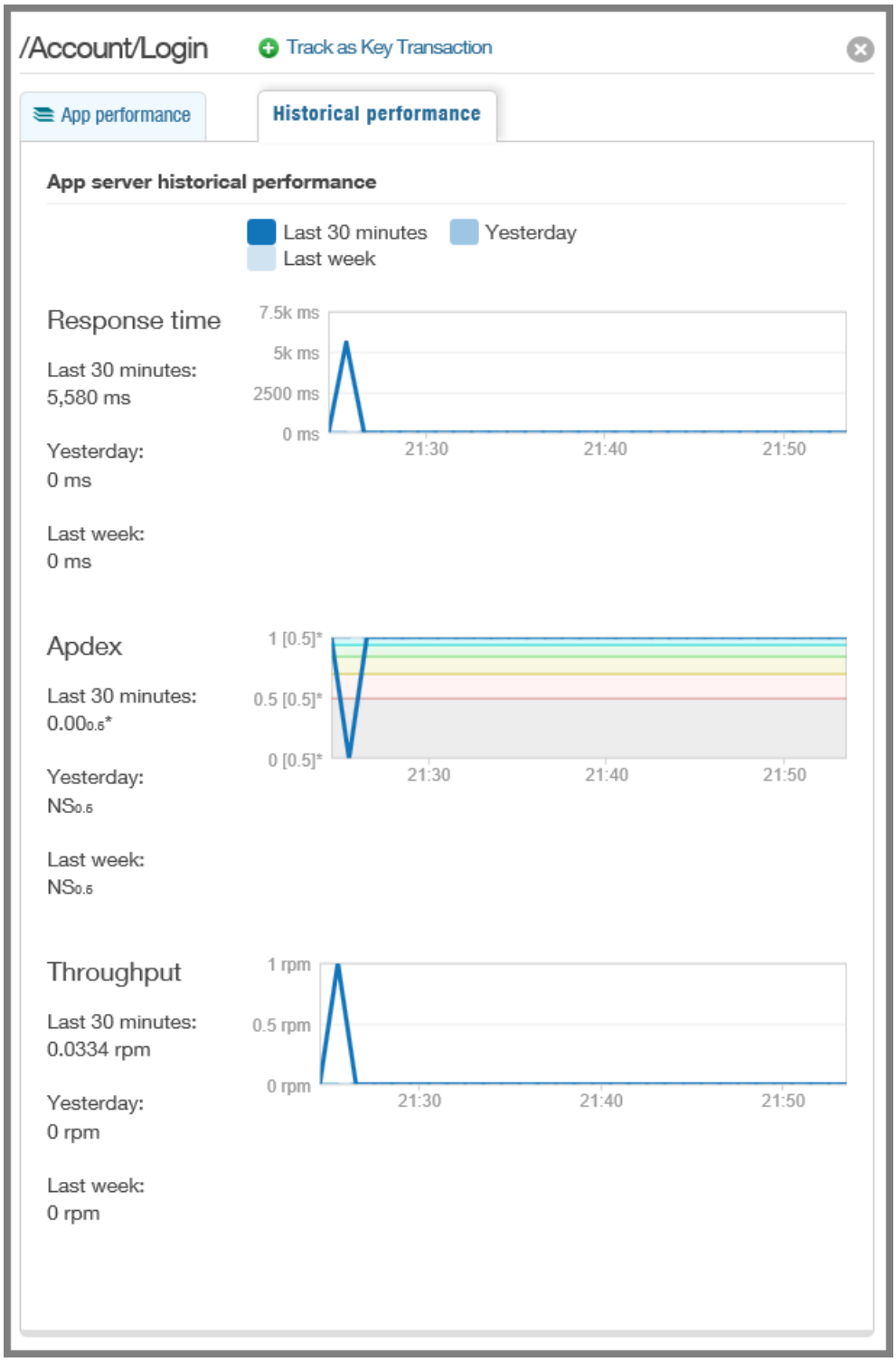
- CPU time spent in different parts of the application code:

Summary | **Trace details**

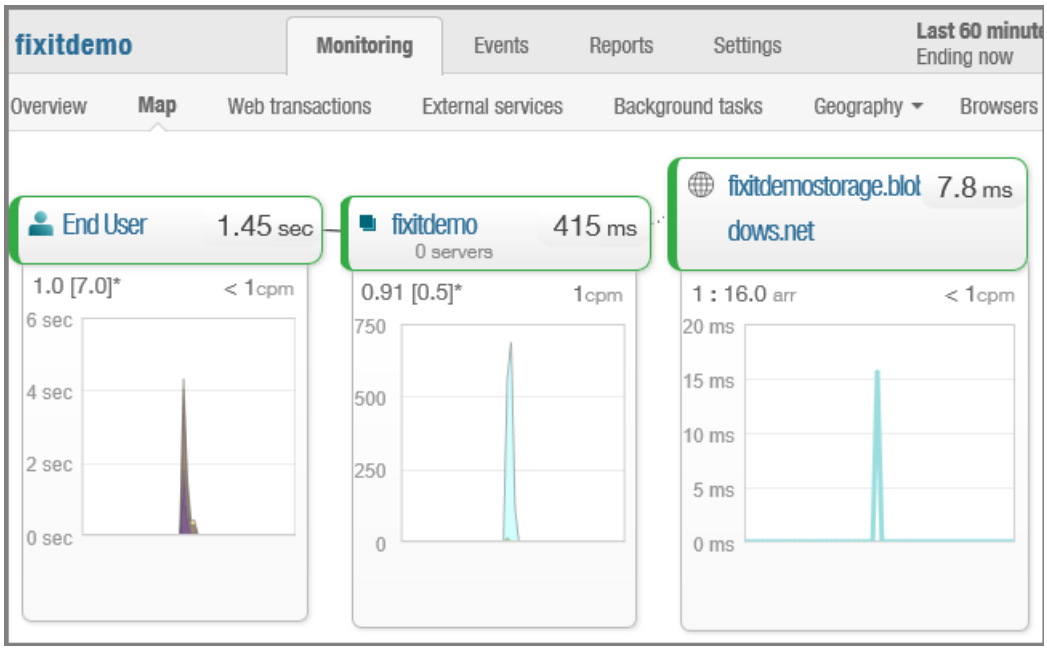
Expand performance problems Collapse all

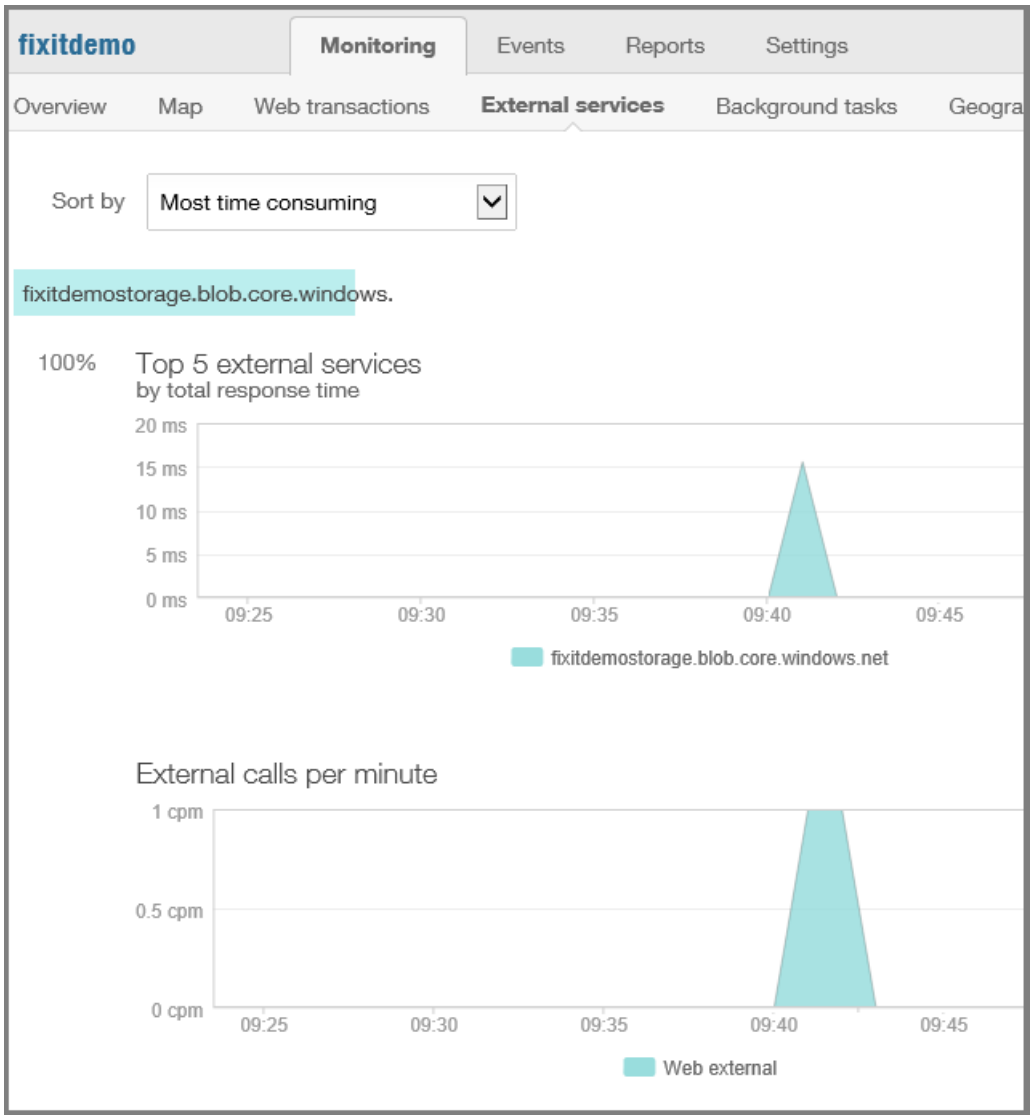
Duration (ms)	Duration (%)	Segment	Drilldown	Timestamp
5,370	**100.00%**	**System.Web.HttpApplication.BeginRequest()**		0.000 s
15.0	0.28%	19 fast method calls		0.000 s
844	15.71%	Application code (in System.Web.HttpApplication.BeginRequest())		0.015 s
0	0.00%	Iterate		0.859 s
1,230	22.95%	Application code (in System.Web.HttpApplication.BeginRequest())		0.859 s
0	0.00%	Iterate		2.092 s
328	6.11%	Application code (in System.Web.HttpApplication.BeginRequest())		2.092 s
0	0.00%	Iterate		2.420 s
0	0.00%	Iterate		2.483 s
1,470	**27.33%**	Application code (in System.Web.HttpApplication.BeginRequest())		2.483 s
515	9.59%	14 calls to Iterate		3.951 s
875	16.29%	Application code (in System.Web.HttpApplication.BeginRequest())		4.466 s
31.0	0.58%	8 fast method calls		5.341 s

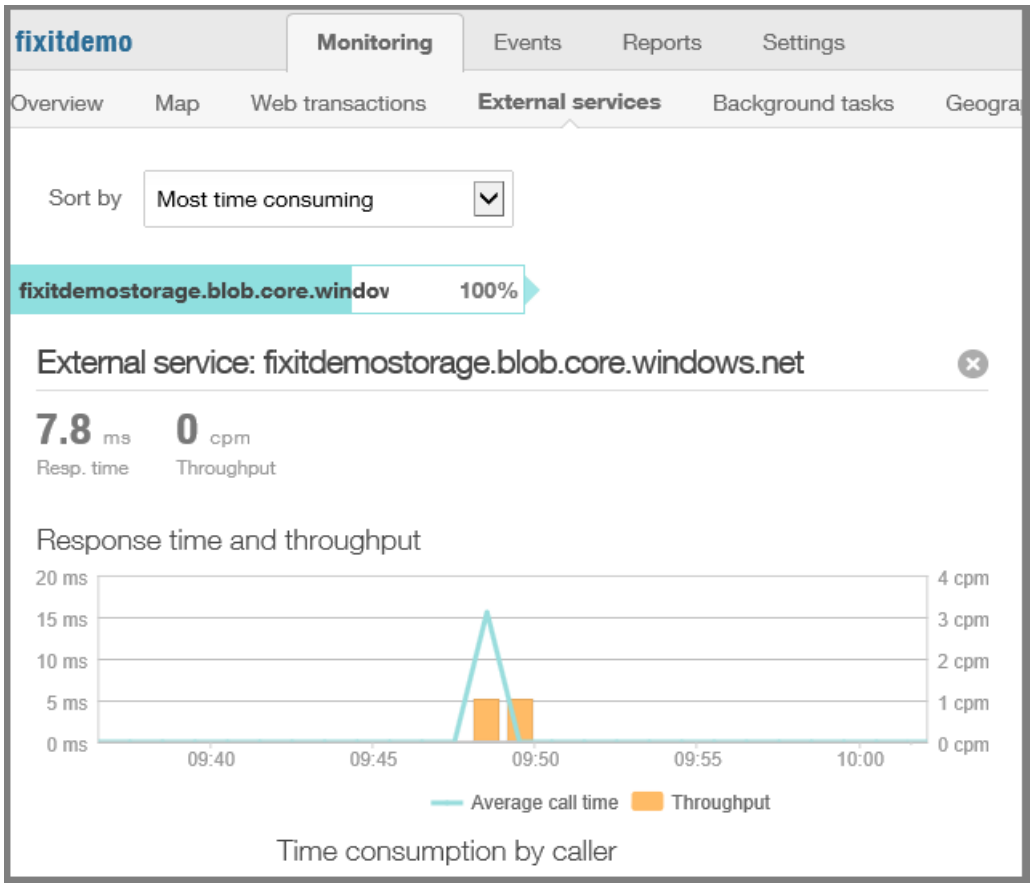
- Historical performance statistics.



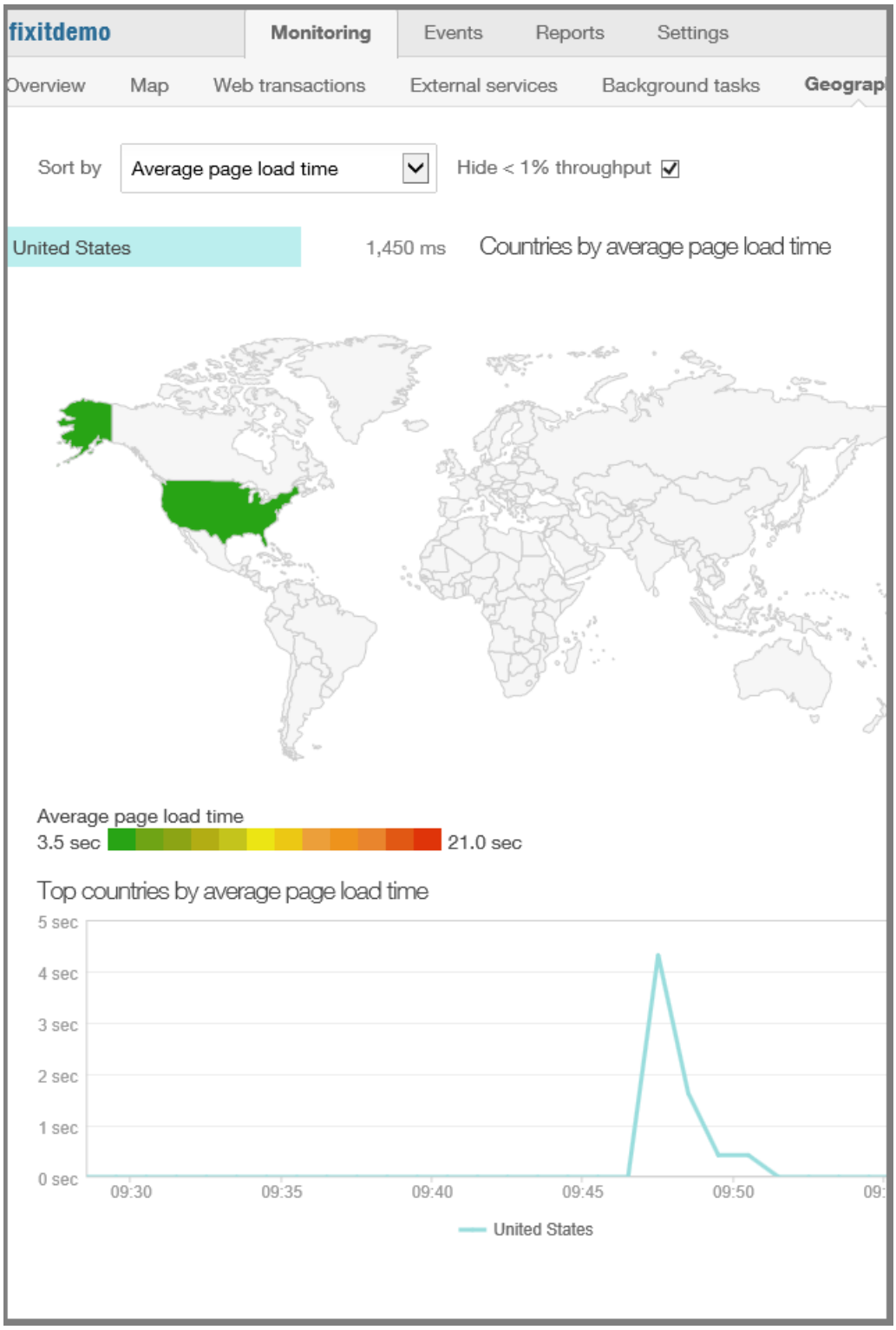
- Calls to external services such as the Blob service and statistics about how reliable and responsive the service has been.


fixitdemo
Monitoring
Events
Reports
Settings
Last 60 minut
Ending now
Overview
Map
Web transactions
External services
Background tasks
Geography
Browsers
End User
1.45 sec
fixitdemo
415 ms
0 servers
fixitdemostorage.blob
dows.net
7.8 ms
1.0 [7.0]*
< 1cpm
6 sec
4 sec
2 sec
0 sec
0.91 [0.5]*
1cpm
750
500
250
0
1 : 16.0 arr
< 1cpm
20 ms
15 ms
10 ms
5 ms
0 ms

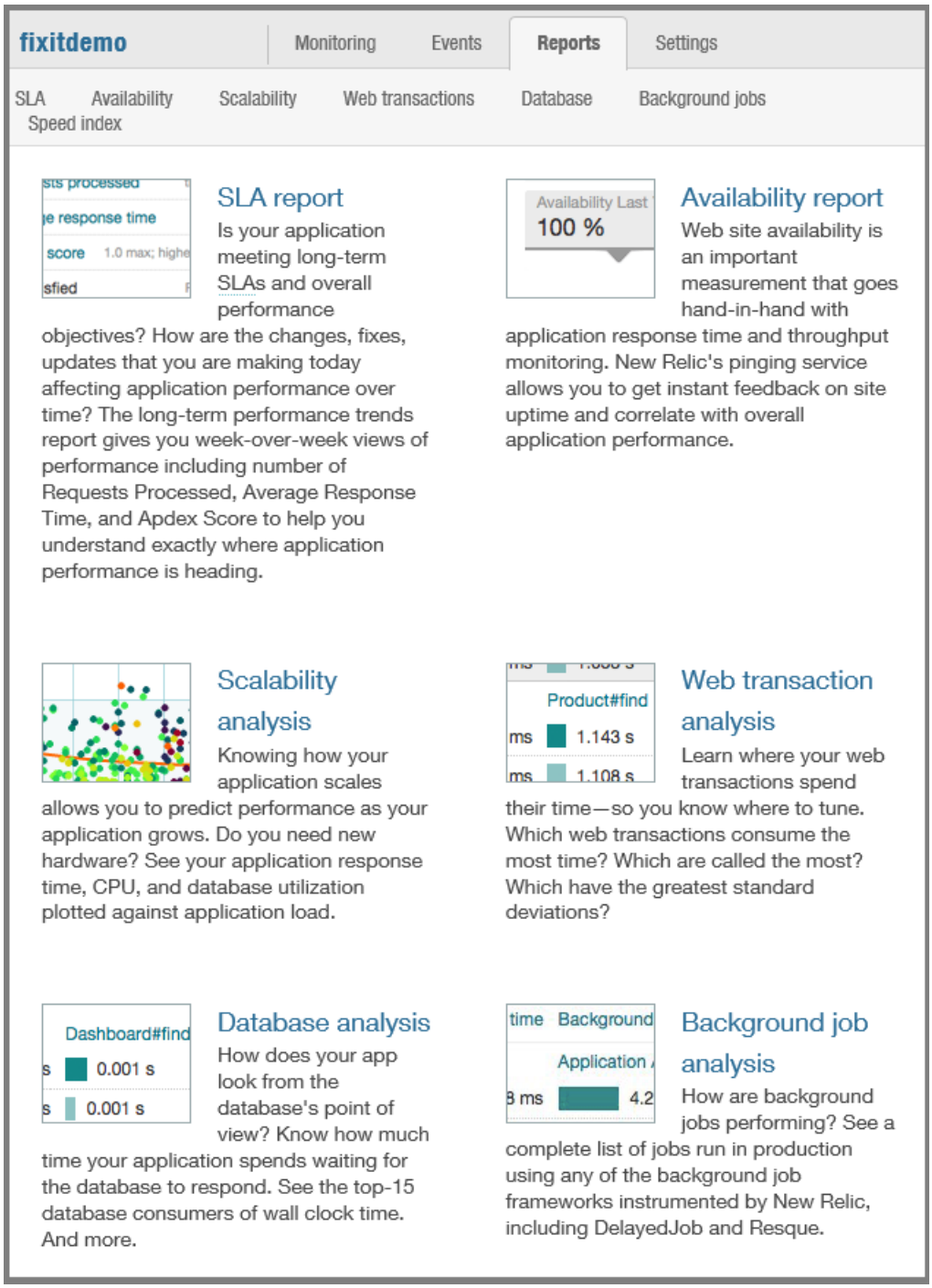

fixitdemo
Monitoring
Events
Reports
Settings
Overview
Map
Web transactions
External services
Background tasks
Geogra
Sort by
Most time consuming
fixitdemostorage.blob.core.windows.
100%
Top 5 external services
by total response time
20 ms
15 ms
10 ms
5 ms
0 ms
09:25
09:30
09:35
09:40
09:45
fixitdemostorage.blob.core.windows.net
External calls per minute
1 cpm
0.5 cpm
0 cpm
09:25
09:30
09:35
09:40
09:45
Web external



- Information about where in the world or where in the U.S. web app traffic came from.



You can also set up reports and events. For example, you can say any time you start seeing errors, send an email to alert support staff to the problem.



New Relic is just one example of a telemetry system; you can get all of this from other services as well. The beauty of the cloud is that without having to write any code, and for minimal or no expense, you suddenly can get a lot more information about how your application is being used and what your customers are actually experiencing.

Log for insight

A telemetry package is a good first step, but you still have to instrument your own code. The telemetry service tells you when there's a problem and tells you what the customers are experiencing, but it may not give you a lot of insight into what's going on in your code.

You don't want to have to remote into a production server to see what your app is doing. That might be practical when you've got one server, but what about when you've scaled to hundreds of servers and you don't know which ones you need to remote into? Your logging should provide enough information that you never have to remote into production servers to analyze and debug problems. You should be logging enough information so that you can isolate issues solely through the logs.

Log in production

A lot of people turn on tracing in production only when there's a problem and they want to debug. This can introduce a substantial delay between the time you're aware of a problem and the time you get useful troubleshooting information about it. And the information you get might not be helpful for intermittent errors.

What we recommend in the cloud environment where storage is cheap is that you always leave logging on in production. That way when errors happen you already have them logged, and you have historical data that can help you analyze issues that develop over time or happen regularly at different times. You could automate a purge process to delete old logs, but you might find that it's more expensive to set up such a process than it is to keep the logs.

The added expense of logging is trivial compared to the amount of troubleshooting time and money you can save by having all of the information you need already available when something goes wrong. Then when someone tells you they had a random error sometime around 8:00 last night, but they don't remember the error, you can readily find out what the problem was.

For less than $4 a month you can keep 50 gigabytes of logs on hand, and the performance impact of logging is trivial so long as you keep one thing in mind -- in order to avoid performance bottlenecks, make sure your logging library is asynchronous.

Differentiate logs that inform from logs that require action

Logs are meant to INFORM (I want you to know something) or ACT (I want you to do something). Be careful to only write ACT logs for issues that genuinely require a person or an automated process to take action. Too many ACT logs will create noise, requiring too much work to sift through it all to find genuine issues. And if your ACT logs automatically trigger some action such as sending email to support staff, avoid letting thousands of such actions be triggered by a single issue.

In .NET System.Diagnostics tracing, logs can be assigned Error, Warning, Info, and Debug/Verbose level. You can differentiate ACT from INFORM logs by reserving the Error level for ACT logs and using the lower levels for INFORM logs.

Level	Context
Error	Always on in production. Any errors will trigger ACTION to resolve (automated or human). • Configuration issues • Application failure (cascading failure or critical service down)
Warning	Always on in production. Warnings will INFORM, and may signal potential ACTION • Timeouts or throttling in external service
Info	Always on in production. Info messages INFORM during diagnostics and troubleshooting
Debug (Verbose)	On during active debugging and troubleshooting on a case by case basis

Configure logging levels at run time

While it's worthwhile to have logging always on in production, another best practice is to implement a logging framework that enables you to adjust at run-time the level of detail that you're logging, without redeploying or restarting your application. For example when you use the tracing facility in `System.Diagnostics` you can create Error, Warning, Info, and Debug/Verbose logs. We recommend you always log Error, Warning, and Info logs in production, and you'll want to be able to dynamically add Debug/Verbose logging for troubleshooting on a case-by-case basis.

Web Apps in Azure App Service have built-in support for writing `System.Diagnostics` logs to the file system, Table storage, or Blob storage. You can select different logging levels for each storage destination, and you can change the logging level on the fly without restarting your application. The Blob storage support makes it easier to run HDInsight analysis jobs on your application logs, because HDInsight knows how to work with Blob storage directly.

Log Exceptions

Don't just put *exception.ToString()* in your logging code. That leaves out contextual information. In the case of SQL errors, it leaves out the SQL error number. For all exceptions, include context information, the exception itself, and inner exceptions to make sure that you are providing everything that will be needed for troubleshooting. For example, context information might include the server name, a transaction identifier, and a user name (but not the password or any secrets!).

If you rely on each developer to do the right thing with exception logging, some of them won't. To ensure that it gets done the right way every time, build exception handling right into your logger interface: pass the exception object itself to the logger class and log the exception data properly in the logger class.

Log calls to services

We highly recommend that you write a log every time your app calls out to a service, whether it's to a database or a REST API or any external service. Include in your logs not only an indication of success or failure but how long each request took. In the cloud environment you'll often see problems related to slow-downs rather than complete outages. Something that normally takes 10 milliseconds might suddenly start taking a second. When someone tells you your app is slow, you want to be able to look at New Relic or whatever telemetry service you have and validate their experience, and then you want to be able to look are your own logs to dive into the details of why it's slow.

Use an ILogger interface

What we recommend doing when you create a production application is to create a simple *ILogger* interface and stick some methods in it. This makes it easy to change the logging implementation later and not have to go through all your code to do it. We could be using the `System.Diagnostics.Trace` class throughout the Fix It app, but instead we're using it under the covers in a logging class that implements *ILogger*, and we make *ILogger* method calls throughout the app.

That way, if you ever want to make your logging richer, you can replace `System.Diagnostics.Trace` with whatever logging mechanism you want. For example, as your app grows you might decide that you want to use a more comprehensive logging package such as NLog or Enterprise Library Logging Application Block. (Log4Net is another popular logging framework but it doesn't do asynchronous logging.)

One possible reason for using a framework such as NLog is to facilitate dividing up logging output into separate high-volume and high-value data stores. That helps you to efficiently store large volumes of INFORM data that you don't need to execute fast queries against, while maintaining quick access to ACT data.

Semantic Logging

For a relatively new way to do logging that can produce more useful diagnostic information, see Enterprise Library Semantic Logging Application Block (SLAB). SLAB uses Event Tracing for Windows (ETW) and EventSource support in .NET 4.5 to enable you to create more structured and queryable logs. You define a different method for each type of event that you log, which enables you to customize the information you write. For example, to log a SQL Database error you might call a `LogSQLDatabaseError` method. For that kind of exception, you know a key piece of information is the error number, so you could include an error number parameter in the method signature and record the error number as a separate field in the log record you write. Because the number is in a separate field you can more easily and reliably get reports based on SQL error numbers than you could if you were just concatenating the error number into a message string.

Logging in the Fix It app

The ILogger interface

Here is the *ILogger* interface in the Fix It app.

```
public interface ILogger
{
    void Information(string message);
    void Information(string fmt, params object[] vars);
    void Information(Exception exception, string fmt, params object[] vars);

    void Warning(string message);
    void Warning(string fmt, params object[] vars);
    void Warning(Exception exception, string fmt, params object[] vars);

    void Error(string message);
    void Error(string fmt, params object[] vars);
    void Error(Exception exception, string fmt, params object[] vars);

    void TraceApi(string componentName, string method, TimeSpan timespan);
    void TraceApi(string componentName, string method, TimeSpan timespan, string properties);
    void TraceApi(string componentName, string method, TimeSpan timespan, string fmt, params object[] vars);
}
```

These methods enable you to write logs at the same four levels supported by *System.Diagnostics*. The TraceApi methods are for logging external service calls with information about latency. You could also add a set of methods for Debug/Verbose level.

The Logger implementation of the ILogger interface

The implementation of the interface is really simple. It basically just calls into the standard *System.Diagnostics* methods. The following snippet shows all three of the Information methods and one each of the others.

```
public class Logger : ILogger
{
    public void Information(string message)
    {
        Trace.TraceInformation(message);
    }

    public void Information(string fmt, params object[] vars)
    {
        Trace.TraceInformation(fmt, vars);
    }

    public void Information(Exception exception, string fmt, params object[] vars)
    {
        var msg = String.Format(fmt, vars);
        Trace.TraceInformation(string.Format(fmt, vars) + ";Exception Details={0}", exception.ToString());
    }

    public void Warning(string message)
    {
        Trace.TraceWarning(message);
    }

    public void Error(string message)
    {
        Trace.TraceError(message);
    }

    public void TraceApi(string componentName, string method, TimeSpan timespan, string properties)
    {
        string message = String.Concat("component:", componentName, ";method:", method, ";timespan:",
timespan.ToString(), ";properties:", properties);
        Trace.TraceInformation(message);
    }
}
```

Calling the ILogger methods

Every time code in the Fix It app catches an exception, it calls an *ILogger* method to log the exception details. And every time it makes a call to the database, Blob service, or a REST API, it starts a stopwatch before the call, stops the stopwatch when the service returns, and logs the elapsed time along with information about success or failure.

Notice that the log message includes the class name and method name. It's a good practice to make sure that log messages identify what part of the application code wrote them.

```
public class FixItTaskRepository : IFixItTaskRepository
{
    private MyFixItContext db = new MyFixItContext();
    private ILogger log = null;

    public FixItTaskRepository(ILogger logger)
    {
        log = logger;
    }

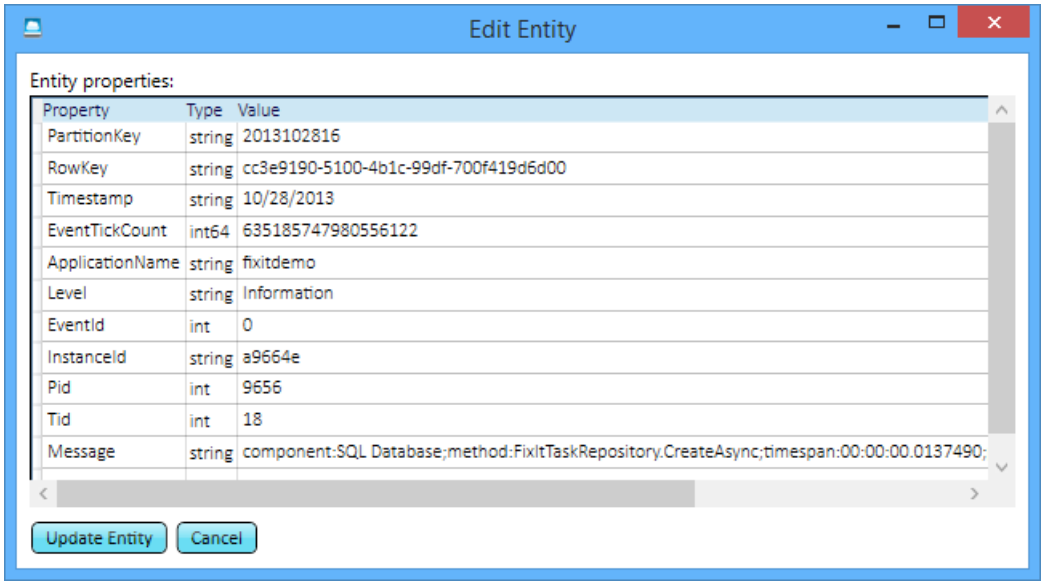
    public async Task<FixItTask> FindTaskByIdAsync(int id)
    {
        FixItTask fixItTask = null;
        Stopwatch timespan = Stopwatch.StartNew();

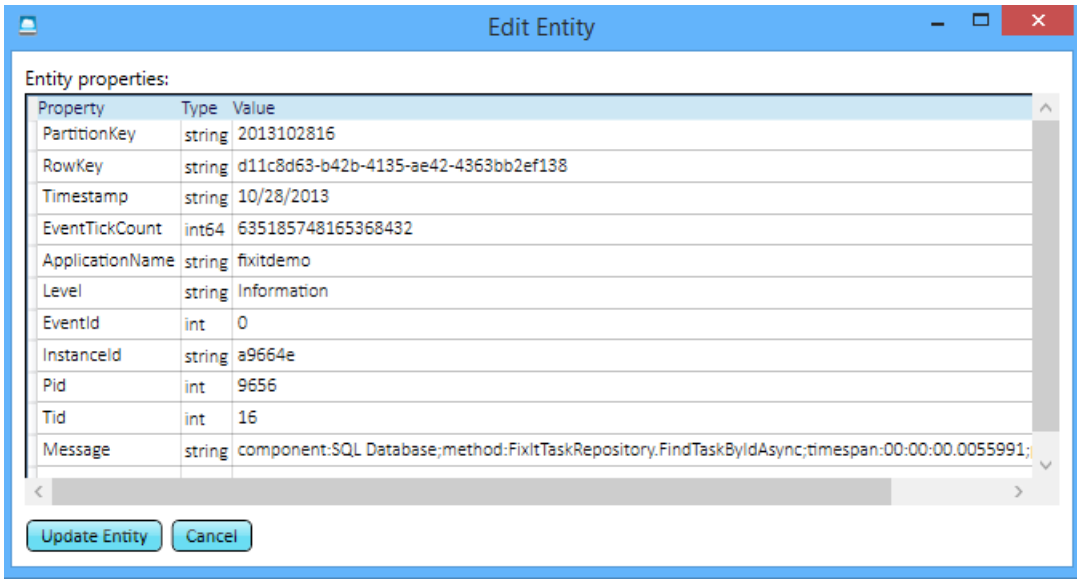
        try
        {
            fixItTask = await db.FixItTasks.FindAsync(id);

            timespan.Stop();
            log.TraceApi("SQL Database", "FixItTaskRepository.FindTaskByIdAsync", timespan.Elapsed, "id=
{0}", id);
        }
        catch(Exception e)
        {
            log.Error(e, "Error in FixItTaskRepository.FindTaskByIdAsynx(id={0})", id);
        }

        return fixItTask;
    }
```

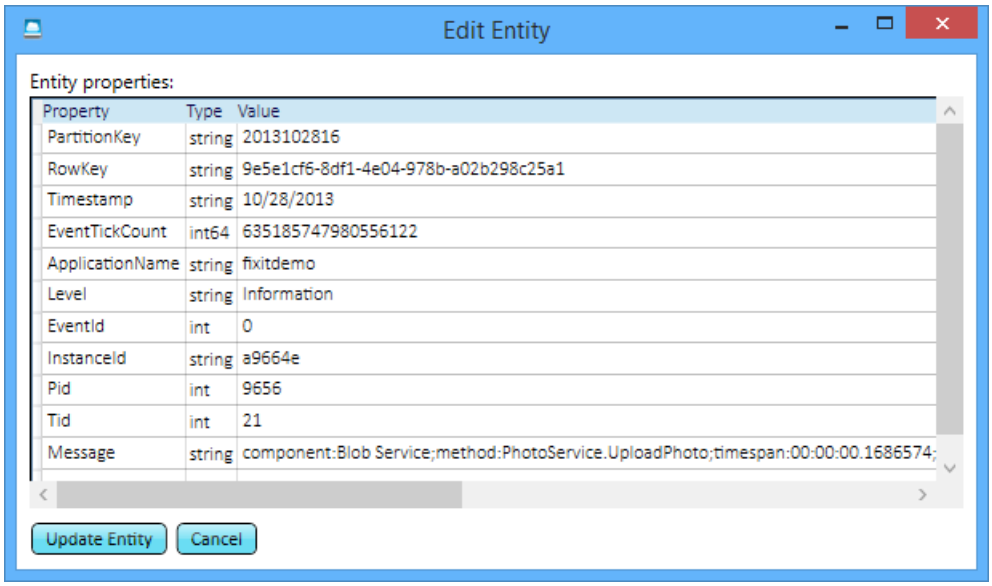
So now for every time the Fix It app has made a call to SQL Database, you can see the call, the method that called it, and exactly how much time it took.





If you go browsing through the logs, you can see that the time database calls take is variable. That information could be useful: because the app logs all this you can analyze historical trends in how the database service is performing over time. For instance, a service might be fast most of the time but requests might fail or responses might slow down at certain times of day.

You can do the same thing for the Blob service – for every time the app uploads a new file, there's a log, and you can see exactly how long it took to upload each file.



It's just a couple extra lines of code to write every time you call a service, and now whenever someone says they ran into an issue, you know exactly what the issue was, whether it was an error, or even if it was just running slow. You can pinpoint the source of the problem without having to remote into a server or turn on logging after the error happens and hope to re-create it.

Dependency Injection in the Fix It app

You might wonder how the repository constructor in the example shown above gets the logger interface implementation:

```
public class FixItTaskRepository : IFixItTaskRepository
{
    private MyFixItContext db = new MyFixItContext();
    private ILogger log = null;

    public FixItTaskRepository(ILogger logger)
    {
        log = logger;
    }
```

To wire up the interface to the implementation the app uses dependency injection(DI) with AutoFac. DI enables you to use an object based on an interface in many places throughout your code and only have to specify in one place the implementation that gets used when the interface is instantiated. This makes it easier to change the implementation: for example, you might want to replace the System.Diagnostics logger with an NLog logger. Or for automated testing you might want to substitute a mock version of the logger.

The Fix It application uses DI in all of the repositories and all of the controllers. The constructors of the controller classes get an *ITaskRepository* interface the same way the repository gets a logger interface:

```
public class DashboardController : Controller
{
    private IFixItTaskRepository fixItRepository = null;

    public DashboardController(IFixItTaskRepository repository)
    {
        fixItRepository = repository;
    }
```

The app uses the AutoFac DI library to automatically provide *TaskRepository* and *Logger* instances for these constructors.

```
public class DependenciesConfig
{
    public static void RegisterDependencies()
    {
        var builder = new ContainerBuilder();

        builder.RegisterControllers(typeof(MvcApplication).Assembly);
        builder.RegisterType<Logger>().As<ILogger>().SingleInstance();

        builder.RegisterType<FixItTaskRepository>().As<IFixItTaskRepository>();
        builder.RegisterType<PhotoService>().As<IPhotoService>().SingleInstance();
        builder.RegisterType<FixItQueueManager>().As<IFixItQueueManager>();

        var container = builder.Build();
        DependencyResolver.SetResolver(new AutofacDependencyResolver(container));
    }
}
```

This code basically says that anywhere a constructor needs an *ILogger* interface, pass in an instance of the *Logger* class, and whenever it needs an *IFixItTaskRepository* interface, pass in an instance of the *FixItTaskRepository* class.

AutoFac is one of many dependency injection frameworks that you can use. Another popular one is Unity, which is recommended and supported by Microsoft Patterns and Practices.

Built-in logging support in Azure

Azure supports the following kinds of logging for Web Apps in Azure App Service:

- System.Diagnostics tracing (you can turn on and off and set levels on the fly without restarting the site).
- Windows Events.
- IIS Logs (HTTP/FREB).

Azure supports the following kinds of logging in Cloud Services:

- System.Diagnostics tracing.
- Performance counters.
- Windows Events.
- IIS Logs (HTTP/FREB).
- Custom directory monitoring.

The Fix It app uses System.Diagnostics tracing. All you need to do to enable System.Diagnostics logging in an web app is flip a switch in the portal or call the REST API. In the portal, click the **Configuration** tab for your site, and scroll down to see the **Application Diagnostics** section. You can turn logging on or off and select the logging level you want. You can have Azure write the logs to the file system or to a storage account.

After you enable logging in Azure, you can see logs in the Visual Studio Output window as they are created.

You can also have logs written to your storage account and view them with any tool that can access the Azure Storage Table service, such as **Server Explorer** in Visual Studio or Azure Storage Explorer.

Summary

It's really simple to implement an out-of-the-box telemetry system, instrument logging in your own code, and configure logging in Azure. And when you have production issues, the combination of a telemetry system and custom logs will help you resolve problems quickly before they become major issues for your customers.

In the next chapter we'll look at how to handle transient errors so they don't become production issues that you have to investigate.

Resources

For more information, see the following resources.

Documentation mainly about telemetry:

- Microsoft Patterns and Practices - Azure Guidance. See Instrumentation and Telemetry guidance, Service Metering guidance, Health Endpoint Monitoring pattern, and Runtime Reconfiguration pattern.
- Penny Pinching in the Cloud: Enabling New Relic Performance Monitoring on Azure Websites.
- Best Practices for the Design of Large-Scale Services on Azure Cloud Services. White paper by Mark Simms and Michael Thomassy. See the Telemetry and Diagnostics section.
- Next-Generation Development with Application Insights. MSDN Magazine article.

Documentation mainly about logging:

- Semantic Logging Application Block (SLAB). Neil Mackenzie presents the case for semantic logging with SLAB.
- Creating Structured and Meaningful Logs with Semantic Logging. (Video) Julian Dominguez presents the case for semantic logging with SLAB.
- EF6 SQL Logging – Part 1: Simple Logging. Arthur Vickers shows how to log queries executed by Entity Framework in EF 6.
- Connection Resiliency and Command Interception with the Entity Framework in an ASP.NET MVC Application. Fourth in a nine-part tutorial series, shows how to use the EF 6 command interception feature to log SQL commands sent to the database by Entity Framework.
- Improve Logging using C# 5.0 Caller Info Attributes. How to easily log the name of the calling method without hard-coding it into literals or using reflection to get it manually.

Documentation mainly about troubleshooting:

- Azure Troubleshooting & Debugging blog.
- AzureTools – The Diagnostic Utility used by the Azure Developer Support Team. Introduces and provides a download link for a tool that can be used on an Azure VM to download and run a wide variety of diagnostic and monitoring tools. Useful when you need to diagnose an issue on a particular VM.
- Troubleshoot a web app in Azure App Service using Visual Studio. A step-by-step tutorial for getting started with System.Diagnostics tracing and remote debugging.

Videos:

- FailSafe: Building Scalable, Resilient Cloud Services. Nine-part series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents high-level concepts and architectural principles in a very accessible and interesting way, with stories drawn from Microsoft Customer Advisory Team (CAT) experience with actual customers. Episodes 4 and 9 are about monitoring and telemetry. Episode 9 includes an overview of monitoring services MetricsHub, AppDynamics, New Relic, and PagerDuty.
- Building Big: Lessons learned from Azure customers - Part II. Mark Simms talks about designing for failure and instrumenting everything. Similar to the Failsafe series but goes into more how-to details.

Code sample:

- Cloud Service Fundamentals in Azure. Sample application created by the Microsoft Azure Customer Advisory Team. Demonstrates both telemetry and logging practices, as explained in the following articles. The sample implements application logging by using NLog. For related documentation, see the series of four TechNet wiki articles about telemetry and logging.

PREVIOUS NEXT

Transient Fault Handling (Building Real-World Cloud Apps with Azure)

2/20/2020 • 7 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the e-book, see the first chapter.

When you're designing a real world cloud app, one of the things you have to think about is how to handle temporary service interruptions. This issue is uniquely important in cloud apps because you're so dependent on network connections and external services. You can frequently get little glitches that are typically self-healing, and if you aren't prepared to handle them intelligently, they'll result in a bad experience for your customers.

Causes of transient failures

In the cloud environment you'll find that failed and dropped database connections happen periodically. That's partly because you're going through more load balancers compared to the on-premises environment where your web server and database server have a direct physical connection. Also, sometimes when you're dependent on a multi-tenant service you'll see calls to the service get slower or time out because someone else who uses the service is hitting it heavily. In other cases you might be the user who is hitting the service too frequently, and the service deliberately throttles you – denies connections – in order to prevent you from adversely affecting other tenants of the service.

Use smart retry/back-off logic to mitigate the effect of transient failures

Instead of throwing an exception and displaying a not available or error page to your customer, you can recognize errors that are typically transient, and automatically retry the operation that resulted in the error, in hopes that before long you'll be successful. Most of the time the operation will succeed on the second try, and you'll recover from the error without the customer ever having been aware that there was a problem.

There are several ways you can implement smart retry logic.

- The Microsoft Patterns & Practices group has a Transient Fault Handling Application Block that does everything for you if you're using ADO.NET for SQL Database access (not through Entity Framework). You just set a policy for retries – how many times to retry a query or command and how long to wait between tries – and wrap your SQL code in a *using* block.

```
public void HandleTransients()
{
    var connStr = "some database";
    var _policy = RetryPolicy.Create < SqlAzureTransientErrorDetectionStrategy(
        retryCount: 3,
        retryInterval: TimeSpan.FromSeconds(5));

    using (var conn = new ReliableSqlConnection(connStr, _policy))
    {
        // Do SQL stuff here.
    }
}
```

TFH also supports Azure In-Role Cache and Service Bus.

- When you use the Entity Framework you typically aren't working directly with SQL connections, so you can't use this Patterns and Practices package, but Entity Framework 6 builds this kind of retry logic right into the framework. In a similar way you specify the retry strategy, and then EF uses that strategy whenever it accesses the database.

  To use this feature in the Fix It app, all we have to do is add a class that derives from *DbConfiguration* and turn on the retry logic.

```
// EF follows a Code based Configuration model and will look for a class that
// derives from DbConfiguration for executing any Connection Resiliency strategies
public class EFConfiguration : DbConfiguration
{
    public EFConfiguration()
    {
        AddExecutionStrategy(() => new SqlAzureExecutionStrategy());
    }
}
```

  For SQL Database exceptions that the framework identifies as typically transient errors, the code shown instructs EF to retry the operation up to 3 times, with an exponential back-off delay between retries, and a maximum delay of 5 seconds. Exponential back-off means that after each failed retry it will wait for a longer period of time before trying again. If three tries in a row fail, it will throw an exception. The following section about circuit breakers explains why you want exponential back-off and a limited number of retries.

  You can have similar issues when you're using the Azure Storage service, as the Fix It app does for Blobs, and the .NET storage client API already implements the same kind of logic. You just specify the retry policy, or you don't even have to do that if you're happy with the default settings.

Circuit breakers

There are several reasons why you don't want to retry too many times over too long a period:

- Too many users persistently retrying failed requests might degrade other users' experience. If millions of people are all making repeated retry requests you could be tying up IIS dispatch queues and preventing your app from servicing requests that it otherwise could handle successfully.
- If everyone is retrying due to a service failure, there could be so many requests queued up that the service gets flooded when it starts to recover.
- If the error is due to throttling and there's a window of time the service uses for throttling, continued retries could move that window out and cause the throttling to continue.
- You might have a user waiting for a web page to render. Making people wait too long might be more annoying that relatively quickly advising them to try again later.

Exponential back-off addresses some of these issue by limiting the frequency of retries a service can get from your application. But you also need to have *circuit breakers*: this means that at a certain retry threshold your app stops retrying and takes some other action, such as one of the following:

- Custom fallback. If you can't get a stock price from Reuters, maybe you can get it from Bloomberg; or if you can't get data from the database, maybe you can get it from cache.
- Fail silent. If what you need from a service isn't all-or-nothing for your app, just return null when you can't get the data. If you're displaying a Fix It task and the Blob service isn't responding, you could display the task details without the image.
- Fail fast. Error out the user to avoid flooding the service with retry requests which could cause service disruption for other users or extend a throttling window. You can display a friendly "try again later" message.

There is no one-size-fits-all retry policy. You can retry more times and wait longer in an asynchronous background worker process than you would in a synchronous web app where a user is waiting for a response. You can wait longer between retries for a relational database service than you would for a cache service. Here are some sample recommended retry policies to give you an idea of how the numbers might vary. ("Fast First" means no delay before the first retry.

Platform	Context	Sample Target e2e latency max	Fast First	Retry Count	Delay	Backoff
Windows Azure Table Storage	Synchronous (e.g. render web page)	100 ms	Yes	3	25 ms	Linear
	Asynchronous (e.g. process queue item)	500 ms	Yes	3	100 ms	Exponential

For SQL Database retry policy guidance, see Troubleshoot transient faults and connection errors to SQL Database.

Summary

A retry/back-off strategy can help make temporary errors invisible to the customer most of the time, and Microsoft provides frameworks that you can use to minimize your work implementing a strategy whether you're using ADO.NET, Entity Framework, or the Azure Storage service.

In the next chapter, we'll look at how to improve performance and reliability by using distributed caching.

Resources

For more information, see the following resources:

Documentation

- Best Practices for the Design of Large-Scale Services on Azure Cloud Services. White paper by Mark Simms and Michael Thomassy. Similar to the Failsafe series but goes into more how-to details. See the Telemetry and Diagnostics section.
- Failsafe: Guidance for Resilient Cloud Architectures. White paper by Marc Mercuri, Ulrich Homann, and Andrew Townhill. Web page version of the FailSafe video series.
- Microsoft Patterns and Practices - Azure Guidance. See Retry pattern, Scheduler Agent Supervisor pattern.
- Fault-tolerance in Azure SQL Database. Blog post by Tony Petrossian.
- Entity Framework - Connection Resiliency / Retry Logic. How to use and customize the transient fault handling feature of Entity Framework 6.
- Connection Resiliency and Command Interception with the Entity Framework in an ASP.NET MVC Application.

Fourth in a nine-part tutorial series, shows how to set up the EF 6 connection resilience feature for SQL Database.

Videos

- FailSafe: Building Scalable, Resilient Cloud Services. Nine-part series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents high-level concepts and architectural principles in a very accessible and interesting way, with stories drawn from Microsoft Customer Advisory Team (CAT) experience with actual customers. See the discussion of circuit breakers in episode 3 starting at 40:55.
- Building Big: Lessons learned from Azure customers - Part II. Mark Simms talks about designing for failure, transient fault handling, and instrumenting everything.

Code sample

- Cloud Service Fundamentals in Azure. Sample application created by the Microsoft Azure Customer Advisory Team that demonstrates how to use the Enterprise Library Transient Fault Handling Block (TFH). For more information, see Cloud Service Fundamentals Data Access Layer – Transient Fault Handling. TFH is recommended for database access using ADO.NET directly (without using Entity Framework).

PREVIOUS NEXT

Distributed Caching (Building Real-World Cloud Apps with Azure)

2/20/2020 • 7 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

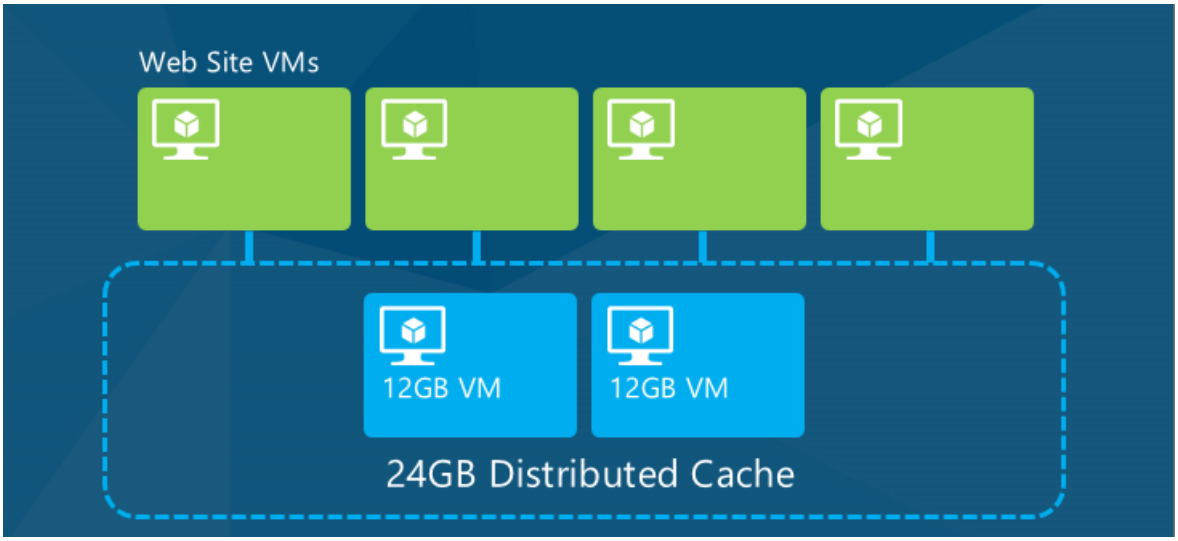
Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the e-book, see the first chapter.

The previous chapter looked at transient fault handling and mentioned caching as a circuit breaker strategy. This chapter gives more background about caching, including when to use it, common patterns for using it, and how to implement it in Azure.

What is distributed caching

A cache provides high throughput, low-latency access to commonly accessed application data, by storing the data in memory. For a cloud app the most useful type of cache is distributed cache, which means the data is not stored on the individual web server's memory but on other cloud resources, and the cached data is made available to all of an application's web servers (or other cloud VMs that are used by the application).



When the application scales by adding or removing servers, or when servers are replaced due to upgrades or faults, the cached data remains accessible to every server that runs the application.

By avoiding the high latency data access of a persistent data store, caching can dramatically improve application responsiveness. For example, retrieving data from cache is much faster than retrieving it from a relational database.

A side benefit of caching is reduced traffic to the persistent data store, which may result in lower costs when there are data egress charges for the persistent data store.

When to use distributed caching

Caching works best for application workloads that do more reading than writing of data, and when the data

model supports the key/value organization that you use to store and retrieve data in cache. It's also more useful when application users share a lot of common data; for example, cache would not provide as many benefits if each user typically retrieves data unique to that user. An example where caching could be very beneficial is a product catalog, because the data does not change frequently, and all customers are looking at the same data.

The benefit of caching becomes increasingly measurable the more an application scales, as the throughput limits and latency delays of the persistent data store become more of a limit on overall application performance. However, you might implement caching for other reasons than performance as well. For data that doesn't have to be perfectly up-to-date when shown to the user, cache access can serve as a circuit breaker for when the persistent data store is unresponsive or unavailable.

Popular cache population strategies

In order to be able to retrieve data from cache, you have to store it there first. There are several strategies for getting data that you need into a cache:

- On Demand / Cache Aside

  The application tries to retrieve data from cache, and when the cache doesn't have the data (a "miss"), the application stores the data in the cache so that it will be available the next time. The next time the application tries to get the same data, it finds what it's looking for in the cache (a "hit"). To prevent fetching cached data that has changed on the database, you invalidate the cache when making changes to the data store.

- Background Data Push

  Background services push data into the cache on a regular schedule, and the app always pulls from the cache. This approach works great with high latency data sources that don't require you always return the latest data.

- Circuit Breaker

  The application normally communicates directly with the persistent data store, but when the persistent data store has availability problems, the application retrieves data from cache. Data may have been put in cache using either the cache aside or background data push strategy. This is a fault handling strategy rather than a performance enhancing strategy.

In order to keep data in the cache current, you can delete related cache entries when your application creates, updates, or deletes data. If it's alright for your application to sometimes get data that is slightly out-of-date, you can rely on a configurable expiration time to set a limit on how old cache data can be.

You can configure absolute expiration (amount of time since the cache item was created) or sliding expiration (amount of time since the last time a cache item was accessed). Absolute expiration is used when you are depending on the cache expiration mechanism to prevent the data from becoming too stale. In the Fix It app, we'll manually evict stale cache items and we'll use sliding expiration to keep the most current data in cache. Regardless of the expiration policy you choose, the cache will automatically evict the oldest (Least Recently Used or LRU) items when the cache's memory limit is reached.

Sample cache-aside code for Fix It app

In the following sample code, we check the cache first when retrieving a Fix It task. If the task is found in cache, we return it; if not found, we get it from the database and store it in the cache. The changes you'd make to add caching to the `FindTaskByIdAsync` method are highlighted.

```
public async Task<FixItTask> FindTaskByIdAsync(int id)
 {
    FixItTask fixItTask = null;
    Stopwatch timespan = Stopwatch.StartNew();
    string hitMiss = "Hit";

    try
    {
       fixItTask = (FixItTask)cache.Get(id.ToString());
       if (fixItTask == null)
       {
          fixItTask = await db.FixItTasks.FindAsync(id);
          cache.Put(id.ToString(), fixItTask);
          hitMiss = "Miss";
       }

       timespan.Stop();
       log.TraceApi("SQL Database", "FixItTaskRepository.FindTaskByIdAsync", timespan.Elapsed,
                    "cache {0}, id={1}", hitMiss, id);
    }
    catch (Exception e)
    {
       log.Error(e, "Error in FixItTaskRepository.FindTaskByIdAsynx(id={0})", id);
    }

    return fixItTask;
 }
```

When you update or delete a Fix It task, you have to invalidate (remove) the cached task. Otherwise, future attempts to read that task will continue to get the old data from the cache.

```
public async Task UpdateAsync(FixItTask taskToSave)
{
   Stopwatch timespan = Stopwatch.StartNew();

   try
   {
      cache.Remove(taskToSave.FixItTaskId.ToString());
      db.Entry(taskToSave).State = EntityState.Modified;
      await db.SaveChangesAsync();

      timespan.Stop();
      log.TraceApi("SQL Database", "FixItTaskRepository.UpdateAsync", timespan.Elapsed, "taskToSave={0}",
taskToSave);
   }
   catch (Exception e)
   {
      log.Error(e, "Error in FixItTaskRepository.UpdateAsync(taskToSave={0})", taskToSave);
   }
}
```

These are samples to illustrate simple caching code; caching has not been implemented in the downloadable Fix It project.

Azure caching services

Azure offers the following caching services: Azure Redis Cache and Azure Managed Cache. Azure Redis cache is based on the popular open source Redis Cache and is the first choice for most caching scenarios.

ASP.NET session state using a cache provider

As mentioned in the web development best practices chapter, a best practice is to avoid using session state. If

your application requires session state, the next best practice is to avoid the default in-memory provider because that doesn't enable scale out (multiple instances of the web server). The ASP.NET SQL Server session state provider enables a site that runs on multiple web servers to use session state, but it incurs a high latency cost compared to an in-memory provider. The best solution if you have to use session state is to use a cache provider, such as the Session State Provider for Azure Cache.

Summary

You've seen how the Fix It app could implement caching in order to improve response time and scalability, and to enable the app to continue to be responsive for read operations when the database is unavailable. In the next chapter we'll show how to further improve scalability and make the app continue to be responsive for write operations.

Resources

For more information about caching, see the following resources.

Documentation

- Azure Cache. Official MSDN documentation on caching in Azure.
- Microsoft Patterns and Practices - Azure Guidance. See Caching guidance and Cache-Aside pattern.
- Failsafe: Guidance for Resilient Cloud Architectures. White paper by Marc Mercuri, Ulrich Homann, and Andrew Townhill. See the section on Caching.
- Best Practices for the Design of Large-Scale Services on Azure Cloud Services. W. White paper by Mark Simms and Michael Thomassy. See the section on distributed caching.
- Distributed Caching On The Path To Scalability. An older (2009) MSDN Magazine article, but a clearly written introduction to distributed caching in general; goes into more depth than the caching sections of the FailSafe and Best Practices white papers.

Videos

- FailSafe: Building Scalable, Resilient Cloud Services. Nine-part series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents a 400-level view of how to architect cloud apps. This series focuses on theory and reasons why; for more how-to details, see the Building Big series by Mark Simms. See the caching discussion in episode 3 starting at 1:24:14.
- Building Big: Lessons learned from Azure customers - Part I. Simon Davies discusses distributed caching starting at 46:00. Similar to the Failsafe series but goes into more how-to details. The presentation was given October 31, 2012, so it does not cover caching service of Web Apps in Azure App Service that was introduced in 2013.

Code sample

- Cloud Service Fundamentals in Azure. Sample application that implements distributed caching. See the accompanying blog post Cloud Service Fundamentals – Caching Basics.

Queue-Centric Work Pattern (Building Real-World Cloud Apps with Azure)

2/20/2020 • 13 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the e-book, see the first chapter.

Earlier, we saw that using multiple services can result in a "composite" SLA, where the app's effective SLA is the *product* of the individual SLAs. For example, the Fix It app uses Web Sites, Storage, and SQL Database. If any one of these services fails, the app will return an error to the user.

Caching is a good way to handle transient failures for read-only content. But what if your application needs to do work? For example, when the user submits a new Fix It task, the app can't just put the task into the cache. The app needs to write the Fix It task into a persistent data store, so it can be processed.

That's where the queue-centric work pattern comes in. This pattern enables loose coupling between a web tier and a backend service.

Here's how the pattern works. When the application gets a request, it puts a work item onto a queue and immediately returns the response. Then a separate backend process pulls work items from the queue and does the work.

The queue-centric work pattern is useful for:

- Work that is time consuming (high latency).
- Work that requires an external service that might not always be available.
- Work that is resource-intensive (high CPU).
- Work that would benefit from rate leveling (subject to sudden load bursts).

Reduced Latency

Queues are useful any time you are doing time-consuming work. If a task takes a few seconds or longer, instead of blocking the end user, put the work item into a queue. Tell the user "We're working on it," and then use a queue listener to process the task in the background.

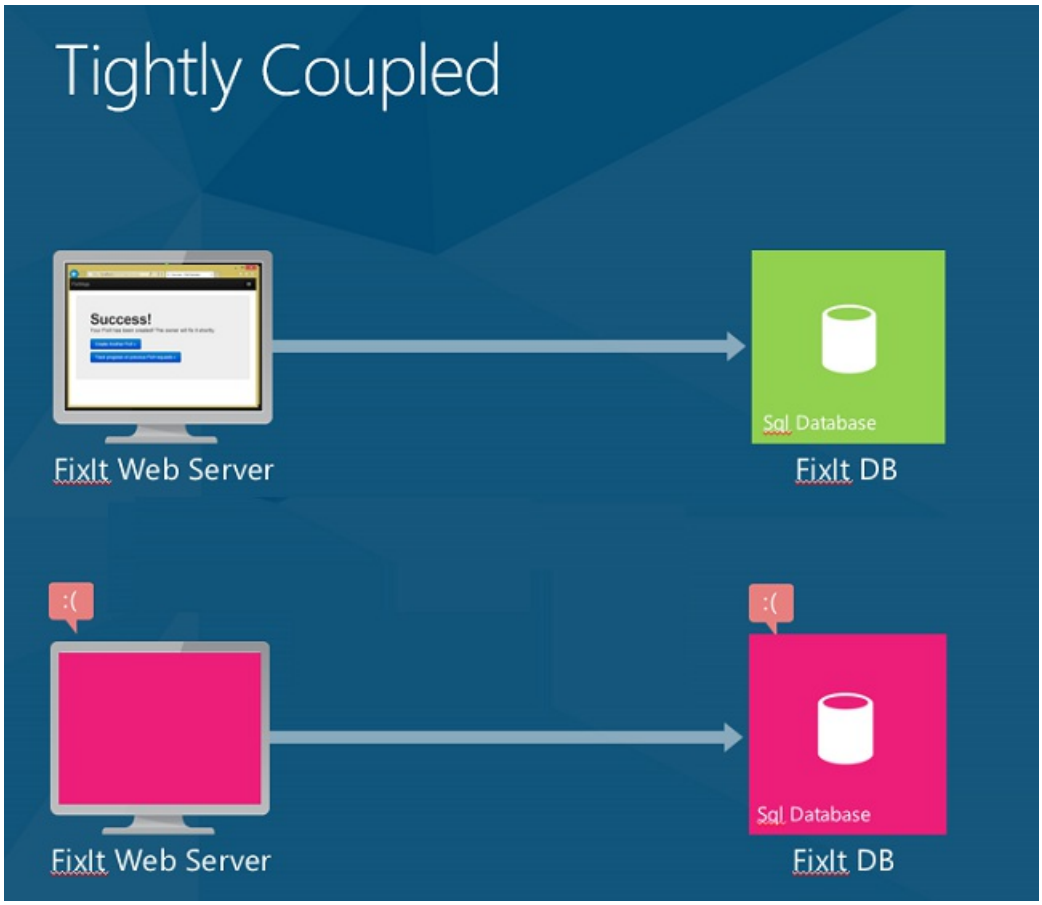
For example, when you purchase something at an online retailer, the web site confirms your order immediately. But that doesn't mean your stuff is already in a truck being delivered. They put a task in a queue, and in the background they are doing the credit check, preparing your items for shipping, and so forth.

For scenarios with short latency, the total end-to-end time might be longer using a queue, compared with doing the task synchronously. But even then, the other benefits can outweigh that disadvantage.

Increased Reliability

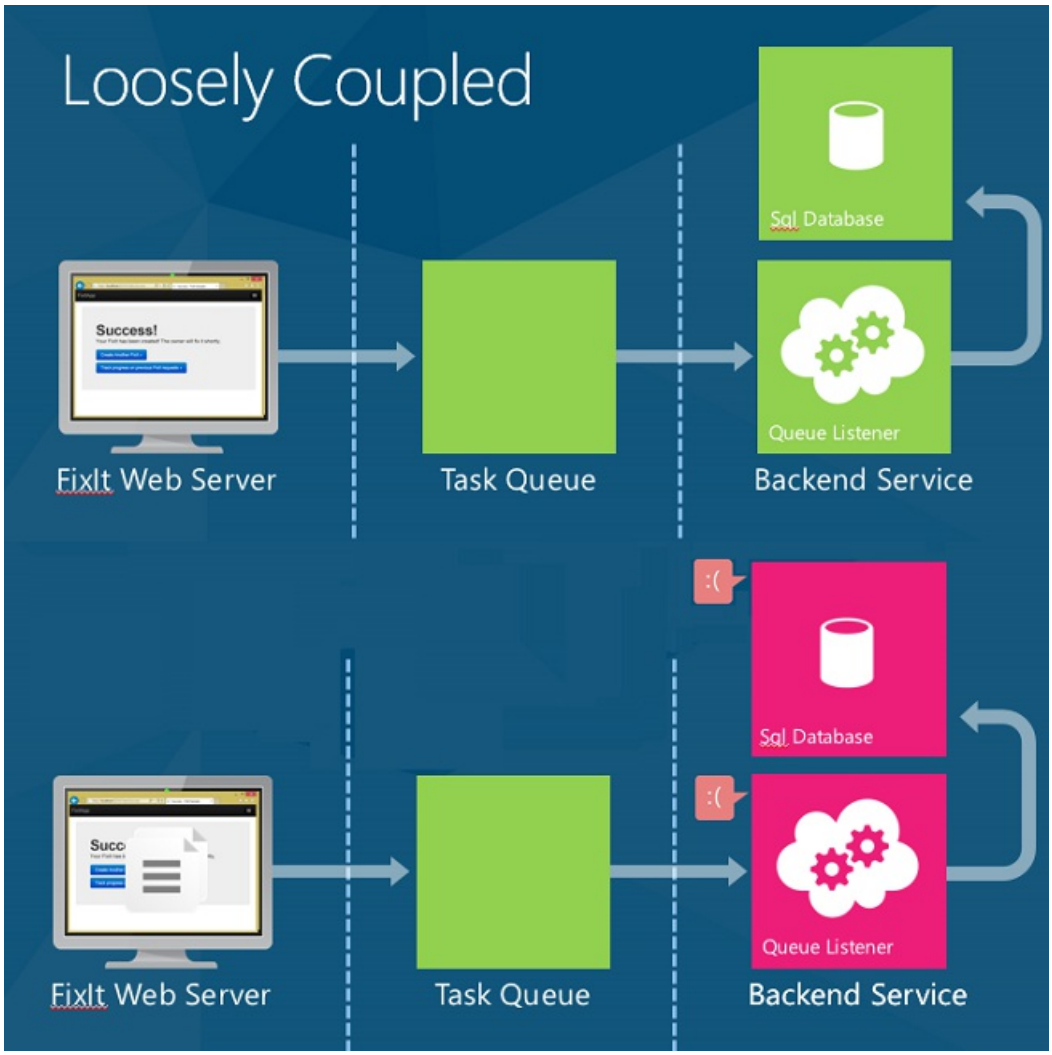
In the version of Fix It that we've been looking at so far, the web front-end is tightly coupled with the SQL Database back-end. If the SQL database service is unavailable, the user gets an error. If retries don't work (that is,

the failure is more than transient), the only thing you can do is show an error and ask the user to try again later.



Using queues, when a user submits a Fix It task, the app writes a message to the queue. The message payload is a JSON representation of the task. As soon as the message is written to the queue, the app returns and immediately shows a success message to the user.

If any of the backend services – such as the SQL database or the queue listener -- go offline, users can still submit new Fix It tasks. The messages will just queue up until the backend services are available again. At that point, the backend services will catch up on the backlog.



Moreover, now you can add more backend logic without worrying about the resiliency of the front end. For example, you might want to send an email or SMS message to the owner whenever a new Fix It is assigned. If the email or SMS service becomes unavailable, you can process everything else, and then put a message into a separate queue for sending email/SMS messages.

Previously, our effective SLA was Web Apps × Storage × SQL Database = 99.7%. (See Design to Survive Failures.)

When we change the app to use a queue, the web front end depends only on Web Apps and Storage, for a composite SLA of 99.8%. (Note that queues are part of the Azure storage service, so they are included in the same SLA as blob storage.)

If you need even better than 99.8%, you can create two queues in two different regions. Designate one as the primary, and the other as the secondary. In your app, fail over to the secondary queue if the primary queue is not available. The chance of both being unavailable at the same time is very small.

Rate Leveling and Independent Scaling

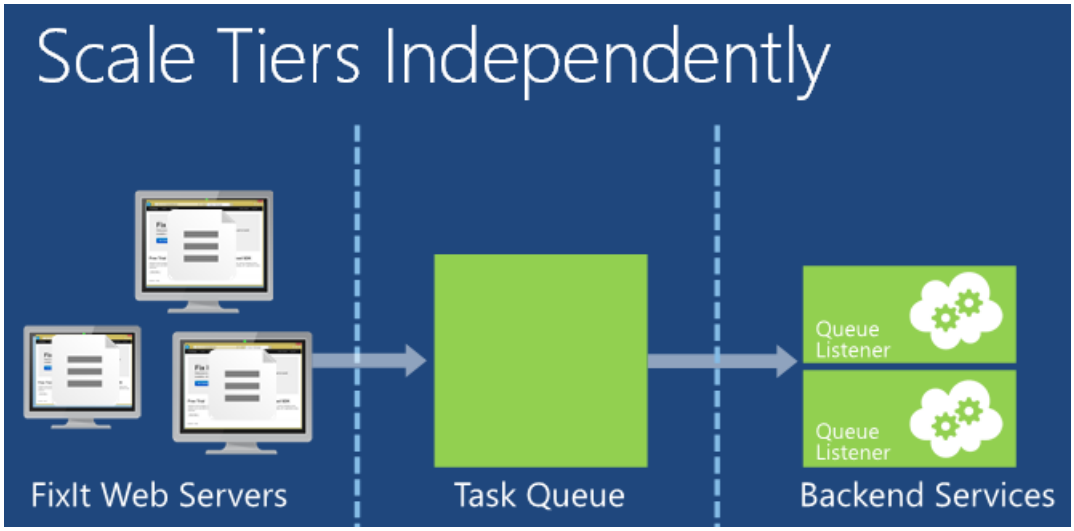
Queues are also useful for something called *rate leveling* or *load leveling*.

Web apps are often susceptible to sudden bursts in traffic. While you can use autoscaling to automatically add web servers to handle increased web traffic, autoscaling might not be able to react quickly enough to handle abrupt spikes in load. If the web servers can offload some of the work they have to do by writing a message to a queue, they can handle more traffic. A backend service can then read messages from the queue and process them. The depth of the queue will grow or shrink as the incoming load varies.

With much of its time-consuming work off-loaded to a backend service, the web tier can more easily respond to sudden spikes in traffic. And you save money because any given amount of traffic can be handled by fewer web

servers.

You can scale the web tier and backend service independently. For example, you might need three web servers but only one server processing queue messages. Or if you're running a compute-intensive task in the background, you might need more backend servers.



Autoscaling works with backend services as well as with the web tier. You can scale up or scale down the number of VMs that are processing the tasks in the queue, based on the CPU usage of the backend VMs. Or, you can autoscale based on how many items are in a queue. For example, you can tell autoscale to try to keep no more than 10 items in the queue. If the queue has more than 10 items, autoscale will add VMs. When they catch up, autoscale will tear down the extra VMs.

Adding Queues to the Fix It Application

To implement the queue pattern, we need to make two changes to the Fix It app.

- When a user submits a new Fix It task, put the task in the queue, instead of writing it to the database.
- Create a back-end service that processes messages in the queue.

For the queue, we'll use the Azure Queue Storage Service. Another option is to use Azure Service Bus.

To decide which queue service to use, consider how your app needs to send and receive the messages in the queue:

- If you have cooperating producers and competing consumers, consider using Azure Queue Storage Service. "Cooperating producers" means multiple processes are adding messages to a queue. "Competing consumers" means multiple processes are pulling messages off the queue to process them, but any given message can only be processed by one "consumer." If you need more throughput than you can get with a single queue, use additional queues and/or additional storage accounts.
- If you need a publish/subscribe model, consider using Azure Service Bus Queues.

The Fix It app fits the cooperating producers and competing consumers model.

Another consideration is application availability. The Queue Storage Service is part of the same service that we're using for blob storage, so using it has no effect on our SLA. Azure Service Bus is a separate service with its own SLA. If we used Service Bus Queues, we would have to factor in an additional SLA percentage, and our composite SLA would be lower. When you're choosing a queue service, make sure you understand the impact of your choice on application availability. For more information, see the Resources section.

Creating Queue Messages

To put a Fix It task on the queue, the web front end performs the following steps:

1. Create a CloudQueueClient instance. The `CloudQueueClient` instance is used to execute requests against the Queue Service.
2. Create the queue, if it doesn't exist yet.
3. Serialize the Fix It task.
4. Call CloudQueue.AddMessageAsync to put the message onto the queue.

We'll do this work in the constructor and `SendMessageAsync` method of a new `FixItQueueManager` class.

```
public class FixItQueueManager : IFixItQueueManager
{
    private CloudQueueClient _queueClient;
    private IFixItTaskRepository _repository;

    private static readonly string fixitQueueName = "fixits";

    public FixItQueueManager(IFixItTaskRepository repository)
    {
        _repository = repository;
        CloudStorageAccount storageAccount = StorageUtils.StorageAccount;
        _queueClient = storageAccount.CreateCloudQueueClient();
    }

    // Puts a serialized fixit onto the queue.
    public async Task SendMessageAsync(FixItTask fixIt)
    {
        CloudQueue queue = _queueClient.GetQueueReference(fixitQueueName);
        await queue.CreateIfNotExistsAsync();

        var fixitJson = JsonConvert.SerializeObject(fixIt);
        CloudQueueMessage message = new CloudQueueMessage(fixitJson);

        await queue.AddMessageAsync(message);
    }

    // Processes any messages on the queue.
    public async Task ProcessMessagesAsync()
    {
        CloudQueue queue = _queueClient.GetQueueReference(fixitQueueName);
        await queue.CreateIfNotExistsAsync();

        while (true)
        {
            CloudQueueMessage message = await queue.GetMessageAsync();
            if (message == null)
            {
                break;
            }
            FixItTask fixit = JsonConvert.DeserializeObject<FixItTask>(message.AsString);
            await _repository.CreateAsync(fixit);
            await queue.DeleteMessageAsync(message);
        }
    }
}
```

Here we are using the Json.NET library to serialize the fixit to JSON format. You can use whatever serialization approach you prefer. JSON has the advantage of being human-readable, while being less verbose than XML.

Production-quality code would add error handling logic, pause if the database became unavailable, handle recovery more cleanly, create the queue on application start-up, and manage "poison" messages. (A poison message is a message that cannot be processed for some reason. You don't want poison messages to sit in the queue, where the worker role will continually try to process them, fail, try again, fail, and so on.)

In the front-end MVC application, we need to update the code that creates a new task. Instead of putting the task

into the repository, call the `SendMessageAsync` method shown above.

```
public async Task<ActionResult> Create(FixItTask fixittask, HttpPostedFileBase photo)
{
    if (ModelState.IsValid)
    {
        fixittask.CreatedBy = User.Identity.Name;
        fixittask.PhotoUrl = await photoService.UploadPhotoAsync(photo);
        //previous code:
        //await fixItRepository.CreateAsync(fixittask);
        //new code:
        await queueManager.SendMessageAsync(fixittask);
        return RedirectToAction("Success");
    }
    return View(fixittask);
}
```

Processing Queue Messages

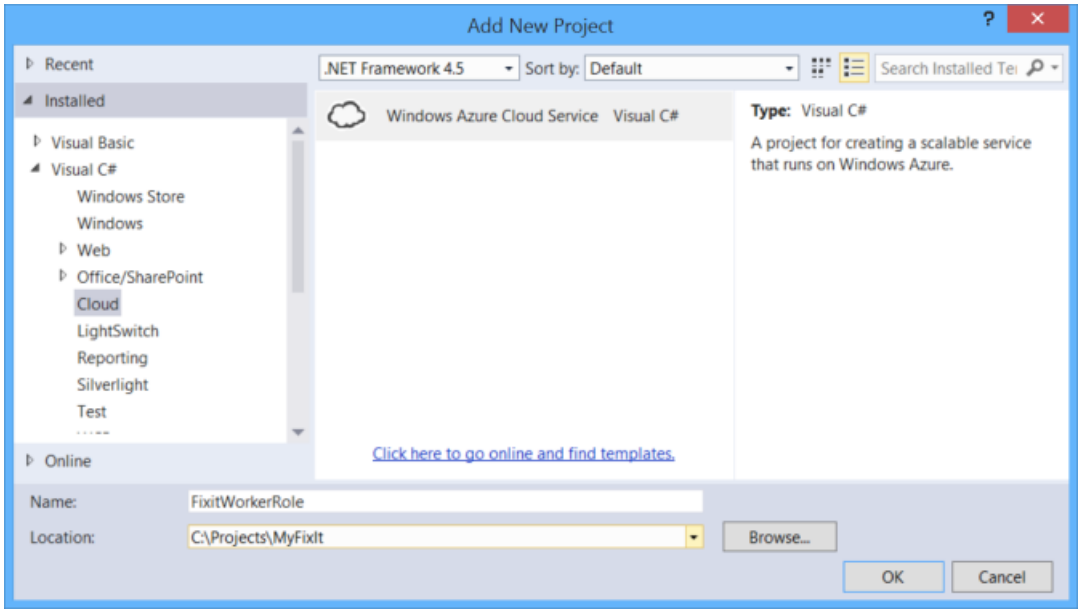
To process messages in the queue, we'll create a backend service. The backend service will run an infinite loop that performs the following steps:

1. Get the next message from the queue.
2. Deserialize the message to a Fix It task.
3. Write the Fix It task to the database.

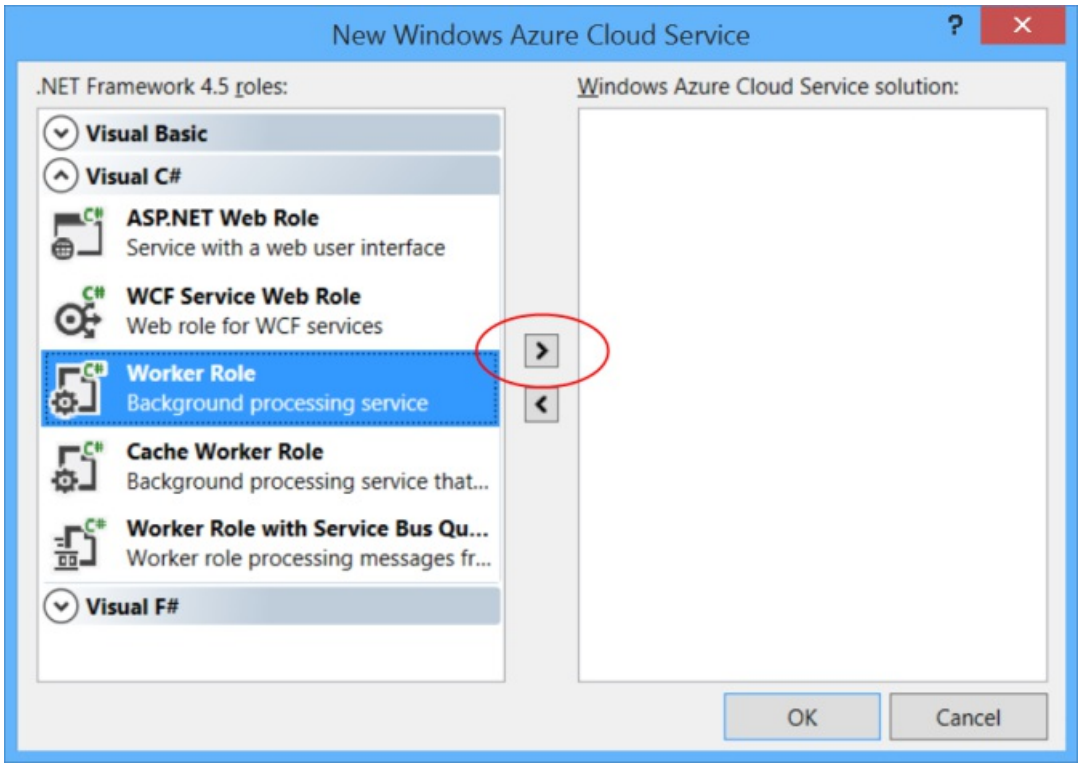
To host the backend service, we'll create an Azure Cloud Service that contains a *worker role*. A worker role consists of one or more VMs that can do backend processing. The code that runs in these VMs will pull messages from the queue as they become available. For each message, we'll deserialize the JSON payload and write an instance of the Fix It Task entity to the database, using the same repository that we used earlier in the web tier.

The following steps show how to add a worker role project to a solution that has a standard web project. These steps have already been done in the Fix It project that you can download.

First add a Cloud Service project to the Visual Studio solution. Right-click the solution and select **Add**, then **New Project**. In the left pane, expand **Visual C#** and select **Cloud**.

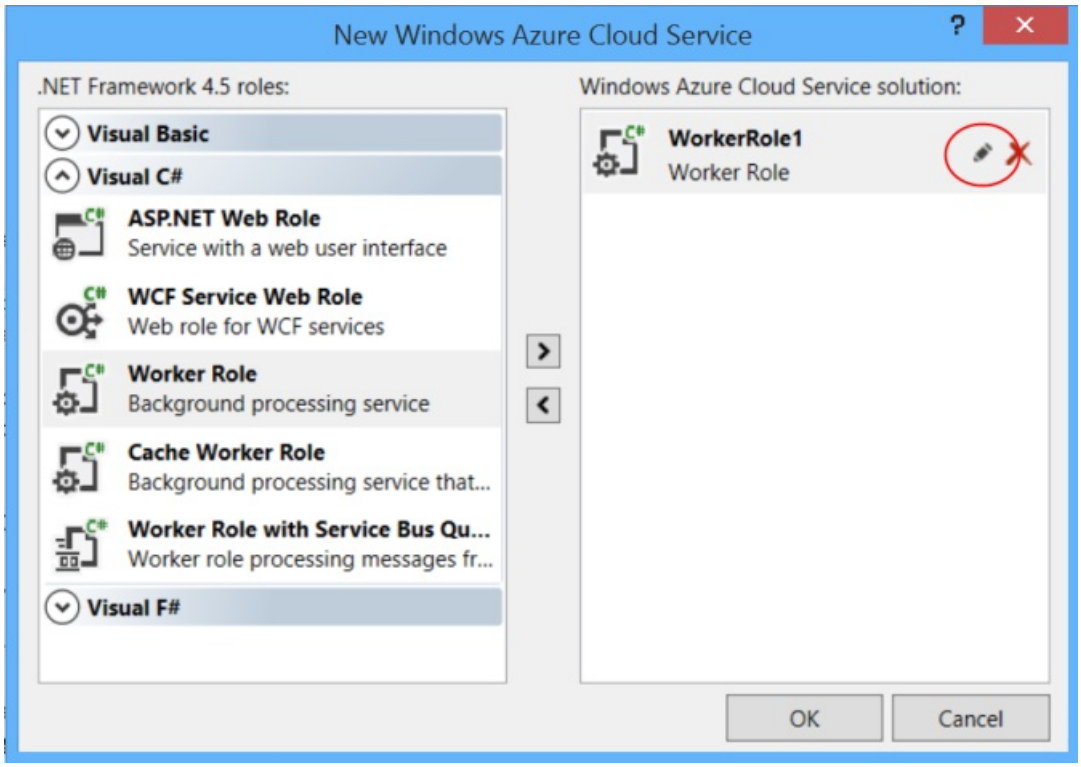


In the **New Azure Cloud Service** dialog, expand the **Visual C#** node on the left pane. Select **Worker Role** and click the right-arrow icon.



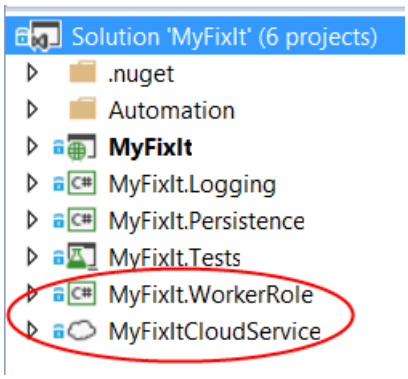
(Notice that you can also add a *web role*. We could run the Fix It front-end in the same Cloud Service instead of running it in an Azure Web Site. That has some advantages in making connections between front-end and back-end easier to coordinate. However, to keep this demo simple, we're keeping the front-end in an Azure App Service Web App and only running the back-end in a Cloud Service.)

A default name is assigned to the worker role. To change the name, hover the mouse over the worker role in the right pane, then click the pencil icon.



Click **OK** to complete the dialog. This adds two projects to the Visual Studio solution.

- an Azure project that defines the cloud service, including configuration information.
- A worker role project that defines the worker role.



For more information, see Creating an Azure Project with Visual Studio.

Inside the worker role, we poll for messages by calling the `ProcessMessageAsync` method of the `FixItQueueManager` class that we saw earlier.

```
public class WorkerRole : RoleEntryPoint
{
    public override void Run()
    {
        Task task = RunAsync(tokenSource.Token);
        try
        {
            task.Wait();
        }
        catch (Exception ex)
        {
            logger.Error(ex, "Unhandled exception in FixIt worker role.");
        }
    }

    private async Task RunAsync(CancellationToken token)
    {
        using (var scope = container.BeginLifetimeScope())
        {
            IFixItQueueManager queueManager = scope.Resolve<IFixItQueueManager>();
            while (!token.IsCancellationRequested)
            {
                try
                {
                    await queueManager.ProcessMessagesAsync();
                }
                catch (Exception ex)
                {
                    logger.Error(ex, "Exception in worker role Run loop.");
                }
                await Task.Delay(1000);
            }
        }
    }
    // Other code not shown.
}
```

The `ProcessMessagesAsync` method checks if there's a message waiting. If there is one, it deserializes the message into a `FixItTask` entity and saves the entity in the database. It loops until the queue is empty.

```
public async Task ProcessMessagesAsync()
{
    CloudQueue queue = _queueClient.GetQueueReference(fixitQueueName);
    await queue.CreateIfNotExistsAsync();
    while (true)
    {
        CloudQueueMessage message = await queue.GetMessageAsync();
        if (message == null)
        {
            break;
        }
        FixItTask fixit = JsonConvert.DeserializeObject<FixItTask>(message.AsString);
        await _repository.CreateAsync(fixit);
        await queue.DeleteMessageAsync(message);
    }
}
```

Polling for queue messages incurs a small transaction charge, so when there's no message waiting to be processed, the worker role's `RunAsync` method waits a second before polling again by calling `Task.Delay(1000)`.

In a web project, adding asynchronous code can automatically improve performance because IIS manages a limited thread pool. That is not the case in a worker role project. To improve scalability of the worker role, you can write multi-threaded code or use asynchronous code to implement parallel programming. The sample doesn't implement parallel programming but shows how to make the code asynchronous so you can implement parallel programming.

Summary

In this chapter you've seen how to improve application responsiveness, reliability, and scalability by implementing the queue-centric work pattern.

This is the last of the 13 patterns covered in this e-book, but there are of course many other patterns and practices that can help you build successful cloud apps. The final chapter provides links to resources for topics that haven't been covered in these 13 patterns.

Resources

For more information about queues, see the following resources.

Documentation:

- Microsoft Azure Storage Queues Part 1: Getting Started. Article by Roman Schacherl.
- Executing Background Tasks, chapter 5 of Moving Applications to the Cloud, 3rd Edition from Microsoft Patterns and Practices. (In particular, the section "Using Azure Storage Queues".)
- Best Practices for Maximizing Scalability and Cost Effectiveness of Queue-Based Messaging Solutions on Azure. White paper by Valery Mizonov.
- Comparing Azure Queues and Service Bus Queues. MSDN Magazine article, provides additional information that can help you choose which queue service to use. The article mentions that Service Bus is dependent on ACS for authentication, which means your SB queues would be unavailable when ACS is unavailable. However, since the article was written, SB was changed to enable you to use SAS tokens as an alternative to ACS.
- Microsoft Patterns and Practices - Azure Guidance. See Asynchronous Messaging primer, Pipes and Filters pattern, Compensating Transaction pattern, Competing Consumers pattern, CQRS pattern.
- CQRS Journey. E-book about CQRS by Microsoft Patterns and Practices.

Video:

- FailSafe: Building Scalable, Resilient Cloud Services. Nine-part video series by Ulrich Homann, Marc Mercuri,

and Mark Simms. Presents high-level concepts and architectural principles in a very accessible and interesting way, with stories drawn from Microsoft Customer Advisory Team (CAT) experience with actual customers. For an introduction to the Azure Storage service and queues, see episode 5 starting at 35:13.

PREVIOUS NEXT

More Patterns and Guidance (Building Real-World Cloud Apps with Azure)

2/20/2020 • 5 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download Fix It Project or Download E-book

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the e-book, see the first chapter.

You've now seen 13 patterns that provide guidance on how to be successful in cloud computing. These are just a few of the patterns that apply to cloud apps. Here are some more cloud computing topics, and resources to help with them:

- Migrating existing on-premises applications to the cloud.
  - Moving Applications to the Cloud. E-book by Microsoft Patterns and Practices. Also available as a hard-copy paperback.
  - Migrating Microsoft's ASP.NET and IIS.NET. Case study by Robert McMurray.
  - Moving 4th & Mayor to Azure Web Sites. Blog post by Jeff Wilcox chronicling his experience moving a web app from Amazon Web Services to Web Apps in Azure App Service.
  - Moving Apps to Azure: What changes? Short video by Stefan Schackow, explains file system access in Web Apps in Azure App Service.
  - Azure Hybrid Cloud. Hardcopy book or e-book by Danny Garber, Jamal Malik, and Adam Fazio.
- Security, authentication, and authorization issues unique to cloud applications
  - Azure Security Guidance
  - Microsoft Patterns and Practices - Azure Guidance. See Gatekeeper pattern, Federated Identity pattern.
  - Azure Network Security. White paper by Ashin Palekar.

See also additional cloud computing patterns and guidance at Microsoft Patterns and Practices - Azure Guidance.

Resources

Each of the chapters in this e-book provides links to resources for more information about that specific topic. The following list provides links to overviews of best practices and recommended patterns for successful cloud development with Azure.

Documentation

- Best Practices for the Design of Large-Scale Services on Azure Cloud Services. White paper by Mark Simms and Michael Thomassy.
- Failsafe: Guidance for Resilient Cloud Architectures. White paper by Marc Mercuri, Ulrich Homann, and Andrew Townhill. Web page version of the FailSafe video series.
- Azure Guidance Portal page for official documentation related to developing applications for Azure.

Videos

- Building Real World Cloud Apps with Azure - Part 1 and Part 2. Video of the presentation by Scott Guthrie that this e-book is based on. Presented at Tech Ed Australia in September, 2013. An earlier version of the same presentation was delivered at Norwegian Developers Conference (NDC) in June, 2013: NDC part 1, NDC part 2.
- FailSafe: Building Scalable, Resilient Cloud Services. Nine-part video series by Ulrich Homann, Marc Mercuri, and Mark Simms. Presents a 400-level view of how to architect cloud apps. This series focuses on theory and reasons behind recommended patterns; for more how-to details, see the Building Big series by Mark Simms.
- Building Big: Lessons learned from Azure customers- Part 1 and Part 2. Two-part video series by Simon Davies and Mark Simms, similar to the FailSafe series but oriented more toward practical implementation.

Code sample

- The Fix It application that accompanies this e-book.
- Cloud Service Fundamentals in Azure in C# for Visual Studio 2012. Downloadable project in the Microsoft Code Gallery site, includes both code and documentation developed by the Microsoft Customer Advisory Team (CAT). Demonstrates many of the best practices advocated in the FailSafe and Building Big video series and the FailSafe white paper. The Code Gallery page also links to extensive documentation by the authors of the project -- see especially the Cloud Service Fundamentals wiki collection link in the blue box near the top of the project description. This project and the documentation for it still actively being developed, making it a better choice for information on many topics than similar but older white papers.

Hard copy books

- Cloud Computing Bible. By Barrie Sosinsky.
- Release It! Design and Deploy Production-Ready Software. By Michael T. Nygard.
- Cloud Architecture Patterns: Using Microsoft Azure. By Bill Wilder.
- Windows Azure Platform. By Tejaswi Redkar.
- Windows Azure programming patterns for Start-ups. By Riccardo Becker.
- Microsoft Windows Azure Development Cookbook. By Neil Mackenzie.

Finally, when you get started building real-world apps and running them in Azure, sooner or later you'll probably need assistance from experts. You can ask questions in community sites such as Azure forums or StackOverflow, or you can contact Microsoft directly for Azure support. Microsoft offers several levels of technical support Azure: for a summary and comparison of the options, see Azure Support.

Acknowledgments

This content was written by Tom Dykstra, Rick Anderson, and Mike Wasson. Most of the original content came from Scott Guthrie, and he in turn drew on material from Mark Simms and the Microsoft Customer Advisory Team (CAT).

Many other colleagues at Microsoft reviewed and commented on drafts and code:

- Tim Ammann - Reviewed the automation chapter.
- Christopher Bennage - Reviewed and tested the Fix It code.
- Ryan Berry - Reviewed the CD/CI chapter.
- Vittorio Bertocci - Reviewed the SSO chapter.
- Chris Clayton - Helped resolve technical problems in the PowerShell scripts.
- Conor Cunningham - Reviewed the data storage options chapter.
- Carlos Farre - Reviewed and tested the Fix It code for security issues.
- Larry Franks - Reviewed the telemetry and monitoring chapter.
- Jonathan Gao - Reviewed Hadoop and MapReduce sections of the data storage options chapter.
- Sidney Higa - Reviewed all chapters.

- Gordon Hogenson - Reviewed the source control chapter.
- Tamra Myers - Reviewed data storage options, blob, and queues chapters.
- Pranav Rastogi - Reviewed the SSO chapter.
- June Blender Rogers - Added error handling and help to the PowerShell automation scripts.
- Mani Subramanian - Reviewed all chapters and led the code review and testing process for the Fix It code.
- Shaun Tinline-Jones - Reviewed the data partitioning chapter.
- Selcin Tukarslan - Reviewed chapters that cover SQL Database and SQL Server.
- Edward Wu - Provided sample code for the SSO chapter.
- Guang Yang - Wrote the PowerShell automation scripts.

Members of the Microsoft Developer Guidance Advisory Council (DGAC) also reviewed and commented on drafts:

- Jean-Luc Boucho
- Catalin Gheorghiu
- Wouter de Kort
- Carlos dos Santos
- Neil Mackenzie
- Dennis Persson
- Sunil Sabat
- Aleksey Sinyagin
- Bill Wagner
- Michael Wood

Other members of the DGAC reviewed and commented on the preliminary outline:

- Damir Arh
- Edward Bakker
- Srdjan Bozovic
- Ming Man Chan
- Gianni Rosa Gallina
- Paulo Morgado
- Jason Oliveira
- Alberto Poblacion
- Ryan Riley
- Perez Jones Tsisah
- Roger Whitehead
- Pawel Wilkosz

Appendix: The Fix It Sample Application (Building Real-World Cloud Apps with Azure)

9/16/2020 • 22 minutes to read • Edit Online

by Mike Wasson, Rick Anderson, Tom Dykstra

Download The Fix It Project

> The **Building Real World Cloud Apps with Azure** e-book is based on a presentation developed by Scott Guthrie. It explains 13 patterns and practices that can help you be successful developing web apps for the cloud. For information about the e-book, see the first chapter.

This appendix to the Building Real World Cloud Apps with Azure e-book contains the following sections that provide additional information about the Fix It sample application that you can download:

- Known issues
- Best practices
- How to run the app from Visual Studio on your local computer
- How to deploy the base app to Azure App Service Web Apps by using the Windows PowerShell scripts
- Troubleshooting the Windows PowerShell scripts
- How to deploy the app with queue processing to Azure App Service Web Apps and an Azure Cloud Service

Known issues

The Fix It app was originally developed in order to illustrate as simply as possible some of the patterns presented in this e-book. However, since the e-book is about building real-world apps, we subjected the Fix It code to a review and testing process similar to what we'd do for released software. We found a number of issues, and as with any real-world application, some of them we fixed and some of them we deferred to a later release.

The following list includes issues that should be addressed in a production application, but for one reason or another we decided not to address in the initial release of the Fix It sample application.

Security

- Ensure that you can't assign a task to a non-existent owner.
- Ensure that you can only view and modify tasks that you created or are assigned to you.
- Use HTTPS for sign-in pages and authentication cookies.
- Specify a time limit for authentication cookies.

Input validation

In general, a production app would do more input validation than the Fix It app. For example, the image size / image file size allowed for upload should be limited.

Administrator functionality

An administrator should be able to change ownership on existing tasks. For example, the creator of a task might leave the company, leaving no one with authority to maintain the task unless administrative access is enabled.

Queue message processing

Queue message processing in the Fix It app was designed to be simple in order to illustrate the queue-centric work pattern with a minimum amount of code. This simple code would not be adequate for an actual production

application.

- The code does not guarantee that each queue message will be processed at most once. When you get a message from the queue, there is a timeout period, during which the message is invisible to other queue listeners. If the timeout expires before the message is deleted, the message becomes visible again. Therefore, if a worker role instance spends a long time processing a message, it is theoretically possible for the same message to get processed twice, resulting in a duplicate task in the database. For more information about this issue, see Using Azure Storage Queues.
- The queue polling logic could be more cost-effective, by batching message retrieval. Every time you call CloudQueue.GetMessageAsync, there is a transaction cost. Instead, you can call CloudQueue.GetMessagesAsync (note the plural 's'), which gets multiple messages in a single transaction. The transaction costs for Azure Storage Queues are very low, so the impact on costs is not substantial in most scenarios.
- The tight loop in the queue message-processing code causes CPU affinity, which does not utilize multi-core VMs efficiently. A better design would use task parallelism to run several async tasks in parallel.
- Queue message-processing has only rudimentary exception handling. For example, the code doesn't handle poison messages. (When message processing causes an exception, you have to log the error and delete the message, or the worker role will try to process it again, and the loop will continue indefinitely.)

SQL queries are unbounded

Current Fix It code places no limit on how many rows the queries for Index pages might return. If a large volume of tasks is entered into the database, the size of the resulting lists received could cause performance issues. The solution is to implement paging. For an example, see Sorting, Filtering, and Paging with the Entity Framework in an ASP.NET MVC Application.

View models recommended

The Fix It app uses the FixItTask entity class to pass information between the controller and the view. A best practice is to use view models. The domain model (e.g., the FixItTask entity class) is designed around what is needed for data persistence, while a view model can be designed for data presentation. For more information, see 12 ASP.NET MVC Best Practices.

Secure image blob recommended

The Fix It app stores uploaded images as public, meaning that anyone who finds the URL can access the images. The images could be secured instead of public.

No PowerShell automation scripts for queues

Sample PowerShell automation scripts were written only for the base version of Fix It that runs entirely in Azure App Service Web Apps. We haven't provided scripts for setting up and deploying to the web app plus Cloud Service environment required for queue processing.

Special handling for HTML codes in user input

ASP.NET automatically prevents many ways in which malicious users might attempt cross-site scripting attacks by entering script in user input text boxes. And the MVC `DisplayFor` helper used to display task titles and notes automatically HTML-encodes values that it sends to the browser. But in a production app you might want to take additional measures. For more information, see Request Validation in ASP.NET.

Best practices

Following are some issues that were fixed after being discovered in code review and testing of the original version of the Fix It app. Some were caused by the original coder not being aware of a particular best practice, some simply because the code was written quickly and wasn't intended for released software. We're listing the issues here in case there's something we learned from this review and testing that might be helpful to others who are also developing web apps.

Dispose the database repository

The `FixItTaskRepository` class must dispose the Entity Framework `DbContext` instance. We did this by implementing `IDisposable` in the `FixItTaskRepository` class:

```
public class FixItTaskRepository : IFixItTaskRepository, IDisposable
{
    private MyFixItContext db = new MyFixItContext();

    // other code not shown

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            // Free managed resources.
            if (db != null)
            {
                db.Dispose();
                db = null;
            }
        }
    }
}
```

Note that AutoFac will automatically dispose the `FixItTaskRepository` instance, so we don't need to explicitly dispose it.

Another option is to remove the `DbContext` member variable from `FixItTaskRepository`, and instead create a local `DbContext` variable within each repository method, inside a `using` statement. For example:

```
// Alternate way to dispose the DbContext
using (var db = new MyFixItContext())
{
    fixItTask = await db.FixItTasks.FindAsync(id);
}
```

Register singletons as such with DI

Since only one instance of the `PhotoService` class and `Logger` class is needed, these classes should be registered as single instances for dependency injection in *DependenciesConfig.cs*:

```
builder.RegisterType<Logger>().As<ILogger>().SingleInstance();
builder.RegisterType<FixItTaskRepository>().As<IFixItTaskRepository>();
builder.RegisterType<PhotoService>().As<IPhotoService>().SingleInstance();
```

Security: Don't show error details to users

The original Fix It app didn't have a generic error page and just let all exceptions bubble up to the UI, so some exceptions such as database connection errors could result in a full stack trace being displayed to the browser. Detailed error information can sometimes facilitate attacks by malicious users. The solution is to log the exception details and display an error page to the user that doesn't include error details. The Fix It app was already logging, and in order to display an error page, we added `<customErrors mode=On>` in the Web.config file.

```
<system.web>
  <customErrors mode="On"/>
  <authentication mode="None" />
  <compilation debug="true" targetFramework="4.5" />
  <httpRuntime targetFramework="4.5" />
</system.web>
```

By default this causes *Views\Shared\Error.cshtml* to be displayed for errors. You can customize *Error.cshtml* or create your own error page view and add a `defaultRedirect` attribute. You can also specify different error pages for specific errors.

Security: only allow a task to be edited by its creator

The Dashboard Index page only shows tasks created by the logged-on user, but a malicious user could create a URL with an ID to another user's task. We added code in *DashboardController.cs* to return a 404 in that case:

```
public async Task<ActionResult> Edit(int id)
{
    FixItTask fixittask = await fixItRepository.FindTaskByIdAsync(id);
    if (fixittask == null)
    {
        return HttpNotFound();
    }

    // Verify logged in user owns this FixIt task.
    if (User.Identity.Name != fixittask.Owner)
    {
       return HttpNotFound();
    }

    return View(fixittask);
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Edit(int id, [Bind(Include =
"CreatedBy,Owner,Title,Notes,PhotoUrl,IsDone")]FormCollection form)
{
    FixItTask fixittask = await fixItRepository.FindTaskByIdAsync(id);

    // Verify logged in user owns this FixIt task.
    if (User.Identity.Name != fixittask.Owner)
    {
       return HttpNotFound();
    }

    if (TryUpdateModel(fixittask, form))
    {
        await fixItRepository.UpdateAsync(fixittask);
        return RedirectToAction("Index");
    }

    return View(fixittask);
}
```

Don't swallow exceptions

The original Fix It app just returned null after logging an exception that resulted from a SQL query:

```
catch (Exception e)
{
    log.Error(e, "Error in FixItTaskRepository.FindTasksByOwnerAsync(userName={0})", userName);
    return null;
}
```

This would make it look to the user as if the query succeeded but just didn't return any rows. Solution is to re-throw the exception after catching and logging:

```
catch (Exception e)
{
    log.Error(e, "Error in FixItTaskRepository.FindTasksByCreatorAsync(creater={0})", creator);
    throw;
}
```

Catch all exceptions in worker roles

Any unhandled exceptions in a worker role will cause the VM to be recycled, so you want to wrap everything you do in a try-catch block and handle all exceptions.

Specify length for string properties in entity classes

In order to display simple code, the original version of the Fix It app didn't specify lengths for the fields of the FixItTask entity, and as a result they were defined as varchar(max) in the database. As a result, the UI would accept almost any amount of input. Specifying lengths sets limits that apply both to user input in the web page and column size in the database:

```
public class FixItTask
{
    public int FixItTaskId  { get; set; }
    [StringLength(80)]
    public string CreatedBy { get; set; }
    [Required]
    [StringLength(80)]
    public string Owner { get; set; }
    [Required]
    [StringLength(80)]
    public string Title { get; set; }
    [StringLength(1000)]
    public string Notes { get; set; }
    [StringLength(200)]
    public string PhotoUrl { get; set; }
    public bool IsDone      { get; set; }
}
```

Mark private members as readonly when they aren't expected to change

For example, in the `DashboardController` class an instance of `FixItTaskRepository` is created and isn't expected to change, so we defined it as readonly.

```
public class DashboardController : Controller
    {
        private readonly IFixItTaskRepository fixItRepository = null;
```

Use list.Any() instead of list.Count() > 0

If you all you care about is whether one or more items in a list fit the specified criteria, use the Any method, because it returns as soon as an item fitting the criteria is found, whereas the `Count` method always has to iterate through every item. The Dashboard *Index.cshtml* file originally had this code:

```
@if (Model.Count() == 0) {
    <br />
    <div>You don't have anything currently assigned to you!!!</div>
}
```

We changed it to this:

```
@if (!Model.Any()) {
    <br />
    <div>You don't have anything currently assigned to you!!!</div>
}
```

Generate URLs in MVC views using MVC helpers

For the **Create a Fix It** button on the home page, the Fix It app hard coded an anchor element:

```
<a href="/Tasks/Create" class="btn btn-primary btn-large">Create a New FixIt &raquo;</a>
```

For View/Action links like this it's better to use the Url.Action HTML helper, for example:

```
@Url.Action("Create","Tasks")
```

Use Task.Delay instead of Thread.Sleep in worker role

The new-project template puts `Thread.Sleep` in the sample code for a worker role, but causing the thread to sleep can cause the thread pool to spawn additional unnecessary threads. You can avoid that by using Task.Delay instead.

```
while (true)
{
    try
    {
        await queueManager.ProcessMessagesAsync();
    }
    catch (Exception ex)
    {
        logger.Error(ex, "Exception in worker role Run loop.");
    }
    await Task.Delay(1000);
}
```

Avoid async void

If an async method doesn't need to return a value, return a `Task` type rather than `void`.

This example is from the `FixItQueueManager` class:

```
// Correct
public async Task SendMessageAsync(FixItTask fixIt) { ... }

// Incorrect
public async void SendMessageAsync(FixItTask fixIt) { ... }
```

You should use `async void` only for top-level event handlers. If you define a method as `async void`, the caller cannot **await** the method or catch any exceptions the method throws. For more information, see Best Practices in Asynchronous Programming.

Use a cancellation token to break from worker role loop

Typically, the **Run** method on a worker role contains an infinite loop. When the worker role is stopping, the RoleEntryPoint.OnStop method is called. You should use this method to cancel the work that is being done inside the **Run** method and exit gracefully. Otherwise, the process might be terminated in the middle of an operation.

Opt out of Automatic MIME Sniffing Procedure

In some cases, Internet Explorer reports a MIME type different than the type specified by the web server. For instance, if Internet Explorer finds HTML content in a file delivered with the HTTP response header Content-Type: text/plain, Internet Explorer determines that the content should be rendered as HTML. Unfortunately, this "MIME-sniffing" can also lead to security problems for servers hosting untrusted content. To combat this problem, Internet Explorer 8 has made a number of changes to MIME-type determination code and allows application developers to opt out of MIME-sniffing. The following code was added to the *Web.config* file.

```
<system.webServer>
     <httpProtocol>
        <customHeaders>
           <add name="X-Content-Type-Options" value="nosniff"/>
        </customHeaders>
     </httpProtocol>
     <modules>
      <remove name="FormsAuthenticationModule" />
    </modules>
  </system.webServer>
```

Enable bundling and minification

When Visual Studio creates a new web project, bundling and minification of JavaScript files is not enabled by default. We added a line of code in BundleConfig.cs:

```
// For more information on bundling, visit https://go.microsoft.com/fwlink/?LinkId=301862
public static void RegisterBundles(BundleCollection bundles)
{
    bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
                "~/Scripts/jquery-{version}.js"));

   // Code removed for brevity/

   BundleTable.EnableOptimizations = true;
}
```

Set an expiration time-out for authentication cookies

By default, authentication cookies expire in two weeks. A shorter time is more secure. You can change this setting in *StartupAuth.cs*:

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
    LoginPath = new PathString("/Account/Login"),
    ExpireTimeSpan = System.TimeSpan.FromMinutes(20)
});
```

How to run the app from Visual Studio on your local computer

There are two ways to run the Fix It app:

- Run the base application that writes new tasks directly to the SQL database.
- Run the application using a queue plus a backend service to create tasks. The queue pattern is described in the

chapter Queue-Centric Work Pattern.

Run the base application

1. Install Visual Studio 2017.
2. Install the Azure SDK for .NET for Visual Studio.
3. Download the .zip file from the MSDN Code Gallery.
4. In File Explorer, right-click the .zip file and click Properties, then in the Properties window click Unblock.
5. Unzip the file.
6. Double-click the .sln file to launch Visual Studio.
7. From the **Tools** menu, click **NuGet Package Manager**, then **Package Manager Console**.
8. In the Package Manager Console (PMC), click Restore.
9. Exit Visual Studio.
10. Start the Azure Storage Emulator.
11. Restart Visual Studio, opening the solution file you closed in the previous step.
12. Make sure the FixIt project is set as the startup project, and then press CTRL+F5 to run the project.

Run the application with queue processing

1. Follow the directions for Run the base application, and then close the browser and close Visual Studio.
2. Start Visual Studio with administrator privileges. (You'll be using the Azure compute emulator, and that requires administrator privileges.)
3. In the application *Web.config* file in the *MyFixIt* project (the web project), change the value of `appSettings/UseQueues` to "true":

   ```
   <appSettings>
       <!-- Other settings not shown -->
       <add key="UseQueues" value="true"/>
   </appSettings>
   ```

4. If the Azure Storage Emulator isn't still running, start it again.
5. Run the FixIt web project and the MyFixItCloudService project simultaneously.

   Using Visual Studio:

   a. Press **F5** to run the FixIt project.
   b. In **Solution Explorer**, right-click the MyFixItCloudService project, and then click **Debug** > **Start New Instance**.

   Using Visual Studio 2013 Express for Web:

   c. In Solution Explorer, right-click the FixIt solution and select **Properties**.
   d. Select **Multiple Startup Projects**.
   e. In the **Action** dropdown list under MyFixIt and MyFixItCloudService, select **Start**.
   f. Click **OK**.
   g. Press **F5** to run both projects.

      When you run the MyFixItCloudService project, Visual Studio starts the Azure compute emulator. Depending on your firewall configuration, you might need to allow the emulator through the firewall.

How to deploy the base app to Azure App Service Web Apps by using the Windows PowerShell scripts

To illustrate the Automate Everything pattern, the Fix It app is supplied with scripts that set up an environment in Azure and deploy the project to the new environment. The following instructions explain how to use the scripts.

If you want to run in Azure without using queues, and you made the changes to run locally with queues, make sure you set the UseQueues appSetting value back to false before proceeding with the following instructions.

These instructions assume you have already downloaded and run the Fix It solution locally, and that you have an Azure account or have an Azure subscription that you are authorized to manage.

1. Install the **Azure PowerShell** console. For instructions, see How to install and configure Azure PowerShell.

   This customized console is configured to work with your Azure subscription. The Azure module is installed in the *Program Files* directory and is automatically imported on every use of the Azure PowerShell console.

   If you prefer to work in a different host program, such as Windows PowerShell ISE, be sure to use the Import-Module cmdlet to import the Azure module or use a command in the Azure module to trigger automatic importing of the module.

2. Start Azure PowerShell with the **Run as administrator** option.

3. Run the Set-ExecutionPolicy cmdlet to set the Azure PowerShell execution policy to `RemoteSigned`. Enter **Y** (for Yes) to complete the policy change.

   ```
   PS C:\> Set-ExecutionPolicy RemoteSigned
   ```

   This setting enables you to run local scripts that aren't digitally signed. (You can also set the execution policy to `Unrestricted`, which would eliminate the need for the unblock step later, but this is not recommended for security reasons.)

4. Run the `Add-AzureAccount` cmdlet to set up PowerShell with credentials for your account.

   ```
   PS C:\> Add-AzureAccount
   ```

   These credentials expire after a period of time and you have to re-run the `Add-AzureAccount` cmdlet. As this e-book is being written, the time limit before credentials expire is 12 hours.

5. If you have multiple subscriptions, use the Select-AzureSubscription cmdlet to specify the subscription you want to create the test environment in.

6. Import a management certificate for the same Azure subscription by using the `Get-AzurePublishSettingsFile` and `Import-AzurePublishSettingsFile` cmdlets. The first of these cmdlets downloads a certificate file, and in the second one you specify the location of that file in order to import it.
   > [!IMPORTANT]

   > Keep the downloaded file in a safe location or delete it when you're done with it, because it contains a certificate that can be used to manage your Azure services.

   ```
   PS C:\Users\username\Documents\Visual Studio 2013\Projects\MyFixIt\Automation> Get-
   AzurePublishSettingsFile
   PS C:\Users\username\Documents\Visual Studio 2013\Projects\MyFixIt\Automation> Import-
   AzurePublishSettingsFile "C:\Users
   \username\Downloads\Azure MSDN - Visual Studio Ultimate-12-14-2013-credentials.publishsettings"
   ```

The certificate is used for a REST API call that detects the development machine's IP address in order to set a firewall rule on the SQL Database server.

7. Run the Set-Location cmdlet (aliases are `cd`, `chdir`, and `sl`) to navigate to the directory that contains the scripts. (They're located in the *Automation* folder in the Fix It solution folder.) Put the path in quotes if any of the directory names contain spaces. For example, to navigate to the `c:\Sample Apps\FixIt\Automation` directory you could enter the following command:

```
PS C:\> cd "c:\Sample Apps\MyFixIt\Automation"
```

8. To allow Windows PowerShell to run these scripts, use the Unblock-File cmdlet. (The scripts are blocked because they were downloaded from the Internet.)

> **WARNING**
> Security - Before running `Unblock-File` on any script or executable file, open the file in Notepad, examine the commands, and verify that they do not contain any malicious code.

For example, the following command runs the `Unblock-File` cmdlet on all scripts in the current directory.

```
PS C:\Sample Apps\FixIt\Automation> Unblock-File -Path .\*.ps1
```

9. To create the web app for the base (no queues processing) Fix It app, run the environment creation script.

The required `Name` parameter specifies the name of the database and is also used for the storage account that the script creates. The name must be globally unique within the azurewebsites.net domain. If you specify a name that is not unique, like Fixit or Test (or even as in the example, fixitdemo), the `New-AzureWebsite` cmdlet fails with an Internal Error that reports a conflict. The script converts the name to all lower-case to comply with name requirements for web apps, storage accounts, and databases.

The required `SqlDatabasePassword` parameter specifies the password for the admin account that will be created for SQL Database. Don't include special XML characters in the password (& < > ;). This is a limitation of the way the scripts were written, not a limitation of Azure.

For example, if you want to create a web app named "fixitdemo" and use a SQL Server administrator password of "Passw0rd1", you could enter the following command:

```
PS C:\Sample Apps\FixIt\Automation> .\New-AzureWebsiteEnv.ps1 -Name
fixitdemo <required params here>
```

The name must be unique in the azurewebsites.net domain, and the password must meet SQL Database requirements for password complexity. (The example Passw0rd1 does meet the requirements.)

Note that the command begins with ".". To help prevent malicious execution of scripts, Windows PowerShell requires that you provide the fully qualified path to the script file when you run a script. You can use a dot to indicate the current directory (".") or provide the fully qualified path, such as:

```
PS C:\Temp\FixIt\Automation> C:\Temp\FixIt\Automation\New-AzureWebsiteEnv.ps1 -Name fixitdemo -
SqlDatabasePassword Pas$w0rd
```

For more information about the script, use the `Get-Help` cmdlet.

```
PS C:\Sample Apps\FixIt\Automation> Get-Help -Full .\New-AzureWebsiteEnv.ps1
```

You can use the `Detailed`, `Full`, `Parameters`, and `Examples` parameters of the Get-Help cmdlet to filter the help that is returned.

If the script fails or generates errors, such as "New-AzureWebsite : Call Set-AzureSubscription and Select-AzureSubscription first," you might not have completed the configuration of Azure PowerShell.

After the script finishes, you can use the Azure Management Portal to see the resources that were created, as shown in the Automate Everything chapter.

10. To deploy the FixIt project to the new Azure environment, use the *AzureWebsite.ps1* script. For example:

```
PS C:\Sample Apps\FixIt\Automation> .\Publish-AzureWebsite.ps1 ..\MyFixIt\MyFixIt.csproj -Launch
```

When deployment is done, the browser opens with Fix It running in Azure.

Troubleshooting the Windows PowerShell scripts

The most common errors encountered when running these scripts are related to permissions. Make sure that `Add-AzureAccount` and `Import-AzurePublishSettingsFile` were successful and that you used them for the same Azure subscription. Even if `Add-AzureAccount` was successful you might have to run it again. The permissions added by `Add-AzureAccount` expire in 12 hours.

Object reference not set to an instance of an object.

If the script returns errors, such as "Object reference not set to an instance of an object," which means that Windows PowerShell can't find an object to process (this is a null reference exception), run the `Add-AzureAccount` cmdlet and try the script again.

```
New-AzureSqlDatabaseServer : Object reference not set to an instance of an object.
At C:\ps-test\azure-powershell-samples-master\WebSite\create-azure-sql.ps1:80 char:19
+ $databaseServer = New-AzureSqlDatabaseServer -AdministratorLogin $UserName -Admi ...
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo          : NotSpecified: (:) [New-AzureSqlDatabaseServer], NullReferenceException
    + FullyQualifiedErrorId :
Microsoft.WindowsAzure.Commands.SqlDatabase.Server.Cmdlet.NewAzureSqlDatabaseServer
```

InternalError: The server encountered an internal error.

The `New-AzureWebsite` cmdlet returns an internal error when the name is not unique in the azurewebsites.net domain. To resolve the error, use a different value for the name, which is in the Name parameter of *New-AzureWebsiteEnv.ps1*.

```
New-AzureWebsite : InternalError: The server encountered an internal error.
Please retry the request.
At line:1 char:1
+ New-AzureWebsite -Name fixitdemo
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo
: CloseError: (:) [New-AzureWebsite], Exception
+ FullyQualifiedErrorId :
Microsoft.WindowsAzure.Commands.Websites.NewAzureWebsiteCommand
```

Restarting the script

If you need to restart the *New-AzureWebsiteEnv.ps1* script because it failed before it printed the "Script is

complete" message, you might want to delete resources that the script created before it stopped. For example, if the script already created the ContosoFixItDemo web app and you run the script again with the same name, the script will fail because the name is in use.

To determine which resources the script created before it stopped, use the following cmdlets:

- `Get-AzureWebsite`
- `Get-AzureSqlDatabaseServer`
- `Get-AzureSqlDatabase`: To run this cmdlet, pipe the database server name to `Get-AzureSqlDatabase`: `Get-AzureSqlDatabaseServer | Get-AzureSqlDatabase.`

To delete these resources, use the following commands. Note that if you delete the database server, you automatically delete the databases associated with the server.

- `Get-AzureWebsite -Name <WebsiteName> | Remove-AzureWebsite`
- `Get-AzureSqlDatabase -Name <DatabaseName> -ServerName <DatabaseServerName> | Remove-SqlAzureDatabase`
- `Get-AzureSqlDatabaseServer | Remove-AzureSqlDatabaseServer`

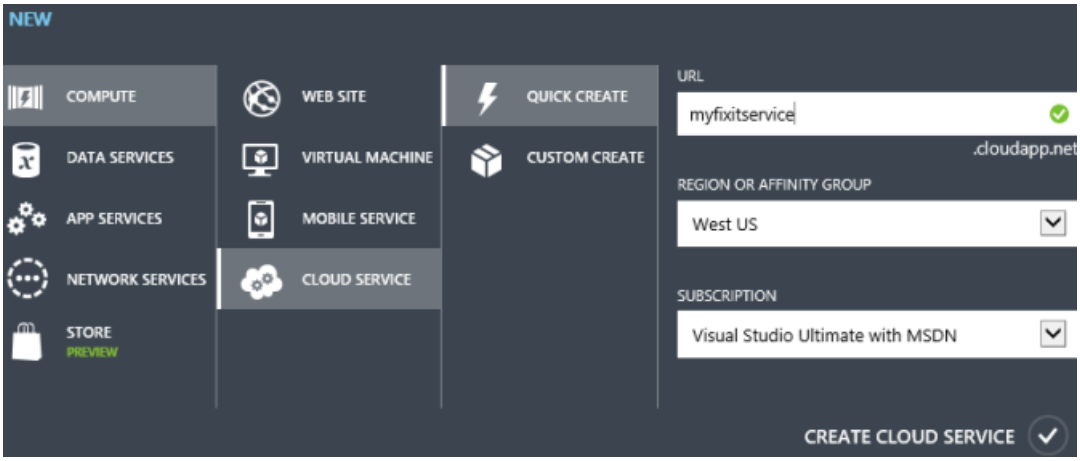
How to deploy the app with queue processing to Azure App Service Web Apps and an Azure Cloud Service

To enable queues, make the following change in the MyFixIt\Web.config file. Under `appSettings`, change the value of `UseQueues` to "true":

```
<appSettings>
    <!-- Other settings not shown -->
    <add key="UseQueues" value="true"/>
</appSettings>
```

Then deploy the MVC application to an web app in Azure App Service, as described earlier.

Next, create a new Azure cloud service. The scripts included with the Fix It app do not create or deploy the cloud service, so you must use Azure portal for this. In the portal, click **New** -- **Compute** – **Cloud Service** -- **Quick Create**, and then enter a URL and a data center location. Use the same data center where you deployed the web app.



Before you can deploy the cloud service, you need to update some of the configuration files.

In MyFixIt.WorkerRole\app.config, under `connectionStrings`, replace the value of the `appdb` connection string with the actual connection string for the SQL Database. You can get the connection string from the portal. In the portal, click **SQL Databases** - **appdb** - **View SQL Database connection strings for ADO .Net, ODBC, PHP, and JDBC**. Copy the ADO.NET connection string and paste the value into the app.config file. Replace "

{your_password_here}" with your database password. (Assuming you used the scripts to deploy the MVC app, you specified the database password in the `SqlDatabasePassword` script parameter.)

The result should look like the following:

```
<add name="appdb" connectionString="Server=tcp:####.database.windows.net,1433;Database=appdb;User
ID=####;Password=####;Trusted_Connection=False;Encrypt=True;Connection Timeout=30;"
providerName="System.Data.SqlClient" />
```

In the same MyFixIt.WorkerRole\app.config file, under `appSettings`, replace the two placeholder values for the Azure storage account.

```
<appSettings>
  <add key="StorageAccountName" value="{StorageAccountName}" />
  <add key="StorageAccountAccessKey" value="{StorageAccountAccessKey}" />
</appSettings>
```

You can get the access key from the portal. See How To Manage Storage Accounts.

In MyFixItCloudService\ServiceConfiguration.Cloud.cscfg, replace the same two placeholders values for the Azure storage account.

```
<ConfigurationSettings>
    <Setting name="Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString"
             value="DefaultEndpointsProtocol=https;AccountName={StorageAccountName};AccountKey=
{StorageAccountAccessKey}" />
  </ConfigurationSettings>
```

Now you are ready to deploy the cloud service. In Solution Explore, right-click the MyFixItCloudService project and select **Publish**. For more information, see "Deploy the Application to Azure", which is in part 2 of this tutorial.

PREVIOUS

Hands on Lab: Maintainable Azure Websites: Managing Change and Scale

2/20/2020 • 34 minutes to read • Edit Online

by Web Camps Team

Download Web Camps Training Kit

Microsoft Azure makes it easy to build and deploy websites to production. But you're not done when your application is live, you're just getting started! You'll need to handle changing requirements, database updates, scale, and more. Fortunately, Azure App Service has you covered, with plenty of features to help you keep your sites running smoothly.

Azure offers secure and flexible development, deployment and scaling options for any size web application. Leverage your existing tools to create and deploy applications without the hassle of managing infrastructure.

Provision a production web application yourself in minutes by easily deploying content created using your favorite development tool. You can deploy an existing site directly from source control with support for **Git**, **GitHub**, **Bitbucket**, **TFS**, and even **DropBox**. Deploy directly from your favorite IDE or from scripts using **PowerShell** in Windows or **CLI** tools running on any OS. Once deployed, keep your sites constantly up-to-date with support for continuous deployment.

Azure provides scalable, durable cloud storage, backup, and recovery solutions for any data, big or small. When deploying applications to a production environment, storage services such as Tables, Blobs and SQL Databases, help you scale your application in the cloud.

With SQL Databases, it is important to keep your productive database up-to-date when deploying new versions of your application. Thanks to **Entity Framework Code First Migrations**, the development and deployment of your data model has been simplified to update your environments in minutes. This hands-on lab will show you the different topics you could encounter when deploying your web app to production environments in Microsoft Azure.

All sample code and snippets are included in the Web Camps Training Kit, available at https://aka.ms/webcamps-training-kit.

For more in-depth coverage of this topic, see the Building Real-World Cloud Apps with Azure e-book.

Overview

Objectives

In this hands-on lab, you will learn how to:

- Enable Entity Framework Migrations with an existing model
- Update the object model and database accordingly using Entity Framework Migrations
- Deploy to Azure App Service using Git
- Rollback to a previous deployment using the Azure Management portal
- Use Azure Storage to scale a web app
- Configure auto-scaling for a web app using the Azure Management Portal
- Create and configure a load test project in Visual Studio

Prerequisites

The following is required to complete this hands-on lab:

- Visual Studio Express 2013 for Web or greater
- Azure SDK for .NET 2.2
- GIT Version Control System
- A Microsoft Azure subscription
  - Sign up for a Free Trial
  - If you are a Visual Studio Professional, Test Professional, Premium or Ultimate with MSDN or MSDN Platforms subscriber, activate your MSDN benefit now to start developing and testing on Azure
  - BizSpark members automatically receive the Azure benefit through their Visual Studio Ultimate with MSDN subscriptions
  - Members of the Microsoft Partner Network Cloud Essentials program receive monthly Azure credits at no charge

Setup

In order to run the exercises in this hands-on lab, you will need to set up your environment first.

1. Open Windows Explorer and browse to the lab's **Source** folder.
2. Right-click on **Setup.cmd** and select **Run as administrator** to launch the setup process that will configure your environment and install the Visual Studio code snippets for this lab.
3. If the User Account Control dialog is shown, confirm the action to proceed.

> **NOTE**
> Make sure you have checked all the dependencies for this lab before running the setup.

Using the Code Snippets

Throughout the lab document, you will be instructed to insert code blocks. For your convenience, most of this code is provided as Visual Studio Code Snippets, which you can access from within Visual Studio 2013 to avoid having to add it manually.

> **NOTE**
> Each exercise is accompanied by a starting solution located in the **Begin** folder of the exercise that allows you to follow each exercise independently of the others. Please be aware that the code snippets that are added during an exercise are missing from these starting solutions and may not work until you have completed the exercise. Inside the source code for an exercise, you will also find an **End** folder containing a Visual Studio solution with the code that results from completing the steps in the corresponding exercise. You can use these solutions as guidance if you need additional help as you work through this hands-on lab.

Exercises

This hands-on lab includes the following exercises:

1. Using Entity Framework Migrations
2. Deploying a Web App to Staging
3. Performing Deployment Rollback in Production
4. Scaling Using Azure Storage
5. Using Autoscale for Web Apps (Optional for Visual Studio 2013 Ultimate edition)

Estimated time to complete this lab: **75 minutes**

> **NOTE**
>
> When you first start Visual Studio, you must select one of the predefined settings collections. Each predefined collection is designed to match a particular development style and determines window layouts, editor behavior, IntelliSense code snippets, and dialog box options. The procedures in this lab describe the actions necessary to accomplish a given task in Visual Studio when using the **General Development Settings** collection. If you choose a different settings collection for your development environment, there may be differences in the steps that you should take into account.

Exercise 1: Using Entity Framework Migrations

When you are developing an application, your data model might change over time. These changes could affect the existing model in your database (if you are creating a new version) and it is important to keep your database up-to-date to prevent errors.

To simplify the tracking of these changes in your model, **Entity Framework Code First Migrations** automatically detect changes comparing your model with the database schema and generates specific code to update your database, creating new *versions* of your database.

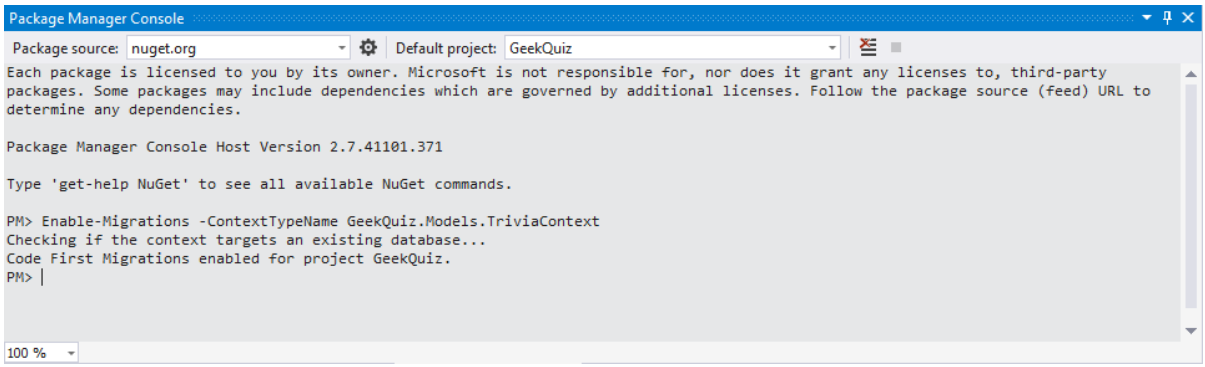
This exercise shows you how to enable **Migrations** for your application and how you can easily detect and generate changes to update your databases.

Task 1 – Enabling Migrations

In this task, you will go through the steps of enabling **Entity Framework Code First Migrations** to the **Geek Quiz** database, changing the model and understanding how those changes are reflected in the database.

1. Open Visual Studio and open the **GeekQuiz.sln** solution file from **Source\Ex1-UsingEntityFrameworkMigrations\Begin**.
2. Build the solution in order to download and install the **NuGet** package dependencies. To do this, right-click the solution and click **Build Solution** or press **Ctrl + Shift + B**.
3. From the **Tools** menu in Visual Studio, select **NuGet Package Manager**, and then click **Package Manager Console**.
4. In the **Package Manager Console**, enter the following command and then press **Enter**. An initial migration based on the existing model will be created.

```
Enable-Migrations -ContextTypeName GeekQuiz.Models.TriviaContext
```



Enabling Migrations

> **NOTE**
>
> This command adds a **Migrations** folder to Geek Quiz project that contains a file called **Configuration.cs**. The **Configuration** class allows you to configure how Migrations behaves for your context.

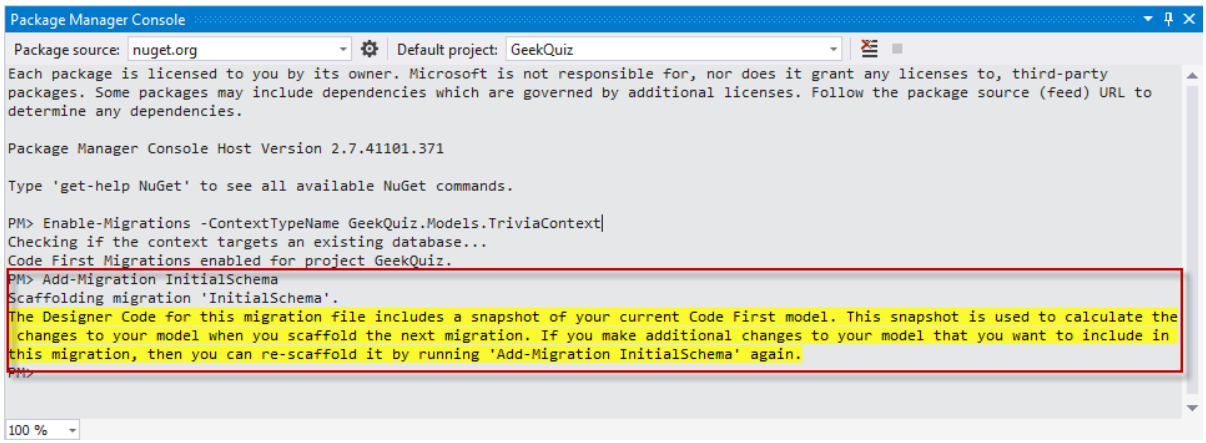
5. With Migrations enabled, you need to update the **Configuration** class to populate the database with the initial data that **Geek Quiz** requires. Under **Migrations**, replace the **Configuration.cs** file with the one located in the **Source\Assets** folder of this lab.

> **NOTE**
>
> Since **Migrations** will call the **Seed** method with every database update, you need to be sure that records are not duplicated in the database. The **AddOrUpdate** method will help to prevent duplicate data.

6. To add an initial migration, enter the following command and then press **Enter**.

> **NOTE**
>
> Make sure that there is no database named "GeekQuizProd" in your LocalDB instance.

```
Add-Migration InitialSchema
```



Adding base schema migration

> **NOTE**
>
> **Add-Migration** will scaffold the next migration based on changes you have made to your model since the last migration was created. In this case, as it is the first migration of the project, it will add the scripts to create all the tables defined in the **TriviaContext** class.

7. Execute the migration to update the database by running the following command. For this command you will specify the **Verbose** flag to view the SQL statements being applied to the target database.

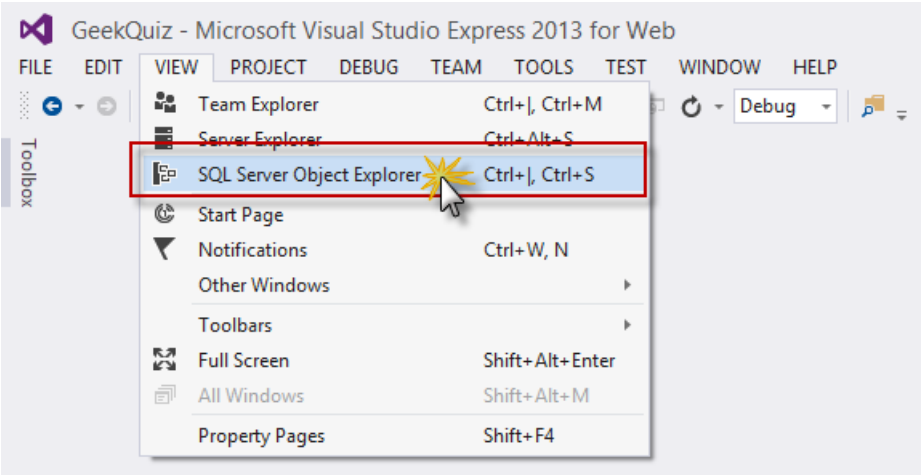
```
Update-Database -Verbose
```

```
Package Manager Console
Package source: nuget.org    Default project: GeekQuiz
PM> Update-Database -Verbose
Using Startup project 'GeekQuiz'.
Using NuGet project 'GeekQuiz'.
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Target database is: 'GeekQuizProd' (DataSource: (LocalDb)\v11.0, Provider: System.Data.SqlClient, Origin: Configuration).
Applying explicit migrations: [201402271418493_InitialSchema].
Applying explicit migration: 201402271418493_InitialSchema.
CREATE TABLE [dbo].[TriviaAnswers] (
    [QuestionId] [int] NOT NULL,
    [OptionId] [int] NOT NULL,
    [Id] [int] NOT NULL IDENTITY,
    [UserId] [nvarchar](max),
    CONSTRAINT [PK_dbo.TriviaAnswers] PRIMARY KEY ([Id])
)
CREATE INDEX [IX_QuestionId_OptionId] ON [dbo].[TriviaAnswers]([QuestionId], [OptionId])
CREATE TABLE [dbo].[TriviaOptions] (
    [QuestionId] [int] NOT NULL,
    [Id] [int] NOT NULL IDENTITY,
    [Title] [nvarchar](max) NOT NULL,
    [IsCorrect] [bit] NOT NULL,
    CONSTRAINT [PK_dbo.TriviaOptions] PRIMARY KEY ([QuestionId], [Id])
)
CREATE INDEX [IX_QuestionId] ON [dbo].[TriviaOptions]([QuestionId])
CREATE TABLE [dbo].[TriviaQuestions] (
    [Id] [int] NOT NULL IDENTITY,
    [Title] [nvarchar](max) NOT NULL,
    CONSTRAINT [PK_dbo.TriviaQuestions] PRIMARY KEY ([Id])
)
ALTER TABLE [dbo].[TriviaAnswers] ADD CONSTRAINT [FK_dbo.TriviaAnswers_dbo.TriviaOptions_QuestionId_OptionId] FOREIGN KEY
([QuestionId], [OptionId]) REFERENCES [dbo].[TriviaOptions] ([QuestionId], [Id]) ON DELETE CASCADE
ALTER TABLE [dbo].[TriviaOptions] ADD CONSTRAINT [FK_dbo.TriviaOptions_dbo.TriviaQuestions_QuestionId] FOREIGN KEY ([QuestionId])
REFERENCES [dbo].[TriviaQuestions] ([Id]) ON DELETE CASCADE
CREATE TABLE [dbo].[__MigrationHistory] (
    [MigrationId] [nvarchar](150) NOT NULL,
    [ContextKey] [nvarchar](300) NOT NULL,
    [Model] [varbinary](max) NOT NULL,
    [ProductVersion] [nvarchar](32) NOT NULL,
    CONSTRAINT [PK_dbo.__MigrationHistory] PRIMARY KEY ([MigrationId], [ContextKey])
)
100 %
```

Creating initial database

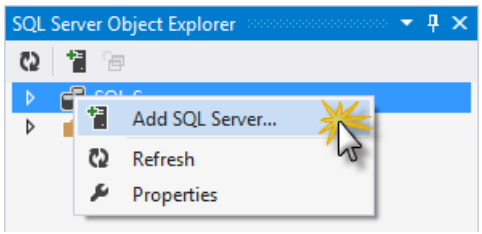
> **NOTE**
>
> **Update-Database** will apply any pending migrations to the database. In this case, it will create the database using the connection string defined in your web.config file.

8. Go to **View** menu and open **SQL Server Object Explorer**.



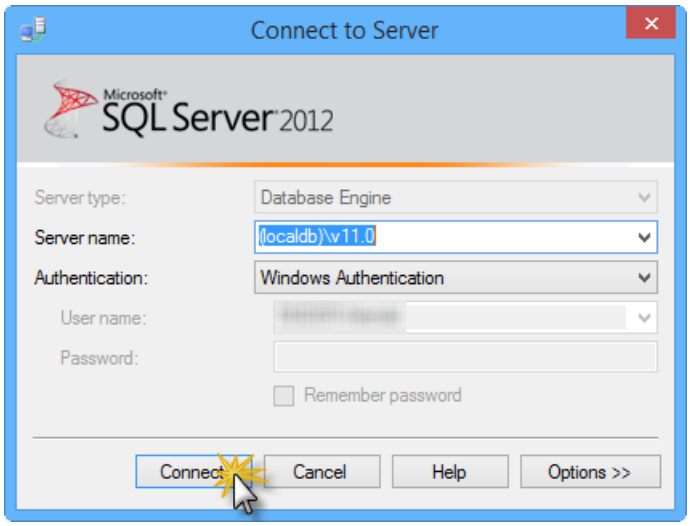
Open in SQL Server Object Explorer

9. In the **SQL Server Object Explorer** window, connect to your LocalDB instance by right-clicking the **SQL Server** node and selecting **Add SQL Server...** option.



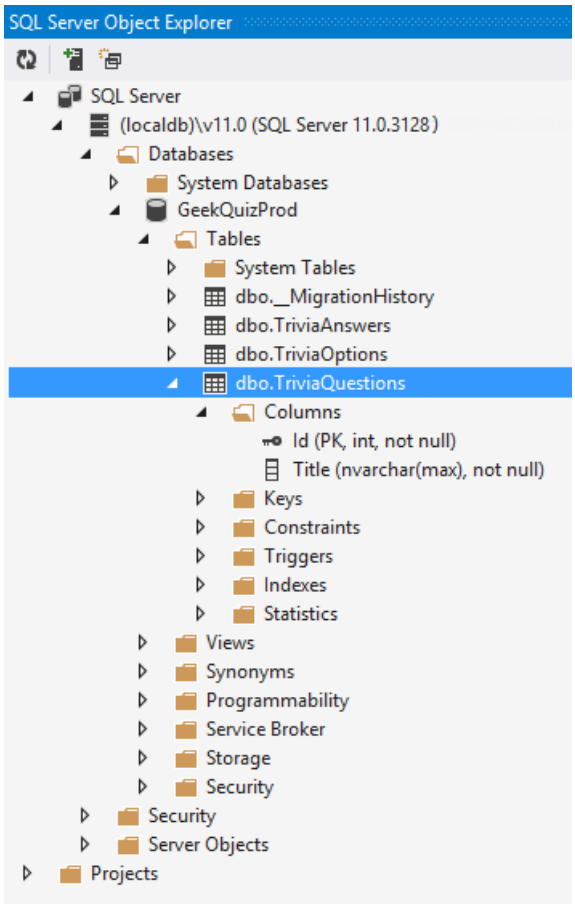
Adding a SQL Server instance to SQL Server Object Explorer

10. Set the **server name** to *(localdb)\v11.0* and leave **Windows Authentication** as your authentication mode. Click **Connect** to continue.



Connecting to LocalDB

11. Open the **GeekQuizProd** database and expand the **Tables** node. As you can see, the **Update-Database** command generated all the tables defined in the **TriviaContext** class. Locate the **dbo.TriviaQuestions** table and open the columns node. In the next task, you will add a new column to this table and update the database using **Migrations**.



Trivia Questions Columns

Task 2 – Updating Database Schema Using Migrations

In this task, you will use **Entity Framework Code First Migrations** to detect a change in your model and generate the necessary code to update the database. You will update the **TriviaQuestions** entity by adding a new

property to it. Then you will run commands to create a new Migration to include the new column in the table.

1. In **Solution Explorer**, double-click the **TriviaQuestion.cs** file located inside the **Models** folder.
2. Add a new property named **Hint**, as shown in the following code snippet.

```
public class TriviaQuestion
{
     public int Id { get; set; }

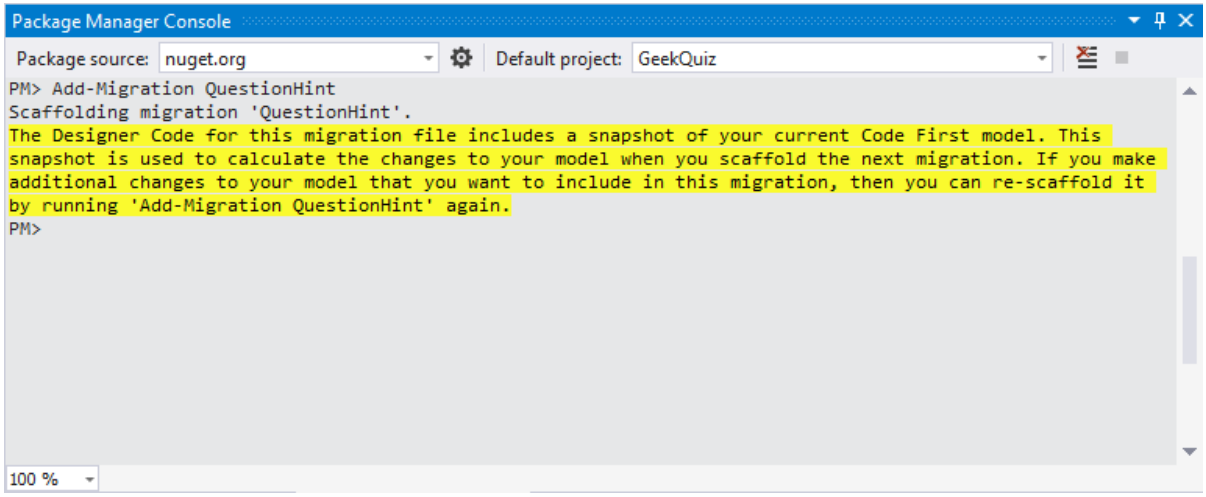
     [Required]
     public string Title { get; set; }

     public virtual List<TriviaOption> Options { get; set; }

     public string Hint { get; set; }
}
```

3. In the **Package Manager Console**, enter the following command and then press **Enter**. A new migration will be created reflecting the change in our model.

```
Add-Migration QuestionHint
```



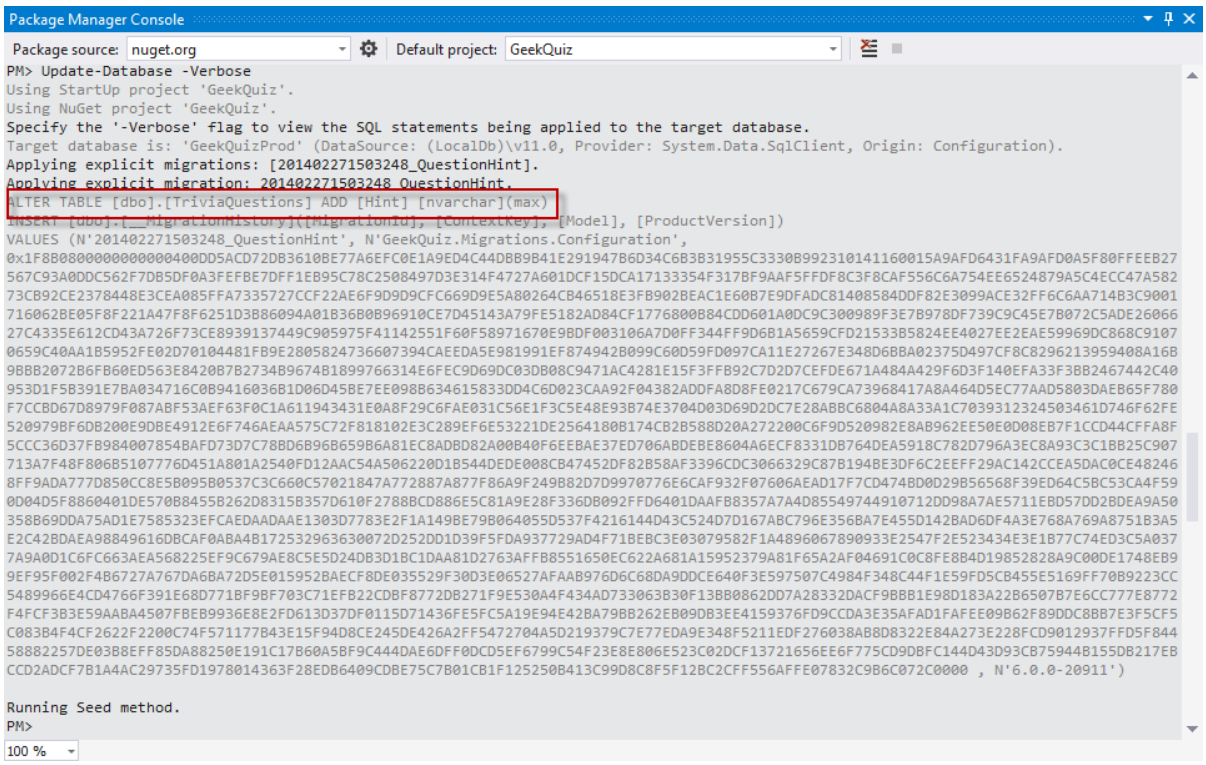
Add-Migration

> **NOTE**
>
> A Migration file is composed of two methods, **Up** and **Down**.
>
> - The **Up** method will be used to specify what changes the current version of our application need to apply to the database.
> - The **Down** is used to reverse the changes we have added to the **Up** method.
>
> When the Database Migration updates the database, it will run all migrations in the timestamp order, and only those that have not been used since the last update (The _MigrationHistory table keeps track of which migrations have been applied). The **Up** method of all migrations will be called and will make the changes we have specified to the database. If we decide to go back to a previous migration, the **Down** method will be called to redo the changes in a reverse order.

4. In the **Package Manager Console**, enter the following command and then press **Enter**.

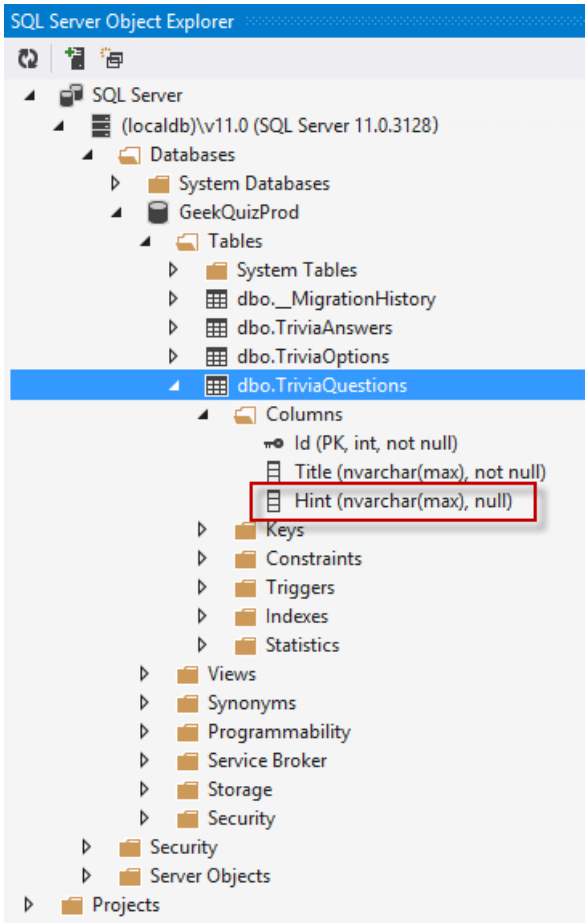
```
Update-Database -Verbose
```

5. The output of the **Update-Database** command generated an **Alter Table** SQL statement to add a new column to the **TriviaQuestions** table, as shown in the image below.



Add column SQL statement generated

6. In **SQL Server Object Explorer**, refresh the **dbo.TriviaQuestions** table and check that the new **Hint** column is displayed.



Showing the new Hint Column

7. Back in the **TriviaQuestion.cs** editor, add a **StringLength** constraint to the *Hint* property, as shown in the following code snippet.

```
public class TriviaQuestion
{
     public int Id { get; set; }

     [Required]
     public string Title { get; set; }

     public virtual List<TriviaOption> Options { get; set; }

     [StringLength(150)]
     public string Hint { get; set; }
}
```

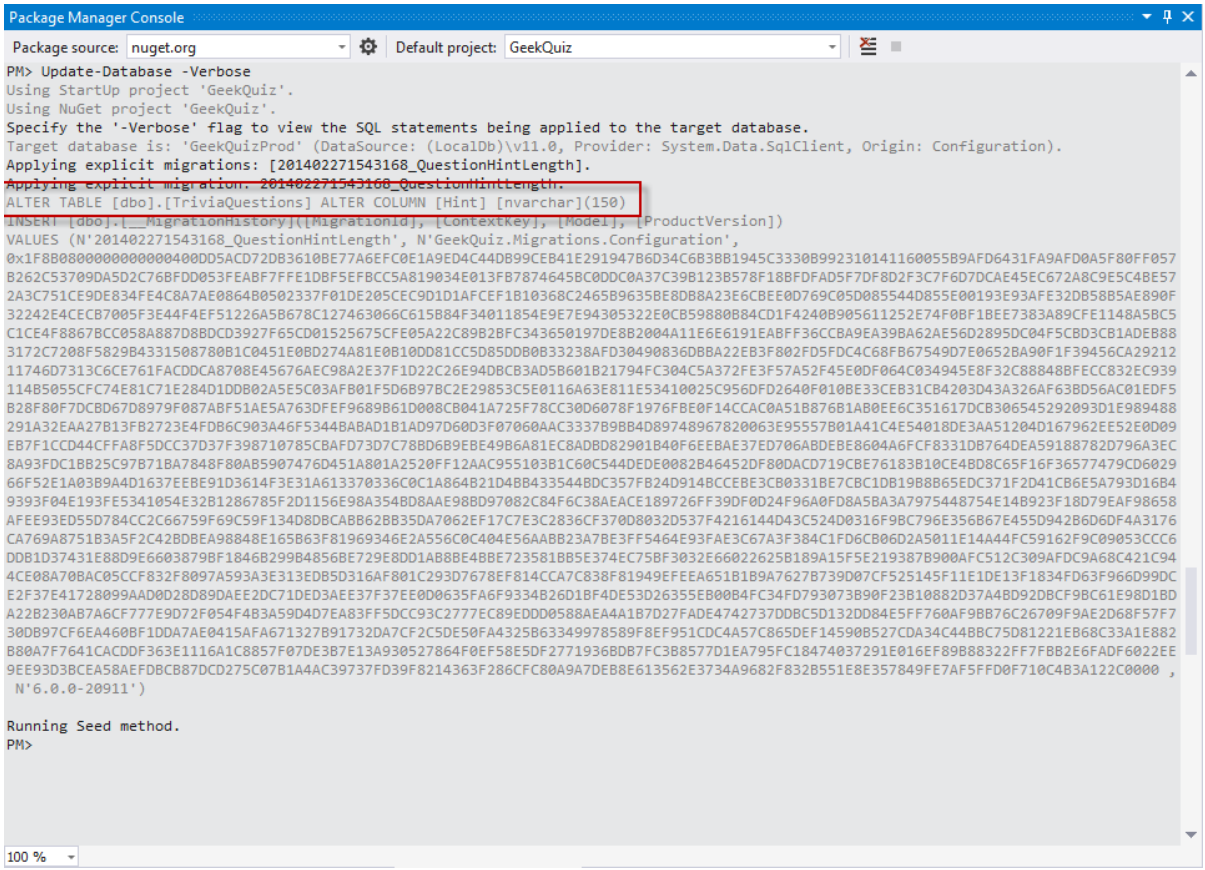
8. In the **Package Manager Console**, enter the following command and then press **Enter**.

```
Add-Migration QuestionHintLength
```

9. In the **Package Manager Console**, enter the following command and then press **Enter**.

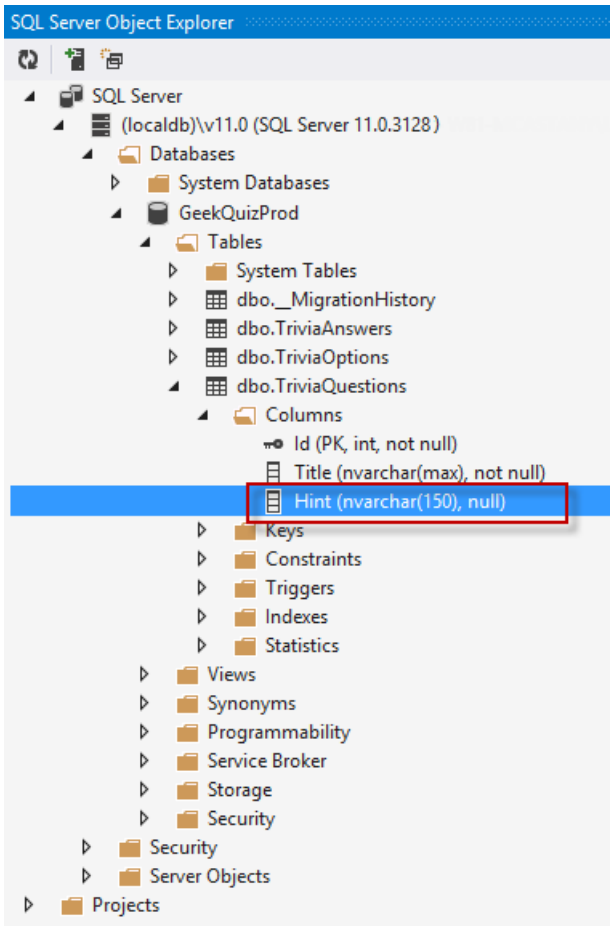
```
Update-Database -Verbose
```

10. The output of the **Update-Database** command generated an **Alter Table** SQL statement to update the *hint* column type of the **TriviaQuestions** table, as shown in the image below.



Alter column SQL statement generated

11. In **SQL Server Object Explorer**, refresh the **dbo.TriviaQuestions** table and check that the **Hint** column type is **nvarchar(150)**.



Showing the new constraint

Exercise 2: Deploying a Web App to Staging

Web Apps in Azure App Service enables you to perform staged publishing. Staged publishing creates a staging site slot for each default production site and enables you to swap these slots with no down time. This is really useful to validate changes before releasing to the public, incrementally integrate site content, and roll back if changes are not working as expected.

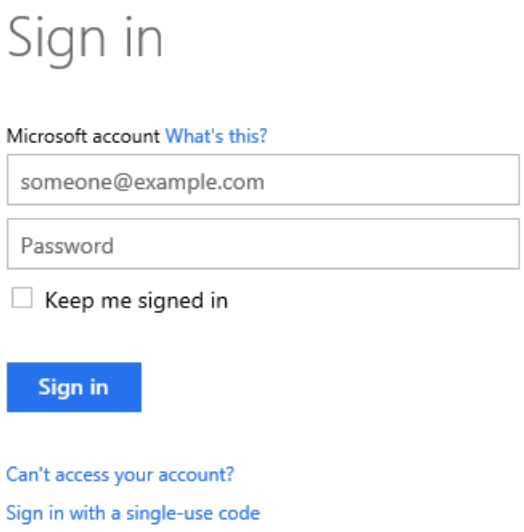
In this exercise, you will deploy the **Geek Quiz** application to the staging environment of your web app using Git source control. To do this, you will create the web app and provision the required components at the management portal, configure a **Git** repository and push the application source code from your local computer to the staging slot. You will also update your production database with the **Code First Migrations** you created in the previous exercise. You will then execute the application in this test environment to verify its operation. Once you are satisfied that it is working according to your expectations, you will promote the application to production.

> **NOTE**
>
> To enable staged publishing, the web app must be in **Standard mode**. Note that additional charges will be incurred if you change your web app to Standard mode. For more information about pricing, see App Service Pricing.

Task 1 – Creating a Web App in Azure App Service

In this task, you will create a web app in **Azure App Service** from the management portal. You will also configure a **SQL Database** to persist the application data, and configure a local Git repository for source control.

1. Go to the Azure management portal and sign in using the Microsoft account associated with your subscription.

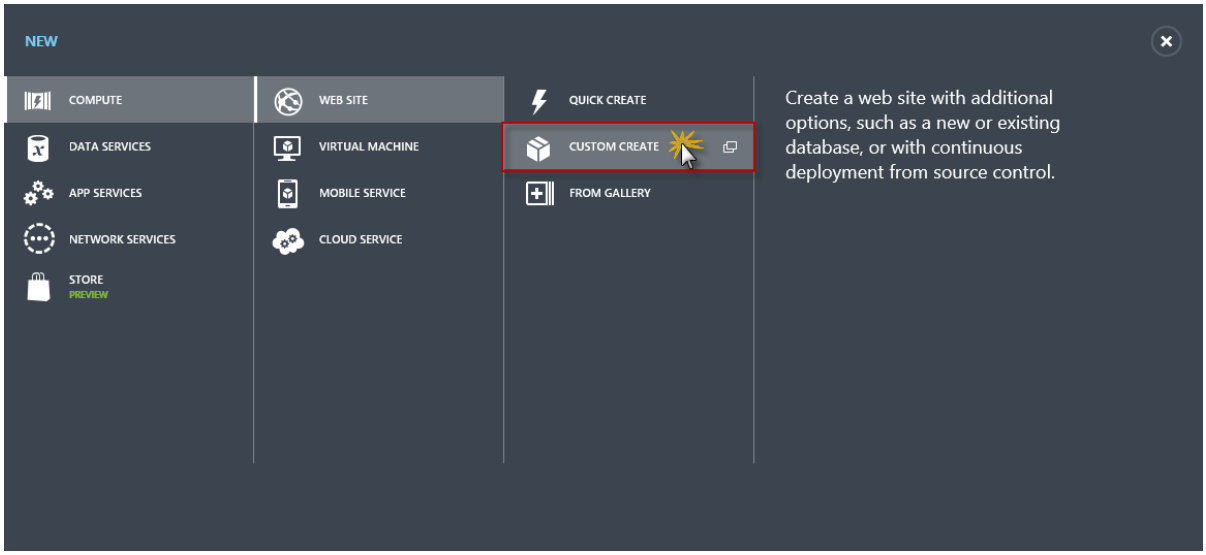


Sign in to the Azure management portal

2. Click **New** in the command bar at the bottom of the page.

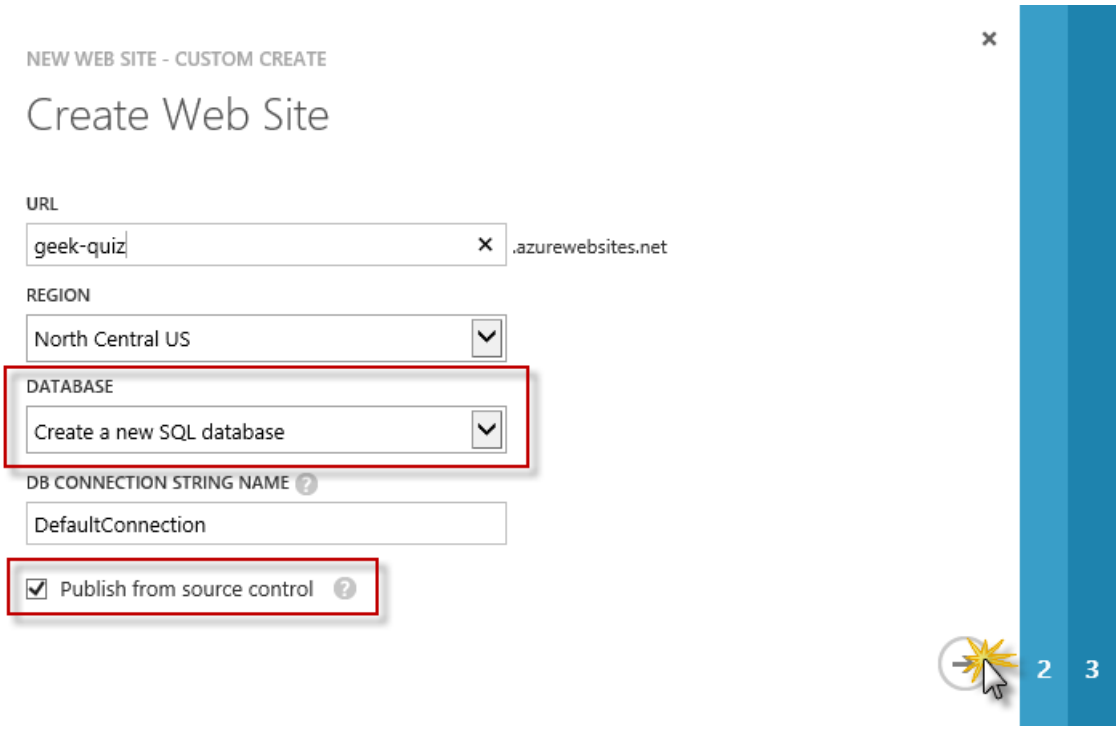
Creating a new web app

3. Click **Compute**, **Website** and then **Custom Create**.



Creating a new web app using Custom Create

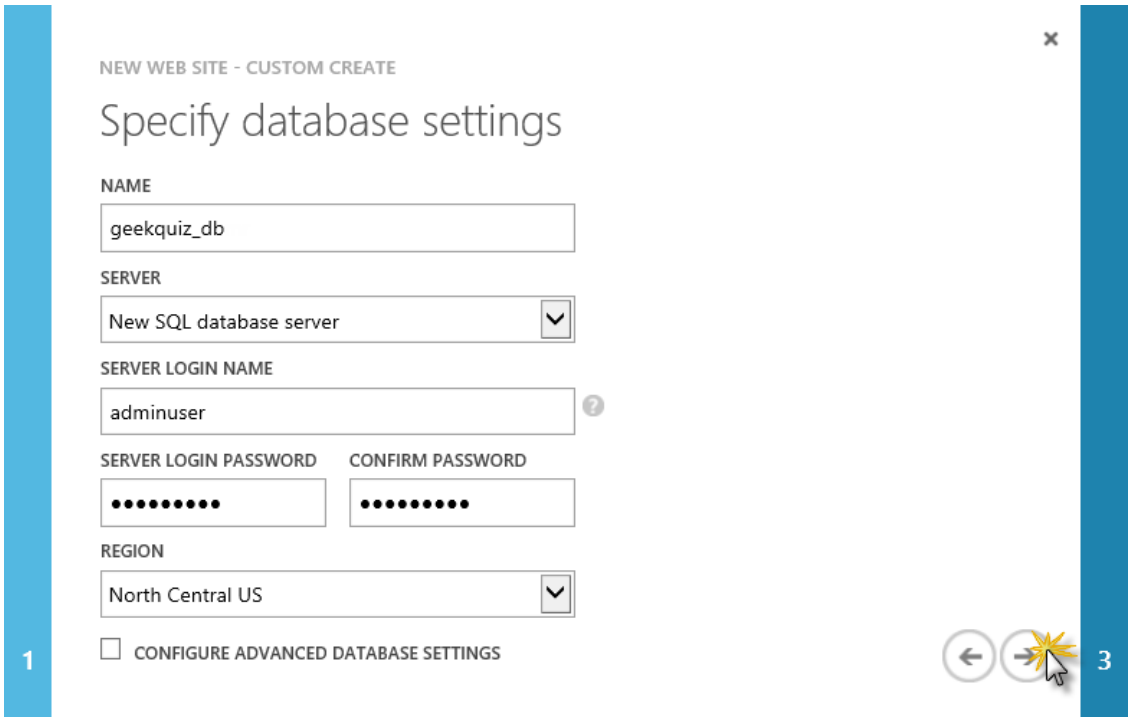
4. In the **New Website - Custom Create** dialog box, provide an available **URL** (e.g. *geek-quiz*), select a location in the **Region** drop-down list, and select **Create a new SQL database** in the **Database** drop-down list. Finally, select the **Publish from source control** check box and click **Next**.



Customizing the new web app

5. Specify the following information for the database settings:

   - In the **Name** text box, enter a database name (e.g. *geekquiz_db*)
   - In the Server **drop-down** list, select **New SQL database server**. Alternatively, you can select an existing server.
   - In the **Database username** and **Database password** boxes, enter the administrator username and password for the SQL database server. If you select a server you have already created, you will be prompted for the password.

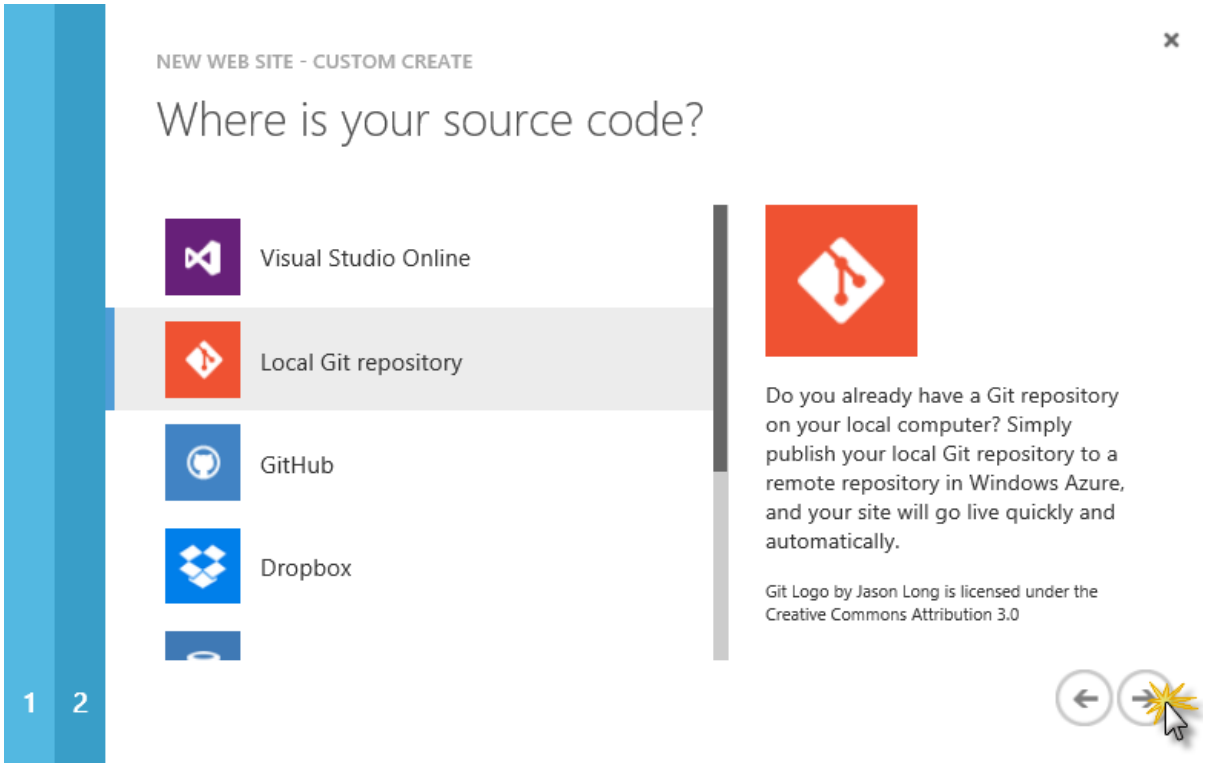


Specifying the database settings

6. Click **Next** to continue.

7. Select **Local Git repository** for the source control to use and click **Next**.

> **NOTE**
>
> You may be prompted for the deployment credentials (a username and password).



Creating the Git repository

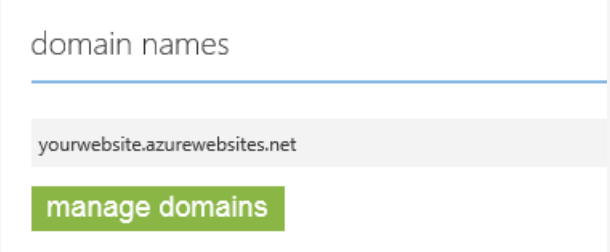
8. Wait until the new web app is created.

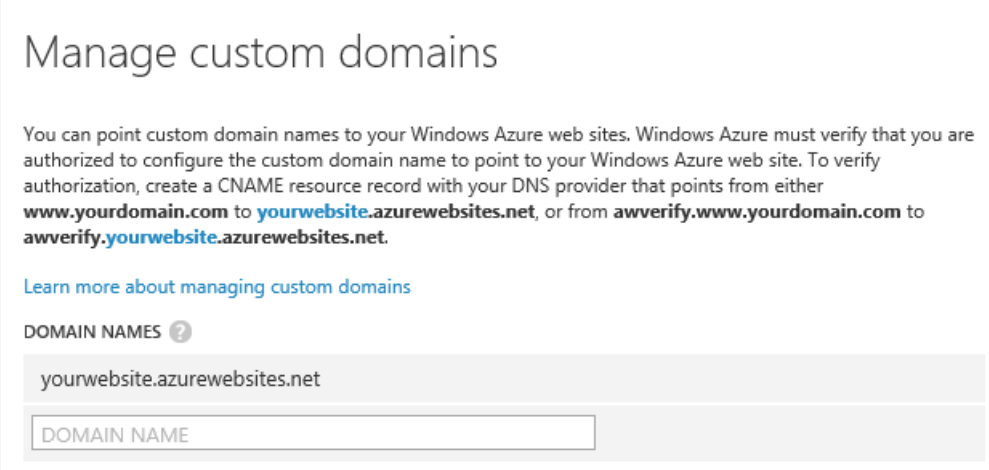
NOTE

By default, Azure provides domains at *azurewebsites.net* but also gives you the possibility to set custom domains using the Azure management portal. However, you can only manage custom domains if you are using certain Azure App Service modes.

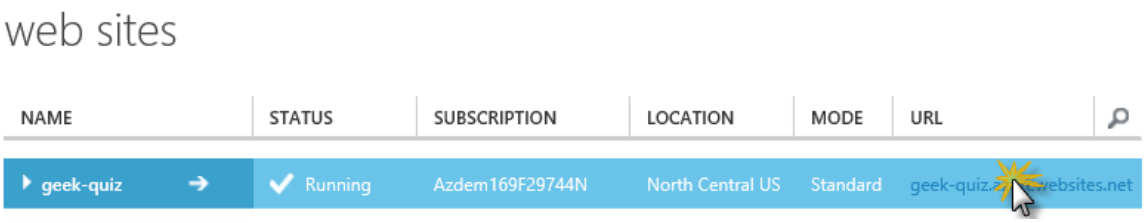
Azure App Service is available in Free, Shared, Basic, Standard, and Premium editions. In Free and Shared mode, all web apps run in a multi-tenant environment and have quotas for CPU, Memory, and Network usage. The maximum number of free apps may vary with your plan. In Standard mode, you choose which apps run on dedicated virtual machines that correspond to the standard Azure compute resources. You can find the web app mode configuration in the **Scale** menu of your web app.

If you are using **Shared** or **Standard** mode, you will be able to manage custom domains for your web app by going to your app's **Configure** menu and clicking **Manage Domains** under *domain names*.





9. Once the web app is created, click the link under the **URL** column to check that the new web app is running.



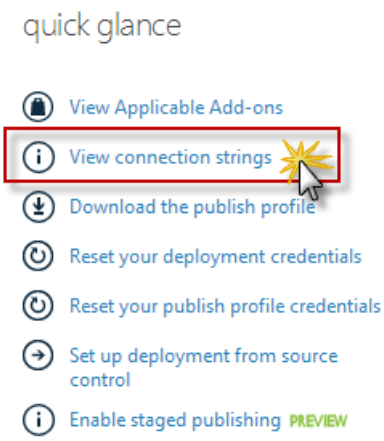
Browsing to the new web app

web app running

Task 2 – Creating the Production SQL Database

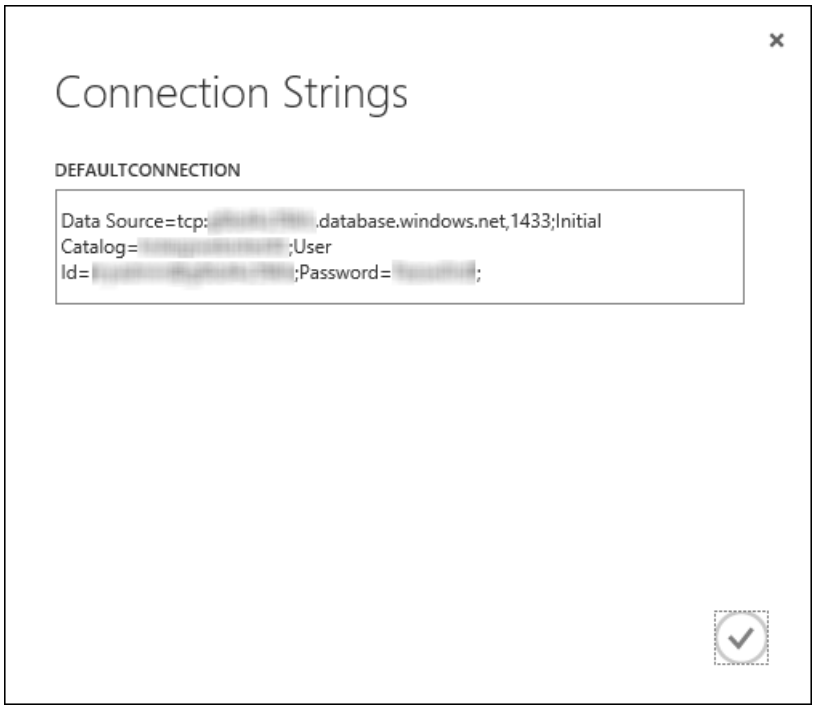
In this task, you will use the **Entity Framework Code First Migrations** to create the database targeting the **Azure SQL Database** instance you created in the previous task.

1. In the Management Portal, navigate to the web app you created in the previous task and go to its **Dashboard**.
2. In the **Dashboard** page, click **View connection strings** link under the **quick glance** section.



View connection strings

3. Copy the **connection string** value and close the dialog box.

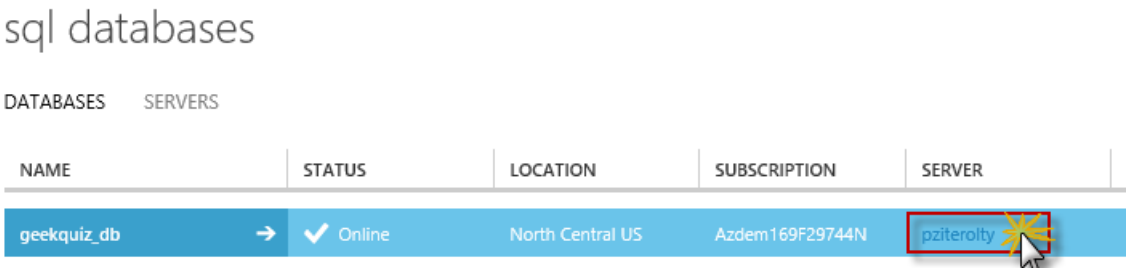


Connection String in Azure Management Portal

4. Click **SQL Databases** to see the list of SQL databases in Azure

SQL Database menu

5. Locate the database you created in the previous task and click on the Server.

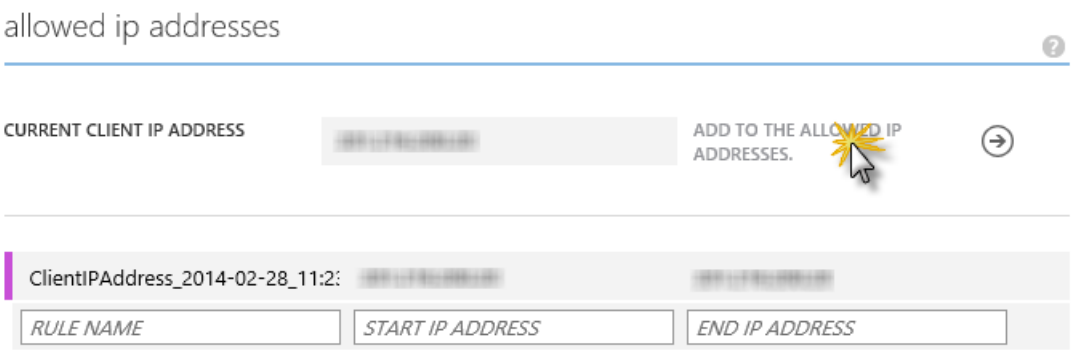


SQL Database server

6. In the **Quick Start** page of the server, click on **Configure**.

Configure menu

7. In the **Allowed IP addresses** section, click on **Add to the allowed IP addresses** link to enable your IP to connect to the SQL Database server.



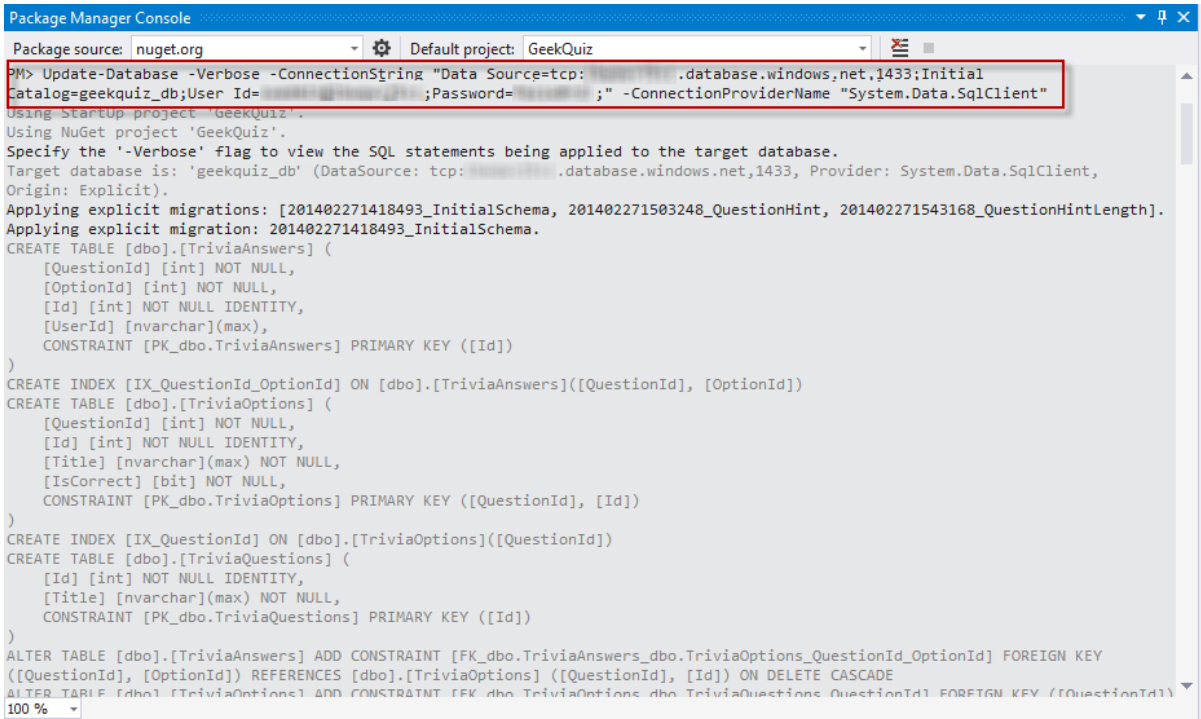
Allowed IP addresses

8. Click **Save** at the bottom of the page to complete the step.

9. Switch back to Visual Studio.

10. In the **Package Manager Console**, execute the following command replacing *[YOUR-CONNECTION-STRING]* placeholder with the connection string you copied from Azure

```
Update-Database -Verbose -ConnectionString "[YOUR-CONNECTION-STRING]" -ConnectionProviderName
"System.Data.SqlClient"
```

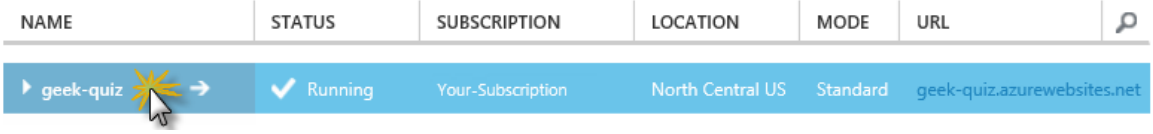


Update database targeting Azure SQL Database

Task 3 – Deploying Geek Quiz to Staging Using Git

In this task, you will enable staged publishing in your web app. Then, you will use Git to publish the Geek Quiz application directly from your local computer to the staging environment of your web app.

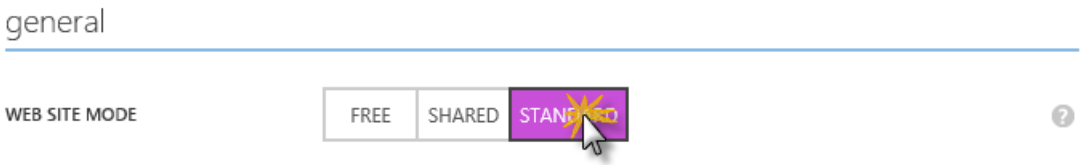
1. Go back to the portal and click the name of the web app under the **Name** column to display the management pages.



Opening the web app management pages

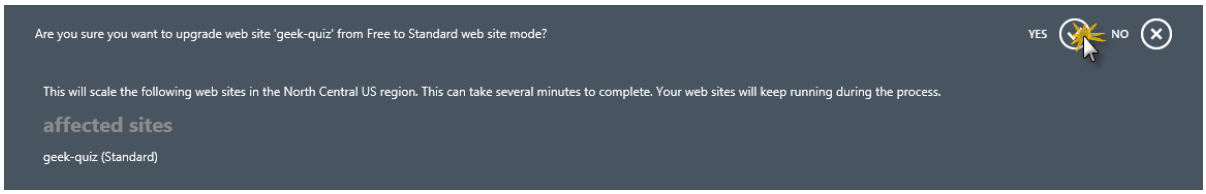
2. Navigate to the **Scale** page. Under the **general** section, select **Standard** for the configuration and click **Save** in the command bar.

> **NOTE**
>
> To run all web apps in the current region and subscription in **Standard** mode, leave the **Select All** check box selected in the **Choose Sites** configuration. Otherwise, clear the **Select All** check box.



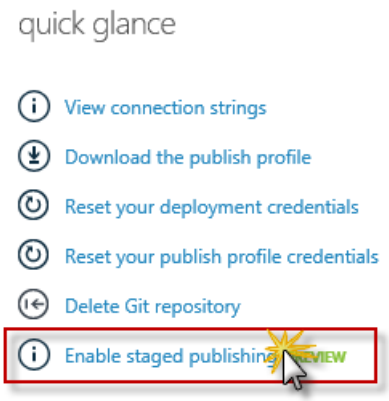
Upgrading the Web App to Standard mode

3. Click **Yes** to confirm the changes.



Confirming the change to Standard mode

4. Go to the **Dashboard** page and click **Enable staged publishing** under the **quick glance** section.

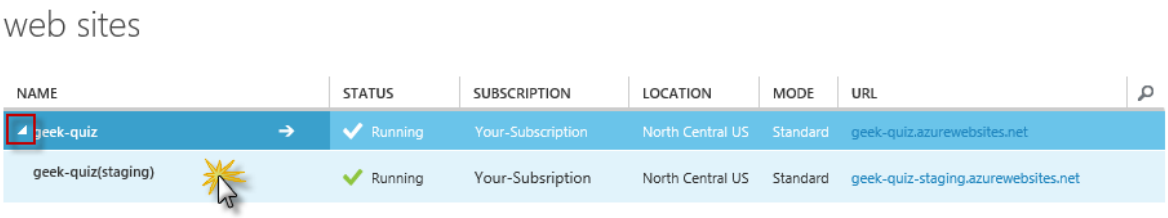


Enabling staged publishing

5. Click **Yes** to enable staged publishing.

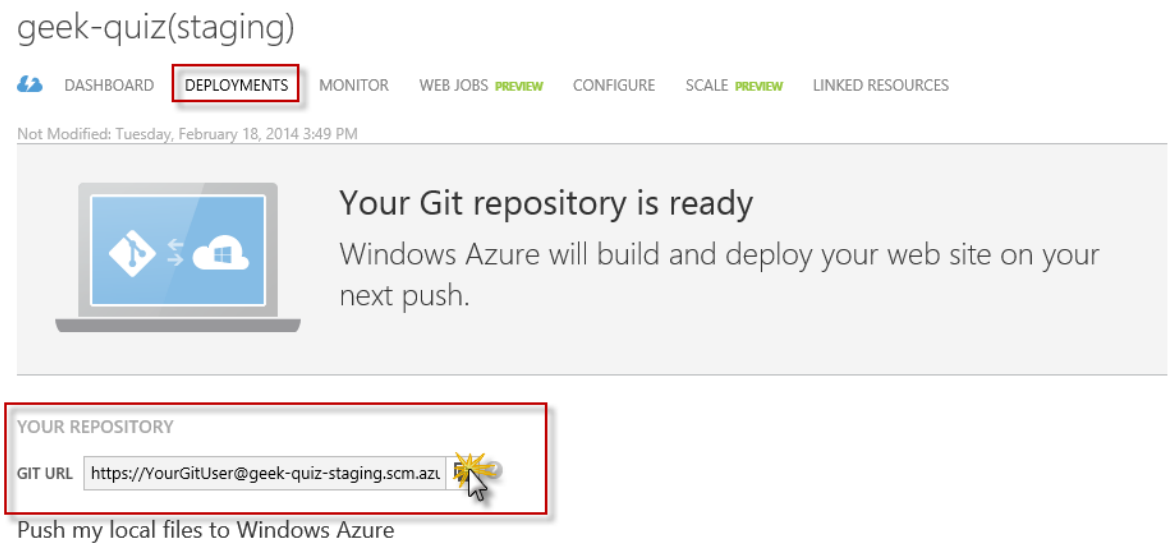
Confirming staged publishing

6. In the list of web apps, expand the mark to the left of your web app name to display the staging site slot. It has the name of your web app followed by ***(staging)***. Click the staging site to go to the management page.



Navigating to the staging app

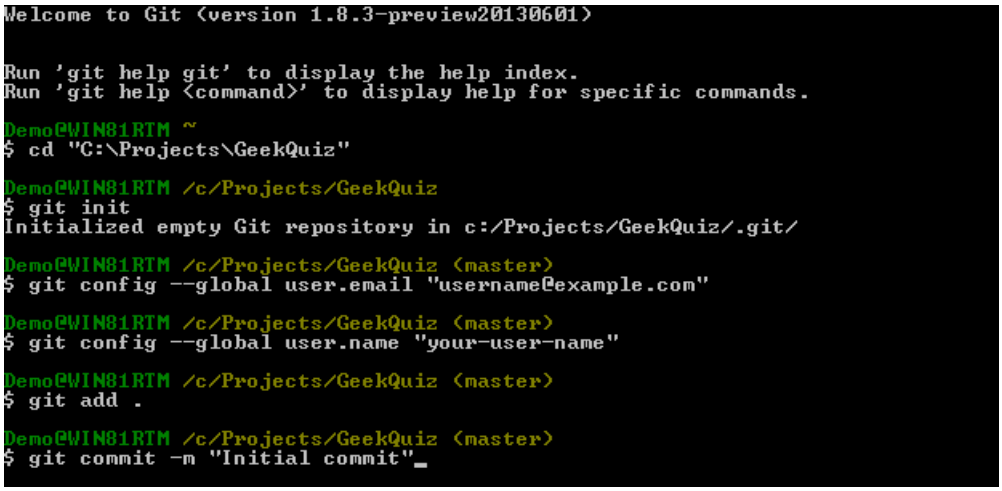
7. Notice that he management page looks like any other web app's management page. Navigate to the **Deployments** page and copy the **Git URL** value. You will use it later in this exercise.



Copying the Git URL value

8. Open a new **Git Bash** console and execute the following commands. Update the *[YOUR-APPLICATION-PATH]* placeholder with the path to the **GeekQuiz** solution, located in the **Source\Ex1-DeployingWebSiteToStaging\Begin** folder of this lab.

```
cd "[YOUR-APPLICATION-PATH]"
git init
git config --global user.email "{username@example.com}"
git config --global user.name "{your-user-name}"
git add .
git commit -m "Initial commit"
```



Git initialization and first commit

9. Run the following command to push your web app to the remote **Git** repository. Replace the placeholder with the URL you obtained from the management portal. You will be prompted for your deployment password.

```
git remote add azure [GIT-CLONE-URL]
git push azure master
```

```
create mode 100644 GeekQuiz/Views/Home/Contact.cshtml
create mode 100644 GeekQuiz/Views/Home/Index.cshtml
create mode 100644 GeekQuiz/Views/Shared/Error.cshtml
create mode 100644 GeekQuiz/Views/Shared/_Layout.cshtml
create mode 100644 GeekQuiz/Views/Shared/_LoginPartial.cshtml
create mode 100644 GeekQuiz/Views/Web.config
create mode 100644 GeekQuiz/Views/_ViewStart.cshtml
create mode 100644 GeekQuiz/Web.Debug.config
create mode 100644 GeekQuiz/Web.Release.config
create mode 100644 GeekQuiz/Web.config
create mode 100644 GeekQuiz/favicon.ico
create mode 100644 GeekQuiz/fonts/glyphicons-halflings-regular.eot
create mode 100644 GeekQuiz/fonts/glyphicons-halflings-regular.svg
create mode 100644 GeekQuiz/fonts/glyphicons-halflings-regular.ttf
create mode 100644 GeekQuiz/fonts/glyphicons-halflings-regular.woff
create mode 100644 GeekQuiz/packages.config
create mode 100644 Settings.StyleCop

Demo@WIN81RTM /c/Projects/GeekQuiz (master)
$ git remote add azure https://admin01@geek-quiz-staging.scm.azurewebsites.ne
t:443/geek-quiz.git

Demo@WIN81RTM /c/Projects/GeekQuiz (master)
$ git push azure master
Password for 'https://admin01@geek-quiz-staging.scm.azurewebsites.net:443'
```

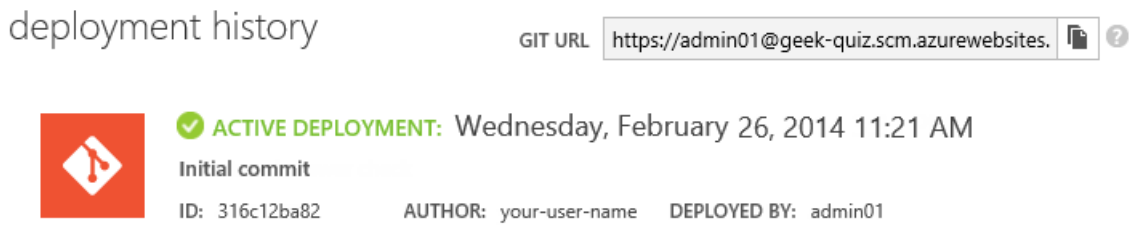
Pushing to Azure

> **NOTE**
>
> When you deploy content to the FTP host or GIT repository of a web app, you must authenticate using the **deployment credentials** that you created from the web app's **Quick Start** or **Dashboard** management pages. If you do not know your deployment credentials you can easily reset them using the management portal. Open the web app **Dashboard** page and click the **Reset your deployment credentials** link. Provide a new password and click **OK**. Deployment credentials are valid for use with all web apps associated with your subscription.

10. In order to verify the web app was successfully pushed to Azure, go back to the management portal and click **Websites**.

11. Locate your web app and expand the entry to display the staging site slot. Click its **Name** to go to the management page.

12. Click **Deployments** to see the **deployment history**. Verify that there is an **Active Deployment** with your *"Initial Commit"*.



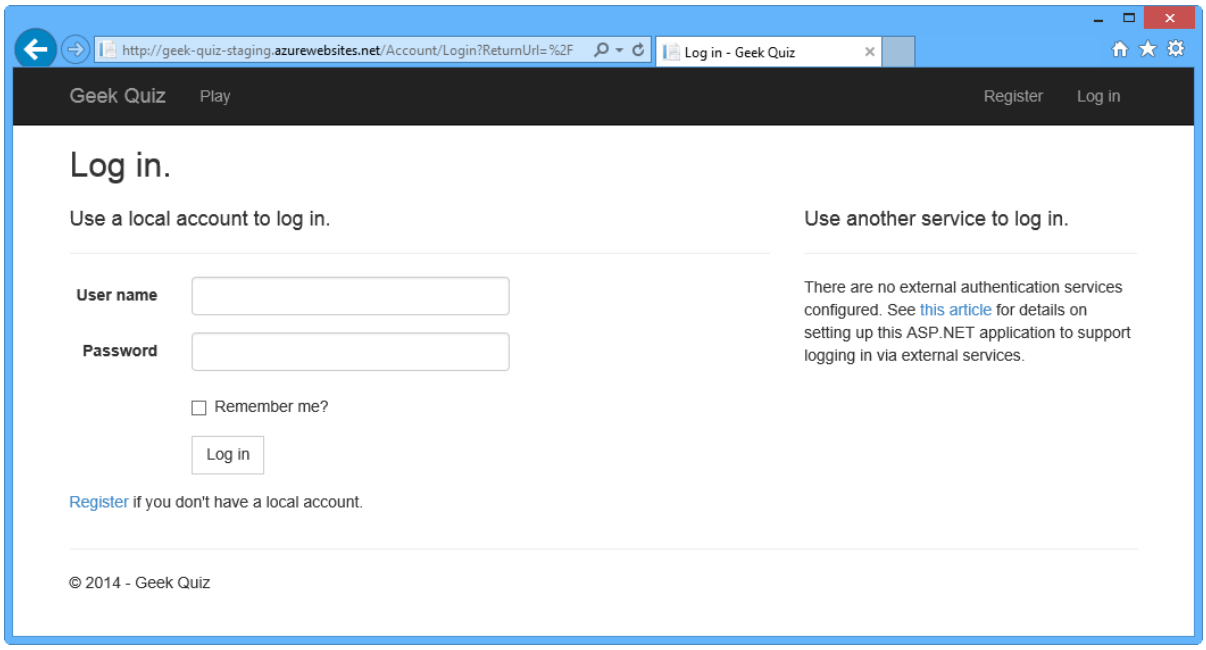
Active deployment

13. Finally, click **Browse** in the command bar to go to the web app.

Browse web app

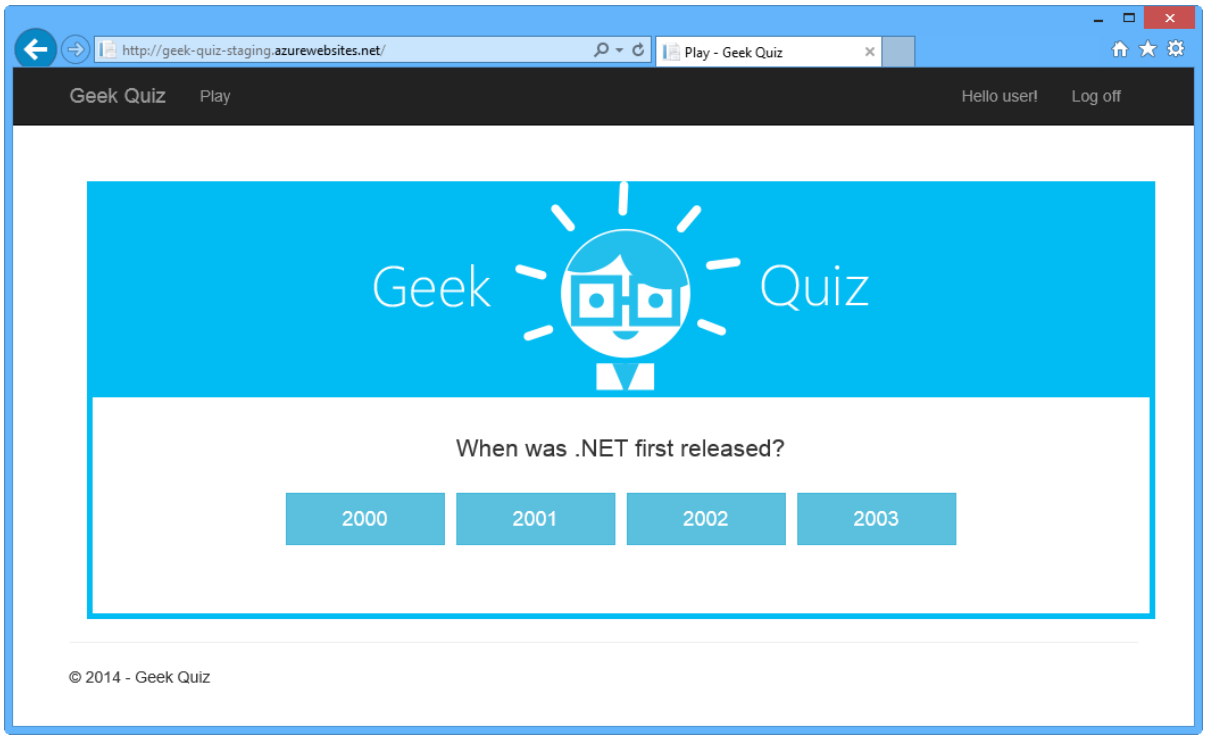
14. If the application is successfully deployed, you will see the Geek Quiz login page.

> **NOTE**
>
> The address URL of the deployed application contains the name of your web app followed by *-staging*.



Application running in the staging environment

15. If you wish to explore the application, click **Register** to register a new user. Complete the account details by entering a user name, email address and password. Next, the application shows the first question of the quiz. Answer a few questions to make sure it is working as expected.



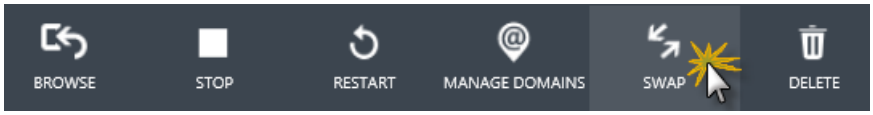
Application ready to be used

Task 4 – Promoting the Web App to Production

Now that you have verified that the web app is working correctly in the staging environment, you are ready to

promote it to production. In this task, you will swap the staging site slot with the production site slot.

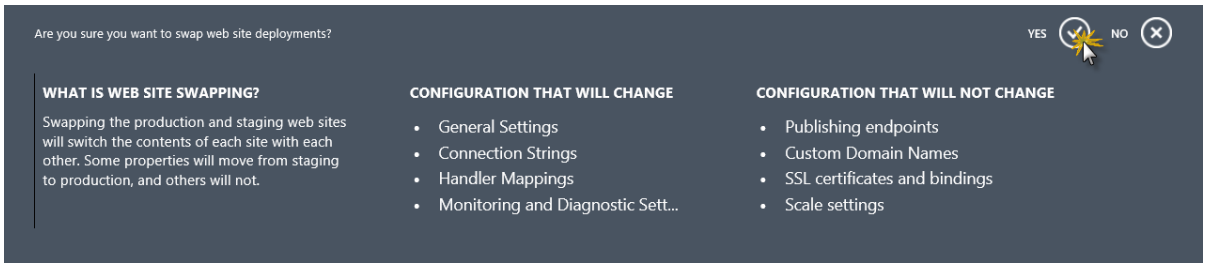
1. Go back to the management portal and select the staging site slot. Click **Swap** in the command bar.



Swap to production

2. Click **Yes** in the confirmation dialog box to proceed with the swap operation. Azure will immediately swap the content of the production site with the content of the staging site.

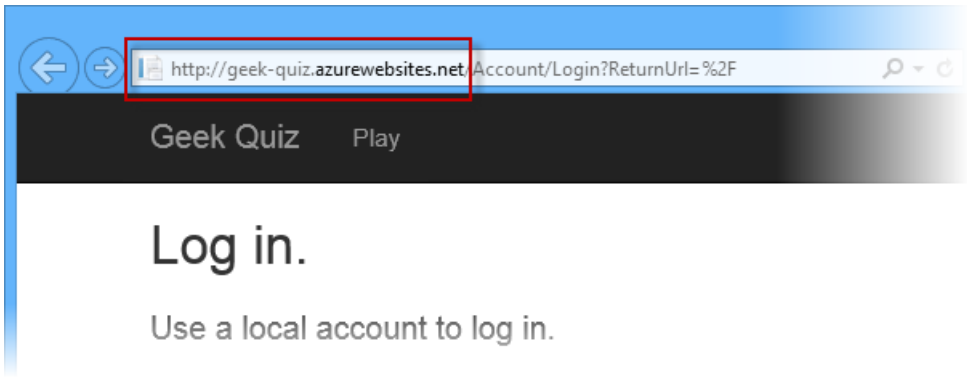
> **NOTE**
>
> Some settings from the staged version will automatically be copied to the production version (e.g. connection string overrides, handler mappings, etc.) but other settings will not change (e.g. DNS endpoints, SSL bindings, etc.).



Confirming swap operation

3. Once the swap is complete, select the production slot and click **Browse** in the command bar to open the production site. Notice the URL in the address bar.

> **NOTE**
>
> You might need to refresh your browser to clear cache. In Internet Explorer, you can do this by pressing **CTRL+R**.



4. In the **GitBash** console, update the remote URL for the local Git repository to target the production slot. To do this, run the following command replacing the placeholders with your deployment username and the name of your web app.

> **NOTE**
>
> In the following exercises, you will push changes to the production site instead of staging just for the simplicity of the lab. In a real-world scenario, it is recommended to verify the changes in the staging environment before promoting to production.

```
git remote set-url azure https://<your-user>@<your-web-site>.scm.azurewebsites.net:443/<your-web-site>.git
```

Exercise 3: Performing Deployment Rollback in Production

There are scenarios where you do not have a staging slot to perform hot swap between staging and production, for example, if you are working with **Free** or **Shared** mode. In those scenarios, you should test your application in a testing environment –either locally or in a remote site– before deploying to production. However, it is possible that an issue not detected during the testing phase may arise in the production site. In this case, it is important to have a mechanism to easily switch to a previous and more stable version of the application as quickly as possible.

In **Azure App Service**, continuous deployment from source control makes this possible thanks to the **redeploy** action available in the management portal. Azure keeps track of the deployments associated with the commits pushed to the repository and provides an option to redeploy your application using any of your previous deployments, at any time.

In this exercise you will perform a change to the code in the **Geek Quiz** application that intentionally injects a *bug*. You will deploy the application to production to see the error, and then you will take advantage of the redeploy feature to go back to the previous state.

Task 1 – Updating the Geek Quiz Application

In this task, you will refactor a small piece of code of the **TriviaController** class to extract part of the logic that retrieves the selected quiz option from the database into a new method.

1. Switch to the Visual Studio instance with the **GeekQuiz** solution from the previous exercise.
2. In **Solution Explorer**, open the **TriviaController.cs** file inside the **Controllers** folder.
3. Locate the **StoreAsync** method and select the code highlighted in the following figure.

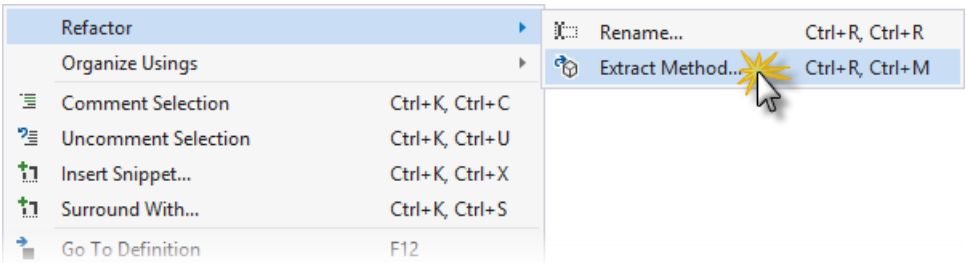
```
private async Task<bool> StoreAsync(TriviaAnswer answer)
{
    this.db.TriviaAnswers.Add(answer);

    await this.db.SaveChangesAsync();
    var selectedOption = await this.db.TriviaOptions.FirstOrDefaultAsync(o => o.Id == answer.OptionId
        && o.QuestionId == answer.QuestionId);

    return selectedOption.IsCorrect;
}
```

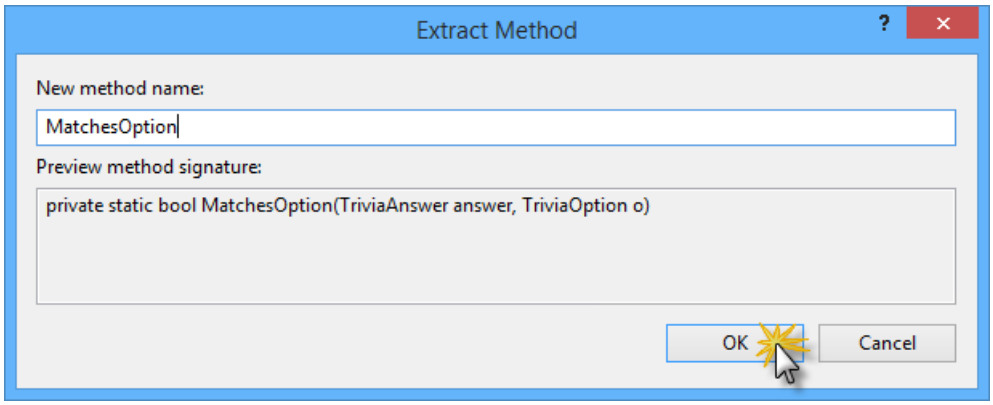
Selecting the code

4. Right-click the selected code, expand the **Refactor** menu and select **Extract Method...**.



Selecting Extract Method

5. In the **Extract Method** dialog box, name the new method *MatchesOption* and click **OK**.



Specifying the name for the extracted method

6. The selected code is then extracted into the **MatchesOption** method. The resulting code is shown in the following snippet.

```
private async Task<bool> StoreAsync(TriviaAnswer answer)
{
    this.db.TriviaAnswers.Add(answer);

    await this.db.SaveChangesAsync();
    var selectedOption = await this.db.TriviaOptions.FirstOrDefaultAsync(o => MatchesOption(answer, o));

    return selectedOption.IsCorrect;
}

private static bool MatchesOption(TriviaAnswer answer, TriviaOption o)
{
    return o.Id == answer.OptionId
            && o.QuestionId == answer.QuestionId;
}
```

7. Press **CTRL + S** to save the changes.

Task 2 – Redeploying the Geek Quiz Application

You will now push the changes you made in the previous task to the repository, which will trigger a new deployment to the production environment. Then, you will troubleshot an issue using the **F12 development tools** provided by Internet Explorer, and then perform a rollback to the previous deployment from the Azure management portal.

1. Open a new **Git Bash** console to deploy the updated application to Azure App Service.

2. Execute the following commands to push the changes to Azure. Update the *[YOUR-APPLICATION-PATH]* placeholder with the path to the **GeekQuiz** solution. You will be prompted for your deployment password.

```
cd "[YOUR-APPLICATION-PATH]"
git add .
git commit -m "Refactored answer check"
git push azure master
```

```
Demo@WIN81RTM /c/Projects/GeekQuiz (master)
$ git add .

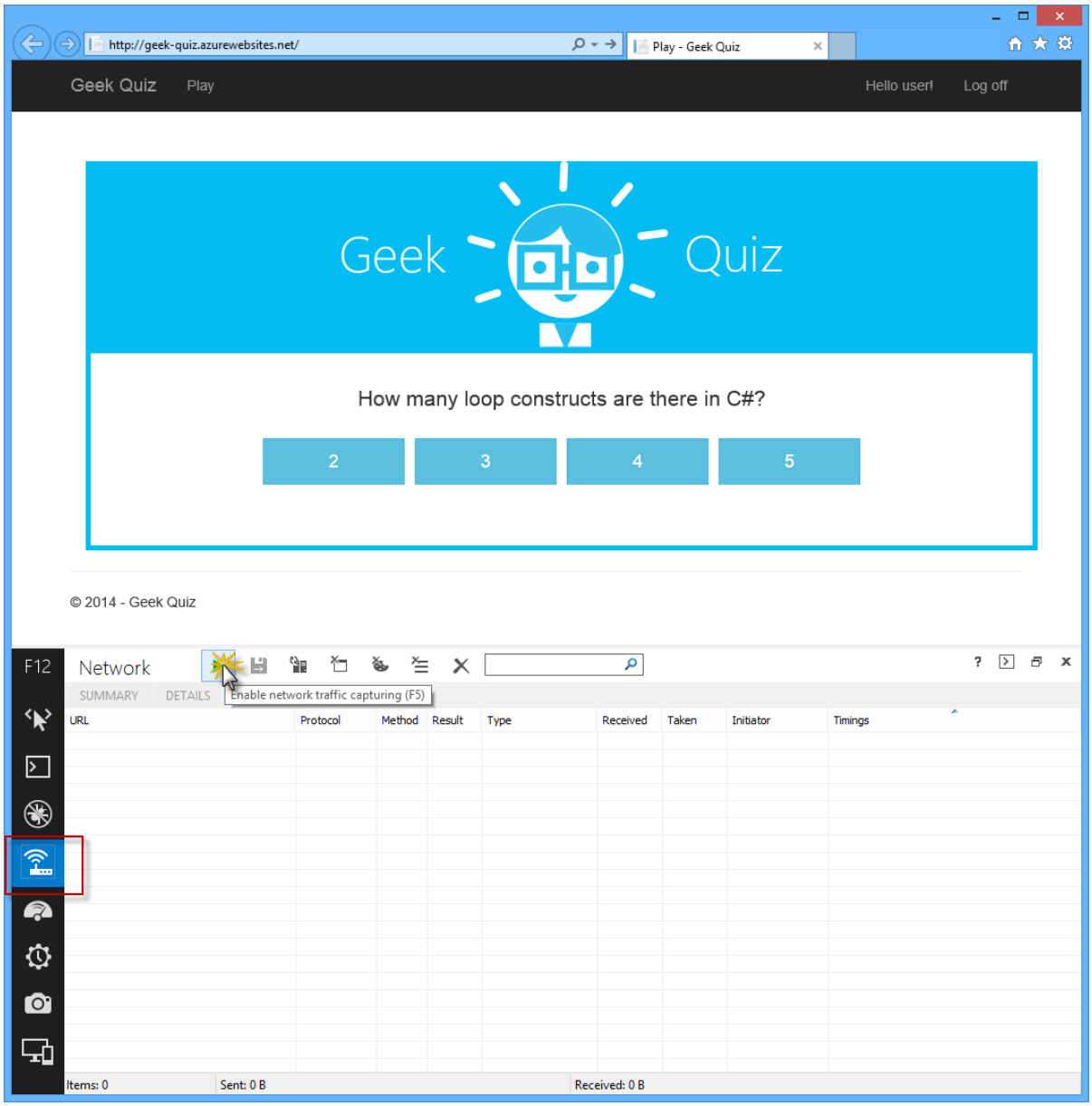
Demo@WIN81RTM /c/Projects/GeekQuiz (master)
$ git commit -m "Refactored answer check"
[master c7a7de7] Refactored answer check
 1 file changed, 7 insertions(+), 2 deletions(-)

Demo@WIN81RTM /c/Projects/GeekQuiz (master)
$ git push azure master
Password for 'https://admin01@geek-quiz.scm.azurewebsites.net:443': _
```

Pushing refactored code to Azure

3. Open Internet Explorer and navigate to your web app (e.g. `http://<your-web-site>.azurewebsites.net`). Log in using the previously created credentials.

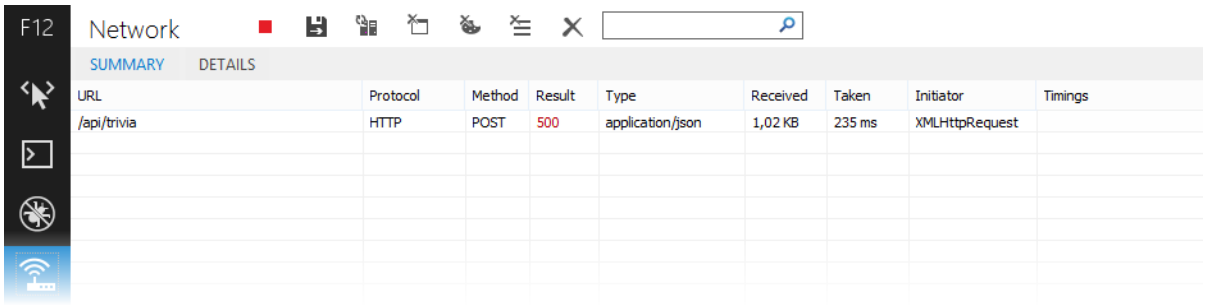
4. Press **F12** to launch the development tools, select the **Network** tab and click the **Play** button to start recording.



Starting network recording

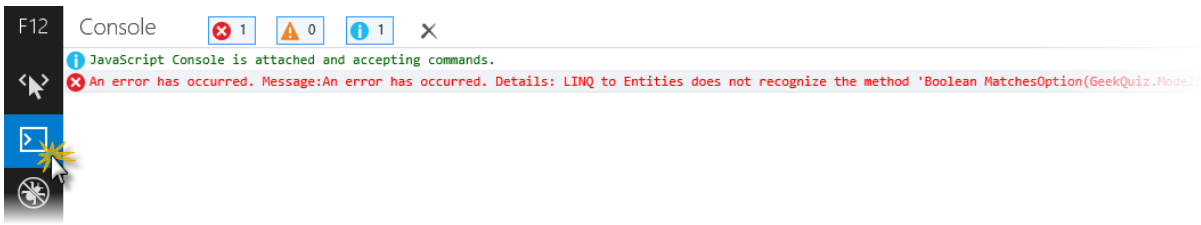
5. Select any option of the quiz. You will see that nothing happens.

6. In the **F12** window, the entry corresponding to the POST HTTP request shows an HTTP **500** result.



HTTP 500 error

7. Select the **Console** tab. An error is logged with the details of the cause.



Logged error

8. Locate the details part of the error. Clearly, this error is caused by the code refactoring you committed in the previous steps.

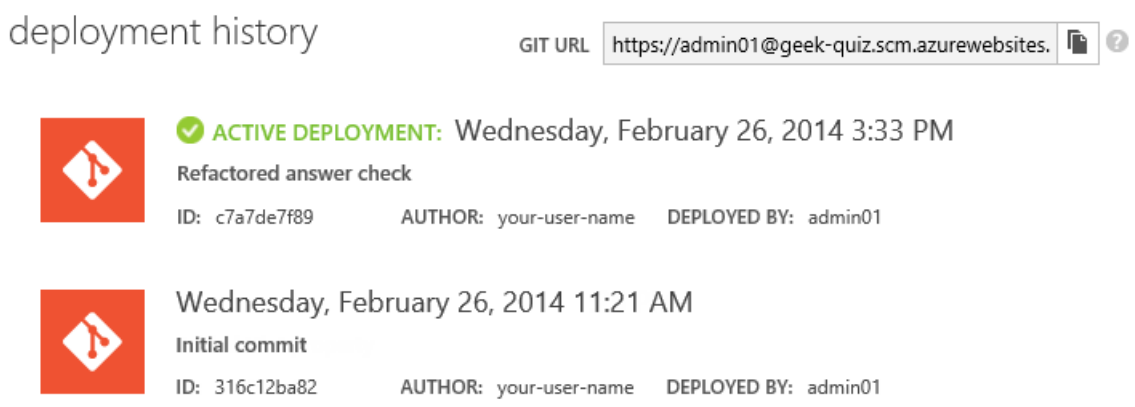
```
Details: LINQ to Entities does not recognize the method 'Boolean MatchesOption ...
```

9. Do not close the browser.

10. In a new browser instance, navigate to the Azure management portal and sign in using the Microsoft account associated with your subscription.

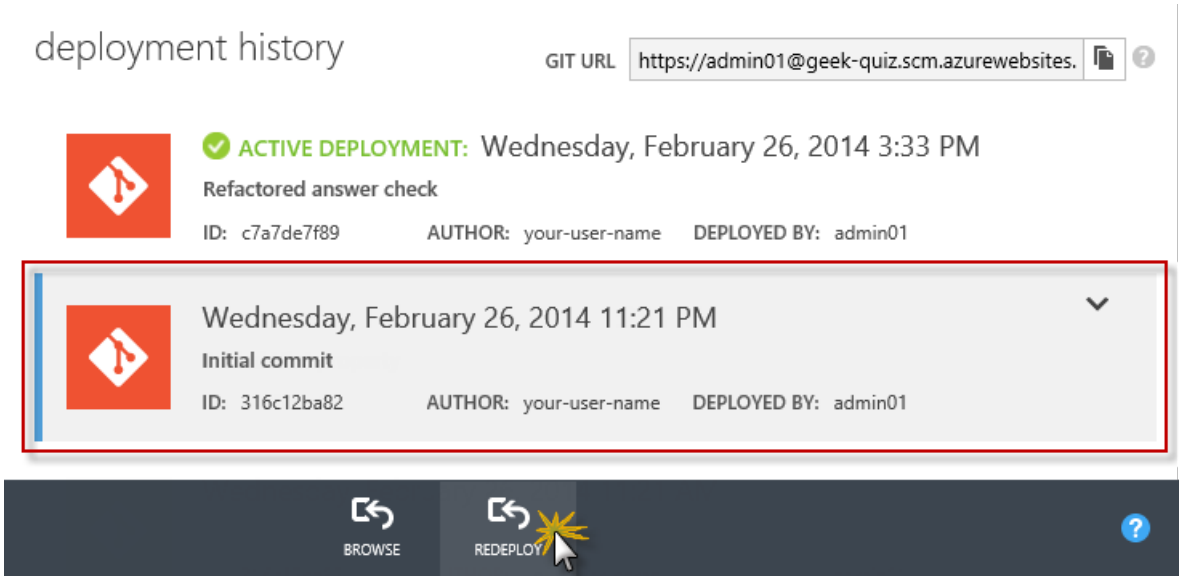
11. Select **Websites** and click the web app you created in Exercise 2.

12. Navigate to the **Deployments** page. Notice that all the commits performed are listed in the deployment history.



List of existing deployments

13. Select the previous commit and click **Redeploy** on the command bar.



Redeploying the previous commit

14. When prompted to confirm, click **Yes**.

15. When the deployment completes, switch back to the browser instance with your web app and press **CTRL + F5**.

16. Click any of the options. The flip animation will now take place and the result (*correct/incorrect*) will be displayed.

17. (Optional) Switch to the **Git Bash** console and execute the following commands to revert to the previous commit.

> **NOTE**
>
> These commands create a new commit that undoes all changes in the Git repository that were made in the bad commit. Azure will then redeploy the application using the new commit.

```
git revert HEAD --no-edit
git push azure master
```

Exercise 4: Scaling Using Azure Storage

Blobs are the simplest way to store large amounts of unstructured text or binary data such as video, audio and images. Moving the static content of your application to Storage, helps to scale your application by serving images or documents directly to the browser.

In this exercise, you will move the static content of your application to a Blob container. Then you will configure your application to add an **ASP.NET URL rewrite rule** in the **Web.config** to redirect your content to the Blob container.

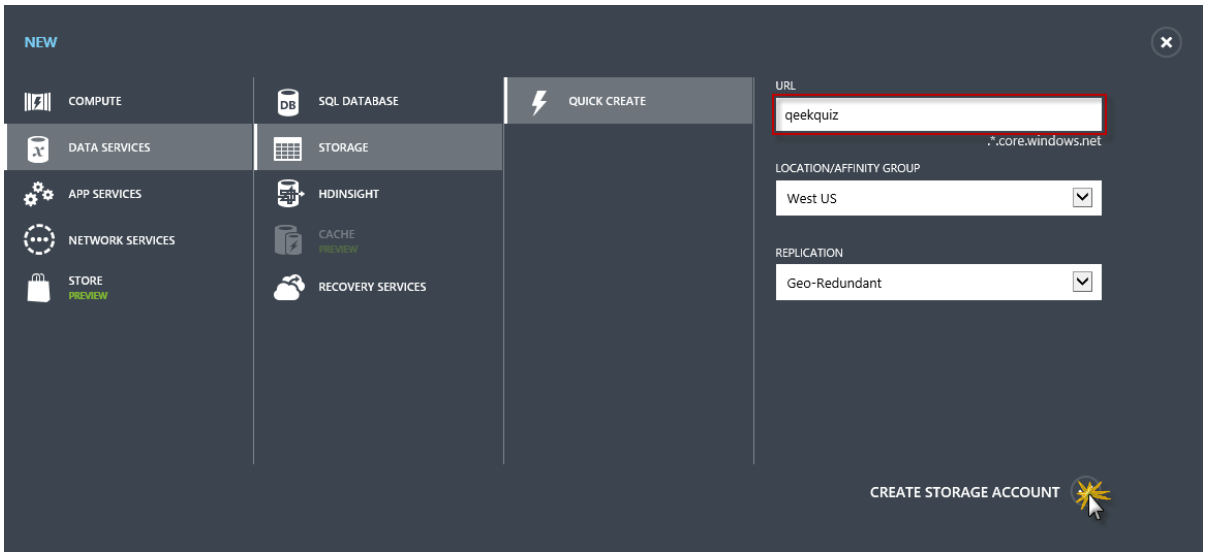
Task 1 – Creating an Azure Storage Account

In this task you will learn how to create a new storage account using the management portal.

1. Navigate to the Azure management portal and sign in using the Microsoft account associated with your subscription.

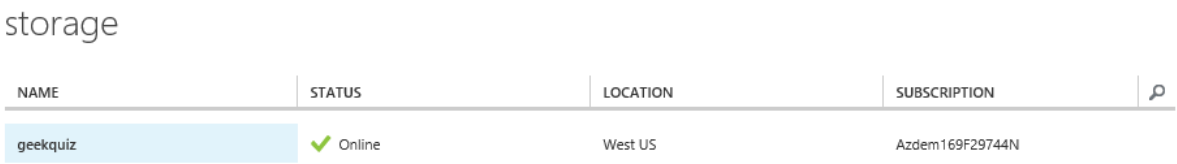
2. Select **New | Data Services | Storage | Quick Create** to start creating a new storage account. Enter a

unique name for the account and select a **Region** from the list. Click **Create Storage Account** to continue.



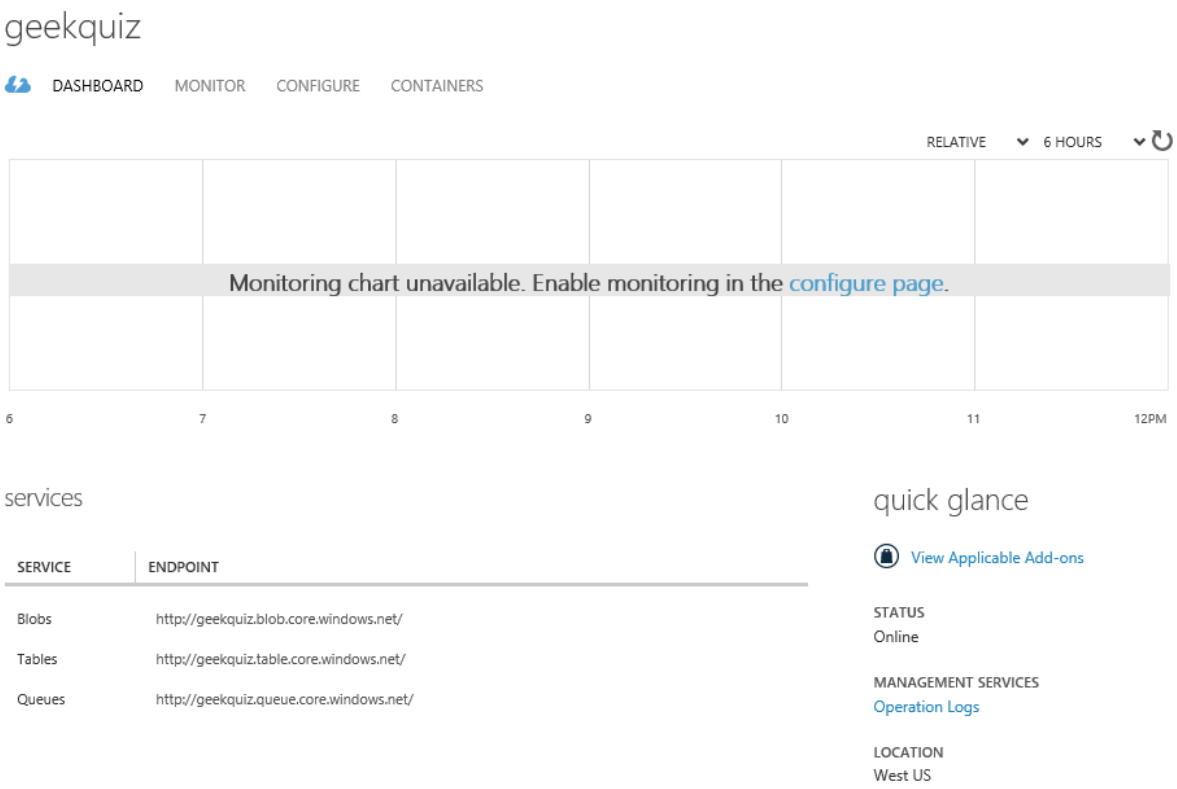
Creating a new storage account

3. In the **Storage** section, wait until the status of the new storage account changes to *Online* in order to continue with the following step.



Storage Account created

4. Click on the storage account name and then click the **Dashboard** link at the top of the page. The **Dashboard** page provides you with information about the status of the account and the service endpoints that can be used within your applications.

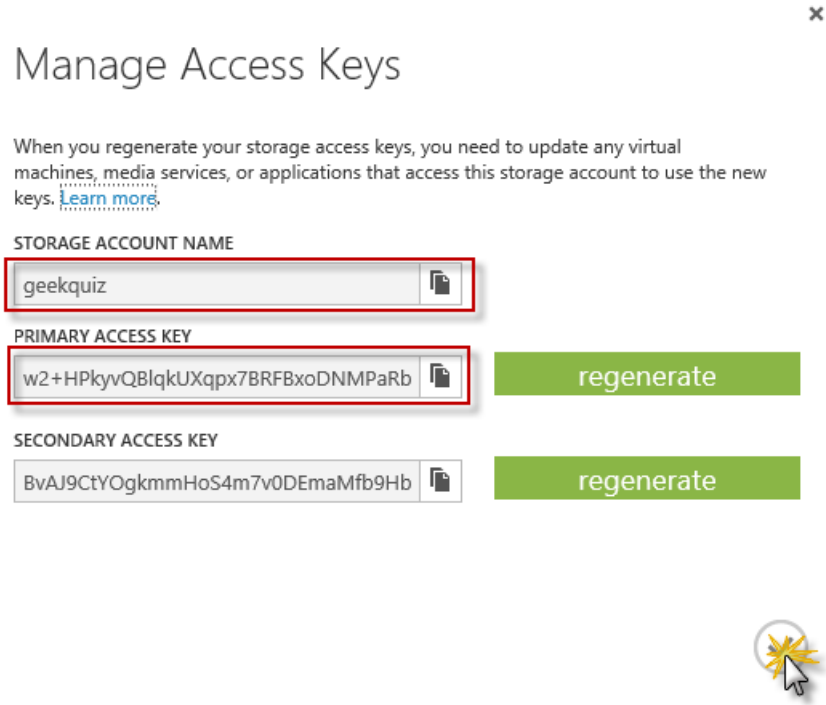


Displaying the Storage Account Dashboard

5. Click the **Manage Access Keys** button in the navigation bar.

Manage Access Keys button

6. In the **Manage Access Keys** dialog box, copy the **Storage Account Name** and **Primary Access Key** as you will need them in the following exercise. Then, close the dialog box.



Manage Access Key dialog box

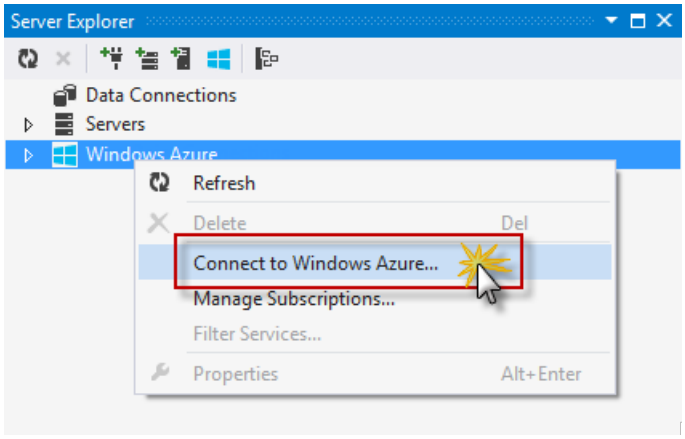
Task 2 – Uploading an Asset to Azure Blob Storage

In this task, you will use the Server Explorer window from Visual Studio to connect to your storage account. You will then create a blob container and upload a file with the Geek Quiz logo to the container.

1. Switch to the Visual Studio instance with the **GeekQuiz** solution from the previous exercise.

2. From the menu bar, select **View** and then click **Server Explorer**.

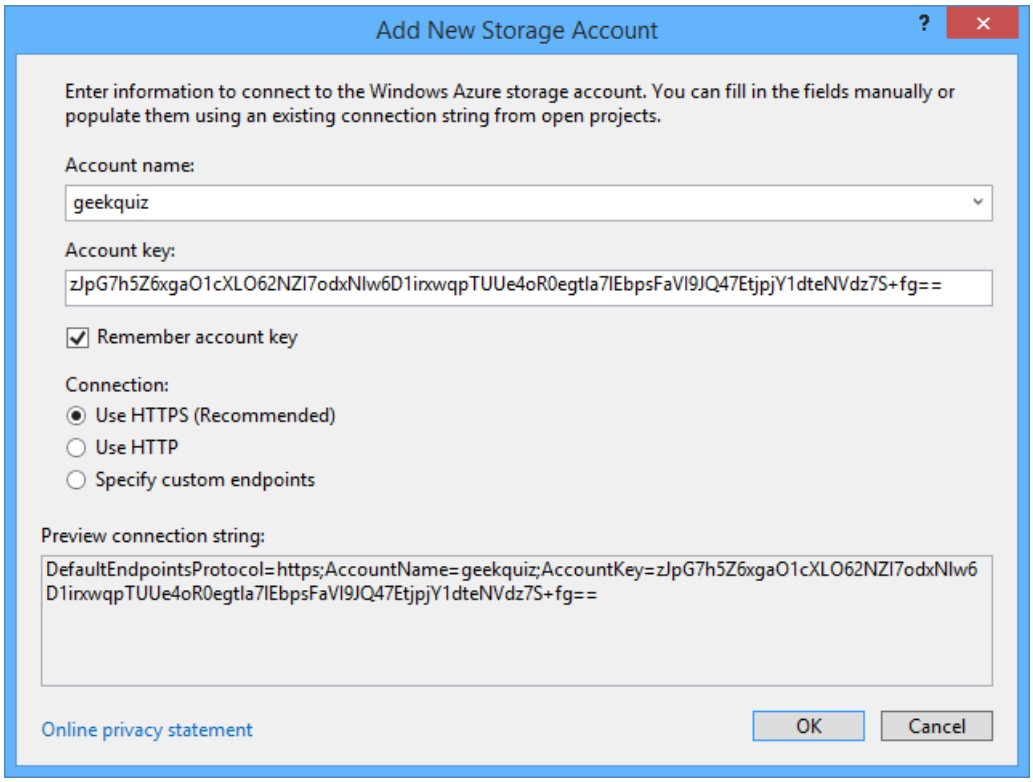
3. In **Server Explorer**, right-click the **Azure** node and select **Connect to Azure...**. Sign in using the Microsoft account associated with your subscription.



Connect to Azure

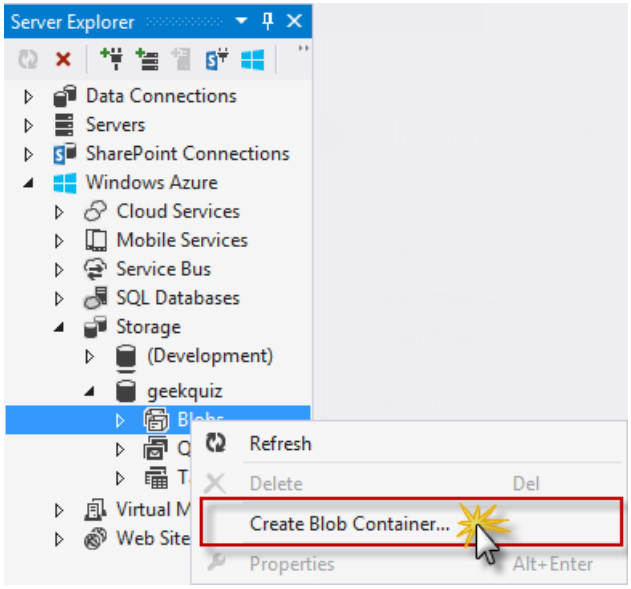
4. Expand the **Azure** node, right-click **Storage** and select **Attach External Storage...**.

5. In the **Add New Storage Account** dialog box, enter the **Account name** and **Account key** you obtained in the previous task and click **OK**.



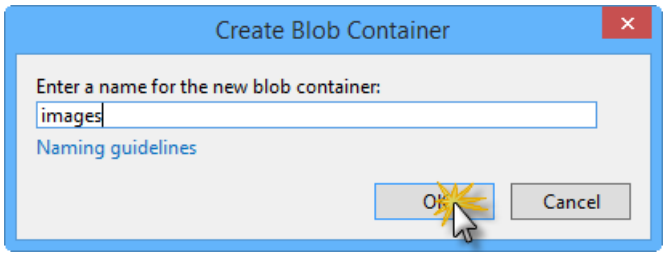
Add New Storage Account dialog box

6. Your storage account should appear under the **Storage** node. Expand your storage account, right-click **Blobs** and select **Create Blob Container...**.



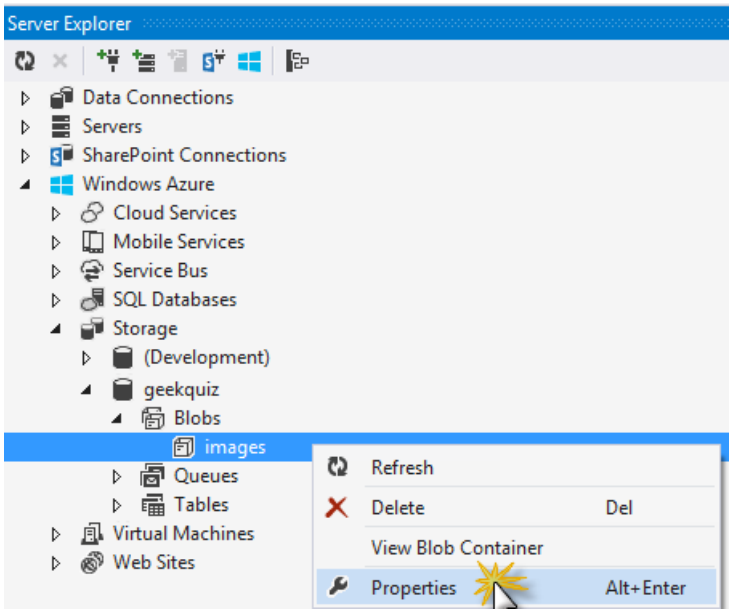
Create Blob Container

7. In the **Create Blob Container** dialog box, enter a name for the blob container and click **OK**.



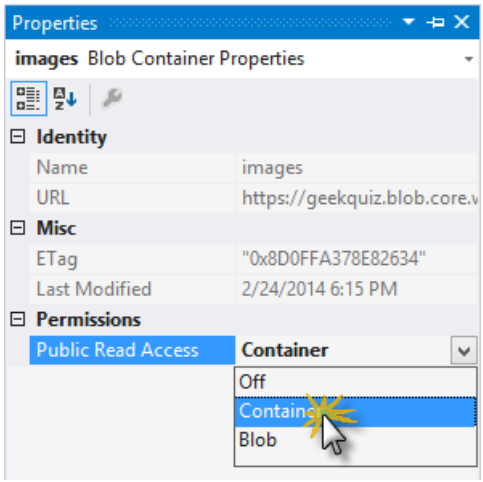
Create Blob Container dialog box

8. The new blob container should be added to the **Blobs** node. Change the access permissions in the container to make the container public. To do this, right-click the **images** container and select **Properties**.



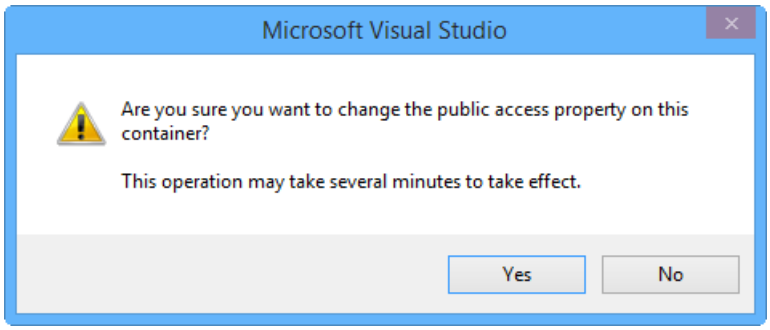
Images container properties

9. In the **Properties** window, set the **Public Read Access** to **Container**.



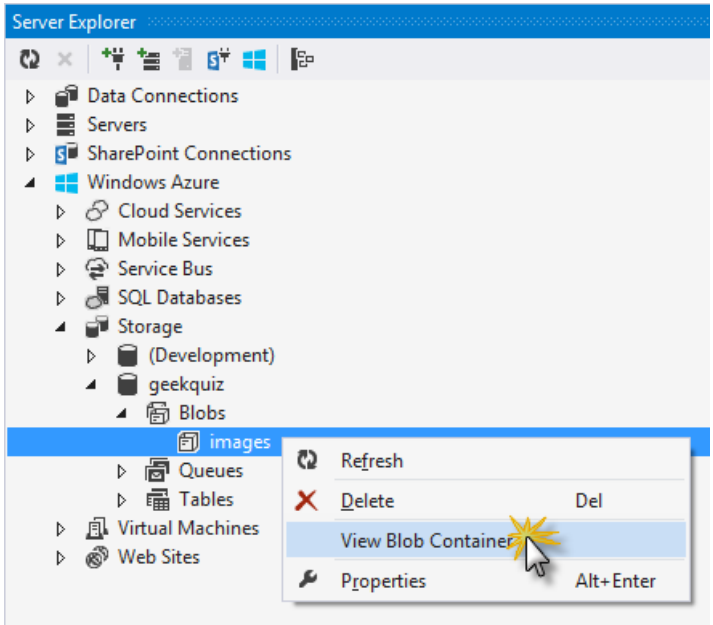
Changing public read access property

10. When prompted if you are sure you want to change the public access property, click **Yes**.



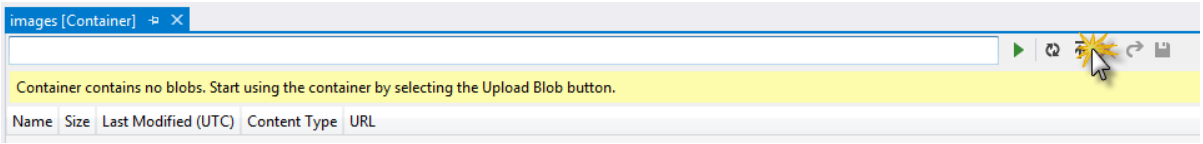
Microsoft Visual Studio warning

11. In **Server Explorer**, right-click in the **images** blob container and select **View Blob Container**.



View Blob Container

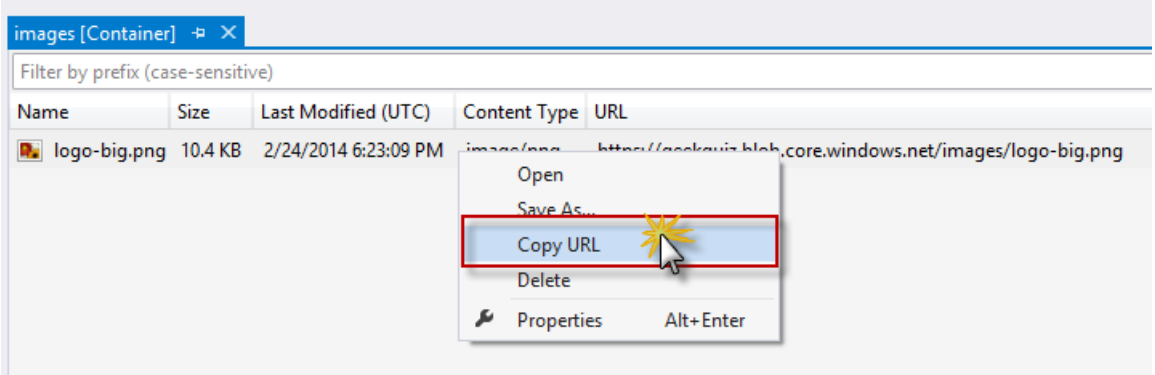
12. The images container should open in a new window and a legend with no entries should be shown. Click the **upload** icon to upload a file to the blob container.



Images container with no entries

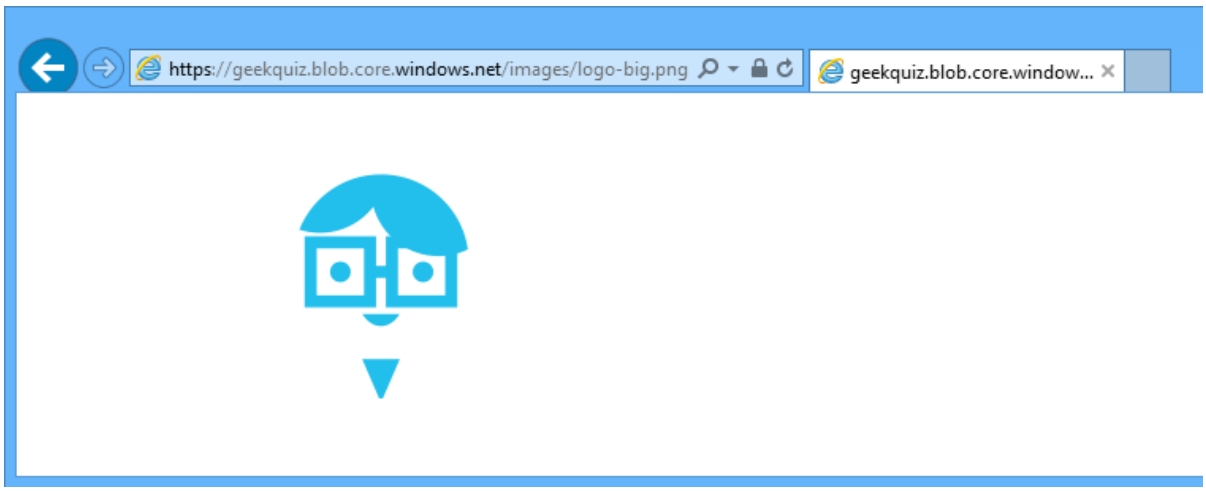
13. In the **Upload Blob** dialog box, navigate to the **Assets** folder of the lab. Select the **logo-big.png** file and click **Open**.

14. Wait until the file is uploaded. When the upload completes, the file should be listed in the images container. Right-click the file entry and select **Copy URL**.



Copy blob URL

15. Open Internet Explorer and paste the URL. The following image should be shown in the browser.



logo-big.png image from Azure Blob Storage

Task 3 – Updating the Solution to Consume Static Content from Azure Blob Storage

In this task, you will configure the **GeekQuiz** solution to consume the image uploaded to Azure Blob Storage (instead of the image located in the web app) by adding an ASP.NET URL rewrite rule in the **web.config** file.

1. In Visual Studio, open the **Web.config** file inside the **GeekQuiz** project and locate the **<system.webServer>** element.

2. Add the following code to add an URL rewrite rule, updating the placeholder with your storage account name.

   (Code Snippet - *WebSitesInProduction - Ex4 - UrlRewriteRule*)

```
<system.webServer>
    <rewrite>
        <rules>
            <rule name="redirect-images" stopProcessing="true">
                <match url="img/(.*)"/>
                <action type="Redirect" url="http://[YOUR-STORAGE-
ACCOUNT].blob.core.windows.net/images/{R:1}"></action>
            </rule>
        </rules>
    </rewrite>
```

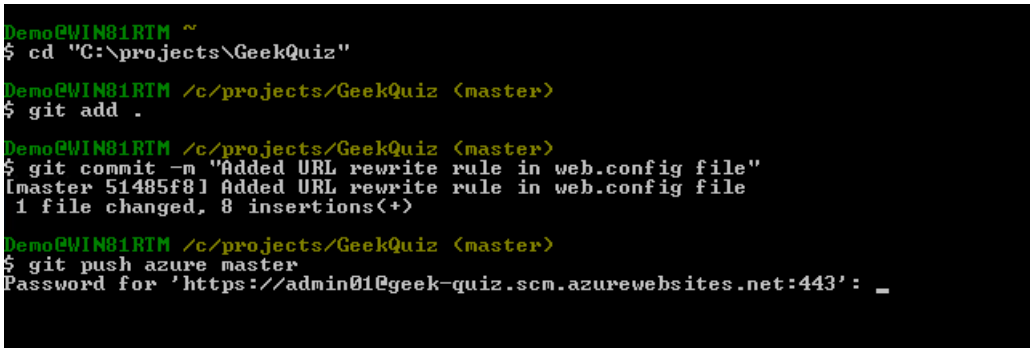
> **NOTE**
>
> URL rewriting is the process of intercepting an incoming Web request and redirecting the request to a different resource. The URL rewriting rules tells the rewriting engine when a request needs to be redirected, and where should they be redirected. A rewriting rule is composed of two strings: the pattern to look for in the requested URL (usually, using regular expressions), and the string to replace the pattern with, if found. For more information, see URL Rewriting in ASP.NET.

3. Press **CTRL + S** to save the changes.

4. Open a new **Git Bash** console to deploy the updated application to Azure App Service.

5. Execute the following commands to push the changes to Azure. Update the *[YOUR-APPLICATION-PATH]* placeholder with the path to the **GeekQuiz** solution. You will be prompted for your deployment password.

```
cd "[YOUR-APPLICATION-PATH]"
git add .
git commit -m "Added URL rewrite rule in web.config file"
git push azure master
```

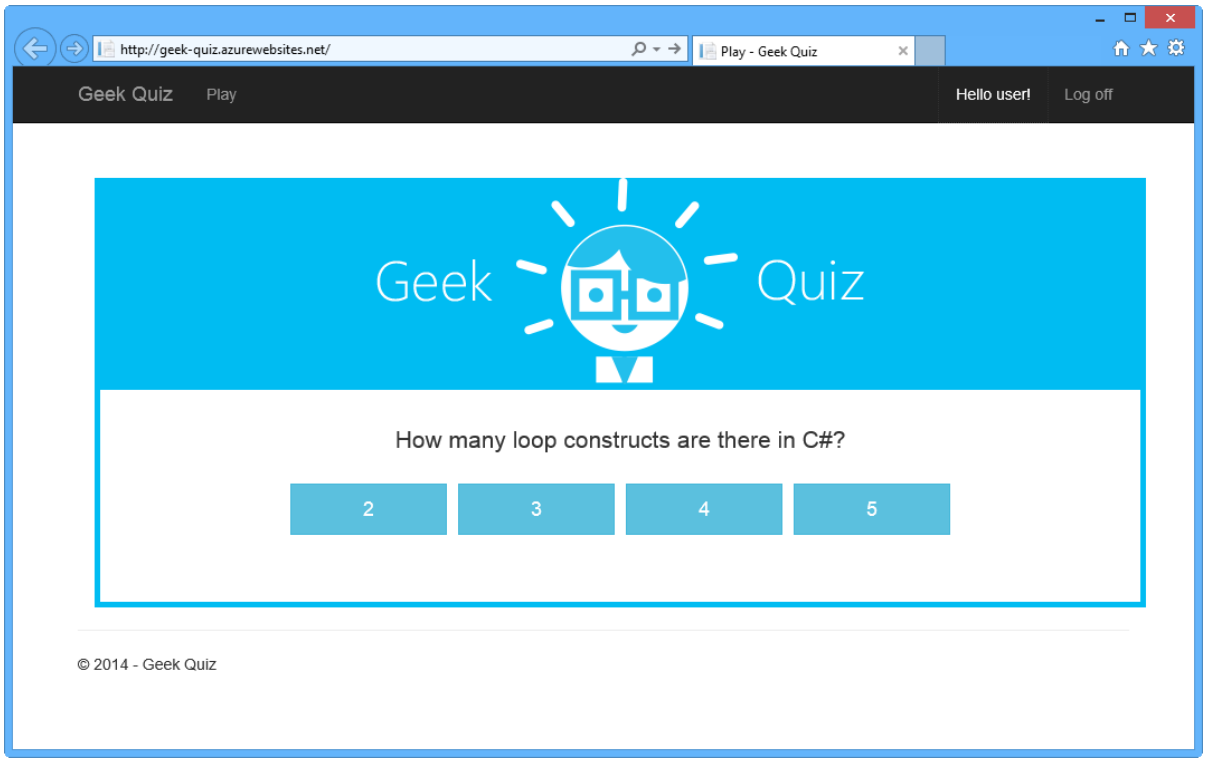


Deploying update to Azure

Task 4 – Verification

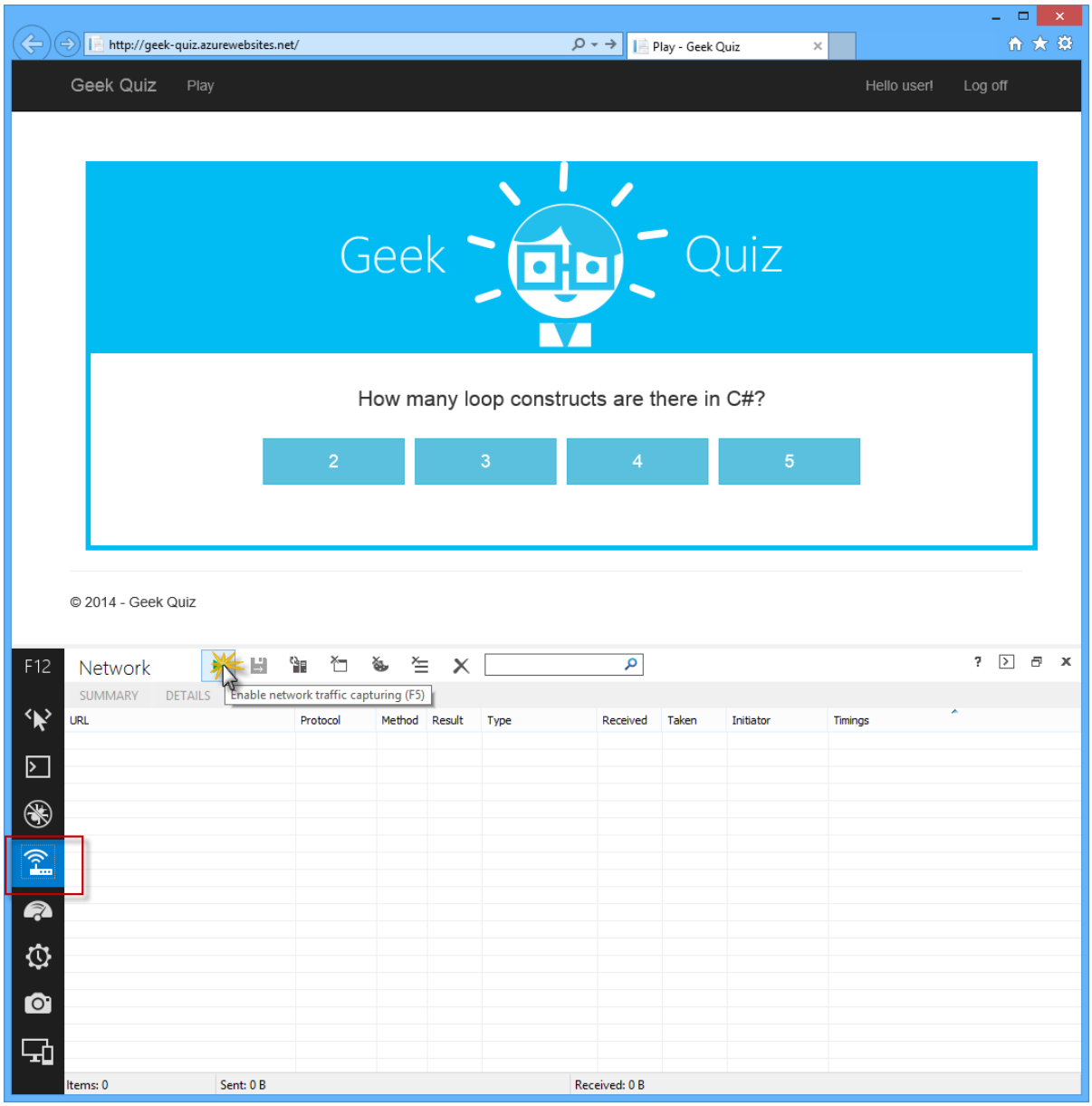
In this task you will use **Internet Explorer** to browse the **Geek Quiz** application and check that the URL rewrite rule for images works and you are redirected to the image hosted on **Azure Blob Storage**.

1. Open Internet Explorer and navigate to your web app (e.g. `http://<your-web-site>.azurewebsites.net`). Log in using the previously created credentials.



Showing the Geek Quiz web app with the image

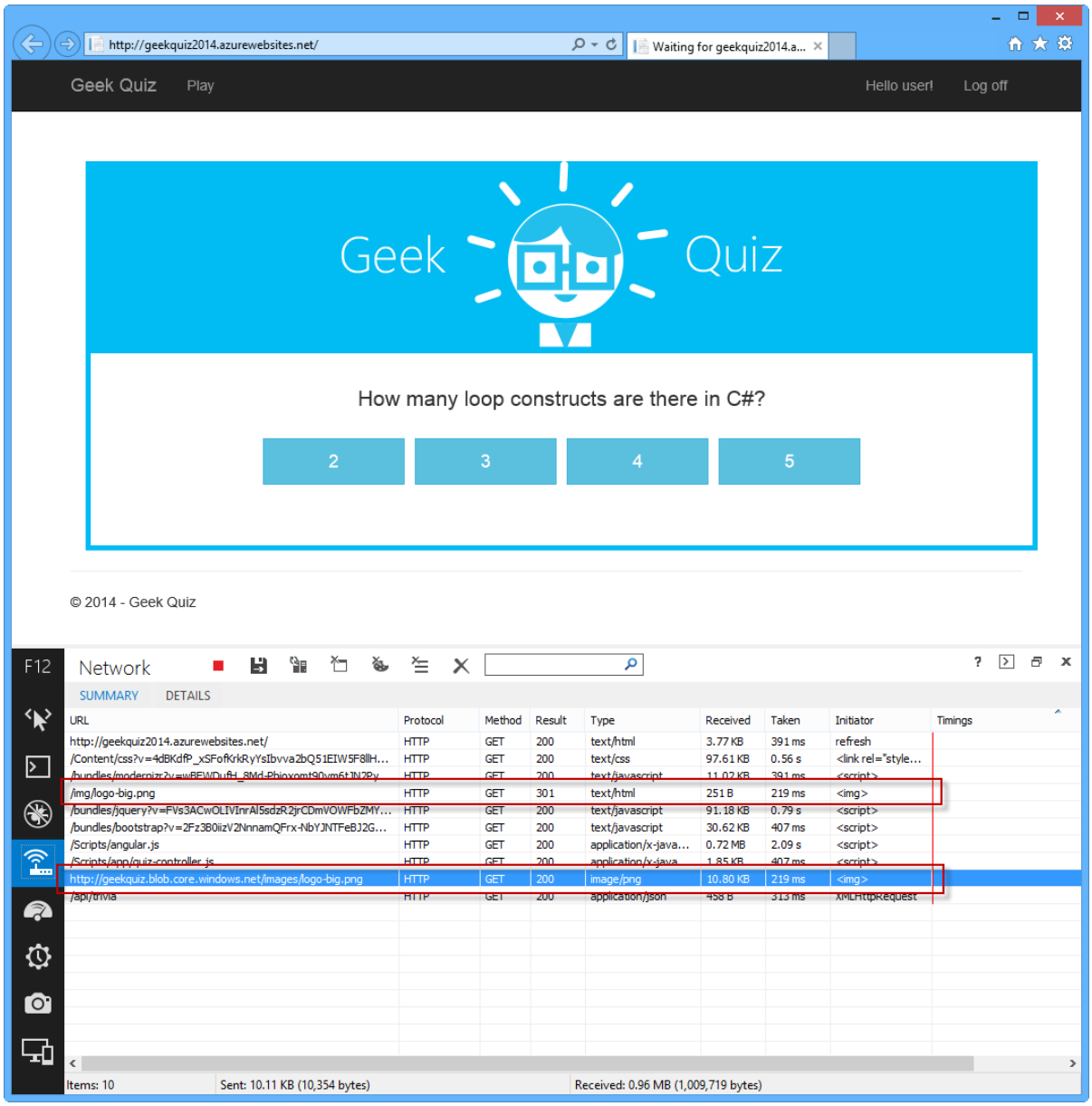
2. Press **F12** to launch the development tools, select the **Network** tab and start recording.



Starting network recording

3. Press **CTRL + F5** to refresh the web page.

4. Once the page has finished loading, you should see an HTTP request for the **/img/logo-big.png** URL with an HTTP **301** result (redirect) and another request for `http://[YOUR-STORAGE-ACCOUNT].blob.core.windows.net/images/logo-big.png` URL with a HTTP **200** result.



Verifying the URL redirect

Exercise 5: Using Autoscale for Web Apps

> **NOTE**
>
> This exercise is optional, since it requires support for Web Load & Performance Testing which is only available for **Visual Studio 2013 Ultimate Edition**. For more information on specific Visual Studio 2013 features, compare versions here.

Azure App Service Web Apps provides the Autoscale feature for web apps running in **Standard Mode**. Autoscale lets Azure automatically scale the instance count of your web app depending on the load. When Autoscale is enabled, Azure checks the CPU of your web app once every five minutes and adds instances as needed at that point in time. If the CPU usage is low, Azure will remove instances once every two hours to ensure that the performance of your web app is not degraded.

In this exercise you will go through the steps required to configure the **Autoscale** feature for the **Geek Quiz** web app. You will verify this feature by running a Visual Studio load test to generate enough CPU load on the application to trigger an instance upgrade.

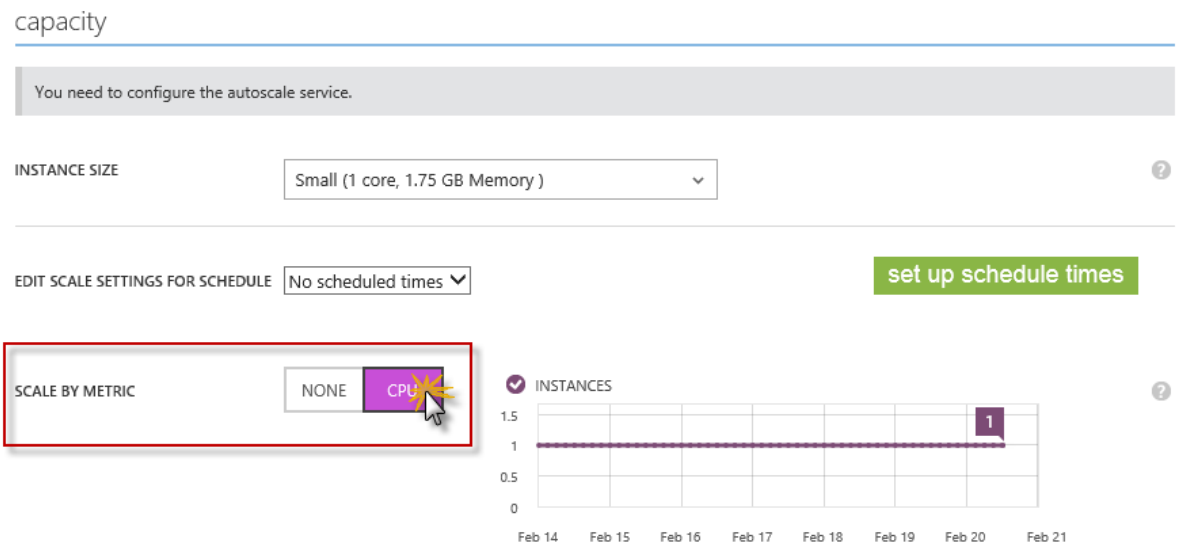
Task 1 – Configuring Autoscale Based on the CPU Metric

In this task you will use the Azure management portal to enable the Autoscale feature for the web app you created in Exercise 2.

1. In the Azure management portal, select **Websites** and click the web app you created in Exercise 2.

2. Navigate to the **Scale** page. Under the **capacity** section, select **CPU** for the **Scale by Metric** configuration.

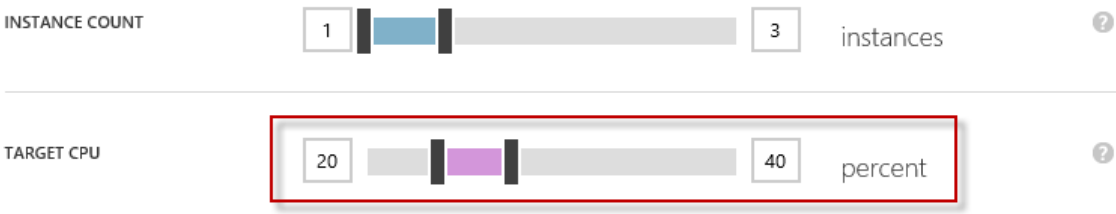
> **NOTE**
>
> When scaling by CPU, Azure dynamically adjusts the number of instances that the app uses if the CPU usage changes.



Selecting to scale by CPU

3. Change the **Target CPU** configuration to **20-40** percent.

> **NOTE**
>
> This range represents the average CPU usage for your web app. Azure will add or remove instances to keep your web app in this range. The minimum and maximum number of instances used for scaling is specified in the **Instance Count** configuration. Azure will never go above or beyond that limit.
>
> The default **Target CPU** values are modified just for the purposes of this lab. By configuring the CPU range with small values, you are increasing the chances to trigger Autoscale when a moderate load is placed on the application.



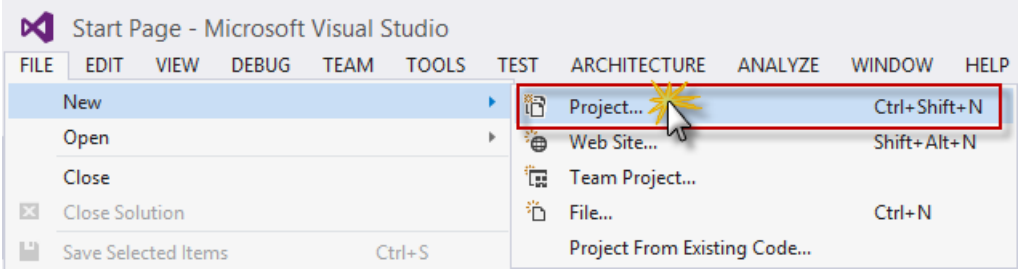
Changing the Target CPU to be between 20 and 40 percent

4. Click **Save** in the command bar to save the changes.

Task 2 – Load Testing with Visual Studio

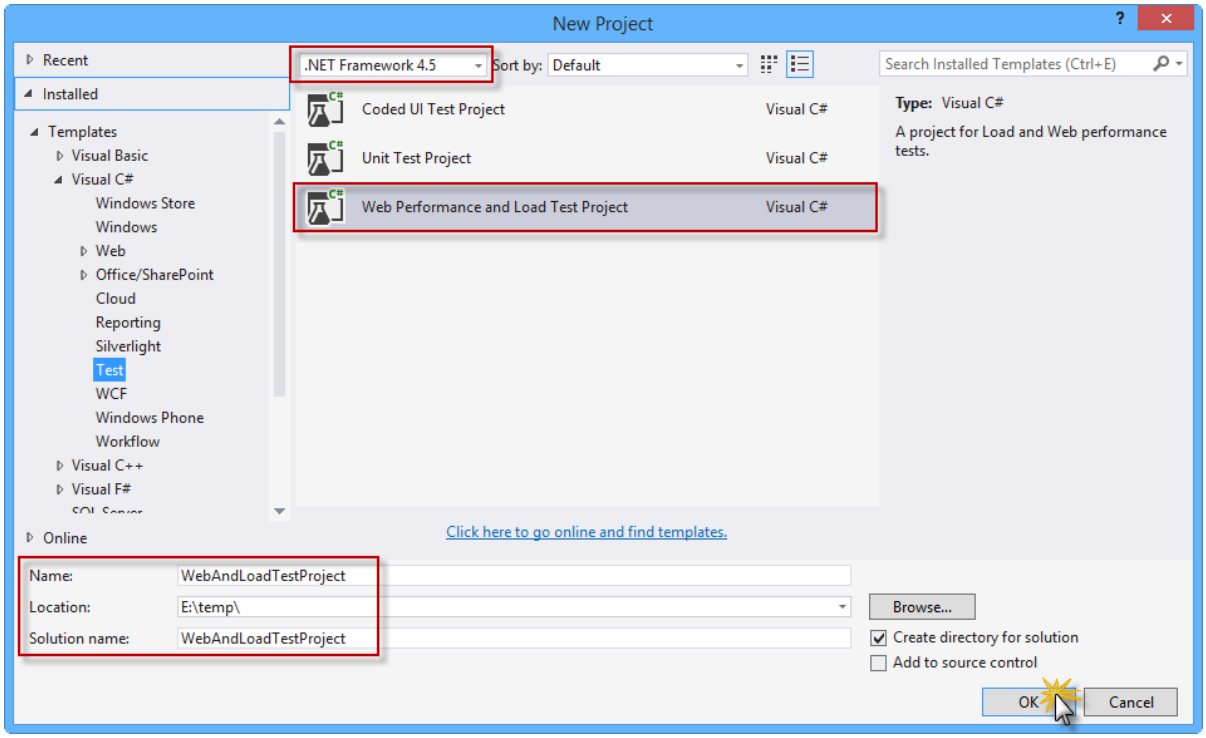
Now that **Autoscale** has been configured, you will create a **Web Performance and Load Test Project** in Visual Studio to generate some CPU load on your web app.

1. Open **Visual Studio Ultimate 2013** and select **File | New | Project...** to start a new solution.



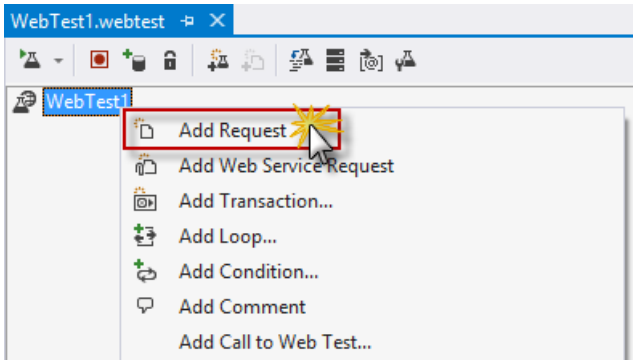
Creating a new project

2. In the **New Project** dialog box, select **Web Performance and Load Test Project** under the **Visual C# | Test** tab. Make sure **.NET Framework 4.5** is selected, name the project *WebAndLoadTestProject*, choose a **Location** and click **OK**.



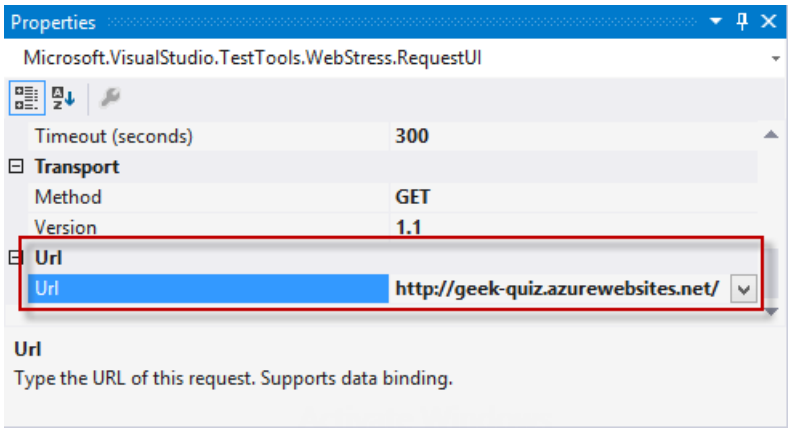
Creating a new Web and Load Test project

3. In the **WebTest1.webtest** Right-click the **WebTest1** node and click **Add Request**.



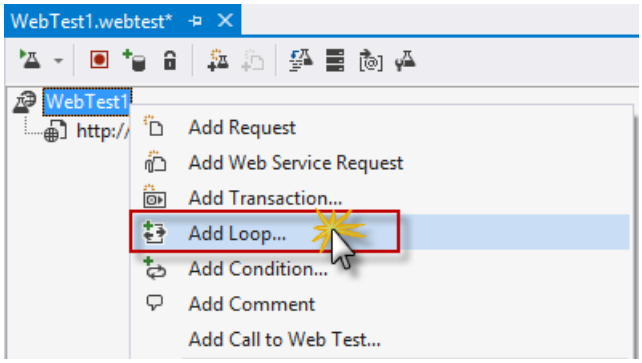
Adding a request to WebTest1

4. In the **Properties** window of the new request node, update the **Url** property to point to the URL of your web app (e.g. *http://geek-quiz.azurewebsites.net/*).



Changing the Url property

5. In the **WebTest1.webtest** window, right-click **WebTest1** and click **Add Loop...**.



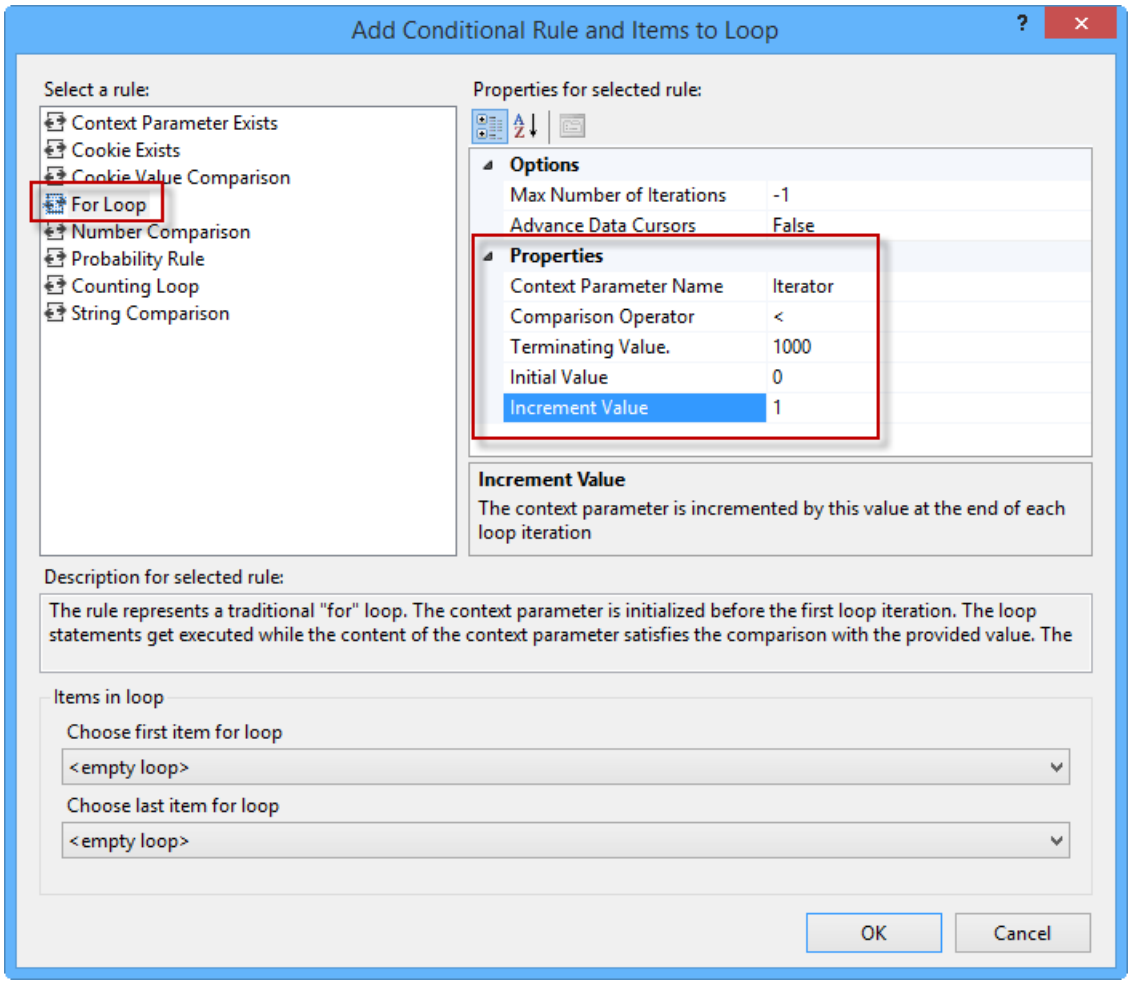
Adding a loop to WebTest1

6. In the **Add Conditional Rule and Items to Loop** dialog box, select the **For Loop** rule and modify the following properties.

   a. **Terminating value:** 1000

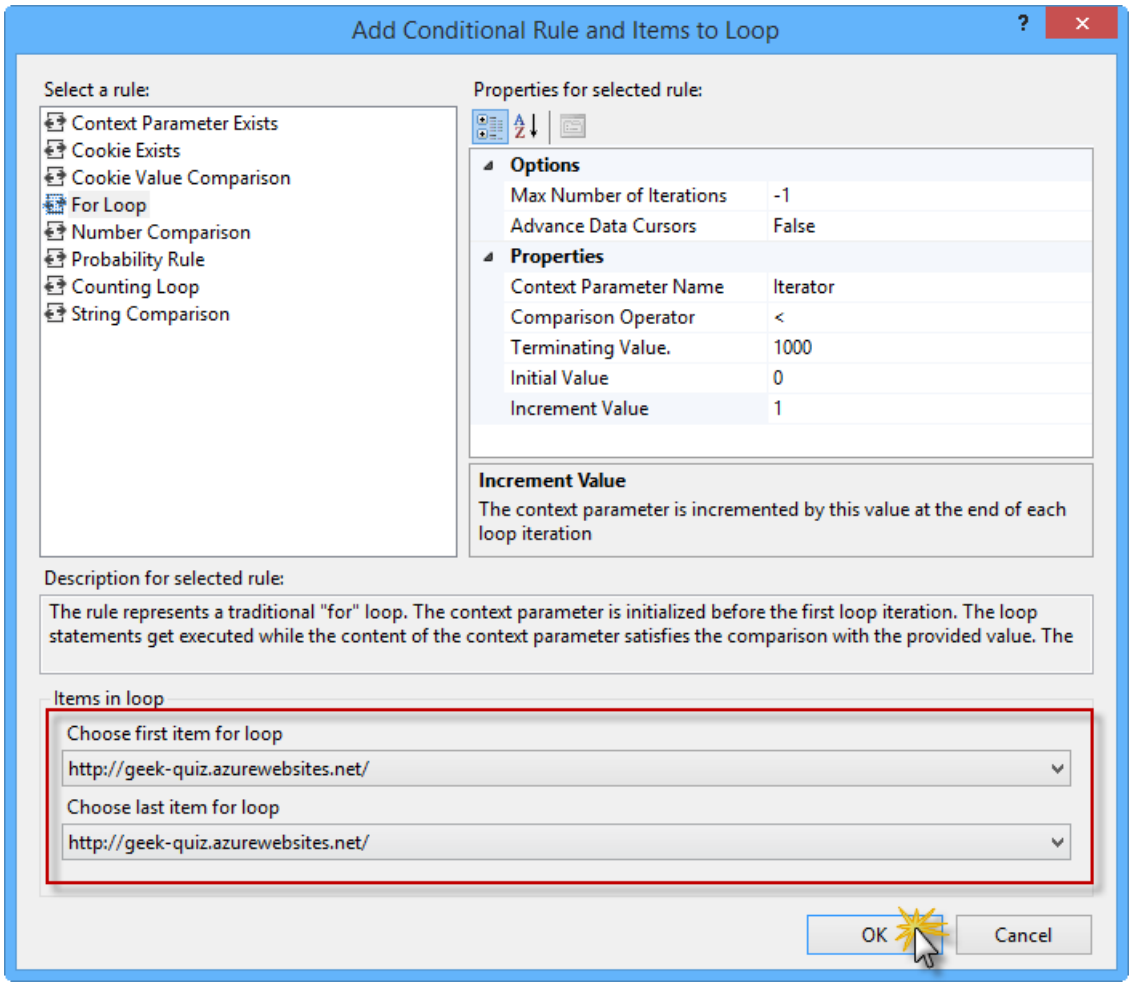
   b. **Context Parameter Name:** Iterator

   c. **Increment Value:** 1



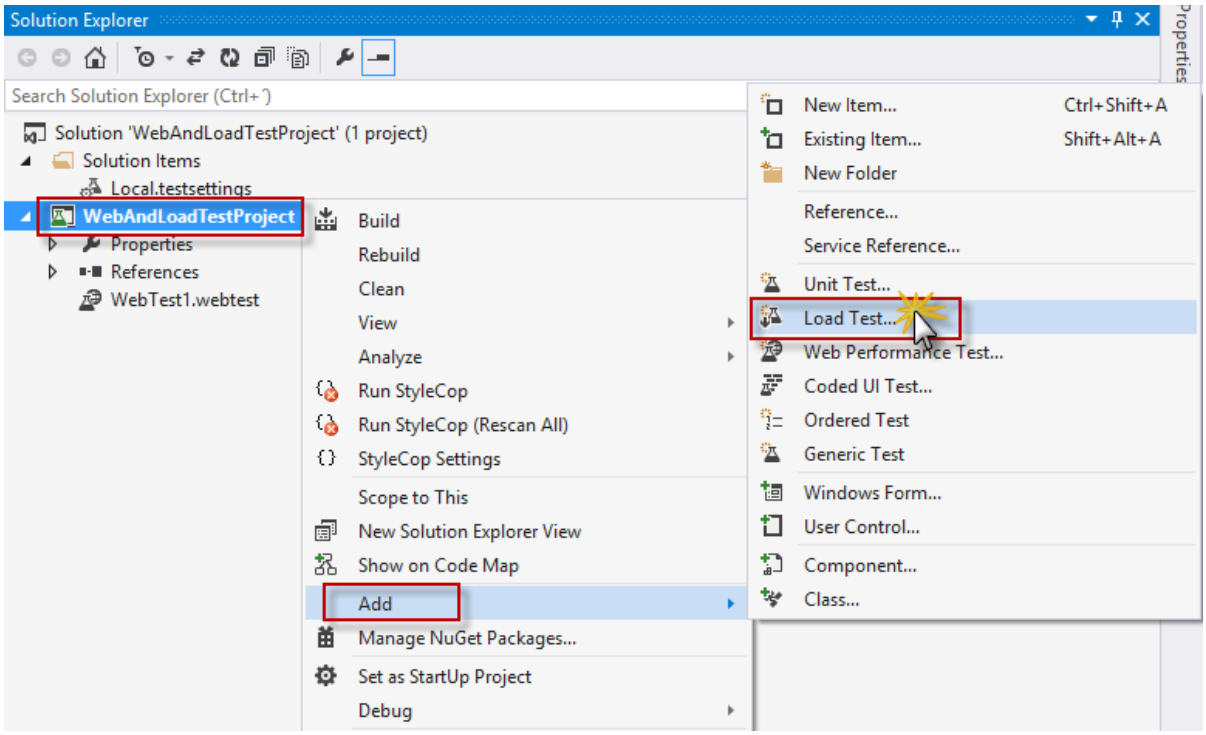
Selecting the For Loop rule and updating the properties

7. Under the **Items in loop** section, select the request you created previously to be the first and last item for the loop. Click **OK** to continue.



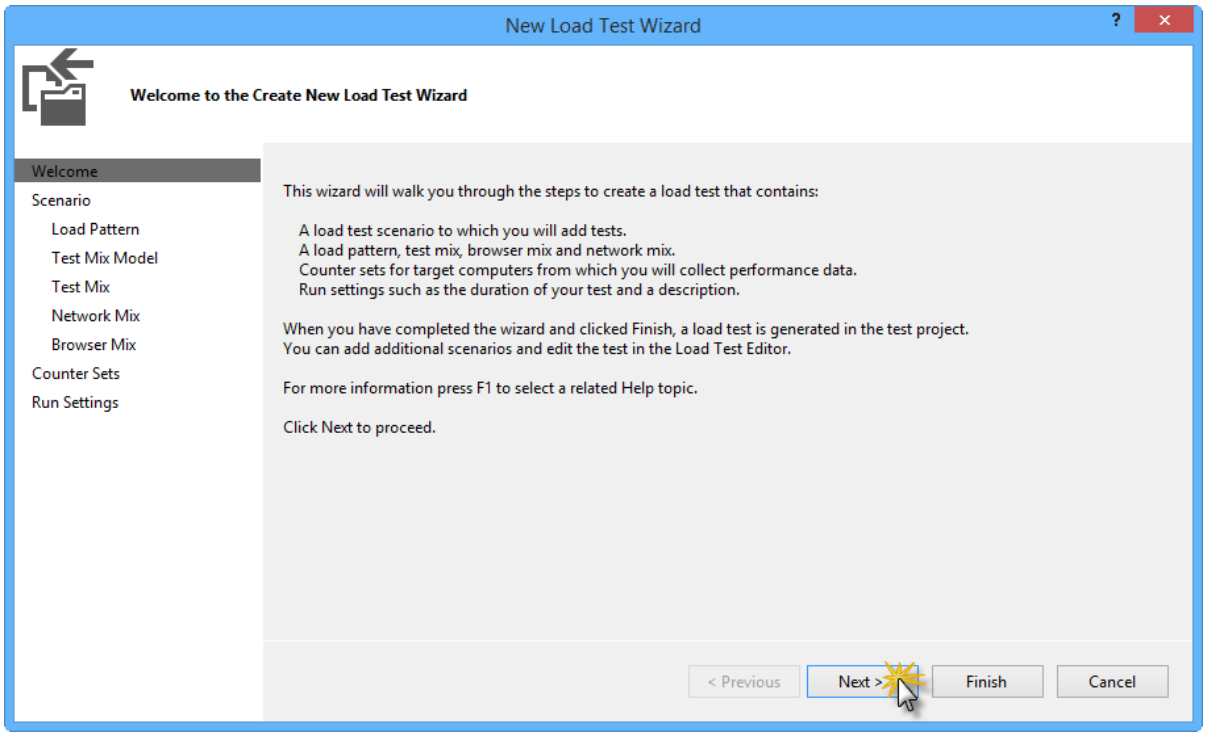
Selecting the first and last items for the loop

8. In **Solution Explorer**, right-click the **WebAndLoadTestProject** project, expand the **Add** menu and select **Load Test...**.



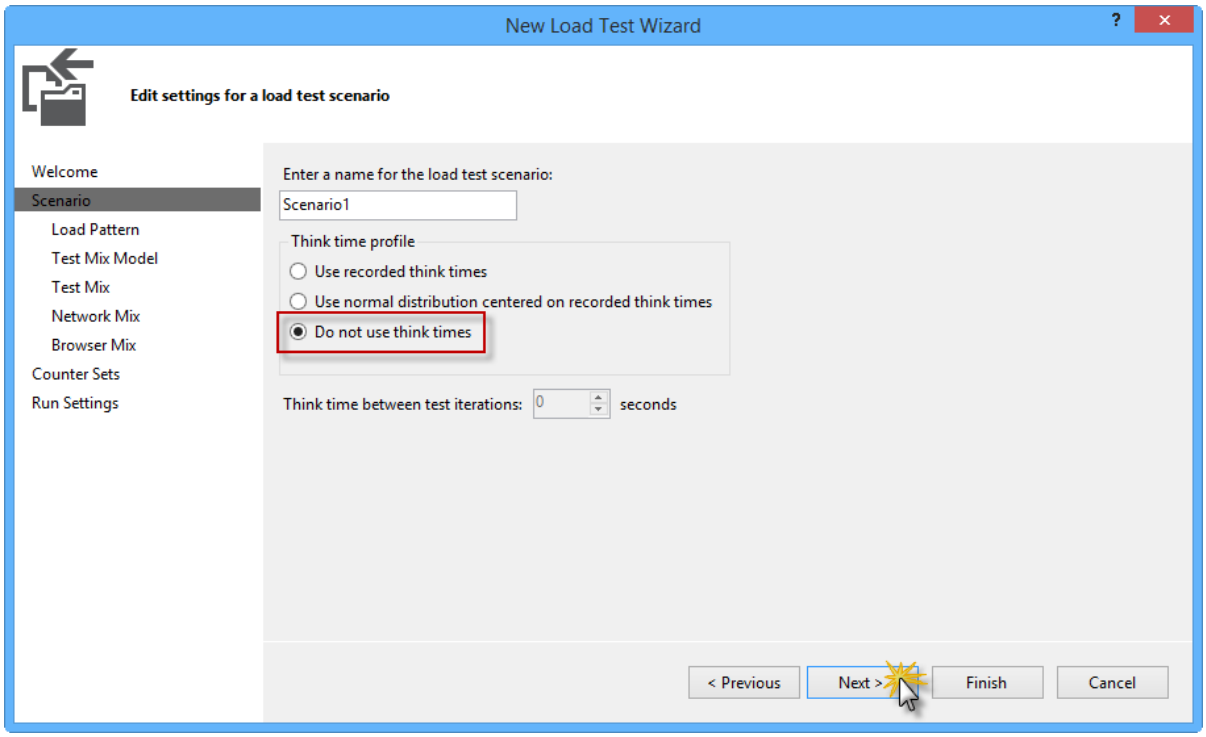
Adding a Load Test to the WebAndLoadTestProject project

9. In the **New Load Test Wizard** dialog box, click **Next**.



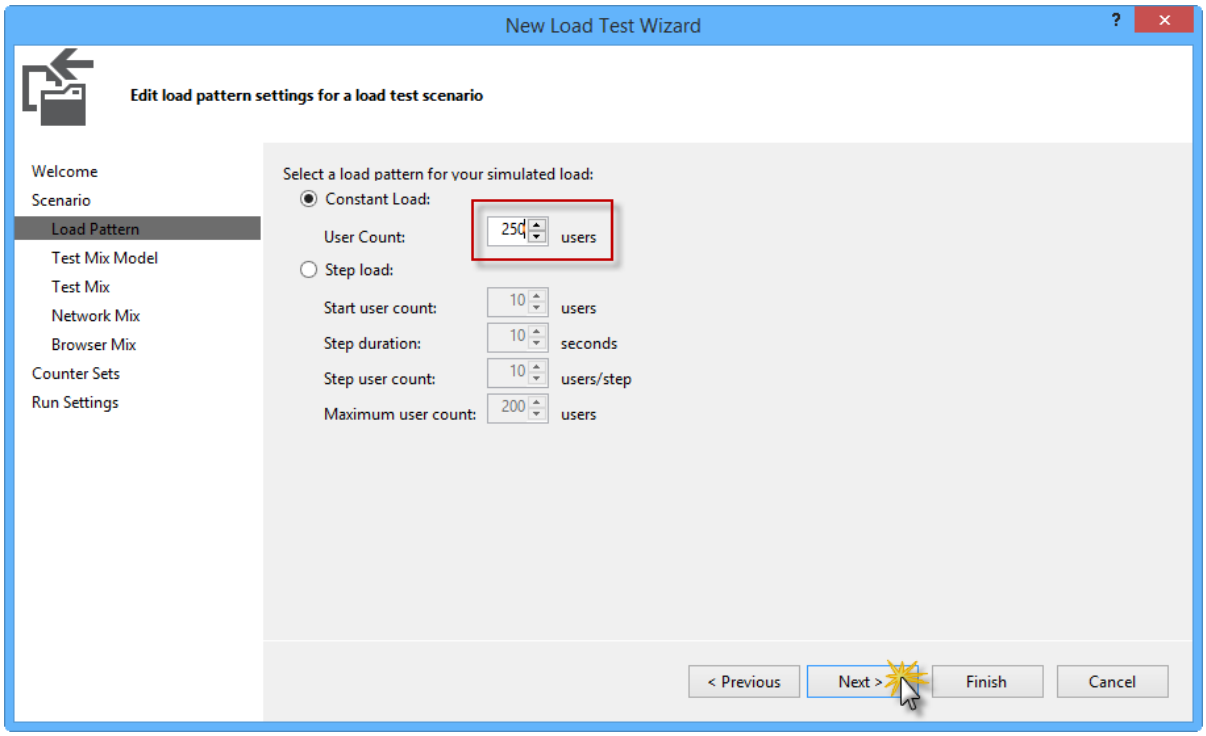
New Load Test Wizard

10. In the **Scenario** page, select **Do not use think times** and click **Next**.



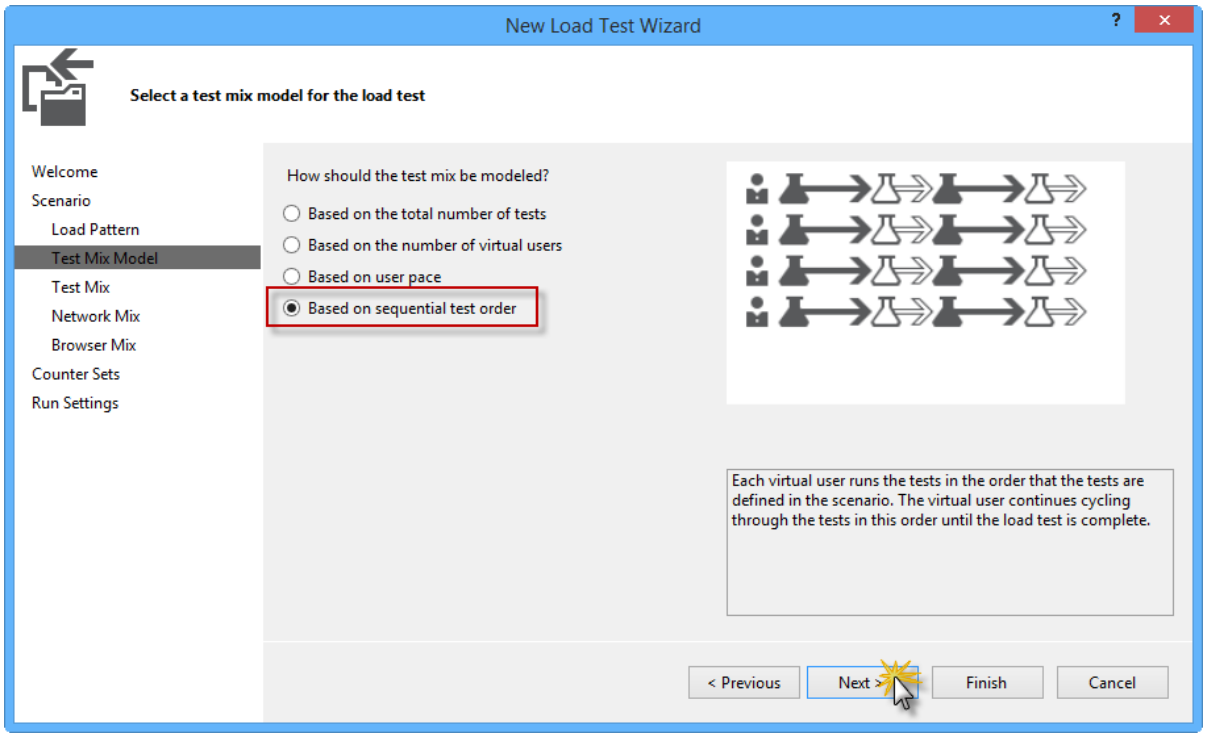
Selecting not to use think times

11. In the **Load Pattern** page, make sure that the **Constant Load** option is selected. Change the **User Count** setting to **250** users and click **Next**.



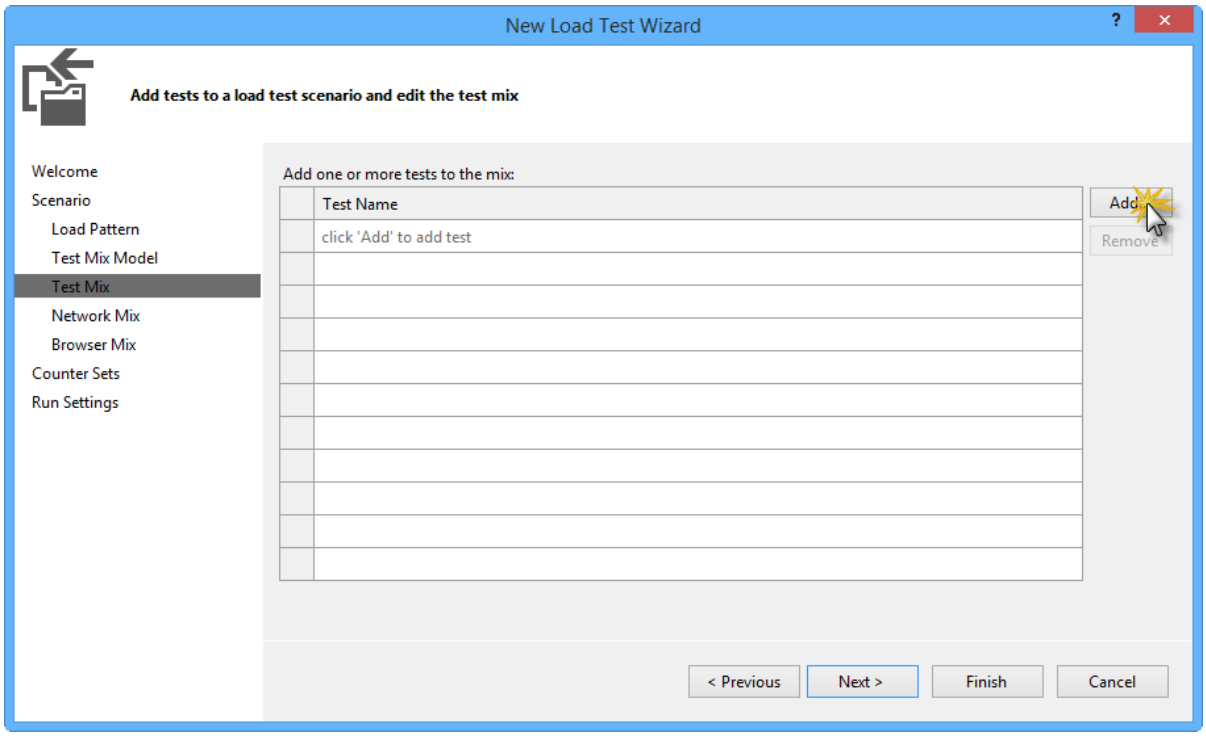
Changing the user count to 250

12. In the **Test Mix Model** page, select **Based on sequential test order** and click **Next**.



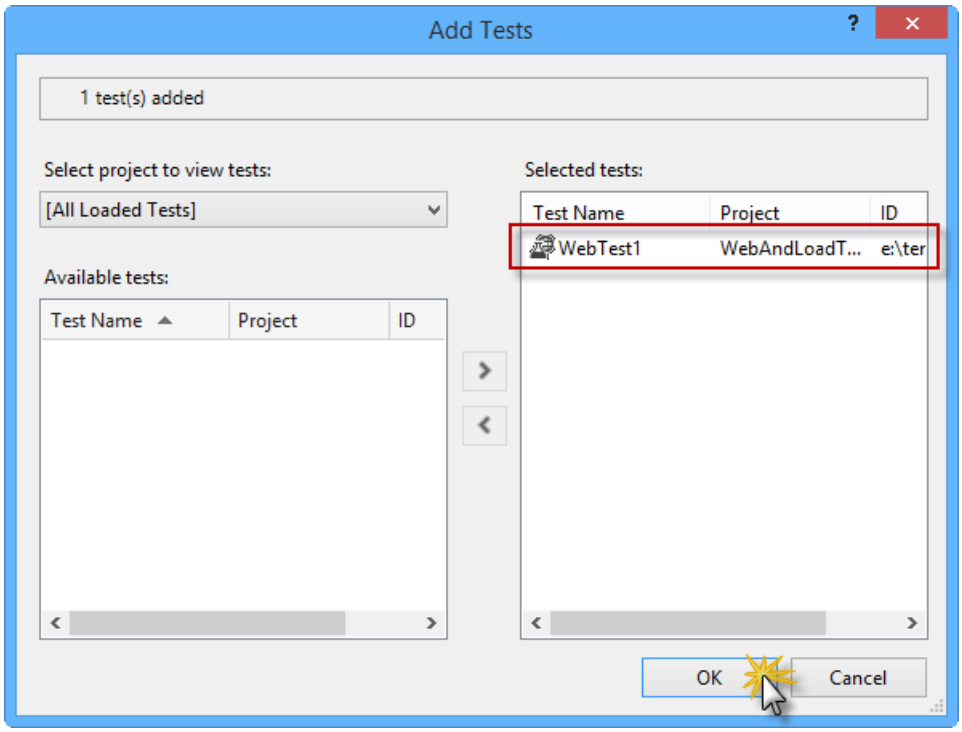
Selecting the test mix model

13. In the **Test Mix Model** page, click **Add...** to add a test to the mix.



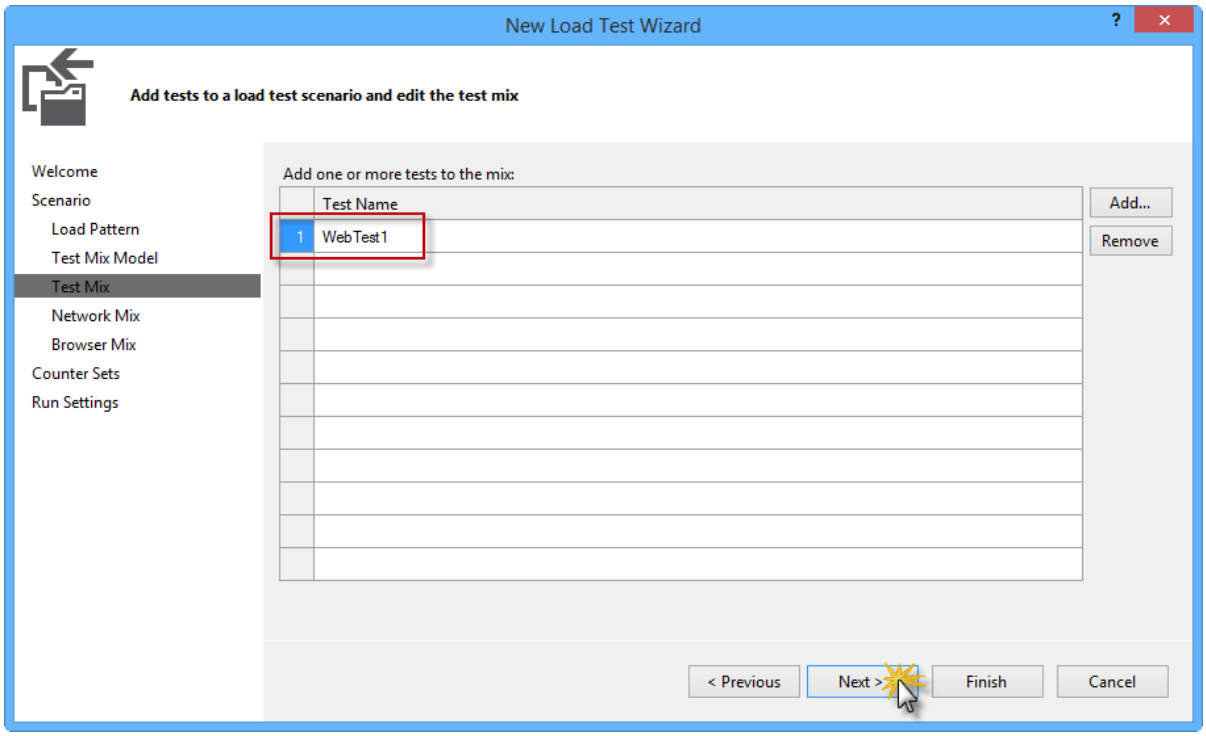
Adding a test to the test mix

14. In the **Add Tests** dialog box, double-click **WebTest1** to add the test to the **Selected tests** list. Click **OK** to continue.



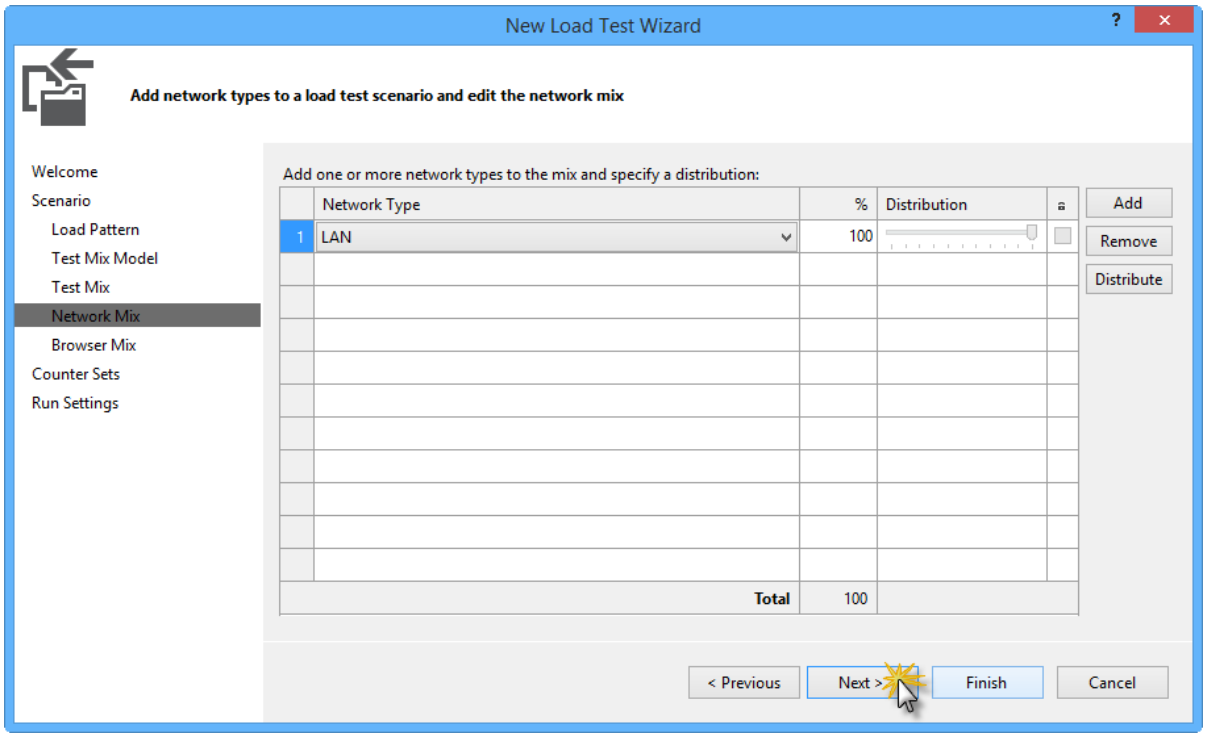
Adding the WebTest1 test

15. Back in the **Test Mix** page, click **Next**.



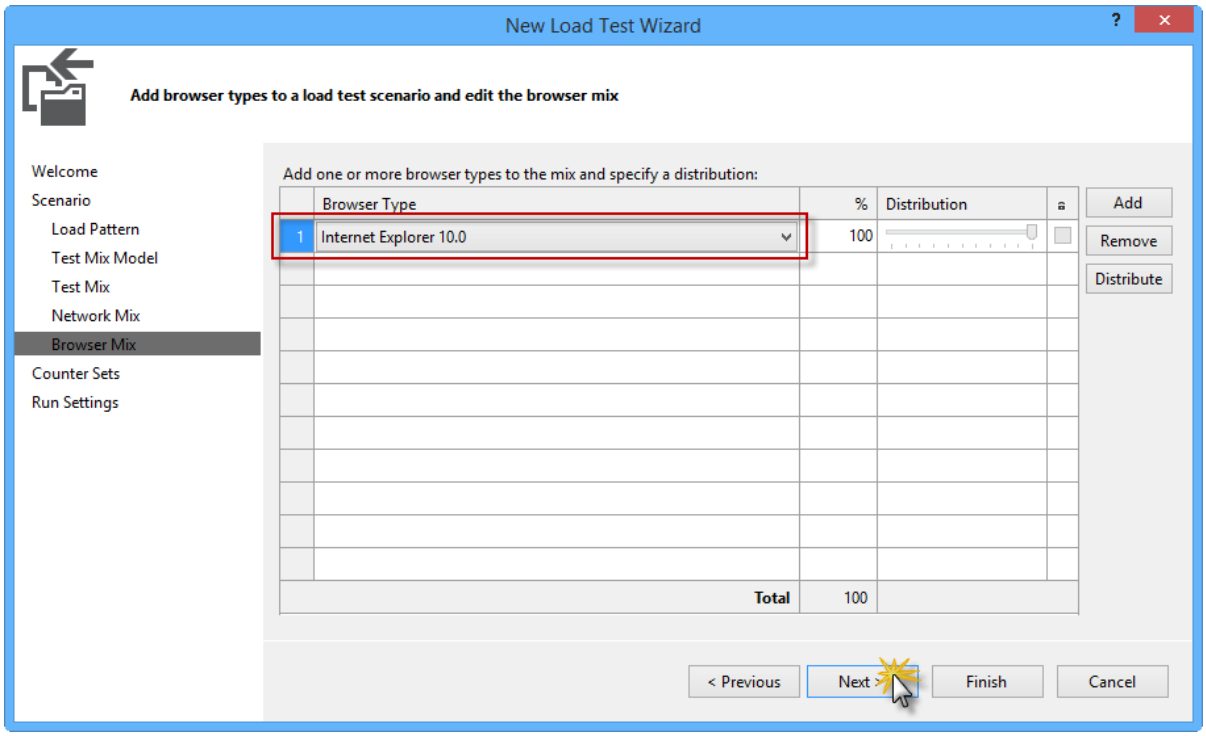
Completing the Test Mix page

16. In the **Network Mix** page, click **Next**.



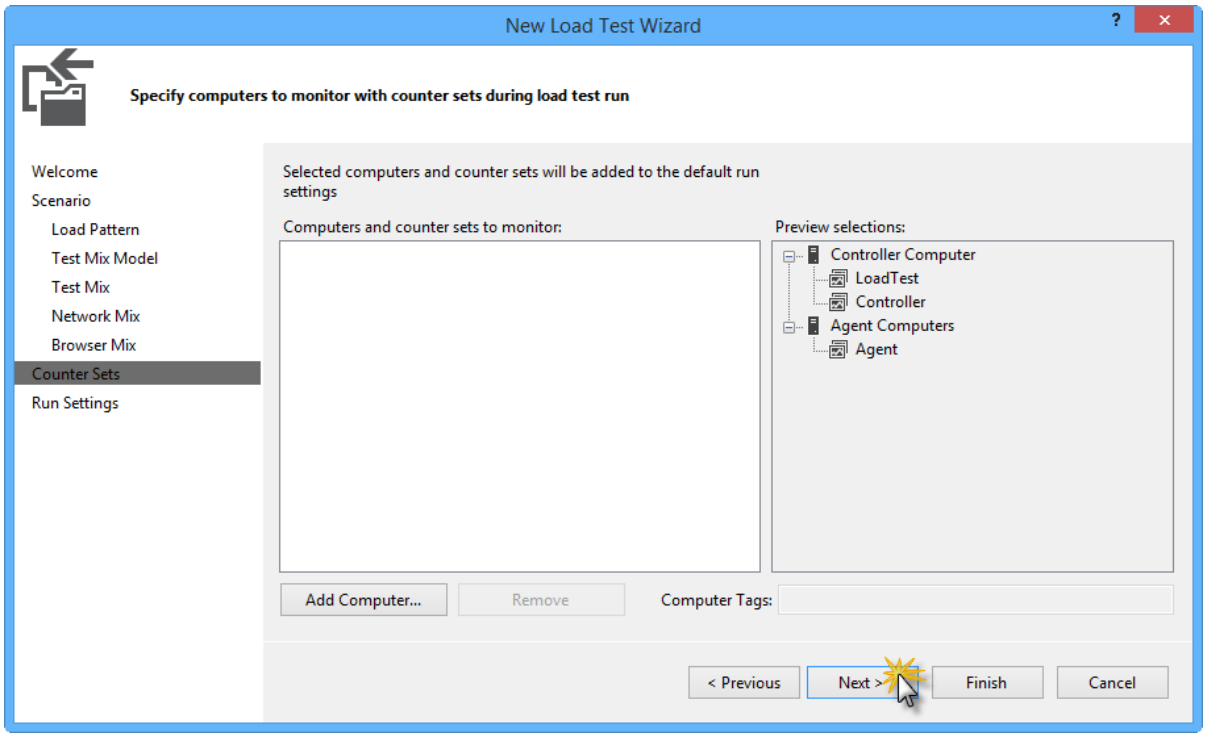
Clicking next in the Network Mix page

17. In the **Browser Mix** page, select **Internet Explorer 10.0** as the browser type and click **Next**.



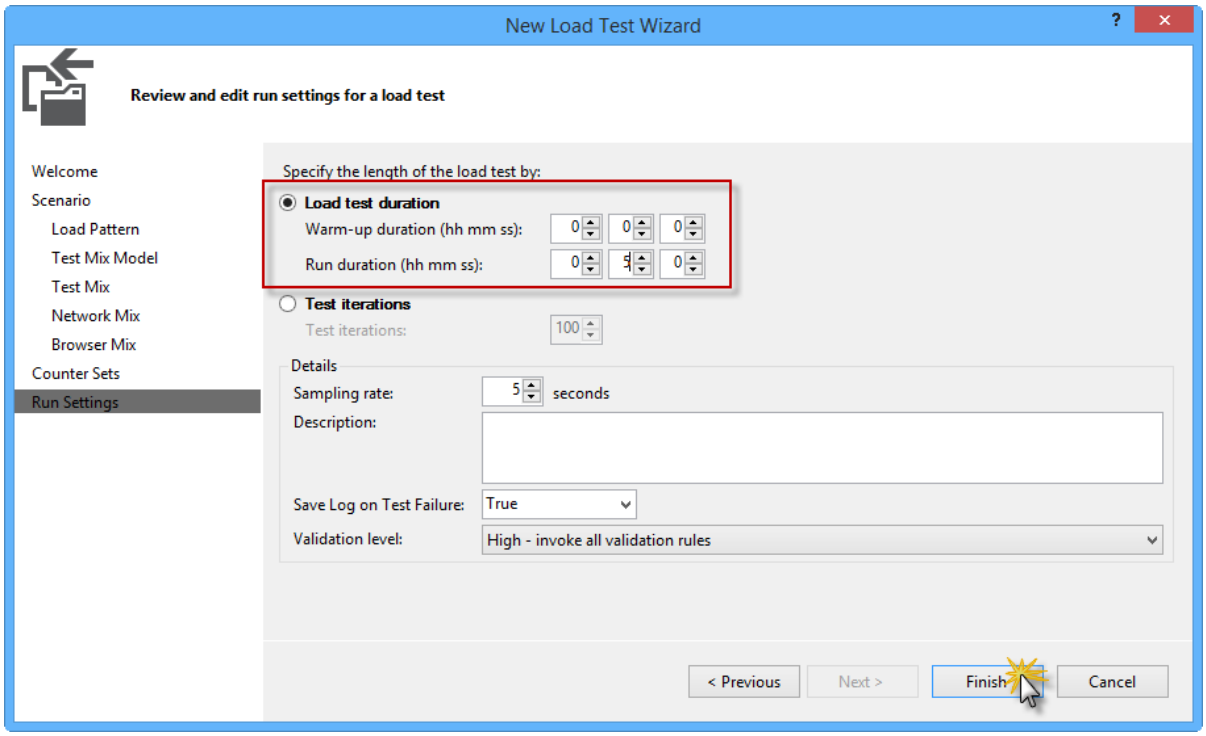
Selecting the browser type

18. In the **Counter Sets** page, click **Next**.



Clicking Next in the Counter Sets page

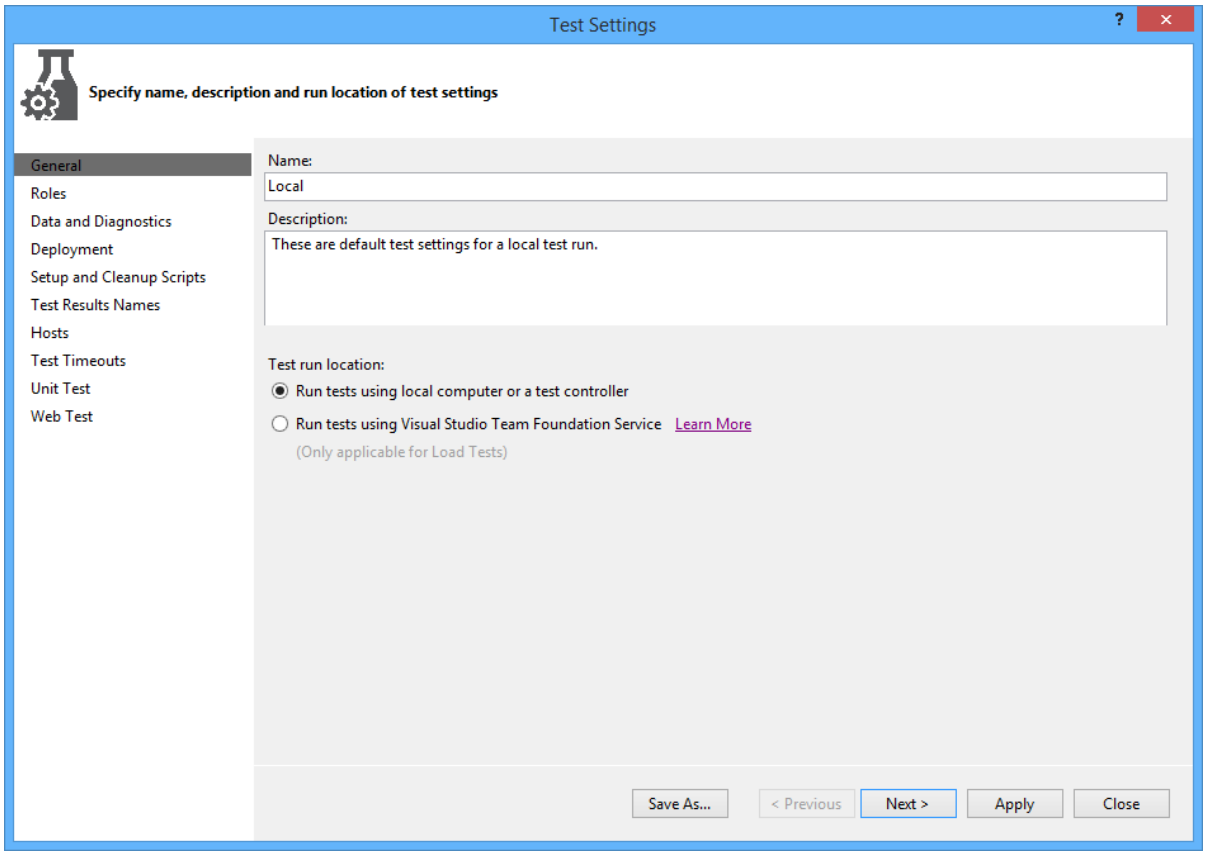
19. In the **Run Settings** page, set the **Load test duration** to **5 minutes** and click **Finish**.



Setting the load test duration to 5 minutes

20. In **Solution Explorer**, double-click the **Local.settings** file to explore the test settings. By default, Visual Studio uses your local computer to run the tests.

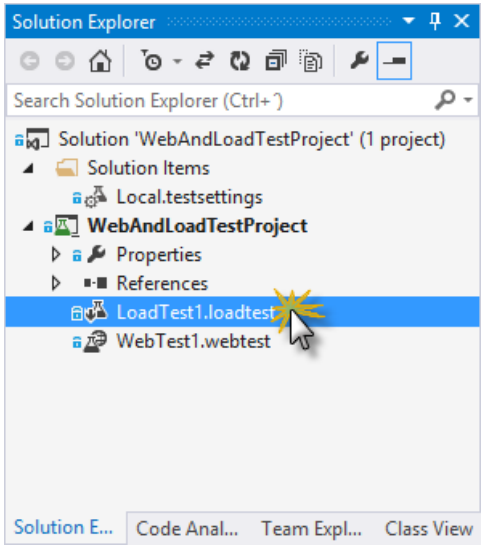
> **NOTE**
>
> Alternatively, you can configure your test project to run the load tests in the cloud using **Azure Test Plans**. Azure Test Plans provides a cloud-based load testing service that simulates a more realistic load, avoiding local environment constraints like CPU capacity, available memory, and network bandwidth. For more information about using Azure Test Plans to run load tests, see Load testing scenarios.



Task 3 – Autoscale Verification

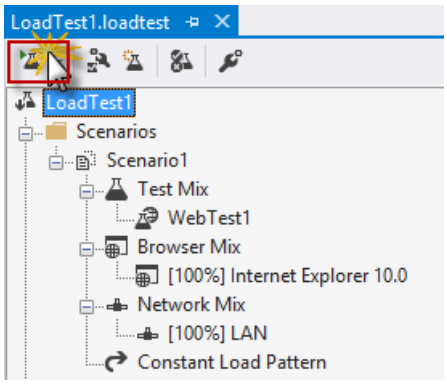
You will now execute the load test you created in the previous task and see how your web app behaves under load.

1. In **Solution Explorer**, double-click **LoadTest1.loadtest** to open the load test.



Opening LoadTest1.loadtest

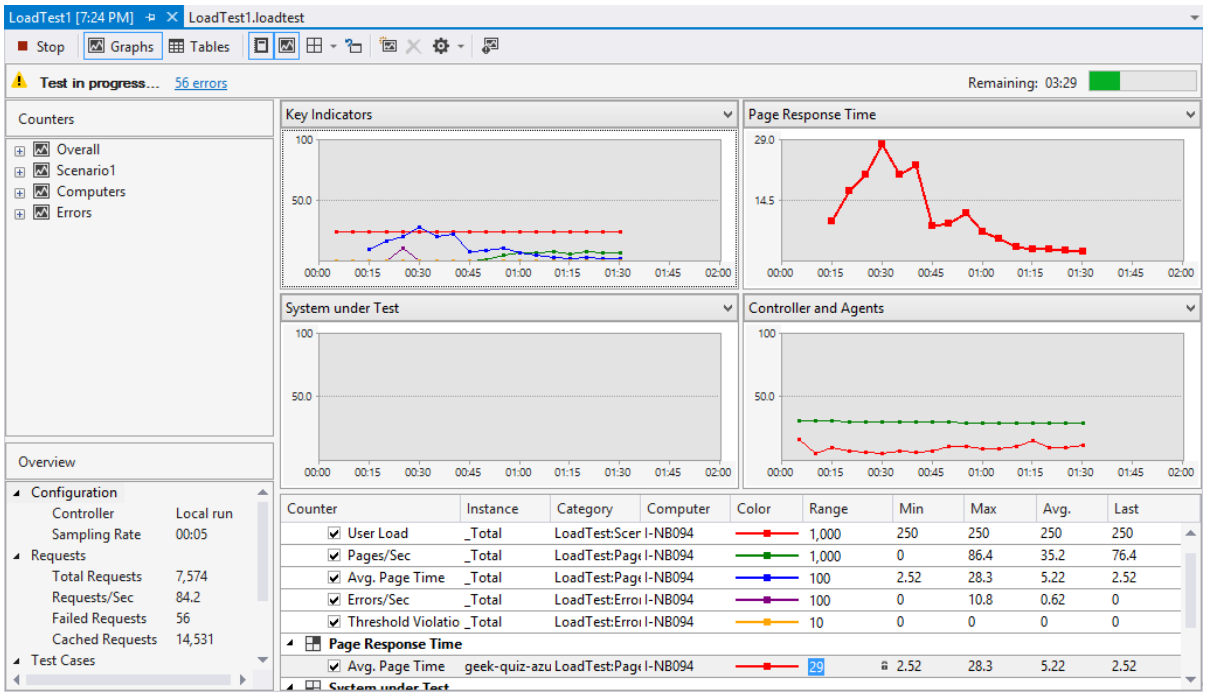
2. In the **LoadTest1.loadtest** window, click the first button in the toolbox to run the load test.



Running the load test

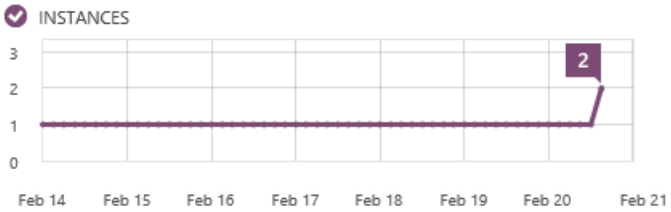
3. Wait until the load test completes.

> **NOTE**
>
> The load test simulates multiple users that send requests to the web app simultaneously. When the test is running, you can monitor the available counters to detect any errors, warnings or other information related to your load test run.



Load test running

4. Once the test completes, go back to the management portal and navigate to the **Scale** page of your web app. Under the **capacity** section, you should see in the graph that a new instance was automatically deployed.



New instance automatically deployed

> **NOTE**
>
> It may take several minutes for the changes to appear in the graph (press **CTRL + F5** periodically to refresh the page). If you do not see any changes, you can try the following:
>
> - Increase the duration of the load test (e.g. to **10 minutes**)
> - Reduce the maximum and minimum values of the **Target CPU** range in the Autoscale configuration of your web app
> - Run the load test in the cloud with **Azure Test Plans**. More information here

---

Summary

In this hands-on lab, you learned how to set up and deploy your application to production web apps in Azure. You started by detecting and updating your databases using **Entity Framework Code First Migrations**, then continued by deploying new versions of your site using **Git** and performing rollbacks to the latest stable version of your site. Additionally, you learned how to scale your app using Storage to move your static content to a Blob container.

OWIN and Katana

2/20/2020 • 2 minutes to read • Edit Online

Katana is a flexible set of components for building and hosting Open Web Interface for .NET (OWIN)-based web applications.

- An Overview of Project Katana
- Getting Started with OWIN and Katana
- Katana Samples
- OWIN Startup Class Detection
- Enabling Windows Authentication in Katana
- OWIN Middleware in the IIS Integrated Pipeline
- OWIN OAuth 2.0 Authorization Server
- Host OWIN in an Azure Worker Role

An Overview of Project Katana

2/20/2020 • 21 minutes to read • Edit Online

by Howard Dierking

The ASP.NET Framework has been around for over ten years, and the platform has enabled the development of countless Web sites and services. As Web application development strategies have evolved, the framework has been able to evolve in step with technologies like ASP.NET MVC and ASP.NET Web API. As Web application development takes its next evolutionary step into the world of cloud computing, project Katana provides the underlying set of components to ASP.NET applications, enabling them to be flexible, portable, lightweight, and provide better performance – put another way, project Katana cloud optimizes your ASP.NET applications.

Why Katana – Why Now?

Regardless whether one is discussing a developer framework or end-user product, it's important to understand the underlying motivations for creating the product – and part of that includes knowing who the product was created for. ASP.NET was originally created with two customers in mind.

The first group of customers was classic ASP developers. At the time, ASP was one of the primary technologies for creating dynamic, data-driven Web sites and applications by interweaving markup and server-side script. The ASP runtime supplied server-side script with a set of objects that abstracted core aspects of the underlying HTTP protocol and Web server and provided access to additional services such session and application state management, cache, etc. While powerful, classic ASP applications became a challenge to manage as they grew in size and complexity. This was largely due to the lack of structure found in scripting environments coupled with the duplication of code resulting from the interleaving of code and markup. In order to capitalize on the strengths of classic ASP while addressing some of its challenges, ASP.NET took advantage of the code organization provided by the object-oriented languages of the .NET Framework while also preserving the server-side programming model to which classic ASP developers had grown accustomed.

The second group of target customers for ASP.NET was Windows business application developers. Unlike classic ASP developers, who were accustomed to writing HTML markup and the code to generate more HTML markup, WinForms developers (like the VB6 developers before them) were accustomed to a design time experience that included a canvas and a rich set of user interface controls. The first version of ASP.NET – also known as "Web Forms" provided a similar design time experience along with a server-side event model for user interface components and a set of infrastructure features (such as ViewState) to create a seamless developer experience between client and server side programming. Web Forms effectively hid the Web's stateless nature under a stateful event model that was familiar to WinForms developers.

Challenges Raised by the Historical Model

The net result was a mature, feature-rich runtime and developer programming model. However, with that feature-richness came a couple notable challenges. Firstly, the framework was **monolithic**, with logically disparate units of functionality being tightly coupled in the same System.Web.dll assembly (for example, the core HTTP objects with the Web forms framework). Secondly, ASP.NET was included as a part of the larger .NET Framework, which meant that the **time between releases was on the order of years.** This made it difficult for ASP.NET to keep pace with all of the changes happening in rapidly evolving Web development. Finally, System.Web.dll itself was coupled in a few different ways to a specific Web hosting option: Internet Information Services (IIS).

Evolutionary steps: ASP.NET MVC and ASP.NET Web API

And lots of change was happening in Web development! Web applications were increasingly being developed as a series of small, focused components rather than large frameworks. The number of components as well as the frequency with which they were released was increasing at an ever faster rate. It was clear that keeping pace with the Web would require frameworks to get smaller, decoupled and more focused rather than larger and more feature-rich, therefore the **ASP.NET team took several evolutionary steps to enable ASP.NET as a family of pluggable Web components rather than a single framework**.

One of the early changes was the rise in popularity of the well-known model-view-controller (MVC) design pattern thanks to Web development frameworks like Ruby on Rails. This style of building Web applications gave the developer greater control over her application's markup while still preserving the separation of markup and business logic, which was one of the initial selling points for ASP.NET. To meet the demand for this style of Web application development, Microsoft took the opportunity to position itself better for the future by **developing ASP.NET MVC out of band** (and not including it in the .NET Framework). ASP.NET MVC was released as an independent download. This gave the engineering team the flexibility to deliver updates much more frequently than had been previously possible.

Another major shift in Web application development was the shift from dynamic, server-generated Web pages to static initial markup with dynamic sections of the page generated from client-side script communicating **with backend Web APIs through AJAX requests**. This architectural shift helped propel the rise of Web APIs, and the development of the ASP.NET Web API framework. As in the case of ASP.NET MVC, the release of ASP.NET Web API provided another opportunity to evolve ASP.NET further as a more modular framework. The engineering team took advantage of the opportunity and **built ASP.NET Web API such that it had no dependencies on any of the core framework types found in System.Web.dll**. This enabled two things: first, it meant that ASP.NET Web API could evolve in a completely self-contained manner (and it could continue to iterate quickly because it is delivered via NuGet). Second, because there were no external dependencies to System.Web.dll, and therefore, no dependencies to IIS, ASP.NET Web API included the capability to run in a custom host (for example, a console application, Windows service, etc.)

The Future: A Nimble Framework

By decoupling framework components from one another and then releasing them on NuGet, frameworks could now **iterate more independently and more quickly**. Additionally, the power and flexibility of Web API's self-hosting capability proved very attractive to developers who wanted a **small, lightweight host** for their services. It proved so attractive, in fact, that other frameworks also wanted this capability, and this surfaced a new challenge in that each framework ran in its own host process on its own base address and needed to be managed (started, stopped, etc.) independently. A modern Web application generally supports static file serving, dynamic page generation, Web API, and more recently real-time/push notifications. Expecting that each of these services should be run and managed independently was simply not realistic.

What was needed was a single hosting abstraction that would enable a developer to compose an application from a variety of different components and frameworks, and then run that application on a supporting host.

The Open Web Interface for .NET (OWIN)

Inspired by the benefits achieved by Rack in the Ruby community, several members of the .NET community set out to create an abstraction between Web servers and framework components. Two design goals for the OWIN abstraction were that it was simple and that it took the fewest possible dependencies on other framework types. These two goals help ensure:

- New components could be more easily developed and consumed.
- Applications could be more easily ported between hosts and potentially entire platforms/operating systems.

The resulting abstraction consists of two core elements. The first is the environment dictionary. This data structure is responsible for storing all of the state necessary for processing an HTTP request and response, as well as any relevant server state. The environment dictionary is defined as follows:

```
IDictionary<string, object>
```

An OWIN-compatible Web server is responsible for populating the environment dictionary with data such as the body streams and header collections for an HTTP request and response. It is then the responsibility of the application or framework components to populate or update the dictionary with additional values and write to the response body stream.

In addition to specifying the type for the environment dictionary, the OWIN specification defines a list of core dictionary key value pairs. For example, the following table shows the required dictionary keys for an HTTP request:

KEY NAME	VALUE DESCRIPTION
`"owin.RequestBody"`	A Stream with the request body, if any. Stream.Null MAY be used as a placeholder if there is no request body. See Request Body.
`"owin.RequestHeaders"`	An `IDictionary<string, string[]>` of request headers. See Headers.
`"owin.RequestMethod"`	A `string` containing the HTTP request method of the request (e.g., `"GET"`, `"POST"`).
`"owin.RequestPath"`	A `string` containing the request path. The path MUST be relative to the "root" of the application delegate; see Paths.
`"owin.RequestPathBase"`	A `string` containing the portion of the request path corresponding to the "root" of the application delegate; see Paths.
`"owin.RequestProtocol"`	A `string` containing the protocol name and version (e.g. `"HTTP/1.0"` or `"HTTP/1.1"`).
`"owin.RequestQueryString"`	A `string` containing the query string component of the HTTP request URI, without the leading "?" (e.g., `"foo=bar&baz=quux"`). The value may be an empty string.
`"owin.RequestScheme"`	A `string` containing the URI scheme used for the request (e.g., `"http"`, `"https"`); see URI Scheme.

The second key element of OWIN is the application delegate. This is a function signature which serves as the primary interface between all components in an OWIN application. The definition for the application delegate is as follows:

```
Func<IDictionary<string, object>, Task>;
```

The application delegate then is simply an implementation of the Func delegate type where the function accepts the environment dictionary as input and returns a Task. This design has several implications for developers:

- There are a very small number of type dependencies required in order to write OWIN components. This greatly increases the accessibility of OWIN to developers.
- The asynchronous design enables the abstraction to be efficient with its handling of computing resources, particularly in more I/O intensive operations.
- Because the application delegate is an atomic unit of execution and because the environment dictionary is

carried as a parameter on the delegate, OWIN components can be easily chained together to create complex HTTP processing pipelines.

From an implementation perspective, OWIN is a specification (http://owin.org/html/owin.html). Its goal is not to be the next Web framework, but rather a specification for how Web frameworks and Web servers interact.

If you've investigated OWIN or Katana, you may also have noticed the Owin NuGet package and Owin.dll. This library contains a single interface, IAppBuilder, which formalizes and codifies the startup sequence described in section 4 of the OWIN specification. While not required in order to build OWIN servers, the IAppBuilder interface provides a concrete reference point, and it is used by the Katana project components.

Project Katana

Whereas both the OWIN specification and *Owin.dll* are community owned and community run open source efforts, the Katana project represents the set of OWIN components that, while still open source, are built and released by Microsoft. These components include both infrastructure components, such as hosts and servers, as well as functional components, such as authentication components and bindings to frameworks such as SignalR and ASP.NET Web API. The project has the following three high level goals:

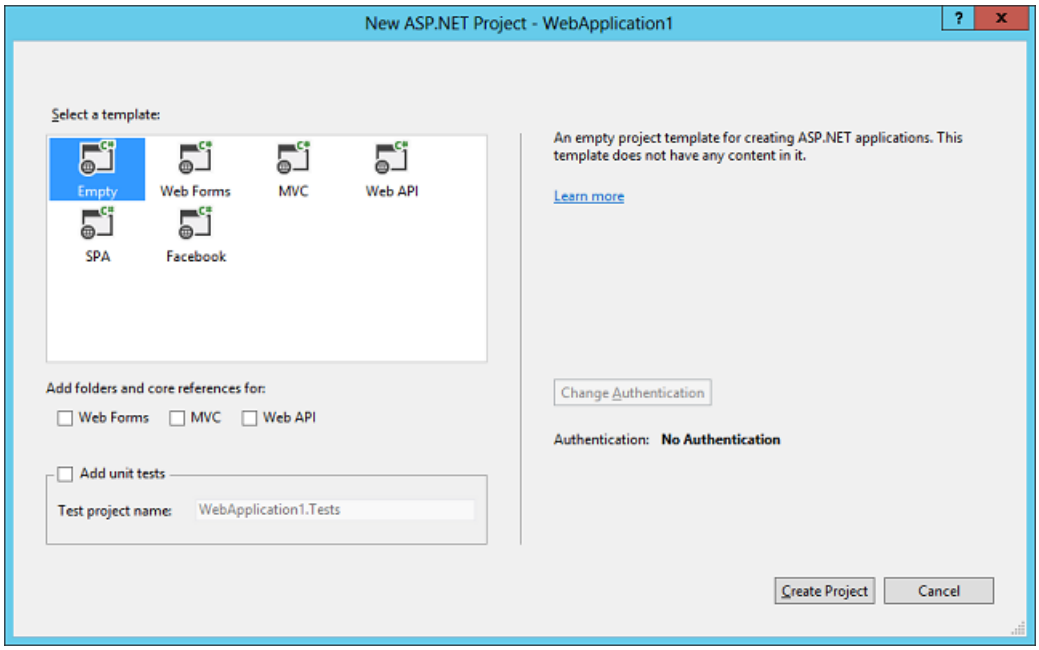
- **Portable** – Components should be able to be easily substituted for new components as they become available. This includes all types of components, from the framework to the server and host. The implication of this goal is that third party frameworks can seamlessly run on Microsoft servers while Microsoft frameworks can potentially run on third party servers and hosts.
- **Modular/flexible**– Unlike many frameworks which include a myriad of features that are turned on by default, Katana project components should be small and focused, giving control over to the application developer in determining which components to use in her application.
- **Lightweight/performant/scalable** – By breaking the traditional notion of a framework into a set of small, focused components which are added explicitly by the application developer, a resulting Katana application can consume fewer computing resources, and as a result, handle more load, than with other types of servers and frameworks. As the requirements of the application demand more features from the underlying infrastructure, those can be added to the OWIN pipeline, but that should be an explicit decision on the part of the application developer. Additionally, the substitutability of lower level components means that as they become available, new high performance servers can seamlessly be introduced to improve the performance of OWIN applications without breaking those applications.

Getting Started with Katana Components

When it was first introduced, one aspect of the Node.js framework that immediately drew people's attention was the simplicity with which one could author and run a Web server. If Katana goals were framed in light of Node.js, one might summarize them by saying that Katana brings many of the benefits of Node.js (and frameworks like it) without forcing the developer to throw out everything she knows about developing ASP.NET Web applications. For this statement to hold true, getting started with the Katana project should be equally simple in nature to Node.js.

Creating "Hello World!"

One notable difference between JavaScript and .NET development is the presence (or absence) of a compiler. As such, the starting point for a simple Katana server is a Visual Studio project. However, we can start with the most minimal of project types: the Empty ASP.NET Web Application.



Next, we will install the Microsoft.Owin.Host.SystemWeb NuGet package into the project. This package provides an OWIN server that runs in the ASP.NET request pipeline. It can be found on the NuGet gallery and can be installed using either the Visual Studio package manager dialog or the package manager console with the following command:

```
install-package Microsoft.Owin.Host.SystemWeb
```

Installing the `Microsoft.Owin.Host.SystemWeb` package will install a few additional packages as dependencies. One of those dependencies is `Microsoft.Owin`, a library which provides several helper types and methods for developing OWIN applications. We can use those types to quickly write the following "hello world" server.

```
public class Startup
{
   public void Configuration(IAppBuilder app)
   {
      app.Run(context =>
      {
         context.Response.ContentType = "text/plain";
         return context.Response.WriteAsync("Hello World!");
      });
   }
}
```

This very simple Web server can now be run using Visual Studio's **F5** command and includes full support for debugging.

Switching hosts

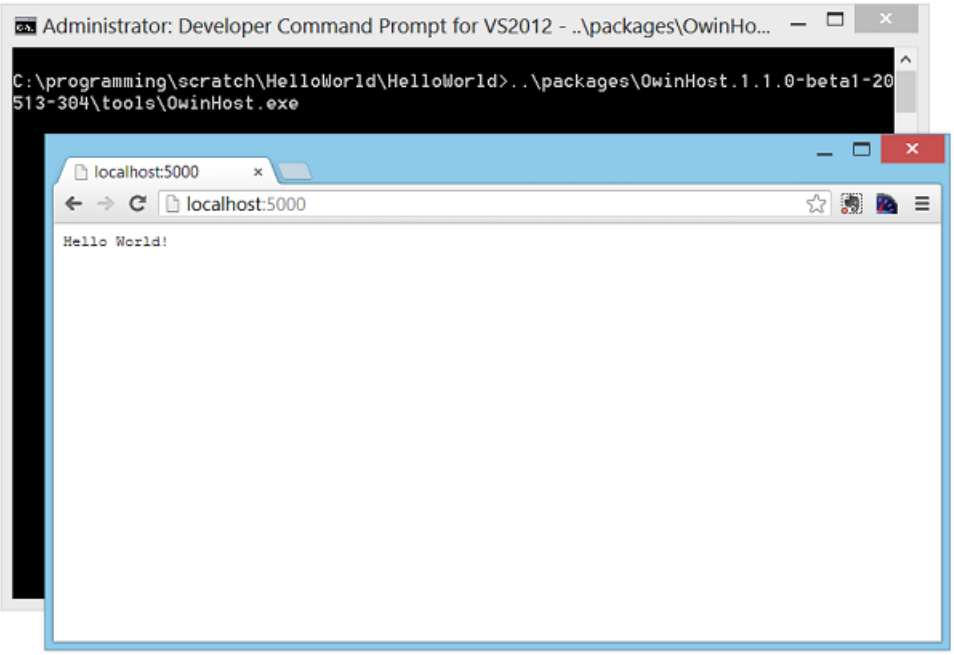
By default, the previous "hello world" example runs in the ASP.NET request pipeline, which uses System.Web in the context of IIS. This can by itself add tremendous value as it enables us to benefit from the flexibility and composability of an OWIN pipeline with the management capabilities and overall maturity of IIS. However, there may be cases where the benefits provided by IIS are not required and the desire is for a smaller, more lightweight host. What is needed, then, to run our simple Web server outside of IIS and System.Web?

To illustrate the portability goal, moving from a Web-server host to a command line host requires simply adding the new server and host dependencies to project's output folder and then starting the host. In this example, we'll host our Web server in a Katana host called `OwinHost.exe` and will use the Katana HttpListener-based server.

Similarly to the other Katana components, these will be acquired from NuGet using the following command:

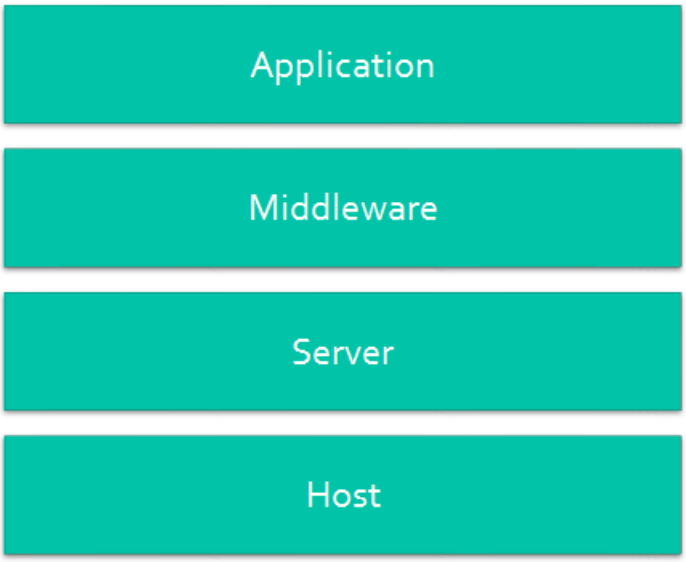
```
install-package OwinHost
```

From the command line, we can then navigate to the project root folder and simply run the `OwinHost.exe` (which was installed in the tools folder of its respective NuGet package). By default, `OwinHost.exe` is configured to look for the HttpListener-based server and so no additional configuration is needed. Navigating in a Web browser to `http://localhost:5000/` shows the application now running through the console.



Katana Architecture

The Katana component architecture divides an application into four logical layers, as depicted below: *host, server, middleware,* and *application*. The component architecture is factored in such a way that implementations of these layers can be easily substituted, in many cases, without requiring recompilation of the application.



Host

The host is responsible for:

- Managing the underlying process.

- Orchestrating the workflow that results in the selection of a server and the construction of an OWIN pipeline through which requests will be handled.

  At present, there are 3 primary hosting options for Katana-based applications:

IIS/ASP.NET: Using the standard HttpModule and HttpHandler types, OWIN pipelines can run on IIS as a part of an ASP.NET request flow. ASP.NET hosting support is enabled by installing the Microsoft.AspNet.Host.SystemWeb NuGet package into a Web application project. Additionally, because IIS acts as both a host and a server, the OWIN server/host distinction is conflated in this NuGet package, meaning that if using the SystemWeb host, a developer cannot substitute an alternate server implementation.

Custom Host: The Katana component suite gives a developer the ability to host applications in her own custom process, whether that is a console application, Windows service, etc. This capability looks similar to the self-host capability provided by Web API. The following example shows a custom host of Web API code:

```
static void Main()
{
    var baseAddress = new Uri("http://localhost:5000");

    var config = new HttpSelfHostConfiguration(baseAddress);
    config.Routes.MapHttpRoute("default", "{controller}");

    using (var svr = new HttpSelfHostServer(config))
    {
        svr.OpenAsync().Wait();
        Console.WriteLine("Press Enter to quit.");
        Console.ReadLine();
    }
}
```

The self-host setup for a Katana application is similar:

```
static void Main(string[] args)
{
    const string baseUrl = "http://localhost:5000/";

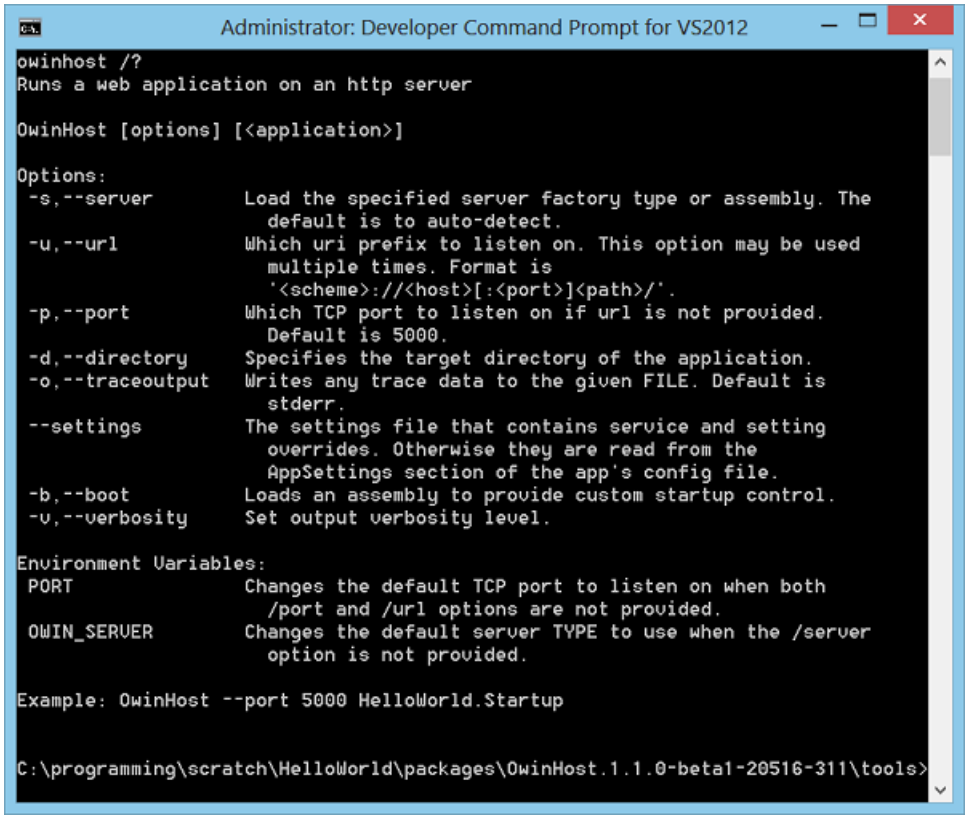
    using (WebApplication.Start<Startup>(new StartOptions { Url = baseUrl }))
    {
        Console.WriteLine("Press Enter to quit.");
        Console.ReadKey();
    }
}
```

One notable difference between the Web API and Katana self-host examples is that the Web API configuration code is missing from the Katana self-host example. In order to enable both portability and composability, Katana separates the code that starts the server from the code that configures the request processing pipeline. The code that configures Web API, then is contained in the class Startup, which is additionally specified as the type parameter in WebApplication.Start.

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        var config = new HttpConfiguration();
        config.Routes.MapHttpRoute("default", "{controller}");
        app.UseWebApi(config);
    }
}
```

The startup class will be discussed in greater detail later in the article. However, the code required to start a Katana self-host process looks strikingly similar to the code that you may be using today in ASP.NET Web API self-host applications.

OwinHost.exe: While some will want to write a custom process to run Katana Web applications, many would prefer to simply launch a pre-built executable that can start a server and run their application. For this scenario, the Katana component suite includes `OwinHost.exe`. When run from within a project's root directory, this executable will start a server (it uses the HttpListener server by default) and use conventions to find and run the user's startup class. For more granular control, the executable provides a number of additional command line parameters.



Server

While the host is responsible for starting and maintaining process within which the application runs, the responsibility of the server is to open a network socket, listen for requests, and send them through the pipeline of OWIN components specified by the user (as you may have already noticed, this pipeline is specified in the application developer's Startup class). Currently, the Katana project includes two server implementations:

- **Microsoft.Owin.Host.SystemWeb**: As previously mentioned, IIS in concert with the ASP.NET pipeline acts as both a host and a server. Therefore, when choosing this hosting option, IIS both manages host-level concerns such as process activation and listens for HTTP requests. For ASP.NET Web applications, it then sends the requests into the ASP.NET pipeline. The Katana SystemWeb host registers an ASP.NET HttpModule and HttpHandler to intercept requests as they flow through the HTTP pipeline and send them through the user-specified OWIN pipeline.
- **Microsoft.Owin.Host.HttpListener**: As its name indicates, this Katana server uses the .NET Framework's HttpListener class to open a socket and send requests into a developer-specified OWIN pipeline. This is currently the default server selection for both the Katana self-host API and OwinHost.exe.

Middleware/framework

As previously mentioned, when the server accepts a request from a client, it is responsible for passing it through a pipeline of OWIN components, which are specified by the developer's startup code. These pipeline

components are known as middleware.
At a very basic level, an OWIN middleware component simply needs to implement the OWIN application delegate so that it is callable.

```
Func<IDictionary<string, object>, Task>
```

However, in order to simplify the development and composition of middleware components, Katana supports a handful of conventions and helper types for middleware components. The most common of these is the `OwinMiddleware` class. A custom middleware component built using this class would look similar to the following:

```
public class LoggerMiddleware : OwinMiddleware
{
    private readonly ILog _logger;

    public LoggerMiddleware(OwinMiddleware next, ILog logger) : base(next)
    {
        _logger = logger;
    }

    public override async Task Invoke(IOwinContext context)
    {
        _logger.LogInfo("Middleware begin");
        await this.Next.Invoke(context);
        _logger.LogInfo("Middleware end");
    }
}
```

This class derives from `OwinMiddleware`, implements a constructor that accepts an instance of the next middleware in the pipeline as one of its arguments, and then passes it to the base constructor. Additional arguments used to configure the middleware are also declared as constructor parameters after the next middleware parameter.

At runtime, the middleware is executed via the overridden `Invoke` method. This method takes a single argument of type `OwinContext`. This context object is provided by the `Microsoft.Owin` NuGet package described earlier and provides strongly-typed access to the request, response and environment dictionary, along with a few additional helper types.

The middleware class can be easily added to the OWIN pipeline in the application startup code as follows:

```
public class Startup
{
   public void Configuration(IAppBuilder app)
   {
      app.Use<LoggerMiddleware>(new TraceLogger());

   }
}
```

Because the Katana infrastructure simply builds up a pipeline of OWIN middleware components, and because the components simply need to support the application delegate to participate in the pipeline, middleware components can range in complexity from simple loggers to entire frameworks like ASP.NET, Web API, or SignalR. For example, adding ASP.NET Web API to the previous OWIN pipeline requires adding the following startup code:

```
public class Startup
{
   public void Configuration(IAppBuilder app)
   {
      app.Use<LoggerMiddleware>(new TraceLogger());

      var config = new HttpConfiguration();
      // configure Web API
      app.UseWebApi(config);

      // additional middleware registrations
   }
}
```

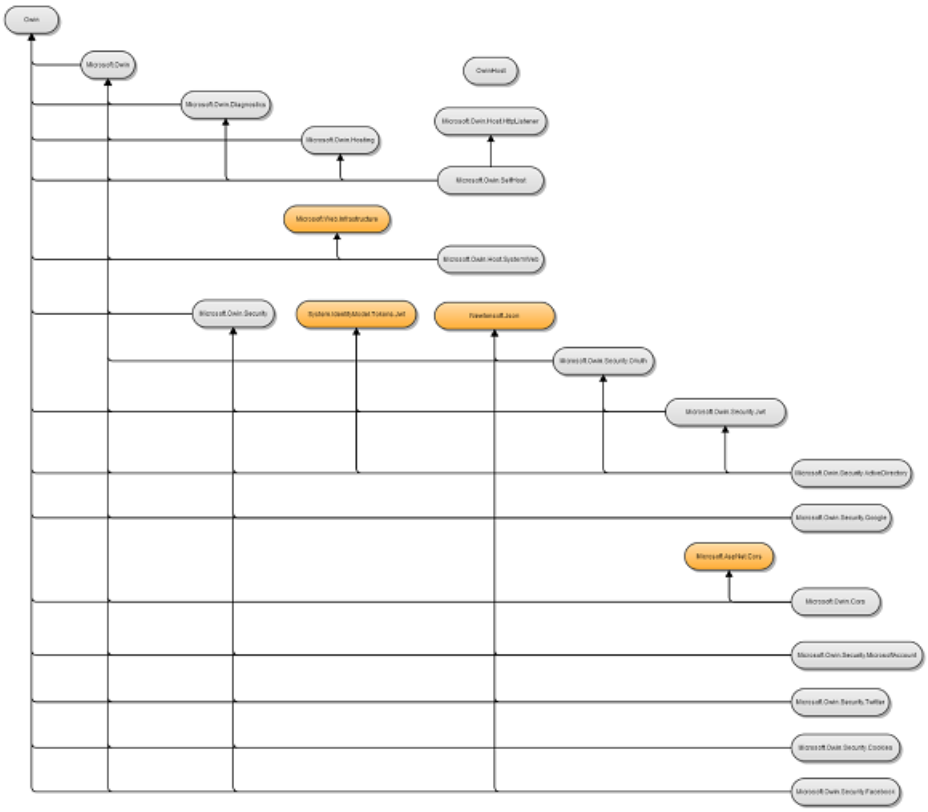
The Katana infrastructure will build the pipeline of middleware components based on the order in which they were added to the IAppBuilder object in the Configuration method. In our example, then, LoggerMiddleware can handle all requests that flow through the pipeline, regardless of how those requests are ultimately handled. This enables powerful scenarios where a middleware component (e.g. an authentication component) can process requests for a pipeline that includes multiple components and frameworks (e.g. ASP.NET Web API, SignalR, and a static file server).

Applications

As illustrated by the previous examples, OWIN and the Katana project should not be thought of as a new application programming model, but rather as an abstraction to decouple application programming models and frameworks from server and hosting infrastructure. For example, when building Web API applications, the developer framework will continue to use the ASP.NET Web API framework, irrespective of whether or not the application runs in an OWIN pipeline using components from the Katana project. The one place where OWIN-related code will be visible to the application developer will be the application startup code, where the developer composes the OWIN pipeline. In the startup code, the developer will register a series of UseXx statements, generally one for each middleware component that will process incoming requests. This experience will have the same effect as registering HTTP modules in the current System.Web world. Typically, a larger framework middleware, such as ASP.NET Web API or SignalR will be registered at the end of the pipeline. Cross-cutting middleware components, such as those for authentication or caching, are generally registered towards the beginning of the pipeline so that they will process requests for all of the frameworks and components registered later in the pipeline. This separation of the middleware components from each other and from the underlying infrastructure components enables the components to evolve at different velocities while ensuring that the overall system remains stable.

Components – NuGet Packages

Like many current libraries and frameworks, the Katana project components are delivered as a set of NuGet packages. For the upcoming version 2.0, the Katana package dependency graph looks as follows. (Click on image for larger view.)



Nearly every package in the Katana project depends, directly or indirectly, on the Owin package. You may remember that this is the package that contains the IAppBuilder interface, which provides a concrete implementation of the application startup sequence described in section 4 of the OWIN specification. Additionally, many of the packages depend on Microsoft.Owin, which provides a set of helper types for working with HTTP requests and responses. The remainder of the package can be classified as either hosting infrastructure packages (servers or hosts) or middleware. Packages and dependencies that are external to the Katana project are displayed in orange.

The hosting infrastructure for Katana 2.0 includes both the SystemWeb and HttpListener-based servers, the OwinHost package for running OWIN applications using OwinHost.exe, and the Microsoft.Owin.Hosting package for self-hosting OWIN applications in a custom host (e.g. console application, Windows service, etc.)

For Katana 2.0, the middleware components are primarily focused on providing different means of authentication. One additional middleware component for diagnostics is provided, which enables support for a start and error page. As OWIN grows into the de facto hosting abstraction, the ecosystem of middleware components, both those developed by Microsoft and third parties, will also grow in number.

Conclusion

From its beginning, the Katana project's goal has not been to create and thereby force developers to learn yet another Web framework. Rather, the goal has been to create an abstraction to give .NET Web application developers more choice than has previously been possible. By breaking up the logical layers of a typical Web application stack into a set of replaceable components, the Katana project enables components throughout the stack to improve at whatever rate makes sense for those components. By building all components around the simple OWIN abstraction, Katana enables frameworks and the applications built on top of them to be portable across a variety of different servers and hosts. By putting the developer in control of the stack, Katana ensures that the developer makes the ultimate choice about how lightweight or how feature-rich her Web stack should be.

For more information about Katana

- The Katana project on GitHub: https://github.com/aspnet/AspNetKatana/.
- Video: The Katana Project - OWIN for ASP.NET, by Howard Dierking.

Acknowledgements

- Rick Anderson: (twitter @RickAndMSFT) Rick is a senior programming writer for Microsoft focusing on Azure and MVC.
- Scott Hanselman: (twitter @shanselman)
- Jon Galloway: (twitter @jongalloway)

Getting Started with OWIN and Katana

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson

Open Web Interface for .NET (OWIN) defines an abstraction between .NET web servers and web applications. By decoupling the web server from the application, OWIN makes it easier to create middleware for .NET web development. Also, OWIN makes it easier to port web applications to other hosts—for example, self-hosting in a Windows service or other process.

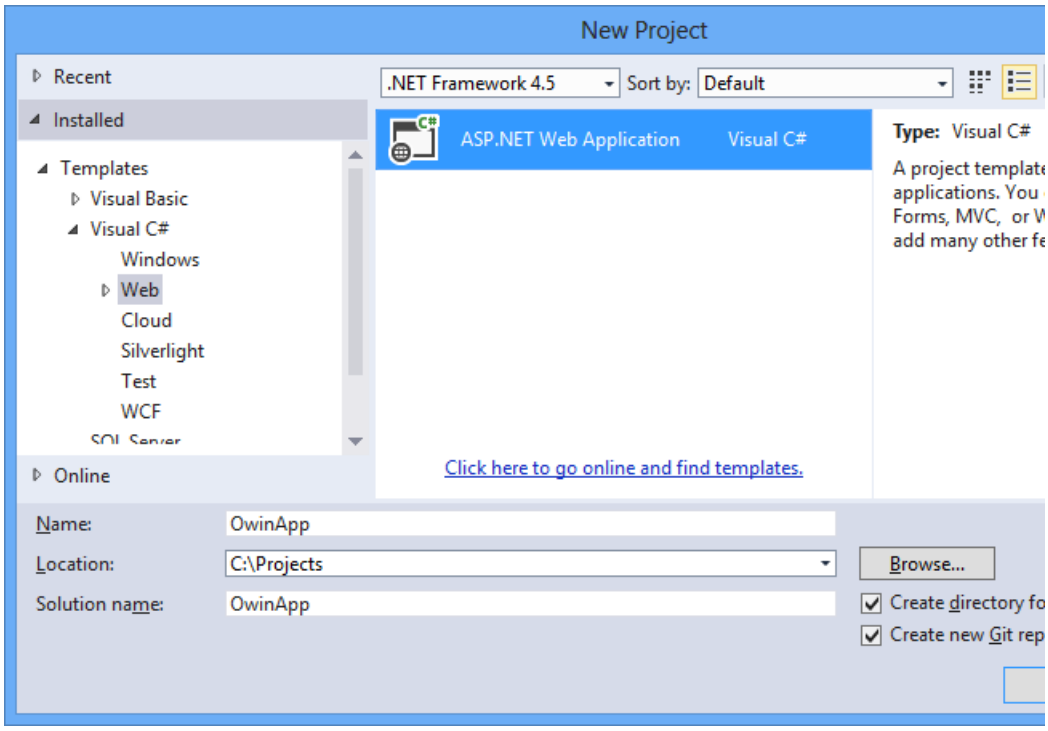
OWIN is a community-owned specification, not an implementation. The Katana project is a set of open-source OWIN components developed by Microsoft. For a general overview of both OWIN and Katana, see An Overview of Project Katana. In this article, I will jump right into code to get started.

This tutorial uses Visual Studio 2013 Release Candidate, but you can also use Visual Studio 2012. A few of the steps are different in Visual Studio 2012, which I note below.

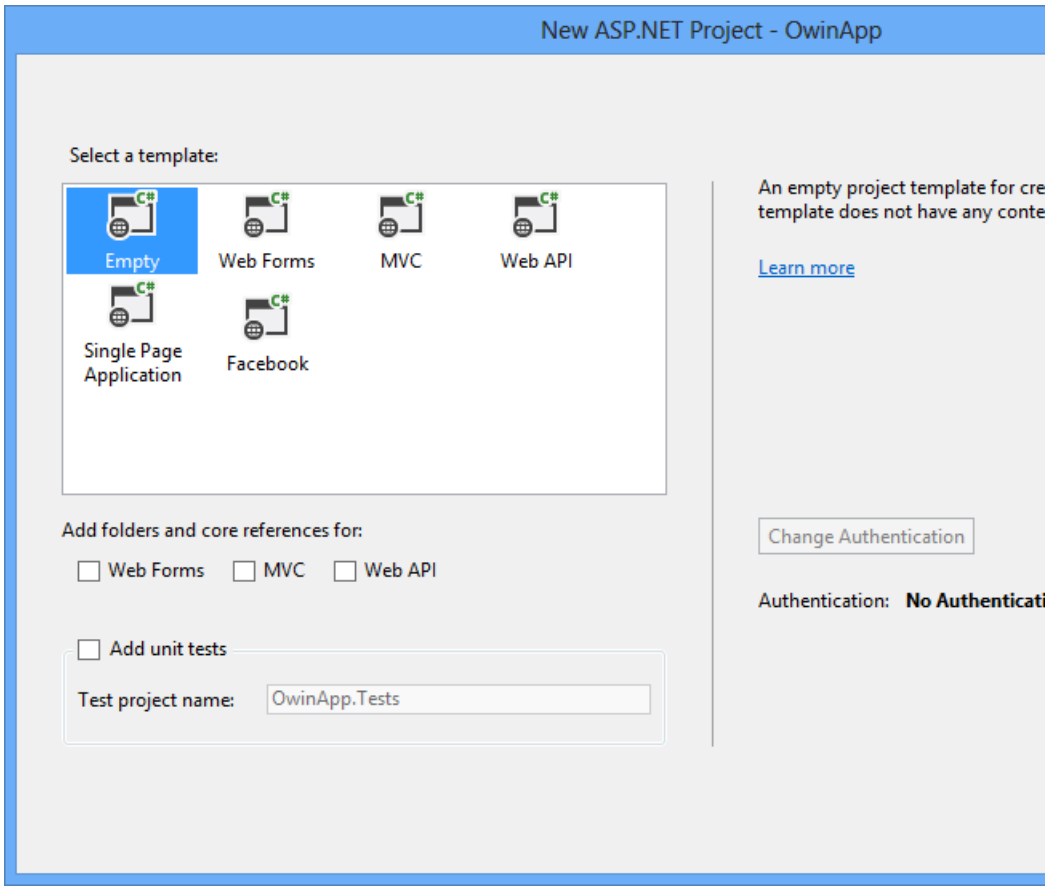
Host OWIN in IIS

In this section, we'll host OWIN in IIS. This option gives you the flexibility and composability of an OWIN pipeline together with the mature feature set of IIS. Using this option, the OWIN application runs in the ASP.NET request pipeline.

First, create a new ASP.NET Web Application project. (In Visual Studio 2012, use the ASP.NET Empty Web Application project type.)



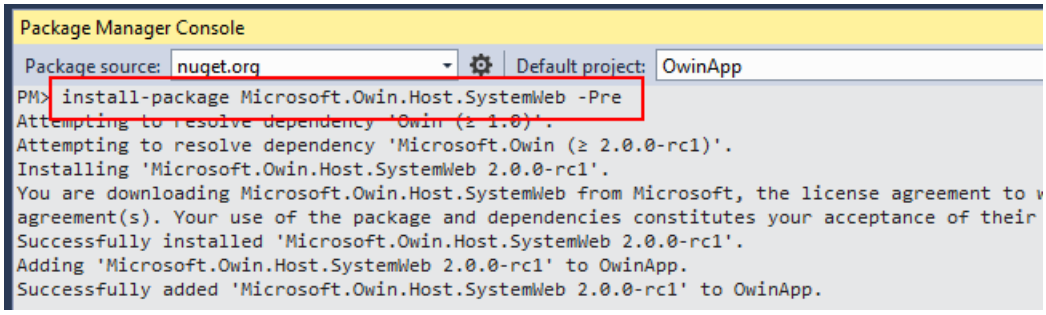
In the **New ASP.NET Project** dialog, select the **Empty** template.



Add NuGet Packages

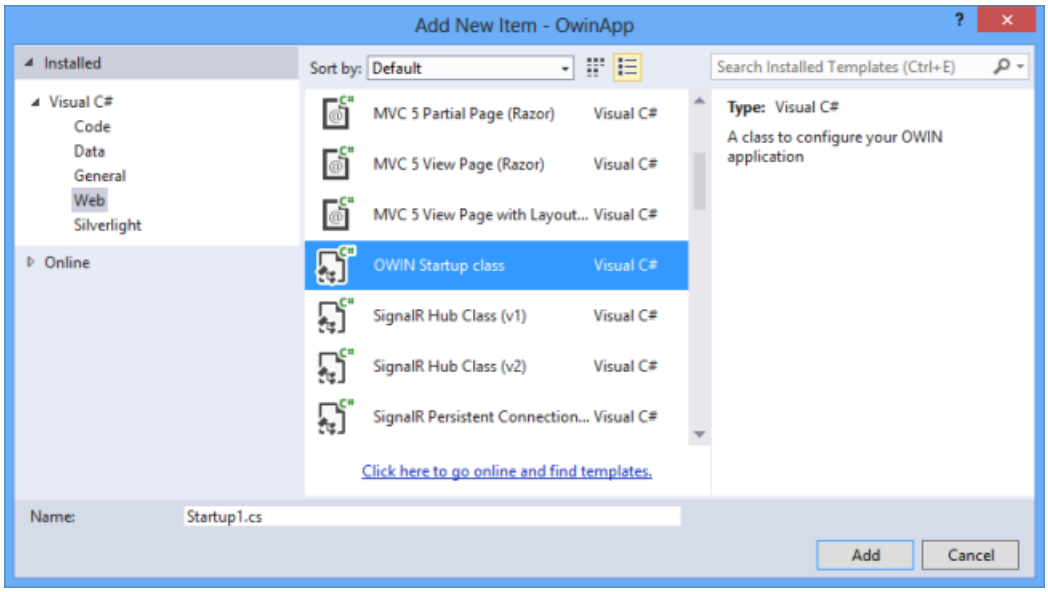
Next, add the required NuGet packages. From the **Tools** menu, select **NuGet Package Manager**, then select **Package Manager Console**. In the Package Manager Console window, type the following command:

```
install-package Microsoft.Owin.Host.SystemWeb -Pre
```



Add a Startup Class

Next, add an OWIN startup class. In Solution Explorer, right-click the project and select **Add**, then select **New Item**. In the **Add New Item** dialog, select **Owin Startup class**. For more info on configuring the startup class, see OWIN Startup Class Detection.



Add the following code to the `Startup1.Configuration` method:

```
public void Configuration(IAppBuilder app)
{
    // New code:
    app.Run(context =>
    {
        context.Response.ContentType = "text/plain";
        return context.Response.WriteAsync("Hello, world.");
    });
}
```

This code adds a simple piece of middleware to the OWIN pipeline, implemented as a function that receives a **Microsoft.Owin.IOwinContext** instance. When the server receives an HTTP request, the OWIN pipeline invokes the middleware. The middleware sets the content type for the response and writes the response body.

> **NOTE**
>
> The OWIN Startup class template is available in Visual Studio 2013. If you are using Visual Studio 2012, just add a new empty class named `Startup1`, and paste in the following code:

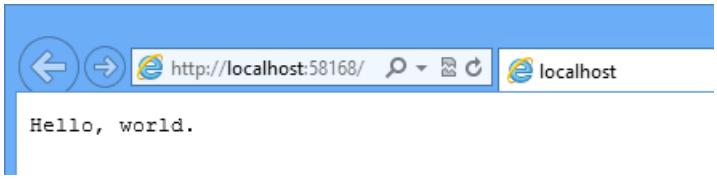
```
using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(OwinApp.Startup1))]

namespace OwinApp
{
    public class Startup1
    {
        public void Configuration(IAppBuilder app)
        {
          app.Run(context =>
          {
              context.Response.ContentType = "text/plain";
              return context.Response.WriteAsync("Hello, world.");
          });
        }
    }
}
```

Run the Application

Press F5 to begin debugging. Visual Studio will open a browser window to `http://localhost:*port*/`. The page should look like the following:



Self-Host OWIN in a Console Application

It's easy to convert this application from IIS hosting to self-hosting in a custom process. With IIS hosting, IIS acts as both the HTTP server and as the process that hosts the service. With self-hosting, your application creates the process and uses the **HttpListener** class as the HTTP server.

In Visual Studio, create a new console application. In the Package Manager Console window, type the following command:

```
Install-Package Microsoft.Owin.SelfHost -Pre
```

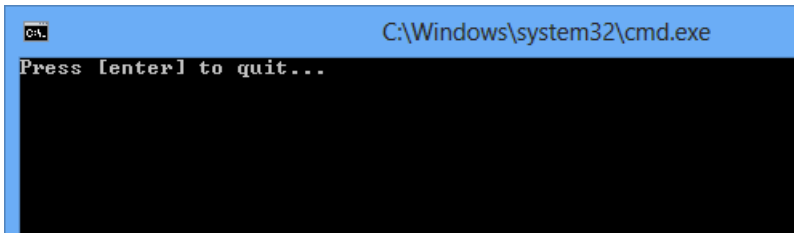
Add a `Startup1` class from part 1 of this tutorial to the project. You don't need to modify this class.

Implement the application's `Main` method as follows.

```
class Program
{
    static void Main(string[] args)
    {
        using (Microsoft.Owin.Hosting.WebApp.Start<Startup1>("http://localhost:9000"))
        {
            Console.WriteLine("Press [enter] to quit...");
            Console.ReadLine();
        }
    }
}
```

When you run the console application, the server starts listening to `http://localhost:9000`. If you navigate to this

address in a web browser, you will see the "Hello world" page.



Add OWIN Diagnostics

The Microsoft.Owin.Diagnostics package contains middleware that catches unhandled exceptions and displays an HTML page with error details. This page functions much like the ASP.NET error page that is sometimes called the "yellow screen of death" (YSOD). Like the YSOD, the Katana error page is useful during development, but it's a good practice to disable it in production mode.

To install the Diagnostics package in your project, type the following command in the Package Manager Console window:

```
install-package Microsoft.Owin.Diagnostics -Pre
```

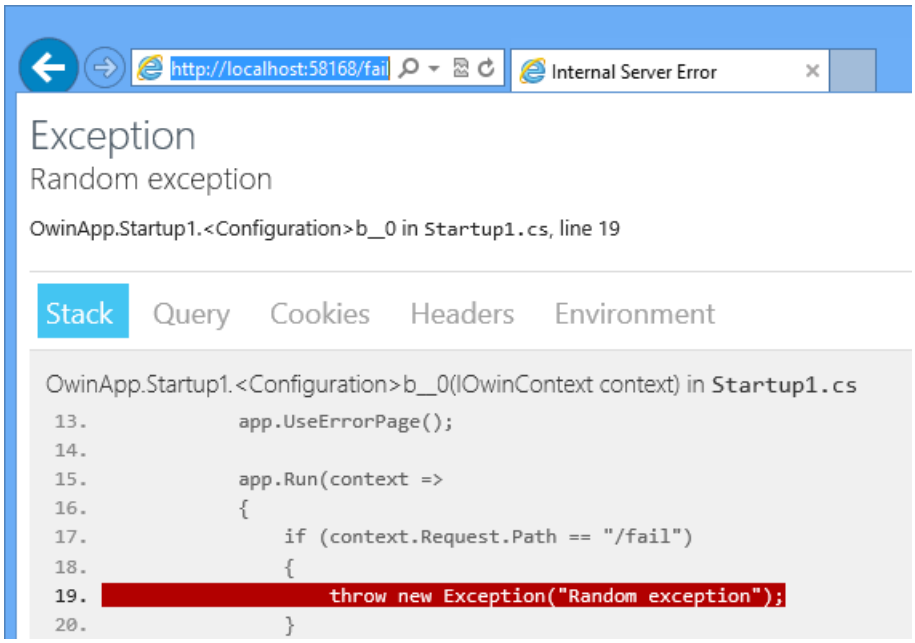
Change the code in your `Startup1.Configuration` method as follows:

```
public void Configuration(IAppBuilder app)
{
    // New code: Add the error page middleware to the pipeline.
    app.UseErrorPage();

    app.Run(context =>
    {
        // New code: Throw an exception for this URI path.
        if (context.Request.Path.Equals(new PathString("/fail")))
        {
            throw new Exception("Random exception");
        }

        context.Response.ContentType = "text/plain";
        return context.Response.WriteAsync("Hello, world.");
    });
}
```

Now use CTRL+F5 to run the application without debugging, so that Visual Studio will not break on the exception. The application behaves the same as before, until you navigate to `http://localhost/fail`, at which point the application throws the exception. The error page middleware will catch the exception and display an HTML page with information about the error. You can click the tabs to see the stack, query string, cookies, request header, and OWIN environment variables.



Next Steps

- OWIN Startup Class Detection
- Use OWIN to Self-Host ASP.NET Web API
- Use OWIN to Self-Host SignalR

SameSite cookies and the Open Web Interface for .NET (OWIN)

2/20/2020 • 6 minutes to read • Edit Online

By Rick Anderson

`SameSite` is an IETF draft designed to provide some protection against cross-site request forgery (CSRF) attacks. The SameSite 2019 draft:

- Treats cookies as `SameSite=Lax` by default.
- States cookies that explicitly assert `SameSite=None` in order to enable cross-site delivery should be marked as `Secure`.

`Lax` works for most app cookies. Some forms of authentication like OpenID Connect (OIDC) and WS-Federation default to POST based redirects. The POST based redirects trigger the `SameSite` browser protections, so `SameSite` is disabled for these components. Most OAuth logins aren't affected due to differences in how the request flows. All other components do **not** set `SameSite` by default and use the clients default behavior (old or new).

The `None` parameter causes compatibility problems with clients that implemented the prior 2016 draft standard (for example, iOS 12). See Supporting older browsers in this document.

Each OWIN component that emits cookies needs to decide if `SameSite` is appropriate.

For the ASP.NET 4.x version of this article, see Work with SameSite cookies in ASP.NET.

API usage with SameSite

`Microsoft.Owin` has its own `SameSite` implementation:

- That is not directly dependent on the one in `System.Web`.
- `SameSite` works on all versions targetable by the `Microsoft.Owin` packages, .NET 4.5 and later.
- Only the SystemWebCookieManager component directly interacts with the `System.Web` `HttpCookie` class.

`SystemWebCookieManager` depends on the .NET 4.7.2 `System.Web` APIs to enable `SameSite` support, and the patches to change the behavior.

The reasons to use `SystemWebCookieManager` are outlined in OWIN and System.Web response cookie integration issues. `SystemWebCookieManager` is recommended when running on `System.Web`.

The following code sets `SameSite` to `Lax`:

```
owinContext.Response.Cookies.Append("My Key", "My Value", new CookieOptions()
{
    SameSite = SameSiteMode.Lax
});
```

The following APIs use `SameSite`:

- Microsoft.Owin.SameSiteMode
- CookieOptions.SameSite
- CookieAuthenticationOptions Class

- CookieAuthenticationOptions.CookieSameSite
- ICookieManager
- SystemWebCookieManager
- SystemWebChunkingCookieManager
- CookieAuthenticationOptions.CookieManager
- OpenIdConnectAuthenticationOptions.CookieManager

History and changes

Microsoft.Owin never supported the `SameSite` 2016 draft standard.

Support for the SameSite 2019 draft is only available in `Microsoft.Owin` 4.1.0 and later. There are no patches for prior versions.

The 2019 draft of the `SameSite` specification:

- Is **not** backwards compatible with the 2016 draft. For more information, see Supporting older browsers in this document.
- Specifies cookies are treated as `SameSite=Lax` by default.
- Specifies cookies that explicitly assert `SameSite=None` in order to enable cross-site delivery should be marked as `Secure`. `None` is a new entry to opt out.
- Is scheduled to be enabled by Chrome by default in Feb 2020. Browsers started moving to this standard in 2019.
- Is supported by patches issued as described in KB articles. For more information, see KB articles that support SameSite in .NET Framework.

Supporting older browsers

The 2016 `SameSite` standard mandated that unknown values must be treated as `SameSite=Strict` values. Apps accessed from older browsers which support the 2016 `SameSite` standard may break when they get a `SameSite` property with a value of `None`. Web apps must implement browser detection if they intend to support older browsers. ASP.NET doesn't implement browser detection because User-Agents values are highly volatile and change frequently. An extension point in ICookieManager allows plugging in User-Agent specific logic.

In `Startup.Configuration`, add code similar to the following:

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        app.UseOpenIdConnectAuthentication(
             new OpenIdConnectAuthenticationOptions
             {
                 // … Your preexisting options …
                 CookieManager = new SameSiteCookieManager(
                                     new SystemWebCookieManager())
             });

        // Remaining code removed for brevity.
```

The preceding code requires the .NET 4.7.2 or later `SameSite` patch.

The following code shows an example implementation of `SameSiteCookieManager`:

```
public class SameSiteCookieManager : ICookieManager
{
    private readonly ICookieManager _innerManager;

    public SameSiteCookieManager() : this(new CookieManager())
    {
    }

    public SameSiteCookieManager(ICookieManager innerManager)
    {
        _innerManager = innerManager;
    }

    public void AppendResponseCookie(IOwinContext context, string key, string value,
                                     CookieOptions options)
    {
        CheckSameSite(context, options);
        _innerManager.AppendResponseCookie(context, key, value, options);
    }

    public void DeleteCookie(IOwinContext context, string key, CookieOptions options)
    {
        CheckSameSite(context, options);
        _innerManager.DeleteCookie(context, key, options);
    }

    public string GetRequestCookie(IOwinContext context, string key)
    {
        return _innerManager.GetRequestCookie(context, key);
    }

    private void CheckSameSite(IOwinContext context, CookieOptions options)
    {
        if (options.SameSite == Microsoft.Owin.SameSiteMode.None
                            && DisallowsSameSiteNone(context))
        {
            options.SameSite = null;
        }
    }
```

In the preceding sample, `DisallowsSameSiteNone` is called in the `CheckSameSite` method. `DisallowsSameSiteNone` is a user method that detects if the user agent doesn't support `SameSite` `None`:

```
private void CheckSameSite(IOwinContext context, CookieOptions options)
{
    if (options.SameSite == Microsoft.Owin.SameSiteMode.None
                        && DisallowsSameSiteNone(context))
    {
        options.SameSite = null;
    }
}
```

The following code shows a sample `DisallowsSameSiteNone` method:

WARNING

The following code is for demonstration only:

- It should not be considered complete.
- It is not maintained or supported.

```
public static bool DisallowsSameSiteNone(IOwinContext context)
{
    var userAgent = context.Request.Headers["User-Agent"];

    if (string.IsNullOrEmpty(userAgent))
    {
        return false;
    }

    // Cover all iOS based browsers here. This includes:
    // - Safari on iOS 12 for iPhone, iPod Touch, iPad
    // - WkWebview on iOS 12 for iPhone, iPod Touch, iPad
    // - Chrome on iOS 12 for iPhone, iPod Touch, iPad
    // All of which are broken by SameSite=None, because they use the iOS
    // networking stack.
    if (userAgent.Contains("CPU iPhone OS 12") ||
        userAgent.Contains("iPad; CPU OS 12"))
    {
        return true;
    }

    // Cover Mac OS X based browsers that use the Mac OS networking stack.
    // This includes:
    // - Safari on Mac OS X.
    // This does not include:
    // - Chrome on Mac OS X
    // Because they do not use the Mac OS networking stack.
    if (userAgent.Contains("Macintosh; Intel Mac OS X 10_14") &&
        userAgent.Contains("Version/") && userAgent.Contains("Safari"))
    {
        return true;
    }

    // Cover Chrome 50-69, because some versions are broken by SameSite=None,
    // and none in this range require it.
    // Note: this covers some pre-Chromium Edge versions,
    // but pre-Chromium Edge does not require SameSite=None.
    if (userAgent.Contains("Chrome/5") || userAgent.Contains("Chrome/6"))
    {
        return true;
    }

    return false;
}
```

Test apps for SameSite problems

Apps that interact with remote sites such as through third-party login need to:

- Test the interaction on multiple browsers.
- Apply the browser detection and mitigation discussed in this document.

Test web apps using a client version that can opt-in to the new `SameSite` behavior. Chrome, Firefox, and Chromium Edge all have new opt-in feature flags that can be used for testing. After your app applies the `SameSite` patches, test it with older client versions, especially Safari. For more information, see Supporting older browsers in this document.

Test with Chrome

Chrome 78+ gives misleading results because it has a temporary mitigation in place. The Chrome 78+ temporary mitigation allows cookies less than two minutes old. Chrome 76 or 77 with the appropriate test flags enabled provides more accurate results. To test the new `SameSite` behavior toggle `chrome://flags/#same-site-by-default-cookies` to **Enabled**. Older versions of Chrome (75 and below) are reported

to fail with the new `None` setting. See Supporting older browsers in this document.

Google does not make older chrome versions available. Follow the instructions at Download Chromium to test older versions of Chrome. Do **not** download Chrome from links provided by searching for older versions of chrome.

- Chromium 76 Win64
- Chromium 74 Win64

Test with Safari

Safari 12 strictly implemented the prior draft and fails when the new `None` value is in a cookie. `None` is avoided via the browser detection code Supporting older browsers in this document. Test Safari 12, Safari 13, and WebKit based OS style logins using MSAL, ADAL or whatever library you are using. The problem is dependent on the underlying OS version. OSX Mojave (10.14) and iOS 12 are known to have compatibility problems with the new `SameSite` behavior. Upgrading the OS to OSX Catalina (10.15) or iOS 13 fixes the problem. Safari does not currently have an opt-in flag for testing the new spec behavior.

Test with Firefox

Firefox support for the new standard can be tested on version 68+ by opting in on the `about:config` page with the feature flag `network.cookie.sameSite.laxByDefault`. There haven't been reports of compatibility issues with older versions of Firefox.

Test with Edge browser

Edge supports the old `SameSite` standard. Edge version 44 doesn't have any known compatibility problems with the new standard.

Test with Edge (Chromium)

`SameSite` flags are set on the `edge://flags/#same-site-by-default-cookies` page. No compatibility issues were discovered with Edge Chromium.

Test with Electron

Versions of Electron include older versions of Chromium. For example, the version of Electron used by Teams is Chromium 66, which exhibits the older behavior. You must perform your own compatibility testing with the version of Electron your product uses. See Supporting older browsers in the following section.

Additional resources

- Chromium Blog:Developers: Get Ready for New SameSite=None; Secure Cookie Settings
- SameSite cookies explained
- OWIN and System.Web response cookie integration issues

Katana Samples

4/10/2020 • 2 minutes to read • Edit Online

by Microsoft

Katana Samples

ASP.NET Routes Sample | Source Code
In some applications you will want to hook up OWIN components in the Asp.Net route table side by side with non-OWIN components. This sample shows how to use the RouteCollection extension methods MapOwinPath and MapOwinRoute provided by Microsoft.Owin.Host.SystemWeb.

Branching Pipelines Sample | Source Code
OWIN request processing pipelines do not need to be linear, they can be branched to process requests in different ways. This sample shows how to construct a branching pipeline based on request paths or other request data such as headers. These components are available in the Microsoft.Owin.Mapping nuget package.

Custom Server Sample | Source Code
Shows how to use a custom OWIN server when self-hosting OWIN.

Embedded Sample | Source Code
Some OWIN servers can be run inside of your own process ("self-hosted"). This sample shows how to start an OWIN application using the tools provided by the Microsoft.Owin.Hosting nuget package.

HelloWorld Sample | Source Code
OWIN is a HTTP server API abstraction that enables application portability across various servers. This sample demonstrates how to write a Hello World application using some **simple wrappers** around the raw OWIN abstraction and run it on a web server like ASP.NET.

Hello World Raw OWIN Sample | Source Code
This sample demonstrates how to write a Hello World application using the **raw** OWIN abstraction and run it on a web server like Asp.Net.

SignalR Sample | Source Code
Shows how to self-host SignalR using OWIN / Katana. For more info about self-hosting SignalR, see Tutorial: SignalR Self-Host.

Static Files Sample | Source Code
Shows how to support HTTP requests for static files using OWIN / Katana.

Web API | Source Code
This sample shows how to host OWIN in IIS and add Web API to the OWIN pipeline.

Web Socket Sample | Source Code
Shows how to support Web Sockets in OWIN by using the System.Net.WebSockets.WebSocket class.

OWIN Startup Class Detection

2/20/2020 • 6 minutes to read • Edit Online

This tutorial shows how to configure which OWIN startup class is loaded. For more information on OWIN, see An Overview of Project Katana. This tutorial was written by Rick Anderson (@RickAndMSFT), Praburaj Thiagarajan, and Howard Dierking (@howard_dierking).

Prerequisites

Visual Studio 2017

OWIN Startup Class Detection

Every OWIN Application has a startup class where you specify components for the application pipeline. There are different ways you can connect your startup class with the runtime, depending on the hosting model you choose (OwinHost, IIS, and IIS-Express). The startup class shown in this tutorial can be used in every hosting application. You connect the startup class with the hosting runtime using one of the these approaches:

1. **Naming Convention**: Katana looks for a class named `Startup` in namespace matching the assembly name or the global namespace.

2. **OwinStartup Attribute**: This is the approach most developers will take to specify the startup class. The following attribute will set the startup class to the `TestStartup` class in the `StartupDemo` namespace.

   ```
   [assembly: OwinStartup(typeof(StartupDemo.TestStartup))]
   ```

   The `OwinStartup` attribute overrides the naming convention. You can also specify a friendly name with this attribute, however, using a friendly name requires you to also use the `appSetting` element in the configuration file.

3. **The appSetting element in the Configuration file**: The `appSetting` element overrides the `OwinStartup` attribute and naming convention. You can have multiple startup classes (each using an `OwinStartup` attribute) and configure which startup class will be loaded in a configuration file using markup similar to the following:

   ```
   <appSettings>
     <add key="owin:appStartup" value="StartupDemo.ProductionStartup" />
   </appSettings>
   ```

   The following key, which explicitly specifies the startup class and assembly can also be used:

   ```
   <add key="owin:appStartup" value="StartupDemo.ProductionStartup, StartupDemo" />
   ```

   The following XML in the configuration file specifies a friendly startup class name of `ProductionConfiguration`.

```
<appSettings>
  <add key="owin:appStartup" value="ProductionConfiguration" />
</appSettings>
```

The above markup must be used with the following `OwinStartup` attribute which specifies a friendly name and causes the `ProductionStartup2` class to run.

```
[assembly: OwinStartup("ProductionConfiguration", typeof(StartupDemo.ProductionStartup2))]

namespace StartupDemo
{
    public class ProductionStartup
    {
        public void Configuration(IAppBuilder app)
        {
            app.Run(context =>
            {
                string t = DateTime.Now.Millisecond.ToString();
                return context.Response.WriteAsync(t + " Production OWIN App");
            });
        }
    }
    public class ProductionStartup2
    {
        public void Configuration(IAppBuilder app)
        {
            app.Run(context =>
            {
                string t = DateTime.Now.Millisecond.ToString();
                return context.Response.WriteAsync(t + " 2nd Production OWIN App");
            });
        }
    }
}
```

4. To disable OWIN startup discovery add the `appSetting owin:AutomaticAppStartup` with a value of `"false"` in the web.config file.

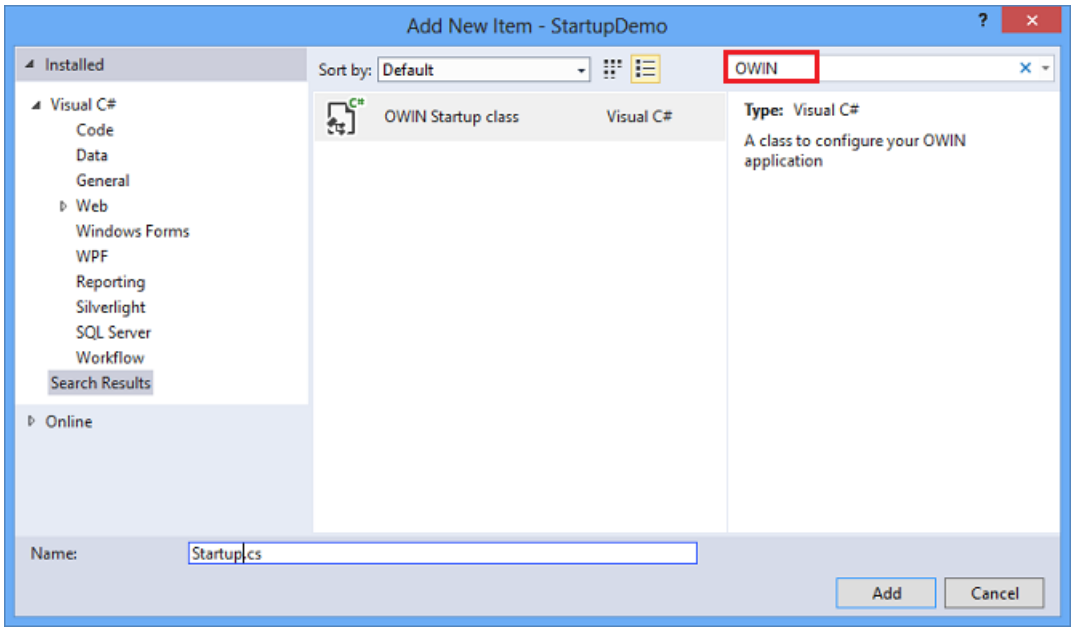
```
<add key="owin:AutomaticAppStartup" value="false" />
```

Create an ASP.NET Web App using OWIN Startup

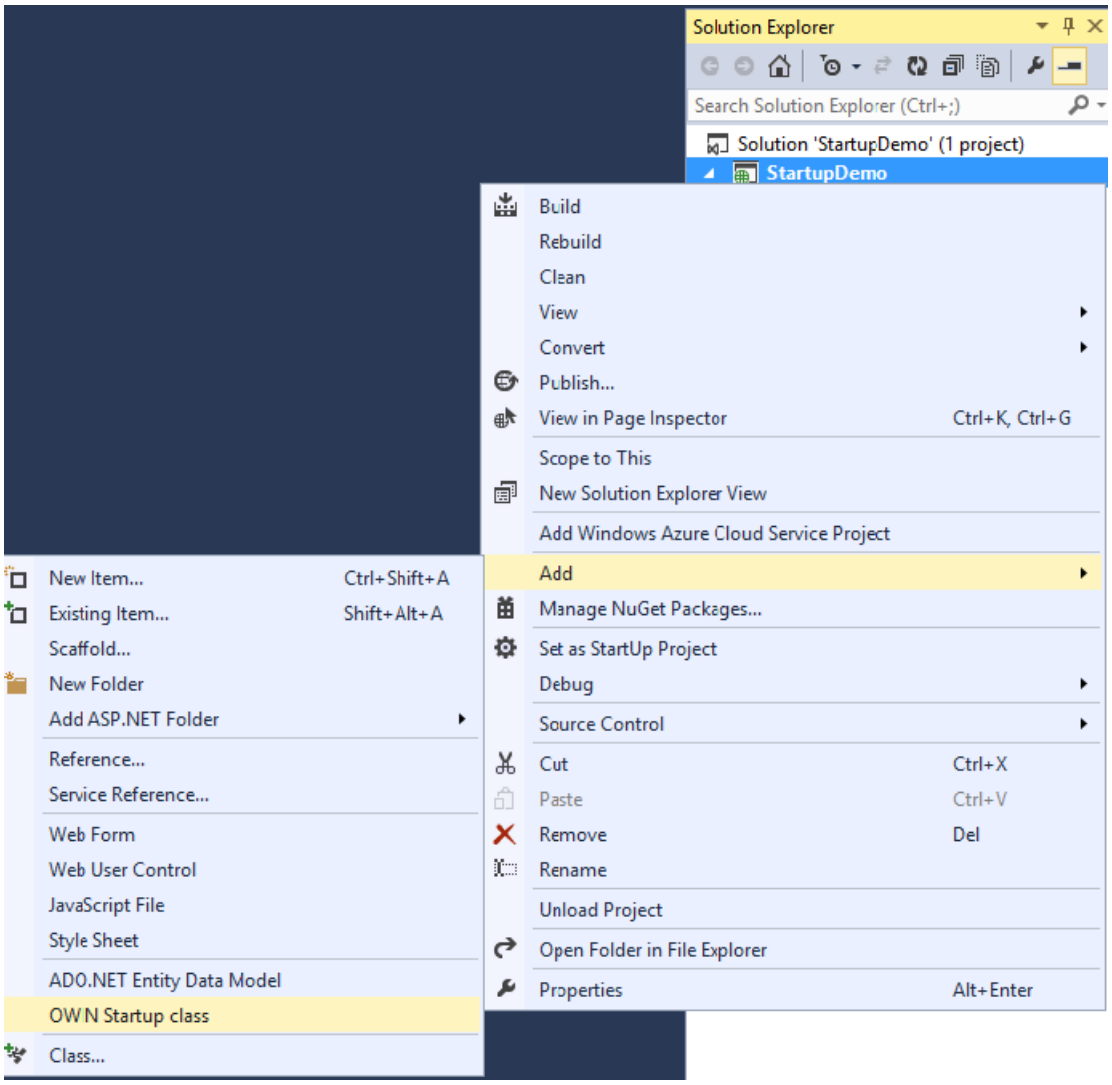
1. Create an empty Asp.Net web application and name it **StartupDemo**. - Install `Microsoft.Owin.Host.SystemWeb` using the NuGet package manager. From the **Tools** menu, select **NuGet Package Manager**, and then **Package Manager Console**. Enter the following command:

```
Install-Package Microsoft.Owin.Host.SystemWeb
```

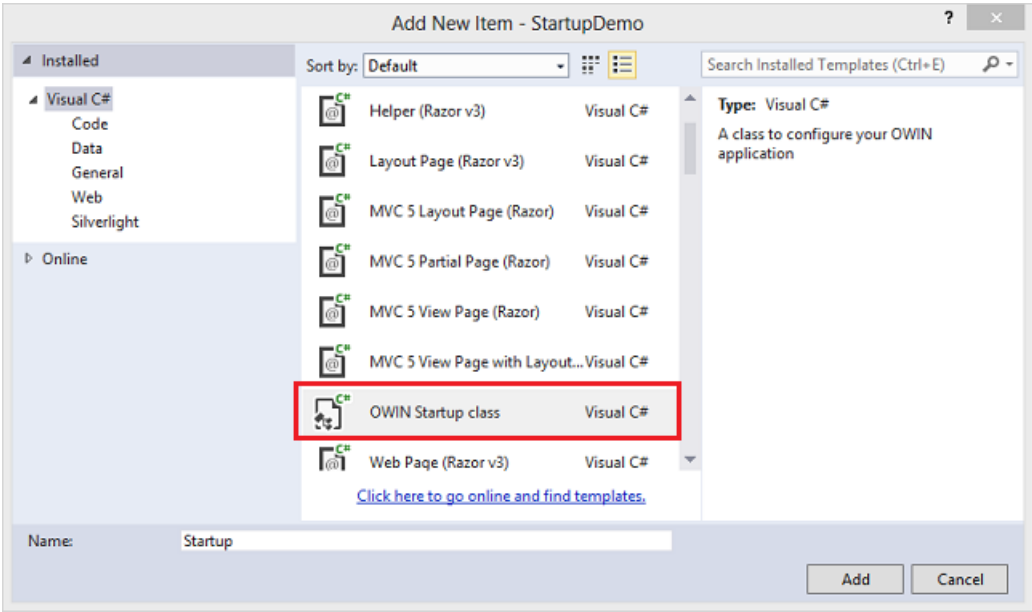
2. Add an OWIN startup class. In Visual Studio 2017 right-click the project and select **Add Class**.- In the **Add New Item** dialog box, enter *OWIN* in the search field, and change the name to Startup.cs, and then select **Add**.



The next time you want to add an *Owin Startup class*, it will be in available from the **Add** menu.



Alternatively, you can right-click the project and select **Add**, then select **New Item**, and then select the **Owin Startup class**.



- Replace the generated code in the *Startup.cs* file with the following:

```
using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;
using System.IO;

//[assembly: OwinStartup(typeof(StartupDemo.Startup))]

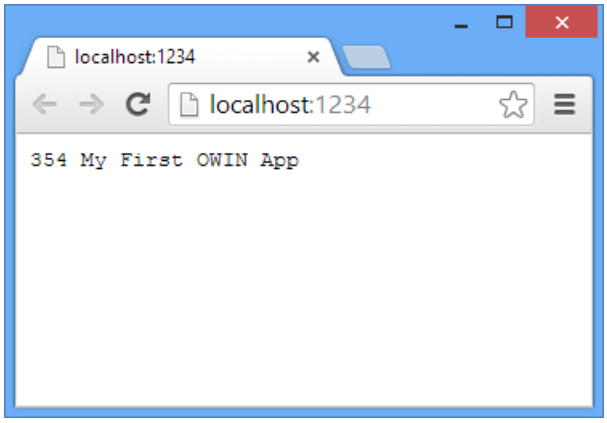
namespace StartupDemo
{
   public class Startup
   {
      public void Configuration(IAppBuilder app)
      {
         app.Use((context, next) =>
         {
            TextWriter output = context.Get<TextWriter>("host.TraceOutput");
            return next().ContinueWith(result =>
            {
               output.WriteLine("Scheme {0} : Method {1} : Path {2} : MS {3}",
               context.Request.Scheme, context.Request.Method, context.Request.Path, getTime());
            });
         });

         app.Run(async context =>
         {
            await context.Response.WriteAsync(getTime() + " My First OWIN App");
         });
      }

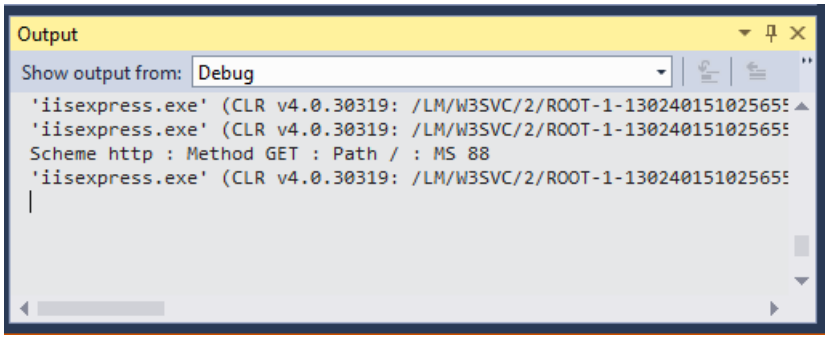
      string getTime()
      {
         return DateTime.Now.Millisecond.ToString();
      }
   }
}
```

The `app.Use` lambda expression is used to register the specified middleware component to the OWIN pipeline. In this case we are setting up logging of incoming requests before responding to the incoming request. The `next` parameter is the delegate (Func < Task >) to the next component in the pipeline. The `app.Run` lambda expression hooks up the pipeline to incoming requests and provides the response mechanism.

> **NOTE**
>
> In the code above we have commented out the `OwinStartup` attribute and we're relying on the convention of running the class named `Startup` .- Press ***F5*** to run the application. Hit refresh a few times.



Note: The number shown in the images in this tutorial will not match the number you see. The millisecond string is used to show a new response when you refresh the page. You can see the trace information in the **Output** window.



Add More Startup Classes

In this section we'll add another Startup class. You can add multiple OWIN startup class to your application. For example, you might want to create startup classes for development, testing and production.

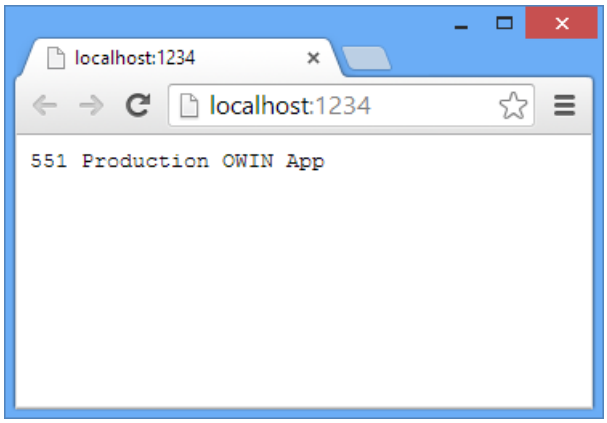
1. Create a new OWIN Startup class and name it `ProductionStartup`.
2. Replace the generated code with the following:

```
using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(StartupDemo.ProductionStartup))]

namespace StartupDemo
{
    public class ProductionStartup
    {
        public void Configuration(IAppBuilder app)
        {
            app.Run(context =>
            {
                string t = DateTime.Now.Millisecond.ToString();
                return context.Response.WriteAsync(t + " Production OWIN App");
            });
        }
    }
}
```

3. Press Control F5 to run the app. The `OwinStartup` attribute specifies the production startup class is run.



4. Create another OWIN Startup class and name it `TestStartup`.

5. Replace the generated code with the following:

```
using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup("TestingConfiguration", typeof(StartupDemo.TestStartup))]

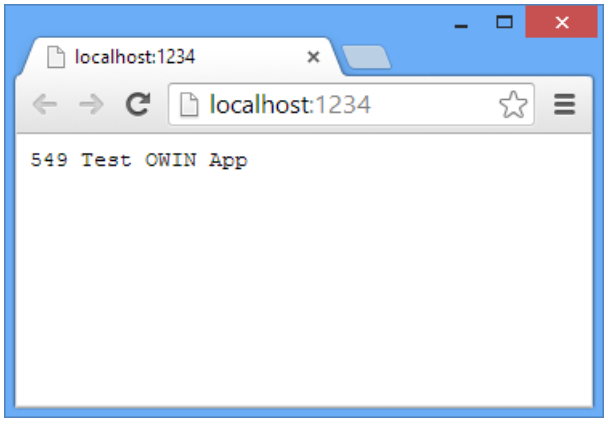
namespace StartupDemo
{
    public class TestStartup
    {
        public void Configuration(IAppBuilder app)
        {
            app.Run(context =>
            {
                string t = DateTime.Now.Millisecond.ToString();
                return context.Response.WriteAsync(t + " Test OWIN App");
            });
        }
    }
}
```

The `OwinStartup` attribute overload above specifies `TestingConfiguration` as the *friendly* name of the Startup class.

6. Open the *web.config* file and add the OWIN App startup key which specifies the friendly name of the Startup class:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="owin:appStartup" value="TestingConfiguration" />
  </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>
</configuration>
```

7. Press Control F5 to run the app. The app settings element takes precedent, and the test configuration is run.



8. Remove the *friendly* name from the `OwinStartup` attribute in the `TestStartup` class.

```
[assembly: OwinStartup(typeof(StartupDemo.TestStartup))]
```

9. Replace the OWIN App startup key in the *web.config* file with the following:

```
<add key="owin:appStartup" value="StartupDemo.TestStartup" />
```

10. Revert the `OwinStartup` attribute in each class to the default attribute code generated by Visual Studio:

```
[assembly: OwinStartup(typeof(StartupDemo.Startup))]
[assembly: OwinStartup(typeof(StartupDemo.ProductionStartup))]
[assembly: OwinStartup(typeof(StartupDemo.TestStartup))]
```

Each of the OWIN App startup keys below will cause the production class to run.

```
<add key="owin:appStartup" value="StartupDemo.ProductionStartup" />
<add key="owin:appStartup" value="StartupDemo.ProductionStartup, StartupDemo" />
<add key="owin:appStartup" value="StartupDemo.ProductionStartup.Configuration, StartupDemo" />
```

The last startup key specifies the startup configuration method. The following OWIN App startup key allows you to change the name of the configuration class to `MyConfiguration` .

```
<add key="owin:appStartup" value="StartupDemo.ProductionStartup2.MyConfiguration" />
```

Using Owinhost.exe

1. Replace the Web.config file with the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
   <appSettings>
      <add key="owin:appStartup" value="StartupDemo.Startup" />
      <add key="owin:appStartup" value="StartupDemo.TestStartup" />
   </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>
</configuration>
```

   The last key wins, so in this case `TestStartup` is specified.

2. Install Owinhost from the PMC:

```
Install-Package OwinHost
```

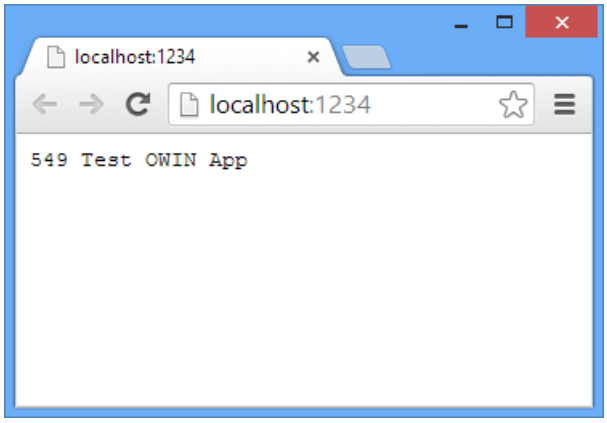
3. Navigate to the application folder (the folder containing the *Web.config* file) and in a command prompt and type:

```
..\packages\Owinhost<Version>\tools\Owinhost.exe
```

   The command window will show:

```
C:\StartupDemo\StartupDemo>..\packages\OwinHost.2.0.0\tools\Owin
Host.exe
Starting with the default port: 5000 (http://localhost:5000/)
Started successfully
Press Enter to exit
```

4. Launch a browser with the URL `http://localhost:5000/`.



   OwinHost honored the startup conventions listed above.

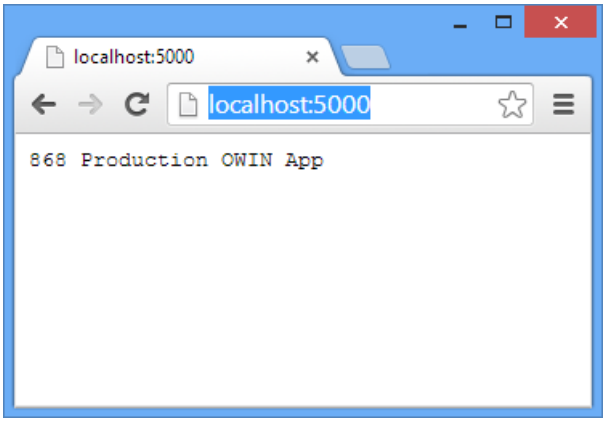
5. In the command window, press Enter to exit OwinHost.

6. In the `ProductionStartup` class, add the following OwinStartup attribute which specifies a friendly name of *ProductionConfiguration*.

```
[assembly: OwinStartup("ProductionConfiguration",
           typeof(StartupDemo.ProductionStartup))]
```

7. In the command prompt and type:

```
..\packages\OwinHost.2.0.0\tools\OwinHost.exe ProductionConfiguration
```

The Production startup class is loaded. 

Our application has multiple startup classes, and in this example we have deferred which startup class to load until runtime.

8. Test the following runtime startup options:

```
..\packages\OwinHost.2.0.0-rc1\tools\OwinHost.exe StartupDemo.TestStartup
..\packages\OwinHost.2.0.0-rc1\tools\OwinHost.exe "StartupDemo.TestStartup,StartupDemo"
..\packages\OwinHost.2.0.0-rc1\tools\OwinHost.exe StartupDemo.TestStartup.Configuration
..\packages\OwinHost.2.0.0-rc1\tools\OwinHost.exe "StartupDemo.TestStartup.Configuration,StartupDemo"
```

Enabling Windows Authentication in Katana

2/20/2020 • 3 minutes to read • Edit Online

by Mike Wasson

This article shows how to enable Windows Authentication in Katana. It covers two scenarios: Using IIS to host Katana, and using HttpListener to self-host Katana in a custom process. Thanks to Barry Dorrans, David Matson, and Chris Ross for reviewing this article.

Katana is Microsoft's implementation of OWIN, the Open Web Interface for .NET. You can read an introduction to OWIN and Katana here. The OWIN architecture has several layers:

- Host: Manages the process in which the OWIN pipeline runs.
- Server: Opens a network socket and listens for requests.
- Middleware: Processes the HTTP request and response.

Katana currently provides two servers, both of which support Windows Integrated Authentication:

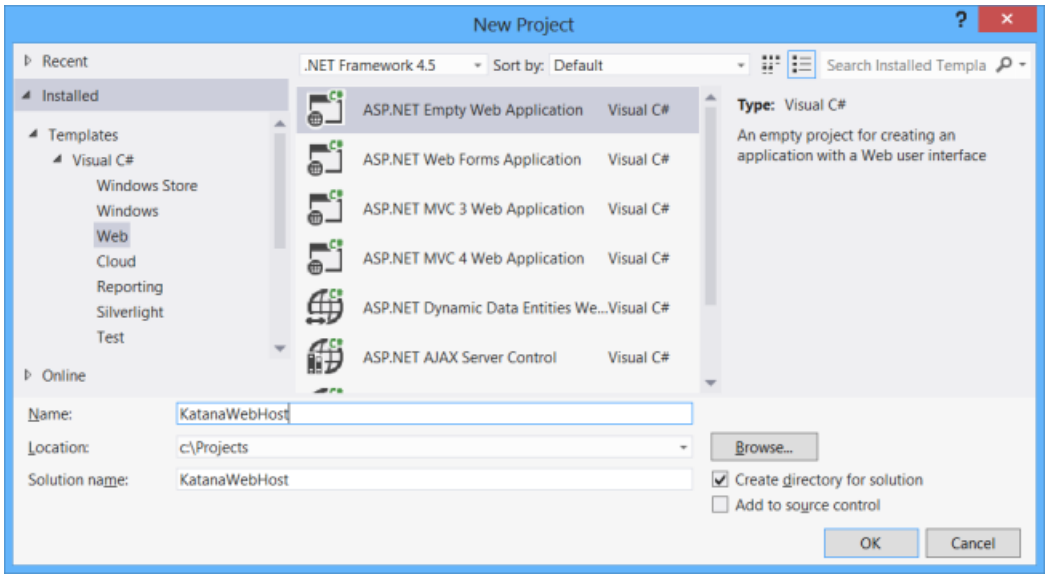
- **Microsoft.Owin.Host.SystemWeb**. Uses IIS with the ASP.NET pipeline.
- **Microsoft.Owin.Host.HttpListener**. Uses System.Net.HttpListener. This server is currently the default option when self-hosting Katana.

> **NOTE**
>
> Katana does not currently provide OWIN middleware for Windows Authentication, because this functionality is already available in the servers.

Windows Authentication in IIS

Using Microsoft.Owin.Host.SystemWeb, you can simply enable Windows Authentication in IIS.

Let's start by creating a new ASP.NET application, using the "ASP.NET Empty Web Application" project template.



Next, add NuGet packages. From the **Tools** menu, select **NuGet Package Manager**, then select **Package Manager Console**. In the Package Manager Console window, enter the following command:

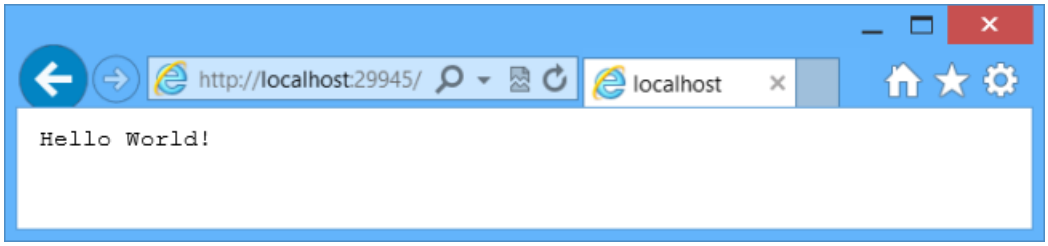
```
Install-Package Microsoft.Owin.Host.SystemWeb -pre
```

Now add a class named `Startup` with the following code:

```
using Owin;

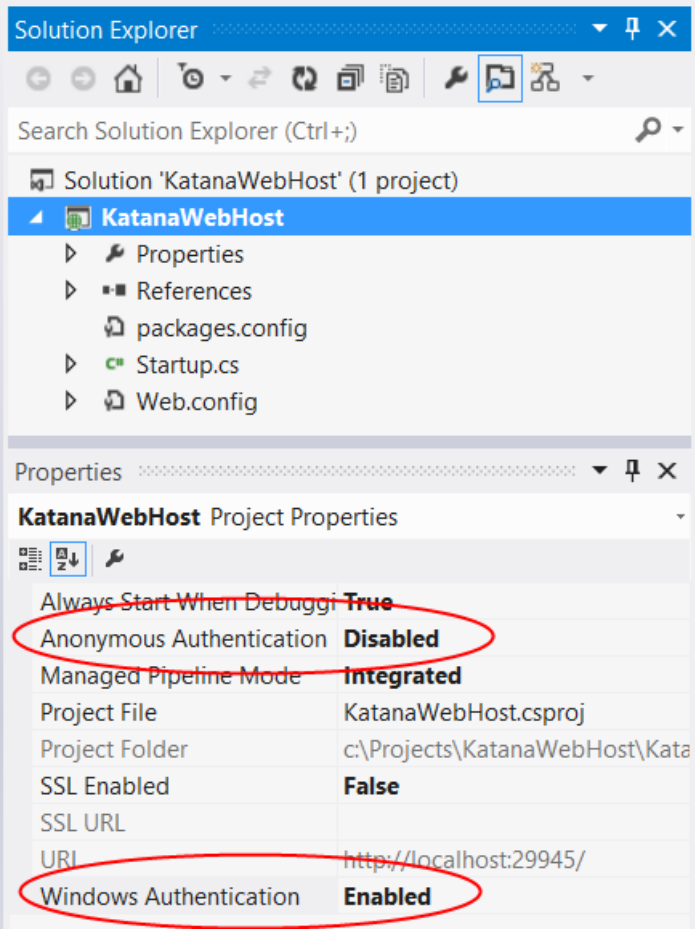
namespace KatanaWebHost
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.Run(context =>
            {
                context.Response.ContentType = "text/plain";
                return context.Response.WriteAsync("Hello World!");
            });
        }
    }
}
```

That's all you need to create a "Hello world" application for OWIN, running on IIS. Press F5 to debug the application. You should see "Hello World!" in the browser window.



Next, we'll enable Windows Authentication in IIS Express. From the **View** menu, select **Properties**. Click on the project name in Solution Explorer to view the project properties.

In the **Properties** window, set **Anonymous Authentication** to **Disabled** and set **Windows Authentication** to **Enabled**.



When you run the application from Visual Studio, IIS Express will require the user's Windows credentials. You can see this by using Fiddler or another HTTP debugging tool. Here is an example HTTP response:

```
HTTP/1.1 401 Unauthorized
Cache-Control: private
Content-Type: text/html; charset=utf-8
Server: Microsoft-IIS/8.0
WWW-Authenticate: Negotiate
WWW-Authenticate: NTLM
X-Powered-By: ASP.NET
Date: Sun, 28 Jul 2013 07:28:51 GMT
Content-Length: 6062
Proxy-Support: Session-Based-Authentication
```

The WWW-Authenticate headers in this response indicate that the server supports the Negotiate protocol, which uses either Kerberos or NTLM.

Later, when you deploy the application to a server, follow these steps to enable Windows Authentication in IIS on that server.

Windows Authentication in HttpListener

If you are using Microsoft.Owin.Host.HttpListener to self-host Katana, you can enable Windows Authentication directly on the **HttpListener** instance.

First, create a new console application. Next, add NuGet packages. From the **Tools** menu, select **NuGet Package Manager**, then select **Package Manager Console**. In the Package Manager Console window, enter the following command:

```
Install-Package Microsoft.Owin.SelfHost -Pre
```

Now add a class named `Startup` with the following code:

```
using Owin;
using System.Net;

namespace KatanaSelfHost
{
    class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            HttpListener listener =
                (HttpListener)app.Properties["System.Net.HttpListener"];
            listener.AuthenticationSchemes =
                AuthenticationSchemes.IntegratedWindowsAuthentication;

            app.Run(context =>
            {
                context.Response.ContentType = "text/plain";
                return context.Response.WriteAsync("Hello World!");
            });
        }
    }
}
```

This class implements the same "Hello world" example from before, but it also sets Windows Authentication as the authentication scheme.

Inside the `Main` function, start the OWIN pipeline:

```
using Microsoft.Owin.Hosting;
using System;

namespace KatanaSelfHost
{
    class Program
    {
        static void Main(string[] args)
        {
            using (WebApp.Start<Startup>("http://localhost:9000"))
            {
                Console.WriteLine("Press Enter to quit.");
                Console.ReadKey();
            }
        }
    }
}
```

You can send a request in Fiddler to confirm that the application is using Windows Authentication:

```
HTTP/1.1 401 Unauthorized
Content-Length: 0
Server: Microsoft-HTTPAPI/2.0
WWW-Authenticate: Negotiate
WWW-Authenticate: NTLM
Date: Sun, 28 Jul 2013 21:02:21 GMT
Proxy-Support: Session-Based-Authentication
```

Related Topics

An Overview of Project Katana

System.Net.HttpListener

Understanding OWIN Forms Authentication in MVC 5

OWIN Middleware in the IIS integrated pipeline

2/20/2020 • 5 minutes to read • Edit Online

by Praburaj Thiagarajan, Rick Anderson

> This article shows how to run OWIN middleware Components (OMCs) in the IIS integrated pipeline, and how to set the pipeline event an OMC runs on. You should review An Overview of Project Katana and OWIN Startup Class Detection before reading this tutorial. This tutorial was written by Rick Anderson (@RickAndMSFT), Chris Ross, Praburaj Thiagarajan, and Howard Dierking (@howard_dierking).

Although OWIN middleware components (OMCs) are primarily designed to run in a server-agnostic pipeline, it is possible to run an OMC in the IIS integrated pipeline as well (**classic mode is *not* supported**). An OMC can be made to work in the IIS integrated pipeline by installing the following package from the Package Manager Console (PMC):

```
Install-Package Microsoft.Owin.Host.SystemWeb
```

This means that all application frameworks, even those that are not yet able to run outside of IIS and System.Web, can benefit from existing OWIN middleware components.

> **NOTE**
>
> All of the `Microsoft.Owin.Security.*` packages shipping with the new Identity System in Visual Studio 2013 (for example: Cookies, Microsoft Account, Google, Facebook, Twitter, Bearer Token, OAuth, Authorization server, JWT, Azure Active directory, and Active directory federation services) are authored as OMCs, and can be used in both self-hosted and IIS-hosted scenarios.

How OWIN Middleware Executes in the IIS Integrated Pipeline

For OWIN console applications, the application pipeline built using the startup configuration is set by the order the components are added using the `IAppBuilder.Use` method. That is, the OWIN pipeline in the Katana runtime will process OMCs in the order they were registered using `IAppBuilder.Use`. In the IIS integrated pipeline the request pipeline consists of HttpModules subscribed to a pre-defined set of the pipeline events such as BeginRequest, AuthenticateRequest, AuthorizeRequest, etc.

If we compare an OMC to that of an HttpModule in the ASP.NET world, an OMC must be registered to the correct pre-defined pipeline event. For example, the HttpModule `MyModule` will get invoked when a request comes to the AuthenticateRequest stage in the pipeline:

```
public class MyModule : IHttpModule
{
    public void Dispose()
    {
        //clean-up code here.
    }
    public void Init(HttpApplication context)
    {
        // An example of how you can handle AuthenticateRequest events.
        context.AuthenticateRequest += ctx_AuthRequest;
    }
    void ctx_AuthRequest(object sender, EventArgs e)
    {
        // Handle event.
    }
}
```

In order for an OMC to participate in this same, event-based execution ordering, the Katana runtime code scans through the startup configuration and subscribes each of the middleware components to an integrated pipeline event. For example, the following OMC and registration code enables you to see the default event registration of middleware components. (For more detailed instructions on creating an OWIN startup class, see OWIN Startup Class Detection.)

1. Create an empty web application project and name it **owin2**.
2. From the Package Manager Console (PMC), run the following command:

   ```
   Install-Package Microsoft.Owin.Host.SystemWeb
   ```

3. Add an `OWIN Startup Class` and name it `Startup`. Replace the generated code with the following (the changes are highlighted):

```
using System;
using System.Threading.Tasks;
using Microsoft.Owin;
using Owin;
using System.Web;
using System.IO;
using Microsoft.Owin.Extensions;
[assembly: OwinStartup(typeof(owin2.Startup))]
namespace owin2
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.Use((context, next) =>
            {
                PrintCurrentIntegratedPipelineStage(context, "Middleware 1");
                return next.Invoke();
            });
            app.Use((context, next) =>
            {
                PrintCurrentIntegratedPipelineStage(context, "2nd MW");
                return next.Invoke();
            });
            app.Run(context =>
            {
                PrintCurrentIntegratedPipelineStage(context, "3rd MW");
                return context.Response.WriteAsync("Hello world");
            });
        }
        private void PrintCurrentIntegratedPipelineStage(IOwinContext context, string msg)
        {
            var currentIntegratedpipelineStage = HttpContext.Current.CurrentNotification;
            context.Get<TextWriter>("host.TraceOutput").WriteLine(
                "Current IIS event: " + currentIntegratedpipelineStage
                + " Msg: " + msg);
        }
    }
}
```

4. Hit F5 to run the app.

The Startup configuration sets up a pipeline with three middleware components, the first two displaying diagnostic information and the last one responding to events (and also displaying diagnostic information). The `PrintCurrentIntegratedPipelineStage` method displays the integrated pipeline event this middleware is invoked on and a message. The output windows displays the following:

```
Current IIS event: PreExecuteRequestHandler Msg: Middleware 1
Current IIS event: PreExecuteRequestHandler Msg: 2nd MW
Current IIS event: PreExecuteRequestHandler Msg: 3rd MW
```

The Katana runtime mapped each of the OWIN middleware components to PreExecuteRequestHandler by default, which corresponds to the IIS pipeline event PreRequestHandlerExecute.

Stage Markers

You can mark OMCs to execute at specific stages of the pipeline by using the `IAppBuilder UseStageMarker()` extension method. To run a set of middleware components during a particular stage, insert a stage marker right after the last component is the set during registration. There are rules on which stage of the pipeline you can execute middleware and the order components must run (The rules are explained later in the tutorial). Add the `UseStageMarker` method to the `Configuration` code as shown below:

```
public void Configuration(IAppBuilder app)
{
    app.Use((context, next) =>
    {
        PrintCurrentIntegratedPipelineStage(context, "Middleware 1");
        return next.Invoke();
    });
    app.Use((context, next) =>
    {
        PrintCurrentIntegratedPipelineStage(context, "2nd MW");
        return next.Invoke();
    });
    app.UseStageMarker(PipelineStage.Authenticate);
    app.Run(context =>
    {
        PrintCurrentIntegratedPipelineStage(context, "3rd MW");
        return context.Response.WriteAsync("Hello world");
    });
    app.UseStageMarker(PipelineStage.ResolveCache);
}
```

The `app.UseStageMarker(PipelineStage.Authenticate)` call configures all the previously registered middleware components (in this case, our two diagnostic components) to run on the authentication stage of the pipeline. The last middleware component (which displays diagnostics and responds to requests) will run on the `ResolveCache` stage (the ResolveRequestCache event).

Hit F5 to run the app.The output window shows the following:

```
Current IIS event: AuthenticateRequest Msg: Middleware 1
Current IIS event: AuthenticateRequest Msg: 2nd MW
Current IIS event: ResolveRequestCache Msg: 3rd MW
```

Stage Marker Rules

Owin middleware components (OMC) can be configured to run at the following OWIN pipeline stage events:

```
public enum PipelineStage
{
    Authenticate = 0,
    PostAuthenticate = 1,
    Authorize = 2,
    PostAuthorize = 3,
    ResolveCache = 4,
    PostResolveCache = 5,
    MapHandler = 6,
    PostMapHandler = 7,
    AcquireState = 8,
    PostAcquireState = 9,
    PreHandlerExecute = 10,
}
```

1. By default, OMCs run at the last event (`PreHandlerExecute`). That's why our first example code displayed "PreExecuteRequestHandler".
2. You can use the a `app.UseStageMarker` method to register a OMC to run earlier, at any stage of the OWIN pipeline listed in the `PipelineStage` enum.
3. The OWIN pipeline and the IIS pipeline is ordered, therefore calls to `app.UseStageMarker` must be in order. You cannot set the event handler to an event that precedes the last event registered with to

`app.UseStageMarker`. For example, *after* calling:

```
app.UseStageMarker(PipelineStage.Authorize);
```

calls to `app.UseStageMarker` passing `Authenticate` or `PostAuthenticate` will not be honored, and no exception will be thrown. OMCs run at the latest stage, which by default is `PreHandlerExecute`. The stage markers are used to make them to run earlier. If you specify stage markers out of order, we round to the earlier marker. In other words, adding a stage marker says "Run no later than stage X". OMC's run at the earliest stage marker added after them in the OWIN pipeline.

4. The earliest stage of calls to `app.UseStageMarker` wins. For example, if you switch the order of `app.UseStageMarker` calls from our previous example:

```
public void Configuration(IAppBuilder app)
{
    app.Use((context, next) =>
    {
        PrintCurrentIntegratedPipelineStage(context, "Middleware 1");
        return next.Invoke();
    });
    app.Use((context, next) =>
    {
        PrintCurrentIntegratedPipelineStage(context, "2nd MW");
        return next.Invoke();
    });
    app.UseStageMarker(PipelineStage.ResolveCache);
    app.Run(context =>
    {
        PrintCurrentIntegratedPipelineStage(context, "3rd MW");
        return context.Response.WriteAsync("Hello world");
    });
    app.UseStageMarker(PipelineStage.Authenticate);
}
```

The output window will display:

```
Current IIS event: AuthenticateRequest Msg: Middleware 1
Current IIS event: AuthenticateRequest Msg: 2nd MW
Current IIS event: AuthenticateRequest Msg: 3rd MW
```

The OMCs all run in the `AuthenticateRequest` stage, because the last OMC registered with the `Authenticate` event, and the `Authenticate` event precedes all other events.

Host OWIN in an Azure Worker Role

2/20/2020 • 4 minutes to read • Edit Online

by Mike Wasson

This tutorial shows how to self-host OWIN in a Microsoft Azure worker role.

Open Web Interface for .NET (OWIN) defines an abstraction between .NET web servers and web applications. OWIN decouples the web application from the server, which makes OWIN ideal for self-hosting a web application in your own process, outside of IIS–for example, inside an Azure worker role.

In this tutorial, you'll learn how to self-host an OWIN applications inside a Microsoft Azure worker role. To learn more about worker roles, see Azure Execution Models.

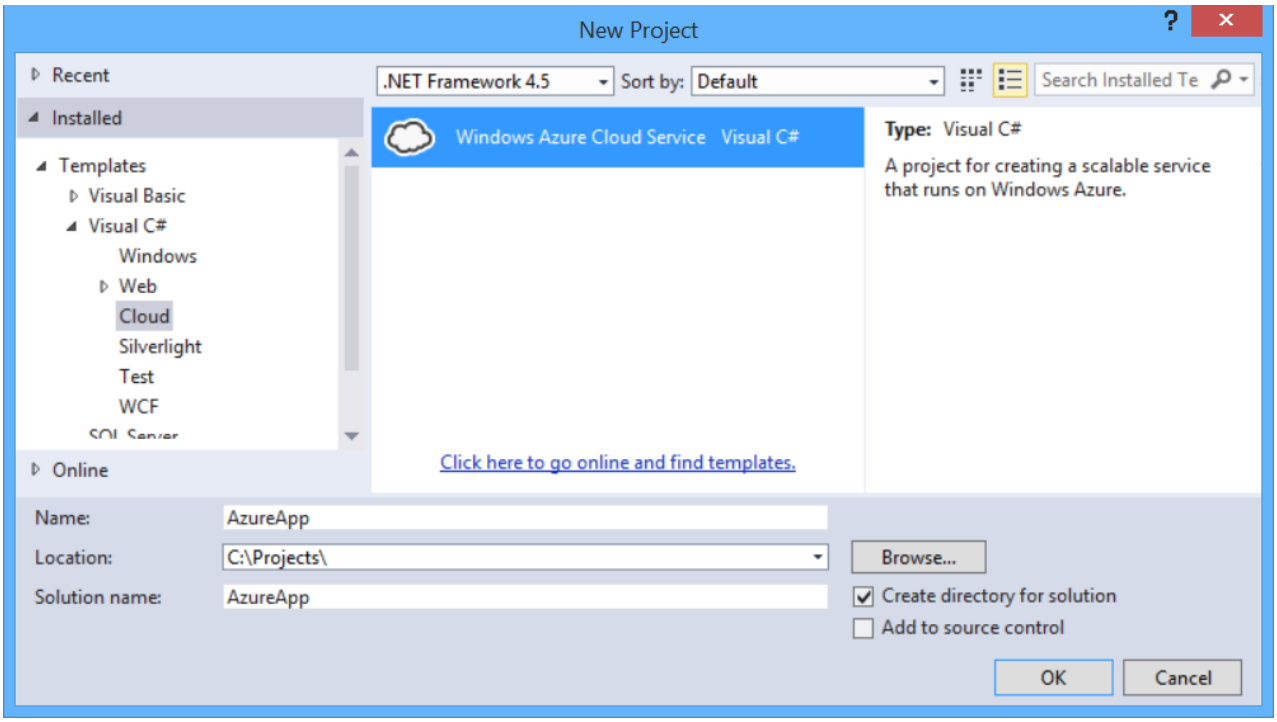
Software versions used in the tutorial

- Visual Studio 2013
- Azure SDK for .NET 2.3
- Microsoft.Owin.Selfhost 2.1.0

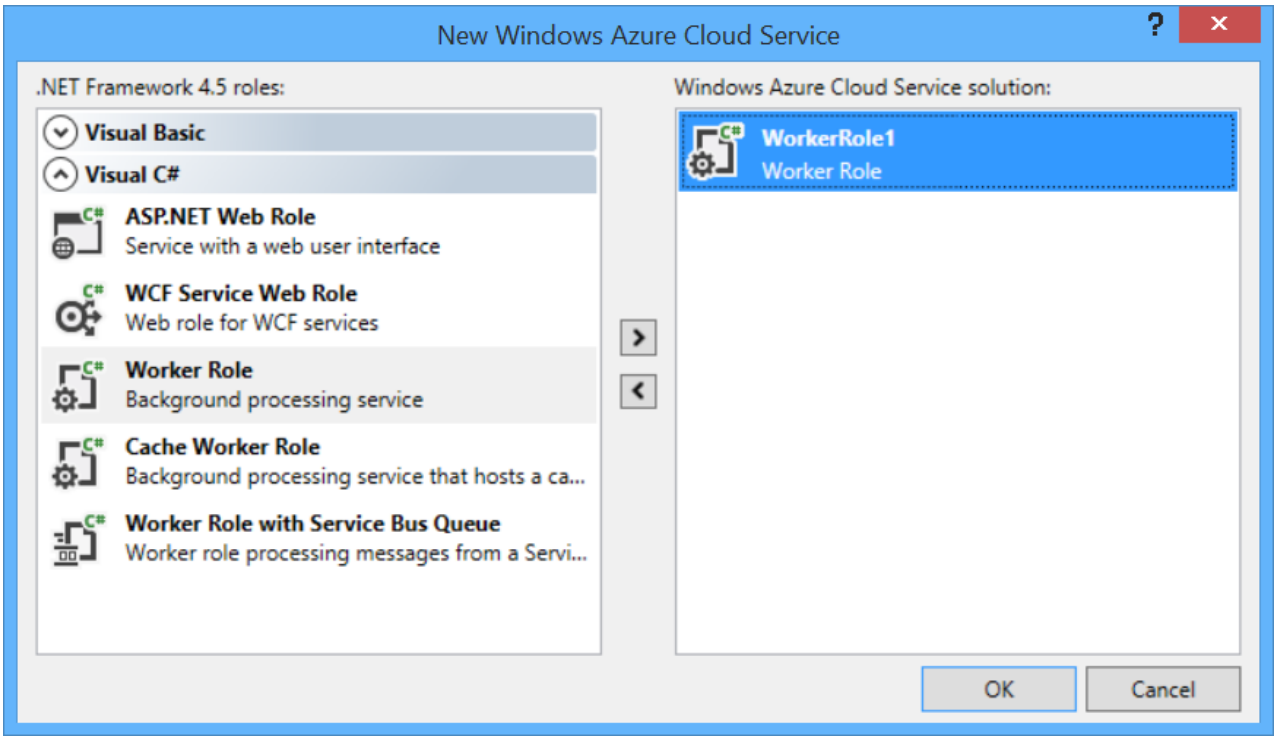
Create a Microsoft Azure Project

Start Visual Studio with administrator privileges. Administrator privileges are needed to debug the application locally, using the Azure compute emulator.

On the **File** menu, click **New**, then click **Project**. From **Installed Templates**, under Visual C#, click **Cloud** and then click **Windows Azure Cloud Service**. Name the project "AzureApp" and click **OK**.



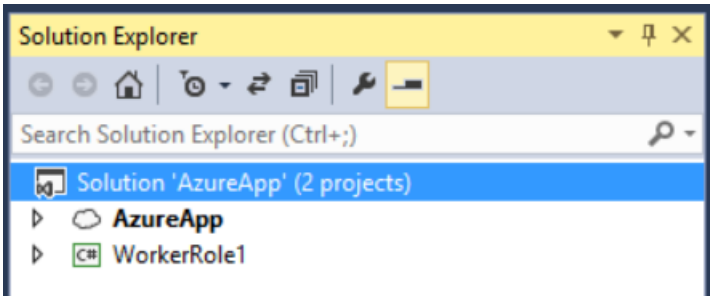
In the **New Windows Azure Cloud Service** dialog, double-click **Worker Role**. Leave the default name ("WorkerRole1"). This step adds a worker role to the solution. Click **OK**.



The Visual Studio solution that is created contains two projects:

- "AzureApp" defines the roles and configuration for the Azure application.
- "WorkerRole1" contains the code for the worker role.

In general, an Azure application can contain multiple roles, although this tutorial uses a single role.



Add the OWIN Self-Host Packages

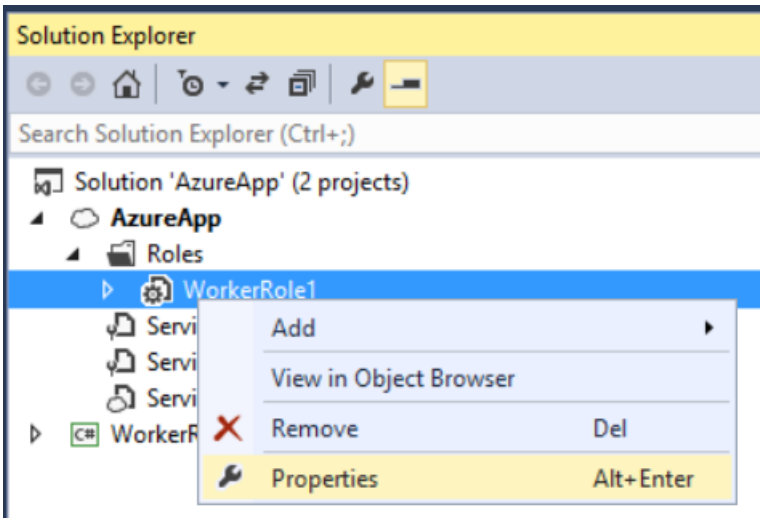
From the **Tools** menu, click **NuGet Package Manager**, then click **Package Manager Console**.

In the Package Manager Console window, enter the following command:

```
Install-Package Microsoft.Owin.SelfHost
```

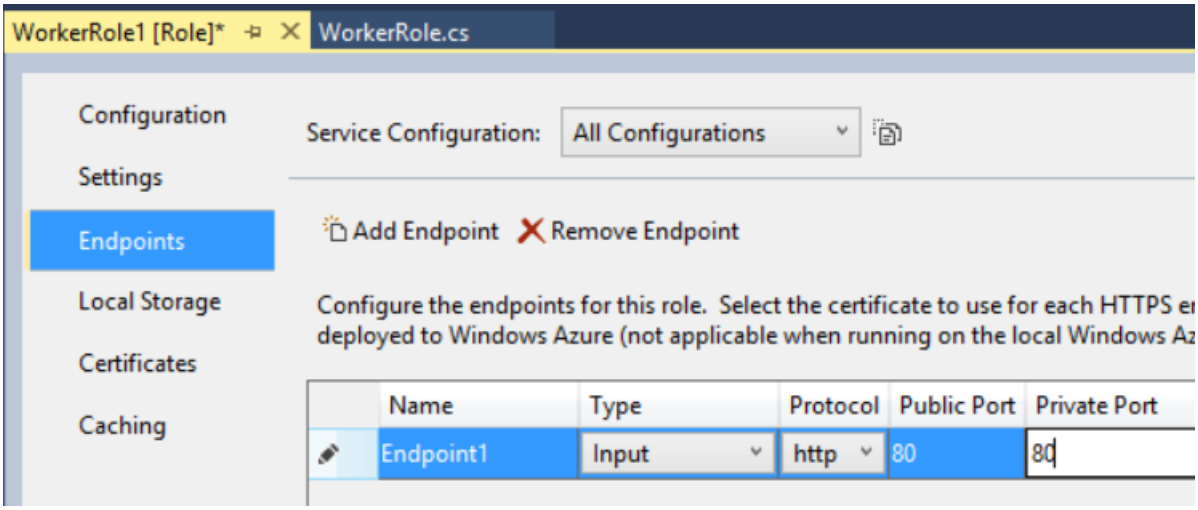
Add an HTTP Endpoint

In Solution Explorer, expand the AzureApp project. Expand the Roles node, right-click WorkerRole1, and select **Properties**.



Click **Endpoints**, and then click **Add Endpoint**.

In the **Protocol** dropdown list, select "http". In **Public Port** and **Private Port**, type 80. These port numbers can be different. The public port is what clients use when they send a request to the role.



Create the OWIN Startup Class

In Solution Explorer, right click the WorkerRole1 project and select **Add** / **Class** to add a new class. Name the class `Startup`.

Replace all of the boilerplate code with the following:

```
using Owin;

namespace WorkerRole1
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.UseWelcomePage("/");
        }
    }
}
```

The `UseWelcomePage` extension method adds a simple HTML page to your application, to verify the site is working.

Start the OWIN Host

Open the WorkerRole.cs file. This class defines the code that runs when the worker role is started and stopped.

Add the following using statement:

```
using Microsoft.Owin.Hosting;
```

Add an **IDisposable** member to the `WorkerRole` class:

```
public class WorkerRole : RoleEntryPoint
{
    private IDisposable _app = null;

    // ....
}
```

In the `OnStart` method, add the following code to start the host:

```
public override bool OnStart()
{
    ServicePointManager.DefaultConnectionLimit = 12;

    // New code:
    var endpoint = RoleEnvironment.CurrentRoleInstance.InstanceEndpoints["Endpoint1"];
    string baseUri = String.Format("{0}://{1}",
        endpoint.Protocol, endpoint.IPEndpoint);

    Trace.TraceInformation(String.Format("Starting OWIN at {0}", baseUri),
        "Information");

    _app = WebApp.Start<Startup>(new StartOptions(url: baseUri));
    return base.OnStart();
}
```

The **WebApp.Start** method starts the OWIN host. The name of the `Startup` class is a type parameter to the method. By convention, the host will call the `Configure` method of this class.

Override the `OnStop` to dispose of the *_app* instance:

```
public override void OnStop()
{
    if (_app != null)
    {
        _app.Dispose();
    }
    base.OnStop();
}
```

Here is the complete code for WorkerRole.cs:

```
using Microsoft.Owin.Hosting;
using Microsoft.WindowsAzure.ServiceRuntime;
using System;
using System.Diagnostics;
using System.Net;
using System.Threading;

namespace WorkerRole1
{
    public class WorkerRole : RoleEntryPoint
    {
        private IDisposable _app = null;

        public override void Run()
        {
            Trace.TraceInformation("WorkerRole entry point called", "Information");

            while (true)
            {
                Thread.Sleep(10000);
                Trace.TraceInformation("Working", "Information");
            }
        }

        public override bool OnStart()
        {
            ServicePointManager.DefaultConnectionLimit = 12;

            var endpoint = RoleEnvironment.CurrentRoleInstance.InstanceEndpoints["Endpoint1"];
            string baseUri = String.Format("{0}://{1}",
                endpoint.Protocol, endpoint.IPEndpoint);

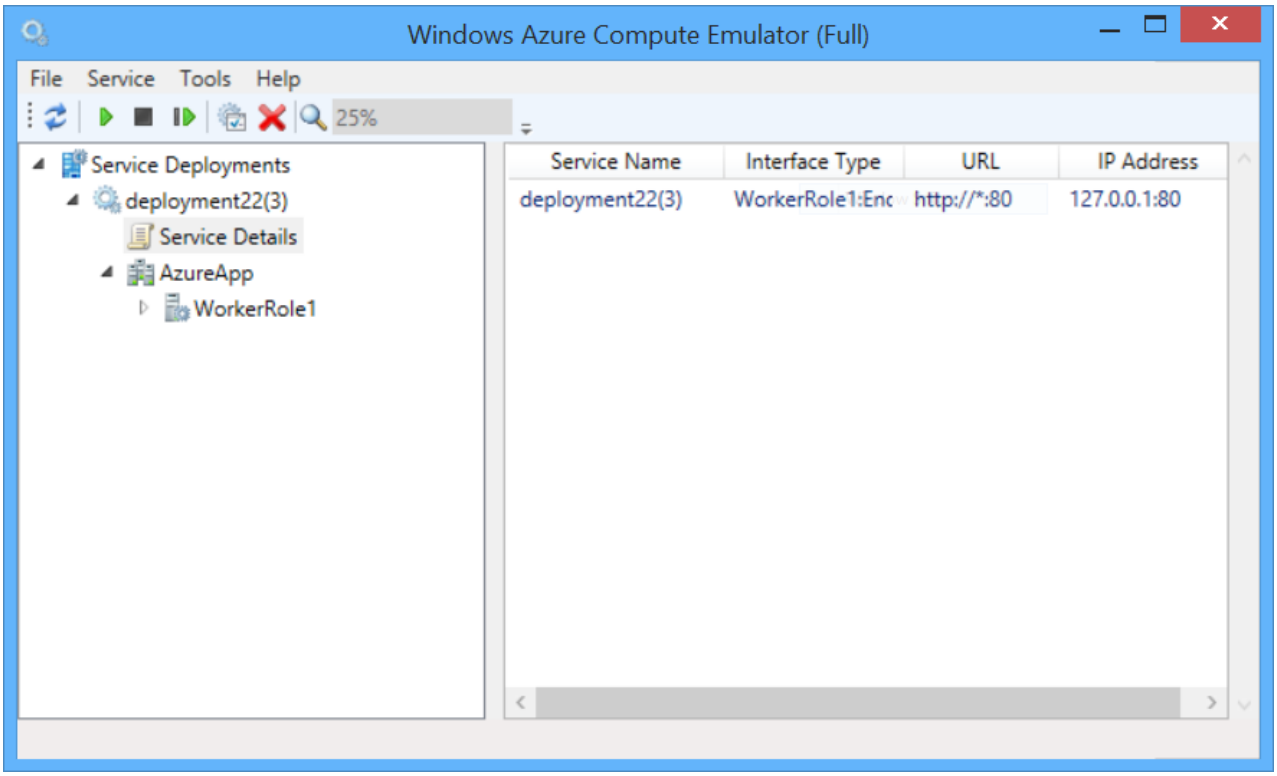
            Trace.TraceInformation(String.Format("Starting OWIN at {0}", baseUri),
                "Information");

            _app = WebApp.Start<Startup>(new StartOptions(url: baseUri));
           return base.OnStart();
        }

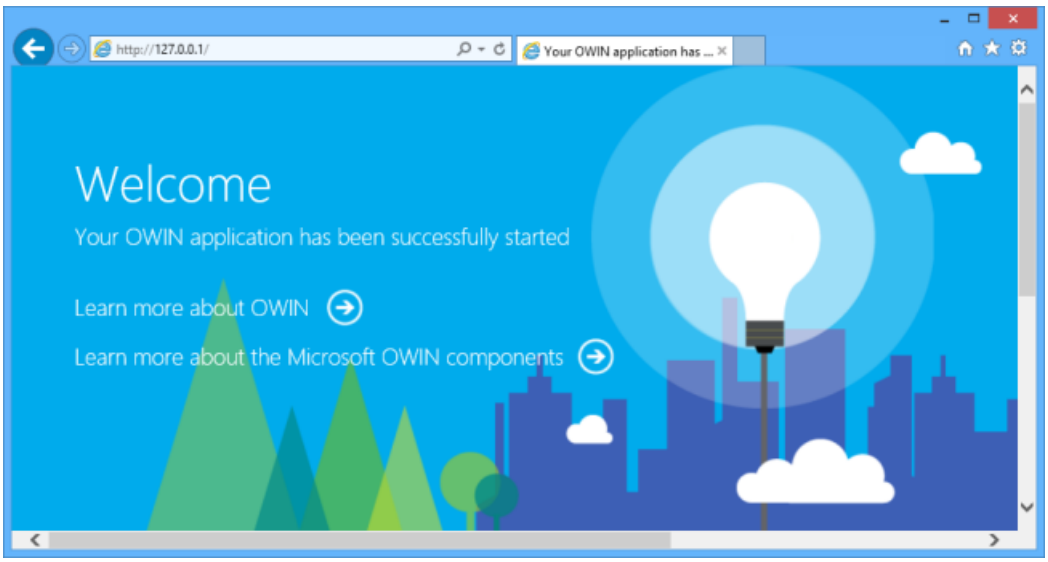
        public override void OnStop()
        {
            if (_app != null)
            {
                _app.Dispose();
            }
            base.OnStop();
        }
    }
}
```

Build the solution, and press F5 to run the application locally in the Azure Compute Emulator. Depending on your firewall settings, you might need to allow the emulator through your firewall.

The compute emulator assigns a local IP address to the endpoint. You can find the IP address by viewing the Compute Emulator UI. Right-click the emulator icon in the task bar notification area, and select **Show Compute Emulator UI**.



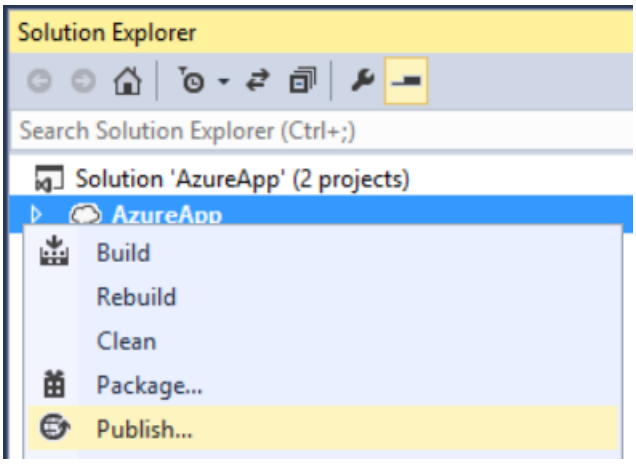
Find the IP address under Service Deployments, deployment [id], Service Details. Open a web browser and navigate to http://*address*, where *address* is the IP address assigned by the compute emulator; for example, `http://127.0.0.1:80`. You should see the OWIN welcome page:



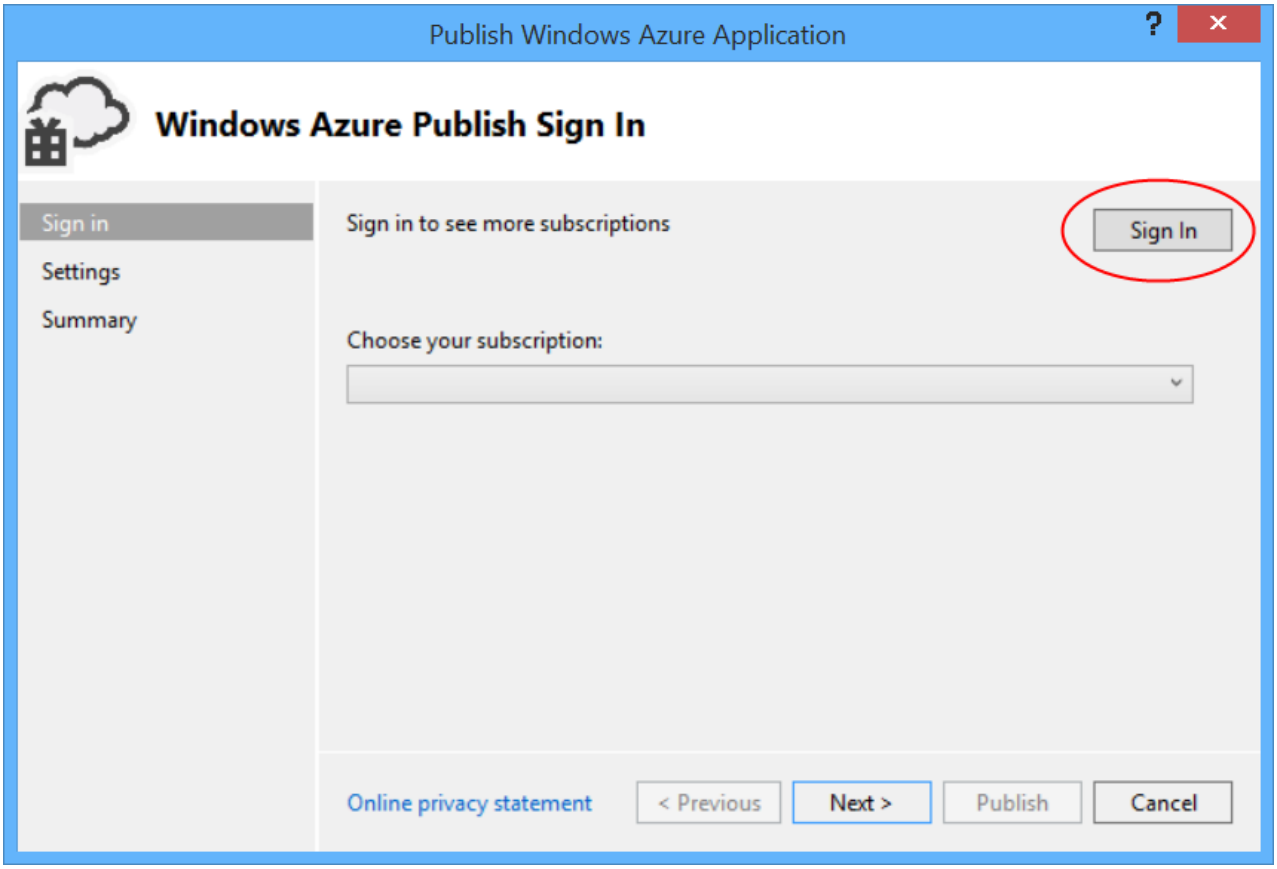
Deploy to Azure

For this step, you must have an Azure account. If you don't already have one, you can create a free trial account in just a couple of minutes. For details, see Microsoft Azure Free Trial.

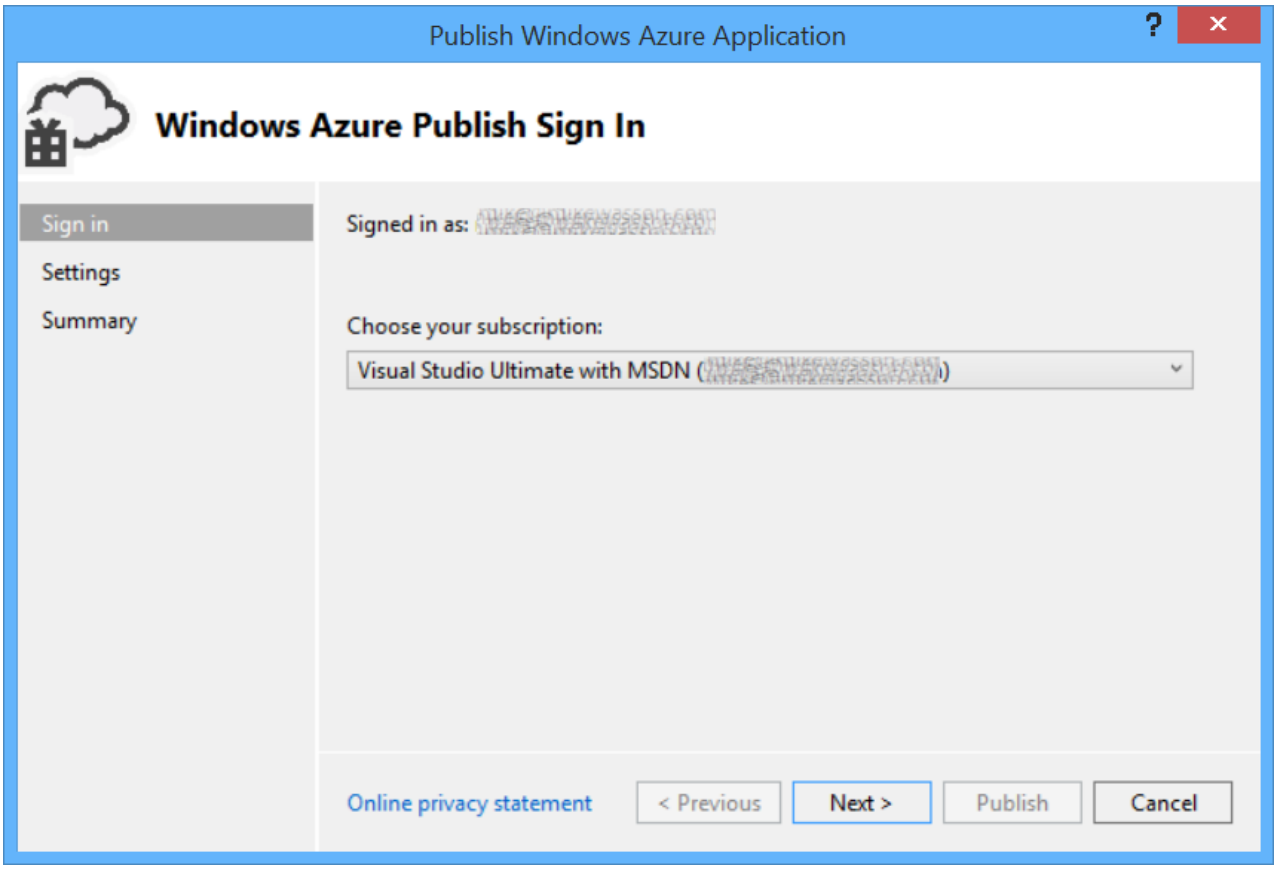
In Solution Explorer, right-click the AzureApp project. Select **Publish**.



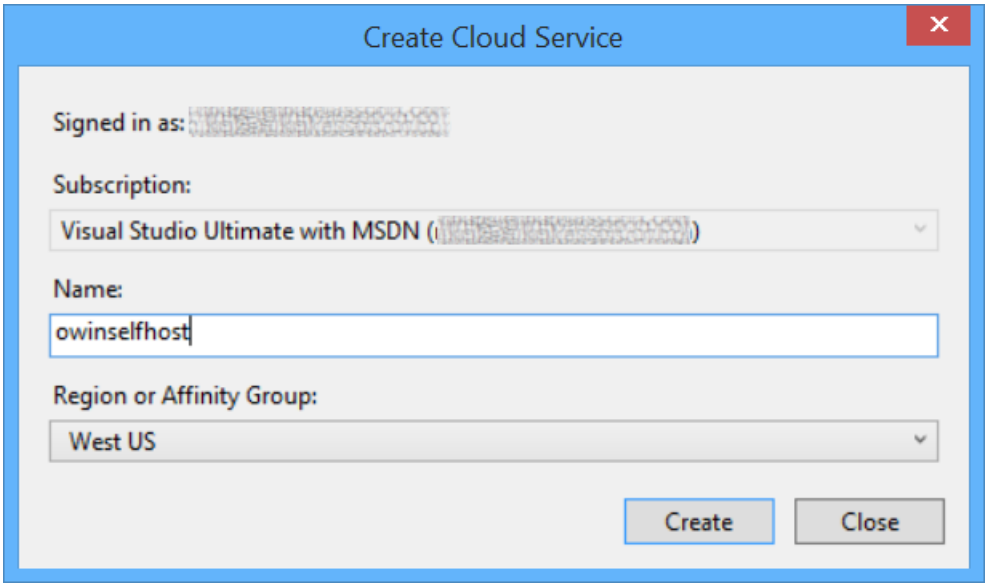
If you are not signed in to your Azure account, click **Sign In**.



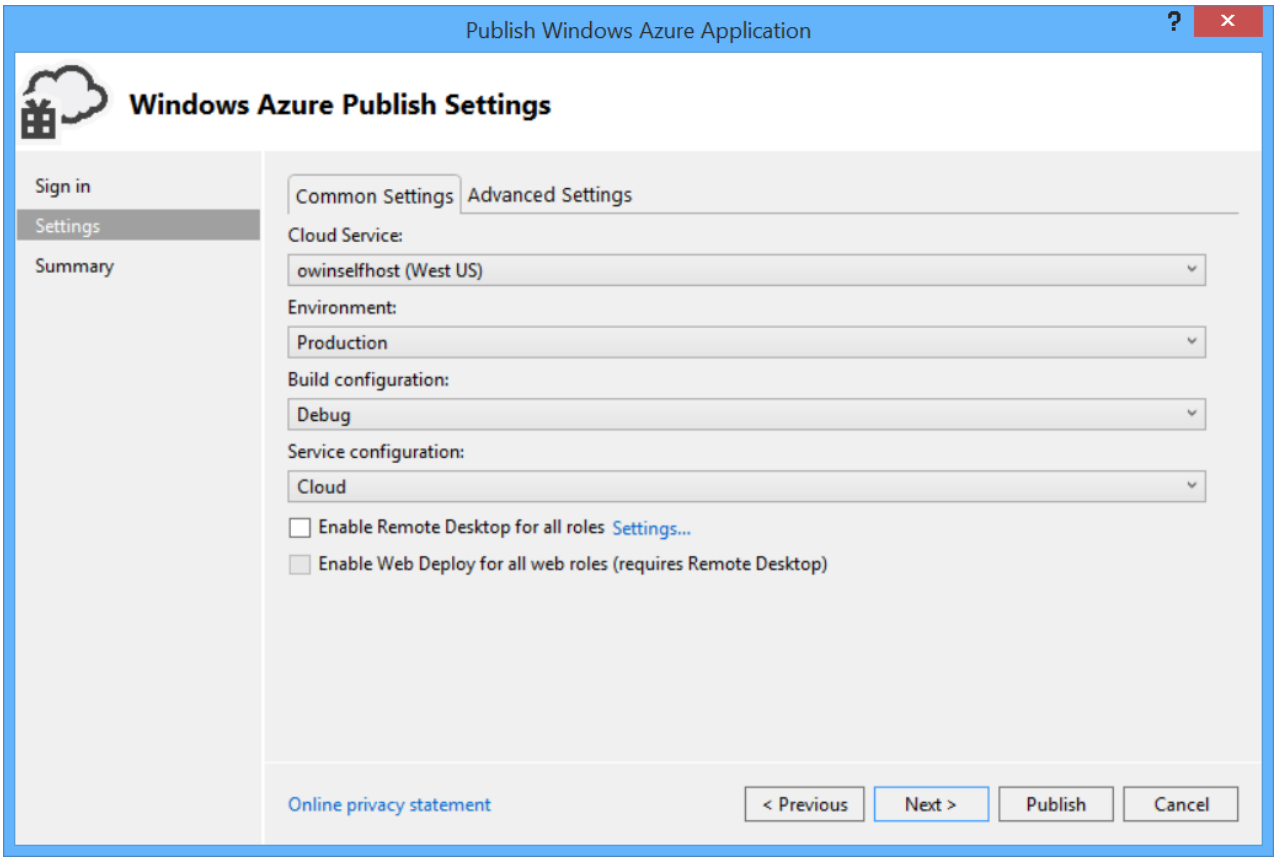
After you are signed in, choose a subscription and click **Next**.



Enter a name for the cloud service and choose a region. Click **Create**.



Click **Publish**.



The Azure Activity Log window shows the progress of the deployment. When the app is deployed, browse to `http://appname.cloudapp.net/`, where *appname* is the name of your cloud service.

Additional Resources

- An Overview of Project Katana
- Katana Project on GitHub

Whitepapers

2/20/2020 • 2 minutes to read • Edit Online

- Whitepapers
- ASP.NET 4 and Visual Studio 2010 Web Development Overview
  - ASP.NET 4 and Visual Studio 2010 Web Development Overview
  - ASP.NET 4 Breaking Changes
- Upgrading an ASP.NET MVC 1.0 Application to ASP.NET MVC 2
- ASP.NET Denied Access to IIS Directories
- What's New in ASP.NET MVC 2
- ASP.NET MVC 3
- ASP.NET MVC 4
- Running ASP.NET 1.1 with IIS 6.0
- How To: Add Mobile Pages to Your ASP.NET Web Forms / MVC Application
- Fix for 'Server Application Unavailable' Error after Applying Security Update for IE
- ASP.NET Side-by-Side Execution of .NET Framework 1.0 and 1.1
- ASP.NET MVC 4
- Request Validation - Preventing Script Attacks
- ASP.NET Data Access - Recommended Resources
- ASP.NET Web Deployment - Recommended Resources
- ASP.NET and Web Tools 2012.2 Release Notes
- What's New in ASP.NET 4.5 and Visual Studio 2012

Whitepapers

2/20/2020 • 15 minutes to read • Edit Online

On this page you will find whitepapers to help you install and configure ASP.NET, and to assist you to write secure, fast and flexible ASP.NET applications.

- ASP.NET 4
- ASP.NET Security Whitepapers
- Installation and Setup Whitepapers
- SQL Server Whitepapers
- General Whitepapers

ASP.NET 4

Information related to ASP.NET 4 and Visual Studio 2010.

ASP.NET MVC 4 Release Notes

This document describes new features and improvements introduced in the ASP.NET MVC 4 Developer Preview for Visual Studio 2010, as well as installation notes and known issues.

ASP.NET MVC 3 Release Notes

This document describes new features and improvements introduced in ASP.NET MVC 3, as well as installation notes and known issues.

ASP.NET 4 and Visual Studio 2010 Web Development Overview

Many exciting changes for ASP.NET are coming in the .NET Framework version 4. This document gives an overview of many of the new features that are included in the upcoming release.

ASP.NET 4 Beta 2 Breaking Changes

This document describes changes that have been made for the .NET Framework version 4 Beta 2 release (that is, the ASP.NET 4 Beta 2 release) that can potentially affect applications that were created using earlier releases, including the ASP.NET 4 Beta 1 release.

What's New in ASP.NET MVC 2

This document describes new features and improvements introduced in ASP.NET MVC 2.

Upgrading an ASP.NET MVC 1.0 Application to ASP.NET MVC 2

ASP.NET MVC 2 can be installed side by side with ASP.NET MVC 1.0 on the same server. This gives application developers flexibility in choosing when to upgrade an ASP.NET MVC 1.0 application to ASP.NET MVC 2. This document descibes both how to upgrade manually and with a wizard in Visual...

ASP.NET Security Whitepapers

Security is an important aspect of internet applications, and these whitepapers discuss how to design and implement secure ASP.NET applications.

Instrument ASP.NET 2.0 Applications for Security

This How To shows you how to use custom health monitoring events to instrument your ASP.NET application to

track security-related events and operations. ASP.NET version 2.0 provides health monitoring that includes instrumentation for many standard ...

Perform a Security Deployment Review for ASP.NET 2.0

This How To shows you how to perform a security deployment review for an ASP.NET 2.0 application to identify potential security vulnerabilities introduced by inappropriate configuration settings. The majority of the review process involves making...

Use ADAM for Roles in ASP.NET 2.0

This How To shows you how you can develop an ASP.NET Web site that uses Active Directory Application Mode (ADAM) to store ASP.NET roles. It shows you how to configure ADAM and the Authorization Manager (AzMan) policy store, how to create new roles and...

Use Authorization Manager (AzMan) with ASP.NET 2.0

This How To shows you how to use the Authorization Manager (AzMan) in conjunction with the ASP.NET role manager API to manage roles, check user role membership, and authorize roles to perform specific operations against an AzMan policy store. The How To...

Use Membership in ASP.NET 2.0

This How To shows how to use the membership feature in ASP.NET version 2.0 applications. It shows you how to use two different membership providers: the ActiveDirectoryMembershipProvider and the SqlMembershipProvider. The membership feature...

Use Role Manager in ASP.NET 2.0

This How To shows you how to use the ASP.NET 2.0 role manager. The role manager eases the task of managing roles and performing role-based authorization in your application. It shows how to configure the various role providers for use with your...

Use Windows Authentication in ASP.NET 2.0

This How To shows you how to configure and use Windows authentication in an ASP.NET Web application. Windows authentication is the preferred approach whenever users are a part of your Windows domain. This approach enables you to use an existing identity store...

Perform a Security Code Review for Managed Code (Baseline Activity)

This How To shows you how to perform security code reviews. This module presents the steps involved in the activity, and techniques for analyzing your results. Use this How To with "Security Question List: Managed Code (.NET Framework 2.0)" ...

Perform a Security Deployment Review for ASP.NET 2.0

This How To shows you how to perform a security deployment review for an ASP.NET 2.0 application to identify potential security vulnerabilities introduced by inappropriate configuration settings. The majority of the review process involves making...

Implement Kerberos Delegation for Windows 2000

Kerberos delegation allows you to flow an authenticated identity across multiple physical tiers of an application to support downstream authentication and authorization. This How To shows you the configuration steps required to make this work.

Use Impersonation and Delegation in ASP.NET 2.0

This How To shows you how and when you should use impersonation in ASP.NET 2.0 applications. By default, impersonation is turned off, and you can access resources by using the ASP.NET Web application's process identity.

However, you can use...

Create a Threat Model for a Web Application at Design Time

This How To describes an approach for creating a threat model for a Web application. The threat modeling activity helps you to model your security design so that you can expose potential security design flaws and vulnerabilities before you invest...

Forms Authentication

Protect Forms Authentication in ASP.NET 2.0

This How To shows you how to securely configure and use forms authentication with ASP.NET 2.0 applications. Key factors to consider include properly securing the authentication ticket and securing the user identity store and access to that store. ...

Use Forms Authentication with Active Directory in ASP.NET 2.0

This How To shows you how to use forms authentication with Microsoft® Active Directory® directory service by using the ActiveDirectoryMembershipProvider. The How To shows you how to configure the provider and create and authenticate users....

Use Forms Authentication with Active Directory in Multiple Domains in ASP.NET 2.0

This How To shows you how to use forms authentication with Microsoft® Active Directory® directory service by using the ActiveDirectoryMembershipProvider. The How To shows you how to configure the provider and create and authenticate users....

Use Forms Authentication with SQL Server in ASP.NET 2.0

This How To shows you how you can use forms authentication with the SQL Server membership provider. Forms authentication with SQL Server is most applicable in situations where users of your application are not part of your Windows domain, and as a result,...

Create GenericPrincipal Objects with Forms Authentication in ASP.NET 1.1

This How To shows you how to create and handle GenericPrincipal and FormsIdentity objects when using Forms authentication.

Use Forms Authentication with Active Directory in ASP.NET 1.1

This How To article shows you how to implement Forms authentication against an Active Directory credential store.

Use Forms Authentication with SQL Server in ASP.NET 1.1

This How To shows you how to implement Forms authentication against a SQL Server credential store. It also shows you how to store password digests in the database.

User Input Data Validation

Request Validation - Preventing Script Attacks

This paper describes the request validation feature of ASP.NET where, by default, the application is prevented from processing unencoded HTML content submitted to the server. This request validation feature can be disabled when the application has been...

Prevent Cross-Site Scripting in ASP.NET

This How To shows how you can help protect your ASP.NET applications from cross-site scripting attacks by using proper input validation techniques and by encoding the output. It also describes a number of other protection mechanisms that you can use in...

Protect From SQL Injection in ASP.NET

This How To shows a number of ways to help protect your ASP.NET application from SQL injection attacks. SQL injection can occur when an application uses input to construct dynamic SQL statements or when it uses stored procedures to connect to the...

Use Regular Expressions to Constrain Input in ASP.NET

This How To shows how you can use regular expressions within ASP.NET applications to constrain untrusted input. Regular expressions are a good way to validate text fields such as names, addresses, phone numbers, and other user information. You can use...

Code Access Security

Use Code Access Security in ASP.NET 2.0

This How To shows you how to select an appropriate trust level for your application, and where necessary, how to create a custom ASP.NET code access security policy file to define a custom trust level. You can use different code access security trust...

Create a Custom Encryption Permission

This How To describes how to create a custom code access security permission to control programmatic access to unmanaged encryption functionality that Win32® Data Protection API (DPAPI) provides. Use this custom permission with the managed DPAPI wrapper ...

Use Code Access Security Policy to Constrain an Assembly

An administrator can configure code access security policy to constrain the operations of .NET Framework code (assemblies). In this How To, you configure code access security policy to constrain the ability of an assembly to perform file I/O and restrict...

Communications Security

Set Up SSL on a Web Server

A Web server must be configured for SSL in order to support https connections from client applications. This How To shows you how to configure SSL on a Web Server.

Set Up Client Certificates

IIS supports client certificate authentication. This How To shows you how to configure a Web application to require client certificates. It also shows you how to install a certificate on a client computer and use it when calling the Web application.

Use IPSec for Filtering Ports and Authentication

Internet Protocol security (IPSec) is a protocol, not a service, that provides encryption, integrity, and authentication services for IP-based network traffic. Because IPSec provides server-to-server protection, you can use IPSec to counter internal threats...

Use IPSec to Provide Secure Communication Between Two Servers

IPSec is a technology provided by Windows 2000 that allows you to create encrypted channels between two servers. IPSec can be used to filter IP traffic and to authenticate servers. This How To shows you how to configure IPSec to provide a secure (encrypted) ...

Use SSL to Secure Communication with SQL Server

It is often vital for applications to be able to secure the data passed to and from a SQL Server database server. With SQL Server, you can use SSL to create an encrypted channel. This How To shows you how to install a certificate on the database server, ...

Call a Web Service Using Client Certificates from ASP.NET 1.1

This How To describes how you can pass a client certificate to a Web service for authentication from an ASP.NET Web application or from a Windows Forms application. You can install the client certificate in either the local machine store or the user store. If...

Call a Web Service Using SSL from ASP.NET 1.1

Secure Sockets Layer (SSL) encryption can be used to guarantee the integrity and confidentiality of the messages passed to and from a Web service. This How To shows you how to use SSL with Web services.

Cryptography

Create a DPAPI Library in .NET 1.1

This How To shows you how to create a managed class library that exposes DPAPI functionality to applications that want to encrypt data, for example, database connection strings and account credentials.

Create an Encryption Library in .NET 1.1

This How To shows you how to create a managed class library to provide encryption functionality for applications. It allows an application to choose the encryption algorithm. Supported algorithms include DES, Triple DES, RC2, and Rijndael.

Store an Encrypted Connection String in the Registry in ASP.NET 1.1

Applications may choose to store encrypted data such as connection strings and account credentials in the Windows registry. This How To shows you how to store and retrieve encrypted strings in the registry.

Use DPAPI (Machine Store) from ASP.NET 1.1

This How To shows you how to use DPAPI from an ASP.NET Web application or Web service to encrypt sensitive data.

Use DPAPI (User Store) from ASP.NET 1.1 with Enterprise Services

This How To shows you how to use DPAPI from an ASP.NET Web application or service to encrypt sensitive data. This How To uses DPAPI with the user store, which requires the use of an out of process Enterprise Services component.

Installation and Setup Whitepapers

These whitepapers provide step-by-step instructions for installing and configuring ASP.NET on your server.

Create a Service Account for an ASP.NET 2.0 Application

This How To shows you how to create and configure a custom least-privileged service account to run an ASP.NET Web application. By default, an ASP.NET application on Microsoft Windows Server 2003 and IIS 6.0 runs using the built-in Network Service ...

Improve Security When Hosting Multiple Applications in ASP.NET 2.0

This How To shows how you can isolate multiple applications from one another and from shared system resources in a Web hosting environment. The hosting environment might be a Web server provided by an Internet Service Provider (ISP) that hosts multiple ...

Use Medium Trust in ASP.NET 2.0

This How To shows you how to configure ASP.NET Web applications to run in medium trust. If you host multiple applications on the same server, you can use code access security and the medium trust level to provide application isolation. By setting ...

Use the Network Service Account to Access Resources in ASP.NET 2.0

This How To shows you how you can use the NT AUTHORITY\Network Service machine account to access local and network resources. By default on Windows Server 2003, ASP.NET applications run using this account's identity. It is a least privileged...

Use Protocol Transition and Constrained Delegation in ASP.NET 2.0

This How To shows you how to configure and use protocol transition and constrained delegation to allow your ASP.NET application to access network resources while impersonating the original caller. The Microsoft® Windows® 2000 operating system ...

ASP.NET Side-by-Side Execution of .NET Framework 1.0 and 1.1

This whitepaper describes how to install both .NET 1.0 and .NET 1.1 on your machine, allowing an ASP.NET Web application to run on either version of the framework.

ASP.NET Denied Access to IIS Directories

This whitepaper describes what you must do if a request to your ASP.NET application returns the error, "Denied Access to *DirectoryName* directory. Failed to start monitoring directory changes."

Running ASP.NET 1.1 with IIS 6.0

While Windows Server 2003 includes both IIS 6.0 and ASP.NET 1.1, these components are disabled by default. This whitepaper describes how to enable IIS 6.0 and ASP.NET 1.1, and recommends several configuration settings to get the optimal...

Fix for 'Server Application Unavailable' Error after Applying Security Update for IE

This paper describes the patch that fixes an issue with the MS03-32 Security Update for Internet Explorer that affects ASP.NET 1.0 applications running on Windows XP Professional.

Create a Custom Account To Run ASP.NET 1.1

ASP.NET Web applications usually run using the built-in ASPNET account. On occasion, you may want to use a custom account instead. This How To article shows you how to create a least privileged local account to run ASP.NET Web applications. ...

Configuration

Configure the Machine Key in ASP.NET 2.0

This How To explains the <machineKey> element in the Web.config file and shows how to configure the <machineKey> element to control tamper proofing and encryption of ViewState, forms authentication tickets, and role cookies. ViewState is signed...

Encrypt Configuration Sections in ASP.NET 2.0 Using DPAPI

This How To shows how to use the Windows Data Protection application programming interface (DPAPI) protected configuration provider and the Aspnet_regiis.exe tool to encrypt sections of your configuration files. You can use the Aspnet_regiis.exe tool to ...

Encrypt Configuration Sections in ASP.NET 2.0 Using RSA

This How To shows how to use the RSA Protected Configuration provider and the Aspnet_regiis.exe tool to encrypt sections of your configuration files. You can use Aspnet_regiis.exe tool to encrypt sensitive data, such as connection strings, held in the ...

Use Impersonation and Delegation in ASP.NET 2.0

This How To shows you how and when you should use impersonation in ASP.NET 2.0 applications. By default, impersonation is turned off, and you can access resources by using the ASP.NET Web application's process identity. However, you can use...

SQL Server Whitepapers

While ASP.NET works with a variety of databases, these whitepapers look specifically at connecting ASP.NET applications to SQL Server.

Connect to SQL Server Using SQL Authentication in ASP.NET 2.0

This How To shows you how to connect an ASP.NET application securely to Microsoft® SQL Server™ when database access authentication uses native SQL authentication. Windows authentication is the recommended way to connect to SQL Server because you...

Connect to SQL Server Using Windows Authentication in ASP.NET 2.0

This How To shows you how to connect to SQL Server 2000 using a Windows service account from an ASP.NET version 2.0 application. You should use Windows authentication instead of SQL authentication whenever possible because you avoid storing credentials in ...

Use SSL to Secure Communication with SQL Server 2000

It is often vital for applications to be able to secure the data passed to and from a SQL Server database server. With SQL Server, you can use SSL to create an encrypted channel. This How To shows you how to install a certificate on the database server,...

General Whitepapers

The following case study describes the process that was used to migrate Microsoft's .NET community websites from a traditional hosting environment to Microsoft Azure.

Migrating Microsoft's ASP.NET and IIS.NET Community Websites to Microsoft Azure

These whitepapers cover a variety of topics concerning ASP.NET.

Use Health Monitoring in ASP.NET 2.0

This How To shows you how to use health monitoring to instrument your application for a custom event. To create a custom health monitoring event, you create a class that derives from System.Web.Management.WebBaseEvent, configure the <healthMonitoring> ...

Implement Patch Management

This How To explains patch management, including how to keep single or multiple servers up to date. Additional software is not required, except for the tools available for download from Microsoft. Operations and security policy should adopt a patch management...

ASP.NET Server Controls for Silverlight in the Silverlight 3 SDK

The ASP.NET Server Controls for Silverlight ("ASP.NET Silverlight controls"), which are the ASP.NET MediaPlayer and Silverlight controls, have been removed from the Silverlight SDK for Silverlight version 3. This document provides guidance for developers who worked with these ASP.NET...

Building High Performance Web Applications

Learn how to use the new features in ASP.NET Ajax Library to build High Performance Web Applications

Work with SameSite cookies in ASP.NET

2/21/2020 • 12 minutes to read • Edit Online

By Rick Anderson

SameSite is an IETF draft standard designed to provide some protection against cross-site request forgery (CSRF) attacks. Originally drafted in 2016, the draft standard was updated in 2019. The updated standard is not backward compatible with the previous standard, with the following being the most noticeable differences:

- Cookies without SameSite header are treated as `SameSite=Lax` by default.
- `SameSite=None` must be used to allow cross-site cookie use.
- Cookies that assert `SameSite=None` must also be marked as `Secure`.
- Applications that use `<iframe>` may experience issues with `sameSite=Lax` or `sameSite=Strict` cookies because `<iframe>` is treated as cross-site scenarios.
- The value `SameSite=None` is not allowed by the 2016 standard and causes some implementations to treat such cookies as `SameSite=Strict`. See Supporting older browsers in this document.

The `SameSite=Lax` setting works for most application cookies. Some forms of authentication like OpenID Connect (OIDC) and WS-Federation default to POST based redirects. The POST based redirects trigger the SameSite browser protections, so SameSite is disabled for these components. Most OAuth logins are not affected due to differences in how the request flows.

Each ASP.NET component that emits cookies needs to decide if SameSite is appropriate.

See Known Issues for problems with applications after installing the 2019 .Net SameSite updates.

Using SameSite in ASP.NET 4.7.2 and 4.8

.Net 4.7.2 and 4.8 supports the 2019 draft standard for SameSite since the release of updates in December 2019. Developers are able to programmatically control the value of the SameSite header using the HttpCookie.SameSite property. Setting the `SameSite` property to `Strict`, `Lax`, or `None` results in those values being written on the network with the cookie. Setting it equal to `(SameSiteMode)(-1)` indicates that no SameSite header should be included on the network with the cookie. The HttpCookie.Secure Property, or 'requireSSL' in config files, can be used to mark the cookie as `Secure` or not.

New `HttpCookie` instances will default to `SameSite=(SameSiteMode)(-1)` and `Secure=false`. These defaults can be overridden in the `system.web/httpCookies` configuration section, where the string `"Unspecified"` is a friendly configuration-only syntax for `(SameSiteMode)(-1)`:

```
<configuration>
 <system.web>
  <httpCookies sameSite="[Strict|Lax|None|Unspecified]" requireSSL="[true|false]" />
 <system.web>
<configuration>
```

ASP.Net also issues four specific cookies of its own for these features: Anonymous Authentication, Forms Authentication, Session State, and Role Management. Instances of these cookies obtained in runtime can be manipulated using the `SameSite` and `Secure` properties just like any other HttpCookie instance. However, due to the patchwork emergence of the SameSite standard, configuration options for these four features cookies is inconsistent. The relevant configuration sections and attributes, with defaults, are shown below. If there is no `SameSite` or `Secure` related attribute for a feature, then the feature will fall back on the defaults configured in the

`system.web/httpCookies` section discussed above.

```
<configuration>
 <system.web>
  <anonymousIdentification cookieRequireSSL="false" /> <!-- No config attribute for SameSite -->
  <authentication>
   <forms cookieSameSite="Lax" requireSSL="false" />
  </authentication>
  <sessionState cookieSameSite="Lax" /> <!-- No config attribute for Secure -->
  <roleManager cookieRequireSSL="false" /> <!-- No config attribute for SameSite -->
 <system.web>
<configuration>
```

Note: 'Unspecified' is only available to `system.web/httpCookies@sameSite` at the moment. We hope to add similar syntax to the previously shown cookieSameSite attributes in future updates. Setting `(SameSiteMode)(-1)` in code still works on instances of these cookies.*

If you are reading this in a language other than English, let us know in this GitHub discussion issue if you'd like to see the code comments in your native language.

Retarget .NET apps

To target .NET 4.7.2 or later:

- Ensure *web.config* contains the following:

  ```
  <system.web>
    <compilation targetFramework="4.7.2"/>
    <httpRuntime targetFramework="4.7.2"/>
  </system.web>
  ```

- Verify the project file contains the correct TargetFrameworkVersion:

  ```
  <TargetFrameworkVersion>v4.7.2</TargetFrameworkVersion>
  ```

  The .NET Migration Guide has more details.

- Verify NuGet packages in the project are targeted at the correct framework version. You can verify the correct framework version by examining the *packages.config* file, for example:

  ```
  <?xml version="1.0" encoding="utf-8"?>
  <packages>
    <package id="Microsoft.AspNet.Mvc" version="5.2.7" targetFramework="net472" />
    <package id="Microsoft.ApplicationInsights" version="2.4.0" targetFramework="net451" />
  </packages>
  ```

  In the preceding *packages.config* file, the `Microsoft.ApplicationInsights` package:

  - Is targeted against .NET 4.5.1.
  - Should have its `targetFramework` attribute updated to `net472` if an updated package targeting your framework target exists.

.NET versions earlier than 4.7.2

Microsoft does not support .NET versions lower that 4.7.2 for writing the same-site cookie attribute. We have not found a reliable way to:

- Ensure the attribute is written correctly based on browser version.

- Intercept and adjust authentication and session cookies on older framework versions.

December patch behavior changes

The specific behavior change for .NET Framework is how the `SameSite` property interprets the `None` value:

- Before the patch a value of `None` meant:
  - Do not emit the attribute at all.
- After the patch:
  - A value of `None` it means "Emit the attribute with a value of `None`".
  - A `SameSite` value of `(SameSiteMode)(-1)` causes the attribute not to be emitted.

The default SameSite value for forms authentication and session state cookies was changed from `None` to `Lax`.

Summary of change impact on browsers

If you install the patch and issue a cookie with `SameSite.None`, one of two things will happen:

- Chrome v80 will treat this cookie according to the new implementation, and not enforce same site restrictions on the cookie.
- Any browser that has not been updated to support the new implementation will follow the old implementation. The old implementation says:
  - If you see a value you don't understand, ignore it and switch to strict same site restrictions.

So either the app breaks in Chrome, or you break in numerous other places.

History and changes

SameSite support was first implemented in .NET 4.7.2 using the 2016 draft standard.

The November 19, 2019 updates for Windows updated .NET 4.7.2+ from the 2016 standard to the 2019 standard. Additional updates are forthcoming for other versions of Windows. For more information, see KB articles that support SameSite in .NET Framework.

The 2019 draft of the SameSite specification:

- Is **not** backwards compatible with the 2016 draft. For more information, see Supporting older browsers in this document.
- Specifies cookies are treated as `SameSite=Lax` by default.
- Specifies cookies that explicitly assert `SameSite=None` in order to enable cross-site delivery should also be marked as `Secure`.
- Is supported by patches issued as described in the KB's listed above.
- Is scheduled to be enabled by Chrome by default in Feb 2020. Browsers started moving to this standard in 2019.

Known Issues

Because the 2016 and 2019 draft specifications are not compatible, the November 2019 .Net Framework update introduces some changes that may be breaking.

- Session State and Forms Authentication cookies are now written to the network as `Lax` instead of unspecified.
  - While most apps work with `SameSite=Lax` cookies, apps that POST across sites or applications that make use of `iframe` may find that their session state or forms authorization cookies aren't being used as expected. To remedy this, change the `cookieSameSite` value in the appropriate configuration section as discussed previously.
- HttpCookies that explicitly set `SameSite=None` in code or configuration now have that value written with the

cookie, whereas it was previously omitted. This may cause issues with older browsers that only support the 2016 draft standard.

- When targeting browsers supporting the 2019 draft standard with `SameSite=None` cookies, remember to also mark them `Secure` or they may not be recognized.
- To revert to the 2016 behavior of not writing `SameSite=None`, use the app setting `aspnet:SupressSameSiteNone=true`. Note that this will apply to all HttpCookies in the app.

Azure App Service—SameSite cookie handling

See Azure App Service—SameSite cookie handling and .NET Framework 4.7.2 patch for information about how Azure App Service is configuring SameSite behaviors in .Net 4.7.2 apps.

Supporting older browsers

The 2016 SameSite standard mandated that unknown values must be treated as `SameSite=Strict` values. Apps accessed from older browsers which support the 2016 SameSite standard may break when they get a SameSite property with a value of `None`. Web apps must implement browser detection if they intend to support older browsers. ASP.NET doesn't implement browser detection because User-Agents values are highly volatile and change frequently.

Microsoft's approach to fixing the problem is to help you implement browser detection components to strip the `sameSite=None` attribute from cookies if a browser is known to not support it. Google's advice was to issue double cookies, one with the new attribute, and one without the attribute at all. However we consider Google's advice limited. Some browsers, especially mobile browsers have very small limits on the number of cookies a site, or a domain name can send. Sending multiple cookies, especially large cookies like authentication cookies can reach the mobile browser limit very quickly, causing app failures that are hard to diagnose and fix. Furthermore as a framework there is a large ecosystem of third party code and components that may not be updated to use a double cookie approach.

The browser detection code used in the sample projects in is contained in two files

- C# SameSiteSupport.cs
- VB SameSiteSupport.vb

These detections are the most common browser agents we have seen that support the 2016 standard and for which the attribute needs to be completely removed. It isn't meant as a complete implementation:

- Your app may see browsers that our test sites do not.
- You should be prepared to add detections as necessary for your environment.

How you wire up the detection varies according the version of .NET and the web framework that you are using. The following code can be called at the HttpCookie call site:

```
private void CheckSameSite(HttpContext httpContext, HttpCookie cookie)
{
    if (cookie.SameSite == SameSiteMode.None)
    {
        var userAgent = httpContext.Request.UserAgent;
        if (BrowserDetection.DisallowsSameSiteNone(userAgent))
        {
            cookie.SameSite = (SameSiteMode)(-1);
        }
    }
}
```

See the following ASP.NET 4.7.2 SameSite cookie topics:

- C# MVC

- C# WebForms
- VB WebForms
- VB MVC

Ensuring your site redirects to HTTPS

For ASP.NET 4.x, WebForms and MVC, IIS's URL Rewrite feature can be used to redirect all requests to HTTPS. The following XML shows a sample rule:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>
    <rewrite>
      <rules>
        <rule name="Redirect to https" stopProcessing="true">
          <match url="(.*)"/>
          <conditions>
            <add input="{HTTPS}" pattern="Off"/>
            <add input="{REQUEST_METHOD}" pattern="^get$|^head$" />
          </conditions>
          <action type="Redirect" url="https://{HTTP_HOST}/{R:1}" redirectType="Permanent"/>
        </rule>
      </rules>
    </rewrite>
  </system.webServer>
</configuration>
```

In on-premises installations of IIS URL Rewrite is an optional feature that may need installing.

Test apps for SameSite problems

You must test your app with the browsers you support and go through your scenarios that involve cookies. Cookie scenarios typically involve

- Login forms
- External login mechanisms such as Facebook, Azure AD, OAuth and OIDC
- Pages that accept requests from other sites
- Pages in your app designed to be embedded in iframes

You should check that cookies are created, persisted and deleted correctly in your app.

Apps that interact with remote sites such as through third-party login need to:

- Test the interaction on multiple browsers.
- Apply the browser detection and mitigation discussed in this document.

Test web apps using a client version that can opt-in to the new SameSite behavior. Chrome, Firefox, and Chromium Edge all have new opt-in feature flags that can be used for testing. After your app applies the SameSite patches, test it with older client versions, especially Safari. For more information, see Supporting older browsers in this document.

Test with Chrome

Chrome 78+ gives misleading results because it has a temporary mitigation in place. The Chrome 78+ temporary mitigation allows cookies less than two minutes old. Chrome 76 or 77 with the appropriate test flags enabled provides more accurate results. To test the new SameSite behavior toggle `chrome://flags/#same-site-by-default-cookies` to **Enabled**. Older versions of Chrome (75 and below) are reported to fail with the new `None` setting. See Supporting older browsers in this document.

Google does not make older chrome versions available. Follow the instructions at Download Chromium to test

older versions of Chrome. Do **not** download Chrome from links provided by searching for older versions of chrome.

- Chromium 76 Win64
- Chromium 74 Win64
- If you're not using a 64bit version of Windows you can use the OmahaProxy viewer to look up which Chromium branch corresponds to Chrome 74 (v74.0.3729.108) using the instructions provided by Chromium.

Starting in Canary version `80.0.3975.0`, the Lax+POST temporary mitigation can be disabled for testing purposes using the new flag `--enable-features=SameSiteDefaultChecksMethodRigorously` to allow testing of sites and services in the eventual end state of the feature where the mitigation has been removed. For more information, see The Chromium Projects SameSite Updates

Test with Chrome 80+

Download a version of Chrome that supports their new attribute. At the time of writing, the current version is Chrome 80. Chrome 80 needs the flag `chrome://flags/#same-site-by-default-cookies` enabled to use the new behavior. You should also enable (`chrome://flags/#cookies-without-same-site-must-be-secure`) to test the upcoming behavior for cookies which have no sameSite attribute enabled. Chrome 80 is on target to make the switch to treat cookies without the attribute as `SameSite=Lax`, albeit with a timed grace period for certain requests. To disable the timed grace period Chrome 80 can be launched with the following command line argument:

```
--enable-features=SameSiteDefaultChecksMethodRigorously
```

Chrome 80 has warning messages in the browser console about missing sameSite attributes. Use F12 to open the browser console.

Test with Safari

Safari 12 strictly implemented the prior draft and fails when the new `None` value is in a cookie. `None` is avoided via the browser detection code Supporting older browsers in this document. Test Safari 12, Safari 13, and WebKit based OS style logins using MSAL, ADAL or whatever library you are using. The problem is dependent on the underlying OS version. OSX Mojave (10.14) and iOS 12 are known to have compatibility problems with the new SameSite behavior. Upgrading the OS to OSX Catalina (10.15) or iOS 13 fixes the problem. Safari does not currently have an opt-in flag for testing the new spec behavior.

Test with Firefox

Firefox support for the new standard can be tested on version 68+ by opting in on the `about:config` page with the feature flag `network.cookie.sameSite.laxByDefault`. There haven't been reports of compatibility issues with older versions of Firefox.

Test with Edge (Legacy) browser

Edge supports the old SameSite standard. Edge version 44+ doesn't have any known compatibility problems with the new standard.

Test with Edge (Chromium)

SameSite flags are set on the `edge://flags/#same-site-by-default-cookies` page. No compatibility issues were discovered with Edge Chromium.

Test with Electron

Versions of Electron include older versions of Chromium. For example, the version of Electron used by Teams is Chromium 66, which exhibits the older behavior. You must perform your own compatibility testing with the version of Electron your product uses. See Supporting older browsers.

Reverting SameSite patches

You can revert the updated sameSite behavior in .NET Framework apps to its previous behavior where the

sameSite attribute is not emitted for a value of `None`, and revert the authentication and session cookies to not emit the value. This should be viewed as an *extremely temporary fix*, as the Chrome changes will break any external POST requests or authentication for users using browsers which support the changes to the standard.

Reverting .NET 4.7.2 behavior

Update *web.config* to include the following configuration settings:

```
<configuration>
  <appSettings>
    <add key="aspnet:SuppressSameSiteNone" value="true" />
  </appSettings>

  <system.web>
    <authentication>
      <forms cookieSameSite="None" />
    </authentication>
    <sessionState cookieSameSite="None" />
  </system.web>
</configuration>
```

Additional resources

- Upcoming SameSite Cookie Changes in ASP.NET and ASP.NET Core
- Tips for testing and debugging SameSite-by-default and "SameSite=None; Secure" cookies
- Chromium Blog:Developers: Get Ready for New SameSite=None; Secure Cookie Settings
- SameSite cookies explained
- Chrome Updates
- .NET SameSite Patches
- Azure Web Applications Same Site Information
- Azure ActiveDirectory Same Site Information

KB articles that support SameSite in .NET Framework

2/20/2020 • 2 minutes to read • Edit Online

PRODUCT VERSION	CUMULATIVE	UPDATE
Windows 10 1909 and Windows Server, version 1909		
.NET Framework 4.8	Catalog	4533002
Windows 10 1903 and Windows Server, version 1903		
.NET Framework 4.8	Catalog	4533002
Windows 10 1809 (October 2018 Update) and Windows Server 2019		
.NET Framework 4.7.2	Catalog	4533013
.NET Framework 4.8	Catalog	4533001
Windows 10 1803 (April 2018 Update)		
.NET Framework 4.7.2	Catalog	4530717
.NET Framework 4.8	Catalog	4533000
Windows 10 1709 (Fall Creators Update)		
.NET Framework 4.7.1, 4.7.2	Catalog	4530714
.NET Framework 4.8	Catalog	4532999
Windows 10 1703 (Creators Update)		
.NET Framework 4.7, 4.7.1, 4.7.2	Catalog	4530711
.NET Framework 4.8	Catalog	4532998
Windows 10 1607 (Anniversary Update) and Windows Server 2016		
.NET Framework 4.6.2, 4.7, 4.7.1, 4.7.2	Catalog	4530689

PRODUCT VERSION	CUMULATIVE	UPDATE
.NET Framework 4.8	Catalog	4532997

The following table is for earlier versions of Windows and Windows Server.

PRODUCT VERSION	SECURITY AND	QUALITY ROLLUP
Windows 8.1 and Windows Server 2012 R2		
.NET Framework 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2	Catalog	4533011
.NET Framework 4.8	Catalog	4533004
Windows Server 2012		
.NET Framework 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2	Catalog	4533010
.NET Framework 4.8	Catalog	4533003
Windows 7 SP1 and Windows Server 2008 R2 SP1		
.NET Framework 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2	Catalog	4533012
.NET Framework 4.8	Catalog	4533005
Windows Server 2008		
.NET Framework 4.6	Catalog	4533012

SameSite cookie sample for ASP.NET 4.7.2 C# MVC

2/20/2020 • 3 minutes to read • Edit Online

.NET Framework 4.7 has built-in support for the SameSite attribute, but it adheres to the original standard. The patched behavior changed the meaning of `SameSite.None` to emit the attribute with a value of `None`, rather than not emit the value at all. If you want to not emit the value you can set the `SameSite` property on a cookie to -1.

Writing the SameSite attribute

Following is an example of how to write a SameSite attribute on a cookie;

```
// Create the cookie
HttpCookie sameSiteCookie = new HttpCookie("SameSiteSample");

// Set a value for the cookieSite none.
// Note this will also require you to be running on HTTPS
sameSiteCookie.Value = "sample";

// Set the secure flag, which Chrome's changes will require for Same
sameSiteCookie.Secure = true;

// Set the cookie to HTTP only which is good practice unless you really do need
// to access it client side in scripts.
sameSiteCookie.HttpOnly = true;

// Add the SameSite attribute, this will emit the attribute with a value of none.
// To not emit the attribute at all set the SameSite property to -1.
sameSiteCookie.SameSite = SameSiteMode.None;

// Add the cookie to the response cookie collection
Response.Cookies.Add(sameSiteCookie);
```

If you are reading this in a language other than English, let us know in this GitHub discussion issue if you'd like to see the code comments in your native language.

The default sameSite attribute for session state is set in the 'cookieSameSite' parameter of the session settings in `web.config`

```
<system.web>
  <sessionState cookieSameSite="None">
  </sessionState>
</system.web>
```

MVC Authentication

OWIN MVC cookie based authentication uses a cookie manager to enable the changing of cookie attributes. The SameSiteCookieManager.cs is an implementation of such a class which you can copy into your own projects.

You must ensure your Microsoft.Owin components are all upgraded to version 4.1.0 or greater. Check your `packages.config` file to ensure all the version numbers match, for example.

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <!-- other packages -->
  <package id="Microsoft.Owin.Host.SystemWeb" version="4.1.0" targetFramework="net472" />
  <package id="Microsoft.Owin.Security" version="4.1.0" targetFramework="net472" />
  <package id="Microsoft.Owin.Security.Cookies" version="4.1.0" targetFramework="net472" />
  <package id="Microsoft.Web.Infrastructure" version="1.0.0.0" targetFramework="net472" />
  <package id="Owin" version="1.0" targetFramework="net472" />
</packages>
```

The authentication components must then be configured to use the CookieManager in your startup class;

```
public void Configuration(IAppBuilder app)
{
    app.UseCookieAuthentication(new CookieAuthenticationOptions
    {
        CookieSameSite = SameSiteMode.None,
        CookieHttpOnly = true,
        CookieSecure = CookieSecureOption.Always,
        CookieManager = new SameSiteCookieManager(new SystemWebCookieManager())
    });
}
```

A cookie manager must be set on *each* component that supports it, this includes CookieAuthentication and OpenIdConnectAuthentication.

The SystemWebCookieManager is used to avoid known issues with response cookie integration.

Running the sample

If you run the sample project load your browser debugger on the initial page and use it to view the cookie collection for the site. To do so in Edge and Chrome press `F12` then select the `Application` tab and click the site URL under the `Cookies` option in the `Storage` section.

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite
.ASPXAUTH	authentication	localhost	/	Session	23	✓	✓	None
sampleCookie	sample	localhost	/	Session	18	✓	✓	Lax

You can see from the image above that the cookie created by the sample when you click the "Create Cookies" button has a SameSite attribute value of `Lax`, matching the value set in the sample code.

Intercepting cookies you do not control

.NET 4.5.2 introduced a new event for intercepting the writing of headers, `Response.AddOnSendingHeaders`. This can be used to intercept cookies before they are returned to the client machine. In the sample we wire up the event to a static method which checks whether the browser supports the new sameSite changes, and if not, changes the cookies to not emit the attribute if the new `None` value has been set.

See global.asax for an example of hooking up the event and SameSiteCookieRewriter.cs for an example of handling the event and adjusting the cookie `sameSite` attribute which you can copy into your own code.

```
public static void FilterSameSiteNoneForIncompatibleUserAgents(object sender)
{
    HttpApplication application = sender as HttpApplication;
    if (application != null)
    {
        var userAgent = application.Context.Request.UserAgent;
        if (SameSite.BrowserDetection.DisallowsSameSiteNone(userAgent))
        {
            HttpContext.Current.Response.AddOnSendingHeaders(context =>
            {
                var cookies = context.Response.Cookies;
                for (var i = 0; i < cookies.Count; i++)
                {
                    var cookie = cookies[i];
                    if (cookie.SameSite == SameSiteMode.None)
                    {
                        cookie.SameSite = (SameSiteMode)(-1); // Unspecified
                    }
                }
            });
        }
    }
}
```

You can change specific named cookie behavior in much the same way; the sample below adjust the default authentication cookie from `Lax` to `None` on browsers which support the `None` value, or removes the sameSite attribute on browsers which do not support `None`.

```
public static void AdjustSpecificCookieSettings()
{
    HttpContext.Current.Response.AddOnSendingHeaders(context =>
    {
        var cookies = context.Response.Cookies;
        for (var i = 0; i < cookies.Count; i++)
        {
            var cookie = cookies[i];
            // Forms auth: ".ASPXAUTH"
            // Session: "ASP.NET_SessionId"
            if (string.Equals(".ASPXAUTH", cookie.Name, StringComparison.Ordinal))
            {
                if (SameSite.BrowserDetection.DisallowsSameSiteNone(userAgent))
                {
                    cookie.SameSite = -1;
                }
                else
                {
                    cookie.SameSite = SameSiteMode.None;
                }
                cookie.Secure = true;
            }
        }
    });
}
```

More Information

Chrome Updates

OWIN SameSite Documentation

ASP.NET Documentation

.NET SameSite Patches

SameSite cookie sample for ASP.NET 4.7.2 C# WebForms

2/20/2020 • 3 minutes to read • Edit Online

.NET Framework 4.7 has built-in support for the SameSite attribute, but it adheres to the original standard. The patched behavior changed the meaning of `SameSite.None` to emit the attribute with a value of `None`, rather than not emit the value at all. If you want to not emit the value you can set the `SameSite` property on a cookie to -1.

Writing the SameSite attribute

Following is an example of how to write a SameSite attribute on a cookie;

```
// Create the cookie
HttpCookie sameSiteCookie = new HttpCookie("SameSiteSample");

// Set a value for the cookie
sameSiteCookie.Value = "sample";

// Set the secure flag, which Chrome's changes will require for SameSite none.
// Note this will also require you to be running on HTTPS
sameSiteCookie.Secure = true;

// Set the cookie to HTTP only which is good practice unless you really do need
// to access it client side in scripts.
sameSiteCookie.HttpOnly = true;

// Add the SameSite attribute, this will emit the attribute with a value of none.
// To not emit the attribute at all set the SameSite property to -1.
sameSiteCookie.SameSite = SameSiteMode.None;

// Add the cookie to the response cookie collection
Response.Cookies.Add(sameSiteCookie);
```

If you are reading this in a language other than English, let us know in this GitHub discussion issue if you'd like to see the code comments in your native language.

The default sameSite attribute for a forms authentication cookie is set in the `cookieSameSite` parameter of the forms authentication settings in `web.config`

```
<system.web>
  <authentication mode="Forms">
    <forms name=".ASPXAUTH" loginUrl="~/" cookieSameSite="None" requireSSL="true">
    </forms>
  </authentication>
</system.web>
```

The default sameSite attribute for session state is also set in the 'cookieSameSite' parameter of the session settings in `web.config`

```
<system.web>
  <sessionState cookieSameSite="None">
  </sessionState>
</system.web>
```

The November 2019 update to .NET changed the default settings for Forms Authentication and Session to `lax` as is the most compatible setting, however if you embed pages into iframes you may need to revert this setting to None, and then add the interception code shown below to adjust the `none` behavior depending on browser capability.

Running the sample

If you run the sample project load your browser debugger on the initial page and use it to view the cookie collection for the site. To do so in Edge and Chrome press `F12` then select the `Application` tab and click the site URL under the `Cookies` option in the `Storage` section.

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite
.ASPXAUTH	authentication	localhost	/	Session	23	✓	✓	None
sampleCookie	sample	localhost	/	Session	18	✓	✓	Lax

You can see from the image above that the cookie created by the sample when you click the "Create Cookies" button has a SameSite attribute value of `Lax`, matching the value set in the sample code.

Intercepting cookies you do not control

.NET 4.5.2 introduced a new event for intercepting the writing of headers, `Response.AddOnSendingHeaders`. This can be used to intercept cookies before they are returned to the client machine. In the sample we wire up the event to a static method which checks whether the browser supports the new sameSite changes, and if not, changes the cookies to not emit the attribute if the new `None` value has been set.

See global.asax for an example of hooking up the event and SameSiteCookieRewriter.cs for an example of handling the event and adjusting the cookie `sameSite` attribute.

```
public static void FilterSameSiteNoneForIncompatibleUserAgents(object sender)
{
    HttpApplication application = sender as HttpApplication;
    if (application != null)
    {
        var userAgent = application.Context.Request.UserAgent;
        if (SameSite.BrowserDetection.DisallowsSameSiteNone(userAgent))
        {
            HttpContext.Current.Response.AddOnSendingHeaders(context =>
            {
                var cookies = context.Response.Cookies;
                for (var i = 0; i < cookies.Count; i++)
                {
                    var cookie = cookies[i];
                    if (cookie.SameSite == SameSiteMode.None)
                    {
                        cookie.SameSite = (SameSiteMode)(-1); // Unspecified
                    }
                }
            });
        }
    }
}
```

You can change specific named cookie behavior in much the same way; the sample below adjust the default authentication cookie from `Lax` to `None` on browsers which support the `None` value, or removes the sameSite attribute on browsers which do not support `None`.

```
public static void AdjustSpecificCookieSettings()
{
    HttpContext.Current.Response.AddOnSendingHeaders(context =>
    {
        var cookies = context.Response.Cookies;
        for (var i = 0; i < cookies.Count; i++)
        {
            var cookie = cookies[i];
            // Forms auth: ".ASPXAUTH"
            // Session: "ASP.NET_SessionId"
            if (string.Equals(".ASPXAUTH", cookie.Name, StringComparison.Ordinal))
            {
                if (SameSite.BrowserDetection.DisallowsSameSiteNone(userAgent))
                {
                    cookie.SameSite = -1;
                }
                else
                {
                    cookie.SameSite = SameSiteMode.None;
                }
                cookie.Secure = true;
            }
        }
    });
}
```

More Information

Chrome Updates

ASP.NET Documentation

.NET SameSite Patches

SameSite cookie sample for ASP.NET 4.7.2 VB WebForms

2/20/2020 • 3 minutes to read • Edit Online

.NET Framework 4.7 has built-in support for the SameSite attribute, but it adheres to the original standard. The patched behavior changed the meaning of `SameSite.None` to emit the attribute with a value of `None`, rather than not emit the value at all. If you want to not emit the value you can set the `SameSite` property on a cookie to -1.

Writing the SameSite attribute

Following is an example of how to write a SameSite attribute on a cookie;

```
' Create the cookie
Dim sameSiteCookie As New HttpCookie("sameSiteSample")

' Set a value for the cookie
sameSiteCookie.Value = "sample"

' Set the secure flag, which Chrome's changes will require for SameSite none.
' Note this will also require you to be running on HTTPS
sameSiteCookie.Secure = True

' Set the cookie to HTTP only which is good practice unless you really do need
' to access it client side in scripts.
sameSiteCookie.HttpOnly = True

' Expire the cookie in 1 minute
sameSiteCookie.Expires = Date.Now.AddMinutes(1)

' Add the SameSite attribute, this will emit the attribute with a value of none.
' To Not emit the attribute at all set the SameSite property to -1.
sameSiteCookie.SameSite = SameSiteMode.None

' Add the cookie to the response cookie collection
Response.Cookies.Add(sameSiteCookie)
```

If you are reading this in a language other than English, let us know in this GitHub discussion issue if you'd like to see the code comments in your native language.

The default sameSite attribute for a forms authentication cookie is set in the `cookieSameSite` parameter of the forms authentication settings in `web.config`

```
<system.web>
  <authentication mode="Forms">
    <forms name=".ASPXAUTH" loginUrl="~/" cookieSameSite="None" requireSSL="true">
    </forms>
  </authentication>
</system.web>
```

The default sameSite attribute for session state is also set in the 'cookieSameSite' parameter of the session settings in `web.config`

```
<system.web>
  <sessionState cookieSameSite="None">
  </sessionState>
</system.web>
```

The November 2019 update to .NET changed the default settings for Forms Authentication and Session to `lax` as is the most compatible setting, however if you embed pages into iframes you may need to revert this setting to None, and then add the interception code shown below to adjust the `none` behavior depending on browser capability.

Running the sample

If you run the sample project load your browser debugger on the initial page and use it to view the cookie collection for the site. To do so in Edge and Chrome press `F12` then select the `Application` tab and click the site URL under the `Cookies` option in the `Storage` section.

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite
.ASPXAUTH	authentication	localhost	/	Session	23	✓	✓	None
sampleCookie	sample	localhost	/	Session	18	✓	✓	Lax

You can see from the image above that the cookie created by the sample when you click the "Create Cookies" button has a SameSite attribute value of `Lax`, matching the value set in the sample code.

Intercepting cookies you do not control

.NET 4.5.2 introduced a new event for intercepting the writing of headers, `Response.AddOnSendingHeaders`. This can be used to intercept cookies before they are returned to the client machine. In the sample we wire up the event to a static method which checks whether the browser supports the new sameSite changes, and if not, changes the cookies to not emit the attribute if the new `None` value has been set.

See global.asax for an example of hooking up the event and SameSiteCookieRewriter.cs for an example of handling the event and adjusting the cookie `sameSite` attribute.

```
Sub FilterSameSiteNoneForIncompatibleUserAgents(ByVal sender As Object)
    Dim application As HttpApplication = TryCast(sender, HttpApplication)

    If application IsNot Nothing Then
        Dim userAgent = application.Context.Request.UserAgent

        If SameSite.DisallowsSameSiteNone(userAgent) Then
            application.Response.AddOnSendingHeaders(
                Function(context)
                    Dim cookies = context.Response.Cookies

                    For i = 0 To cookies.Count - 1
                        Dim cookie = cookies(i)

                        If cookie.SameSite = SameSiteMode.None Then
                            cookie.SameSite = CType((-1), SameSiteMode)
                        End If
                    Next
                End Function)
        End If
    End If
End Sub
```

You can change specific named cookie behavior in much the same way; the sample below adjust the default authentication cookie from `Lax` to `None` on browsers which support the `None` value, or removes the sameSite attribute on browsers which do not support `None`.

```
Public Shared Sub AdjustSpecificCookieSettings()
    HttpContext.Current.Response.AddOnSendingHeaders(Function(context)
            Dim cookies = context.Response.Cookies

            For i = 0 To cookies.Count - 1
            Dim cookie = cookies(i)

            If String.Equals(".ASPXAUTH", cookie.Name, StringComparison.Ordinal) Then

                If SameSite.BrowserDetection.DisallowsSameSiteNone(userAgent) Then
                    cookie.SameSite = -1
                Else
                    cookie.SameSite = SameSiteMode.None
                End If

                cookie.Secure = True
            End If
            Next
        End Function)
End Sub
```

More Information

Chrome Updates

ASP.NET Documentation

.NET SameSite Patches

SameSite cookie sample for ASP.NET 4.7.2 VB MVC

2/20/2020 • 3 minutes to read • Edit Online

.NET Framework 4.7 has built-in support for the SameSite attribute, but it adheres to the original standard. The patched behavior changed the meaning of `SameSite.None` to emit the attribute with a value of `None`, rather than not emit the value at all. If you want to not emit the value you can set the `SameSite` property on a cookie to -1.

Writing the SameSite attribute

Following is an example of how to write a SameSite attribute on a cookie;

```
' Create the cookie
Dim sameSiteCookie As New HttpCookie("sameSiteSample")

' Set a value for the cookie
sameSiteCookie.Value = "sample"

' Set the secure flag, which Chrome's changes will require for SameSite none.
' Note this will also require you to be running on HTTPS
sameSiteCookie.Secure = True

' Set the cookie to HTTP only which is good practice unless you really do need
' to access it client side in scripts.
sameSiteCookie.HttpOnly = True

' Expire the cookie in 1 minute
sameSiteCookie.Expires = Date.Now.AddMinutes(1)

' Add the SameSite attribute, this will emit the attribute with a value of none.
' To Not emit the attribute at all set the SameSite property to -1.
sameSiteCookie.SameSite = SameSiteMode.None

' Add the cookie to the response cookie collection
Response.Cookies.Add(sameSiteCookie)
```

If you are reading this in a language other than English, let us know in this GitHub discussion issue if you'd like to see the code comments in your native language.

The default sameSite attribute for session state is set in the 'cookieSameSite' parameter of the session settings in `web.config`

```
<system.web>
  <sessionState cookieSameSite="None">
  </sessionState>
</system.web>
```

MVC Authentication

OWIN MVC cookie based authentication uses a cookie manager to enable the changing of cookie attributes. The SameSiteCookieManager.vb is an implementation of such a class which you can copy into your own projects.

You must ensure your Microsoft.Owin components are all upgraded to version 4.1.0 or greater. Check your `packages.config` file to ensure all the version numbers match, for example.

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <!-- other packages -->
  <package id="Microsoft.Owin.Host.SystemWeb" version="4.1.0" targetFramework="net472" />
  <package id="Microsoft.Owin.Security" version="4.1.0" targetFramework="net472" />
  <package id="Microsoft.Owin.Security.Cookies" version="4.1.0" targetFramework="net472" />
  <package id="Microsoft.Web.Infrastructure" version="1.0.0.0" targetFramework="net472" />
  <package id="Owin" version="1.0" targetFramework="net472" />
</packages>
```

The authentication components must be configured to use the CookieManager in your startup class;

```
Public Sub Configuration(app As IAppBuilder)
    app.UseCookieAuthentication(New CookieAuthenticationOptions() With {
        .CookieSameSite = SameSiteMode.None,
        .CookieHttpOnly = True,
        .CookieSecure = CookieSecureOption.Always,
        .CookieManager = New SameSiteCookieManager(New SystemWebCookieManager())
    })
End Sub
```

A cookie manager must be set on *each* component that supports it, this includes CookieAuthentication and OpenIdConnectAuthentication.

The SystemWebCookieManager is used to avoid known issues with response cookie integration.

Running the sample

If you run the sample project please load your browser debugger on the initial page and use it to view the cookie collection for the site. To do so in Edge and Chrome press `F12` then select the `Application` tab and click the site URL under the `Cookies` option in the `Storage` section.

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite
.ASPXAUTH	authentication	localhost	/	Session	23	✓	✓	None
sampleCookie	sample	localhost	/	Session	18	✓	✓	Lax

You can see from the image above that the cookie created by the sample when you click the "Create SameSite Cookie" button has a SameSite attribute value of `Lax`, matching the value set in the sample code.

Intercepting cookies you do not control

.NET 4.5.2 introduced a new event for intercepting the writing of headers, `Response.AddOnSendingHeaders`. This can be used to intercept cookies before they are returned to the client machine. In the sample we wire up the event to a static method which checks whether the browser supports the new sameSite changes, and if not, changes the cookies to not emit the attribute if the new `None` value has been set.

See global.asax for an example of hooking up the event and SameSiteCookieRewriter.vb for an example of handling the event and adjusting the cookie `sameSite` attribute which you can copy into your own code.

```
Sub FilterSameSiteNoneForIncompatibleUserAgents(ByVal sender As Object)
    Dim application As HttpApplication = TryCast(sender, HttpApplication)

    If application IsNot Nothing Then
        Dim userAgent = application.Context.Request.UserAgent

        If SameSite.DisallowsSameSiteNone(userAgent) Then
            application.Response.AddOnSendingHeaders(
                Function(context)
                    Dim cookies = context.Response.Cookies

                    For i = 0 To cookies.Count - 1
                        Dim cookie = cookies(i)

                        If cookie.SameSite = SameSiteMode.None Then
                            cookie.SameSite = CType((-1), SameSiteMode)
                        End If
                    Next
                End Function)
        End If
    End If
End Sub
```

You can change specific named cookie behavior in much the same way; the sample below adjust the default authentication cookie from `Lax` to `None` on browsers which support the `None` value, or removes the sameSite attribute on browsers which do not support `None`.

```
Public Shared Sub AdjustSpecificCookieSettings()
    HttpContext.Current.Response.AddOnSendingHeaders(Function(context)
            Dim cookies = context.Response.Cookies

            For i = 0 To cookies.Count - 1
            Dim cookie = cookies(i)

            If String.Equals(".ASPXAUTH", cookie.Name, StringComparison.Ordinal) Then

                If SameSite.BrowserDetection.DisallowsSameSiteNone(userAgent) Then
                    cookie.SameSite = -1
                Else
                    cookie.SameSite = SameSiteMode.None
                End If

                cookie.Secure = True
            End If
            Next
        End Function)
End Sub
```

More Information

Chrome Updates

OWIN SameSite Documentation

ASP.NET Documentation

.NET SameSite Patches

ASP.NET 4 and Visual Studio 2010 Web Development Overview

2/20/2020 • 2 minutes to read • Edit Online

- ASP.NET 4 and Visual Studio 2010 Web Development Overview
- ASP.NET 4 Breaking Changes

ASP.NET 4 and Visual Studio 2010 Web Development Overview

2/20/2020 • 80 minutes to read • Edit Online

This document provides an overview of many of the new features for ASP.NET that are included in the.NET Framework 4 and in Visual Studio 2010.

Download This Whitepaper

Contents

Core Services

Web.config File Refactoring
Extensible Output Caching
Auto-Start Web Applications
Permanently Redirecting a Page
Shrinking Session State
Expanding the Range of Allowable URLs
Extensible Request Validation
Object Caching and Object Caching Extensibility
Extensible HTML, URL, and HTTP Header Encoding
Performance Monitoring for Individual Applications in a Single Worker Process
Multi-Targeting

Ajax

jQuery Included with Web Forms and MVC
Content Delivery Network Support
ScriptManager Explicit Scripts

Web Forms

Setting Meta Tags with the Page.MetaKeywords and Page.MetaDescription Properties
Enabling View State for Individual Controls
Changes to Browser Capabilities
Routing in ASP.NET 4
Setting Client IDs
Persisting Row Selection in Data Controls
ASP.NET Chart Control
Filtering Data with the QueryExtender Control
Html Encoded Code Expressions
Project Template Changes
CSS Improvements
Hiding div Elements Around Hidden Fields
Rendering an Outer Table for Templated Controls
ListView Control Enhancements
CheckBoxList and RadioButtonList Control Enhancements
Menu Control Improvements
Wizard and CreateUserWizard Controls 56

ASP.NET MVC

Areas Support
Data-Annotation Attribute Validation Support
Templated Helpers

Dynamic Data
Enabling Dynamic Data for Existing Projects
Declarative DynamicDataManager Control Syntax
Entity Templates
New Field Templates for URLs and E-mail Addresses
Creating Links with the DynamicHyperLink Control
Support for Inheritance in the Data Model
Support for Many-to-Many Relationships (Entity Framework Only)
New Attributes to Control Display and Support Enumerations
Enhanced Support for Filters

Visual Studio 2010 Web Development Improvements
Improved CSS Compatibility
HTML and JavaScript Snippets
JavaScript IntelliSense Enhancements

Web Application Deployment with Visual Studio 2010
Web Packaging
Web.config Transformation
Database Deployment
One-Click Publish for Web Applications
Resources

Disclaimer

Core Services

ASP.NET 4 introduces a number of features that improve core ASP.NET services such as output caching and session-state storage.

Web.config File Refactoring

The `Web.config` file that contains the configuration for a Web application has grown considerably over the past few releases of the .NET Framework as new features have been added, such as Ajax, routing, and integration with IIS 7. This has made it harder to configure or start new Web applications without a tool like Visual Studio. In .the NET Framework 4, the major configuration elements have been moved to the `machine.config` file, and applications now inherit these settings. This allows the `Web.config` file in ASP.NET 4 applications either to be empty or to contain just the following lines, which specify for Visual Studio what version of the framework the application is targeting:

```
<?xml version="1.0"?>
  <configuration>
   <system.web>
    <compilation targetFramework="4.0" />
   </system.web>
  </configuration>
```

Extensible Output Caching

Since the time that ASP.NET 1.0 was released, output caching has enabled developers to store the generated output of pages, controls, and HTTP responses in memory. On subsequent Web requests, ASP.NET can serve content more quickly by retrieving the generated output from memory instead of regenerating the output from scratch. However, this approach has a limitation — generated content always has to be stored in memory, and on servers

that are experiencing heavy traffic, the memory consumed by output caching can compete with memory demands from other portions of a Web application.

ASP.NET 4 adds an extensibility point to output caching that enables you to configure one or more custom output-cache providers. Output-cache providers can use any storage mechanism to persist HTML content. This makes it possible to create custom output-cache providers for diverse persistence mechanisms, which can include local or remote disks, cloud storage, and distributed cache engines.

You create a custom output-cache provider as a class that derives from the new *System.Web.Caching.OutputCacheProvider* type. You can then configure the provider in the `Web.config` file by using the new *providers* subsection of the *outputCache* element, as shown in the following example:

```
<caching>
  <outputCache defaultProvider="AspNetInternalProvider">
    <providers>
      <add name="DiskCache"
          type="Test.OutputCacheEx.DiskOutputCacheProvider, DiskCacheProvider"/>
    </providers>
  </outputCache>

</caching>
```

By default in ASP.NET 4, all HTTP responses, rendered pages, and controls use the in-memory output cache, as shown in the previous example, where the *defaultProvider* attribute is set to AspNetInternalProvider. You can change the default output-cache provider used for a Web application by specifying a different provider name for *defaultProvider*.

In addition, you can select different output-cache providers per control and per request. The easiest way to choose a different output-cache provider for different Web user controls is to do so declaratively by using the new *providerName* attribute in a control directive, as shown in the following example:

```
<%@ OutputCache Duration="60" VaryByParam="None" providerName="DiskCache" %>
```

Specifying a different output cache provider for an HTTP request requires a little more work. Instead of declaratively specifying the provider, you override the new *GetOuputCacheProviderName* method in the `Global.asax` file to programmatically specify which provider to use for a specific request. The following example shows how to do this.

```
public override string GetOutputCacheProviderName(HttpContext context)
{
    if (context.Request.Path.EndsWith("Advanced.aspx"))
       return "DiskCache";
    else
        return base.GetOutputCacheProviderName(context);
}
```

With the addition of output-cache provider extensibility to ASP.NET 4, you can now pursue more aggressive and more intelligent output-caching strategies for your Web sites. For example, it is now possible to cache the "Top 10" pages of a site in memory, while caching pages that get lower traffic on disk. Alternatively, you can cache every vary-by combination for a rendered page, but use a distributed cache so that the memory consumption is offloaded from front-end Web servers.

Auto-Start Web Applications

Some Web applications need to load large amounts of data or perform expensive initialization processing before serving the first request. In earlier versions of ASP.NET, for these situations you had to devise custom approaches to "wake up" an ASP.NET application and then run initialization code during the *Application_Load* method in the

`Global.asax` file.

A new scalability feature named *auto-start* that directly addresses this scenario is available when ASP.NET 4 runs on IIS 7.5 on Windows Server 2008 R2. The auto-start feature provides a controlled approach for starting up an application pool, initializing an ASP.NET application, and then accepting HTTP requests.

> **NOTE**
>
> IIS Application Warm-Up Module for IIS 7.5
>
> The IIS team has released the first beta test version of the Application Warm-Up Module for IIS 7.5. This makes warming up your applications even easier than previously described. Instead of writing custom code, you specify the URLs of resources to execute before the Web application accepts requests from the network. This warm-up occurs during startup of the IIS service (if you configured the IIS application pool as *AlwaysRunning*) and when an IIS worker process recycles. During recycle, the old IIS worker process continues to execute requests until the newly spawned worker process is fully warmed up, so that applications experience no interruptions or other issues due to unprimed caches. Note that this module works with any version of ASP.NET, starting with version 2.0.
>
> For more information, see Application Warm-Up on the IIS.net Web site. For a walkthrough that illustrates how to use the warm-up feature, see Getting Started with the IIS 7.5 Application Warm-Up Module on the IIS.net Web site.

To use the auto-start feature, an IIS administrator sets an application pool in IIS 7.5 to be automatically started by using the following configuration in the `applicationHost.config` file:

```
<applicationpools>
  <add name="MyApplicationPool" startMode="AlwaysRunning" />
</applicationpools>
```

Because a single application pool can contain multiple applications, you specify individual applications to be automatically started by using the following configuration in the `applicationHost.config` file:

```
<sites>
  <site name="MySite" id="1">
    <application path="/"
      serviceAutoStartEnabled="true"
      serviceAutoStartProvider="PrewarmMyCache" >
      <!-- Additional content -->
    </application>
  </site>

</sites>

<!-- Additional content -->

<serviceautostartproviders>
  <add name="PrewarmMyCache"
    type="MyNamespace.CustomInitialization, MyLibrary" />
</serviceautostartproviders>
```

When an IIS 7.5 server is cold-started or when an individual application pool is recycled, IIS 7.5 uses the information in the `applicationHost.config` file to determine which Web applications need to be automatically started. For each application that is marked for auto-start, IIS7.5 sends a request to ASP.NET 4 to start the application in a state during which the application temporarily does not accept HTTP requests. When it is in this state, ASP.NET instantiates the type defined by the *serviceAutoStartProvider* attribute (as shown in the previous example) and calls into its public entry point.

You create a managed auto-start type with the necessary entry point by implementing the *IProcessHostPreloadClient* interface, as shown in the following example:

```
public class CustomInitialization : System.Web.Hosting.IProcessHostPreloadClient
{
    public void Preload(string[] parameters)
    {
        // Perform initialization.
    }
}
```

After your initialization code runs in the *Preload* method and the method returns, the ASP.NET application is ready to process requests.

With the addition of auto-start to IIS .5 and ASP.NET 4, you now have a well-defined approach for performing expensive application initialization prior to processing the first HTTP request. For example, you can use the new auto-start feature to initialize an application and then signal a load-balancer that the application was initialized and ready to accept HTTP traffic.

Permanently Redirecting a Page

It is common practice in Web applications to move pages and other content around over time, which can lead to an accumulation of stale links in search engines. In ASP.NET, developers have traditionally handled requests to old URLs by using by using the *Response.Redirect* method to forward a request to the new URL. However, the *Redirect* method issues an HTTP 302 Found (temporary redirect) response, which results in an extra HTTP round trip when users attempt to access the old URLs.

ASP.NET 4 adds a new *RedirectPermanent* helper method that makes it easy to issue HTTP 301 Moved Permanently responses, as in the following example:

```
RedirectPermanent("/newpath/foroldcontent.aspx");
```

Search engines and other user agents that recognize permanent redirects will store the new URL that is associated with the content, which eliminates the unnecessary round trip made by the browser for temporary redirects.

Shrinking Session State

ASP.NET provides two default options for storing session state across a Web farm: a session-state provider that invokes an out-of-process session-state server, and a session-state provider that stores data in a Microsoft SQL Server database. Because both options involve storing state information outside a Web application's worker process, session state has to be serialized before it is sent to remote storage. Depending on how much information a developer saves in session state, the size of the serialized data can grow quite large.

ASP.NET 4 introduces a new compression option for both kinds of out-of-process session-state providers. When the *compressionEnabled* configuration option shown in the following example is set to *true*, ASP.NET will compress (and decompress) serialized session state by using the .NET Framework *System.IO.Compression.GZipStream* class.

```
<sessionState
  mode="SqlServer"
  sqlConnectionString="data source=dbserver;Initial Catalog=aspnetstate"
  allowCustomSqlDatabase="true"
  compressionEnabled="true"
/>
```

With the simple addition of the new attribute to the `Web.config` file, applications with spare CPU cycles on Web servers can realize substantial reductions in the size of serialized session-state data.

Expanding the Range of Allowable URLs

ASP.NET 4 introduces new options for expanding the size of application URLs. Previous versions of ASP.NET constrained URL path lengths to 260 characters, based on the NTFS file-path limit. In ASP.NET 4, you have the

option to increase (or decrease) this limit as appropriate for your applications, using two new *httpRuntime* configuration attributes. The following example shows these new attributes.

```
<httpRuntime maxUrlLength="260" maxQueryStringLength="2048" />
```

To allow longer or shorter paths (the portion of the URL that does not include protocol, server name, and query string), modify the *maxUrlLength* attribute. To allow longer or shorter query strings, modify the value of the *maxQueryStringLength* attribute.

ASP.NET 4 also enables you to configure the characters that are used by the URL character check. When ASP.NET finds an invalid character in the path portion of a URL, it rejects the request and issues an HTTP 400 error. In previous versions of ASP.NET, the URL character checks were limited to a fixed set of characters. In ASP.NET 4, you can customize the set of valid characters using the new *requestPathInvalidCharacters* attribute of the *httpRuntime* configuration element, as shown in the following example:

```
<httpRuntime requestPathInvalidCharacters="&lt;,&gt;,*,%,&amp;,:,\,?"  />
```

By default, the *requestPathInvalidCharacters* attribute defines eight characters as invalid. (In the string that is assigned to *requestPathInvalidCharacters* by default, the less than (<), greater than (>), and ampersand (&) characters are encoded, because the `Web.config` file is an XML file.) You can customize the set of invalid characters as needed.

> **NOTE**
>
> Note ASP.NET 4 always rejects URL paths that contain characters in the ASCII range of 0x00 to 0x1F, because those are invalid URL characters as defined in RFC 2396 of the IETF (http://www.ietf.org/rfc/rfc2396.txt). On versions of Windows Server that run IIS 6 or higher, the http.sys protocol device driver automatically rejects URLs with these characters.

Extensible Request Validation

ASP.NET request validation searches incoming HTTP request data for strings that are commonly used in cross-site scripting (XSS) attacks. If potential XSS strings are found, request validation flags the suspect string and returns an error. The built-in request validation returns an error only when it finds the most common strings used in XSS attacks. Previous attempts to make the XSS validation more aggressive resulted in too many false positives. However, customers might want request validation that is more aggressive, or conversely might want to intentionally relax XSS checks for specific pages or for specific types of requests.

In ASP.NET 4, the request validation feature has been made extensible so that you can use custom request-validation logic. To extend request validation, you create a class that derives from the new *System.Web.Util.RequestValidator* type, and you configure the application (in the *httpRuntime* section of the `Web.config` file) to use the custom type. The following example shows how to configure a custom request-validation class:

```
<httpRuntime requestValidationType="Samples.MyValidator, Samples" />
```

The new *requestValidationType* attribute requires a standard .NET Framework type identifier string that specifies the class that provides custom request validation. For each request, ASP.NET invokes the custom type to process each piece of incoming HTTP request data. The incoming URL, all the HTTP headers (both cookies and custom headers), and the entity body are all available for inspection by a custom request validation class like that shown in the following example:

```
public class CustomRequestValidation : RequestValidator
{
    protected override bool IsValidRequestString(
      HttpContext context, string value,
      RequestValidationSource requestValidationSource,
      string collectionKey,
      out int validationFailureIndex)
    {...}
 }
```

For cases where you do not want to inspect a piece of incoming HTTP data, the request-validation class can fall back to let the ASP.NET default request validation run by simply calling *base.IsValidRequestString*.

Object Caching and Object Caching Extensibility

Since its first release, ASP.NET has included a powerful in-memory object cache (*System.Web.Caching.Cache*). The cache implementation has been so popular that it has been used in non-Web applications. However, it is awkward for a Windows Forms or WPF application to include a reference to `System.Web.dll` just to be able to use the ASP.NET object cache.

To make caching available for all applications, the .NET Framework 4 introduces a new assembly, a new namespace, some base types, and a concrete caching implementation. The new `System.Runtime.Caching.dll` assembly contains a new caching API in the *System.Runtime.Caching* namespace. The namespace contains two core sets of classes:

- Abstract types that provide the foundation for building any type of custom cache implementation.
- A concrete in-memory object cache implementation (the *System.Runtime.Caching.MemoryCache* class).

The new *MemoryCache* class is modeled closely on the ASP.NET cache, and it shares much of the internal cache engine logic with ASP.NET. Although the public caching APIs in *System.Runtime.Caching* have been updated to support development of custom caches, if you have used the ASP.NET *Cache* object, you will find familiar concepts in the new APIs.

An in-depth discussion of the new *MemoryCache* class and supporting base APIs would require an entire document. However, the following example gives you an idea of how the new cache API works. The example was written for a Windows Forms application, without any dependency on `System.Web.dll`.

```
private void btnGet_Click(object sender, EventArgs e)
{
    //Obtain a reference to the default MemoryCache instance.
    //Note that you can create multiple MemoryCache(s) inside
    //of a single application.
    ObjectCache cache = MemoryCache.Default;

    //In this example the cache is storing the contents of a file string
    fileContents = cache["filecontents"] as string;

    //If the file contents are not currently in the cache, then
    //the contents are read from disk and placed in the cache.
    if (fileContents == null)
    {
        //A CacheItemPolicy object holds all the pieces of cache
        //dependency and cache expiration metadata related to a single
        //cache entry.
        CacheItemPolicy policy = new CacheItemPolicy();

        //Build up the information necessary to create a file dependency.
        //In this case we just need the file path of the file on disk.
        List filePaths = new List();
        filePaths.Add("c:\\data.txt");

        //In the new cache API, dependencies are called "change monitors".
        //For this example we want the cache entry to be automatically expired
        //if the contents on disk change. A HostFileChangeMonitor provides
        //this functionality.
        policy.ChangeMonitors.Add(new HostFileChangeMonitor(filePaths));

        //Fetch the file's contents
        fileContents = File.ReadAllText("c:\\data.txt");

        //And then store the file's contents in the cache
        cache.Set("filecontents", fileContents, policy);

    }
    MessageBox.Show(fileContents);
}
```

Extensible HTML, URL, and HTTP Header Encoding

In ASP.NET 4, you can create custom encoding routines for the following common text-encoding tasks:

- HTML encoding.
- URL encoding.
- HTML attribute encoding.
- Encoding outbound HTTP headers.

You can create a custom encoder by deriving from the new *System.Web.Util.HttpEncoder* type and then configuring ASP.NET to use the custom type in the *httpRuntime* section of the `Web.config` file, as shown in the following example:

```
<httpRuntime encoderType="Samples.MyCustomEncoder, Samples" />
```

After a custom encoder has been configured, ASP.NET automatically calls the custom encoding implementation whenever public encoding methods of the *System.Web.HttpUtility* or *System.Web.HttpServerUtility* classes are called. This lets one part of a Web development team create a custom encoder that implements aggressive character encoding, while the rest of the Web development team continues to use the public ASP.NET encoding APIs. By centrally configuring a custom encoder in the *httpRuntime* element, you are guaranteed that all text-encoding calls from the public ASP.NET encoding APIs are routed through the custom encoder.

Performance Monitoring for Individual Applications in a Single Worker Process

In order to increase the number of Web sites that can be hosted on a single server, many hosters run multiple ASP.NET applications in a single worker process. However, if multiple applications use a single shared worker process, it is difficult for server administrators to identify an individual application that is experiencing problems.

ASP.NET 4 leverages new resource-monitoring functionality introduced by the CLR. To enable this functionality, you can add the following XML configuration snippet to the `aspnet.config` configuration file.

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
  <runtime>
    <appDomainResourceMonitoring enabled="true"/>
  </runtime>

</configuration>
```

> **NOTE**
>
> Note The `aspnet.config` file is in the directory where the .NET Framework is installed. It is not the `Web.config` file.

When the *appDomainResourceMonitoring* feature has been enabled, two new performance counters are available in the "ASP.NET Applications" performance category: *% Managed Processor Time* and *Managed Memory Used*. Both of these performance counters use the new CLR application-domain resource management feature to track estimated CPU time and managed memory utilization of individual ASP.NET applications. As a result, with ASP.NET 4, administrators now have a more granular view into the resource consumption of individual applications running in a single worker process.

Multi-Targeting

You can create an application that targets a specific version of the .NET Framework. In ASP.NET 4, a new attribute in the *compilation* element of the `Web.config` file lets you target the .NET Framework 4 and later. If you explicitly target the .NET Framework 4, and if you include optional elements in the `Web.config` file such as the entries for *system.codedom*, these elements must be correct for the .NET Framework 4. (If you do not explicitly target the .NET Framework 4, the target framework is inferred from the lack of an entry in the `Web.config` file.)

The following example shows the use of the *targetFramework* attribute in the *compilation* element of the `Web.config` file.

```
<compilation targetFramework="4.0"/>
```

Note the following about targeting a specific version of the .NET Framework:

- In a .NET Framework 4 application pool, the ASP.NET build system assumes the .NET Framework 4 as a target if the `Web.config` file does not include the *targetFramework* attribute or if the `Web.config` file is missing. (You might have to make coding changes to your application to make it run under the .NET Framework 4.)
- If you do include the *targetFramework* attribute, and if the *system.codeDom* element is defined in the `Web.config` file, this file must contain the correct entries for the .NET Framework 4.
- If you are using the *aspnet_compiler* command to precompile your application (such as in a build environment), you must use the correct version of the *aspnet_compiler* command for the target framework. Use the compiler that shipped with the .NET Framework 2.0 (%WINDIR%\Microsoft.NET\Framework\v2.0.50727) to compile for the .NET Framework 3.5 and earlier versions. Use the compiler that ships with the .NET Framework 4 to compile applications created using that framework or using later versions.
- At run time, the compiler uses the latest framework assemblies that are installed on the computer (and therefore in the GAC). If an update is made later to the framework (for example, a hypothetical version 4.1 is

installed), you will be able to use features in the newer version of the framework even though the *targetFramework* attribute targets a lower version (such as 4.0). (However, at design time in Visual Studio 2010 or when you use the *aspnet_compiler* command, using newer features of the framework will cause compiler errors).

Ajax

jQuery Included with Web Forms and MVC

The Visual Studio templates for both Web Forms and MVC include the open-source jQuery library. When you create a new website or project, a Scripts folder containing the following 3 files is created:

- jQuery-1.4.1.js – The human-readable, unminified version of the jQuery library.
- jQuery-14.1.min.js – The minified version of the jQuery library.
- jQuery-1.4.1-vsdoc.js – The Intellisense documentation file for the jQuery library.

Include the unminified version of jQuery while developing an application. Include the minified version of jQuery for production applications.

For example, the following Web Forms page illustrates how you can use jQuery to change the background color of ASP.NET TextBox controls to yellow when they have focus.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="ShowjQuery.aspx.cs" Inherits="ShowjQuery" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head runat="server">
    <title>Show jQuery</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        <asp:TextBox ID="txtFirstName" runat="server" />
        <br />
        <asp:TextBox ID="txtLastName" runat="server" />
    </div>
    </form>
    <script src="Scripts/jquery-1.4.1.js" type="text/javascript"></script>

    <script type="text/javascript">

        $("input").focus( function() { $(this).css("background-color", "yellow"); });

    </script>
</body>
</html>
```

Content Delivery Network Support

The Microsoft Ajax Content Delivery Network (CDN) enables you to easily add ASP.NET Ajax and jQuery scripts to your Web applications. For example, you can start using the jQuery library simply by adding a `<script>` tag to your page that points to Ajax.microsoft.com like this:

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-1.4.2.js" type="text/javascript"></script>
```

By taking advantage of the Microsoft Ajax CDN, you can significantly improve the performance of your Ajax applications. The contents of the Microsoft Ajax CDN are cached on servers located around the world. In addition, the Microsoft Ajax CDN enables browsers to reuse cached JavaScript files for Web sites that are located in different

domains.

The Microsoft Ajax Content Delivery Network supports SSL (HTTPS) in case you need to serve a web page using the Secure Sockets Layer.

Implement a fallback when the CDN is unavailable. Test the fallback.

To learn more about the Microsoft Ajax CDN, visit the following website:

https://www.asp.net/ajaxlibrary/CDN.ashx

The ASP.NET ScriptManager supports the Microsoft Ajax CDN. Simply by setting one property, the EnableCdn property, you can retrieve all ASP.NET framework JavaScript files from the CDN:

```
<asp:ScriptManager ID="sm1" EnableCdn="true" runat="server" />
```

After you set the EnableCdn property to the value true, the ASP.NET framework will retrieve all ASP.NET framework JavaScript files from the CDN including all JavaScript files used for validation and the UpdatePanel. Setting this one property can have a dramatic impact on the performance of your web application.

You can set the CDN path for your own JavaScript files by using the WebResource attribute. The new CdnPath property specifies the path to the CDN used when you set the EnableCdn property to the value true:

```
[assembly: WebResource("Foo.js", "application/x-javascript", CdnPath = "http://foo.com/foo/bar/foo.js")]
```

ScriptManager Explicit Scripts

In the past, if you used the ASP.NET ScriptManger then you were required to load the entire monolithic ASP.NET Ajax Library. By taking advantage of the new ScriptManager.AjaxFrameworkMode property, you can control exactly which components of the ASP.NET Ajax Library are loaded and load only the components of the ASP.NET Ajax Library that you need.

The ScriptManager.AjaxFrameworkMode property can be set to the following values:

- Enabled -- Specifies that the ScriptManager control automatically includes the MicrosoftAjax.js script file, which is a combined script file of every core framework script (legacy behavior).
- Disabled -- Specifies that all Microsoft Ajax script features are disabled and that the ScriptManager control does not reference any scripts automatically.
- Explicit -- Specifies that you will explicitly include script references to individual framework core script file that your page requires, and that you will include references to the dependencies that each script file requires.

For example, if you set the AjaxFrameworkMode property to the value Explicit then you can specify the particular ASP.NET Ajax component scripts that you need:

```
<asp:ScriptManager ID="sm1" AjaxFrameworkMode="Explicit" runat="server">

<Scripts>
    <asp:ScriptReference Name="MicrosoftAjaxCore.js" />
    <asp:ScriptReference Name="MicrosoftAjaxComponentModel.js" />
    <asp:ScriptReference Name="MicrosoftAjaxSerialization.js" />
    <asp:ScriptReference Name="MicrosoftAjaxNetwork.js" />
</Scripts>
</asp:ScriptManager>
```

Web Forms

Web Forms has been a core feature in ASP.NET since the release of ASP.NET 1.0. Many enhancements have been in

this area for ASP.NET 4, including the following:

- The ability to set *meta* tags.
- More control over view state.
- Easier ways to work with browser capabilities.
- Support for using ASP.NET routing with Web Forms.
- More control over generated IDs.
- The ability to persist selected rows in data controls.
- More control over rendered HTML in the *FormView* and *ListView* controls.
- Filtering support for data source controls.

Setting Meta Tags with the Page.MetaKeywords and Page.MetaDescription Properties

ASP.NET 4 adds two properties to the *Page* class, *MetaKeywords* and *MetaDescription*. These two properties represent corresponding *meta* tags in your page, as shown in the following example:

```
<head id="Head1" runat="server">
  <title>Untitled Page</title>
  <meta name="keywords" content="These, are, my, keywords" />
  <meta name="description" content="This is the description of my page" />
</head>
```

These two properties work the same way that the page's *Title* property does. They follow these rules:

1. If there are no *meta* tags in the *head* element that match the property names (that is, name="keywords" for *Page.MetaKeywords* and name="description" for *Page.MetaDescription*, meaning that these properties have not been set), the *meta* tags will be added to the page when it is rendered.
2. If there are already *meta* tags with these names, these properties act as get and set methods for the contents of the existing tags.

You can set these properties at run time, which lets you get the content from a database or other source, and which lets you set the tags dynamically to describe what a particular page is for.

You can also set the *Keywords* and *Description* properties in the *@ Page* directive at the top of the Web Forms page markup, as in the following example:

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeFile="Default.aspx.cs"
  Inherits="_Default"
  Keywords="These, are, my, keywords"
  Description="This is a description" %>
```

This will override the *meta* tag contents (if any) already declared in the page.

The contents of the description *meta* tag are used for improving search listing previews in Google. (For details, see Improve snippets with a meta description makeover on the Google Webmaster Central blog.) Google and Windows Live Search do not use the contents of the keywords for anything, but other search engines might. For more information, see Meta Keywords Advice on the Search Engine Guide Web site.

These new properties are a simple feature, but they save you from the requirement to add these manually or from writing your own code to create the *meta* tags.

Enabling View State for Individual Controls

By default, view state is enabled for the page, with the result that each control on the page potentially stores view state even if it is not required for the application. View state data is included in the markup that a page generates and increases the amount of time it takes to send a page to the client and post it back. Storing more view state

than is necessary can cause significant performance degradation. In earlier versions of ASP.NET, developers could disable view state for individual controls in order to reduce page size, but had to do so explicitly for individual controls. In ASP.NET 4, Web server controls include a *ViewStateMode* property that lets you disable view state by default and then enable it only for the controls that require it in the page.

The *ViewStateMode* property takes an enumeration that has three values: *Enabled*, *Disabled*, and *Inherit*. *Enabled* enables view state for that control and for any child controls that are set to *Inherit* or that have nothing set. *Disabled* disables view state, and *Inherit* specifies that the control uses the *ViewStateMode* setting from the parent control.

The following example shows how the *ViewStateMode* property works. The markup and code for the controls in the following page includes values for the *ViewStateMode* property:

```
<form id="form1" runat="server">
  <script runat="server">
      protected override void OnLoad(EventArgs e) {
      if (!IsPostBack) {
        label1.Text = label2.Text = "[DynamicValue]";
      }
      base.OnLoad(e);
    }
  </script>
  <asp:PlaceHolder ID="PlaceHolder1" runat="server" ViewStateMode="Disabled">
      Disabled: <asp:Label ID="label1" runat="server" Text="[DeclaredValue]" /><br />
    <asp:PlaceHolder ID="PlaceHolder2" runat="server" ViewStateMode="Enabled">
        Enabled: <asp:Label ID="label2" runat="server" Text="[DeclaredValue]" />
    </asp:PlaceHolder>
  </asp:PlaceHolder>
  <hr />
  <asp:button ID="Button1" runat="server" Text="Postback" />
  <%-- Further markup here --%>
```

As you can see, the code disables view state for the PlaceHolder1 control. The child label1 control inherits this property value (*Inherit* is the default value for *ViewStateMode* for controls.) and therefore saves no view state. In the PlaceHolder2 control, *ViewStateMode* is set to *Enabled*, so label2 inherits this property and saves view state. When the page is first loaded, the *Text* property of both *Label* controls is set to the string "[DynamicValue]".

The effect of these settings is that when the page loads the first time, the following output is displayed in the browser:

Disabled `: [DynamicValue]`

Enabled: `[DynamicValue]`

After a postback, however, the following output is displayed:

Disabled `: [DeclaredValue]`

Enabled: `[DynamicValue]`

The label1 control (whose *ViewStateMode* value is set to *Disabled*) has not preserved the value that it was set to in code. However, the label2 control (whose *ViewStateMode* value is set to *Enabled*) has preserved its state.

You can also set *ViewStateMode* in the *@ Page* directive, as in the following example:

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeBehind="Default.aspx.cs"
  Inherits="WebApplication1._Default"
  ViewStateMode="Disabled" %>
```

The *Page* class is just another control; it acts as the parent control for all the other controls in the page. The default value of *ViewStateMode* is *Enabled* for instances of *Page*. Because controls default to *Inherit*, controls will inherit the *Enabled* property value unless you set *ViewStateMode* at page or control level.

The value of the *ViewStateMode* property determines if view state is maintained only if the *EnableViewState* property is set to *true*. If the *EnableViewState* property is set to *false*, view state will not be maintained even if *ViewStateMode* is set to *Enabled*.

A good use for this feature is with *ContentPlaceHolder* controls in master pages, where you can set *ViewStateMode* to *Disabled* for the master page and then enable it individually for *ContentPlaceHolder* controls that in turn contain controls that require view state.

Changes to Browser Capabilities

ASP.NET determines the capabilities of the browser that a user is using to browse your site by using a feature called *browser capabilities*. Browser capabilities are represented by the *HttpBrowserCapabilities* object (exposed by the *Request.Browser* property). For example, you can use the *HttpBrowserCapabilities* object to determine whether the type and version of the current browser supports a particular version of JavaScript. Or, you can use the *HttpBrowserCapabilities* object to determine whether the request originated from a mobile device.

The *HttpBrowserCapabilities* object is driven by a set of browser definition files. These files contain information about the capabilities of particular browsers. In ASP.NET 4, these browser definition files have been updated to contain information about recently introduced browsers and devices such as Google Chrome, Research in Motion BlackBerry smartphones, and Apple iPhone.

The following list shows new browser definition files:

- *blackberry.browser*
- *chrome.browser*
- *Default.browser*
- *firefox.browser*
- *gateway.browser*
- *generic.browser*
- *ie.browser*
- *iemobile.browser*
- *iphone.browser*
- *opera.browser*
- *safari.browser*

Using Browser Capabilities Providers

In ASP.NET version 3.5 Service Pack 1, you can define the capabilities that a browser has in the following ways:

- At the computer level, you create or update a `.browser` XML file in the following folder:

  ```
  \Windows\Microsoft.NET\Framework\v2.0.50727\CONFIG\Browsers
  ```

- After you define the browser capability, you run the following command from the Visual Studio Command Prompt in order to rebuild the browser capabilities assembly and add it to the GAC:

  ```
  aspnet_regbrowsers.exe -I c
  ```

- For an individual application, you create a `.browser` file in the application's `App_Browsers` folder.

These approaches require you to change XML files, and for computer-level changes, you must restart the

application after you run the aspnet_regbrowsers.exe process.

ASP.NET 4 includes a feature referred to as *browser capabilities providers*. As the name suggests, this lets you build a provider that in turn lets you use your own code to determine browser capabilities.

In practice, developers often do not define custom browser capabilities. Browser files are hard to update, the process for updating them is fairly complicated, and the XML syntax for `.browser` files can be complex to use and define. What would make this process much easier is if there were a common browser definition syntax, or a database that contained up-to-date browser definitions, or even a Web service for such a database. The new browser capabilities providers feature makes these scenarios possible and practical for third-party developers.

There are two main approaches for using the new ASP.NET 4 browser capabilities provider feature: extending the ASP.NET browser capabilities definition functionality, or totally replacing it. The following sections describe first how to replace the functionality, and then how to extend it.

Replacing the ASP.NET Browser Capabilities Functionality

To replace the ASP.NET browser capabilities definition functionality completely, follow these steps:

1. Create a provider class that derives from *HttpCapabilitiesProvider* and that overrides the *GetBrowserCapabilities* method, as in the following example:

   ```
   public class CustomProvider : HttpCapabilitiesProvider
   {
       public override HttpBrowserCapabilities
       GetBrowserCapabilities(HttpRequest request)
       {
           HttpBrowserCapabilities browserCaps = new HttpBrowserCapabilities();
           Hashtable values = new Hashtable(180, StringComparer.OrdinalIgnoreCase);
           values[String.Empty] = request.UserAgent;
           values["browser"] = "MyCustomBrowser";
           browserCaps.Capabilities = values;
           return browserCaps;
       }
   }
   ```

   The code in this example creates a new *HttpBrowserCapabilities* object, specifying only the capability named browser and setting that capability to MyCustomBrowser.

2. Register the provider with the application.

   In order to use a provider with an application, you must add the *provider* attribute to the *browserCaps* section in the `Web.config` or `Machine.config` files. (You can also define the provider attributes in a *location* element for specific directories in application, such as in a folder for a specific mobile device.) The following example shows how to set the *provider* attribute in a configuration file:

   ```
   <system.web>
   <browserCaps provider="ClassLibrary2.CustomProvider, ClassLibrary2,
     Version=1.0.0.0, Culture=neutral" />
   </system.web>
   ```

   Another way to register the new browser capability definition is to use code, as shown in the following example:

```
void Application_Start(object sender, EventArgs e)
{
    HttpCapabilitiesBase.BrowserCapabilitiesProvider =
    new ClassLibrary2.CustomProvider();
    // ...
 }
```

This code must run in the *Application_Start* event of the `Global.asax` file. Any change to the *BrowserCapabilitiesProvider* class must occur before any code in the application executes, in order to make sure that the cache remains in a valid state for the resolved *HttpCapabilitiesBase* object.

Caching the HttpBrowserCapabilities Object

The preceding example has one problem, which is that the code would run each time the custom provider is invoked in order to get the *HttpBrowserCapabilities* object. This can happen multiple times during each request. In the example, the code for the provider does not do much. However, if the code in your custom provider performs significant work in order to get the *HttpBrowserCapabilities* object, this can affect performance. To prevent this from happening, you can cache the *HttpBrowserCapabilities* object. Follow these steps:

1. Create a class that derives from *HttpCapabilitiesProvider*, like the one in the following example:

```
public class CustomProvider : HttpCapabilitiesProvider
{
    public override HttpBrowserCapabilities
    GetBrowserCapabilities(HttpRequest request)
    {
        string cacheKey = BuildCacheKey();
        int cacheTime = GetCacheTime();
        HttpBrowserCapabilities browserCaps =
        HttpContext.Current.Cache[cacheKey] as
        HttpBrowserCapabilities;
        if (browserCaps == null)
        {
            HttpBrowserCapabilities browserCaps = new
            HttpBrowserCapabilities();
            Hashtable values = new Hashtable(180,
            StringComparer.OrdinalIgnoreCase);
            values[String.Empty] = request.UserAgent;
            values["browser"] = "MyCustomBrowser";
            browserCaps.Capabilities = values;
            HttpContext.Current.Cache.Insert(cacheKey,
            browserCaps, null, DateTime.MaxValue,
            TimeSpan.FromSeconds(cacheTime));
        }
        return browserCaps;
    }
}
```

   In the example, the code generates a cache key by calling a custom BuildCacheKey method, and it gets the length of time to cache by calling a custom GetCacheTime method. The code then adds the resolved *HttpBrowserCapabilities* object to the cache. The object can be retrieved from the cache and reused on subsequent requests that make use of the custom provider.

2. Register the provider with the application as described in the preceding procedure.

Extending ASP.NET Browser Capabilities Functionality

The previous section described how to create a new *HttpBrowserCapabilities* object in ASP.NET 4. You can also extend the ASP.NET browser capabilities functionality by adding new browser capabilities definitions to those that are already in ASP.NET. You can do this without using the XML browser definitions. The following procedure shows how.

1. Create a class that derives from *HttpCapabilitiesEvaluator* and that overrides the *GetBrowserCapabilities* method, as shown in the following example:

```
public class CustomProvider : HttpCapabilitiesEvaluator
{
    public override HttpBrowserCapabilities
    GetBrowserCapabilities(HttpRequest request)
    {
        HttpBrowserCapabilities browserCaps =
        base.GetHttpBrowserCapabilities(request);
        if (browserCaps.Browser == "Unknown")
        {
            browserCaps = MyBrowserCapabilitiesEvaulator(request);
        }
        return browserCaps;
    }
}
```

   This code first uses the ASP.NET browser capabilities functionality to try to identify the browser. However, if no browser is identified based on the information defined in the request (that is, if the *Browser* property of the *HttpBrowserCapabilities* object is the string "Unknown"), the code calls the custom provider (MyBrowserCapabilitiesEvaluator) to identify the browser.

2. Register the provider with the application as described in the previous example.

Extending Browser Capabilities Functionality by Adding New Capabilities to Existing Capabilities Definitions

In addition to creating a custom browser definition provider and to dynamically creating new browser definitions, you can extend existing browser definitions with additional capabilities. This lets you use a definition that is close to what you want but lacks only a few capabilities. To do this, use the following steps.

1. Create a class that derives from *HttpCapabilitiesEvaluator* and that overrides the *GetBrowserCapabilities* method, as shown in the following example:

```
public class CustomProvider : HttpCapabilitiesEvaluator
{
    public override HttpBrowserCapabilities
    GetBrowserCapabilities(HttpRequest request)
    {
        HttpBrowserCapabilities browserCaps =
          base.GetHttpBrowserCapabilities(request);
        if (browserCaps.Browser == "Unknown")
        {
            browserCaps = MyBrowserCapabilitiesEvaulator(request);
        }
        return browserCaps;
    }
}
```

   The example code extends the existing ASP.NET *HttpCapabilitiesEvaluator* class and gets the *HttpBrowserCapabilities* object that matches the current request definition by using the following code:

```
HttpBrowserCapabilities browserCaps =
    base.GetHttpBrowserCapabilities(request);
```

   The code can then add or modify a capability for this browser. There are two ways to specify a new browser capability:

   - Add a key/value pair to the *IDictionary* object that is exposed by the *Capabilities* property of the *HttpCapabilitiesBase* object. In the previous example, the code adds a capability named MultiTouch

with a value of *true*.

- Set existing properties of the *HttpCapabilitiesBase* object. In the previous example, the code sets the *Frames* property to *true*. This property is simply an accessor for the *IDictionary* object that is exposed by the *Capabilities* property.

> **NOTE**
>
> Note This model applies to any property of *HttpBrowserCapabilities*, including control adapters.

2. Register the provider with the application as described in the earlier procedure.

Routing in ASP.NET 4

ASP.NET 4 adds built-in support for using routing with Web Forms. Routing lets you configure an application to accept request URLs that do not map to physical files. Instead, you can use routing to define URLs that are meaningful to users and that can help with search-engine optimization (SEO) for your application. For example, the URL for a page that displays product categories in an existing application might look like the following example:

```
http://website/products.aspx?categoryid=12
```

By using routing, you can configure the application to accept the following URL to render the same information:

```
http://website/products/software
```

Routing has been available starting with ASP.NET 3.5 SP1. (For an example of how to use routing in ASP.NET 3.5 SP1, see the entry Using Routing With WebForms on Phil Haack's blog.) However, ASP.NET 4 includes some features that make it easier to use routing, including the following:

- The *PageRouteHandler* class, which is a simple HTTP handler that you use when you define routes. The class passes data to the page that the request is routed to.
- The new properties *HttpRequest.RequestContext* and *Page.RouteData* (which is a proxy for the *HttpRequest.RequestContext.RouteData* object). These properties make it easier to access information that is passed from the route.
- The following new expression builders, which are defined in *System.Web.Compilation.RouteUrlExpressionBuilder* and *System.Web.Compilation.RouteValueExpressionBuilder*.
- *RouteUrl*, which provides a simple way to create a URL that corresponds to a route URL within an ASP.NET server control.
- *RouteValue*, which provides a simple way to extract information from the *RouteContext* object.
- The *RouteParameter* class, which makes it easier to pass data contained in a *RouteContext* object to a query for a data source control (similar to *FormParameter*).

Routing for Web Forms Pages

The following example shows how to define a Web Forms route by using the new *MapPageRoute* method of the *Route* class:

```
public class Global : System.Web.HttpApplication
{
    void Application_Start(object sender, EventArgs e)
    {
        RouteTable.Routes.MapPageRoute("SearchRoute",
          "search/{searchterm}", "~/search.aspx");
        RouteTable.Routes.MapPageRoute("UserRoute",
          "users/{username}", "~/users.aspx");
    }
}
```

ASP.NET 4 introduces the *MapPageRoute* method. The following example is equivalent to the SearchRoute definition shown in the previous example, but uses the *PageRouteHandler* class.

```
RouteTable.Routes.Add("SearchRoute", new Route("search/{searchterm}",
  new PageRouteHandler("~/search.aspx")));
```

The code in the example maps the route to a physical page (in the first route, to `~/search.aspx`). The first route definition also specifies that the parameter named searchterm should be extracted from the URL and passed to the page.

The *MapPageRoute* method supports the following method overloads:

- *MapPageRoute(string routeName, string routeUrl, string physicalFile, bool checkPhysicalUrlAccess)*
- *MapPageRoute(string routeName, string routeUrl, string physicalFile, bool checkPhysicalUrlAccess, RouteValueDictionary defaults)*
- *MapPageRoute(string routeName, string routeUrl, string physicalFile, bool checkPhysicalUrlAccess, RouteValueDictionary defaults, RouteValueDictionary constraints)*

The *checkPhysicalUrlAccess* parameter specifies whether the route should check the security permissions for the physical page being routed to (in this case, search.aspx) and the permissions on the incoming URL (in this case, search/{searchterm}). If the value of *checkPhysicalUrlAccess* is *false*, only the permissions of the incoming URL will be checked. These permissions are defined in the `Web.config` file using settings such as the following:

```
<configuration>
  <location path="search.aspx">
    <system.web>
      <authorization>
        <allow roles="admin"/>
        <deny users="*"/>
      </authorization>
    </system.web>
  </location>
  <location path="search">
    <system.web>
      <authorization>
        <allow users="*"/>
      </authorization>
    </system.web>
  </location>

</configuration>
```

In the example configuration, access is denied to the physical page `search.aspx` for all users except those who are in the admin role. When the *checkPhysicalUrlAccess* parameter is set to *true* (which is its default value), only admin users are allowed to access the URL /search/{searchterm}, because the physical page search.aspx is restricted to users in that role. If *checkPhysicalUrlAccess* is set to *false* and the site is configured as shown in the previous example, all authenticated users are allowed to access the URL /search/{searchterm}.

Reading Routing Information in a Web Forms Page

In the code of the Web Forms physical page, you can access the information that routing has extracted from the URL (or other information that another object has added to the *RouteData* object) by using two new properties: *HttpRequest.RequestContext* and *Page.RouteData*. (*Page.RouteData* wraps *HttpRequest.RequestContext.RouteData*.) The following example shows how to use *Page.RouteData*.

```
protected void Page_Load(object sender, EventArgs e)
{
    string searchterm = Page.RouteData.Values["searchterm"] as string;
    label1.Text = searchterm;
}
```

The code extracts the value that was passed for the searchterm parameter, as defined in the example route earlier. Consider the following request URL:

```
http://localhost/search/scott/
```

When this request is made, the word "scott" would be rendered in the `search.aspx` page.

Accessing Routing Information in Markup

The method described in the previous section shows how to get route data in code in a Web Forms page. You can also use expressions in markup that give you access to the same information. Expression builders are a powerful and elegant way to work with declarative code. (For more information, see the entry Express Yourself With Custom Expression Builders on Phil Haack's blog.)

ASP.NET 4 includes two new expression builders for Web Forms routing. The following example shows how to use them.

```
<asp:HyperLink ID="HyperLink1" runat="server"
  NavigateUrl="<%$RouteUrl:SearchTerm=scott%>">Search for Scott</asp:HyperLink>
```

In the example, the *RouteUrl* expression is used to define a URL that is based on a route parameter. This saves you from having to hard-code the complete URL into the markup, and lets you change the URL structure later without requiring any change to this link.

Based on the route defined earlier, this markup generates the following URL:

```
http://localhost/search/scott
```

ASP.NET automatically works out the correct route (that is, it generates the correct URL) based on the input parameters. You can also include a route name in the expression, which lets you specify a route to use.

The following example shows how to use the *RouteValue* expression.

```
<asp:Label ID="Label1" runat="server" Text="<%$RouteValue:SearchTerm%>" />
```

When the page that contains this control runs, the value "scott" is displayed in the label.

The *RouteValue* expression makes it simple to use route data in markup, and it avoids having to work with the more complex Page.RouteData["x"] syntax in markup.

Using Route Data for Data Source Control Parameters

The *RouteParameter* class lets you specify route data as a parameter value for queries in a data source control. It works much like the class, as shown in the following example:

```
<asp:sqldatasource id="SqlDataSource1" runat="server"
    connectionstring="<%$ ConnectionStrings:MyNorthwind %>"
    selectcommand="SELECT CompanyName,ShipperID FROM Shippers where
      CompanyName=@companyname"
  <selectparameters>
    <asp:routeparameter name="companyname" RouteKey="searchterm" />
  </selectparameters>

</asp:sqldatasource>
```

In this case, the value of the route parameter searchterm will be used for the @companyname parameter in the *Select* statement.

Setting Client IDs

The new *ClientIDMode* property addresses a long-standing issue in ASP.NET, namely how controls create the *id* attribute for elements that they render. Knowing the *id* attribute for rendered elements is important if your application includes client script that references these elements.

The *id* attribute in HTML that is rendered for Web server controls is generated based on the *ClientID* property of the control. Until ASP.NET 4, the algorithm for generating the *id* attribute from the *ClientID* property has been to concatenate the naming container (if any) with the ID, and in the case of repeated controls (as in data controls), to add a prefix and a sequential number. While this has always guaranteed that the IDs of controls in the page are unique, the algorithm has resulted in control IDs that were not predictable, and were therefore difficult to reference in client script.

The new *ClientIDMode* property lets you specify more precisely how the client ID is generated for controls. You can set the *ClientIDMode* property for any control, including for the page. Possible settings are the following:

- *AutoID* – This is equivalent to the algorithm for generating *ClientID* property values that was used in earlier versions of ASP.NET.
- *Static* – This specifies that the *ClientID* value will be the same as the ID without concatenating the IDs of parent naming containers. This can be useful in Web user controls. Because a Web user control can be located on different pages and in different container controls, it can be difficult to write client script for controls that use the *AutoID* algorithm because you cannot predict what the ID values will be.
- *Predictable* – This option is primarily for use in data controls that use repeating templates. It concatenates the ID properties of the control's naming containers, but generated *ClientID* values do not contain strings like "ctlxxx". This setting works in conjunction with the *ClientIDRowSuffix* property of the control. You set the *ClientIDRowSuffix* property to the name of a data field, and the value of that field is used as the suffix for the generated *ClientID* value. Typically you would use the primary key of a data record as the *ClientIDRowSuffix* value.
- *Inherit* – This setting is the default behavior for controls; it specifies that a control's ID generation is the same as its parent.

You can set the *ClientIDMode* property at the page level. This defines the default *ClientIDMode* value for all controls in the current page.

The default *ClientIDMode* value at the page level is *AutoID*, and the default *ClientIDMode* value at the control level is *Inherit*. As a result, if you do not set this property anywhere in your code, all controls will default to the *AutoID* algorithm.

You set the page-level value in the *@ Page* directive, as shown in the following example:

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeFile="Default.aspx.cs"
  Inherits="_Default"
  ClientIDMode="Predictable" %>
```

You can also set the *ClientIDMode* value in the configuration file, either at the computer (machine) level or at the application level. This defines the default *ClientIDMode* setting for all controls in all pages in the application. If you set the value at the computer level, it defines the default *ClientIDMode* setting for all Web sites on that computer. The following example shows the *ClientIDMode* setting in the configuration file:

```
<system.web>
  <pages clientIDMode="Predictable"></pages>
</system.web>
```

As noted earlier, the value of the *ClientID* property is derived from the naming container for a control's parent. In some scenarios, such as when you are using master pages, controls can end up with IDs like those in the following rendered HTML:

```
<div id="ctl00_ContentPlaceHolder1_ParentPanel">
  <div id="ctl00_ContentPlaceHolder1_ParentPanel_NamingPanel1">
    <input name="ctl00$ContentPlaceHolder1$ParentPanel$NamingPanel1$TextBox1"
      type="text" value="Hello!"
      id="ctl00_ContentPlaceHolder1_ParentPanel_NamingPanel1_TextBox1" />
</div>
```

Even though the *input* element shown in the markup (from a *TextBox* control) is only two naming containers deep in the page (the nested *ContentPlaceholder* controls), because of the way master pages are processed, the end result is a control ID like the following:

```
ctl00_ContentPlaceHolder1_ParentPanel_NamingPanel1_TextBox1
```

This ID is guaranteed to be unique in the page, but is unnecessarily long for most purposes. Imagine that you want to reduce the length of the rendered ID, and to have more control over how the ID is generated. (For example, you want to eliminate "ctlxxx" prefixes.) The easiest way to achieve this is by setting the *ClientIDMode* property as shown in the following example:

```
<tc:NamingPanel runat="server" ID="ParentPanel" ClientIDMode="Static">
  <tc:NamingPanel runat="server" ID="NamingPanel1" ClientIDMode="Predictable">
    <asp:TextBox ID="TextBox1" runat="server" Text="Hello!"></asp:TextBox>
  </tc:NamingPanel>

</tc:NamingPanel>
```

In this sample, the *ClientIDMode* property is set to *Static* for the outermost *NamingPanel* element, and set to *Predictable* for the inner *NamingControl* element. These settings result in the following markup (the rest of the page and the master page is assumed to be the same as in the previous example):

```
<div id="ParentPanel">
  <div id="ParentPanel_NamingPanel1">
    <input name="ctl00$ContentPlaceHolder1$ParentPanel$NamingPanel1$TextBox1"
        type="text" value="Hello!" id="ParentPanel_NamingPanel1_TextBox1" />
</div>
```

The *Static* setting has the effect of resetting the naming hierarchy for any controls inside the outermost

NamingPanel element, and of eliminating the *ContentPlaceHolder* and *MasterPage* IDs from the generated ID. (The *name* attribute of rendered elements is unaffected, so the normal ASP.NET functionality is retained for events, view state, and so on.) A side effect of resetting the naming hierarchy is that even if you move the markup for the *NamingPanel* elements to a different *ContentPlaceholder* control, the rendered client IDs remain the same.

> **NOTE**
>
> Note It is up to you to make sure that the rendered control IDs are unique. If they are not, it can break any functionality that requires unique IDs for individual HTML elements, such as the client *document.getElementById* function.

Creating Predictable Client IDs in Data-Bound Controls

The *ClientID* values that are generated for controls in a data-bound list control by the legacy algorithm can be long and are not really predictable. The *ClientIDMode* functionality can help you have more control over how these IDs are generated.

The markup in the following example includes a *ListView* control:

```
<asp:ListView ID="ListView1" runat="server" DataSourceID="SqlDataSource1"
    OnSelectedIndexChanged="ListView1_SelectedIndexChanged"
    ClientIDMode="Predictable"
    RowClientIDRowSuffix="ProductID">
</asp:ListView>
```

In the previous example, the *ClientIDMode* and *RowClientIDRowSuffix* properties are set in markup. The *ClientIDRowSuffix* property can be used only in data-bound controls, and its behavior differs depending on which control you are using. The differences are these:

- *GridView* control — You can specify the name of one or more columns in the data source, which are combined at run time to create the client IDs. For example, if you set *RowClientIDRowSuffix* to "ProductName, ProductId", control IDs for rendered elements will have a format like the following:

  ```
  rootPanel_GridView1_ProductNameLabel_Chai_1
  ```

- *ListView* control — You can specify a single column in the data source that is appended to the client ID. For example, if you set *ClientIDRowSuffix* to "ProductName", the rendered control IDs will have a format like the following:

  ```
  rootPanel_ListView1_ProductNameLabel_1
  ```

- In this case the trailing 1 is derived from the product ID of the current data item.
- *Repeater* control— This control does not support the *ClientIDRowSuffix* property. In a *Repeater* control, the index of the current row is used. When you use ClientIDMode="Predictable" with a *Repeater* control, client IDs are generated that have the following format:

  ```
  Repeater1_ProductNameLabel_0
  ```

- The trailing 0 is the index of the current row.

The *FormView* and *DetailsView* controls do not display multiple rows, so they do not support the *ClientIDRowSuffix* property.

Persisting Row Selection in Data Controls

The *GridView* and *ListView* controls can let users select a row. In previous versions of ASP.NET, selection has been based on the row index on the page. For example, if you select the third item on page 1 and then move to page 2, the third item on that page is selected.

Persisted selection was initially supported only in Dynamic Data projects in the .NET Framework 3.5 SP1. When this feature is enabled, the current selected item is based on the data key for the item. This means that if you select the third row on page 1 and move to page 2, nothing is selected on page 2. When you move back to page 1, the third row is still selected. Persisted selection is now supported for the *GridView* and *ListView* controls in all projects by using the *EnablePersistedSelection* property, as shown in the following example:

```
<asp:GridView id="GridView2" runat="server" EnablePersistedSelection="true">
</asp:GridView>
```

ASP.NET Chart Control

The ASP.NET *Chart* control expands the data-visualization offerings in the .NET Framework. Using the *Chart* control, you can create ASP.NET pages that have intuitive and visually compelling charts for complex statistical or financial analysis. The ASP.NET *Chart* control was introduced as an add-on to the .NET Framework version 3.5 SP1 release and is part of the .NET Framework 4 release.

The control includes the following features:

- 35 distinct chart types.
- An unlimited number of chart areas, titles, legends, and annotations.
- A wide variety of appearance settings for all chart elements.
- 3-D support for most chart types.
- Smart data labels that can automatically fit around data points.
- Strip lines, scale breaks, and logarithmic scaling.
- More than 50 financial and statistical formulas for data analysis and transformation.
- Simple binding and manipulation of chart data.
- Support for common data formats such as dates, times, and currency.
- Support for interactivity and event-driven customization, including client click events using Ajax.
- State management.
- Binary streaming.

The following figures show examples of financial charts that are produced by the ASP.NET Chart control.

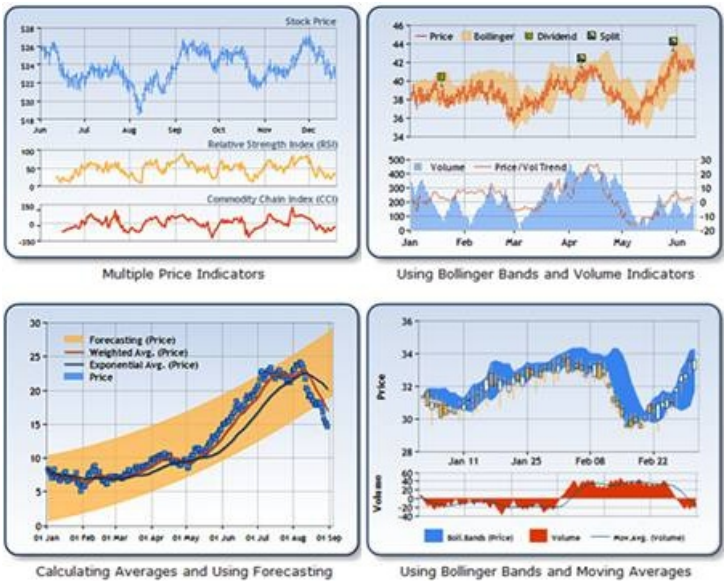


Figure 2: ASP.NET Chart control examples

For more examples of how to use the ASP.NET Chart control, download the sample code on the Samples Environment for Microsoft Chart Controls page on the MSDN Web site. You can find more samples of community content at the Chart Control Forum.

Adding the Chart Control to an ASP.NET Page

The following example shows how to add a *Chart* control to an ASP.NET page by using markup. In the example, the *Chart* control produces a column chart for static data points.

```
<asp:Chart ID="Chart1" runat="server">
  <Series>
    <asp:Series Name="Series1" ChartType="Column">
      <Points>
        <asp:DataPoint AxisLabel="Product A" YValues="345"/>
        <asp:DataPoint AxisLabel="Product B" YValues="456"/>
        <asp:DataPoint AxisLabel="Product C" YValues="125"/>
        <asp:DataPoint AxisLabel="Product D" YValues="957"/> &

      lt;/Points>
    </asp:Series>
  </Series>
  <ChartAreas>
    <asp:ChartArea Name="ChartArea1">
      <AxisY IsLogarithmic="True" />
    </asp:ChartArea>
  </ChartAreas>
  <Legends>
    <asp:Legend Name="Legend1" Title="Product Sales" />
  </Legends>

</asp:Chart>
```

Using 3-D Charts

The *Chart* control contains a *ChartAreas* collection, which can contain *ChartArea* objects that define characteristics of chart areas. For example, to use 3-D for a chart area, use the *Area3DStyle* property as in the following example:

```
<asp:ChartArea Name="ChartArea1">
  <area3dstyle
      Rotation="10"
      Perspective="10"
      Enable3D="True"
      Inclination="15"
      IsRightAngleAxes="False"
      WallWidth="0"
      IsClustered="False" />

  <%-- Additional markup here --%>
</asp:ChartArea>
```

The figure below shows a 3-D chart with four series of the *Bar* chart type.

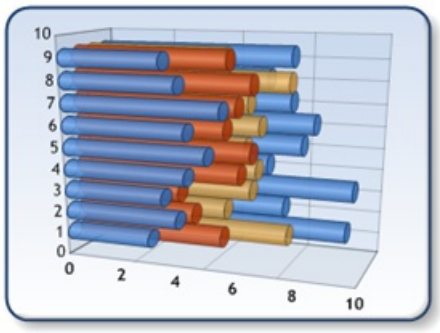


Figure 3: 3-D Bar Chart

Using Scale Breaks and Logarithmic Scales

Scale breaks and logarithmic scales are two additional ways to add sophistication to the chart. These features are specific to each axis in a chart area. For example, to use these features on the primary Y axis of a chart area, use the *AxisY.IsLogarithmic* and *ScaleBreakStyle* properties in a *ChartArea* object. The following snippet shows how to use scale breaks on the primary Y axis.

```
<asp:ChartArea Name="ChartArea1">

  <axisy>

    <ScaleBreakStyle
        BreakLineStyle="Wave"
        CollapsibleSpaceThreshold="40"
        Enabled="True" />
  </axisy>

<%-- Additional markup here --%>
</asp:ChartArea>
```

The figure below shows the Y axis with scale breaks enabled.

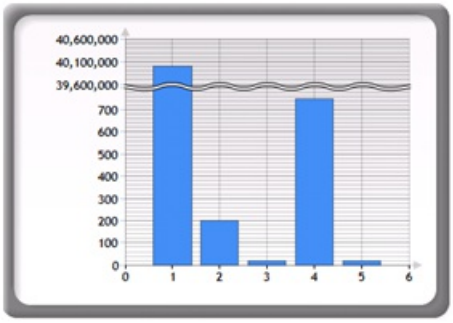


Figure 4: Scale Breaks

Filtering Data with the QueryExtender Control

A very common task for developers who create data-driven Web pages is to filter data. This traditionally has been performed by building *Where* clauses in data source controls. This approach can be complicated, and in some cases the *Where* syntax does not let you take advantage of the full functionality of the underlying database.

To make filtering easier, a new *QueryExtender* control has been added in ASP.NET 4. This control can be added to *EntityDataSource* or *LinqDataSource* controls in order to filter the data returned by these controls. Because the *QueryExtender* control relies on LINQ, the filter is applied on the database server before the data is sent to the page, which results in very efficient operations.

The *QueryExtender* control supports a variety of filter options. The following sections describe these options and provide examples of how to use them.

Search

For the search option, the *QueryExtender* control performs a search in specified fields. In the following example, the control uses the text that is entered in the TextBoxSearch control and searches for its contents in the `ProductName` and `Supplier.CompanyName` columns in the data that is returned from the *LinqDataSource* control.

```
<asp:LinqDataSource ID="dataSource" runat="server"> TableName="Products">
</asp:LinqDataSource>
<asp:QueryExtender TargetControlID="dataSource" runat="server">
  <asp:SearchExpression DataFields="ProductName, Supplier.CompanyName"
      SearchType="StartsWith">
    <asp:ControlParameter ControlID="TextBoxSearch" />
  </asp:SearchExpression>
</asp:QueryExtender>
```

Range

The range option is similar to the search option, but specifies a pair of values to define the range. In the following example, the *QueryExtender* control searches the `UnitPrice` column in the data returned from the *LinqDataSource* control. The range is read from the TextBoxFrom and TextBoxTo controls on the page.

```
<asp:LinqDataSource ID="dataSource" runat="server"> TableName="Products">
</asp:LinqDataSource>
<asp:QueryExtender TargetControlID="dataSource" runat="server">
  <asp:RangeExpression DataField="UnitPrice" MinType="Inclusive"
      MaxType="Inclusive">
    <asp:ControlParameter ControlID="TextBoxFrom" />
    <asp:ControlParameter ControlID="TexBoxTo" />
  </asp:RangeExpression>

</asp:QueryExtender>
```

PropertyExpression

The property expression option lets you define a comparison to a property value. If the expression evaluates to *true*, the data that is being examined is returned. In the following example, the *QueryExtender* control filters data by comparing the data in the `Discontinued` column to the value from the CheckBoxDiscontinued control on the page.

```
<asp:LinqDataSource ID="dataSource" runat="server" TableName="Products">
</asp:LinqDataSource>
<asp:QueryExtender TargetControlID="dataSource" runat="server">
   <asp:PropertyExpression>
      <asp:ControlParameter ControlID="CheckBoxDiscontinued" Name="Discontinued" />
   </asp:PropertyExpression>
</asp:QueryExtender>
```

CustomExpression

Finally, you can specify a custom expression to use with the *QueryExtender* control. This option lets you call a function in the page that defines custom filter logic. The following example shows how to declaratively specify a custom expression in the *QueryExtender* control.

```
<asp:LinqDataSource ID="dataSource" runat="server" TableName="Products">
</asp:LinqDataSource>
<asp:QueryExtender TargetControlID="dataSource" runat="server">
  <asp:CustomExpression OnQuerying="FilterProducts" />
</asp:QueryExtender>
```

The following example shows the custom function that is invoked by the *QueryExtender* control. In this case, instead of using a database query that includes a *Where* clause, the code uses a LINQ query to filter the data.

```
protected void FilterProducts(object sender, CustomExpressionEventArgs e)
{
    e.Query = from p in e.Query.Cast()
      where p.UnitPrice >= 10
      select p;
}
```

These examples show only one expression being used in the *QueryExtender* control at a time. However, you can include multiple expressions inside the *QueryExtender* control.

Html Encoded Code Expressions

Some ASP.NET sites (especially with ASP.NET MVC) rely heavily on using `<%=` `expression %>` syntax (often called "code nuggets") to write some text to the response. When you use code expressions, it is easy to forget to HTML-encode the text, If the text comes from user input, it can leave pages open to an XSS (Cross Site Scripting) attack.

ASP.NET 4 introduces the following new syntax for code expressions:

```
<%: expression %>
```

This syntax uses HTML encoding by default when writing to the response. This new expression effectively translates to the following:

```
<%= HttpUtility.HtmlEncode(expression) %>
```

For example, <%: Request["UserInput"] %> performs HTML encoding on the value of *Request["UserInput"]*.

The goal of this feature is to make it possible to replace all instances of the old syntax with the new syntax so that you are not forced to decide at every step which one to use. However, there are cases in which the text being output is meant to be HTML or is already encoded, in which case this could lead to double encoding.

For those cases, ASP.NET 4 introduces a new interface, *IHtmlString*, along with a concrete implementation, *HtmlString*. Instances of these types let you indicate that the return value is already properly encoded (or otherwise examined) for displaying as HTML, and that therefore the value should not be HTML-encoded again. For example, the following should not be (and is not) HTML encoded:

```
<%: new HtmlString("<strong>HTML that is not encoded</strong>") %>
```

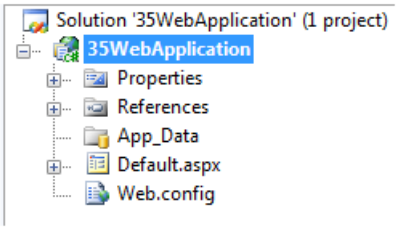
ASP.NET MVC 2 helper methods have been updated to work with this new syntax so that they are not double encoded, but only when you are running ASP.NET 4. This new syntax does not work when you run an application using ASP.NET 3.5 SP1.

Keep in mind that this does not guarantee protection from XSS attacks. For example, HTML that uses attribute values that are not in quotation marks can contain user input that is still susceptible. Note that the output of ASP.NET controls and ASP.NET MVC helpers always includes attribute values in quotation marks, which is the recommended approach.

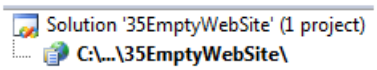
Likewise, this syntax does not perform JavaScript encoding, such as when you create a JavaScript string based on user input.

Project Template Changes

In earlier versions of ASP.NET, when you use Visual Studio to create a new Web Site project or Web Application project, the resulting projects contain only a Default.aspx page, a default `Web.config` file, and the `App_Data` folder, as shown in the following illustration:



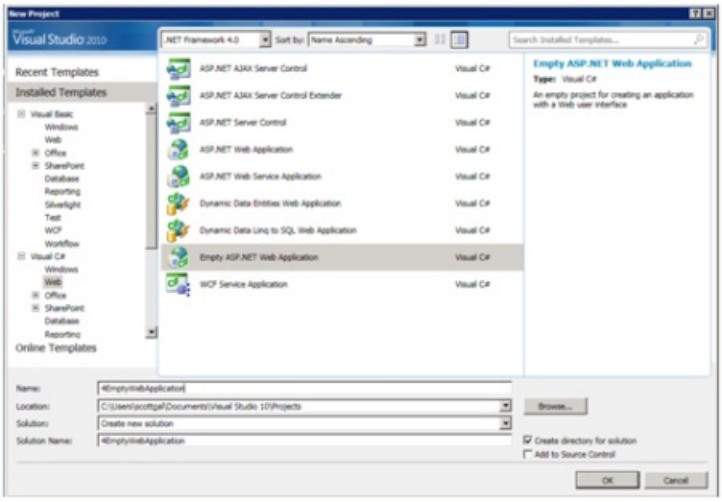
Visual Studio also supports an Empty Web Site project type, which contains no files at all, as shown in the following figure:



The result is that for the beginner, there is very little guidance on how to build a production Web application. Therefore, ASP.NET 4 introduces three new templates, one for an empty Web application project, and one each for a Web Application and Web Site project.

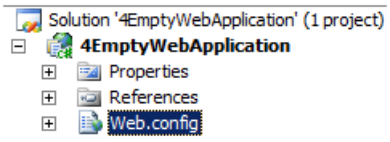
Empty Web Application Template

As the name suggests, the Empty Web Application template is a stripped-down Web Application project. You select this project template from the Visual Studio New Project dialog box, as shown in the following figure:



(Click to view full-size image)

When you create an Empty ASP.NET Web Application, Visual Studio creates the following folder layout:



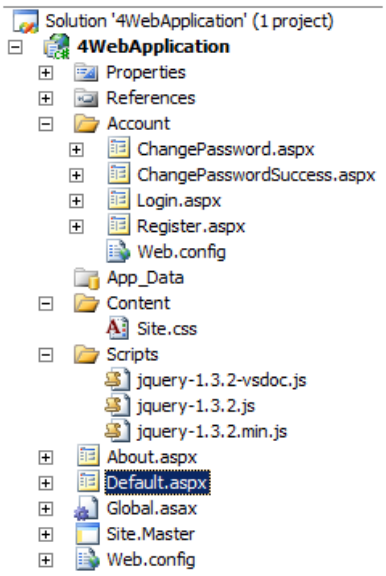
This is similar to the Empty Web Site layout from earlier versions of ASP.NET, with one exception. In Visual Studio 2010, Empty Web Application and Empty Web Site projects contain the following minimal `Web.config` file that contains information used by Visual Studio to identify the framework that the project is targeting:

```
<?xml version="1.0"?>
<configuration>
    <system.web>
        <compilation debug="true" targetFramework="4.0" />
    </system.web>
</configuration>
```

Without this *targetFramework* property, Visual Studio defaults to targeting the .NET Framework 2.0 in order to preserve compatibility when opening older applications.

Web Application and Web Site Project Templates

The other two new project templates that are shipped with Visual Studio 2010 contain major changes. The following figure shows the project layout that is created when you create a new Web Application project. (The layout for a Web Site project is virtually identical.)



The project includes a number of files that were not created in earlier versions. In addition, the new Web Application project is configured with basic membership functionality, which lets you quickly get started in securing access to the new application. Because of this inclusion, the `Web.config` file for the new project includes entries that are used to configure membership, roles, and profiles. The following example shows the `Web.config` file for a new Web Application project. (In this case, *roleManager* is disabled.)

```
<?xml version="1.0"?>
<configuration>
    <connectionStrings>
      <add name="ApplicationServices" connectionString="data source=.\SQLEXPRESS;Integrated Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User
Instance=true" providerName="System.Data.SqlClient"/>
    </connectionStrings>

    <system.web>
      <compilation debug="true" targetFramework="4.0" />

      <authentication mode="Forms">
        <forms loginUrl="~/Account/Login.aspx"/>
      </authentication>

      <membership>
        <providers>
          <clear/>
          <add name="AspNetSqlMembershipProvider" type="System.Web.Security.SqlMembershipProvider, System.Web, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" connectionStringName="ApplicationServices" enablePasswordRetrieval="false" enablePasswordReset="true"
requiresQuestionAndAnswer="false" requiresUniqueEmail="false" passwordFormat="Hashed" maxInvalidPasswordAttempts="5" minRequiredPasswordLength="6"
minRequiredNonalphanumericCharacters="0" passwordAttemptWindow="10" passwordStrengthRegularExpression="" applicationName="/"/>
        </providers>
      </membership>

      <profile>
        <providers>
          <clear/>
          <add name="AspNetSqlProfileProvider" type="System.Web.Profile.SqlProfileProvider, System.Web, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" connectionStringName="ApplicationServices" applicationName="/"/>
        </providers>
      </profile>

      <roleManager enabled="false">
        <providers>
          <clear/>
          <add connectionStringName="ApplicationServices" applicationName="/" name="AspNetSqlRoleProvider" type="System.Web.Security.SqlRoleProvider,
System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"/>
          <add applicationName="/" name="AspNetWindowsTokenRoleProvider" type="System.Web.Security.WindowsTokenRoleProvider, System.Web, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"/>
        </providers>
      </roleManager>

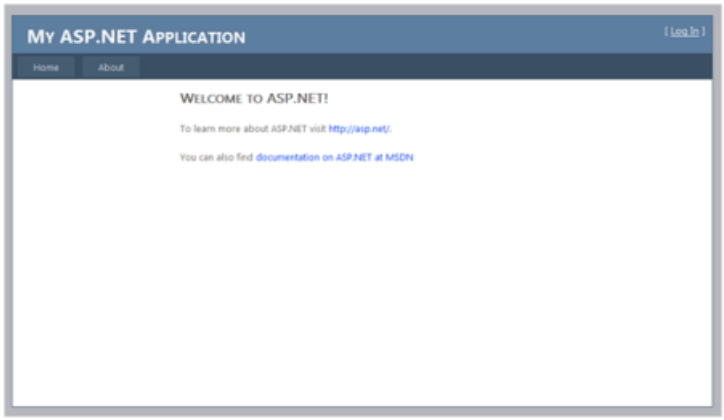
      <pages clientIDMode="Predictable">
      </pages>
    </system.web>
</configuration>
```

(Click to view full-size image)

The project also contains a second `Web.config` file in the `Account` directory. The second configuration file provides a way to secure access to the ChangePassword.aspx page for non-logged in users. The following example shows the contents of the second `Web.config` file.

```
<?xml version="1.0"?>
<configuration>
    <location path="ChangePassword.aspx">
      <system.web>
        <authorization>
          <deny users="?"/>
        </authorization>
      </system.web>
    </location>
</configuration>
```

The pages created by default in the new project templates also contain more content than in previous versions. The project contains a default master page and CSS file, and the default page (Default.aspx) is configured to use the master page by default. The result is that when you run the Web application or Web site for the first time, the default (home) page is already functional. In fact, it is similar to the default page you see when you start up a new MVC application.



(Click to view full-size image)

The intention of these changes to the project templates is to provide guidance on how to start building a new Web application. With semantically correct, strict XHTML 1.0-compliant markup and with layout that is specified using CSS, the pages in the templates represent best practices for building ASP.NET 4 Web applications. The default pages also have a two-column layout that you can easily customize.

For example, imagine that for a new Web Application you want to change some of the colors and insert your company logo in place of the My ASP.NET Application logo. To do this, you create a new directory under `Content` to store your logo image:

To add the image to the page, you then open the `Site.Master` file, find where the My ASP.NET Application text is defined, and replace it with an *image* element whose *src* attribute is set to the new logo image, as in the following example:

```
<div class="title">
    <img src="Content/Images/ASPNETLogo.jpg" width="376px" height="62px" border="0" alt="ASP.NET Logo" />
</div>
```

(Click to view full-size image)

You can then go into the Site.css file and modify CSS class definitions to change the background color of the page as well as that of the header.

The result of these changes is that you can display a customized home page with very little effort:

(Click to view full-size image)

CSS Improvements

One of the major areas of work in ASP.NET 4 has been to help render HTML that is compliant with the latest HTML standards. This includes changes to how ASP.NET Web server controls use CSS styles.

Compatibility Setting for Rendering

By default, when a Web application or Web site targets the .NET Framework 4, the *controlRenderingCompatibilityVersion* attribute of the *pages* element is set to "4.0". This element is defined in the machine-level `Web.config` file and by default applies to all ASP.NET 4 applications:

```
<system.web>
  <pages controlRenderingCompatibilityVersion="3.5|4.0"/>
</system.web>
```

The value for *controlRenderingCompatibility* is a string, which allows potential new version definitions in future releases. In the current release, the following values are supported for this property:

- "3.5". This setting indicates legacy rendering and markup. Markup rendered by controls is 100% backward compatible, and the setting of the *xhtmlConformance* property is honored.
- "4.0". If the property has this setting, ASP.NET Web server controls do the following:
- The *xhtmlConformance* property is always treated as "Strict". As a result, controls render XHTML 1.0 Strict markup.
- Disabling non-input controls no longer renders invalid styles.
- *div* elements around hidden fields are now styled so they do not interfere with user-created CSS rules.
- Menu controls render markup that is semantically correct and compliant with accessibility guidelines.
- Validation controls do not render inline styles.
- Controls that previously rendered border="0" (controls that derive from the ASP.NET *Table* control, and the ASP.NET *Image* control) no longer render this attribute.

Disabling Controls

In ASP.NET 3.5 SP1 and earlier versions, the framework renders the *disabled* attribute in the HTML markup for any control whose *Enabled* property set to *false*. However, according to the HTML 4.01 specification, only *input* elements should have this attribute.

In ASP.NET 4, you can set the *controlRenderingCompatibilityVersion* property to "3.5", as in the following example:

```
<system.web>
  <pages controlRenderingCompatibilityVersion="3.5"/>
</system.web>
```

You might create markup for a *Label* control like the following, which disables the control:

```
<asp:Label id="Label" runat="server" Text="Test" Enabled="false">
```

The *Label* control would render the following HTML:

```
<span id="Label1" disabled="disabled">Test</span>
```

In ASP.NET 4, you can set the *controlRenderingCompatibilityVersion* to "4.0". In that case, only controls that render *input* elements will render a *disabled* attribute when the control's *Enabled* property is set to *false*. Controls that do not render HTML *input* elements instead render a *class* attribute that references a CSS class that you can use to define a disabled look for the control. For example, the *Label* control shown in the earlier example would generate

the following markup:

```
<span id="Span1" class="aspnetdisabled">Test</span>
```

The default value for the class that specified for this control is "aspNetDisabled". However, you can change this default value by setting the static *DisabledCssClass* static property of the *WebControl* class. For control developers, the behavior to use for a specific control can also be defined using the *SupportsDisabledAttribute* property.

Hiding div Elements Around Hidden Fields

ASP.NET 2.0 and later versions render system-specific hidden fields (such as the *hidden* element used to store view state information) inside *div* element in order to comply with the XHTML standard. However, this can cause a problem when a CSS rule affects *div* elements on a page. For example, it can result in a one-pixel line appearing in the page around hidden *div* elements. In ASP.NET 4, *div* elements that enclose the hidden fields generated by ASP.NET add a CSS class reference as in the following example:

```
<div class="aspNetHidden">...</div>
```

You can then define a CSS class that applies only to the *hidden* elements that are generated by ASP.NET, as in the following example:

```
<style type="text/css">
  DIV# aspNetHidden {border:0;}

</style>
```

Rendering an Outer Table for Templated Controls

By default, the following ASP.NET Web server controls that support templates are automatically wrapped in an outer table that is used to apply inline styles:

- *FormView*
- *Login*
- *PasswordRecovery*
- *ChangePassword*
- *Wizard*
- *CreateUserWizard*

A new property named *RenderOuterTable* has been added to these controls that allows the outer table to be removed from the markup. For example, consider the following example of a *FormView* control:

```
<asp:FormView ID="FormView1" runat="server">
  <ItemTemplate>
      Content
  </ItemTemplate>

</asp:FormView>
```

This markup renders the following output to the page, which includes an HTML table:

```
<table cellspacing="0" border="0" id="Table1" style="border-collapse:collapse;">
  <tr>
    <td colspan="2">
      Content
    </td>
  </tr>

</table>
```

To prevent the table from being rendered, you can set the *FormView* control's *RenderOuterTable* property, as in the following example:

```
<asp:FormView ID="FormView1" runat="server" RenderOuterTable="false">
```

The previous example renders the following output, without the *table*, *tr*, and *td* elements:

```
Content
```

This enhancement can make it easier to style the content of the control with CSS, because no unexpected tags are rendered by the control.

> **NOTE**
>
> Note This change disables support for the auto-format function in the Visual Studio 2010 designer, because there is no longer a *table* element that can host style attributes that are generated by the auto-format option.

ListView Control Enhancements

The *ListView* control has been made easier to use in ASP.NET 4. The earlier version of the control required that you specify a layout template that contained a server control with a known ID. The following markup shows a typical example of how to use the *ListView* control in ASP.NET 3.5.

```
<asp:ListView ID="ListView1" runat="server">
  <LayoutTemplate>
    <asp:PlaceHolder ID="ItemPlaceHolder" runat="server"></asp:PlaceHolder>
  </LayoutTemplate>
  <ItemTemplate>
    <% Eval("LastName")%>
  </ItemTemplate>
</asp:ListView>
```

In ASP.NET 4, the *ListView* control does not require a layout template. The markup shown in the previous example can be replaced with the following markup:

```
<asp:ListView ID="ListView1" runat="server">
  <ItemTemplate>
    <% Eval("LastName")%>
  </ItemTemplate>

</asp:ListView>
```

CheckBoxList and RadioButtonList Control Enhancements

In ASP.NET 3.5, you can specify layout for the *CheckBoxList* and *RadioButtonList* using the following two settings:

- *Flow*. The control renders *span* elements to contain its content.

- *Table*. The control renders a *table* element to contain its content.

The following example shows markup for each of these controls.

```
<asp:CheckBoxList ID="CheckBoxList1" runat="server" RepeatLayout="Flow">
  <asp:ListItem Text="CheckBoxList" Value="cbl" />
</asp:CheckBoxList>

<asp:RadioButtonList runat="server" RepeatLayout="Table">
  <asp:ListItem Text="RadioButtonList" Value="rbl" />
</asp:RadioButtonList>
```

By default, the controls render HTML similar to the following:

```
<span id="Span2"><input id="CheckBoxList1_0" type="checkbox"
    name="CheckBoxList1$0" /><label for="CheckBoxList1_0">CheckBoxList</label></span>

<table id="RadioButtonList1" border="0">
  <tr>
    <td><input id="RadioButtonList1_0" type="radio" name="RadioButtonList1" value="rbl" /><label
for="RadioButtonList1_0">RadioButtonList</label></td>
  </tr>

</table>
```

Because these controls contain lists of items, to render semantically correct HTML, they should render their contents using HTML list (*li*) elements. This makes it easier for users who read Web pages using assistive technology, and makes the controls easier to style using CSS.

In ASP.NET 4, the *CheckBoxList* and *RadioButtonList* controls support the following new values for the *RepeatLayout* property:

- *OrderedList* – The content is rendered as *li* elements within an *ol* element.
- *UnorderedList* – The content is rendered as *li* elements within a *ul* element.

The following example shows how to use these new values.

```
<asp:CheckBoxList ID="CheckBoxList1" runat="server"
      RepeatLayout="OrderedList">
  <asp:ListItem Text="CheckBoxList" Value="cbl" />
</asp:CheckBoxList>

<asp:RadioButtonList ID="RadioButtonList1" runat="server"
      RepeatLayout="UnorderedList">
  <asp:ListItem Text="RadioButtonList" Value="rbl" />
</asp:RadioButtonList>
```

The preceding markup generates the following HTML:

```
<ol id="CheckBoxList1">

  <li><input id="CheckBoxList1_0" type="checkbox" name="CheckBoxList1$0" value="cbl" /><label
for="CheckBoxList1_0">CheckBoxList</label></li>
</ol>

<ul id="RadioButtonList1">
  <li><input id="RadioButtonList1_0" type="radio" name="RadioButtonList1" value="rbl" /><label
for="RadioButtonList1_0">RadioButtonList</label></li>

</ul>
```

NOTE

Note If you set *RepeatLayout* to *OrderedList* or *UnorderedList*, the *RepeatDirection* property can no longer be used and will throw an exception at run time if the property has been set within your markup or code. The property would have no value because the visual layout of these controls is defined using CSS instead.

Menu Control Improvements

Before ASP.NET 4, the *Menu* control rendered a series of HTML tables. This made it more difficult to apply CSS styles outside of setting inline properties and was also not compliant with accessibility standards.

In ASP.NET 4, the control now renders HTML using semantic markup that consists of an unordered list and list elements. The following example shows markup in an ASP.NET page for the *Menu* control.

```
<asp:Menu ID="Menu1" runat="server">
  <Items> <asp:MenuItem Text="Home" Value="Home" />
    <asp:MenuItem Text="About" Value="About" />
  </Items>

</asp:Menu>
```

When the page renders, the control produces the following HTML (the *onclick* code has been omitted for clarity):

```
<div id="Menu1">
  <ul>
    <li><a href="#" onclick="...">Home</a></li>
    <li><a href="#" onclick="...">About</a></li>
  </ul>

</div>
<script type="text/javascript">
  new Sys.WebForms.Menu('Menu1');
</script>
```

In addition to rendering improvements, keyboard navigation of the menu has been improved using focus management. When the *Menu* control gets focus, you can use the arrow keys to navigate elements. The *Menu* control also now attaches accessible rich internet applications (ARIA) roles and attributes following thefor improved accessibility.

Styles for the menu control are rendered in a style block at the top of the page, rather than in line with the rendered HTML elements. If you want to take full control over the styling for the control, you can set the new *IncludeStyleBlock* property to *false*, in which case the style block is not emitted. One way to use this property is to use the auto-format feature in the Visual Studio designer to set the appearance of the menu. You can then run the page, open the page source, and then copy the rendered style block to an external CSS file. In Visual Studio, undo the styling and set *IncludeStyleBlock* to *false*. The result is that the menu appearance is defined using styles in an

external style sheet.

Wizard and CreateUserWizard Controls

The ASP.NET *Wizard* and *CreateUserWizard* controls support templates that let you define the HTML that they render. (*CreateUserWizard* derives from *Wizard*.) The following example shows the markup for a fully templated *CreateUserWizard* control:

```
<asp:CreateUserWizard ID="CreateUserWizard1" runat="server" ActiveStepIndex="0">
  <HeaderTemplate>
  </HeaderTemplate>

  <SideBarTemplate>
  </SideBarTemplate>

  <StepNavigationTemplate>
  </StepNavigationTemplate>

  <StartNavigationTemplate>
  </StartNavigationTemplate>

  <FinishNavigationTemplate>
  </FinishNavigationTemplate>

  <WizardSteps>
    <asp:CreateUserWizardStep ID="CreateUserWizardStep1" runat="server">
      <ContentTemplate>
      </ContentTemplate>

      <CustomNavigationTemplate>
      </CustomNavigationTemplate>

    </asp:CreateUserWizardStep>

    <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">
      <ContentTemplate>
      </ContentTemplate>
    </asp:CompleteWizardStep>

  </WizardSteps>

</asp:CreateUserWizard>
```

The control renders HTML similar to the following:

```
<table cellspacing="0" cellpadding="0" border="0" id="CreateUserWizard1" style="border-collapse:collapse;">
  <tr>
    <td>Header</td>
  </tr>
  <tr style="height:100%;">
    <td>
      <table cellspacing="0" cellpadding="0" border="0" style="height:100%;width:100%;border-
collapse:collapse;">
        <tr>
          <td style="height:100%;width:100%;"></td>
        </tr>
      </table>
    </td>
  </tr>
  <tr>
    <td align="right"></td>
  </tr>

</table>
```

In ASP.NET 3.5 SP1, although you can change the template contents, you still have limited control over the output of the *Wizard* control. In ASP.NET 4, you can create a *LayoutTemplate* template and insert *PlaceHolder* controls (using reserved names) to specify how you want the *Wizard control* to render. The following example shows this:

```
<asp:CreateUserWizard ID="CreateUserWizard1" runat="server" ActiveStepIndex="1">
  <LayoutTemplate>
    <asp:PlaceHolder ID="headerPlaceholder" runat="server" />
    <asp:PlaceHolder ID="sideBarPlaceholder" runat="server" />
    <asp:PlaceHolder id="wizardStepPlaceholder" runat="server" />
    <asp:PlaceHolder id="navigationPlaceholder" runat="server" />
  </LayoutTemplate>
  <HeaderTemplate>
    Header
  </HeaderTemplate>
  <WizardSteps>
    <asp:CreateUserWizardStep runat="server">
      <ContentTemplate>
      </ContentTemplate>
    </asp:CreateUserWizardStep>
    <asp:CompleteWizardStep runat="server">
      <ContentTemplate>
      </ContentTemplate>
    </asp:CreateUserWizardStep>
  </WizardSteps>

</asp:CreateUserWizard>
```

The example contains the following named placeholders in the *LayoutTemplate* element:

- *headerPlaceholder* – At run time, this is replaced by the contents of the *HeaderTemplate* element.
- *sideBarPlaceholder* – At run time, this is replaced by the contents of the *SideBarTemplate* element.
- *wizardStepPlaceHolder* – At run time, this is replaced by the contents of the *WizardStepTemplate* element.
- *navigationPlaceholder* – At run time, this is replaced by any navigation templates that you have defined.

The markup in the example that uses placeholders renders the following HTML (without the content actually defined in the templates):

```
<span>
</span>
```

The only HTML that is now not user-defined is a *span* element. (We anticipate that in future releases, even the *span* element will not be rendered.) This now gives you full control over virtually all the content that is generated by the *Wizard* control.

ASP.NET MVC

ASP.NET MVC was introduced as an add-on framework to ASP.NET 3.5 SP1 in March 2009. Visual Studio 2010 includes ASP.NET MVC 2, which includes new features and capabilities.

Areas Support

Areas let you group controllers and views into sections of a large application in relative isolation from other sections. Each area can be implemented as a separate ASP.NET MVC project that can then be referenced by the main application. This helps manage complexity when you build a large application and makes it easier for multiple teams to work together on a single application.

Data-Annotation Attribute Validation Support

DataAnnotations attributes let you attach validation logic to a model by using metadata attributes. *DataAnnotations* attributes were introduced in ASP.NET Dynamic Data in ASP.NET 3.5 SP1. These attributes have

been integrated into the default model binder and provide a metadata-driven means to validate user input.

Templated Helpers

Templated helpers let you automatically associate edit and display templates with data types. For example, you can use a template helper to specify that a date-picker UI element is automatically rendered for a *System.DateTime* value. This is similar to field templates in ASP.NET Dynamic Data.

The *Html.EditorFor* and *Html.DisplayFor* helper methods have built-in support for rendering standard data types as well as complex objects with multiple properties. They also customize rendering by letting you apply data-annotation attributes like *DisplayName* and *ScaffoldColumn* to the *ViewModel* object.

Often you want to customize the output from UI helpers even further and have complete control over what is generated. The *Html.EditorFor* and *Html.DisplayFor* helper methods support this using a templating mechanism that lets you define external templates that can override and control the output rendered. The templates can be rendered individually for a class.

Dynamic Data

Dynamic Data was introduced in the .NET Framework 3.5 SP1 release in mid-2008. This feature provides many enhancements for creating data-driven applications, including the following:

- A RAD experience for quickly building a data-driven Web site.
- Automatic validation that is based on constraints defined in the data model.
- The ability to easily change the markup that is generated for fields in the *GridView* and *DetailsView* controls by using field templates that are part of your Dynamic Data project.

> **NOTE**
>
> Note For more information, see the Dynamic Data documentation in the MSDN Library.

For ASP.NET 4, Dynamic Data has been enhanced to give developers even more power for quickly building data-driven Web sites.

Enabling Dynamic Data for Existing Projects

Dynamic Data features that shipped in the .NET Framework 3.5 SP1 brought new features such as the following:

- Field templates – These provide data-type-based templates for data-bound controls. Field templates provide a simpler way to customize the look of data controls than using template fields for each field.
- Validation – Dynamic Data lets you use attributes on data classes to specify validation for common scenarios like required fields, range checking, type checking, pattern matching using regular expressions, and custom validation. Validation is enforced by the data controls.

However, these features had the following requirements:

- The data-access layer had to be based on Entity Framework or LINQ to SQL.
- The only data source controls supported for these features were the *EntityDataSource* or *LinqDataSource* controls.
- The features required a Web project that had been created using the Dynamic Data or Dynamic Data Entities templates in order to have all the files that were required to support the feature.

A major goal of Dynamic Data support in ASP.NET 4 is to enable the new functionality of Dynamic Data for any ASP.NET application. The following example shows markup for controls that can take advantage of Dynamic Data functionality in an existing page.

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="True"
    DataKeyNames="ProductID" DataSourceID="LinqDataSource1">
</asp:GridView>
<asp:LinqDataSource ID="LinqDataSource1" runat="server"
    ContextTypeName="DataClassesDataContext" EnableDelete="True" EnableInsert="True"
    EnableUpdate="True" TableName="Products">

</asp:LinqDataSource>
```

In the code for the page, the following code must be added in order to enable Dynamic Data support for these controls:

```
GridView1.EnableDynamicData(typeof(Product));
```

When the *GridView* control is in edit mode, Dynamic Data automatically validates that the data entered is in the proper format. If it is not, an error message is displayed.

This functionality also provides other benefits, such as being able to specify default values for insert mode. Without Dynamic Data, to implement a default value for a field, you must attach to an event, locate the control (using *FindControl*), and set its value. In ASP.NET 4, the *EnableDynamicData* call supports a second parameter that lets you pass default values for any field on the object, as shown in this example:

```
DetailsView1.EnableDynamicData(typeof(Product), new { ProductName = "DefaultName" });
```

Declarative DynamicDataManager Control Syntax

The *DynamicDataManager* control has been enhanced so that you can configure it declaratively, as with most controls in ASP.NET, instead of only in code. The markup for the *DynamicDataManager* control looks like the following example:

```
<asp:DynamicDataManager ID="DynamicDataManager1" runat="server"
    AutoLoadForeignKeys="true">
  <DataControls>
    <asp:DataControlReference ControlID="GridView1" />
  </DataControls>

</asp:DynamicDataManager>
<asp:GridView id="GridView1" runat="server"
</asp:GridView>
```

This markup enables Dynamic Data behavior for the GridView1 control that is referenced in the *DataControls* section of the *DynamicDataManager* control.

Entity Templates

Entity templates offer a new way to customize the layout of data without requiring you to create a custom page. Page templates use the *FormView* control (instead of the *DetailsView* control, as used in page templates in earlier versions of Dynamic Data) and the *DynamicEntity* control to render Entity templates. This gives you more control over the markup that is rendered by Dynamic Data.

The following list shows the new project directory layout that contains entity templates:

```
\DynamicData\EntityTemplates
\DynamicData\EntityTemplates\Default.ascx
\DynamicData\EntityTemplates\Default_Edit.ascx
\DynamicData\EntityTemplates\Default_Insert.ascx
```

The `EntityTemplate` directory contains templates for how to display data model objects. By default, objects are rendered by using the `Default.ascx` template, which provides markup that looks just like the markup created by the *DetailsView* control used by Dynamic Data in ASP.NET 3.5 SP1. The following example shows the markup for the `Default.ascx` control:

```
<asp:EntityTemplate runat="server" ID="TemplateContainer1">
  <ItemTemplate>
    <tr
      <td>
        <asp:Label ID="Label1" runat="server" OnInit="Label_Init" />
      </td>
      <td>
        <asp:DynamicControl runat="server" OnInit="DynamicControl_Init" />
      </td>
    </tr>
  </ItemTemplate>

</asp:EntityTemplate>
```

The default templates can be edited to change the look and feel for the entire site. There are templates for display, edit, and insert operations. New templates can be added based on the name of the data object in order to change the look and feel of just one type of object. For example, you can add the following template:

```
\DynamicData\EntityTemplates\Products.aspx
```

The template might contain the following markup:

```
<tr>
  <td>Name</td>
  <td><asp:DynamicControl runat="server" DataField="ProductName" /></td>
  <td>Category</td>
  <td><asp:DynamicControl runat="server" DataField="Category" /></td>

</tr>
```

The new entity templates are displayed on a page by using the new *DynamicEntity* control. At run time, this control is replaced with the contents of the entity template. The following markup shows the *FormView* control in the `Detail.aspx` page template that uses the entity template. Notice the *DynamicEntity* element in the markup.

```
<asp:FormView runat="server" ID="FormView1"
    DataSourceID="DetailsDataSource"
    OnItemDeleted="FormView1_ItemDeleted">
  <ItemTemplate>
    <table class="DDDetailsTable" cellpadding="6">
      <asp:DynamicEntity runat="server" />
      <tr class="td">
        <td colspan="2">
          <asp:DynamicHyperLink ID="EditHyperLink" runat="server"
              Action="Edit" Text="Edit" />
          <asp:LinkButton ID="DeleteLinkButton" runat="server"
              CommandName="Delete"
              CausesValidation="false"
              OnClientClick='return confirm("Are you sure you want to delete this item?");'
              Text="Delete" />
        </td>
      </tr>
    </table>
  </ItemTemplate>

</asp:FormView>
```

New Field Templates for URLs and Email Addresses

ASP.NET 4 introduces two new built-in field templates, `EmailAddress.ascx` and `Url.ascx`. These templates are used for fields that are marked as *EmailAddress* or *Url* with the *DataType* attribute. For *EmailAddress* objects, the field is displayed as a hyperlink that is created by using the *mailto:* protocol. When users click the link, it opens the user's email client and creates a skeleton message. Objects typed as *Url* are displayed as ordinary hyperlinks.

The following example shows how fields would be marked.

```
[DataType(DataType.EmailAddress)]
public object HomeEmail { get; set; }

[DataType(DataType.Url)]
public object Website { get; set; }
```

Creating Links with the DynamicHyperLink Control

Dynamic Data uses the new routing feature that was added in the .NET Framework 3.5 SP1 to control the URLs that end users see when they access the Web site. The new *DynamicHyperLink* control makes it easy to build links to pages in a Dynamic Data site. The following example shows how to use the *DynamicHyperLink* control:

```
<asp:DynamicHyperLink ID="ListHyperLink" runat="server"
    Action="List" TableName="Products">
    Show all products
</asp:DynamicHyperLink>
```

This markup creates a link that points to the List page for the `Products` table based on routes that are defined in the `Global.asax` file. The control automatically uses the default table name that the Dynamic Data page is based on.

Support for Inheritance in the Data Model

Both the Entity Framework and LINQ to SQL support inheritance in their data models. An example of this might be a database that has an `InsurancePolicy` table. It might also contain `CarPolicy` and `HousePolicy` tables that have the same fields as `InsurancePolicy` and then add more fields. Dynamic Data has been modified to understand inherited objects in the data model and to support scaffolding for the inherited tables.

Support for Many-to-Many Relationships (Entity Framework Only)

The Entity Framework has rich support for many-to-many relationships between tables, which is implemented by exposing the relationship as a collection on an *Entity* object. New `ManyToMany.ascx` and `ManyToMany_Edit.ascx` field templates have been added to provide support for displaying and editing data that is involved in many-to-many relationships.

New Attributes to Control Display and Support Enumerations

The *DisplayAttribute* has been added to give you additional control over how fields are displayed. The *DisplayName* attribute in earlier versions of Dynamic Data allowed you to change the name that is used as a caption for a field. The new *DisplayAttribute* class lets you specify more options for displaying a field, such as the order in which a field is displayed and whether a field will be used as a filter. The attribute also provides independent control of the name used for the labels in a *GridView* control, the name used in a *DetailsView* control, the help text for the field, and the watermark used for the field (if the field accepts text input).

The *EnumDataTypeAttribute* class has been added to let you map fields to enumerations. When you apply this attribute to a field, you specify an enumeration type. Dynamic Data uses the new `Enumeration.ascx` field template to create UI for displaying and editing enumeration values. The template maps the values from the database to the names in the enumeration.

Enhanced Support for Filters

Dynamic Data 1.0 shipped with built-in filters for Boolean columns and foreign-key columns. The filters did not allow you to specify whether they were displayed or in what order they were displayed. The new *DisplayAttribute* attribute addresses both of these issues by giving you control over whether a column is displayed as a filter and in what order it will be displayed.

An additional enhancement is that filtering support has been rewritten to use the new feature of Web Forms. This lets you create filters without requiring knowledge of the data source control that the filters will be used with. Along with these extensions, filters have also been turned into template controls, which allows you to add new ones. Finally, the *DisplayAttribute* class mentioned earlier allows the default filter to be overridden, in the same way that *UIHint* allows the default field template for a column to be overridden.

Visual Studio 2010 Web Development Improvements

Web development in Visual Studio 2010 has been enhanced for greater CSS compatibility, increased productivity through HTML and ASP.NET markup snippets and new dynamic IntelliSense JavaScript.

Improved CSS Compatibility

The Visual Web Developer designer in Visual Studio 2010 has been updated to improve CSS 2.1 standards compliance. The designer better preserves integrity of the HTML source and is more robust than in previous versions of Visual Studio. Under the hood, architectural improvements have also been made that will better enable future enhancements in rendering, layout, and serviceability.

HTML and JavaScript Snippets

In the HTML editor, IntelliSense auto-completes tag names. The IntelliSense Snippets feature auto-completes entire tags and more. In Visual Studio 2010, IntelliSense snippets are supported for JavaScript, alongside C# and Visual Basic, which were supported in earlier versions of Visual Studio.

Visual Studio 2010 includes over 200 snippets that help you auto-complete common ASP.NET and HTML tags, including required attributes (such as runat="server") and common attributes specific to a tag (such as *ID*, *DataSourceID*, *ControlToValidate*, and *Text*).

You can download additional snippets, or you can write your own snippets that encapsulate the blocks of markup that you or your team use for common tasks.

JavaScript IntelliSense Enhancements

In Visual 2010, JavaScript IntelliSense has been redesigned to provide an even richer editing experience.

IntelliSense now recognizes objects that have been dynamically generated by methods such as *registerNamespace* and by similar techniques used by other JavaScript frameworks. Performance has been improved to analyze large libraries of script and to display IntelliSense with little or no processing delay. Compatibility has been dramatically increased to support nearly all third-party libraries and to support diverse coding styles. Documentation comments are now parsed as you type and are immediately leveraged by IntelliSense.

Web Application Deployment with Visual Studio 2010

When ASP.NET developers deploy a Web application, they often find that they encounter issues such as the following:

- Deploying to a shared hosting site requires technologies such as FTP, which can be slow. In addition, you must manually perform tasks such as running SQL scripts to configure a database and you must change IIS settings, such as configuring a virtual directory folder as an application.
- In an enterprise environment, in addition to deploying the Web application files, administrators frequently must modify ASP.NET configuration files and IIS settings. Database administrators must run a series of SQL scripts to get the application database running. Such installations are labor intensive, often take hours to complete, and must be carefully documented.

Visual Studio 2010 includes technologies that address these issues and that let you seamlessly deploy Web applications. One of these technologies is the IIS Web Deployment Tool (MsDeploy.exe).

Web deployment features in Visual Studio 2010 include the following major areas:

- Web packaging
- Web.config transformation
- Database deployment
- One-click publish for Web applications

The following sections provide details about these features.

Web Packaging

Visual Studio 2010 uses the MSDeploy tool to create a compressed (.zip) file for your application, which is referred to as a *Web package*. The package file contains metadata about your application plus the following content:

- IIS settings, which includes application pool settings, error page settings, and so on.
- The actual Web content, which includes Web pages, user controls, static content (images and HTML files), and so on.
- SQL Server database schemas and data.
- Security certificates, components to install in the GAC, registry settings, and so on.

A Web package can be copied to any server and then installed manually by using IIS Manager. Alternatively, for automated deployment, the package can be installed by using command-line commands or by using deployment APIs.

Visual Studio 2010 provides built in MSBuild tasks and targets to create Web packages. For more information, see ASP.NET Web Application Project Deployment Overview on the MSDN Web site and 10 + 20 reasons why you should create a Web Package on Vishal Joshi's blog.

Web.config Transformation

For Web application deployment, Visual Studio 2010 introduces XML Document Transform (XDT), which is a feature that lets you transform a `Web.config` file from development settings to production settings. Transformation settings are specified in transform files named `web.debug.config`, `web.release.config`, and so on. (The names of these files match MSBuild configurations.) A transform file includes just the changes that you need to make to a deployed `Web.config` file. You specify the changes by using simple syntax.

The following example shows a portion of a `web.release.config` file that might be produced for deployment of your release configuration. The Replace keyword in the example specifies that during deployment the *connectionString* node in the `Web.config` file will be replaced with the values that are listed in the example.

```
<connectionStrings xdt:Transform="Replace">
  <add name="BlogDB" connectionString="connection string detail]" />

</connectionStrings>
```

For more information, see Web.config Transformation Syntax for Web Application Project Deployment on the MSDN Web site andWeb Deployment: Web.Config Transformation on Vishal Joshi's blog.

Database Deployment

A Visual Studio 2010 deployment package can include dependencies on SQL Server databases. As part of the package definition, you provide the connection string for your source database. When you create the Web package, Visual Studio 2010 creates SQL scripts for the database schema and optionally for the data, and then adds these to the package. You can also provide custom SQL scripts and specify the sequence in which they should run on the server. At deployment time, you provide a connection string that is appropriate for the target server; the deployment process then uses this connection string to run the scripts that create the database schema and add the data.

In addition, by using one-click publish, you can configure deployment to publish your database directly when the application is published to a remote shared hosting site. For more information, see How to: Deploy a Database With a Web Application Project on the MSDN Web site and Database Deployment with VS 2010 on Vishal Joshi's blog.

One-Click Publish for Web Applications

Visual Studio 2010 also lets you use the IIS remote management service to publish a Web application to a remote server. You can create a publish profile for your hosting account or for testing servers or staging servers. Each profile can save appropriate credentials securely. You can then deploy to any of the target servers with one click by using the Web one-click publish toolbar. With Visual Studio 2010, you can also publish by using the MSBuild command line. This lets you configure your team build environment to include publishing in a continuous-integration model.

For more information, see How to: Deploy a Web Application Project Using One-Click Publish and Web Deploy on the MSDN Web site and Web 1-Click Publish with VS 2010 on Vishal Joshi's blog. To view video presentations about Web application deployment in Visual Studio 2010, see VS 2010 for Web Developer Previews on Vishal Joshi's blog.

Resources

The following Web sites provide additional information about ASP.NET 4 and Visual Studio 2010.

- ASP.NET 4 — The official documentation for ASP.NET 4 on the MSDN Web site.
- https://www.asp.net/ — The ASP.NET team's own Web site.
- https://www.asp.net/dynamicdata/ and ASP.NET Dynamic Data Content Map — Online resources on the ASP.NET team site and in the official documentation for ASP.NET Dynamic Data.
- https://www.asp.net/ajax/ — The main Web resource for ASP.NET Ajax development.
- https://blogs.msdn.com/webdevtools/ — The Visual Web Developer Team blog, which includes information about features in Visual Studio 2010.
- ASP.NET WebStack — The main Web resource for preview releases of ASP.NET.

Disclaimer

This is a preliminary document and may be changed substantially prior to final commercial release of the software

described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, email addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2009 Microsoft Corporation. All rights reserved.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

ASP.NET 4 Breaking Changes

2/20/2020 • 30 minutes to read • Edit Online

This document describes changes that have been made for the .NET Framework version 4 release that can potentially affect applications that were created using earlier releases, including the ASP.NET 4 Beta 1 and Beta 2 releases.

Download This Whitepaper

Contents

ControlRenderingCompatibilityVersion Setting in the Web.config File
ClientIDMode Changes
HtmlEncode and UrlEncode Now Encode Single Quotation Marks
ASP.NET Page (.aspx) Parser is Stricter
Browser Definition Files Updated
System.Web.Mobile.dll Removed from Root Web Configuration File
ASP.NET Request Validation
Default Hashing Algorithm Is Now HMACSHA256
Configuration Errors Related to New ASP.NET 4 Root Configuration
ASP.NET 4 Child Applications Fail to Start When Under ASP.NET 2.0 or ASP.NET 3.5 Applications
ASP.NET 4 Web Sites Fail to Start on Computers Where SharePoint Is Installed
The HttpRequest.FilePath Property No Longer Includes PathInfo Values
ASP.NET 2.0 Applications Might Generate HttpException Errors that Reference eurl.axd
Event Handlers Might Not Be Not Raised in a Default Document in IIS 7 or IIS 7.5 Integrated Mode
Changes to the ASP.NET Code Access Security (CAS) Implementation
MembershipUser and Other Types in the System.Web.Security Namespace Have Been Moved
Output Caching Changes to Vary * HTTP Header
System.Web.Security Types for Passport are Obsolete
The MenuItem.PopOutImageUrl Property Fails to Render an Image in ASP.NET 4
Menu.StaticPopOutImageUrl and Menu.DynamicPopOutImageUrl Fail to Render Images When Paths Contain Backslashes
Disclaimer

ControlRenderingCompatibilityVersion Setting in the Web.config File

ASP.NET controls have been modified in the .NET Framework version 4 in order to let you specify more precisely how they render markup. In previous versions of the .NET Framework, some controls emitted markup that you had no way to disable. By default, ASP.NET 4 this type of markup is no longer generated.

If you use Visual Studio 2010 to upgrade your application from ASP.NET 2.0 or ASP.NET 3.5, the tool automatically adds a setting to the `Web.config` file that preserves legacy rendering. However, if you upgrade an application by changing the application pool in IIS to target the .NET Framework 4, ASP.NET uses the new rendering mode by default. To disable the new rendering mode, add the following setting in the `Web.config` file:

```
<pages controlRenderingCompatibilityVersion="3.5" />
```

The major rendering changes that the new behavior brings are as follows:

- The **Image** and **ImageButton** controls no longer render a `border="0"` attribute.
- The **BaseValidator** class and validation controls that derive from it no longer render red text by default.
- The **HtmlForm** control does not render a **name** attribute.
- The **Table** control no longer renders a `border="0"` attribute.
- Controls that are not designed for user input (for example, the **Label** control) no longer render the `disabled="disabled"` attribute if their **Enabled** property is set to **false** (or if they inherit this setting from a container control).

ClientIDMode Changes

The **ClientIDMode** setting in ASP.NET 4 lets you specify how ASP.NET generates the **id** attribute for HTML elements. In previous versions of ASP.NET, the default behavior was equivalent to the **AutoID** setting of **ClientIDMode**. However, the default setting is now **Predictable**.

If you use Visual Studio 2010 to upgrade your application from ASP.NET 2.0 or ASP.NET 3.5, the tool automatically adds a setting to the `Web.config` file that preserves the behavior of earlier versions of the .NET Framework. However, if you upgrade an application by changing the application pool in IIS to target the .NET Framework 4, ASP.NET uses the new mode by default. To disable the new client ID mode, add the following setting in the `Web.config` file:

```
<pages clientIDMode="AutoID" />
```

HtmlEncode and UrlEncode Now Encode Single Quotation Marks

In ASP.NET 4, the **HtmlEncode** and **UrlEncode** methods of the **HttpUtility** and **HttpServerUtility** classes have been updated to encode the single quotation mark character (') as follows:

- The **HtmlEncode** method encodes instances of the single quotation mark as ' .
- The **UrlEncode** method encodes instances of the single quotation mark as %27.

ASP.NET Page (.aspx) Parser is Stricter

The page parser for ASP.NET pages (`.aspx` files) and user controls (`.ascx` files) is stricter in ASP.NET 4 and will reject more instances of invalid markup. For example, the following two snippets would successfully parse in earlier releases of ASP.NET, but will now raise parser errors in ASP.NET 4.

```
<asp:HiddenField runat="server" ID="SomeControl" Value="sampleValue"  ; />
```

Notice the invalid semicolon at the end of the **HiddenField** tag.

```
<asp:LinkButton runat="server" ID="SomeControl" onclick="someControlClicked"
      style="display:inline;  CssClass="searchLink"  />
```

Notice the unclosed **style** attribute that runs into the **CssClass** attribute.

Browser Definition Files Updated

The browser definition files have been updated to include information about new and updated browsers and devices. Older browsers and devices such as Netscape Navigator have been removed, and newer browsers and devices such as Google Chrome and Apple iPhone have been added.

If your application contains custom browser definitions that inherit from one of the browser definitions that have

been removed, you will see an error. For example, if the `App_Browsers` folder contains a browser definition that inherits from the IE2 browser definition, you will receive the following configuration error message:

- The browser or gateway element with ID 'IE2' cannot be found.

> **NOTE**
>
> The **HttpBrowserCapabilities** object (which is exposed by the page's **Request.Browser** property) is driven by the browser definitions files. Therefore, the information returned by accessing a property of this object in ASP.NET 4 might be different than the information returned in an earlier version of ASP.NET.

You can revert to the old browser definition files by copying the browser definition files from the following folder:

```
Windows\Microsoft.NET\Framework\v2.0.50727\CONFIG\Browsers
```

Copy the files into the corresponding `\CONFIG\Browsers` folder for ASP.NET 4. After you copy the files, run the Aspnet_regbrowsers.exe command-line tool.

System.Web.Mobile.dll Removed from Root Web Configuration File

In previous versions of ASP.NET, a reference to the System.Web.Mobile.dll assembly was included in the root `Web.config` file in the **assemblies** section under. In order to improve performance, the reference to this assembly was removed.

The System.Web.Mobile.dll assembly is included in ASP.NET 4, but it is deprecated. If you want to use types from the System.Web.Mobile.dll assembly, add a reference to this assembly to either the root `Web.config` file or to an application `Web.config` file. For example, if you want to use any of the (deprecated) ASP.NET mobile controls, you must add a reference to the System.Web.Mobile.dll assembly to the `Web.config` file.

ASP.NET Request Validation

The request validation feature in ASP.NET provides a certain level of default protection against cross-site scripting (XSS) attacks. In previous versions of ASP.NET, request validation was enabled by default. However, it applied only to ASP.NET pages (`.aspx` files and their class files) and only when those pages were executing.

In ASP.NET 4, by default, request validation is enabled for all requests, because it is enabled before the **BeginRequest** phase of an HTTP request. As a result, request validation applies to requests for all ASP.NET resources, not just .aspx page requests. This includes requests such as Web service calls and custom HTTP handlers. Request validation is also active when custom HTTP modules are reading the contents of an HTTP request.

As a result, request validation errors might now occur for requests that previously did not trigger errors. To revert to the behavior of the ASP.NET 2.0 request validation feature, add the following setting in the `Web.config` file:

```
<httpRuntime requestValidationMode="2.0" />
```

However, we recommend that you analyze any request validation errors to determine whether existing handlers, modules, or other custom code accesses potentially unsafe HTTP inputs that could be XSS attack vectors.

Default Hashing Algorithm Is Now HMACSHA256

ASP.NET uses both encryption and hashing algorithms to help secure data such as forms authentication cookies and view state. By default, ASP.NET 4 now uses the HMACSHA256 algorithm for hash operations on cookies and

view state. Earlier versions of ASP.NET used the older HMACSHA1 algorithm.

Your applications might be affected if you run mixed ASP.NET 2.0/ASP.NET 4 environments where data such as forms authentication cookies must work across.NET Framework versions. To configure an ASP.NET 4 Web application to use the older HMACSHA1 algorithm, add the following setting in the `Web.config` file:

```
<machineKey validation="SHA1" />
```

Configuration Errors Related to New ASP.NET 4 Root Configuration

The root configuration files (the `machine.config` file and the root `Web.config` file) for the .NET Framework 4 (and therefore ASP.NET 4) have been updated to include most of the boilerplate configuration information that in ASP.NET 3.5 was found in the application `Web.config` files. Because of the complexity of the managed IIS 7 and IIS 7.5 configuration systems, running ASP.NET 3.5 applications under ASP.NET 4 and under IIS 7 and IIS 7.5 can result in either ASP.NET or IIS configuration errors.

We recommend that you upgrade ASP.NET 3.5 applications to ASP.NET 4 by using the project upgrade tools in Visual Studio 2010, if practical. Visual Studio 2010 automatically modifies the ASP.NET 3.5 application's `Web.config` file to contain the appropriate settings for ASP.NET 4.

However, it is a supported scenario to run ASP.NET 3.5 applications using the .NET Framework 4 without recompilation. In that case, you might have to manually modify the application's `Web.config` file before you run the application under the .NET Framework 4 and under IIS 7 or IIS 7.5.

The next two sections describe changes that you might need to make for different combinations of software.

Windows Vista SP1 or Windows Server 2008 SP1, where neither hotfix KB958854 nor SP2 are installed. In this configuration, the IIS 7 configuration system incorrectly merges an application's managed configuration by comparing the application-level `Web.config` file to the ASP.NET 2.0 `machine.config` files. Because of this, application-level `Web.config` files from the .NET Framework 3.5 or later must have a **system.web.extensions** configuration section definition (the element) in order not to cause an IIS 7 validation failure.

However, manually modified application-level `Web.config` file entries that do not precisely match the original boilerplate configuration section definitions that were introduced with Visual Studio 2008 will cause ASP.NET configuration errors. (The default configuration entries that are generated by Visual Studio 2008 work correctly.) A common problem is that manually modified `Web.config` files leave out the **allowDefinition** and **requirePermission** configuration attributes that are found on various configuration section definitions. This causes a mismatch between the abbreviated configuration section in application-level `Web.config` files and the complete definition in the ASP.NET 4 `machine.config` file. As a result, at run time, the ASP.NET 4 configuration system throws a configuration error.

Windows Vista SP2, Windows Server 2008 SP2, Windows 7, Windows Server 2008 R2, and also Windows Vista SP1 and Windows Server 2008 SP1 where hotfix KB958854 is installed.

In this scenario, the IIS 7 and IIS 7.5 native configuration system returns a configuration error because it performs a text comparison on the **type** attribute that is defined for a managed configuration section handler. Because all `Web.config` files that are generated by Visual Studio 2008 and Visual Studio 2008 SP1 have "3.5" in the type string for the **system.web.extensions** (and related) configuration section handlers, and because the ASP.NET 4 `machine.config` file has "4.0" in the **type** attribute for the same configuration section handlers, applications that are generated in Visual Studio 2008 or Visual Studio 2008 SP1 always fail configuration validation in IIS 7 and IIS 7.5.

Resolving These Issues

The workaround for the first scenario is to update the application-level `Web.config` file by including the boilerplate configuration text from a `Web.config` file that was generated automatically by Visual Studio 2008.

An alternative workaround for the first scenario is to install Service Pack 2 for Vista or Windows Server 2008 on your computer or to install hotfix KB958854 (https://support.microsoft.com/kb/958854) to fix the incorrect configuration-merge behavior of the IIS configuration system. However, after you perform either of these actions, your application will likely encounter a configuration error due to the issue described for the second scenario.

The workaround for the second scenario is to delete or comment out all the **system.web.extensions** configuration section definitions and configuration section group definitions from the application-level `Web.config` file. These definitions are usually at the top of the application-level `Web.config` file and can be identified by the **configSections** element and its children.

For both scenarios, it is recommended that you also manually delete the **system.codedom** section, although this is not required.

ASP.NET 4 Child Applications Fail to Start When Under ASP.NET 2.0 or ASP.NET 3.5 Applications

ASP.NET 4 applications that are configured as children of applications that run earlier versions of ASP.NET might fail to start because of configuration or compilation errors. The following example shows a directory structure for an affected application.

`/parentwebapp` (configured to use ASP.NET 2.0 or ASP.NET 3.5)
`/childwebapp` (configured to use ASP.NET 4)

The application in the `childwebapp` folder will fail to start on IIS 7 or IIS 7.5 and will report a configuration error. The error text will include a message similar to the following:

- `The requested page cannot be accessed because the related configuration data for the page is invalid.`
- `The configuration section 'configSections' cannot be read because it is missing a section declaration.`

On IIS 6, the application in the `childwebapp` folder will also fail to start, but it will report a different error. For example, the error text might state the following:

- ```
  The value for the 'compilerVersion' attribute in the provider options must be 'v4.0' or later if you are compiling for version 4.0 or later of the .NET Framework. To compile this Web application for version 3.5 or earlier of the .NET Framework, remove the 'targetFramework' attribute from the element of the Web.config file
  ```

These scenarios occur because the configuration information from the parent application in the `parentwebapp` folder is part of the hierarchy of configuration information that determines the final merged configuration settings that are used by the child web application in the `childwebapp` folder. Depending on whether the ASP.NET 4 Web application is running on IIS 7 (or IIS 7.5) or on IIS 6, either the IIS configuration system or the ASP.NET 4 compilation system will return an error.

The steps that you must follow to resolve this issue and to get the child ASP.NET 4 application to work depend on whether the ASP.NET 4 application runs on IIS 6 or on IIS 7 (or IIS 7.5).

Step 1 (IIS 7 or IIS 7.5 only)

This step is necessary only on operating systems that run IIS 7 or IIS 7.5, which includes Windows Vista, Windows Server 2008, Windows 7, and Windows Server 2008 R2.

Move the **configSections** definition in the `Web.config` file of the parent application (the application that runs ASP.NET 2.0 or ASP.NET 3.5) into the root `Web.config` file for the.NET Framework 2.0. The IIS 7 and IIS 7.5 native configuration system scans the **configSections** element when it merges the hierarchy of configuration files. Moving the **configSections** definition from the parent Web application's `Web.config` file to the root `Web.config`
  ```

file effectively hides the element from the configuration merge process that occurs for the child ASP.NET 4 application.

On a 32-bit operating system or for 32-bit application pools, the root `Web.config` file for ASP.NET 2.0 and ASP.NET 3.5 is normally located in the following folder:

`C:\Windows\Microsoft.NET\Framework\v2.0.50727\CONFIG`

On a 64-bit operating system or for 64-bit application pools, the root `Web.config` file for ASP.NET 2.0 and ASP.NET 3.5 is normally located in the following folder:

`C:\Windows\Microsoft.NET\Framework64\v2.0.50727\CONFIG`

If you run both 32-bit and 64-bit Web applications on a 64-bit computer, you must move the **configSections** element up into root `Web.config` files for both the 32-bit and the 64-bit systems.

When you put the **configSections** element in the root `Web.config` file, paste the section immediately after the **configuration** element. The following example shows what the top portion of the root `Web.config` file should look like when you have finished moving the elements.

> **NOTE**
>
> In the following example, lines have been wrapped for readability.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- The root web configuration file -->
<configuration>
  <configSections>
    <sectionGroup name="system.web.extensions"
        type="System.Web.Configuration.SystemWebExtensionsSectionGroup,
      System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
      PublicKeyToken=31BF3856AD364E35">

      <sectionGroup name="scripting"
        type="System.Web.Configuration.ScriptingSectionGroup,
        System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
        PublicKeyToken=31BF3856AD364E35">

        <section name="scriptResourceHandler"
          type="System.Web.Configuration.ScriptingScriptResourceHandlerSection,
          System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
          PublicKeyToken=31BF3856AD364E35" requirePermission="false"
          allowDefinition="MachineToApplication" />

        <sectionGroup name="webServices"
          type="System.Web.Configuration.ScriptingWebServicesSectionGroup,
          System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
          PublicKeyToken=31BF3856AD364E35">

          <section name="jsonSerialization"
            type="System.Web.Configuration.ScriptingJsonSerializationSection,
            System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
            PublicKeyToken=31BF3856AD364E35" requirePermission="false"
            allowDefinition="Everywhere" />

          <section name="profileService"
            type="System.Web.Configuration.ScriptingProfileServiceSection,
            System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
            PublicKeyToken=31BF3856AD364E35" requirePermission="false"
            allowDefinition="MachineToApplication" />
          <section name="authenticationService"
            type="System.Web.Configuration.ScriptingAuthenticationServiceSection,
          System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
          PublicKeyToken=31BF3856AD364E35" requirePermission="false"
            allowDefinition="MachineToApplication" />

          <section name="roleService"
            type="System.Web.Configuration.ScriptingRoleServiceSection,
          System.Web.Extensions, Version=3.5.0.0, Culture=neutral,
          PublicKeyToken=31BF3856AD364E35" requirePermission="false"
            allowDefinition="MachineToApplication" />

        </sectionGroup>
      </sectionGroup>
    </sectionGroup>
  </configSections>
```

Step 2 (all versions of IIS)

This step is required whether the ASP.NET 4 child Web application is running on IIS 6 or on IIS 7 (or IIS 7.5).

In the `Web.config` file of the parent Web application that is running ASP.NET 2 or ASP.NET 3.5, add a **location** tag that explicitly specifies (for both the IIS and ASP.NET configuration systems) that the configuration entries only apply to the parent Web application. The following example shows the syntax of the **location** element to add:

```
<location path="" inheritInChildApplications="false" >
  <!-- Additional settings -->
</location>
```

The following example shows how the **location** tag is used to wrap all configuration sections starting with the **appSettings** section and ending with **system.webServer** section.

```
<location path="" inheritInChildApplications="false" >
  <appSettings />
  <connectionStrings />
  <system.web>
    <!-- Removed for brevity -->
  </system.web>
  <system.codedom>
    <!-- Removed for brevity -->
  </system.codedom>
  <system.webServer>
    <!-- Removed for brevity -->
</system.webServer>
</location>
```

When you have completed steps 1 and 2, child ASP.NET 4 Web applications will start without errors.

ASP.NET 4 Web Sites Fail to Start on Computers Where SharePoint Is Installed

Web servers that run SharePoint have a `Web.config` file that is deployed at the root of a SharePoint Web site (for example, `c:\inetpub\wwwroot\web.config` for Default Web Site). In this `Web.config` file, SharePoint sets a custom partial-trust level named WSS_Minimal.

If you try to run an ASP.NET 4 Web site that is deployed as a child of this type of SharePoint Web site, you will see the following error:

`Could not find permission set named 'ASP.NET'.`

This error occurs because the ASP.NET 4 code access security (CAS) infrastructure looks for a permission set named ASP.NET. However, the partial trust configuration file that is referenced by WSS_Minimal does not contain any permission sets with that name.

Currently there is not a version of SharePoint available that is compatible with ASP.NET. As a result, you should not attempt to run an ASP.NET 4 Web site as a child site underneath SharePoint Web sites.

The HttpRequest.FilePath Property No Longer Includes PathInfo Values

Previous versions of ASP.NET included a **PathInfo** value in the value returned from various file path-related properties, including **HttpRequest.FilePath**, **HttpRequest.AppRelativeCurrentExecutionFilePath**, and **HttpRequest.CurrentExecutionFilePath**. ASP.NET 4 no longer includes the **PathInfo** value in the return values from these properties. Instead, the **PathInfo** information is available in **HttpRequest.PathInfo**. For example, imagine the following URL fragment:

`/testapp/Action.mvc/SomeAction`

In earlier versions of ASP.NET, **HttpRequest** properties have the following values:

HttpRequest.FilePath: `/testapp/Action.mvc/SomeAction`

HttpRequest.PathInfo: (empty)

In ASP.NET 4, **HttpRequest** properties instead have the following values:

HttpRequest.FilePath: `/testapp/Action.mvc`

HttpRequest.PathInfo: `SomeAction`

ASP.NET 2.0 Applications Might Generate HttpException Errors that Reference eurl.axd

After ASP.NET 4 has been enabled on IIS 6, ASP.NET 2.0 applications that run on IIS 6 (in either Windows Server 2003 or Windows Server 2003 R2) might generate errors such as the following:

```
System.Web.HttpException: Path '/[yourApplicationRoot]/eurl.axd/[Value]' was not found.
```

This error occurs because when ASP.NET detects that a Web site is configured to use ASP.NET 4, a native component of ASP.NET 4 passes an extensionless URL to the managed portion of ASP.NET for further processing. However, if virtual directories that are below an ASP.NET 4 Web site are configured to use ASP.NET 2.0, processing the extensionless URL in this way results in a modified URL that contains the string "eurl.axd". This modified URL is then sent to the ASP.NET 2.0 application. ASP.NET 2.0 cannot recognize the "eurl.axd" format. Therefore, ASP.NET 2.0 tries to find a file named `eurl.axd` and execute it. Because no such file exists, the request fails with an **HttpException** exception.

You can work around this issue using one of the following options.

Option 1

If ASP.NET 4 is not required in order to run the Web site, remap the site to use ASP.NET 2.0 instead.

Option 2

If ASP.NET 4 is required in order to run the Web site, move any child ASP.NET 2.0 virtual directories to a different Web site that is mapped to ASP.NET 2.0.

Option 3

If it is not practical to remap the Web site to ASP.NET 2.0 or to change the location of a virtual directory, explicitly disable extensionless URL processing in ASP.NET 4. Use the following procedure:

1. In the Windows registry, open the following node:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\ASP.NET\4.0.30319.0
```

1. Create a new **DWORD** value named **EnableExtensionlessUrls**.
2. Set **EnableExtensionlessUrls** to 0. This disables extensionless URL behavior.
3. Save the registry value and close the registry editor.
4. Run the **iisreset** command-line tool, which causes IIS to read the new registry value.

> **NOTE**
>
> Setting **EnableExtensionlessUrls** to 1 enables extensionless URL behavior. This is the default setting if no value is specified.

Event Handlers Might Not Be Not Raised in a Default Document in IIS 7 or IIS 7.5 Integrated Mode

ASP.NET 4 includes modifications that change how the **action** attribute of the HTML **form** element is rendered when an extensionless URL resolves to a default document. An example of an extensionless URL resolving to a default document would be http://contoso.com/, resulting in a request to http://contoso.com/Default.aspx.

ASP.NET 4 now renders the HTML **form** element's **action** attribute value as an empty string when a request is made to an extensionless URL that has a default document mapped to it. For example, in earlier releases of ASP.NET, a request to http://contoso.com would result in a request to `Default.aspx`. In that document, the opening **form** tag would be rendered as in the following example:

```
<form action="Default.aspx" />
```

In ASP.NET 4, a request to http://contoso.com also results in a request to `Default.aspx`. However, ASP.NET now renders the HTML opening **form** tag as in the following example:

```
<form action="" />
```

This difference in how the **action** attribute is rendered can cause subtle changes in how a form post is processed by IIS and ASP.NET. When the **action** attribute is an empty string, the IIS **DefaultDocumentModule** object will create a child request to `Default.aspx`. Under most conditions, this child request is transparent to application code, and the `Default.aspx` page runs normally.

However, a potential interaction between managed code and IIS 7 or IIS 7.5 Integrated mode can cause managed .aspx pages to stop working properly during the child request. If the following conditions occur, the child request to a `Default.aspx` document will result in an error or in unexpected behavior:

1. An .aspx page is sent to the browser with the **form** element's **action** attribute set to "".
2. The form is posted back to ASP.NET.
3. A managed HTTP module reads some part of the entity body. For example, a module reads **Request.Form** or **Request.Params**. This causes the entity body of the POST request to be read into managed memory. As a result, the entity body is no longer available to any native code modules that are running in IIS 7 or IIS 7.5 Integrated mode.
4. The IIS **DefaultDocumentModule** object eventually runs and creates a child request to the `Default.aspx` document. However, because the entity body has already been read by a piece of managed code, there is no entity body available to send to the child request.
5. When the HTTP pipeline runs for the child request, the handler for `.aspx` files runs during the handler-execute phase.
6. Because there is no entity body, there are no form variables and no view state, and therefore no information is available for the .aspx page handler to determine what event (if any) is supposed to be raised. As a result, none of the postback event handlers for the affected .aspx page run.

You can work around this behavior in the following ways:

- Identify the HTTP module that is accessing the request's entity body during default document requests and determine whether it can be configured to run only for managed requests. In Integrated mode for both IIS 7 and IIS 7.5, HTTP modules can be marked to run only for managed requests by adding the following attribute to the module's **system.webServer/modules** entry:

- `precondition="managedHandler"`

- This setting disables the module for requests that IIS 7 and IIS 7.5 determine as not being managed requests. For default document requests, the first request is to an extensionless URL. Therefore, IIS does not run any managed modules that are marked with a precondition of managed Handler during initial request processing. As a result, managed modules will not accidentally read the entity body and thus the entity body is still available and is passed along to the child request and to the default document.

- If the problematic HTTP modules have to run for all requests (for static files, for extensionless URLs that resolve to the **DefaultDocumentModule** object, for managed requests, etc.), modify the affected .aspx pages by explicitly setting the **Action** property of the page's **System.Web.UI.HtmlControls.HtmlForm** control to a non-empty string. For example, if the default document is `Default.aspx`, modify the page's code to explicitly set the **HtmlForm** control's **Action** property to "Default.aspx".

Changes to the ASP.NET Code Access Security (CAS) Implementation

ASP.NET 2.0, and by extension the ASP.NET features that were added in 3.5, use the .NET Framework 1.1 and 2.0 code access security (CAS) model. However, the implementation of CAS in ASP.NET 4 has been substantially overhauled. As a result, partial-trust ASP.NET applications that rely on trusted code running in the global assembly

cache (GAC) might fail with various security exceptions. Partial-trust applications that rely on extensive modifications to machine CAS policy might also fail with security exceptions.

You can revert partial-trust ASP.NET 4 applications to the behavior of ASP.NET 1.1 and 2.0 using the new **legacyCasModel** attribute in the **trust** configuration element, as shown in the following example:

```
<trust level= "Medium" legacyCasModel="true" />
```

When you revert to the legacy CAS model, the following old CAS behaviors are enabled:

- Machine CAS policy is honored.
- Multiple different permission sets in a single application domain are allowed.
- Explicit permission assertions are not required for assemblies in the GAC that are invoked when only ASP.NET or other .NET Framework code is on the stack.

One scenario cannot be reverted in the .NET Framework 4: non-Web partial-trust applications can no longer call certain APIs in System.Web.dll and System.Web.Extensions.dll. In previous versions of the .NET Framework, it was possible for non-Web partial-trust applications to be explicitly granted **AspNetHostingPermission** permissions. These applications could then use **System.Web.HttpUtility**, types in the **System.Web.ClientServices.*** namespaces, and types related to membership, roles, and profiles. Calling these types from non-Web partial trust applications is no longer supported in the .NET Framework 4.

> **NOTE**
>
> The **HtmlEncode** and **HtmlDecode** functionality of the **System.Web.HttpUtility** class was moved to the new .NET Framework 4 **System.Net.WebUtility** class. If that was the only ASP.NET functionality that was being used, modify the application's code to use the new **WebUtility** class instead.

The following is a high-level summary of the changes to the default CAS implementation in ASP.NET 4:

- ASP.NET application domains are now homogeneous application domains. Only partial-trust and full-trust grant sets are available in an application domain.
- ASP.NET partial-trust grant sets are independent from any enterprise-level, machine-level, or user-level CAS policy.
- ASP.NET assemblies that shipped in 3.5 and 3.5 SP1 have been converted to use the .NET Framework 4 transparency model.
- Use of the ASP.NET **AspNetHostingPermission** attribute has been substantially reduced. Most instances of this attribute have been removed from the public ASP.NET APIs.
- Dynamically compiled assemblies that are created by ASP.NET build providers have been updated to explicitly mark assemblies as transparent.
- All ASP.NET assemblies are now marked in such a way that the APTCA attribute is honored only in Web hosting environments. Partially trusted non-Web hosting environments like ClickOnce will not be able to call into ASP.NET assemblies.

For more information about the new ASP.NET 4 code access security model, see Using Code Access Security in ASP.NET Applications on the MSDN Web site.

MembershipUser and Other Types in the System.Web.Security Namespace Have Been Moved

Some types that are used in ASP.NET membership have been moved from `System.Web.dll` to the new System.Web.ApplicationServices.dll assembly. The types were moved in order to resolve architectural layering dependencies between types in the client and in extended .NET Framework SKUs.

Web site projects do not have problems as a result of moving these types, because System.Web.ApplicationServices.dll was added to the list of referenced assemblies that is used by default by the ASP.NET compilation system. If you upgrade a Web site project that was created using an earlier version of ASP.NET to ASP.NET 4 by opening it in Visual Studio 2010, the project will compile without errors.

Similarly, if you upgrade a Web application project that was created in an earlier version of ASP.NET to ASP.NET 4 by opening it in Visual Studio 2010, the upgrade process adds a reference to System.Web.ApplicationServices.dll to the project. Therefore, upgraded Web application projects will also compile without errors.

Compiled (binary) files that were created using earlier versions of ASP.NET will also run without errors on ASP.NET 4, even though the membership types were moved to a different assembly. Type forwarding information has been added to the ASP.NET 4 version of `System.Web.dll` that automatically routes run-time references for these types to the new location for the types.

However, class libraries that use specific membership types and that have been upgraded from earlier versions of ASP.NET will fail to compile when used in an ASP.NET 4 project. For example, a class library project might fail to compile and report an error such as the following:

- ```
  The type 'System.Web.Security.MembershipUser' is defined in an assembly that is not referenced. You must add a reference to assembly 'System.Web.ApplicationServices, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35'.
  ```
- ```
  The type name 'MembershipUser' could not be found. This type has been forwarded to assembly 'System.Web.ApplicationServices, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35'. Consider adding a reference to that assembly.
  ```

You can work around this problem by adding a reference in your class library project to System.Web.ApplicationServices.dll.

The following list shows the *System.Web.Security* types that were moved from `System.Web.dll` to System.Web.ApplicationServices.dll:

- *System.Web.Security.MembershipCreateStatus*
- *System.Web.Security.Membership.CreateUserException*
- *System.Web.Security.MembershipPasswordException*
- *System.Web.Security.MembershipPasswordFormat*
- *System.Web.Security.MembershipProvider*
- *System.Web.Security.MembershipProviderCollection*
- *System.Web.Security.MembershipUser*
- *System.Web.Security.MembershipUserCollection*
- *System.Web.Security.MembershipValidatePasswordEventHandler*
- *System.Web.Security.ValidatePasswordEventArgs*
- *System.Web.Security.RoleProvider*
- *System.Web.Configuration.MembershipPasswordCompatibilityMode*

Output Caching Changes to Vary * HTTP Header

In ASP.NET 1.0, a bug caused cached pages that specified `Location="ServerAndClient"` as an output–cache setting to emit a `Vary:*` HTTP header in the response. This had the effect of telling client browsers to never cache the page locally.

In ASP.NET 1.1, the **System.Web.HttpCachePolicy.SetOmitVaryStar** method was added, which you could call to suppress the `Vary:*` header. This method was chosen because changing the emitted HTTP header was considered a potentially breaking change at the time. However, developers have been confused by the behavior in ASP.NET, and bug reports suggest that developers are unaware of the existing **SetOmitVaryStar** behavior.

In ASP.NET 4, the decision was made to fix the root problem. The `Vary:*` HTTP header is no longer emitted from responses that specify the following directive:

```
<%@OutputCache Location="ServerAndClient" %>
```

As a result, **SetOmitVaryStar** is no longer needed in order to suppress the `Vary:*` header.

In applications that specify `Location="ServerAndClient"` in the **@ OutputCache** directive on a page, you will now see the behavior implied by the name of the **Location** attribute's value – that is, pages will be cacheable in the browser without requiring that you call the **SetOmitVaryStar** method.

If pages in your application must emit `Vary:*`, call the **AppendHeader** method, as in the following example:

```
HttpResponse.AppendHeader("Vary","*");
```

Alternatively, you can change the value of the output caching **Location** attribute to "Server".

System.Web.Security Types for Passport are Obsolete

The Passport support built into ASP.NET 2.0 has been obsolete and unsupported for a few years due to changes in Passport (now LiveID). As a result, the five types related to Passport in **System.Web.Security** are now marked with the **ObsoleteAttribute** attribute.

The MenuItem.PopOutImageUrl Property Fails to Render an Image in ASP.NET 4

In ASP.NET 3.5, the *MenuItem.PopOutImageUrl* property lets you specify the URL for an image that is displayed in a menu item to indicate that the menu item has a dynamic submenu. The following example shows how to specify this property in markup in ASP.NET 3.5.

```
<asp:menu id="NavigationMenu"
    staticdisplaylevels="1"
    staticsubmenuindent="10"
    orientation="Vertical"
    target="_blank"
    runat="server">
  <items>
    <asp:menuitem navigateurl="default2.aspx"
        text="Home"
        PopOutImageUrl="~/Images/Popout.gif"
        tooltip="Home">
      <asp:menuitem navigateurl="default3.aspx"
          text="Movies"
          PopOutImageUrl="~/Images/Popout.gif"
          tooltip="Movies">
      </asp:menuitem>
    </asp:menuitem>
  </items>
</asp:menu>
```

As a result of a design change in ASP.NET 4, no output is rendered for the *PopOutImageUrl* if the property is set for the *MenuItem* class. Instead, you must specify an image URL directly in the *Menu* control using either the *StaticPopOutImageUrl* property or the *DynamicPopOutImageUrl* property. When you work with a static menu, the *Menu.StaticPopOutImageUrl* property specifies the URL for an image that is displayed in order to indicate that the static menu item has a submenu, as shown in the following example:

```
<asp:menu id="NavigationMenu"
    staticdisplaylevels="1"
    staticsubmenuindent="10"
    orientation="Vertical"
    target="_blank"
    StaticPopOutImageTextFormatString="More..."
    StaticPopOutImageUrl="Images/Popout.gif"
    runat="server">
  <items>
    <asp:menuitem navigateurl="default2.aspx"
        text="Home"
        tooltip="Home">
      <asp:menuitem navigateurl="default3.aspx"
          text="Movies"
          tooltip="Movies">
      </asp:menuitem>
    </asp:menuitem>
  </items>
</asp:menu>
```

If you are working with a dynamic menu, you use the *Menu.DynamicPopOutImageUrl* property to specify the URL for an image that indicates that a dynamic menu item has a submenu. The following example is similar to the previous one, but shows how to set the *DynamicPopOutImageUrl* property for a dynamic menu.

```
<asp:menu id="NavigationMenu"
    staticdisplaylevels="1"
    staticsubmenuindent="10"
    orientation="Vertical"
    target="_blank"
    DynamicPopOutImageTextFormatString="More..."
    DynamicPopOutImageUrl="Images/Popout.gif"
    runat="server">
  <items>
    <asp:menuitem navigateurl="default2.aspx"
        text="Home"
        tooltip="Home">
      <asp:menuitem navigateurl="default3.aspx"
          text="Movies"
          tooltip="Movies">
      </asp:menuitem>
    </asp:menuitem>
  </items>
</asp:menu>
```

If the *Menu.DynamicPopOutImageUrl* property is not set and the *Menu.DynamicEnableDefaultPopOutImage* property is set to *false*, no image is displayed. Similarly, if the *StaticPopOutImageUrl* property is not set and the *StaticEnableDefaultPopOutImage* property is set to *false*, no image is displayed.

When you set the paths for these properties, use a forward slash (/) instead of a backslash (). For more information, see Menu.StaticPopOutImageUrl and Menu.DynamicPopOutImageUrl Fail to Render Images When Paths Contain Backslashes elsewhere in this document.

Menu.StaticPopOutImageUrl and Menu.DynamicPopOutImageUrl Fail to Render Images When Paths Contain Backslashes

In ASP.NET 4, the images that you specify using the *Menu.StaticPopOutImageUrl* and *Menu.DynamicPopOutImageUrl* properties will not render if the path contains backlashes (). This is a change from earlier versions of ASP.NET.

The following example of *Menu* control markup shows the *StaticPopOutImageUrl* property set using a path that

contains a backslash. In ASP.NET 4, the image specified in the property will not render.

```
<asp:menu id="NavigationMenu"
    staticdisplaylevels="1"
    staticsubmenuindent="10"
    orientation="Vertical
    target="_blank"
    StaticPopOutImageTextFormatString="More..."
    StaticPopOutImageUrl="Images\Popout.gif"
    runat="server">
  <items>
    <asp:menuitem navigateurl="default2.aspx"
        text="Home"
        tooltip="Home">
      <asp:menuitem navigateurl="default3.aspx"
          text="Movies"
          tooltip="Movies">
      </asp:menuitem>
    </asp:menuitem>
  </items>
</asp:menu>
```

To correct this issue, change path values that are specified in the *StaticPopOutImageUrl* and *DynamicPopOutImageUrl* properties to use forward slashes (/). The following example shows this change:

```
<asp:menu id="NavigationMenu"
    staticdisplaylevels="1"
    staticsubmenuindent="10"
    orientation="Vertical
    target="_blank"
    StaticPopOutImageTextFormatString="More..."
    StaticPopOutImageUrl="Images/Popout.gif"
    runat="server">
  <items>
    <asp:menuitem navigateurl="default2.aspx"
        text="Home"
        tooltip="Home">
      <asp:menuitem navigateurl="default3.aspx"
          text="Movies"
          tooltip="Movies">
      </asp:menuitem>
    </asp:menuitem>
  </items>
</asp:menu>
```

Note that applications that were migrated from earlier versions of ASP.NET to ASP.NET 4 might also be affected, because the *MenuItem.PopOutImageUrl* property has been changed. For more information, see The MenuItem.PopOutImageUrl Property Fails to Render an Image in ASP.NET 4 elsewhere in this document.

Disclaimer

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, email addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Upgrading an ASP.NET MVC 1.0 Application to ASP.NET MVC 2

2/20/2020 • 2 minutes to read • Edit Online

This document describes both how to upgrade manually and with a wizard an ASP.NET MVC 1.0 Application to ASP.NET MVC 2. This document is also available for Download

Introduction

ASP.NET MVC 2 can be installed side by side with ASP.NET MVC 1.0 on the same server. This gives application developers flexibility in choosing when to upgrade an ASP.NET MVC 1.0 application to ASP.NET MVC 2.

Visual Studio 2010 includes a wizard that upgrades existing ASP.NET MVC 1.0 projects built with Visual Studio 2008 to ASP.NET MVC 2. The upgrade wizard is initiated by opening an ASP.NET MVC 1.0 project in Visual Studio 2010.

Upgrade Wizard for ASP.NET MVC 1.0 on Visual Studio 2008 SP1

To upgrade an ASP.NET MVC 1.0 application to ASP.NET MVC 2 in Visual Studio 2008 SP1, use the (unsupported) MvcAppConverter application. You can download this application from the following URL:

https://go.microsoft.com/fwlink/?LinkID=185351

Manually Upgrading an ASP.NET MVC 1.0 Project

To manually upgrade an existing ASP.NET MVC 1.0 application to version 2, follow these steps:

1. Make a backup of the existing project.
2. In a text editor, open the project file (the file with the .csproj or .vbproj file extension) and find the ProjectTypeGuid element. As the value of that element, replace the GUID {603c0e0b-db56-11dc-be95-000d561079b0} with {F85E285D-A4E0-4152-9332-AB1D724D3325}. When you are done, the value of that element should be as follows:

   ```
   {F85E285D-A4E0-4152-9332-AB1D724D3325};{349c5851-65df-11da-9384-00065b846f21};{fae04ec0-301f-11d3-bf4b-00c04f79efbc}
   ```

3. In the Web application root folder, edit the Web.config file. Search for System.Web.Mvc, Version=1.0.0.0 and replace all instances with System.Web.Mvc, Version=2.0.0.0.
4. Repeat the previous step for the Web.config file located in the Views folder.
5. Open the project using Visual Studio, and in **Solution Explorer**, expand the **References** node. Delete the reference to System.Web.Mvc (which points to the version 1.0 assembly). Add a reference to System.Web.Mvc (v2.0.0.0).
6. Add the following bindingRedirect element to the Web.config file in the application root under the configuraton section:

```
<runtime>
     <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
       <dependentAssembly>
         <assemblyIdentity name="System.Web.Mvc"

   publicKeyToken="31bf3856ad364e35"/>
         <bindingRedirect oldVersion="1.0.0.0"
   newVersion="2.0.0.0"/>
       </dependentAssembly>
     </assemblyBinding>
   </runtime>
```

7. Create a new empty ASP.NET MVC 2 application. Copy the files from the Scripts folder of the new application into the Scripts folder of the existing application.
8. Update the existing applicationâ€™s CSS file with the CSS style definitions in the Site.css file.
9. Compile the application and run it. If any errors occur, refer to the Breaking Changes section of the What's New in ASP.NET MVC 2 page.

ASP.NET Denied Access to IIS Directories

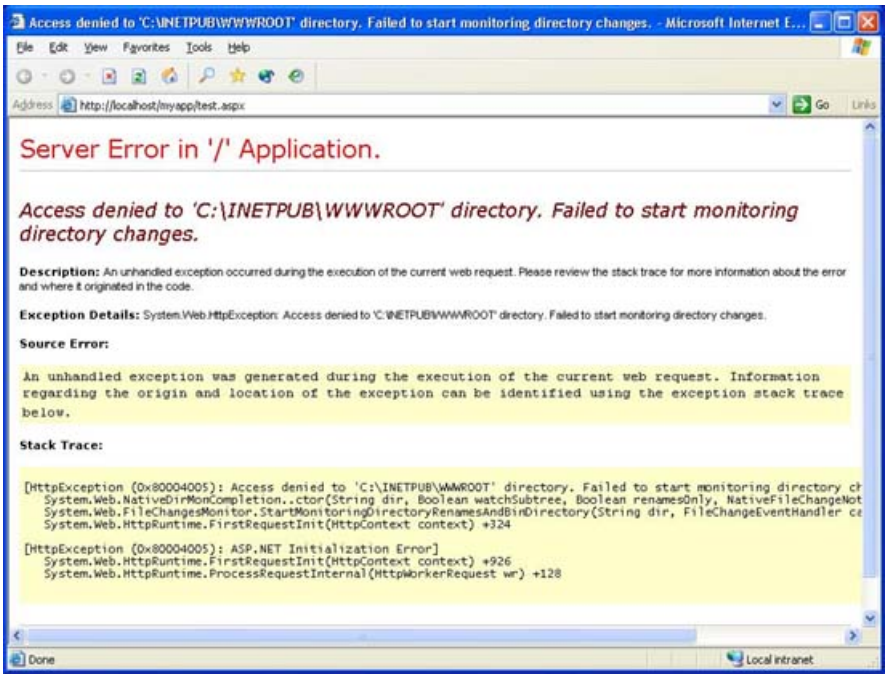
2/20/2020 • 2 minutes to read • Edit Online

This whitepaper describes what you must do if a request to your ASP.NET application returns the error, "Denied Access to *DirectoryName* directory. Failed to start monitoring directory changes."

Applies to ASP.NET 1.0 and ASP.NET 1.1.

ASP.NET V1 RTM now runs using a less privileged windows account - registered as the "ASPNET" account on a local machine.

On some locked down systems, this account may not by default have read security access to a website's content directories, the application root directory, or the web site root directory. In this case you will receive the following error when requesting pages from a given web application:



To fix this you will need to change the security permissions on the appropriate directories.

Specifically, ASP.NET requires read, execute, and list access for the ASPNET account for the web site root (for example: c:\inetpub\wwwroot or any alternative site directory you may have configured in IIS), the content directory and the application root directory in order to monitor for configuration file changes. The application root corresponds to the folder path associated with the application virtual directory in the IIS Administration tool (inetmgr).

For example, consider the following application hierarchy under the wwwroot folder.

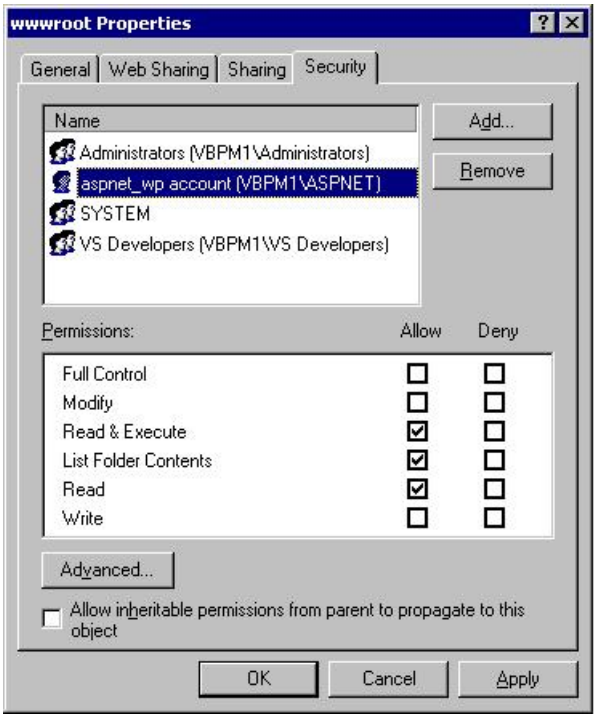
```
C:\inetpub\wwwroot\myapp\default.aspx
```

For this example, the ASPNET account needs the read permissions defined above for content in both the myapp and the wwwroot directory. A single inherited ACL on the root folder can also be optionally used for both directories if they're nested.

To add permissions to a directory, perform the following steps:

- Using the Windows explorer, navigate to the directory

- Right click on the directory folder and choose "Properties"
- Navigate to the "Security" tab on the property dialog
- Click the "Add" button and enter the machine name followed by the ASPNET account name. For example, on a machine named "webdev", you would enter webdev\ASPNET and hit "OK".
- Ensure that the ASPNET account has the "Read & Execute", "List Folder Contents", and "Read" checkboxes checked.
- Hit OK to dismiss the dialog and save the changes.



If desired, these changes can be automated using scripts or the "cacls.exe" tool that ships with Windows. For more information on the ASPNET account, please see the FAQ document.

If a given web application relies on having write or modify permissions to a particular folder or file, this can be granted by following the same procedure and checking the "Write" and/or "Modify" checkboxes.

On machines that allow Everyone or the Users group read access to these directories (which is the default configuration), no issues will be encountered and the above steps will not be required.

What's New in ASP.NET MVC 2

9/16/2020 • 17 minutes to read • Edit Online

This document describes new features and improvements introduced in ASP.NET MVC 2. This document is also available for Download

Introduction
Upgrading an ASP.NET MVC 1.0 Project to ASP.NET MVC 2
New Features
Templated Helpers
Areas
Support for Asynchronous Controllers
Support for DefaultValueAttribute in Action-Method Parameters
Support for Binding Binary Data with Model Binders
ModelMetadata and ModelMetadataProvider Classes
Support for DataAnnotations Attributes
Model-Validator Providers
Client-Side Validation
New Code Snippets for Visual Studio 2010
New RequireHttpsAttribute Action Filter
Overriding the HTTP Method Verb
New HiddenInputAttribute Class for Templated Helpers
Html.ValidationSummary Helper Method Can Display Model-Level Errors
T4 Templates in Visual Studio Generate Code that is Specific to the Target Version of the .NET FrameworkAPI Improvements
Breaking Changes
Disclaimer

Introduction

ASP.NET MVC 2 builds on ASP.NET MVC 1.0 and introduces a large set of enhancements and features that are focused on increasing productivity. This release is compatible with ASP.NET MVC 1.0, so all your knowledge, skills, code, and extensions for ASP.NET MVC 1.0 continue to apply.

For more information about ASP.NET MVC, visit the following resources:

- ASP.NET MVC documentation on MSDN
- The ASP.NET MVC Web site
- The ASP.NET MVC forums

Upgrading an ASP.NET MVC 1.0 Project to ASP.NET MVC 2

ASP.NET MVC 2 can be installed side by side with ASP.NET MVC 1.0 on the same server, which gives application developers flexibility in choosing when to upgrade an ASP.NET MVC 1.0 application to ASP.NET MVC 2. For information on how to upgrade, see the document Upgrading an ASP.NET MVC 1.0 Application to ASP.NET MVC 2.

New Features

This section describes features that have been introduced in the MVC 2 release.

Templated Helpers

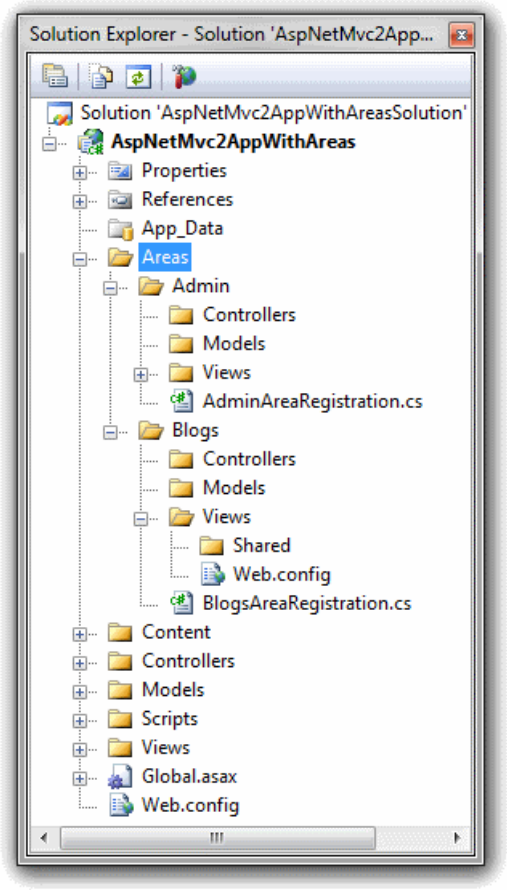
Templated helpers let you automatically associate HTML elements for edit and display with data types. For example, when data of type System.DateTime is displayed in a view, a date-picker UI element can be automatically rendered. This is similar to how field templates work in ASP.NET Dynamic Data. For more information, see Using Templated Helpers to Display Data on the MSDN Web site.

Areas

Areas let you organize a large project into multiple smaller sections in order to manage the complexity of a large Web application. Each section ("area") typically represents a separate section of a large Web site and is used to group related sets of controllers and views. For more information, see Walkthrough: Organizing an ASP.NET MVC Application by Areas on the MSDN Web site.

To create a new area, in Solution Explorer, right-click the project, click Add, and then click Area. This displays a dialog box that prompts you for the area name. After you enter the area name, Visual Studio adds a new area to the project.

The following figure shows an example layout for a project with two areas, Admin and Blogs.



When you create an area, Visual Studio adds a class that derives from AreaRegistration to each area. This class is required in order to register the area and its routes, as shown in the following example:

```
namespace MyApplication.Areas.Blog {
    public class BlogAreaRegistration : AreaRegistration {
        public override string AreaName {
            get { return "blog"; }
        }

        public override void RegisterArea(AreaRegistrationContext context) {
            context.MapRoute(
                "blog_default",
                "blog/{controller}/{action}/{id}",
                new { action = "Index", id = UrlParameter.Optional }
            );

            context.MapRoute(
                "blog_whatsnew",
                "whats-new",
                new { action = "WhatsNew", id = UrlParameter.Optional  }
            );
        }
    }
}
```

The default project template for ASP.NET MVC 2 includes a call to the RegisterAllAreas method in the code for the Global.asax file. This method registers each area in the project by looking for all types that derive from the AreaRegistration class, instantiating an instance of the type, and then calling the RegisterArea method on the instance. The following example shows how this is done.

```
public class MyMvcApplication : HttpApplication {

    void App_Start() {
        AreaRegistration.RegisterAllAreas();
        RegisterRoutes(RouteTable.Routes);
    }

    public static void RegisterRoutes(RouteCollection routes) {
        routes.MapRoute("default", "{controller}/{action}/{id}", ...);
    }
}
```

If you do not specify the namespace in the RegisterArea method by calling the context.Namespaces.Add method, the namespace of the registration class is used by default.

Support for Asynchronous Controllers

ASP.NET MVC 2 now allows controllers to process requests asynchronously. This can lead to performance gains by allowing servers which frequently call blocking operations (like network requests) to call non-blocking counterparts instead. For more information, see the Using an Asynchronous Controller in ASP.NET MVC topic on MSDN.

Support for DefaultValueAttribute in Action-Method Parameters

The System.ComponentModel.DefaultValueAttribute class allows a default value to be supplied for the argument parameter to an action method. For example, assume that the following default route is defined:

```
{controller}/{action}/{id}
```

Also assume that the following controller and action method is defined:

```
public class ArticleController {
    public ActionResult View(int id, [DefaultValue(1)]int page) {
    }
}
```

Any of the following request URLs will invoke the View action method that is defined in the preceding example.

- /Article/View/123
- /Article/View/123?page=1 (Effectively the same as the previous request)
- /Article/View/123?page=2

Without the DefaultValueAttribute attribute, the first URL from the preceding list would not work, because the page argument is a non-nullable value type whose value has not been provided.

If your code is written in Visual Basic 2010 or Visual C# 2010, you can use optional parameters instead of the DefaultValueAttribute attribute, as shown in the following example:

```
Function View(ByVal id As Integer, Optional ByVal page As Integer = 1) _
        As ActionResult
    ' ...
End Function

public ActionResult MyAction(int id, int page = 1) {
    // ...
}
```

Support for Binding Binary Data with Model Binders

There are two new overloads of the Html.Hidden helper that encode binary values as base-64-encoded strings:

```
public static string Hidden(this HtmlHelper htmlHelper, string name, Binary value);

public static string Hidden(this HtmlHelper htmlHelper, string name, byte[] value);
```

A typical use is to embed a timestamp for an object in the view. For example, your application might include the following Product object:

```
public class Product {
    //... other properties ...
    public byte[] TimeStamp {
        get;
        set;
    }
}
```

An edit form can render the TimeStamp property in the form as shown in the following example:

```
<%@ Page Inherits="ViewPage<Product>" %>
<%= Html.Hidden("TimeStamp", Model.TimeStamp) %>
```

This markup renders a hidden input element with the timestamp value as a base-64-encoded string that resembles the following example:

```
<input type="hidden" name="TimeStamp" value="QVNQLk5FVCBNVkMgaXMgZnVuIQ==" />
```

This form might be posted to an action method that has an argument of type Product, as shown in the following example:

```
public ActionResult Edit(Product p) {
    // p.TimeStamp is populated from the form
}
```

In the action method, the TimeStamp property is populated correctly because the posted base-64-encoded string is converted to a byte array.

ModelMetadata and ModelMetadataProvider Classes

The ModelMetadataProvider class provides an abstraction for obtaining metadata for the model within a view. MVC 2 includes a default provider that makes available the metadata that is exposed by the attributes in the System.ComponentModel.DataAnnotations namespace. It is possible to create metadata providers that provide metadata from other data stores, such as databases or XML files.

The ViewDataDictionary class exposes a ModelMetadata object that contains the metadata that is extracted from the model by the ModelMetadataProvider class. This enables the templated helpers to consume this metadata and adjust their output accordingly.

For more information, see the documentation for the ModelMetadata and ModelMetadataProvider classes.

Support for DataAnnotations Attributes

ASP.NET MVC 2 supports using the RangeAttribute, RequiredAttribute, StringLengthAttribute, and RegexAttribute validation attributes (defined in the System.ComponentModel.DataAnnotations namespace) when you bind to a model in order to provide input validation.

For more information, see How to: Validate Model Data Using DataAnnotations Attributes on the MSDN Web site. A sample project that illustrates the use of these attributes is available for download at https://go.microsoft.com/fwlink/?LinkId=157753.

Model-Validator Providers

The model-validation provider class represents an abstraction that provides validation logic for the model. ASP.NET MVC includes a default provider based on validation attributes that are included in the System.ComponentModel.DataAnnotations namespace. You can also create your own validation providers that define custom validation rules and custom mappings of validation rules to the model. For more information, see the documentation for the ModelValidatorProvider class.

Client-Side Validation

The model-validator provider class exposes validation metadata to the browser in the form of JSON-serialized data that can be consumed by a client-side validation library. ASP.NET MVC 2 includes a client validation library and adapter that supports the DataAnnotations namespace validation attributes noted earlier. The provider class also enables you to use other client-validation libraries by writing an adapter that processes the JSON data and calls into the alternate library.

New Code Snippets for Visual Studio 2010

A set of HTML code snippets for ASP.NET MVC 2 is installed with Visual Studio 2010. To view a list of these snippets, in the Tools menu, select Code Snippets Manager. For the language, select HTML, and for location, select ASP.NET MVC 2. For more information about how to use code snippets, see the Visual Studio documentation.

New RequireHttpsAttribute Action Filter

ASP.NET MVC 2 includes a new RequireHttpsAttribute class that can be applied to action methods and controllers. By default, the filter redirects a non-SSL (HTTP) request to the SSL-enabled (HTTPS) equivalent.

Overriding the HTTP Method Verb

When you build a Web site by using the REST architectural style, HTTP verbs are used to determine which action to perform for a resource. REST requires that applications support the full range of common HTTP verbs, including GET, PUT, POST, and DELETE.

ASP.NET MVC 2 includes new attributes that you can apply to action methods and that feature compact syntax. These attributes enable ASP.NET MVC to select an action method based on the HTTP verb. In the following example, a POST request will call the first action method and a PUT request will call the second action method.

```
[HttpPost]
public ActionResult Edit(int id)

[HttpPut]
public ActionResult Edit(int id, Tag tag)
```

In earlier versions of ASP.NET MVC, these action methods required more verbose syntax, as shown in the following example:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id)

[AcceptVerbs(HttpVerbs.Put)]
public ActionResult Edit(int id, Tag tag)
```

Because browsers support only the GET and POST HTTP verbs, it is not possible to post to an action that requires a different verb. Thus it is not possible to natively support all RESTful requests.

However, to support RESTful requests during POST operations, ASP.NET MVC 2 introduces a new HttpMethodOverride HTML helper method. This method renders a hidden input element that causes the form to effectively emulate any HTTP method. For example, by using the HttpMethodOverride HTML helper method, you can have a form submission appear be a PUT or DELETE request. The behavior of HttpMethodOverride affects the following attributes:

- HttpPostAttribute
- HttpPutAttribute
- HttpGetAttribute
- HttpDeleteAttribute
- AcceptVerbsAttribute

The hidden input element has its name X-HTTP-Method-Override and its value set to the HTTP verb to emulate. The override value can also be specified in an HTTP header or in a query string value as a name/value pair.

The override can only be used when the real request is a POST request. The override value will be ignored for requests that use any other HTTP verb.

New HiddenInputAttribute Class for Templated Helpers

You can apply the new HiddenInputAttribute attribute to a model property to indicate whether a hidden input element should be rendered when displaying the model in an editor template. (The attribute sets an implicit UIHint value of HiddenInput). The attribute's DisplayValue property lets you specify whether the value is displayed in editor and display modes. When DisplayValue is set to false, nothing is displayed, not even the HTML markup that normally surrounds a field. The default value for DisplayValue is true.

You might use HiddenInputAttribute attribute in the following scenarios:

- When a view lets users edit the ID of an object and it is necessary to display the value as well as to provide a hidden input element that contains the old ID so that it can be passed back to the controller.
- When a view lets users edit a binary property that should never be displayed, such as a timestamp property. In

that case, the value and surrounding HTML markup (such as the label and value) are not displayed.

The following example shows how to use the HiddenInputAttribute class.

```
public class ProductViewModel {
    [HiddenInput] // equivalent to [HiddenInput(DisplayValue=true)]
    public int Id { get; set; }

    public string Name { get; set; }

    [HiddenInput(DisplayValue=false)]
    public byte[] TimeStamp { get; set; }
}
```

When the attribute is set to true (or no parameter is specified), the following occurs:

- In display templates, a label is rendered and the value is displayed to the user.
- In editor templates, a label is rendered and the value is rendered in a hidden input element.

When the attribute is set to false, the following occurs:

- In display templates, nothing is rendered for that field.
- In editor templates, no label is rendered and the value is rendered in a hidden input element.

Html.ValidationSummary Helper Method Can Display Model-Level Errors

Instead of always displaying all validation errors, the Html.ValidationSummary helper method has a new option to display only model-level errors. This enables model-level errors to be displayed in the validation summary and field-specific errors to be displayed next to each field.

T4 Templates in Visual Studio Generate Code that is Specific to the Target Version of the .NET Framework

A new property is available to T4 files from the ASP.NET MVC T4 host that specifies the version of the .NET Framework that is used by the application. This enables T4 templates to generate code and markup that is specific to a version of the .NET Framework. In Visual Studio 2008, the value is always .NET 3.5. In Visual Studio 2010, the value is either .NET 3.5 or .NET 4.

API Improvements

This section describes changes to existing ASP.NET MVC types and members.

- Added a protected virtual CreateActionInvoker method in the Controller class. This method is invoked by the ActionInvoker property of Controller and allows for lazy instantiation of the invoker if no invoker is already set.
- Added a protected virtual HandleUnauthorizedRequest method in the AuthorizeAttribute class. This enables filters that derive from AuthorizeAttribute to control the behavior when authorization fails.
- Added an Add(string key, object value) method in the ValueProviderDictionary class. This enables you to use the dictionary initializer syntax for ValueProviderDictionary, as in the following example:

```
Controller c = new MyController();
c.ValueProvider = new ValueProviderDictionary(null) {
    { "example1", "example1Value" },
    { "example2", "example2Value" },
    { "example3", new int[] { 1, 2, 3 } }
};
```

- Added a get_object method in the Sys.Mvc.AjaxContext class. This is a JavaScript method that is similar to the get_data method, but if the content type of the response is application/json, get_object returns the JSON object.
- Added an ActionDescriptor property in the AuthorizationContext class.

- Added a UrlParameter.Optional token that can be used to work around problems when binding to a model that contains an ID property when the property is absent in a form post. For more detail, see the entry ASP.NET MVC 2 Optional URL Parameters on Phil Haack's blog.

Breaking Changes

The following changes might cause errors in existing ASP.NET MVC 1.0 applications.

Change in property validation behavior for classes that implement IDataErrorInfo

For model objects that use IDataErrorInfo to perform validation, every property is validated, regardless of whether a new value was set. In ASP.NET MVC 1.0, only properties that had new values set were validated. In ASP.NET MVC 2, the Error property of IDataErrorInfo is called only if all the property validators were successful.

IIS script mapping script is no longer available in the installer

The IIS script-mapping script is a command-line script that is used to configure script maps for IIS 6 and for IIS 7 in Classic mode. The script-mapping script is not needed if you use the Visual Studio Development Server or if you use IIS 7 in Integrated mode. The scripts are available as a separate unsupported download on the ASP.NET WebStack.

The Html.Substitute helper method in MVC Futures is no longer available

Due to changes in the rendering behavior of MVC view engines, the Html.Substitute helper method does not work and has been removed.

The IValueProvider interface replaces all uses of IDictionary

Every property or method argument that accepted IDictionary in MVC 1.0 now accepts IValueProvider. This change affects only applications that include custom value providers or custom model binders. Examples of properties and methods that are affected by this change include the following:

- The ValueProvider property of the ControllerBase and ModelBindingContext classes.
- The TryUpdateModel methods of the Controller class.

New CSS classes were added in the Site.css file

The Site.css file in the ASP.NET MVC project templates has been updated to include new styles used by the validation functionality and by the templated helpers.

Helpers now return an MvcHtmlString object

In order to take advantage of the new HTML-encoding expression syntax in ASP.NET 4, the return type for HTML helpers is now MvcHtmlString instead of a string. If you use ASP.NET MVC 2 and the new helpers on ASP.NET 3.5, you will not be able to take advantage of the HTML-encoding syntax; the new syntax is available only when you run ASP.NET MVC 2 on ASP.NET 4.

JsonResult now responds only to HTTP POST requests

In order to mitigate JSON hijacking attacks that have the potential for information disclosure, by default, the JsonResult class now responds only to HTTP POST requests. Ajax GET calls to action methods that return a JsonResult object should be changed to use POST instead. If necessary, you can override this behavior by setting the new JsonRequestBehavior property of JsonResult. For more information about the potential exploit, see the blog post JSON Hijacking on Phil Haack's blog.

Model and ModelType property setters on ModelBindingContext are obsolete

A new settable ModelMetadata property has been added to the ModelBindingContext class. The new property encapsulates both the Model and the ModelType properties. Although the Model and ModelType properties are obsolete, for backward compatibility the property getters still work; they delegate to the ModelMetadata property to retrieve the value.

Changes to the DefaultControllerFactory class break custom controller factories that derive from it

The DefaultControllerFactory class was fixed by removing the RequestContext property. In place of this property, the request context instance is passed to the protected virtual GetControllerInstance and GetControllerType

methods. This change affects custom controller factories that derive from DefaultControllerFactory.

Custom controller factories are often used to provide dependency injection for ASP.NET MVC applications. To update the custom controller factories to support ASP.NET MVC 2, change the method signature or signatures to match the new signatures, and use the request context parameter instead of the property.

"Area" is a now a reserved route-value key

The string "area" in Route values now has special meaning in ASP.NET MVC, in the same way that "controller" and "action" do. One implication is that if HTML helpers are supplied with a route-value dictionary containing "area", the helpers will no longer append "area" in the query string.

If you are using the Areas feature, make sure to not use {area} as part of your route URL.

Disclaimer

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, email addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

ASP.NET MVC 3

2/20/2020 • 46 minutes to read • Edit Online

- Overview
- Installation Notes
- Software Requirements
- Documentation
- Support
- Upgrading an ASP.NET MVC 2 Project to ASP.NET MVC 3 Tools Update
- ASP.NET MVC 3 Tools Update (April 12, 2011)
  - "Add Controller" dialog box can now scaffold controllers with views and data access code
  - Improvements to the "ASP.NET MVC 3 New Project" Dialog Box
  - Project templates now include Modernizr 1.7
  - Project templates include updated versions of jQuery, jQuery UI, and jQuery Validation
  - Project templates now include ADO.NET Entity Framework 4.1 as a pre-installed NuGet package
  - Project templates include JavaScript libraries as pre-installed NuGet packages
  - Known Issues
- ASP.NET MVC 3 RTM (January 13, 2011)
  - Change: Updated the version of jQuery UI to 1.8.7
  - Change: Changed the default ModelMetadataProvider back to DataAnnotationsModelMetadataProvider
  - Fixed: Pasting part of a Razor expression that contains whitespace results in it being reversed
  - Fixed: Renaming a Razor file that is opened in the editor disables syntax colorization and IntelliSense
  - Known Issues
  - Breaking Changes
- ASP.NET MVC 3 Release Candidate 2 (December 10, 2010)
  - Project Templates Changed to Include jQuery 1.4.4, jQuery Validation 1.7, and jQuery UI 1.8.6y UI 1.8.6
  - Added "AdditionalMetadataAttribute" Class
  - Improved View Scaffolding
  - Added Html.Raw Method
  - Renamed "Controller.ViewModel" Property and the "View" Property To "ViewBag"
  - Renamed "ControllerSessionStateAttribute" Class to "SessionStateAttribute"
  - Renamed RemoteAttribute "Fields" Property to "AdditionalFields"
  - Renamed "SkipRequestValidationAttribute" to "AllowHtmlAttribute"
  - Changed "Html.ValidationMessage" Method to Display the First Useful Error Message
  - Fixed @model Declaration to not Add Whitespace to the Document
  - Added "FileExtensions" Property to View Engines to Support Engine-Specific File Names
  - Fixed "LabelFor" Helper to Emit the Correct Value for the "For" Attribute
  - Fixed "RenderAction" Method to Give Explicit Values Precedence During Model Binding
  - Breaking Changes

  - Known Issues
- ASP.NET MVC 3 Release Candidate (Nov 9, 2010)
  - New Features in ASP.NET MVC 3 RC
  - NuGet Package Manager
  - Improved "New Project" Dialog Box
  - Sessionless Controllers
  - New Validation Attributes
  - New Overloads for "LabelFor" and "LabelForModel" Methods
  - Child Action Output Caching
  - "Add View" Dialog Box Improvements
  - Granular Request Validation
  - Breaking Changes
  - Known Issues
- ASP.MVC 3 Beta Notes (Oct 6, 2010)
  - New Features in ASP.NET MVC 3 Beta
  - NuPack Package Manager
  - Improved New Project Dialog Box
  - Simplified Way to Specify Strongly Typed Models in Razor Views
  - Support for New ASP.NET Web Pages Helper Methods
  - Additional Dependency Injection Support
  - New Support for Unobtrusive jQuery-Based Ajax
  - New Support for Unobtrusive jQuery Validation
  - New Application-Wide Flags for Client Validation and Unobtrusive JavaScript
  - New Support for Code that Runs Before Views Run
  - New Support for the VBHTML Razor Syntax
  - More Granular Control over ValidateInputAttribute
  - Helpers Convert Underscores to Hyphens for HTML Attribute Names Specified Using Anonymous Objects
  - Bug Fixes
  - Breaking Changes
  - Known Issues
- Disclaimer

Overview

This document describes the release of ASP.NET MVC 3 RTM for Visual Studio 2010. ASP.NET MVC is a framework for developing Web applications that uses the Model-View-Controller (MVC) pattern. The ASP.NET MVC 3 installer includes the following components:

- ASP.NET MVC 3 runtime components
- ASP.NET MVC 3 Visual Studio 2010 tools
- ASP.NET Web Pages run-time components
- ASP.NET Web Pages Visual Studio 2010 tools
- Microsoft Package Manager for .NET (NuGet)
- An update for Visual Studio 2010 that enables support for Razor syntax. (For details, see KnowledgeBase article 2483190.)

The full set of release notes for each pre-release version of ASP.NET MVC 3 can be found on the ASP.NET website

at the following URL:

https://www.asp.net/learn/whitepapers/mvc3-release-notes

Installation Notes

To install ASP.NET MVC 3 RTM using the Web Platform Installer (Web PI), visit the following page:

https://www.microsoft.com/web/gallery/install.aspx?appid=MVC3

Alternatively, you can download the installer for ASP.NET MVC 3 RTM for Visual Studio 2010 from the following page:

https://go.microsoft.com/fwlink/?LinkID=208140

ASP.NET MVC 3 can be installed and can run side-by-side with ASP.NET MVC 2.

Software Requirements

The ASP.NET MVC 3 run-time components require the following software:

- .NET Framework version 4.

  ASP.NET MVC 3 Visual Studio 2010 tools require the following software:

- Visual Studio 2010 or Visual Web Developer 2010 Express.

Documentation

Documentation for ASP.NET MVC is available on the MSDN Web site at the following URL:

https://go.microsoft.com/fwlink/?LinkId=205717

Tutorials and other information about ASP.NET MVC are available on the MVC page of the ASP.NET Web site at the following URL:

https://www.asp.net/mvc/

Support

This is a fully supported release. Information about getting technical support can be found at the Microsoft Support website.

Also feel free to post questions about this release to the ASP.NET MVC forum, where members of the ASP.NET community are frequently able to provide informal support:

https://forums.asp.net/1146.aspx

Upgrading an ASP.NET MVC 2 Project to ASP.NET MVC 3 Tools Update

ASP.NET MVC 3 can be installed side by side with ASP.NET MVC 2 on the same computer, which gives you flexibility in choosing when to upgrade an ASP.NET MVC 2 application to ASP.NET MVC 3.

To manually upgrade an existing ASP.NET MVC 2 application to version 3, do the following:

1. Create a new empty ASP.NET MVC 3 project on your computer. This project will contain some files that are required for the upgrade.

2. Copy the following files from the ASP.NET MVC 3 project into the corresponding location of your ASP.NET MVC 2 project. You'll need to update any references to the jQuery library to account for the new filename

(jQuery-1.5.1.js):

- /Views/Web.config
- /packages.config
- /scripts/*.js
- /Content/themes/*.*

3. Copy the *packages* folder in the root of the empty ASP.NET MVC 3 project solution into the root of your solution, which is in the directory where the solution's .sln file is located.

4. If your ASP.NET MVC 2 project contains any areas, copy the /Views/Web.config file to the *Views* folder of each area.

5. In both Web.config files in the ASP.NET MVC 2 project, globally search and replace the ASP.NET MVC version. Find the following:

   ```
   System.Web.Mvc, Version=2.0.0.0
   ```

   Replace it with the following:

   ```
   System.Web.Mvc, Version=3.0.0.0
   ```

6. In Solution Explorer, delete the reference to *System.Web.Mvc* (which points to the DLL from version 2), then add a reference to *System.Web.Mvc* (v3.0.0.0).

7. Add a reference to System.Web.WebPages.dll and System.Web.Helpers.dll. These assemblies are located in the following folders:

   - %ProgramFiles%\ Microsoft ASP.NET\ASP.NET MVC 3\Assemblies
   - %ProgramFiles%\ Microsoft ASP.NET\ASP.NET Web Pages\v1.0\Assemblies

8. In Solution Explorer, right-click the project name and select Unload Project. Then right-click the project name again and select Edit *ProjectName*.csproj.

9. Locate the *ProjectTypeGuids* element and replace {F85E285D-A4E0-4152-9332-AB1D724D3325} with {E53F8FEA-EAE0-44A6-8774-FFD645390401}.

10. Save the changes, right-click the project, and then select Reload Project.

11. In the application's root Web.config file, add the following settings to the *assemblies* section.

    ```
    <add assembly="System.Web.WebPages, Version=1.0.0.0, Culture=neutral,
         PublicKeyToken=31BF3856AD364E35" />

    <add assembly="System.Web.Helpers, Version=1.0.0.0, Culture=neutral,
         PublicKeyToken=31BF3856AD364E35" />
    ```

12. If the project references any third-party libraries that are compiled using ASP.NET MVC 2, add the following highlighted *bindingRedirect* element to the Web.config file in the application root under the *configuration* section:

```
<runtime>
   <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
     <dependentAssembly>
       <assemblyIdentity name="System.Web.Mvc"
           publicKeyToken="31bf3856ad364e35"/>
       <bindingRedirect oldVersion="1.0.0.0-2.0.0.0" newVersion="3.0.0.0"/>
     </dependentAssembly>
   </assemblyBinding>
</runtime>
```

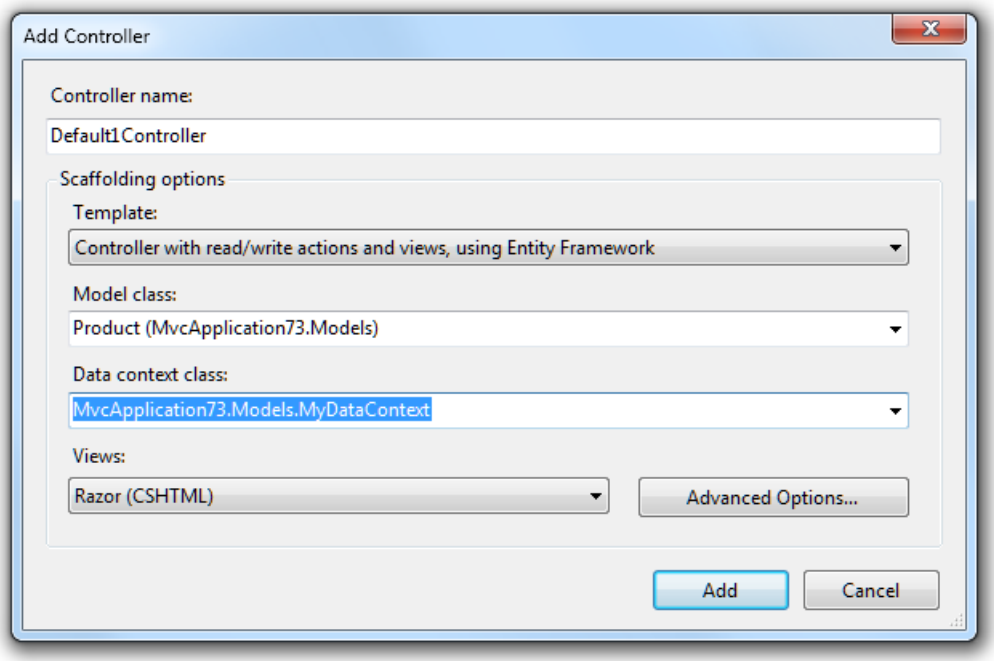
Changes in ASP.NET MVC 3 Tools Update

This section describes changes made in the ASP.NET MVC 3 Tools Update release since the ASP.NET MVC 3 RTM release.

"Add Controller" dialog box can now scaffold controllers with views and data access code

Scaffolding is a way of quickly generating a controller and views for your application. After the code has been generated, you can edit it to meet your project's requirements.

To launch the *Add Controller* dialog box in ASP.NET MVC 3, right-click the *Controllers* folder in *Solution Explorer*, click *Add*, and then click *Controller*. The dialog box has been enhanced to offer additional scaffolding options.



There are three scaffolding templates available by default.

Empty Controller

This template generates an empty controller file. This template is equivalent to not checking *Add actions for create, edit, details, delete scenarios* in previous versions of ASP.NET MVC. If you choose this, no further options are available.

Controller with empty read/write actions

This template generates a controller file that has all the required action methods but no implementation code in the methods. This template is equivalent to checking *Add actions for create, edit, details, delete scenarios* in previous versions of ASP.NET MVC. If you choose this, no further options are available.

Controller with read/write actions and views, using Entity Framework

This template enables you to quickly create a working data-entry user interface. It generates code that handles a range of common requirements and scenarios, such as the following:

- *Data access*. The generated code reads and writes entities in a database. It works with the Entity Framework Code First approach if you choose an existing data context class or if you let the template generate a new *DbContext* class. It also works with the Entity Framework Database First or Model First approach if you choose an existing *ObjectContext* class.
- *Validation*. The generated code uses ASP.NET MVC model binding and metadata features so that form submissions are validated according to rules declared on your model class. This includes built-in validation rules, such as the *Required* and *StringLength* attributes, and custom validation rules.
- *One-to-many relationships*. If you define one-to-many foreign-key relationships between your model classes, the generated code will produce drop-down lists for selecting related entities. For example, you might define the following model classes following Entity Framework Code First conventions:

```
public class Product
{
     public int ProductId { get; set; }
     [Required]
     public string Name { get; set; }

     // Product belongs to Category
     public int CategoryId { get; set; }
     public virtual Category Category { get; set; }
}
public class Category
{
     public int CategoryId { get; set; }
     [Required]
     public string Name { get; set; }
}
```

  When you then scaffold a controller for the *Product* class, its views will allow users to choose a *Category* object for each *Product* instance.

  This template enables additional options in the *Add Controller* dialog box. For *Model class*, you can choose any model class in your solution, which determines the type of data that users will be able to create or edit:

- If you want to use Entity Framework Code First, you can choose any model class.
- If you are using Entity Framework Database First or Entity Framework Model First, be sure to choose an entity class defined in your conceptual model.

For *Data Context class*, you can make these choices:

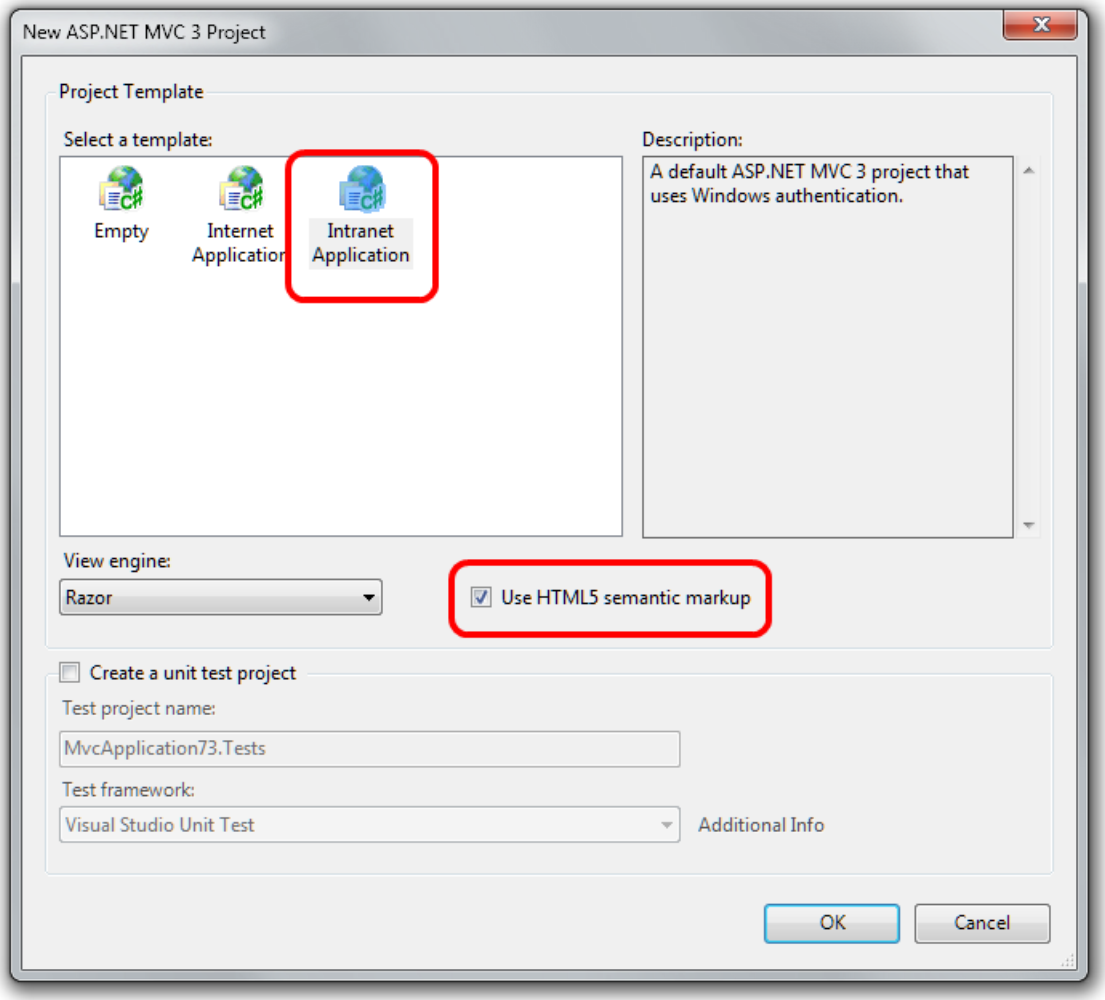
- If you want to use Code First and have no existing data context class, choose **New data context **. A data context class will then be generated for you.
- If you want to use Code First and have an existing data context class, choose it here. It will be updated to persist the model class you have selected.
- If you are using Database First or Model First, choose your object context class here.

For Views, choose the view engine you want to use, or choose None if you don't want to scaffold any views.

You can select Advanced Optionsto specify further options for the generated views. For example, you can choose the layout or master page to use.

Improvements to the "ASP.NET MVC 3 New Project" Dialog Box

The dialog box you use to create new ASP.NET MVC 3 projects includes multiple improvements, as listed below.



New "Intranet Project" Template

The Project Template list includes a new Intranet Application template. This template contains settings for building a web application using Windows authentication instead of forms authentication. Because an intranet application requires some IIS settings that can't be encapsulated in a project template, the template includes a readme file with instructions for how to make the project template work in IIS. Documentation for the a new Intranet Application template is available on the MSDN website at the following URL:

https://msdn.microsoft.com/library/gg703322(VS.98).aspx

Project templates are now HTML5 enabled

The new-project dialog box now contains an option to add HTML5-specific features to the project templates. Selecting the option causes views to be generated that contain the new HTML5 `<header>`, `<footer>`, and `<navigation>` elements.

Note that earlier versions of browsers do not support HTML5-specific tags. To address this limitation, the HTML5 project templates include a reference to the Modernizr library. (See the next section.)

Project templates now include Modernizr 1.7

Modernizr is a JavaScript library that enables support for CSS 3 and HTML5 in browsers that do not yet support these features. This library is included as a pre-installed NuGet package in templates for ASP.NET MVC 3 projects. For more information about Modernizr, see http://www.modernizr.com/.

Project templates include updated versions of jQuery, jQuery UI, and jQuery Validation

The project templates now include the following versions of the jQuery scripts:

- jQuery 1.5.1
- jQuery Validation 1.8
- jQuery UI 1.8.11

These libraries are included as pre-installed NuGet packages.

Project templates now include ADO.NET Entity Framework 4.1 as a pre-installed NuGet package

The ADO.NET Entity Framework 4.1 includes the Code First feature. Code First is a new development pattern for the ADO.NET Entity Framework that provides an alternative to the existing Database First and Model First patterns.

Code First is focused around defining your model using POCO classes ("plain old CLR objects") written in Visual Basic or C#. These classes can then be mapped to an existing database or be used to generate a database schema. Additional configuration can be supplied using *DataAnnotations* attributes or using fluent APIs.

Documentation for using Code Firstwith ASP.NET MVC is available on the ASP.NET website at the following URLs:

https://www.asp.net/mvc/tutorials/getting-started-with-mvc3-part1-cs https://www.asp.net/entity-framework/tutorials/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application

Project templates include JavaScript libraries as pre-installed NuGet packages

When you create a new ASP.NET MVC 3 project, the project includes the JavaScript files mentioned previously (for example, the Modernizr library) by installing them using NuGet instead of directly adding the scripts to the Scripts folder in the project template contents. This enables you to use NuGet to update the scripts to the latest version when new versions of the scripts are released.

For example, given the frequency of new jQuery releases, the version of jQuery included in the project template will at some point be out of date. However, because jQuery is included as an installed NuGet package, you will be notified in the NuGet dialog box when newer versions of jQuery are available.

Because jQuery includes the version number in the file name, updating jQuery to the latest version also requires updating the `<script>` tag that references the jQuery file to use the new file name. Other included script libraries do not include the version number in the script name, so they can be more easily updated to their latest versions.

Known Issues

- In some cases, installation may fail with the error message "Installation failed with error code (0x80070643)". For information about how to work around this issue, see KnowledgeBase article 2531566.
- The scaffolding for adding a controller does not scaffold entities that take advantage of entity inheritance support within Entity Framework. For example, given a base *Person* class that's inherited by a *Student* class, scaffolding the *Student* class will result in generated code that does not compile.
- Creating a new ASP.NET MVC 3 project within a solution folder causes a *NullReferenceException* error. The workaround is to create the ASP.NET MVC 3 project in the root of the solution and then move it into the solution folder.
- IntelliSense for Razor syntax does not work when ReSharper is installed. If you have ReSharper installed and want to take advantage of the Razor IntelliSense support in ASP.NET MVC 3, see the entry Razor Intellisense and ReSharper on Hadi Hariri's blog, which discusses ways to use them together today.
- During installation, the EULA acceptance dialog box displays the license terms in a window that is smaller than intended.
- When you are editing a Razor view (.cshtml or *.vbhtml* file), views. ASP.NET MVC 3 does not include any snippets for Razor views..aspxselecting a code snippet for ASP.NET MVC will show snippets for
- If you install ASP.NET MVC 3 for Visual Web Developer Express on a computer where Visual Studio is not installed, and then later install Visual Studio, you must reinstall ASP.NET MVC 3. Visual Studio and Visual Web Developer Express share components that are upgraded by the ASP.NET MVC 3 installer. The same issue applies if you install ASP.NET MVC 3 for Visual Studio on a computer that does not have Visual Web Developer Express and then later install Visual Web Developer Express.

Changes in ASP.NET MVC 3 RTM

This section describes changes and bug fixes made in the ASP.NET MVC 3 RTM release since the RC2 release.

Change: Updated the version of jQuery UI to 1.8.7

The ASP.NET MVC project templates for Visual Studio were updated to include the latest version of the jQuery UI library. The templates also include the minimal set of resource files required by jQuery UI, such as the associated CSS and image files.

Change: Changed the default ModelMetadataProvider back to DataAnnotationsModelMetadataProvider

The RC2 release of ASP.NET MVC 3 introduced a *CachedDataAnnotationsMetadataProvider* class that provided caching on top of the existing *DataAnnotationsModelMetadataProvider* class as a performance improvement. However, some bugs were reported with this implementation, so the change has been reverted and moved into the MVC Futures project, which is available at ASP.NET WebStack.

Fixed: Pasting part of a Razor expression that contains whitespace results in it being reversed

In pre-release versions of ASP.NET MVC 3, when you paste a part of a Razor expression that contains whitespace into a Razor file, the resulting expression is reversed. For example, consider the following Razor code block:

```
@SomeMethod("first param",
100)
@AnotherMethod()
```

If you select the text "first param" in the first method and paste it as an argument into the second method, the result is as follows:

```
@AnotherMethod(param""first)
```

The correct behavior is that the paste operation should result in the following:

```
@AnotherMethod("first param")
```

This issue has been fixed in the RTM release so that the expression is correctly preserved during the paste operation.

Fixed: Renaming a Razor file that is opened in the editor disables syntax colorization and IntelliSense

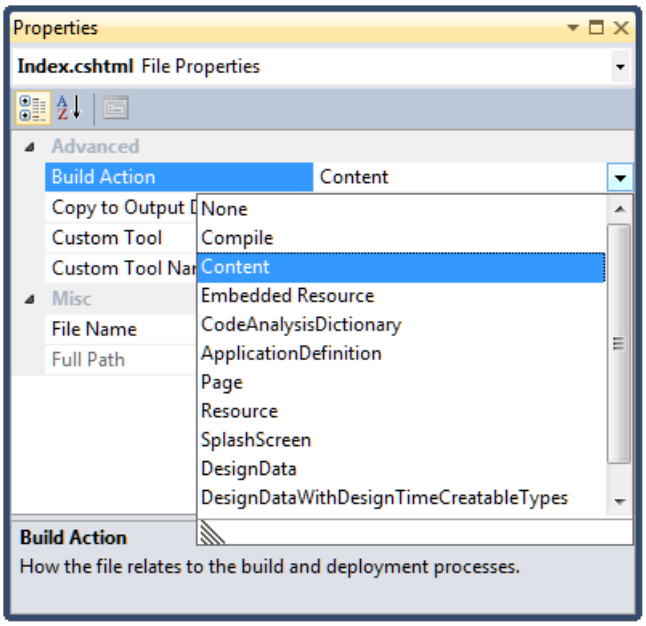
Renaming a Razor file using Solution Explorer while the file is opened in the editor window causes syntax highlighting and IntelliSense to stop working for that file. This has been fixed so that highlighting and IntelliSense are maintained after a rename.

Known Issues

- If you close Visual Studio 2010 SP1 Beta while the NuGet Package Manager Console is open, Visual Studio crashes and attempts to restart. This will be fixed in the RTM release of Visual Studio 2010 SP1.
- The ASP.NET MVC 3 installer is only able to install an initial version of the NuGet package manager. After you have installed the initial version, NuGet can be installed and updated using Visual Studio Extension Manager. If you already have NuGet installed, go to the Visual Studio Extension Gallery to update to the latest version of NuGet.
- Creating a new ASP.NET MVC 3 project within a solution folder causes a *NullReferenceException* error. The workaround is to create the ASP.NET MVC 3 project in the root of the solution and then move it into the solution folder.
- The installer might take much longer than previous versions of ASP.NET MVC to complete. This is because it

updates components of Visual Studio 2010.

- IntelliSense for Razor syntax does not work when ReSharper is installed. If you have ReSharper installed and want to take advantage of the Razor IntelliSense support in ASP.NET MVC 3, see the entry Razor Intellisense and ReSharper on Hadi Hariri's blog, which discusses ways to use them together today.
- CCSHTML and VBHTML views created with the Beta version of ASP.NET MVC 3 do not have their build action set correctly, with the result that these view types are omitted when the project is published. The Build Action value for these files should be set to "Content". ASP.NET MVC 3 RTM fixes this issue for new files, but doesn't correct the setting for existing files for a project created with prerelease versions.



-
- During installation, the EULA acceptance dialog box displays the license terms in a window that is smaller than intended.
- When you are editing a Razor view (.cshtml file), the Go To Controller menu item in Visual Studio will not be available, and there are no code snippets.
- If you install ASP.NET MVC 3 for Visual Web Developer Express on a computer where Visual Studio is not installed, and then later install Visual Studio, you must reinstall ASP.NET MVC 3. Visual Studio and Visual Web Developer Express share components that are upgraded by the ASP.NET MVC 3 installer. The same issue applies if you install ASP.NET MVC 3 for Visual Studio on a computer that does not have Visual Web Developer Express and then later install Visual Web Developer Express.

Breaking Changes

- In previous versions of ASP.NET MVC, action filters are create per request except in a few cases. This behavior was never a guaranteed behavior but merely an implementation detail and the contract for filters was to consider them stateless. In ASP.NET MVC 3, filters are cached more aggressively. Therefore, any custom action filters which improperly store instance state might be broken.
- The order of execution for exception filters has changed for exception filters that have the same *Order* value. In ASP.NET MVC 2 and earlier, exception filters on the controller that have the same *Order* value as those on an action method are executed before the exception filters on the action method. This would typically be the case when exception filters are applied without a specified *Order* value. In ASP.NET MVC 3, this order has been reversed so that the most specific exception handler executes first. As in earlier versions, if the *Order* property is explicitly specified, the filters are run in the specified order.
- A new property named *FileExtensions* was added to the *VirtualPathProviderViewEngine* base class. When ASP.NET looks up a view by path (not by name), only views with a file extension contained in the list specified by this new property are considered. This is a breaking change in applications where a custom build provider is registered in order to enable a custom file extension for Web Form views and where the provider

references those views by using a full path rather than a name. The workaround is to modify the value of the *FileExtensions* property to include the custom file extension.

- Custom controller factory implementations that directly implement the *IControllerFactory* interface must provide an implementation of the new *GetControllerSessionBehavior* method that was added to the interface in this release. In general, it is recommended that you do not implement this interface directly and instead derive your class from *DefaultControllerFactory*.

Changes in ASP.NET MVC 3 RC2

This section describes changes (new features and bug fixes) made in the ASP.NET MVC 3 RC2 release since the RC release.

Project Templates Changed to Include jQuery 1.4.4, jQuery Validation 1.7, and jQuery UI 1.8.6

The project templates for ASP.NET MVC 3 now include the latest versions of jQuery, jQuery Validation, and jQuery UI. jQuery UI is a new addition to the project templates and provides useful user interface widgets. For more information about jQuery UI, visit their homepage: http://jqueryui.com/.

Added "AdditionalMetadataAttribute" Class

You can use the *AdditionalMetadataAttribute* class to populate the *ModelMetadata.AdditionalValues* dictionary for a model property.

For example, suppose a view model has properties that should be displayed only to an administrator. That model can be annotated with the new attribute using AdminOnly as the key and true as the value, as in the following example:

```
public class ProductViewModel {
    [AdditionalMetadata("AdminOnly", true)]
    public string RefundCode {get; set;}
}
```

This metadata is made available to any display or editor template when a product view model is rendered. It is up to you as application developer to interpret the metadata information.

Improved View Scaffolding

The T4 templates used for scaffolding views now generate calls to template helper methods such as *EditorFor* instead of helpers such as *TextBoxFor*. This change improves support for metadata on the model in the form of data annotation attributes when the Add View dialog box generates a view.

The Add View scaffolding also includes improved detection and usage of primary key information on the model, based on convention. For example, the Add View dialog box uses this information to ensure that the primary key value is not scaffolded as an editable form field.

The default Edit and Create templates include references to the jQuery scripts needed for client validation.

Added Html.Raw Method

By default, the Razor view engine HTML-encodes all values. For example, the following code snippet encodes the HTML inside the greeting variable so that it is displayed in the page as `<strong>Hello World!</strong>`.

```
@{
  string greeting = "<strong>Hello World!</strong>";
}

<p>@greeting</p>
```

The new *Html.Raw* method provides a simple way of displaying unencoded HTML when the content is known to

be safe. The following example displays the same string, but the string is rendered as markup:

```
@{
  string greeting = "<strong>Hello World!</strong>";
}

<p>@Html.Raw(greeting)</p>
```

Renamed "Controller.ViewModel" Property and the "View" Property To "ViewBag"

Previously, the *ViewModel* property of *Controller* corresponded to the *View* property of a view. Both of these properties provide a way to access values of the *ViewDataDictionary* object using dynamic property-accessor syntax. Both properties have been renamed to be the same in order to avoid confusion and to be more consistent.

Renamed "ControllerSessionStateAttribute" Class to "SessionStateAttribute"

The *ControllerSessionStateAttribute* class was introduced in the RC release of ASP.NET MVC 3. The property was renamed to be more succinct.

Renamed RemoteAttribute "Fields" Property to "AdditionalFields"

The *RemoteAttribute* class's *Fields* property caused some confusion among users. Renaming this property to *AdditionalFields* clarifies its intent.

Renamed "SkipRequestValidationAttribute" to "AllowHtmlAttribute"

The *SkipRequestValidationAttribute* attribute was renamed to *AllowHtmlAttribute* to better represent its intended usage.

Changed "Html.ValidationMessage" Method to Display the First Useful Error Message

The *Html.ValidationMessage* method was fixed to show the first useful error message instead of simply displaying the first error.

During model binding, the *ModelState* dictionary can be populated from multiple sources with error messages about the property, including from the model itself (if it implements *IValidatableObject*), from validation attributes applied to the property, and from exceptions thrown while the property is being accessed.

When the *Html.ValidationMessage* method displays a validation message, it skips model-state entries that include an exception, because these are generally not intended for the end user. Instead, the method looks for the first validation message that is not associated with an exception and displays that message. If no such message is found, it defaults to a generic error message that is associated with the first exception.

Fixed @model Declaration to not Add Whitespace to the Document

In earlier releases, the `@model` declaration at the top of a view added a blank line to the rendered HTML output. This has been fixed so that the declaration does not introduce whitespace.

Added "FileExtensions" Property to View Engines to Support Engine-Specific File Names

A view engine can return a view using an explicit view path as in the following example:

```
return View("~/views/home/index.cshtml");
```

The first view engine always attempts to render the view. By default, the Web Forms view engine is the first view engine; because the Web Forms engine cannot render a Razor view, an error occurs. View engines now have a *FileExtensions* property that is used to specify which file extensions they support. This property is checked when ASP.NET determines whether a view engine can render a file. This is a breaking change and more details are included in the Breaking Changes section of this document.

Fixed "LabelFor" Helper to Emit the Correct Value for the "For" Attribute

A bug was fixed where the *LabelFor* method rendered a *for* attribute that matches the *input* element's *name* attribute instead of its ID. According to the W3C, the *for* attribute should match the *input* element's ID.

Fixed "RenderAction" Method to Give Explicit Values Precedence During Model Binding

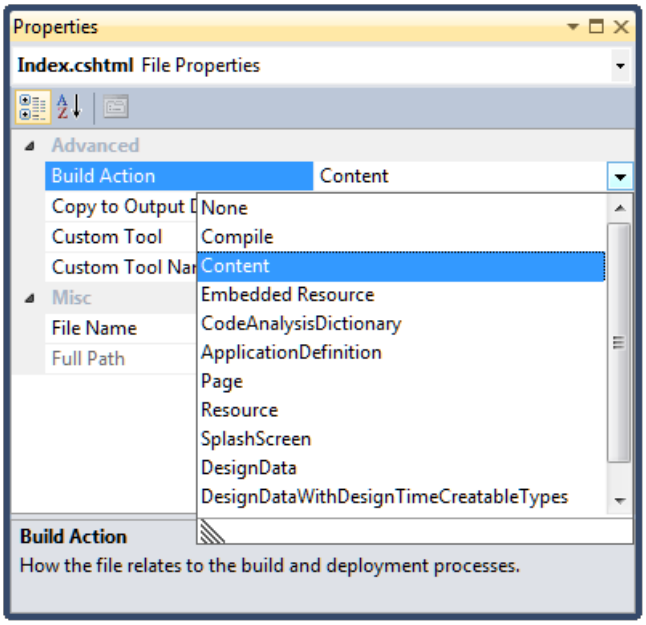
In earlier versions, explicit values that were passed to the *RenderAction* method were being ignored in favor of the current form values during model binding inside a child action. The fix ensures that explicit values take precedence during model binding.

Breaking Changes

- In previous versions of ASP.NET MVC, action filters were created per request except in a few cases. This behavior was never a guaranteed behavior but merely an implementation detail and the contract for filters was to consider them stateless. In ASP.NET MVC 3, filters are cached more aggressively. Therefore, any custom action filters which improperly store instance state might be broken.
- The order of execution for exception filters has changed for exception filters that have the same *Order* value. In ASP.NET MVC 2 and earlier, exception filters on the controller that had the same *Order* value as those on an action method were executed before the exception filters on the action method. This would typically be the case when exception filters were applied without a specified *Order* value. In ASP.NET MVC 3, this order has been reversed so that the most specific exception handler executes first. As in earlier versions, if the *Order* property is explicitly specified, the filters are run in the specified order.
- A new property named *FileExtensions* was added to the *VirtualPathProviderViewEngine* base class. When ASP.NET looks up a view by path (not by name), only views with a file extension contained in the list specified by this new property are considered. This is a breaking change in applications where a custom build provider is registered in order to enable a custom file extension for Web Form views and where the provider references those views by using a full path rather than a name. The workaround is to modify the value of the *FileExtensions* property to include the custom file extension.
- Custom controller factory implementations that directly implement the *IControllerFactory* interface must provide an implementation of the new *GetControllerSessionBehavior* method that was added to the interface in this release. In general, it is recommended that you do not implement this interface directly and instead derive your class from *DefaultControllerFactory*.

Known Issues

- The ASP.NET MVC 3 installer is only able to install an initial version of the NuGet package manager. After you have installed the initial version, NuGet can be installed and updated using Visual Studio Extension Manager. If you already have NuGet installed, go to the Visual Studio Extension Gallery to update to the latest version of NuGet.
- Creating a new ASP.NET MVC 3 project within a solution folder causes a *NullReferenceException* error. The workaround is to create the ASP.NET MVC 3 project in the root of the solution and then move it into the solution folder.
- The installer might take much longer than previous versions of ASP.NET MVC to complete. This is because it updates components of Visual Studio 2010.
- IntelliSense for Razor syntax does not work when ReSharper is installed. If you have ReSharper installed and want to take advantage of the Razor IntelliSense support in ASP.NET MVC 3 RC2, see the entry Razor Intellisense and ReSharper on Hadi Hariri's blog, which discusses ways to use them together today.
- CSHTML and VBHTML views created with the Beta version of ASP.NET MVC 3 do not have their build action set correctly, with the result that these view types are omitted when the project is published. The *Build Action* value for these files should be set to Content". ASP.NET MVC 3 RC2 fixes this issue for new files, but doesn't correct the setting for existing files for a project created with the Beta version.



- During installation, the EULA acceptance dialog box displays the license terms in a window that is smaller than intended.
- When you are editing a Razor view (.cshtml file), the Go To Controller menu item in Visual Studio will not be available, and there are no code snippets.
- If you install ASP.NET MVC 3 for Visual Web Developer Express on a computer where Visual Studio is not installed, and then later install Visual Studio, you must reinstall ASP.NET MVC 3. Visual Studio and Visual Web Developer Express share components that are upgraded by the ASP.NET MVC 3 installer. The same issue applies if you install ASP.NET MVC 3 for Visual Studio on a computer that does not have Visual Web Developer Express and then later install Visual Web Developer Express.
- Installing ASP.NET MVC 3 RC 2 does not update NuGet if you already have it installed. To upgrade NuGet, go to the Visual Studio Extension manager and it should show up as an available update. You can upgrade NuGet to the latest release from there.

ASP.NET MVC 3 Release Candidate

ASP.NET MVC Release Candidate was released on November 9, 2010.

New Features in ASP.NET MVC 3 RC

This section describes features that have been introduced in the ASP.NET MVC 3 RC release since the Beta release.

NuGet Package Manager

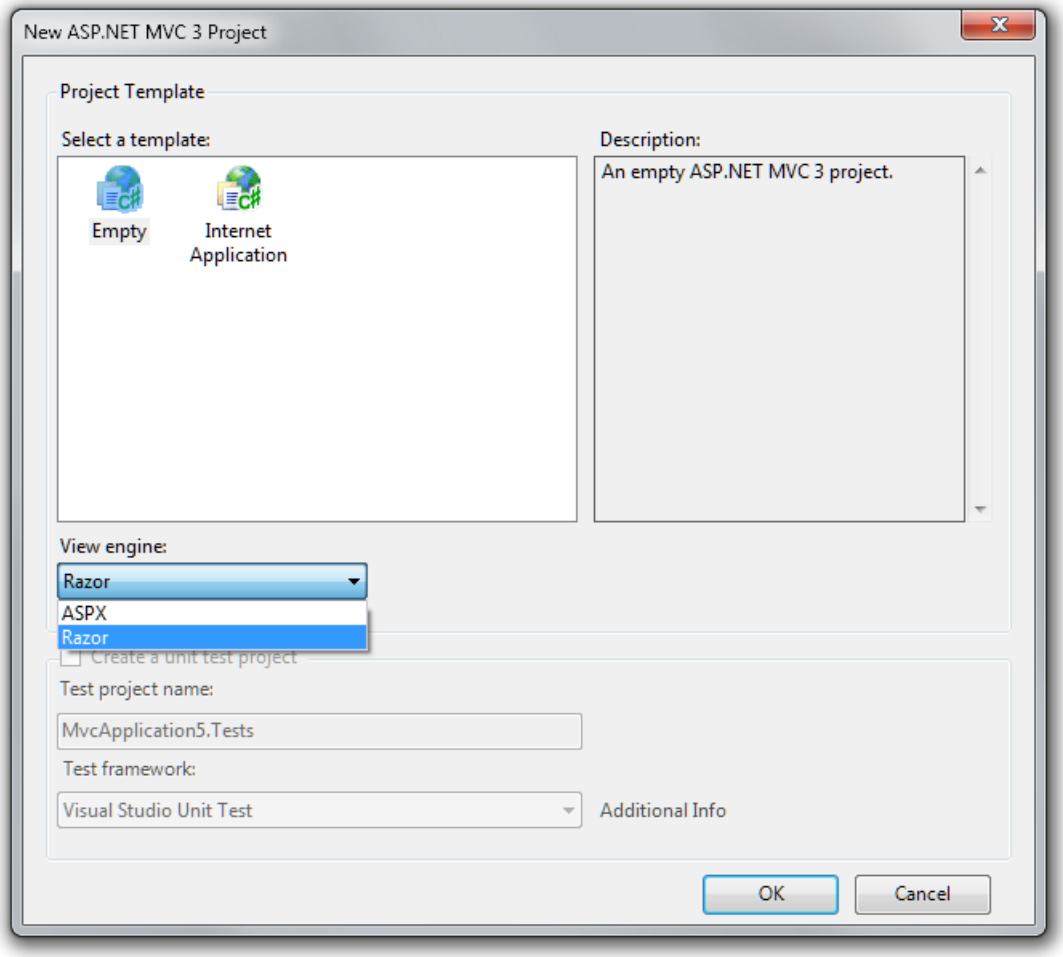
ASP.NET MVC 3 includes the NuGet Package Manager (formerly known as NuPack), which is an integrated package management tool for adding libraries and tools to Visual Studio projects. This tool automates the steps that developers take today to get a library into their source tree.

You can work with NuGet as a command-line tool, as an integrated console window inside Visual Studio 2010, from the Visual Studio context menu, and as a set of PowerShell cmdlets.

For more information about NuGet, read the Nuget Documentation.

Improved "New Project" Dialog Box

When you create a new project, the New Project dialog box now lets you specify the view engine as well as an ASP.NET MVC project type.



Support for modifying the list of templates and view engines listed in the dialog box is included in this release.

The default templates are the following:

Empty. Contains a minimal set of files for an ASP.NET MVC project, including the default directory structure for ASP.NET MVC projects, a Site.css file that contains the default ASP.NET MVC styles, and a Scripts directory that contains the default JavaScript files.

Internet Application. Contains sample functionality that demonstrates how to use the membership provider with ASP.NET MVC.

The list of project templates that is displayed in the dialog box is specified in the Windows registry.

Sessionless Controllers

The new *ControllerSessionStateAttribute* gives you more control over session-state behavior for controllers by specifying a System.Web.SessionState.SessionStateBehavior enumeration value.

The following example shows how to turn off session state for all requests to a controller.

```
[ControllerSessionState(SessionStateBehavior.Disabled)]
public class CoolController : Controller {
    public ActionResult Index() {
        object o = Session["Key"]; // Causes an exception.

    }
}
```

The following example shows how to set read-only session state for all requests to a controller.

```
[ControllerSessionState(SessionStateBehavior.ReadOnly)]
public class CoolController : Controller {
    public ActionResult Index() {
    Session["Key"] = "value"; // Value is not available in
    the next request
    }
}
```

New Validation Attributes

CompareAttribute

The new *CompareAttribute* validation attribute lets you compare the values of two different properties of a model. In the following example, the *ComparePassword* property must match the *Password* field in order to be valid.

```
public class User {
    [Required]

    public string Password { get; set; }
    [Required, Compare("Password")]
    public string ComparePassword { get; set; }
}
```

RemoteAttribute

The new *RemoteAttribute* validation attribute takes advantage of the jQuery Validation plug-in's remote validator, which enables client-side validation to call a method on the server that performs the actual validation logic.

In the following example, the *UserName* property has the *RemoteAttribute* applied. When editing this property in an Edit view, client validation will call an action named *UserNameAvailable* on the *UsersController* class in order to validate this field.

```
public class User {
    [Remote("UserNameAvailable", "Users")]
    public string UserName { get; set; }
}
```

The following example shows the corresponding controller.

```
public class UsersController {
    public bool UserNameAvailable(string username) {
        if(MyRepository.UserNameExists(username)) {
            return "false";
        }
        return "true";
    }
}
```

By default, the property name that the attribute is applied to is sent to the action method as a query-string parameter.

New Overloads for "LabelFor" and "LabelForModel" Methods

New overloads have been added for the *LabelFor* and *LabelForModel* methods that let you specify the label text. The following example shows how to use these overloads.

```
@Html.LabelFor(m => m.PropertyName,
"Label Text");
@Html.LabelForModel("Label Text");
```

Child Action Output Caching

The *OutputCacheAttribute* supports output caching of child actions that are called by using the *Html.RenderAction* or *Html.Action* helper methods. The following example shows a view that calls another action.

```
Hi there. The uncached time is:
@DateTime.Now
The cached time is: @Html.Action("GetDate")
```

The *GetDate* action is annotated with the *OutputCacheAttribute*:

```
[OutputCache(Duration = 100,
VaryByParam = "none")]
public string GetDate() {
    return DateTime.Now.ToString();
}
```

When this code runs, the result of the call to Html.Action("GetDate") is cached for 100 seconds.

"Add View" Dialog Box Improvements

When you add a strongly typed view, the Add View dialog box now filters out more non-applicable types than in previous releases, such as many core .NET Framework types. Also, the list is now sorted by the class name and not by the fully qualified type name, which makes it easier to find types. For example, the type name is now displayed as in the following example:

ClassName (namespace)

In earlier releases, this would have been displayed as the following:

Namespace.ClassName

Granular Request Validation

The *Exclude* property of *ValidateInputAttribute* no longer exists. Instead, to have request validation skipped for specific properties of a model during model binding, use the new *SkipRequestValidationAttribute*.

For example, suppose an action method is used to edit a blog post:

```
[HttpPost]
public ActionResult Edit(BlogPostViewModel post) {
    // Save the post in the database
}
```

The following example shows the view model for a blog post.

```
public class BlogPostViewModel {
    public int Id {get; set;}

    public string Subject {get; set;}

    public string Description {get; set;}
}
```

When a user submits some markup for the Description property, model binding will fail due to request

validation. To disable request validation during model binding for the blog post Description, apply the *SkipRequpestValidationAttribute* to the property, as shown in this example:.

```
public class BlogPostViewModel {
    public int Id {get; set;}

    public string Subject {get; set;}

    [SkipRequestValidation]

    public string Description {get; set;}
}
```

Alternatively, to turn off request validation for every property of the model, apply *ValidateInputAttribute* with a value of *false* to the action method:

```
[HttpPost]
[ValidateInput(false)]
public ActionResult Edit(BlogPostViewModel post) {
    // Save the post in the database
}
```

Breaking Changes

- The order of execution for exception filters has changed for exception filters that have the same *Order* value. In ASP.NET MVC 2 and earlier, exception filters on the controller that had the same *Order* as those on an action method were executed before the exception filters on the action method. This would typically be the case when exception filters were applied without a specified *Order* value. In ASP.NET MVC 3, this order has been reversed so that the most specific exception handler executes first. As in earlier versions, if the *Order* property is explicitly specified, the filters are run in the specified order.
- Added a new property named *FileExtensions* to the *VirtualPathProviderViewEngine* base class. When looking up a view by path (and not by name), only views with a file extension contained in the list specified by this new property is considered. This is a breaking change for those who register a custom build provider to enable a custom file extension for web form views and are referencing those views by using a full path rather than a name. The workaround is to modify the value of the *FileExtensions* property to include the custom file extension.

Known Issues

- The installer may take much longer than previous versions of ASP.NET MVC to complete because it updates components of Visual Studio 2010.
- The Add View scaffolding when selecting astrongly typed view scaffolds write-only properties. These should always be ignored by scaffolding. The Add View dialog also scaffolds read-only properties when generating an "Edit" or "Create" view. Read-only properties should only be scaffolded for the Display and List views.
- Debugging doesn't work when ASP.NET MVC 3 is installed alongside the Async CTP. ASP.NET MVC 3 cannot be installed side-by-side with the Async CTP. Uninstall the Async CTP to repair debugging. For more details, read this blog post about uninstalling all the pieces of ASP.NET MVC 3 RC.
- Razor Intellisense does not work when Resharper is installed. If you have ReSharper installed and want to take advantage of the Razor intellisense support in ASP.NET MVC 3 RC, please read this blog post from JetBrains which discusses ways to use them together today.
- CSHTML and VBHTML views created with Beta of ASP.NET MVC 3 do not have their build action correctly which omits them from publishing. The *Build Action* for these files should be set to "Content". ASP.NET MVC 3 RC fixes this issue for new files, but doesn't correct the setting for existing files for a project created with the

Beta.

- The installer may take much longer than previous versions of ASP.NET MVC to complete because it updates components of Visual Studio 2010.
- The Add View scaffolding when selecting an "Edit" strongly typed view scaffolds read only properties. Likewise, write-only properties are scaffolded for "Display" views.
- During installation, the EULA acceptance dialog box displays the license terms in a window that is smaller than intended.
- Installing the Visual Studio Async CTP causes a conflict with the Razor release that is included as part of the ASP.NET MVC 3 tooling installation. Make sure that you do not try to install both the Visual Studio Async CTP and the Razor release on the same machine.
- When you are editing a Razor view (.cshtml file), the Go To Controller menu item in Visual Studio will not be available, and there are no code snippets.

ASP.NET MVC 3 Beta

ASP.NET MVC 3 Beta was released on October 6, 2010. The following notes are specific to the Beta release, and are subject to any updates or changes referenced in the ASP.NET MVC 3 Release Candidate section above.

New Featuresin ASP.NET MVC 3 Beta

This section describes features that have been introduced in the ASP.NET MVC 3 Beta release.

NuGet Package Manager

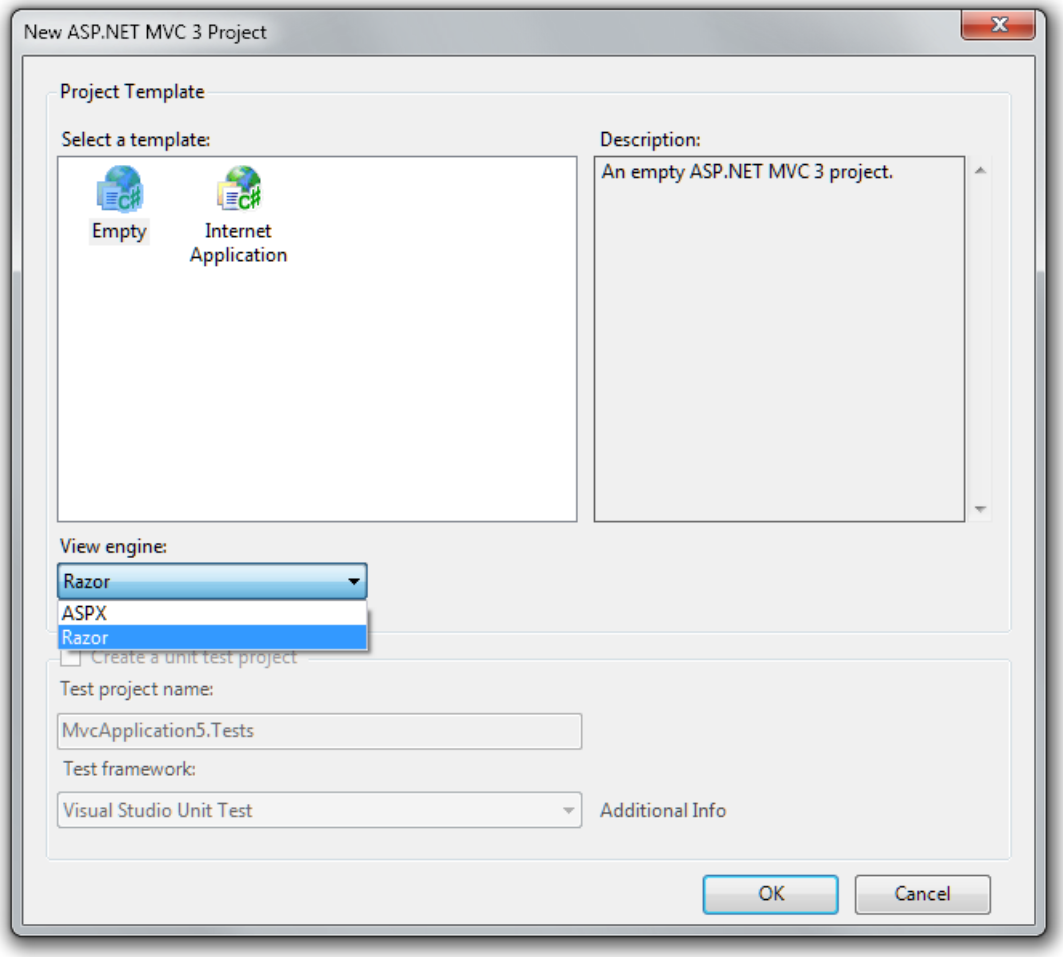
ASP.NET MVC 3 includes NuGet Package Manager, which is an integrated package management tool for adding libraries and tools to Visual Studio projects. For the most part, it automates the steps that developers take today to get a library into their source tree.

You can work with NuGet as a command line tool, as an integrated console window inside Visual Studio 2010, from the Visual Studio context menu, and as set of PowerShell cmdlets.

For more information about NuGet, read the NuGet Documentation.

Improved New Project Dialog Box

When you create a new project, the New Project dialog box now lets you specify the view engine as well as an ASP.NET MVC project type.



Support for modifying the list of templates and view engines listed in the dialog box is not included in this release.

The default templates are the following:

Empty. Contains a minimal set of files for an ASP.NET MVC project, including the default directory structure for ASP.NET MVC projects, a small Site.css file that contains the default ASP.NET MVC styles, and a Scripts directory that contains the default JavaScript files.

Internet Application. Contains sample functionality that demonstrates how to use the membership provider within ASP.NET MVC.

Simplified Way to Specify Strongly Typed Models in Razor Views

The way to specify the model type for strongly typed Razor views has been simplified by using the new @model directive for CSHTML views and @ModelType directive for VBHTML views. In earlier versions of ASP.NET MVC, you would specify a strongly typed model for Razor views this way:

```
@inherits System.Web.Mvc.WebViewPage
```

In this release, you can use the following syntax:

```
@model MyModelNamespace.MyModelType
```

Support for New ASP.NET Web Pages Helper Methods

The new ASP.NET Web Pages technology includes a set of helper methods that are useful for adding commonly used functionality to views and controllers. ASP.NET MVC 3 supports using these helper methods within controllers and views (where appropriate). These methods are contained in the System.Web.Helpers assembly.

The following table lists a few of the ASP.NET Web Pages helper methods.

HELPER	DESCRIPTION
Chart	Renders a chart within a view. Contains methods such as Chart.ToWebImage, Chart.Save, and Chart.Write.
Crypto	Uses hashing algorithms to create properly salted and hashed passwords.
WebGrid	Renders a collection of objects (typically, data from a database) as a grid. Supports paging and sorting.
WebImage	Renders an image.
WebMail	Sends an email message.

A quick reference topic that lists the helpers and basic syntax is available as part of the ASP.NET Razor syntax documentation at the following URL:

https://www.asp.net/webmatrix/tutorials/asp-net-web-pages-api-reference

Additional Dependency Injection Support

Building on the ASP.NET MVC 3 Preview 1 release, the current release includes added support for two new services and four existing services, and improved support for dependency resolution and the Common Service Locator.

New IControllerActivator Interface for Fine-Grained Controller Instantiation

The new IControllerActivator interface provides more fine-grained control over how controllers are instantiated via dependency injection. The following example shows the interface:

```
namespace System.Web.Mvc {
    using System.Web.Routing;

    public interface IControllerActivator {
        IController Create(RequestContext requestContext, Type controllerType);
    }
}
```

Contrast this to the role of the controller factory. A controller factory is an implementation of the IControllerFactory interface, which is responsible both for locating a controller type and for instantiating an instance of that controller type.

Controller activators are responsible only for instantiating an instance of a controller type. They do not perform the controller type lookup. After locating a proper controller type, controller factories should delegate to an instance of IControllerActivator to handle the actual instantiation of the controller.

The DefaultControllerFactory class has a new constructor that accepts an IControllerFactory instance. This lets you apply Dependency Injection to manage this aspect of controller creation without having to override the default controller-type lookup behavior.

IServiceLocator Interface Replaced with IDependencyResolver

Based on community feedback, the ASP.NET MVC 3 Beta release has replaced the use of the IServiceLocator interface with a slimmed-down IDependencyResolver interface specific to the needs of ASP.NET MVC. The following example shows the new interface:

```
namespace System.Web.Mvc {
    using System.Collections.Generic;

    public interface IDependencyResolver {
        object GetService(Type serviceType);
        IEnumerable<object> GetServices(Type serviceType);
    }
}
```

As part of this change, the ServiceLocator class was also replaced with the DependencyResolver class. Registration of a dependency resolver is similar to earlier versions of ASP.NET MVC:

```
DependencyResolver.SetResolver(myResolver);
```

Implementations of this interface should simply delegate to the underlying dependency injection container to provide the registered service for the requested type.

When there are no registered services of the requested type, ASP.NET MVC expects implementations of this interface to return null from GetService and to return an empty collection from GetServices.

The new DependencyResolver class lets you register classes that implement either the new IDependencyResolver interface or the Common Service Locator interface (IServiceLocator). For more information about Common Service Locator, see CommonServiceLocator on GitHub.

New IViewActivator Interface for Fine-Grained View Page Instantiation

The new IViewPageActivator interface provides more fine-grained control over how view pages are instantiated via dependency injection. This applies to both WebFormView instances and RazorView instances. The following example shows the new interface:

```
namespace System.Web.Mvc {
    public interface IViewPageActivator {
        object Create(ControllerContext controllerContext, Type type);
    }
}
```

These classes now accept an IViewPageActivator constructor argument, which lets you use dependency injection to control how the ViewPage, ViewUserControl, and WebViewPage types are instantiated.

New Dependency Resolver Support for Existing Services

The new release includes dependency resolution support for the following services:

- Model validation providers. Classes that implement ModelValidatorProvider can be registered in the dependency resolver, and the system will use them to support client- and server-side validation.
- Model metadata provider. A single class that implements ModelMetadataProvider can be registered in the dependency resolver, and the system will use it to provide metadata for the templating and validation systems.
- Value providers. Classes that implement ValueProviderFactory can be registered in the dependency resolver, and the system will use them to create value providers that are consumed by the controller and during model binding.
- Model binders. Classes that implement IModelBinderProvider can be registered in the dependency resolver, and the system will use them to create model binders that are consumed by the model binding system.

New Support for Unobtrusive jQuery-Based Ajax

ASP.NET MVC includes Ajax helper methods such as the following:

- Ajax.ActionLink

- Ajax.RouteLink
- Ajax.BeginForm
- Ajax.BeginRouteForm

These methods use JavaScript to invoke an action method on the server rather than using a full postback. This functionality has been updated to take advantage of jQuery in an unobtrusive manner. Rather than intrusively emitting inline client scripts, these helper methods separate the behavior from the markup by emitting HTML5 attributes using the *data-ajax* prefix. Behavior is then applied to the markup by referencing the appropriate JavaScript files. Make sure that the following JavaScript files are referenced:

- jquery-1.4.1.js
- jquery.unobtrusive.ajax.js

This feature is enabled by default in the Web.config file in the ASP.NET MVC 3 new project templates, but is disabled by default for existing projects. For more information, see Added application-wide flags for client validation and unobtrusive JavaScript later in this document.

New Support for Unobtrusive jQuery Validation

By default, ASP.NET MVC 3 Beta uses jQuery validation in an unobtrusive manner in order to perform client-side validation. To enable unobtrusive client validation, make a call like the following from within a view:

```
Html.EnableClientValidation();
```

This requires that ViewContext.UnobtrusiveJavaScriptEnabled property is set to true, which you can do by making the following call:

```
Html.EnableUnobtrusiveJavaScript();
```

Also make sure the following JavaScript files are referenced.

- jquery-1.4.1.js
- jquery.validate.js
- jquery.validate.unobtrusive.js

This feature is enabled on by default in the Web.config file in the ASP.NET MVC 3 new project templates, but is disabled by default for existing projects. For more information, see New application-wide flags for client validation and unobtrusive JavaScript later in this document.

New Application-Wide Flags for Client Validation and Unobtrusive JavaScript

You can enable or disable client validation and unobtrusive JavaScript globally using static members of the HtmlHelper class, as in the following example:

```
HtmlHelper.ClientValidationEnabled = true;
HtmlHelper.UnobtrusiveJavaScriptEnabled = true;
```

The default project templates enable unobtrusive JavaScript by default. You can also enable or disable these features in the root Web.config file of your application using the following settings:

```
<configuration>
        <appSettings>
            <add key="ClientValidationEnabled" value="true"/>
            <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
        </appSettings>
    </configuration>
```

Because you can enable these features by default, new overloads were introduced to the HtmlHelper class that let you override the default settings, as shown in the following examples:

```
public void EnableClientValidation();
public void EnableClientValidation(bool enabled);
public void EnableUnobtrusiveJavaScript();
public void EnableUnobtrusiveJavaScript(bool enabled);
```

For backward compatibility, both of these features are disabled by default.

New Support for Code that Runs Before Views Run

You can now put a file named _viewstart.cshtml (or _viewstart.vbhtml) in the Views directory and add code to it that will be shared among multiple views in that directory and its subdirectories. For example, you might put the following code into the _viewstart.cshtml page in the ~/Views folder:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

This sets the layout page for every view within the Views folder and all its subfolders recursively. When a view is being rendered, the code in the _viewstart.cshtml file runs before the view code runs. The _viewstart.cshtml code applies to every view in that folder.

By default, the code in the _viewstart.cshtml file also applies to views in any subfolder. However, individual subfolders can have their own version of the _viewstart.cshtml file; in that case, the local version takes precedence. For example, to run code that is common to all views for the HomeController, put a _viewstart.cshtml file in the ~/Views/Home folder.

New Support for the VBHTML Razor Syntax

The previous ASP.NET MVC preview included support for views using Razor syntax based on C#. These views use the .cshtml file extension. As part of ongoing work to support Razor, the ASP.NET MVC 3 Beta introduces support for the Razor syntax in Visual Basic, which uses the .vbhtml file extension.

For an introduction to using Visual Basic syntax in VBHTML pages, see the tutorial at the following URL:

https://www.asp.net/webmatrix/tutorials/asp-net-web-pages-visual-basic

More Granular Control over ValidateInputAttribute

ASP.NET MVC has always included the ValidateInputAttribute class, which invokes the core ASP.NET request validation infrastructure to make sure that the incoming request does not contain potentially malicious input. By default, input validation is enabled. It is possible to disable request validation by using the ValidateInputAttribute attribute, as in the following example:

```
[ValidateInput(false)]
public ActionResult SomeAction() {
    return View();
}
```

However, many web applications have individual form fields that need to allow HTML, while the remaining fields should not. The ValidateInputAttribute class now lets you specify a list of fields that should not be included in request validation.

For example, if you are developing a blog engine, you might want to allow markup in the Body and Summary fields. These fields might be represented by two input element, each with a name attribute corresponding to the property name ("Body" and "Summary"). To disable request validation for these fields only, specify the names (comma-separated) in the Exclude property of the ValidateInput class, as in the following example:

```
[ValidateInput(true, Exclude="Body, Summary")]
public ActionResult About() {
    return View();
}
```

Helpers Convert Underscores to Hyphens for HTML Attribute Names Specified Using Anonymous Objects

Helper methods let you specify attribute name/value pairs using an anonymous object, as in the following example:

```
Html.TextBox("Name", "Value", new {title = "Title"})
```

This approach doesn't let you use hyphens in the attribute name, because a hyphen cannot be used for a property name in ASP.NET. However, hyphens are important for custom HTML5 attributes; for example, HTML5 uses the "data-" prefix.

At the same time, underscores cannot be used for attribute names in HTML, but are valid within property names. Therefore, if you specify attributes using an anonymous object, and if the attribute names include an underscore,, helper methods will convert the underscores to hyphens. For example, the following helper syntax uses an underscore:

```
Html.TextBox("Name", "Value", new {data_required = "true"})
```

The previous example renders the following markup when the helper runs:

```
<input data-required="true" id="Name" name="Name"
       type="textbox" value="Value" />
```

Bug Fixes

The default object template for the EditorFor and DisplayFor template helpers now supports the ordering specified in the DisplayAttribute.Order property. (In previous versions, the Order setting was not used.)

Client validation now supports validation of overridden properties that have validation attributes applied.

JsonValueProviderFactory is now registered by default.

Breaking Changes

The order of execution for exception filters has changed for exception filters that have the same Order value. In ASP.NET MVC 2 and earlier, exception filters on the controller with the same Order as those on an action method were executed before the exception filters on the action method. This would typically be the case when exception filters were applied without a specified Order value. In ASP.NET MVC 3, this order has been reversed so that the most specific exception handler executes first. As in earlier versions, if the Order property is explicitly specified, the filters are run in the specified order.

Known Issues

During installation, the EULA acceptance dialog box displays the license terms in a window that is smaller than intended.

Razor views do not have IntelliSense support nor syntax highlighting. It is anticipated that support for Razor syntax in Visual Studio will be included as part of a later release.

When you are editing a Razor view (CSHTML file), the Go To Controller menu item in Visual Studio will not be available, and there are no code snippets.

When using the @model syntax to specify a strongly typed CSHTML view, language-specific shortcuts for types are not recognized. For example, @model int will not work, but @model Int32 will work. The workaround for this bug is to use the actual type name when you specify the model type.

When using the @model syntax to specify a strongly typed CSHTML view (or @ModelType to specify a strongly typed VBHTML view), nullable types and array declarations are not supported. For example, @model int? is not supported. Instead, use `@model Nullable<Int32>`. The syntax @model string[] is also not supported; instead, use `@model IList<string>`.

When you upgrade an ASP.NET MVC 2 project to ASP.NET MVC 3, make sure to add the following to the appSettings section of the Web.config file:

```
<appSettings>
  <add key="enableSimpleMembership" value="false" />
</appSettings>
```

There's a known issue that causes Forms Authentication to always redirect unauthenticated users to ~/Account/Login, ignoring the forms authentication setting used in Web.config. The workaround is to add the following app setting.

```
<add key="autoFormsAuthentication" value="false" />
```

Disclaimer

© 2011 Microsoft Corporation. All rights reserved. This document is provided "as-is." Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

ASP.NET MVC 4

2/20/2020 • 16 minutes to read • Edit Online

This document describes the release of ASP.NET MVC 4 Beta for Visual Studio 2010.

> **NOTE**
> This is not the most current release. The ASP.NET MVC 4 RC release notes are available here.

- Installation Notes
- Documentation
- Support
- Software Requirements
- Upgrading an ASP.NET MVC 3 Project to ASP.NET MVC 4
- New Features in ASP.NET MVC 4 Beta
  - ASP.NET Web API
  - ASP.NET Single Page Application
  - Enhancements to Default Project Templates
  - Mobile Project Template
  - Display Modes
  - jQuery Mobile, the View Switcher, and Browser Overriding
  - Recipes for Code Generation in Visual Studio
  - Task Support for Asynchronous Controllers
  - Azure SDK
  - Known Issues and Breaking Changes

Installation Notes

ASP.NET MVC 4 Beta for Visual Studio 2010 can be installed from the ASP.NET MVC 4 home page using the Web Platform Installer.

You must uninstall any previously installed previews of ASP.NET MVC 4 prior to installing ASP.NET MVC 4 Beta.

This release is not compatible with the .NET Framework 4.5 Developer Preview. You must uninstall the .NET 4.5 Developer Preview before installing the ASP.NET MVC 4 Beta.

ASP.NET MVC 4 can be installed and can run side-by-side with ASP.NET MVC 3.

Documentation

Documentation for ASP.NET MVC is available on the MSDN website at the following URL:

https://go.microsoft.com/fwlink/?LinkID=243043

Tutorials and other information about ASP.NET MVC are available on the MVC 4 page of the ASP.NET website (https://www.asp.net/mvc/mvc4).

Support

This is a preview release and is not officially supported. If you have questions about working with this release, post them to the ASP.NET MVC forum (https://forums.asp.net/1146.aspx), where members of the ASP.NET community are frequently able to provide informal support.

Software Requirements

The ASP.NET MVC 4 components for Visual Studio require PowerShell 2.0 and either Visual Studio 2010 with Service Pack 1 or Visual Web Developer Express 2010 with Service Pack 1.

Upgrading an ASP.NET MVC 3 Project to ASP.NET MVC 4

ASP.NET MVC 4 can be installed side by side with ASP.NET MVC 3 on the same computer, which gives you flexibility in choosing when to upgrade an ASP.NET MVC 3 application to ASP.NET MVC 4.

The simplest way to upgrade is to create a new ASP.NET MVC 4 project and copy all the views, controllers, code, and content files from the existing MVC 3 project to the new project and then to update the assembly references in the new project to match the old project. If you have made changes to the Web.config file in the MVC 3 project, you must also merge those changes into the Web.config file in the MVC 4 project.

To manually upgrade an existing ASP.NET MVC 3 application to version 4, do the following:

1. In all Web.config files in the project (there is one in the root of the project, one in the Views folder, and one in the Views folder for each area in your project), replace every instance of the following text:

   ```
   System.Web.Mvc, Version=3.0.0.0
   System.Web.WebPages, Version=1.0.0.0
   System.Web.Helpers, Version=1.0.0.0
   System.Web.WebPages.Razor, Version=1.0.0.0
   ```

   with the following corresponding text:

   ```
   System.Web.Mvc, Version=4.0.0.0
   System.Web.WebPages, Version=2.0.0.0
   System.Web.Helpers, Version=2.0.0.0,
    System.Web.WebPages.Razor, Version=2.0.0.0,
   ```

2. In the root Web.config file, update the *webPages:Version* element to "2.0.0.0" and add a new *PreserveLoginUrl* key that has the value "true":

   ```
   <appSettings>
     <add key="webpages:Version" value="2.0.0.0" />
     <add key="PreserveLoginUrl" value="true" />
   </appSettings>
   ```

3. In Solution Explorer, delete the reference to *System.Web.Mvc* (which points to the version 3 DLL). Then add a reference to *System.Web.Mvc* (v4.0.0.0). In particular, make the following changes to update the assembly references. Here are the details:

   a. In Solution Explorer, delete the references to the following assemblies:

      - *System.Web.Mvc*(v3.0.0.0)
      - *System.Web.WebPages*(v1.0.0.0)
      - *System.Web.Razor*(v1.0.0.0)

- *System.Web.WebPages.Deployment*(v1.0.0.0)
- *System.Web.WebPages.Razor*(v1.0.0.0)

b. Add a references to the following assemblies:

- *System.Web.Mvc*(v4.0.0.0)
- *System.Web.WebPages*(v2.0.0.0)
- *System.Web.Razor*(v2.0.0.0)
- *System.Web.WebPages.Deployment*(v2.0.0.0)
- *System.Web.WebPages.Razor*(v2.0.0.0)

4. In Solution Explorer, right-click the project name and then select Unload Project. Then right-click the name again and select Edit *ProjectName*.csproj.

5. Locate the *ProjectTypeGuids* element and replace {E53F8FEA-EAE0-44A6-8774-FFD645390401} with {E3E379DF-F4C6-4180-9B81-6769533ABE47}.

6. Save the changes, close the project (.csproj) file you were editing, right-click the project, and then select Reload Project.

7. If the project references any third-party libraries that are compiled using previous versions of ASP.NET MVC, open the root Web.config file and add the following three *bindingRedirect* elements under the *configuration* section:

```
<configuration>
  <!--... elements deleted for clarity ...-->

  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="System.Web.Helpers"
             publicKeyToken="31bf3856ad364e35" />
        <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="System.Web.Mvc"
             publicKeyToken="31bf3856ad364e35" />
        <bindingRedirect oldVersion="1.0.0.0-3.0.0.0" newVersion="4.0.0.0"/>
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="System.Web.WebPages"
             publicKeyToken="31bf3856ad364e35" />
        <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

New Features in ASP.NET MVC 4 Beta

This section describes features that have been introduced in the ASP.NET MVC 4 Beta release.

ASP.NET Web API

ASP.NET MVC 4 now includes ASP.NET Web API, a new framework for creating HTTP services that can reach a broad range of clients including browsers and mobile devices. ASP.NET Web API is also an ideal platform for building RESTful services.

ASP.NET Web API includes support for the following features:

- **Modern HTTP programming model**: Directly access and manipulate HTTP requests and responses in your

Web APIs using a new, strongly typed HTTP object model. The same programming model and HTTP pipeline is symmetrically available on the client through the new HttpClient type.

- **Full support for routes**: Web APIs now support the full set of route capabilities that have always been a part of the Web stack, including route parameters and constraints. Additionally, mapping to actions has full support for conventions, so you no longer need to apply attributes such as [HttpPost] to your classes and methods.
- **Content negotiation**: The client and server can work together to determine the right format for data being returned from an API. We provide default support for XML, JSON, and Form URL-encoded formats, and you can extend this support by adding your own formatters, or even replace the default content negotiation strategy.
- **Model binding and validation:** Model binders provide an easy way to extract data from various parts of an HTTP request and convert those message parts into .NET objects which can be used by the Web API actions.
- **Filters:** Web APIs now supports filters, including well-known filters such as the [Authorize] attribute. You can author and plug in your own filters for actions, authorization and exception handling.
- **Query composition:** By simply returning IQueryable<T>, your Web API will support querying via the OData URL conventions.
- **Improved testability of HTTP details:** Rather than setting HTTP details in static context objects, Web API actions can now work with instances of HttpRequestMessage and HttpResponseMessage. Generic versions of these objects also exist to let you work with your custom types in addition to the HTTP types.
- **Improved Inversion of Control (IoC) via DependencyResolver:** Web API now uses the service locator pattern implemented by MVC's dependency resolver to obtain instances for many different facilities.
- **Code-based configuration:** Web API configuration is accomplished solely through code, leaving your config files clean.
- **Self-host:** Web APIs can be hosted in your own process in addition to IIS while still using the full power of routes and other features of Web API.

For more details on ASP.NET Web API please visit https://www.asp.net/web-api.

ASP.NET Single Page Application

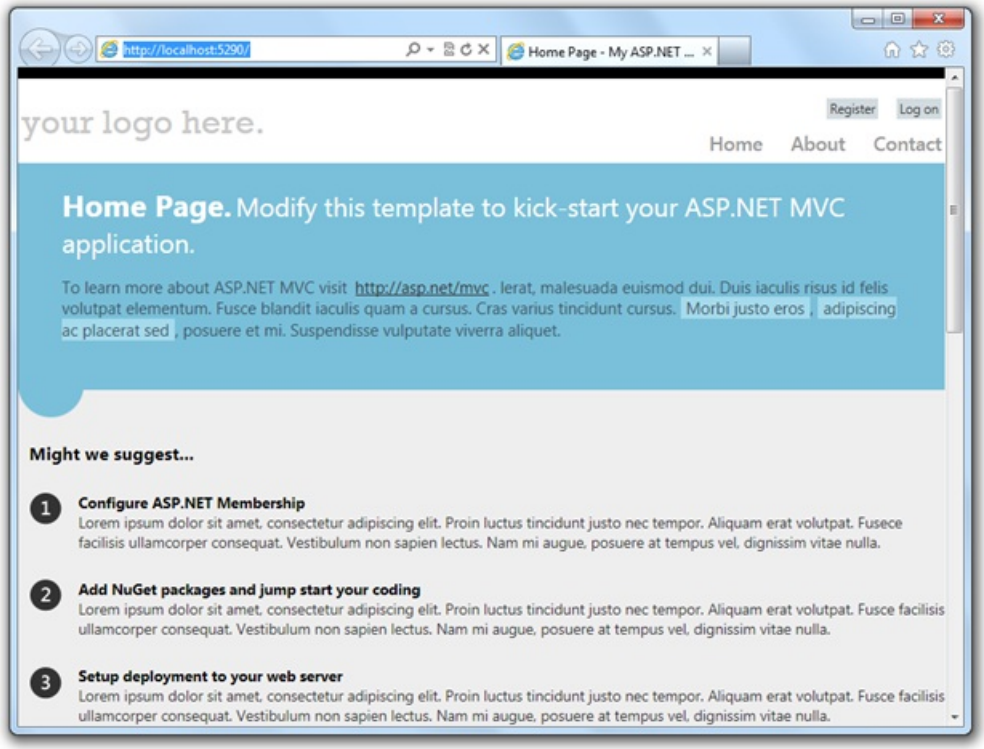
ASP.NET MVC 4 now includes an early preview of the experience for building single page applications with significant client-side interactions using JavaScript and Web APIs. This support includes:

- A set of JavaScript libraries for richer local interactions with cached data
- Additional Web API components for unit of work and DAL support
- An MVC project template with scaffolding to get started quickly

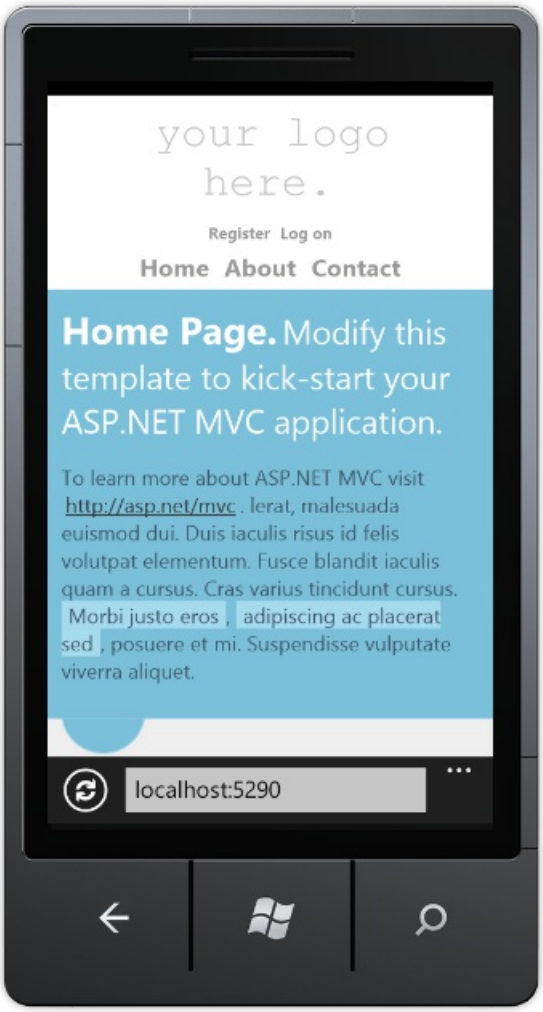
For more details on the Single Page Application support in ASP.NET MVC 4 please visit https://www.asp.net/single-page-application.

Enhancements to Default Project Templates

The template that is used to create new ASP.NET MVC 4 projects has been updated to create a more modern-looking website:



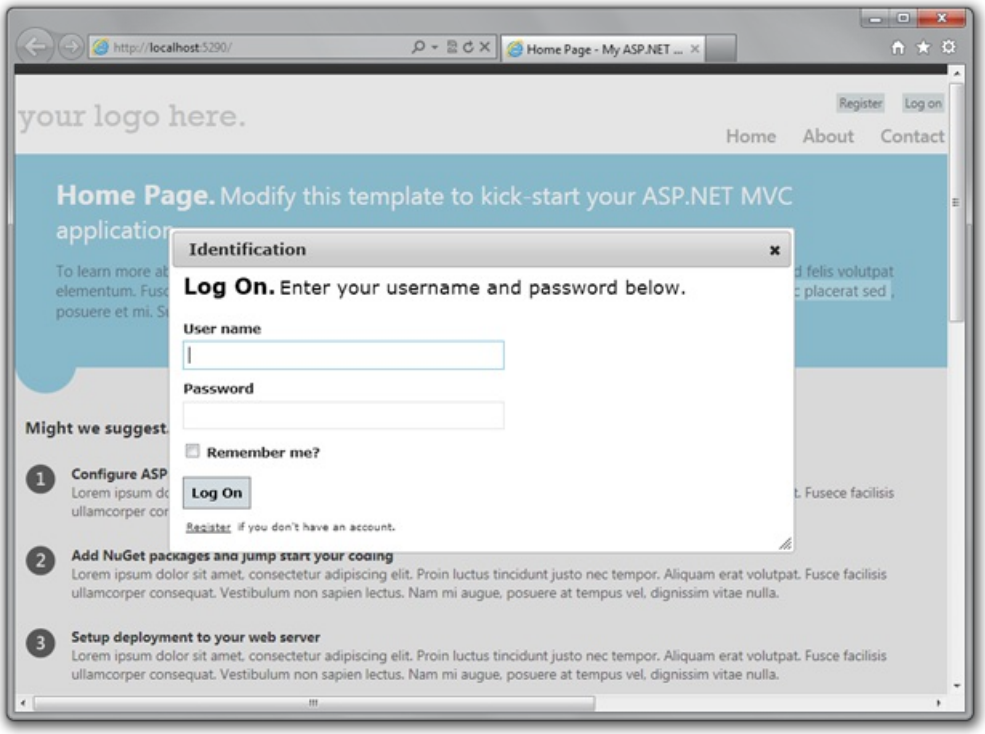
In addition to cosmetic improvements, there's improved functionality in the new template. The template employs a technique called adaptive rendering to look good in both desktop browsers and mobile browsers without any customization.



To see adaptive rendering in action, you can use a mobile emulator or just try resizing the desktop browser

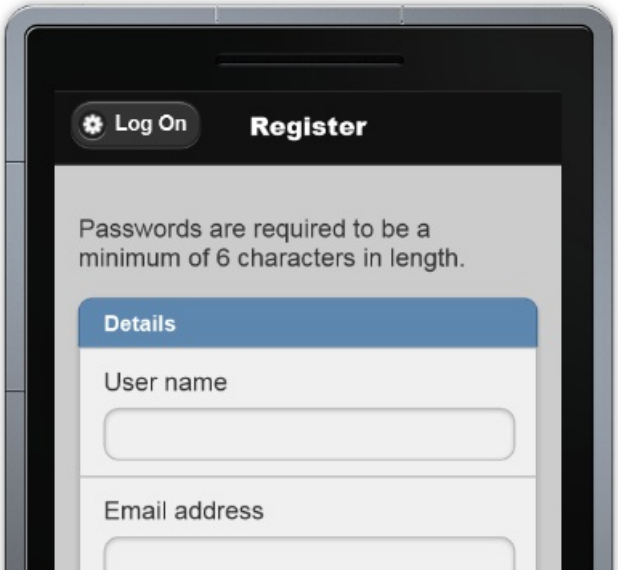
window to be smaller. When the browser window gets small enough, the layout of the page will change.

Another enhancement to the default project template is the use of JavaScript to provide a richer UI. The Login and Register links that are used in the template are examples of how to use the jQuery UI Dialog to present a rich login screen:



Mobile Project Template

If you're starting a new project and want to create a site specifically for mobile and tablet browsers, you can use the new Mobile Application project template. This is based on jQuery Mobile, an open-source library for building touch-optimized UI:



This template contains the same application structure as the Internet Application template (and the controller code is virtually identical), but it's styled using jQuery Mobile to look good and behave well on touch-based mobile devices. To learn more about how to structure and style mobile UI, see the jQuery Mobile project website.

If you already have a desktop-oriented site that you want to add mobile-optimized views to, or if you want to create a single site that serves differently styled views to desktop and mobile browsers, you can use the new Display Modes feature. (See the next section.)

Display Modes

The new Display Modes feature lets an application select views depending on the browser that's making the request. For example, if a desktop browser requests the Home page, the application might use the Views\Home\Index.cshtml template. If a mobile browser requests the Home page, the application might return the Views\Home\Index.mobile.cshtml template.

Layouts and partials can also be overridden for particular browser types. For example:

- If your Views\Shared folder contains both the _Layout.cshtml and _Layout.mobile.cshtml templates, by default the application will use _Layout.mobile.cshtml during requests from mobile browsers and _Layout.cshtml during other requests.
- If a folder contains both _MyPartial.cshtml and _MyPartial.mobile.cshtml, the instruction @Html.Partial("_MyPartial") will render _MyPartial.mobile.cshtml during requests from mobile browsers, and _MyPartial.cshtml during other requests.

If you want to create more specific views, layouts, or partial views for other devices, you can register a new *DefaultDisplayMode* instance to specify which name to search for when a request satisfies particular conditions. For example, you could add the following code to the *Application_Start* method in the Global.asax file to register the string "iPhone" as a display mode that applies when the Apple iPhone browser makes a request:

```
DisplayModeProvider.Instance.Modes.Insert(0, new
DefaultDisplayMode("iPhone")
{
    ContextCondition = (context => context.GetOverriddenUserAgent().IndexOf
        ("iPhone", StringComparison.OrdinalIgnoreCase) >= 0)
 });
```

After this code runs, when an Apple iPhone browser makes a request, your application will use the Views\Shared_Layout.iPhone.cshtml layout (if it exists).

jQuery Mobile, the View Switcher, and Browser Overriding

jQuery Mobile is an open source library for building touch-optimized web UI. If you want to use jQuery Mobile with an ASP.NET MVC 4 application, you can download and install a NuGet package that helps you get started. To install it from the Visual Studio Package Manager Console, type the following command:

```
Install-Package jQuery.Mobile.MVC
```

This installs jQuery Mobile and some helper files, including the following:

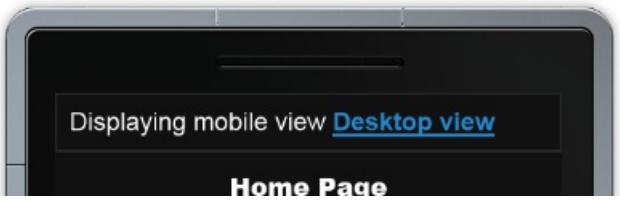
- Views/Shared/_Layout.Mobile.cshtml, which is a jQuery Mobile-based layout.
- A view-switcher component, which consists of the Views/Shared/_ViewSwitcher.cshtml partial view and the ViewSwitcherController.cs controller.

After you install the package, run your application using a mobile browser (or equivalent, like the Firefox User Agent Switcher add-on). You'll see that your pages look quite different, because jQuery Mobile handles layout and styling. To take advantage of this, you can do the following:

- Create mobile-specific view overrides as described under Display Modes earlier (for example, create Views\Home\Index.mobile.cshtml to override Views\Home\Index.cshtml for mobile browsers).
- Read the jQuery Mobile documentation to learn more about how to add touch-optimized UI elements in mobile views.

A convention for mobile-optimized web pages is to add a link whose text is something like Desktop view or Full site mode that lets users switch to a desktop version of the page. The jQuery.Mobile.MVC package includes a sample view-switcher component for this purpose. It's used in the default Views\Shared_Layout.Mobile.cshtml

view, and it looks like this when the page is rendered:



If visitors click the link, they're switched to the desktop version of the same page.

Because your desktop layout will not include a view switcher by default, visitors won't have a way to get to mobile mode. To enable this, add the following reference to _*ViewSwitcher* to your desktop layout, just inside the *<body>* element:

```
<body>
    @Html.Partial("_ViewSwitcher")
    ...
```

The view switcher uses a new feature called Browser Overriding. This feature lets your application treat requests as if they were coming from a different browser (user agent) than the one they're actually from. The following table lists the methods that Browser Overriding provides.

`HTTPCONTEXT.SETOVERRIDDENBROWSER(USERAGENTSTRING)`	**OVERRIDES THE REQUEST'S ACTUAL USER AGENT VALUE USING THE SPECIFIED USER AGENT.**
`HttpContext.GetOverriddenUserAgent()`	Returns the request's user agent override value, or the actual user agent string if no override has been specified.
`HttpContext.GetOverriddenBrowser()`	Returns an *HttpBrowserCapabilitiesBase* instance that corresponds to the user agent currently set for the request (actual or overridden). You can use this value to get properties such as *IsMobileDevice*.
`HttpContext.ClearOverriddenBrowser()`	Removes any overridden user agent for the current request.

Browser Overriding is a core feature of ASP.NET MVC 4 and is available even if you don't install the jQuery.Mobile.MVC package. However, it affects only view, layout, and partial-view selection — it does not affect any other ASP.NET feature that depends on the *Request.Browser* object.

By default, the user-agent override is stored using a cookie. If you want to store the override elsewhere (for example, in a database), you can replace the default provider (*BrowserOverrideStores.Current*). Documentation for this provider will be available to accompany a later release of ASP.NET MVC.

Recipes for Code Generation in Visual Studio

The new Recipes feature enables Visual Studio to generate solution-specific code based on packages that you can install using NuGet. The Recipes framework makes it easy for developers to write code-generation plugins, which you can also use to replace the built-in code generators for Add Area, Add Controller, and Add View. Because recipes are deployed as NuGet packages, they can easily be checked into source control and shared with all developers on the project automatically. They are also available on a per-solution basis.

Task Support for Asynchronous Controllers

You can now write asynchronous action methods as single methods that return an object of type *Task* or *Task<ActionResult>*.

For example, if you're using Visual C# 5 (or using the Async CTP), you can create an asynchronous action method that looks like the following:

```
public async Task<ActionResult> Index(string city) {
    var newsService = new NewsService();
    var sportsService = new SportsService();

    return View("Common",
        new PortalViewModel {
        NewsHeadlines = await newsService.GetHeadlinesAsync(),
        SportsScores = await sportsService.GetScoresAsync()
    });
}
```

In the previous action method, the calls to *newsService.GetHeadlinesAsync* and *sportsService.GetScoresAsync* are called asynchronously and do not block a thread from the thread pool.

Asynchronous action methods that return *Task* instances can also support timeouts. To make your action method cancellable, add a parameter of type *CancellationToken* to the action method signature. The following example shows an asynchronous action method that has a timeout of 2500 milliseconds and that displays a *TimedOut* view to the client if a timeout occurs.

```
[AsyncTimeout(2500)]
[HandleError(ExceptionType = typeof(TaskCanceledException), View = "TimedOut")]
public async Task<ActionResult> Index(string city,
    CancellationToken cancellationToken) {
    var newsService = new NewsService();
    var sportsService = new SportsService();

    return View("Common",
        new PortalViewModel {
        NewsHeadlines = await newsService.GetHeadlinesAsync(cancellationToken),
        SportsScores = await sportsService.GetScoresAsync(cancellationToken)
    });
}
```

Azure SDK

ASP.NET MVC 4 Beta supports the September 2011 1.5 release of the Windows Azure SDK.

Known Issues and Breaking Changes

- **After installing ASP.NET MVC 4 Beta, the CSHTML/VBHTML editor in Visual Studio 2010 Service Pack 1 CSHTML/VBHTML editor may pause for a long time after typing snippet or JavaScript inside cshtml or vbhtml files.** This occurs only in ASP.NET MVC 4 applications which have just been created and have not yet been compiled.

  The workaround is to compile the project to get the assemblies in the bin folder. Note, if you clean the project which removes the assemblies from the bin folder, the editor problem will come back.

  This will be corrected in the next release.

- **C# Project templates for Visual Studio 11 Beta contain an incorrect connection string in Global.asax.cs.** The default connection specified in the Application_Start method for projects created in Visual Studio 11 Beta contain a LocalDB connection string which contains an unescaped backslash () character. This results in a connection error upon attempts to access a Entity Framework DbContext, which generates a SqlException.

  To correct this issue, escape the backslash character in the App_Start method of Global.asax.cs so that it reads as follows:

```
Database.DefaultConnectionFactory =
   new SqlConnectionFactory(
     "Data Source=(localdb)\\v11.0; Integrated Security=True; MultipleActiveResultSets=True");
```

- **ASP.NET MVC 4 applications which target .NET 4.5 will throw a FileLoadException upon attempt to access the System.Net.Http.dll assembly when run under .NET 4.0.** ASP.NET MVC 4 applications created under .NET 4.5 contain a binding redirect that will result in a FileLoadException which that states "Could not load file or assembly 'System.Net.Http' or one of its dependencies." when the application is executed on a system with .NET 4.0 installed. To correct this issue, remove the following binding redirect from web.config:

```
<dependentAssembly>
 <assemblyIdentity name="System.Net.Http" publicKeyToken="b03f5f7f11d50a3a"
    culture="neutral" />
 <bindingRedirect oldVersion="0.0.0.0-4.0.0.0" newVersion="4.0.0.0" />
</dependentAssembly>
```

  The assembly binding element in the modified web.config should appear as follows:

```
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
 <dependentAssembly>
  <assemblyIdentity name="System.Web.Helpers"
    publicKeyToken="31bf3856ad364e35" />
  <bindingRedirect oldVersion="1.0.0.0-2.0.0.0" newVersion="2.0.0.0" />
 </dependentAssembly>
 <dependentAssembly>
  <assemblyIdentity name="System.Web.Mvc"
    publicKeyToken="31bf3856ad364e35" />
  <bindingRedirect oldVersion="1.0.0.0-4.0.0.0" newVersion="4.0.0.0" />
 </dependentAssembly>
 <dependentAssembly>
  <assemblyIdentity name="System.Web.WebPages"
    publicKeyToken="31bf3856ad364e35" />
  <bindingRedirect oldVersion="1.0.0.0-2.0.0.0" newVersion="2.0.0.0" />
 </dependentAssembly>
</assemblyBinding>
```

- **The "Add Controller" item template in Visual Basic projects generates an incorrect namespace when invokedfrom inside an area.** When you add a controller to an area in an ASP.NET MVC project that uses Visual Basic, the item template inserts the wrong namespace into the controller. The result is a "file not found" error when you navigate to any action in the controller.

  The generated namespace omits everything after the root namespace. For example, the namespace generated is *RootNamespace* but should be *RootNamespace.Areas.AreaName.Controllers* .

- **Breaking changes in the Razor View Engine.** As part of a rewrite of the Razor parser, the following types were removed from *System.Web.Mvc.Razor*:

  - *ModelSpan*
  - *MvcVBRazorCodeGenerator*
  - *MvcCSharpRazorCodeGenerator*
  - *MvcVBRazorCodeParser*

  The following methods were also removed:

  - *MvcCSharpRazorCodeParser.ParseInheritsStatement(System.Web.Razor.Parser.CodeBlockInfo)*
  - *MvcWebPageRazorHost.DecorateCodeGenerator(System.Web.Razor.Generator.RazorCodeGenerator)*
  - *MvcVBRazorCodeParser.ParseInheritsStatement(System.Web.Razor.Parser.CodeBlockInfo)*

- **When WebMatrix.WebData.dll is included in the /bin directory of an ASP.NET MVC 4 apps, it takes over the URL for forms authentication.** Adding the WebMatrix.WebData.dll assembly to your application (for example, by selecting "ASP.NET Web Pages with Razor Syntax" when using the Add Deployable Dependencies dialog) will override the authentication login redirect to /account/logon rather than /account/login as expected by the default ASP.NET MVC Account Controller. To prevent this behavior and use the URL specified already in the authentication section of web.config, you can add an appSetting called PreserveLoginUrl and set it to true:

```
<appSettings>
    <add key="PreserveLoginUrl" value="true"/>
</appSettings>
```

- **The NuGet package manager fails to install when attempting to install ASP.NET MVC 4 for side by side installations of Visual Studio 2010 and Visual Web Developer 2010.** To run Visual Studio 2010 and Visual Web Developer 2010 side by side with ASP.NET MVC 4 you must install ASP.NET MVC 4 after both versions of Visual Studio have already been installed.

- **Uninstalling ASP.NET MVC 4 fails if prerequisites have already been uninstalled.** To cleanly uninstall ASP.NET MVC 4you must uninstall ASP.NET MVC 4 prior to uninstalling Visual Studio.

- **Running a default Web API project shows instructions that incorrectly direct the user to add routes using the RegisterApis method, which doesn't exist.** Routes should be added in the RegisterRoutes method using the ASP.NET route table.

- **Installing ASP.NET MVC 4 Beta breaks ASP.NET MVC 3 RTM applications.** ASP.NET MVC 3 applications that were created with the RTM release (not with the ASP.NET MVC 3 Tools Update release) require the following changes in order to work side-by-side with ASP.NET MVC 4 Beta. Building the project without making these updates results in compilation errors.

  Required updates

  1. In the root Web.config file, add a new *<appSettings>* entry with the key *webPages:Version* and the value *1.0.0.0*.

     ```
     <appSettings>
         <add key="webpages:Version" value="1.0.0.0"/>
         <add key="ClientValidationEnabled" value="true"/>
         <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
     </appSettings>
     ```

  2. In Solution Explorer, right-click the project name and then select Unload Project. Then right-click the name again and select Edit *ProjectName*.csproj.

  3. Locate the following assembly references:

     ```
     <Reference Include="System.Web.WebPages"/>
     <Reference Include="System.Web.Helpers" />
     ```

     Replace them with the following:

     ```
     <Reference Include="System.Web.WebPages, Version=1.0.0.0,
     Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL "/>
     <Reference Include="System.Web.Helpers, Version=1.0.0.0,
     Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL" />
     ```

4. Save the changes, close the project (.csproj) file you were editing, and then right-click the project and select Reload.

Running ASP.NET 1.1 with IIS 6.0

2/20/2020 • 7 minutes to read • Edit Online

While Windows Server 2003 includes both IIS 6.0 and ASP.NET 1.1, these components are disabled by default. This whitepaper describes how to enable IIS 6.0 and ASP.NET 1.1, and recommends several configuration settings to get the optimal performance from IIS and ASP.NET.

Applies to ASP.NET 1.1 and IIS 6.0.

ASP.NET 1.1 ships with Windows Server 2003, which also includes the latest version of Internet Information Server (IIS) version 6.0. IIS 6.0 and ASP.NET 1.1 are designed to integrate seamlessly and ASP.NET now defaults to the new IIS 6.0 worker process model.

ASP.NET 1.1 is not installed by default

Unlike previous versions of Microsoft's server operating systems, Internet Information Server (IIS) is not enabled by default; nor is ASP.NET 1.1. There are two options for enabling IIS:

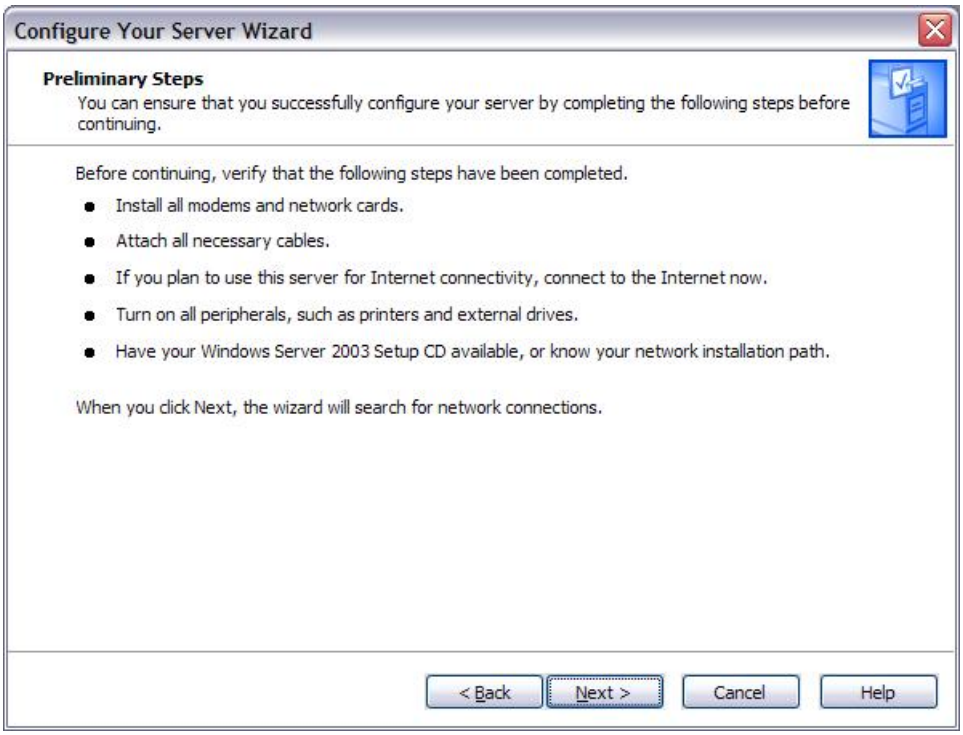
Enabling IIS, option #1 - Configure Your Server Wizard

Windows Server 2003 ships a new 'Configure Your Server Wizard' to help you properly configure your server in the desired mode.

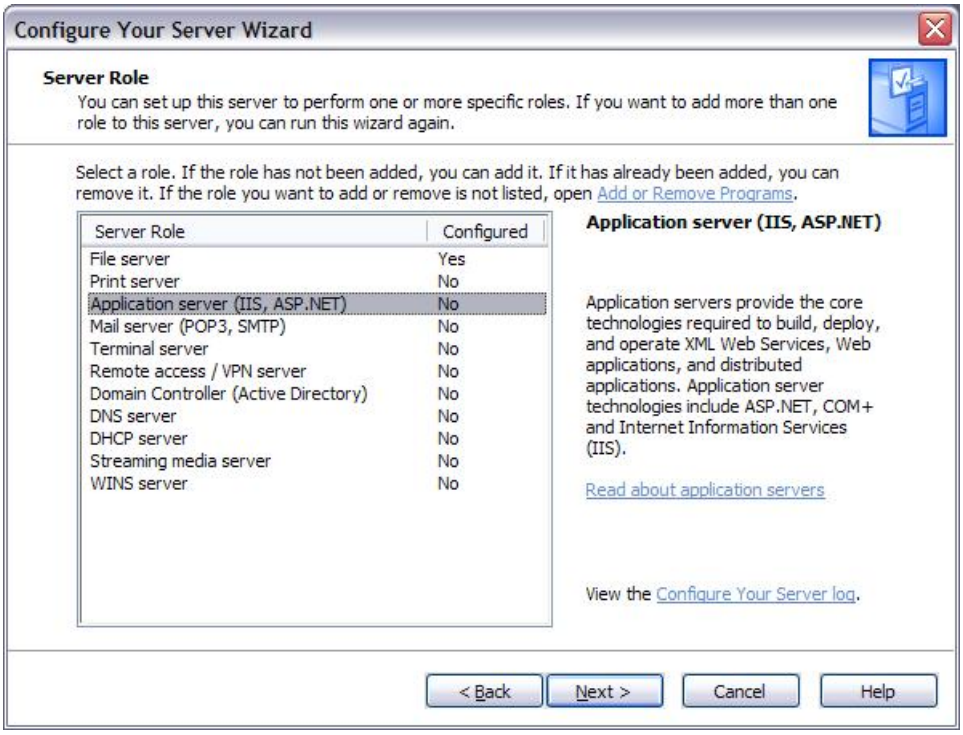
To start the wizard - note, to run the wizard you must be logged in as an administrator - go to: Start | Programs | Administrative Tools and select 'Configure Your Server'.

Once selected you should see the 'Configure Your Server Wizard' opening screen:

Click 'Next >':



Click 'Next >'

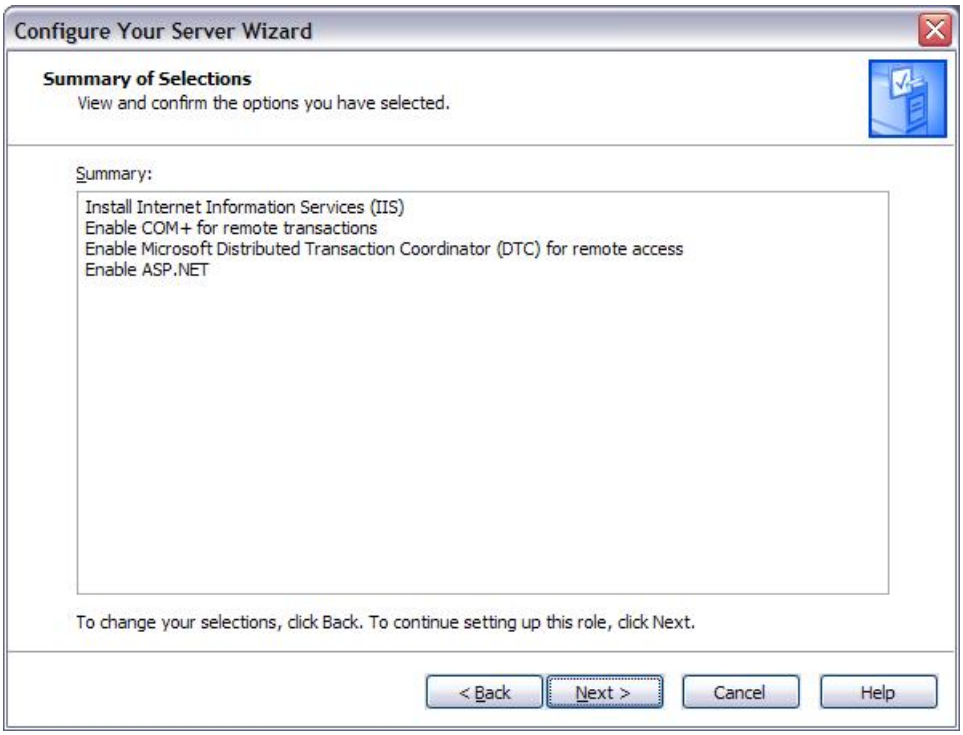


On this screen you will need to select 'Application server (IIS, ASP.NET) as the options to configure.

Click 'Next >'.

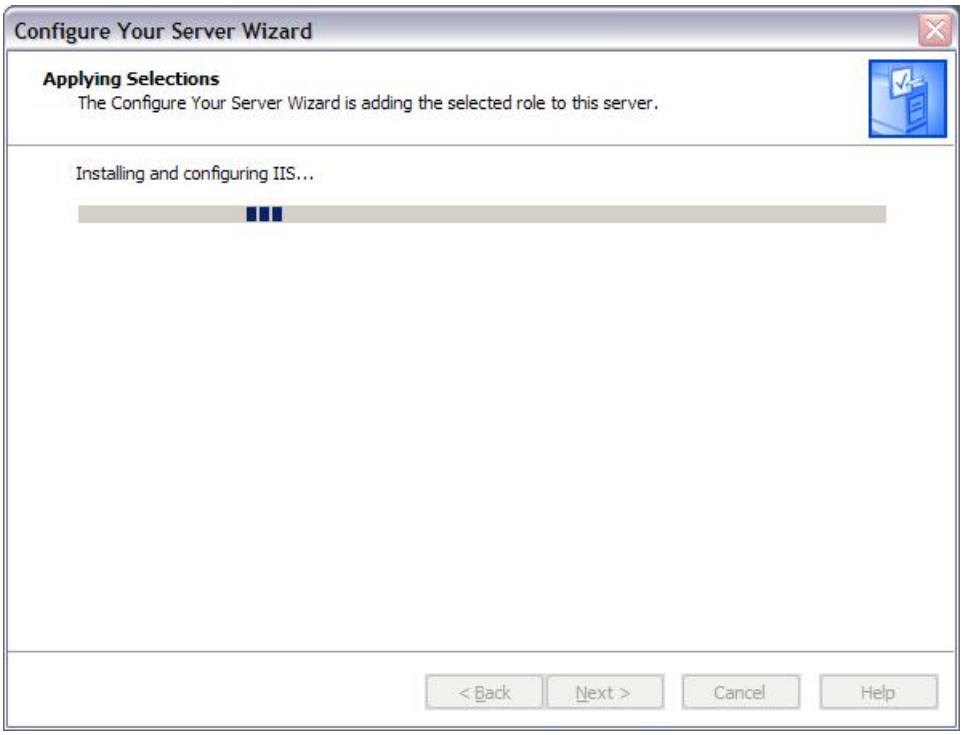
After selecting to configure the server as an Application Server, this screen will be displayed prompting what additional capabilities should be installed. Neither option is selected by default. To enable ASP.NET automatically, you need to select 'Enable ASP.NET'.

Click 'Next >'.



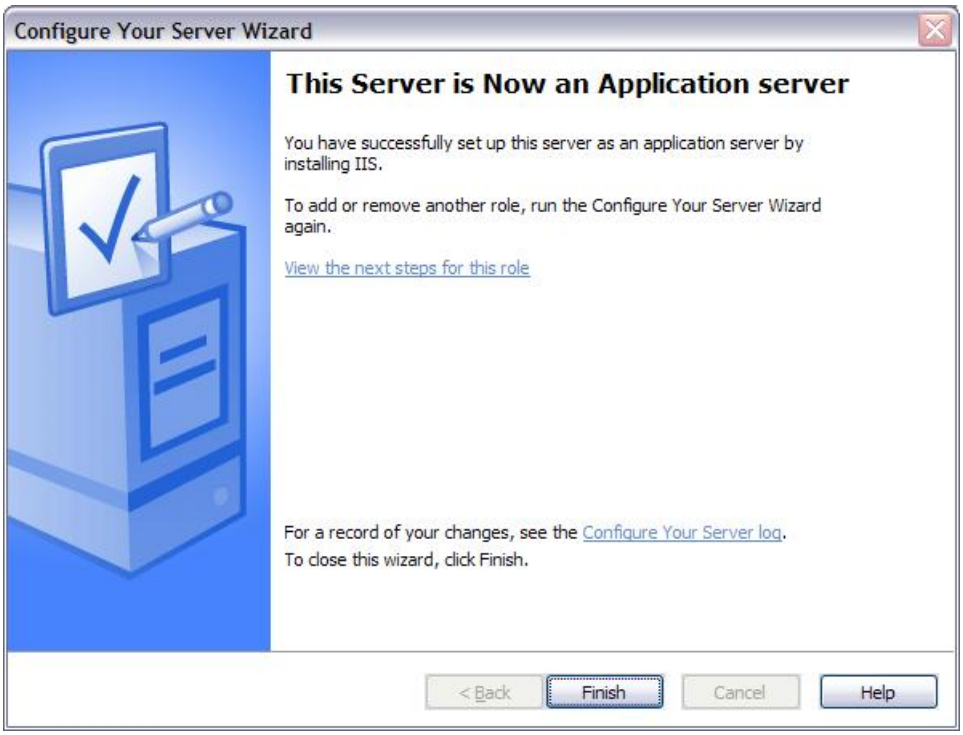
This screen displays what options are to be installed.

Click 'Next >'.



You will see this screen while the options you selected are being installed. It is normal to see other dialog boxes appear as services are being installed. You may additionally be prompted for the location of the Windows 2003 Server installation CD.

Click 'Next >' when complete.

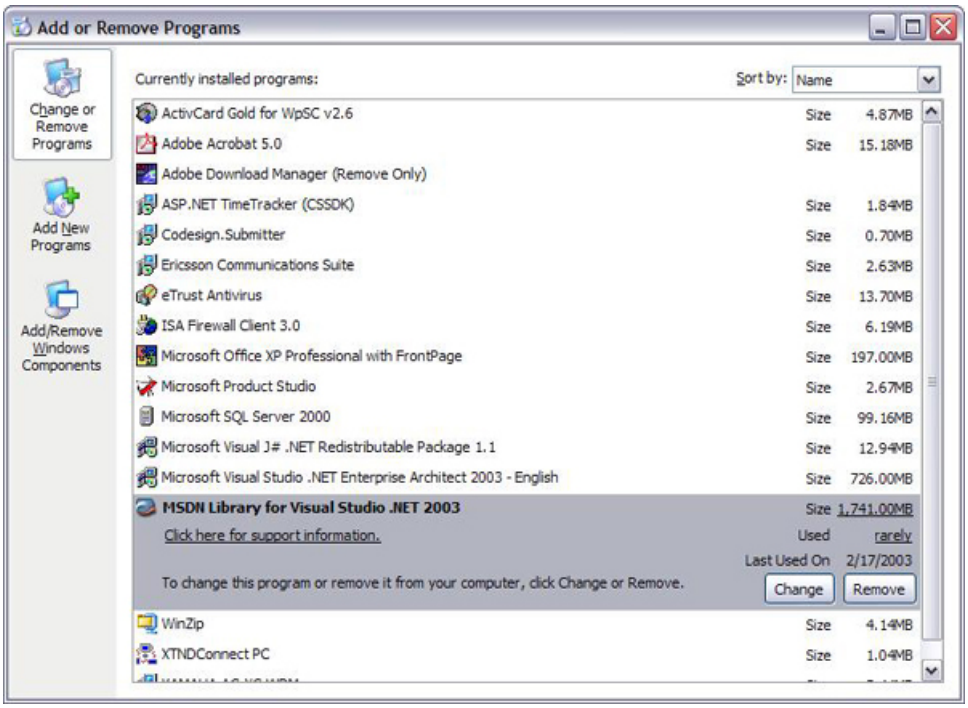


Click 'Finish' - the Windows Server 2003 is now configured to support IIS 6.0 and ASP.NET 1.1.

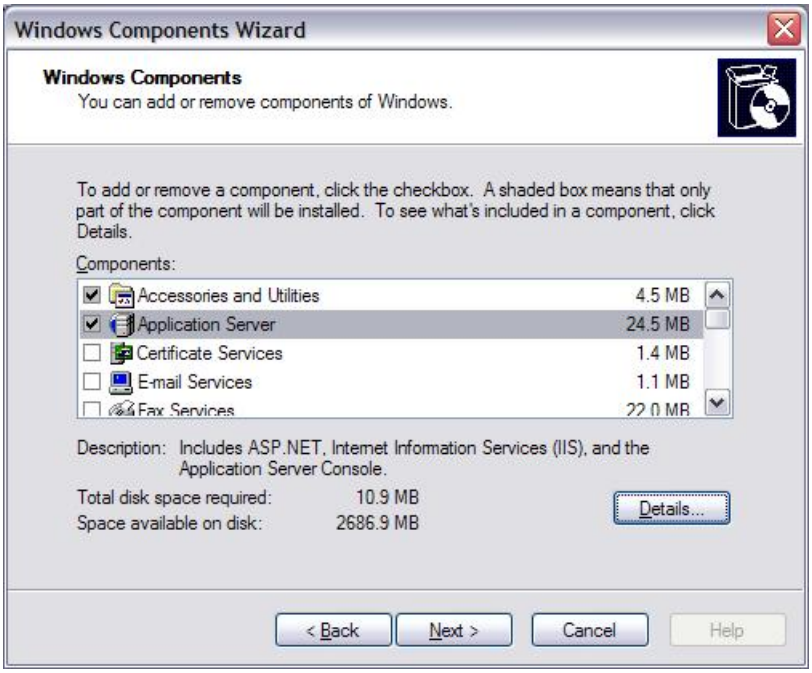
Enabling IIS, option #2 - Manually configuring IIS and ASP.NET

If you do not wish to use the 'Configure Your Server Wizard' you can optionally install IIS 6.0 and ASP.NET 1.1 using 'Add or Remove Programs' from the Control Panel.

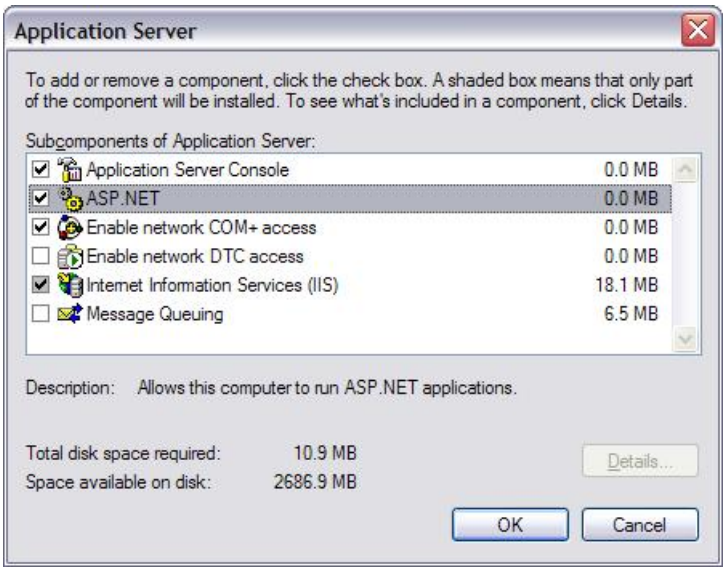
First open the Control Panel:



Next, click on 'Add/Remove Windows Components' which will open the 'Windows Components Wizard':

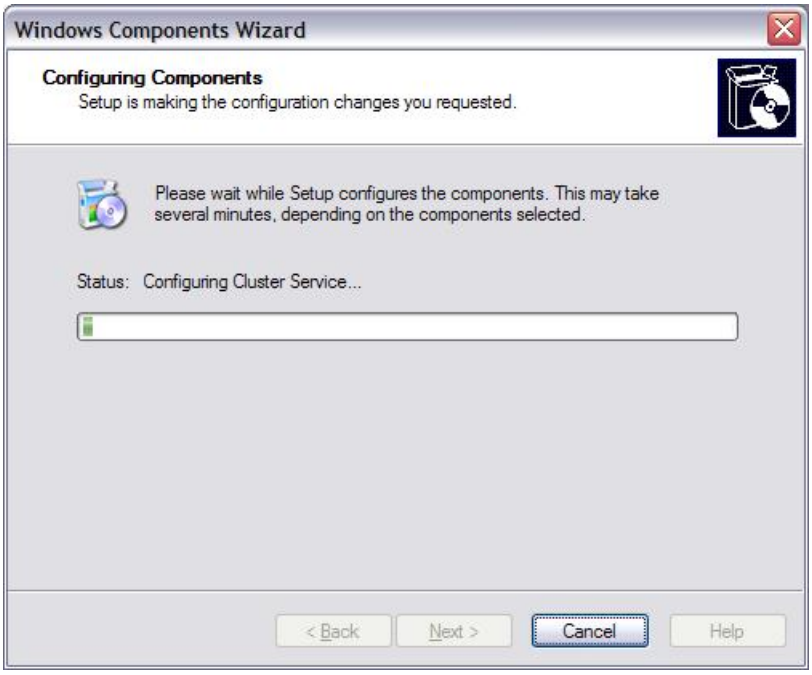


Highlight and check 'Application Server' and then click the 'Details?' button:



To install ASP.NET, check 'ASP.NET'.

Click 'OK' to return to the Windows Component Wizard. Click 'Next >' from the Windows Component Wizard to begin installing:



It is normal to see other dialog boxes appear as services are being installed. You may additionally be prompted for the location of the Windows 2003 Server installation CD.

When installation is complete you will see the last screen of the Windows Component Wizard:

IIS 6.0 and ASP.NET 1.1 are now configured and available.

Recommended Settings

When running ASP.NET 1.1 with IIS 6.0 there are several configuration settings that are recommended to get the optimal performance from ASP.NET:

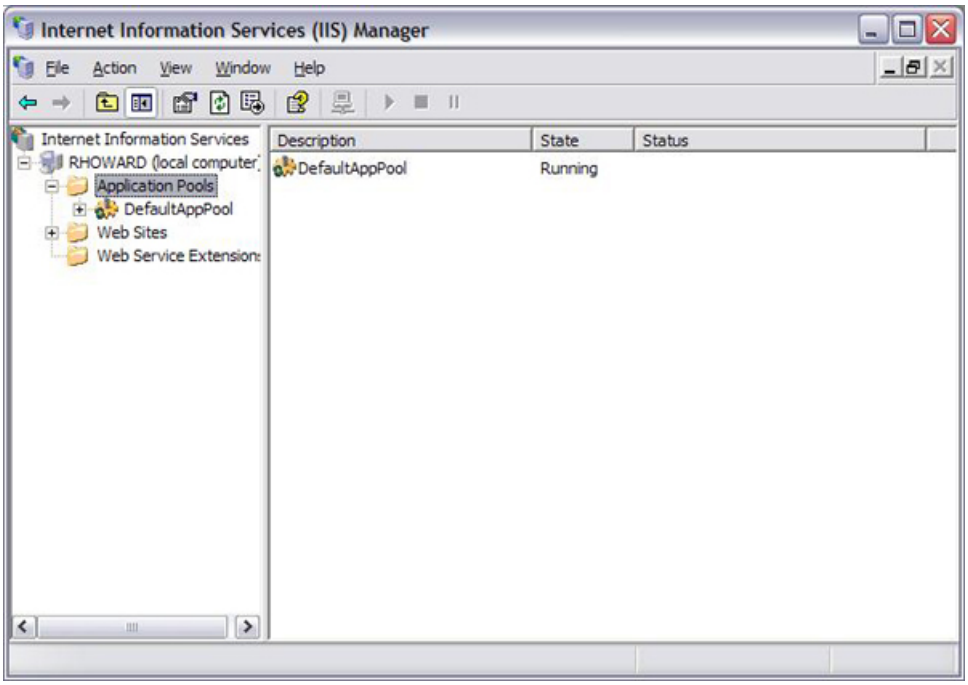
- Configuring worker process memory limits
- Configuring worker process recycling

Configuring worker process memory limits

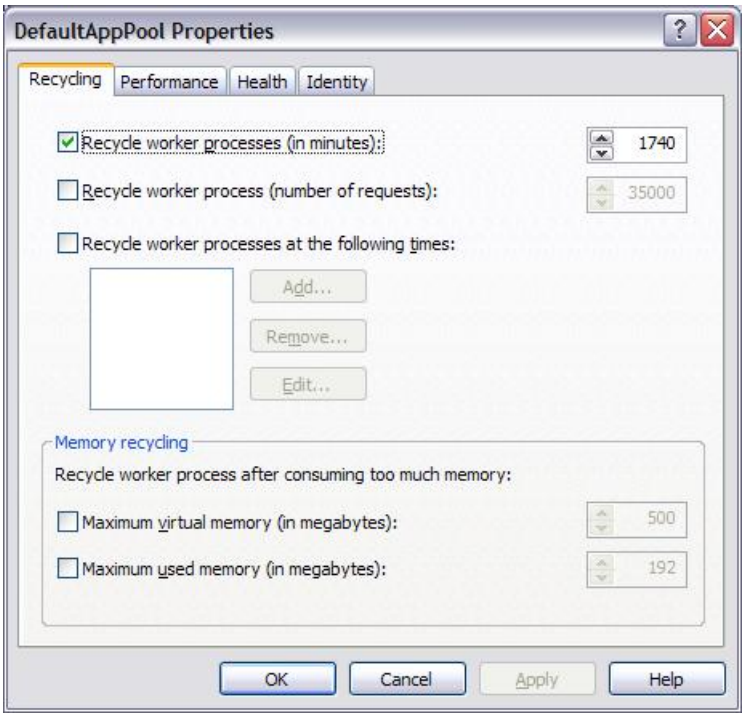
By default IIS 6.0 does not set a limit on the amount of memory that IIS is allowed to use. ASP.NET's Cache feature relies on a limitation of memory so the Cache can proactively remove unused items from memory.

It is recommended that you configure the memory recycling feature of IIS 6.0. To configure this open Internet Information Services Manager (Start | Programs | Administrative Tools | Internet Information Services). Once open, expand the 'Application Pools' folder:

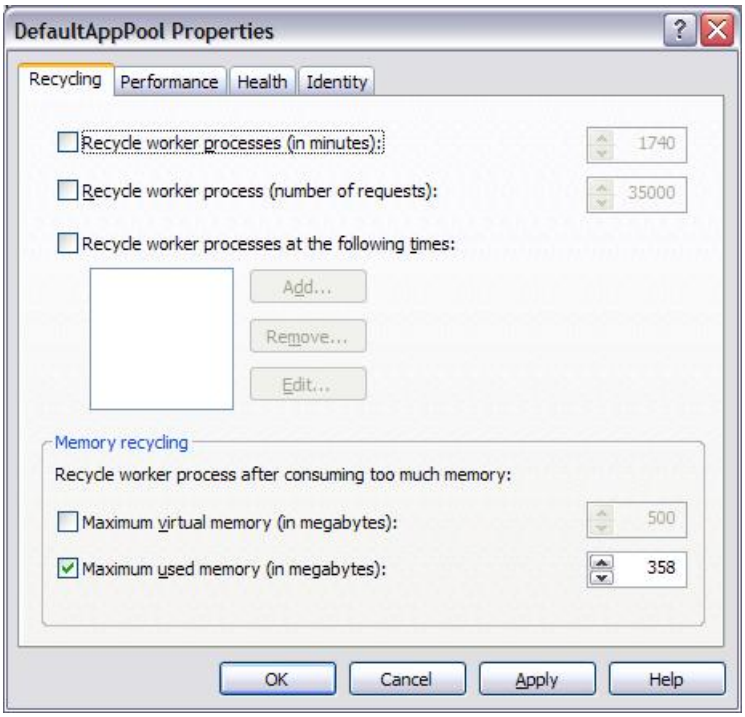
For each application pool:



1. Right-click on the application pool, e.g. 'DefaultAppPool', and select 'Properties':



2. Next, enable Memory recycling by clicking on either 'Maximum used memory (in megabytes):'. The value should not be more than the amount of physical (not virtual) memory on the server, a good approximation is 60% of the physical memory, i.e. for a server with 512MB of physical memory select 310. It is also recommended that the maximum not exceed 800MB when using a 2GB address space. If the memory address space of the server is 3GB, the maximum memory limit for the worker process can be as high as 1,800MB:

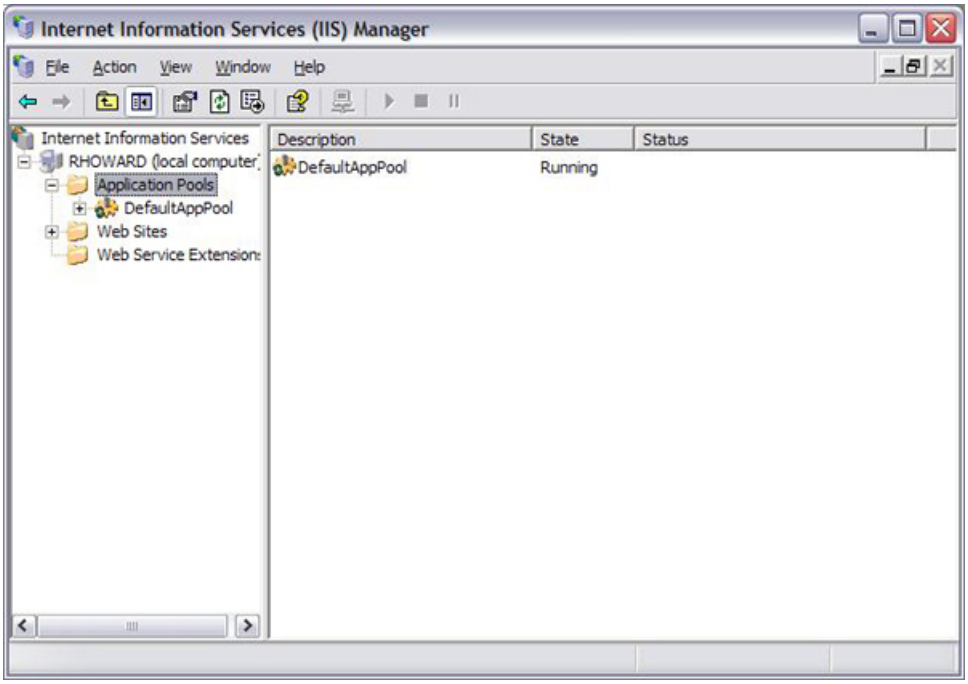


Click 'Apply' and the 'OK' to exit the properties dialog. Repeat this for all available application pools.

Configuring worker recycling

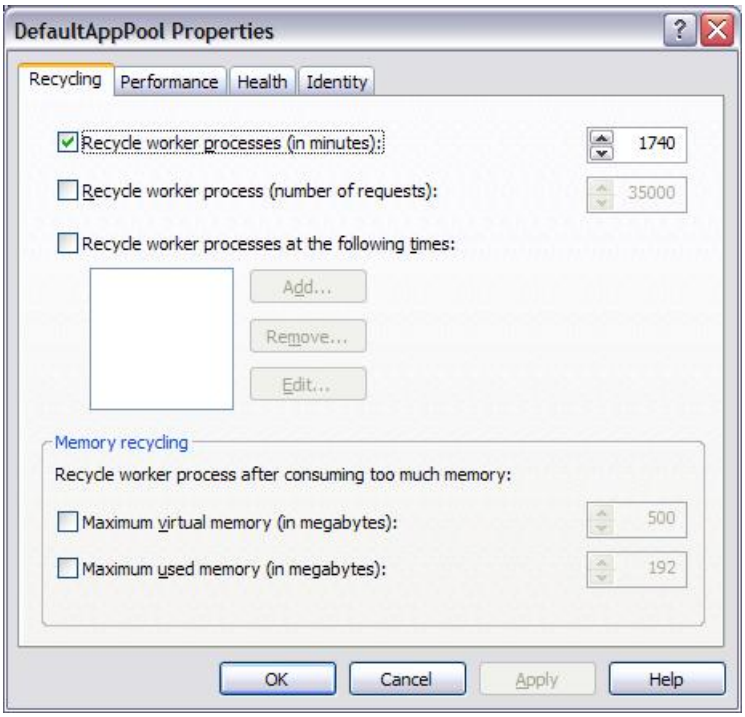
By default IIS 6.0 is configured to recycle its worker process every 29 hours. This is a bit aggressive for an application running ASP.NET and it is recommended that automatic worker process recycling is disabled.

To disable automatic worker process recycling, first open Internet Information Services Manager (Start | Programs | Administrative Tools | Internet Information Services). Once open, expand the 'Application Pools' folder:

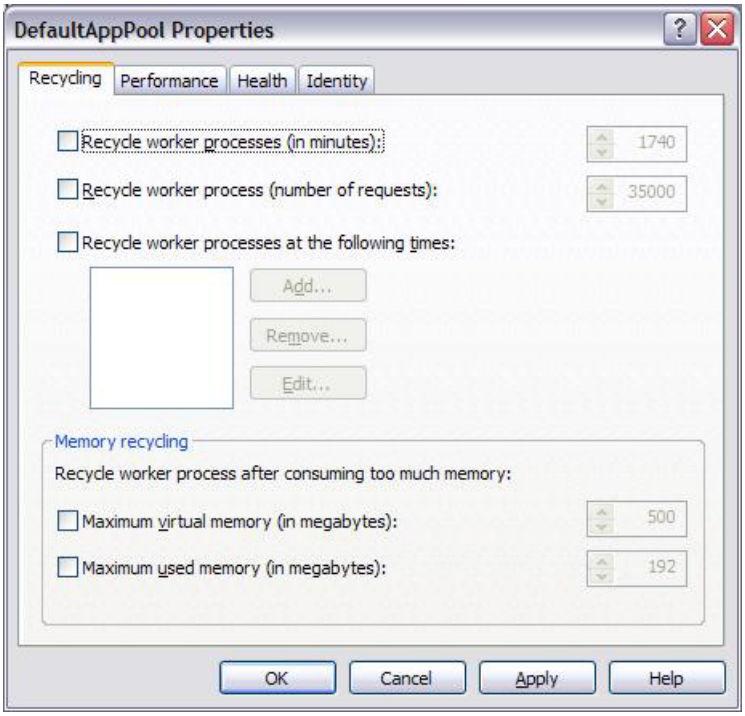


For each application pool:

1. Right-click on the application pool, e.g. 'DefaultAppPool', and select 'Properties':



2. Uncheck 'Recycle worker process (in minutes):':

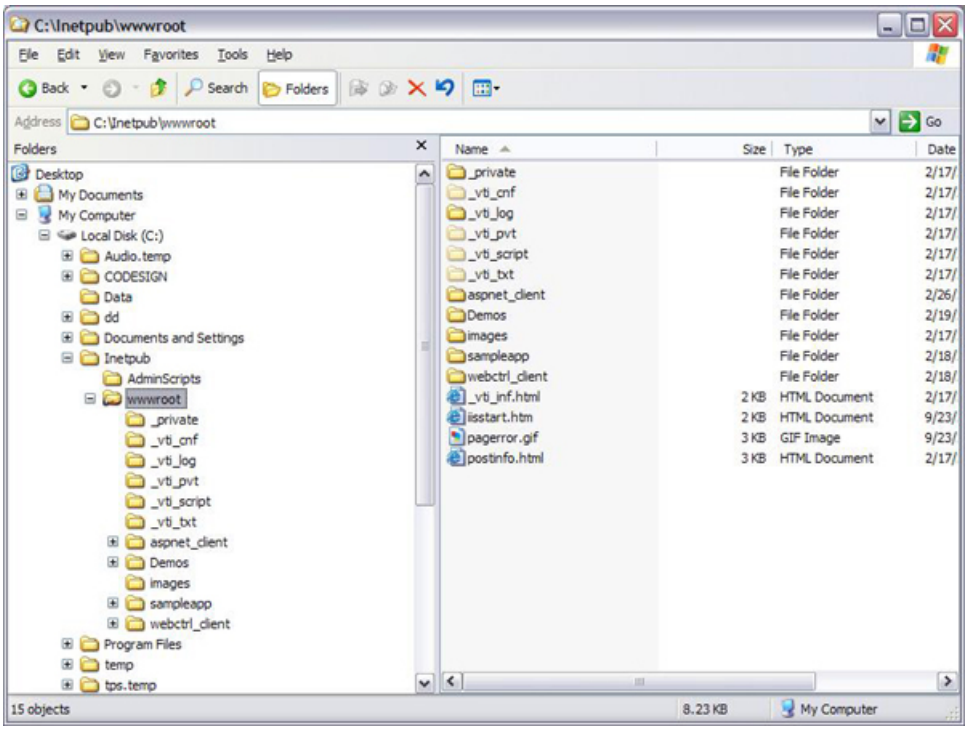


Click 'Apply' and the 'OK' to exit the properties dialog. Repeat this for all available application pools.

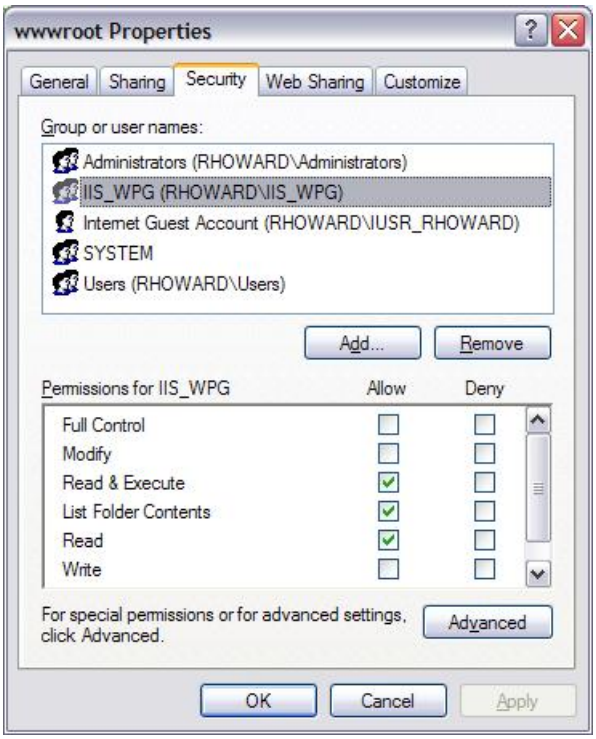
Granting write access to the file system

If your application requires write access to the file system and you are using NTFS you will need to modify an Access Control List (ACL) on the folder or file to grant ASP.NET access to.

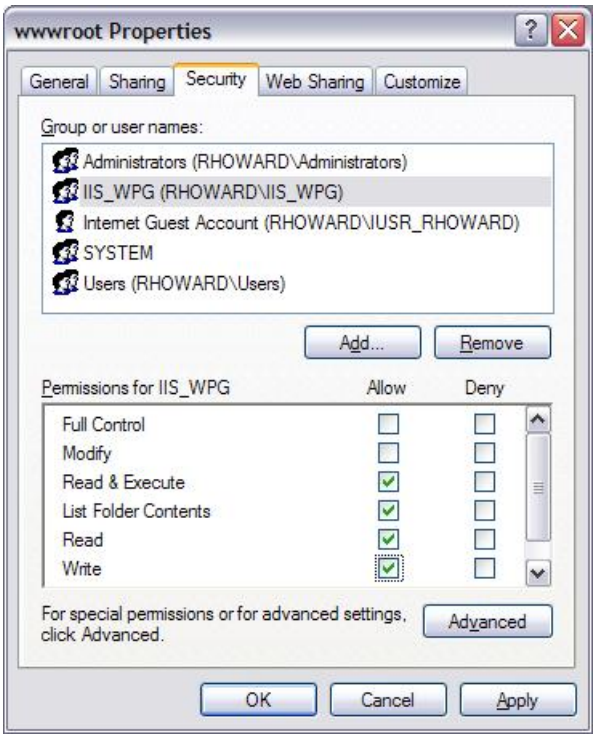
For example, to grant ASP.NET write access to the c:\inetpub\wwwroot first open explorer and navigate to the directory:



Next, right-click on the directory, e.g. 'wwwroot' and select properties. After the properties dialog opens, select the 'Security' tab:



The c:\inetpub\wwwroot\ directory is a special directory in that the special IIS 6.0 group 'IIS_WPG' is already granted Read & Execute, List Folder Contents, and Read permissions. However, to grant Write permission, you need to click the Allow checkbox for Write:



IIS 6.0 now has write permission on this folder. To grant write permissions on other folders, follow these steps - note, you may need to add the IIS_WPG group if it does not already exist.

Caution

Granting write permission to IIS_WPG will allow any ASP.NET application to write to this directory.

Supporting integrated authentication with SQL Server

Integrated authentication allows for SQL Server to leverage Windows NT authentication to validate SQL Server logon accounts. This allows the user to bypass the standard SQL Server logon process. With this approach, a network user can access a SQL Server database without supplying a separate logon identification or password because SQL Server obtains the user and password information from the Windows NT network security process.

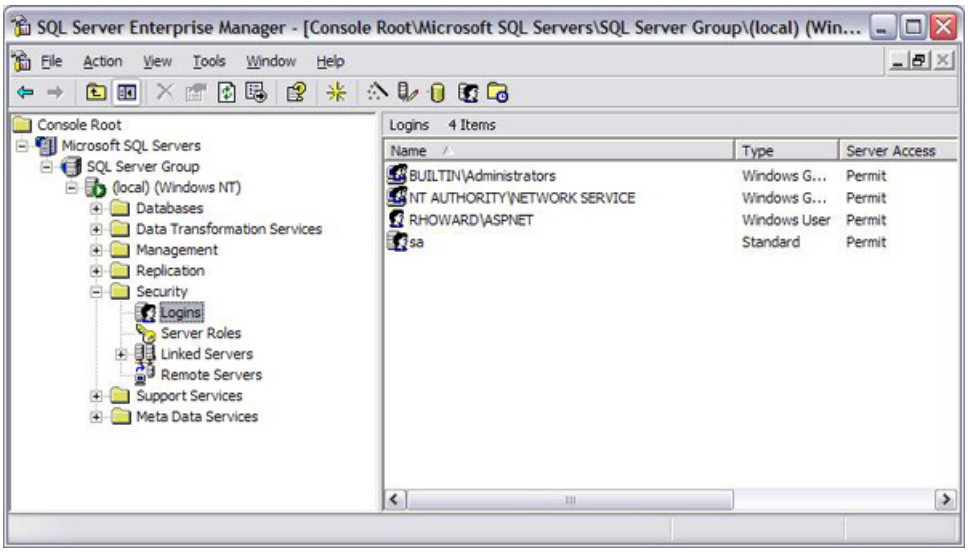
Choosing integrated authentication for ASP.NET applications is a good choice because no credentials are ever stored within your connection string for your application. Rather the connection string used to connect to SQL will look as follows:

```
"server=localhost; database=Northwind;Trusted_Connection=true"
```

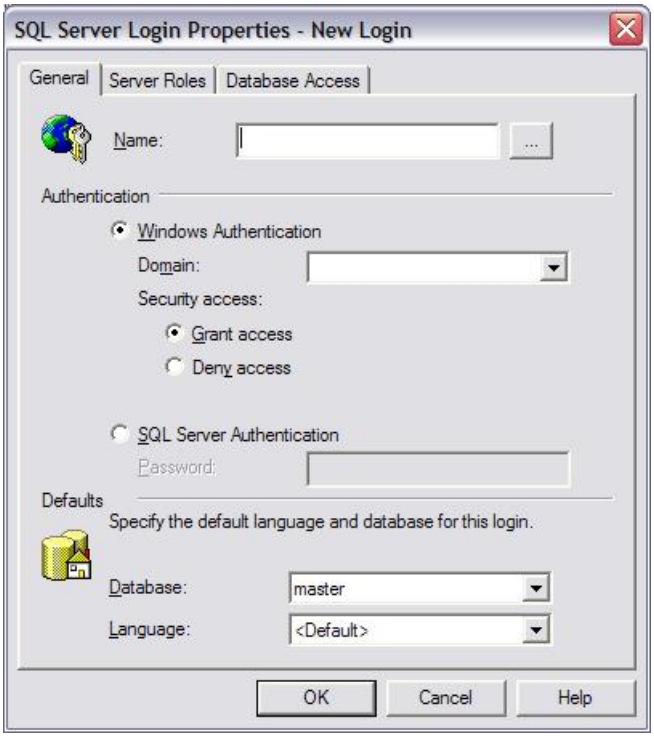
This connection string tells SQL Server to use the Windows credentials of the application attempting to access SQL Server. In the case of ASP.NET/IIS 6 this would be an account in the IIS_WPG group.

To enable integrated authentication between SQL Server and ASP.NET, you will need to first ensure that SQL Server is configured for either Integrated authentication or Mixed-Mode authentication - check with your DBA to determine this. If SQL Server is in one of these two modes, you can use integrated authentication.

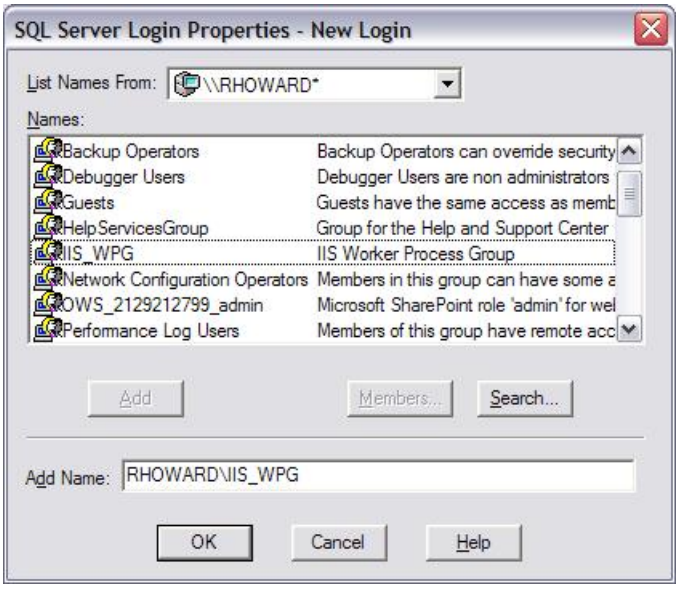
Open SQL Server Enterprise Manager (Start | Programs | Microsoft SQL Server | Enterprise Manager), select the appropriate server, and expand the Security folder:



If 'BUILTINT\IIS_WPG' group is not listed, right-click on Logins and select 'New Login':

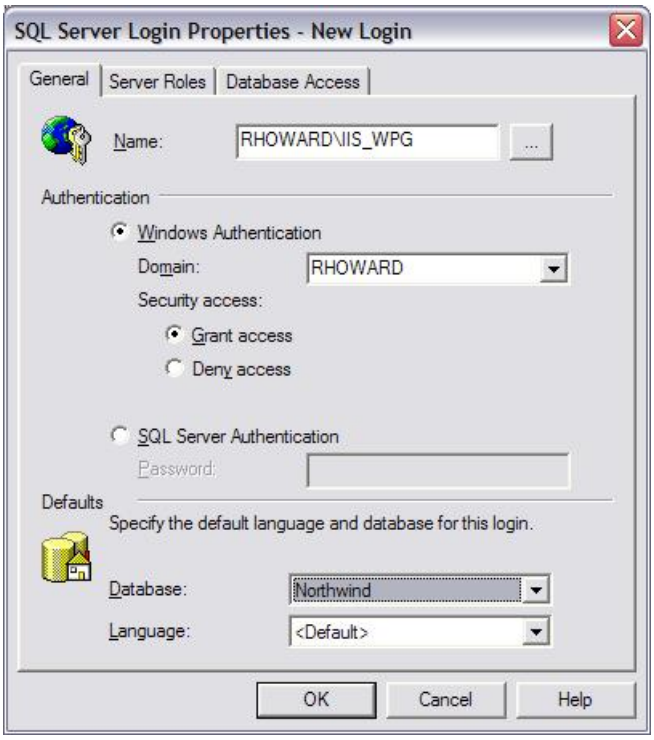


In the 'Name:' textbox either enter '[Server/Domain Name]\IIS_WPG' or click on the ellipses button to open the Windows NT user/group picker:

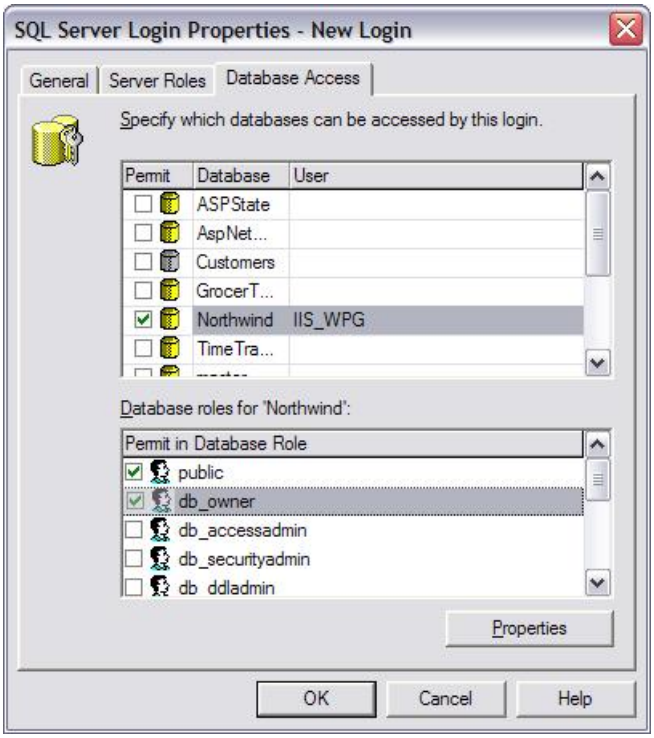


Select the current machine's IIS_WPG group and click 'Add' and OK to close the picker.

You then need to also set the default database and the permissions to access the database. To set the default database choose from the drop down list, e.g. below Northwind is selected:



Next, click on the Database Access tab:



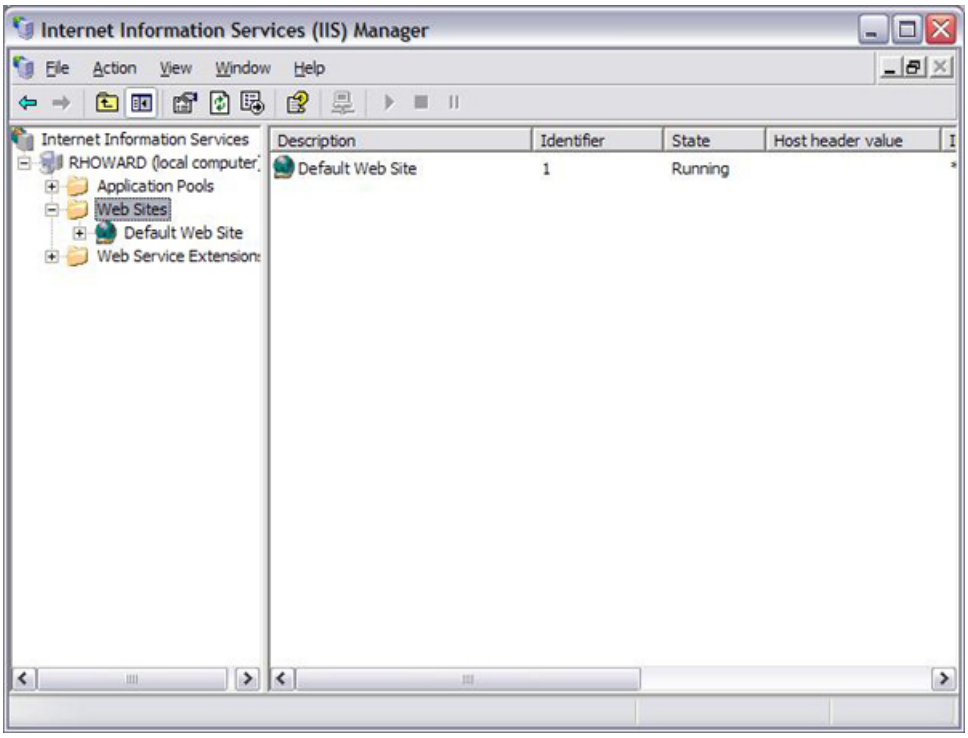
Click on the Permit checkbox for every database that you wish to allow access to. You will also need to select database roles, checking db_owner will ensure your login has all necessary permissions to manage and use the selected database.

Click OK to exit the property dialog. Your ASP.NET application is now configured to support integrated SQL Server authentication.

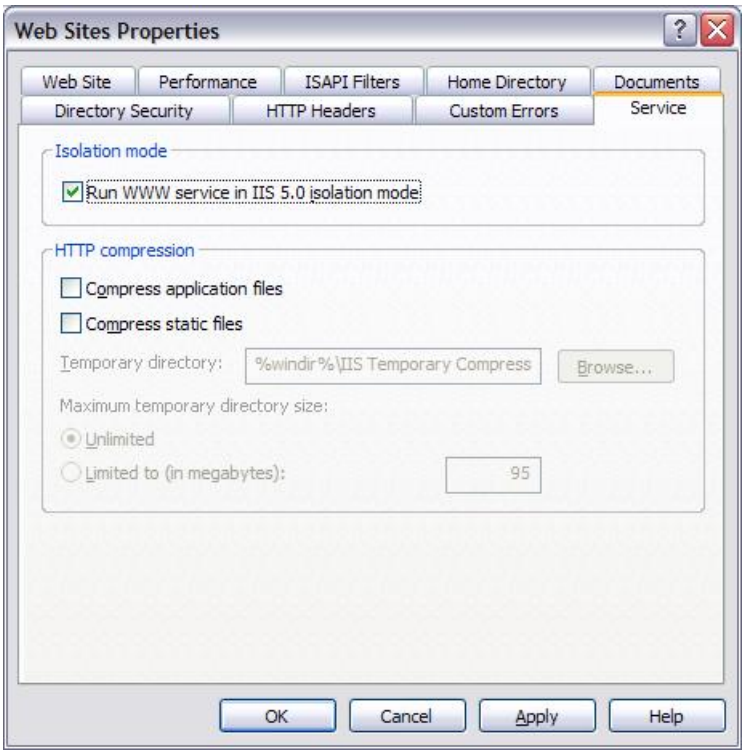
Don't run ASP.NET 1.0 in IIS 6.0 native mode

ASP.NET 1.0 on IIS 6.0 is only supported in IIS 5 compatibility mode.

To configure ASP.NET 1.0 to run in IIS 5.0 compatibility mode, open Internet Services Manager and right click Web Sites and select properties:



Switch to the Service Tab and check ?Run WWW Service in IIS 5.0 Isolation Mode?:



How To: Add Mobile Pages to Your ASP.NET Web Forms / MVC Application

2/20/2020 • 26 minutes to read • Edit Online

Applies To

- ASP.NET Web Forms version 4.0
- ASP.NET MVC version 3.0

Summary

This How To describes various ways to serve pages optimized for mobile devices from your ASP.NET Web Forms / MVC application, and suggests architectural and design issues to consider when targeting a broad range of devices. This document also explains why the ASP.NET Mobile Controls from ASP.NET 2.0 to 3.5 are now obsolete, and discusses some modern alternatives.

Contents

- Overview
- Architectural options
- Browser and device detection
- How ASP.NET Web Forms applications can present mobile-specific pages
- How ASP.NET MVC applications can present mobile-specific pages
- Additional resources

For downloadable code samples demonstrating this white paper's techniques for both ASP.NET Web Forms and MVC, see Mobile Apps & Sites with ASP.NET.

Overview

Mobile devices – smartphones, feature phones, and tablets – continue to grow in popularity as a means to access the Web. For many web developers and web-oriented businesses, this means it's increasingly important to provide a great browsing experience for visitors using those devices.

How earlier versions of ASP.NET supported mobile browsers

ASP.NET versions 2.0 to 3.5 included *ASP.NET Mobile Controls*: a set of server controls for mobile devices in the *System.Web.Mobile.dll* assembly and the *System.Web.UI.MobileControls* namespace. The assembly is still included in ASP.NET 4, but it is deprecated. Developers are advised to migrate to more modern approaches, such as those described in this paper.

The reason why ASP.NET Mobile Controls have been marked as obsolete is that their design is oriented around the mobile phones that were common around 2005 and earlier. The controls are mainly designed to render WML or cHTML markup (instead of regular HTML) for the WAP browsers of that era. But WAP, WML, and cHTML are no longer relevant for most projects, because HTML has now become the ubiquitous markup language for mobile and desktop browsers alike.

The challenges of supporting mobile devices today

Even though mobile browsers now almost universally support HTML, you will still face many challenges when aiming to create great mobile browsing experiences:

- *Screen size* - Mobile devices vary dramatically in form, and their screens are often much smaller than desktop monitors. So, you may need to design completely different page layouts for them.
- *Input methods* – Some devices have keypads, some have styluses, others use touch. You may need to consider multiple navigation mechanisms and data input methods.
- *Standards compliance* – Many mobile browsers do not support the latest HTML, CSS, or JavaScript standards.
- *Bandwidth* – Cellular data network performance varies wildly, and some end users are on tariffs that charge by the megabyte.

There's no one-size-fits-all solution; your application will have to look and behave differently according to the device accessing it. Depending on what level of mobile support you want, this can be a bigger challenge for web developers than the desktop "browser wars" ever was.

Developers approaching mobile browser support for the first time often initially think it's only important to support the latest and most sophisticated smartphones (e.g., Windows Phone 7, iPhone, or Android), perhaps because developers often personally own such devices. However, cheaper phones are still extremely popular, and their owners do use them to browse the web – especially in countries where mobile phones are easier to get than a broadband connection. Your business will need to decide what range of devices to support by considering its likely customers. If you're building an online brochure for a luxury health spa, you might make a business decision only to target advanced smartphones, whereas if you're creating a ticket booking system for a cinema, you probably need to account for visitors with less powerful feature phones.

Architectural options

Before we get to the specific technical details of ASP.NET Web Forms or MVC, note that web developers in general have three main possible options for supporting mobile browsers:

1. *Do nothing* – You can simply create a standard, desktop-oriented web application, and rely on mobile browsers to render it acceptably.

   - **Advantage**: It's the cheapest option to implement and maintain – no extra work

   - **Disadvantage**: Gives the worst end-user experience:

     - The latest smartphones may render your HTML just as well as a desktop browser, but users will still be forced to zoom and scroll horizontally and vertically to consume your content on a small screen. This is far from optimal.
     - Older devices and feature phones may fail to render your markup in a satisfactory way.
     - Even on the latest tablet devices (whose screens can be as big as laptop screens), different interaction rules apply. Touch-based input works best with larger buttons and links spread further apart, and there's no way to hover a mouse cursor over a fly-out menu.
2. *Solve the problem on the client* – With careful use of CSS and progressive enhancement you can create markup, styles, and scripts that adapt to whatever browser is running them. For example, with CSS 3 media queries, you could create a multi-column layout that turns into a single column layout on devices whose screens are narrower than a chosen threshold.

   - **Advantage**: Optimizes rendering for the specific device in use, even for unknown future devices according to whatever screen and input characteristics they have
   - **Advantage**: Easily lets you share server-side logic across all device types – minimal duplication of code or effort
   - **Disadvantage**: Mobile devices are so different from desktop devices that you may really want your mobile pages to be completely different from your desktop pages, showing different information. Such variations can be inefficient or impossible to achieve robustly through CSS alone, especially considering how inconsistently older devices interpret CSS rules. This is particularly true of CSS 3 attributes.

- **Disadvantage**: Provides no support for varying server-side logic and workflows for different devices. You can't, for example, implement a simplified shopping cart checkout workflow for mobile users by means of CSS alone.
- **Disadvantage**: Inefficient bandwidth use. You server may have to transmit the markup and styles that apply to all possible devices, even though the target device will only use a subset of that information.

3. ***Solve the problem on the server*** – If your server knows what device is accessing it – or at least the characteristics of that device, such as its screen size and input method, and whether it's a mobile device – it can run different logic and output different HTML markup.

- **Advantage:** Maximum flexibility. There's no limit to how much you can vary your server-side logic for mobiles or optimize your markup for the desired, device-specific layout.
- **Advantage:** Efficient bandwidth use. You only need to transmit the markup and styling information that the target device is going to use.
- **Disadvantage:** Sometimes forces repetition of effort or code (e.g., making you create similar but slightly different copies of your Web Forms pages or MVC views). Where possible you will factor out common logic into an underlying layer or service, but still, some parts of your UI code or markup may have to be duplicated and then maintained in parallel.
- **Disadvantage:** Device detection is not trivial. It requires a list or database of known device types and their characteristics (which may not always be perfectly up-to-date) and isn't guaranteed to accurately match every incoming request. This document describes some options and their pitfalls later.

To get the best results, most developers find they need to combine options (2) and (3). Minor stylistic differences are best accommodated on the client using CSS or even JavaScript, whereas major differences in data, workflow, or markup are most effectively implemented in server-side code.

This paper focuses on server-side techniques

Since ASP.NET Web Forms and MVC are both primarily server-side technologies, this white paper will focus on server-side techniques that let you produce different markup and logic for mobile browsers. Of course, you can also combine this with any client-side technique (e.g., CSS 3 media queries, progressive-enhancement JavaScript), but that's more a matter of web design than ASP.NET development.

Browser and device detection

The key prerequisite for all server-side techniques for supporting mobile devices is to know what device your visitor is using. In fact, even better than knowing the manufacturer and model number of that device is knowing the *characteristics* of the device. Characteristics may include:

- Is it a mobile device?
- Input method (mouse/keyboard, touch, keypad, joystick, ...)
- Screen size (physically and in pixels)
- Supported media and data formats
- Etc.

It's better to make decisions based on characteristics than model number, because then you'll be better equipped to handle future devices.

Using ASP.NET's built-in browser detection support

ASP.NET Web Forms and MVC developers can immediately discover important characteristics of a visiting browser by inspecting properties of the *Request.Browser* object. For example, see

- Request.Browser.IsMobileDevice
- Request.Browser.MobileDeviceManufacturer, Request.Browser.MobileDeviceModel
- Request.Browser.ScreenPixelsWidth

- Request.Browser.SupportsXmlHttp
- ...and many others

Behind the scenes, the ASP.NET platform matches the incoming *User-Agent* (UA) HTTP header against regular expressions in a set of Browser Definition XML files. By default the platform includes definitions for many common mobile devices, and you can add custom Browser Definition files for others you wish to recognize. For more details, see the MSDN page ASP.NET Web Server Controls and Browser Capabilities.

Using the WURFL device database via 51Degrees.mobi Foundation

While ASP.NET's built-in browser detection support will be sufficient for many applications, there are two main cases when it might not be enough:

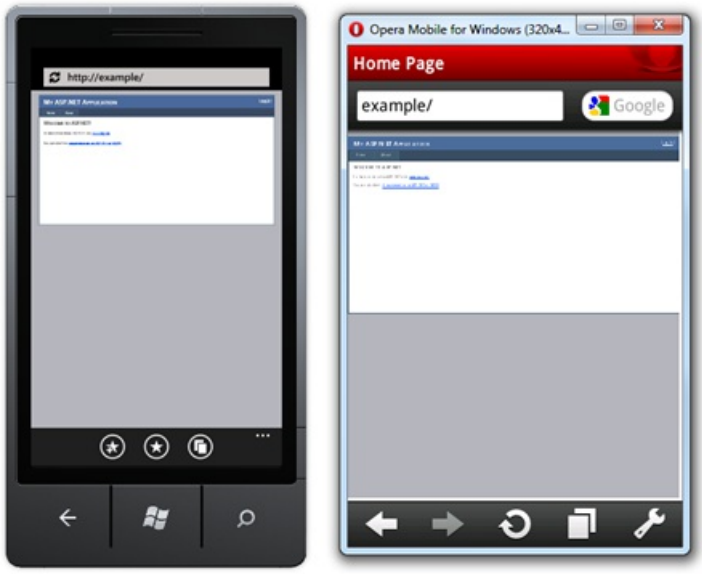
- ***You want to recognize the latest devices***(without manually creating Browser Definition files for them). Note that .NET 4's Browser Definition files are not recent enough to recognize Windows Phone 7, Android phones, Opera Mobile browsers, or Apple iPads.
- ***You need more detailed information about device capabilities***. You may need to know about a device's input method (e.g., touch vs keypad), or what audio formats the browser supports. This information isn't available in the standard Browser Definition files.

The *Wireless Universal Resource File* (WURFL) project maintains much more up-to-date and detailed information about mobile devices in use today.

The great news for .NET developers is that ASP.NET's browser detection feature is extensible, so it's possible to enhance it to overcome these problems. For example, you can add the open source *51Degrees.mobi Foundation* library to your project. It's an ASP.NET IHttpModule or Browser Capabilities Provider (usable on both Web Forms and MVC applications), that directly reads WURFL data and hooks it into ASP.NET's built-in browser detection mechanism. Once you've installed the module, *Request.Browser* will suddenly contain a lot more accurate and detailed information: it will correctly recognize many of the devices previously mentioned and list their capabilities (including additional capabilities such as input method). See the project's documentation for more details.

How Web Forms applications can present mobile-specific pages

By default, here's how a brand new Web Forms application displays on common mobile devices:



Windows Phone 7

Opera Mobile
(using same resolution as HTC Hero)

Clearly, neither layout looks very mobile-friendly – this page was designed for a large, landscape-oriented monitor, not for a small portrait-oriented screen. So what can you do about it?

As discussed earlier in this paper, there are many ways to tailor your pages for mobile devices. Some techniques

are server-based, others run on the client.

Creating a mobile-specific master page

Depending on your requirements, you may be able to use the same Web Forms for all visitors, but have two separate master pages: one for desktop visitors, another for mobile visitors. This gives you the flexibility of changing the CSS stylesheet or your top-level HTML markup to suit mobile devices, without forcing you to duplicate any page logic.

This is easy to do. For example, you can add a PreInit handler such as the following to a Web Form:

```
protected void Page_PreInit(object sender, EventArgs e)
{
    if (Request.Browser.IsMobileDevice)
        MasterPageFile = "~/Mobile.Master";
}
```

Now, create a master page called Mobile.Master in the top-level folder of your application, and it will be used when a mobile device is detected. Your mobile master page can reference a mobile-specific CSS stylesheet if necessary. Desktop visitors will still see your default master page, not the mobile one.

Creating independent mobile-specific Web Forms

For maximum flexibility, you can go much further than just having separate master pages for different device types. You can implement two *totally separate sets of Web Forms pages* – one set for desktop browsers, another set for mobiles. This works best if you want to present very different information or workflows to mobile visitors. The rest of this section describes this approach in detail.

Assuming you already have a Web Forms application designed for desktop browsers, the easiest way to proceed is to create a subfolder called "Mobile" within your project, and build your mobile pages there. You can construct an entire sub-site, with its own master pages, style sheets, and pages, using all the same techniques that you'd use for any other Web Forms application. You don't necessarily need to produce a mobile equivalent for *every* page in your desktop site; you can choose what subset of functionality makes sense for mobile visitors.

Your mobile pages can share common static resources (such as images, JavaScript, or CSS files) with your regular pages if you wish. Since your "Mobile" folder will *not* be marked as a separate application when hosted in IIS (it's just a simple subfolder of Web Forms pages), it will also share all the same configuration, Session data, and other infrastructure as your desktop pages.

> **NOTE**
>
> Since this approach usually involves some duplication of code (mobile pages are likely to share some similarities with desktop pages), it's important to factor out any common business logic or data access code into a shared underlying layer or service. Otherwise, you'll double the effort of creating and maintaining your application.

Redirecting mobile visitors to your mobile pages

It's often convenient to redirect mobile visitors to the mobile pages only on the *first* request in their browsing session (and not on every request in their session), because:

- You can then easily allow mobile visitors to access your desktop pages if they wish – just put a link on your master page that goes to "Desktop version". The visitor won't be redirected back to a mobile page, because it's no longer the first request in their session.
- It avoids the risk of interfering with requests for any dynamic resources shared between desktop and mobile parts of your site (e.g., if you have a common Web Form that both desktop and mobile parts of your site display in an IFRAME, or certain Ajax handlers)

To do this, you can place your redirection logic in a `Session_Start` method. For example, add the following

method to your Global.asax.cs file:

```
void Session_Start(object sender, EventArgs e)
{
    // Redirect mobile users to the mobile home page
    HttpRequest httpRequest = HttpContext.Current.Request;
    if (httpRequest.Browser.IsMobileDevice)
    {
        string path = httpRequest.Url.PathAndQuery;
        bool isOnMobilePage = path.StartsWith("/Mobile/",
                                StringComparison.OrdinalIgnoreCase);
        if (!isOnMobilePage)
        {
            string redirectTo = "~/Mobile/";

            // Could also add special logic to redirect from certain
            // recognized pages to the mobile equivalents of those
            // pages (where they exist). For example,
            // if (HttpContext.Current.Handler is UserRegistration)
            //     redirectTo = "~/Mobile/Register.aspx";

            HttpContext.Current.Response.Redirect(redirectTo);
        }
    }
}
```

Configuring Forms Authentication to respect your mobile pages

Note that Forms Authentication makes certain assumptions about where it can redirect visitors during and after the authentication process:

- When a user needs to be authenticated, Forms Authentication will redirect them to your desktop login page, regardless of whether they're a desktop or mobile user (because it only has a concept of *one* login URL). Assuming you want to style your mobile login page differently, you need to enhance your desktop login page so that it redirects mobile users to a separate mobile login page. For example, add the following code to your **desktop** login page code-behind:

  ```
  public partial class Login : System.Web.UI.Page
  {
      protected void Page_Load(object sender, EventArgs e)
      {
          // Ensure that if Forms Authentication forces a mobile user
          // to log in, we display the mobile login page
          string returnUrl = Request.QueryString["ReturnUrl"];
          if (!String.IsNullOrEmpty(returnUrl) && returnUrl.StartsWith("/Mobile/",
                                                StringComparison.OrdinalIgnoreCase))
          {
              Response.Redirect("~/Mobile/Account/Login.aspx?ReturnUrl="
                                + HttpUtility.UrlEncode(returnUrl));
          }

          RegisterHyperLink.NavigateUrl = "Register.aspx?ReturnUrl="
                                        + HttpUtility.UrlEncode(returnUrl);
      }
  }
  ```

- After a user successfully logs in, Forms Authentication will by default redirect them to your desktop home page (because it only has a concept of *one* default page). You need to enhance your mobile login page so that it redirects to the mobile home page after a successful log-in. For example, add the following code to your **mobile** login page code-behind:

```
public partial class Login : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        // Ensure that after logging in, mobile users stay on mobile pages
        string returnUrl = Request.QueryString["ReturnUrl"];
        if (String.IsNullOrEmpty(returnUrl))
        {
            returnUrl = "~/Mobile/";
        }
        LoginUser.DestinationPageUrl = returnUrl;

        // (the following line is already present by default)
        RegisterHyperLink.NavigateUrl = "Register.aspx?ReturnUrl="
                                        + HttpUtility.UrlEncode(returnUrl);
    }
}
```

This code assumes your page has a Login server control called LoginUser, as in the default project template.

Working with Output Caching

If you're using output caching, beware that by default it's possible for a desktop user to visit a certain URL (causing its output to be cached), followed by a mobile user who then receives the cached desktop output. This warning applies whether you're just varying your master page by device type, or implementing totally separate Web Forms per device type.

To avoid the problem, you can instruct ASP.NET to vary the cache entry according to whether the visitor is using a mobile device. Add a VaryByCustom parameter to your page's OutputCache declaration as follows:

```
<%@ OutputCache VaryByParam="*" Duration="60" VaryByCustom="isMobileDevice" %>
```

Next, define *isMobileDevice* as a custom cache parameter by adding the following method override to your Global.asax.cs file:

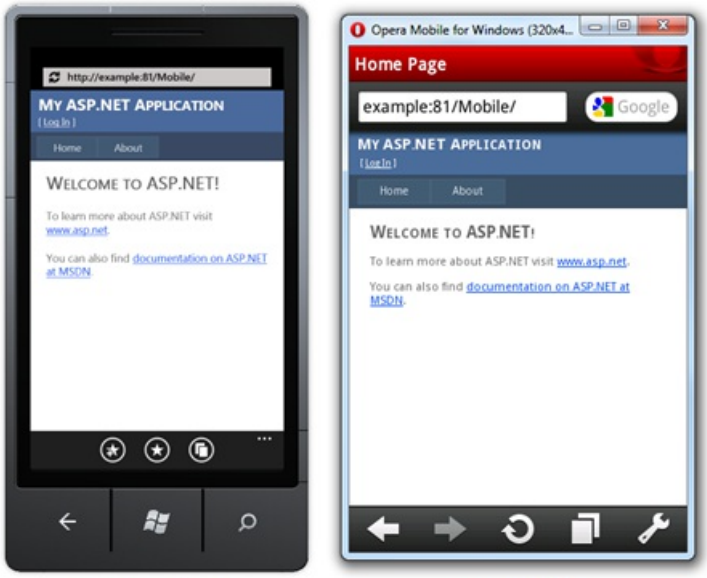
```
public override string GetVaryByCustomString(HttpContext context, string custom)
{
    if (string.Equals(custom, "isMobileDevice", StringComparison.OrdinalIgnoreCase))
        return context.Request.Browser.IsMobileDevice.ToString();

    return base.GetVaryByCustomString(context, custom);
}
```

This will ensure that mobile visitors to the page don't receive output previously put into the cache by a desktop visitor.

A working example

To see these techniques in action, download this white paper's code samples. The Web Forms sample application automatically redirects mobile users to a set of mobile-specific pages in a subfolder called Mobile. The markup and styling of those pages is better optimized for mobile browsers, as you can see from the following screenshots:



Windows Phone 7 **Opera Mobile**
(using same resolution as HTC Hero)

For more tips about optimizing your markup and CSS for mobile browsers, see the section "Styling mobile pages for mobile browsers" later in this document.

How ASP.NET MVC applications can present mobile-specific pages

Since the Model-View-Controller pattern decouples application logic (in controllers) from presentation logic (in views), you can choose from any of the following approaches to handling mobile support in server-side code:

1. ***Use the same controllers and views for both desktop and mobile browsers, but render the views with different Razor layouts depending on the device type.*** This option works best if you're displaying identical data on all devices, but simply want to supply different CSS stylesheets or change a few top-level HTML elements for mobiles.
2. ***Use the same controllers for both desktop and mobile browsers, but render different views depending on the device type.*** This option works best if you're displaying roughly the same data and providing the same workflows for end users, but want to render very different HTML markup to suit the device being used.
3. ***Create separate areas for desktop and mobile browsers, implementing independent controllers and views for each.*** This option works best if you're displaying very different screens, containing different information and leading the user through different workflows optimized for their device type. It may mean some repetition of code, but you can minimize that by factoring out common logic into an underlying layer or service.

If you want to take the **first** option and vary only the Razor layout per device type, it's very easy. Just modify your _ViewStart.cshtml file as follows:

```
@{
    Layout = Request.Browser.IsMobileDevice ? "~/Views/Shared/_LayoutMobile.cshtml"
                                            : "~/Views/Shared/_Layout.cshtml";
}
```

Now you can create a mobile-specific layout called _LayoutMobile.cshtml with a page structure and CSS rules optimized for mobile devices.

If you want to take the **second** option render totally different views according to the visitor's device type, see Scott Hanselman's blog post.

The rest of this paper focuses on the **third** option – creating separate controllers *and* views for mobile devices – so

you can control exactly what subset of functionality is offered for mobile visitors.

Setting up a Mobile area within your ASP.NET MVC application

You can add an area called "Mobile" to an existing ASP.NET MVC application in the normal way: right-click on your project name in Solution Explorer, then choose Add à Area. You can then add controllers and views as you would for any other area within an ASP.NET MVC application. For example, add to your Mobile area a new controller called HomeController to act as a homepage for mobile visitors.

Ensuring the URL /Mobile reaches the mobile homepage

If you want the URL /Mobile to reach the Index action on HomeController inside your Mobile area, you will need to make two small changes to your routing configuration. First, update your MobileAreaRegistration class so that HomeController is the default controller in your Mobile area, as shown in the following code:

```
public override void RegisterArea(AreaRegistrationContext context)
{
    // By default there is no "controller" parameter in the following line.
    // Add one referencing "Home" as shown.
    context.MapRoute(
        "Mobile_default",
        "Mobile/{controller}/{action}/{id}",
        new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    );
}
```

This means the mobile homepage will now be located at /Mobile, rather than /Mobile/Home, because "Home" is now the implicitly default controller name for mobile pages.

Next, note that by adding a second HomeController to your application (i.e., the mobile one, in addition to the existing desktop one), you'll have broken your regular desktop homepage. It will fail with the error "*Multiple types were found that match the controller named 'Home'*". To resolve this, update your top-level routing configuration (in Global.asax.cs) to specify that your desktop HomeController should take priority when there's ambiguity:

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        "Default",
        "{controller}/{action}/{id}",
        new { controller = "Home", action = "Index", id = UrlParameter.Optional },
        // Add the namespace of your desktop controllers here
        new[] { "YourApplication.Controllers" }
    );
}
```

Now the error will go away, and the URL http://*yoursite*/ will reach the desktop homepage, and http://*yoursite*/mobile/ will reach the mobile homepage.

Redirecting mobile visitors to your mobile area

There are many different extensibility points in ASP.NET MVC, so there are many possible ways to inject redirection logic. One neat option is to create a filter attribute, [RedirectMobileDevicesToMobileArea], that performs a redirection if the following conditions are met:

1. It's the first request in the user's session (i.e., Session.IsNewSession equals true)
2. The request comes from a mobile browser (i.e., Request.Browser.IsMobileDevice equals true)
3. The user is not already requesting a resource in the mobile area (i.e., the *path* part of the URL does not begin with /Mobile)

The downloadable sample included with this white paper includes an implementation of this logic. It's implemented as an authorization filter, derived from AuthorizeAttribute, which means it can work correctly even if you are using output caching (otherwise, if a desktop visitor first accesses a certain URL, the desktop output might be cached and then served to subsequent mobile visitors).

As it's a filter, you can choose either to apply it to specific controllers and actions, e.g.,

```
public class HomeController : Controller
{
    [RedirectMobileDevicesToMobileArea] // Applies just to this action
    public ActionResult Index()
    {
        // ...
    }
}
```

... or you can apply it to all controllers and actions as an MVC 3 *global filter* in your Global.asax.cs file:

```
protected void Application_Start()
{
    // (rest of method unchanged)

    // Using "order" value 1 means it will run after unordered filters
    // associated with specific controllers or actions, so the redirection
    // location can be overridden for specific actions
    GlobalFilters.Filters.Add(new RedirectMobileDevicesToMobileAreaAttribute(), 1);
}
```

The downloadable sample also demonstrates how you can create subclasses of this attribute that redirect to specific locations within your mobile area. This means, for example, you can:

- Register a global filter as shown above that sends mobile visitors to the mobile homepage by default.
- Also apply a special [RedirectMobileDevicesToMobileProductPage] filter to a "view product" action that takes mobile visitors to the mobile version of whatever product page they had requested.
- Also apply other special subclasses of the filter to specific actions, redirecting mobile visitors to the equivalent mobile page

Configuring Forms Authentication to respect your mobile pages

If you're using Forms Authentication, you should note that when a user needs to log in, it automatically redirects the user to a single specific "log on" URL, which by default is `/Account/LogOn`. This means that mobile users may be redirected to your desktop "log on" action.

To avoid this problem, add logic to your desktop "log on" action so that it redirects mobile users again to a mobile "log on" action. If you're using the default ASP.NET MVC application template, update AccountController's LogOn action as follows:

```
public ActionResult LogOn()
{
    string returnUrl = Request.QueryString["ReturnUrl"];
    if ((returnUrl != null) && returnUrl.StartsWith("/Mobile/",
                                StringComparison.OrdinalIgnoreCase))
    {
        return RedirectToAction("LogOn", "Account",
                                 new { Area = "Mobile", ReturnUrl = returnUrl });
    }
    return View();
}
```

... and then implement a suitable mobile-specific "log on" action on a controller called AccountController in your Mobile area.

Working with Output Caching

If you're using the [OutputCache] filter, you must force the cache entry to vary by device type. For example, write:

```
[OutputCache(Duration = 60, VaryByParam = "*", VaryByCustom = "isMobileDevice")]
```

Then, add the following method to your Global.asax.cs file:

```
public override string GetVaryByCustomString(HttpContext context, string custom)
{
    if (string.Equals(custom, "isMobileDevice", StringComparison.OrdinalIgnoreCase))
        return context.Request.Browser.IsMobileDevice.ToString();

    return base.GetVaryByCustomString(context, custom);
}
```

This will ensure that mobile visitors to the page don't receive output previously put into the cache by a desktop visitor.

A working example

To see these techniques in action, download this white paper's code associated samples. The sample includes an ASP.NET MVC 3 (Release Candidate) application enhanced to support mobile devices using the methods described above.

Further guidance and suggestions

The following discussion applies both to Web Forms and MVC developers who are using the techniques covered in this document.

Enhancing your redirection logic using 51Degrees.mobi Foundation

The redirection logic shown in this document may be perfectly sufficient for your application, but it won't work if you need to disable sessions, or with mobile browsers that reject cookies (these can't have sessions), because it won't know whether a given request is the first one from that visitor.

You already learned how the open source 51Degrees.mobi Foundation can improve the accuracy of ASP.NET's browser detection. It also has a built-in ability to redirect mobile visitors to specific locations configured in Web.config. It's able to work without depending on ASP.NET Sessions (and hence cookies) by storing a temporary log of hashes of visitors' HTTP headers and IP addresses, so it knows whether or not each request is the first one from a given vistor.

The following element added to the fiftyOne section of the web.config file will redirect the first request from a detected mobile device to the page ~/Mobile/Default.aspx. Any requests to pages under the Mobile folder will *not* be redirected, regardless of device type. If the mobile device has been inactive for a period of 20 minutes or more the device will be forgotten and subsequent requests will be treated as new ones for the purposes of redirection.

```
<redirect firstRequestOnly="true"
          mobileHomePageUrl="~/Mobile/Default.aspx"
          timeout="20"
          devicesFile="~/App_Data/Devices.dat"
          mobilePagesRegex="/Mobile/" />
```

For more details, see 51degrees.mobi Foundation documentation.

> **NOTE**
> You *can* use 51Degrees.mobi Foundation's redirection feature on ASP.NET MVC applications, but you will need to define your redirection configuration in terms of plain URLs, not in terms of routing parameters or by putting MVC filters on actions. This is because (at the time of writing) 51Degrees.mobi Foundation does not recognize filters or routing.

Disabling Transcoders and Proxy Servers

Mobile network operators have two broad objectives in their approach to the mobile internet:

1. Provide as much relevant content as possible
2. Maximize the number of customers who can share the limited radio network bandwidth

Since most web pages were designed for large desktop-sized screens and fast fixed-line broadband connections, many operators use *transcoders* or *proxy servers* that dynamically alter web content. They may modify your HTML markup or CSS to suit smaller screens (especially for "feature phones" that lack the processing power to handle complex layouts), and they may recompress your images (significantly reducing their quality) to improve page delivery speeds.

But if you've taken the effort to produce a mobile-optimized version of your site, you probably don't want the network operator to interfere with it any further. You can add the following line to the Page_Load event in any ASP.NET Web Form:

```
Response.Cache.SetNoTransforms();
```

Or, for an ASP.NET MVC controller, you could add the following method override so that it applies to all actions on that controller:

```
protected override void OnActionExecuting(ActionExecutingContext filterContext)
{
    filterContext.HttpContext.Response.Cache.SetNoTransforms();
}
```

The resulting HTTP message informs W3C compliant transcoders and proxies not to alter content. Of course, there is no guarantee that mobile network operators will respect this message.

Styling mobile pages for mobile browsers

It's beyond the scope of this document to describe in great detail what kinds of HTML markup work correctly or which web design techniques maximize usability on particular devices. It's up to you to find a sufficiently simple layout, optimized for a mobile-sized screen, without using unreliable HTML or CSS positioning tricks. One important technique worth mentioning, however, is the *viewport meta tag*.

Certain modern mobile browsers, in an effort display web pages meant for desktop monitors, render the page on a virtual canvas, also called "viewport" (e.g., the virtual viewport is 980 pixels wide on iPhone, and 850 pixels wide on Opera Mobile by default) and then scale the result down to fit onto the screen's physical pixels. The user can then zoom in and pan around that viewport. This has the advantage that it lets the browser display the page in its intended layout, but it's also has the disadvantage that it forces zooming and panning, which is inconvenient for the user. If you're designing for mobile, it's better to design for a narrow screen so that no zooming or horizontal scrolling is necessary.

A way to tell the mobile browser how wide the viewport should be is through the nonstandard *viewport* meta tag. For example, if you add the following to your page's HEAD section,

```
<meta name="viewport" content="width=480">
```

... then supporting smartphone browsers will lay out the page on a 480-pixel wide virtual canvas. This means that if your HTML elements define their widths in percentage terms, the percentages will be interpreted with respect to this 480-pixel width, not the default viewport width. As a result, the user is less likely to have to zoom and pan horizontally – considerably improving the mobile browsing experience.

If you want the viewport width to match the device's physical pixels, you can specify the following:

```
<meta name="viewport" content="width=device-width">
```

For this to work correctly, you must not explicitly force elements to exceed that width (e.g., using a *width* attribute or CSS property), otherwise the browser will be forced to use a larger viewport regardless. See also: more details about the nonstandard viewport tag.

Most modern smartphones support *dual orientation*: they can be held in either portrait or landscape mode. So, it's important not to make assumptions about the screen width in pixels. Don't even assume that the screen width is fixed, because the user can re-orient their device while they are on your page.

Older Windows Mobile and Blackberry devices may also accept the following meta tags in the page header to inform them content has been optimized for mobile and therefore should not be transformed.

```
<meta name="MobileOptimized" content="width" />
<meta name="HandheldFriendly" content="true" />
```

Additional Resources

For a list of mobile device emulators and simulators you can use to test your mobile ASP.NET web application, see the page Simulate popular mobile devices for testing.

Credits

- Primary author: Steven Sanderson
- Reviewers / additional content writers: James Rosewell, Mikael Söderström, Scott Hanselman, Scott Hunter

Fix for 'Server Application Unavailable' Error after Applying Security Update for IE

2/20/2020 • 3 minutes to read • Edit Online

> This paper describes the patch that fixes an issue with the MS03-32 Security Update for Internet Explorer that affects ASP.NET 1.0 applications running on Windows XP Professional.
>
> Applies to ASP.NET 1.0 and Windows XP Professional.

Microsoft identified an issue with the MS03-32 Security Update for Internet Explorer security patch and ASP.NET 1.0 running on Windows XP. This patch can be installed manually or by obtaining recent critical updates from the Windows Update site.

The symptom of this issue is that after installing the patch on a Windows XP machine, all requests to ASP.NET applications running on the local IIS 5.1 web server result in an error message saying "Server Application Unavailable". Requests to remote web servers are unaffected.

This issue only impacts installations running ASP.NET 1.0 on Windows XP. It does not impact machines running Windows 2000 or Windows Server 2003. It also does not impact machines running Windows XP with ASP.NET 1.1 installed.

Please note that this issue **is not** a security bug with ASP.NET. It **does not** open up or allow any malicious attacks against an ASP.NET application or server. Instead, it is purely a functional bug caused by the patch itself.

We are working hard on a permanent solution for this issue. In the meantime, you can execute the following batch file as a workaround for the issue. The batch file does the following:

1. Stops the IIS and ASP.NET state services
2. Deletes and recreates the ASPNET account with a known temporary password
3. Uses the Windows `runas` command to launch an executable that creates an ASPNET user profile
4. Re-registers ASP.NET. This creates a new random password for the account and applies default ASP.NET access control settings for it
5. Restarts the IIS service

The batch file contains a hardcoded temporary password of "**1pass@word**" which you will be prompted to enter for the runas command when the batch file is run. After the runas command completes, the ASPNET account password is recreated with a strong random value. Note that the batch file may fail if the hardcoded password does not meet the password complexity requirements in your environment. If that's the case, you can change it to another value that is appropriate for your environment.

> *[!IMPORTANT]* If you have added custom access control settings or database account permissions for the ASPNET account, they will need to be recreated after this batch file completes. This is because when the account is recreated, it will get a new security identifier (SID).

> *[!IMPORTANT]* If you are running the ASP.NET worker process with a custom account other than the ASPNET account, then you should not run this batch file. Instead, you should log in interactively or use the runas command with that account which will create a user profile for that account.

The batch file is included in the self-extracting archive below. To use it:

1. You must be running as an account with Administrator privileges

2. Download and open the self-extracting executable file
3. Extract the contents to c:\
4. Select Run... from the start menu, and enter `cmd.exe`
5. In the open command windows, type `c:\fixup.cmd`.
6. When prompted, enter **1pass@word** as the password.
7. If you have previously custom access control settings or database account permissions for the ASPNET account, you'll need to re-apply these settings now.

Many apologies for the inconvenience that this has caused. We'll post additional information as it becomes available.

The matrix below details platforms and versions impacted by this issue.

.NET FRAMEWORK	PLATFORM	AFFECTED
Version 1.0	Windows 2000 Professional	No
Version 1.0	Windows 2000 Server	No
Version 1.0	Windows XP Professional	Yes
Version 1.0	Windows Server 2003	No
Version 1.0	Windows XP Home with Cassini	No
Version 1.1	Windows 2000 Professional	No
Version 1.1	Windows 2000 Server	No
Version 1.1	Windows XP Professional	No
Version 1.1	Windows Server 2003	No
Version 1.1	Windows XP Home with Cassini	No

Thanks,
The ASP.NET Team

ASP.NET Side-by-Side Execution of .NET Framework 1.0 and 1.1

2/20/2020 • 5 minutes to read • Edit Online

This whitepaper describes how to install both .NET 1.0 and .NET 1.1 on your machine, allowing an ASP.NET Web application to run on either version of the framework.

Applies to ASP.NET 1.0 and ASP.NET 1.1.

In ASP.NET, applications are said to be running side by side when they are installed on the same computer, but use different versions of the .NET Framework. The following topic describes how to configure ASP.NET applications for side-by-side execution and provides detailed steps to:

- Maintain your Web application's mapping to .NET Framework version 1.0 during installation
- Map a Web application to a specific version of the .NET Framework
- Find the version of the .NET Framework that a Web site is using

Traditionally, when a component or application is updated on a computer, the older version is removed and replaced with the newer version. If the new version is not compatible with the previous version, this usually breaks other applications that use the component or application. The .NET Framework provides support for side-by-side execution, which allows multiple versions of an assembly or application to be installed on the same computer at the same time. Because multiple versions can be installed simultaneously, managed applications can select which version to use without affecting applications that use a different version.

By default, during the installation of the .NET Framework version 1.1, all existing ASP.NET applications are automatically reconfigured to use the latest version of the .NET Framework. If you do not want your ASP.NET applications to default to .NET Framework 1.1, click here to learn how to prevent this during installation.

If you update your Web server to .NET Framework 1.1 and want one or more Web applications to run .NET Framework 1.0, you need to update the Internet Information Services (IIS) Script Map. The script mapping is the mechanism to map the .aspx file extension for a specific Web application to a version of the .NET Framework. Click here to learn how to map a Web application to a specific version of the .NET Framework.

You can use the Internet Information Manger or the ASP.NET IIS Registration Tool (Aspnet_regiis.exe) to find which .NET Framework version is running a particular Web application. Click here to learn how to find the version of the .NET Framework that a Web site is using.

One import consideration when migrating to .NET Framework 1.1 is that each version of the .NET Framework uses its own Machine.config file. As a result, if a Web administrator has made changes to the Machine.config file, those changes must be migrated to the .NET Framework 1.1 Machine.config file.

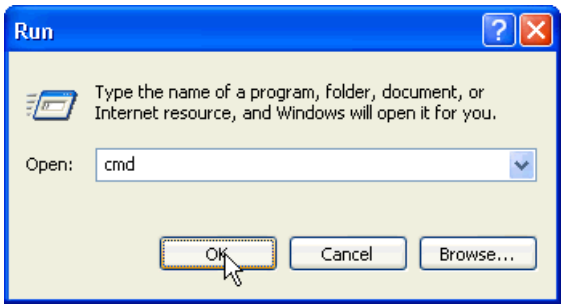
Maintaining your Web application's mapping to .NET Framework 1.0 during installation

By default, all existing ASP.NET applications are automatically reconfigured during installation to use the newer version of the .NET Framework. Using the newer version of the .NET Framework, applications can take full advantage of improvements and new features included in the new release. At the same time, the Web administrator, who might want granular control over which applications are updated, can prevent the automatic remapping of all existing ASP.NET applications during installation of the .NET Framework.

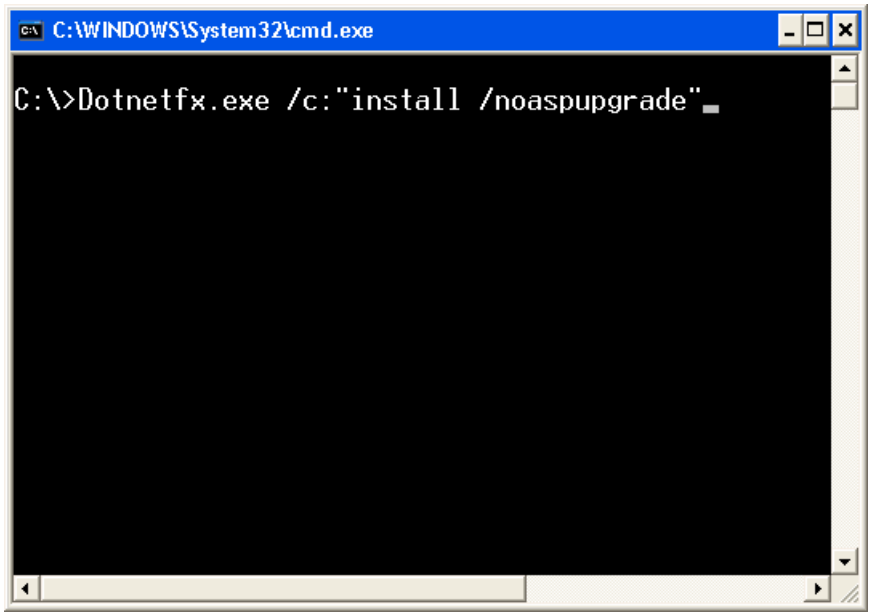
To prevent the automatic remapping of the entire ASP.NET application to the newer version of the .NET Framework, the Web administrator can use the /noaspupgrade command-line option with the Dotnetfx.exe setup program.

To prevent total remapping of ASP.NET application to newer version

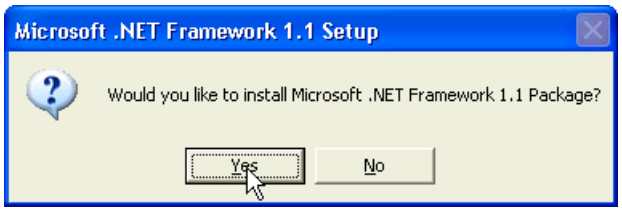
1. Go to **Start**.
2. Click on **run**.
3. Type **cmd**.
4. Click **OK**.



5. From the command prompt, type the following line to start the installation of the .NET Framework: **Dotnetfx.exe /c:"install /noaspupgrade?**.



6. Click **Yes** in the Microsoft .NET Framework 1.1 Setup. This will start the setup process of the .NET Framework 1.1.



Map a Web application to a specific version of the .NET Framework

Each version of the .NET Framework includes a version of the ASP.NET IIS Registration Tool (Aspnet_regiis.exe). This tool enables administrators to specify that a Web application be run under a particular version of the .NET Framework. This is referred to as mapping a Web application to a version of the .NET Framework. Administrators must select the Aspnet_regiis.exe that corresponds to the version of the .NET Framework that will be associated

with the Web application. For example, an administrator who wants to specify that a Web site use .NET Framework 1.1 must use the Aspnet_regiis.exe that comes with .NET Framework 1.1.

The Aspnet_regiis.exe for version 1.0 is located at:

- C:\WINDOWS\Microsoft.NET\Framework**v1.0.3705**\aspnet_regiis

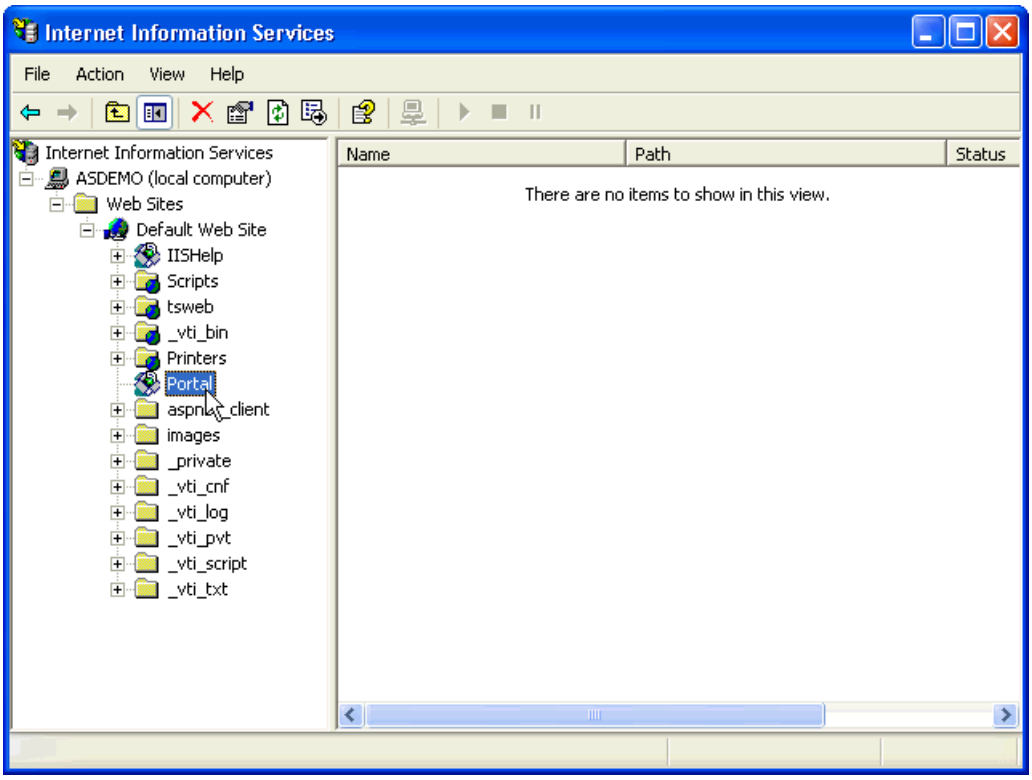
The Aspnet_regiis.exe for version 1,1 is located at:

- C:\WINDOWS\Microsoft.NET\Framework**v1.1.4322**\aspnet_regiis

The Aspnet_regiis.exe provides two options for script mapping a Web application:

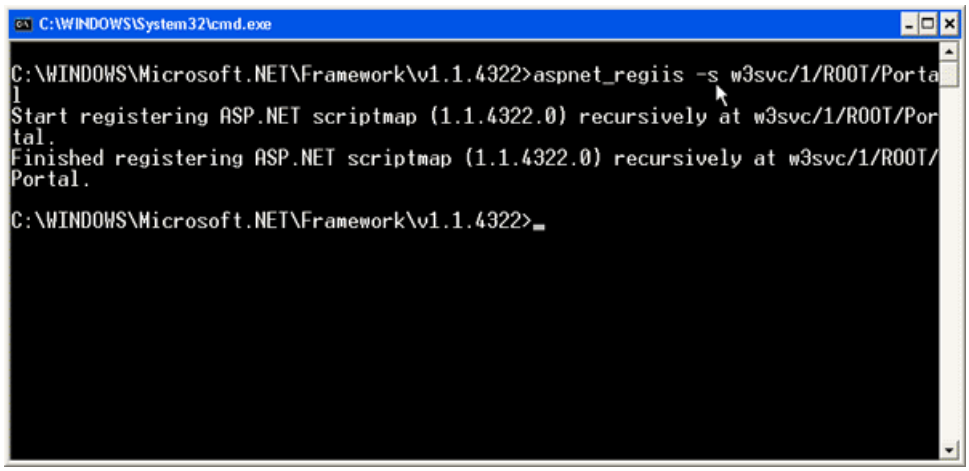
- **-s** sets the script map in the path and in its child directories.
- **-sn** sets the script map in the path only.

The path defines the Web application IIS metadata path, which is defined in the form of W3SVC/ROOT/{WebSiteNumber}/{Application_Name}. For example, for a Web application called Portal located under the default Web site, the metabase path is W3SVC/1/ROOT/Portal.

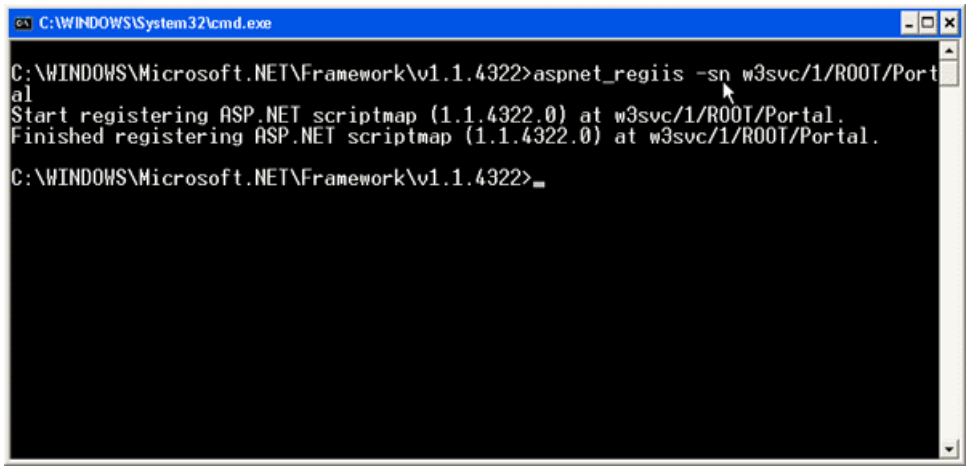


Note You can also use a tool called the Metabase Editor to get the metabase path. You can download this tool from the Microsoft Support site at https://support.microsoft.com/default.aspx?scid=kb;en-us;232068.

- Run Aspnet_regiis.exe -s W3SVC/1/ROOT/Portal to update the portal IIS script map and its subapplication.



- Run Aspnet_regiis.exe -sn W3SVC/1/ROOT/Portal to update the portal IIS script map, without affecting applications in the portal?s subdirectories.

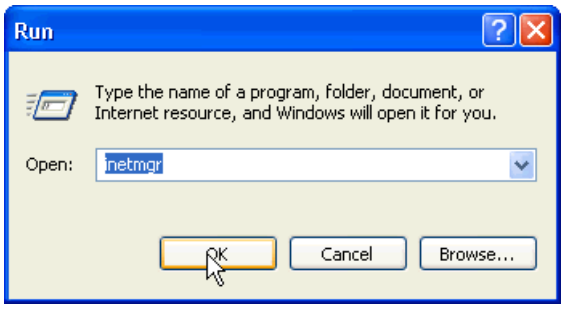


Find the .NET Framework version that a Web application is using

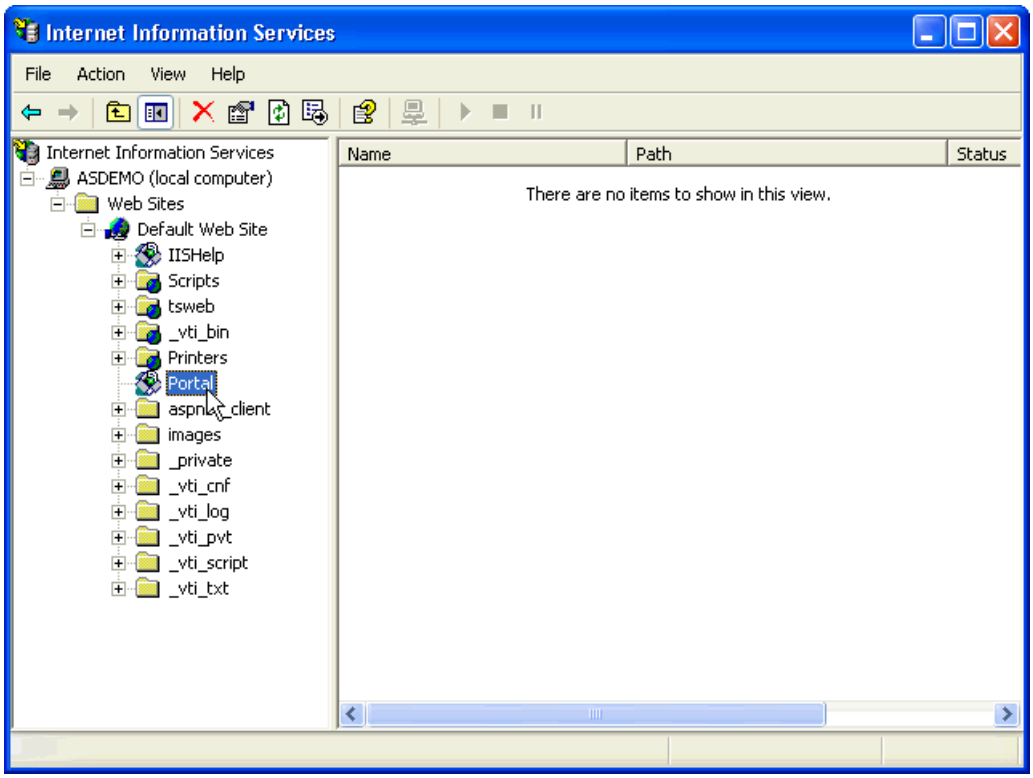
An administrator can use the Internet Service Manager to find which version of the .NET Framework runs a Web site. Different operating system versions launch the Internet Service Manager differently. To start the service manager, follow the steps listed below.

To start Internet Service Manager

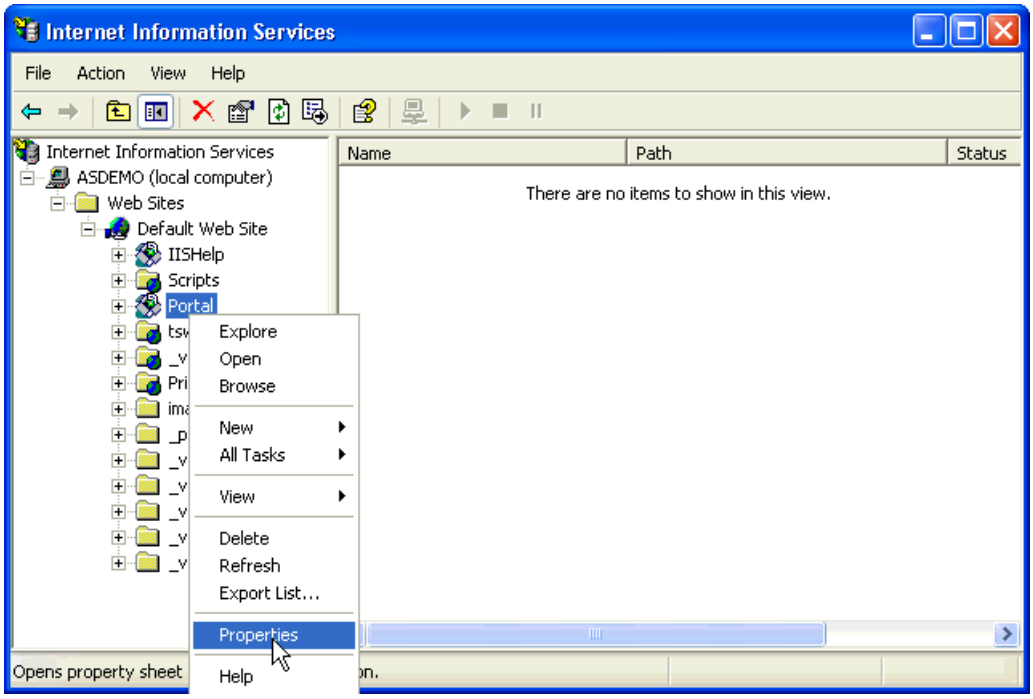
1. Go to **Start**.
2. Click on **run**.
3. Type **inetmgr**.



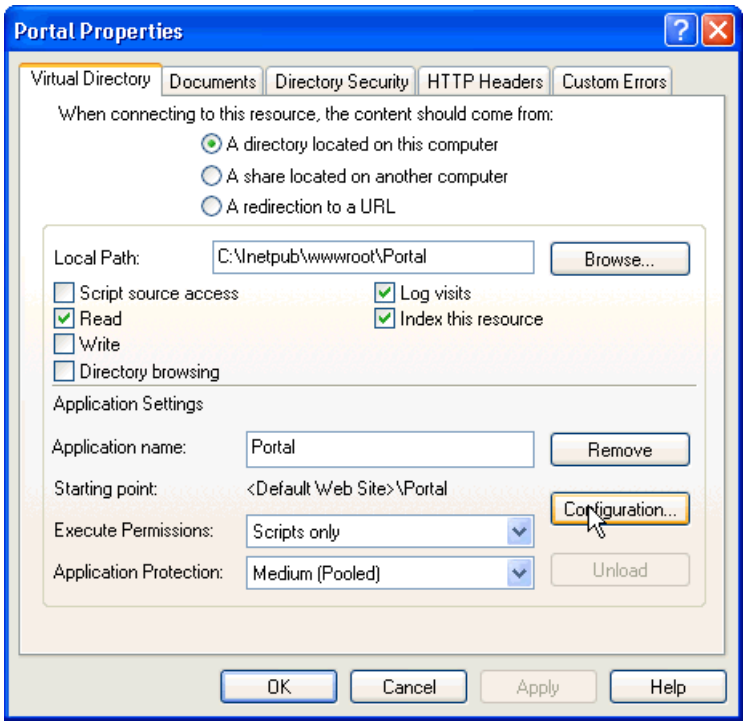
4. From the Internet Service Manager, select the Web application whose version of the .NET Framework you want to know.



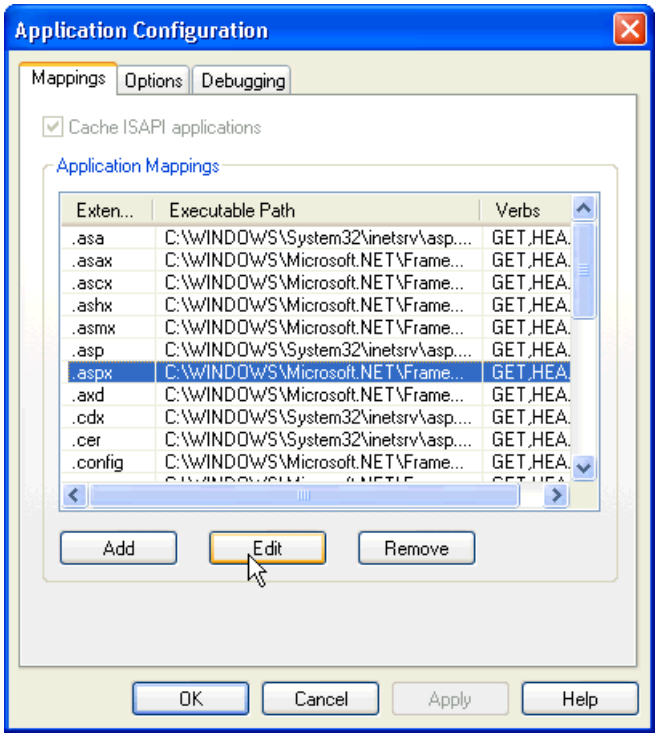
5. Right-click on the Web application, and click on **Properties**.



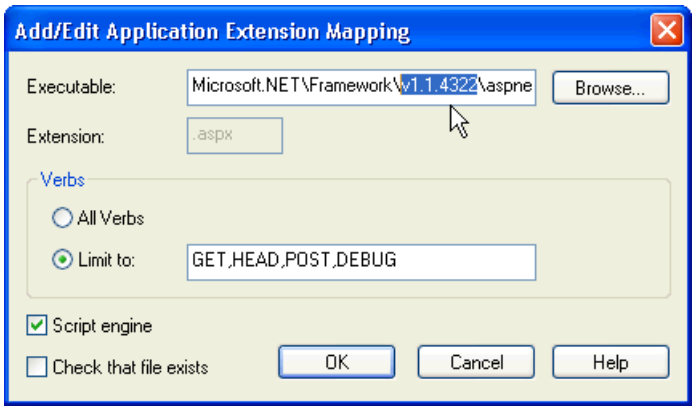
6. From the Property window, select **Configuration**.



7. From the application mapping table, select **.aspx**, and click **Edit**.



8. From the **Executable** text box, look at the version directory by scrolling. If the version directory is v.1.1.4322, the application is mapped to .NET Framework 1.1. Conversely, if the version directory is v1.0.3705, the application is mapped to .NET Framework 1.0.


Add/Edit Application Extension Mapping
Executable:
Microsoft.NET\Framework\v1.1.4322\aspne
Browse...
Extension:
.aspx
Verbs
All Verbs
Limit to:
GET,HEAD,POST,DEBUG
Script engine
Check that file exists
OK
Cancel
Help

ASP.NET MVC 4

2/20/2020 • 19 minutes to read • Edit Online

This document describes the release of ASP.NET MVC 4 .

- Installation Notes
- Documentation
- Support
- Software Requirements
- New Features in ASP.NET MVC 4
  - ASP.NET Web API
  - Enhancements to Default Project Templates
  - Mobile Project Template
  - Display Modes
  - jQuery Mobile, the View Switcher, and Browser Overriding
  - Task Support for Asynchronous Controllers
  - Azure SDK
  - Database Migrations
  - Empty Project Template
  - Add Controller to any project folder
  - Bundling and Minification
  - Enabling Logins from Facebook and Other Sites Using OAuth and OpenID
- Upgrading an ASP.NET MVC 3 Project to ASP.NET MVC 4
- Changes from ASP.NET MVC 4 Release Candidate
- Known Issues and Breaking Changes

Installation Notes

ASP.NET MVC 4 for Visual Studio 2010 can be installed from the ASP.NET MVC 4 home page using the Web Platform Installer.

We recommend uninstalling any previously installed previews of ASP.NET MVC 4 prior to installing ASP.NET MVC 4. You can upgrade the ASP.NET MVC 4 Beta and Release Candidate to ASP.NET MVC 4 without uninstalling.

This release is not compatible with any preview releases of .NET Framework 4.5. You must separately upgrade the any installed preview releases of .NET Framework 4.5 to the final version prior to installing ASP.NET MVC 4.

ASP.NET MVC 4 can be installed and run side-by-side with ASP.NET MVC 3.

Documentation

Documentation for ASP.NET MVC is available on the MSDN website at the following URL:

https://go.microsoft.com/fwlink/?LinkID=243043

Tutorials and other information about ASP.NET MVC are available on the MVC 4 page of the ASP.NET website (https://www.asp.net/mvc/mvc4).

Support

ASP.NET MVC 4 is fully supported. If you have questions about working with this release you can also post them to the ASP.NET MVC forum (https://forums.asp.net/1146.aspx), where members of the ASP.NET community are frequently able to provide informal support.

Software Requirements

The ASP.NET MVC 4 components for Visual Studio require PowerShell 2.0 and either Visual Studio 2010 with Service Pack 1 or Visual Web Developer Express 2010 with Service Pack 1.

New Features in ASP.NET MVC 4

This section describes features that have been introduced in the ASP.NET MVC 4 release.

ASP.NET Web API

ASP.NET MVC 4 includes ASP.NET Web API, a new framework for creating HTTP services that can reach a broad range of clients including browsers and mobile devices. ASP.NET Web API is also an ideal platform for building RESTful services.

ASP.NET Web API includes support for the following features:

- **Modern HTTP programming model:** Directly access and manipulate HTTP requests and responses in your Web APIs using a new, strongly typed HTTP object model. The same programming model and HTTP pipeline is symmetrically available on the client through the new *HttpClient* type.
- **Full support for routes:** ASP.NET Web API supports the full set of route capabilities of ASP.NET Routing, including route parameters and constraints. Additionally, use simple conventions to map actions to HTTP methods.
- **Content negotiation:** The client and server can work together to determine the right format for data being returned from a web API. ASP.NET Web API provides default support for XML, JSON, and Form URL-encoded formats and you can extend this support by adding your own formatters, or even replace the default content negotiation strategy.
- **Model binding and validation:** Model binders provide an easy way to extract data from various parts of an HTTP request and convert those message parts into .NET objects which can be used by the Web API actions. Validation is also performed on action parameters based on data annotations.
- **Filters:** ASP.NET Web API supports filters including well-known filters such as the *[Authorize]* attribute. You can author and plug in your own filters for actions, authorization and exception handling.
- **Query composition:** Use the *[Queryable]* filter attribute on an action that returns *IQueryable* to enable support for querying your web API via the OData query conventions.
- **Improved testability:** Rather than setting HTTP details in static context objects, web API actions work with instances of *HttpRequestMessage* and *HttpResponseMessage*. Create a unit test project along with your Web API project to get started quickly writing unit tests for your Web API functionality.
- **Code-based configuration:** ASP.NET Web API configuration is accomplished solely through code, leaving your config files clean. Use the provided service locator pattern to configure extensibility points.
- **Improved support for Inversion of Control (IoC) containers:** ASP.NET Web API provides great support for IoC containers through an improved dependency resolver abstraction
- **Self-host:** Web APIs can be hosted in your own process in addition to IIS while still using the full power of routes and other features of Web API.
- **Create custom help and test pages:** You now can easily build custom help and test pages for your web

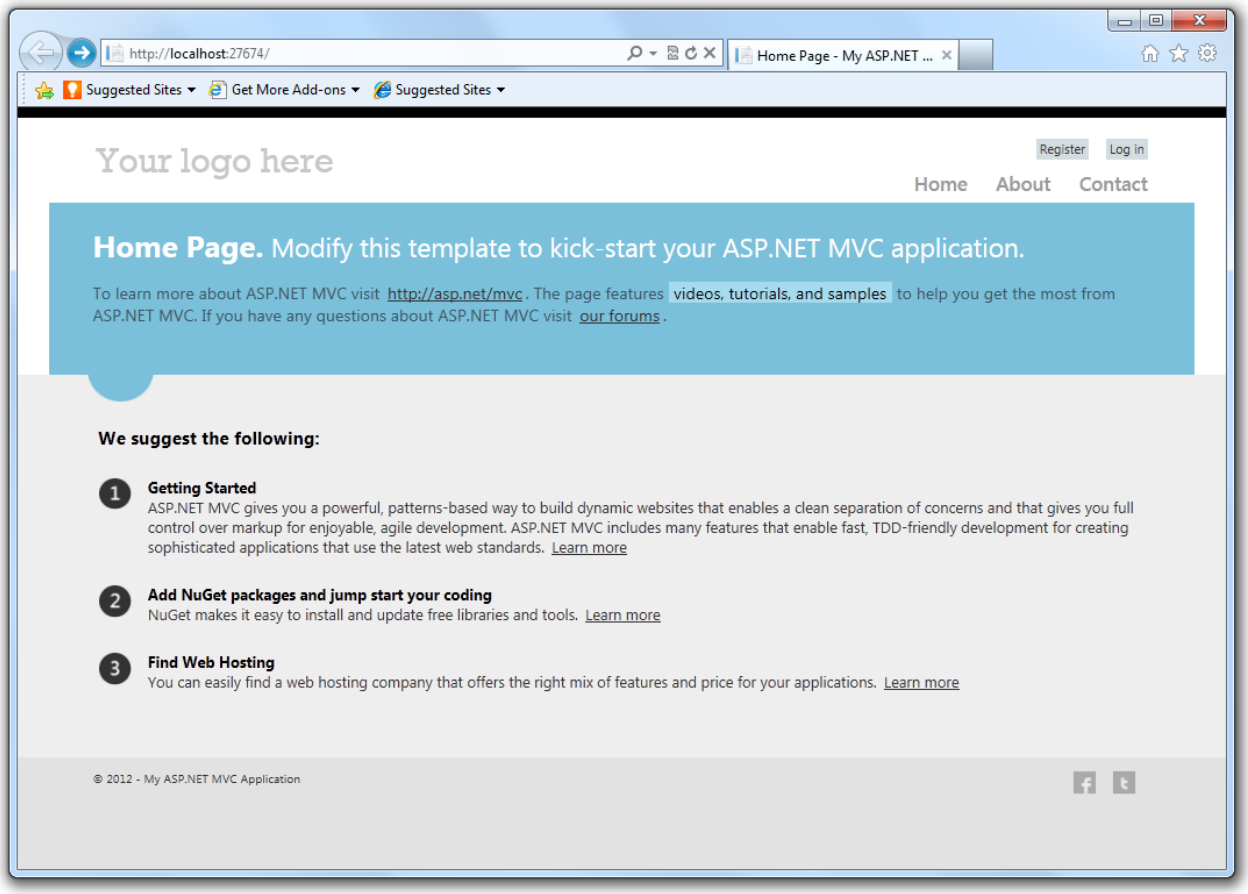
APIs by using the new *IApiExplorer* service to get a complete runtime description of your web APIs.

- **Monitoring and diagnostics:** ASP.NET Web API now provides light weight tracing infrastructure that makes it easy to integrate with existing logging solutions such as System.Diagnostics, ETW and third party logging frameworks. You can enable tracing by providing an *ITraceWriter* implementation and adding it to your web API configuration.
- **Link generation:** Use the ASP.NET Web API *UrlHelper* to generate links to related resources in the same application.
- **Web API project template:** Select the new Web API project form the New MVC 4 Project wizard to quickly get up and running with ASP.NET Web API.
- **Scaffolding:** Use the **Add Controller** dialog to quickly scaffold a web API controller based on an Entity Framework based model type.

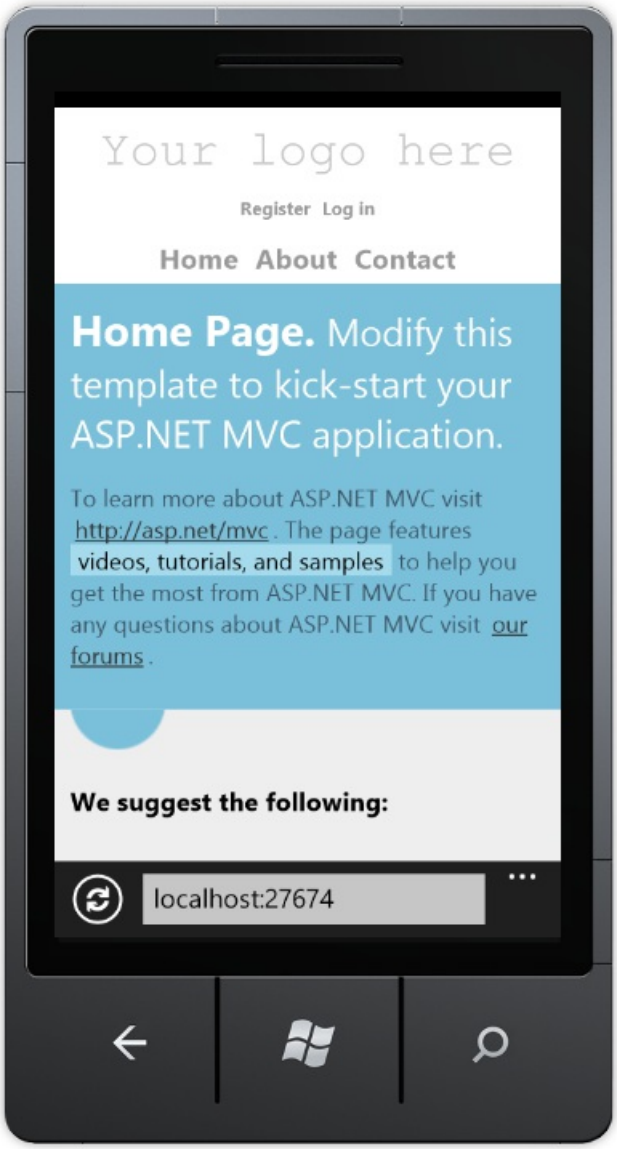
For more details on ASP.NET Web API please visit https://www.asp.net/web-api.

Enhancements to Default Project Templates

The template that is used to create new ASP.NET MVC 4 projects has been updated to create a more modern-looking website:



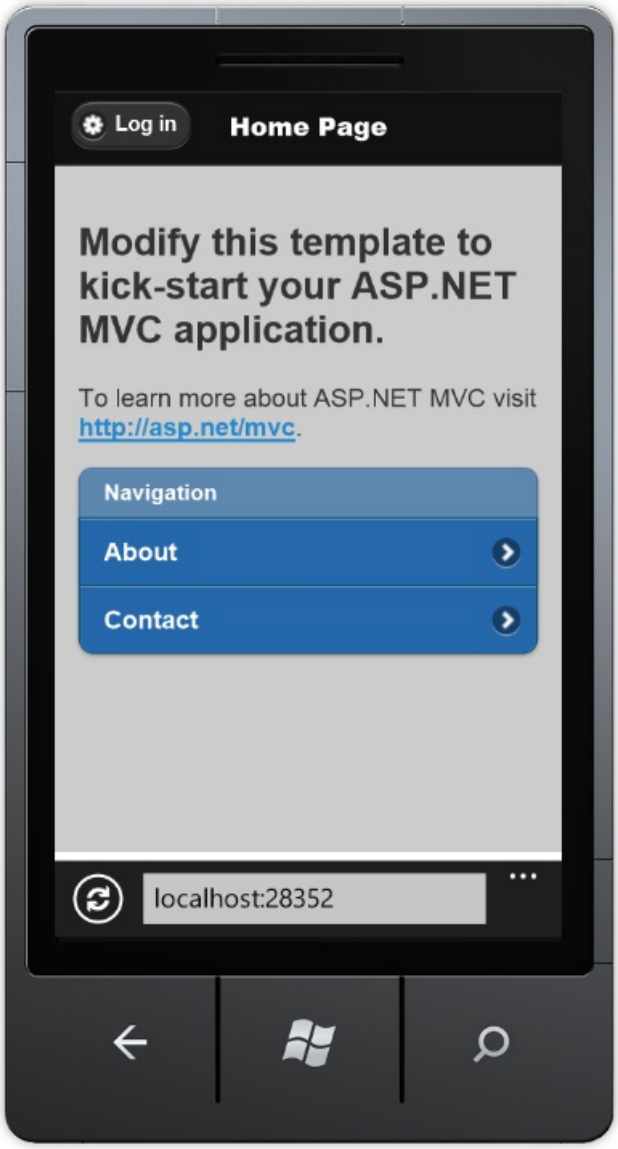
In addition to cosmetic improvements, there's improved functionality in the new template. The template employs a technique called adaptive rendering to look good in both desktop browsers and mobile browsers without any customization.



To see adaptive rendering in action, you can use a mobile emulator or just try resizing the desktop browser window to be smaller. When the browser window gets small enough, the layout of the page will change.

Mobile Project Template

If you're starting a new project and want to create a site specifically for mobile and tablet browsers, you can use the new Mobile Application project template. This is based on jQuery Mobile, an open-source library for building touch-optimized UI:



This template contains the same application structure as the Internet Application template (and the controller code is virtually identical), but it's styled using jQuery Mobile to look good and behave well on touch-based mobile devices. To learn more about how to structure and style mobile UI, see the jQuery Mobile project website.

If you already have a desktop-oriented site that you want to add mobile-optimized views to, or if you want to create a single site that serves differently styled views to desktop and mobile browsers, you can use the new Display Modes feature. (See the next section.)

Display Modes

The new Display Modes feature lets an application select views depending on the browser that's making the request. For example, if a desktop browser requests the Home page, the application might use the Views\Home\Index.cshtml template. If a mobile browser requests the Home page, the application might return the Views\Home\Index.mobile.cshtml template.

Layouts and partials can also be overridden for particular browser types. For example:

- If your Views\Shared folder contains both the _Layout.cshtml and _Layout.mobile.cshtml templates, by default the application will use _Layout.mobile.cshtml during requests from mobile browsers and _Layout.cshtml during other requests.
- If a folder contains both _MyPartial.cshtml and _MyPartial.mobile.cshtml, the instruction @Html.Partial("_MyPartial") will render _MyPartial.mobile.cshtml during requests from mobile browsers, and _MyPartial.cshtml during other requests.

If you want to create more specific views, layouts, or partial views for other devices, you can register a new

DefaultDisplayMode instance to specify which name to search for when a request satisfies particular conditions. For example, you could add the following code to the *Application_Start* method in the Global.asax file to register the string "iPhone" as a display mode that applies when the Apple iPhone browser makes a request:

```
DisplayModeProvider.Instance.Modes.Insert(0, new
DefaultDisplayMode("iPhone")
{
    ContextCondition = (context => context.GetOverriddenUserAgent().IndexOf
        ("iPhone", StringComparison.OrdinalIgnoreCase) >= 0)
});
```

After this code runs, when an Apple iPhone browser makes a request, your application will use the Views\Shared_Layout.iPhone.cshtml layout (if it exists). For more information on Display Mode, see ASP.NET MVC 4 Mobile Features. Applications using DisplayModeProvider should install the Fixed DisplayModes NuGet package. The ASP.NET Fall 2012 Update includes the Fixed DisplayModes NuGet package in the new project templates. See ASP.NET MVC 4 Mobile Caching Bug Fixedd for details on the fix.

jQuery Mobile and Mobile Features

For information on building Mobile applications with ASP.NET MVC 4 using jQuery Mobile, see the tutorial ASP.NET MVC 4 Mobile Features.

Task Support for Asynchronous Controllers

You can now write asynchronous action methods as single methods that return an object of type *Task* or *Task<ActionResult>*.

For more information see Using Asynchronous Methods in ASP.NET MVC 4.

Azure SDK

ASP.NET MVC 4 supports the 1.6 and newer releases of the Windows Azure SDK.

Database Migrations

ASP.NET MVC 4 projects now include Entity Framework 5. One of the great features in Entity Framework 5 is support for database migrations. This feature enables you to easily evolve your database schema using a code-focused migration while preserving the data in the database. For more information on database migrations, see Adding a New Field to the Movie Model and Table in the Introduction to ASP.NET MVC 4 tutorial.

Empty Project Template

The MVC Empty project template is now truly empty so that you can start from a completely clean slate. The earlier version of the Empty project template has been renamed to Basic.

Add Controller to any project folder

You can now right click and select **Add Controller** from any folder in your MVC project. This gives you more flexibility to organize your controllers however you want, including keeping your MVC and Web API controllers in separate folders.

Bundling and Minification

The bundling and minification framework enables you to reduce the number of HTTP requests that a Web page needs to make by combining individual files into a single, bundled file for scripts and CSS. It can then reduce the overall size of those requests by minifying the contents of the bundle. Minifying can include activities like eliminating whitespace to shortening variable names to even collapsing CSS selectors based on their semantics. Bundles are declared and configured in code and are easily referenced in views via helper methods which can generate either a single link to the bundle or, when debugging, multiple links to the individual contents of the bundle. For more information see Bundling and Minification.

Enabling Logins from Facebook and Other Sites Using OAuth and OpenID

The default templates in ASP.NET MVC 4 Internet Project template now includes support for OAuth and OpenID login using the DotNetOpenAuth library. For information on configuring an OAuth or OpenID provider, see OAuth/OpenID Support for WebForms, MVC and WebPages and the OAuth and OpenID feature documentation in ASP.NET Web Pages.

Upgrading an ASP.NET MVC 3 Project to ASP.NET MVC 4

ASP.NET MVC 4 can be installed side by side with ASP.NET MVC 3 on the same computer, which gives you flexibility in choosing when to upgrade an ASP.NET MVC 3 application to ASP.NET MVC 4.

The simplest way to upgrade is to create a new ASP.NET MVC 4 project and copy all the views, controllers, code, and content files from the existing MVC 3 project to the new project and then to update the assembly references in the new project to match any non-MVC template included assembiles you are using. If you have made changes to the Web.config file in the MVC 3 project, you must also merge those changes into the Web.config file in the MVC 4 project.

To manually upgrade an existing ASP.NET MVC 3 application to version 4, do the following:

1. In all Web.config files in the project (there is one in the root of the project, one in the Views folder, and one in the Views folder for each area in your project), replace every instance of the following text (note: System.Web.WebPages, Version=1.0.0.0 is not found in projects created with Visual Studio 2012):

   ```
   System.Web.Mvc, Version=3.0.0.0
   System.Web.WebPages, Version=1.0.0.0
   System.Web.Helpers, Version=1.0.0.0
   System.Web.WebPages.Razor, Version=1.0.0.0
   ```

   with the following corresponding text:

   ```
   System.Web.Mvc, Version=4.0.0.0
   System.Web.WebPages, Version=2.0.0.0
   System.Web.Helpers, Version=2.0.0.0
   System.Web.WebPages.Razor, Version=2.0.0.0
   ```

2. In the root Web.config file, update the *webPages:Version* element to "2.0.0.0" and add a new *PreserveLoginUrl* key that has the value "true":

   ```
   <appSettings>
     <add key="webpages:Version" value="2.0.0.0" />
     <add key="PreserveLoginUrl" value="true" />
   </appSettings>
   ```

3. In Solution Explorer, right-click on the References and select Manage NuGet Packages. In the left pane, select **Online\NuGet official package source**, then update the following:

   - ASP.NET MVC 4
   - (Optional) jQuery, jQuery Validation and jQuery UI
   - (Optional) Entity Framework
   - (Optonal) Modernizr

4. In Solution Explorer, right-click the project name and then select Unload Project. Then right-click the name again and select Edit *ProjectName*.csproj.

5. Locate the *ProjectTypeGuids* element and replace {E53F8FEA-EAE0-44A6-8774-FFD645390401} with {E3E379DF-F4C6-4180-9B81-6769533ABE47}.

6. Save the changes, close the project (.csproj) file you were editing, right-click the project, and then select Reload Project.

7. If the project references any third-party libraries that are compiled using previous versions of ASP.NET MVC, open the root Web.config file and add the following three *bindingRedirect* elements under the *configuration* section:

```
<configuration>
  <!--... elements deleted for clarity ...-->

  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="System.Web.Helpers"
             publicKeyToken="31bf3856ad364e35" />
        <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="System.Web.Mvc"
             publicKeyToken="31bf3856ad364e35" />
        <bindingRedirect oldVersion="1.0.0.0-3.0.0.0" newVersion="4.0.0.0"/>
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="System.Web.WebPages"
             publicKeyToken="31bf3856ad364e35" />
        <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Changes from ASP.NET MVC 4 Release Candidate

The release notes for ASP.NET MVC 4 Release Candidate can be found here:

The major changes from ASP.NET MVC 4 Release Candidate in this release are summarized below:

- **Per controller configuration:** ASP.NET Web API controllers can be attributed with a custom attribute that implements *IControllerConfiguration* to setup their own formatters, action selector and parameter binders. The *HttpControllerConfigurationAttribute* has been removed.
- **Per route message handlers:** You can now specify the final message handler in the request chain for a given route. This enables support for ride-along frameworks to use routing to dispatch to their own (non-*IHttpController*) endpoints.
- **Progress notifications:** The *ProgressMessageHandler* generates progress notification for both request entities being uploaded and response entities being downloaded. Using this handler it is possible to keep track of how far you are uploading a request body or downloading a response body.
- **Push content:** The *PushStreamContent* class enables scenarios where a data producer wants to write directly to the request or response(either synchronously or asynchronously) using a stream. When the *PushStreamContent* is ready to accept data it calls out to an action delegate with the output stream. The developer can then write to the stream for as long as necessary and close the stream when writing has completed. The *PushStreamContent* detects the closing of the stream and completes the underlying asynchronous *Task* for writing out the content.
- **Creating error responses:** Use the *HttpError* type to consistently represent error information from such as validation errors and exceptions while still honoring the *IncludeErrorDetailPolicy*. Use the new *CreateErrorResponse* extension methods to easily create error responses with *HttpError* as content. The *HttpError* content is fully content negotiated.
- **MediaRangeMapping removed:** Media type ranges are now handled by the default content negotiator.

- **Default parameter binding for simple type parameters is now [FromUri]:** In previous releases of ASP.NET Web API the default parameter binding for simple type parameters used model binding. The default parameter binding for simple type parameters is now *[FromUri]*.
- **Action selection honors required parameters:** Action selection in ASP.NET Web API will now only select an action if all required parameters that come from the URI are provided. A parameter can be specified as optional by providing a default value for the argument in the action method signature.
- **Customize HTTP parameter bindings:** Use the *ParameterBindingAttribute* to customize the parameter binding for a specific action parameter or use the *ParameterBindingRules* on the *HttpConfiguration* to customize parameter bindings more broadly.
- **MediaTypeFormatter improvements:** Formatters now have access to the full *HttpContent* instance.
- **Host buffering policy selection:** Implement and configure the *IHostBufferPolicySelector* service in ASP.NET Web API to enable hosts to determine the policy for when buffering is to be used.
- **Access client certificates in a host agnostic manner:** Use the *GetClientCertificate* extension method to get the supplied client certificate from the request message.
- **Content negotiation extensibility:** Customize content negotiation by deriving from the *DefaultContentNegotiator* and overriding any aspect of content negotiation that you would like.
- **Support for returning 406 Not Acceptable responses:** You can now return 406 Not Acceptable responses in ASP.NET Web API when a suitable formatter is not found by creating a *DefaultContentNegotiator* with the *excludeMatchOnTypeOnly* parameter set to *true*.
- **Read form data as NameValueCollection or JToken:** You can read form data in the URI query string or in the request body as a *NameValueCollection* using the *ParseQueryString* and *ReadAsFormDataAsync* extension methods respectively. Similarly, you can read form data in the URI query string or in the request body as a *JToken* using the *TryReadQueryAsJson* and *ReadAsAsync*<T> extension methods respectively.
- **Multipart improvements:** It is now possible to write a *MultipartStreamProvider* that is completely tailored to the type of MIME multipart data that it can read and present the result in the optimal way to the user. You can also hook a post processing step on the *MultipartStreamProvider* that allows the implementation to do whatever post processing it wants on the MIME multipart body parts. For example, the *MultipartFormDataStreamProvider* implementation reads the HTML form data parts and adds them to a *NameValueCollection* so they are easy to get at from the caller.
- **Link generation improvements:** The *UrlHelper* no longer depends on *HttpControllerContext*. You can now access the *UrlHelper* from any context where the *HttpRequestMessage* is available.
- **Message handler execution order change:** Message handlers are now executed in the order that they are configured instead of in reverse order.
- **Helper for wiring up message handlers:** The new *HttpClientFactory* that can wire up *DelegatingHandlers* and create an *HttpClient* with the desired pipeline ready to go. It also provides functionality for wiring up with alternative inner handlers (the default is *HttpClientHandler*) as well as do the wiring up when using *HttpMessageInvoker* or another *DelegatingHandler* instead of *HttpClient* as the top-invoker.
- **Support for CDNs in ASP.NET Web Optimization:** ASP.NET Web Optimization now provides support for CDN alternate paths enabling you to specify for each bundle an additional URL which points to that same resource on a content delivery network. Supporting CDNs enables you to get your script and style bundles geographically closer to the end consumers of your Web applications. Production apps should implement a fallback when the CDN is unavailable. Test the fallback.
- **ASP.NET Web API routes and configuration moved to *WebApiConfig.Register* static method that can be resused in test code.** ASP.NET Web API routes previously were added in *RouteConfig.RegisterRoutes* along with the standard MVC routes. The default ASP.NET Web API routes and configuration are now handled in a separate *WebApiConfig.Register* method to facilitate testing.

Known Issues and Breaking Changes

- **The RC and RTM version of ASP.NET MVC 4 incorrectly returned cached desktop views when**

mobile views should be returned.

  - See ASP.NET MVC 4 Mobile Caching Bug Fixed for details on the fix. The fix can be installed from the Fixed DisplayModes NuGet package.

- **Breaking changes in the Razor View Engine**. The following types were removed from *System.Web.Mvc.Razor*:

  - *ModelSpan*
  - *MvcVBRazorCodeGenerator*
  - *MvcCSharpRazorCodeGenerator*
  - *MvcVBRazorCodeParser*

  The following methods were also removed:

  - *MvcCSharpRazorCodeParser.ParseInheritsStatement(System.Web.Razor.Parser.CodeBlockInfo)*
  - *MvcWebPageRazorHost.DecorateCodeGenerator(System.Web.Razor.Generator.RazorCodeGenerator)*
  - *MvcVBRazorCodeParser.ParseInheritsStatement(System.Web.Razor.Parser.CodeBlockInfo)*

- **When WebMatrix.WebData.dll is included in the /bin directory of an ASP.NET MVC 4 apps, it takes over the URL for forms authentication.** Adding the WebMatrix.WebData.dll assembly to your application (for example, by selecting "ASP.NET Web Pages with Razor Syntax" when using the Add Deployable Dependencies dialog) will override the authentication login redirect to /account/logon rather than /account/login as expected by the default ASP.NET MVC Account Controller. To prevent this behavior and use the URL specified already in the authentication section of web.config, you can add an appSetting called PreserveLoginUrl and set it to true:

```
<appSettings>
    <add key="PreserveLoginUrl" value="true"/>
</appSettings>
```

- **The NuGet package manager fails to install when attempting to install ASP.NET MVC 4 for side by side installations of Visual Studio 2010 and Visual Web Developer 2010.** To run Visual Studio 2010 and Visual Web Developer 2010 side by side with ASP.NET MVC 4 you must install ASP.NET MVC 4 after both versions of Visual Studio have already been installed.

- **Uninstalling ASP.NET MVC 4 fails if prerequisites have already been uninstalled.** To cleanly uninstall ASP.NET MVC 4you must uninstall ASP.NET MVC 4 prior to uninstalling Visual Studio.

- **Installing ASP.NET MVC 4 breaks ASP.NET MVC 3 RTM applications.** ASP.NET MVC 3 applications that were created with the RTM release (not with the ASP.NET MVC 3 Tools Update release) require the following changes in order to work side-by-side with ASP.NET MVC 4. Building the project without making these updates results in compilation errors.

  Required updates

  1. In the root Web.config file, add a new *<appSettings>* entry with the key *webPages:Version* and the value *1.0.0.0*.

     ```
     <appSettings>
         <add key="webpages:Version" value="1.0.0.0"/>
         <add key="ClientValidationEnabled" value="true"/>
         <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
     </appSettings>
     ```

  2. In Solution Explorer, right-click the project name and then select Unload Project. Then right-click the name again and select Edit *ProjectName*.csproj.

3. Locate the following assembly references:

```
<Reference Include="System.Web.WebPages"/>
<Reference Include="System.Web.Helpers" />
```

   Replace them with the following:

```
<Reference Include="System.Web.WebPages, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL "/>
<Reference Include="System.Web.Helpers, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35, processorArchitecture=MSIL" />
```

4. Save the changes, close the project (.csproj) file you were editing, and then right-click the project and select Reload.

- **Changing an ASP.NET MVC 4 project to target 4.0 from 4.5 does not update the EntityFramework assembly reference:** If you change an ASP.NET MVC 4 project to target 4.0 after targetting 4.5 the reference to the EntityFramework assembly will still point to the 4.5 version. To fix this issue uninstall and reinstall the EntityFramework NuGet package.
- **403 Forbidden when running an ASP.NET MVC 4 application on Azure after changing to target 4.0 from 4.5:** If you change an ASP.NET MVC 4 project to target 4.0 after targetting 4.5 and then deploy to Azure you may see a 403 Forbidden error at runtime. To workaround this issue add the following to your web.config: `<modules runAllManagedModulesForAllRequests="true" />`
- **Visual Studio 2012 crashes when you type a '' in a string literal in a Razor file.** To work around the issue enter the closing quote of the string literal first.
- **Browsing to "Account/Manage" in the Internet template results in a runtime error for CHS, TRK and CHT languages.** To fix the issue modify the page to separate out *@User.Identity.Name* by puting it as the only content within the ** tag.
- **Google and LinkedIn providers are not supported within Azure Web Sites.** Use alternative authentication providers when deploying to Azure Web Sites.
- **When using UriPathExtensionMapping with IIS 8 Express/IIS, you would receive 404 Not Found errors when you try to use the extension.** The static file handler will interfere with requests to web APIs that use *UriPathExtensionMappings*. Set *runAllManagedModulesForAllRequests=true* in web.config to work around the issue.
- **Controller.Execute method is no longer called.** All MVC controllers are now always executed asynchronously.

Request Validation - Preventing Script Attacks

2/20/2020 • 3 minutes to read • Edit Online

This paper describes the request validation feature of ASP.NET where, by default, the application is prevented from processing unencoded HTML content submitted to the server. This request validation feature can be disabled when the application has been designed to safely process HTML data.

Applies to ASP.NET 1.1 and ASP.NET 2.0.

Request validation, a feature of ASP.NET since version 1.1, prevents the server from accepting content containing un-encoded HTML. This feature is designed to help prevent some script-injection attacks whereby client script code or HTML can be unknowingly submitted to a server, stored, and then presented to other users. We still strongly recommend that you validate all input data and HTML encode it when appropriate.

For example, you create a Web page that requests a user's email address and then stores that email address in a database. If the user enters <SCRIPT>alert("hello from script")</SCRIPT> instead of a valid email address, when that data is presented, this script can be executed if the content was not properly encoded. The request validation feature of ASP.NET prevents this from happening.

Why this feature is useful

Many sites are not aware that they are open to simple script injection attacks. Whether the purpose of these attacks is to deface the site by displaying HTML, or to potentially execute client script to redirect the user to a hacker's site, script injection attacks are a problem that Web developers must contend with.

Script injection attacks are a concern of all web developers, whether they are using ASP.NET, ASP, or other web development technologies.

The ASP.NET request validation feature proactively prevents these attacks by not allowing unencoded HTML content to be processed by the server unless the developer decides to allow that content.

What to expect: Error Page

The screen shot below shows some sample ASP.NET code:

```
<script runat="server">

  Public Sub Button_Click(sender As Object, e As EventArgs)

    ' Set the Label to the value of the textbox
    Label1.Text = TextBox1.Text

  End Sub

</script>

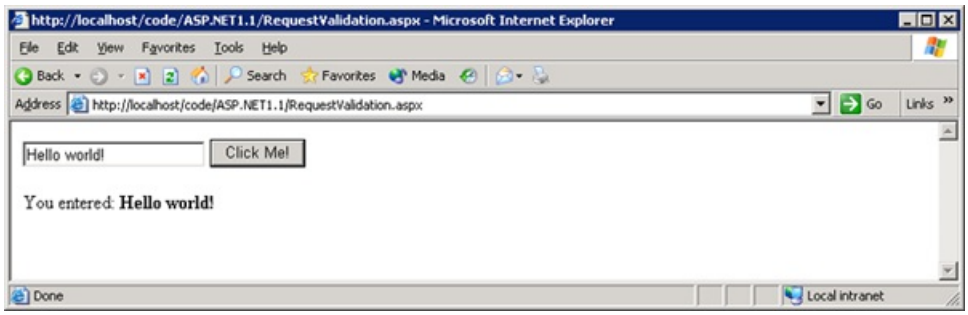
<form runat="server">

  <asp:TextBox id="TextBox1" runat="server" />
  <asp:Button OnClick="Button_Click" runat="server" Text="Click Me!"/>

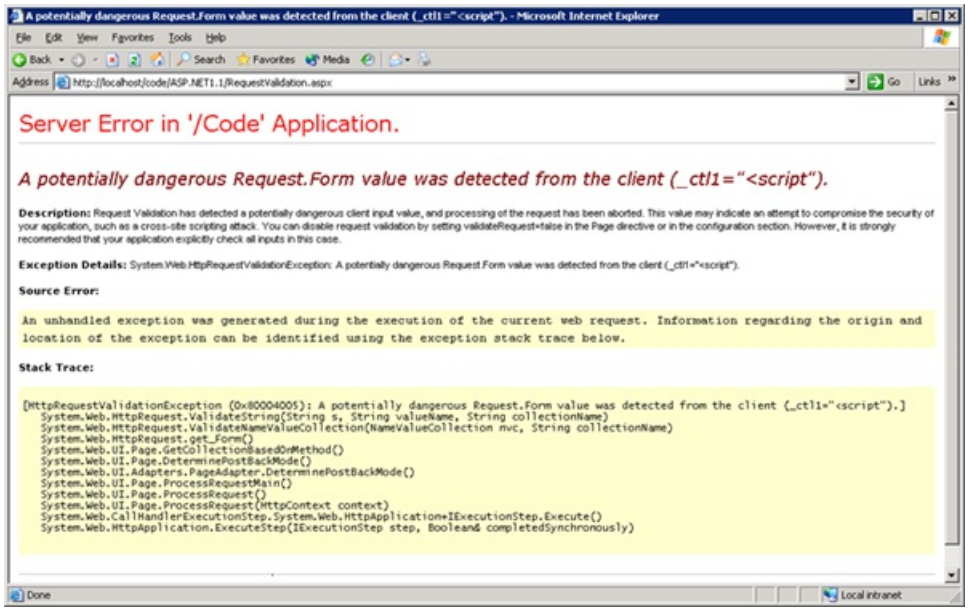
</form>

You entered: <b><asp:Label id="Label1" runat="server" /></b>
```

Running this code results in a simple page that allows you to enter some text in the textbox, click the button, and display the text in the label control:



However, were JavaScript, such as `<script>alert("hello!")</script>` to be entered and submitted we would get an exception:



The error message states that a 'potentially dangerous Request.Form value was detected' and provides more details in the description as to exactly what occurred and how to change the behavior. For example:

Request validation has detected a potentially dangerous client input value, and processing of the request has been aborted. This value may indicate an attempt to compromise the security of your application, such as a cross-site scripting attack. You can disable request validation by setting `validateRequest=false` in the Page directive or in the configuration section. However, it is strongly recommended that your application explicitly check all inputs in this case.

Disabling request validation on a page

To disable request validation on a page you must set the `validateRequest` attribute of the Page directive to `false` :

```
<%@ Page validateRequest="false" %>
```

Caution

When request validation is disabled, content can be submitted to a page; it is the responsibility of the page developer to ensure that content is properly encoded or processed.

Disabling request validation for your application

To disable request validation for your application, you must modify or create a Web.config file for your application and set the validateRequest attribute of the `<pages />` section to `false` :

```
<configuration>
   <system.web>
      <pages validateRequest="false" />
   </system.web>
</configuration>
```

If you wish to disable request validation for all applications on your server, you can make this modification to your Machine.config file.

Caution

When request validation is disabled, content can be submitted to your application; it is the responsibility of the application developer to ensure that content is properly encoded or processed.

The code below is modified to turn off request validation:

```
<%@ Page Language="VB" validateRequest="false" %>

<script runat="server">

  Public Sub Button_Click(sender As Object, e As EventArgs)

    ' Set the Label to the value of the textbox
    Label1.Text = TextBox1.Text

  End Sub

</script>

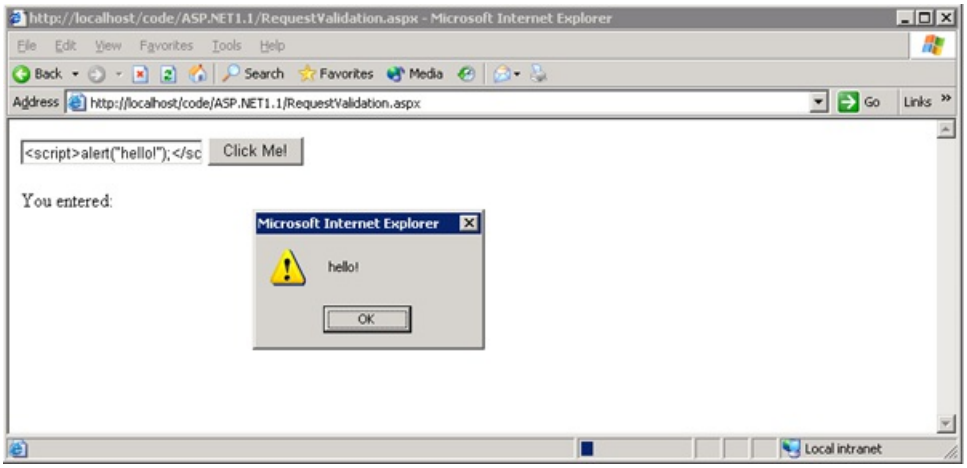
<form runat="server">

  <asp:TextBox id="TextBox1" runat="server" />
  <asp:Button OnClick="Button_Click" runat="server" Text="Click Me!"/>

</form>

You entered: <b><asp:Label id="Label1" runat="server" /></b>
```

Now if the following JavaScript was entered into the textbox `<script>alert("hello!")</script>` the result would be:



To prevent this from happening, with request validation turned off, we need to HTML encode the content.

How to HTML encode content

If you have disabled request validation, it is good practice to HTML-encode content that will be stored for future use. HTML encoding will automatically replace any '<' or '>' (together with several other symbols) with their corresponding HTML encoded representation. For example, '<' is replaced by '<' and '>' is replaced by '>'. Browsers use these special codes to display the '<' or '>' in the browser.

Content can be easily HTML-encoded on the server using the `Server.HtmlEncode(string)` API. Content can also be easily HTML-decoded, that is, reverted back to standard HTML using the `Server.HtmlDecode(string)` method.

```
<%@ Page Language="VB" validateRequest="false" %>

<script runat="server">

  Public Sub Button_Click(sender As Object, e As EventArgs)

    ' Set the Label to the value of the textbox
    Label1.Text = Server.HtmlEncode(TextBox1.Text)

  End Sub

</script>

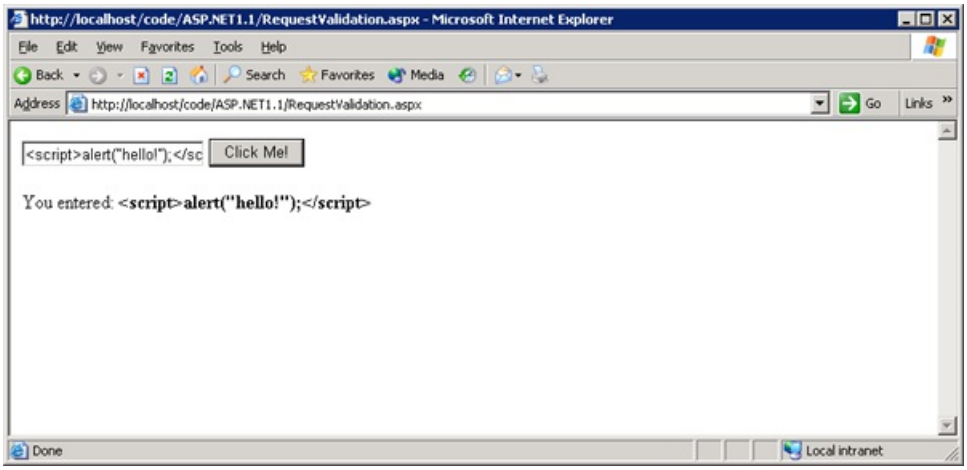
<form runat="server">

  <asp:TextBox id="TextBox1" runat="server" />
  <asp:Button OnClick="Button_Click" runat="server" Text="Click Me!"/>

</form>

You entered: <b><asp:Label id="Label1" runat="server" /></b>
```

Resulting in:



ASP.NET Data Access - Recommended Resources

2/20/2020 • 13 minutes to read • Edit Online

This topic provides links to documentation resources about how to access data in ASP.NET web applications, primarily by using the Entity Framework and SQL Server.

If you know a great blog post, stackoverflow thread, or any other link that would be useful, [send us an email](aspnetue@microsoft.com?subject=Data Access Content Map) with the link.

Last updated 4/3/2014

The topic contains the following sections:

- Getting Started with Data Access in ASP.NET
- Using the Entity Framework
  - Using Entity Framework Code First
  - Using Entity Framework Code First Migrations
  - Using Entity Framework Database First or Model First (the EF Designer)
  - Loading Related data in Entity Framework (Lazy Loading, Eager Loading, and Explicit Loading)
  - Optimizing Entity Framework Performance
  - Handling Concurrency in an Entity Framework Application
  - Books about the Entity Framework
  - Additional Entity Framework resources
- Data Binding in ASP.NET Web Forms Applications
  - Using Web Forms Model Binding
  - Using Web Forms Data Source Controls
  - Using Web Forms Data-Bound Controls and Data-Binding Expressions
- Working with SQL Server Databases
  - Working with SQL Server Express LocalDB Databases
  - Working with SQL Server Express Databases
  - Working with Windows Azure SQL Database
  - Choosing between SQL Server and Windows Azure SQL Database
- Working with NoSQL Database Management Systems
- Using LINQ queries in ASP.NET Applications
- Using Dynamic Data scaffolding
- Securing Data Access
- Optimizing Data Access Performance
- Deploying a Database
- Accessing Data through a Web Service
- Additional Resources

Getting Started with Data Access in ASP.NET

- Data Storage Options (Building Real-World Cloud Apps with Windows Azure). Chapter of an e-book about developing for the cloud. Introduces NoSQL databases as an alternative that many developers familiar with relational databases tend to overlook. Presents guidelines on what to think about when choosing relational or NoSQL, or choosing a particular platform.
- ASP.NET Data Access Options (MSDN). An introduction to data access options for relational databases for ASP.NET and guidance on how to choose platforms and access methods that are appropriate for your scenario.
- Relational database. Wikipedia). If you haven't worked with relational databases, see this page for an introduction to relational database terminology and concepts. For an introduction to SQL Server in particular see Working with SQL Server databases later in this topic.

Using the Entity Framework

- Entity Framework Development Approaches (MSDN). Guidance on how to choose an Entity Framework development approach Database First, Model First, or Code First.

Using Entity Framework Code First

The following tutorials offer downloadable sample applications:

- Getting Started with EF 6 using MVC 5. Covers a wide range of Entity Framework Code First scenarios, including Migrations and EF 6 features such as connection resiliency, command interception, and async. This is an updated version of the EF 5 / MVC 4 series. The earlier series includes a tutorial on the repository and unit-of-work patterns that is not included in the new series.
- Introduction to ASP.NET MVC 5. Covers a narrower range of Entity Framework Code First scenarios but does a more comprehensive job of introducing MVC features.
- Model Binding and Web Forms. Uses Code First in a Web Forms application.
- Getting Started with ASP.NET 4.5 Web Forms. An introduction to Web Forms with some coverage of Code First. Uses Model Binding.
- MVC Music Store. Uses Code First in an e-commerce MVC 3 application that also implements membership and authorization. The MVC version and ASP.NET membership (authentication and authorization) system used here are outdated; for more up-to-date information on ASP.NET membership, see https://asp.net/identity.

Other resources:

- Entity Framework - Code First to an Existing Database. MSDN. Video and walkthrough that shows how to use Code First with an existing database.
- Data Developer Center - Entity Framework. MSDN. For a guide to Entity Framework documentation that has been created and maintained by the Entity Framework team, see the Get Started link.

See also Books about the Entity Framework and Additional Entity Framework Resources later in this topic.

Using Entity Framework Code First Migrations

Most of the Code First tutorials listed above cover migrations. See also the following resources.

- ASP.NET Web Deployment using Visual Studio. 2-part tutorial series that shows how to use Code First Migrations to deploy a database.
- Deploy a Secure ASP.NET MVC 5 app with Membership, OAuth, and SQL Database to a Windows Azure Web Site. Microsoft Azure). How to use migrations to deploy membership and application data to Azure.
- Web Deployment Overview for Visual Studio and ASP.NET. See the **Configuring Database Deployment in Visual Studio** section for an explanation of how Code First Migrations is integrated into Visual Studio web deployment features.

- Data Developer Center - Code First Migrations (MSDN). The Entity Framework team's Migrations documentation.
- Migrations Screencast Series. EF blog). Three videos on advanced topics in Code First Migrations.
- Code First Migrations With ASP.NET Web Pages Sites. Mikesdotnetting blog). Shows how to use Code First migrations with an ASP.NET Web Pages site by putting the data context in a Visual Studio class library project.

Using Entity Framework Database First or Model First (the EF Designer)

- Getting Started with Entity Framework 6 Database First using MVC 5. Run a script in Server Explorer to create a database, and then use the Entity Framework designer to create the data model. Shows how to create simple CRUD web pages, and for other data handling functions you can follow one of the Code First tutorials since all EF workflows use the same DbContext API.

The following resources are older. They are useful if you want to use version 4.0 of the Entity Framework, and you want to use a data source control for data binding in a Web Forms application.

- Getting Started with the Entity Framework 4.0. Shows how to use the **EntityDataSource** control.
- Continuing with the Entity Framework(Shows how to use the **ObjectDataSource** Control. Includes a tutorial on concurrency handling, a tutorial on EF performance, and a tutorial on what's new in EF 4.0.

Handling related data in Entity Framework (Lazy Loading, Eager Loading, and Explicit Loading)

- Reading Related Data with the Entity Framework in an ASP.NET MVC Application. Code First, MVC sample application. The methods shown apply also to Web Forms model binding and the Database First workflow.
- Data Developer Center - Loading Related Entities (MSDN). The Entity Framework team's documentation about loading related data.

Optimizing Entity Framework performance

- Advanced Entity Framework Scenarios for an ASP.NET Application. Shows how to execute your own SQL statements or call your own stored procedures, how to disable change detection, and how to disable validation when saving changes.
- Performance Considerations for Entity Framework 5 (MSDN).
- Performance Considerations (Entity Framework) (MSDN).
- Maximizing Performance with the Entity Framework in an ASP.NET Web Application. Applies to Entity Framework 4.0.
- See also Optimizing ASP.NET data access later in this topic.

Handling Concurrency in an Entity Framework Application

- Handling Concurrency with the Entity Framework in an ASP.NET MVC Application. Code First, DbContext API, using an MVC sample application.
- Data Developer Center – Optimistic Concurrency Patterns (MSDN). The Entity Framework team's concurrency documentation.
- Handling Concurrency with the Entity Framework in an ASP.NET Web Application. Applies to Entity Framework 4.0. Database First, ObjectContext API, using a Web Forms sample application.

Books about the Entity Framework

- Programming Entity Framework: DbContext by Julie Lerman and Rowan Miller.
- Programming Entity Framework: Code First by Julie Lerman and Rowan Miller.

Both of these books are up-to-date with current recommended techniques. They provide a more comprehensive yet easy-to-follow introduction to the Entity Framework than anything available on the Internet. Another book, Programming Entity Framework by Julie Lerman, is larger and more comprehensive but it is older and many of the techniques it covers are no longer the recommended way to use the Entity

Framework. See also the list of books recommended by the Entity Framework team at Data Developer Center - Books on the MSDN site.

Other Entity Framework Resources

- Entity Framework (ADO.NET) team blog. One of the best resources for the most current information and announcements of new enhancements. For other EF-related blogs, see the Blogroll at Get Started with Entity Framework.
- MSDN Magazine. See the **Data Points** column, which is frequently about topics related to the Entity Framework.

Data Binding in ASP.NET Web Forms Applications

- ASP.NET Web Forms Data Access Options (MSDN).

Using Web Forms Model Binding

- Model Binding and Web Forms. Tutorial series using EF Code First.
- Web Forms Model Binding Part 1: Selecting Data (Scott Guthrie's blog). In these older blog posts, the property that is currently named ItemType was named ModelType, but otherwise the information they contain is valid.
- Web Forms Model Binding Part 2: Filtering Data (Scott Guthrie's blog).
- Web Forms Model Binding Part 3: Updating and Validation (Scott Guthrie's blog).
- ASP.NET 4.5 Web Forms Model Binding. (video).
- Model Binding Part 1 - Selecting Data (video).
- Model Binding Part 2 - Filtering (video).
- Getting Started with ASP.NET 4.5 Web Forms - Display Data Items and Details.

Using Web Forms Data Source Controls

- Data Source Web Server Controls (MSDN).
- Announcing the release of Dynamic Data provider and EntityDataSource control for Entity Framework 6 (Microsoft Web Development blog).

Using Web Forms Data-Bound Controls and Data-Binding Expressions

- Model Binding and Web Forms. Tutorial series that uses EF Code First.
- Getting Started with ASP.NET 4.5 Web Forms - Display Data Items and Details.
- Strongly Typed Data Controls (Scott Guthrie's blog).
- Strongly Typed Data Controls (video).
- ASP.NET 4.5 Web Forms Strong Typed Data Controls (video).
- Data-Bound Web Server Controls (MSDN).
- Data-Binding Expressions Overview (MSDN). This page only covers **Eval** and **Bind**; it has not been updated to include **Item** and **BindItem**.

Working with SQL Server Databases

- SQL Server Database Features (MSDN). For a general introduction to a wide variety of SQL Server topics, see the entries under this one in the TOC.
- SQL Server Editions (MSDN). A summary of the available SQL Server editions, with links to more information about each one.)
- SQL Server Connection Strings for ASP.NET Web Applications (MSDN).
- Using SQL Server Compact for ASP.NET Web Applications (MSDN).
- Microsoft SQL Server: Database Product Samples. Sample AdventureWorks databases.
- Installing Sample Databases. In addition to the methods shown here, you can also download one of the

sample .mdf files to the App_Data folder of a web project, convert the database to LocalDB, and create a LocalDB connection string. For information about how to do that, see How to: Upgrade to LocalDB.

See also the following sections on working with SQL Server Express and LocalDB, and choosing between SQL Server and SQL Database.

Working with SQL Server Express LocalDB Databases

- SQL Server Express 2012 LocalDB (MSDN). The official MSDN introduction to LocalDB.
- SQL Server Connection Strings for ASP.NET Web Applications (MSDN).
- How to: Upgrade to LocalDB (MSDN). How to migrate an .mdf file from an earlier version of SQL Server Express to LocalDB. You also have to go through this process if you download one of the SQL Server 2012 sample databases.
- Introducing LocalDB, an improved SQL Express (SQL Server Express blog). Has more background on why LocalDB was created than is included in MSDN.
- LocalDB: Where is My Database? (SQL Server Express blog). Information about where LocalDB database files are created.
- Using LocalDB with Full IIS, Part 1: User Profile (SQL Server Express blog). LocalDB is not designed to work with IIS. This series of blog posts explains the issues and some workarounds.

Working with SQL Server Express Databases

- SQL Server Connection Strings for ASP.NET Web Applications (MSDN). If you use the AttachDBFileName connection string setting with SQL Server Express, see especially the User Instance section of this page.
- How to take ownership of your local SQL Server Express 2008 (SQL Server Express blog). A common problem is not being able to work with SQL Server Express databases because you're not an administrator on the SQL Server Express instance. By default, only the person who installed SQL Server Express is an administrator. This blog explains how to make yourself a SQL Server Express administrator if you're an administrator on the computer.
- Can my ASP.NET web application use a SQL Server Express database in production? (MSDN).

Working with Windows Azure SQL Database

- Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL Database to a Windows Azure Web Site (Microsoft Azure site).
- SQL Databases (Microsoft Azure site). Getting started tutorials and how-to guides.
- Windows Azure SQL Database (MSDN). The top-level node of the table of contents for SQL Database in MSDN.
- Windows Azure SQL Database TechNet Wiki Articles Index (Microsoft TechNet site).
- Transient Fault Handling Application Block. A framework that enables you to handle transient network faults and connection errors that result from throttling. Available in a NuGet package: Enterprise Library 5.0 - Transient Fault Handling Application Block.
- Getting Started with SQL Database and Entity Framework (MSDN).
- Windows Azure Training Kit (Microsoft Download Center). Includes hands-on labs for SQL Database.
- Windows Azure SQL Database Community Forum.
- Moving to Windows Azure SQL Database (MSDN). One chapter of a comprehensive end-to-end scenario by the Microsoft Patterns and Practices team. Covers why you might want to migrate and how to migrate from SQL Server to SQL Database.
- Migrating SQL Server Databases to Windows Azure SQL Database (MSDN).
- SQL Database Migration Wizard. An open source tool for migrating databases to and from SQL Database.

Choosing between SQL Server and Windows Azure SQL Database

- Compare SQL Server with Windows Azure SQL Database (Microsoft TechNet site).
- Data Migration to Windows Azure SQL Database: Tools and Techniques (MSDN). Includes sections that

compare SQL Server to SQL Database and provide guidance on when to migrate from SQL Server to SQL Database.

- Windows Azure SQL Database Delivery Guide (Microsoft TechNet site).
- SQL Server Feature Limitations (Windows Azure SQL Database) (MSDN).
- Windows Azure Table Storage and Windows Azure SQL Database - Compared and Contrasted (MSDN). For an application that you deploy to Windows Azure, Windows Azure Table storage might be an alternative to Windows Azure SQL Database. This topic helps you decide between these alternatives.
- Windows Azure SQL Database (MSDN).
- Guidelines and Limitations (Windows Azure SQL Database)

Working with NoSQL Database Management Systems

- Windows Azure Data Services (Microsoft Azure site). See the Table Service feature guide and the **Big Data** section of the page.
- ASP.NET Multi-Tier Application Using Storage Tables, Queues, and Blobs (Microsoft Azure site). End-to-end tutorial with downloadable sample application that uses Windows Azure storage NoSQL tables.

Using LINQ Queries in ASP.NET Applications

- ASP.NET Data Access Options (MSDN). Includes an introduction to LINQ.
- LINQ Training Videos (Joe Stagner's blog).
- ASP.NET Forum thread with links to dynamic LINQ resources.

Using Dynamic Data Scaffolding

- Dynamic Data Project Templates (MSDN). Guidance on when to use Dynamic Data projects.
- ASP.NET Dynamic Data (MSDN).

Securing Data Access

- Securing Data Access in ASP.NET (MSDN).
- Security Considerations (Entity Framework) (MSDN).
- How To: Secure Connection Strings when Using Data Source Controls (MSDN).

Optimizing Data Access Performance

- ASP.NET Performance Overview (MSDN).
- ASP.NET Caching (MSDN).
- Improving ASP.NET Performance (MSDN). There is a "Retired Content" warning at the top of this page, but most of the information is still relevant and there is no comparable updated resource.
- Improving SQL Server Performance (MSDN). Same comment as the previous link.

See also Optimizing Entity Framework performance earlier in this topic.

Deploying a Database

- ASP.NET Web Deployment - Recommended Resources (MSDN).

Accessing Data through a Web Service

- Accessing Data through a Web Service (MSDN). Guidance on when to use Web API versus WCF.
- Getting Started with ASP.NET Web API.

- WCF Data Services (MSDN).

Additional Resources

- ASP.NET Data Access FAQ (MSDN).
- ASP.NET Web Forms Tutorials - Data. Most of these tutorials are relatively old; make sure you read ASP.NET Data Access Options and Data Storage Options (Building Real-World Cloud Apps with Windows Azure) first so that you don't get too far into a data access method that isn't right for your scenario.
- ASP.NET MVC Content Map.
- ASP.NET Web Pages Tutorials - Data.
- Accessing Data in Visual Studio (MSDN). Provides a list of links similar to this content map but with a focus on Visual Studio rather than ASP.NET.

ASP.NET Web Deployment - Recommended Resources

2/20/2020 • 11 minutes to read • Edit Online

This topic provides links to documentation resources about how to deploy (publish) ASP.NET web applications to IIS by using Visual Studio 2010, Visual Web Developer 2010, and later versions.

If you know a great blog post, stackoverflow thread, or any other link that would be useful, send us an email with the link.

> **NOTE**
> Many of these resources describe deployment features that are available only if you install a recent release of the Visual Studio Web Publish Update. Some of the features are available only in Visual Studio 2012 or Visual Studio 2013.

This topic contains the following sections:

- Understanding deployment options for web projects
- Finding hosting providers for an ASP.NET application
- Deploying a web application from Visual Studio
- Deploying a web application by creating and installing a web deployment package
- Deploying a web application using a continuous integration (CI) process
- Using Web.config transformations to change settings in the destination Web.config file or app.config file during deployment
- Using Web Deploy parameters to change settings in the destination web application during deployment
- Making sure an application is off-line during deployment
- Deploying a database or changes to a database as part of web application deployment
- Deploying a database separately from web application deployment
- Deploying a web application that uses ASP.NET application services such as membership and profiling
- Precompiling for deployment
- Deploying an intranet web application
- Automating common deployment tasks that are not automated out of the box
- Configuring web servers so that developers can deploy web applications to them using Web Deploy
- Configuring servers for a hosting provider
- Troubleshooting deployment problems
- Getting help with a specific deployment question
- Additional Resources

Understanding deployment options for web projects

- Web Deployment Overview for Visual Studio and ASP.NET (MSDN).
- How to Deploy a Windows Azure Web Site. Explains options and links to resources for deploying web projects to Windows Azure Web Sites, including continuous delivery (automated from source control) as well as using Visual Studio.
- Visual Studio 2012 Web Publishing Improvements (Video by Scott Hanselman).

- Overview Post for Web Deployment in VS 2010 (Vishal Joshi's blog). An older blog post but some of the Visual Studio 2010 resources it links to have information that is still relevant for Visual Studio 2012.

Finding hosting providers for an ASP.NET application

- ASP.NET Hosting

Deploying a web application from Visual Studio

- How to Deploy a Windows Azure Web Site. Explains options and provides links to resources for deploying web projects to Windows Azure Web Sites. Includes a section about deploying from Visual Studio.
- ASP.NET Web Deployment using Visual Studio. 12-part tutorial series, shows how to deploy web applications with SQL Server databases. For database deployment uses both the dbDacFx provider and Entity Framework Code First Migrations. Also includes information about Web.config file transformations, deploying individual files, command-line deployment, and how to customize the Visual Studio web publish pipeline by editing .pubxml files. Applies to all ASP.NET web projects, including Web Forms, MVC, and Web API.)
- How to: Deploy a Web Project Using One-Click Publish in Visual Studio (Reference information for the Visual Studio Web Publish wizard.)
- Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio. This is an earlier version of **ASP.NET Web Deployment using Visual Studio** listed at the top of this section. Mainly useful now for information about how to deploy SQL Server Compact databases and how to migrate from SQL Server Compact to a full edition of SQL Server.
- .NET Multi-Tier Application Using Storage Tables, Queues, and Blobs (Microsoft Azure site). 5-part tutorial series, shows how to create an MVC project and deploy it to a Windows Azure Cloud Service.

Deploying a web application by creating and installing a web deployment package

- How to: Create a Web Deployment Package in Visual Studio (MSDN).
- How to: Install a Deployment Package Using the deploy.cmd File Created by Visual Studio (MSDN).
- Using a Web Deploy package to deploy to IIS on the dev box and to a third party host (Sayed Hashimi's blog). How to use IIS Manager to install a deployment package in IIS on the local computer and at a hosting company that supports IIS Manager for Remote Administration.
- Building a Web Deploy Package From Visual Studio 2010 (IIS.NET web site). Includes instructions for command-line package creation and installation.
- Package Once Publish Anywhere (Sayed Hashimi's blog). Introduces a NuGet package that automates the process of transforming the Web.config file for multiple destination environments, so that you can deploy one package to multiple servers. See also the PackageWeb video by Sayed Hashimi.

See also the following section.

Deploying a web application using a continuous integration (CI) process

- Continuous Integration and Continuous Delivery (Building Real-World Cloud Apps with Windows Azure). E-book chapter that introduces continuous integration and continuous delivery.
- How to Deploy a Windows Azure Web Site. Explains options and links to resources for deploying web projects to Windows Azure Web Sites. Includes a section about automating deployment from source control.
- Deploying Web Applications in Enterprise Scenarios. 40-part tutorial series, shows how to automate deployment in a CI process using Visual Studio 2010 and Team Foundation Server 2010.
- Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build, by Sayed Hashimi and William

Bartholomew. This is a book, not a web resource, but it is an essential guide for learning how to configure MSBuild for continuous integration scenarios.

- MSBuild Extension Pack. Includes deployment tasks.
- Team Foundation Build Customization Guide. Documentation by ALM Rangers on setting up Team Foundation Server covers web deployment and includes tutorials and videos.
- SlowCheetah XML transforms from a CI server (Sayed Hashimi's blog). Explains how to use SlowCheetah, A Visual Studio add-in for transforming app.config and other XML files.

See also Making sure an application is off-line during deployment later in this page.

Using Web.config transformations to change settings in the destination Web.config file or app.config file during deployment

- Web.config File Transformations.
- Web.config Transformation Syntax for Web Project Deployment Using Visual Studio (MSDN).
- Web Tools 2012.2 - web.config transforms (YouTube video by Sayed Hashimi). Shows how to set up and preview Web.config transforms.
- How do I disable Web.config transformation? (MSDN).
- When should I use Web Deploy parameters instead of Web.config transformations? (MSDN).
- XDT (XML Document Transform) released on codeplex.com (.NET Web Development and Tools blog). Announces availability of the source code for the Web.config file transformation engine and lists some tools that use it.
- Windows Azure Web Sites: How Application Strings and Connection Strings Work (Microsoft Azure blog). An alternative to Web.config transforms if your destination environment is Windows Azure Web Sites and you want to transform `appSettings` or `connectionStrings`.

Using Web Deploy parameters to change settings in the destination web application during deployment

- How to: Use Web Deploy Parameters in a Web Deployment Package (MSDN).
- MSDeploy: How to update app settings on publish based on the publish profile (Sayed Hashimi's blog). Shows how to integrate Web deploy parameters into Visual Studio publish profiles.
- Web Deploy Parameterization (IIS.NET web site).
- Web Deploy Parameterization in Action (Vishal Joshi's blog).
- Web Deploy Parameterization vs. Web.config Transformation (Vishal Joshi's blog).
- Windows Azure Web Sites: How Application Strings and Connection Strings Work (Microsoft Azure blog). An alternative to Web deploy parameters if your destination environment is Windows Azure Web Sites and you want to parameterize `appSettings` or `connectionStrings`.

Making sure an application is off-line during deployment

- ASP.NET Web Deployment using Visual Studio: Deploying a Code Update. See the section **Take the application offline during deployment.**
- Taking an Application Offline before Publishing (IIS.net site). Explains a feature built into Web Deploy 3.0 that automates handling of an app_offline.htm file. This feature does not work with custom app_offline.htm files.
- How to take your web app offline during publishing (Sayed Hashimi's blog). How to automate the process of using a custom app_offline.htm file.
- Web publishing updates for app offline and usechecksum (Microsoft Web Development blog). Another option for automating use of app_offline.htm file.
- Web Deploy 3.5 RTW (IIS.net site). New feature in Web Deploy 3.5 for custom app_offline.htm files.

Deploying a database or changes to a database as part of web application deployment

- Configuring Database Deployment in Visual Studio (MSDN). Overview of options for deploying a database with a web project.
- ASP.NET Web Deployment using Visual Studio. 12-part tutorial series, shows database deployment by using dbDacFx provider and Entity Framework Code First Migrations.
- How to: Deploy a Web Project Using One-Click Publish in Visual Studio (MSDN).
- Deploy a Secure ASP.NET MVC 5 app with Membership, OAuth, and SQL Database to a Windows Azure Web Site. A long tutorial that builds and deploys an application that uses a single SQL Server database both for membership and application data.
- Deploying an ASP.NET Web Application with SQL Server Compact using Visual Studio. 12-part tutorial series, shows how to deploy SQL Server Compact databases and how to migrate from SQL Server Compact to a full edition of SQL Server.

See also Deploying a web application by creating and installing a web deployment package and Deploying a web application using a continuous integration (CI) process earlier in this page.

Deploying a database separately from web application deployment

- SQL Server Data Tools (MSDN).
- Including Data in a SQL Server Database Project (SQL Server Data Tools team blog). How to deploy both schema and data when deploying a database.
- How to Deploy a Database to Windows Azure (Microsoft Azure site)
- Migrating Databases to Windows Azure SQL Database (formerly SQL Azure) (MSDN).
- Migrating a Database to SQL Azure using SSDT (SQL Server Data Tools team blog).
- Migrating Data-Centric Applications to Windows Azure (MSDN).
- Migrating SQL Server Databases to Windows Azure SQL Database (MSDN).

Deploying a web application that uses ASP.NET application services such as membership and profiling

- Deploy a Secure ASP.NET MVC 5 app with Membership, OAuth, and SQL Database to a Windows Azure Web Site. A long tutorial that builds and deploys an application that uses a single SQL Server database both for membership and application data.
- ASP.NET Identity. Resources for ASP.NET Identity.
- ASP.NET Web Deployment using Visual Studio. 12-part tutorial series, shows how to deploy an ASP.NET membership database.
- Configuring a Website that Uses Application Services. For web site projects but is also relevant for web application projects.
- Users and Roles On The Production Website. For web site projects but is also relevant for web application projects.

Precompiling for deployment

- ASP.NET Web Application Project Precompilation Overview (MSDN).
- Package/Publish Web Tab, Project Properties (MSDN).
- Advanced Precompile Settings Dialog Box (MSDN).

Deploying an intranet web application

- Use the On-Premises Organizational Authentication Option (ADFS) With ASP.NET in Visual Studio 2013 (Blog by Vittorio Bertocci.).
- How to Create an Intranet Site Using ASP.NET MVC (MSDN). Older walkthrough writen for Visual Studio 2010, does not reflect major changes in intranet project templates introduced in Visual Studio 2013.

Automating common deployment tasks that are not automated out of the box

- ASP.NET Web Deployment using Visual Studio: Deploying Extra Files.
- Setting Folder Permissions on Web Publish (Sayed Hashimi's blog).
- How to extend the targets file to include registry settings for a web project package (Web Development Tools blog).
- Extending XML (Web.config) transformation (Sayed Hashimi's blog). Shows how to create custom XDT transforms.
- Web Deployment Tool (MSDeploy) Custom Provider Take 1 (Sayed Hashimi's blog). Shows how to create a Web Deploy custom provider.
- How to package and deploy COM components (Web Development Tools blog).
- How to package .NET assemblies (Web Development Tools blog). How to deploy assemblies to the GAC.
- Script Out Everything - Initialize Your Windows Azure VM for Your Web Server with IIS, Web Deploy and Other Stuff (Tugberk Ugurlu's blog).

Configuring web servers so that developers can deploy web applications to them using Web Deploy

- Installing and Configuring Web Deploy for Administrator and non-administrator Deployments (IIS.net site).

Configuring servers for a hosting provider

- Microsoft ASP.NET 4 Hosting Deployment Guide (Microsoft Download Center).
- Generate a Profile XML File (IIS.net site).

Troubleshooting deployment problems

- Troubleshooting Windows Azure Web Sites in Visual Studio (Microsoft Azure site).
- ASP.NET Web Deployment using Visual Studio: Troubleshooting.
- Troubleshooting Common Problems With Web Deploy.
- Web Deploy Error Codes (IIS.net site).
- Web Deployment FAQ for Visual Studio and ASP.NET (MSDN).
- Core Differences Between IIS and the ASP.NET Development Server.
- Common Configuration Differences Between Development and Production.
- Hosting ASP.NET Applications in Medium Trust (4 Guys from Rolla site).

Getting help with a specific deployment question

- [ASP.NET Configuration and Deployment forum](https://forums.asp.net/26.aspx/1?Configuration and Deployment).
- StackOverflow.com.

Additional Resources

This section provides links to additional resources that are useful for learning more about how to use Visual Studio and IIS deployment tools.

The following blogs frequently contain information about Visual Studio web deployment:

- Web Development Tools at Microsoft blog.
- Sayed Hashimi's blog.

The following resources provide documentation about Web Deploy, the IIS framework that Visual Studio uses to perform web application project deployment tasks. You can ask questions about Web Deploy in the Web Deployment Tool forum on the IIS.net web site.

- Introduction to Web Deploy.
- Installing and Configuring Web Deploy.
- PowerShell Scripts for Automating Web Deploy Setup.
- Web Deployment Tool. Top-level table of contents node for Web Deploy documentation on the TechNet site. Includes useful reference information but most of the TechNet pages have not been updated for years.
- Microsoft.Web.Deployment Namespace. API documentation, has not been updated since version 1.0.
- The Microsoft Web Deployment Team blog.
- Publish tab in IIS.net web site.

ASP.NET and Web Tools 2012.2 Release Notes

2/20/2020 • 11 minutes to read • Edit Online

This document describes the release of ASP.NET and Web Tools 2012.2. It is an update to Visual Studio Web Tooling and ASP.NET.

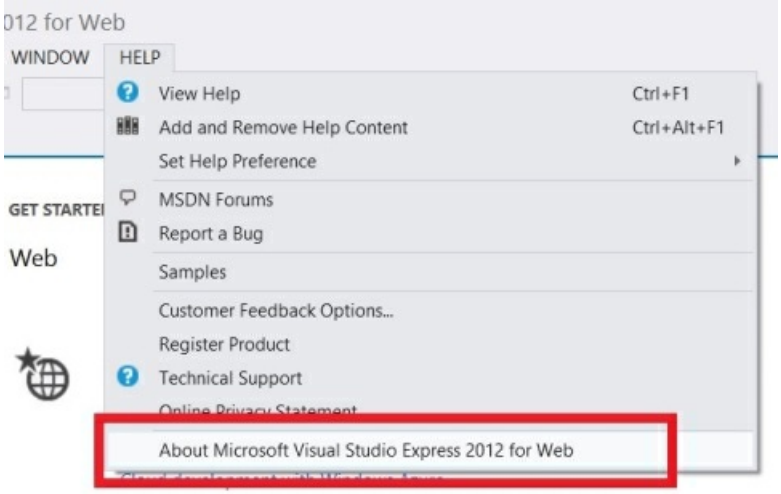
- Installation Notes
- Documentation
- Support
- Software Requirements
- New Features in ASP.NET and Web Tools 2012.2
  - Tooling
  - Web Publishing
  - ASP.NET MVC Templates
  - ASP.NET Web API
  - ASP.NET SignalR
  - ASP.NET Friendly URLs
- Known Issues and Breaking Changes

Installation Notes

ASP.NET and Web Tools 2012.2 for Visual Studio 2012 can be installed using Web Platform installer. This is an update to Visual Studio 2012 or Visual Studio Express 2012 for Web, which is required. If you do not have Visual Studio installed, Visual Studio Express 2012 for Web will be installed.

You can also install ASP.NET and Web Tools 2012.2 manually. You must have Visual Studio 2012 or Visual Studio Express 2012 for Web installed. Then use the following instructions:

1. Download ASP.NET and Web Frameworks 2012.2 installer from Download Center.
2. When prompted click Run. You can also save the file to run it later.
3. Verify the version of Visual Studio you will update. You can do this by launching the Visual Studio you wish to update. Then click the Help menu item.



4. If you see the menu item "About Microsoft Visual Studio 2012 for Web" then download Web Developer Tools 2012.2 - Visual Studio Express 2012 for Web. Otherwise download Web Developer Tools 2012.2 - Visual Studio 2012.
5. When prompted click Run. You can also save the file to run it later.

> **NOTE**
> ASP.NET and Web Tools 2012.2 release does not include SQL Server Data Tools. SQL Server and Windows Azure SQL Databases provides a richer set of database tooling including offline project-backed development, schema comparison and enhanced database deployment capabilities. For more information or to install SQL Server Data Tools visit https://go.microsoft.com/fwlink/?LinkID=237127.

Documentation

Tutorials and other information about ASP.NET and Web Tools 2012.2 are available from ASP.NET web site (https://www.asp.net).

Support

ASP.NET and Web Tools 2012.2 is officially released and supported. You can use your normal support channel. You can also post questions to the ASP.NET forums (https://forums.asp.net/), where members of the ASP.NET community are frequently able to provide informal support.

Software Requirements

The ASP.NET and Web Tools 2012.2 requires Visual Studio 2012 or Visual Studio Express 2012 for Web.

New Features in ASP.NET and Web Tools 2012.2

This section describes features that have been introduced in the ASP.NET and Web Tools 2012.2 release.

Tooling

- Page Inspector
  - Support JavaScript selection mapping allowing Page Inspector to map items that were dynamically added to the page back to the corresponding JavaScript code.
  - The ability to see CSS updates in real-time.
  - For more information, read CSS Auto-Sync and JavaScript Selection Mapping in Page Inspector.
- Editor

  - Support syntax highlighting for CoffeeScript, Mustache, Handlebars, and JsRender.
  - The HTML editor provides Intellisense for Knockout bindings.
  - LESS editing and compiler support to enable building dynamic CSS using LESS.
  - Paste JSON as a .NET class. Using this Special Paste command to paste JSON into a C# or VB.NET code file, and Visual Studio will automatically generate .NET classes inferred from the JSON.
- Mobile Emulator support adds extensibility hooks so that third-party emulators can be installed as a VSIX. The installed emulators will show up in the F5 dropdown, so that developers can preview their websites on a variety of mobile devices. Read more about this feature in Scott Hanselman's blog entry on the new BrowserStack integration with Visual Studio.

Web Publishing

- Web site projects now have the same publishing experience as Web Application projects including publishing to Windows Azure Web Sites.
- Selective publish – for one or more files you can perform the following actions (after publishing to a Web Deploy endpoint):
  - Publish selected files.
  - See the difference between a local file and a remote file.
  - Update the local file with the remote file or update the remote file with the local file.

ASP.NET MVC Templates

- The new Facebook Application template makes writing Facebook Canvas applications easy. In a few simple steps, you can create a Facebook application that gets data from a logged in user and integrates with their friends. The template includes a new library to take care of all the plumbing involved in building a Facebook app, including authentication, permissions, accessing Facebook data and more. For more information on using the Facebook Application template see https://go.microsoft.com/fwlink/?LinkID=269921.
- A new Single Page Application MVC template allows developers to build interactive client-side web apps using HTML 5, CSS 3, and the popular Knockout and jQuery JavaScript libraries, on top of ASP.NET Web API. The template includes a "todo" list application that demonstrates common practices for building a JavaScript HTML5 application that uses a RESTful server API. You can read more at https://www.asp.net/single-page-application.
- You can now create a VSIX that adds new templates to the ASP.NET MVC New Project dialog. Learn how here: https://go.microsoft.com/fwlink/?LinkId=275019
- FixedDisplayModes package – MVC project templates have been updated to include the new 'FixedDisplayModes' NuGet package, which contains a workaround for a bug in MVC 4. For more information on the fix contained in the package, refer to this blog post (https://blogs.msdn.com/b/rickandy/archive/2012/09/17/asp-net-mvc-4-mobile-caching-bug-fixed.aspx) from the MVC team.

ASP.NET Web API

ASP.NET Web API has been enhanced with several new features:

- ASP.NET Web API OData
- ASP.NET Web API Tracing
- ASP.NET Web API Help Page

ASP.NET Web API OData

ASP.NET Web API OData gives you the flexibility you need to build OData endpoints with rich business logic over any data source. With ASP.NET Web API OData you control the amount of OData semantics that you want to expose. ASP.NET Web API OData is included with the ASP.NET MVC 4 project templates and is also available from NuGet (http://www.nuget.org/packages/microsoft.aspnet.webapi.odata).

ASP.NET Web API OData currently supports the following features:

- Enable OData query semantics by applying the [Queryable] attribute.
- Easily validate OData queries and restrict the set of supported query options, operators and functions.
- Parameter bind to ODataQueryOptions directly to get an abstract syntax tree representation of the query that can then be validated and applied to an IQueryable or IEnumerable.
- Enable service-driven paging and next page link generation by specifying result limits on [Queryable] attribute.
- Request an inlined count of the total number of matching resources using $inlinecount.
- Control null propagation.
- Any/All operators in $filter.
- Infer an entity data model by convention or explicitly customize a model in a manner similar to Entity Framework Code-First.
- Expose entity sets by deriving from EntitySetController.
- Simple, customizable conventions for exposing navigation properties, manipulating links and implementing OData actions.
- Simplified routing using the MapODataRoute extension method.
- Support for versioning by exposing multiple EDM models.
- Expose service document and $metadata so you can generate clients (.NET, Windows Phone, Windows Store, etc.) for your Web API.
- Support for the OData Atom, JSON, and JSON verbose formats.
- Create, update, partially update (PATCH) and delete entities.
- Query and manipulate relationships between entities.
- Create relationship links that wire up to your routes.
- Complex types.
- Entity Type Inheritance.
- Collection properties.
- Enums.
- OData actions.
- Built upon the same foundation as WCF Data Services, namely ODataLib (http://www.nuget.org/packages/microsoft.data.odata).

For more information on ASP.NET Web API OData see https://go.microsoft.com/fwlink/?LinkId=271141.

ASP.NET Web API Tracing

ASP.NET Web API Tracing integrates tracing data from your web APIs with .NET Tracing. It is now enabled by default in the Web API project template. Tracing data for your web APIs is sent to the Output window and is made available through IntelliTrace. ASP.NET Web API Tracing enables you to trace information about your Web API when hosted on Windows Azure through integration with Windows Azure Diagnostics. You can also install and enable ASP.NET Web API Tracing in any application using the ASP.NET Web API Tracing NuGet package (http://www.nuget.org/packages/microsoft.aspnet.webapi.tracing).

For more information on configuring and using ASP.NET Web API Tracing see https://go.microsoft.com/fwlink/?LinkID=269874.

ASP.NET Web API Help Page

The ASP.NET Web API Help Page is now included by default in the Web API project template. The ASP.NET Web API Help Page automatically generates documentation for web APIs including the HTTP endpoints, the supported HTTP methods, parameters and example request and response message payloads. Documentation is automatically pulled from comments in your code. You can also add the ASP.NET Web API Help Page to any application using the ASP.NET Web API Help Page NuGet package (http://www.nuget.org/packages/microsoft.aspnet.webapi.helppage).

For more information on setting up and customizing the ASP.NET Web API Help Page see https://go.microsoft.com/fwlink/?LinkId=271140.

ASP.NET SignalR

ASP.NET SignalR makes it simple to add real-time web capabilities to your ASP.NET application, using WebSockets if available and automatically falling back to other techniques when it isn't.

For more information on using ASP.NET SignalR see https://go.microsoft.com/fwlink/?LinkId=271271.

ASP.NET Friendly URLs

ASP.NET FriendlyURLs makes it very easy for web forms developers to generate cleaner looking URLs(without the .aspx extension). It requires little to no configuration and can be used with existing ASP.NET v4.0 applications. The FriendlyURLs feature also makes it easier for developers to add mobile support to their applications, by supporting switching between desktop and mobile views.

For more information on installing and using ASP.NET Friendly URLs see http://www.hanselman.com/blog/IntroducingASPNETFriendlyUrlsCleanerURLsEasierRoutingAndMobileViewsForASPNETWebForms.aspx.

Known Issues and Breaking Changes

This section describes known issues and breaking changes that are in the ASP.NET and Web Tools 2012.2 release.

Installation Issues

Out of order installs of Visual Studio 2012

Installing an additional SKU of Visual Studio 2012 after installing the ASP.NET and Web Tools 2012.2 will require a repair operation. Consider the following sequence:

1. Install Visual Studio 2012 Express for Web
2. Install ASP.NET and Web Tools 2012.2
3. Install Visual Studio 2012 Professional, Premium or Ultimate

Step 2 would only result in installing updates for Express for Web. To ensure that the additional SKU installed during step 3 contains the update you will need to repair the ASP.NET and Web Tools 2012.2 to install the updates for the last SKU installed. This also applies if the SKUs in Step 1 and 3 are reversed.

Installing Microsoft ASP.NET and Web Tools 2012.2 when Visual Studio is open

If VS is open during installation of Microsoft ASP.NET and Web Tools 2012.2, Visual Studio might end up in a bad state. It is recommended that users close all instances of Visual Studio before proceeding with install.

Canceling ASP.NET and Web Tools 2012.2 setup in the middle of installation

Canceling ASP.NET and Web Tools 2012.2 setup in the middle of installation will leave Visual Studio in a bad state. To address this problem follow these steps:

- Go to Add Remove Programs
- Uninstall Microsoft ASP.NET and Web Tools 2012.2, if present.
- Reinstall Microsoft ASP.NET and Web Tools 2012.2

After uninstalling ASP.NET and Web Tools 2012.2 the ASP.NET MVC 4 templates and Razor v2 Web Site templates are missing

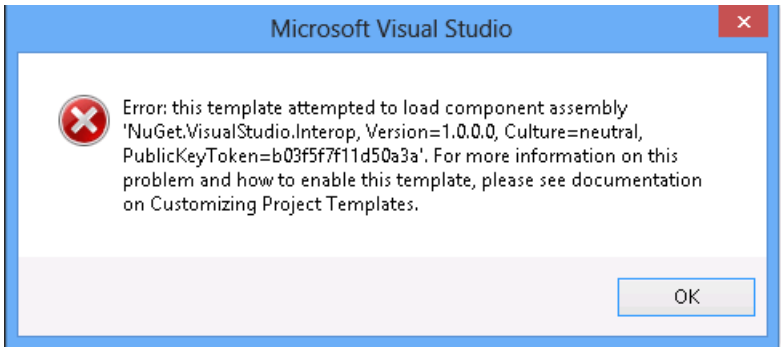
Uninstalling ASP.NET and Web Tools 2012.2 will also uninstall all of ASP.NET MVC 4 and Razor v2 Web Site templates from Visual Studio 2012.

The workaround is to repair your Visual Studio 2012 installation to reinstall ASP.NET MVC 4 and Razor v2 Web Site templates.

Tooling Issues

NuGet error reported during project creation

After installing ASP.NET and Web Tools 2012.2 you may see the following error when creating an MVC 4 project



The ASP.NET and Web Tools 2012.2 ships NuGet 2.1 and will upgrade the extension in Visual Studio 2012. In some cases, the VSIX installer will fail to correctly update the VSIX. The following steps will allow you to address this problem:

1. Start Visual Studio 2012 as an Administrator
2. Go to Tools->Extensions and Updates and uninstall NuGet.
3. Close Visual Studio
4. Navigate to the ASP.NET and Web Tools 2012.2 installation folder:
   a. For Visual Studio 2012: **Program Files\Microsoft ASP.NET\ASP.NET Web Stack\Visual Studio 2012**
   b. For Visual Studio 2012 Express for Web: **Program Files\Microsoft ASP.NET\ASP.NET Web Stack\Visual Studio Express 2012 for Web**
5. Double click on the NuGet.Tools.vsix to reinstall NuGet

Web API Issues

Parsing issues in $filter and DateTime literals

The OData URI parser fails to parse partial datetime literals properly. For example, $filter=start eq datetime'2012-12-31T12:00' fails to parse properly. A workaround is to use the full literal, $filter=start eq datetime'2012-12-31T12:00:00'.

OData doesn't support case-insensitive property names.

OData doesn't support case-insensitive property names in OData queries and odata path. See workitems:

- http://aspnetwebstack.codeplex.com/workitem/366
- http://aspnetwebstack.codeplex.com/workitem/704

If users have different casing on javascript client side and server side, they probably will encounter this issue. This issue is by design in odata protocol. However, many users reports this issue. To work around it, users have to correct their cases in URL.

Default OData routing conventions doesn't support POST/PUT on navigation property.

Default OData routing conventions doesn't support POST/PUT on navigation property. See workitem http://aspnetwebstack.codeplex.com/workitem/366. We are missing this commonly used convention in default conventions.

To work around it, users need to extend new routing convention to support it.

Facebook Template Issues

Facebook Application template only works using .NET 4.5

You must select .NET 4.5 in the framework dropdown list in the New Project dialog to see the Facebook Application template in ASP.NET MVC 4.

Real-time Update Controller

The Facebook Application template allows user easily create a Web API Controller to handle real-time updates from

Facebook. If your development machine is behind NAT, your Controller may not work without further network configuration. See here for details: http://facebook.stackoverflow.com/questions/5259467/can-a-computer-behind-a-nat-router-receive-realtime-updates-from-facebook

Query string values conflict with Facebook OAuth parameters

The following fields conflict with Facebook OAuth dialog's call back URL. Do not add your own query string values with the following names: code, error, error_description, error_reason.

Using Page Inspector with Facebook Template

You can't use the Page Inspector feature in Visual Studio 2012 while debugging your Facebook Application. The Page Inspector does not currently support iframes.

Single Page Application Template Issues

With JQuery 1.9/Knockout 2.2.1 update, when running default MVC SPA project, new todo item edit enter focus event is not handled properly.

With JQuery 1.9/Knockout 2.2.1 update, when running default MVC SPA project, new todo item edit enter no longer focus back to the new todo item edit box after entering the new todo item to the todo list.

To workaround reference http://knockoutjs.com/documentation/hasfocus-binding.html, and make similar fix to the following sample code:

File todo.model.js
function todolist(data), add following:
self.isSelected = ko.observable(false);

function todoList.prototype.addTodo, add the following blacked text:
self.isSelected(true);
self.newTodoTitle("");

File index.cshtml, add the following blacked text:
<form data-bind="submit: addTodo">
<input class="addTodo" type="text" data-bind="value: newTodoTitle, placeholder: 'Type here to add', blurOnEnter: true, **hasfocus: isSelected**, event: { blur: addTodo }" />
</form>

What's New in ASP.NET 4.5 and Visual Studio 2012

2/20/2020 • 45 minutes to read • Edit Online

This document describes new features and enhancements that are being introduced in ASP.NET 4.5. It also describes improvements being made for web development in Visual Studio 2012. This document was originally published on February 29, 2012.

- ASP.NET Core Runtime and Framework
  - Asynchronously Reading and Writing HTTP Requests and Responses
  - Improvements to HttpRequest handling
  - Asynchronously flushing a response
  - Support for *await* and *Task*-Based Asynchronous Modules and Handlers
  - Asynchronous HTTP modules
  - Asynchronous HTTP handlers
  - New ASP.NET Request Validation Features
  - Deferred ("lazy") request validation
  - Support for unvalidated requests
  - AntiXSS Library
  - Support for WebSockets Protocol
  - Bundling and Minification
  - Performance Improvements for Web Hosting
    - Key Performance Factors
    - Requirements for New Performance Features
    - Sharing Common Assemblies
    - Using multi-Core JIT compilation for faster startup
    - Tuning garbage collection to optimize for memory
    - Prefetching for web applications
- ASP.NET Web Forms
  - Strongly Typed Data Controls
  - Model Binding
    - Selecting data
    - Value providers
    - Filtering by values from a control
  - HTML Encoded Data-Binding Expressions
  - Unobtrusive Validation
  - HTML5 Updates

- ASP.NET MVC 4
- ASP.NET Web Pages 2
- Visual Studio 2012 Release Candidate
  - Project Sharing Between Visual Studio 2010 and Visual Studio 2012 Release Candidate (Project Compatibility)
  - Configuration Changes in ASP.NET 4.5 Website Templates
  - Native Support in IIS 7 for ASP.NET Routing
  - HTML Editor
    - Smart Tasks
    - WAI-ARIA support
    - New HTML5 snippets
    - Extract to user control
    - IntelliSense for code nuggets in attributes
    - Automatic renaming of matching tag when you rename an opening or closing tag
    - Event handler generation
    - Smart indent
    - Auto-reduce statement completion
  - JavaScript Editor
    - Code outlining
    - Brace matching
    - Go to Definition
    - ECMAScript5 support
    - DOM IntelliSense
    - VSDOC signature overloads
    - Implicit references
  - CSS Editor
    - Auto-reduce statement completion
    - Hierarchical indentation.
    - CSS hacks support
    - Vendor specific schemas (-moz-,-webkit)
    - Commenting and uncommenting support
    - Color picker
    - Snippets
    - Custom regions
  - Page Inspector
  - Publishing
    - Publish profiles
    - ASP.NET precompilation and merge
- IIS Express
- Disclaimer

ASP.NET Core Runtime and Framework

Asynchronously Reading and Writing HTTP Requests and Responses

ASP.NET 4 introduced the ability to read an HTTP request entity as a stream using the *HttpRequest.GetBufferlessInputStream* method. This method provided streaming access to the request entity. However, it executed synchronously, which tied up a thread for the duration of a request.

ASP.NET 4.5 supports the ability to read streams asynchronously on an HTTP request entity, and the ability to flush asynchronously. ASP.NET 4.5 also gives you the ability to double-buffer an HTTP request entity, which provides easier integration with downstream HTTP handlers such as .aspx page handlers and ASP.NET MVC controllers.

Improvements to HttpRequest handling

The Stream reference returned by ASP.NET 4.5 from *HttpRequest.GetBufferlessInputStream* supports both synchronous and asynchronous read methods. The *Stream* object returned from *GetBufferlessInputStream* now implements both the BeginRead and EndRead methods. The asynchronous *Stream* methods let you asynchronously read the request entity in chunks, while ASP.NET releases the current thread between each iteration of an asynchronous read loop.

ASP.NET 4.5 has also added a companion method for reading the request entity in a buffered way: *HttpRequest.GetBufferedInputStream*. This new overload works like *GetBufferlessInputStream*, supporting both synchronous and asynchronous reads. However, as it reads, *GetBufferedInputStream* also copies the entity bytes into ASP.NET internal buffers so that downstream modules and handlers can still access the request entity. For example, if some upstream code in the pipeline has already read the request entity using *GetBufferedInputStream*, you can still use *HttpRequest.Form* or *HttpRequest.Files*. This lets you perform asynchronous processing on a request (for example, streaming a large file upload to a database), but still run .aspx pages and MVC ASP.NET controllers afterward.

Asynchronously flushing a response

Sending responses to an HTTP client can take considerable time when the client is far away or has a low-bandwidth connection. Normally ASP.NET buffers the response bytes as they are created by an application. ASP.NET then performs a single send operation of the accrued buffers at the very end of request processing.

If the buffered response is large (for example, streaming a large file to a client), you must periodically call *HttpResponse.Flush* to send buffered output to the client and keep memory usage under control. However, because *Flush* is a synchronous call, iteratively calling *Flush* still consumes a thread for the duration of potentially long-running requests.

ASP.NET 4.5 adds support for performing flushes asynchronously using the *BeginFlush* and *EndFlush* methods of the *HttpResponse* class. Using these methods, you can create asynchronous modules and asynchronous handlers that incrementally send data to a client without tying up operating-system threads. In between *BeginFlush* and *EndFlush* calls, ASP.NET releases the current thread. This substantially reduces the total number of active threads that are needed in order to support long-running HTTP downloads.

Support for *await* and *Task* - Based Asynchronous Modules and Handlers

The .NET Framework 4 introduced an asynchronous programming concept referred to as a *task*. Tasks are represented by the *Task* type and related types in the *System.Threading.Tasks* namespace. The .NET Framework 4.5 builds on this with compiler enhancements that make working with *Task* objects simple. In the .NET Framework 4.5, the compilers support two new keywords: *await* and *async*. The *await* keyword is syntactical shorthand for indicating that a piece of code should asynchronously wait on some other piece of code. The *async* keyword represents a hint that you can use to mark methods as task-based asynchronous methods.

The combination of *await*, *async*, and the *Task* object makes it much easier for you to write asynchronous code in .NET 4.5. ASP.NET 4.5 supports these simplifications with new APIs that let you write asynchronous HTTP modules and asynchronous HTTP handlers using the new compiler enhancements.

Asynchronous HTTP modules

Suppose that you want to perform asynchronous work within a method that returns a *Task* object. The following code example defines an asynchronous method that makes an asynchronous call to download the Microsoft home page. Notice the use of the *async* keyword in the method signature and the *await* call to *DownloadStringTaskAsync*.

```
private async Task
ScrapeHtmlPage(object caller, EventArgs e)
{
    WebClient wc = new WebClient();
    var result = await wc.DownloadStringTaskAsync("http://www.microsoft.com");
    // Do something with the result
}
```

That's all you have to write — the .NET Framework will automatically handle unwinding the call stack while waiting for the download to complete, as well as automatically restoring the call stack after the download is done.

Now suppose that you want to use this asynchronous method in an asynchronous ASP.NET HTTP module. ASP.NET 4.5 includes a helper method (*EventHandlerTaskAsyncHelper*) and a new delegate type (*TaskEventHandler*) that you can use to integrate task-based asynchronous methods with the older asynchronous programming model exposed by the ASP.NET HTTP pipeline. This example shows how:

```
public void Init(HttpApplication
context)
 {
   // Wrap the Task-based method so that it can be used with
   // the older async programming model.
   EventHandlerTaskAsyncHelper helper =
       new EventHandlerTaskAsyncHelper(ScrapeHtmlPage);

   // The helper object makes it easy to extract Begin/End methods out of
   // a method that returns a Task object. The ASP.NET pipeline calls the
   // Begin and End methods to start and complete calls on asynchronous
   // HTTP modules.
   context.AddOnPostAuthorizeRequestAsync(
       helper.BeginEventHandler, helper.EndEventHandler);
}
```

Asynchronous HTTP handlers

The traditional approach to writing asynchronous handlers in ASP.NET is to implement the *IHttpAsyncHandler* interface. ASP.NET 4.5 introduces the *HttpTaskAsyncHandler* asynchronous base type that you can derive from, which makes it much easier to write asynchronous handlers.

The *HttpTaskAsyncHandler* type is abstract and requires you to override the *ProcessRequestAsync* method. Internally ASP.NET takes care of integrating the return signature (a *Task* object) of *ProcessRequestAsync* with the older asynchronous programming model used by the ASP.NET pipeline.

The following example shows how you can use *Task* and *await* as part of the implementation of an asynchronous HTTP handler:

```
public class MyAsyncHandler : HttpTaskAsyncHandler
{
    // ...

    // ASP.NET automatically takes care of integrating the Task based override
    // with the ASP.NET pipeline.
    public override async Task ProcessRequestAsync(HttpContext context)
    {
        WebClient wc = new WebClient();
        var result = await
           wc.DownloadStringTaskAsync("http://www.microsoft.com");
        // Do something with the result
    }
}
```

New ASP.NET Request Validation Features

By default, ASP.NET performs request validation — it examines requests to look for markup or script in fields, headers, cookies, and so on. If any is detected, ASP.NET throws an exception. This acts as a first line of defense against potential cross-site scripting attacks.

ASP.NET 4.5 makes it easy to selectively read unvalidated request data. ASP.NET 4.5 also integrates the popular AntiXSS library, which was formerly an external library.

Developers have frequently asked for the ability to selectively turn off request validation for their applications. For example, if your application is forum software, you might want to allow users to submit HTML-formatted forum posts and comments, but still make sure that request validation is checking everything else.

ASP.NET 4.5 introduces two features that make it easy for you to selectively work with unvalidated input: deferred ("lazy") request validation and access to unvalidated request data.

Deferred ("lazy") request validation

In ASP.NET 4.5, by default all request data is subject to request validation. However, you can configure the application to defer request validation until you actually access request data. (This is sometimes referred to as lazy request validation, based on terms like lazy loading for certain data scenarios.) You can configure the application to use deferred validation in the Web.config file by setting the *requestValidationMode* attribute to 4.5 in the *httpRUntime* element, as in the following example:

```
<httpRuntime requestValidationMode="4.5" ... />
```

When request validation mode is set to 4.5, request validation is triggered only for a specific request value and only when your code accesses that value. For example, if your code gets the value of Request.Form["forum_post"], request validation is invoked only for that element in the form collection. None of the other elements in the *Form* collection are validated. In previous versions of ASP.NET, request validation was triggered for the entire request collection when any element in the collection was accessed. The new behavior makes it easier for different application components to look at different pieces of request data without triggering request validation on other pieces.

Support for unvalidated requests

Deferred request validation alone doesn't solve the problem of selectively bypassing request validation. The call to Request.Form["forum_post"] still triggers request validation for that specific request value. However, you might want to access this field without triggering validation because you want to allow markup in that field.

To allow this, ASP.NET 4.5 now supports unvalidated access to request data. ASP.NET 4.5 includes a new *Unvalidated* collection property in the *HttpRequest* class. This collection provides access to all of the common values of request data, like *Form*, *QueryString*, *Cookies*, and *Url*.

Using the forum example, to be able to read unvalidated request data, you first need to configure the application to

use the new request validation mode:

```
<httpRuntime requestValidationMode="4.5" ...
/>
```

You can then use the *HttpRequest.Unvalidated* property to read the unvalidated form value:

```
var s = context.Request.Unvalidated.Form["forum_post"];
```

WARNING

Security - *Use unvalidated request data with care!* ASP.NET 4.5 added the unvalidated request properties and collections to make it easier for you to access very specific unvalidated request data. However, you must still perform custom validation on the raw request data to ensure that dangerous text is not rendered to users.

AntiXSS Library

Due to the popularity of the Microsoft AntiXSS Library, ASP.NET 4.5 now incorporates core encoding routines from version 4.0 of that library.

The encoding routines are implemented by the *AntiXssEncoder* type in the new *System.Web.Security.AntiXss* namespace. You can use the *AntiXssEncoder* type directly by calling any of the static encoding methods that are implemented in the type. However, the easiest approach for using the new anti-XSS routines is to configure an ASP.NET application to use the *AntiXssEncoder* class by default. To do this, add the following attribute to the Web.config file:

```
<httpRuntime ...
  encoderType="System.Web.Security.AntiXss.AntiXssEncoder,System.Web, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" />
```

When the *encoderType* attribute is set to use the *AntiXssEncoder* type, all output encoding in ASP.NET automatically uses the new encoding routines.

These are the portions of the external AntiXSS library that have been incorporated into ASP.NET 4.5:

- *HtmlEncode*, *HtmlFormUrlEncode*, and *HtmlAttributeEncode*
- *XmlAttributeEncode* and *XmlEncode*
- *UrlEncode* and *UrlPathEncode* (new)
- *CssEncode*

Support for WebSockets Protocol

WebSockets protocol is a standards-based network protocol that defines how to establish secure, real-time bidirectional communications between a client and a server over HTTP. Microsoft has worked with both the IETF and W3C standards bodies to help define the protocol. The WebSockets protocol is supported by any client (not just browsers), with Microsoft investing substantial resources supporting WebSockets protocol on both client and mobile operating systems.

WebSockets protocol makes it much easier to create long-running data transfers between a client and a server. For example, writing a chat application is much easier because you can establish a true long-running connection between a client and a server. You do not have to resort to workarounds like periodic polling or HTTP long-polling to simulate the behavior of a socket.

ASP.NET 4.5 and IIS 8 include low-level WebSockets support, enabling ASP.NET developers to use managed APIs for asynchronously reading and writing both string and binary data on a WebSockets object. For ASP.NET 4.5, there is

a new *System.Web.WebSockets* namespace that contains types for working with WebSockets protocol.

A browser client establishes a WebSockets connection by creating a DOM *WebSocket* object that points to a URL in an ASP.NET application, as in the following example:

```
socket = new WebSocket("ws://contoso.com/MyWebSocketApplication.ashx");
```

You can create WebSockets endpoints in ASP.NET using any kind of module or handler. In the previous example, an .ashx file was used, because .ashx files are a quick way to create a handler.

According to the WebSockets protocol, an ASP.NET application accepts a client's WebSockets request by indicating that the request should be upgraded from an HTTP GET request to a WebSockets request. Here's an example:

```
HttpContext.Current.AcceptWebSocketRequest(// WebSocket delegate goes here)
```

The *AcceptWebSocketRequest* method accepts a function delegate because ASP.NET unwinds the current HTTP request and then transfers control to the function delegate. Conceptually this approach is similar to how you use *System.Threading.Thread*, where you define a thread-start delegate in which background work is performed.

After ASP.NET and the client have successfully completed a WebSockets handshake, ASP.NET calls your delegate and the WebSockets application starts running. The following code example shows a simple echo application that uses the built-in WebSockets support in ASP.NET:

```
public async Task MyWebSocket(AspNetWebSocketContext context)
{
    WebSocket socket = context.WebSocket;
    while (true)
    {
        ArraySegment<byte> buffer = new ArraySegment<byte>(new byte[1024]);

        // Asynchronously wait for a message to arrive from a client
        WebSocketReceiveResult result =
           await socket.ReceiveAsync(buffer, CancellationToken.None);

        // If the socket is still open, echo the message back to the client
        if (socket.State == WebSocketState.Open)
        {
           string userMessage = Encoding.UTF8.GetString(buffer.Array, 0,
               result.Count);
           userMessage = "You sent: " + userMessage + " at " +
               DateTime.Now.ToLongTimeString();
           buffer = new ArraySegment<byte>(Encoding.UTF8.GetBytes(userMessage));

           // Asynchronously send a message to the client
           await socket.SendAsync(buffer, WebSocketMessageType.Text,
               true, CancellationToken.None);
        }
        else { break; }
    }
}
```

The support in .NET 4.5 for the *await* keyword and asynchronous task-based operations is a natural fit for writing WebSockets applications. The code example shows that a WebSockets request runs completely asynchronously inside ASP.NET. The application waits asynchronously for a message to be sent from a client by calling *await socket.ReceiveAsync*. Similarly, you can send an asynchronous message to a client by calling *await socket.SendAsync*.

In the browser, an application receives WebSockets messages through an *onmessage* function. To send a message from a browser, you call the *send* method of the *WebSocket* DOM type, as shown in this example:

```
// Receive a string message from the server.
socket.onmessage = function(msg)
{
    document.getElementById("serverData").innerHTML = msg.data;
};
// Send a string message from the browser.
socket.send(document.getElementById("msgText"));
```

In the future, we might release updates to this functionality that abstract away some of the low-level coding that is required in this release for WebSockets applications.

Bundling and Minification

Bundling lets you combine individual JavaScript and CSS files into a bundle that can be treated like a single file. Minification condenses JavaScript and CSS files by removing whitespace and other characters that are not required. These features work with Web Forms, ASP.NET MVC, and Web Pages.

Bundles are created using the Bundle class or one of its child classes, ScriptBundle and StyleBundle. After configuring an instance of a bundle, the bundle is made available to incoming requests by simply adding it to a global BundleCollection instance. In the default templates, bundle configuration is performed in a BundleConfig file. This default configuration creates bundles for all of the core scripts and css files used by the templates.

Bundles are referenced from within views by using one of a couple possible helper methods. In order to support rendering different markup for a bundle when in debug vs. release mode, the ScriptBundle and StyleBundle classes have the helper method, Render. When in debug mode, Render will generate markup for each resource in the bundle. When in release mode, Render will generate a single markup element for the entire bundle. Toggling between debug and release mode can be accomplished by modifying the debug attribute of the compilation element in web.config as shown below:

```
<system.web>
 <compilation targetframework="4.5" debug="true" />
 ...</system.web>
```

Additionally, enabling or disabling optimization can be set directly via the BundleTable.EnableOptimizations property.

```
BundleTable.EnableOptimizations = true;
```

When files are bundled, they are first sorted alphabetically (the way they are displayed in **Solution Explorer**). They are then organized so that known libraries and their custom extensions (such as jQuery, MooTools, and Dojo) are loaded first. For example, the final order for the bundling of the Scripts folder as shown above will be:

1. jquery-1.6.2.js
2. jquery-ui.js
3. jquery.tools.js
4. a.js

CSS files are also sorted alphabetically and then reorganized so that reset.css and normalize.css come before any other file. The final sorting of the bundling of the Styles folder shown above will be this:

1. reset.css
2. content.css
3. forms.css
4. globals.css
5. menu.css

6. styles.css

Performance Improvements for Web Hosting

The .NET Framework 4.5 and Windows 8 introduce features that can help you achieve a significant performance boost for web-server workloads. This includes a reduction (up to 35%) in both startup time and in the memory footprint of web hosting sites that use ASP.NET.

Key performance factors

Ideally, all websites should be active and in memory to assure quick response to the next request, whenever it comes. Factors that can affect site responsiveness include:

- The time it takes for a site to restart after an app pool recycles. This is the time it takes to launch a web server process for the site when the site assemblies are no longer in memory. (The platform assemblies are still in memory, since they are used by other sites.) This situation is referred to as "cold site, warm framework startup" or just "cold site startup."
- How much memory the site occupies. Terms for this are "per-site memory consumption" or "unshared working set."

The new performance improvements focus on both of these factors.

Requirements for New Performance Features

The requirements for the new features can be broken down into these categories:

- Improvements that run on the .NET Framework 4.
- Improvements that require the .NET Framework 4.5 but can run on any version of Windows.
- Improvements that are available only with .NET Framework 4.5 running on Windows 8.

Performance increases with each level of improvement that you are able to enable.

Some of the .NET Framework 4.5 improvements take advantage of broader performance features that apply to other scenarios as well.

Sharing Common Assemblies

Requirement: .NET Framework 4 and Visual Studio 11 Developer Preview SDK

Different sites on a server often use the same helper assemblies (for example, assemblies from a starter kit or sample application). Each site has its own copy of these assemblies in its Bin directory. Even though the object code for the assemblies is identical, they're physically separate assemblies, so each assembly has to be read separately during cold site startup and kept separately in memory.

The new interning functionality solves this inefficiency and reduces both RAM requirements and load time. Interning lets Windows keep a single copy of each assembly in the file system, and individual assemblies in the site Bin folders are replaced with symbolic links to the single copy. If an individual site needs a distinct version of the assembly, the symbolic link is replaced by the new version of the assembly, and only that site is affected.

Sharing assemblies using symbolic links requires a new tool named aspnet_intern.exe, which lets you create and manage the store of interned assemblies. It is provided as a part of the Visual Studio 11 Developer Preview SDK. (However, it will work on a system that has only the .NET Framework 4 installed, assuming you have installed the latest update.)

To make sure all eligible assemblies have been interned, you run aspnet_intern.exe periodically (for example, once a week as a scheduled task). A typical use is as follows:

```
aspnet_intern -mode exec -sourcedir
"C:\Windows\Microsoft.NET\Framework\v4.0.30319\Temporary ASP.NET Files" -interndir C:\ASPNETCommonAssemblies
```

To see all options, run the tool with no arguments.

Using multi-Core JIT compilation for faster startup

Requirement: .NET Framework 4.5

For a cold site startup, not only do assemblies have to be read from disk, but the site must be JIT-compiled. For a complex site, this can add significant delays. A new general-purpose technique in the .NET Framework 4.5 reduces these delays by spreading JIT-compilation across available processor cores. It does this as much and as early as possible by using information gathered during previous launches of the site. This functionality implemented by the System.Runtime.ProfileOptimization.StartProfile method.

JIT-compiling using multiple cores is enabled by default in ASP.NET, so you do not need to do anything to take advantage of this feature. If you want to disable this feature, make the following setting in the Web.config file:

```
<configuration>
  <!-- ... -->
  <system.web>
<compilation profileGuidedOptimizations="None"  />
  <!-- ... -->
```

Tuning garbage collection to optimize for memory

Requirement: .NET Framework 4.5

Once a site is running, its use of the garbage-collector (GC) heap can be a significant factor in its memory consumption. Like any garbage collector, the .NET Framework GC makes tradeoffs between CPU time (frequency and significance of collections) and memory consumption (extra space that is used for new, freed, or free-able objects). For previous releases, we have provided guidance on how to configure the GC to achieve the right balance (for example, see ASP.NET 2.0/3.5 Shared Hosting Configuration).

For the .NET Framework 4.5, instead of multiple standalone settings, a workload-defined configuration setting is available that enables all of the previously recommended GC settings as well as new tuning that delivers additional performance for the per-site working set.

To enable GC memory tuning, add the following setting to the Windows\Microsoft.NET\Framework\v4.0.30319\aspnet.config file:

```
<configuration>
  <!-- ... -->
  <runtime>
<performanceScenario value="HighDensityWebHosting"  />
  <!-- ... -->
```

(If you're familiar with the previous guidance for changes to aspnet.config, note that this setting replaces the old settings — for example, there is no need to set gcServer, gcConcurrent, etc. You do not have to remove the old settings.)

Prefetching for web applications

Requirement: .NET Framework 4.5 running on Windows 8

For several releases, Windows has included a technology known as the prefetcher that reduces the disk-read cost of application startup. Because cold startup is a problem predominantly for client applications, this technology has not been included in Windows Server, which includes only components that are essential to a server. Prefetching is now available in the latest version of Windows Server, where it can optimize the launch of individual websites.

For Windows Server, the prefetcher is not enabled by default. To enable and configure the prefetcher for high-density web hosting, run the following set of commands at the command line:

```
sc config sysmain start=auto
reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory
Management\PrefetchParameters" /v EnablePrefetcher /t REG_DWORD /d 2 /f
reg add "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Prefetcher" /v MaxPrefetchFiles /t
REG_DWORD /d 8192 /f
net start sysmain
```

Then, to integrate the prefetcher with ASP.NET applications, add the following to the Web.config file:

```
<configuration>
  <!-- ... -->
  <system.web>
<compilation enablePrefetchOptimization="true" />
  <!-- ... -->
```

ASP.NET Web Forms

Strongly Typed Data Controls

In ASP.NET 4.5, Web Forms includes some improvements for working with data. The first improvement is strongly typed data controls. For Web Forms controls in previous versions of ASP.NET, you display a data-bound value using *Eval* and a data-binding expression:

```
<ul>
<asp:Repeater runat="server" ID="customers">
       <ItemTemplate>
           <li>
               First Name: <%# Eval("FirstName")%><br />
               Last Name: <%# Eval("LastName")%><br />
           </li>
       </ItemTemplate>
</asp:Repeater>
</ul>
```

For two-way data binding, you use *Bind*:

```
<asp:FormView runat="server" ID="editCustomer">
<EditItemTemplate>
       <div>
           <asp:Label runat="server" AssociatedControlID="firstName">
               First Name:</asp:Label>
           <asp:TextBox ID="firstName" runat="server"
               Text='<%#Bind("FirstName") %>' />
       </div>
       <div>
           <asp:Label runat="server" AssociatedControlID="lastName">
               First Name:</asp:Label>
           <asp:TextBox ID="lastName" runat="server"
               Text='<%#
Bind("LastName") %>' />
       </div>
       <asp:Button runat="server" CommandName="Update"/>
</EditItemTemplate>
</asp:FormView>
```

At run time, these calls use reflection to read the value of the specified member and then display the result in the markup. This approach makes it easy to data bind against arbitrary, unshaped data.

However, data-binding expressions like this don't support features like IntelliSense for member names, navigation

(like Go To Definition), or compile-time checking for these names.

To address this issue, ASP.NET 4.5 adds the ability to declare the data type of the data that a control is bound to. You do this using the new *ItemType* property. When you set this property, two new typed variables are available in the scope of data-binding expressions: *Item* and *BindItem*. Because the variables are strongly typed, you get the full benefits of the Visual Studio development experience.

For two-way data-binding expressions, use the *BindItem* variable:

```
<asp:FormView runat="server" ID="editCustomer">
<EditItemTemplate>
       <div>
           <asp:Label runat="server" AssociatedControlID="firstName">
               First Name:</asp:Label>
           <asp:TextBox ID="firstName" runat="server"
               Text='<%#BindItem.FirstName %>' />
       </div>
       <div>
           <asp:Label runat="server" AssociatedControlID="lastName">
               First Name:</asp:Label>
           <asp:TextBox ID="lastName" runat="server"
               Text='<%#BindItem.LastName %>' />
       </div>
       <asp:Button runat="server" CommandName="Update"/>
</EditItemTemplate>
</asp:FormView>
```

Most controls in the ASP.NET Web Forms framework that support data binding have been updated to support the *ItemType* property.

Model Binding

Model binding extends data binding in ASP.NET Web Forms controls to work with code-focused data access. It incorporates concepts from the *ObjectDataSource* control and from model binding in ASP.NET MVC.

Selecting data

To configure a data control to use model binding to select data, you set the control's *SelectMethod* property to the name of a method in the page's code. The data control calls the method at the appropriate time in the page life cycle and automatically binds the returned data. There's no need to explicitly call the *DataBind* method.

In the following example, the *GridView* control is configured to use a method named *GetCategories*:

```
<asp:GridView ID="categoriesGrid"
runat="server"
ItemType="WebApplication1.Model.Category"
SelectMethod="GetCategories" AutoGenerateColumns="false">
<Columns>
       <asp:BoundField DataField="CategoryID" HeaderText="ID" />
       <asp:BoundField DataField="CategoryName" HeaderText="Name" />
       <asp:BoundField DataField="Description" HeaderText="Description" />
       <asp:TemplateField HeaderText="# of Products">
           <ItemTemplate><%# Item.Products.Count %></ItemTemplate>
       </asp:TemplateField>
</Columns>
</asp:GridView>
```

You create the *GetCategories* method in the page's code. For a simple select operation, the method needs no parameters and should return an *IEnumerable* or *IQueryable* object. If the new *ItemType* property is set (which enables strongly typed data-binding expressions, as explained under Strongly Typed Data Controls earlier), the generic versions of these interfaces should be returned — *IEnumerable<T>* or *IQueryable<T>*, with the *T* parameter matching the type of the *ItemType* property (for example, *IQueryable<Category>*).

The following example shows the code for a *GetCategories* method. This example uses the Entity Framework Code First model with the Northwind sample database. The code makes sure that the query returns details of the related products for each category by way of the *Include* method. (This ensures that the *TemplateField* element in the markup displays the count of products in each category without requiring an n+1 select.)

```
public IQueryable<Category>
GetCategories()
{
    var db = new Northwind();
    return db.Categories.Include(c => c.Products);
}
```

When the page runs, the *GridView* control calls the *GetCategories* method automatically and renders the returned data using the configured fields:

ID	Name	Description	# of Products
1	Beverages	Soft drinks, coffees, teas, beers, and ales	13
2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings	12
3	Confections	Desserts, candies, and sweet breads	13
4	Dairy Products	Cheeses	10
5	Grains/Cereals	Breads, crackers, pasta, and cereal	7
6	Meat/Poultry	Prepared meats	6
7	Produce	Dried fruit and bean curd	5
8	Seafood	Seaweed and fish	12

Because the select method returns an *IQueryable* object, the *GridView* control can further manipulate the query before executing it. For example, the *GridView* control can add query expressions for sorting and paging to the returned *IQueryable* object before it is executed, so that those operations are performed by the underlying LINQ provider. In this case, Entity Framework will ensure those operations are performed in the database.

The following example shows the *GridView* control modified to allow sorting and paging:

```
<asp:GridView ID="categoriesGrid"
runat="server"
AutoGenerateColumns="false"
AllowSorting="true" AllowPaging="true" PageSize="5"
ItemType="WebApplication1.Model.Category" DataKeyNames="CategoryID"
SelectMethod="GetCategories"
UpdateMethod="UpdateCategory">
<Columns>
       <asp:BoundField DataField="CategoryID" HeaderText="ID" SortExpression="CategoryID" />
       <asp:BoundField DataField="CategoryName" HeaderText="Name" SortExpression="CategoryName" />
       <asp:BoundField DataField="Description" HeaderText="Description" />
       <asp:TemplateField HeaderText="# of Products">
           <ItemTemplate><%# Item.Products.Count %></ItemTemplate>
       </asp:TemplateField>
</Columns>
<EmptyDataTemplate>No categories found with a product count of
      <%# minProductsCount.SelectedValue %></EmptyDataTemplate>
</asp:GridView>
```

Now when the page runs, the control can make sure that only the current page of data is displayed and that it's ordered by the selected column:

ID	Name	Description	# of Products
3	Confections	Desserts, candies, and sweet breads	13
2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings	12
1	Beverages	Soft drinks, coffees, teas, beers, and ales	13
1 2			

To filter the returned data, parameters have to be added to the select method. These parameters will be populated by the model binding at run time, and you can use them to alter the query before returning the data.

For example, assume that you want to let users filter products by entering a keyword in the query string. You can add a parameter to the method and update the code to use the parameter value:

```
public IQueryable<Product>
GetProducts(string keyword)
{
    IQueryable<Product> query = _db.Products;

    if (!String.IsNullOrWhiteSpace(keyword))
    {
        query = query.Where(p => p.ProductName.Contains(keyword));
    }
    return query;
}
```

This code includes a *Where* expression if a value is provided for *keyword* and then returns the query results.

Value providers

The previous example was not specific about where the value for the *keyword* parameter was coming from. To indicate this information, you can use a parameter attribute. For this example, you can use the *QueryStringAttribute* class that's in the *System.Web.ModelBinding* namespace:

```
public IQueryable<Product>
GetProducts([QueryString]string keyword)
{
    IQueryable<Product> query = _db.Products;

    if (!String.IsNullOrWhiteSpace(keyword))
    {
        query = query.Where(p => p.ProductName.Contains(keyword));
    }
    return query;
}
```

This instructs model binding to try to bind a value from the query string to the *keyword* parameter at run time. (This might involve performing type conversion, although it doesn't in this case.) If a value cannot be provided and the type is non-nullable, an exception is thrown.

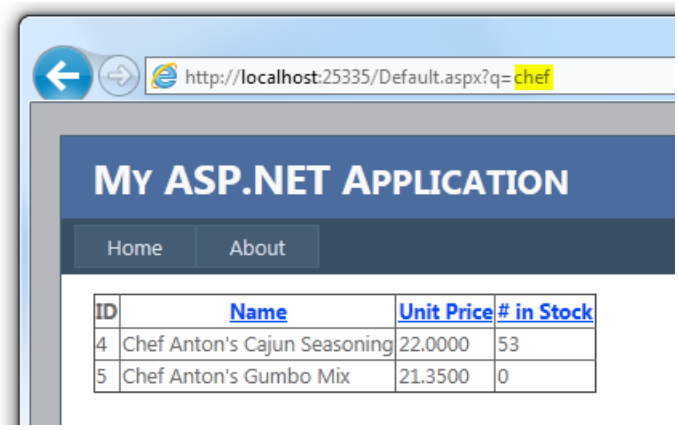
The sources of values for these methods are referred to as value providers, and the parameter attributes that indicate which value provider to use are referred to as value provider attributes. Web Forms will include value providers and corresponding attributes for all of the typical sources of user input in a Web Forms application, such as the query string, cookies, form values, controls, view state, session state, and profile properties. You can also write custom value providers.

By default, the parameter name is used as the key to find a value in the value provider collection. In the example, the code will look for a query-string value named keyword (for example, ~/default.aspx?keyword=chef). You can specify a custom key by passing it as an argument to the parameter attribute. For example, to use the value of the query-string variable named q, you could do this:

```
public IQueryable<Product>
GetProducts([QueryString("q")]string keyword)
{
    IQueryable<Product> query = _db.Products;

    if (!String.IsNullOrWhiteSpace(keyword))
    {
       query = query.Where(p => p.ProductName.Contains(keyword));
    }
    return query;
}
```

If this method is in the page's code, users can filter the results by passing a keyword using the query string:



Model binding accomplishes many tasks that you would otherwise have to code by hand: reading the value, checking for a null value, attempting to convert it to the appropriate type, checking whether the conversion was successful, and finally, using the value in the query. Model binding results in far less code and in the ability to reuse the functionality throughout your application.

Filtering by values from a control

Suppose you want to extend the example to let the user choose a filter value from a drop-down list. Add the following drop-down list to the markup and configure it to get its data from another method using the *SelectMethod* property:

```
<asp:Label runat="server" AssociatedControlID="categories"
Text="Select a category to show products for: " />
<asp:DropDownList runat="server" ID="categories"
SelectMethod="GetCategories" AppendDataBoundItems="true"
DataTextField="CategoryName" DataValueField="CategoryID"
AutoPostBack="true">
  <asp:ListItem Value="" Text="- all -" />
</asp:DropDownList>
```

Typically you would also add an *EmptyDataTemplate* element to the *GridView* control so that the control will display a message if no matching products are found:

```
<asp:GridView ID="productsGrid"
runat="server" DataKeyNames="ProductID"
AllowPaging="true" AllowSorting="true" AutoGenerateColumns="false"
SelectMethod="GetProducts" >
<Columns>
       <asp:BoundField DataField="ProductID" HeaderText="ID" />
       <asp:BoundField DataField="ProductName" HeaderText="Name"
            SortExpression="ProductName" />
       <asp:BoundField DataField="UnitPrice" HeaderText="Unit Price"
            SortExpression="UnitPrice" />
       <asp:BoundField DataField="UnitsInStock" HeaderText="# in Stock"
            SortExpression="UnitsInStock" />
</Columns>
<EmptyDataTemplate>
        No products matching the filter criteria were found</EmptyDataTemplate>
</asp:GridView>
```

In the page code, add the new select method for the drop-down list:

```
public IQueryable<Category>
GetCategories()
{
    return _db.Categories;
}
```

Finally, update the *GetProducts* select method to take a new parameter that contains the ID of the selected category from the drop-down list:

```
public IQueryable<Product>
GetProducts(
[QueryString("q")] string keyword,
[Control("categories")] int? categoryId)
{
    IQueryable<Product> query = _db.Products;

    if (!String.IsNullOrWhiteSpace(keyword))
    {
        query = query.Where(p => p.ProductName.Contains(keyword));
    }
    if (categoryId.HasValue && categoryId > 0)
    {
        query = query.Where(p => p.CategoryID == categoryId);
    }
    return query;
}
```

Now when the page runs, users can select a category from the drop-down list, and the *GridView* control is automatically re-bound to show the filtered data. This is possible because model binding tracks the values of parameters for select methods and detects whether any parameter value has changed after a postback. If so, model binding forces the associated data control to re-bind to the data.

Select a category to show products for: Confections

ID	Name	Unit Price	# in Stock
16	Pavlova	17.4500	29
19	Teatime Chocolate Biscuits	9.2000	25
20	Sir Rodney's Marmalade	81.0000	40
21	Sir Rodney's Scones	10.0000	3
25	NuNuCa Nuß-Nougat-Creme	14.0000	76
26	Gumbär Gummibärchen	31.2300	15
27	Schoggi Schokolade	43.9000	49
47	Zaanse koeken	9.5000	36
48	Chocolade	12.7500	15
49	Maxilaku	20.0000	10
1 2			

HTML Encoded Data-Binding Expressions

You can now HTML-encode the result of data-binding expressions. Add a colon (:) to the end of the <%# prefix that marks the data-binding expression:

```
<asp:TemplateField HeaderText="Name">
<ItemTemplate><%#: Item.Products.Name %></ItemTemplate>
</asp:TemplateField>
```

Unobtrusive Validation

You can now configure the built-in validator controls to use unobtrusive JavaScript for client-side validation logic. This significantly reduces the amount of JavaScript rendered inline in the page markup and reduces the overall page size. You can configure unobtrusive JavaScript for validator controls in any of these ways:

- Globally by adding the following setting to the *<appSettings>* element in the Web.config file:

  ```
  <add name="ValidationSettings:UnobtrusiveValidationMode" value="WebForms" />
  ```

- Globally by setting the static *System.Web.UI.ValidationSettings.UnobtrusiveValidationMode* property to *UnobtrusiveValidationMode.WebForms* (typically in the *Application_Start* method in the Global.asax file).
- Individually for a page by setting the new *UnobtrusiveValidationMode* property of the *Page* class to *UnobtrusiveValidationMode.WebForms*.

HTML5 Updates

Some improvements have been made to Web Forms server controls to take advantage of new features of HTML5:

- The *TextMode* property of the *TextBox* control has been updated to support the new HTML5 input types like *email*, *datetime*, and so on.
- The *FileUpload* control now supports multiple file uploads from browsers that support this HTML5 feature.
- Validator controls now support validating HTML5 input elements.
- New HTML5 elements that have attributes that represent a URL now support runat="server". As a result, you can use ASP.NET conventions in URL paths, like the ~ operator to represent the application root (for example, <video runat="server" src="~/myVideo.wmv" />).
- The *UpdatePanel* control has been fixed to support posting HTML5 input fields.

ASP.NET MVC 4

ASP.NET MVC 4 Beta is now included with Visual Studio 11 Beta. ASP.NET MVC is a framework for developing highly testable and maintainable Web applications by leveraging the Model-View-Controller (MVC) pattern. ASP.NET MVC 4 makes it easy to build applications for the mobile Web and includes ASP.NET Web API, which helps

you build HTTP services that can reach any device. For more information, see the ASP.NET MVC 4 Release Notes.

ASP.NET Web Pages 2

New features include the following:

- New and updated site templates.
- Adding server-side and client-side validation using the *Validation* helper.
- The ability to register scripts using an assets manager.
- Enabling logins from Facebook and other sites using OAuth and OpenID.
- Adding maps using the *Maps* helper.
- Running Web Pages applications side-by-side.
- Rendering pages for mobile devices.

For more information about these features and full-page code examples, see The Top Features in Web Pages 2 Beta.

Visual Web Developer 11 Beta

This section provides information about improvements for web development in Visual Web Developer 11 Beta and Visual Studio 2012 Release Candidate.

Project Sharing Between Visual Studio 2010 and Visual Studio 2012 Release Candidate (Project Compatibility)

Until Visual Studio 2012 Release Candidate, opening an existing project in a newer version of Visual Studio launched the Conversion Wizard. This forced an upgrade of the content (assets) of a project and solution to new formats that were not backward compatible. Therefore, after the conversion you could not open the project in the older version of Visual Studio.

Many customers have told us that this was not the right approach. In Visual Studio 11 Beta, we now support sharing projects and solutions with Visual Studio 2010 SP1. This means that if you open a 2010 project in Visual Studio 2012 Release Candidate, you will still be able to open the project in Visual Studio 2010 SP1.

> **NOTE**
> A few types of projects cannot be shared between Visual Studio 2010 SP1 and Visual Studio 2012 Release Candidate. These include some older projects (such as ASP.NET MVC 2 projects) or projects for special purposes (such as Setup projects).

When you open a Visual Studio 2010 SP1 Web project for the first time in Visual Studio 11 Beta, the following properties are added to the project file:

- FileUpgradeFlags
- UpgradeBackupLocation
- OldToolsVersion
- VisualStudioVersion
- VSToolsPath

FileUpgradeFlags, UpgradeBackupLocation, and OldToolsVersion are used by the process that upgrades the project file. They have no impact on working with the project in Visual Studio 2010.

VisualStudioVersion is a new property used by MSBuild 4.5 that indicates the version of Visual Studio for the current project. Because this property didn't exist in MSBuild 4.0 (the version of MSBuild that Visual Studio 2010 SP1 uses), we inject a default value into the project file.

The VSToolsPath property is used to determine the correct .targets file to import from the path represented by the MSBuildExtensionsPath32 setting.

There are also some changes related to Import elements. These changes are required in order to support compatibility between both versions of Visual Studio.

> **NOTE**
> If a project is being shared between Visual Studio 2010 SP1 and Visual Studio 11 Beta on two different computers, and if the project includes a local database in the App_Data folder, you must make sure that the version of SQL Server used by the database is installed on both computers.

Configuration Changes in ASP.NET 4.5 Website Templates

The following changes have been made to the default *Web.config* file for site that are created using website templates in Visual Studio 2012 Release Candidate:

- In the `<httpRuntime>` element, the `encoderType` attribute is now set by default to use the AntiXSS types that were added to ASP.NET. For details, see AntiXSS Library.
- Also in the `<httpRuntime>` element, the `requestValidationMode` attribute is set to "4.5". This means that by default, request validation is configured to use deferred ("lazy") validation. For details, see New ASP.NET Request Validation Features.
- The `<modules>` element of the `<system.webServer>` section does not contain a `runAllManagedModulesForAllRequests` attribute. (Its default value is false.) This means that if you are using a version of IIS 7 that has not been updated to SP1, you might have issues with routing in a new site. For more information, see Native Support in IIS 7 for ASP.NET Routing.

These changes do not affect existing applications. However, they might represent a difference in behavior between existing websites and new websites that you create for ASP.NET 4.5 using the new templates.

Native Support in IIS 7 for ASP.NET Routing

This is not a change to ASP.NET as such, but a change in templates for new website projects that can affect you if you are working a version of IIS 7 that has not had the SP1 update applied.

In ASP.NET, you can add the following configuration setting to applications in order to support routing:

```
<configuration>
  <system.webServer>
<modules runAllManagedModulesForAllRequests="true">
     <!-- more -->
</modules>
  </system.webServer>
</configuration>
```

When **runAllManagedModulesForAllRequests** is true, a URL like `http://mysite/myapp/home` goes to ASP.NET, even though there is no *.aspx*, *.mvc*, or similar extension on the URL.

An update that was made to IIS 7 makes the **runAllManagedModulesForAllRequests** setting unnecessary and supports ASP.NET routing natively. (For information about the update, see the Microsoft Support article An update is available that enables certain IIS 7.0 or IIS 7.5 handlers to handle requests whose URLs do not end with a period.)

If your website is running on IIS 7 and if IIS has been updated, you do not need to set **runAllManagedModulesForAllRequests** to true. In fact, setting it to true is not recommended, because it adds unnecessary processing overhead to request. When this setting is true, all requests, including those for *.htm*, *.jpg*, and other static files, also go through the ASP.NET request pipeline.

If you create a new ASP.NET 4.5 website using the templates that are provided in Visual Studio 2012 RC, the configuration for the website does not include the **runAllManagedModulesForAllRequests** setting. This means that by default the setting is false.

If you then run the website on Windows 7 without SP1 installed, IIS 7 will not include the required update. As a consequence, routing will not work and you will see errors. If you have a problem where routing does not work, you can do either the following:

- Update Windows 7 to SP1, which will add the update to IIS 7.
- Install the update that's described in the Microsoft Support article listed previously.
- Set **runAllManagedModulesForAllRequests** to true in that website's Web.config file. Note that this will add some overhead to requests.

HTML Editor

Smart Tasks

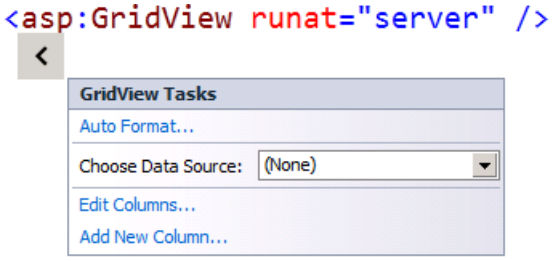
In Design view, complex properties of server controls often have associated dialog boxes and wizards to make it easy to set them. For example, you can use a special dialog box to add a data source to a *Repeater* control or add columns to a *GridView* control.

However, this type of UI help for complex properties has not been available in Source view. Therefore, Visual Studio 11 introduces Smart Tasks for Source view. Smart Tasks are context-aware shortcuts for commonly used features in the C# and Visual Basic editors.

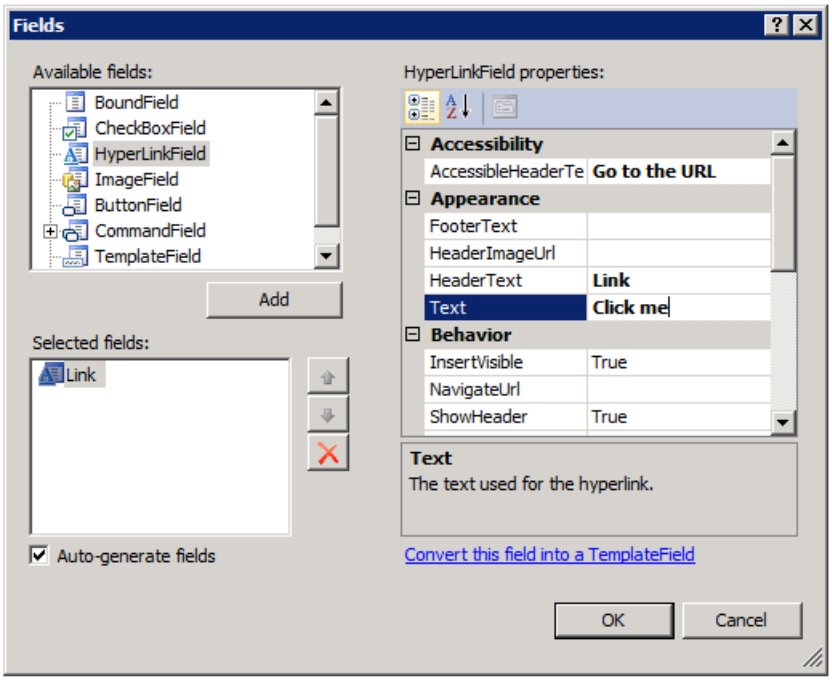
For ASP.NET Web Forms controls, Smart Tasks appear on server tags as a small glyph when the insertion point is inside the element:

```
<asp:GridView runat="server" />
```

The Smart Task expands when you click the glyph or press CTRL+. (dot), just as in the code editors. It then displays shortcuts that are similar to the Smart Tasks in Design view.



For example, the Smart Task in the previous illustration shows the GridView Tasks options. If you choose Edit Columns, the following dialog box is displayed:



Filling in the dialog box sets the same properties you can set in Design view. When you click OK, the markup for the control is updated with the new settings:

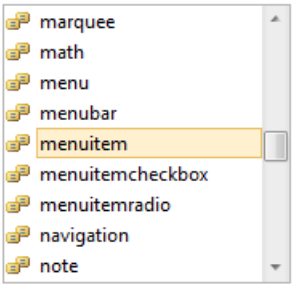
```
<asp:GridView runat="server">
    <Columns>
        <asp:HyperLinkField Target="_blank"
            Text="Click me"
            AccessibleHeaderText="Go to the URL"
            HeaderText="Link"></asp:HyperLinkField>
    </Columns>
</asp:GridView>
```

WAI-ARIA support

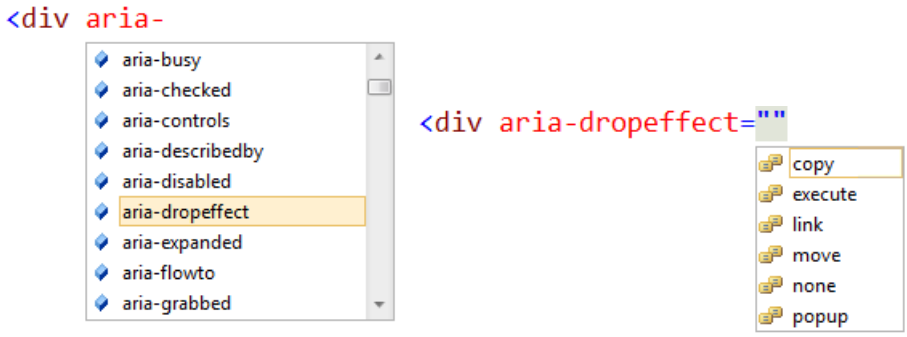
Writing accessible websites is becoming increasingly important. The WAI-ARIA accessibility standard defines how developers should write accessible websites. This standard is now fully supported in Visual Studio.

For example, the *role* attribute now has full IntelliSense:

```
<nav role="menu">
    <ul>
        <li role="menuitem"></li>
    </ul>
</nav>
```

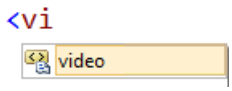


The WAI-ARIA standard also introduces attributes that are prefixed with *aria-* that let you add semantics to an HTML5 document. Visual Studio also fully supports these *aria-* attributes:



New HTML5 snippets

To make it faster and easier to write commonly used HTML5 markup, Visual Studio includes a number of snippets. An example is the video snippet:



To invoke the snippet, press Tab twice when the element is selected in IntelliSense:

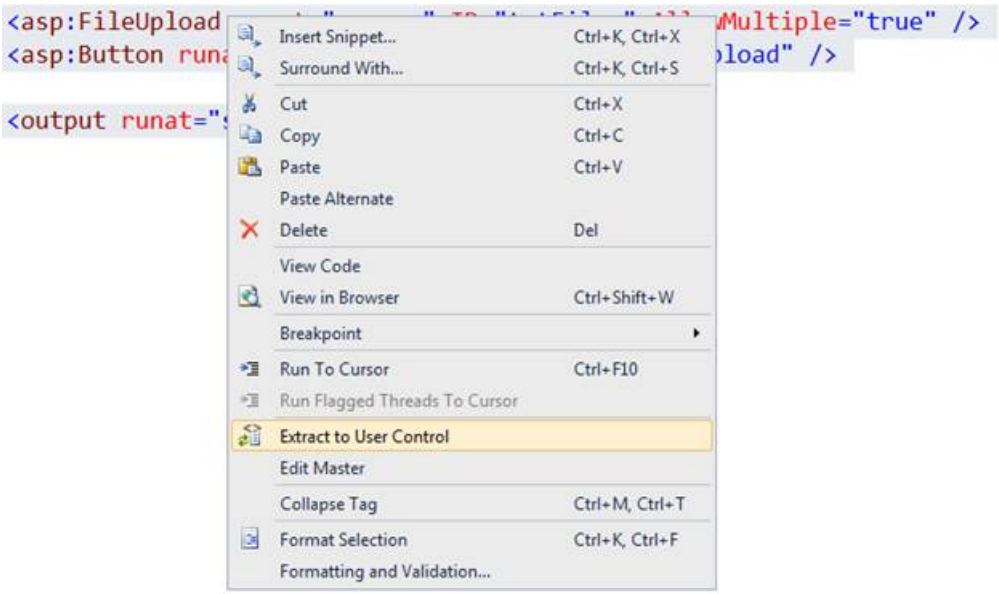
```
<video controls="controls">
    <source src="file.mp4" type="video/mp4" />
    <source src="file.webm" type="video/webm" />
    <source src="file.ogv" type="video/ogg" />
</video>
```

This produces a snippet that you can customize.

Extract to user control

In large web pages, it can be a good idea to move individual pieces into user controls. This form of refactoring can help increase the readability of the page and can simplify the page structure.

To make this easier, when you edit Web Forms pages in Source view, you can now select text in a page, right-click it, and then choose Extract to User Control:



IntelliSense for code nuggets in attributes

Visual Studio has always provided IntelliSense for server-side code nuggets in any page or control. Now Visual Studio includes IntelliSense for code nuggets in HTML attributes as well.

```
<div class="<%= Request.q %>"
```

Path
PathInfo
PhysicalApplicationPath
PhysicalPath
QueryString
RawUrl
ReadEntityBodyMode
RequestContext
RequestType

This makes it easier to create data-binding expressions:

```
<asp:repeater runat="server">
    <itemtemplate>
        <input value="<%# HttpUtility. %>"
    </itemtemplate>
</asp:repeater>
```

Equals
HtmlAttributeEncode
HtmlDecode
HtmlEncode
JavaScriptStringEncode
ParseQueryString
ReferenceEquals
UrlDecode
UrlDecodeToBytes

Automatic renaming of matching tag when you rename an opening or closing tag

If you rename an HTML element (for example, you change a *div* tag to be a *header* tag), the corresponding opening or closing tag also changes in real time.

```
<heade>
    <hgroup>
        <h1>Coffee Shop</h1>
        <h2>Best coffee in town!</h2>
    </hgroup>
</heade>
```

This helps avoid the error where you forget to change a closing tag or change the wrong one.

Event handler generation

Visual Studio now includes features in Source view to help you write event handlers and bind them manually. If you are editing an event name in Source view, IntelliSense displays <Create New Event>, which will create an event handler in the page's code that has the right signature:

```
<asp:Button ID="btnSave" runat="server" OnClick="" />
```

<Create New Event>

By default, the event handler will use the control's ID for the name of the event-handling method:

```
<asp:Button ID="btnSave" runat="server" OnClick="btnSave_Click" />
```

The resulting event handler will look like this (in this case, in C#):

```
protected void btnSave_Click(object sender, EventArgs e)
{

}
```

Smart indent

When you press Enter while inside an empty HTML element, the editor will put the insertion point in the right

place:

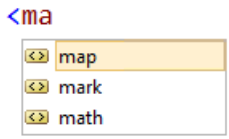
```
<div></div>
```

If you press Enter in this location, the closing tag is moved down and indented to match the opening tag. The insertion point is also indented:

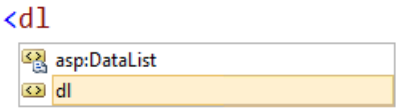
```
<div>
    |
</div>
```

Auto-reduce statement completion

The IntelliSense list in Visual Studio now filters based on what you type so that it displays only relevant options:



IntelliSense also filters based on the title case of the individual words in the IntelliSense list. For example, if you type "dl", both dl and asp:DataList are displayed:



This feature makes it faster to get statement completion for known elements.

JavaScript Editor

The JavaScript editor in Visual Studio 2012 Release Candidate is completely new and it greatly improves the experience of working with JavaScript in Visual Studio.

Code outlining

Outlining regions are now automatically created for all functions, allowing you to collapse parts of the file that aren't pertinent to your current focus.

Brace matching

When you put the insertion point on an opening or closing brace, the editor highlights the matching one.

Go to Definition

The Go to Definition command lets you jump to the source for a function or variable.

ECMAScript5 support

The editor supports the new syntax and APIs in ECMAScript5, the latest version of the standard that describes the JavaScript language.

DOM IntelliSense

IntelliSense for DOM APIs has been improved, with support for many new HTML5 APIs including *querySelector*, DOM Storage, cross-document messaging, and *canvas*. DOM IntelliSense is now driven by a single simple JavaScript file, rather than by a native type library definition. This makes it easy to extend or replace.

VSDOC signature overloads

Detailed IntelliSense comments can now be declared for separate overloads of JavaScript functions by using the new *<signature>* element, as shown in this example:

```
function GetOrSet(key, value) {
/// <signature>
///  <summary>Gets the value</summary>
///  <param name="key" type="String">The key to get the value for</param>
///  <returns type="String" />
/// </signature>
/// <signature>
///  <summary>Sets the value</summary>
///  <param name="key" type="String">The key to set the value for</param>
///  <param name="value" type="String">The value to set</param>
///  <returns type="MyLib" />
/// </signature>
    if (value) {
        values[key] = value;
        return this;
    } else {
        return values[key];
    }
}
```

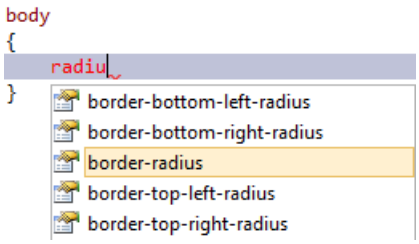
Implicit references

You can now add JavaScript files to a central list that will be implicitly included in the list of files that any given JavaScript file or block references, meaning you'll get IntelliSense for its contents. For example, you can add jQuery files to the central list of files, and you'll get IntelliSense for jQuery functions in any JavaScript block of file, whether you've referenced it explicitly (using /// <reference />) or not.

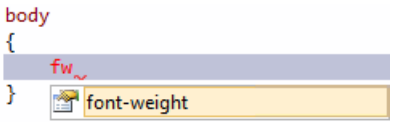
CSS Editor

Auto-reduce statement completion

The IntelliSense list for CSS now filters based on the CSS properties and values supported by the selected schema.



IntelliSense also supports title case searches:



Hierarchical indentation

The CSS editor uses indentation to display hierarchical rules, which gives you an overview of how the cascading rules are logically organized. In the following example, the #list a selector is a cascading child of list and is therefore indented.

```
#list
{
    color: #f00;
}

    #list a
    {
        text-decoration: none;
    }
```

The following example shows more complex inheritance:

```
#list
{
    color: #f00;
}

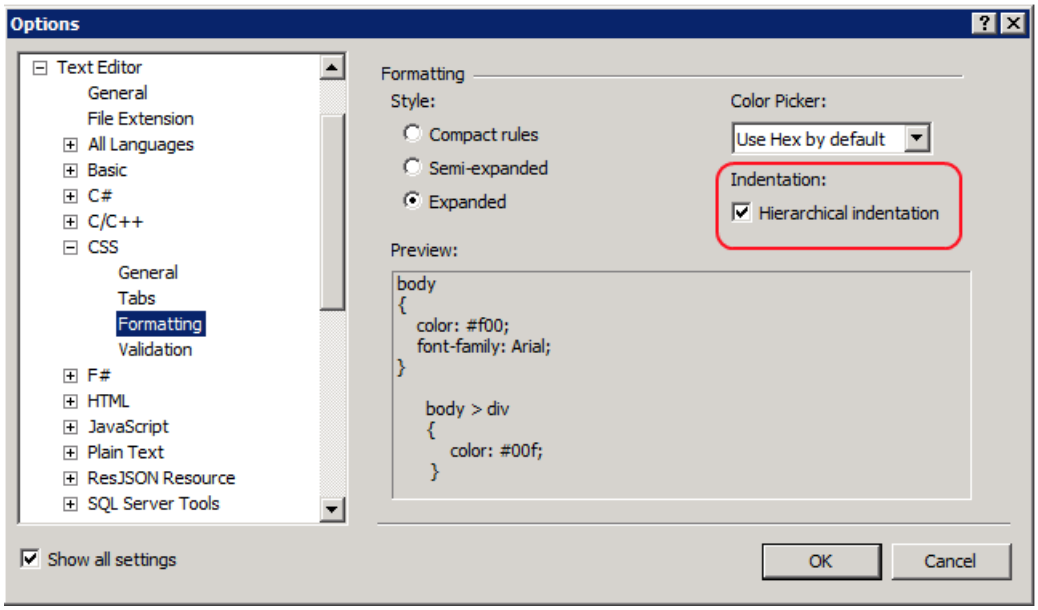
    #list a
    {
        text-decoration: none;
    }

    #list div
    {
        color: #f00;
    }

        #list div a
        {
            text-decoration: none;
        }

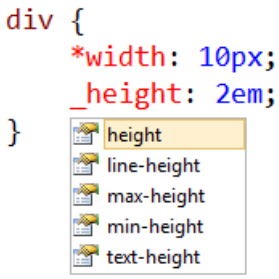
    #list footer
    {
        color: #f00;
    }
```

The indentation of a rule is determined by its parent rules. Hierarchical indentation is enabled by default, but you can disable it the Options dialog box (Tools, Options from the menu bar):



CSS hacks support

Analysis of hundreds of real-world CSS files shows that CSS hacks are very common, and now Visual Studio supports the most widely used ones. This support includes IntelliSense and validation of the star (*) and underscore (_) property hacks:



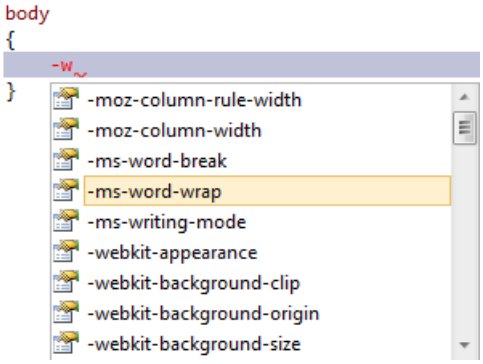
Typical selector hacks are also supported so that hierarchical indentation is maintained even when they are applied. A typical selector hack used to target Internet Explorer 7 is to prepend a selector with **:first-child + html*. Using that rule will maintain the hierarchical indentation:

```
#menu {
    list-style-type: none;
}

    *:first-child + html #menu li {
        margin: 0;
        float: left;
    }
```

Vendor specific schemas (-moz-, -webkit)

CSS3 introduces many properties that have been implemented by different browsers at different times. This previously forced developers to code for specific browsers by using vendor-specific syntax. These browser-specific properties are now included in IntelliSense.



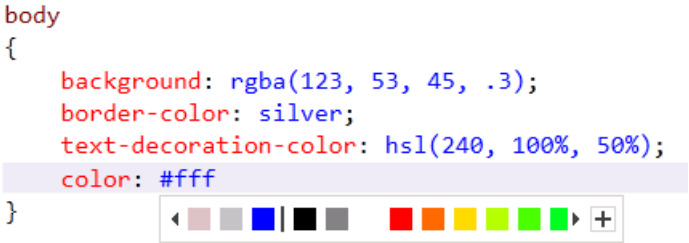
Commenting and uncommenting support

You can now comment and uncomment CSS rules using the same shortcuts that you use in the code editor (Ctrl+K,C to comment and Ctrl+K,U to uncomment).

Color picker

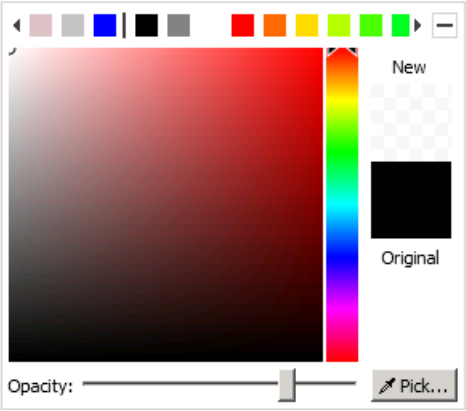
In previous versions of Visual Studio, IntelliSense for color-related attributes consisted of a drop-down list of named color values. That list has been replaced by a full-featured color picker.

When you enter a color value, the color picker is displayed automatically and presents a list of previously used colors followed by a default color palette. You can select a color using the mouse or the keyboard.



The list can be expanded into a complete color picker. The picker lets you control the alpha channel by automatically converting any color into RGBA when you move the opacity slider:

```
body
{
    background: rgba(123, 53, 45, .3);
    border-color: silver;
    text-decoration-color: hsl(240, 100%, 50%);
    color: rgba(255, 255, 255, 0.76)
}
```

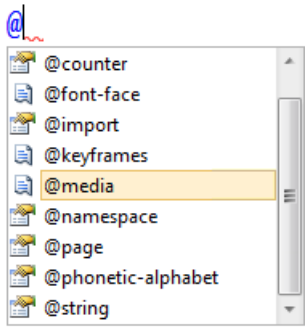


Snippets

Snippets in the CSS editor make it easier and faster to create cross-browser styles. Many CSS3 properties that require browser-specific settings have now been rolled into snippets.

```
body
{
    -ms-transition-timing-function: ease-in-out;
    -moz-transition-timing-function: ease-in-out;
    -o-transition-timing-function: ease-in-out;
    -webkit-transition-timing-function: ease-in-out;
    transition-timing-function: ease-in-out;
}
```

CSS snippets support advanced scenarios (like CSS3 media queries) by typing the at-symbol (@), which shows the IntelliSense list.



When you select @media value and press Tab, the CSS editor inserts the following snippet:

```
@media only screen and (max-width: 1024px) and (max-height: 768px) {

}
```

As with snippets for code, you can create your own CSS snippets.

Custom regions

Named code regions, which are already available in the code editor, are now available for CSS editing. This lets you easily group related style blocks.

```
/*#region Menu */

#menu {
    list-style-type: none;
}

    #menu li {
        margin: 0;
        float:left;
    }

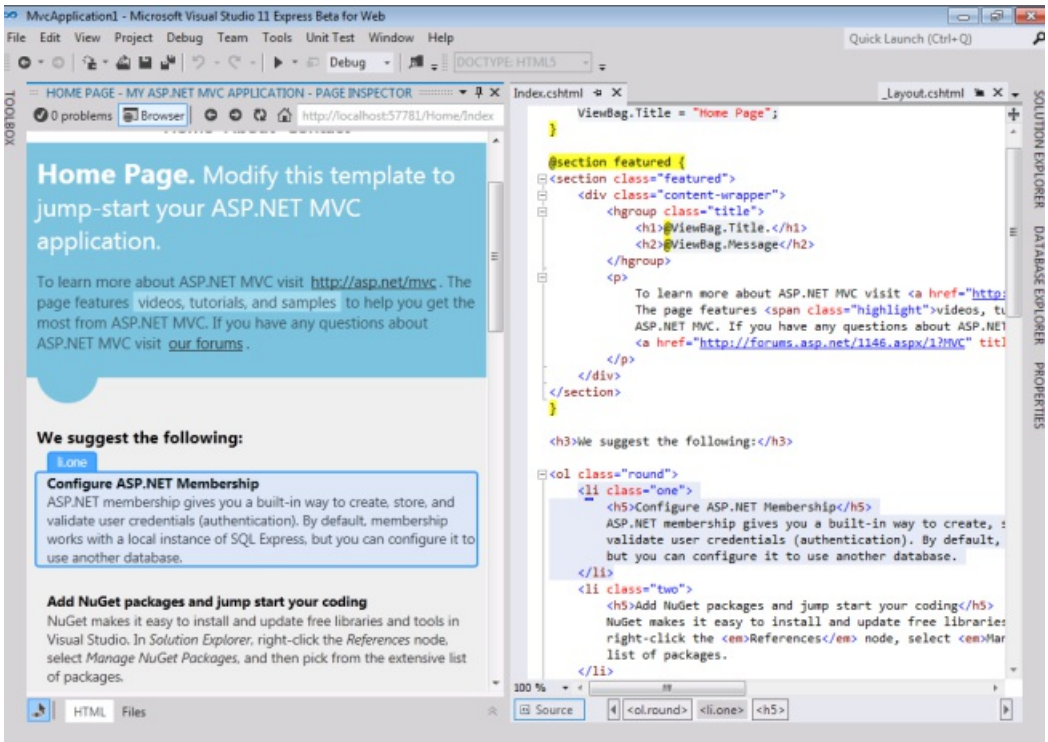
/*#endregion */
```

When a region is collapsed it displays the name of the region:

```
Menu
```

Page Inspector

Page Inspector is a tool that renders a web page (HTML, Web Forms, ASP.NET MVC, or Web Pages) in the Visual Studio IDE and lets you examine both the source code and the resulting output. For ASP.NET pages, Page Inspector lets you determine which server-side code has produced the HTML markup that is rendered to the browser.



For more information about Page Inspector, please see the following tutorials:

- Using Page Inspector in ASP.NET MVC
- Using Page Inspector in ASP.NET Web Forms

Publishing

Publish profiles

In Visual Studio 2010, publishing information for Web application projects is not stored in version control and is not designed for sharing with others. In Visual Studio 2012 Release Candidate, the format of the publish profile has been changed. It has been made a team artifact, and it is now easy to leverage from builds based on MSBuild. Build configuration information is in the Publish dialog box so that you can easily switch build configurations before publishing.

Publish profiles are stored in the PublishProfiles folder. The location of the folder depends on what programming language you are using:

- C#: Properties\PublishProfiles
- Visual Basic: My Project\PublishProfiles

Each profile is an MSBuild file. During publishing, this file is imported into the project's MSBuild file. In Visual Studio 2010, if you want to make changes to the publish or package process, you have to put your customizations in a file named **ProjectName**.wpp.targets. This is still supported, but you can now put your customizations in the publish profile itself. That way, the customizations will be used only for that profile.

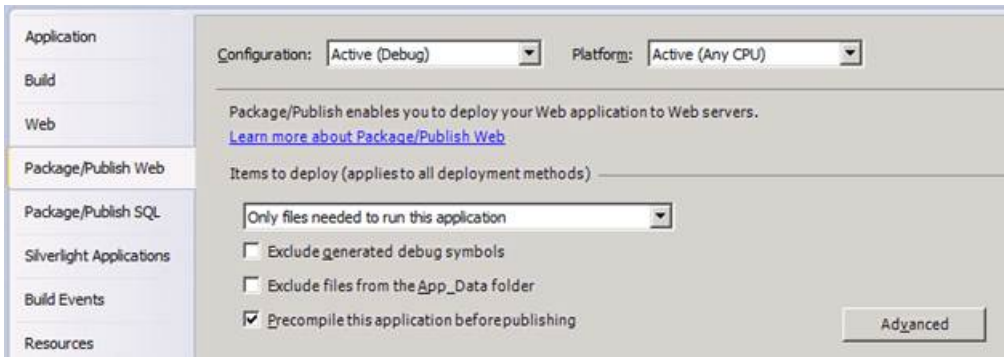
You can now also leverage publish profiles from MSBuild. To do so, use the following command when you build the project:

```
msbuild.exe project.csproj /t:WebPublish /p:PublishProfile=ProfileName
```

The project.csproj value is the path of the project, and ProfileName is the name of the profile to publish. Alternatively, instead of passing the profile name for the *PublishProfile* property, you can pass in the full path to the publish profile.

ASP.NET precompilation and merge

For Web application projects, Visual Studio 2012 Release Candidate adds an option on the Package/Publish Web properties page that lets you precompile and merge your site's content when you publish or package the project. To see these options, right-click the project in Solution Explorer, choose Properties, and then choose the Package/Publish Web property page. The following illustration shows the Precompile this application before publishing option.



When this option is selected, Visual Studio precompiles the application whenever you publish or package the web application. If you want to control how the site is precompiled or how assemblies are merged, click the Advanced button to configure those options.

IIS Express

The default web server for testing web projects in Visual Studio is now IIS Express. The Visual Studio Development Server is still an option for local web server during development, but IIS Express is now the recommended server. The experience of using IIS Express in Visual Studio 11 Beta is very similar to using it in Visual Studio 2010 SP1.

Disclaimer

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, email addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2012 Microsoft Corporation. All rights reserved.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

ASP.NET API reference

2/20/2020 • 2 minutes to read • Edit Online

- ASP.NET API Reference
- ASP.NET MVC API Reference
- ASP.NET Web API API Reference
- ASP.NET Web Pages API Reference
- ASP.NET SignalR API Reference
- ASP.NET Identity API Reference
- Microsoft OWIN Components API Reference